Subject:

Year. Month. Date. ( )

برای ایجاد Thread ← نیازمند منابع جدید نداریم و در حافظه مشترک هستند بنابراین سرعت ایجاد و اتمام آن
کمتر است
این حوزه خود جدول در process هم هست
TLB نیازی به خالی شدن ندارد چون فضای حافظه عوض نشده

اضافه کردن تعدادی از Thread های مختلف تقریباً random نیست
برنامه نویس می‌تواند Thread ها را مشخص کند که هم compiler ، OS این را برای او دیده
و به نظر برنامه‌نویس کار می‌کند

OS یک دست سرویس است و به دنبال فرایند گذاری ندارد و فرایندها می‌توانند بدون آن نیاز کنند
از طریق System Call می‌توان به این سرویس دسترسی پیدا کرد.

حافظه بین Thread ها مشترک است اگر یک Thread قسمتی از حافظه را تغییر دهد بقیه آن را می‌بینند
Thread برای کسانی مختلف نیست اما می‌تواند اطلاعات یکدیگر را خراب کند. کدهای ارتباط Thread ها از
طریق فضای مشترک است. کدهای ارتباط process ها از طریق message است)

مزایای Thread

کم هزینه بودن Thread برای ایجاد و پایان

نیازی به منابع جدید هم ندارد منابع اصلی

حافظه مشترک و ارتباط کمتر سربار Message خیلی بیشتر است.

هر چیزی که سربار کمتر هم می‌پذیرد

caching وجود دارد ، اطلاعات را به disk می‌آوریم در حافظه نگه داریم (موقتاً) چون خیلی است
دوباره آن نیاز شود اگر من یک اطلاعات را Cache آوردم بعدها هم برای بقیه می‌توانند از آن
استفاده کنند پس مدیریتی خودم پس کار تا این حیلی سریع ، کمپایلر بدهیم

P4PCO

Subject:

Year. Month. Date. ( )

س ۵

اختصاصی Thread : PC

Stack ✓

register ✓ حداقل reg های محیط حذف دارند

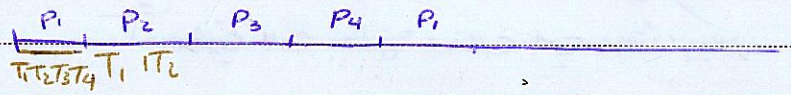
spread sheet ✓

بقیه مباحث هم تقریبی هستند

(مبحث ۴ کتاب Stalling ، Silberschatz)

زمان بندی :

$P_1 \dots P_4$



زمانی که P_1 اختصاص داده می‌شود شامل Thread هم هست. این زمان باید بین Thread ها تقسیم شود اگر بیشتر باشد دیگر زمان کوچک نیست ولی اگر کمتر باشد برای اجرای درست برنامه کفایت نمی‌کند

OS وجود زمان بندی اجرای هر کدام از موارد زیر را دارد: process ✓ ، Thread ✓

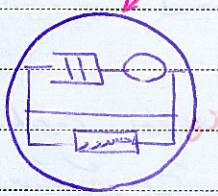
برخی از حافظه‌ها برای اجرای درست باشند

OS این حافظه‌ها را برای هر process یک صف جداگانه در حافظه در نظر می‌گیرد

زمان بندی اصلی، process متفاوت نیست

OS باید بتواند وجود زمان بندی اجرا را بدهد

process یک



Subject:

Year. Month. Date. ( )

پردازنده چنین سیستمی یک State machine است که برای هر دستور العمل یک حالت تعریف شده است و یک مدار برای هر یک از این دستورالعمل‌ها وجود دارد.

هنگامی که برنامه‌ای توسط CPU اجرا می‌شود ممکن است توقف‌هایی داشته باشد.

* ادامه: هنگامی که یک برنامه در حال اجرا است CPU به چه چیزهایی نیاز دارد؟ برای پاسخ به این سوال باید ویژگی‌های زبان‌های برنامه‌نویسی را بررسی کنیم.

زبان‌هایی که عملیات floating point زیاد دارند مثل متلب

زبان‌هایی که spelling check زیاد دارند مثل پایتون

زبان‌هایی که عملیات floating point کم دارند مثل C

زبان‌هایی که spelling check کم دارند مثل C

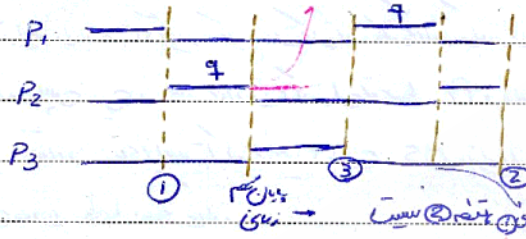
حال ببینیم اگر یک برنامه در حال اجرا باشد CPU به چه چیزهایی نیاز دارد؟

مثلا برای یک کاربر یک webpage هنگامی که کاربر click می‌کند اتفاقی می‌افتد. این عملیات است که محاسبات CPU را به خود مشغول می‌کند.

پس احتمال دارد برنامه‌ای از زبان‌های نوع C که دستیابی کاربر (interactive) مشکل دارد در کدهای غیر interactive مشکل ندارد چون منتظر کاربر منتظر چیزی نمی‌ماند.

تنها راه حل این مشکل Time sharing است؛ یعنی هر چقدر خواستی از CPU استفاده کن ولی به هر برنامه فقط یک کوانتوم زمانی می‌دهیم که تمام شد به برنامه بعدی می‌دهیم (مشغول نگه داشتن سیستم عمل)

Time sharing در حالت کلی به دو دسته تقسیم می‌شود



طول کوانتوم زمانی q در این حالت برای همه یکسان است.

اگر برنامه‌ای بیشتر از کوانتوم زمانی نیاز داشت به انتهای صف می‌رود.

در Time sharing می‌خواهیم حداکثر برنامه‌ای که همزمان می‌توانند اجرا شوند را تضمین کنیم.

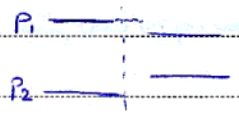
در این حالت باید کوانتوم زمانی را به اندازه $\frac{1}{n}q$ در نظر بگیریم تا بتوانیم اجرای برنامه‌ها را تضمین کنیم.

در حالت Time sharing زمان پاسخ تضمین می‌شود و زمان پاسخ حداکثر $(n-1)q$ است (point 1)

اگر n برنامه باشد.

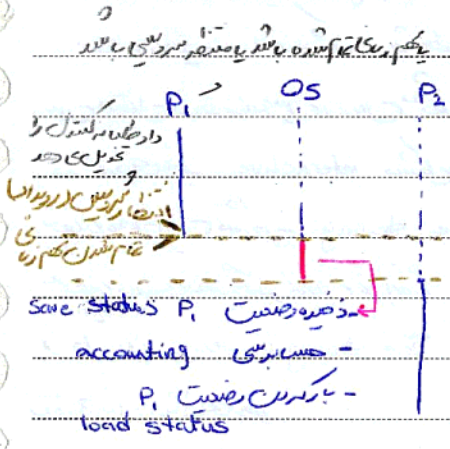
Subject:
Year. Month. Date. ( )

بین اجرای برنامه‌ها در زمان عوض کردن پردازنده زمانی وجود دارد سیستم عامل باید اجرا شدن تا پردازنده را زمانی
بگیرد برگردی بعد پس بین این دو برنامه ناصله زمانی وجود دارد که حتما راست بعد اصحبت کنیم
بین زمان تغلیب برنامه و شروع برنامه بعدی ناصله ای است



جلسه دوم 6/26

سیستم عامل باید بتواند همیشه برنامه را به صورت مجدد اجرا کند و کار عامل نیز پردازنده متقابل عامل را در هر لحظه‌ای که اجرا کند و در کنترل نظارتی بتواند. چونکه عامل است که برنامه نیز پردازنده را در دست طولانی در اختیار نداشته باشد و کامپیوتر به نتیجه نرسد در این صورت نتیجه کلی زمانی مطلوب نیست

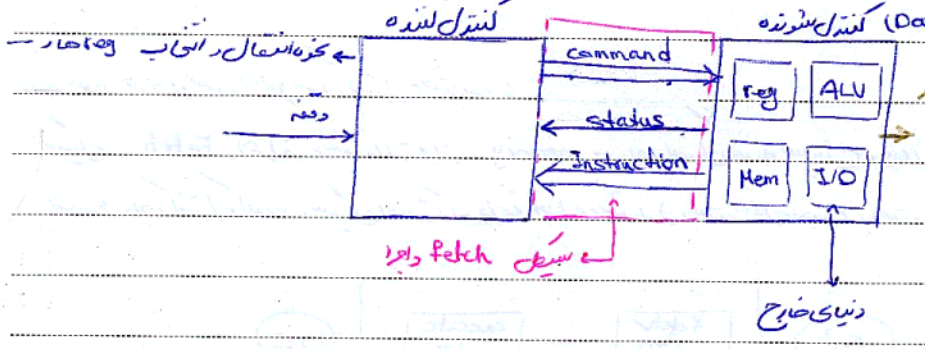


هرکدام از این اسامی یعنی
P1 کاربر عادی شود و P2 شروع
به کار کند در این صورت برنامه‌ریزی است
باید کنترل را از P1 بگیرد و به P2 تحویل بدهد چون خودش نمی‌تواند
این کار را بکند OS این کار را انجام می‌دهد. OS در حالت
P1 کنترل را به دست گرفته و کار کند و کنترل را به P2
می‌دهد. پس بین از دست دادن کنترل توسط P1 و دادن کنترل به
P2 یک اصطلاح زمانی است که در این زمان OS نیز پردازنده را در اختیار دارد
و کنترل‌ها را switch می‌کنند.
OS باید کدام وضعیت P2 کنترل را به P2 واگذاری کند پس OS این کار را انجام می‌دهد چون P1، P2
نمی‌فهمند که این کار را انجام دهند
OS تا پردازنده را در اختیار نگیرد هیچ کاری از آن ساخته نیست پس OS همیشه تا برنامه اجرا است در حال
اجرا است.

Subject:
Year. Month. Date. ( )

مروری بر معماری کامپیوتر:

معماری سیستم (بخت افزار) کامپیوتر از دو بخش تشکیل شده:



واحد کنترل کننده: دستورالعمل ها را می خواند و command های لازم را برای دستورالعمل ها می سازد، داده ها را می خواند و می نویسد، memory هست، آدرس داده ها را می خواند و command های مربوط به آن را می سازد، علاوه بر دستورالعمل ها باید از وضعیت (status) مطلع باشد (carry, overflow, zero, ...)

دقت: می توان کد را دقیق نوشت برنامه را تفکیک کند که هر کدام را بعد دوباره برنامه قبلی را اجرا کند

واحد کنترل کننده: hard wire ← بهتر است.
microprogram (از این دقیق)
به خاطر سرعت پایین از این دقیق

reg ها: رویت شونده (observable) ← هنگام برنامه نویسی می توان از آن ها استفاده کرد
Data reg, address reg, stack pointer, ...
رویت نشونده (nanobservable) ← برنامه نویس از این ها می تواند استفاده کند برنامه نویسی که سیستم را کرش می دهد می تواند از این ها استفاده کند ← کنترل های حافظه، وضعیت دیسک و شبکه از دیسک برنامه نویس است
کسی سیستم عامل مثل OS ... باشد می تواند از reg های رویت نشونده هم استفاده کند
reg های رویت نشونده ← سیستم عامل با این ها کار می کند (وضعیت از ضربه می خورد تا حتما با خطا کار کند)

Subject:

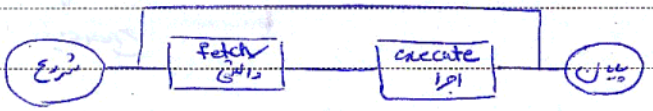
Year. Month. Date. ( )

← چطور میشه این reg ها برای OS دیده میشود ولی برای P_1, P_2 قابل دسترسی نیست از کجا میخ
کدام برنامه است که اجازه میدهد reg ها را ببیند یا نه؟

← نحوه اجرای دستورالعمل در کنترل شده:

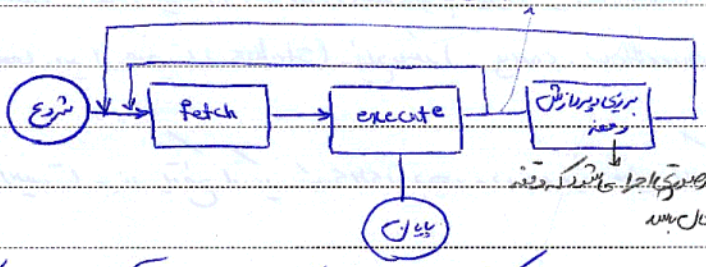
سیکل Fetch و اجرا: دستورات از memory به واحد کنترل کننده سپس ارسال command به واحد کنترل شده
توسط واحد کنترل کننده. سیکل است که دائماً اجرای شود (دستورالعمل بعدی، دستورالعمل بعدی ...)

سیکل دستورالعمل →



خال بودن پردازش وقفه

بررسی خال بودن وقفه →



وجود وقفه برای اجرای شدن
خال نبود

سیکل پایان اجرای دستورالعمل یک برنامه ای که وقفه ای کنیم که ببینیم چیزی آمده یا نیامده اگر نیامده برو به دستور بعدی برنامه
پردازش وقفه ← بعد از اجرای دستورالعمل و وسط سیکل fetch اجرا اجرا نمی شود

← پردازش وقفه های خارجی سیکل اجرای دستورالعمل (وقفه ای داخلی هم داریم)
اگر وقفه ها نباشد امکان پیاده سازی کمپ زمانی وجود ندارد

حافظه: عناصر ذخیره سازی
حافظه در سیستم کامپیوتر سلسله مراتبی است (hierarchical)

سریع ترین و پرهزینه ← ثبات register
حافظه نهان (Cache memory)
حافظه اصلی (main memory)
حافظه نهان disk (disk cache)
ذخیره انبوه (disk) (Mass storage)

حافظه‌های cache در این جهت قرار دارند

دورتر از پردازنده (ظرفیت بیشتر)

از این جا اطلاعات پردازنده و ظرفیت

افزایش سرعت

Subject:
Year. Month. Date. ( )

حافظه‌های سلسله مراتبی ← علاوه بر بالا بردن ظرفیت سرعت هم بالا برد و هدف این است که ما بالا کار کنیم (در register ها) ← ما مستقیم با حافظه‌های disk کار نمی‌کنیم بیشتر با regs ها کار می‌کنیم

← عوامل ذخیره سازی چگونگی پیاده سازی کنترل می‌کند.

register ها: compiler (زبان‌های سطح بالا)، خود برنامه‌نویس (زبان‌های سطح پایین)

حافظه اصلی: سیستم عامل (ذخیره اصلی ذخیره سازی حافظه...)

disk: سیستم عامل ← دیسک را به block ها تقسیم‌بندی می‌کند برای یک file system directory ساخته...

حافظه پنهان cache: خودش خودش را کنترل می‌کند چون یک سخت‌افزار کنترل‌کننده همراه خودش دارد و OS اصلا از وجود چنین چیزی خبر ندارد. کسی نمی‌تواند cache را برای بیشتر کنترل کند خودش در کارهای خلاقانه می‌کند تا کارها با سرعت بیشتر ← کنترلش سخت افزاری است.

حافظه پنهان disk: کنترلش با خودش است و نمی‌گذارد کسی خراب بشود. disk زیستی پیدا کند و کارآیی را با دخالت کردن سرعتی بخشیدن هم می‌تواند

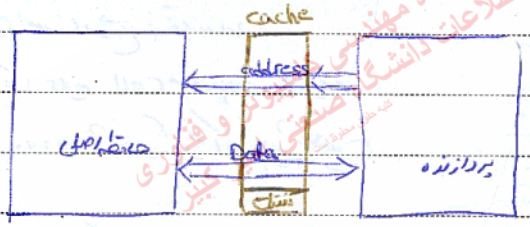
← OS فقط حافظه اصلی و disk را برای بیشتر از وجود حافظه‌های پنهان خبری ندارد.

کلاس سوم:

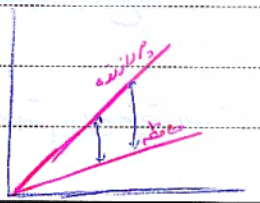
* سیستم عامل برای کارخود نیاز به دقت دارد و بدون وجود دقت نمی‌تواند کار خود را بدرستی انجام دهد

حافظه پنهان یا cache:

مثل‌هایی که زدیم سرعت بالا را در حافظه سلسله مراتبی بیشتر دیدیم که هدف ما خواهیم در حافظه پنهان ادامه پیدا کنیم



حافظه مکانیکی ← شانه و دیسک خودش سرعت کند



Subject:
Year. Month. Date. ( )

برداشته و به اصل آدرس‌ها را به حافظه اصلی برای آن فرستاده می‌شود. cache در این واسطه قرار گرفته و آدرس را نظارت می‌کند اگر این آدرس در cache باشد آدرس را به حافظه اصلی نمی‌دهد و خودش Data را می‌فرستد. بخش‌هایی از حافظه در cache کپی می‌شود که حافظه ندارد. cache از حافظه static استفاده می‌کند و سرعت بالا و حافظه کم ظرفیت کم است. چون قطعه‌ها در cache کپی می‌شود temporal و spacial

اصل محلیت (locality) هم از لحاظ زمان و هم از لحاظ مکان نیز می‌دهد.

Temporal → اگر آیتم‌هایی که اخیراً در حافظه دسترسی پیدا کرده احتمالاً باز هم دسترسی داشته باشد (آدرس‌ها)

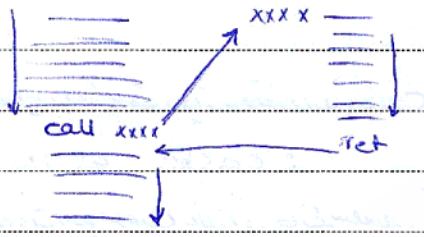
Spacial → اگر یک آدرس دسترسی پیدا شود، احتمالاً به آدرس‌های مجاور آن دسترسی پیدا می‌کنیم

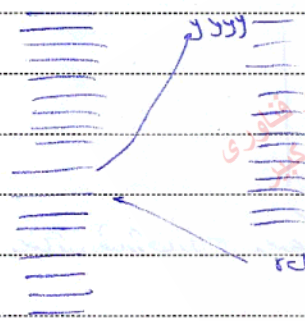
نقش این است که لازم نیست هر حافظه را در cache داشته باشیم بلکه به دلیل اصل محلیت بیشتر به cache دسترسی داریم و میزان موفقیت ۹۵٪ است.

cache را تحت اختیاری که برای این کامپیوتر کنترل می‌کند همراه آن دستکنترل می‌کند عکس العمل‌ها را این کنترل اجرا می‌دهد. اگر دستوری داشت Data را تقاضا می‌کند به حافظه می‌دهد آن آدرس را برمی‌دارد و درصورت لزوم چیزی را از خودش حذف می‌کند و آن را جایگزین می‌کند و تمام این کارها را بدون اطلاع از OS و CPU اجرا می‌دهد

مکانیزم وقفه:

روال جاری یک ... call و return





روال وقفه هم شبیه همین است و هیچ فرقی ندارد

فرق call با وقفه این است که محل call معلوم است ولی وقفه هر وقت دلش بخواهد ممکن است رخ دهد و به دلیلی که از پیش تعیین نشده است می‌رود.

هر وقت وقفه رخ می‌دهد برنامه را تقاضا کند

این مکانیزم وقفه برای سیستم عامل حیاتی است.

Subject:
Year. Month. Date. ( )

← کلا OS برسر وقفه کار می‌کند (به شدت قوی است)

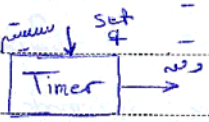
دسته‌بندی وقفه‌ها: (interrupt, exception)

1. اجرای برنامه: مثلا برنامه در حال اجرست و عملی از او سر می‌زند (تقسیم بر صفر، استفاده از دستور الحل نوشته نشده، سرریز یا overflow، آدرس خارج از فضای تعریف شده، ...) پردازنده باید چکار کند برای اینجا باید چه کار کند؟ تنها دفاع پردازنده در این موارد تولید وقفه است و تولید وقفه باعث تقلع برنامه می‌شود و به جای دیگری می‌رود (محل تولید وقفه می‌داند و کار دیگری می‌کند این بررسی وقفه است که کار را اجرا می‌دهد) یا تصحیح می‌کند یا برنامه را قطع می‌کند (پردازنده هم چین سخت‌افزاری برای این کار ندارد و باید برنامه‌ها این کارها را اجرا دهد) ← در این اجرای دستور الحل قطع می‌شود

← عکس العمل پردازنده در برابر خطا یا عملیات غیرمجاز (تولید وقفه)

2. زمان‌گیر (Timer): هر برنامه یک چیزهایی دارد و یک Timer وجود دارد که وقتی زمان اجرای برنامه تمام شد تولید وقفه می‌کند و دیگر است کار هم‌چنان برنامه نیست

← چیزهایی که برنامه دارد (بخشی از اطلاعاتی که آن را در زمان مخفی set می‌کنیم باشد) چیزهایی (IP) و شروع به کار می‌کند و هر وقت به زمان q رسید تولید وقفه می‌کند OS این را set می‌کند و قبل از این که کنترل را به برنامه‌ها بدهد این را set می‌کند.



یعنی به OS کنترل را برگشتی دهد Timer.

set می‌شود.

3. وسیله‌ی خروجی I/O: هر سیستم کامپیوتری یک سری port دارد به I/O

ارتباط با دنیای خارج از طریق این port هاست که برای خود وقفه دارند.

برنامه‌ای منتظر اطلاعات ورودی است از کی؟ منتظر اطلاعاتی که وارد شده است کنترل کلی برعهده OS است. ورود اطلاعات خودش را به خودش را به او خبر می‌دهد و اطلاعات را

اطلاعات نوشته شده

وقتی بعد باعث تقلع شدن برنامه در حال اجرست که این را در جایی از حافظه ذخیره می‌کند و اطلاعات ورودی را باید در جایی از حافظه ذخیره کند و آن جا برای برنامه حفظ است اما نباید باشد وقفه‌های I/O پایان عملیات ورودی خروجی را اعلام می‌کنند. پایان خروجی ← اطلاعات نامه خارج شده و دیگر نمی‌توانیم روی out بنویسیم

Subject:
Year. Month. Date. ( )

4 آخرین وقفه به سیستم عامل ربطی ندارد (نرم‌افزار) و این سه بیت 0 مربوط بودند.

4- نقص سخت‌افزار (HW Fault) می‌تواند طوری ساخته شده باشد که در صورت خراب شدن تولید وقفه کند
ساده‌ترین راه برای حافظه: وجود parity که در صورت دستکاری تولید وقفه کند
تغییر برق می‌تواند طوری طراحی شده باشد که تولید وقفه کند تا خاموش شدن خازن‌های مدار وقفه‌ای تولید کند و اطلاعات کم را ذخیره کرده، هرچند کمتر از اندازه‌ای که این وقفه را تولید کند به بیرون که این طوری ساخته شده باشد.

← وقفه نوع اول در وسط اجرای دستور العمل می‌تواند باعث شود وقفه چون دستورالعمل غیری است را صدا بزند
آن را برای همه‌ی وقفه‌های دیگر اجرا می‌کند تا آخر اجرا شود
بعد نوع اول وقفه باید دوباره دستورالعمل قبلی که در حال اجرا بود اجرا شود که در بین وقفه‌ها چون دستورالعمل کامل اجرا شده بود توقع می‌شود به دستور بعدی برود اما دستور بعدی خراب است، حال وقفه را اجرا کند.

← برنامه‌های کاربری نباید بتوانند به port ها دسترسی داشته باشند و باید سیستم عامل یک واسط بین کاربر و این port ها برای همه برای اینکه از هرکس بخواهد این port ها را بخواند درج و صحیح شود، OS این وسط کمک می‌کند. همچنین به port ها دسترسی دارد تا از رسیدن اطلاعات کم زیاد جلوگیری شود و اطلاعات رسیده برای کی است. این کار باید دست OS باشد چون ترتیب‌ها را باید دست داشته باشد هر اطلاعات دیدی را که می‌خواهد، این یک نعمت است ولی می‌تواند طوری باشد که پس باید کاری کنیم که نعمت نباشد

۵- دوگانه (Dual mode)
برنامه‌ی در حال اجرا در mode اجرای دارای اختیارات کمتر است، برنامه‌ی می‌تواند حتی کارهای غیری را بخواهد و تولید وقفه کند
اختیارات زیاد برای هیچ دستوری ندارد حتی کارهای خاصی می‌تواند بکند
System mode : اختیارات زیاد دارد (دستوری مثل port، Set، Timer، ...)
User mode : اختیارات کمتر

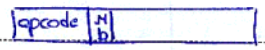
Subject:
Year. Month. Date. ( )

جلسه چهارم:

با port های ورودی-خروجی از طریق دستورات ورودی-خروجی کار کردیم. اداره port ها با OS است و هر برنامه‌ای که بخواهد port ها را کنترل کند باید از سیستم عامل بخواهد.
در سیستم‌هایی که توان port های reg دیده می‌شود کاربر دیگر نمی‌تواند مستقیم با port ها کار کند.
در سیستم‌های کاربر عملیات ورودی-خروجی توسط وقفه انجام می‌شود و به جای کنترل مستقیم، درخواست‌هایی به OS است.

برای طراحی چنین چیزی یک بیت اضافه به نام mode bit به سیستم می‌دهیم.
۱ کاربر
۰ سیستم

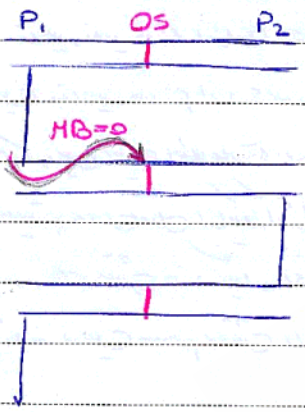
یک بیت به دستورالعمل‌ها (opcode) اضافه می‌کنیم تا mode bit آن‌ها را مشخص کند.



opcode ← در اجرای دستورالعمل‌هایی که این بیت را دارند، mode bit جاری چک می‌شود.
دستورالعمل اجرا می‌شود در غیر این صورت تولید وقفه می‌کند.

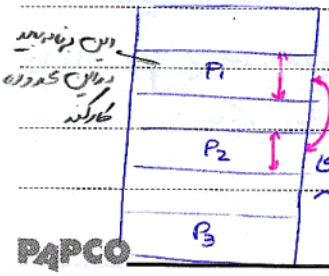
چگونه mode bit تغییر می‌کند

هنگام شروع کار MB باید صفر باشد (در reset و اجرای برنامه‌های سیستم) و زمانی که OS برنامه‌ای را جهت شروع کار به CPU می‌دهد (OS است) حالای خواهد کنترل را به برنامه‌ای بدهد قبل از اینکه برنامه کنترل را پس دهد باید MB را ۱ کند.
تنها دستوراتی که می‌توانند mode bit را عوض کنند دستورات OS است و وقتی کنترل دوباره دست OS می‌افتد حتماً صفر می‌شود.
برنامه‌ای که بخواهد mode bit را صفر کند برنامه تله می‌شود.



حفاظت حافظه (memory protection)

چند برنامه داریم که هست هستند اجرا می‌شوند و این برنامه‌ها باید برای اجرا در جای از حافظه باشند
چون برای اجرا PC باید به کدام برنامه اشاره کند که حافظه‌ای آن برنامه حافظه اشاره کند هر برنامه
کجا خرش را دارد و هر کدام کجا هستند باید بدانند محدوده‌ی خرش چیست



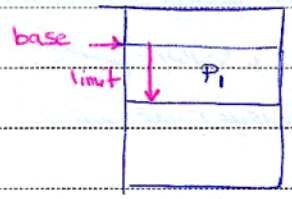
P4PCO

Subject:
Year. Month. Date. ( )

بستر کاربر که چون باعث تغییر data می‌شود (بنابراین نمی‌شود به حافظه دسترسی پیدا کند)

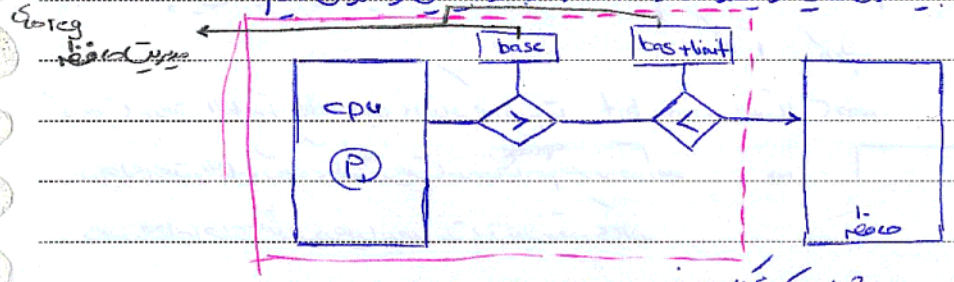
حفاظت حافظه ← هر کس در محدوده خودش بتواند کار کند

هر برنامه یک آدرس base دارد که محل شروع برنامه در یک limit دارد که طول برنامه است. هر برنامه با این دو عدد در اختیار OS است (هنگام اجرای یک برنامه OS نیاز دارد که در غیر این صورت وجودش احساس نشود)



در این حافظه برای کاربر جدول ایجاد شده است

حالتی که P در حال اجرا است باید آدرس‌هایی تولید کند که محدوده P باشد. بنابراین کنترل کنیم



قبل از اینکه آدرس را به حافظه بفرستیم دو شرط را چک کنیم تا بفرستیم در محدوده است یا نه:

1- باید حتماً بزرگتر از base باشد

2- باید کوچکتر از base+limit باشد تا در محدوده P باشد

این مقایسه باید بسیار سخت افزاری انجام شود ← مقایسه کننده‌ها داخل CPU است

اگر این مقایسه حافظه سخت افزاری بود این سیستم دیگر به درد نمی‌خورد. کار OS در این مقایسه تا base و limit را تامین کردن این اطلاعات را داخل reg (دیده نشده) از دست

← سیستم عامل قبل از اینکه کنترل را به P بدهد، base و limit مربوط به P را کجا کند و بعد کنترل را بدهد. OS نقش خود را ایفا می‌کند

مدیریت حافظه نقش دارد ← که این reg‌ها (چون دیده نشده هستند)

دستورالعمل‌های مربوط به reg‌های حافظه از نوع ممتاز (privilage) هستند

privilage ← دستورالعمل‌هایی که فقط در سیستم قابل اجرا هستند و در صورتی که کاربر اجرا کند نمی‌شود و تنها Mode bit هر یک دستورالعمل privilage است

Subject:
Year. Month. Date. ( )

به طور کلی برنامه فقط دستورالعمل‌های تعریف شده اجرا شده می‌توان آن را طوری طراحی کرد که در مقابل کارهای که به زیان ایجاد تولید وقفه کند پیش از آن به پردازنده وقفه‌ای بدهد...

یعنی جدایی آن در شرایط خیلی عادی به صورت تولید وقفه است

← سیستم عامل در صورتی می‌تواند خوب کار کند که مدیری سیستم توی وقفه باشد.

← در حالتی CPU را از دست داده و به OS بدهد

دست افتاد ← خودش داوطلبانه این کار را انجام دهد یعنی یک چیزی از OS مثلاً درخواست چیزی از سیستم عامل ← درخواست از سیستم عامل

از کسی؟ که سیستم عامل برایش تایین کند طول کشید

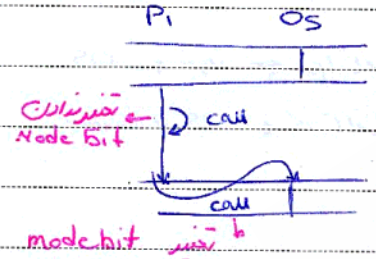
آسان وقفه که کمتر وقفه کم زیان است ← وقفه

وقتی که درخواستی از یک برنامه از OS کرد باید منتظر جوابی باشد تا نوبت به برنامه برسد درخواست‌های جدید برنامه اجرا شود و به محض رسیدن جواب درخواست کنترل به آن برنامه منتقل می‌شود

(point) برنامه‌های درون خود OS هستند و دیکته OS تعریف شده‌اند

← حالتی که یک برنامه درخواست کالند ۰ احتمالاً یک routin را call کالند

این call باید call چه تفاوت دارد چگونه سیر call که درخواست برنامه هست Mode bit را عوض می‌کند و call برای درخواست باید Mode bit را عوض کند.



برای call که system call می‌گذیم

System call ← مجموعه‌ای از call که برای OS هستند و از پیش تعریف شده‌اند. (مجموعه‌ای از توابع تعریف شده برای OS)

فرق بین call معمولی و system call به طوری است که نمی‌توان از راه زیر به system call رفت. Mode bit را تغییر دهد.

P4PCO

Subject:
Year. Month. Date. ( )

نکته: Intel برای این کار یک دستورالعمل تعریف کرده که نرم‌افزار Interrupt گذاشته و آن را بعد صدا می‌کند

INT XX (58)

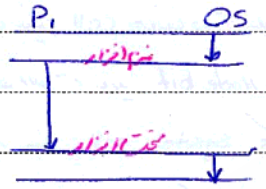
Intel یک opcode INT دارد به همراه 58 (یعنی همه جا INT است) یک آدرس را برای دارد داده برنامه را آن جا اجرا کند. این همان call هست دیگه اینجا call به آدرس غیرمستقیم است.

(چون از قبل همگی کارها در اختیار سیستم‌عامل هستند)

← وقتی به این اجرا شود به طور اتوماتیک mode bit صفر می‌شود و کنترل را به برنامه برنامه‌ریز یا همان OS می‌دهد.

