جزوه سیستم عامل

استاد : جناب دکتر پدرام

ab-rafiee.com publish
All copyright reserved©2013
http://www.ab-rafiee.com

برای ایجاد Thread ← نیاز به حافظه جدید ندارند و در حافظه مشترک هستند بنابراین کم هزینه است کم پیاده‌سازی و راه‌اندازی

که این جدای خود چون در process هم هست

TLB نیازی به خالی شدن ندارد چون فضای حافظه عوض نشده

← اختصاص تکه‌های کد به Thread های مختلف تقریباً random نیست

برنامه‌نویس می‌تواند Thread ها را مشخص کند که هم compiler، OS این ها را برای او در

← به نظر برنامه‌نویس کار می‌کند

← OS یک دستگاه سرویس است و به داخل فرایند کاری ندارد و فرایندها می‌توانند بدون آن هماهنگی کنند

← از طریق System call می‌توان به این ها دسترسی پیدا کرد.

← حافظه بین Thread ها مشترک است اما اگر یک Thread قسمتی از حافظه را تغییر دهد بقیه آن را می‌بینند

Thread برای کسان مختلفی نیست اما می‌تواند اطلاعات یکدیگر را خراب کند. کنه‌ی ارتباط Thread ها هم از

طریق فضای مشترک است. (کنه ارتباط process ها از طریق message است)

مزایای Thread

کم هزینه بودن Thread برای ایجاد و پایان

← مقداری منابع می‌بریم و کمتر منابع اصلی

حافظه مشترک ← ارتباط بهتر برقراری نسبت به message خیلی بیشتر است.

← هر چی یکی بخواهد بقیه هم می‌بینند

caching وجود caching، اطلاعات را به از disk می‌آوریم در حافظه نگه داریم (موقتاً) چون ممکن است

دوباره به آن نیاز شود. اگر من یک اطلاعات را به cache آوردم بقیه هم برای بیشتر می‌توانند از آن

استفاده کنند پس به دو بقیه‌ی خود پس می‌توانیم خیلی سریع به اطلاعات بگیریم

Subject:
Year. Month. Date. ( )

ص ①

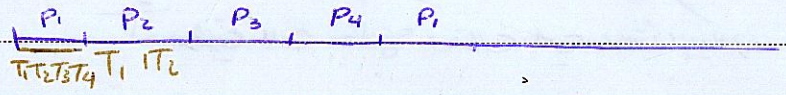
اختصاصی Thread :

PC
Stack ✓
register ✓ صاحبی reg های محیط خود را دارند
spread sheet ✓

بقیه مباحث با هم شریک هستند
(مثل ۴ کتاب Stalling ، Silberschatz)

زمان بندی :

$P_1 \dots P_4$

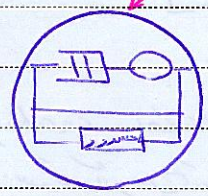


زمانی که P1 اختصاص داده می‌شود شامل Thread هم هست، این زمان باید بین Thread ها تقسیم شود اگر سه عدد باشد دیگر زمان کمتری می‌خواهند ولی اگر کاری برای انجام داشته باشند زمان بین آنها تقسیم می‌شود

OS در جدول زمان بندی انجام می‌دهد در سطح { process ، Thread } برخی از کارها برای انجام داشته باشند

OS این کار را انجام می‌دهد
یک صف باید در خود process مشترک گرفته
زمان بندی آن خیلی با process متفاوت نیست
OS باید بتواند در جدول زمان بندی انجام دهد

یک process



Subject:
Year. Month. Date. ( )

(point

← پردازنده چیزی نیست جز یک state machine تنها کاری که انجام می‌دهد اجرای مجموعه دستورالعمل تعریف شده است

دیگر نمی‌فهمد چرا و برای چی این دستورالعمل را اجرا می‌کند.

تمام کارهایی که بیشتر CPU انجام می‌دهد توسط سیستم عامل انجام می‌شود مثلاً توقف برنامه

ادامه * ← وقتی که یک برنامه پردازنده را در اختیار دارد چقدر است؟ می‌دانیم دستورالعمل برنامه دارد و برای محاسبات طولانی این

زمان خیلی طول می‌کشد مثلاً عملیات محاسباتی floating point spelling check

خیلی طولانی خیلی کوتاه

نرم‌افزار CPU را در اختیار دارد چه مشکلی دارد؟

مثلاً برای باز کردن یک webpage چقدر click کنیم اتفاق می‌افتد به این علت است که ممکن است

CPU در اختیار برنامه‌ای باشد که کار طولانی باشد.

پس احتمال دارد برنامه پس از زمان مشخص به کار نیفتد ← کاربردهای درسای (interactive) مشکل دارد

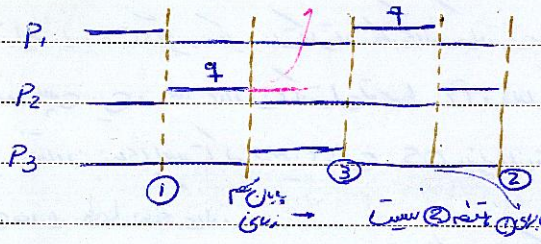
در کارهای غیر interactive مشکل ندارد چون منتظر جواب می‌باشد ناراحت کننده است.

تنها راه حل این مشکل Time sharing است؛ یعنی هر چقدر خواستی می‌ترای پردازنده را در اختیار بگیری باز زمان

مشخصی به تو می‌دهیم اگر کارت تمام نشد بعداً دوباره نوبت پردازنده را می‌دهیم (منظور از سیستم عامل

نمودار زمانی در رشته Time sharing

نقطه ③ قبل از نقطه ② است



کم زمانی q (در این مدت می‌ترای کار را

انجام می‌دهی)

اگر دوباره آماده شد قبل از اینکه به سراغ ادامه ② برویم به سراغ یکی دیگر می‌رویم.

در Time sharing می‌دانیم حداکثر برنامه‌ای با چه سرعتی به کار می‌افتد یعنی در این جای ترای تضمین کنیم

در اینجا حداکثر به اندازه 2q باید صبر کنیم تا نوبت به اجرای برنامه برسد. در این جای ترای تضمین کنیم

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

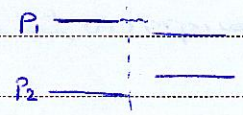
اگر n برنامه باشد.

PAPCO

Subject:
Year. Month. Date. ( )

بین اجرای برنامه‌ها در زمان عوض کردن پردازنده زمانی وجود دارد سیستم عامل باید اجرا شود تا پردازنده را از یکی بگیرد به دیگری بدهد پس بین این دو برنامه فاصله زمانی وجود دارد که چند ثانیه است بعدا صحبت می‌کنیم

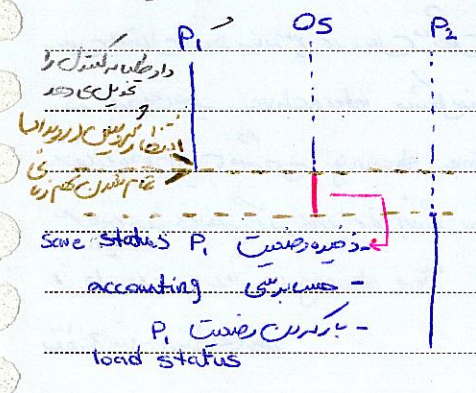
بین زمان قطع برنامه و شروع برنامه بعدی فاصله ای است



جلسه دوم 6/26

سیستم عامل باید بتواند همزمان برنامه را به صورت همزمان اجرا کند و کار عمل پردازنده متناسب عمل را در هر لحظه‌ای تواند اجرا کند و در کل نظم منظمی به نظر می‌رسد. چونکه عمل است یک برنامه پردازنده را درست طولانی در اختیار داشته باشد و کاربر به نتیجه نرسد در این جا قضیه مهم زمانی مطرح شد

یکی زمانی تمام شده باشد یا منتظر ورودی باشد



هر کدام از این اسامی یعنی

P_1 کارش تمام می‌شود و P_2 شروع به کار می‌کند در این جا برنامه دیگری است

باید کنترل را از P_1 بگیرد و به P_2 تحویل بدهد چون خودش نمی‌تواند

این کار را نداشت OS این کار را انجام می‌دهد. OS در تمام شدن P_1 کنترل را بدست می‌گیرد یک مدت کوتاه کار می‌کند و کنترل را به P_2 می‌دهد. پس بین از دست دادن کنترل توسط P_1 و دادن کنترل به P_2 یک اختلاف زمانی است که در این زمان OS پردازنده را در اختیار دارد و کنترل‌ها را switch می‌کند.

OS با بار کردن وضعیت P_2 کنترل را به P_2 واگذاری کند پس OS این کار را انجام می‌دهد چون P_1 و P_2 نمی‌فهمند کی این کار را انجام دهند

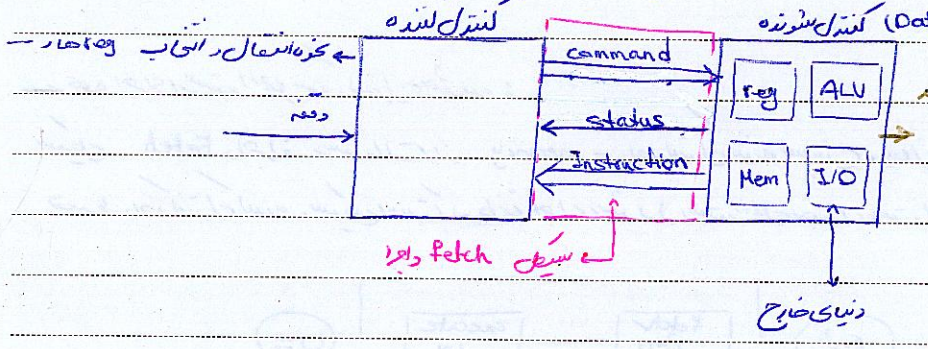
OS تا پردازنده را در اختیار نگیرد هیچ کاری از آن ساخته نیست پس OS همیشه تا آماده اجراست یا در حال اجراست

PAPCO

Subject:
Year. Month. Date. ( )

مروری به معماری کامپیوتر:

معماری سیستم (نحوه ارتباط) کامپیوتر از دو بخش تشکیل شده:



خود کنترل کننده باید بداند که چه کار کند ← دستورالعمل‌ها را می‌خواند و command های لازم را می‌فرستد،
دستورالعمل‌ها در memory هستند، کنترل کننده می‌خواند و می‌فهمد باید چه کار کند، command های مربوط به آن را می‌فرستد. علاوه بر دستورالعمل‌ها باید از وضعیت (status) مطلع باشد (carry، overflow، zero، ...)

وقفه ← می‌توانیم بگوییم کار وقفه اینست برنامه را قطع کند یک کاری انجام دهد و دوباره برنامه قبلی را اجرا کند

واحد کنترل کننده ← hard wire ← بهتر است.
microprogram (از بین رفته)
← به خاطر سرعت پایین از بین رفته

reg ها: ← رویت شونده (observable) ← هنگام برنامه‌نویسی می‌توان از آن‌ها استفاده کرد
Stack pointer، address reg، Data reg، ...
رویت نشونده (nonobservable) ← برنامه‌نویس از این‌ها می‌تواند استفاده کند برنامه‌نویسی که سیستم را کار می‌دهد می‌تواند از این‌ها استفاده کند ← کنترل‌های حافظه، وضعیت دیده نشدن، نشدن از دید برنامه‌نویس است
اگر سیستمی بنویسید مثل OS ... باشد می‌توان از reg های رویت نشونده کار کرد
reg های رویت نشونده ← سیستم عامل به این‌ها کاری دارد (وضعیت از خیلی کاری دارد تا جزء آن کار کند)

PAPCO

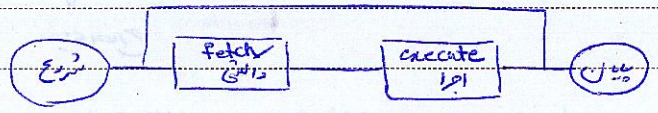
Subject:
Year. Month. Date. ( )

← چطور کلن است این reg ها برای OS دیده می‌شود ولی برای P_1 و P_2 قابل مشاهده نیست از کجا می‌فهمد کدام برنامه است که اجازه بدهد reg ها را ببیند یا نه؟

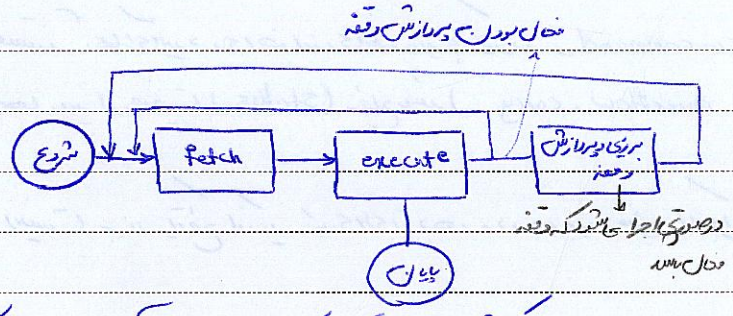
← نحوه اجرای دستورالعمل در کنترل شده:

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

سیکل دستورالعمل →



با فرض فعال بودن وقفه →



در پایان اجرای دستورالعمل یک برگشتی به وقفه می‌کنیم که ببینیم چیزی آمده یا نیامده. اگر نیامده بود به دستور بعدی می‌رویم

پردازش وقفه ← بعد از اجرای دستورالعمل در وسط سیکل fetch اجرا انجام نمی‌شود

← پردازش وقفه‌های خارجی در پایان اجرای دستورالعمل (وقفه‌ی داخلی هم داریم)

اگر وقفه‌ها نباشند امکان پیاده‌سازی کلم زمانی وجود ندارد

حافظه: عوامل ذخیره‌سازی

حافظه در سیستم کامپیوتر سلسله مراتبی است (hierarchical)

حافظه‌های cache در این جا صدق می‌کنند

← نزدیک‌ترین به پردازنده	register
حافظه پنهان	(Cache memory)
حافظه اصلی	(main memory)
حافظه پنهان disk	(disk cache)
ذخیره انبوه (disk)	(Mass storage)

دورتر از پردازنده ظرفیت بیشتر

افزایش سرعت

افزایش حاصل از پردازنده و ظرفیت

Subject:
Year. Month. Date. ( )

حافظه‌های سلسله مراتبی ← علاوه بر بالا بردن ظرفیت سرعت هم بالا برد و مقصودش اینست که ما بالا کار کنیم (در register ها) ← ما خیلی کمتر با حافظه‌های disk کار می‌کنیم بیشتر با reg ها کار می‌کنیم

← عوامل ذخیره‌سازی را چه کسی یا چی کنترل می‌کند؟

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

حافظه اصلی: سیستم عامل (ضمیمه اصلی تخصیص منابع (حافظه...))

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

حافظه پنهان cache: خودش خودش را کنترل می‌کند چون یک واحد کنترل کننده همراه خودش دارد و OS اصلاً از وجود چنین چیزی خبر ندارد. کسی نه cache را می‌بیند نه کنترل می‌کند. خودش در کارها دخالت می‌کند تا کارها را سرعت ببرد ← کنترلش تحت اختیاری است.

حافظه پنهان disk: کنترلش با خودش است و نگاه می‌کند کسی خواهد به disk دسترسی پیدا کند کار آن را با دخالت کردن سرعت می‌بخشد شاید هم نتواند

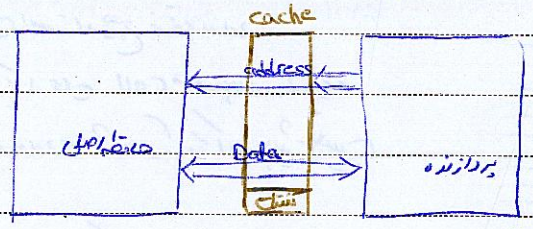
← OS فقط حافظه اصلی یا disk را می‌بیند از وجود حافظه‌های پنهان خبری ندارد.

جلسه سوم:

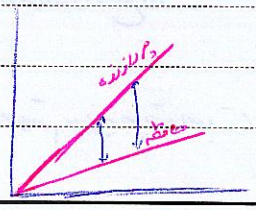
* سیستم عامل برای کار خود نیاز به دقت دارد (بدون وجود دقت نمی‌تواند کار خود را به درستی انجام دهد

حافظه پنهان یا cache:

به شرطی می‌توانیم سرعت بالا را در حافظه سلسله مراتبی بدست آوریم که هر چیزی خواهیم در حافظه پنهان اول پیدا کنیم



حافظه مکانیکی ← شانس دسترسی خازن ← سرعت کند



PAPCO

Subject:
Year. Month. Date. ( )

پردازنده در اصل آدرس‌ها را به حافظه اصلی می‌دهد که آدرس‌ها برای آن فرستاده می‌شود. cache در این وسط قرار گرفته و آدرس را نگاه می‌کند اگر این آدرس را خودش داشت 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 و ظرفیت کم است. چه چیزی در cache کپی می‌شود (temporal / spacial)

اصل محلیت (locality) هم از لحاظ زمانی و هم از لحاظ مکانی معنی می‌دهد.

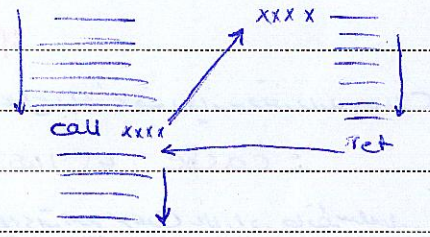
Temporal → اول آیتم‌هایی که اخیراً در حافظه دسترسی پیدا کرده احتمالاً باز هم دسترسی داشته باشیم (دوباره آدرس)

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

نقشش این است که لازم نیست کل 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 کار به نام کنترل همراه آن دست کنترل می‌کند. کلیه اعمال را این کنترل انجام می‌دهد. اگر آدرسی را نداشت Data را فچ می‌کند به حافظه می‌رود آن آدرس را برمی‌دارد و در صورت لزوم چیزی را از خودش حذف می‌کند و آن را جایگزین می‌کند. تمام این کارها پنهان از OS و CPU انجام می‌دهد.

مکانیزم وقفه: روال عادی یک ... call و return

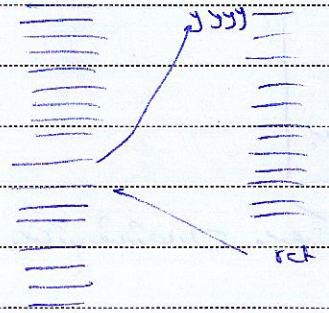


روال وقفه هم مثل همین است و هیچ فرقی ندارد

فرق call با وقفه اینست که محل call معلوم است ولی وقفه هر وقت دلش بخواهد وارد می‌شود و به آدرسی که از پیش تعیین شده است می‌رود.

هر وقت وقفه آمد می‌تواند برنامه را فچ کند

وقفه →



این مکانیزم وقفه برای سیستم عامل حیاتی است.

Subject:
Year. Month. Date. ( )

← کلاً OS بر اساس وقفه کار می‌کند (به شدت قوی است)

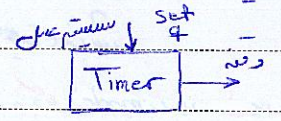
دسته‌بندی وقفه‌ها: (interrupt, exception)

1. اجرای برنامه: مثلاً برنامه در حال اجراست و عملی انجام می‌شود که نمی‌داند چه کار کند (تقسیم بر صفر، استفاده از دستور الحاق تعریف نشده، سرریز یا overflow، آدرس خارج از فضای تعریف شده ...) پردازنده باید چه کار کند برای اینجا باید چه کار کند؟ تنها دفاع پردازنده در این مواقع تولید وقفه است. و تولید وقفه باعث قطع برنامه می‌شود و به جای دیگری می‌رود (فقط تولید وقفه می‌کند و کار دیگری نمی‌کند این بدین وقفه است که کار را انجام می‌دهد) یا تصحیح می‌کند یا برنامه را قطع می‌کند (پردازنده هم چنین سخت‌افزاری برای انجام این کار ندارد و یک برنامه مثلاً این کارها را انجام می‌دهد) ← در حین اجرای دستور الحاق قطع می‌شود

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

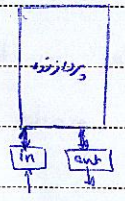
2. زمان‌سنج (Timer): هر برنامه یک هم‌زمانی دارند و یک Timer وجود دارد که وقتی زمان اجرای برنامه تمام شد تولید وقفه می‌کند و باز گشت کنترل هم حتماً به همان برنامه نیست

← هم‌زمانی ساعتی دارد (سخت‌افزار کوچکی که آن را روی زمان مشخصی set می‌کنیم باندازه هم‌زمانی (1ms) و شروع به شمارش می‌کند و هر وقت به زمان q رسید تولید وقفه می‌کند OS این را set می‌کند و قبل از این که کنترل را به برنامه دیگری بدهد این را set می‌کند.



در حینی که OS کنترل را به عهده می‌گیرد Timer set می‌شود.

3. ورودی/خروجی I/O: هر سیستم کامپیوتری یک سری port دارد ← I/O (mouse، شبکه)



ارتباط با دنیای خارج از طریق این port هاست که برای خود وقفه دارند. برنامه‌ای منتظر اطلاعات ورودی است از کی؟ منتظر که اطلاعات وارد شده توسط کنترل‌کننده برعهده OS است. ورود اطلاعات خودش باید خودش را معرفی کند. ورود اطلاعات یک وقفه می‌دهد و باعث قطع شدن برنامه در حال اجراست که این را در جایی از حافظه ذخیره می‌کند و اطلاعات ورودی را بعد در جایی از حافظه ذخیره کند و آن جا را برای برنامه منتظر است آماده می‌کند و وقفه‌های I/O پایان عملیات ورودی/خروجی را اعلام می‌کنند. پایان خروجی ← اطلاعات کاملاً خارج شده و دوباره می‌توانیم روی out بنویسیم

PAPCO

Subject:
Year. Month. Date. ( )

آخرین وقفه به سیستم عامل ربطی ندارد (زیاد) و این سه به OS مربوط بودند.

4. نقص سخت افزار (HW Fault) می‌تواند طوری ساخته شوند که درصدت خراب شدن تولید وقفه کند
ساده‌ترین راه برای حافظه: وجود parity که درصدت اشتباه تولید وقفه کند
قطع برق می‌تواند طوری طراحی شود که تولید وقفه کند تا خاموش شدن خازن‌های توان وقفه‌ای تولید کرده که اطلاعات مهم را ذخیره کرد. هر اشکال سخت افزاری می‌تواند این وقفه را تولید کند به شرطی که این طوری ساخته شده باشد.

← وقفه نوع اول در وسط اجرای دستورالعمل می‌تواند برنامه را قطع کند چونکه دستورالعمل غیری از است اما اصلاً می‌تواند آن را انجام دهد ولی سایر وقفه‌ها دستورالعمل تا آخر اجرا می‌شود
در نوع اول وقفه باید دوباره دستورالعمل قبلی که در حال اجرا بود اجرا شود ولی در سایر وقفه‌ها چون دستورالعمل کامل اجرا شده بعد قطع می‌شود به دستور بعدی در بازگشت می‌رود یعنی دستور بعدی کامل آمده وقفه را اجرا می‌کند.

← برنامه‌های کاربری نباید بتوانند به port‌ها کار کنند و باید از سیستم عامل بخواهد این کار را انجام دهد برای اینکه اگر هر کس بخواهد این کار بکند هرج و مرج می‌شود. OS این مسئله مدیریت است. چند برنامه به port کار کنند قبل از رسیدن اطلاعات مهم زمانی تمام می‌شود حالا این اطلاعات رسیده برای کی است؟ این کار باید دست OS باشد چون ترتیب‌ها را بدست می‌دارد هر اطلاعات دومی را به کی بدهد. این یک نصیحت است ولی می‌تواند جلوی برایلید پس باید کاری کنیم که نصیحت نباشد

مد دوگانه (Dual mode)

پردازنده دو mode اجرایی دارد: با اختیارات کمتر: پردازنده می‌تواند جلوی کارهای غیری از برایلید و تولید وقفه
با اختیارات زیاد: به هر چی دستری دارد دهد کاری بخواهد می‌تواند بکند
system mode : اختیارات زیاد دارد (دستری به port، set کردن Timer، ...)
user mode : اختیارات کمتر

Subject:
Year. Month. Date. ( )

جلسه چهارم:

با port های ورودی-خروجی از طریق دستورات ورودی-خروجی می‌توانیم کار کنیم. اداره port ها با OS است و هر برنامه‌ای که بخواهد با port ها کار کند باید از سیستم عامل بخواهد.

در سیستم می‌توان با port های reg هایی که دیده می‌شود کار کرد ولی در مد کاربر باید بتوانیم این کار را کنیم.

در مد کاربر عملیات غیرمجاز توسط وقفه تله می‌شود و به جای آن دسته دستورهایی در OS است.

برای طراحی چنین چیزی یک بیت اضافه به نام Mode bit ← 0 سیستم
1 کاربر

یک بیت به دستورالعمل (opcode) اضافه می‌کنیم تا Mode bit را نشان دهد

opcode	Mb

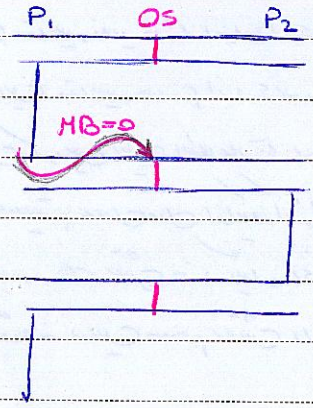
در اجرای دستورالعمل نگاه می‌کند اگر بیت مد با opcode جور درآمد دستورالعمل اجرا می‌شود در غیر این صورت تولید وقفه می‌کند

Mode bit چگونه تغییر می‌کند

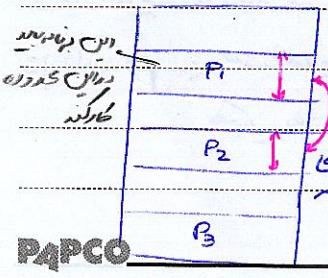
در شروع کار Mb باید صفر باشد (در reset - اختیارات سیستم و زیاد روی OS بکار می‌کنند چون شروع کار با OS است)

حالا اگر خواست کنترل را به برنامه‌ای بدهد قبل از اینکه به برنامه تحویل دهد برای یک کار عوض کردن Mode bit از دستورات OS است و وقتی که کنترل دوباره در دست OS خودش هستی می‌شود

برنامه اگر بخواهد Mode bit را عوض کند برنامه تله می‌شود.



حفاظت حافظه (memory protection)

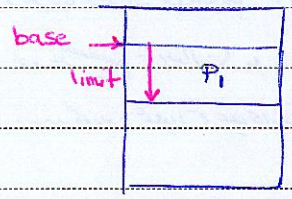


چند برنامه داریم که همزمان اجرا می‌شوند و این برنامه باید برای اجرا در جایی از حافظه باشد چون برای اجرا PC باید به کل برنامه اشاره کند که قطعی آن اندازه حافظه اشاره کند هر برنامه محیط خودش را دارد. هر کاری که می‌خواهد در محدوده خودش باشد نباید بتواند به محدوده دیگر

Subject:
Year. Month. Date. ( )

بستری باید کند چون باعث تغییر data می‌شود (نباید نصیحتی باشد، باید جلوی آن را بگیرند)

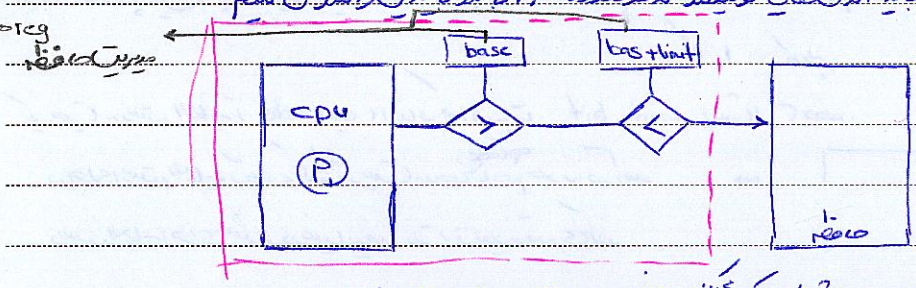
حافظه حافظه ← هر کس در محدوده خودش بتواند کار کند



هر برنامه یک آدرس base دارد که محل شروع برنامه و یک limit دارد که طول برنامه است. هر برنامه با این دو عدد در اختیار OS است (هنگام اجرای یک برنامه OS نیست ولی این توانایی را دارد که در غیابش و جودش احساس شود)

در این جا جلوی بازی‌گیری بدون اینکه حضور داشته باشد

هنگامی که P_1 در حال اجراست باید آدرس‌هایی تولید کند که در محدوده P_1 باشد، باید این را کنترل کنیم



قبل از اینکه آدرس را به حافظه بدهیم دست شرط را چک کنیم تا ببینیم در محدوده است یا نه:

1- باید حتماً بزرگتر از base باشد

2- باید کوچکتر از base+limit باشد تا در محدوده P_1 باشد

این مقایسه باید بخشی از زمان انجام شود ← مقایسه کننده‌ها داخل CPU است

اگر این مقایسه‌ها نرم‌افزاری بود این سیستم دیگر به درد نمی‌خورد. کار OS در این مقایسه base و limit را تامین کردن این اطلاعات را داخل reg (دیده نشونده) ذخیره

← سیستم عامل قبل از اینکه کنترل را به P_1 بدهد base و limit مربوط به P_1 را کار می‌کند و بعد کنترل را می‌دهد. OS نقش خود ایفا می‌کند

مدیر حافظه در این جا هم نقش دارد ← کار با این reg ها (چون دیده نشونده هستند)

دستورالعمل‌های مربوط به reg های حافظه از نوع ممتاز (privilage) هستند

privilage ← دستورالعمل‌هایی که فقط در سیستم قابل اجرا هستند و در مد کاربر اجرا نمی‌شوند و تغییر mode bit هم یک دستورالعمل privilage است

Subject:
Year. Month. Date. ( )

به طور کلی برنامه فقط دستورالعمل های تعریف شده اجرا کند نمی تواند آن را طوری طراحی کرد که در مقابل کارهایی که نباید انجام دهیم تولید وقفه کند پس از آن به سراغ وقفه می رود ...

یعنی جلوگیری آن در شرایط غیرعادی به صورت تولید وقفه است

← سیستم عامل در صورتی می تواند خوب کار کند که مدیری سیستم قوی وقفه باشد.

← در چند حالتی P کنترل را از دست داده به OS بدهد

دو تا احتمال ← خودش داوطلبانه این کار را انجام دهد یعنی یک چیزی از OS بخواهد (مثلاً دوری یا خواست چیزی از دیسک) که تا سیستم عامل آن را تامین کند طول می کشد ← درخواست از سیستم عامل

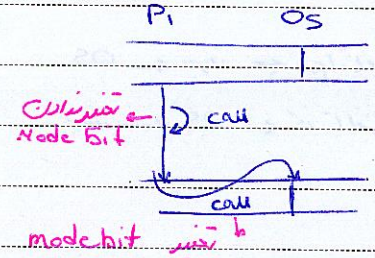
آن وقفه که عمداً وقفه مهم زیادی است ← وقفه

وقتی که درخواستی که یک برنامه از OS کرده آماده شد خبر می کند تا نوبت به برنامه برسد بعد درخواست را می دهد تا برنامه اجرا شود و به محض آمدن جواب درخواست کنترل به آن برنامه منتقل می شود

(point) برنامه های وقفه جزو OS هستند و در محدوده OS تعریف شده اند

← چگونه یک برنامه درخواست می کند؟ احتمالاً یک routin را call می کند

این call با بقیه call ها تفاوت دارد چرا که سایر call ها که درخود برنامه هستند Mode bit را عوض نمی کنند و call برای درخواست باید Mode bit را عوض کند.



بنابراین call ها system call می گویند

System call ← مجموعه ای از call ها که برای OS هستند و از پیش تعریف شده اند. (مجموعه ای از فراخوان های تعریف شده برای OS)

فرق بین call معمولی و system call باید طوری باشد که بخش افزار آن را متوجه شود system call. Mode bit را تغییر دهد.

PAPCO

Subject:
Year. Month. Date. ( )

مثلاً) Intel برای این کار یک دستورالعمل تعریف کرده که نام آنرا Interrupt گذاشته و آنرا با عدد صدا می‌کند

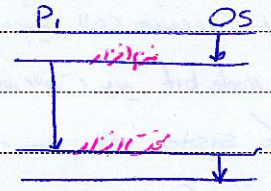
INT XX (58)

Intel یک جدول INT دارد به خانه 58 (عددی که جلوی INT است) می‌رود آدرس را برمی‌دارد
و ادامه برنامه را از آنجا اجرا می‌کند مثل همان call هست ولی اینجا call به آدرس غیرمستقیم است
(جدول وقفه‌ها که در اختیار سیستم عامل هستند)

← وقتی که این اجرا می‌شود به طور اتوماتیک mode bit صفر می‌شود ولی کنترل را به برنامه برنمی‌گرداند بلکه به OS می‌دهد.

