

به نافع خدر

سرلی دانشجو

| |
|--|
| |
|--|

دانش اگر در ثریا هم باشد مردانی انر سرزمین پارس بدان دست خواهند یافت. رسول اکرم (ص)

به نام خدا

جلسه اول

سیستم عامل:

□ نرم افزاری است که کامپیوتر را کنترل می کند.

□ نرم افزاری است که سفت افزار را برای کاربر، قابل استفاده می کند.

سرویس های سیستم عامل در سه سطح قرار دارند

1- kernel Service 2- library service 3- application service

هسته سیستم عامل kernel میباشد که در هر ممتاز اجرا می شود. سیستم حداقل از دو حالت اجرا حمایت می کند

1- حالت ممتاز (مق بالا) 2- حالت عادی (مق پایین)

توضیحات اضافی: حالت غیر ممتاز (مق پایین) حالتی است که اجازه اجرای دستورالعمل های حساس سفت افزاری مثل دستورالعمل توقف و دستورالعمل های ورودی / خروجی را نمی دهند، این حالت را حالت کاربر نیز گویند چرا که برنامه های کاربری معمولاً در این حالت اجرا می شوند حالت ممتاز یک وضعیت اجرائی است که به تمام دستورالعمل های سفت افزار اجازه اجرا می دهد، که به این حالت حالت سیستم، حالت کنترل یا حالت هسته می گویند

Kernel یا هسته سیستم:

1- به وقفه واکنش نشان می دهد

2- به سفت افزار دسترسی مستقیم دارد

3- مقیم دائمی حافظه میباشد

4- ایبار پرداز، قتم پرداز، زمانبندی و ... در قسمت Kernel انجام میشود.

اهداف سیستم عامل:

1- مدیریت منابع مانند پردازنده، حافظه، گذرگاهها، تایمرها و ...

2- ماشین توسعه یافته (extend machine)

3- ایبار لایه تجریر: مجموعه برنامه هایی است که جزئیات سفت افزاری را از دید کاربر مخفی نگه می دارد

4- ایبار واسط کاربری قوی و فوش تعریف

تاریفیه سیستم عامل: بهوت بررسی تاریفیه سیستم عامل می بایست تاریفیه معماری ماشین هائی را در نظر گرفت که سیستم عامل ها بر روی آنها اجرا

1- نسل اول 55-1945: در ساخت کامپیوترها از لامپ فلا استفاده می شد- ماشین های آن موقع قادر بودند در هر لحظه یک بیت را اجرا کنند- زبان های برنامه نویسی حتی اسمبلی ناشناخته بودند- پس چیزی به نام سیستم عامل وجود نداشت.

2- نسل دوم 65-1955: کامپیوترها از ترانزیستور ساخته شدند- شرکت جنرال سیستم اولین سیستم عامل را برای کامپیوتر IBM 701 نوشت- سیستم عامل ها به صورت batch کار می کردند.

3- نسل سوم 80-1965: پیدایش مدارات مجتمع IC- پیدایش multiprograming- پیدایش vlsi-lsi؛ این دوره معادل میشود با عرضه سیستم عامل برای کامپیوترهای شخصی، همچنین سیستم عامل ها روی شبکه کار می کردند، معمولا اولین سیستم عامل هایی که در این دوره به برتری دست یافتند می توان به Ms Dos برای خانواده کامپیوتری Intel و Unix برای کامپیوترهای motorola، اشاره کرد.

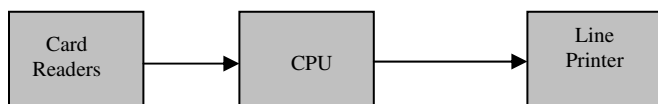
سیستم ها از جهت ارتباط با کاربر به دو دسته تقسیم می شوند

1- سیستم های مبادره ای (Interactive): سیستم هایی هستند که در آنها کاربر به طور مستقیم و روی خط (on-line) با کامپیوتر در ارتباط است. کاربر دستوراتی را وارد می کند و منتظر پاسخ می ماند، پس از دریافت پاسخ مجددا دستوراتی را وارد می کند.

