جلسه اول

سیستم عامل چیست؟ دو وجه دارد:

1. os. as Resources manager مدیر منابع
2. os. as virtual machine ماشین مجازی

منابع (Resource): هر آنچه فرآیندها برای شروع و ادامه اجرا به آن نیاز دارند

فرآیندها منابع را: ۱. درخواست (Request) ← اگر آزاد نبود ← صف انتظار
۲. تخصیص داده می‌شود
۳. using استفاده می‌کند
۴. آزادسازی

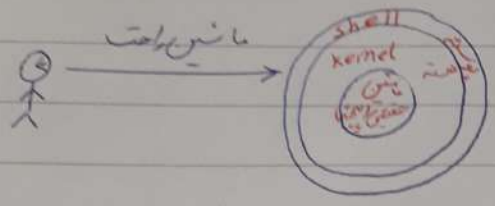
منابع بر دو گونه‌اند:
- physical فیزیکی: دستگاه‌های سخت‌افزاری مثل cpu، حافظه، دیسک، سایر دستگاه‌های I/O
- logical منطقی: مانند فایل‌ها - ساختمان داده‌ها

چرا مدیر منابع؟ جلوگیری از هرج و مرج و استفاده بهینه از منابع

وظایف سیستم عامل به عنوان مدیر منابع:

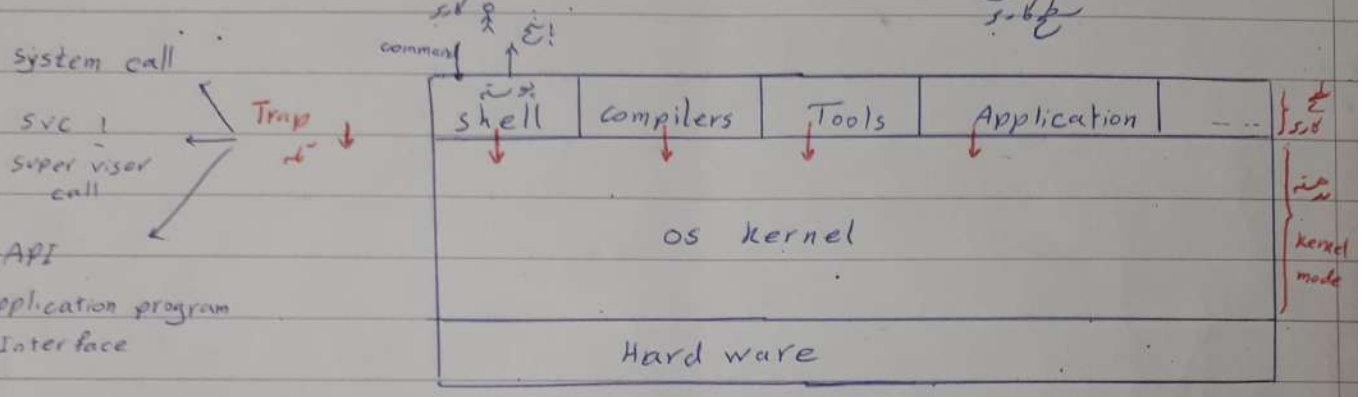
1) افزایش راندمان cpu و سایر منابع
2) Queueing & Scheduling صف‌بندی و زمان‌بندی
3) مدیریت دسترسی همزمان (multiaccess)، اشتراک منابع
4) امنیت و حفاظت protection, security
5) جلوگیری از ایده‌های ناخواسته و مشکل‌ساز مانند بن‌بست، قحطی، تداخل حاصل از رقابت
6) مدیریت فرآیندها (ایجاد، حذف، اجرا و ...)
7) تخصیص و آزادسازی منابع
8) مدیریت سطح پایین (ISR ها، Driver ها) و سطح بالا (مدیران I/O ها)
9) مدیریت حافظه و دیسک
10) مکانیزم‌های ارتباط بین فرآیندها مثل تبادل پیغام و لوله و ...
11) همگام‌سازی فرآیندها
12) مدیریت (ایجاد، حذف، کپی و ...) فایل‌ها و مرزبندی‌ها

سیستم عامل به عنوان ماشین مجازی



هدف از o.s از ماشین مجازی، پنهان سازی پیچیدگی‌های ماشین حقیقی (سخت افزار) و ایجاد واسط انتزاعی سطح بالا user friendly

بین کاربر / برنامه‌ها / سطح کاربر با منابع



• هرگونه رفتن از مد کاربر به مد هسته را تله گوئیم

- کدام تعریف برای o.s کامل‌تر است؟

a) نرم افزار سیستمی راه‌انداز و مدیر سخت افزار

b) نرم افزار سیستمی، راه‌انداز و واسط بین کاربر و سخت افزار ✓

c) مدیر منابع و ماشین مجازی ✓

d) ترکیب گزینه‌های ۱ و ۲ ← ۱. منابع منطقی را شامل می‌شود

۲. واسط بین برنامه‌های کاربر با منابع را شامل می‌شود

- کار با یک ماشین بدون سیستم عامل

a) محال است

b) بسیار سخت است اما حداقل باید یک نرم افزار راه‌انداز مثل Bios

c) بسیار سخت است (حتی بدون راه‌انداز مانند Bios) ✓

d) به سادگی امکان پذیر است

نسلهای معماری :

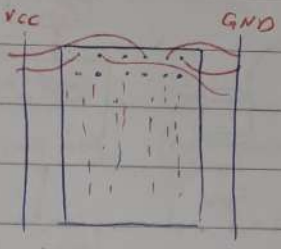
نسل اول : لامپهای خلاء ، تخته های مدارهای سوراخدار — چراغهای روشن و خاموش

ورودی — خروجی

برنامه نویس

اسمبلی — نوشتن — زبان ماشین

وارد تخته مدار ← حرکت الکترونها

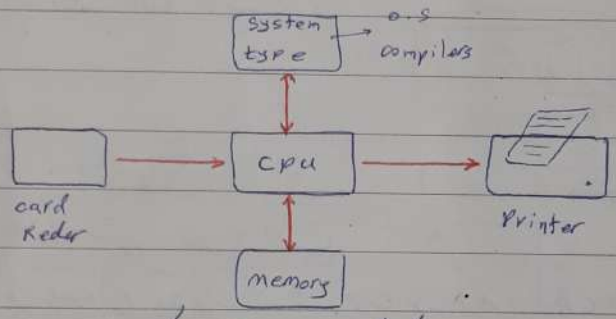


نسل دوم : ترانزیستورها

Page Driver

نوار گردانها — چاپگر — کارت خوانها

ذخیره سازها — خروجی — ورودی



ظهور سیستمهای عامل: Batch (دسته ای) ، سیستم عاملی که یک دسته کار را یکی پس از انجام دیگری پشت سر هم انجام می دهد

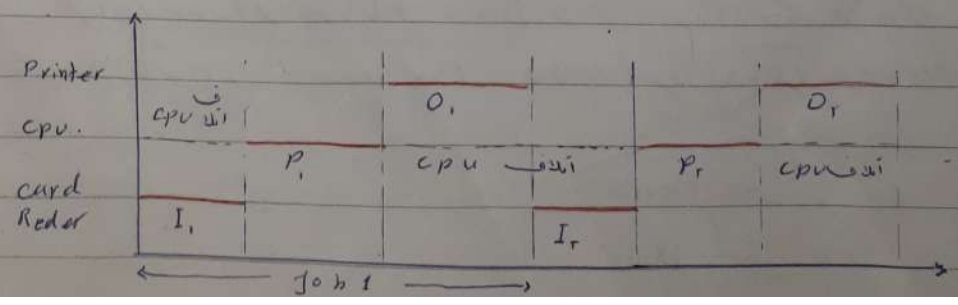
Run to complate — Batch — job

Job مجموعه ای است از : ۱. برنامه (ها) ۲. داده های ورودی Input Data .

۳. یک مجموعه (script) از فرامین o.s. - زبان jcl برای کنترل نحوه اجرای کار

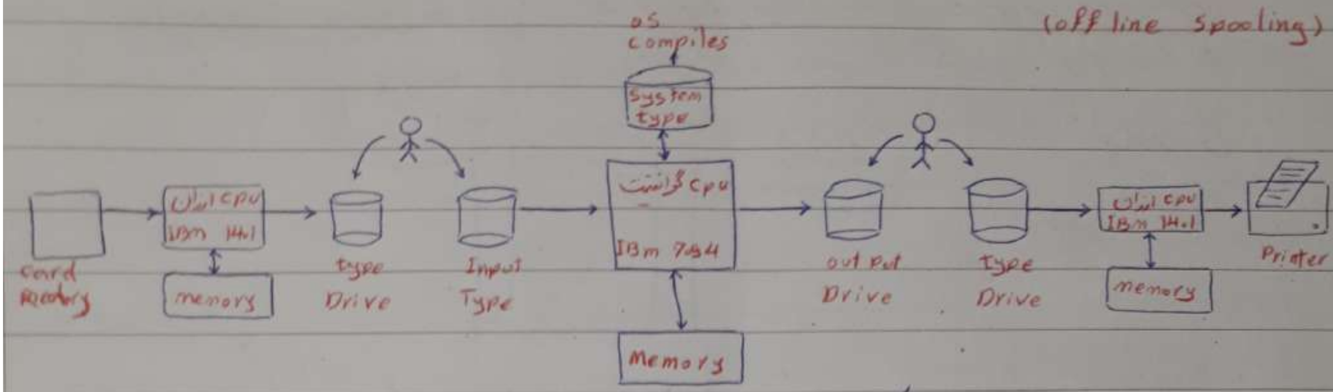
Job control Language

مشکلات Batch : ۱. اتلاف منابع خصوصاً cpu ۲. ارتباط off line با کاربر



Batch پیشرفته — هدف: افزایش بهره‌وری cpu (نه به صورت بهینه)

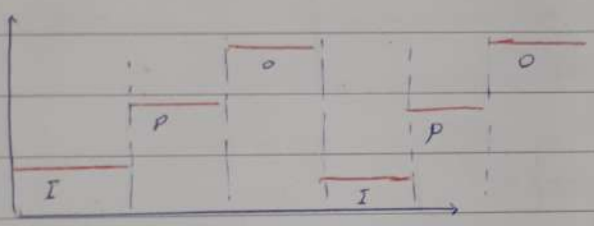
(off line spooling)



دستگاه‌های جانبی همزمان با هم کار می‌کنند

ولی هنوز Batch است،

زیرا cpu گرانقیمت:



فقط I/O روی Type است

- چرا راندمان بالا رفت؟ چون نوار سریع‌تر از CR، جایگزین است زمان بیکاری cpu کاهش یافته
- به چه قیمتی؟ خرید دو پردازنده ارزان + ۴ نوارگردان اضافه
- می‌ارزد؟ بله

نسل سوم: ظهور IC ها - دیسک ها - بافرها - کنترل‌کننده‌ها و غیره

ظهور سیستم‌های عامل چندبرنامگی multi programing،

هدف: استفاده بهینه از منابع مخصوصاً cpu

ت end استفاده از cpu

همزمان چندین فرآیند را با هم در حافظه قرار می‌دهیم تا هرگاه فرآیندی cpu را داوطلبانه رها کند

(یا خاتمه یابد یا مسدود شود) بلافاصله با سرعت به فرآیند بعدی درون حافظه سوییچ شود

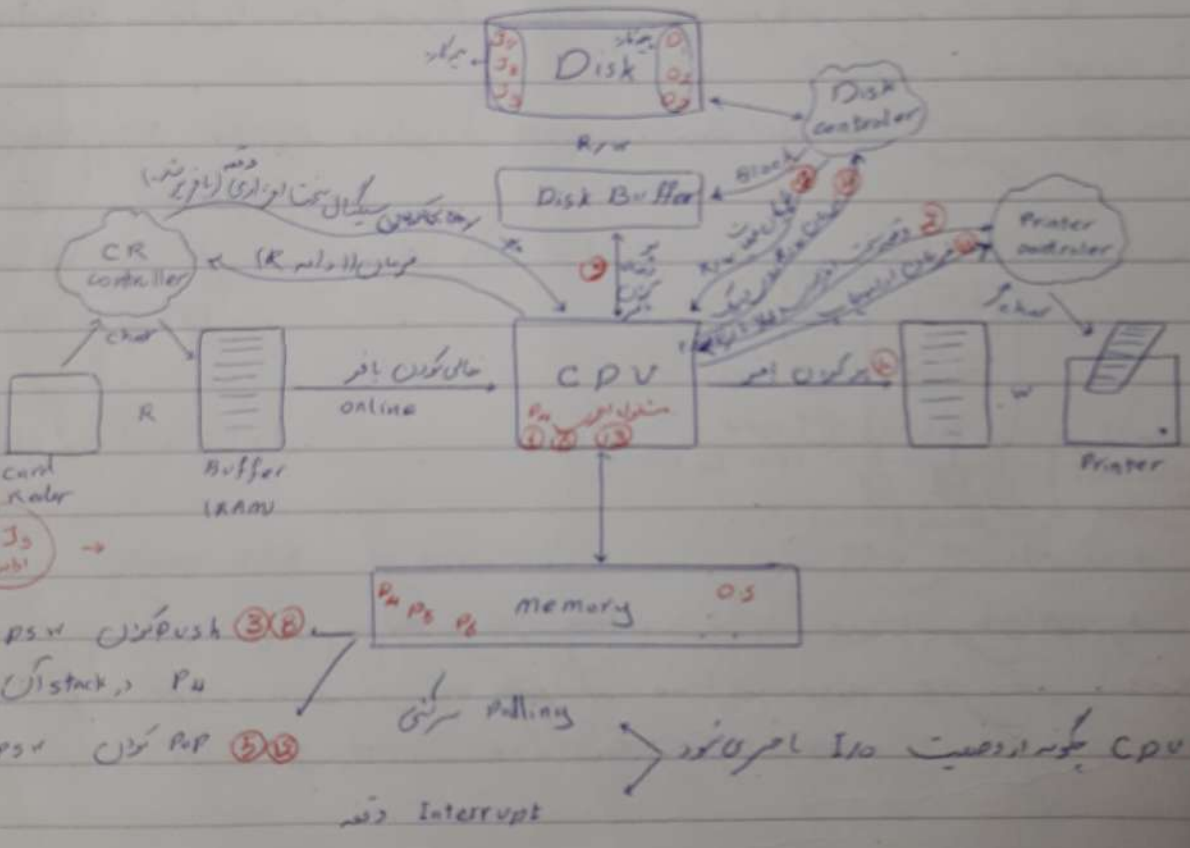
تخصیص انحصاری

- خاتمه (exit)
- کشتن (kill)
- مسدود Bloked

- تخصیص CPU به دو نوع است
  - انحصاری — Non preemptive
  - غیر انحصاری — preemtive
    - غیر قابل قبضه کردن
    - غیر قابل پس گرفتن
    - تخلیه پیش هنگام : انحصاری

- کارها به دو نوعند
  - CPU bound (CPU Bound) — بیشتر وقتشان صرف CPU و کمتر I/O می‌کنند
  - I/O bound (I/O Bound) — بیشتر وقتشان صرف I/O می‌کنند

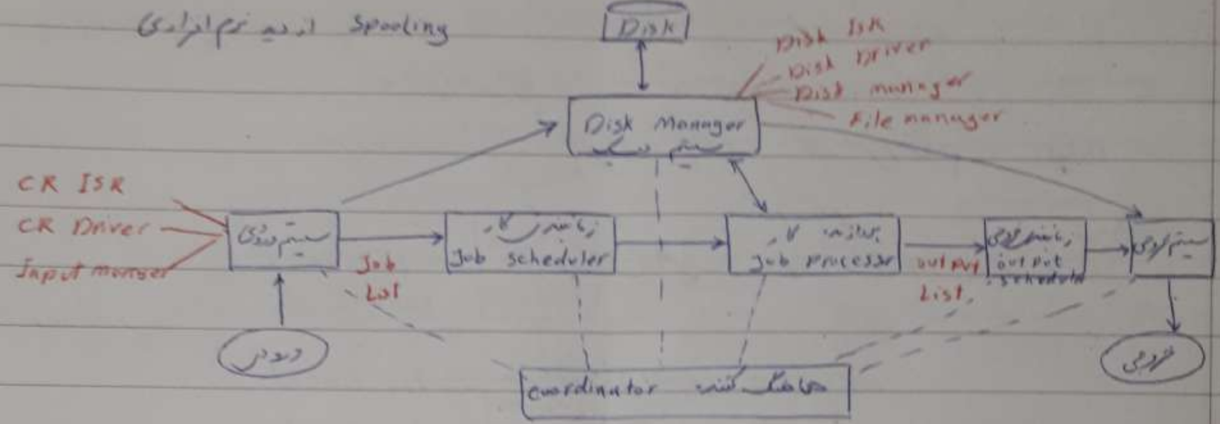
- Spooling online تکنیک



- CPU چگونه از وضعیت I/O باخبر شود
  - سرکشی Polling
  - وقفه Interrupt

پس سیستم عامل کیست؟ تمام کارهای فوق را سیستم عامل انجام می‌دهد و کارایی فوق را دهد CPU

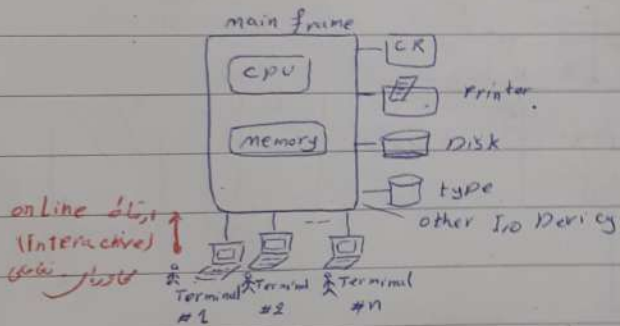
Spooling از دید نرم‌افزاری



• سه ابزار یا مکانیزم اصلی spooling چیست؟ 1. دیسک Disk 2. Intrupt؛ 3. Buffering

spooling یک تکنیک است که در چندبرنامگی به وجود آمد و در تمامی سیستم عاملهای پیشرفته به کار رفت.

نسل چهارم: IC های بسیار فشرده (VLSI)، ترمینال‌ها Terminals {صفحه کلید / صفحه نمایش} سپس PC ها (کامپیوتر شخصی) شبکه‌ای کامپیوترها ...

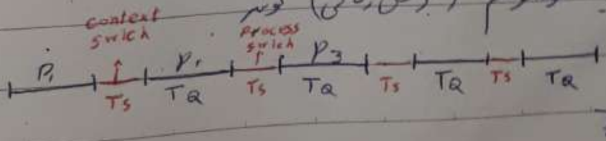


• سیستم چندبرنامگی نسل سوم با مشکل نیاز online مثبت چون زمانبندی آن انحصاری است و اگر یک فرآیند cpu را مدت زمان زیاد مشغول کند Response time کارهای تعاملی Tipical کوچک بیش از حد آستانه مورد قبول (DIR، ...) (یک ثانیه) بیشتر خواهد بود

Response time: از لحظه‌ای که کاربر Enter را بزند تا وقتی که اولین خروجی ظاهر شده

راه حل:

سیستم‌های عامل Time sharing (اشتراک زمانی) زمان cpu را به تکه‌های مساوی کوچک تقسیم می‌کنیم و هر یک از این تکه‌ها را Time slice یا کوانتوم (برش زمانی) گویند



TQ: زمان کوانتوم

Ts: زمان سوئیچ (تعویض متن)

حداکثر بازدهی $= \frac{T_Q}{T_Q + T_S} \times 100\%$

حداکثر درصد سربار تعویض متن $= \frac{T_S}{T_S + T_Q} \times 100\%$

- درصد استفاده از cpu در کدام روش بالاتر است.

a) Batch b) off line spooling c) multi programing ✓ d) Time sharing

سربار تعویض متن بالا است

با فرض زمانبندی RR یک کوانتوم به یک فرآیند می‌دهیم. دو حالت به وجود می‌آید:

۱) فرآیند در طی کوانتوم cpu را داوطلبانه رها می‌کند (خاتمه یا مسدود) در اینصورت بلافاصله به فرآیند بعد swich می‌کنیم (نمی‌گذاریم کوانتوم تا پایان هدر رود)

کی contect swich را شروع کرد؟ تایمر

Timer به cpu وقفه می‌دهد cpu برنامه جاری را قطع می‌کند وارد هسته می‌شود اجرای Timer ISR

هسته ← مرور

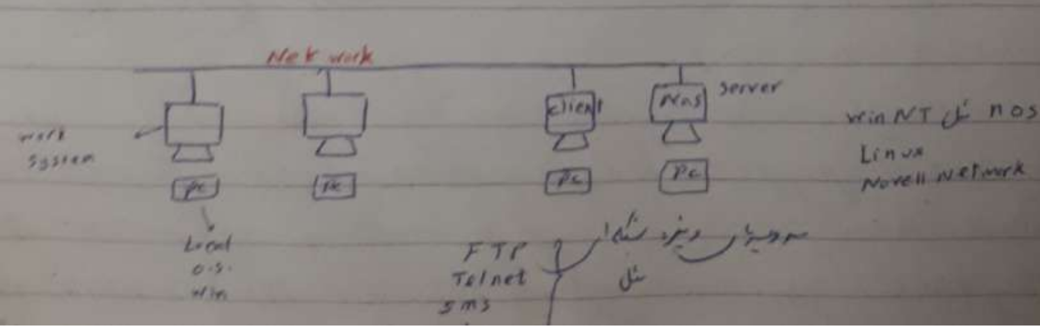
سیستمی Time shairny با n فرآیند هم روند دارید حداکثر زمان پاسخ برای یک کار جاری که طی یک کوانتوم انجام شود چقدر است؟

$n(T_Q + T_S)$

حداکثر زمان مورد انتظار ← $(n-1)(T_Q + T_S)$

۲) فرآیند تا انتهای کوانتوم مصرف می‌کند که با وقفه Timer و اجرای Timer ISR، اجرای Process scaduler cpu را از این فرآیند می‌گیریم و به فرآیند بعدی انتخاب شده می‌دهیم

(Network oprating system) Nos:



DOS ، (Distributed operating system) ، سیستم‌های توزیع شده

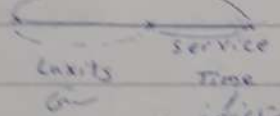
علی رغم اینکه چندین کامپیوتر مستقل وجود دارند اما طوری کار می‌کنند مانند سیستم واحد متمرکز به نظر می‌رسند