← درخواست از OS ← system call ← تغییر mode bit بطور اتوماتیک (در پیاده‌سازی دستور این تغییر شده و نیاز به دستوری جداگانه برای تغییر mode bit نیست)

تغییر از حالت کاربر به سیستم فقط از طریق Interrupt (وقفه) انجام می‌شود (عجب جالبی شد :D)
در واقع دیتا احتمال اینکه P کنترل را از دست بدهد یکی کم‌هم (با وقفه)
تغییر mode bit برنامه از مد کاربر به سیستم از طریق سخت افزار است
(point) Interrupt باعث تغییر mode bit می‌شود
(کار اتوماتیک ← وجود سخت افزار برای انجام این کار)



OS ← برنامه ← نرم افزار
برنامه ← OS ← سخت افزار

Subject:
Year. Month. Date. ( )

جلسه پنجم:

← پردازنده‌ها فقط اجزای یک دستور را انجام تعریف شده انجام می‌دهد

← برنامه‌ای که بخواهد کاری انجام دهد باید پردازنده را در اختیار بگیرد و اگر در اختیار نداشته باشد کاری نمی‌تواند انجام دهد.

← بعضی از امکانات سیستم از دید OS مخفی هستند و تحت افزاری کنترل می‌شوند مثل حافظه نهان یا cache

اجزای سیستم عامل:

- مدیریت فرآیند — process management
- مدیریت حافظه — memory management ← اینجا منظور حافظه اصلی است
- مدیریت حافظه ثانوی — Secondary memory management ← منظور از حافظه ثانوی، دیسک است.
- مدیریت ورودی، خروجی — Input Output management
- مدیریت فایل — File management
- شبکه سازی — Networking کمک آزادی ارتباط بین سیستم‌های مستقل
- سیستم مفسر زبان‌ها — command Interpreter System ← چیزی که ما سر در در مورد OS می‌بینیم جزء از OS نیست بلکه دیگر OS است

مربوط به این درس

مدیریت فرآیند:

فرآیند یک برنامه در حال اجرا است و یک موجودیت فعال است. فرآیند برای اجرا به منابع احتیاج دارد

برای اجرا باید پردازنده را در اختیار بگیرد — برنامه باید در حافظه قرار گیرد تا اجرا شود — پردازنده — حافظه — صفحه‌های جنبی و monitor, print

active

program — process ← process یک چیز فعال است ولی program الزاماً در حال اجرا نیست ولی اگر به حالت اجرا درآمد یک process می‌شود. process یک موجود شروع می‌شود و یک چیز اجرایی می‌شود ولی program اینطور نیست و فعال است یکسال همین طوری بماند

مدیریت فرآیند شاید مهم‌ترین کاری است که OS انجام می‌دهد. وقتی OS یک process را آغاز می‌کند تا کارهای انجام آن با آن کار دارد.

مدیریت فرآیند بین کارها انتقال می‌شود.

1- ایجاد و حذف فرآیند کاربر ← فرآیند غیر مشخصی دارد

2- معلق کردن و دوباره آغاز کردن ← معلق کردن: موقتاً از دور خارج کنیم. دیگر در اختیارش نداریم ← به علت کمبود منابع

suspension — resumption

PAPCO

Subject:
Year. Month. Date. ( )

به دلیل اینکه تعداد فرآیند زیاد شده و منابع کافی ندارند یک سری را کنار می‌گذارد این فرآیندها نمرده‌اند بلکه نیمه جان هستند وقتی که منابع شرایطش حل شده و وضع آن بهتر شد دوباره فرآیندهای معلق را به حالت عادی برمی‌گرداند

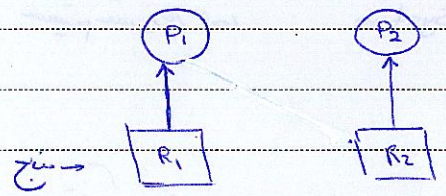
3- مکانیزم‌هایی برای هماهنگ‌سازی فرآیندها (synchronization): مثلاً اسکادرتا فرآیند باهم همکاری کنند و باید باهم هماهنگ باشند مثلاً باید اول فشار را اندازه بگیریم و شیر را باز کنیم می‌شود که اول شیر را باز کرد بعد فشار را اندازه گرفت پس یعنی باید باهم هماهنگ باشند

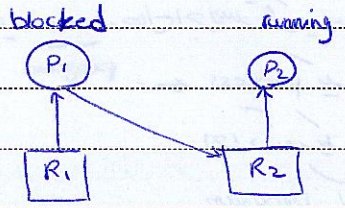
در این جزء مهم است یعنی اگر مثلاً ۲ ساعت از زمان اطلاعات گذشته اون اطلاعات دیگر به درد نمی‌خورد

4- مکانیزم‌هایی برای ارتباط فرآیندها: برنامه‌ای مثال اجرا چندین را محاسبه کرده این مقدار باید در برنامه‌ای دیگر مورد استفاده قرار گیرد پس باید ارتباط داشته باشند در هماهنگ‌سازی زمان مهم است ولی اینجا یک چیزی که محاسبه شده به برنامه‌ای که به آن احتیاج دارد می‌رسد حالا این برنامه هر وقت دلش خواست می‌تواند از آن استفاده کند در کامپیوتر هیچ دو برنامه به طور مستقیم باهم در ارتباط نیستند بلکه توسط OS باهم در ارتباط‌اند

5- روش‌هایی برای حل مسئله بن‌بست (Dead lock)

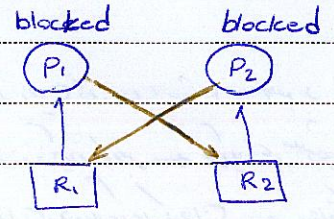
این دو برنامه این روش را در اختیار دارند به محض اولین چیزی که از P1 به R2 نیاز پیدا کند





در این حالت P1 نمی‌تواند حرکت کند چون R2 برای خواهد R2 در اختیار کس دیگری است پس باید منتظر شود تا R2 آزاد شود اما این است که وقتی P2 تمام شد R2 را آزاد کند

این برنامه‌ها از OS اجازه می‌گیرند این که چه کاری انجام می‌دهند OS بیچاره خبر ندارد و قفل‌های اند هر کس چند منابع احتیاج دارد



وقتی P2 به R1 احتیاج پیدا می‌کند نمی‌تواند از آن استفاده کند چون R1 در اختیار P1 است و چون در وسط اجرای P1 است R1 را رها نمی‌کند به حالت Dead lock می‌رسیم

Dead lock

Subject:
Year. Month. Date. ( )

این که OS بخواهد این حالت اختلاق افتاده کار راحتی نیست ولی اگر بخواهد ممکن است راه حلی داشته باشد تنها کسی که این بلوک می‌تواند راه حل ارائه دهد OS است چون خود P_1 و P_2 خراب هستند (غیرفعال) پس نمی‌توانند کاری انجام دهند

مدیریت حافظه:

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

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

کارهای مدیریت حافظه:

1- پیگیری قطعات استفاده شده و صاحب آن: یعنی ببیند کدام قسمت‌ها از حافظه استفاده شده و صاحب آن کی است (خود OS این کار را انجام می‌دهد)

2- تخصیص حافظه به پیگیری آن

3- پیگیری قطعات آزاد: باید بداند کجا خالی است تا اگر خواست به کسی حافظه بدهد کجا را بدهد

مدیریت حافظه ثانوی (Disk):

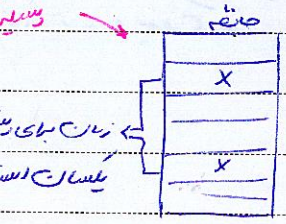
خیلی شبیه مدیریت حافظه است.

1- مدیریت فضای آزاد

2- تخصیص مکان ذخیره سازی

3- زمان بندی (یا نوبت بندی) دیسک: تقاضاهایی برای دیسک آمده باید به آنها نوبت دهیم تا بدانیم کدام را اول ... انجام دهیم. این را در این جا مطرح می‌کنیم حافظه با دیسک فرق دارد در تمام مکان‌های آن یکسان است (scheduling)

دیسک یک وسیله مکانیکی است چون حرکت توش است همین باعث می‌شود مکان‌ها با هم فرق کنند (مکان نزدیک به صفحه زمان دسترسی کم و برعکس و زمان دسترسی بستگی به موقعیت فیزیکی دارد)



Subject:
Year. Month. Date. ( )

هاشم: زمان دسترسی مکان حافظه یکسان است

دیسک: زمان دسترسی مکان ها متفاوت است.

برای اینکه دسترسی سریعتر را انجام دهیم از زمانبندی استفاده می‌کنیم مثلاً اول به فضای بزرگی می‌دهیم که مکان آن به هم نزدیک‌تر است

مدیریت ورودی خروجی:

یکی از کارهای مهم IO management مدیریت صف‌هاست.

1- مدیریت صف‌ها: برای اینکه بتواند سرویس‌ها را انجام دهد

2- کنترل ورودی‌ها و خروجی‌ها: مثلاً می‌خواهیم با port ورودی ارتباط برقرار کنیم قبل از این لازم است port را چک کرد. اگر بخواهیم خود با این port کار کنیم باید همه چیز را در مورد port بدانیم ولی از قبل کنترل‌ها برای آن‌ها را برای ما نوشتند

3- سیستم بافر: بافر چیزی شبیه حافظه cache است اگر بخواهیم با port ها کار کنیم با بافر کار می‌کنیم چون port ها کند است و وسط بین port و پردازنده هستند ولی بر خلاف cache ها کنترل کمتر افزایش ندارد پس کار اصلی بافر سرعت بخشیدن به کار است. اداره کنترل این‌ها به عهده OS است.

مدیریت فایل:

فایل: دید منطقی یکسان از اطلاعات را ارائه می‌دهد

اطلاعات می‌توانند جاهای مختلف ذخیره شوند (روی دیسک، CD، فلش مموری، ...) Data ها یکسان‌اند؟ نه، یکسان نیستند. اگر چیزی که نگاه کنیم این اطلاعات یکسان نیستند به لحاظ فیزیکی متفاوتند چون هر کدام از این‌ها چیزی را ضبط می‌کنند به برنامه اطلاعات چیزهای مختلفی را اطلاع می‌کنند فایل کاری می‌کند که به این تفاوت‌ها را نبینیم و این تفاوت‌ها را پنهان می‌کند

Subject:
Year. Month. Date. ( )

کارهای مدیریت فایل

ایجاد و حذف فایل

ایجاد و حذف راهنما (یا directory)

مکانیزم‌هایی برای کار کردن با فایل

جلسه ششم: 7.9

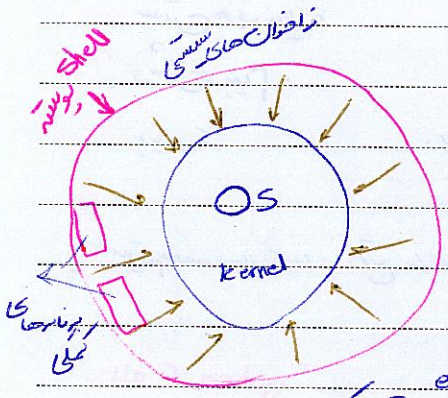
سیستم مفسر فرمان‌ها Command Interpreter System

واقعاً بخشی از OS نیست و چیزی است که همه با آن سروکار دارند

واسطه‌ای برای کاربر که به وسیله آن بتواند فرمان دهد

سیستم عامل مجموعه‌ای بزرگ از ابزاری است که فقط از طریق System Call

می‌توان با آن تماس گرفت یا برنامه‌ی را از آن تقاضا کرد.



برخی از راهنماهای را مستقیم صدا نمی‌زنیم

صرفاً سیستم عامل راهنماهای سیستمی است

exit() ← پایان یافتن یک فرآیند

به OS می‌گوید این فرآیند تمام شد تمام منابعش را آزاد کن

create() ← آغاز یک فرآیند (درست کردن یک فرآیند)

به منابع مورد نیاز فرآیند رایوشن می‌دهد

malloc()

سیستم مفسر فرمان‌ها:

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

نیست و برای انجام آن از تعداد زیادی System Call استفاده می‌کند مدرنترین copy یک برنامه است. مجموعه این

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

به کار خود ادامه می‌دهد.

Subject:
Year. Month. Date. ( )

نحوه‌ی انجام کپی:

copy F_1, F_2

- دریافت نام
- باز کردن فایل مبدی ← شاید هم چنین فایلی نباشد
openFile() ← یک سیستم کال است
- ایجاد فایل خروجی (اگر قبلاً F_2 نبود قبلاً وجودی داشته) ← در صورت وجود فایل خروجی
← به صورت یک loop اطلاعات را از F_1 به F_2 می‌بریم چون اطلاعات را نمی‌شود بر روی F_1 گذاشت چون از باز استفاده می‌کنیم و اندازه باز محدود است باید به صفحات به اندازه باز از مبدی بخوانیم
- حلقه: از فایل مبدی بخوان
در فایل خروجی بنویس
- بستن هر دو فایل
- نوشتن پیام
- پایان ← در این جا error ها را مستقیم نمی‌نویسیم و خطاها را می‌سپاریم و در صورت وجود خطا باید کاری بکنیم مثلاً یک پیغام بدهیم.
← سیستم مفسر زبان‌ها بدون توجه برای راحتتر کردن کار کاربر است و OS است

System Calls:

زبان‌های سیستمی به صورت اینترپریت‌های نرم‌افزاری هستند و این زبان‌ها باید به جیز دیگری compile شوند که همان Interrupt های نرم‌افزاری است. اگر قرار باشد این زبان‌ها compile شوند باید پارامترها را به کدی بدهیم

تبدیل به زبان ماشین

```
LD   R1, Parm
LD   R2, Parm
INT  28
```

← openFile(...)

← یک روش ارسال پارامترها استفاده از رجیستر هاست

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

این Interrupt های نرم‌افزاری می‌توانند به عنوان library routine باشند و توسط compiler از library بخوانند و آن را استفاده کنند

به خاطر این پشته از Interrupt های نرم‌افزاری System Call ها را پیاده‌سازی کردیم چون باید به call ها

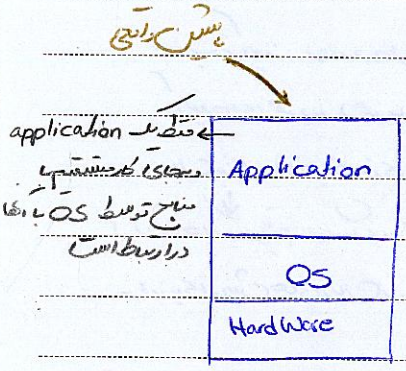
PAPCO

Subject:
Year. Month. Date. ( )

معمولی فرق داشته باشد چرا که باید mode را به سیستم ببرد.

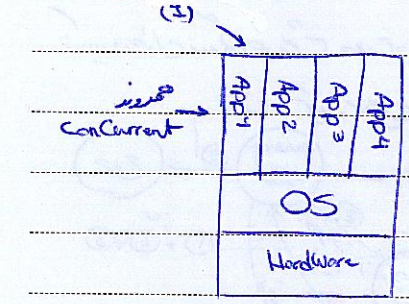
ماشین مجازی (virtual machine)

اول باید ببینیم ماشین واقعی چیست



← فقط یک application
منابع توسط OS با آنها
در ارتباط است

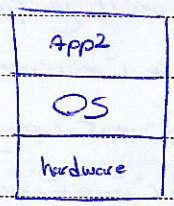
← از سرویس‌های OS استفاده می‌کنند تا بتوانند
از منابع استفاده کنند



همروند
Concurrent

← به جای یک application چند application به صورت همروند در حال اجرا است و هر لحظه قطعه‌ای کل منابع پردازنده را در اختیار دارد

اگر دید برنامه‌ها در ماشین مجازی نگاه کنیم فقط خودمان را می‌بینیم و متوجه نمی‌شویم که در منابع با دیگران شریک هستیم این به دلیل مجازی‌سازی است که OS انجام می‌دهد این همه این OS است



← این واقعی نیست بلکه واقعیت (I) است

ماشین مجازی

← شراکت منابع از دید برنامه‌ها مخفی است به جز OS
وقتی که وقفه‌ای در اجرای برنامه ایجاد می‌شود برنامه‌ها می‌فهمند به علت شراکت در منابع است
فقط این را به عنوان delay می‌بینند

Subject:
Year. Month. Date. ( )

مدیریت فرآیند Process Management

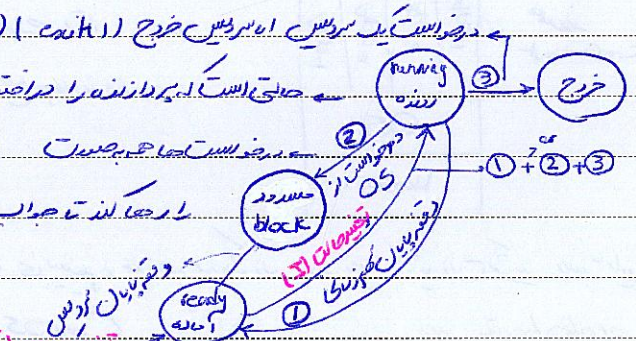
- ترتیب فرآیند: برنامه در حال اجرا
- فرآیندهای همروند (همزمان)

} قبلاً گفته شد

- زمان‌بندی (نوبت بندی) فرآیند: فرآیندها همزمان نمی‌توانند پردازنده را در اختیار بگیرند بلکه باید یکی یکی پردازنده را در اختیار بگیرند که در این جا الگوریتم نوبت آنها چگونه باید باشد.
(روش‌ها، الگوریتم‌های زمان‌بندی)
- انواع روش‌های زمان‌بندی: مزایا این الگوریتم‌ها چگونه است و هر یک کجا باید استفاده شوند.

حالت فرآیند Process State

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



- حالتی است که پردازنده را در اختیار دارد عملاً درست چند اتفاق نیفتد
- درخواست‌ها همه به صورت System Call است. وقتی که درخواست می‌کند باید پردازنده را رها کند تا جواب درخواست خود را بگیرد (هر نوع درخواست می‌تواند باشد)
- فرآیند جدید باید یک جوری وارد سیستم شود.

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

- وقتی در حالت مسدود است زمانی اتفاق می‌افتد که جواب درخواست آمده (سرویس مورد نظر ایجاد شده) نمی‌توان پردازنده را به این بدهیم چون دست یکی است پس باید در صف برود تا در وقت نوبتش شد پردازنده را در اختیار بگیرد پس به حالت آماده می‌رود تا در وقت پردازنده را در اختیار گرفت دوباره running شود. بقیه برنامه‌ها هم حق دارند ولی توان پردازنده را در وسط کار از ش بگیریم و به برنامه دیگری بدهیم

در تمام شدن: هم زمانی چون آماده‌ی اجرا دارد باید به حالت آماده برود چون آماده‌ی برای اجرا دارد و با نظری ادامه دادن را بحبس نداریم

← پایان سرویس‌ها توسط وقفه اعلام می‌شود

تغییر حالت‌ها ← بر اساس وقفه
بر اساس درخواست

← که این را باید تغییر mode می‌اندازد

Subject:
Year. Month. Date. ( )

دو تا تغییر حالت توضیح داده نشده ← تغییر حالت (I)

(II) " "

تغییر حالت (I)

زمانی که هر یک از تغییر حالت هایی که باعث شود برنامه ای که در running است این حالت را ترک کند (پردازنده را ترک کند)

این تغییر حالت اتفاق می افتد یعنی هر کدام یکی از تغییر حالت های 1 یا 2 یا 3 اتفاق بیفتد

تغییر حالت (II)

ایجاد فرآیند درخواست است و فرآیند جدیدی زمانی ایجاد می شود که فرآیند موجود درخواست ایجاد آن را بدهد. چرا که ایجاد فرآیند

یک System Call است

point) ایجاد فرآیند فقط بوسیله درخواست یک فرآیند دیگر است (توسط یک فرآیند دیگر)

وقتی به مثلا برای یک Icon کلیک می کنیم فرآیند ایجاد نمی کنیم و ایجاد فرآیند اصلا توسط کاربر انجام نمی شود.

point) ایجاد فرآیند فقط توسط فرآیند دیگر انجام می شود و راه دیگری نیست.

اگر create فقط توسط فرآیند انجام می شود پس فرآیند اولیه چطوری ایجاد می شود؟ } ابهامات

کاربر چی کاره است؟

هنوز درخواستی باعث می شود که برنامه ریز پردازنده را از دست بدهد حتی درخواست create و fork این تغییر حالت ها

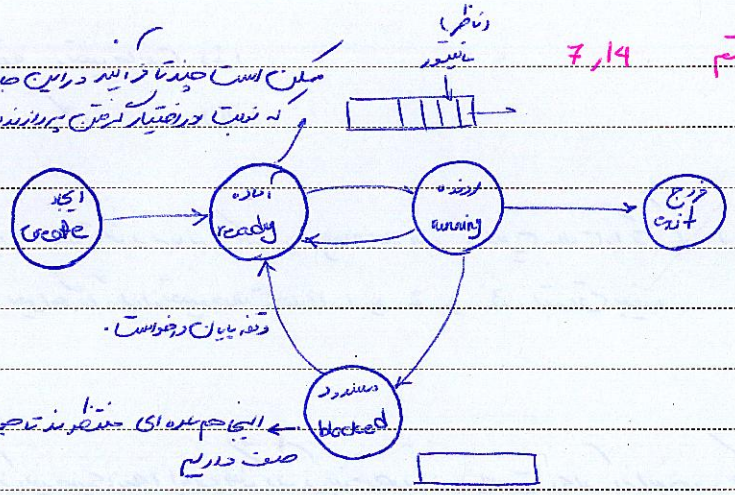
باعث از دست دادن پردازنده می شود.

PAPCO

Subject:
Year. Month. Date. ( )

جلسه هفتم 7,14

ممکن است چند تا فرآیند در این جا باشند پس صفی در این جا هست که نوبت در اختیار گرفتن پردازنده را تعیین می‌کند



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

تغییر حالت { وقفه / درخواست

بعضی از تغییر حالت‌ها ممکن است بر اساس هم وقفه و هم درخواست باشد

سیستم عامل توسط وقفه یا درخواست به کار می‌افتد

← ایجاد فرآیند فقط توسط فرآیند دیگر. تنها راه Active کردن یک برنامه فقط اینست که یک برنامه که در حالت running است این درخواست را بکند

به عنوان کاربری نمی‌توانیم برنامه‌های مختلف به فرآیند یا حالت اجرایی درآوریم. چه جوری؟

یک فرآیند داریم که دائماً در حال اجراست و فرآیند ناظر را به monitor

← monitor فرآیند مانیتور ورودی را چک می‌کند مثلاً وقتی یک کلید موس را می‌بیند فرآیند جدید مربوط به آن را ایجاد می‌کند کارش همین است و کار دیگری ندارد و فقط فرآیند ایجاد می‌کند ایجاد فرآیند را بر اساس ورودی‌هایی که برای ما می‌دهیم انجام می‌دهد چون فقط کارش ایجاد فرآیند است پس اکثر مواقع در حالت blocked است و کاری انجام نمی‌دهد.

برنامه monitor وقتی که ورودی آمد توسط OS بیدار کشیده می‌شود و پردازنده در اختیارش قرار داده می‌شود.

← فرآیند به صورت فیزیکی چی هست؟ برنامه در حال اجرا یعنی چی؟ OS چطوری آن را تشخیص می‌دهد؟

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

Subject:
Year. Month. Date. ( )

process control Block (PCB) (مجموعه‌ای است از داده‌ها). تا زمانی که ساختار داده‌ای وجود دارد فرآیند هم است. از دید OS، PCB نماینده فرآیند است. خود OS این PCB را ایجاد می‌کند و وقتی حذف شد، فرآیند هم حذف می‌شود و منابع آزاد می‌شوند.

داخل PCB چی هست؟

- حالت فرآیند Process state
- برنامه شمارنده program counter — برای اینکه بفهمیم اجرای برنامه چی هست و وقتی اجرای برنامه یهو گم می‌شد تمام شد باید بدانیم از کجا باید ادامه دهیم پس PC را در اینجا save می‌کنیم
- رجیسترها CPU registers
- اشاره‌گر پشته stack pointer
} برای اینکه اجرای برنامه بعد از اینکه دوباره CPU را در اختیار گرفت برنامه به درستی اجرا شود
- اطلاعات مدیریت حافظه Memory management information ← بیشتر مربوط به reg های دیده نشدن
- اطلاعات زمان‌بندی پردازنده CPU scheduling Info ← باید این را داشته باشیم تا بفهمیم چه کسی در صف قرار گیرد
  1. آیا اولویتی دارد
  2. درجه اولویت آن چی هست
  اگر هم باشد باید زودتر اجرا شود
- اطلاعات حسابرسی Accounting Info ← تا به حال چقدر از منابع استفاده کرده، چقدر زمان پردازنده، چقدر از disk استفاده کرده

هر برنامه‌ای که پردازنده را در اختیار می‌گیرد اول باید اطلاعاتش در پردازنده load شود تا برنامه به درستی اجرا شود و وقتی هم که می‌خواهد پردازنده را از برنامه بگیرد اول اطلاعات جدیدی را در PCB ذخیره می‌کند. پس PCB در حافظه زمان تغییر می‌کند.

وقتی که PCB از بین می‌رود همه اطلاعات مربوط به یک برنامه از بین می‌رود. پس دیگر برنامه‌ای وجود ندارد پس لزومی ندارد که منابع را جدا نگه داریم. آزاد کنیم چون دیگر کسی از آن منابع استفاده نمی‌کند پس از حذف PCB منابع آزاد شده حساب می‌شوند.

ایجاد فرآیند = ساخت یک PCB ← pointer

خروج از فرآیند = حذف PCB

Subject:
Year. Month. Date. ( )

point ایجاد فرآیند : یکی از متن منابع از OS

- یک ساختار داده‌ای برای هر فرآیند ایجاد می‌شود

← فرآیند اول چگونه ایجاد می‌شود؟

مجبوریم از حرف " فرآیند فقط توسط فرآیند ایجاد می‌شود " کوتاه 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 معلوم بودن (چون اکثر مواقع حالت معلوم صفر است می‌گوییم صفر می‌شود) به حالت معلوم برمی‌گردیم PC ، SP ، reg و ... به یک عدد خاص (حالت معلوم) می‌دهند

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 ولی اطلاعات لازم را تامین می‌کند همیشه به حالت معلوم خاصی می‌برند و اشکال ندارد در دو دفعه reset کردن دو تا مقدار متفاوت بدهد داخل انجا برود

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

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

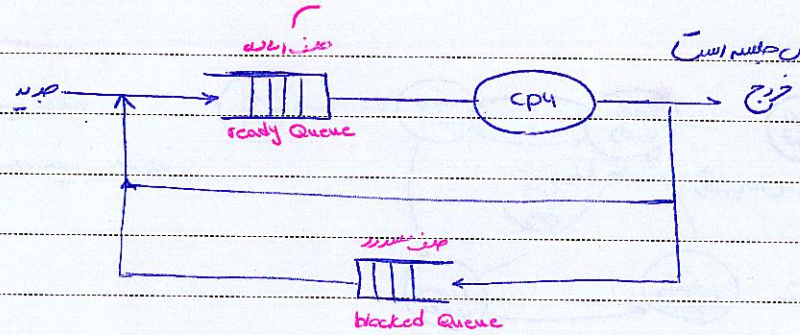
← PCB که اول به صورت دستی ساخته می‌شود همیشه یک ثابت انجا می‌دهد . OS را در جای خاصی از حافظه قرار می‌دهد و بنابراین همیشه آدرس شروع OS ثابت است

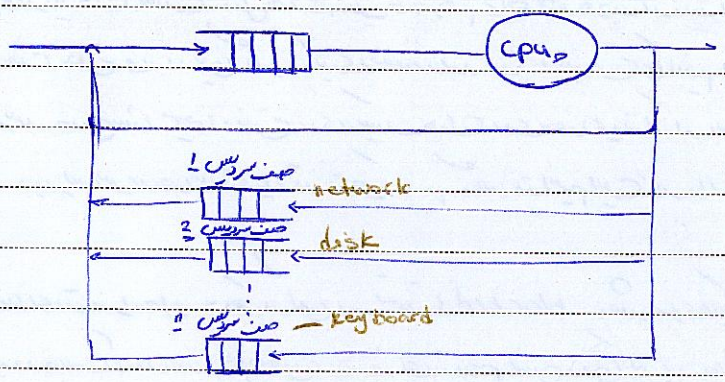
← این PCB همیشه ثابت است برخلاف PCB هایی که توسط OS ساخته می‌شود

← OS اطلاعات حیاتی خود را در جدول مربوطه دارد و در واقع PCB برای OS همین است و نیازی شدید به ساخت PCB ندارد . OS وقتی درخواست را می‌گیرد آن را تحلیل می‌کند و بعد پردازنده را از دست می‌دهد پس نیازی آن چنانی به PCB نداریم .

Subject:
Year. Month. Date. ( )

دیاگرام فرآیند: همان شکل اول جلسه است





چون سرویس ها با هم تفاوت دارند یعنی یکی منتظر keyboard، یکی منتظر port و ... است برای هر کدام از این سرویس ها یک صف داریم. کسانی که سرویس یکسان می خواهند باید با هم در یک صف باشند و معنی ندارد که کسانی که منتظر سرویس های مختلف هستند در یک صف باشند.

مدیریت کنترل تمام این صف ها با OS است. کسانی که در صف هستند اصلاً نمی دانند در صف هستند منتظر جواب درخواست خود هستند. چون پایان وقفه درخواست توسط وقفه است که پس هیچ گاه دو تا صف با هم جواب نمی دهند چون منتظر یک وقفه در یک لحظه رسیدگی می شود.

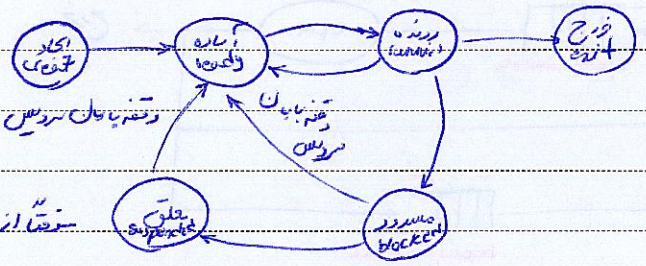
فرآیند شامل: کد، داده، پشته، PCB

وقتی می گوییم فرآیند در صف است پس باید همه اینها در این صف باید قرار گیرند که این صحت نیست در واقع PCB که نماینده فرآیند است در صف است که تازه این هم که چون PCB بزرگ است تمام PCB در صف نمی رود بلکه فقط PC در صف باشد کافی است.

Subject:
Year. Month. Date. ( )

جلسه هشتم:

تغییر شکل قبل برای اینکه فرآیندهای بیشتری بتوانند اجرا شوند



تقاضا برای فرآیند محدود است مثل حافظه. یک فایده هم می‌خواهد چند تا فرآیند که می‌خواهد اجرا کند چون منابع محدود است در حالت معلق می‌رود یعنی اگر فرآیندی مسدود بود موقتاً پیدا می‌کنیم (از حافظه) تا زمانی که اطلاعات که خواسته به آن برسد خارج شدن یعنی رفتن روی دیسک چون روی هوا می‌تواند باشد بلکه باید در جایی ذخیره شود تمام تغییر حالت‌ها باید بر اساس درخواست یا وقفه توصیه کنیم. اگر نتوانیم این کار را انجام دهیم هر چه می‌گذریم مقدم است.

این شکل تا زمانی می‌تواند راه‌حل دهد که کلی در قسمت blocked باشد برای اینکه فرآیندی در حالت انتظار نداریم و هم فرآیند در حالت آماده باشد و هم چین حافظه ای نداریم پس وقتی کلی در حالت blocked نیست می‌توانیم حافظه آن را بگیریم در نتیجه حافظه ای نداریم به فرآیند بدهیم.