2- سیستم های دسته ای (batch): سیستم هایی هستند که در آنها دریافت دستورات (برنامه های کاربر) و سپس اجرای آنها در دو مرحله انجام می گیرد. ابتدا برنامه هایی که عموما دارای نیازهای مشابه نظیر کامپایلر یکسان هستند در یک گروه به سیستم وارد شده و پس از بار شدن کامپایلر مورد نیازشان اجرای آنها به طور متوالی انجام میشود.

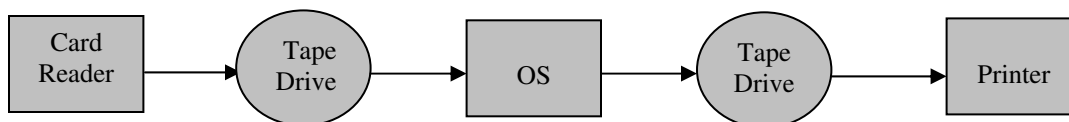
تقسیم بندی سیستم ها از جهت ارتباط با وسایل جانبی

1- سیستم های On-Line با ارتباط مستقیم: در این سیستمها پردازنده مستقیما با دستگاههای ورودی خروجی در ارتباط است. و به دلیل کند بودن این دستگاهها، کارائی پردازنده به مقدار قابل توجهی کاهش می یابد



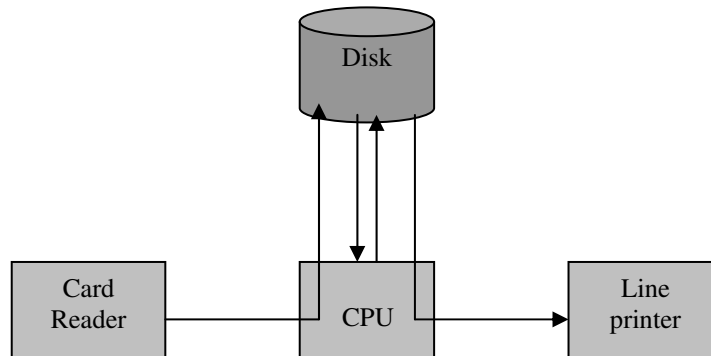
2- سیستم های Off-Line با ارتباط غیر مستقیم

در این سیستم ها پردازنده به طور مستقیم با دستگاههای ورودی و خروجی کند در ارتباط نیست. ابتدا عمل خواندن ورودی توسط کارت خوانهای مستقل از سیستم اصلی انجام میگردد و ورودی به حافظه ی جانبی (نوار مغناطیسی) که سریعتر هستند منتقل شده، سپس این نوارها در سیستم اصلی استفاده میشود، ارسال خروجی نیز بر روی نوار مغناطیسی انجام میگردد و نهایتا در محل دیگری بر روی چاپگرها ارسال می شود، در چنین حالتی امکان همپوشانی عمل I/O و کار پردازنده وجود دارد



با فرینگ: نامیه ای از حافظه است که جهت ایجاد هماهنگی بین وسایل I/O کند و پردازنده استفاده می شود. با فرینگ اجازه همپوشانی I/O یک کارو پردازش همان کار را فراهم می سازد.

اسپولینگ: اسپولینگ یک رسانه ی ذخیره سازی سریع مانند دیسک مغناطیسی جهت انجام اعمال I/O استفاده میکند طوری که اطلاعات از وسایل ورودی بر روی دیسک ذخیره شده و CPU با دیسک در تماس است، همین طور اطلاعاتی که به وسایل خروجی مانند چاپگر می بایست ارسال شود، بر روی دیسک ذخیره شده و بعد به چاپگر ارسال می شود. اسپولینگ مانند نفی است که به دور قرقره تابیده میشود به این امید که بعدا استفاده شود.