← درخواست از OS ← System call ← تغییر mode bit به طور اتوماتیک (به پیاده‌سازی دستور این تغییر شده و نیازی به دستوری جداگانه برای تغییر mode bit نیست)

تغییر از حالت کاربر به سیستم فقط از طریق Interrupt (وقفه) اجرا می‌شود (یعنی جلوی شد :D)
در واقع دیگه احتمال اینکه P کنترل را از دست بدهد نیست کلی کرهیم (با وقفه)
تغییر mode bit برعکس از حالت کاربر به سیستم از طریق سخت‌افزار است
(point) Interrupt سبب تغییر mode bit می‌شود
(کد اتوماتیک ← و چون سخت‌افزار برای اجرای این کار)



OS ← برنامه ← نرم‌افزار
برنامه ← OS ← سخت‌افزار

Subject:
Year. Month. Date. ( )

جلسه پنجم:

- برنامه‌هایی که جزء اجزای هسته سیستم‌عامل تعریف شده‌اند را اجرا می‌کند.
- برنامه‌هایی که به سیستم‌عامل سرویس می‌دهند را در اختیار برنامه‌نویس قرار می‌دهد.
- بعضی از اعضای سیستم‌عامل OS هستند که توسط الگوریتم‌های خاصی در حافظه cache نگهداری می‌شوند.

اجزای سیستم‌عامل:

- مدیریت فرایند — process management
- مدیریت حافظه اصلی — memory management — این جلسه روی حافظه اصلی است
- مدیریت حافظه ثانویه — Secondry memory management — حافظه‌های ثانویه دیسک هست.
- مدیریت ورودی/خروجی — Input_Output management
- مدیریت فایل — File management
- شبکه‌سازی — Networking — کدهایی که ارتباط بین سیستم‌های مختلف را برقرار می‌کند
- سیستم مفسر زبان دستورات — command Interpreter System — چیزی که با آن در محیط OS می‌نویسیم و OS می‌فهمد. خارج از OS هست ولی جزء OS است.

سرویس‌هایی که در این درس می‌خوانیم

مدیریت فرایند:

- فرایند یک برنامه در حال اجرا است و برای اجرا نیاز به منابعی دارد.
- برنامه: یک فایل روی دیسک که حاوی دستورالعمل‌هایی برای اجرا است.
- فرایند: برنامه‌ای که در حال اجرا است و برای اجرا نیاز به منابعی دارد.
- برنامه‌نویس برای اجرای برنامه‌اش باید فرایند را در اختیار داشته باشد.
- فرایند active است.
- برنامه‌هایی مثل monitor, print برنامه‌هایی هستند که برای اجرا نیاز به فرایند ندارند.

program — process

- یک process یک چیز مجازی است ولی program الزاماً در حال اجرا نیست ولی همیشه یک process در حال اجرا است.
- یک process می‌تواند چند بار اجرا شود و هر بار یک process جدید ایجاد می‌شود.
- program ایستا و بی‌جان است مثل یک فیلم.

مدیریت فرایند یکی از مهم‌ترین کارهای OS است که OS برای اجرای هر برنامه‌ای یک process به آن اختصاص می‌دهد تا برنامه اجرا شود.

مدیریت فرایند شامل این کارها می‌شود:

1. ایجاد و حذف فرایند ← فرایند مشخص می‌دهد
2. معلق کردن و معلق نکردن ← معلق کردن: موقتاً از حافظه خارج کنیم و نیاز نداریم ← به علت کمبود نیاز

suspension — resumption

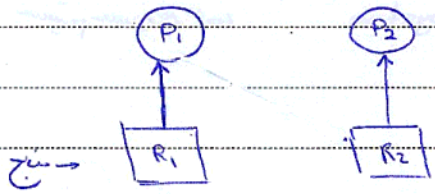
Subject:
Year. Month. Date. ( )

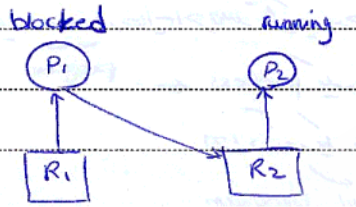
به دلیل اینکه تمام فرآیندها از منابع مشترک استفاده می‌کنند یک سری ارتباط دارند این فرآیندها نه تنها از هم مستقل نیستند بلکه به نحوی همدیگر را تحت تاثیر قرار می‌دهند و وقتی که منابع مشترک آزاد می‌شوند فرآیندهای بعدی که منتظر آن منابع بودند وضعیتشان بهتر می‌شود و از حالت blocked به حالت ready می‌روند

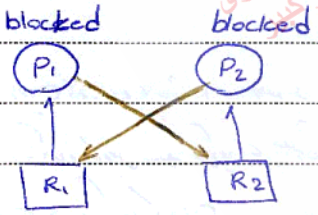
3. مکانیزم‌هایی برای همگام‌سازی فرآیندها: (synchronization)
اول نشان دادیم که نیاز به همگام‌سازی فرآیندها داریم و بعد نشان دادیم که چرا باید همگام‌سازی انجام شود. حال می‌خواهیم ببینیم چگونه می‌توانیم همگام‌سازی را انجام دهیم.

4. مکانیزم‌هایی برای ارتباط فرآیندها: برنامه‌ای که نوشته‌اید برای اجرا شدن به سیستم‌عامل داده می‌شود و برنامه‌ای که نوشته‌اید توسط سیستم‌عامل اجرا می‌شود. پس برنامه‌ی شما با سیستم‌عامل ارتباط دارد. اما ارتباط شما با سیستم‌عامل چگونه است؟ در این قسمت می‌خواهیم ببینیم که برنامه‌ی ما چگونه با سیستم‌عامل ارتباط برقرار می‌کند. گاهی اوقات برنامه‌ی ما نیاز دارد که از سرویس‌های سیستم‌عامل استفاده کند. در این صورت باید به نحوی با سیستم‌عامل ارتباط برقرار کند. برنامه‌ی ما نیاز دارد که با سیستم‌عامل ارتباط برقرار کند. برنامه‌ی ما نیاز دارد که با سیستم‌عامل ارتباط برقرار کند.

5. روش‌هایی برای حل بن‌بست (Deadlock)
این دو برنامه این روش را داشتند که هر کدام از آن‌ها یک منبع را در اختیار دارند و منتظر منبع دیگری هستند.







Dead lock

P4PCO

Subject:

Year. Month. Date. ( )

این که OS بخواهد این حالت اختلاف افتاده کار درستی نیست ولی الزامی نیست راه حلی داشته باشد تنها کسی که این وسط می‌تواند راه حلی ارائه دهد OS است چون خود P_1 و P_2 خواب هستند نمی‌دانند پس می‌تواند کاری انجام دهد

مدیریت حافظه:

حافظه به صورت فیزیکی یک آرایه خطی از کلمات است linear array

← لزوماً نباید از ساختار array استفاده کرد بلکه از هر ساختار دیگری می‌توان استفاده کرد

کارهای مدیریت حافظه:

1. پگیری: قطعات استفاده نشده را به صاحبان آن: یعنی ببینید اندازه متن‌ها از حافظه استفاده نشده و صاحب آن کی است (خود OS این کار را انجام می‌دهد)

2. تخصیص حافظه به پردازه‌ها

3. پگیری: قطعات آزاد را به پردازه‌ها خالی است تا الم خواست به کسی حافظه بدهد کار را بکند

مدیریت حافظه مجازی (Disk):

طی شبیه مدیریت حافظه است.

1. مدیریت فضای آزاد

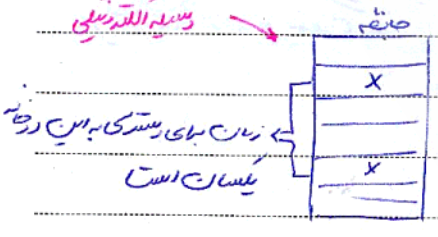
2. تخصیص مکان ذخیره سازی

3. زمان‌بندی (اولویت‌بندی) دیسک: تقاضاهایی برای دیسک آمده باید زمان‌بندی کنیم یا با الگوریتم را اول ... انجام دهیم. این را در این جا طرح می‌کنیم حافظه با دیسک فرق دارد و مکان‌های آن یکسان است

scheduling

دیسک یک وسیله مکانیکی است چون حرکت توش است همین سبب کندی می‌شود

مکان‌ها به هم فرق کنند (مکان نزدیک به جلو ← زمان کمتر برعکس زمان بیشتری بیشتر جلو قیمت نزدیکی دارد)



Subject:
Year. Month. Date. ( )

هارتم : زبان وسیله ی کمک حافظه کامپیوتر است

سیسل : زبان وسیله ی کمک حافظه است.

برای اینکه دقیق تر بتوانیم زبان وسیله ای را انجام دهیم از نوبت بندی استفاده می کنیم مثلا اول به تقاضای پرنتر می دهیم که کمک آن به حافظه کامپیوتر است

مدیریت ورودی خروجی :

یکی از کارهای هم IO management مدیریت صف هاست.

1- مدیریت صف ها : برای اینکه بتواند سرویس ها را انجام دهد

2- کنترل ورودی های خروجی ها : مثلا می خواهیم با port ورودی ارتباط برقرار کنیم قبل از این لازم است port را باز کرده باشیم می خواهیم port کار کنیم باید چیزی در مورد port بدانیم برای همین کنترل آن ها را برای ما نگه دارند

3- سیستم بافر : بافر چیزی شبیه حافظه cache است می خواهیم با port ها کار کنیم با بافر کار می کنیم چون port ها کند است و واسطه بین port و برنامه هستند و می خواهند cache ها کنترل کمتر انرژی ندارند پس کار اصلی با نرم افزار بخش بافر است. اداره کنترل این ها به عهده OS است.

سرویس خطا :

خطا : زمانی که یکسان از اطلاعات را ارائه می دهد

اطلاعات که در جای دیگر ذخیره شده (روی دیسک ، CD ، فلش ، ...) Data می گویند.

نه : یکسان نیستند اگر چنین که گفتیم این اطلاعات یکسان نیستند به یک فرقی حتما دارند حال هر کدام از این ها چیزی را اضافه کرده به برنامه اطلاعات چیزهای مختلفی اطلاع داده اند پس خطا دارند که به این تفاوت ها می خواهیم این تفاوت ها را پنهان کنند

Subject:
Year. Month. Date. ( )

کارهای مدیریت فایل

ایجاد و حذف فایل

ایجاد و حذف راحتی (ها directory)

و مکانیزم‌هایی برای کاربران با فایل

جلسه ششم: 7.9

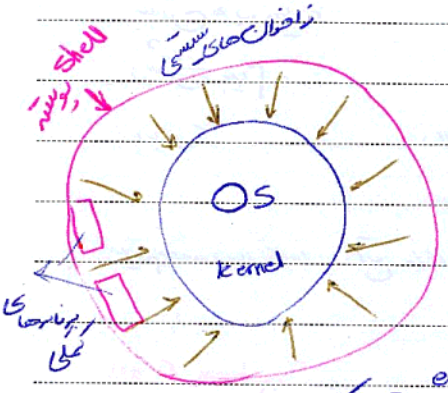
مفسر دستورات Command Interpreter System

واقعاً بخشی از OS نیست و چیزی است که به ما این سرویس را می‌دهد

واسطه‌ای برای کاربر که به وسیله آن بتواند فرمان دهد

سیستم عامل مجموعه‌ای از راه‌هایی است که توسط آن طریق System Call

می‌توان با آن تماس گرفت یا درخواستی را از آن نقاضا کرد.



در همین درخواست‌های از این سیستم عامل هستیم داریم

هسته سیستم عامل درخواست‌های سیستمی است

پایان یافتن یک فرایند exit( )

به OS می‌گوییم این فرایند کارش تمام شده بیا حافظه را از آن بگیر

ساختن یک فرایند (دستور دادن یک فرایند) create( )

به سراغ حافظه برو و فضا به او بده

malloc( )

سیستم مفسر فرمان‌ها:

شامل یک سری برنامه‌های کاربردی که مثل یک پوسته دور OS را گرفته‌اند مثل copy که یک سیستم call

نیست، برای اجرای آن از تعداد زیادی System Call استفاده می‌کند به ترتیب copy یک برنامه است. مجموعه این

برنامه‌ها است که سیستم مفسر فرمان‌ها است و جزو سیستم مفسر فرمان‌ها، OS نیست و در نبود آن OS می‌تواند

به کار خود ادامه دهد.

P4PCO

Subject:
Year. Month. Date. ( )

نحوه‌ی اجرای کپی:

copy F_1, F_2

- دریافت نام ← ساده‌ترین حالت ممکن است
- باز کردن فایل مبدا ← openFile() سیستم کال است
- ایجاد فایل خروجی (به تناسب F_2 نباید قبلا وجود داشته) ← صورت بعد فایل خروجی
- حلقه: خواندن از فایل مبدا
  نوشتن در فایل خروجی
- بستن هر دو فایل
- نوشتن پیام
- پایان

← چون loop اطلاعات را از F_1 به F_2 می‌بریم چون اطلاعات را می‌خوانیم و در F_2 می‌نویسیم

← در این‌ها error ها را مستقیم نمی‌نویسیم و ضمن اجرا می‌بینیم و در صورت وجود خطا باید کاری کنیم مثلا پیغام بدهیم.

← سیستم عمدتا برای راحت‌تر کردن کار کاربر است، OS است

System Calls:

زبان‌های سطح بالا جهت اینترفیس‌های نرم‌افزاری هستند. این زبان‌ها به جدیدی compile شده‌اند که توابع Interrupt های نرم‌افزاری است. اگر برنامه‌ی این زبان‌ها compile شده به اسمبلی برود به جدی برمی‌گردیم

نحوه‌ی ارسال پارامترها ← استفاده از توابع است

روش سوم ارسال پارامتر استفاده از Stack است (push on stack)

```
LD  R1, Parm
LD  R2, Parm   ← openfile(...)
INT 28
```

← تبدیل به زبان ماشین

این Interrupt های نرم‌افزاری که مانند library routine عمل می‌کنند از library compiler خوانده شده و استفاده می‌شوند

به طور کلی به مجموعه‌ی Interrupt های نرم‌افزاری که برنامه‌ی System Call ها را پیاده‌سازی می‌کنیم System Call می‌گوییم

Subject:
Year. Month. Date. ( )

Subject:
Year. Month. Date. ( )

مدیریت فرایند Process Management

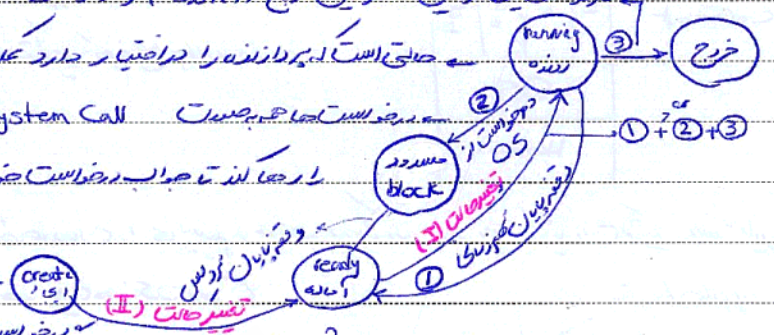
- ترتیب فرایند: برنامه در حال اجرا
  فرایندهای همزمان (همزمان) } قبلاً گفته شد
- زمان‌بندی (نوبت بندی) فرایند: فرایندها همزمان می‌توانند پردازنده را در اختیار بگیرند بگذارید یکی یکی پردازنده را در اختیار بگیرند که در این حالت نوبت آنکه چطور نوبت بدهیم.
  (روش‌ها ← الگوریتم‌های زمان‌بندی)
- انواع روش‌های زمان‌بندی: مزایا این الگوریتم‌ها چیزهایی هست در کجا باید استفاده شوند.

حالت فرایند Process State

فرایند در طول زندگی خود حالت‌های مختلفی دارد که این حالت‌ها تغییر می‌کند

درخواست یک سرویس از سرویس خروج (exit) (درخواست سخت‌افزاری است)

- حالتی است که پردازنده را در اختیار دارد عملیات درست اجرا می‌شود
- درخواست‌ها همه به صورت System Call است. وقتی که درخواست می‌کند باید برنامه را رها کند تا جواب درخواست خود را بگیرد (هر نوع درخواستی که تواند باشد)



- فرایند جدید باید یک جوری وارد سیستم شود.

در حالت خروج منابع را از دست می‌دهد و عملش تمام می‌شود

- در حالت مسدود است زمانی اتفاقی می‌افتد که جواب درخواست آمده (سرویس مورد نظر ایجاد شده) می‌تواند پردازنده را در اختیار بگیرد.

به این برچسب چون دست یکی است پس باید درصف برود تا نوبتش شود پردازنده را در اختیار بگیرد پس به حالت آماده می‌رود تا نوبتش شود پردازنده را در اختیار بگیرد running شود تغییر برنامه هم حق دارند وقتی از پردازنده را در وسط کار از ما پس بگیریم و برنامه دیگری بدهیم

در تمام این کارها چون کاری اجرا دارد باید به حالت آماده برود چون کاری برای اجرا دارد و با نوبت اجرا می‌شود را بحث نداریم

← پایان سرویس حداکثر وقفه اعلام می‌شود

تغییر حالت‌ها ← بر اساس وقفه
بر اساس درخواست ✓

← که این را باید تغییر mode می‌اندازد.

Subject:
Year. Month. Date. ( )

دیتا تغییر حالت توضیح داده نشده ← تغییر حالت (I)
(II) " "

تغییر حالت (I)
زمانی که هریک از تغییر حالت‌هایی که باعث شود برنامه‌ای که در running است این حالت را ترک کند (پردازنده را ترک کند) این تغییر حالت اتفاق می‌افتد یعنی هرکدام از تغییر حالت‌های 1 یا 2 یا 3 اتفاق بیفتد

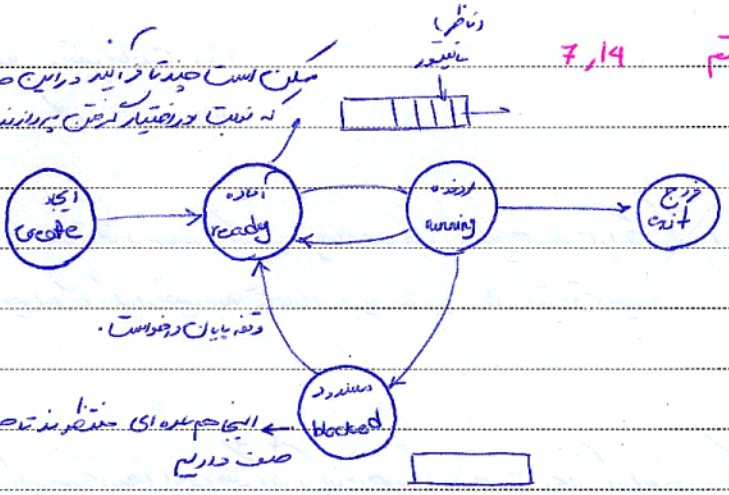
تغییر حالت (II)
ایجاد فرآیند درخواست است و فرآیند جدیدی زمانی ایجاد می‌شود که فرآیندی وجود درخواست ایجاد آن را بدهد. جواب ایجاد فرآیند یک System Call است
ایجاد فرآیند (point) توسط برنامه درخواست یک فرآیند دیگر است (توسط فرآیند دیگر)
وقتی به شاخه بروی یک Icon کلیک می‌کنیم فرآیند ایجاد نمی‌شود و ایجاد فرآیند اصلا توسط کاربر ایجاد نمی‌شود.
ایجاد فرآیند (point) توسط فرآیند دیگر ایجاد می‌شود و راه دیگری نیست.

اگر create فقط توسط فرآیند ایجاد می‌شود پس فرایند اول چطوری ایجاد می‌شود؟ } ایجاد آن کاربر چی کاره است؟

هرنوع درخواستی باعث می‌شود برنامه به پردازنده از دست بدهد حق درخواست create و تمام این تغییر حالت‌ها بهت از دست دادن پردازنده می‌شود.

Subject:
Year. Month. Date. ( )

جلسه هشتم 7/19



ممکن است چند تا فرایند در این حالت باشند پس صفی در این حالت است که نسبت به اختیار گرفتن پردازنده رقابت می‌کنند

وقفه پایان درخواست.

اینجا هم صف‌هایی منتظرند تا جواب درخواست خود را بگیرند پس اینجا هم صف داریم

تغییر حالت { وقفه / درخواست

بعضی از تغییر حالت‌ها عکس است برای مثال هم وقفه، هم درخواست باعث

سیستم عامل توسط وقفه یا درخواست به کار می‌افتد

← ایجاد فرایند توسط فرایند دیگر. تحت کدام Active گردان یک برنامه منتظر است که یک برنامه که در حالت running است این درخواست را بکند

به عنوان مثال می‌توانیم برنامه‌های مختلف فرایند با حالت اجرای داشته باشیم. چه جوری؟

یک فرایند داریم که دائماً در حال اجراست (فرایند ناظر) یا monitor

← فرایند مانیتور ورودی را چک می‌کند مثلاً وقتی یک کلید ماوس را می‌بیند فرایند جدید مربوطه برای ایجاد می‌کند

کار جالبی است و کار پیچیده‌ای ندارد و فقط فرایند ایجاد می‌کند (برای سرویس دوره‌ای‌هایی که می‌خواهیم اجرا کند) چون فقط کارش ایجاد فرایند است پس اگر درخواست در حالت Hacked نیست و کاری انجام نمی‌دهد.

برنامه monitor وقتی که دوره‌ای آمد توسط OS پیدا کنید که دوره‌ای سنگ‌دوره پردازنده در اختیارش قرار داده می‌شود.

← فرایند چیست؟ برنامه در حال اجرا یعنی چی؟ OS چطوری اینا را تشخیص می‌دهد؟

وقتی برنامه به فرایند تبدیل می‌شود OS باید منابعی را به آن بدهد. ساختارهای داده‌ای برای آن بسازد. OS برای هر فرایند ساختار داده‌ای می‌سازد و همین است که باعث می‌شود OS فرایندها را بشناسد. این ساختار داده‌ای دارد که به آن

Subject:
Year. Month. Date. ( )

PCB (process control Block) (بلوک کنترل فرایند) یک ساختار داده‌ای است که حاوی اطلاعاتی است که OS برای مدیریت فرایند به آن نیاز دارد. هر فرایند در سیستم عامل توسط یک PCB نمایش داده می‌شود. وقتی فرایندی ایجاد می‌شود OS یک PCB برای آن ایجاد می‌کند و وقتی فرایندی خاتمه می‌یابد PCB آن حذف می‌شود و منابع آزاد می‌شوند.

داخل PCB چی هست؟

- حالت فرایند Process state
- برنامه شمار PC program counter
- رجیسترها CPU registers
- اشاره‌گر پشته stack pointer

برای اینکه بخواهیم اجرای برنامه چی هست و وقتی اجرای برنامه‌ای کم می‌شود تا اجرا شود باید به این اطلاعات نیاز داریم پس PC را در این Save می‌کنیم

برای اینکه اجرای برنامه بتواند از اینجا دوباره CPU را در اختیار بگیرد برنامه بتواند اجرا شود

- اطلاعات مدیریت حافظه Memory management information ← رجیسترهای مربوط به reg هایی دیده نشدند
- اطلاعات زمان‌بندی پردازنده CPU scheduling Info ← باید این را داشته باشیم تا بفهمیم چقدر در صف قرار بگیرد تا بفهمیم باید چه زمانی اجرا شود
  1. آیا اولویتی دارد
  2. درجه اولویت آن چی هست
- اطلاعات حسابرسی Accounting Info ← تا به حال چقدر از منابع استفاده کرده، چقدر زمان پردازنده را گرفته، چقدر از disk استفاده کرده

هر برنامه‌ای که پردازنده را در اختیار می‌گیرد باید اطلاعاتش در پردازنده load شود تا برنامه بتواند اجرا شود و وقتی هم که می‌خواهد پردازنده را از برنامه بگیرد اول اطلاعات جدید را در PCB ذخیره می‌کند. پس PCB در واقع زمان تنفس می‌کند.

وقتی که PCB از بین می‌رود هر اطلاعات مربوط به برنامه از بین می‌رود. پس دیگر برنامه‌ای وجود ندارد پس لزومی ندارد که منابع را هم آزاد کنیم چون دیگر کسی از آن منابع استفاده نمی‌کند پس با حذف PCB منابع آزاد شده، حساب می‌شوند

ایجاد فرایند = ساخت یک PCB ← (point)

خروج از فرایند = حذف PCB

Subject:
Year. Month. Date. ( )

ایجاد فرآیند (point) : یکی از متدهای ارث از OS

یک ساختار داده‌ای برای هر فرآیند ایجاد می‌شود

← فرآیند اول چطور ایجاد می‌شود؟

مجبوریم از جنس "فرآیند فقط توسط فرآیند ایجاد می‌شود" دور بیاییم. فرآیند اول دستی یعنی توسط کاربر ایجاد می‌شود

(دستی ایجاد کردن فرآیند = روشن کردن سیستم ، reset کردن سیستم)

reset کردن یعنی چی؟ reset کردن یعنی به حالت مقدار صفر است (حالت مقدار صفر) می‌گوییم صفری شود) به همه مقدار برمی‌گردد PC ، SP ، reg و ... هر یک عدد خاص (حالت مقدار) می‌شود

PC = ...

SP = ...

reg = ...

← MMU = ... → memory management Unit

ساختن یک PCB به صورت دستی

در واقع با reset کردن این ها داریم یک PCB درست می‌کنیم. همه اطلاعات را برای PCB تعیین می‌کند و اطلاعات لازم را تامین می‌کند همیشه به حالت مقدار خاصی برمی‌گردد راه‌کار ندارد در دو دفعه reset کردن دو تا مقدار متفاوت برداشته شود

این فرآیند اول اصلا جزئی از OS نیست چون این فرآیند پردازنده را در اختیار OS قرار می‌دهد و باعث ایجاد فرآیند OS می‌شود که بردارج جزئی از BIOS است. یعنی در واقع اول OS را داخل حافظه قرار می‌دهد بعد از آن کنترل پردازنده را به OS می‌دهد بعد از این همه "فرآیند توسط فرآیند ایجاد می‌شود" و کار خواهیم بود

← هر یک فرآیند است که توسط PCB که اول ساخته می‌شود create می‌شود.

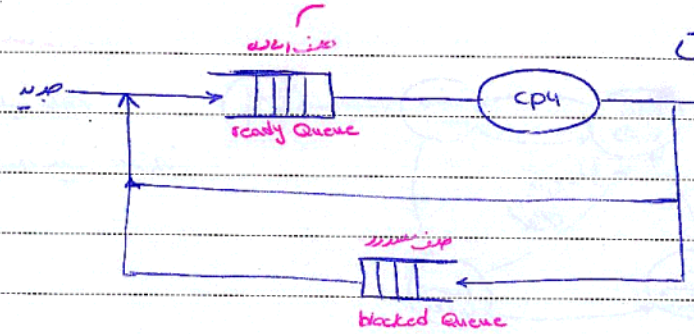
← PCB که اول به صورت دستی ساخته می‌شود همیشه باید آدرس اجرای همه OS را در جای خاصی از حافظه قرار دهد و بنابراین همیشه آدرس شروع OS ثابت است

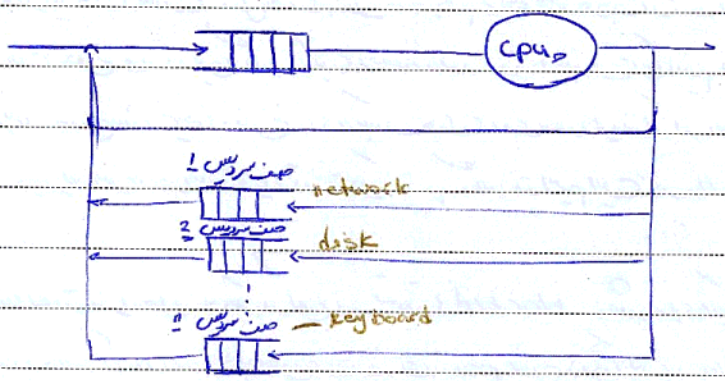
← این PCB همیشه ثابت است برخلاف PCB‌هایی که توسط OS ساخته می‌شود

← OS اطلاعات حیاتی خود را در جدول حافظه دارد و بردارج PCB برای OS همین است و نیازی ندارد به ساخت PCB ندارد. OS وقتی درخواست را می‌گیرد آنرا تحلیل می‌کند متوجه پردازنده را از دست می‌دهد بعد نیازی از جنس PCB ندارد.

Subject:
Year. Month. Date. ( )

دیاگرام زمانبندی: همان شکل اول جلسه است





چون سرویس ها با هم تفاوت دارند مثلاً keyboard یک سرعت part ... است برای هرکدام از این سرویس ها یک صف داریم. کسانی که سرویس کیبرد می خواهند باید با هم یک صف داشته و معنی ندارد که کسانی که منتظر سرویس های مختلف هستند یک صف باشند

مدیریت کنترل تمام این صف ها با OS است. کسانی که در صف هستند اصطلاحاً دانند در صف هستند منتظر جواب درخواست خود هستند. چون پایان دقیق درخواست توسط دیسک است پس هیچ گاه دو تا صف با هم جواب نمی دهند چون حتماً یک دقیقه بعد یک خط سیگنالی می زند.

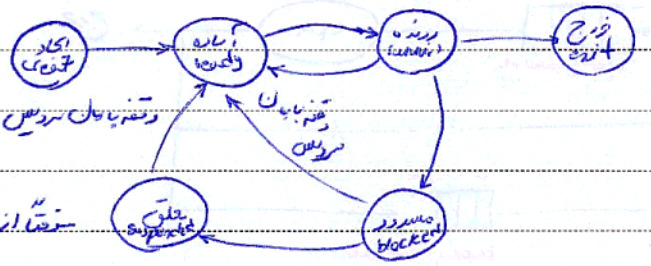
زمانبندی: کد داده پشته، PCB

وقتی گفتیم فرایند در صف است پس باید همه این صف ها در آن فرایند به این صورت نیست بلکه در واقع PCB فرایند است در صف است که تازه این هم که چون PCB بزرگ است تمام PCB در صف نمی رود بلکه فقط PC در صف پشته ها می است

Subject:
Year. Month. Date. ( )

حاسه هشتم.

تغییر شکل قبل برای آنکه فرایندهای بیشتری بتوانند اجرا شوند



سعتا از حافظه خارج می‌شود.

تا اینجا برای فرایندهایی است مثل حافظه یکی مدیریتی خواهد چندتا فرایند کاربری خواهد اجرا شود جدول مدیریت کمی درست است درحالت تعلق یا در زمانی که فرایندی مسدود بود سعتا پیدا می‌کنیم داده‌هایی را، اطلاعاتی که خواسته بودن برسد خارج شدن یعنی از دستی برای سرویس چون برای همه‌ی فرایندها نمی‌توانند باشد بلکه باید درجایی ذخیره شود تغییر حالت صدا بدهد براساس درخواست یا وقفه تصمیم بگیریم اگر نتوانیم این کار را انجام دهیم هرچه‌ی آنها تعلیق است

این شکل تابعی‌ای توانند راه‌حل دهند کلی در ست blocked باشد برای آنکه فرایندی درحالت انتظار نداریم دهم فرایندی درحالت آماده باشد. هم‌چنین حافظه‌ای نداریم پس دیگر کلی درحالت Hacked نیست ای‌گرایم حافظه آن را بگیریم در نتیجه حافظه‌ای نداریم به فرایند بدهیم.

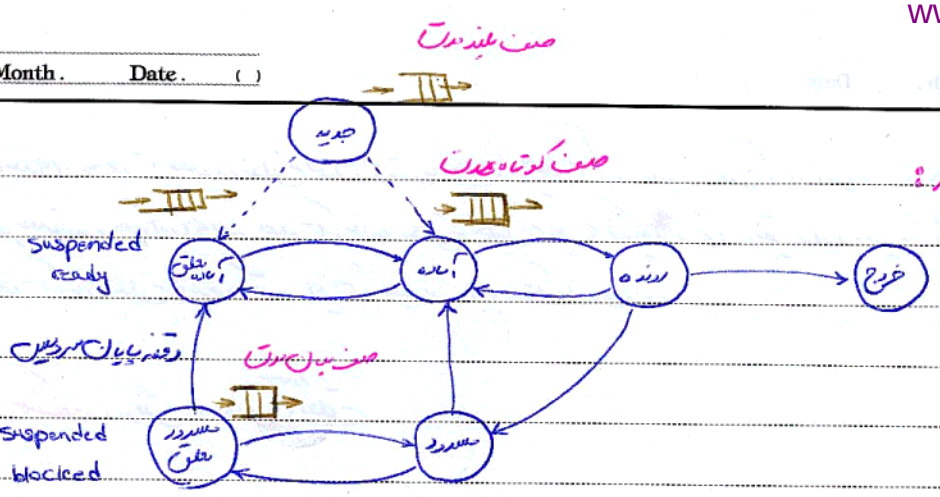
← این راه‌حل برای حداری است همواره فرایندهای درحالت مسدود باشند

چه موقع از حالت مسدود به حالت تعلق می‌بریم؟

- درخواست جدید
- وقفه پایان سرویس

← این شکل دیدیم فرایندهای بیشتری را در آن واحد اجرای کلی می‌خواهیم جواب می‌دهد.

Subject:
Year. Month. Date. ( )



مسدود معلق ← وقتی فرآیند جدیدی خواهد وارد سیستم شود. اگر این نگاه کنیم می‌توانیم اگر حافظه نداشته باشیم از حالت مسدود به حالت معلق می‌آییم

آماده معلق ← اگر در حالت مسدود چیزی برنگشتیم از حالت آماده استفاده می‌کنیم و از حالت آماده کسی به معلق می‌کنیم اولویت با برای معلق کردن از حالت مسدود است اگر در حالت مسدود کسی نباشد از آخر صف حالت آماده استفاده می‌کنیم چیزهایی که در حالا حالا ها به نیاز دارند می‌شود.

