Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

به متفاوت ترین ارائه درس سیستم عامل خوش آمدید

هدف : ۱- چشم دل باز کن که جان بینی آنچه نادیدنی است آن بینی
از ۵ تا ۱۰۰ نحوه کار کامپیوتر
۲- لذت ببرید و عاشق شدن

مناسب برای :

۱- دانشجویان کلیه مقاطع کارشناسی رشته های مرتبط با کامپیوتر ← اگر می خواهید بفهمید توی کامپیوتر چه خبره
۲- افرادی که می خواهند بصورت حرفه ای و آکادمیک در رشته کامپیوتر ریسرچ و کار کنند
۳- " " علاقه مند هستند با جزئیات و در تمامی لایه ها برنامه کامپیوتر چگونه کار می کند
۴- اگر می خواهید برنامه نویس حرفه ای شوید

قوانین سوال پرسیدن :

۱- با توجه به تعداد بالای شرکت کننده و همین در راستای حداکثر کارایی به هیچ عنوان در حین تدریس من نه سوال بپرسید و نه چت کنید
زیرا حجم مطالب زیادی قرار است تدریس بشه و وقفه های بی مورد باعث می شود
اولا : کند شویم و کلاس بی مورد طولانی شود
دوما : تمرکز دوستان و آنهایی که بعدا فیلم ها را می بینند بهم بخوره
بنابراین این امکان وجود ندارد که وسط درس بحث را قطع کنیم

۲- اگر سوالی برایتون پیش می آید به احتمال بالا پاسخ شما در ادامه تدریس من وجود دارد و اگر صبر کنید پاسخ تان را دریافت می کنید،
بنابراین در زمانهای مختلف و شخص که از من مناسب است و کلیت یک مبحث را بیان کردم اعلام می کنم که اگر کسی سوالی داره بپرسه

۳- در هنگام پرسیدن سوال نقطه و نقطه در خصوص همان مطالبی که در حال بررسی آن هستیم سوال بپرسید، مطالبی که خارج
از چارچوب لحظه جاری باشد باعث گیج کردن دانشجویان و کم کردن throughput فیلم ها می خواهد شد که دوستان شما خواهند کرد

معجزه من برای این درس :

۱- جابجایی مطالب
۲- از بین بردن پیچیدگی ها برای شما
۳- تدریس پازل گونه
۴- تدریق لایه ای و چرایی
۵- تدریق نگاه بالا به پایین (perspective)
۶- عاشق کردن

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

قوانین استفاده از دوره :

۱- این فیلم‌ها و جزوات حاصل سال‌ها تلاش پیوسته و مطالعه است ، بنابراین هرگونه استفاده از آنها به جز از طریق خرید فیلم‌ها از سایت کافه تدریس حرام است و قطعا پیگرد قانونی نیز خواهد داشت

۲- قطعا کار نادرست منجر به موفقیت نخواهد

۳- به هیچ عنوان جزوات را در اختیار هیچ شخصی قرار ندهید که به هیچ عنوان بنده رافی نیستم ، جزوه فقط و فقط مختص کسانی است که فیلم‌ها را تهیه کرده‌اند

نحوه برگزاری این دوره :

۱- سعی من بر این است که جزوات را قبل از کلاس بنویسم که با توجه به حمایت من (دانوس D:) این کار زمان بسیاری زیادی از من می‌گیرد ، هدف من از اینکار این است که بتوانم ۱۰۰ ساعت داده را در حدود ۴۰ الی ۵۰ ساعت ارائه کنم که این کار نتایج زیر را در بر خواهد داشت

اول : زمان کلاس‌ها و فیلم‌ها کوتاه خواهد شد و دانشجویان در زمان معقول این درس را مطالعه و تمام می‌کنند چون در سر کلاس زمانمان برای نوشتن تلف نمی‌شود و ۱۰۰ ساعت داده زمان بهینه خواهد بود

دوما : مطالب بیشتری را می‌توانیم پوشش دهیم

سوما : مرور و جمع بندی برای شما بسیار راحت خواهد شد و در مرورها نیاز کمتری به مراجعه به فیلم‌ها خواهد داشت

* چهارما : هر ۱ جلسه ما حدا قل معادل ۲ الی ۳ جلسه در حالت عادی خواهد بود

نحوه استفاده از دوره :

۱- بعد از کلاس (همان شب یا صبح فردا) جزوه را در حد ۱ ساعت مرور کنید و قبل از هر جلسه جزوه جلسه قبل را مرور کنید

۲- به هیچ عنوان حاشیه نویسی نکنید و فقط جزوه را مرور کنید : دانوس = مرگ D: مطالب بیشتری نسبت به آنچه لازم است به اینه به شما خواهم گفت

انرژی که به من می‌دهید در انرژی من و کیفیت دوره مهم است.

کار بسیار سنگینی قرار است انجام شود و به کمک خدا اثر ماندگاری ایجاد شود در این مسیر انرژی شما یاری دهنده من خواهد بود

جمله‌ای به یادگار برای من و شما :

موفقیت خود صدا می‌کند ← پس بگذار موفقیت صدای تو باشد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

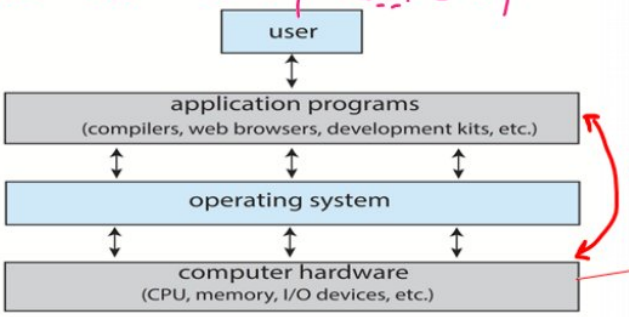
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

جهان کامپیوتر چگونه است؟

سیستم عامل چیست؟ سیستم عامل یک برنامه (سیستمی) است که مدیریت سیستم است. سیستم عامل یک لایه نرم افزاری است (پس نرم افزار است) و نزدیکترین لایه به سخت افزار است یعنی os (این لایه) بیشترین درک را نسبت به سخت افزار دارد.

* os یک لایه است بین برنامه های کاربردی و یوزرها و همین طور سخت افزار (منابع)، و os نقش مدیریتی دارد، یک نقش مدیریتی بر لایه بالا که قرار است (باید) برنامه های کاربردی را خوب مدیریت کند و یک نقش مدیریتی بر لایه پایین که قرار است (باید) منابع را خوب مدیریت بکند، به این ترتیب لایه میانی os که نقش مدیر را دارد هم از لایه پایین دهم از لایه بالا مراجعه کننده دارد



لایه پایین + لایه بالا : محیط os

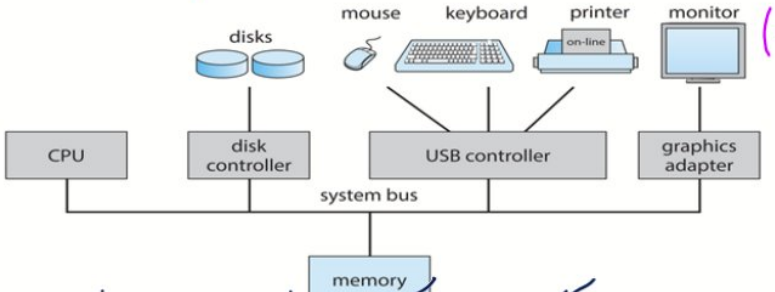
سخت افزار کامپیوتر چیست؟
به مجموعه ای از منابع (Recource) سخت افزار می گویند

همان طور که در شکل بالا مشاهده می شود os یک لایه واسط است که از پایین با سخت افزارهای سیستم در ارتباط است و از بالا با برنامه های کاربردی و برنامه های سیستمی و کاربرها در ارتباط است. سیستم عامل ۲ لایه بالا و پایین خود را مدیریت می کند که بعدا بیشتر توضیح می دهیم.

* os چیست؟ انتظاراتی که از یک os وجود دارد چیست؟ چرا به os نیاز داریم؟ وظایف و خط مشی های os چیست

۱- یک واسط است بین سخت افزار و برنامه های کاربردی و کاربران • یعنی چی؟
برنامه های کاربردی و کاربران اگر می خواهند از منابع استفاده بکنند باید از os این را درخواست کنند و os این ارتباط را برایشان برقرار کند

دلیل اش چیه؟
کار کردن با منابع یک سیستم کامپیوتری کار پیچیده ای است و کار کاربر عادی یا برنامه کاربردی نیست، os این کار را انجام می دهد تا دیگر کاربر یا برنامه کاربردی درگیر جزئیات و پیچیدگی های استفاده از سخت افزار نشود، اینکه چرا کار کردن با منابع (سخت افزار) پیچیده است را بعدا توضیح خواهم داد)



تعریف Driver : درایور نرم افزاری است که درون سیستم عامل قرار دارد (قرار می گیرد) و زبان کنترلر مربوط به خودش را می شناسد و می تواند تنظیمات مربوط به آن منبع را (Device) را انجام دهد (مقادیر رجیسترها را تغییر دهد)
چرا؟ چون Driver هر منبع را شرکت سازنده آن منبع ارائه کرده

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

مدیریت:
۲- مدیریت بهینه منابع (سخت افزار) (استفاده کارآمد منابع)
۳- مدیریت و پاسخ به نیازهای کاربران و برنامه های کاربردی
در نظر گرفتن و توجه اولویت و اهمیت بین آنها (برآوردن مطلوب نیاز کاربران)

os دو لایه را دارد مدیریت می کند:
مدیریت لایه زیرین os
مدیریت لایه بالایی

- وقتی os لایه سخت افزار را مدیریت می کند دارد مدیریت منابع را انجام می دهد و وقتی os دارد لایه برنامه های کاربردی را مدیریت می کند می گیم داره مدیریت کاربران و برنامه های کاربردی را انجام می دهد.

مدیریت بهینه منابع: از منابعی که در سیستم مان وجود دارد، مثل حافظه، دیسک، cpu و ... چگونه و به چه نحوی استفاده کنیم بطوریکه منابع خوب و بهینه استفاده شوند و هدر نروند

مدیریت نیازهای برنامه های کاربردی

مثلاً وقتی می خواهیم cpu را به برنامه های مختلف بدهیم از چه الگوریتم زمانبندی استفاده کنیم؟ گرسنگی؟ عدالت؟

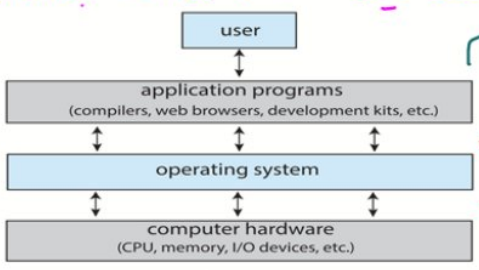
- گفتیم os یک لایه است بین ۱- برنامه های کاربردی و یوزرها و ۲- همین طور سخت افزار (منابع)، و os نقش مدیریتی دارد، یک نقش مدیریتی برای لایه بالا که قرارست (باید) برنامه های کاربردی را خوب مدیریت کند و یک نقش مدیریتی برای لایه پایین که قرارست (باید) منابع را خوب مدیریت بکند، به این ترتیب لایه میانی os که نقش مدیر را دارد هم از لایه پایین و هم از لایه بالا مراجعه کننده دارد

* پس ما ۲ نوع مراجعه کننده به os داریم: ۱- یکی از لایه software ۲- یکی از لایه Hardware

نکته: در سیستم عامل های مبتنی بر وقفه (Interrupt Driven) هر دو این مراجعه کننده ها از طریق وقفه مراجعه می کنند، بر همین اساس ما ۲ نوع وقفه داریم ۱- وقفه نرم افزاری ۲- وقفه سخت افزاری

توجه: در اثر رخداد وقفه چه وقفه نرم افزاری باشد و چه سخت افزاری یک ISR در کرنل سیستم عامل (Kernel mode) اجرا خواهد شد

نکته*) مشخص است که os فقط زمانهایی روی cpu اجرا می شود که کار مدیریتی وجود داشته باشد ← اگر اتفاق در لایه نرم افزار یا سخت افزار بیفتد که به واسطه وقفه به os خبر داده شود os فعال خواهد شد تا آن رخداد را مدیریت کند، با اجرا ISR مربوط به واکنش مدیریتی نشان می دهد (وقتی لایه بالا یا پایین os را با وقفه خبر کنند os فعال می شود چون os مدیر سیستم است با اجرا ISR به آن رخداد واکنش مدیریتی نشان خواهد داد

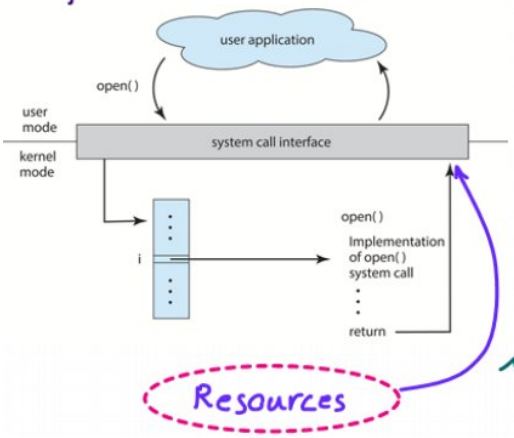


نکته در خصوص وقفه نرم افزاری: یا خود برنامه کاری دارد که خودش نمی تواند انجام دهد و فقط os می تواند آن کار را برایش انجام دهد و بنابراین برنامه برای اینکه کارش انجام شود خودش میاد و os را صدا میزند (فعال اش می کند) یا برنامه کار غیرمجاز انجام می دهد که بوسیله سخت افزارهایی

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

برای تشخیص کار غیرمجاز توی سیستم تعبیه کردیم (جاهای مختلف سیستم کار گذاشتیم) اون کار غیرمجاز شناسایی میشه و در اثر آن باز هم وقفه صادر می‌شود و باز کنترل میفته دست OS.



* تنها کاری که یک برنامه می‌تواند برای انجام کارهای خاص* انجام دهد این است که به OS درخواست دهد.

درخواست هم که نتیجه‌اش همیشه مثبت نیست :)

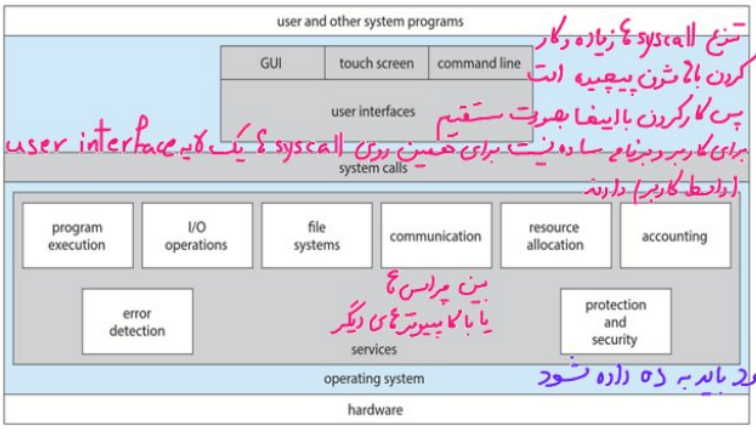
گفتیم اگر برنامه کاری را خودش نتواند انجام دهد OS را صدا می‌زند و آن کار را از او می‌خواهد

خب چه کارهایی را برنامه عادی خودش می‌تواند انجام دهد و چه کارهایی را خودش نمی‌تواند؟

برنامه می‌تواند دستورات عادی مانند ADD، AND، mul، sub، Div و ... را خودش انجام دهد اینها کار مدیریتی نیست

نمی‌تواند بصورت مستقیم مثلاً چیزی را روی صفحه نمایش نشان دهد یا چیزی را روی آن بنویسد یا صدایی را پخش کند و ...

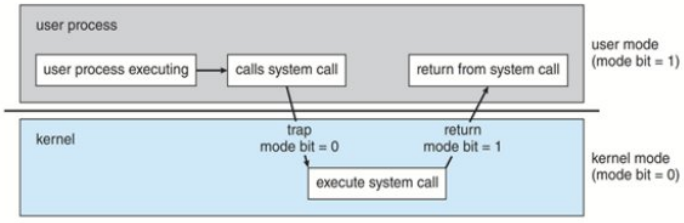
← نمی‌تواند و نباید هم بتواند چون اگر هر کسی بتوند که هرج و مرج میشه



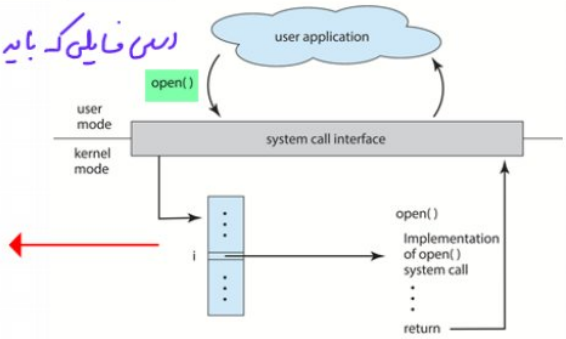
تنوع syscall ها زیاده و کار کردن با هاشون پیچیده است پس کار کردن با اینها بصورت مستقیم برای کاربر و برنامه ساده نیست برای همین روی syscall ها یک لایه user interface (واسط کاربر) دارند

بین پراسس ها یا با کامپیوترهای دیگر

اسمی فایلی که باید open بشود باید به OS داده شود



Transition from user to kernel mode.



برنامه باید شماره system call و پارامترهایی را که نیاز هست به آن system call داده بشود از user mode به Kernel mode ارسال شود

* ۳ روش برای دادن پارامتر از مد یوزر به مد کرنل داریم که فعلا کاری بهشون نداریم

سوال: کی (چه زمانی) برنامه OS (کد OS) روی cpu اجرا (Run) می‌شود؟ اگر کسی از لایه سخت افزار یا نرم افزار با آن کار داشته باشد

* اگر هر کسی کار مدیریتی داشته باشد بعد فقط OS بتوند آن را هندل کند چند تا خوبی دیگه هم داره، فعلا ۲ تا شو می‌گیم:

۱- توسعه پذیری (scalability) و سادگی (simplicity)

۲- حفاظت ← زندان گوانتانامو

* گفتیم که OS فقط زمانهایی روی cpu اجرا می‌شود که کار مدیریتی وجود داشته باشد: جمله نیاز به موشکافی داره

← این جمله یکجورایی به ما می‌فهمونه که در هر لحظه فقط یک برنامه، حالا چه برنامه کاربر چه برنامه OS می‌تواند روی cpu اجرا بشود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

(مشخصا در single core)

مثال: شرکت تاکسیرانی

مسافرها زنگ می‌زنند و درخواست ماشین می‌کنند ← سیر هر مسافر فرق می‌کند

cpu = راننده

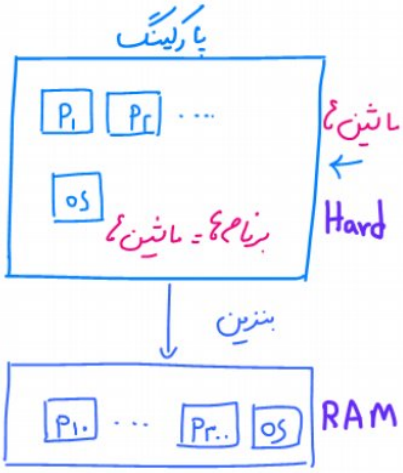
پارکینگ = Hard Disk

ماشین‌ها = برنامه‌ها (برنامه‌های کاربران + os)

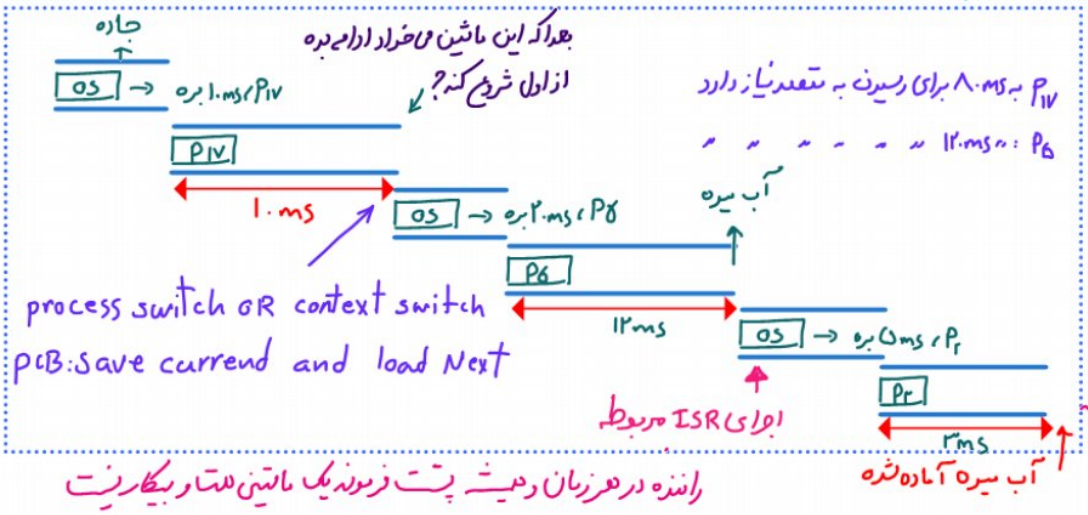
ماشین‌های توی پارکینگ بنزین ندارند و اگر قرار باشد بروند توی جاده اول باید بنزین بزنند و بروند توی اتاق آماده‌ها

* یک ماشین خاص داریم به اسم os که اون مشخص میکند ماشین بعدی که باید حرکت کند چی باشه، اگر ماشینی (مثلاً مسافرش) هم خدمتی می‌خواست (مثلاً آب میوه یا درخواست تعویض لاستیک) باید از ماشین خاص os بخواد که بره و براش بیاره

* اول اول هم راننده پشت ماشین os است.



وضعیت راننده در زمان‌های مختلف



راننده در هر زمان و همیشه پشت فرمونه یک ماشینی هست و بیکار نیست

* ۸۰ms مربوط به P17 که تمام شد باکش خالی میشه و باید بره توی پارکینگ

* هر فرآیند آنقدر در حافظه می‌ماند تا کارش تموم بشه (خاتمه عادی): با یک sys call (exit) به os میگه کارم تموم است

os به دلایل دیگری فرآیند را از بین می‌برد

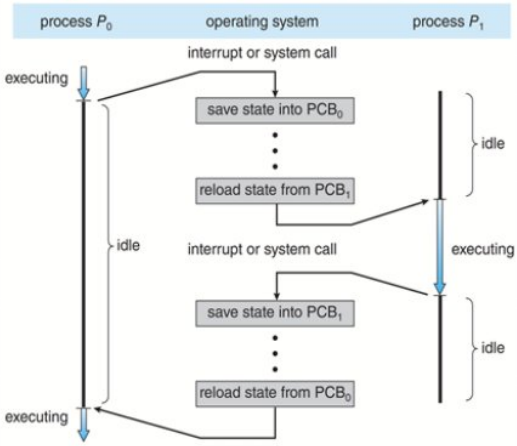
* صف ready ← scheduler ← os براساس یک الگوریتم زمانبندی مشخص * صف Block * تایمر

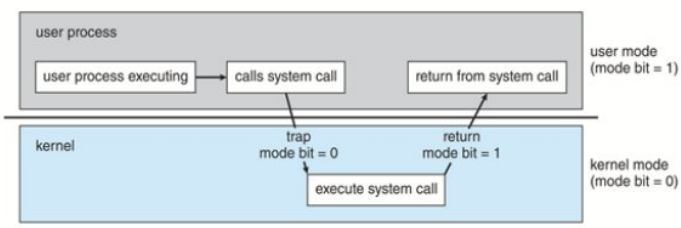
* سوال: توی مثال بالا هدف اصلی ما چیه؟ سرویس دادن خوب مسافرها

نحوه پرسش دیگر همین سوال: ما دوست داریم راننده بیشتر پشت فرمول ماشین‌های عادی باشه یا پشت فرمول os؟ ماشین‌های عادی

← هدف اصلی os سرویس دادن به کاربران برای اجرا برنامه‌ها است پس بهتر است os کمتر از cpu استفاده کند و بهتر است cpu بیشتر برنامه‌ها را اجرا کند، در عمل نیز به همین صورته و در حالت عادی cpu همیشه در حال اجرا برنامه‌های کاربردی است و تا کار مدیریتی نیاز نباشه (تا کسی از لایه نرم‌افزار یا لایه سخت‌افزار با os کار نداشته باشه) os روی cpu اجرا نمی‌شود

* زمان مفید





Transition from user to kernel mode.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

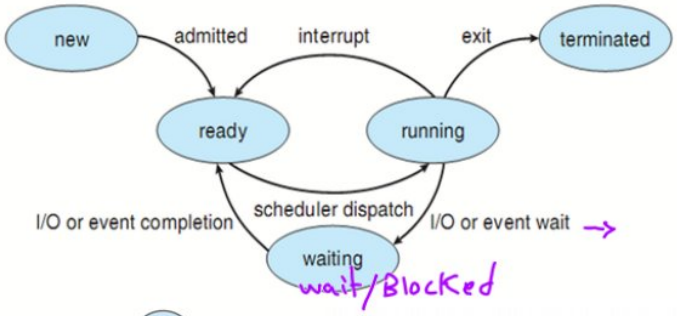
برنامه برای کار خاص ← فقط درخواست ← با توقف نرم افزاری (sys call) : یک دستورات خاص هستند)
← OS یک گوش نشسته ← که cpu بخش نگیره بیاد وسط

- اینکه اومدیم و Time sharing انجام دادیم باعث می‌شود به همه پراسس‌ها توجه کنیم و بنابراین زمان انتظار همه برنامه‌ها مناسب می‌شود و در این حالت OS داره به انتظارات برنامه‌ها توجه می‌کند ولی اگر از نگاه مدیریت منابع این switch کردن‌ها سربار دارند و بخشی از زمان cpu تلف این switch کردن می‌شود، اگر فقط بخواهیم پارامتر دوم را بهینه کنیم باید برنامه ۱ را تا آخر اجرا کنیم و بعد که تمام شد سراغ برنامه ۲ می‌رویم و بنابراین زمان انتظار برنامه دوم خیلی بالا می‌رفت ← نارضایتی کاربر

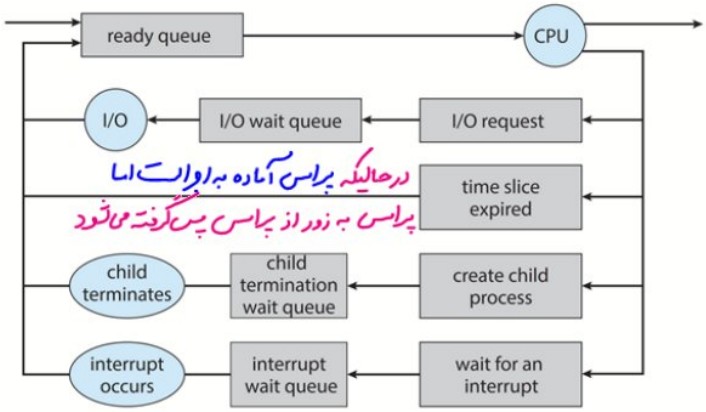
* مشخص است خیلی وقت‌ها اهداف OS می‌توانند در تضاد با هم قرار گیرند

OS چه زمانهایی اجرا می‌شود ← همیشه !!!

چرخه ساده‌ای از زندگی یک فرآیند در Multi programming : یک فرآیند می‌تواند در چه وضعیت‌هایی باشد

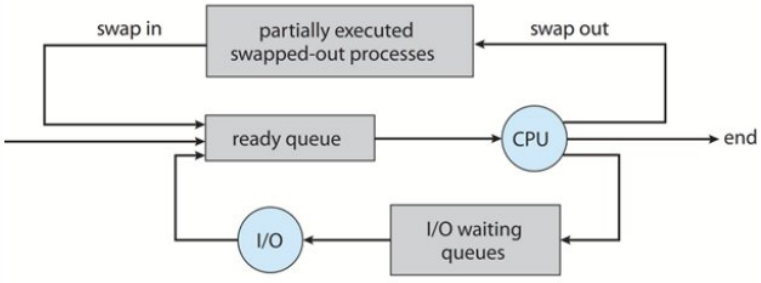


نیاز به خواندن یا نوشتن (Blocked)
منتظر پایان از فرآیند دیگر (wait)

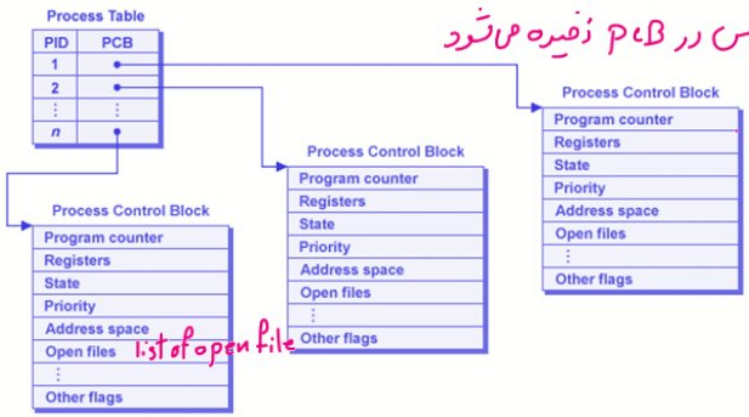


درحالیکه پراسس آماده به اجرا است اما پراسس به زور از پراسس پس گرفته می‌شود

Queueing-diagram representation of process scheduling.



Addition of medium-term scheduling to the queueing diagram.



PCB شناسنامه یک پراسس است که اطلاعات زیر به ازای هر پراسس در PCB ذخیره می‌شود

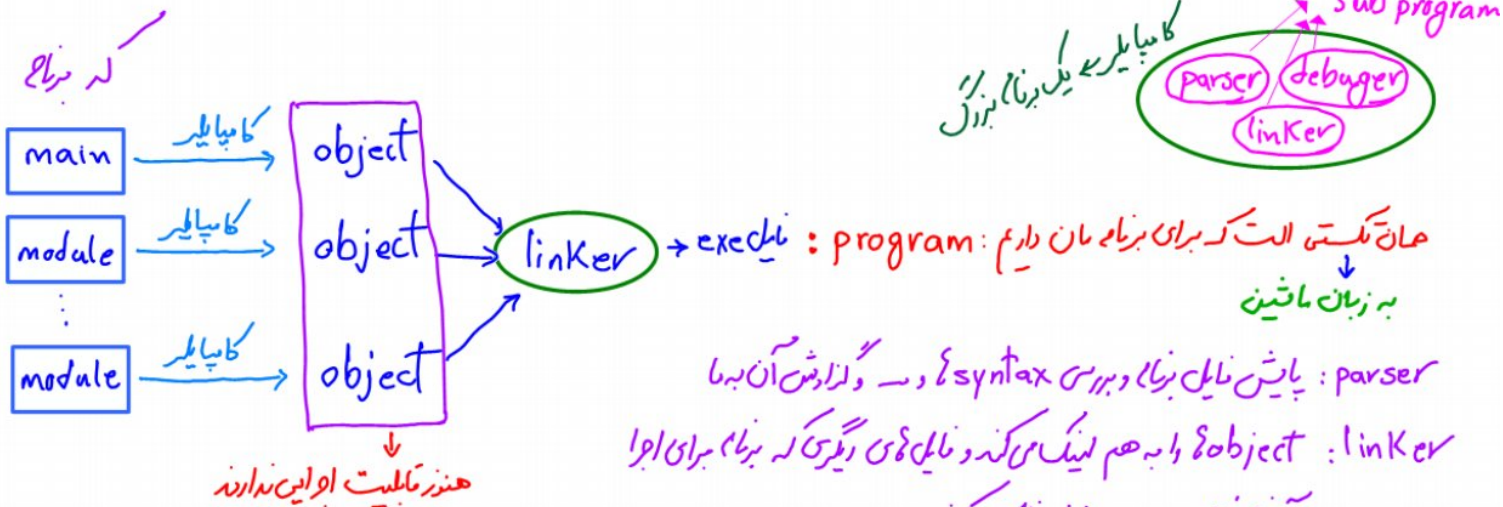
+ cpu-scheduling information
اولویت (process)، چقدر از سهم زمانی اش استفاده کرده
+ memory managment information
کدام یک از قسمت‌های حافظه توسط پراسس قابل دسترسی است
+ I/O state information
کدام وسیله I/O در اختیار این پراسس است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

* مقدمات : نکات بسیار مهم برای شروع که شاید نمی دانید !

نکته ۱: یک کد (exe = یک کد قابل اجرا = دستورات اجرایی) چگونه ایجاد می شود؟ کد برنامه → کامپایلر → لینکر → exe



همان تکستی است که برای برنامه مان داریم : program : فایل exe ← به زبان ماشین

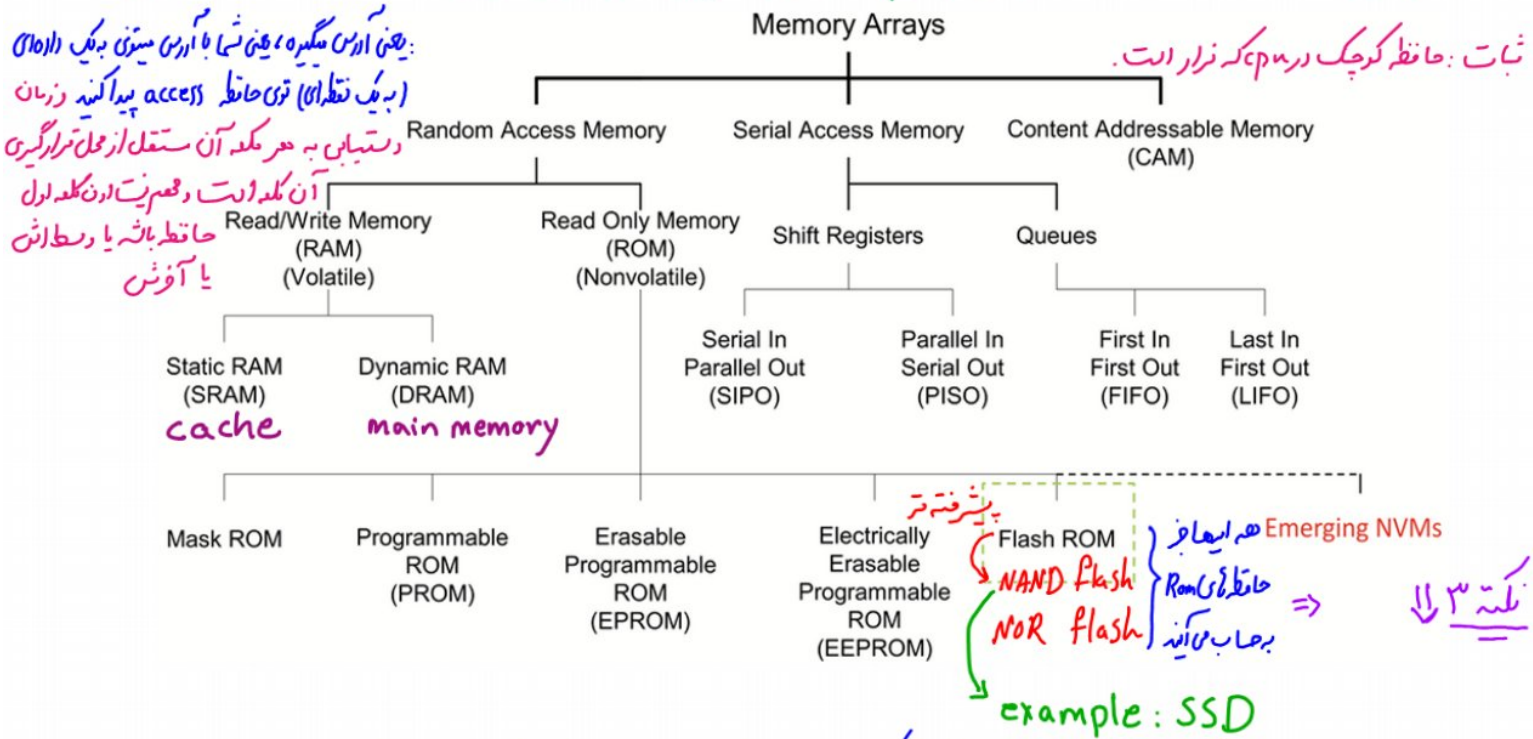
parser : پایش فایل برنامه و بررسی syntax ها و ... و گزارش آن به ما

linker : object ها را به هم لینک می کند و فایل های دیگری که برنامه برای اجرا به آنها نیاز مبرم دارد را ادغام می کند

نکته ۲ـ فایل اجرایی ایجاد شده روی هارد قرار می گیرد. چرا؟ چون حافظه هارد از نوع nonvolatile است، یعنی با قطع برق دیتا درون اش از بین نمی رود

* حافظه های اصلی از نوع volatile هستند و با قطع برق دیتاشون از بین میره و ما نمی توانیم فایل که های قابل اجرایمان را روی RAM (main memory = m.m) ال بذاریم که با هر بار قطع برق دیتایش از بین می رود (بعد از 1 ثانیه می پرد)

ثبات: حافظه کوچک در cpu که نرار است.



یعنی آدرس میگیره، یعنی شما با آدرس میتونی به یک داده ای (به یک نقطه ای) توی حافظه access پیدا کنید و زمان دستیابی به هر مکعه آن مستقل از محل قرارگیری آن کلمه است و مهم نیست اون کلمه اول حافظه باشه یا وسط اش یا آخرش

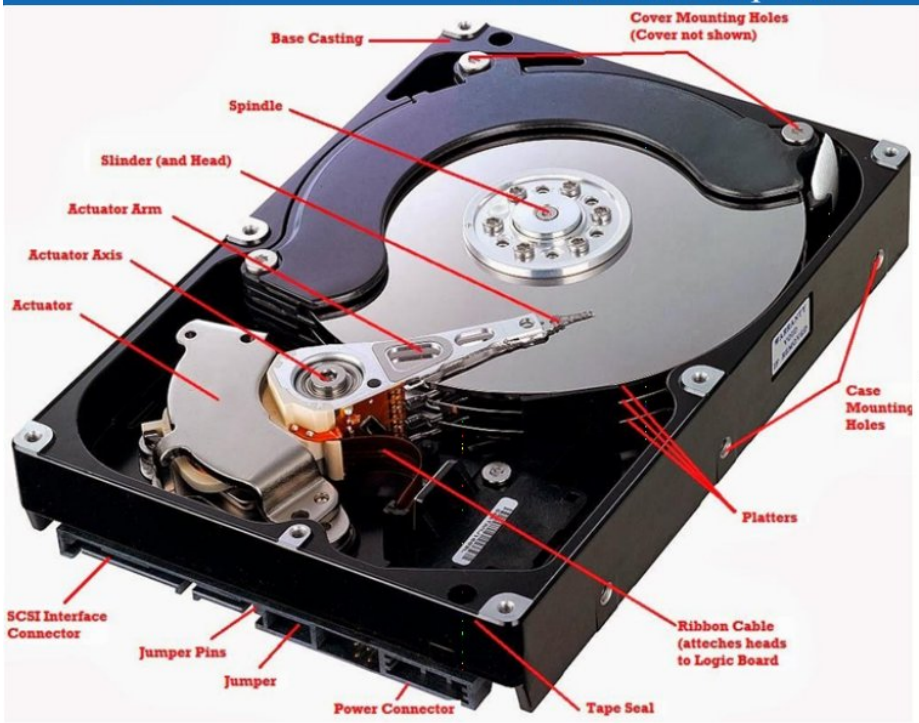
نکته ۳ ⇓

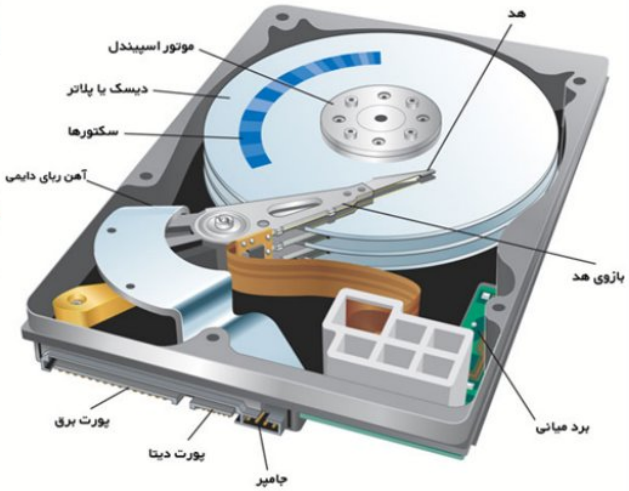
HDD : Hard Disk Drive : هارد یک حافظه مکانیکی است و اطلاعات روی سطح مغناطیسی دیسک ذخیره می شود، در هارد با چرخش پلاتر هد خواندن و نوشتن توسط یک بازو مکانیکی به داده مورد نظر دسترسی پیدا می کند و اطلاعات توسط هد و با بهره گیری از مغناطیس خوانده و نوشته می شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

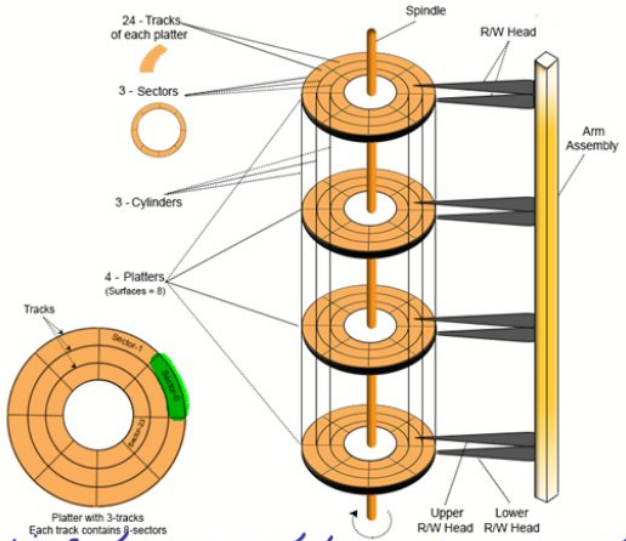
Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

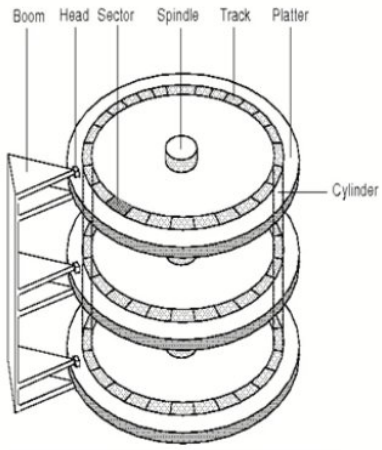
نمایی از هارد دیسک

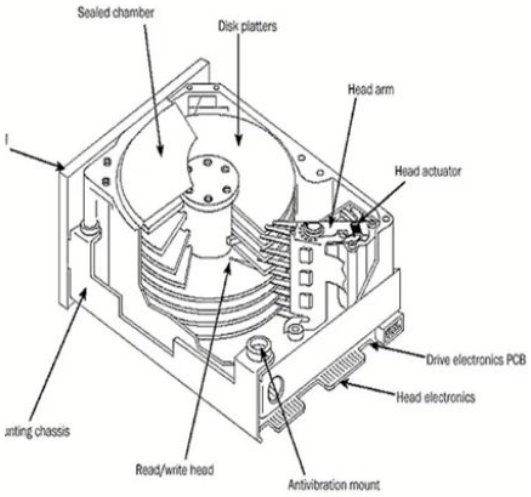




توجه: واحد خواندن و نوشتن از روی هارد سکتور است و کمتر از آن نمی‌توانیم بخوانیم، اگر بخواهیم توی هارد حداقل یک سکتور باید بخوانیم (بیشتر می‌توانیم)
حجم اطلاعاتی که در هر سکتور ذخیره می‌شود ثابت و ۵۱۲ بایت است. تعداد سکتورهای هر ترک هارد ثابت است، مثلاً بالای ۵۰۰ تا



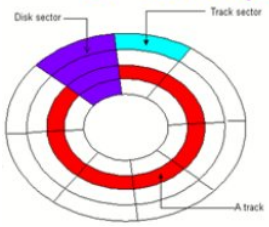




نکته۳) یک بلاک می‌تواند شامل چندین سکتور باشد مثلاً یک بلاک 8KB شامل ۱۶ سکتور است (یک بلاک می‌تواند یک کلاستری (خوشه) از سکتورها باشد)

نکته۴) آیا تعداد سکتورهای هر ترک ثابت است و ۳۳ تا است یا اگر از spindle که فاصله می‌گیریم سکتورها می‌توانند بیشتر شوند؟

نکته۵: توی تمام این Rom ها عملیات Read به مراتب سریعتر از عملیات write است اینها ذات شون بیشتر برای Read کردن تا write کردن البته با پیشرفت تکنولوژی مشکل که در عملیات write دارند داره آروم آروم حل میشه به همین خاطر هم ما امروزه مثلاً حافظه های SSD رو داریم که به جای Hard disk داره ازشون استفاده میشه که به مراتب از هارد که یک چیز مکانیکال است سریعتره، اگرچه باز read شون به مراتب سریعتر از رایت شون هست ولی رایت شون هم خیلی سریع شده



نکته۶ - هدف اصلی cpu اجرا برنامه‌های کاربر است. (برنامه کاربر نه برنامه os)

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاه

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته ۷: توجه کنید که cpu کاری جز execute, fetch دستورات بلد نیست و تا کامپیوتر روشن است cpu در یک حلقه همین کار را می‌کند پس به محض اینکه کامپیوتر را روشن می‌کنیم باید یک برنامه وجود داشته که cpu آن را fetch و execute کند.

نکته ۸: یک برنامه برای آنکه Run (اجرا) بشود باید در حافظه اصلی (بعداً که بحث آشنا شدیم متوجه کارکردش نیز خواهید شد) باشد چون پردازنده از حافظه اصلی دستورات را fetch می‌کند و در ثبات IR می‌گذارد و بعد دیکد و بعد اجرا می‌کند. چرا؟ چون اگر cpu بخواهد fetch را از روی هارد دیسک انجام دهد زمان اجرا برنامه‌ها به شدت زیاد میشه و عملاً این کارشدنی و عملیاتی نیست.

نکته ۹: پس متوجه شدیم که برنامه‌ای که قرار است اجرا شود باید بخشی از کد اجرایی آن که در آن لحظه نیاز است از هارد به داخل حافظه اصلی آورده شود.

نکته ۱۰: بعداً متوجه خواهید شد آن کسی که قسمت‌های مورد نیاز کد اجرایی برنامه را از هارد به RAM می‌آورد OS است.

نکته ۱۱: حافظه اصلی فرار است و با قطع برق اطلاعات آن از دست می‌رود، بنابراین در ابتدایی که کامپیوتر را روشن می‌کنیم حافظه اصلی خالی است. (حافظه اصلی = RAM ≠ هارد)

نکته ۱۲: OS نیز خود یک برنامه بزرگ است (برنامه سیستمی)، پس OS نیز در ابتدا روشن شدن کامپیوتر روی هارد قرار دارد و برای اجرا شدن نیاز دارد که بخش‌هایی از آن که در ابتدا نیاز است به روی RAM آورده شود، در ادامه توضیح خواهیم داد که OS چگونه در حافظه لود می‌شود. (یعنی پاسخ این پرسش را در ادامه می‌دهم)

OS برنامه‌ها را روی RAM لود می‌کند پس چه کسی خود OS را روی RAM لود می‌کند؟

هدیه من به شما : نحوه بالا آمدن سیستم عامل

concept:

cpu فقط بلده یک دستور رو از روی یک حافظه‌ای که زمان خواندنش به سرعت cpu بخوره و متناسب باشه برداره و اون رو Fetch, Decode و اجرا و هیچ کار دیگری بلد نیست، برای همین بود که گفتیم cpu نمی‌تونه با هارد کار کنه و گفتیم هر برنامه‌ای که بخواهد اجرا شود باید بخشی از آن که نیاز است روی RAM باشد، البته حافظه‌های دیگری هم داریم که سرعت‌شون خوبه و در جاهای مختلف برای ما کاربرد دارند مثل حافظه کش یا رجیسترهای (ثبات‌های) درون پردازنده

* گفتیم که OS نیز مانند برنامه‌های دیگر یک برنامه است و بنابراین OS نیز مانند همه برنامه‌ها روی هارد قرار دارد، البته پس از روشن شدن کامپیوتر به نحوی که در ادامه توضیح می‌دهیم قسمت‌هایی از آن که نیاز است وارد RAM خواهد شد.

وقتی برنامه‌های عادی می‌خواهند اجرا شوند OS بخشی از آنها که لازم است را از روی هارد به RAM منتقل می‌کند و پراسس مربوط به آن را ایجاد می‌کند ← پس یک برنامه (که همان OS باشد) دارد یک برنامه دیگر را از هارد برمیدارد و میذاره توی RAM

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

حالا سوال این است که وقتی کامپیوتر خاموش است RAM چون قرار است خالی است و OS روی هارد است حال وقتی شما کامپیوتر را روشن می‌کنید OS چطور وارد RAM می‌شود؟ چه کسی OS را وارد RAM می‌کند و چگونه؟

* از همان ایده بالا باید استفاده کنیم: اینجا هم نیاز به یک برنامه داریم که cpu با اجرای آن برنامه بیاید و OS را از هارد برداره و بذاره توی RAM

* خب این برنامه را کجا بذاریم؟ روی RAM و کش و رجیسترها پردازنده که نمی‌تونیم بذاریم.

چرا نمی‌تونیم؟ چون هم اینها فرارند و با قطع برق داده درون شون می‌پره

خب چیکار کنیم؟ یک حافظه می‌خوام که اولا: فرار نباشه (نیاز به مقداردهی اولیه نداره)

دوما: سرعت اش به cpu بخوره (مثلا به RAM باشه)

سوما: براحتی توسط دیوایس کامپیوتری آلوده نشود

یافتم: ROM

جمع بندی: کامپیوتر رو که روشن میکنی cpu گشنه هست و دنبال خوراکه، خوراکش هم دستوره خب اگر هرچی حافظه داریم روی سیستم فرار (volatile) باشه خب موقعی که روشن میشه دستوری وجود نداره که بخواد بره اجرا کنه، پس نیاز داریم یک حافظه بذاریم که اولا: فرار نباشه دوما: سرعتش بخوره به cpu سوما: بهتره فقط خواندنی باشه که کسی (کاربران) نتونه تغییرش بده و خرابش کنه

Bootstrap loader = Bootstrap program = Firmware

در حافظه فقط خواندنی (Rom یا EEPRom) روی مادربورد برنامه‌ای به این نام وجود دارد، هر تولیدکننده سخت افزار یک firmware روی Rom خود می‌ریزد (برنامه به زبان ماشین تبدیل شده و روی Rom ریخته شده)

* firmware (یک general term است): اکثرا به که درون Rom اشاره دارد، به محتوای Rom اصطلاحا firmware می‌گویند ولی برخی به ترکیب Rom (حافظه فقط خواندنی که به سختی قابل برنامه ریزی مجدد است) و که روی آن نیز firmware گفته می‌شود پس در تعریف دوم firmware یک تراشه روی برد با قابلیت برنامه ریزی مجدد است که Bootstrap روی آن است.

چرا بهش میگن firmware: چون از یک نظر شبیه نرم افزار است چون بدون دستکاری سخت افزار تغییر می‌کند ولی قابلیت برنامه ریزی مجدد دارد (مثل EEPRom) و از این نظر هم که نوشتن firmware نیاز به زبان توصیف سخت افزار دارد شبیه سخت افزار است و در واقع نه نرم افزار است و نه سخت افزار، برای همین اسم اش رو گذاشتن firmware.

نکته) IBM روی اولین کامپیوتری که ساخت که اسم اش XT بود اسم این برنامه را (firmware را) Bios گذاشته بود که Bios مخفف Basic input output system است.

وقتی کامپیوتر را روشن می‌کنیم بصورت سخت افزاری آدرس شروع firmware در pc قرار می‌گیرد و به اینصورت کامپیوتر می‌دانه که از کجا باید fetch و دیکد و execute کند، در واقع زمانیکه ما کامپیوتر را روشن می‌کنیم cpu شروع به اجرا دستورات برنامه Bootstrap program

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

که در خانه ی حافظه فقط خواندنی گفته شده قرار دارد می گذه، مشخص است که در اثر اجرا این برنامه یکسری اتفاقات می افتد که در ادامه توضیح می دهیم اما قبل از ادامه خوب است بدانید که firmware یک برنامه معمولی است که می راند و برایش هم فرقی ندارد os چه باشد و کامپیوتر را که روشن می کنیم شروع به اجرا برنامه ای که درونش گذاشته شده می کند

firmware چه کارهایی انجام می دهد

۱_ POST) power-on-self-test = خود آزمایی روشن شدن): وقتی کامپیوتر روشن می شود باید مطمئن شود که دستگاه های اساسی که بهشون نیاز هست که ما در ادامه امکان کار با سیستم را داشته باشیم و برنامه های دیگر توانایی اجرا روی سیستم را داشته باشند درست کار می کنند پس شروع میکند چک کردن دستگاه های خاص که حتما باید به سیستم متصل و بدون اشکال باشند

۱_ این کار بصورت خودکار و بدون دخالت کاربر انجام می شود

۲_ هم Device ها چک نمی شود و فقط اساسی ها چک میشه : mouse, monitor, Keyboard, RAM, HARD

۳_ اگر خطا رخ بده کامپیوتر بوق میزند

* پس این مرحله شد : اجرا عیب یابی (Run diagnostics) برای تعیین وضعیت دستگاه (state of the machine)

۲_ Basic I/O Initialization : برنامه firmware تمام جاهایی که لازم است از رجیسترهای cpu گرفته تا کنترل کننده های دستگاه و محتویات حافظه اصلی را مقدار دهی اولیه می کند

concept:

ما متوجه شدیم که وقتی دکمه روشن شدن کامپیوتر را می زنیم در ابتدا RAM خالی است و os روی هارد است پس بلافاصله os نمی تواند اجرا شود و باید طبق یکسری از مراحل os را به RAM منتقل کنیم و فرمان را بدهیم به دست os (بوت شدن سیستم) و os است که می تواند با فراخوانی درایورها با وسایل جانبی کار کند و آنها را راه اندازی کند، حال تصور کنید بالا آمدن سیستم زمان برانت (مثلا ۲۰s)

حالا یک کاربر کامپیوتر وقتی که کامپیوتر را روشن می کند انتظار دارد متوجه این امر شود و روی صفحه نمایش یک چیزی را مشاهده کند یا اگر از ماوس یا کیبورد استفاده کرد بتواند Reaction آن را ببیند، بنابراین در راستای user friendly بودن سیستم و در راستای برقراری حداقل تعامل با کاربر تا زمان بوت شدن os باید یک درایور حداقلی توسط firmware فعال شود و امکانات پایه مورد نیاز را فراهم کند

توضیحات اضافه و تکمیلی:

وقتی سیستم مان را روشن می کنیم کمی طول می کشد تا os مان Boot شود و بنابراین درایور منابع مختلف هم نمی توانند فعال شوند، ولی روی مانیتور یک چیزی می بینیم یا ماوس را که تکان می دهیم کرسر تکان می خورد یا برخی از دکمه های کیبورد کار می کند

گفتیم زبان کنترلرها را درایورهای آنها بلد بودند و بنابراین در این لحظه ما هنوز بلد نیستیم این کنترلرها را برنامه ریزی کنیم پس چگونه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

که ما روی صفحه نمایش داده می‌بینیم یا ماوس تکان می‌خورد یا کیبورد کار می‌کند یا نه.

- پس برای برقراری ارتباط حداقلی با کاربر قبل از بوت شدن OS باید یک حداقل I/O را فراهم کنیم، پس:

firmware میاد و یک initialization اولیه انجام میده که باعث میشه سیستم برخی از منابع را در شرایط پایه‌شان که از قبل استاندارد شده بشناسد.

مثلاً کارت گرافیک را در حالت استانداردی که تمامی کارت گرافیک‌ها پشتیبانی می‌کنند می‌شناسد

کیبورد I/O کلیدی را می‌شناسد، عملیات پایه ماوس را می‌شناسد و...

* ترجمه: در عمل می‌توانستیم firmware نداریم و از همان ابتدا بریم OS را لود کنیم و وقتی OS یا Vir آمد درایورها را فعال کنیم ولی اینجوری تا چندین ثانیه کاربر نمی‌توانست نه تصویری ببیند و نه زمانی از طریق ماوس و کیبورد به سیستم بدهد و در واقع در این حالت در ابتدای کار ابتدایی‌ترین روش‌های تعامل را نیز نداشتیم و user friendly بودن از بین می‌رفت

نکته۱) چون Rom قابل تغییر نیست فقط برنامه‌های استاتیک مثل برنامه Bootstrap را می‌توان در آن ذخیره کرد

نکته۲) درست است که EEPRom را می‌توان تغییر داد اما تحت شرایط خاص می‌توان این کار را کرد و نمی‌توان آن را بصورت مکرر تغییر داد و بنابراین EEPRom نیز شامل برنامه‌های استاتیک است.

نکته۳) تلفن‌های همراه برای ذخیره برخی از برنامه نصب شده خود در کارخانه از EEPRom استفاده می‌کنند

نکته۴) در برخی از سیستم‌ها مانند سربایل‌ها، تبلت‌ها و گیم کنسول‌ها کل سیستم عامل روی Rom ذخیره می‌شود، ذخیره OS روی Rom در موارد زیر می‌تواند مناسب باشد اما اگر بخواهید که Bootstrap را تغییر بدهید باید Rom را بصورت تحت اتزاری تغییر دهید

۱- سیستم عامل‌های کوچک (مثلاً سیستم عامل‌های برخی از embedded system‌ها)

۲- simple supporting Hardware

۳- سیستم‌هایی برای شرایط غیرعمل (خشن و نامعمول)

نکته۵) firmware یک برنامه کوچک نیست و شامل توابع و function‌های بسیاری است، این function‌ها کارهای زیادی انجام می‌دهند، حالا قبول دارید firmware یک برنامه است که روی Rom است، بنابراین در زمانهایی که کامپیوتر در وسط کارهایش است هروقت که OS بخواهد می‌تواند توابع درون firmware را فراخوانی کند و از آنها استفاده کند، جالب است بدانید که OS همیشه از توابع firmware استفاده می‌کند

نکته۶) یکی از مشکلات Rom (firmware) آن است که اجرا که روی آن از اجرای که روی RAM کندتر است.

نکته۷) برخی از سیستم‌ها OS شان را در Rom ذخیره می‌کنند اما برای اینکه اجرا دستورات سریع باشد OS را در RAM کپی می‌کنند.

نکته۸) " دیگر از " " " " " نسبتا گران است و بنابراین فقط مقدار کمی از آن در دسترس است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته ۹) فرآیند راه اندازی کامپیوتر (starting a computer) توسط لود کردن (بارگذاری) کرنل بعنوان بوت کردن سیستم شناخته می شود

نکته ۱۰) ممکن است جایی ببینید که به Rom، firmware می گویند، مثلاً می گویند Bootstrap loader در firmware ذخیره می شود

نکته ۱۱) سیستم عامل هایی که بزرگ هستند یا سیستم هایی که مرتبا تغییر می کنند OS در Hard ذخیره می شود و Bootstrap، OS را بار می کند

concept : سوال : توی سیستم عامل های بزرگ مثل windows، unix و mac OS x چرا کل OS را نیاریم توی Rom

۱ـ OS بزرگه و فضای کافی نداریم توی Rom : اینقدر بزرگه که حتی توی RAM کامل نمی یاریم اش ← در ضمن اصلا نیاز هم نیست

۲ـ سرعت ROM پایین است برای اجرا دستورات

۳ـ تغییر پذیری اش سخته

۴ـ user friendly نیست، flexibility رو میاره پایین

← این OS هایی که ما گفتیم OS هایی هستند که برای عامه مردم مورد استفاده قرار می گیرند، حالا کاربر باید خودش انتخاب کند که دوست داره با چه سیستم عاملی کار کند و بنابراین ما نمی توانیم بیایم روی Rom اش سیستم عامل ـ x رو نصب کنیم بگیم همینی که هست، میخوای بخواه نمی خوای نخواه (دندت نرم)، بنابراین سیستم کامپیوتری ما باید به نحوی باشد که کاربران بتوانند

۱ـ هر OS ای که دوست دارند نصب کنند

۲ـ حتی بتوانند چندین OS نصب کنند و برحسب نیاز هر زمان که خواستند از هر کدام استفاده کنند

در یک کلام سیستم باید این انعطاف را داشته باشد به همین دلیل کاربرها میان و OS های خودشان را روی Hard نصب می کنند و بعد یک Bootstrap program که یک برنامه عمومی است و نمی داند OS چیست و برایش فرقی ندارد شروع می کند به بالا آوردن OS ای که ما انتخاب کردیم (اگر یک OS باشد که اتومات همان را بالا می آورد، اگر چند OS باشد یک زمان محدودی به ما می دهد که انتخاب کنیم چه OS ای را می خوایم که در صورتیکه در آن زمان انتخاب نکنیم اولی بصورت پیش فرض بالا می آید)

ادامه وظایف Bootstrap program : ۲ مورد را گفتیم حالا مورد سوم را می خواهیم بررسی کنیم :

۳ـ امکان تغییر اطلاعات درون حافظه cmos : Bootstrap برنامه ای است که به ما اجازه میده یکسری اطلاعات را در cmos ذخیره کنیم ما برای مواردی همچون ذخیره تاریخ و ساعت سیستم، password، مشخص کردن ترتیب رسانه هایی که باید OS از روی آنها به حافظه آورده شود، سیاست های خاموش کردن اتوماتیک مانیتور و هارد برای مدیریت مصرف انرژی و ... نیاز به یک حافظه ای داریم که

اولاً خواندنی و نوشتنی باشد

دوماً Non volatie (غیر فرار) باشد

* برای این منظور حافظه ای داشتیم به نام حافظه cmos (complementary Metal oxide semiconductor) که یک

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

حافظه کم مصرف است (یک حافظه SRAM بود که بهش CMOS RAM هم می‌گفتند) ظرفیت اش همگی از ۶۴ تا ۲۵۶ بایت بود) که توسط باتری به نام باتری CMOS تغذیه می‌شه که همگی عمری حدود ۳ سال (یا کمتر یا بیشتر مثلاً تا ۱۰ سال) داشت و قابل شارژ نبود.

ویندوز + R → msinfo32 → BIOS Version/Date
Last Bios time

CMOS RAM

CR2032

نکته) CMOS RAM در مادربوردهای امروزی همگی یک تراشه مستقل نیست و با تمت های دیگر یکپارچه شدند

نکته) در مادربوردهای امروزی از NVRAM که جز Flash Rom هاست استفاده می‌شود

از آن رسانه به حافظه اصلی

نکته) در CMOS ترتیب نگاه کردن به Bootable media را مشخص می‌کنیم و مشخص می‌کنیم که Bootstrap برای لود کردن OS به ترتیب سراغ کجاها برود، مثلاً می‌گیم اول برو سراغ فلش درایو بعد برو سراغ CD، بعد برو سراغ فلش و ... (با هر بار روشن کردن کامپیوتر)

Bootable media : مدیایی (رسانه‌ای) که نسخه قابل اجرا OS روی آن نصب شده است و به کامپیوتر متصل است

نکته) Hard، CD، فلش، هارد اکسترنال از جمله رسانه‌هایی هستند که می‌توان هر کدام را به توانایی load کردن (بوت کردن) OS مجهز کرد.

ترجمه) به عبارت نسخه قابل اجرا توجه کنید : ما باید یکبار OS را یکجا نصب کنیم (روی هارد، CD و ...) که همگی روی هارد نصب می‌کنیم، خب در اثر نصب OS که یک برنامه است مثل نصب هر برنامه دیگری یکسری کارها انجام میشه که در نهایت منتج میشه به اینکه روی هارد یک نسخه قابل اجرا خواهیم داشت

سوال) موقع نصب یک برنامه چه کارهایی انجام میشه؟ مثلاً مشخص کردن مقدار اولیه برخی از ساختمان داده‌هایی که استفاده می‌کند، ایجاد فایل‌هایی که لازم دارد، تخصیص فضا در هارد به این برنامه که بتواند دیتاهایی که مخابره می‌کند در آن نگهداری کند و ...

سوال) چرا OS همگی روی هارد نصب میشه؟ همیشه به کامپیوتر متصل است، نسبتا سریع است

۱- لود کردن مقداری از OS درون RAM از یک Bootable media به ترتیبی که ما برایش مشخص کردیم (می‌ره سراغ اولین حافظه (مدیا) Bootable ای که ما برایش مشخص کردیم و اگر آن مدیا فعال باشه بخش کوچکی از OS را از آن برمی‌دارد و آن بخش را در حافظه اصلی قرار می‌دهد و اگر آن رسانه فعال نباشه سراغ رسانه بعدی مشخص شده توسط ما می‌رود)

۶- در نهایت PC را با آدرس ابتدا قرارگیری Boot sector در RAM پر می‌کند. ← بعد از این دیگه کار Bootstrap تمام است (البته فعلا)

چون تقریبا همیشه OS از درون هارد به درون RAM لود می‌شود در خصوص لود شدن از طریق هارد صحبت می‌کنیم.

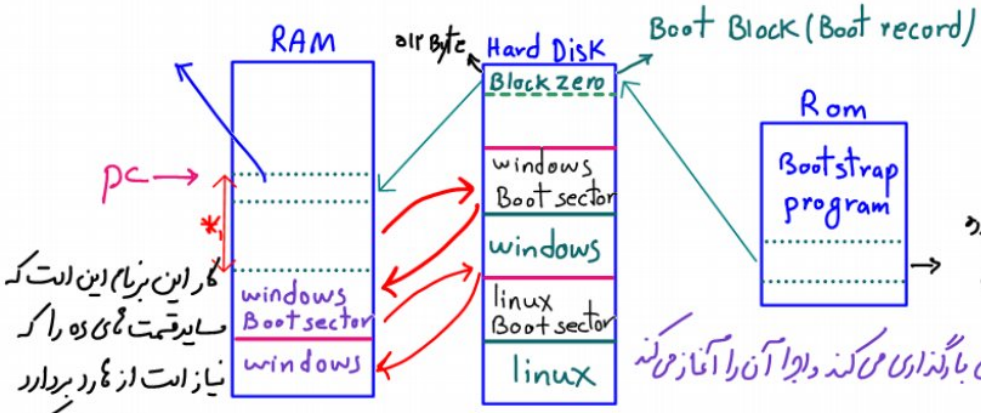
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

مراحل بوت شدن os

اطلاعاتی از جمله اینکه هارد چقدر است، چند تا ترک داره، چند تا بلاک داره، حجم هر بلاک چقدره و آدرس ابتدای لیست بلاک های خالی

1. post
2. Basic I/O Initialization
3. set and change cmos
4. Boot کردن os به RAM
5. پر کردن pc با آدرس نالب



سیره و از ترک یه، سکتوره هارد و ۵۱۲ بایت را برمیداره و میذاره توی حافظه اصلی (نمی دونه اطلاعات درون این سکتور چیه)

در واقع هسته os را از یک مکان مشخص در حافظه اصلی بارگذاری می کند و اجرا آن را آغاز می کند

کار این برنامه این است که سایر قسمت های os را که نیاز است از هارد بردارد و وارد حافظه اصلی می کند و اجرا os شروع می شود

* ترجمه کنید که برنامه firmware برای همه os ها یکسان است ولی Boot sector جزئی از os است و از یک os به os دیگر فرق می کند

* حجم این برنامه برای os های مختلف متفاوت است، مثلاً در ویندوز خیلی بزرگتر است یا در unix یک کیلو بایت است ولی firmware چون نمی داند os چیست و بنابراین نمی داند حجم این برنامه چقدر است بنابراین ۵۱۲ بایت ابتدایی را می آورد حالا اگر این برنامه بزرگتر باشد خود همین ۵۱۲ بایت سیره و بقیه برنامه را از هارد وارد RAM می کند

* در pc ها و لپ تاپ ها معمولاً برای بالا آمدن os یک فرآیند ۲ مرحله ای داریم که یک Bootstrap loader ساده یک boot program پیچیده را از روی هارد (دیسک) وارد RAM می کند و این برنامه است که هسته os را بارگذاری می کند

* برخی به برنامه توی Rom میگن BIOS و به اون برنامه‌ای که توی RAM اومده و os را بالا میاره میگن Boot strap program

* اگر بیش از یک os روی سیستم نصب شده باشد به برنامه اولی که روی RAM آورده می شود system chooser می گویند و این برنامه است که با توجه به انتخاب کاربر می رود و Boot strap مربوط به os ای که انتخاب کرده را به حافظه اصلی می آورد و ادامه اجرا برنامه را به آن واگذار می کند (ترجمه کنید که در اثر نصب os دوم روی سیستم قطعه که اول دچار تغییراتی خواهد شد)

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

فرآیند

چرا به فرآیند نیاز داریم؟

برنامه یک کدی است که باطی کردن مراحلی به کد زبان ماشین تبدیل شده و روی هارد قرار گرفته، برای اجرا هر برنامه‌ای داشتن خطوط دستورات آن و داده‌ها روی برنامه کافی نیست، در واقع برنامه روی هارد در حالت passive قرار دارد.

یک برنامه برای آنکه اجرا شود نیاز است تا از روی آن برنامه فرآیند بسازیم ← پس فرآیند از روی برنامه ایجاد می‌شود و تا فرآیندی برای آن برنامه ایجاد نشود آن برنامه قابلیت اجرا شدن روی cpu را نخواهد داشت ← فرآیند برنامه را اجرا می‌کند

گفتیم که فرآیندها (و ترده‌ها) تنها موجودات Active کامپیوتر هستند و هرچیز دیگری از جمله cpu، حافظه، کش و... ابزارهایی هستند که فرآیندها برای انجام دادن کارهایشان از آنها استفاده می‌کنند و بنابراین می‌گوییم هر چیزی به جز فرآیند passive است

نکته) هر برنامه برای اجرا به حداقل یک فرآیند نیاز دارد، شما وقتی در مرورگر یک تب را باز می‌کنید یک فرآیند جدید ایجاد می‌شود

ترجمه) برای ایجاد پراسس یک روالی (system call) باید فراخوانی شود

همان طور که اشاره کردیم

۱) وقتی روی یک برنامه دابل کلیک کنید یک پراسس ایجاد می‌شود (= صدور فرمان اجرا برنامه به هر طریق)

نکته) با دابل کلیک هم در واقع در یک گرانول همان system call تولید پراسس صدا زده می‌شود

۲) توجه کنید که os نیز یک برنامه است و بنابراین برای اجرا شدن نیاز به پراسس دارد، بنابراین در همان ابتدایی که کامپیوتر را روشن می‌کنیم و قرار است os کنترل را بدست گیرد نیاز است پراسس‌هایی ساخته شوند

* تا کامپیوتر را روشن می‌کنید فرآیند بالا آوردن os بعنوان اولین فرآیند ایجاد می‌شود و فرآیندهایی از os که نیاز است را به وجود می‌آورد و پس از بین می‌رود، از این به بعد os است که با درخواست اجرا برنامه توسط کاربران فرآیندهای جدید را می‌سازد و مدیریت می‌کند

* ما در این درس در خصوص سیستم عامل‌های مبتنی بر فرآیند (process based) مثل unix یا مبتنی بر نخ (Thread base) مثل windows صحبت می‌کنیم و در خصوص os‌های دیگر صحبتی نمی‌کنیم ← مگه چیز دیگه هم داریم؟ بله مثلاً Agent ها!

سیستم عامل مبتنی بر فرآیند: os ای که موجود زنده و فعالش فرآیند باشد، بطور دقیق‌تر os ای را مبتنی بر فرآیند می‌گوییم که کرنل آن از تعدادی فرآیند تشکیل شده باشد و برنامه‌ها توسط فرآیندها اجرا شوند ← فرآیندها برنامه‌ها را با استفاده از os و Hardware اجرا کنند.

۳) توجه کنید که گفتیم یک برنامه ممکن است برای آنکه اجرا شود پراسس‌های زیادی داشته باشد (مثل باز کردن تب در edge) بنابراین در داخل کد یک برنامه می‌توانیم آن تابعی که (system call) که پراسس ایجاد می‌کند را صدا بزنیم

نکته) در ویندوز برای تولید پراسس از system call = create process و در یونیکس از سیستم کال fork استفاده می‌شود

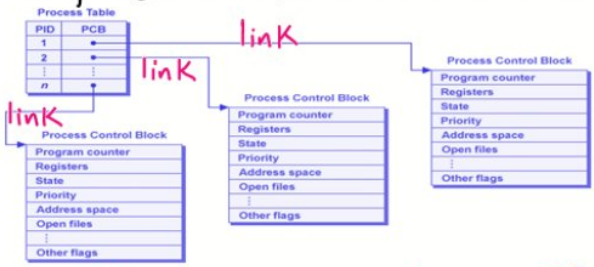
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

مراحل ایجاد یک فرآیند

- process table : پراسس‌های فعالی که در حال حاضر در سیستم هستند را نشان می‌دهد، os فقط به پراسس‌ها درون pT خدمات ارائه می‌دهد، توجه کنید که pT در حافظه اصلی است.



os وقتی می‌خواهد یک پراسس درست کند کارهای زیر را انجام می‌دهد
(شلاً بعد از دابل کلیک روی فایل اجرایی به ترتیب مراحل زیر انجام می‌شود)

توجه) چرا نیاز داریم اطلاعات کاملی از هر یک از فرآیندها نگه داریم؟ برای مدیریت فرآیندها توسط os با اعمال قوانین روی آنها
نکته) به مجموعه همه اطلاعات لازم از فرآیندها نگه داشته می‌شود process attributes یا صفات فرآیند می‌گویند.

همان طور که مشخص است برای متمایز کردن فرآیندها باید به آنها یک ID نسبت دهیم تا بتوانیم بوسیله ID به فرآیند خدمت دهیم و با فرآیند کار کنیم، پس یک فضایی باید برای ذخیره ID های فرآیندها در pT داشته باشیم، خب فرض کنید این فیلد ۱۰ بیتی باشد پس حداکثر در یک لحظه $2^{10}=1024$ فرآیند می‌تواند در سیستم حضور داشته باشد که برخی ready، برخی Block و... ویکی در حال running است.

۱- بررسی اینکه به سقف تعداد فرآیندهای موجود در سیستم رسیدیم یا خیر.
۲- اگر به سقف نامبرده نرسیده، می‌رویم و در pT آن پراسس را ثبت می‌کنیم و یک unique ID نیز به آن پراسس اختصاص می‌دهد.
۳- برای آن پراسس یک فضایی در حافظه اصلی بنام pcB اختصاص می‌دهیم که اطلاعات مربوط به آن پراسس را در pcB اش قرار می‌دهیم، اطلاعاتی نظیر اولویت اش، تا حالا چه فایل‌هایی را باز کرده، چقدر و از چه I/O ای استفاده کرده، محتویات ثبات‌های pc، AR، DR، psw و... ثبات‌های عمومی

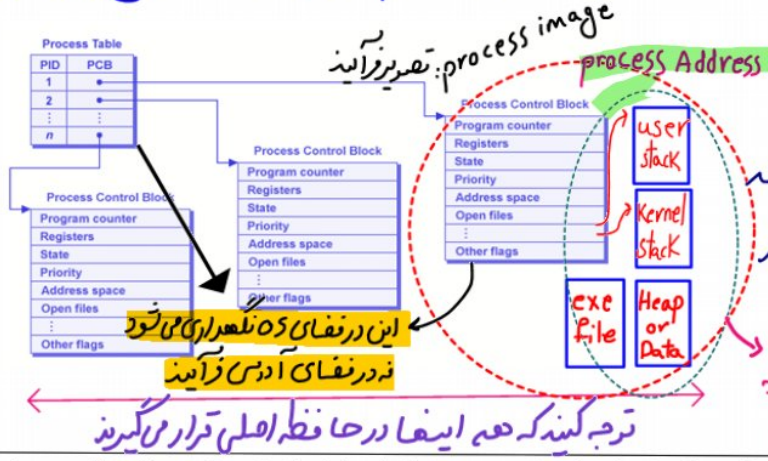
توجه) مشخصات چون فرآیند یک موجود Active است برخی از اطلاعات pcB یک فرآیند به تدریج در pcB اش قرار می‌گیرد
(شلاً آدرس به ابتدای پشته user و همین طور پشته Kernel)

۴- ۲ تا پشته برای آن فرآیند در حافظه اصلی در نظر گرفته می‌شود

۱- User Area stack : برنامه به یک پشته نیاز دارد برای ذخیره‌ای روال‌های خودش
۲- Kernel " " : systemcall هایی که برنامه ما صدا می‌زند ممکن است برای انجام کارهایشان روالها و توابع دیگری از کرنل را صدا بزنند.

۵- پراسس چون یک موجود فعال است نیاز به یک فضایی دارد که بتواند ساختارهای پویایش را مانند لیست پیوندی در آن قرار دهد بنابراین os یک فضای اولیه برای این منظور برای پراسس در نظر می‌گیرد که اصطلاحاً به آن Data یا Heap می‌گویند

* اگر این فضا کم بیاید برای پراسس در صورت وجود فضا با توجه به پارامترهایی به آن پراسس فضای بیشتری ممکن است بدهد



سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۶- یک فضا دیگر نیز در حافظه اصلی در نظر می‌گیرد و فایل exe برنامه را درون آن قرار می‌دهد

۷- pc که یکی از فیلدهای pcB است را با آدرس شروع برنامه پر می‌کند

توجه) توجه کنید که این pc توی ثبات pc مربوط به پردازنده load شده و هر وقت که برای اجرا انتخاب شود load می‌شود

۸- state این فرآیند که یکی دیگر از فیلدهای pcB است را ready می‌کند

نکته) در کامپیوتر فقط یک process table وجود دارد که هر ردیف آن به یک پراسس متعلق است.

نکته) PT و pcB جزئی از فضای os در حافظه اصلی است نه فضای فرآیند.

نکته) به ۲ صورت می‌توان PT و pcB را پیاده سازی کرد

۱- فقط یک PT داشته باشیم و در هر ردیفش هم اطلاعات pcB را قرار بدهیم ← PT بزرگ نیاز است

۲- یک PT داشته باشیم که هر سطر آن به یک pcB اشاره کند

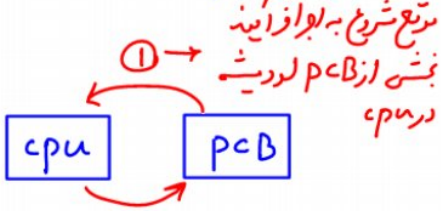
نکته) معمولاً PT و pcB بصورت جدا پیاده سازی می‌شوند و دلیل این امر آن است که در این حالت PT یک حافظه ایستا و pcB هر فرآیند یک حافظه پویا است و با از بین رفتن یک فرآیند می‌توانیم، فضایی که به pcB آن فرآیند داریم را آزاد کنیم و pcB آن را کاملاً حذف کنیم. حال اگر هم اطلاعات pcB را در PT نگهداری کنیم پس از اتمام فرآیند از سطر آزاد شده برای فرآیند جدید دیگری می‌توان استفاده کرد.

نکته) process attributes یا صفات فرآیندها در سه دسته قرار می‌گیرند

Metadata (فرا داده یا داده نما) (برخی شاید بهش ابر داده یا فوق داده بگویند)
1. process identifications information
2. process state information
3. process controls information

سوال- چرا به این اطلاعات Metadata می‌گویند؟ چون این داده‌ها، داده‌هایی هستند که برای تشریح برنامه‌ها و داده‌های متعلق به آنها استفاده می‌شوند (داده‌هایی هستند که جزئیات یک داده دیگر را تشریح می‌کنند) و این داده‌ها را فرآیندها پردازش نمی‌کنند و os است که برای مدیریت فرآیندها این داده‌ها را نگهداری می‌کند.

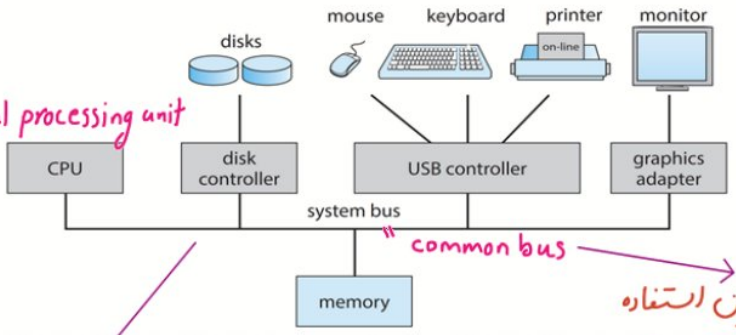
سوال- pcB کی آپدیت می‌شود؟ زمانیکه بخواهیم cpu را از فرآیندی که روی cpu در حال اجرا است بگیریم ← یعنی وقتی فرآیند از حالت running خارج می‌شود



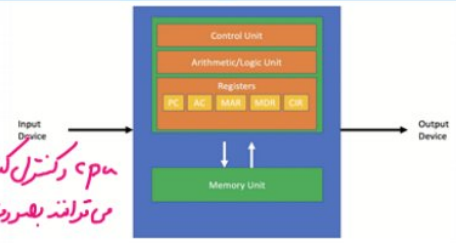
ترجمه) شاید پیاده سازی به نحوی باشد که در زمانهای دیگری نیز اطلاعات فرآیند در حال اجرا در pcB ذخیره شود) بصورت پریودیک

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



A typical PC computer system.



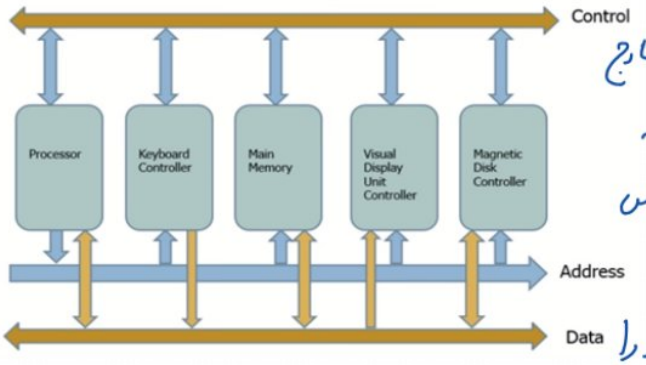
cpu و کنترل کننده های دستگاه های I/O می توانند بصورت موازی اجرا شوند

این باس مشترک دسترسی همه به حافظه مشترک را فراهم کرده

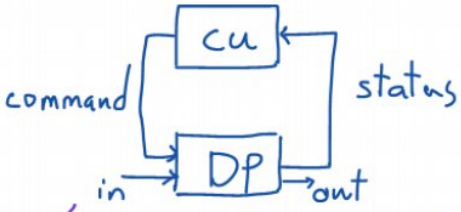
* واژه مشترک از اینجا آمده که همه Device ها می توانند از این استفاده کنند ولی توجه کنید که همزمان نمیشه از این استفاده کرد و conflict پیش میاد، در هر لحظه فقط 1 نفر میتوند از باس برای انتقال (ارسال یا دریافت) استفاده کند

خب اینکه خیلی مسخره شد! چرا؟ فکر کن یک اتوبان بذاری که چند تا شهر رو بهم وصل کند بعد بگی در هر لحظه فقط یکی میتوند توی جاده باشه و تا وقتی هم اون به مقصد نرسه کسی حق نداره بره توی اون جاده، عجب!!!

جواب من: یک چیپ که الان خیلی چیزا درونش هست معمولاً زیر 1cm هست و مادربورد کوچیکه و همین طور سعی می کنند اجزایی که به این باس وصل نزدیک به هم باشند و همچنین سرعت تبادل اطلاعات بسیار بالاست که به این دلایل مشکلی با اون داستانی که گفتم نداریم



* در واقع ما یک آدرس باس و یک دیتا باس داریم، توجه کنید که آدرس از cpu خارج می شود و وارد دیگر I/O Device ها و حافظه می شود، اگر به خطوط دیتا نیز توجه کنید متوجه می شوید که مثلاً به مانیتور دیتا داده می شود و مثلاً از کیبورد دیتا دریافت می شود یک باس کنترل هم داریم که توی، معماری دیدید چه جوری هست، واحد کنترل با ارسال منظم سیگنال (فرمان = command) در لبه هر کلاک ترتیب انجام کارها برای انجام یک دستور را مشخص می کند، بنابراین مشخص است که واحد کنترل باید همه اطلاعات مربوط به وضعیت I/O Device ها، حافظه و مقادیر psw و ... تا بتواند فرمانهای (سیگنالهای) مناسبی را برای Data path ارسال کند



status: سیگنال هایی که وضعیت مقادیر DP را به cu اطلاع می دهد

* توجه کنید که سر حافظه خیلی شلوغه و همه میخوان برن سراغ اش و حافظه های معمولی اینگونه هستند که در هر لحظه فقط 1 نفر می تواند از حافظه استفاده کند پس نیاز به مکانیزم هایی داریم که مشخص کنند که در هر لحظه چه کسی از حافظه استفاده کند

سوال) چرا خیلی از حافظه استفاده میشه؟

1- cpu که برای اجرا هر دستور حداقل 1 بار برای خواندن دستور از حافظه و در زمان fetch سراغ حافظه می رود (البته اگر

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

دستور یا داده درکش باشد از کش بر می دارد)

چرا می گیم حداقل؟ چون ممکن است برای خواندن عملوندهای یک دستور هم به سراغ حافظه بریم

۲- در کامپیوتر فقط cpu نیست که برای fetch دستور سراغ حافظه اصلی میره و خیلی از کارهای دیگه یک پاش حافظه است، مثلاً وقتی می خواید دیتاای را از دیسک بخوانید و به حافظه منتقل کنید، یا از حافظه دیتایی را بردارید و بخواهید روی پرینتر چاپ کنید یا چیزی را اسکن کنید و دیتایش را روی حافظه اصلی بریزید

* کلا هر وقفه سخت افزاری که صادر میشه، OS میاد و متناظر با آن یک ISR مشخص میکند که cpu اجرا کند که در اثر اجراش در اصطلاح می گویند به آن I/O سرویس دادیم و توجه کنید که کد ISR در RAM است پس مراجعه به RAM را در هر صورت داریم.

جمع بندی: در هر لحظه در سیستم کلی پراسی داریم که هر کدوم شون ممکن است با یکی از Device های I/O یا پردازنده کار داشته باشند حالا همه این پراسی ها برای اجرا دستوراتشان و یا ورود و خروج اطلاعات شون با حافظه کار دارند

مکانیزم های کار با حافظه اصلی

۱- programmed input output (PIO)

۲- DMA (Direct memory access)

PIO: در این روش همه نقل و انتقالات از حافظه به وسایل جانبی و از وسایل جانبی به حافظه از طریق cpu انجام می شود

مثلاً فرض کنید دکمه ای از کیبورد را فشار می دهید، کیبورد با فشردن هر کلید وقفه صادر می کند (به پردازنده) (کلید را که فشار می دهید کاراکتر مربوط در بافر محلی کیبورد قرار می گیرد و پس از قرارگیری شماره پورت status اش عوض می شود که عوض شدن پورت status کیبورد باعث صدور وقفه می شود) به بیان دیگر می گویند کنترلر دستگاه از طریق ارسال سیگنال به cpu و از طریق وقفه به درایور دستگاه اطلاع می دهد که کار خود را به پایان رسانده و پس درایور کنترل را به همراه داده یا اشاره گر به داده ها به سایر قسمت های OS که نیاز است برمی گرداند (آن را فراخوانی می کند و داده را نیز بخش می دهد

فشردن کلید ← ایجاد وقفه ← اجرای روتین مربوط ← انتقال داده به حافظه اصلی

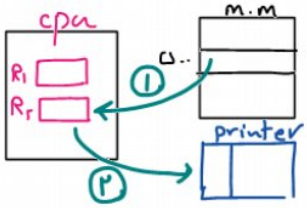
اگر همه ورود و خروج اطلاعات با روش programmed input output باشد و یک cpu داشته باشیم همه ورودها و خروج ها به ترتیب اجرا می شود و conflict ای رخ نخواهد داد.

* وقتی cpu کار نقل و انتقال داده ها را انجام می دهد بین دریافت یا ارسال داده ها فاصله ایجاد می شود (حداقل (بیش از) دوتا Bus cycle بین ۲ داده فاصله می اندازد)، در فیلم I/O دلیل آن را توضیح دادم

1- STR 500, R1
2- L1: LD R2, [R1]
3- STR R2, [216H]
4- INC R1
5- cmp R1, 2500
6- JNE L1

فرض کنید می خواهیم ۲۰۰۰ لغت از خانه ۵۰۰ حافظه بخوانیم و آن را به پورت دیتای یک وسیله جانبی که آدرس اش ۲۱۶H است منتقل کنیم

یک loop داریم که داده ۲۰۰۰ بار اجرا میشه حالا می خوایم یک دور این لوپ، یعنی انتقال ۱ کلمه را از حافظه اصلی به پورت دیتای وسیله جانبی بررسی کنیم.



سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

چرا بین ۲ داده دریافتی توسط وسیله جانبی فاصله می افتد

۱- درخط ۲ داریم یک خونه از حافظه برمی داریم و توی ثبات R_2 که در cpu هست می زاریم پس یک Bus cycle درخط ۲ داریم (وقت مصرف می شود)
۲- در ۳ باید R_2 را از توی cpu برداریم و از طریق باس مشترک آن را در پورت دیتای وسیله جانبی بذاریم که این هم یک Bus cycle زمان می بره
۳- هرخط که هم باید جداگانه fetch و Decode و execute بشود که توی مرحله fetch باز یک Bus cycle نیاز داریم
۴- loop نیز وقت گیر است

* پس مشخص است که بین یک داده و داده بعدی که قراره به وسیله جانبی برسیم فاصله می افتد

حالا اینا رو برای چی گفتم؟

نکته: برخی از وسایل جانبی فقط در مد Burst می توانند کار کنند، یعنی برخی از وسایل جانبی نیاز دارند که میزان دیتای مشخص را بصورت پیوسته دریافت کنند و هیچ فاصله ای در آن میزان مشخص نیفتد

هارد دیسک، کارت صوتی، کارت شبکه، سوم و ... نمونه هایی از این دستگاه ها هستند

بعنوان مثال هارد دیسک نمی تواند یک بیت یا یک بایت را بخواند و حداقل مقداری که در یک مراجعه به هارد دیسک باید خوانده شود یک بلاک (سکتور) است که هر بلاک 512bit = 64Byte است، وقتی هارد دور می گیرد وسط خواندن یک بلاک نمی تواند وایسته، پس یا نباید بریم سراغ هارد یا اگر رفتیم باید حداقل تا آخر یک بلاک سایز بخونیم.

درهفدص کارت صدا هم وقتی مقدار مشخصی از اطلاعات را می خواهد از کارت صوتی بگیرد حتما باید آنها را بصورت پیوسته بگیرد و در حافظه ذخیره کند تا بتواند درست سمپل برداری کند

* خب مشخص است که روش programmed input output برای این وسایل کاربرد ندارد و این روش بیشتر برای Device های جانبی نظیر کیبورد، ماوس و... کاربرد دارد.

DMA: DMA دو تا مد کاری داره که با توجه به نیاز زمان گاهی اوقات از مد ۱ وگاهی از مد ۲ استفاده می کنیم:

۱- Burst mode: در دستگاه هایی که نیاز به این مد دارند

۲- cycle steal/cycle stealing mode: هدف: جلوگیری از اتلاف وقت cpu: برای اینکه وقت cpu برای نقل و انتقالات داده بین حافظه و وسیله جانبی تلف نشود و cpu بتواند به جای انجام این کارها وقت خودش را برای کارهای با ارزش تری نظیر محاسبات بگذارد از این مد DMA استفاده می شود و DMA در این مد سیکل هایی از حافظه اصلی (Bus cycle) که cpu به حافظه نیاز ندارد، به وسایل جانبی دیگر که به حافظه نیاز دارند داده می شود. اگر cpu برای پاسخ به آماده شدن وسیله جانبی از روش Busy waiting استفاده کند خیلی وقت اش تلف می شود (مثال: فرض کنیم داده ای در پورت دیتا چاپگر بنویسیم ولی چاپگر در حال چاپ داده قبلی باشد) برای جلوگیری از اینکار مکانیزم وقفه را ایجاد کردیم و گفتیم به جای اینکه cpu همش پورت status وسیله جانبی را چک کند هر وقت وسیله جانبی کارش تمام شد وقفه بده. اما هنوز جاهای دیگری وجود دارد که وقت cpu تلف میشه مثلاً در کدی که در بالا گفتیم وقت cpu به شدت تلف میشه و cpu باید ۲۰۰... کاراکتر را یکی یکی از حافظه بخواند و آن را درون رجیسترهای خودش بگذارد و پس از درون رجیسترخودش به درون پورت دیتا وسیله جانبی بگذارد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

memory cycle: از دید معماری یادمان هست که واحد خواندن و نوشتن در حافظه اصلی کلمه است (البته ظرفیت حافظه برحسب Byte بیان می‌شود) و همچنین با توجه به Random Access memory بودن حافظه اصلی می‌دانیم زمان خواندن و نوشتن یک کلمه در حافظه اصلی ثابت است بنابراین می‌توان فرض کرد زمان به بازه‌های خواندن یا نوشتن کلمه از حافظه اصلی تقسیم شده که هریک از این بازه‌ها را سیکل خواندن یا نوشتن و در برخی از منابع Bus cycle می‌گویند

Bus cycle: مدت زمانیکه صرف می‌شود که یک کلمه را از رون باس بخوانیم و در حافظه بنویسیم یا بالعکس

نکته) دستگاه‌های مختلف درخواست در اختیار گرفتن حافظه اصلی را در Bus cycle هایی که cpu نیاز ندارد می‌دهند پس باید بین اینها اولویت بندی کنیم که عموماً به اینصورت است که آن کسی که سرعت انتقال بیشتر است اولویتش بالاتر است (تا بتواند زودتر به حافظه دسترسی پیدا کند) بالاترین اولویت با Hard است و عموماً بعد از آن اولویت با باس است (رقابت سر memory cycle)

نکته) برای اطمینان از دسترسی منظم به حافظه مشترک (shared memory) یک memory controler دسترسی به حافظه را سنکرون می‌کند

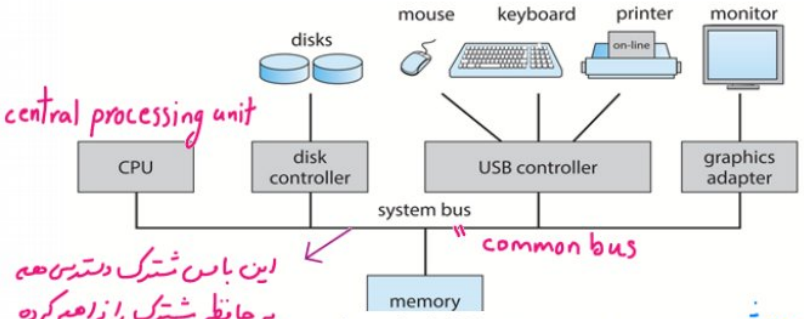
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

یک سیستم کامپیوتری مجموعه‌ای از منابع (Recource) است.

تعریف منبع: موجودیتی در سیستم کامپیوتری که می‌شود آن را گرفت و استفاده‌اش کرد و پس پسش داد، پس رسورس‌هایی با قابلیت استفاده‌ی بود را دارند و به همین دلیل بهشون رسورس‌های reusable می‌گویند ← پس در مورد منابعی صحبت می‌کنیم که بعد از استفاده می‌شود ردیابی‌شان کرد و قابل استفاده مجدد هستند



A typical PC computer system.

cpu کنترل‌کننده‌های دستگاه‌های I/O → می‌توانند به‌صورت سازی اجرا شوند

این باس مشترک دسترسی همه به حافظه مشترک را فراهم کرده

انواع منبع

Active Recource یا processor (پراسسور)

به یک رسورس Active گویند اگر پارامتر سرعت برای آن رسورس مطرح باشد

مثال‌هایی از رسورس Active: cpu ⋮ شبکه (کارت شبکه): برایش سرعت قائلیم و می‌گیم چقدر باند شبکه ما اینقدر است

Hard Disk: یک نرخ انتقال داده از هارد دیسک به سایر اجزا سیستم وجود دارد که می‌توانه به سرعت چرخش صفحه دیسک (مثلاً ۷۲۰۰ RPM = Revolutions Per minute) و کابل انتقال هارد دیسک به مادربرد و seek time (زمان جستجو شیار: مدت زمان حرکت بازوی دیسک و قرارگرفتن هد روی شیار مورد نظر) بستگی دارد

پس نقطه cpu scheduling داریم

* هر پراسسوری را می‌توان زمانبندی کرد (scheduling) بنابراین واژه‌ای بنام processor scheduling داریم

cpu scheduling
Disk scheduling
Network "

scheduling به معنای زمانبندی و بعد از آن نوبت دهی یک معنا عام است و به این معناست که یک رسورس در زمان‌های مختلف در اختیار چه کسی باشد

۲- passive Resource: منابعی که پارامتر سرعت برایشان مطرح نیست

فضا دیسک و فضا حافظه نمونه‌هایی از این نوع منبع هستند. مثلاً یک رم 32GB یا یک هارد 4TB یک passive Resource است.

نکته: همان‌طور که در مثال‌های بالا مشخص است برخی از منابع بسته به اینکه به کدام ویژگی آن توجه کنیم می‌توانیم آن را Active یا passive در نظر بگیریم.

مثلاً در Hard Disk اگر به نرخ انتقال داده‌اش توجه کنیم یک منبع Active است ولی اگر به ظرفیت اش نگاه کنیم یک منبع passive است

انواع passive resource:

۱- سخت افزاری یا فیزیکی (physical): مثل همان فضا دیسک یا فضا حافظه

۲- منطقی (logical): lock نمونه‌ای از یک PR = منطقی است، وقتی یک فایل را باز می‌کنید که روی آن بنویسید (ادیت اش کنید) آن فایل قفل می‌شود (قفل نوشتن روی آن فعال می‌شود) به این معنا که مثلاً یک بیت بعنوان قفل در نظر گرفته می‌شود و در این حالت از ۰ به ۱ تبدیل می‌شود و هر کس که می‌خواهد این فایل را بخواند یا بنویسد به این

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

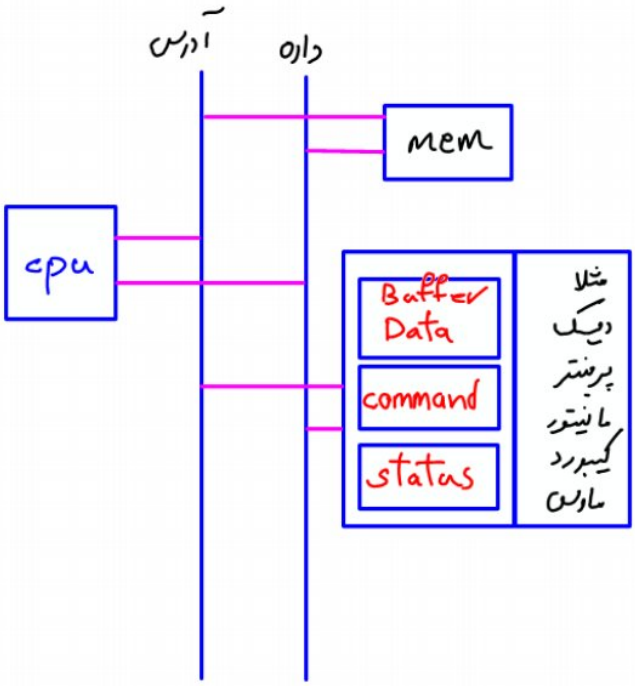
Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

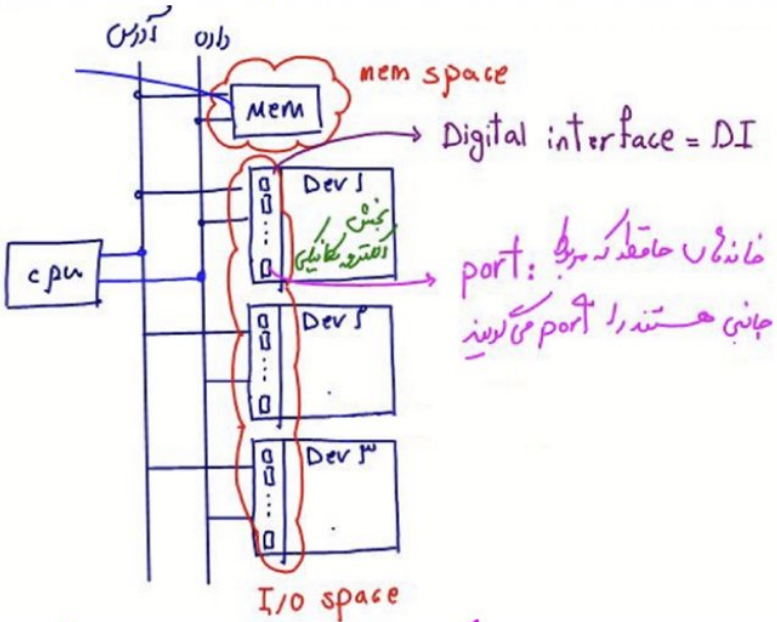
بیت busy می‌زند و اگر ۱ بود نمی‌تواند آن فایل را برای اینکه آن را بخواند یا در آن بنویسد باز کند.
اگر این ریسورس یک بیتی است که وقتی از ۰ به ۱ تغییر می‌کند به این معناست که قفل روی آن فایل گرفته شده و کس دیگری نمی‌تواند آن قفل را بگیرد و اگر این بیت ۰ باشد یعنی آن قفل آزاد است. یک برنامه می‌تواند یک فایل را برای مثلاً نوشتن قفل کند و بعد آن را برگرداند

Adapter = Device controler

همان‌طور که گفتیم یک سیستم کامپیوتری مجموعه‌ای از منابع است و کنار هر منبع یک کنترلر داریم، کنترلر یک سخت‌افزار است که یکسری رجیستر دارد که می‌توان با نوشتن در این رجیسترها تنظیماتی را روی آن منبع انجام دهیم و مشخص کنیم که از چه امکاناتی از آن منبع استفاده می‌کنیم. همچنین برخی از این رجیسترها برای خواندن یا نوشتن داده استفاده می‌شود، به این رجیسترها پورت نیز گفته می‌شود

توجه: اگر یک منبع قابل برنامه‌ریزی باشد آن منبع یک کنترلر سخت‌افزاری خواهد داشت که با نوشتن در رجیسترهای آن کنترلر (برنامه‌ریزی کردن آن کنترلر) می‌توانیم از امکانات آن منبع استفاده کنیم و نرم‌افزاری که می‌تواند آن کنترلر را برنامه‌ریزی کند را درایور گوییم
* تنظیم کردن کنترل‌های منابع کار بسیار پیچیده‌ای است و یک کاربر معمولی توانایی چنین کاری را ندارد، به همین دلیل این تنظیمات توسط نرم‌افزاری به نام درایور (Driver) انجام می‌شود.





portهای (همان رجیسترهای) کنترلر به ۳ دسته تقسیم می‌شوند:

۱- Data یا بافر: این پورت ۲ طرفه است و هم خواندنی و هم نوشتنی است: هم سخت‌افزار می‌تواند در آن بنویسد یا از آن بخواند و هم درایور (OS) ← درایور کنترلر را programl می‌کند

۲- command (فرمان): از دید cpu فقط نوشتنی است، درایور با نوشتن در این رجیستر از طریق کنترلر منبع به سخت‌افزار فرمان‌های خاص را می‌دهد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۳- status (وضعیت) : از دید cpu فقط خواندنی است

- کنترلر با بررسی این رجیستر متوجه می‌شود که چه کاری باید انجام دهد، نوشتن در این رجیستر باعث آغاز عملیات I/O در سخت‌افزار می‌شود

مثال از command :

خواندن ورودی‌های دستگاه‌های I/O ای که از کاربر ورودی می‌گیرند، مثلاً خواندن از ماوس، کیبورد

نوشتن روی صفحه مانیتور

خواندن و نوشتن روی دیسک (هارد) یا CD یا فلاپی

حرکت بازوی دیسک

- وقتی درخواست خواندن می‌دهد، سخت‌افزار مربوطه داده‌ای که می‌خواند را در بافرش قرار می‌دهد، مثلاً خواندن فرمی شماره K از یک CD یا خواندن سکتور K از هارد دیسک یا بلاک K از Tape یا...

- وقتی درخواست نوشتن می‌دهیم باید داده‌ای که قرار است نوشته شود را نیز در بافر کنترلر مربوط به آن سخت‌افزار قرار دهیم مثلاً نوشتن در سکتور K هارد دیسک یا پرینت اطلاعات یا...

مثال از status :

- عملیات به پایان رسیده
- بافر من خالی یا پر است
- در حال اجرا عملیات هستم
- با فلان خطا مواجه شدم (مثلاً پرینتر جوهرش تمام شده، در مورد هارد سکتوری که می‌خواسته از آن بخواند خراب است و ...)

نکته : بیان آنکه یک وسیله جانبی یک کاری انجام دهد باید یکسری از کارها به ترتیب (ترتیب کدها و عملیاتی که باید کنترلر انجام دهد) انجام شود.

* فیلم مربوط به I/O که در داشبوردتان قرار گرفته است و مربوط به درس معماری است مشاهده کنید

مثال :

فرض کنید Dev1 پرینتر و آدرس پورت کامپیوتر بصورت زیر است.

status : ۱۳۰ H
data : ۲۸۸ H
command : DDD H

command
۱ : کاراکتری که بهت میدم چاپ کن
۲ : کاغذ را بیرون بده
۳ : خاموش بشو
۴ : stand by برو به
۵ : از stand by خارج شو

status
۰ : جوهرم تمام شده
۱ : کاغذ درون گیر کرده
۲ : من خاموشم
۳ : کاغذم تموم شده
۴ : مشغول چاپ‌ام
۵ : من stand by
۶ : من آماده چاپ‌ام

مثالی از آدرس پورت‌ها و نمونه‌ای از مواردی که command و status می‌توانند باشند.

مثلاً هر یک از command و status می‌توانند ۸ بیتی باشند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته: هر چقدر هم یک فرد به رشته کامپیوتر و کد زنی مسلط باشد برای ارتباط با وسایل جانبی باید:

۱- آدرس پورت ها را بداند (پورت خانه‌ای از حافظه است و آدرس دارد)

۲- کدهای command، status هر دستگاه I/O ای که از آن استفاده می‌کند را باید بلد باشد

۳- ترتیب صحیح آنکه هر وسیله جانبی هر کار را چطور انجام می‌دهد بلد باشد

نکته) به اینها می‌گویند Device language آن وسیله جانبی، هر وسیله جانبی DL خودش را دارد و هر شرکتی هم DL خودش را دارد

تعریف Driver: درایور نرم افزاری است که درون سیستم عامل قرار دارد (قرار می‌گیرد) و زبان کنترلر مربوط به خودش را می‌شناسد و می‌تواند تنظیمات مربوط به آن منبع را (Device) را انجام دهد (مقادیر رجیسترها را تغییر دهد)

چرا؟ چون Driver هر منبع را شرکت سازنده آن منبع ارائه کرده

* Driver هر دستگاه جانبی یک برنامه است، اگر یک وسیله جانبی مثلاً ۱۰ تا کار بلده انجام بده، خود شرکت سازنده آن وسیله جانبی ۱۰ تا روتین می‌نویسه برای هرکدام از آن کارها و به مجموعه‌ای از روتین‌ها یک وسیله جانبی درایور آن وسیله می‌گویند، برای انجام یک کار مشخص درایور روتین مربوط به آن را فراخوانی می‌کند، OS درایور را فراخوانی می‌کند و درایور روتین مربوط را.

- درایور یک مترجم است که باید هم زبان سخت افزار مربوط به خودش را بلد باشد و بشناسد و از طرف دیگر زبان سیستم عاملی که روی آن سیستم نصب است (ویندوز یا لینوکس یا...) را بلد باشد و بشناسد، پس مشخص است که چه منبع عوض بشود و چه OS عوض بشود باید Driver عوض شود.

نکته: برای انجام یک عمل I/O درایور مشخص می‌کند که کدامیک از عملیات کنترلر و با چه ترتیبی انجام شود.

* درایور در ۲ جا استفاده می‌شود

۱- انجام عمل I/O ای که OS از آن می‌خواهد ← یک برنامه از OS خواسته و OS از درایور می‌خواهد

۲- راه اندازی اولیه وسایل جانبی هنگام Boot شدن سیستم.

- درخصوص مورد اول در بالا مثال‌ها فراوانی زدیم، حال چند مثال درخصوص مورد ۲ می‌زنیم:

با نوشتن در برخی از رجیسترهای کنترلر graphice adapter مواردی نظیر میزان روشنایی صفحه، رزولوشن صفحه، تعداد رنگ‌های صفحه (از چند رنگ برای نمایش تصاویر استفاده شود، مثلاً از monochrome تا ۶۸ میلیارد رنگ در هر پیکسل در پنل‌های ۱۲ بیتی)، کنتراست صفحه، تنظیم نرخ رفرش ریت صفحه (۶۰Hz یا ۱۲۰ یا ۱۴۴ یا ۲۴۰ و...) را تنظیم می‌کنیم

Buffering and spooling: بین فرستنده و گیرنده اختلاف سرعت وجود دارد و باید اختلاف سرعت بین فرستنده و گیرنده را تعدیل کنیم، اگر بین فرستنده و گیرنده اختلاف سرعتی وجود نداشته باشد اصلاً نیازی به نگهداری داده در حافظه (هر حافظه‌ای اعم از حافظه اصلی یا حافظه‌های ...) وجود ندارد.

Buffering: دستگاه‌های مرتبط با هم در یک سیستم کامپیوتری لزوماً از نظر سرعت مانند هم نیستند، یا اینکه درگیر کار دیگری

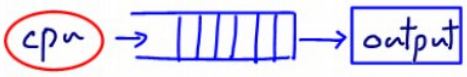
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

هستند حال برای آنکه بتوانیم این موضوع را مدیریت کنیم باید از بافر استفاده کنیم، بافر یک حافظه‌ای است بین یک دستگاه نرسنده و گیرنده.

نرسنده می‌تواند یک دستگاه ورودی باشد و گیرنده cpu باشد در این حالت ممکن است وقتی یک دستگاه ورودی می‌خواهد چیزی را برای cpu ارسال کند، cpu درگیر کار دیگری باشد و نتواند به آن ورودی یا ورودی‌ها پاسخ بدهد در اینصورت ورودی‌ها باید بافر شوند تا cpu بعداً به آنها سرویس دهد

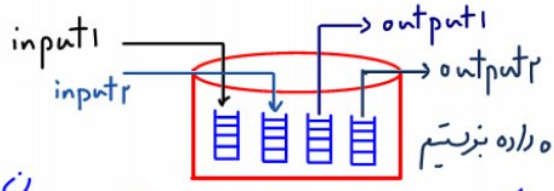
از طرف دیگر فرستنده می‌تواند cpu و گیرنده یک دستگاه خروجی باشد و ما داریم بین دستگاه‌های ورودی و خروجی و cpu اختلاف سرعت وجود دارد (سرعت cpu بیشتر است معمولاً) در اینصورت در داده‌ای که cpu برای دستگاه خروجی ارسال می‌کند باید بافر شود



spooling: از دیسک بعنوان یک بافر خیلی بزرگ استفاده کنیم.

ورودی دستگاه ورودی ۱ در یک مکان از دیسک بافر شود

" " " ۲ " " مکان دیگری از دیسک بافر می‌شود



از یک بافری که قبلا توسط یک ورودی به وجود آمده می‌توانیم برای یک خروجی مانند output1 داده بزنیم

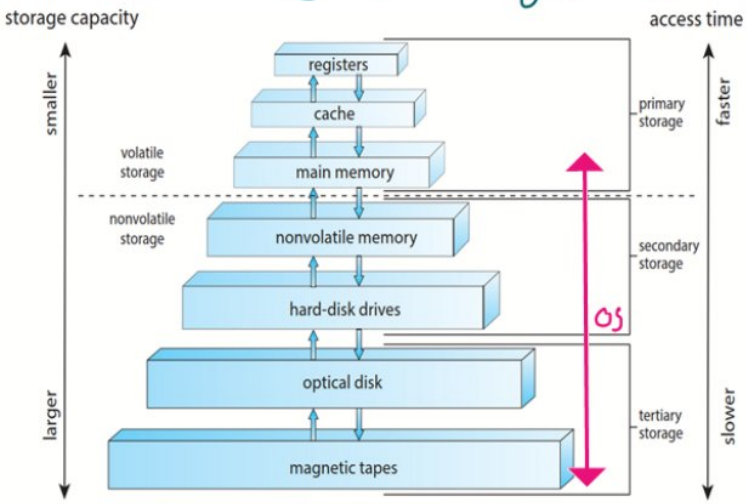
* اگر فقط بافر داشته باشیم و تکنیک spooling را نداشته باشیم عملیات ورودی و خروجی یک پراسس با پردازش همان پراسس همزمان می‌شود ولی با وجود spooling عملیات ورودی و خروجی پراسس‌ها مختلف می‌توانند با پردازش پراسس‌های دیگر همزمان شود، بنابراین بطور همزمان می‌توانیم با چند دستگاه I/O در ارتباط باشیم (همراه با مکانیزم‌هایی مانند DMA و... موثر است)

توجه) اگر از یک حدی بیشتر بخواهیم از بافر بعنوان بافر استفاده کنیم چون اینجوری تعاملات با بافر خیلی زیاد میشه و بافر خیلی کند است عملکرد سیستم مختل خواهد شد

توجه) در برخی از کاربردها که سرعت مهم است ممکن است طراح نخواهد از spooling استفاده نکند به این طریق که بافر برای آن عملیات را در نظر بگیرد.

با توجه به سرعتی که می‌توانیم در spooling داشته باشیم

* توجه کنید که از منظر سرعت بافر در سه پراسور است و بنابراین بحث نوبت دهی (scheduling) مطرح است.



سوال: ثبات‌ها کجان؟ cpu مدیریت شون با کیه؟

کاربر ساده: خیر
برنامه‌نویس (کاربر) سطح بالا: خیر
برنامه‌نویس اسمبلی: بله

کامپایلر: بله
اسمبلر: بله

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

program Vs process (context)

نکته: یک فایل اجرایی یک برنامه کامپایل شده و قابل اجرا روی هارد است وقتی آن را فعال کنیم مثلاً با دابل کلیک روی آن، برنامه مورد نظر فعالیت خود را از آغاز خواهد کرد.

برنامه ها به ۲ دسته سیستمی و کاربردی تقسیم می‌شوند

۱- system program: برنامه هایی که برای کار با سیستم توسط متخصصان نوشته و ت شده اند و سطح دسترسی بیشتری دارند و برخی از آنها به سخت افزار آگاه ترند.

۲- Application program:

نکته: آیا اگر که اجرایی یک برنامه که توی هارد قرار دارد را به RAM منتقل کنیم و هیچ کار دیگری انجام نشود آن که توسط cpu قابل اجرا است؟ خیر.

* پس چه چیزایی می‌خواد تا بتونه اجرا بشه؟ برای پاسخ به این سوال باید ببینیم اصلا برنامه ها چیکار می‌کنند موقع اجرا؟

۱- یک برنامه میاد روی یکسری داده ورودی کار میکند و داده خروجی تولید میکند و روی اونها هم باز میتونه کار کند، پس یک فضایی نیاز داره برای کار با این داده ها که این فضا هم پویا است ⇐ پس به یک فضا Data و Heap نیاز دارد

۲- یک برنامه برای کارکردنش نیاز داره توابع خودش یا توابع موجود در OS را فراخوانی کند، خب می دونیم برای فراخوانی توابع بصورت تودر تو نیاز به پشته داریم، از پشته برای ذخیره متغیرهای محلی تابع فراخوانی شده، نگهداری آدرس برگشت و دادن پارامتر به توابع استفاده می‌شود ⇐ پس به یک فضا stack نیاز دارد

۳- برای انجام برخی از کارهایش به منابع سیستم نیاز دارد ⇐ پس یکسری منبع باید بش تخصیص داده بشود مثل فضای RAM، cpu، فایل هایی که نیاز دارد، چاپگر... منابعی که فرآیند دارد باهاش کار می‌کند و در انحصار و مالکیت آن است جزئی از فرآیند محسوب می‌شود

مثال: یک نقشه ساختمان بدید به یک مهندس عمران می‌تونه خونه بسازه؟

۴- pcB مربوط به یک فرآیند (به ازای هر فرآیند یک entry در page table نیاز داریم)

* پس یک فرآیند ۳ بخش دارد

۱- text or code

۲- بخش هایی که برای اجرا شدن نیاز دارد pcB و stack، Heap، Data

۳- resource

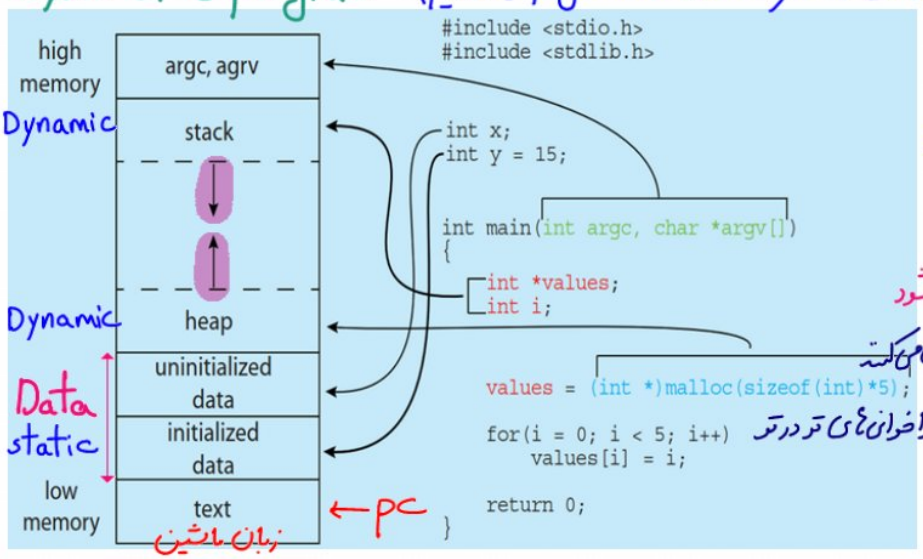
- stack: ۳ مورد در پشته ذخیره می‌شود

۱- متغیرهای محلی که توابع (زیر برنامه ها) تعریف می‌کنند

۲- آدرس های برگشت به همه تابع ⇐ برای فراخوانی های تودرتو

۳- پارامترهایی که به توابع پاس می‌کنند

layout of c program

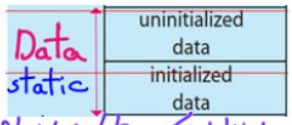


سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

متغیرهای سراسری: متغیرهای سراسری را می‌توان در همه قسمت‌ها و توابع برنامه استفاده کرد



Heap: حافظه‌ای که در طول زمان اجرای برنامه به صورت پویا به آن تخصیص داده می‌شود

داده‌های داینامیک در اینجا ذخیره می‌شود، مثلاً برنامه می‌تواند لیست پیوندی داشته باشد، اگر برنامه نیاز به حافظه پویا داشته باشد تابع malloc را صدا می‌زند (تابعی در کتابخانه c، API) و درخواست حافظه می‌کند که اگر OS بخواهد بخش فضا را Allocate می‌کند.

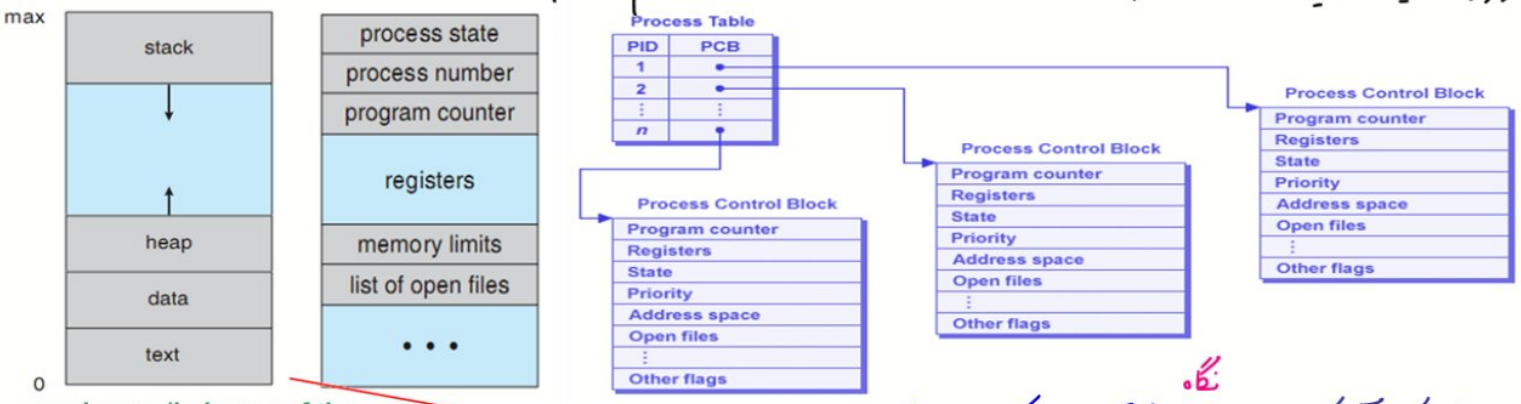
* یک بخش جداگانه در بالای فضای آدرس برای پارامترهای main وجود دارد

program vs process

job چند برنامه‌ای است (process = task)

program یک موجودیت غیرفعال (passive) است، program همان تکست زبان ماشین است که ما برای برنامه‌مان داریم.

حال هرگاه یک برنامه قابل اجرا را برای اجرا فعال کنیم (مثلا با دابل کلیک) OS علاوه بر آوردن بخشی از آن (در برخی از روش‌های قدیمی یا ماشین خاص متفاوت می‌تواند کامل باشد) به RAM و ثبت و تهیه و نگهداری اطلاعات و شناسنامه برای مدیریت آن و تعریف یکسری متغیر ایستا و مقداردهی اولیه برخی از آنها یک موجودیت جدید می‌سازد که به جز متن برنامه موارد دیگری که به آنها اشاره شد مانند stack, Data در کنارش دارد، وظیفه فرآیند آن است که برنامه مورد نظر را به کمک سخت افزار و سیستم عامل اجرا کند.



* حافظه یک فرآیند تک رشته: تصور برای این است که از ابتدا تا انتهایش بصورت پشت سرهم در حافظه قرار گرفته (از ۰ تا سایز حافظه مورد نیاز برنامه) اما می‌دانیم که چون از حافظه مجازی استفاده می‌کنیم حافظه بصورت پیوسته به پرایس داده نمی‌شود

نکته: فرآیند فقط به Address space خودش و مناجی که به آن اختصاص داده شده (مثل فایل‌ها و ...) اجازه دسترسی (خواندن و نوشتن) دارد و اجازه دسترسی به جای دیگری را ندارد.

توجه: فرآیند (و ترد) موجود Active کاپسوله است و بقیه هم passive هستند، بقیه هم ابزارهایی هستند که فرآیندها از آنها استفاده می‌کنند تا بتوانند کارهایشان را انجام دهند، مثلاً cpu یک ابزار است که فرآیند از آن استفاده می‌کند برای اینکه خودش را اجرا کند، کش، حافظه و ... همگی برای فرآیندها ابزار هستند.

نکته: برنامه ما (که exe) روی هارد (CD یا فلش و ...) در حالت passive قرار دارد و غیرفعال است، حال وقتی روی فایل exe برنامه دابل کلیک می‌کنید که بازش کنید در این زمان یک برنامه‌ای از OS بنام مدیر فرآیندها یک پرایس ایجاد می‌کند

شلاً در ویندوز روال create process این کار را می‌کند.

" " " " " fork " unix

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

* تا اینجا متوجه شدیم که فرآیند موجودیتی است که برای اجرا برنامه ساخته می‌شود
این موجودیت (پراسس) با بکارگیری سخت افزارها و منابع سیستم و به کمک OS وظیفه محول شده به خودش را انجام می‌دهد

نکته: یک برنامه کاربردی ممکن است بزرگ باشد و برای اجرا شدن آن فرآیندهای زیادی دست به دست هم دهند که هریک مسئولیت جداگانه‌ای داشته باشند و با هم به تبادل اطلاعات بپردازند

مثال: word : وقتی نرم‌افزار word را باز می‌کنید، به مرور زمان فرآیندهای متفاوتی به وجود می‌آید (یک فرآیند می‌تواند فرآیندهای دیگری را به وجود می‌آورد)، مثلاً فرآیندهای زیر به وجود می‌آید:

۱- یک فرآیند مسئول دریافت اطلاعات از کاربر و درج آنها
۲- " " " چک کردن املای کلمات تایپ شده یا کپی شده توسط کاربر
۳- " " " ایجاد صفحات جدید و صفحه آرایی
⋮

نکته: توجه کنید که وقتی یک cpu داریم در یک لحظه از زمان فقط یک فرآیند در سیستم می‌تواند در حال اجرا باشد

نکته: هر فرآیند آنقدر در کامپیوتر می‌ماند تا کارش تمام شود یا در صورت بروز شکل توسط OS از بین برود (Kill شود) که در هر دو این حالات منابع تخصیص داده شده به آن فرآیند ازش گرفته می‌شود

چرخه ساده‌ای از زندگی یک فرآیند در single programming: یک فرآیند می‌تواند در چه وضعیت‌هایی باشد

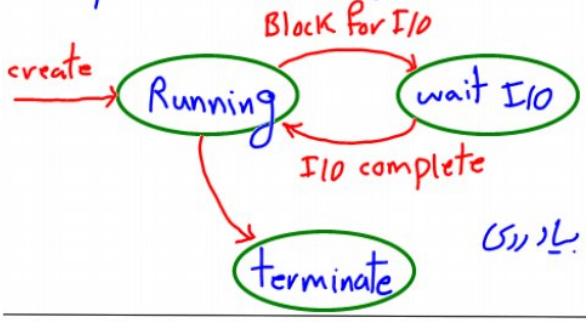
single programming: در هر لحظه فقط یک فرآیند در سیستم وجود داشته باشد و تمامی زمان cpu در اختیار آن قرار گیرد (فقط یک برنامه به همراه OS درون RAM است ← تلاف شدید RAM + تلاف سایر منابع + برنامه اگر بزرگ تر از حافظه موجود باشد؟

بعد از تولد فرآیند اجرا دستورالعمل‌های آن به صورت روبه رو آغاز می‌شود:

Running — Nonpreemptiv
→ I/O request: خواندن داده از ورودی یا نوشتن داده در خروجی
cpu wait for I/O processor
← I/O complete
Running
⋮
→ process termination

* ← خود cpu در قبال انجام عمل I/O نمی‌رود و cpu زمان آن را صادر می‌کند و I/O processor‌ها هستند که داده ورودی را به حافظه اصلی منتقل می‌کنند و یا نتایج را از حافظه اصلی به خروجی منتقل می‌کنند

* در single programming در مدتی که I/O processor دارد کارهای I/O مربوط به پراسس را انجام می‌دهد cpu در حالت wait است و بیکار است !



* در ۲ جا cpu بیکار می‌ماند
۱- فرآیند جاری درخواست I/O بدهد و همچنین هیچ کار دیگری برای انجام نداشته باشد پس باید Block شود (بخوابد) تا جواب I/O آماده شود
۲- فرآیند جاری تمام شود در این صورت تا طول جدید تبدیل به فرآیند شود (از روی هارد بیاید روی RAM) cpu کاری برای انجام ندارد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته: مشخص است که single programming انتفاع است، اگر برنامه‌ای مانند ورد منتظر ورود داده توسط کاربر باشد و کاربر در حال نوشیدن قهوه باشد یا اصلا اینکه کاربر تصمیم بگیرد چه داده‌ای را وارد کند فقط ۳ ثانیه طول بکشد در این مدت cpu بیکار است در صورتیکه در این مدت می‌تواند چندین میلیون دستور را اجرا کند

سرعت پردازنده

- وقتی می‌گویند فرکانس پردازنده مثلاً 4GH است به چه معناست؟ یعنی در ۱ ثانیه ۴ میلیارد کلاک زده می‌شود
- یک دستور برای اجرا شدن باید سیکل اجرا که شامل fetch، Decode، operand fetch و exe را طی کند، حال طی شدن این مراحل برای دستورات مختلف نیاز به تعداد کلاک‌های متفاوتی دارد، مثلاً یک دستور ۱۰ کلاک و دستور دیگری ۲۰ کلاک نیاز دارد، بنابراین پارامتری تعریف می‌شود بنام MIPS (million inst per sec) که بیانگر متوسط تعداد میلیون دستوری است که یک پردازنده در یک ثانیه اجرا می‌کند

بعنوان مثال اگر MIPS یک پردازنده ۷۵۰۰۰۰ است یعنی بطور متوسط در هر ثانیه می‌تواند ۷۵۰۰۰۰ میلیون دستور را اجرا کند

نتیجه‌گیری: انسان‌ها و وسایل جانبی مثل پرینتر، ماوس و ... بسیار کند هستند و cpu بسیار بسیار سریع است، بنابراین cpu نباید ۱- خودش به فکر انجام دادن کارهای I/O باشد

۲- cpu نباید معطل I/O شود (زمان انجام I/O بیکار بنشیند)

۳- cpu قدرت کار با وسایل جانبی را ندارد !

Multi programming

بیش از یک پراسس در حافظه لود بشود ولی در هر لحظه یکی شون اجرا بشود، یعنی یک پراسس در حال اجرا به هر دلیلی متوقف شد یا تمام شد، cpu، یک فرآیند دیگر که آماده است تا اجرا شود (ready است) را اجرا خواهد کرد.

توجه: وقتی یک فرآیند ایجاد می‌شود سیستم عامل state آن را ready (آماده) می‌گذارد، اگر cpu را به یک فرآیند ready بدهیم شروع به اجرا برنامه مورد نظرش می‌کند و state اش به running تغییر خواهد کرد

* پس یک صفی از فرآیندهای ready وجود دارد که وقتی cpu خالی می‌شود scheduler ← سیستم عامل بر اساس یکسری الگوریتم زمانبندی یکی از آنها را انتخاب و برای اجرا به cpu ارسال می‌کند به اینکار Dispatch می‌گویند

* پس این امکان که کامپیوتر بتواند از یک فرآیند در حال اجرا به یک فرآیند دیگر سوی کند قبل از اینکه کار فرآیند اول تمام شده باشد و بعد امکان این وجود داشته باشد که برگردد و کار نیمه‌ها کرده را ادامه دهد مالتی پروگرامینگ می‌گویند (وقتی مثلاً برمی‌گردیم روی ورد که نباید از اول شروع کند)

هدف Multi programming

افزایش بهره وری کامپیوتر و استفاده بهینه از منابع ← بهینه شدن استفاده از I/O Device

* cpu utilization: cpu: میزان کار مفید انجام شده توسط cpu در واحد زمان

* این روش باعث شد انتظار یک پراسس برای I/O باعث پایین آمدن cpu utilization نشود. Nonpreemptiv

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

برای multi programming نیازبه ۳ تا چیز داریم :

۱- memory management : برای آنکه حافظه برنامه ها باهم قاطی نشود

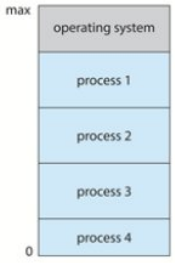
۲- cpu scheduling : پس از اتمام یک برنامه و همین طور پس از توقف یک برنامه برای I/O کدامیک از پراسس های RAM را برای Dispatch کردن به cpu انتخاب کنیم

۳- حفاظت

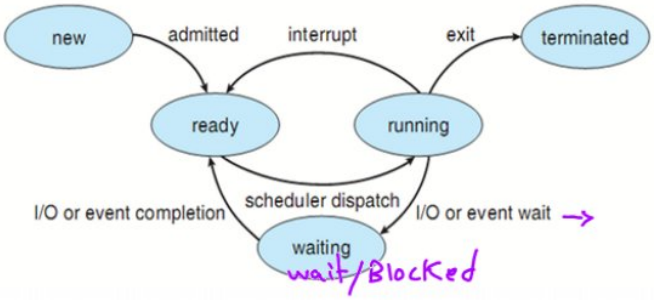
نکته : اضافه کردن پراسس (برنامه) تا یکجایی کارایی را بالا می برد ولی از یکجایی به بعد به دلیل وجود سربار os کارایی پایین می آید

نکته : درجه چند برنامه سازی (چند برنامگی) : حداکثر تعداد پراسس هایی که می توانیم در یک زمان فعال داشته باشیم

چرخه ساده ای از زندگی یک فرآیند در Multi programming : یک فرآیند می تواند در چه وضعیت هایی باشد



Memory layout for a multiprogramming system.



نیاز به خواندن یا نوشتن (Blocked)
منتظر پایان از فرآیند دیگر (wait)

نکته : ورد را که باز می کنیم یک فرآیند است (می شود) و تا زمانیکه دکمه close اش را نزدیم (از بین اش نبردیم) یک فرآیند به حساب می آید فقط در این مدت (از زمان ایجاد تا زمانیکه می بندیم اش) می state اش عوض می شود، حالت های یک فرآیند در طول زندگی اش چندین بار تکرار می شود. ← اگر یک انسان را یک فرآیند در نظر بگیریم آیا وقتی انسان می خوابد دیگر انسان نیست؟ * مسابقه نوبتال

نکته : در کامپیوتر در یک لحظه ممکن است صدها یا هزاران فرآیند زنده وجود داشته باشند و ده ها فرآیند در حالت آماده باشند ولی اگر یک پردازنده هم منظوره داشته باشیم در هر لحظه حداکثر یک فرآیند در حالت Running است.

نکته : os یک برنامه است و os نیز مجموعه ای از فرآیندها است، برخی از قسمت های os مکرر و زیاد مورد استفاده قرار می گیرند به همین علت معمولاً این بخش ها به حافظه اصلی می آیند و در آنجا تا خاموش شدن سیستم مقیم می شوند، اما پراسس های اغلب برنامه ها با اتمام کاری که قرار است انجام دهند از بین می روند اما برخی از قسمت های os از این قاعده مستثنی هستند.

نکته : بیشتر تلاش پیاده سازی os ها صرف فراهم آوردن امکان multi programming است، os ها به گونه ای طراحی شده اند که با وجود آنکه صدها فرآیند همزمان فعالیت می کنند و هر یک نیازهای متفاوتی دارند و کارهای متعادلی می کنند هیچ فرآیندی اثر منفی روی عملکرد فرآیند دیگر نگذارد ← یکی از کارهای مهم os مدیریت فرآیندها است.

نکته : program
process or task
job ← تعریف یا استاندارد تر است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

تعریف ۱: کوچکترین واحد کاری معنادار در مقیاس پراسس مربوط به خودش است، بعنوان مثال در پخش ویدیو، پخش هر فریم یک job است، یک پراسس ممکن است بسیار طولانی باشد و بخواهد تعداد زیادی کار پشت سرهم انجام دهد

ترجمه: توجه کنید که درست است که inst کوچکترین واحد کاری معنادار است اما به job، inst نمی گویند چون job کوچکترین واحد کاری معنادار در مقیاس پراسس خودش است و inst یک کار معنادار در مقیاس یک پراسس نیست

تعریف ۲: وقتی یک کاری هنوز فقط request اش وجود دارد ولی باهاش موافقت نشده که بیاد لود بشه توی حافظه (بیاد توی سیستم)

multi tasking = Time sharing

time slice = quanta = quantum

نه تنها کار multi programming را انجام بدهیم بلکه حتی اگر برای ال نرفت دنبال I/O ولی بهمیه زمانی اش تمام شد در آن شرایط هم cpu را به پراسس بعدی بدهیم بطوریکه همه برنامه ها (پراسس ها) همروند باهم پیشرفت کنند و جلو بروند.

همروند (concurrent): روی یک core دو تا پراسس (یا نخ) می توانند باهم اجرا شوند، یعنی یکی از پراسس ۱ اجرا میشه، بعد می ریم یکی از ۲ را اجرا می کنیم، بعد باز sw می کنیم روی اولی و ... ، در این حالت می گیم این پراسس ها دارند باهم همروند اجرا می شوند، یعنی روندشون باهم همراه است و همروند هستند ولی موازی نیستند چون یک core داریم.

موازی (parallel): اگر حداقل دو core داشته باشیم و مثلاً پراسس ۱ را روی یک core و پراسس ۲ را روی core دیگری بفرستیم در این حالت می گوییم این ۲ پراسس (یا نخ) دارند بصورت موازی اجرا می شوند.

هدف: کاهش متوسط زمان انتظار کاربران کامپیوتر برای بدست آوردن result هایی که به آنها نیاز هستند.

interactive multi programming = Time sharing

نکته) Time sharing باعث بهبود رضایت کاربران می شود اما تعداد context switch ها را افزایش می دهد بنابراین بهره وری cpu را کمی نسبت به multi programming کاهش می دهد

سوال) آیا Time sharing به معنای عدالت است؟ آیا در Time sharing قحطی (گرسنگی = starvation) برای پراسس رخ نمی دهد؟ خیر به معنا عدالت نیست

اگر مکانیزم زمانبندی ما عادلانه نباشد یک فرآیند با اولویت پایین می تواند دچار starvation شود

* خب از کجا باید بفهمیم که بهمیه زمانی یک فرآیند تمام شده و اصلا کی این بهمیه رو مشخص میکند؟

Timer: تایمر یک سخت افزار است که کمی جلوتر بصورت دقیق در مورد نحوه پیاده سازی تایمر صحبت می کنیم

شاید این ایده به ذهن شما برسد:

خود برنامه ها زمان سیستم را بخوانند و مثلاً اگر 15ms سپری شد خودش بفهمه که نوبت اش تمام شده و یک syscall (دلتدر) به os بده و نوبت را بده نفر بعدی.

این روش ۲ ایراد دارد:

۱- شاید آن برنامه عمداً cpu را به os برنگرداند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۲- " " " باگ نرم افزاری داشته باشد و مثلاً در یک لوپ بی نهایت گیر کند و اصلا به دستوری که در آن قرار است زمان cpu را بخواند نرسد

راه حل: OS تایمر را به یک عدد مشخص برای هر پراسس ست کند.