DOS	NOS
- کاربران چندگانگی شبکه نمی‌بینند	- کاربران در جریان چندگانگی شبکه هستند
- call by name	- call by Address
- LAN / WAN	- LAN ≠ WAN
- Transparent (شفافیت) تعدد سیستم‌ها و پیچیدگی‌های شبکه را مخفی می‌کند	

Real Time operating system ، سیستم‌هایی که کارها را تا قبل از Dead line انجام می‌دهند

یعنی برای کارها مهلت می‌دهد و کارها را قبل از پایان Dead line پایان می‌دهد

Dead line = مهلت



• سیستم‌های Real Time:
- نرم: در Dead line شان خیلی سخت نمی‌گیرند.
- سخت: در Dead line شان اهمیت بسیاری قائلند و اگر انجام نشود فاجعه رخ می‌دهد

Multiprocessing ، multiprocessor ،

وجود بیش از یک پردازنده، داشته که از همه [illegible] می‌شوند

سیستم‌های خاص Embedded ، (توکار)

سیستم‌های خاص منظوره‌اند که یک هدف خاص دارند و معمولاً درون دستگاه‌های خاص قرار می‌گیرند مانند سیستم‌های کنترل یک هواپیما، ماشین، موشک

سیستم‌های عامل mult

معمولاً یک کار بزرگ به کارهای کوچک شکسته شده به طوری که هر کدام قابلیت اجرای همزمان داشته باشند به هر کدام Task می‌گویند

هر Task یک زمانی [illegible] پردازنده را می‌گیرد، مثلاً این فرآیند را Time sharing [illegible]. [illegible] multiprocessor

۵

اما اگر همه Task ها به صورت نخ (Thread) در درون یک فرآیند قرار گیرند به آن چند نخی یا multiThread گویند

انواع وقفه (انواع تله) وقفه ها بر دو نوعند
- سخت افزاری: آسنکرون هستند معلوم نیست کجای برنامه می آیند
- نرم افزاری: سنکرون هستند و جایشان مشخص هستند

• وقفه های سخت افزاری به ۴ دسته تقسیم می شوند:

1) I/o Interupts → keybord printer
2) External Interupts → Timer
3) machine check Interupts → RAM, parity,
4) Restart → Reset کلید

استالینگ تمام وقفه های سخت افزاری را External می داند

• وقفه های نرم افزاری بر دو نوعند
دسته اول: هرگاه برنامه سطح کاربر خودش یک function درون هسته را صدا بزند و cpu وارد Kernel می شود و function مربوط را به صورت ISR اجرا می کند

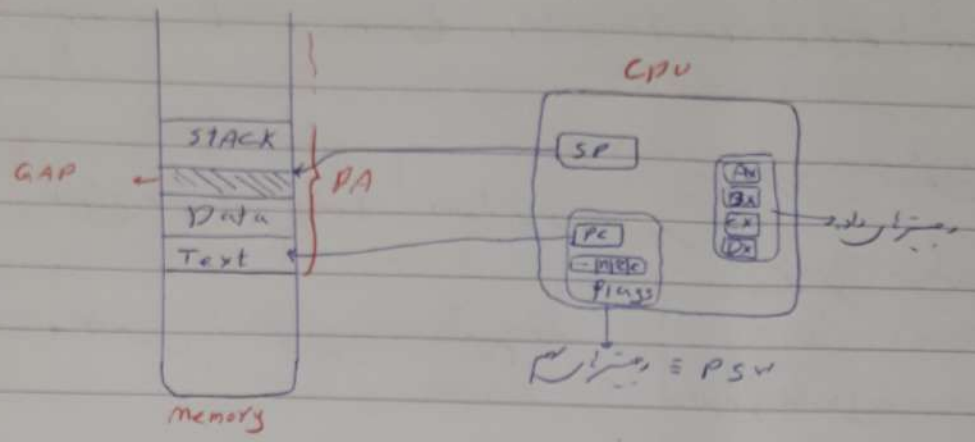
- System calls
- Syc Supervisor Call
- API Application progrm Interface

دسته دوم: برنامه کارهایی می کند که خلاف است مثل Devide by zero ، overflow ، Protection faults
دسترسی به حافظه دیگران ، دستورات ممتاز

- Program check خطای برنامه
- Exception استثنا
- Trap تله

فرآیند (فرا روند ، پروسه ، پردازه ، پردازش)
یک برنامه در حال اجرا - ولی الزاماً cpu را در اختیار ندارند از لحظه ای که یک job را انتخاب کرده و در
گردونه در حال اجرا قرار می گیرند
فاصله بین اجرا و از بین رفتن یک برنامه را فرآیند گویند

فرآیندها در حافظه چی دارند؟
1) Text 2) Data 3) stack



- فرآیند Running در cpu چی دارد؟ سه محتویات رجیستری

- یک فرآیند به Time sharing هرروند با بقیه انجام می‌شود کوانتوم آن تمام شده است اما هنوز اتمام دارد در لحظه switch به رجیستری و اطلاعات هم دیگری از فرآیند در یک جای مخصوص ذخیره شوند

- کجا؟ در Process table جدول فرآیند که جای آن در حافظه است (اطلاعات شناسایی فرآیندها را در آن نگهداری دارد) توسط سیستم عامل ساخته می‌شود

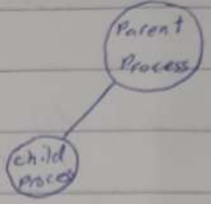
هر ردیف از این جدول یک فرآیند اختصاص دارد و PCB فرآیند نامیده می‌شود

Process control Block

فیلدهای Pcb:

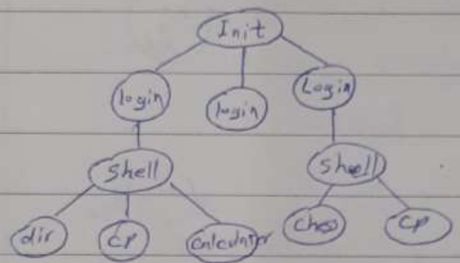
1) PID: Process Identifer
2) UID: User Identifer
3) GID: Group Identifer
4) Priority اولویت
5) PC (IP) } PSW
6) flags }
7) SP
8) other Registers
9) childs PID
10) cpu Time used
11) Process state
12) Pointer to text segment
13) Pointer to Data segment
14) Pointer to stack segment
15) Process received message (mail box)
16) ...

فرزندان فرآیند (Child Process) هرگاه فرآیندی بخواهد برنامه‌ای اجرا کند که خارج از متن برنامه خود است باید برای اجرای برنامه ابتدا یک فرآیند بسازد. فرآیند جدید را فرزند فرآیند مذکور می‌دانیم



هر فرآیند می‌تواند چند فرزند داشته باشد و فرزندان آن هم می‌توانند فرزندانی داشته باشند در نتیجه یک درخت در سلسله مراتبی به وجود می‌آید

در Unit این درخت واحدی باشد و ریشه درخت Init می‌باشد



ابتدا که سیستم عامل بالا می‌آید یک فرآیند به نام Init ایجاد می‌کند در Init به تعداد ترمینال‌های روشن فرآیند Login می‌سازد. Login، Username و Password را چک می‌کند
در اینجا user در ترمینال Login می‌کند که در این حالت پوسته که به عنوان فرزند می‌باشد
shell فرامین تحریر را می‌گیرد اگر درست باشد درخواست اجرا می‌کند
فرزندان به درخواست پدر و با واسطه هسته سیستم عامل تولید می‌شوند

در unix فراخوان سیستمی fork این کار را انجام می‌دهد

system call	توضیح
fork ()	Create a child process Identical the Parent یک فرآیند فرزند عین و دقیقاً مثل است را می‌سازد
exit ()	Terminate the Process Execution (خاتمه اجرای فرآیند) فرآیند به هسته می‌گوید با سلام من را چک کن
waitpid (Pid)	wait for a child Process (Pid) to terminate فرآیند پدر منتظر فرزندش با شماره Pid می‌شود تا خاتمه یابد (exit کند)
execve (Command/Parameter)	Replace the process core Image تصویر حافظه فرآیند را جایگزین می‌کند (با تصویر command عوض می‌کند)

شبه کد پوسته MINIX :

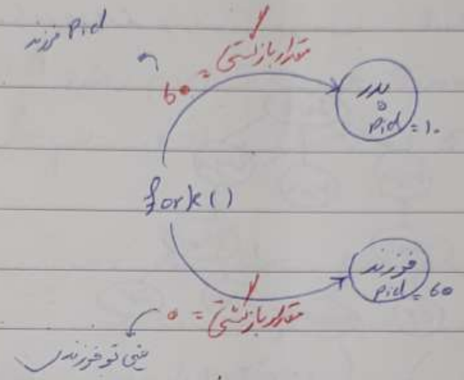
```
while (True) {
    Read_Command (comand, Parameters);
    if ( fork () != 0) {
        wait pid (x);   ← کد پدر
    } else {
        execve (Command, Parameters);   ← کد فرزند
    }   خودش را عوض کند (این کار را حتما انجام دهد)
}
```

1- shell منتظر فرزندان نمی شود ← multitaking

شرح fork() : پدر و فرزند دقیقاً یکی است

پدر	فرزند
Text	Text همان
Data	Data همان
Stack	Stack همان
PC	PC همان
SP	SP همان
⋮	⋮

فقط Pid فرزند با پدرش فرق دارد



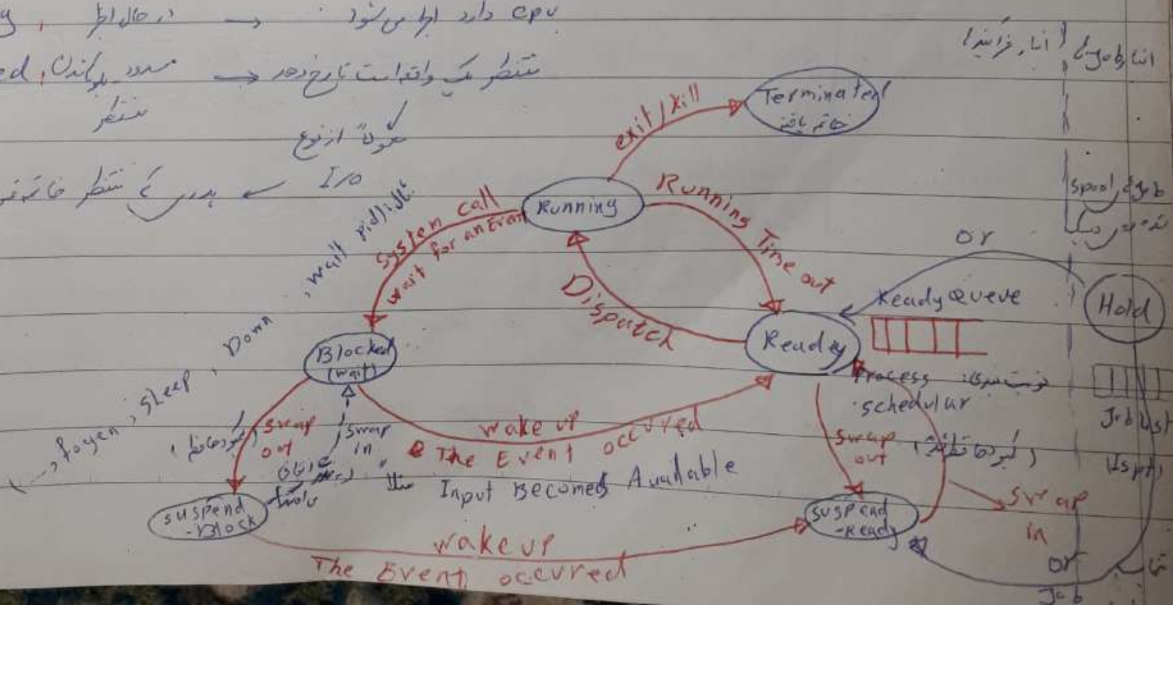
core Image : تصویر حافظه اصلی فرآیند بروی حافظه جانبی (دیسک)

پدر و فرزند می توانند برای تبادل داده از لوله و massage استفاده کنند

وضعیت فرآیندها (process stats) : فرآیندها ۳ وضعیت دارند :

1. Ready ، آماده اجرا ← منتظر دریافت cpu
2. Running ، در حال اجرا ← cpu دارد اجرا می شود
3. Blocked ، بلوکه، مسدود، منتظر ← منتظر یک واقعه است تا رخ دهد

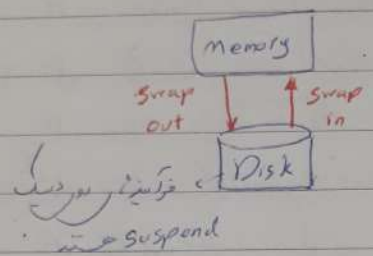
نمونه از نوع I/O ← پدر که منتظر خاتمه فرزندش است



• Stalling در کابش وضعیت Suspend را اضافه کرده

معلق - موقت

فرآیند به دلیل کمبود حافظه موقتاً به Disk منتقل شده است



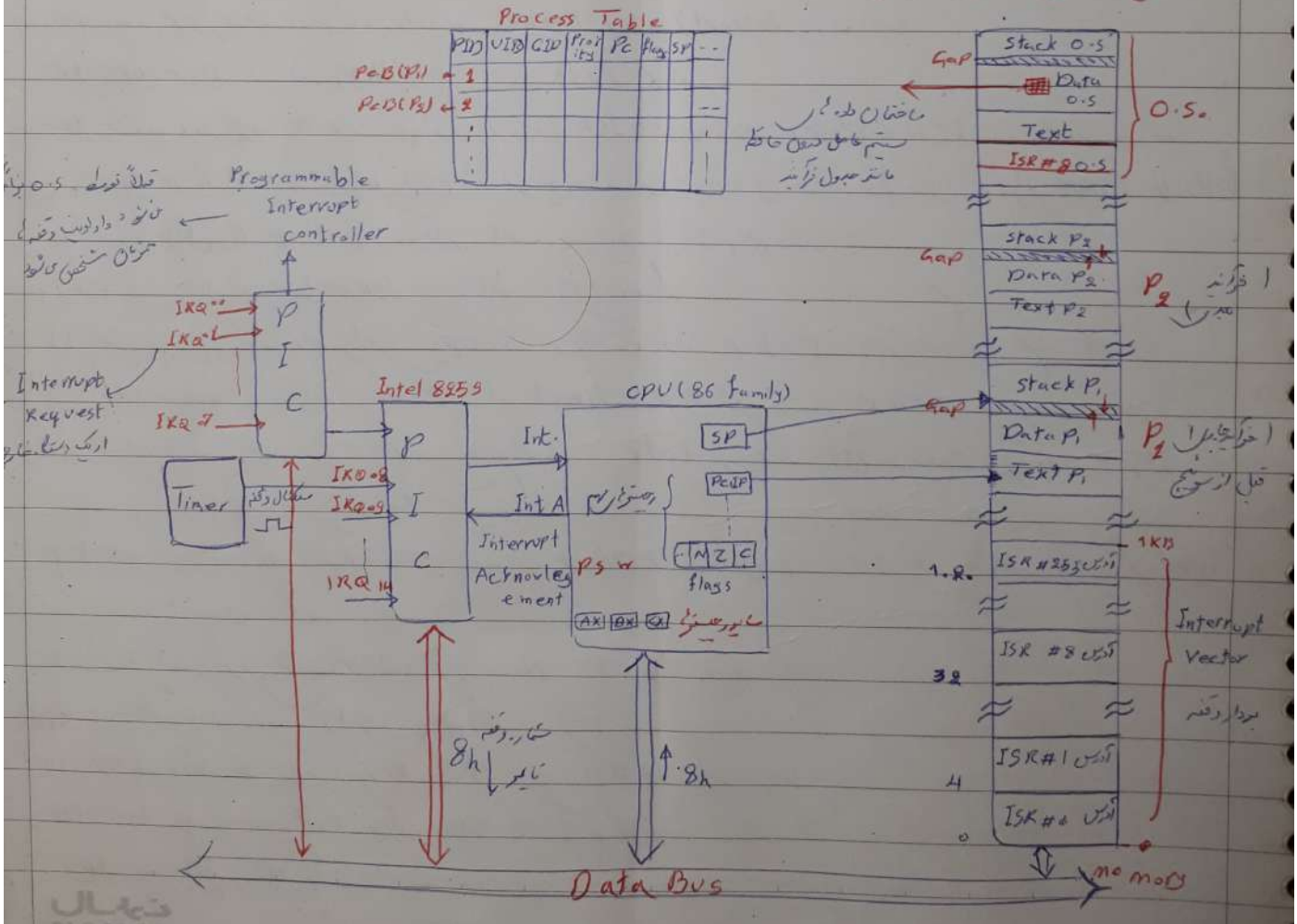
فرآیندی بود دیسک ← Suspend هستند

1) Low Level Scheduler : Job → Process Queue

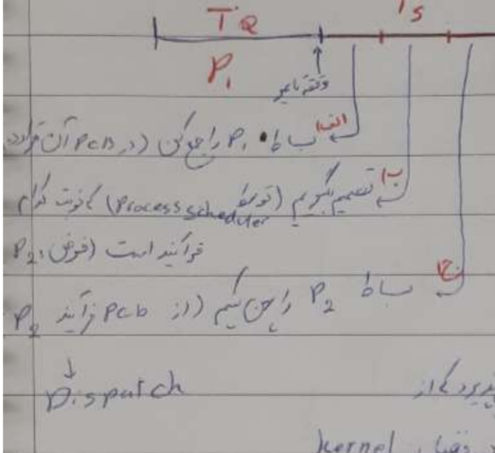
2) midel level : Process Ready → Running

3) high Level : suspend ready → Ready

تعویض متن (تعویض فرآیند)



مراحل Switch :



۱) Timer سیگنال وقفه را ارسال می‌کند

۲) PIC وقفه تایمر را به cpu ارسال می‌کند

۳) cpu دستورالعمل instruction جاری را تمام می‌کند

۴) cpu سیگنال Intrupt Acknowledge را فعال می‌کند (پاسخ به Int)

۵) PIC شماره وقفه را در Data Bus می‌گذارد و cpu برمی‌دارد

در خانواده cpu های 86 (Intel) که 256 وقفه سخت افزاری می‌پذیرد که از شماره 0 - 255 دارند. هر کدام از این وقفه‌ها بلوک ISR در فضای kernel دارد آدرس این 256 ISR آدرس 32 بیتی (4 بایتی) می‌باشند در جایی به نام بردار وقفه در پایین حافظه قرار دارند (256 × 4 = 1k)

به عنوان مثال وقفه timer در pc (کامپیوتر شخصی) شماره 8 را به خود اختصاص می‌دهد (ISR#8)

۶) مد را عوض می‌کند از مد کاربر وارد مد kernel می‌شود (Intrupt رخ می‌دهد)

۷) cpu، psw را در stackP1 push می‌کند

۸) شماره وقفه را در 4 ضرب می‌کند و آدرس ISR را از بردار وقفه برمی‌دارد و آن را در PC قرار می‌دهد

تا اینجا همه کارها سخت افزاری انجام شد از این جا به بعد سیستم عامل می‌آید و ظهورش با اجرای ISR#8 وارد عمل می‌شود

سیستم عامل کار خود را در سه مرحله الف، ب، ج انجام می‌دهد

الف) سیستم عامل در ابتدا ISR#8 بساط P1 را جمع می‌کند

۹) سیستم عامل کلیه رجیسترهای cpu را برداشته و در pcb P1 ذخیره می‌کند

حتی سیستم عامل می‌تواند psw را از stack pop کرده و در pcb P1 ذخیره کند

۱۰) در ادامه سیستم عامل بقیه فیلدهای pcb را update می‌کند مثلاً وضعیت فرآیند P1 را از Runing به Ready تبدیل می‌کند

۱۱) stack عوض می‌شود از stackP1 به stack سیستم عامل، sp را تغییر می‌دهد (عملاً ثابت‌ها) اصلی هستند

ب ۱) تصمیم گیری توسط process scheduler (معمولاً به زبان c)

۱۲) process scheduler فراخوانی می‌شود

۱۳) زمانبند فرآیند بر اساس الگوریتم زمانبندی (RR) فرآیند بعدی را

۱۴) زمانبند فرآیند Return می‌کند و به ادامه ISR برمی‌گردد

۱۵) Dispatch (آماده کردن) فرآیند انتخاب شده

۱۵) stack عوض می‌شود sp فرآیند P_2 از Pcb آن برداشته و در CPU بار می‌شود

۱۶) سایر رجیسترهای P_2 از Pcb بازیابی و در CPU بار می‌شود

۱۷) Psw از Pcb برداشته و در stack Push می‌شود

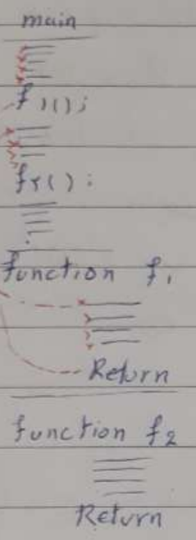
۱۸) سایر فیلدهای Pcb ~~bx~~ به روزی می‌شود مثلاً وضعیت P_1 از Running ~~Ready~~ Ready تغییر می‌کند

۱۹) دستور Iret اجرا می‌شود (بازگشت وقفه) حال CPU ابتدا مدار عوض کرده به مد کاربر برمی‌گردد سپس Psw را از stack Pop کرده و اجرای P_2 را Switch می‌کند

جلسه چهارم

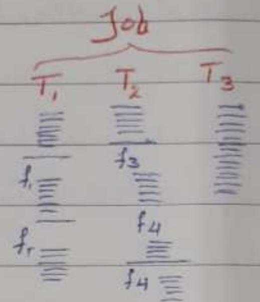
چندنخی (Multithreading)

Thread (نخ، رشته، ریسمان): رشته اجرای دستورالعمل‌های یک برنامه



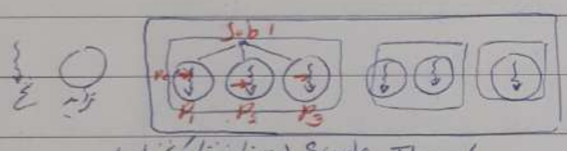
در برنامه‌سازی به شیوه سنتی این برنامه یک نخ می‌باشد

- در چه صورت ما چند نخ داریم؟ در صورتی که یک کار (برنامه) به اجزا تشکیل شود که بتوان آن‌ها را به صورت همروند (موازی) اجرا کرد