← هرچه فرآیند زیاد شود کارایی سیستم خیلی کم می‌شود. زمان بسیار زیادی می‌برد تا یک فرآیند را از حافظه به رسیم. بالاس بیرد و حال اگر تعداد حالات زیادی این طور انجام دهد سیستم کندی می‌شود و دائما صدای دیسکی به صدا میاید چقدر فرآیند می‌تواند اجرا شود تحلیل بعدی است اینکه چقدر می‌تواند به کار خود ادامه دهد. OS نباید برای این کار بدرد بخور باشد اگر درست طراحی نشده باشد مثل OS های microsoft که درست طراحی نشده‌اند اگر تعداد فرآیندها زیاد شود تحمل می‌کند. crash می‌کند

← حالا کسی را از حالت مسدود به حالت مسدود معلق می‌بریم انتظار بازگشت به آن را بدهیم و منتظر می‌شویم که حتما وقفه پایان سرویس بیاید و این حافظه‌ای که شد آن را به حافظه می‌دهیم. اگر وقفه پایان سرویس آمد باید از حالت مسدود معلق به حالت آماده برود و کاری که آماده معلق کردن است چیزی که شد به آن برود اگر نرود بعد از آن به حالت آماده می‌رود. اگر کسی در حالت مسدود نباشد به جای حالت آماده می‌رویم از آن به حالت آماده معلق که نیست دیگر انتظار بازگشت به آن را بدهیم

Subject:
Year. Month. Date. ( )

اگر فرایندی در حالت مسدود بقیه بود و مقدار پایان سرویس‌اش رسید به حالت آماده برود و در حالت آماده هم کسی منتظرش نیست و هم کسی را در حالت آماده بقیه ندارد باید منتظر بماند پس دوباره به حالت آماده برود. اگر در این حالتی نبود به حالت آماده می‌رود.

← ترجمه این شکل: رفتن از حالت مسدود به حالت آماده

توجه شکل: صف‌های متعددی که تا اندازه‌ی سطح سرچوب باشد وجود دارد چه به چند صف نیاز است چه زمانی باید صف create شود و چه سیاستی برای این صف‌بندی پیش گیریم؟ الگوریتم صف‌بندی، زمان‌بندی، نوبت‌بندی (scheduling)

برای اینکه از حالت مسدود به سمت آماده برویم هم بهتر است صف باشد چون اگر نباشد نمی‌دانیم point random انتخاب کنیم و باید بررسی کنیم همه‌ی صف

صفی که در حالت آماده است صفی هم است که زیاد بررسی می‌کند و OS دارد که به این صف کوتاه‌مدت می‌گوییم (short term queue)

نزدیک صف مسدود ← صف میان مدت mid term queue

" " آماده ← بلندمدت long term queue

اسم‌گذاری این صف‌ها بر اساس مدت زمانی است که صبر می‌کنیم

short term queue ← در زمان کوتاه (چند میلی‌ثانیه) صبر می‌کنیم یا نه و اصلا زمانی که آماده شد آن را می‌گیریم

mid term queue ← در زمان‌های متوسطی به آن‌ها سرکشی می‌کنیم یا نه و اصولا طولانی‌تر نسبت به کوتاه‌مدت

long term queue ← مدت خیلی زیاد به آن‌ها سرکشی می‌کنیم

صف نوبت‌بندی: فرایندی در حالت آماده، هیچ‌کدام از کم‌زمان‌ها را ندارند چون کم‌زمان کوتاه است پس نمی‌شود به آن سری بزنیم. اگر کم‌زمان باشد حداکثر زمان کم‌زمان طولانی است اگر کم‌زمان نباشد اجرا پیوسته است و چقدر تا زمانی است که یک درخواست بدهد البته حالت‌هایی است که کوتاه نباشد که جزئیات در این کوتاه است.

Subject:
Year. Month. Date. ( )

برای دوتایی بیلد بازای هر بار صدا شدن برنامه برای بخوابد درصورتی که در صف کوتاه مدت پس از هر بار خالی شدن برنامه برای آن کار داریم

چرا در صف ایجاد صف بلندمدت داریم؟

اگر دقت داشته باشیم وقتی برنامه را اجرا می کنیم داده سیستم میشنود که باید برنامه را در حافظه بارگذاری کند و فرآیندهای لازم را ایجاد کند سپس برنامه را اجرا کند اما صف ایجاد وجود ندارد اگر چنین صفی داشته باشیم هر بار که برنامه ای اجرا می شود باید به این صف اضافه شود تا سیستم عامل آن را بخواند و اجرا کند.

سیاست ها: معمولا برای صف نگاره سیاست های زمان بندی به کار برد و برای صف های بلندمدت و میان مدت از این سیاست ها استفاده می کنند چون زیادی سیستم تاثیر ندارد و دقیق ترین سیاست صف بندی برای صف های بلندمدت و میان مدت از سیاست FIFO استفاده می کنند (دقیق صف) چون مراجعه به دیسک است پس تاثیری زیادی سیستم ندارد

میان مدت	FIFO
بلندمدت	FCFS First Come First Served

برای اینکه پیاده سازی سیاست FIFO ساده است بخواهیم برای طراحی صف پیچیده نخواهیم چون تاثیری زیادی کارایی سیستم ندارد از همین سیاست ساده استفاده می کنیم چون مراجعه چند کم است همینقدر آن نیده ای ندارد

چون که نیاز به صف های کوتاه مدت برای ما پس همینقدر این صف زیادی کارایی سیستم تاثیر دارد پس زیادی همینقدر این صف تلاش می کنیم

الگوریتم های زمان کنترل برنامه:

- non preemptive (بدون پیش)
- preemtive (با پیش)

non preemptive . برنامه را تا وقتی که تمام شود یا خودش برنامه را پس بدهد کنترل را در این الگوریتم های اینده

preemptive . برنامه را گرفته و پس میدهد

Subject:
Year. Month. Date. ( )

point ← این ایده کم زمانی برای کاربرست چون که هر بار ای کاش نتیجه کار خود را بگیرد روی بیشتر هم سیستم‌ها یک کاربر هستند است مثل ماشین حساب‌ها. اکثر سیستم‌های کامپیوتری کارکی ندارد عمر این رده سیستم‌های کامپیوتری embeded است ← پس در نتیجه چون است بسیاری از OS ها نسخه non-preemptive ای استفاده می‌توان گفت این الگوریتم‌ها کاربرد استفاده است.

← OS های که نسخه non preemptive ها هستند می‌توان پردازنده را از انحصار درآورد پس می‌توان از حالت انحصار بیرون آورد. اما نسخه‌های OS های که نسخه preemptive هستند در کارهای توان از حالت انحصار نسخه حالت آماده است

معیارهای مقایسه روش‌های زمان‌بندی:

1- زمان انتظار (waiting time) : از زمان‌هایی که فرایند واردی شود چه مدت در حالت انتظار بوده و پردازنده را در اختیار نداشته تا زمانی که خارج شود یکی از معیارها که چون زمان انتظار است.
کل زمان‌های که پردازنده را در اختیار نداشته

2- زمان کل (Turnaround time) : کل زمان running + انتظار و ... را چه چون بین از زمانی که وارد می‌شود تا زمانی که خارج می‌شود چقدر طول کشد که هستی باید به کل زمان انتظار کم شود این هم خود به خود کم می‌شود

3- زمان پاسخ (response time) از زمانی که وارد سیستم می‌شود تا زمانی که اولین بار پردازنده را در اختیار بگیرد مدت زمان لازم برای اولین بار سرویس پردازنده کسانی هستند که از همه بیشتر منتظر بیشتر تا زمانی که پردازنده را در اختیار بگیریم مطلوبی کاهش

point ← کی کمتر است بهتر ولی بعضی وقت‌ها این کار همیشه ندارد باید ببینیم برای کاربرد خاصی کدام از آنها بهتر است

4- استفاده از پردازنده (processor utilization) : کاری کنیم که پردازنده اگر حواسمان نباشد کاری را انجام دهد مثل استفاده در سیستم کامپیوتر استفاده از پردازنده در حالت چند تایی بهتر است باید اولویت را بین این‌ها چی هست.

Subject:
Year. Month. Date. ( )

B

5_ توان عملیاتی (Throughput) : اصطلاح Throughput یعنی تعداد کارهای انجام شده در واحد زمان. اینکه چقدر انتظار داشته باشیم یک کار چه‌قدر طول بکشد، چقدر کم یا زیاد باشد، کم‌بیش کم‌تر مقدار کارهای بیشتری انجام شود این‌ها برای سیستم‌های interactive کم‌بیش

← کاربرد آن در زمینه‌های دیگری هم هست مثل: عدالت :D

جلسه نهم: 7, 21

الگوریتم‌های زمان‌بندی پردازنده: (Process Scheduling)

بدون وقفه ← non-preemptive ← اینجوری هست تا خودش پردازنده را رها نکند

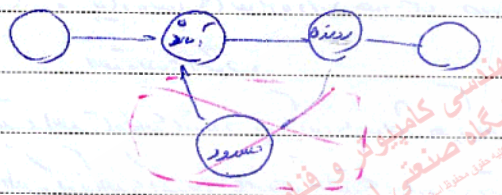
با وقفه ← preemptive ← پردازنده را از دست می‌دهد

1_ FCFS ← روش‌های بدون وقفه

مثال:

فرایندها	A	B	C	D	E
زمان ورود به سیستم	0	2	4	6	8
زمان سرویس (T_S)	3	6	4	5	2
زمان کل (T_T)	3	7	9	12	12
زمان انتظار	0	9	13	18	20

روش ابتدایی‌ترین روش زمان‌بندی است FCFS است یعنی هرکه زودتر وارد شد زودتر سرویس می‌گیرد. (مثال خیلی ساده است یعنی یک سرویس گرفت نوبت نوبتی‌ها می‌شود) یعنی چیزی نمی‌گذرد صف تشکیل می‌شود.



A در زمان صف‌بندی می‌شود و چون کسی پردازنده را ندارد کارش را انجام می‌دهد و بعد از 3 واحد زمان رها می‌کند چون کارش تمام شده. B در 2 وارد می‌شود ولی چون تا 3 پردازنده در دست A است باید صبر کند تا کار A تمام شود. وقتی در 5 وارد می‌شود باید صبر کند تا کار B تمام شود یعنی در 9 کارش شروع می‌شود با اینکه در 4 وارد شده است.

P4PCO

Subject:
Year. Month. Date. ( )

نرخ نرمالیزه‌ی ترین کالیم (T_T/T_S)

میانگین	E	D	C	B	A	
2.56	6.0	2.90	2.25	1.17	1	(T_T/T_S)

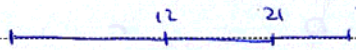
بنابراین در این نسبت ۱ نزدیک‌ترین نشان‌دهنده‌ی انتظار در سیستم تا راست‌ترین زمان انتظار کم باشد این عدد به یک نزدیک‌تر می‌شود و از بین اعداد میانگین یکی هر چه یک نزدیک‌تر باشد بهتر است

مشکل این روش (FCFS) با مثال زیر معلوم می‌شود

ترتیب ورود		ورود	زمان سرویس (T_S)	T_T
	A	0	3	3
	B	0	9	12
↓	C	0	12	24

میانگین (T_T) → $\frac{3+12+24}{3} = 13$

ترتیب‌های دیگر را امتحان می‌کنیم C→B→A



$\frac{24+21+12}{3} = 19$

اولی بهترین ترتیب است چون زودتر سرویس می‌دهد و زودتر خروجی‌ها بیرون می‌دهد

نتیجه چون این ترتیب‌ها تصادفی است پس نمی‌دانیم جواب این روش به تصادف بستگی دارد و تضمینی روی خوب بودن جواب نشان وجود ندارد

اشکال ← تضمین وجود ندارد

خوبی ← پیاده‌سازی ساده دارد چون یک صف ساده است و به همین دلیل خیلی وقت‌ها از آن استفاده می‌شود.

این روش بهترین است ولی نمی‌دانیم چه زمانی باید هر فرایندی کار اول شده باشد زودتر سرویس می‌دهد

ابتدا می‌خواهیم کاری کنیم که فرایندی که زمان سرویس کمتری دارد زودتر اجرا شود زمان کمتری خواهیم راست‌ترین قابل رسیدن است یعنی آن‌هایی که کوتاه‌تر است زودتر سرویس دهیم میانگین کمتری خواهیم داشت.

Subject:
Year. Month. Date. ( )

2- کوتاه‌ترین فرایند بعدی (Shortest Process Next) SPN

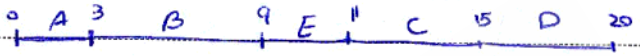
این روش بهینه است

مثال قبلی این روش را استفاده می‌کنیم

	A	B	C	D	E	میانگین
زمان ورود	0	2	4	6	8	
زمان سرویس	3	6	4	5	2	
زمان پایان	3	9	15	20	11	
T_r	3	7	11	14	3	7.60
T_r/T_s	1	1.17	2.75	2.8	1.5	1.79

در لحظه 0 هنوز A است و رقابتی وجود ندارد در زمان 3 هنوز B است پس باز هم رقابتی وجود ندارد در لحظه 9 هر سه فرایند وجود دارد سیستم می‌تواند در اینجا باید کوتاه‌ترین را انتخاب کنیم و به همین ترتیب

در اینجا انتظار داریم میانگین‌ها از روش قبلی بهتر باشد. همان طور که انتظار داشتیم کمتر شد و به دست آمد که جواب بهتری از بقیه دارد.



زمان سرویس از کجا معلوم است؟ علم نیست از کجا بدانیم CPU از کجا بداند که باید به سیستم OS از کجای برنامه

اشکالات ← 1- از کجا زمان سرویس را قبل از اجرا شدن کامل برنامه بدانیم زمان بهینه را می‌آید و شدنی دارد ولی روش‌هایی برای تخمین وجود دارد که کامل است از راه‌های دور باشد. زمان سرویس را از این خوش هیچ‌کدام نمی‌توانیم چیزی بزنیم راه‌حل!

2- حالتی پیش می‌آید که زمان سرویسی طولانی دارد و از بدشانسی آن همیشه زمینه‌ی بعدی زمان سرویس کوتاه‌تر وجود دارد همین این زمینه‌ی بعدی افتاد و این اتفاق وجود دارد که نوبت به این نرسد اصلا. به این حالت starvation یا ... اتفاق غیرخوبی برای این وجود دارد که هیچ سرویسی به زمینه داده نشود.

Subject:
Year. Month. Date. ( )

1 point اصول الگوریتم های اولیت داری توانند به شکل بهتر بیشتر شوند

سعی کنیم مشکلات را حل کنیم. راه حل روی حساب است و اینست که الگوریتم را عوض کنیم

حل مسئله اول:

روش هایی برای تخمین وجود دارد که با این حالت سرباز سیستم زیادی شود.

برای تخمین زمان سرویس

میانگین نمایی: $S_{n+1} = \alpha T_n + (1-\alpha) S_n$

واقعیت اینست که یک زمانید بارها سرویس می‌گیرد و یکبار سرویس گرفتن عادی عادی می‌شود. ما می‌گوییم چندین بار است که سرویس می‌گیرد. این تخمین را با استفاده از سابقه زمانیدی زنیم

$0 < \alpha < 1$ ← α

S ← تخمین

T ← زمان واقعی

به این علت می‌گوییم نمایی که می‌شود این رابطه را داده

$S_n = \alpha T_{n-1} + (1-\alpha) S_{n-1}$

برای اینکه از رابطه اصلی استفاده کنیم برای اینکه بهتر بفهمیم چی می‌گوید از این رابطه می‌بینیم

$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \dots + (1-\alpha)^i \alpha T_{n-i} + \dots + (1-\alpha)^n S_1$

توان به رابطه اهمیت اعداد کمتر از یک کم می‌شود این یعنی تاریخچه زمانید را دارد. و زمانی که اخیراً انجام داده اهمیت بیشتری دارد تا زمان‌های گذشته. می‌گوییم زمانید بیشتر نیست. α چگونه شود الان حکم می‌شود.

هرچه α را یک نزدیک کنیم ← زمان‌های اخیر تاثیرشان خیلی زیاد می‌شود. زمان‌های قدیمی از بین می‌روند یعنی

قضیه زمان‌های اخیر زیاد می‌کند

هرچه α را صفر نزدیک کنیم ← کل تاریخچه را در نظر می‌گیرد. قدیمی‌ها را هم زیاد می‌کند

Subject:
Year. Month. Date. ( )

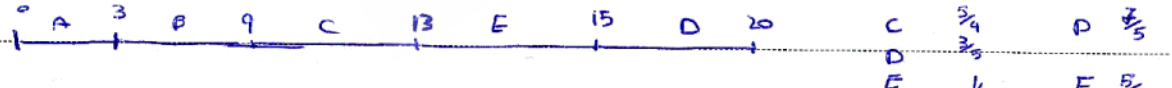
کسری که این کسری باید علامت کند نه نیم بیشتر نباشد پس بهتر است $\alpha = 1/2$
باین هم مشکل حل نمیشود چون کامپیوتر Floating point (Floating point) نیست و اگر این را بیل
بالاخره رقم حلی زمان بری شود
$\alpha = 1/2$ ← همانطور که دیدید
کاستی تنظیم Shift است و زمان بر است
← مشکل اولی را ترمیم کردیم با اینکه حل دقیق نکردیم

حل مشکل دوم:
این مشکل را کاملاً برطرف کردیم احتمالاً از این الگوریتم استفاده نمی‌کنیم بنابراین الگوریتم را اصلاح می‌کنیم
پس:

3- HRRN Highest Respone Ration Next
جلوی گرسنگی جاهای آخر را میگیرد جای به خوری SPN خواهد درست و جلوی سوء چیزی را اصلاح می‌کند پس
باید هزینه‌ای بدهد
از نیاز بیشتر صحبت می‌کند علاوه بر Ts زمان انتظار را در نظر می‌گیرد به انتظار اینها که در SPN برش
اضافه هم بودند چند انتظار کشیدی
امتیاز برای انتظار T_w / T_s
هر چه بیشتر انتظار بکشی امتیازت بالاتر می‌رود و هر چه زمان کوتاه‌تر باشد باز هم امتیازت بالاتر می‌رود

← برای مثال قبل

	E	D	C	B	A	
	8	6	4	2	0	زمان ورود
	2	5	4	6	3	T_s
	15	20	13	9	3	زمان پایان
8.0	7	14	9	7	3	زمان گردش (T_r)
2.14	3.5	2.8	2.25	1.17	1	(T_r/T_s)



Subject:
Year. Month. Date. ( )

نباید بین دو بیش برخوردی قبلی باشد چون ما خواهیم از سبقی را از بین ببریم پس باید هزینه‌ای بدهیم.
انتظار داشتیم که از FCFS بهتر شود چونکه FCFS تقاضایی است و کاربران از این بهتر شود.

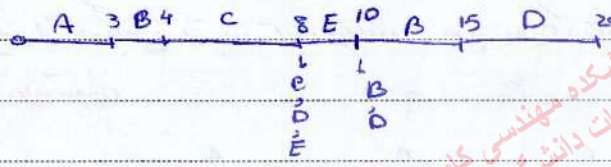
اگر بین خواصاندن‌های non-preemptive استفاده کنند انتخاب بهتر بیش و بهره‌وری FCFS است
چون SPN که مشکل گرسنگی دارد و HRRN هم برای کسانی زیاد و سربار بالایی باشد.

روش‌های preemptive :

4- SRT Shortest Remaining Time (تقدم)
هر لحظه می‌توان فرآیند را نقض کرد. همین الگوریتم SPN را اجرا می‌کند اما در هر لحظه اجرا می‌کند

E	D	C	B	A
8	6	4	2	0
2	5	4	6	3

در اینجا در لحظه 2، 2 تا انتخاب خواهیم داشت چون کار باقیمانده را از فرآیند تمام چون A کوتاه‌تر است
از انتخاب می‌کنیم در لحظه 3، B را داریم و در لحظه چهارم B و C که B چون یک واحد از آن انجام شده 5 تا مانده و
چون 3 تا برای C کمتر از 5 است پس
B نقض می‌شود و C اجرای شود.



انتظار داریم از SPN هم بهتر شود
در هر لحظه spn را اجرا می‌کرد و آن فرایندی که زمان کمتر رسیدن را اجرا می‌کند

Subject:
Year. Month. Date. ( )

جلسه هشتم: 7/23

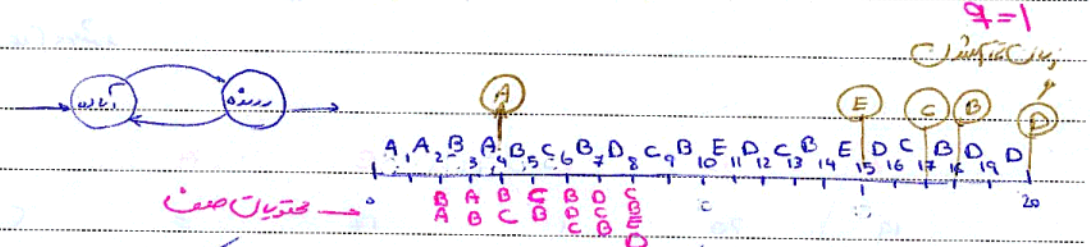
۵- روش کم‌ترین زمانی (پیش‌فرض)

RR (Round Robin)

	E	D	C	B	A
زمان ورود	8	6	4	2	0
زمان سرویس	2	5	4	6	3

درون دولت بررسی می‌کنیم ← $q=1$ ← زمان نسبتاً کوچکی است

$q=?$ ← زمانی که از بعضی از اینها بزرگ‌تر است



در زمان ۰، نقطه A است و کم خودرای ایجاد و به حالت آماده‌ی رود می‌شود و صف کسی نیست پس دوباره به حالت پردازنده می‌رود و یک واحد کار انجام می‌دهد و از کار آن هم ۱ واحد باقی می‌ماند. B هم آمده و این فرایند هم به حالت آماده می‌رود و صف می‌شود. حال A هست و B هم هست اما A سرویس گرفته پس B سرویس می‌گیرد (خاصیت هم‌زمانی عاملانه بودن آن است). اگر فرایند وارد شده در همان لحظه از حالت پردازنده به حالت آماده‌ی خواهد رود اول از همه صف می‌شود.

سیاست	E	D	C	B	A	
	15	20	17	18	4	زمان پایان
10.80	7	14	13	16	4	زمان T_T کل
2.71	3.80	2.8	3.25	2.67	1.33	T_T/T_S

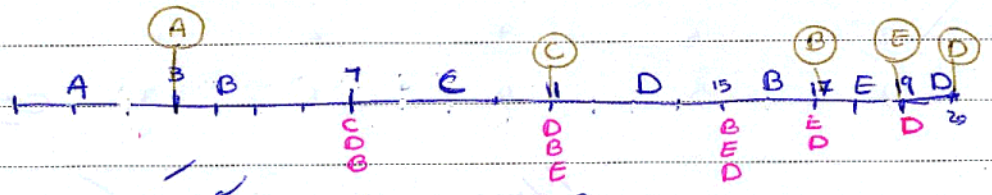
در سیاست با اعداد سیستم حالی بهتر است ← طبیعی است چون زمان بیشتری را در سیستم صرف کرده و همین که می‌باید به صف آماده برگردد زمان بیشتری خواهد.

اما در کل زمان پاسخ این روش بهتر است اولین بار که سرویس می‌گیرد قبلاً در روش‌های قبلی هیچ وقت این‌قدر نمی‌شد در زمان 5 به پاسخ می‌رسید و هیچ‌چنین در این روش تضمینی وجود ندارد.

P4PCO

Subject:
Year. Month. Date. ( )

محیط‌های interactive به‌دنبال زمان پاسخ هستیم و همین که به زمان پاسخ این روش خیلی خوب است
این روش به‌بهره‌ی خود دو تا هارد عدد زمان پایان و زمان کل شناسید با آلگوریتم بزنند



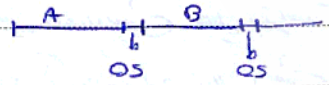
A زمان 4 را نیاز دارد چون در 3 زمان کوانتوم عادی شود در این لحظه به صف آماده‌ی رسیم چون در حین
تنفذ B است 4 تای B را به آن می‌رسیم. سپس که B مشغول بوده C وارد شده و 4 تای C را به آن
می‌رسیم و C عادی شود

	E	D	C	B	A	
	19	20	11	17	3	زمان پایان
10	11	14	7	15	3	T_T کل زمان
2.71	5.50	2.80	1.75	2.5	1	T_T/T_S

اگر باز هم به زمان پاسخ نگاه کنیم می‌بینیم زمانی که $q=1$ است خیلی بهتر است و اگر q را بزرگ بگیریم
همان FCFS می‌شود حتی به کم زمان‌های اصلاً توجه نکردیم

اولاً) اگر quantum کوچک شود شبیه SPN می‌شود که در SPN نزدیک‌ترین زمان می‌شود و در
این جا هم همین طور است ولی دلیلی ندارد که همیشه یکسان شود چون از یک آلگوریتم استفاده می‌کنند

بیایی تو انیم q را خیلی کوچک بگیریم چون سربار این طوری بسیار بالا رفته و شکل زیر است



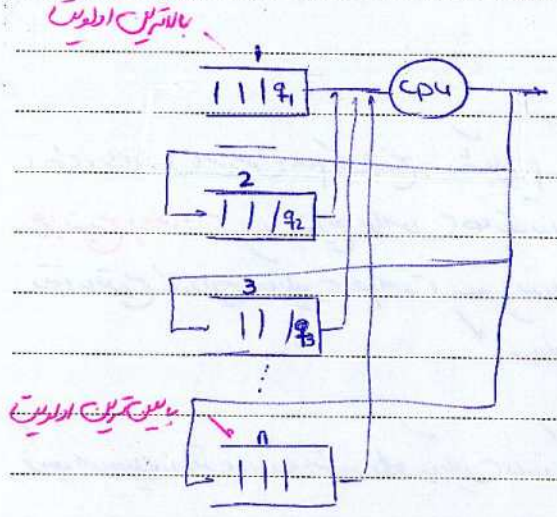
در این مثال از زمان حیاتی که OS کاری انجام نمی‌دهد صرف نظر کردیم
OS زمان بیشتری را برای هدر دهد CPU و نه خود تر می‌کند این کار را می‌کند اگرچه کوچک است ولی همیشه می‌توان از آن
صرف نظر کرد اگر q خیلی کوچک باشد مجبور هستیم زمان OS را هم به حساب بیاوریم.

Subject:
Year. Month. Date. ( )

درهم زنی OS قبل از اینکه پردازنده را به زائد بدهد Timer را صفر می‌کند تا کم زنی به درستی انجام شود.

point 1 q باید طوری انتخاب شود که ۱۰ برابر زمانی باشد که OS نیاز دارد وگرنه خیلی بزرگ می‌شود مثل FCFS می‌شود که خوب است ولی اگر چه زیاد باشد عالی‌تر است مثل SPN می‌شود ولی در سیستم حدودیت دارد.

روش Feedback Queue : (بازخورد)



در این روش به جای یک صف از چندین صف استفاده می‌کند کسی وارد می‌شود وارد صف اول می‌شود و یک q دارد که اگر کافی نباشد به صف دوم می‌رود و یک q دارد که اگر کافی نباشد به صف سوم می‌رود و می‌تواند q دیگری داشته باشد اولویت با صف اول است و تا زمانی که فرایندی در صف اول است نوبت به صف دوم نمی‌رسد. احتمال اینکه پس در صف اول نباشد زیاد است چون پرشت به صف اول نداریم اگر در صف دوم کامل شد چه بهتر اگر نیاز شد به صف سوم می‌رود و می‌تواند q دیگری داشته باشد صف آخر دیگر q ندارد که برود صف بعدی پس باید صبر کند تا از فرایند طولانی و کامل می‌شود

این روش بهترین الگوریتم SPN را اجرا می‌کند چون اولویت با کوتاه‌ترها است اگر فرایندی طولانی باشد ممکن است چهار صفات آن سوئی شود. چون این روش بهترین را اجرا می‌کند دارد SPN را شبیه‌سازی می‌کند

هرچه صف بیشتر شود دقیق OS بیشتر می‌شود

این صف جزء راه‌های OS هستند و در کنار OS ضروری می‌شوند

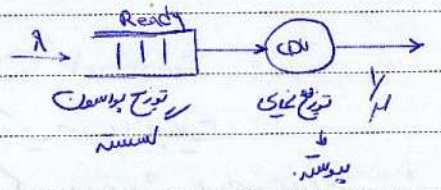
← تا حالا گفتیم روش‌ها را با مثال بیشتر داریم می‌بینیم چطور عمل می‌کنند. برای اثبات از چندین روش استفاده می‌کنیم می‌آید ساده‌ترین نوع آن گفته می‌شود.

الان با روش آماری کار می‌کنیم می‌این بسیار پیچیده است و ما آن را نمی‌گوییم

P4PCO

Subject:
Year. Month. Date. ( )

نیز صف داشتیم و برنامه‌ای که به آن خدمت می‌کرد. سیستم بایستی چه توزیع بازی را برای این در نظر بگیریم
خدمت رسان می‌شود. پس روشی به انتخاب می‌کنیم FCFS است. ساده‌ترین توزیع عمل را برای داریم



از طرفی برای زمان سرویس هم یک توزیع در نظر بگیریم ← توزیع نمایی

چرا توزیع پواسون ← توزیع پواسون هیچ‌وقت ندارد و می‌داند قبلاً چه اتفاقی افتاده است. برایش مهم نیست
وقتی با توزیع کار داریم سیانلین مهم است ← یعنی واحد زمان چند تا درخواست وارد صف می‌شوند (λ سیانلین تعداد)
سیانلین زمان سرویس: درخواست چقدر است ($\frac{1}{\mu}$ سیانلین زمان)

درخواست‌ها اصلی نامرتب ارسال می‌شوند ولی سیانلین دارند چون نامرتب بودن آنها را همان توزیع مشخص می‌کند.

رویدادی دوره درخواست به سیستم است (1, 2, 3, ...) و وصل 2.5 تا درخواست نداریم ولی 5 تا درخواست داریم
زمان بین دو درخواست‌ها چقدر است؟ عدد ندارد بلکه پیوسته است. توزیع آن پیوسته است و تفاوت زمان صدا
توزیع آن مشخص می‌کند

سیانلین دوره

$$P(k) = (\lambda T)^k e^{-\lambda T} / k! \qquad k = 0, 1, \dots$$

احتمال k درخواست در فاصله زمانی T

پست امید سیانلین این توزیع:

$$E(k) = \sum_{k=0}^{\infty} k P(k) = \lambda T$$

واریانس: $\sigma^2 = \lambda T$ ← چقدر از سیانلین دوریا نزدیک هستیم
کم ← خیلی به سیانلین نزدیک، زیاد ← از سیانلین دور هستیم

Subject:
Year. Month. Date. ( )

توزیع نمایی:

τ : زمان سرویس

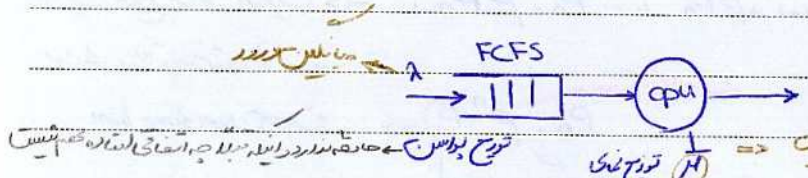
$f(\tau) = \mu e^{-\mu\tau}$

اگر برای مدل‌ها توزیع پواسون داشته باشیم زمان بین ورودی‌ها نمایی است.

$E(\tau) = \int_0^\infty \tau f(\tau) d\tau = \frac{1}{\mu}$ → میانگین

$\sigma_\tau^2 = \frac{1}{\mu^2}$ → واریانس

جلسه یازدهم 28/7/



میانگین ورود

توزیع پواسون ← حافظه ندارد و ایده‌آل است اما واقعی نیست

میانگین زمان بین ورود = 1 / تعداد

میانگین سرویس

تعداد در زمان نسبت عکس دارد.

$\rho = \frac{\lambda}{\mu n}$ ← بهره‌وری

اگر توانایی سرویس دهی بیشتر باشد

$\lambda < \mu$ ← توانایی سرویس دهی باید از تعداد ورود بیشتر باشد

در غیر اینصورت صف بحرانی ایجاد می‌شود و جمع نمی‌توانند بشوند

$\rho < 1$ ← هر چه زمان‌های بیکاری خواهد بیشتر شود

اگر نسبت ورود به نسبت توانایی سرویس مشغول‌تر نیست

مشخصات یک صف گسسته

اگر ظرفیت محدود دارد اگر نتوانند عضویت ادامه پیدا کند

میانگین طول صف

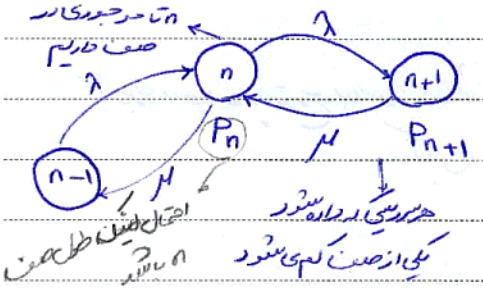
- میانگین زمان انتظار

- " " طول

- احتمال مسدود شدن صف

Subject:
Year. Month. Date. ( )

- میانگین توان عملیاتی (Throughput) ← چند کار در واحد زمان



صف می‌تواند از حالت n-1 به حالت n+1 برود
تغییر چندتایی از آن نمی‌تواند بلکه تغییر حالت یکی یکی است.

تعادل جریان ← $\lambda P_n = \mu P_{n+1}$

خاصیت توزیع پواسن داریم ⇐ $P_{n+1} = \rho P_n$ یا $P_n = \rho P_{n-1}$ ⇐ با استفاده از ترتیب ρ
روابط به راحتی بازنویسی می‌شود،

احتمال خالی بودن صف ⇐ $P_n = \rho^n P_0$ ← عددی ثابت است.