دقیق تر: موج سینوسی بنام frequency refrence signal و با مدارهای الکتریکی به موج مربعی تبدیل می شود

آشنایی با برخی از اصطلاحات:

cpu's clock: clock generator یک cpu با استفاده از crystal oscillator پالس (pulse) تولید می کند که برای سنکرون کردن operation ها و المان هایی که با کلاک کار می کنند استفاده می شود

ساعت سیستم: داستان cmos و حافظه هایی که نقش آن را ایفا می کند

Timer: در زیر توضیح داده شده

computer operating properly = watchdog timer: سیستم کامپیوتری در صورتی که درست کار کند بصورت پریودیک این را ریست می کند، اگر کامپیوتر هنگ کند یا خیلی کند کار کند این تایمر ریست نمیشه و تا انتهایی که برایش تعیین شده برود سیستم را ریست می کند ← فضاپیما، ربات های جستجوگر دیگر سیارات

پیاده سازی تایمر (Timer): ما باید اطمینان حاصل کنیم که OS کنترل cpu را تحت کنترل خودش نگه می دارد و نباید اجازه دهیم (OS نباید اجازه دهد) که یک برنامه یا به وسیله گیر کردن در حلقه بی نهایت و یا عدم فراخوانی system call هرگز کنترل را به OS برنگرداند، پس OS قبل از اینکه cpu را به یک پراسس بدهد با تنظیم تایمر به اندازه یک پریود مشخص زمان در اختیارگیری cpu توسط آن پراسس را مشخص می کند. این پریود می تواند ثابت (مثلاً $\frac{1}{60}$ ثانیه) یا متغیر باشد (مثلاً از ۱ میلی ثانیه تا ۱ ثانیه)

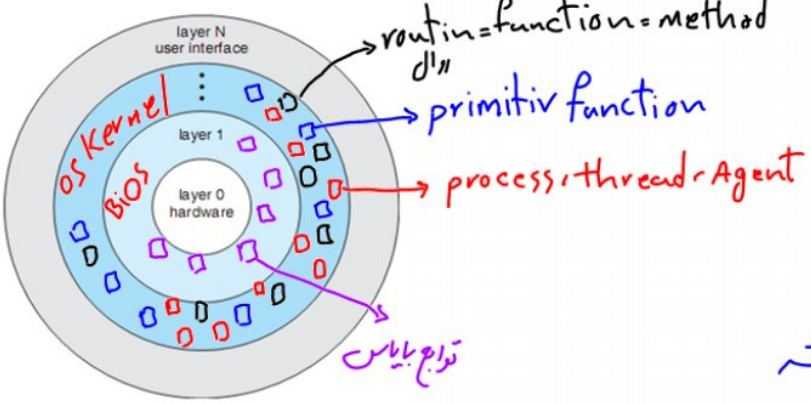
یک تایمر متغیر (variable timer) معمولاً با fixed-rate clock و یک کانتر پیاده سازی می شود

نحوه پیاده سازی: فرض کنید شمارنده ۱۰ بیتی باشد و پریود کلاک ۱ میلی ثانیه باشد، OS می تواند شمارنده را مقداردهی اولیه کند و با آمدن هر کلاک شمارنده یکی کم شود و وقتی که شمارنده به ۰ برسد یک وقفه تایمر صادر کند، به این ترتیب اجازه وقفه در فواصل زمانی ۱ تا ۱۰۲۴ میلی ثانیه با گام های 1ms را خواهیم داشت.

ساعت سیستم: برای نگهداری زمان استفاده می شود روش کار ساعت به اینصورت است که یک زمان سنج (تایمر) مثلاً هر یک میلی ثانیه (یک هزارم ثانیه) یک وقفه ارسال می کند و از شمارش این وقفه ها و تقسیم های لازم تاریخ و زمان هر لحظه مشخص می شود

پراسس ها چطور کارهایشان (کارهای مربوط به اجرای برنامه ها) را انجام می دهند

قبل از پاسخ به این پرسش بهتر است کمی در خصوص معماری سیستم عامل های لایه ای صحبت کنیم



Bios: بعدا بیشتر در مورد بایاس صحبت می کنم ولی بدانید برخلاف آنچه که اکثر اساتید می گویند Bios یک برنامه وسیع است و یکسری از کارهای اساسی (input و output های اساسی) را برایمان انجام می دهد و Bios همیشه هست و همیشه هم بهش نیاز داریم و ازش استفاده میشه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

- primitive function : برنامه های کوتاهی هستند که قسمت های کوچکی از کارها را انجام می دهند، خودشان برای انجام کارهایشون از دستورات زبان ماشین و توابع Bios استفاده می کنند، وقتی عملیات primitive شروع می شود تا پایان نمی توان آن را متوقف کرد و کار دیگری را شروع کرد.

- پراسس های درون کرنل : برای انجام کارهایشان می توانند از روتین های primitive ها، دستورات زبان ماشین و توابع Bios استفاده کنند

* کرنل روال ها و... اساسی زیادی دارند و توسط طراحان os به دقت ساخته و تست شده و آماده شده تا ما بتوانیم از آنها استفاده کنیم، روال های درون کرنل می توانند همه دستوراتی که در آن کامپیوتر تعریف شده و وجود دارند را اجرا کنند، مثلاً اگر ثبات IR در یک کامپیوتر (در cpu) ۸ بیتی باشد، آن ماشین ۲۵۶ دستور دارد (البته معمولاً ۲۵۶ تا!) که در مد کرنل هم این ۲۵۶ دستور قابل اجرا است

* دستورالعمل های یک کامپیوتر به ۲ دسته تقسیم می شوند

۱- دستورات معمولی (Non-privilege instruction) : دستوراتی که همه برنامه های کاربردی می توانند از آنها استفاده کنند دستوراتی مانند ADD، sub، mul، cmp و...

۲- " ممتاز (privilege instruction) : دستوراتی که فقط هسته سیستم عامل می تواند آنها را اجرا کند دستوراتی مانند نوشتن در ساعت سیستم، نوشتن در رجیسترهای کنترلر کننده های منابع، نوشتن در مود کاربری، نوشتن در vector interupt و موارد دیگری که تحت مدیریت os و سخت افزار است.

حال برگردیم به پرسش مان : پراسس ها چطور کارهایشان (کارهای مربوط به اجرای برنامه ها) را انجام می دهند

یک پراسس برای انجام کارهایش یک قطعه کد دارد، در این قطعه کد یکسری دستورات ان نوشته که جز دستورات معمولی است که براحتی توسط cpu اجرا می شود اما برای یکسری از کارها نیاز به دستورات ممتاز دارد که تحت اختیارش نیست بنابراین انجام آن کارها را باید از os بخواهد، بنابراین روال هایی برای انجام آن کارها در os در نظر گرفته شده و مشخص شده اند که برنامه ها می توانند آن روال ها را صدا بزنند و از آنها استفاده کنند، توجه کنید که فقط برخی از روالهای os از طریق برنامه ها قابل فراخوانی هستند

مثال : در که یک برنامه می توانیم روال هایی مانند open file، close file، rename file/folder، Delete file/folder و ... را صدا بزنیم تا os آن کار را برای ما انجام دهد.

* آن روال هایی که برنامه ها می توانند آنها را صدا بزنند Kernel services (خدمات هسته) یا system call (فراخوانی سیستم) یا supervisor call (svc) یا monitor call می گویند که یکی از انواع software interrupt است.

نکته مهم : ورود به داخل هسته os و استفاده از امکانات اش فقط با روش هایی که خودش مشخص کرده (طراحان os) امکان پذیر است، وقتی یک برنامه یک system call می زند دیگر آن برنامه هیچ اختیاری ندارد و اختیار دست os می افتد و os آن کار را برای آن برنامه انجام می دهد و نتیجه را به آن برنامه برمی گرداند، یک زندانی را تصور کنید که درخواست خرید یک کتاب را به

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

زندان بان می دهد، زندانی (برنامه) درخواست را از پنجره سلول اش به زندان بان (OS) می دهد، حال زندان بان آن کار را برایش انجام می دهد، برای انجام این کار زندان بان ممکن است افراد مختلفی را صدا کند (از چندین روال دیگر درون OS استفاده کند) که این موارد از دید زندانی پنهان است.

بنابراین وقتی یک برنامه یک system call را صدا می زند خود آن روال ممکن است روال های دیگر درون کرنل را صدا بزند

نکته) حالا متوجه شده اید که هسته OS چگونه از لایه های بالایی OS و برنامه های کاربران حفاظت شده است.

نکته) وقتی در بیرون هسته باشیم می گوییم در محیط کاربر (user mode) فعالیت می کنیم و زمانیکه درون هسته باشیم می گوییم کارها در محیط هسته (Kernel mode) فعالیت می کنیم، در محیط هسته همه دستورات از جمله دستورات ممتاز قابل اجرا است ولی در محیط کاربر مجاز به استفاده از دستورات ممتاز نیستیم.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File management	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device management	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communications	CreatePipe()	pipe()
	CreateFileMapping()	shm_open()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

در unix هم اجرا به شکل فایل دیده می شوند

- در شکل سمت چپ می توانید برخی از system call های unix و ویندوز را مشاهده کنید

- system call های ویندوز مخفی است و در ویندوز ما API داریم و API را صدا می زنیم، API توابع کتابخانه ای ویندوز است و سیستم کال نیست که توی کرنل باشه ما API رو صدا می زنیم و اینکار در نهایت منجر به فراخوانی های سیستمی میشه

اگر سیستم عامل open source باشد به هم sys call های آن آگاه هستیم اما اگر نباشد تمای sys call ها را نمی شناسیم (نیازی هم نداریم)

نکته: پس متوجه شدیم که یک دستور ساده text ممکن است تبدیل بشود به تعداد قابل توجهی روال درون کرنل.

نکته: متناسب با کاربرد و OS مربوط به آن تنوع سیستم کال ها می تواند خیلی زیاد باشد بطوریکه یک OS حتی می تواند بین ۲۰۰ تا ۳۰۰ سیستم کال داشته باشد

user running
→ open file
Kernel running
→ done
user running

سوال: اگر در یک لحظه بیش از یک system call رخ دهد چه می شود؟ در یک پردازنده single core فقط ۱ برنامه در لحظه جاری روی cpu اجرا می شود و فقط همان می تواند system call دهد بنابراین اصلا این امکان وجود ندارد

* همان طور که در بالا گفتیم فراخوان سیستمی یکی از انواع وقفه نرم افزاری (software interrupt) است که برخی فقط به این وقفه، وقفه نرم افزاری می گویند. ما در آینده در خصوص اینکه چطور یک OS متوجه وقایع خواهد شد بطور کامل صحبت خواهیم کرد، فعلا می خواهیم یک دسته از آن را توضیح دهیم:

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

API (Application programming interface)

حتی برنامه‌هایی ساده از os بطور گسترده استفاده می‌کنند و اغلب کامپیوترها هزاران system call را در ثانیه اجرا می‌کنند، با این حال اکثر برنامه نویسان هرگز این سطح از جزئیات را نمی بینند و بطور معمول برنامه‌ها را بر اساس API ها طراحی می‌کنند.

API مجموعه‌ای از توابع را مشخص می‌کند که برای برنامه نویسان برنامه‌ها در دسترس است و مشخص می‌کند که چه پارامترهایی به هریک از این توابع باید ارسال شود و همچنین مقادیری که برنامه نویس انتظار دارد را به آن return می‌کند (برمی گرداند)

- در os های مختلف سیستم کال های مختلف زیادی وجود دارد بنابراین وقتی یک برنامه برای یک os بنویسیم به سادگی نمی‌توان آن را روی os دیگری استفاده کرد برای حل این معضل API اختراع شد.
- هدف API این بود که استفاده از سیستم کال ها استاندارد شود بطوریکه اگر دو os از یک API پشتیبانی کند برنامه ها از یک os به os دیگر قابل انتقال خواهد بود

* داستان چیه؟ یک برنامه نویس میاد و برنامه‌ای که می‌خواد بنویسه رو با استفاده از API ـ x می نویسه، حالا هم os هایی که از API x پشتیبانی می‌کنند قابلیت اجرا این برنامه را دارد

↳ در واقع بین هم این os هایی که از API x استفاده می‌کنند ارسال و انتقال پارامترها و گرفتن return value از برنامه به os طبق استاندارد API x می‌شود

۳ مورد از API های موجود برای برنامه نویسان

۱- windows API برای سیستم های مبتنی بر ویندوز (win 32 API)

۲- posix API برای سیستم های مبتنی بر posix (portable operating system interface)

↳ تقریبا همه نسخه‌های unix, linux, macos

IEEE آمده system call های posix را استاندارد کرد.

۳- JAVA API برای برنامه‌هایی که روی java virtual machin

نکته) یک برنامه نویس به API ها از طریق کتابخانه‌ای از کدهای ارائه شده توسط os دسترسی پیدا می‌کند، بعنوان مثال در linux, unix برای برنامه‌هایی که به زبان c نوشته می‌شوند از کتابخانه libc استفاده می‌کنند

نکته) وقتی ورژن جدیدی از یک os داده می‌شود، مثلاً وقتی ویندوز ۱۱ منتشر می‌شود ممکن است لیست سیستم کال هایی که دارد نسبت به ورژن های قبل آپدیت شده باشد و برای ارائه قابلیت های جدید system call های جدیدی در os قرار داده باشند، بنابراین برنامه نویس اگر می‌خواهد با توجه به win 32 API برنامه اش را بنویسد نمی‌تواند از system call های جدید ویندوز ۱۱ استفاده کند و اگر برنامه نویس بخواهد از امکانات و سیستم کال های جدید ویندوز ۱۱ استفاده کند مجبور است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

خارج از استاندارد win32 API نیز کدنویسی کند که در اینصورت ممکن است این برنامه دیگر قابلیت اجرا روی نسخه های قبلی و بعدی ویندوز را نداشته باشد، بنابراین باید بین قابلیت حمل و یا امکانات جدید یک os جدید یکی را انتخاب کنیم

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

آدرس اشاره گر به بازی که فایل خوانده شده باید در آن نوشته شود

حداکثر چند بایت از این فایل خوانده شود

اسم فایلی که می خواهد بخواند

void pointer: اشاره گر عمومی
این اشاره گر از نوع void تعریف شده تا بتواند به هر نوع داده ای تعریف شوند

مزایای استفاده از API:

۱) قابلیت حمل برنامه (program portability): یک برنامه نویس که یک برنامه را با استفاده از API طراحی کرده است انتظار دارد برنامه اش روی هر سیستم دیگری که از همان API پشتیبانی می کند کامپایل و اجرا شود

۲) system call های واقعی اغلب جزئیات بیشتری دارند و کار کردن با آنها برای برنامه نویسان سخت تر از کار کردن با API ها در دسترس برنامه نویسان است.

ترجمه) با این وجود اغلب یک correlation قوی بین یک تابع درون API و system call مرتبط (یا آن تابع) درون کرنل وجود دارد در واقع بسیاری از API ها مثل POSIX و Windows شامل native system call هایی هستند که توسط سیستم عامل های unix و Linux, windows وجود دارد (مخفی شده است)

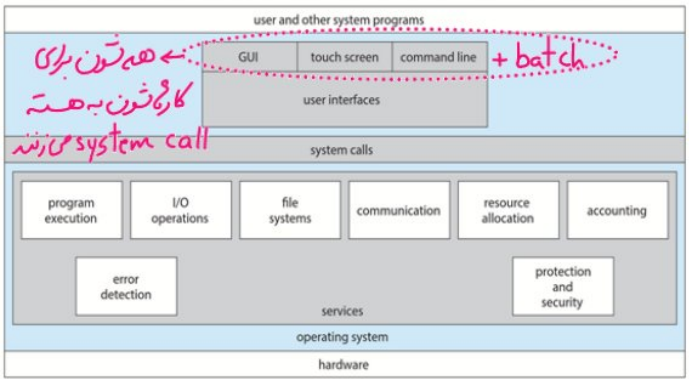
نکته) system call یک وقفه نرم افزاری است که برای فراخوانی آن یک دستور خاص (مثلاً INT) با شماره مربوط به آن system call توسط cpu باید اجرا شود، معمولاً اکثر cpu ها برای system call یک instruction خاص مثل INT یا Trap یا sys call دارند که شماره system call خود پارامتر این دستورات است و در اثر اجرا مثلاً دستور sys call ۳۵۰، روتین متناظر با سیستم کال شماره ۳۵۰ اجرا می شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

- گفتیم واسط بین os و برنامه‌ها و کاربران system call هستند که با استفاده از system call سرویس‌هایی که در زیر می‌بینید را می‌تواند دریافت کنند. همان‌طور که قبلا اشاره کردیم تعداد system call در یک os بسیار زیاد است (شلا ۴۰۰ تا) و همین طور کار کردن و استفاده از سیستم کال‌ها پیچیده است به همین دلیل یوزرها و برنامه‌ها بسادگی نمی‌توانند بصورت مستقیم با system call کار کنند و به همین دلیل رول system call و os یک واسط به نام user interface یا واسط کاربر دارند که به گونه‌های مختلف به کاربران و برنامه‌ها ارائه می‌شود واسط کاربری می‌تواند

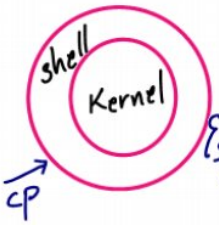


۱- گرافیکی باشد (GUI = Graphical user interface)
۲- text mode باشد: (CLI) command line interface

۲-۱: یک برنامه خاص، بعنوان مثال shell (شل یک command interpreter (مفسر فرمان) است)
۲-۲: پیاده سازی در داخل کرنل (in the Kernel)

۲-۱: توضیح نوع اول CLI: shell (پوسته)

در CLI ما یک نرم‌افزاری به نام shell در دور هسته داریم که این shell یک مفسر (interpreter = مترجم) و فرمان‌هایی که کاربران بصورت text بهش میدن به system call تبدیل می‌کند و به کرنل ارسال می‌کند



- مثلاً وقتی command ـ cp را به shell بدهید، shell آن را به تعدادی از system call تبدیل می‌کند که آن system call برای هسته (Kernel) قابل درک هستند و به واسطه اجرای پشت سر هم آن system call دستور cp انجام می‌شود (سرویس کپی ارائه می‌شود)

cp آدرس مقصد آدرس مبدا

ممکن است یک دستور ساده text mode به تعداد زیادی system call تبدیل شود، به همین علت چون shell یک برنامه مفسری است که در سطح کاربر است، تعداد زیادی سوئیچ بین مد یوزر و مد کرنل خواهیم داشت و تعداد زیاد این سوئیچ‌ها می‌توانند کارایی سیستم ما را کم کنند به همین دلیل برخی از دستورات پرکاربرد را به داخل هسته منتقل می‌کنند و اینها بصورت یک دستور اضافه کنار سیستم کال‌ها (در درون هسته) قرار می‌گیرند که در اینصورت هم این کارها در داخل خود کرنل و بصورت توابع درونی داخلی خودش انجام می‌شود و اینگونه کارایی سیستم بالا می‌رود.

توجه کنید که شل یک برنامه سطح کاربر است و مانند همه برنامه‌ها (مانند os و برنامه‌های کاربری) روی ram اجرا می‌شود، دستوری (command) که به شل داده می‌شود شل در نقش مفسری خودش ظاهر می‌شود و دستور را تفسیر و بررسی می‌کند و چک می‌کند که دستور و پارامترهایش درست یا غلط است و همین طور بررسی می‌کند که آیا شما مجوز استفاده از آن دستور را دارید یا خیر، اگر همه چی درست بود باید آن کارها را با استفاده از system call از کرنل بخواهد، توجه کنید که شل نه مجوز دسترسی به سخت افزارها رو داره و نه بلد هست که با اونها کار کنه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Example System Call Sequence

1. Acquire input file name
   Write prompt to screen
2. Accept input
3. Acquire output file name
   Write prompt to screen
4. Accept input
5. Open the input file
   if file doesn't exist, abort
6. Create output file
   if file exists, abort
7. Loop
   Read from input file
8. Write to output file
9. Until read fails
10. Close output file
11. Write completion message to screen
12. Terminate normally

مثال: فرض کنید قرار است فرمان cp را به shell ـ unix (یا linux) بدهیم

فرمان copy را تایپ می‌کنید و اینتر را می‌زنید، حالا یکسری اتفاقات و سیستم کال‌هایی پشت سر هم رخ می‌دهد که در باکس آبی می‌بینید

ابتدا با یک system call I/O روی صفحه درخواست گرفتن اسم فایل مبدا از یوزر داده می‌شود

پس یوزر اسم فایل مبدا را تایپ می‌کند و اینتر را می‌زند پس با یک system call ـ input اسم فایل مبدا را دریافت می‌کنید

پس یک system call I/O روی صفحه درخواست گرفتن اسم فایل مقصد از یوزر داده می‌شود

پس یوزر اسم فایل مقصد را تایپ می‌کند و اینتر را می‌زند پس با یک system call ـ input اسم فایل مقصد را دریافت می‌کند، این اعمال، اعمال I/O محسوب می‌شوند

پس برای باز کردن آن فایل از سیستم کال open استفاده می‌کنیم، حالا اگر فایل وجود نداشته باشد abortion اتفاق می‌افتد که به معنای ختم غیرعادی پراسس است که در حال اجرا شدن است

پس برای ایجاد فایل خروجی سیستم کال creat استفاده می‌شود (فراخوانی می‌شود)

پس در یک loop با سیستم کال Read بلاک به بلاک از فایل ورودی می‌خوانیم
و " " write " " " " در " خروجی می‌نویسیم

تعداد این سیستم کال‌های read و write به سایز فایل مبدا برمی‌گردد

و این کار را تا زمانیکه فایل ورودی تمام نشود ادامه می‌دهیم

بعد از اتمام کپی با سیستم کال close باید فایل خروجی را ببندیم

و پس با یک سیستم کال پیام خاتمه این کپی را روی صفحه نمایش دهیم

در نهایت پراسس را بصورت نرمال خاتمه یافته تلقی می‌کنیم

* برای یک copy ساده این همه سیستم کال داریم، حالا اگر یک دستور خیلی در os استفاده بشود به جای اینکه در شل پیاده سازی‌اش بکنیم که تعدد switch کردن از مد کاربر به مد هسته را به وجود می‌آورد، آن را با مدل in the Kernel پیاده کنیم

batch

در ویندوز فایل‌هایی با پسوند .bat، user interface از نوع batch را به وجود می‌آورند، آن دسته از دستوراتی که می‌خواهیم پشت سر هم و بصورت یک دسته از دستورات و بصورت یکباره برای اجرا به os داده بشوند درون فایل .bat ذخیره می‌شوند

نکته) در ابتدای بوت شدن os بسیاری از این فایل‌های .bat اجرا می‌شوند

نکته) یکسری دستورات کمکی مانند if و else و ... برای این روش وجود دارد که می‌توان با آن مشخص کرد که بسته به اینکه چه وضعیتی وجود دارد کدام دسته از دستورات اجرا بشوند

نکته) در os های unix base اگر c shell داشته باشیم (نوع شل مان c باشد)، فایل‌های batch دارای پسوند .cshrc است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

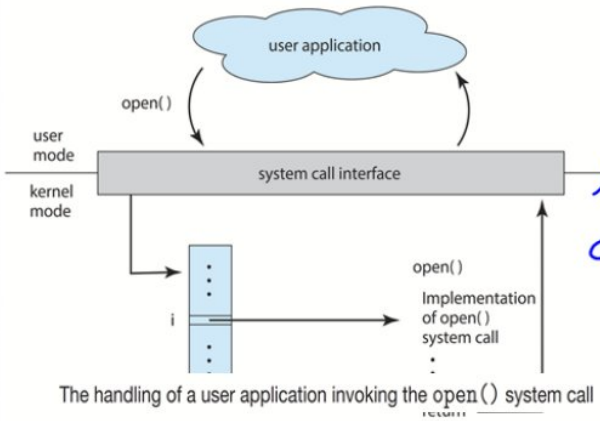
Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Handling a system call

وقتی یک دستور وقفه نرم افزاری می دهد (system call می دهد) چطور متوجه می شویم که کدام یک از system call ها فراخوانی شده

برنامه با یک وقفه نرم افزاری OS را مطلع می کند که کاری از OS می خواهد (کاری با OS دارد) و به این ترتیب از user mode به Kernel mode سوئیچ می شود

برای اینکار باید هم شماره سیستم کال و هم پارامتری که نیاز است به آن سیستم کال پاس بشود را از user mode به Kernel mode ارسال کنیم تا OS بتواند روتین متناظر با آن سیستم کال را در کرنل مد اجرا بکند و در انتها با دستور return مقدار بازگشتی را به فراخوانده برگرداند. مثلاً اگر از سیستم کال open استفاده می کنیم باید اسم فایلی که قرار است open بشود به OS اطلاع داده بشود



The handling of a user application invoking the open() system call

*خلاصه: برنامه باید به OS اطلاع بدهد که با سیستم کال شماره چند کار دارد و پارامترهای آن را نیز بهش بدهد

روش هایی که برای انتقال دادن پارامتر از یوزر مد به کرنل مد برای system call ها وجود دارد : ۳ روش

۱- Registers

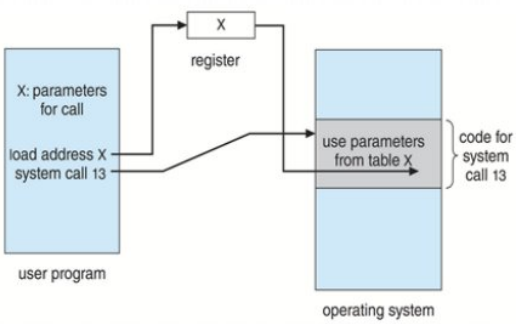
۲- Blocks or table in memory

۳- stack

توضیح ۱) تعدادی از رجیسترهای پردازنده را برای نوشتن پارامترها و شماره سیستم کال در نظر بگیریم. برای OS باید مشخص کنیم که شماره سیستم کال در کدام رجیستر قرار می گیرد و همین طور مشخص کنیم که برای هر سیستم کال کدام رجیسترهای پردازنده مخصوص پارامترهای ارسال شده توسط آن سیستم کال از یوزر مد به کرنل مد است

* خب مشخص است که چون سرعت ثبات های پردازنده زیاد است، سرعت این روش هم خوب است ولی مشخصا چون تعداد رجیسترهای پردازنده که می توانند برای این کار مورد استفاده قرار بگیرند محدود است از نظر فضایی که نیاز است با محدودیت روبه رو هستیم

توضیح روش ۲) اگر پارامترهای بیشتری نسبت به ثبات ها وجود داشته باشد عموماً یک بلاکی از حافظه را که داخل فضای حافظه خود پراسس هست را برای نوشتن پارامترها اختصاص می دهیم و آدرس شروع مربوط به آن فضای حافظه را از طریق یک رجیستر به OS منتقل می کنیم، حال وقتی system call فراخوانی شد در رجیستر مربوط آدرس شروع بلاکی از حافظه را داریم که OS به آن آدرس مراجعه می کند و برای اجرا کردن system call از محتواهای آن حافظه بعنوان پارامتر استفاده می کند



نکته) این روش از روش قبلی کندتر هست چون OS باید به حافظه پراسس دسترسی پیدا کند و حافظه کند است ولی مزیت اش این است که با محدودیت فضا مواجه نیستیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

توضیح روش ۳) همان طور که برای فراخوانی توابع در یک پراسس از پشته استفاده می‌کردیم و فراخوانده پارامترهایی که می‌خواست به فراخوانده شده ارسال کند را روی پشته قرار میداد و فراخوانده شده آنها را از پشته برمیداشت، حالا وقتی یک user process یک system call می‌زند، پارامترهای آن را در پشته os قرار می‌دهد و os برای اجرا system call مد نظر می‌تواند پارامترها یش را از رکوردی که باید پشته است استخراج کند

نکته) از نظر سرعت نسبت به استفاده از رجیسترها کندتر ولی از روش استفاده از یک بلاک از حافظه سریعتر است چون این پشته حافظه مربوط به os است که دسترسی به این حافظه برای os ساده‌تر هست از دسترسی به حافظه پراسس‌های دیگر.

همین طور توجه کنید که حافظه‌ای که برای پشته os در نظر می‌گیرند معمولاً حافظه کوچکی است بنابراین از نظر حافظه وقتا نه به اندازه روش اول با محدودیت فضا مواجه هست و نه به اندازه روش دوم فضا کافی در اختیار دارد

through mode bit = Dual mode operation

از یک نسلی به بعد در سخت افزارها امکان وجود بیش از یک مد عملیات برای cpu تعبیه شد (حداقل ۲ تا مد)، بدین منظور یک بیت درون سخت افزار به نام mode bit قرار دادند، به این ترتیب cpu می‌تواند در ۲ مد با اختیارات مختلف قرار داشته باشد

۱- user mode: اگر mode bit=1 باشد در این مد هستیم، معمولاً برنامه‌ها در این مد اجرا می‌شوند

۲- privilege mode = system mode = supervisor mode = Kernel mode: اگر مد بیت ۰ باشد، معمولاً کرنل در این مد اجرا می‌شود

دستورات ممتاز (privilege instruction): دستوراتی که اجرا آنها در انحصار os است

*۱- تمامی عملیات و دستورات I/O در انحصار os است: یعنی هیچ برنامه‌ای بدون اجازه os نمی‌تواند عملیات I/O انجام دهد

۲- دستورات مربوط به تنظیمات تایمر (Timer)

نکته) مشخصا خواندن از تایمر در مد کاربر مشکلی ندارد و آن چیزی که مشکل دارد نوشتن در آن است.

۳- " " " تغییر mode bit: اگر این دستور در انحصار os نباشد خب یک برنامه mode bit اش را ۰ می‌کند و میره توی کرنل مود اجرا شود

۴- دستورات مربوط به تغییر اشاره‌گرهای ISR درون interrupt vector table: اگر این دستور در انحصار os نباشد یک برنامه مخرب می‌تواند IVT را دستکاری کند و بگوید اگر وقفه x رخداد برو ISR را از اینجایی که من می‌گم بردار و اجرا کن و بعد برنامه مخرب اش را در آن آدرس بگذارد و چون وقفه‌ها همشون در مد کرنل اجرا می‌شوند، پس آن برنامه مخرب می‌تواند در مد کرنل تمام اختیارات سیستم را در اختیارش بگیرد

۵- دستورات از کار انداختن و راه اندازی وقفه‌ها

۶- عدم دسترسی به رجیسترهای MMU (memory management unit: واحد مدیریت حافظه): اگر برنامه (کاربر) بتواند Base و limit خودش را تغییر دهد می‌تواند کل حافظه را تحت اختیار خودش بگیرد

۷- تغییر اولویت فرآیندها در مد کاربر مجاز نیست: برنامه (کاربر) بالاترین اولویت را به خودش می‌دهد و فرآیندهای دیگر اجرا نخواهند شد

نکته) هر چه که در مد کاربر مجاز است مشخصا در مد کرنل نیز مجاز است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

OS چطور متوجه وقایع می‌شود و وارد عمل می‌شود؟

چند مثال از وقایع و اتفاقات: فشار دادن یک دکمه کیبورد، ماوس را تکان می‌دهید یا کلیک می‌کنید، یک برنامه را اجرا می‌کنید، کاغذ پرینتر تمام، کاغذ پرینتر تمام می‌شود، روی دکمه close یک برنامه یا پنجره کلیک می‌کنید

اینها اتفاقاتی هستند که باید توسط OS مدیریت شود، حال OS باید متوجه این رخدادها شود

داریم وقایع را با سرکشی مطلع می‌شویم

۱- polling (سرکشی): OS در زمان‌های مشخص به Device ها مختلف سر می‌زند و بررسی می‌کند که آیا برای هیچ کدوم شون اتفاق جدید افتاده یا نه

polling کی مفید است؟ در مواردی که ما در مورد رخداد یکسری اتفاقات و نظم آنها اطلاع داریم، اگر این نظم را می‌شناسیم polling از وقفه ارجحتر است، زیرا عواملی که می‌توانند باعث رخدادها بشوند متعدد هستند و اگر قرار باشد با وقفه کار کنیم هم اینها می‌آیند و پایه وقفه cpu را فعال می‌کنند، ما در cpu یک پایه وقفه داریم و تعداد زیادی Device به این پایه متصل هستند، بنابراین اگر یک وقفه از صادر شود یک زمانی نیاز داریم تا بگردیم و ببینیم عامل آن وقفه چه کسی بوده است و این دارد به ما یک سرباری تحمیل می‌کند مثلاً اگر بدانیم کارت شبکه هر ۳۰ ثانیه یکبار با ما کار دارد و پرینتر هر ۲۰ دقیقه یکبار با ما کار دارد، OS می‌تواند هر ۳۰ ثانیه یکبار به کارت شبکه سر بزند و مطمئن باشد که متوجه اتفاق جدی می‌شود ولی به پرینتر هر ۳۰ ثانیه سر نمی‌زند و هر ۲۰ دقیقه یکبار به پرینتر سر می‌زند و به این ترتیب دارد بهتر از منابع سیستم استفاده می‌کند

سوال: آیا polling فقط برای سخت افزارهاست یا برای نرم افزارها هم داریم؟ بله، یک نرم افزار می‌تواند یک فلگ را set کند و با ست کردن آن از OS کاری را بخواهد و OS هم در زمان‌های مشخص می‌تواند آن فلگ را چک کند و در صورت ست بودن به آن سرویس دهد.

۲- interrupt (وقفه) OS به هیچ جا سر نمی‌زند و منتظر می‌ماند تا هر اتفاقی بهش اطلاع داده بشود، سیستم عامل‌های جدید interrupt driven (مبتنی بر وقفه) هستند

* OS یک لایه است بین برنامه‌های کاربردی و سخت افزار و همین طور سخت افزار (منابع)، و OS نقش مدیریتی دارد، یک نقش مدیریتی برای لایه بالا که قرار است (باید) برنامه‌های کاربردی را خوب مدیریت کند و یک نقش مدیریتی برای لایه پایین که قرار است (باید) منابع را خوب مدیریت بکند، به این ترتیب لایه میانی OS که نقش مدیر را دارد هم از لایه پایین و هم از لایه بالا مراجعه کننده دارد

* پس ما ۲ نوع مراجعه کننده به OS داریم: ۱- یکی از لایه software ۲- یکی از لایه Hardware

نکته: در سیستم عامل‌های مبتنی بر وقفه (Interrupt Driven) هر دو این مراجعه کننده‌ها از طریق وقفه مراجعه می‌کنند، بر همین اساس ما ۲ نوع وقفه داریم ۱- وقفه نرم افزاری ۲- وقفه سخت افزاری

توجه: در اثر رخداد وقفه چه وقفه نرم افزاری باشد و چه سخت افزاری یک ISR در کرنل سیستم عامل (Kernel mode) اجرا خواهد شد

Hardware interrupt: cpu یک پایه وقفه دارد (معمولاً به اسم INT) و Device ها و منابع مختلف (ماوس و کیبورد و ...) پایان عمل I/O خود را

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد. استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

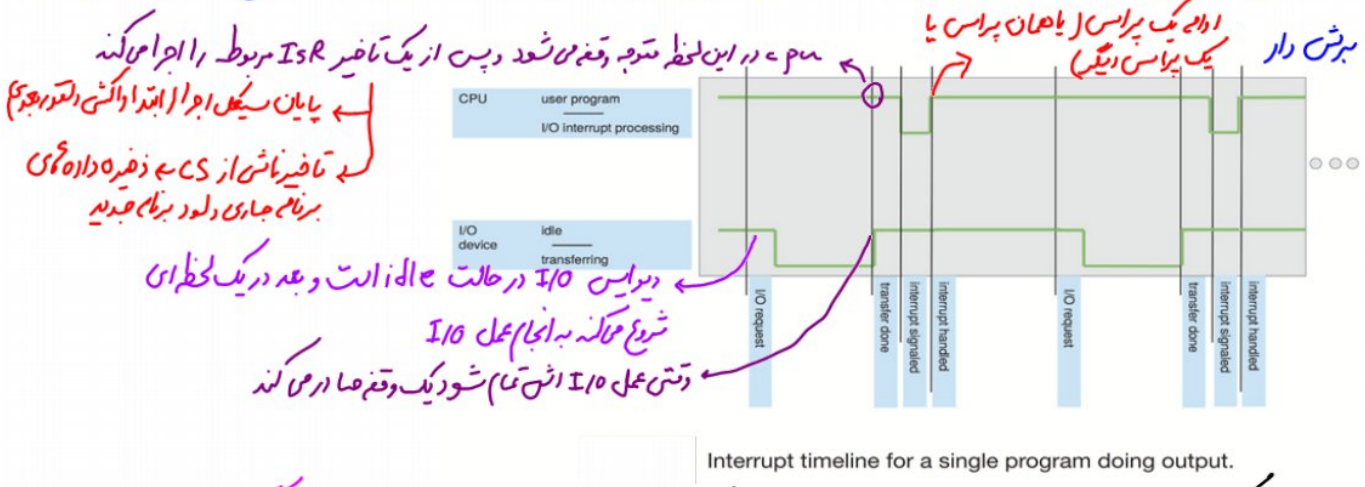
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

از طریق گذاشتن یک سیگنال روی پایه وقفه cpu، cpu را از پایان کار خودش مطلع می‌کند (می‌گوییم دیوایس‌ها می‌توانند به cpu وقفه بدهند) و cpu برای هندل کردن و پاسخ دادن به هر وقفه یک روال (روتین) به اسم ISR اجرا می‌کند

توجه مهم: برخلاف تصور اینکه وقفه دارد آغاز عملیات I/O را نشان می‌دهد، وقفه **پایان** عملیات I/O را به cpu اطلاع می‌دهد.

بعنوان مثال شما هنوز دکمه کیبورد را فشار ندادید یا ماوس را تکان ندادید یا بسته‌ای به کارت شبکه نرسیده کامپیوتر که علم غیب نداره بدونه در آینده قراره چه اتفاقی بیفته و قبل از اینکه این موارد رخ بده که وقفه صادر نمیشه و بعد از اینکه یکی از این کارها را انجام دادید و یک داده‌ای (کاراکتر برای کیبورد، مختصات برای ماوس و پکت داده شبکه) رفت توی بافر خودش یک وقفه صادر می‌شود و به OS اطلاع داده می‌شود یک داده‌ای آمده و بیا برش دار



Interrupt timeline for a single program doing output.

نکته *) مشخص است که OS فقط زمانهایی روی cpu اجرا می‌شود که کار مدیریتی وجود داشته باشد ← اگر اتفاقی در لایه نرم افزار بافت افتاد بیفتد که به واسطه وقفه به OS خبر داده شود OS فعال خواهد شد تا آن رخداد را مدیریت کند، با اجرا ISR مربوط واکنش مدیریتی نشان می‌دهد

Interrupt vector table (جدول بردار وقفه)

Interrupt handler: وقتی وقفه‌ای صادر می‌شود cpu باید به ISR مربوط به آن منتقل شود حال یک روش ساده برای مدیریت این انتقال (که البته سریع نیست و بنابراین دیگر کمتر استفاده می‌شود) فراخوانی یک روال عمومی است که اطلاعات وقفه را بررسی می‌کند و متوجه می‌شود چه interrupt-specific handler ای را باید فراخوانی کند، خب این کنده

خب برای افزایش سرعت باید آن روال میانی را از بین ببریم برای این کار میان در یک جدول بنام interrupt vector table که معمولاً در صدهٔ مکان اول حافظه قرار داده می‌شود آدرس (اشاره‌گر) ISR ای مربوط به هر Device را ذخیره می‌کنند، حال یک Device ای که یک وقفه ای صادر می‌کند و پایه وقفه cpu را فعال می‌کند یک شناسه‌ای از خودش را روی باس قرار می‌دهد، پس آن شناسه از روی باس برداشته می‌شود و نگاشت می‌شود به آدرسی از جدول IVT پس این جدول مشخص می‌کند که به کجای حافظه بپریم که ISR متناظر با آن Device اجرا شود.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نگاه دیگری به وقفه سخت افزاری و ماجرای آن:

در یک کامپیوتر یک cpu (پردازشگر هم نظیره) و ده‌ها I/O processor (پردازشگر ورود/خروج) وجود دارد، وقت cpu با ارزش است بنابراین کارهای I/O را به پردازشگرهای ورود و خروج می سپارد، بنابراین از نظر سخت افزاری یک نگاه Master-slave را می‌توان در کامپیوترها تصور کرد که در آن cpu یا همان Master کارهای مربوط به I/O را به دستگاه‌های ورودی و خروجی (slave) واگذار می‌کند، حال برای اینکه Master متوجه شود که هر slave چه موقع کارش را تمام می‌کند ۲ روش polling و interrupt وجود دارد، در روش وقفه Device های I/O (slave) پایان ماموریت واگذار شده به آنها را به پردازشگر اصلی اعلام می‌کنند.

نکته) اینکه با هر دستگاه ورودی و خروجی بصورت سرکشی برخورد کنیم یا وقفه به Driver هر دستگاه I/O بستگی دارد چون برخی از Driver ها از روش سرکشی و برخی از روش وقفه پیروی می‌کنند

انواع وقفه‌های سخت افزاری

۱- وقفه‌های مربوط به دیوایس های I/O (وقفه‌های I/O): وقتی کار دستگاه I/O تمام شد کنترلر کننده آن دستگاه از طریق ارسال وقفه (ارسال یک سیگنال به cpu) به درایور آن دستگاه اطلاع می دهد که کار خود را به پایان رسانده است

۲- وقفه Timer یا Real time clock یا interval timer: ۱- تعریف پراسس‌ها ۲- زدودن RAM و ...

۳- " ریست کردن کامپیوتر

۴- Hardware fault یا Machine check (خطای ماشین): در صورت خرابی در سخت افزار. در کامپیوترها مدارات سخت افزاری متعددی در سرتاسر کامپیوتر وجود دارند که بر کار مدارات دیگر نظارت می‌کنند و در صورت بروز خطا به cpu وقفه می‌دهند

Masking interrupt: در داخل برنامه‌ها می‌توان فرمان نادیده گرفتن برخی از وقفه‌های اجازه داده شده را درج کرد، مثلاً یک برنامه نویس می‌خواهد بداند اگر وقفه سریز را در برنامه اش نادیده بگیرد چه اتفاقی می‌افتد. توجه کنید وقفه‌هایی وجود دارند که برنامه نویسان (یا کاربران) نمی‌توانند فرمان نادیده گرفتن آنها را بدهند مانند وقفه‌های ناشی از خطای سخت افزاری. توجه کنید که کرنل می‌تواند هر وقفه‌ای را نادیده بگیرد، حتی کرنل می‌تواند دستور معادل Disable interrupt را اجرا کند که باعث نادیده گرفتن کلیه وقفه‌ها می‌شود

وقفه‌های نرم افزاری (software interrupt)

۱- system call

۲- exeption

۳- trap

۱- قبلاً system call را توضیح دادیم، در اینجا مجدداً به نحو دیگری می‌خواهیم آن را توضیح دهیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

اصلا شاید os با درخواست‌های شما موافقت نکند

سوال: اصلا چرا نرم افزارها (برنامه‌های کاربری) باید وقفه دهند؟

اگر وقتی یک برنامه کاربری مثلاً بخواهد یک چیزی روی صفحه نمایش نشان دهد یا یک صدایی در بلندگو ایجاد کند یا یک چیزی را پرینت کند یا یک حافظه‌ای بخواهد اگر خودش بتواند این کار را انجام دهد که یک هرج و مرج و بی نظمی زیادی در سیستم به وجود می‌آید، بنابراین یک برنامه نباید بتواند خودش مستقیماً این کارها را انجام دهد، برای همین باید مدیریت سیستم به عهده سیستم عامل باشد، اگر هر برنامه‌ای خودش مستقیما بتواند کارهایی که گفتیم را انجام دهد دیگه نقش مدیریتی os از بین می‌رود.

پس تنها کاری که یک برنامه کاربردی می‌تواند انجام دهد این است که به os درخواست بدهد، os یک گوشه ایستاده (روی cpu اجرا نمی‌شود) و منتظر درخواست‌های برنامه‌هاست، برنامه‌ها با وقفه نرم افزاری خواسته‌هایشان را از os طلب می‌کنند که به این نوع فراخوانی os، سیستم کال گویند

نکته) وقفه نرم افزاری اجرا یک instruction خاص توسط برنامه کاربردی است (رخ می‌دهد)

نکته) " " یک logic داخل cpu را فعال می‌کند و فعال شدن این لاجیک باعث می‌شود که cpu به os اطلاع دهد که یک برنامه‌ای با آن کار دارد، در واقع cpu که IsR مربوط به آن وقفه که در درون os است و باعث انجام روال syscall مربوطه می‌شود را اجرا می‌کند.

۲- exception (استثنا)

اگر برنامه در حال اجرا کار خلافی انجام دهد مکانیزم‌هایی سخت افزاری در سیستم تعبیه شده که باعث فعال شدن نوعی از وقفه به اسم exception می‌شود، در این شرایط os مطلع می‌شود که یک کار غیر مجازی اتفاق افتاده و میره با اجرا IsR متناظر با آن استثنا exception handeling انجام می‌دهد، بطور خلاصه وقفه استثنا در مقابل کارهای غیر مجاز و نادرست داخل cpu فعال می‌شود

- در صورت رخداد وقفه سخت افزاری یا نرم افزاری (system call) پس از قطع برنامه می‌رفتیم و IsR مربوطه را انجام می‌دادیم و پس در نهایت برمی‌گشتیم و اجرا برنامه را ادامه می‌دادیم ولی در exception دیگه احتمالا بعد از اجرا IsR به ادامه اجرا برنامه برنمی‌گردیم و برنامه terminate می‌شود

مثال‌هایی از exception: مثلاً os برای یک پراسس مشخص کرده که حدود حافظه‌اش از ۲۰۰ تا ۵۰۰ است ولی پوینتر برنامه دچار مشکل می‌شود و به آدرس ۷۰۰ مراجعه می‌کند، در این شرایط چون از حدود حافظه‌ای که برایش مشخص شده خارج شده مکانیزم سخت افزاری تعبیه شده برای مطلع شدن این موضوع وقفه مربوطه را صادر می‌کند (به خاطر دسترسی نامناسب به حافظه در برنامه‌ای که نوشتیم باعث شویم segmentation fault رخ دهد)

- یک نمونه exception دیگر خطای تقسیم بر ۰ است (Division by zero): وقتی یک instruction تقسیم داشته باشیم و اگر مخرج کسر ۰ باشد داخل cpu یک فلگ‌هایی فعال می‌شود که باعث رخداد یک exception می‌شود

- خطای سرریز و سه موارد دیگری از exception هستند

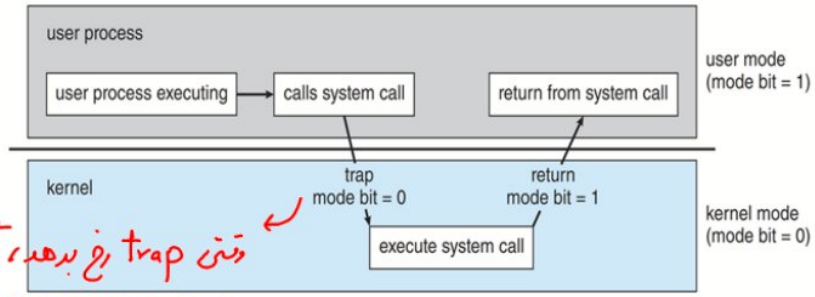
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) exception نسبت به وقفه سخت افزاری و وقفه نرم افزاری اولویت دارد، یعنی اگر هر سه نوع این وقفه‌ها را داشته باشیم اول به وقفه exception پاسخ داده می‌شود

trap_۳: گفتیم که دوتا مد داریم در سیستم و پردازنده می‌تواند در ۲ مد با اختیارات مختلف قرار داشته باشد، درکرنل مد تمامی دستورات را می‌توان اجرا کرد و هم اختیارات وجود دارد ولی در user mode اختیارات محدود است و برنامه‌ها دستورات ممتاز را نمی‌توانند اجرا کنند حال اگر برنامه‌ای بخواهد inst ای را اجرا کند که جز inst های ممتاز است (جز inst های محدود شده است در این شرایط یک وقفه نرم افزاری رخ می‌دهد که به آن trap می‌گویند، دلیل این نام گذاری آن است که شما (برنامه) با انجام چنین کاری در تله os گرفتار می‌شود پس با اجرا یک دستور غیرمجاز یک وقفه نرم افزاری رخ می‌دهد و trap می‌شود به os و یک ISR که قرار است به این trap رسیدگی کند (پاسخ بدهد) فعال می‌شود، در این حالت ممکن است os تصمیم بگیرد مثلاً برنامه شما را terminate کند (فرض کنید برنامه بخواهد یک عمل مدیریتی که در اختیار os است را اجرا کند)



Transition from user to kernel mode.

وقتی trap رخ بدهد، mode bit صفر می‌شود و کنترل می‌افتد دست os و os می‌رود و ISR مربوطه را اجرا می‌کند

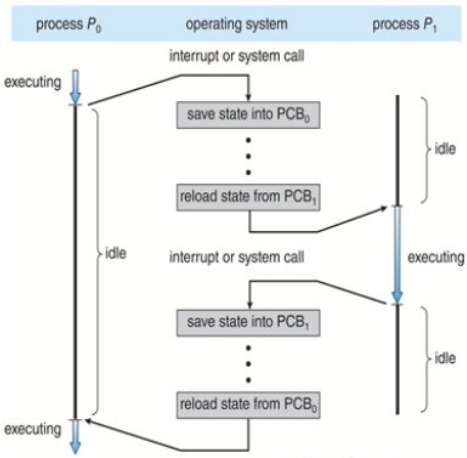
نکته: برخی از مراجع مثل تنباوم و سیلبرشاتز هر وقفه نرم افزاری که رخ دهد می‌گویند trap رخ داده (یا برخی انگلونه می‌گویند که همه وقفه‌های نرم افزاری تله هستند)، حال دلیل این صحبت چیست؟

یک برنامه دارد روی cpu اجرا می‌شود حالا این برنامه یا یک کار مجاز با os داره یا عمدا یک inst ممتاز اجرا می‌کند، در حالت اول که برنامه می‌خواهد یک کاری انجام دهد که آن کار جز اختیارات os است، برنامه خودش درخواست گیر افتادن در دام os را توسط system call می‌دهید و این یک درخواست مجاز است که os می‌آید و به آن درخواست رسیدگی می‌کند و اگر برنامه بخواهد کار غیرمجاز انجام دهد در دام os گرفتار می‌شود

نتیجه: چه system call بزنید چه exception رخ دهد چه trap در هر صورت در دام os گرفتار می‌شوید، حالا یا خودتون درخواست گیر افتادن را می‌دهید (sys call) یا کار غیرمجازی رخ می‌دهد یا انجام می‌دهید که باز در دام os گیر می‌افتید (os همیشه یک گوشه نشسته و تا رخدادی رخ ندهد نمیاد وسط)

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



نکته: گفتیم که عمل context switch سربارهای زیر را دارد

۱- زمان ذخیره کردن وضعیت یک پراسس و بازیابی وضعیت پراسس جدید

۲- زمان لازم برای انتخاب پراسس بعدی برای اجرا توسط سیستم عامل

نکته: در برخی از سخت افزارها برای کاهش سربار context switch کمک شود در پردازنده چندین Register set در نظر می‌گیرند (منابع بیشتری در cpu می‌گذارند) بعد به هر پراسس یک Register set نسبت می‌دهند (Register seti) و آدرس وضعیت هر یک از پراسس‌های سیستم در Register set مربوط به خودش قرار دارد، حالا وقتی قرار است یک پراسس دیگر بورد روی cpu فقط کافی است که برای cpu مشخص کنیم که Register set شماره چند فعال است، بنابراین دیگر نیازی نیست برای ذخیره و بازیابی PCB ها سرانج حافظه برویم و بنابراین سرعت context switch افزایش می‌یابد

ترجمه: توجه کنید که پیاده‌سازی‌های مختلفی وجود دارد و چیزی که گفته شد فقط ایده کلی بود

مثال: Sun UltraSPARC نمونه‌ای است که از این روش استفاده کرده

Queue

اگر توجه کرده باشید ما هم جا صف داشتیم و OS باید تصمیم بگیرد که در هر صف به چه کسی اولویت دهد

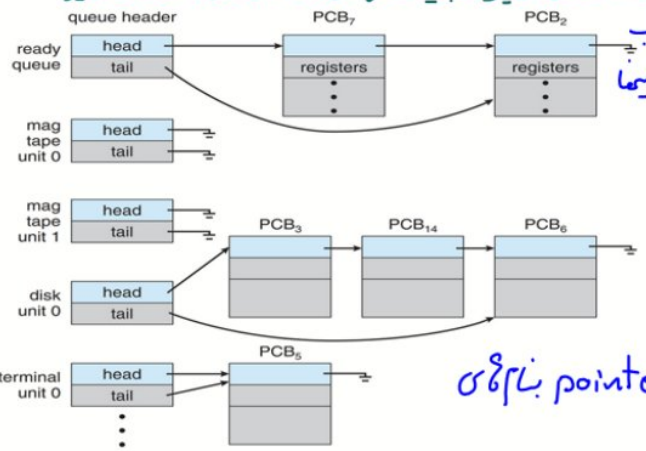
۱- صف job ها یی که می‌خواهند وارد سیستم بشوند، در واقع درخواست دارند که به پراسس تبدیل بشوند

۲- " ready

۳- " Block ← Block for Disk، Block for printer

۴- " wait ← wait برای اینکه یک حادثه‌ای رخ بدهد، مثلاً می‌تواند منتظر یک پیام از فرآیند دیگر بشود

۵- " I/O Device ← I/O Device : دستگاه‌های I/O خیلی کندتر از cpu هستند پس صف شکل می‌گیره و باید منتظر شوند تا I/O در دسترس قرار بگیره



آن I/O Device ها یی که پاراسته سرعت برایشون مطرح است و پراسور محسوب می‌شوند بحث صف و زمانبندی برایشون مطرح می‌شود، پس برای هر یک از اینها یک scheduler داریم، مثل:

Disk scheduler
printerscheduler
⋮

Disk scheduler تصمیم میگیره ابتدا عمل I/O کدوم پراسس را انجام بدهد

* صف‌ها بصورت لیست پیوندی (linked list) پیاده‌سازی می‌شوند که دو pointer بنام‌های

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

head و tail در ابتدای این لیست پیوندی وجود دارد که ابتدا و انتهای صف مربوطه اش را مشخص می کند، همان طور که مشخص است در این لیست پیوندی pcb پراسس هایی مربوطه هست، هر pcb نیز شامل یک فیلد اشاره گر است که به pcb بعدی اشاره می کند

* مشخص است که این لیست های پیوندی بطور پیوسته توسط scheduler مربوط به آن صف (OS) آپدیت می شود و در واقع pcb ها بین صف های مختلف جابجا می شوند، آپدیت شدن این صف ها بسیار راحت است و فقط با آپدیت فیلد Next هر pcb شدنی است

نکته) البته می توانیم لیست های پیوندی را بصورت ۲ طرفه پیاده سازی کنیم که در اینصورت هر pcb شامل ۲ فیلد اشاره گر است، یکی برای اشاره به pcb بعدی و یکی برای اشاره به pcb قبلی.

نکته) ۲ متغیر Head یا front و tail یا rear که برای هر صف وجود دارد در فضا OS و در جای جداگانه ای از جای قرارگیری PT و pcb قرار می گیرد.

توجه) توجه کنید که ما در اینجا یک core داریم و همین طور فعلا در بحث پراسس ها تک نخی (single thread) هستیم، بنابراین در عکس بالا یک پراسس نمی تواند همزمان در ۲ صف قرار داشته باشد

مثلاً $P_۲$ در ready است یعنی آماده است که cpu بهش داده بشه پس نمی تونه در صف های Block و I/O Device wait باشه

هر پراسس که کارش در صف مربوطه اش تمام شد، بعنوان مثال فرض کنید $P_{۱۲}$ عملیات I/O مربوطه اش تمام می شود و پایان آن با وقفه به cpu اعلام می شود، در این شرایط تغییرات زیر در این لیست های پیوندی صورت می گیرد

۱- $P_{۱۲}$ به ready queue اضافه شود ← اینکه چگونه اضافه شود بسته به نحوه پیاده سازی دارد

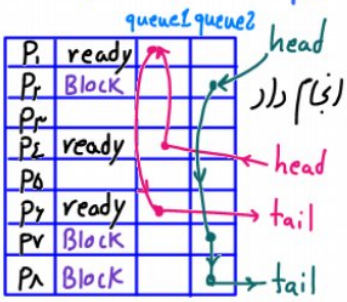
۲- صف آن که $P_{۱۲}$ در آن قرار داشته آپدیت شود

براساس آنکه انتخاب پراسس ها با اولویت بالاتر در O(۱) انجام شود یا در O(n) به ۲ طریق می توان هرکدام از این صف ها را پیاده سازی کرد

(Remove)

۱- انتخاب پراسس در O(n) باشد ← در این حالت هر پراسس که می خواهیم به یک صف اضافه کنیم چون tail رو داریم خیلی راحت می ریم و pcb رو میذاریم آخر صف مربوطه اش، در اینصورت اولویت pcb ها درهم و برای انتخاب پراسس با بالاترین اولویت باید لیست مربوطه را پیمایش کنیم

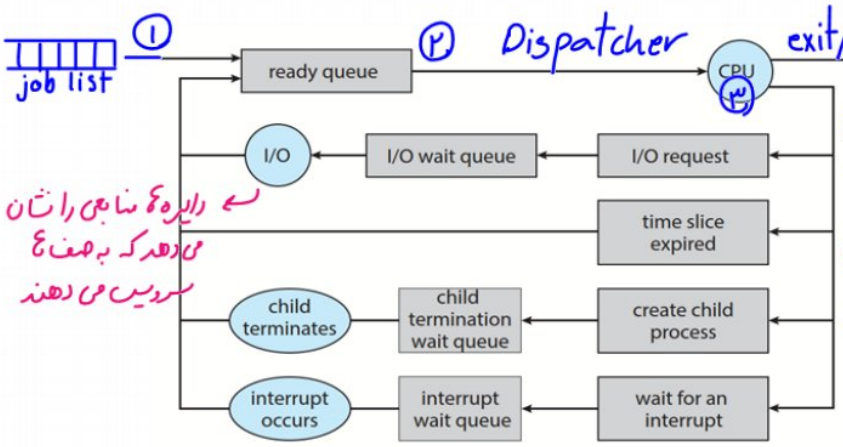
۲- انتخاب پراسس در O(۱) باشد ← در این حالت درج pcb مربوط به هر پراسس در O(n) انجام می شود، pcb ها را براساس اولویت شان در موقع درج در جای درست شان قرار دهیم، در اینصورت می توانیم pcb مربوط به پراسس با بالاترین اولویت را همیشه در ابتدای صف قرار دهیم



نکته) برای پیاده سازی صف ها می توان به جای استفاده از pcb این کار را در page table انجام داد به اینصورت در داخل page table می توانیم چند ده صف خود را تشکیل دهیم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



Queueing-diagram representation of process scheduling.

۱- انتخاب توسط job scheduler و ایجاد فرآیند جدید، یک job جدید از job list انتخاب و اگر مجاز دانسته شد که به پراسس تبدیل بشود این کار انجام می‌شود و وارد ready queue می‌شود

۲- cpu scheduler : وقتی فرآیندی که در حال اجرا روی cpu است cpu را رها می‌کند Dispatcher که وظیفه‌اش توزیع cpu بین پراسس‌های آماده است فرآیندی را از ready queue انتخاب می‌کند و آن پراسس روی cpu شروع به اجرا می‌کند و به حالت running می‌رود

دایره‌ها منابعی را نشان می‌دهد که به صف‌ها سرویس می‌دهند

۳- اگر فرآیند تمام (exit) نشود (بصورت نرمال یا با خطا که در ۲ اختیاری است و توسط فرآیند انجام می‌شود) یا Kill نشود (یا خطا کرده که Kill شده یا به علت Kill شدن پدرش Kill می‌شود و یا کاربر Kill اش می‌کند) یکی از ۴ اتفاق زیر می‌تواند برایش بیفتد

۳-۱: I/O request : فرآیند به واسطه یک system call درخواست I/O (I/O request) می‌دهد و به حالت Block می‌رود (در انتظار دریافت خدمت آن دستگاه I/O بلوکه می‌شود)، سپس این درخواست در صف I/O دستگاه مربوطه قرار می‌گیرد و سپس وقتی توسط I/O scheduler مربوط به آن دستگاه انتخاب شد و پردازش I/O مربوط به آن انجام شد با یک وقفه انتهای عمل I/O به cpu اطلاع داده می‌شود و در نتیجه پراسس از حالت Block خارج می‌شود (wake up می‌شود) و به صف ready برمی‌گردد

۳-۲: time slice expired : وقفه پایان سهم زمانی پراسس توسط تایمری که os قبلا آن را set کرده صادر بشود که در این حالت پراسس بصورت مستقیم به ready queue برمی‌گردد (timer interrupt)

۳-۳: یک پراسس، پراسس جدیدی به وجود آورد : در سیستم‌های مبتنی بر unix که از posix API استفاده می‌کنند می‌توانیم با استفاده از system call ـ fork توسط یک پراسس، یک پراسس فرزند ایجاد کنیم که در این حالت یکی از حالت‌های پیاده سازی به این صورت است که پراسس والد صبر می‌کند (به حالت wait می‌رود = می‌خوابد) تا زمانیکه پراسس فرزند کارش کاملا تمام شود و بمیرد (terminate شود) و بعد از آن والد از حالت wait خارج می‌شود (از خواب بیدارش می‌کنیم) و به صف ready منتقل می‌شود

۳-۴: wait for an interrupt : یک پراسس برای رخداد یک رویداد خاص wait می‌کند، مثلاً با استفاده از دستور sleep می‌کند برای سپری شدن یک میزان زمان مشخص، مثلاً sleep(۲۰ms) را اجرا کند، حالا وقتی این دستور اجرا می‌شود ۲۰ms بعد وقفه پایان اش صادر می‌شود و در نتیجه این پراسس از حالت wait به صف ready برمی‌گردد

مثال دیگر: فرآیند روی صفحه چاپ می‌کند اگر میخواهد کار x را برای انجام بدهد کلید ۱ و اگر کار y روی خواهد کلید ۲ (read key)

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Batch systems (سیستم‌های دسته‌ای/بسته‌ای) : اغلب در یک سیستم دسته‌ای فرآیندهای بیشتری از آنچه می‌توان بلافاصله اجرا کرد ارائه می‌شود، این فرآیندها در یک mass-storage device که عمدتا دیسک است spool می‌شوند و زمانبند بلندمدت (job scheduler) فرآیندها را از این pool انتخاب می‌کند و آن‌ها را برای اجرا در حافظه load می‌کند

این سیستم‌ها برخلاف سیستم‌های interactive به محض اینکه هر کاری وجود داشته باشد به آن واکنش نشان نمی‌دهند و کارها را جمع می‌کنند تا به یک دسته برسد و سپس به یکباره به همه آنها پاسخ می‌دهند.

یکی از کاربردهای سیستم‌های دسته‌ای در mainframe اینگونه بود که معمولاً یک پرینتر وجود داشت و تعداد زیادی کاربر وجود داشت و اینطوری نبود که هر یوزری پرینت‌اش را برای پرینتر ارسال کند و بلافاصله بتواند پرینت‌اش را بردارد و اینگونه بود که یوزرها درخواست‌هایشان را می‌دادند و وقتی این درخواست‌ها به یک مقدار مشخصی می‌رسید یا به یک زمان مشخصی که می‌رسیدیم بصورت دسته‌ای همه اینها پرینت می‌شد.

کاربر امروزی: طراح os ممکن است با توجه اهدافی که دارد و با توجه به منابع، کارها و ماهیت کارها تصمیم بگیرد به محض انجام برخی از درخواست‌ها به آن‌ها پاسخ ندهد و وقتی آن درخواست‌ها به یک حد نصاب مشخصی رسید و یک دسته را به وجود آورد بعد به آنها پاسخ بدهیم

مثال) گوگل در سیستم عامل اندروید از این ایده برای کنترل مصرف باتری و جلوگیری از روشن ماندن همیشگی ماژول wifi گوشی از Batch system برای ارسال بسته‌های شبکه استفاده می‌کند، چون اگر بخواهد به محض دریافت هر بسته از هر برنامه آن بسته را ارسال کند ماژول wifi همه‌اش داره فعالیت میکند و در نتیجه مصرف باتری را بالا خواهد برد، بنابراین os بسته‌هایی که فوریت ندارند را در یک بافر تحویل می‌گیرد و وقتی به یک حد مشخصی رسید و یا در زمان‌های مشخص (مدیریت‌های گوناگون و ترکیبی وجود دارد) این بسته‌ها را ارسال می‌کند

type of schedulers

یک فرآیند در طول عمر خود در بین صف‌های زمانبندی مختلف مهاجرت می‌کند.

۱- job scheduler or long term scheduler (admission scheduler or high level scheduler)
۲- " " or short " cpu (" or Dispatcher " process)
۳- " " or midium " swapper or memory
۴- " I/O

۱- job scheduler or long term scheduler

* این زمانبند در تمام os‌ها وجود ندارد و در برخی از os‌ها وجود دارد یا بصورت حداقلی وجود دارد (اغلب در batch system‌ها) مثلاً در سیستم‌های time-sharing مثل unix و windows اغلب این زمانبند وجود ندارد و بسادگی هر فرآیند جدید را در حافظه قرار می‌دهد تا توسط cpu-scheduler استفاده شود.

* همان طور که از اسم این زمانبند مشخص است این زمانبند دیر به دیر اجرا می‌شود، چند دقیقه فاصله بین ایجاد یک فرآیند جدید و فرآیند بعدی که می‌خواهد ایجاد شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

* یکسری request برای ورود به سیستم وجود دارد (درخواست برای تبدیل شدن به پراسس و قرارگیری در RAM) که در یک صف (در یک باند بر روی دیسک، spool شدند) قرار می‌گیرند حالا اگر os ببیند منابع کافی در اختیار دارد و بخواهد درجه multiprogramming سیستم را برای بالا بردن کارایی افزایش دهد اجازه می‌دهد که تعداد برنامه‌ها بیشتری در RAM لود شود (اجازه میده یک job دیگر به پراسس‌های دیگر سیستم اضافه شود)

✅ توجه کنید که اضافه کردن برنامه‌ها به سیستم تا یک جایی کارایی سیستم را بالا می‌برد و از یکجایی به بعد به علت وجود سربار os کارایی سیستم را پایین می‌آورد

* همان طور که قبلا اشاره کردم هر برنامه‌ای cpu time و I/O time دارد و بنابراین دو نوع پراسس داریم:

cpu-bound process: پراسسی که اکثر وقت‌اش صرف عملیات پردازشی شود

I/O-bound: " " " " " انتظار برای I/O شود ⇐ I/O های زیادی داشته باشد

باتوجه به موارد گفته شده ۲ نکته مهم در مورد long-term scheduler خواهیم داشت:

۱- استفاده بهینه از منابع سخت افزاری سیستم شامل Device ها، I/O و cpu

زمانبند بلند مدت باید ترکیب خوبی از پراسس‌های cpu-bound و I/O bound را وارد سیستم کند

در اینصورت وقتی پراسس I/O-bound به دنبال انجام عملیات I/O یشان می‌روند، پراسس‌های cpu-bound ای در سیستم وجود دارند که cpu را مشغول نگه خواهند داشت.

- اگر همه فرآیندها I/O-bound باشند ready queue تقریبا همیشه خالی خواهد بود و cpu scheduler کارچندانی برای انجام دادن نخواهد داشت و اگر همه فرآیندها cpu-bound باشند صف انتظار برای I/O ها تقریبا همیشه خالی می‌ماند در اینصورت سیستم unbalance خواهد شد بدین معنی که دستگاه‌ها بلا استفاده می‌مانند و بصورت بهینه مورد استفاده قرار نمی‌گیرند.

۲- پایداری یا stability سیستم:

باید process creation rate ⩽ process completion rate باشد تا سیستم در شرایط پایدار باشد

نرخ تولید پراسس توسط زمانبند بلند مدت مشخص می‌شود و این زمانبند هست که مشخص می‌کند هرچه مدت یکبار اجازه ورود یک پراسس جدید به سیستم را بدهد و سایر پراسس‌ها را در صف job ها منتظر نگه دارد

نرخ تمام شدن پراسس‌ها نیز به مقدار عملیات I/O و پردازشی که پراسس‌ها دارند وابسته است

حال اگر نرخ تولید پراسس توسط این زمانبند از نرخ اتمام پراسس‌ها موجود در سیستم بیشتر باشد تعداد پراسس‌های موجود در سیستم درگذر زمان بیشتر و بیشتر می‌شود و منابع سیستم پر می‌شوند (overflow می‌شوند) و بنابراین سیستم به حالت انفجاری می‌رسد نکته) اگر درجه multiprogramming پایدار باشد باید متوسط نرخ ایجاد فرآیند با متوسط نرخ خروج فرآیندهایی که از سیستم خارج می‌شوند برابر باشد، بنابراین ممکن است زمانبند بلند مدت تنها زمانی نیاز به فراخوانی داشته باشد که فرآیندی از سیستم خارج شود، بنابراین به دلیل فاصله زمانی (interval) طولانی‌تر بین اجراها، زمانبند بلند مدت زمان بیشتری برای تصمیم‌گیری در خصوص اینکه فرآیند بعدی که باید انتخاب کند کدام فرآیند باشد دارد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۲- cpu scheduler or short term scheduler

برخلاف job scheduler، cpu scheduler زودبه زود اجرا می‌شود و دارای High frequency هستند، بعنوان مثال هر ۱۰ تا ۱۰۰ میلی ثانیه یکبار اجرا می‌شود

وقتی cpu خالی می‌شود، cpu scheduler از بین فرآیندهای داخل صف ready (فرآیندهایی که می‌خواهند cpu را در اختیار بگیرند) یکی را برای اجرای روی cpu انتخاب می‌کند و به آن پراسس یک time slice می‌دهد، حال اگر آن پراسس در این مدت کارش تمام شد که خودش cpu را رها می‌کند اما اگر تمام time slice خودش را استفاده کرد و کارش تمام نشد، cpu scheduler، cpu را به زور از این فرآیند می‌گیرد (preempt می‌کند) و آن را به پراسس بعدی می‌دهد، البته ممکن است قبل از وقوع وقفه Time slice با وقفه نرم افزاری یا سخت افزاری cpu از پراسس گرفته شود.

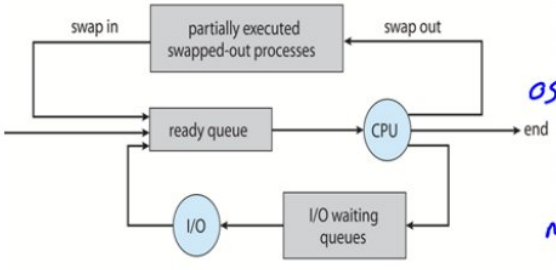
توجه) توجه کنید که در سیستم‌های time sharing ما برای ایجاد تعامل از سیاست preemption استفاده می‌کنیم ولی می‌توانیم در برخی از جاها از nonpreemptive هم استفاده کنیم

نکته) با توجه به اینکه cpu scheduler زود به زود اجرا می‌شود و در هر بار اجرا احتمال اینکه context switch انجام بشه بالاست، بنابراین سربار context switch برای ما اهمیت دارد و باید طوری زمانبندی را انجام بدهیم (طراحی کنیم) و طوری سیستم را طراحی کنیم که سربار context switch پایین باشد.

نکته) وقتی یک فرآیند روی cpu در حال اجرا است ممکن است در اثر وقوع وقفه یا cpu scheduler بیره وسط (اجرا شود) ولی اگر اولویت پراسس در حال اجرا از اولویت پراسس‌های دیگر درون صف ready بیشتر باشد ممکن است cpu را از پراسس مذکور نگیرد و دوباره cpu را به همین پراسس دهد، بنابراین تعداد اجرا cpu scheduler می‌تواند از تعداد بار switch بین فرآیندهای گوناگون بیشتر باشد.

۳- midium term scheduler (swapper or memory scheduler) (معمولاً در سیستم‌های Time sharing)

گاهی اوقات حذف یک فرآیند از حافظه و از رقابت فعال (active contention) برای cpu و در نتیجه کاهش درجه multiprogra... سودمند است، بعداً می‌توان این فرآیند را مجدداً به حافظه وارد کرد و اجرای آن را از جایی که متوقف شده بود ادامه داد، این کار swapping نامیده می‌شود.



Addition of medium-term scheduling to the queueing diagram.

اگر در مقاطعی از زمان پراسس‌هایی که داخل سیستم هستند بخواهند از منابع در اختیار os هستند منابع بیشتری داشته باشند و os با کمبود منابع مواجه شود، os باید یک یا چند فرآیند را از سیستم خارج کند که این ویژگی با استفاده از medium-term scheduler محقق می‌شود، با این زمانبند os می‌تواند درجه multiprogramming را پایین بیاورد.

- توجه کنید که فرآیندهای موجودات زنده سیستم ما هستند و مثلاً یک پراسس می‌تواند در ابتدای کار خودش حافظه کمی درخواست کند و os آن حافظه را بهش بدهد و آن را وارد سیستم کند ولی در مقاطعی از زمان‌های اجرایش نیاز به حافظه بیشتری داشته باشد
- اگر os در مقاطعی از زمان با کمبود منابع مواجه شد زمانبند میان مدت ممکن است یک فرآیندی که تاکنون در سیستم در حال اجرا بوده

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

، یعنی این فرآیند بصورت جزئی اجرا شده (partially executed) را بطور موقت متوقف کند و حافظه و سایر منابع اش را از آن بگیرد و وضعیت اش را در دیسک ذخیره کند که به این کار swap out می‌گویند ، حال زمانیکه os تشخیص بدهد که منابع سیستم در شرایطی هستند که می‌توان به این پراسس اجازه ورود دوباره را داد این کار را انجام می دهد که به اینکار swap in می‌گویند

نکته) با ترکیب زمانبند بلند مدت و میان مدت می‌توانیم با ماهیت پویا سیستم مقابله کنیم و بالانس مناسبی را در سیستم فراهم کنیم و از رسیدن سیستم به حالت انفجاری خودداری کنیم

نکته) همان طور که می دانید هارد کند است و اگر بخواهیم swapping زیادی داشته باشیم بسیاری از Bus/memory cycle صرف این امر می‌شود و در نتیجه باعث کند شدن هر چیزی که با هارد کار دارد خواهد شد.

نکته) اگر زمانبند میان مدت بخواهد فرآیندی را برای swap out انتخاب کند بهتر است آن را از بین فرآیندهای Block انتخاب کند و در صورت نیاز سراغ فرآیندهای ready برود

اگر فرآیندی در حالت بلاک باشد و آن را به هارد swap out کنیم در برخی از دیاگرام های حالت برای این فرآیند حالت جدیدی بنام (state) Blocked/waiting/Asleep ↔ swapped/suspend در نظر می‌گیرند

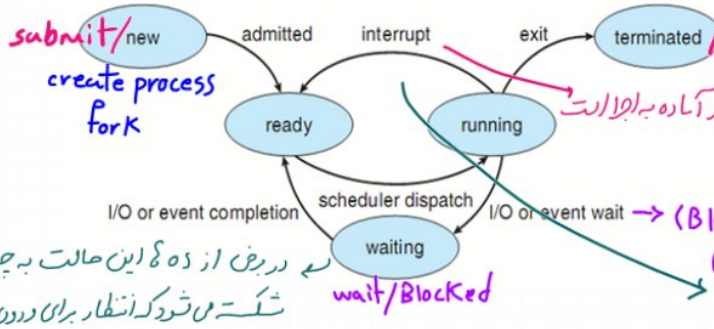
توجه) اگر فرآیندی که در حالت ready است را به هارد swap out کنیم در برخی از دیاگرام های حالت برای این فرآیند حالت جدیدی بنام (state) ready ↔ swapped/suspend

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

process state diagram or state transition diagram

قبلا دیاگرام حالت‌های یک پراسس را در حالت پایه بررسی کردیم و دیدیم که ۵ حالت پایه داشت که اساس کار سیستم‌های چند برنامه‌ای است
توجه) برخی از منابع دیاگرام حالت یک فرآیند را ۳ حالته می‌دانند و می‌گویند چرخه زندگی یک فرآیند شامل new و terminate می‌شود و می‌گویند در new هنوز فرآیندی وجود ندارد و در terminate هم دیگر فرآیندی وجود ندارد.
همان‌طور که قبلا به شما نشان دادم در سیستم‌های single programming دیاگرام حالت فرآیند ساده‌تر است.



وقتی یک job می‌خواهد وارد سیستم شود (به پراسس تبدیل شود) OS اگر حافظه و منابع کافی برای اینکه اجازه این کار را بدهد داشته باشد این اجازه را می‌دهد.
با وقفه تایمر cpu با اینکه پراسس هنوز آماده به اجرا است cpu به زور از پراسس گرفته می‌شود
نیاز به خواندن یا نوشتن (Blocked)
منتظر پایان از فرآیند دیگر (wait)
برخی از OS‌ها این حالت به چند حالت waiting شکسته می‌شود که انتظار برای دردی یا فروش یا یک رویداد یا دیوایس خاص را می‌تواند مشخص کند

* لزوما با وقفه تایمر cpu به زور از فرآیند در حالت اجرا روی cpu گرفته نمی‌شود، ممکن است مثلاً I/O یک فرآیند دیگری که Block است و اولویت اش بسیار بالاست تکمیل شود و در اثر وقفه OS می‌آید روی کار و فرآیند جاری را preempt می‌کند و به صف ready می‌فرستد و cpu را به فرآیندی که بیدار شده و اولویت اش بالاتر است می‌دهد

نکته) فرآیند از هرکدام از سه حالت اصلی ممکن است به حالت exit یا terminated برود، مثلاً فرآیند پدر فرآیند فرزند را به دلیل آنکه دیگر به سرویسی که ازانه می‌دهد نیازی نداشته باشد می‌کشد و منابعی که به فرآیند فرزند داده شده را ازش می‌گیریم، یا اینکه فرآیند پدر خاتمه پیدا می‌کند و در برخی از سیستم‌ها مثل mainframe کل زیردرخت مربوط به آن بصورت آبشاری (بعدا توضیح می‌دهم) خاتمه پیدا می‌کند.

نکته) اگر از حافظه مجازی استفاده شود درصورتی برای برنامه در حال اجرا page fault رخ دهد فرآیند از حالت ready به Block منتقل می‌شود یا OS صفحه مورد نظر را برایش از دیسک به حافظه منتقل کند که بعد از آن فرآیند از Block به ready می‌رود

سوال) عمل Dispatch بلافاصله بعد از وقوع چه رخدادهایی می‌تواند رخ دهد؟
هر وقت cpu خالی شود ← پس ببین چه فلش‌هایی از حالت running خارج شده.
در اثر ۳ تا از transition‌ها ۱) preempted (که به ۲ دلیل رخ میداد)
۲) exit/Kill
۳) I/O or event wait

نکته) اگر cpu خالی شود و برنامه‌ای برای اجرا وجود نداشته باشد cpu بالاخره باید یکاری بکند و نمی‌تواند بیکار بنشیند بنابراین یک برنامه مخصوص اجرا می‌کند که در OS‌های مختلف نام‌های مختلفی می‌تواند داشته باشد مثلاً Halt یا idle

- بسیاری از OS‌ها با استفاده از همین حالت‌ها ساخته شده‌اند، اما در برخی از سیستم‌ها نیاز به حالت‌های دیگری هم وجود دارد.
درسته که ما یک عالم فرآیند رو داریم حافظه اصلی کردیم که برخی از اینها ready و برخی Block دیگی running است و وقتی فرآیند در حال اجرایی Block می‌شود ما می‌توانیم فرآیند دیگری را از ready برای اجرا انتخاب کنیم اما پردازنده خیلی خیلی سریعه و در حالت Block

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

بدون هم آیند در این سیستم یک امر عادی است! یعنی با multiprogramming هم پردازنده می‌تواند اکثر اوقات بیکار باشد و بنابراین بهره وری سیستم میاد پایین. حالا چاره چیست؟

۱- بزرگ کردن حافظه اصلی ← خیلی خوب نیست، چرا؟

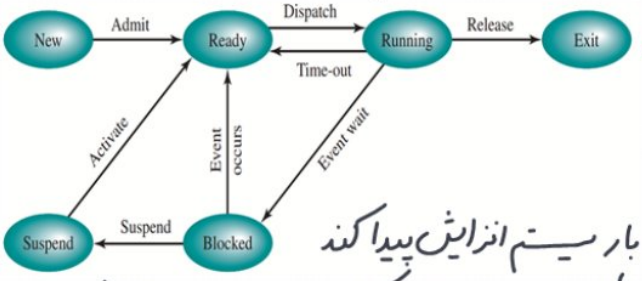
۱- افزایش هزینه

۲- حافظه بزرگتر باعث می‌شود برنامه‌ها از فضا بیشتری استفاده کنند و فرآیندهای بزرگتری ساخته می‌شود و لزوما باعث نمی‌شود تعداد فرآیندهای درون حافظه افزایش یابد ← با گذر زمان برنامه‌ها حافظه‌های بیشتری مصرف می‌کنند

۳- با افزایش اندازه DRAM به علت‌های سخت افزاری Access time اش زیاد می‌شود

راه حل: swapping ← بخشی یا تمام یک فرآیند از حافظه اصلی به دیسک منتقل می‌شود

زمانیکه هیچ یک از فرآیندهای موجود در حافظه اصلی در حالت ready نباشند OS یکی از فرآیندهای Block را از حافظه اصلی خارج و آن را به دیسک می‌برد و آن فرآیند را در صف فرآیندهای suspend می‌برد



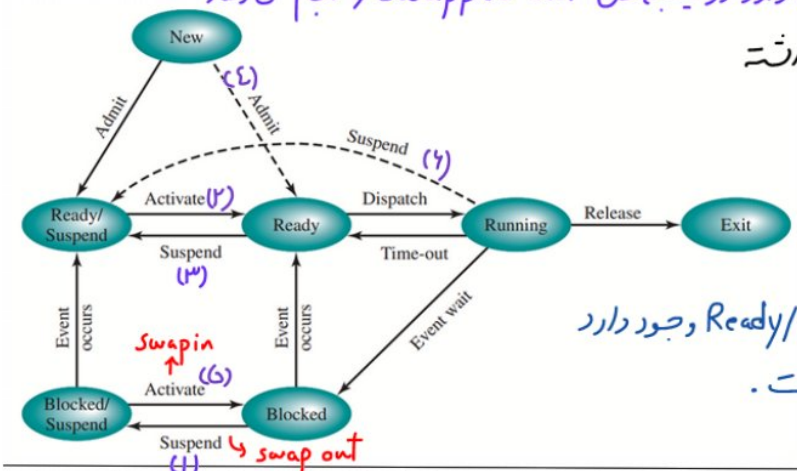
سوال) بعد از آنکه OS یک فرآیند را swapped out می‌کند چه فرآیندی را به حافظه اصلی بیاورد تا آن را اجرا کند؟

فرآیندی که جدیدا ایجاد شده را می‌توانند بپذیرند: این انتخاب باعث می‌شود بار سیستم افزایش پیدا کند

آیا انتخاب بهتری وجود دارد؟ اگر فرآیندی که قبلا Blocked بوده و آن را suspend (معلق) کردیم و سرویس I/O اش را در این فاصله دریافت کرده را بیاوریم انجام دهیم انتخاب بهتری است چون داریم به فرآیندی که توی سیستم بوده سرویس بیشتری می‌دهیم، حالا یک فرآیندی که suspend است وقتی سرویس اش را می‌گیرد باید حالت اش عوض بشه که بفهمیم آماده اجرا است و اگر خواستیم swapped in اش کنیم به حافظه اصلی و ببریم اش توی صف ready

بنابراین نیاز هست حالت جدیدی داشته باشیم

نکته: اگر در حافظه اصلی نیاز به فضا بیشتری داشته باشیم، OS ابتدا فضاهایی که در اختیار فرآیندها هست و زیاد استفاده نمی‌شود را ازشون می‌گیره و اگر مشکل حل نشد نیازه که یکی فرآیند را کلا ببریم روی دیسک، توجه کنید که swapped out کردن یک فرآیند فقط به دلیل نبود فرآیندهای ready نیست و گاهی اوقات OS می‌بیند که فرآیند در حال اجرا یا فرآیند بعدی که قرار هست بلافاصله بعد از فرآیند در حال اجرا فعلی اجرا شود نیاز به حافظه بیشتری دارد در نتیجه عمل swapped out را انجام می‌دهد



۱- اگر فرآیند آماده‌ای وجود نداشته باشد یا OS نیاز به وجود فضا داشته باشد حداقل 1 فرآیند Block به دیسک برده می‌شود

۲- در ۲ حالت این اتفاق ممکن است رخ دهد

۱- فرآیند ready در RAM نداریم

۲- در صف ready فرآیند داریم ولی فرآیندی در صف Ready/suspend وجود دارد که اولویت اش از تمامی فرآیندهای درون ready بیشتر است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) معمولاً OS اگر بخواهد فرآیندی را swapped out کند از فرآیندهای Block شده (مثلاً فرآیندی که اولویت اش هم پایین است) که نمی‌توانند اجرا بشوند و الکی حافظه را اشغال کردند انتخاب می‌کند.

۳- ولی در ۲ صورت ممکن است OS از بین فرآیندهای ready نیز فرآیندی را suspend کند

۱- اگر چاره‌ای جز این برای خالی کردن حافظه به اندازه کافی بزرگ نداشته باشیم.

۲- OS به جای آنکه فرآیند Block ای که اولویت اش زیاد است و حدس می‌زند به زودی کار I/O اش تمام می‌شود و ready می‌شود یک فرآیند با اولویت پایین را از ready، suspend می‌کند.

۴- وقتی فرآیندی جدیدی ایجاد میشه می‌تواند به صف ready یا ready/suspend برود که معمولاً به ready/suspend میره.

در هر صورت به هر کدوم بخواد بره باید PCB اش ساخته بشه و همین طور Address space اش بهش تخصیص داده شود

چرا معمولاً از new به ready/suspend میره؟

اگر OS بخواد هم new ها را در RAM بگذارد درسته که همیشه می‌تواند یک pool بزرگی از فرآیندهای ready اما

اولا: یک فرآیند جدید رو که بخواهیم بذاریم توی حافظه مرتبا براش حافظه ای نیست

دوما: ممکن است سیستم دچار حالت انفجاری بشه

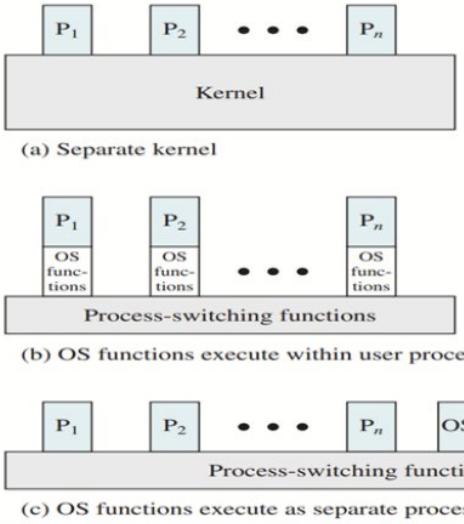
۵- این حالت کی استفاده میشه؟ در این حالت فرآیند هم Block است و هم حافظه ندارد.

فرض کنید فرآیندی تمام می‌شود و مقداری حافظه آزاد می‌شود و OS تصمیم می‌گیرد که یکی از فرآیندهای suspend را به RAM بیاورد، حالا فرآیندی در صف Blocked/suspend وجود دارد که اولویت اش از تمام فرآیندهای صف ready/suspend بیشتر است و OS تخمین می‌زند که I/O این فرآیند به زودی تمام می‌شود در این شرایط تصمیم می‌گیرد این فرآیند را به حافظه RAM بیاورد تا یک فرآیند ready که قبلا swapped out شده.

۶- اگر یک فرآیند با اولویت بالا از صف Blocked/suspend هم اکنون از حالت Block خارج شود OS می‌تواند فرآیند در حال اجرا را preempt کند و مستقیماً آن را به صف ready/suspend بفرستد تا بتواند بخشی از حافظه اصلی را برای این فرآیند با اولویت بالا آزاد کند

نکته) از هر حالتی می‌تواند به حالت خروج رفت

نحوه اجرا OS های مختلف



۱- Nonprocess Kernel

در یکی از سنتی و رایج در بسیاری از OS های قدیمی این بود که هسته OS خارج از هر فرآیندی اجرا شود، در این روش OS حافظه و پشته مختص به خودش را برای کنترل فراخوانی ها دارد، در این روش concept فرآیند فقط برای برنامه‌های کاربر اعمال می‌شود و کد OS بعنوان یک موجودیت جداگانه در مد سمتا اجرا می‌شود

۲- Execution within user processes

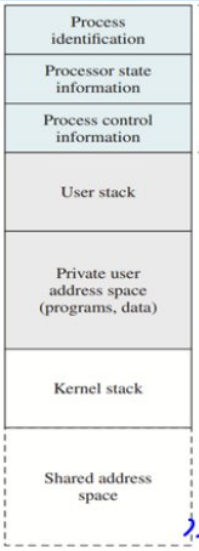
(اجرا درون پراسس های فرآیند)

بخش اعظم OS در محیط فرآیند کاربر اجرا می‌شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



یکی از رویکردهایی است که در OS های کامپیوترهای کوچک نظیر pc و workstation متداول است
در این روش تمام نرم افزار OS در context ـ فرآیندهای کاربران اجرا می‌شود، در این روش OS مجموعه‌ای از روتین‌هایی است که کاربران برای انجام کارهایشان فراخوانی می‌کنند و در محیط فرآیند هر کاربر اجرا می‌شود و OS image process را مدیریت می‌کند، همان طور که در شکل مقابل می‌بینید تصویر هر فرآیند برنامه، داده‌ها و پشته برنامه‌های هسته را نیز در خود دارد

ـ کدها و داده‌های OS در یک حافظه مشترک است که برای تمامی فرآیندها به اشتراک گذاشته شده

⭐ برنامه‌های OS که در فرآیندهای کاربران مختلف اجرا می‌شوند یکسان است.

وقتی یک فرآیند به مد هسته می‌رود از یک پشته هسته مجزا برای مدیریت فراخوانی‌ها و بازگشت آنها استفاده می‌شود

ـ وقتی یک وقفه رخ می‌دهد مدیر دارنده به privilege تغییر می‌کند و کنترل به OS داده می‌شود، توجه به این نکته ضروری است که در اینجا تعویض فرآیند انجام نمی‌شود و اجرا در درون فرآیند جاری کاربر ادامه می‌یابد و فقط در همان فرآیند تعویض حالت رخ می‌دهد ← در state diagram ـ unix این مدل پیاده‌سازی شده که به شما نشان خواهیم داد.

⛔ خراب کاری نشد؟ در حین اجرا فرآیند کاربر داریم کد OS را اجرا می‌کنیم!
خیر، اگرچه کد OS در محیط کاربر اجرا می‌شود ولی کد OS در مد هسته اجرا می‌شود و کاربران (فرآیندها) نمی‌توانند توابع OS را دستکاری کنند.

🔔 توجه کنید که برنامه با فرآیند فرق می‌کند و رابطه این دو تا یک به یک نیست و در داخل یک فرآیند ممکن است هم برنامه کاربر و هم برنامه‌های OS اجرا شوند

حال وقتی OS کارش را تمام می‌کند ۲ تا تصمیم می‌تواند بگیرد:

۱ـ اجرای فرآیند جاری ادامه پیدا کند: در این حالت OS با یک تعویض حالت اجرا فرآیند جاری که در اثر وقفه رخ شده بود را در داخل همان فرآیند از سر می‌گیرد ← خلاصه: فرآیند کاربر برای استفاده از یک روال OS با وقفه قطع می‌شود و جدا از سرگرفته می‌شود بدون آنکه متحمل جریمه دو تا تعویض فرآیند شود

← در این حالت ۲ تعویض فرآیند save می‌شود (جریمه ۲ تعویض را نخواهیم داشت که این موضوع یکی از مزایای اصلی این روش است)

۲ـ یک فرآیند دیگر اجرا شود: دیگه توی این حالت باید تعویض فرآیند انجام بشه، در این حالت پس اینکه روال فراخوانی شده OS تمام شد، برنامه تعویض فرآیند اجرا خواهد شد. معمولا اجرای برنامه تعویض فرآیند در خارج از تمام فرآیندها صورت می‌گیرد

۳ـ process-Based operating system
در این روش OS بصورت مجموعه‌ای از فرآیندهای سیستمی پیاده‌سازی می‌شود یعنی اعمال اصلی هسته بصورت فرآیندهای جداگانه پیاده‌سازی می‌شوند.
در اینجا نیز ممکن است بخشی از کد تعویض فرآیند خارج از هر فرآیندی اجرا گردد
این روش برای پیاده‌سازی modular OS با ارتباط شفاف و حداقل و همین طور سیستم‌های چند پردازنده‌ای مفید است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

process states diagram in unix

تعداد حالت هایی که یک فرآیند می تواند در آن باشد در یک os واقعی می تواند بیشتر از ۳، ۵ یا ۷ تایی باشد که مشاهده کردید
بعنوان مثال unix ۹ تا حالت دارد و ویندوزهای قدیمی ۶ حالت دارد که در زیر به دیاگرام حالت یک پراسس در unix می پردازیم.
unix از مدل Execution within user processes استفاده می کند که در آن بخش اعظم os در محیط فرآیند کاربر اجرا می شود

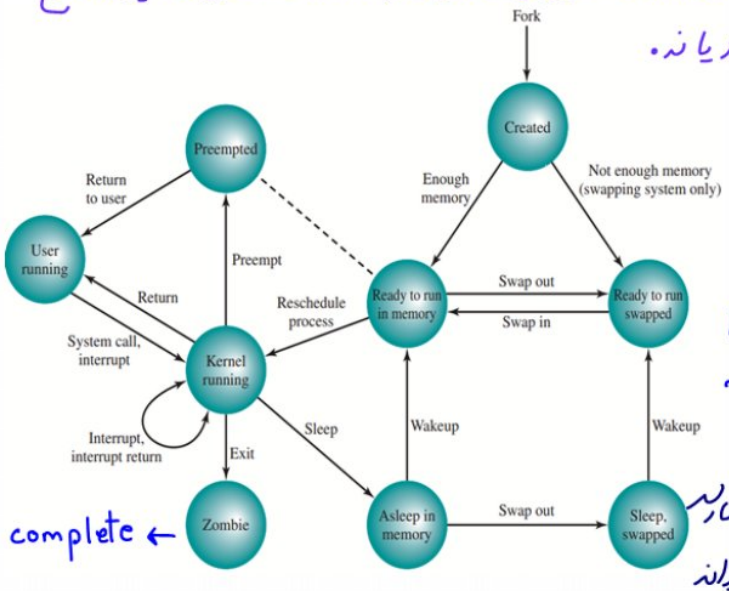
★ unix از ۲ دسته (نوع) فرآیندها استفاده می کند:

۱- user processes: فرآیندهای کاربر برای اجرا برنامه های کاربر در مد کاربر اجرا می شود و برای اجرای دستورات متعلق به هسته در مد هسته اجرا می شوند

۲- " system: این فرآیندها کد os را در مد هسته اجرا می کنند

توضیح حالت های مختلف یک پراسس در unix

created: پراسس خلق شده ولی ready نشده و هم شرایط برای ready شدن اش مهیا نشده و بنابراین سیستم به آن سرویس نمی دهد (فرآیند به تازگی توسط فراخوانی سیستمی ایجاد شده اما آماده اجرا نیست)، هر وقت تمامی شرایط برای آن مهیا شد و به حافظه اصلی منتقل شد حالت اش ready می شود و سیستم می تواند به آن سرویس دهد. توجه کنید که creat شدن یک فرآیند در unix مستقل از این موضوع است که آیا فضا کافی برای قرارگیری این فرآیند در حافظه اصلی وجود دارد یا نه.



Ready:

ready to run in memory: فرآیند ready (آماده برای اجرا شدن) و در حافظه اصلی است و منتظر است تا کرنل آن را زمانبندی کند

preempted: ۲ حالت Ready to run in memory و preempted در اصل ۱ حالت هستند، برای همین هم بین شون نقطه چین گذاشته دلیل این تمایز برای تاکید بر نحوه ورود به حالت preempted هست.

وقتی فرآیند در مد هسته در حال اجرا است (در اثر وقوع وقفه) زمانی فرا می رسد که هسته کارش را کامل کرده و آماده است تا کنترل را به برنامه کاربر برگرداند در این نقطه هسته ممکن است تصمیم بگیرد به خاطر فرآیندی که در صف Ready حافظه اصلی است و اولویت بالاتری دارد فرآیند جاری را preempt کند که در اینصورت فرآیند جاری به حالت (نه صف) preempted می رود

نکته) فرآیندهایی که در حالت preempted هستند و فرآیندهای ready to run in mem با هم یک صف را تشکیل می دهند.

نکته) توجه کنید که امکان preempt شدن فرآیندی که در مد هسته در حال اجراست وجود ندارد و فقط زمانی می تواند preemption رخ دهد که یک فرآیند در حال رفتن از حالت هسته به حالت کاربر باشد.

نکته) فرآیندهایی که وارد سیستم می شوند نمی دونیم چقدر به cpu نیاز دارند و نمی دونیم کوتاه هستند یا بلند پس اگر بخوایم interactive (تعاملی) بودن سیستم رو حفظ کنیم باید preemption داشته باشیم، فرآیندی که توی زمانی که بهش دادیم تموم نشده و preempt شده احتمالا

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

cpu-bound است و لحظاتی از زمان که بهره وری cpu پایین آمده است گزینه مناسبی هستند که در اولویت قرار بگیرند و برای اجرا انتخاب شوند تا بهره وری cpu را بالا ببرند، حالت preempted ای که در unix برای فرآیندها در نظر گرفتند option جدید و جالبی برای زمانبندی به وجود می آورد.

ready swapped OR ready to run_swapped:

فرآیند ready ولی چون با کمبود حافظه اصلی مواجه بودیم فعلا آن را روی هارد گذاشتیم، هر زمان که حافظه اصلی جا داشته باشد زمانبند حافظه می تواند این فرآیند را به حافظه اصلی ببرد که در اینصورت حالت اش به ready to run in memory تغییر می کند

Block

Asleep in memory (به خواب رفته در حافظه اصلی): همان Block است، فرآیند در انتظار تکمیل I/O یا یک رویداد به خواب رفته

sleep swapped: فرآیند در حالت Asleep در حافظه بوده و به علت کمبود حافظه زمانبند حافظه آن را swapped out کرده به هارد، توجه کنید که این فرآیند هنوز فرآیند است و ماهیت فرآیند بودن خودش را از دست نداده و فرآیند زنده ای است.

running

user running : فرآیند در حال اجرا در مد کاربر است.

Kernel running : فرآیند در حال اجرا در مد هسته است.

Zombie : زمانیکه اجرا منطقی فرآیند فرزند تمام می شود os منابع ای که به آن فرآیند اختصاص داده را از آن فرآیند می گیرد ولی سطر آن فرآیند در جدول فرآیند (PCB و process table اش) را فعلا پاک نمی کند تا زمانی که پدر آن wait کند که در آن صورت os آن فرآیند را بصورت فیزیکی نیز حذف می کند، در واقع چون جدول فرآیند (process table) حاوی وضعیت خروج فرآیند است و پدر به این وضعیت نیاز دارد تا متوجه شود آیا فرزندش کارش را با موفقیت انجام داده یا خیر و موارد دیگری از این دست، بنابراین entry جدول فرآیند مربوط به فرآیند فرزند تا زمانی که والد یک فرآیند سیستم کال wait را صدا می زند باید باقی بماند، وقتی والد wait را فراخوان کند در wait وضعیت فرآیند فرزند به آن داده می شود و بنابراین os می تواند، pcb و pid فرآیند فرزند را از آن بگیرد

فرآیندی که پایان یافته اما والد آن هنوز wait را فراخوانی نکرده بعنوان یک فرآیند زامبی شناخته می شود. همه فرآیندهایی که terminate می شوند وارد حالت زامبی می شوند اما عموما فقط برای مدت کوتاهی در حالت زامبی هستند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

operations on processes

۱- process creation
۲- " termination
۳- " communication

process creation

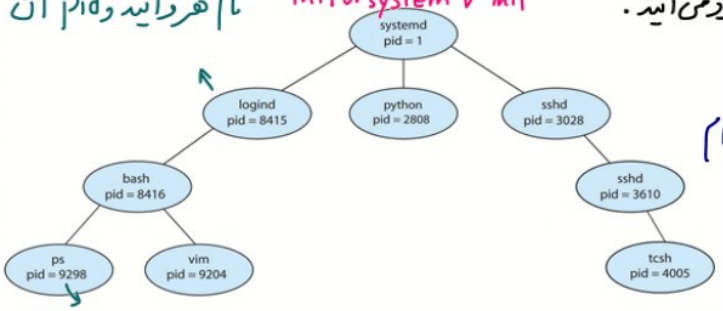
یک پراسس فقط توسط یک پراسس والد (parent، پدر) به وجود می‌آید

سوال: اولین پراسس چگونه به وجود می‌آید؟ اصطلاحا می‌گویند اولین پراسس بصورت دستی ایجاد می‌شود

وقتی Bootstrap program سیستم‌عامل را توی RAM لود می‌کند کنترل به دست os می‌افتد و به این ترتیب اولین پراسس ایجاد شده است، پس از این پراسس تمام فرآیندها صرفا توسط پراسس والد به وجود می‌آیند.

نام هر فرآیند و pid آن

init or system V init



A tree of processes on a typical Linux system.

لیست پراسس‌ها
ps -el

توجه) لینوکس به جای واژه پراسس از task استفاده می‌کند

- فرآیند systemd همیشه pid=1 دارد اولین فرآیند کاربرانت و هنگام بوت شدن سیستم ایجاد می‌شود و بعنوان فرآیند والد ریشه برای همه فرآیندهای کاربرانت.

پس از بوت شدن سیستم فرآیند systemd فرآیندهایی را ایجاد می‌کند که آن فرآیندها خدماتی مانند وب سرور، پرینت سرور، login، ssh server و... را ارائه می‌کند.

فرآیند login مسئول مدیریت کلاینت‌هایی است که مستقیما وارد سیستم می‌شوند که در شکل بالا یک کلاینت وارد شده و از پوسته bash (bash shell) استفاده می‌کند که به آن pid = ۸۴۱۶ تخصیص داده شده، این یوزر با استفاده از bash command line interface پراسس ps و همین طور ویرایشگر vim را ایجاد کرده

فرآیند sshd مسئول مدیریت کلاینت‌هایی است که با استفاده از ssh (secure shell) به سیستم وصل می‌شوند.

- os‌های لینوکس در ابتدا از system V init استفاده می‌کردند اما توزیع‌های جدید از systemd استفاده می‌کنند که انعطاف‌پذیرتر است و خدمات بیشتری را نسبت به init اولیه ارائه می‌کنند.

ps -el : اطلاعات کاملی را از تمامی فرآیندهای فعال در سیستم لیست می‌کند.

pstree : این دستور در لینوکس درختی از تمام فرآیندهای سیستم را نشان می‌دهد.

وقتی یک parent process یک child process به وجود می‌آورد

۱- از نظر منابع

سلما فرآیند فرزند مثل هر فرآیند دیگری برای اجرا شدنش نیاز به cputime، حافظه، files، I/O devices دارد حال حالت‌های زیر می‌تواند وجود داشته باشد

۱- ممکن است بتواند منابع موردنیازش را مستقیما از os بگیرد
۲- " " به زیرمجموعه‌ای از منابع فرآیند والد محدود شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) فرآیند والد ممکن است منابع خود را بین فرزندانش تقسیم کند یا ممکن است بتواند برخی از منابع اش مثل حافظه یا فایل ها را بین چند فرزندش به اشتراک بگذارد.

نکته) محدود کردن یک فرآیند فرزند به زیرمجموعه ای از منابع والدش، جلوگیری می کند که یک پراسس بتواند به وسیله تولید فرآیندهای فرزند زیاد سیستم را overload کند.

نکته) علاوه بر تامین منابع منطقی و فیزیکی مختلف فرزند توسط والد، والد ممکن است داده های ورودی (initialization data) مثل نام فایلی که فرزند می خواهد آن را باز کند و روی آن بنویسد یا نام دستگاه فروجی که قرار است با آن کار کند را به فرزندش بدهد

۲- از نظر نحوه اجرای پراسس والد و پراسس فرزند

۱- پراسس والد بصورت همروند (concurrent) با پراسس فرزند به اجرایش ادامه دهد.

۲- " " صبر می کند تا پراسس فرزند تمام شود و بعد از آن اجرای خودش را از سر می گیرد

توجه: توجه کنید که همروند به معنای موازی نیست و در اجرای همروند هر فرآیندی در time slice ای که بهش اختصاص داده می شود اجرا می شود.

۳- از نظر محتوای حافظه ۲ حالت برای پراسس والد و فرزند وجود دارد (از نظر محتوای Address space)

۱- هنگامیکه فرآیند فرزند به وجود می آید (در همان لحظه ابتدایی) حافظه اش کپی (عین، Duplicate) پراسس والد باشد، یعنی فرآیند فرزند دقیقاً همان کد و همان داده فرآیند والد را دارد ولی بعنوان یک موجودیت جداگانه اجرا می شود، در unix به اینصورت است

۲- هنگامیکه فرآیند فرزند به وجود می آید فرآیند فرزند که داده جداگانه ای دارد یعنی یک فرآیند کاملاً متفاوت است و یک برنامه ای است که بصورت جداگانه در حافظه اصلی لود می شود و پس اجرا می شود (هر زمان که بهش time slice داده شود اجرا می شود)، در ویندوز به اینصورت است

- هر os برای ایجاد فرآیند فرزند می تواند روال مختص به خودش را داشته باشد که ما در زیر ایجاد فرآیند فرزند در os های unix و ویندوز را به ترتیب خواهیم گفت.

Unix Fork system call

ایجاد فرآیند جدید (فرزند) در سیستم عامل unix با استفاده از system call (از فرمان هسته) fork است. وقتی فرآیندی (فرآیند والد) درخواست fork می کند سیستم عامل unix اعمال زیر را انجام می دهد.

۱- برای فرآیند فرزند یک سطر در جدول فرآیند (process table) تخصیص می دهد، یک pcb جدید برای فرزند ایجاد می کند

۲- یک شناسه یکتا برای این فرزند در نظر می گیرد.

۳- یک کپی از تصویر فرآیند پدر (Address space) به استثنای حافظه های مشترک ایجاد می کند.

۴- شمارنده های هر file ای که به فرآیند والد متعلق هست را افزایش می دهد تا نشان دهد که یک فرآیند اضافی نیز اکنون مالک آن فایل ها است

۵- به فرآیند فرزند حالت ready را تخصیص می دهد.

۶- pid فرزند را به فرآیند پدر و مقدار ۰ را به فرآیند فرزند برمی گرداند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

تمام این اقدامات در هسته در فرآیند پدر صورت می‌گیرد، وقتی هسته هم این اعمال را کامل کند می‌تواند یکی از اعمال زیر را بعنوان بخش از dispatcher routine انجام دهد:

۱- ماندن در فرآیند پدر، در نقطه فراخوانی fork توسط والد، کنترل در مد کاربر به والد برمی‌گردد

۲- کنترل به فرآیند فرزند منتقل می‌شود: اجرا فرآیند فرزند از نقطه شبه درکد فرآیند پدر (آخرین pc پدر) یعنی نقطه پس از بازگشت از فراخوانی fork شروع می‌شود

۳- کنترل به فرآیند دیگری منتقل شود: بنابراین هم فرآیند والد و هم فرآیند فرزند در حالت ready قرار می‌گیرد

- در زیر مثال و یک نمونه برنامه‌ای می‌بینید که در درون خودش از فراخوانی سیستمی fork استفاده کرده

پس از فراخوانی سیستم کال fork (در صورتیکه این سیستم کال موفقیت آمیز باشد) ما ۲ فرآیند در سیستم داریم و هر دوی این فرآیندها دقیقا یک کد، یک دیتا و یک pc دارند (فرآیند فرزند Duplicate فرآیند پدر است) و هر دو فرآیند پس از اجرای fork دقیقا روی دستور (خط) پس از اجرا fork یعنی آن assignment ال که خروجی fork را درون یک متغیر (در شال زیر متغیر K) می‌گذارد است.

نکته مهم دیگری که باید بدانید این است که اگر fork موفقیت آمیز باشد OS به فرآیند پدر، شناسه (pid) فرآیند فرزند که یک عدد بزرگتر از ۰ است را بعنوان خروجی می‌دهد و به فرآیند فرزند خروجی ۰ می‌دهد

← نتیجه: پس از اجرا fork ما یک کد پدر داریم و یک کد که فرزند که خروجی fork برای پدر بزرگتر از ۰ و برای فرزند ۰ است.

← پس شخص است که با استفاده از همین مقدار خروجی ۲ فرآیند والد و فرزند می‌توانند متوجه بشوند که کدوم شون والد هست و کدوم شون فرزند است، در واقع با یک if و else می‌توانیم مسیر پرایس فرزند و والد را از هم جدا کنیم

پس اگر pid > ۰ باشد فرآیند متوجه می‌شود که فرآیند والد است و اگر pid = ۰ باشد فرآیند متوجه می‌شود که فرآیند فرزند است.

اگر به کدهای زیر توجه کنید فرآیند والد وارد else می‌شود و فرآیند فرزند وارد if می‌شود، فرآیند فرزند در if یک فراخوانی سیستمی به اسم execlp که یک فراخوانی سیستمی از خانواده exec است را صدا می‌زند که این فراخوانی سیستمی کارش این است که یک برنامه دیگر را که کاملا متفاوت است و روی هارد است را توی RAM لود می‌کند، در اینجا با فراخوانی execlp با این پارامترها برنامه ls که کارش لیست کردن اسم فولدرها است لود می‌شود و کل کد فرآیند فرزند رونویسی می‌شود و کد جدیدی داریم که دیگر کپی والد نیست و بنابراین مسیر فرآیند فرزند و پدر عوض می‌شود

- متغیر x مقدار ابتدایی ۰ دارد.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t K;

   /* fork a child process */
   K = fork();

   if ( K < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if ( K == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

if (fork!=0){ parent code } else { child code }

فرآیند والد

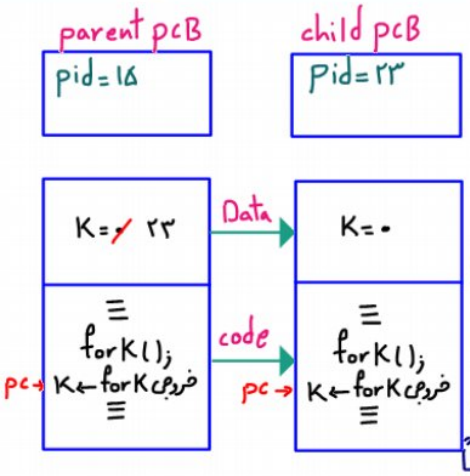
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t K;

   /* fork a child process */
   K = fork();

   if ( K < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if ( K == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

فرآیند فرزند پس از اجرا دستور fork

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



فرآیند والد در سیستم حضور دارد و pcB و فضای آدرسی خودش را دارد فرض کنید pid والد ۱۵ باشد وقتی fork توسط والد فراخوانی می‌شود از مدیوزر به مد کرنل سوئیچ می‌شود، حال os برای فرآیند فرزند pcB جدیدی می‌سازد و یک شناسه، مثلاً ۲۳ را اختصاص می‌دهد همچنین پس از آنکه سیستم عامل pcB فرزند را ساخت حافظه‌ای به فرآیند فرزند اختصاص می‌دهد و همان کد و همان داده فرآیند والد را در این حافظه کپی می‌کند، حالا چون فرآیند والد سیستم کال fork را صدا زده os، pid فرآیند فرزند یعنی ۲۳ را به عنوان خروجی fork به والد برمی‌گرداند و پس assignment مذکور اجرا می‌شود و K مقدار ۲۳ را به خود می‌گیرد، حالا هر وقت نوبت به فرآیند فرزند برسد فرآیند فرزند دقیقاً از مکان pc شروع به اجرا می‌کند ولی چون خروجی سیستم کال fork به فرآیند فرزنده برگردانده نشده، بنابراین برای فرآیند فرزند K=۰ است و بنابراین به واسطه ۰ بودن K مسیر فرآیند فرزند از فرآیند پدر جدا می‌شود.

توجه) در مثال با ۴۷ تا فراخوانی سیستمی استفاده شد: fork و wait و execlp و exit

نکته) سیستم کال exec باعث می‌شود کل فضای آدرسی فرآیند فرزند شامل کد، دیتا، استک و ... با برنامه مورد نظر جایگزین شود

نکته) لزومی ندارد فرآیند والد پس از fork حتما wait کند و اگر کاری داشته باشد می‌تواند به کارش ادامه دهد یا حتی می‌تواند فرزندان دیگری را با fork به وجود آورد اما اگر در حین اجرا فرآیند فرزند کار دیگری برای انجام دادن نداشته باشد می‌تواند سیستم کال wait را صدا بزند تا خودش را تا زمان پایان فرآیند فرزند از صف ready خارج کند.

نکته) شما دبل کلیک هم که می‌کند در بک گراند fork صدا زده می‌شود

نکته) فرآیند فرزند می‌تواند یکسری از روال‌های مربوط به همان برنامه پدر را انجام دهد که در اینصورت exe فرزند با exe والد یکسان خواهد بود و یا بطور کلی برود یک برنامه دیگر را اجرا کند که در اینصورت باید کد exe اش را عوض کنیم

توجه) فرزند و والد همه‌چی شان مانند هم است و فرزند مواردی از جمله نام کاربر اجرا کننده، دایرکتوری جاری (در همان دایرکتوری که پدر دارد کار می‌کند، کار می‌کند)، توصیف‌گر فایل‌های باز (فرزند همه منابع از جمله این را از پدر می‌گیرد) از والد به ارث می‌برد به جز pid و time slice و اولویت

نکته) گفته می‌شود که فرآیند فرزند نسخه یکسانی از فرآیند پدر است اما در اصل فرآیند فرزند تا زمانیکه چیزی را تغییر ندهد از فضایی که برای برنامه و داده‌های پدر وجود دارد استفاده می‌کند حال با اولین تغییری که توسط فرزند یا پدر به وجود آید قبل از اعمال تغییر یک کپی از فضای آدرسی تهیه می‌شود و اصل آن به پدر و کپی آن به فرزند داده می‌شود که این روش با نام copy on write یا copy on modify شناخته می‌شود، و به اینصورت است که فلگ copy on write یک صفحه‌ای از حافظه که بین چند فرآیند مشترک می‌شود را ۱ می‌کنیم و هر فرآیندی که بخواهد در این فضا write کند باید یک copy از این فضا برایشان گرفته شود و به آن فرآیند داده شود تا در کپی خودشان عمل write را انجام بدهند تا این فرآیندها روی هم تاثیر نگذارند

توجه) توجه کنید که پس از آنکه راه فرآیند فرزند و پدر را جدا کردیم تغییراتی که فرآیند فرزند در متغیرهایش می‌دهد در داده‌های فرآیند پدر تاثیری ندارد و بالعکس.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

توجه) قطعه کد بالا از کتاب سیلبرشاتز آورده شده بود در قطعه کد کتاب به جای متغیر x از متغیر pid استفاده کرده که چون باعث گمراهی شما می‌شود اسم آن را عوض کردم

نکته) ۲ نسخه از wait و ۲ نسخه از exec داریم

جدید: waitpid(pid, &statloc, opts)
قدیمی: wait(&status)
جدید: execve(name, argv, envp)
قدیمی: exec

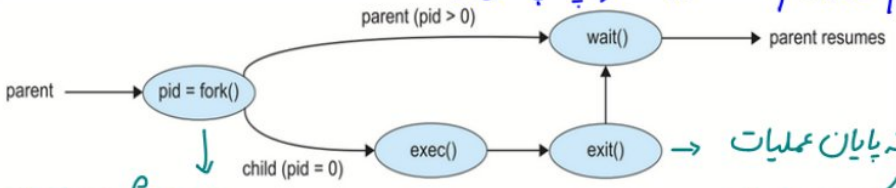
تا بیداری کدام فرزند بخوابد

فرآیند والد باید از وضعیت خروج فرزندش با خبر شود (اینکه فرزند با موفقیت کارش را انجام داده یا خیر یا آیا بواسطه انجام کار غیرمجاز کشته شده یا ...) این وضعیت از طریق پارامتر statloc به پدر داده می‌شود تا پس از آنکه از حالت بلاک خارج شد متوجه وضعیت فرزندش شود. مقدار statloc با فراخوانی exit توسط فرزند مشخص می‌شود که یه نشان دهنده خاتمه عادی است.

exit(status)

با دادن پارامتر به waitpid می‌توانیم فرآیند را برای پایان کار یک فرزند مشخص بلاک کنیم، مثلاً اگر برنامه نویس درکدش بنویسد waitpid(۲۳) به این معناست که فرآیند جاری را تا پایان کار فرآیند فرزند با ۲۳ = pid بخوابان، اگر برنامه نویس چندتا فرزند تولید کرده باشد و بخواهد تا اتمام تمامی فرزندان به حالت بلاک برود می‌نویسد waitpid(-۱)

فرآیند پدر هم زمانی که time slice بهش برسه فراخوانی سیستمی wait را فراخوان می‌کند که کار این فراخوانی سیستمی این است که صبر می‌کند (فرآیند به حالت Block می‌رود) تا زمانیکه وقفه پایان فرآیند فرزند بیاید و وارد صف ready شود و بعد از اینکه time slice بهش رسید به اجرایش ادامه می‌دهد و در نهایت پیام child complete را چاپ می‌کند



Process creation using the fork() system call.

وقتی سیستم کال exit توسط فرزند فراخوانی می‌شود وقفه پایان عملیات فرزند به وسیله این سیستم کال به فرآیند والد که wait کرده بوده می‌رسد و والد بیدار می‌شود و پس می‌تواند کارش را ادامه دهد

والد fork را صدا می‌زند و خروجی fork برای هردو فرآیند والد و فرزند برمی‌گردد

فراخوانی exit می‌تواند implicit یا explicit باشد ← اگر فراخوانی exit وجود نداشته باشد کامپایلر که آن را در انتهای دستورات برنامه درج می‌کند

creating a seperate process using win32 API

در win32 API از سیستم کال createprocess(...) برای ساخت فرآیند جدید استفاده می‌شود

نکته) در سیستم عامل ویندوز فرآیند فرزند برخلاف یونیکس Duplicate فرآیند والد نیست بلکه از همان اول با آدرس برنامه‌ای (باینری) که به سیستم کال createprocess می‌دهیم در RAM لود می‌شود بنابراین فرآیند فرزند و والد از همان لحظه به وجود آمدن فرزند می‌توانند دو برنامه کاملا متفاوت را بصورت همزمان اجرا کنند.

وقتی فرآیند فرزند به وجود می‌آید مستقیما توسط برنامه microsoft paint لود می‌شود

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

process termination

هر فرآیند به هر حال خاتمه پیدا می‌کند، حال یا بصورت عادی و یا اجباری، هیچ چیز حتی فرآیندها ابدی نیستند (تنسنبام)

خاتمه نرمال: خود فرآیند سیستم کال exit را فراخوانی می‌کند و در اینصورت فرآیند بصورت نرمال خاتمه پیدا می‌کند و OS هم منابعی که به آن داده است را ازش می‌گیرد، تمام منابع شامل حافظه فیزیکی و مجازی، فایل‌های باز، I/O buffers.

در نقطه‌ای که فرآیند سیستم کال exit را فراخوانی می‌کند فرآیند ممکن است یک status value (که معمولاً یک عدد صحیح است) را به فرآیند والدی که با استفاده از سیستم کال wait در انتظار خاتمه فرآیند فرزند مذکور است برگرداند.

یک فرآیند والد ممکن است با استفاده از فراخوان سیستمی wait برای terminate شدن هریک از فرآیندهای فرزندش منتظر بماند (wait شود) به فراخوان سیستمی wait پارامتر وضعیت پاس می‌شود که به والدین اجازه می‌دهد وضعیت خروج فرزند را بدست آورند، همچنین این سیستم کال شناسه فرزند خاتمه یافته را برمی‌گرداند تا والدین بتوانند بفهمند که کدامیک از فرزندانشان خاتمه یافته است.

abortion: اما اگر یک فرآیند توسط یک فرآیند دیگر خاتمه داده شود (یک فراخوان سیستمی برای خاتمه برنامه اجرا شود که این فراخوان سیستمی بصورت نرمال توسط والد اجرا می‌شود) (بعنوان مثال فراخوان سیستمی Terminate process در ویندوز) می‌گوییم فرآیند بصورت غیرعادی خاتمه یافته است و از واژه abort به معنای ختم غیرعادی استفاده می‌شود.

توجه) فرآیندهای والد اگر بخواهند فرزندان خود را terminate کنند باید شناسه آنها را بدانند بنابراین هنگامی که یک فرآیند، فرآیند جدیدی به وجود می‌آورد شناسه فرآیند جدید ایجاد شده به والد منتقل می‌شود

توجه) وقتی فرآیندی terminate می‌شود حالت اش به exit تغییر می‌کند و دیگر قابلیت اجرا شدن ندارد با این وجود ممکن است جدولهای دیگر اطلاعات مربوط به آن موقتا توسط سیستم عامل نگهداری شود، این کار به برنامه‌هایی که به اطلاعات این فرآیند نیاز دارند فرصت می‌دهد تا اطلاعات مربوطه را استخراج کنند، برای مثال یک برنامه accounting برای ثبت processor time و دیگر منابعی که فرآیند استفاده کرده برای مقاصد صورت حساب (که مثلا نداره به حساب پدرش) نیاز به زمان داشته باشد یا یک برنامه (utility program) برای تحلیل بهره وری و کارایی نیاز به زمان برای استخراج اطلاعات مربوط به تاریخچه فرآیند تمام شده داشته باشد، وقتی این برنامه‌ها اطلاعات مورد نیاز را برداشتند دیگر دلیلی ندارد که OS اطلاعاتی را از آن فرآیند نگهداری کند.

abortion می‌تواند به دلیل رخ دادن شکل در برنامه باشد و باعث ایجاد یک error trap شود که در اینصورت گاهی اوقات یک dump of memory در حافظه گرفته می‌شود (ایجاد می‌شود) و یک پیام خطا ایجاد می‌شود، Dump در یک special log file روی هارد نوشته می‌شود و ممکن است توسط Debugger که یک برنامه سیستمی است که برای کمک به برنامه نویس برای یافتن و تصحیح خطاها و اشکالات طراحی شده است بررسی شود تا علت شکل را مشخص کند.

abnormally ⇒ abort() ⇒ یک سیستم کال است

cascading termination (ختم آبشاری): وقتی پراسس خاتمه پیدا می‌کند تمام فرآیندهای زیرشاخه‌های آن فرآیند (تمامی نوادگان آن فرآیند: همه کسانی که زیرش هستند) به صورت آبشاری از پایین به بالا از بین می‌روند.

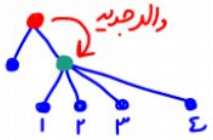
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

حالت‌هایی که پس از خاتمه یافتن یک فرآیند والد ممکن است رخ دهد

۱- کل زیر درخت مربوط به آن خاتمه پیدا کند ← cascading termination ← در VMS (OpenVms) در MainFrame

۲- لزومی ندارد که فرزندان خاتمه پیدا کنند ← در این حالت چون هر فرآیند باید یک فرآیند والد داشته باشد (به جز فرآیند ابتدایی سیستم) نیازمند است که عمل re-parenting انجام بشه که به این معناست که والد فرآیندی که قرار است terminate شود، والد جدید زیر درخت این فرآیند می‌شود.



اگر فرآیند سبز خاتمه پیدا کند، فرآیند قرمز والد جدید ۴ فرآیند مشخص شده است.

نکته) os های سنتی unix ریشه درخت فرآیند ها (system V init=init) را به عنوان والد جدید قرار می‌دهد، در این os ها فرآیند init به صورت پریودیک سیستم کال wait() را فراخوانی می‌کند در نتیجه با این کار وضعیت exit هم فرآیندهای یتیم را جمع آوری می‌کند و شناسه فرآیندهای یتیم و سطر process table مربوط به آنها آزاد می‌شود.

نکته) اکثر توزیع‌های linux، systemd را جایگزین init کردند، اگرچه در لینوکس عمل re-parenting می‌تواند روی systemd انجام شود (به جای init در unix) ولی linux همچنین به فرآیندهای دیگر نیز اجازه می‌دهد فرآیندهای یتیم را به ارث (inherit) ببرند و خاتمه آن فرآیندهای یتیم را مدیریت کنند

Zombie : زمانیکه اجرا منطقی فرآیند فرزند تمام می‌شود os منابع ای که به آن فرآیند اختصاص داده را از آن فرآیند می‌گیرد ولی سطر آن فرآیند در جدول فرآیند (PCB و process table اش) را فعلا پاک نمی‌کند تا زمانی که پدر آن wait کند که در آن صورت os آن فرآیند را بصورت فیزیکی نیز حذف می‌کند، در واقع چون جدول فرآیند (process table) حاوی وضعیت خروج فرآیند است و پدر به این وضعیت نیاز دارد تا متوجه شود آیا فرزندش کارش را با موفقیت انجام داده یا خیر و موارد دیگری از این دست، بنابراین entry جدول فرآیند مربوط به فرآیند فرزند تا زمانی که والد یک فرآیند سیستم کال wait را صدا می‌زند باید باقی بماند، وقتی والد wait را فراخوانی کند در wait وضعیت فرآیند فرزند به آن داده می‌شود و بنابراین os می‌تواند، PCB و pid فرآیند فرزند را از آن بگیرد فرآیندی که پایان یافته اما والد آن هنوز wait را فراخوانی نکرده بعنوان یک فرآیند زامبی شناخته می‌شود. همه فرآیندهایی که terminate می‌شوند وارد حالت زامبی می‌شوند اما عموما فقط برای مدت کوتاهی در حالت زامبی هستند.

Orphan (یتیم) : وقتی فرآیندی می‌میرد فرزندان آن یتیم خواهند شد، حالا برخی از این فرآیندهای یتیم می‌توانند زامبی هم باشند. فرض کنید فرآیند فرزندی داریم که exit کرده و زامبی است و منتظر است تا فرآیند والد سیستم کال wait را اجرا کند حالا فرض کنید فرآیند پدر قبل از آنکه wait را اجرا کند به هر ترتیبی خاتمه پیدا کرده، در این حالت فرآیند فرزند که زامبی است پدر ندارد پس یتیم است.

فرآیند پدر به ۳ دلیل ممکن است تصمیم بگیرد فرآیند فرزند را terminate کند
(که برای این کار اصطلاح abort یا خاتمه آبشاری بکار می‌رود)

۱- فرآیند فرزند از حدود برخی از منابعی که برایش در نظر گرفته شد تخطی کند (بیش از حد استفاده کند)

۲- دیگر به وظیفه و سرویسی که به آن محول شده نیازی نباشد

۳- فرآیند پدر در حال exit کردن است و os در صورت اتمام والد به فرزندانش اجازه ادامه کار را نمی‌دهد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

(process communication ارتباط بین پراسس ها)
systemcall های مربوط به ارتباط بین پراسس ها

دلایل فراهم کردن امکان همکاری فرآیندها

۱- به اشتراک گذاری اطلاعات : چند برنامه ممکن است به قطعه اطلاعات یکسانی علاقه مند باشند، مثل کپی و paste کردن یکسری از اطلاعات، بنابراین باید محیطی را فراهم کنیم که امکان دسترسی همزمان به چنین اطلاعاتی را مهیا کنیم

۲- افزایش سرعت محاسبات (computation speedup) : اگر بخواهیم یک کار خاص سریعتر انجام شود باید آن را به subtask های تقسیم کنیم که هرکدام موازی (parallel) با دیگران اجرا شود، البته این کار فقط روی کامپیوترهایی که چندین core دارند قابل انجام است، ترجیح کنید وقتی کاری را به کارهای کوچکتر می شکنید اینها برای انجام کارها یشان نیاز دارند باهم اطلاعات رد و بدل کنند.

۳- modularity : شاید بخواهیم سیستم را بصورت ماژولار بسازیم و عملکردهای سیستم را به فرآیندها یا نخ های مجزایی تقسیم کنیم.

خیلی وقت ها پراسس ها برای انجام کارها یشان باید از پراسس های دیگر کمک بگیرند و بصورت مستقل نمی توانند کارها یشان را انجام بدهند، بنابراین باید بتوانند با پراسس های دیگر در ارتباط باشند

- بطور کلی چند فرآیند یا می توانند بصورت مستقل از هم اجرا شوند که به آنها Independent process می گویند.
- " " باهمکاری باهم کارها را انجام دهند " " " cooperative "

- اگر پراسس ها مستقل از هم باشند به این معناست که :
۱- هیچ دیتای مشترکی (shared data) باهم ندارند.
۲- اجرا هر یک قطعی (Deterministic) و تکرارپذیر (repeatable) است.

- اگر پراسس ها همکار باشند به این معناست که :
۱- بین آنها دیتا مشترک وجود دارد (بصورت نرمال از طریق حافظه).
۲- روی هم اثر متقابل (mutual effect) دارند.
۳- اجرا غیر قطعی دارند = اجرای احتمالی Non-deterministic execution = stochastic execution
یا تکرار ناپذیر (non-repeatable)

mutual effect : چون بین پراسس ها حافظه مشترک وجود دارد و پراسس ها از طریق حافظه مشترک با هم در تعامل هستند، حال تغییر در این حافظه مشترک توسط یک فرآیند ممکن است باعث شود فرآیند دیگری که ممکن است تصمیم گیری اش برای انتخاب بین مسیر های اجرایی تحت تاثیر آن تغییر باشد، مسیر اجرایی خودش را عوض کند.

repeatable and Deterministic execution

هرزمان به یک برنامه یک ورودی مشخص بدهیم خروجی یکسانی تولید می شود و بنابراین اگر بارها همان برنامه، با همان ورودی ها اجرا شود خروجی یکسانی تولید می شود.

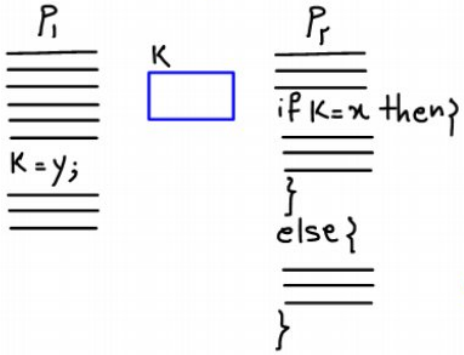
نکته) به دلیل آنکه فرآیندهایی از هم مستقل هستند اثر متقابل (mutual effect) روی هم ندارند اجرا این فرآیندها قطعی و تکرارپذیر است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Nonrepeatable and stochastic and Non Deterministic execution

اگر در هر بار اجرای فرآیندهای همکار دوره‌ای یکسان را به آنها بدهیم ممکن است نتایج متفاوتی را دریافت کنیم

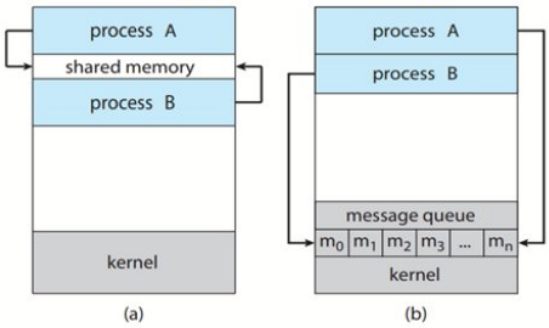


- فرض کنید P_1 و P_2 دو فرآیند همکار هستند که حافظه K بین این دو share شده و این دو فرآیند از این حافظه استفاده می‌کنند، حال فرض کنید در این حافظه مقدار x قرار گرفته است حال فرض کنید P_1 در خط ۱۰۰۰ برنامه‌اش می‌خواهد K را y کند و P_2 در خط ۵۰۰ می‌خواهد با توجه به مقدار K تصمیم بگیرد که چه مسیری را پیش بگیرد و ما هدف و برنامه‌ریزی‌مان به گونه‌ای بوده که P_1 زودتر از آنکه P_2 مقدار K را تغییر بدهد به if برسد، اما ممکن است به علت وقفه‌هایی که در سیستم به وجود می‌آید و ما از زمان آمدن آنها اطلاع نداریم و می‌توانند باعث توقف P_2 و رسیدن زمان به P_1 شوند و یا به دلیل randomness ای که ممکن است در تایمر سیستم وجود داشته باشد، فرآیند ۱ زودتر به خط ۱۰۰۰ خودش برسد و K را y کند و بنابراین فرآیند دو وارد else اش خواهد شد و مسیر دیگری را در پیش می‌گیرد.

همان طور که مشخص است با یکسری دوره‌ای ثابت برای این دو فرآیند ممکن است در اجراهای مختلف نتایج مختلفی حاصل شود و بنابراین اجرای فرآیندهای همکار repeatable نیست.

نکته) انجام دیباگ فرآیندهای همکار به همین دلیل مشکل است، زیرا ممکن است برنامه‌نویس با دارن برخی از دوره‌ای ها یکسری ایراداتی را مشاهده کند و آن ایرادات را به زعم خود رفع می‌کند، شاید آن ایراد را هم دیگر نبیند ولی آن ایراد وجود داشته باشد چون تکرار آن شرایطی که در موقع بروز مشکل رخ داده لزوما کار ساده‌ای نیست، برنامه‌نویس ممکن است بارها همان دوره‌ای ها را به برنامه بدهد ولی نتواند آن ایرادات را دوباره مشاهده کند

روش‌های ارتباط بین پراسس‌های همکار: بین پراسس‌ها به ۲ روش می‌توانیم ارتباط برقرار کنیم



Communications models. (a) Shared memory. (b) Message passing.

message passing_۱
shared memory_۲

Message-passing model(۱

پراسس‌ها با استفاده از ارسال و دریافت پیام با هم ارتباط برقرار می‌کنند

در این روش یک طرف فرستنده و یک طرف دیگر گیرنده است و فرستنده پیام‌اش را به همراه آدرس گیرنده در یک بسته قرار می‌دهد و بوسیله os آن را به سمت فرآیند گیرنده می‌فرستد.

توضیح شکل b: فرآیند A، n تا پیام به فرآیند B فرستاده است که اینها در یک صف قرار گرفتند که اینها توسط os به فرآیند B تحویل داده می‌شود، در ادامه به این موضوع بیشتر می‌پردازیم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

در این روش به پراسس فرستنده client و به پراسس گیرنده server می‌گوییم

نکته) در این روش ارتباط می‌تواند بین فرآیندهای روی یک سیستم باشد یا بین فرآیندهایی که از طریق شبکه می‌توانند با هم ارتباط برقرار کنند

نکته) قبل از برقراری ارتباط بین فرستنده و گیرنده یک connection باید باز شود

نکته) در شبکه هر ماشین یک ip منحصر به فرد دارد، همچنین هر فرآیند یک نام دارد (process name) و این نام به یک شناسه (identifier) ترجمه می‌شود که os می‌تواند بوسیله آن به فرآیند مراجعه کند، سیستم کال‌های get hostid() و get processid() این ترجمه را انجام می‌دهند، سپس بسته به مدل ارتباطی سیستم این شناسه‌ها به سیستم کال‌های general purpose = open() و close() (که توسط file system ارائه می‌شوند) یا به سیستم کال‌های خاص open connection() و close connection() داده می‌شوند، همچنین فرآیند گیرنده حتماً باید با سیستم کال accept connection() مجوز برقراری این ارتباط را بدهد.

نکته) اکثر فرآیندهایی که اتصالات را دریافت می‌کنند یک برنامه سیستمی روی سرور هستند (مخصوص آن کاربری که ما بعنوان کلاینت به سرور وصل می‌شویم) که سیستم کال wait for connection() را اجرا کردند (و بنابراین به خواب رفته‌اند، در حالت Block هستند) و هنگامی که یک اتصال برقرار می‌شود بیدار می‌شوند، که اصطلاحا این برنامه‌های سیستمی special-purpose daemons نامیده می‌شوند.
کسی که ارتباط را آغاز می‌کند کلاینت و daemon دریافت کننده سرور نامیده می‌شود، کلاینت و سرور از طریق سیستم کال‌های read message() و write message() به تبادل پیام با یکدیگر می‌پردازند و در نهایت نیز با سیستم کال close connection() ارتباط شان را خاتمه می‌دهند.

مزیت: مزیت این روش سادگی پیاده‌سازی این روش است و پراسس client فقط کافی است پیام را آماده کند و آدرس گیرنده را به آن بدهد

نکته) این روش برای انتقال داده‌های کوچک مناسب است، یعنی اگر سایز پیام‌ها از یک حدی بزرگتر شود چون باید یکبار پیام را از user mode به os بدهید و یکبار دیگر باید os این را به پراسس گیرنده بدهید که حداقل ۲ بار کپی انجام می‌شود که این موضوع باعث ایجاد سربار زیادی می‌شود

synchronization

message passing می‌تواند به ۲ صورت زیر باشد

۱- Blocking یا synchronous

۲- Nonblocking یا asynchronous

با توجه به اینکه فرآیند در روش message passing عمل send انجام می‌دهد یا receive خواهیم داشت:

□ send

Blocking send: فرآیندی که می‌خواهد عمل send را انجام بدهد تا زمانیکه مطمئن شود send انجام شده بلاک می‌شود (تا زمانیکه پیام توسط فرآیند گیرنده دریافت شود یا پیام در mailbox قرار گیرد.