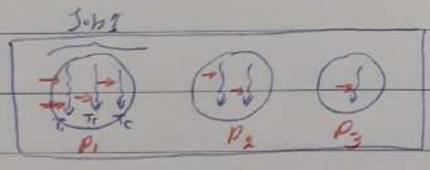
- سه تکه برنامه فوق را چگونه می‌توان همروند اجرا کرد؟

الف) اگر یک سیستم عامل SingleThread: می‌توان به صورت ۳ Task جدا که هر کدام یک فرآیند مستقل است می‌توان آن‌ها را همروند اجرا کرد



Single Thread (هر فرآیند فقط یک نخ دارد)

ب) در multi thread، هر فرآیند می‌تواند به تعداد دلخواه نخ داشته باشیم



multi thread

- چندنخی چه فایده‌ای دارد؟ ارتباط بین نخ‌ها در چندنخی کارآمد است چون در اینجا نخ‌ها درون فرآیند می‌توانند از حافظه مشترک به سرعت بالا استفاده کنند. اما در SingleThread ارتباط بین نخ‌ها حتماً باید از کانال سیستم عامل بگذرد و هسته باید دخالت کند (به خاطر امنیت) فرآیندها به هم دسترسی مستقیم ندارند و چون سربار کار هسته دارد کند می‌باشد

اگر ما ۳ (به عنوان مثال) نخ داریم درنتیجه ۳ stack نیز داریم

مثال هایی از multi thread

• فایل سرور : سه درخواست داریم با سه که متفاوت و سه Text متفاوت داریم
c1 : فایل f1 را کپی کن c2 : فایل f2 را پاک کن c3 : در فایل f3 بنویس

```
while (True) {
    Receive (Any & Req-mess)   ← تک نخ
    Process-Request();
    Send (Request-client, &Reply-mess)
```

در بیشتر اوقات بدین صورت cpu بیکار است چون I/O limited می باشد

اما در چند نخی کار بهتر است از این روش استفاده شود؛ چون نخ زیاد داریم در هر زمان تعدادی از آن ها از cpu استفاده می کنند لذا cpu بیکار نیست

• در web : در یک صفحه ۳۰ image داریم
به صورت multi thread ، ۳۰ thread برای هر کدام ساخته می شوند و این ۳۰ thread به صورت همزمان حرکت می کنند

- نخ ها را چه کسی مدیریت می کند؟ دانشمندان اختلاف دارند ، گروه اول می گویند که خود سیستم عامل و هسته سیستم عامل مدیریت شوند گروه دوم می گویند نخ ها در سطح کاربر مدیریت شود

مدیریت نخ

در سطح هسته O.S.	در سطح فرآیند
O.S. نخ ها را می شناسد - جدول نخ ها را نگهداری می کند	همه چیز به عهده فرآیند سطح کاربر است (O.S. نخ ها را نمی شناسد)
سربار تعویض نخ در هسته O.S. بالا است	اگر یک نخ Block شود فراخوان سیستمی (O.S. هسته استثنا کل فرآیند را می خواباند)
	cpu ها نمی تواند به طور متوازن بین نخ ها توزیع می شوند

معایب (← سطر دوم و سوم جدول)

ساختار سیستم عامل: ۴ ساختار دارد:

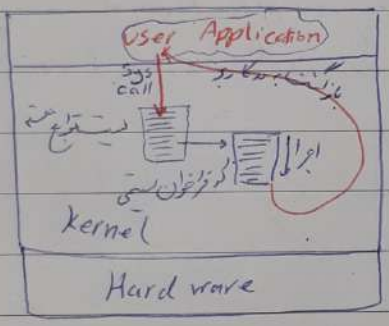
1) Monolithic یکپارچه
2) Layered لایه‌ای
3) Virtual machine ماشین مجازی
4) client / server (micro Kernel) مشتری

Monolithic: یک هسته غول‌پیکر یکپارچه

یک برنامه main + ساختاری تودرتو

معایب: ۱. قابلیت اطمینان بسیار پایین دارد (افتضاح است) چون احتمال Bug داشتن آن زیاد است
نگهداری و عیب‌یابی آن بسیار افتضاح است
۲. modularity ندارد سیستم از بخش‌های جزا خبری نیست
۳. قابلیت انعطاف‌پذیری (flexibility) آن پایین است

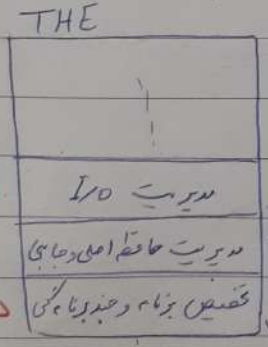
مزیت: efficient است performance آن بالاست



یک درخواست یک تله هسته است
کارآمد است

خواهیم دید در روش‌های بعدی تعداد تله بیشتر داریم
مثلاً در micro kernel حداقل ۴ تله است

Layered: هسته‌ای Layered بیازی شکل است. صورت لایه‌ای می‌باشند و هر لایه وظیفه مستقل خود را دارند (لایه‌ای از هم جزا هستند)

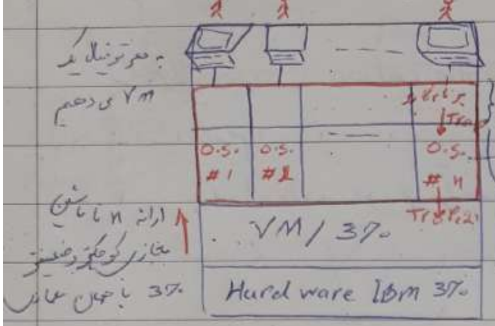


هر لایه، لایه‌های بالاتر از خود سرویس می‌دهد
لایه‌های پایین تر حدود اختیارات بیشتری دارند

modularity، Flexibility، قابلیت اطمینان آن از یکپارچه بیشتر است
ولی Performance کمتری دارد (چون لایه، لایه می‌رود پایین)

Virtual Machine : یک ماشین حقیقی بزرگ را به n ماشین مجازی کوچک تقسیم کنیم بدون پنهان سازی پیچیدگی سخت افزار یعنی تنها تقسیم منابع می کنیم

- چگونه منابع ماشین بزرگ را به n ماشین کوچک تقسیم می کنیم: با پارتیشن بندی Disk و پارتیشن بندی حافظه و اشتراک زمانی cpu

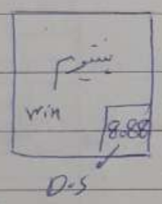


اگر برنامه کاربر بخواهد Run شود Trap رخ می دهد
بنابراین Performance پایین می آید

CMS → یک سیستم عامل تک کاربره محاوره ای است

مثال هایی از Virtual machine

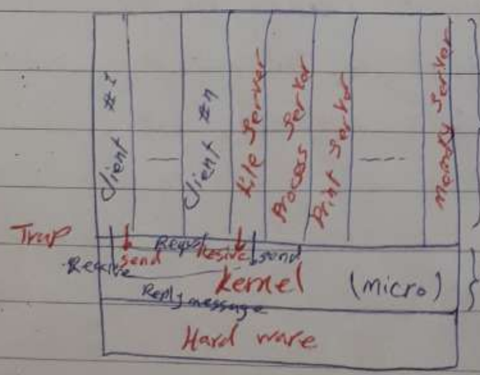
- VM-ware : روی یک PC چندین سیستم عامل را نصب می کند
- Exokernel : توسط دانشگاه MIT، این صورت است بر اساس VM است ولی سریع تر از VM/370
- اجرای ms-Dos در windows دو راه دارد یکی شبیه سازی است و راه دوم روش Virtual machine است؛



Client server (مشتری خدمتگذار)

micro kernel : هسته کوچک باشد برنامه نویسی راحت می شود، Bug کم می شود نگهداری آن راحت می شود احتمال fail کم می شود (به ندرت fail می کند)

سایر قسمتهای هسته را (قسمتهای مختلف که گفته شد [illegible] ها و سرورها) مجزا قرار می دهیم
این سرورهای مجزا در مود کاربر اجرا می شود



هیچ ارتباط مستقیمی بین فرآیندهای سطح کاربر (مشتری ها و سرویس دهنده ها) وجود ندارد
تنها مکانیزم ارتباطی آنها فراخوانی سیستمی Send، Resive و message passing می باشد

حداقل ۴ بار Send، Recive دارد ۲۰٪ کاهش efficiency دارد

مزایا: End سازه پذیری است، End felexibility و End reliability است
مناسب‌ترین ساختار برای سیستم‌های شبکه است
در این سیستم کل سیستم به صورت file می‌کند

Inter Process Commication IPC

سه بحث داریم:

1. تبادل داده بین فرآیندها Data communication

- shared Memory
- Message passing
- Pipe
- file sharing

(سریع‌تر / بیشتر)

2. هماهنگ‌سازی فرآیندها Process synchronization

- فرآیندها نباید از هم پیشی بگیرند هر کس کار خود را درست در سر جای خود انجام بدهد

3. Race condition باید مواظب باشیم وقتی دو کار با هم را انجام می‌دهیم دچار سوء نشویم

مثال 1) دو فرآیند P_1 و P_2 در یک سیستم اشتراک زمانی به صورت همروند اجرا می‌شود فرض کنید فرآیند P_1 در بخشی از کد خود ماتریس X را محاسبه و ایجاد می‌کند همچنین فرآیند P_2 در بخشی از کد خود دترمینان ماتریس تشکیل شده را محاسبه و ... برمی‌گرداند. برنامه‌ای P_1 و P_2 را به گونه‌ای بنویسید که سنکرون باشند (P_2 در محاسبه دترمینان از P_1 پیشی نگیرد) برای انتظار از Busy waiting استفاده نکند به جای آن از فراخوان سیستمی Sleep استفاده کند و به Block state ببرید. استفاده از Share memory مجاز است

نکته: همیشه احتمال بن بست در سیستم های تبادل پیغام وجود دارد

فصل دوم:

Scheduling (زمانبندی)

الگوریتم انتخاب کار یا فرآیند بعدی است

از بین کارهای Hold یکی را انتخاب کرده و آن را تبدیل به فرآیند می کند: زمانبند کار (سطح بالا) • Job scheduler
و آن را وارد گردونه اجرا می کند

فرآیند بعدی را انتخاب می کند و به cpu می دهد زمانبند فرآیند (سطح پایین) Process scheduler

• زمانبندی بر دو نوع است 1. Preemptive غیرانحصاری - پس گرفتنی - قابل گرفتن هنگام
2. Non preemptive انحصاری
وقفه ها را از این موضوع (Non preemptive) استثنا کنید

معیارهای زمانبندی:

فرآیندها را به یک چشم نگاه کنیم

از دید کاربر	از دید منابع سیستم
Fairness (زمانبند) عدالت، انصاف	Efficiency (کارایی - بازدهی) cpu (حداکثر)
Priority (زمانبند) اولویت	Throughput (توان عملیاتی - گذردهی) (حداکثر)
Response Time زمان پاسخ (حداقل)	
Turn arraund زمان بازگشت (حداقل)	
waiting Time زمان انتظار (حداقل)	
Deadline مهلت (زمانبند)	
Non starvation بدون قحطی	

- Fairness ← فرآیندها را به یک چشم نگاه کنیم
- Response Time ← از لحظه ای که کار وارد می شود تا وقتی که اولین خروجی نمایش داده شود
- Turn arraund ← از لحظه ورود کار تا لحظه خروج کار
- waiting Time ← مجموع زمانهای انتظار فرآیند در صف
- Non starvation ← انتظار حد بالایی داشته باشد

Efficiency: درصد استفاده مفید از cpu
Throughput: تعداد کارهایی که در واحد زمان اجرا می شود

انواع الگوریتمهای زمانبندی انحصاری

1. FCFS / FIFO اولویت با کاری است که بیشتر انتظار کشیده است (زودتر وارد شده)
* ساده است * قدیمی است * Fair (منصفانه است) * No starvation بدون قحطی
* Non preemptive انحصاری است

۱۲) SJF shortest job first (SPN) ابتدا کوتاهترین کار

اولویت با کاری است که زمان اجرایش کوتاهتر باشد

* No Fair
* starvation موجب قحطی کارهای طولانی می‌شود
* Non preemptive

مزایا: به حداقل رساندن { میانگین زمان انتظار (AWT) / میانگین زمان پاسخ (ART) } اما فقط در بین الگوریتم‌های انحصاری.

نکته: در کل یک الگوریتم غیرانحصاری به نام SRT (preemptive SJF) ، AWT و ART حداقل می‌کند

در کدامیک حداقل زمان انتظار را داریم:

a) FIFO b) SJF ✓ c) HRN d) RR

a) FIFO b) RR c) SJF d) SRT ✓ طبق نکته بالا

زمان اجرا - زمان پاسخ / زمان ورود - زمان خروج

کار	زمان ورود	زمان اجرا (سرویس)	زمان پاسخ RT	زمان انتظار WT
J_1	0	c	a+b+c	a+b
J_2	0	b	a+b	a
J_3	0	d	a+b+c+d	a+b+c
J_4	0	a	a	0

$a < b < c < d$

SJF: 0 — J_4 — a — J_2 — a+b — J_1 — a+b+c — J_3 — a+b+c+d

بزرگترین ضریب برای کوچکترین زمان سرویس، کوچکترین ... بزرگترین

$$ART = \frac{4a + 3b + 2c + d}{4}$$

$$AWT = \frac{3a + 2b + c}{4}$$

معایب SJF: قحطی دارد. در عمل قابل پیاده‌سازی نیست

با حل تخمین زدن می‌باشد

الگوریتم تخمین Aging

هدف تخمین زمان اجرای کارها با توجه به الگوریتم‌های کاربردی است

مثلاً فرض کنید کارهای پشت یک ترمینال :

تا اکنون ۵ کار را اجرا کرده و این کارها به ترتیب از قدیم به جدید T_0, T_1, T_2, T_3, T_4

حال نگفته‌اند زمان اجرای کار بعدی چیست کار برد با الگوریتم Aging تخمین بزنید

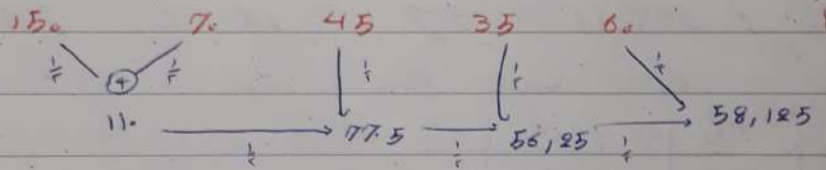
Tanenbaum: $a T_0 + (1-a) T_1$ — stalling: $(1-a) T_0 + a T_1$ برعکس!

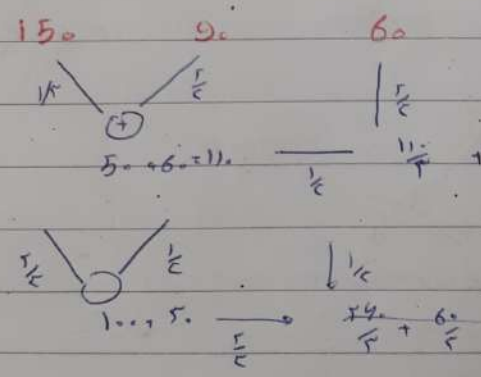
a ضریب است Aging

$0 < a < 1$

T_5 تخمین: $\frac{1}{2} T_4 + \frac{1}{2}\left(\frac{1}{2} T_3 + \frac{1}{2}\left(\frac{1}{2} T_2 + \frac{1}{2}\left(\frac{1}{2} T_1 + \frac{1}{2} T_0\right)\right)\right)$

$= \frac{1}{2} T_4 + \frac{1}{4} T_3 + \frac{1}{8} T_2 + \frac{1}{16} T_1 + \frac{1}{16} T_0$





Aging ($a = \frac{1}{2}$)

اگر جواب بدانند روش عکس (استاتیکی) نزدیم

(HRRN) HRN (3

Highest Respose Ratio Next بالاترین نسبت پاسخ

نسبت پاسخ یک فرآیند در لحظه t = Response Ratio = $\frac{w+s}{s} = \frac{w}{s}+\frac{s}{s} = \frac{w}{s}+1$

w: waiting Time [از لحظه ورود تا لحظه] زمان انتظار فرآیند تاکنون

s: Service Time زمان سرویس (اجرا) فرآیند

Running Time

Processing Time

CBT

- اولویت اجرا با فرآیندی است که نسبت پاسخ آن از بقیه بالاتر است
- الگوریتم انحصاری Non preemptive است
- $\frac{w+s}{s} = \frac{w}{s}+1$ = نسبت پاسخ

می‌توان ۱ را حذف کرد چون از اولویت همه یک واحد کم شود

در نتیجه: $\frac{w}{s}$ بالاتر ⇐ اولویت بیشتر

- (با w یکسان) فرآیند کوتاه‌تر ⇐ s کوچکتر ⇐ $\frac{w}{s}$ بزرگتر ⇐ اولویت بیشتر [رفتار SJF گونه دارد]

هدف: کاهش ART و AWT

- (با s یکسان) مدت انتظار بیشتر ⇐ w بزرگتر ⇐ $\frac{w}{s}$ بزرگتر ⇐ اولویت بیشتر [رفتار FCFS گونه دارد]

هدف: از بین بردن قحطی کارهای بزرگ

پس این الگوریتم یک رفتار بینابینی بین SJF و FCFS دارد

- قحطی ندارد
- هر بار باید $\frac{w}{s}$ همه فرآیندها را در حل مسئله به روز درآورید [از مقادیر قبلی استفاده نکنید]
- در مورد AWT و ART:

SRT < SJF < HRN < FIFO

(SRTF) (SPN) (HRRN) (FCFS)

- مانند SJF نیاز به تخمین s دارد (از قبل معلوم نیست)

مثال:

کار	زمان ورود	زمان اجرا
J1	0	9
J2	4	5
J3	6	4
J4	8	2

J1: 0–9, J2: 9–14, J4: 14–16, J3: 16–20

(در لحظه 4) چون انحصاری است در لحظه ۴ کاری انجام نمی‌دهیم

$\frac{w}{s}|_{J2} = \frac{9-4}{5} = 1$ Highest

$\frac{w}{s}|_{J3} = \frac{9-6}{4} = \frac{3}{4}$

$\frac{w}{s}|_{J4} = \frac{9-8}{2} = \frac{1}{2}$

$\frac{w}{s}|_{J3} = \frac{14-6}{4} = 2$

$\frac{w}{s}|_{J4} = \frac{14-8}{2} = 3$ Highest

هر بار محاسبه شود (بررسی در نقطه‌ها)

زمان پاسخ RT	(RT - S) زمان انتظار WT
9 - 0 = 9	9 - 9 = 0
14 - 4 = 10	10 - 5 = 5
20 - 6 = 14	14 - 4 = 10
16 - 8 = 8	8 - 2 = 6

$ART = \frac{41}{4} = 10.25$ $AWT = \frac{21}{4} = 5.25$

• در همه الگوریتم ها: $WT_i = RT_i - S_i$

$$\frac{\sum_{i=1}^{n} WT_i}{n} = \frac{\sum_{i=1}^{n} RT_i}{n} - \frac{\sum_{i=1}^{n} S_i}{n} \Rightarrow \boxed{AST = ART - AWT}$$

در مثال قبل زمان اجرا:

زمان اجرا(ی)
9
5
4
2

$AST = \frac{20}{4} = 5$

پس فاصله ART-AWT باید ۵ باشد

1) 5.25 , 10.25 ✓ تنها گزینه‌ای که اختلاف AWT ART برابر 5 است

2) 5.25 , 7.5

3) 7.5 , 10.25

4) 5 , 11

	ورود	اجرا
J_1	0	9
J_2	6	3
J_3	8	1

$t = 9 \longrightarrow \frac{w}{s} = \frac{3}{3} = 1$

$\frac{w}{s} = \frac{1}{1} = 1$

هر دو مساوی درآمد در اینجور مواقع اولی را انجام دهید.

	ورود	اجرا
J_1	0	2
J_2	0	1
J_3	4	3

$\frac{w}{s} = \frac{0}{2} = 0$

$\frac{w}{s} = \frac{0}{1} = 0$

در اینجور مواقع طبق فلسفه الگوریتم کاری باید انتخاب شود که جلوتر باشد

ب) الگوریتم های غیرانحصاری

۱) الگوریتم RR : (Round Robin)

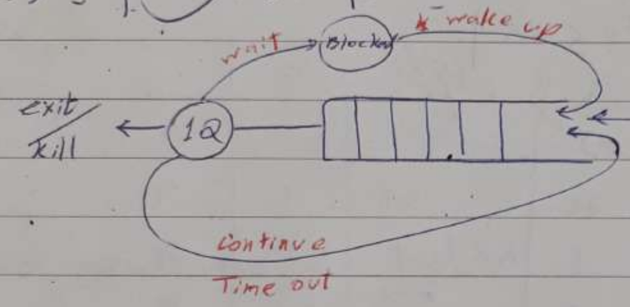
نوبتی - نوبت چرخشی

یک صف Ready داریم و فرآیند تازه وارد، به انتهای صف می رود. به فرآیند سر صف یک کوانتوم CPU داده می شود و ممکن است یکی از سه اتفاق زیر بیفتد:

۱) فرآیند تا آخر کوانتوم اجرا شود و بخواهد ادامه یابد: CPU را می گیریم و این فرآیند را انتهای صف می گذاریم

۲) فرآیند در حین کوانتوم exit یا kill شود: بلافاصله، به فرآیند بعدی سوئیچ می کنیم و دقیقاً یک کوانتوم CPU به آن می دهیم

۳) فرآیند در حین کوانتوم Block می شود: بلافاصله یک کوانتوم به فرآیند بعدی می دهیم و این فرآیند می خوابد. وقتی بیدار شد، به انتهای صف می رود



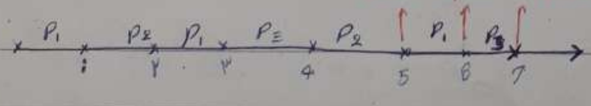
همیشه فرآیند تازه وارد ← مگر اینکه در صورت سوال خلاف آن قید شود

در برخی از سوالات ذکر می گویند: به فرآیند تازه وارد اولین کوانتوم را بدهید

P_1	۰	3
P_2	۰	2
P_3	3	2

RR, $T_Q = 1$

$T_S = 0$ (سربار سوئیچ نداریم)



خواص الگوریتم: • غیرانحصاری Preemptive • Fair • ساده است • قحطی ندارد

• preemption at Time slice (قبضه کردن در برهه زمانی)

• برای سیستم های Time sharing می باشد

- اگر اندازه کوانتوم کوچک باشد: ۱- سربار context switch زیاد می شود ۲- اینقدر کوچک نشود که کارهای نوعی کوچک ناتمام بمانند *

۷.