تا اینجا اندازه‌گیری نسبت‌ها را داشتیم حالا می‌خواهیم مقدار واقعی را بدست آوریم
گفتیم صف نامحدود است اما برای سادگی جمله جمله جلو می‌رویم پس صف را نامحدود در نظر می‌گیریم

$$\sum_{i=0}^{\infty} P_i = 1 \Rightarrow \sum_{i=0}^{\infty} \rho^i P_0 = P_0 \sum_{i=0}^{\infty} \rho^i = \frac{P_0}{1-\rho} = 1 \Rightarrow P_0 = 1-\rho$$

تصاعد هندسی

$$E(n) = \sum_{i=0}^{\infty} i P_i = \frac{\rho}{1-\rho}$$ ← میانگین طول صف نامحدود

اینجا برای صف نامحدود بود حالا می‌خواهیم برای صف محدود (طول آن را N می‌گیریم). هرجا نهایت (∞) استفاده کردیم حالا محدودش می‌کنیم و N تا برای جمع و متغیر زیر بدست می‌آید.

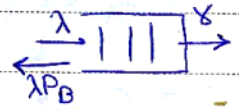
$$P_0 = (1-\rho) / (1-\rho^{N+1})$$
$$P_n = (1-\rho)^n / (1-\rho^{N+1})$$
$$P_N = (1-\rho)\rho^N / (1-\rho)^{N+1}$$ ← احتمال اینکه صف پر شود

Subject:

Year. Month. Date. ( )

$P_B = P_N$ ⇒ چون تمکی که صف پر شده تمیز نرسیده‌ها block می‌شوند

تمکی که صف پر شده نرسیده‌های جدید که وارد می‌شوند پذیرفته نخواهند شد.



توان عملیاتی ← $\gamma = \lambda(1 - P_B)$

* به خاطر توزیع های انتخاب شده تمام اینها از نظیت درست

یک رابطه ای مستقل از نوع توزیع است و همیشه صادق است.

n طول صف

w زمان انتظار

$\Rightarrow \quad n = w \cdot \lambda$ ⇒ رابطه لیتل

از روی این رابطه به راحتی میانگین زمان انتظار پیدا می‌شود:

$$w = \frac{n}{\lambda}$$

$$\text{زمان کل} = w + \frac{1}{\lambda}$$

μ ← میزان پردازنده برای سرویس دادن است پس از این استفاده نمی‌کنیم و از آن به زمان پردازش می‌گذرد استفاده می‌کنیم

تا زمانی که نرسیده وارد صف می‌شود که صف پر نشده نشده و λP_B تا از آنجا نیز خواهند شد

↓ احتمال block شدن

تمرین) از زمان‌بندی HRRN استفاده می‌کنیم

کمترین مقدار $1 + \frac{t_w}{t_s}$ را انتخاب می‌کنیم

زمان سرویس دهی / زمان صدور / زمان پردازش

P_1	t_1	r_1
P_2	t_2	r_2
P_3	t_3	r_3

از این رابطه در حالت کلی استفاده می‌کنیم ولی اینجا برای

ترتیب اجرای نرسیده‌ها:

$t_1 < t_2 < t_3$

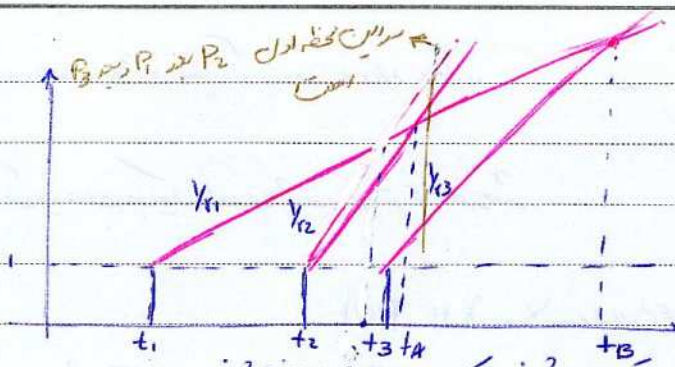
$r_2 < r_3 < r_1$

$1 + \frac{t_w}{t_s}$ ← در واقع نتایج تعیین اولویت است

PqPCO

Subject:

Year. Month. Date. ( )



ترتیب اولویت سه خط همسان چون که به هم جمع شدند هیچ کدام از صف خارج نمی‌شوند

$R_1 = 1 + \frac{t_w}{t_s} = 1 + \frac{t - t_1}{r_1}$ ← سالم‌ترین سیب

t_w: زمان انتظار است نقطه‌ای که از صف خارج کرده بعدی
t_s: زمانی که سرویس‌دهنده

$R_2 = 1 + \frac{t - t_2}{r_2}$ ← تیزترین سیب

$R_3 = 1 + \frac{t - t_3}{r_3}$

در نمودار در یک لحظه اولویت با کسی است که بالاتر است یعنی سیب بیشتری دارد.

$t \le t_A$ ⇒ $\vec{P_1, P_2, P_3}$

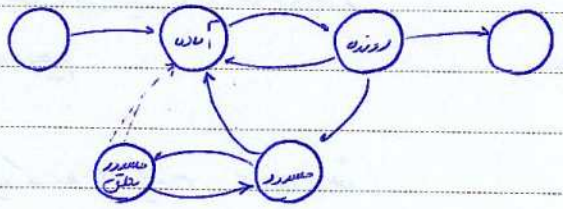
$t_A < t \le t_B$ ⇒ $\vec{P_2, P_1, P_3}$

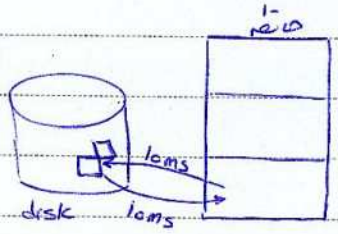
$t_B < t$ ⇒ $\vec{P_2, P_3, P_1}$

وضعیتی که تمام شده بعد از بقیه است اولویت بیشتری دارد

Subject:
Year. Month. Date. ( )

تمرین) مفهوم swaping

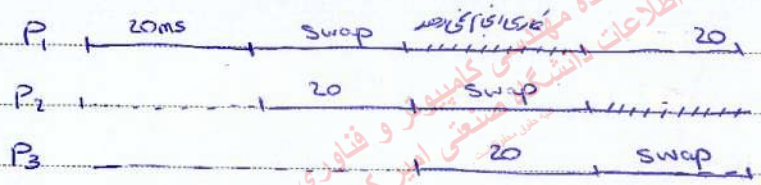




حافظه این سرت partition ثابت دارد بین درخواست 3 ت
زمینه بیشتری تواند در حافظه باشد و اگر بیشتر شد باید بعضی
بسدود بطئی بروند
برای دادن زمینه یکی را از حافظه پیدا میدیم و
زمینه را از دیسک به حافظه میدیم .

زمان دسترسی به دیسک 4ms ← برای هر یکی از اطلاعات است
زمان انتقال 6ms
بین هرکدام از اینها برای انتقال حافظه 1oms طول میکشد (یعنی جمع زمان دسترسی + زمان انتقال)
هم برای جابه جایی زمینه یک زمینه چقدر به پردازش آسیب وارد میشود ؟
به پردازنده هیچ کاری نباشد که برای انجام داشته باشد

برای انتقال هر زمینه باید اول یکی را بیرون کنیم یکی را باید وارد کنیم پس هر انتقال 20ms طول میکشد
زمان یک swap ، 20ms است و برای اینکه زمینه جدید آماده کار شود باید زمینه بیدار اجرا 20ms کار کند پس این
عمل swap اجرا شد که تنها عمل swap اجرا شده زمینه جدید 20ms اجرا دهد .



پردازنده هیچ وقت بیکار نیست و همیشه چیزی برای اجرا است . swap های تو انتخاب هم انجام دهند جدول تا
یک دسترسی به دیسک ها توانیم داشته باشیم

Subject:
Year. Month. Date. ( )

ب) اگر هر کاربر یک دوره‌ی دوران نیز بعد صدا شدن تقاضا کند چه؟
هیچ‌زمان باج نمی‌دهد! ثانیه بیشتر چون اگر بیشتر باشد متوجه می‌شود

$\frac{1s}{20ms} = 50$

در هر ثانیه به 50 کاربر باج می‌دهیم

ج) شغل تل را برای حالتی بکشید که زمان بررسی 10ms باشد

جلسه دوازدهم 7,30

تمرین) دو صف ادرس فرایند سازیم که هم‌زمان اجرا شوند چه
اتفاقی می‌افتد.
کدام یک از دو شخص tally را بیشتر می‌کند

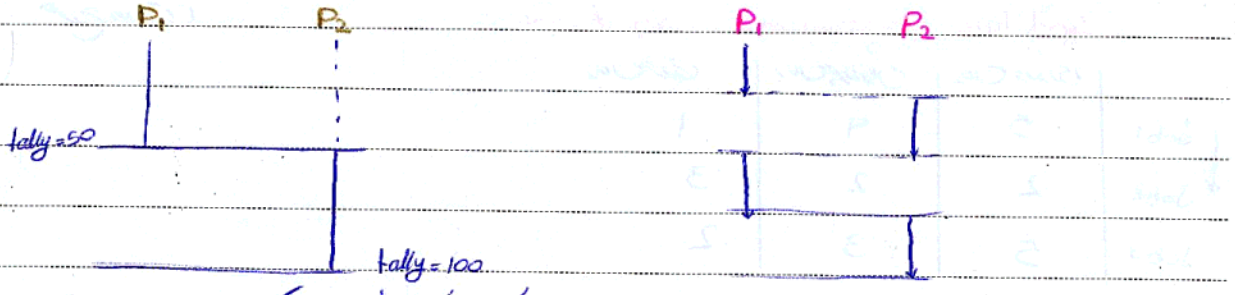
```
const int n=20;
int tally;
void total()
{
    int count;
    for (count=1; count<=n; count++)
    {
        tally++;
    }
}

void main()
{
    tally=0;
    parbegin(total(), total());   ← دستور شروع همزمان اجرا
    write(tally);
}
```

Subject:

Year. Month. Date. ( )

P_1 و P_2 چون به صورت جدا جدا هستند زمان بین آن‌ها تقسیم می‌شود حالا چی می‌دانیم و هر ترکیبی از این‌ها می‌تواند پشت سر هم بیاید و ترکیب‌های مختلفی که می‌تواند وجود داشته باشد کران بالا و پایین هستند

کران بالا کامل مشخص است چون P_1 اجرا می‌شود و P_2 هم اجرا می‌شود ← 100



هر حالت دیگری را بیاییم از این کمتر می‌شود

tally++ = tally+1; ← برای اجرا باید به زبان سخت‌افزار تبدیل شود

Reg1 = tally;
Reg = Reg1+1;
tally = Reg1;

← این را compile کردیم (خودش compile شد)

چیزی که اجرا می‌شود این است

اگر کم زدن را در نظر می‌گیریم برای چیزهایی که اجرا می‌شود تاثیر دارد.

اگر در حال از بین tally تقاطع کم زدنی در وسط کار یعنی بعد از اجرای Reg = Reg1+1 متوقف شد، خودمان می‌گوییم چه اشکالی دارد، دفعه‌ی بعد که نوبتش می‌شود می‌نویسیم ← ذهن خیال باطل چون ذهن را به کس دیگری دادیم مثلاً کس دیگر متوجه بود که ۱۱ را بنویسد و کران ۱۵ را می‌بیند

Reg2 = tally 10

Reg2 = 11

tally = 11

یخ به تراشه‌ی لوب ← tally = 15

tally = 11 ← دوباره نوبت P_1 می‌رسد

متوجه می‌شویم 50 جواب بشود برای کران پایین دیگر جواب بین نیست چون جواب $\frac{2}{2}$ است، حالا چرا؟

کم زدنی دقیقی اجازه‌ی متوقف شدن خیلی وسط‌ها را نداده و باعث این کم‌های شود.

Subject:
Year. Month. Date. ( )

کارهایی که دسته به آن می‌دهند دسته‌ای

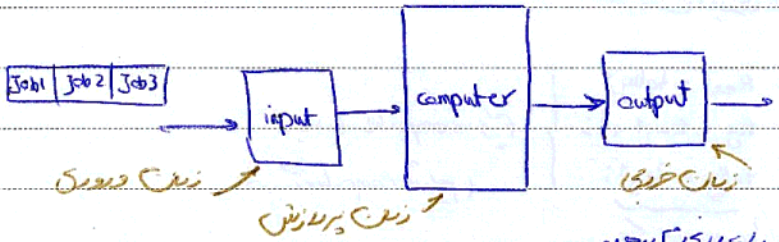
تمرین) راجع به یک مسئله خیلی ریزی (در درس نگفته نشده) جدول زیر زمان‌های لازم برای ورود و خروج سه کار را در یک سیستم batch نشان می‌دهد. حداقل کل زمان مصرفی برای اجرای هر سه کار چقدر است؟ (ترتیب ورود تعیین کننده ترتیب پردازش و خروج است)

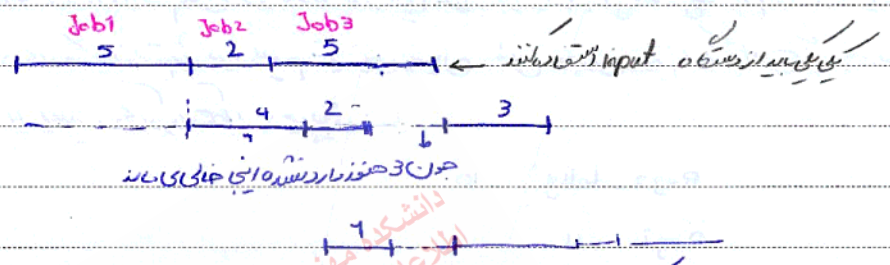
Input time processing time output time

	زمان ورودی	زمان پردازش	زمان خروجی
Job1	5	4	1
Job2	2	2	3
Job3	5	3	2

سیستم batch

این سیستم‌ها عملاً متوالی هستند جدول ورودی برای پردازنده در خروجی را تحویل می‌دهند می‌توانیم به این سیستم یک سری Job برای اجرا بدهیم که کدام از این‌ها زمان‌های لازم دارند که خواسته می‌شوند ← زمان ورودی





پردازش باید وقتی شروع شود که ورودی به طور کامل خوانده شود یعنی پردازش Job1 مجبور 5 واحد زمانی شروع می‌شود و بعد از آن چون Job2 به طور کامل وارد نشده آن را پردازش می‌کنیم و چون بعد از آن Job3 هنوز کامل وارد نشده می‌توانیم آن را شروع کنیم پس سیستم تا صدای بیکاری می‌ماند و وقتی Job3 کاملاً آمد شروع به پردازش آن می‌کند.

Subject:
Year. Month. Date. ( )

مثال) دو تا فرآیند P_1 و P_2 به طور همزمان اجرا می‌شوند

P_1	P_2
while (TRUE) {	while (TRUE);
print "A";	print "C";
print "B";	print "D";
}	}

(حلقه‌ی بی‌نهایت while)

بدون هم‌زمانی ↓

ACDCD

اگر خروجی را برای درست بودن چون هر ترکیبی امکان داشت

(AB)*(CD)*

A(CD)*B

می‌تواند جواب باشد چون هم‌زمانی در پایان هر loop تمام می‌شود

یک A یک print می‌کنیم هم‌زمانی ما تا زمانی که P_2 چندین بار تکرار می‌شود

BCAD ← متممی از خروجی

از یک نخ‌ای شروع می‌کنیم B را print می‌کند هم‌زمانی تمام می‌شود

C را print می‌کند و ... را درست می‌دهد

C*A*B*C* ... این امکان ندارد باشد

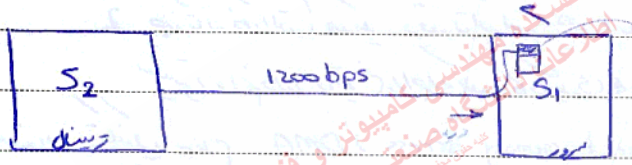
مثال دیگر) در مورد کنترل دسترسی‌ها و خروجی‌ها با دقت

دو تا سیستم با هم ارتباط هستند

سیستم اصلی

با اصول دقت در امکان جا نیفتاده

می‌کنیم



S_2 برای S_1 مثل ترمینال عمل می‌کند و برنامه‌ش هم S_1 اجرا می‌کند

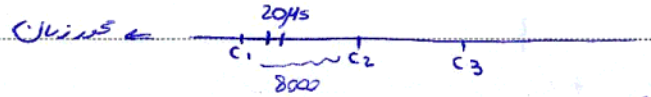
سرعت ارتباط بین این‌ها خیلی کم نوشته ← 1200 bitps ، واحد ارتباطی این دو سیستم کاراکتر است،

هر کاراکتر 8bit است

دریافت هر کاراکتر با یک وقفه معرفی می‌شود با این سرعت 120 کاراکتر در ثانیه می‌توانیم ارسال کنیم

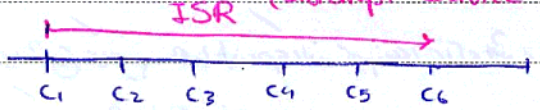
Subject:
Year. Month. Date. ( )

وقفه حاصل می‌شود ← باید داخل روتین وقفه برویم و آن را اجرا کنیم
دوال وقفه خواندن پورت و ذخیره در حافظه ← 20 μs
سوال اول: آیا این سیستم مشکل پیدا می‌کند؟
فاصله دو کاراکتر $8000\,\mu s = 8\,ms = \frac{1}{120}\,s$

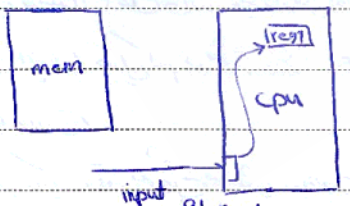


در 8000 می‌خواهیم 20 تا برای این کار مصرف کنیم ← هیچ مشکلی پیش نمی‌آید.

حالت دوم:
در حالت دوم می‌بینیم شبکه را اینجوری می‌کند به طریقه بعد از هر 4μs یک کاراکتر وارد می‌شود
1 char per 4μs



کاراکتر اول یک وقفه تولید می‌کند، برنامه هیچ فرقی نمی‌کند، C1 روتین وقفه را راه‌اندازی می‌کند
برای این که به روتین وقفه برای C1 رسیدگی کند 9 تا char را از دست داده و همچنین تا ISR تمام می‌شود دوباره یک char جدید وارد می‌شود و همینطور ISR در حال اجراست.
راه حل اول: 1- کندش کنیم و اجازه ندهیم هر 4μs بیاید (در صورتی که امکان پذیر باشد)
2- چند تا کاراکتر را مدیریت وقفه تولید کنیم، همزمان به خودشان بیایم و این چنین سیستمی داشته باشیم روتین را اجرا کند و کارهایی را انجام بدهد
راه حل اصلی: استفاده از تکنیکی به نام DMA (Direct memory access) ← دسترسی مستقیم به حافظه
ما دسترسی غیر مستقیم به حافظه داریم
دسترسی port به حافظه غیر مستقیم است
و توسط CPU است.

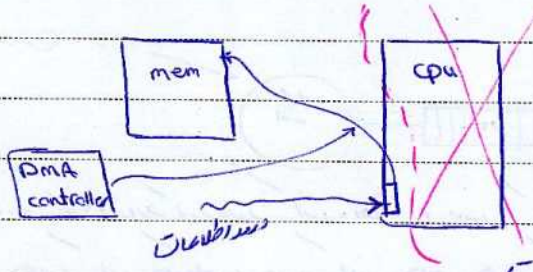


P4PCO

جدول می‌خواهیم ولی همان خانه‌ها را پر نمی‌کنیم

Subject:
Year. Month. Date. ( )

برای هر چیزی که واردی شود باید چیزی Instruction اجرا رحیم
حال DMA ای گوید که وسط را برداد و ارتباط را مستقیم برقرار کن
چون لازم نیست Instruction اجرا هی رحجی امر خروجی ی دود برجاش پس خیلی بریح اجرای شود ولی
کمی از زرفیتی برای اجرای این کاری خواهد

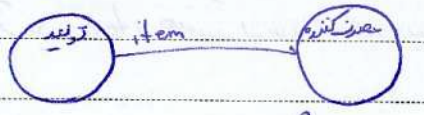


کنترل DMA Controller دروس OS است
هروقت char ،Iooo آمد Interrupt بهش میده این هاهم می‌شوند بلکه در حافظه ذخیره می‌شوند و اگه در این حا
حافظه‌ش buffer می‌گویند)
مثال: یک وقفه برای Iooo ورودی

جلسه سیزدهم ۲۰۰۷/۰۱/۰۷

هماهنگی فرایندها: process synchronizing
چند تا برنامه خروجی دارای منابع مشترک بین باید برنامه‌ای هماهنگ شوند که مزاحم کار یکدیگر نشوند. یعنی استفاده
مشترک از فرایندها به صورت حفاظت شده باشد.
این هماهنگی ← توسط خود فرایندها
← توسط OS

مثالی از هماهنگی فرایندها
مثال تولیدکننده (producer) و مصرف‌کننده (consumer)
دو فرایند که یک چیزی تولید می‌کند که فرایند

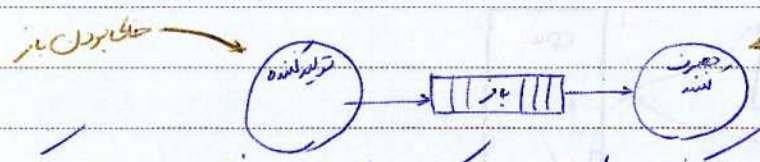


مصرف‌کننده باید با هم هماهنگ باشند
مصرف‌کننده اگر خواهد چیزی را مصرف کند باید قبلش تولید شده باشد

Subject:
Year. Month. Date. ( )

پس تولید باید قبل از مصرف باشد و اگر چیزی تولید نشده باشد نمی‌تواند مصرف کننده، مصرف کند.
اگر تولید کننده بخواهد تولید کند و مصرف کننده بخواهد مصرف کند
این مثال محدودکننده است و
واقعیتش این در به هم خیلی زیاد است

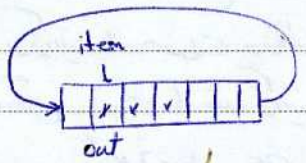
مصرف ← تولید
تولید ← "



در این جا تولیدکننده کاری ندارد که مصرف کننده item را مصرف کرده یا نه، کارش فقط این است که تولید کند چه بلند و مصرف کند چه بلند و هم زمان هر چقدر که بخواهد می‌تواند item ها را مصرف کند
اگر خالی باشد

هماهنگی: برای تولیدکننده: باز پر نباشد
" مصرف کننده: از پشت جای ورود باز

← در این جا هماهنگی زیر ذره بیشتر شده و نسبت به حالت قبل که یک پله به هم نزدیک بودند کمتر شده.



پیاده‌سازی باز

یک آرایه است و pointer داریم به نشان می‌دهد که جای باز را می‌توانیم input کنیم و از طرف دیگر یک pointer برای مصرف کننده است که نشان می‌دهد تا کجا خوانده شده و عقب‌تر از pointer اول است این دو پشت هم هستند و هر چقدر خانه‌های از ابتدا خوانده شده آزاد می‌شود، این باز به صورت باز حلقوی می‌تواند تا اندازه زیاد رشد

هر کدام از خانه‌های باز یک item هستند که ساختاری دارند و هر چیزی می‌تواند باشد

Subject:
Year. Month. Date. ( )

typedef item ____ ;
item buffer[n] ; ← آرایه‌ای از item تعریف شد
int in, out ; ← pointer ها

وقتی بافر خالی است in=out است، زمان خالی بودن: in=out
به نظر شما هم همین است؟ اگر همین باشد وقت پر خالی بودن چه فرقی دارند؟
فرقشان نیم ← برگردیم تا جایی که out بسیم و یک خانه عقب‌تر به out بگردیم پر است و برای این که یک خانه فرق بگذاریم پر بودن: in+1=out
↳ (in+1) mod n = out
حال حالت پر هنوز باید بررسی شود
اگر اندازه بافر یک باشد از دست دادن یک مکان (خانه) خیلی مهم نیست

PRODUCER:
while (True) { ← حلقه‌ای که تا بی‌نهایت اجرا می‌شود
___
produce an item in nextp ← nextp آدرس item را در بافر دریافت می‌کند
__
while ((in+1) mod n = out); ← تا زمانی که پر است این حلقه تکرار شود تا یک مکان خالی شود
buffer[in] = nextp;
in = (in+1) % n; ← یک واحد pointer اضافه می‌شود
}

CONSUMER
while (True) {
while (in == out); ← تا زمانی که خالی است در حلقه بچرخ
nextc = buffer[out];
out = (out+1) % n;
__
consume the item in nextc; ← item در nextc مصرف شود

Subject:
Year. Month. Date. ( )

update کردن این متغیر هم صورت نمی‌گیرد

متغیرهای تبادل توسط OS تعریف کرده که بین فرایندها مشترک باشد در این حالت مشکل پیش نمی‌آید زیرا مشکل پیش می‌آید که دو فرایند با هم بخواهند متغیر را تغییر دهند

← می‌خواهیم راه حلی بدهیم که بتوان از دست‌اندازی‌ها جلوگیری کرد

یک متغیر counter در این مثال داریم که تعداد خانه‌های پر بافر را نشان می‌دهد

اگر counter=0 ← خالی

اگر counter=n ← پر

و قسمت‌های کد زیر به صورت زیر تغییر می‌کند،

producer: while (counter==n);
buffer[in]=nextp;
I counter=counter+1

consumer: while (counter==0)
nextc=buffer[out];
II counter=counter-1;

این راه حل کار نمی‌کند؛ چون counter را هم دو پردازه تغییر می‌دهند و ممکن است در این بخش counter اشتباه شود ← پردازه‌ها بدون رعایت ترتیب اجرا می‌شوند

برای جلوگیری از این مشکل به یک مکانیزم همگام‌سازی نیاز داریم

دستورات I و II به صورت سطح پایین کامپایل شده

counter=counter+1
- reg1=counter
- reg1=reg1+1
- counter=reg1

counter=counter-1
- reg2=counter
- reg2=reg2-1
- counter=reg2

Subject:
Year. Month. Date. ( )

در producer هشتم مقدار counter=6 است

counter = 6

reg1 = 6

reg = 7 ⇒ حال از نخ کمر بای عاج شده

⇒ الان باید counter=7 شود

consumer counter = 6

reg2 = 6

reg2 = 5

counter = 5 ⇒ کشاورز است این عدد غلط است

در حالت اصلی باید به 6 می‌رسیدیم

← حدود ۱۰ اول هر دو نخ را فقط می‌خوانند و چون رشته‌ای نمی‌نویسند مشکلی برای ما ندارند و فقط کل است

پس خالی بودن را زودتر از پر برگشتن می‌گیریم

← در این حالت یک متغیر مشترک int x استفاده کردیم و صدای آن هرجای خواندن باشد پس

استفاده از یک منبع مشترک بدون حفاظت

همه مشکلات اصلی در این حالت بود

نباید همزمان استفاده مشترک از منابع باعث حاصل شود در حدود نیمی از حاصل‌ها نشدنی حفاظت منابع مشترک از بین برود

حفاظت شده به طرفی که زودتر از رشته‌ای حفاظت می‌کند

یک مشکل حل است برای هر منابع پس باید یک راه حل کلی باشد و فقط در حدود این متغیر نباشد

هر منبع مشترکی که استفاده می‌شود باید حتماً یک controller باشد مسئولیت به سراغ منابع مشترک می‌آید درخواست‌های خود را به OS می‌دهیم و OS این را handle می‌کند چون همه چیز را می‌بیند این توانایی را دارد و همه چیز را به دست OS می‌سپاریم برای همین است چون همه‌جا را می‌تواند ببیند

Subject:
Year. Month. Date. ( )

اینکه OS این کارا انجام بده دیگه خودمان مزایا و رقابتی دارد.

اگر بیایم برای OS برنامه کار می‌کنیم ولی سیستم را کند می‌کند و کارهایش را چون کلی این کار بده برای یک کار کوچک کد زیادی را اجرا کند چون باید همه حالت‌ها را مستقر بگیرد.

مسئله را به صورت کلی بیان می‌کنیم:

مسئله ناحیه بحرانی (Critical Section problem)

ناحیه بحرانی: جایی که می‌خواهیم حفاظت شده باشد مثل قبل ناحیه بحرانی حواردیز است

counter = counter + 1 و counter = counter - 1

برای حل این یک مدل برای ناحیه بحرانی ترین می‌کنیم

مدلی برای ناحیه بحرانی:

n تا فرآیند را در نظر می‌گیریم که می‌خواهیم با هم حافظت کار کنند

می‌توانند مختلف باشند ولی به منبع مشترکی نیاز دارند

```
void Pi(int i)
{
    while (True) {
        enter_critical(i);
        /* critical_section i */ ;
        exit_critical(i);
        /* noncritical_section */ ;
    }
}
```

این چه چیزی باید تا بتواند وارد ناحیه بحرانی شود یا نه.

حفاظت بین دو سطر

کد اصلی برنامه چیست، در این بخش ... منبع مشترک دستری پیدا می‌کند

در این جا به منبع مشترک کاری ندارد ...

بیشترین حالت که ما دیدیم در ناحیه بحرانی نیستیم

بهترین حالت است که n تا فرآیند از یک منبع مشترک را به حال استفاده هستند

```
void main()
{
    parbegin(P1(), ..., Pn());
}
```


واضح است که برای n تا فرآیند کار کند وجود دارد روی خیلی زیاد است و همه‌ی را در نظر می‌گیرد و صدق نه خیلی وقت‌ها لازم نیست برای انجام این راه حل هزینه‌اش می‌ارزد.

Subject:
Year. Month. Date. ( )

چون این مشکل راه حل ساده‌ای ندارد برای دو فرایند ابتدا حل می‌کنیم و سپس برای n تعمیم می‌دهیم

جلسه چهاردهم

http://groups.google.com/group/OS_aut_fall_87

برای این هفته (1-2-3-12-13)

- راه حل های نرم افزاری را برای دو فرایند بررسی می‌کنیم
- راه حل های نرم افزاری برای مسئله ناحیه بحرانی
- برای درست بودن راه حل باید سه شرط برقرار باشد

شرط های راه حل صحیح:

1. اگر فرایندی در ناحیه بحرانی باشد دیگر فرایندی نمی‌تواند وارد این ناحیه شود ← انحصار متقابل mutual exclusion
متقابل فرایند ناحیه بحرانی باشد
2. انتظار محدود ← bounded wait ← برای وارد شدن به ناحیه بحرانی باید تضمین شود که میزان انتظار فرایند محدود است
← این دو شرط برای صحیح بودن راه حل کافی است اما یک شرط دیگر هم بعدا می‌گوییم

تلاش اول (first attempt)

P_0 →
```
while (True) {
    while (turn != 0) ;
    critical_section0();
    turn = 1;
    noncritical_section0();
}
```

P_1 →
```
while (True) {
    while (turn != 1) ;
    critical_section1();
    turn = 0;
    noncritical_section1();
}
```

boolean متغیر turn

فرض کنید P_0 به خط while برسد و متوقف شود.
اگر turn = 0 شود نوبت برای P_0 است و می‌تواند وارد ناحیه بحرانی شود و turn را 1 کند
ولی وقتی شرط برقرار بود وارد ناحیه بحرانی خودش می‌شود وقتی کارش در ناحیه بحرانی تمام شد نوبت را به P_1 می‌دهد

Subject:
Year. Month. Date. ( )

در این روش اعضا متقابل وجود دارد و چون turn با عزاست با ! در یک لحظه فقط یک حقدار را دارد. نوبت بین P_0 و P_1 در نوبت است و آمد است تعداد نوبت ها مساوی است چون هر کی نوبت را بهش برادن یکی می دهد ← هم توی چیز تایپری درست یا غلط بودن این که پیش نمی آید.

← شرط دوم برقرار است چون هر بی حداکثر یک نوبت منتظر می شود چون نوبت بعد از این که خارج شد به تایپری می دهد این که این یک نوبت چقدر طول می کشد ربطی به موضوع ندارد.

← با از این راه حل خوشمان می آید چون شرط سوم که نگفتیم نباید یعنی ممکن راه حل خوبی نیست حالا چرا؟ وغیر بحرانی

فرض کنیم ناحیه بحرانی P_0 کم و برای P_1 زیاد باشد.

نوبت با P_0 ← نوبت با P_0 وارد ناحیه بحرانی می شود و کارش را تمام
P_1 در ناحیه غیر بحرانی
ناحیه بحرانی P_0 ← راغا می کند نوبت را به P_1 می دهد و کار ناحیه غیر بحرانی خود را تمام می کند
نوبت با P_1
ناحیه غیر بحرانی P_0
انتظار

P_0 باید انتظار بکشد که کار ناحیه غیر بحرانی P_1 تمام شود و خیلی طول می کشد ولی الان P_1 کاری با ناحیه بحرانی ندارد پس باید P_0 بتواند کار خود را با ناحیه بحرانی اش بدهد و نکته نوبت ها را مساوی می دهد اصلا خوب نیست (چون P_1 اصلا ناحیه بحرانی را نمی خواهد ولی ما صبر می کنیم تا بخواهد) اگر یکی نتواند نوبت بخواهد دیگری پاسش پوچ - پس تر را به برعکس فرایند با برعکس گفتن می رساند و به هر دری را به برعکس پایین می رساند

شرط سوم: پیشرفت ← کسی که وارد طلب ورود به ناحیه بحرانی نیست در نوبت ترکیب نکند

این راه حل خوبی نیست چون شرط سوم را ندارد می خواهیم راه حل بدهیم ← متغیری تعریف می کنیم که نشان دهیم کی وارد طلب ورود به ناحیه بحرانی است.