نکته) هرگاه برای ما (فرآیند) مهم باشد که پیام‌ای که می‌خواهیم ارسال کنیم بدرستی ارسال بشود و تا زمانیکه فرآیند فرستنده از این موضوع اطمینان پیدا نکند نمی‌تواند به کارش ادامه دهد از این روش استفاده می‌شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

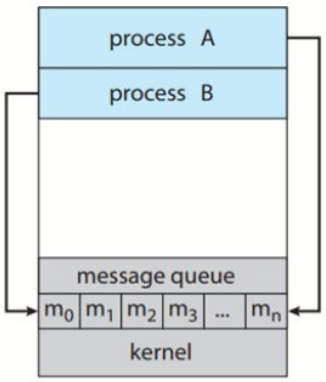
Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Nonblocking send : فرآیند فرستنده یک پیام را ارسال می‌کند و کارش را ادامه می‌دهد (منتظر این نمی‌ماند که ببیند ارسال اش با موفقیت انجام شده یا نه)

receive □

Blocking receive: گیرنده بلاک می‌شود (جلوی کارش رو می‌گیریم) و معطل می‌ماند تا زمانیکه پیامی را که منتظرش هست را دریافت بکند نکند) هرگاه فرآیند گیرنده بدون دریافت یک پیام نتواند به کارش ادامه دهد از این روش استفاده می‌شود

Nonblocking receive: گیرنده در زمانهایی به صف (یا mailbox) سر می‌زند و اگر پیامی برایش وجود داشت که برمی دارد و به کارش ادامه می‌دهد و اگر هم پیامی برایش وجود نداشت برایش Null برمی‌گردد و باز به کارش ادامه می‌دهد، پس در این روش فرآیند گیرنده برای آنکه به کارش ادامه دهد منتظر پیام مد نظرش نیست



-گفتیم فرآیند A ، n تا پیام به فرآیند B فرستاده است که اینها در یک صف قرار گرفتند که اینها توسط os به فرآیند B تحویل داده می‌شود، در این حالت ۲ نوع صف می‌تواند استفاده شود (پیاده سازی شود)

Buffering

۱- bounded capacity

۲- unbounded capacity

۱- صف محدود به اینصورت است که اندازه صف ثابت در نظر گرفته می‌شود، حالا اگر فرستنده پیام‌های زیادی را در صف قرار بدهد صف پر خواهد شد و در این حالت فرستنده باید صبر کند تا حداقل ۱ فضای خالی در این صف به وجود بیاید، پس در روش صف محدود در صورت خالی بودن صف مصرف کننده و در صورت پر بودن صف تولید کننده باید منتظر بماند

نکته) اگر سرعت برداشتن پیام‌ها از درون صف توسط گیرنده از سرعت قرار دادن پیام‌ها توسط فرستنده بیشتر باشد یک صف محدود را می‌توانیم بصورت یک صف نامحدود در نظر بگیریم (فرض کنیم) که در اینصورت هیچ گاه نیازنیست که فرآیند فرستنده منتظر برداشتن پیام توسط فرآیند گیرنده باشد

۲- صف نامحدود هیچ محدودیت عملی برای اندازه صف قرار نمی‌دهد (می‌توانید طول صف را بصورت بالقوه بی نهایت فرض کنید)، بنابراین هر تعداد پیام می‌تواند در صف منتظر بماند، در این شرایط مصرف کننده ممکن است برای itemهای جدید منتظر شود اما تولید کننده همیشه می‌تواند itemهای جدید را تولید کند بنابراین فرستنده هرگز بلاک نمی‌شود

۳- zero capacity : حداکثر طول صف ۰ است (هیچ بافری بین فرستنده و گیرنده وجود ندارد) بنابراین هیچ پیامی نمی‌تواند در صف منتظر بماند، در این حالت فرستنده باید تا زمانیکه گیرنده پیام را دریافت نکرده بلاک شود.

توجه) با توجه به سایز بافر Blocking می‌تواند سطوح مختلفی داشته باشد بعنوان مثال اگر send را در نظر بگیریم:

□ در zero capacity ، ارسال پیام از نوع Blocking send است چون فرآیند فرستنده وقتی پیام را ارسال می‌کند باید صبر کند تا فرآیند گیرنده آن پیام را تحویل بگیرد، پس zero capacity buffer به معنی Blocking send خواهد بود، در این حالت ما قوی ترین نوع Blocking را داریم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

□ درحالت bounded capacity که صف با ظرفیت محدود داریم، تا زمانیکه صف فرآیند گیرنده فضا دارد فرآیند فرستنده می‌تواند پیام‌های خودش را در صف قرار بدهد و به کارش ادامه دهد و در این شرایط نیازی نیست فرستنده بلاک شود اما اگر صف پر شود و فرآیند فرستنده هنوز پیام‌هایی برای ارسال داشته باشد چون صف گیرنده فضا ندارد فرآیند فرستنده باید بلاک شود تا حداقل ۱ فضای خالی در صف ایجاد شود، در این حالت فرستنده فقط باید مطمئن شود که پیام را در بافر گیرنده بگذارد و گیرنده موقع ارسال بطل اینکه گیرنده حتما دریافت کند که بعد فرستنده بتواند ارسال کند نمی‌شود پس در اینجا Blocking ضعیف‌تری داریم

□ درحالت unbounded capacity buffer فرآیند فرستنده بلاک نمی‌شود و برای فرستنده مهم نیست که چه زمانی گیرنده پیام‌هایش را برمیدارد.

Naming

فرآیندهایی که می‌خواهند با هم ارتباط برقرار کنند باید راهی برای رجوع (refer) به یکدیگر داشته باشند، از این حیث ۲ تفکر زیر وجود دارد

۱- Direct communication

۲- indirect " "

۱- در ارتباط Divect هر فرآیندی که می‌خواهد ارتباط برقرار باید صراحتا (explicitly) نام فرستنده یا گیرنده ارتباط را ببرد (در این روش فرستنده و گیرنده باید همزمان حضور داشته باشند، مثل ارتباط تلفنی)

یک لینک ارتباطی در این روش دارای ویژگی‌های زیر است:

۱- بین هر ۲ فرآیندی که می‌خواهند ارتباط برقرار کنند یک link بصورت خودکار به وجود می‌آید (فرآیندها برای برقراری ارتباط فقط نیاز به شناسه یکدیگر دارند)

۲- یک لینک مرتبط است دقیقاً با ۲ فرآیند

۳- بین هر جفت فرآیند دقیقاً یک پیوند وجود دارد

symmetry in addressing: گونه‌ای از روش Divect از تقارن در آدرس دهی استفاده می‌کند هم فرآیند فرستنده و هم فرآیند گیرنده باید نام دیگری را برای برقراری ارتباط ببرند

send(p, message): ارسال پیام به فرآیند p

receive(q, message): دریافت پیام از فرآیند q

asymmetric in addressing:

در این روش فقط فرستنده گیرنده را نام می‌برد و گیرنده نیازی به نام بردن فرستنده ندارد.

send(p, message)

receive(id, message): یک پیام از هر فرآیندی دریافت می‌کند، پس از ارتباط متغیر id به نام فرآیندی که ارتباط با آن انجام شده set می‌شود

- بطور کلی تکنیک‌های hard-coding که در آنها شناسه فرآیندها باید به صراحت بیان شود نسبت به تکنیک‌های غیرمستقیم کمتر مطلوب هستند

۲- indirect communication: شبیه ایمیل است و می‌توانید یک پیامی را از طریق ایمیل به گیرنده بفرستید درحالیکه فعلا گیرنده حضور ندارد.

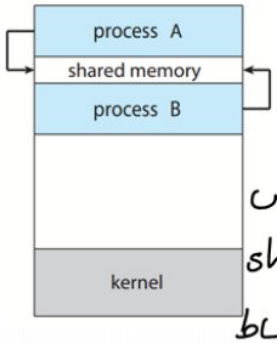
نکته) در روش ارتباط غیر مستقیم پیام‌ها از طریق mail box یا پورت‌ها ارسال و دریافت می‌شوند. هر صندوق پستی یک شناسه منحصر به فرد دارد.

نکته) یک فرآیند می‌تواند از طریق تعدادی صندوق پستی مختلف با یک فرآیند دیگر ارتباط برقرار کند، اما توجه داشته باشید که دو فرآیند تنها در صورتی می‌توانند با یکدیگر ارتباط برقرار کنند که صندوق پستی مشترک داشته باشند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



۲) shared_memory model

سربار روش بالایی را ندارد و سریعتر از روش تبادل پیام است.

پراسس ۱ و ۲ هرکدام حافظه خودشان را دارند که نسبت به همدیگر هم حفاظت می‌شود، حالا در این روش برنامه‌نویس (پراسس) با استفاده از سیستم کال‌های shared_memory_create() و shared_memory_attach() برای ایجاد و دسترسی به مناطق حافظه متعلق به سایر فرآیندها استفاده می‌کند، پس یک پراسس برای ارتباط می‌کند پس از موافقت، هر ۲ فرآیند این حافظه را جزئی از حافظه خودشون تلقی می‌کنند، پس این ۲ پراسس می‌توانند با خواندن و نوشتن داده‌ها در حافظه share شده به تبادل داده بپردازند

پراسس ۱ برای ارسال پیام به پراسس ۲، این پیام را در حافظه مشترک که حافظه خودش محسوب می‌کند می‌نویسد و به محض اینکه در این حافظه می‌نویسد این پیام به دست گیرنده رسیده است چون پراسس ۲ نیز این حافظه را متعلق به خودش می‌داند و بنابراین می‌تواند با مراجعه به این آدرس داده را بخواند

نکته ۱) به طور معمول os سعی می‌کند از دسترسی یک فرآیند به حافظه فرآیند دیگر جلوگیری کند بنابراین حافظه مشترک مستلزم آن است که دو یا چند فرآیند که می‌خواهند با هم ارتباط برقرار کنند با حذف این محدودیت موافقت کنند

نکته ۲) shared memory بالاترین سرعت ارتباط را فراهم می‌کند و زمانیکه در داخل یک کامپیوتر انجام می‌شود می‌توان این کار را با سرعت حافظه انجام داد، چون انتقال پیام از سمت فرستنده برابر نوشتن در حافظه خودش است و گرفتن پیام توسط گیرنده برابر خواندن از حافظه خودش است و بنابراین حداکثر یکبار پیام را در حافظه می‌نویسیم (یکبار بیشتر نیاز به نوشتن پیام نداریم)

نکته ۳) فرم و شکل داده‌ها (form of data) توسط فرآیندها تعیین می‌شود و تحت کنترل os نیست و os هیچ نقشی در مدیریت این حافظه مشترک برعهده ندارد چون os این حافظه را بین این ۲ پراسس حفاظت نمی‌کند و این ۲ پراسس هستند که باید مسئولیت حفاظت از این حافظه مشترک را برعهده بگیرند

نکته ۴) پراسس‌ها همچنین مسئول اطمینان از عدم نوشتن همزمان ۲ پراسس در یک مکان هستند

نکته ۵) بین پراسس‌هایی که با هم حافظه share کردن باید هماهنگ سازی (synchronization) وجود داشته باشد، یعنی پراسس ۱ و ۲ باید به این مورد توجه کنند که اگر یکی از پراسس‌ها در حال نوشتن در این حافظه است تا کار آن پراسس با حافظه تمام نشده دیگری از آن نخواند، توجه کنید که داریم از Time sharing استفاده می‌کنیم، به مثال زیر توجه کنید:

فرض کنید در یک قسمت از این فضای share شده یک آرایه به طول ۲۰ وجود دارد و P_1 باید همه درایه‌های این آرایه را در عدد c ضرب کند و در انتها P_2 از این آرایه برای کار خودش استفاده کند، حالا فرض کنید که P_1 تا عنصر ۱۰ این کار را انجام می‌دهد و سپس کوانتوم اش تمام می‌شود و نوبت P_2 می‌شود، حالا نباید P_2 بیاید همه این آرایه را بخواند و باید کاری کنیم که P_2 از این موضوع مطلع باشد و نتواند از این آرایه که عمل نوشتن در آن تکمیل نشده عمل خواندن را انجام دهد و کاری کنیم که P_2 تا زمانیکه P_1 نوشتن در این آرایه را تکمیل می‌کند صبر کند، یعنی با اینکه زمان می‌رسد به P_2 ولی P_2 نمی‌تواند از این حافظه بخواند و مجاز به این کار نیست که این کار همان بحث synchronization هست.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

posix shared memory as an example of Ipc systems (Ipc: interprocess communication)

گفتیم که در این روش برای آنکه ۲ فرآیند با هم ارتباط برقرار کنند باید قسمتی (تکه‌ای) از حافظه باید توسط OS بین ۲ فرآیند به اشتراک گذاشته بشود، بدین منظور از API های زیر که برای استاندارد posix است استفاده می‌کنیم.

۱- ابتدا یک پراسی باید تکه حافظه‌ای که می‌خواهد بصورت مشترک استفاده کند را از OS درخواست کند

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR)
```

با دستور shmget = shared memory get (API) درخواست یک حافظه با سایز مشخص را به OS می‌دهیم و اگر OS با این درخواست موافقت کند شناسه آن سگمنتی از حافظه که به این پراسی اختصاص داده برمی‌گرداند

۲- حال هر فرآیند که این شناسه در اختیارش قرار بگیرد می‌تواند با API = shmat = shared memory attach این سگمنت حافظه را به حافظه خودش Attach کند

```
shared_memory = (char*) shmat(id, null, 0)
```

این فرآیند می‌تواند مشخص کند که این حافظه را به چه عنوانی ازش استفاده کند، در واقع این فرآیند مشخص می‌کند که می‌خواهد چگونه به این حافظه نگاه کند. مثلاً در این مثال فرآیند گفته که می‌خواهد به حافظه با این id بصورت charactor pointer باشد

پس از اینکه این سگمنت حافظه به این فرآیند تخصیص یافت این فرآیند این حافظه را حافظه خودش می‌داند و بنابراین مانند دیگر حافظه‌های خودش روی این حافظه هم می‌تواند بصورت عادی بخواند و بنویسد، مثلاً این فرآیند می‌تواند با دستور زیر توی این حافظه بنویسد

```
sprintf(shared_memory, "Hi everybody")
```

حال فرآیندهای دیگری که بخواهند از این حافظه بصورت مشترک استفاده کنند با در اختیار داشتن شناسه این سگمنت می‌توانند با API = shmat این سگمنت حافظه را به حافظه خودشان پیوند بدهند و در اینصورت می‌توانند در این حافظه بخوانند و بنویسند

۳- هر وقت فرآیندی به این حافظه مشترک نیازی نداشته باشد با استفاده از API = shmdt = shared memory detach می‌تواند این حافظه مشترک را از حافظه خودش جدا کند

```
shmdt(shared_memory)
```

حال اگر تمامی فرآیندهایی که این حافظه را به خودشان Attach کردند این را detach اش کنند، OS می‌تواند این حافظه را پاک کند و آن را به حافظه‌های آزاد خودش اضافه کند.

نکته ۱) در سیستم‌های posix مکانیزم‌های دیگری از جمله message passing برای IPC وجود دارد.

نکته ۲) shared memory در posix براساس memory-mapped files پیاده سازی شده ← حافظه مشترک از طریق فایل‌هایی که در فضای آدرس چندین فرآیند نگاشت شده‌اند قابل دسترس است.

نکته ۳) سیستم کال‌های دیگری نیز برای این کار وجود دارد.

object برای خواندن و نوشتن باز است

```
fd = shm open(name, O_CREAT | O_RDWR, 0666); ⇒
```

یک فرآیند یک shared memory obj ایجاد می‌کند

پارامتر name نام " " را مشخص می‌کند

مجوزهای دسترسی به obj مشترک اگر این شی هنوز وجود نداره ایجاد شود

- اگر این سیستم کال موفق بود یک integer file descriptor برای آن shared-memory object برمی‌گرداند

- پس از ایجاد شی از این سیستم کال برای اینکه مشخص کنیم اندازه شی چند بایت باشد استفاده می‌شود

```
ftruncate(fd, 4.96);
```

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

یک memory mapped file حاوی shared-memory object تولید می‌کند و همچنین یک pointer به ... که برای دسترسی به shared-memory object استفاده می‌شود را return می‌کند.
mmap(....);

مثال) در کد زیر یک مدل producer-consumer را می‌بینید که از حافظه مشترک استفاده می‌کنند که در آن تولید کننده یک shared-memory object ایجاد می‌کند و در حافظه مشترک می‌نویسد و مصرف کننده از این حافظه مشترک می‌خواند.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* create the shared memory object */
    fd = shm_open(name,O_CREAT | O_RDWR,0666);

    /* configure the size of the shared memory object */
    ftruncate(fd, SIZE);

    /* memory map the shared memory object */
    ptr = (char *)
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Producer process illustrating POSIX shared-memory API.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = (char *)
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Consumer process illustrating POSIX shared-memory API.

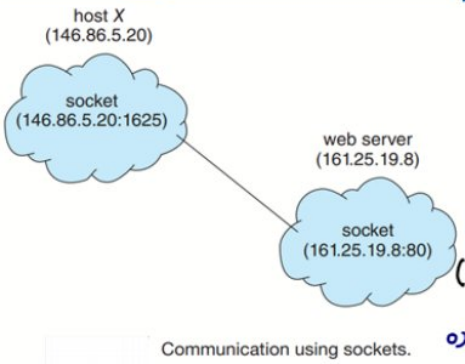
communication in client-server system

از روش‌های shared-memory و message passing می‌توان برای ارتباطات در سیستم‌های client-server استفاده کرد ولی در اینجا میخوام چند مدل دیگرش رو توضیح بدم:

۱- sockets
۲- (Remote Procedure call) RPC
۳- (Remote Method Invocation) RMI

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



Communication using sockets.

sockets

یک مکانیزم message passing هست که با استفاده از آن می‌توانیم بین ۲ فرآیند یک ارتباط کلاینت سرور برقرار کنیم

سوکت: یک سوکت با جفت IP و شماره پورت مشخص می‌شود مثلاً: (۱۴۲.۱۷۰.۸.۲۵:۲۵۱۶)

یک جفت فرآیند که از طریق شبکه با هم ارتباط برقرار می‌کنند از یک جفت سوکت استفاده می‌کنند.

سرور با گوش دادن به یک پورت مشخص منتظر درخواست‌های مشتری‌های ورودی می‌شود و پس از دریافت درخواست سرور باید درخواست اتصال کلاینت را قبول کند تا اتصال ایجاد شود (راستان HTTP و FTP و SSH) و بین آنها ارتباط شکل می‌گیرد.

نکته) وقتی از سوکت برای ارتباط بین فرآیندها استفاده می‌کنیم یک unstructured stream of bytes بین کلاینت و سرور برقرار می‌شود و این بدان معناست که هیچ استانداردی برای محتوای پیام این بسته‌ها نداریم و فقط کلاینت و سرور می‌دانند که محتوایی که از طریق آن سوکت رد و بدل می‌شود چیست، در برخی از روش‌ها باید پیامی که رد و بدل می‌شود structured و استاندارد باشد تا طرفین بتوانند به ارتباط با هم بپردازند

RPC (Remote Procedure call)

سرویس است که کرنل آن را به کاربر عرضه می‌کند

یکی دیگر از مکانیزم‌های message passing است که کاربردش برای زمانی است که یک فرآیند روی یک کامپیوتر می‌خواهد یک procedure که روی یک سرور (با نام RPC daemon) راه دور است را فراخوانی کند، بنابراین سرور روی یک پورت مشخص در حال گوش دادن است.

پیامی که کلاینت به سمت سرور می‌فرستد شامل محتوای زیر است:

۱- شناسه تابعی که می‌خواهد فراخوانی کند.

۲- پارامترهایی که باید به آن تابع بدهد: parameter marshaling

نکته) برای مجموعه یک شده این آرگومان‌ها از اصطلاح marshaling استفاده می‌شود

نکته) در RPC پیام‌ها استاندارد و structured هستند به این معنی که پیام‌ها از یک ساختار خاص بایت‌ها باید برخوردار باشند و ساخت یافته باشد، یعنی باید کاملا برای سرور مشخص باشد که کدامیک از اجزای آن پیام شناسه تابع فراخوانی شوند و کدامیک از اجزای آن پارامترهایی است که ارسال شده.

وقتی پیام کلاینت به سرور رسید، سرور محاسبات درخواست شده را انجام می‌دهد و پاسخ را در قالب یک پیام به کلاینت برمی‌گرداند

روش‌های پیدا کردن پورت سرویس مورد نظر روی سرور توسط کلاینت در RPC

۱- predetermined binding: از قبل برای فرستنده مشخص است که پورت مربوط به سرور RPC چه چیزی است و کلاینت بسته‌هایش را به این پورت ارسال می‌کند و به این صورت است که وقتی یک برنامه که توش از RPC call استفاده شده میخواد کامپایل شده در زمان کامپایل یک شماره پورت ثابت به اون RPC call اختصاص داده میشه، خب مشخصه که چون تری این روش کلاینت شماره پورت مقصد رو داره و نمی‌خواد دنبالش بگرده overhead اش پایینه ولی یک مشکلی داره:

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۱- سرور نمی‌تواند شماره پورت سرویس درخواستی را تغییر دهد ← سرور دیگه نباید این پورت رو عوض کند

۲- اگر یک روزی شماره پورت سرور تغییر بکند چی میشه؟ داستانی میشه، کلاینت‌ها باید از این موضوع مطلع بشوند و کدهای روی کلاینت‌ها را باید مجددا کامپایل کنیم

۲- ,, Dynamic : در زمان کامپایل پورت سرور RPC به RPC call داده نمی‌شود و برای بدست آوردن شماره پورت RPC daemon از یک سرور دیگر بنام matchmaker daemons/rendezvous (rendezvous)(قرار/مکان ملاقات) استفاده می‌کنیم که شبیه همون DNS توی شبکه خودمون است، بهش می‌گیم من فلان سرویس را می‌خوام و اون میاد IP و پورت سرور را بهمون میده. خب این روش flexible تر است ولی خب overhead اش نیز بالاتر است.

مثال) یکی از ایده‌های رایانش ابری همینه، فرض کنید ما نیاز داریم با گوشی موبایلمان می‌خواهیم یک کاری انجام دهیم که نیاز به حافظه زیاد یا قدرت پردازشی زیاد و یا هر ۲ داشته باشد مثلاً میخواید یک متن فارسی رو ببرید و یک متن انگلیسی تحویل بگیرید یا میخواید یک برنامه بزرگ را اجرا کنید و خروجی شو تحلیل کنید، خب گوشی ما نه حافظه زیادی داره که ما بتوانیم دیتابیس مورد نیاز برای ترجمه را روی گوشی‌مان نگه داریم و نه قدرت پردازشی خیلی زیادی داره که بتونه یک برنامه سنگین را اجرا کنه، در این حالت فقط کافی است سرویس ترجمه را که روی یک سروری قرار گرفته با پارامترهای مناسب فراخوانی کنیم، مثلا بگیم زبان مبدا مان فارسی است و زبان خروجی انگلیسی هم می‌خوام و داده را هم براش مشخص کنم، اگر آن سرور چندتا کار انجام میده باید مشخص کنم که کدوم کارش رو می‌خوام.

یادآوری) وقتی یک فرآیند یک procedure را فراخوانی می‌کند باید پارامترهایی که باید به procedure داده شود را روی پشته قرار دهد و بعد procedure فراخوانده شده این پارامترها را از روی پشته بر می‌دارد و پس از انجام محاسبات اش نتیجه را روی پشته می‌گذارد و پس فراخوانده می‌تواند این نتیجه را از روی پشته بردارد و ازش استفاده کند، حالا توی RPC همین اتفاق از راه دور می‌افتد

نکته) ۲ فرآیند روی یک ماشین هم می‌توانند از مکانیزم RPC برای ارتباط استفاده کنند، یعنی روی یک ماشین ولی با ۲ پورت مختلف از این استفاده کنند و با هم داده رد و بدل کنند

مراحل انجام RPC:

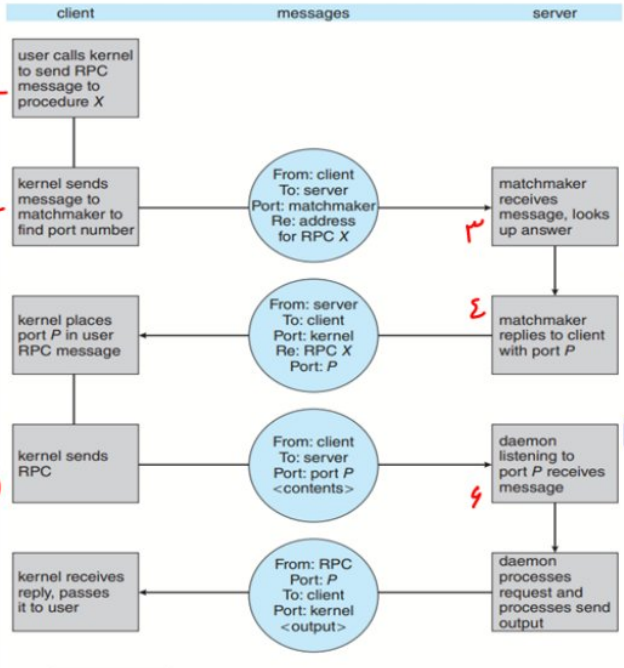
در این مثال کلاینت و سرور روی یک کامپیوتر هستند.

سرور روی پورت مربوط به خودش در حال listening است.

۱- فرآیند کاربر به کرنل می‌گوید که می‌خواهد با ارسال پیام RPC، procedure x را فراخوانی کند

۲- کرنل نمی‌داند که سرور مربوط به RPC x روی چه پورتی گوش می‌کند یک پیام از کلاینت به سرور matchmaker می‌فرستد و می‌گوید من آدرس RPC x را می‌خواهم

۳و۴- matchmaker پس از دریافت این پیام در جداول خودش سرچ میکند و پورت P را بعنوان پاسخ در یک پیام برای کرنل ارسال می‌کند



Execution of a remote procedure call (RPC).

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۵- کرنل متوجه می‌شود که باید پیام را برای پورت p ارسال کند (p پورت سرویس دهی سرور RPc است)، بنابراین پیامی ایجاد می‌کند و در آن شناسه‌ای که باید فراخوانی شود و پارامترهایی که باید به procedure = x داده شود را در پیام می‌گذارد و آن را برای پورت p ارسال می‌کند (با RPc daemon تماس می‌گیرد)

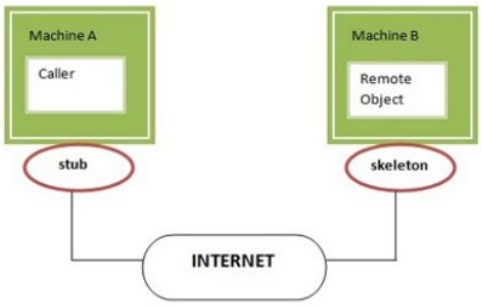
۶- RPc daemon که دارد روی پورت p گوش میدهد پس از دریافت این پیام procedure = x را پراسس می‌کند و پس از بدست آوردن پاسخ آن را توسط یک پیام برای کرنل ارسال می‌کند تا کرنل پاسخ فرآیند فراخواننده را بدهد

نکته) فرآیند فراخواننده متوجه نمی‌شود که procedure = x کجا هست و متوجه نمیشه که سرور روی همین کامپیوتر بوده یا روی یک کامپیوتر روی شبکه بوده و همه این کارها توسط سرویس RPc ارائه شده است.

RMI: Remote Method Invocation: فراخوانی متد از راه دور

اگر مکانیزم RPc را بخواهیم بصورت object oriented استفاده کنیم به جای procedure call از method invocation استفاده می‌شود یک متد روی یک سرور وجود دارد و ما می‌خواهیم از طریق یک client آن را فراخوانی کنیم. RMI یک API است که مکانیزمی را برای ایجاد برنامه‌های توزیع شده در جاوا فراهم می‌کند، RMA ارتباط از راه دور برنامه‌ها را با استفاده دوشی stub و skeleton فراهم می‌کند

stub: (یک شی است) بعنوان یک gateway برای سمت مشتری عمل می‌کند که تمامی درخواست‌های خروجی مشتری از طریق آن ارسال می‌شود وقتی مشتری یک متد را فراخوانی می‌کند stub کارهای زیر را انجام می‌دهد:



۱- اتصال با JVM راه دور را ایجاد می‌کند

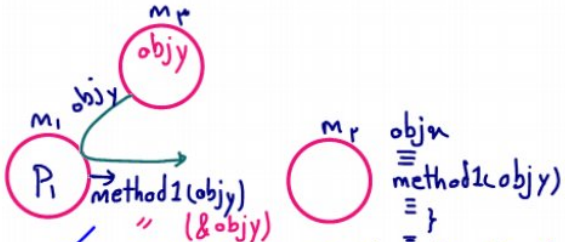
۲- پارامترها را به JVM راه دور ارسال می‌کند (marshaling)

۳- منتظر نتیجه از سمت سرور می‌ماند

۴- مقدار return شده را می‌خواند (unmarshaling)

۵- مقادیر را به کلاینت برمی‌گرداند

skeleton (یک شی است) بعنوان یک gateway سمت سرور عمل می‌کند و تمام درخواست‌های دریافتی از طریق آن دریافت (route) می‌شوند وقتی skeleton پیامی را دریافت می‌کند و کارهای زیر را انجام می‌دهد



۱- پارامترهای پیام را برای متد راه دور می‌خواند

۲- متد را فراخوانی می‌کند

۳- نتیجه را از متد می‌گیرد و برای کلاینت می‌فرستد (marshaling)

object serialization: P_1 که روی ماشین ۱ است ممکن است بخواهد متدی که برای یک object روی ماشین ۲ است را فراخوانی کند و ممکن است پارامتری که می‌خواهیم به این متد بدهیم خودش یک object باشد که روی ماشین ۳ است، حال به ۲ روش می‌توان عمل کرد

۱- ماشین ۱ object ای که می‌خواهیم به متد ماشین ۲ بدهیم را از ماشین ۳ بگیریم و بعد ماشین ۱ این object را بصورت یک استریمی از بایت‌ها (object serialization) به متد روی ماشین ۲ بدهیم، در این روش ما یک obj بزرگ را ۲ بار روی شبکه جابجا کردیم که سربارش زیاد است

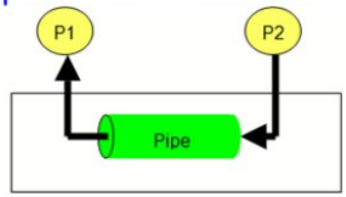
۲- call/pass by reference: آدرس obj روی ماشین ۳ را در اختیار obj روی ماشین ۲ که متدش را می‌خواهیم فراخوانی کنیم قرار بدهیم و بنابراین متد روی ماشین ۲ اگر به obj روی ماشین ۳ نیاز داشت خودش می‌تواند با object serialization، obj را بگیرد که ۱ جابجایی کمتر میشه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

pipes

یک pipe بعنوان یک مجرا عمل می‌کند و اجازه میده که ۲ فرآیند با هم ارتباط برقرار کنند، pipe یک مکانیزم است که اجازه می‌دهد فرآیندها با هم ارتباط برقرار کنند. یک pipe دو طرف دارد به طوریکه اگر از یک طرف اش پیامی را وارد کنید از سمت دیگر حتما آن پیام دریافت می‌شود. pipe ها یکی از اولین مکانیزم‌های IPC در سیستم‌های اولیه unix بودند



در پیاده سازی pipe ها ۴ سوال مطرح است.

۱- آیا pipe یک ارتباط دوطرفه (bidirectional) را برقرار می‌کند یا یک ارتباط یکطرفه (unidirectional)

۲- اگر ارتباط ۲ طرفه بود آیا این ارتباط half duplex است یا full duplex ؟
(۲ طرفه است اما در لحظه ۱ طرفه است)

۳- آیا بین فرآیندها باید یک رابطه پدر و فرزندی وجود داشته باشد که این‌ها بتوانند با هم از طریق pipe ارتباط برقرار می‌کنند

۴- آیا برای استفاده از pipe فرآیندها باید روی یک سیستم باشند یا روی شبکه هم قابل استفاده هستند

ما ۲ نوع pipe داریم
۱- unix : ordinary pipes / windows : anonymous pipes
۲- unix : FIFO / window : Named pipes

اگر از نوع ساده و پایه پایپ استفاده کنیم پاسخ سوالات بالا منفی است و اگر از نوع پیشرفته pipe استفاده کنیم پاسخ‌شان مثبت است.

ordinary pipes

نکته) در unix پایپ معمولی با دستور pipe(fd) = pipe(int fd[]) ساخته می‌شود که از طریق fd[0] و fd[1] قابل دسترس است fd[0] انتهای جایی که از لوله خوانده شده و fd[1] انتهای جایی که نوشته شده است را نشان می‌دهد.

پایپ در ویندوز با دستور createpipe() ساخته می‌شود.

نکته) unix یک پایپ را بعنوان یک فایل خاص در نظر می‌گیرد بنابراین براحتی می‌توان با توابع سیستمی معمولی read() و write() از پایپ خواند یا در آن نوشت (به پایپ دستری داشت)، در ویندوز خواندن و نوشتن در پایپ با توابع معمولی ReadFile() و WriteFile() است.

- یک پایپ معمولی خارج از فرآیندی که آن را ایجاد کرده قابل دسترس نیست، بطور معمول یک فرآیند والد یک پایپ ایجاد می‌کند و از آن برای برقراری با فرزندش که از طریق fork() ایجادش کرده استفاده می‌کند، به این نکته توجه کنید که فرآیند فرزند فایل‌های باز را از والدخود به ارث می‌برد و از آنجایی که pipe نوع خاصی از فایل است فرآیند فرزند پایپ را از پدرش به ارث می‌برد

نکته) برخلاف unix که فرزند بطور خودکار پایپی که والد ایجاد کرده را به ارث می‌برد ویندوز از برنامه نویس می‌خواهد که مشخص کند که فرآیند فرزند کدام ویژگی‌ها را به ارث می‌برد

خواص لوله‌های معمولی

۱- یکطرفه هستند : یک فرآیند تولید کننده و یک فرآیند مصرف کننده است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) درصورت نیاز به ارتباط ۲ طرفه باید از ۲ پایپ معمولی استفاده کنیم که هر پایپ داده‌ها را در جهت متفاوتی ارسال می‌کند

۲- پایپ‌های معمولی به یک رابطه پدر و فرزندی بین فرآیندهای ارتباط‌طی در هر ۲ سیستم unix و windows نیاز دارند و در نتیجه این پایپ‌ها فقط برای ارتباط بین فرآیندها در یک ماشین قابل استفاده است.

۳- در نسخه معمولی پایپ اگر یکی از ۲ طرف ارتباط از بین بروند آن پایپ نیز از بین رفته محسوب می‌شود ولی در named pipe، پایپ‌ها برای خودشون یک موجودیت در ده هستند و شناسه دارند و اگر ۲ فرآیندی که از یک named pipe استفاده می‌کنند از بین بروند هنوز این پایپ می‌تواند در سیستم باقی بماند به نحوی که جدا فرآیندهای دیگری می‌توانند از این پایپ استفاده کنند و از طریق این پایپ ارتباط شون را برقرار کنند

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Greetings";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
   }

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }

   if (pid > 0) { /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
   }
   else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("read %s",read_msg);

      /* close the read end of the pipe */
      close(fd[READ_END]);
   }

   return 0;
}
```

برخی خواص Named pipes

۱- ارتباط می‌تواند ۲ طرفه باشد و نیازی به رابطه پدر و فرزندی وجود ندارد

۲- چندین فرآیند می‌توانند از یک named pipe برای ارتباط استفاده کنند در واقع در یک سناریوی معمولی یک named pipe چند نویسنده دارد

نکته) Named pipe ها در سیستم‌های unix بعنوان FIFO شناخته می‌شوند و پس از ایجاد بعنوان یک فایل معمولی در نظر گرفته می‌شوند یک FIFO با فراخوانی mkfifo() ایجاد و با فراخوانی‌های سیستمی read() و write() و close() و open() مورد استفاده قرار می‌گیرد و تا زمانی که صراحتا از file system حذف نشوند به وجود خود ادامه می‌دهند.

نکته) اگرچه FIFO اجازه ارتباط bidirectional را می‌دهد اما فقط امکان ارتباط half duplex را می‌دهد و اگر بخواهید داده‌ها هم‌زمان در ۲ جهت منتقل شوند باید از ۲ تا FIFO استفاده کنید

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۳- در unix فرآیندهایی که می‌خواهند با هم ارتباط برقرار کنند باید روی یک ماشین باشند و اگر فرآیندها روی ماشین‌های متفاوتی باشند باید از socket استفاده کنیم

۴- در Named pipe در ویندوز هم full duplex مجاز است و همین طور فرآیندها می‌توانند روی ماشین‌های متفاوتی باشند

۵- در unix FIFO داده‌ها فقط بصورت byte-oriented می‌توانند منتقل شوند ولی در ویندوز هم می‌تواند byte oriented و هم message oriented باشد.

- در ویندوز این پایپ با دستور createNamedpipe() ایجاد می‌شود و یک کلاینت می‌تواند با connectNamed pipe() به یک Named pipe وصل شود و با استفاده از readFile(), writeFile() می‌توان با پایپ مذکور ارتباط برقرار کرد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

thread (نخ، ریسه، ریسمان)

ما تا الان در مورد فرآیندهای تک نخی (single threaded) صحبت می‌کردیم به چنین فرآیندی Heavyweight process می‌گویند، در این فصل می‌خواهیم در مورد فرآیندهای multithreaded (چند نخی) صحبت کنیم.

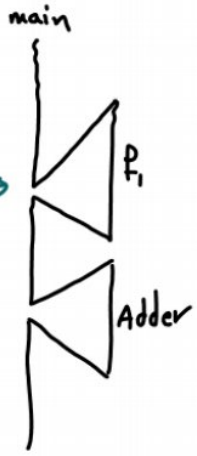
داستان ترد

از pthread library استفاده کردیم و در این دستور اشاره‌گر به تابعی (و آرگومان‌های ورودی ای که می‌خوایم با آن اجرا بشه) که می‌خواهیم به یک ترد تبدیل اش کنیم که بصورت همروند با ترد main اجرا بشود را می‌دهیم.

single thread

```
main() {
    ≡
    A = f1(....);
    ≡
    B = Adder(....);
    ≡
    f1(....) {
    ≡
    }
    Adder(....) {
    ≡
    }
```

وقتی این را بصورت عادی call می‌کنیم



multithread

```
main() {
    ≡
    pthread_create(f1, argvf1);
    ≡
    pthread_create(Adder, argvAdder);
    ≡
    f1(....) {
    ≡
    }
    Adder(....) {
    ≡
    }
```

حال main می‌رود روی دستور بعدی اش و به اجرا برنامه اش ادامه می‌دهد و این ترد هم وارد ماجرای time slice گرفتن میشه و همروند با main اجرا می‌شود

نکته ۱) pthread یا posix thread یک thread library است که می‌توانیم توسط آن ترد ایجاد کنیم.

" ۲) همه تردهای مربوط به یک فرآیند از کد، داده و رسورس های پراس بصورت اشتراکی استفاده می‌کنند.

" ۳) پس مشخص است که وقتی ما یک فرآیند multithread داشته باشیم می‌شود بخشی از کد یک ترد اجرا شود و بعد که کوانتوم اش تمام شد و کوانتوم به ترد دیگری رسید بخشی از کد یک ترد دیگر اجرا شود، یعنی دقیقاً همان اتفاقی که بین فرآیندها رخ میداد در این روش بین تردهای یک پراس رخ می‌دهد، بنابراین مشخص است که وقتی time slice یکی از تردهای فرآیند تمام شد و خواستیم به یک ترد دیگر switch کنیم باید اطلاعات مربوط به آن ترد را ذخیره کنیم که بعدا بتوانیم آن را از جای قبلی اش ادامه اش دهیم

نکته) پس مادامیکه کارهایی که گفتم را برنامه نویس انجام ندهد فرآیند ما single thread است، بنابراین در حالت عادی تابع main که خودش می‌تواند توابع دیگری را در بدنه خود صدا بزند (call کند) یک ترد است و تمامی توابعی را نیز که صدا می‌زند در همان یک ترد اجرا می‌شوند، اما ما (برنامه نویس) می‌توانیم در بادی main تصمیم بگیریم یک تابع جدیدی را که می‌خواهیم فراخوانی کنیم بصورت یک ترد جدید درش بیاریم (یک موجودیت جدید) (که برای این کار باید به نحوی که در بالا شرح دادم از pthread library استفاده کنیم) که در اینصورت ترد مربوط به تابع main و این ترد جدید می‌توانند بصورت همروند (concurrent) اجرا شوند، اگر هم چند core داشته باشیم که موازی (parallel) هم می‌توانند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

توجه) فرآیند اصلی شما با main شروع می‌شود، وقتی یک درخواست اجرای یک فرآیند داده می‌شود (مثلاً با سیستم کال fork در یک خط از برنامه یا ...) OS به آن فرآیند PCB و حافظه اختصاص می‌دهد و فرآیند ایجاد می‌شود و در RAM لود می‌شود و وارد صف ready می‌شود حالا هروقت OS بخش time slice بدهد، پردازنده خط اول main را اجرا می‌کند.

توجه) در برخی از زبانهای برنامه نویسی ممکن است دقیقاً واژه main را نداشته باشیم و این بدان معنا نیست که در آنها مفهوم ترد و توانایی ایجاد آن را نداشته باشیم.

Multithreaded C program using the Pthreads API

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

باید یک متغیر برای ترد در نظر بگیریم که شماره شناسایی ترد در آن قرار بگیرد

attr و attribute های ترد را در بر می‌گیرد و پس این ها به یکسری مقادیر مقدار دهی اولیه می‌شود

اشاره‌گر به یک تابع و آرگومان های دوردی آن تابع را داریم به pthread_create

*۱ تا قبل از اینکه خط pthread_create نیامده، ما یا ترد داریم و ترد جدیدی تولید نمی‌شود و تا اینجا این قسمت از برنامه یک قسمت سریال است و قابل موازی سازی با قسمت دیگری نیست

ولی بعد از اجرا خط pthread_create(runner) ما دو تا ترد داریم که این دو می‌توانند همروند یا موازی با هم پیش بروند.

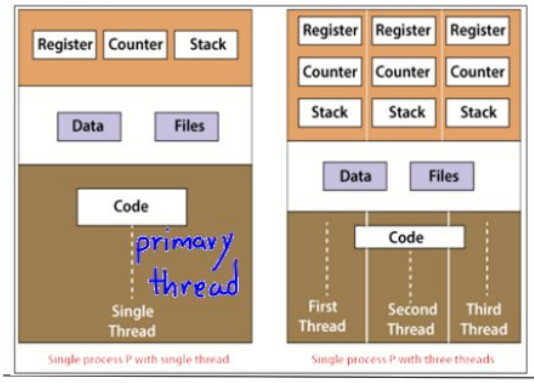
این صحبت های من شما رو یاد چی میندازه؟ قانون آمدال که کمی جلوتر توضیح اش میدهم.

*۲ دستور pthread join: این دستور باعث میشه که ترد main برنامه در این خط معطل می‌شود تا ترد دومی که ساخته اجرایش تمام بشود و بعد این ۲ ترد با هم join می‌شوند و از آنجایه بعد با یک ترد اجرا برنامه ادامه پیدا می‌کند، یعنی بعد از این اتفاق ما single thread داریم و برنامه بعد از اجرا دستور printf خاتمه پیدا می‌کند

join

*۳ pthread_exit: توجه کنید که ما پایان تابع runner نگفتیم exit و گفتیم pthread_exit، اگر exit می‌نوشتیم کل فرآیند خاتمه پیدا می‌کرد نه فقط این ترد، یعنی این کار باعث می‌شد که شاید ترد۲ هم که شاید وسط اجرایشان باشند قطع بشوند، توجه کنید که exit یک سیستم کالی بود که به OS می‌گفت این برنامه/فرآیند خاتمه پیدا کرده

نکته) ما Thread library های مختلفی داریم که ۳ مورد آن:
1) pthreads
2) win32 threads
3) java threads



- وقتی روی یک برنامه دبل کلیک می‌کنید ابتدا یک پراس روت می‌شود و بعد یک ترد اولیه بنام primary thread ایجاد میشه که برنامه را اجرا کند
اولین ترد که روت میشه اسم اش primary thread است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Amdahl's Law

S : هر برنامه ای یکسری از قسمت هاش حتما سریال است و باید روی یک core اجرا بشه و این قسمت ها قابل موازی سازی نیست.
اگر S بخشی از برنامه باشد که باید روی یک core انجام بشود و اگر N تا core داشته باشیم و برنامه ما N تا ترد داشته باشه زمان اجرا این برنامه بصورت زیر بدست می آید

زمان اجرا روی N تا core = $S t + \frac{(1-S)t}{N}$

اگر با یک core برنامه t واحد طول بکشد خواهیم داشت:

$$speedup \leq \frac{t}{St + \frac{(1-S)t}{N}} = \frac{1}{S + \frac{1-S}{N}}$$

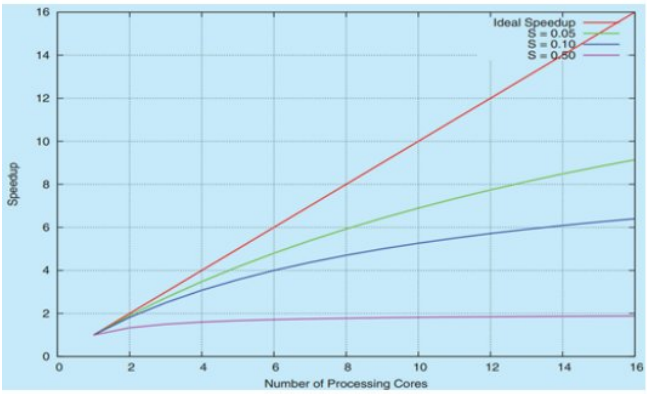
if $S=1 \rightarrow$ speed up $=1$
if $S=0 \rightarrow$ " $=N$

توجه کنید که می تواند $1 \geq S \geq 0$ باشد

$S=1$: هیچ قسمت برنامه قابل موازی سازی نیست

$S=0$: بتوانیم همه جاشو موازی کنیم

توجه) قسمت موازی برنامه را نمی توانیم به هر تعدادی ترد تبدیل اش کنیم و بنابراین اضافه کردن تعداد core ها تا یک حدی میتونه به ما کمک کند و speedup را ببره بالا مانند حالتی که وقتی ما یک برنامه تک رده داشته باشیم افزایش تعداد core ها به ما کمکی نمی کند، پس مشخصه که برای speedup گرفتن روی یک برنامه نیاز به ایجاد تردهای مختلف داریم تا اینها را روی core های مختلف ایجاد کنیم



اگر ما تعداد تردهایمان را از تعداد core هایی که داریم بالاتر ببریم، خب core اضافه ای نداریم که به آن تردها اختصاص بدهیم، بنابراین تردهای اضافی باید در صف ready مربوط به هر core منتظر بمانند تا زمانیکه یک کوانتوم بهش اختصاص دهیم.

مثال) فرض کنید یک برنامه که دارای $S=0.2$ است قرار است روی یک پردازنده با ۴ تا core اجرا شود، حداکثر speedup را بدست آورید

$$speedup \leq \frac{1}{0.2 + \frac{0.8}{4}} = \frac{10}{4} = 2.5$$

اگر اگر اینجا تعداد core را ببریم به بینهایت، speedup = ۵ می شود

پس چه چیزهایی از یک ترد باید نگهداری شود؟

۱- Thread ID

۲- pc : باید مشخص باشد ترد تا چه دستوری از تابع خودش را اجرا کرده و جلو برده که وقتی به این ترد برگشتیم ادامه اش رو اجرا کنیم.

۳- register هایی که تغییر داره : وقتی برگشتیم به این ترد با همان مقادیر قبلی که بدست آورده ادامه کدهاش رو اجرا کنیم.

۴- وضعیت هر ترد : running ، Block ، ready

۵- هر ترد باید استک خودش را داشته باشد. چرا؟

از استک برای اینکه بتوانیم در یک فرآیند (برنامه) فراخوانی تو در تو داشته باشیم استفاده می کردیم، مثلا اگر ترتیب فراخوانی یک برنامه بصورت زیر باشد، داده های هرکدام از این توابع بصورت زیر در استک قرار می گرفتند

ترتیب فراخوانی = $f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4 \dots$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

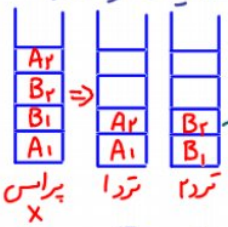
حالا فرض کنید ترد‌های ۱و۲ هر کدام به ترتیب فراخوانی ‌های زیر را انجام بدهند

ترتیب فراخوانی ترد ۱ : $A_1 \rightarrow A_2$

۲ " " : $B_1 \rightarrow B_2$

حال فرض کنید برای کل فرآیند فقط یک پشته می داشتیم، حال سناریو زیر را در نظر بگیرید

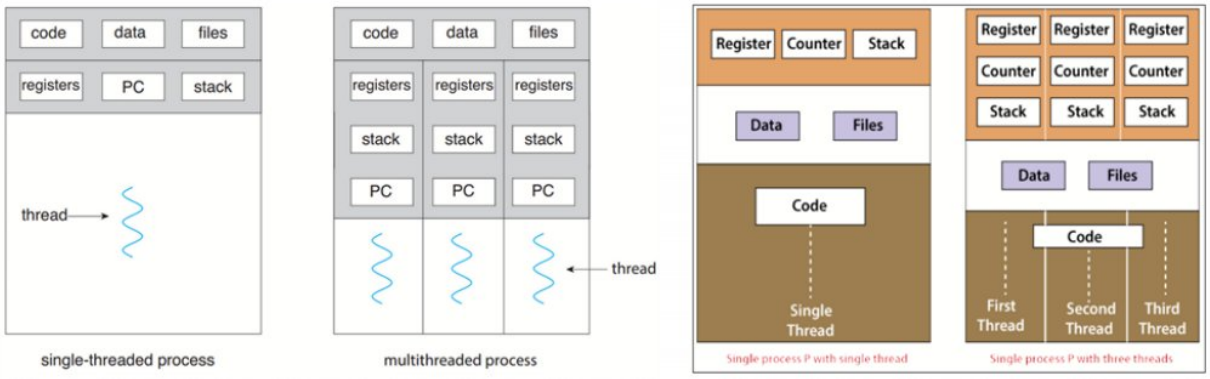
ابتدا زمان دست ترد ۱ باشد و A_1 را فراخوانی کند و بعد زمان دست ترد ۲ بیفتد و B_1 را صدا کند و سپس B_2 را صدا کند و قبل از اینکه اجرا B_2 تمام شود زمان دست ترد ۱ بیفتد و A_2 را صدا کند و قبل از تمام شدن A_2 زمان به ترد ۲ برسد و به اجرایش ادامه دهد و B_2 را تمام کند، حالا ترد ۲ در این شرایط نمی تواند داده مربوط به B_2 را از بالای پشته بردارد چون B_2 اصلا بالای پشته نیست و ما در ساختمان داده پشته فقط می توانیم در هر لحظه به اطلاعات بالای پشته دسترسی پیدا کنیم



پس برای اینکه هر ترد بتواند توابع خودش را بصورت بازگشتی فراخوانی کند (یعنی بتواند به توابعی که فراخوانی می‌کند بصورت LIFO دسترسی پیدا کند) باید برای هر ترد یک پشته جداگانه در نظر بگیریم

- در شکل سمت چپ که داره یک فرآیند تک نخی رانشون می دهد، دارید مشاهده می‌کنید که فضای حافظه این فرآیند به موارد متعددی تقسیم داده شده که قبلا توضیح دادم.

Files : منظور همان رسورس ‌هایی است که تحت اختیار این فرآیند است، همان طور که قبلا گفتم در os های unix base به هم رسورس ‌ها به شکل فایل نگاه می شود تا بتواند یک مفهوم یکنواخت را تولید کنند تا کارکردن با اجزا سیستم راحت تر بشود



شکل سمت راست :

این شکل مربوط به یک فرآیند multithreaded است، ترد یک موجودیتی است درون یک فرآیند، یک فرآیند در ساده ترین (ابتدایی ترین) حالت اش یک ترد دارد ولی در حالت پیشرفته تر می تواند تعداد بیشتری ترد داشته باشد

همان طور که در شکل مشخص است بین تمام ترد‌های یک فرآیند که، دیتا و رسورس ‌های فرآیند یکسان و مشترک است ولی هر ترد pc و stack و register set مربوط به خودش را دارد

فرآیند هر رسورسی از os بگیرد آن رسورس ‌ها برای همه ترد‌ها است و همه ترد‌ها به تمام رسورس ‌های این فرآیند دسترسی دارند و بنابراین برنامه نویسان باید حواس شون جمع باشه که دسترسی ‌های موازی ترد‌ها به رسورس ‌ها باعث ایجاد اختلال در سیستم نشود

نکته) فرآیند را می‌توان به خانه تشبیه کرد و اعضای آن خانه را به ترد‌ها تشبیه کرد، مشخصا رسورس ‌هایی مانند آب و برق و گاز را به یک خانه می دهند و این رسورس ‌ها در اختیار کلید افراد درون آن خانه قرار می‌گیرد و درون خانه هر شخص یکسری وسیله شخصی دارد و همین طور یکسری موارد درون خانه بین همه مشترک است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

سوال: یک فرآیند از خودش چه چیزهایی دارد؟ یک ترد چه چیزهایی از خودش ندارد؟

یک فرآیند برای خودش یک pid داشت
یک فرآیند برای خودش یک pc داشت
" " " " " registerset "
" " " " یک PCB "
" " " " حافظه "
" " " " رسورس "
" " " " پشته "

یک ترد از خودش کد ندارد و دارد از کد فرآیند استفاده می‌کند ← کدش کد فرآیند است
یک ترد از خودش حافظه و داده ندارد و از حافظه فرآیند استفاده می‌کند
" " " " رسورس " " " رسورس‌های فرآیند استفاده می‌کند

نکته) کد، حافظه و رسورس‌های فرآیند بصورت ذاتی در اختیار ترد‌های آن فرآیند قرار می‌گیرد و هم ترد از این ۳ مورد بصورت اشتراکی استفاده می‌کنند

نکته) بنابراین چون ترد از خودشون حافظه ندارند اگر مثلاً ترد ۱۰ یک فرآیند از ده حافظه بگیرد (با malloc) مالک این حافظه فرآیند است و بنابراین این حافظه برای تمامی ترد‌های دیگر این فرآیند قابل استفاده است.

۲ یا چند ترد چگونه می‌توانند با هم ارتباط برقرار کنند

۲ حالت در اینجا وجود دارد:

۱- این ۲ ترد متعلق به ۲ فرآیند مجزا باشند: این ارتباط را می‌توانیم ارتباط بین ۲ فرآیند در نظر بگیریم بنابراین این دو می‌توانند از طریق همان روش‌های ارتباط بین فرآیند‌ها که قبلا توضیح دادم ارتباط برقرار کنند

۲- " " " " ۱ " باشند: گفتیم که حافظه یک فرآیند ذاتا بین ترد‌های آن فرآیند shave می‌شود، بنابراین ترد‌ها ذاتا shared memory دارند و این بدان معناست که هر کدام از ترد‌ها هر جای حافظه فرآیند مربوطه‌شان بنویسند آن داده دست ترد دیگر رسیده است و ارتباط (communication) انجام شده است.

سوال) در حالت چند نخی آیا به ازای هر تابع یک ترد داریم یا چند تا تابع می‌توانند در یک ترد باشند؟

درسته که یک تابع کوچکترین جزئی است که می‌تواند به یک ترد تبدیل شود اما لزومی ندارد که یک ترد فقط یک تابع باشد توجه کنید که حافظه فرآیند بین همه ترد‌های یک فرآیند shave است بنابراین همه ترد‌های یک فرآیند به همه توابع درون main دسترسی دارند فرض کنید در کد main ما ۳۰ تا تابع f1 تا f30 را برای کار‌های مختلف‌مان نوشتیم، حالا main می‌تواند ۲۰ تا از این توابع را در ترد اصلی خودش فراخوانی کند و هریک از ۱۰ تا تابع دیگر را به یک ترد مجزا تبدیل کند، حالا هر کدام از این ۱۰ تا ترد می‌توانند در دل خودشان چند تابع درون main را فراخوانی کنند

چند مثال از برنامه‌هایی که هم می‌توانند single thread باشند و هم multithread

۱- web server: یک سرور وب داریم که دارد روی پورت ۸۰ (۸۰ برای HTTP و ۴۴۳ برای HTTPs) گوش می‌کند و بعد کلاینت‌ها می‌توانند به آن متصل شوند و صفحاتی که می‌خواهند مشاهده کنند را از وب سرور درخواست کنند، در هر لحظه ممکن است صدها یا هزاران نفر به یک وب سرور متصل شوند و هر کدام درخواست دریافت صفحات متفاوتی را داشته باشند اما جنس درخواست هم این کلاینت‌ها یکی است و هم اینها درخواست دریافت محتوای صفحه مربوط به خودشان را دارند (سرور محتوای هر صفحه را برای کلاینت مربوطه ارسال می‌کند).

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

حالا سرور دارد چه کاری برای همه این کلاینت ها انجام می دهد؟

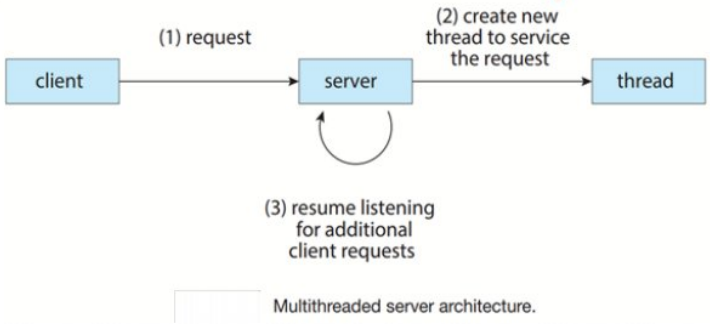
سرور دارد برای همه این کلاینت ها یک کد یکسان را اجرا می کند ، به ازای هر کلاینت یک کدی باید اجرا بشه که کار این کد این است که برود و برای آن کلاینت محتوایی که می خواهد را از روی hd بخواند و برای آن کلاینت روی شبکه ارسال کند و همان طور که مشخص است کدی که برای هر کلاینت اجرا می شود کاملا یکسان است و فقط برای یک کلاینت فایل های مربوط به خودش را ارسال می کند و برای یک کلاینت دیگر فایل های مربوط به خودش را ارسال می کند.

سرور به ازای هر کلاینت باید پاسخ گو باشد حالا ۲ روش برای این کار وجود دارد

۱- سرور به ازای هر کلاینت یک فرآیند جدید به وجود آورد ، یعنی سرور به ازای هر کلاینتی که بهش درخواست می دهد یک fork انجام بدهد ، حالا ایجاد یک فرآیند به ازای هر کلاینت بدان معناست که ما باید به ازای هر کلاینت که گفته شده را در حافظه کپی کنیم و همین طور از os حافظه و رسورس های مورد نیاز را بگیریم

۲- راه حل چیه ؟ thread

⭐ این در حالی است که تمامی کلاینت ها می خواهند یک کد را اجرا کنند ، بنابراین لزومی نداره که سرور به ازای همه کلاینت ها این کد را در حافظه کپی کند (تکرار کند) و فقط کافی است سرور یکبار این کد را بیاره توی RAM اش و بعد به ازای هر کلاینت این کد را اجرا بکند ، در اینصورت هم در حافظه RAM صرفه جویی می شود و هم مزایای دیگری که ترد به همراه خود می آورد را خواهیم داشت و با منابع کمتری همان کار قبلی را انجام می دهیم



Multithreaded server architecture.

بنابراین هر وقت هر کلاینتی به یک سرور درخواست می دهد سرور برای پاسخ دهی به آن کلاینت به جای آنکه یک فرآیند جدید ایجاد کند یک ترد جدید که به این ترتیب ما یک فرآیند multithread (چندنخی) داریم که بین نخ ها switch می کنیم

سرور در این حالت چگونه عمل می کند؟

یک فرآیند داریم روی سرور که یک کد main دارد که کارش این هست که به هر کلاینتی که درخواست میده سرویس بده ، حالا درون این main یک تابعی نوشتیم که کار انتقال داده از سرور به هر کلاینت را انجام می دهد ، حالا main داره اجرا میشه که از یک کلاینت درخواست میاد ، حالا main چون می خواهد منتظر درخواست سایر کلاینت ها باشد و می خواهد به اجرایش ادامه بدهد خودش نمیره و درخواست اون کلاینت رو جواب بده و میاد تابعی که کار انتقال را انجام میدهد را بگوید که تبدیل به یک ترد بشود صدا می زند در اینصورت main داره همروند یا سوازی با این تابع اجرا می شود ، در این صورت main می تواند به اجرایش ادامه دهد و منتظر درخواست سایر کلاینت ها بماند . توجه کنید که در اینصورت هم این ترد ها که هرکدام برای یک کلاینت ایجاد شده به کد و دیتا و رسورس های متعلق به فرآیند بصورت اشتراکی دسترسی دارند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

گفتیم که این کار در server های وب یا مواردی شابه مزایای دیگری نیز به همراه دارد که می خواهم به برخی از آنها اشاره کنم

۱ـ scalability: فرض کنید فیلد PID که OS تعریف شده ۱۰ بیت باشد (در PCB، PID هر فرآیند را نگه می داشتیم) بنابراین ۱۰۲۴ تا فرآیند می توانیم در این سرور داشته باشیم حالا اگر این سرور به ازای هر درخواست یک فرآیند جدید بسازد دیگر نمی تواند به بیشتر از ۱۰۲۴ کلاینت بصورت همزمان سرویس بدهد، تازه به همین تعداد هم نمی تواند چون خود سرور هم یکسری فرآیند برای خودش دارد، در صورتیکه اگر از مدل چندتردی استفاده کند، هر فرآیند یک شماره شناسایی در سطح OS دارد ولی داخل هر شماره شناسی تعدادی زیر شماره شناسایی در سطح فرآیند دارد که OS به آنها توجه نمی کند

۲ـ استفاده بهینه منابع سرور: طبق نکته بالا وقتی تردهای بیشتری در سرور داریم، سرور بهتر از منابع اش استفاده می کند

از سمت کلاینت هم میشه به این داستان نگاه کرد

از درس شبکه می دانیم وقتی کاربر در مرورگر درخواست یک صفحه را می دهد (مثلاً در گوگل روی یک صفحه کلیک می کند، یا در یک سایت روی یک صفحه جدید کلیک می کند) ابتدا مرورگر یک اتصال Tcp به پورت ۸۰ سرور HTTP ای مد نظر می دهد و پس از آنکه سرور پیامی مبنی بر قبول این درخواست برای کلاینت فرستاد، کلاینت از سرور فایل HTML مربوط به صفحه مد نظرش را می خواهد، حال سرور HTML را برای کلاینت می فرستد و در این HTML مشخص است که آن صفحه شامل چه objectهایی است، فرض کنید این صفحه شامل ۲۰ شی شامل چند عکس، فایل صوتی، متن و تصویری باشد، حال مرورگر کلاینت باید برای یکایک اشیا درون این صفحه پیام درخواست (HTTP) جداگانه ای به سمت سرور ارسال کند، حال این درخواست ها می تواند بصورت سری یا موازی به سمت سرور ارسال شود، روش درخواست سری به اینصورت است که تا جواب یک درخواست (HTTP) دریافت نشود مرورگر درخواست (HTTP) دیگری به سمت سرور ارسال نمی کند و موازی به اینصورت است که مرورگر (کلاینت) درخواست این اشیا را بصورت موازی به سمت سرور ارسال می کند، بطور پیش فرض مرورگرها برای دریافت اشیا درخواست موازی می دهند به همین دلیل هم هست که وقتی یک صفحه را باز می کنید اینطوری نیست که تصاویر و متن ها و... یکی یکی و پشت سر هم نمایش داده بشوند و مثلاً یک مقدار از فایل صوتی و یک مقدار از هر تصویر و... آمده

حالا به این مباحث از دید OS نگاه کنیم:

در واقع درخواست سری توسط یک فرآیند single thread و درخواست های موازی توسط یک فرآیند multithread انجام می شود، بنابراین در حالت درخواست موازی توسط مرورگر، لود شدن صفحه دارد توسط تعدادی ترد انجام می شود، اگر مرورگر گرفتن همه اشیا را با یک ترد انجام میداد، این ترد درخواست شی ۱ را برای سرور ارسال می کرد و بعد فرآیند مربوط به لود کردن این صفحه بلاک می شود تا شی ۱ برسد و وقتی شی ۱ رسید از حالت بلاک خارج می شد و بعد که رفت و مجددا روی cpu اجرا شد درخواست شی ۲ را میداد و به همین ترتیب تا شی آخر، درصورتیکه اگر فرآیند مربوط به لود کردن این صفحه را بصورت مالتی ترد می نوشتیم، این فرآیند برای هر شی یک ترد ایجاد می کرد (البته به درجه موازی سازی ربط دارد)، حالا ترد اول روی آمدن شی ۱ بلاک می شود و بلافاصله ترد ۲ روی cpu اجرا میشه و ترد ۲ روی آمدن شی ۲ بلاک می شود و بلافاصله ترد.... در این شرایط response time مرورگر به شدت کاهش پیدا می کند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

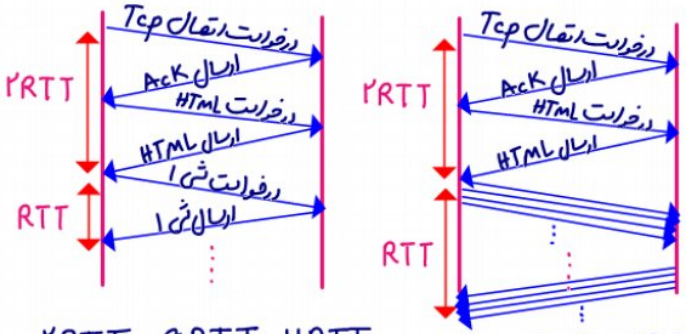
Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

⭐ درخواست هر یک از این اشیا به رسیدن اشیا قبلی وابستگی ندارد، بنابراین این درخواست هایی که بهم وابستگی ندارند را می توانیم موازی به سمت سرور ارسال کنیم، حال برای اینکه بتوانیم موازی بفرستیم نیاز داریم که موجودیت های موازی ایجاد کنیم که به این موجودیت های موازی درون یک فرآیند ترد می گیم.

مثال) فرض کنید یک صفحه شامل ۱۰ شی داریم که یکی از این اشیا فایل HTML است و زمان رفت و برگشت بین کلاینت و سرور RTT باشد در ۲ حالت زیر زمان دریافت این صفحه وب را محاسبه کنید، فرض کنید آدرس IP مرتبط با URL سایت روی کامپیوتر کلاینت کش شده و از زمان انتقال شی صرف نظر کنید:

الف) در حالت single thread

ب) در حالت multithread که درجه موازی سازی ۱۰ باشد



۲RTT + ۹RTT = ۱۱RTT

۲RTT + RTT = ۳RTT

مشخص است که در حالت ب بهطلی کلاینت دارد کاهش پیدا می کند

حالا سوال میتونه میزان زمانی که در سرور داره پراسس انجام میشه را هم بدهد

- در مدت زمانی که درخواست ترد ۱ دارد به سمت سرور منتقل می شود، ترد ۲ می تواند اجرا بشود و بلافاصله درخواست اش را به سرور بفرستد و... پس حدودا بعد از $\frac{RTT}{2}$ همه درخواست ها تقریبا همزمان به سرور می رسند.

⭐ الان یک مثالی زدم برای یکی از مزیت های مالتی ترد به اسم Responsiveness

۲- یک سرور RPC : یک سروری بود که یکسری خدمات ارائه میده که کلاینت ها می توانند با استفاده از فراخوانی به آنها دسترسی پیدا کنند، این سرور روی یکی از پورت هایش listening می کرد و منتظر تا کلاینت ها درخواست های call کردن procedure های روی خودش را بدهند نکته) پس وقتی درخواست هایی که برای یک سرور وجود دارد مشابه است باید از ایجاد فرآیندهای جدید که می توانند بسیار پرهزینه باشد باید خود داری کنیم.

مزایای برنامه نویسی مالتی ترد (multithreaded programming)

۱- Responsiveness (پاسخگویی) : اگر یک فرآیند تک رشته (single thread) سیستم کالی اجرا کند که باعث بلوکه شدنش شود (یا به هر دلیلی بلاک شود) دیگر این فرآیند پاسخ گو نیست (response ندارد) تا event ای که منتظرش است رخ دهد، بنابراین اگر event های دیگری برای این فرآیند رخ دهد به آنها نمی تواند واکنش نشان دهد، اما اگر یک فرآیند مالتی ترد داشته باشیم که تردها دارند بصورت موازی یا همروند با هم اجرا می شوند اگر یکی از تردهای آن همان سیستم کال را فراخوانی کند به جای آنکه کل فرآیند بلاک شود فقط همان یک ترد بلاک شود و سایر تردها به اجرایشان ادامه می دهند و این تردهایی که بلاک نشدند می توانند به وقایع واکنش نشان بدهند و فرآیند ما در این شرایط پاسخ گو است و response دارد.

۲- Resource sharing : Data, code (memory) و resources بین تردها مشترک است بنابراین نیازی نیست به ازای هر ترد یک فضای حافظه جدید برای کد و دیتا در نظر بگیریم و همین طور رسورس های مختلف را به تردها اختصاص بدهیم و همان کد و دیتا و رسورس هایی که برای یک فرآیند اختصاص داده می شود برای تمام تردها قابل استفاده است و این باعث میشه که حافظه کمتری اشغال

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

بشه و رسورس های کنترل در اختیار یک فرآیند (مجموع تردهای یک فرآیند) قرار بگیرد.

۳- Economy :

۱- سرعت switch بین تردهای از سرعت switch بین فرآیندها به مراتب بالاتر است:

وقتی می خواهیم بین دو تا فرآیند سوئیچ کنیم باید هم pcb عوض می شد (pcb قدیم را سیو و pcb جدید را بارگذاری می کردیم) و هم از حافظه فرآیند جاری سوئیچ کنیم به حافظه فرآیندی که قرار است اجرا بشود که این کارها سربار (overhead) زیادی دارند، اما اگر بخواهیم بین دو تا ترد سوئیچ کنیم

اول: نیازی نیست بین حافظه ها سوئیچ کنیم (نیازی به عوض کردن فضای حافظه نداریم)

دوما: نیازی به ذخیره و بازیابی کل pcb نداریم.

نکته) بعنوان مثال سایز pcb می تواند در حد 4KB باشد در صورتیکه اطلاعاتی که از یک ترد باید نگه داریم مثلاً در حد ۳۰ بایت است.

یادآوری: برای یک ترد باید شماره شناسایی آن ترد، pc اش، اشاره گر به پشته ای که دارد (sp) و محتویات رجیسترهایش را نگهداری کنیم

مثال) در سیستم عامل solaris سرعت context switch بین تردها ۵ برابر سرعت cs بین فرآیندها است.

۲- سرعت ایجاد ترد از سرعت ایجاد فرآیندها به مراتب بالاتر است.

برای ایجاد فرآیند مثلاً با fork، سیستم عامل باید به این فرآیند هم pcb بدهد و هم بخش حافظه بدهد و Address space اش را بسازد که این کار زمان برات (گرفتن این منابع از os زمان برات) پس سرعت ایجاد فرآیند کم است. تازه اگر os با ایجاد فرآیند موافقت کند!

مثال) در سیستم عامل solaris سرعت ایجاد یک ترد ۳۰ برابر سرعت ایجاد یک فرآیند است.

۴- افزایش بهره وری در multiprocessor: فرض کنید ما ۴ تا core داریم روی cpu مان و هر ۴ تا core خالی باشد حال اگر فرآیند ما single thread باشد (heavyweight process) این فرآیند در هر لحظه روی یک core می تواند اجرا بشود و از core های دیگر نمی تواند هیچ استفاده ای بکند، در صورتیکه اگر این فرآیند ۴ ترد می داشت این امکان برایش وجود داشت که ۴ تا تردش را بصورت موازی روی هر ۴ تا core اجرا بکند بنابراین در این حالت می توانیم روی یک عدد فرآیند speed up بگیریم ولی یک فرآیند heavyweight نمی تواند هیچ speed up ای روی سیستم های مالتی پروسسور بگیرد، پس با مالتی ترد می توانیم یک فرآیند را سریع کنیم، توجه کنید در حالتی که ترد نداریم می توانیم فرآیندهای مختلف را روی core های مختلف اجرا کنیم ولی اینجا داریم در خصوص افزایش سرعت یک فرآیند صحبت می کنیم و در سطح یک فرآیند هم داریم پردازش موازی ایجاد می کنیم.

قبلا در خصوص speed up مفصلا صحبت کردم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها ندارد.استفاده غیرمجاز از این فیلم کاملا حرام و پیگرد قانونی دارد

Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

بشه و رسورس های کمتری در اختیار یک فرآیند (مجموع تردهای یک فرآیند) قرار بگیرد.

۳- Economy:

۱- سرعت switch بین تردهای از سرعت switch بین فرآیندها به مراتب بالاتر است:

وقتی می‌خواهیم بین دو تا فرآیند سوئیچ کنیم باید هم pcb عوض می‌شد (pcb قدیم را سیو و pcb جدید را بارگذاری می‌کردیم) و هم از حافظه فرآیند جاری سوئیچ کنیم به حافظه فرآیندی که قرار است اجرا بشود که این کارها سربار (overhead) زیادی دارند، اما اگر بخواهیم بین دو تا ترد سوئیچ کنیم

اول: نیازی نیست بین حافظه‌ها سوئیچ کنیم (نیازی به عوض کردن فضای حافظه نداریم)

دوما: نیازی به ذخیره و بازیابی کل pcb نداریم.

نکته) همینطور مثال سایز pcb می‌تواند در حد 4KB باشد در صورتیکه اطلاعاتی که از یک ترد باید نگه داریم مثلاً در حد ۳۰ بایت است.

یادآوری: برای یک ترد باید شماره شناسایی آن ترد، pc اش، اشاره‌گر به پشته‌ای که دارد (sp) و محتویات رجیسترهایش را نگهداری کنیم

مثال) در سیستم عامل solaris سرعت context switch بین تردها ۵ برابر سرعت cs بین فرآیندها است.

۲- سرعت ایجاد تردها از سرعت ایجاد فرآیندها به مراتب بالاتر است.

برای ایجاد فرآیند مثلاً با fork، سیستم عامل باید به این فرآیند هم pcb بدهد و هم بخش حافظه بدهد و Address space اش را بسازد که این کار زمان‌بر است (گرفتن این منابع از os زمان‌بر است) پس سرعت ایجاد فرآیند کم است. تازه اگر os با ایجاد فرآیند موافقت کند!

مثال) در سیستم عامل solaris سرعت ایجاد یک ترد ۳۰ برابر سرعت ایجاد یک فرآیند است.

۴- افزایش بهره‌وری در Multiprocessor: فرض کنید ما ۴ تا core داریم روی cpu مان و هر ۴ تا core خالی باشد حال اگر فرآیند ما single thread باشد (heavyweight process) این فرآیند در هر لحظه روی یک core می‌تواند اجرا بشود و از core‌های دیگر نمی‌تواند هیچ استفاده‌ای بکند، در صورتیکه اگر این فرآیند ۴ ترد می‌داشت این امکان برایش وجود داشت که ۴ تا تردش را بصورت موازی روی هر ۴ تا core اجرا بکند بنابراین در این حالت می‌توانیم روی یک عدد فرآیند speed up بگیریم ولی یک فرآیند heavyweight نمی‌تواند هیچ speed up ای روی سیستم‌های مالتی پروسسور بگیرد، پس با مالتی ترد می‌توانیم یک فرآیند را سریع کنیم، توجه کنید در حالتی که ترد نداریم می‌توانیم فرآیندهای مختلف را روی core‌های مختلف اجرا کنیم ولی اینجا داریم در خصوص افزایش سرعت یک فرآیند صحبت می‌کنیم و در سطح یک فرآیند هم داریم پردازش موازی ایجاد می‌کنیم.

قبلا در خصوص speed up مفصلا صحبت کردم.

challenges in programming for multicore systems

برنامه نویسی مالتی ترد ساده نیست و اگر بخواهیم از core‌هایمان خوب استفاده کنیم چالش‌های زیادی دارد

۱- Identifying concurrent task

تشخیص دادن اینکه چه تست‌هایی از برنامه را می‌توان به کارهای موازی تقسیم کرد کار ساده‌ای نیست، نباید بین تردهایی که می‌خواهیم ایجاد کنیم وابستگی وجود داشته باشد تا یک ترد معطل انجام شدن کار تردهای دیگر نشود

۲- Balance

باید سعی کنیم تردهایی که به وجود می‌آوریم هم وزن باشند که این هم کار ساده‌ای نیست

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

چون اگر هم وزن نباشند اینها دیر و زود تمام می‌شوند و به اینصورت به آن speedup ای که گفتیم نمی‌رسیم.

۳- Data spliting

داده‌هایی که در اختیار تردها قرار می‌دهیم باید بین آنها تقسیم بشوند (split بشوند) تا بتوانند روی هسته‌های جداگانه اجرا بشوند. فرض کنید ۱۰ تا ترد داشته باشید که همه اینها دارند روی یک داده‌ای کار می‌کنند و بنابراین همه اینها سر حافظه با هم رقابت دارند و معطل همدیگر می‌شوند

۴- Data Dependency

گاهی اوقات کارهایی که یک ترد انجام میدهد به کار یک ترد دیگر مربوط است (به داده‌های دیگری وابسته است) در اینصورت برنامه نویس باید حواسش به هماهنگ کردن و synchronization بین این دو ترد باشد، در غیر اینصورت ممکن است مشکلات جدی‌ای در حافظه و درصحت و درستی اجرا برنامه به وجود آید.

بین تردهای یک فرآیند که ذاتا با هم حافظه share شده دارند باید همگام سازی (synchronization) وجود داشته باشد، یعنی تردهای ۱ و ۲ باید به این مورد توجه کنند که اگر یکی از تردها در حال نوشتن در یک قسمت مشخص از این حافظه است تا کار آن ترد با آن قسمت خاص حافظه تمام نشده دیگری از آن نخواند، توجه کنید که داریم از Time sharing استفاده می‌کنیم، به مثال زیر توجه کنید:
فرض کنید در یک قسمت از این فضای share شده یک آرایه به طول ۲۰ وجود دارد و T_1 باید همه درایه‌های این آرایه را در عدد c ضرب کند و در انتها T_2 از این آرایه برای کار خودش استفاده کند، حالا فرض کنید که T_1 تا عنصر ۱۰ این کار را انجام می‌دهد و سپس کوانتوم اش تمام می‌شود و نوبت T_2 می‌شود، حالا نباید T_2 بیاد همه این آرایه را بخواند و باید کاری کنیم که T_2 از این موضوع مطلع باشد و نتواند از این آرایه که عمل نوشتن در آن تکمیل نشده عمل خواندن را انجام دهد و کاری کنیم که T_2 تا زمانیکه T_1 نوشتن در این آرایه را تکمیل می‌کند صبر کند، یعنی با اینکه زمان می‌رسد به T_2 ولی T_2 نمی‌تواند از این حافظه بخواند و مجاز به این کار نیست که این کار همان بحث synchronization هست.

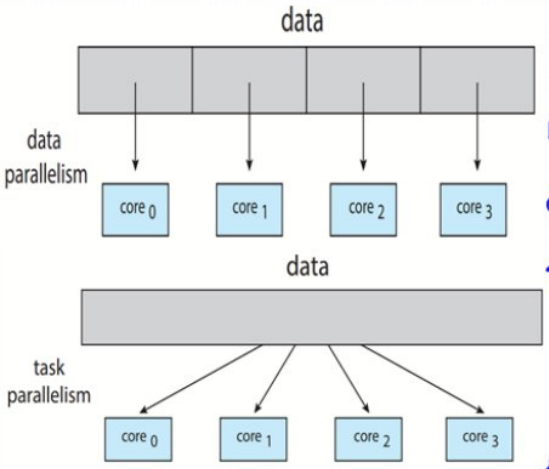
۵- Testing and debugging

چون برنامه ما داره بصورت موازی روی core های مختلف اجرا میشه چندین سید اجرایی مختلف وجود دارد و دنبال کردن منطق این برنامه توسط برنامه نویس و تشخیص اینکه دقیقا چه ترکیبی از اجرای موازی دستورالعمل ها باعث به وجود آمدن باگ شده کار دشواری است، پس تست و دیباگ برنامه‌های همروند (concurrent program) ذاتا کار دشواری است.

Type of parallelism

معمولا برنامه‌ها هم از Data parallelism و هم از task parallelism استفاده می‌کنند، حالا این دو تا چیه؟

۱- Data parallelism



یک تسک یکسان را روی داده‌های متفاوتی روی core های مختلف اجرا کنیم. بعنوان مثال می‌خواهیم تمامی عناصر یک آرایه ۱۰۰ تایی را با هم جمع کنیم حالا توی این روش می‌توانیم مثلا این ۱۰۰ تا عنصر آرایه را به ۴ تا عنصر ۲۵ تایی بشکونیم و روی هر کدام از این ۲۵ عنصر عمل تکراری و یکسان جمع کردن را روی یک core جدا انجام بدهیم و در نهایت حاصل اینها را با هم جمع کنیم.

۲- Task parallelism

روی یکسری داده‌های یکسان و مشترک عملیات متفاوتی را روی core های مختلف انجام دهیم، مثلا فرض کنید یک برنامه ای داریم که ۲ تا آرایه ورودی می‌گیرد (یا دارد) و نیاز دارد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

هم جمع نظیر به نظیر، هم ضرب نظیر به نظیر این ۲ آرایه و هم ضرب آرایه A در عدد ثابت c و هم مجموع عناصر آرایه B را محاسبه کند

thread library

یک thread library (کتابخانه نخ) یک API برای ایجاد و مدیریت نخ در اختیار برنامه نویس قرار می دهد. نرم افزار کارهای پیچیده ای می تواند انجام دهد که برخی از آنها را در جای لازم بیان می کنم، اما یکسری از کارهایی که انجام میدهد شامل کارهای زیر است: (کدهایی برای) ایجاد نخ ها، ازبین بردن نخ ها، انتقال پیام ها و داده ها بین نخ ها، زمانبندی نخ ها، ذخیره کردن و load کردن context نخ ها در هنگام switch بین نخ ها و... را برعهده دارد.

۲ راه اصلی برای پیاده سازی thread library وجود دارد

۱- رویکرد اول این است که این کتابخانه کاملا در فضای کاربر و بدون پشتیبانی کرنل ارائه شود، یعنی تمام کدها و توابع لازم برای مدیریت نخ ها در فضای کاربر وجود دارد و این بدان معناست که فراخوانی یک تابع از این thread library منجر به یک local function call در فضای کاربر (user space) می شود و نه منجر به system call.

۲- رویکرد دوم پیاده سازی این library در سطح هسته (Kernel_level) است که مستقیما توسط OS پشتیبانی شود یعنی در این حالت تمام کدها و توابع و Data structure های مورد نیاز در Kernel space قرار دارد و در نتیجه فراخوانی توابع این کتابخانه نخ معمولا منجر به یک system call به کرنل می شود

نکته) ما از توابع thread library برای ارتباط بین user thread ها با لایه کرنل و یا برای مدیریت نخ ها در لایه بالا و یا پایین استفاده می کنیم

نکته) thread library یک مشت کده و میتوند بخشی از اون توی سطح کاربر و بخشی ازش توی کرنل پیاده سازی بشه

نکته مهم

اگر برنامه نویس برای ایجاد و کار و مدیریت نخ از thread library استفاده کند که در فضای ارائه و پیاده سازی شده چون فراخوانی توابع درون thread library بصورت function call در فضای کاربر است و کارهایی که خود ترد لایبری هم برای مدیریت نخ ها انجام میدهد با اجرا کدهایش در فضای کاربر اتفاق می افتد و منجر به system call نمی شود بنابراین ایجاد، حذف، سوئیچ کردن، تغییر حالت های نخ ها، سوئیچ کردن بین نخ ها و... سریع انجام می شود ولی اگر thread library در فضای کرنل پیاده سازی شده باشد چون بسیاری از عملیات با system call انجام می شود کارهای گفته شده کند هستند کلا وقتی سوئیچ مد پردازنده انجام میشه باعث کندی میشه ← ۲ تا mode switch لازم دارید، از مد کاربر به مد هسته و بعد از مد هسته به مد کاربر

همان طور که قبلا گفتم ۳ thread library اصلی امروز برای ایجاد و کار با نخ ها وجود دارد

۱) posix pthreads: pthread ها نسخه های توسعه یافته ای از استاندارد posix است که کتابخانه های سطح کاربر و سطح کرنل را ارائه می کند.

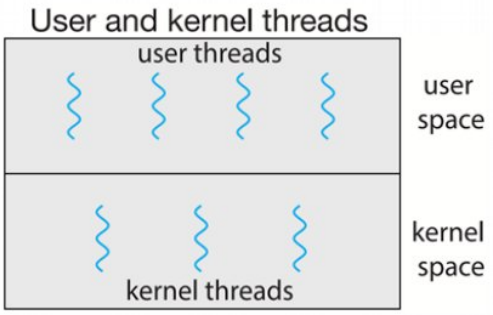
۲) windows: thread library ویندوز یک کتابخانه در سطح هسته است که در سیستم های ویندوز وجود دارد.

۳) java: java thread API اجازه میدهد که نخ ها مستقیما در برنامه های جاوا ایجاد و مدیریت شوند، اما چون اغلب jvm روی os میزبان (host) اجرا می شود، java thread API بصورت generaly با استفاده از thread library موجود در سیستم میزبان پیاده سازی می شود، این بدان معناست که در سیستم های ویندوز java thread ها معمولا با استفاده از windows API پیاده سازی می شوند و در سیستم های unix، Linux و macOS توسط pthread ها پیاده سازی می شوند

- قبلا به شما برنامه ای نشان دادم که در آن ما بعنوان برنامه نویس از توابع thread library استفاده کردیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



انواع ترد

ما ۲ نوع ترد داریم:

user thread : موجودیت های موازی که در user space وجود دارد. سیستم عامل خیلی وقت ها از وجود user thread ها بی اطلاع است و thread library که واسط بین سیستم عامل و نخ های سطح کاربر است از این تردها مطلع است.

user thread را بالای کرنل و در سطح کاربر مدیریت می کنیم، یعنی بدون پشتیبانی کرنل مدیریت می شوند

" Kernel : موجودیت های موازی که در Kernel space وجود دارد.

Kernel thread ها مستقیما توسط سیستم عامل پشتیبانی و مدیریت می شوند. کرنل تردها را os می بیند و scheduler ـ os اینها را می بیند و می تواند تصمیم بگیرد که هر کدام از اینها را چه زمانی اجرا بکند

تردهای کرنل اول تردهایی هستند که os آنها را می شناسد و بهشون خدمات نیز می دهد، خدماتی از جمله cpu time، فضا، اگر درخواست Device داشته باشند بهشون Device میدهد، برایشان فایل open می کند

ما به ازای هر فرآیند باید حداقل یک عدد کرنل ترد داشته باشیم که آن کرنل ترد نماینده آن فرآیند در سطح os است، یعنی مثلاً وقتی فرآیند P_1 (یکی از نخ هایش) یک system call از نوع Blocking را فراخوانی می کند، این کرنل ترد که نماینده این فرآیند است، هستش که از صف running خارج میشه و میره توی صف دستگاه I/O مربوطه قرار می گیرد (صف هارد یا کارت شبکه یا..)

مدل های ارتباطی بین یوزر تردها و کرنل تردها Multithreading Models

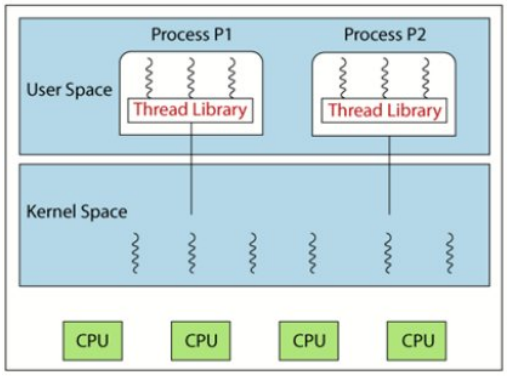
مالتی تردینگ بصورت های گوناگونی در os های مختلف پیاده سازی شده که توضیح تمامی آنها از بحث ما خارج است ولی چند مدل مهم و رایج آن را بررسی خواهیم کرد

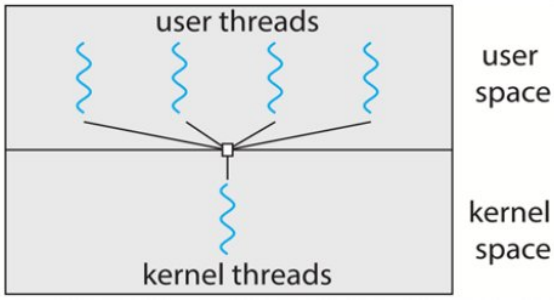
۱- Many-to-one
۲- one-to-one
۳- Many-to-many

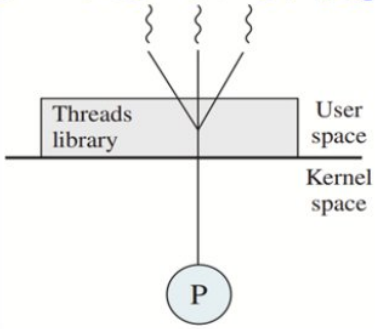
Many-to-one تعدادی یوزر ترد به یکدونه کرنل ترد وصل هستند

پیاده سازی پیش فرض این روش با کتابخانه های نخ سطح کاربری انجام می شود.

در شکل روبه رو process P_1 را مشاهده می کنید که دارد از یک thread library استفاده می کند و سه تا user thread ساخته، همه این یوزر تردها به یک کرنل ترد متصل هستند.

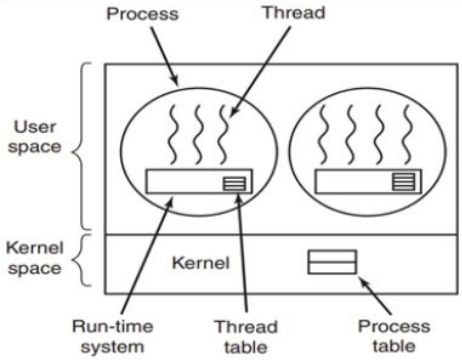






سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



A user-level threads package

- وقتی تردی در فضای کاربر مدیریت می‌شوند هر فرآیند به private thread table خود نیاز دارد تا بتواند تردهای موجود در این فرآیند را پیگ کند، این جدول شابه process table هسته است با این تفاوت که شخصات هر ترد مانند pc هر ترد، sp هر ترد، رجیستر و state هر ترد را دارد. این جدول توسط run time system مدیریت می‌شود. توابعی که وضعیت رشته‌ها را ذخیره می‌کند و توابع scheduler همگی توابع محلی و فراخوانی شان بسیار کارا تر از system call است.

مزایای پیاده سازی کامل thread library (thread package) در user space

۱- چون کرنل چیزی در مورد تردهای یک فرآیند نمی‌داند و فرآیندهای سطح کاربر را بصورت فرآیندهای معمولی و تک رشته‌ای مدیریت می‌کند بنابراین این روش را می‌توان روی یک os ای که از ترد پشتیبانی نمی‌کند پیاده سازی کرد (نخ‌های سطح کاربر می‌توانند روی هر os ای اجرا شوند و هیچ تغییری در کرنل برای پشتیبانی از نخ‌های سطح کاربر لازم نیست)

۲- thread switching سریعتر است و نیازی به سیستم کال رفتن در تله os ندارد.

برای تعویض نخ به رفتن به مد ستاز هسته نداریم، چون تمام ساختمان داده‌های مورد نیاز برای مدیریت نخ در داخل فضای آدرس کاربر یک فرآیند قرار دارد، بنابراین در تعویض مد صرفه جویی می‌شود (فقط کافی است محتوای ثبات‌های cpu برای نخ جاری ذخیره و محتوای مورد نیاز برای اجرا ترد بعدی در ثبات‌های cpu قرار داده شود.

چون همه کارهای تردها در سطح یوزر انجام می‌شود و بدون سوئیچ کردن به سطح کرنل بین تردهای درون یک فرآیند سوئیچ می‌کنیم بنابراین سرعت cs بین تردها بالاست.

توجه) عمل سوئیچینگ بین ۲ ترد متعلق به ۲ فرآیند از نوع تعویض متن فرآیندی (process context switch) است.

۳- تمامی توابع مربوط به تردها در حالت کاربری اجرا می‌شود و در اجرای توابع تردها نیازی به فراخوانی سیستمی نیست و بنابراین سرعت اجرا بالا می‌رود

۴- نیازی به flush کردن TLB نیست.

۵- زمانبندی بین تردها و توزیع زمان اجرا بین آنها توسط کتابخانه‌های سطح کاربری انجام می‌شود.

۶- یک برنامه می‌تواند زمانبندی خاص خودش را داشته باشد، مثلاً شاید برای یک برنامه زمانبندی Round Robin مناسب‌تر باشد و برای یک برنامه دیگر الگوریتم زمانبندی priority-based. همین طور برخی از برنامه‌ها دوست ندارند برخی از نخ‌های آنها در لحظه نامناسب قطع بشه. پس در این روش می‌توان متناسب با برنامه یک الگوریتم زمانبندی در نظر گرفت بدون اینکه در زمانبندی os بخواهد ایجاد مزاحمت کند.

۷- نخ‌های سطح کاربر scale پذیرترند، چون نخ‌های هسته همیشه به مقداری فضای پشته و جدول در کرنل نیاز دارند که اگر تعداد رشته‌ها زیاد باشد می‌تواند مشکل ساز باشد

معایب:

۱- اگر یک یوزر ترد یک سیستم کال از نوع Blocking اجرا کند (مثلاً نوشتن پیام روی فایل)، کرنل ترد به خاطر این فراخوانی سیستمی از حالت running برود در صف فایل و در این حالت دیگر کسی نیست که برای فرآیند p در صف ready قرار بگیرد تا بتواند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

زمان cpu را در اختیار بگیرد، بنابراین در این شرایط کل فرآیند Block می‌شود.
پس در این مدل یک Blocking system call که توسط یکی از نخ‌های یک فرآیند صدا زده می‌شود باعث می‌شود کل فرآیند بلاک شود چون تنها نماینده این فرآیند در سطح os بلاک می‌شود (می‌رود در صف یک Device قرار می‌گیرد)

۲- اگر چندتا core داشته باشیم هر core یک صف آماده (ready queue) دارد بنابراین اگر یک فرآیند می‌خواست از K تا core استفاده کند باید K تا نماینده در سطح os میداشت که هرکدام برود در یکی از این K صف آماده قرار بگیرد ولی چون در این مدل فقط یک نماینده به ازای هر فرآیند وجود دارد بنابراین در این مدل نمی‌توانیم از مزیت multicore یا tiprocessor استفاده کنیم.

چطور بیشتر روی این مشکلات کار کرد؟

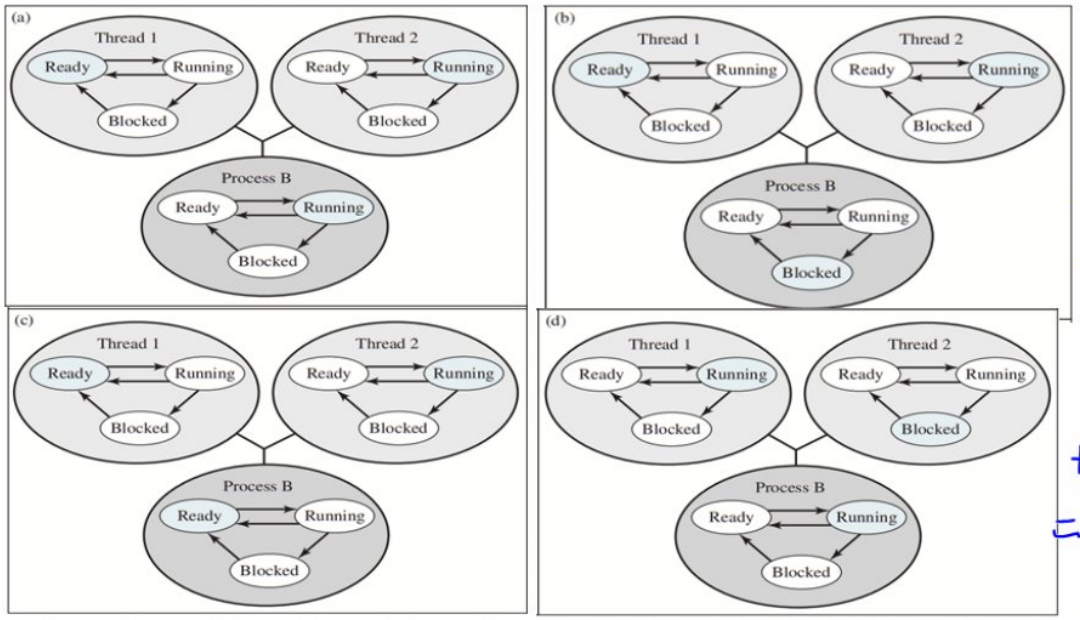
۱- به جای اینکه یک برنامه را بصورت مالتی ترد بنویسیم، آن را بصورت چند فرآیندی بنویسیم اش، خب اینجوری دیگه مزیت‌های مالتی ترد را نداریم و به جای هر تعریف ترد یک تعریف فرآیند داریم که سربار خیلی بیشتری دارد

۲- jacketing: یک راه برای غلبه بر Blocking thread است، هدف jacketing تبدیل یک Blocking system call به یک nonblocking system call است. بعنوان مثال یک ترد به جای اینکه مستقیما یک I/O system call بزند یک Application-level I/O jacket routine را فراخوانی کند، حالا توی روتین jacket کدهایی وجود دارد که بررسی می‌کند که آیا دستگاه I/O مشغول است یا نه و در صورتی که مشغول باشد این ترد دارد Block - state می‌شود و کنترل از طریق thread library به ترد دیگری در آن فرآیند داده می‌شود، حالا زمانی که ترد لایبرری دوباره کنترل را به ترد بلاک شده دارد، روتین jacket مجددا دستگاه I/O را چک می‌کند.

رابطه زمانبندی نخ و فرآیند

کرنل فرآیند را می‌شناسد و آن را schedule می‌کند و به فرآیند یک حال اجرای واحد اختصاص می‌دهد (ready، running، Block و...)

فرض کنید یک فرآیند داریم شامل ۲ ترد که در ابتدا در حالت (a) است.



شکل b: فرض کنید برنامه داده ترد ۲ را اجرا میکند و یک فراخوان سیستمی Blocking اجرا می‌کند، در این حالت کرنل فرآیند B را بلاک می‌کند و به یک فرآیند دیگر switch می‌کند.

حالا به حالت ترد ۲ توجه کنید، با توجه به Data structure ای که توسط thread library است ترد ۲ فرآیند B هنوز در حالت running است توجه کنید که داشتن این حالت توسط ترد ۲ به معنای اجرا شدن آن روی پردازنده نیست، اما توسط thread library بعنوان حالت اجرا تلقی می‌شود. بعدا وقتی کرنل مجددا زمان را به فرآیند B می‌دهد اجرا در ترد ۲ از سر گرفته می‌شود.

شکل c: time slice فرآیند B تمام می‌شود و بنابراین وقفه ساعت باعث می‌شود کنترل به کرنل منتقل شود، حالا کرنل state -

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

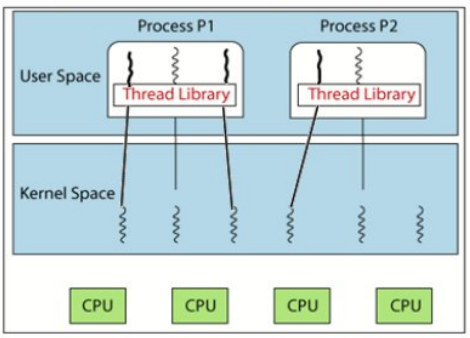
فرآیند B را ready می‌کند و فرآیند دیگری را dispach می‌کند روی cpu، اگر به شکل c توجه کنید ترد ۲ فرآیند B هنوز در حالت running است.
بعد از وقتی کرنل مجدداً زمان را به فرآیند B می‌دهد اجرا در ترد ۲ از سر گرفته می‌شود.

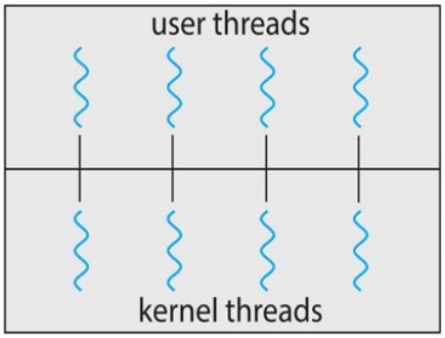
شکل d: ترد ۲ به نقطه‌ای رسیده که نیاز به انجام یکسری کار توسط ترد ۱ فرآیند B دارد، در این شرایط state ـ ترد ۲ بلاک می‌شود و state ـ ترد ۱ از Ready به Running تغییر می‌کند و توجه کنید که خود فرآیند B در حالت Running باقی می‌ماند

نکته) فرض کنید فرآیند B وسط اجرای کدهای thread library اش قطع می‌شود و زمان توسط کرنل به فرآیند دیگری داده می‌شود، حالا فرض کنید thread library وسط کار سوئیچ کردن بین دو تا ترد بوده، در این حالت چه اتفاقی می‌افتد؟
وقتی زمان دوباره به این فرآیند رسید، اجرای کدهای thread library اش از جای قبلی از سر گرفته میشه و عمل سوئیچ بین ۲ ترد را کامل می‌کند و کنترل را به ترد مربوطه در آن فرآیند منتقل می‌کند.

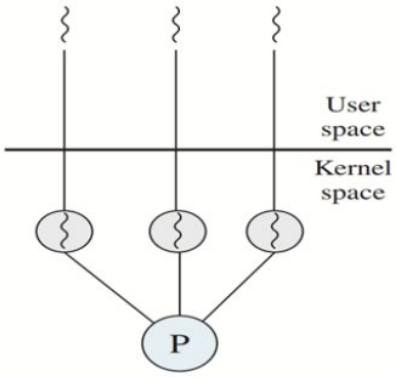
one-to-one Model

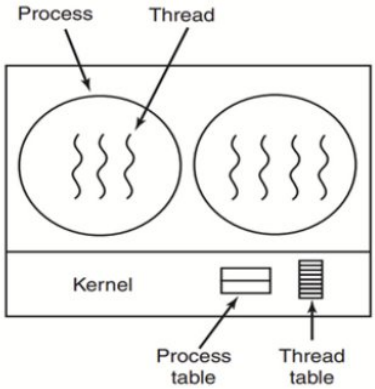
به ازای هر user thread یک Kernel thread وجود دارد و هر زمان فرآیند (برنامه) یک user thread جدید ایجاد کند به ازایش یک Kernel thread جدید به وجود بیاید. وارتباط و مدیریت بین تردهای سطح کاربر با تردهای سطح هسته توسط thread library انجام می‌شود





user space
kernel space





A threads package managed by the kernel

- شکل سمت راست حالتی را نشان می‌دهد که thread library در کرنل وجود دارد، در این حالت در هر فرآیند هیچ thread table ای وجود ندارد و در عوض کرنل یک thread table دارد که همه تردهای درون سیستم را ترک می‌کند. پس به ازای هر ترد یک TCB در کرنل ایجاد می‌شود.
- کرنل context information کل فرآیند را نیز نگهداری می‌کند.

در این روش تردها توسط کرنل پشتیبانی می‌شوند و os از تمامی تردهای حالت هسته اطلاع دارد و تک به تک توسط os زمانبندی می‌شوند، هر ترد مربوط به یک فرآیند بصورت مستقل از سایر تردهای آن فرآیند بخشی از کدهای فرآیند مربوطه اش را اجرا می‌کند و هر یک مستقل از دیگر تردهای آن فرآیند می‌توانند به حالت بلاک بروند

اگرچه هزینه عملیاتی این روش (تردهای هسته) از فرآیندها کمتر است اما برای زمانبندی و اجرا توابع مربوط به تردها به system call نیاز است. وقتی یک ترد Block شود خود os است که تعیین می‌کند که time slice را به چه تردی بدهد و تردی که الگوریتم زمانبندی os انتخاب می‌کند می‌تواند متعلق به فرآیند جاری باشد یا متعلق به فرآیند دیگری باشد.

در این روش os خودش همه تردها را می‌سازد و عمل mapping بین تردهای سطح کرنل و سطح کاربر را انجام میدهد.

نکته) linux و windows از این مدل استفاده می‌کنند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) ترد‌های جاوا از مدل one-to-one استفاده می‌کنند، البته پروژه Loom ترد‌های یوزرلول را به جاوا اضافه می‌کند که تحت عنوان virtual threads شناخته می‌شوند.

مزایا:

۱- اگر یک یوزر ترد متعلق به یک فرآیند یک Blocking system call فراخوانی کند نماینده مربوط به آن بلوکه می‌شود و بنابراین سایر یوزر ترد‌های دیگر آن فرآیند و نمایندگان مربوط به آنها می‌توانند به فعالیت خودشان ادامه بدهند و بنابراین کل فرآیند بلوکه نمی‌شود. (پس اگر برای یکی از ترد‌های فرآیند A نقص صفحه رخ بدهد باز کل فرآیند A بلوکه نمی‌شود چون هسته سایر ترد‌های آن فرآیند را می‌شناسد)

۲- می‌توانیم از مزایای multicore و multiprocessor استفاده کنیم. چون یک فرآیند مثلاً K تا یوزر ترد دارد که به K تا کرنل ترد متصل هستند و هرکدام از این کرنل ترد‌ها می‌توانند در ready queue کور‌های مختلف قرار بگیرند. توجه کنید درجه توازی در این روش به تعداد ترد‌های سطح کرنل که البته با تعداد ترد‌های سطح کاربر برابر است خواهد بود.

۳- خود روال‌های هسته نیز می‌تواند مالتی ترد باشد

معایب:

۱- در این مدل هر زمان فرآیند (برنامه) بخواهد یک یوزر ترد به وجود بیاورد به ازایش یک کرنل به وجود می‌آید، ایجاد کرنل ترد نیاز به سوئیچ کردن به مد کرنل دارد، همچنین نیاز به تخصیص منابع توسط os و نیاز به تخصیص id به آن کرنل ترد دارد که این موارد باعث کند شدن ایجاد ترد در این مدل می‌شود.

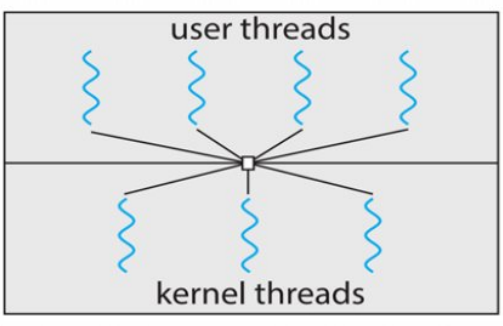
۲- این مدل تعداد ترد‌هایی که سیستم می‌تواند پشتیبانی کند را محدود می‌کند

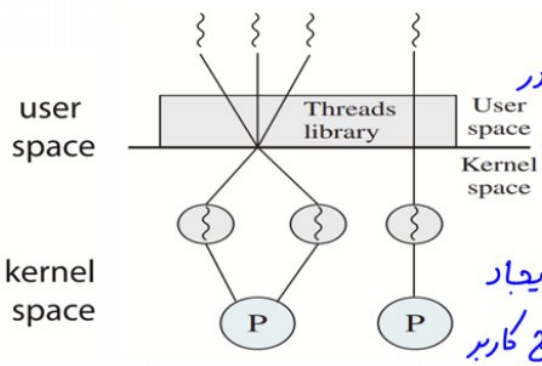
فرض کنید شما می‌گفتید که برای ID کرنل ترد‌ها در نظر گرفتیم ۱۰ بیتی باشد، بنابراین ما نهایتا ۱۰۲۴ تا کرنل ترد می‌توانیم ایجاد کنیم و چون هر کرنل ترد را به یک یوزر ترد می‌توانیم مپ کنیم ماکزیمم یوزر ترد‌هایی که می‌توانیم ایجاد کنیم نیز ۱۰۲۴ تا خواهد بود. در صورتی که در مدل many to one در این حالت می‌توانیم ۱۰۲۴ تا فرآیند داشته باشیم که هرکدام می‌توانند به تعداد بیت‌هایی که برای ترد‌های در سطح فرآیند تعریف شده ترد ایجاد کنند.

توجه) البته طراح سیستم هم خیلی دوست ندارد تعداد کرنل ترد‌ها در این روش محدودیتی نداشته باشد چون همان طور که گفتم ما به ازای هر ترد باید یکسری منابع اختصاص بدهیم که اگر محدودیتی در ایجاد ترد وجود نداشته باشد می‌تواند باعث بروز مشکل شود بنابراین طراح os تعداد بیتی که برای id کرنل ترد‌ها در نظر می‌گیرد را بگونه‌ای در نظر می‌گیرد که مشکلی برای سیستم رخ ندهد.

۳- انتقال کنترل از یک ترد به یک ترد دیگر در داخل حتی یک فرآیند مشخص نیازمند این است که به مد هسته وارد شویم.

many to many





این روش ترکیب ۲ روش بالاست.
در این روش یک مقداری از ترد کابیری در os پیاده‌سازی میشه و یک مقداری اش در سطح کاربری.
در این روش ترد کابیری سطح کاربر، ایجاد، حذف، زمانبندی و مدیریت ترد‌های سطح کاربر

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

را برعهده دارد و بنابراین این کارها توسط توابع کتابخانه سطح کاربر انجام می‌شود و سبک است و یکی ترد سطح os داریم بنام Lwp (light weight processess) که ایجاد، حذف، زمانبندی و مدیریت اینها را کرنل‌های کتابخانه سطح os انجام میدهد، توجه کنید که Lwp ها هستند که واقعا روی core اجرا می‌شوند.

در این روش برنامه نویس به هر تعدادی دوست دارد ترد یوزر لول می‌سازد اما ترد لایبری به تعدادی مشخص و کمتر مساوی یوزر ترد، بهش کرنل ترد یا همان Lwp می‌دهد، هر Lwp در هر لحظه می‌تواند یک نخ یوزر ترد مربوط به خودش را اجرا کند، حالا اگر کار ترد 1 مربوط به فرآیند A تمام شد، ترد لایبری سطح کاربر می‌تواند یک ترد دیگر را برای اجرا به همان Lwp بدهد، توجه کنید که اگر یوزر تردی که روی یک Lwp در حال اجرا است یک Blocking system call اجرا کند درست است که Lwp مربوطه (نماینده مربوطه) نیز Block می‌شود اما اگر Lwp های دیگری در اختیار این فرآیند باشد کل فرآیند Block نمی‌شود و فقط در صورتیکه یک فرآیند Block می‌شود که تمام Lwp های متعلق به آن Block شوند و مشخص است که درجه توازی یک فرآیند به تعداد Lwp هایی که به آن فرآیند داده شده نه به تعداد یوزر تردهایی که فرآیند ایجاد کرده و افزایش یوزر تردهای توازی واقعی را لزوما افزایش نمی‌دهد نکته) یک ترد یوزر لول که روی Lwpx، Block می‌شود بعد از اینکه بیدار شد می‌تواند روی Lwpy که متعلق به همان فرآیند است به اجرایش ادامه بدهد.

نکته) در این مدل ترد لایبری سطح کاربر است که باید تصمیم بگیرد که کی کدوم یکی از یوزر تردها به کدوم یکی از کرنل تردها متصل بشوند بنابراین عملیات پیچیده‌ای از جنس interface و از جنس مدیریت یوزر تردها در ترد لایبری سطح کاربر اتفاق می‌افتد.

scheduler Activation

وقتی یک یوزر ترد که داره روی یک Lwp اجرا میشه یک Blocking system call فراخوانی کند اجرا از مد کاربر به مد هسته منتقل می‌شود و ISR مربوطه روی همین Lwp اجرا می‌شود و درنهایت این نماینده (Lwp) بلاک می‌شود و می‌رود و در صف یک I/O می‌نشیند و os یک Lwp دیگر را روی این core آزاد شده می‌اندازد، حالا فرض کنید os به این ترد 10ms اختصاص داده بوده ولی این ترد پس از 2ms یک فراخوانی Blocking اجرا می‌کند، حالا چون برنامه ما نمیتواند از بقیه سهم زمانی اش استفاده کند، os شاید بخواهد یک Lwp دیگر به این برنامه اختصاص بدهد، اما در این لحظه os نمی‌داند که کدامیک از یوزر تردهای شما را به این Lwp اختصاص دهد و تصمیم گیری برای اینکه چه یوزر تردی برای اجرا انتخاب شود با ترد لایبری سطح کاربر است، بنابراین os زمانی که یک ترد به حالت Block می‌رود به کتابخانه سطح کاربر یک upcall میزند (اطلاع میدهد) (فراخوانی رو به بالا) که در اثر این upcall، Lwp جدید تخصیص داده شده به این فرآیند یک upcall Handler از ترد لایبری سطح کاربر اجرا می‌شود و نخ بعدی که باید روی این Lwp اجرا شود مشخص می‌کند و از اینجا به بعد این یوزر ترد انتخاب شده روی این Lwp جدید اجرا می‌شود.

در یک حالت دیگر هم upcall صورت می‌گیرد و آن موقعی است که یک Lwp (نماینده) کار I/O اش تمام شده و با یک وقفه بیدار می‌شود، حالا چون ترد لایبری سطح کاربر بخواهد فعلا یک یوزر ترد دیگری را به این Lwp بدهد باید upcall انجام بشود

scheduler Activations: می‌توان گفت یک حالت میانی بین user thread و Kernel thread است (از windows7 پیاده سازی شده در ویندوز) و در واقع برای رفع مشکل بلاک شدن تردهای حالت کاربر است و روش کار به این صورت است که ما از ترد لایبری سطح کاربری برای انجام کارهای پیمان استفاده می‌کنیم ولی به os خبر می‌دهیم که ما داریم از تردهای سطح کاربری

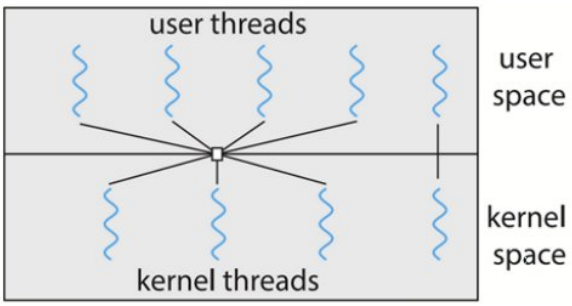
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

استفاده می‌کنیم، یادمان هست که در حالت ساده به اینصورت بود که ما از تردهای سطح کاربری استفاده می‌کردیم و خودمان هم این را می‌دانستیم و os هیچ چیز در این مورد نمی‌دانست، توجه داشته باشید که در این روش هم os از تعداد تردهای پراسس ما اطلاعی ندارد و فقط میداند که ما داریم از تردهای سطح کاربری استفاده می‌کنیم، حالا وقتی یکی از تردهای پراسس ما Block شود، os چون میداند ما داشتیم از تردها در حالت کاربری استفاده می‌کردیم میاد و با یک upcall به تردلایبری سطح کاربری اطلاع میدهد که آن تردی از توکه داشت اجرا می‌شد بلاک شده و هنوز از time slice ای که بهت اختصاص دادم باقی مانده و اگر می‌خوای میتونی یک ترد دیگه بری که سند os برات اجرا کنم با این کار یکی از مشکلات حالت many to one که با بلاک شدن یک ترد کل پراسس بلاک می‌شد رفع می‌شود.

نکته) اگرچه به نظر می‌رسه مدل many-to-many، انعطاف پذیرترین مدل است اما در واقعیت پیاده سازی آن دشوار هست همین طور با افزایش تعداد core در اکثر سیستم‌ها محدود کردن تعداد تردهای کرنل اهمیت کمتری پیدا کردند و در نتیجه در حال حاضر اکثر سیستم عامل‌ها از مدل one-to-one استفاده می‌کنند

مدل many-to-many در سیستم عامل solaris که منسوخ شده پیاده سازی شد و تقریبا در هیچ os دیگری وجود ندارد و در solaris به اینصورت بود که ما بعنوان برنامه نویس به os می‌گفتیم که به من x تا کرنل ترد بده و بعد برنامه نویس هر تعدادی یوزر ترد که می‌خواست می‌ساخت و زمانبندی اینکه در هر لحظه کدام یوزر ترد به کدام کرنل ترد نسبت بشود توسط تردلایبری سطح کاربر انجام می‌شود



Two-level model

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

تردسازی ضمنی : Implicit threading

با رشد پیوسته multicore processing برنامه‌هایی حاوی صدها یا حتی هزاران ترد در حال به وجود آمدن هستند
گفتیم بین تردها حافظه، آیند مربوطشان مشترک است و اگر برنامه‌نویس به این موضوع توجه نکند ممکن است این حافظه مشترک بصورت درست توسط این تردها استفاده نشود و عملکرد صحیح برنامه را دچار اشکال کند و یا اجرا برنامه با مشکل مواجه شود و گفتیم به دلایلی از این دست برنامه‌نویسی بصورت مالتی ترد پیچیده و سخت است و نیاز به تجربه بالایی دارد، حال برای تعدیل در این موضوع یکی از راه‌های رسیدگی به این سختی‌ها و پیچیدگی‌ها و پشتیبانی بهتر برای طراحی برنامه‌های همروند و موازی این هست که ایجاد و مدیریت تردها را از توسعه‌دهندگان برنامه‌ها به کامپایلرها و run_time library‌ها منتقل کنیم.
پس به جای اینکه برنامه‌نویس خودش صراحتا ترد ایجاد کند این کار را به یک ابزاری که این کار را بصورت خودکار انجام میدهد محول می‌کند، مثلاً این ابزار می‌تواند یک کامپایلر باشد که بیاد خودش ترد ایجاد بکند.
این کار باعث می‌شود یکی از پیچیدگی‌هایی که برعهده برنامه‌نویس است از بین برود.
- بعنوان مثال زبان برنامه‌نویسی مالتی ترد openMP از تردسازی ضمنی استفاده می‌کند

مثال: میخواهیم درایه‌های آرایه a و b را تطبیر به تطبیر با هم جمع کنیم و در آرایه c بریزیم، خب این کار را می‌شود بصورت موازی برای هر درایه انجام داد، بنابراین کافی است در یک زبان برنامه‌نویسی مثلاً این امکان که این for بطور کامل تابل موازی‌سازی باشند و کامپایلر مستقل به وجود آوردن تردهای مربوط به این کد باشد و برنامه‌نویس کمترین دخالت را برای ایجاد تردها و مدیریت نکاتی که برای کار با تردها باید رعایت کند را داشته باشد.

```
for(i=0;i<100; i++)
    c[i]=a[i]+b[i];
```

- یا مثلاً چون GPU از تعداد زیادی core تشکیل شده برنامه‌نویس‌هایی که از زبان‌های برنامه‌نویسی برای GPU استفاده می‌کنند می‌توانند تردهای زیادی رو به این روش به وجود بیارند و روی این core‌ها اجرا کنند

عملیاتی که روی تردها انجام می‌شود

۱- fork : یک فرآیندی داریم که چندتا ترد ایجاد کرده، حالا یکی از تردهای این فرآیند میاد و سیستم کال fork را فراخوانی می‌کند (اجرا می‌کند)، حال میدانیم که این کار باعث می‌شوند که فرآیند فرزندی به وجود بیاید.
حالا این سوال مطرح می‌شود: آیا فرآیند فرزند جدیدی که به وجود می‌آید یک فرآیند مالتی ترد مانند پدرش باشد و تردهای پدرش برای فرآیند فرزند هم به وجود بیاید یا اینکه فرآیند فرزند یک فرآیند single thread باشد.
نکته) توجه کنید که هزینه به وجود آوردن یک فرآیند فرزندی که چندتی مانند پدر باشد بیشتر است از حالتی که بخواهد تک‌تی باشد حالا بسته به هر کاربرد باید ببینیم که آیا پرداخت این هزینه می‌صرفه و به درد ما می‌خورد یا نه؟

* بسته به اینکه ماهیت فرآیند پدر چیست

۱- فرآیند فرزند بهتر است single thread باشد.

۱-۱) در مثال web server گفتیم که سربرد بهتر است به ازای هر کلاینت به جای ایجاد یک فرآیند فرزند جدید به وجود آورد یک ترد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

جدید ایجاد کند (در خصوص مزایای این کار صحبت کردیم)، حالا اگر یکی از تردهای این فرآیند بخواهد یک فرآیند فرزندی ایجاد کند، بهتر است این فرآیند فرزند single thread باشد، چون فرآیند پدر اگر تا آن لحظه K تا ترد داشته به خاطر این بوده که K تا کلاینت داشته ولی فرآیند فرزندی که ایجاد کردیم که هنوز مشتری ندارد، بنابراین در این شرایط و در این مثال دلیلی وجود ندارد که اگر فرآیند پدر K تا ترد دارد، فرآیند فرزند هم K تا ترد داشته باشد.

۲-۱) فرض کنید گذری که نوشتیم به این صورت است که وقتی والد با استفاده از fork یک فرآیند فرزند جدید به وجود می‌آورد، فرزند بخواهد سیستم کال exec() را فراخوانی کند، یادمان هست که exec میومد و کل حافظه فرآیند فرزند را با برنامه جدید رونویسی می‌کرد، خب حالا اگر والد K ترد داشته باشد که میدانیم همه این K ترد در یک فضای حافظه هستند و فرآیند فرزند بخواهد Duplicate والدش باشد، یعنی K ترد والدش را Duplicate کند، خب به محض اینکه فرزند exec را فراخوانی کند کل حافظه فرزند رونویسی می‌شه و بنابراین همه آن تردها هم رونویسی می‌شوند و از بین می‌روند، بنابراین در این سناریو ما اومدیم هزینه پرداخت کردیم که فرآیند فرزند مالتی ترد باشه ولی این هزینه الکی بوده و بنابراین در این حالت بهتر است فرآیند فرزند تک نخی ایجاد بشود. یعنی برنامه نویس وقتی می‌خواهد در کد فرزند exec را فراخوانی کند، از مدلی از ایجاد فرزند استفاده کند که ترد Duplicate نشوند

۲- فرآیند فرزند بهتر است مانند پدرش مالتی ترد باشد.

فرض کنید فرآیندی داریم که دارد video streaming انجام میدهد، فرض کنید این فرآیند ۳ کار اساسی زیر را انجام می‌دهد:

۱: capture ← ویدیو

۲: encode کردن ویدیو capture شده و بافر کردن encode ویدیو capture در حافظه

۳: انتقال این ویدیو encode شده روی شبکه

توجه کنید که سه عمل متفاوت دارد انجام می‌شود که می‌تواند مستقل و موازی با هم انجام بشود (مثل یک پایپ‌لاین)، در این حالت ما برای افزایش سرعت و بهتر کردن response time دوتا کار می‌توانیم انجام بدهیم:

۱: برای این سه کار سه فرآیند مجزا ایجاد کنیم

۲: " " " " ترد " در یک فرآیند ایجاد کنیم

برای انجام این کار (نوشتن برنامه مربوطه) بهتر است سه تا ترد به وجود بیاریم تا سه تا فرآیند، چون این تردها حافظه مشترک دارند سرعت ارتباط بین این ۳ ترد بسیار بالاتر از سرعت ارتباط بین فرآیندها است، چون اگر سه فرآیند ایجاد کنیم بعد هر فرآیند که می‌خواهد داده‌اش را به فرآیند بعدی بدهد باید از مکانیزم‌های os استفاده کند که این روش کند است.

حالا اگر یکی از تردها سیستم کال fork را فراخوانی کند آیا بهتر است فرآیند مالتی تردی مثل پدرش به وجود بیاید یا یک فرآیند تک نخی؟

اگر فرآیند فرزند تک نخی ایجاد شود، یعنی فقط یکی از ۳ ترد بالا را داشته باشد، این فرآیند فرزند اصلا قادر به انجام این کار نخواهد بود

thread cancellation: از بین رفتن ترد

۱- Asynchronous cancellation

۲- Deferred "

همان طور که وقتی به یک ترد نیاز داشتیم آن را به وجود آوردیم می‌توانیم وقتی به تردی نیاز نداریم آن را از بین ببریم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Asynchronous cancellation

یک ترد به محض اینکه متوجه شد که دیگر نیازی به کارهای ترد مورد نظرش ندارد در همان لحظه فورا آن ترد را از بین ببرد (کنسل کند)

مثال: یک وب سرور رو در نظر بگیرید که برای پاسخ گویی به هر کلاینت یک ترد ایجاد می‌کند، حالا اگر وب سرور به محض اینکه متوجه بشود که کلاینت x درخواست اتمام ارتباط داده یا متوجه بشود ارتباط کلاینت قطع شده است بیاد و ترد مربوط به آن کلاینت را از بین ببرد دارد از مدل آسنکرون استفاده می‌کند.

مشکلات کنسل کردن آسنکرون ترد

کنسل کردن ترد بصورت آسنکرون می‌تواند عوارض بدی داشته باشد و خطرناک باشد با چند مثال چندتا از این عوارض را توضیح خواهم داد

۱) فرض کنید تردی داشتیم که در آن ترد با سیستم کال malloc از os حافظه گرفته‌ایم و در انتهای کد مربوط به این ترد می‌خواستیم این حافظه را آزاد کنیم، حالا اگر قبل از اینکه این ترد بخواهد حافظه گرفته شده را آزاد کند ما این ترد را بصورت آسنکرون از بین ببریم آن گاه این حافظه دیگر آزاد نمی‌شود. توجه کنید که برنامه نویس اول اون تابعی که قراره یک ترد رش بکنه اومده و حافظه گرفته و آخر همین تابع هم میخواسته آن حافظه رو آزاد بکنه ولی قبل از اینکه ترد به این خط برسه، ترد از بین رفته و بنابراین هیچ تابع دیگری در برنامه ما به جز همان تابع مذکور نمی‌تواند آن حافظه را آزاد کند.

۲) فرض کنید در یک تابعی که داریم با یک ترد اجرایش می‌کنیم یک فایل را باز کرده‌ایم (توجه کنید که این فایل برای فرآیند است، یعنی فرآیند مالک این فایل می‌شود نه تردی که این رسورس را گرفته) و بعد می‌خواستیم توی این تابع close اش کنیم ولی اگر قبل از اینکه close اش کنیم ترد از بین برود تا زمانیکه فرآیند مربوط به آن ترد فعال است از نگاه os آن فایل برای آن فرآیند باز است، حالا این فایل مثلاً برای این فرآیند بصورت قفل شده باز می‌ماند، بنابراین هیچ ترد دیگری هم نمی‌تواند به این فایل دسترسی پیدا کند.

۳) فرض کنید تردی که الان می‌خواهیم کنسل کنیم در همین لحظه وسط تغییر دادن اعضای یک آرایه باشد و قرار بوده همه عناصر آرایه را تغییر بدهد حالا وقتی ما وسط کارش اون رو از بین می‌بریم، یک آرایه‌ای باقی می‌ماند که نه برای قبل از تغییر هست و نه برای بعد از تغییر و این خوب نیست. در اصطلاح می‌گیم یک آرایه inconsistent خواهیم داشت.

چه زمانی از این نوع cancellation استفاده کنیم

این موضوع خیلی به کاری که آن ترد انجام میدهد بستگی دارد.

۱- اگر ما بعنوان برنامه نویس می‌دانیم که از بین بردن یک ترد بصورت آسنکرون باعث بروز مشکل نمی‌شود، از بین بردن آن ترد را به تعویق نمی‌اندازیم، مثلاً فرض کنید ترد داره یکسری عملیات محاسباتی انجام میدهد که خود اون ترد میخواد بعدا ازشون استفاده بکند و ما فهمیدیم که دیگه به اون ترد نیاز نداریم پس فورا از بین می‌بریم اش

۲- فرض کنید متوجه می‌شویم که یک ترد داره کاری رو انجام میدهد که اگر ادامه پیدا کند می‌تواند برای سیستم خطرناک باشد، بنابراین بلافاصله از بین اش می‌بریم

Deferred cancellation (کنسل کردن معوق)

درحین اجرا ترد یک نقاط مشخص بنام cancellation point وجود داشته باشد که در این نقاط از بین بردن آن ترد باعث بروز اشکال نمی‌شود (کنسل کردن در آن نقاط ایمن است) حالا وقتی به یک ترد می‌گیم از بین برود، در هر نقطه‌ای که باشه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

کنسل نمیشه و وقتی به اولین cancellation point برسه کنسل می‌شود.
پس نقطه cancellation point نقطه‌ای است که ملاحظات بالا را در نظر گرفته و هر فایلی که باید close میشده را close کردیم، هر حافظه‌ای که باید آزاد میشده را آزاد کردیم، هم داده‌ی مشترکی که یک ترد باید تغییر میداده است را صبر کردیم تا بطور کامل تغییر داده شود و....

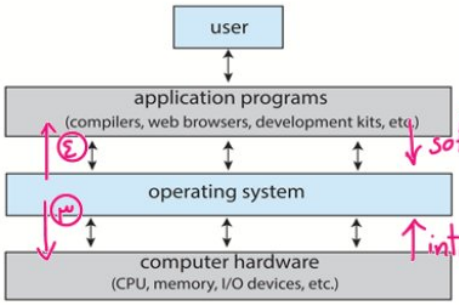
توجه) توجه کنید که برنامه‌نویس درکدش می‌تواند از هر ۲ نوع cancellation استفاده کند.

Signal Handling

interrupt: وقتی از جانب Hardware رخدادی رخ میدهد و در نتیجه آن OS صدا زده می‌شود.

software: " وقتی از سمت App، OS صدا زده می‌شود

و exception و trap



۳) حالا اگر OS با سخت افزار کار داشته باشد خودOS میداند که با سخت افزار کار دارد بنابراین نیازی نیست OS کسی را خبر کند وچون OS مدیر سیستم است می‌تواند هر کاری دارد را توسط سخت افزار انجام بدهد

۴) حالا اگر OS بخواهد با برنامه‌ها (Applications) ارتباط برقرار کند و خبری را به برنامه‌ها بدهد این کار را با استفاده از signal انجام میدهد، پس signal یک اطلاع برای رخداد یک اتفاق از جانب OS به App است

صادر شدن سیگنال به چه معناست؟ صادر شدن سیگنال می‌تواند یک بیت باشد که از ۰ به ۱ تغییر می‌کند، یعنی به محض اینکه یک بیت خاصی از ۰ به ۱ تغییر کرد یعنی آن سیگنال صادر شده است.

چرا ممکن است OS بخواهد یک خبری به یک برنامه بدهد؟

برنامه ممکن است یک کار غیرمجازی انجام بده و OS می‌خواهد به برنامه اطلاع بدهد که آن کاری که انجام داده غیرمجاز (اشتباه) بوده یا برنامه از OS میخواد رخ دادن یک event ای را بهش اطلاع بدهد.

مثال) مثلاً برنامه یک جایی از کدش عمل تقسیم بر ۰ ای را انجام میدهد، حالا خطای تقسیم بر ۰ را سخت افزارهای cpu تشخیص میدهد و در اثر آن یک exception به وقوع می‌پیوندد (یک exception (استثنا) raise می‌شود، نوعی از وقفه به اسم exception رخ می‌دهد) و در اثر آن OS باید آن exception را هندل کند (OS میره و با اجرا ISR متناظر با این استثنا exception handeling را انجام میدهد) و OS برای آنکه به برنامه مذکور بگوید (اطلاع دهد) که تو باعث رخ دادن این exception بودی یک signal برای آن برنامه صادر می‌کند، حالا آن سیگنال باید (فقط) یکبار هندل شود.

type of signal

۱- synchronous signal: سیگنال بعد از اجرا یک instruction مشخص توسط برنامه می‌آید (صادر می‌شود)، یعنی دقیقاً محل صادر شدن اش مشخص است. الان لحظه صادر شدن سیگنال در مثال بالا مشخص و محل رخ دادنش

۲- Asynchronous signal: سیگنالی که OS برای برنامه صادر می‌کند مشخص نیست موقع اجرا چه instruction ای توسط برنامه آمده است.

مثال برای مورد ۲: برنامه ممکن است با فراخوانی یک سیستم کال به OS بگوید که ۵ms بعد از اجرا این سیستم کال به من اطلاع بده

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

(برنامه یک تایمر برای خودش ست می‌کند) و سپس برنامه به اجرا دستورات خودش ادامه میدهد (یک سیستم کال از نوع non blocking) حالا هر وقت تایمری که os برای این برنامه ست کرده تمام شد (5ms تمام شد) یک وقفه صادر می‌شود و os میاد و ISR مربوط به وقفه تایمر را اجرا می‌کند و متوجه می‌شود که دلیل این وقفه آن برنامه ای بوده که به os گفته بوده 5ms بعد خبر بده، حالا os باید از طریق صادر کردن یک سیگنال به برنامه مذکور اطلاع بدهد، خب وقتی این سیگنال امده مشخص نیست که برنامه مذکور در حال اجرا چه دستوری است.

Signal Handling

همان گونه که وقتی وقفه ای (توسط سخت افزار یا نرم افزار) صادر می‌شد باید آن وقفه توسط os فقط یکبار هندل شود، توجه کنید که os یک وقفه را دوبار هندل نمی‌کند و برای هندل کردن آن os یک interrupt handeling انجام میداد (یک ISR اجرا می‌کرد)، وقتی هم سیگنالی صادر می‌شود باید فقط یکبار signal Handeling انجام بشود (فقط یکبار هندل بشود)
چه سیگنال سنکرون باشد و چه آسنکرون باید هندل شود، حال مسئله این است که چه کسی این سیگنال را هندل کند.
Default signal Handler: بصورت پیش فرض یک سیگنال توسط Default Handler در کرنل هندل می‌شود و اگر برنامه نویس بخواهد خودش سیگنال را هندل بکند (مثلاً برنامه نویس می‌خواهد در صورت اتفاق این رخداد آن را ببیند و جد آن را نادیده بگیرید) می‌تواند یک user_defined signal Handler تنظیم کند که در اینصورت این برنامه به جای Default signal Handler اجرا می‌شود و بنابراین خود یوزر این سیگنال را هندل می‌کند.

signal را اگر بخواهد خود برنامه هندل کند چه کسی هندل می‌کند

به مدل هایی که برای یوزر تردها و کرنل تردها وجود دارد بستگی دارد، دانش شما از ارتباطی که بین یوزر تردها و کرنل تردها وجود دارد می‌تواند به شما این دانش را بدهد که signal را چه کسی می‌تواند هندل کند و برنامه نویس چقدر دقیق می‌تواند یک سیگنال را به مقصد خودش برساند، چون اگر یک سیگنال از os به کل یک فرآیند داده بشود سخت است که برنامه نویس تشخیص بدهد مقصد سیگنال چه کسی بوده.

- اگر مدل ارتباطی یوزر ترد و کرنل ترد one-to-one باشد، بنابراین os می‌تواند بصورت دقیق بفهمد که عامل سیگنال کدام یوزر ترد بوده و پس از صادر شدن سیگنال همان یوزر ترد این سیگنال را هندل بکند
- اگر مدل many-to-one باشد، اگر os سیگنالی صادر کند مشخص نیست که به کدامیک از این تردها صادر شده، بنابراین این سیگنال می‌تواند به هر یوزر ترد اعمال بشود ولی فقط یکی از تردها این را می‌تواند هندل کند یا thread library این کار را بر عهده می‌گیرد یا حتی می‌توانید برخی از تردها را مسئول این موضوع بکنید
  ۱- ممکن است همان تردی که سیگنال بهش داده شده هندل بکند
  ۲- " " سیگنال به همه تردهای یک فرآیند داده بشود ولی فقط یک ترد آن را هندل بکند
  ۳- " " فقط به یکی ترد خاص ارسال بشود

Thread pools: مخزن تردها

گفتیم معمولاً یک وب سرور برای پاسخ به درخواست هر یک کلاینت های خودش یک ترد ایجاد می‌کند، حال می دانیم که تعداد درخواست هایی که به یک وب سرور بزرگ می‌آید می‌تواند خیلی زیاد باشد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

گفتیم ایجاد ترد از ایجاد فرآیند خیلی سبک تر است، چون برای ساخت فرآیند باید یک محدوده حافظه درست کنیم و کد و دیتا را از حافظه اصلی توی RAM بذاریم و stack و Heap را باید برایش بسازیم (new کنیم) و Allocate کنیم و ...) ولی برای ساخت ترد چون قبلا توسط فرآیند همه موارد لود شده در فضای آدرس فقط کافی است یک پشته بهش بدهیم و یک TcB هم تولید می کنیم و تمام. حالا درسته که ایجاد ترد زمان خیلی کمتری نسبت به ایجاد فرآیند می گیرد اما اگر قرار باشد همین ساخت ترد را در حجم بالا انجام بدهیم، مشکلات زیر را خواهیم داشت:

۱- زمان قابل ملاحظه ای برای ساخت تعداد خیلی زیادی ترد مصرف می شود

۲- حتی اگر هر ترد میزان کمی از فضای حافظه را اشغال کند وقتی تعداد زیادی ترد ایجاد شود حافظه قابل ملاحظه ای اشغال خواهد شد

۳- وقتی تعداد تردهای داخل صف ready زیاد بشه response time سایت افزایش می یابد (سایت کند به فعالیت کاربران پاسخ می دهد). وقتی تعداد ترد زیادی توی صف ready داشته باشیم وقتی یکی یک کوانتوم می گیرد، طول میکشه تا بخوایم کوانتوم بعدی رو بهش بدهیم و زمان زیادی باید توی صف وایستند

حالا برای رفع این مشکلات میایم از مفهوم استخر ترد (ریسمان) استفاده می کنیم، یعنی ما بعنوان مهندس میایم بررسی می کنیم که این وب سرور ما اگر چند تا ترد داشته باشه optimize هست و فاکتورهایی که ما می خوایم را برآورده می کند، فرض کنید به عدد x می رسیم، حالا وقتی برنامه وب ما می خواد روی سرور شروع به اجرا کند همان ابتدا x تا ترد آماده به کار ایجاد می کند، یعنی x تا ایجاد می شود و آماده به کار هستند. حالا درخواست هایی که از کلاینت ها به این سرور می رسند توی یک صفی گذاشته می شوند و هر ترد که اجرایش تمام شد (پاسخ به یک درخواست را انجام داد) میان و یک درخواست را برمی دارند و اجرایش می کنند.

چطور مشکل ۱ حل شد؟ قبل از اینکه وب سرور شروع به کار کردن کند x تا ترد را برایش ساختیم و بهش دادیم

〃 〃 ۲ 〃 〃 ؟ درسته که در این روش هم حافظه را اشغال می کند ولی چون ما از قبل خودمان بررسی کردیم و حالت بهینه را در نظر گرفتیم میزان مشخصی از حافظه اشغال میشه که ما مشخص کردیم ولی در حالت قبل به ازای هر درخواست یک ترد ایجاد می کنیم و وقتی محدودیتی روی ایجاد ترد نداشته باشیم حافظه پر می شود و احتمالش هست سیستم به حالت انفجاری برسد

چگونه مشکل ۳ 〃 〃 ؟ ما حساب کتاب کردیم که اگر x تا درخواست هم باهم بیاد زمان پاسخ هر کلاینت OK باشه.