← این راه‌حل برای مواردی است همواره فرآیندهای در حالت مسدود باشند

چه موقع از حالت مسدود به حالت معلق می‌رویم؟

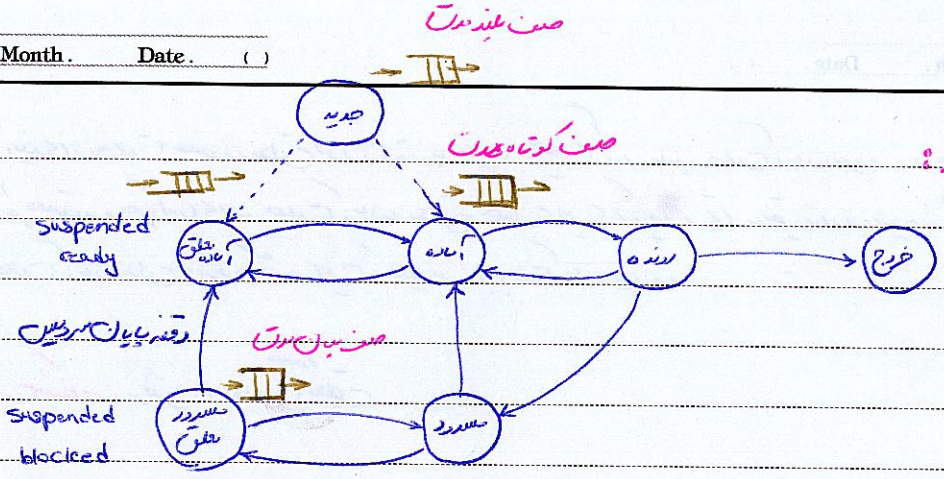
- درخواست جدید
- وقفه پایان سرویس

← این شکل و بالا فرآیندهای بیشتری را در آن واحد اجرا می‌کند ولی همیشه جواب نمی‌دهد.

PAPCO

Subject:
Year. Month. Date. ( )

شکل کامل تر:



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

آماده معلق ← اگر در حالت مسدود چیزی نداشتیم از حالت آماده استفاده می‌کنیم و از حالت آماده کسی را 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 حالا حالا ها نوبتشان می‌شود.

← هرچه فرآیند زیاد شود کیفیت سیستم خیلی کم می‌شود. زمان سیاب زیادی می‌گیرد تا یک فرآیند را از حافظه به دیسک بالکلس ببرد و حالا اگر تعداد حالات زیادی این کار را انجام دهد سیستم کندی می‌شود و دائما صدای دیسکی می‌آید جدا اینکه چقدر فرآیند می‌توانند اجرا شود تحمل بحثی است اینکه چقدر می‌توانند با کیفیت کم کار کنند. OS نباید برای این کار دیسک بگذارد اگر درست طراحی نشده باشد مثل OS های microsoft که درست طراحی نشده‌اند اگر تعداد فرآیندها زیاد شود تحمل نمی‌کند. crash می‌کند

← حالا اگر کسی را از حالت مسدود به حالت مسدود معلق می‌بریم امکان بازگشت به آن بدهیم و منتظر می‌شویم که حتما رفع پایان سرویس بیاید و اگر حافظه خالی شد آن را به حافظه می‌بریم. اگر رفع پایان سرویس آمد باید از حالت مسدود معلق به حالت آماده برود ولی چون در آماده معلق ممکن است چیزی باشد که جای برود اگر نبود بعد از آن به حالت آماده می‌رود. اگر کسی در حالت مسدود نباشد به سراغ حالت آماده می‌بریم از آن به حالت آماده معلق می‌فرستیم ولی امکان بازگشت به آن می‌دهیم

Subject:
Year. Month. Date. ( )

اگر فرآیندی در حالت مسدود معلق بود و وقتش پایان می‌رسید باید به حالت آماده برود و در حالت آماده هم کسی منتظر هستند رهم کسانی که در حالت آماده معلق چون این فرآیند زمان باقیمانده دارد باید منتظر بماند سپس دوباره به حالت آماده معلق می‌رود اگر در این جا کافی نبود به حالت آماده می‌رود.

← توجیه این شکل: رفتن از حالت مسدود معلق

دو مورد دیگر) صف‌های متعددی می‌تواند در شکل موجود باشد حتی در فرآیند جدید چون ممکن است چند فرآیند با هم create شوند پس درخواست‌ها در صف تراکنش قرار می‌گیرند به نسبت به آنها سیاست‌گذاری شود. چه سیاستی را برای این صف‌بندی پیش بگیریم؟ (الگوریتم صف‌بندی، زمان‌بندی، نوبت‌بندی، scheduling)

point) برای اینکه از حالت مسدود به مسدود معلق ببریم هم بهتر است صف باشد چون اگر نباشد یعنی اینکه random انتخاب کنیم ولی باید براساس برنامه‌ریزی باشد

صفی که در حالت آماده است خیلی مهم است تاثیری زیادی در کارایی سیستم و OS دارد که به آن صف کوتاه‌مدت می‌گویند (short term queue)

دو نکته: صف‌های صف مسدود معلق ← صف میان‌مدت mid term queue

" " " ایجاد ← بلندمدت long term queue

اسم‌گذاری این صف‌ها براساس مدت زمانی است که صف سرکشی می‌کنیم

short term queue ← در زمان کوتاه (ثانیه) به آنها سرکشی می‌کنیم یا فواصل زمانی کوتاه به آن سرکشی می‌کنیم

mid term queue ← در زمان‌های متوسطی به آن‌ها سرکشی می‌کنیم یا فواصل طولانی‌تر نسبت به کوتاه‌مدت

long term queue ← مدت فاصله زیاد به آنها سرکشی می‌کنیم

صف کوتاه‌مدت: فرآیند در حالت آماده، بیشتر از حجم زمانی‌ای که داریم پس چون حجم زمانی کوتاه است پس تند تند به آن سری می‌زنیم. اگر حجم زمانی باشد حداکثر زمان حجم زمانی طولانی کشد اگر حجم زمانی نباشد اجرا پیوسته است و خیلی تا زمانی است که یک درخواست برسد البته حالت جایی است که کوتاه نباشد ولی چون میان‌مدت کوتاه است.

PAPCO

Subject:
Year. Month. Date. ( )

برای دستگاه بیکار برای هدر رفتن پردازنده برای آن نمی‌ماند در صورتی که در صف کوتاه‌مدت پس از هر بار خالی شدن پردازنده برای آن می‌آوریم

چرا به صف ایجاد صف بلندمدت می‌آوریم؟

اگر واقعاً داخل سناریو دچار مشکل شویم قاعدتاً چیزی را برای کسانی که می‌خواهند وارد سیستم بشوند می‌گذاریم به آنها که وارد کار می‌کنند کار نداریم و فرآیندهایی که می‌خواهند create شوند را معطل می‌کنیم در واقع اصلاً صف ایجاد وجود ندارد اگر جا داشته باشیم یعنی در سناریو کمبودی نداشته باشیم صف ورودی نداریم ولی اگر صفی باشد باید بیشتر از همه تحمل کند.

سیاست‌ها: معمولاً برای صف‌ها ده سیاست‌های زمان‌بندی به کار برد و برای صف‌های بلندمدت و میان‌مدت از این سیاست‌ها استفاده می‌کنند چون زیادروی سیستم تاثیر ندارد و منطقی‌ترین سیاست صف‌بندی برای صف‌های بلندمدت و میان‌مدت از سیاست FIFO استفاده می‌کنند (منطق صف) چون مراجعه به دیسک کم است پس تاثیری زیادروی سیستم ندارند

میان‌مدت | FIFO
بلندمدت | FCFS — First Come First Served

برای اینکه پیاده‌سازی سیاست FIFO ساده است به خودمان برای طراحی صف پیچیده زحمت نمی‌دهیم و چون تاثیری روی کارایی سیستم ندارد از همین سیاست ساده استفاده می‌کنیم چون مراجعه صف کم است بهینه کردن آن فایده‌ای ندارد

چون که ما زیاد به صف‌های کوتاه‌مدت سر می‌زنیم پس بهینه کردن این صف روی کارایی سیستم تاثیر دارد پس روی بهینه کردن این صف تمرکز می‌کنیم

الگوریتم‌های زمان‌بندی پردازنده:

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

non preemptive: پردازنده را نمی‌توان گرفت بلکه باید خودش پردازنده را پس بدهد تمرین‌های درس الگوریتم‌های اند

preemptive: پردازنده را می‌گیرند یعنی قبضه می‌کنند

Subject:
Year. Month. Date. ( )

point ← در این ایده کم زمانی برای کاربر است چون که هر کاری که نتیجه کار خود را بگیرد روی بیشتر همه سیستم‌ها یک کاربر نشسته است مثل رایانه‌ها. اکثر سیستم‌های کامپیوتری کاری ندارد کم‌ترین رده سیستم‌های کامپیوتری embeded است ← پس در نتیجه ممکن است بسیاری از OS ها به گرده non-preemptive ی اعتقاد پس می‌توان گفت این الگوریتم‌ها کم استفاده است.

← OS های که در گرده non preemptive ها هستند می‌توان پردازنده را از آنها گرفت پس می‌توان از حالت دوم به حالت آماده رفت ولی OS های که در گرده preemptive هستند در آنجا نمی‌توان از حالت دوم به حالت آماده رفت

معیارهای ارزیابی روش‌های زمان‌بندی:

1- زمان انتظار (waiting time): از زمانی که فرآیند وارد می‌شود چه میزان در حالت انتظار بوده و پردازنده را در اختیار نداشته تا زمانی که خارج شود یکی از معیارها که همین زمان انتظار است.
کل زمان‌هایی که پردازنده را در اختیار نداشته

2- زمان گردش (Turnaround time): کل زمان running + انتظار ... را با هم جمع کن یعنی از زمانی که وارد می‌شود تا زمانی که خارج می‌شود چقدر طول کشید که کاهش باید باشد اگر 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 سرویس پردازنده. کسانی هستند که از زمان جلوتر باشند پس تا زمانی که پردازنده را در اختیار بگیریم سختی طول می‌کشد

point ← ممکن است مجبور باشیم بعضی دیگری تمام شود روی این اشکال ندارد باید ببینیم برای کاربر خاصی کدام از آنها مهم است

4- استفاده از پردازنده (processor utilization): کاری کنیم که پردازنده اکثر مواقع بیکار نباشد و کاری را انجام دهد مثل استفاده در سرور کامپیوتر استفاده از پردازنده در حالت چندتایی قبل است باید اولویت را ببینیم با کی هست.

P4PCO

Subject:
Year. Month. Date. ( )

B

5_ توان عملیاتی (Throughput) : اصولاً Throughput یعنی ما می‌دانیم: تعداد کارهای انجام شده در واحد زمان. اینکه چقدر انتظار کشیدی یا چقدر کاربری یا کارت چیست برایش مهم نیست، مهم اینست که تعداد کارهای بیشتری انجام دهد این زیاد برای سیستم‌های interactive مهم نیست.

← می‌توان گزینه‌های دیگری را هم گفت مثل: عدالت :D

جلسه نهم: 7,21

الگوریتم‌های زمان‌بندی پردازنده: (Process Scheduling)

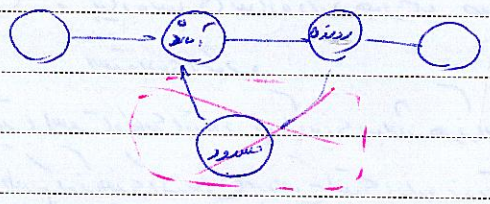
- بدون قبضه non-preemptive ← اینقدری رفت تا خودش پردازنده را از دست بدهد
- با قبضه preemptive ← پردازنده را ازش می‌گیریم

1_ FCFS ← روش‌های بدون قبضه

مثال)

فرایندهای	A	B	C	D	E
زمان ورود (تشریح وارد سیستم می‌شوند)	0	2	4	6	8
زمان سرویس (T_s) (زمانی که در سیستم بود)	3	6	4	5	2
زمان کل (T_T) (زمان پایان - زمان ورود)	3	7	9	12	12

روش ابتدایی که استفاده می‌کنیم FCFS است یعنی هرکه زودتر وارد شد زودتر سرویس می‌گیرد. (مثال خیلی ساده است یعنی یک باجه سرویس گرفتن کارت کنکور تمام می‌شود) یعنی بخش سرور خنثی می‌شود.



زمان پایان	3	9	13	18	20

A در زمان صفر وارد 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 که C وارد می‌شود باید صبر کند تا کار B تمام شود یعنی در 9 کارش شروع می‌شود با اینکه در 4 وارد شده است.

PAPCO

Subject:
Year. Month. Date. ( )

فاکتور دیگری ترین کالنیم (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 داریم این نسبت به ۱ نزدیک شود این نشان می‌دهد زمان انتظار در سیستم زیاد است اگر زمان انتظار کم باشد این عدد به یک نزدیک می‌شود اگر از این اعداد میانگین بگیریم هرچه یک نزدیک‌تر باشد بهتر است

← مشکل این روش (FCFS) با مثال زیر معلوم می‌شود

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

T_T میانگین $\rightarrow \frac{3+12+24}{3} = 13$

3 — 12 — 24

ترتیب‌ها را عوض می‌کنیم C→B→A

12 — 21 — 24

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

اولی بهتر است (ترتیب) چون زودتر سرویس می‌دهد و زودتر فرآیندها را پردازش می‌کند

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

انتظار ← تضمین وجود ندارد

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

این روش به ترتیب است و کاری ندارد چه فرآیندی باشد هر فرآیندی که اول آمده باشد زودتر سرویس می‌گیرد اگر بتوانیم کاری کنیم که فرآیندی که زمان سرویس کمتری دارد زودتر انجام شود زمان بهتری خواهیم داشت که این قابل اثبات است یعنی اگر به آنهایی که کوتاه‌تر است زودتر سرویس دهیم میانگین بهتری خواهیم داشت.

Subject:
Year. Month. Date. ( )

2- SPN (Shortest Process Next) (بعدی کوتاه‌تر) $1/T_s$

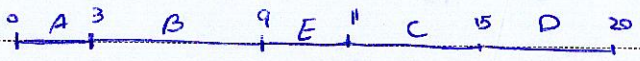
این روش بهینه است

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 پاسخ	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 فقط A است و رقابتی وجود ندارد در زمان 3 فقط B است پس باز هم رقابتی وجود ندارد در لحظه 9 و سه گزینه وجود دارد سیستم می‌شوند در اینجا باید کوتاه‌ترین را انتخاب کنیم و به همین ترتیب

در اینجا انتظار داریم میانگین‌ها از روش قبلی بهتر باشند. همان طور که انتظار داشتیم کمتر شد و به درستی رسیدیم که جواب بهتری از قبلی دارد.



زمان سرویس از کجا معلوم است؟ معلوم نیست (می‌پرسیم CPU از کجا می‌داند بلکه باید بپرسیم OS از کجا می‌داند)

اشکالات ← 1- از کجا زمان سرویس را قبل از اجرا شدن می‌دانیم پس زمان بهینه به راحتی به دست نمی‌آید و دشواری دارد ولی روش‌های برای تخمین وجود دارد که ممکن است از واقعیت دور باشد. زمان سرویس را می‌دانیم خودش هیز کنیم می‌توانیم حدس بزنیم (تخمین)!

2- حالتی پیش می‌آید که زمان سرویس طولانی دارد و از بدشانسی آن همیشه زمان سرویس کوتاه‌تر وجود دارد و همیشه این زمینه عقب می‌افتد و این امکان وجود دارد که نسبت به این زمینه اصلاً به این حالت برسیم یا starvation. احتمال غیرصفر برای این وجود دارد که هرگز سرویس به زمینه داده نشود.

Subject:
Year. Month. Date. ( )

point: اصولاً الگوریتم های اولویت داری می توانند به شکل بهتر عمل کنند

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

حل مشکل اولی:

روش هایی برای تخمین وجود دارد که با این حالت سربار سیستم زیاد می شود.

برای تخمین زمان سرویس

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

واقعیت اینست که یک فرآیند بارها سرویس می گیرد و یکبار سرویس گرفتن کارش تمام می شود. n می گوید چندین بار است که سرویس می گیرد. این تخمین را با استفاده از سابقه فرآیند می زنیم

$\alpha \rightarrow 0 < \alpha < 1$

$S \rightarrow$ تخمین

$T \rightarrow$ زمان واقعی

به این حالت می گوییم نمایی که می شود این رابطه داده

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

برای محاسبه از فرمول اصلی استفاده می کنیم برای اینکه بهتر بفهمیم چی می گوید آن را بسط می دهیم

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

توان با افزود اهمیت اعداد کوچکتر از یک کم می شود این بسط تمام تاریخچه فرآیند را دارد. رفتاری که اخیراً انجام داده اهمیت بیشتری دارد تا رفتارهای گذشته و می گوید رفتار الانش با رفتار گذشته نیست. α چگونه تنظیم شود الان حکم می شود.

هر چه α را به یک نزدیک کنیم ← رفتارهای اخیر تاثیرشان خیلی زیاد می شود. رفتارهای قدیمی ارزش نخواهند داشت یعنی قدیمی رفتارهای اخیر نگاه می کند

هر چه α را به صفر نزدیک کنیم ← کل تاریخچه را در نظر می گیرد. قدیمی ها را هم نگاه می کند

Subject:
Year. Month. Date. ( )

کسی که این کار می‌کند باید عملی فکر کند که به هیچ سیستمی نمی‌رسد پس بهتر است $\alpha = 1/2$

با این حال هم خیلی آسان می‌شود ولی محاسبات floating point لازم نیست و اگر این می‌خواهیم (floating point)

را انجام دهیم خیلی زمان می‌برد

$\alpha = 1/2$ ← همه‌ی تاریخ را در نظر می‌گیرد

محاسبات متناظر shift است و زمان برنیست

← مشکل اولی را توانستیم حل کنیم به اینکه خیلی دقیق نتوانستیم

راه حل دوم:

این مشکل را کاهش می‌توان کرد پس احتمالاً از این الگوریتم استفاده نمی‌کنیم بلکه این الگوریتم را اصلاح می‌کنیم

پس:

3- HRRN Higlest Response Ration Next

چون می‌خواهد با حالت گرسنگی مقابله کند جای به خوبی SPN نخواهد داشت و چون دارد چیزی را اصلاح می‌کند پس

باید هزینه‌ای بدهد

از یک نسبت صحبت می‌کند علاوه بر Ts، زمان انتظار را در نظر می‌گیرد و به انتظار امتیازی می‌دهد در SPN برش

اصلاً مهم نبود که چقدر انتظار کشیدی

امتیاز برای انتظار T_w / T_s

هرچه بیشتر انتظار بکشی امتیازت بالاتر می‌رود و هرچه زمان کوتاه‌تر باشد باز هم امتیازت بالاتر می‌رود

← برای مثال قبل

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

0 A 3 B 9 C 13 E 15 D 20

C 5/4 D 3/5 E 1/2 نسبت T_w/T_s

D 7/5 E 5/2 T_w/T_s

Subject:
Year. Month. Date. ( )

نباید بین همه‌ی پروسس‌ها به طور قبلی باشد چون می‌خواهیم گرسنگی را از بین ببریم پس باید هزینه‌ای بدهیم.
انتظار نداشتیم که از FCFS بهتر شود چونکه FCFS تقاضای راست دیگر آن از بین بهتر شود.

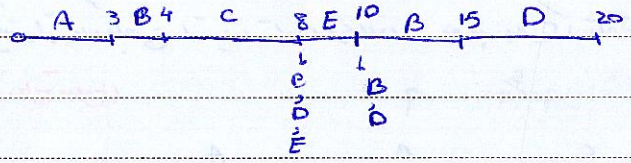
اگر کسی بخواهد از روش‌های non-preemptive استفاده کند انتخاب بهترینش و به درد بخورش FCFS است
چون SPN که مشکل گرسنگی دارد و HRRN هم دارای محاسبات زیاد و سربار بالایی باشد.

روش‌های preemptive :

4_ SRT Shortest Remaining Time : (پیش‌قبضه)
در هر لحظه می‌توان فرآیند را قطع 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 می‌توان برداشته را از فرآیند گرفت چون A کوتاه‌تر است
اگر انتخاب می‌کنیم در لحظه 3، B را داریم و در لحظه چهار هم B و C که B چون یک واحدش انجام شده 5 تا مانده و C
چون 7 تا برای C کمتر از 5 است پس
B قطع می‌شود و C اجرا می‌شود.



انتظار داریم از SPN هم بهتر شود
در هر لحظه SPN را اجرا می‌کرد و آن فرآیندی که زمان کمتر است را اجرا می‌کند

Subject:
Year. Month. Date. ( )

جلسه هفتم: 7,23

5- روش گردش زمانی (round robin) RR

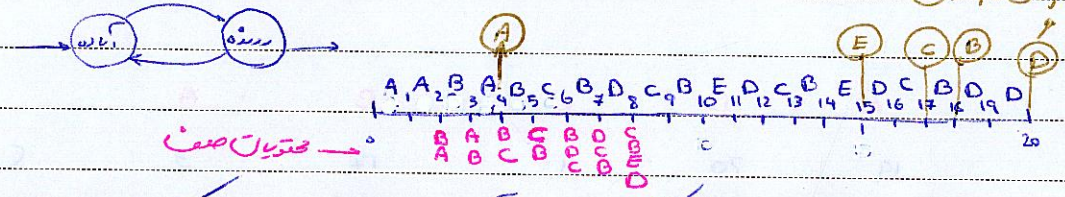
	E	D	C	B	A
زمان ورود	8	6	4	2	0
زمان سرویس	2	5	4	6	3

در دو حالت بررسی می‌کنیم ← q=1 ← زمان نسبتاً کوچک است

q= ← زمانی که از بعضی از اینها بزرگتر است

q=1

زمان تاخیر شدن



← محتویات صف

در زمان 0، نقطه A است و کم خود را می‌گیرد و به حالت آماده‌ی ورود می‌بیند در صف کسی نیست پس دوباره به حالت رسته‌ی ورود و یک واحد کار را انجام می‌دهد در لحظه 2، B هم آمده این قرار را می‌گذاریم که در این حالت B هر صف باشد و چون A هنوز الان سرویس گرفته پس B سرویس می‌گیرد (خاصیت هم زمانی عادلانه بودن آن است) اگر فرآیند وارد شد و در همان لحظه فرآیندی از حالت رسته به حالت آماده می‌خواهد برود اول فرآیند جدید در صف قرار می‌گیرد.

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

در مقایسه با اعداد بیش حیلی بهتر است ← طبیعی است چون زمان بیشتری را در سیستم صرف کرده و هنوز که هی باید به صف آماده برگردد زمان بیشتری می‌خواهد.

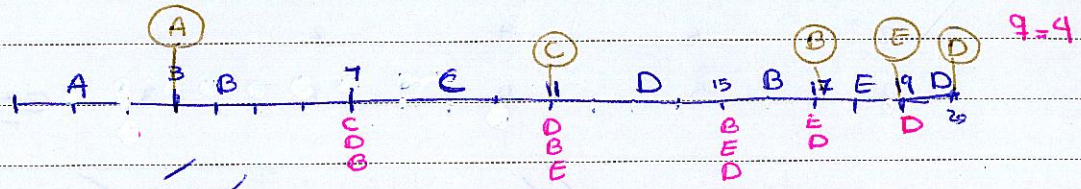
اما در کل زمان پاسخ این روش بهتر است ولی اولین بار که سرویس می‌دهد مثلاً در روش های قبلی هیچ وقت نمی‌توانست در زمان 5 به A نوبت بدهد. هم چنین در این روش کرسنلی وجود ندارد.

PAPCO

۲۰/

Subject:
Year. Month. Date. ( )

کارهای interactive به دنبال زمان پاسخ هستیم و همین که بزمان پاسخ این روش خیلی خوب است
این روش به بردی خود دو تفاوت عدد زمان پایان و زمان کل نباید با آدول بزند



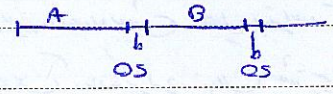
A زمان 4 را نیاز ندارد چون در 3 زمان کارش تمام می‌شود در این لحظه به صف نگاه می‌کنیم چون در صف منتظر B است 4 تای B را به آن می‌دهیم. در زمانی که B مشغول بوده D و C وارد شدند و 4 تای C را به آن می‌دهیم و C تمام می‌شود

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

اگر باز هم به زمان پاسخ نگاه می‌کنیم می‌بینیم در حالتی که q=1 است خیلی بهتر است و اگر q را بزرگ بگیریم همان FCFS می‌دهیم یعنی به کم زمانی اصلا توجه نکردیم

ادعا) اگر quantum کوچک شود شبیه SPN می‌شود ← اگر در SPN نزدیک کوتاه‌تر زودتر زمان می‌شود و در این جا هم همین طور است ولی دلیلی ندارد که همیشه یکسان شود چون از یک الگوریتم استفاده می‌کنند

ما نمی‌توانیم q را خیلی کوچک کنیم چون شکل این طوری نیست بلکه دقیقاً به شکل زیر است



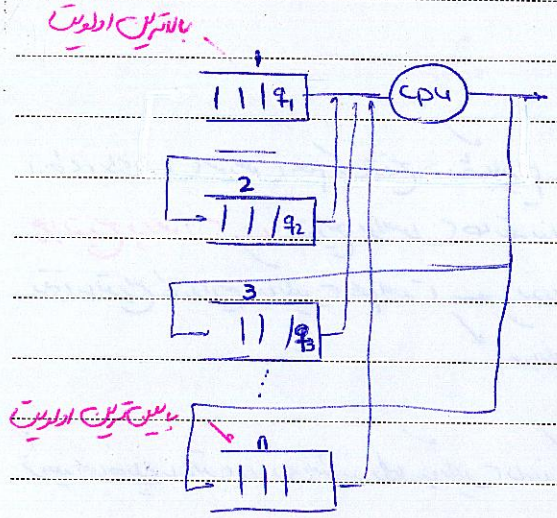
در این مثال از زمان‌هایی که OS کار می‌کند صرف‌نظر می‌کنیم
OS زمان بندی را ای‌ای می‌دهد نه CPU و نه خود زمان این کار را می‌کند اگرچه کوچک است ولی همیشه نمی‌توان از آن صرف‌نظر کرد اگر q خیلی کوچک باشد مجبور هستیم زمان OS را هم به حساب بیاوریم.

Subject:
Year. Month. Date. ( )

در هم زمانی OS قبل از اینکه پردازنده را به فرآیند بدهد Timer را صفر می‌کند تا کم زمانی به درستی اعمال شود.

(point) q باید طوری انتخاب شود که ۱۰ برابر زمانی باشد که OS نیاز دارد ولی اگر خیلی بزرگ باشد شبیه FCFS می‌شود که خوب است ولی هرچه کوچکتر باشد عملکرد بهتر است و شبیه SPN می‌شود ولی سیستم سربار دارد.

روش Feedback Queue : (پیشنهاد)



در این روش به جای یک صف از چندین صف استفاده می‌کنند که کمی متفاوت می‌شود دارد صف اول می‌شود یک q دارد که اگر کافی نباشد به صف دوم می‌رود که در صف دوم q می‌تواند همان قبلی باشد یا نباشد اولویت با صف اول است و تا زمانی که فرآیندی در صف اول است نوبت به صف دوم نمی‌رسد. احتمال اینکه یکی در صف اول نباشد زیاد است چون برگشت به صف اول نداریم اگر در صف دوم کارش انجام شد چه بهتر اگر انجام نشد به صف سوم می‌رود و می‌تواند q دیگری داشته باشد

صف آخر دیگر q نمی‌تواند q داشته باشد چون اگر q آن تمام شود کجا برود صف دیگری نداریم که بفرستیم باید صبر کند تا انجام فرآیند به طور کامل و کامل انجام شود

این روش دیگر الگوریتم SPN را اجرا می‌کند چون اولویت با کوتاه‌ترها است و اگر فرآیندی طولانی باشد خیلی است چه بسا حالت گرسنگی شود و بدون اینکه زمان سرویس را بداند دارد SPN را شبیه‌سازی می‌کند

هرچه صف بیشتر شود وظیفه OS بیشتر می‌شود

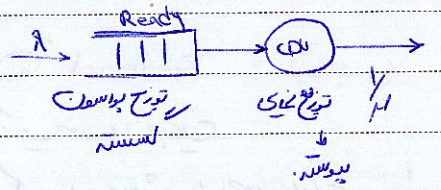
این صف جزو پارامترهای OS هستند و در کنار OS ذخیره می‌شوند

← تا حالا بهتر بودن روش‌ها را با مثال نشان دادیم ولی باید به طور کلی اثبات کنیم. برای اثبات از چندین روش استفاده می‌کنیم ولی الان ساده‌ترین نوع آن گفته می‌شود.

الان با روش آماری کار می‌کنیم ولی این بسیار پیچیده است و آسانش را انتخاب می‌کنیم

Subject:
Year. Month. Date. ( )

ند صف داشتیم و پردازنده‌ای که به آن خدمت می‌کرد. بسته به اینکه چه توزیع آمدی و چه سرعتی را برای این در نظر بگیریم
تحت رسان می‌شود. پس روشی که انتخاب می‌کنیم FCFS است. حالا ساده‌ترین توزیع برای آن را برای داریم



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

چرا توزیع پواسون ← توزیع پواسون حافظه ندارد و می‌داند قبلاً چه اتفاقی افتاده است. برایش مهم نیست
وقتی با توزیع کار داریم میانگین مهم است ← در واحد زمان چند تا فرآیند وارد صف می‌شوند (λ میانگین تعداد)
میانگین زمان سرویس: فرآیند چقدر است ($1/\mu$ میانگین زمان)

فرآیند خیلی نامرتب وارد می‌شوند ولی میانگین دارند. چون نامرتب بودن آنها را همان توزیع مشخص می‌کند.

رویدادی ورود فرآیند است که گسسته است (1, 2, 3, ...) ← وصف 2.5 تا فرآیند نداریم ولی 5 تا فرآیند داریم
زمان بین ورود فرآیندها چقدر است؟ عدد که ندارد بلکه پیوسته است. توزیع آن پیوسته است و تفاوت زمان همان
توزیع آن مشخص می‌کند

میانگین ورود

$$P(k) = (\lambda T)^k e^{-\lambda T} / k!$$

$k = 0, 1, \ldots$

← احتمال ورود k فرآیند در فاصله زمانی T

پیدا کردن میانگین این توزیع:

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

واریانس: $\sigma^2 = \lambda T$ ← چقدر از میانگین دور یا نزدیک هستیم

← کم ← خیلی به میانگین نزدیک، دور ← از میانگین دور هستیم

Subject:
Year. Month. Date. ( )

توزیع نمایی:

τ : زمان سرویس

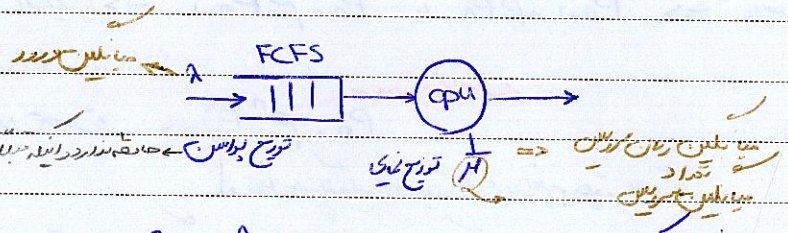
$f(\tau) = \mu e^{-\mu\tau}$

اگر برای رویدادها توزیع پواسون داشته باشیم زمان بین آنها 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,



حافظه ندارد و اینکه جلوی اتفاق افتاده هم نیست

$\rho = \frac{\lambda}{\mu}$ بهره‌وری ← تعداد بار زمان نسبت عکس دارد.

← توانایی سرویس دادن

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

در غیر اینصورت صف به سمت بی‌نهایت می‌رود حتی مسدود هم می‌تواند باشد.

به هر حال زمان‌های بیکاری خواهد داشت و همیشه ← $\rho < 1$

مشغول نیست ← نسبت ورود به نسبت توانایی سرویس

مشخصات آماری یک صف محدود

← ظرفیت سیستم دارد که می‌تواند تای نهایت ادامه پیدا کند

میانگین طول صف

- میانگین زمان انتظار

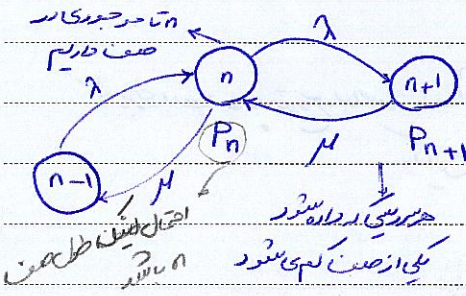
" " کل

- احتمال مسدود شدن صف

PAPCO

Subject:
Year. Month. Date. ( )

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



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

$\lambda P_n = \mu P_{n+1}$ ← تا اینجا درست است

$P_n = \rho P_{n-1}$ یا $P_{n+1} = \rho P_n$ => با استفاده از ترتیب ρ => خاصیت توزیع پواسن در این روابط برآورده می‌شود.

$P_n = \rho^n P_0$ => احتمال خالی بودن صف

عدد ثابتی است.