Subject:
Year. Month. Date. ( )

(second attemp)

P_0 →
```
while (True) {
    flag[0] = True;
    while (flag[1]);
    critical_section0();
    flag[0] = false;
    noncritical_section0();
}
```

P_1 →
```
while (True)
    flag[1] = True;
    while (flag[0]);
    critical_section1();
    flag[1] = false;
    noncritical_section1();
}
```

پایان همزمانی
wait
wait

flag ← نشان می‌دهد که او طلب است اگر کسی خواهد وارد ناحیه بحرانی شود flag خود را true می‌کند
و برای وارد شدن به ناحیه بحرانی flag دیگری را چک می‌کند

این راه حل از نظر انحصار متقابل صحیح است و هر دوی آن‌ها وارد ناحیه بحرانی نمی‌شوند.
شرط سوم را هم دارد ← کسی که داوطلب نیست در وقت تخصیص چون flag آن false است
آیا شرط دوم هم برقرار است؟
اصلا به شرط دوم نمی‌رسد و به مشغول دیگری کار دارد ← ① ضعف تسهیم همزمانی برای P_0 است، flag خود را True
می‌کند تا وارد شدن به ناحیه بحرانی همزمانی آغاز می‌شود و بررسی داوطلبی می‌شویم. نوبت ② P_0 می‌رسد و flag
خود را True می‌کند و وارد حلقه شده که ③ flag[1] است که گیر می‌کند و می‌ماند و دیگر نمی‌تواند وارد شود و همین باعث می‌شود تا همزمانی آغاز
شود نوبت ④ P_0 می‌رسد و چون flag[1] هم true است P_0 در حلقه گیر می‌کند و هر دو منتظر هستند flag
دیگری false شود ← شبیه حالت بن بست است و راهی برای بازنشستن و احتمال غیر صفر برای اتفاق افتادن
این وجود دارد.

راه حل درست: این دو راه حل را با هم غلط می‌کنیم ← هم turn داریم هم flag

Subject:
Year. Month. Date. ( )

تلاش سوم (third attemp)

```
P0 → while (True) {
        flag [0] = True;        ← ورود
        turn = 1;               ← برای نوبت دیگری
        while (flag[1] && turn == 1)
        critical_section0();
        flag[0] = false;        ← خروج
        noncritical_section0();
     }
```

```
P1 → while (True)
        flag [1] = true;        ← ورود
        turn = 0;
        while (!flag[0] && turn == 0);
        critical_section1();
        flag [1] = false;       ← خروج
        noncritical_section1();
     }
```

flag خودرا True میکند ← می‌خواهم وارد ناحیه بحرانی شوم
اگر کسی Turn را دارد اما flag ندارد ← می‌تواند جلوی کسی را برای ورود به ناحیه بحرانی بگیرد.
احتمال بن بست وجود دارد.

← اگر هر کس turn را قبل از ورود به ناحیه بحرانی به خودش بدهد (الگوریتم خوبیانه) ممکن است طرف مقابل زمانی بیند از که نوبت به دیگری برسد. یعنی مثلاً P_0 چندین بار وارد شود و به P_1 نرسد و این چندین بار بستگی نسبت چقدر است ← احتمال اتفاق افتادن این غیرصفر است

اما اینکه نوبت را به دیگری بدهیم اگر داوطلب نباشد هیچ اتفاقی نخواهد افتاد و یک نوبت بیشتر حذف می‌شود و نه بیشتر چون کسی که الان turn را دارد اول turn را به من می‌دهد.

تحلیل زمانی پیچیده‌ای نیاز است و تحلیل تأثیری که دارد این است که حداکثر یک نوبت می‌تواند من را عقب بیندازد

تعمیم به n فرایند:

همین راه‌حل را برای n فرایند تعمیم دهید. البته کار خیلی کثیفی است. اینکه این را بدانیم خوب است ولی اگر خودمان بخواهیم پیاده‌سازی کنیم سخت است

در حالت دو فرایند می‌خواهند وارد ناحیه بحرانی شوند
می‌خواهند ...

Subject:
Year. Month. Date. ( )

idle — اینجا در این حالت رقیبی نداریم ← مجاز است وارد شود
want_in — تمایل دارد وارد ناحیه بحرانی شود
in_cs — وارد ناحیه بحرانی شده

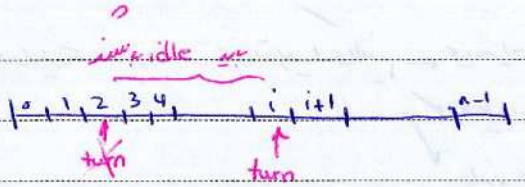
```
enum state {idle, want_in, in_cs}
state flag[n];
int turn;
enter_critical_section:
do { flag[i] = want_in;
     j = turn;
     while (j != i)
         if (flag[j] != idle) j = turn;
         else j = (j+1) % n;
     }
     flag[i] = in_cs;
     j = 0;
     while (j<n && (j==i || flag[j] != in_cs)) j++;
     if while (j>=n && (turn==i || flag[turn]==idle)); break
     } while(true);
     turn = i;
```

حلقه درونی باید چک کند تا به in_cs نرسیده است
سپس دوباره از اول آرایه شروع شود وارد ناحیه بحرانی شود.

وقتی چند نخ وارد ناحیه بحرانی شوند ← turn را خودش در همین حلقه از این وضعیت خارج کرده بود
آخرین نخی است که اجازه ورود به ناحیه بحرانی را دارد

Subject:
Year. Month. Date. ( )

ناصر جم چی کار میکنند؟



چی کار میکنند جدیدترین را از turn تا خودش که همه باید idle باشند اگر اینطور باشد می گوییم حق با من است اگر idle نباشند در این صورت میگردد و اندازه ای صبر میکند تا همه را idle ببیند یعنی مطمئن حق خودش است اگر این حالت اتفاق افتاد flag خودش را به in_cs تغییر میدهد

اما هنوز این راه حل، راه حل کاملی نیست در این مشکلات را در ناصیه خروج حل می کنیم

```
exit_critical_section:
    j = (turn+1) % n;
    while (flag[j] = idle) j = (j+1) % n;
    turn = j;
    flag[i] = idle
```

همراه چی گفتیم بیشتر کی هست که idle نیست اولین کسی که idle نیست کی است turn را به اول کسی که هست اگر در این n تا فرایند کسی وجود نداشت که idle نیست turn خودم برگردم چون با نوبت کار می کند حداکثر n-1 نوبت طول می کشد تا نوبت به من برسد

تا چی هایی که آخر می کند ← برای ایجاد انحصار متقابل
شرط انتظار محدود وجود دارد ← حداکثر n-1 نوبت
واضح است شرط سوم هم هست چون اجازه idle را دقیقا نمی دهد

Subject:
Year. Month. Date. ( )

جلسه پانزدهم 8,19

ارسال تمرین‌ها به Pedram@aut.ac.ir

راه‌حل جلسه قبل برای n تا در OS نداریم

الگوریتم نانوایی baker's algorithm که در استاندارد posix هم استفاده می‌شود و متعلق لمپورت است

هرکس که می‌خواهد وارد ناحیه بحرانی شود شماره می‌گیرد (نوبت‌دهی) و بررسی می‌کند که آیا کسی شماره‌ای دارد که زودتر باید وارد ناحیه بحرانی شود یا نه

```
boolean choosing[n];
int number[n];
```

متغیرهای سراسری و مقداردهی اولیه صفر

```
enter_critical_section:
    choosing[i] = True;
    number[i] = max(number[0], ..., number[n-1]) + 1;
    choosing = False;
    For (j=0; j<n; j++)
        while (choosing[j]);
        while (number[j] != 0 && (number[j], j) < (number[i], i));
}
```

شماره‌ای که به هر فرایند می‌دهیم

یکی بیشتر از max شماره‌هایی که تا الان دیده‌ایم

علت اینکه حلقه‌ی for از 0 شروع می‌شود

منتظر می‌ماند تا فرایند j شماره‌اش را انتخاب کند

منتظر می‌ماند تا فرایند j شماره‌اش را انتخاب کند

برای سادگی شرط را دو بخشی کرده‌ایم

اگر number[j] صفر باشد یعنی j قصد ورود به ناحیه بحرانی را ندارد

```
exit_critical_section:
    number[i] = 0;
```

Subject:
Year. Month. Date. ( )

مرحله تخاره گیری، مرحله طولانی است، بدین ترتیب max بلندترین زمان ماشین آن طول‌های سوزن‌بین در وسط کاری‌تان قطع شود، بنابراین حاضر همراه باید به حساب آورد حتی کلی که در حال تخاره گیری است چون عمل دست قبل از من باشد و وسط کار قطع شده

همچنین امکان تخاره دهی خسارتی وجود دارد، این کار را کند اگر دستگاه تخاره خسارتی بود به تخاره زنید برای شکستن حالت استادی نکنید، اول تخاره‌ها چک می‌شود اگر خسارتی بود تخاره نزنید چک می‌شود

مکانیسم الگوریتم در حال شده: number حتا زمانی بالای دوده overflow بعد وقتی overflow نادیده کار کنیم؟ حل آن بسیاری نیست، دچار اشتباه می‌شویم

این راه حل تحت نرم‌افزار کاری کند، اگر زمان‌های وجود داشته باشد به چرا این حادثه در ناحیه بحرانی نیست در این صورت number حاضر می‌شود یعنی زمانی کاری کند که این‌ها دائم در رقابت باشند و از این زمان‌ها این وسط باشد در عمل هم این طوری یعنی دائم در حال رقابت نیستند

راه حل سیار ساده

تا حال در مورد دستورالعمل‌های ماشین صحبت کردیم (سیار ساده‌اند) الان الان صحبت می‌کردیم که حتماری دستورالعمل پیچیده دارد (دستورالعمل‌های غیرساده)

دستورالعمل‌های ماشین

tsl rx Test_and_Set

test_and_set_logical

بیش از یک عمل را می‌دهد

مقدار اولیه

فرض بر این است

① مختبات حافظه reg

② عدد غیرصفر بخواند ! tsl می‌کند

اگر دستم بلندیم cpu این دستورالعمل را دارد راه حل سیار ساده می‌شود.

Subject:

Year. Month. Date. ( )

زبان assembly :

```
enter_critical_Section
    tsl   reg2 , flag        ← عملیات اتمی
    cmp   reg2 , #0
    jnz   enter_critical_Section
    ret
exit_critical_Section
    move  flag , #0
    ret
```

tsl reg2, flag ← عملیات اتمی

cmp reg2, #0 ← یعنی حساب کن ببین صفر شد یا نه، حساب کن ببین Zero=1 میشود یا نه

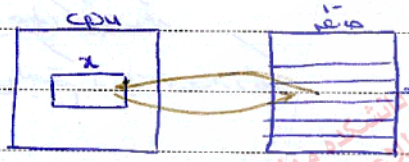
jnz enter_critical_Section ← اگر Zero=0 بود برو به اول

هر زمانی که وارد ناحیه بحرانی می‌شود flag را یک می‌کند یعنی flag=1 میشود و از ناحیه بحرانی استفاده می‌کند سپس در خروجی چک می‌کند ببیند کسی داخل وارد ناحیه بحرانی شده یا نه اگر نشده flag صفر شده می‌تواند وارد شود در غیر این صورت دوباره برمی‌گردد و تست می‌کند.

در tsl ← تعریف زبان به وسیله ارجاع می‌شود چون یک دستور اتمی واحد است.

نوعی پردازنده CPU درست بعد ← Intel این را اندازه چیزی شبیه این دارد

swap →



محتویات این حافظه جابه‌جا می‌کند

```
enter :
    mov   regx , 1
    swap  regx , flag
    cmp   regx , #0
    jnz   enter
    ret
```

```
exit :
    mov   flag , #0
    ret
```

برای هر صنف می‌توانیم ناحیه بحرانی بسازیم که تعریف کنیم چند flag بسازد

Subject:
Year. Month. Date. ( )

این راه حل انحصار متقابل را کاملا حفظ می کند. (تضمین شده و می تواند خراب کند)

مثال: این راه حل کجاست؟
شرط محدودیت زمانی را تضمین نمی کند. (اگر شرط سوم را دارد (کسی که داخل نیست، دخالت داده نمی شود) محدودیت زمانی تضمین نمی شود.
نسبت به حالت کاملا تصادفی گرفته می شود یعنی اگر 5 تا فرایند داریم یکی گرفته، 4 تای خواهند بین این 4 تا تصادفی است. احتمال غیر صفر دارد که یکی اصلا نوبت بهش نرسد

← این راه حل ضمن سادگی صحیح نیست
ما دنبال راه حل صحیح بودیم. دنبال چیزی بودیم که تحت شرایط خاص کار کند

این راه حل تحت چه شرایطی کار می کند؟ اگر بطور دقیق واقعی راشته باشیم که هیچ کس داخل نباشد کار می کند ولی اگر همیشه فرایند ها در حالت باشند این راه حل کار نمی کند و در واقعیت این دست که زمان های وجود دارد که هیچ کس داخل نباشد

← اگر در ناحیه بحرانی این شرایط بین بیت است و برای تمام این راه حل ها محتمل است اتفاق بیفتد
ما فقط ناحیه بحرانی بدیم خیلی کوچک باشد
این راه حل بد است و ما به دنبال راه حلی نیستیم که برای همه جوا درد کار کند

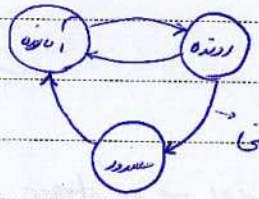
این راه حل زمانی درست کار می کند که ← دائم در حال رقابت نباشند
↳ ناحیه بحرانی کوتاه باشد

← نقطه مشترک تمام راه حل هایی ارائه شده:
از انتظار مشغولی (busy waiting) استفاده می کند
ما برای انتظار از حلقه استفاده می کنند ← زمان پردازنده می گیرد (هیچ کاری نمی شود) هیچ کاری نمی شود
نمی شود که راه حل هایی ارائه شده که فرایند انتظار را حذف می کند

Subject:
Year. Month. Date. ( )

می خواهیم راه حلی ارائه کنیم که انتظار مشغول بیدار نشسته باشد و از حالت مسدود استفاده می کنیم.

آنهایی که می خواهند منتظر ناحیه بحرانی باشند به حالت مسدود می روند



انتظار برای ناحیه بحرانی را مثل درخواست برای سرویس در نظر بگیریم

شعوری پیاده می شود که تحت عنوان یک ابزار پیاده سازی شده

تعاریف (Semaphor یا سمافور) یک متغیر int به همراه صف که اندیسی از آن می خوانیم چه موقع باید بیاید ناحیه بحرانی شده و یک سری تحریه استرات مرد

ابزاری برای تعامل فرایندها در ورود به ناحیه بحرانی

ساختار داده ای است ← یک متغیر int
↙ یک صف

در حالت مسدود فرایندها بنا بر سیاستی که می خواستند در یک صف منتظر می شوند این صف Semaphor را هم در این جا می گذاریم ← متغیری که به بقیه سرویس می دهیم

Semaphor ← یک مجوز برای وارد شدن به ناحیه بحرانی

← برای هماهنگی به هم کار داریم

Subject:
Year. Month. Date. ( )

جلسه شانزدهم 19/8/

busy waiting استفاده می‌کنند تا از راه‌حل‌های ارائه شده از

Semaphore (یک نوع راه‌حل)

ابزاری برای هماهنگی

ساختاری شامل یک integer و یک صف دارد. (از خروجی‌ها)

عملیاتی که روی این ساختار داده هست را هم باید بگوییم

عملیات روی Semaphore

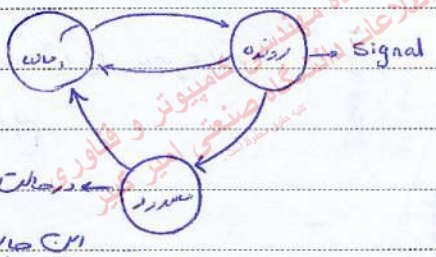
مقداردهی اولیه به مقدار integer باید set کنیم و یک روش برای شروع به کار هم بدهیم

عملیات هم روی Semaphore ← wait

signal ↙

wait ← یک واحد از integer کم می‌کند و نگاه می‌کند که آیا مقدار منفی شد یا نه. اگر منفی شد فرایند حبس و در صف قرار می‌گیرد در غیر اینصورت از منطق یک واحد کم می‌کند و عبور می‌کند

Signal ← یک واحد اضافه می‌کند و چک می‌کند که آیا مثبت یا صفر شد یکی از فرایندهای حبس شده آزاد می‌شود (که در صف است) آزاد می‌کند. (کدام یک اولویتش است)



در حالت مسدود یکی از فرایندهایی که signal

این جا بود به حالت آماده می‌رود برای

هر کسی که signal را اجرا کرده اتفاقی می‌افتد

Subject:
Year. Month. Date. ( )

```
struct Semaphore {
    int count;
    queueType queue;
}

void wait (Semaphore s)
{
    s.count--;
    if (s.count < 0) {
        place this process in s.queue;
        block this process;
    }
}

void signal (semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        remove a process P from s.queue;
        place Process P on ready list.
    }
}
```

یک صف انتظار برای

یعنی منتظر blocked می‌ماند

Binary Semaphore

نوع دیگری از semaphore ← binary sema ← integer که به جای

استفاده می‌شود bool

کاربرد است

اصطلاحاً به این دو عمل signal و wait می‌گویند که اتمیک اجرا می‌شوند (Test & Set مثل

به این معنی که وقتی شروع به کار کرد تا wait اجرا شود و هیچ کار دیگری انجام نشود.

پیاده‌سازی همین چیزی راحت نیست چون باید حلقه‌های بی‌نهایت نداشته باشیم

Subject:
Year. Month. Date. ( )

این استفاده از semaphore در حیطه‌ی ناصیه است.

حل مسئله ناصیه‌ی برای (برای n فرایند) ← نیاز busy waiting ← اجتناب

```
/* program mutualexclusion */
const int n = /* number of processes */
Semaphore S = 1;
void P(int i)
{
    while (True) {
        wait(s);
        critical_section(i);
        signal(s);
        non_critical_section(i);
    }
}

void main()
{
    parbegin (P(1), P(2), ..., P(n));
}
```

مقدار اولیه 1 یعنی یکی از فرایندها می‌تواند وارد ناصیه‌ی شود

هر فرایند برای ورود به ناصیه‌ی باید یک واحد از s کم کند

اگر صفر شد یعنی فرایند دیگری در ناصیه‌ی است و باید صبر کند تا آزاد شود

هر فرایند که از ناصیه‌ی خارج می‌شود یک واحد به s اضافه می‌کند تا آزاد شود

این راه حل ساده هم کاری کند هم busy waiting نیست

قدرت‌های عددی (S.count) نشان می‌دهد چند فرایند در صف داریم.

1. انتظار محدود داریم چون در صف می‌توانیم جای نوبت دهی بگیریم
2. چون مقدار اولیه 1 بود فقط یک فرایند می‌تواند وارد شود بقیه صف می‌کشند
3. کسی که زودتر طلب نشست زودتر می‌گیرد پس انصاف برقرار می‌شود

Subject:
Year. Month. Date. ( )

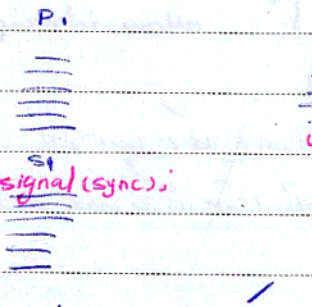
پیاده‌سازی این به شرط اجرا بودن wait و signal خیلی سخت است و بدون این شرط همین راه‌حل‌ها نمی‌توانند مشکلات زمانی را حل کنند. wait را اجرا کرد و کنترل را به سیستم‌عامل داد و سیستم‌عامل هم دارد ناخواسته اجرا می‌شود.

با داشتن این ابزار می‌توانیم کاربردهای دیگری نیز داشته باشیم

← دو برنامه P_1 و P_2 داریم که دستورالعمل‌هایی به صورت S_1 و S_2 اجرا می‌شوند.
سیستم طوری است که S_1 زودتر از S_2 اجرا شود. یعنی چیزی را تولید می‌کند که S_2 به آن نیاز دارد (استفاده می‌کند)
S_1 زودتر از S_2 اجرا شود. حل

معمولاً تابع تعیین نیست ← بهترین یک semaphore این کار را انجام می‌دهد

Semaphore Sync = 0 ⇒



قبل S_2، wait می‌کنیم و S_1 را اجرا می‌دهیم حال S_2 block می‌شود و حتی اگر اجرا شد، S_2 را آزاد می‌کند. از آنجا که S_1 زودتر باید اجرا شود، sync = 1 می‌شود حال wait می‌کنیم 0 می‌شود و منفی نیست S_2 هم اجرا می‌شود.

پیاده‌سازی؟

دو راه‌حل وجود دارد:

راه‌حل اول: چیزی که این‌جا را تضمین می‌کند وقفه‌ها هستند که از کار بیندازیم تا قطع نشود

البته به این برای از کار انداختن وقفه‌ها نیاز به System mode داریم

```
void wait (semaphore S)
{
    inhibit interrupt;        → از کار انداختن interrupt
    s.count;
    if (s.count < 0) {
        place this ...
        block this ... , allow interrupts   → به کار انداختن interrupt
    }
    else allow interrupts
}
```

Subject:
Year. Month. Date. ( )

← کی داره این کارها را انجام میده؟ یک فرایند خیلی در user mode است و عملا این کارها را OS انجام می‌دهد و وقتی مسدود می‌شود خودش را به مسدودی می‌زند

```
void Signal (semaphore s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        remove a ... ;
        place ... ;
    }
    allow interrupts;
}
```

← راه حل خوبی نیست چون از کار انداختن interrupt خطرناک است چون ممکن است interrupt های مهمی بیایند که نتوانند پاسخ دهند (مثلا اینکه یک متغیری از max خودکار شت مثلا در حال حرکت است) و کل سیستم از استفاده بیفتد و بسته شود و کاربر دارد ... میگه کی کلیم کردین با تفنگ ← نفهم ← زمان مارو این خیلی از دست ما میره

← باید حل مناسب تری بیاریم ؟ همان حلقه جادوی

```
void wait (semaphore s)
{
    while (!testset (s.flag));   ← مکانیزمی که flag هم دارد و توسط OS بررسی میشود
    s.count--;
    if (s.count < 0) {
        place ... ;
        block ... ;
        set s.flag to zero;
    }
    else set s.flag to zero;
}
```


P4PCO

Subject:
Year. Month. Date. ( )

```
void Signal(s, flag)
{
while (!testset(s.flag));
s.count++;
if (s.count<=0) {
  remove ...
  place ...
}
  s.flag=0;
}
```

← signal(s), wait(s) را به صورت برنامه‌ی ترجمه‌ی کنیم که اینم اول test&set را صدا بزند و flag را برابر ۱ کند

```
test & set (flag)
wait (s)
set flag to 0
```

```
test & set (flag)
signal (s)
set flag to 0
```

test&set ← حفظ شرکه flag را برابر ۱ کردیم

این جا به interrupt نداریم پس نیازی به این نداریم که flag را برابر ۱ کنیم
این جا هم می‌گوییم چون wait و signal می‌کنیم چون می‌خواهیم flag را برابر ۱ کنیم پس
می‌توانی wait و signal را صدا کنی. صدا کردن را زیاد نباید هم برابر ۱ کنی چون می‌توانند flag را برابر ۱ کنند
می‌توانند wait و signal را

ظاهر قضیه این است که داریم نقض غرض می‌کنیم روش‌های قبلی برای ناحیه‌ی بحرانی کوچک مناسب بودند و
برنامه‌ی بحرانی می‌نویسیم دقیقاً چقدر طول می‌کشد ولی در این جا دقیقاً نمی‌دانیم چقدر طول می‌کشد

Subject:
Year. Month. Date. ( )

دیدیم test & set برای حل بسیاری از مسائل کمک کننده است و هیچ کدام از آن‌ها غلط نمی‌شدند ولی بعضی از مسائل چون انجام عمل wait و signal کوتاه است.

این عملیات جزئی اصولا wait و signal کنترل می‌کنند

این عملیات بر روی حساب دستکاری می‌کنند

```
test & set (flag)
wait (s)          →   ورودی را مسدود می‌کند
flag = 0
ناحیه بحرانی
test & set (flag)
signal            →   از حالت مسدود خارج می‌کند
flag = 1
```

اجرای wait و signal توسط OS را می‌خواهد

که مسدود و در صف قرار دادن

حتما نباید System call باشد می‌تواند library باشد که از چند System call استفاده می‌کند

Subject:
Year. Month. Date. ( )

جلسه چهارم 97.8.21

حل مسئله حاصلضرب ماتریس‌ها با چند پردازه

1- تولیدکننده-مصرف‌کننده ← حاصلضرب، کدام قسمت بیشتر وقت را می‌گیرد؟

```
/* Program boundedbuffer */
const int sizeofbuffer = /* buffer size */
Semaphor s=1;
Semaphor n=0;
Semaphor e=n ; sizeofbuffer
```

تولیدکننده:

```
void producer()
{
while (True) {
→ produce();
← wait(e);
   wait(s);
→  append(s);
   signal(s);
   signal(n);
}
```

انتظار برای بافر

دسترسی به بافر کنترل می‌شود

آیتم را به بافر append یعنی در بافر بگذار

```
void consumer()
{
while (True) {
← wait(n);
   wait(s);
→  take;
   signal(s);
```

انتظار برای بافر

اگر خانه‌ای پر شده‌ایم از آن خانه چیزی برداریم

معادل را از بافر بردار

Subject:
Year. Month. Date. ( )

تعداد منابع های خالی را یکی زیاد می کنند

signal (e);

consume ( );

void main

یعنی Semaphore s برای سنگری اعضای که به ظرفیت رسیده مقدار اولیه = 1

اگر می خواستیم دو نفر بتوانند به جایگاه برسند باید پر نباشد

" " از " بخواهیم سه " " " خالی "

چون e برای تعداد خالی بودن است باید به اندازه سایز بافر باشد چون اگر اضافه کنیم یکی از آن کم می شود

می توانیم اضافه کنیم تا جایگاه صفر نشود و بعد نفی می شود و دیگر می توانیم از wait عبور کنیم

n برای تعداد خانه های پر بودن است و تعداد منابع های پر شده را نشان می دهد

e تعداد منابع های خالی را نشان می دهد.

← بین همه این کدها خانه ها فقط s برای ناحیه بحرانی به ظرفیت یکی برای تعداد خالی ها پر است و این مثال

مثال دیگر که خانه برای ناحیه بحرانی فقط نیست

2 خواننده ها و نویسنده ها : Readers & writers prob.

اینجا یک object مشترک مثل file وجود دارد و نویسنده های مختلف می توانند از این object بخوانند یا بنویسند

بدون هماهنگی می توانند این کار را انجام دهند مثلاً دو نفر می خواهند یکی می توانند در آن بنویسد

برای read کردن همزمان می توانند این کار را انجام دهند چون اطلاعات

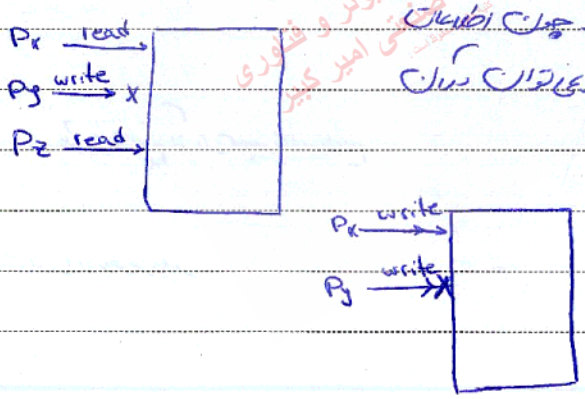
عوض نمی شود و اگر یکی دارد read می کند بقیه می توانند read

write کنند

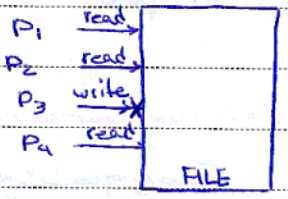
ولی وقتی write می کنیم هیچ کس دیگری نمی توانیم

این کار را بکنیم چون اطلاعات قاطی می شود

عملیات write انحصاری است



Subject:
Year. Month. Date. ( )



اینبار P4 هم باید منتظر بماند پس مسئله دارد ؛

1. سیستم عامل به صورتی است که هر دو عمل به ترتیب انجام شوند

2. شاید اینجا از ابتدا بخواهند بخوانند write به چه کار ماند

First readers & writes prob ← P4 اجازه خواندن بدهیم

Second " " " " ← P4 اجازه بدهیم قبل از write بخواند

اینجا (بدون write) هر دو حالت مشکل است.

First readers & writers prob :

چند فرآیند هم read هم write می‌کنند

فرآیندی که می‌نویسد : Pw (پشت اینجا یکی است)

wrt ← برای پشتی اینجا به کار می‌رود پس حتماً در اولین آن یک است

```
Pw
wait(wrt);
writing();
signal(wrt);
```

Pr ← پشت readcount برای حفظ

```
wait(mutex);
readcount++;
if(readcount==1)
    wait(wrt);
signal(mutex);
reading();
wait(mutex)
readcount--;
if readcount==0
    signal(wrt);
signal(mutex);
```

از این به بعد دیگر نیازی به خواندن را نمی‌کنیم

فرآیند خواننده : Pr

اگر خواننده اول باشد با نویسنده متفاوت است

اول بودن ← به هیچ کدام کاری ندارد

دوم بودن ← لازم نیست کاری

readcount ← مقدار خواننده‌ها پس هر کدام خواهد

بخواند باید این حساب را بداند

به mutex دسترسی ← readcount را اضافه می‌کنیم

پس مقدار اولیه‌اش یک است

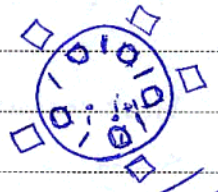
بعد از خواندن باید بررسی کند آخرین خواننده هست یا نه

اگر هست باید منتظر write است آزاد کند

Subject:
Year. Month. Date. ( )

مثال آخرین: Second readers & writers prob. را بخوانید

3- فلاسفه در حال غذا خوردن: Dining Philosophers



این فیلسوف ها چی هستن
دورتا می خوان غذا بخورن باید دو تا چوب و از طرف راست بردارن
و نمیزارن بهم برسه

فرضیات: فیلسوف ها (همیشه) در حال غذا خوردن نیستند (یعنی وقتا فکر می کنند)

منابع: چوب ها

Function: eating (در حال غذا خوردن) و thinking

برای هر chopstick یک Semaphore

```
Semaphore chopstick[5];
void P(i)
{
  while True {
  wait(chopstick[i]);
  wait(chopstick[(i+1)%5]);
  eating();
  signal(chopstick[i]);
  signal(chopstick[(i+1)%5]);
  thinking();
}

void main()
{
  parbegin(P(0), P(1), ..., P(4));
}
```

وقتی اکتیوی است که دو تا چوب رو برداشته باشه

بالا چون wait داریم اگه یکی نباشه میره تو صف

اما این راه حل یک مشکل دارد.

Semaphore ابزاری قوی برای حل اکثر مشکلات است ولی مشکلات دارد

Subject:
Year. Month. Date. ( )

مشکلات Semaphore

1. کلاس Semaphore در اختیار برنامه‌نویس است و این باعث مشکلاتی می‌شود.

2. بکارگیری صحیح wait و signal برای هر سمافور به دقت زیاد نیاز دارد.

wait(s); signal(s); ⇒ صحیح

wait(s); = signal(s); — این جفت اشتباه است

signal(s); = wait(s); — این جفت اشتباه است

wait(s); = wait(s); — این جفت اشتباه است

3. بن‌بست: حالتی است که دو یا چند فرایند منتظر رخدادی هستند که فقط توسط یکی از همین فرایندها می‌تواند رخ دهد.

P_1	P_2
① wait(P);	② wait(Q);
④ wait(Q);	③ wait(P);
signal(P);	signal(Q);
signal(Q);	signal(P);

اگر P_1 پس از اجرای ① پردازنده را از دست بدهد و P_2 اجرا شود، P_2 در ③ مسدود می‌شود و بن‌بست رخ می‌دهد.

4. گرسنگی: حالتی است که یک فرایند برای مدت نامحدود منتظر رخدادی می‌ماند که ممکن است هیچ‌گاه رخ ندهد.

5. سربار: زمان پردازنده صرف مدیریت همگام‌سازی می‌شود.

این مشکلات در مسئله Dining Philosopher وجود دارند و هیچ‌یک از راه‌حل‌های ارائه‌شده برای این مسئله بدون مشکل نیست.

راه‌حل این مسئله استفاده از کلاس دیگری به نام Monitor است.

Monitor مانیتور یا ناظر

Subject:
Year. Month. Date. ( )

تمرین سری 9, 5, 6, 7, 14, 15

حل تمرین 8, 26

یکی از ابزارهای سطح بالا جهت ایجاد همگامی بین فرایندها می باشد البته می توان حایگزین های دیگری نیز نام برد. اختیارات محدودتر است و برای ارتباطات نامحدود مناسب نمی باشد.

منیتور کنترل بهتری روی منبع دارد و روی عملیات هم کنترل بهتر دارد پس اختیارات نامحدودتری می شود.

monitor یا منیتور

۱. هر موقع متغیر به یک فرایند داخل monitor فعال باشد دیگر فرایندها اجازه ورود ندارند و منتظر می مانند تا فرایند فعال monitor را ترک کند.

۲. تعداد این صف ها متغیر می تواند چندین تا باشد

۳. برای هر یک از صف ها متغیر شرطی تعریف می شود که فرایند فعال در صورت لزوم می تواند بر روی آن متغیر شرطی بلوک شود.

فرایند فعلی یا خارج شده است یا خوابیده.

این صف ها متغیرهای شرطی هستند.