وقتی یک فرآیند ایجاد می شود به جای آنکه در ابتدا کار فرآیند ما single thread باشد یک فرآیند داشته باشیم با یک thread pool ثابت، مثلاً یک مخزن K تایی از ترد، حالا فرآیند که به وجود می آید فقط ۱ ترد فعال دارد و K-1 ترد غیر فعال داریم، حالا اگر برنامه نویس نیاز به ایجاد یک ترد جدید داشت نیازی نیست ابتدا به آن ترد ID و استک و ... اختصاص بدهیم و K-1 ترد در مخزن آماده است و فقط کافی برنامه نویس آن را فعال کند و بعد اشاره گر به یک تابع را بهش بدهد، پس از اینکه کار این ترد تمام شد، این ترد مجددا به مخزن برمی گردد، یعنی برنامه نویس آن را از بین نمی برد (ID و استک و ... را پس نمی گیریم). داشتن thread pool باعث می شود ایجاد کردن و از کار انداختن ترد خیلی سریعتر انجام بشود ولی تعداد تردهای یک فرآیند محدود می شود به K تایی که برایش مشخص کردیم.

دانستنی) از جاوا ۶ به بعد در جاوا Thread pool داریم.

در جاوا چند مدل Threadpool داریم:

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۱- تعداد تردهای Fix ای ابتدا برنامه مان می سازیم. NewFixedThreadPool(count);

۲- می توانیم Thread pool کش مانند ایجاد کنیم: یک تعداد minimum تردی را همیشه زنده نگه می دارد، مثلاً ۵۰ تا، ولی بیش می گیریم اگر تعداد درخواست ها بالا رفت تا ۱۰۰ تا هم تعداد تردها را زیاد کن، حالا اگر یک مدت زمان شخصی با آن ۵۰ تا ترد دوم کار نداشتیم این ها Kill می شوند و مجددا تعداد تردها به ۵۰ تای اولیه می رسد.

نکته) توجه داشته باشید که در انتهای کار برنامه مان باید threadpool را با دستور shutdown، shutdownNow() کنیم وگرنه تردها زنده می مانند و memory leakage خواهیم داشت.

تعداد تردهای درون استخر ریسمان می توانند براساس یک تابع هیوریستیک باشد با فاکتورهای

۱- اینکه چقدر حافظه اصلی ما ظرفیت دارد

۲- چند تا core یا پردازنده وجود دارد

۳- برنامه چقدر cpu-bound یا I/O Bound است.

۴- اگر برنامه وب سروری است کلاینت ها چقدر درخواست همزمان میدن (concurent client request)

dispatching/context switch سیستم عامل

این شبه کد یکی از مهمترین بخش های os است، این که اگر سیستم crash نکند (در لینوکس می گیم Kernel panic) و یا شما سیستم را خاموش نکنید همیشه اجرا می شود و این لوپ، لوپ اصلی انجام کار os است

```
while(true){
  new_tcb = select_next_tcb();→ تردهایی که باید اجرا شود را از صف ready انتخاب کن و آن را در new_tcb بذار.
  save_cpu_state(cur_tcb);→ context = تردی که میخوایم cpu رو ازش بگیریم را save کن
  load_cpu_state(new_tcb);→ تردی که انتخاب کردی را در cpu لود کن که اجرایش را شروع کنیم
  cur_tcb = new_tcb;
  run_thread();→ ترد مربوط را اجرا می کند
  thread_house_keeping();وقتی اجرا ترد تمام شد (thread_exit فراخوانی شد) یکسری پاکسازی ها انجام میدهد، مثلاً استک اش را release میکند
}
```

* توجه کنید که تابع run_thread که داره یک ترد رو اجرا میکند خیلی وقت ها قبل از اینکه ترد کل کارش تموم بشه باره و باره تمام می شود!

توابع تردها

thread_fork(function name, args): چون تردها داخل یک فرآیند هستند و کد و داده یکسان با فرآیندی که در و نشان هستند دارد برای ایجاد ترد کافی است فقط اشاره به یک تابع و آرگومانهای آن تابع را بدهیم که در اینصورت بصورت موازی با ترد اصلی شروع به اجرا آن تابع می کند. با این دستور ترد ابتدا به وضعیت new و سپس ready می رود.

توجه) برای سادگی یک تابع می دهیم ولی می شود یک خط برنامه را نیز داد.

```
int main(void){
  while(1);
  printf("Finished!");→ هیچ وقت این خط اجرا نمی شود و Finished چاپ نمی شود
}
```

```
int main(void){
  thread_fork(while(1));
  thread_fork(printf("Finished!"));→ برای هر خط یک ترد ایجاد می شود که همزمان یا موازی با هم می توانند اجرا شوند
}
```

thred_join(thread_handle): تردی که این دستور را اجرا کند ابتدا به وضعیت waiting می رود و وقتی thread_handle کارش تمام شد، وضعیت اش ready می شود. شبیه wait در فرآیندها است ولی در اینجا رابطه پدر و فرزندی نداریم.

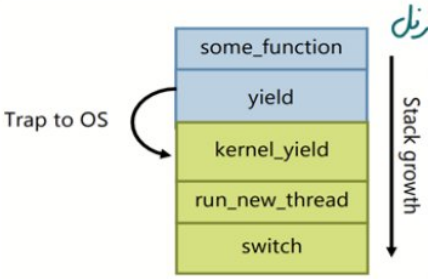
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

thread_handle: وقتی دستور thread_fork اجرا می‌شود یک شناسه‌ای برای دسترسی به آن ترد برمی‌گرداند.

thread_exit(): تردی که این دستور را اجرا می‌کند پایان کار خودش را به os اطلاع میدهد و وضعیت اش terminate می‌شود.

thread_yield(): تردی که این دستور را اجرا می‌کند بصورت داوطلبانه قبل از اجرا یک سیستم کال یا قبل از تمام شدن زمانش (قبل از آمدن وقفه تایمر) cpu را رها می‌کند، در اینصورت وضعیت این ترد از running به ready تغییر می‌کند.



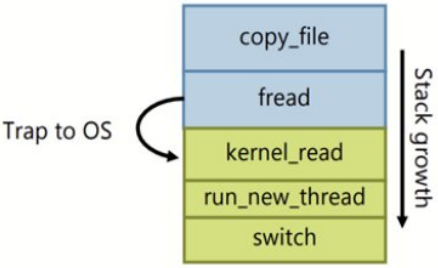
در واقع وقتی تابع yield را فراخوانی می‌کنیم درون تابع yield تابع context switch درون کرنل را فراخوانی می‌کند، پس در واقع تابع yield یک ApI و یک سیستم کال است که ما از ناحیه کاربر تابع cs کرنل را صدا می‌زنیم

با فراخوانی yield، Kernel yield اجرا می‌شود و Kernel yield نیز run_new_thread را اجرا می‌کند که در آن تابع switch قرار دارد.

```
run_new_thread() {
        newThread = select_next_thread();
        switch(curThread, newThread);
        thread_house_keeping(); /* do any cleanup */
}
```

وقتی run_new_thread اجرا می‌شود عمل cs انجام می‌شود و در cs مقدار pc عوض می‌شود و بنابراین خط آخر اجرا نمی‌شود و pc مربوط به ترد جدید است و تابع thread_house_keeping فقط وقتی اجرا می‌شود که یک ترد exit کند.

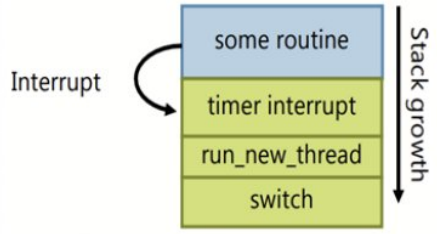
switch ترد وقتی I/O فراخوانی می‌شود



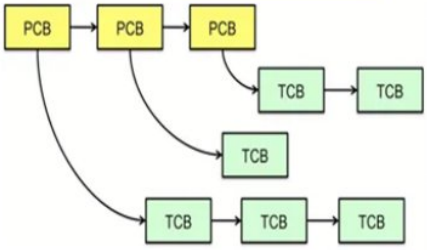
وقتی فایلی را می‌خواهیم کپی کنیم فراخوان سیستمی به کرنل داریم و در کرنل، تابع Kerne_read اجرا میشه و این تابع در انتهای کارش که دستور خواندن را مثلاً از هارد صادر می‌کند می‌آید و تابع run_new_thread را انجام میدهد.

یعنی در پیاده سازی توابع (handler های) I/O باید در انتهایش تابع context switch فراخوانی شود

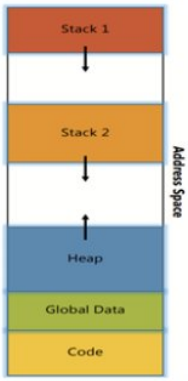
switch ترد وقتی وقفه می‌آید



در وقفه‌ها هم ابتدا اجرا ترد جاری متوقف می‌شود و سپس روتین وقفه مذکور انجام می‌شود و در انتهای آن روتین باید تابع run_new_thread اجرا شود.



نکته) اطلاعات اختصاصی هر ریسمان در بلاک TCB خودش ذخیره می‌شود و هر pcB (بلاک فرآیند) به TCB های مربوط به تردهایش اشاره می‌کند

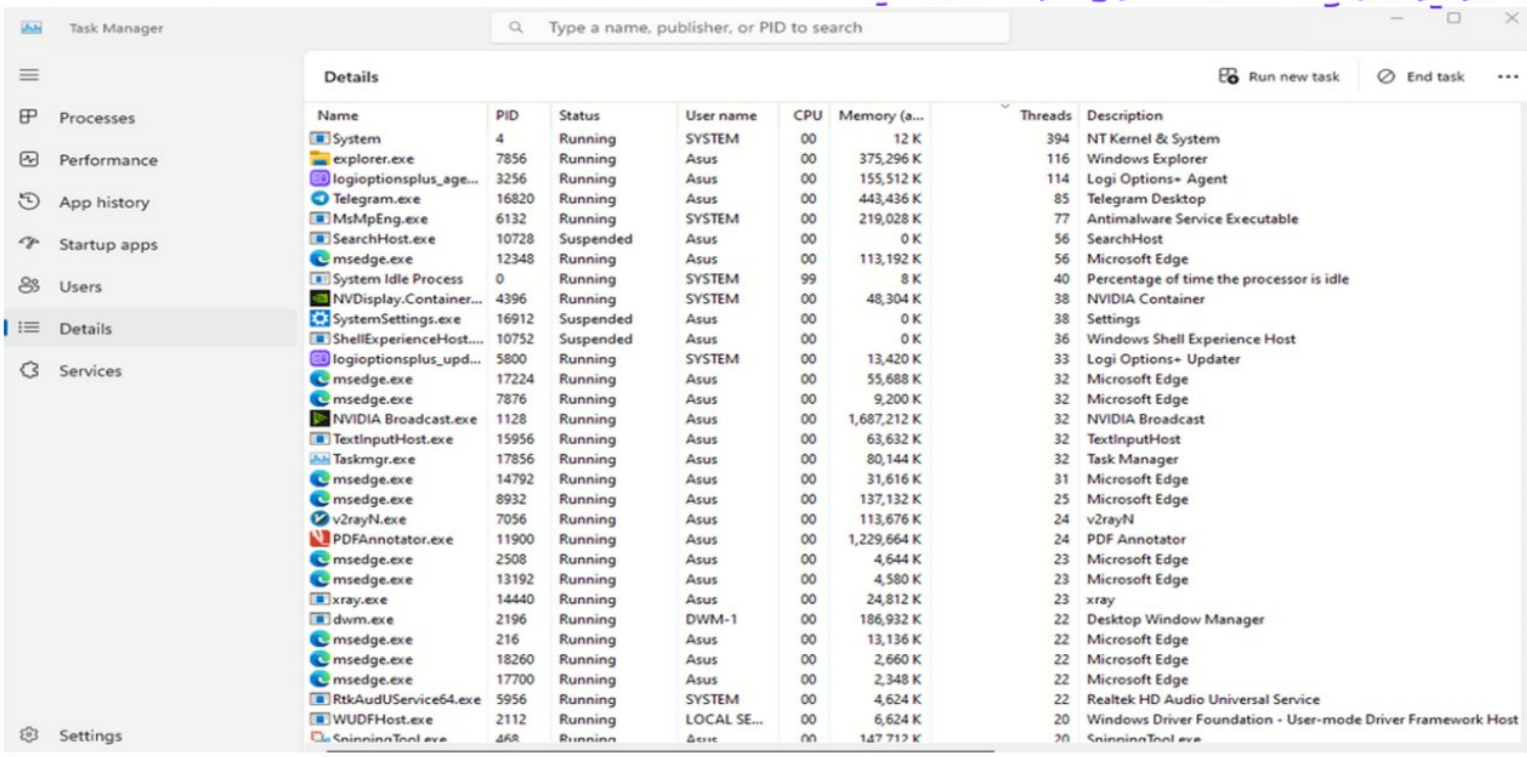


نکته) گفتیم هر ترد مسیر اجرایی مجزای خودش را دارد و بنابراین هر ترد باید استک خودش را داشته باشد، حالا هر پشته می‌تواند به سمت آدرس های پایین اش رشد کند و میکند اینقدر رشد کند که بزند و پشته یک ترد دیگر را خراب کند که باید حواسمون به این مورد باشه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

- در زیر می‌توانید تعداد تردهای فرآیندها را مشاهده کنید.



Name	PID	Status	User name	CPU	Memory (a...	Threads	Description
System	4	Running	SYSTEM	00	12 K	394	NT Kernel & System
explorer.exe	7856	Running	Asus	00	375,296 K	116	Windows Explorer
logioptionsplus_age...	3256	Running	Asus	00	155,512 K	114	Logi Options+ Agent
Telegram.exe	16820	Running	Asus	00	443,436 K	85	Telegram Desktop
MsMpEng.exe	6132	Running	SYSTEM	00	219,028 K	77	Antimalware Service Executable
SearchHost.exe	10728	Suspended	Asus	00	0 K	56	SearchHost
msedge.exe	12348	Running	Asus	00	113,192 K	56	Microsoft Edge
System Idle Process	0	Running	SYSTEM	99	8 K	40	Percentage of time the processor is idle
NVDisplay.Container...	4396	Running	SYSTEM	00	48,304 K	38	NVIDIA Container
SystemSettings.exe	16912	Suspended	Asus	00	0 K	38	Settings
ShellExperienceHost....	10752	Suspended	Asus	00	0 K	36	Windows Shell Experience Host
logioptionsplus_upd...	5800	Running	SYSTEM	00	13,420 K	33	Logi Options+ Updater
msedge.exe	17224	Running	Asus	00	55,688 K	32	Microsoft Edge
msedge.exe	7876	Running	Asus	00	9,200 K	32	Microsoft Edge
NVIDIA Broadcast.exe	1128	Running	Asus	00	1,687,212 K	32	NVIDIA Broadcast
TextInputHost.exe	15956	Running	Asus	00	63,632 K	32	TextInputHost
Taskmgr.exe	17856	Running	Asus	00	80,144 K	32	Task Manager
msedge.exe	14792	Running	Asus	00	31,616 K	31	Microsoft Edge
msedge.exe	8932	Running	Asus	00	137,132 K	25	Microsoft Edge
v2rayN.exe	7056	Running	Asus	00	113,676 K	24	v2rayN
PDFAnnotator.exe	11900	Running	Asus	00	1,229,664 K	24	PDF Annotator
msedge.exe	2508	Running	Asus	00	4,644 K	23	Microsoft Edge
msedge.exe	13192	Running	Asus	00	4,580 K	23	Microsoft Edge
xray.exe	14440	Running	Asus	00	24,812 K	23	xray
dwm.exe	2196	Running	DWM-1	00	186,932 K	22	Desktop Window Manager
msedge.exe	216	Running	Asus	00	13,136 K	22	Microsoft Edge
msedge.exe	18260	Running	Asus	00	2,660 K	22	Microsoft Edge
msedge.exe	17700	Running	Asus	00	2,348 K	22	Microsoft Edge
RtkAudUService64.exe	5956	Running	SYSTEM	00	4,624 K	22	Realtek HD Audio Universal Service
WUDFHost.exe	2112	Running	LOCAL SE...	00	6,624 K	20	Windows Driver Foundation - User-mode Driver Framework Host

Firefox در مقابل chrome

- Firefox برای هر تب یک ترد ایجاد می‌کند
- کروم: برای هر تب یک فرآیند ایجاد می‌کند
  - دسترسی به دیسک و ترافیک شبکه از یک فرآیند مرکزی بنام Browser استفاده می‌کند
  - نمایش و تعامل با کاربر برای هر تب یک فرآیند Renderer ایجاد می‌کند

هر add-on در کروم یک فرآیند جداست.

مقایسه این دو مرورگر از نظر:

- مصرف حافظه: firefox بهتر است، چون هر تبی که باز می‌کند بخش کد و دیتا و هیپ اش و اطلاعاتی که از روی سایت‌ها نگهداری می‌شود مشترک است.
- سرعت جابجایی بین تب‌ها: firefox بهتره چون جابجایی بین تب‌ها در واقع cs است و در مایکروفاکس داریم بین ۲ ترد جابجا می‌شویم ولی در کروم باید بین فرآیندها سوئیچ کنیم
- استفاده از منابع مشترک مثل کش سایت‌ها: firefox سریعتر و راحت‌تر است، چون اطلاعات کش معمولا توی هیپ ریخته می‌شود و هیپ نیز بین ترد‌ها مشترک است.
- شروع به کار سریعتر: firefox بهتر است، وقتی firefox را باز می‌کنید به جای لود شدن چندین فرآیند یک فرآیند لود می‌شود و کارها درون آن فرآیند انجام می‌شود
- امنیت: کروم بهتر است، چون فضای اجرا تب‌ها از هم جداست، در firefox تب‌ها ممکن است بتوانند به حافظه هم دسترسی داشته باشند

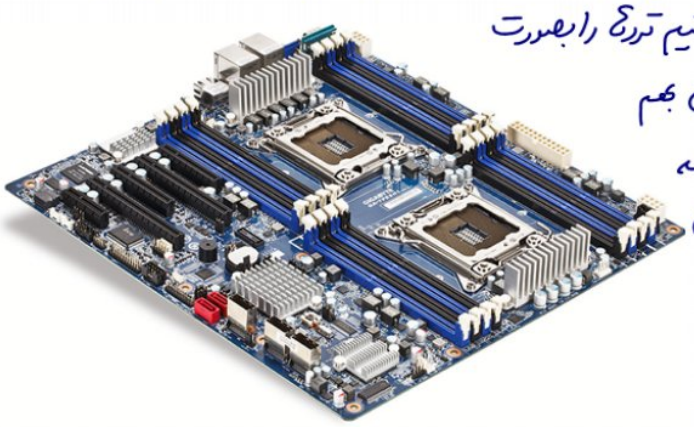
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

- اطمینان اجرایی بالا، کروم بهتر است چون فضای آدرس تب‌ها مجزاست و وقتی یک تب از کار می‌افتد فقط آن تب را می‌توانید Kill کنید بدون اینکه کل مرورگر شما بسته بشه اما توی firefox ممکن هست یک ترد بزند و حافظه تردهای دیگه که در واقع بقیه تب‌های دیگر ما هست را خراب کند و کلا مرورگر firefox از کار بیفتد

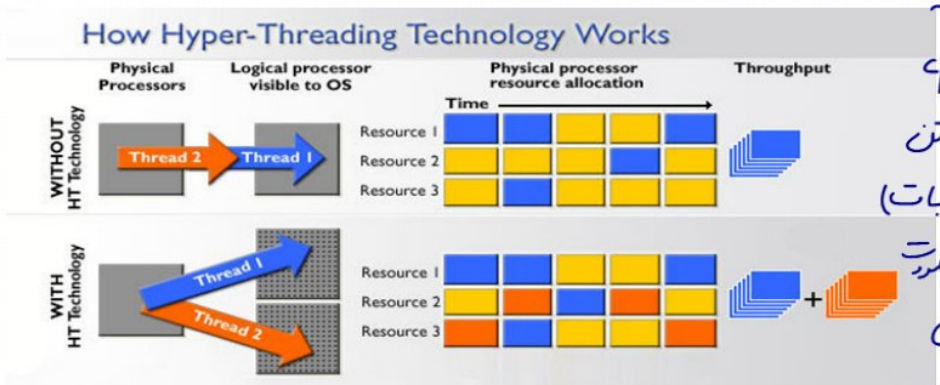
پشتیبانی سخت افزاری از موازات

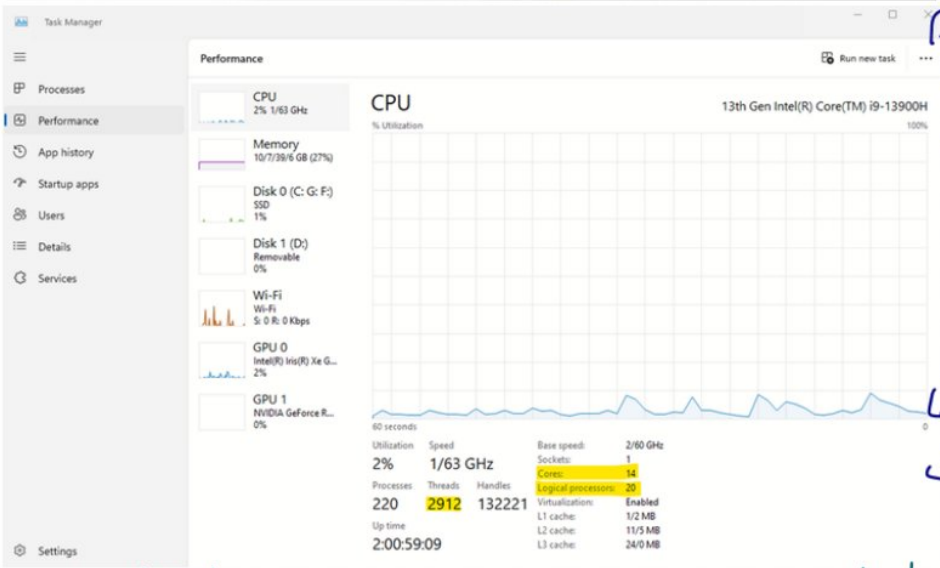
ابتدا یک پردازنده تک هسته‌ای داشتیم که در اینصورت فقط می‌توانستیم concurrency (همروندی) داشته باشیم، سپس سیستم‌های چند پردازنده، تک هسته‌ای آمدند که در این حالت توانستیم تردها را بصورت موازی اجرا کنیم، در این مدل پردازنده‌ها از طریق یک Bus اختصاصی بهم وصل اند. توجه داشته باشید تکلا یک os در سیستم داریم که همان os هم پردازنده‌ها را مدیریت می‌کند. ما در بوردهای چند پردازنده‌ای بیشتر برای سرورها کاربرد داشتند



پردازنده‌های مالتی ترد، تردهای سخت افزاری (logical core)

تا یک زمانی تعداد تردهای یک پردازنده یکی بود. برای اجرای یک ترد ما درکنار محتوای حافظه نرم‌افزار مربوط اش که در RAM قرار دارد به یکسری ثبات در cpu برای اجرا شدنش نیاز است، بنابراین اومدن و گفتن یک پردازنده به‌جای یک register set (مجموعه ثبات) بیاد و چندتا register set داشته باشیم که در اینصورت می‌توانیم خیلی سریع بین تردها switch کنیم بدون اینکه لازم باشه رجیسترهای مربوط به تردجاری را بخوایم درهمان لحظه بریم توی RAM بذاریم و داده مربوط به ترد بعدی را لود کنیم و کافی است os به cpu موقع switch بگوید از مجموعه ثبات y به‌جای x استفاده کن.





تردهایی که درون یک پردازنده لود شدند می‌توانند از فرآیندهای مختلفی باشند و نیازی نیست که حتما تردهای متعلق به یک فرآیند خاص در تردهای سخت افزاری گذاشته بشوند.

مزیت logical thread این است که context switch بسیار سریع می‌شود و درنتیجه بهره وری (utilization) منابع افزایش پیدا می‌کند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

دانستنی : مالتی ترد سخت افزاری را می توان در تنظیمات BIOS غیر فعال کرد

زمانبندی تردهای سخت افزاری توسط زمانبند سخت افزار

۱- coarse-grained : روش مالتی ترد راند روبین : فرض کنید ۴ مجموعه ثبات دارد که در هر مجموعه یک ترد لود شده است، حالا ترد شماره ۱ را تا زمانی که سیستم کال مسدودکننده نزد و یا دستور طولانی اجرا نکند اجرا می کنیم، حالا وقتی ترد ۱ رفت توی حالت انتظار می دیم سراغ ترد بعدی

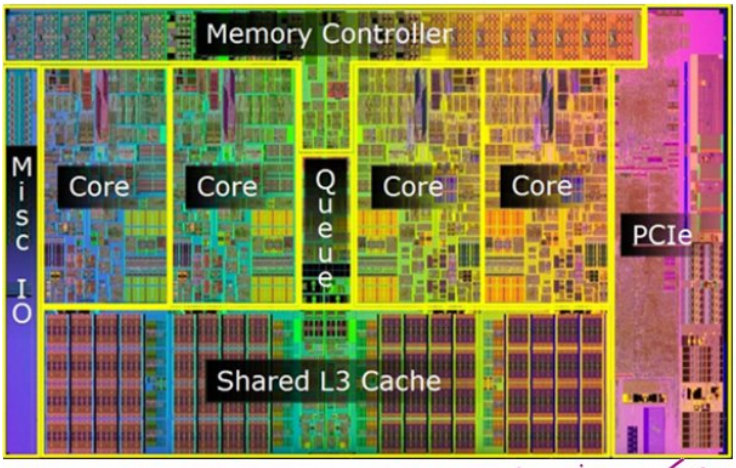
۲- fine-grained : " " " ریز : اگر زمانبند سخت افزار بتواند ریزتر از بالا و در سطح دستورالعمل (instruction) زمانبندی را انجام بدهد.

۳- SMT : بعدا توضیح میدهم.

توجه) بعدا بصورت دقیق تر و درست تر این سه مدل را توضیح خواهم داد.

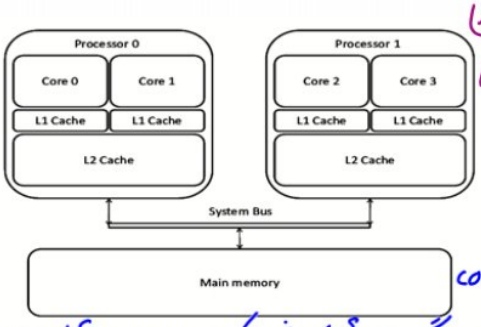
نکته) اینتل به استفاده از تکنولوژی SMT، Hyper threading می گوید

Multicore



در یک cpu چندین core (تراشه cpu) وجود دارد. در پردازنده هایی که تردهای سخت افزاری را کار می کنند درسته که ما چند مجموعه ثبات داریم ولی رسورس های درون پردازنده مثل Alu و جمع کننده و ضرب کننده و... به تعداد محدود و مشخص است و فقط registerset ها تکرار شدند، اما در پردازنده های multicore، هر core رسورس های مجزای خودش را دارد.

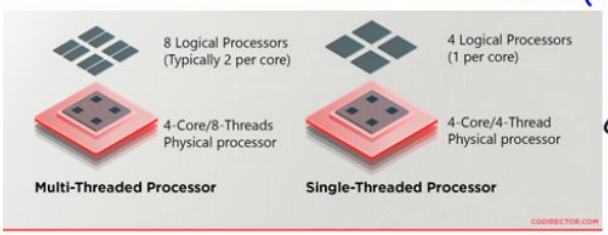
- معمولا core ها از طریق کش لایه سه باهم ارتباط دارند (البته اگر ۳ لایه کش داشته باشند) و بنابراین می شود داده های بزرگ را در کش لایه ۳ قرار داد و core ها بصورت مشترک از آنها استفاده کنند.



بعضی وقت ها core ها از طریق کش لایه ۲ باهم در ارتباط اند که در شکل روبه رو می بینید

نکته) تفاوت multicore با سیستم های چند پردازنده این است که در مالتی کور چون core ها به هم نزدیک هستند هم قیمت کاهش پیدا می کند و هم ارتباط بین core ها سریعتره و هم اگر داده های مشترکی بین هسته ها استفاده کنیم این داده مشترک را می توان در کش لایه سه گذاشت و به دلیل سرعت خیلی زیاد کش این ارتباط خیلی سریع میشه

نکته) هر core می تواند خودش شامل چند logical core (چند ترد سخت افزاری) باشد.



آیا کرنل چند تردی است؟

تا حدود ۱۲ سال پیش هسته های os بصورت تک تردی قابل اجرا بود ولی وقتی core ها زیاد شد دیدند اگر کرنل تک تردی بخواد اجرا بشه مشکل پیش میاد و os گلوگاه میشه و دیدند دیگه افزایش core فایده نداره و در واقع جلوی scalability رو گرفته بود.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

وقتی تعداد core ها از یک تعداد مشخص بیشتر شود و اگر os بخواهد تک تردی اجرا بشود کرنل به انجام کارهای این core های موازی نمی رسد. چون هر core دارد دستور دیگری را اجرا می کند و هر کدام از اینها سیستم کال میدهند و با کرنل کار دارند و اگر کرنل بخواهد دستورات هر یک از core ها را بصورت سریال انجام دهد خودش گلوگاه می شود و اصلا نقض غرض به وجود می آید و پس چرا اصلا multicore راه انداختیم، بنابراین کرنل رو هم بصورت موازی پیاده سازی کردند.

کرنل ترد های مجزایی برای کارهای مختلف می تواند داشته باشه

۱- برای هر فرآیند کاربری می تواند یک ترد جدا داشته باشه

۲- انجام یک I/O می تواند چند گام داشته باشه، os می تواند هر گام را با 1 ترد پیاده سازی کند تا بتواند به تعداد بیشتری از درخواست ها جواب بدهد

۳- os برای ارتباط با هر یک از Driver ها می تواند ترد مجزا داشته باشه

مالتی تردینگ همگام و ناهمگام

خودتون یاداشت کنید. احسنت 😊

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

cpu scheduling

در این فصل می خواهیم در خصوص زمانبندی پردازنده صحبت کنیم، فقط خواستون باشه که منظور از پردازنده هر منبعی در کامپیوتر هست که برای آن پارامتر سرعت مطرح است، اما ما بیشتر روی زمانبندی cpu تمرکز می کنیم.

در زیر به یادآوری این مفهوم می پردازیم:

۱- processor or Active Recource (پردازنده)

به یک ریسورس Active گوییم اگر پارامتر سرعت برای آن ریسورس مطرح باشد.

مثال هایی از ریسورس Active: cpu ⋮ شبکه (کارت شبکه): برایش سرعت قائلیم و می گوییم پهنای باند شبکه ما اینقدر است

Hard Disk: یک نرخ انتقال داده از هد دیسک به سایر اجزا سیستم وجود دارد که می تواند به سرعت چرخش صفحه دیسک (مثلاً ۷۲۰۰ RPM - RPM=Revolutions Per minute) و کابل انتقال هد دیسک به مادربرد و seek time (زمان جستجو شیار: مدت زمان حرکت بازوی دیسک و قرارگرفتن هد روی شیار مورد نظر) بستگی دارد ← پس نقط cpu scheduling نداریم

cpu scheduling
DisK scheduling
Network "

* هر پردازنده را می توان زمانبندی کرد (scheduling) بنابراین واژه ای بنام processor scheduling داریم

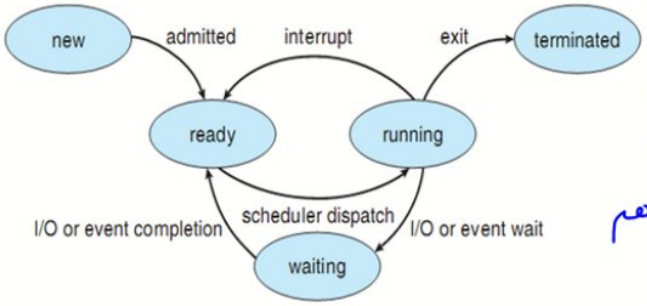
scheduling به معنای زمانبندی و بعد از آن نوبت دهی یک معنا عام است و به این معناست که یک ریسورس در زمان های مختلف در اختیار چه کسی باشد

۲- passive Resource: منابعی که پارامتر سرعت برایشان مطرح نیست.

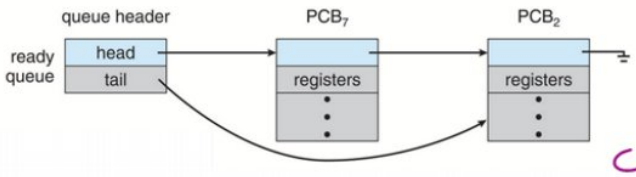
فضای دیسک و فضای حافظه نمونه هایی از این نوع منبع هستند، مثلاً یک رم ۳۲GB یا یک هارد ۴TB یک passive resource هستند.

زمانبندی کدوم صف ها رو می خواهیم بررسی کنیم؟

صف ready.



هر فرآیند در طول حضورش در سیستم در هر زمان در یکی از state های متابل قرار می گیرد که یکی از این state ها، حالت ready بود که وقتی یک فرآیند در این حالت قرار دارد در صف ready در کنار بقیه فرآیندهایی که آنها هم در این حالت هستند قرار می گیرد.



- حالا cpu scheduler که یکی از قسمت های اصلی os است کارش این هست که از بین فرآیندهایی که در صف ready هستند (از بین فرآیندهایی که در حالت ready قرار دارند) فرآیند بعدی ای که باید cpu در اختیارش قرار داده شود را انتخاب کند و حالت اش را به running تغییر بدهد.

توجه) وقتی می گوییم یک فرآیند در صف ready است یعنی pcb اش در صف ready قرار دارد و بنابراین pcb اش در صف های مربوط به I/O Device های درون سیستم نیست.

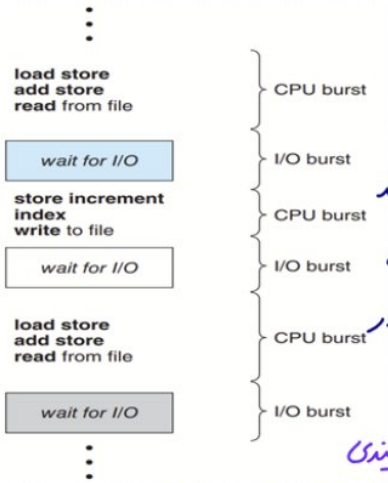
یادآوری: صف ها بصورت لیست پیوندی (linked list) پیاده سازی می شوند.

cpu and I/O Burst

همان طور که قبلا گفتیم یک برنامه از توالی از cpu burst و I/O burst به وجود آمده و اجرای یک فرآیند شامل چرخه ای از

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



cpu burst ها و I/O burst ها است، اجرا یک فرآیند با cpu burst آغاز می‌شود و به دنبال آن I/O burst خواهیم داشت و این روند ادامه دارد.

cpu burst: یک برنامه تک‌تودی تعدادی دستور پردازشی دارد که این دستورات را روی cpu اجرا می‌کند

I/O burst: یک برنامه پس از انجام یکسری دستور محاسباتی احتمالا یک عمل I/O انجام میدهد و در نتیجه یک مدت زمانی در state ـ بلاک منتظر انجام عمل I/O است، بنابراین این فرآیند در این مدت در در I/O burst خودش قرار دارد.

cpu-bound process: پراسی که اکثر وقت اش صرف عملیات پردازشی شود. اگر فرآیندی دارای cpu burst های بزرگی باشد خیلی کم به سراغ عمل I/O می رود و زمان زیادی صرف عملیات پردازشی اش می شود

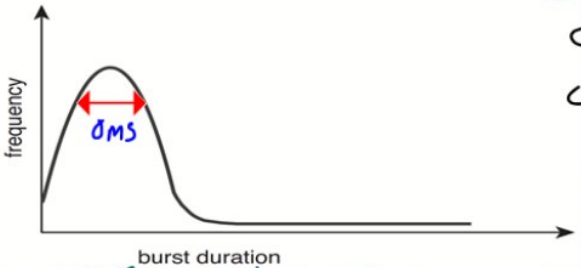
" I/O-bound: " " " " " انتظار برای I/O شود ⇐ I/O های زیادی داشته باشد

همان طور که از روی شکل مشخص است اگر فرآیندی دارای cpu burst های خیلی کوچکی باشد مکررا به سراغ عمل I/O می رود و زمان کمی صرف عملیات پردازشی اش می شود

به این نکته مهم توجه کنید که وقتی یک فرآیند در cpu burst اش است در state ـ ready و در نتیجه در صف ready قرار دارد و در این شرایط است که cpu scheduler می تواند این فرآیند را انتخاب کند.

- اگر همه فرآیندها I/O-bound باشند ready queue تقریبا همیشه خالی خواهد بود و cpu scheduler کار چندانی برای انجام دادن نخواهد داشت و اگر همه فرآیندها cpu-bound باشند صف انتظار برای I/O ها تقریبا همیشه خالی می ماند در اینصورت سیستم unbalance خواهد شد بدین معنی که دستگاه ها بلا استفاده می مانند و بصورت بهینه مورد استفاده قرار نمی گیرند.

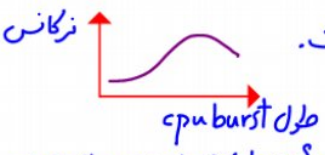
Histogram of cpu-burst duration



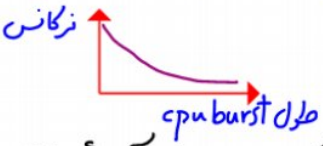
این نمودار دارد فرکانس (پراکندگی) cpu burst یک فرآیند را بصورت میانگین نشان میدهد. اومدن تعداد بسیار زیادی فرآیند رو روی کامپیوترهای مختلف اجرا کردن و این نمودار را بدست آوردن، درسته که از یک فرآیند به فرآیند دیگه و از یک کامپیوتر به کامپیوتر دیگه متفاوت میشه اما بطور کلی نمودارشون مشابه رو به رو هست.

- در این نمودار محور عمودی فرکانس رخ دادن های طول cpu burst های یک فرآیند را نشان میدهد و محور افقی طول cpu burst ها را نشان میدهد

- اگر یک فرآیند cpu bound باشه دارای تعداد کمی cpu burst کوتاه و دارای تعداد زیادی cpu burst های طولانی است، بنابراین فراوانی cpu burst های طولانی در آن زیاد است بنابراین هیستوگرام cpu burst duration اش بصورت زیر است.



- اگر یک فرآیند I/O bound باشه دارای تعداد زیادی cpu burst کوتاه و دارای تعداد کمتری cpu burst های طولانی است، بنابراین هیستوگرام cpu burst duration اش بصورت زیر است.



- می تونیم بگیم هیستوگرامی که در بالا داشتیم هیستوگرام فرآیندهای مرسوم را نشان می دهد و این نمودار نشان میدهد که بسیاری از فرآیندها دارای تعداد زیادی cpu burst کوتاه در حد چند میلی ثانیه و تعداد کمی cpu burst طولانی است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

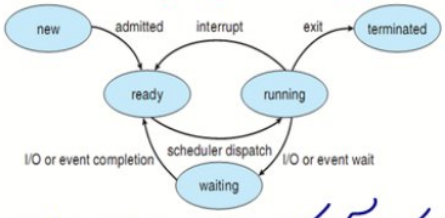
این توزیع burst time ها می‌تواند هنگام اجرای cpu scheduling algorithm مهم باشد.

چه زمان‌هایی cpu scheduler فعال می‌شود

cpu scheduler باید از بین فرآیندهای درون صف ready فرآیند بعدی را انتخاب کند، حالا سوال اینجاست که چه زمانهایی باید cpu scheduler فعال بشه (روی cpu اجرا بشه) و فرآیند بعدی رو انتخاب کند؟

درواقع می‌خواهیم بدانیم عمل Dispatch بلافاصله بعد از وقوع چه رخدادهایی می‌تواند رخ دهد؟

هر وقت cpu خالی شود ← پس ببین چه فلش‌هایی از حالت running خارج شده و دلیل آن فلش‌ها چه می‌تواند باشد



در اثر ۳ تا از transition هایی زیر cpu خالی (رها) می‌شود:

۱) preempted (که به ۲ دلیل رخ میدهد)

۲) exit: فرآیند درحال اجرا بوده که تمام می‌شود.

۳) I/O or event wait: یکی از زمانهایی که cpu scheduling فعال می‌شود زمانی است که یک فرآیند که درحالت running است به دلیل فراخوانی یک عمل I/O یا به دلیل wait کردن برای یک event از حالت running خارج می‌شود و به حالت waiting می‌رود.

توجه) درحالت ۲ و ۳ خود فرآیند هست که cpu را رها می‌کند.

توضیح مورد ۱: در ۲ زمان دیگر نیز cpu scheduling باید فعال شود که در این ۲ حالت خود فرآیند cpu را رها نمی‌کند و os است که cpu را از فرآیند به زور می‌گیرد:

۱- timer interrupt: سهمیه زمانی که به فرآیند داده شده تمام شود

۲- I/O completion: اگر یک فرآیند که درحالت waiting بوده عمل I/O اش تمام شود و اولویت اش از تمامی فرآیندهایی که در حال حاضر در صف ready هستند بیشتر باشد این فرآیند که الان ready شده باید cpu را در اختیار بگیرد، بنابراین در این حالت هم باید cpu scheduler فعال بشه تا cpu را از فرآیند جاری بگیرد و به این فرآیندی که همین الان وارد صف ready شده بدهد.

نکته) اگر در هر ۴ حالت‌های قبلی امکان اجرا کردن cpu scheduling وجود داشته باشد می‌گیم زمانبندی preemptive (غیرانحصاری یا انقطاعی) است و اگر فقط در موارد ۲ و ۳ امکان اجرا کردن cpu scheduling وجود داشته باشد می‌گیم زمانبندی non-preemptive (انحصاری یا غیرانقطاعی است). بنابراین ۲ دسته زمانبند یا الگوریتم‌های زمانبندی وجود دارد.

نکته) وقتی از preemptive scheduling استفاده می‌کنیم cpu غیر انحصاری است و یک فرآیند علاوه بر اینکه خودش می‌تواند cpu را رها کند (تحویل بدهد) ممکن است os، cpu را از آن به زور بگیرد، بنابراین در این روش cpu در انحصار آن فرآیند نیست.

نکته) به nonpreemptive، cooperative نیز می‌گویند.

نکته) تحت nonpreemptive scheduling وقتی cpu به یک فرآیند تخصیص داده می‌شود، فرآیند cpu را تا زمانیکه تمام شود یا با رفتن به حالت waiting آزاد نکند در اختیار خودش نگه می‌دارد.

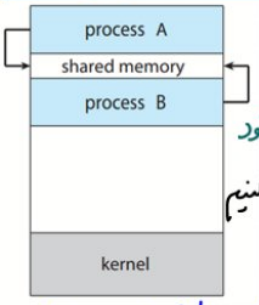
نکته) تقریبا همه os های مدرن از جمله ویندوز، لینوکس، یونیکس و macos از preemptive scheduling algorithm ها استفاده می‌کنند.

عوارض و پیامدهای استفاده از preempt scheduling

اگر حواسمان به عوارض ناشی از preemptive scheduling نباشد ممکن است با مشکلات جدی ای مواجه شویم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



۱_ synchronization: ناسازگاری (inconsistency) داده‌های مشترک را باید کنترل کنیم

preemptive scheduling زمانیکه داده‌ها بین چند فرآیند share شده اند می‌تواند منجر به race condition (سابقه) شود

race condition: یک رقابت بین ۲ فرآیند که دارند بصورت مشترک به یک داده دسترسی پیدا می‌کنند و ما باید کاری کنیم که این مشکل پیش نیاید.

فرض کنید فرآیند A,B یک حافظه را با هم share کردند، حالا فرض کنید در یک قسمت از این فضای share شده یک آرایه به طول ۲۰ وجود دارد و فرآیند A باید همه درایه‌های این آرایه را در عدد c ضرب کند و در انتها فرآیند B از این آرایه برای کار خودش استفاده کند، حالا فرض کنید که A تا عنصر ۱۰ این کار را انجام می‌دهد و سپس بدلیل preemption پردازنده به فرآیند B میرسد و فرآیند B نخواهد دید تا این آرایه را بخواند داده از یک آرایه inconsistence می‌خواند، پس در این حالت چند تا کار می‌توان انجام داد، مثلاً:

۱_ اگر قراره زمان به B داده بشه باید کاری کنیم که B از این موضوع مطلع باشد و نتواند از این آرایه که عمل نوشتن در آن تکمیل نشده عمل خواندن را انجام دهد و کاری کنیم که B تا زمانیکه A نوشتن در این آرایه را تکمیل می‌کند صبر کند، یعنی با اینکه زمان می‌رسد به B ولی B نتواند از این حافظه بخواند و مجاز به این کار نباشد که این کار همان بحث synchronization هست.

۲_ اصلا توی این حالت زمان را به B ندهیم و هر وقت فرآیند A تغییرات اش را کامل کرد زمان به B داده شود.

۲_ باید حواسمان به preempt شدن کرنل باشد چون preemption وسط یک سیستم کال ممکن است منجر به ناسازگاری شود به همین دلیل گاهی اوقات کرنل os ها را بصورت nonpreemptive تا از هرج و مرج جلوگیری کنند.

preemption بر طراحی os تاثیر می‌گذارد.

فرض کنید در طراحی os مان اجازه preempt شدن کرنل os را داریم، حالا این preemption اگر در هر نقطه ای بتواند رخ بدهد می‌تواند عوارض بدی داشته باشد، مثلاً فرض کنید یک فرآیند یک سیستم کال I/O را فراخوانی کرده و در نتیجه یکی از کارهایی که کرنل باید انجام دهد این است که کرنل باید pcb این فرآیند را از صف ready به صف مربوط به آن device منتقل کند، حالا فرض کنید کرنل در وسط انجام (اجرا) این کار است که وسط این کار preempt می‌شود در اینصورت ممکن است pointer های لیست های پیوندی صف ready و صف مربوط به آن I/O Device خراب بشوند. چطور؟ مثلاً فرض کنید در لحظه‌ای که لیست پیوندی صف ready به لیست پیوندی صف I/O device وصله و هنوز تنظیمات مربوط به اشاره گرها کامل نشده و در همین لحظه کرنل preempt شود، حالا cpu scheduler باید فرآیند بعدی را از صف ready انتخاب کند، حالا وقتی cpu scheduler لیست پیوندی صف ready را برای انتخاب فرآیند بعدی دنبال میکند ممکن است اشتباهاً به یکی از pcb هایی که در صف I/O device وجود دارد برسد و فرآیندی که بلاک است و ready نیست را برای گذاشتن روی cpu انتخاب کند.

⭐ پس اگر کرنل در حال تغییر داده‌های مهم مثل صف‌های I/O باشد و وسط این تغییرات preemption رخ بدهد و بعد از آن هسته یا درایور یک دستگاه I/O نیاز به خواندن یا اصلاح آن داده‌های مهم باشد ممکن است هرج و مرج رخ بدهد به همین دلیل در قدیم اکثر کرنل‌های هسته بصورت nonpreemptive پیاده سازی می‌شدند و به اینصورت پیاده سازی می‌کردند که وقتی می‌خواستند وارد این بخش از کرنل شوند وقفه‌ها را Disable می‌کردند تا چنین مشکلاتی که ممکن است کل سیستم را دچار مشکل کند رخ ندهد، این بخش از کرنل معمولاً دستورالعمل‌های کمی داشتند، البته بعدا متوجه شدند که این روش پیاده سازی سیستم‌های real time را که باید تعامل جدی‌ای با محیط داشته باشند و به موقع به اتفاقات محیط reaction نشان بدهند با مشکل جدی مواجه می‌کند، چون ممکن است وقتی وقفه‌ها را غیر

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

فعال کردیم اتفاق مهمی در محیط رخ دهد ولی ما نتوانیم به موقع به آن اتفاق واکنش نشان بدهیم.

نکته) کرنل os را می‌توان بصورت preemptive یا nonpreemptive طراحی کرد.

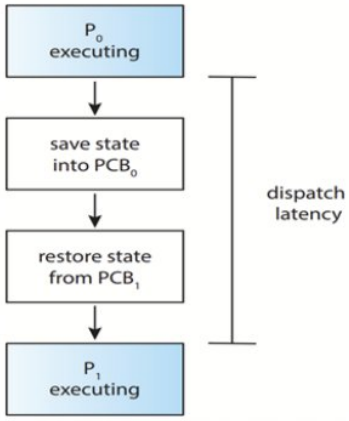
نکته) به دلایل گفته شده در خیلی از os ها در حالت (امکان) کامپایل بصورت preemptive و nonpreemptive برای کرنل قرار دارند که اگر ما بر اساس امکان preemptive کرنل را کامپایل کنیم آن بخش‌هایی از کدهای کرنل که در آنها وقفه‌ها غیرفعال می‌شوند به حداقل ممکن رسیده است و فقط در وقت‌های بسیار حساس این اتفاق می‌افتد.

نکته) یک preemptive Kernel به مکانیزم‌هایی مانند mutex lock برای جلوگیری از race condition هنگام دسترسی به داده‌های مشترک هسته نیاز دارد.

نکته) اکثر سیستم عامل‌های مدرن امروزی هنگام اجرا در Kernel mode بصورت Fully preemptive هستند.

Dispatcher

Dispatcher یکی دیگر از اجزای درگیر در عملکرد cpu scheduling ماژولی است که کنترل core ـ cpu را به فرآیندی که توسط cpu scheduler انتخاب شده است می‌دهد.



The role of the dispatcher.

کار Dispatcher: ۱- تغییر (سوئیچ کردن) context از یک فرآیند به فرآیند دیگر
۲- " " به user mode
۳- پرش به مکان مناسب در برنامه کاربر برای از سرگیری آن

Dispatcher در هر context switch فراخوانی می‌شود بنابراین باید تا حد امکان سریع باشد.

Dispatch latency: تاخیر ناشی از سوئیچ کردن از فرآیند جاری به فرآیند بعدی.

معیارهای مقایسه الگوریتم‌های زمانبندی

۱- cpu utilization (بهره وری پردازنده): درصد استفاده مفید از cpu

$U_{cpu} = \frac{\text{زمان استفاده مفید از cpu}}{\text{زمان کل}} \times 100\%$

زمان کل = غیرمفید + مفید

غیرمفید = cpu idle + dispatch time

نکته) اگر از Time sharing استفاده کنیم بین فرآیندها context switch داریم و Dispatch latency باعث می‌شود که بخشی از زمان cpu هدر برود و بهره وری پردازنده قطعا زیر ۱۰۰٪ خواهد بود، پس اگر بخواهیم cpu utilization ما نزدیک ۱۰۰٪ بشود باید time sharing نداشته باشیم که os کمی داشته باشیم، که الگوریتم‌هایی مثل FcFs در این حالت استفاده می‌شود.

نکته) در اکثر سیستم‌های واقعی utilization بین ۴۰٪ درصد در slightly loaded system (بار کافی برای اینکه از cpu بصورت خیلی مفید استفاده کنیم وجود ندارد) یا زمانهایی که می‌خواهیم مقداری از پتانسیل cpu را زرد کنیم که اگر load سنگینی که برای ما مهم است رخ داد بتوانیم به آن به موقع پاسخ بدهیم، تا ۹۰٪ برای heavily loaded system.

۲- Throughput (گذردهی یا توان عملیاتی): تعداد کار انجام شده در واحد زمان

$\text{Throughput} = \frac{\text{تعداد کارهای انجام شده}}{\text{زمان کل انجام آن کارها}}$

نکته) اگر cpu burst کارها کوتاه باشد تعداد اجرا شدنشان در واحد زمان افزایش پیدا می‌کند و بالعکس پس گذردهی به طول cpu burst کارها ربط دارد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۳- Turnaround time (زمان تکمیل، زمان برگشت، زمان گردش): از زمانیکه درخواست اجرای یک فرآیند داده می شود تا زمانیکه آن فرآیند خاتمه پیدا می کند و سیستم را ترک می کند، این زمان شامل زمانهای (بارشدن در حافظه)، انتظار در صف ready، انتظار در صف I/O و زمانهای اجرا است

$Turnaround\ Time = T_{out} - T_{in}$

توجه) T_{in} زمان ورود فرآیند به سیستم است نه اولین زمانی که cpu گرفته و T_{out} هم زمان خروج فرآیند از سیستم است نه زمانی که فرآیند cpu را رها کرده.

۴- waiting time (زمان انتظار): مجموع مدت زمانیکه یک فرآیند در صف ready انتظار می کشد. ممکن است قبل از اولین بار اجرا فرآیند در صف ready انتظار بکشد و همینطور ممکن است دفعات دیگری هم خودش cpu را رها کند یا preempt بشود و باز در صف ready برای در اختیار گرفتن cpu انتظار بکشد

به این نکته مهم توجه کنید که وقتی یک فرآیند در cpu burst اش است در state = ready و در نتیجه در صف ready قرار دارد و در این شرایط است که cpu scheduler می تواند این فرآیند را انتخاب کند، به همین دلیل است که برای محاسبه WT ما زمان انتظار در صف ready را در نظر می گیریم.

۵- Response time (زمان پاسخ): مدت زمانیکه از ارسال درخواست تا تولید اولین پاسخ (خروجی) طول می کشد

شاید! مدت زمانیکه فرآیند برای دسترسی اول به cpu انتظار می کشد: زمان انتظار تا اولین واکنش cpu به یک فرآیند.

نکته) پس اگر یکبار cpu به این فرآیند داده شد ولی بارها بدلیل preemption مجددا در صف ready انتظار کشید دیگه زمان پاسخ عوض نمی شود

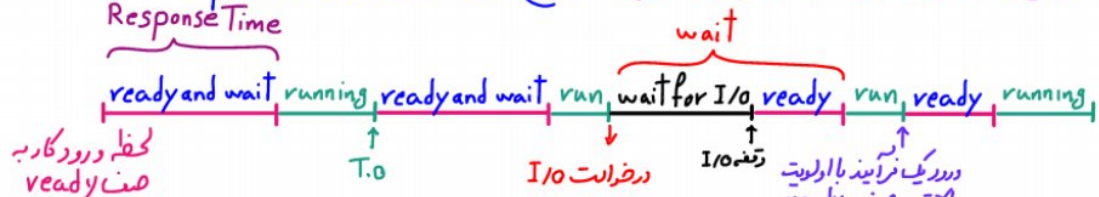
نکته) چرا معیار Response time تعریف شده؟

در یک سیستم تعاملی (Interactive system)، Turnaround time ممکن است بهترین معیار نباشد، اغلب یک فرآیند می تواند مقداری خروجی زود هنگام تولید کند و به محاسبه نتایج جدید ادامه دهد (به اجرایش ادامه دهد) این در حالی است که نتایج اولیه‌ای که این فرآیند تولید کرده یک فرد خوبی برای کاربر محسوب می شود (در نتیجه نتایج اولیه با کاربر تعامل به وجود آمده)، پس توجه کنید که زمان پاسخ زمانی است که تا شروع پاسخ دادن طول می کشد نه زمانیکه برای خروجی پاسخ لازم است.

نکته) در سوالات کنکور معمولا در خصوص اولین پاسخ (خروجی) صحبتی نمی کنند در این حالت فرض در مورد آن است که زمان پاسخ را تا اولین زمانی که فرآیند cpu را گرفته در نظر بگیریم ولی متاسفانه در اکثر سوالات کنکور فرض کردند خروجی اش را در انتهای اجرایش تولید کرده است و به همین دلیل در این شرایط می توان گفت که Response time تقریبا برابر Turnaround time می شود، چرا می گم تقریبا؟ (چون باز اینجا هم داریم از مدت زمان مثلاً نمایش یا چاپ صرف نظر می کنیم، در واقع RT<TT)

نکته) ما بدنبال ماکزیمم کردن ۲ معیار Throughput و utilization و مینیمم کردن معیارهایی که از جنس زمان هستند یعنی معیارهای ۳ و ۴ و ۵ هستیم.

۶- service time (زمان سرویس، زمان اجرا، زمان پردازش): مجموع زمان اجرای فرآیند روی cpu



* تایم لاین از نگاه یک فرآیند

مجموع سرویس = ST

مجموع قربزخ = WT

از اول تا آخر = TT

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

★ حالا توی Time line بالا اگر برای سادگی از زمان I/o burst صرف نظر کنیم خواهیم داشت:

Turnaround time = service time + waiting time ⇒ TT = ST + WT

نکته) یکی از راه‌های مناسب محاسبه WT به این صورت است: WT = TT - ST

نکته) Average TT یعنی چی؟ یعنی میانگین TT هم فرآیندهای داده شده و داریم: ATT = AST + AWT

برای محاسبه Average هریک از پارامترهای بالا باید آن پارامتر را به ازای هر فرآیند محاسبه کنیم و سپس میانگین بگیریم، یعنی هم مقادیر یک پارامتر را با هم جمع کنیم و بر تعداد فرآیندها تقسیم کنیم.

AWT یکی از معیارهای مقایسه الگوریتم‌های زمانبندی است.

چند معیار کلی

۱- توازن (Balance): ایجاد توازن بار در استفاده از همه منابع سیستم

۲- عدالت و انصاف (Fairness): عدالت به معنای مساوات نیست هرچند گاهی اوقات به این معنی نیز ممکن است به کار برود

عدالت یعنی به همه فرآیندها با توجه به اهمیت و اولویت‌شان یک کم از cpu بدهیم

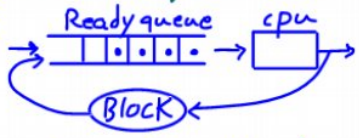
starvation (گرسنگی یا قحطی زدگی): انتظار طولانی بدون آنکه سقف داشته باشد.

معرفی الگوریتم‌های زمانبندی

۱- FcFs (First_come First serve): ساده‌ترین الگوریتم زمانبندی است. برخی از سوالات از این به اسم FIFo نام می‌برند که البته خیلی کار خوبی نیست :)

بین فرآیندهایی که در حالت ready هستند (در صف ready هستند) آن فرآیندی که زودتر وارد شده است اولویت بالاتری دارد.

توجه کنید که بالاخره هر فرآیندی یک زمانی وارد صف ready میشه دیگه، پس فرآیندهای تازه وارد در آخر صف ready قرار می‌گیرند، همین طور اگر یک فرآیند بلاک می‌شود، بعد از آنکه بیدار شد به انتهای صف می‌رود، توجه کنید که این الگوریتم انحصاری است



مثال) فرض کنید سه فرآیند P_1 تا P_3 در زمان ۰ به ترتیب وارد صف ready شدند با توجه به cpu burst های داده شده ترتیب اجرا فرآیندها و زمان شروع و پایان هر فرآیند و زمان انتظار هر فرآیند و میانگین زمان انتظار را بدست آورید

Process	Burst Time
P_1	24
P_2	3
P_3	3

- چون لحظه ورود فرآیند P_1 را ۰ در نظر گرفتیم، این فرآیند ۰ واحد زمانی منتظر بوده و فرآیند P_2، ۲۴ واحد زمانی منتظر بوده و فرآیند P_3، ۲۷ واحد زمانی منتظر بوده

بنابراین: $AWT = \frac{0+24+27}{3} = 17$

Gantt chart

سوال) اگر ترتیب ورود فرآیندها بصورت P_2 و P_3 و P_1 باشد خواهیم داشت:

$AWT = \frac{6+0+3}{3} = 3$ ← کمترین AWT برای این سه فرآیند

همان طور که مشخص است کاهش قابل توجهی رخ داد بنابراین AWT تحت FcFs عموما minimal نیست و درصورتیکه burst time فرآیندها بسیار متفاوت باشد، AWT می‌تواند بسیار متفاوت باشد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

سوال) ATT در شال اول چه می شود؟ $ATT = AST + AWT$ ، پس باید AST را محاسبه کنیم: $AST = \frac{24+3+3}{3} = 10$

$\Rightarrow ATT = 10 + 17 = 27$

- اگر بخواهیم ATT را بصورت مستقیم حساب کنیم: $ATT = \frac{(24-0)+(27-0)+(30-0)}{3} = 27$

توجه) در سوالات پیچیده برای محاسبه AWT و ATT از فرمول $WT = TT - ST$ سوال را حل کنید چون همیشه به همین سادگی نیست و ممکن است در یک الگوریتم یک فرآیند بارها و بارها cpu را بگیرد و رها کند در اینصورت محاسبه AWT و ATT بصورت مستقیم کار زمان بر و پیچیده ای است.

توجه) توی شال دوم که AWT، ۱۴ واحد نسبت به شال اول کاهش پیدا کرد می توانیم بگیم ATT اش هم ۱۴ واحد کاهش پیدا می کند به فرمول $ATT = AST + AWT$ توجه کنید، برعکس این هم درسته، یعنی اگر ATT، x مقدار کم بشه AWT هم x واحد کم میشه

سوال) با استفاده از الگوریتم LcFs (Last-come first serve) AWT چند می شود؟ اول P_3 اجرا می شود که ۵ واحد منتظر می شود و بعد P_2 اجرا می شود که ۳ واحد منتظر می ماند و بعد هم P_1 اجرا می شود که ۶ واحد منتظر است.

نکته) هیچ کدام از ۲ الگوریتم FcFs و LcFs کمکی به کمینه کردن AWT نمی کنند.

convoy effect: این حالت که تمامی فرآیندها منتظر بمانند تا بزرگترین فرآیند cpu را رها بکند، این پدیده نا مطلوب است و ما دوست نداریم الگوریتم های ما convoy effect داشته باشند

در شال اول تحت FcFs فرآیندها برای شروع به اجرا منتظر خاتمه طولانی ترین فرآیند یعنی P_1 شدند

نکته) FcFs الگوریتم مناسبی برای Time sharing نیست چون پردازنده را بصورت انحصاری به یک فرآیند می دهد و تا یک فرآیند خاتمه پیدا نکند نوبت را به فرآیند دیگری نمی دهد و بنابراین زمان را بین فرآیندها به اشتراک نمی گذارد.

نکته) الگوریتم FcFs جز الگوریتم های priority Base (مبتنی بر اولویت) است، چون دارد براساس رسیدن فرآیندها (job، task) به آنها اولویت میدهد.

نکته) توجه کنید که در الگوریتم FcFs هیچ وقت starvation رخ نمی دهد چون اولویت در اختیار گرفتن cpu به زمان رسیدن کارها ربط دارد و به هر حال نوبت به همه کارها می رسد، چون کارهای جلوتر از این کار یا تموم می شوند و سیستم را ترک می کنند یا مقداری از آنها اجرا میشه و با یک سیستم کال به انتهای صف ready می رود.

نکته) از مزایای الگوریتم FcFs می توان به ساده بودن آن و قابلیت پیاده سازی راحت آن اشاره کرد، همین طور چون انحصاری است cs اش کمه و بنابراین utilization اش بالاست.

نکته) TT فرآیندهای I/O bound بالاست، چون هر بار که به واسطه سیستم کال بلاک می شوند به انتهای صف ready می روند.

نکته) شاید بتوان گفت یکی از مزیت های الگوریتم FcFs عدالت است و به این معناست که بین کسانی که در صف وجود دارند فقط براساس زمان رسیدنشان سرویس میدهیم و هر وقت نوبت هرکس بشود سرویس می گیرد

نکته) اگر ۲ کار دقیقا در یک زمان برسند می توانیم از یک Tie-breaker (گره گشا) که یک الگوریتم ثانویه هست استفاده کنیم

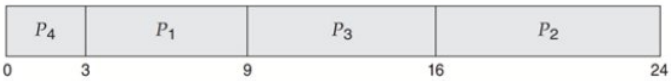
۲- SJF (Shortest-Job-First): هم با اینو بگین: اولویت به فرآیندی داده می شود که زمان اجرایش از سایر فرآیندها کوتاهتر است

درست: وقتی cpu در دسترس است، cpu به فرآیندی داده می شود که کوچکترین Next cpu burst بعدی را دارد، بنابراین بهتر بود اسم الگوریتم رو میذاشتن shortest-Next-cpu burst

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



مثال) با توجه به SJF اولویت در ابتدا با فرآیند P_4 بعد P_1، بعد P_3 و بعد P_2 است.

$$AWT = \frac{3+16+9+0}{4} = 7$$

نکته) در الگوریتم‌های SJF نوع non preemptive، به شرط آنکه همه کارها با هم وارد شوند AWT و ATT را minimize می‌کند، چون اگر هر تغییری روی ترتیب اجرای فرآیندهایی که SJF میدهد بدهیم AWT افزایش پیدا می‌کند آوردن فرآیندهای کوتاه قبل از فرآیندهای طولانی باعث بیشتر زمان انتظار فرآیندهای کوتاه بیشتر از افزایش زمان انتظار فرآیندهای طولانی کاهش پیدا کند.

چرا برای نکته بالا شرط همه کارها با هم وارد شوند را گذاشتیم؟ چون اگر وسط اجرای یک کار بزرگ یک کار کوچکی بیاید الگوریتم nonpreemptive SJF اجرای آن کار بزرگ را رها نمی‌کند و آن کار کوچکتر را انجام دهد بنابراین اگر بخواهیم SJF حتی اگر کارها با هم وارد نشوند همیشه AWT، ATT را minimize کند باید SJF را preemptive کنیم.

نکته) الگوریتم SJF یک الگوریتم priority based است و اولویت را به کوتاه بودن cpu burst‌های فرآیندها می‌دهد.

سوال) مگه Next Burst time فرآیندها را اصلا داریم که بتوانیم الگوریتم SJF را اجرا کنیم؟ محاسبه Next cpu burst یک فرآیند کار ساده‌ای نیست و بنابراین پیاده سازی SJF سخت است.

Exponential Average

الگوریتم بسیار ابتدایی است و الگوریتم قابل اعتمادی برای تخمین cpu burst نیست. این روش بر اساس گذشته تخمین میزند

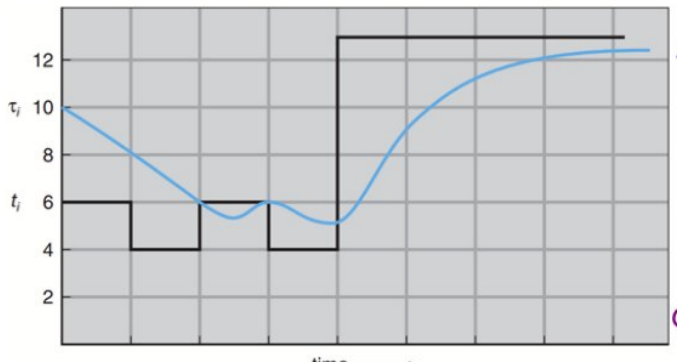
داستان چیه؟ فرض کنید زمان cpu burst‌های یک فرآیند خاص مثل A در 5 بار اجرای قبلی خودش بصورت زیر بوده:

t_0, t_1, t_2, t_3, t_4

حالا می‌خواهیم زمان cpu burst بعدی‌اش τ_5 را حدس بزنیم، حالا یک راه این هست که میانگین ساده بگیریم که اصلا خوب نیست، کاری که می‌کنیم این هست که میایم و میانگین وزن دار می‌گیریم و به cpu burst‌های اخیر وزن بیشتری میدهیم

حالا اینجوری هست که یک پارامتر α که بین 0 و 1 است در نظر می‌گیریم که این α به 2 مقدار طول واقعی n امین cpu burst (وقتی n-1 امین I/O burst را پشت سر نداریم به n امین cpu burst می‌رسیم) و مقدار تخمینی که ما از n امین cpu burst قبلا بدست آورده بودیم و با یک جمع وزن دار تخمین n+1 امین cpu burst را محاسبه می‌کند.

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$



*1 حدس اولیه برای cpu burst، 10 بوده ولی در واقعیت مقدار 6 داشته

$\tau_1 = \frac{1}{2}\times 6 + \frac{1}{2}\times 10 = \frac{1}{2}\times 6 + \frac{1}{2}\times 10 = 3+5 = 8$

*2 حدس دوم را بدست آورید:

$\tau_2 = \frac{1}{2}\times t_1 + \frac{1}{2}\tau_1 = \frac{1}{2}\times 4 + \frac{1}{2}\times 8 = 2+4 = 6$

* اگر فرمول $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$ را با جایگذاری پایین به بالا یا بالا به پایین باز کنیم به فرمول زیر می‌رسیم:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

که مشخص است که تخمین n+1 امین cpu burst به زمان واقعی burst‌های

	t_0	t_1	t_2						
CPU burst (t_i)	*1 → 6	*2 → 4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...
		τ_1	τ_2						

$\alpha = 0.5$ Prediction of the length of the next CPU burst.

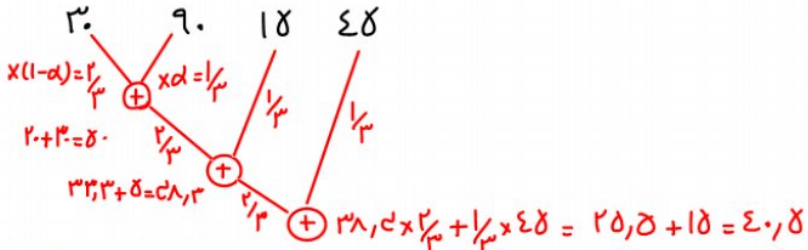
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

اول تا n ام و همین طور تخمین cpu burst صفر وابسته است و هرچه به سمت زمان های واقعی قبلتر می رویم اهمیت شون در تخمین ما کمتر می شود

سوال) زمان cpu burst های قبلی یک فرآیند به ترتیب از t_0 تا t_3 برابر ۳۰، ۹۰، ۱۸ و ۴۸ است با $\alpha=\frac{1}{3}$ و با exponential avg زمان کار جدی را تخمین بزنید.

$\alpha=\frac{1}{3}$ ، $1-\alpha=\frac{2}{3}$



نکته) الگوریتم SJF، starvation دارد چون ممکن است مثلاً یک کار طولانی وجود داشته باشد که همیشه کارهای جدید کوچکتر وجود داشته باشد که بیاد جلوی این کار قرار بگیرد.

نکته) الگوریتم SJF می تواند preemptive یا nonpreemptive باشد.

وقتی یک فرآیند جدید وارد صف ready می شود ممکن است Next cpu burst این فرآیند تازه وارد کوچکتر از زمان اجرای باقیمانده فرآیند در حال اجرا روی cpu باشد، حالا یک الگوریتم preemptive SJF فرآیند در حال اجرا روی cpu را preempt می کند، درحالیکه یک الگوریتم nonpreemptive SJF به فرآیند در حال اجرا اجازه میدهد تا cpu burst اش را به پایان برساند، گاهی اوقات به preemptive SJF: shortest_remaining_time_first (SRTF) نیز می گویند (SRT هم بخش نگین)

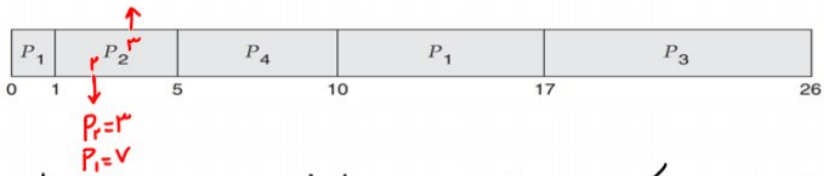
SRTF : برای تخصیص cpu فرآیندی انتخاب شود که کمترین زمان باقیمانده را دارد

وقتی کار جدیدی وارد می شود زمان cpu burst اش را با زمان باقی مانده از cpu burst فرآیند جاری مقایسه می کند و اگر زمان کار جدید کمتر بود فرآیند جدید جایگزین می شود (در صورت مساوی بودن الگی سربار switch نمی دهد)

مثال) درلحظه ۰ فقط فرآیند P_1 در سیستم حضور دارد پس انتخاب می شود، حالا در لحظه ۱ فرآیند P_2 با cpu burst=۴ از راه می رسد و چون در این لحظه زمان باقیمانده از فرآیند ۱ برابر ۷ است، پس پردازنده از P_1 گرفته می شود و به P_2 داده می شود. درلحظه ۲ فرآیند P_3 وارد می شود، در این لحظه زمان باقیمانده فرآیند جاری P_2 سه تا است و زمان P_3 نه تا است، پس کمتر از P_2 نیست و P_2 به کارش ادامه میدهد، درلحظه ۳ فرآیند P_4 وارد می شود که زمان اجرایش ۵ است که بیشتر از $P_2=2$ است، پس P_2 به کارش ادامه میدهد، پس از اتمام P_2 چون همه فرآیندها در سیستم وارد شدند به ترتیب زمان های باقیمانده شون اجرا می شوند

Process	Arrival Time	Burst Time
P_1	0	8 → 7
P_2	1	4
P_3	2	9
P_4	3	5

$P_2=2$



سوال) AWT همه فرآیندها را بدست آورید:

$WT=TT-ST=(17-0)-8=9$ — از ۱ تا ۱۰ برای اجرا شدن قسمت دوم اش انتظار کشیده $WT(P_1)=10-1=9$

$WT=(5-1)-4=0$ — در ۱ رسیده و در ۱ هم اجرا شده $WT(P_2)=1-1=0$

$WT=(26-2)-9=15$ — " ۲ " " " ۱۷ " " " $WT(P_3)=17-2=15$

$WT=(10-3)-5=2$ — " " " " ۵ " " " $WT(P_4)=5-3=2$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

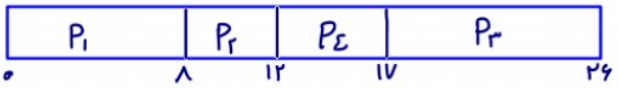
$AWT = \frac{9+0+15+2}{4} = 6,5$

⭐ الگوریتم‌های sjF preemptiv اگر کارها با هم نیز وارد نشوند AWT, ATT را minimize می‌کند.

سوال) روی جدول با ۴ الگوریتم nonpreemptive SJF اجرا کنید.

هیچ کاری اگر شروع به اجرا کار تا تمام نشود cpu را رها نمی‌کند، اگر انجام بدید متوجه می‌شید که AWT=۷,۷۵ می‌شود

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



$AWT(Direct) = \frac{0+(8-1)+(17-2)+(12-3)}{4} = 7,75$

$AWT = ATT - AST \Rightarrow AWT = 14,25 - 6,5 = 7,75$

$TT(P_1) = 8-0 = 8$

$TT(P_2) = 12-1 = 11$

$TT(P_3) = 17-2 = 15$

$TT(P_4) = 26-3 = 23$

$\Rightarrow ATT = \frac{8+11+15+23}{4} = 14,25$

$AST = \frac{26}{4} = 6,5$

تمرین) با توجه به جدول زیر Gantt chart را برای preemptive sjF, nonpreemptive sjF, FcFs رسم کنید و سپس AWT هر سه الگوریتم را بدست آورید.

process	Arrival time	cpu burst
A	۰	۷
B	۰	۸
C	۲	۵
D	۴	۵
E	۶	۲

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

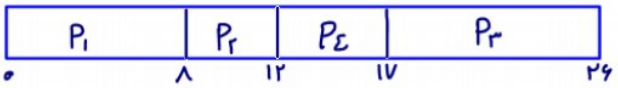
$AWT = \frac{9+0+15+2}{4} = 6,5$

⭐ الگوریتم‌های preemptiv SJF اگر کارها با هم نیز وارد نشوند AWT, ATT را minimize می‌کند.

سوال) روی جدول با الگوریتم nonpreemptive SJF اجرا کنید.

هیچ کاری اگر شروع به اجرا کار تا تمام نشود cpu را رها نمی‌کند، اگر انجام بدید متوجه می‌شید که AWT=7,75 می‌شود

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



$AWT(Direct) = \frac{0+(8-1)+(17-2)+(12-3)}{4} = 7,75$

$AWT = ATT - AST \Rightarrow AWT = 14,25 - 6,5 = 7,75$

$TT(P_1) = 8-0 = 8$

$TT(P_2) = 12-1 = 11$

$TT(P_3) = 17-2 = 15$

$TT(P_4) = 26-3 = 23$

$\Rightarrow ATT = \frac{8+11+15+23}{4} = 14,25$

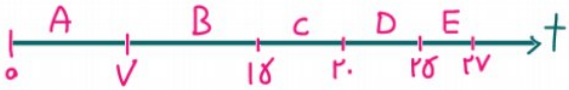
$AST = \frac{26}{4} = 6,5$

تمرین) باتوجه به جدول زیر Gantt chart را برای FcFs, nonpreemptive SJF, preemptive SJF رسم کنید و سپس AWT هر سه الگوریتم را بدست آورید.

process	Arrival time	cpu burst
A	10	7
B	0	8
C	2	5
D	4	5
E	6	2

$AST = \frac{7+8+5+5+2}{5} = \frac{27}{5} = 5,4$

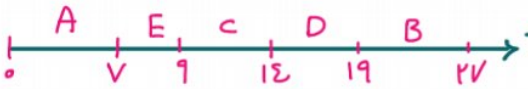
FcFs:



$ATT = \frac{(7-0)+(18-0)+(20-2)+(25-4)+(27-6)}{5} = \frac{82}{5} = 16,4$

$AWT = ATT - AST = 16,4 - 5,4 = 11$

nonpreemptive SJF:



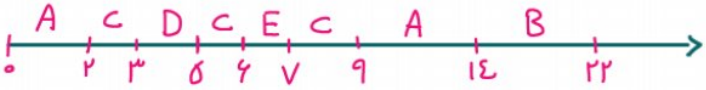
$ATT = \frac{(7+9+14+19+27)-(0+0+2+4+6)}{5} = 12,8$

$AWT = 12,8 - 5,4 = 7,4$

preemptive SJF = SRTF:

process	Arrival time	cpu burst
A	0	7→5
B	0	8
C	2	4→3→2
D	3	2
E	6	1

SRTF را روی نمودار مقابل انجام دهید:



$ATT = \frac{(22+14+9+7+5)-(2+3+6)}{5} = 9,2$

$AST = \frac{22}{5} = 4,4$

$AWT = 9,2 - 4,4 = 4,8$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Multi tasking = Time sharing

time slice = quanta = quantum

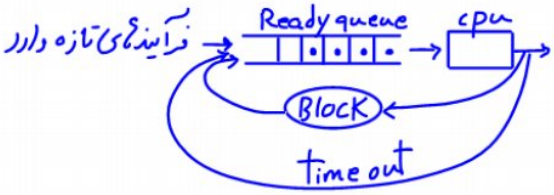
نه تنها کار Multi programming را انجام بدهیم بلکه حتی اگر برنامه ای نرفت دنبال I/O ولی سهمیه زمانی اش تمام شد در آن شرایط هم cpu را به پراسس بعدی بدهیم بطوریکه همه برنامه ها (پراسس ها) همزمان باهم پیشرفت کنند وجلو بروند.

Round Robin = RR

RR معروف ترین الگوریتمی است که برای Time sharing در سیستم های Multitasking استفاده می شود.

دانستنی: الگوریتم RR در بسیاری از OS ها نظیر ویندوز و لینوکس پیاده سازی شده

این الگوریتم شبیه الگوریتم FcFs است با این تفاوت که به آن preemption اضافه شده تا سیستم را قادر به switch بین فرآیندها در زمانهای مشخص بکند. این الگوریتم بصورت preemptive عمل می کند و به اینصورت است که یک واحد کوچکی از زمان به اسم time-quantum = time-slice = time_slot در نظر گرفته می شود که معمولاً بین ۱۰ تا ۱۰۰ میلی ثانیه است که تمام شدن این سهمیه زمانی با آمدن T.O تایمر مشخص می شود، حال به اندازه این کوانتوم به هر فرآیند داده می شود. برای پیاده سازی RR، صف ready را بصورت یک صف FIFO در نظر می گیریم (شبیه FcFs) و فرآیندهای جدید به انتهای (tail) صف اضافه می شوند، cpu scheduler اولین فرآیند را از سر صف ready بر میدارد و یک تایمر به اندازه یک کوانتوم تنظیم می کند، حالا فرآیندی که cpu در اختیارش هست سه تا اتفاق میفتد براش بیفتد که در هر سه حالت cpu scheduler فعال میشه و context switch اتفاق میفتد وکسی که سر صف است میره روی cpu



۱- قبل از اتمام کوانتوم کارش تموم بشه و exit کند

۲- کوانتوم اش تمام بشود که در اینصورت به انتهای صف ready منتقل میشه

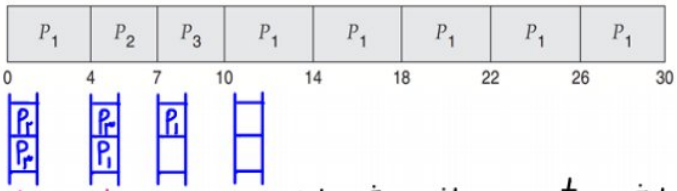
۳- فرآیند در وسط کوانتوم اش سیستم کال بزند که بلاک می شود و بعد از انجام شدن کارش به انتهای صف اضافه می شود.

به اینصورت تمامی فرآیندها دارند زمان سیستم را بین هم به اشتراک می گذارند و همه بصورت همزمان باهم پیشرفت می کنند.

توجه) در RR به هیچ فرآیندی دو کوانتوم پشت سر هم اختصاص داده نمی شود مگر اینکه آن فرآیند تنها فرآیند قابل اجرا باشد.

مثال) فرض کنید کوانتوم ۴ باشد و همه در زمان ۰ وارد شوند.

Process	Burst Time
P_1	24
P_2	3
P_3	3



وضعیت صف ready

$AWT(direct) = \frac{(10-4)+(4-0)+(7-0)}{3} = \frac{17}{3} = 5,66$

$AWT(indirect) = 15,66 - 10 = 5,66$

$ATT = \frac{30+7+10}{3} = 15,66$ $AST = \frac{24+3+3}{3} = 10$

نکته) در RR چون بعد از هر کوانتوم اگر کار فرآیند تمام نشده باشد preemption انجام می شود این باعث می شود که زمان پاسخ برای کارهای کوچک کم بشود و در نتیجه RR مناسب کارهای تعاملی است.

نکته) الگوریتم RR عدالت دارد زیرا زمان کوانتوم برای همه یکسان است واگر فرآیندی K کوانتوم نیاز داشته باشد باید پس از هر بار گرفتن کوانتوم برای گرفتن کوانتوم بعدی اش باید مجدداً به انتهای صف برود و بنابراین مشخص است که فاقد starvation نیز هست.

نکته) در این الگوریتم چون نیاز به تخمین cpu burst فرآیندها وجود ندارد پیاده سازی اش ساده است.

نکته) هرچقدر کوانتوم را کوچکتر بگیریم تعداد cs ها افزایش می یابد و بهره وری پردازنده کمتر می شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

quantum

به تناسب اینکه سایز کوانتوم را چقدر بگیریم الگوریتم زمانبندی RR رفتارهای مختلفی را از خود نشان می دهد:

۱- اگر q (کوانتوم زمانی) به سمت یک عدد خیلی بزرگ میل کند ($q \to \infty$) الگوریتم RR به الگوریتم FcFs تبدیل (degenerate) می‌شود

۲- " " " " " " " کوچک " " " " ($q \to \varepsilon$) " " processor sharing (ps) تبدیل (degenerate) می‌شود.

processor sharing: با فرض اینکه سربار زمانی cs صفر باشد که در واقعیت اینگونه نیست

فرض کنید ۲ فرآیند P_1 و P_2 قرار است روی پردازنده اجرا شوند، حال ۳ حالت زیر را در نظر بگیرید: (می خواهیم q را زیر زیر کم کنیم)

۱- اگر q را ۱ در نظر بگیریم:

- حال اگر بعد از ۲ ثانیه به سیستم نگاه کنیم، می بینیم که ۱ ثانیه به P_1 و ۱ ثانیه به P_2 رسیده و این یعنی در ۲ ثانیه پراسسور بصورت مساوی بین این دو share شده
- حال اگر بعد از ۱ ثانیه به سیستم نگاه کنیم می بینیم که P_1 ثانیه از پراسسور گرفته ولی P_2 هیچ سهمی از پراسسور نگرفته است و این یعنی در ثانیه ۱ ، ۱۰۰٪ پراسسور را P_1 و ۰٪ را P_2 گرفته.

۲- اگر q را ۰٫۵ در نظر بگیریم

- حالا اگر در ثانیه ۱ به سیستم نگاه کنیم متوجه می شیم که پراسسور بین P_1 و P_2 ، share شده است.

۳- حال اگر روند کاهش q را ادامه دهیم و q را به سمت ۰ میل بدهیم چه می شود؟

در اینصورت در هر لحظه که به سیستم نگاه کنیم مشاهده می‌کنیم که هر یک از فرآیندها سهم خودش را از پراسسور گرفته که در این حالت می‌گیم ps داریم.

★ پس بصورت تئوری اگر کوانتوم به سمت ۰ میل کند پراسسور بین فرآیندها share می‌شود اما در واقعیت ما نمی توانیم اندازه کوانتوم را از یک حدی کمتر کنیم. چرا؟

می دانیم که cs برای ما سربار (overhead) به همراه دارد، حال اگر $q < cs$ overhead باشد، بیشتر از اینکه زمان پراسسور صرف پردازش فرآیندها بشود دارد صرف cs بین فرآیندها می‌شود!

نکته) سربار cs باید بین $\frac{1}{1000}$ تا $\frac{1}{10000}$ زمان کل پراسسور باشد، یعنی utilization پراسسور باید $U = 1 - (\frac{1}{1000} \text{ or } \frac{1}{10000})$ باشد.

★★ سوال) اگر در سیستمی بخواهیم ۱/۱۰ سربار cs باشد، با توجه به فرضیات زیر چه time slice ای برای این سیستم مناسب است؟

فرکانس پردازنده = 1GHz

زمان دسترسی به حافظه = 10ns (memory Access time) : فرض کنید خواندن (Read) و نوشتن (write) به یک میزان زمان می برد

فرض کنید که pcb برابر 4KB است که ۸۰۰ بایت آن نوشته و بازیابی می‌شود

حل: باید time slice را با توجه به فرمول زیر بدست آوریم: $\frac{1}{1000}$

$cs\ time < time\ slice \times 0.1\% \Rightarrow time\ slice > 1000 \times cs\ time$

$cs\ time = 1000 \times 10 \times 10^{-9} s$

توجه کنید که هنگام cs، ۸۰۰ بایت برای فرآیند اول باید ذخیره (write) و ۸۰۰ بایت برای فرآیند دوم باید از حافظه بازیابی شود (Read)

$\Rightarrow time\ slice > 1000 \times 10^{-8} s \Rightarrow time\ slice > 10^{-2} s \Rightarrow time\ slice > 10^{-2} \times \frac{10}{10} s \Rightarrow time\ slice > 10\ ms$

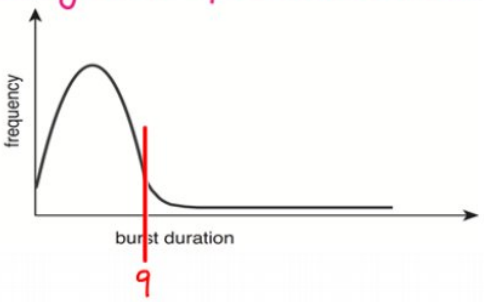
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

80% of cpu burst < time quantum : rule of thumb

این قانون تجربی می‌گوید q را به اینصورت تنظیم کنید که زمان ۸۰ درصد cpu burst ها از q کوچکتر باشد. چرا؟

Histogram of cpu-burst duration



اگر q را طوری بگیریم که cpu burst اکثر فرآیندها کمی بیشتر از q باشد اونوقت اکثر فرآیندها قبل از اینکه cpu burst شون تموم بشه، q شون تموم میشه و cs اتفاق میفته و دفعه بعدی که نوبت به این فرآیندها برسه چون یکی از cpu burst شون باقی مونده، یکی روی cpu اجرا می‌شوند و burst شون تموم میشه و cpu رو رها می‌کنند و این باعث میشه که ما حدوداً در هر q به جای یک cs، دو تا cs داشته باشیم که این امر باعث میشه سربار cs به جای x٪ حدود ۲x٪ بشه

بهره وری cpu را می‌تونیم بصورت زیر هم درنظر بگیریم:

ما بصورت پریودیک یک q داریم و بعد یک cs : q cs q cs ...

$$U = \frac{q}{q+cs} \times 100\%$$

حالا اگر همه فرآیندها تمامی کوانتوم هایشان را مصرف کنند، پس در هر پریود از جمع q+cs، q اش مفید است و ما حداکثر بهره وری را برای cpu خواهیم داشت:

(توجه داشته باشید که این حالت فقط در صورتی رخ میدهد که تمامی cpu burst های تمامی فرآیندها مضرب صحیحی از q باشد)

همان طور که از روی فرمول نیز مشخص است اگر میزان q کوچکتر و نزدیکتر به زمان cs شود، سربار cs بیشتر خودشو نشون میده.

اگر میانگین burst time تمامی فرآیندها کمتر از q باشند، بهره وری cpu تقریبا برابر است با:

$$U = \frac{\text{Average burst time}}{\text{Average burst time} + cs}$$

نکته) هدف RR و اینکه پس از هر کوانتوم cs می‌کنیم این است که پاسخ کارهای کوچک داده شود و respons time شون کم بشه پس اندازه کوانتوم را نباید کوچکتر از اندازه کارهای مرسوم کوچک موجود در سیستم درنظر بگیریم، همین طور برای افزایش بهره وری نمی‌توانیم اندازه کوانتوم را هرچقدر دوست داریم ببریم بالا چون اگر اندازه کوانتوم را خیلی بزرگ بگیریم response time کارهای مرسوم کوچک بالا می‌رود.

نکته) شاید بتوان بصورت حدودی و میانگین گفت که ATT و AWT الگوریتم RR در جایگاه زیر قرار می‌گیرد:

البته وقتی سوال میدن باید حل کنیم

SRTF < non preemptive SJF < FCFS < RR

پس اگرچه TT و WT برای کارهای کوچک پایین است ولی ATT و AWT در RR بالاست

نکته) TT فرآیندهای I/O bound بالاست، چون هر بار که به واسطه سیستم کال بلاک می‌شوند به انتهای صف ready می‌روند.

* اگر n فرآیند در صف ready وجود و کوانتوم زمانی q باشد، درحالتی که هر فرآیند تمام q خودش را مصرف کند (worst case)

- هر فرآیند حدوداً $\frac{1}{n}$ از زمان cpu را در chunk های حداکثر q واحد زمانی دریافت می‌کند
- حداکثر زمان انتظار هر فرآیند برای دریافت کوانتوم بعدی ای که می‌گیرد $(n-1)q$ است.
- 〃 〃 〃 یک 〃 که برای اجرای کامل اش به x تا q نیاز دارد: $x \times (n-1)q$
- حداکثر TT یک فرآیند درون صف که با گرفتن یک q کارش تمام می‌شود: $(n-1)q + q = nq$
- 〃 〃 〃 〃 〃 〃 〃 〃 x تا q 〃 〃 〃 : $x \times nq$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

حل تست‌های RR

سوالات RR بسته به تناسب بین زمان اجرا فرآیندها و میزان زمان q را می‌توان به ۲ صورت حل کرد.

۱- اگر زمان اجرا فرآیندها مضرب معقولی از q باشد که بتوانیم نمودار گانت را رسم کنیم.

۲- " " " " " بزرگ " " " " " نتوانیم " " استاندارد را رسم کنیم: processor sharing

سوال) ATT را در RR با q=۱ در ۳ حالت زیر بدست آورید

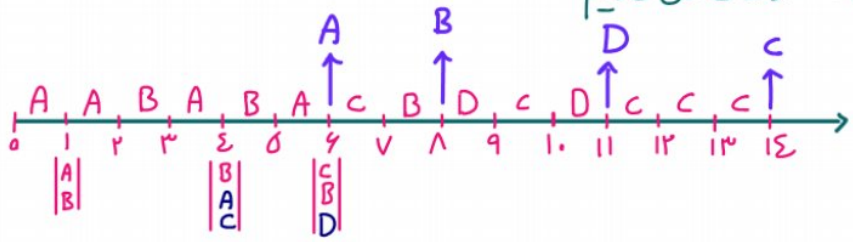
process	Arrival time	cpu burst
A	۰	۴
B	۱	۳
C	۴	۵
D	۶	۲

الف) بین فرآیندی که در لحظه t جدید وارد می‌شود و فرآیندی که در همان لحظه از cpu خارج شده اولویت با فرآیند قبلی درون سیستم باشد.

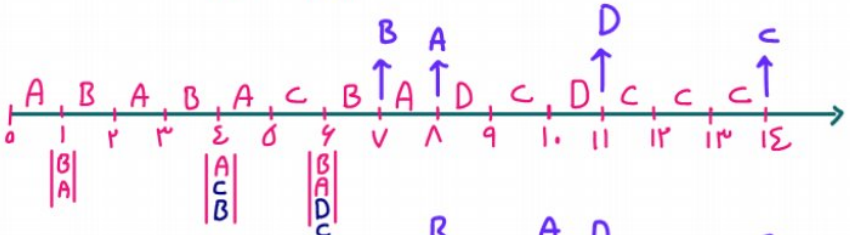
ب) در حالت الف اولویت با فرآیند تازه وارد باشد

ج) فرآیندی که جدید وارد می‌شود در همان ابتدای ورودش اجرایش آغاز شود

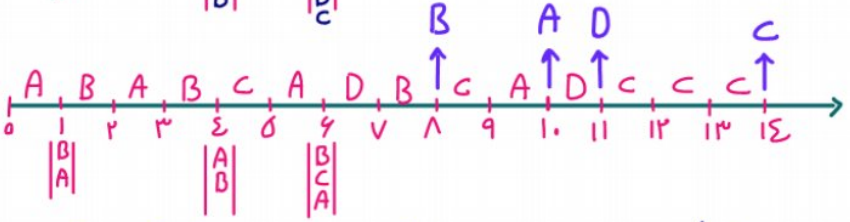
توجه) چون زمان اجرا فرآیندها مضرب معقولی از q است از روش اول می‌ردیم.



$$ATT = \frac{(14+11+8+6)-(0+1+4+6)}{4} = 7$$



$$ATT = \frac{(14+11+8+7)-11}{4} = 7,25$$



$$ATT = \frac{(14+11+10+8)-11}{4} = 8$$

تست سراسری ۷۷) جدول زیر اطلاعات ۵ کار را در یک سیستم اشتراک زمانی نشان میدهد، اگر برای مدیریت فرآیندها از روش RR با برش زمانی ۱ میلی ثانیه استفاده شود میانگین زمان پاسخ چقدر است؟

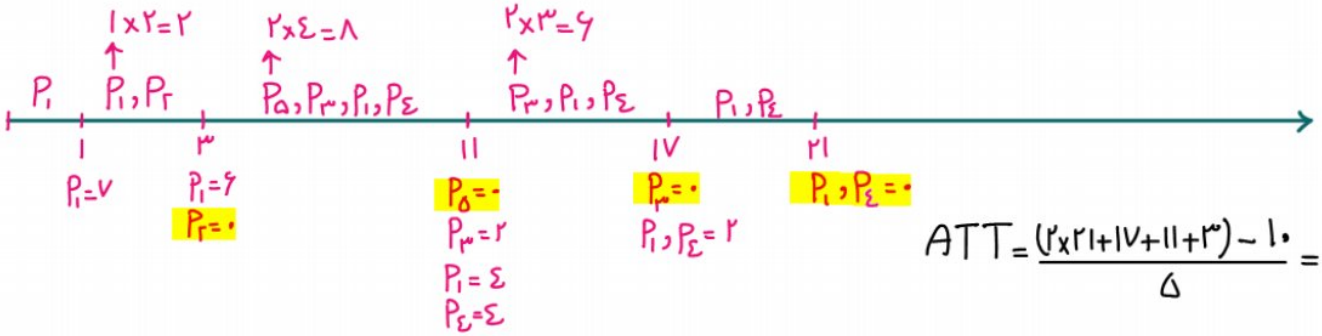
	زمان اجرا به دقیقه	زمان ورود به سیستم به دقیقه
P_1	۸	۰
P_2	۱	۱
P_3	۴	۳
P_4	۶	۳
P_5	۲	۳

۱) ۸,۴

۲) ۱۶,۸

۳) ۱۲,۶

۴) ۲۱



$$ATT = \frac{(2\times21+17+11+3)-10}{5} = \frac{63}{5} = 12,6$$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Highest Response Ratio Next : HRRN : بالاترین نسبت پاسخ

یک الگوریتم non preemptive است که همین طور که از اسم اش مشخصه که هر بار که cpu خالی میشه اون کاری رو انتخاب میکند که R بیشتری دارد.
چون انحصاری است cs اش کم است.

$R = \frac{w+s}{s} = \frac{w}{s} + 1$

R = response ratio

w = wating time : تا لحظه ای که زمانبند می خواهد کار بعدی را انتخاب کند

s = expected service time = cpu burst time

در زمانبند غیرقبضه‌ای «بعدی – بیشترین – نسبت – زمان پاسخ» (HRRN) پردازه‌ای برای اجرا انتخاب می‌شود که بیشترین نسبت Ratio=$\frac{waiting\ time}{CPU\ Brust\ Time}$ را داشته باشد. چنانچه ۴ پردازه به صورت جدول زیر داشته باشیم، در مقایسه با زمانبند غیرقبضه‌ای «اول – کمترین – زمان» (Shortest Job First) کدام مورد درخصوص متوسط زمان پاسخ درست است؟

(فناوری اطلاعات ۱۳۹۸)

پردازه	زمان ورود	زمان مورد استفاده از CPU (CPU Burst Time)
p_1	۰	۸
p_2	۱	۹
p_3	۲	۵
p_4	۳	۴

۱) متوسط زمان انتظار روش HRRN برابر روش SJF است.

۲) متوسط زمان انتظار روش HRRN کمتر از روش SJF است.

۳) متوسط زمان انتظار روش HRRN بیشتر از روش SJF است.

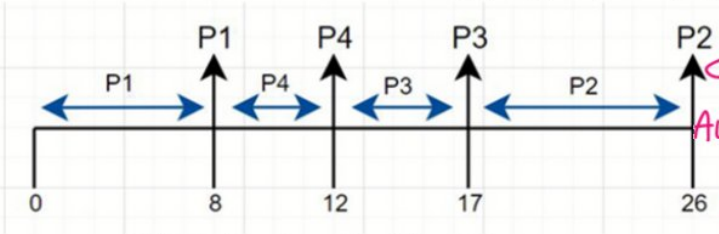
۴) متوسط زمان انتظار روش HRRN قابل محاسبه نیست.

نکته) برای محاسبه R نیازی نیست $\frac{w}{s}+1$ را بدست آورید و بهتر است $\frac{w}{s}$ را بدست آورید.

توجه) هر بار که یک کاری تمام می شود و ما می خواهیم کار بعدی را برای ما Dispatch انتخاب کنیم باید مجددا محاسبات را انجام دهیم.

پاسخ گزینه ۱ است.

نمودار گانت این ۴ فرایند برای الگوریتم SJF به صورت زیر است:

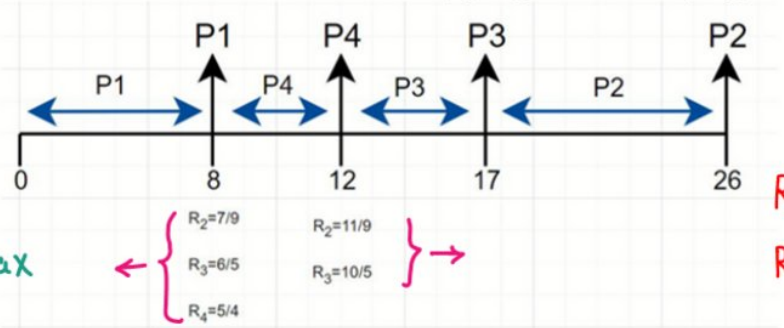


نکته) چون نمودار گانت دو تا شون عین هم شده نیازی به محاسبه AWT وجود نداره و مشخصه که مثل همه

$AWT = \frac{\sum_1^n (\text{زمان پردازش} - \text{زمان ورود} - \text{زمان خروج})}{n}$ = میانگین زمان انتظار فرایندها

$AWT = \frac{(8-0-8)+(26-1-9)+(17-2-5)+(12-3-4)}{4} = 7.75$

نمودار گانت این ۴ فرایند برای الگوریتم HRRN به صورت زیر است:



$R = \frac{w}{s}$

$R_2 = \frac{8-1}{9} = \frac{7}{9} < 1$

$R_3 = \frac{8-2}{5} = \frac{6}{5} = 1 + \frac{1}{5}$

$R_4 = \frac{8-3}{4} = \frac{5}{4} = 1 + \frac{1}{4}$: max

$R_2 = \frac{12-1}{9} = \frac{11}{9} = 1 + \frac{2}{9}$

$R_3 = \frac{12-2}{5} = \frac{10}{5} = 2$: max

$R_2=7/9$ $R_3=6/5$ $R_4=5/4$ ← { → { $R_2=11/9$ $R_3=10/5$

$AWT = \frac{\sum_1^n (\text{زمان پردازش} - \text{زمان ورود} - \text{زمان خروج})}{n}$ = میانگین زمان انتظار فرایندها

$AWT = \frac{(8-0-8)+(26-1-9)+(17-2-5)+(12-3-4)}{4} = 7.75$

همانطور که مشخص است متوسط زمان انتظار این ۴ فرایند در هر دو الگوریتم یکسان است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) در این الگوریتم نیز ما نیاز به Next burst time فرآیندها داریم و بنابراین در این الگوریتم نیز ما نیاز به روشهایی برای تخمین این موضوع داریم.

- همان‌طور که دیدیم HRRN می‌گوید هرکه $\frac{w}{s}$ اش بیشتر باشد اولویت بالاتری دارد، حالا چرا w را با s ترکیب کرده؟
- فرض کنید همه s ها با هم مساوی باشد در این شرایط اولویت کی بالاتره؟ هرکسی که w اش بیشتر باشد.

در یک صف FIFO چه فرآیندی انتظار بیشتری می‌کشد؟ هر فرآیندی که در صف جلوتر باشد انتظار بیشتری کشیده. خب در این حالت عادل الگوریتم FcFs میشه

- فرض کنید همه w ها با هم مساوی باشند در این شرایط اولویت کی بالاتره؟ هرکسی که s اش کوچکتر باشد.

خب HRRN انحصاری عمل می‌کند پس در این شرایط چون داره شبیه nonpreemptive SJF عمل می‌کند

الگوریتم HRRN در صدد این است رفتاری بین FcFs، nonpreemptive SJF از خود نشان دهد و می‌خواهد نه فقط به زمان ورود و در نتیجه w توجه کند که در نتیجه دچار starvation نمی‌شدیم ولی AWT و ATT زیاد می‌شد توجه کند و نه فقط به s که باعث می‌شد AWT و ATT کم بشه ولی starvation داشته باشیم.

نکته) در این الگوریتم starvation رخ نمی‌دهد چون هرچه زمان انتظار یک فرآیند بیشتر شود اولویت اش بالاتر می‌رود و بالاخره اجرا می‌شود.

نکته) این الگوریتم چون به s ها و هنرمان توجه می‌کند هم به کارهای کوچک توجه می‌کند و بنابراین AWT، ATT بدی ندارد و هم به w ها توجه می‌کند و بنابراین قطعی ندارد و هم عدالت دارد چون اگر $\frac{w}{s}$ فرآیند P_1 با فرآیند P_2 برابر باشد ولی زمان P_1-burst، K برابر، P_2 باشد به این علت که w اش نیز K برابر P_2 بوده و الگوریتم HRRN به این ۲ فرآیند یک اولویت میدهد.

priority scheduling

الگوریتم‌های قبلی ای که بررسی کردیم بر مبنای اولویت عمل می‌کردند. معمولاً در OS ها تعدادی سطوح اولویت وجود دارد، در اکثر OS های رایج معمولاً ۲۵۶ سطح اولویت وجود دارد و هر فرآیند یک اولویتی دارد و معمولاً عدد کوچکتر دارای اولویت بیشتری است و دارای بیشترین اولویت و ۲۵۵ دارای کمترین اولویت است، البته OS هایی هم وجود دارند که بالعکس عمل می‌کنند.

نکته) ممکن است در مواقعی چندین فرآیند اولویت برابری داشته باشند که در این حالت نیاز به یک tie-breaker (گره‌گشا) داریم که یک الگوریتم ثانویه است که بین فرآیندهایی که اولویت یکسانی دارند برای آنکه مشخص کند که فرآیند بعدی که باید به آن cpu بدهد چه فرآیندی باشد از یک الگوریتم دیگر مثل FcFs استفاده می‌کند.

نکته) الگوریتم SJF یک priority Algorithm است که در آن اولویت یک فرآیند برابر معکوس Next cpu burst پیش بینی شده است و هرچه cpu burst یک فرآیند بیشتر باشد اولویت اش کمتر و هرچه کمتر باشد اولویت اش بیشتر است.

نکته) یک priority scheduling می‌تواند preemptive یا nonpreemptive باشد، این دو چگونه عمل می‌کنند؟

فرض کنید یک فرآیند وارد صف ready می‌شود (حالت اش ready می‌شود)، حال اولویت این فرآیند با اولویت فرآیند در حال اجرا مقایسه می‌شود و حالا فرض کنید که اولویت اش بیشتر باشد.

preemptive priority scheduling Algorithm

cpu را از فرآیند جاری می‌گیرد و به فرآیند با اولویت بالاتری که همین الان اومده میدهد.

nonpreemptive priority scheduling Algorithm

فرآیند جدید را در سر صف ready قرار میدهد و فرآیند جاری را preempt نمی‌کند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

مثال) فرض کنید عدد پایین تر نشان دهنده اولویت بالاتر باشد و Burst time & براساس میلی ثانیه باشد AWT را محاسبه کنید.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

$$AWT(Direct) = \frac{6+0+16+18+1}{5} = 8.2$$

نکته) شکل اصلی priority scheduling Algorithms دو شکل مسدودشدن نامحدود (indefinit blocking) یا starvationاست، فرآیندی که آماده اجرا است ولی منتظر cpu است را می توان Block در نظر گرفت. در یک سیستم کامپیوتری heavily loaded یک جریان ثابت از فرآیندهای با اولویت بالاتر، می توانند مانع از دریافت cpu توسط یک فرآیند با اولویت پایین شود، بطور کلی برای این چنین فرآیندی در سیستم مذکور ۲ اتفاق خواهد افتاد، یا وقتی سیستم لود خیلی کمی دارد بعنوان مثال ساعت ۲ صبح روز جمعه اجرا می شود و یا سیستم درنهایت crash می کند و تمامی فرآیندهای ناتمام با اولویت پایین از دست می روند.

★ starvation به جز مشکلاتی که می تواند برای کاربر فرآیندی که دچار گرسنگی شده است به وجود بیاورد می تواند مشکلاتی را برای منابع سیستم کامپیوتری به وجود بیاورد، بنابراین باید چاره ای برای آن بیندیشیم.

مثال از starvation: یک کامپیوتر بزرگ IBM 7094 مربوط به دانشگاه MIT بوده، عموما سیستم های کامپیوتری بزرگ هر چند سال یکبار و برای سرویس های دوره ای خاموش می شوند، وقتی سال 1973 برای سرویس دوره ای این سرویس را Down می کنند متوجه می شوند که یک فرآیند با اولویت پایین از سال 1967 در سیستم باقی مانده که به دلیل starvation موفق نشده اجرا بشود، در اینصورت پس از چند دقیقه احتمالا کسی دیگر منتظر اجرا این فرآیند نمانده ولی این فرآیند در صف ready باقیمانده و همواره در الگوریتم زمانبندی شرکت داده میشده و همواره منابعی دست اش بوده، درصورتیکه دیگه کسی منتظر اون فرآیند نبوده.

نکته) عموما الگوریتم های priority based دارای مشکل starvation هستند.

مثلاً الگوریتم FcFs یک الگوریتم priority based است ولی هیچ وقت دچار starvation نمی شود ولی الگوریتم SJF که اون هم یک الگوریتم priority based هست یک فرآیند می تواند دچار starvation شود و آن زمانی است که یک فرآیند cpu burst طولانی داشته باشد و همیشه قبل از اینکه نوبت به این فرآیند برسد فرآیندهایی با cpu burst های کوتاهتر از این فرآیند برسند و cpu scheduler، پردازنده را به آن فرآیندها اختصاص بدهد و در این صورت ممکن است اجرای آن فرآیند طولانی یا شروع نشود یا اگر شروع شده و SJF ما از نوع preemptive است اجرایش شروع بشود ولی تمام نشود.

★ مشکل starvation یک یا تعدادی فرآیند ممکن است بصورت کاملا شانسی از بین برود.

بعنوان مثال فرض کنید در SJF یک فرآیند دچار starvation شده حال ممکن است بصورت کاملا اتفاقی و شانسی یک مدتی طول هایی با cpu burst های کوتاهتر از فرآیند مذکور وارد صف ready شوند و زمان به فرآیند مذکور که cpu burst طولانی داشته و دچار starvation شده بوده برسد و روی cpu اجرا شود.

Indefinit Blocking: شرایطی که در آن تعدادی فرآیند بصورت indefinit درگیر Blocking می شوند و دیگر از آن حالت خارج نمی شوند.

راه حل: یکی از راه حل ها Aging است: راه حلی برای حل مشکل انسداد نامحدود فرآیندهای با اولویت های پایین (و مشکل starvation)

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

یک تایمر ست کنیم که بصورت پریودیک وقفه بدهد و با هر بار وقفه ای که می دهد برویم و اولویت فرآیندها را شلاً یک لول اتریبش بدهیم و بدین ترتیب حتی کم اولویت ترین فرآیندها پس از یک مدت زمانیکه میگذره اولویت شون به بالاترین سطح می رسه و اجرا خواهند شد.

Multilevel queue scheduling : زمانبند صف چند سطحی

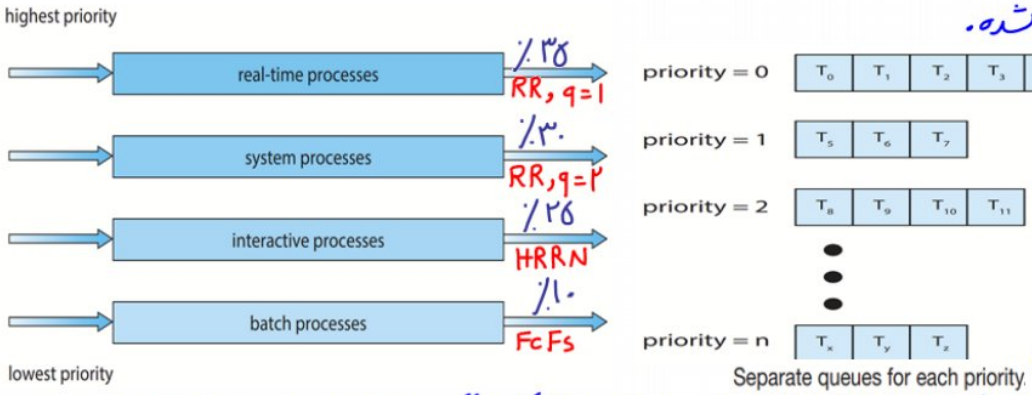
در priority scheduling اگر همه فرآیندها را در یک صف قرار بدیم بسته به نحوه مدیریت صف ممکن است هر بار به یک جستجو با مرتبه O(n) برای پیدا کردن فرآیند با بالاترین اولویت نیاز داشته باشیم، بنابراین در عمل اغلب ساده تر است که صف های جداگانه برای هر اولویت مشخص داشته باشیم

بعضی از فرآیندها نیاز دارند سریع اجرا بشوند و برخی می توانند دیرتر اجرا بشوند و برخی از فرآیندها نیاز به کوانتوم کوچک و برخی نیاز به کوانتوم بزرگتر دارند، حالا اگر قرار باشه همه توی یک صف باشند و کوانتوم رو ثابت بگیریم

ایراد: بسته به پیاده سازی ممکن است هر بار برای پیدا کردن فرآیندی که بالاترین اولویت را دارد نیاز به جستجو داشته باشیم

دوما: اگر کوانتوم بزرگ باشه responsetime کلی میاد پایین و اگر کوچیک بگیریم سربار cs بالا میره و UT میاد پایین

براساس ایده priority scheduling است ولی فرآیندها براساس ماهیت شون در سطوح مختلف پخش می شود، بالاترین سطح به فرآیندهای real time که نیازمند توجه فوری هستند داده شده.



- فرآیندهای تعاملی فرآیندهایی هستند که تعامل و I/O زیادی با یوزر دارند، مانند پخش یک فیلم که در یک ثانیه ۲۵ تا ۳۰ فریم باید برای کاربر پخش کنیم و گویی این تعداد با سیستم تعامل دارید.

interactive editing process : تعامل کاربر کمتر است، مانند کار با word یا تایپ کردن.

batch process : آن دسته از دستوراتی که پشت سر هم و بصورت یک دسته از دستورات و بصورت یکباره توسط os اجرا می شوند، پردازش دسته ای عموماً برای کارهای غیر تعاملی و background task مانند چاپ و... استفاده می شود، این روش استفاده کارا از منابع را فراهم می کند و به ساده سازی و اجرای موثر عملیات تکراری کمک می کند.

این سیستم ها برخلاف سیستم های interactive به محض اینکه هر کاری وجود داشته باشد به آن واکنش نشان نمی دهند و کارها را جمع می کنند تا به یک دسته برسد و سپس به یکباره به همه آنها پاسخ می دهند.

یکی از کاربردهای سیستم های دسته ای در mainframe اینگونه بود که عموماً یک پرینتر وجود داشت و تعداد زیادی کاربر وجود داشت و اینطوری نبود که هر یوزری پرینت اش را برای پرینتر ارسال کند و بلافاصله بتواند پرینت اش را بردارد و اینگونه بود که یوزرها درخواست های پرینت شان را می دادند و وقتی این درخواست ها به یک مقدار مشخص می رسید یا به یک زمان مشخص که می رسیدیم بصورت دسته ای همه اینها پرینت می شد.

کاربر امروزی: طراح os ممکن است با توجه اهدافی که دارد و باتوجه به منابع، کارها و ماهیت کارها تصمیم بگیرد به محض انجام برخی از درخواست ها به آن ها پاسخ ندهد و وقتی آن درخواست ها به یک حد نصاب مشخص رسید و یک دسته را به وجود آورد بعد به آنها پاسخ بدهیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

- اگر بصورت ایستا به یک فرآیند اولویت دهیم، فرآیندها در طول مدت اجرایشان در صف خودشان باقی می مانند.

یک تقسیم بندی رایج دیگر به این شکل است که دو صف اولویت ایجاد می‌کنند: یکی برای فرآیندهای foreground = interactive و یکی برای فرآیندهای background = batch. که این ۲ نوع فرآیند نیازهای response time متفاوت و بنابراین نیازمند زمانبندی متفاوتی هستند، همچنین فرآیندهای foreground با توجه به تعریف خارجی که برایشان انجام می‌شود ممکن است اولویت بیشتری نسبت به فرآیندهای background داشته باشند. بنابراین هر صف می‌تواند الگوریتم زمانبندی خاص خود را داشته باشد، مثلاً صف foreground می‌تواند RR باشد در حالیکه صف background می‌تواند FcFs باشد.

- حالا علاوه بر اینکه ما باید در هر صف یک زمانبندی داشته باشیم باید یک زمانبندی هم بین صف‌ها داشته باشیم که معمولا بصورت fixed-priority preemptive scheduling پیاده‌سازی می‌شود.

الان در مثال اولیه ما هر صف دارای اولویت مطلق نسبت به صف‌های با اولویت پایین‌تر است و هیچ فرآیندی در یک صف نمی‌تواند اجرا شود مگر اینکه همه صف‌های بالاتر از آن همگی خالی باشند، حال اگر مثلاً در حین اجرای یک فرآیند مربوط به صف چهارم یک فرآیند وارد هر یک از صف‌های آماده ۱ و ۲ و ۳ شود، فرآیند batch در حال اجرا preempt می‌شود و به فرآیند با اولویت بالاتر که تازه آمده داده می‌شود.

- اولویت مطلق که بین این صف‌ها وجود دارد باعث می‌شود که سطوح پایین‌تر اولویت بتوانند با starvation مواجه بشوند.

چگونه می‌توانیم از وقوع starvation جلوگیری کنیم؟

بیایم و به هر سطح اولویت یک سهمی از زمان پردازنده را اختصاص بدهیم و مثلاً می‌توان نوعی از Round Robin را بین سطوح به وجود آوردیم، برای هر کدام از سطوح اولویت هم می‌توانیم یک الگوریتم زمانبندی داخلی مربوط به خودش را داشته باشیم، حالا چون هر چند وقت یکبار یک کم زمانی به همه سطوح اولویت میدهیم پس همه فرآیندهای متعلق به سطوح مختلف در زمان پیشرفت می‌کنند و starvation رخ نمی‌دهد، البته به شرطیکه الگوریتم زمانبندی که برای هر سطح انتخاب می‌کنیم خودش starvation نداشته باشد

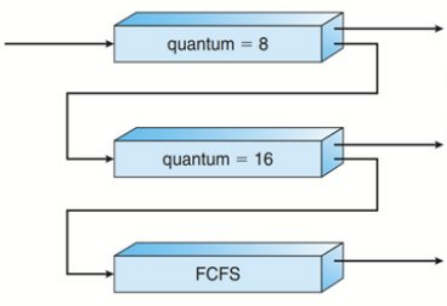
- مثلاً وقتی دو تا صف foreground و background داریم می‌توانیم ۸۰٪ از زمان cpu را به صف foreground بدهیم و برای این صف از زمانبندی RR استفاده کنیم و ۲۰٪ از زمان cpu را به صف background دهیم و برای این صف از زمانبندی FcFs استفاده کنیم.

Multilevel Feedback queue scheduling (MLFQ)

به طور معمول در multilevel queue scheduling وقتی فرآیندی وارد سیستم می‌شود بطور دائم به یک صف اختصاص داده می‌شود و فرآیندها از یک صف به صف دیگر منتقل نمی‌شوند، این روش سربار زمانبندی را کم می‌کند اما انعطاف پذیری را نیز کم می‌کند

★ رفتار یک فرآیند از نظر interactivity می‌تواند بصورت پویا عوض شود و برآورد اولیه ما می‌تواند درست نباشد

MLFQ به یک فرآیند اجازه میدهد تا بین صف‌ها حرکت کند و ایده‌اش این است که فرآیندها را با توجه به cpu burst هایشان از هم جدا کند و اگر فرآیندی از زمان cpu بیش از حد استفاده کند به این معنی است که کمتر تعاملی و I/O bound است و بنابراین آن را به سطوح پایین‌تر منتقل می‌کند، علاوه بر این فرآیندی که به مدت زیادی در یک صف با اولویت پایین منتظر مانده را ممکن است به صف‌های با اولویت بالاتر منتقل کند که این نوعی از Aging است برای جلوگیری از starvation است.



Multilevel feedback queues.

همان زمانبند صف چند سطحی است که از اتفاقاتی که برای فرآیندها می‌افتد فیدبک می‌گیرد و با توجه به این فیدبک‌ها فرآیندها را بصورت زیر زمانبندی می‌کند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

- هرچه تعاملات (interaction‌های) یک فرآیند بیشتر باشد به این معنی است که آن فرآیند I/o bound تر است و cpu burst‌های کوتاه‌تری دارد و بنابراین کوانتوم‌های کوتاه برای این فرآیندها مفیدتر هست و کارشون رو راه میندازه.

حالا اگر به شکل بالا که مربوط به زمانبندی MLFQ هست توجه کنید متوجه می‌شید که هرچه از اول‌های بالا به سمت پایین‌تر می‌آییم کوانتوم‌ها بزرگتر می‌شوند و در سطح آخر که دیگه اصلا کوانتوم نداریم و FCFS است.

بنابراین: هرچه فرآیندی تعاملش با یوزر بیشتر باشد باید در صف‌های بالاتر و هرچه کمتر باشد باید در صف‌های پایین‌تر قرار بگیرد.

روش الگوریتم:

فرض می‌کنیم فرآیندهای تازه وارد highly interactive هستند و آنها را به صف اول که بالاترین اولویت را دارد واردمی‌کنیم که اگر واقعا اینگونه باشند در کوانتوم زمانی کوتاهی که در صف اول بهشون داده می‌شود کارشون تموم می‌شوند و از سیستم خارج می‌شوند، به این ترتیب response time کارهای کوچک، کوچک می‌شود و متقاضیان این فرآیندها معطل نمی‌شوند، ولی اگر در این کوانتوم اجرایش تمام نشود تصورمان را نسبت به این فرآیند عوض می‌کنیم و به این نتیجه می‌رسیم که این فرآیند interactivity کمتری دارد و بنابراین باید به سطح پایین‌تری از اولویت منتقل شود و در واقع این همان بازخوردی (feedback ای) است که ما از سیستم می‌گیریم. و این داستان در صف‌های دیگر نیز ادامه دارد.

پس ابتدا توی صف اول که کوانتوم اش کوچک است میذاریم که اگر highly interactive است response time اش کم شود و اگر اینگونه نبود به این معناست که اگر توی این صف نگه اش داریم برای اجراش مجبوریم هی cs کنیم و سربار cs میره بالا و ۸ پایین می‌آید، پس می‌بریم اش جایی که کوانتوم‌هاش بزرگتره که cs کمتری داشته باشه و سربار cs کم بشود زمانبند ابتدا تمامی فرآیندهای صف ۰ را اجرا می‌کند و تنها زمانیکه صف ۰ خالی شود فرآیندهای صف ۱ را اجرا می‌کند و به همین ترتیب و اگر مثلاً cpu در حال اجرای فرآیندی از صف ۲ ام است و یک فرآیند جدید وارد صف ۰ شود باید فرآیند مربوط به صف ۲ ام که در حال اجرا است را preempt کنیم و فرآیند وارد شده به صف ۰ را اجرا کنیم.

- هر فرآیندی که در هر صفی cpu burst اش را تمام کند با یک سیستم کال به سراغ Next I/o burst اش می‌رود و دفعه بعدی که بخواد cpu burst بعدی اش را انجام دهد وارد صف ۰ ام می‌شود. ← هر فرآیندی که بلاک شد بعد از بیداری وارد صف ۰ ام میشه

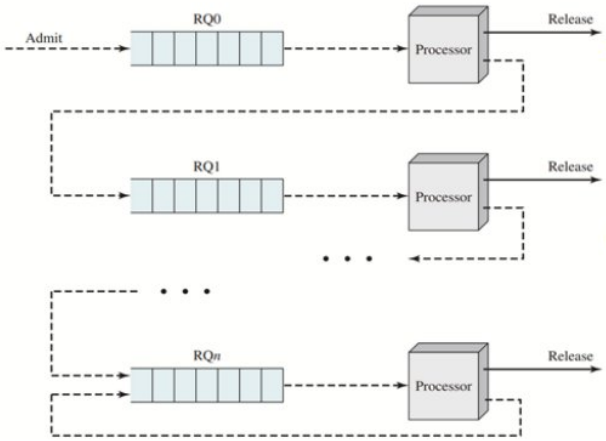
⭐ ما داریم یک برآوردی از نوع فرآیند می‌کنیم و آن را درون یک صف میذاریم ولی شاید برآورد ما از آن فرآیند برای همان مقطع از فرآیند درست باشد و برای همه فرآیند درست نباشد، یعنی چی؟

بعنوان مثال وقتی روی یک فیلم کلیک می‌کنید که پخش شود در ابتدای کار که در حال لود شدن است هنوز interaction ای با کاربر اتفاق نیفتاده و در این حالت اینگونه تصور می‌شود که این فرآیند خیلی تعاملی نیست و بنابراین به سطوح پایین‌تر منتقل می‌شود ولی بعدا متوجه می‌شویم که interactivity این فرآیند بالا است و بهتر است به سطوح بالاتر اولویت منتقل شود، بنابراین در این حالت می‌شود از مکانیزم Aging استفاده کرد و اگر یک فرآیند خیلی در سیستم ماند هرچند وقت یکبار اولویت اش را افزایش بدهیم و به سطوح بالاتر منتقل اش کنیم به اینصورت اگر برای یک فرآیند صف مناسبی انتخاب نشده باشد پس از مدتی در جایگاه درست و مناسب خودش قرار می‌گیرد.

⭐ توجه کنید که ما در ابتدا در خصوص بسیاری از فرآیندها اطلاع دقیقی از Next cpu burst شان نداریم و برای همین بود که در خیلی از الگوریتم‌ها مثل nonpreemptive SJF و SRTF و HRRN نیاز به تخمین داشتیم ولی در این الگوریتم دیگر نیازی به تخمین نداریم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



نکته) استالینگ به این الگوریتم feedback یا multilevel feedback می‌گوید.
- وقتی یک فرآیند وارد صف آخر می‌شود تا زمانیکه اجرایش تمام شود بصورت مکرر به این صف برگردانده می‌شود و این صف بصورت RR رفتار می‌کند
- هنگامی که یک فرآیند Block یا preempt می‌شود آن فرآیند را به یکی از چند صف اولویت باز می‌گرداند

نکته) تنسنبام به این میگه multiple queue.

★ استالینگ و تنسنبام یک ورژنی از این رو توضیح دادن که هم صف‌ها RR کار می‌کنند و کوانتوم در صف i ام برابر 2^i است (با شروع i از ۰) و هر فرآیندی در هر جایی که سیستم کال زد بعد از بیداری به صف اول وارد میشد. (فرآیندی که مدت زیادی خواب بوده و cpu نگرفته وقتی بیدار شد بخش اولویت بدیم)

سوال) فرض کنید فرآیند A یک cpu burst ـ ۱۲۰ کوانتومی دارد، برای این فرآیند چند بار cs انجام می‌شود اگر:
الف) از RR با q=1 استفاده کنیم؟ ۱۲۰ بار :response time کارهای کوچک خوبه ولی خیلی cs داریم و سربار cs زیاده
ب) " " " q=6 " " ؟ ۲۴ " " " " بهتره " cs میاد پایین
ج) از ورژن MLFQ با ۷ صف استفاده کنیم؟ ۷ بار

شماره صف	۰	۱	۲	۳	۴	۵	۶
تعداد کوانتومی که در آن صف می‌گیرد	۱	۲	۴	۸	۱۶	۳۲	۶۴
مجموع کوانتوم‌هایی که می‌گیرد	۱	۳	۷	۱۵	۳۱	۶۳	۱۰۰

+۳۷

تعداد cs = $\lfloor \lg 120 \rfloor + 1 = 6 + 1 = 7$

در حالت ج هم response time کارهای کوچک از حالت الف هم بهتر است و هم cs اش از هم بهتره.

در خصوص مقایسه " حالت الف و ج توجه کنید که درسته که کارهای کوچک در هر ۲ یک کوانتوم می‌گیرند و کارشون تموم میشه ولی در RR وقتی یک فرآیند وارد صف اول می‌شود هم فرآیندهایی که از قبل ready هستند فارغ از تعداد کوانتومی که قبلا گرفتند در جلوی فرآیندی که الان وارد صف اول می‌شوند هستند، اما در MLFQ فرآیندهایی که ready هستند در صف‌های مختلف پخش شده اند و بنابراین تعداد فرآیندهایی که جلوی این فرآیندی است که الان وارد صف اول شده کمتر است.

نکته) اگر هیچ نشانه‌ای از طول next cpu burst فرآیندهای مختلف نداشته باشیم هیچ کدام از الگوریتم‌های SJF، SRTF، HRRN و ... را نمی‌توان استفاده کرد برای همین یک راه برای ایجاد اولویت برای کارهای کوچکتر جریمه کردن کارهایی است که مدت طولانی‌تری اجرا می‌شوند به عبارت دیگر درسته که نمی‌توانیم روی زمان باقیمانده اجرا تمرکز کنیم ولی می‌توانیم روی زمانی که فرآیند تا الان برای اجرا صرف کردند تمرکز کنیم.

نکته) از ویژگی‌های این الگوریتم مناسب بودن آن برای سیستم‌های interactive و قابل پیاده سازی بودن آن است.

نکته) درسته این الگوریتم response time را برای کارهای کوچک و تعاملی (I/O bound) کاهش میدهد ولی ART و AWT را افزایش میدهد

پاراستگی و پیچیدگی‌های موجود در طراحی این زمانبند

۱- چند صف اولویت کافی است؟

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

۲- الگوریتم زمانبندی مورد استفاده در هر صف چه باشد و چه پارامترهایی داشته باشد؟
۳- طبق چه قواعدی اولویت یک فرآیند می‌تواند زیاد یا کم بشود؟
۴- آیا هم فرآیندها از همان ابتدا به بالاترین صف وارد بشود یا از ابتدا وارد صف‌های دیگر هم بشود و پس با توجه به شرایط سیستم سطح اش را عوض کنیم؟

lottery scheduling (تنبادم)

به هر فرآیند متناسب با اولویتی که دارد برای استفاده از منابع مختلف سیستم مثل cpu یک تعداد بلیط lottery tickets میدهیم مثلاً به فرآیندی که اولویت اش n است، n تا بلیط میدهیم، حالا هر زمان که قرار است تصمیمی برای زمانبندی گرفته شود یک بلیط قرعه‌کشی بصورت تصادفی انتخاب می‌شود و فرآیندی که آن بلیط را در اختیار دارد آن منبع را دریافت می‌کند.

- وقتی از این الگوریتم برای زمانبندی cpu استفاده می‌شود، این الگوریتم ممکن است ۵۰ بار در ثانیه عمل قرعه‌کشی را انجام بدهد و هر برنده ۲۰ms از زمان cpu را بعنوان جایزه دریافت می‌کند، پس کوانتوم داریم و مکانیزم preemption داریم.

- فرض کنید به یکسری فرآیند ۱۰۰ تا بلیط داریم و یک فرآیند با توجه به اولویت اش ۲۰ تا بلیط دریافت کرده، پس این فرآیند در هر بار قرعه‌کشی ۲۰ درصد شانس برنده شدن دارد و در درازمدت ۲۰ درصد از cpu را در اختیار خواهد داشت.

- در priority scheduler وقتی مثلاً اولویت فرآیندی ۴۰ است، بیان اینکه داشتن اولویت ۴۰ به چه معناست دشوار است ولی در این روش کسی که ۴ درصد از بلیط‌ها را در اختیار دارد تقریبا ۴ درصد از پردازنده را نیز در اختیار می‌گیرد.

- توجه کنید تا زمانیکه یک فرآیند جدید وارد سیستم بشود یا یک فرآیند کارش تمام شود بلیط‌های داده شده به فرآیندها پس گرفته نمی‌شود و تغییری در تعداد بلیط‌ها نداریم، وقتی فرآیند جدیدی وارد شود براحتی و با توجه به اولویت اش بخش بلیط جدید می‌دهیم، پس فرآیندهای جدید در قرعه‌کشی بعدی نسبت به تعداد بلیطی که دارد شانس برنده شدن خواهد داشت و بنابراین این الگوریتم highly responsive است و اگر فرآیندی خارج شود می‌توانیم بلیط‌هایش را بداریم کنار و در نظر نگیریم.

- فرآیندهای همکار (cooperating processes) می‌توانند در صورت تمایل بلیط‌های خودشان را مبادله کنند.

مثلاً اگر فرآیند مشتری A پیامی به فرآیند سرور B ارسال می‌کند و سرویسی از آن می‌خواهد که برای ادامه کارش به آن نیاز دارد و بنابراین باید بلاک شود می‌تواند تمام بلیط‌های خود را به فرآیند B قرض دهد تا شانس اجرا بعدی سرور زیاد شود که کار سرور زودتر تمام شود و جواب فرآیند A را سریعتر بدهد، حالا وقتی کار سرور تمام شد تیکت‌های فرآیند A را برمی‌گرداند تا این فرآیند بتواند مجددا اجرا شود، توجه کنید که در غیاب مشتری‌ها، سرورها اصلا نیازی به بلیط ندارند. فرآیندی که بلاک است غیر مستقیم از کمک اش استفاده می‌کند نکته) این الگوریتم می‌تواند برای حل مشکلاتی که رسیدگی به آنها با روش‌های دیگر دشوار است استفاده شود، مثلاً در یک سرور ویدیویی که چندین فرآیند می‌خواهند video stream هایی با نرخ‌های مختلف برای مشتریانشان بفرستند مفید است، مثلاً فرض کنید سه فرآیند داریم که هریک مسئول ارسال video stream هایی با فریم ریت‌های ۱۰، ۲۰ و ۲۵ فریم در ثانیه هستند، حالا با تخصیص ۱۰، ۲۰ و ۲۵ بلیط به هر کدام از اینها به طور خودکار cpu را به همین درصدها بین ۳ فرآیند تقسیم می‌کنیم، بنابراین می‌توانیم منابع را بصورت زمانی بین فرآیندها مالتی پلکس کنیم

Fair-share scheduling (یک RR است که بین کاربرها می‌چرخد نه بین فرآیندها)

این الگوریتم برخلاف الگوریتم‌های قبلی به جای توجه به فرآیندها به مالکان فرآیندها توجه می‌کند، ما تا اینجا فرض کردیم که هر فرآیند به

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

تنهایی زمانبندی می‌شود بدون اینکه به این توجه کنیم که مالک آن فرآیند چه کسی است.

در روش‌های قبلی اگر یوزر ۱، ۹ فرآیند ایجاد کند و یوزر ۲، ۱ فرآیند، با یک الگوریتم RR یا اولویت برابر، یوزر ۱، ۹۰٪ از cpu و یوزر ۲، ۱۰٪ از cpu را می‌گیرد، برای جلوگیری از این وضعیت برخی از سیستم‌ها قبل از زمانبندی فرآیندها در نظر می‌گیرند که کدام کاربرها مالک کدام فرآیندها هستند و در این مدل به هر کاربر کسری از cpu داده می‌شود و زمانبند به گونه‌ای فرآیندها را برای اجرا روی cpu انتخاب می‌کند که آن سهم‌ها رعایت بدهد، بنابراین اگر به ۲ یوزر قول داده شده باشد که هرکدام ۵۰ درصد از cpu را دریافت کنند، مهم نیست که چه تعداد فرآیند یوزر ۱ دارد و چه تعداد یوزر ۲ حتما یوزر ۱ و ۲ هرکدام ۵۰٪ از زمان cpu را دریافت می‌کنند.

- بعنوان مثالی برای حالت بالا فرض کنید یوزر ۱ دارای چهار فرآیند A و B و C و D و یوزر ۲ فقط یک فرآیند E را دارد، حالا

RR بین فرآیندها: ABCDEABCDE...

" " کاربران: AEBECEDEAEBECEDE...

حالا اگر به یوزر ۱ سهمی ۲ برابری نسبت به یوزر ۲ داده باشیم خواهیم داشت: ABECDEABECDE...

Multiple-processor scheduling

تا الان در خصوص زمانبندی در سیستم‌های تک پردازنده‌ای صحبت کردیم، حالا می‌خواهیم برخی از ملاحضات مربوط به سیستم‌های چند پردازنده‌ای را بررسی کنیم.

از یک منظر سیستم‌های چند پردازنده‌ای به ۲ دسته زیر تقسیم می‌شوند:

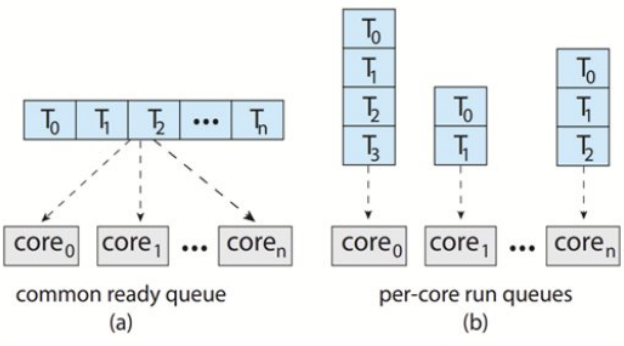
۱- Asymmetric multiprocessing: برخی از پردازنده‌ها نسبت به دیگر پردازنده‌ها وظایف متفاوتی داشته باشند، مثلاً ممکن است یکی از این پردازنده‌ها Master باشد (Master server) و تمام تصمیمات زمانبندی، مدیریت عملیات I/O و سایر فعالیت‌های سیستمی توسط این پردازنده واحد مدیریت شود و سایر پردازنده‌ها فقط کدهای کاربر را اجرا کنند. این روش ساده است چون فقط یک core به data structure های سیستم دسترسی دارد و این موضوع نیاز data sharing را کاهش میدهد.

نقطه ضعف این رویکرد این است که Master server پتانسیل این را دارد که به گلوگاه تبدیل شود و عملکرد کلی سیستم را کاهش دهد

۲- symmetric multiprocessing (SMP): متقارن: یک رویکرد استاندارد برای پشتیبانی از چند پردازنده است، در این رویکرد همه پردازنده‌ها وظایف یکسانی دارند و بنابراین هر core خودش زمانبندی خودش را انجام میدهد (هر core self scheduling است) به این معنا که هر core فرآیند بعدی را از ready queue بر میدارند و اجرایش می‌کنند. حالا ۲ حالت برای ready queue می‌تواند وجود داشته باشد:

۱- به ازای هر core یک صف ready در سیستم داشته باشیم و همه تردها در این ready queue مشترک باشند

۲- هر پردازنده صف ready خصوصی خود را برای تردها داشته باشد



توجه) اگر فقط یک صف وجود داشته باشد یک possible race condition وجود دارد و باید اطمینان حاصل کنیم که دو پردازنده مجزا یک ترد را برای schedule کردن روی خودشان انتخاب نمی‌کنند، یا دو تا ترد اگر نخواهند همزمان به انتهای این صف اضافه شوند حواسمان باید باشد که یکی شون به لیل رونویسی دیگری از بین نرود. دسترسی به یک صف مشترک خودش می‌تواند گلوگاه باشد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

بنابراین رایج ترین رویکرد در سیستم‌هایی که از SMP استفاده می‌کنند این است که هر core صف ready جداگانه خودش را داشته باشد همچنین داشتن private ready queue منجر به استفاده کارآمدتر هر core از حافظه کش خودش می‌شود

دانستنی: تمام os‌های مدرن از جمله ویندوز، لینوکس، macOS، Android و iOS از رویکرد

در ادامه فرض می‌کنیم هر پردازنده صف ready خودش را دارد، بنابراین یک پخش کننده‌ای وجود دارد که تصمیم می‌گیرد کدام فرآیند یا ترد به صف ready کدام پردازنده برود، حال شاید برای این که بخواهیم بار پردازنده‌ها را بالانس کنیم بخواهیم پخش کننده را به اینصورت پیاده‌سازی کنیم که هر فرآیند را بصورت چرخشی به یکی از صف‌های ready مربوط به هر پردازنده بدهیم درصورتیکه این کار لزوما باعث بالانس بار روی پردازنده‌ها نمی‌شود و دلیل اش آن است که لزوما cpu burst فرآیندهای مختلف یکسان نیست و این که هر فرآیند چه cpu burst‌ای داشته باشد به ماهیت آن فرآیند بستگی دارد و os هم نمی‌تواند تخمین دقیقی از آن داشته باشد درنتیجه ممکن است سر یکی از پردازنده‌ها خلوت بشود و سر تعدادی دیگر شلوغ بشود و بنابراین یک عدم توازن (imbalance) بین بار روی پردازنده‌ها به وجود می‌آید.