- اگر کوانتوم را بزرگ بگیریم . - هدف اصلی ما کوچک کردن RT کارهای کوچک است

به FIFO می‌رسیم

RT

	ورود	اجرا	SJF	FIFO	RR
J_1	۰	۱۰	۳۷	۱۰	۳۶
J_2	۰	۱۰	۲۷	۲۰	۳۷
J_3	۰	۹	۱۷	۲۹	۳۵
J_4	۰	۸	۸	۳۷	۳۲
			AKT: $\frac{89}{4}$	AKT: $\frac{96}{4}$	AKT: $\frac{140}{4}$

	اجرا	ورود	FIFO	RR
J_1	۱۰	۰	۱۰	۱۸
J_2	5	۰	۱۵	۱۳
J_3	2	۰	۱۷	۷
J_4	1	۰	۱۸	۴
			$\frac{۶۰}{۴}$	$\frac{۴۲}{۴}$

$SRT < SJF < HRN < FIFO < RR$ | کوانتوم کوچک

در حالت اگر کارهای کوچک داشتیم RR خوب عمل می‌کند

$\lim_{T_Q \to \infty} RR = FIFO$

$RR|_{T_Q \to 0}$ = cpu sharing ایده‌آل

RT		ورود (دقیقه)	اجرا (دقیقه)	
12,5-0	P_1	0	~~7~~	~~5~~ ~~4~~ ~~3~~ ~~2,5~~ 1
4-2	P_2	2	~~2~~	
10,5-5	P_3	5	~~2~~ 1,5	
13-6	P_4	6	~~3~~ 1,5	

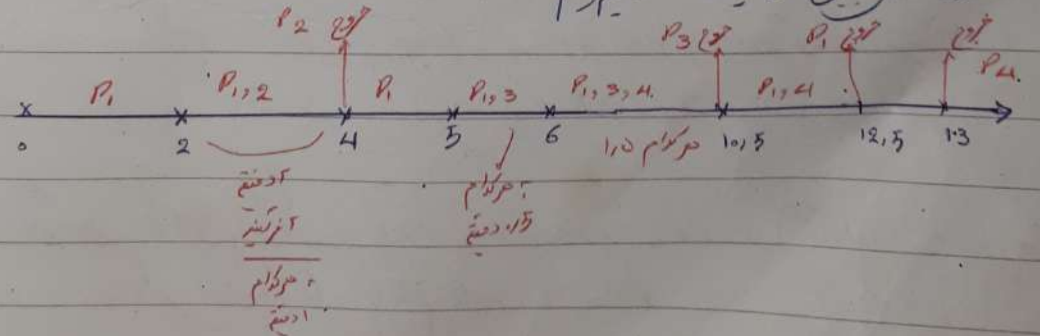
RR

$T_Q = 1ms$

$T_S = 0$

$AKT = ?$

یعنی زمان cpu را بطور مساوی بین فرآیندها تقسیم کنیم



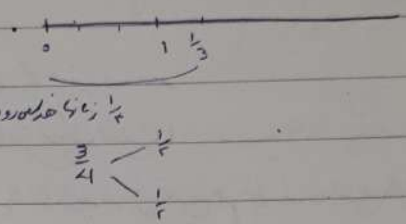
	ورود	اجرا
J_1	0	~~4~~ $3\frac{5}{8}$
J_2	0	2 $1\frac{5}{8}$
J_3	1	1

RR

TQ = 1ms

Ts = $\frac{1}{3}$ms

ART = ?



Shortest Remaining Time First (SRTF) SRT

ابتدا کوتاه‌ترین زمان انتخاب

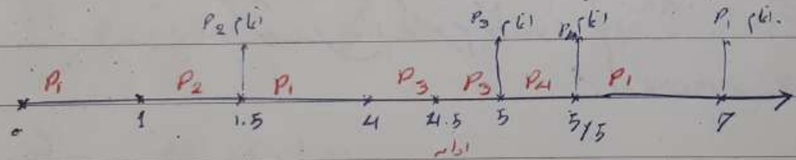
این الگوریتم مانند SJF عمل می‌کند اما اگر در طی انجام یک فرآیند، فرآیند دیگر وارد شود که زمان اجرای آن کوتاه‌تر از زمان باقیمانده کار جاری باشد، فرآیند جدید سوئیچ می‌کند

- Preemptive
- قسمت کردن در لحظه ورود
- کوانتمی و Time sharing نیست
- قحطی کارهای بزرگ دارد
- نیاز به تخمین S (زمان اجرا) دارد
- No Fair
- AWT و ART ایده‌آل می‌باشد (در همه الگوریتم‌های انحصاری و غیر انحصاری)

مثال: P_3

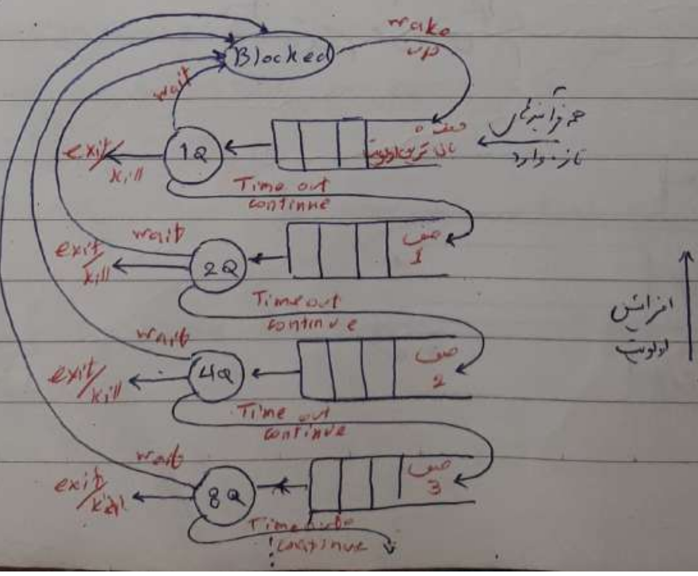
	ورود	اجرا	پاسخ
P_1	0	~~5~~ ~~4~~ 1.5	7
P_2	1	0.5	0.5
P_3	4	~~1~~ 0.5	1
P_4	4.5	0.5	1

$$\frac{7+0.5+1+1}{4} = \frac{9.5}{4}$$



multi Level Feedback Queus — MFQ ، MLFQ

صف‌های چند سطحی با بازخورد — FQ ، FB ، multiple Queues



• غیرانحصاری Preemptive • Time sharing • تقسیم کردن سر Time slice

• اولویت داریم (No Fair)

اهداف: ۱. تعداد سوئیچ را نسبت به RR تا حد زیادی (لگاریتمی) کاهش می‌دهد

۲. اولویت را به کارهای کوچک می‌دهد (بدون اینکه از قبل بداند چه کاری کوچک است)

۳. تا حدی شبیه SRT ، SJF عمل می‌کند، بدون اینکه از قبل ؟ را بداند [تخمین هم نمی‌زند، از FeedBack استفاده می‌کند!!!]

نکته: با اینکه تا حد زیادی ART و AWT را پایین می‌آورد

* بدیهی است که کارهای بزرگ تشنه می‌شوند و به صفهای پایین رفته دچار قحطی می‌شوند

— در این روش چندین صف داریم، اولویت با صفهای بالاتر است. یعنی وقتی به سراغ صف ۲ برویم که صف ۱ خالی باشد و وقتی به سراغ صف ۳ می‌رویم که صف ۲ خالی باشد.

همه کارهایی که تازه وارد می‌شوند در صف ۱ قرار می‌گیرند (الگوریتم خوش بینی است)

در صف اول (صف ۰) به هر فرآیند یک کوانتوم می‌دهد اگر خاتمه نیافت و مسدود هم نشد در انتهای کوانتوم به صف ۱ تبعید می‌شود و آنجا می‌ماند تا صف صفر خالی شود

اگر صف صفر خالی شد از آنجا به صف ۱ رفته و در آنجا به هر فرآیند ۲ کوانتوم می‌دهیم و ...

اگر فرآیندی ۱۰۰ کوانتوم بخواهد { RR ، ۱۰۰ سوئیچ / MLFQ ، ۷ بار

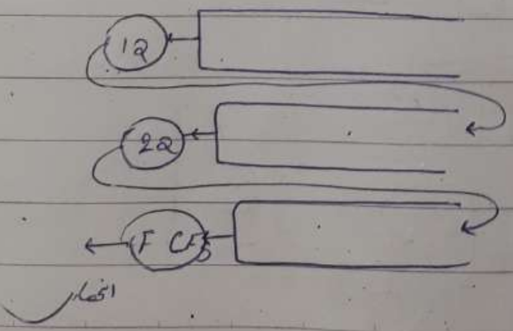
این الگوریتم روشهای مختلفی دارد:

— روش فوق را 2Q-MLFQ گویند

— روش 1Q-MLFQ ، در صف ۱ام ۱ کوانتوم می‌دهیم

— روش‌های فوق تعداد صفهاشون محدودیت ندارد می‌توانیم محدودش کنیم

مثال: صف n ام RR باشد   مثال ۱۲



جلسه سوم

نکته: هرگاه در حین پردازش مثلاً صف چهارم یک فرآیند وارد صف صفر شود صف چهارم را رها می‌کنیم و به صف صفر بر می‌گردیم

زمانبندی اولویت Priority Scheduling

تعیین اولویت‌ها توسط سیستم عامل انجام نمی‌شود و اولویت‌ها را کاربران سیستم از بیرون تعیین می‌کنند

- Policy (سیاست‌گذاری): این توسط کاربران سیستم تعیین می‌شود مانند اینکه اولویت فلان فرآیند چند است
- Mechanism / Policy: روش یا الگوریتمی است که برای تحقق در درون سیستم عامل به کار می‌بریم

مکانیزم ۱: CPU را به فرآیندی دهیم که اولویت بالاتری دارد و آن تا آخر اجرا کنیم. نوع زمانبندی در اینجا انحصاری است. مشکل آن وجود قحطی می‌باشد.

مکانیزم ۲: CPU را به فرآیند اولویت بالاتر دهیم و اگر در حین اجرای آن فرآیند، فرآیند دیگری با اولویت بالاتر از آن وارد شود بلافاصله به فرآیند جدید سوئیچ کنیم: نوع غیرانحصاری یا preemtion در لحظه ورود. دارای قحطی می‌باشد.

مکانیزم ۳: به صورت time sharing عمل می‌کند و کوانتوم بعدی را به فرآیندی دهیم که اولویت بالاتری دارد. غیرانحصاری است و قحطی دارد.

مکانیزم ۴: مثل مکانیزم ۲ عمل کنیم، اما هرگاه به یک فرآیند ۱ کوانتوم می‌دهیم یکی به اولویت سایر فرآیندهای منتظر اضافه کن. این مکانیزم باعث می‌شود که قحطی نداشته باشد.

مکانیزم ۵: مثل مکانیزم ۳ عمل کنیم اما یکی از اولویت فرآیندی که کوانتوم را می‌گیرد کم کنیم که این هم باعث جلوگیری از قحطی می‌شود

مکانیزم ۶: سیستم Time sharing است فرآیند تازه‌وارد در انتهای صف قرار می‌گیرد و هر کس به تعداد n برابر اولویتش کوانتوم می‌گیرد. و اگر باز هم به CPU احتیاج داشت به انتهای صف می‌رود

خصوصیات:

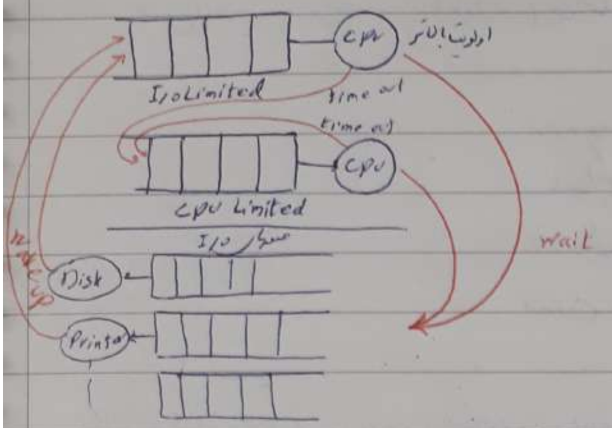
۱. به هر کس متناسب با اولویتش CPU می‌دهد
۲. قحطی ندارد

- به این مکانیزم WFQ یا (Weighted Fair Queuing) گویند

در کوانتوم بعدی نیز ۴ [illegible]

مکانیزم 7: در Time sharing کوانتوم در time sharing اولویت را به فرآیندهای I/O Limited می‌دهیم. یعنی اگر فرآیندی استفاده کرد.
هنگام دوباره آماده شود به آن اولویت $\frac{1}{4}$ دهد

مکانیزم 8: اولویت دادن به کارهای I/O Limited.



مکانیزم (multi level queue) MLQ

چند صف داریم اولویت با صف‌های بالاتر است. فرآیندها برحسب اولویتشان در نظر و رود در یکی از این صف‌ها قرار می‌گیرند. مادامی که صف اول خالی نشده به سراغ صف دوم نمی‌رویم و به همین ترتیب
پس بدیهی است که صف‌های پایین دچار قحطی می‌شوند
مکانیزم زمانبندی هر صف دلخواه و متفاوت است

مثال ۱: چهار صف MLQ داریم. که صف اول مربوط به Task های سیستم عامل است و آنها را FIFO انجام می‌دهد صف دوم کارهای I/O Limited که برای تعاملی و صف سوم کارهای cpu Limited تعاملی را انجام می‌دهیم صف ۲ و ۳ RR هستند صف چهارم را به کارهای Batch پس‌زمینه مربوط به کارهای of line اختصاص می‌دهیم و FIFO عمل می‌کند

زمانبندی بخت آزمایی (lottery Scheduling)

سیستم Time sharing است و به فرآیندها متناسب با اولویتشان تعدادی بلیط بخت آزمایی می‌دهند

مثال: فرآیندها با اولویت ۱ تا ۴ مفروض است

فرآیند	اولویت	بلیط‌ها
P_1	۱	۱
P_2	۲	۲-۳
P_3	۳	۴-6
P_4	۴	۷-۱۰

افزایش اولویت ↓

سر هر کوانتوم یک قرعه‌کشی می‌کنیم بلیط هر فرآیندی که برنده شد آن فرآیند یک کوانتوم cpu جایزه می‌گیرد (الگوریتم غیر انحصاری)

مزایای این الگوریتم: ۱. به هر کس متناسب با اولویتش cpu می‌دهد ۲. خیلی سریع پروسس را با شرایط جدید وفق می‌دهد (انعطاف پذیر است)

• جالب‌ترین خاصیت آن تبادل بلیط است مثلاً اگر فرآیند A خوابیده و منتظر این است که فرآیند B محاسباتی را انجام دهد و نتایج را به صورت پیغام برای آن send کند. فرآیند A می‌تواند بلیط‌های خود را به B (موقتاً) قرض دهد تا شانس B بیشتر شده و سریع‌تر اجرا می‌گردد تا اینجا این الگوریتم این امکان را فراهم می‌کند که یک فرآیند در حالت خواب هم از اولویتش بهره ببرد

زمانبندی تضمین شده Guaranteed Scheduling

در این روش می‌خواهیم تضمین کنیم که هر فرآیند سهم عادلانه و مساوی خودش از cpu را بگیرد. در روش غیر اعمال با سیستم Time sharing کار می‌کنیم و کوانتوم بعدی را به فرآیندی می‌دهیم که کسر زیر در مورد آن از همه فرآیندها کوچک‌تر باشد

$$\frac{\text{میزان cpu مصرفی فرآیند تاکنون}}{\text{سهم یکسان فرآیند از cpu (تاکنون)}} \rightarrow \frac{\text{مدت زمان حضور}}{\text{تعداد فرآیندها}}$$

چون صورت کسر به صورت پویا تغییر می‌کند سر هر کوانتوم محاسبات زیادی را برای همه فرآیندها انجام می‌دهیم (سربار الگوریتم بالا است)

- هدف این الگوریتم fairnes (عادلانه بودن) است

این الگوریتم end عدالت است ولی روش خوبی نیست (در عدالت افراط شده است)

زمانبندی بلادرنگ Real time scheduling:

- Real Time: Hard / Soft
- Real Time: Deterministic (قطعی بودن) => پریودیک / Stochastic (تصادفی)

- زمانبندی بلادرنگ پریودیک:

الگوریتم‌های زمانبندی زیادی در اینجا وجود دارد که بعضی از آنها Dynamic و بعضی از آنها static می‌باشد

یکی از الگوریتم‌های زمانبندی Dynamic آن Monotonic Rate نرخ یکنواخت است

اگر چند واقعه رخ داد. ابتدا آن واقعه‌ای را پردازش کنید که نرخ بالاتری دارد (فرکانس رخداد بالاتری دارد)

کلیه الگوریتم‌های زمانبندی معین باید شرط زیر را داشته باشند که سیستم قابل زمانبندی باشد

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

n: تعداد Task‌ها (تعداد وقایع)

C_i: میزان cpu مورد نیاز برای پردازش Task شماره i

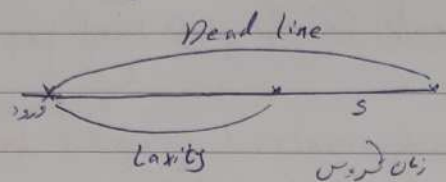
P_i: پریود رخ دادن واقعه مربوط به Task شماره i

- الگوریتم های بلادرنگ سیستم های تصادفی (Real Time)

۱) EDF Earliest Dead line First

اولویت را اول انجام بده که Dead line کمتری دارد (نزدیک ترین مهلت)

۲) LLF List Laxity (First) کمترین سستی



اشکال این روش این است که نیاز به تخمین دارد

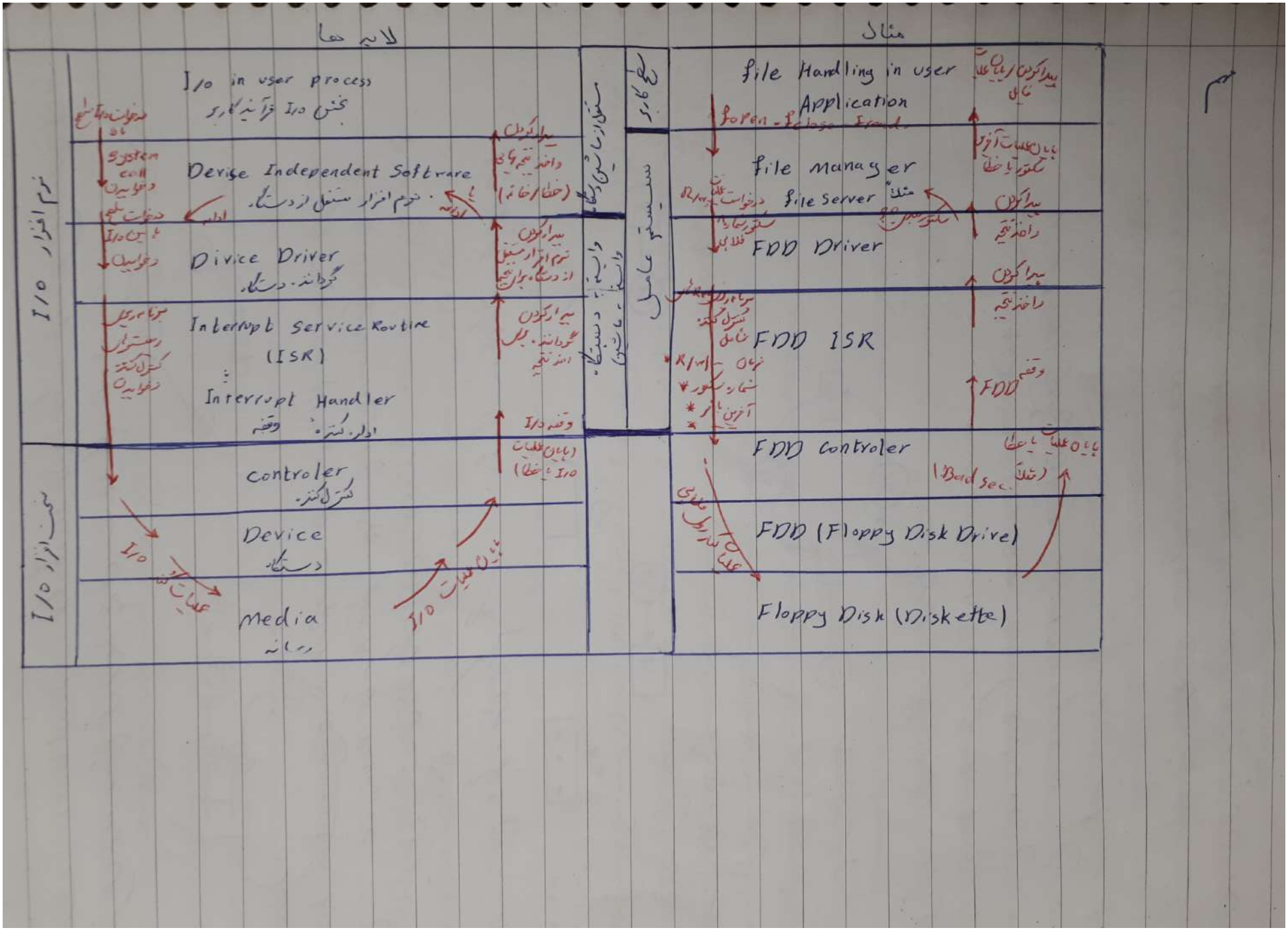
اولویت را اول انجام بده که Laxity کمتری دارد

فصل سوم مدیریت I/O

DMA را مطالعه کنید

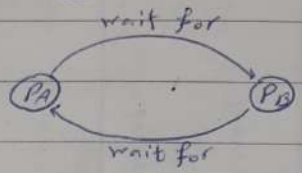
وظایف نرم افزار مستقل از دستگاه

۱. ایجاد یک واسط یکسان بین برنامه های کاربرد و Driver های مختلف و متعدد
۲. نام گذاری دستگاه (یک نامگذاری یکسان و همسان)
۳. بافرینگ، ایجاد و مدیریت بافرهای I/O
۴. فراهم سازی اندازه بلاک های I/O مستقل از دستگاه
۵. گزارش خطا
۶. تخصیص و آزادسازی دستگاه I/O
۷. صف بندی ← Sche درخواست های I/O
۸. حفاظت و امنیت دستگاههای I/O