سوال) تولید کننده و مصرف کننده.

```
/* Program producer Consumer */
monitor boundedbuffer;
char buffer[N];
int nextin, nextout;
int count;
cond notfull, notempty
```

متغیرهای داخلی منیتور

buffer[N]; → می تواند هر چیزی باشد char

nextin, nextout, count; → Semaphor mutex

notfull, notempty → condition

Subject:

Year. Month. Date. ( )

```
void append (char x)
{   if (count==N)  wait(mutex)  → شرط بیرون
        cwait(notfull)
    buffer[nextin]=x;
    nextin=(nextin+1)%N;
    count++;
    csignal(notempty)   signal(mutex)
}

void take(char x)
{   wait(mutex);
    if (count==0)
        cwait(notempty);
    X=buffer[nextout];
    nextout=(nextout+1)%N;
    count--;
    csignal(notfull);   signal(mutex);
}
    nextin=0; nextout=0; count=0;
```

برای monitor ← consumer برای producer برای

```
void Producer()
{   char x;
    while(true){
        produce(x);
        boundedbuffer.append(x);
    }}
```

Subject:
Year. Month. Date. ( )

```
void Consumer
{ char x;
  while (True) {
  boundedbuffer.take(x);
  consumer(x);
  }}

void main()
{
  parbegin(producer, consumer);
}
```

استفاده از این monitor ساده‌ی هست. این اضافه‌ی بودن که مشخص شده است.
در keyword ، monitor هست که وقتی این کلمه compiler می‌بیند، compiler میدونه این
keyword یعنی که یک کلاسی باید بسازه. اگر اضافه شده بود این یکی رو نمی‌شد باید یکسری چیزها رو اون اضافه
شده تا صراحت بشه دارد. چیزهایی مثل ✱ را اضافه می‌کنند

جمله‌ای که از monitor استفاده می‌شود باید از compiler به معنای خاصی برای monitor داشته باشد
استفاده کند. ... مدت زیاد ترین دستور wait هم چیزهای اضافه‌ی می‌شود

مثال) فیلسوف‌ها ← می‌توانند thinking ، eating و hungry شرط‌اند

```
monitor Dining Philosophers.

enum itemation { hungry, eating, thinking };
int.cion state[5];
condition self[5];
```

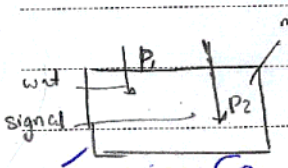
Subject:
Year. Month. Date. ( )

```
void pickup(i)
{ state[i] = hungry;
  test(i);
  if(state[i] != eating)
      cwait(self[i]);
}

void putdown(i)
{ state[i] = thinking;
  test((i+1)%5);
  test((i+4)%5);
}

test(i)
{ if(state[i] == hungry && state[(i+1)%5] != eating &&
                           state[(i+4)%5] != eating) {
    state[i] = eating;
    csignal(self[i]);
  }
}

for(i=0; i<5; i++)
{ state[i] = thinking;
}

void P(i)
{ while(True) {
  DiningPhilosophers.pickup(i);
  eating();
  DiningPhilosophers.putdown(i);
  thinking(i);
}}
```


P4PCO

Subject:
Year. Month. Date. ( )

(ادامه) زمانی که یک proc P_1 وارد یک monitor می‌شود و سط کار wait می‌کند از طرف دیگر P_2 وارد monitor می‌شود پس از سری P_2 متوجه می‌شود که P_1 یک wait شده است پس از این لحظه P_2 باید وارد صف شود یا P_1 بیدار شود یا P_2



جواب صحیح سوال چهل است باید P_2 را متوقف کرده است تا بیدار شدن P_1 را سیگنال کرده است تا بیدار شدن P_1 حداکثر تاخیر کند

پس ما باید کاری کنیم که سیستم به حالتی برود که بهترین راه حل است که signal کردن P_2 آخرین کاری باشد که P_2 می‌کند و پس از آن سیستم برود به کاری که بهتر است بتواند چنین کاری کند

مثال 2 کدام راه حل را اجرا می‌کند

1- monitor که عملیات را اتمیک می‌کند وقتی که این monitor را فراخوانی می‌کند

2- نیازمندی‌های compiler داریم مثلا اگر بخواهیم یک برنامه بفرستیم برای کرنل حتما نیاز است که از قبل نیازمندی‌های OS نیز داریم که compiler ما با compile می‌کند سپس برای کرنل می‌فرستد که اضافی‌ها را حذف کند پس باید OS هم داشته باشد

3- اینکه P_1 باید بیدار شود یا P_2

جاوا monitor را شناسنده

کلمه monitor را نداریم ولی synchronized یا object ترجمه این خاصیت را دارد و ما هم چنین چیزی را با compiler خود اضافه می‌کنیم

Subject:
Year. Month. Date. ( )

جلسه نوزدهم 8, 28

OS های موجود بطور مستقیم از معماری پشتیبانی نمی‌کنند بنابراین داشتن حسگر است.

compiler ← مانند monitor پشتیبانی

semaphore ← مانند پشتیبانی OS دارد

اگر بتوانیم برای یک library routine تعریف کنیم کار ما راحت‌تر شود.

استفاده از فراخوان‌های سیستمی Send ، receive ← تبادل پیام message passing

این راهم OS ها دارند و نیاز به پشتیبانی خاصی ندارد خیلی ساده است ولی کندتر است (حسگر است)

دو تابع داریم و می‌توان از هر زبانی به زبانی دیگر message ارسال کند

این کار توسط OS انجام می‌شود و ما نیاز به حسگر نداریم

ارتباط است

message یک آرایه‌ای از بایت‌ها است (pointer) ساختار پیچیده‌ای نیست

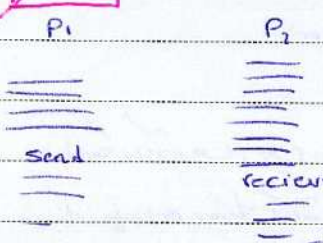
Send (m, P2) ← ارسال header این محتوا به کجا برود

receive (M) ← message دریافت کرد. ← هر حق دارد message بنویسد

تابع Send ، P2 را صدا می‌زند و message را برای Send

Send ، زمینه متعدد حتماً باید receive را اجرا کند

mailbox



حتی اگر Send زودتر از receive اجرا شد receive های Send شده جایی ذخیره می‌شوند

انتظار برای Send باید زودتر از receive اجرا شد

این شرط سنگینی است

راه حل: (در جلسات بعدی Send باید منتظر receive بماند)

برای هر زبان OS یک جعبه پستی mailbox برای دریافت پیام‌ها دارد هر زبانی که بخواهد پیام بفرستد mailbox دارد

هر وقت دستور receive را اجرا کرد یک پیام از mailbox دریافت می‌کند

(چند message ها در mailbox ها صف می‌شوند)

receive از نوع سیستم blocking call است یعنی تا زمانی که پیامی نیاید کار خود را متوقف می‌کند

receive از نوع سیستم blocking call است یعنی تا زمانی که پیامی نیاید کار خود را متوقف می‌کند

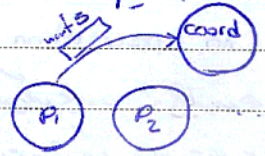
Subject:
Year. Month. Date. ( )

مثال حل مسئله producer consumer

```
void Producer ( )
{
    int item ;
    message m;
    while (True) {
        produce (item);
        recieve (m);
        build_message (m, item) ;   ← item را در message قرار می‌دهد
        Send (consumer, m) ;        → می‌گوید که send شود
    }
}
```

برای consumer هم مشابه

```
void Consumer ( )
{
    int item, i;
    message m;
** for (i=0; i<N; i++) Send (producer, m)
    while (True) {
    → recieve (m) ;                 recieve می‌شود  Send برای پروسس
      extract_item (item);          item را از message استخراج می‌کند
      Send (producer, m);
      consume (item);               مصرف کردن
    }
}
```



فرض می‌کنیم mailbox نداریم حالا تا send کنیم می‌توانیم recieve کنیم.

← recieve(m) در producer یک مجوز است که consumer می‌گوید من جا دارم بفرست و وقتی جا نباشد producer در این جا معطل می‌شود

← برای اینکه این دو با هم جلو روند در ازای هر دریافت consumer مجوز صادر می‌کند و این طوری است که تولید می‌شود مصرف می‌شود تولید می‌شود و تا وقتی مصرف نشده چیزی تولید نمی‌شود

** در این خط قبل از هر کاری N تا مجوز می‌فرستد پس هیچ وقت یعنی در recieve در producer گیر نمی‌کند چون که به mailbox ها پر است و هیچ وقت block نمی‌شود. اگر ۱ مجوز بفرستیم با این معنی است یا این یکی تو است در این جا producer می‌تواند تولید کند ولی ندارد باشد که consumer مصرف می‌کند

← ما می‌توانیم Semaphore را با استفاده از این بسازیم
اگر بتوانیم سمافور را شبیه‌سازی کنیم می‌توانیم تمام کارهایی را که می‌کردیم الان هم انجام بدهیم

شبیه‌سازی Semaphore

تعدادی فرایند هستیم که می‌خواهیم از یک Semaphore مشترک را اجرا کنیم

rec
open message
wait Operation
if not negative send

[illegible]
یکی از این‌ها باید ~~coord~~ Semaphore را اجرا کند به این Semaphore: یک فرایند کلی ایجاد می‌کنیم. فرایند کلی در حالت recieve قرار می‌گیرد (block)

منتظر می‌مانیم شاید چند تا منتظر بزند signal کند
P1 یک message را wait درست می‌کند به coord می‌فرستد بلافاصله در حالت rec می‌رود

P1 : wait(S)

messag | wait | S | .... |

coordinate message را بررسی می‌کند می‌بیند wait بوده است

send(coord,)
recieve => block شد

پس wait را اجرا می‌کند اگر منفی نشد send می‌کند
و در حالتی که از شمارنده کم کند اگر منفی شد چیزی برایش نفرستاده است (مثل wait است)

Subject:
Year. Month. Date. ( )

اگر بخواهیم بررسی کنیم monitor را چطور پیاده سازی کنیم (اگر OS آن را نداشته باشد)

مثال 5 ← 2, 9, 5, 1

جلسه بیستم

بن بست Deadlock

این بحث را نباید در کلاس های تک برنامه ای مثال هایی که داریم مطرح نیست چون اگر OS درخواست می کند البته خودش هم می تواند اقدام کند ولی اگر این چنین نیست ـ البته این بحث در جای خود بحث است.

اگر 4 شرط زیر برقرار باشد بن بست پیش می آید

1. انحصار متقابل mutual exclusion یعنی اگر منبع انحصاری نباشد بن بست پیش نمی آید. (مثالی از منبع غیر انحصاری همان تم است)

2. نگهداری و انتظار hold & wait ـ دو تا منبع داریم برای هر پردازه دو تا فرمان داریم اگر یکی را صدا بزنیم که یکی از فرمان های مال یک منبع را از دست می دهیم اگر را صدا می زنیم تا فیلیپی را بدست بیاوریم) وقتی همه پیش می روند که همه فرمان ها را بدست بیاورند و هیچ کس پس نمی دهد

3. منتفی نباشد no preemption

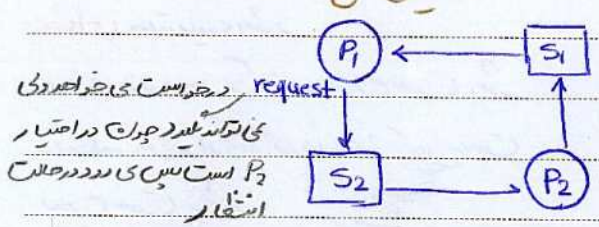
منتفی نباشد است که اگر یکی را داشت دیگر هیچ اتفاقی نمی افتد ولی هر منبع این چنین نیست (اگر یک منبع را داشته باشد تا وقتی تمام شود نمی توان از آن گرفت این است بن بست پیش می آید ولی اگر بتوان آن را منتفی کرد دیگر بن بست پیش نمی آید)

4. انتظار چرخشی circular wait

شرط ساختاری نیست بلکه نحوه است دو تا پیش می آید هر چه را که یکی داریم این یکی را بدست

Subject:
Year. Month. Date. ( )

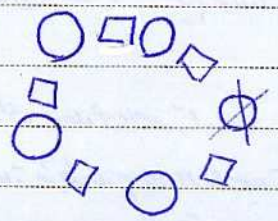
allocation تخصیص منبع



درخواست می‌خواهد ولی نمی‌تواند بلید چون در اختیار P_2 است پس می‌رود در حالت انتظار

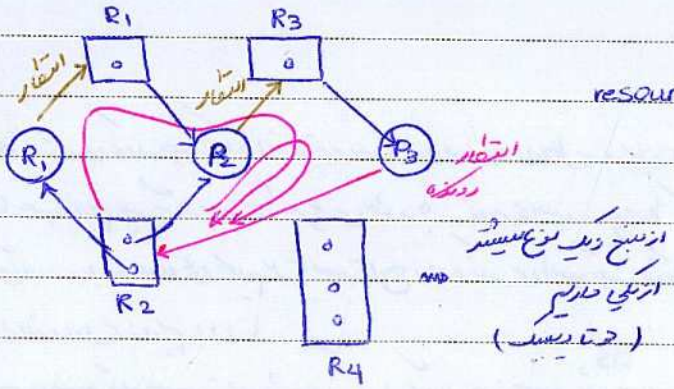
حالا P_2 هم درخواست دارد ولی در اختیار P_1 است پس می‌رود در حالت انتظار

مثال: سیستم منابع



در مثال سیستم حالت انتظار چرخشی بود که در آن انتظار چرخشی پس 5 تا فرمیز است ولی چیز راهی هم ندارد که این یکی را بازدولی ببیرا هیچ کس

ساده



تخصیص منابع +
request +

R_1 را می‌خواهد ولی در اختیار P_2 است می‌رود در انتظار

مثال:

resource ← R منبع

انتظار دوگانه

از منبع زیاد نوع هستند

AND

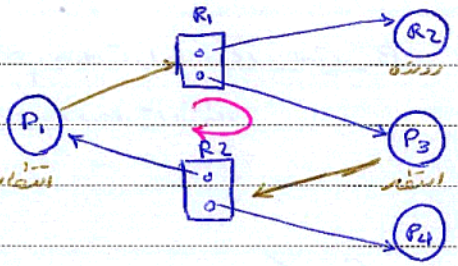
از یکی داریم

(حتما بیشتر)

در این وضعیت سیستم در حالت بن بست نیست چون ما امیدواریم (مطمئن نیستیم) که P_3 تا آخر برود و منبع را آزاد کند و بعد P_2 تا آخر برود P_1 تا آخر. همه می‌شود.

* حالا اگر 3 تا در انتظار بشند و صفر هم الان بن بست داریم. در حالت بدون بسته

Subject:
Year. Month. Date. ( )



مثال) حلقه دیده می‌شود

اما لزومی ندارد همه آن‌ها از این بن‌بست باشند

در حلقه، هر تعداد فرایندی که در حلقه‌اند، گیر این بن‌بست باشند

با وجود حلقه در بن‌بست نیستیم می‌توانیم امید داشته باشیم که P_2 تا R_1 را آزاد کند پس P_1 می‌تواند R_1 را بگیرد و اجرا شود

راه‌برای خلاصی است.

← وجود حلقه شرط لازم برای این بن‌بست است اما کافی نیست

(اگر این بن‌بست باشد حتماً حلقه هست)

راه‌حل‌های این بن‌بست

1. در OS های تک پردازنده‌ای به‌کاری نداریم چون کم اتفاق می‌افتد ← بدترین راه‌حل. در این حالت چون راه‌حل نداریم یعنی تشخیص این بن‌بست هم نداریم پس اگر پیش آمد چی‌می‌شود؟ هنگ می‌بیند hang کرده پس یا process را از بین می‌برد یا reset می‌کند. و یا اصلاً می‌گوییم این بن‌بست رخ داده فکر می‌کنیم یک مشکلی پیش آمده و راه‌حل هم نیست راه‌حل ارائه دادن نداریم!!!)

در تک پردازنده چون یک CPU داریم خود این بن‌بست پیش می‌آید (ولی الزاماً چند CPU بود ...)

2. الگوریتم بانکدار ← اجتناب از بن‌بست Dead lock Avoidance

سیستم را طوری تعیین می‌کند که به بن‌بست نزدیم

3. تشخیص این بن‌بست (متضمن خطر) و راه‌حل می‌دهد خود همین گفتیم قدم اول است که معمولاً ساده هم نیست)

4. جلوگیری از این بن‌بست Dead lock Prevention ← اصلاً در سیستم‌هایی که پردازنده‌ای که در آن دارد ... در سیستم‌های توزیع شده دارد چون به شدت ایراد و سنگ دارد

طراحی اولیه سیستم به گونه‌ای که یکی از این ۴ شرط حتماً وجود نداشته باشد ← نقض تحمیلی یکی از این ۴ شرط

Subject:
Year. Month. Date. ( )

الگوریتم بانکدار: که روی آن سابقه نیازی داده شده در هیچ جا کاربرد ندارد و دیگر برای سوال طراحی کردن خوب است! (ی شود بچپانید)

منابع را به جوری تخصیص می دهیم که بعدا تمام بن بست نروی. از ترس اینکه بن بست نشود منبع را ندارد توهم داری نه نه می زنی ریخت می ره!!!

مثال زیبا (در مورد خرید) همان حسینی را به امید پرویز!!!

تشخیص بن بست در سیستم های توزیع خیلی صرف دارد (به ساختن گراف تخصیص منابع) وجدا از تشخیص باز است به عقب از بن بست خلاص می شود (به جایی که هنوز بن بست رخ نداده ...)

(جزو درس تخصیص نیست)

جلوگیری از بن بست در سیستم های توزیع شده دیده شده روی زیاد رایج نیست

مثال برای الگوریتم بانکدار:

سه فرآیند P_1 و P_2 و P_3 یک منبع R_1 که 12 مورد از آن موجود است.

از قبل این اطلاعات اولیه به ما داده شده به جدا گفته اضافه از منبع R_1 به صورت زیر است:

(هم نیست از کجای آن را نشنید)

P_1 10
P_2 4
P_3 8

15

حالت فعلی در زمان T

P_1 5
P_2 2
P_3 2

در این لحظه

این حالت به است یا خوب؟ یعنی آیا احتمال دارد برای همه بن بست پیدا بشه؟

این (safe)

95 عضوی
3 موجودی

Subject:
Year. Month. Date. ( )

باید ببینیم با این 3 تا می‌توانیم هیچ یک از این 3 فرایند ها را به پایان برسانیم

- با انتخاب P_2 به آخری رسید.

↓ حالا که تمام شد حرجوری 5

P_1 را با 5 تا می‌توان به آخر رساند انتخاب P_1 به آخری رسید

حرجوری 10

P_3 هم به آخری رسید

این حرجوری ما یک مسیر پیدا کردیم، یعنی اگر به این ترتیب اجرا کنیم به آخری رسید

ترتیب امن ⟨Safe Sequence⟩ $\langle P_2, P_1, P_3 \rangle$

پس بله این حالت امن است.

ادامه مسئله: P_3 تقاضای یک چیز اضافه می‌کند، آیا حالت جدید امن است؟

P_1	5
P_2	2
P_3	3

حرجوری 2

انتخاب P_2 به آخری رسید

حرجوری 4 ← با این حرجوری P_1 یا P_3 را می‌توان به آخر رساند ترتیب امن وجود ندارد

پس حالت جدید امن نیست

الگوریتم بانکداران می‌گوید اگر P_3 تقاضای یکی اضافه کرد چون حالت جدید امن نیست یعنی احتمال بن بست هست، این یکی اضافه را به او نمی‌دهیم و می‌گذاریم در حالت انتظار بماند

(خلاف ترتیب P_2, P_1, P_3 تقاضا کرد)

برای همین منبع به درد خوری است چون منبع را راست کی از دری ترس ندارد ①

از کجای دانیم فرآیند چه‌قدر چیز تا از یک منبع می‌خواهد حتی اگر 100 تا باشد؟

مجبور می‌شویم تازه دست بالا هم ⇒ شده با شد اگر دوری فرق کند، این هم فرق دارد ②

نکردیم

Subject:
Year. Month. Date. ( )

هوش بیست یکم

الگوریتم بانکدار (الان)

n فرایند

m منبع

بردار available[m] منابع از نوع k را j-تا موجود ← Available[j] = k

ماتریس max[n,m] ← هر سطر آن یک max یک فرایند

ماتریس allocation[n,m] ← سطر i ام بیانگر فرایند i ام ← چه مقدار از منابع استفاده کرده

ماتریس تخصیص

ماتریس claim ← هر فرایند چه مقدار نیاز دارد تا به max برسد claim = max - allocation

این سناریوی ماده هم به اندازه دهم برحسب تغییر است. فرایند ما این است ایجاد شود تا به این برسد

می‌رویم سمتی که ... چیزی که ... چی کار کنیم

وقتی می‌گیم request i رسیده می‌خواهیم چی کار کنیم؟

یک برداشت

از max بیشترین چه؟

Requesti ≤ claimi

این چیزی است که ما فکر می‌کنیم نیست نه غیر قابل‌پیش‌بینی باشد

این برداشت به وجود غیر این صورت خطا است

2- اگر Requesti ≤ Available ادامه بدهیم ← منابع را ریکوئست بدهیم یا نه (حتما نیاز هست)

3- به حالت جدید می‌رسیم بررسی کنیم این حالت

حالت جدید ←

Available = Available - Requesti

Allocationi = Allocationi + Requesti

Claimi = Claimi - Requesti

اگر حالت جدید امن باشد تخصیص انجام می‌شود در غیر این صورت

چه چیزی تضمین این است؟

Subject:
Year. Month. Date. ( )

۴

الگوریتم جهت ایمنی:

1. work = Available
   Finish [i] = false برای i = 1, ..., n
2. دنبال فرایندی می‌گردیم که Finish [i] = false و claim_i ≤ work
   اگر پیدا نشد به مرحله ۴ برو
3. work = work + allocation_i
   به مرحله ۲ برو
   این مرحله یعنی فرایندی که به اتمام رسیده، منابعش آزاد شود و بین جمعی آزادی شود
4. اگر برای i = 1, ..., n، finish[i] = True باشد ← ایمن است

اگر حالت امن نباشد به معنای این نیست که حتما حالت انتظار است چون ممکن است این بن بست نباشد
یعنی در حالت کل منابع را دارد ولی اجازه استفاده نمی‌دهد ← کاری که به بن بست نرسد

مثال: فرایندها P_1 ... P_5

منابع A B C

کل منابع ما 10 5 7

مقدار max هم جزو داده‌های مسئله است

	A	B	C
P_1	7	3	2
P_2	3	2	2
P_3	9	0	2
P_4	2	2	2
P_5	4	3	3

مرحله ۰: سیستم را تشکیل و allocation را این صورت پر می‌کنیم

Subject:

Year. Month. Date. ( )

	A	B	C
P_1	0	1	0
P_2	2	0	0
P_3	3	0	2
P_4	2	1	1
P_5	0	0	2
	7	2	5

← T لحظه‌ای allocation

=> Available = A B C = 3 3 2

2 1 0

7 2 5 => چیزی که تخصیص شده

	A	B	C
P_1	7	2	5
P_2	1	2	2
P_3	6	0	0
P_4	0	1	1
P_5	4	3	1

max-allocation ← need (Claim) ترتیب

available 3 2 1

1. آیا این حالت امن است؟ احتمال اینکه بن‌بست برسد هست یا نه

need را نگاه می‌کنیم ← P_2 و P_4 می‌توانند بدون منتظر به آخر برسند

یک ترتیب امن پیدا کنیم لزوماً منحصر به فرد (unique) نیست ولی حداقل P_2 را انتخاب کنیم و اجرا می‌شود

و در نتیجه P_2 موجودی خود را آزاد خواهد کرد پس داریم

Available = 5 3 2 ✓ 5 2 1

انتخاب P_4 7 4 3

انتخاب P_3 10 4 5

P_1

P_5

یک ترتیب امن پیدا شد پس حالت امن است $\langle P_2, P_4, P_3, P_1, P_5 \rangle$

Subject:
Year. Month. Date. ( )

2. $Request_1 = 0\ 2\ 0$ می‌شد، به حالت جدید این حست یا نه؟

	allocation			need (claim)		
	A	B	C	A	B	C
P_1	0	3	0	7	0	5

available => 3 1 2

بیدیم این حالت این است یا نه

در این حا P_2 را می‌توانیم انتخاب کنیم پس باید P_4 را انتخاب کنیم

available	5 2 3	P_4	انتخاب
"	7 2 3	P_2	"
"	10 2 5	P_3	"

:

این حالت هم این حست که آنرا پیدا کردیم این نبود.

اگر خواهیم ترتیبی انتخاب کنیم باید در Safe Sequence باشد و بنابراین حست به حالت این نیست را نشان می‌دهیم.

تشخیص این نیست Deadlock Detection

الگوریتم قبلی باید برای هر request پیدا می‌شود و طبیعتاً request باید اجرا شود که این کم وقت پردازنده‌ای گیرد

این الگوریتم هر زمان قبل اجرا شدن و هر وقت اجرایی می‌گوید این نیست یا نه یک بار اجرا می‌شود که ما هر چند وقت یک بار می‌خواهیم

این نیست یا نه از اطلاعات موجود استفاده می‌کند

شبه کد این الگوریتم قبلی است.

1. work = available
2. اگر allocation ≠ 0 آنگاه Finish[i] = false
   وگرنه Finish[i] = True
   (از نشانگذاری برای اعضای حقیقی دارند استفاده می‌کند)
   از متغیری استفاده می‌کنیم که هنگامی که allocation صفر دارند برحسب می‌دهیم
   یعنی هیچ نیازی ندارد

Subject:

Year. Month. Date. ( )

الگوریتم بانکدار برای تشخیص امنیت بودن سیستم مناسب است ← در اصل می‌خواد محاسبه کنه که شرط امنیت برقرار هست یا نه

2. از یک نوع‌های پیمایش کن ← Finish[i] = false ← یک چیزی پیدا کن که هنوز نتونستیم تخصیص بدیم

request_i ≤ work

در غیر این صورت برو مرحله 4

3. work = work + allocation_i ← فرض می‌کنیم بهش تخصیص دادیم

Finish[i] = True

برو مرحله 2

4. اگر همه Finish ها True هست ← سیستم امن هست

اگر ... ← ... امن نیست

اگر Finish[i] = True برای همه i ها ← امن نیست

این الگوریتم برای پیدا کردن امن بودن سیستم هست

← ممکن است request ها امن نیست ولی ما اون رو تشخیص نمی‌دیم

مثال: فرایندها $P_1 \dots P_5$

منابع A B C

7 2 6

می‌دان request نداشته باشیم به اون نیاز نداریم

	allocation			request		
	A	B	C	A	B	C
P_1	0	1	0	0	0	0
P_2	2	0	0	2	0	2
P_3	3	0	3	0	0	0
P_4	2	1	1	1	0	0
P_5	0	0	2	0	0	2
	7	2	6			

← یک‌سری request ندارد

available 0 0 0 ← هرچی داشتیم تحویل دادیم

P4PCO

Subject:
Year. Month. Date. ( )

در این سیستم یا سیستم ؟

انتخاب P_1 یا P_3 چون هر دو صف هستند

	A	B	C	
available	0	1	0	
انتخاب P_1 ←				
P_3 ←	3	1	3	
P_2 ←	5	1	5	
P_4				
P_5				

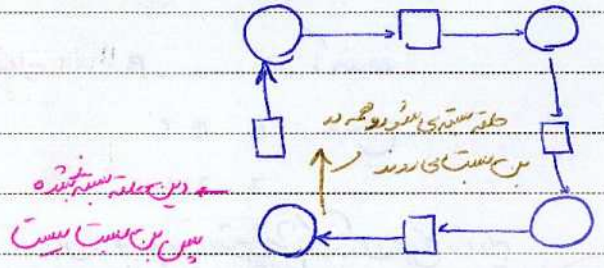
پس System در این سیست نیست، همگی True شدند

Request $_3$ = 0 ، 0 ، 1 ← زمانیکه درخواست یک واحد از نوع C دارد

ماتریس request به P_3 0 0 1

انتخاب P_1 0 1 0 ← پس چیزی برای ندادیم به آخر رسیدیم پس سیستم داریم

← P_5 ، P_4 ، P_3 ، P_2 ← این سه حلقه هستند



می‌دانیم این سیست وجود دارد که گیر این سیست
است ولی در اینجا در این راهکارها برای شکستن حلقه
استفاده کنیم

← سیستم خلوت ← ready Queue ← خلوت‌تر است

" شلوغ ← " " ← برای request زیادتر است

اگر بتوانیم این سیست را با کدام سیست اجرا کنیم ؟

در سیستم خلوت ... در آنکه شلوغ است احتمال دقت بیشتر پس را کاهش

← شاید این خلوت بتوان این سیست را شکست پس منطقی است که این را بیشتر اجرا کنیم

Subject:
Year. Month. Date. ( )

جلسه بیست و یکم 87, 9, 12

ترین OS برای جستجو

مدیریت حافظه: Memory management

← جلسه بعدی نرم‌افزار حاضر بین ست ← دانشگاه جلسه دوم ← پنج شنبه 28, 9 ← ساعت 8

تعریف فضای حافظه، آزادسازی حافظه، پیگیری فضاهای استفاده شده، پیگیری فضای آزاد

← وظایف مدیریت حافظه

جابجایی relocation

هر زمان برای ما ترانیم برجسب های حافظه را آزاد بود تر در جسم

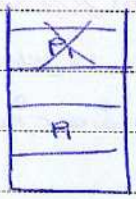
← کد، Data، PCB ... ← یعنی هر آنچه داخل حافظه

این جابجایی قابل جابجایی هستند در جای حافظه ای ترانیم قرار جسم

وقتی چیزهایی می‌شود جابجا کارد جسم کمی است دردسر ایجاد کند، دقت بکار ببریم و حد برنامه جسم های خودم مختلف از

تابع دهی استفاده نیم (نسبت به Base address)

تابع دهی های دیگر ← از تغییر حافظه آدرس استفاده نشود



حفاظت protection

برنامه ها نباید به فضاهای غیر مجاز دست پیدا کنند

این جاها باید حفاظت کنند

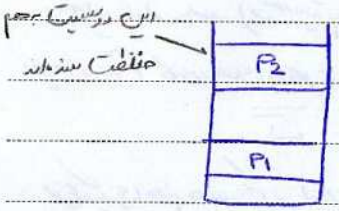
گاهی این ها نیاز دارند که مشترک استفاده کنند و حفاظت جست

می‌توانند از بعضی چیزها به صورت مشترک استفاده کنند ولی حفاظت شده

← اشتراک sharing ← از این طریق می‌توانند هم data رد و بدل کنند

حافظه های مجازی ← logical organization

physical "



Subject:
Year. Month. Date. ( )

باید این دو تعاریف قایل شویم

در تخصیص حافظه ممکن است بیوسته نباشد و هر بخش فرایند یکی باشد ← فیزیکی

ولی ریسورس منطقی، نظری شده که پشت سر هم است

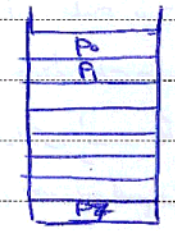
چیزی که واقعاً هست ← physic

چیزی که به نظری آید که هست ← logical

روش های مدیریت حافظه

۱. روش ایستا (static)

این روش حافظه از قبل تقسیم بندی شده ← بخش بندی ثابت (partitioning)



→ هر بخش حافظه ای است و هر کدام را آدرس می دهیم 8

فرایند داشته باشیم که فضا هایش هم محدود است

این روش خیلی محدود شده ولی خیلی ساده است

در سیستم هایی که کاربر محدودی دارند می توان از این استفاده کرد چون کمتر برنامه ها زیاد باشند یا اینکه یک برنامه خیلی بزرگ باشد (در سیستم های general purpose)

این روش کاربرد های خاص دارد به تعداد شان هم زیاد هست ← روش خیلی ساده مدیریتی دارد ← 8 تا دارد

✓ این فضا خالی می هست

به embeded (سیستم هایی که کاربری دارند)

نوع دیگری هم هست که اندازه partition ها ثابت است ولی اندازه همان بخش ها هم فرق دارد

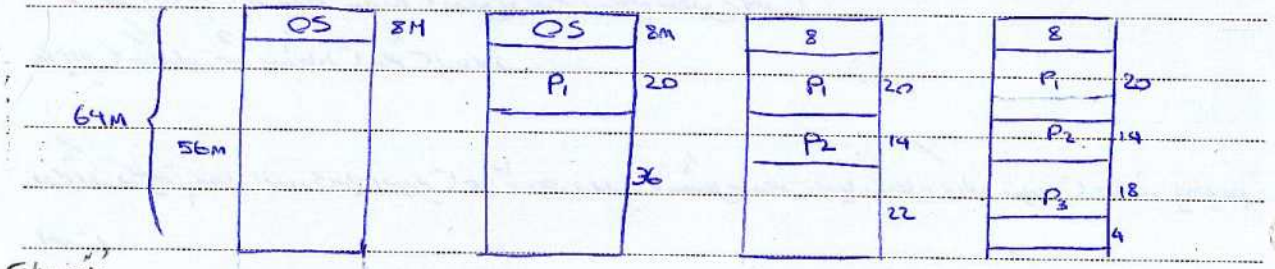
بخش بندی ثابت

اندازه های متفاوت

Subject:
Year. Month. Date. ( )

2. روش پویا Dynamic

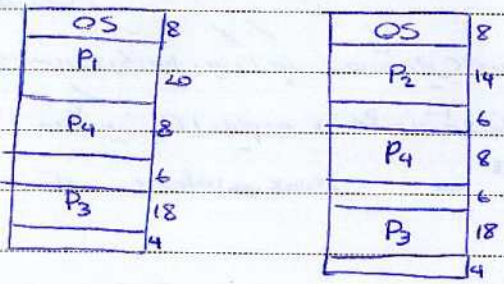
بخش بندی حافظه متغیر ← فضای حافظه یک فضای پویا است که به صورت پویا می‌توان آن را بخش بندی کرد.