حالا یکی از روش‌هایی که برای توازن بار استفاده می‌شود Migration است که به این معناست که فرآیندها از روی پردازنده‌های شلوغ مهاجرت کنند روی پردازنده‌های خلوت.

البته خود مهاجرت هزینه‌هایی دارد که باید حواسمان به آن هزینه‌ها نیز باشد

processor Affinity (وابستگی به پردازنده)

نکته) فرض کنید یک فرآیند (ترد) روی یکی از cpu‌ها در حال اجرا است بنابراین داده‌هایی که این فرآیند اخیرا به آنها دسترسی پیدا کرده روی کش مربوط به این cpu قرار گرفته است، درنتیجه دسترسی‌های متوالی این فرآیند به حافظه اغلب از طریق کش خودش (warm cache) برآورده می‌شود، حالا اگر بخواهیم این فرآیند (ترد) را به دلیل load balancing به cpu دیگری منتقل کنیم، اولا باید محتوای کش در source processor را invalidate کنیم (cache invalidating) و ثانیا حالا این فرآیند روی پردازنده مقصد برای دسترسی به اکثر اطلاعاتی که نیاز دارد باید به حافظه مراجعه کند تا محتوای کش به مرور زمان در پردازنده مقصد آپدیت شود (کش مقصد باید repopulate شود که این مورد باعث می‌شود این فرآیند تا یک مدت زمانی روی پردازنده مقصد کندتر اجرا بشود

- به دلیل هزینه بالای invalidating و repopulating حافظه کش اکثر os‌هایی که از SMP پشتیبانی می‌کنند سعی می‌کنند از migrate کردن یک فرآیند (ترد) از یک پردازنده به پردازنده دیگر پرهیز کنند و درعوض تلاش می‌کنند تا یک فرآیند (ترد) را روی همان پردازنده‌ای که اجرا می‌شود نگه دارند تا از مزایای warm cache استفاده کنند.

processor Affient: یک فرآیند به پردازنده‌ای که در حال حاضر روی آن اجرا می‌شود وابستگی دارد

پس به دلیل وابستگی فرآیندها به پردازنده مبدا باعث می‌شود خیلی تمایلی به migration نداشته باشیم، که بسته به اینکه چقدر تمایل داریم در ده‌های مختلف از واژه‌های مختلفی استفاده می‌شود.

Hard Affinity: اصلا تمایلی برای مهاجرت وجود ندارد و os تضمین می‌کند که هیچ وقت یک فرآیند مهاجرت نکند.

برخی از os‌ها فراخوان‌هایی دارند که hard Affinity را پشتیبانی می‌کنند و به یک فرآیند اجازه میدهند تا زیر مجموعه‌ای از پردازنده‌ها را مشخص کند که فرآیند می‌تواند روی آن اجرا بشود

Soft Affinity: خیلی " " " " " " سعی می‌کند از مهاجرت اجتناب کند ولی تضمین نمی‌کند که هرگز مهاجرتی اتفاق نیفتد و یک فرآیند ممکن است در اثر load balancing بین پردازنده‌ها مهاجرت کند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

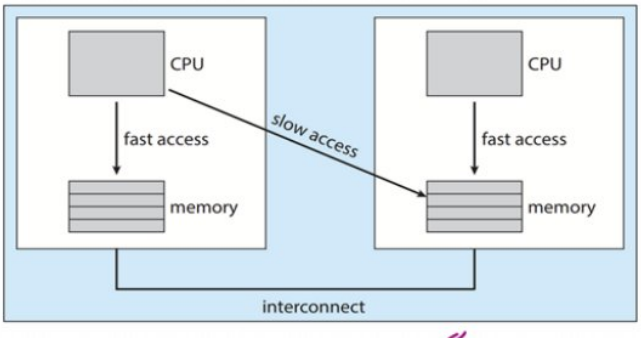
نکته) بسیاری از os ها هم soft و هم Hard، Affinity را ارائه میدهند، عنوان مثال لینوکس soft Affinity را پیاده سازی کرده اما فراخوانی سیستمی sched_setaffinity() را نیز فراهم کرده که با اجازه دادن به یک ترد برای مشخص کردن مجموعه ای از cpu های واجد شرایط که می تواند روی آنها اجرا شود از Hard Affinity هم پشتیبانی می کند.

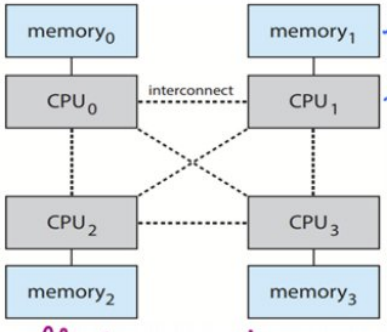
نکته) ما ۲ رویکرد برای ready queue بیان کردیم، یکی داشتن یک ready queue مشترک برای همه پردازنده ها و یکی اینکه هر پردازنده صف ready خودش را داشته باشد، اگر رویکرد ready queue مشترک را اتخاذ کنیم ممکن است یک ترد پس از هر باری که به صف ready مشترک می آید روی یک پردازنده جدید schedule بشود در اینصورت حافظه کش پردازنده جدید مجددا باید repopulate بشود، اما وقتی هر پردازنده صف ready خودش را داشته باشد یک ترد همیشه روی پردازنده مربوط به خودش schedule می شود و بنابراین در این حالت می تواند از محتویات یک warm cache بهره مند بشود. بطور کلی داشتن صف ready برای هر پردازنده processor Affinity را بصورت رایگان فراهم می کند.

Numa(Non-Uniform Memory Access)

اگر تعداد پردازنده ها زیاد بشه رقابت برای بدست آوردن باس سیستم افزایش پیدا می کند و به bottleneck تبدیل میشه و در نتیجه کارایی کاهش پیدا می کند یکی از رویکردهای جایگزین برای حل این مشکل و افزایش scalability این هست که هر cpu یا گروهی از cpu ها حافظه اختصاصی و محلی خودشان را داشته باشند که از طریق یک باس کوچک local به آن حافظه متصل هستند و در عین حال هم پردازنده ها از طریق یک سیستم میان ارتباطی مشترک (shared system interconnect) هم متصل می شوند به نحوی که (گویی) همه cpu ها یک فضای آدرس نیز یکی را به اشتراک می گذارند (گویی همه cpu ها به یک حافظه مشترک بزرگ دسترسی دارند)، این رویکرد دسترسی غیر یکنواخت حافظه (NUMA) شناخته می شود.

نکته) مزیت این روش این است که وقتی یک cpu به حافظه محلی خود دسترسی پیدا می کند نه تنها سریع است بلکه هیچ رقابتی روی شبکه ارتباطی سیستم به وجود نمی آورد (آن را درگیر نمی کند)، بنابراین با اضافه شدن پردازنده های بیشتر NUMA systems ها می توانند بطور موثرتری scale شوند. یکی از اشکالات بالقوه در سیستم های NUMA این است که اگر یک cpu بخواهد به حافظه remote دسترسی پیدا کند چون باید این کار را از طریق interconnect انجام دهد تاخیر افزایش پیدا می کند و باعث کاهش کارایی می شود که os باید از طریق مدیریت خوب حافظه و cpu scheduling دقیق این NUMA penalty را به حداقل برساند.





نکته) از آنجاییکه سیستم های NUMA می توانند تعداد زیادی پردازنده را در خود جای دهند به طور فزاینده ای در سرورها و سیستم های High-performance computing محبوب هستند.

نکته) معماری حافظه اصلی یک سیستم می تواند روی مسائل مربوط به processor Affinity نیز اثر بگذارد

شکل بالا سمت چپ یک معماری NUMA را نشان میدهد که در آن دو چیپ پردازنده وجود دارد که هر کدام cpu و حافظه محلی خود را دارند اگرچه یک system interconnect به همه cpu های یک سیستم NUMA اجازه میده تا یک فضای آدرس فیزیکی را به اشتراک بگذارند اما یک cpu دسترسی سریع تری به حافظه محلی خود نسبت به حافظه محلی cpu دیگر دارد، حال اگر cpu scheduler و الگوریتم های memory-placement NUMA-aware باشند و با هم کار کنند، تردی که بر روی یک cpu زمانبندی شده می تواند به نزدیکترین حافظه به آن cpu منتقل شود و بنابراین

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

سریعترین دسترسی ممکن به حافظه را برای آن ترد فراهم می‌کند.

نکته) load balancing معمولا مزایای process Affinity را خنثی می‌کند و بطور بالقوه انتقال یک ترد بین پردازنده‌ها ممکن است برای سیستم‌های NUMA جریمه به همراه داشته باشد، جایی‌که یک ترد ممکن است به پردازنده‌ای منتقل شود که زمان دسترسی اش به حافظه بیشتر شود، بعبارت دیگر یک کشمکش ذاتی بین load balancing و به حداقل رساندن زمان دسترسی به حافظه وجود دارد

load balancing in SMP

در سیستم‌های SMP خیلی مهم که برای استفاده کامل و بهینه از مزایای داشتن بیش از یک پردازنده بتونیم workload رو بین تمامی پردازنده‌ها بالانس نگه داریم، اگرنه ممکنه یک یا چند پردازنده بیکار بشوند در حالیکه چند پردازنده دیگه workload خیلی بالا به همراه صف‌های آماده‌ای که پر از ترد در حال انتظار برای گرفتن cpu هستند.

هدف load balancing این هست که تلاش کند که workload بصورت یکنواخت در تمام پردازنده‌های یک SMP توزیع بشه.

★ load balancing معمولا فقط در سیستم‌هایی ضروری است که هر پردازنده ready queue مختص به خودش را دارد و در سیستم‌هایی که یک ready queue مشترک وجود دارد load balancing ضروری نیست.

چرا؟ چون وقتی یک پردازنده بیکار می‌شود بلافاصله یک ترد را برای اجرا از ready queue برمیدارد.

حالا اگر قرار باشد مهاجرت انجام شود باید ببینیم برای رسیدن به توازن بار انواع سیاست‌های مهاجرت چی هست.

two approaches to load balancing

۱- pull migration : وقتی یک پردازنده بیکار میشه به بقیه پردازنده‌ها سر میزند و اگر ببیند سر یک پردازنده شلوغ ممکن است یک فرآیند را که در حالت انتظار سرویس گرفتن روی پردازنده شلوغ است را بردارد و روی خودش ببرد که در این حالت اصطلاح pull کردن برای این کار استفاده می‌شود.

۲- push migration: یک task خاص بصورت دوره‌ای (پریودیک) وضعیت روی پردازنده‌ها را چک می‌کند و اگر عدم توازن(imbalance) شناسایی شد، load از ست پردازنده overloaded به ست پردازنده کم بارتر (less-busy processor) ، push می‌شود.

نکته) هیچ یک از این دو ایده نسبت به دیگری بهتر نیست و به همین دلیل اغلب بصورت موازی و از ترکیب این دو در سیستم‌های متعادل کننده بار برای Migration استفاده می‌شود که بهش push-pull می‌گویند.

نکته) مفهوم تعادل بار (balance load) می‌تواند معانی مختلفی داشته باشد، مثلاً

- همه صف‌ها تقریبا به اندازه هم ترد داشته باشند
- توزیع مساوی اولویت‌های نخ‌ها در تمامی صف‌ها

Heterogeneous Multiprocessing (HMP) (چندپردازشی ناهمگن)

تا الان همه پردازنده‌ها را از نظر قابلیت‌هاشون یکسان در نظر گرفتیم، اما الان برخی از سیستم‌های موبایل هستند که دارای coreهایی هستند از نظر سرعت کلاک و مدیریت توان متفاوت هستند (کنترل مصرف توان (power consumption) یک core را تا رسیدن به حد بیکاری) چنین سیستم‌هایی بعنوان HMP شناخته می‌شوند.

توجه) توجه کنید این شکلی از Asymmetric multiprocessing نیست زیرا در اینجا هم taskهای سیستم و هم taskهای کاربر می‌توانند روی همه هسته‌ها اجرا شوند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

هدف HMP مدیریت بهتر مصرف توان با تخصیص taskهایی که نیازهای خاصی دارند به هسته‌های خاص است.
مثلاً coreهای بزرگ با کارایی بالاتر که انرژی بیشتری مصرف می‌کنند باید برای پریودهای کوتاهی مصرف شوند و هسته‌های کوچکتر که انرژی کمتری مصرف می‌کنند می‌توانند برای پریودهای طولانی‌تری استفاده شوند.

مزیت‌های ترکیب هسته‌های بزرگ با هسته‌های کوچک

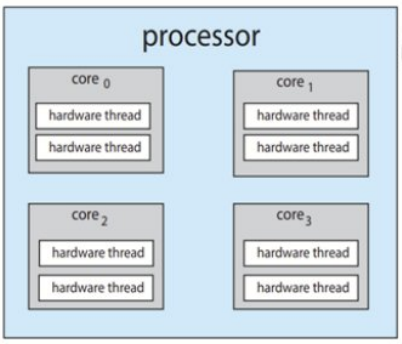
۱- با ترکیب تعدادی از هسته‌های کندتر با هسته‌های سریعتر یک cpu scheduler می‌تواند taskهایی که نیاز به عملکرد بالا (High performance) ندارد اما ممکن است نیاز به اجرا برای پریودهای طولانی‌تری داشته باشد مثل background task را به هسته‌های کوچک بدهد و در نتیجه به حفظ شارژ باتری کمک کند.
به طور مثال interactive applicationها که به قدرت پردازشی بیشتری نیاز دارند اما ممکن است برای مدت زمان کوتاه‌تری اجرا شوند می‌توانند به هسته‌های بزرگتر تخصیص داده شوند

۲- اگر دستگاه موبایل روی power-saving mode باشد می‌توانیم هسته‌های بزرگ انرژی بر (energy-intensive) را غیرفعال کنیم و سیستم فقط با هسته‌های کوچک که energy efficient هستند کار کند

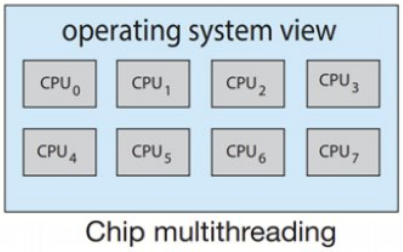
دانستنی: windows 11 از HMP scheduling پشتیبانی می‌کند و به یک ترد اجازه میدهد تا scheduling policy ای را انتخاب کند که بهترین پشتیبانی را از خواسته‌های مدیریت انرژی‌اش را انجام میدهد.

symmetric Multithreding (SMT)

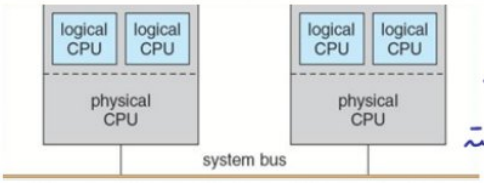
SMP: تعدادی پردازنده فیزیکی داریم و داریم فرآیندها یا ترد‌ها را بین آنها پخش می‌کنیم که بصورت concurrent اجرا شوند.
SMT: " " منطقی " (logical processor) که ترد‌ها بصورت concurrent روی آنها اجرا می‌شوند.



در شکل سمت چپ ۴ تا core فیزیکال داریم که روی هر کدام دو تا ترد سخت افزاری ایجاد شده (شبیه‌سازی شده) (core منطقی) که این امر بصورت سخت افزاری (در ۲ یه سخت افزار) پیاده سازی می‌شود
- هر core حافظه کش اختصاصی خود را دارد (L1 و شاید L2) و هر ترد سخت افزاری رجیسترهای خود را دارد و بنابراین برای os این تصور وجود دارد که ۸ تا core روی پردازنده وجود دارد.
- بطور سنتی سیستم‌های SMP با ارائه چندین پردازنده فیزیکی اجازه میدند چندین فرآیند بصورت موازی اجرا شوند، با این حال اکثر رایانه‌های امروزی در حال حاضر چندین core را روی یک چیپ فیزیکی قرار میدهند و در این شرایط هر هسته از نظر os یک cpu logical جداگانه است.
سیستم‌های SMP که از پردازنده‌های چند هسته‌ای استفاده می‌کنند نسبت به سیستم‌هایی که هر cpu دارای چیپ فیزیکی خود هست سریعتر و مصرف انرژی کمتری دارد.



Chip multithreading

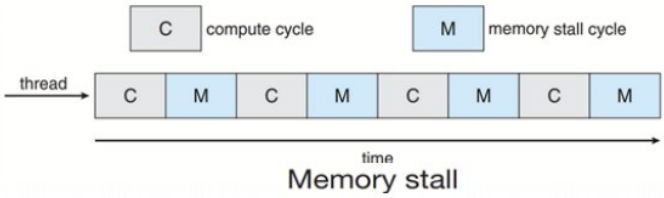


خب حالا چرا می‌ریم سراغ ترد‌های سخت افزاری؟ مزیت‌های ترد‌های سخت افزاری

۱- وقتی multi core ساده داریم زمانیکه یک پردازنده می‌خواد به حافظه رجوع کند به علت کند بودن حافظه نسبت به کش، خیلی معطل میشه که این وضعیت بعنوان memory stall شناخته می‌شود. وقتی cache miss هم رخ میده ما memory stall خواهیم داشت.
در این سناریو پردازنده می‌تواند حتی تا ۵۰٪ از زمان خود را صرف انتظار برای available شدن داده‌ها از حافظه بکند.

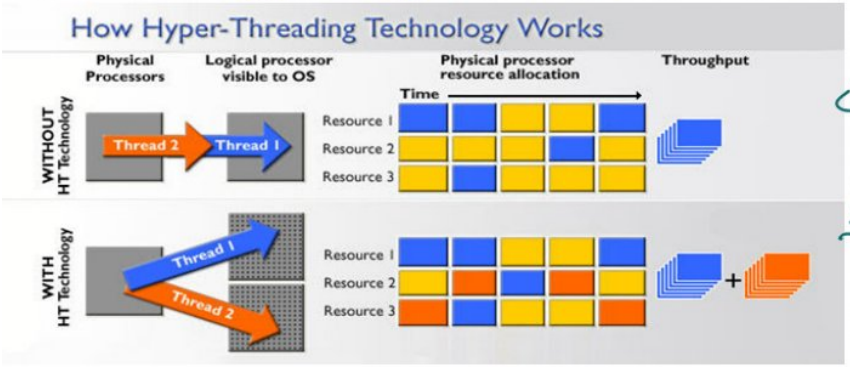
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



۲- یکی دیگر از مزیت های ترد نخ افزاری همان دلایلی بود که باعث شد multiprogrammig و multitasking شکل بگیرد، ماجرا رفتیم سراغ multiprogramming ؟ گفتیم بیایم چندین فرآیند نداریم توی RAM که وقتی یک فرآیند رفت دنبال عمل I/O و cpu رو رها کرد، cpu را بیکار نداریم که هدر بره و یک فرآیند آماده دیگر که توی RAM هست رو روی cpu زمانبندی کنیم. حالا اینجا هم می خوایم همون ایده رو روی منابع سخت افزاری هر core انجام بدهیم.

فرض کنید یک فرآیند رو داریم به core1 و آن فرآیند داره عملیات محاسباتی Integer روی واحد محاسباتی Integer مربوط به آن core انجام میدهد و با واحد محاسباتی ممیز شناور (FPU) آن core کاری ندارد، حالا اگر ما بتوانیم یک فرآیند دیگر را که عملیات ممیز شناور دارد را به این core بدهیم و این core بتونه این ۲ فرآیند را همروند با هم اجرا کنه بدین ترتیب هر ۲ واحد محاسباتی Integer و FP این core داره استفاده میشه و هیچ جزء سخت افزاری آن core بیکار نمی ماند و بدین ترتیب ما داریم بهتر از منابع سخت افزاری مان استفاده می کنیم.



بنابراین داستان ترد سخت افزاری به وجود آمد که این امکان را برای ما فراهم کرد که بیش از یک ترد روی هر core بتوانیم اجرا کنیم و به این ترتیب تحت مدیریت سخت افزار سعی بیشتر تا جای ممکن منابع سخت افزاری هدر نرود.

حالا os می تواند از وجود تردهای سخت افزاری آگاه باشد یا نه

۱- اگر os نسبت به وجود تردهای سخت افزاری آگاه نباشد ممکن است ۲ فرآیند که هر دوی آن ها مثلا با واحد محاسباتی integer کار دارند را روی یک core قرار بدهد که در اینصورت یک ترد به دلیل تداخل با ترد دیگر معطل می شود و به این ترتیب مزیتی از حیث نکته بالا بدست نمی آورد

۲- اگر os نسبت به وجود تردهای سخت افزاری آگاه باشد می تواند دو تا فرآیند که با هم تداخل دارند را روی دو core مختلف قرار بدهد و پس دو تا فرآیند سنگین را روی آن دو core قرار بدهد که در اینصورت ۲ ترد روی یک core با هم تداخلی نخواهند داشت و همروندی و در نتیجه تسریع بالاتر خواهد رفت

★ در این شرایط یک core بصورت سخت افزاری می تواند بین تردهای سخت افزاری (logical core) در لول instruction (fine-grained) یا بصورت coarse-grained (درشت دانه) زمانبندی انجام بدهد.

Thread Scheduling

ما چه سطوحی از زمانبندی را داشتیم ؟

سطح ۱) process-contention scope (PCS) : رقابت درون یک فرآیند :

یک سطح از زمانبندی داشتیم که بین user thread و توسط thread library انجام می شد که در این روش thread library رقابت بین تردهای یک فرآیند را مدیریت می کرد.

سطح ۲) system-contention scope (SCS) : رقابت در سطح سیستم :

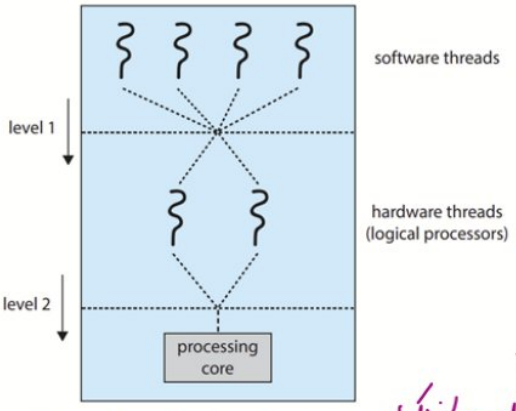
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

در سطح بعدی بین کرنل تردهای که همان تردهای منتخب thread library در سطح os هستند توسط cpu scheduler ـ os زمانبندی می‌شوند، یادمان هست که برای هر فرآیند ممکن است یک یا چند کرنل ترد وجود داشته باشد که cpu scheduler سیستم عامل از بین آنها یکی را برای زمانبندی روی cpu انتخاب می‌کند.

سطح ۳) زمانبند سطح سخت افزار بین تردهای روی یک core زمانبندی می‌کند

اول os scheduler تصمیم میگیره که کدام یک از تردهای نرم افزاری روی یک ترد سخت افزاری (logical core) اجرا شود که تمام الگوریتم‌هایی که قبلا خوندیم در این راستا بود و برای این سطح از زمانبندی os ممکن است هر الگوریتم زمانبندی ای را انتخاب کند، حالا این سطح از زمانبندی مشخص میکند که هر هسته چگونه تصمیم بگیرد که کدام ترد سخت افزاری در هر لحظه اجرا شود



Two levels of scheduling

⭐ توجه کنید که os هر ترد سخت افزاری را معادل یک core می‌بیند بنابراین os scheduler تردهای را روی تردهای سخت افزاری قرار میدهد، اما حواستون باشه که روی یک core در هر لحظه فقط یک ترد سخت افزاری اجرا میشه، پس حالا زمانبند سطح سخت افزار مربوط به هر core باید بین تردهای سخت افزاری مربوط به یک core زمانبندی انجام بدهد و تصمیم بگیره که در هر لحظه کدامیک از رسورس‌های آن core در اختیار کدامیک از تردهای سخت افزاری فعال روی آن core قرار بگیرد

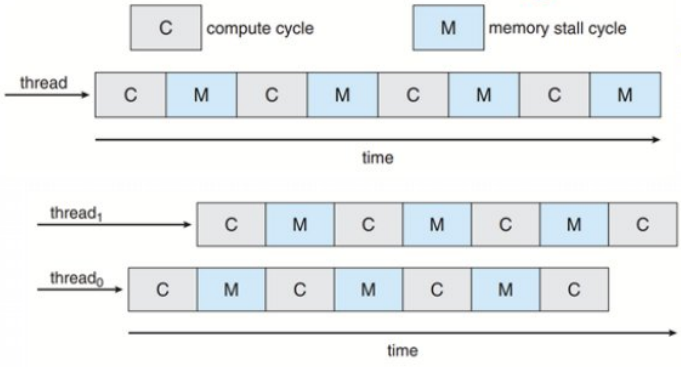
الگوریتم‌های زمانبندی که در سطح سخت افزار وجود دارد از یک پردازنده به پردازنده دیگر و از یک شرکت به شرکت دیگر می‌تواند متفاوت باشد.

مثلاً توی ماشین UltraSPARC T3 روی یک core که ۸ تا ترد سخت افزاری داره به سادگی الگوریتم RR در سطح این core انجام میشه فقط حواستون باشه کوانتوم زمانی RR در اینجا در حد چند سایکل هست و مثل کوانتوم زمانی RR در سطح os در سطح میلی ثانیه نیست.

یا مثلاً در پردازنده Itanium ـ اینتل که دو تا core داره و هر core اش دو تا ترد سخت افزاری دارد، معیاری به اسم Dynamic urgency که عددی بین ۰ تا ۷ است وجود دارد که به هر ترد سخت افزاری یک DU داده می‌شود و هر زمان DU یک ترد سخت افزاری بالاتر باشد آن ترد برای اجرا انتخاب می‌شود.

نکته) به این تکنیک که هر core چند تا ترد سخت افزاری داشته باشه CMT (chip multithreading) می‌گویند و اینتل به این تکنیک SMT یا hyper-threading (simultaneous multithreading) می‌گویند.

coarse-grained and fine-graine Multithreading



Multithreaded multicore system.

۱- زمانبندی coarse-grained: در یک نگاه درشت دانه می‌توانیم اینطور فرض کنیم که یک ترد یک مدت زمانی را صرف computation می‌کند که به این مدت compute cycle می‌گیم و یک مدت زمانی را صرف انتظار برای مهیا شدن داده اش توسط حافظه می‌کند که به این memory stall cycle می‌گیم.

حالا اگر فرض کنیم هر ترد با همین نظم اجرا بشه و ما دو تا ترد سخت افزاری داشته باشیم زمانبندی دوبه دو می‌تواند برای این ۲ ترد ایده آل باشد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

یعنی وقتی تردd به memory stall cycle اش می رسد زمانبند سخت افزار بیاد و compute cycle تردd را اجرا کند و بالعکس.
در اینصورت نه هیچ وقت حافظه و نه هیچ وقت منابع پردازشی آن core بیکار نیستند و بالاترین همروندی بین تردها برقرار می شود.

۲- Interleaved or Fine-graine:
زمانبند سخت افزار بتواند خیلی ریزتر و در سطح instruction زمانبندی را انجام بدهد می گوییم آن زمانبندی fine-graine است.
نکته) سخت افزار زمانبند محقق این روش پیچیده تر است چون باید بتواند به inst تصمیم بگیرد که در هر لحظه بهتر است کدام inst مربوط به کدام ترد را برای اجرا روی core انتخاب کند، این پیاده سازی باعث می شود بتوانیم همروندی بهتری را بین inst ها داشته باشیم

Real-Time cpu scheduling

سیستم های بی درنگ یا real time یا حساس به زمان: سیستم هایی که محاسبات آنها باید تا یک زمان مشخص تمام شود
سیستم های بی درنگ به ۲ دسته اصلی تقسیم می شوند

۱- HRT (Hard Real-time): قیدهای زمانی و ددلاین ها همیشه باید رعایت بشوند و حتی اگر یکبار این مسئله نقض بشود ممکن است فاجعه بزرگی در آن سیستم به وجود آید.

۲- SRT (" soft): این امکان وجود دارد که در درصدی از ددلاین task ها رعایت نشود که در اینصورت شاید کیفیت کار ما کم بشه و به یک فاجعه ختم نمی شود.

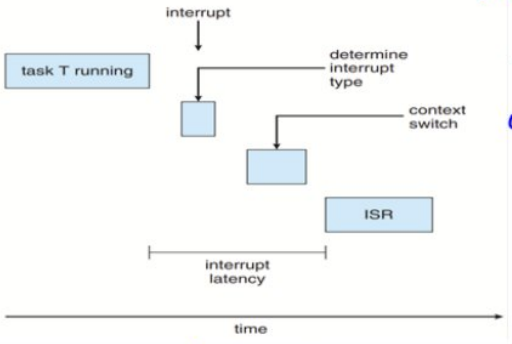
Deadline (موعد زمانی)
از زمانیکه یک event در محیط اتفاق می افتد (t.) ما یک مهلتی داریم که باید قبل از مهلت باید به آن event واکنش نشان بدهیم

laxity (استراحت): مدت زمانیکه از لحظه ورود یک task فرصت داریم تا اجرا آن را شروع کنیم به نحوی که آن کار قبل از ددلاینش تمام شود.

Event latency: از زمان وقوع یک رویداد (t.) تا زمانیکه سیستم بتواند واکنش نشان بدهد

موارد که روی Event Latency تاثیر میگذارند

۱) interrupt latency: تاخیر وقفه یکی از پایدارترین پارامترهایی که روی event latency تاثیر می گذار است و برابر است با: از زمانیکه یک وقفه به cpu می آید تا زمانیکه بتوانیم ISR مربوط به آن وقفه را فراخوانی کنیم.



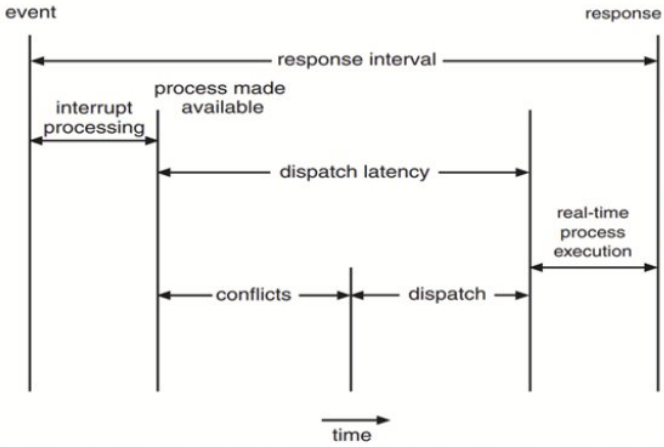
وقتی event ای در محیط رخ بدهد این event به واسطه وقفه به اطلاع سیستم می رسد، حالا پس از وقوع وقفه ابتدا باید صبر کنیم تا اجرای دستور جاری تمام شود که این مدت زمان برمی گردد به تعداد سیکل هایی که آن دستور نیاز دارد و همین طور اندازه هر سیکل و بعد باید نوع آن وقفه مشخص شود و پس از مشخص شدن نوع وقفه باید context switch انجام بشود تا در نهایت ما بتوانیم ISR مربوط به آن وقفه را شروع کنیم.

نکته) اگر به واسطه اینکه os رفته توی جاهای حساس کرنل، کرنل را برای مدتی non preemptive کرده باشیم، یعنی وقفه را غیرفعال

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

کرده باشیم، خب در این مدت زمان به وقفه‌ها واکنش نشان نمی‌دهیم و بنابراین این مدت روی interrupt latency تاثیر منفی میگذارد پس در سیستم‌های real_time باید سعی کنیم مدت زمانی که وقفه‌ها غیرفعال می‌کنیم را minimum کنیم.



۲) Dispatch latency: پس از اجرا شدن ISR مربوط به وقفه مورد نظر ممکن است یک فرآیند مربوط به آن event برای اجرا مشخص شود و در صف ready قرار بگیرد.
حالا مدت زمانی‌که طول میکشه تا cpu scheduler این فرآیند را Dispatch کند Dispatch latency می‌گوییم.

نکته) باید سعی کنیم این تاخیر را به حداقل برسانیم، موثرترین تکنیک برای کم نگه داشتن این تاخیر محیا کردن preemptive Kernel است.
برای سیستم‌های Hard Real time این تاخیر در حد چند میکروثانیه است.

فاز conflict مربوط به Dispatch latency شامل ۲ بخش است:

۱- preemption هر فرآیند اجرا شونده در صف

۲- آزادسازی منابع مورد نیاز یک فرآیند با اولویت بالا توسط فرآیندهای با اولویت پایین

Dispatch latency می‌تواند تحت تاثیر ۲ زمان باشد، یکی اینکه ممکن است اولویت برخی از فرآیندهای درون ready queue بیشتر از این فرآیند باشد و این فرآیند لازم باشد صبر کند یا اینکه منابعی که این فرآیند برای اجرایش نیاز دارد در اختیار فرآیندهای دیگری باشد و بنابراین ما باید صبر کنیم تا آن منبع آزاد شود و بعد وارد صف ready می‌شود.
پس از آنکه فرآیند مورد نظر Dispatch شد روی cpu یک burst time ای دارد و پس از پایان اجرا ممکن است سیستم واکنش به محیط نشان بدهد. جمع همه این زمان‌ها event latency می‌شود.

آیا event latency می‌تواند باعث miss شدن Deadline بشود؟

ساده‌سازی: یکی از پارامترهای تاثیرگذار در event latency را کنار میذاریم و نشان میدهیم هنوز زمانبندی در سیستم‌های Real Time سخت است.

interrupt latency را ۰ در نظر می‌گیریم و همچنین فرض می‌کنیم که هیچ وقت این اتفاق نمی‌افتد که یک فرآیند با اولویت بالا یک منبعی را بخواهد که آن منبع در اختیار یک فرآیند با اولویت پایین است و در نتیجه بخواهد معطل بشود، پس فرض می‌کنیم فرآیندها روی هیچ resource مشترکی به جز cpu رقابت نمی‌کنند و بنابراین هر زمان که یک فرآیند با اولویت بالا وارد می‌شود می‌تواند فرآیند با اولویت پایین را preempt کند، بنابراین در این حالت conflict و در اثر معطل شدن فرآیندهای کم اولویت تر برای فرآیندهای با اولویت بالا تر در صف ready خواهد بود.

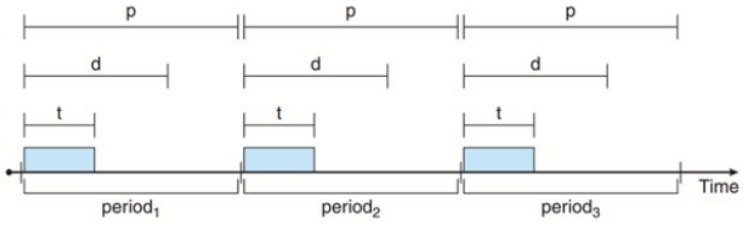
زمانبندی فرآیندهای بی‌درنگ

تعدادی فرآیند real time داریم که هر یک Deallline مربوط به خودشان را دارند حالا می‌خواهیم به گونه‌ای اینها را روی cpu زمانبندی کنیم که همه فرآیندها Deallline شون meet بشود.

در اکثر سیستم‌های Real time یک فرآیند Real time بصورت پریودیک اجرا می‌شود و هیچ‌گاه خاتمه پیدا نمی‌کند. مثلاً یک سیستم بی‌درنگ درون

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



Periodic task

هواپیما داره وضعیت دمای درون هواپیما یا دمای موتور را مانیتور می‌کند
این کار را بصورت پریودیک هر مدتی یکبار تا همیشه انجام میدهد.

p: بنابراین یکی از پارامترهای یک فرآیند real time پریود است
و این فرآیند هر p واحد زمان یکبار می آید

t: execution time = burst time: یک پارامتر دیگر فرآیند زمان اجرای آن است

d = Deadline

Periodic Task T:(t,p,d)

هر p واحد زمان یک فرآیند می‌آید ما انتظار داریم اجرایش قبل از Deadline اش تمام شود.

نکته) در بسیاری از سیستم‌های real time، پریود و Deadline مساوی است که در اینصورت فرآیند را بصورت زیر نشان میدهند

periodic Task T:(t,p) p=d

که در این شرایط باید $p \geq t$ باشد.

نکته) می دانیم که $Rate = 1/p$

نکته) اگر فقط یک فرآیند (Task) در سیستم داشته باشید برای آنکه در این اش را meet کند فقط کافی است $t \leq p$ باشد.

اگر n تا فرآیند داشته باشیم

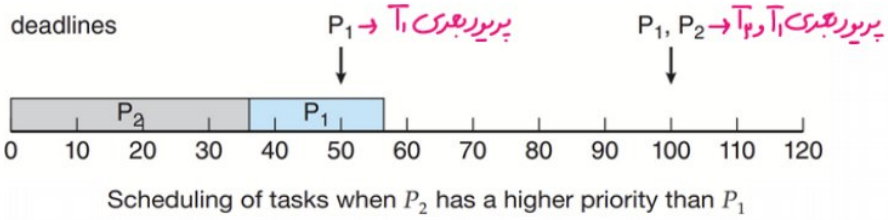
فرض کنید n فرآیند داریم با پریودهای (P_1 تا P_n) و زمان اجرای (t_1 تا t_n) خودشان با فرض d=p حالا چیکار کنیم که همه فرآیندها در این شان را meet کنند

Rate_Monotonic scheduling

این الگوریتم یک الگوریتم استاتیک با preemption است و اولویت را به فرآیندی میدهد که Rate اش بیشتر است یا پریودش کوچکتر است

static priority Algorithm: اگر از ابتدا اولویت یک فرآیند از فرآیند دیگری بیشتر باشد تا انتها اولویت به همان شکل است

فرض کنید ۲ تسک زیر را داریم:



Scheduling of tasks when P_2 has a higher priority than P_1

$T_1:(20,50)$, $T_2:(35,100)$
+ P

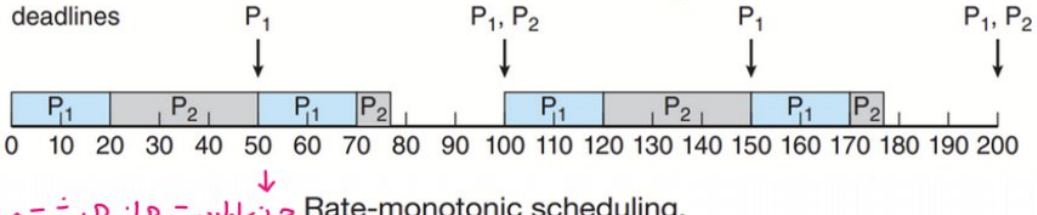
اولویت این ۲ تسک به ۲ صورت می تواند باشد

۱- اولویت T_2 بیشتر از T_1 باشد.

۲- " T_1 " " T_2 . طبق RM.

۱) هر ۲ تسک در زمان ۰ وارد سیستم می‌شوند و چون اولویت T_2 بیشتره بین T_1 و T_2، T_2 اجرا میشه و ۳۵ واحد اجرا می‌شود و در لحظه ۳۵ تمام می‌شود و در لاین صف خودش را meet میکند، پس از T_2، اجرا T_1 شروع می‌شود که ۲۰ واحد اجرا می‌شود و در لحظه ۵۵ تمام می‌شود و در لاین ۵۰ خودش را miss می‌کند و این زمانبندی نتوانست هم در لاین ها را meet کند.

۲) RM:



Rate-monotonic scheduling.

T_1 و T_2 طبق RM در لاین شان را meet می‌کنند و زمانبندی این مثال طبق RM feasible است.

چون اولویت P_1 از P_2 بیشتره P_2 توسط P_1 preempt میشه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

چگونه می‌توانیم مطمئن شویم که RM بصورت موفق زمانبندی می‌کند؟

$$\text{Task utilization} = \frac{\text{زمان اجرا آن تسک}}{\text{پریود}}$$

$$u_1 = \frac{t_1}{P_1} = \frac{۲۰}{۵۰} = ۰,۴ \quad , \quad u_2 = \frac{t_2}{P_2} = \frac{۳۵}{۱۰۰} = ۰,۳۵$$

- $u_1 = ۰,۴$ به چه معناست؟ تسک T_1، ۴۰ درصد از زمان پردازنده را می‌گیرد چون هر ۵۰ واحد زمانی ۲۰ واحد زمانی cpu میخواهد

شرط لازم: برای آنکه n تسک قابلیت اجرا شدن روی یک پردازنده را داشته باشد باید: $\sum_{i=1}^{n} u_i \leq 1$

مشخص است که اگر $\sum_{i=1}^{n} u_i > 1$ باشد اصلا پردازنده نمی‌تواند همه آن تسک‌ها را اجرا کند، اصلا ظرفیت اش رو ندارد و زورش نمیرسه

شرط کافی: $U_{RM} = n(2^{1/n} - 1)$ — Admission control

اگر جمع u تسک‌ها از این عدد کوچکتر شود RM گارانتی می‌کند که زمانبندی feasible باشد، یعنی همه تسک‌ها ددلاین‌شان را meet میکنند.

$n=1 \rightarrow U_{RM}=1$		
$n=2 \rightarrow U_{RM}=۰,۸۲$	$\Rightarrow \sum u_i \leq ۰,۸۲$	Admission control
$n=3 \rightarrow U_{RM}=۰,۷۸$	$\Rightarrow \sum u_i \leq ۰,۷۸$	
$\vdots$		
$n=\infty \rightarrow U_{RM}\simeq ۰,۶۹$	$\Rightarrow \sum u_i \leq ۰,۶۹$	

نکته) اگر فقط ۱ تسک داشته باشیم اگر utilization آن تسک زیر ۱ باشد یعنی $D \geq P$ باشد می‌گیم زمانبندی طبق RM اوکیه.

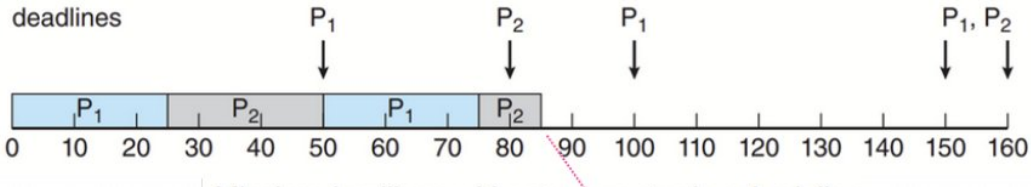
★ این شرط یک شرط کافی است و ممکن است برقرار نباشد ولی باز هم طبق RM قابل زمانبندی باشند ولی دیگه گارانتی شده نیست.

Earliest-Deadline-First scheduling (EDF)

یک الگوریتم Dynamic priority - preemptive است و اولویت با تسکی است که Deadline اش زودتر است.

Dynamic priority: ممکن است در لحظاتی از زمانبندی اولویت با T_1 و در لحظاتی دیگر اولویت با T_2 باشد

مثال) فرض کنید دو تسک بصورت T_1: (۲۵,۵۰) و T_2: (۳۵,۸۰) داریم که هر ۲ زمان ۰ وارد شدن و میخواهیم این دو را با RM و EDF زمانبندی کنیم

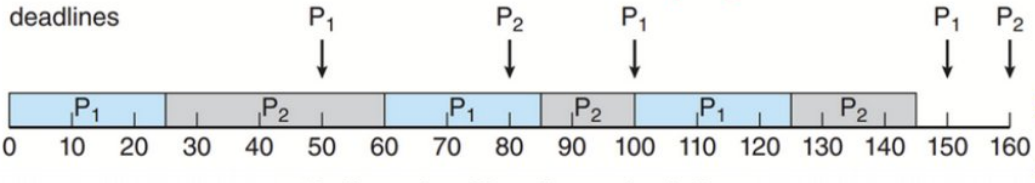


Missing deadlines with rate-monotonic scheduling.

$$u_1 = \frac{۲۵}{۵۰} = ۰,۵$$
$$u_2 = \frac{۳۵}{۸۰} = ۰,۴۳۷۵$$
$$\sum u_i \simeq ۰,۹۴$$

شرط لازم برقرار است و شرط کافی برای ۲ تسک برای RM برقرار نیست.

در لحظه ۰ دو تا تسک داریم، T_1 با ددلاین ۵۰ و T_2 با ددلاین ۸۰ پس اولویت با T_1 است تا ثانیه ۲۵ که تمام می‌شود، در ثانیه ۲۵ فقط T_2 در سیستم است پس تا ۵۰ اجرا میشه، در ثانیه ۵۰ ما هم T_1 را داریم و هم T_2، یکی T_2 که ددلاین اش ۸۰ است و یکی T_1 که ددلاین‌ش ۱۰۰ است که بین این دو ددلاین T_2 جلوتر است پس T_2 ادامه میده



Earliest-deadline-first scheduling.

و ...

★ تا کجا باید این زمانبندی رو جلو ببریم تا مطمئن بشیم تا آخر هم تسک‌ها ددلاین‌شون رو meet میکنند؟

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

تا کوچکترین مضرب مشترک پریود تسک ها = ک.م.م = Least common multiple

LCM(۵۰, ۸۰) = ۴۰۰

شرط لازم و کافی برای EDF : $\sum_{i=1}^{n} U_i \leq 1$

در EDF مادامیکه $\sum U \leq 1$ باشد، EDF تقسیم می‌کند که هم تسک ها در لاین شون را می بینند پس یک الگوریتم بهینه محسوب می‌شود.

نکته) الگوریتم RM بصورت بسیار گسترده در OS های Real time استفاده می‌شود ولی EDF خیلی در OS های Real time استفاده نمی‌شود!

چرا؟ الگوریتم EDF در تئوری عالی است ولی مشکلاتی مانند سربار context switch و انجام رفتارهای پیش بینی نشده را در زمان اجرا در برخی از شرایط از خود نشان میدهد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are automatically brought in later.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are automatically brought in later.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Memory Management

توجه: از الان هر وقت گفتیم حافظه یا memory منظورمون حافظه اصلی = RAM است.

چرا به مدیریت حافظه نیاز داریم؟ ۱- سرعت ۲- حفاظت

سرعت: هارد خیلی کنده و cpu نمی‌تونه با هارد کار کنه و cpu فقط می‌تواند با رجیسترهایش و حافظه اصلی کار کند، Instructionهایی وجود دارند که آدرس‌های حافظه را بعنوان آرگومان می‌گیرند ولی هیچ کدام آدرس دیسک را نمی‌گیرند، بنابراین هر دستور العمل در حال اجرا و هر داده‌ای که توسط این دستورات اجرا می‌شوند باید یا در ثبات‌های cpu یا حافظه گذاشته بشوند، حالا حافظه کوچیکه و فراره و برنامه‌ها هم زیاد و بزرگ هستند، پس ما نمی‌تونیم همه برنامه‌ها رو بیاریم توی حافظه و همه‌اش باید از توی هارد بیاریم شون توی RAM و باز ببریم شون توی هارد.

یک نکته دیگه هم هست و این هست که باید سرعت دسترسی به اطلاعاتی که توی RAM میذاریم باید قابل مدیریت و قابل پیش‌بینی باشد، اگر سعی کنیم داده‌هایی که قرار هست به ترتیب استفاده بشوند را کنار هم بچینیم سرعت میره بالاتر

حفاظت:

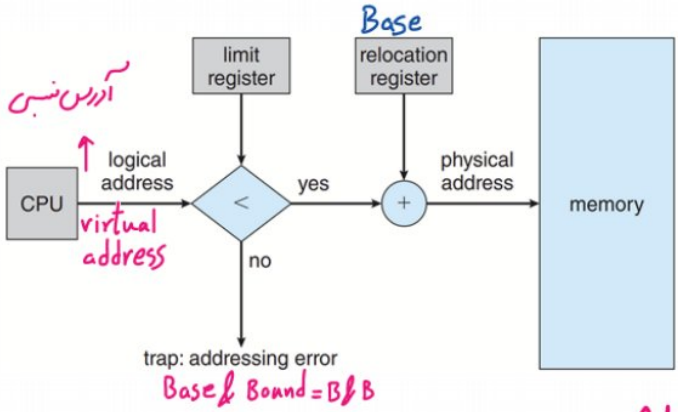
۱- برای عملکرد صحیح سیستم ما باید هم os را از دسترسی‌های کاربر محافظت کنیم و هم فرآیندها نباید به حافظه خصوصی سایر فرآیندها دسترسی داشته باشند، این حفاظت باید توسط سخت افزار انجام شود (تحت مدیریت os)

۲- اگر لازم بود که چند فرآیند مقداری حافظه مشترک داشته باشند باید بصورت کنترل شده بتوانیم این دسترسی رو بهشون بدهیم

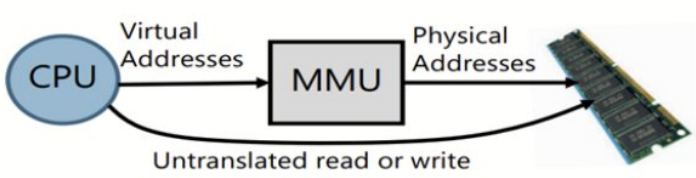
۳- باید بتوانیم به بخش‌های مختلف حافظه دسترسی‌های متفاوت نسبت بدیم، مثلاً به بخش کد دسترسی فقط خواندنی داشته

یکی از ساده‌ترین روش‌های مدیریت حافظه

- تا الان ما اینجوری فکر می‌کردیم که وقتی یک فرآیندی اجرا بشود باید بخش‌های کد و دیتایش از دیسک خوانده می‌شد و بخش هیپ و استک بهش اضافه می‌شد (و اگر چند تردی بود چند تا استک اضافه می‌شد) و کل این فضای آدرس در یک جای RAM قرار می‌گرفت و بعد توسط ۲ ثبات Base و limit محافظت می‌شد. به این روش Base & Bound = B&B نیز می‌گویند، در اینجا os به ازای هر فرآیند در context sw به MMU دو تا عدد Base و limit را در ثبات‌هایش میگذارد و اگر آدرس در محدوده خودش نباشد یک trap به os اتفاق می‌افتاد



در واقع اینجوری داریم Memory protection انجام میدیم



Address Binding

- آدرس‌ها در source program عموماً بصورت symbolic هستند و کامپایلر معمولاً این آدرس‌های symbolic را به آدرس‌های relocatable تبدیل می‌کند و مثلاً می‌گوید x بایت جلوتر از ابتدای module مورد نظر)، پس linker یا loader آدرس‌های relocatable را به آدرس‌های مطلق (absolute)، bind می‌کند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Binding: نگاشت از یک فضای آدرس به فضای دیگر آدرس.

مثلاً ما توی یک برنامه یک آدرس دهی داریم ولی وقتی این آدرس به حافظه فیزیکی داده می‌شود ممکن است به یک عدد دیگری تبدیل شده باشد که به این نگاشت Binding می‌گوییم. به طور کلاسیک Bind کردن inst و داده‌ها به آدرس‌های حافظه را می‌توان در دو یک از زمان‌های زیر انجام داد.

انواع Binding

Binding (این نگاشت) می‌تواند در زمان‌های مختلفی اتفاق بیفتد.

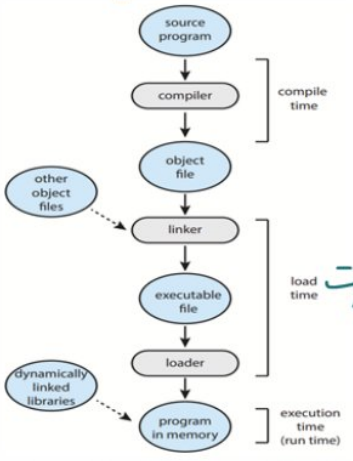
۱- compiler time: نگاشت بین آدرس‌های یک برنامه و آدرس فیزیکی که بوسیله آن به حافظه مراجعه می‌کنیم در زمان کامپایل انجام شود، اگر کدی به این نحو تولید شود به آن absolute code می‌گوییم، اگر به کد مطلق آن آدرس‌های حافظه‌ای را که در زمان کامپایل برایش مشخص شده را اندهیم آن کد تابلیت اجرا شدن را نخواهد داشت. اگر در زمان کامپایل بدانید که یک فرآیند در کجای حافظه قرار می‌گیرد کد مطلق می‌تواند تولید شود، حالا اگر جدا مکان شروع برنامه تغییر کند دوباره لازم است این کد کامپایل شود.

۲- load time: اگر در زمان کامپایل مشخص نباشد که فرآیند باید در کجای حافظه قرار بگیرد کامپایلر باید کد جابجاپذیر (relocated code) ایجاد کند. در اینجا کدی که کامپایلر ایجاد می‌کند اینگونه است که آدرس‌ها نسبت به ابتدای برنامه ایجاد شدند و زمانیکه loader (os) میخواد این برنامه را در حافظه لود کند با توجه به اینکه این برنامه در کجای حافظه load می‌شود، بایتری فایل‌ها را باز میکند و دستکاری‌شون میکند و آدرس‌های فیزیکی رو به جاشون قرار میدهد، loader برای جایگذاری آدرس‌های فیزیکی واقعی، آدرس‌های نسبی درون برنامه را با آدرس شروع مکانی که در حافظه که قرار است این برنامه قرار بگیرد (Base) جمع می‌کند، به این ترتیب Binding تا زمان load شدن به تعویق می‌افتد، پس این نوع از Binding در زمان load اتفاق می‌افتد.

⭐ دقت کنید که درسته که اینگونه می‌توانیم این کد را در هرجایی از حافظه load کنیم، اما اگر کد را خارج کنیم و بخواهیم دوباره وارد کنیم چون یک جای دیگه حافظه ممکن است قرار بگیره و آدرس شروع RAM اش عوض می‌شه دوباره باید Bind بشه. جایزه ۱ میلیون دلاری 😀😀

۳- run time: execution time: Binding تا زمان اجرا به تعویق می‌افتد

ما برای صرفه‌جویی در حافظه معمولا هم برنامه را وارد حافظه اصلی نمی‌کنیم و قسمت‌هایی که الان لازم داریم را load می‌کنیم و اگر قسمت‌های دیگری را نیاز داشتیم همان زمان آنها را وارد می‌کنیم (اصلا خیلی از قسمت‌های یک برنامه، شاید در هنگام استفاده از آن توی حافظه لود نشود و این به رفتار ما و برنامه‌ها مربوط می‌شود) حالا همون زمانیکه داریم قسمت‌های دیگه برنامه رو وارد می‌کنیم هست که مشخص میشه آن قسمت‌ها در کجای حافظه قرار می‌گیرند و بنابراین در همان زمان هست که آدرس‌های قرارگیری آنها در حافظه مشخص میشه - اگر بتوان فرآیند را در حین اجرای آن از یک بخش حافظه به بخش دیگری منتقل کرد بنابراین Binding باید تا زمان اجرا به تعویق بیفتد نکته) الان اکثر os‌های general-purpose اینجوری هستند.



اصطلاحات مربوط به آدرس دهی

۱- physical Address: آدرسی که توسط آن به حافظه فیزیکی (RAM) مراجعه می‌کنیم.

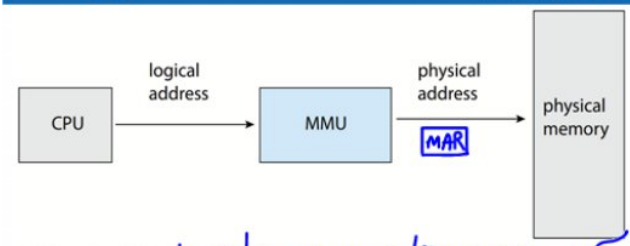
۲- logical Address: آدرسی که از cpu خارج می‌شود. نقطه 😀

⭐ اگر Binding آدرس‌ها در زمان کامپایل یا load باشد آدرس منطقی با آدرس فیزیکی یکسان است

۳- Virtual " : آدرس منطقی‌ای که در زمان اجرا (execution time Binding) به آدرس فیزیکی تبدیل می‌شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



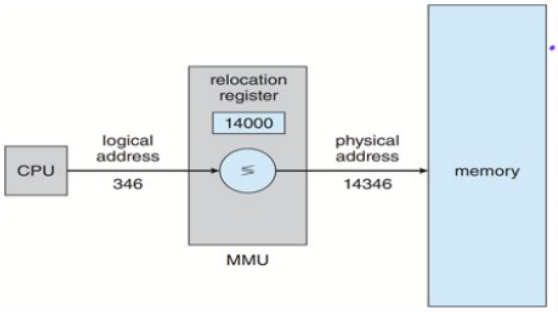
Binding آدرس ها در زمان اجرا منجر میشه که آدرس های منطقی و فیزیکی متفاوت بشود در این حالت معمولا به آدرس منطقی آدرس مجازی می گویند!!!

از یک جایی به بعد اگر گفتم آدرس منطقی منظورم همون مجازیه و اگر گفتم آدرس مجازی منظورم همون منطقیه!

logical address space : مجموعه تمام آدرس های منطقی تولید شده توسط یک برنامه

" " physical : " " " فیزیکی مربوط به آدرس های منطقی یک برنامه

پس در یک Binding زمان اجرا فضای آدرس منطقی و فیزیکی یک برنامه متفاوت است.



نکته) نگاشت آدرس های مجازی به فیزیکی در زمان اجرا توسط واحد سخت افزاری MMU انجام می شود، در این حالت دیگه نمی گیم ثبات base و ثبات base اینجا ثبات relocation نامیده می شود

- در این روش برنامه کاربر هرگز به آدرس های فیزیکی واقعی دسترسی ندارد، برنامه می تواند یک اشاره گره به مکان ۳۴۶ ایجاد کند و همه کار با این بکند ولی اغلب به آدرس های فیزیکی واقعی دسترسی ندارد (indirect)

پس برنامه کاربر با آدرس های منطقی سروکار دارد و MMU در زمان اجرا آدرس های منطقی را به فیزیکی تبدیل می کند.

توجه) برنامه کاربر با آدرس های منطقی سروکار دارد و memory mapping hardware آدرس های منطقی را به آدرس های فیزیکی در زمان اجرا تبدیل می کند و مکان نهایی یک آدرس ارجاع داده شده توسط برنامه تا این زمان مشخص نمی شود.

بریم ببینیم از ابتدا مدیریت حافظه چطوری بوده تا موارد بالا هم کامل جا بیفتد:

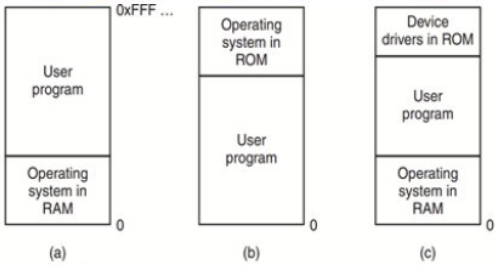
مکانیزم ۱) single contiguous Memory managment (تخصیص حافظه بصورت یک قطعه یکجا و پشت سرهم)

سیستم های اولیه بصورت uniprogramming کار می کردند و فقط یک فرآیند و یک OS در حافظه وجود داشت و این فرآیند تکی از یک جای از قبل مشخص توی حافظه لود می شد.

سیستم های embedded و کامپیوترهایی که در جاهای حساس استفاده می شود می تواند یک برنامه داشته باشد و مرتبا آن را اجرا کند.

وقتی OS برنامه را در آدرس فیزیکی مشخص شده لود کرد pc را به آن آدرس مقداردهی می کند و مقدار pc ابتدا کار ثابت است، کار یک برنامه که تمام شد آخرین دستورش OS را فراخوانی میکند تا OS اگر بخواهد برنامه دیگری وارد حافظه کند بتواند این کار را انجام بدهد

موقعی که برنامه را اجرا می کنیم نباید بتوانیم به فضای OS دسترسی داشته باشیم، بنابراین می توان آدرس شروع (یا انتها، بستگی به این دارد که OS کجا باشد) برنامه را در یک ثبات گذاشت و هر زمان برنامه آدرسی تولید می کند با این عدد مقایسه می شود و اگر از آن کوچکتر باشد اجازه دسترسی برنامه به آن گرفته می شود.



Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

- خب یک برنامه داریم و کارایی سیستم میاد پایین
- فرآیند بصورت پیوسته در حافظه قرار می گرفت
- " " " " " بزرگتر " این سیستم نمی توانست آن را اجرا بکند (در آن زمان هنوز بحث صفحه بندی و Demand paging و حافظه مجازی مطرح نشده بود)
- اگر برنامه ها از فضای در دسترس کوچکتر باشد حافظه اصلی هدر می رود ← مشکل Internal Fragmentation ← Hole در داخل فضای

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

تخصیص یافته به یک فرآیند.

- آدرس‌هایی که موقع اجرا فرآیند توسط cpu تولید می‌شود آدرس‌های فیزیکی (واقعی) است و Binding در زمان کامپایل انجام می‌شود ← چون میدانیم هر برنامه دقیقا کجای حافظه فیزیکی لود میشه.
- در این روش چون فقط 1 فرآیند در هر لحظه در حافظه وجود دارد نیاز به حفاظت بین فرآیندها وجود ندارد.

سوال: آیا میشه اینجا مقداری مزایات ایجاد کنیم؟ برنامه نویسی مالتی ترد ← هم ترد‌های یک فرآیند تصویر حافظه یکسان دارند
البته این ایده کاربرد محدودی دارد چون اغلب مردم می‌خواهند برنامه‌های نامرتبط را همزمان اجرا کنند، بعلاوه یک سیستم ای که اینقدر ابتدایی است بعیده که چند نخی رو بخواد ساپورت بکنه.

- در این سیستم‌ها خیلی مکانیزم مدیریت حافظه خاصی نداشتیم.

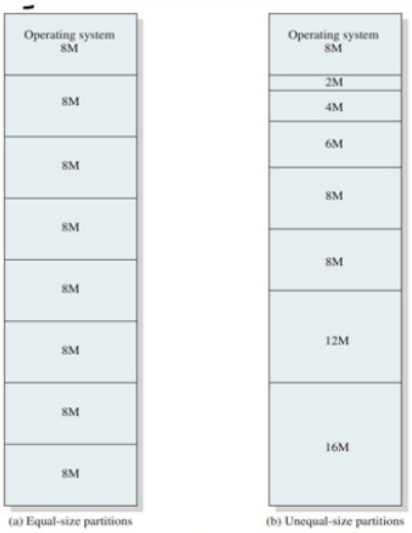
مکانیزم ۲) Multiprogramming بدون محافظت: static partition memory managment

حافظه را به یکسری پارتیشن با سایزهای یکسان یا مختلف و ثابت تقسیم می‌کنیم و سربار هنگ اجرای برنامه‌ها قابل تغییر نیست. چون سایز برنامه‌ها با هم متفاوت است و نیازهای مختلفی دارند می‌توانیم سایزهای مختلفی برای پارتیشن‌ها ایجاد کنیم
برای افزایش بهره وری cpu و Device‌ها میایم چندتا فرآیند میذاریم توی حافظه که اگر یک فرآیند برای I/O بلاک شد یا تمام شد بتونیم سریع سوئیچ کنیم روی یک فرآیند دیگر.

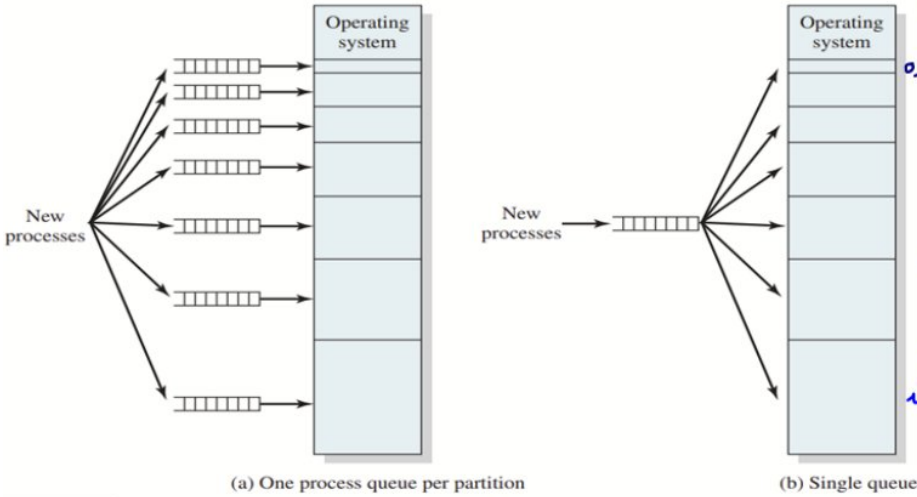
نکات:

۱- هر فرآیند یک آدرس Base و یک آدرس کران (Bound یا limit دارد)
۲- ★ آدرس‌هایی که موقع اجرا فرآیند توسط cpu تولید می‌شود آدرس‌های فیزیکی (واقعی) است و Binding در زمان load انجام می‌شود ← چون میدانیم هر برنامه دقیقا کجای حافظه فیزیکی لود میشه.
۳- در این جا هنوز محافظت نداریم.
۴- درجه چند برنامگی به تعداد پارتیشن‌های ماست.
۵- در اینجا نیز مشکل Internal Fragmentation داریم.
۶- اندازه بزرگترین برنامه قابل اجرا به اندازه، اندازه بزرگترین پارتیشن است. مگر از تکنیک overlay استفاده کنیم که بعدا توضیح میدم.
۷- هر فرآیند باید بصورت پشت سرهم در حافظه قرار گیرد

وقتی اندازه پارتیشن‌ها یکسان باشه ۲ تا شکل داریم:
۱- بهره وری RAM بسیار پایین است و هر برنامه‌ای هر چقدر هم کوچک باشد یک پارتیشن کامل را اشغال می‌کند
۲- یک برنامه ممکن است بزرگتر از پارتیشن‌ها باشد



Example of Fixed Partitioning of a 64-Mbyte Memory



Memory Assignment for Fixed Partitioning

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

و در یک پارتیشن قرار نگیرد که در اینصورت برنامه نویس باید از تکنیک overlay استفاده کند و در اینصورت تنها بخشی از برنامه در هر زمان در حافظه است.

هر دوی این مشکلات را می‌توان با استفاده از پارتیشن بندی با اندازه نابرابر کاهش داد اگرچه کامل حل نمی‌شوند

بطور کلی در تکه تکه شدن داخلی:

حتی اگر فرآیندی در آن حفره جا نشود نمی‌توان از این حفره‌ها استفاده کرد، اما در External Fragmentation در صورتی حفره‌ها غیر قابل استفاده اند و هدر می‌روند که هیچ فرآیند دیگری در آن حفره‌ها جا نگیرد.

placement Algorithm

۱- وقتی اندازه پارتیشن‌ها برابر باشد: اینکه فرآیندها در کدام پارتیشن قرار بگیرند مهم نیست و تا زمانیکه یک پارتیشن خالی باشد می‌تواند یک فرآیند جدید در آنها قرار دارد و اگر همه پارتیشن‌ها با فرآیندهایی اشغال شده باشند که آماده اجرا نیستند ولی هم در حال اجرا نباشد باید یکی از این فرآیندها از حافظه swap out شود تا فضا برای یک فرآیند جدید باز شود

۲- وقتی اندازه پارتیشن‌ها برابر نباشند: در این حالت ۲ راه برای تخصیص فرآیندها به پارتیشن‌ها وجود دارد.

۱- هر فرآیند را به کوچکترین پارتیشن که در آن قرار می‌گیرد اختصاص بدهیم که برای این منظور یک صف زمانبندی برای هر صف نیاز داریم تا فرآیندهای swapped-out شده برای آن پارتیشن را نگه دارد. مزیت این روش این است که فرآیندها به گونه‌ای تخصیص پیدا میکنند که حافظه هدر رفته در هر پارتیشن یعنی internal fragmentation به حداقل می‌رسد.

این روش اگرچه برای یک پارتیشن تکی بهینه به نظر می‌رسد اما در کل بهینه نیست! و رویکرد ترجیحی بصورت زیر است:

۲- استفاده از یک صف برای همه فرآیندها: وقتی زمان لود یک فرآیند به حافظه رسید کوچکترین پارتیشن که این فرآیند در آن جایش انتخاب میشد و اگر همه پارتیشن‌ها اشغال شده باشند باید تصمیم swapping گرفته شود، حالا باید تصمیم بگیریم کی را بکشیم بیرون.

استفاده از این روش درجه‌ای از انعطاف پذیری را برای روش پارتیشن بندی ثابت به وجود می‌آورد و همین طور روش پارتیشن بندی ثابت ساده است و به حداقل نرم افزار os و سربار پردازشی نیاز دارد ولی عایبی از جمله محدود کردن تعداد فرآیندهای فعال، استفاده غیر بهینه فضای پارتیشن توسط فرآیندهای کوچک.

در محیطی که نیاز حافظه همه کارها از قبل مشخص باشد می‌توان از این روش استفاده کرد اما در اکثر موارد این روش ناکارآمد است.

مکانیزم۳) multiprograming با محافظت

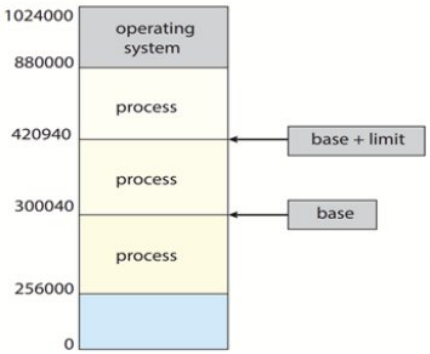
با پیشرفت‌هایی که در پردازنده‌ها به وجود آمد، مقدار Base و (limit)Bound هر فرآیند موقع cs هر فرآیند در حافظه توسط os در ثبات‌هایی که به همین منظور در پردازنده قرار داشت مقداردهی می‌شد، در این حالت هر آدرس فیزیکی (واقعی) که توسط یک فرآیند تولید می‌شد توسط سخت افزار با Base+limit, Base آن فرآیند چک می‌شد و اگر خارج از این محدوده بود پردازنده یک trap به os ایجاد می‌کرد.

توجه کنید که در اینجا هنوز آدرس‌هایی که فرآیندها موقع اجرا در cpu تولید می‌کنند آدرس‌های فیزیکی واقعی است و Binding در زمان load انجام می‌شود ← چون میدانیم هر برنامه دقیقا کجای حافظه فیزیکی لود میشد.

در ساده‌ترین حالت منظور از مدیریت حافظه این هست که وقتی تعدادی برنامه در حافظه وجود دارد، یک برنامه فقط بتواند به حدود مجاز متعلق به خودش دسترسی پیدا کند.

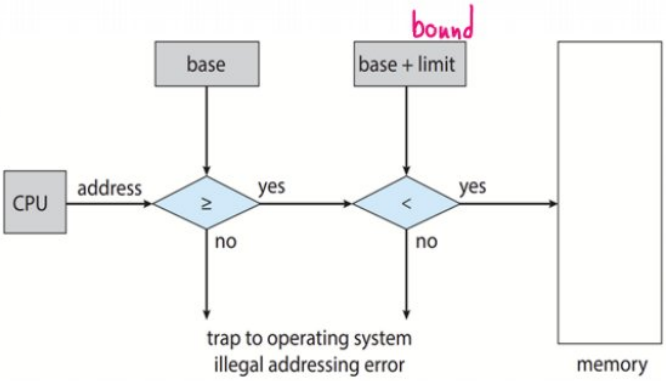
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



از ابتدایی ترین سیستم هایی که وجود داشته به اینصورت بوده که برای کل سیستم دو تا رجیستر Base و limit در نظر می گرفتن و توی مرحله context switch وقتی یک فرآیند قراره داده بشه به cpu، ثبات Base به آدرس شروع حافظه مربوط به این فرآیند مقداردهی می شد و ثبات limit هم به طول حافظه متعلق به این فرآیند مقداردهی می شد و به اینصورت ابتدای حافظه مجاز این فرآیند برابر Base و انتهای حافظه مجاز آن برابر Base+limit بود.

حالا سخت افزار هر یک از آدرس های مربوط به اجراشون یک فرآیند که از cpu خارج می شد را بصورت رو به رو با base و limit مربوط به آن فرآیند مقایسه می کرد.

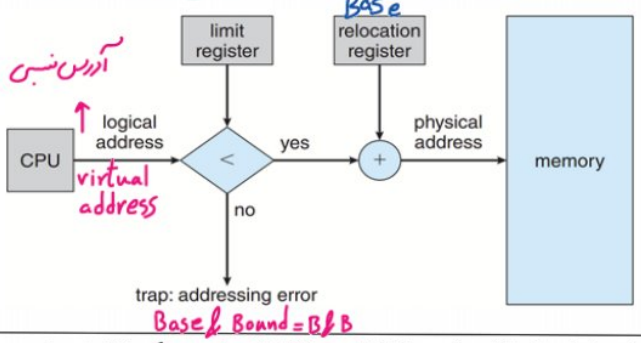


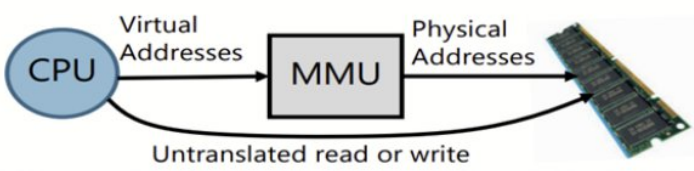
اگر آدرس تولید شده از base کوچکتر باشد یک trap به os اتفاق می افتد ولی اگر از base بزرگتر باشد حالا چک می شد که از base+limit کوچکتر باشد که در اینصورت این آدرس در ثبات MAR قرار می گرفت و می رفتیم و از آن آدرس می خواندیم یا می نوشتیم. این آدرس، آدرس فیزیکی است.

مکانیزم ۴) (روش کران (پایه) Base & Bound) Contiguous Memory Allocation (تخصیص حافظه پیوسته)
Dynamic partitioning: با مکانیزم محافظت و واحد ترجمه آدرس مجازی به آدرس فیزیکی Multiprograming

در مکانیزم ۲ و ۳ این شکل را داریم که os (loader) باید موقعی که میخواد برنامه ای رو توی حافظه لود بکنه باید کدهای باینری را تغییر بده و آدرس ها را به آدرس فیزیکی تبدیل بکنه، این تبدیل لود کردن فرآیند ها را دچار کندی می کنه، در اینجا ما Binding را هنگام اجرای برنامه ها انجام میدیم و بنابراین آدرس که پردازنده تولید میکنه یک آدرس مجازی است، در این روش سیستم به واحد MMU مجهز می شود در اینجا برنامه نویس می تواند در برنامه اش از آدرس ۰ تا یک Max ای که os برای برنامه ها مشخص می شود استفاده کنه و برنامه بصورت یک کد relocatable کامپایل می شود و در ع رد قرار می گیرد و وقتی loader می خواهد این کد را درون حافظه قرار بده، بدون دست زدن به کد باینری، آن را درون حافظه قرار میدارد و Base و limit آن را نیز در pcb این فرآیند قرار میدهد، حالا موقع cs وقتی os میخواهد این فرآیند را روی cpu قرار بدهد، Base و limit آن را در ثبات های Base و limit پردازنده قرار میدهد. حالا واحد MMU در زمان اجرای یک فرآیند، آدرس های تولید شده توسط فرآیند (آدرس های مجازی) را به آدرس فیزیکی ترجمه می کند، تا قبل از این روش آدرس تولید شده توسط فرآیند ترجمه نمی شد. در اینجا ترجمه بسیار راحت است و آدرس خارج شده از cpu اگر از limit کمتر باشد با Base آن فرآیند جمع می شود و به آدرس فیزیکی تبدیل می شود و روی باس آدرس حافظه قرار می گیرد

★ انعطاف پذیری افزوده شد چون الان به برنامه ها اجازه میدیم هر آدرسی که دوست دارند کامپایل بشوند و موقع لود کردن کدها هم به باینری کدها دست نمی زنیم و هر چی بصورت transparent توسط MMU هندل میشه





سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

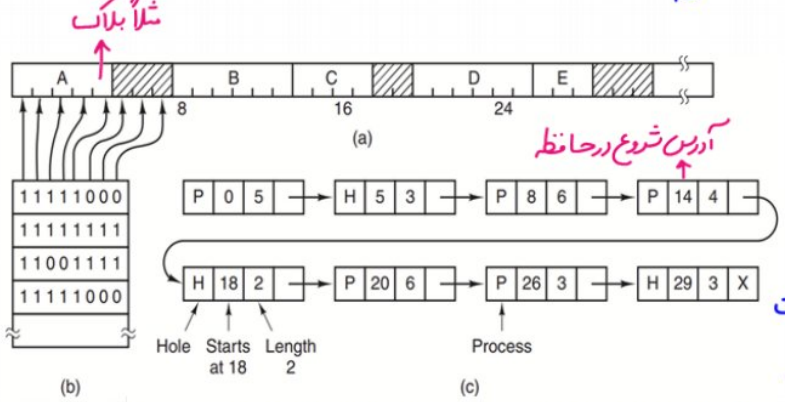
Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) در اینجا محافظت هم بیشتر می‌شود، چون آدرس‌هایی که فرآیندها تولید می‌کنند مجازی هستند و توسط MMU ترجمه می‌شوند و بنابراین اگر حتی یک فرآیند آدرس یک فرآیند دیگر را بداند و بخواهد آدرس اش را در برنامه خودش تولید کند نمی‌تواند و فقط می‌تواند آدرس‌های محدوده خودش را تولید کند. پس محافظت بین فرآیندها با هم و محافظت بین فرآیندها با OS هم وجود دارد

توجه) این محافظت توسط ۲ ثبات Base و limit انجام میشه و اگر آدرسی در محدوده خودش نباشد یک trap به OS اتفاق می‌افتد.

Managing free memory

یک فرآیند وارد شده و می‌خوایم اون رو ببریم توی حافظه، چطور متوجه شویم کجاهای حافظه حفره است و کجاها پره و اندازه حفره‌ها چقدر است تا بتوانیم یک حفره مناسب فرآیندمان پیدا کنیم.



(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

۱- Bitmaps : هر بلاک به ۱ بیت نگاشت می‌شود

۲- Free lists

۱- Bitmaps : اینکه اندازه هر بلاک یا واحد تخصیص چقدر باشد مهم است اگر هر بلاک 1KB باشد اندازه فرآیند A بزرگتر از 4KB و کوچکتر از 0KB است پس توی بلاک آخر internal fragmentation داریم، بطور میانگین به ازای هر فرآیند ما نصف یک بلاک فضای هدر رفته داریم.

- اگر بلاک بزرگ باشد : هدر رفت زیاد میشه ولی Bitmap کوچیک میشه.
- " " کوچک " : " " کم " " " بزرگ " . اگر اندازه بلاک را خیلی کوچیک کنیم مثلاً ۱۶ بیت، در این روش به ازای هر ۱۶ بیت داریم ۱ بیت برای Bitmap نیاز داریم، پس هر چه بلاک کوچکتر بشه سرباره بیشتر به چشم میاد.

مشکل Bitmaps کند بودن پیدا کردن ۰ های متوالی به اندازه تعداد بلاکی است که فرآیندی که می‌خواهیم در حافظه قرار دهیم است.

Free lists:

سوال: اگر فرآیند E از حافظه خارج بشه چه کاری انجام میدهیم؟ باید آن را با حفره بعدی اش ادغام اش کنیم و میشه H | ۲۶ | ۶ →

" " D " " " " " " " " ? " " " قبلی " " " ⇐ پس باید وقتی یکی خارج میشه

به هر ۲ نفر سمت چپ و راست اش نگاه کند، خب اشاره به بعدی هر کی رو داریم ولی قبلی شو نداریم، برای پیدا کردن قبلی یا باید هر بار لیست را از اول پیمایش کنیم که وقت گیره پس بهتر است که لیست پیوندی ما ۲ طرفه باشد

توجه) " یک فرآیند ۲ طرف اش حفره باشه با حفره‌های دو طرف اش ترکیب میشه

نکته) استفاده از free lists بهتر از Bitmaps است چون در Bitmaps نیاز به شمردن تعداد ۰ های متوالی را برای پیدا کردن حفره مناسب داشتیم

روش جایابی در این روش: Contiguous Memory Allocation (تخصیص حافظه پیوسته)

حافظه اصلی باید هم OS و هم فرآیندهای کاربران را در خود جای دهد بنابراین باید RAM را به کاراترین نحو به فرآیندها اختصاص بدهیم. در اینجا می‌خوایم یک روش ابتدایی تخصیص حافظه پیوسته را توضیح دهیم. حافظه معمولا به ۲ بخش تقسیم می‌شود یکی برای OS و یکی برای فرآیندهای کاربر، می‌شود OS را در آدرس‌های پایین یا بالای حافظه قرار داد که به عوامل زیادی مثل محل قرارگیری بردار وقفه بستگی دارد

نکته) بسیاری از OS مثل ویندوز و لینوکس OS را در بالای حافظه قرار میدهند.

یکی از ساده‌ترین روش‌های تخصیص حافظه تخصیص فرآیندها به پارتیشن‌هایی با اندازه متغیر در حافظه است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

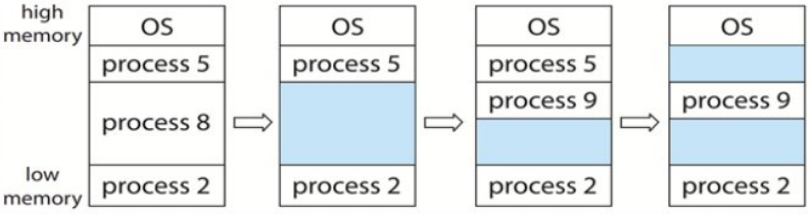
Memory Manager که یک جزئی از OS است باید آگاه باشد که کجاهای حافظه آزاد است که به این‌ها memory holes گفته می‌شود و کدام قسمت‌ها استفاده شده و سپس با توجه به تقاضاهایی که وجود دارد باید تصمیم بگیرد که چه memory hole‌ای را به چه تقاضایی تخصیص دهد. همچنین اگر یک فرآیندی تمام می‌شود یا می‌خواهد یک حافظه را به OS پس بدهد چطور می‌توان آن حافظه‌ها را از فرآیندها گرفت و جدا به فرآیندهای دیگر داد.

در این روش در ابتدا تمام حافظه (به جز فضایی که OS می‌گیرد) برای قرارگیری فرآیندهای کاربر در دسترس است و در ابتدا تمام این حافظه به‌عنوان یک memory hole در نظر گرفته می‌شود. این memory hole‌ها بصورت یک لیست پیوندی در هسته OS ذخیره می‌شوند.

وقتی فرآیندی وارد می‌شود و به حافظه نیاز دارد، OS لیست memory hole را برای پیدا کردن حفره‌ای که این فرآیند جایش جستجو میکند و آن فرآیند را به حفره مناسبی که پیدا می‌کند تخصیص میدهد و اگر فضایی از آن حافظه باقی بماند آن را به مجموعه hole‌ها اضافه میکند، وقتی هم فرآیندی تمام می‌شود حافظه‌اش آزاد می‌شود و این حافظه به مجموعه hole‌ها اضافه می‌شود، حالا اگر hole‌های جدیدی که به وجود می‌آیند در مجاورت hole‌های دیگر باشند با آنها ترکیب می‌شوند تا hole بزرگتری به وجود بیاید.

اگر یک فرآیندی از راه برسد و حافظه‌ای اندازه آن وجود نداشته باشد یک راه ساده این است که فرآیند را قبول نکنیم و یک پیام خطا مناسب ارائه دهیم، یک راه دیگر این است که چنین فرآیندهایی را در یک صف قرار دهیم و وقتی حافظه آزاد می‌شود OS این صف را بررسی کند

انواع الگوریتم‌های مکان یابی



۱- first_fit (اولین برازش): از اول لیست پیوندی (از ابتدای حافظه) شروع به جستجو می‌کنیم و فرآیند را در اولین حفره‌ای که جا می‌شود میذاریم.

نکته) همیشه نیاز نیست کل لیست جستجو شود بنابراین worst case اش از O(n) است ولی زمان متوسط اش می‌تواند O(n/2) باشد

۲- Best_fit (بهترین برازش): در لیست حفره‌ها باید سرچ کنیم و حفره‌ای به فرآیند اختصاص بدهیم که کمترین دور ریز (کمترین حافظه هدر رفتن) (حفره‌ای انتخاب میشه که کمترین فاصله بین سایز درخواست شده توسط فرآیند با آن حفره وجود دارد)

مرتبه همیشه O(n) است

۳- worste_fit (بدترین برازش): حفره‌ای را اختصاص میدهیم که بیشترین دور ریز را داشته باشیم به امید آنکه ما بتوانیم از فضای memory hole باقیمانده برای فرآیندهای دیگر هم استفاده کنیم. بدترین برازش میگه کاری کن که بهترین برازش میکنه اینه که توی حافظه یک عالم فضاهای کوچیک دور ریز به وجود می‌آورد که دیگه به درد هیچ فرآیندی نمی‌خورد.

این روش بدترین کارایی را دارد، همین طور با این کار حفره‌های بزرگ رو از دست میدهیم و بعدا فرآیندهای بزرگتر اصلا جا ندارند.

۴- Next_fit: مثل first_fit است ولی مثل first_fit همیشه از ابتدای حافظه شروع نمی‌کند و جستجو را از آخرین جایی که دفعه قبل اختصاص انجام شده ادامه میدهد و به اینصورت داریم بصورت چرخشی به فرآیندها حافظه میدهیم و هم فشار به ابتدای حافظه نمی‌آید

نکته) مثلا در خصوص حافظه‌های flash که ما ترجیح میدهیم نوشتن‌ها بین همه بلاک‌ها پخش بشه تا یک عده از بلاک‌ها مستعمل نشوند Next_fit می‌تواند مناسب باشد

مثال) فرض کنید لیست حفره‌ها از ابتدای حافظه بصورت زیر باشد، پس فرض کنید P_1= ۲۷KB , P_2= ۱۳KB , P_3= ۱۰ KB

لیست حفره‌ها: ۱۵KB → ۲۵KB → ۱۳KB → ۶۵KB → ۴۵KB → ۱۱KB

چهار روش بالا را بررسی کنید

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۱) first fit

۱۵KB → ۲۵KB → ۱۳KB → ۶۵KB → ۴۵KB → ۱۱KB
↓۱۳KB ↓۲۴KB ↓۱۰K
۲KB ۱KB ۳KB

۲) Best fit

۱۵KB → ۲۵KB → ۱۳KB → ۶۵KB → ۴۵KB → ۱۱KB
↓۲۴KB ↓۱۳KB ↓۱۰KB
۱KB ۰KB ۱KB

۳) worst fit

۱۵KB → ۲۵KB → ۱۳KB → ۶۵KB → ۴۵KB → ۱۱KB
↓۲۴KB ↓۱۳KB
۴۱KB ۳۲KB
↓۱۰KB
۳۱KB

۴) Next fit فرض کنید دفعه قبلی وجود ندارد و از ابتدا شروع کنید
اگر بعد از P_1 یک فرآیند ۱KB بیاید این روش آن را کجا میگذارد؟

۱۵KB → ۲۵KB → ۱۳KB → ۶۵KB → ۴۵KB → ۱۱KB
↓۲۴KB ↓۱۳KB ↓۱۰KB
۱KB ۰KB ۵۵KB

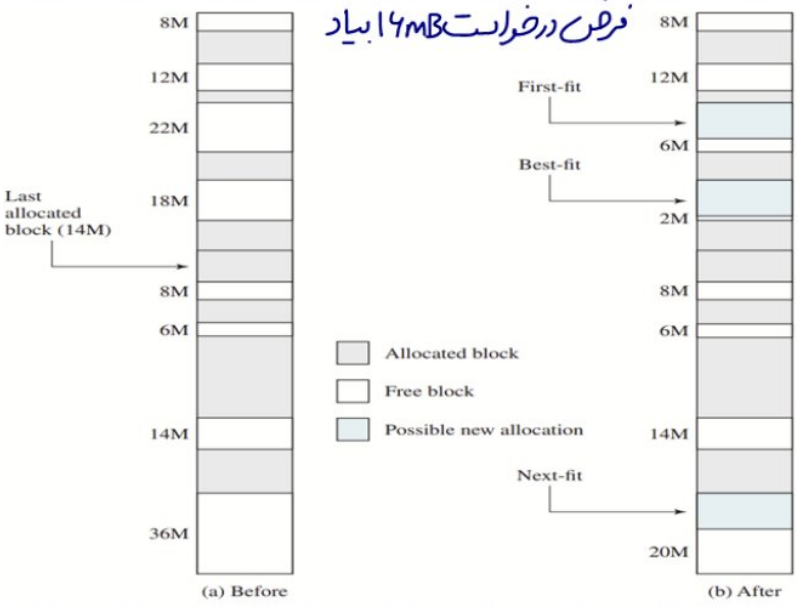
الان اگر یک فرآیند ۶۰KB فقط first fit و Best fit براش جا دارند.

نکته) first fit از Best fit سریعتر است و کارایی مناسبی هم دارد و Best fit نیز از first fit کاراتر است، worst fit هم کند است و هم بدترین کارایی را دارد.

۵_ quick fit: این روش شبیه Best fit است ولی برای اینکه OS بهترین حفره را سریعتر پیدا کند، OS حفره‌ها را بر اساس اندازه شون دسته‌بندی میکند بطوریکه هر دسته شامل یک رنجی از حفره‌ها با اندازه مشخص است. فرض کنید آمار حفره‌های خالی توسط ساختمان داده‌ای بصورت زیر نگهداری می‌شود

اشاره‌گر به لیست حفره‌های از ۰ تا ۴KB → ۱ → ۳ → ۲ → ...
" " " " " ۴KB تا ۱۶KB → ۱۵ → ۵ → ۷ → ...
" " " " " ۱۶KB تا ۳۲KB → ۲۰ → ۳۰ → ۱۸ → ...

وقتی یک فرآیند از حافظه خارج میشه باید لیست حفره‌ها بروزرسانی بشه، حالا OS باید ببیند چپ و راست حفره‌ای که تازه به وجود اومده حفره دیگه‌ای هست یا نه که اگر مجاورش باشد با هم ترکیب کند

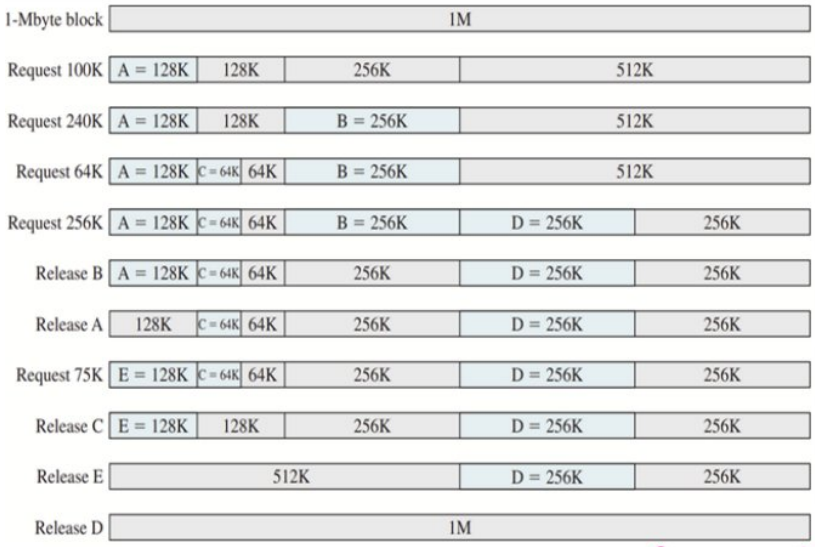


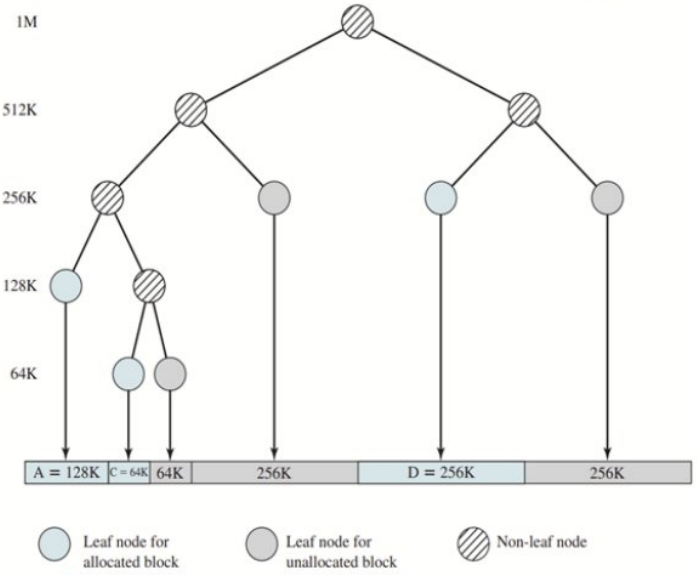
حالا درسته که توی این روش OS سریع‌تر میتوند حفره مناسب یک فرآیند رو پیدا کند ولی این بروزرسانی آخرش ممکنه بیشتر طول بکشه، فرض کنید یک فرآیند ۳۰KB تمام شده و فضاش آزاد میشه و توی حافظه پایین اش یک حفره‌ی ۵KB و بالاش یک حفره ۱۲۰KB ای وجود دارد، حالا ما باید بگردیم توی این لیست‌ها و حفره‌های مجاور فضای خالی شده جدید را پیدا کنیم و بعد آنها را از لیست‌هایشان خارج کنیم و پس لیست‌هاشون رو آپدیت کنیم، بعد این حفره‌ها را با هم ترکیب کنیم و باز بریم توی لیست مربوط به خودش بذاریم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

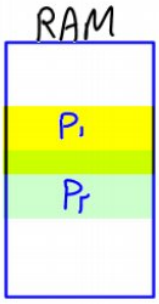
Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

Buddy system : پارتیشن بندی ثابت و پویا دارای مشکلاتی هستند، پارتیشن بندی ثابت تعداد فرآیندهای فعال را محدود می‌کند و از فضا بصورت ناکارآمد استفاده می‌کند، در پارتیشن بندی پویا مدیریت فضای حافظه پیچیده است و تکه تکه شدن خارجی داریم و عمل compaction سربار زیادی دارد، این روش یک مصالحه برای غلبه بر معایب این ۲ است اما در OSهای جدید حافظه‌ی مجازی مبتنی بر صفحه بندی و segmentation بهتره





مشکلات مکانیزم B&B



۱- در این روش تعریف حافظه مشترک بین فرآیندها بسیار سخت است. گاهی اوقات نیاز داشتیم بخشی از حافظه ۲ فرآیند برای همدیگر قابل دسترسی باشد.

در روش B&B تنها راه برای ایجاد حافظه مشترک بین ۲ فرآیند این است که ۲ فرآیند در حافظه پشت سر هم باشند و Base فرآیند دوم از limit فرآیند اول کمی کوچکتر باشد تا به اندازه limit(P_1) - Base(P_2) حافظه مشترک داشته باشیم.

حالا اگر حافظه به نحوی باشد که نتونیم دو فرآیند را پشت سر هم بذاریم نمی‌توانیم بین آن دو حافظه مشترک داشته باشیم، یا اینکه OS باید فرآیندها را به نحوی جابجا کند که بتواند ۲ فرآیندی که می‌خواهند با هم ارتباط برقرار کنند را در کنار هم بگذارد.

۲- در تمامی مکانیزم‌هایی که تا الان داشتیم از جمله این روش حافظه فرآیند باید پشت سر هم و پیوسته باشد.

مشکلات تخصیص و بازپس‌گیری حافظه (۳ External and Internal Fragmentation

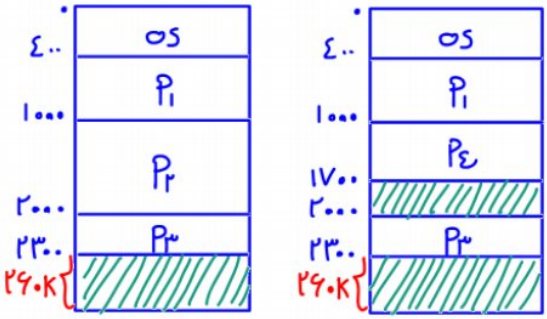
فرض کنید حافظه‌ای با سایز ۲۵۶۰KB در اختیار داریم که از آدرس ۰ تا ۴۰۰KB به OS تخصیص داده شدند، فرض کنید الگوریتم زمانبندی OS، RR با کوانتوم ۱ است، فرض کنید ۵ فرآیند با فضاها و زمانهای اجرای زیر می‌خواهند وارد سیستم شوند و این فرآیندها پس از سپری شدن زمانشان حافظه‌شان را پس میدهند. فرآیندها به ترتیب شماره‌شون وارد می‌شوند

P_1 : 600KB, 10ms
P_2 : 1000KB, 5ms
P_3 : 300KB, 20ms
P_4 : 700KB, 8ms
P_5 : 500KB, 15ms

- ۱۲۶۰KB از که باقی می‌ماند نه می‌تواند پاسخ‌گوی P_4 و نه P_5 باشد پس P_4 و P_5 باید منتظر بشوند تا فضای خالی به وجود بیاید
- P_2 زودتر از P_1 و P_3 در لحظه ۱۴ms سیستم را ترک می‌کند
- حالا ما در مجموع 560KB = 300 + 260 آزاد داریم که می‌تواند پاسخ‌گوی P_5 باشد ولی چون حافظه ما تکه تکه شده در کنار هم نیست نمی‌توانیم این کار را انجام بدهیم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir



external fragmentation: حافظه آزاد به اندازه کافی داریم ولی چون تکه تکه شده و چون در این روش فرآیندها مجبورند در حافظه بصورت پیوسته لود بشوند نمی‌توانیم حافظه را اختصاص بدهیم. مجموع فضاهای آزاد از نیاز فرآیند جدید بیشتره اما چون پشت سر هم نیستند نمی‌توانیم به فرآیند اختصاص بدهیم.

compaction: قسمت‌های آزاد حافظه را بهم متصل کنیم.

در این شال شلاقی توانیم ۳۰۰KB مربوط به P_4 که بین آدرس‌های ۲۰۰۰ تا ۲۳۰۰ است را به آدرس ۱۷۰۰ منتقل کنیم.

فرض کنید زمان خواندن و نوشتن در حافظه ۴۰ns باشد، این کار چقدر زمان می‌برد؟

باید ۳۰۰KB را ابتدا بخوانیم و پس این ۳۰۰KB را بنویسیم: $\text{compaction time} = 600KB \times 40ns = 24000 \times 10^{3} \times 10^{-9} = 24ms$

خب اگر این ۲۴ ms را به اجرا فرآیندها اختصاص میدادیم هم P_1 تمام می‌شد و فضا آزاد می‌شد و هم از این زمان برای اجرا فرآیندها استفاده کردیم و فرآیندها رفتند جلو و زمان هدر نمی‌شد، این تصمیمات را باید memory manager بگیرد.

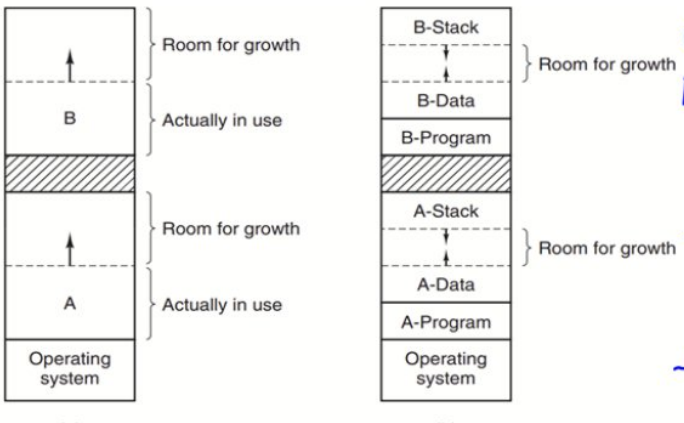
نکته) عمل compaction فقط در مورد کدهایی قابل انجام است که relocatable باشند، یعنی کدهایی که Binding شون مربوط به زمان اجرا باشد که ما بتوانیم در زمان اجرا آدرس‌هایی که یک فرآیند نیاز دارد را عوض کنیم.

Internal fragmention:

فرض کنید یک memory hole با سایز ۱۰۰۰۰B داریم و یک درخواست حافظه نیز با سایز ۹۹۹۸B وجود داشته باشد، حالا اگر ما این hole را به این فرآیند بدیم و ۲ بایت آزاد حافظه را به قسمت‌های آزاد حافظه اضافه کنیم، با این ۲ بایت کار هیچ فرآیندی راه نمی‌افتد و بنابراین نمی‌ترنیم این ۲ بایت را به کسی تخصیص بدهیم ولی هر زمانی که میخوایم برای یک فرآیندی که حافظه میخواد لیست حفره‌ها را جستجو کنیم این ۲ بایت هم مورد بررسی قرار می‌گیرد و بنابراین این ۲ بایت هم سربار جستجو تحمیل می‌کند و هم بلااستفاده می‌ماند.

اما یک allocation policy خوب می‌توانست به این صورت باشد که تمام ۱۰۰۰۰B را به فرآیند اختصاص بدهد و این ۲ بایت را در فرآیندی که این حافظه روبهش دادیم هدر میدادیم که در اینصورت از واژه internal fragmentation استفاده می‌کنیم و در نتیجه سربار ناشی از ایجاد حفره ۲ بایتی به memory manager تحمیل نمی‌شود.

Room for growth یا Gap را با hole اشتباه نگیرید



در فضایی که به فرآیندها اختصاص میدیم یک فضای خالی بیشتر اختصاص میدیم برای رشد پشته یا پشته‌های یک فرآیند اگر چندتری باشد و همین طور برای قسمت heap که می‌توانست بصورت داینامیک رشد کند. دقت کنید ما تا قبل از اجرای فرآیندها نمی‌دانیم که اندازه این‌ها چقدر است و مثلاً نمی‌دانیم چندتا تابع تو در تو فراخوانی می‌کند فرآیند ما، برای همین معمولا برای استک و هیپ یک مقدار پیش فرض برای هر فرآیندها در نظر گرفته می‌شود و بین Heap و استک یک فضای خالی داریم که شاید فرآیندها از همه این فضای خالی استفاده نکنند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

قبل از شروع مکانیزم جدی چند تا مفهوم را ببینیم

Dynamic loading

برای استفاده بهتر از فضای حافظه ممکن است تا زمانیکه یک routine رو نیاز نداریم آن را لود نکنیم، بنابراین اندازه یک فرآیند به اندازه سایز حافظه نیز یکی محدود نمی‌شود و اینگونه نیست که اگر اندازه یک فرآیند از اندازه RAM بزرگتر باشد دیگر نتوانیم آن را اجرا بکنیم و خیلی وقت‌ها ما با لود کردن بخشی از یک فرآیند امکان اجرا آن فرآیند را داریم.

قانون نود و ده (یا هشتاد، بیست): این قانون میگه معمولا در ۹۰ درصد مواقع فقط شغل اجرا کردن ۱۰ درصد از کد برنامه‌مون هستیم

نکته) Dynamic loading را خود برنامه‌نویس می‌تواند انجام بدهد و کاری کند که ابتدا فقط بخشی از برنامه‌اش که نیاز است load بشود و اگر قسمت‌های دیگری از برنامه‌اش را جدا نیاز دارد همون موقع از OS حافظه بگیرد و آن قسمت‌ها را لود کند، پس این روش لزوما به پشتیبانی OS نیاز ندارد

overlay Technique

تکنیک overlay بدون استفاده از OS و با یک مکانیزم مدیریت حافظه ضعیف همین کاری که در بالا توضیح دادم را انجام میداده است.

یعنی ما یک کدی می‌نویسیم به نام overlay که این کد میاد این کار رو برامون انجام میده.

نکته) اسمبلرهای قدیمی برای اسمبل کردن یک قطعه کد اسمبلی و همین طور کامپایلرهای قدیمی برای کامپایل یک کد سطح بالا به زبان ماشین از همین تکنیک overlay استفاده می‌کردند و برای اینکه فضای حافظه کمتری اشغال کنند عملیات شون روی چند تا فاز انجام میدادند.

مثال) یک اسمبلر را در نظر بگیرید که اسمبل کردن یک کد اسمبلی را طی ۲ مرحله (فاز) زیر انجام میدهد، اگر برنامه overlay ای که این عملیات را انجام میدهد ۱۰KB فضا بگیرد، با توجه به فرضیات زیر چقدر حافظه برای این اسمبلر نیاز است؟

pass1: کد برنامه مد نظر را می‌خواند و اطلاعات متغیرهای آن برنامه از قبیل نوع و آدرس‌هایشان و ... را در جدولی بنام symbol table قرار میدهد، فرض کنید که کدی که اسمبلر برای این فاز نیاز دارد ۷۰KB باشد و اندازه جدول نمادها نیز ۲۰KB باشد

pass2: براساس اطلاعاتی که در pass1 بدست آورده میاد و با کمک یکسری روتین دیگر، کد اسمبلی را به زبان ماشین تبدیل میکند فرض کنید که کدی که اسمبلر برای این فاز نیاز دارد ۸۰KB باشد و اندازه کد روتین‌هایی که استفاده میکند ۳۰KB باشد

پاسخ: توجه کنید که اگر ۲۰۰KB فضا در اختیار نداشته باشیم و بیایم از تکنیک overlay استفاده کنیم با ۱۴۰KB می‌توانیم اسمبلرمان را اجرا کنیم.

۱۵۰KB available

دیگه به کد فاز ۱ نیازنداریم و می‌تونیم آن را از حافظه خارج کنیم

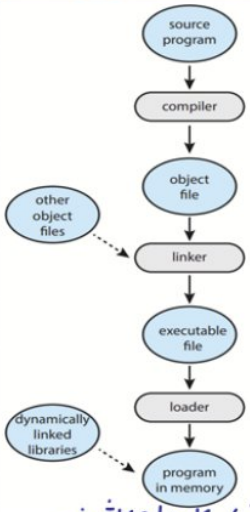
۱۵۰KB available → بعد از بدست آمدن symbol table میتوانیم کد مربوط به pass2 و همین طور روتین‌هایی را وارد حافظه کنیم

Dynamic linKing and shared libraries

یکی از ایده‌های دیگری که می‌تواند در صرفه‌جویی هم حافظه Disk و هم RAM (memory) کمک کند و البته مزیت‌های دیگری هم دارد که بعد از تعریف شان خواهم گفت بحث Dynamic linKing است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



- source file و بعد از کامپایل به object file تبدیل می‌شوند که برای بارگذاری در هر مکانی از حافظه فیزیکی طراحی شده‌اند (relocatable object file)، پس linker این‌ها را در یک فایل باینری واحد قابل اجرا ترکیب می‌کند (single binary executable file). حالا برنامه ما از یکسری کتابخانه هم ممکنه استفاده کنه حالا این کتابخانه‌ها می‌تونند در فاز linker به فایل اجرایی اضافه بشوند که می‌گیم static linking، در مقابل ما Dynamic linking داریم که در این روش کتابخانه‌ها بصورت مشروط به برنامه link می‌شوند و اگر در طول اجرای برنامه نیاز باشند در حافظه اصلی لود می‌شوند، یکی از مزیت‌های این روش این است که از link کردن و لود کردن کتابخانه‌هایی که ممکن است در اجرای یک برنامه استفاده نشود جلوگیری می‌کند.
- DLL (Dynamic linked libraries) کتابخانه‌هایی هستند که هنگامی که برنامه‌ها در حال اجرا هستند به آنها link می‌شوند.
- برخی از os‌ها فقط از static linking پشتیبانی می‌کنند

static linking: خلاصه: وقتی ای استفاده بشه وقتی توی برنامه یک فراخوانی از یک لایبرری انجام بدیم کدش کپی می‌شود در کد exe ما

Dynamic " : به جای اینکه کد مربوط به کتابخانه کپی بشه فقط یک stub که فقط یک interface است و جای آن library را میداند و می‌تواند تبادل پارامتر با توابع آن کتابخانه را انجام دهد در کد ما قرار داده می‌شود

نکته) به DLL، shared library نیز می‌گویند و بصورت گسترده در os‌های ویندوز و لینوکس استفاده می‌شود

مزایای Dynamic linking

۱- هم فضای دیسک و هم memory، سیو می‌شود

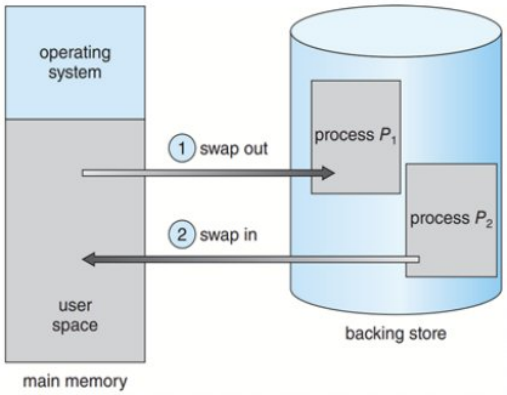
۲- مزیت بعدی DLL‌ها این هست که این کتابخانه‌ها می‌توانند بین چند فرآیند share شوند. یعنی اگر برنامه‌های مختلف از روتین x این لایبرری‌ها استفاده کنند این روتین x فقط کافی است که یکبار در حافظه لود بشه.

۳- library bug: اگر باگی در کتابخانه وجود داشته باشه و آن را حل کنیم، این موضوع روی همه برنامه‌هایی که از این کتابخانه استفاده می‌کنند اعمال می‌شود و نیاز به کامپایل مجدد همه آن برنامه‌ها وجود ندارد

۴- می‌توانیم یک کتابخانه را با نسخه جدید جایگزین کنیم و همه برنامه‌هایی که از آن استفاده می‌کنند به طور اتوماتیک از نسخه جدید استفاده میکنند

نکته) این تکنیک نیاز به پشتیبانی os دارد، چون وقتی فرآیندهای درون حافظه نسبت به هم محافظت می‌شوند، os تنها موجودی است که می‌تواند بررسی کند که یک روتینی از این کتابخانه‌ها که صدا زده می‌شود آیا فضای حافظه یک فرآیند دیگر است که دسترسی به آن غیر مجاز است یا یک فضایی هست که بین چندین فرآیند اجازه دسترسی به آن داده شده است.

swapping (مبادله)



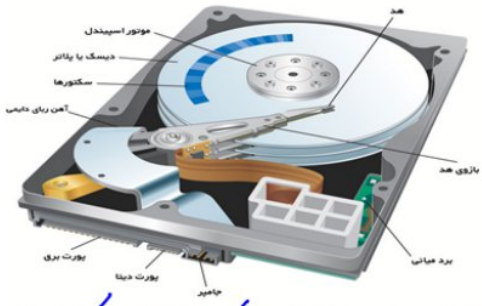
swapping: وقتی فضا در حافظه اصلی کم داریم بخشی از حافظه اصلی (حافظه) را با حافظه دیسک (حافظه ثانویه، حافظه جانبی) جابجا بکنیم.

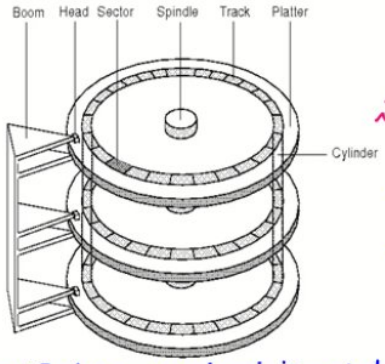
swapping = swap out + swap in

چون دیسک کند است زمان swapping خیلی زیاده و باید سعی کنیم از swapping‌های بی مورد پرهیز کنیم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com





تا فیزهای دیسک :

۱- seeK Time : زمان جستجو (شیار)، زمان جستجوی استوانه
مدت زمانیکه طول میکشه تا بازوی دیسک حرکت کند و هد متصل به جلوی بازو روی trackK (شیار) مد نظر قرار بگیرد

۲- Rotational latency
از بعد از اینکه هد رفت روی شیار مربوط بلاکی که می خوایم اون رو بخونیم یا بنویسیم، استارت تایم رو میزنیم تا زمانیکه ابتدای بلاک بچرخه و بیاد زیر هد. min ام این زمان ۰ است و در حالتی است که تا هد روی ترک مد نظر قرار میگیره، ابتدای بلاکی که کارش داریم دقیقا میفته زیر هد و max اش زمان چرخش یک دور دیسک است.

نکته) اگر در خصوص این مورد چیزی نگفتند میانگین این ۲ زمان یعنی زمان نیم دور چرخش را باید در نظر بگیریم

۳- transfer time : بعد از آنکه ابتدای بلاک زیر هد قرار گرفت چقدر طول میکشه که بلاک خوانده یا نوشته شود.

نکته) علت اصلی کندی زمان دیسک seeK Time هست اش

مثال) فرض کنید سایز حافظه فرآیندهایی که قرار است عمل swapping روی آنها رخ دهد ۱۰MB باشد، با توجه به فرضیات زیر زمان swapping چقدر می شود؟

Average latency and seeK time = ۸ms
Transfer rate to HDD: ۴۰MB/s
user process: ۱۰MB

بخشی از زمان انتقال مربوط به تکنولوژی هست که از طریق آن داده بین هارد دیسک و RAM اتفاق می افتد

$$۲x(۸x۱۰^{-۳}s + \frac{۱۰}{۴۰}s) = ۲x(۸x۱۰^{-۳}s + \frac{۱}{۴}x\frac{۱۰۰۰}{۱۰۰۰}s) = ۲x(۸ms + ۲۵۰ms) = ۲x۲۵۸ms = ۵۱۶ms$$

عوارض swap کردن یک فرآیند وقتی pending I/O دارد

فرض کنید یک فرآیند بواسطه انتظار برای عمل I/O به حالت Blocking رفته و DMA را مسئول انجام عمل I/O مربوط به این فرآیند کردیم حالا اگر در این شرایط این فرآیند را با یک فرآیند دیگه swapping کنیم، ممکن است DMA بیاید و عمل I/O مربوط به فرآیند اول را روی حافظه فرآیند دیگری انجام دهد، در اینصورت هم کار فرآیند قبلی دچار اختلال می شه و هم حافظه فرآیندی که رویش نوشته شده خراب میشه و برای امنیت هم خوب نیست.

راه حل ۱) فرآیندی که pending I/O دارد را هیچ وقت swap out نکنیم و حافظه اش را به فرآیند دیگری اختصاص ندهیم.

راه حل ۲) اجرای عملیات I/O فقط از طریق بافرهای OS صورت بگیرد: سخت افزاری که اطلاع چندانی در مورد فرآیندها ندارد را مسئول عملیات کپی کردن مستقیم به حافظه فرآیند نکنیم و آن سخت افزار را با OS مرتبط کنیم و OS آن داده را از جزء سخت افزاری بگیرد و آن داده ها را از حافظه خودش به حافظه فرآیند کپی کند، در این شرایط چون OS خودش کارهای swapping رو انجام میده، اگر یک فرآیند را swap out کرده و حافظه اش را گرفته و به یکی دیگه اختصاص داده، دیگه حواسش هست که توی حافظه مذکور که الان متعلق به یک فرآیند دیگه هست عمل کپی را انجام بدهد.

نکته) بافر OS محدود است و OS باید این پارامتر رو هم در نظر بگیرد.

چرا در سیستم های موبایل خیلی از swapping استفاده نمی شود

Mobile Device ها عموما از flash memory به جای هارد دیسک های بزرگ استفاده می کنند. حالا چرا طراحان OS های Mobile از

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

swapping پرهیز می‌کند؟

۱- space constraints: در دیوایس‌های موبایل اندازه حافظه‌های فلش نسبت به حافظه اصلی تفاوت فاحشی ندارد و مانند تفاوت بین هارد و حافظه اصلی نیست که انجام swapping به صرفه باشد.

۲- پهنای باند و throughput انتقال داده بین حافظه فلش و حافظه اصلی بالا نیست و بنابراین swapping زمانبرات و بیشتر اوقات انجام swapping به صرفه نیست.

۳- حافظه فلش به اینصورت هستند که وقتی روی یک بلاکی بیش از چند هزار بار (بین هزار تا صد هزار) بنویسیم مستعمل محسوب می‌شوند و دیتای درون بلاکی به این صورت unrelible می‌شود، بنابراین دوست نداریم با swapping زیاد، نوشتن روی حافظه فلش را بالا ببریم

paging and segmentation

این روش‌هایی که تا الان برای تخصیص حافظه توضیح دادیم مکانیزم‌های مدیریت حافظه ساده‌ای هستند و بنابراین سخت افزار پیچیده‌ای هم برای مدیریت حافظه نیاز ندارند، بنابراین در خیلی از سیستم‌های خاص منظوره که نمی‌خواهیم سخت افزار پیچیده‌ای برای مدیریت حافظه اضافه کنیم استفاده می‌شوند ولی در سیستم‌های general purpose که نیاز به استفاده بهینه از حافظه داریم این روش‌ها کارایی لازم را ندارند، برای این سیستم‌ها از ۲ تکنیک زیر استفاده می‌کنیم.

۱- segmentation

۲- paging

تا اینجا حافظه‌ای که به فرآیندها میدادیم یک حافظه پیوسته و پشت سرهم بود ولی توی paging و segmentation کل حافظه‌ای که به فرآیندها میدیم پشت سرهم نیست (توی segmentation تکه‌های هم پیوسته در جاهای مختلف میدهیم)

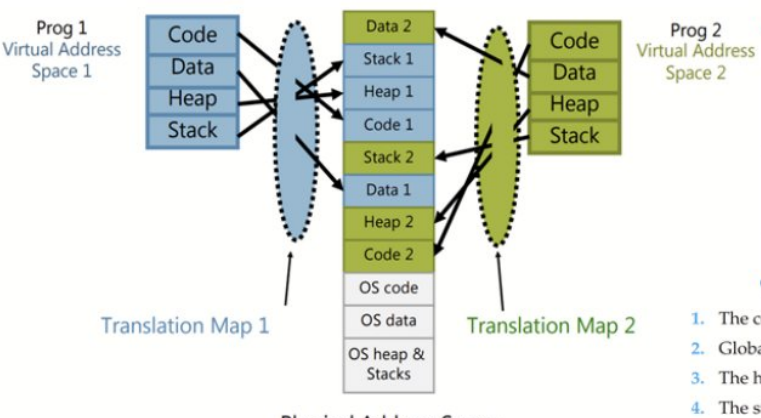
segmentation

مکانیزم ۵)

در روش B & B دیدیم مشکل External Fragmentation جدی است و خیلی وقت‌ها حافظه در دسترس مان بیشتر از درخواست برخی از فرآیندها است ولی نمی‌توانیم بهشون حافظه اختصاص بدیم و نمی‌توانیم درجه multiprogramming را بالا ببریم.

۲ تا مشکل اصلی داشتیم در روش B & B، یکی مسئله تکه تکه شدن خارجی بود و یکی هم ایجاد حافظه مشترک بین فرآیندها سخت بود که هر دوتای این مشکلات از این امر ناشی می‌شد که فضای فرآیندها درون حافظه اصلی باید پیوسته می‌بود، حالا در روش segmentation می‌گیم خب فضای یک فرآیند درون حافظه اصلی پشت سرهم نباشه.

خب گفتیم یک فرآیند شامل بخش‌های جداگانه‌ای مثل کد، پشته، Data، Heap و ... است، حالا این روش اجازه میده این بخش‌ها را در جاهای مختلف حافظه قرار بدهیم ← اما هرکدوم باید پیوسته باشه



- تفاوت بین روش paging و segmentation در نوع نگاه به حافظه است، نگاه paging برای استفاده بهتر از حافظه است و نگاه segmentation بیشتر بدنبال رضایت برنامه نویس است برنامه نویس برایش راحت تر است که به حافظه برنامه خودش بصورت قطعه قطعه نگاه کند.

1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

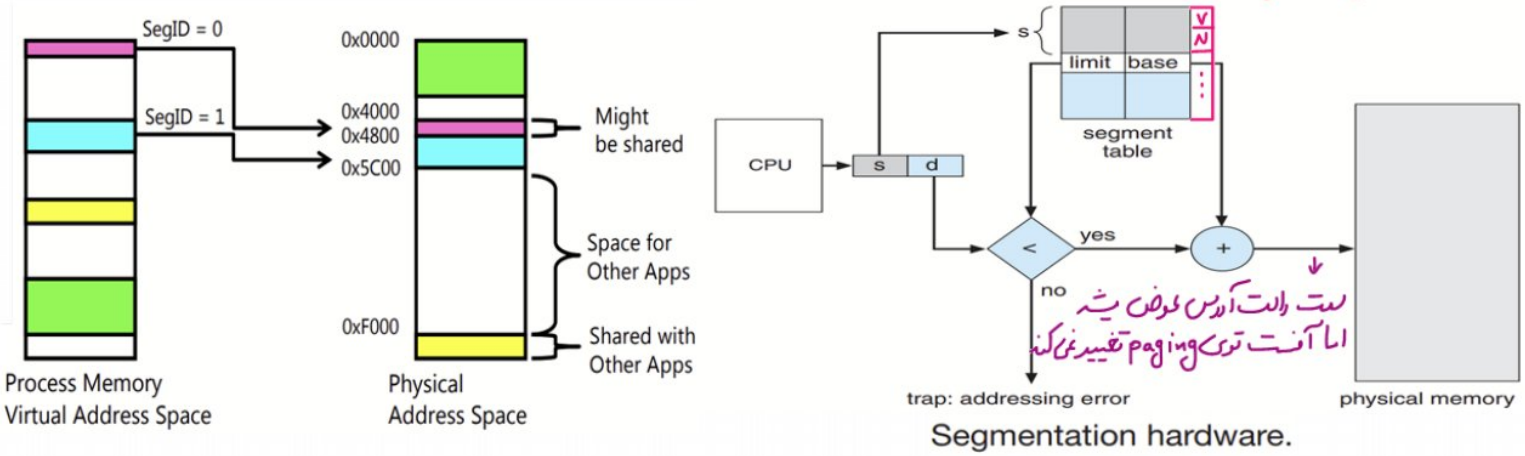
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

پس این روش نگاهش اینه که به جای اینکه کل فرآیند یک حافظه بهم پیوسته داشته باشد هر بخش آن فرآیند یک حافظه بهم پیوسته داشته باشد ← بنابراین می فهمیم در این حالت هر بخش فرآیند باید یک Base و Bound جداگانه داشته باشند

خب قبلا به MMU که مسئول ترجمه آدرس بود برای یک فرآیند یک Base و Bound میدادیم ولی الان برای یک فرآیند باید برای هر بخش اش یک Base و Bound بدیم، پس اینجا دیگه ما یک جدول در MMU داریم و OS برای هر segment برنامه Base و Bound مربوطه اش را هنگام CS در جدول segment table میگذارد.

نکته) با این روش خیلی نمیشه در خصوص تکه تکه شدن داخلی اظهار نظر کرد ولی تکه تکه شدن خارجی را کم می کند چون دیگه برای قرار دادن فرآیند در حافظه نیاز نیست یک قطعه بزرگ حافظه را بصورت پیوسته به فرآیند بدهیم و چون سگمنت ها کوچکتر شدند احتمال اینکه بتوانیم برای آنها جا پیدا کنیم بیشتره و بنابراین flexibility بالاتر میره.

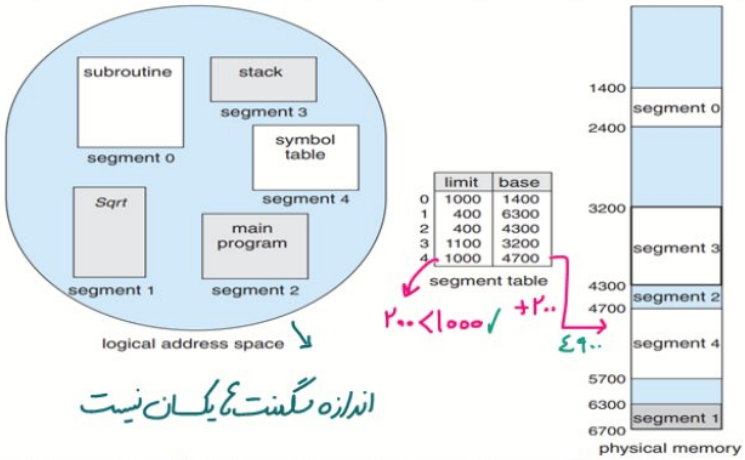


Segmentation hardware.

حالا داستان چیه؟ cpu یک آدرس مجازی ترکیب می کند، حالا باید مشخص باشه که این آدرس برای کدوم segment است و توی اون سگمنت با کدوم offset کار داریم، پس آدرس مجازی در این روش شامل ۲ بخشه: ۱) فیلد شماره سگمنت
پس برای مراجعه به حافظه (تولید آدرس فیزیکی) به این ۲ تا اطلاعات نیاز داریم ۲) فیلد آفست

حالا این آدرس که از cpu میاد بیرون با توجه به s اش میریم توی segment table و می فهمیم که آدرس Base این سگمنت چیه و طول این سگمنت چقدره (limit)، حالا اگر فیلد آفست آدرس مجازی تولید شده (d یا offset) بزرگتر از limit باشد، trap می شه به OS و میگه غیر مجازه و در غیر اینصورت آفست (d) با Base آن سگمنت جمع می شود و آدرس فیزیکی آن مشخص می شود

- فرض کنید می خوایم به آدرس ۲۰۰ سگمنت ۴ مراجعه کنیم



اندازه سگمنت ها یکسان نیست

حالا آدرس مجازی مجبوری ۲ بخشی شد و این آدرس سگمنت و آفست از کجا اومد؟

کامپایلر باید این کار را بکند، در مکانیزم B&B ما قالبی برای تولید آدرس مجازی نداشتیم و کامپایلر میومد و هر جوری دوست داشت آدرس ها را به قسمت های مختلف یک فرآیند میداد، اما اینجا کامپایلر نمیتوند این کار را بکند چون می خوایم بتونیم تشخیص بدهیم آدرس مجازی ای که تولید میشه برای کدوم سگمنت و برای کدوم آفسته اون سگمنت، پس باید یک قالب برای کامپایلر مشخص میکنیم و بهش میگیم باید طبق این قالب آدرس مجازی تولید کنی، بهش میگیم مثلاً قسمت سگمنت ۳ بیت باشد و قسمت آفست ۱۳ بیت باشد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

چرا ۳ تا؟ بررسی کردیم دیدیم فرآیندها حداکثر ۸ تا segment دارند.

پس به کامپایلر می‌گیم هر آدرسی که تولید میکنی ۳ بیت بیا اولش بذار که اون ۳ بیت نشون بده اون آدرس برای کدوم segment است

فرض کنید کامپایلر طبق جدول مقابل بیاد و کامپایل کند، پس کامپایلر میاد وقتی میخواد instruction های قسمت code برنامه ما ایجاد کند میره و برای ۳ بیت اول آدرس هم اینها ۰۰۰ میده و اگر داده برای قسمت دیتا آدرس تولید می‌کند ۳ بیت اولش رو ۰۰۱ میذاره و...

segment	ID
code	۰
Data	۱
stack	۲
Heap	۳
⋮	⋮

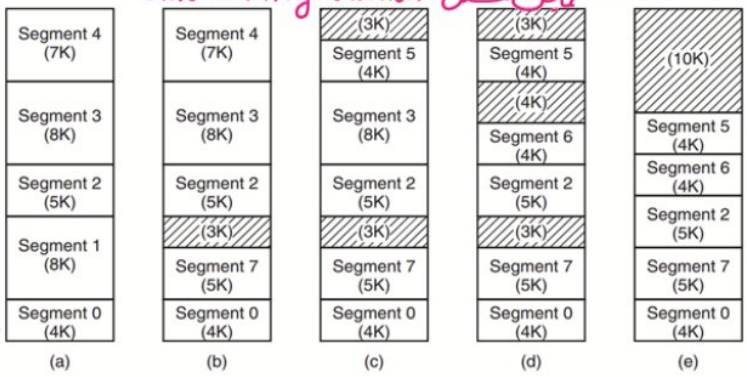
این تعداد بیت‌ها را چه کسی مشخص می‌کند؟ شرکت سازنده cpu، cpu‌هایی که روش segmentation را ساپورت می‌کند این را نیز مشخص می‌کنند. قسمت آفست هم مشخص می‌کند که هر سگمنت حداکثر چقدر می‌تواند باشد، در مثال بالا هر سگمنت نهایتا $2^{13}B = 8KB$ است و اگر برنامه‌ای بنویسید که یکی از بخش‌های آن بیش از 8KB باشد آن برنامه روی این cpu قابل اجرا نیست

```
0x240  main:   la $a0, varx
0x244          jal strlen
  ...            ...
0x360  strlen: li  $v0, 0  ;count
0x364  loop:   lb  $t0, ($a0)
0x368          beq $r0,$t1, done
  ...            ...
0x4050 varx    dw  0x314159
```

کد کامپایل شده یک برنامه

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

نمایش شکل external fragmentation



(a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

← هر سگمنت باید کامل بیاد، حالا شاید یک قسمت کوچیکی از اون رو لازم داشته باشیم.

★ segment table کی درست میشه؟ وکجاست؟ وقتی کاربر روی یک برنامه دابل کلیک میکنه Address space فرآیند تشکیل میشه، حالا سیستم مدیریت حافظه (os و loader) برای سگمنت‌های کد، دیتا و استک و هیپ،... توی حافظه رندوم حفره خالی میگرده و هر سگمنتی را که توی یک حفره‌ای گذاشت میره و جدول سگمنت اون فرآیند را می‌سازد و در حافظه اصلی قرار میدهد و آدرس شروع جدول سگمنت هر فرآیند (STBA = segment table base Add) را در PCB آن فرآیند قرار میدهد و هنگام CS، مقدار این ثبات در یکی از رجیسترهای MMU قرار داده می‌شود.

نکته) segment table یک فرآیند در زمان لود آن فرآیند رندوم آپدیت میشه و هر سگمنت که توی یک حفره قرار میگیره Base اش رو میره آپدیت میکنه و limit اش هم از روی بالاترین کد تولید شده برای آن بخش مشخص

نکته) در segment table می‌تونیم ستون‌های دیگری نیز داشته باشیم که بوسیله آنها می‌تونیم یکسری خصوصیات برای هر segment در نظر می‌گیریم، مثلاً شاید یک برنامه ۵ تا segment بود، خب باید مشخص کنیم که ۳ تا segment اش not valid است و استفاده نشده و اگر cpu آدرس مجازی تولید کرد که این نیلدش ۰ بود نقض حفاظتی رخ میدهد.

بیت‌های مربوط به حفاظت: V، Ro، Rw و x (rwx)، مثلاً بیت Ro مشخص میکنه که یک سگمنت فقط خواندنی است.

بیت present، Absent یا In/out: اگر این بیت ۱ باشه یعنی آن سگمنت در RAM وجود دارد و اگر ۰ باشه segmentation fault رخ میدهد و باید os بره و سگمنت مورد نظر را از دیسک بیاره بذاره توی RAM (برای جایابی Best fit خوبه)

بیت Referenced: آیا اخیرا به این سگمنت مراجعه صورت گرفته یا نه، موقع swapping این بیت به کارمون میاد.

و Modified: سگمنتی که توی RAM هست آیا در دیسک حضورش توی حافظه modify شده یا نه

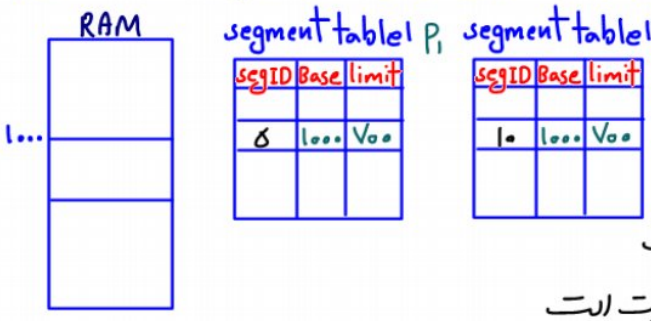
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

* خب حالا فهمیدیم که هر سطر segmentation table یک عرض مشخص داره، حالا از روی #s باید بریم سراغ یک سطر این جدول برای این کار #s را در اندازه‌ی درایه‌ی جدول قطعه ضرب می‌کنیم و آدرس سطر مد نظرمان را با جمع زدن با آدرس شروع جدول قطعه این فرآیند جمع می‌کنیم و سراغ سطر مربوط به خودمان می‌ریم (index کردن)

مزایای این روش:

۱- اشتراک داده در این مکانیزم ساده‌است و کافی است فرآیندهایی که می‌خواهند اشتراک داده داشته باشند یک سگمنت اشتراکی داشته باشند



برای اینکار کافی است در جدول سگمنت فرآیند ۱ و ۲ برای بخش مشترک یک پایه و کران یکسان تعریف کنید.

اگر شماره سگمنت تخصیص یافته به سگمنت مشترک در ۲ فرآیند P_1 و P_2 متفاوت باشه آدرس‌های مجازی تولید شده برای قسمت مشترک توسط فرآیندها متفاوت است

۲- امکان حفاظت از سگمنت‌ها به وجود آمده

۳- جابایی راحت‌تر شده چون برای هر سگمنت جا پیدا می‌کنیم که از کل فرآیند کوچکتند ولی چون فضایی که به هر سگمنت داده می‌شود باید پیوسته باشد هنوز به طور کامل مشکل حل نشده و انعطاف پذیری در این زمینه به بالاترین حد خودش نرسیده.

معایب

۱- هنوز مشکل تکه تکه شدن خارجی را داریم، مشکل تکه تکه شدن داخلی رو هم که داریم

۲- هنوز اتلاف حافظه رو داریم

۳- افزایش پویا اندازه سگمنت‌ها در این روش ساده نیست و os باید این را هندل می‌کند، برخی از سگمنت‌ها مثل heap و stack در طول زمان دارای طول متغیر هستند، حالا اگر اندازه اولیه‌ای که بهشون داریم پر بشه یک آدرس توسط cpu تولید میشه که خارج از محدوده سگمنت است و یک trap به os اتفاق میفته و os میاد و طول سگمنت رو افزایش میده. (البته از B&B بهتر شدیم)

۴- این روش انعطاف زیادی برای swapping نداره. چرا؟

چون معمولا یک فرآیند برای اجرا شدن به تمامی سگمنت‌های خود نیاز دارد و مثلاً نمی‌توان بخش کد را در RAM داشته باشیم ولی بخش پشته و داده را روی هارد نگه داریم

این در حالی است که یک فرآیند در یک بازه زمانی مشخص فقط به یک قسمت محدودی از دستورات، داده‌ها، پشته و... خود نیاز دارند و در ۹۰٪ مواقع فرآیندها با ۱۰٪ حافظه مجازی خود کار دارند.

به همین دلیل خیلی وقت‌ها در روش segmentation به os اجازه نمی‌دهند که فقط یکی سگمنت را در حافظه نگه دارد و بقیه را روی هارد ببرد که در این شرایط اگر بخواهیم یک فرآیند را swap out و بعدا swap in کنیم باید همه سگمنت‌ها را به هارد ببریم و بیاریم که مثل روش B&B میشه.

مکانیزم ۶) paging (روش صفحه‌بندی) (بعنوان روش اول مدیریت حافظه مطرح است)

دیدیم که پیوسته بودن حافظه فرآیند در B&B و پیوسته بودن حافظه هر سگمنت در روش segmentation خوب نیست، حالا برای رفع این باید بزنیم به قلب دشمن 😀 و اصلا بگیم حافظه پیوسته اختصاص ندهیم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

در این روش حافظه را به یکسری frame = قاب با سایز ثابت تقسیم می‌کنیم و برنامه کاربران را نیز به یکسری page = صفحه با همان سایز تقسیم می‌کنیم، حالا هروقت خواستیم به یک فرآیند حافظه اختصاص بدیم کافی است ببینیم آن فرآیند چندتا page میخواد و به همان اندازه فریم توی حافظه اصلی بهش بدهیم. حالا صفحات فرآیند میرند و در فریم‌های حافظه پخش می‌شوند، حالا os وقتی میخواد pageهای فرآیند را در frameهای مختلف حافظه قرار بده یک page table برای آن فرآیند تشکیل میده و این اطلاعات را توش میذاره و این PT را میذاره توی RAM و آدرس شروع اش رو (PTBA : page table base addr) را میذاره توی PCB فرآیند و موقع cs مقدار این رو در رجیستر PTBR مربوط به MMU لود می‌کند.

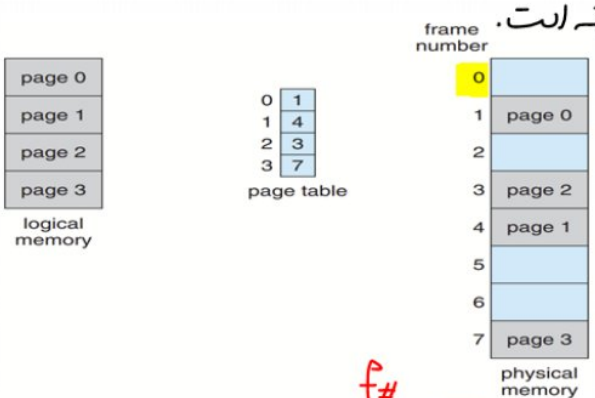
- توی اینجا ما اصلا با سگمنت‌های یک فرآیند کار نداریم و os به برنامه میگه به من چه که هرجا چی گذاشتی و هرجا که دوست داری بذار، فقط به من بگو چندتا page داری و من میرم وادن صفحات رو برات میذارم توی فریم‌هایی از حافظه که خالی است

مزایای paging

۱- نیازی به حافظه پیوسته برای فرآیند نداریم و pageهای یک فرآیند می‌توانند در مکان‌های مختلف حافظه RAM (حافظه فیزیکی) باشند

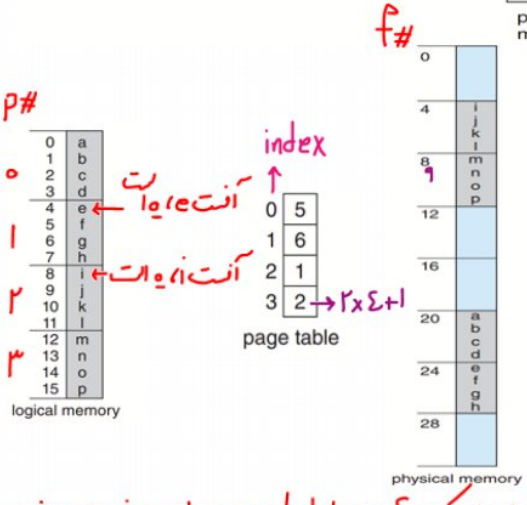
۲- چون سایز frame و page یکسان در نظر گرفته میشد قرار دادن pageها در فریم‌ها کار بسیار ساده‌ای برای os است.

۳- وقتی داریم یک برنامه را از روی هارد توی RAM لود می‌کنیم اگر سایز pageها با سایز واحد خواندن از روی دیسک که بلاک است متناسب باشه قرار دادن هر بار خواندن از هارد در frameها هم راحت و مقرون به صرفه است.