لایه ها
مثال
نرم افزار I/O
بخش I/O فرآیند کاربر
I/O in user process
Device Independent Software
نرم افزار مستقل از دستگاه
Divice Driver
گرداننده دستگاه
Interrupt Service Routine (ISR)
Interrupt Handler
کنترل کننده
controler
Device
دستگاه
Media
رسانه
سخت افزار I/O
مستقل از ماشین دستگاه
وابسته به دستگاه / ماشین
سیستم عامل
File Handling in user Application
File Manager
File Server
FDD Driver
FDD ISR
FDD controler
FDD (Floppy Disk Drive)
Floppy Disk (Diskette)

جلسه دهم

بن بست : (Dead lock) حالتی که گروهی از فرآیندها هریک منتظر رویدادی باشند که وقوع آن فقط از اعضای همان مجموعه بوجود آید این فرآیندها دچار بن بست یا سیکل انتظار ابدی شده اند

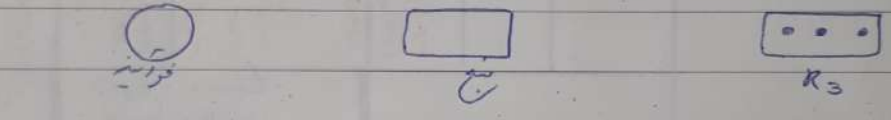


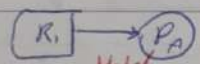
دلایل انتظار :

۱. ارتباط communication : مثلاً A منتظر است B کاری را انجام دهد یا یک پیغام برای A بفرستد
۲. منبع Resource فرآیند A منتظر منبعی است که آن در اختیار فرآیند B ای باشد

• بن بست می تواند ترکیبی از این دو هم باشد

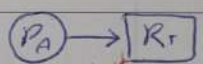
- بن بست منابع را چگونه مدل می کنیم ؟ با یک گراف ، نام گراف تخصیص منابع



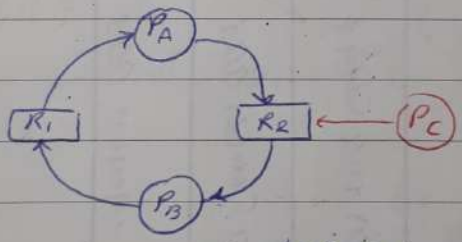


منبع R1 در اختیار فرآیند A است

Allocated



فرآیند A درخواست منبع R1 را دارد و منتظر است تا آنرا دریافت کند



بن بست

اگر PC را هم اضافه کنیم باز هم هر ۳ فرآیند دچار بن بست شده اند

- در این گراف به طور ضمنی نکات زیر فرض است

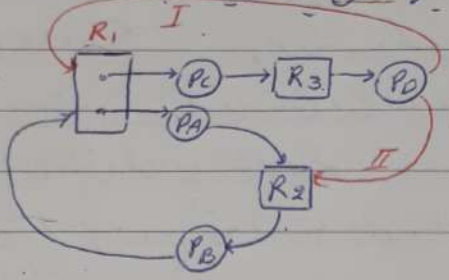
۱. منابع R1 و R2 منحصر فرد و قابل اشتراک نبوده هستند
۲. منابع Mutual Exclusion هستند (دو به دو با سازگارند)
۳. منابع Non Premtive هستند (غیر قابل پس گرفتن و انحصاری هستند)
۴. فرآیندها غیر قابل بازگشتند

مثال: هیچ وقت DBMS نمی تواند منجر به بن بست شود (چون انحصاری نیست)

مثال: خواندن از فایل نمی تواند منجر به بن بست شود (چون MutualEx ندارد)

بلاتر می‌تواند منجر به بن بست شود

• اگر بن بست رخ داده است آنگاه هیچ راهی جز کشتن یکی از فرآیندها وجود ندارد



۱. حلقه دارد بن بست است
۲. حلقه ندارد بن بست است
۳. حلقه دارد بن بست نیست ✓
۴. حلقه ندارد بن بست نیست

P_D منتظر هیچ کس نیست در نتیجه R_3 آزاد می‌شود. وقتی R_3 آزاد شد P_C انتظارش تمام می‌شود و می‌تواند ادامه یابد P_C به کارش ادامه دهد یک نسخه از R_1 آزاد می‌شود. P_B از حلقه در می‌آید و حلقه شکسته می‌شود

← بن بست نداریم

اگر حالت I یا II را اضافه کنیم بن بست می‌شود

شرایط بن بست: برای بن بست ۴ شرط لازم و کافی وجود دارد (بن بست رخ می‌دهد اگر و فقط اگر):

۱. Mutual Exclusion ← انحصار متقابل برقرار شود (منبع یا آزاد است یا در اختیار یک فرآیند است)
۲. Non Preemptive ← انحصاری باشد (منبع از زور توان در وسط کار پس گرفته نشود فرآیند خودش داوطلبانه آزاد کند)
۳. Hold & wait ← نگهداری و انتظار: فرآیند اجازه می‌گیرد منبع را نگه دارد و درخواست منابع جدید
۴. Circular wait ← انتظار چرخشی ایجاد می‌شود (سیکل انتظار به وجود می‌آید) فرآیند ۱ منتظر منبع

تست: کدام یک از شرایط زیر جزء شرایط بن بست نیست؟

۱۱ فرآیندهای غیر قابل بازگشت ۱۲ سیکل انتظار ۱۳ تخصیص ناقص ۱۴ Preemtion ✓

نه No Preemtion چنین

تخصیص ناقص همان Hold & wait می‌باشد

استراتژی‌های بن بست (راه‌حل‌های بن بست):

۱. ostrich (شترمرغ) ← بی‌خیال شدن. آنرا ندید بگیریم
۲. Detection & Recovery کشف و ترمیم
۳. Prevention پیشگیری. جلوگیری
۴. Avoidance اجتناب (پویا) ← (Dynamic Av

۲) کشف و ترمیم: بن‌بست را کشف کنیم سپس آن را ترمیم کنیم. مثل روش قبلی هیچ کاری برای جلوگیری از رخداد بن‌بست ندارد

چگونه آن را کشف می‌کند؟ در یک ساختمان داده (گراف تخصیص منابع) به دنبال سیکل بن‌بست می‌گردد

کی انجام می‌شود؟ هر وقت یک درخواست و انتظار جدید شکل بگیرد. ولی این راه حل بسیار پرهزینه است. سربار محاسباتی آن بالاست چون تعداد فرآیندها زیاد و تعداد منابع بسیار زیاد است و مکرراً این اتفاق طولانی انجام می‌شود

تشخیص دادیم بن‌بست است حال Recover می‌کنیم که یک فرآیند را می‌کشیم

کدام فرآیند را بکشیم:

فرآیند جوان را بکشیم چون فرآیندهای قدیمی از منابع بسیاری استفاده کردند

این روش می‌تواند به طور کلی بن‌بست را حل کند و در این روش باز هم بن‌بست به وجود می‌آید

PCB جز منابع دیدنی نیست که موجب بن‌بست می‌شود. ظرفیت محدود PCB موجب بن‌بست می‌شود. از این منابع در سیستم کم هست

3) Prevention: پرهزینه‌ترین راهکار بن‌بست است.

هر کاری بکنیم باز هم در حالت عمومی راه حلی نداریم

کاری کنیم که هیچ‌کدام یکی از ۴ شرط بن‌بست تحقق نیابد و لذا هزینه زیادی را متحمل می‌شوند. یک سری منابع را هدر می‌دهد که شاید دچار بن‌بست نشود

روش بانکداری در جدول صفحه بعد توضیح داده می‌شود

شرط	روش پیشگیری	دلایل عدم موفقیت کامل روش
Mutual Exclusion انحصار متقابل	منابع را به روی Spool, Disk می‌کنیم (مثال: Spool کردن چاپگر، رسانه‌ها و ...)	۱) این کار برای همه منابع امکان پذیر نیست. مثلاً رکوردهای DB یا PCB ها ... ۲) غیرممکن است. چون خود فضای Spool روی دیسک محدود است می‌تواند منجر به بن‌بست شود
No Preemption انحصاری	منابع را به زور پس بگیریم	غیرممکن است (چون انحصاری است) گرفتن منابع = کشتن فرایند
Hold & wait نگهداری و انتظار	۱) منابع نگهداری شده را آزاد کند و سپس دوباره آنها را به همراه منبع جدید درخواست کند → تخصیص فقط Hold، منتظر فقط wait ۲) فرآیند در ابتدا همه منابع مورد نیاز خود (تا انتهای فرآیند) را با هم درخواست کند → یا فقط Hold / یا فقط wait	غیرممکن است (چون انحصاری) ۱) بسیار پرهزینه است. منابع را هدر می‌دهد ۲) غیرممکن است. پیش‌بینی آینده نیاز دارد. پیش‌بینی اینکه فرآیندها چه منابعی نیاز دارند
Circular wait انتظار چرخشی	منابع را شماره‌گذاری کنیم. فرآیندها باید منابع را به ترتیب شماره درخواست کنند (مثلاً صعودی) و چرخه تشکیل می‌شود اما هرگز به بن‌بست تبدیل نمی‌شود	۱) پرهزینه است چون منابع را اگر خارج ترتیب نیاز داشته باشیم هدر می‌دهیم ۲) غیرممکن است. پیش‌بینی آینده نیاز دارد. تا بداند در آینده چه منبعی را خارج ترتیب نیاز دارد و پیشاپیش آن را اخذ کند

جلسه یازدهم

Avoidance (اجتناب)

by: Dijkstra

الگوریتم بانکدار (Banker)

یک بانکدار فرض کنید تعداد E موجودی اولیه (پول) دارد. یک تعدادی مشتری دارد، هرکدام یک سقف اعتبار دارد. سقف اعتبار حداکثر وامی است که می‌تواند درخواست کند و وام بگیرند. حال اگر یک مشتری درخواست جدیدی بدهد و ما نتوانیم آن را با تخصیص جدید وام‌های قبلیش هم پس نمی‌دهد (حق دارد پس نخواهد نداد)

فرض کنیم بانکدار ۴ مشتری A و B و C و D دارد.

مشتری	سقف اعتبار Max
A	8
B	5
C	3
D	4

مشتری	الان وام دریافتی Allocated
A	4
B	2
C	1
D	1

جمع P = 8

A = E - P = 10 - 8 = 2

Available در دسترس

E = 10 موجودی اولیه Exist

وضعیت الف

قبل از تخصیص بررسی می‌کنیم در صورت تخصیص وام در وضعیت ب می‌شویم

درخواست جدید می‌دهد برای دریافت یک واحد وام

مشتری	Alloc.
A	4
B	3
C	1
D	1

P = 9 A = 1

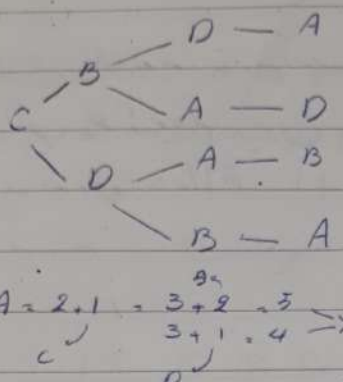
ریسک بن بست = ناامن

وضعیت ب می‌تواند به وجود بیاید چون می‌شود مشتری‌ها وام‌های خود را پس دهند و شاید به وجود نشود ولی خطر وجود دارد که تمام مشتری‌ها درخواست جدید بیش از یک واحد بخواهند و همه به آن‌ها جواب wait بدهند و بن بست می‌شود در نتیجه پس وضعیت ب ناامن است ولی می‌تواند بن بست رود

وضعیت ب به این شکل باشد:

Alloc	درخواست
5	3
2	3
1	-
1	3

باز هم بن بست می‌شود چون اگر مشتری C وام خود را هم پس دهد A = 2 خواهد شد، درخواست هیچ‌کدام از مشتری‌ها را نمی‌توان پاسخ داد و هیچ‌کدام آزاد نمی‌شود



مشترا من (راه حل)

باید تا آخر برود — در وضعیت الف

در نتیجه وضعیت بدالف امن می‌باشد

$A = 2,1 = 3+2,5 = 3+1,4$ ⇒ هر فرآیند را خواست آزاد کرد

C D

- برای پیدا کردن بیشترین حالت امن حیث بدترین حالت را بگیریم اگر توانستیم راه حل پیدا کنیم آن بیشترین امن خواهد بود

O.S.	: نگدار
فرآیند	مشتری
انواع منابع	منبع = بول
بن بست	درشگی

- الگوریتم بانکدار برای منابع چند گانه :

مثال زیر را در نظر بگیرید که در آن 5 فرآیند و 4 منبع داریم

(11) تست شماره 8 بن بست

منابع تخصیص یافته (Alloc)

منابع هنوز مورد نیاز NEED

MAX = Alloc + Need

A = E - P

	R_1	R_2	R_3	R_4
منابع موجود P :	4	3	2	2
کل منابع E :	5	3	4	2

⇒ A = 1 0 2 0

	NEED				A			
P_0 :	1	1	0	0	<	1	0	2	0
P_1 :	2	1	1	2	<	1	0	2	0
P_2 :	3	1	0	0	<	1	0	2	0
✓ P_3	0	0	(1)	0	<	1	0	(2)	0 +
						1	1	0	1

2 1 2 1 → A جدید

الآن با A جدید دو فرآیند P_0 , P_4 را بردیم

اگر P_4 را بردیم چون Alloc آن تماماً ۰ است چیزی ندارد، ما بعد وی P_1

در جدول Alloc ۱۱ و ۳ را به ما می‌دهد بنابراین P_0 را می‌بریم

```
  2 1 2 1
+ 3 0 1 1
---------
  5 1 3 2  →  A جدید
```

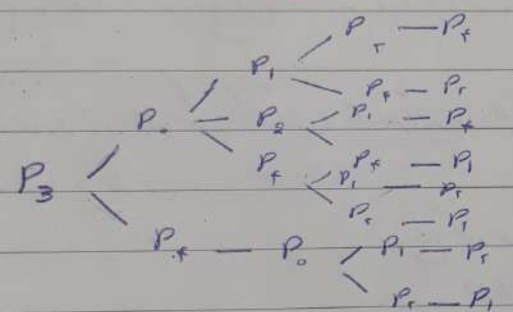
با A جدید چون از NEEd بقیه (P_1 و P_2 و P_4) بزرگتر است ⇐ امن می‌باشد

در نتیجه گزینه ۴ صحیح است

اگرچه لازم نیست:

```
P1 :  2 1 1 2   ≤   5 1 3 2
                  + 0 1 0 0
                  ---------
P2 :  3 1 0 0   ≤   5 2 3 2
                  + 0 1 1 0
                  ---------
P4 :  2 1 1 0   ≤   5 3 4 2
                  ---------
      موجودی اولیه E  ⇐  5 3 4 2
```

II. تعداد مسیرهای امن را در تست فوق تشخیص دهید



در نتیجه ۸ مسیر وجود دارد

۳. در تست ۱ فرآیند P_1 و P_4 هرکدام درخواست ۱ واحد از منبع R_2 را نموده‌اند طبق الگوریتم بانکدار،

۱. تخصیص می‌دهیم چون بن‌بست می‌شود
۲. تخصیص نمی‌دهیم چون ناامن می‌شود ✓
۳. تخصیص نمی‌دهیم چون منبع موجود نداریم
۴. تخصیص می‌دهیم چون امن است
۵. تخصیص می‌دهیم چون بن‌بست نیست

NEED

P	R_2	R_2
P_0	0	1
P_1	~~1~~ 0	~~2~~ 1
P_2	0	1
P_3	1	0
P_4	~~1~~ 0	~~2~~ 1

```
        5  3  4  2
P2 :    4  3  2̶ 2
                4
----------------
A →     1  0  0  0
```

۱۴ فرض کنید که در تست ۳ به تخصیص‌ها صورت گرفته است (یعنی ۱ واحد R_2 به ترتیب به P_1 و P_4 داده‌ایم) حال فرض کنید P_3 درخواست ۱ واحد R_2 و ۴ فرآیند دیگر هرکدام درخواست ۱ واحد R_1 نموده‌اند
حال چه وضعیتی پیش آمده است

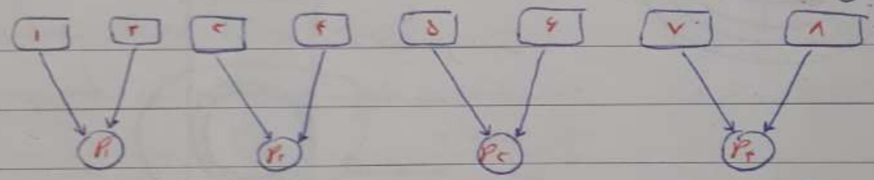
۱۱ بن‌بست است ✓ ۱۲ غیر ممکن است ۱۳ ناامن است ۱۴ امن است

ناامن بود ← درخواست آمد، چون بن‌بست شده

	R_0	R_1	R_2	R_3
P_0	۱	۱	۰	۰
P_1	۲	۱	۰	۲
P_2	۳	۱	۰	۰
P_3	۰	۰	۱	۰
P_4	۲	۱	۱	۰

- چون باز هم به بیش‌تر نیاز داریم و max را نداریم باز هم این الگوریتم نتوانست بن‌بست را حل کند

مثال) فرض کنید که ۴ فرآیند داریم و ۸ منبع قابل استفاده مجدد (یکسان هستند) اگر هر فرآیند حداکثر ۳ منبع نیاز داشته باشد آیا احتمال بن‌بست وجود دارد؟



هرکدام ۲ تا گرفته درخواست سومی را دارند ⟸ بن‌بست

- در مسئله فوق تنها یک وضعیت بن‌بست وجود دارد

- در تست فوق اگر ۹ منبع وجود داشت در سیستم بن‌بست وجود نداشت

- فرض کنید که m تا منبع داریم و n تا فرآیند که هرکدام حداکثر k منبع نیاز دارد شرط اینکه سیستم عاری از بن‌بست باشد چیست؟

بدترین حالت بن‌بست

① ② — — P_n (در بدترین شرایط) بن‌بست ← $n*(k-1)=m$

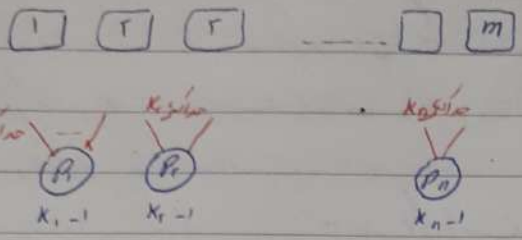
$k-1$ بگیرد، $k-1$ بگیرد، $k-1$ بگیرد

درخواست منبع kام

عاری از بن‌بست ← $n*(k-1)<m$

$nk-n<m \Rightarrow nk<n+m$

کل نیاز (مجموع نیاز)



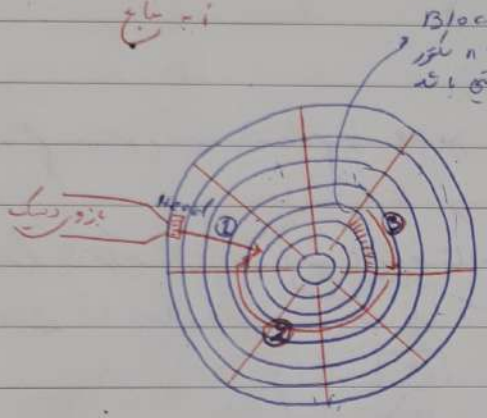
$(k_1 - 1) + (k_2 - 1) + \dots (k_n - 1) < m$ حاصل این است

$\sum_{i=1}^{n} k_i - n < m \Rightarrow \sum_{i=1}^{n} k_i < n + m$

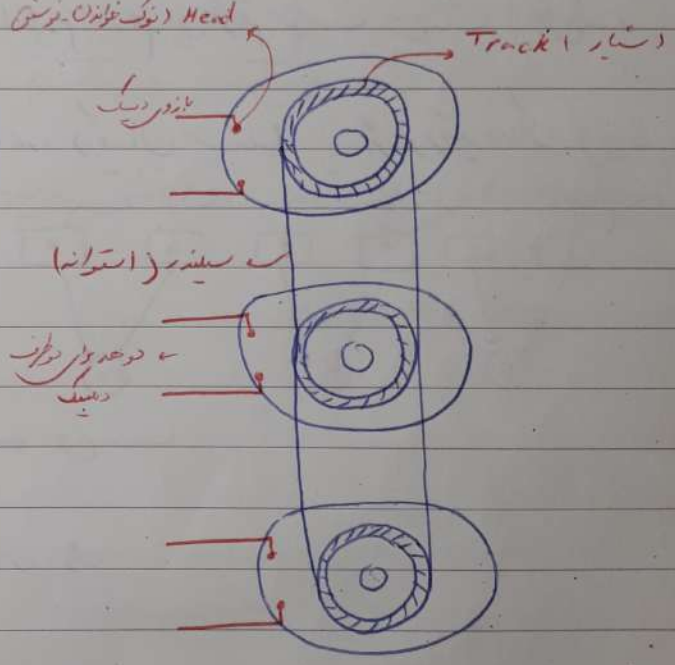
$\sum_{i=1}^{n} Req(i) < n + m$

حداکثر درخواست فرآیند

به جایی



DISK (دیسک)



خال ۷ تا

1. Seek Time · زمان جستجو

مدت زمانی که طول می‌کشد تا حداز مکان اولیه خود به سیاه مورد نظر برسد

2. Rotational latency . تاخیر چرخشی

" Delay دورانی

از لحظه‌ای که هد روی سیاه قرار می‌گیرد ؛ مدت زمانی که طول می‌کشد تا ابتدای بلوک (ای) مورد نظر زیر هد قرار بگیرد (دورانی چرخش دیسک)