فرق بافرینگ و اسپولینگ چیست؟ بافرینگ امکان همزمانی پردازش و I/O یک کار را به کمک حافظه ی اصلی فراهم میکند، در حالی که Spooling امکان هم زمانی پردازش و I/O چند کار را به کمک حافظه ی جانبی انجام می دهد.

چند برنامه گری (multi programming): در چند برنامه گری اجرای یک کار تا زمان نیاز به ورودی یا خروجی ادامه می یابد، سپس ورودی یا خروجی آن شروع شده و پردازنده اجرای کار دیگری را شروع کرده یا ادامه می دهد.

اشتراک زمانی (Time sharing): اشتراک زمانی شکل خاصی از چند برنامه گری است که در آن تعویض یک کار بر اساس یک معیار زمانی و نه بر اساس زمان نیاز آن کار به ورودی یا خروجی صورت می گیرد. اجرای یک کار تا پایان بازه ی زمانی ادامه یافته سپس پردازنده اجرای برنامه ی دیگری را شروع میکند و عمل Switch بین پردازنده ها چنان سریع است که هر کاربر فکر میکند سیستم به تنهایی در اختیار اوست

برنامه: یک موجود غیر فعال است که شامل یک یا چند فایل می باشد که بر روی دیسک ذخیره شده اند.

پردازنده (فرا روند): یک موجود فعال (Active) است که دارای ساختار خاصی به نام PCB می باشد.

مراحل ایجاد پردازنده:

1-ایجاد پردازنده: انتخاب یکی از کارها (برنامه های) موجود بر روی دیسک جهت تبدیل شدن به پردازنده که به این عمل زمان بندی بلند مدت میکوبند که توسط زمان بند کار انجام میشود.

2-گذراندن پرفه حالت: ایجاد و تفصیص ساختارهای لازم برای پردازنده (بلاک کنترلی پردازنده PCB: Process Control Block)

PCB شامل اطلاعات زیر است

1- حالت پردازنده

2- شمارنده پردازنده

3- ثبات های پردازنده

4- اطلاعات زمانبندی پردازنده

5- اطلاعات مدیریت حافظه

6- اطلاعات مسابرسی پردازنده (میزان استفاده از منابع سیستم)

7- اطلاعات وضعیت پردازنده در رابطه با دستگاههای ورودی و خروجی و فایل ها

3- **فازمه پردازنده:** قرار گرفتن پردازنده ایجا شده در صف پردازنده های آماده اجرا

یک پردازنده می تواند در چند حالت اصلی قرار گیرد.

1- **حالت آماده (Ready state):** حالتی است که پردازنده همه منابع مورد نیازش را در اختیار دارد و فقط منتظر CPU میباشد.

2- **حالت منتظر یا مسرود (Waiting-Blocked):** حالتی است که در آن پردازنده منتظر به دست آوردن یک منبع از سیستم می باشد در این حالت اگر پردازنده، پردازنده را هم بدست آورد، قابل اجرا نیست پردازنده از حالت اجرا با نیاز به یک منبع در این حالت قرار میگیرد.

3- **حالت اجرا (Running Excuting):** در این حالت پردازنده همه منابع و پردازنده را در اختیار دارد و در حال اجراست؛ پردازنده با بدست آوردن پردازنده از حالت آماده در این حالت قرار می گیرد.

4- **حالت معلق:** حالتی است که اگر پردازنده به مدت زیادی در حالت آماده، منتظر CPU باشد و CPU به آن توجهی نکند که در آن صورت پردازنده موقتاً بر روی دیسک انتقال می یابد. با توجه مجدد CPU به این پردازنده، پردازنده از حالت معلق به حالت آماده میرو.