تا اینجا اندازه‌گیری نیست ولی دانیم چقدر است

گفتیم صف نامحدود است اما برای سادگی جسته‌ها حرفمان را پیش می‌گیریم پس صف را نامحدود در نظر می‌گیریم

$$\sum_{i=0}^{\infty} P_i = 1 \Rightarrow \sum_{i=0}^{\infty} \rho^i P_0 = P_0 \underbrace{\sum_{i=0}^{\infty} \rho^i}_{\text{تصاعد هندسی}} = \frac{P_0}{1-\rho} = 1 \Rightarrow \underline{P_0 = 1-\rho}$$

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

اینجا برای صف نامحدود بود حالا می‌آییم برای صف محدود (طولش را N می‌گیریم). هرجا که می‌خواست (∞) استفاده کردیم حالا محدودش می‌کنیم و N قرار می‌دهیم و مقادیر زیر بدست می‌آید.

$P_0 = (1-\rho) / (1-\rho^{N+1})$

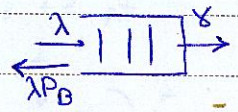
$P_n = (1-\rho)^n / (1-\rho^{N+1})$

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

Subject:
Year. Month. Date. ( )

$P_B = P_N$ ⇒ چون وقتی که صف پر شود بقیه فرآیندها block می‌شوند

وقتی که صف پر شود فرآیندهای جدید که وارد می‌شوند پذیرفته نخواهند شد.



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

* به خاطر توزیع های انتخاب شده تمام اینها از دقت درست است

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 می‌توان زمان انتظار پیدا می‌شود:

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

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

μ ← توان: پردازنده برای سرویس دادن است پس از این استفاده می‌کنیم و از λ که زمان ورود می‌کند استفاده می‌کنیم

تا زمانی فرآیند وارد صف می‌شود که صف پر نشده باشد و λP_B تا از آن انجام نخواهند شد

↳ احتمال block شدن

	زمان ورود	زمان سرویس
P_1	t_1	r_1
P_2	t_2	r_2
P_3	t_3	r_3

تمرین) از زمان بندی HRRN استفاده کنیم

کمترین منتظری باشد آن امتیاز بیشتری دارد $1+\frac{t_w}{t_s}$

از این رابطه در حالت کلی استفاده می‌کنیم ولی اینجا t_s را می‌گیریم

ترتیب اجرای فرآیندها:

$t_1 < t_2 < t_3$

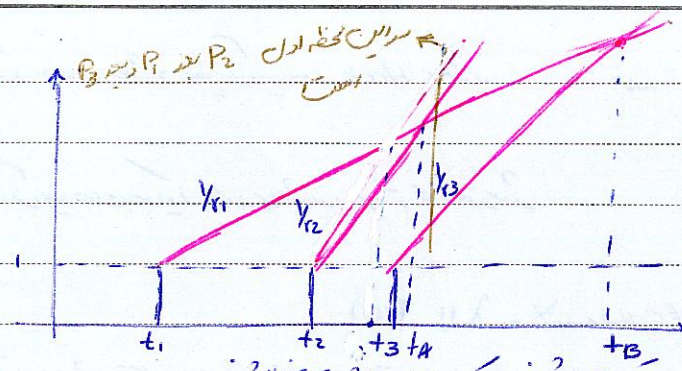
$r_2 < r_3 < r_1$

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

PAPCO

Subject:

Year. Month. Date. ( )



$t \le t_A$

$t_A \le t \le t_B$

$t \ge t_B$

زمان پایان سرویس فرآیند قبلی

توابع اولویت سه خط هستند چونکه به یک جمع شدند هیچ کدام از صفر شروع نمی‌شوند

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

در این زمان وارد شده

زمان انتظار است نقطه‌ای که شروع کرده منهای زمانی که وارد شده

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

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

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

$t \le t_A$ ⇝ $\overrightarrow{P_1, P_2, P_3}$

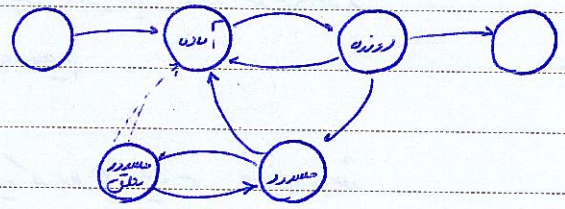
$t_A < t \le t_B$ ⇝ $\overrightarrow{P_2, P_1, P_3}$

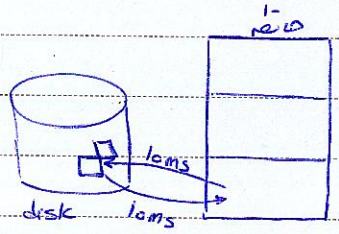
$t_B < t$ ⇝ $\overrightarrow{P_2, P_3, P_1}$ ⇝ وقتی که تمام شده باشند اولویت‌های بعدی بلند

وقتی تمام شده باشند این فرآیند که تمام شده از این ترتیب حذف می‌شود.

Subject:
Year. Month. Date. ( )

تمرین) مفهوم سازی swaping





حافظه این سیستم partition ثابت دارد بین دو حفره 3 تا فرآیند بیشتری نمی‌توانند در حافظه باشند و اگر بیشتر شد باید به حالت مسدود معلق بروند

برای وارد کردن فرآیند بعدی را از حافظه دیسک بیاوریم و فرآیند را از دیسک به حافظه بیاوریم.

زمان دسترسی به دیسک 4ms ← برای هر ایجاد اطلاعات است

زمان انتقال 6ms

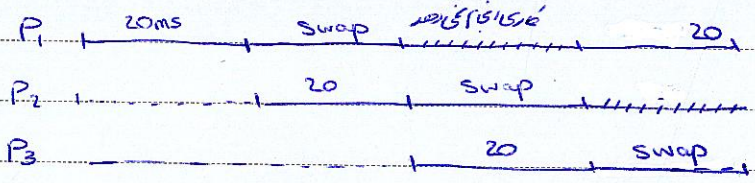
پس هر کدام از اینها برای انتقال به حافظه 10ms طول می‌کشد (یعنی جمع زمان دسترسی + زمان انتقال)

ه) برای بهره‌دهی از CPU یک فرآیند چقدر به پردازنده اجازه دهد؟

← پردازنده بیکار نباشد کاری برای انجام داشته باشد

برای انتقال هر فرآیند 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 فرآیند جدیدی آماده کار شود باید فرآیند در حال اجرا 20ms کار کند که در این عمل swap انجام شود که متعاقب عمل swap انجام شده فرآیند جدیدی 20ms انجام دهد.



پردازنده هیچ وقت بیکار نیست و همیشه چیزی برای اجرا است. swap ها نمی‌توانند هم انجام دهند چون فقط یک دسترسی به دیسک می‌توانیم داشته باشیم

PAPCO

Subject:

Year. Month. Date. ( )

b) اگر هر کاربر یک دوری دو ثانیه بعد صدا شد تعداد کاربران؟

یعنی زمان پاسخ باید ۲ ثانیه باشد چون اگر بیشتر باشد متوجه می‌شود

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

در هر ثانیه به 50 کاربر پاسخ می‌دهیم

c) شخص قبل را برای حالتی بکشید که زمان پردازش 10ms باشد

جلسه دوازدهم 7,30

تمرین) دو عملیات ادغام را اینگونه سازیم که هم زمان اجرا شود چه اتفاقی می‌افتد

که اینجا بالا در پایین tally را مشخص کن

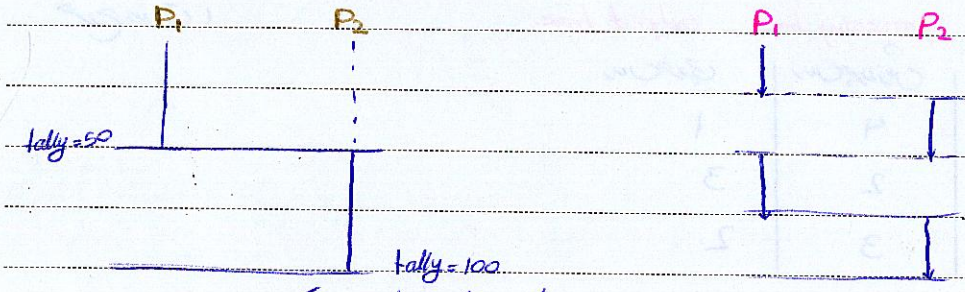
```
const int n=10;
int tally;
void total()
{
    int count;
    for (count=1; count<=n; count++)
    {
        tally++;
    }
}

void main()
{
    tally=0
    Parbegin(total(), total());
    write(tally);
}
```

P_1 R_2

دو تا فرایند همزمان اجرا کن

Subject:
Year. Month. Date. ( )

P_1 و P_2 چون به صورت همزمان هستند زمان بین آنها تقسیم می‌شود و ما هیچی نمی‌دانیم و هر ترکیبی از این‌ها می‌تواند باشد، ترکیب‌های مختلفی که می‌تواند وجود داشته باشد که ران بالا و پایین هیچیک کران بالا کاملاً مشخص است چون P_1 اجرا می‌شود و P_2 هم اجرا می‌شود ← 100



هر دفعه دیگری را بکشیم از این کمتر می‌شود

tally++ = tally+1;

برای اجرا باید به زبان سخت‌افزار تبدیل شود

```
Reg1 = tally;
Reg1 = Reg1+1;
tally = Reg1;
```

پیاده‌سازی

این را compile کردیم (خودش compile شد)

چیزی که اجرا می‌شود اینست

اگر کم زمانی را در نظر می‌گیریم روی چیزهایی که اجرا می‌شود تاثیر دارد.

اگر در حال از ایشان tally تقاضا کم زمانی در وسط کار یعنی بعد از اجرای Reg=Reg1+1 تمام شد، بخوردمان می‌گوییم چه اشکالی دارد، دفعه‌ی بعد که نوبتم شد می‌نویسیم ← ذهن خیال باطل چون نوبت را به کس دیگری دادیم مثلاً کس دیگر متوجه بود که 11 را ببیند روی جواب 10 را می‌بیند

Reg2 = tally 10

Reg2 = 11

tally = 11

پنج بار تکرار بشود ⇐ tally = 15

دوباره نوبت به P_1 می‌رسد ← tally = 11

به نظر می‌آید 50 جواب باشد برای کران پایین ولی جواب این نیست چونکه جواب 2 است، حالا چرا؟

کم زدن دقتی بی‌اجازه ما تفحص می‌شود خیلی مشکلات می‌تواند ایجاد کند و باعث این کارهای می‌شود.

Subject:
Year. Month. Date. ( )

کارهای دیگر دسته به آن می‌دهند
دسته‌ای

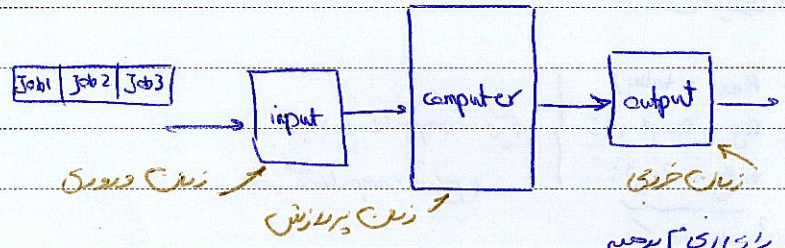
تمرین) راجع به یک مسئله خیلی قدیمی (در درس گفته نشده)

جدول زیر زمان های لازم برای ورود و پردازش 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)

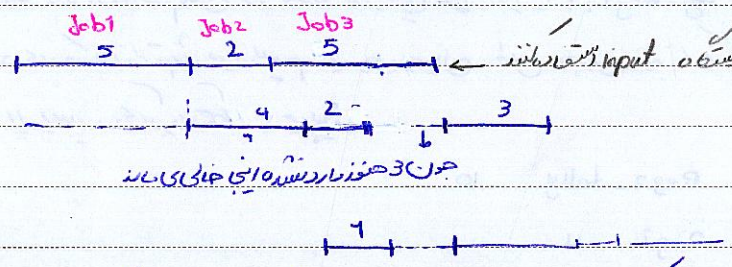
Input time processing time output time

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

سیستم batch



این سیستم‌ها قطعاً تعاملی نیستند
جدول ورودی را می‌گیرد و از آن در
خروجی را تحویل می‌دهند
می‌توانیم به این سیستم یک سری Job برای انجام بدهیم
که کدام از این‌ها زمان‌های لازم دارند که خوانده شوند ← زمان ورودی



پردازش باید وقتی شروع شود که ورودی به طور کامل خوانده شود یعنی پردازش Job1 بعد از 5 واحد زمانی شروع می‌شود ولی بعد از آن چون Job2 به طور کامل وارد نشده آن را پردازش نمی‌کنیم ولی چون بعد از آن Job3 هنوز کامل وارد نشده نمی‌توانیم آن را شروع کنیم پس سیستم تا مدتی بیکار می‌ماند و وقتی Job3 کامل آمد شروع به پردازش آن می‌کند.

PAPCO

Subject:
Year. Month. Date. ( )

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

P_1 — loop بی‌نهایت

```
while (TRUE) {
  print "A";
  print "B";
}
```

P_2

```
while (TRUE);
  print "C";
  print "D";
}
```

بدون هم‌زمانی

ACDCD

اگر خروجی را می‌خواست تحت پوشش هر ترکیبی امکان داشت

$(AB)^*(CD)^*$

می‌تواند جواب باشد چون هم‌زمانی در پایان هر loop تمام می‌شود

$A(CD)^*B$

یک A و یک print می‌کنیم هم‌زمانی ما تمام می‌شود ولی P_2 چندین بار تکرار می‌شود

BCAD ← قسمتی از خروجی

از یک نقطه‌ای شروع می‌کنیم B را print می‌کند هم‌زمانی تمام می‌شود C را print می‌کند و A را از دست می‌دهد

$C^*A^*B^*C^*$

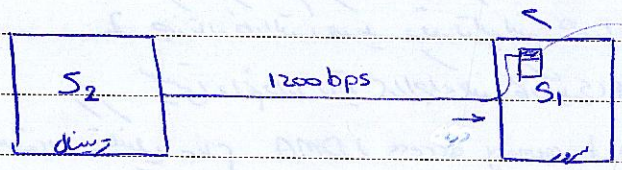
این امکان ندارد باشد

مثال دیگر) در مورد کنترل درسی‌ها و خروجی‌ها با وقفه

دو تا سیستم که با هم در ارتباط هستند

سیستم اصلی

با اصل وقفه در مکان حافظه ذخیره می‌کنیم



S_2 برای S_1 مثل ترمینال عمل می‌کند و پردازش‌ها را S_1 انجام می‌دهد

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

هر کاراکتر 10bit است

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

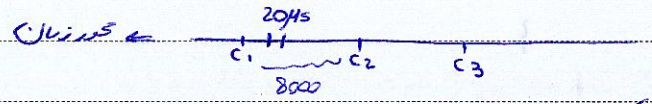
Subject:
Year. Month. Date. ( )

وقفه حاصل می‌شود ← باید داخل روتین وقفه برویم و آن را اجرا کنیم

روال وقفه: خواندن پورت و ذخیره در حافظه ← 20 μs

سوال اول: آیا این سیستم مشکلی پیدا می‌کند؟

فاصله ورود دو کاراکتر $8000\,\mu s = 8\,ms = \frac{1}{120}\,s$

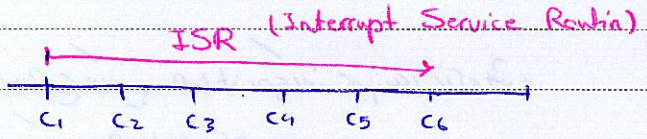


در 8000 می‌خواهیم، 20 تا برای این کار مصرف کنیم ← هیچ مشکلی پیش نمی‌آید.

حالت دوم:

در حالت دوم می‌بینید شبکه را ریع می‌کند به طوری که به ازای هر 4μs یک کاراکتر وارد می‌شود

1 char per 4μs



کاراکتر اول یک وقفه تولید می‌کند، برنامه هیچ فرقی نمی‌کند، C1 سرویس وقفه را راه می‌اندازد

برای این که به روتین وقفه برای C1 رسیدگی کند 9 تا char را از دست می‌دهد و همچنین تا ISR تمام می‌شود دوباره

یک char جدید وارد می‌شود و همیشه ISR در حال اجراست.

راه حل ها: 1- کندش کنیم و اجازه ندهیم هر 4μs بیاید (در صورتی که امکان پذیر باشد)

2- چند تا کاراکتر آمد یک وقفه تولید کنیم، همه را بچسبانیم به هم و این چنین

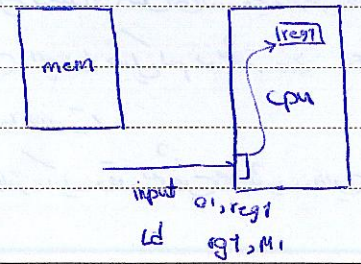
سیستمی دائم دارد روتین را اجرا می‌کند و کار دیگری نمی‌تواند بکند

راه حل اصلی: استفاده از تکنیکی به نام DMA (Direct memory access) ← دسترسی مستقیم به حافظه

سه دسترسی غیر مستقیم به حافظه داریم

دسترسی port به حافظه غیر مستقیم است

و توسط CPU است.



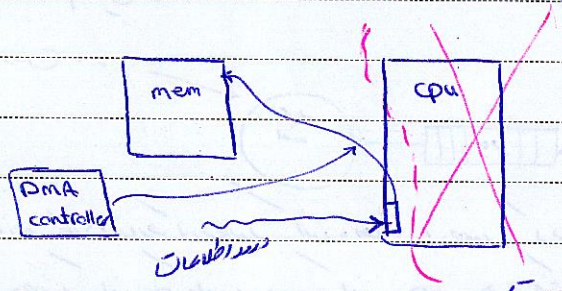
ld a1, reg1

ld reg1, M1

inc address → جدول می‌خواهیم روی همان خانه بنویسیم که

Subject:
Year. Month. Date. ( )

برای هر چیزی که وارد می‌شود باید چندین Instruction اجرا رهیم
جای DMA می‌گوید که واسطه را بردار و ارتباط را مستقیم برقرار کن
چون لازم نیست Instruction اجرا دهی و هر چی امر خودش می‌رود برجایش پس خیلی بهتر اجرا می‌شود ولی
کمتر از زمان‌هایی برای انجام این کاری خواهد



کنترل DMA controller در دست OS است
هر وقت load ، char آمد یک Interrupt به من بده این هاهم نمی‌شوند بلکه در حافظه ذخیره می‌شوند (عملا در این جا
حافظه‌ش buffer می‌گویند)
مثال: یک وقفه برای load ورودی

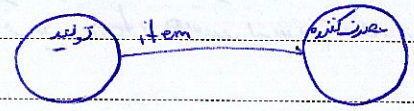
جلسه سیزدهم 7,8,1

هماهنگی فرایندها: process synchronizing

چند تا فرایند هستند دارای منابع مشترک پس باید به گونه‌ای هماهنگ شوند که مزاحم کار یکدیگر نشوند یعنی استفاده
مشترک از فرایندها به صورت حفاظت شده باشد.
این هماهنگی ← توسط خود فرایندها
← توسط OS

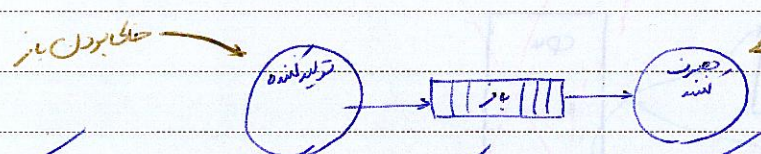
monitor

مثال از هماهنگی فرایندها
مثال تولیدکننده (producer) ، مصرف‌کننده (consumer)
دو فرایند که یک چیزی تولید می‌کند که فرایند
دیگر آن را مصرف می‌کند برای اینکه با هم
کار کنند باید با هم هماهنگ باشند
مصرف‌کننده اگر بخواهد 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 که نمی‌تواند مصرف کننده آن را مصرف کند.
اگر تولید کننده به طور مرتب تولید کند و مصرف کننده به طور مرتب مصرف کند
این مثال محدود کننده است و
تولید ← مصرف
" ← تولید
و البته این در به هم خیلی زیاد است

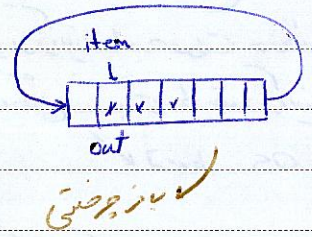


در این جا تولید کننده کاری ندارد که مصرف کننده item را مصرف کرده یا نه کار خود را انجام می‌دهد و تولید می‌کند چه مصرف کننده مصرف کند چه نکند و مصرف کننده هم به هر میزانی که بخواهد می‌تواند item ها را مصرف کند اگر خالی نباشد

ها حتی: برای تولید کننده: باز پر نباشد
" مصرف کننده: آزمون خالی بودن باز

← در این جا حتی از این رفته بلکه سست تر شده و نسبت به حالت قبل که پله پله به هم وابسته بودند کمتر شده

پیاده سازی باز



یک آرایه است و pointer داریم که نشان می‌دهد کجای 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 دست این دو پشت به هم هستند و می‌چرخند و خانه‌های از باز که خوانده شده آزاد می‌شود. این باز به صورت باز حلقوی می‌تواند تا ابد ادامه دهد

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

in = out : آزمون خالی بودن ، in=out است وقتی بافر خالی است

به نظر پر بودن هم همین است، اگر همین باشد فرقی پر خالی بودن چه فرقی دارند؟

فرقشون می‌زنیم ← پس می‌کنیم تا جایی که به قبل out برسیم و یک خانه مانده به out می‌گوییم پر است و برای این که یک خانه رو برای کجوییم پس داریم: آزمون پر بودن: in+1=out

↳ (in+1) mod n = out جون حالت چرخشی دارد باید محاسبه شود

اگر تعداد بافر زیاد باشد از دست دادن یک مکان (خانه) خیلی مهم نیست

PRODUCER:

while (True) { ⇝ حلقه‌ای که تا بی‌نهایت تولید می‌کند

produce an item in nexp ⇝ item را تولید کرده در متغیر nextp قرار بده

while ((in+1) mod n = out)); ⇝ تا زمانی که پر هست در این حلقه باش تا یک مکان خالی شود

buffer[in] = nextp;

in = (in+1) % n; ⇝ یک واحد pointer اضافه می‌کند

}

CONSUMER

while (True){

while (in == out); ⇝ تا زمانی که خالی هست در حلقه بچرخ

nextc = buffer[out];

out = (out+1) % n;

consume the item in nextc; ⇝ محتوی nextc را مصرف کن item

}

PAPCO

Subject:
Year. Month. Date. ( )

update کردن این هم به صورت نمی‌گیرد

متغیرهای موقت توسط OS تعریف کرد که بین فرایندها مشترک باشد در این حالت مشکل پیش می‌آید برای مشکل پیش می‌آید که دو فرایند با هم بخواهند متغیر را تغییر دهند

← می‌خواهیم راه حلی بدهیم که مکان از دست رفته نداشته باشیم؟

یک متغیر counter در این جا تعریف می‌کنیم که تعداد خانه‌های پر بافر را به ما می‌گوید

اگر counter=0 ← خالی

اگر counter=n ← پر

و قسمت های کد قبل 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 هر دو دستکاری می‌کنند و کافیست در ازایش و کاهش counter اشتباه شود ← پر و خالی بودن را عوض می‌کنیم

روی یک چیزی بوسیله کاما جدا کنید یک خانه خالی شده را جدید می‌خوانیم و یک چیزی را الکی می‌خوانیم

صحبت نشده پس یک item از بین می‌رود

← دستورات I و II باید به زبان سطح پایین اجرا شوند

```
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 هستیم و مقدار counter=6 است

counter = 6

reg1 = 6

reg = 7 ⇒ قبل از نوشتن هم رخ داده شده

⇒ الان باید counter=7 باشد

consumer counter = 6

reg2 = 6

reg2 = 5

counter = 5 ⇒ کاملا واضح است این غلط است

در حالت اصلی باید به 6 می‌رسیدیم

← در روش اول هر دو فرآیند فقط می‌خوانند و چون رشته‌ای می‌کنند به مشکل برنمی‌خوریم فقط ممکن است پس خالی بودن را زودتر یا دیرتر بفهمیم

← در این جا از یک متغیر مشترک integer (part pointers) استفاده کردیم در صورتی که هر جای تواند باشد پس استفاده از یک منبع مشترک بدون حفاظت

همه مشکل اصلی در این جا بود

فرآیندها باید در استفاده مشترک از منابع به هم اصلی کنند در مورد بافر هم اصلی داشتند ولی تنها منبع مشترک اگه اله با فر نبود

اگه حفاظت شده با بافر کار کنند قبل از دستری حتما تست می‌کنند

— یک مشکل کلی است برای همه منابع پس باید یک راه حل کلی باشد و فقط در مورد این تغییر نباشد

هر منبع مشترکی که استفاده می‌شود باید حتما یک controller باشد تا مدیریت به سراغ منابع مشترک می‌آوریم و 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 برایش سادگی است چون همه جا را می‌تواند ببیند

PAPCO

Subject:
Year. Month. Date. ( )

اینکه OS این کار را انجام دهد یا خودمان مزایا و معایبی دارد.

اگر زیاد به برای OS بروم کار می‌کند ولی سیستم را کند می‌کند و کارهایش را چون کلی انجام دهد برای یک کار کوچک کد زیادی را اجرا می‌کند چون باید همه حالت‌ها را مد نظر بگیرد.

مسئله را به صورت کلی بیان می‌کنیم:

مسئله ناحیه بحرانی (Critical Section problem)

ناحیه بحرانی: جایی که می‌خواهیم حفاظت شده باشد در مثال قبل ناحیه بحرانی موارد زیر است

counter = counter + 1 و counter = counter - 1

برای حل این یک حل برای ناحیه بحرانی ترین می‌کنیم

حل برای ناحیه بحرانی:

۱) تا فرآیند را در نظر می‌گیریم که می‌خواهیم با هم همانند کار کنند

۲) می‌توانند مختلف باشند ولی به منبع مشترکی نیاز دارند

```
void P(int i)
{
    while (True) {
        enter_critical (i);
        /* critical_section i */ ;
        exit critical (i);
        /* noncritical_section */ ;
    }
}
```

این جا چک می‌کند آیا می‌تواند وارد ناحیه بحرانی شود یا نه

کد اصلی در این جا هست که در این بخش به منبع مشترک دسترسی پیدا می‌کند

در این جا به منبع مشترک کاری ندارد

حفاظت منبع مشترک

بدیگران اطلاع می‌دهد من در ناحیه بحرانی نیستم

بهترین حالت است که ۱) تا فرآیند از یک منبع مشترک دائماً در حال استفاده هستند

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

راه‌حلی که برای همه موارد کار کند وجود دارد ولی خیلی کند است و همه چیز را در نظر می‌گیرد و در صورتی که خیلی دقت‌ها لازم نیست را انجام این راه‌حل به هزینه‌اش نمی‌ارزد.

Subject:
Year. Month. Date. ( )

چون این مسئله راه حل ساده‌ای ندارد برای دو فرآیند آن را حل می‌کنیم و سپس آن را به 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 ناحیه بحرانی را برای دو فرآیند بررسی می‌کنیم

الگوریتم های دو فرآیندی برای مسئله ناحیه بحرانی

← برای درست بودن راه حل باید سه شرط برقرار باشد

شرط های راه حل صحیح:

1. اگر فرآیندی وارد ناحیه بحرانی شود دیگر فرآیندی نمی‌تواند وارد این ناحیه شود ← انحصار متقابل mutual exclusion

2. انتظار محدود bounded wait ← برای وارد شدن به ناحیه بحرانی باید تضمین شود که میزان انتظار فرآیند محدود است

← این دو تا شرط برای صحیح بودن راه حل کافی است اما یک شرط دیگر هم بعداً می‌گوییم

تلاش اول (first attemp)

```
P0 → while (True) {
        while (turn == 0);
        critical_section0();
        turn = 1;
        noncritical_section0();
     }
```

```
P1 → while (True) {
        while (turn == 1);
        critical_section1();
        turn = 0;
        noncritical_section1();
     }
```

متغیر turn برای کنترل کاری که به نوبت مشترک می‌کنند یکی باشد.

اگر turn = 0 باشد نوبت برای P_0 است و می‌تواند وارد ناحیه بحرانی است اگر 0 نبود در این حلقه گیر می‌کند

ولی وقتی شرط برقرار بود وارد ناحیه بحرانی خودش می‌شود وقتی کارش در ناحیه بحرانی تمام شد نوبت را به P_1 می‌دهد

boolean متغیر

PAPCO

Subject:
Year. Month. Date. ( )

در این روش ایضا رقابل وجود دارد چون turn به صفر است یا ! در یک لحظه فقط یک مقدار را دارد. نوبت بین P_0 و P_1 در وقت آمده است و تعداد نوبت ها مساوی است چون هر کی نوبت را بعدش به اون یکی می دهد ← کلم توی جمع تاثیری در درست یا غلط بودن این که روش ندارد.

← شرط سوم برقرار است چون هر کسی حداکثر یک نوبت منتظر می شود چون نوبت بعد از این که کارش پیدا نام شد به دیگری می دهد این که این یک نوبت چقدر طول می کشد ربطی به موضوع ندارد.

← اما از این راه حل خوشمان می آید چون شرط سوم که نگفتیم ندارد یعنی در کل راه حل خوبی نیست حالا چرا؟

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

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

P_0 باید انقدر انتظار می کشد که کار ناحیه غیر بحرانی P_1 تمام شود و خیلی طول می کشد ولی اگر P_1 کاری با ناحیه بحرانی ندارد پس باید P_0 بتواند کار خود را با ناحیه بحرانی انجام دهد و اینکه 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 تند تند نوبت بخواهد دیگری بیرون بیاید. سریع تر را به سرعت می رساند و به هر دوی آنها به سرعت پایین می رساند

نوبت با P_0	
ناحیه بحرانی P_0	
نوبت با P_1	P_1 در ناحیه غیر بحرانی
ناحیه غیر بحرانی P_0	
انتظار	

شرط سوم) پیشرفت Progress ← کسی که داوطلب ورود به ناحیه بحرانی نیست در رقابت شرکت نکند

این راه حل خوبی نیست چون شرط سوم را ندارد می خواهیم راه حلی بدهیم ← متخصص تجربیات می کنیم که نشان دهیم کی داوطلب ورود به ناحیه بحرانی است.

Subject:
Year. Month. Date. ( )

(second attemp

```
P0 → while (True) {
     ① flag[0] = True;
پایان هم‌زمانی
     ② while (flag[1]);   ← wait
        critical_section0();
        flag[0] = false;
        noncritical_section0();
     }
```

```
P1 → while (True)
     ② flag[1] = True;
     ③ while (flag[0]);   ← wait
        critical_section1();
        flag[1] = false;
        noncritical_section1();
     }
```

flag ← نشان می‌دهد کی داوطلب است و اگر کسی بخواهد وارد ناحیه بحرانی شود flag خود را true می‌کند و برای وارد شدن به ناحیه بحرانی flag دیگری را چک می‌کند

این راه‌حل از نظر اختصار متقابل صحیح است و هر دو نمی‌توانند وارد ناحیه بحرانی شوند.
شرط سوم را هم دارد ← کسی که داوطلب نیست در رقابت نخواهد بود چون flag آن true نیست
آیا شرط دوم هم برقرار است؟
اصلا به شرط دوم نمی‌رسد و یک مشکل دیگری را رد نه کار می‌کند ← فرض کنیم ① هم‌زمانی برای P_0 است و flag خود را True می‌کند قبل از وارد شدن به ناحیه بحرانی هم‌زمانی تمام می‌شود و در بعدی وارد می‌شویم. نوبت ② به P_1 می‌رسد و flag خود را True می‌کند و وی در حلقه بعد که ③ flag[0] True است گیر می‌کند نمی‌تواند رد شود و صبر می‌کند تا هم‌زمانی تمام شود نوبت ④ به 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 این بیت است و راهی برای نجات ندارند و احتمال غیرصفر برای اتفاق افتادن این وجود دارد.

راه‌حل درست: این دو راه‌حل را با هم مخلوط می‌کنیم ← هم turn داریم و هم flag

Subject:
Year. Month. Date. ( )

تلاش سوم (third attemp)

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

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

flag خود را True کند ← می‌خواهم وارد ناحیه بحرانی شوم

اگر کسی Turn را دارد اما flag ندارد ← نمی‌تواند جلوی کسی را برای ورود به ناحیه بحرانی بگیرد.

احتمال دارد طرف مقابل را دیگر پیدا نکند.

← اگر هر کس turn را قبل از ورود به ناحیه بحرانی به خودش بدهد (الگوریتم حریص‌انه) ممکن است طرف مقابل از حالتی بیند از که نوبت به دیگری برسد. یعنی مثلاً P_0 چندین بار turn را بگیرد و به P_1 ندهد این چندین بار معلوم نیست چقدر است ← امکان اتفاق افتادن این غیرصفر است

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

محم زدن ساده‌ترین است و تضمین تاثیری که دارد اینست که حداکثر یک نوبت می‌تواند من را عقب بیندازد

تعمیم به n فرآیند:

همین راه‌حل را می‌توان به n فرآیند تعمیم داد البته کار خیلی سختی است. اینکه این را بدانیم خوب است ولی اگر خودمان بخواهیم پیاده‌سازی کنیم سخت است

در قبل دو حالت داشتیم ← می‌خواهند وارد ناحیه بحرانی شوند

می‌خواهند ... ...

Subject:
Year. Month. Date. ( )

idle — اما در این جا سه حالت را تعریف می‌کنیم ← می‌خواهد 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[i] != in_cs)) j++;
             if (j>=n && (turn = i || flag[turn] = idle)) break;
             }
        while(true)
        turn = i;
```

ناحیه بحرانی

می‌بینیم با هم برابر باشند

شرط اختصاص متغیر

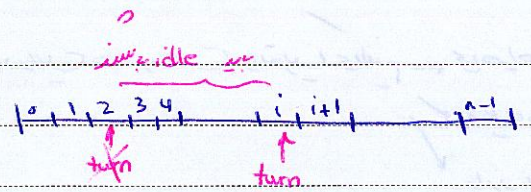
اگر نبینیم چیزی باشد تا in_cs نبینیم اگر دیدیم به از اول اگر مشکل نبود وارد ناحیه بحرانی می‌شود.