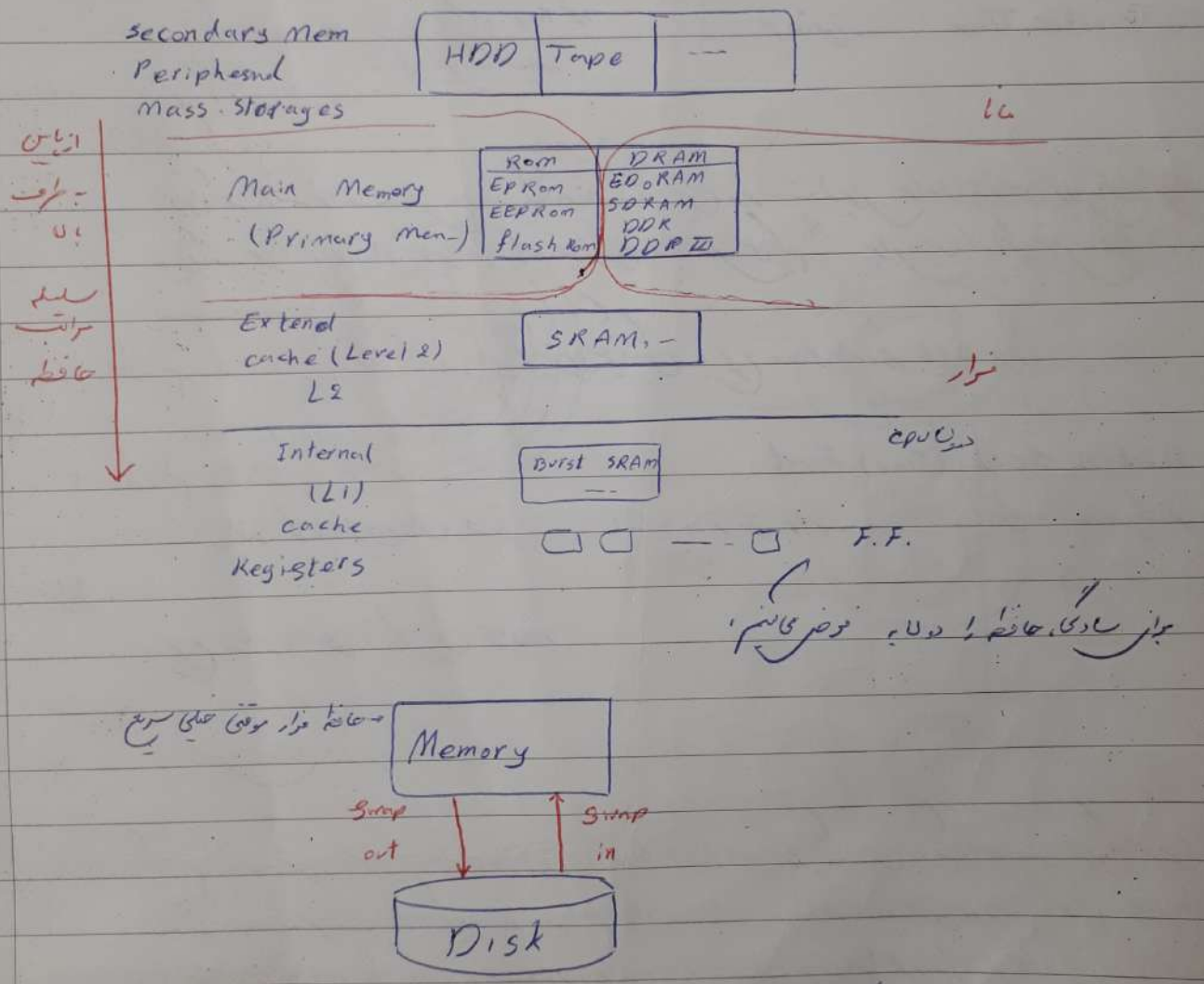
3. Transfer Time (R/W) زمان انتقال (خواندن-نوشتن)

از لحظه‌ای که ابتدای بلوک زیر هد قرار می‌گیرد ؛ مدت زمانی که طول می‌کشد تا دیسک بچرخد و بلوک (ها) از زیر هد بگذرند و عمل خواندن یا نوشتن انجام شود

۴. الگوریتم CSCAN : حالت آسانسوری دارد و برای زمان برگشت، درخواستها عمل نمی‌کند
دارای حداقل زمان درخواست دیسک می‌باشد
ثبت: کدام یک از الگوریتم‌های زیر حداکثر زمان درخواستهای دیسک را حداقل می‌کند

✓ CSCAN SCAN SSF FIFO

فصل ۴ مدیریت حافظه Memory Management



برای سادگی، حافظه را دو لایه فرض می‌کنیم

هدف از مدیریت حافظه چیست؟ یک جواب: استفاده بهینه، حفاظت شده، و این از حافظه است

تکنیک های مدیریت حافظه:

۱. تک برنامگی ۲. چند برنامگی با پارتیشن های ثابت

۳. مبادله (Swapping) [با حفظ پارتیشن بندی تغییر]

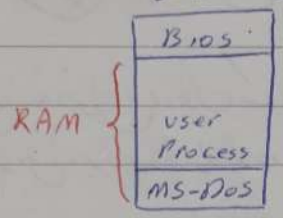
۴. حافظه مجازی:

۴.۱ صفحه بندی Paging

۴.۲ قطعه بندی Segmentation

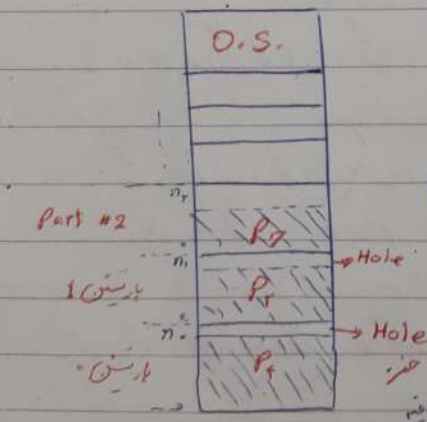
۴.۳ ترکیب Segmentation & paging

۱. تک برنامگی: در هر لحظه فقط یک فرآیند کاربر در کنار سیستم عامل در حافظه قرار می‌گیرد و فرآیندهای دیگر باید صبر کنند تا آن فرآیند تمام شود سپس وارد حافظه شوند



- چون اجازه چند برنامگی نمی‌دهد منسوخ شده است
- ممکن است آن فرآیندی که داخل حافظه است ۱۰٪ از آن را بگیرد بنابراین اتلاف حافظه آن زیاد است بنابراین روش خوبی نیست

۲. چند برنامگی: در این روش حافظه را پارتیشن بندی می‌کنیم تعداد و اندازه پارتیشن ها دلخواه است



مزایا:

۱. چند برنامگی دارد

۲. یک فرآیند باید به طور کامل و یکجا درون یک پارتیشن قرار گیرد (اجازه نداریم نصف یک فرآیند را در یک پارتیشن و بقیه را در پارتیشن دیگر)

۳. اجازه نداریم بیش از یک فرآیند در یک پارتیشن قرار دهیم

فضای خالی بی مصرف درون پارتیشن (درون فضای تخصیص یافته برای فرآیند)

معایب:

بزرگترین فرآیندی که امکان اجرا دارد اندازه پارتیشن بزرگ است (فرآیند بزرگتر از آن نمی‌تواند اجرا شود)

چون اندازه فرآیندها تصادفی است (دلخواه است) در نتیجه وقتی درون یک پارتیشن قرار می‌گیرد یک فضای خالی بی مصرف به نام Hole درون پارتیشن ایجاد می‌شود - این پدیده Internal Fragmentation یا تکه تکه شدن داخلی نامیده می‌شود

حافظه به شدت هدر می‌رود

- تخصیص پارتیشن های حافظه به فرآیندها چگونه صورت می‌گیرد ؟

پیشنهاد اول: هر پارتیشن صف مجزای خود را داشته باشد (به تفکیک سایز فرآیندها را در صف قرار دهد)
مثلاً در صف پارتیشن‌ها و وابسته که بیشتر به آن‌ها می‌خورد

اشکال:
ممکن است بعضی از صف‌ها خیلی شلوغ در حالی که بعضی از پارتیشن‌ها اصلاً خالی باشند

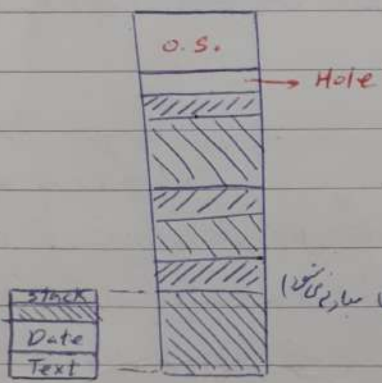
پیشنهاد دوم: یک صف داشته باشیم و همه فرآیندها یکجا بایستند در صورت Best feet عمل کنیم

اشکال) فرآیندهای کوچکتر دچار قحطی می‌شود (اگر Best feet انتخاب کنیم)

- برای حذف قحطی می‌توان یک cunter داشته باشیم برای فرآیندها که هر وقت k بار حق یک فرآیند را از بین بردیم آن را وارد حافظه بگذارد و آن را Run کنیم

این روش نیز منسوخ شده است

3) مبادله (Swapping) : فرآیندها را به حافظه می‌آوریم و تا جدا کردن آن‌ها را کنار هم در حافظه بدون فاصله قرار می‌دهیم



- باز هم فرآیند کامل و یکجا به حافظه بار می‌شود

- یک فرآیند در طی اجرا ممکن است بارها به حافظه swap in بشود و یا از حافظه به Disk swap out شود (بین حافظه و دیسک جابجا می‌شود)

- هر بار که فرآیند به حافظه swap in می‌شود ممکن است در یک آدرس جدید بار شود

• چرا دوست داریم فرآیندها را suspend می‌کنیم؟ چون می‌خواهیم درجه چند برنامگی بالا برود در واقع از ظرفیت حافظه بیشتر استفاده شود

نکته مهم: از آنجایی که اندازه فرآیندها متفاوت است swapping باعث می‌شود پس از مدتی حافظه پر از حفره‌هایی با اندازه‌های مختلف شود. این حفره‌ها خارج از فضای تخصیص یافته به فرآیندهای مجاور است. به این پدیده External fragmentation گویند
تکه تکه شدن خارجی

- معایب مبادله:

الف) معایبی که حل شدنی نیست:
1) فرآیندهای بزرگتر از حافظه فیزیکی (منهای o.s.) قابل اجرا نیستند
2) بیهوده یک فرآیند طولانی کامل به حافظه می‌آورد بهتر بود بخش‌های مورد نیاز فرآیند به حافظه وارد شود که در حال حاضر مورد نیاز است
3) External fragmentation دارد

نکته. بعضی‌ها به غلط فکر می‌کنند که مشکل External fragmentation حل می‌شود و اسم این را حل را memory compaction (defragmentation) می‌گذارند

3× 32 M × 60 ms
$32 \times 10^6 \times 60 \times 10^{-9} = 2 \sec$
ده‌ها صدها گیگابایت

پس هیچ مهندس عاقلی از این روش استفاده نمی‌کند

ب) معایب قابل حل مبادله:
1. حفاظت Protection
2. جابجایی Relocation

حفاظت: فرآیندها در حافظه کنار هم نشسته‌اند و حافظه یک فضای آدرس خطی یکپارچه است در نتیجه فرآیندها فضای آدرس هم دسترسی را دارند و باید جلوی دسترسی‌های غیرمجاز را بگیریم

جابجایی: هر بار که فرآیند بار می‌شود در آدرس جدید قرار می‌گیرد (آدرس شروع فرآیند در حافظه را Base می‌نامیم) تمام آدرس‌های درون حافظه تغییر کرده و از اعتبار می‌افتند. تمام آدرس‌های درون برنامه با فرض اینکه از صفر شروع می‌شوند محاسبه شده‌اند (آدرس نسبی (منطقی)) هستند

آدرس فیزیکی = Base + آدرس منطقی

راه حل‌های این معایب

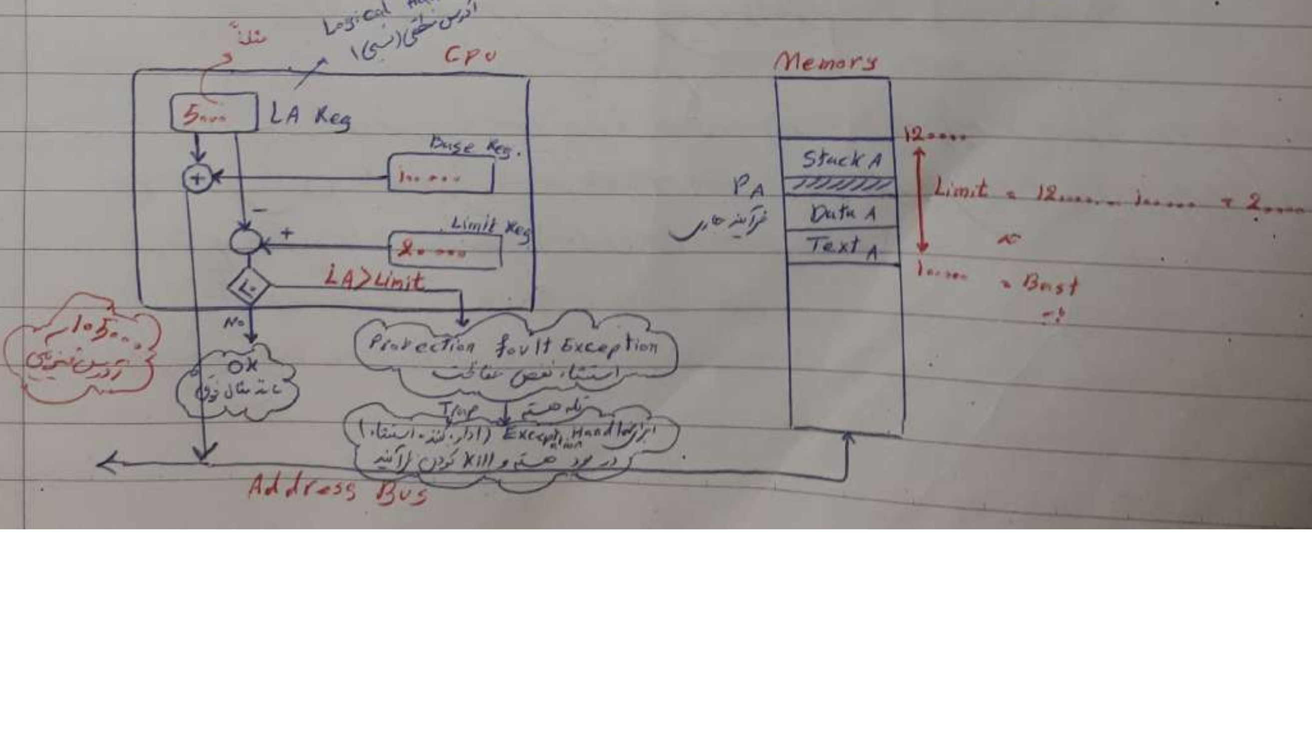
۲. راه‌حل‌های حفاظت:
- رمزهای ۴ بیتی برای بلوک‌های حافظه
- رجیستر Limit (ثبات حد) و تفریق کننده.

۲. راه‌حل‌های جابجایی:
- جمع کردن همه آدرس‌های درون برنامه با Base توسط O.S. پس از Swap in
- رجیستر Base (ثبات پایه) و جمع کننده.

• راه حل اول حفاظت: حافظه را به بلوک‌های مساوی تقسیم کنید برای هر Block یک رمز ۴ بیتی قرارداد. فرض کنید یک فرآیند در ۱۰ بلاک قرار گرفته است و مثلاً رمز هر این ۱۰ بلاک برابر (۰۱۰۱) است این فرآیند یک کلید ۴ بیتی دارد که آن هم برابر (۰۱۰۱) است CPU یک رجیستر ۴ بیتی دارد که کلید فرآیند در حال اجرا درون آن قرار می‌گیرد هرگاه این فرآیند بخواهد به آدرسی درون یک بلاک غیرمجاز (که رمز آن با کلید فرآیند متفاوت است) مراجعه کند CPU آن دستورالعمل را اجرا نمی‌کند و استثناء نقض حفاظتی رخ می‌دهد

• یک راه حل برای جابجایی: هنگامی که سیستم عامل فرآیندی را به حافظه بار می‌کند همه آدرس‌های درون آن را با Base جمع می‌کند
این راه حل دو عیب دارد
a) بسیار کند است که نرم افزار بخواهد این همه آدرس را با Base جمع کند
b) سیستم عامل از کجا بداند کدام بایت‌های برنامه آدرس اند (بعضی‌ها پیشنهاد کردند Linker آدرس حافظه‌ها را به همراه به O.S. دهد)

راه حل خوب برای مشکلات حفاظت و جابجایی: Base & Limit Registers



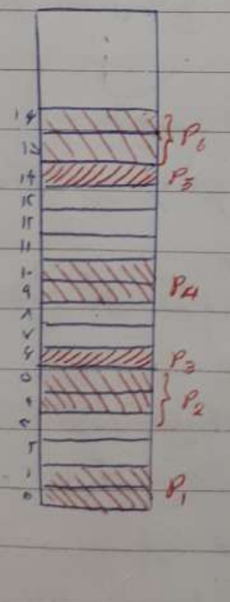
- چرا این روش سریع است ؟ چون این عمل را به صورت Parallel و به صورت Pipeline با بقیه را انجام می‌دهد
- چه کسی محتویات رجیسترهای Base و Limit را پر می‌کند؟ سیستم عامل
- چه موقع این کار را انجام می‌دهد: هنگامی که بخواهد به یک فرآیند Swich کند (هنگام Dispach)
- سیستم عامل این مقادیر را از کجا می‌آورد ؟ از PCB فرآیند
- چه کسی این مقادیر را در PCB قرار داد. ؟ خود سیستم عامل
- چه موقع ؟ هر وقت فرآیند را در حافظه Swapin می‌کند

روش‌های مدیریت فضاهای پر و خالی حافظه:

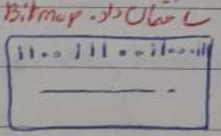
سیستم عامل از کجا می‌فهمد کجا پر و کجا خالی است ؟ برای این کار باید یک ساختمان داده داشته باشیم ← دو پیشنهاد داده شده است

۱. روش نگاشت بیتی Bit Map ۲. روش لیست پیوندی دوطرفه (Doble Linked List)

در هر دو روش ابتدا حافظه را به بلاک‌های ساوی تقسیم و آنها را شماره‌گذاری می‌کنیم



ساختمان داده Bitmap



روش نگاشت بیتی: از پایین حافظه شروع کنید
هر بلوک پر ← ۱
هر بلوک خالی ← ۰

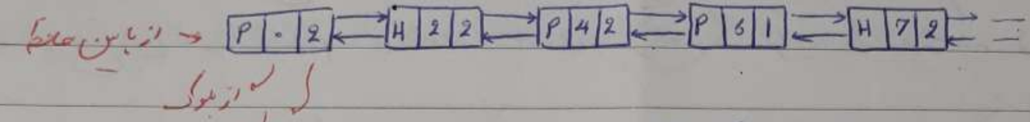
B: اندازه بلوک برحسب بیت (Bit)

$$\frac{1}{B+1} = \text{سربار حافظه نگاشت بیتی}$$

عیب: به شمارش صفرها نیاز دارد هنگامی که می‌خواهیم برای یک فرآیند یک حفره مناسب پیدا کنیم و آن‌را بارگذاری کنیم باید صفرهای پشت سر هم را بشماریم که وقت‌گیر است

P: Process
H: Hole

روش لیست پیوندی دوطرفه: در این روش از پایین حافظه شروع می‌کنیم



- چرا لیست پیوندی دوطرفه است؟ فرض کنید فرآیند P_4 می‌خواهد از حافظه خارج شود باید با حفره‌های ۲ طرفش ترکیب شود بنابراین باید دو پوینتر با نودهای دوطرفش داشته باشد.

۳. **الگوریتم‌های تخصیص حافظه:** فرض کنید می‌خواهیم یک فرآیند را وارد حافظه کنیم و چندین حفره کوچک و بزرگ پراکنده داریم حال فرآیند را در کدام حفره قرار می‌دهیم. برای این کار ۶ الگوریتم پیشنهاد شده است

1. First Fit (اولین Fit)
2. Best Fit (بهترین Fit)
3. Worst Fit (بدترین Fit)
4. Quick Fit (Fit سریع)
5. Next Fit (Fit بعدی)
6. Buddy system (سیستم رفاقتی)

First Fit: از پایین حافظه شروع کن فرآیند را در اولین حفره‌ای بگذار که جا می‌شود (دنبال سایز خیلی مناسب نیست) مزیت: سرعت عیب: تخصیص حافظه کارآمد نیست حفره مناسبی پیدا کند تصادفی است

Best Fit: کل حافظه را بگرد بهترین حفره را پیدا کن (کوچکترین حفره‌ای که فرآیند در آن جا می‌شود را پیدا کن)

مزیت: تخصیص حافظه کارآمدتر است عیب: کند است (نیاز به جستجو دارد) و حافظه را پر از حفره‌های خیلی ریز می‌کند

Worst Fit: کل حافظه را بگرد و فرآیند را در بزرگترین حفره موجود قرار بده.