5- **حالت کامل:** هنگامی که پردازنده به اندازه کافی وقت پردازنده را به دست آورده و کارش فازمه یافته است (ولی هنوز از لیست

پردازنده های سیستم خارج نشده است)

نکته: در مورد پردازنده های فرزند و پدر، فازمه یک پردازنده پدر می تواند منوط به فازمه کلیه پردازنده های فرزند آن گردد. (ابتدا پردازنده های

فرزند ختم میشوند سپس پردازنده پدر به پایا ن می رسد) که به آن ختم پی در پی (Cascaded Termination) گفته می شود

...فازمه فرزند فرزند فرزند → فازمه فرزند → فازمه پدر پردازنده

فازمه پردازنده پدر → فازمه فرزند → فازمه فرزند فرزند...

صف های سیستم:

صف کار (Job Queue): در این صف برنامه هایی که منتظر تبدیل شدن به پردازنده هستند قرار میگیرند مدیریت این صف به عهده زمانبند کار و یا زمانبند بلند مدت است.

صف آماده (Ready Queue): پردازنده هایی که منتظر CPU هستند قرار می گیرند و توسط زمانبند کوتاه مدت و یا زمانبند CPU مدیریت میشود.

صف انتظار (waiting Queue): پردازنده هایی که منتظر وسایل جانبی (I/O) هستند قرار می گیرند.

انواع زمانبندی:

1- **زمانبندی کوتاه مدت (Short Term scheduler):** در مرحله ی اجرای پردازنده ها، به ممض نیاز به ورودی یا خروجی و در سیستم های اشتراک زمانی، به ممض پایان برش زمانی، و در سیستم هایی که امکان ایجا پردازنده فرزند وجود دارد، به ممض یک درخواست برای

ایجاد فرزند (Fork) (در حالتی که پردازنده پدر می بایست متوقف شود) ، پردازنده در حال اجرا باید از پردازنده و از صف پردازنده هائی آماده اجرا خارج شود و بر اساس الگوریتم زمانبندی یک پردازنده دیگر انتخاب شده و به پردازنده بار (Load) شود. این مرحله تعویض پردازنده، عمل تعویض متن (context switching) نام دارد و کل اعمال فوق توسط مدیر زمانبند کوتاه مدت انجام میشود این زمانبند با توجه به کوتاه بودن برش زمانی، در فواصل کم و با فرکانس بالا انجام میشود. تعویض متن توسط بخشی از سیستم عامل به نام Dispatcher انجام میشود.

2- زمانبندی میان مدت (Medium Term scheduler): در بعضی شرایط به دلیل زیاد شدن تعداد پردازنده های موجود در چرخه حالت پردازنده ها (چرخه آماده-اجرا-منتظر) و در نتیجه کم شدن حافظه ی آزاد سیستم و کاهش کارائی، مناسب است تعدادی از پردازنده ها از حالت فعال در پردازنده خارج شده (به طور موقت) و پس از اتمام انجام اجرای تعدادی از پردازنده ها، اجرای آنها ادامه یابد. عمل انتقال پردازنده های موجود در حافظه ی اصلی و در حالت آماده به حافظه ی جانبی به منظور کاهش بار سیستم Swap out و انتقال دوباره پردازنده های آماده از حافظه ی جانبی به حافظه ی اصلی و فعال شدن دوباره آنها، Swap in نام دارد که هر دو توسط بخشی از سیستم عامل به نام مدیریت زمانبند میان مدت انجام میشود.

3- زمانبند بلند مدت: انتخاب یکی از کارهای موجود در سیستم جهت تبدیل شدن به پردازنده، چون این عمل در فواصل طولانی انجام می شود به زمانبندی بلند مدت مشهور است.

پایان جلسه ی اول

پلسه ی دو۴:

عموما کارها در دو دسته قرار می گیرند

- 1- کارهای I/O bound (I/O Limited): کارهایی که بخش زیادی از اجرای آنها در ارتباط با دستگاههای ورودی/خروجی بوده و مناسبات زیادی ندارند. مثل برنامه ای که می بایست کارنامه ی دانشجویان را چاپ کند
 - 2- کارهای CPU bound (CPU Limited): کارهایی که مهم زیادی مناسبات داشته و بخش عمده نیاز آنها برای اجرا وقت پردازنده است. مثل برنامه ای که می بایست یک دستگاه معادله ی صد مجهولی را حل کند
- نکته:

در مورد کارهای CPU bound و I/O bound استفاده از بافر تاثیر زیادی در بهبود اجرا ندارد.