وقتی می‌خواهد وارد ناحیه بحرانی شود ← turn را به خودش می‌دهد بعد از اینکه همه چیز را چک کرده باشد

آخرین کاری است که انجام می‌شود یعنی مطمئن شده که می‌تواند وارد ناحیه بحرانی شود

Subject:
Year. Month. Date. ( )

ناحیه بحرانی چجوری کار می‌کند؟



چک می‌کند دونه دونه را از turn تا خودش که همه باید idle باشند اگر اینطور باشد می‌گوید حق با من است اگر idle نباشند در این ناحیه گرفتاری شود و انقدری بماند تا همه idle ببیند یعنی مجدد حق با خودش است اگر این حالت اتفاق افتاد flag خودش را به in_cs تغییر می‌دهد

اما هنوز این راه‌حل راه‌حل کاملی نیست و این مشکلات را در ناحیه خروج حل می‌کنیم

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

همه را چک می‌کنیم ببینیم کسی هست که idle نیست (اولین کسی که idle نیست) (نوبتش رسیده) idle نیست کسی است turn را به اول می‌دهیم اگر در این n تا فرآیند کسی وجود نداشت که idle نیست turn به خودم برمی‌گردم چون با نوبت کار می‌کند حداکثر n-1 نوبت طول می‌کشد تا نوبت به من برسد

تا چک‌هایی که آخر می‌کند برای ایجاد انحصار متقابل
شرط انتظار محدود وجود دارد حداکثر n-1 نوبت
واضح است شرط سوم هم هست چون اصلا افراد idle را دخالت نمی‌دهد

Subject:
Year. Month. Date. ( )

جلسه پانزدهم 8,19

ایمیل تمرین‌ها: Pedram@aut.ac.ir

راه‌حل جلسه قبل برای ۲ OS نداشت

الگوریتم نانوایی baker's algorithm ← در عمل استفاده نمی‌شود و فقط گفته می‌شود

هر کس که می‌خواهد وارد ناحیه بحرانی شود شماره می‌دهیم (نوبت‌دهی) و براساس آن شماره می‌توانند سرویس بگیرند

یعنی وارد ناحیه بحرانی شوند

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

وقتی صحبت شد یعنی داخل نیست و داخل حساب نمی‌آید

```
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));
}
```

شانس‌ی که همه در حال number گرفتن هستیم

همه شماره بگیرند

max را بگیر و یکی به آن اضافه کن

یک به یک هر کس طوری است به بررسی آن بپردازیم

نوبتی است که در حال شماره گیری است صبر می‌کند تا آن شماره‌اش را بگیرد

منتظر می‌مانیم تا 5 بیرون بیاید تا تمام شود

برای سلسله‌مراتب درست شماره

من میدم number کمتر نباشد

تا وقتی number کسی از من کوچکتر است من می‌توانم وارد ناحیه بحرانی شوم

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

Subject:
Year. Month. Date. ( )

مرحله شماره گیری، مرحله طولانی است ← بدین معنی تعداد max بالا می‌رود و مقابل زمان ماشین آن طولانی می‌شود پس در وسط کار می‌توان قطع شود ← بنابین خاطر همه را باید به حساب آورد حتی کسی که در حال شماره گیری است چون عمل دست قبل از من باشد و وسط کار قطع شده

همچنین امکان شماره دهی مساوی وجود دارد ← این کار را می‌کند اگر دمتا شماره مساوی بود به شماره فرایند برای شکستن حالت تساوی نگاه کن ← اول شماره ها چک می‌شود اگر مساوی بود شماره فرایند چک می‌شود

یک مشکل الگوریتم رو حل نشده: number هنا تا زمانی بالا می‌رود که overflow بدهد وقتی overflow دارد چکار کنیم؟ حل آن بسادگی نیست ← دچار اشتباه می‌شویم

این راه حل تحت شرایطی کار می‌کند ← اگر زمان های وجود داشته باشد که هیچکس درخواست ورود به ناحیه بحرانی نیست در این صورت number ها صفر می‌شود یعنی زمانی کار می‌کند که این ها دائم در رقابت نباشند و از این زمان ها این وسط باشد در عمل هم این طوری یعنی دائم در حال رقابت نیستند

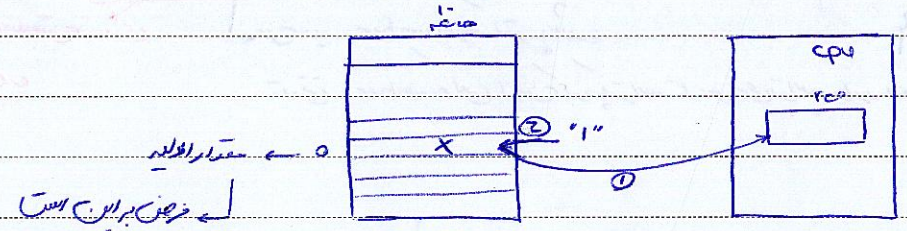
راه حل بسیار ساده

تا به حال در مورد دستورالعمل های ماشین فرض کردیم (بسیار ساده اند) الان الان مستقر می‌کنیم که مقداری دستورالعمل پیچیده دارد (دستورالعمل های غیرساده)

دستورالعمل های ماشین

tsl → Test_and_Set → بیش از یک عمل را انجام می‌دهد

test_and_set_logical



① محتویات حافظه به reg

② به عدد غیرصفر بخورد ! set می‌کند

اگر دستم بگیریم cpu این دستورالعمل را دارد راه حل بسیار ساده می‌شود.

Subject:
Year. Month. Date. ( )

زبان assembly :

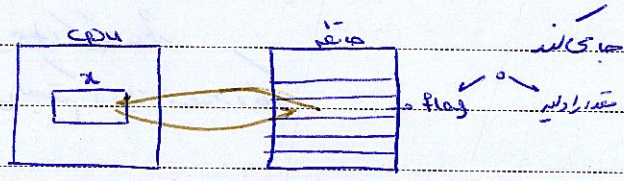
```
enter_critical_section
    tsl   reg2 , flag
    cmp   reg2 , #0
    jnz   enter_critical_section
    ret
exit_critical_section
    move  flag , #0
    ret
```

یعنی حسابی نباشد جلوی حساب می‌شود zero = 1 ← یعنی حساب نباشد جلوی حساب می‌شود

هر فرآیندی که وارد ناحیه بحرانی می‌شود flag را یک می‌کند مثلا flag=1 باشد یعنی کسی وارد ناحیه بحرانی استفاده می‌کند
پس اگر فرآیند جدیدی خواست وارد ناحیه بحرانی شود تا زمانی که flag صفر نشده نمی‌تواند وارد شود در ضمن حریص زمان
دو تا فرآیند نمی‌توانند tsl انجام دهند
در tsl ← مهم زمانی در وسط کار قطع نمی‌شود چون یک دستور العمل واحد است.

← فرض در مورد CPU درست بعد ← Intel این را ندارد چیزی شبیه این دارد

swap →



محتویات این جا با جلو جا می‌کند

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

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

برای هر منبع می‌توانیم ناحیه بحرانی زیادی تعریف کنیم پس کلی تعداد flag زیادی داشته باشیم

Subject:
Year. Month. Date. ( )

این راه حل انحصار متقابل را کاملاً حفظ می‌کند (تضمین شده و نمی‌تواند خراب کند)

مشکل این راه حل کجاست؟

شرط محدودیت زمانی را تضمین نمی‌کند (شرط سوم را دارد (کسی که داوطلب نیست، مداخلت داده نمی‌شود)
محدودیت زمانی تضمین نمی‌شود.

← نوبت به طور کاملاً تصادفی گرفته می‌شود یعنی اگر 5 تا فرآیند داریم یکی گرفت، 4 تای خواهند بین این 4 تا تصادفی است. احتمال غیرصفر دارد که یکی اصلاً نوبت بهش نرسد

← این راه حل ضمن سادگی، صحیح نیست

← دنبال راه حل صحیح نبودیم، دنبال چیزی بودیم که تحت شرایطی کار کند

این راه حل تحت چه شرایطی کار می‌کند؟ اگر به طور دقیق واقعی داشته باشیم که هیچ کس داوطلب نباشد کار می‌کند ولی اگر همیشه فرآیندها در رقابت باشند این راه حل کار نمی‌کند و در واقعیت این است که زمان‌هایی وجود دارد که هیچ کس داوطلب نباشد

← اگر در ناحیه بحرانی لیدر شدن بین سه است و برای تمام این راه حل‌ها ممکن است اتفاق بیفتد

← قسمت ناحیه بحرانی باید خیلی کوچک باشد

این راه حل درست و مناسب دنبال راه حلی نیستیم که برای همه موارد کار کند

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

← نکته مشترک تمام راه حل‌های ارائه شده:

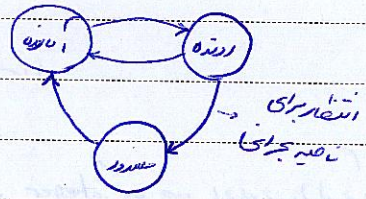
از انتظار مشغولی (busy waiting) استفاده می‌کنند

← برای انتظار از حلقه استفاده می‌کنند ← زمان پردازنده‌ی جهت تلف می‌شود و هیچ کاری خفید انجام نمی‌شود ← راه حل‌های ارائه شده که فرآیندها را حسد می‌کنند

PAPCO

Subject:
Year. Month. Date. ( )

می‌خواهیم راه‌حلی ارائه کنیم که انتظار مشغول نداشته باشد و از حالت مسدود استفاده می‌کنیم.



کجایی که می‌خواهند منتظر ناحیه بحرانی باشند به حالت مسدود ببرند

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

مفهومی بیان می‌شود که تحت عنوان یک ابزار پیاده‌سازی شده

سمافور (Semaphor راهنما) ← یک متغیر int به همراه صف که اندیسی می‌گیریم چه موقع باید وارد ناحیه بحرانی شده و یک موقع باید استراحت کند

ابزاری برای هماهنگی فرآیندها (ورود به ناحیه بحرانی)

یک ساختار داده‌ای است ← یک متغیر int
یک صف

در حالت مسدود فرآیندها بنابر سرویسی که می‌خواستند در یک صف منتظر می‌شوند این صف Semaphor را هم در این جا می‌گذاریم ← فرآیندی که به بقیه سرویس‌ها می‌شود

Semaphor ← یک مجوز برای وارد شدن به ناحیه بحرانی
← برای هماهنگی به کار می‌بریم

۸۳۲۰۹۴۶۹ | ۱۱۳۸۵
۷۵۵۸-۱۳۹۵۲۵-۱ | ۵۵۹-۸۳۷۲۹۰-۲

Subject:
Year. Month. Date. ( )

جلسه شانزدهم 19/8/

تمام راه‌حل‌های ارائه شده از busy waiting استفاده می‌کنند

Semaphore (سمافور، راهنما)

ابزاری برای هماهنگی

ساختاری شامل یک integer یک صف دارد. (از فرآیندها)

عملیاتی که روی این ساختار داده هست را هم باید بگوییم

عملیات روی Semaphore

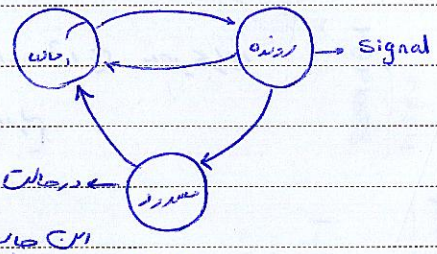
مقداردهی اولیه به مقدار integer را باید set کنیم و عددی برای شروع به آن بدهیم

عملیات مهم روی Semaphore ← wait

signal

wait ← یک واحد از integer کم می‌کند و نگاه می‌کند که آیا مقدار منفی شد یا نه. اگر منفی شد فرآیند مسدود می‌شود و در صف قرار می‌گیرد (در صف) در غیر اینصورت فقط یک واحد کم می‌کند و عبور می‌کند

signal ← یک واحد اضافه می‌کند و چک می‌کند که مثبت نباشد ← اگر مثبت نباشد یکی از فرآیندهای مسدود شده آزاد می‌شود (آن که در صف است آزاد می‌کند، آن که در نوبتش است)



← در حالت signal یکی از فرآیندهایی که این جا بود به حالت آماده می‌آید و برای خودکسی که signal را اجرا کرده اتفاقی نمی‌افتد

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 پروسه‌ها

=> اگر منفی شد blocked می‌کند

Binary Semaphore

نوع دیگری از semaphore ← binary sem ← به جای integer یک bool استفاده می‌کند و برای شرط روشن خاموش است

اصطلاحاً می‌گویند عملیات wait و signal اتمی هستند یعنی نمی‌شود متقاطع کرد (مثل Test & set) یا یک یک چه انجام شود می‌شود تجزیه کرد. اگر wait می‌کنیم باید کل wait انجام شود و یا هیچی انجام نشود. پیاده‌سازی همین چیزی راحت نیست مثل اینکه بخواهیم می‌گوییم نباید

Subject:
Year. Month. Date. ( )

این استفاده از semaphore در ناحیه بحرانی است.

حل مسئله ناحیه بحرانی (برای n فرآیند) ← بدون busy waiting داریم

اختصار متقابل

```
/* 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 داریم برای این کار که 1 به در بدی خورد ...

وارد ناحیه بحرانی می‌شود، هر کس بیاد برای این 1 → 0 ...

کم کند منفی می‌شود می‌تواند وارد ناحیه بحرانی شود.

signal(s): هر صدا جمع شد یکی از فرآیندهای منتظر را آزاد می‌کند اگر کسی نبود که آزاد کند count را 1 می‌کند تا هر کسی آمد بتواند از ناحیه بحرانی استفاده کند.

این راه حل ساده هم کار می‌کند هم busy waiting نیست

← قدر مطلق عدد منفی (s.count) نشان می‌دهد چند فرآیند در صف داریم.

1. انتظار محدود وارد چون در صف می‌گذاریم جای نوبت دهی می‌کنیم
2. چون مقدار اولیه 1 بود فقط یک فرآیند می‌تواند وارد شود اگر 2 می‌دادیم، 2 تا فرآیند وارد می‌شوند
3. کسانی که داوطلب نیستند که در صف قرار نمی‌گیرند پس اصل شرایط داده می‌شود

Subject:
Year. Month. Date. ( )

پیاده‌سازی این با شرط انتخابی بودن wait و signal خیلی سخت است و بدون این شرط‌ها این راه‌حل کار نمی‌کند. مثلاً یک زمانی 1 میده، wait را انجام داد ولی هنوز signal نزده بعدی هم 1 می‌بیند و دل هم وارد ناحیه بحرانی می‌شود.

با داشتن این ابزار می‌توانیم کاربردهای دیگری نیز داشته باشیم

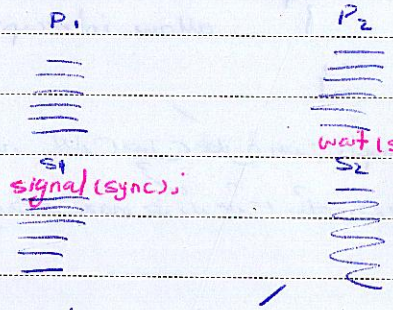
← دو فرآیند P_1 و P_2 داریم که دستورالعمل‌ها به صورت همروند انجام می‌شوند

سیستم طوری است که S_1 باید زودتر از S_2 انجام شود (وقتی که S_1 چیزی را تولید می‌کند که S_2 باید از آن استفاده کند)

S_1 زودتر از S_2 اجرا شود: حکم

مختصاً قابل تضمین نیست ← حکم زمانی ← با تعریف یک semaphore این کار را انجام می‌دهیم

Semaphore Sync = 0 =>



پس در S_2 wait می‌کنیم و S_1 را انجام می‌دهیم و بعد block S_2 می‌شود و وقتی S_1 انجام شد S_2 را آزاد می‌کند. اگر حکم S_1 زودتر پیدا اجرا شود و sync = 1 می‌شد بعد wait می‌کنیم 0 می‌شود و منفی نیست S_2 هم انجام می‌شود.

پیاده‌سازی:

دو راه‌حل وجود دارد:

راه‌حل اول: چیزی که اینجا را قفل می‌کند وقفه حکم زمانی است پس اگر همه وقفه‌ها را از کار بیندازیم دیگر قطع نمی‌شود البته باید برای از کار انداختن وقفه‌ها در System mode باشیم

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

Subject:
Year. Month. Date. ( )

← کی داره این کارها را انجام می‌دهد؟ یکی غیر از فرآیند چون در user mode است و عملاً این کارها را OS انجام می‌دهد و وقتی مسدود می‌کند خودش را که مسدود می‌کند

```
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 همان موقع باید پاسخ دهیم (مثلاً اینکه یک متغیری از max خودگذشت مثل درجه حرارت) ولی ممکن است استفاده درست نشود و بستگی به کاربرد دارد.
ما کاری می‌کنیم که این را قفل نکنیم ← زمان ما در این فاصله از دست می‌رود

این را حل مناسب‌تری داریم؟ همان جعبه جادویی

```
void wait (semaphore s)
{
  while (!testset (s.flag));   ← نکته: یک flag هم داره OS که عوض شده
  s.count --;
  if (s.count < 0) {
      place ___ ;
      block ... ;
      set s.flag to zero;
  }
  else set s.flag to zero;
}
```

PAPCO

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) ناحیه بحرانی تعریف کردیم یعنی گفتیم اول test & set کند که ببیند می‌تواند s.flag را بدست بیاورد یا نه

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

test&set ← مطمئن شدیم که flag را بدست آوردیم

این جا دیگر interrupt نداریم پس روش اجتناب بودن نداریم ولی اثر اجتناب بودن را بدست می‌آوریم
این جا هم می‌گوییم چون داریم wait یا signal می‌کنیم چون می‌توانی flag را بدست آوردی پس
می‌توانی wait و signal کنی. حالا کم زمانی را از ما بگیرد مهم نیست چون می‌توانند flag را بگیرد پس
می‌توانند wait یا signal کند

ظاهر قضیه این است که داریم نقض غرض می‌کنیم روش‌های قبلی برای ناحیه بحرانی کوچک مناسب بود ولی
در ناحیه بحرانی می‌ایستیم دقیقاً چقدر طول می‌کشد ولی در این جا دقیقاً می‌دانیم چقدر طول می‌کشد

Subject:
Year. Month. Date. ( )

داریم test & set را برای جایی به کاری برسیم که کوتاه است و هیچ کار گرانی انجام نمی‌شود و کسی معطل نمی‌ماند چون انجام عمل signal، wait کوتاه است.

اما ناحیه بحرانی اصلی را wait و signal کنترل می‌کنند

← بر سر منابع حساس و مشترک کار می‌کنند

test & set (flag)
wait (s) → گرفتن را حساس می‌کند
flag = 0
ناحیه بحرانی
test & set (flag)
signal → از حالت حساس خارج کند
flag = 1

اجرای wait و signal به پشتیبانی OS را می‌خواهد

← حساس و در صف قرار دادن

حتماً باید system call باشد نمی‌تواند باشد بلکه همچنین می‌تواند یک library باشد که از چند system call استفاده کند به طور صریح یا ضمنی از system call استفاده می‌کند

Subject:
Year. Month. Date. ( )

جلسه هفدهم 87/8/21

مسئله حافظه محدود

۱- تولیدکننده و مصرف کننده ← حافظه محدود، کفا توسط بافر بود حالا با کامپیوتر انجام می‌دهیم

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

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

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

تولیدکننده:

آزاد پیوستن

دیگری باز کنترل می‌کند

آیتم را به بافر append می‌کند یعنی در بافر بگذار

اگر خالی باشد نمی‌توانیم از آن چیزی برداریم

مقدار را از بافر برمی‌دارد

Subject:
Year. Month. Date. ( )

```
signal(e);
consume();
void main
```

تعداد سلول های خالی را یکی زیاد می کند

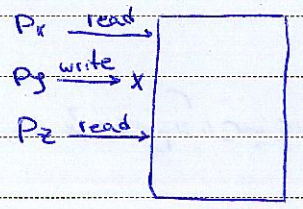
یعنی semaphore s برای دسترسی اعضای که به کار رفته در نتیجه مقدار اولیه = 1
اگر می خواستیم دو نفر بگذاریم ← جلو می کردیم باز پر نباشد
" " از " بخوانیم ← " " " " خالی "

چون e برای تعداد پر " بودن است باید به اندازه سایز باز باشد چون یکی اضافه کنیم یکی ازش کم می شود
می توانیم اضافه کنیم تا جایی که صفر بشود بعد منفی می شود دیگر نمی توانیم از wait عبور کنیم
f برای تعداد خالی بودن است ← تعداد سلول های پر شده را نشان می دهد
e تعداد سلول های خالی را نشان می دهد.

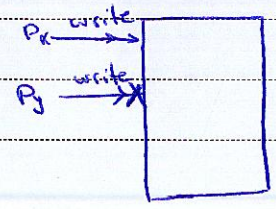
← پس همین که این کاربرها فقط s برای ناحیه بحرانی به کار رفته و بقیه برای تعداد سلول های پر است و این مثال
می گوید که کاربرها برای ناحیه بحرانی فقط نیست

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

اینجا یک object مشترک مثل file وجود دارد و فرایندهای مختلف می توانند از این object بخوانند یا بنویسند ولی
بدون هماهنگی نمی توانند این کار را انجام دهند مثلاً وقتی خواننده دیگری می تواند در آن بنویسد
برای read کردن همزمان می توانند انجام دهند چون اطلاعات
عوض نمی شود و اشکلی ندارد read می کنند فقط می توان در آن
write کرد



ولی وقتی write می کنیم هیچ دیگری نمی توانیم
انجام دهیم چون اطلاعات قابل اعتماد نیست
و عملیات write انحصاری است

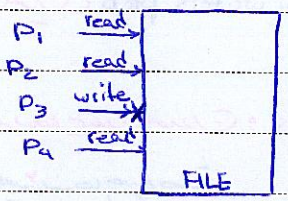


Subject:
Year. Month. Date. ( )

اینجا P_4 هم باید منتظر بماند یا نه بستگی به مسئله دارد:

1. سیستم عامل به صورتی است که هر کدام که جلوتر است هم انجام شوند

2. شاید اینجا تا ابد بخواهند بخوانند write به چی کار دارند



First readers & writes prob ← به P_4 اجازه خواندن دهیم

Second " " " " ← به P_4 اجازه ندهیم قبل از write بخواند

(انحصاری بودن write در هر دو مشترک است.)

First readers & writers prob: ←

دستور آیند ← هم read و هم write می‌کنند

فرآیندی نویسنده: P_w (پوشش انحصاری است.)

wrt ← برای پوشش انحصاری به کار بردیم پس مقدار اولیه آن یک است

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

از این به بعد دلیل‌های توالی خواندن را انجام می‌دهی

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

برای تعیین ست readcount

چون کسی در حال write نباشد ←

فرآیند خواننده: P_r

اگر خواننده اول باشد یا نه رفتارش متفاوت است

اول بودن ← باید چک کنی که کسی می‌نویسد

اول نبودن ← لازم نیست چک کنی

readcount ← مقدار خواننده‌ها پس هر کی می‌خواهد بخواند باید این حساب کی نگه دارند

← mutex دیگری به readcount را انحصاری می‌کنیم

پس مقدار اولیه‌اش یک است

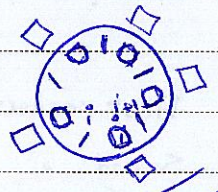
بعد از خواندن باید بینی اگر آخرین خواننده هستی باید آن کسی را که منتظر write است آزاد کنی

P4PCO

Subject:
Year. Month. Date. ( )

بعنوان تمرین Second readers & writers prob. را انجام دهید

3- فلاسفه در حال غذا خوردن : Dining Philosophers



این فیلسوف ها چینی هستند

هر کسی می خواهد غذا بخورد باید دو تا چوب در اختیار داشته باشد

دیگر نباید منتظر بماند

فرضیات: فیلسوف ها همیشه در حال خوردن نیستند یعنی وقتا متفکر می شوند

منابع: چوب ها

Function: eating (عمل 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));
  }
```

بالا بردن wait کردن این دو است

=> گیر دادیم

دیدیم این راه حل یک مشکل دارد.

semaphore ابزاری قوی برای هماهنگی است ولی مشکلات دارد

Subject:
Year. Month. Date. ( )

مشکلات Semaphore

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

2. بکارگیری صحیح ← برای هر wait یک signal داشته باشیم و برعکس چون ممکن به deadlock برسد

wait(s);
=
signal(s);
=> صحیح

signal(s);
=
wait(s);
این دیگر اشتباه است

wait(s);
=
wait(s);
بن بست

wait(s);
=
wait(s);

3. بن بست. ممکن است پیش بیاید حتی اگر درست استفاده کنیم

هر دو قرارداد کاملاً قانونی هستند

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

① باعث کم زمانی
④ ← مسدود چون wait در P_2 شده

③ ← نمی‌تواند رد شود
مسدود تا یکی p را آزاد کند

اینجا هر دو گرفتار شدند و هیچ راهی برای خلاص شدن ندارند.

این مشکلات در مسئله Dining Philosopher وجود دارند ← همگی بدون جواب بروند منتظر دوری باشند در نتیجه هیچ وقت این مشکل حل نمی‌شود جلوی نمی‌رانند یکی دیگری منتظر. این منبع است

← می‌شود کاربردهای از نحو پیدا کرد که به درد نمی‌خورد

راه حل این مشکل اینست که اساساً اختیار به برنامه‌نویس ندهیم که هر طوری می‌خواهد بکند پس به سراغ ابزار سطح بالاتری می‌رویم ← monitore (مانیتور) یا ناظر

اینجا جانشین هم نیستند بلکه جایی در جای خودش استفاده می‌شود و جای همدیگر را نمی‌گیرند

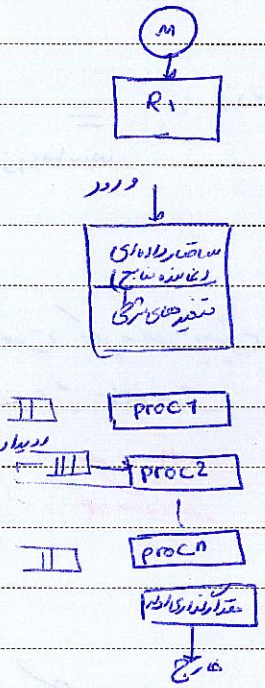
Subject:
Year. Month. Date. ( )

تمرین های 9,5,6,7,14,15

جلسه هجدهم 8/26

یکی از ابزارهای کمکی جهت ایجاد هماهنگی بین فرایندهای باشد البته می تواند جایگزین سمافور شود. اختیارات آن محدودتر است و یک سری از اشتباهات سمافور را نمی دهد.

مثلا کنترل عملیات سری منبع دارد و سری عملیات هم کنترل عملیات دارد پس اختیارات آن محدودتر می شود



مثال از monitor

۱. در هر لحظه فقط باید یک فرایند داخل monitor فعال باشد

در غیر این صورت اختصاص متقابل نقض می شود

تعداد این صف ها فقط می تواند چندین تا باشد

اگر یکی سری سیداری رفت و منتظر شد می توان فرایند دیگری را وارد کرد و این اختصاص متقابل را از بین می برد

فرایند قبلی یا خارج شده است یا مسدود.

این سیدارها متغیرهای شرطی هستند

مثال) تولید کننده و مصرف کننده.

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

ساختمان داده های سابقه

char → می تواند هر چیزی غیر از char باشد

semaphor mutex

condition

Subject:
Year. Month. Date. ( )

```
void append (char x)
{       wait(mutex)
    if (count==N)                  →                    شرط پدر
        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;
```

این ها در monitor ← consumer بوده producer بوده

```
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 اعضای هست این اعضای بولن که تعریف شده است.