معایب: کند است، حفره‌های بزرگ را از بین می‌برد در نتیجه فرآیندهای بزرگ نمی‌توانند اجرا شود

Next Fit: دقیقاً مثل First Fit است با این تفاوت که هر بار از پایین حافظه شروع نمی‌کند بلکه از محل آخرین تخصیص شروع می‌کند. شبیه‌سازی نشان می‌دهد که First Fit بهتر از Next Fit عمل می‌کند

QuickFit: در اینجا طرفدار Best Fit هستیم اما می‌خواهیم این کار را سریع انجام دهیم بدین منظور حفره‌ها را دسته‌بندی می‌کنیم (چندین List دسته‌بندی می‌کنیم به تفکیک سایز) حال پیدا کردن حفره مناسب سریع است چون راست به لیست حفره مورد نظر می‌رویم

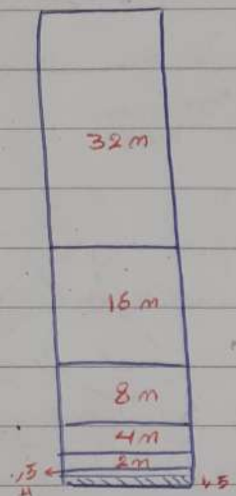
جلسه دوازدهم

عیب: آن سرعتی را در هنگام رفت دارند ولی در هنگام برگشت چون احتیاج به merg کردن دارد سرعت آن کم می‌شود

سیستم رفاقتی: همه جزءها دارای اندازه‌ای هستند که دارای توانی از ۲ است و با نصف کردن جزءهای بزرگتر اجرا می‌شود

فرض کنید دو فرآیند به اندازه‌های ۵ mB را و 6.5 mB در حافظه‌ای به اندازه 64 mB قرار دهیم

جزءهای مجاور در صورتی ترکیب می‌شوند که سایز آنها توانی از ۲ باشند



حافظه مجازی (Virtual Memory)

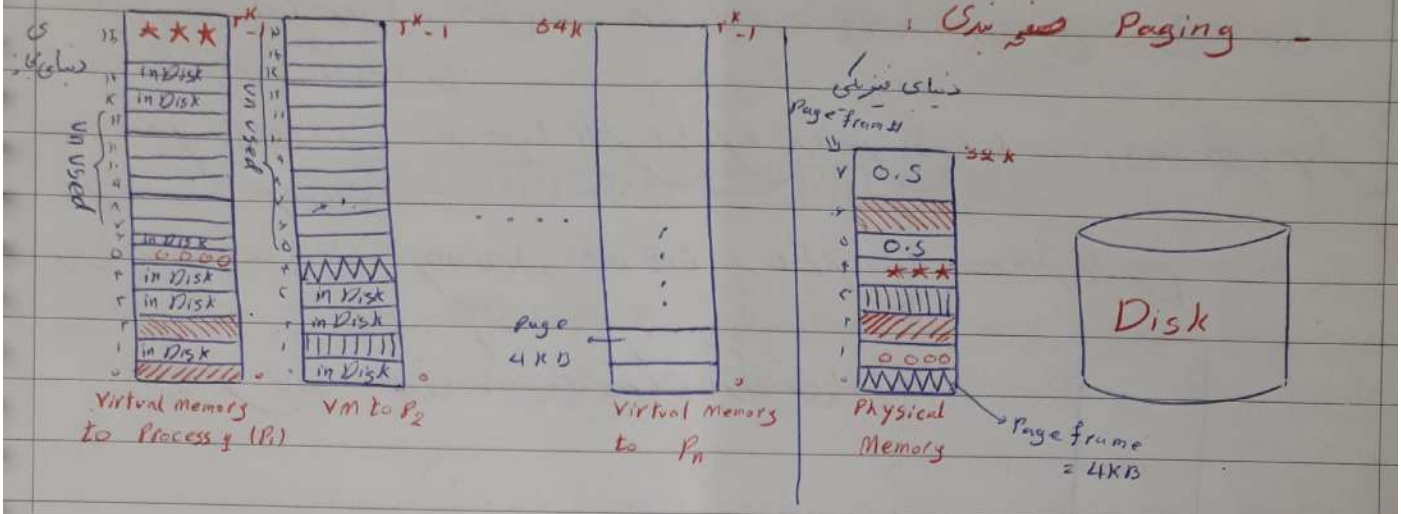
دلایل نیاز به حافظه مجازی:

1. امکان اجرای برنامه‌های بزرگتر از حافظه فیزیکی
2. عدم انتقال بیهوده کل فرآیندهای طول‌های به حافظه (فقط بخش‌های [صفحات یا قطعات] مورد نیاز فعلی به حافظه منتقل می‌شود
3. کاهش مشکل fragmentation تا حد امکان
4. استفاده بهینه از حافظه

برای این تکنیک کار ۲ تکنیک صفحه‌بندی، قطعه‌بندی و ترکیب روش ۱ و ۲ بیشنهاد شد.

قطعه Segment	صفحه Page
اندازه متفاوت	اندازه یکسان
اندازه متغیر	اندازه ثابت
اندازه‌اش می‌تواند بسیار بزرگ (چندین MB) باشد	اندازه نسبتاً کوچک (۰.۵ تا ۱۰ KB)
معمولاً تعداد اندک	معمولاً تعداد بسیار زیاد
خود برنامه‌نویس قطعات را تعریف می‌کند	برنامه‌نویس از وجود صفحات بی‌خبر است

Paging - صفحه بندی :



- Virtual Memory کاملاً مجازی است و نه بخشی از دیسک
- هرفرآیند برای خود یک Virtual memory مجزا دارد
- اندازه Virtual memory می‌تواند بزرگتر از حافظه‌ی فیزیکی باشد

- چگونه می‌توان اندازه Virtual Memory را بدست آورد؟ اندازه VM هیچ ربطی به اندازه حافظه‌ی فیزیکی ندارند بلکه VM هرفرآیند یک فضای خطی یکپارچه از ۰ تا $2^k - 1$ می‌باشد (با اندازه 2^k) که k تعداد بیت Address Bus cpu می‌باشد

مثلاً cpu شما دارای آدرس Bus 32 بیتی است اندازه VM برای هرفرآیند $2^{32} = 4GB$ است

2^{10} K
2^{20} M
2^{30} G

$2^{32} = 2^2 \times 2^{30} = 4\ GB$

- نحوه پیاده سازی با استفاده از صفحه بندی به شرح زیر است :

(برای سادگی حافظه اصلی 32KB اندازه صفحه را 4KB و تعداد بیتهای Address Bus را 16 بیت فرض کنید)

تعریف page (صفحه) : حافظه مجازی هرفرآیند را به تکه‌های ثابت، یکسان و نسبتاً کوچکی به نام صفحه تقسیم می‌کنیم (این کار در درون سیستم انجام می‌شود و برنامه‌نویس خبر ندارد)

تعداد صفحه حافظه مجازی = $\frac{64\ KB}{4\ KB} = 16\ Page$ $\quad k=16 \Rightarrow Vm = 64\ KB$

Page frame (قاب صفحه) : حافظه فیزیکی را به تکه هایی با اندازه صفحه تقسیم می‌کنیم و به هرکدام یک قاب صفحه می‌گوییم

تعداد قاب صفحه = $\frac{32k}{4k} = 8$ page frames

آدرس مجازی Virtual Address : آدرس‌هایی که فرآیند با آن سروکار دارند در فضای آدرس مجازی بیان می‌شوند مثلاً در مثال فوق از ۰ تا 64KB ($2^{16}-1$) هستند

نکته : در هنگام اجرای یک فرآیند صفحات مورد نیاز آن (مجموعه کاری که هم اکنون به آن صفحات مراجعه می‌شود) از دیسک به درون قاب‌های صفحه حافظه فیزیکی منتقل می‌شود (بارمی‌شود) بقیه صفحات روی دیسک می‌مانند تا موقع نیاز

آدرس فیزیکی Physical Address : وقتی CPU می‌خواهد به یک کلمه از یک صفحه مراجعه کند در واقع به حافظه فیزیکی مراجعه می‌کند و با فضای آدرس فیزیکی سروکار دارد بنابراین باید آدرس‌های مجازی به فیزیکی ترجمه (نگاشت) شود

به عنوان مثال صفحه ۰ از فرآیند ۱ در درون قاب صفحه ۲ قرار دارد

نکته مهم : صفحاتی از یک فرآیند که به قاب‌های صفحه منتقل می‌شوند می‌توانند بدون هیچ نظمی در حافظه فیزیکی پراکنده شوند بنابراین مشکل External fragmentation حل می‌شود یعنی معنی ندارد که بگوییم بخشی از حافظه فیزیکی خالی و بی مصرف است در صورتی که صفحه‌ای باید به حافظه منتقل شوند هیچ دلیلی برای خالی ماندن حتی یک قاب وجود ندارد

نکته : هنوز مشکل کوچکی به نام internal fragmentation وجود دارد ولی چون مشکل کوچکی است دیگر برای ما مهم نیست چون اندازه فرآیندها دقیقاً مضرب صحیحی از اندازه صفحات نیستند بطور متوسط هر فرآیند $\frac{1}{2}$ (نصف اندازه صفحه) حافظه را هدر می‌دهد که ناچیز است

- هرگاه حافظه فیزیکی پر باشد و به صفحه جدیدی نیاز داشته باشیم مجبوریم یکی از صفحات را که در آن احتیاج داریم از قاب صفحه خارج و صفحه جدید را جایگزین کنیم (به تشخیص سیستم عامل)

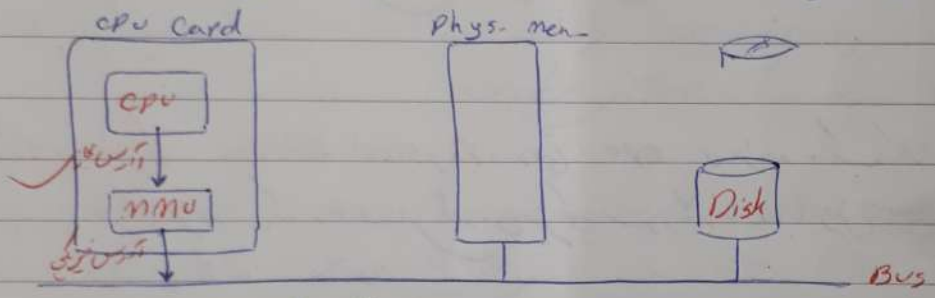
Page fault نقص صفحه : هرگاه فرآیند به آدرسی مراجعه کند که صفحه مربوط به آن در حافظه فیزیکی نیست (در حافظه فیزیکی نیست) Page fault رخ می دهد. این یک تله هسته است و باید سیستم عامل صفحه را در حافظه فیزیکی بارگذاری کند.

- برنامه ای که دچار این پدیده می شود باید بخوابد تا اینکه صفحه اش بیاید

ترجمه آدرس:

ترجمه آدرس توسط چه موجودی انجام می شود؟ ترجمه آدرس نرم افزاری هیچ معنایی ندارد و انجام ترجمه آدرس باید حتماً سخت افزاری باشد

سخت افزار ویژه ای به نام MMU ترجمه آدرس را برعهده دارد که کنار CPU یا حتی روی کارت CPU قرار می گیرد Memory Managment Unit



برای ترجمه آدرس، MMU به جداول صفحه نیاز دارد. هر فرآیند یک جدول صفحه مخصوص خود دارد به عنوان مثال به جدول صفحه فرآیند ۱ نگاه کنید. در این جدول:

۱. شماره صفحات به صورت index است (چون نیازی به جستجو نداشته باشد سرعت ترجمه آدرس بالا رود)
۲. بیت P/A (Present / Absent) نشان می دهد که صفحه در حافظه فیزیکی حاضر یا غایب (روی دیسک یا در راه) حضور / غیاب
۳. شماره قاب صفحه نشان می دهد که صفحه ای حاضر در کجا قرار دارد
۴. بیت R (Referenced) نشان می دهد که آیا اخیراً صفحه مراجعه شده است (۱) یا خیر (۰)
۵. بیت M (Modified) نشان می دهد که آیا صفحه از هنگام بارشدن در حافظه (فیزیکی) تاکنون تغییر کرده است (۱) یا خیر (۰)

Page #	P/A	Page frame #	R	M	Protection	---
۱۵	1	4				
۱۴	۰	۰				
۱۳	۰	۰				
۱۲	۰	۰				
۱۱	۰	۰				
۱۰	۰	۰				
۹	۰	۰				
۸	۰	۰				
۷	۰	۰				
۶	۰	۰				
۵	1	1				
۴	۰	۰				
۳	۰	۰				
index → ۲	1	6				
۱	۰	۰				
۰	1	2				

جدول صفحه فرآیند P_1.

سوال: آدرس 2752H را ترجمه کنید.

تعداد صفحه = $16 = 2^4$

اندازه صفحه = $4K = 2^{12}$

4	12	
0010	0111 0101 0010	VA
Page #	offset (Displacement)	

① ② ③

0110	0111 0101 0010

۱. MMU شماره صفحه را از سمت چپ آدرس مجازی برمی دارد

۲. براساس اندیس مستقیماً به درایه مورد نظر در جدول صفحه مراجعه می کند

۳. به بیت حضور غیاب نگاه می کند اگر صفحه غایب بود (۰) Page fault رخ داده است (تا حالت خواهید فرآیند و انتقال صفحه از Disk به حافظه) در غیر اینصورت شماره قاب صفحه را برمی داریم و به سمت چپ offset جایگزین شماره صفحه می کنیم و آدرس فیزیکی ساخته می شود

جای Page table کجاست؟ درون حافظه اصلی

- مشکلات سیستم فوق

۱. جدول صفحه بسیار غول پیکر است فرض کنید سیستم 32 بیتی است اندازه VM, 4Gb می شود به اندازه صفحات 4KB هر فرآیند 1M (۱۰۲۴) صفحه داریم فرض کنید هر درایه ۴ تا ۸ بایت باشد لذا اندازه هر جدول صفحه ۴ یا ۸ مگابایت می شود این را در تعداد فرآیندها ضرب کنید

۱. دائماً یک جدول صفحه داریم که شاید از اندازه حافظه بزرگتر باشد (!!!) اما دقت کنید جداول خلوتی هستند بیشتر ردیف‌های آن خالی است

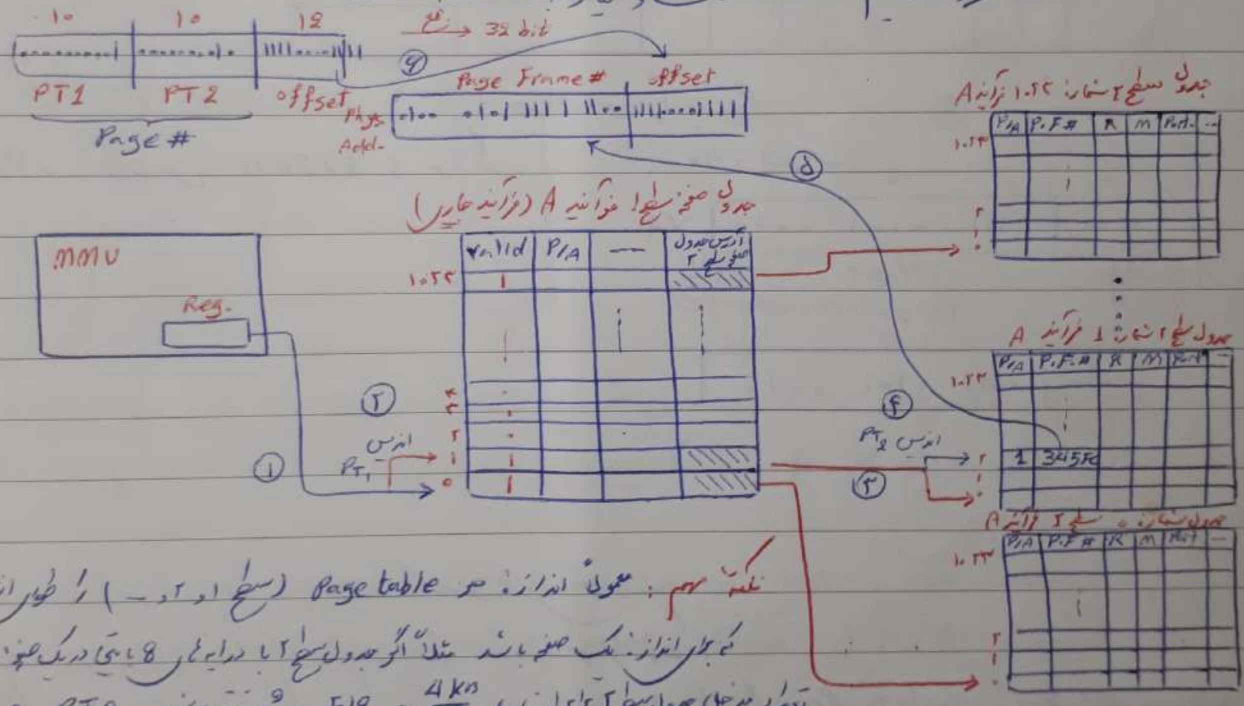
۲. سرعت ترجمه آدرس باید بسیار بالا باشد حتی حافظه برای این کار کند است

برای حل مشکل اول ۲ پیشنهاد ارائه شد:

۱. جداول صفحه چندسطحی ۲. جداول صفحه وارونه

جداول صفحه چندسطحی (دوسطحی)

پیشنهاد می‌شود به عنوان مثال یک جدول 1M درایه‌ای را به ۱۰۲۴ جدول ۱۰۲۴ درایه‌ای (مدخلی) یا مثلاً 512 جدول 2048 درایه‌ای تقسیم کنیم. بسیاری از این جداول کاملاً خالی بوده و حذف خواهند شد (با) همین سیستم index است و نیازی به Search ندارد



نکته مهم: معمولاً اندازه هر Page table (سطح ۱، ۲، ...) را طوری انتخاب می‌کنند که برابر اندازه یک صفحه باشد. مثلاً اگر جدول سطح ۲ با درایه‌های 8 بایتی در یک صفحه 4KB جا شود تعداد مدخل جدول سطح ۲ برابر است با $\frac{4KB}{8B} = 512 = 2^9$ نتیجه می‌شود PT2 9 بیت است

توجه با این هم به صورت اندیسی است و نیازی به Search ندارد اما در ۲ مرحله انجام می‌شود سرعت را دو برابر کند شده است اما ۱۰ برابر جداول صفحه کوچکتر شده که می‌صرفد

جداول صفحه وارونه: در اینجا می‌خواهیم اندازه جدول صفحه را به حداقل برسانیم در کل سیستم فقط یک جدول صفحه به صورت زیر وجود دارد

Page frame #	Valid	Process	Page #	R	M	Protection	--
n-1							
			:				
			:				
			:				
۶							
۵							
۴	1	۲	۷۳۱				
۳	1	۶	11				
۲	1	۵	۲۰				
۱	1	3	17				
۰	1	۶	42				

اندازه حافظه مجازی / اندازه صفحه = n = تعداد فایل

بزرگترین مشکل این روش این است که احتیاج به Search دارد
به کمک ۸ فانکشن می توان سرعت این دستیابی را بالا برد اما به هرحال باز هم کند است و علی
بعضی سیستم ها از آن استفاده می کنند

در هیچیک از روشهای فوق سرعت قابل قبول نیست مشکل سرعت همچنان باقی است تنها راه
حل این مشکل استفاده از Register ها است البته بدهی است که ما آنقدر رجیستر نداریم

رجیسترهای TLB

دانشمندان برای حل مشکل سرعت تلاش کردند تا اطلاعات ترجمه آدرس را (حداقل بخشی از آن را که در حال حاضر مورد
نیاز است) درون رجیسترها قرار دهند معمولاً 16، 32 یا حداکثر 64 رجیستر برای این کار
در نظر می گیرند به رجیسترها نیز نگاه کنید،
این رجیسترها هیچ ترتیبی ندارند چون برالگه و تصادفی به صفحات مراجعه می شود

Valid	Page#	Pageframe#	R	M	Port	--
1	4	32				
1	72	721				
1	126	123				
1	12	451				
۰	4	13				
1	95	97				
		:				

- فقط اطلاعات ترجمه آدرس صفحاتی درون TLB است که اخیراً به این صفحات مراجعه شده است

- طبق اصل مراجعات محلی (Locality References) به احتمال زیاد در آینده نزدیک نیز اغلب مراجعات به این صفحات خواهد بود

- معمولاً 32 یا 64 رجیستر کافی است که ٪90 اوقات TLB Hit اتفاق افتد یعنی MMU فقط می‌تواند فقط با کمک رجیسترهای TLB و بدون مراجعه به جداول صفحه درون حافظه کمک کند آدرس‌ها را ترجمه کند اگر با احتمال پایین (TLBmiss) و TLB شد باید به سراغ جداول صفحه درون حافظه (خیلی‌ها به وارونه) رفته و آدرس را به کندی ترجمه کنیم ولی برای مراجعات بعدی باید TLB آپدیت کنیم و بنابراین مشکل سرعت حل شد

هر سطر TLB یک رجیستر است که چندین فیلد دارد.

چرا TLB نام دارد؟ بازنگاری ترجمه Translation Lookaside Buffer

چون کنار MMU است

نام‌های دیگر این رجیسترها PAR ، Associative Memory (Content Addressable Mem) (CAM)

(Parallel Addressable Register)

طرز کار TLB نام‌های آن را توجیه می‌کند شاید تاکنون فکر کرده باشید مشکل جستجو در TLB وجود دارد راه حل این است که این رجیسترها را به صورت موازی مقایسه کنیم (PAR) مثلاً 64 مقایسه‌گر داریم که هم‌زمان شماره صفحه ورودی را با PageNum درون رجیستر خود مقایسه می‌کند اگر همه صفر شود OR آن‌ها TLB miss اگر یکی از آن‌ها OK باشد TLBHit می‌شود

این حافظه شبیه مغز انسان انجمنی کار می‌کند (Associative memory) شبیه مغز انسان بر اساس محتوا جستجو می‌شود نه آدرس (CAM)

مدیریت TLB :

مسئله این است که اگر TLBFault رخ دهد چه کسی آن را مدیریت می‌کند دانشمندان به تفاهم نرسیده‌اند و ۲ نظریه وجود دارد

نظریه ۱: TLB را سخت‌افزار خود MMU اداره کند (که هسته نیست) و خود MMU به جداول صفحه مراجعه می‌کند (چرا؟) به خاطر سرعت

نظریه ۲: TLB فالت یک که هسته باشد و سیستم عامل آن را اداره کند (چرا؟) به خاطر سرعت

گروه اول طرفداران CISC و گروه دوم طرفداران RISC هستند

رویه ترجمه آدرس:

۱. mmu از سمت چپ آدرس Page Number را بر می دارد و تحویل TLB می دهد

۲. اگر TLB Hit شود شماره قاب را می دهد تا سمت چپ offset بگذارد در غیر اینصورت گام ۳ را می رود

۳. mmu یا OS (بستگی به طراحی دارد) سراغ جدول صفحه داروند یا دوسطحی یا ... می رود سراغ PT_1 می رود بعد PT_2 می رود نهایتاً یکی از ۲ اتفاق می افتد بیت حضور غیاب یا ۰ است یا ۱ اگر ۱ باشد دو کار می کند آپدیت کردن TLB و ترجمه آدرس اگر ۰ باشد PagFault رخ داده است

Page Replacment Alguritms — الگوریتم های جایگزینی صفحه

فرض کنید Page Fault رخ داده است و قاب ها همه صفحه هایی پر اند سیستم عامل کدام صفحه را بیرون بیاندازد و جایگزین کند برای این کار بیش از ۱۰ الگوریتم وجود دارد

1. Bo (Belady optimal) [optimal بهینه]
2. FIFo (First in First out)
3. NRU (Not Recently used), modified NRU — اخیراً استفاده نشده
4. Second chance (دومین شانس شانس مجدد)
5. clock (ساعت)
6. LRU (Least Recently used) — حداقل استفاده اخیر
   - شمارنده ارجاعی
   - ماتریس n×n
7. LFU (NFU) Last Frequently used — کمترین فرکانس استفاده
8. Aging (سالمندی) ⇒ شبیه سازی LRU
9. ws-clock (working set clock) — ساعت مجموعه کاری

الگوریتم optimal: صفحه را بیرون بیانداز که در آینده دورترین ارجاع به آن را خواهی کرد

مزیت: بهینه است — عیب: نمی شود، قابل پیاده سازی نیست ما از آینده خبری نداریم
این الگوریتم شاخص مقایسه است الگوریتم ها خودشان را با این الگوریتم مقایسه می کنند

- الگوریتم FIFO : صفحاتی را بیرون بیاندازد که بیشتر از همه در سیستم حاضر بوده است (زودتر وارد حافظه شده است)

مزیت: ساده است پیاده سازی ساده دارد کافی است یک List پیوندی داشته باشیم

عیب: صفحاتی ممکن است قدیمی باشند ولی مکرراً استفاده شوند

- الگوریتم NRU : در سیستم پریود تعریف کنید سر هر دوره یک کلاک باش از تایمر به صورت وقفه صادر می‌شود دیدیم در جدول صفحه دو بیت R و M داریم سر هر دوره بیت R همه صفحات را Reset (صفر) کنیم (گذشته را فراموش کنیم) ولی دست به بیت M نمی‌زنیم در طی دوره هر صفحه‌ای که رجوع شود MMU به طور اتوماتیک بیت‌های R و M را آپدیت می‌کند حال فرض کنید یک page fault رخ داده است صفحات را به ۴ کلاس تقسیم می‌کنیم:

کلاس	R	M
۰	۰	۰
۱	۰	۱
۲	۱	۰
۳	۱	۱

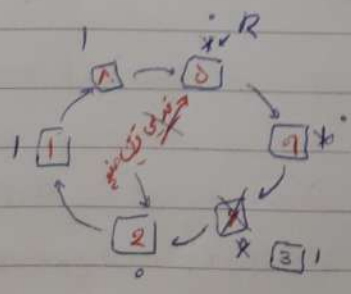
ابتدا سعی می‌کنیم صفحه‌ای را از کلاس ۰ بیرون بیاندازیم (اگر چند تا بود تصادفی یکی را بیرون می‌اندازیم) اگر این کلاس خالی بود به سراغ کلاس بعدی می‌رویم و ...