زمانبند بلند مدت می بایست تلفیق مناسبی از کارهای CPU bound و I/O bound را انتخاب کند.

حالت پایدار: زمانی که نرخ ایجاد پردازها برابر با نرخ خروج یا خاتمه پردازها باشد. بنا بر این زمانبند بلند مدت سعی می کند که سیستم به حالت پایدار برسد.

الگوریتم های زمانبندی کوتاه مدت (پردازنده):

عمل زمانبندی کوتاه مدت بر اساس الگوریتم های زمانبندی انجام می شود. این الگوریتم ها سعی در برآورده کردن معیارهای زیر را دارند.

1- عدالت: یعنی سهم پردازها از CPU به طور منصفانه باشد

2- افزایش کارایی: هدف از کارایی مشغول بودن CPU به طور ایده آل است.

3- افزایش گذردهی: منظور تعداد پردازهای انجام شده در واحد معینی از زمان می باشد.

4- کاهش زمان پاسخ: زمان ما بین مطرح شدن یک پردازها تا اولین اجرای آن پردازها می باشد.

این اهداف کاملا باهم در تناقض اند بنا براین می بایست سعی شود مصالحه ای بین این اهداف صورت گیرد.

□ زمان انتظار (waiting Time): مدت زمانی را که پردازها در حالت آماده است و منتظر CPU می باشد را زمان انتظار گویند.

□ زمان اجرا (Runing Time): مدت زمانی را که پردازها بر روی CPU در حال اجراست را زمان اجرا گویند

□ زمان I/O: مدت زمانی است که پردازها نیاز به عمل I/O دارد

□ زمان گردش کار (Turnatound Time): مدت زمان ما بین ورود یک پردازها به سیستم تا اجرای کامل یک پردازها را زمان گردش کار گویند.

□ زمان تعویض متن + زمان I/O + زمان اجرا + زمان انتظار = زمان گردش کار

□ زمان تعویض متن: زمانی است که برای تعویض پردازها بر روی CPU صرف میشود. که این عمل شامل ذخیره حالت پردازها در

حال اجرا و Update ثبات های سیستم به مقادیر پردازها جدید می باشد.

الگوریتم های زمانبندی به دو دسته تقسیم می شوند.

- 1- انحصاری یا انقطاع نا پذیر (Preemptive): در این الگوریتم ها تا زمان خاتمه پردازها و یا تا زمان نیاز به عمل I/O نمی توان CPU را از پردازها در حال اجرا گرفت و به پردازها دیگری انتساب داد.

2- غیر انحصاری یا انقطاع پذیر (Non Preemptive): در این الگوریتم ها در پایان برش زمانی (Time Slice) و یا تغییر شرایط سیستم، مدیر زمانبندی می تواند کنترل پردازنده را از یک پردازنده در حال اجرا گرفته و به پردازنده دیگری بدهد.

الگوریتم الویت با اولین ورودی - FCFS (Frist Come Frist Service یا Frist In Frist Out)

این الگوریتم به سادگی هر پردازنده ورودی را در صف FIFO قرار داده و از سر صف اجرایش را شروع می کند و اجرای پردازنده ها را به طور انحصاری انجام میدهد.

مزایا:

سادگی اجرا - عملی بودن

معایب:

1- زیاد بودن میانگین زمان انتظار (زمان گردش کار)

2- انقطاع نا پذیری و غیر قابل استفاده بودن در سیستم های اشتراک زمانی است

الگوریتم الویت با کوتاهترین کار - SJF (Shortest Job Frist):

این الگوریتم از بین پردازنده های موجود در صف آماده، پردازنده ای را جهت اجرا انتخاب می کند که زمان اجرای کمتری نیاز داشته باشد و پردازنده در حال اجرا اجرایش را به صورت انقطاع ناپذیر ادامه می دهد.