مثال توضیح می‌دهیم

افزایش زمان ←

در این حافظه P4 جای P2 می‌خواهد ← داریم پس P4 ۱۰ تعریف می‌شود ← پس از آن فرایند P2 را خلوت می‌کنیم، فضای P2 خالی می‌شود، حالا فضای حافظه P2 دوباره وارد شده و جای می‌دهیم که P3 جا نگرفته و فضایش آزاد است



الان اگر درخواستی بیاید که ۱۹M فضای حافظه ← درخواست ← حافظه نداریم (پشت سر هم) ← ولی داریم که پیوسته نیست و ولی می‌توانیم به آن بدهیم

این پدیده که هرچی جلو برود باعث می‌شود حافظه پیوسته نباشد سبب می‌شود fragmentation می‌گویند ← fragmentation

فضاهای حافظه‌ای که در دسترس نمی‌باشند

برای حل این مشکل ← هرچند وقت یکبار عمل فشرده سازی انجام شود ← یعنی همه را یکجا متصل کند ← فضای پیوسته آزاد شود ← compaction

این کار در سیستم انجام می‌شود ولی این جا در پشت صحنه این اتفاق نمی‌افتد

← عدم نیاز به سربار ← سربار زمان ذخیره کردن اطلاعات جهت استفاده بعدی

فضای خالی پیدا می‌شود

هزینه زیاد است ← مقدار زیادی از اطلاعات که جای دیگر متصل کنیم

↓

زمان زیاد ← زمان اشغال

external fragmentation

Subject:
Year. Month. Date. ( )

Fragmentation ← در سرعت بسیار تاثیر دارد چون فضاهای پشت سر هم برای رسیدن هر چی فاصله فضاهای بیشتر باشد سرعت کندی میشود ولی در حافظه تاثیری ندارد چون عقب گردی الکترونیکی است

← پس در حافظه کمتر از این روش استفاده می کنند

* دلیل این که به کریم سازمان منطقی و فیزیکی برهم منطبق است.
تا این حالت شد مشکلات روش پویا بود.

روشی که در سیستم های امروزی برای مدیریت حافظه وجود دارد و مشکل روش پویا را حل می کند مدیریت حافظه paging است.

مهمترین روش مدیریت حافظه ←

Paging مدیریت حافظه صفحه بندی

← از نوع روش های پویا است

حافظه از نظر OS:

تمام های حسابی روی خیلی بزرگ (حافظه این حالت تقسیم بندی شده)

مثال: Frame ← حافظه 1G داریم حافظه فیزیکی تولید شده 4K باشد

مربوط به پردازنده Intel

$\frac{2^{30}}{2^{12}} = 2^{18}$ ⟹ 256k تا Frame تعداد

اندازه Frame در زمان تعیین شده است.

← نسبت به حافظه خیلی بزرگ است چندین برابر جمع فرایندی جا می شود

فرض کنیم فرایندی داریم که فضای مجازی خواهد آن را در Frame تقسیم بندی کنیم ← مثلا 4 تا Frame فضای خواهد

P_2

0	Page 0
1	" 1
2	" 2
3	" 3

← بیشتر از فضای پوشش

پس فرایندی 4 تا Page است ← می خواهیم این را بدهیم در حافظه ← فرض ندارد مجبور Frame پیشته پیدا کنیم

P_3

Page 0
Page 1
Page 2

2	P_3
	Page 0 of P_3
1	P_3
	Page 1 of P_3
6	P_3
	Page 2 of P_3
	Page 3 of P_3

← پیوستگی

یک سطر Frame که فضای بدیک Page می تواند قرار گیرد

→ هر سطر Page می تواند Frame قرار گیرد

پس یک پیوستگی نداریم Fragmentation داریم

Subject:
Year. Month. Date. ( )

در این حالت پیوستگی فضا از بین رفته پس دیگر مشکل برش پویا نمی‌شود هر Frame خالی‌ای برای هر Page استفاده کنیم.

← پیوستگی را از بین بردیم پس در حین اجرا چگونه از Page به Page دیگر Jump می‌کند؟

سوال‌های پیش می‌آید

- اندازه‌ی Frame
- آدرس‌دهی Frame
- مدیریت پروسه‌ی حافظه؟ اگر حافظه آزاد شد دیگر نمی‌دانیم چه جایی است.

می‌توانیم بخش‌های آزاد را نگه داریم که اگر پروسه‌ای آمد به اندازه‌ی آن از حافظه جدا کنیم

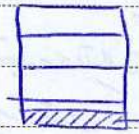
← ایراد این است که ممکن است حافظه پر شود ← الان فرض می‌کنیم که همیشه در حافظه است اگر نبود این فرض را حذف می‌کنیم

Fragmentation

خود Paging باعث Fragmentation ایجاد می‌شود

هر Page را به یک Page تقسیم می‌کنیم پس Page آخر همیشه استفاده نشده نیست و دارای بخش خالی است یعنی Page آخر خالی داشته باشد

این فضاهای خالی به نوعی Fragmentation است که قابل استفاده نیست به چیز دیگری هم نمی‌خورد مقدارش خیلی کم است پس می‌گوییم اشکال ندارد



مواجه شده

← OS از کجا بداند هر Frame مال کیست ← کجای جدول ذکر شده ← OS برای هر فرایند جدول صفحه درست می‌کند و از این جدول می‌فهمد که Page مال کیست؟

جدول صفحه P_2

Page	Frame
0	3
1	6
2	8
3	9

Page 0 من کجاست

جدول صفحه P_j

0	6
1	4
2	5

پس همیشه هر Page کجاست

Subject:
Year. Month. Date. ( )

آدرس شروع

حالا سیستم عامل از کجای حافظه این جدول کجاست ← جدول حاوی pointer ندارد PCB

برای حافظه آدرس شروع هم است و کلن است که جدول در حافظه باشد

OS همیشه می داند کدام frame ها در حافظه خالی هست.

جلسه بیست و هفتم 87/9/17

امتحان پایانی 87/10/1 ساعت 10:30 تا 12:30

مدیریت حافظه صفحه بندی Paging

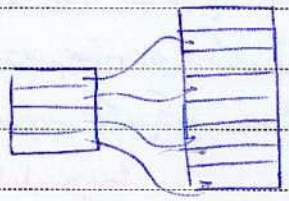
حافظه frame ها تقسیم شده

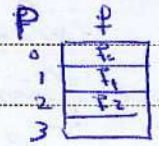
فرایندها را به page ها تقسیم می کنیم

که اندازه frame = page در حافظه قرار می گیرد

برای اینکه بدانیم page ها کجاست از یک جدول استفاده می کنیم

جدول صفحه





ترتیب frame ها مهم نیست ← جدول فضای پیوسته نمی خواهد

frame ها تمام شدند یک page را بیرون می کنیم

پیوستگی از بین رفته ← چگونه می توان فرایند را اجرا کرد Data ها را پیوسته دید

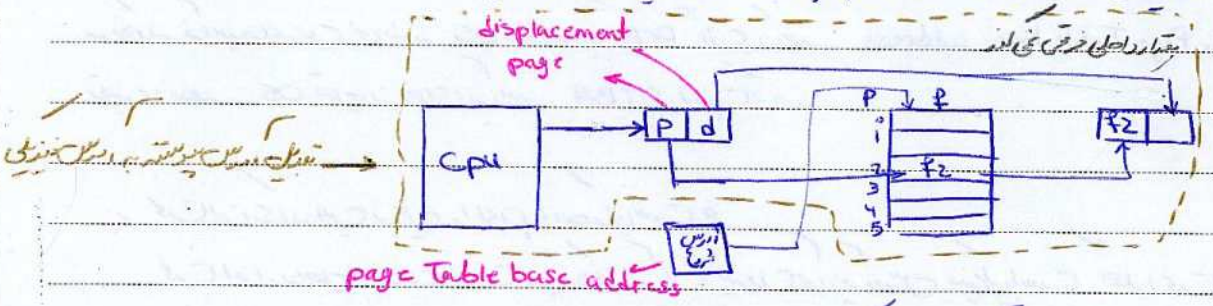
پردازنده این ها را پیوسته تصور می کند ← کارکرد افزاری

دید پیوسته به منطقی تبدیل شود

حتی آدرس منطقی به آدرس فیزیکی

Subject:
Year. Month. Date. ( )

این تکنیک عمل کند اجرای است و اگر برنامه اجرایی شود خیلی برنامه کند می‌شود.



CPU ← آدرس‌ها را برای حافظه تولید می‌کند به صورت منطقی

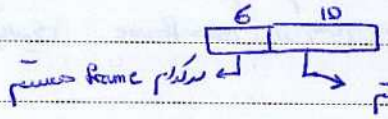
Instruction ← مربوط به
Data ← مربوط به

فرض می‌کنیم ادرک‌ها را پردازنده می‌دهد منطقی است ← پیوسته

در عمل CPU آدرس فیزیکی تولید می‌کند

برای حافظه 64K ← 16 بایت آدرس داریم.

frame ⇒ 1K ... frame ⇒ 64



هر آدرس که تولید می‌کنیم در کدام frame است ← هر آدرس

آدرس ← به صورت خطی نگاه می‌کنیم یعنی 16 بایت داریم

به صورت منطقی نگاه می‌کنیم یک 16 بایت ← به این صورت نگاه می‌کنیم

کدام page ← کجای page

page ← برای حافظه است ← از جدول صفحه مربوط به برنامه استفاده می‌کنیم

جدول صفحه است برای حافظه پیوسته نیست.

آدرس‌هایی که می‌بینیم پیوسته است ولی چیزی که در حافظه داریم می‌رویم ناپیوسته است.

OS کمک افزارهای ساده‌ای برای خودش لازم است انجام همین کارها که در بین اجرا حضور دارد اجرا می‌شود و تا پایان خود را ادامه می‌دهد

این کارها مخفی است و هیچ کس نمی‌بیند کی OS می‌آید جدول صفحه را درست کرده و می‌رود اجرای برنامه OS وجود ندارد.

Subject:
Year. Month. Date. ( )

(point) هر فرایند جدول صفحه خودرا دارد و فرایندها به همدیگر هم دسترسی ندارند

سوئیچ بین فرایندها توسط OS صورت میگیرد. OS با توجه به PCB آن فرایند Page Table base address را

تغییر میدهد OS قبل از اجرای فرایند PTBA را ست میکند

← سخت افزاری که این تبدیل را انجام میدهد چیست؟

سخت افزار داخل CPU است و چیزی که از بیرون میدهیم شماره صفحه است، ولی سخت افزار

داخل CPU هست و PTBA در CPU نیست چون در حافظه است

اندازه هر Page را طراح CPU تعیین کرده و اختیاری نیست و ما اختیاری نداریم

مشکلات حل نشده:

- میدانیم Fragmentation نداریم (داریم به طور مختصر و در آخرین Page هر فرایند)
- مشکل پیدا کردن فضای حافظه را نداریم

مزایا: قابلیت اشتراک هر Frame اجازه میدهد

کارایی Frame های آزاد سازی شده به راحتی یک آرایه حافظه 64K بیتی یک رشته 64 تایی در اختیار فرد قرار میدهد.

اندازه Page از دو جنبه بررسی میکنیم

1. اندازه جدول صفحه ← اندازه page کوچک شد جدول صفحه خیلی بزرگ میشود
2. تکه تکه شدن یا Fragmentation ← کمتر از حافظه از دست میرود این مقدار خیلی قابل توجه میشود

بعد از حل این مشکلات یک مشکل جدید بوجود میاید: مشکل سرعت حافظه و سیستم باید از دو بار دسترسی

یکبار به حافظه برای پیدا کردن آدرس فیزیکی داده باید برویم ← یکبار برای page Table

یکی برای آدرس اصلی

این مشکل خیلی سرعت را پایین می‌آورد ← مشکل حافظه CPU است و سرعت ما را کم می‌کند

این سیستم کارایی بسیار پایین دارد

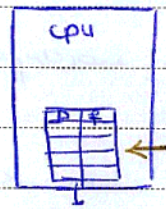
performance

Subject:
Year. Month. Date. ( )

سیستم ها سرعت ما نقطه خیلی مهم نیست یا سرعت است که تعیین کننده است
بعداً این مشکل را حل کنیم.

حل مشکل سرعت پایین:
آدرس هم مستقیم به حافظه‌ی اصلی رفته بلکه اول به cache می‌رود و در صورتی که آدرس در cache نباشد
از همین ترفند جدول cache استفاده می‌کنیم
که بخش کوچکی از حافظه را دارد درصد $\frac{1}{50}$ یا $\frac{1}{100}$ دفعه اول cache می‌کنیم

برای حل مشکل داخل خود پردازنده جدولی درست کنیم که بخشی از page table در آن باشد از نوع cache که خیلی کوچک می‌شود)



جدول داخل پردازنده
خیلی سریع است
از نوع reg است

به جای cache جدول TLB

بررسی سرعت به صورت سریال reg

مشخص است که اندازه‌اش کوچک است

TLB: Translation lookaside buffer (بافر ترجمه‌ی آدرس)

برای CPU دسترسی است.

در پردازنده Intel ← 32 تا خانه دارد و می‌تواند page table داشته باشد

در TBL ← 98٪ آدرس‌ها جواب صد تا صد و 2٪ مجبور است سراغ page table برود

اگر دستورها با سرعت بالا اجرا می‌شود

TLB به OS هیچ ربطی ندارد و OS اصلاً نمی‌داند که TBL اصلاً وجود دارد و جدول صفحه کار دارد

همه‌ی این‌ها سخت‌افزار اجرا می‌شود

برای تبدیل آدرس (سخت‌افزار)

1- ابتدا در TLB ← اگر موجود بود تبدیل انجام می‌شود

2- اگر موجود نبود جدول صفحه و update کردن TLB ← یعنی این‌ها الان سخت‌افزاری می‌شوند در TLB

Subject:
Year. Month. Date. ( )

TLB ← از سمت OS مخفی است ← سخت افزاری است

Paging ← از دید برنامه کاربری مخفی است ← OS می بیند

یک مثال: فرض کنیم تا الان P_2 کار می کرده الان P_1 به کار نیفتد داخل TLB چی هست؟ ← هم زمان شروع شود

هیچ جدول صفحه ای داخل نیست و مربوط به P_2 است دفعه پاک می شود و اولین بار به حافظه برای ارجاع می کنیم در TLB نیست برای همین که نیست اجبار است از RAM داخل PTBR قرار می گیرد

← با تاخیر می رویم (تاخیر کم زیاد) TLB خالی است ← TLB به تدریج پر می شود.

↓ مجبوریم که خودمان از طریق جدول صفحه اقدام کنیم

← جدول صفحه مربوط به P_1 نیست ← در TLB نیست ← می آید داخل TLB

خود OS هم برای خود page table دارد ولی طبیعتا اولین چیزی که در صفحه قرار می گیرد پیوسته است.

مثال: کم زمانی 1ms

6 billion Instruction / sec ← یک بیلیون دستورالعمل در ثانیه اجرا می شود

$\frac{10^9}{10^2} = 10^7$ ← در حد کم زمانی 10^7 تا Instruction اجرا می شود

↓ تعداد جداول (تنظیم) تراست چون کمتر است تنظیم آن بیشتر دستی راست تر است

TLB در کامپیوترهای بدون تنظیم چقدر است؟

این نوع ذخیره سازی در حد سرعت data هست

Content Addressable Memory → مثل cache

2	
4	
5	
1	
8	

اطلاعات در TLB را چطور پیدا کنیم؟

به ترتیب نیست چون کل pageهای

که در آن دستوری پیدا کردیم در TLB است

چیزی که از این نیست در نهایت TLB را خطا کنیم و برای

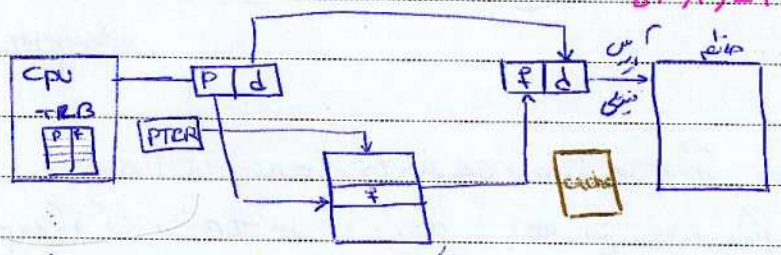
هر دستوری باید کل TLB را بگردیم ← 32 تا حتما سیم کشی و راست تنظیم (سخت افزاری)

باهم موازی کار کنند

↓ یکی از دلایل کم بودن خانه ها همین است.

Subject:
Year. Month. Date. ( )

جلسه بیست و ششم 87.9.24



یکی از روش‌های بهبود سرعت که از PCB، load و ... می‌شود این است که جدول صفحه کجاست. چون شکل سرعت پایین است باعث کندی می‌شود. برای حل این مشکل از TLB استفاده کردیم.

مثال) حافظه: زمان دسترسی 50ns

TLB ، ، 2ns

hit ratio (نسبت اصابت): 98%

میانگین زمان دسترسی را حساب کنید (به حافظه)

Teff

جدول صفحه در حافظه است پس برای دسترسی به هر داده نیاز به 2 بار مراجعه به حافظه داریم. یکبار از TLB و یکبار از جدول صفحه

$$Teff = 0.98(2ns + 50ns) + 0.02(2ns + 50ns + 50ns)$$

(زمان دسترسی TLB + زمان دسترسی حافظه) با احتمال 98% (زمان دسترسی TLB + زمان دسترسی حافظه + زمان دسترسی حافظه) با احتمال 2% خواهد بود

$$Teff = 51ns + 2ns = 53ns$$

در مقایسه با دسترسی به حافظه با سیستم 50ns زمان کاهش سیستم برای الان حدود 3ns اضافه شده که کمتر از 10% است و این خوب است تا 10% هم خوب است یعنی تا 10% را خطا بگیریم.

معمولاً سیستم‌های واقعی این طور نیست که همیشه حافظه خیلی کند است و ما راهی پیدا کنیم این را تندتر کنیم (50ns در مثال 2ns خیلی زیاد است. بیشتر حافظه‌ها خیلی کند نیستند)

برای حل این مشکل از حافظه‌ی کش یا cache استفاده می‌کنیم. cache گرچه گران‌قیمت است ولی سرعت بالایی دارد به همین خاطر کم است.

اصلاً با حافظه کش چه تناسبی دارد؟ cache قرار است چیزی از حافظه را در خود نگه دارد (به خاطر همین گفتیم حافظه کش نداریم) که ما بیشتر دفعات به آن مراجعه می‌کنیم (مثلاً Table ، data و ...) یعنی کاری که ما به حافظه داریم بیشتر دفعات آن چیزهایی است که در cache پیدا کنیم.

Subject:
Year. Month. Date. ( )

← ساختار cache ، بحث افزاری است و OS از وجودش اطلاع است و OS با حافظه کاری ندارد و تمام مدیریت‌ها روی حافظه انجام می‌شود

حافظه cache (حافظه‌ی کش): زمان دسترسی خیلی بهتر از حافظه است مثلاً 10ns

نسبت تعداد تراکم کمتر از TLB هست : 90٪ ~ 95٪ مراجع در cache مشکل پیدا می‌شود.

برای کش حافظه روی مدیریت حافظه ندارد ← در اصل بهتر از این است.

cache با حافظه برنامه‌ریزی می‌شود و در اصل ما فرض می‌کنیم 50ns به آن تبدیل می‌شود.

مثال تمرینی ← زمان دسترسی حافظه cache ۱۰ns و زمان دسترسی حافظه ۵۰ns و نسبت دسترسی به cache ۹۰٪ است. زمان دسترسی موثر حافظه چقدر است؟

$T_{cache} = 0.9(10ns) + 0.1(10ns + 50ns) = 9 + 6 = 15ns$

زمان دقیق از این بیشتر است و در اصل چیزی را برنامه‌ریزی می‌کنیم

$T_{eff} = 0.98(2ns + 15ns) + 0.02(2ns + 15ns + 15ns) = 17.3ns$

تاثیر cache در جایی است که به حافظه دسترسی داریم عدد دقیق از 17.3ns بهتر است اصل بحث چیزها را برنامه‌ریزی می‌کنیم:

ممکن است در cache چیزی را پیدا نکنیم می‌رویم سراغ حافظه این طور نیست بلکه اگر در cache پیدا کردیم از حافظه

cache ی را بردد در اصل با ارتباط بین cache و حافظه اصلی را برنامه‌ریزی می‌کنیم که این زمان برداشت، CPU با

حافظه کاری ندارد و از cache با حافظه ارتباط دارد

cache شامل دو ← L1 cache داخل CPU ← سرعت خیلی بالا و ظرفیت خیلی کم

L2 cache ی روی main board

این زمان دسترسی به حافظه هنوز هم جواب نیست یعنی تبدیل نیست. CPU سرعت پایین تری دارد و هنوز هم گلوگاه سیستم است

مدیریت حافظه به روش paging بحث شکست Fragmentation و مدیریت را حل کنیم

← نحوه‌ی فضای خالی را OS باید حساب دقیق نگهداری کند.

S_1 b_1, l_1 S_2 b_2, l_2

base ← آدرس شروع طول بخش‌های حافظه

S_3 b_3, l_3

P_4 وارد می‌شود و فضای خواهد و بیشتر از یک choice داریم

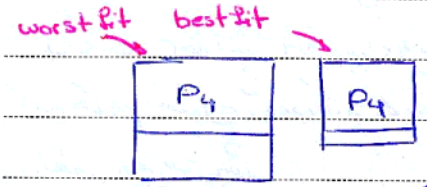
Subject:
Year. Month. Date. ( )

وی تراپیم جریمه تا از این حفره‌های ایجاد بکنیم، فضاهای را پیدا کنیم و بینیم کدام فضای خالی از همه به P_4 نزدیک‌تر است برای این شروع کنیم به Search کنیم که این Search خیلی طول می‌کشد ← روش best fit بهترین برازش بهترین انتخاب Search طولانی

روش بعدی این است که به جای Search کردن به اولین جایی که رسیدیم که P_4 می‌تواند در آن قرار گیرد آن را انتخاب می‌کنیم پس نیاز به search نداریم اولین برازش first fit

صبر کوتاه بهتر است بهترین انتخاب

روش بعدی هم چیست ← بیشترین را پیدا می‌کند search می‌خواهد بزرگترین را پیدا می‌کنیم و یعنی اینکه list را sort کردیم ← روش worst fit بدترین صبر



روش بعدی worst fit است: بزرگترین حفره را برای قرار دادن P_4 انتخاب می‌کنیم فضایی که بعد از جا دادن P_4 اندازه‌ی کافی بزرگ است که بتوان برای قرار دادن دیگر دوباره آن را استفاده کرد

روش آشنا بعدی benchmarking است ← روش ویندوز برای نوشته است.

مجموعه‌ای از برنامه‌های رایج را برای استفاده می‌شود.

روی این برنامه‌ها چیزکار می‌خواهیم انجام دهیم اجرای کنیم و اندازه‌گیری می‌کنیم و بینیم کدام یک از حافظه بهتری تواند استفاده کند ← مقدار صبر

مقدار صبر نه حافظه ← بیش از $\frac{1}{2}$ حافظه هدر می‌رود.

با این روش تخمین best و first بهتر از worst برخاستند

$\frac{1}{3}$ حافظه تلف می‌شود

روش که بیشتر از همه استفاده می‌شود روش first fit است.

← fragmentation تکه تکه شدن paging راه‌حل می‌دهد برای مدیریت بهینه فضاهای بیشتر را حل می‌دهد یک مدیریت ساده می‌دهد، بعضی مدیریت فضای بیشتری خواهیم

segment کاملاً برعکس paging بود به ترتیب حل کردیم و مدیریت غالب در سیستم‌های حافظه‌دار است

مدیریتی را الان گفتیم که سیستم‌عامل انجام می‌دهد Intel از آن استفاده شده مدیریت حافظه به برنامه چسبیده شده چون نوع تبدیلات را CPU باید انجام دهد بیشتر با CPU را مدیریت حافظه داشته باشد ← segmentation

Subject:
Year. Month. Date. ( )

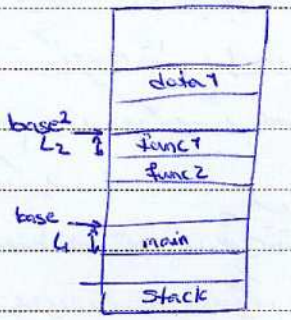
جلسه بیست و پنجم 9,26

مدیریت حافظه: قطعه‌بندی Segmentation

این مدیریت مخصوص OS است و در حال OS این را صدا می‌زند و می‌گوید باید چیزی را برداشته باشد و CPU این کار را دارد.

← قطعه‌بندی از دیدگاه برنامه‌نویسی یعنی این است که برعکس قطعه‌بندی باید دیدگاه برنامه‌نویسی اجرا می‌شود همین دیدگاه را می‌توان در مدیریت حافظه به کار برد

می‌توان این برنامه را چند قطعه تقسیم کرد



می‌بینیم که آدرس‌ها باید به هم بچسبند

که ما هر کدام جدا جدا نگاه می‌کنیم برای هر قطعه باید دید جدا باشد

این نوع مدیریت همان مشکل fragmentation را دارد. یکی شده جدول در این جا فضاهای کوچک نیاز داریم

OS از روی Seg Table می‌فهمد این قطعه‌ها کجاست.
هر قطعه یک شماره base دارد در این جا می‌آید
از شماره‌گذاری قطعه‌ها استفاده کنیم چون طول قطعه‌ها ثابت نیست

این مدیریت نسبت به paging کند تر است. باید حساب کتاب فضاهای خالی را داشته باشیم و باید برای پیدا کردن فضای خالی بگردیم.

مدیریت سازنده Intel ← بزرگ پدر سازنده 8086

نکته: مدیریت Segmentation است که خیلی ساده‌تر باید مشخص کنیم که کجاش code seg، stack seg، data seg ... است و این را ما تعریف می‌کنیم ← این خیلی شبیه سیستمی است که جدول Seg های قدیمی را دارد.
این نوع کامل‌تر است که بهتر است هر چند تا قطعه که می‌خواهیم داشته باشیم.

Subject:
Year. Month. Date. ( )

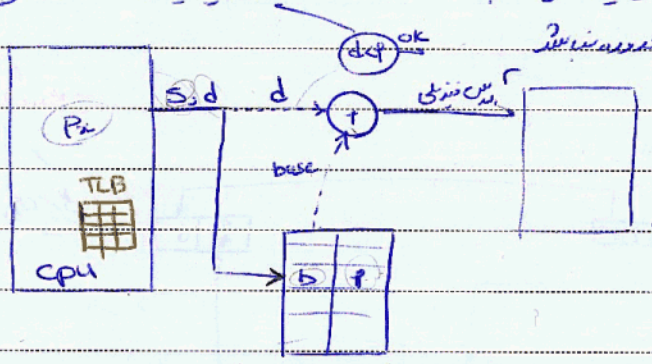
سگمنتیشن Segmentation

آدرس به دو بخش تقسیم میشود که کدام Seg و این Seg برنامه کاربری ما را میدهد

داخل این Seg کجاست.

قبل از رفتن d را ببین از حدود Seg خارج نشده باشد

و غیر این صورت آن را به بخش استثنا برای وقفه تولید می‌کند



تقریبا شبیه paging است ولی در این جا یک adder یک جمع‌کننده اضافه کنیم.

این سیستم عامل مشکلات را دارد. Intel برای حفظ این ساختار و استفاده از paging این دو را با هم ترکیب کرده یعنی به جای page کردن فضای Seg ها Seg می‌کند هر Seg را بصورت page در حافظه ذخیره می‌کند

TLB سه field دارد ← شماره Seg

base

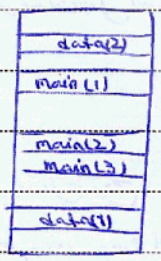
limit

پیجینگ paged Segmentation (ترکیب سگمنتیشن و صفحه‌بندی)

← سگمنتیشن حافظه صفحه‌بندی صفحه‌بندی شده

هر Seg را به چند مستقل صفحه‌بندی می‌کنیم

Seg توسط برنامه‌کاربری انجام می‌شود ولی paging به صورت خودکار انجام می‌شود



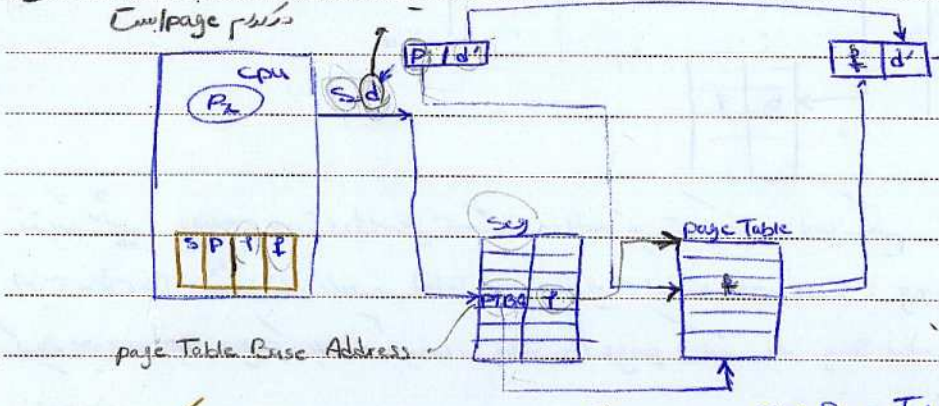
دیگه Seg نداریم بلکه بصورت page در حافظه قرار می‌گیرد

Subject:

Year. Month. Date. ( )

برای هر Seg در Seg Table یک سطر در نظر می‌گیریم که آدرس page Table آن Seg است

مزیت‌های سیستم بخش‌بندی:



این شکل نحوه‌ی یافتن آدرس فیزیکی Seg است که در کدام page است

آدرس فیزیکی

۵ ← کدام page Table برویم

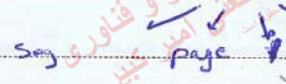
محدودیت: طول هر چک کنیم که limit را رد نکند

در این حالت برای هر Seg یک page Table داریم که کدام برای کدام Seg است ولی در paging حالتی داشتیم که برای کل فرایند یک page Table داشتیم

Seg table ← در این حالت بسته به سگمنت‌ها حافظه پر می‌شود

page ← حافظه

برای حل از سرعت‌های قبل استفاده می‌کنیم ← استفاده از TLB



کدام page مربوط به کدام Seg

های سگمنت‌هایی که

در TLB به صورت زوج (Seg, page) نگه می‌داریم ← ابتدا سگمنت Page جستجو می‌کنیم

f در واقع شماره‌ی صفحه‌های Seg است.

← حافظه‌ی تخصیص داده شده به هر سگمنت متغیر است و دیگر ثابت نیست

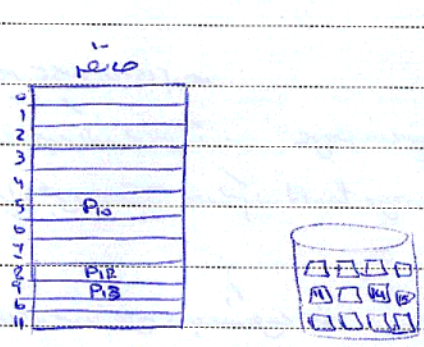
Subject:
Year. Month. Date. ( )

پردازنده Intel علاوه بر این که اجازه‌ی segmentation را می‌دهد از paging استفاده کنیم به این ترتیب که هر segment را یک page table در نظر گرفته و برای هر segment از page بندی استفاده می‌کنیم.

← دلیل اینکه همیشه حافظه مجازی می‌گیریم paging است و خیلی از مشکلات را حل می‌کند

حافظه مجازی: Virtual Memory

فعلاً فرض کردیم برای اجرای هر برنامه باید تمام فضای حافظه پر شود (فرض محدود کننده‌ای است) و هر چقدر هم برنامه بزرگ باشد چه فضای بزرگ مجازی آنرا پر کند و نتیجه‌ی مطلوب حاصل شود (برنامه اجرا شود).
حافظه مجازی این فرض را نقض می‌کند و می‌گوید لازم نیست که فضای برنامه حافظه باشد



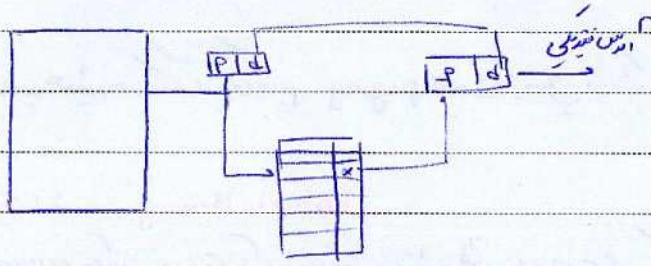
با توجه به اصل locality فقط بخشی از برنامه اجرا می‌شود و هر لحظه هم جهت page بندی می‌شود داخل حافظه می‌رویم تا وقتی به بقیه احتیاج نکرده آن‌ها را در حافظه نمی‌گذاریم یعنی در اصل بخش‌هایی از فضای حافظه پیشروی می‌کنند پس لازم داریم بدانیم منطق خیلی فضا می‌توانیم برنامه اجرا کنیم

page Table ← یک field جدید برای این page Table اضافه می‌کنیم تا ببینیم محتوای کدام خانه page table است یعنی در دیسک است یا حافظه.
valid یعنی در حافظه است و invalid یعنی در دیسک است.
جدول صفحه ساخته می‌شود (توسط OS) و مشخص می‌کند که کدام قسمت‌های آن معتبر کدام‌ها نامعتبر هستند (یعنی در اصل disk را هم به حافظه اضافه کردیم)
در واقع disk می‌شود Virtual Memory چون به ما این توهم را می‌دهد که حافظه بسیار بزرگی داریم ولی در واقع نیست
حافظه مجازی: حافظه بسیار بزرگی که واقعی نیست
از کمک disk به عنوان حافظه مجازی صحبت کردیم چون در disk فضا هست

Subject:
Year. Month. Date. ( )

حالا تبدیل آدرس چطور انجام می‌شود؟ تبدیل آدرس توسط سخت‌افزار است

تبدیل آدرس تنها وقتی با paging ندارد و مفهوم جدیدی ندارد



سخت‌افزار به بیت valid توجه کند، در صورت غیر valid بودن تولید وقفه کند

Page Fault (وقفه صفحه)

LD Rx, 349(Ry)

مشخص خواهد این دستور را اجرا کند آدرس را تولید می‌کند می‌بیند که در page حافظه نیست پس دستور قابل اجرا شدن نیست اجرای آن قطع می‌شود و به سراغ تابعی از OS می‌رویم که page fault را handle می‌کند

1. اجرای دستور قابل اجرا شدن دچار مشکل می‌شود ← page در حافظه نیست!
2. تولید وقفه page fault
3. اجرای برنامه قطع می‌شود، OS کار می‌کند ← سخت‌افزار در page fault OS کار می‌کند

از بین آن‌هایی که انجام شده‌اند کدام page است

توسط OS اجرا می‌شود:
- page را پیدا می‌کند
- از disk حافظه منتقل می‌شود
- page Table را اصلاح کند
- کنترل به برنامه کاربر بازگردانده شود

کنترل دوباره به برنامه کاربر برگردانده شود و دوباره دستور اجرا می‌شود

← سخت‌افزار آدرس OS نیست اما کمک‌افزار است که در صورت عدم انجام تولید وقفه می‌کند

اگر حافظه جای خالی نباشد نیاز است page ای از حافظه را بیرون بفرستیم و جایگزین کنیم

برای TLB، cache

برای بیرون کردن روش‌های مختلف مطرح است

PAPCO

این سوال مطرح است

Subject:
Year. Month. Date. ( )

روش بهینه‌ای که استفاده می‌شود LRU است ولی در cache از روش‌های ساده‌تر مثل FIFO استفاده می‌شود

جلسه بیست و ششم ۹۷.۹.۱

دو توان به طور خیره‌کننده‌ای نشان دار که حافظه بزرگ است.