در شکل رو به رو نگاه کنید برنامه‌ای داریم شامل ۴ صفحه است، حالا os با توجه به اینکه کجاها فریم آزاد داره اینها رو میذاره توی RAM و آدرس این محل‌ها را در جدولی بنام page table نگهداری می‌کنیم که این جدول یک نگاشت بین حافظه منطقی (logical memory) و حافظه فیزیکی.

در هر صفحه یک تعداد word جا میشه، مثلاً در این مثال ۴ کاراکتر که مثلاً هر کاراکتر ۱ بایت هست جا میشه، اینجا هر word را ۱ کاراکتر گرفته، حالا توجه کنید که وقتی cpu آدرس تولید میکند آدرس یک word رو تولید می‌کند، پس برای مراجعه به یک word از یک page به ۲ تا آدرس نیاز داریم:



۱) اول باید ببینیم که آن آدرس توی کدوم page = آن فرآیند است که بعد از روی این می‌تونیم به page table = اون فرآیند مراجعه کنیم و ببینیم اون page در کدوم frame = حافظه RAM قرار گرفته

۲) کدوم word = اون page رو نیاز داریم؟ آفست در آن page

توجه) همیشه آدرس صفحات در حافظه منطقی یک فرآیند و شماره فریم‌های حافظه اصلی و شماره ایندکس‌های page table از ۰ شروع می‌شود.

مثال) فرض کنید در شکل بالا می‌خوایم به n مراجعه کنیم و می‌خوایم ببینیم n توی حافظه اصلی کجاست؟ میخوایم آدرس فیزیکی n رو محاسبه کنیم، بیان آدرس منطقی به اینصورت است: page3 :offset1 از page = شماره ۳، آفست ۱ را میخوام، حالا با استفاده از p# به page table مراجعه میشه و MMU می‌فهمه که پیج ۳ در فریم ۲ قرار گرفته، حالا با توجه به اینکه حافظه به فریم‌هایی باسایز برابر pageها تقسیم شده پس هر ۱ فریم هم چهارتا word = پس آدرس n میشه: آفست ← ۲×۴+۱ ← آدرس شروع فریم شماره ۲

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

نکته) دیگه ما مثل segmentation که میومدیم برای هر سگمنت یک Base و یک limit نگه میداشتیم نیاز نداریم برای هر page بخوایم ابتدا و انتهایش را نگه داریم، چون در اینجا همه صفحات و فریم ها یکسان است.

- در مثال بالا فرض کنید پردازنده آدرس مجازی ۱۳ را تولید کرده حالا p# و آفست دستی چطور بدست میاد؟

page size

۱۳ | ۴
۱۲ | ۳ = p#
۱ = offset = w#

۱۳ = ۱۱ ۰۱
p# w#

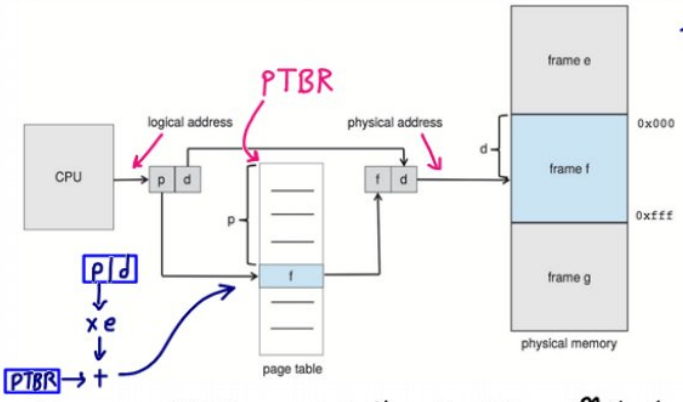
حالا آدرس فیزیکی چند بیتی است؟ ۵ بیتی

حالا باید بریم و از روی p#، page table - index رو بدست بیاریم تا بفهمیم این page ای که بدست آوردیم در چه فریمی از حافظه قرار گرفته و f# اش رو پیدا کنیم: پیج سوم این فرآیند در فریم دوم حافظه است، پس آدرس فیزیکی word - ۱۳ برابر:

۰۱۰ ۰۱
f# offset
مجموعا ۵ بیت

f# چند بیتی است؟ باید ببینیم RAM چند تا فریم داره و بعد ببینیم برای اون تعداد فریم چند بیت آدرس نیاز داریم:

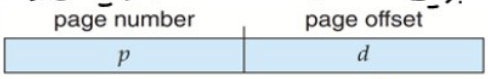
تعداد فریم = $\frac{۳۲}{۴}$ = ۸ → ۳ بیت آدرس برای اینکه مشخص کنیم پیج ما در کدوم فریم RAM است



بخش پرارزش آدرس مجازی شماره صفحه درون فرآیند مربوطه را مشخص میکند و سایر بیت ها آفست درون آن پیج را مشخص میکند

- سایز فریم و پیج توسط هاردور مشخص می شود، البته در برخی از سیستم ها ما می توانیم صفحاتی با سایزهای مختلف داشته باشیم که برخی از os ها نیز این موضوع را ساپورت می کنند.

- فرض کنید سایز فریم یا پیج ۲^n، word باشد و آدرس های مجازی m بیتی باشد، یعنی سایز فضای آدرس مجازی هر فرآیند (logical Address space) ۲^m بیت است، بنابراین n بیت از این m بیت برای آفست است و m-n بیت باقیمانده شماره page درون فرآیند را مشخص می کند، بنابراین مشخص است که هر فرآیند می تواند ۲^{m-n} پیج دارد.



سوال) فرض کنید اندازه آدرس مجازی ۳۲ بیت باشد که ۲۰ بیت با ارزش آن شماره صفحه را مشخص می کند و معماری پردازنده ۶۴ بیتی باشد و هر درایه PT دارای ۱۲ بیت کنترلی باشد، حالا به سوالات زیر پاسخ دهید:

۱- چند فریم در حافظه داریم و هر فریم چه اندازه ای داره؟

۲- چند فرآیند را می توانیم بصورت کامل در حافظه قرار دهیم (بدون در نظر گرفتن فضایی که PT ها نیاز دارند)

۳- سایز PT هر فرآیند چقدر است؟

۴- اگر حافظه از فرآیندها پر باشد حجم PT ها چقدر می شود؟

۱- اندازه هر فریم: $۲^{۳۲-۲۰} = ۲^{۱۲}B = ۴KB$ بنابراین داریم: تعداد فریم در حافظه = $\frac{۲^{۶۴}}{۲^{۱۲}} = ۲^{۵۲}$

۲- هر فرآیند $۲^{۲۰} = ۱MB$ صفحه (پیج) دارد، بنابراین: $\frac{۲^{۵۲}}{۲^{۲۰}} = ۲^{۳۲}$ تا فرآیند می توانیم بطور همزمان در حافظه داشته باشیم

۳- PT size = $۲^{۲۰} \times ۸B = ۸MB$

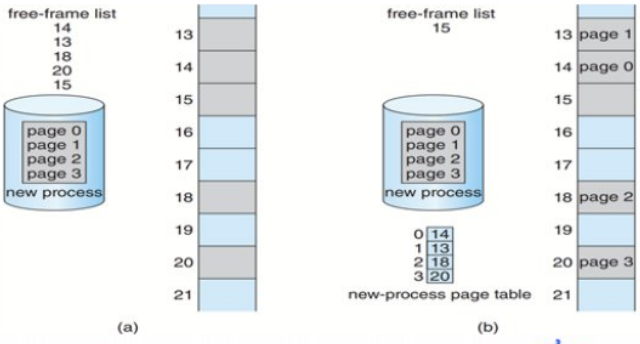
۴- حجم همه PT ها روی هم: $۲^{۳۲} \times ۲^{۲۳}B = ۲^{۵۵}B = ۲^{۲۵}GB$!

حالا معمولا ۲، ۳ هزار تا فرآیند توی حافظه داریم $۲۰۴۸ \times ۲^{۲۳}B = ۲^{۱۱} \times ۲^{۲۳}B = ۲^{۳۴}B = ۱۶GB$

یک فکری باید به حال این بکنیم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



یک فرآیند فریم‌هایی که نیاز دارد را چگونه تامین می‌کند؟

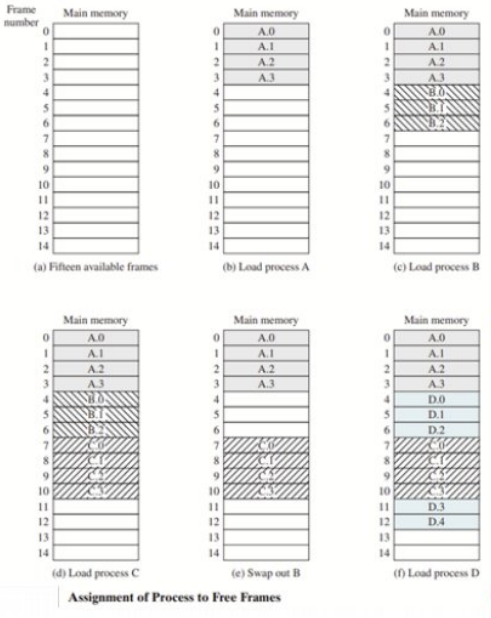
یک فرآیند که یکسری پیج دارد را می‌خواهیم وارد RAM کنیم، حالا این os چگونه به این فرآیند فریم میدهد؟

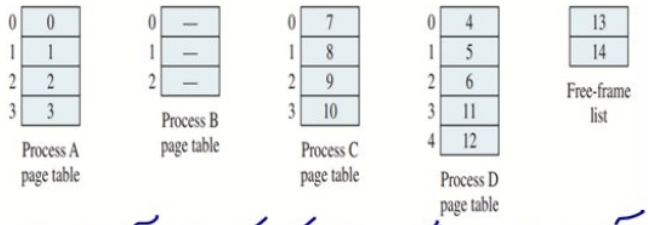
os یک لیستی از فریم‌های آزاد نگه میدارد، حالا وقتی os میخواهد به یک پیج یک فرآیند، فریم بدهد میره و از سر فریم‌های آزادش به اون فریم میده.

الان در این شال می‌خوایم به یک فرآیند ۴ پیجی، فریم بدیم، os میاد از سر لیست فریم‌های آزادش، فریم ۱۴ رو بر میداره و پیج ۰ را در این آدرس قرار میدهد و PT ـ این فرآیند را ایجاد میکند و توی ایندکس ۰ اش ۱۴ مینویسه ولیست فریم‌های آزاد خودش رو هم آپدیت میکند، بعد پیج ۱ را میداره توی فریم ۱۳ و PT ـ این فرآیند را آپدیت می‌کند و لیست فریم‌های آزاد خودش رو هم آپدیت میکند و ... و در نهایت که PT این فرآیند به وجود اومد میره و آدرس شروع این PT را (PTBA) توی PCB آن فرآیند می‌نویسد.

★ همانند روش segmentation کامپایلر باید برای فرآیند آدرس مجازی تولید کند، چون باید توی آدرس‌هایی که وجود دارد شماره صفحات و همین طور آفست هر صفحه مشخص باشد.

نکته) فضای آدرس مجازی که در اختیار برنامه‌ها قرار میدهیم در یک سیستم ثابت است و برای همه برنامه‌ها یکسان است، مثلاً بهشون می‌گیم فضای آدرس‌هایتان ۳۲ بیتی است، و اینجوری نیست که به برخی از برنامه‌ها کمتر و به برخی بیشتر بدهیم و چون اندازه PT هم بر اساس اندازه آدرس مجازی مشخص می‌شود پس PT همه فرآیندها نیز یکسان است.





توجه کنید که در شکل D ما نمی‌توانیم بخشی از فرآیند D را وارد کنیم چون در روش صفحه‌بندی ساده ما باید همه صفحات یک فرآیند را وارد حافظه کنیم. توجه کنید که وقتی فرآیند B خارج میشه ما می‌توانیم فرآیند D را وارد کنیم و همون طور که می‌بینیم دیگه در این روش لازم نیست همه فرآیند پیوسته یکجا قرار بگیرد.

خب اینجوری دیگه چون همه صفحات مربوط به همه فرآیندها می‌توانند هر جا بشینند ما external Fragmentation نداریم و اگر فضایی به اندازه یک فرآیند وجود داشته باشد می‌تونیم اون رو بیاریم حتی اگر اون حفره یکپارچه نباشه و در اینجا فقط در صورتی فضای خالی در حافظه خواهیم داشت که هیچ فرآیندی در حافظه جا نشود نه اینکه فرآیندی در یک حفره جا نشود.

نکته) روش paging تکه تکه شدن خارجی که نداره و داخلی را حداقل می‌کند چون فقط در آخرین پیج یک فرآیند به اندازه نصف صفحه حافظه هدر می‌رود

سوال) اگر اندازه حافظه یک فرآیند ۷۲۸۷۰ بایت باشد و اندازه هر صفحه ۲KB باشد مقدار internal fragmentation چقدر است؟

این فرآیند در چند صفحه جای می‌شود = $\frac{۷۲۸۷۰}{۲۰۴۸} = ۳۵,۵۸ \Rightarrow$ ۳۶ صفحه

$۳۵ \times ۲۰۴۸ = ۷۱۶۸۰ \Rightarrow ۷۲۸۷۰ - ۷۱۶۸۰ = ۱۱۹۰ \Rightarrow ۲۰۴۸ - ۱۱۹۰ = ۸۵۰$ Byte

حالا سوالی که اینجا مطرح می‌شه اینه که اندازه page را کوچیک بگیریم یا بزرگ؟

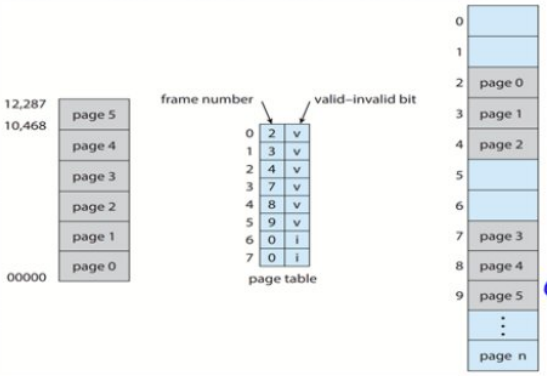
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com
@konkurcomputer
www.konkurcomputer.ir

اگر اندازه پیج را کوچیک بگیریم:
۱- internal fragmentation کاهش می یابد
۲- اندازه جدول صفحه هر فرآیند بزرگ می شود

" " " " بزرگ " :
۱- " " افزایش "
۲- اندازه جدول صفحه هر فرآیند کوچک تر می شود
۳- برای برنامه هایی که I/O حجیم دارند مفید است

مرور۳: فرض کنید به یک سرور ۴ تا کارت شبکه ۱۰گیگ وصل است و بصورت پیوسته ترافیک وارد سرور می شود، حالا فرض کنید اندازه هر صفحه ۴KB باشد، حالا در ۱ ثانیه چند گیگابایت داده وارد سیستم می شود، حالا اگر os بخواد برای این چند گیگابایت، ۴K بایت، ۴K بایت صفحه اختصاص بده، خب خودهمین کار خیلی زمان بر است، می باید صفحه اختصاص بده به فرآیندهای PT آن فرآیند و لیست صفحه های آزادش را آپدیت بکند، در این شرایط خاص میان و به این فرآیند ۴ صفحه مثلاً ۱ یا ۲ گیگا بایتی میدهند. توجه کنید در کاپیوترهای عادی این کار زمانی کند چون میزان هدر رفت حافظه خیلی زیاد میشه



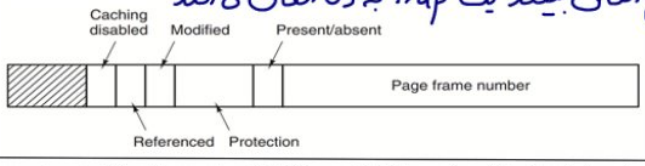
protection

باتوجه به اینکه به فرآیند ها اجازه میدیم آدرس های مجازی بزرگی برای خودشان داشته باشند بنابراین تعداد سطرهای PT می تواند خیلی بزرگ بشود و این قابلیت برای فرآیند ها به وجود میاد که بتوانند به کل فضای RAM آدرس دهی انجام بدهد و به همه اش دسترسی داشته باشد، ولی لزوما اندازه فرآیند ما ممکن است آنقدر بزرگ نباشد که بخواهد از کل فضای حافظه استفاده کند، مثلاً در مثال روبه رو فرض کنید که هر برنامه می تواند ۸ تا پیج داشته باشد ولی این برنامه فقط از ۶ پیج اول فضای آدرس خودش استفاده کرده، مثلاً آدرس مجازی ۱۴ بیتی بوده (۰ تا ۱۶۳۸۳) و اندازه صفحه ۲KB ای بوده و این برنامه فقط از آدرس ۰ تا ۱۰۴۶۸ استفاده کرده. به همین علت یک بیت میذارند توی PT به نام valid-invalid bit وهنگامی که صفحه ای در فضای منطقی وجود داشته باشد valid گذاشته می شود و یعنی این صفحه legal یا valid است. حالا اگر فرآیند باک یک آدرس اشتباهی تولید کند که در page های valid اش نباشد، مثلاً یک pointer اشتباهی وجود داشته باشد که باعث شود بخواهیم به پیج ۱۷ام یا ۱۸ام مراجعه کنیم، چون این ۲ پیج برای این فرآیند invalid است، MMU متوجه می شود که این آدرس نادرست است و با یک وقفه trap به os ایجاد میکند تا os به این موضوع رسیدگی کند و مثلاً شاید به خاطر این exception برنامه را terminate کند.

page-table length register: PTLR : اگر تمامی بیت های هر entry جدول صفحه را نیاز داشته باشیم می توانیم به جای اینکه ۱ بیت را به valid-invalid bit اختصاص بدیم بیایم یک رجیستر PTLR بذاریم که طول PT = valid هر فرآیند را مشخص می کند، این با هر آدرس مجازی ای که برنامه تولید میکند چک میشه و اگر آدرس در محدوده معتبر آدرس های فرآیند نبود یک trap به os اتفاق میفتد.

نکته) به نوعی می توان گفت در PT هم internal fragmentation داریم، چون ما تعدادی سطر در PT در نظر گرفتیم که مورد استفاده فرآیند ما نیستند و بنابراین حافظه هدر رفته محسوب می شوند

- می توانیم ۲ بیت دیگه بذاریم که ۴ حالت از نظر r و w و x می توانند برای ما به وجود بیاورند، مثلاً read-only، read-write، exe-only حالا مثلاً اگر پیجی read-only باشه و اگر قرار بود write ای روی اون پیج اتفاق بیفتد یک trap به os اتفاق می افتد



- در زیر یک سطر از PT را می بینید، توجه کنید که ساختار PT خیلی machine dependent است و طول اش هم متغیر است ولی معمولا

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

۳۲ بیت است و هشتمین فیلدش page frame number است. که همشون رو قبلا توضیح دادم

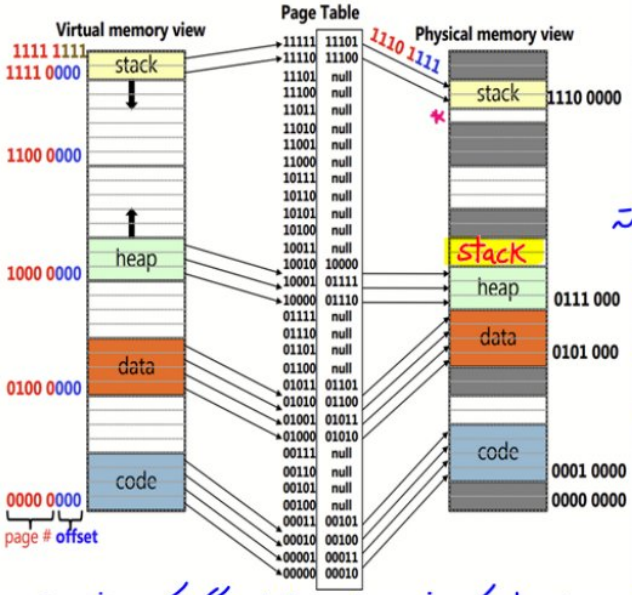
caching disabled: اسم اش روشه، این بیت امکان استفاده از کش را برای این صفحه غیرفعال میکند. برخی از فریم های حافظه رو به I/O دیوایس ها اختصاص میدیم، مثلاً فرض کنید قراره یک محتوایی قراره در طی یک زمانی از I/O بیاد و ما هی اون رو بخونیم، پس کش برای I/O اینجا بی معنی است، حالا اون کامپیوترهایی که Memory Map I/O هستند یعنی: { فضای حافظه / فضای I/O } در این کامپیوترها از دستورات یکسان برای دسترسی به memory space و I/O space استفاده می شود بنابراین باید آدرس های دو تا space باهم متفاوت باشد، میان اگر لازم باشد caching روی فضای I/O رو Disable می کنند، کامپیوترهایی که I/O mapped I/O هستند نیازی به این بیت ندارند

copy on write: وقتی قرار باشد یک صفحه بین چند فرآیند شیر بشه این بیت برای آن صفحه ۱ می شود و با این یک کردن این بیت به هر فرآیندهایی که از این استفاده می کنند می گیم در این صفحه write نکنند و اگر کسی بخواد write بکند باید این صفحه برایش کپی بشه و بعد توی کپی خودش write کند، مثلاً وقتی یک پدر چند فرزند بدنیا میاره میاد و هر یا برخی از صفحاتش رو بین اونها شیر میکند تا توی حافظه صرفه جویی بشه و این بیت رو ۱ میکند و تا مادامیکه هر فرزند فقط داره read میکند برایش صفحه جدیدی ایجاد نمی شه

- حالا توی مثال بالا فرض کنید آدرس شروع page table خانه ۲۰۰۰ باشه و هر خانه (درایه = entry) PT، ۳۲ بیت باشد حالا MMU چطور از روی p# ایندکس PT را پیدا میکند؟

PT index = PTBR + p# x entry size = ۲۰۰۰ + ۳ x ۴ = ۲۰۱۲

پس MMU برای پیدا کردن f# توی PT سرچ نمیکند که کند باشه و بصورت سخت افزاری از روی p# میره سراغ سطر p# ام جدول فرآیند جاری و f# رو برمیداره و میذاره سمت چپ آدرس



افزایش طول فضای آدرس یک بخشی از برنامه

فرض کنید استک می خواهد بزرگ شود و به جای ۲ تا پیج به ۴ پیج نیاز دارد حالا وقتی یک دستوری به آدرس از پشته انجام میشه که خارج از آدرس اختصاص یافته به پشته است (cpu یک آدرس مجازی تولید میکند و میگه فلان مقدار رو توی این آدرس پشته بنویس) این آدرس میفته توی فضای * که مالِ پشته نیست میره توی PT رو نگاه میکند و میبینند فیلد V اش صفر است و یعنی این صفحه توی فضای حافظه نیست و بنابراین page fault رخ میده و بعد os میاد چون این PF برای رشد پشته بوده os میاد و ۲ تا صفحه پیدا میکند و میره به استک و بعد میره و PT را آپدیت میکند و میگه پیج های مربوط به استک رو توی کدوم فریم ها گذاشتم. همان طور که مشخص است ۲ فریم دیگری که به پشته داده شده پشت سر حافظه قبلی نیست و براحتی و بدون اینکه بخواهیم فضای پشته قبلی توی RAM را جا بجا کنیم یا کار خاصی بکنیم

shared pages

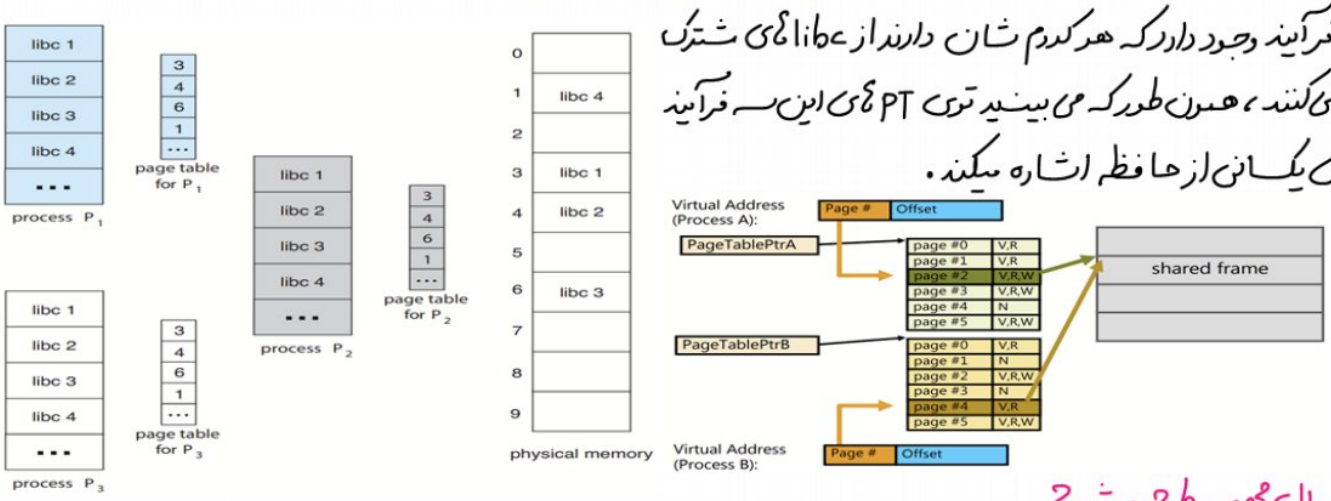
مثال: فرض کنید که یک برنامه editor با سایز ۱۵۰KB داریم و سایز هر page، ۵۰KB است، بنابراین که editor به ۳ صفحه نیاز دارد، حالا فرض کنید هر یوزری که بخواهد از این ادیتور استفاده کند به یک پیج نیاز دارد تا حاصل تایپ خودش را در آن نگهداری کند، بنابراین هر یوزر به ۲۰۰KB فضا برای کار کردن با این editor نیاز دارد (۴ پیج)، حالا اگر ۴۰ تا یوزر به این سرور متصل هستند و دارند همزمان با این editor کار میکنند، خب اگر هرکس فضای جدای خودش رو داشته باشه، ۸۰۰۰KB = ۸MB که برابر ۱۶۰ پیج است را اشغال می کند در صورتیکه همه این یوزرها دارند از یک کد استفاده می کنند حالا اگر بیایم این سه پیج editor رو

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

بین این ۴ share کنیم و فقط به ازای دیتای هر یوزر بهشون صفحه جداگانه بدیم $۱۵۰K+۴۰\times۵۰K=۲۱۵۰K$

نکته) به شرطی می توانید این کار را بکنید که کد ما reentrant باشه، مثل DLL ۴ که قبلا مفصل توضیح دادم، کد reentrant کدی است که اگر یک فرآیند با توجه به نیاز خودش مقادیر متغیرهای آن را تغییر بدهد فرآیندهای دیگر هم می توانند بصورت جداگانه به تناسب نیازهای خودشان مقادیر متغیرها را تغییر بدهند و تداخلی در کار هیچکی به وجود نمی آید

در این شکل ۳ فرآیند وجود دارد که هرکدوم شان دارند از libهای مشترک استفاده می کنند، همون طور که می بینید توی PTهای این سه فرآیند داره به فریم های یکسان از حافظه اشاره میکنه.



- حالا ۳ تا سوال مهم مطرح میشه؟

۱- چه کسانی به PT یک فرآیند دسترسی دارند؟

۲- PT را کجا بذاریم؟

۳- قرار دادن PT در حافظه ما را کند نمی کند؟ چه راه کاری وجود دارد؟

جواب ۱) os به ۲ دلیل باید به PT همه فرآیندها دسترسی داشته باشه

۱- گفتیم به ازای هرکدام از فرآیندهایمان یک PT داریم که وقتی بین فرآیندها cs انجام می شه باید اطلاعات PT جاری عوض بشه که این کار را cpu dispatcher در زمان cs انجام میده تا قسمت های مجاز دسترسی به حافظه برای فرآیند جدید براساس PT اش مشخص شود پس os باید به PT فرآیندها دسترسی داشته باشه تا بتواند این کار را انجام بدهد (باید بداند PT هر فرآیند کدام است)

۲- وقتی فرآیندی در حال اجرا است ممکن است یک system call بزند و پارامتری که می خواهد به آن system call بدهد را رفته توی حافظه خودش گذاشته و حالا اون پارامتر رو با آدرس این حافظه به os، pass می کند (یعنی انگار داره p# و آفست پارامتر را به os میده) خب پس os باید به PT این فرآیند دسترسی داشته باشه که بتواند بفهمد اون پارامتر کجای RAM است.

PT را در کجا بذاریم؟

۱) یکی رجیستر خاص در cpu در نظر بگیریم و PT را در آنها قرار بدهیم، اینجوری سرعت میره بالا ولی...
و وقتی cs انجام میشه این رجیستر در PCB آن فرآیند ذخیره بشوند و PCB فرآیند بعدی را در این رجیستر لود کنیم.

۱) در این مکانیزم باید دسترسی به این رجیستر و modify کردنشان privilege باشه و فقط در انحصار os باشه، چون اگر یک فرآیند خودش بتواند محتوای این رجیستر را تغییر بدهد خب حفاظت از بین میره و میتونه قسمت های دیگه حافظه هم دسترسی داشته باشه

۲) ما نمی توانیم حجم رجیستر زیادی در پردازنده ها داشته باشیم بنابراین این تعداد محدود برای سایز PT هم محدودیت ایجاد میکند
خب دیدیم که مثلاً PT می تواند 4MB باشه و این یعنی باید 2^{20} تا رجیستر ۳۲ بیتی در پردازنده میداشتیم که عدد بزرگی است و معمولا پردازنده ها نمی توانند این تعداد رجیستر را تامین بکنند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

- از این مکانیزم بیشتر در سیستم‌هایی که سایز PT هاشون کوچکتر هست استفاده کنیم، همچنین در سیستم‌هایی که page های بزرگتری دارند می‌توان از این روش استفاده کرد، مثلاً اگر سایز page ۸MB ای باشد در یک سیستمی که ۳۲ بیت آدرس مجازی دارد خواهیم داشت: اندازه پیج‌ها: $2^{23} \times 2^{20} B$ → تعداد پیج‌های هر فرآیند $Max = 2^9 = 512$

فرض کنید هر entry جدول صفحه ۴B باشد، پس ۵۱۲ تا رجیستر ۳۲ بیتی کافی بود در این شرایط

۳- در این مکانیزم درسته که وقتی می‌خوایم یک آدرس رو ترجمه کنیم سریع از طریق رجیسترها انجام میدیم و سراغ حافظه نمی‌ریم ولی موقع cs باید یک تعداد زیادی رجیستر ذخیره و بازیابی بشه که زمان cs را خیلی بالا میبره.

۲- به جای اینکه PT یک فرآیند را در رجیسترها بذاریم، در حافظه باشه و فقط موقع cs آدرس شروع شون رو بذاریم توی PTBR.

۱- در راستای حفاظت تغییر PTBR باید privil edge باشد و فقط در انحصار os باشد

۲- زمان cs در این حالت پایین اومد

۳- زمان دستیابی به حافظه خیلی زیاده، چون به ازای هر مراجعه‌ای که بخواهیم به حافظه داشته باشیم باید دوتا مراجعه به حافظه انجام بدیم، یکبار باید به PT توی حافظه مراجعه کنیم تا ترجمه آدرس انجام بدیم (آدرس شروع فریم توی حافظه رو پیدا کنیم) و آدرس فیزیکی رو پیدا کنیم و بعد یکبار دیگه بریم سراغ این حافظه فیزیکی که محتوای مورد نظرمون رو برداریم

توی روش قبلی برای مرحله ترجمه آدرس میرفتیم سراغ رجیسترکه سریعه نه حافظه

TLB (Translation look-aside Buffer)

خب اگر بخوایم به ازای هر مراجعه به حافظه ۲ مراجعه به حافظه داشته باشیم خیلی کارایی سیستم‌مان میاد پایین. حالا چیکار کنیم؟

بیایم و بخشی از PT را کش کنیم: بیایم و ترجمه‌های آدرس‌هایی که وقت گذاشتیم براش و یکبار ترجمه شون کردیم رو کش کنیم.

بیایم و اطلاعات مربوط به ترجمه صفحه (در ایده‌ی جدول صفحه) صفحاتی که اخیرا استفاده می‌شوند را در TLB در کنار رم MMU قرار بدیم.

توجه) سرعت پردازنده‌ها هر سال حدودا ۶۰٪ زیاد میشه اما سرعت DRAM هر سال حدوداً ۹٪ زیاد میشه، بنابراین سرعت دستیابی به حافظه خیلی عقب‌تر از سرعت اجرا دستورات است.

برای این منظور از کش Fully associative استفاده می‌کنیم که اصطلاحا بهش TLB گفته میشه که معمولاً بین ۶۴ تا ۱۰۲۴ سطر دارد که می‌تواند یا تمام PT یا بخشی از آن را در خودجای دهد و cpu می‌تواند با سرعت بالا بهش دستیابی پیدا کند و وقتی توی این حافظه CAM (content Addressable memory) به دنبال #p یک #f هستیم تمام خانه‌های TLB بصورت موازی جستجو می‌شود و اگر در این حافظه باشد hit رخ میدهد و اگر نباشد miss رخ میدهد، حالا اگر با TLB miss مواجه شدیم دیگه چاره‌ای نداریم که بریم سراغ حافظه اصلی که در آن همه PT وجود دارد. پس به شرطیکه صفحه‌ای که ما به دنبال #f اش هستیم جزو صفحاتی باشه که اخیرا بهش مراجعه شده به احتمال زیاد توی TLB هست و Hit رخ میده

locality of refrence: اصل مراجعات محلی

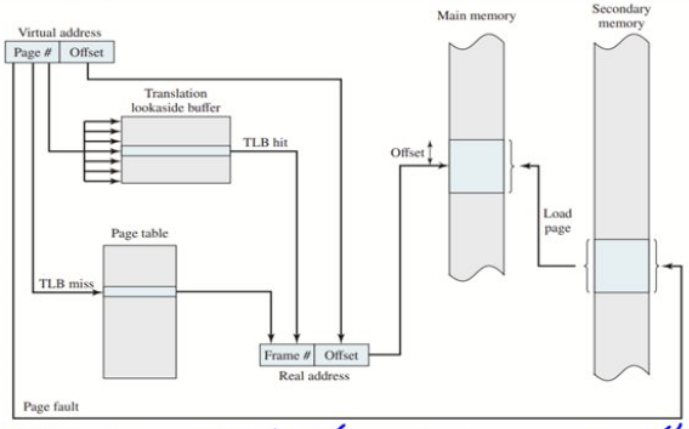
۱) Temporal locality of refrence: اگر داری از داده‌ای استفاده میکنی احتمالا در آینده نزدیک دوباره ازش استفاده میشه، مثلاً تعریف متغیر

۲) " " " spatial: برنامه‌ها sequential هستند و بطور عادی پشت سر هم اجرا می‌شوند

نکته) در برخی از مراجع به TLB ممکن است current page register: CPR می‌گویند

نکته) برای یک پردازنده یک TLB وجود دارد. چرا TLB بزرگ نمی‌دارند؟ هزینه، پیچیدگی ← احتمال باگ بیشتر میشه، افزایش زمان دستیابی

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner



آدرس مجازی که از cpu میاد بیرون، ابتدا با قسمت p# اش به TLB مراجعه می‌کنیم و اگر hit شد که شماره فریمی که page ما در آن قرار گرفته را میذاریم سمت چپ displacement و آدرس فیزیکی را بدست می‌آوریم و حالا اگر TLB miss اتفاق افتاد به اجبار باید به PT که توی RAM ذخیره شده مراجعه کنیم و f# رو از اونجا برداریم و بعد باید درایه‌ای که در PT پیدا کردیم را بذاریم توی TLB، حالا اگر TLB جا داشت که میذاریم و اگر جا نداشت باید یکی از درایه‌های TLB را خارج کنیم و این رو بذاریم اونجا. حالا اگر دوباره به همین صفحه مراجعه کنیم hit رخ میده. وقتی TLB پر هست و مجبوریم یکی رو بندازیم بیرون باید ببینیم modify bit این سطر رو کپی کنیم توی PT مربوط به این فرآیند توی RAM، مقادیر دیگر TLB از قبل توی PT وجود داره (به جز بیت R). توجه کنید وقتی از توی PT میخوایم یک چیزی بذاریم توی TLB هم فیلدهایی که TLB داره از PT کپی میشه

توجه) TLB miss (TLB fault) با page fault فرق دارد: فرض کنید یک فرآیندی ۲۰ تا صفحه داره که OS، ۸ تاش رو از روی که در گذاشته توی RAM، یعنی ۸ تاش present هستند و ۱۲ تاش absent هستند، حالا page fault فقط برای اون ۱۲ تا پیجی که توی RAM نیستند رخ میده، حالا TLB miss نه تنها برای هم این ۱۲ تا پیج رخ میده بلکه ممکن هست برای صفحاتی که توی RAM هستند یعنی اون ۸ تا هم رخ بده، چون ممکنه از این ۸ تا مثلاً ۳ تاشون اخیرا استفاده نشده باشند و توی TLB نباشند و براشون miss رخ بده، توجه کنید که برای صفحات present، یعنی اون ۸ تا هیچ وقت page fault رخ نمیده

نکته) به هر حال TLB پس از یک مدتی پر می‌شود و بنابراین حتما miss رخ میدهد که در این شرایط باید بحثی از TLB را جایگزین بکنیم که در این حالت نیاز به سیاست‌های جایگزینی خواهیم داشت مانند LRU.

non_removable = wired down TLB: در برخی از سیستم‌ها می‌توانیم wire down TLB داشته باشیم به این معنا که قسمت‌هایی از حافظه که پر مصرف هستند مثل قسمت‌هایی از کرنل را مشخص می‌کنند که وقتی TLB miss رخ داد هیچ وقت اینها را از TLB خارج نمی‌کند (جایگزین نمی‌کند) و بنابراین اینها همیشه hit می‌شوند (یعنی ما مطمئن هستیم که برای مراجعه به آن قسمت‌های حافظه نیاز به ۲ بار مراجعه به حافظه پیدا نمی‌کنیم)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

ASID (Address_space identifier): تا الان توی TLB نوشتیم مثلاً صفحه شماره ۱۴۰ در فریم ۳۱ قرار گرفته، صفحه شماره ۲۰ در فریم ۳۸ قرار گرفته حالا وقتی از یک فرآیند به فرآیند دیگر switch می‌کنیم دیگه ما نباید برای فرآیند جدید از این مقادیر استفاده کنیم، پس تا الان وقتی cs انجام می‌شد میومدیم و کل TLB قبلی را بی ارزش در نظر می‌گرفتیم و میریختیم و همه بیت‌های valid شون رو ۰ میکردیم و مجددا برای فرآیند جدید TLB شروع به پر شدن می‌کرد. خب حالا یک راه حل برای اینکه جلوی هدر رفتن زمان گرفته بشه این هست که ما به ازای هر فرآیند یک شناسه منحصر به فرد به اسم ASID داشته باشیم و این شناسه را هم توی TLB نگه داریم، در این حالت وقتی بین فرآیندها switch می‌کنیم نیازی نداریم TLB را flushed (erased) کنیم.

effective memory access time

وقتی cpu یک آدرس تولید میکند چقدر طول میکشه تا دیتا به دست اش برسه؟

خب وقتی آدرس مجازی تولید شد میریم توی TLB رو نگاه می‌کنیم حالا یا hit رخ میده یا miss، پس زمان دسترسی به فریم فرق میکند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

مثال) در ماشین ۸۰۳۸۶ که اوایل دهه ۹۰ میلادی جز کامپیوترهای خوب بود، ۱ سطر TLB در نظر گرفته بودن که فقط همین یک سطر باعث می‌شد ۸۰٪ مواقع آدرس مجازی توی TLB پیدا بشه و hit ratio، ۸۰٪ بوده، حالا اگر TLB access = ۲۰ns و memory access = ۱۰۰ns باشد داریم:

EAT = Effective access time = ۰٫۸ x ۱۲۰ns + ۰٫۲ x ۲۲۰ = ۱۴۰ns

TLB hit: ۲۰+۱۰۰ | TLB miss: ۲۰+۲۰۰

اگر فقط میخواستیم زمان پیدا کردن f# رو بدست بیاریم داشتیم:

$t_{TLB} + (1-h_{TLB}) \times t_m$ = TLB hit: ۲۰ | TLB miss: ۲۰+۱۰۰

مثال) در ماشین ۸۰۴۸۶ با ۳۲ سطر TLB داشتیم و h_{TLB}=٪۹۸ در این شرایط: EAT = ۰٫۹۸ x ۱۲۰ + ۰٫۰۲ x ۲۲۰ = ۱۱۷٫۶ + ۴٫۴ = ۱۲۲

خب minimum برابر ۱۲۰ اِلت، حالا این شده ۱۲۲، پس خیلی این minimum اِلت

نکته) cpu‌های امروزه عموماً چند لول TLB دارند، مثلاً core i7 اینتل ۲ تا L_1 داره یکی برای دیتا و دیگری برای داده و یک L_2 داره (۵۱۲ | ۶۴ | ۱۲۸)

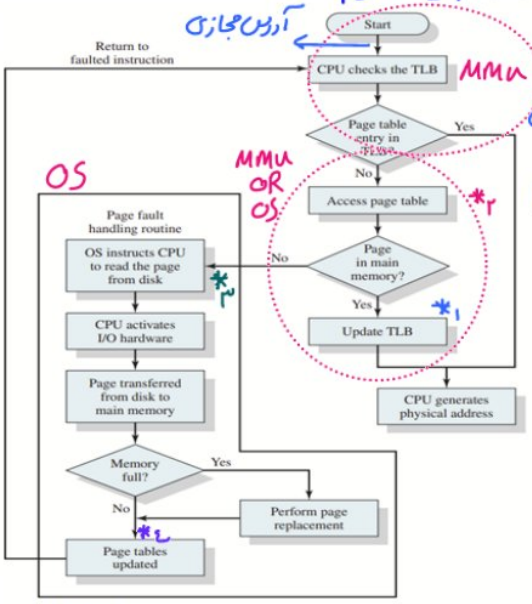
نکته) اگر فرآیندی I/O bound باشد تندتند میره خواب cs میکنند و حالا باید TLB رو مدام پر و خالی کنیم و خیلی هم ازش استفاده نمیشه و این یعنی overhead ی cs زیاد میشه

consistency in TLB

زمانی که PT بروزرسانی می‌شود باید TLB هم آپدیت شود

مثلاً وقتی یک پیج valid می‌شود: وقتی استک یا heap فضای جدیدی بخواهد. یا یک پیج invalid می‌شود: وقتی یک حافظه‌ای را آزاد می‌کند یا یک صفحه به هارد منتقل می‌شود، حالا اگر پیجی مربوطه توی TLB باشد باید سطر آن را آپدیت کنیم.

روال کل فرآیند

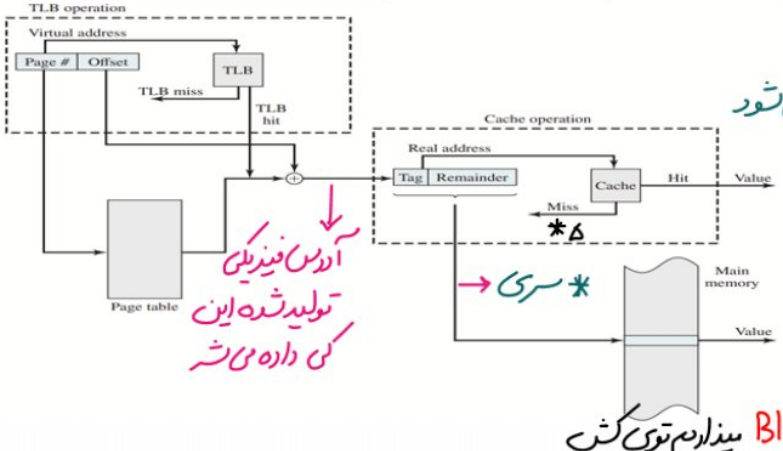


*۱ چرا باید TLB را آپدیت کنیم؟ چون باید در TLB اطلاعات ترجمه آدرس صفحاتی که اخیرا استفاده می‌شوند باشد، خب الان یک صفحه‌ای را می‌خوایم استفاده کنیم که نبودش توی TLB دیگه. اینجا اگر TLB پر باشه باید Replace داشته باشیم مثلاً اونکه اخیرا از همه کمتر استفاده شده رو بندازیم دور

*۲ اگر os بخواهد این کار را بکند کند میشه؟ نه لزوماً، چون اگر قرار باشه هم کارِ رو MMU بکند سخت افزارش پیچیده‌تر چون باید هم این کارهایی که دورش خط کشیدم رو انجام بده، حالا اگر این کارها رو نکند MMU کوچک می‌شود، حالا می‌تونیم شبیه ایده ریسک از فضای اضافه مثلاً برای بزرگتر کردن TLB استفاده کنیم که هم hit rate ی TLB میره بالا تر و هم MMU سریعتر شده

*۳: وقتی page fault رخ میده، فرآیند در حال اجرا Block می‌شود و بعد os میره PF را هندل می‌کند

*۴: بعد که فریم مدنظر از توی هارد به RAM منتقل شد PT را به روز می‌کنیم که دفعه بعدی که فرآیند اجرا می‌شود دچار PF نشود و فرآیند Block شده نیز ready می‌شود



*۵: اگر توی کش نباشد چیکار می‌کنیم؟ از توی حافظه یک Block میذاریم توی کش

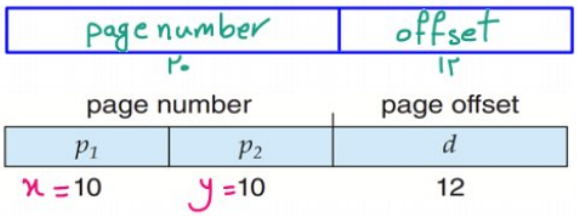
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

روش‌های حل مشکل PT های بزرگ: ۱) صفحه بندی سلسله مراتبی: Hierarchical paging (جدول صفحه چند سطحی)

فرض کنید یک سیستم داریم که آدرس های مجازی اش ۳۲ بیتی است و اندازه هر صفحه آن ۴KB است، پس ۱۲ بیت برای آفست نیاز داریم و بنابراین ۲۰ بیت P# رو مشخص می کرد و این یعنی یک فرآیند می تواند 2^{20} صفحه داشته باشد، بنابراین یک PT ساده 2^{20} سطر دارد، و هر سطر فرض کنید ۴ بایت باشد و بنابراین سایز هر PT برابر $2^{20} \times 4Byte = 4MB$ است

خب نگه داشتن تخصیص حافظه در روش paging، فریم نیست؟ خب اگر فریم های ۴KB ای هستند ما نمی توانیم این ۴MB را در یک فریم جا بدیم، بنابراین چون نمی تونیم حافظه پیوسته به PT یک فرآیند اختصاص بدهیم که بتونیم بهش دسترسی پیدا کنیم مجبوریم بیایم و خود این PT را صفحه بندی کنیم. جایزه ۱ میلیون دلاری

حالا وقتی PT را میایم به پیج های مختلف می شکنیم اش، به یک PT دیگر برای آدرس دهی به PT هایی که آنها را صفحه بندی کردیم نیاز داریم.

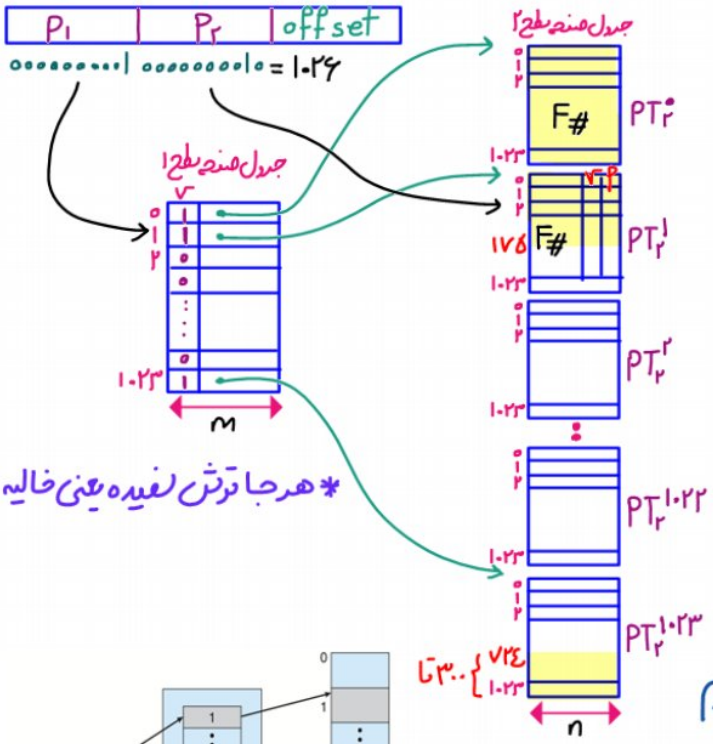


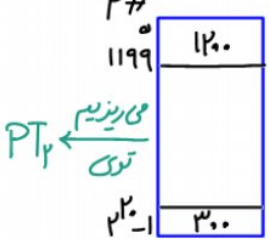
به مثال زیر توجه کنید: ما میایم PT مان که 2^{20} تا سطر دارد را می شکنیم به ۱۰۲۴ تا جدول ۱۰۲۴ تایی $1024 \times 1024 = 2^{10} \times 2^{10} = 2^{20}$

2^x تا جدول 2^y تایی

اندازه جداول سطح ۲ را چقدری بگیریم خوبه؟ ۴KB ای. چرا؟ چون Internal Fragmentation نداشته باشیم

یعنی فرمت آدرس چجوری میشه؟ $x=8$, $y=12$



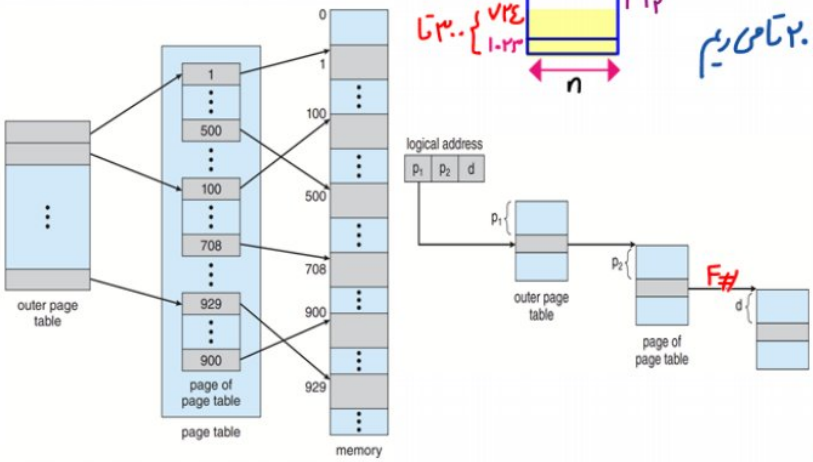


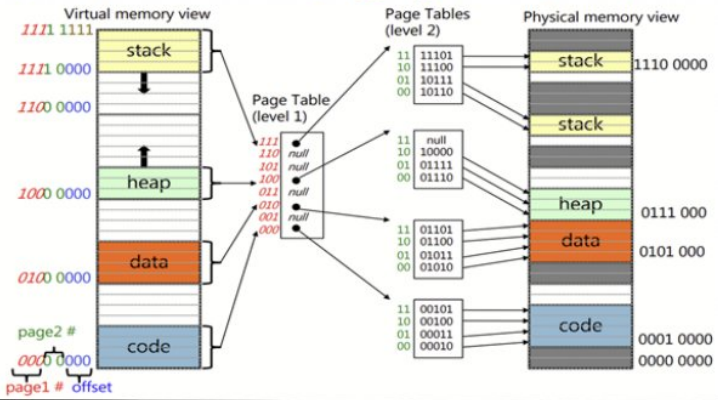
فرض کنید فرآیند ما بصورت زیر باشد: یعنی از M اصفحه ای که دارد فقط ۱۲۰۰ صفحه بالایی اش و ۳۰۰ صفحه پایینی اش را استفاده کرده و بقیه رو استفاده نکرده بنابراین در مجموع ۱۵۰۰ صفحه استفاده کرده

از PT_2 ۴، ۱۰۲۱ جدول خالی داریم که یا جداول به این ۱۰۲۱ ای اختصاص نمی دیم یا حتی اگر هم اختصاص داده بشوند می توان این جداول را بدون لطمه به اجرای فرآیند روی هارد ذخیره کرد.

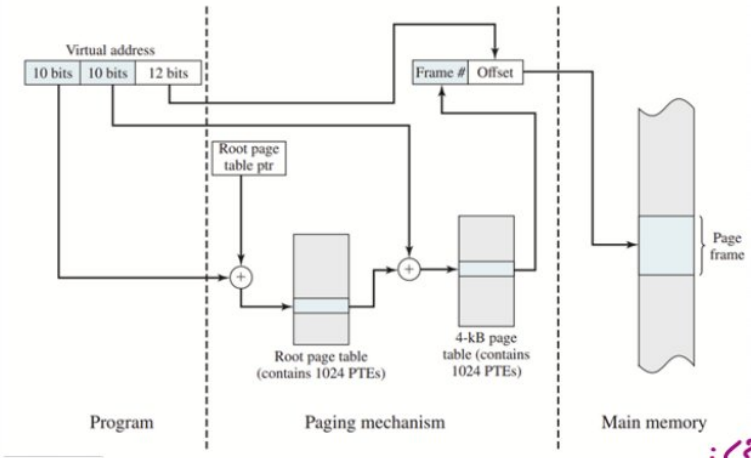
حالا ۵ آدرس شروع هر یک از این جداول را در PT_1 می گذارد

قبلا با چند بیت می رفتیم سراغ PT؟ ۲۰ تا، الان هم باهون ۲۰ تا می ریم





سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner



Address Translation in a Two-Level Paging System

چقدر کوچیک شد PT؟ به جای یک جدول با 2^{20} = 1M سطر نقط ۴ تا جدول ۱۰۲۴ تایی داریم استفاده می‌کنیم توی RAM، یعنی توی این مثال $\frac{2^{20}}{2^2 \times 2^{10}} = 2^8 = 256$ برابر کم کردیم حافظه را، اما برای ترجمه آدرس باید ۲ بار به RAM مراجعه کنیم یکبار به PT_1 و یکبار به PT_2^K

* توجه کنید که به خاطر وجود TLB ما مثلاً ۱ درصد از اوقات میایم سراغ جداول صفحه.

زمان ترجمه آدرس به ازای هر دسترسی در PT های K سطحی:

KT_m = زمان ترجمه یک آدرس اگر در TLB نباشد

زمان دسترسی به یک آدرس اگر در TLB نباشد: $KT_m + T_m = (K+1)T_m$

جمع بندی: برای کاهش اندازه PT ها همه بیت های P# را که بخشی از آدرس مجازی است را یک جا برای اینکس به PT در نظر نمی گیریم و ابتدا با P_1 بیت با ارزش P# سراغ PT_1 میریم، حالا اگر آدرس در PT_1 استفاده نشده باشد بخش PT_2 را اختصاص نمی دیم

نکته) اگر برنامه از همه فضای آدرس اش استفاده کرده باشد مصرف حافظه روش چند سطحی برای ذخیره PT آن فرآیند بیشتر از روش عادی است.

نکته) در معماری اینتل که از سیستم چند سطحی استفاده می‌شود به سطوح بالای جدول صفحه Directory می گویند.

مثال: فرض کنید آدرس مجازی ماشین ۳۲ بیتی است و سه سطح ۵، ۴، ۳ بیتی برای جداول صفحه چند سطحی ماشین، در بدترین حالت تعداد سطرهای این حالت چند تا از PT تک سطحی بیشتر است؟

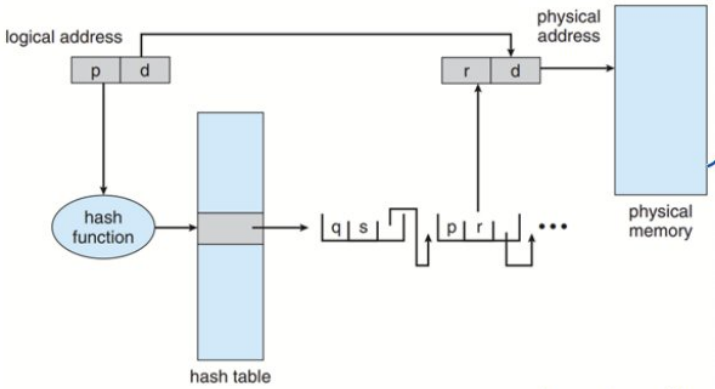
تعداد سطر در حالت ۳ سطحی Max = $32 + 32 \times 16 + 32 \times 16 \times 8 = 32 + 512 + 4096$

پاسخ سوال = $512 + 32 = 544$

روش ۲) Hashed page Tables (جدول صفحه Hash شده)

یک روش برای مدیریت فضاهای آدرس بزرگتر از ۳۲ بیت استفاده از جدول صفحه هش شده است.

به جای استفاده از جدول صفحه چند سطحی از Hash table استفاده می‌کنیم، برای هر فرآیند یک PT با تعداد سطوح مشخص مثلاً ۲۰۴۸ تا در نظر بگیریم و جدهم OS موقعی که PT یک فرآیند رو میخواد بازه یا آپدیت بکنه و هم موقع ترجمه آدرس از hash function استفاده کنند



Hashed page table.

هر سطر درون hash table شامل یک لیست پیوندی از عناصری است که همشون به اون سطر از hash table، هش شدند که هر عنصر سه تا فیلد داره: ۱- شماره صفحه مجازی
۲- P# مربوط به پیج مربوطه
۳- اشاره گر به عنصر بعدی لیست

حالا وقتی آدرس مجازی ای تولید می‌شود با استفاده از hash function میریم سراغ سطر مربوط

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.
Scanned with CamScanner

توی hash table، بعد p# با فیلد اول عنصر لیست مقایسه میشه درصورت تطابق فیلد شماره ۲ اش به سمت چپ offset اضافه میشه و اگر تطابق نداشته باشه میریم سراغ عنصر بعدی لیست، جستجو بصورت خطی انجام می‌شود

چرا باید شماره پیج را در هر عنصر نگه داریم؟ چون تعداد صفحات یک برنامه می‌تواند بیشتر از اندازه hash table ما باشد و بنابراین در هر سطر hash table یک لیست پیوندی داریم، بنابراین باید با توجه به p# توی این لیست جستجو کنیم

روش ۳) Inverted page Tables (جدول صفحه معکوس یا وارونه)

یک جدول صفحه وارونه داریم برای تمامی فرآیندها که به جای اینکه index اش براساس p# باشه براساس F# است!

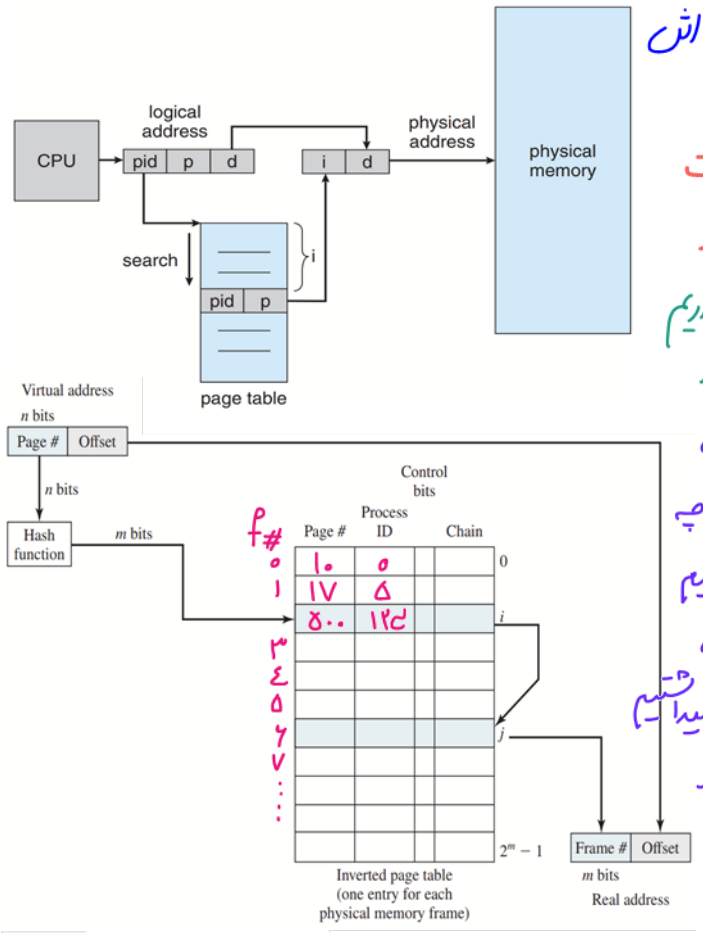
پس تعداد سطرهای این جدول حداکثر به تعداد فریم‌های حافظه است

یعنی اگر RAM، 2^m فریم دارد این جدول هم حداکثر 2^m سطر دارد

چرا می‌گیم حداکثر؟ چون می‌تونیم فقط فریم‌هایی که استفاده شدند رو نگه داریم و پیج‌هایی از فرآیندها که روی‌ها رد هستند اونا که فریمی از RAM اشغال نکردند

خب پس از این کلند رنگه نمیشه و این روش PT مینیمم رو میده، چون توی جدول صفحه ساده همه صفحات ممکن چه آنهایی که استفاده می‌شوند و چه آنهایی که استفاده نمی‌شوند وجود داشت و PT هم فرآیندها رو هم میداشتیم

توی RAM، توی جدول چند سطری هم درسته که فقط صفحات استفاده شده در برنامه کاربر را نگه میداشتیم ولی کمی از صفحات خالی را هم نگه میداشتیم ولی توی این روش می‌توانیم فقط صفحاتی که الان توی RAM هستند (present&) را نگه داریم.



Inverted Page Table Structure

توجه کنید که ما دنبال فریم نمی‌گردیم و دنبال page می‌گردیم، از cpu یک آدرس مجازی مربوط به فرآیند جاری که PID اش برابر a است میاد بیرون و به دنبال این میگرده که این p# در چه فریمی از RAM است، حالا باید بریم و توی جدول صفحه معکوس براساس p# و PID سرچ کنیم و f# رو پیدا کنیم، خب در حالت عادی باید جستجوی خطی انجام بدیم، یک راه برای جستجوی سریعتر استفاده از Hash Function است که ورودی <PID و p#> را می‌گیرد و f# را میدهد.

دانستنی: از آنجایی ممکن است بیش از یک آدرس مجازی به یک سطر Hash table نگاشت شود از تکنیک chaining برای مدیریت overflow استفاده می‌شود، تکنیک هش کردن منجر به chain‌هایی می‌شود که معمولا کوتاه هستند (بین یک تا دو تا)

combined paging and segmentation

برخی از cpu‌ها مانند cpu‌های اینتل اومدن و صفحه بندی و قطعه بندی را با هم ترکیب کردند و سعی کردند مزایای هر ۲ را بدست بیاورند ایده قطعه بندی از این نظر مناسب است که نظر برنامه نویس را از لحاظ نوع نگاه به حافظه برآورده کنیم و ایده صفحه بندی باعث استفاده بهینه از حافظه می‌شود.

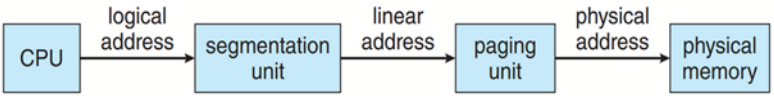
حالا برای این کار میایم و به جای اینکه هر سگمنت را یک حافظه پیوسته در نظر بگیریم و یک حافظه پیوسته به هر سگمنت اختصاص بدهیم، بیایم و هر سگمنت را بشکنیم به page‌ها و page‌ها رو بذاریم توی حافظه و بعد برای هر سگمنت یک PT در نظر بگیریم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

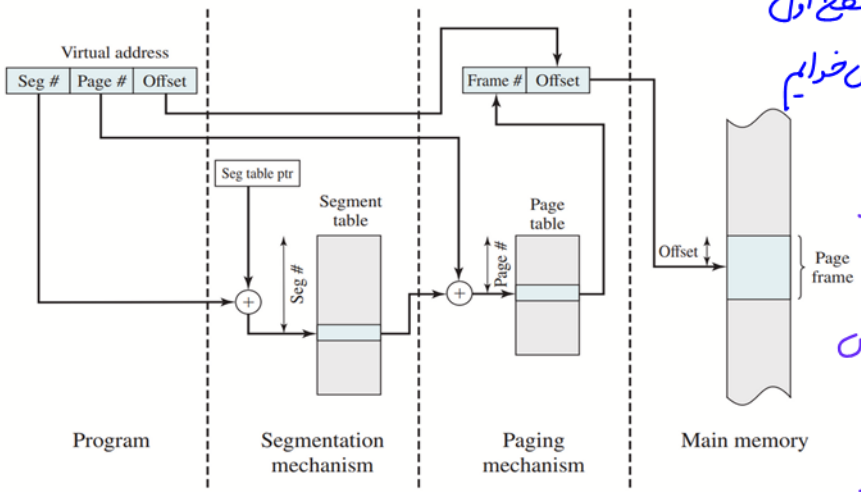
s	g	p

حالا آدرس مجازی ما ۳ قسمت خواهد داشت:

حالا وقتی یک آدرس مجازی از CPU خارج میشد، #seg به جای اینکه ما رو ببره به Base ـ RAM ما رو میبره به Base ـ PT مربوط به خودش و سپس با استفاده از #page و آفست آدرس فیزیکی رو بدست میاریم، به این ترتیب segmentation unit و paging unit با هم دیگه MMU رو به وجود می آورند



حالا چرا این کار را می کنند؟ چون تعداد سگمنت های برنامه محدوده و مثلاً نهایتا ۸ تا یا ۱۶ تا سگمنت دارد، پس مطمئن هستیم که در سطح اول برای هر آدرس مجازی (هر برنامه) ۴ بیت بیشتر سگمنت نمی خوایم



مزایا:

۱) لازم نیست یک سگمنت بصورت بهم پیوسته در RAM قرار گیرد

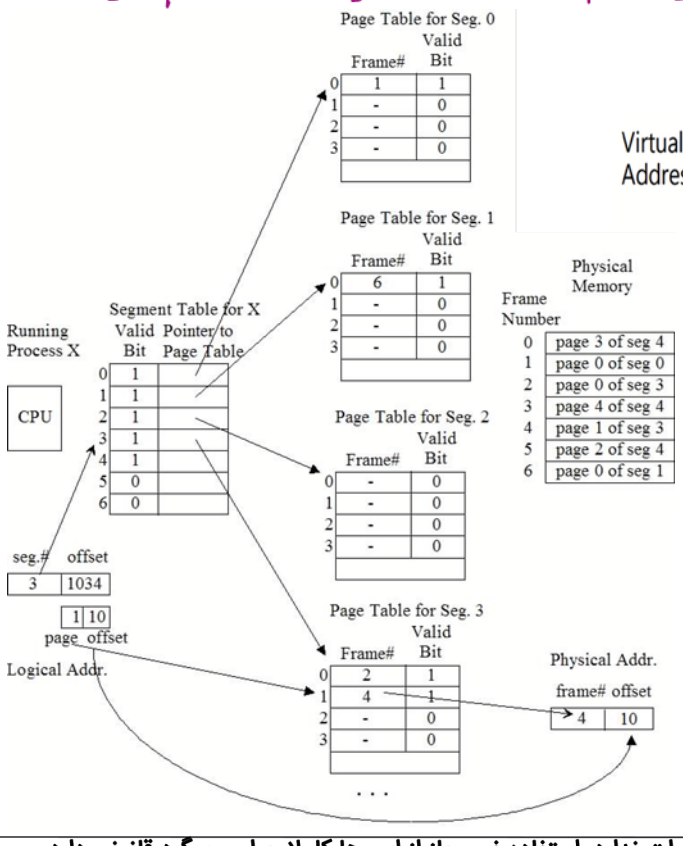
۲) دیگه افزایش اندازه سگمنت ها براحتی انجام می شود

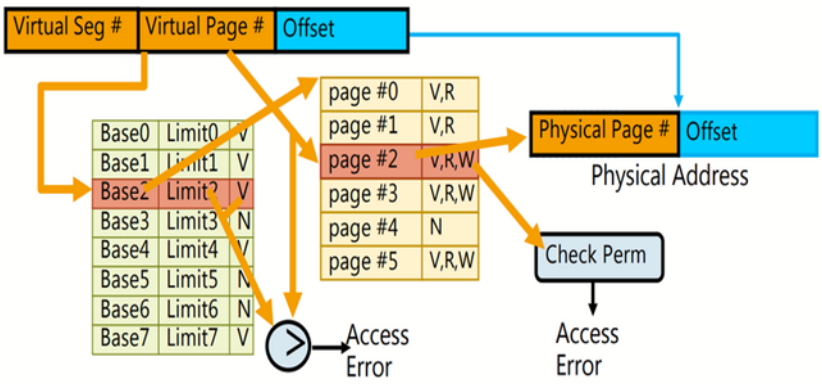
۳) در paging یک برنامه محدودیتی از نظر زیاد کردن صفحات اش تا میزانی که حافظه مجازی بهش اختصاص دادیم نداره ولی با segmentation می تونیم برای هر سگمنت limit بذاریم و در نتیجه تعداد page هایش هم محدود می شود

۴) شماره صفحات هر سگمنت می تواند از ۰ شروع بشود چون جدول سگمنت هر فرآیند آدرس شروع PT های مربوط به هر سگمنت را نشان می دهد

۵) مشکل تکه تکه شدن خارجی که در segmentation داشتیم حل می شود

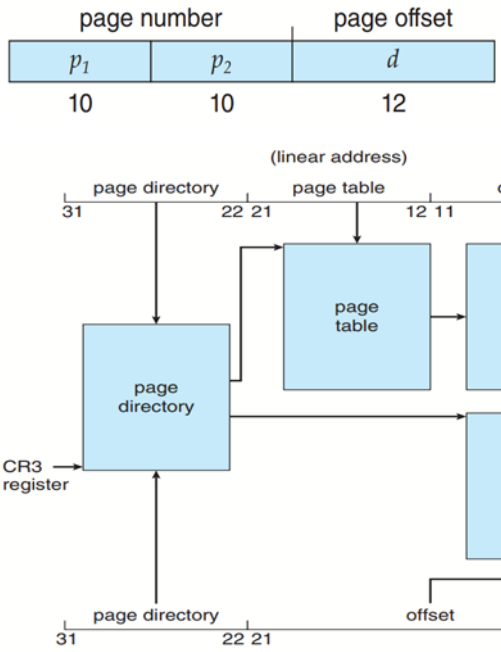
★ پس هر فرآیند یک segmentation table دارد و به تعداد سگمنت هایش PT دارد. حالا توجه کنید که میشه هرکدام از این جدول صفحات را مثلاً ۲ سطحی اش کرد





سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

پردازنده پنتیوم



این معماری امکان variable page size را فراهم می‌کند، پردازنده پنتیوم می‌تواند حافظه ۳۲ بیتی خودش را به ۲ صورت استفاده کند، که این موضوع باعث می‌شود که در پنتیوم بتوانیم pageهای 4KB و 4MB ای داشته باشیم.

۱- به همان صورت قبل که برای ۱۰ بیت شمال زدم با همین اعداد، یعنی ۱۲ بیت بعدان آفست برای pageهای 4KB ای و در تا ۱۰ بیت برای صفحه بندی سلسله مراتبی

۲- یک حالت هم داره که ۲۲ بیت را بعنوان آفست pageهای 4MB ای درنظر می‌گیرد و ۱۰ بیت را برای دسترسی به این pageها

صفحه بندی براساس تقاضا: (on) Demand paging

اگر تعداد فرآیندهای آماده اجرا زیاد باشد هم اونها بطور کامل و بصورت همزمان توی RAM جا نمیشه، حالا بعضا با تعداد زیادی فرآیند آماده اجرا داریم، حالا باید چیکار کنیم؟

۱- یا باید بعضی از برنامه‌ها را Kill کنیم، که خوب نیست

۲- از swapping استفاده کنیم و با استفاده از swapping درجه چندبرنامه‌گی رو ببریم بالا

- روش BfB و segmentation، swapping دارند ولی انعطاف لازم رو ندارند، چون در BfB حافظه فرآیند بصورت پیوسته است و در segmentation هم یک برنامه بعضا به سگمنت‌های هم‌اش برای اجرا شدن نیاز دارد نمی‌ترانیم که مثلاً Data و استک اش را در حافظه بذاریم ولی کد وهیپ اش را نذاریم، بنابراین در BfB و segmentation وقتی می‌خوایم swapping انجام بدهیم بعضا تمام حافظه آن فرآیند را می‌بریم روی هارد و هروقت هم نیاز بود کل حافظه اش رو می‌آوریم توی رم.

حالا paging انعطاف زیادی برای swapping دارد، حالا می‌خوام در این باره صحبت کنم.

Virtual memory

تا حالا وقتی یک فرآیند می‌خواست اجرا بشه ما می‌رفتیم و کل pageهای مربوط به اون فرآیند را می‌آوردیم توی RAM، ولی خب paging این انعطاف را برای ما به وجود می‌آورد که همه حافظه مورد نیاز یک فرآیند را نیاریم توی RAM و فقط pageهایی که فرآیند در همان لحظه نیاز دارد را بیاریم روی RAM.

می‌دانیم که کش لول ۱ نسبت به کش لول ۲ داده‌های دانم تر را نگه می‌دارد (داده‌هایی که مراجعه cpu به آنها بالاتر است) و کش لول ۲ هم داده‌های دانم تری از RAM (یا کش لول ۳) درون خودش نگه می‌دارد

⭐ حال از همین ایده در مورد RAM و هارد استفاده کنیم، یعنی RAM مثل کش برای هارد عمل کند، یعنی صفحات مورد نیاز یک برنامه را به صورت on Demand از هارد به RAM منتقل کنیم.

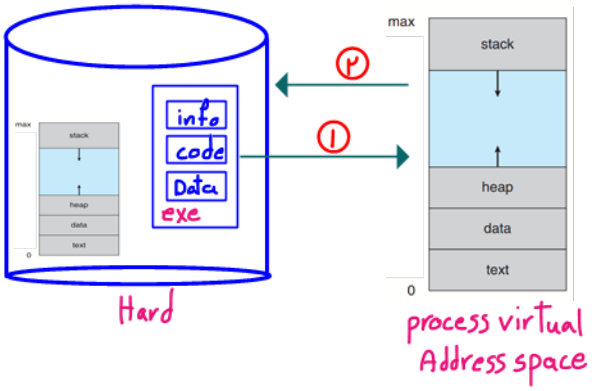
⭐ حالا برای اینکه بتونیم این کار رو بکنیم باید یکسری تغییراتی توی راه و روش قبلی مون بدیم.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

قبلا می‌گفتیم وقتی روی یک برنامه کلیک می‌کنید که بازش کنید os آن برنامه را وارد رم می‌کند و برایش بخش‌های مورد نیازش را می‌سازد اما حالا os به اینصورت عمل می‌کند که os میاد بخش‌های مختلف برنامه را تشخیص می‌دهد و آن را می‌ذاره توی هارد نه توی RAM حالا هرقسمتی که برنامه لازم داشت از هارد به رم منتقل می‌شود و در واقع در این حالت RAM داره بصورت کش برای هارد عمل می‌کند، حالا بازقسمت‌های فارغ RAM می‌تواند وارد کش لایه ۲ شود و به همین ترتیب.

البته توجه داشته باشید که چون مدیریت حافظه من paging است، داده‌ها بصورت page از هارد به رم منتقل می‌شود

یادآوری: فرآیند در یک بازه زمانی مشخص فقط به یک قسمت محدودی از دستورات، داده‌ها، پشته و... خود نیاز دارند و در ۹۰٪ مواقع فرآیندها با ۱۰٪ حافظه مجازی خود کار دارند.



وقتی روی برنامه دابل کلیک می‌کنید os برای فایل exe برنامه VAS اش را می‌سازد و استک و هیپ را به برنامه اختصاص می‌دهد و بعد مجددا این process VAS را توی هارد می‌گذارد، روی هارد یک محدوده مشخص به اسم swap space (فضای مبادله) وجود دارد که در آن محدوده VAS برنامه‌های آماده به کار در آن ذخیره می‌شود.

پس بخش‌هایی از استک و هیپ و دیتا و کد براساس PT فرآیند که نیاز است در RAM قرار می‌گیرند و سایر قسمت‌ها روی swap space است.

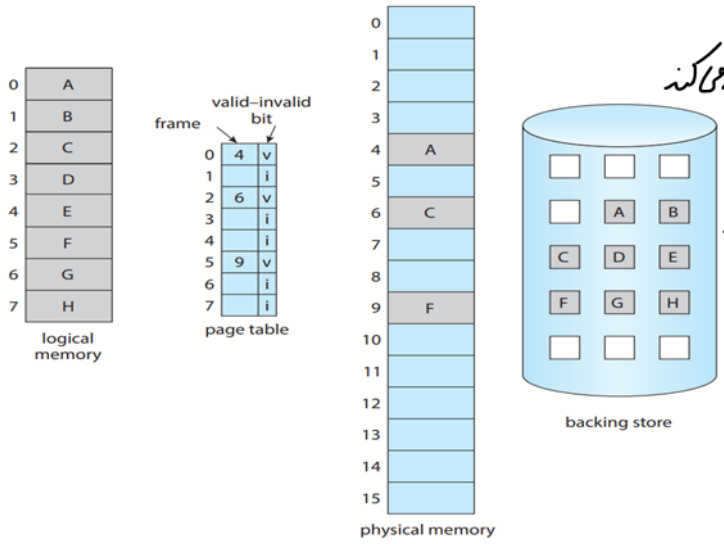
مراحل لود برنامه در RAM

۱- os برنامه را از هارد می‌خواند

۲- os استک و هیپ را به برنامه اختصاص می‌دهد و PCB را مقداردهی اولیه می‌کند

۳- os، VAS را برای فرآیند می‌سازد و آن را روی هارد می‌گذارد

۴- صفحات بخش‌هایی از فرآیند که قرار است اجرا بشه رو میبره توی RAM و PT را براساز

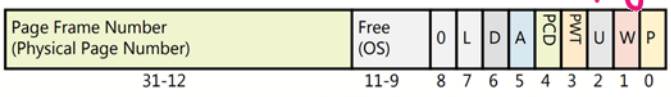


در ابتدا چند تا صفحه بذاریم توی RAM؟

در شکل روبه رو سناریوی که گفته شد را مشاهده می‌کنید.

پیج‌های حافظه منطقی برنامه روی هارد قرار می‌گیرد و از این ۸ تا پیج فقط A و C و F در RAM هستند و در PT فقط سطرهای مرتبط با این ۳ صفحه بیت valid شون ۱ است

page table structure



P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1 ⇒ 4MB page (directory only).

در روبه روساختار یک سطر از PT را مشاهده می‌کنید که بیت‌هایش را قبلا توضیح دادم. (ساختار PT یک پردازنده intel)

توجه) در اینجا بیت valid جدا نداریم و از بیت P بجایش ۲ هم استفاده میشه به اینصورت که اگر Demand paging نداشته باشیم و آدرسی رفت سراغ یک سطر PT و دیدیم که P=0 است یعنی آدرس غلط است و خطا رخ میدهد ولی اگر Dp داشته باشیم P=0 به این معنی است که این صفحه الان توی RAM نیست و برو

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

توی هارد را نگاه کن، حالا می‌ریم توی هارد، اگر توی هارد نبود میاریم اش توی RAM و اگر صفحه درستی نبود این بار خطا به وجود می‌آید

بیت W= یک page قابل نوشتن هست یا نه، مثلاً برای پیج‌های مربوط به کد این بیت ۰ است

بیت present و Absent یا In/out : اگر این بیت ۱ باشد یعنی آن پیج در RAM وجود دارد و اگر ۰ باشد page fault رخ میدهد و باید دیسک بره و page مورد نظر رو از هارد بیاره بذاره توی RAM

بیت A = Referenced : آیا اخیرا به این page مراجعه صورت گرفته یا نه، موقع swapping این بیت به کارمون میاد. هروقت توی یک بازه زمانی به یک سطر مربوط به پیج x از PT دسترسی داشتیم، یعنی با پیج x کار داشتیم و از آن سطر جدول برای ترجمه آدرس استفاده کردیم یا درونش چیزی نوشتیم و یا ازش چیزی خوندیم

بیت D = Modified : page ای که توی RAM هست آیا در مدت حضورش توی حافظه modify شده یا نه. دسترسی که به این پیج مراجعه کرده همشون دسترسات خواندنی بودند یا دستور نوشتنی هم وجود داشته؟

خب در روش صفحه بندی برحسب تقاضا RAM مثل یک کش برای هارد است، پس چون پایگاه اصلی برنامه در این روش هارد است، پس اگر در RAM محتوای صفحه‌ای عوض شده باید بریم در هارد هم مقدار آن را درست کنیم. شبیه بیت Dirty درکش

روال صفحه بندی برحسب تقاضا

۱- یک برنامه یک آدرس تولید و می‌ریم سراغ PT، اگر بیت valid اش ۰ باشد که آدرس غلط و برنامه terminate میشه ولی اگر ۱=V باشد و P اش ۰ است (MMU متوجه می‌شود که پیج مربوط توی RAM نیست)

۲- MMU خطای page fault تولید می‌کند و ادامه اجرا به OS واگذار میشه

۳- OS روتین page fault را اجرا می‌کند و فرآیند را به وضعیت بلاک می‌برد و می‌رود دنبال صفحه مورد نظر را از هارد می‌خواند ← سیستم کال + وقفه

خواندن از هارد یک سیستم کال است و بعد منتظر وقفه هارد می‌شیم یعنی هارد که خواندنش تموم شد وقفه می‌دهد

۴- وقتی صفحه آماده شد حالا باید این را تو RAM کپی کنیم

۴-۱: OS باید در RAM یک فریم خالی پیدا کند

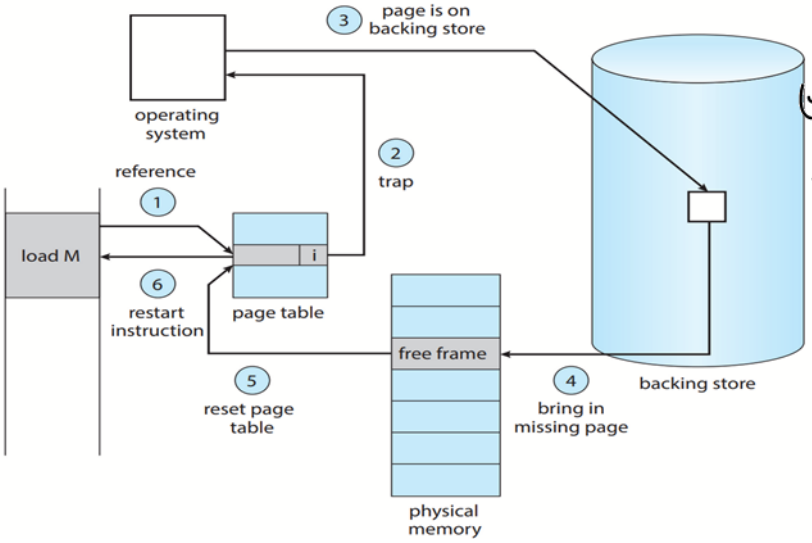
۴-۲: اگر فریم خالی در RAM وجود ندارد باید یکی از فریم‌های درون رم را انتخاب کنیم. حالا آیا اون را به هارد منتقل کنیم؟

نه لزوما. بستگی به بیت Dirty دارد.

۴-۳: اگر فریمی را از رم انداختیم بیرون PT مربوط به فرآیندش را باید آپدیت کنیم و مثلاً P اش را ۰ کنیم یا اگر نوشتیم اش توی هارد Dirty اش رو ۱ کنیم

۵: PT مربوط به صفحه‌ای که بردیم توی رم را آپدیت کنیم و P اش را ۱ می‌کنیم

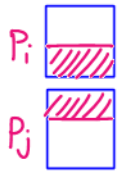
۶: فرآیندی که بلاک شده بود را ready می‌بریم تا دوباره از دستور قبلی که PF خورده بودیم به خاطرش شروع کند به اجرا نکته) وقتی برای فرآیندی PF رخ می‌دهد باید به وضعیت بلاک برود و فرآیند دیگری را بیاریم روی CPU پس CS خواهیم داشت.



سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

آیا اجرا دوباره دستورات راحته؟

نکته) اینکه یک دستور رو بخوایم از اول اجرا کنیم همیشه کار راحتی نیست، این موضوع به ویژه در دستوراتی که به چندین مکان حافظه کار دارند سخت است، مثلاً دستور strcpy را که یک قسمت از حافظه را در قسمت دیگری از حافظه کپی می‌کند را در نظر بگیرید حالا فرض کنید آدرس مقصدی که باید داده مبدا را در آن کپی کنیم شامل چندین صفحه می‌شود که مثلاً چند تا صفحه اول مقصد در رم هست ولی بقیه صفحات توی رم نیست، پس قسمتی از داده‌ها را کپی می‌کنیم و وسط کپی کردن PF رخ می‌دهد. حالا تا OS بره صفحه‌ای که PF خورده رو بیاره شاید محتوای مبدا عوض شده باشد، یا کسی از مقصد ناصحیح استفاده کند، پس اوضاع خراب میشه



روش‌هایی که برای حل این مشکل وجود دارد

۱- در بعضی از معماری‌ها از دستوراتی که این مشکل را به وجود می‌آورند ۲ نسخه وجود دارد، مثلاً strcpy و strcpytest، حالا میان اول strcpytest را اجرا می‌کند ولی هیچ داده‌ای از مبدا به مقصد کپی نمی‌شود و هدف از اجرا این دستور این است که پیج‌هایی که در رم نیستند PF بخورند و قبل از اجرا دستور اصلی در رم لود بشوند و بعد دستور اصلی را اجرا می‌کند اینگونه وسط اجرا دستور اصلی PF رخ نمی‌دهد.

۲- در بعضی از معماری‌ها قبل از اجرا چنین دستوراتی مقادیر قبلی حافظه‌ای که قرار است تغییر کند را در یک حافظه موقت دیگر ذخیره می‌کنند و بعد دستور را اجرا می‌کنند، حالا اگر دستور تا انتها اجرا شد و PF نخورد که هیچی ولی اگر PF خورد، حافظه دست خورده را به حالت قبل برمی‌گردانند

۳- در بعضی از معماری‌ها برعکس بالا رفتار می‌کنند، ابتدا خروجی دستور را در یک حافظه موقت که مطمئن هستند PF رخ نمی‌دهد کپی می‌کنند و بعد صفحات مقصد را هم آماده می‌کنند و بعد آنها را در مقصد کپی می‌کنند

۴- در سیستم‌های کنونی از مکانیزم precise exception (خطاهای دقیق) استفاده می‌کنند که میاد اگر دستوری باعث این اتفاقات شد و در اثر آن خطایی رخ داد، آن دستور و دستورات بعدی آن را به گونه‌ای اجرا می‌کند که گویی هرگز اجرا نشده‌اند، که این روش در معماری‌های دارای پایپلاین و out of order execution پیچیده است.

۵) مکانیزم خطاهای غیردقیق: کار شماره ۴ بصورت نرم افزاری.

محاسبه زمان موثر دسترسی در صورت رخ دادن PF
effective access time of Demand paging

p: احتمال رخ دادن PF : احتمال آنکه پیجی که می‌خواهیم در رم نباشد

ma : memory access time = ۲۰۰ ns

page fault service time : ۸ms : شامل مراحل ۲ تا ۶ بالا

$$EAT = (1-p) \times ma + p \times \text{page fault time} = ma - p \times ma + p \times PF\,time = ۲۰۰ns - p \times ۲۰۰ns + p \times ۸۰۰۰۰۰۰ns$$

$$= (۲۰۰ + ۷۹۹۸۰۰ \times p)\,ns$$

حالا فرض کنید، $p = ۰/۰۰۱$ باشد، یعنی از هر ۱۰۰۰ دسترسی به حافظه فقط یک PF داشته باشیم : $EAT = ۸/۲\,\mu s$

که این عدد حدوداً ۴۰ برابر ma است که عدد بزرگی است. حالا اگر بخواهیم PF فقط ۱۰٪ زمان effective ما رو زیاد کند PF باید

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

چقدر باشد؟ $200 + 7998000p < 220 \Rightarrow 7998000p < 20 \Rightarrow p < 0.0000028 \Rightarrow p < 2.8 \times 10^{-6}$

و این بدان معناست که حدوداً باید از هر ۴۰۰ هزار (۳۹۹۹۹۰) پیجی که می‌خواهیم یکی‌اش در رم نباشد

پس باید مکانیزم‌هایی را در os بکار ببریم که این اتفاق بیفتد

حالا چه کارهایی برای بهینه‌سازی Demand paging انجام بدهیم؟

موقع PF مجبوریم از هارد بخونیم و شاید نیاز باشه هم توی هارد پیجی که از رم اخراج کردیم رو بنویسیم و هم پیج خودمان را بخوانیم، حالا برای کاهش این زمان چه کارهایی می‌توانیم بکنیم؟

۱- برای درخواست‌های swap space اولویت بالاتری نسبت به سایر درخواست‌ها از هارد (مثلاً اگر در یک زمان ۵ تا درخواست خواندن از هارد وجود دارد و یک درخواست خواندن از swap space به os داده شد، os آن را در اولویت قرار بدهد)

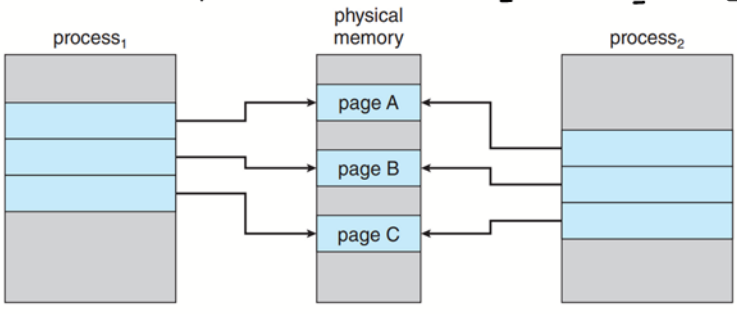
۲- یک فرآیند چه بخواد یک پیج از هارد بخواند چه بیشتر در هر صورت به حالت بلاک می‌رود و منتظر وقفه گذاشتن آن داده در RAM می‌شود، حالا طبق لوکالیتی مکانی بیایم هر بار که فرآیند قراره برای PF به حالت بلاک بره مقدار بیشتری از داده‌های مربوط به آن فرآیند را از هارد بیاریم توی رم، البته خیلی هم نمی‌تونیم اندازه داده‌ای که می‌یاریم (chunck) را بزرگ بگیریم چون باعث کندی میشه

۳- در os های فعلی کل VAS یک فرآیند در swap space قرار نمی‌گیرد و بخش‌هایی از آن که استفاده می‌شود به تدریج درون swap space کپی می‌شود، حالا یک کار میتونه این باشه که کل VAS یک فرآیند را در swap space قرار بدهیم

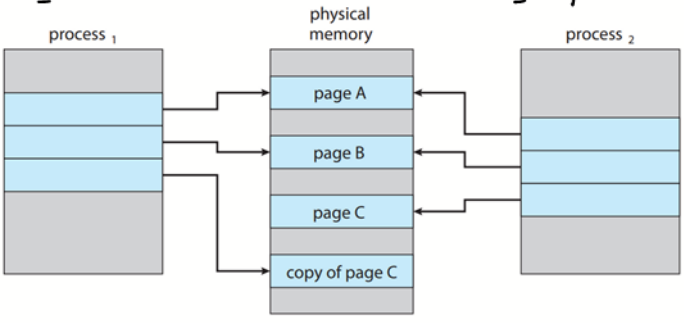
(توجه) os برای کار با swap space بهینه‌سازی شده و انتقال داده از swap space به رم سریعتر از file system های عادی در هارد است.

(نکته) os های موبایل معمولا swap space ندارند، چون همان طور که قبلا گفتم توی موبایل ها معمولا swapping نداریم.

۴- استفاده از copy on write (cow) : وقتی فرآیند پدر فرآیند فرزندی را با fork ایجاد می‌کند، Address space ـ پدر برای فرزند کپی نمی‌شود و فقط pcb فرآیند فرزند توسط os ساخته می‌شود و تا زمانیکه نخواهد چیزی در این فضا بنویسد از همان فضای آدرسی پدر استفاده می‌کند



Before process 1 modifies page C.



After process 1 modifies page C.

۵- با استفاده از مکانیزم‌های پیش‌بینی صفحاتی که فرآیند در ادامه کارش نیاز دارد و prefetch کردن آن‌ها کاری کنیم که PF کمتر رخ دهد

دستورات prefetch توسط کامپایلرها و برنامه‌نویس‌ها می‌تواند انجام شود

__builtin_prefetch(const void *addr, int rw, int level) → intel Xeon

level [0..3] : مشخص کردن اینکه در کدام لول کش لود شود ۳: load in all cache level

وقتی می‌گیم دیتا وارد کش بشود وارادش کردیم توی رم هم بیاید (DMA این کار را انجام می‌دهد)

۶- افزایش اندازه RAM

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۷- تنظیم تعداد فرآیندهای بهینه‌ای که باید توی رم باشند ← اگر تعداد فرآیندها خیلی زیاد باشد به هر فرآیند تعداد کمی پیج می‌رسد و این موضوع نیز باعث PF فرآیندها می‌شود

۸- تنظیم تعداد صفحاتی که به فرآیندهای مختلف باید اختصاص بدهیم

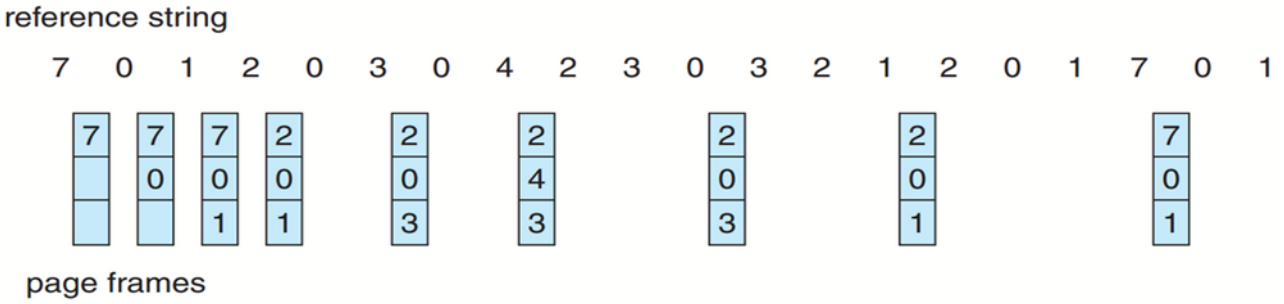
۹- انتخاب سیاست جایگزینی صفحه مناسب ← اگر رم جا نداره و PF رخ داده و باید برای پیج جدید یک فریم را از رم اخراج کنیم مثلاً صفحه‌ای که در آینده نزدیک مورد نیاز است را از توی رم نندازیم بیرون، چون اگر چنین صفحه‌ای رو از توی رم بندازیم بیرون خب چند لحظه بعد که می‌خوایم اش باید دوباره بریم از توی هارد بیاریم اش و این هزینه مضاعف برای ما به همراه دارد. (طبق اصل مراجعات محلی) یا مثلاً بین یک صفحه‌ای که بیت dirty اش ۰ است یا ۱ بهتره کدوم رو برای بیرون انداختن از رم انتخاب کنیم؟

الگوریتم‌های جایگزینی صفحه

۱- optimal: بهینه (Belady optimal: Bo) (روشن کسینه: Minimum: Min)

صفحه‌ای را خارج کنیم که در آینده دورتری بهش مراجعه می‌شود

- چون این الگوریتم نیاز به پیش بینی آینده دارد و ما از آینده خبر نداریم این الگوریتم قابل پیاده سازی نیست.
- از این الگوریتم برای مقایسه میزان خوبی سایر الگوریتم‌ها استفاده می‌شود
- پیدا کردن پیجی که در آینده دورتری مورد استفاده قرار می‌گیرد یک مسئله Np-complete است و در زمان چند جمله‌ای قابل حل کردن نیست



۲- FIFO (صف)

همیشه قدیمی‌ترین صفحه درون حافظه را خارج کن، از بین کسایی که توی حافظه هستند اونی که زودتر از بقیه آمده

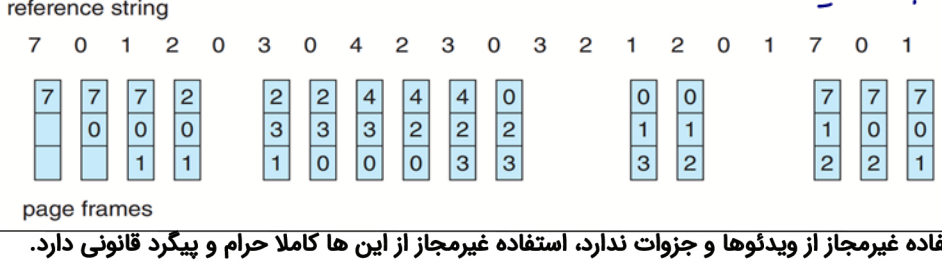
پیاده سازی این الگوریتم ساده است و با یک لیست پیوندی پیاده سازی می‌شود

مثال: فرض کنید ۴ تا فریم داریم و به ترتیب پیج‌های روبه رو را وارد حافظه کردیم ۱۲۰ → ۲۳ → ۷۸ → ۹۰

۲۳ → ۷۸ → ۹۰ → ۱۰ → ورود ۱۰: برای پیج ۱۰، PF رخ می‌دهد

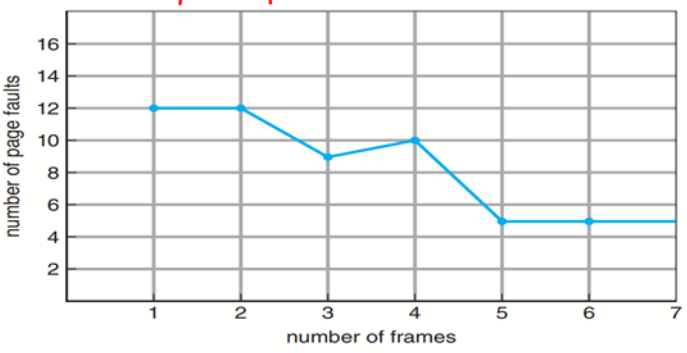
به ازای هر بار PF این الگوریتم اجرا می‌شود و به ازای هر مراجعه نیازی نیست کاری انجام بدهد

خوبه به نظرتون؟ مثلاً خوبه که صفحات مربوط به cpu scheduler رو بندازیم بیرون؟ یا مثلاً یک صفحه مربوط به یک کتابخانه را در ابتدای بوت شدن سیستم گذاشتیم توی رم که خیلی از برنامه‌ها مدام دارن بهش دسترسی پیدا می‌کنند، حالا این رو بندازیم بیرون؟ زود آمدن دلیل بر کم استفاده بودن نیست



سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

★ FIFO، Belady's Anomaly دارد، یعنی در برخی از حالت‌های خاص با افزایش تعداد فریم‌های رم نرخ PF به جای کمتر شدن بیشتر می‌شود (Belady malfunction)



Page-fault curve for FIFO replacement on a reference string.

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

۱	۵	۴	
۲	۱	۵	
۳	۲		
۴	۳		

۱۰PF

۱	۴	۵	
۲	۱	۳	
۳	۲	۴	

۹PF

بطور معمول انتظار داریم با افزایش تعداد فریم‌های حافظه (افزایش حجم حافظه) تعداد خطاهای صفحه کاهش یابد، اما در برخی الگوریتم مثل FIFO اینگونه نیست

نکته) الگوریتم‌هایی که ویژگی پشته (stack) دارند و چار این بد رفتاری نمی‌شوند مثلاً LRU و optimal ناهنجاری بلیدی ندارند

الگوریتم‌های پشته الگوریتم‌هایی هستند که در آنها مجموع صفحات در رم وقتی n فریم در اختیار دارند همواره زیرمجموعه‌ای از صفحات موجود در رم به ازای n+1 فریم باشند

$P(n) = \{P_1, P_2, \cdots, P_n\}$

$P(n) \subseteq P(n+1)$

P(n): مجموعه شماره صفحاتی که الان در رم است

حالا تعداد فریم‌ها را یکی زیاد می‌کنیم می‌کنیم n+1، حالا اگر $P(n) \subseteq P(n+1)$ بود الگوریتم پشته‌ای است، یعنی هر پیجی که با n تا فریم در رم بوده همان صفحات در n+1 فریم هم حضور داشته باشند (علاوه یک پیج دیگه)، خب در این حالت امکان ندارد که با داشتن n+1 فریم PF ای رخ بدهد که در n فریمی رخ نداده، چون مجموعه صفحات درون n+1 فریم کامل‌تر است و شامل صفحات n فریمی که هست و یکی هم بیشتر دارد، بنابراین مشخص که هیچ وقت PF در حالت n+1 فریمی بدتر از n فریمی بیشتر نمی‌شود

حالا چرا LRU اینجوریه؟ چون LRU میاد و صفحاتی که اخیرا استفاده شده را در حافظه نگه میدارد، مثلا فرض کنید الان به پیج x ارجاع داده شده و رم ۳ تا فریم داره، حالا توی LRU حتما توی رم x و y و z است، حالا اگر Reference: n m z y x

۴ تا فریم داشته باشه رم توی رم x و y و z و m است، همینه هم استدلالش شبیه اینه ولی توی FIFO چون بر اساس زمان ورود replace میکند باعث میشه مجموعه صفحات توی رم در n فریم بتواند متفاوت از n+1 فریم باشد

۳- روش تصادفی

پیاده‌سازی ساده دارد ولی نمی‌توانیم رفتار سیستم را پیش بینی کنیم و یک برنامه ممکن است در اجراهای مختلف کارایی‌های مختلفی داشته باشد

- در حافظه‌ها و حافظه‌های کش کوچک و TLB در برخی از سیستم‌ها استفاده می‌شوند.

۴- LRU (least recently used) (توجه) این را نگیم اخیرا کمتر استفاده شده چون به اشتباه به تعداد اشاره می‌کند

صفحه‌ای را از حافظه خارج کن که از آخرین استفاده‌اش بیشترین زمان گذشته باشد (صفحه‌ای را از حافظه خارج کن که در گذشته دورتری به آن مراجعه شده)

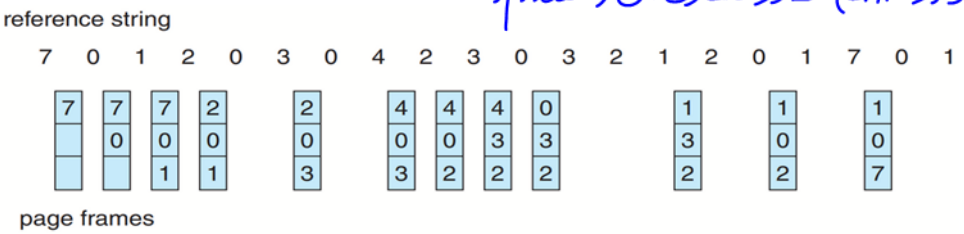
با توجه به اینکه در رفتارش پردازنده LOR زمان وجود دارد، بهتر است برای replace کردن عنصر درون رم، پیج LRU را replace کنیم (یعنی صفحه‌ای که در گذشته اخیر استفاده شده احتمالا در آینده نزدیک هم استفاده میشه)

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

- LRU در واقع یک تقریب برای پیاده سازی روش optimal است.

نکته) پیاده سازی الگوریتم بعیند که غیر ممکن است، پیاده سازی LRU نیز خیلی پرهزینه است و برای حافظه رم در عمل استفاده نمیشه و پیاده سازی اش به شکل دقیق امکان پذیر نیست و از روش های تقریبی برای پیاده سازی آن استفاده می کنیم که الگوریتم Aging یک نمونه مهم آن است.

تذکر) الگوریتم LRU در کش استفاده می شود (مثلا در CAM) که در معماری آن را گفته ام.



پیاده سازی LRU بصورت سخت افزاری: چون نرخ مراجعه به هر صفحه در حد نانوثانیه باید بصورت سخت افزاری پیاده شود

۱- counters: بیایم و یک ساعت در سیستم قرار بدهیم و بعد به page ها براساس زمان دسترسی بهشون، بهشون lable بزنیم، این ساعت باید دقت اش در حد ns باشه زیرا دسترسی های cpu به پیج ها با سرعت بالایی انجام میشه.

حالا به چند دلیل این کار را انجام نمی دهیم:

۱- این روش نیازمند پیاده سازی یک ساعت بسیار دقیق دارد

ما باید مثلاً یک شمارنده ۶۴ بیتی در نظر بگیریم که تعداد دستوراتی که اجرا شده است را بصورت سخت افزاری بشماریم، خب خود این همون شبیه زمان است و هدف این عدد کوچکتر باشه یعنی گذشته دورتر. (۳۲ بیت کم است چون حدوداً ۴ میلیارد دستور را می شمارد، خب فرکانس پردازنده الان حدود ۴، ۵ گیگا هرتز است و بنابراین در مثلاً ۴ ثانیه کانتر مان ریست می شود)

۲- باید به هر صفحه ای که مراجعه می شود بریم و این lable زمان را در PT آن صفحه ذخیره کنیم، حالا خود این ۲ تا مشکل دارد

۱- همین جوریش ما با PT مشکل داشتیم و گفتیم بزرگه و چند تا روش برای مقابله با آن داریم، حالا اگر یادتون باشه گفتم بعهره هر سطر PT، ۳۲ بیت (۴ بایت) است، حالا ما باید به هر سطر ۶۴ بیت فقط برای این lable اضافه کنیم که باعث میشه PT سه برابر بشود

۲- باید از وقتی که cpu به یک پیج مراجعه می کند تا مراجعه بعدی اش که در هر چندین نانوثانیه است بریم و این لیبل را در PT این فرآیند که در رم است بنویسیم، یعنی ۸ بایت را از کانتر بخوانیم و آن را در رم بنویسیم، حالا read کردنش از توی رجیستر ولی رایت کردنش خیلی کنده ← غیر قابل تحمل

۳- حالا فرض کنید هم این کار ها را کردیم، حالا هروقت PF رخ میده باید بریم و لیبل تمام صفحه های توی رم رو چک کنیم و کوچکترین اش را ببریم بیرون، حالا چون PF هر چند میلی ثانیه رخ میده این مورد قابل تحمله

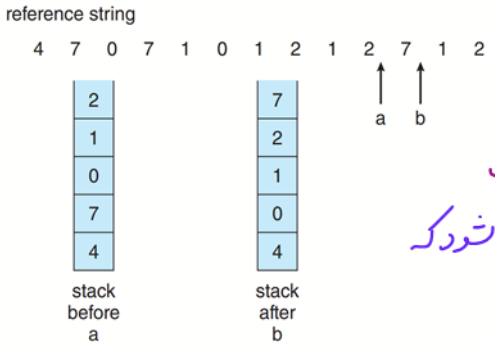
۲- روش دوم: stack (لیست پیوندی)

ما برای اینکه بفهمیم عنصر LRU چه کسی است نیازی نداریم که زمان دسترسی ها را بدانیم، بلکه فقط کافی است ترتیب دسترسی ها را بدانیم، بدین ترتیب سربار به شدت کاهش پیدا می کند و در این حالت برای هر صفحه کافی است فقط به تعداد log تعداد فریم های رم بیت نگه داریم

یک پشته با استفاده از لیست پیوندی ۲ طرفه پیاده سازی می کنیم و هر بار که صفحه ای استفاده می شود، چه خواندن باشد و چه

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

نوشتن آن صفحه به ابتدای لیست می‌آید و آخرین عضو این لیست (عضو انتهای پشته) همان کاندید جایگزینی است، پس برای یافتن جایگزین نیاز به جستجو ندارد



۴ ↔ ~~۷~~ ↔ ۰ ↔ ۱ ↔ ۲ ↔ ۷

به این ترتیب همیشه یک لیست سورت شده بر اساس recent Access ها داریم و همان طور که در مثال بالا می‌بینید ۴ از همه دورتر استفاده شده و کاندید جایگزینی است

نکته) به ازای هر بار مراجعه به حافظه که در حد ns است باید این لیست پیوندی آپدیت شود که در بدترین حالت در هر بار باید ۶ اشاره‌گر عوض شود

۳- روش سوم: پیاده‌سازی با ماتریس

در ابتدا کل ماتریس ۰ است، حالا هر وقت به فریم شماره x رجوع می‌شود ابتدا همه سطرهای x ام را ۱ می‌کنیم و سپس همه ستون‌های x ام را ۰ می‌کنیم

Reference: ۰, ۱, ۲, ۳, ۲, ۱, ۰, ۳, ۲, ۳

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

(b)

	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

(c)

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

(d)

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	1
3	1	1	0	0

(e)

	0	1	2	3
0	0	0	0	0
1	1	0	1	1
2	1	0	0	1
3	1	0	0	0

(f)

	0	1	2	3
0	0	1	1	1
1	0	0	1	1
2	0	0	0	1
3	0	0	0	0

(g)

	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	0
3	1	1	1	0

(h)

	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	1	1	0	1
3	1	1	0	0

(h)

	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	1	1	0	0
3	1	1	1	0

(i)

→ ۱ کوچکترین عدد است

→ ۳ بزرگترین عدد را دارد

فرض کنید حافظه ۸GB باشد و اندازه صفحه ۴KB باشد، حجم این ماتریس چقدر می‌شود؟

$$\text{تعداد فریم‌های رم} = \frac{2^{3} \times 2^{30}}{2^{12}} = 2^{21}$$

$$\text{حجم ماتریس} = \frac{2^{21} \times 2^{21}\,\text{bit}}{2^{3}} = 2^{39}\,\text{byte} = 512\,\text{GB}$$

زمانش هم افتضاحه: هر سطر و هر ستون چند بایت است؟ $\frac{2^{21}}{2^{3}} = 2^{18}\,\text{B}$

حالا ما باید به ازای هر مراجعه به هر صفحه $2^{18}\,\text{B} = 256\,\text{KB}$ را ابتدا یک کنیم و بعد ۲۵۶KB رو بریم ۰ کنیم که اگر هم سطری ذخیره کردیم باشیم توی حافظه انجام این کار بصورت ستونی هم خودش داستان داره.

می‌خواهیم با هزینه کمتری نسبت به LRU به صفحات اخیرا استفاده شده توجه کنیم

LRU-Approximation page Replacement

۵- Additional Reference Bit (الگوریتم Aging: مسالمندی): تخمینی از LRU

مثلاً ۸ بیت اضافه در PT در نظر بگیریم، حالا در فواصل زمانی منظم مثلاً هر ۱۰۰ میلی ثانیه وقفه ساعت کنترل را به OS می‌دهد و OS میاد و برای تمامی صفحات موجود در رم بیت R را از سمت چپ این ۸ بیت وارد می‌کند و سمت چپ‌ترین بیت را دور می‌اندازد و بیت R این صفحات را نیز ۰ می‌کند، حالا این ۸ بیت تاریخچه استفاده از هر صفحه را برای ۸ دوره زمانی گذشته نشان می‌دهد، مثلاً اگر این ۸ بیت ۰۰۰۰۰۰۰۰ باشد یعنی برای ۸ دوره زمانی استفاده نشده. صفحه‌ای که حداقل یکبار در هر دوره استفاده می‌شود دارای مقدار

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۱۱۱۱۱۱۱۱ است. حالا هر وقت PF رخ داد بیایم و اون صفحه‌ای را بندازیم بیرون که این ۸ بیت اش از همه کوچکتره و صفحه جدید رو بیاریم جای این صفحه و این ۸ بیت را برای صفحه جدید ۰ می‌کنیم و بیت R و P این صفحه‌ای که آوردیم توی رم را ۱ می‌کنیم

نکته) در الگوریتم سالمندی به این ۸ بیت Age می‌گویند

✅ اگر چندین صفحه مقادیر Age برابری داشتند چیکار کنیم؟

۱✓ ـ همه این صفحات را swap out کنیم

۲✓ ـ از روش FIFO برای انتخاب از بین اینها استفاده کنیم } ۱ یا ۲ بیشتر مد نظر طراح است

۳ـ رندم یکی رو بندازیم بیرون

⭐ مشکلات این الگوریتم

۱ـ نیاز به سرچ بین مقادیر Age و وجود دارد

۲ـ مقادیر برابر باعث میشه این الگوریتم تخمینی از LRU شود. چرا؟

۱ـ قبل از ۸ دوره زمانی را در نظر نمی‌گیرد

۲ـ تعداد مراجعات در یک بازه زمانی را در نظر نمی‌گیرد

۶ـ NRU (Not Recently used)

صفحه‌ای را از حافظه رم خارج کنیم که در گذشته نزدیک استفاده نشده است. (کم و زیاد نمی‌گیم)

در NRU از بیت R استفاده می‌شود، به هر صفحه‌ای که مراجعه می‌شود MMU بصورت اتوماتیک این بیت را ۱ می‌کند

بیت A = Referenced : آیا اخیرا به این page مراجعه صورت گرفته یا نه، موقع swapping این بیت به کارمون میاد. هر وقت توی یک بازه زمانی به یک سطر مربوط به پیج n از PT دستبرد داشتیم، یعنی با پیج n کار داشتیم و از آن سطر جدول برای ترجمه آدرس استفاده کردیم یا درونش چیزی نوشتیم و یا ازش چیزی خوندیم

نحوه عملکرد این الگوریتم: یک دوره زمانی و یک پریود تعریف می‌کنیم که ابتدای هر دوره زمانی (مثلاً ۱۰۰ ms) بیت R همه صفحات درون رم را در PT شان ۰ می‌کند، حالا هر وقت به هر صفحه مراجعه شود MMU بیت R آن صفحه را ۱ می‌کند، حالا اگر رم پر بود و PF رخ داد و خواستیم یک صفحه‌ای را خارج کنیم از بین صفحاتی که بیت R شون ۰ است یکی را خارج می‌کنیم.

۷ـ Advanced/enhanced NRU: همان NRU است فقط بیایم در شرایط برابر بیت Dirty رو هم در نظر بگیریم

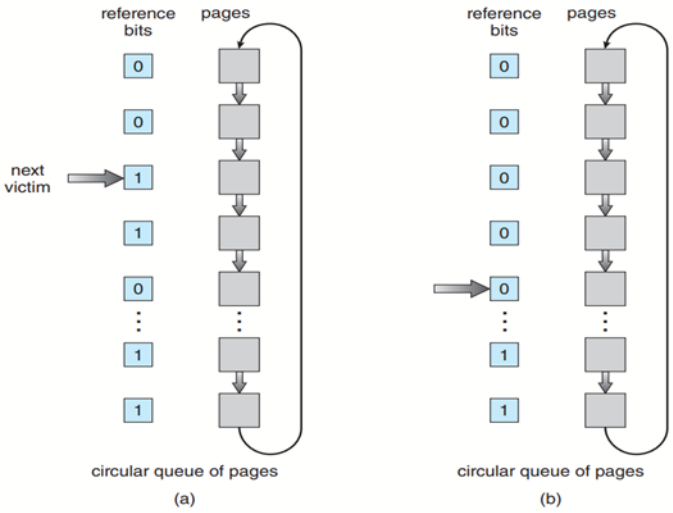
وقتی PF رخ می‌دهد OS از بین صفحاتی که در کلاس پایین‌تری هستند OS یکی را بصورت تصادفی انتخاب می‌کند

	class	R	M
Class 0: not referenced, not modified.	۰	۰	۰
Class 1: not referenced, modified.	۱	۰	۱
Class 2: referenced, not modified.	۲	۱	۰
Class 3: referenced, modified.	۳	۱	۱

۸ـ second chance (دومین شانس، شانس دوباره)

بریم ایده NRU و FIFO را با هم ترکیب کنیم، در NRU از بین صفحاتی که R شون ۰ بود یکی رو رندم میانداختیم بیرون ولی رندم چیز خوبی نیست و بیایم از بین صفحاتی که استفاده نشدند قدیمی‌ترین رو بندازیم بیرون (در FIFO می‌گفتیم شاید قدیمی باشد ولی زیاد استفاده بشه ولی اینجا دیگه چون R شون رو چک می‌کنیم مطمئن هستیم استفاده نشده

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.



Second-chance (clock) page-replacement algorithm.

روش کار الگوریتم: مثل FIFO برای صفحاتی که وارد رم می‌شوند یک لیست پیوندی ساخته می‌شود و هر وقت PF رخ داد می‌ریم سراغ ابتدای لیست پیوندی (قدیمی‌ترین صفحه)، بعد به بیت R هر صفحه نگاه می‌کنیم، اگر ۰ باشد آن صفحه را از رم خارج می‌کنیم (چون هم قدیمی است و هم بدون استفاده) و صفحه جدید را در رم جای آن قرار می‌دهیم و بیت R این صفحه را ۱ می‌کنیم و لیست پیوندی را نیز آپدیت می‌کنیم

چگونه؟ صفحه ابتدا لیست را خط می‌زنیم و صفحه جدید را به انتها لیست اضافه می‌کنیم

اما اگر بیت R صفحه ابتدا FIFO برابر ۱ بود آن را ۰ می‌کنیم و صفحه را به انتها لیست پیوندی می‌بریم و به اینصورت به این صفحه شانس مجدد داده می‌شود ← چون در این مدت دور زدن ممکن است صفحاتی که R شون رو ۰ کردیم دوباره استفاده بشند و R شون ۱ بشه

مثال: فرض کنید یک PF برای صفحه ۱۲۳ رخ داده است و رم ما ۴ فریم دارد و FIFO بصورت زیر است:

- OS این کارها را انجام می‌دهد
- MMU فقط به ازای مراجعه به هر صفحه بیت R آن صفحه را تغییر می‌دهد

R=1 R=1 R=0 R=0 R=0 R=0 R=1

~~۲۳~~ → ~~۶۲~~ → ~~۱۴~~ → ۱۰ → ۲۳ → ۶۲ → ۱۲۳

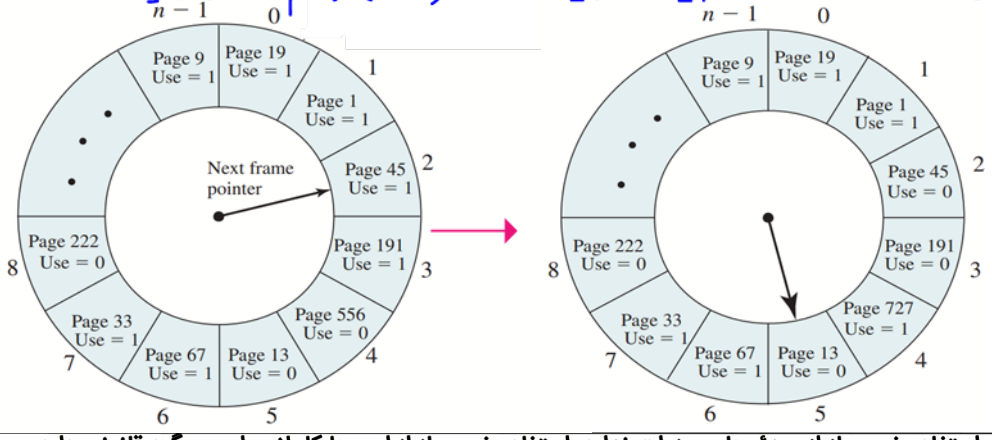
★ اگر بیت R همه صفحات ۱ باشد Degenerate می‌شود به FIFO

چون از سر صف به هر کی میرسه بیت R شون که ۱ است را ۰ می‌کند و می‌گذارد انتهای صف تا دوباره می‌رسیم به همان صفحه اول

خب این روش پیاده‌سازی کمی پرهزینه است چون باید همه‌اش پوینترها را آپدیت کنیم، البته توجه کنید که به ازای هر PF این کارها انجام می‌شود نه به ازای هر دسترسی به صفحه

یکی از روش‌های پیاده‌سازی دومین شانس این است که لیست پیوندی را بصورت یک صف حلقوی (صف دایره‌ای) پیاده‌سازی کنیم که گاهی از آن بعنوان clock Algorithm یاد می‌شود و اشاره‌گر (همان عقربه روی ساعت) نشان میدهد که بررسی باید از کجا شروع شود و اشاره‌گر تا زمانیکه بیت R ـ ۰ پیدا شود جلو می‌رود و با پیشروی بیت‌های R ـ یک را ۰ می‌کنیم، وقتی صفحه victim پیدا شد، صفحه جدید در رم جایگزین می‌شود و صفحه جدید در صف حلقوی در آن موقعیت قرار می‌گیرد

نکته) الگوریتم شانس دوباره پیشرفته به اینصورت است که علاوه بر بیت R بیت M (Dirty) را هم درنظر می‌گیرد.



توجه) اگر توی گزینه‌ها هم دومین شانس بود و هم clock جفتش رده. انشاالله!

اینجا ایران است.

نکته) clock پیشرفته همان دومین شانس پیشرفته است.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

دومین شانس پیشرفته (کلاک پیشرفته): وقتی جایگزینی صفحه فراخوانی می‌شود از همان روش الگوریتم ساعت استفاده می‌کند اما به جای بررسی بیت R هر پیج، به کلاس هر پیج نگاه می‌کند، در صف حلقوی می‌ریم جلو که قدیمی‌ترین صفحه در کلاس ۰ را پیدا کنیم حالا اگر یکدور رفتیم و کسی در کلاس ۰ پیدا نکردیم این بار دنبال پیج کلاس ۱ می‌گردیم و...

نکته) ممکن الت مجبور باشیم چندین بار صف حلقوی را اسکن کنیم تا صفحه‌ای را پیدا کنیم که باید جایگزین شود

1. Beginning at the current position of the pointer, scan the frame buffer. During this scan, make no changes to the use bit. The first frame encountered with $(u = 0; m = 0)$ is selected for replacement.
2. If step 1 fails, scan again, looking for the frame with $(u = 0; m = 1)$. The first such frame encountered is selected for replacement. During this scan, set the use bit to 0 on each frame that is bypassed.
3. If step 2 fails, the pointer should have returned to its original position and all of the frames in the set will have a use bit of 0. Repeat step 1 and, if necessary, step 2. This time, a frame will be found for the replacement.

counting-Based page Replacement

۹- LFu (least-frequently used): گذشته دور را فراموش می‌کند (NFU)

PT	R	counter
	۰	۲۳
	۱	۱۱ ۱۰
	۱	۱۵ ۱۶
	۰	۴۰
:		

برای هر پیج در PT یک کانتر در نظر می‌گیریم و OS در هر دوره زمانی مشخص میاد و بیت R را با کانتر جمع می‌کند و پس بیت R هم پیج‌های درون رم را ۰ می‌کند، حالا اعداد درون کانتر نشان می‌دهد که به هر پیج در چند دوره زمانی مورد مراجعه قرار گرفته. حالا وقتی PF رخ میده صفحه‌ای را میندازیم بیرون که کانترش از همه کمتره. توجه کنید اگر صفحه‌ای از رم خارج شد و دوباره وارد شد کانترش باید ۰ شود

- فلسفه این الگوریتم این الت که صفحه‌ای که بطور فعال استفاده میشه باید reference count بالایی داشته باشه، اما ممکن الت یک فرآیند در مراحل ابتدایی اش به شدت مورد استفاده قرار بگیره اما پس از آن هرگز مورد استفاده قرار نگیره، حالا این صفحات در رم باقی می‌مانند حتی اگر دیگه نیازی به آنها نباشه

solution: بیایم و در intervalهای زمانی کانترها را هر بار یکی به سمت راست شیفت بدهیم به این صورت داریم یک میانگین نمایی کاهشی از کانت مورد استفاده تشکیل میدهیم: exponentially decaying average usage count

توجه) اگر چند پیج کاندید جایگزینی بودند، tie breaker نیاز داریم که در بیشتر مواقع FIFO استفاده میشه

۱۰- MFu (Most frequently used)

عکس بالایی الت و ایده اش اینه که کسی که کانترش کوچکتره احتمالا به تازگی وارد شده و هنوز استفاده زیادی نشده

۲ تا الگوریتم بالا دوتایی داره و رایج نیستند

۱- پیاده سازی شون گران الت

۲- OPT رو خوب تقریب نمی‌زنند

۱۱- page-Buffering Algorithm مدل‌های مختلفی دارد.

یک pool از فریم‌های آزاد را بصورت رزرو نگه می‌داریم، حالا هروقت PF رخ بده، روال پیدا کردن victim مثل قبل انجام میشه ولی قبل از اینکه بخوایم victim رو توی کارد بنویسیم و بعد صفحه مورد نظرمون رو بیاریم توی رم که اینجوری این صفحه معطل میشه می‌یایم و صفحه مورد نظرمان رو میاریم توی یکی از فریم‌های خالی pool مون، این روش به فرآیندی که رجعا PF شده اجازه میده

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

دیگه منتظر نوشتن صفحه victim توی هارد نشه و سریعتر از بلاک ready بشه. حالا هروقت صفحه victim توی هارد نوشته شد، فریم‌اش را به استخر فریم‌های آزادی که برای این کار در نظر گرفتیم اضافه می‌کنیم.

ورژن دیگر: در روش بالا بیایم و فریم‌هایی که می‌خوایم از رم بندازیم بیرون را بذاریم توی pool بیرون و بیت P اش رو ۰ کنیم، حالا وقتی برای یک صفحه PF رخ میده، OS اول بیاد و ببینه که آیا این صفحه توی این pool هست یا نه و اگر بود از همین جا بخونه و اگر نبود دیگه بره و اون صفحه رو میاره توی یکی از فریم‌های آزاد

expansion: هر زمان که paging Device بیکار بود بیاد و یک صفحه‌ای توی رم که بیت M (Dirty/modify) اش ۱ است را در هارد می‌نویسه و بیت M این صفحه را ۰ می‌کنه، اینجوری احتمال تمیز بودن صفحات را هنگام انتخاب برای جایگزینی و عدم نیاز به نوشتن صفحات از رم به هارد را بالا می‌بره.

ورژن دیگر به همراه دوستان شان:

۳ تا لیست تشکیل می‌دهیم و صفحات درون لیست ۲و۳ کاندیدا اخراج هستند که وقتی PF رخ می‌دهد، از سر لیست ۲ آن صفحه را می‌اندازیم بیرون (قدیمی‌ترین پیج تمیز که کاندیدا اخراج است) و پیج جدید را به انتهای لیست ۱ اضافه می‌کنیم، حالا لیست ۱و۲ چگونه تشکیل می‌شود؟ وقتی صفحه جدید را به انتهای لیست ۱ اضافه می‌کنیم، صفحه سر لیست ۱ را با توجه به بیت M اش به انتهای لیست ۲ یا ۳ اضافه می‌کنیم

۱- لیست صفحات توی رم

۲- لیست " توی pool (بافر) که بیت M شون ۰ است.

۳- " " " " " " ۱ " .

۱ لیست) ~~۱۰~~ → ۲۰ → ۳۰ → ۴۰ → ۵۰ → ۱۰۰

۲ ") ~~۵~~ → ۱۵ → ۲۵ → ۳۵

۳ ") ۳ → ۶ → ۹ → ۱۰

فرض کنید می‌خوایم به صفحه ۱۰۰ دستی پیدا کنیم و نیت و براش PF رخ میده و بیت M صفحه ۱۰ هم ۱ است

نکته) صفحاتی که می‌بریم شون توی لیست ۲و۳ بیت P شون رو ۰ می‌کنیم، حالا هروقت این آدرس‌ها تولید شد براشون PF رخ میده و OS اول میاد توی اینها رو نگاه میکند و اگر توی این لیست بود، از همین جا میخونه و این پیج را هم به انتهای لیست ۱ اضافه می‌کنیم و P اش را نیز ۱ می‌کنیم

چرا بیت P افراد توی لیست ۲و۳ را ۰ کردیم؟ چون اگر نمی‌کردیم مجبور بودیم که به ازای هر مراجعه بیایم و لیست ۲و۳ را هم بگردیم و ببینیم اگر توی این ۲ تا لیست هست بیاریم اش توی لیست ۱

حالا چرا شبیه دوستان شانسه؟ اگر به پیجی درون لیست ۲و۳ دستی پیدا کردیم، می‌ریم و میذاریم اش توی لیست ۱ و بعش شانس مجور می‌دیم که توی رم بمونه

نکته) با ایجاد لیست ۲و۳ می‌یایم و صفحات لیست ۲ را برای اخراج در اولویت قرار می‌دهیم و سربار نوشتن از رم به هارد رو کم می‌کنیم

Frame Allocation

به یک فرآیند در ابتدای کارش و در ادامه اجرایش چند فریم اختصاص بدهیم؟

۱- pure Demand paging: در این روش فرآیندها با تعداد ۰ فریم در رم شروع به کار می‌کنند و به تدریج working set خودشان را می‌سازند، در این روش پیج‌های یک فرآیند فقط وقتی از هارد به رم برده می‌شوند که فرآیند بهشون نیاز داره و از قبل

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

نمی یاریم شون توی رم.
وقتی فضای آدرس یک فرآیند درگیر ساخته می‌شود هیچ پیجی از فرآیند در رم گذاشته نمی‌شود، حالا وقتی cpu به فرآیند داده می‌شود و فرآیند می‌خواد از آدرس اولین دستورش fetch کند میریم توی PT نگاه می‌کنیم و می‌بینیم که بیت p اش ۰ است و PF رخ میدهد و به حالت wait می‌رود تا داده‌اش از توی هارد به رم آورده شود و به همین ترتیب.
خب این روش زمان شروع به کار فرآیندها را کند می‌کند چون وقتی یک فرآیند در ابتدای کارش هست داده‌هایش در رم نیست و PF های مکرر براش رخ میده و هم اش بلاک میشه تا کم کم بیفته روی غلتک.
اگر قرار باشه هروقت که هر فرآیندی رو می‌خوایم بیاریم توی رم کلی PF رخ بدهد سیستم کند میشه و همین طور بهره وری cpu هم میاد پایین چون با هر PF، کد os باید روی cpu اجرا بشه.
به همین دلیل الگوریتم های براساس آمار یک حداقل تعدادی رو در شروع بکار در نظر می‌گیرند

نکته*) حداقل تعداد فریمی که هر فرآیند برای اجرا نیاز دارد به شدت به معماری سیستم بستگی دارد، مثلا کامپیوتری که در آن تمامی دستوراتی که مراجعه به حافظه دارند فقط شامل یک آدرس حافظه باشد حداقل به ۲ صفحه نیاز دارد، کامپیوتری که تمامی این نوع دستورات شامل یک آدرس indirect یک مطلبی به حداقل ۳ صفحه نیاز دارد و اگر در کامپیوتری این نوع دستورات بتوانند ۲ فیلد آدرس غیرمستقیم داشته باشد حداقل ۴ صفحه برای اجرا نیاز دارد!

Frame Allocation Algorithm

m فریم داریم و n فرآیند، به هر فرآیند چند فریم تخصیص بدهیم؟

Equal Allocation: $\frac{m}{n}$ frames per process

خب هر فرآیند جدیدی بیاد می‌تعداد فریم‌هایی که اختصاص داریم به فرآیندها کمتر میشه و همین طور فرآیندها بصورت هم اندازه به فریم‌ها نیاز ندارند

" proportional:
بیاییم تعداد فریم‌ها را به اندازه Vm هر فرآیند تخصیص بدهیم

$s_i = \text{Vm size of process } p_i$
$S = \sum s_i$
$m = \text{total number of frames}$
$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

" priority: با توجه به اولویت فریم بدهیم به فرآیندها

نکته) می‌تونیم ترکیبی از اندازه و اولویت در نظر بگیریم
خب این روش‌ها خیلی مناسب نیستند چون نیازهای فرآیندها به تعداد لزوما نه به اندازه شون بستگی دارد و نه به اولویت شون، علاوه بر این یک فرآیند رفتارش در طول زمان تغییر می‌کند.

Global vs local Algorithm

وقتی PF رخ می‌دهد یک صفحه از رم را باید خارج کنیم. حالا قدیمی ترین صفحه کدام فرآیند را اجرا کنیم؟

local replacement: وقتی می‌خواهیم فریمی را از رم بندازیم بیرون فقط می‌تونیم از فریم‌های فرآیند جاری بندازیم بیرون
زمان جستجو برای یافتن کاندیدا اذیت کننده میشه
تخصیص محلی می‌تواند همراه با مکانیزم‌های ایستا و پویا همراه باشد

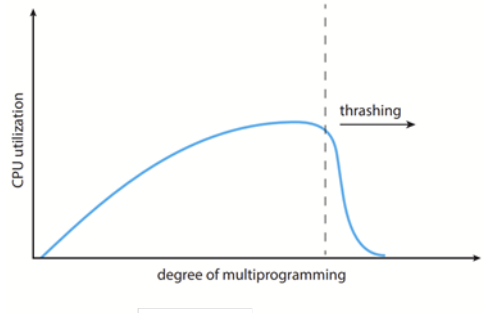
*ایستا: یک تعداد مشخصی فریم به فرآیندها داده می‌شود. که این خوب نیست چون با توجه به رفتار فرآیند گاهی اوقات آن تعداد ثابت کم است و گاهی اوقات زیاد است، اون زمان‌هایی که زیاده حافظه هدر میره و اون زمان‌هایی که کمه برای فرآیند گرفتگی رخ میدهد.
پویا: با توجه به نیاز فرآیند تعداد فریم‌هایی که بهش تخصیص می‌دهیم کم و زیاد بشود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

global replacement : ازبین تمام صفحات درون رم یکی رو میندازیم بیرون، این روش ذاتا پویا است چون یک فرآیند که برایش PF رخ می دهد، os ممکن است فریم های تخصیص یافته به فرآیندهای دیگران را برای این فرآیند انتخاب کند و بنا براین تعداد فریم های اختصاص داده شده به این فرآیند زیاد می شود، همچنین ممکن است به فرآیندهای دیگر موقع اجرا از صفحات این فرآیند داده شود پس کوچک تر می شود

پس یکی از مشکلات این روش این است که مجموعه صفحات یک فرآیند در رم نه تنها به رفتار صفحه بندی آن فرآیند بلکه به رفتار صفحه بندی سایر فرآیندها نیز بستگی دارد، بنا براین یک فرآیند ممکن است در اجراهای مختلف بصورت کاملا متفاوت عمل کند، مثلا یکبار توی ۱ ثانیه اجرا بشه و یکبار توی ۵ ثانیه، در صورتیکه این مورد را در روش جایگزینی محلی نداریم.

زمان جستجو افزایش پیدا می کند ولی از حافظه بصورت بهینه استفاده می شود، بطورکلی روش جایگزینی global ای منجر به توان عملیاتی بیشتر می شود و رایج تر است.



Thrashing.

تعداد فریم هایی که به هر فرآیند اختصاص می دهیم بسیار مهم است

تا یکجایی افزایش تعداد فرآیندها باعث افزایش بهره وری cpu میشه، تعداد فرآیندهای توی رم که زیاد میشه به خاطر یکسری موارد بهره وری cpu کم میشه، مثلاً

۱- مثلاً cs زیاد میشه

۲- وقتی تعداد فرآیندهای رم زیاد میشه چون به هر فرآیند تعداد فریم کمتری می رسد فرآیندها بیشتر دچار PF می شوند و os و cpu بیشتر زمان خود را درگیر جایگزین کردن صفحات است و این یعنی انجام I/O های بیهوده

وقتی تعداد فرآیند زیادی بیاریم توی رم دیگه os فرآیندها در رم جا نمی شود و کوبیدگی رخ میدهد

- نتیجه PF ها این است که فرآیندهایی که PF براشون رخ داده در paging device منتظر بشوند و بهره وری cpu میاد پایین

پدیده کوبیدگی : زمانی رخ میدهد که به بیشتر فرآیندهای زنده تعداد مناسب فریم اختصاص داده نشده و برای فرآیندها خیلی PF رخ می دهد و os به جای کمک در اجرا مفید فرآیندها پیوسته درحال آوردن و بردن فریم ها به/از رم است.

برای رفع این مشکل باید درجه چندبرنامگی را کاهش بدهیم که فریم های رم آزاد بشه که بدیم شون به سایر فرآیندها

- یک فرآیند thrashing است درصورتی که زمان بیشتری را صرف صفحه بندی کند نسبت به اجرا، که اگر تعداد فریم های در اختیارش کمتر از تعداد فریم های مورد نیازش باشد این اتفاق برایش رخ می دهد.