در keyword ، monitor هست که وقتی این کلمه compiler می‌بیند، compiler به بدنه این keyword می‌رود که یک کلاس یا به آن اضافه شود و این یک کلاس می‌باشد و باید یکسری چیزها به آن اضافه شود تا ساخت شود حال به جزئیات متغیرهای * را اضافه می‌کند

← جایی که از monitor استفاده می‌شود باید از compiler که معنای خاصی برای monitor داشته باشد استفاده کند. ← در حالت دقیق‌تر باید در تست cwait هم چیزهایی اضافه می‌شود

مثال) فیلسوف‌ها ← غیرانحصاری thinking ، eating حالت hungry هم دارند

```
monitor Dining Philosophers;

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

Subject:
Year. Month. Date. ( )

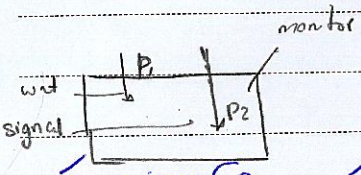
```
void pickup(i)
{ state[i] = hungry;
  test(i);
  if(state[i] != eating)
        cwait(self[i]);
}

void put down(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) {
  Dining Philosophers.pickup(i);
   eating();
   Dining philosophers.put down(i);
   thinking(i);
}}
```

مقتاد الزاري الدلي

P4PCO

Subject:
Year. Month. Date. ( )

نکته) فرض کنید یک monitor داریم و یک proc P_1 وارد این می‌شود و وسط کار wait می‌کند از طرف دیگر یک P_2 وارد monitor می‌شود و پس از مدتی P_2 متغیری که P_1 روی آن wait شده signal می‌شود از این لحظه به بعد P_1 باید ادامه پیدا کند یا P_2



جواب صحیح مدرس چون کمی سخت بگوید P_2 در تخلیه P_1 را سیگنال کرده است باید انجام شود تا P_2 بعد از آن شروع کند

پس اینکه کدام یک از دو صورت بسته به مثال دارد اما بهترین راه حل اینست که signal کردن P_2 آخرین کاری باشد که P_2 می‌کند و پس از آن بیرون برود و کمی سخت‌تر است چون کاری کرد

مثال 2 کدام راه حل را اجرا می‌کند؟

1- monitor که عملیات را کنترل می‌کند ولی خود این کار را نمی‌کند

2- نیاز به پشتیبانی compiler داریم تا بتوان این کار را انجام داد ولی باید دقت نمایی کرد ولی خود قطعه پشتیبانی OS نیاز دارد ولی compiler آن را compile می‌کند سپس برای کنترل نمایش دهد که اضافه می‌شود نیاز به OS هم دارد

3- اینکه P_1 باید انجام شود یا P_2

جنبال برنامه‌نویسی را می‌شناسید که monitor را شناسه Java

کلمه monitor را ندارد ولی اگر یک object synchronized تعریف کنیم این خاصیت را دارد در قدیم جاوا هم چون compiler نبود خودمان باید این کار را اضافه می‌کردیم

PAPCO

Subject:
Year. Month. Date. ( )

جلسه نوزدهم 8, 28

OS های موجود بطور مستقیم از سخت‌افزار پشتیبانی نمی‌کنند بنابراین داشتن مشکل است.

monitor ← نیاز به پشتیبانی compiler

semaphore ← نیاز به پشتیبانی OS دارد

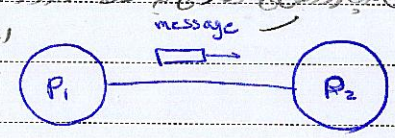
ما می‌توانیم برایش یک library routine تهیه کنیم که انجام شود.

استفاده از فراخوان‌های سیستمی Send , recieve ← تبادل پیام یا message passing

این راه OS ها دارند و نیاز به پشتیبانی خاصی ندارد خیلی ساده است ولی کاربرش مشکل است

دو تا فرآیند داریم و می‌خواهیم از هر فرآیندی به فرآیندی دیگر message ارسال کند

این کجا ذخیره می‌شود؟ جایی در OS ندارد و فقط ایجاد کننده این ارتباط است



message هر چیزی است و راحتش هر چی می‌تواند باشد (مثلاً pointer یک ساختار)

Send (m, P2) ← Send می‌کند

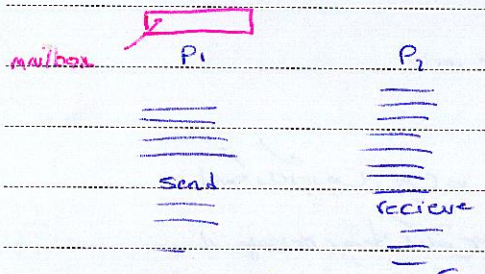
از سمت header این می‌فهمد کجا برسد

message یک قطعه ← recieve (m) ← هر message که دریافت کرد.

هر کی حق داره message رو بنویسد

تا زمانی که این اجرا شد P2 ، message را می‌گیرد

در مقابل Send ، فرآیند مقصد حتماً باید recieve را اجرا کند



وقتی Send نشده چرا recieve می‌کنی؟

انتظار برای Send باید زودتر از recieve اجرا شود

این شرط سنگین است

راه حل: (در حالت منطقی Send باید زودتر از recieve باشد)

برای هر فرآیند OS یک جعبه به نام mailbox برایش درست می‌کند هر پیامی که به فرآیندی می‌رسد داخل mailbox می‌رود

هر وقت بستر recieve را اجرا کرد یک پیام از mailbox برمی‌دارد.

(چون message ها در mailbox می‌روند پس گم نمی‌شوند)

← recieve از نوع سیستم call های blocking است یعنی اگر نتوانست کاری انجام دهد (هیچی نداشتیم) فرآیند را مسدود می‌کند و نگه می‌دارد در حالت مسدود تا زمانی چیزی داخل mailbox بیاید ← کاربر چیزی نمی‌فهمد

بهترین نوع recieve همین است چون مثلاً دستش به دختی از block خارج می‌شوی حتماً اول چیزی را که می‌خواستی داری ←

Subject:
Year. Month. Date. ( )

حل مسئله producer consumer

```
void Producer ()
{
  int item ;
  message m;
  while (True) {
      produce (item);
      recieve (m);
      build_message (m, item) ;   ← item را کپی message میکند و پشت پیغام میگذارد
      Send (consumer, m) ;   → میگاییم که Send شود
  }
}