- الگوریتم شانس مجدد Second Chance : به نحوی تقریباً ترکیب NRU و FIFO می‌باشد

قدیمی‌ترین صفحه را بیرون بیاندازد که اخیراً به آن مراجعه نشده است مانند FIFO یک Link List درست کنید از قدیمی‌ترین صفحه به سمت جدیدترین ← قدیمی‌ترین صفحه نگاه کنید اگر بیت R آن صفر بود آنرا بیرون بیاندازید اگر بیت R آن ۱ بود به این صفحه یک شانس مجدد (صفر) یعنی بیت R آنرا صفر کنید و آنرا از سر لیست (قدیمی) جدا کرده و به انتهای لیست (صفحه جدید) اضافه کنید فرآیند این کار جدیداً به جدول وارد می‌شود R آن ۱ است

عیب سربار update کردن Link List زیاد است

راه حل: الگوریتم ساعت یعنی LinkList را حرفی نکنید



عملکرد این دو الگوریتم یکی است (هر دو یک جواب است)

الگوریتم LRU : صفحای را بیرون بیاندازد که برای آخرین بار در گذشته دورتری به آن مراجعه شده است

این الگوریتم نیاز دارد براساس زمان آخرین مراجعه باشد عیب: پیاده سازی این الگوریتم خیلی سخت است دارای سربار بسیار زیاد است

مزیت: خیلی کارآمد است (محبوب است)

تلاش اول: مثلاً یک Link List درست کنیم تا آخرین صفحای که به آن مراجعه شده است (تلاش سخت است!)

تلاش دوم: هر صفحه یک فیلد 64 بیتی داشته باشد ساعت نرم افزار سیستم هر بار یک شمارنده 64 بیتی باشد با هر کلاک CPU یک واحد اضافه می شود هر صفحه که مراجعه می شود MMU اتوماتیک این کانتر را در فیلد صفحه مورد نظر copy کند (دارای سربار زیادی است و 32 برابر جدول صفحه را بزرگ می کند!) مربوط به دیتا

تلاش سوم: یک ماتریس n×n درست کنیم (تعداد قابهای صفحه n) به هر قابی که مراجعه می کنید سطر مربوط به آن را در جدول 1 کنید سپس ستون مربوط به آنرا صفر کنید هنگام page fault صفحای بیرون بیاندازید که سطر مربوط به آن کوچکتر باشد (این روش بسیار سربار دارد)

الگوریتم LFU : برای هر صفحه در جدول صفحه یک بایت LFU در نظر بگیرید سیستم دارای پریودهای زمانی است با وقفه پایان ساعت بیت R همه صفحه ها را با بایت LFU جمع می کنیم سپس بیت R را 0 می کنیم

با هر نقص صفحه صفحای را بیرون می اندازیم که بایت LFU کوچکتری دارد

تست: فرض کنید بایت LFU یک صفحه 5 است یعنی

۱) در آخرین پریود 5 بار به آن مراجعه شده است ۱۲) در 5 پریود گذشته (متوالی) آن مراجعه شده است

۱۳) تاکنون 5 بار به آن مراجعه شده است (با پریود آن مشخص نیست) ۱۴) در 5 پریود متوالی یا غیر متوالی به آن مراجعه شده است (اما معلوم نیست چند بار)

الگوریتم Aging : شبیه سازی نرم افزاری LRU است و تقریباً تزدیک به LRU عمل می‌کند و تقریباً سربار قابل قبولی دارد (می‌توان ساخت)

مانند روش LFU یک بایت به جدول صفحه اضافه کنید (بایت Aging) سرجمع پدید به جای جمع کردن بیت R ابتدا بایت هر مدل Aging را یک بیت به سمت راست Shift می‌دهیم سپس بیت R را در msb آن قرار می‌دهیم

مثال: بایت Aging صفحه ۱ و ۲ به شرح زیر است

صفحه ۱	۱۰۰۰۰۰۰۰
صفحه ۲	۰۱۱۱۱۱۱۱

الگوریتم Aging صفحه ۲ را بیرون می‌اندازد

نگاهی راجع به صفحه بندی:

working set : صفحاتی است که اخیراً به آن مراجعه شده. هم‌اکنون در حال استفاده است
مجموعه کاری از یک برآمد

مثال: صفحاتی که K (مثلاً ۵) بیت بازدیدشان Aging آنها حداقل یک عدد یک دارد

الگوریتم ws_clock : مانند ساعت به قدیمی‌ترین صفحه نگاه کن در صورتی که بیرون بیاندازد که جزء working set نباشد (مثلاً به ۵ بیت بالای Aging آن نگاه کن)

Belady Anomaly (ناهنجاری Belady) : بعضی از الگوریتم‌های جایگزینی هستند که گاهی از اوقات با افزایش تعداد قاب صفحه تعداد تقصیرهایشان به جای کم شدن، افزایش می‌یابد مانند FIFO
(به سوال کنکور مربوط مراجعه کنید)

(چرا؟) الگوریتم‌هایی که stack هستند مانند optimal و LRU این ناهنجاری را ندارند
زیرا صفحات درون قاب‌های صفحه در حالتی که n قاب صفحه داریم زیرمجموعه‌ای است از صفحات درون قاب‌ها در حالتی که n+1 قاب صفحه داریم

Trashing (کوبیدگی) چیست؟ اگر تعداد قاب‌های تخصیص یافته به یک برآمد کمتر از مجموعه کار آن باشد که هیچگی رخ می‌دهد فرکانس رخداد Page Fault خیلی بالا می‌رود و برنامه بدلیل مراجعه زیاد به حافظه بسیار کند می‌شود

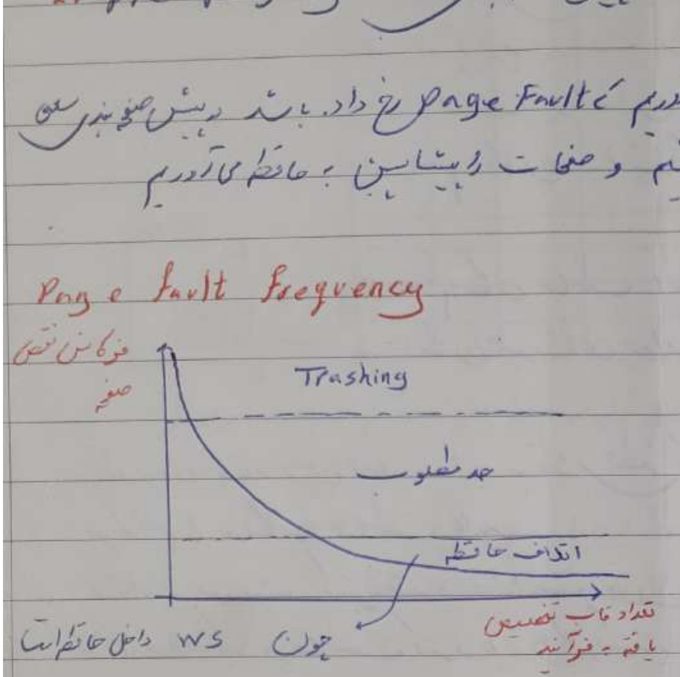
صفحه بندی بر دو نوع است

1) Demand paging صفحه بندی درخواستی
2) pre paging پیش صفحه بندی

- در صفحه بندی درخواستی وقتی صفحه را به حافظه می‌آوریم که Page Fault رخ داده باشد و در پیش صفحه بندی سعی می‌کنیم تخمین و پیش بینی از آینده داشته باشیم و صفحات را پیشاپیش به حافظه می‌آوریم

فرکانس نقص صفحه : P.F.F

Page Fault Frequency



- الگوریتمهای تخصیص صفحه بر دو دسته اند

1) Local Alucation
2) globale Alucation

• در تخصیص local :
در تخصیص محلی اگر Page Fault رخ داده است طبق یکی از الگوریتمهای جایگزینی صفحه فقط صفحات فرآیندی را بیرون بیاندازیم که در آن Page Fault رخ داده است

• در تخصیص globale می‌توانید طبق الگوریتم یکی از صفحات را خارج کنید و اصلاً هم مهم نیست که مربوط به کدام فرایند است

اندازه صفحه : page size

سربار صفحه چیست؟ صفحه بندی دو نوع سربار دارد - یکی حاصل از سربار جداول است $\frac{S}{P} \times e$
- یکی سربار تکه تکه شدن داخلی $\frac{P}{2}$

واحد بیت:
- S : متوسط اندازه فرآیند در سیستم
- P : اندازه صفحه
- e : اندازه درایه جدول صفحه

$$\text{سربار کل} = \frac{P}{2} + \frac{Se}{P}$$

حداقل ← مشتق = 0

$$\frac{\partial\, \text{سربار}}{\partial P} = \frac{1}{2} - \frac{Se}{P^2} = 0 \Rightarrow P^2 = 2Se \Rightarrow P = \sqrt{2Se}$$

جلسه سیزدهم

قطعه بندی (Segmentation)

- مشکلات صفحه بندی: ۱. صفحه بندی مازولاریتی ندارد (طراحی آن بیانداری نیست) یعنی فرآیند به چند
۲. یکپارچه است از دید برنامه نویسی در نتیجه برنامه نویسی ها مدیریتی روی قسمتهای مختلف برنامشان ندارند
۳. عدم امکان: داشتن اک گذاشتن کدهای مختلف فرآیند به دلخواه کاربر
۴. عدم امکان حفاظت انطباف پذیر از بخشهای مختلف فرآیند
۵. امکان همپوشانی جداول و داده هایی که رشته می کنند وقتی در صفحه بندی وجود دارد

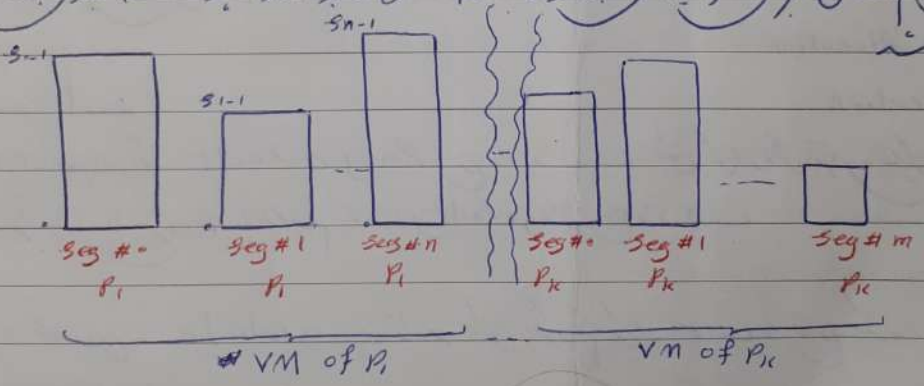
در قطعه بندی برنامه نویس اجازه دارد به تعدادی قطعه کوچک و بزرگ و پویا به دلخواه خود تعریف و ایجاد کند و هرکدام را جداگانه حفاظت نماید

صاحب قطعه	اعضاء گروه	سایرین
۲ W R	۲ W R	۲ W R

1 Read, Write, Execute با

می تواند به صورت کنترل شده، قطعات مختلف را share کند، حفاظت کند و با آنها کار کند

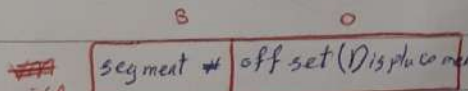
در قطعه بندی بازهم امکان اجرای فرآیندهای بزرگتر از حافظه اصلی وجود دارد چون حافظه مجازی بزرگتر از حافظه فیزیکی می تواند باشد



قطعاتی از حافظه اصلی می آیند جزء مجموعه کاری باشند

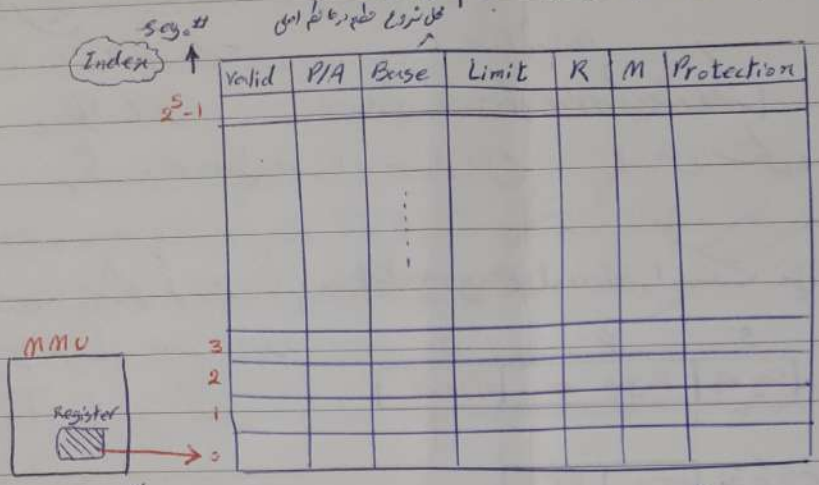
- Segment ها چگونه در حافظه قرار می گیرند؟ به صورت پراکنده

- فرمت آدرس مجازی در قطعه بندی چگونه می باشد؟



- حداکثر 2^S قطعه
- حداکثر اندازه هر قطعه 2^O ● حداکثر اندازه فرآیند (VM) = 2^{S+O}

در اینجا MMU ترجمه آدرس را به کمک جدول قطعه انجام می دهد



در اینجا معمولاً TLB هم داریم و اگر بخواهیم از جدول قطعه در حافظه استفاده کنیم سرعت کندی می شود

- برای ترجمه آدرس:

ابتدا MMU شماره قطعه را از سمت چپ حافظه می گیرد و دید به TLB می دهد.

اگر TLB ، Hit شود ، Base ، Limit قطعه را به ما می دهد

if offset ≤ Limit ⇒ آدرس فیزیکی = Base + offset

مشکلات قطعه بندی:

۱. معایب External fragmentation (تکه تکه شدن خارجی) به وجود می آید چون اندازه قطعات متفاوت است در نتیجه وقتی حفره های خارجی با اندازه های متفاوت ایجاد می شود

۲. ما بیهوده ممکن است کل یک قطعه خیلی بزرگ در حافظه بارشود درحالی که به بخش کوچکی از قطعه نیاز داریم

قطعه بندی در کل حافظه را هدر می دهد پس روش خوبی نیست

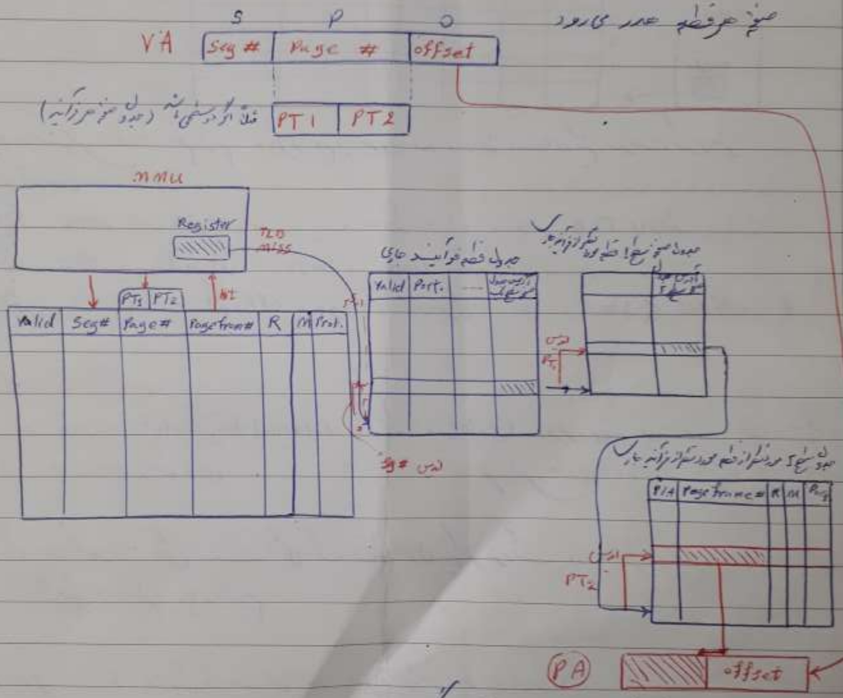
ترکیب قطعه بندی با صفحه بندی: در این روش به برنامه نویس اجازه می دهیم فرآیند خودش را قطعه قطعه کند. سیستم عامل و معماری باید هر قطعه را به طور مجزا صفحه بندی کند (از دید کاربران پنهان است) بنابراین هر قطعه باید برای خود یک جدول صفحه مجزا داشته باشد

هر چه را در مورد صفحه بندی یک فرآیند گفته شد برای صفحه بندی هر قطعه نیز در نظر می گیریم

- مزایای این ترکیب عبارتند از:

۱. ماژولاریتی دارد
۲. Protection
۳. Sharing = اشتراک گذاشتن قطعات
۴. سادگی (سادگی جداول را رشد دهد)
۵. عدم استقلال بودن کل قطعات بزرگ به حافظه و استفاده بهینه از آن
۶. حل مشکل External Fragmentation می‌باشد

- اندک مشکلی که وجود دارد internal fragmentation است یعنی به طور متوسط $\frac{p}{2}$ از آخرین صفحه هر قطعه هدر می‌رود



- در مورد این سیستم نکات زیر را در نظر بگیرید:

۱. هر فرآیند حداکثر 2^s قطعه دارد
۲. هر قطعه حداکثر 2^p صفحه دارد
۳. هر فرآیند حداکثر 2^{s+p} صفحه دارد
۴. اندازه هر صفحه 2^o است
۵. حداکثر اندازه هر قطعه 2^{p+o} است
۶. حداکثر اندازه هر فرآیند 2^{s+p+o} است
۷. هر فرآیند یک جدول قطعه دارد
۸. هر قطعه ۱ جدول صفحه سطح ۱ دارد
۹. هر قطعه حداکثر 2^{PT1} جدول صفحه سطح ۲ دارد
۱۰. هر فرآیند حداکثر 2^s جدول صفحه سطح ۱ دارد

۱۱- هر فرآیند حداکثر 2^{S+PT_1} جدول صفحه سطح ۲ دارد

- محاسبه زمان ترجمه آدرس در زمان کل دسترسی به حافظه

مثال ۱: فرض کنید یک سیستم صفحه‌بندی محض با جداول تک سطحی بدون TLB داریم اگر زمان Axces Time حافظه اصلی (T_{mem}) برابر ۱۰ns باشد زمان ترجمه آدرس و کل زمان دسترسی را محاسبه نمایید

$$T_{ترجمه} = T_{mem} = 10\,ns \qquad T_{کل} = T_{ترجمه} + T_{Mem} = 2T_{mem} = 20\,ns$$

مثال ۲: در مثال فوق فرض کنید TLB داریم و زمان دسترسی به TLB ($T_{TLB}=2ns$) باشد اگر نرخ اصابت TLB (H_{TLB}) برابر ۰/۹ باشد زمان ترجمه و زمان کل دسترسی را محاسبه نمایید

$$T_{ترجمه} = T_{TLB} + (1-H_{TLB})\,T_{mem} = 2_{ns} + (1-0/9)\,10 = 3\,ns$$

$$T_{ترجمه} = H_{TLB} \cdot T_{TLB} + (1-H_{TLB})(T_{TLB} + T_{mem})$$

$$T_{کل} = T_{ترجمه} + T_{mem} = 3 + 10 = 13\,ns$$

مثال ۳: اگر در سیستم فوق جدول صفحه دوسطحی باشد:

$$T_{ترجمه} = T_{TLB} + (1-H_{TLB}) * 2\,T_{mem} = 2_{ns} + (1-0/9)\,2*10 = 4\,ns$$

$$T_{کل} = 4 + 10 = 14\,ns$$

مثال ۴: در مثال فوق فرض کنید قطعه‌بندی با صفحه‌بندی دوسطحی داریم:

$$T_{ترجمه} = T_{TLB} + (1-H_{TLB}) * 3\,T_{mem} = 2 + (1-0/9)*3*10 = 5$$

(جدول قطعه / جدول صفحه / جدول صفحه)

$$T_{کل} = 5 + 10 = 15$$

مثال ۵: فرض کنید در سیستم فوق Cache Memory داریم Axces time cache (T_{chache}) برابر ۵ns و نرخ اصابت Cache (H_{cache}) برابر ۰/۹۵ فرض شود:

$$T^{C}_{mem} = T_{cache} + (1-H_{cache})\,T_{mem} = 5 + (1-0/95)\,10 = \underline{5.5\,ns}$$

(زمان دسترسی با حافظه دارای cache)

$$T_{ترجمه} = T_{TLB} + (1-H_{TLB}) * 3\,T^{C}_{mem} = 2 + (1-0/9)*(3*5/5) = \underline{3/65\,ns}$$

$$T_{کل} = T_{ترجمه} + T^{C}_{mem} = 3/65 + 5.5 = \underline{9/15\,ns}$$