نکته) با استفاده از جایگزینی صفحه محلی اگر یک فرآیند شروع به thrashing کند نمی تواند فریم هایی را از سایر فرآیندها بگیرد

نکته) thrashing منجر به افزایش زمان انتظار در paging device می شود که این موضوع روی زمان موثر دسترسی به حافظه حتی برای فرآیندهایی که thrashing نیستند تاثیر می گذارد

اگر تخصیص حافظه به یک فرآیند کوچکتر از working set آن فرآیند باشد نرخ PF ای که برای آن فرآیند رخ می دهد زیاد است و برنامه دچار thrashing می شود و دچار کندی بیش از حد می شود، در این حالت وقتی برای فرآیند PF رخ میدهد برای آنکه این صفحه os خودش را از ع رد و وارد رم کند چون فضای کمی در رم دارد باید یکی دیگر از صفحات os اش را که بعدا باز به آن نیاز دارد را از رم بیرون می اندازد

۲) Demand paging: prepaging : تا حد امکان قبل از اینکه به فرآیند cpu بدهیم صفحاتی که آن فرآیند نیاز دارد را برایش مهیا کنیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

برای جلوگیری از thrashing ، یک فرآیند باید فریم‌های مورد نیاز خود را داشته باشد
چگونه بفهمیم هر فرآیند به چند فریم نیاز دارد
۱) working set Model
۲) page-Fault Frequency

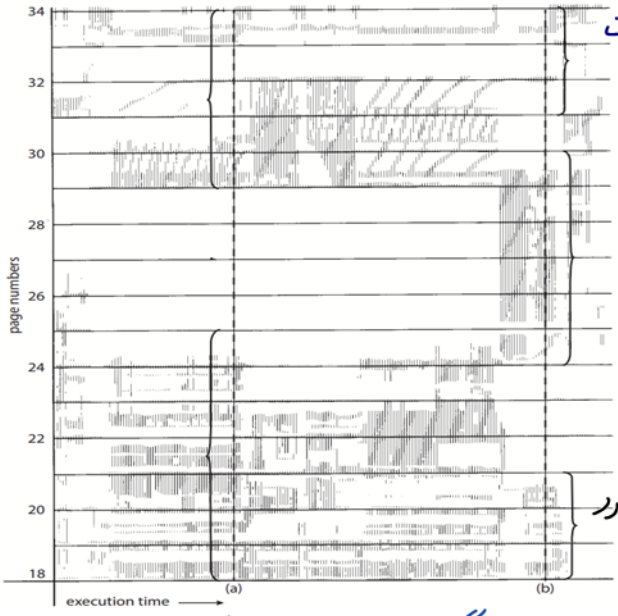
(مجموعه کاری) working set

حداقل میزان حافظه‌ای که باید به یک فرآیند تخصیص یابد تا برنامه بتواند با سرعت قابل قبولی اجرا شود و اگر این حداقل را در اختیار آن فرآیند قرار ندهیم برای برنامه PF مکرر به وجود می‌آید و برنامه دچار thrashing می‌شود

فرآیندها بصورت یکنواخت به فضای آدرس‌شون مراجعه نمی‌کنند و رفتار پویا دارند، حالا ما میایم می‌گیم رفتار گذشته یک فرآیند رفتار آینده اش را هم مشخص می‌کند و رفتار آینده فرآیندها شبیه رفتار گذشته‌شان است

بنابراین این روش رفتار محلی فرآیندها را در نظر می‌گیرد، محلی بودن اجرا یعنی مجموعه‌ای از صفحات برنامه هستند که در بیشتر زمانها با هم برای آن فرآیند استفاده می‌شود و بنابراین در اجرا یک فرآیند همواره صفحات اش از یک مجموعه محلی اش به مجموعه محلی دیگرش منتقل می‌شود (locality model)

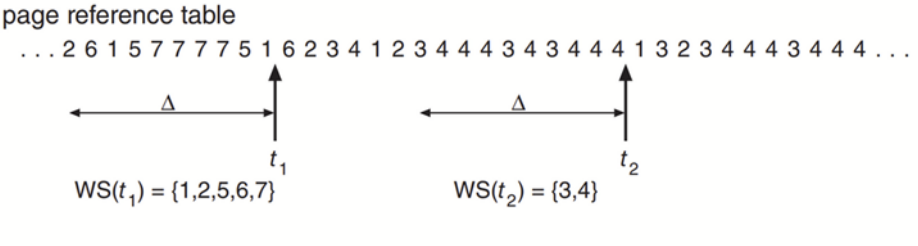
- در شکل روبه رو دارید نیاز یک فرآیند در طول اجرا به فضای آدرس اش را می‌بینید، خب همون طور که مشخصه رسیدن یک فرآیند به حافظه اش الگوی محلی دارد و بخش‌هایی که باهاشون کار می‌کند به بخش‌های مجاورش هم نیاز دارد



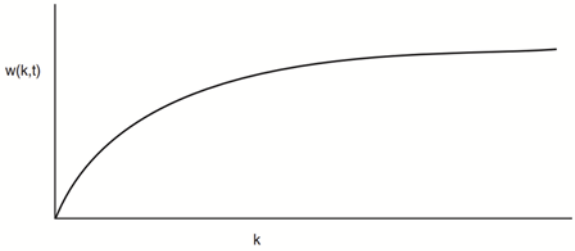
Δ : working set window : Δ = مثلاً ۱۰ هزار ارجاع افید

WS(t) = working set(t) = مجموعه صفحاتی که در لحظه t در پنجره Δ مورد استفاده قرار گرفتند

w(Δ,t)



Working-set model.



The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

اندازه ws با اندازه K رابطه خطی ندارد و در ابتدای که K را زیاد می‌کنیم ws بصورت خطی زیاد می‌شود ولی از یک جایی به بعد افزایش K باعث افزایش اندک ws می‌شود، چون وقتی K را از یک حدی بیشتر می‌کنیم دیگه ws فرآیند مشخص شده و بدست آمده و فرآیند دیگه دارد با همان صفحات درون ws اش کار می‌کند.

نکته) اگر Δ را کوچک بگیریم خب ws نمی‌تواند در برگیرنده الگوی محلی فعلی فرآیند در حال اجرا باشه و اگر Δ بزرگ باشه دقیق نیست و ws بیهوده بزرگ می‌شود.

WSS_i : اندازه مجموعه کاری یک فرآیند (working set size of process p_i)

$D = \sum WSS_i$: مقدار فریمی فعلی که تمامی فرآیندها به آن نیاز دارند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

m : تعداد فریم‌های رم

- اگر D>m باشد یعنی تعداد فریم‌های مورد نیاز فرآیندها بیشتر از تعداد فریم‌های رم است و بنابراین پدیده thrashing رخ می‌دهد و os باید یک فرآیند را برای suspend شدن انتخاب کند و صفحاتش را swap out کند و فریم‌هایش را به فرآیندهای دیگر بدهد
- " D<m : حالا اگر m به اندازه کافی بزرگ باشد os تعداد فرآیندهای سیستم را می‌تواند افزایش بدهد

نکته) این استراتژی از thrashing جلوگیری می‌کند و درعین حال درجه چندبرنامگی را تا حد امکان بالا نگه می‌دارد و بنابراین استفاده از cpu را بهینه می‌کند.

نکته) os بصورت پیوسته درحال بدست آوردن ws فرآیندها است و

- برای پیاده سازی این روش باید برای هر فرآیند ثبت کنیم که در ۵ مراجعه اخیر به چه صفحاتی مراجعه کرده و این پرهزینه است و قابل پیاده سازی نیست

R	Age
	01010111
	00000101
	11010011

m

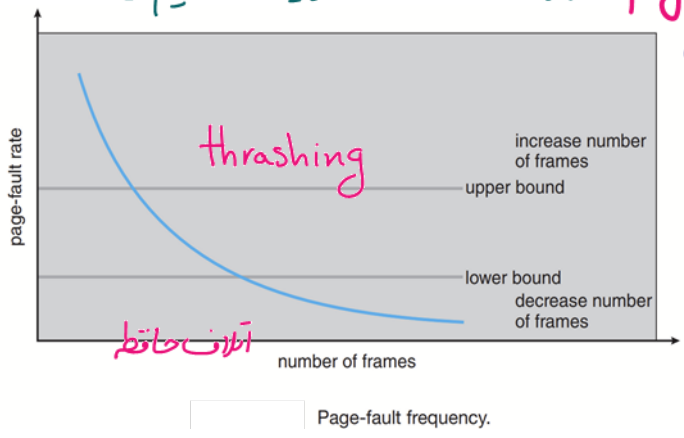
از فیلد Age در pT استفاده می‌شود، تایمر هر یک مدت یکبار وقفه می‌دهد، حالا می‌رم و صفحاتی که مثلاً در m پریود اخیر استفاده شده، یعنی حداقل یک 1 دارند را جزو مجموعه کاری هر فرآیند می‌گیریم

دانستنی: ویندوز از مدل مجموعه کاری استفاده می‌کند و به هر فرآیند یک حداقل و یک حداکثر مجموعه کاری نسبت می‌دهد، حداقل مجموعه کاری مقداری است که برای هر فرآیند تضمین می‌کند در طول اجرایش بدهد

ویندوز از مدل clustring هم استفاده می‌کند که به اینصورت است که وقتی یک فرآیند دچار PF می‌شود صفحات اطراف آن را نیز به رم می‌آورد

- اگر کل تعداد فریم‌های آزاد از یک مقدار مشخص کمتر بشه ویندوز روال خودکار کمتر کردن ws را برای فرآیندها انجام می‌دهد در این روال فریم‌هایی که بیشتر از مقدار حداقل به فرآیندها داده شده از آنها گرفته می‌شود (automatic working set triming)

page-Fault Frequency اندازه ws داده شده به فرآیندها را تنظیم می‌کند



Page-fault frequency.

این نمودار نشان می‌دهد که اگر الگوریتمی را انتخاب کنیم که ناهنجاری بلیدی نداشته باشد هرچه تعداد صفحات داده شده به یک فرآیند بیشتر شود تعداد PF کم می‌شود

حالا روش PFF تعداد PF‌هایی که برای فرآیندها رخ می‌دهد را در دوره‌های زمانی می‌شمارد و:

۱- وقتی PF یک فرآیند از lower bound مشخص شده کمتر است تعداد فریم‌های تخصیص یافته از آن را کم می‌کند

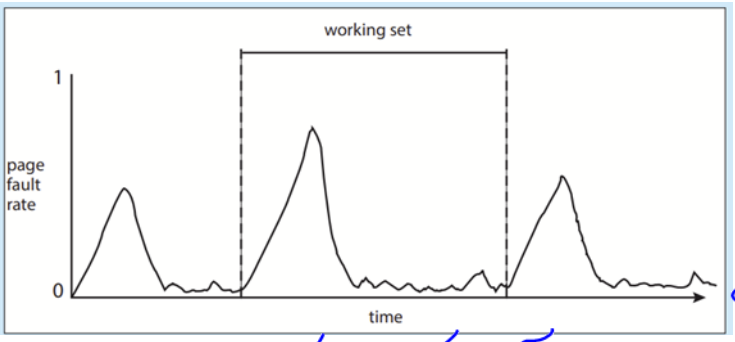
۲- وقتی PF یک فرآیند از upper bound بیشتر شد داره به سمت thrashing میره و تعداد فریم‌های بیشتری بهش تخصیص می‌دهیم

نکته) هرگاه os متوجه بشه که نمی‌تواند تعداد صفحه هم فرآیندها را در ناحیه مطلوب نگه دارد آن گاه یک یا چند فرآیند را سیبره روی هارد و فریم‌هایش را بین بقیه تقسیم می‌کند

پراکندگی پویا فرآیند یا برنامه: مجموعه کاری فرآیند کوچک است و با سرعت زیاد تغییر می‌کند ← pure demand paging به این شخص بیشتر اهمیت میده

پراکندگی ایستا " " : " " " بزرگ " " " کمی " " ← یا prepaging به این شخص بیشتر اهمیت میده و براش مفیدتره

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.



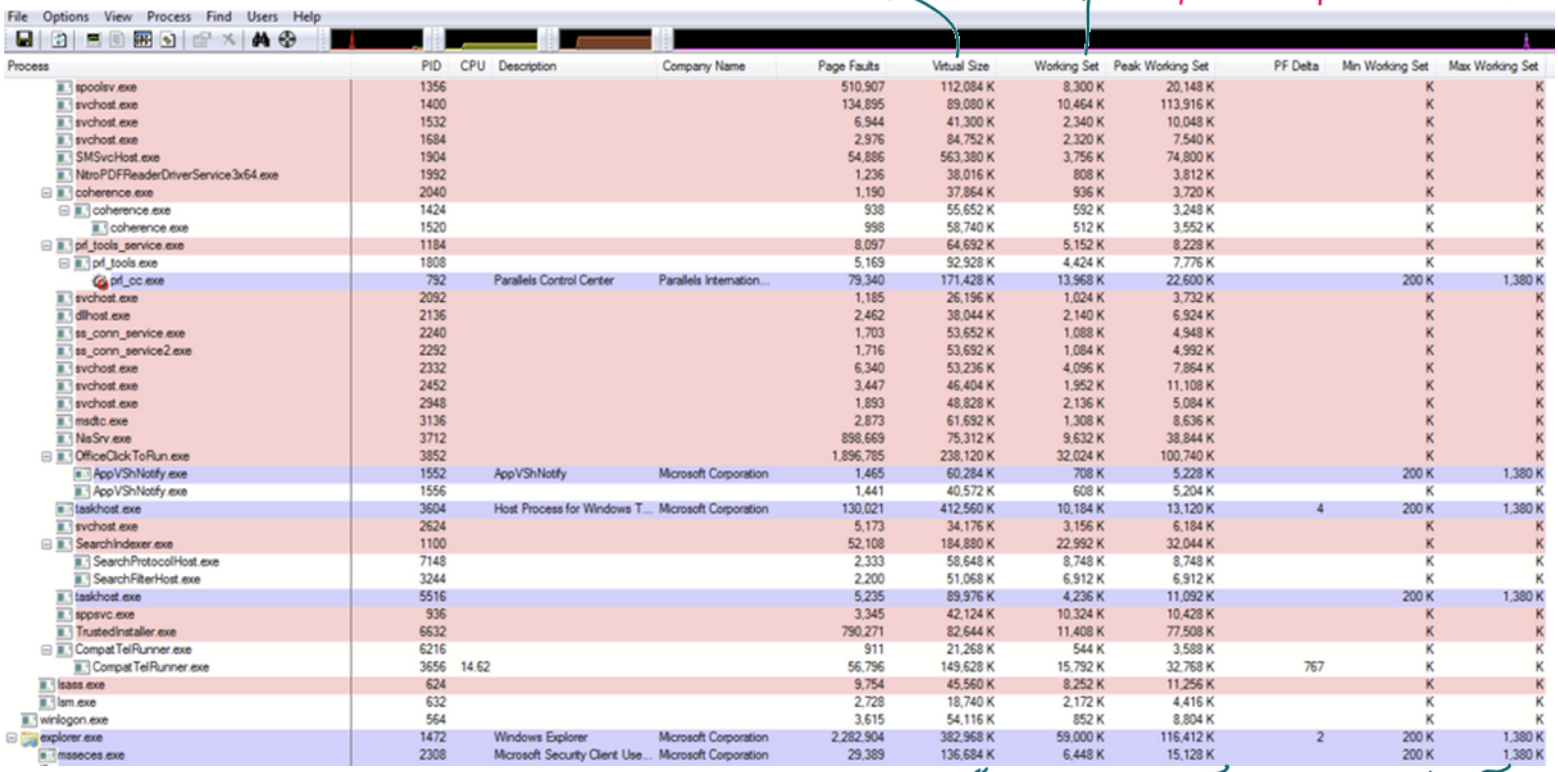
رابطه مستقیمی بین مجموعه کاری یک فرآیند و نرخ PF آن وجود دارد. می دانیم که مجموعه کاری یک فرآیند در طول زمان تغییر می کند. با فرض آنکه حافظه کافی برای ذخیره مجموعه کاری یک فرآیند وجود داشته باشد یعنی thrashing نباشد نرخ PF فرآیند در طول زمان بین قله ها و دره ها تغییر می کند.

اوج نرخ PF زمانی اتفاق می افتد که فرآیند می خواهد به صفحه های جدیدی دست بزند ولی وقتی این مجموعه کاری وارد رم می شود نرخ PF کاهش می یابد، باز وقتی فرآیند به یک مجموعه کاری جدید می رود نرخ PF زیاد می شود و به همین ترتیب. بازه زمانی بین شروع یک پیک و شروع پیک بعدی نشان دهنده انتقال از یک مجموعه کاری به یک مجموعه کاری دیگر است.

نرم افزار process explorer

فرآیند اول ۱۱۲MB رم می خواسته ولی کلا با ۸MB اجرا شده

اندازه حافظه Virtual ای که هر فرآیند نیاز دارد



Process	PID	CPU	Description	Company Name	Page Faults	Virtual Size	Working Set	Peak Working Set	PF Delta	Min Working Set	Max Working Set
spoolsv.exe	1356				510,907	112,084 K	8,300 K	20,148 K		K	K
svchost.exe	1400				134,895	89,080 K	10,464 K	113,916 K		K	K
svchost.exe	1532				6,944	41,300 K	2,340 K	10,048 K		K	K
svchost.exe	1684				2,976	84,752 K	2,320 K	7,540 K		K	K
SMSvcHost.exe	1904				54,886	563,380 K	3,756 K	74,800 K		K	K
NitroPDFReaderDriverService3x64.exe	1992				1,236	38,016 K	808 K	3,812 K		K	K
coherence.exe	2040				1,190	37,864 K	936 K	3,720 K		K	K
coherence.exe	1424				938	55,652 K	592 K	3,248 K		K	K
coherence.exe	1520				998	58,740 K	512 K	3,552 K		K	K
prl_tools_service.exe	1184				8,097	64,692 K	5,152 K	8,228 K		K	K
prl_tools.exe	1808				5,169	92,928 K	4,424 K	7,776 K		K	K
prl_cc.exe	792		Parallels Control Center	Parallels Internation...	79,340	171,428 K	13,968 K	22,600 K		200 K	1,380 K
svchost.exe	2092				1,185	26,196 K	1,024 K	3,732 K		K	K
dllhost.exe	2136				2,462	38,044 K	2,140 K	6,924 K		K	K
ss_conn_service.exe	2240				1,703	53,652 K	1,088 K	4,948 K		K	K
ss_conn_service2.exe	2292				1,716	53,692 K	1,084 K	4,992 K		K	K
svchost.exe	2332				6,340	53,236 K	4,096 K	7,864 K		K	K
svchost.exe	2452				3,447	46,404 K	1,952 K	11,108 K		K	K
svchost.exe	2948				1,893	48,828 K	2,136 K	5,084 K		K	K
msdtc.exe	3136				2,873	61,692 K	1,308 K	8,636 K		K	K
NisSrv.exe	3712				898,669	75,312 K	9,632 K	38,844 K		K	K
OfficeClickToRun.exe	3852				1,896,785	238,120 K	32,024 K	100,740 K		K	K
AppVShNotify.exe	1552		AppVShNotify	Microsoft Corporation	1,465	60,284 K	708 K	5,228 K		200 K	1,380 K
AppVShNotify.exe	1556				1,441	40,572 K	608 K	5,204 K		K	K
taskhost.exe	3604		Host Process for Windows T...	Microsoft Corporation	130,021	412,560 K	10,184 K	13,120 K	4	200 K	1,380 K
svchost.exe	2624				5,173	34,176 K	3,156 K	6,184 K		K	K
SearchIndexer.exe	1100				52,108	184,880 K	22,992 K	32,044 K		K	K
SearchProtocolHost.exe	7148				2,333	58,648 K	8,748 K	8,748 K		K	K
SearchFilterHost.exe	3244				2,200	51,068 K	6,912 K	6,912 K		K	K
taskhost.exe	5516				5,235	89,976 K	4,236 K	11,092 K		200 K	1,380 K
sppsvc.exe	936				3,345	42,124 K	10,324 K	10,428 K		K	K
TrustedInstaller.exe	6632				790,271	82,644 K	11,408 K	77,508 K		K	K
CompatTelRunner.exe	6216				911	21,268 K	544 K	3,588 K		K	K
CompatTelRunner.exe	3656	14.62			56,796	149,628 K	15,792 K	32,768 K	767	K	K
lsass.exe	624				9,754	45,560 K	8,252 K	11,256 K		K	K
lsm.exe	632				2,728	18,740 K	2,172 K	4,416 K		K	K
winlogon.exe	564				3,615	54,116 K	852 K	8,804 K		K	K
explorer.exe	1472		Windows Explorer	Microsoft Corporation	2,282,904	382,968 K	59,000 K	116,412 K	2	200 K	1,380 K
msseces.exe	2308		Microsoft Security Client Use...	Microsoft Corporation	29,389	136,684 K	6,448 K	15,128 K		200 K	1,380 K

ویندوز آمار working set هر فرآیند را براساس بایت نگه می دارد.

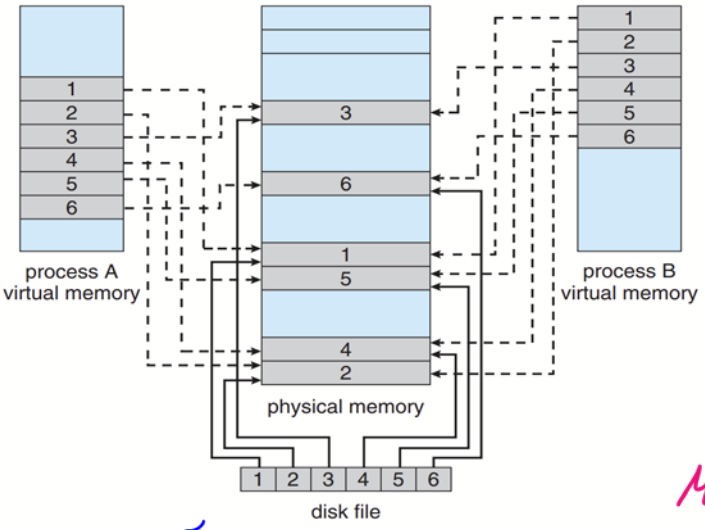
Memory mapped Files

OS به فایل ها همان نگاه paging ای را داشته باشد که به فضای آدرس فرآیند دارد و اگر ما فایلی را خواستیم OS می رود و فایل ها را با استفاده از سیستم Demand paging اش بصورت پیج پیج میاره توی رم و انگار ما از سیستمی که برای Demand paging ایجاد شده برای آوردن و بردن فایل ها استفاده می کنیم و در نتیجه سرعت افزایش پیدا می کند.

وقتی می خوایم از فایل هایی که توی هارد است استفاده کنیم و می خوایم توشون بنویسیم یا ازشون بخونیم می آییم و با دستورات fwrite و fread با فایل های روی هارد مستقیم کار کنیم، یعنی با هارد بصورت یک دستگاه I/O رفتار می کنیم و با دستورات مرتبط با خواندن و نوشتن از/در هارد باهاش کار می کنیم، حالا یک راه دیگه اینه که به جای اینکه با این دستورات بصورت مستقیم با فایل های روی هارد کار کنیم مثل کار کردن با یک بخش حافظه باهاش کار کنیم

- در این روش کار با فایل ها مثل کار با یک بخش از رم است، مثل همونجوری که با متغیرها کار می کنیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.



- در این حالت اشتراک یک فایل بین فرآیندها راحت است
- تغییرات فایل‌ها در زمان‌های مختلفی می‌تواند روی‌ها رد نوشته شود، مثلا در زمان بستن فایل‌ها یا در بازه‌های زمانی منظم

Memory mapped I/O

در برخی از سیستم‌ها ثبات‌های کنترلی دیوایس‌های I/O به آدرس‌های حافظه map می‌شود و نوشتن و خواندن در/از این آدرس‌ها باعث دستیابی به آن ثبات‌ها می‌شود.

یک فایل در کرنل به آدرس‌های یک دیوایس I/O، map می‌شود و کار با آن دیوایس به همان صورت کار با فایل‌ها تبدیل می‌شود

Allocating Kernel Memory

وقتی یک فرآیند که در مد کاربر اجرا می‌شود حافظه اضافی درخواست می‌کند، os از لیست فریم‌های آزادی که در اختیارش هست بخش اختصاص میده، حالا اگر یک فرآیند درخواست یک بایت هم بدهد باز os یک صفحه بخش میده که درنتیجه قطعه قطعه شدن داخلی به وجود میاد.

اما معمولا حافظه‌ای که به کرنل اختصاص داده می‌شود از یک pool آزاد متفاوت از لیستی که برای فرآیندهای حالت کاربر معمولی استفاده می‌شود اختصاص می‌یابد.

۲ دلیل برای این کار وجود دارد

۱) کرنل از ساختارهای داده با اندازه‌های مختلف که اندازه برخی از آنها کمتر از یک صفحه است استفاده می‌کند بنابراین باید بطور محافظه کارانه از حافظه استفاده کند و سعی کند اتلاف ناشی از تکه تکه شدن را حداقل کند

۲) صفحاتی که به فرآیندهای مد کاربر اختصاص می‌دهیم نیازی نیست توی رم پیوسته باشند ولی برخی از دیوایس‌های سخت افزاری بصورت مستقیم با رم ارتباط برقرار می‌کنند، بدون اینکه بخواهند از interface ـ حافظه مجازی استفاده کنند و درنتیجه ممکن است نیاز به حافظه بصورت پیوسته در رم داشته باشند

بنابراین باید استراتژی‌هایی برای مدیریت حافظه‌های آزادی که به فرآیندهای کرنل اختصاص می‌دهیم داشته باشیم

* حافظه‌هایی که کرنل آزاد می‌کند باعث باعث fragmentation ـ زیادی در حافظه خودش می‌شود

حالا ۲ روش مدیریت حافظه‌های آزاد کرنل را توضیح می‌دهیم

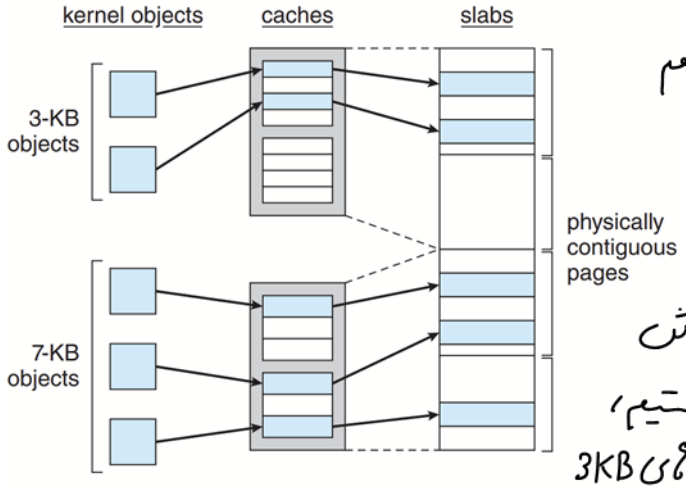
۱- Buddy system ← شکل تکه تکه شدن داخلی دارد

coalescing : یکی از مزیت‌های Buddy system ادغام سریع دوستان مجاور است

۲- slab Allocation ← در این روش هیچ فضایی به دلیل تکه تکه شدن از بین نمی‌رود

برای هر ساختمان داده کرنل یک ساختمان داده‌ای به اسم cache می‌سازد که هدرش

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.



Slab allocation.

دارای چند slab است و هر slab هم از یک یا چند پیج پیوسته پشت سر هم تشکیل می‌شود، slab های توی هر کش هم سایزند.

حالا میایم در این روش یک کش می‌سازیم برای مثلا object های 3KB ای و یک کش می‌سازیم برای object های 7KB ای و...

حالا وقتی یک object ـ 3KB ای داشت کرنل میاد سراغ کش 3KB اش و... حالا اینجا اگر حافظه هر slab مثلاً 3KB ای آزاد بشود نگران نیستیم، چون بعدا هم می‌خوایم توی slab های مربوط به کش 3KB ای object های 3KB ای ذخیره کنیم، بنابراین فضاها هدر نمی‌روند

* وقتی slab های یک کش xKB ای پر شد به آن کش slab جدید اختصاص داده می‌شود

- برای هر نوع از Data structure ـ کرنل یک کش وجود دارد، مثلاً یک کش برای سمافورها، یک کش برای توصیف فرآیندها (PCB)
- در یک slab می‌تواند چندین شی نیز قرار گیرد
- وقتی کرنل یک حافظه درخواست کند اول از slab هایی که بصورت partial پر شدند بهش اختصاص میدیم و بعد نشد slab خالی و اگر نبود slab جدید برای کش مربوطه اش می‌سازیم.

مزایا:

۱) هیچ حافظه ای به علت fragmentation هدر نمی‌رود

۲) درخواست حافظه توسط کرنل سریع برآورده می‌شود

page size

۱- در روش segmentation که از صفحه بندی هم استفاده می‌کنیم داریم:

متوسط میزان حافظه تلف شده در internal fragmentation : $n \times \frac{p}{2}$ Byte

n : تعداد سگمنت ها در رم

p : اندازه صفحه به بایت

پس در اینجا افزایش اندازه صفحات باعث اتلاف بیشتر می‌شود

۲- بطور کلی وقتی اندازه یک صفحه را بزرگ بگیریم اگر یک فرآیند به حجم کمی از حافظه نیاز داشته باشد، باز حافظه هدر می‌رود

مثال (مانو): فرض کنید یک برنامه ۸ تا فاز اجرا داره و در هر فاز ۴KB فضا می‌خواد و اندازه صفحات ۳۲KB ای باشد

۳- اگر اندازه صفحات را کوچک بگیریم تعداد صفحات هر فرآیند خیلی زیاد میشد و در نتیجه PT همه فرآیندها بزرگ میشد و حافظه هدر می‌رود. علاوه بر این فکر کنید PT را بخواهیم بذاریم توی رجیسترهای cpu اینجوری زمان cs هم زیاد میشد

۴- اندازه صفحه بهینه با توجه به اندازه PT و internal fragmentation :

s : میانگین اندازه فرآیندها

$\frac{s}{p}$ = میانگین تعداد صفحه هر فرآیند

e : اندازه هر سطر PT

$\frac{s}{p} \times e$ = اندازه هر سطر PT × میانگین تعداد صفحه هر فرآیند = میانگین اندازه هر PT

p : اندازه صفحه

سربار صفحه بندی = سربار PT + سربار تکه تکه شدن داخلی = $\frac{s \times e}{p} + \frac{p}{2}$ → از این برحسب p مشتق می‌گیریم و مساوی ۰ قرار می‌دهیم

$$\frac{-se}{p^2} + \frac{1}{2} = 0 \Rightarrow \frac{se}{p^2} = \frac{1}{2} \Rightarrow p^2 = 2se \Rightarrow p = \sqrt{2se}$$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

process synchronization

زمانی که preemption در زمانبندی مجاز باشد موضوع همگامی فرآیندها اهمیت پیدا می‌کند

producer-consumer (Bounded Buffer)

اگر یک بافری وجود داشته باشد با ظرفیت محدود، اگر پیام‌هایی که فرستنده در بافر قرار بدهد به حدی برسد که بافر پر بشود و پیام بعدی را نتواند در بافر قرار بدهد فرستنده بلوکه می‌شود و زمانیکه بافر خالی بشود و همه پیام‌های آن مصرف بشود و گیرنده دیگر نتواند پیام بعدی را از بافر بردارد می‌گوییم که گیرنده بلوکه شده خب پس باید بدانیم که در هر لحظه چند داده در بافر هستند

producer: (فرستنده)

حلقه بینهایت

```
while (true) {
    /* produce an item in next_produced */

    while (count == BUFFER_SIZE)
        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

اگر بافر پر است

یک شمارنده (متغیر، حافظه) مشترک بین تولیدکننده و مصرف‌کننده که تعداد خانه‌های پر بافر را نشان می‌دهد

اگر کانتر با سایز بافر برابر بشود تولیدکننده در اینجا گیر می‌کند و آیتم تولید شده تو دست تولیدکننده می‌ماند و نمی‌تواند آن را در بافر بگذارد و فرستنده در اینجا بلوکه می‌شود

ولی اگر کانتر با بافر سایز سازی نشود یعنی بافر هنوز ظرفیت دارد، فرستنده آیتم بعدی را در بافر قرار می‌دهد و اشاره‌گر in را آپدیت می‌کند و کانتر نیز زیاد می‌شود

* اشاره‌گر in و out به یک بافر صف حلقوی اشاره می‌کند که اگر:

if in = out ⇒ بافر خالی است

if out = in+1 ⇒ پر است

البته در این شرایط یک خونه بافر هدر می‌رود برای همین در اینجا از پوینتر in و out برای اینکه بدانیم خانه بعدی که باید آیتم در آن قرار بگیرد کجاست یا خونه بعدی که باید آیتم از آن برداشته بشود کجاست هست ولی برای اینکه بفهمیم ظرفیت بافر پر شده یا نه از متغیر مشترک کانتر استفاده می‌کنیم

Consumer: (گیرنده)

```
while (true) {
    while (count == 0)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    /* consume the item in next_consumed */
}
```

- تا زمانیکه کانتر برابر ۰ است، خب آیتمی وجود ندارد که مصرف‌کننده بخواهد بردارد، بنابراین " " " " " باشد مصرف‌کننده در اینجا می‌ماند و بلوکه می‌شود، در غیر اینصورت اگر بافر خالی نباشد، داده بعدی را برمی‌دارد و out را یکی زیاد می‌کند و کانتر را یکی کم می‌کند و می‌رود از داده‌ای که از بافر برداشته استفاده می‌کند

- پس به‌جای اینکه با استفاده از out = in+1 پر بودن یا نبودن بافر را چک کنیم این کار را با استفاده از مقایسه بین counter و سایز بافر (Buffer size) انجام داریم و همان‌طور که مشخص است متغیر counter بین تولیدکننده و مصرف‌کننده (بین ۲ کد) مشترک است، پس یک حافظه مشترک داریم بین این ۲ فرآیند، که همان‌طور که مشخص است کانتر توسط تولیدکننده زیاد میشه و توسط مصرف‌کننده کم می‌شود

Race condition

حالا بریم که اسمبلی قسمتی که تولیدکننده کانتر را زیاد می‌کرد و قسمتی که مصرف‌کننده کانتر را کم می‌کند را بررسی کنیم

کد اسمبلی counter++ توسط تولیدکننده

$$register_1 = count$$
$$register_1 = register_1 + 1$$
$$count = register_1$$

کد اسمبلی counter-- توسط مصرف‌کننده

$$register_2 = count$$
$$register_2 = register_2 - 1$$
$$count = register_2$$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

سوال: اگر مقدار جاری counter=۷ باشد و یک آیتم تولید شود و پس مصرف شود (یا بالعکس) مقدار نهایی کانتر چند خواهد بود؟
الان می‌گید تغییری نمی‌کند، چون می‌گید اول زیاد شده و بعد کم شده
اما اینطوری نیست

فرض کنید در کد تولید کننده count=۵ باشد و در register۱ قرار داده شود و پس رجیستر۱، یکی زیاد بشود و مقدار رجیستر یک ۶ می‌شود و حالا کوانتوم مربوط به فرآیند تولید کننده تمام بشود و قبل از اجرا مجدد تولید کننده کوانتوم به مصرف کننده برسد، مصرف کننده چون از count که مقدارش ۵ است می‌خواند، میاد و ۵ را یکی ازش کم میکند و ۴ اش میکند و در count مقدار ۴ می‌نویسد، حالا اگر کوانتوم مصرف کننده تمام بشود و زمان به تولید کننده برسد، مقدار رجیستر۱ که ۶ بود در متغیر count قرار می‌گیرد و بنابراین در نهایت مقدار count برابر ۶ است.

counter++ کد اسمبلی توسط تولید کننده

$register_1 = count$
$register_1 = register_1 + 1$
$count = register_1$

اگر در اینجا کوانتوم این پراسس تمام شود (وقفه پایان time slice مثلاً در الگوریتم RR که یک الگوریتم preemptiv است) نوبت به مصرف کننده برسد

counter-- کد اسمبلی توسط مصرف کننده

$register_2 = count$
$register_2 = register_2 - 1$
$count = register_2$

حالا اگر برعکس این روند رو بریم مقدار count می‌تواند ۴ بشود، اگر هم تولید کننده و مصرف کننده به ترتیب اجرا بشه مقدار count همان ۵ باقی می‌ماند، بنابراین در این حالت مشخص نیست در نهایت مقدار count چقدر است و گارانتی ای وجود ندارد که بعد از اجرا این ۲ مقدار count همان ۵ باقی بماند.

حالا فرض کنید مقدار count در ابتدا ۵ باشد، نحوه ی شدنش که همان روال با ۷ است، روال ۴ شدنش هم بصورت زیر است:

T_0:	*producer*	execute	$register_1 = count$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = count$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$count = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$count = register_2$	$\{count = 4\}$

Race condition

به این پدیده Race condition می‌گویند، چون در واقع داره یک رقابت بین ۲ پراسس که دارند بصورت مشترک به متغیر count دستری پیدا می‌کنند به وجود می‌آید و ما باید کاری کنیم که این مشکل پیش نیاید.

مثلاً در همین مثال ما باید کاری کنیم که در اینجا کوانتوم از فرآیند تولید کننده گرفته نشود و به فرآیند مصرف کننده

$register_1 = count$
$register_1 = register_1 + 1$
$count = register_1$

داده بشود و در واقع باید کاری کنیم که وقتی قرارات count++ شود خیالمان راحت باشد که این عمل بصورت کامل انجام می‌شود و بعد مصرف کننده می‌آید و count-- را اجرا می‌کند و همین طور برعکس، یعنی باید مطمئن شویم که وقتی قرارات فرآیند مصرف کننده count-- را انجام بدهد باید خیالمان راحت باشد که این عمل بصورت کامل انجام می‌شود و کد سه خطی زیر تمام می‌شود و بعد تولید کننده بیاید و count++ را اجرا کند

$register_2 = count$
$register_2 = register_2 - 1$
$count = register_2$

* اگر این کار را نتوانیم انجام بدهیم این Race condition منجر به رخداد ناسازگاری (inconsistency) در متغیر count می‌شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

حالا ما می‌خواهیم روش‌هایی را یاد بگیریم که جلوی این مشکلات را بگیریم تا ناسازگاری در داده‌ها یمان به وجود نیاید

critical section (ناحیه بحرانی): در یا چند فرآیند روی یک قسمتی از کدشان با هم رقابت داشته باشند، در این شرایط به آن قسمت از کد ناحیه بحرانی می‌گویند

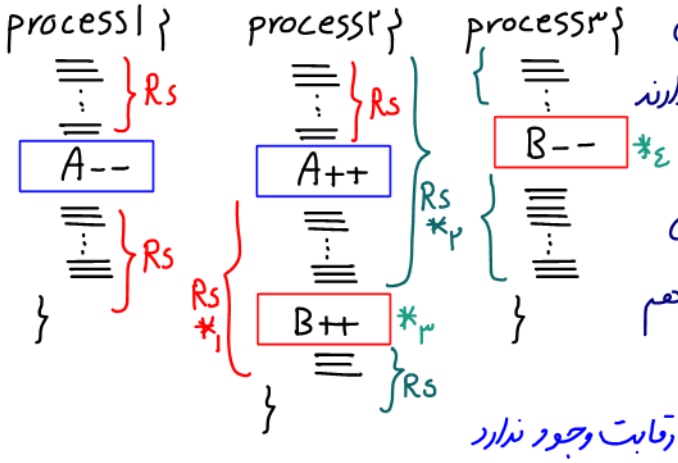
```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

توجه) ما در ادامه روی ۲ فرآیند تمرکز می‌کنیم: قسمتی از P_i که نسبت به P_j رویش رقابت وجود دارد

- در مثال قبل متغیر count بین فرآیند تولید کننده و مصرف کننده یک ناحیه بحرانی بود

مثال) فرض کنید ۳ فرآیند P_1 و P_2 و P_3 داشته باشیم

اگر حواسمان به متغیر A که هم در P_1 و هم در P_2 استفاده می‌شود نباشد می‌تواند باعث به وجود آمدن ناسازگاری شود، بنابراین این ۲ باکس آبی ناحیه بحرانی محسوب می‌شود و P_1 و P_2 روی این قسمت‌ها با هم رقابت دارند و باید این قسمت‌های P_1 و P_2 را از هم محافظت کنیم.



P_2 و P_3 هم روی متغیر B با هم رقابت دارند و از دید P_2 و P_3 این باکس‌های قرمز ناحیه بحرانی محسوب می‌شود و ما باید این قسمت‌های P_2 و P_3 را از هم محافظت کنیم.

remainder section: قسمتی از برنامه P_i که نسبت به P_j رویش رقابت وجود ندارد

بین P_1 و P_2 قسمت‌هایی که با پرانتز قرمزشان داده شده ناحیه باقیمانده‌شان است. توجه کنید که اگرچه ما در پایین P_2، متغیر B++ است که یک ناحیه بحرانی نسبت به P_3 محسوب می‌شود ولی نسبت به P_1 این ناحیه *۱ ناحیه باقیمانده محسوب می‌شود و رویش رقابتی وجود ندارد. ناحیه باقیمانده بین P_2 و P_3 نیز در شکل بالا با پرانتزهای سبز مشخص شده، حالا قسمت *۲ نسبت به P_3 ناحیه باقیمانده محسوب می‌شود چون P_3 هیچ رقابتی با این قسمت P_2 ندارد.

از دید P_3، *۳ ناحیه بحرانی است و از دید P_2 در P_3، *۴ ناحیه بحرانی است، پس بین P_2 و P_3 باید *۳ و *۴ را نسبت به هم محافظت کنیم.

* آیا می‌توانیم critical section را طوری حفاظت کنیم که وقتی یک پراسس در ناحیه بحرانی خودش است، هیچ یک از پراسس‌های رقیب در آن ناحیه بحرانی نباشند، به این معنا که این نواحی CS نسبت به هم حفاظت بشوند.

اگر بتوانیم چنین حفاظتی بین پراسس‌ها به وجود آوریم یعنی توانستیم مسئله ناحیه بحرانی را حل کنیم

این مسئله به این صورت حل می‌شود که ما یک گیت ورود قبل از ناحیه بحرانی و یک گیت خروج بعد از ناحیه بحرانی در نظر می‌گیریم حال اگر یک پراسس بخواهد وارد ناحیه بحرانی اش شود باید از گیت ورود (entry section) عبور کند (باید از طریق entry section وارد ناحیه بحرانی اش شود و اگر پراسس می‌خواهد از ناحیه بحرانی اش خارج شود باید از طریق گیت خروج اش خارج شود و پس وارد remainder section خودش بشود.

- حال اگر ما بتوانیم قطعه برنامه‌هایی را در ناحیه ورود و ناحیه خروج قرار بدهیم بطوریکه وقتی یک پراسس می‌خواهد وارد ناحیه بحرانی اش شود و از ناحیه ورود عبور می‌کند مطمئن باشیم که پراسس‌های رقیب در حال کار کردن (اجرا) در ناحیه بحرانی این فرآیند نیستند، تضمین می‌شود که ناحیه بحرانی بصورت انحصاری و فقط توسط یکی از آنها در حال اجرا است و بنابراین تداخلی که منجر به ناسازگاری (inconsistency) می‌شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

روی داده ناحیه بحرانی بشود رخ نخواهد داد.

- فرض کنید این کار را انجام دادیم، حالا اگر P_1 در ناحیه بحرانی اش باشد یعنی P_1 روی A-- کار کند، حالا ما مطمئن هستیم که P_2 وقتی می خواهد از گیت ورودی که برای A++ اش گذاشتیم عبور کند همان جا گیر می افتد و نمی تواند وارد ناحیه بحرانی A++ اش بشود، حالا اگر در همین شرایط P_3 وارد ناحیه بحرانی خودش نشده باشد آن گاه اگر P_2 بخواهد از گیت ورود مربوط به B++ عبور بکند باید اجازه این کار را داشته باشد چون P_3 که رقیب P_2 روی این ناحیه بحرانی است در Rs خودش است و وارد ناحیه بحرانی خودش نشده است، حالا اگر P_2 رفت توی ناحیه بحرانی B++ و داشت روی B کار می کرد اون وقت P_3 باید پشت ناحیه ورودش گیر بکند. حالا اگر بتوانیم با استفاده از گیت ورود و خروج اجازه ندهیم که در هر لحظه بیش از یک رقیب در ناحیه بحرانی اش باشد راه حل ناحیه بحرانی را ارائه داده ایم.

راه حل ناحیه بحرانی = طراحی ۲ گیت ورود و خروج بصورت صحیح است.

* یک راه حل ناحیه بحرانی یعنی طراحی گیت ورود و گیت خروج درست انجام شده اگر ۳ خاصیت زیر برقرار باشد: راه حل صحیح ناحیه بحرانی

۱- انحصار متقابل (mutual exclusion): در صورتی این شرط برقرار است که مطمئن باشیم وقتی یک پراسس توی ناحیه بحرانی خودش قرار دارد هیچ یک از پراسس های رقیب در ناحیه بحرانی نیستند و یا در remainder section های خودشون هستند و یا در پشت گیت های ورود خودشون گیر کردند

۲- پیشرفت (progress) این شرط برقرار است اگر ۲ خاصیت زیر برقرار باشد.

۲-۱: اگر یک پراسس در ناحیه باقیمانده خودش است در رقابت شرکت داده نشود و جز رقبا محسوب نشود

۲-۲: پراسس ها دچار انسداد نامحدود نشوند (indefinite Blocking)، یعنی حالتی پیش نیاید که بصورت نامحدود یک یا چند پراسس گیر کنند حالت خاص پیش نیاید که یک یا چند پراسس بصورت نامحدود (از نظر زمانی) و بی پایان پشت ناحیه بحرانی گیر کنند و نتوانند از این حالت خارج شوند

۳- انتظار محدود (Bounded waiting): انتظار یک پراسس برای ورود به ناحیه بحرانی اش محدود باشد و بالاخره وارد شود

وقتی پراسس می خواهد وارد ناحیه بحرانی خودش شود و پشت گیت ورودش گیر کرده و منتظر ورود به ناحیه بحرانی است و پراسس های رقیب موفق می شوند وارد این ناحیه بحرانی بشوند تعداد دفعاتی که می بینند که پراسس های رقبا می توانند وارد ناحیه بحرانی مذکور شوند محدود باشد (تعداد دفعات موفقیت رقبا محدود باشد و بالاخره نوبت به این پراسس برسد = کرانی روی تعداد دفعاتی که یک پراسس می تواند پشت سرهم وارد ناحیه بحرانی بشود وجود داشته باشد

* شرط انتظار محدود به تعداد دفعات ورود رقبا ربط دارد و نه به مدت زمانی که رقبا در ناحیه بحرانی شون هستند

Two-Process Solutions

الگوریتم ۱: نوبت دهی

ابتدا یک الگوریتم ۲ فرآیندی را بررسی می کنیم و فرض می کنیم فقط ۲ فرآیند می خواهند برای ورود به ناحیه بحرانی با هم رقابت کنند. پس ما دو فرآیند P_1 و P_2 یک حلقه بی نهایت while(true) دارند و بنابراین مکررا این فرآیند ها دارند وارد ناحیه بحرانی خودشون می شوند و پس از ناحیه بحرانی خارج می شوند و وارد Rs خودشون می شوند.

- Algorithm 1
  - Shared: int turn; ← از یک متغیر turn استفاده شده که بین ۲ فرآیند P_1 و P_2 مشترک است و می تواند مقداره ۰ و ۱ داشته باشد

```
Pi:
while (TRUE) {
  while (turn!=i);   entry section
  critical-section
  turn = 1-i;        exit section
  remainder-section
}
```

هر فرآیند در حلقه while خودش مخالف بودن turn را با i که اندیس خودش است چک می کند

پس P_0 مخالف بودن turn با ۰ را چک می کند و P_1 مخالف بودن turn با ۱ را چک می کند

P_0	P_1
turn!=0	turn!=1
turn=1	turn=0

turn=0 یعنی نوبت پراسس ۰ برای ورود به CS است.

" " " " " " 1 " " " turn=1

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

- فکر کنید P_i داره اجرا میشه : P_i { turn!=0 } : اگر turn صفر باشه دیگه تو حلقه while(turn!=i) گیر نمی‌کنیم، چون نوبت خودش الان وارد CS خودش می‌شود و وقتی داره میاد بیرون turn را 1 می‌کند و با این کار برای دفعه بعدی نوبت را به پراسس 1 میدهد

* P_0 در حلقه while می‌یاد مخالف بودن turn رو با ایندکس خودش یعنی 0 چک می‌کنه و P_1 مخالف بودن turn را با ایندکس خودش یعنی 1 چک می‌کند

* هر پراسس turn را چک می‌کند و اگر نوبت خودش بود وارد CS اش می‌شود و موقع خروج نوبت را به طرف مقابل می‌دهد، یعنی P_0 موقع خروج از ناحیه بحرانی turn را 1 می‌کند و P_1 موقع خروج از ناحیه بحرانی خودش turn را 0 می‌کند

حال بیایید بررسی کنیم که آیا این راه حل یک راه حل صحیح ناحیه بحرانی است یا خیر؟

۱- انحصار متقابل (mutual exclusion): در صورتی این شرط برقرار است که مطمئن باشیم وقتی یک پراسس توی ناحیه بحرانی خودش قرار دارد هیچ یک از پراسس‌های رقیب در ناحیه بحرانی نیستند و یا در remainder section های خودشون هستند و یا در پشت گیت‌های ورود دشون گیر کردند

برقرار است: وقتی یک پراسس وارد ناحیه بحرانی اش می‌شود turn برای آن پراسس است و پراسس مقابل پشت حلقه while اش گیر می‌کند.

۲- پیشرفت (progress) این شرط برقرار است اگر ۲ خاصیت زیر برقرار باشد.

۲-۱: اگر یک پراسس در ناحیه باقیمانده خودش است در رقابت شرکت داده نشود و جز رقبا محسوب نشود

۲-۲: پراسس‌ها دچار انسداد نامحدود نشوند (Indefinite Blocking)، یعنی حالتی پیش نیاید که بصورت نامحدود یک یا چند پراسس گیر کنند حالت خاص پیش نیاید که یک یا چند پراسس بصورت نامحدود (از نظر زمانی) و بی‌پایان پشت ناحیه بحرانی گیر کنند و نتوانند از این حالت خارج شوند

بررسی خاصیت ۱) اگر P_0، CS اش را پشت سر بگذارد و نوبت را به P_1 بدهد (turn را 1 کند) و وارد ناحیه باقیمانده خودش بشود حالا فرض کنید P_0 فعلا نمی‌خواهد وارد ناحیه بحرانی اش شود و دارد توی RS اش یکسری کارها انجام می‌دهد ولی نوبت به P_0 برسد و کارهای درون RS اش را تمام کند و به انتهای حلقه برسد و دوباره به بالا برگردد و بخواهد وارد CS خودش شود ولی turn=1 است و نوبت P_1 است، ولی P_1 اصلا فعلا نمی‌خواهد وارد ناحیه بحرانی اش شود و پراسس P_0 توی while(turn!=0) گیر می‌کند و نمی‌تواند وارد ناحیه بحرانی اش شود، چون turn برابر 1 است و مساوی 0 نیست

- Algorithm 1
  - Shared: int turn;

```
Pi:
while (TRUE) {
  while (turn!=i);
  critical-section
  turn = 1-i;
  remainder-section
}
```

پس P_1 که در ناحیه باقیمانده خودش هست داره توی رقابت شرکت می‌کند و جلوی پیشرفت P_0 را می‌گیرد، پس شرط اول پیشرفت برقرار نیست چون شرط اول اش برقرار نیست.

پس این راه حل نمی‌تواند یک راه حل درست دوفرآیندی برای ناحیه بحرانی باشد.

۳- انتظار محدود (Bounded waiting)

تعداد دفعاتی که یک فرآیند متقاضی ورود به ناحیه بحرانی می‌بیند که پراسس‌های رقیبش توانند وارد ناحیه بحرانی مذکور شوند محدود باشد

برقرار است. اگر P_0 می‌خواهد وارد ناحیه بحرانی بشود ولی پشت گیت ورود گیر کرده یکبار که P_1 بره توی CS اش و بیاد بیرون نوبت پراسس P_0 می‌شود، پس تعداد دفعاتی که P_1 می‌بیند و انتظار P_0 محدود خواهد بود به یکبار و این شرط برقرار است.

حالا می‌خواهیم یک راه حل دیگه دوفرآیندی را بررسی کنیم:

در کد زیر نیز یک حلقه do داریم که همیشه اجرا می‌شود و بدنه فرآیندها (برنامه‌ها) در آن قرار می‌گیرد

در این روش از یک آرایه‌ای از فلگ‌ها استفاده می‌شود، حالا چون ما داریم در مورد ۲ فرآیند صحبت می‌کنیم، آرایه فلگ‌ها دو تا عنصر

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

دارد، آرایه فلگ از جنس boolean است

هر فرآیند یک فلگ دارد، فلگ ها اول کار به False مقداردهی اولیه می شوند، حالا هرکسی که بخواهد وارد ناحیه بحرانی اش شود توی ناحیه ورود فلگ خودش را true می کند به این معنا که من متقاضی ورود به ناحیه بحرانی هستم ولی فلگ رقیب اش را چک می کند که ببیند اگر فلگ رقیب اش هم true است که یعنی رقیب اش هم متقاضی ورود هست روی خط while (flag[j]) صبر می کند ولی اگر فلگ رقیب false است که به این معنات که رقیب اش در ناحیه باقیمانده خودش است و بنابراین این پراسس از این while عبور می کند و وارد cs می شود و در انتها بعد از خروج از ناحیه بحرانی اش فلگ اش را False می کند

پس در این راه حل یک فرآیند اول وضعیت رقیب را چک می کند و اگر رقیب در RS باشد این فرآیند از گیت ورود عبور می کند و اگر رقیب متقاضی ورود بود یا در cs اش بود در گیت ورود توقف می کند

```
Algorithm 2
  Shared: boolean flag[2];
  Pi:
  flag[i] = FALSE;
for ever ← do {
    flag[i] = TRUE;        ناحیه ورود
    while (flag[j]);
    critical-section
    flag[i] = FALSE;       ناحیه خروج
    remainder-section
  } while (TRUE);
```

۱- انحصار متقابل (mutual exclusion): درصورتی این شرط برقرار است که مطمئن باشیم وقتی یک پراسس توی ناحیه بحرانی خودش قرار دارد هیچ یک از پراسس های رقیب در ناحیه بحرانی نیستند و یا در remainder section های خودشون هستند و یا در پشت گیت های ورود دشون گیر کردند

برقرار است: چون هیچ وقت دو تا پراسس نمی توانند با هم وارد ناحیه بحرانی بشوند، چون اگر پراسس i قبل از پراسس j وارد ناحیه بحرانی شود که به این معنات که فلگ اش true بوده که توانسته وارد cs اش شود وقتی پراسس j در while خودش فلگ پراسس i را چک می کند فلگ پراسس i را true می بیند و روی خط while گیر می کند و بنابراین ۲ فرآیند با هم دیگر نمی توانند وارد cs بشوند

۲- پیشرفت (progress) این شرط برقرار است اگر ۲ خاصیت زیر برقرار باشد.

۲-۱: اگر یک پراسس در ناحیه باقیمانده خودش است در رقابت شرکت داده نشود و جز رقبا محسوب نشود

۲-۲: پراسس ها دچار انسداد نامحدود نشوند (indefinite Blocking)، یعنی حالتی پیش نیاید که بصورت نامحدود یک یا چند پراسس گیر کنند حالت خاص پیش نیاید که یک یا چند پراسس بصورت نامحدود (از نظر زمانی) و بی پایان پشت ناحیه بحرانی گیر کنند و نتوانند از این حالت خارج شوند

بررسی شرط ۱: برقرار است: کسی که در ناحیه باقیمانده است نتواند در رقابت شرکت کند، که چون کسی که می خواهد از cs خارج شود و وارد ناحیه باقیمانده خودش شود فلگ خودش را false می کند و در رقابت شرکت نمی کند

بررسی شرط ۲: برقرار نیست: اگر پراسس i فلگ خودش را true کند و دقیقاً قبل از while کوانتوم اش تمام شود و نوبت به پراسس دوم برسد و پراسس دوم هم بخواهد وارد ناحیه بحرانی شود و برود فلگ خودش را true کند (فلگ j خودش)، در این شرایط وقتی پراسس دوم به while می رسد چون فلگ i هم true است و روی خط while گیر می کند تا کوانتوم اش تمام شود، حال وقتی کوانتوم به پراسس i برسد، i هم چون فلگ j، true است روی خط while خودش گیر می کند و این انسداد پایان ندارد و indefinite Blocking داریم.

شرایطی به وجود آمد که هر ۲ فلگ فرآیند ۱ و ۲، true هستند و هر ۲ فرآیند روی خط while(flag[]) گیر می کنند

پس چون شرط دوم پیشرفت برقرار نیست پیشرفت حاصل نمی شود.

۳- انتظار محدود (Bounded waiting)

تعداد دفعاتی که یک فرآیند متقاضی ورود به ناحیه بحرانی می بیند که پراسس های رقبای می توانند وارد ناحیه بحرانی مذکور شوند محدود باشد

برقرار است: اگر یک پراسس روی خط while گیر کند، پراسس رقیب یا به دلیل همان indefinite Bloking که در بالا گفتیم هرگز موفق نمی شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

وارد ناحیه بحرانی اش شود یا اینکه حداکثر یکبار واردمی شود و فلگ خودش را false می‌کند و پس نوبت پراسس رقیب می شود و بنابراین پراسس i (ول) می بیند که رقیب حداکثر یکبار وارد ناحیه بحرانی اش می شود، پس waiting یک پراسس از نظر تعداد دفعات ورود رقیب Bounded است.
پس بطورکلی این راه حل، یک راه حل صحیح ۲ فرآیندی برای حل ناحیه بحرانی نیست.

Peterson's Solution

در این راه حل هم از فلگ استفاده می‌شود و هم از نوبت دهی و به این شکل است که پراسس که می‌خواهد وارد ناحیه بحرانی شود ابتدا فلگ خودش را true می‌کند و نوبت را به پراسس رقیب می‌دهد که اگر پراسس رقیب هم فلگ اش true است، پراسس رقیب وارد شود، پراسس j هم همین کار را می‌کند، یعنی اگر فرآیند i هم بخواهد وارد cs بشود، فلگ j خودش را true می‌کند و نوبت را به فرآیند i می‌دهد، حالا در این الگوریتم اگر حالتی پیش بیاید که فلگ هر ۲ پراسس نیز true بشوند چون turn یا i هست یا j (بسته به اینکه در نهایت کدامتوم دست چه پراسس بوده است یکی از این پراسس ها بوده که آخرین بار دستور turn را اجرا کرده و نوبت را به دیگری داده)، پس حالا وقتی به حلقه while می رسیم ۲ تا چیز را چک می‌کنیم ۱- آیا فلگ رقیب true است؟
۲- آیا نوبت رقیب هست؟

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

    /* critical section */

    flag[i] = false;

    /*remainder section */
}
```

entry section

exit section

اگر هر دوی اینها برقرار باشد این پراسس روی خط while صبر می‌کند.
ولی اگر نوبت این پراسس باشد مستقل از اینکه فلگ رقیب چی هست واردمی شود
یا اگر فلگ رقیب false باشد مستقل از اینکه نوبت چه کسی است بازهم واردمی شویم
و نهایت پراسس که وارد cs شود در انتها فلگ خودش را false می‌کند

در اینجا هر سه شرط برقرار است

mutual exclusion: برقرار است: اگر دو تا پراسس فلگ شون true باشد به دلیل turn فقط یک پراسس واردمی شود و پراسس دیگر در ناحیه ورود گیر می‌کند پس ۲ تقریبا هم نمی‌توانند وارد ناحیه بحرانی شوند.

progress: شرط ۱: این بود که کسی که توی ناحیه باقیمانده است در رقابت شرکت نکند، که چون یک پراسس بعد از خروج از ناحیه بحرانی فلگ خودش را false می‌کند، باعث توقف رقیب اش نمی شود و در رقابت شرکت می‌کند

شرط ۲: این بود که دچار indefinite blocking نشویم، مشاهده می‌کنیم که اگر هر دو تا فلگ هم true باشد به خاطر اینکه متغیر turn در نهایت فقط متعلق به یکی از پراسس ها خواهد بود یکی شون از while عبور می‌کند و دچار انتظار نامحدود نمی‌شویم

Bounded waiting: این بود که یک پراسس متقاضی ورود به ناحیه بحرانی که پشت ناحیه بحرانی گیر کرده مشاهده کند که رقیب اش به تعداد محدود می‌تواند وارد ناحیه بحرانی شود، فرض کنید پراسس i روی خط while گیر کرده که بدان معنات که فلگ رقیب اش true است و turn (نوبت) هم متعلق به رقیب اش است (توجه کنید که در این حالت فلگ i نیز true است) حالا وقتی پراسس j از ناحیه بحرانی خارج شود فلگ خودش را False می‌کند و حالا وقتی اگر این پراسس (j) سریع دور بزند و بخواهد مجددا وارد ناحیه بحرانی شود، چون نوبت را به پراسس i می دهد و فلگ i هم true است روی خط while گیر می‌کند (while دارد فلگ و نوبت پراسس رقیب را چک می‌کند) و بنابراین وقتی کدامتوم به i برسد، i می‌تواند از while عبور کند و وارد cs خودش شود، بنابراین پراسس i می بیند که رقیب حداکثر یکبار وارد ناحیه بحرانی اش می شود، پس waiting یک پراسس از نظر تعداد دفعات ورود رقیب Bounded هست.

* پس راه حل peterson را یک راه حل درست ناحیه بحرانی برای ۲ فرآیند می‌دانیم، البته باید بتوانیم راه حل ی را به n فرآیند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

تصمیم بر هیم.

قبل از ورود به ارائه راه حل های n فرآیندی ابتدا نیاز است با Hardware synchronization آشنا شویم:

Synchronization Hardware

به این معنا است برخی از instruction ها خاص توسط یک Hw در اختیار ما قرار داده شوند که بتوان در سرفصل طراحی راه حل های نواحی بحرانی راکم کنیم.

Atomic instruction: به یک instruction اتمیک گوییم اگر تجزیه ناپذیر باشد، یعنی نشود وسط اجرا آن دستور وقفه بیاید و اجرایش تکه شود. مثلاً برای راحتی فکر کنید سخت افزار این inst را مثلاً در یک کلاک اجرا می کند

یکی از این Atomic inst ها که در بعضی از Hw وجود دارد دستور TestAndSet است، که بصورت زیر عمل می کند:

یک متغیر Boolean بعنوان ورودی می گیرد (که این متغیر می تواند ۲ مقدار true و false را به خودش بگیرد) و مقدار آن متغیر چه true باشد و چه false در یک متغیر دیگر ذخیره می کند (rv)، سپس متغیر ورودی ای که گرفته است را true می کند و true را داخل همان متغیر ورودی می ریزد و مقدار اولیه آن را که در یک متغیر کمکی ذخیره کرده بود را بعنوان خروجی برمی گرداند (rv را برمی گرداند)

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

(target) true → TestAndset
- مقداری که برمی گرداند = true
- target جدید: مقدار جدید ورودی داده شده = true

(target) false → TestAndSet
- مقداری که برمی گرداند = false
- target جدید: مقدار جدید ورودی داده شده = true

راه حل برای n فرآیند با استفاده از دستور TestAndSet ← ۲ مشکل دارد
۱- راه حل صحیح ناحیه بحرانی نیست
۲- busy waiting دارد

(خلاصه: چون lock در ابتدا false است، هرکسی lock را بگیرد = آن را true کند، وارد ناحیه بحرانی اش می شود)

lock در اینجا یک متغیر مشترک بین n پراسس است که می خواهد وارد یک cs شفص شوند

در ابتدا کار مقدار lock برابر false است، حال از این n تا پراسس آن پراسسی که برای اولین بار و در while قبل از ورود به ناحیه بحرانی اش TestAndSet را اجرا می کند lock را true می کند و مقدار TestAndSet اش false می شود، بنابراین خود آن پراسس شرط while برایش برقرار نیست و در نتیجه while عبور می کند و وارد cs می شود ولی بعد از آن کلاک دیگه مقدار lock برابر true است و اگر بعد از این کلاک یک پراسس دیگر TestAndSet را اجرا بکند مقدار برگشت داده شده (return شده) به TestAndSet اش true است و بنابراین شرط while برای آن پراسس true می شود و روی خط while گیر می کند و نمی تواند وارد ناحیه بحرانی اش شود، این داستان تا زمانی ادامه دارد که پراسسی که در ناحیه بحرانی اش بود، این ناحیه را ترک کند، در اینصورت lock را false می کند که در این شرایط مقدار TestAndSet یکی از پراسس های دیگر که زمان بهشون میده و این دستور را اجرا می کند می تواند false شود و در نتیجه از حلقه while اش عبور می کند و وارد ناحیه بحرانی خودش می شود.

```
do{ do for ever
   while (test_and_set(&lock))
      ; /* do nothing */          entry section

      /* critical section */

   lock = false;                  exit section

      /* remainder section */
} while (true);
```

حال بیایید بررسی کنیم که آیا این راه حل یک راه حل صحیح ناحیه بحرانی است یا خیر؟

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۱- mutual exclusion: برقرار است: دو تا پراسس با هم نتوانند وارد ناحیه بحرانی شوند، قطعا فقط یک پراسس می‌تواند وارد ناحیه بحرانی باشد و بقیه پراسس ها پشت حلقه while شون گیر می‌کنند پس این خاصیت برقرار است.

۲- progress: شرط ۱: این بود که کسی که توی ناحیه باقیمانده است در رقابت شرکت نکند، که چون یک پراسس بعد از خروج از ناحیه بحرانی lock را false می‌کند و بعد وارد ناحیه باقیمانده اش می‌شود و بنابراین رقابت نمی‌کند برقرار است

شرط ۲: پراسس ها دچار انسداد نامحدود نشوند (indefinite Blocking)، یعنی حالتی پیش نیاید که بصورت نامحدود یک یا چند پراسس گیر کنند حالت خاص پیش نیاید که یک یا چند پراسس بصورت نامحدود (از نظر زمان) و بی پایان پشت ناحیه بحرانی گیر کنند و نتوانند از این حالت خارج شوند هیچ وقت این اتفاق نمی‌افتد که هر پراسس ها پشت حلقه while شون گیر کنند پس پراسس ها دچار انسداد نامحدود نمی‌شوند

۳- انتظار محدود (Bounded waiting): برقرار نیست

تعداد دفعاتی که یک فرآیند متقاضی ورود به ناحیه بحرانی می‌بیند که پراسس‌های رقبا می‌توانند وارد ناحیه بحرانی مذکور شوند محدود باشد

فرض کنید یک پراسس که همین الان cs را ترک کرده هنوز کوانتوم اش تمام نشده و موفق می‌شود دور بزند و برگردد بالا (یعنی RS کوچک و کوتاهی داشته باشد) و وارد while شود و چون هنوز look برابر false است و کسی نتوانسته آن را بگیرد = آن را true کند، چون کوانتوم هنوز به بقیه نرسیده و کوانتوم دست خود همین فرآیند است، دوباره خودش می‌تواند lock = TestAndSet را انجام دهد و چون خروجی TestAndSet برابر false خواهد شد، شرط while اش دوباره false می‌شود و مجددا وارد ناحیه بحرانی می‌شود، حالا اگر وسط اجرا کردن ناحیه بحرانی اش کوانتوم اش تمام شود و نوبت به پراسس های رقیب اش برسد چون lock برابر true، حاصل اجرا TestAndset سایر فرآیندها true می‌شود و بنابراین شرط while شون true می‌شود و بنابراین کماکان روی خط while شون گیر می‌کنند تا مجددا نوبت به این پراسس می‌رسد و این ماجرا تکرار می‌شود بنابراین کران روی تعداد دفعاتی که یک پراسس متقاضی ورود به cs مشاهده می‌کند که یکی از رقبا می‌تواند مکررا وارد ناحیه بحرانی اش شود وجود ندارد

Atomic instructions: Swap

۱- راه حل جمیع ناحیه بحرانی نیست ۲ تا مشکل دارد ۲- busy waiting دارد

یک دستور اتمیک دیگه داریم به نام swap که این دستور دوتا متغیر Boolean می‌گیرد و جای این ۲ متغیر Boolean را در یک کلاک عوض می‌کند.

کد زیر یک راه حل n فرآیندی برای حل مشکل ناحیه بحرانی را نشان می‌دهد

در این الگوریتم از ۲ متغیر look و Key استفاده شده، متغیر lock بین همه n فرآیند shared است و متغیر Key لوکال هر کدام از n فرآیند است (هر فرآیند یک متغیر Key ـ لوکال دارد)

هر پراسس که می‌خواهد وارد ناحیه بحرانی اش شود متغیر Key اش را true می‌کند و در یک حلقه while برون true بودن Key را چک می‌کند و تا مادامیکه Key=true است ۲ تغیر lock و Key را swap می‌کند و برین ترتیب اگر true/lock باشد کماکان Key، true می‌ماند و این حلقه while ادامه دارد و اگر که lock برابر false بشود در دستور swap که اتمیک است و یک کلاک طول می‌کشد Key=false می‌شود و بنابراین lock=true می‌شود در این حالت وقتی دوباره می‌آییم توی while ولی این بار شرط while، false می‌شود و بنابراین دستور swap دیگر اجرا نمی‌شود (از روی آن می‌پریم) و پراسس وارد ناحیه بحرانی می‌شود، و در این حالت بقیه پراسس ها lock را true می‌بینند و بنابراین وقتی Key و lock را swap می‌کنند، Key شان true می‌شود و در حلقه while شان گیر می‌کنند و بنابراین شرط انحصار متقابل برقرار است، حالا پراسس که داره ناحیه بحرانی اش را ترک می‌کند lock را

```
void Swap(boolean *a, boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
do {
  key = TRUE;
  while (key == TRUE)
    Swap(&lock, &key);   // entry section

    // critical section

  lock = FALSE;          // exit section

    // remainder section
}while (TRUE);
```

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

false می‌کند، پس حالا یک پراسس دیگر می‌تواند با swap کردن Key و lock اش وارد ناحیه بحرانی اش بشود و بنابراین پراسس که ناحیه بحرانی اش را ترک می‌کند و وارد Rs خودش می‌شود در رقابت شرکت نمی‌کند و بنابراین شرط ۱ پیشرفت برقرار است وهمین‌طور هیچ‌وقت هم پراسس ۶ روی خط while گیر نمی‌کند و بنابراین انتظار نامحدود نداریم، ولی در اینجا هم مثل الگوریتم قبلی یک پراسس می‌تواند lock را false کند و برگردد بالا و دوباره خودش lock را در اختیار بگیرد و بنابراین کرانی روی تعداد دفعاتی که یک پراسس می‌تواند وارد ناحیه بحرانی بشود وجود ندارد حالا اگر این فرآیند وسط اجرای کدهای ناحیه بحرانی اش کوانتوم اش تمام شود و نوبت به پراسس های رقیب اش برسد چون lock را برابر true می‌بینند بنابراین شرط while شون true می‌شود و بنابراین کماکان روی خط while شون گیر می‌کنند تا مجددا نوبت به این پراسس می‌رسد و این ماجرا تکرار می‌شود بنابراین کرانی روی تعداد دفعاتی که یک پراسس متقاضی ورود به CS مشاهده می‌کند که یکی از رقبا می‌تواند مکررا وارد ناحیه بحرانی اش شود وجود ندارد

پس این راه حل نیز یک راه حل صحیح n فرآیندی محسوب نمی‌شود

Atomic instruction : compare-and-swap

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}

while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

در این روش هم مشکل حل نمی‌شه.

شکل busy waiting = Spin lock

در راه حل هایی که شلا با دستورات اتمیک TestAndSet و یا swap داشتیم آنجایی که یک پراسس lock را در اختیار ندارد کل بسته زمانیکه به آن پراسس اختصاص داده شده در یک حلقه while سپری می‌شود و در آنجا wait می‌کند، شلاً دارد lock را Test and set می‌کند وچون lock، True است نتیجه ای که Test and set برمی‌گرداند True می‌ماند و کل بسته زمانی اش را دارد در اجرا دستور Test and set می‌گذارند وچون در این مدت lock، false نمی‌شود، کاری که فرآیندهای دیگر به جز فرآیندی که در ناحیه بحرانی اش است کاری بیهوده است و وهیچ کار مفیدی انجام نمی‌دهند و فقط وقت cpu هدر می‌رود، cpu را Busy نگه می‌داریم ولی فرآیندهای دیگر فقط دارند wait می‌کنند و در واقع در یک حالت چرخشی قفل شدیم، وقتی ما مطمئن هستیم که این فرآیندها مرتفق نمی‌شوند وارد ناحیه بحرانی شوند چرا کل بسته زمانی شان را هدر بدهیم ؟!

راه حل های قبلی که از دستورات اتمیک استفاده می‌کردند، هم چون شرط Bounded waiting در آنها برقرار نبود یک راه حل درست ناحیه بحرانی نبودند وهم دارای مشکل spin lock بودند و زمان cpu هم در آنها هدر می‌رفت

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

یک راه حل صحیح n فرآیندی با استفاده از Hw synchronization

حالا در ابتدا روی حل شکل bounded waiting تمرکز می‌کنیم و سپس روی حل شکل Busy waiting

راه حلی که می خواهیم ارائه بدهیم یک راه حل n فرآیندی است که یک متغیر shave به اسم lock بین n فرآیند داره و یک آرایه به تعداد پرانتز ۴ از نوع Boolean به اسم waiting دارد و بنابراین چون n تا فرآیند داریم waiting ما n تا عنصر دارد.

```
boolean waiting[n];
int lock;
```

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

entry section

exit section

از متغیرهای waiting در قسمت entry section استفاده شده است، اگر waiting برای پراسس i true باشد نشان می دهد که این پراسس در مرحله ای است که می خواهد وارد ناحیه بحرانی اش بشود

حالا پراسس متغیر Key را در ابتدا کار true می‌کند و متغیر Key، حاصل TestAndSet ـ متغیر lock را در حلقه while دریافت می‌کند، حال اگر lock برابر false باشد حاصل TestAndset و در نتیجه مقدار Key برابر false می‌شود و از حلقه while خارج می‌شیم و اگر lock=true باشد کماکان در این حلقه باقی می ماند، حالا اگر از حلقه بیرون آمدیم waiting این پراسس را false می‌کنیم و این یعنی دوران انتظار برای ورود تمام شده و وارد cs می‌شود. حال وقتی پراسس می خواهد از cs اش خارج شود برای آنکه نوبت رعایت بشود، از پراسس که شماره اش بعد از شماره خودش است شروع می‌کند و در یک حلقه while چک می‌کند (مادا میله دوباره به شماره پراسس خودش برنگشته باشد) تا اولین پراسس بعد از خودش را پیدا کند که waiting اش true باشد که true بودن waiting یکی از پراسس‌های بعدی یعنی رعایت که آن پراسس در ناحیه ورودش منتظر ورود به ناحیه بحرانی اش است.

یک بیان دیگه از جمله بالا:

وقتی پراسس از ناحیه بحرانی اش اومد بیرون تا زمانیکه waiting پراسس‌های بعد از خودش برابر false باشد شماره پراسس بعدی یعنی j را یکی یکی جلو می برد (j را j+1 می‌کند: j=j+1) تا به اولین پراسس برسد که waiting اش true است. پس چی شد؟

میایم از پراسس بعدی این پراسس (یعنی با شروع از i+1=j) یکی یکی پراسس‌های بعدی را بررسی می‌کنیم تا به اولین پراسس برسیم که waiting اش true باشد و در این حالت ما یک پراسس پیدا کردیم که می خواهد وارد ناحیه بحرانی اش شود

یا داریم: از متغیرهای waiting در قسمت entry section استفاده شده است، اگر waiting برای پراسس i true باشد نشان می دهد که این پراسس در مرحله ای است که می خواهد وارد ناحیه بحرانی اش بشود

ادامه: حالا اگر این پراسس یک waiting ـ true پیدا کرد این پراسس از این حلقه while خارج می‌شود. فقط توجه کنید که با توجه به شروطی که برای while گذاشتیم، یعنی: ۱) i!=j ۲) !waiting[j] خروج از این حلقه while به ۲ دلیل می‌تواند باشد

۱- یا به این دلیل است که i=j می‌شود، یعنی هیچ فرآیندی وجود ندارد که بخواهد وارد cs خودش شود که در اینصورت این فرآیند می‌تواند lock را آزاد کند، که برای این منظور lock را false می‌کند

۲- یا به این دلیل است که یک پراسس با waiting=true پیدا کرده، که در اینصورت شرط if(j==i) برقرار نیست و وارد قسمت else می‌شود و waiting آن پراسس را (j) (پراسس رقیب را) false می‌کند و در این حالت پراسس j می‌تواند وارد cs خودش بشود. چرا؟

چون پراسس j قبلا در while قبل از ورود به ناحیه بحرانی اش گیر کرده بود، چون هم waiting اش true بود و هم به واسطه true بودن lock اش

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

متغیر Key اش هم true بود (توجه کنید که پراسسی که درون ناحیه بحرانی است lock را true کرده و وارد ناحیه بحرانی شده و بنابراین تمای پراسس ها رقیب دیگر lock شان true است) و بنابراین شرط while اش true بود و در حلقه دور می‌زد و الان که waiting[j] را false اش می‌کنیم اگرچه lock هنوز برابر true است ولی پراسس j از while عبور می‌کند و وارد cs خودش می‌شود.

پس کلیدکان lock توسط پراسس i گرفته شده بود و پراسس i، lock را تحویل پراسس j نداد ولی به واسطه false کردن waiting[j] lock تحویل پراسس j داده شد و بنابراین با این کاری که کردیم پراسس j توانست از ناحیه ورودش عبور کند و وارد cs خودش بشود.

حالا پراسس j هم وقتی از cs اش خارج می‌شود از شماره بعد از پراسس خودش دوباره جستجو برای پیدا کردن فرآیند بعد از خودش که waiting ای برابر true داشته باشد شروع می‌کند

- همان طور که مشخص است mutual exclusion برقرار است و هیچ ۲ پراسس با هم نمی‌توانند وارد ناحیه بحرانی شوند و فقط یک پراسس می‌تواند وارد ناحیه بحرانی بشود و بقیه در ناحیه ورودگید می‌کنند، شرط پیشرفت هم برقرار است چون وقتی یک پراسس در ناحیه باقیمانده خودش است lock را false می‌کند و همین طور از بین پراسس ها هیچ وقت این اتفاق نمی‌افتد که همه پراسس ها پشت حلقه while شون گیر کنند پس پراسس ها دچار انسداد نامحدود نمی‌شوند

چطور در این راه حل Bounded waiting برقرار شده؟ یک پراسس که داره ناحیه بحرانی اش را ترک می‌کند پراسس ها را به ترتیب شماره پراسس های بعد از خودش چک می‌کند و اگر یک پراسس (i) بخواهد وارد ناحیه بحرانی شود حداکثر دفعاتی که مشاهده می‌کند که رقبا وارد ناحیه بحرانی شون می‌شود بدین صورت خواهد بود که مشاهده می‌کند که هر کدام از رقبای بعد از خودش یکبار وارد ناحیه بحرانی اش می‌شود و بعد از خروج از cs اش نوبت را به پراسس بعد از خودش بدهد و به همین ترتیب، یعنی حداکثر اینطوری میشه: i ← i+1 ← i+2 ← ...

و ماچون داریم mod n می‌گیریم بالاخره دور می‌زنیم و بعد از اینکه n-1 بار سایر فرآیندها وارد ناحیه بحرانی شوند دیگه حتما نوبت به ما می‌رسد البته توجه کنید که این تعداد حداکثر انتظار یک فرآیند است و ممکن است فرآیندهای بعدی (n-1 فرآیند) لزوما نخواهند وارد ناحیه بحرانی شون بشوند و زودتر نوبت به ما برسد ولی کران بالا همان n-1 است. (پس از n-1 بار ورود رقبا به ناحیه بحرانی نوبت به پراسس مذکور می‌رسد). بنابراین خاصیت آخر هم در این راه حل وجود دارد.

- اما این راه حل هنوز مشکل Busy waiting را دارد، یعنی مدت زمانیکه یک پراسس نوبت اش نشده باشد را داره توی حلقه while سپری می‌کند و بنابراین زمان cpu در این تایم ها هدر می‌رود.

Mutex Locks

در زبان های برنامه نویسی یکسری دستورات و امکانات در اختیار برنامه نویسان قرار می‌گیرد که درگیر جزئیات راه حل ناحیه ورود و ناحیه خروج نشوند، بعنوان مثال وقتی یک برنامه نویس می‌خواهد وارد cs بشود می‌تواند از دستور acquire lock استفاده کند که این دستور یک قفل را می‌گیرد که cs را نسبت به رقبا قفل می‌کند و بعد که برنامه نویس می‌خواهد ناحیه بحرانی را ترک کند با دستور release lock قفل را آزاد می‌کند.

متغیر از جنس mutex: mutex مخفف Mutual exclusion است که یک نوع lock است و ما بعنوان برنامه نویس اگر بخواهیم از یک cs محافظت کنیم می‌توانیم یک متغیر از جنس lock تعریف کنیم و پس قبل از ورود به cs با دستور acquire lock قفل را ببندیم و بعد از خروج از cs قفل را با release lock باز کنیم. به اینصورت کاری کردیم که رقبا نسبت به هم محافظت بشوند.

```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
```

```
release() {
    available = true;
}
```

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

semaphore

یکی دیگر از متغیرهایی که با همان هدف بالا استفاده می‌شود متغیر سمافور و دستورات مربوط به آن است.

سمافور یک نوع متغیر است که در برخی از زبانهای برنامه نویسی ازش استفاده میشه و در ساده‌ترین حالت یک متغیر integer است ولی اگر فقط این حالت ساده‌اش را در نظر بگیریم مشکلاتی در تعریف کردن سمافور به وجود می‌آید، بنابراین بعدا مجبور هستیم که یک نسخه پیچیده‌تری از سمافور را تعریف بکنیم

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

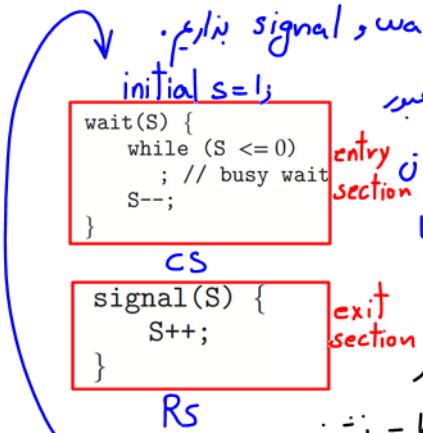
```
signal(S) {
    S++;
}
```

روی متغیر از نوع سمافور ۲ عمل می‌توان انجام داد ۱- wait ۲- signal

۱- wait: چک می‌کند که اگر مقدار سمافور کوچکتر مساوی ۰ (S≤۰) است، گیر می‌کند و روی همان خط می‌ایستد، البته توجه کنید که چون دستور wait اتمیک است اجرا دستور wait باعث decrement شدن S می‌شود

۲- signal: S را increment می‌کند (S++)

بنابراین برای اینکه ما بخواهیم مشکل (مسئله) ناحیه بحرانی را حل کنیم کافی است cs را بین دستورات wait و signal بذاریم.



در واقع در یک حلقه while for ever، اگر مقدار اولیه S را یک بگذاریم آن گاه با دستور wait(s) از while عبور می‌کنیم و بعد S== می‌شود و پراسس وارد ناحیه بحرانی می‌شود، حال پراسس‌های دیگر که بخواهند از ناحیه ورودشان عبور کنند چون S برایشان صفر است در حلقه while شون گیر می‌کنند و نمی‌توانند وارد ناحیه بحرانی شوند، تا زمانیکه پراسس که وارد ناحیه بحرانی خودش بوده از آن خارج شود و S را ۱ کند

- اگر دستور wait, signal اتمیک در نظر گرفته نشوند، خب در کد wait متغیر S دارد کم می‌شود و در دستور signal دارد زیاد می‌شود که در این صورت اگر این ۲ دستور اتمیک انجام نشوند، یعنی اگر نسبت به هم حفاظت نشوند صحت مقدار سمافور تحت شعاع قرار می‌گیرد و بنابراین شروطی که برای درست بودن یک راه حل ناحیه بحرانی باید برقرار باشد، برقرار نخواهد بود (اگر مقدار سمافور نامعتبر باشد نمی‌توانه تضمین کرد که شروطی که " " "

اجرا نمی‌شود، چون P_2 می‌رود پشت wait(synch) گیر می‌کند، چون مقدار اولیه synch=0 است و وقتی در پراسس ۱، دستور S_1 اجرا شد و بعد دستور signal(synch) را اجرا کرد synch=1 می‌شود و بنابراین پراسس ۲ می‌تواند از دستور wait عبور کند. پس ما اومدیم این ۲ پراسس را با هم سنکرون کردیم، به این معنا که کاری کردیم که پراسس ۲ نسبت به پراسس ۱ جلو نیفتد.

```
P1                  P2
S1;                 wait(synch);
signal(synch);      S2;
```

Binary semaphor : مقدار سمافور می‌تواند ۰ و ۱ باشد

counting " : " " " " هرعددی را بگیرند، مثلاً اگر شما مقدار اولیه سمافور را ۱۰ بدارید به این معنا است که ۱۰ تا پراسس می‌توانند با هم وارد CS بشوند. خب این که تناقض داره با چیزی که ما دنبالش بودیم؟!

بله اگر ما در مسئله‌ای نخواهیم inconsistency بین متغیرها به وجود بیاید باید از همان Binary semaphor استفاده بکنیم ولی این نوع سمافور کاربردهای دیگری می‌تواند داشته باشد.

مثال از کاربردهای counting Binary semaphore:

فرض کنید یک web server داریم که می‌خواهد به‌طور همزمان فقط به ۱۰۰۰ تا کلاینت سرویس بدهد، در این حالت می‌توانیم از یک counting semaphor با مقدار اولیه ۱۰۰۰ استفاده کنیم و هر وقت یک کلاینت جدید به این سرور درخواست میدهد، سرور می‌رود در ناحیه‌ای که به آن کلاینت سرویس می‌دهد و یکی از عدد سمافور کم می‌شود و حالا اگر ۱۰۰۰ کلاینت همزمان در حال سرویس گرفتن باشند مقدار سمافور ۰ می‌شود و سرور به بقیه کلاینت‌هایی که می‌خواهند به سرور وصل بشوند پیام مناسب را نشان می‌دهد و این کار باعث می‌شود که لود سرور شما از یک تعداد همزمان بالاتر نرود.

A Better Definition for semaphore

برای حل مشکلات spin-lock و Bounded-waiting می‌آییم و تعریف سمافور را از یک متغیر ساده integer به یک Data structur تبدیل می‌کنیم که یکی از متغیرهایش همان متغیر integer ساده است و یک فیلد دیگرش یک پوینتر هست از Data structure ای از نوع PCB

همچنین تعریف ۲ عمل wait و signal به شکل زیر تغییر می‌دهیم.

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            sleep();
        }
}
```

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

بطوریکه ذات Decrement کردن و wait، ذات increment کردن و signal سرجای خودش باقی بماند.

wait: اگر بعد از کم کردن در wait، value (همان متغیر integer) کوچکتر از ۰ شد بدان معنا است که قبلا پراسس از سمافور استفاده کرده و وارد CS شده و بنابراین این پراسس دیگه نباید موفق بشود که وارد CS بشود، در این شرایط pcb این پراسس از صف ready خارج می‌شود و توی لیست این سمافور قرار می‌گیرد، به معنا اینکه این پراسس روی این سمافور در حالت wait قرار می‌گیرد و بعد سیستم کال Block فراخوانی می‌شود به این معنا که این پراسس به حالت بلوکه در میاد و از حالت Ready خارج شده است.

signal: اگر بعد از اضافه کردن value ـ سمافور ما متوجه شدیم value کماکان کوچکتر مساوی ۰ است، یعنی یا نشده، به این معناست که هنوز بعد از اضافه کردن value پراسس در صف منتظر بوده، بنابراین آن پراسس را از صف آن سمافور خارج می‌کنیم و سیستم کال wakeup را روی آن پراسس اجرا می‌کنیم و به این ترتیب آن پراسس به حالت ready برمی‌گردد و جز پراسس‌هایی خواهد بود که زمانبند توی تخصیص cpu روی آن‌ها هم تصمیم می‌گیرد.

اگر value ـ سمافور ۱۰ - باشد به چه معناست؟ یعنی ۱۰ تا پراسس در شرایط Blocking به سر می‌برند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

دیگه spin lock نداریم: چون پراسی که متوجه می‌شود که موفق نمی‌شود فعلا سمافور را در اختیار بگیرد و وارد cs بشود بلوکه می‌شود و بنابراین کرانتوی که بخش اختصاص داده شده را صرف پردازش بیهوده نمی‌کند

دیگه Bounded waiting نداریم: چون اگر پراسی رفت و wait کرد روی یک سمافور و بلوکه شد، این پراسس‌ها ان این مدلی با یک ترتیبی توی صف آن سمافور قرار می‌گیرند، مثلاً هر کدام دیرتر برسند می‌روند آخر صف قرار می‌گیرند و وقتیکه ما در هنگام signal داریم یک پراسی را wakeup می‌کنیم از سر آن صف برمی داریم پس طبق FcFs پراسس‌ها به ترتیبی که بلوکه شدند ready می‌شوند، پس امکان ندارد که یک پراسی که دیرتر بلوکه می‌شود بتواند جلوی یک پراسی زودتر بلوکه شده قرار بگیرد و به این ترتیب حداکثر تعداد دفعاتی که یک پراسی میبیند که پراسس‌های رقیب وارد ناحیه بحرانی می‌شوند به تعداد پراسس‌هایی خواهد بود که قبل از خودش بلوکه شدند و بنابراین کران وجود دارد.

* کماکان نرفتمان این است که هم wait و هم signal اتمیک هستند

چطور می‌شود درون عمل اتمیک wait و signal این تعداد دستور داشته باشیم و حتی داخل شون syscall هم داشته باشیم

به جای اینکه بگیم wait یک دستور اتمیک است می‌گیم wait روی سمافور s نسبت به wait(s) یا signal(s) پراسس‌های رقیب اتمیک است، یعنی دستورات wait(s) و signal(s) با دستورات wait(s) و signal(s) پراسس‌های رقیب اجازه ندارند لابه‌لای هم اجرا بشوند، ولی ایراد ندارد اگر دستورات wait(s) و signal(s) با دستورات wait(x) یا signal(x) که x یک سمافور دیگر است لابه‌لای هم اجرا بشوند

پس یکی از راه حل‌ها این است که ما خود wait و signal را یک cs در نظر بگیریم که اگر یکی از پراسس‌ها وارد عمل wait روی s شد

entry section

wait cs

exit section

wait(s) و signal(s) برای پراسس‌های رقیب cs ان باشد که آنها نتوانند وارد آن شوند.

پس باید بدنه wait(s) را توسط یک entry section و یک exit section حفاظت اش بکنیم که اینجا کافی است

از یکی از راه حل‌های ناحیه بحرانی استفاده کنیم، مشخص است که اگر بخواهیم درباره از سمافور استفاده کنیم با مشکل مواجه می‌شویم چون بصورت بازگشتی همین داستان تکرار می‌شود، چون باز باید wait و signal های آن را هم اتمیک کنیم و این داستان ادامه دارد.

ولی ما راه حل‌های درست دیگری برای n فرآیند داشتیم که از جنس سمافور هم نبودند و می‌توان از آنها برای entry section و exit section و wait و signal استفاده کنیم ولی آنها شکل busy waiting داشتند ولی سایر شروط را داشتند.

اما چون cs ما در اینجا دستورات wait و signal هست، این بدنه، بدنه بسیار کوچکی است و اگر دچار busy waiting شویم اولا کوتاه خواهد بود، دوما اصلا احتمال رخداد این موضوع خیلی پایین است که یک پراسی مثلاً وسط wait(s) باشد و کوانتوم اش تمام شود و بعد به پراسی دیگری داده شود که آن هم اتفاقا بخواهد وارد wait یا signal اش شود و نه دستورات دیگرش و درنتیجه busy waiting پیش بیاید ولی تازه بعد از این اگر نوبت به پراسی اول رسید که وسط این cs اش بود (وسط اجرا دستورات wait یا signal اش) کوانتوم اش تمام شده بود دیگه قطعا می‌تواند اگر چند ده دستوری که از ش باقیمانده را تمام کند و قطعا دیگر برای بار دوم این اتفاق نخواهد افتاد، یعنی دفعه بعد که نوبت به پراسی رقیب برسد حتما پراسی اول wait یا signal اش را ترک کرده و پراسی رقیب می‌تواند از entry section خودش عبور کند و وارد عمل wait(s) یا signal(s) شود. پس wait و signal را اتمیک نمی‌کنیم ولی کاری می‌کنیم که نسبت به رقیب خودشون اتمیک محسوب بشوند

راه حل ۲: ابتدا wait و signal و وقفه‌ها را غیرفعال کنیم و انتهای آنها وقفه‌ها را فعال کنیم، که این ایده خیلی خوب نیست چون وقتی شما وقفه‌ها را غیرفعال می‌کنید در واقع دارید برای یک مدت زمانی به سیستم اجازه نمی‌دید که به اتفاقات محیطی عکس العمل نشان بدهد، که البته این کار می‌تواند برای سیستم عوارض داشته باشد به خصوص اگر این سیستم Real time باشد

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

استفاده نادرست از سمافور چه عوارضی می‌تواند ایجاد کند؟

در زیر دو تا پراسس می‌بینید که هر دوشون به سمافورهای S و Q نیاز دارند ولی به ترتیب متفاوتی روی S و Q تلاش کردند، در اینجا اگر P_0، S را بدست بیاورد و بعد کوانتوم اش تمام بشود و بعد کوانتوم به P_1 برسه و P_1، Q را بدست آورد آن‌گاه P_1 روی خط wait(S) گیر می‌کند و بعد که نوبت به P_0 برسه آن‌گاه P_0 روی خط wait(Q) گیر می‌کند که به این ترتیب با بن بست مواجه می‌شویم، چون هرکدام از این ۲ پراسس یک رسورس را گرفته و منتظر رسورس است که دست رقیب اش است و این شرایط هیچ وقت شکسته نمی‌شود مگر اینکه یکی از این پراسس‌ها را ما terminate کنیم

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

قبلا گفته بودم که بن بست با starvation متفاوت است، چون starvation همان‌طور که شانس به وجود می‌آید می‌تواند شانس هم از بین برود ولی بن بست اگر رخ بدهد دیگر از بین نمی‌رود مگر اینکه یک عامل خارجی مثل OS بیاد و بن بست را بشکنه، مثلاً با terminate کردن یکی از پراسس‌ها

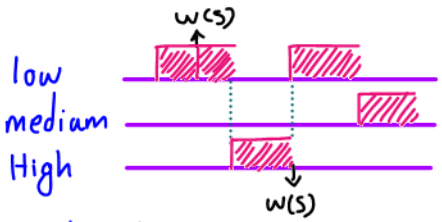
- یک مشکل دیگر می‌تواند به اینصورت باشد:

```
signal(mutex);
   ...
   critical section
   ...
wait(mutex);
```

برنامه نویس به اشتباه ابتدا signal و بعد wait گذاشته باشد و به این ترتیب نقض mutual exclusion خواهیم داشت

یا اینکه اشتباه به جای signal، wait بگذارد و به این ترتیب این پراسس با deadlock بن بست مواجه می‌شود

```
wait(mutex);
   ...
   critical section
   ...
wait(mutex);
```



وارونگی اولویت : Priority Inversion

فرض کنید در الگوریتم‌های مبتنی بر اولویت ما سه سطح High و medium و low داشته باشیم و پراسس low اجرایش را زودتر شروع کند و در یک مقطعی از زمان روی سمافور S، wait بکند و به اجرایش با CS ادامه بدهد و بعد از کمی اجرا فرآیند High از راه برسد، بنابراین preemption رخ می‌دهد و CPU به پراسس High داده می‌شود، حالا اگر بعد از یک مدت پراسس High بخواهد روی سمافور S، wait کند، در این حالت پراسس High بلاک می‌شود و پراسس low که سمافور در اختیارش هست ادامه می‌دهد، به این پدیده وارونگی اولویت می‌گویند که الان چون رسورس در اختیار پراسس low priority بود اجرای آن به پراسس با اولویت High مقدم شمرده شده، حالا اگر قبل از اینکه پراسس low، سمافور را رها کند یک پراسس medium که اصلا کاری هم با سمافور نداره وارد بشه پراسس low را preempt می‌کند و به این ترتیب زمان اجرا پراسس medium هم مقدم بر پراسس High اجرا می‌شود که به این پدیده uncontrol priority inversion می‌گویند و می‌تواند زمان Blocking پراسس High را طولانی کند، حالا یکی از ایده‌هایی که برای مقابله با UPI وجود دارد priority-Inheritance protocol یا پروتکل ارث بری اولویت است به این شکل که اگر پراسس High روی پراسس low دچار Blocking شد، پراسس low موقتا priority پراسس High را به ارث می‌برد و در سطح اولویت پراسس High اجرایش را ادامه می‌دهد به این ترتیب پراسس‌های medium اگر در حین اجرا low وارد شوند دیگر نمی‌توانند اجرا پراسس را preempt کنند، در اینصورت اگرچه هنوز priority inversion داریم ولی uncontrol نیستند و پراسس High حداکثر به اندازه critical section پراسس low معطل می‌شود و وقتی پراسس low، سمافورش را signal کند اولویت اش به سطح اصلی خودش (low) برگردانده می‌شود و در اینصورت پراسس High که اولویت اش بیشتر از medium هست با گرفتن سمافور از پراسس low به اجرا خودش ادامه می‌دهد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

دانستنی: این اتفاق در سال 1997 برای Mars path finder وقتی روی سطح مریخ فرود میاد رخ می دهد و باعث می شود فرآیند با اولویت بالا یک مدت طولانی وکانش نشون ندهد و watch dog timer سیستم را reset می کند. این مشکل مکررا رخ میده تا مسئولان ایستگاه زمینی مشکل راکشف می کنند و امکان PIP را در OS این مریخ نورد فعال می کنند و مشکل حل می شود

watch dog timer = computer operating properly: سیستم کامپیوتری درصورتی که درست کار نکنه بصورت پریودیک این را ریست می کند، اگر کامپیوتر هنگ کند یا خیلی کند کار کند این تایمر ریست نمیشه و تا انتهایی که برنامش تعیین شده برود سیستم را ریست می کنه ← فضاپیما، ربات های جستجوگر دیگر سیارات

producer-consumer (Bounded Buffer)

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

producer

```
while (true) {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);
   wait(mutex);
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
}
```

consumer

```
while (true) {
   wait(full);
   wait(mutex);
    . . .
   /* remove an item from buffer to next_consumed */
    . . .
   signal(mutex);
   signal(empty);
    . . .
   /* consume the item in next_consumed */
    . . .
}
```

مقدار اولیه empty را برابر ظرفیت بافر یعنی N درنظر می گیریم و مقدار اولیه full را ۰ درنظر می گیریم. برای دسترسی انحصاری به بافر از سمافور mutex استفاده شده است.

producer: پس از اینکه تولیدکننده یک آیتم تولید می کند و بعد ابتدا روی سمافور empty، wait می کند و پس روی سمافور mutex، wait می کند و بعد از اینکه آیتمی که تولید کرده را به بافر که یک رسورس مشترک است اضافه کرد ابتدا روی سمافور mutex و پس روی سمافور سومی به اسم full، signal می کند. چون مقدار اولیه empty=N است تولیدکننده می تواند تا N مرتبه آیتم تولید و به بافر اضافه کند، حالا هر وقت تولیدکننده یک آیتم به بافر اضافه می کند با اجرا signal روی full، به full یکی زیاد می شود

مشخص است که به بافر بیشتر از N آیتم اضافه نمی شود و تولیدکننده اگر بخواهد N+1 امین آیتم را تولید کند روی سمافور empty بلوکه خواهد شد تا وقتی که مصرف کننده empty را حداقل یکبار signal کند که حداقل یک جای خالی در بافر به وجود بیاد

consumer: تاوقتیکه full برابر ۰ باشد مصرف کننده بلوکه خواهد بود، حالا اگر تولیدکننده چیزی تولید کرده باشد به واسطه اینکه بعد از اضافه کردن آیتم full را زیاد می کند، که تولیدکننده این کار را با signal کردن full انجام می دهد، حالا اگر تولیدکننده حداقل یک آیتم تولید کند، در این شرایط مصرف کننده می تواند از wait روی full عبور کند و بعد با توجه به دستور wait(mutex) و signal(mutex) دسترسی انحصاری به بافر خواهد داشت و بعد که یک آیتم برداشت میاد و با اجرا دستور signal(empty) یکی به تعداد خونه های خالی بافر اضافه می کند

یادآوری:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            sleep();
        }
}
```

OS این فرآیند را روی صف سمافوری می خواباند

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

OS یک فرآیند را از صف سمافوری برمیدارد و ready اش می کند.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

Readers_writers problem

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

writer

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

Reader

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

اگر یک فایل مشترک وجود داشته باشد تعدادی reader می‌توانند بصورت مشترک از این فایل مشترک بخوانند ولی در طول زمانی که حداقل یک reader در حال خواندن این فایل باشد هیچ writer‌ای نمی‌تواند روی فایل بنویسد و همین بالعکس، یعنی اگر یک writer در حال نوشتن روی این فایل باشد هیچ reader‌ای نمی‌تواند بخواند و البته هیچ writer دیگری هم نمی‌تواند در آن فایل بنویسد

در تا سمافور mutex و rw_mutex با مقدار اولیه ۱ در نظر گرفته شده و یک متغیر read_count با مقدار اولیه ۰ که بین کل writer‌ها و reader‌ها shared است، تعداد reader‌هایی که دارند بصورت همزمان از فایل می‌خوانند را نشان می‌دهد.

writer: رایتر قبل از اینکه بخواهد اقدام به نوشتن در فایل کند روی سمافور rw_mutex، wait می‌کند و بعد از اینکه نوشتن اش روی فایل تمام شد روی rw_mutex، signal می‌کند، که چون مقدار اولیه rw_mutex برابر ۱ است، اولین نفری که writer است می‌تواند روی فایل write کند و writer‌های دیگری اگر بخواهند در این حین در این فایل بنویسند باید صبر کنند تا عمل نوشتن این writer تمام بشود

reader: از سمافور mutex بدین منظور استفاده کرده که دسترسی به متغیر read_count را انحصاری کند که همان رخدادی‌یی که برای متغیر count در اولین مثال این فصل در دوم اتفاق می‌افتاد برای متغیر read_count نیفتد، چون متغیر read_count ممکن است توسط reader‌های مختلفی increment یا decrement بشود، در قسمت پایین کد reader هم مجددا به همین دلیل دسترسی به متغیر read_count را با استفاده از سمافور mutex به cs تبدیل کرده است.

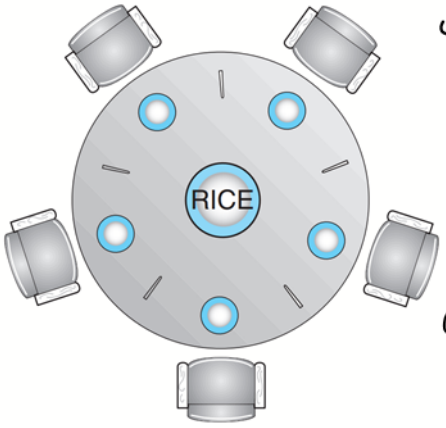
حالا اگر یک reader بتواند از wait(mutex) عبور کند read_count را یکی زیاد می‌کند، یعنی به تعداد reader‌ها یکی اضافه می‌کند (خودش را اضافه می‌کند) و اگر بار اولی باشد که یک reader می‌خواهد از فایل بخواند (می‌خواهد وارد شود) یعنی read_count=۱ شده باشد، روی سمافور rw_mutex، wait می‌کند، بنابراین اگر قبلا نویسنده‌ای در حال نوشتن باشد باعث می‌شود reader در این خط بلوکه بشود و بنابراین این reader نمی‌تواند عمل خواندن از فایل را شروع کند و reader‌های بعد از این reader (reader‌های دوم به بعد) که دیگر اصلا نمی‌توانند از wait(mutex) خط اول عبور کنند و آنها روی این خط (wait(mutex)) بلوکه می‌شوند، بنابراین در این شرایط هیچ reader‌ای نمی‌تواند اقدام به خواندن بکند، اما اگر وقتی یک reader به خط wait(rw_mutex) می‌رسد هیچ نویسنده‌ای در حال نوشتن نباشد در اثر اجرای wait(rw_mutex) مقدار rw_mutex از ۱ به ۰ تغییر می‌کند و reader وارد عمل خواندن از فایل می‌شود، در این لحظه اگر reader دیگری (دوم) بخواهد از این فایل بخواند، می‌تواند از wait(mutex) عبور کند و read_count=۲ می‌شود و چون read_count دو است دیگه نیازی نیست روی rw_mutex، wait بکند، چون یکبار این اتفاق افتاده و ما می‌دانیم که در این شرایط هیچ writer‌ای فعال نیست (چون اولین reader با اجرا wait(rw_mutex)، writer‌ها را غیرفعال می‌کند)، بنابراین هر تعداد reader‌ای در این شرایط برسند می‌توانند وارد عمل خواندن بشوند. حالا وقتی یک reader خواندن از فایل را ترک می‌کند، از read_count یکی کم می‌شود و تا وقتی که read_count

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

برابر ۰ نشود ما rw_mutex را signal نمی‌کنیم، یعنی تا زمانیکه readerای وجود دارد اجازه نمی‌دهیم که یک writer عمل نوشتن را شروع کند ولی آخرین reader که خواندن از فایل را ترک کرد باعث می‌شود یک signal روی rw_mutex رخ بدهد و در این شرایط یک writer که کوانتوم دست اش میرسه می‌تواند عمل write را انجام بدهد. ← مثلاً readerآخر که در cs خودش است، readerهای دیگری برسند

توجه کنید که اگر قبل از اینکه آخرین reader به خط if(read_count==0) برسد، readerهای جدیدی بیایند و شغل عمل خواندن شوند چون read_count زیاد میشه و بنابراین این شرط ↑ برقرار نمی‌شود و signalای روی rw_mutex رخ نمی‌دهد و writer باید معطل بشوند، در این کد ما داریم به readerهایی که دیرتر از یک writer می‌رسند اولویت می‌دهیم و تا زمانیکه حتی یک reader هم وجود داشته باشد writer باید صبر کنند، البته می‌توان این کد را به گونه‌ای تغییر دهیم که وقتی یک writer می‌رسد readerهایی که بعد از آن می‌آیند نتوانند عمل خواندن از فایل را شروع کنند.

Dining philosophers problem



یک ظرف غذای مشترک وسط یک میز قرار دارد و ۵ فیلسوف دور میز نشستند و بین هر ۲ فیلسوف یک chopstick قرار دارد، هر فیلسوف به ۲ تا chopstick برای غذا خوردن نیاز دارد، بنابراین اگر یک فیلسوف بخواهد شروع کند به خوردن غذا باید ۲ فیلسوف سمت چپ و راست اش به انتظار بشینند چون chopstick یک طرف این ۲ فیلسوف درحال حاضر وجود ندارد بین هر ۲ فرآیند (فیلسوف) ما یک منبع مشترک (chopstick) داریم و یک فرآیند باید همه منابع مشترک با همسایه‌هایش را بدست بیاورد تا بتواند اجرا خودش را شروع کند.

حالا می‌خواهیم با سمافور یک راه حل برای این مسئله ارائه کنیم:

```
semaphore chopstick[5];

while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

راه حل ابتدایی که وجود دارد این هست که بیاییم و به ازای هر chopstick یک سمافور تعریف کنیم پس ۵ سمافور خواهیم داشت.

حالا فیلسوف شماره i وقتی می‌خواهد عمل غذا خوردن را شروع کند روی chopstick سمت چپ و راست خودش باید wait کند حالا اگر جفت این chopstick ها در دسترس بودند این ۲ را بدست می‌آورد و می‌تواند وارد بخش غذا خوردن بشود (غذا خوردن را آغاز کند) و هروقت غذا خوردنش تمام شد باید هر دو chopstick را signal کند و وارد قسمت فکر کردن می‌شود.

اگر یکی یا هردو تا chopstick آماده نبود فیلسوف مربوطه بلاک می‌شود و نمی‌تواند وارد بخش eat اش بشود

این راه حل، راه حل درستی محسوب نمی‌شود، حالتی را در نظر بگیرید که هرکدام از فیلسوف ها بیان و روی chopstick سمت چپ شون wait کنند یعنی فیلسوف اول این کار را بکند، بعد کوانتوم به فیلسوف بعدی برسد و آن هم همین کار را بکند و... بنابراین هر فیلسوف chopstick سمت چپ شون رو در اختیار دارند ولی روی chopstick سمت راست شون بلاک می‌شوند و در نتیجه بن بست رخ میدهد و هیچ کدام از فیلسوف ها نمی‌تواند به غذا خوردن بپردازند.

- مشخص است که در این مسئله هیچ وقت ۲ فیلسوف مجاور نمی‌توانند با هم غذا بخورند

وقتی ۵ تا فیلسوف داریم در یک لحظه یا ۱ نفر یا نهایتا ۲ نفر بصورت همزمان می‌توانند غذا بخورند، اگر ۶ تا باشند حداکثر ۳ نفر

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۳ راه حل برای حل مشکل فیلسوف ها

۱- یک سمافور دیگری به نام mutex تعریف کنیم بطوریکه وقتی یکی از فیلسوف ها در حال برداشتن چوب ها (غذاخوری خودش است)، هیچ فیلسوف دیگری درحال انجام این کار نباشد، اینطوری فیلسوف می تواند دو تا چوب غذاخوری را با هم بردارد و پس mutex را ۱ بکند که نفر بعدی بتواند اقدام به برداشتن چوب های غذاخوری کند، به اینصورت حداقل تنفر اول می تواند اقدام به غذاخوردن بکند و دیگر بن بست به وجود نمی آید

```
wait(mutex);
wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
signal(mutex);
```

۲- یک سمافور با مقدار اولیه ۴ دور کل کد برنامه مون نداریم، اینطوری ما اجازه نمی دیم بیشتر از ۴ فیلسوف دور میز بشینند و بنا به اصل لانه کبوتری حداقل یکی از فیلسوف ها دو تا چوب غذاخوری گیرش میاد و در اینصورت باز هم با مسئله بن بست مواجه نمی شویم.

۳- بیایم و فیلسوف های دور میز را از ۰ تا ۴ شماره گذاری کنیم و همین طور چوب های غذاخوری را هم شماره گذاری کنیم و سپس برنامه را به اینصورت بنویسیم که فیلسوف های شماره زوج باید ابتدا چوب غذاخوری سمت چپ شون رو بردارند و بعد چوب سمت راست شون و فیلسوف های شماره فرد اول چوب سمت راست شون رو بردارند و بعد چوب سمت چپ شون رو، به این ترتیب از بین ۲ همسایه که یکی شون فرد و یکی شون زوج است حتما یکی شون موفق می شود که شروع به غذا خوردن کند چون اگر رقابت به چوب وسط بین یک شماره فرد و یک شماره زوج کشیده شود به این معناست که جفت شون حتما چوب سمت دیگرشون رو موفق شدند بردارند، بنابراین در نهایت یکی از این ۲ موفق می شود چوب غذاخوری ای که بین شون هست را بردارد و شروع به غذاخوردن بکند و باز هم مسئله بن بست حل می شود

Monitors

می خواهیم ابزاری را برای طراحی گیت ورود و گیت خروج طراحی کنیم که جایگزین سمافور بشود

یک ابزار دیگر برای هماهنگ سازی است و ابزار سطح بالاتری نسبت به سمافور است، همون طور که گفتیم کدزدن با سمافور سخت و اگر برنامه نویس حواسش نباشد ممکن است خطاهایی در برنامه مارخ دهد، مثل نقض شدن mutual exclusion و یا بن بست.

برای اینکه جزئیات را از دست برنامه نویسان خارج کنیم و ابزار مانیتور طراحی شده که یک ابزار در سطح زبان برنامه نویسی محسوب می شود این ابزار را در زبان های برنامه نویسی تولید می کنند و کامپایلر کارها را انجام می دهد و سطح بالاتر از سمافور که یک راه حل سیستم عاملی بود و wait و signal فراخوان های سیستمی بودند (مانیتور درون برنامه تعریف میشه و OS نمی داند درون برنامه ها چیست، مانیتور یک ماژول برنامه است و در وسط برنامه ها تعریف می شود و کامپایلر است که این ماژول را می خواند و آن را به زبان ماشین تبدیل می کند)

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
    . . .
  }

  function P2 ( . . . ) {
    . . .
  }

      .
      .
      .

  function Pn ( . . . ) {
    . . .
  }

  initialization_code ( . . . ) {
    . . .
  }
}
```

در رویه روساختار کلی مانیتور را می بینید، یک مانیتور شبیه به کلاس است که یکسری متغیر دارد که ابتدای کار مقداردهی اولیه می شوند و تعدادی function دارد

متغیرها ۲ نوع هستند: ۱) متغیرهای محلی ۲) متغیرهای شرطی

این functionها ۲ دسته هستند: ۱) توابعی که از بیرون قابل فراخوانی هستند

۲) " " " " رویت نیستند و صرفا کاربرد داخلی دارند

نکات:

۱) از متغیرهای شرطی فقط و فقط درون مانیتور استفاده می شود و کارکردشون بلوکه کردن و بیدارکردن فرآیندها است و فرآیندها میرن روی متغیرهای شرطی خوابن

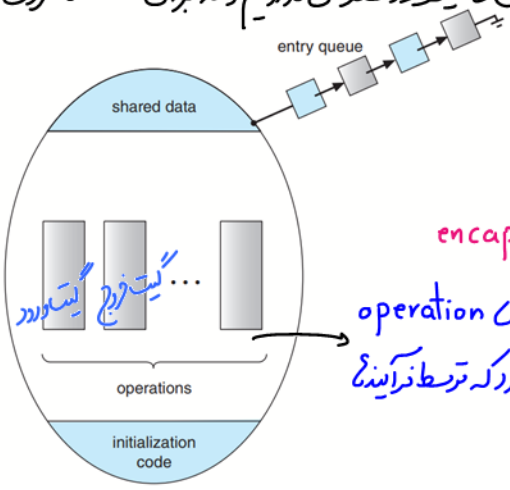
۲) در داخل مانیتور اگر می خواهیم یک فرآیند را بلوکه کنیم یا بیدار کنیم فقط و فقط از طریق دستورات signal و wait روی متغیرهای شرطی که

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

درون مانیتور وجود دارند می‌توانیم این کار را بکنیم.

داخل مانیتور به هیچ روش دیگری مثل wait و signal روی سمافور نمی‌توانیم فرآیندی را بلاک کنیم یا بیدار کنیم.

- توجه کنید که متغیرهای شرطی درون مانیتور تعریف می‌شوند و به متغیرهای داخل مانیتور از بیرون مانیتور دسترسی نداریم و نه برای wait کردن روی متغیرهای شرطی و نه برای signal زدن روی آنها نمی‌توانیم از بیرون مانیتور کاری کنیم



مانیتور ۲ تا ویژگی کلیدی دارد:

۱) تنها راه دسترسی به متغیرهای درون مانیتور (چه خواندن و چه نوشتن) فراخوانی توابع درون یک مانیتور است و هیچ فرآیندی از بیرون مانیتور به متغیرها دسترسی ندارد

encapsulation :

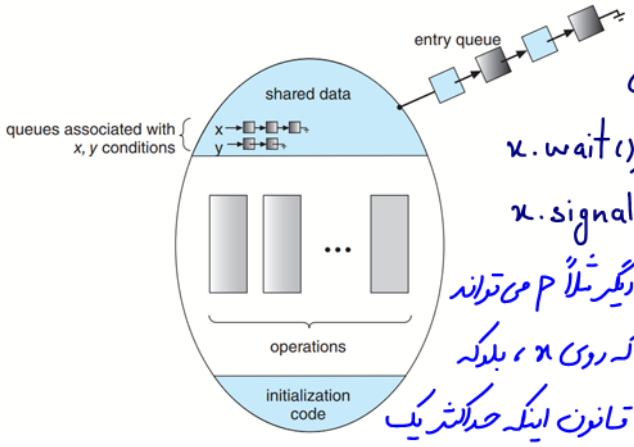
یک مانیتور تعدادی operation یا function دارد که توسط فرآیندها اجرا می‌شوند

۲) در هر لحظه فقط یک فرآیند می‌تواند درون مانیتور فعال باشد

* مورد ۱ و ۲ در کنار هم باعث نمی‌شود race پیش نیاد

فرآیندهای بیرون مانیتور صف می‌بندند و یکی یکی می‌توانند وارد مانیتور بشوند، هر وقت یک فرآیند درون مانیتور بلاک شده بود، به یک فرآیند دیگر اجازه ورود به مانیتور داده می‌شود، فقط توجه کنید که اگر چندین فرآیند درون مانیتور باشند در هر لحظه فقط ۱ فرآیند می‌تواند درون مانیتور فعال باشد (پس اگر یک فرآیند توی صف مانیتور بود و از گیت ورود وارد شد و توی مانیتور فعال بود، دیگه به فرآیند دیگری اجازه عبور از گیت ورود داده نمی‌شود

- یک فرآیند می‌تواند ۲ عمل wait و signal را روی condition variable انجام بدهد، توجه کنید که شباهت اسمی شون با wait و signal روی سمافور باعث برداشت اشتباه نشود، این متغیرها که روشون wait و signal می‌زنیم از جنس condition هستند نه از نوع سمافور. wait روی یک condition یعنی برو توی صف اون condition بلاک شو و signal روی condition یعنی یکی از فرآیندهایی که روی آن condition بلاک شده آزاد بشود. پس اگر signal روی x.wait(); انجام بشود ولی هیچ کسی روی x معطل نباشد این یک عمل بی اثر است.



x.signal();

اگر فرآیند p درون مانیتور باشد و x.wait را فراخوانی کند بلاک می‌شود و یک فرآیند دیگر مثلاً p می‌تواند وارد مانیتور شود، حالا اگر فرآیند p، x.signal را فراخوانی کرد، در این شرایط فرآیند p که روی x، بلاک شده بود آزاد می‌شود، حالا ۲ تا فرآیند فعال در مانیتور وجود دارد، حالا چیکار کنیم که قانون اینکه حداکثر یک فرآیند فعال باید در مانیتور وجود داشته باشد حفظ بشود. یکی از ۲ سناریو زیر باید در داخل مانیتور فعال بشود که اصل اصلی ما رعایت بشود

۱- signal and wait : فرآیندی که signal کرده (p) خودش بلاک بشود تا زمانیکه یا q مانیتور را ترک بکند که در اینصورت q توی مانیتور فعال نیست و p می‌تواند ادامه بدهد یا اینکه بره روی یک condition دیگه بلاک بشه (مثلاً k)

۲- signal and continue : فرآیندی که signal کرده (p) دیگه توی مانیتور فعال نباشد، یعنی تا زمانیکه p مانیتور را ترک کند یا اینکه یک جای دیگری توی مانیتور روی یک condition دیگه بلاک بشود، در این شرایط q که آزاد شده بود و موقتا یک جای دیگه بلاک شده بود می‌تواند به کار خودش ادامه بدهد

نکته) برای اینکه اصطلاحات wait و signal در سمافور با مانیتور قاطی نشه در برخی از کتب مثل تنباوم از Down و up به جای wait و signal استفاده می‌شود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
    int item;
    while (TRUE) {                              /* repeat forever */
        item = produce_item();                  /* generate next item */
        if (count == N) sleep();                /* if buffer is full, go to sleep */
        insert_item(item);                      /* put item in buffer */
        count = count + 1;                      /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);       /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {                              /* repeat forever */
        if (count == 0) sleep();                /* if buffer is empty, got to sleep */
        item = remove_item();                   /* take item out of buffer */
        count = count - 1;                      /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer);   /* was buffer full? */
        consume_item(item);                     /* print item */
    }
}
```

① فرض کنید در ابتدای کار هستیم و بنابراین count=0 است و ابتدا مصرف کننده اجرا میشه و تا وسط خط ۳ میره و قبل sleep وقفه میاد و cpu به تولید کننده می رسد، در این حالت مصرف کننده هنوز ready است، حالا مصرف کننده شرط خط ۳ برایش false میشه و بعد آیتم اول رو میذاره توی بافر و کانت رو ۱ میکنه و یک سیگنال wakeup برای مصرف کننده میده ولی چون مصرف کننده نخوابیده بوده این سیگنال هدر میره و بعد میره بالا و ۹ دور یگه میزنه و ۱۰ تا آیتم میذاره توی بافر و کوانتوم اش تموم میشه و بعد که نوبت مصرف کننده میشه sleep رو اجرا میکنه و به خواب ابدی میره، چون بعد اهر وقت بریم توی تولید کننده هیچ وقت کانت ۱ نمیشه، چون تولید کننده قبلا مسئولیت اش که بیدار کردن مصرف کننده بوده رو انجام داده و سیگنال wakeup رو برای مصرف کننده فرستاده ولی سیگنال هدر رفته حالا تولید کننده هم اینقدر آیتم تولید میکنه که کانت برابر N بشه و اون هم به خواب ابدی میره

② فرض کنید که بافر پره و count=N و تولیدکننده داره اجرا میشه و در ① وقبل از اجرای فرمان sleep کوانتوم اش تمام می شود و cpu به مصرف کننده می رسد و می رسه به خط ۵ و کانت N-1 می شه و بعد توی خط ۶ یک سیگنال wakeup برای تولید کننده می فرسته ولی چون تولید کننده نخوابیده این سیگنال هدر میرود حالا مصرف کننده ۹ دور دیگر می چرخد و ۱۰ تا آیتم از توی بافر برمی دارد تا کوانتوم اش تمام بشود و بعد کوانتوم به تولید کننده می رسد و تولید کننده به خواب ابدی می رود حالا بالاخره کوانتوم تولید کننده تموم میشه و cpu به مصرف کننده می رسد و مصرف کننده هم آنقدر دور می زند تا در نهایت بافر خالی می شود و مصرف کننده هم به خواب ابدی می رود

شکل ۳) count=count+1 و count=count-1 خوردش مشکل ایجاد می کند و سر کانترش race condition دارد.

راه حل مانیتور:

۱) هرچی تغییر مشترک مثل count و بافر بذارید توی مانیتور (توی مانیتور تعریف کنید)

۲) توابعی بجز این رو به تابع تبدیل کنید و بذارید توی مانیتور هرجا با count کار داری (می خونی یا می نویسی) به تابع تبدیل اش کنید بذارید توی مانیتور

۳) هرجا می خواید کسی رو بلوک کنید یا بیدار کنید (هرجا sleep یا wakeup داری) اونها رو بردارید و به جاش توی مانیتور condition بذارید و داخل مانیتور wait و signal بزنید روی تغییرهای condition

wakeup و sleep سنوع. چرا؟ استفاده از wakeup و sleep در سر بن بست دارد، پس بیاید از wakeup و sleep استفاده نکنید و بیدار کردن و خوابیدن ها را روی تغییرهای شرطی داخل مانیتور انجام بدهید

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

حالا سناریو ۱ قبلی رو اجرا کنیم:

۱: فرض کنید بافر خالی است، پس count=0 است و اول توی مصرف کننده باشید، می ریم جلو و تابع remove توی مانیتور را صدا می زنیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۲: تابع remove درون مانیتور اجرا می‌شود و بعد از count-- کوانتوم مصرف‌کننده تمام می‌شود

۳: چون مصرف‌کننده هنوز توی مانیتور نخوابیده و فعاله، پس تولیدکننده توی صف مانیتور می‌خوابد (دقت کنید که در هر لحظه فقط ۱ نفر می‌تواند در داخل مانیتور فعال باشد). پس فرستنده نمی‌ره توی مانیتور که سیگنال wakeup بده و سیگنال بخواد هدر برود (signal(empty) در این جا)

۴: در اینجا فرآیند مصرف‌کننده بلوکه می‌شود، توجه کنید که اینجا به خواب ابدی نمی‌رود و در اینجا مثل قبل سیگنال wake up تولیدکننده هدر نمی‌رود که مصرف‌کننده بخواهد به خواب ابدی برود، چرا سیگنال هدر نمی‌رود اینجا؟ چون اصلا ما در اینجا اجازه نمی‌دهیم که تولیدکننده وارد مانیتور بشود که بخواهد signal(empty) را اجرا کند که نتیجه‌اش بخواهد هدر رفتن سیگنال باشد

۵: پس از خوابیدن مصرف‌کننده حالا به تولیدکننده اجازه ورود به مانیتور را می‌دهد، حالا تولیدکننده دستورات اش را تا signal(empty) اجرا می‌کند و با این سیگنال مصرف‌کننده را بیدار می‌کند (مصرف‌کننده از حالت بلاک به ready می‌رود)

حالا آیا ۲ فرآیند فعال در مانیتور مشکل ایجاد نمی‌کند؟

مشکلی وجود ندارد چون دستور signal(empty) در تولیدکننده آخرین دستورش در مانیتور است و بعد از اجرا این دستور تولیدکننده از مانیتور خارج می‌شود

حالا فرض کنید که تولیدکننده ادامه داشت، مشکلی نیست و میایم و یکی از تولیدکننده یا مصرف‌کننده را بلوکه می‌کنیم تا وقتی کار فرآیندی که توی مانیتور است تمام شود و از مانیتور خارج شود و پس فرآیندی را که بلوکه کرده بودیم را بیدار می‌کنیم تا کارش را انجام بدهد و پس از مانیتور خارج شود

Implementation a Monitor using Semaphors

همان‌طور که گفتیم مانیتور یک ابزار سطح زبان برنامه‌نویسی است، بنابراین به راحتی با کامپایلر، نهایتا با سمافور پیاده‌سازی می‌شود.

- مانیتور تعدادی function دارد که بدنه هرکدام از آنها درکد روبه‌رو قرار می‌گیرد. مقدار اولیه mutex برابر ۱ است، بنابراین وقتی یکی از فرآیندها وارد مانیتور می‌شود mutex برابر ۰ می‌شود و فرآیندهای بعدی بیرون مانیتور صف می‌بندند، تا زمانیکه این تابع تمام بشود، وقتی یک تابع تمام می‌شود و مانیتور را ترک کرد mutex را سیگنال می‌کند و بدین ترتیب فرآیند بعدی می‌تواند وارد مانیتور شود و یک تابع را اجرا بکند، حالا اگر در بدنه‌ی F، این فرآیند بگه x.wait این فرآیند بلوکه می‌شود. x یک متغیر از جنس condition است و وقتی تعریف می‌شود به ازایش یک کانتر (x-count) و یک سمافور با مقدار اولیه ۰ تعریف می‌شود حالا وقتی فرآیند می‌گوید x.wait، میایم توی x.wait و x-count یکی زیاد میشه و یعنی تعداد فرآیندهایی که روی این condition منتظر هستند بخشون یکی اضافه می‌شود و بعد mutex سیگنال می‌شود و این یعنی مانیتور آزاد می‌شود و فرآیند بعدی می‌تواند وارد مانیتور بشود و در همین لحظه این فرآیند روی x-sem که متناظر با متغیر condition است متوقف می‌شود، چون مقدار اولیه x-sem برابر ۰ است به محض اینکه این فرآیند mutex را سیگنال می‌کند و اجازه می‌دهد فرآیند بعدی وارد بشود، خودش وقتی wait می‌کند روی x-sem بلوکه می‌شود و می‌رود توی صف این سمافور، حالا فرآیند بعدی که وارد مانیتور شده و داره بدنه‌ی یک تابع دیگر مثل G را اجرا می‌کند اگر بگوید x.signal چه می‌شود؟ با سناریو signal and wait، این فرآیند که سیگنال می‌کند خودش باید wait کند، حالا بریم توی x.signal: اول چک می‌کنیم که condition ای که الان سیگنال شده است آیا اصلا کسی روی خوابیده است یا خیر

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

x.wait() {
```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);        *1
x_count--;
```
}

x.signal() {
```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);      *2
   next_count--;
}
```
}

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

که اگر کسی روی آن معطل نباشد و x_count مساوی ۰ باشد هیچ اتفاقی نمی‌افتد ولی اگر x_count بزرگتر از ۰ بود که در مثال ما نیز اینگونه است، آن گاه این فرآیندی که x.signal زده باید wait کند، چون قرار شد این را یک جایی بلوکه کنیم تا آن فرآیندی که بیدارش کرده یا مانیتور را ترک کند یا روی یک condition دیگری بلوکه بشود.

حالا اونجا کجاست؟ یک condition مخفی و داخلی توی مانیتور وجود دارد به نام Next، پس Next یک کانتر و یک سمافور دارد، پس حالا کانتر مربوط به Next یعنی Next_count یکی زیاد می‌شود و در مثال ما الان ۱ می‌شود و بعد x_sem، سیگنال می‌شود، یعنی پراسس قبلی که در اینجا *۱ بلوکه شده بود آزاد می‌شود و بلافاصله، این فرآیندی که سیگنال کرده که الان توی x.signal است روی سمافور Next که مقدار اولیه‌اش ۰ است بلوکه می‌شود

*۲ فرآیندی که سیگنال را انجام داد خودش روی Next بلوکه می‌شود

حالا پراسس اول (مربوط به F) در *۱ آزاد می‌شود و x_count را یکی کم می‌کند و x_count صفر می‌شود و بقیه اجرا باری F را از سر می‌گیرد. حالا فرض کنید این فرآیند می‌خواهد مانیتور را ترک کند، حالا قبل از ترک کردنش Next_count را چک می‌کند، حالا اگر دید Next_count بزرگتر از صفره به این معناست که به خاطر آزاد شدن خودش یک فرآیند دیگری روی Next خوابیده، قبل از ترک مانیتور Next را سیگنال می‌کند و دیگه این بار mutex را سیگنال نمی‌کند، یعنی دیگه اجازه نمی‌دهد یک فرآیند دیگر بخواهد از بیرون مانیتور وارد مانیتور بشود، چون در حال حاضر یک فرآیند معطل و خوابیده داخل خود مانیتور وجود دارد که به خاطر فعال شدن خودش (F) خوابیده و معطل شده. حالا با سیگنال کردن Next، فرآیندی که در *۲ بلوکه شده بود آزاد می‌شود و پس Next_count را یکی کاهش می‌دهد، حالا این فرآیند که آزاد شد باری خودش (y) را ادامه می‌دهد، حالا زمانیکه این فرآیند می‌خواهد مانیتور را ترک کند وقتی Next_count را چک می‌کند می‌بیند ۰ است و بنابراین حالا mutex و در واقع مانیتور را آزاد می‌کند و حالا فرآیند بعدی می‌تواند از بیرون مانیتور وارد بشود

حالا فرض کنید فرآیند F که آزاد می‌شود باری F را ادامه بدهد و باز دوباره y.wait را انجام بدهد، حالا تابع y.wait را اجرا می‌کند و y_count را زیاد می‌کند و next_count را چک می‌کند و حالا چون next_count بزرگتر از ۰ است این بار Next را سیگنال می‌کند و در اینجا هم mutex سیگنال نمی‌شود تا فرآیند جدیدی وارد مانیتور نشود و بعد خودش میره روی سمافور y معطل می‌شود

```
y.wait(){
y_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(y_sem);
x_count--;
}
```

پس: چه فرآیندی که آزاد می‌شود (F) مانیتور را ترک کند و چه برود روی یک condition دیگر (y) wait کند باعث می‌شود که ما فرآیند سیگنال کننده‌ای که روی Next معطل شده است را آزاد کنیم.