void Consumer ( )
{
  int item, i;
  message m;
** for (i=0; i<N, i++)  Send (producer, m)
  while (True) {
  → recieve (m) ;   Send یک پیغام برمیگردد و recieve می‌شود
  extract_item (item);   از message، item را پیشش میبرد و item را جدا کند
  Send (producer, m) ;  →
  consume (item) ;  →   مصرف کن
  }
}
```

Subject:
Year. Month. Date. ( )

فرض می‌کنیم mailbox نداریم یعنی نمی‌توانیم Send کنیم می‌توانیم recieve کنیم.

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

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

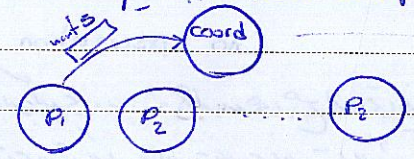
** در این خط قبل از هر کاری N تا مجوز می‌فرستد پس هیچ وقت بعد recieve در producer لیدر می‌کند چون که mailbox ها پر است و هیچ وقت block نمی‌شود اگر ۱ مجوز بفرستیم به این معنی است یکی من یکی تو است ولی در این جا producer می‌تواند تولید کند زیادی نداشته باشد که consumer مصرف می‌کند یا نه

← ما می‌توانیم Semaphore را با استفاده از این بسازیم

اگر بتوانیم semaphore را شبیه‌سازی کنیم می‌توانیم تمام مدهایی که می‌گوییم الان هم انجام بدهیم

شبیه‌سازی Semaphore

تجاری نرم‌افزار هستیم می‌خواهیم از یک Semaphore مشترک ایجاد کنیم



rec
open message
wait Operation
if not negative send

یکی از این‌ها باید Semaphore را ایجاد کند ← ایجاد Semaphore: یک فرآیند محلی ایجاد می‌کنیم فرآیند محلی در حالت recieve قرار می‌گیرد (block)

P1 یک message (wait) به دست می‌گیرد به Coord می‌فرستد بلند صدا چطور؟ منتظری دائم شاید چند تا منتظر شده Signal کند

P1 .. wait(S) rec می‌آورد

messag | wait | S | ...

coordinate message را بازخوانی می‌کند می‌بیند wait سری S است

پس wait را اجرا می‌کند اگر منفی نشد send می‌کند

send(coord,)

recieve => block شد

← محال که از بین نرفتی و اگر منفی شدی چیزی عقب نفرستاد دست

مثل wait است)

P4PCO

Subject:
Year. Month. Date. ( )

اگر می‌خواهیم داشته باشیم، monitor را هم می‌توانیم پیاده‌سازی کنیم (اگر OS آن را نداشته باشد)

معمولی 5 ← 2، 9، 5، 11

جلسه بیستم

بن‌بست Deadlock

این بحث را زیاد کش می‌دهیم چون در سیستم‌های تک پردازنده که ما با آن‌ها کار داریم چندان مطرح نیست چون اکثراً هر فرآیندی که چیزی می‌خواهد از OS درخواست می‌کند البته خودش هم می‌تواند اقدام کند ولی اکثراً این جوری نیست - البته این بحث در جای خود معتبر است.

اگر 4 شرط زیر برقرار باشد بن‌بست پیش می‌آید

1. انحصار متقابل mutual exclusion - یعنی اگر منبع انحصاری نباشد بن‌بست پیش نمی‌آید - (منابعی از منبع غیرانحصاری رها شده است)

2. نگهداری و انتظار hold & wait - دو تا منبع داریم برای هر دو مصرف‌کننده داریم اگر یکی را رها کنیم که نمی‌کنیم) ← (وقتی که یکی از مصرف‌کننده‌های مال یک منبع را از دست می‌دهیم آن را رها نمی‌کنیم تا فیلدی را بدست بیاوریم) وقتی کار ما پیش می‌رود که همه مصرف‌کننده‌ها را بدست بیاوریم و در نهایت کرسنگی پیش می‌آید

3. قبضه‌کن نباشد no preemption

منظور پردازنده است که اگر یکی را بخواهد دوباره بدهی هیچ اتفاقی نمی‌افتد ولی هر منبع این جوری نیست (اگر یک منبع را که داریم تا خودش تمام شود کن است بن‌بست پیش آید ولی اگر بتوان آن را قبضه کرد دیگر بن‌بست پیش نمی‌آید)

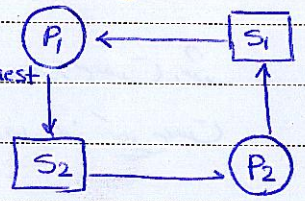
4. انتظار چرخشی circular wait

شرط ساختاری نیست بلکه کن است در حین کار پیش بیاید. هر چه را که بررسی کردیم این دیگری را داشت

PAPCO

Subject:
Year. Month. Date. ( )

allocation تخصیص منبع

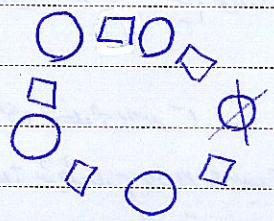


حالا P_2 هم درخواست را دارد ولی در اختیار P_1 است پس می‌رود در حالت انتظار

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

در مثال فلسفه‌ها انتظار چرخشی بود که در آن انتظار چرخشی بین 5 تا فرمیز است و هیچ راهی هم ندارد مگر اینکه یکی را نابود کنیم و بیرا همه‌ی بشر

ساده



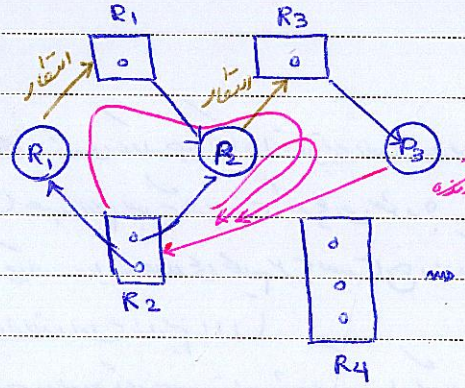
مثال:

R ← منبع resource

تخصیص منابع

request ←

R_1 را می‌خواهد ولی در اختیار P_2 است می‌رود در انتظار



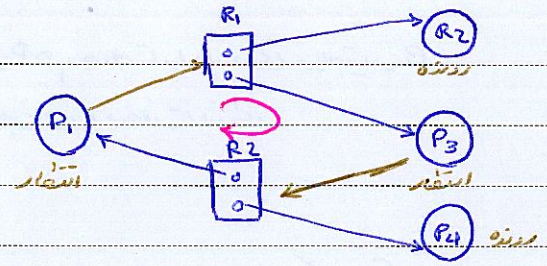
از منبع دیگر نوع بیشتر از یکی داریم (چند تا دیسک)

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

* حالا که هر 3 تا در انتظار رفتند و منتظر هم اند بن بست داریم. در تا حلقه به وجود آمده

P4PCO

Subject:
Year. Month. Date. ( )



مثال) حلقه دیده می‌شود

* لزومی ندارد همه فرآیندها از این بست باشند

در حلقه - هر تعداد از کسانی که در حلقه اند کمین بست

درین بست باشند

با وجود حلقه درین بست نیستیم می‌توانیم امید وار باشیم که P_2 تا R_1 را آزاد کند پس P_1 می‌تواند R_1 را بگیرد تا

انجام شود

راه برای خلاصی است.

← وجود حلقه شرط لازم برای این بست است ولی کافی نیست

(اگر این بست باشد قطعاً حلقه هست)

راه حل های این بست.

1- در OS های یک پردازنده ای به کاری بود که چون کم اتفاق می‌افتد ← بعد راه حل. در این حالت چون راه حل

نداریم یعنی تشخیص این بست هم نداریم پس اگر پیش آمد چی می‌شود؟ کاربر می‌بیند hang کرده پس یا process

را از بین می‌برد یا reset می‌کند. ویا با اصلاح می‌خیم این بست رخ داده فکر می‌کنیم یک مشکلی بوده و چون

هم نیست راه حل ارائه دادن نداریم!!!)

در یک پردازنده چون یک پیش داریم حتی اگر این بست پیش می‌آید (ولی اگر چند تا پیش بود OS ...)

2- الگوریتم بانکدار - اجتناب از این بست Dead lock Avoidance

مسیر ما را طوری تعیین می‌کند که به کمتر این بست نزدیم

3- تشخیص این بست (متاسفانه هنوز راه حل برای خود همین تشخیص قدم اول است که قطعاً ساده هم نیست)

4- جلوگیری از این بست Dead lock Prevention ← اصلاً در سیستم های یک پردازنده ای که برد ندارد

در سیستم های توزیع شده برد دارد چون به شدت اتلاف منابع دارد

طراحی اولیه سیستم به گونه ای که یکی از این 4 شرط اصلاً وجود نداشته باشد ← نقض همیشگی یکی از این 4 شرط

Subject:
Year. Month. Date. ( )

- الگوریتم بانکدار که روی آن مانور زیادی داده شده در هیچ جا کاربرد ندارد ولی برای سوال طرح کردن خوب است! (می‌شود بچپانید)

منابع را یک جوری تخصیص می‌دهیم که همواره قسمت امن بسته نزدی. از ترس اینکه بن بست نشود منبع را دارد توهم داری نه که می‌زنی ولی بعضی می‌رسد!!

مثال: ربات ها (در نقش فرایندها) همان مسیری را که امید برویدا!!

- تشخیص بن بست در سیستم های توزیع شده صرف دارد (با ساختن گراف تخصیص منابع) و بعد از تشخیص با برگشت به عقب از بن بست خلاص می‌شود (حالتی که هنوز بن بست رخ نداده ...)

(جزو روش تخصیص نیست)

جلوگیری از بن بست در سیستم های توزیع شده دیده شده ولی 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 که حداکثر تقاضاها از منبع R_1 به صورت زیر است:

(هم نیست از کجا می‌دانستند)

P_1	10
P_2	4
P_3	18

15

حالت فعلی در زمان T

P_1	5
P_2	2
P_3	2

در این لحظه

این حالت بد است یا خوب؟ یعنی آیا احتمال دارد ما را به سمت بن بست ببرد یا نه؟

امن (safe)

93 مصرفی

3 موجودی

Subject:
Year. Month. Date. ( )

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

- با انتخاب P_2 به آخر می‌رسد.

حالا که تمام شد موجودی 5

P_1 را با 5 تا می‌توان به آخر رساند انتخاب P_1 به آخر می‌رسد

موجودی 10

P_3 هم به آخر می‌رسد

این جوری ما یک مسیر پیدا کردیم، یعنی اگر به این ترتیب اجرا کنیم به آخر می‌رسند

ترتیب امن <Safe Sequence> $\langle P_2, P_1, P_3 \rangle$

پس بله این حالت امن است.

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

P_1	5
P_2	2
P_3	3

موجودی 2

انتخاب P_2 به آخر می‌رسد

موجودی 4 ← با این موجودی P_1 یا P_3 را نمی‌توان به آخر رساند ترتیب امن وجود ندارد

پس حالت جدید امن نیست

الگوریتم بانکدار ان 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 کرده)

برای همین منبع به دزدی است چون منبع را داشت ولی از دزدی ترس ندارد ①

از کجا می‌دانیم فرآیند چقدر طول می‌کشد چند تا از یک منبع می‌خواهد حتی اگر 100 بار تکرار شده باشد اگر دزدی نمی‌کند، این هم فرق دارد ②

مجبور می‌شویم تازه دست بالا هم نگیریم ⇐

Subject:
Year. Month. Date. ( )

جلسه بیست و یکم

الگوریتم بانکدار

n فرآیند

n منبع

available [m] تعداد منابع j ام k تا وجود دارد Available [j] = k

ماتریس max [n,m] ← هر سطرش max یک فرآیند

ماتریس allocation [n,m] ← سطر i ام مربوط به فرآیند i ام ← چه مقدار از منابع استفاده کرد

← ماتریس تخصیص

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

این سه تا ماده هم در اندازه و هم در مقدار در تغییر است. فرآیند ممکن است ایجاد شوند یا از بین بروند

می‌دونیم منبع می‌خواهیم چه منابعی می‌خواهیم چه کاری می‌خواهیم

وقتی می‌گیریم request i رسیده می‌خواهیم چی کار کنیم؟

← یک بردار است از max بیشتر نباشد

Request i ≤ claim i ۱- این چیزی است که چک می‌کنیم اینست که غیرقانونی نباشد

اگر بر خلاف ادعایش درخواست غیر این صورت خطا است

2- اگر Request i ≤ Available ادعایش درست خطا ← منابع داریم بهش بدهیم یا نه (منابع در دسترس وجود دارد)

3- باید حالت جدید را شبیه کنیم ببینیم امن هست یا نه

Available = Available - Request i ← حالت جدید

Allocation i = Allocation i + Request i

claim i = claim i - Request i

اگر حالت جدید امن باشد تخصیص انجام می‌شود در غیر انتظار

چجوری بفهمیم امن هست؟

PAPCO

Subject:
Year. Month. Date. ( )

آزمایش حالت امن:

1. work = Available

finish[i] = false برای i=1,...,n

2. اگر ایستگاه‌هایی پیدا کنیم که فرآیندی که به حد مجاز می‌رسد بتوانیم آن‌ها را به انتها برسانیم

finish[i] = false

claim_i ≤ work

اگر پیدا نشد به مرحله 4 برو

3. work = work + allocation_i

جمع منابع فرآیندی که به انتها رسیده زیاد می‌شود

پس حد مجاز زیاد می‌شود

اگر به این مرحله رسیدیم یعنی فرآیند

بین مراحل 2 و 3 آنقدر می‌گردیم تا یکی را پیدا نکنیم

4. اگر برای i=1,...,n finish[i] = True باشد ← امن است

اگر حالت امن نباشد به معنای این نیست که در حالت انتظار است چون ممکن است این بن‌بست برسیم

یعنی در حالت کل منابع را دارد ولی اجازه استفاده نمی‌دهد ← کاری که به این بن‌بست نرسد

مثال) P_1 ... P_5 فرآیندها

A B C منابع

10 5 7 ← کل منابع موجود

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 + سیستم را نگاه کردیم و allocation را به این صورت دیدیم

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 برکظم allocation

=> Available = 3 3 2

2 1 0

چیزی که مصرف 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) ترسیم

available 3 2 1

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

need را نگاه می‌کنیم ← P_2 و P_4 می‌توانند به این مقدار به آخر برسند

یک ترتیب امن پیدا کنیم لزومی ندارد یکتا (unique) باشد در این جا مثلاً P_2 را انتخاب می‌کنیم و انجام 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	1 4 3
انتخاب P_3	10 4 5	10 4 5
P_1		
P_5		

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

Subject:
Year. Month. Date. ( )

2. $Request_1 = 0\ 2\ 0$ می‌شود آیا حالت جدید امن هست یا نه؟

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

available => 3 1 2

باید ببینیم این حالت امن است یا نه.

در این جا P_2 را نمی‌توانیم 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 امن هست. اگر پیدا نمی‌کردیم امن نبود.

اگر بخواهیم ترتیبی انتخاب کنیم باید در Safe Sequence باشد. در غیر این صورت به حالت بن بست رسیده‌ایم.

تشخیص بن بست Deadlock Detection

الگوریتم قبلی باید برای هر request باید اجرا شود. ولی الگوریتم sequest باید اجرا شود که این کم وقت پردازنده را می‌گیرد.

این الگوریتم هزینه‌ی قابل اجراست و هر وقت اجرا کنی می‌گوید بن بست داریم یا نه. مثلاً هر 50s که بینیم کسی در بن بست نیست. از اطلاعات موجود استفاده می‌کند

خیلی شبیه الگوریتم قبلی است.

1. work = available

اگر allocation ≠ 0 Finish[i] = false

وگرنه Finish[i] = True

از n متغیر باینری داریم که یعنی که allocation صفر دارند به حساب نمی‌آوریم (زمانی که برای اعضا حافظه دارند استفاده می‌کند)

یعنی هیچ اهمیتی ندارد

Subject:
Year. Month. Date. ( )

منابعی که برای شرط اعضاء متقابل دارند در این بست وضعیت دارند ← در اصل منابعی را محاسبه می‌کنیم که شرط اعضاء متقابل نداشته باشند

2. i را به گره‌ای پیدا کن که Finish[i] = false ← یک چیزی پیدا کن که منابعی که خواست از وجودی کمتر باشد

request_i ≤ work

وگرنه به مرحله 4

3. work = work + allocation_i ← فرآیند را انتخاب می‌کنیم و به آخر می‌رسانیم

Finish[i] = True

به مرحله 2 برو

4. به همه Finish ها True هست ← این بست نیست

اگر ... ... نشدند ← این بست هست

اگر Finish[i] = True برای n و ... و 1 ← این بست نداریم

این الگوریتم می‌گوید یک‌سری ها در این بست هست

← کافی است request را به این بست بیاوری تا آن موقع نمی‌فهمیم

مثال) P_1 ... P_5 فرآیندها

A B C ← منابع

7 2 6

می‌توان request داشته باشیم که به آن برخورد نداریم

	allocation A	allocation B	allocation C	request A	request B	request 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			

← کاری کاندید request ندارد

available 0 0 0 ← هرچی داشتیم تحویل داریم

Subject:
Year. Month. Date. ( )

در این سبت هستیم یا نیستیم؟

انتخاب P_1 یا P_3 چون هر دو صفر هستند

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

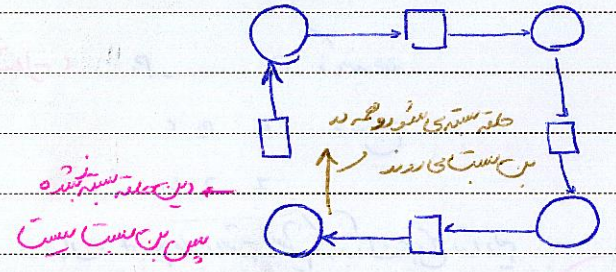
پس System در این سبت نیست، چون همگی True شدند

$Request_3 = 0\ 0\ 1$ می‌رسد ← از سیستم تصادفی ۱ جدا از نفع P که مانده

ماتریس request در P_3 0 0 1

انتخاب P_1 0 1 0 ← با این چیزی که داریم نمی‌توانیم به آخر برسانیم پس سبت می‌رسیم

← P_2، P_3، P_4، P_5 ← در این سبت هستند



می‌دانیم در این سبت وجود دارد که یکی در این سبت است می‌توانیم در این اطلاعات برای شکستن حلقه استفاده کنیم

← سیستم خلوت ← ready Queue ← خلوت‌تر است

" شلوغ ← " " ← دارای request زیادتر است

اگر بخواهیم این سبت را کدام باید بیشتر اجرا کنیم؟

در سیستم خلوت، در جایی که شلوغ است اصلاً وقت ندارد سرش را بخاراند

← شاید دلیل خلوت شدن این سبت باشد پس منطقی است که این جا بیشتر اجرا شود

Subject:
Year. Month. Date. ( )

جلسه بیست و یکم 87/9/12

تمرین OS برای هفته آینده

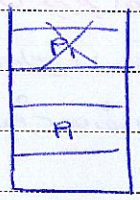
مدیریت حافظه: memory management

← مدیریت فرآیندها بین بست ← آزمون مرحله دوم ← پنج شنبه 9/28 ← ساعت 8

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

← کارهای مدیریت حافظه

جابه‌جایی relocation



هر فرآیند را می‌توانیم در هر جای حافظه به آزادی قرار دهیم

← کد، Data، پشته، PCB ..... ← یعنی همه اینها رفته داخل حافظه

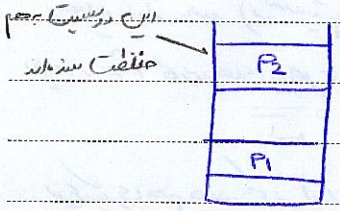
این جا در حافظه قابل جابه‌جایی هستند چون در هر جای حافظه‌ای توانیم قرار دهیم

وقتی همه چیز را می‌شود جابه‌جا کرد چون ممکن است کدها آدرس دقیق بکار ببرند باشیم و در برنامه‌نویسی می‌خواهیم بجای از

مقادیر نسبی استفاده کنیم (نسبت به Base address)

تا بجای جای اول ← از مقادیر مطلق آدرس استفاده نشود

حفاظت protection



فرآیندها نباید به فضاهای یکدیگر دسترسی داشته باشند پس

این محل‌ها حفاظت کنند

اگر این‌ها بخواهند که مشترک استفاده کنند امکانش هست

می‌توانند از بعضی چیزها که همه چیز را بطور مشترک استفاده کنند ولی محافظت شده

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

← حافظه دارای دو سازمان ← logical organization

physical "

Subject:
Year. Month. Date. ( )

باید این دو تفاوت قائل شویم

در تخصیص حافظه ممکن است پیوسته نباشد و هر بخش فرآیند یکی باشد ← فیزیکی

ولی رسیدن منطقی به نظر می‌رسد که پشت سر هم است

physic ← چیزی که واقعاً هست

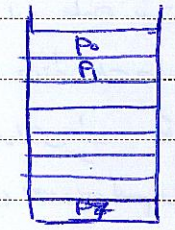
logical ← چیزی که به نظر می‌آید که هست

روش‌های مدیریت حافظه

۱. روش ایستا (static)

partitioning

در این روش حافظه از قبل تقسیم‌بندی شده ← بخش‌بندی ثابت



→ هر بخش حافظه مساوی هستند و هر کدام می‌توانیم 8

فرآیند داشته باشیم که فضاهایش هم محدود است

این روش خیلی محدودکننده ولی خیلی ساده است

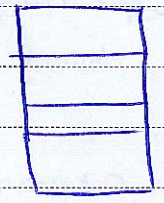
در سیستم‌هایی که کاربر محدودی دارند می‌توان از این استفاده کرد چون ممکن برنامه‌ها زیاد باشند یا اینکه یک برنامه خیلی بزرگ باشد (در سیستم‌های general purpose)

این روش کاربردهای خاص دارد که تعدادشان هم زیاد هست ← روش خیلی ساده مدیریتی دارد ← 8 تا دارد

این فضا حل می‌شود

← embeded (سیستم‌هایی که کاربرد دارند)

نوع دیگری هم هست که اندازه partition ها ثابت است و اندازه‌هایشان به هم فرق دارد



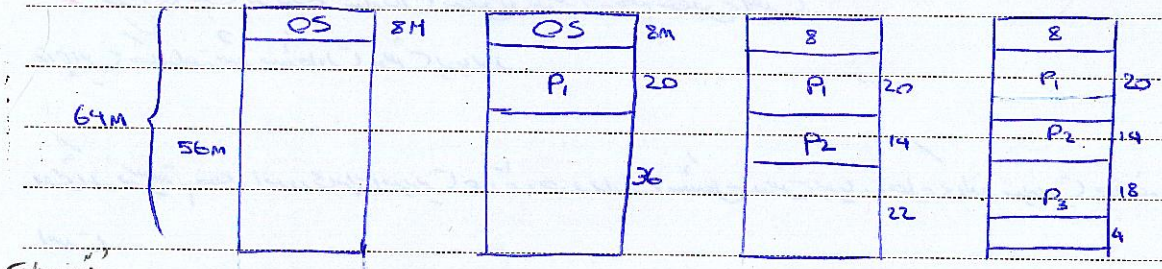
بخش‌بندی ثابت

اندازه‌های متفاوت

Subject:
Year. Month. Date. ( )

2. روش پویا Dynamic

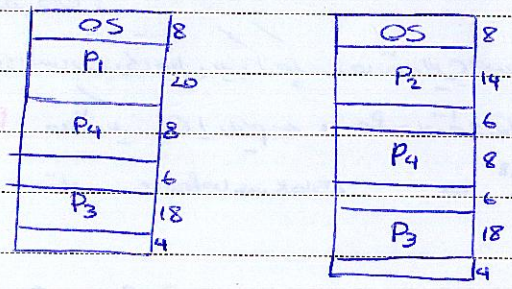
بخش بندی حافظه متغیر ← فضای حافظه یک فضای یکپارچه است که به هر صورت می‌توان آن را بخش بندی کرد



مثلاً فرض می‌کنیم

افزایش زمان →

در این جا فرایند P_4 می‌آید فضای 8M می‌خواهد ← نداریم پس P_4 منتظر می‌شود ← پس از مدتی فرایند P_2 را خلوت می‌کنیم، فضای P_2 خالی می‌شود، حالا می‌خواهد P_2 دوباره وارد شود ولی می‌دانیم که P_1 تمام شده و فضایش آزاد است

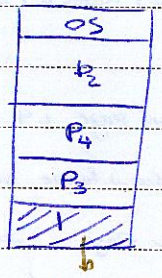


الان اگر فرآیندی بیاید که 14M فضای حافظه بخواهد ← حافظه نداریم (پشت سر هم) ← ولی داریم که پیوسته نیست ولی می‌توانیم به آن برسیم

این پدیده که هر چی جلو برود باعث می‌شود حافظه بیشتر تکه تکه شود fragmentation → fragmentation می‌گویند

↳ فضای حافظه داریم ولی در دسترس نمی‌باشد

برای حل این مشکل ← هر چند وقت یکبار عمل فشرده سازی انجام بدهد ← یعنی همه را به یک سمت منتقل کند ← تا فضای پیوسته آزاد شود compaction



این کار در سیستم انجام می‌شود ولی این جا اولویت ندارد ولی این جا در سیستم اولویت دارد

↳ به عنوان مثال یک مشکل بزرگ که اگر پشت هم باشند زمان زیادی می‌گیرد

→ فضای خالی یکپارچه می‌شود

internal fragmentation

هزینه زیاد است ← مقدار زیادی از اطلاعات که جای به جای یکدیگر منتقل کنیم

↓

زمان زیاد (زمان انتقال)

Subject:
Year. Month. Date. ( )

← Fragmentation ← در سرعت دسترسی تاثیر دارد چون فضاهای پشت سر هم قرار می‌دهد هر چی فاصله فضاها در دسترسی زیاد باشد سرعت کندتر می‌شود ولی در حافظه تاثیری ندارد چون عضوی دسترسی است

↳ پس در حافظه کسی از این روش استفاده نمی‌کند

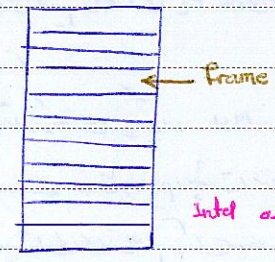
* در این بحث که گفتیم سازمان منطقی و فیزیکی بر هم منطبق است تمام این حالات گفته شد مشکلات روش پویا بود.

روشی که در سیستم‌های امروزی برای مدیریت حافظه وجود دارد و مشکل روش پویا را حل می‌کند مدیریت حافظه paging است.

← بهترین روش مدیریت حافظه

مدیریت حافظه صفحه‌بندی Paging

↳ از نوع روش‌های پویا است



حافظه از نظر OS:

تعداد خانه‌های حسابی ولی خیلی ریز و کوچک (حافظه به این صورت تقسیم‌بندی شده)

مثال frame: حافظه یک 1G داریم ← frame حافظه می‌تواند 4K باشد

Intel ← مربوط به پردازنده

$\frac{2^{30}}{2^{12}} = 2^{18} \Rightarrow$ 256k frame تعداد

اندازه frame ها از پیش تعیین شده است.

↳ نسبت به کل حافظه خیلی ریز است به طوری که در آن جمع فرآیندی جا می‌شود

فرض کنیم فرآیندی داریم که فضای ۴ خواهد اگر دو را frame بگیریم و تقسیم‌بندی کنیم ← مثلاً frame ۱ تا ۴ فضای خواهد

	P_2
0	Page 0
1	" 1
2	" 2
3	" 3

→ پیوسته (فضای پیوسته)

پس فرآیند ما ۴ تا page است ← می‌خواهیم این را بیاوریم در حافظه ← لزومی ندارد حتماً frame پیوسته پیدا کنیم

بلکه هر frame که جای بود یک page می‌تواند قرار گیرد

هر page هر frame می‌تواند قرار گیرد

پس یک نیازی به fragmentation نداریم

پیوستگی

P_y
Page 0
Page 1
Page 2

2 Py
page 0 P_y
1 Py
Page 1 P_y
0 Py
Page 2 P_y
Page 3 P_y

PAPCO

Subject:
Year. Month. Date. ( )

در این جا مسئله پیوستگی فضا از بین رفت پس دچار مشکل روش پویا می‌شود هر جا frame های بیدی‌ای برای آن استفاده کنی

← پیوستگی را از بین بردیم پس در حین اجرا جابجا می‌شود از Page به هم Jump می‌کند؟

سوال‌های پیش می‌آید

← اختصاص دادن frame

← آزاد کردن frame

← وضعیت برگرداندن حافظه؟ اگر کل حافظه اشغال شده یک فرآیند آمد چکار باید بکنیم

می‌توانیم بخش‌هایی از فرآیندی که باید اجرا می‌شود که الان به دردمان نمی‌خورد از حافظه خارج کنیم

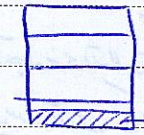
← آیا لازم است کل فرآیند در حافظه باشد ← الان فرض می‌کنیم که کل فرآیند در حافظه است اما بعداً این فرض را بهم می‌زنیم

Fragmentation

خود paging باعث Fragmentation ایجاد می‌کند

هر فرآیند برای اتصال به page تقسیم کردن لزوماً page آخر همش استفاده شده نیست، دارای بخش خالی باشد یعنی page آخر خالی داشته باشد

این فضاهای خالی که Fragmentation است قابل استفاده نیست به هیچ وجه برای دیگران مقدارش خیلی کم است پس نمی‌گوییم اشکال ندارد



مواظب فرآیند اتفاق افتاده

internal fragmentation

← OS از کجا بداند هر frame حال کی هست ← یکجایی باید این را ثبت کند ← OS برای هر فرآیند یک جدول صفحه درست می‌کند و در این می‌گذارد که page های کجاست؟

جدول صفحه P_2

Page	frame
0	3
1	6
2	8
3	9

page 0 من کجاست

P_1

0	6
1	4
2	5

پس همه هر page کجاست

Subject:
Year. Month. Date. ( )

حالا سیستم عامل از کجا می‌فهمد این جدول کجاست ← pointer جدول را در PCB نگه می‌دارد
آدرس شروع
برای حافظه آدرس شروع هم است، ممکن است کل جدول در حافظه نباشد

OS همیشه می‌داند که کدام frame ها در حافظه خالی هستند.

جلسه بیست و هفتم 87/9/17

امتحان یکشنبه 87/10/1 ساعت 12:30 تا 13:30

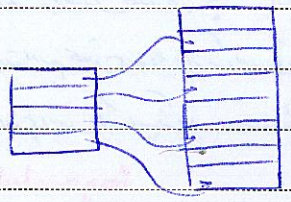
مدیریت حافظه صفحه‌بندی Paging

حافظه frame ها تقسیم شده

فرایندها را به page ها تقسیم می‌کنیم

که اندازه frame و page ها برابر می‌باشد

برای اینکه بدانیم page ها کجاست از یک جدول استفاده می‌کنیم



P	f
0	f_0
1	f_1
2	f_2
3	

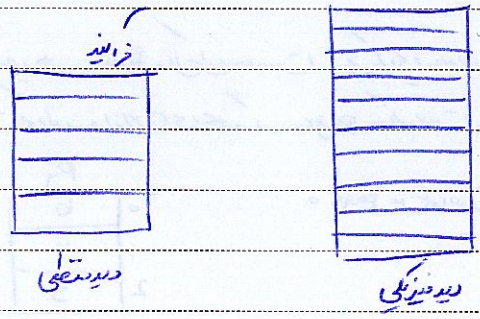
← جدول صفحه

ترتیب frame ها مهم نیست ← مشکل فضای پیوسته وجود ندارد

اگر frame ها تمام شدند با یک سری راه‌حل کنیم

پیوستگی از بین رفته ← چگونه می‌توان فرایند را اجرا کرد یا Data ها را پیوسته دید

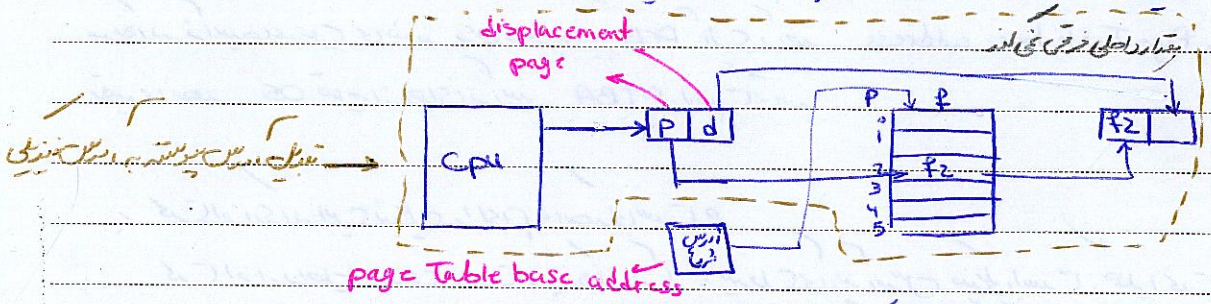
پردازنده این ها را به پیوسته تبدیل می‌کند ← کارکرد افزاری



دید منطقی به منطقی تبدیل شود
یعنی آدرس منطقی به آدرس فیزیکی

Subject:
Year. Month. Date. ( )

این کار یک عمل سخت افزاری است و اگر نرم افزاری باشد خیلی سرعت برنامه کم می‌شود.



← CPU در اجرای برنامه‌ها آدرس تولید می‌کند به حافظه می‌دهد

Instruction ← مربوط به

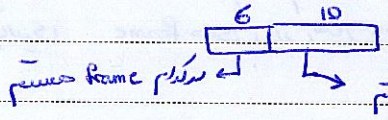
Data ← مربوط به

فرض می‌کنیم آدرسی که از پردازنده می‌آید منطقی است ← پیوسته

در عمل CPU آدرس غیرخطی تولید می‌کند

برای حافظه 64K ← 16 بایت آدرس داریم.

16 frame => 1K ... 64 frame داریم



هر آدرسی که تولید می‌کنیم می‌خواهیم در کدام frame می‌افتد ← هر آدرس

به آدرس ← به طور کلی نگاه می‌کنیم یعنی 16 بایت را می‌بینیم

به قسمتی نگاه می‌کنیم یک 16 بایت یک 10 بیت ← به این صورت نگاه می‌کنیم

کدام page ← کجای page

باید page کجای حافظه است ← از جدول صفحه مربوط به فرآیند استفاده می‌کنیم

صفحه پیوسته است ولی در حافظه پیوسته نیستند

آدرس جایی که می‌بینیم پیوسته است ولی چیزی که به حافظه داده می‌شود

ناپیوسته است.

OS کمک افزار را آماده می‌کرد خودش کاری نداشت انجام همین کار را کرده در عین اما حضور ندارد ولی کار خود را انجام داده و تایید خود را نداشته

← این کار کاملاً مخفی است و هیچ کس نمی‌بیند ولی OS می‌داند جدول صفحه خودش جدول صفحه را درست کرده ولی در عین

اجرای فرآیند OS وجود ندارد.

Subject:
Year. Month. Date. ( )

point: هر فرآیند جدول مخصوص خود را دارد و فرآیندها به همدیگر دسترسی ندارند
موقعی که فرآیندها عوض می‌شوند OS با توجه به PCB آن فرآیند Page Table base address را
تغییر می‌دهد OS قبل از اجرای فرآیند PTBA را ست می‌کند

← سخت افزاری که این تبدیل را انجام می‌دهد کجاست؟
سخت افزار داخل CPU است و چیزی که از آن بیرون می‌آید آدرس فیزیکی است، ولی سخت افزارها
داخل CPU هستند ولی 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 شده:
- دیگر Fragmentation نداریم (داریم ولی بسیار کمتر و در آخرین Page هر فرآیند)
- مشکل پیدا کردن فضای حافظه را نداریم

مزایا:
- قابلیت انتخاب هر Frame آزاد شده
- جابجایی Frame های آزاد آسان شده ← به وسیله صفر یک کردن خانه‌ها مثلاً اگر حافظه 64K باشد یک رشته 64 تایی می‌تواند این کار را انجام دهد.

اندازه Page از کجا آمده؟ دو نکته باید در نظر بگیریم
1. اندازه جدول صفحه ← اندازه page کوچک باشد جدول صفحه خیلی بزرگ می‌شود
2. تکه تکه شدن یا Fragmentation ← بخشی از حافظه از دست می‌رود و این مقدار خیلی قابل توجه می‌شود

بعد از حل این مشکلات یک مشکل جدید بوجود می‌آید: قبلاً مشکل مدیریت حافظه داشتیم به ازای هر آدرس
یک بار به حافظه می‌رفتیم ولی الان به ازای هر آدرس دو بار باید برویم ← یکی برای page Table
یکی برای آدرس اصلی

این مسئله خیلی سرعت را پایین می‌آورد ← مشکل حافظه خاموش است CPU است و سرعت ما کم می‌کند
این سیستم برای سیاست پایین دارد
performance

PAPCO

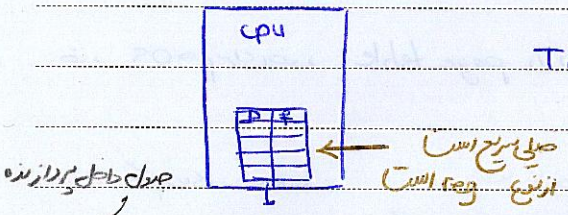
Subject:
Year. Month. Date. ( )

سیستم حافظه مدیریت حافظه خیلی مهم نیست، سرعت است که تعیین کننده است
بیایین مشکل را حل کنیم:

حل مشکل سرعت پایین:

آدرس هم مستقیم به حافظه نمی‌رود بلکه اول به سراغ cache می‌رود و در صورتی سراغ حافظه می‌رود که در cache نباشد
از همین ترفند بخشی از cache استفاده می‌کنیم
بخشی کوچکی از حافظه را دارد درصد $\frac{1}{50}$ یا $\frac{1}{100}$ ولی همیشه اول cache را می‌گردیم

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



اگر به جای اینکه سراغ حافظه برویم سراغ cache یا جدول TLB برویم سرعت ما خوب می‌شود سرعت ما درست مثل reg
مشکلش اینست که اندازه‌اش کوچک است

TLB: Translation Lookaside buffer (بافر ترجمه)

برای cpu دم دستی است.

در پردازنده Intel ← 32 تا شکاف دارد ولی یک زیر مجموعه می‌تواند 1000 page table داشته باشد
در TBL ← 98% آدرس‌ها جواب می‌دهد و تنها در 2% مجبور است سراغ page table برود
اکثر دستورها با سرعت بالایی انجام می‌شود
TLB به OS هیچ ربطی ندارد و OS اصلاً نمی‌فهمد که TBL اصلاً وجود دارد و با جدول صفحه کار دارد
همگی در سخت افزار انجام می‌شود

برای تبدیل آدرس (سخت افزار)

1. آزمایش TLB ← اگر موجود بود تبدیل انجام می‌شود
2. اگر موجود نبود جدول صفحه و update کردن TLB ← یعنی اینجا هم اتلاف پیدا کردن می‌کند در TLB

Subject:
Year. Month. Date. ( )

← TLB از دید OS مخفی است ← سخت افزاری هستند

← Paging از دید برنامه کاربری مخفی است ← OS می‌بیند

یک نکته: فرض کنیم تا الان P_2 کار می‌کرده الان P_1 به کار بیفتد (هم‌زمان شروع شود) داخل TLB چی هست؟

هیچ چیز داخلش بود مربوط به P_2 است دفعه پاک می‌شود و اولین بار به هر آدرسی که ارجاع می‌کنیم در TLB نیست ری‌هیت نیست چون این آدرس داخل PLB قرار گرفته

← با شروع هر فرآیند (شروع هم‌زمانی) TLB خالی است، TLB به تدریج پر می‌شود.

↳ مجبوریم کار خود را از طریق جدول صفحه انجام دهیم

به هر خانه‌ای در جدول صفحه مراجعه کنیم که در TLB نباشد ← می‌آید داخل TLB

خود OS هم برای خود page table دارد ولی مجدداً اولین چیزی که در حافظه قرار گیرد پیوسته است.

مثال) هم‌زمانی 10ms

6 billion Instruction / sec ← یک بیلیون دستورالعمل در ثانیه انجام می‌شود

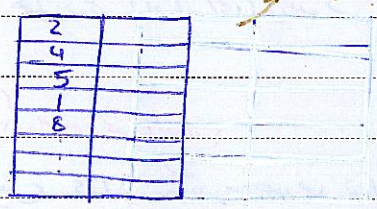
$\frac{10^9}{10^2} = 10^7$ ← در هر هم‌زمانی 10^7 دستور Instruction اجرا می‌شود

↳ تعداد مراجعات از این بیشتر است چون ممکن است بیش از یک دستور داشته باشد

TLB در کامپیوترهای بدون قبضه چگونه است؟

این نوع ذخیره‌سازی در هم‌ریخته data چه هستند

content Addressable Memory → مثل cache



اطلاعات در TLB را چگونه پیدا کنیم؟

به ترتیب نیست چون تمام pageهایی که به آن دسترسی پیدا کردیم در TLB است

هیچ راهی جز این نیست که تمام TLB را چک کنیم به ازای هر دسترسی باید کل TLB را چک کنیم ← 32 تا مقایسه کننده داشته باشیم (سخت افزاری) باهم موازی کار کنند

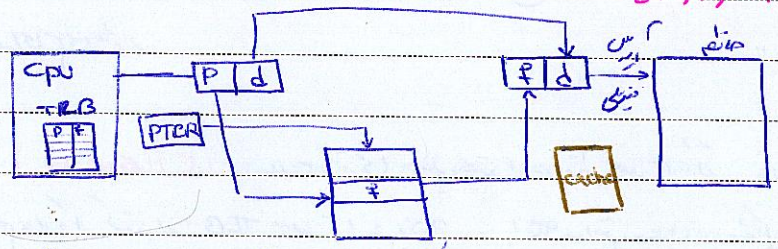
↳ به تعداد entry

↳ یکی از دلایل کم بودن خانه‌ها همین است.

P4PCO

Subject:
Year. Month. Date. ( )

جلسه بیست و ششم 87,9,24



یکی از reg های اضافی برداشته که از PCB، load می‌شود که این عمل صفحه‌گذاری است. چون مشکل سرعت پایین را حل کنیم 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 همیشه دسترسی به حافظه، زمان یکسان ندارد ← بعضی وقت‌ها از TLB بعضی وقت‌ها از طریق جدول صفحه

$T_{eff} = 0.98(2ns + 50ns) + 0.02(2ns + 50ns + 50ns)$

اول برای TLB بررسی می‌شود ← جدول صفحه ← خود حافظه

$T_{eff} = 51ns + 2ns = 53ns$

اگر تنها یک دسترسی به حافظه داشتیم 50ns وقت می‌خواستیم ولی الان فقط 3ns اضافه شده که کمتر از 10% است و این خوب است تا 10% هم خوب است ولی 10% را خطا در نظر می‌گیرد

← معمولاً سیستم‌های واقعی این طور نیستند. حافظه خیلی کند است و باید راهی پیدا کنیم این را تندتر کنیم (50ns در مقابل 2ns خیلی زیاد است، سرعت حافظه بسیار کند است)

برای حل این مشکل از حافظه کمکی یا cache استفاده می‌کنیم. cache سرعت بالایی دارد به قیمت خرج کردن

که اصلاً با حافظه قابل مقایسه نیست. بخشی از حافظه را در cache قرار می‌دهیم (به صورت بخش حافظه فایل نداریم

که خاصه کد، data، Table ... باشد) مبنای کار ما حافظه وصل است و امید داریم بیشتر وقت‌ها آن چیزی که می‌خواهیم

در cache پیدا کنیم.

Subject:
Year. Month. Date. ( )

← ساختار cache ، بحث افزاری است و OS از وجودش بی اطلاع است و OS با حافظه کاری کند و تمام مدیریت های روی حافظه انجام می شود

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

نسبت اصابت هم داریم که کمتر از TLB است : 90% ← 90% مراجع در cache مشکل حل می شود.

برای کمک حافظه در بحث مدیریت حافظه ندارد ← در اصل بهتر از این است.

cache با حافظه در نظر گرفته می شود و در اصل می خواهیم بگوییم 50ns به چی تبدیل می شود.

حل مثال قبل با استفاده از cache

زمان دسترسی به حافظه — خیلی تقریبی

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

زمان دقیق از این بیشتر است و ما خیلی چیزها را در نظر نگرفتیم

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

تاثیر cache در جایی است که به حافظه دسترسی داریم. عدد دقیق از 17.3ns بهتر است. چون بعضی چیزها را در نظر نگرفتیم:

ممکن است در cache چیزی را پیدا نکنیم می رویم سراغ حافظه این طور نیست بلکه اگر در cache پیدا نکردیم از حافظه به cache می آید و در اصل ما ارتباط بین cache و حافظه اصلی را در نظر نگرفتیم که این زمان بر است. CPU با حافظه رابطه مستقیم ندارد و از cache با حافظه ارتباط دارد.