حافظه فیزیکی $2^{32} = 4GB$ بیشتری حافظه مجازی خیلی بزرگتری تولید شد درحدود $2^{45} = 32TB$ شد

پس در نتیجه اگر OS خوبی داشته باشیم می‌توان تعداد زیادی Task درست

روش‌های جایگزینی: Replacement Policies

حافظه فیزیکی پر شد و برای استرس‌های از disk استفاده می‌کنیم اینکه چکسی را بیرون کنیم تا فرایند جدید را جایگزین کنیم این روش‌ها هست.

ساده‌ترین راه: کاری نداریم که چه‌قدر در حافظه بوده، به ترتیب بیرون می‌کنیم یعنی کسی که زودتر همه بیرون می‌رود

کاری نداریم که هنوز استفاده می‌شود یا نمی‌شود.

رشته مرجع Reference string : برای توضیح مثال

ارجاع به صفحه چاپ است. فرض کنیم سیستمی را monitore می‌کنیم تا رشته‌هایی که تولید می‌کند را ببینیم

0112 0113 0114 0614 0615 0112 0113

P d

این روش‌های کاربردی

جایی که page fault نداریم عوض می‌شود فرض کنیم که

d را حذف می‌کنیم

01 01 01 06 06 01 01

هر جا که page fault هست را نشان می‌دهیم

01 06 01 ← Reference string

اولین page fault است دومین است

Subject:

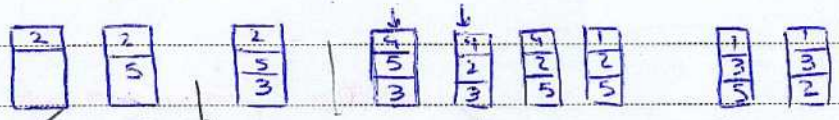
Year. Month. Date. ( )

مثال) یک رشته مرجع برای مثال ازقبل بوده

از این جا به بعد باید سیاست خاصی را به کار ببریم چون حافظه پر شده

2 5 2 3 5 4 2 5 1 2 3 2

سیاست FIFO



ازقبل بوده حشکل ایجاد نمی‌کند

به اینجا که رسیدیم این خیلی بد است

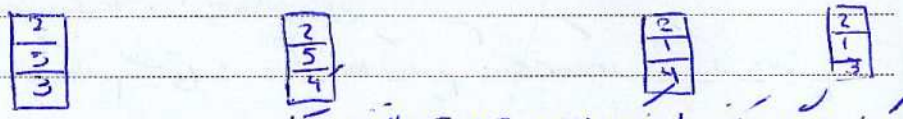
که هی از حافظه خارجی بیاوریم چیزی که هی نیاز داریم

این روش خوب نیست چون وقتی جدید تصمیم غلط می‌گیرد پس به درد نمی‌خورد (در حافظه cache از این روش استفاده می‌کنند و اجرای آن خیلی سریع است)

روش optimal (بهینه) ـ متقاضی نده

براساس چیزی که در آینده می‌بینیم عمل می‌کنیم خیلی است ولی برای بدست آوردن ترتیب خوبی داریم

2 5 2 3 5 4 2 5 1 2 3 2



آینده را نگاه می‌کنیم آن‌که از همه دیرتر است را انتخاب می‌کنیم.

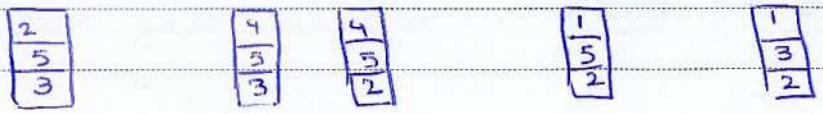
در صورتی که دو تا انتخاب داشته باشیم از روش FIFO بین آن دو استفاده می‌کنیم

برای همین این را می‌توان گفت که در مثال نوشته است و چیزی که در گذشته نزدیک استفاده شده نزدیک هم استفاده می‌شود ـ LRU

Least Recently use (کمترین استفاده اخیر)

روش LRU ـ ترجمه: آن‌که در گذشته نزدیک استفاده نشده را انتخاب می‌کند

2 3 2 3 5 4 2 5 1 2 3 2



در این جا اگر تصمیم درست می‌گیریم زمان است تصمیم غلط هم بگیریم بازهم بهتر از FIFO عمل می‌کند

Subject:
Year. Month. Date. ( )

پیاده سازی این روش آسان نیست زیرا نیاز به یک سخت افزار خاص مثل stack دارد که پیاده سازی آن سخت است و وقت گیر است به همین دلایل کم از این روش استفاده می‌کنند

FIFO الگوریتم FIFO هر چند بد نیست ولی چون هیچ اثری از استفاده نمی‌شود

optimal عیب مهم است و سعی می‌کنیم چیزی بیاوریم که بهتر از آن نداریم

LRU وقت گیر است ولی خوب عمل می‌کند

روش صفحه استفاده نشده ساعت — clocked

page Table محتویات — frame

valid bit

Accessed یا A بیت ← این را اضافه می‌کنیم

هر page که دسترسی به آن انجام شود reset می‌شود

یک pointer می‌گذاریم که به جلو می‌رود اگر دسترسی شده باشد جلو می‌رود و پیش می‌رود اما مثل الگوریتم قبل جلو نمی‌رود بیت accessed آن را صفر می‌کند اگر یک بماند بر لیست و صفر صفر نمی‌شود که کسی از این استفاده نمی‌کند پس این را انتخاب می‌کند و اگر کسی را انتخاب کرد بیت accessed را یک می‌کند اگرچه یک شد یک دور دیگر صفر می‌شود

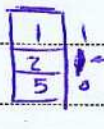
این LRU بیت دوم شکل آن است.

LRU مثال بیت‌های دوتایی

2 5 2 3 5 4 2 5 1 2 3 2

در این نمونه مثل LRU عمل می‌کند در این نمونه مثل FIFO عمل می‌کند

در این صفحه page fault چند مرتبه رخ داده است accessed



P4PCO

Subject:
Year. Month. Date. ( )

در این مثال عملکرد مثل clock نیست LRU و این چیزی است چون خود LRU نیست
در حیف و راستی هم LRU هم clock خیلی بهتر عمل می‌کند

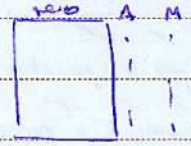
clock ← بیشتر ای کنت افزار برای صفحه چون بیت A به سخت‌افزار می‌رود. هر چیزی که set شود
page Table اضافه می‌کنیم کنت افزار به بیشتر سد به همین دلیل این بیشتر ای را برای صفحه
وی reset شدن A توسط OS است به HW ربطی ندارد.

بیت‌ها را در page Table اضافه کنیم:

بیت M: (Modify) اگر صفحه‌ای محتوی دارد در دیسک دارنده Intel ← Dirty D

بیت A: یا محتویات Page را تغییر داده یا نه (خواندن و یا نوشتن هم خواندن و نوشتن)
این هم HW ای را می‌خواند و هر وقت نوشتن این خروج set می‌شود. OS می‌تواند این بیت را بخواند
(همه‌ی HW ای را می‌خواند ساده‌ها است و OS باید از این استفاده کند و تصمیم بگیرد)

↓ اما اصلی کارهای محلی انجام می‌دهد.



برای جابجایی یک صفحه ① دیسک را روی disk بنویسیم
② دیسک را روی صفحه بیاوریم
اشتباه بین صفحه و disk خیلی زیاد است
اگر ما بتوانیم اول Frame را انتخاب کنیم که تغییر نکرده. استقال ① حذف می‌شود چون اول صفحه‌ای که
روی disk است با این صفحه‌ای است که دیگر خیلی کم می‌شود
HW این اختیار را به ما داده که ما هم حالای توانیم استفاده کنیم یا نکنیم
پس بهتر است اولی را انتخاب کنیم که access نشده و تغییر هم نکرده
A=0 M=0
انتخاب دوم A=0 M=1

Subject:
Year. Month. Date. ( )

جلسه بیست و هفتم 87, 10, 3

مثال جلسه قبل: page Fault احتمال درخواست‌های

$T_{eff} = 0.98(2ns + \underbrace{50ns}_{15ns}) + 0.02(1 - \frac{P}{0.02})(2ns + \underbrace{50ns}_{15ns} + \underbrace{50ns}_{15ns}) + \underbrace{\frac{P}{0.02}}_{1} (2ns + \underbrace{50ns}_{15ns} + \underbrace{10ms}_{\text{دیسک}} + \underbrace{50ns}_{15ns} + \underbrace{50ns}_{15ns})$

cache بعد از تغییر این عددها می‌شود.

اگر حافظه cache داشته باشیم تاثیر دارد؟ $0.9(10ns) + 0.1(15 + 50ns)$

تاثیر cache بستگی به حافظه اصلی است. هر چه سرعت حافظه و سیستم متداول را بالا ببریم

کاربرد

درصورتی که cache قابل مقایسه بود با 50 ns الان قابل مقایسه است با 15

← جایی خواهیم بیشتر که بتوانیم 10ms را کم کنیم

اگر بتوانیم برای خواندن حافظه و نوشتن (نسبت عددی 1/4 داریم)

نوشتن ← 20% ، خواندن ← 80%

← اگر بتوانیم از این استفاده کنیم می‌توانیم زیست M استفاده کنیم وقت‌هایی استفاده

کنیم که درمان نوشته نشده جمله دیسک از روی disk بنویسیم

نوشتن 20% ⟶ 10ms

خواندن 80% ⟶ 5ms

وقت‌هایی از روی disk می‌خوانیم و نیاز به نوشتن نداریم

این حافظه انتخاب شده برای حافظه‌ای تغییر نکرده باشد انتقال از روی حافظه به disk انجام می‌شود جمله لازم نیست

← بیشتر وقت‌ها می‌توان این حافظه را انتخاب کرد ولی همیشه هم می‌توان این کار را کرد بستگی دارد

این کار را به چه سیستم عملی بنویسیم

انتقال برطرف انتقال کیفیت

انجام می‌شود

$0.8(5ms) + 0.2(10ms) = 6ms$ ⟸ زمان کمتر انتقال

با این کار می‌توانیم 1ms را کم کنیم که این بهبود بزرگی در سیستم می‌شود.

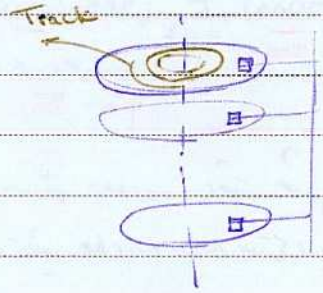
← حافظه cache جمله بیهوده می‌شود پیچیدگی ایجاد نمی‌کند

اگر p=10 قرار دهیم T_{eff} را حساب کنید (بهبود قبل خیلی بهبود پیدا می‌کند.)

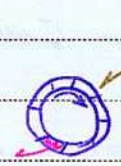
Subject:
Year. Month. Date. ( )

مدیریت دیسک

Disk ساختاری دیسکی دارد که بصورت سیلندری طراحی شده است.



هر کدام یک حداقل دارند که برای هر یک سر دیسک تماس دارد
این سرها هم همزمان در حال حرکت هستند
ظرفیت این است که برای این Track ها حرکت کنند



sector یا block (بخش)

همین طور که این صفحه ها چرخند هر block های مختلف برای خواندن واحد انتقال disk هستند
sector یا block است یعنی اگر بخواهیم اطلاعات را جابجا کنیم یک block یا block ها را جابجا کنیم (واحد انتقال disk بیت نیست)

برای رسیدن به block مورد نظر باید حرکت بازو داریم ← انتقال: حرکت
چرخش: block مورد نظر به هد برسد

Track های هم شعاع یک استوانه هستند ← یک استوانه Cylinder

آدرس دهی اولین یک سکتور یا بلوک و جهت شروع جدول می باشد جدول می داند حرکت انتقالی را کم کند

disk مکانیکی است و حرکت بازو باید انجام شود ← برعکس خیلی کم می شود
جدول وظایف block ها با OS است

می خواهیم disk را این طوری ببینیم
به این صورت می شیم در صفحه زمان بندی دوتا block مختلف
هم فرقی نداشت اندازه disk اگر block هایش حجم یکسان 4 kb
بندی خیلی بهتر می شود اما اگر شیت ها هم بتوانند به طور حرکت همگانی انجام شود تا block بعدی
هم که این طوری است. یعنی اندازه زمان بندی محل block مشکلی دارد.

Subject:
Year. Month. Date. ( )

اگر blockهای یک فایل نیاز دارند پشت سر هم اجرا شوند و جاهای مختلف disk قرار گیرند خیلی سرعت را پایین می‌آورد، وقتی بخواهیم این نوع blockها را در disk پشت سر هم قرار دهیم گفته می‌شود Fragmentation.

مشکل disk اینست که حرکت مکانیکی دارد

از طرفی ترجیح ما این است که disk دائم دستکاری نشود بلکه اون حافظه است که همیشه با آن سروکار داریم.

مثال) زمان پیدا کردن (seek) ← حرکت هد (بر ساختار disk بستگی دارد)

" چرخش ← circular ← زمانی که طول می‌کشد تا به سر block برسد

" انتقال ← transfer ← زمانی که انتقال یک block بین disk و حافظه صرف می‌شود

زمان پیدا کردن (میانگین) = 10ms ← تعیین شده توسط disk است و محاسبه نمی‌کنیم rpm

چرخش = 10000 دور در دقیقه (یعنی هارد چند دور در دقیقه می‌زند)

10000/60 rps ← $\frac{60}{10000} = \frac{6}{1000} = 6ms$ (زمان یک دور)

میانگین چرخش نصف این است چون بعضی وقت‌ها بیشتر از نصف و بعضی از وقت‌ها کمتر از نصف

می‌چرخد میانگین آن می‌شود 3ms

زمان انتقال = بستگی دارد از زمان چرخش پشت سر هم $\frac{3ms}{n}$ ← n sector / track

هر sector یک block است 512 byte، 320 $\frac{sector}{track}$

2560 sector

فرض کنید فایلی به اندازه 1.3MB است برای خواندن کل زمان انتقال را تخمین بزنید

سوال ← همه‌ی بلوک‌ها | بین این دو حالت معمولی حالت اول است یعنی یک سری بلوک‌های پشت سر هم

random ← همه | یک سری random

1 sector های پشت سر

2 " تصادفی از کل دیسک

خواندن یک سیلندر (دور)

10ms

3ms

6ms

19ms

$\frac{2560}{320} = 8$ سیلندر

$19^{ms} + 9 * 7^{ms} = 82ms$

Subject:

Year. Month. Date. ( )

بدترین حالت اینست که همه Sector ها خش بزند

19ms ... زمان خواندن اولین شیار

زمان خواندن اولین sector صحیح (random)

10ms

3ms

6ms/320 ... زمان انتقال داده‌ها به سکتور

$(10ms + 3ms + \frac{6ms}{320}) \times 256 = 33.32s$ زمان خواندن یک فایل

همانطور که در حالت دوم دیدیم راست زاویه بیشتر نزدیک به حالت اول خواهیم بود

Subject:
Year. Month. Date. ( )

جلسه بیست و هشتم 87 ,10 ,7

زمان بندی دیسک : Disk Scheduling

چندین درخواست به OS برای دیسک رسیده اینها چه طوری جواب داده می شوند؟ به ترتیب جواب می دهید یا نه؟

مثال) درخواست های رسیده برای disk :

شماره شیارها: 55 58 39 18 90 160 150 38 184

راه ساده این است که این شیارها را به ترتیب آمدن جواب دهد

می توان راه حل بهتری پیدا کرد که این ترتیب را بهم زد و یک میانگین زمان را کم کرد.

راه اول) head روی شیار 100

درخواست برای شیار 55 ← Seek time

حرکت چرخشی برای همه یکسان است

FIFO

میزان حرکت تا به این شیار برسد

(45 + 3 + 19 + 21 + 72 + 70 + 10 + 112 + 146)/9 = 55.3

که به طور میانگین این مقدار حرکت کرده تا به شیار مورد نظر برسد

FIFO هیچ تلاشی برای کمتر کردن میانگین ندارد و دنبال این است که هر کس زودتر آمده زودتر جواب دهد

Shortest Seek Time First

راه دوم) این درخواست ها رسیده می توانیم ببینیم که کدام شیار به جایی که الان هستیم نزدیکتر است مثلا الان روی 100 هستیم و باید ببینیم کدام به ما نزدیکتر است (در این جا نیازی به تخمین نداریم

ترتیب انتخاب شیارها

SSTF روش 90 58 55 39 38 18 150 160 184

میزان حرکتی که می کند ← (10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24)/9 = 27.5

disk مکانیکی است و با حرکت سروکار دارد. در حد وسط این روش جهت حرکت هر چی می تواند باشد و جهت حرکت می تواند عوض شود. در وسیله مکانیکی اگر دائماً جهتش عوض شود دو تا مشکل پیش می آید

1. در تغییر جهت حرکت نیروی زیادی به آن وارد می شود و به تدریج از بین می رود

2. در هر دو طرف باید دست نیروی در پیدا کردن شیار داشته باشند

اشکال این روش: جهت حرکت مرتب می تواند تغییر کند

Subject:
Year. Month. Date. ( )

SCAN ← برای حل این مشکل راهی پیش می‌گیرند که جهت حرکت تغییر کند

اگر در یک جهت حرکت می‌کنیم، همان جهت حرکت کن تا آخر برو، بعد به جهت مخالف برگرد و بقیه را انجام بده

← اگر به بالای دیسک تا آخر رسید بعد برگرد

برگرد و این بار کمی پایین‌تر

بالای دیسک

Scan →	150	160	184	90	58	55	39	38	18
	50	10	24	94	32	3			19=27

در SSTF اگر فاصله‌ی نسبت باشد معمولاً حرکت Scan است ولی اگر این فاصله نباشد ممکن است

بعضی اوقات بیاید که جهت برعکس جهت باشد.

Scan → 150

105

100

90 → 58

* همانگونه که در این مثال مشاهده شد، 105 را اضافه نکردیم

این حرکت مثل انشعابات نیست، SSTF همیشه بهتر جواب می‌دهد.

اشکال این روش سرعت از دست رفتن می‌باشد.

جهت حرکت

با یک تغییر جهت این کار انجام می‌دهد

در جهت سمت راست دیسک حرکت کند، اگر نیاز باشد

که سمت چپ برود، این روش برای حل این مشکل گفته شد

دقت داشته باشید جهت حرکت را بدانیم

clock ← همان ساعت هم در یک جهت حرکت می‌کند

C-Scan ← این روش هزینه‌ی کمتری دارد

فرض می‌کنیم در جهت بالا رونده دقت دارد

C-Scan:	150	160	184	0	18	38	39	55	58	90

برگرد و دقتی ندارد و برمی‌گردد به ابتدا و باز هم این کار را می‌کند

(50 + 10 + 24 + 184 + 18 + 20 + 1 + 16 + 3 + 32)

عددی بیست می‌شود چون برای 184 تا 0 ، زیادی حساب کردیم که خیلی زیاد است و برای حل این

مسئله پرفرمنس خیلی زیاد است.

PqPCO

Subject:
Year. Month. Date. ( )

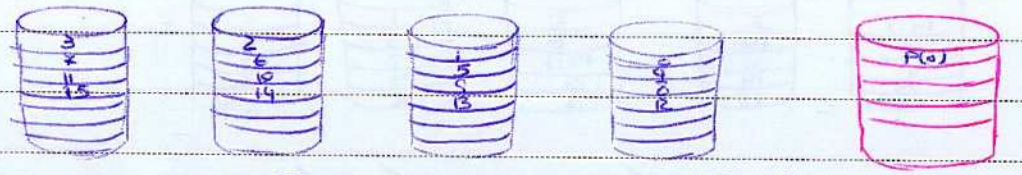
Redundant array of Inexpensive disk : RAID
Independant

سیستم ذخیره‌ساز disk ← اطلاعات حساس داریم

bottle neck
سرعت disk نسبت به حافظه اصلی پایین است (گلوگاه سیستم است)
بیشتر سرور server ها فقط برای ذخیره‌سازی هستند و جهت دسترسی به اطلاعات disk برخی از درخواست‌ها پاسخ نمی‌دهند.
مثل web server

هدف RAID این است که سرعت را افزایش دهد.

$P_0(a) = b_0 \oplus b_1 \oplus b_2 \oplus b_3$



block های داده‌ی disk ها توزیع کرد که چنانچه درخواست برای 5 آمد به جای آن تا اندازه‌ی ای‌ای آن را بدهد
اگر یکی disk نباشد

← اطلاعات داده‌ی چند disk بخش کرده تا بتواند به خیلی از درخواست‌ها به صورت همزمان پاسخ دهد.

مشکل ایجاد می‌کند:

فرض کنیم احتمال خرابی یک disk، P_0 باشد. هرکدام از این disk ها خراب شود کار سخت می‌شود
احتمال اینکه یکی از چهار disk خراب شود ← P

$P > P_0$ به راحتی محاسبات نشان داده.

قبلاً احتمال خرابی خیلی پایین داشتیم که الان این احتمال را افزایش داریم
برای حل این مسئله ← Redundancy در این‌ها داده‌ها را یک disk دیگر اضافه می‌کنیم که دیگر آن
parity را ذخیره می‌کنیم)

با اضافه کردن این دیسک هرکدام از دیسک‌ها از کار افتاد قابل بازیابی است

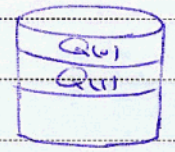
فرض کنیم b_1 از کار افتاده $b_1 = P_0(a) \oplus b_0 \oplus b_2 \oplus b_3$

اگر دو تا disk از این چهار تا از کار افتاده بود دیگر قابل بازیابی نیست و برای حل این مشکل اطلاعات بیشتر ذخیره کنیم

P4PCO

Subject:
Year. Month. Date. ( )

یک redundancy اضافه هم نگه می‌داریم



که از روی دیتاهای اصلی حساب می‌شود

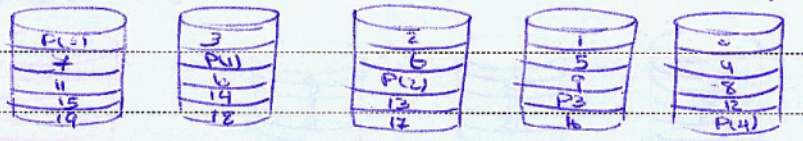
اگر بخواهیم قابلیت اطمینان سیستم را بالاتر ببریم اطلاعات اضافی‌تر ذخیره کنیم

اشکال دیسک‌های سابق چه بود؟ فقط می‌شد read یا write کرد و علاوه بر block باید parity

RAID 5

هم بنویسیم

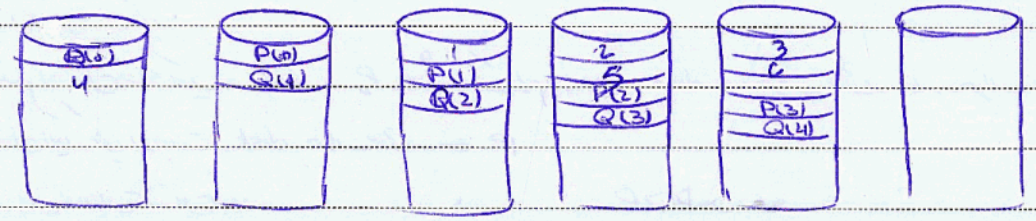
برای حل مشکل سرعت



مسئله سازی ساختن را در صدای ارور سیگنال‌های مختلف نگه‌داری می‌شود

RAID6 : همان Q و دخیل شدن بیشتر (برای Server های مهم‌تر)

هم P و هم Q را چپش می‌کنیم



برای Server های پردخواست زیادی مانند سیستم‌های Transaction

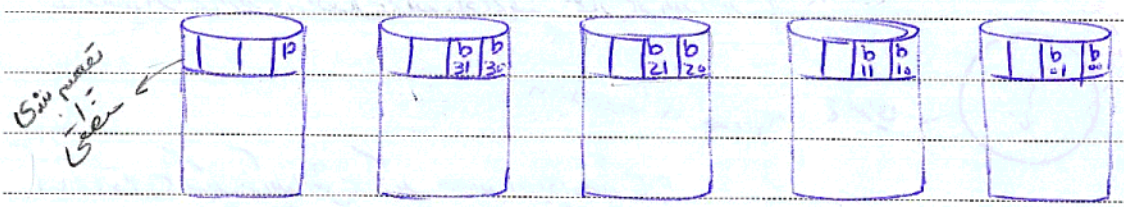
Raid را حتما برای سیستم‌های سنگین و پربرد توصیه می‌کنند و برای این کار

اضافات را باید حتما در نظر گرفت

Subject:
Year. Month. Date. ( )

یک درخواست را سریع تر اجرا کرده

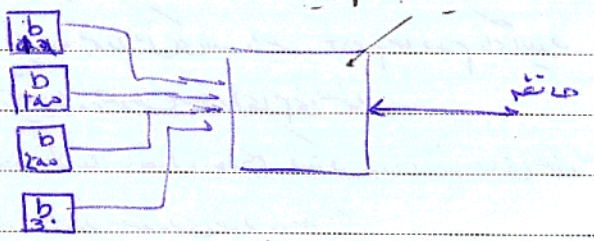
RAID 3 : سطح یک سیاهی نداره که بخوه بخش شدن اطلاعات فرق داره ← برای اجرای یک درخواست



block جزئی از دیسک است و حرکت block بین disk است

همیشه چون یک block را میخواهیم به صورت موازی اجرا می شود ما در هر چند disk خوانده می شود

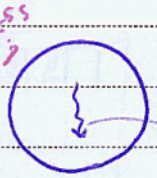
به صورت موازی اجرا می شود



برای کارهای کلی بری سیاک ← سرعت نیاز دو دیسک به disk کم

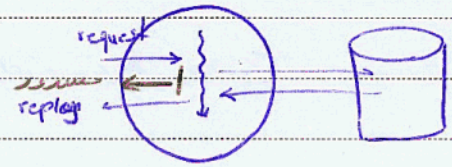
Subject:
Year. Month. Date. ( )

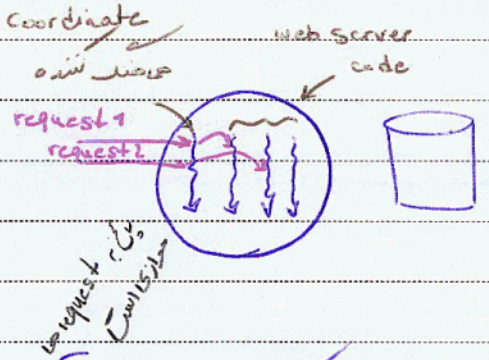
یک مسیر اجرایی کدی را اجرا می‌کند. همان نخ است



آیا می‌توان در یک پراسس بیش از یک مسیر اجرایی داشت؟

مثال: یک web server یک process است که درخواست‌های زیادی از آن می‌شود
یک request را نمی‌توان به پردازش کرد و بعد request دوم را گرفت
درخواست‌ها به ترتیب و یکی پس از دیگری پردازش می‌شوند
این web server اطلاعات خیلی زیادی را در قطعه‌های سکتور در disk است
درصدی که برای disk می‌رود صرف خواندن و نوشتن می‌شود و زمان
نسبت زیادی را باید صبر کند تا درخواست (اطلاعات) برگردد



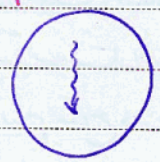


همان نخ اجرایی در نظر گرفتیم، disk همان یکی است
یک Thread باید بیکار شود تا بتواند درخواست جدید را
بپذیرد. این درخواست‌ها را این تیم‌کوچک می‌کند
هماهنگ است که ببیند کدام نخ آزاد است تا درخواست را
به آن بدهد
اجازه نمی‌دهیم یک کد را دنبال کنیم
وقتی request 2 می‌آید چون Thread آزاد داریم می‌توانیم از یکی از Thread های آزاد
می‌دهیم در اصل سرعت request
دستگیری: disk برای است که اینکه چندتا disk دارد و اگر یک request برای disk رفت بقیه نخ‌ها
می‌توانستند حتی تعلق برگه به disk دیگر داشته باشند

Subject:
Year. Month. Date. ( )

همچنین Web server کاری است که Thread برای آن بسیار مناسب است.

مثال: یک spread sheet را در نظر بگیرید



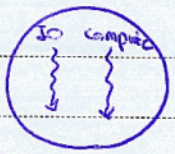
وظیفه: اطلاعات را از ورودی دریافت کند و به جداول مربوطه بفرستد یا update کند

1. input / output
2. محاسبه و کمک تصمیم سازی

الف) ساختار محاسبه بسیار ساده است اما حجم عملیات زیاد است

در حالت اول input کار دیگری ندارد و پس از اتمام آن منتظر input جدید می‌ماند تا محاسبات را انجام دهد

تمام بسته‌های داده جدید عملیات جدید را انجام می‌دهند



در حالت دوم محاسبات و عملیات جدید را می‌توانید در این دو عملیات جداگانه به صورت موازی انجام دهید.

موازی { input / output
computation

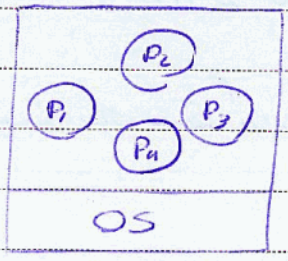
مشکل اول نیست برای اولین input صبر است عملیات محاسبه تمام شود

به همین دلیل سیستم‌های امروزی این امکان را دارند به صورت multiple thread کار کنند

برای شروع process نیاز به زمان دارد حجم و PCB را set می‌کنیم و pc را روی نقطه شروع قرار می‌دهیم (program counter)

تا برنامه را از آنجا اجرا کند پس در صورتی که Thread اجرا داریم

یک کپی بهتر شکل نمی‌دهیم.



در یک سیستم چندپردازشی می‌توانیم تعداد زیادی از این‌ها اجرا کنیم

و همگی تمام شوند

platform ← HW , OS

الف) بر این اساس ما process تولید کنیم

Subject:
Year. Month. Date. ( )

یک چیز خیلی

← در یک platform ی تراشیم چندین process تولید کنیم

← در یک process " " Thread " "

سکوردینیشن

برای تران کانت نسبت platform ، process مثل نسبت process ، Thread است. بله Thread است.

این جمله تقریباً درست است، دقیق نیست

process ها هرکدام منابع خودرا دارند و هیچ منبعی با هم اشتراک نیست پس هر پس منابع اختصاصی خود دارد ← به غیر منابع از هم پیروی هستند که بعضی منابع

اما یک سری منابع مشترک داریم که واقعاً مشترک نیست و هر پس یک سری از آن را خود دارد.

داخل PCB ← رجیسترها ، Stack ، reg برای کنترل حافظه ، PC ، حسابرسی IO

accounting

برای یک فرزند یک Thread داخلش هست جلوی تراشیم Thread اضافه کنیم ، Thread ها منابع process مشترک هستند

اینجا یک Thread چیه:

یک سرویس OS است، همان طور که یک process ی تراشت یک process دیگر تولید کند

اینجا یک system call است

منابع یک Thread:

هر process منابعی دارد که Thread ها تقریباً از آن استفاده کنند و در محدوده process حرکت کنند

یک زیرمجموعه‌ای از منابع فرزند را به کار می‌گیرد همان طور که process زیرمجموعه‌ای از منابع والدین را گرفت

یکی از ویژگی‌های مهم این است که ایجاد پایان آن خیلی کم هزینه‌تر از process است.

برای یک process ← اختصاص داریم منابع حافظه مجازی مستقل

هر وقتی والد process شیفت پیدا کرده

TLB در HW باید invalid شود (منابع مدیریت TLB)

بعد از اتمام فرزند باید منابع حافظه را آزاد شود