* چیکار کردیم؟ کاری کردیم که همیشه حداکثر یک فرآیند فعال در مانیتور وجود داشته باشد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

```
enum {THINKING, HUNGRY, EATING} state[5];
 condition self[5];
 DiningPhilosophers.pickup(i);
                    ...
                    eat
                    ...
 DiningPhilosophers.putdown(i);
```

```
 monitor DiningPhilosophers
 {
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
       state[i] = HUNGRY;
       test(i);
       if (state[i] != EATING)
          self[i].wait();
    }

    void putdown(int i) {
       state[i] = THINKING;
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
          state[i] = EATING;
          self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

- می دانیم که منابع سیستم محدود هستند و همین طور قابلیت استفاده مجدد دارند، یعنی یک فرآیند یک منبع را استفاده می کند و کارش با آن منبع تمام می شود آن منبع آزاد می شود و پس می تواند توسط فرآیند دیگری استفاده شود.

- هر فرآیند برای اینکه اجرا بشود ممکن است به تعدادی منبع نیاز داشته باشد

- یک فرآیند وقتی به منبعی نیاز داشته باشد، ابتدا آن منبع را از OS درخواست می کنند، حال اگر آن منبع آزاد باشد OS آن منبع را فورا به فرآیند تخصیص می دهد و اگر منبع آزاد نباشد فرآیند باید منتظر آزاد شدن آن منبع بشود تا بعد از آزاد شدن بهش داده شود و پس از اینکه منبعی در اختیار فرآیند قرار گرفت از آن استفاده می کند و وقتی کارش تمام شد آن منبع را رها می کند

- منظور از منبع چیست؟

processor oR Active Recource (پراسور)

به یک رسورس Active گویند اگر پارامتر سرعت برای آن رسورس مطرح باشد.

مثال هایی از رسورس Active : cpu | شبکه (کارت شبکه) : براش سرعت قائلیم و می گیم پهنا باند شبکه ما اینقدر است

Hard Disk: یک نرخ انتقال داده از هارد دیسک به سایر اجزا سیستم وجود دارد که می تواند به سرعت چرخش صفحه دیسک (مثلاً ۷۲۰۰ RPM - RPM=Revolutions Per minute) و کابل اتصال هارد دیسک به مادربرد و seek time (زمان جستجو شیار: مدت زمان حرکت بازوی دیسک و قرار گرفتن هد روی شیار مورد نظر) بستگی دارد

پس فقط cpu scheduling نداریم

* هر پراسوری را می توان زمانبندی کرد (scheduling) بنابراین واژه ای بنام processor scheduling داریم

cpu scheduling
Disk scheduling
Network "

scheduling به معنای زمانبندی و بهتر از آن نوبت دهی یک معنا عام است و به این معناست که یک رسورس در زمان های مختلف در اختیار چه کسی باشد

۲- passive Resource : منابعی که پارامتر سرعت برایشان مطرح نیست.

فضای دیسک و فضای حافظه نمونه هایی از این نوع منبع هستند، مثلاً یک رم ۳۲GB یا یک هارد ۴TB یک passive resource هستند.

انواع passive resource:

۱- سخت افزاری یا فیزیکی (physical): مثل همان فضا دیسک یا فضا حافظه

۲- منطقی (logical): lock نمونه ای از یک pr - منطقی است، وقتی یک فایل را باز می کنید که روی آن بنویسید (ادیت اش کنید) آن فایل قفل می شود (قفل نوشتن روی آن فعال می شود) به این معنا که مثلاً یک بیت بعنوان قفل در نظر گرفته می شود و در این حالت از ۰ به ۱ تبدیل می شود و هر کسی که می خواهد این فایل را بخواند یا بنویسد به این

Resource Allocation Graph

$T = \{T_1, T_2, \ldots, T_n\} \rightarrow T_i$ تردها را با یک مجموعه به اسم T نمایش می دهیم (در درژن قدیم این ها فرآیند بودند) و تردها با دایره نشان داده می شوند

$R = \{R_1, R_2, \ldots, R_m\}$ منابع سیستم را با مجموعه R نمایش می دهیم و منابع در باکس های مربع و مستطیلی نشان داده می شوند

instance ← ●●●

- اگر چند نسخه یکسان از یک منبع وجود داشته باشد درون باکس آن منبع دایره توپر میذاریم

$T_i \rightarrow R_j$ اگر یک یال از ترد i ام به منبع j وجود داشته باشد یعنی ترد i متقاضی دریافت منبع j است.

$R_j \rightarrow T_i$ " " " " منبع j ام " ترد i " " " " منبع j الان در اختیار ترد i ام است.

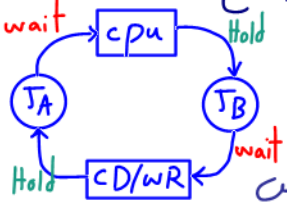
بنابراین ما می توانیم یک گراف بین تردها (فرآیندها) و منابع ایجاد کنیم که بهش گراف تخصیص منابع می گویند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

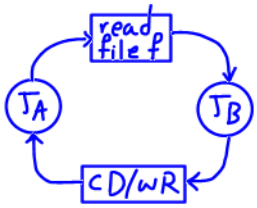
نکته) توجه کنید که بحث بن بست زمانی رخ می دهد که منابع preemptive نباشند، بنابراین ما در بحث بن بست در خصوص منابعی صحبت می کنیم که قابل پس دادن نیستند. بعنوان مثال cpu پس گرفتنی است و وقتی cpu را به یک فرآیند می دهیم نیازی نیست cpu تا انتهای اجرا این فرآیند دست اش باشه و مثلاً بعد از یک کوانتوم cpu رو از فرآیند می گیریم و می دیم به یکی دیگه، اما CD writer یا پرینتر را می دیم دست یک فرآیند وسط کارش نمی تونیم ازش بگیریم و بدیم به یک فرآیند دیگر!

نکته) در بحث بن بست در خصوص منابعی صحبت می کنیم که قابلیت اشتراک گذاری را ندارند، مثلاً از CD/WR را نمی توان بصورت مشترک در یک لحظه استفاده کرد که به این خاصیت انحصار متقابل می گویند، پس CD/WR و پرینتر و پلاتر و... خاصیت انحصار متقابل (mutual exclusion) و مثلاً خواندن از یک فایل خاصیت انحصار متقابل ندارد و چندین فرآیند می توانند با هم از یک فایل بخوانند و از این منبع استفاده کنند

مثال: شکل زیر را تحلیل کنید و بگویید آیا بن بست وجود دارد؟



TA، CD/WR را در اختیار گرفته و داره باهاش کار میکنه پس CD/WR را Hold کرده و پس درخواست cpu هم بیده بنابراین wait کرده برای cpu، حالا cpu دست TB هست یعنی cpu، TB رو Hold کرده و TB برای ادامه کارش به CD/WR نیاز داره پس wait کرده برای CD/WR، خب اینجا یک سیکل داریم که بهش میگن circular wait اما اینجا بن بست نداریم چون cpu یک منبع غیرانحصاری (preemptive) است، اینجوری نیست که این ۲ ترد برای همیشه و تا ابد برای رها کردن منبع توسط دیگری انتظار بکشند. الان توی مثال بالا می تونیم cpu رو از TB بگیریم و بدیم به TA تا (CS) TA کارش را با cpu و CD/WR تمام کنه و بعد می تونیم CD/WR را در اختیار TB قرار بدهیم.



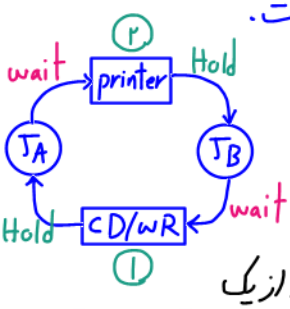
در مثال بالا چرا بن بست نداشتیم؟ چون cpu غیرانحصاری (preemptive) است

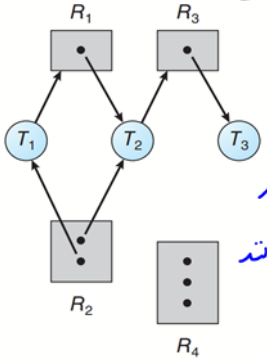
مثال ۲) این مثال هم بن بست ندارد چون TA و TB با هم دیگر می توانند از فایل f بخوانند و خواندن از فایل انحصار متقابل ندارد.

Deadlock

Deadlock: اگر تعدادی فرآیند روی منابع با هم رقابت کنند این امکان وجود دارد که این رقابت منجر به بن بست بشود، مثلاً اگر فرآیند A منبع شماره ۱ را بگیرد و پس نوبت فرآیند B بشود و این فرآیند منبع شماره ۲ را بگیرد و هر کدام از این فرآیندها منبعی را که فرآیند دیگر گرفته است را بخواهد، این اتفاق هیچ وقت محقق نمی شود و در این شرایط می گیم Deadlock رخ داده است.

مثال: در این مثال اولا منابع دارای شرط انحصار متقابل هستند، دوما Non preemptive هستند، سوما یک circular wait تشکیل شده به این معنی که بین فرآیندهایی که درگیر بن بست هستند سیکل تشکیل شده و چهارما شرط Hold & wait برقرار است، یعنی TA، Hold & wait کرده و TB هم Hold & wait کرده و دلیل این اتفاق این است که ما اجازه داریم که ترد و فرآیندها بیان یک منبعی رو بگیرن و Hold اش کنند و ازش استفاده کنند و در حالیکه دارند از یک منبع استفاده می کنند بتوانند درخواست منابع جدیدی بدهند

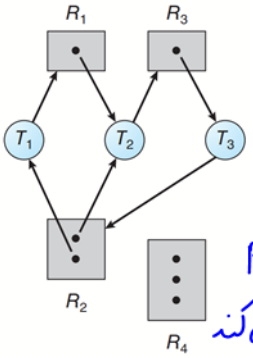




مثال: در این مثال بن بست نداریم، چون T۳، R۴ را در اختیار دارد و چون منبع دیگری نمی خواهد، پس از یک مدت کارش با R۴ تمام میشه و R۳ آزاد میشه و بعد T۲ که متقاضی R۳ است می تواند این منبع را در اختیار بگیرد و چون T۲ تمامی منابعی که نیاز دارد را خواهد داشت می تواند کارش را تمام کند و در نتیجه R۱ آزاد می شود و بنابراین T۱ نیز می تواند کارش را تمام کند، پس تمامی ترد ها می توانند کارشون رو تموم کنند و بنابراین درگیر بن بست نیستند

اینجا سیکل نداریم

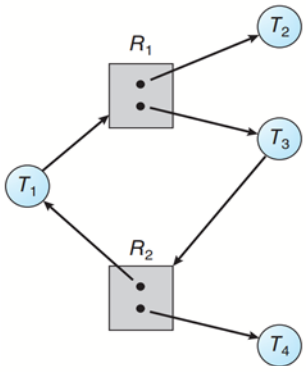
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.



$T_1 \to R_1 \to T_2 \to R_3 \to T_3 \to R_2 \to T_1$
$T_2 \to R_3 \to T_3 \to R_2 \to T_2$

مثال: بن بست داریم. در این مثال ۲ تا سیکل داریم
T_1 متقاضی R_1 است که منبع R_1 در اختیار T_2 است و...

حالا هیچ کدام از این سیکل ها شکسته نمی شود، چون T_3 برای اینکه بتواند منبع R_3 را رها بکند باید بتواند قبل اش منبع R_2 را بدست بیاورد ولی R_2 را نمی تواند بدست آورد مگر اینکه T_1 یا T_2، R_2 را رها بکنند که چون T_2 متقاضی R_3 است، R_2 را رها نمی کند و T_1 هم متقاضی R_1 است پس کارش تمام نمی شود و R_2 را رها نمی کند، پس این ۳ تردرگیر سیکلی هستند که شکسته نمی شود و بنابراین در این شرایط با بن بست مواجه هستیم.



Resource-allocation graph with a cycle but no deadlock.

مثال: در اینجا نیز ما یک سیکل داریم: $T_1 \to R_1 \to T_3 \to R_2 \to T_1$
اما علی رغم وجود این سیکل سیستم درگیر بن بست نیست، چون T_2 تنها منبع مورد نیازش یعنی R_1 را در اختیار دارد پس بعد از اینکه کارش با این منبع تموم شد، این منبع را آزاد می کند و R_1 می تواند در اختیار T_1 قرار بگیرد و بدین ترتیب T_1 هم همه منابع مورد نیازش را در اختیار دارد و می تواند تمام بشود و R_1 و R_2 آزاد می شود و T_3 و T_4 هم می توانند اجرا بشوند

* نتیجه: اگر بن بست وجود داشته باشد حتما سیکلی وجود دارد ولی اگر سیکلی وجود داشته باشد فقط می توانیم بگیم احتمال بن بست وجود دارد مگر آنکه از هر منبع فقط یکی داشته باشیم.
اگر از هر منبع فقط یکی وجود داشته باشد آن گاه وجود سایکل حتما باعث رخداد بن بست می شود.

نکته) ما ۲ تکنیک برای مواجه شدن با بن بست داریم یکی جلوگیری از بن بست و یکی پرهیز از بن بست ولی از الان این رو بدونید که در حالت عمومی بن بست قابل پیشگیری یا اجتناب نیست.

شرایط لازم و کافی برای رخ دادن بن بست: بن بست رخ می دهد اگر و فقط اگر هر ۴ شرط زیر برقرار باشد
۴ شرط لازم که هرکدام برقرار نباشد دیگر امکان رخ دادن بن بست وجود ندارد

۱- mutual exclusion: اگر روی رسورس ها دسترسی انحصاری مطالبه نکنیم روی آن رسورس ها بن بست رخ نمی دهد، مثلاً روی تعدادی رسورس read only هیچ وقت بن بست رخ نمی دهد.

۲- Hold and wait: باید بعضی از فرآیندها منابعی را Hold کرده باشند و منتظر گرفتن منابع دیگر باشند یعنی در شرایط wait به سر ببرند، حالا چه Hold و چه wait توسط این فرآیندها نقض بشود ما دیگر بن بست نخواهیم داشت:

نقض wait: اگر wait نقض بشود به این معنا که یک فرآیند هیچ گاه برای یک منبع wait نکند، بنابراین این فرآیند دارد کارش را اجرا میکند و کارش جلو میره و درگیر بن بست نمی شود.

نقض Hold: اگر Hold نقض بشه یعنی یک فرآیند منبعی را نگرفته باشد، پس این فرآیند عامل بن بست نخواهد بود زیرا هیچ فرآیند دیگری منتظر این فرآیند نمی شود.

۳- No preemption: اگر preemption مجاز باشد ما می توانیم فرآیندی که منبعی را گرفته preempt کنیم و وضعیت آن منبع را برای فرآیند قبلی ذخیره کنیم و منبع ای که گرفته بوده را در اختیار فرآیند دیگری قرار بدهیم و بدین ترتیب دیگر بن بست رخ نمی دهد. پس روی منابع preemptive بن بست وجود ندارد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۴_circular wait: باید بین فرآیندهایی که درگیر بن بست هستند یک سیکل وجود داشته باشد تا بگوییم این سیکل باعث می شود که همه فرآیندهای درون سیکل برای هم انتظار بکشند و در نتیجه درگیر شرایط بن بست بشوند.

Deadlock Handeling

۱_prevention: یعنی اینکه از بن بست جلوگیری شود کافی است یکی از ۴ شرط لازم بن بست نقض بشود، در اینصورت بصورت ساختاری دیگر امکان وجود بن بست وجود نخواهد داشت.

۲_Avoidance: اجتناب از بن بست: بصورت ساختاری امکان وجود بن بست وجود دارد ولی ما در هر تخصیص منبع بررسی می کنیم که اگر این تخصیص احتمال وجود بن بست را به وجود می آورد از آن تخصیص صرف نظر می کنیم.

۳_Deadlock detection and Recovery: هر مدتی یکبار جستجو کنیم که بن بست رخ داده یا نه و اگر رخ داده بود بن بست را بشکنیم چجوری بشکنیم؟ لغو بخشی یا تمام بعضی از فرآیندها، مثلا فرآیندهای جوان را بکشیم، به این ترتیب ما می توانیم یکسری از منابع را آزاد کنیم، هزینه این روش از نمایده اش بیشتره.

۴_Ignoring Deadlock (شترمرغ: ostrich): فرض کنیم هرگز بن بست رخ نمی دهد و هیچ تکنیکی برای مواجهه با آن استفاده نکنیم. استفاده از این روش به ماهیت سیستم هایی که ما ازشون استفاده می کنیم برمی گردد.

بعنوان مثال در سیستم هم منظوره که از os ویندوز استفاده می کنند فرآیندها خیلی روی منابع مشترک رقابت نمی کنند. اما بالعکس این سیستم ها در سیستم های embedded که خاص منظوره هستند و همه تسک های سیستم معمولا یک هدف مشترک را دنبال می کنند از رسورس مشترک بیشتر استفاده می کنند و احتمال رخداد بن بست در این سیستم ها بیشتر است. بنابراین در یک سیستم هم منظوره طراحان os ممکن است سربار ناشی از تکنیک های مواجهه با بن بست را نپذیرند که در این حالت نهایتا اگر در بدترین شرایط و در یک سناریو نادر سیستم با بن بست مواجه شد خود کاربر این موضوع را هندل می کند، حالا یا با پایان دادن به کار یک یا چند فرآیند و یا نهایتا با ریست کردن سیستم. اما این کار در سیستم های هفتد که معمولا کاربر ندارند شدنی نیست.

prevention Deadlock

گفتیم شرایط وقوع بن بست لازم است، پس اگر فقط یکی از ۴ شرط لازم بن بست را نقض کنیم، یعنی کاری کنیم یکی از این ۴ شرط هرگز رخ ندهد، می توانیم اطمینان پیدا کنیم که از بن بست جلوگیری می شود.

در حالت عمومی حمله به هر کدام این ۴ شرط با شکست مواجه می شود

۱_mutual exclusion: اگر این امکان وجود داشته باشد که روی منبع تحت رقابت انحصار متقابل کنار گذاشته بشود، روی آن رسورس بن بستی رخ نمی دهد که این مورد را در خصوص منابع محدودی مثل file های Read only که منابعی با قابلیت اشتراک گذاری هستند می توان انتظار داشت

بعضی جاها با spool کردن همه منابع روی هارد شاید بتوان از بن بست جلوگیری کرد ولی نه.

مثلا spool کردن printer رو در نظر بگیرید، یعنی یک فضایی روی هارد در نظر بگیریم و پرینت ها را مستقیم به پرینتر نفرستیم و همه فرآیندهایی که می خوان از پرینتر استفاده کنند، داده هاشون رو بگیریم و بذاریم روی این فضا توی هارد و بعد از توی این فضا کم کم بفرستیم برای بافر پرینتر اینجوری دیگه فرآیندها منتظر پرینتر نمی شوند و هر فرآیند داده اش رو سپرده توی این فضا و میره به بقیه کارهاش می رسه.

حالا ۲ و ۱: همه منابع قابل spool کردن نیستند، مثل pcb فرآیندها یا فایل ها

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

دوما: فضای spool هارد محدود است و می‌توان سناریوای را در نظر گرفت که این فضا پر بشود و همه فرآیندها بلاک بشوند

۲- نقض Hold and wait: چه Hold و چه wait نقض بشود دیگه امکان رخ دادن بن بست وجود ندارد

۱- کنار گذاشتن شرط wait: هر فرآیند فقط زمانی بتواند اجرایش را شروع کند که تمام منابع مورد نیازش را گرفته باشد که در این حالت هرگز در طول اجرایش دچار waiting نمی‌شود و دیگر این فرآیند درگیر بن بست نمی‌شود

از همون اول یک فرآیند به OS بگه چی می‌خواد حالا یا OS همه اش رو بهش میده یا هیچ کدوم رو بهش نمیده. اگر OS به فرآیند چیزی نده بنابراین فرآیند wait می‌کند و چیزی را Hold نکرده، اگر هم همه منابعی که می‌خواد رو بدیم خب Hold کرده و دیگه wait نمی‌کند پس H&W نداریم و circular wait تشکیل نمیشه و مشکل حل میشه، اما این روش امکان پذیر نیست چون روشی وجود ندارد که بتواند بصورت قطعی و جنرال بتواند پیش بینی کند یا تخمین بزند که فرآیندها در آینده به چه چیزی نیاز دارند

همین طور این روش یک ایراد بزرگ دارد: فرآیندها از ابتدا همه منابع شان را می‌گیرند ولی لزوما در تمام طول اجرا به آنها نیاز ندارند، بنابراین احتمالا utilization منابع پایین خواهد بود.

۲- کنار گذاشتن شرط Hold: هر وقت یک فرآیند درخواست منبع جدیدی داشت ابتدا همه منابع قبلی اش را رها بکند و پس همه منابع مورد نیازش را درخواست بدهد، به اینصورت فرآیندی که دارد انتظار می‌کشد هیچ وقت منبعی دست اش نخواهد بود و بنابراین امکان Holding ازش گرفته می‌شود بدین ترتیب هیچ فرآیند دیگری به خاطر این فرآیند معطل، معطلی نمی‌کشد و بنابراین بن بست رخ نمی‌دهد.

این روش ۲ تا مشکل دارد:

۱) اولا در حالت عمومی ما در مورد منابعی صحبت می‌کنیم که انحصاری (non-preemptive) هستند و نمی‌توانیم پشون بدیم، اگر می‌توانستیم پشون بدیم مثل cpu که اصلا بن بست رخ نمی‌داد که حالا بخواهد حل بشود

*۲) وقتی که یک فرآیند می‌خواهد منبع جدیدی بگیرد منابع قبلی اش را رها می‌کند و برای دریافت منابعی که نیاز دارد wait می‌کند، حالا ممکن است دیگر بسادگی نتواند منابعی که می‌خواهد را بگیرد و احتمال دارد با starvation مواجه شود

۳- نقض non-preemption: اگر روی یک منبع شرایط preemption را فراهم بکنیم دیگر روی آن منبع بن بست رخ نمی‌دهد، مثلاً ما امکان preempt کردن cpu را داریم و بنابراین روی cpu بن بست رخ نمی‌دهد، یا مثلاً در خصوص رم این امکان وجود دارد که بتوانیم محتوای روی رم را به هارد منتقل کنیم و فضای آزاد شده روی رم را بطور موقت به فرآیندهای دیگر بدهیم و در نهایت آن فضا را به فرآیند قبلی برگردانیم پس چون امکان preempt روی رم وجود دارد دچار بن بست نخواهیم شد.

در مورد منابعی که امکان preemption وجود دارد اگر پروتکل زیر را انجام دهیم جلوی بن بست را می‌گیریم

یک فرآیند که به حالت wait می‌رود بصورت ضمنی همه منابع اش را رها کند (یعنی منابع را پس نداده است ولی بصورت ضمنی آنها را آزاد نگه داشته است)، حالا اگر یک فرآیند دیگر منبعی را درخواست بکند بصورت زیر عمل می‌کنیم:

۱- اگر آن منبع آزاد باشد که آن را بهش می‌دهیم.

۲- " " " در اختیار فرآیندی است که در حالت waiting است و بصورت ضمنی منابع اش را رها کرده آن منبع را preempt می‌کنیم و آن را به فرآیند متقاضی تخصیص می‌دهیم.

۳- در غیر اینصورت (اگر منبع دست فرآیندی است که در حالت waiting نیست و دارد اجرا می‌شود) این درخواست قابل انجام نیست و فرآیند متقاضی باید به حالت wait درآید و منابع خودش را بصورت ضمنی آزاد کند

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

عیب این روش: ممکن است با مشکل starvation و low resource utilization مواجه بشیم، شکل بهره وری پایین منبع به این دلیل است که با این روش ما داریم بصورت متعدد منابع را preempt می‌کنیم و مجددا به آن فرآیندهایی که ازشون منابع را گرفتیم بر می‌گردانیم که می‌دانیم این کار دارای overhead است و بنابراین بهره وری منابع کاهش پیدا می‌کند.

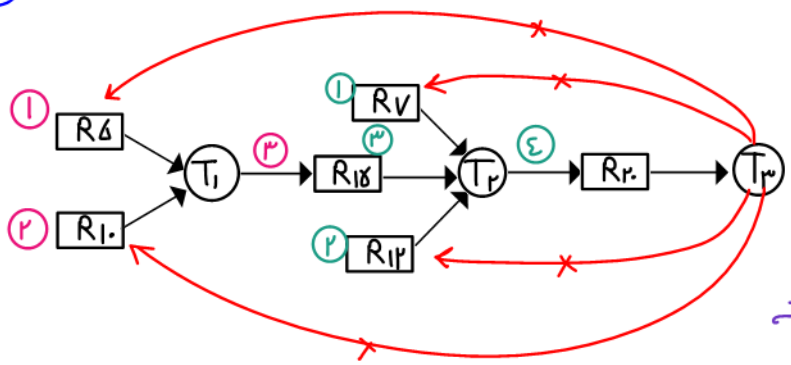
starvation: وقتی یک فرآیند بصورت ضمنی منابع اش را آزاد می‌کند، حال اگر همه یا برخی از آن منابع preempt بشود هیچ گارانتی وجود ندارد که این فرآیند بتواند چه زمانی آن رسورس(ها) را بدست بیاورد

- ازبین بردن شرط انحصاری بودن منابع شدنی نیست و ما نمی‌توانیم یک منبع انحصاری را که یک فرآیند در اختیار گرفته را به زور ازش پس بگیریم و پس گرفتن به زور در این سناریو به معنا کشتن آن فرآیند است، مثلا یک فرآیند داره روی CD چیزی رایت میکند و اگر CD/WR رو ازش بگیریم یعنی فرآیند را کشتیم.

۴- ازبین بردن امکان انتظار چرخشی:

این پروتکل در ماشین‌های mainframe استفاده می‌شود

در این روش بیان منابع را شماره گذاری می‌کنند و نوعی ordering بین منابع قائل بشیم و مکانیزم دریافت منابع توسط فرآیندها به اینصورت باشد که فرآیند فقط بتواند درخواست منابع اش را به ترتیب افزایش رسورس‌ها بدهد، مثلاً اگر فرآیند I ابتدا منبع شماره ۷ و پس منبع ۱۴ را گرفته باشد دیگه امکان این را نخواهد داشت که منبع شماره ۵ را بگیرد، بنابراین در درخواست منابع توسط فرآیندها باید یک نظمی به وجود بیاید و به ترتیب صعودی باشد، حالا اگر یک فرآیند مجبور شد که یک منبع با شماره پایین‌تر را درخواست بدهد باید ابتدا منابعی که شماره بالاتری دارند از این منبع که می‌خواهد درخواست بدهد را رها کند و پس از آن شماره پایین‌تر مجددا شروع کند به دریافت منابع.

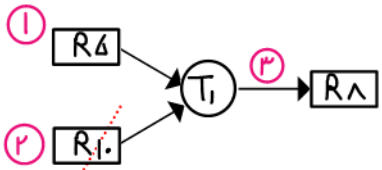


فرض کنید در سیستم ۴۰ تا منبع R_1 تا R_4 وجود دارد

سوال: آیا T۳ می‌تواند R۵ را درخواست بدهد و در نتیجه این درخواست منتظر R۵ بشه که سیکل به وجود بیاد؟ خیر چون ۵ از ۲۰ کوچکتره

بنابراین با این کار زنجیر Hold تشکیل میشه ولی سیکل تشکیل نمیشه

داستان الگوریتم:



فرض کنید T_1 ابتدا R۵ را می‌گیرد و پس R۱۰ را می‌گیرد، حالا نمی‌تونه R۸ را بگیرد و ابتدا باید R۱۰ را رها کند و بعد R۸ را بگیرد. اگر منابع preemptive باشد می‌توان این کار را انجام داد ولی اگر نباشد این روش هم نیاز به پیش آینده دارد، اگر فرآیند بعد از R۵ مجبور است R۱۰ را بگیرد و بعد هنوز که R۱۰ را آزاد نکرده به R۸ نیاز دارد، باید قبل از R۱۰ و پیشاپیش R۸ را می‌گرفت و بعد R۱۰ را می‌گرفت و این نیازمند این هست که از قبل حدس می‌زدیم که در آینده و قبل از آزادکردن R۱۰ به R۸ نیاز داریم و این پیش بینی در حالت کلی غیر ممکن است.

روش دوم مواجهه با بن بست Deadlock Avoidance (اجتناب از بن بست) (Dynamic Avoidanc)

بصورت ساختاری و ذاتی امکان وجود بن بست وجود دارد ولی ما در هر تخصیص منبع بررسی می‌کنیم که آیا آن تخصیص safe است، اگر این تخصیص احتمال وجود بن بست را به وجود به وجود می‌آورد از آن تخصیص صرف نظر می‌کنیم، برای این کار اطلاعات زیر را نیاز داریم:

۱- بدانیم چه میزان منبع آزاد داریم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۲- حداکثر درخواست هرکدام از فرآیندها چقدر است

safe state: به یک state از سیستم = منظور از یک state یک ترکیب تخصیص منابع به فرآیندهای سیستم است، این می گوییم اگر بشود از آن تخصیص شروع کرد و به هر فرآیند حداکثر نیازش را داد هم و هنوز ببینیم که از بن بست اجتناب شده و در نتیجه هم فرآیندها می توانند بدون اینکه درگیر بن بست شوند کارشون رو تموم کنند

safe sequence: ممکن است هر ترکیبی از اجرا فرآیندها نتواند منجر به این بشود که اجرا فرآیندها بدون بن بست پیش برود، حال به آن ترتیبایی که می تواند منجر به اجرا بدون بن بست فرآیندها و اینکه فرآیندها به حداکثر نیازشون برسند ترتیب ایمن می گوییم

مثال: فرض کنید در یک سیستم ۱۲ تا tape خالی وجود دارد و سه فرآیند داشته باشیم که حداکثر نیاز هرکدوم شون مشخص است. حداکثرهایی که در مورد هر فرآیند گفته شده ممکن است یک فرآیند یک دفعه همه اش را بخواهد یا ممکن است به تدریج آنها را بخواهد، حتی ممکن است مثلاً T_0 ممکن است اول ۶ تا بگیرد و بعد ۲ تای دیگر بگیرد و بعد یکی از ۸ تای قبلی را رها کند و بعد ۳ تای دیگه را بگیرد، ولی در بدترین شرایط به ۱۰ منبع نیاز خواهد داشت که البته ممکن است این بدترین شرایط با توجه به ورودی هایش رخ ندهد و مثلاً در یکی از اجراهایش به ۵ تا منبع نیاز داشته باشد

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	2

فرض کنید در حال حاضر درخواست های این سه ترد بصورت روبه رو است.

حالا می خواهیم ببینیم که با این درخواست موافقت بکنیم یا خیر. آیا اگر با این درخواست موافقت کنیم سیستم ما در شرایط ایمن خواهد بود؟ یعنی آیا یک ترتیب ایمن برای اینکه همه ترد ها بتوانند در بدترین شرایط به حداکثر نیازشون برسند وجود خواهد داشت؟

اگر با درخواست های این ۳ ترد موافقت کنیم از ۱۲ منبعی که در اختیار داریم ۹ تاش می رود و منابع در دسترس سیستم ۳ منبع خواهد بود، حالا می دانیم ترد T_0 برای آنکه تمام بشود در آینده حداکثر ۵ تا منبع دیگه می خواهد و T_1 و T_2 در آینده حداکثر ۲ و ۷ تای دیگر می توانند بخواهند حالا اگر این current Need ترد ها رخ بدهد و به این نیاز پاسخ + بدهیم و این تخصیص را انجام بدهیم، آیا سیستم در شرایط ایمن قرار می گیرد؟ با توجه به اینکه برای سیستم فقط ۳ منبع آزادی مانده؟ بله

	Maximum Needs	Current Needs	maximum remaining Needs
T_0	10	5	۵
T_1	4	2	۲
T_2	9	2	۷

system Available = ۳ ① ۱ ② ۵ ③ ۰ ④ ۱۰ → ۳ → ۱۲ $\langle T_1, T_0, T_2 \rangle$

این ۳ منبع آزاد را ابتدا به T_1 که حداکثر نیاز باقیمانده اش ۲ تا است می دهیم و حالا ۱ منبع در دسترس داریم ① و ۴ تا رسورس تخصیص داده شده به T_1 داریم حالا T_1 می تواند با این ۴ منبع کارش را تمام کند، با اتمام T_1 این ۴ تا منبع به سیستم برمی گردد ②، حالا این ۵ منبع را می توانیم به T_0 که حداکثر نیازش ۵ تا هست بدهیم و به این ترتیب هیچ منبع آزادی باقی نمی ماند ③ ولی T_0 هم ۱۰ منبع مورد نیازش را در اختیار دارد پس T_0 می تواند تا آخر و بدون مشکل اجرا شود و کارش را تمام کند، با اتمام T_0 این ۱۰ منبع به سیستم برمی گردد، پس منابع در دسترس سیستم ۱۰ می شوند ④، حالا به T_2، ۷ تایی که می خواد را می دهیم و ۳ تا منبع باقی می ماند و ۹ منبع در اختیار T_2 است و T_2 که تمام شود ۹ منبع را برمی گرداند و تعداد منابع آزاد به ۱۲ برمی گردد، پس همه ترد ها بدون مواجهه با بن بست توانستند کارشون رو تموم کنند، پس ترتیب $\langle T_1, T_0, T_2 \rangle$ یک ترتیب ایمن است پس می توانیم با این تخصیص موافقت

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

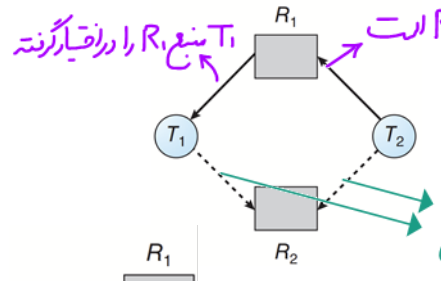
کنیم

سوال مهم: اگر بعد از اینکه با درخواست اولیه موافقت کردیم T_2 می‌آمد از ۷ تا درخواست آینده‌اش مثلاً ۱ درخواست میداد آیا با آن موافقت می‌کردید؟ در این حالت تعداد منبع تخصیص داده شده به T_2 ۳ می‌شد و با قیمانده نیازش ۶ می‌شد و تعداد منبع در دسترس سیستم به جای ۳ ، ۲ می‌شد؟ Available : ۲ → ٤

در این حالت T_1 با ۲ تا منبع باقیمانده کارش راه میفتد و منابع در دسترس ٤ میشد، حالا نمی‌توان به قطعیت بگوییم که این ٤ تا می‌تواند کار T_0 و T_2 را راه می‌اندازد، پس در این شرایط ممکن است بن بست رخ بدهد، چطور، مثلاً اگر T_0 و T_2 هم منابع باقیمانده شان را یکدفعه بدهند T_0 بگوید ۵ تا می‌خواهم یا T_2 بگوید ۶ تا می‌خواهم)، پس در این حالت ممکن است در سیستم شرایطی رخ بدهد که نتواند پاسخ گو باشد، بنابراین برای اجتناب از بن بست باید با این درخواست T_2 مخالف کنیم که در این شرایط T_2 بلوکه می‌شود و اگر چه ۲ تا منبع در اختیارش است سومین منبع را بهش نمی‌دهیم و ۲ تا ترد دیگه به اجرایشان ادامه می‌دهند و بدون برخورد با بن بست کارشون رو تموم می‌کنند. توجه کنید که ممکن بود درخواست T_2 را قبول کنیم و بن بستی هم پیش نیاید، چون ممکن بود شلاً T_2 برنامه‌اش به نحوی جلو برود که با همین ۳ تا منبع اجرا برنامه‌اش تموم بشه و سه تا منبع اش رو آزاد کند

چون می‌خواهیم به هیچ عنوان با بن بست مواجه نشویم باید حداکثر نیاز فرآیندها را در نظر بگیریم و باید بدبینانه‌ترین حالت را در نظر بگیریم.

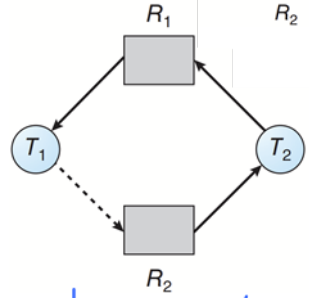
- در گراف تخصیص منابع روبه رو یال‌های جدیدی با نقطه چین مشاهده می‌کنید به این یال‌ها claim edge می‌گویند



T_1 منبع R_1 را در اختیار گرفته — T_2 متقاضی منبع R_1 است

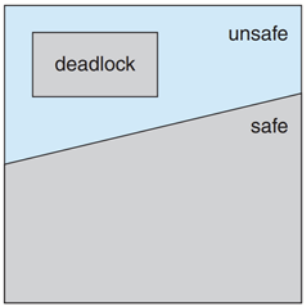
مطالبه

در شرایط کنونی (بدون در نظر گرفتن claim edge ها) سیکلی در این گراف وجود ندارد و بنابراین سیستم در شرایط ایمن قرار دارد



T_1 و T_2 در آینده ممکن است منبع R_2 را درخواست کنند

ولی اگر چند لحظه بعد T_2، منبع R_2 را درخواست کند، حالا چون R_2 آزاد است این اختصاص انجام می‌شود، در این حالت سیستم وارد شرایط unsafe شده است، یعنی اگر طبق چیزی (الگوریتمی) که در بالا آموختیم بریم جلو و حداکثر نیاز تردها را بررسی کنیم، مشاهده می‌کنیم که T_1 نیاز به یک منبع R_2 دارد و اگر در یک مقطعی از زمان بیاد درخواست گرفتن R_2 را که پتانسیل اش وجود دارد را بدهد آن گاه سیکل به وجود می‌آید و چون از هر منبع یک نمونه (instance) وجود دارد بنابراین بن بست خواهیم داشت، توجه کنید که سیستم ما در این لحظه هنوز دچار بن بست نشده زیرا هنوز T_1 این درخواست را نداده است ولی چون پتانسیل این درخواست وجود دارد و پتانسیل رفتن به بن بست وجود دارد باید وقتی T_2 درخواست منبع R_2 را داد ما با این درخواست مخالفت می‌کردیم و R_2 را به T_2 نمی‌دادیم که سیستم در شرایط ایمن باقی بماند. توجه کنید همان طور که قبلاً هم گفتم این امکان وجود دارد که ما با این درخواست موافقت کنیم و T_1 از مسیری برنامه خودش را اجرا کند که درخواست R_2 را ندهد و با بن بست مواجه نشویم، پس اگر این شرایط وجود داشته باشد فقط می‌توان گفت که احتمال وقوع بن بست وجود دارد و نمی‌توانیم بگیم ۱۰۰٪ بن بست رخ می‌دهد.



- برخی از state هایی که unsafe هستند با بن بست مواجه می‌شوند، یعنی اگر با بن بست مواجه بشویم حتما قبل از آن در یک وضعیت unsafe ای قرار داشتیم ولی اگر در شرایط unsafe ای قرار داشته باشیم فقط می‌توانیم بگیم احتمال رخداد بن بست وجود دارد و نمی‌توانیم بگیم ۱۰۰٪ بن بست رخ میدهد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

BanKer's Algorithm

برای اینکه همان بررسی ای بالا را انجام بدهیم و بفهمیم که آیا یک state ای، safe است، یعنی آیا ترتیب ایمنی از آن state وجود دارد یا خیر الگوریتم بانکدار را داریم.

n : تعداد فرآیندها

m : تعداد منابع

$Available_{1xm}$: اگر Available[j] = K باشد به این معناست که ما در سیستم مان از منبع نوع j (Rj) K تا instance داریم

Max_{nxm} : Max[i][j] = K یک ماتریس Max داریم که به تعداد فرآیندها سطر و به تعداد منابع ستون دارد، پس M[i][j] = K یعنی فرآیند i ام از منبع نوع j حداکثر به K تا نیاز دارد

$Allocation_{nxm}$: Allocation[i][j] = K : به فرآیند i ام از منبع j، K تا اختصاص داده شده

Need[i][j] = Max[i][j] - Allocation[i][j] = K : فرآیند i ام از منبع نوع j حداکثر به K تای دیگرش نیاز دارد

مقادیر و ابعاد این ساختمان داده ها با توجه به اختصاص و بازپس گیری منابع تغییر می کند و چون تعداد فرآیندهای سیستم ممکن است کم و زیاد بشود بنابراین تعداد سطرهای این ماتریس ها کم و زیاد می شود

$Allocation_i, Need_i, Max_i$: اگر این ماتریس های nxm را فقط با یک اندیس i نام ببریم یعنی سطر مربوط به فرآیند i ام از این ماتریس ها که هر کدوم شون یک بردار 1xm محسوب می شوند.

بردار X کوچکتر مساوی Y است، اگر و فقط اگر تمامی اعداد x کوچکتر مساوی تمام اعداد Y است $X \leq Y \text{ iff } x[i] \leq Y[i]$

حداقل یکی از اعضای X که مساوی عضو متناظرش در Y نباشد و کوچکتر باشد $X < Y$ iff " and $X \neq Y \Rightarrow$

safety Algorithm

این الگوریتم بررسی می کند که آیا شرایط جاری یک وضعیت safe محسوب می شود و آیا این امکان برای تمامی فرآیندها مهیا است که تا انتها اجرا بشوند و به حداکثر نیازشون برسند. در این الگوریتم ۲ متغیر موقتی work و finish تعریف می شود که work یک بردار معادل Available است که نشان می دهد از هر منبع چه تعداد در دسترس است و finish یک برداری است که به تعداد فرآیندها عنصر دارد و در ابتدا همه عناصرش با False مقداردهی اولیه شده است. در ابتدا فرض می شود تمام فرآیندها تا این لحظه تمام نشده هستند و می خواهیم بررسی کنیم با داشتن Available آیا این امکان وجود دارد که این فرآیندها بتوانند اجرا بشوند. حالا به ازای هر کدام از فرآیندهایی که finish شون برابر False است یعنی تموم نشدند چک می کند که آیا Need شون از work یعنی تعداد منابع در دسترس کنونی هست یا خیر اگر اینگونه بود ما میفهمیم که این فرآیند با توجه به Need اش می تواند تا انتها با همین تعداد منابع در دسترس اجرا بشود خاتمه پیدا کند و منابعی که قبلا بخش تخصیص داده شده بود را به Available ها اضافه کند و بنابراین finish مربوط به آن فرآیند را true در نظر می گیریم و پس همین کار را برای فرآیند بعدی ای که finish اش برابر False است تکرار می کنیم حالا در چه شرایطی این کار خاتمه پیدا می کند؟ ۱- یا به این علت که همه finish شون true شدند که این بدان معنی است که همه فرآیندها می توانستند تمامی منابع مورد نیاز شان را بدون مواجه شدن با بن بست بدست بیاورند که در این حالت می گیم این وضعیت یک وضعیت safe است

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Need_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[i] = *true*
   Go to step 2.
4. If *Finish*[i] == *true* for all i, then the system is in a safe state.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۲- یا اینکه یکی از فرآیندهایی که finish اش هنوز برابر false است، Need اش از work یا همان Available بیشتر بوده و بنابراین این فرآیند به همراه سایر فرآیندهایی که این شرایط را داشتند، فرآیندهای درگیدن بن بست محسوب می‌شوند و بنابراین اگر ما بخواهیم با این وضعیتی که بعنوان ورودی به الگوریتم safety داریم کار را ادامه بدهیم در این شرایط سیستم پتانسیل مواجهه با بن بست را خواهد داشت

خروجی و نتیجه الگوریتم safety : آیا وضعیت فعلی safe هست یا نه و از این خروجی و نتیجه در الگوریتم بانکدار استفاده می‌شود.

Banker's Algorithm

قرارشد هروقت درخواست منبعی از جانب یک فرآیند می‌آید تصمیم بگیریم که آیا با آن درخواست موافقت کنیم یا خیر. اگر درخواست بخواهد سیستم را به حالت نا امن ببرد باید با آن درخواست مخالفت بشود و آن فرآیند بلوکه بشود. این الگوریتم بصورت زیر عمل می‌کند:

ابتدا درخواستی که می‌آید را با Need آن فرآیند مقایسه می‌کند که بررسی کند که اصلا درخواست این فرآیند از Need اش کمتر هست یا خیر که اگر به این صورت نباشد یعنی یک اختلالی وجود دارد، حالا درصورتیکه درخواست فرآیند کوچکتر از نیازش باشد حالا درخواست را با تعداد منابع در دسترس مقایسه می‌کند، اگر این درخواست از تعداد منابع در دسترس بیشتر باشد، خب ما این تعداد منبع را ندادیم و این فرآیند باید فعلا بلوکه بشود تا میزان منابع در دسترس از طریق آزاد شدن منابع سایر فرآیندها افزایش پیدا کند، حالا اگر این شرط هم برقرار بود، هنوز مطمئن نیستم که این منابع را بدهیم به این فرآیند یا خیر، برای همین میایم و به شکل مجازی این تخصیص را انجام می‌دهیم و فرض می‌کنیم که میزان درخواست فرآیند را از تعداد منابع در دسترس سیستم کم اش می‌کنیم و به منابع در اختیار آن فرآیند اضافه اش کردیم و از Need آن فرآیند برش داشتیم و پس الگوریتم safety را اجرا می‌کنیم، حالا اگر خروجی الگوریتم safety این بود که وضعیت امن است این تخصیص را نهایی می‌کنیم و با درخواست این فرآیند موافقت می‌شود و اگر خروجی الگوریتم safety این بود که وضعیت امن نیست مقدار این ساختمان داده‌هایی که تغییر کردند بازیابی می‌شود و فرآیند را موقتا بلاک می‌کند تا شرایط تغییر کند (گویی به فرآیند می‌گیم فعلا منبعی که می‌خوای در دسترس نیست)

1. If $Request_i \le Need_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $Request_i \le Available$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_i by modifying the state as follows:

$$Available = Available - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

If the resulting resource-allocation state is safe, the transaction is completed, and thread T_i is allocated its resources. However, if the new state is unsafe, then T_i must wait for $Request_i$, and the old resource-allocation state is restored.

مثال: در شرایط زیر آیا سیستم امن است یا خیر؟

۵ ترد داریم و سه تایپ از منابع A و B و C. ۲ تا ماتریس Allocation و بردار Available را داریم. بردار Available می‌گوید که در این لحظه سه تا منبع A، سه تا B و دو تا C آزاد است.

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
T_0	0 1 0	7 5 3	3 3 2	7 4 3 → حداکثر نیاز باقیمانده T_0
T_1	2 0 0	3 2 2	T_3: 5 4 3	1 2 2
T_2	3 0 2	9 0 2	T_2: 5 4 5	6 0 0
T_3	2 1 1	2 2 2	T_1: 7 4 5	0 1 1
T_4	0 0 2	4 3 3		4 3 1

چند تا از منبع A به تردهای تخصیص داده شده؟ ۲+۳+۲=۷

پس از ابتدا ۱۰ تا منبع از نوع A داشتیم.

از روی Allocation و Max می‌توانیم به ماتریس Need برسیم.

حالا باید ببینیم با Available موجود چگونه می‌توانیم نیازهای (Need) تردها را پاسخ بدهیم.

بعنوان مثال با این Available = $A_3 B_3 C_2$ می‌توانیم نیاز T_1 و T_3 را برآورده کنیم، حالا مثلاً منابع T_3 را بهش میدهیم و اجرایش می‌کنیم و یک B و یک

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

c بهش می دهیم و پس از اتمام T_3 نه تنها این یک عدد B و c برمی گردد بلکه منابعی که T_3 از قبل هم گرفته بود آزاد می شود، پس A B C ۲ ۱ ۱ هم به Available اضافه می شود و Available برابر ۵،۴،۳ می شود که با این Available می توانیم، T_1 و T_4 را اجرا کنیم، چون ۳ و ۴ و ۵ از حداکثر نیازهای T_4 و T_1 برای B، A و c بیشتر است، حالا مثلاً T_4 را اجرا می کنیم و پس از اتمام T_4، Allocation اش را با Available سیستم جمع می زنیم حالا با Available = ۴، ۴، ۵ فقط می توانیم T_1 را اجرا کنیم و پس از اتمام T_1، Allocation اش را با Available سیستم جمع می زنیم و پس می توانیم T_0 و T_2 را اجرا کنیم، پس یکی از ترتیب های این برابر:

$\langle T_3, T_4, T_1, T_0, T_2 \rangle$

$\langle T_3, T_4, T_1, T_2, T_0 \rangle$

$\langle T_1, T_3, T_4, T_2, T_0 \rangle$

این وضعیت (state) امن است.

★ سوال بسیار مهم (کاندید کنکور): حال که این حالت امن است اگر چند لحظه بعد T_1 درخواست (۲،۰،۱) را می دارد، آیا اگر با این درخواست موافقت کنیم وضعیت امن خواهد بود؟

	Allocation	Need	Available
	A B C	A B C	A B C
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $\langle T_1, T_3, T_4, T_0, T_2 \rangle$ satisfies the safety requirement. Hence, we can immediately grant the request

★ سوال بسیار مهم (کاندید کنکور): حال که این حالت امن است اگر چند لحظه بعد T_4 درخواست (۰،۳،۳) را می دارد، آیا اگر با این درخواست موافقت کنیم وضعیت امن خواهد بود؟ برای درخواست T_0 برای (۰،۲،۰) چطور؟

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by T_4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by T_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

ترین بشتد: الگوریتم بانکدار برای حالتی که single Resource داریم

وضعیت امن: وضعیتی که حداقل یک سید امن دارد، پس اگر بتوانیم فقط ۱ ترتیب پیدا کنیم که بتوانیم یکی یکی فرآیندها را از اول تا آخرشون اجرا کنیم می گیم اون وضعیت امنه

سید امن: یک ترتیب اجرا تا تکمیل برای همه فرآیندها

وضعیت ناامن: احتمال رخ دادن بن بست در آینده وجود دارد که اگر رخ بدهد دیگه راه گریزی نیست

b: وضعیت امنی است: چرا؟ چون در ما حداقل یک سید امن داریم، حداقل یک ترتیب اجرا تا تکمیل برای همه فرآیندها داریم که همشون بدون هیچ مشکلی اجرا بشوند

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

c : ناامن: در وضعیت c ریسک کردیم و اگر بن بست رخ بدهد هیچ راه فراری هیچ مسیر امنی نداریم، یعنی هیچ ترتیب اجرا تا تکمیل برای همه نداریم.

	Has	Max	Need	
A	1	6	۵	> Ava=۲ ✓③
B	1	5	۴	> Ava=۲ ✓④
C	2	4	۲	≤ Ava=۲ ✓①
D	4	7	۳	> Ava=۲ ✓②

باید بتدنیم کدام همه فرآیندها تیک بزنیم که بگیم وضعیت امن، اگر حتی کنار یک فرآیند تیک نخورد می‌گیم این وضعیت ناامن.

$A(New) = A(old) + Has - Allocation(x)$

ابتدا Need هر فرآیند را با Available چک می‌کنیم → فعلا تا اینجا امن → پس باید با c شروع کنیم

۱- منابع c را بهش میدیم و اجراش می‌کنیم و منابع اش رو آزاد می‌کند و Ava = ۴ می‌شود (میدان ما با c شروع می‌شود)

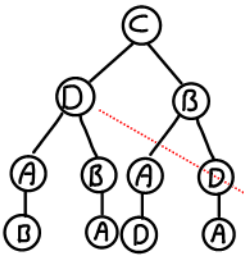
Ava = Free: 2

اجرا c = A = ۲+۲ = ۴

اجرا D = A = ۴+۴ = ۸

اجرا A = A = ۸+۱ = ۹

اجرا B = A = ۹+۱ = ۱۰



۲- حالا با Ava جدید هم می‌توانیم B و هم D را انتخاب و اجرا کنیم، من اومدم را انتخاب می‌کنم که منابع بیشتری در اختیار گرفته، هرچی منبع می‌خواد بهش میدیم و پس D را اجرا می‌کنیم

۳- حالا Ava جدید ۸ است و هم A و هم B می‌توانند اجرا شود

ابتدا c را اجرا کردیم و Ava=۴ شد و بعد از اجرا c دوتا انتخاب داشتیم، اجرا B یا D، وقتی D را اجرا کردیم Ava=۸ شد که با ۸ منبع می‌توانیم A را اجرا کنیم و B هم که از قبل می‌شد و بنابراین حتما الان هم می‌شود، توجه کنید که هرچه جلوتر برویم منبع بیشتری آزاد می‌شود و اوضاع بهتر می‌شود و بدتر نخواهد شد، پس بعد از D می‌توانیم A یا B را اجرا کنیم، حالا بعد از A، B و بعد از B، A اجرا می‌شود

اگر بعد از c که بعد از اجرایش Ava=۴ بود B را اجرا می‌کردیم، Ava=۵ می‌شد که با ۵ منبع آزاد هم بعد از B هم A و هم D را می‌توان اجرا کرد - پس ما در این وضعیت ۴ مسیر امن داریم، فقط ۱ مسیر کافی بود تا بگیم وضعیت امن است.

شکل c:

	Has	Max	Need	
A	1	6	۵	> Ava=۱
B	2	5	۳	> Ava=۱
C	2	4	۲	> Ava=۱
D	4	7	۳	> Ava=۱

Free: 1

این تری همون گام اول نشد، حالا شاید توی گام دوم یا سوم نمی‌شد، هم نیست، اگر توی یک گام نتونیم می‌گیم ناامن است.

پس ما اینجا مسیر امن نداریم و اگر بن بست رخ بدهد دیگه نمی‌تونیم ازش خلاص بشیم.

مثلا مجبوری بن بست بشه؟ مثلاً الان c اگر بگه ۲ تا منبع می‌خوام، خب نداریم و ما بلاک می‌کنیم حالا اگر بعد از c، A و B و D هم حداقل ۲ تا بخوان همشون بلاک می‌شوند و بنابراین چون بلاک می‌شوند اجراشون هم تموم نمیشه که بخواهند منابع در اختیارشون رو آزاد کنند و بنابراین بن بست میشه.

حالا ممکن هم هست بن بست رخ بدهد، چطوری؟ توجه کنید که اولا فرآیندها می‌توانند سیدهای اجرایی مختلفی داشته باشند و در سیدهای مختلف رفتارهای مختلفی داشته باشند و دوما فرآیندها اینجوری نیستند که منابع را بصورت صعودی بگیرند تا به Max نیازشون برسند و همی منابع را می‌گیرند و آزاد می‌کنند، حالا فرض کنید c به جای اینکه بگه من دوتا می‌خوام بیاد دوتایی رو که داره آزاد کند که در اینصورت این وضعیتی که الان ناامن است به وضعیت امن تبدیل میشه، اما ما بدبینانه در نظر می‌گیریم.

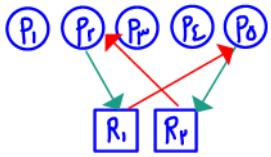
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

اجتناب از بن بست با کنترل پذیرش فرآیند

قبل از اینکه یک فرآیند یا ترد را قبول کنیم چک می‌کنیم که آیا با پذیرش این فرآیند یا ترد سیستم در حالت بدون بن بست باقی می‌ماند یا خیر و فرآیندهایی را می‌پذیریم که امکان بن بست را به وجود نیاورد.

مثال: فرض کنید یک سیستمی داریم که ۵ فرآیند دارد و ۲ منبع داریم و Max نیاز هر فرآیند ۲ منبع است، چند حالت بن بست در این سیستم می‌تواند رخ بدهد؟

فرض کنید الان منبع R_2 دست P_2 است و منبع R_1 دست P_5 است و P_2 منتظر منبع R_1 است و P_5 منتظر منبع R_2 است، در این حالت بن بست داریم، همین داستان را می‌توانیم در خصوص هر ۲ فرآیند دیگری بگوییم

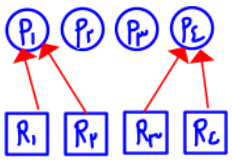


هم منابع یکسان اند

$\binom{5}{2} = \frac{5!}{3!2!} = \frac{5\times4}{2} = 10$

مثال: فرض کنید یک سیستمی داریم که ۴ فرآیند دارد و ۴ منبع داریم و Max نیاز هر فرآیند ۳ منبع است، چند حالت بن بست در این سیستم می‌تواند رخ بدهد؟

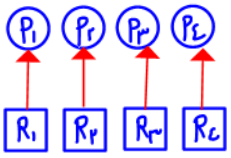
P_1 دو تا منبع را در اختیار دارد و P_4 هم ۲ منبع دیگر را و هر ۲ فرآیند منتظر منبع سوم هستند که منبع دیگری وجود ندارد پس بن بست رخ می‌دهد، حالا همین داستان برای هر ۲ فرآیندی می‌تواند رخ بدهد.



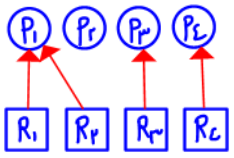
* هم منابع یکسان اند

$\binom{4}{2} = \frac{4!}{2!2!} = \frac{3\times4}{2} = 6$

هر فرآیند ۱ منبع را بگیرد و هر کدام منتظر منبع دوم باشند : ۱ حالت



یک فرآیند ۲ منبع بگیرد و منتظر منبع سوم باشد و ۲ فرآیند دیگر هر کدام ۱ منبع بگیرند و منتظر دو تای باشند



: $3\times4=12$

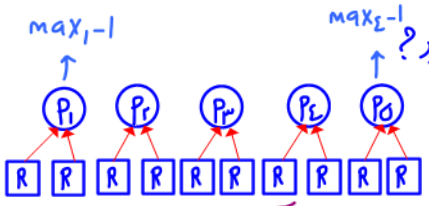
۲ در کجاهای این بردار می‌تواند قرار بگیرد؟ در ۴ جا. یعنی هریک از فرآیندهای P_1 تا P_4 می‌توانند دو تا منبع بگیرند پس $\binom{4}{1}$ تا حالت برای جایگاه ۲ داریم. حالا ۷ حالت * چند حالت دارد؟ $\binom{3}{1} = 3 = \binom{3}{2}$

پاسخ: $12+1+6=19$

- وضعیتی که درونش یکی از فرآیندها ۳ منبع را گرفته و یک فرآیند دیگر یکی بن بست رخ نمی‌دهد، چون فرآیندی که ۳ منبع خود را گرفته Max نیازش را دریافت کرده و می‌تواند اجرا شود و منابع اش را آزاد کند که فرآیندهای دیگر استفاده کنند

مثال: فرض کنید ۵ فرآیند داریم و ۱۰ منبع و Max نیاز هر فرآیند ۳ است، چند وضعیت بن بست وجود دارد؟

هر فرآیند ۲ تا منبع گرفته باشد و همه فرآیندها منتظر منبع سوم باشند، در این حالت بن بست خواهیم داشت، فقط همین ۱ وضعیت بن بست را داریم



* در این مثال اگر ۱۱ منبع داشته باشیم دیگه بن بست نخواهیم داشت و سیستم فاقد بن بست خواهد بود.

در مثال بالا اومدیم به هر فرآیند به اندازه $Max_i - 1$ منبع دادیم، که در این حالت تعداد منابعی که این‌ها اشغال می‌کنند برابر: $(Max_i - 1)$ × تعداد فرآیند خواهد بود، در مثال ما: $5\times(3-1)=10$، حالا اگر ۱۰ منبع داشته باشیم سیستم فقط ۱ وضعیت بن بست دارد ولی اگر تعداد منابع بزرگتر از ۱۰ تا باشد دیگه سیستم فاقد بن بست خواهد بود، بنابراین:

برای آنکه سیستم فاقد بن بست باشد باید: (هم منابع یکسان)

$R > N\times(Max-1)$

(تعداد منابع ← R، تعداد فرآیندها ← N)

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

در صورتی که ماکزیمم نیاز فرآیندها یکسان نباشد شفا فرمول زیر را داریم: (همه منابع یکسان اند)

$$R > \sum_{i=1}^{N}(Max_i - 1) \Rightarrow R > \sum_{i=1}^{N} Max_i - N$$

سوال) در یک سیستم ۴۰ تا فرآیند داریم، حالا اگر هر فرآیند حداکثر به ۳ عدد از سورس x نیاز داشته باشند، سیستم چند منبع x باید داشته باشد که سیستم فاقد بن بست باشد؟

$R > N \times (MAx - 1) \Rightarrow R > 40 \times 2 \Rightarrow R > 80 \Rightarrow$ حداقل به ۸۱ منبع x نیاز داریم

* حالا اگر چند نوع منبع داشته باشیم چیکار کنیم؟

و تا چند منبع داریم باید برای هر منبع شرط زیر برقرار باشد: یعنی برای هر منبع فرمول زیر را باید بدست آورید:

$$R_i : R_i \geq \sum_{i=1}^{n} max_i$$

مثال: فرض کنید ۳ فرآیند داریم و ۲ نوع منبع داریم و هر فرآیند حداکثر ۴ تا R_1 و ۳ تا R_2 می خواهد، چند منبع R_1 و R_2 نیاز داریم که سیستم فاقد بن بست باشد

$R_1 : R_1 \geq 3 \times 4 \Rightarrow \min R_1 = 12$

$R_2 : R_2 \geq 3 \times 3 \Rightarrow \min R_2 = 9$

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

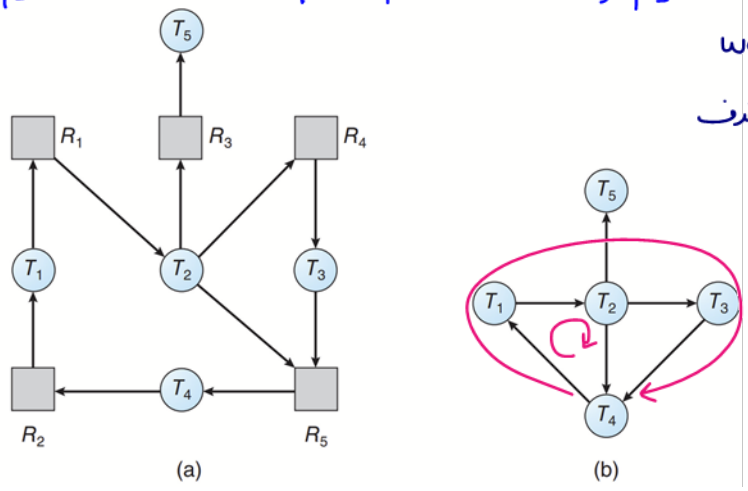
Deadlock Detection and Recovery

روش سومی که برای مواجهه با بن بست وجود دارد کشف و ترمیم بن بست است، یعنی بذاریم بن بست رخ بدهد و پس ما با استفاده از یک الگوریتم Detection بن بست را کشف کنیم و پس با استفاده از یک الگوریتم Recovery سعی کنیم سیستم را از بن بست نجات دهیم.

- از روی گراف تخصیص منبع می توان یک گراف به اسم wait_for graph ایجاد کرد که به اینصورت که باکس های منابع را از توی گراف تخصیص منبع را حذف می کنیم و فقط ارتباط بین فرآیندها باقی می ماند.

مثلاً: میدانیم که T_1 متقاضی R_1 ای است که این منبع در اختیار T_2 است پس می توان گفت که T_1 برای T_2، wait می کند و به همین ترتیب

حالا اگر منابع ما single instance باشند اگر بتوانیم در این گراف سیکل پیدا کنیم یعنی حتما بن بست داریم



(a) Resource-allocation graph. (b) Corresponding wait-for graph

* مرتبه زمانی الگوریتم Deadlock Detection برابر $O(n^2)$ است

الگوریتمی برای زمانی که تعداد instance ـ منابع بیش از یکی باشد

با اجرای این الگوریتم متوجه می شویم که آیا در حال حاضر سیستم دچار بن بست شده است یا خیر

در این الگوریتم دو متغیر موقتی بنام work و finish داریم که مثل قبل work یک بردار است که Available جاری سیستم در آن قرار می گیرد و نشان می دهد در آن لحظه از هر منبع چه تعداد در دسترس است و finish یک برداری است که به تعداد فرآیندها عنصر دارد و در ابتدا هم عناصرش برای فرآیندهایی که هنوز تمام نشدند و Allocation شون هم ۰ نیست به مقدار false مقداردهی اولیه می شوند، یعنی اگر یک فرآیندی Allocation اش ۰ است، finish اش را True فرض می کنیم چون فرآیندی که Allocation جاری اش ۰ است فرآیندی است که هیچ منبعی در دست اش نیست و بنا براین این فرآیند نمی تواند باعث بشود که سایر فرآیندها معطل بکشند و بنا براین فرآیندی نیست که بتواند در گیر بن بست باشد و چون ما داریم به دنبال فرآیندهای درگیر بن بست می گردیم می توانیم finish این چنین فرآیندهایی را True فرض کنیم. بنابراین به دنبال فرآیندی می گردیم که finish اش false است ولی درخواست جاری اش از work=available کمتر است، حالا اگر چنین فرآیندی وجود داشت فرض می کنیم که این فرآیند درادامه درخواستی ندارد و با همان منابعی که در اختیارش هست و درخواست های جاری که بهش می دهیم می تواند اجرا شود و پس تمام شود و پس از آنکه تمام شد می آید و منابعی که بهش اختصاص داده شده بود (Allocation) را آزاد می کند و در نهایت

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available*. For $i = 0, 1, ..., n-1$, if *Allocation*$_i \neq 0$, then *Finish*[i] = *false*. Otherwise, *Finish*[i] = *true*.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. *Request*$_i \leq$ *Work*

   If no such i exists, go to step 4.
3. *Work* = *Work* + *Allocation*$_i$
   Finish[i] = *true*
   Go to step 2.
4. If *Finish*[i] == *false* for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if *Finish*[i] == *false*, then thread T_i is deadlocked.

finish اش true می شود، پس این کار را به ازای هر i ای (هر فرآیندی) انجام می دهیم، حالا در انتها هم finish ها true می شوند که به این معنی است که هر فرآیندی می تواند با منابعی که تحت اختیارشون هست، یعنی Allocation شان و درخواست های جاری شون اجرایشان را تمام کنند و بنابراین می گیم سیستم دچار بن بست نیست، یا اینکه می بینیم تعدادی فرآیند وجود دارد که درخواست هاشون از available باقیمانده بیشتر است و چون با توجه به درخواست هایی که دارند ما نمی توانیم به هیچ کدام شان پاسخ بدهیم و بنابراین این فرآیندها نمی توانند به اجراشون ادامه بدهند و finish شان false می ماند، حالا این فرآیندها هم درگیر بن بست هستند و در واقع

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

این فرآیندی سیکل بن بست را تشکیل می‌دهند که باید با استفاده از الگوریتم‌های ترمیم چاره به حال این بکنیم

نکته: مرتبه زمانی این الگوریتم $O(m \times n^2)$ است.

مثال: عین الگوریتم بانکدار عمل کنید و فقط به جای Need از request استفاده کنید. در مثال زیر دارید تخصیص جاری به هر ترد و همین طور درخواست‌های آن‌ها در لحظه جاری را مشاهده می‌کنید. توجه کنید که در اینجا کاری به الگوریتم اجتناب از بن بست نداریم و بنابراین فرض نمی‌کنیم که حداکثر درخواست هر فرآیند را می‌دانیم و فقط از درخواست لحظه فعلی هر فرآیند آگاه هستیم و فقط بر همین اساس می‌توانیم تصمیم بگیریم که آیا سیستم درگیر بن بست شده یا خیر.

پس مثل قبل نیست که در خصوص اجتناب از درگیر شدن به بن بست در آینده صحبت کنیم

	Allocation	Request	Available
	A B C	A B C	A B C
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	0 1 0
T_2	3 0 3	0 0 0	3 1 3
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

سوال: آیا این سیستم درگیر بن بست است؟ باید بررسی کنیم که در شرایط فعلی آیا همه تردها می‌توانند اجرا بشوند و تمام شوند.

خب T_0 هیچ درخواستی ندارد، پس T_0 با منابعی که الان در اختیارش هست کارش را می‌تواند تمام کند، پس بعد از اتمام T_0، Available سیستم 0 1 0 است، T_2 هم همین شرایط را دارد، پس بعد از اتمام T_2، " " 3 1 3 " ، با این منابع سیستم هم می‌توانیم درخواست سایر تردها را جواب بدهیم، بنابراین یکی از چندین ترتیب این خواهیم داشت و سیستم درگیر بن بست نیست $\langle T_0, T_2, T_3, T_1, T_4 \rangle$

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

سوال: اگر در این وضعیت T_2 یک درخواست روی منبع C انجام بدهد چی؟

بعد از اجرای T_0، Available سیستم برابر 0 1 0 می‌شود ولی با این Available نمی‌توانیم درخواست‌های T_2 و سایر تردها را پاسخ بدهیم و کارشون را راه بندازیم، بنابراین با این درخواستی که T_2 می‌دهد سیستم به بن بست می‌رود و فقط T_0 درگیر بن بست نیست و سایر تردهایی که نمی‌توانند اجرا بشوند تردهای درگیر بن بست هستند حالا باید با استفاده از الگوریتم‌های ریکاوری به دنبال رفع این بست باشیم.

چه زمانی الگوریتم Deadlock Detection را اجرا کنیم؟

۱- بیاییم و به ازای هر تخصیصی که می‌خواهیم به تردها (یا فرآیندها) بدهیم اجرا کنیم، که در اینصورت سربار زیادی خواهیم داشت ولی در این حالت به محض اینکه سیستم درگیر بن بست بشود متوجه می‌شود که کدام فرآیند بوده که عامل ایجاد بن بست بوده

۲- الگوریتم را با فرکانس پایین‌تری اجرا کنیم که سربار پایین‌تری داشته باشد ولی وقتی متوجه بن بست می‌شویم چندین ترد درگیر بن بست شده اند و ما دقیقا نمی‌دانیم که کدام یک از تردهای درگیر بن بست عامل این بن بست بوده است.

۳- هر وقت سربار سیستم شلوغ بیاییم با فرکانس پایین این الگوریتم را اجرا کنیم و هر وقت سربار سیستم خلوت است فرکانس را بالا ببریم، که این ایده هم خیلی خوب نیست چون اتفاقاً وقتی سیستم در شرایط overload قرار دارد احتمال وقوع بن بست بالاتر است و خب حالا اگر در این وضعیت الگوریتم را با فرکانس پایین‌تری اجرا کنیم همان اتفاق ۲ می‌تواند رخ بدهد و دیگه چاره‌ای جز خاتمه دادن به چند ترد نخواهیم داشت که این موضوع نیز می‌تواند کارایی سیستم را پایین بیاورد

Deadlock Recovery

۱- ساده‌ترین راه کشتن فرآیندی است که این کار را می‌توان به چندین روش انجام داد.

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۱-۱: تمام فرآیندهای درگیر بن بست را terminate کنیم، یعنی همه این دور فرآیندها را Abort کنیم، که این کار هزینه بالایی دارد

یادآوری: Abortion: پایان غیرعادی فرآیند، یعنی خودشون در حالت عادی تموم نشدند و ما مجبورشون می‌کنیم که قطع بشوند

۱-۲: تا زمانیکه الگوریتم تشخیص بن بست (Deadlock Detection) تشخیص بدهد که دیگه بن بست نداریم بیایم و فرآیندها را یکی یکی Abort کنیم، که در این روش برای اینکه تصمیم بگیریم که به چه ترتیبی این فرآیندها را Abort کنیم نیاز به یکسری فاکتور هزینه (cost factor) داریم:

فاکتورهای هزینه برای انتخاب victim جهت: هدف این است که فرآیند قربانی به نحوی انتخاب شود که هزینه مینیمم بشود

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated

۳- هر فرآیند چند منبع و چه نوع منابعی استفاده کرده و اگر این منابع را از این فرآیند بگیریم و این فرآیند را Abort کنیم چقدر هزینه برای ما ایجاد می‌کند

۴- اگر یک فرآیند برای تکمیل شدن به تعدادکمی منبع نیاز دارد این فرآیند انتخاب مناسبی برای Abort کردن نیست

۵- با این ترتیبی که داریم فرآیندها را برای شکستن بن بست Abort می‌کنیم در نهایت چند فرآیند terminate می‌شوند، ممکن است در یک ترتیب تعداد کمتر و در ترتیب دیگری تعداد بیشتری فرآیند خاتمه پیدا کنند

۶- فرآیند batch را انتخاب کنیم یا فرآیند interactive؟

Rollback

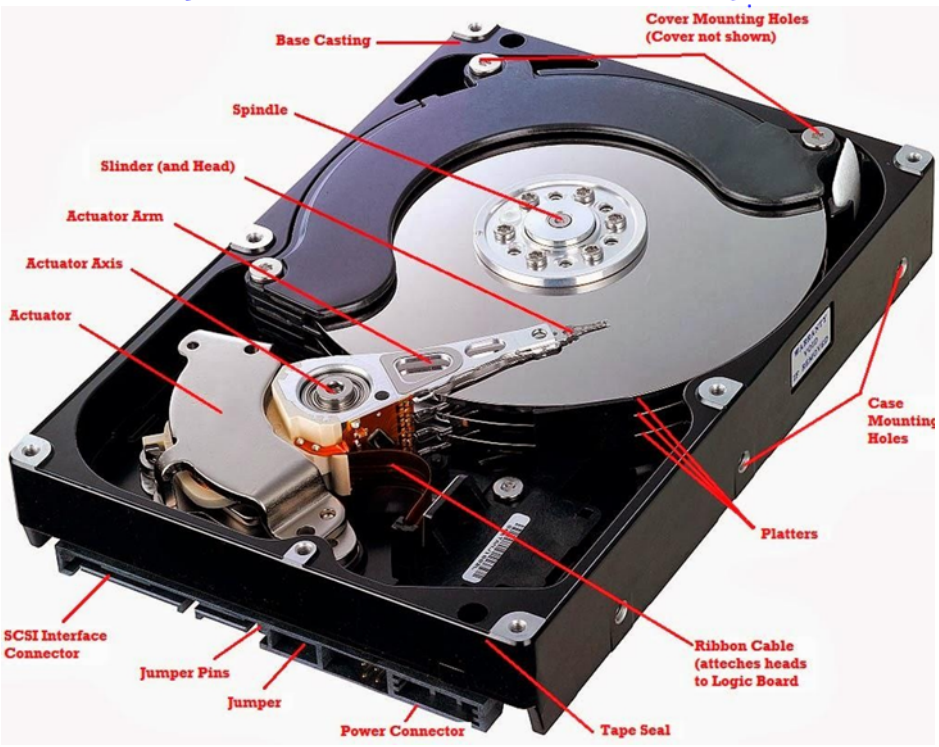
Total rollback: فرآیند را بطور کامل خاتمه می‌دهیم و بعد از اول شروع به اجرا می‌کند

checkpointing. checkpoint نقطه‌ای است که در آن سیستم این است، حالا بیایم هر چند وقت یکبار از سیستم یک checkpoint بگیریم، حالا اگر سیستم باین بست مواجه شد به جای اینکه Total rollback انجام بدهیم و به ابتدای اجرا فرآیند برگردیم می‌توانیم برویم به آخرین checkpoint ای که گرفتیم و از آنجا به بعد کار را تکرار کنیم در این شرایط هزینه Rollback کاهش پیدا می‌کند

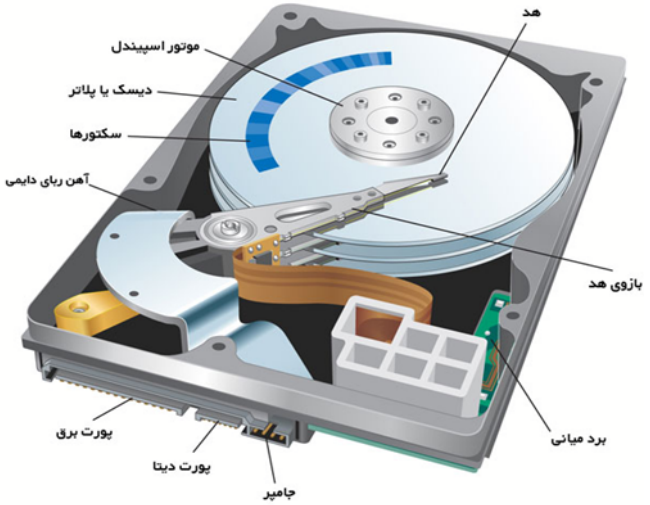
* به هر حال وقتی از total rollback یا checkpointing استفاده می‌کنیم باید مراقب باشیم با گرسنگی (starration) مواجه نشویم، بعنوان مثال فرض کنید سیستم درگیر بن بست شده و ما همیشه از فاکتور هزینه اولویت استفاده می‌کنیم، حالا اگر یک فرآیند کم اولویتی وجود داشته باشد که این فرآیند همیشه بعنوان قربانی استفاده بشود این فرآیند دچار starvation می‌شود. حالا یک راه حل برای حل این مشکل این است که تعداد دفعاتی که فرآیندها rollback می‌کنند را هم بعنوان یک فاکتور هزینه در نظر بگیریم، اینطوری اگر چنین فرآیندی وجود داشته باشد کم کم هزینه Abort کردنش بالا می‌رود

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

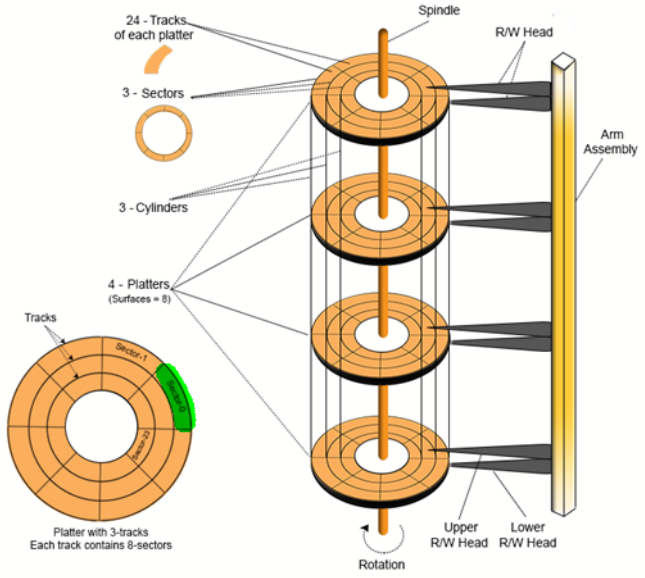
HDD : Hard Disk Drive : هارد یک حافظه مکانیکی است و اطلاعات روی سطح مغناطیسی دیسک ذخیره می‌شود، در هارد با چرخش پلاتر هد خواندن و نوشتن توسط یک بازو مکانیکی به داده مورد نظر دسترسی پیدا می‌کند و اطلاعات توسط هد و با بهره گیری از مغناطیسی خوانده و نوشته می‌شود

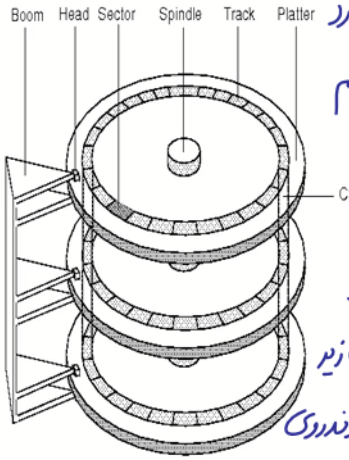


نمایی از هارد دیسک



توجه: واحد خواندن و نوشتن از روی هارد سکتور است و کمتر از آن نمی‌توانیم بخوانیم، اگر رفتیم توی هارد حداقل یک سکتور باید بخوانیم (بیشتر می‌توانیم) حجم اطلاعاتی که در هر سکتور ذخیره می‌شود ثابت و ۵۱۲ بایت است. تعداد سکتورهای هر ترک هارد با آلت، مثلاً بالای ۵۰۰ تا





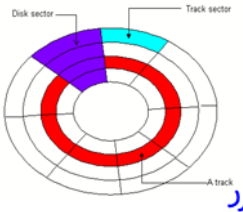
وقتی هدهای دیسک زیر هم هست هر دیسک یک بازویی دارد و یک هد میره زیر دیسک و یک هد میاد روی دیسک چون هم روی دیسک رهم زیرش می‌توانیم بخوانیم و بنویسیم و این بازوها به هم وصل هستند و با هم جلو و عقب می‌روند یعنی هد این ۶ تا هد با هم میرن روی یک شیار، حالا اگر این ۶ تا هد روی شیار ۱۰ ام باشد به همه این شیارهای زیر هم می‌گویند استوانه (cylinder)، پس هدها با هم می‌روند روی یک سیلندر.

نکته) یک بلاک می‌تواند شامل چندین سکتور باشد مثلاً یک بلاک ۸KB شامل ۱۶ سکتور است (یک بلاک می‌تواند یک کلاستری (خوشه) از سکتورها باشد)

نکته) ممکن است جایی بشنوید که تعداد سکتورهای هر ترک ثابت است و ۳۳ تا است اما این برای قدیم و از spindle که فاصله می‌گیریم سکتورها می‌توانند بیشتر شود

نکته) توی تمام این Rom ها عملیات Read به مراتب سریعتر از عملیات write است اینجا ذات تون بیشتر برای Read کردن تا write کردن البته با پیشرفت تکنولوژی مشکلی که در عملیات write دارند داره کم کم حل میشه به همین خاطر هم ما امروزه مثلاً حافظه های SSD رو داریم که به جای Hard disk داره از تون استفاده میشه که به مراتب از هارد که یک چیز مکانیکال است سریعتره، اگرچه باز read تون به مراتب سریعتر از رایت تون هست ولی رایت تون هم خیلی سریع شده

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.



تاخیرهای دیسک:

۱- seek time: زمان جستجو (شیار)، زمان جستجوی استوانه

مدت زمانیکه طول میکشه تا بازوی دیسک حرکت کند و هد متصل به جلوی بازو روی track (شیار) مد نظر قرار بگیرد

مقدار دقیق seek time = زمان لازم برای عبور از هر track x میزان جابجایی بازو + زمان راه اندازه اولیه → احتمال بسیار کم

حداقل " = " " " اینکه از یک track بریم track مجاور

حداکثر " = " " " " " track اول به آخر

۲- Rotational latency

از بعد از اینکه هد رفت روی شیار مربوط بلاکی که می خوایم اون رو بخونیم یا بنویسیم، استارت تایم رو میزنیم تا زمانیکه ابتدای بلاک بچرخه و بیاد زیر هد. min ام این زمان ۰ است و در حالتی است که تا هد روی ترک مد نظر قرار میگیره، ابتدای بلاکی که کارش داریم دقیقا میفته زیر هد و max اش زمان چرخش یک دور دیسک است.

نکته) اگر در خصوص این مورد چیزی نگفتند میانگین این ۲ زمان یعنی زمان نیم دور چرخش را باید در نظر بگیریم

مقدار دقیق تاخیر چرخش = زمان یک دور چرخش کامل x کسری از یک دور که باید طی بشود

۳- transfer time: بعد از آنکه ابتدای بلاک زیر هد قرار گرفت چقدر طول میکشه که بلاک خوانده یا نوشته شود. کل بلاک یا کلاستد یا سکتوری که می خواهیم بخوانیم باید از زیر هد رد بشود که زمان آن هم مثل زمان چرخش محاسبه می شود. مثلاً اگر بلاک شما $\frac{1}{32}$ یک دور باشد، مثلاً شما می خواهید ۲ تا سکتور بخونید وتوی اون track، ۶۰ تا سکتور وجود دارد

زمان انتقال = زمان یک دور چرخش کامل x کسری از سکتورهای درون track که می خوایم بخونیم

نکته) علت اصلی کندی زمان دیسک seek Time هست اش

مثال: سرعت چرخش یک هارد ۶۰۰۰ RPM (۶۰ هزار دور در دقیقه) میانگین Rotational latency را بدست آورید؟

گفته شده یک هارد با سرعت ۶۰ هزار دور در دقیقه می چرخه و باید زمان نیم دور چرخش را حساب کنیم:

هارد در ثانیه چقدر می چرخد = $\frac{۶۰۰۰۰}{۶۰} = ۱۰۰۰$

زمان ۱ دور چرخش چقدره = $\frac{۱}{۱۰۰۰}\ s = ۱ms$

تاخیر چرخش = زمان ۱ دور چرخش x $\frac{۱}{۲}$ = ۰٫۵ms

سوال: فرض کنید هر ترک هارد شامل ۳۲ بلاک است، حالا زمان خواندن ونوشتن روی ۲ بلاک پشت سر هم چقدر است؟ (از زمان جستجو و تاخیر چرخش صرف نظر کنید)

می خواهیم محاسبه زمان انتقال را یاد بگیریم. $T_{trans} = \frac{\text{تعداد بلاکی که می خواهیم بخوانیم یا بنویسیم}}{\text{تعداد بلاک در هر track}} \times$ زمان یک دور چرخش $= \frac{۲}{۳۲} \times ۱ms = ۰٫۰۶۲۵ms$

- با احتساب زمان چرخش خواندن این ۲ بلاک چقدر طول می کشد؟ ۰٫۵ms + ۰٫۰۶۲۵ms = ۰٫۵۶۲۵ms

- فرض کنید می خواهیم از ترک (شیار) ۱۷ به ترک ۴۷ برویم وزمان رفتن از یک ترک به ترک مجاور ۰٫۵ms است، زمان کل چقدر می شود؟

seek time = (۴۷-۱۷) x ۰٫۵ ms = ۱۵ ms

زمان کل = ۱۵ms + ۰٫۵۶۲۵ ms = ۱۵٫۵۶۲۵ ms

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

الگوریتم‌های زمانبندی بازوی هارد

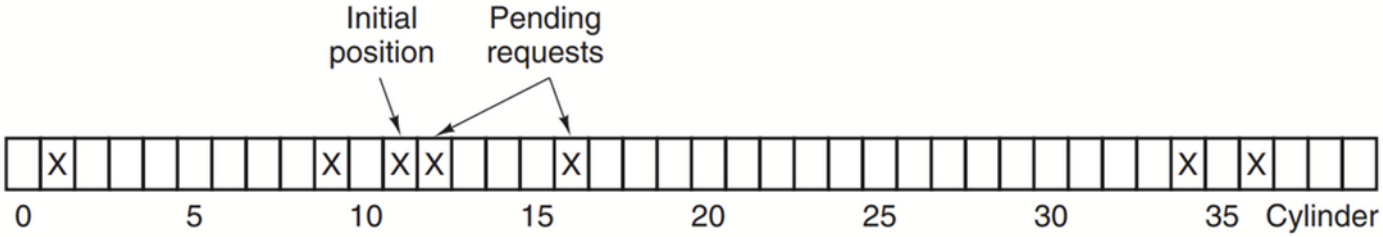
می‌خواهیم زمان جستجو را کم کنیم، چون زمان جستجو گلوگاه ما است

۱- FCFS یا FIFO:

consider an imaginary disk with 40 cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders 1, 36, 16, 34, 9, and 12, in that order.

یک هارد داریم که ۴۰ تا سیلندر دارد، یک درخواست برای خواندن یک بلاک روی سیلندر ۱۱ می‌رسد، حالا تا وقتی می‌خواهیم بدیم روی سیلندر ۱۱ درخواست‌های جدیدی برای سیلندرهای ۱، ۳۶، ۱۶، ۳۴، ۹ و ۱۲ می‌آید، چقدر زمان نیاز است که این سیلندرها را با استفاده از FCFS جستجو کنیم؟ زمان حرکت از هر سیلندر به سیلندر مجاور را ۲ms فرض کنید.

وقتی درخواست برای سیلندر ۱۱ تمام می‌شد درایو هارد به ترتیب زیر به درخواست‌ها سرویس می‌دهد:



۱۱ → ۱ → ۳۶ → ۱۶ → ۳۴ → ۹ → ۱۲

۱۰ ۳۵ ۲۰ ۱۸ ۲۵ ۳

مجموع حرکت: $10+35+20+18+25+3=111$

مجموع جابجایی روی ترک هارد:

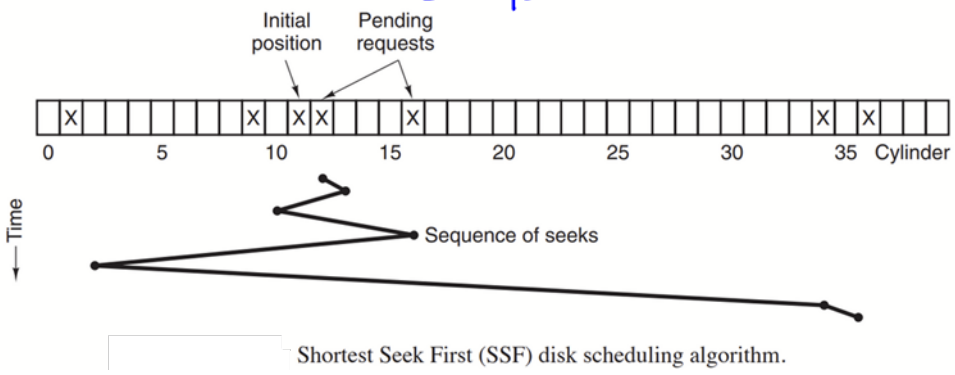
کل زمان جستجو = $111 \times 2ms = 222ms$

میانگین زمان جستجو = $\frac{222ms}{6} = 37ms$

میانگین زمان جستجو در الگوریتم FIFO خیلی بالا است و کارایی پایینی دارد ولی starvation نداریم.

۲- LIFO یا LCFS: عکس بالا

۳- shortest seek(time) first (SSF یا SSTF): همیشه کسی را انتخاب کنیم که کمترین seek time را دارد



Shortest Seek First (SSF) disk scheduling algorithm.

۱۱ → ۱ → ۱۲ → ۳ → ۹ → ۷ → ۱۶ → ۱۵ → ۱ → ۳۳ → ۳۴ → ۲ → ۳۶

مجموع جابجایی روی ترک هارد: $1+3+7+15+33+2=61$

کل زمان جستجو = $61 \times 2 = 122ms$

میانگین زمان جستجو = $\frac{122}{6} = 20.3ms$

* این الگوریتم starvation دارد

توجه کنید که هد اش داره درخواست‌های جدید میاد.

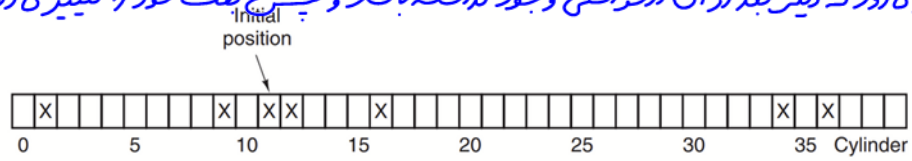
* با این الگوریتم ممکن است دچار چسبندگی شویم ← ممکن است یک یا چند فرآیند (یا ترد) هارد رو در اختیار خودشون بگیرند برای مقاطعی از زمان

* کم کردن زمان جستجو و عدالت با هم در تناقض هستند.

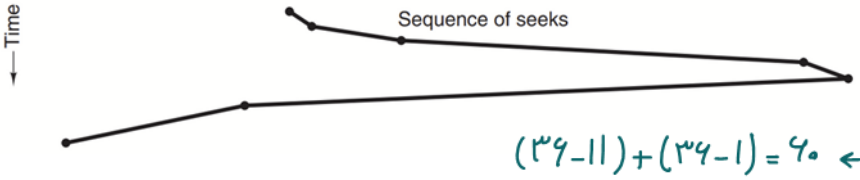
۴- elevator Algorithm یا look:

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

الگوریتم آسانسور برایش مهم نیست که چه درخواستی زودتر آمده و همین طور برایش مهم نیست که بازوی هارد به چه ترکی نزدیک تر است و برای این الگوریتم جهت حرکت است، این الگوریتم جهت حرکت بازوی هارد مهم است، بازوی هارد در یک جهت حرکت می کند و هر درخواستی در جهت حرکت اش وجود داشته باشد را پاسخ می دهد و تا شیاری جلو می رود که دیگر بعد از آن درخواستی وجود نداشته باشد و پس جهت خود را تغییر می دهد و همین کار را نیز در جهت مقابل انجام می دهد.



* حتما باید در سوال جهت حرکت اولیه را مشخص کنند



حرکت به سمت بالا:

$11 \xrightarrow{1} 12 \xrightarrow{4} 16 \xrightarrow{18} 34 \xrightarrow{2} 36 \xrightarrow{27} 9 \xrightarrow{8} 1$

مجموع جابجایی روی ترک ها: $1+4+18+2+27+8=60 \leftarrow (36-11)+(36-1)=60$

کل زمان جستجو = $60 \times 2 = 120\ ms$

میانگین زمان جستجو = $\frac{120}{6} = 20\ ms$

حرکت به سمت پایین: $11 \xrightarrow{2} 9 \xrightarrow{8} 1 \xrightarrow{11} 12 \xrightarrow{4} 16 \xrightarrow{18} 34 \xrightarrow{2} 36 \leftarrow (11-1)+(36-1)=45$

مجموع جابجایی روی ترک ها: $2+8+11+4+18+2=45$

* آسانسور starvation ندارد چون نوسان نمی کند و جهت برایش مهم است، ولی می تواند دچار چسبندگی شود

ماکزیمم زمان پاسخ (انتظار) در آسانسور = یک رفت و برگشت

۵- scan (مرور): برخلاف آسانسور یا look تا آخر می رود و بعد تغییر جهت میدهد و باز تا جهت مخالف تا آخر می رود.

حرکت به سمت بالا: $11 \xrightarrow{1} 12 \xrightarrow{4} 16 \xrightarrow{18} 34 \xrightarrow{2} 36 \xrightarrow{3} 39 \xrightarrow{30} 9 \xrightarrow{8} 1 \rightarrow 0$ ✗ ↓ تا ۰ نمی رویم چون لیست درخواست ها تموم شده، اگر درخواست ها تموم نمی شدند می رفتیم تا ۰ و بعد از ۰ رو به بالا می رویم

حرکت به سمت پایین: $11 \xrightarrow{2} 9 \xrightarrow{8} 1 \xrightarrow{1} 0 \xrightarrow{12} 12 \xrightarrow{4} 16 \xrightarrow{18} 34 \xrightarrow{2} 36 \rightarrow 39$ ✗ $(11-0)+(36-0)=47$

۶- circular elevator یا c_look

ماکزیمم زمان پاسخ (انتظار) در آسانسور = یک رفت (در برگشت فقط یک seek time داریم و در برگشت همه اش برای خواندن داده ها صبر نمی کند و زمان انتقال و زمان چرخش داده هایی که آن وسط هستند را نداریم)

* کاهش حداکثر زمان پاسخ ← چون max را کاهش می دهیم واریانس زمان پاسخ هم کاهش پیدا می کند

حرکت به سمت بالا: $11 \xrightarrow{1} 12 \xrightarrow{4} 16 \xrightarrow{18} 34 \xrightarrow{2} 36 \xrightarrow{35} 1 \xrightarrow{8} 9$ $(36-11)+(36-1)+(9-1)=68$

* تا انتها نمی رویم

حرکت به سمت پایین: $11 \xrightarrow{2} 9 \xrightarrow{8} 1 \xrightarrow{35} 36 \xrightarrow{2} 34 \xrightarrow{18} 16 \xrightarrow{4} 12$ $(11-1)+(36-1)+(36-12)=69$

* starvation ندارد ولی چسبندگی دارد.

۷- c_scan

حرکت به سمت بالا: $11 \xrightarrow{1} 12 \xrightarrow{4} 16 \xrightarrow{18} 34 \xrightarrow{2} 36 \xrightarrow{3} 39 \xrightarrow{39} 0 \xrightarrow{1} 1 \xrightarrow{8} 9$

* تا انتها می رویم * تا ۰ باید بریم

حرکت به سمت پایین: $11 \xrightarrow{2} 9 \xrightarrow{8} 1 \xrightarrow{1} 0 \xrightarrow{39} 39 \xrightarrow{3} 36 \xrightarrow{2} 34 \xrightarrow{18} 16 \xrightarrow{4} 12$

* *

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

۸- F_scan (Fast scan) : برای حل شکل چسبندگی ارائه شده

۲ تا صف دارد و هر صف طبق الگوریتم scan زمانبندی میشه، حالا ابتدا درخواست ها وارد صف اول می‌شوند و بعد که می‌خواد شروع کند به زمانبندی صف اول سایر درخواست‌های ورودی رو می‌فرسته به صف دوم، حالا وقتی پاسخگویی به صف اول تموم شد می‌ریم سراغ صف دوم و سایر درخواست ها از این به بعد به صف اول فرستاده می‌شوند. به این ترتیب مشکل چسبندگی حل می‌شه ولی زمان seek time زیاد میشه، چون وقتی داریم توی یکی از صف ها بصورت scan جلو می‌ریم یکسری درخواست وارد بشوند که در مسیرمان وجود دارد ولی چون ما درخواست‌های جدید رو می‌فرستیم به صف جدید به اونها سرویس نمی‌دیم. در این الگوریتم شکل قطعی هم نداریم.

این الگوریتم درواقع یک مدلی از N step scan است که در اینجا N=۲ است.

track at a time caching

چون بزرگترین زمان در هارد مربوط به seek time است می‌گیم حالا که زحمت کشیدیم و رفتیم روی یک track بیایم و با توجه به اصل مراجعات محلی کل track رو بخونیم.

قبلا می‌خواستیم فقط ۱ سکتور از ۱ ترک بخونیم چقدر طول می‌کشید؟ بطور میانگین نصف زمان چرخش را داریم و زمان خواندن آن سکتور، حالا می‌گیم یک نصفه دیگه هم بذاریم روش و کل سکتورهای یک track رو بخونیم، حالا زمان‌های خواندن سکتورهای دیگه اضافه می‌شه که خیلی زیاد نیست حالا این داده‌ها رو کش کنیم و اگر فرآیندی درخواست اینها رو داد بهش بدیم.

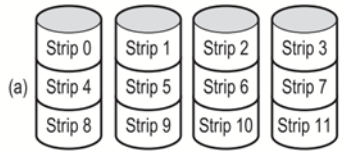
RAID (Redundant Array of Inexpensive/Independent Disks)

به جای اینکه از یک هارد استفاده کنیم بیایم از آرایه‌ای از هاردها استفاده کنیم که در اینصورت بسته به نوع RAID می‌توانیم به یکی یا هر ۲ هدف زیر برسیم :

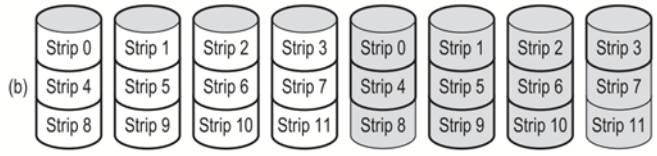
۱- سرعت

۲- قابلیت اطمینان

- Availability، capacity هم می‌تواند از ویژگی‌ها باشد



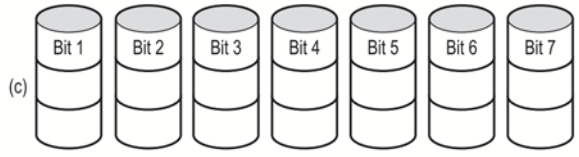
سرعت :

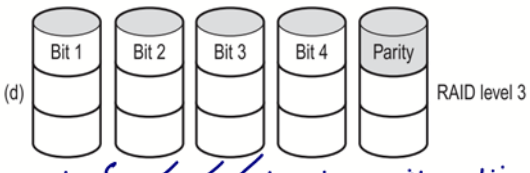


قابلیت اطمینان :

RAID level 0 : وقتی می‌خواهیم یکسری اطلاعات را مثلا ذخیره کنیم بیایم بصورت شکل a این کار را بکنیم، کنترل کننده RAID، سکتورهای 2K را بده به هارد اول و سکتورهای 2K+1 را به هارد دوم و... در اینصورت هرکدام از هارد ها می‌توانند بصورت همزمان کار کنند و سرعت ۲ برابر می‌شود، موقع خواندن هم سوازی می‌خونیم و کنترل کننده خودش ترتیب اش رو درست میکند

این تکنیک striping نام دارد





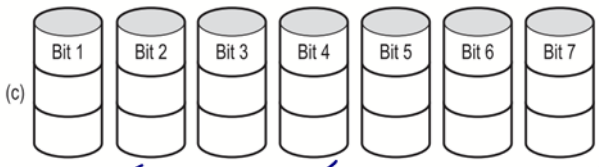
- اگر بخواهیم هر از گاهی و در هر زمان یک سکتور بنویسیم و بخونیم این روش ارزش افزوده‌ای نخواهد داشت و فرقی نمی‌کند که کدام هارد این کار را انجام بدهیم ولی وقتی داده‌ها بصورت بلاک‌های بزرگ باشند و شامل چندین سکتور باشد این روش بهترین عملکرد را دارد.

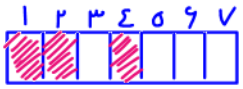
RAID level 1: Mirroring ایجاد کنیم : هدف RAID سطح یک افزایش سرعت نیست و افزایش قابلیت اطمینان است.

در این روش افزونگی (Redundancy) صد در صد داریم.

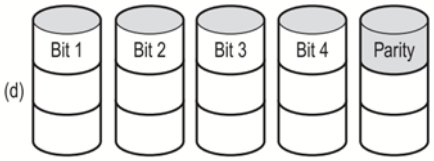
RAID level 2 : افزایش سرعت + قابلیت اطمینان

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.





در روش همینگ از بیت‌های توان ۲ بعنوان افزونگی استفاده می‌کنیم و داده‌های اصلی در سایر خانه‌ها ذخیره می‌شود. برای ذخیره هر بایت، اونها رو به nibble (٤ بیت) تقسیم می‌کنیم و ۷ تا هارد داریم و حالا هر ٤ بیتی که می‌خواهیم ذخیره کنیم این ٤ بیت را در هاردهای ۳ و ٥ و ٦ و ۷ ذخیره می‌کنیم، در این حالت سرعت خواندن و نوشتن ٤ برابر می‌شود، ۳ بیت افزونه را هم برای این ٤ بیت بدست می‌یاریم و توی هاردهای ۱ و ۲ و ٤ ذخیره می‌کنیم، پس ما ۷٥٪ افزونگی داریم در این روش، حالا اگر یکی از هاردها خراب بشود با استفاده از ۳ هارد دیگر می‌توانیم داده‌های آن هارد را بازیابی می‌کنیم.



RAID level 3: برای ذخیره داده‌های ٤ بیتی فقط ۱ بیت parity بعنوان افزونگی می‌زارد، بنابراین در این روش افزونگی ۲٥٪ خواهد بود. حالا اگر یکی از ۳ هارد خراب بشود می‌توانیم اطلاعات آن را بازیابی کنیم.

توجه کنید که ما در شبکه به این دلیل از کد همینگ استفاده می‌کنیم چون نمی‌دانیم کدام بیت خراب است، به همین دلیل در همینگ ۳ بیت اضافه کردیم که اول متوجه بشویم کدام بیت خراب است که بعد بتوانیم آن را تصحیح کنیم. ولی در اینجا اگر یک هارد خراب شود خب ما می‌فهمیم کدام خراب است

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

operating system structure

مهمترین معماری های OS :

۱- simple structure

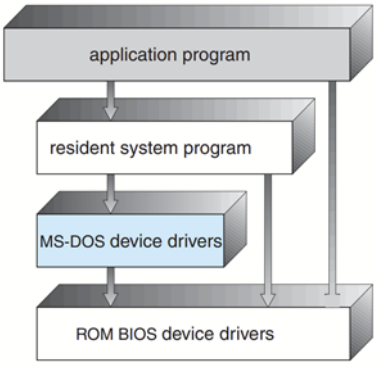
۲- layered approach

۳- (μ-Kernel) MicroKernels

۴- Modules

۵- Virtual Machines

simple structure



MS-DOS layer structure.

روی OS ، Ms-Dos صحبت میکنم (microsoft Disk os) که مایکروسافت در اوافر دهه ۹۰ میلادی ارائه کردش یک OS تک کاربره و single task بود و فقط می توانست در هر زمان یک برنامه را اجرا بکند و در یک شرایط خاص می توانست برنامه دومی بنام resident system program را اجرا بکند، وقتی این OS می خواست بیاد با ۲۱ اول از درایورهای پایه دیوایس های مانیتور و کیبورد و چندتا دیوایس اساسی که در Rom بود را استفاده می کرد و فعال شون می کرد و حالا اگر نیاز بود کارت شبکه و کارت صدا و... در سیستم فعال بشود باید خودمون درایورهای مربوط به آنها را خودمون اضافه می کردیم (Ms-Dos device driver). در این معماری ما محافظت نداریم و برنامه ها که در بالاترین سطح هستند می توانند به Driver ها هستند هم دسترسی داشته باشند و یک برنامه می توانست با دیوایس های I/O ارتباط برقرار کنند، اما در حال حاضر برنامه ها نمی توانند مستقیم با دیوایس های I/O کار کنند و دستورات I/O ، privilege هستند و فقط OS می تواند دستورات I/O را اجرا کند.

حالا برای چی می گفتند به حفاظت نیاز نداریم؟ می گفتند این OS تک کاربره و single task است و مالتی تسک نیست و نیازی به محافظت نداریم و بنابراین برای افزایش کارایی کاری می کنیم که برنامه ها بتوانند بدون واسطه به سطوح پایین دسترسی داشته باشند.

خب برنامه ها نباید به حافظه OS دسترسی داشته باشند چون این کار باعث خرابکاری میشه، فکر کنید بیریم به خونه ی حافظه چی می شد؟

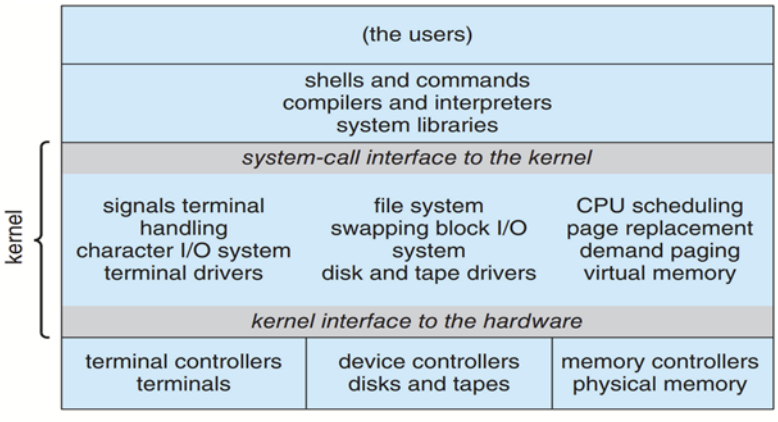
سیستم ریست می شد و مراحل بوت شدن OS دوباره انجام می شد.

این معماری با کمترین سربار پیاده سازی می شود و بنابراین کارایی بالایی دارد

Unix partially layered

سیستم های یکپارچه (Monolithic)

Monolithic : OS بصورت مجموعه ای از توابع و... نوشته شده که هرکدام درصورت نیاز می تواند دیگری را فراخوانی کند، پس توابع و یا فایل های شامل توابع کامپایل می شوند و با linker بهم متصل می شوند و کرنل بصورت یک object یکتا در می آید.



Traditional UNIX system structure.

شبیه همان چیزی است که قبلا در خصوص مکان قرارگیری OS گفتم

لایه کرنل از یک سمت با واسط سیستم کال ها با یوزرها و برنامه ها و از یک سمت دیگه با سخت افزار در ارتباط است.

بسیاری از سرویس ها درون کرنل وجود دارد.

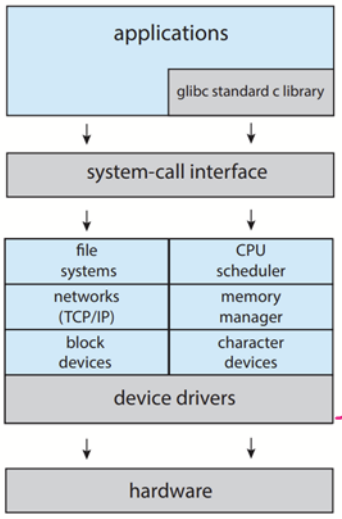
ما می توانیم با نوشتن درون رجیسترهای سخت افزار از آنها استفاده کنیم

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

که این کار توسط درایورها انجام می‌شود. روی درایورها می‌تواند سرویس‌های سطح بالاتری وجود داشته باشد، مثلاً ما می‌دونیم که واحد دسترسی به دیسک بلاک است که در یک سطح بالاتر می‌توان عملیات Block I/O را داشته باشیم و باز در یک نگاه سطح بالاتر file system را می‌توانیم داشته باشیم.

شاید فکر کنید درایورها در پایین‌ترین سطح قرار دارند و هرچه بالاتر می‌رویم سطح سرویس‌ها بالاتر می‌رود ولی در این معماری هم سرویس‌ها در مد کرنل اجرا می‌شوند و هیچ اولویت و تقدمی بین این سرویس‌ها وجود ندارد و اگر در یکی از این سرویس‌ها اختلالی به وجود بیاید چون در مد کرنل هستیم و تمامی اختیارات در اختیار سرویس‌ها وجود دارد آن اختلال می‌تواند منجر به خراب شدن os و کرش کردن سیستم بشود، مثلاً اگر در یکی از درایورها مشکلی وجود داشته باشد این مشکل می‌تواند به هم کرنل منتقل شود و می‌تواند کل os را دچار اختلال بکند

در این معماری ما برخلاف معماری قبل حفاظت داریم و دسترسات و کارهای زیادی در انحصار os است و برنامه‌ها، کامپایلرها، shell و یوزرها فقط از طریق سیستم کال می‌توانند از خدمات کرنل استفاده کنند.



Linux system Architecture

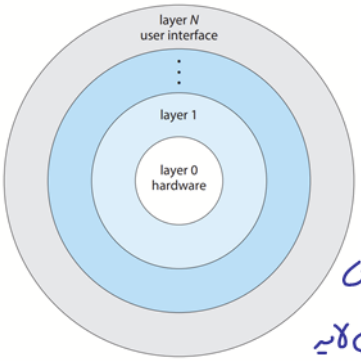
لینوکس یک سیستم عامل unix based است و بنابراین معماری‌اش شبیه معماری قبلی است

می‌دانیم که سیستم کال‌ها وقت‌گیرند و در ساختار یکپارچه برای یک درخواست فقط یک سیستم کال داریم و بنابراین ساختار یکپارچه باعث می‌شود که سریعترین os‌ها به وجود بیاید.

- قابلیت نگهداری پایین، توسعه سخت، قابلیت اطمینان پایین، دیباگ سخت، پیچیدگی بالا

layered Approach (معماری کاملا لایه‌ای)

در این معماری پایین‌ترین لایه سخت‌افزار و بالاترین لایه user interface است و یوزرها از لایه N ام استفاده می‌کنند. در این معماری هر لایه‌ای که نیاز به یک سرویس دارد فقط می‌تواند از لایه پایین‌تر خودش آن سرویس را دریافت کند و بنابراین هر لایه نیز به لایه بالاتر از خودش سرویس می‌دهد.



فرض کنید که لایه N ام از طریق interface‌ای که بین دو لایه N و N-1 وجود دارد یک سرویس را از لایه N-1 می‌خواهد، حال لایه N-1 اگر برای دادن این سرویس به سرویس لایه N-2 نیاز داشت باید از طریق interface‌ای که بین دو لایه N-1 و N-2 وجود دارد اقدام به این کار کند و بنابراین امکان استفاده از خدمات و تعامل از بالاترین لایه تا پایین‌ترین لایه وجود دارد.

معایب این معماری:

تعداد زیاد لایه‌هایی که باید برای ارائه خدمات و تعامل طی بشود که باعث افت کارایی می‌شود

تفکیک لایه‌ها با توجه به سرویس‌ها و اینکه انتظار داشته باشیم هر لایه فقط از لایه پایین‌تر از خودش سرویس بگیرد کار سختی است

مثال: Memory manager برای اینکه از هارد به عنوان حافظه مجازی استفاده کند نیاز دارد که از Disk Device Driver استفاده کند، پس باید لایه‌ای بالاتر از Disk Device Driver، حالا DDD گاهی اوقات نیاز به سرویس cpu scheduler دارد، وقتی کار I/O ای که می‌خواهیم انجام بدهیم کند است و نمی‌توان از cpu خوب استفاده کرد و cpu بیکار می‌شود، باید فرآیند فعلی را بلاک کنیم و فرآیند دیگری توسط

Memory manager
Disk Device Driver
cpu scheduler

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

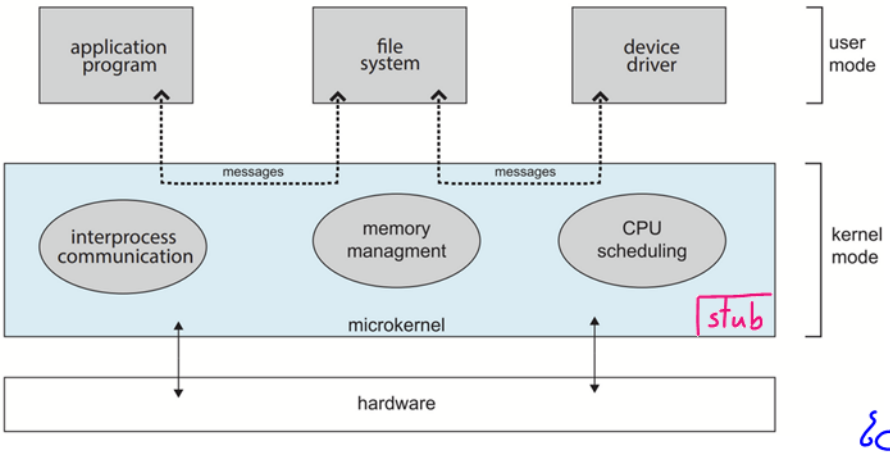
cpu scheduler به cpu داده شود، حالا وقتی DDD می‌فهمد که عملیات I/O ای که می‌خواهد انجام بدهد کند است نیاز دارد از سرویس
" استفاده کند، پس لایه‌بندی ما بصورت بالا می‌شود. از طرف دیگر cpu scheduler گاهی اوقات نیاز دارد که
موقتا یک فرآیند را suspend کند که در این حالت نیاز هست که سرویس DDD استفاده کند که یک فرآیندی که توی رم هست را بفرستیم
روی هارد (virtual memory)، حالا لایه‌بندی ما بهم ریخت و یک سیکل به وجود آمد و قانون این ساختار که هر لایه باید از سرویس
های لایه زیرخودش استفاده کند نقض شد، پس تنها کاری که می‌توان در این شرایط انجام داد این است که باید این سرویس‌ها را در
یک لایه قرار بدهیم که در اینصورت لایه بزرگتری ایجاد می‌شود. این راستان به ثدت در همه جا تکرار بشد و لایه‌های بزرگ می‌شوند
پس پیاده سازی این مدل خیلی عملی نیست و این مدل یک مدل تئوری است و خیلی نمی‌توان os‌ای پیدا کرد که بصورت کاملا لایه‌ای پیاده
سازی شده باشد، چون در این معماری لایه‌ها بزرگ می‌شود بعد از ارائه این مدل رفتند به سمت مدل نیمه لایه‌ای شبیه linux و unix

مزایا:

ساخت این os ساده است

Debugging " " " "، می‌دانیم جای هر سرویس کجاست و بنابراین هر سرویسی که دچار مشکل شد می‌توانیم براحتی مشکلات اش را رفع کنیم

MicroKernels (ریزهسته) QNX, L4, windows NT



بیایم هسته را خیلی ریز کنیم به این نحو که بیایم و تمای اجزا غیرضروری هسته را به بیرون از آن منتقل کنیم و آن اجزا و سرویس‌ها را بصورت سرور در بیرون کرنل و در لایه یوزر قرار دهیم.
پس فرآیندهایی که نقش سرور را دارند در لایه یوزر و در کنار سایر فرآیندهای این سطح قرار می‌گیرند و برنامه‌ها وقتی به این سرویس‌ها نیاز دارند به جای اینکه این سرویس‌ها را مثل قبل از طریق سیستم کال از کرنل بگیرند، این سرویس‌ها را از طریق یک ارتباط client server‌ای از فرآیند دیگری که هم سطح خودشان است و در لایه کاربر اجرا می‌شوند دریافت می‌کنند.

مثلاً file system که در معماری یکپارچه جزو یکی از سرویس‌های درون کرنل بود و با استفاده از یکی سیستم کال open، close، read و write از آن استفاده می‌شود بعنوان یک فرآیند در سطح کاربر قرار دارد و درون کرنل نیست و وقتی یک برنامه می‌خواهد از file system استفاده کند از طریق میکروکرنل یک message به فایل سیستم ارسال می‌کند و فایل سیستم هم پاسخ این فرآیند را از طریق میکروکرنل به برنامه می‌دهد که در این حالت برنامه نقش کلاینت و فایل سیستم نقش سرور را دارد.

چه سرویس‌هایی را می‌توانیم به خارج از MicroKernel منتقل کنیم؟

سرویس‌هایی که لازم نیست حتما در مد کرنل اجرا بشوند را می‌توانیم از میکروکرنل خارج کنیم و در نتیجه یک حداقلی از مدیریت حافظه و زمانبندی و همین طور پیاده سازی Ipc (امکان ارتباط بین کلاینت‌ها و سرورها) درون میکروکرنل باقی می‌ماند، Ipc گلوگاه این معماری است چون تعداد بالایی Ipc داریم در این معماری، در معماری یکپارچه برنامه وقتی سیستم کال می‌زد کرنل پاسخ آن را می‌داد ولی در اینجا وقتی برنامه درخواستی می‌دهد یکبار برای ارسال پیام به سرور به مد کرنل می‌رویم، حالا باز باید به مد یوزر برگردیم تا پیام را به سرور برسانیم و

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

پس برای دادن پاسخ به برنامه باید یکبار دیگر به مد کرنل و پس به مد یوزر رفت.
حالا اگر درایورها در مد یوزر باشد ممکن است فایل سیستم برای انجام کارش به آن نیاز داشته باشد که در اینصورت فایل سیستم هم باید مجددا از طریق Ipc با درایورها ارتباط برقرار کند، درصورتیکه قبلا درخواست‌مان با یک سیستم کال انجام می‌شد و تمام کارهای مربوط به آن درخواست در کرنل انجام می‌شد و پاسخ به برنامه برگردانده می‌شد.

چرا در معماری میکروکرنل File system می‌تواند به مد یوزر منتقل بشود؟

برخی از کارهای فایل سیستم عملیات Accounting است، مثلا کارهایی مثل مشخص کردن اینکه هر فایل با چه مجوزی قابل دسترسی است یا چه برنامه‌هایی مجاز هستند که به فایل x دسترسی داشته باشد و چه برنامه‌هایی مجاز نیستند. این دسترسی‌هایی که برای مشخص کردن این موارد باید اجرا بشوند دسترسی ممتاز نیستند و بنابراین نیازی نیست در مد کرنل اجرا بشوند.
در خصوص درایورها می‌دانیم که درایورها نقش مترجم را بین os و سخت افزار ایفا می‌کند و وقتی برنامه‌ها می‌خواهند با یک دیوایس کار کنند یعنی درون رجیسترهای کنترلی شون بنویسیم درایور فقط نقش مترجم را ایفا می‌کند و این ترجمه نیازی نیست در مد کرنل انجام بشود و دستوراتی که کار ترجمه را انجام می‌دهند دستورات ممتاز نیستند، حواسمان است که درایورها بعد از انجام عمل ترجمه می‌خواهند در رجیسترهای کنترلر دیوایس‌ها بنویسند که این کار، کار I/O است و این کار باید در مد کرنل اجرا بشود، به همین دلیل ما می‌توانیم درایورها را بصورت فرآیندهای سطح کاربر استفاده کنیم ولی قسمتی از درایورها باید در میکروکرنل باشد که اصطلاحا به آن stub می‌گویند و این stub‌ها هستند که درون رجیسترهای کنترلرهای دیوایس‌ها می‌نویسند
پس اگر درایور بعد از انجام عمل ترجمه خواست با دیوایس I/O ارتباط برقرار کند باید از طریق سیستم کال با stub ارتباط برقرار کند.

مزایا:

۱- به واسطه اینکه بسیاری از سرویس‌ها می‌توانند بصورت سرور در بیایند و در مد یوزر اجرا بشوند، درنتیجه سایز کرنل کوچک می‌شود
۲- اگر به سرویس نیاز نداشته باشیم می‌توانیم آن سرویس را اجرا نکنیم و بنابراین این سرویس‌ها از منابع سیستم استفاده نمی‌کنند، در سیستم‌های عام منظوره معمولا نیاز هست که تمام سرویس‌ها فعال باشند، پس از حجم os نیز کم می‌شود
۳- قابلیت پیکربندی بالایی دارند (configurable)
۴- این معماری برای سیستم‌هایی که با محدودیت منابع و حافظه مواجه هستند مثل سیستم‌های هفتد بسیار مناسب است.
۵- یادمان هست که در معماری یکپارچه اگر مشکلی برای یکی از سرویس‌های درون کرنل رخ میداد، این اختلال می‌توانست به تمامی سرویس‌های کرنل سرایت پیدا کند و این امکان وجود دارد os درست کار نکند یا crash کند، درصورتیکه در این معماری اگر یک سرور crash کند سایر سرویس‌ها و سرورها فعال هستند و گفتیم ما یک فرآیند سطح کاربری را از دست دادیم که با اجرا مجددا آن فرآیند می‌توانیم مجددا آن سرویس‌مان را توی سیستم داشته باشیم (reliability)
۶- os توسعه پذیر می‌شود: اگر به سرویس نیاز داریم که os آن را ارائه نمی‌کند کافی است یک فرآیند سطح کاربری برای اضافه شدن سرویس مدنظرمان اجرا کنیم

حالا سایر برنامه‌ها چطور می‌توانند از این سرور (فرآیند) استفاده کنند؟ چون برنامه‌ها برای استفاده از آن سرور (فرآیند) باید آدرس آن فرآیند را داشته باشند

راه۱) در کرنل یک جدول داشته باشیم و os آدرس سرورها را نگه دارد. حالا وقتی یک فرآیند درخواست ارتباط با یک

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

سرویسی را می‌دهد، os می‌داند که سرورها کجا هستند و با آن سرورها ارتباط برقرار می‌کند

راه۲) شبیه DNS : یک سرور بنام Registration server در مد کاربر داشته باشیم و وقتی سرور جدیدی می‌خواهد به سیستم اضافه بشود ابتدا برود و اطلاعات خودش را در این سرور ثبت کند، حالا وقتی یک برنامه نیاز داشت بفهمد که سرویس x روی چه آدرسی ارائه می‌شود می‌تواند این سوال را با Ipc از Registration server بپرسد.

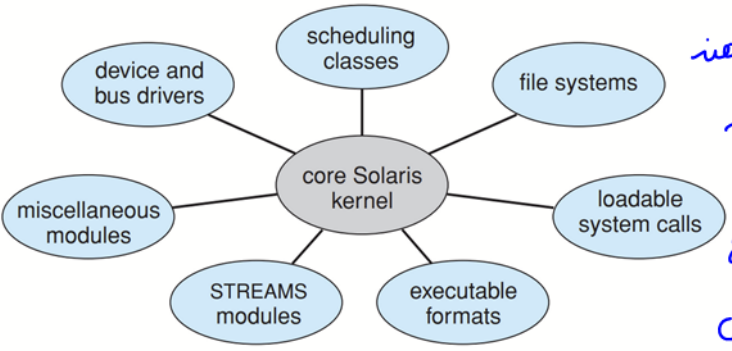
۷- portability : سرویس‌ها portable می‌شوند. چون تعداد سرویس‌های درون میکروکرنل کم می‌شود پس تعداد سیستم کال‌هایی که برای ارتباط با میکروکرنل نیاز داریم محدودتر می‌شود و بنابراین می‌توان براحتی این سیستم کال‌ها را استاندارد کنیم و بنابراین برنامه‌ها ساده‌تر می‌توانند روی os‌های میکروکرنل منتقل کرد

۸- فرآیندها می‌توانند از سرویس‌های روی سیستم‌های دیگر هم استفاده کنند (Distributed service)، سرویس‌ها را می‌توانیم روی سیستم‌های مختلف توزیع کنیم Ip+port

۹- افزایش امنیت (security) : در اینجا مثل معماری یکپارچه همه سرویس‌ها کنار هم توی کرنل نیستند و وقتی یک برنامه با سرور x کار دارد در مد کرنل به اطلاعات سایر سرویس‌ها دسترسی ندارد، توی معماری یکپارچه هر سرویس بخواهید، آن سرویس در مد کرنل به فرآیند داده می‌شود، حالا اگر اون سرویس ضعف امنیتی داشته باشد یا دچار اختلال بشه، ممکن است فرآیند بتواند به سایر جاهای کرنل هم دسترسی پیدا کند، اما در معماری میکروکرنل هر سرور یک فرآیند سطح کاربر است بنابراین حافظه‌اش نسبت به سایر فرآیندها محافظت شده است بنابراین با ارتباط با یک سرور نمی‌توان به اطلاعات سایر سرورها دسترسی پیدا کرد.

نکته) هیچ کدام از os‌های جدید از این معماری استفاده نمی‌کنند چون کارایی کم می‌شود و الان به علت کارایی به سمت معماری یکپارچه برگشتند در حال حاضر ویندوزهای جدید بصورت ترکیبی از معماری‌ها استفاده می‌کنند

Modules (معماری پیمانه‌ای)



Solaris loadable modules.

os یکری کامپونت‌های core دارد که این core‌ها کرنل ما را تشکیل می‌دهند و ماژول‌هایی که سرویس‌های دیگری برای os فراهم می‌کنند بصورت داینامیک به کامپونت‌های core وصل می‌شوند و سرویس‌های خودشان را اضافه می‌کنند و ما با ماژول‌هایی که اضافه می‌کنیم می‌توانیم تنوعی از زمانبندها، فایل سیستم‌ها و... را داشته باشیم، این ماژول‌ها می‌توانند در زمان بوت سیستم و یا در حال اجرا به کامپونت‌های core متصل می‌شوند که این موضوع به این بستگی دارد که آن ماژول چقدر تغییرات می‌خواهد در os و یا بقیه ماژول‌ها ایجاد کند

- مثل OOP از تفکر ماژولار استفاده می‌کنیم
- این معماری وارث از هر ۲ تفکر Kernel, layered و M-Kernel استفاده می‌کند. چون هر ماژول یک interface محافظت شده برای خودش دارد و از طریق این interface می‌توان از سرویس هر ماژول استفاده کرد دارم از تفکر لایه‌ای استفاده می‌کنیم و چون می‌توانیم با اضافه کردن ماژول به core‌ها سرویس اضافی ایجاد کنیم از تفکر M-Kernel استفاده شده و اینکه برخلاف M-Kernel از Ipc استفاده نمی‌کنیم و مستقیما با ماژول‌ها ارتباط برقرار می‌کنیم باز داریم از تفکر لایه‌ای بهره می‌بریم، توجه کنید که در این معماری برخلاف M-Kernel تمامی ماژول‌ها در مد کرنل اجرا می‌شوند

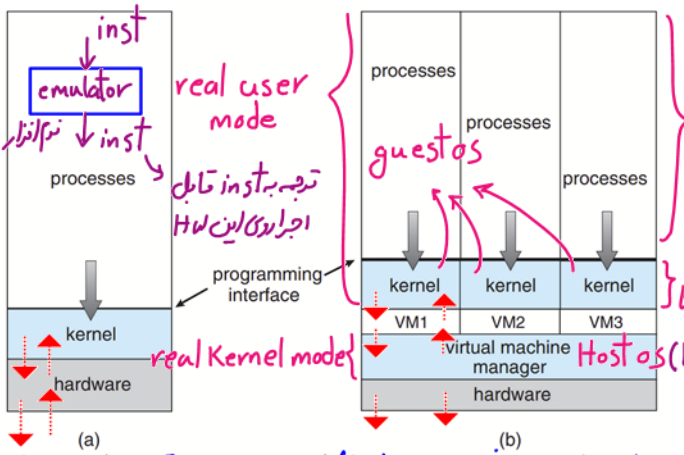
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

Virtual Machines می‌خواهیم امکانات سخت افزار به یک os محدود نشود

یک لایه نرم افزاری به اسم virtual machin manager یا hypervisor روی سخت افزار وجود دارد که به لایه بالا تر از خودش تعدادی VM ارائه می‌کند که روی این VM ها می‌توانیم os نصب کنیم و فرآیندهای روی آن os ها اجرا شوند.

هرکدام از VM ها برای یک os به شکل یک ماشین کامل تصور می‌شوند

به هرکدام از VM ها قسمتی از منابع سخت افزار داده می‌شود و منابع هرکدام از VM ها نسبت به منابع سایر VM ایزوله است، اینکه به هر VM چه بخشی از سخت افزار فیزیکی قرار داده بشود به تنظیمات Hypervisor برمی‌گردد. (به جز VM هیچ چند دیگری روی Hypervisor نداریم، مثلاً ESX روی سخت افزار می‌نشیند و فقط VM می‌دهد، درواقع Hypervisor یک os است ولی نقش کاربری خاص دارد و فقط یکسری VM می‌سازد و یکسری interface ویژه ارائه می‌کند)



در os های Multitask، os باید حواسش به حفاظت باشد، بعنوان مثال باید حواسش باشد که فرآیندها به حافظه همدیگر و حافظه os دسترسی نداشته باشند ولی در معماری Virtual Machin چون VM ها نسبت به هم ایزوله هستند، بنابراین کرنل یک VM و فرآیندهای روی هر VM نگران این نیستند که دارند منابع را بصورت مشترک با بقیه VM ها استفاده می‌کنند

real Kernel mode ـ باید برای real Kernel mode، virtual Kernel mode، virtual user mode این تصور را ایجاد کنند که این‌ها real user mode هستند.

ـ توجه کنید که برای اینکه بتوانیم این معماری را داشته باشیم باید سخت افزار این موضوع را ساپورت کند، درواقع ما قبلا با استفاده از یک بیت mode bit دستورات را به ۲ دسته دستورات عادی و ممتاز تقسیم می‌کردیم که فرآیندهای سطح کاربر فقط حق استفاده از دستورات عادی را داشتند ولی حالا در این معماری ما باید بتوانیم بین اختیارات و دسترسی‌های virtual Machin manager و اختیارات و دسترسی‌های کرنل روی هر VM، و همچنین بین اختیارات و دسترسی‌های فرآیندها (virtual user mode) تمایز قائل شویم، پس به بیشتر از ۲ مد نیاز داریم و سخت افزار باید این موضوع را ساپورت کند.

توجه) Hypervisor، یک os است که می‌تواند هر یک از معماری‌های Micro-Kernel، layered، یکپارچه و... را داشته باشد

os ها باید بصورت native برای cpu کامپایل شده باشند، یعنی os Host، os Guest ها هم‌شون برای یک ماشین کامپایل شده باشند

مزایای معماری Virtual Machin

۱ـ چون VM ها کاملا نسبت به هم ایزوله هستند درگیر مسائل مربوط به حفاظت نیستیم

۲ـ هر VM منابع خودش را دارد که نسبت به منابع سایر VM ها ایزوله است و منابع توسط VM ها بصورت مستقیم، بصورت مشترک استفاده نمی‌شوند که این موضوع باعث می‌شود که مدیریت سیستم از نگاه هرکدام از os های روی یک VM بسیار ساده‌تر شود.

۳ـ معماری Virtual Machine برای توسعه os مناسب است، اگر این معماری را نداشته باشیم و بخواهیم os را توسعه بدهیم و تغییراتی روش انجام بدهیم این خطر وجود دارد که os خراب بشه و دیگه سیستم بالا نیاد که در اینصورت باید دوباره روی سیستم os نصب کنیم همین طور برای توسعه کرنل نیاز بررسی محتوای حافظه کرنل در لحظات مختلف و track خطوط برنامه کرنل (اینکه هر خط برنامه چه تاثیری روی حافظه و متغیرها می‌گذارد) ساده نیست و نیاز به سخت افزار کمکی دارد (باید یک سیستم دیگه رو از طریق پورت JTAG به سیستم خودمان

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

وصل کنیم و با استفاده از break point کد کرنل را تیکه تیکه اجرا کنیم تا بتوانیم محتوای حافظه کرنل را روی سیستم جانبی مشاهده کنیم)
حالا استفاده از معماری Virtual Machine کار توسعه os رو خیلی راحت تر می‌کند چون در این معماری ما یک لایه نرم افزاری (Hypervisor) زیر ماشین‌هایی که داریم کرنل شون رو توسعه می دهیم داریم که با داشتن این لایه نرم افزاری اولا اگر os خراب بشود نهایتا یک VM رو از دست می دهیم که خیلی مشکل جاری نیست و می‌توانیم مجددا با استفاده از Hypervisor یک VM بسازیم و مجددا os مدنظرمان را روی آن بالا بیاریم دوما می‌توانیم محتوای حافظه VM مربوطه در لحظات مختلف را مشاهده کنیم و خطوط برنامه کرنلی که روی آن VM سوار است را track کنیم

نکته) در معماری Virtual Machin برخی از زمان‌ها می‌توانند با زمان‌های real system متفاوت باشند
مثلاً در یک os عادی یک عمل I/O (real I/O) می‌تواند در ۵۰ms انجام بشود ولی یک عمل virtual I/O که روی یکی از VM‌های این سیستم انجام می‌شود ممکن است گاهی اوقات کمتر (به علت spool) و گاهی اوقات بیشتر از (به علت منقطع شدن) ۵۰ms طول می‌کشد
- فرض کنید در شکل (a) می‌خواهید یکبری داده را روی کارت شبکه واقعی بفرستید و یا دریافت کنید، حالا فرض کنید این کار ۵۰ms طول می‌کشد حالا این کار در معماری virtual machine چگونه است؟

مثلاً VM_1 را در نظر بگیرید، در اینجا کرنل داده را به کارت شبکه مجازی ارسال می‌کند و همین طور داده‌ها را از طریق همین کارت شبکه مجازی دریافت می‌کند. فرض کنید VM_1 منتظر داده‌ای است که این بسته هنوز به دست سخت افزار واقعی نیز نرسیده و وقتی به سخت واقعی می‌رسد باید از چند لایه بگذرد تا به دست فرآیند مربوط روی VM_1 برسد که در این حالت می‌تواند بیشتر از ۵۰ms طول بکشد
اگر داده‌ای به سخت افزار واقعی برسد، این داده از روی کارت شبکه واقعی برداشته می‌شود و در حافظه Hypervisor قرار داده می‌شود و بعد از این است که این داده به VM_1 داده می‌شود که در این حالت داریم داده را از حافظه Hypervisor به حافظه‌ای در VM_1 منتقل می‌کنیم که این کار می‌تواند در کمتر از ۵۰ms انجام بشود، همین طور وقتی VM_1 می‌خواهد داده‌ای ارسال کند آن را از روی حافظه خودش به حافظه Hypervisor ارسال می‌کند که این کار می‌تواند زمان کمتری از ۵۰ms ببرد

Emulation (تقلید)

گاهی اوقات ما یک os‌ای یا یک نرم افزاری داریم که بصورت native کامپایل شده است، مثلاً نرم افزاری داریم که برای یک cpu ـی ARM کامپایل شده، ولی cpu ای که ما داریم یک cpu ـی اینتل است، حالا ما می‌خواهیم برنامه‌ای که برای پردازنده دیگری کامپایل شده را روی پردازنده اینتل اجرا کنیم، حالا برای انجام این کار ما نیاز داریم یک نرم افزار برای ما پردازنده ARM را تقلید کند، یعنی یک نرم افزار emulator روی سیستم می‌کنیم که این نرم افزار میاد و inst به inst دستورات پردازنده ARM را به inst‌های معادل پردازنده اینتل تبدیل می‌کند. خب به این دلیل برنامه خیلی کندتر اجرا میشه

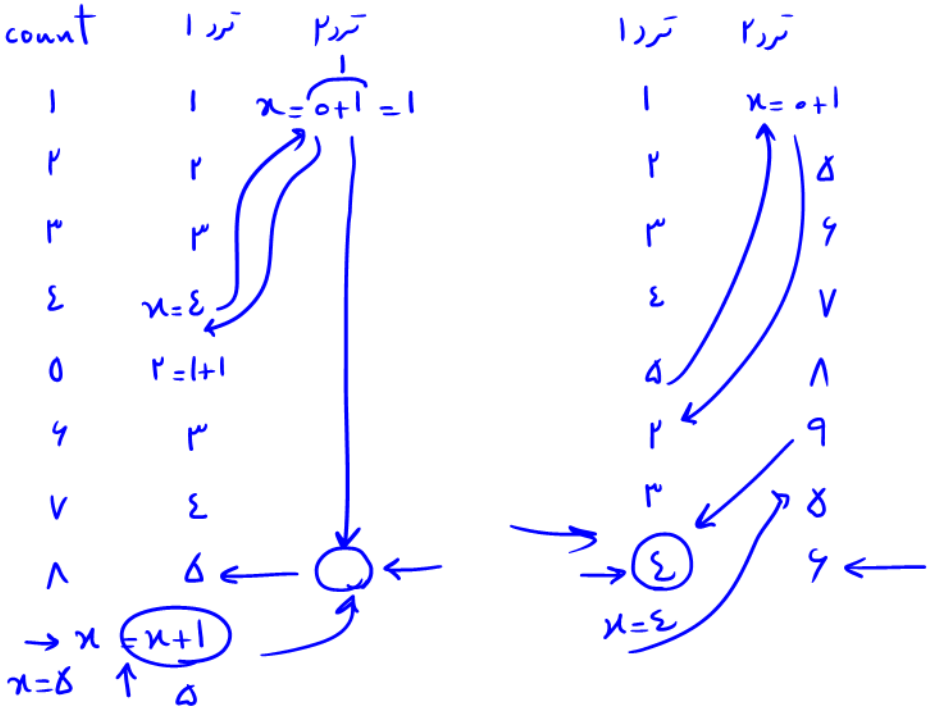
سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

1. برنامه زیر را در نظر بگیرید و دقیقا مشخص نمایید چه روندی می‌بایست دنبال گردد تا جفت خروجی‌های زیر حاصل شوند: 8و8؛ 5و5؛ 6و4.

```
const int n=8;
int x;
void f() {
        int count;
        for (count=1; count<=n; count++) x++;
        printf("%d\n",x); }
void main() {
        x=0;
        createthread(&f);
        createthread(&f);
        while (1) {...}
        return; }
```

توجه) توجه کنید که متغیر x سراسری است و بین تردهای شیر است و متغیر count، local است و هر ترد متغیر count خودش را دارد

تولید ۸,۸:



در یک سیستم با 3 پردازنده P1، P2 و P3 که هر کدام صف امادگی مختص خود را دارند از pull migration استفاده می‌شود. اگر وظایفی با مشخصات زیر به هر کدام از این پردازنده‌ها انتساب داده شده باشد (درایه اول از سمت چپ زمان release شدن و درایه دوم زمان اجرای وظیفه است) و الگوریتم زمانبندی آنها Shortest Remaining-Time First باشد، لحظات مهاجرت را مشخص نمایید (سربار مهاجرت را صفر فرض نمایید):

P1: Taks1(0,5), Task2(0,7)

P2: Task3(2,2)

P3: Task4(0,1), Task (5,2)

two approaches to load balancing

۱- pull migration : وقتی یک پردازنده بیکار میشه به بقیه پردازنده‌ها سر میزند و اگر ببیند سر یک پردازنده شلوغ ممکن است یک فرآیند را که درحالت انتظار سرویس گرفتن روی پردازنده شلوغ است را بردارد و روی خودش ببرد که در این حالت اصطلاح pull کردن برای این کار استفاده می‌شود.

۲- push migration: یک task خاص بصورت دوره‌ای (پریودیک) وضعیت روی پردازنده‌ها را چک می‌کند و اگر عدم توازن (imbalance) شناسایی شد، load از روی پردازنده overloaded به روی پردازنده کم بارتر (less-busy processor) push می‌شود.

نکته) هیچ یک از این دو ایده نسبت به دیگری بهتر نیست و به همین دلیل اغلب بصورت موازی و از ترکیب این دو در سیستم‌های متعادل کننده بار برای Migration استفاده می‌شود که بهش push-pull می‌گویند.

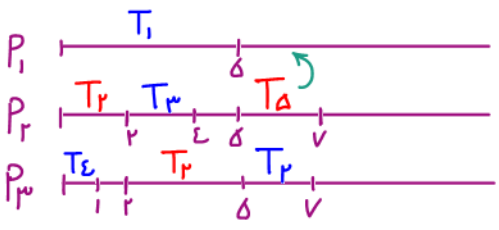
نکته) مفهوم تعادل بار (balance load) می‌تواند معانی مختلفی داشته باشد، مثلاً

- همه صف‌ها تقریبا به اندازه هم ترد داشته باشند
- توزیع مساوی اولویت‌های نخ‌ها در تمامی صف‌ها

release/exe time

P_1	P_2	P_3
$T_1(0,5)$	$T_3(2,2)$	$T_4(0,1)$
$T_2(0,7)$		$T_5(5,2)$

در لحظه ۰، P_2 خالی است و بنابراین T_2 که آن هم در لحظه ۰ وارد شده است را برمی دارد و تا ثانیه ۲ که T_3 وارد می‌شود آن را اجرا می‌کند، حال در ثانیه ۲، ۲ ثانیه از T_3 و ۵ ثانیه از T_2 باقی مانده است، پس T_3 روی P_2 اجرا خواهد شد.



حال در همین ثانیه ۲ پردازنده ۳ نیز خالی می‌شود و بنابراین T_2 را برمی دارد و روی خودش تا ثانیه ۷ که T_5 روی P_3 می آید اجرا می‌کند، حال در ثانیه ۵ وضعیت P_3 به این ترتیب است: زمان باقیمانده هر ۲ تسک ۲ و ۵، ۲ ثانیه است. در این شرایط T_2 را روی P_3 ادامه می دهیم و cs انجام نمی‌دهیم و T_5 را می‌توانیم روی P_1 یا P_2 ببریم.

لحظات شروع مهاجرت = ۰ و ۲ و ۵

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

فرض کنید سایز PCB که در هر Context Switch بطور کامل ذخیره و بازیابی می‌شود برابر با 1000 بایت باشد، هر دسترسی خواندن یا نوشتن به حافظه 20 نانوثانیه باشد، و الگوریتم زمان‌بندی Weighted Round Robin باشد. اگر فرآیندهای CPU Intensive داشته باشیم و بخواهیم سربار Context Switch حدود 0.01% باشد، طول برش زمانی در حالات زیر به چه عددی تنظیم شود؟

1) وزن تمام وظایف یکسان باشد.
2) N وظیفه داشته باشیم که وزن نیمی از آنها دو برابر وزن نیمه دیگر باشد.

memory Access Time = ۲۰ ns

$\frac{1}{10000}$

حل: باید time slice را با توجه به فرمول زیر بدست آوریم:

$cs\ time < time\ slice \times 0.01\% \Rightarrow time\ slice > 10000 \times cs\ time$

توجه کنید که هنگام cs، ۱۰۰۰ بایت برای فرآیند اول باید ذخیره (write) و ۱۰۰۰ بایت برای فرآیند دوم باید از حافظه بازیابی شود (Read)

$cs\ time = 2000 \times 20 \times 10^{-9}\ s = 4 \times 10^{-5}$

$cs = 2KB$

$\Rightarrow time\ slice > 10000 \times 4 \times 10^{-5}\ s \Rightarrow time\ slice > 10^{-1} \times 4\ s \Rightarrow time\ slice > 0.4\ s$

شرایط زیر را برای یک سیستم خاص در نظر بگیرید:

	ALLOCATION				MAX			
	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2
P1	1	0	0	0	1	7	5	0
P2	1	3	5	4	2	3	5	6
P3	0	6	3	2	0	6	5	2
P4	0	0	1	4	0	6	5	6

AVAILABLE: (A=1,B=5,C=2,D=0)

الف) با استفاده از الگوریتم بانکدار بررسی کنید آیا سیستم فوق در حالت امن قرار دارد؟

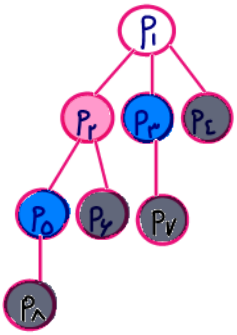
ب) اگر درخواست $request_1=(0,4,2,0)$ از جانب P_1 بیاید، آیا باز هم سیستم در حالت امن قرار خواهد داشت؟

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

قطعه کد زیر را در نظر بگیرید و به سه سئوال پاسخ دهید: الف) چند فرآیند ایجاد خواهند شد؟ ب) چه مقادیری و با چه ترتیبی چاپ خواهند شد؟ (توضیح: فراخوان سیستمی print_pid() شناسه یکتای خود فرآیند را از سیستم‌عامل دریافت کرده و چاپ می‌نماید.)

```
#include <stdio.h>
#include <unistd.h>
int main() {
        /* fork a child process */
        fork();
        print_pid();
        /* fork another child process */
        fork();
        print_pid();
        /* and fork another */
        fork();
        print_pid();
        return 0; }
```

```
P1(){
    ≡
① → x = fork();
    ≡
② → y = fork();
    ≡
    z = fork();
    ≡
}
```



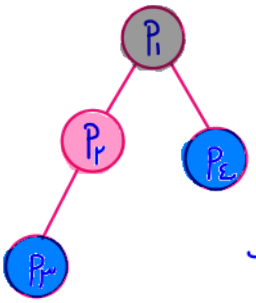
زآیند P_1 درنهایت هم خطوط برنامه را اجرا می‌کند و هم فورک ها را اجرا می‌کند، پس P_1 ← فرزند خواهد داشت

① فرزندان که زیر فورک اول ایجاد می‌شوند تمای خطوط زیرشان را اجرا می‌کنند، بنابراین ۲ فورک زیرشان را اجرا می‌کنند و بنابراین این فرزندان هریک ۲ فرزند خواهند داشت، حال نقط P_1 در اینجا به وجود می‌آید

② فرزندان فورک دوم P_2 شان زیر این فورک قرار می‌گیرد پس فورک های زیر این فورک را اجرا می‌کنند

سوال) اگر اولویت اجرا با فرزندان باشد خروجی برنامه زیر به همراه فرزندانش چیست ؟

```
Main(){
  print("R");
  fork();
* → print("A");
  fork();
  print("M");
}
```



یک فرآیند پدر داریم که وقتی اجرا می شود اولین چیزی که چاپ می‌کند R است، حالا P_1 و P_2 در * هستند و هیچ کدوم از * به بالا را اجرا نمی‌کنند، حالا اولویت با فرزند ترمز است. حالا این فرزند ترمز بعد از چاپ A یک فرزند آبی ایجاد می‌کند و حالا اولویت با P_3 است، حالا P_3، M را چاپ می‌کند و کارش تمام می‌شود، حالا اولویت با P_2 است حالا P_2 هم یک M دیگر چاپ می‌کند و کارش تمام می‌شود، حالا دوباره می‌رویم سراغ P_1 در نقطه * است، حالا P_1، A را چاپ می‌کند و بعد P_4 را ایجاد می‌کند، حالا P_4 میاد M چاپ می‌کند و کارش تمام می‌شود و بعد P_1، M را چاپ می‌کند

خروجی = RAMMAMM

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.

سوال : خروجی برنامه زیر چه خواهد بود؟

```
#define size 5
int nums[SIZE]={0,1,2,3,4};
int main()
{int i;
 pid_t pid;
 pid=forK();
 if(pid==0){
    for(i=0;i<SIZE;i++)
    nums[i] *=-i;
    printf("child:%d", nums[i]);
 }
 else if(pid>0){
    wait(Null);
    for(i=0;i<SIZE;i++)
 printf("parent:%d,nums[i]);
 }
 return 0;
}
```

child: ۰
child: -۱
child: -۴
child: -۹
child: -۱۶
parent: ۰
parent: ۱
parent: ۲
parent: ۳
parent: ۴

سایت کافه تدریس و استاد رضوی هیچ رضایتی نسبت به استفاده غیرمجاز از ویدئوها و جزوات ندارد، استفاده غیرمجاز از این ها کاملا حرام و پیگرد قانونی دارد.