cache سطوح دو level دارد ← L_1 cache داخل CPU ← سرعت خیلی بالاتر و ظرفیت خیلی کوچک

L_2 cache روی board اصلی

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

مدیریت حافظه به روش paging بحث مشکلات Fragmentation و مدیریت آن را حل کنیم

← همه فضاهای خالی را OS باید حساب و کتاب نگه دارد و اشتراک باشد.

S_1 b_1, l_1 — S_2 b_2, l_2

base — طول فضای خالی — آدرس شروع

S_3 b_3, l_3

P_4 وارد می شود و فضای خواهد و بیشتر از یک choice داریم

PAPCO

Subject:
Year. Month. Date. ( )

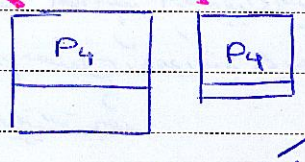
می‌توانیم جدول تا از این فضاهای خالی را به آن بدهیم، فضاهای خالی را باید search کنیم ببینیم کدام فضای خالی از همه به P_4 نزدیک‌تر است که برای این نگاه بدهید کل فضاهای خالی را search کنیم که این search خیلی طول می‌کشد ← روش best fit (بهترین برازش) search طولانی و بهترین انتخاب

روش دیگر اینست که به جای search کردن به اولین جایی که رسیدیم که P_4 می‌تواند در آن قرار گیرد آن را انتخاب می‌کنیم با این نحو search را کوتاه کردیم first fit (اولین برازش)

جستجوی کوتاه بدون بهترین انتخاب

روش دیگری هم هست ← لیست‌ترین را پیدا می‌کند search می‌خواهد جدول بزرگترین را پیدا می‌کنیم و یعنی اینکه در list، sort کردیم ← روش worst fit بدون جستجو

worst fit best fit



منطق worst fit اینست: اگر بزرگترین قطعه را برای قرار دادن فرایند استفاده کنیم فضایی که باقی می‌ماند هم به اندازه‌ی کافی بزرگ است که بتوان برای فرایند دیگر دوباره آن را استفاده کرد

روش اثبات اینها از benchmarking است ← روش و تئوری‌ای تایید کننده است.

مجموعه‌ای از برنامه‌های واقعی که زیاد استفاده می‌شود.

روی این برنامه‌ها چیزی که می‌خواهیم ثابت کنیم اجرا می‌کنیم و اندازه‌گیری می‌کنیم و ببینیم کدام یک از حافظه بهتر می‌تواند استفاده کند ← مقدار جستجو

مقدار هدر رفته حافظه ← بیش از $1/3$ حافظه هدر می‌رود.

با این روش مجموعه best & first بهتر از worst کار می‌کنند

$1/3$ حافظه تلف می‌شود

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

← paging نتها مسئله fragmentation را حل می‌کند بلکه مدیریت پیچیده فضاهای بیشتر را حل می‌کند و یک مدیریت ساده می‌آورد. یعنی ما دیگر فضای بیشتری می‌خواهیم

مشکل paging که برعکس بود با ترفندهایی حل کردیم و مدیریت غالب در سیستم‌های امروزی است

مدیریتی که الان گفته می‌شود که در برازنده Intel از آن استفاده شده مدیریت حافظه به برازنده حسیبیده می‌شود چون تمام تبدیلات را CPU باید انجام دهد که باید پشتیبانی CPU را مدیریت حافظه داشته باشد ← segmentation

PAPCO

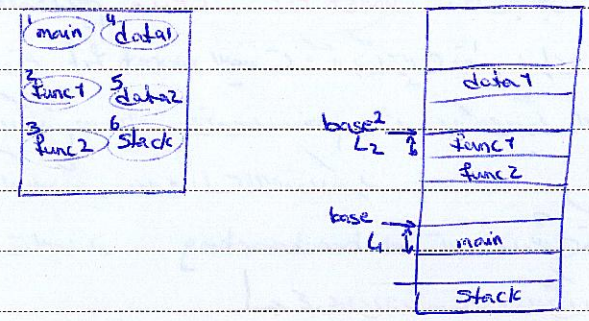
Subject:
Year. Month. Date. ( )

جلسه بیست و پنجم 9,26

مدیریت حافظه قطعه بندی Segmentation

این مدیریت مخصوص OS است و در جدول OS این کار را انجام می‌دهد. وقتی جدول باید پشتیبانی پردازنده را داشته باشد می‌گوییم CPU این اشکال را دارد.

← صفحه بندی از دیدگاه برنامه کاربری مخفی است ولی برعکس قطعه بندی با دیدگاه برنامه کاربری انجام می‌شود همین دیدگاه را می‌توان در مدیریت حافظه به کار برد می‌توان این برنامه را چند قطعه تقسیم کرد



یکی دیگر از اشکال های پشت سر هم پیدا می‌کنیم که هر کدام مستقل نگاه می‌کنیم ولی هر قطعه باید پیوسته باشد

این نوع مدیریت همچنان شکل fragmentation را دارد ولی کمتر شده چون در این جا فضاهای کوچک نیاز داریم

جدول قطعه segment table

1	base1	L1
2	base2	L2
3		
4		
5		
6		

OS از روی Seg Table می‌فهمد این قطعه‌ها کجا هستند. هر قطعه یک آدرس base و یک طول دارد در این جا می‌توانیم از شماره گذاری قطعه‌ها استفاده کنیم چون طول قطعه‌ها ثابت نیست

این مدیریت نسبت به paging سخت‌تر است. باید حساب کتاب فضاهای خالی را داشته باشیم و باید برای پیدا کردن فضای خالی سرچ کنیم

در پردازنده 8086 ← پدر بزرگ پردازنده Intel

نقطه سیستم Segmentation است ولی خیلی ساده‌تر باید مشخص کنیم که کجا code seg، stack seg، data seg ... است و این را ما تعریف می‌کنیم ← این خیلی نوع ساده‌ای است چون Seg های زیادی دارد.

این نوع کامل‌تر اینست که بتوانیم هر چند تا قطعه که می‌خواهیم داشته باشیم.

Subject:
Year. Month. Date. ( )

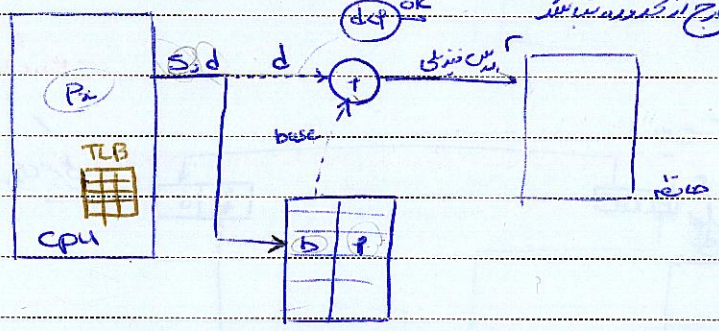
تبدیل آدرس به Segmentation

آدرس به دو بخش تقسیم شده که کدام Seg ← این Seg را برنامه کاربری می‌گوید

داخل این Seg کجاست.

در غیر این صورت می‌گوید خارج از محدوده‌ای که تعریف تولید می‌کند

قبل از جمع d را باید چک کنیم خارج از محدوده نباشد



تقریباً شبیه paging است ولی در این جا یک adder یک مقایسه‌کننده اضافه کنیم.

این روش عام مشکلات را دارد. Intel برای حفظ این ساختار و استفاده از paging این دو را با هم ترکیب کرده یعنی به جای page کردن آدرس‌ها، Seg ها page می‌کند و هر Seg را به صورت page در حافظه ذخیره می‌کند

TLB سه field دارد ← شماره Seg

base

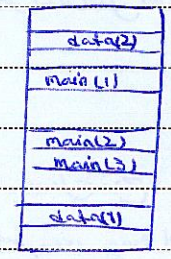
limit

پیجینگ paged Segmentation (ترکیب مدیریت قطعه‌بندی و صفحه‌بندی)

← مدیریت حافظه قطعه‌بندی صفحه‌بندی شده

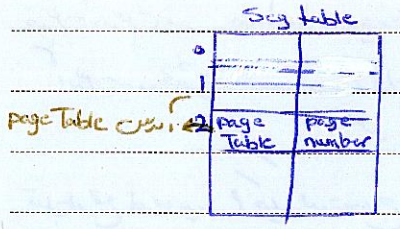
هر Seg را به طور مستقل صفحه‌بندی می‌کنیم

Seg توسط برنامه کاربری انجام می‌شود ولی paging به هم از دیدگاه کاربر مخفی است



دیگه Seg نداریم یک آدرس به صورت page در حافظه قرار می‌گیرد

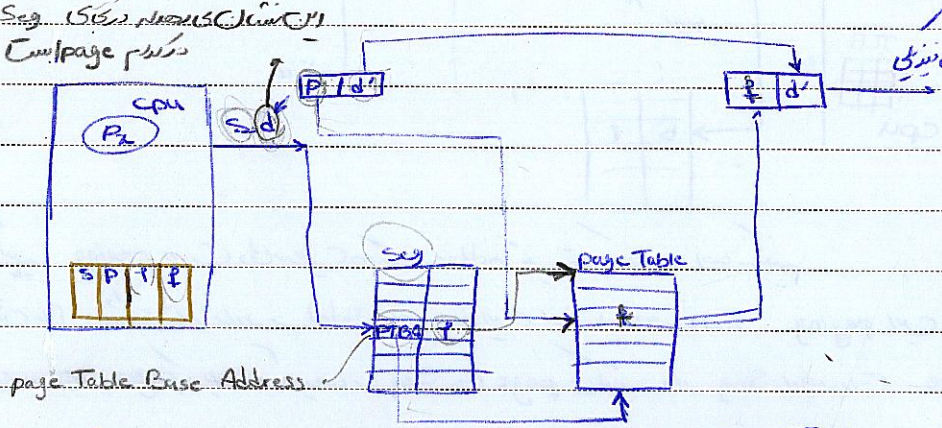
Subject:
Year. Month. Date. ( )



برای هر Seg در Seg Table باید نشان دهیم که page Table آن کجاست

مزایای سیستم آدرس‌دهی

این نشان می‌دهد برای Seg ام چی در کدام page است



S ← می‌گوییم به کدام Page Table بریم

باید طول را هم چک کنیم که limit را رد نکند

در این جا برای یک فرآیند چندین page Table داریم که هر کدام برای یک Seg است ولی اگر paging خالی رشتیم برای کل فرآیند page Table رشتیم

در این جا مسئله مشترکی که حافظه بیشتر می‌شد ← Seg table

page " ↙

حافظه

برای حل از روش‌های قبل استفاده می‌کنیم ← استفاده از TLB

Seg page f

برای سرعت دسترسی می‌کنیم

کدام page مربوط به این Seg

در TLB باید دنبال زوج (seg, page) بگردیم. ← قبلاً متفاوت page جستجو می‌کردیم

در واقع تعداد صفحه‌های Seg است.

← حافظه‌ش متغیر بودن است و هر چی تواند تغییر کند

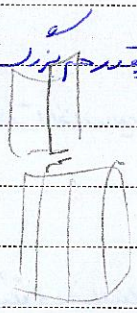
PAPCO

Subject:
Year. Month. Date. ( )

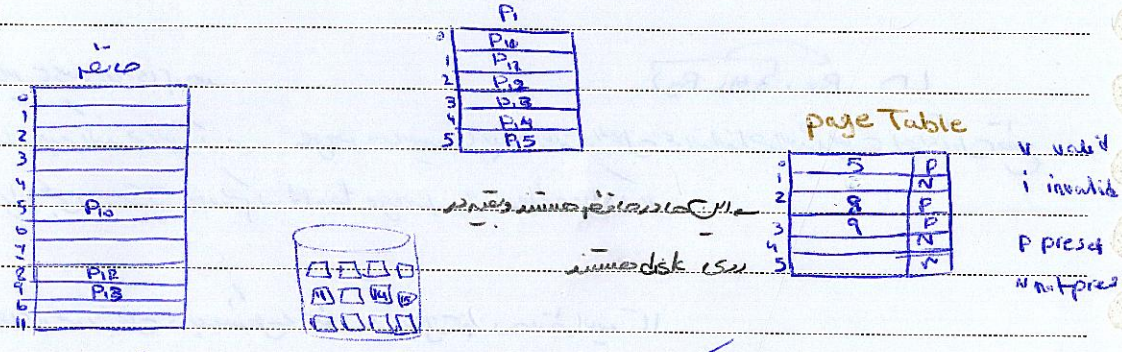
پردازنده Intel علاوه بر این اجازه‌ای دهد که فقط از paging استفاده کنیم به این ترتیب که کل برنامه را یک قطعه تعریف کنیم ولی در اصل بر هم از قطعه‌بندی استفاده می‌کنیم.

← آخرین جواب همیشه چیزی که پیاده می‌کنیم paging است و خیلی از مشکلات را حل می‌کند

حافظه مجازی: Virtual Memory



فعلاً فرض کردیم برای اجرای فرآیند باید تمام فرآیند داخل حافظه باشد (فرض محدودکننده‌ای است و چه بسا حجم بزرگ باشد 2 فرآیند بزرگ می‌توان آنرا پر کند در نتیجه حداکثر دو تا فرآیند با هم اجرا می‌شدند)
حافظه مجازی این فرض را نقض می‌کند و می‌گوید لازم نیست کل فرآیند در حافظه باشد



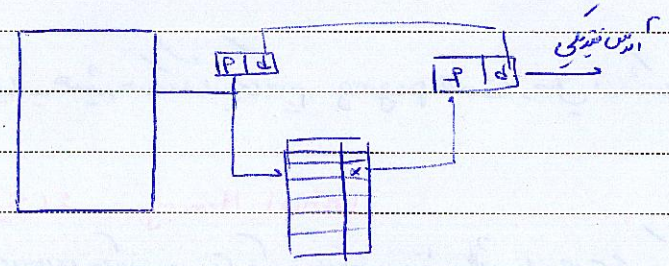
اگر با توجه به اصل locality قطعه‌ای قطعه‌ای برنامه اجرا می‌شود، هر لحظه هر چند تا page که لازم می‌شود داخل حافظه می‌آوریم تا وقتی به بقیه احتیاج نکرده آن‌ها را وارد حافظه نمی‌کنیم یعنی در اصل بخش‌هایی از فرآیند در حافظه باشند که به آن‌ها لازم داریم به این منطق خیلی فرآیند می‌توانیم با هم اجرا کنیم

← page Table یک Field جدید برای این page Table اضافه می‌کنیم تا ببینیم محتوای کدام خانه page valid است یعنی در دیسک disk نیست چون حافظه نمی‌تواند خالی باشد.
به هر حال جدول صفحه ساخته می‌شود (توسط OS) تا مشخص می‌کند که کدام قسمت‌های آن معتبر و کدام‌ها نامعتبر هستند (یعنی در اصل disk را هم به حافظه اضافه کردیم)
در واقع disk می‌شود Virtual Memory چون واقعیت ندارد ولی بزرگ به نظر می‌رسد.
حافظه مجازی: حافظه بسیار بزرگی که واقعی نیست
از بخشی از disk به عنوان حافظه مجازی در نظر گرفتیم چون در disk قطعه‌ها نیستند

Subject:
Year. Month. Date. ( )

حالا تبدیل آدرس چطور انجام می‌شود؟ تبدیل آدرس توسط سخت‌افزار است

تبدیل آدرس تفاوتی با paging ندارد و فقط یک تفاوت کوچک دارد.



سخت‌افزار ببیند بیت valid توجه کند. در صورت غیر valid بودن تولید وقفه کند

Page Fault (خطای صفحه)

مثلاً می‌خواهد این دستور را اجرا دهد LD Rx, 34(Ry)

آدرس را تولید می‌کند می‌بیند که در page حافظه نیست پس دستور انجام نشده پس اجرای آن قطع می‌شود و به سراغ بخشی از OS می‌رویم که page fault را handle می‌کند

1. اجرای دستورالعمل دچار مشکل می‌شود / page در حافظه نیست!
2. تولید وقفه page fault
3. اجرای برنامه قطع می‌شود و OS کار می‌کند ← سخت‌افزار page fault در OS کار می‌کند

از روی آخرین آدرسی که انجام شده می‌فهمد در کدام page است

توسط OS انجام می‌شود:
- page را مشخص می‌کند
- از disk به حافظه منتقل می‌شود
- باید page Table را اصلاح کند
- کنترل به برنامه کاربر برای ادامه برگشت

کنترل دوباره به برنامه کاربر برمی‌گردد و دوباره دستور انجام می‌شود

← هیچ تبدیل آدرس OS نیست! یک کار اضافه است که در صورت عدم انجام تولید وقفه می‌کند

اگر حافظه جای نباشد تکلیف چیست؟ باید page از یک فرآیند را از یک frame بیرون کنیم و جایگزین کنیم

برای بیرون کردن روش‌های مختلف مطرح است

برای TLB , cache

این سوال مطرح است

PAPCO

Subject:
Year. Month. Date. ( )

B

روش بهتری که استفاده می‌شود LRU است ولی در cache از روش‌های ساده‌تر مثل FIFO استفاده می‌شود

جلسه بیست و ششم ۸۷/۹/۱

می‌توان به طور غیرواقعی نشان داد که حافظه بزرگ است.

حافظه فیزیکی $2^{32} = 4GB$ باشد ولی حافظه مجازی خیلی بزرگتر می‌تواند باشد در حدود $2^{45} = 32TB$ باشد

پس در نتیجه اگر OS خوبی داشته باشد می‌توان تعداد زیادی Task رو داشت

روش‌های جایگزینی: Replacement Policies

حافظه فیزیکی پر شود و برای گسترش آن از disk استفاده می‌کنیم اینکه چه کسی را بیرون کنیم تا فرایند جدید را جایگزین کنیم این روش‌ها هستند.

ساده‌ترین راه: کاری نداریم که چه قدر در حافظه بوده، به ترتیب بیرون می‌کنیم یعنی کسی که زودتر از همه به 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 monitor می‌کنیم تا مثال‌هایی که توضیح می‌دهیم را ببینیم

0112 0113 0114 0614 0615 0112 0113

p d

این آدرس‌های مجازی

جاهایی که page عوض می‌شود می‌خواهیم که page fault نداریم.

d را حذف می‌کنیم

01 01 01 06 06 01 01

جاهایی که اشکال page fault هست را در نظر می‌گیریم

01 06 01 ← Reference string

اول page fault است دومی نیست

PAPCO

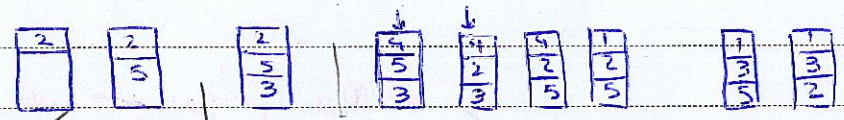
Subject:
Year. Month. Date. ( )

از این جا به بعد باید سیاست خاصی را به کار ببریم چون حافظه پر شده

مثال) یک رشته مرجع برای مثال از قبل بوده

2 5 2 3 5 4 2 5 1 2 3 2

سیاست FIFO



از قبل بوده مشکلی ایجاد نمی‌کند

آینده خبر نداریم و این خیلی بد است

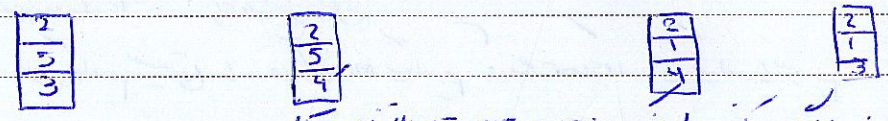
که حتی از حافظه خارج می‌شود در صورتی هنوز به آن احتیاج داریم

این روش خوب نیست چون در خیلی جاها 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 متضر آینده

بر اساس چیزی که در آینده می‌بینیم عمل می‌کنیم خیلی است ولی برای بدست آوردن تقریب می‌سبی می‌کنیم

2 5 2 3 5 4 2 5 1 2 3 2



آینده را نگاه می‌کنیم آنجا که از همه دورتر است را انتخاب می‌کنیم.

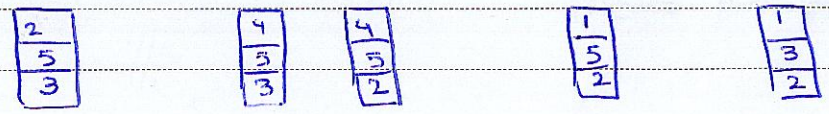
در صورتی که دو تا انتخاب داشته باشیم از روش FIFO بین آن دو استفاده می‌کنیم

برای تخمین این را می‌توان گفت که رفتار آینده مثل گذشته است و چیزی که در گذشته نزدیک اتفاق افتاده در آینده نزدیک هم اتفاق می‌افتد ـ LRU

least Recently use (کمترین استفاده اخیر)

روش LRU ـ با توجه به گذشته آینده را تخمین می‌زند

2 3 2 3 5 4 2 5 1 2 3 2



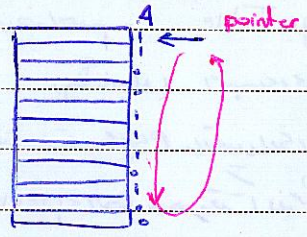
در این جا اگر ما تصمیم درست بگیریم زمانی ممکن است تصمیم غلط هم بگیرد ولی در کل بهتر از FIFO عمل می‌کند

PAPCO

Subject:
Year. Month. Date. ( )

پیاده سازی این روش آسان نیست و کی خیلی شبیه stack عمل می‌کند یک پیاده سازی سخت است و وقت گیر است این روش به همین دلیل کم از این روش استفاده می‌کنند

FIFO داشتیم FIFO هر نسبت کی چون برچ است استفاده می‌شود
optimal بهینه عمل است و سعی می‌کنیم چیزی شبیه این بدست آوریم
LRU وقت گیر است و کی خوب عمل می‌کند



روش محبوب استفاده: ساعت clocked

مختصات page Table ← Frame

valid bit

بیت A یا Accessed ← این را اضافه می‌کنیم

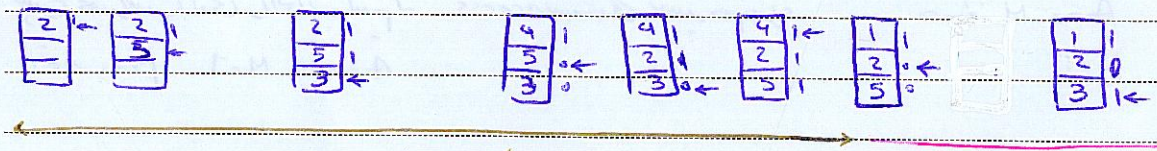
هر page که مورد دسترسی باشد بیت set می‌شود

یک pointer می‌گذاریم که جلوی برود اگر دسترسی شده باشد جلوی برود و ولش می‌کند اما قبل از اینکه جلو برود بیت accessed آن را صفر می‌کند اگر یک بود رد برگشت و صفر بود می‌فهمد که کسی از این استفاده نکرده پس این را انتخاب می‌کند و اگر کسی را انتخاب کرد بیت accessed آن را یک می‌کند اگر همه یک باشد یک دور کامل می‌زند همه‌ی آن‌ها صفر می‌شوند

این LRU نیست ولی مثل آن است.

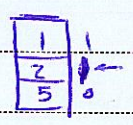
← یک بیت می‌خواهیم LRU را استفاده کرد

2 5 2 3 5 4 2 5 1 2 3 2



در این که مثل LRU عمل می‌کند در این ناحیه مثل FIFO عمل کرده

در این مرحله page fault نیست چون رجوع به 2 داریم بیت accessed آن یک می‌شود



Subject:
Year. Month. Date. ( )

در این مثال عملکرد clock مثل LRU نیست و این طبیعی است چون خود LRU نیست
در حیف و اتفاقی هم LRU هم clock خیلی بهتر عمل می‌کند

← clock پشتیبانی کمتر اضافه برای خواهد چون بیت A باید به طور اتوماتیک set شود. هر چیزی که page Table اضافه می‌کنیم کمتر افزایش پیدا شده به همین دلیل این پشتیبانی برای خواهد
وی reset شدن A توسط OS است به HW ربطی ندارد.

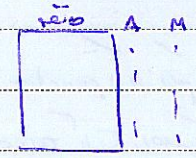
یک بیت دیگر می‌خواهیم به page Table اضافه کنیم:

بیت M: (Modify) اسم‌های مختلفی دارد در برداشته Dirty ← Intel ← D

← آیا محتویات page را تغییر داده‌ایم یا نه (خواندن و یا اینکه هم خواندن و نوشتن)

این هم HW انجام می‌دهد و هر وقت نوشتن این خوش set می‌شود. OS می‌تواند این بیت را بخواند
(کارهایی که HW انجام می‌دهد ساده‌ها است و OS باید از این استفاده کند و تصمیم بگیرد)

← اما خیلی کارهای مهمی انجام می‌دهد.



برای جابه‌جایی یک صفحه ① قدیمی را روی disk بریزیم
② جدید را روی صفحه بیاوریم

انتقال بین صفحه و disk خیلی زمان‌بر است

اگر بتوانیم اول frame را انتخاب کنیم که تغییر نکرده انتقال ① حذف می‌شود چون اول صفحه‌ای که روی disk است با اول صفحه‌ای که روی صفحه است یکی است ← زمان خیلی کم می‌شود

HW این امکان را به ما داده که بفهمیم حافظه‌ای 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

PAPCO

Subject:
Year. Month. Date. ( )

جلسه بیست و هفتم 87, 10, 3

مثال جلسه قبل.

احتمال page fault در حافظه مجازی ها

$$T_{eff} = 0.98(2ns + \underbrace{50ns}_{15ns}) + 0.02(1 - \frac{p}{0.02})(2ns + \underbrace{50ns}_{15ns} + \underbrace{50ns}_{15ns}) + \frac{p}{0.02}(2ns + \underbrace{50ns}_{15ns} + 10ms + \underbrace{50ns}_{15ns} + \underbrace{50ns}_{15ns})$$

cache باعث تاخیر این عددها می شود.

اگر حافظه cache داشته باشیم تاثیرش چی است؟ $0.9(10ns) + 0.1(10 + 50ns)$

تنها تاثیر cache سرعت دسترسی به حافظه اصلی است. هر چه دسترسی به حافظه رسیدیم متداول را کاهش می دهد

در صورت نبود cache، Teff قابل مقایسه بود با 50 ولی الان قابل مقایسه است با 15

حالا می خواهیم ببینیم که می توانیم 10ms را کاهش دهیم

اگر دسترسی به حافظه برای خواندن حافظه و نه نوشتن (نسبت عددی 1/4 دارند) (نوشتن، خواندن)

20% نوشتن و 80% خواندن

اگر بتوانیم از این استفاده کنیم می توانیم از بیت M استفاده کنیم و بجای جاهای استفاده کنیم که در آن نوشته نشده چون در این صورت قبل باید از روی disk بنویسیم

20% نوشتن ⟶ 10ms

80% خواندن ⟶ 5ms

قبل باید از روی disk بخوانیم بعد نوشتن داشتیم

اگر صفحه انتخاب شد برای جابجایی تغییر کرده باشد انتقال از روی حافظه به disk انجام می شود چون لازم نیست

بیشتر وقت ها می توان این صفحه را انتخاب کرد ولی همیشه هم می توان این کار را کرد اگر بتوان همیشه

این کار را کرد یعنی از سیستم عامل نوشتن انتقال دو طرفه انتقال یک طرفه

انجام می شود

$0.8(5ms) + 0.2(10ms) = 6ms$ ⟹ زمان دیگر انتقال

با این کار می توانیم 10 را به 6 کاهش دهیم که این باعث پیچیدگی سیستم می شود.

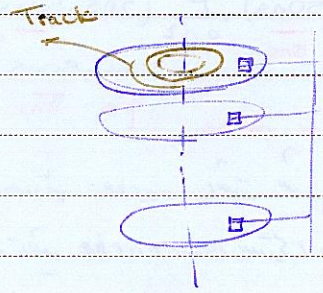
حافظه cache چون دیده می شود پیچیدگی ایجاد نمی کند

اگر $p = 10^{-6}$ قرار دهیم Teff را حساب کنید (نسبت قبل خیلی بهبود پیدا می کند)

Subject:
Year. Month. Date. ( )

مدیریت دیسک

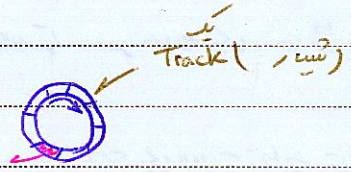
Disk ساختاری است که مجبور به رعایت ظرفیت، سرعت و ... است



هر کدام یک هد دارند که بالای هم یا پایین دیسک تماس دارد

این هدها هم همزمان در حال حرکت هستند

هدف این است که بدون این Track ها حرکت کنند



block یا sector (قطاع)

همین طور که این صفحه‌های چرخنده هد block های مختلف را می‌بیند واحد انتقال disk همین block یا sector است یعنی اگر بخواهیم حتی اطلاعات را جابجا کنیم یک block دو block ... را جابجا یا کند (واحد انتقال disk بیت نیست)

برای رسیدن به block به دو حرکت نیاز داریم ← انتقال: هد

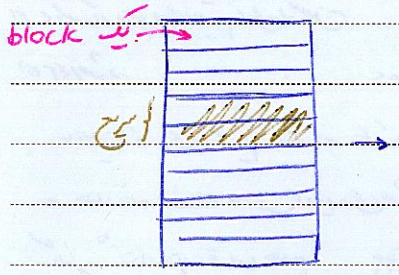
چرخشی: block مورد نظر به هد برسد

Track هایی که شعاع یکسان دارند ← یک استوانه به Cylinder

آدرس دهی اولین یک سیلندر تمام می‌شود و جلو می‌رود جدول می‌خواهند حرکت انتقالی را کم کنند

disk مکانیکی است و حرکت باید در آن انجام شود ← سرعتش خیلی کم می‌شود

جدول و کتاب block به OS است



می‌توانیم disk را این طوری ببینیم

در این صورت فاصله زمان دسترسی دو تا block مختلف با هم فرق نداشت اما در disk اگر block ها پشت سر هم باشند 4 دسترسی خیلی سریع می‌شود اما اگر پشت سر هم نباشد باید کلی حرکت مکانیکی انجام شود تا به block بعدی برسیم که این خیلی بد است. یعنی اگر فاصله زمان دسترسی به محل block بستگی دارد.

Subject:
Year. Month. Date. ( )

اگر blocks های که نیاز دارند پشت سرهم جا بشوند در جاهای مختلف disk قرار گیرند خیلی سرعت را پایین می آید. و اگر بخواهیم این نوع blocks را در disk پشت سرهم قرار دهیم دچار Fragmentation می شویم.

← مشکل disk اینست که مکان به شکل فرق دارد

از طرفی ترنسیت که به disk بالا هم دسترسی داشته باشیم بلکه اون حالتی است که همیشه به آن دسترسی داریم.

مثال) زمان پیگرد (seek) ← حرکت هد (به ساختار disk بستگی دارد)

" چرخشی circular ← زمانی که طول می کشد تا به سر block برسد

" انتقال transfer ← زمانی که انتقال یک block بین disk و حافظه صرف می شود

زمان پیگرد (میانگین) = 10ms ← مشخصه خود disk است و محاسبه نمی کنیم

چرخشی = 10000 دور در دقیقه (به سرعت بستگی دارد (چند دور در دقیقه می زند))

$6ms = 5\frac{6}{1000} = \frac{60}{10000}$ ← 10000/60 rps (زمان یک دور)

میانگین چرخش نصف این است چون بعضی وقت ها بیشتر از نصف و بعضی از وقت ها کمتر از نصف

میانگین چرخش آن می شود 3ms

زمان انتقال = می توان از روی زمان چرخش بدست آورد ← $\frac{3ms}{n}$

n sector / track

هر sector یا block، 512 byte است 320 sector/track

2560 sector

فرض کنید فایلی که اندازه اش 1.3MB است را می خواهیم بخوانیم کل زمان انتقال را تخمین بزنید

دو حالت ← همه جا پشت سر هم | بین این دو خیلی حالت ها ممکن است به شدت یعنی یک سری جا پشت سر هم

random همه | random یک سری

1 sector های پشت سر هم

2 " " تصادفی روی دیسک

خواندن یک بلوک (اول)

10ms
3ms
6ms
19ms

$\frac{2560}{320} = 8$ بلوک

$19^{ms} + 9^{ms} * 7 = 82ms$

P4PCO

Subject:

Year. Month. Date. ( )

بدترین حالت اینست که همه Sectorها پخش شوند

19ms زمان خواندن اولین شیار

زمان خواندن اولین sector به صورت random

10ms

3ms

6ms/320 زمان انتقال داده‌ها به سکتور

زمان خواندن یک فایل $(10ms + 3ms + \frac{6ms}{320}) \times 2560 = 33.32\,s$

نوبت اول و نوبت دوم را خواهیم داشت بلکه بیشتر نزدیک به حالت اول خواهیم بود

Subject:
Year. Month. Date. ( )

جلسه بیست و هشتم 87,10,7

زمان بندی دیسک : Disk Scheduling

چنین درخواست ها به OS برای دیسک رسیده اینجا چه طوری جواب داده می شوند؟ به ترتیب جواب می دهد یا نه؟

مثال) درخواست های رسیده برای disk :

55 58 39 18 90 160 150 38 184 : شماره شیارها

راه ساده اینست که این شیارها را به ترتیب اصلی جواب دهد

می توان راه دیگری پیدا کرد که این ترتیب را بهم زده و کمی میانگین زمان را کم کرد.

راه اول) head روی شیار 100

درخواست برای شیار 55 ← Seek time

✓ حرکت چرخشی برای همه یکسان است

میزان حرکت به این شیار برسد

FIFO

(45 + 3 + 19 + 21 + 72 + 70 + 10 + 112 + 146) / 9 = 55.3

به طور میانگین این مقدار حرکت کرده تا به شیار مورد نظر برسد

FIFO هیچ تلاشی برای بهتر کردن میانگین ندارد و دنبال اینست که به کار که زودتر رسیده زودتر جواب دهد

راه دوم) shortest Seek Time first

این درخواست ها رسیده می توانیم ببینیم که کدام شیار به جایی که ما الان هستیم نزدیکتر است مثلا

الان روی 100 هستیم باید ببینیم کدام به ما نزدیکتر است. در این جا نیازی به تخمین نداریم

ترتیب انتخاب شیارها

SSTF → 90 58 55 39 38 18 150 160 184

(10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24) / 9 = 27.5 ← میانگین حرکت

disk دینامیکی است و حرکت سر وظایف دارد. در هر دوری این امکان هست حرکت چرخشی بتواند باشد و جهت حرکت می تواند عوض شود. در وسیله دینامیکی اگر واقعا جهتش عوض شود دوتا مشکل پیش می آید

1. در قرون جهت حرکت نیروی زیادی به آن وارد می شود و بعد از مدتی از بین می رود

2. در هر دو طرف به دقت خوبی در پیدا کردن شیار داشته باشد

اشکال این دو روش، جهت حرکت هر لحظه می تواند تغییر کند

PAPCO

Subject:
Year. Month. Date. ( )

SCAN

برای حل این مشکل راهی پیش نهاد کرد که جهت حرکت تغییر کند

اگر در یک جهت حرکت می‌کنی به همان جهت حرکت کن تا آخر برو و سپس به جهت مخالف برگرد و بقیه را انجام

بدهد ← اگر به بالای سری تا آخر برو سپس برگرد

برگشتیم پایین بالایی کنی می‌خواهیم

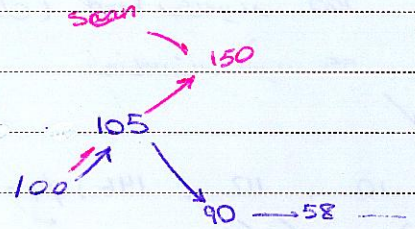
بالای درخت ⟶

Scan ⟶	150	160	184	90	58	55	39	38	18
	50	10	24	94	32	3			19 = 27.8

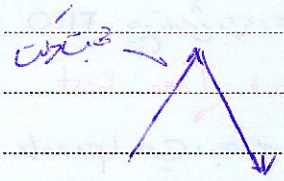
در SSTF اگر تجویزی ثابت باشد معادل حرکت Scan است ولی اگر این تجویز ثابت نباشد ممکن است

مجبور شویم بیاییم که مجبور به تغییر جهت باشد.

* حالا اگر به این مثال بشیم 105 را اضافه کنیم



این جا قابل اثبات نیست که SSTF همیشه بهتر جواب می‌دهد.



یک بار تغییر جهت این کار انجام می‌دهد

اشکال این روش سرعت از دست می‌رود.

در جهت بعضی دیسک حرکت کند که برسیم به بالا

که ساختمان بزرگی می‌شود برای حل این مشکل گفتند

دست را در یک جهت می‌گذاریم

clock ← چون ساعت هم در یک جهت می‌چرخد

C-Scan: این روش هزینه کمتری دارد

در این جا نگفتیم در جهت بالا رفته دست دارد

C-Scan:	150	160	184	0	18	38	39	55	58	90

برگشت در وقتی اشتباه دارد و به تر تیب می‌آید و به عقب زیاد هم این کار می‌کند

(50 + 10 + 24 + 184 + 18 + 20 + 1 + 16 + 3 + 32)

عدد خوبی بدست می‌آید چون برای 184 به 0 زیادی صرف کردیم که خیلی زیاد است ولی در اصل در این

مرحله برکت خیلی زیاد است.

PAPCO

Subject:
Year. Month. Date. ( )

Redundant array of Inexpensive (independant) disk : RAID

اطلاعات اضافی داریم ← بیشتر از یک disk

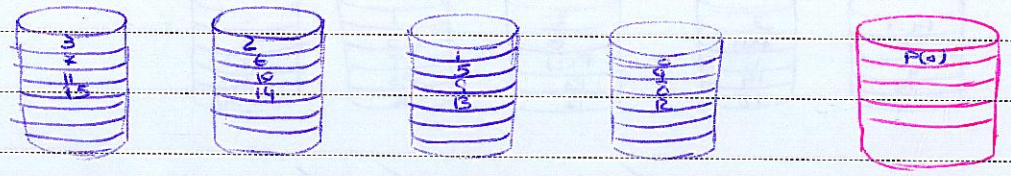
سرعت disk نسبت به حافظه خیلی پرسرعت پایین است (گلوگاه سیستم است) (battle neck)

بیشتر در server خاکبرد دارد ← چون دائماً باید سراغ disk برود تا درخواست‌ها را پاسخ دهد.

مثل web server ←

هدف RAID اینست که بتواند سرعت کار را افزایش دهد.

$P_b(n) = b_0 \oplus b_1 \oplus b_2 \oplus b_3$



block های داده‌ی disk ها توزیع کرد که موقعی درخواست برای 5 آمد اینجا می‌توانند موازی انجام دهند اگر روی یک disk نباشند

← اطلاعات را روی چند disk پخش کرد تا بتواند خیلی از درخواست‌ها را به صورت موازی انجام دهد.

مشکل اینجا چیست:

فرض کنیم احتمال خرابی یک disk ، P_0 باشد هر کدام از این disk ها خراب شود کار خراب می‌شود

احتمال اینکه یکی از چند disk خراب شود ← P

$P > P_0$ به راحتی می‌توان نشان داد.

قبلاً احتمال خرابی خیلی پایین داشتیم که الان این احتمال را افزایش داریم

برای حل این مسئله ← Redundancy در این ها وارد شد یک disk دیگر اضافه می‌کنیم که روی آن parity آن را ذخیره می‌کنیم)

با اضافه کردن این دیسک هر کدام از دیسک‌ها که از کار بیفتد قابل بازیابی است

فرض کنیم b_1 از کار افتاده $b_1 = P_b(n) \oplus b_0 \oplus b_2 \oplus b_3$

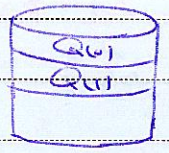
اگر وقتی disk از بین رفت اطلاعاتشان قابل بازیابی نیست و برای حل این باید اطلاعات بیشتر ذخیره کنیم

PAPCO

Subject:
Year. Month. Date. ( )

یک redundancy اضافه هم کاهش می‌دهد

Q از راه دیگری محاسبه می‌شود

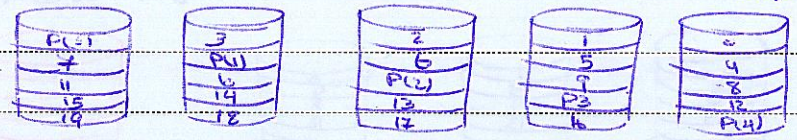


اگر بخواهیم قابلیت اطمینان بالا رود همان اندازه باید اطلاعات اضافی تر ذخیره کنیم

اشکال دیگر سیستم جدید: قبلاً write می‌شد یعنی علاوه بر block باید parity هم بنویسیم

RAID 5

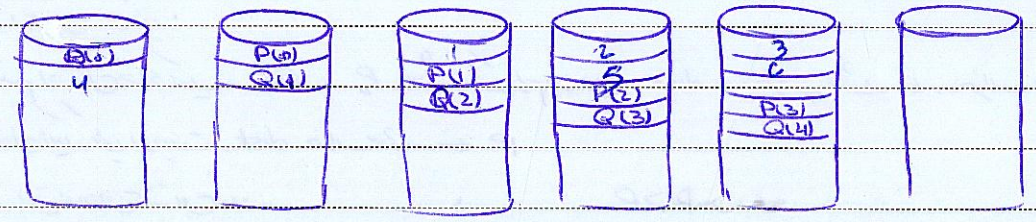
برای حل مشکل ترند



مسئله سازی ساختن را در صورتی که به یک دیسک مختلف نگه کنند حل می‌شود

RAID6 : همان Q روشش بیشتر (برای Server های مهم تر)

هم P و هم Q را چک می‌کنیم



برای Server های که درخواست زیادی دارند سیستم های Transaction

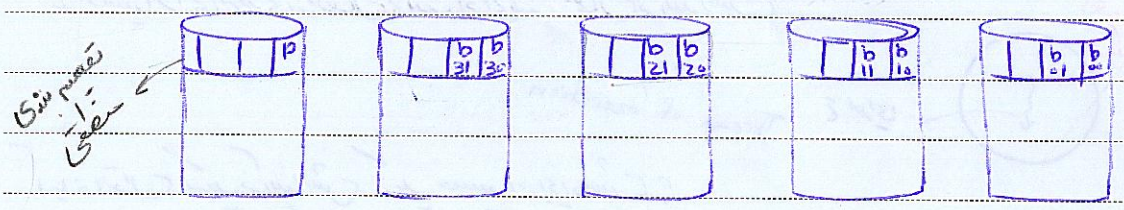
Raid را می‌توان برای سیستم های معمولی به کار برد تا سرعت disk را افزایش داد (برای این کار اطلاعات را باید خیلی ریز پخش کرد)

PAPCO

Subject:
Year. Month. Date. ( )

یک درخواست را سریع تر انجام می‌دهد

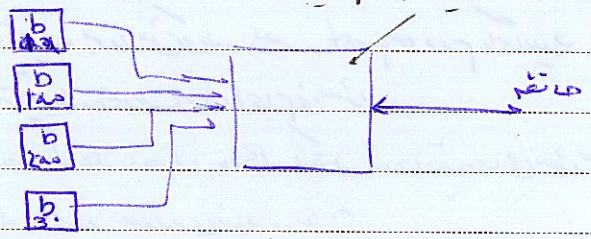
RAID 3 : ساختار یکسانی دارد که نحوه پخش شدن اطلاعات فرق دارد ← برای انجام یک درخواست



block ها را پخش کردیم و هر قسمت block در یک disk است

همیشه چون یک block را می‌خواهیم به صورت موازی انجام می‌شود از هر چهار disk خوانده می‌شود

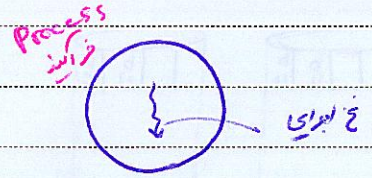
به صورت موازی انجام می‌شود



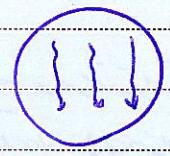
برای کارهای علمی و سیاسی ← محاسبات زیاد و دسترسی به disk کم

Subject:
Year. Month. Date. ( )

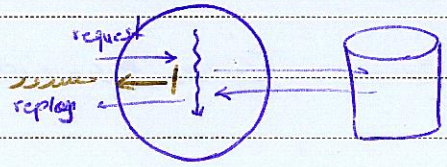
یک مسیر اجرایی که کدی را دنبال می‌کند را اجرایی‌اند، همان فرآیند است

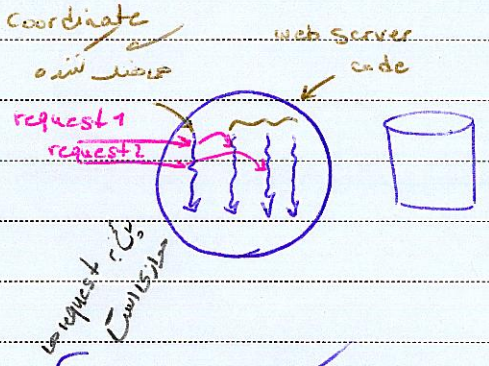


آیا می‌توان در یک فرآیند بیش از یک مسیر اجرایی داشت؟



مثال: یک web server یک process است که درخواست‌های زیادی از آن می‌شود
یک request را می‌گیرد و پردازش می‌کند و بعد request دوم را می‌گیرد
درخواست‌ها به ترتیب و به صورت سری اجرا می‌شود
این web server اطلاعات خیلی زیادی دارد که درخواست‌ها می‌شود پس در disk است
درسایی که برای disk می‌آید سرعت می‌شود و زمان
نسبت زیادی را باید صبر کند تا درخواست (اطلاعات)
برگردد





حالا به جنبه خ اجرایی در نظر می‌گیریم، disk همان یکی است
یک Thread باید باشد که بقیه فرق دارد و درخواست‌ها به آن
می‌آید پس درخواست‌ها را بین بقیه پخش می‌کند
کارش اینست که ببیند کدام خ آزاد است تا درخواست را
به آن بدهد

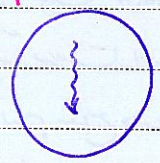
اجرا یعنی اینکه یک کد را دنبال کنیم

وقتی 2 request می‌آید چون Thread آزاد داریم می‌تواند اجرا شود پس یکی از Thread های آزاد را
می‌دهیم در اصل سرعت request
دستیابی به disk سریع است مگر اینکه چندتا disk دارد و اگر یک request برای disk رفت بقیه خ‌ها
می‌توانند قسمت‌های مختلف را که به disk ربطی ندارد اجرا دهند

Subject:
Year. Month. Date. ( )

همه web server ها چنین Thread کاری کنند و سرعت بالایی دارند

spread sheet



مثال: یک spread sheet را در نظر بگیرید

کاربرد: اطلاعات را از نت دریافت می‌کند و به جداول مربوطه می‌برد و آنها را update می‌کند

1- input / output

2- محاسبه و کمک به سازی

← شامل محاسبه کسر بودن مقدار عملیات انجام شده

در حال گرفتن input کار دیگری انجام نمی‌دهد بعد از اتمام گرفتن input، محاسبات را انجام می‌دهد محاسبات تمام شده می‌توانند دوباره input بگیرند ← عملیات به صورت تکراری انجام می‌شوند

در حال محاسبه اگر اطلاعات جدید آمدی می‌توانند بگیرد و این دو عملیات می‌توانند به صورت موازی انجام شوند.

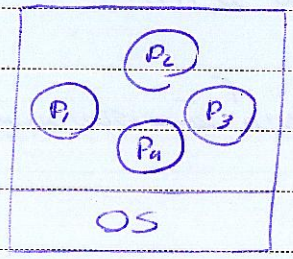
موازی { input / output
computation }

دیگر لازم نیست برای گرفتن input صبر کنند تا عملیات محاسبه تمام شود

به همین دلیل سیستم‌های امروزی این امکان را دارند که به صورت multiple thread کار کنند

برای اجرای process، باید به آن می‌دهیم و PCB را set می‌کنیم و pc (program counter) را روی نقطه شروع قرار می‌دهیم تا برنامه را از آنجا اجرا کند سپس در صورت نیاز یک Thread اجرایی داریم

یک کامپیوتر به شکل زیر داریم:



در یک سیستم کامپیوتری می‌توانیم تعداد زیادی از آنها را اجرا کنیم و کار کنند تا تمام شود

platform ← HW , OS

← بر این اساس می‌توان process تولید کنیم

Subject:
Year. Month. Date. ( )

یک چیز خیلی

← در یک platform می‌توانیم چندین process تولید کنیم

← در یک process " " Thread " "

آیا می‌توان گفت نسبت platform به process مثل نسبت process به Thread است؟ بله

سکوی پرتاب

← این جمله تقریباً درست است، دقیق نیست

process ها هر کدام فضای خود را دارند و از هم جدا هستند حتی حافظه آنها مشترک نیست پس هر کدام فضای اختصاصی خود دارد ← به خاطر فضا از هم بیشتر جدا هستند بخصوص حافظه

← یک حافظه مشترک داریم که واقعاً مشترک نیست و هر کس بخش جزای خود را دارد.

داخل PCB ← رجیسترها، stack، reg برای کنترل حافظه، PC، حسابرسی IO

accounting

برای اجرای یک فرآیند یک Thread داخلش هست چیزی می‌توانیم Thread اضافه کنیم؟ بله Thread ها در فضای process مشترک هستند

ایجاد یک Thread چیست:

یک پروسس OS است، همان طور که یک process می‌تواند یک process دیگر تولید کند

← یعنی یک system call است

فضای برای یک Thread:

هر process فضای نامنظم Thread ها تقسیم می‌تواند از آنها استفاده کند و در محدوده process کار کند

نخ: زیرمجموعه‌ای از فضای فرآیند را به کار می‌گیرد همان طور که process زیرمجموعه‌ای از فضای ماشین را می‌گیرد

یکی از مزایای مهم نخ: ایجاد و پایان آن خیلی کم هزینه‌تر از process است.

برای یک process ← اختصاص دادن فضای حافظه جدید و مستقل

← هر چی داشته حال process قبلی بوده

TLB در HW باید invalid شود (فضای سوئیچ TLB)

بعد از (تا) فرآیند باید فضای حافظه آزاد شود

PAPCO