معایب:

1- نیاز به داشتن اطلاعات مقاطع زمانی مورد نیاز قبل از شروع اجرا

2- احتمال به تعویق افتادن کارهای طولانی

3- انقطاع نا پذیری

مزایا: دارا بودن کمترین زمان برگشت یا زمان انتظار ما بین تمام الگوریتم ها

الگوریتم الویت با کمترین زمان باقیمانده - SRT (Shortest Remaining Time)

در این الگوریتم انتخاب بر اساس کمترین زمان مورد نیاز برای کامل شدن صورت میگیرد. این الگوریتم غیر انحصاری بوده و با ورود هر پردازنده به صف آماده، بر رسی زمان باقیمانده پردازنده ها انجام می گیرد، اگر پردازنده تازه وارد شده، زمان کمتری برای کامل شدن لازم دارد، پردازنده در اختیار آن قرار می گیرد.

معایب: احتمال تعویق کارهای طولانی - نیاز به دانستن زمان مورد نیاز پردازنده ها

مزایا: انقطاع پذیری - زمان پاسخ نسبتاً مناسب

مثال: در سیستمی سه پردازنده P_0 و P_1 و P_2 مطابق جدول زیر با زمان های ورود و اجرای داده شده قرار دارند مطلوب است مناسبه میانگین زمان انتظار و میانگین زمان گردش کار در حالت های زیر

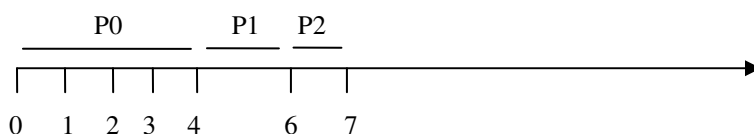
| زمان اجرا | زمان رسیدن | پردازنده |
|-----------|------------|----------|
| 4 | t | P_0 |
| 2 | t+1 | P_1 |
| 1 | t+2 | P_2 |

الف: برای زمانبندی پردازنده ها از الگوریتم FCFS استفاده شود.

ب: برای زمانبندی پردازنده ها از الگوریتم SJF استفاده شود.

ج: برای زمانبندی پردازنده ها از الگوریتم SRT استفاده شود.

الف:



$$\text{میانگین زمان انتظار} = \frac{\text{مجموع زمان انتظار تمام پردازش ها}}{\text{تعداد پردازش ها}} \quad \text{میانگین زمان گردش کار} = \frac{\text{زمان اجرای پردازش + زمان انتظار}}{\text{تعداد پردازش ها}}$$

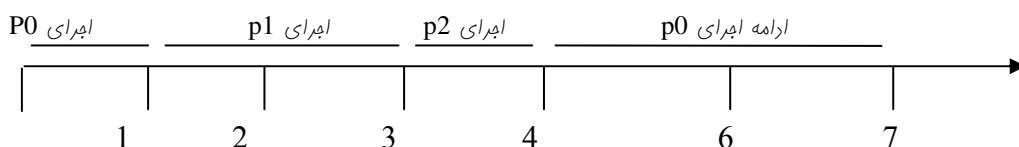
$$\text{میانگین زمان انتظار} = \frac{0+3+4}{3} = \frac{7}{3} \quad \text{میانگین زمان گردش کار} = \frac{(0+4) + (3+2) + (4+1)}{3} = \frac{14}{3}$$

ب) در این شرایط اگر پردازش ای شروع به اجرا شود تا انتها ادامه می یابد در مرحله ی بعد پردازش ای برای اجرا انتخاب می شود که کمترین زمان اجرا را دارد بنا براین اول P0 (در این لحظه فقط همین را داریم) تا لحظه ی 4 اجرا میشود در این لحظه دو تا پردازش آماده داریم که P2 زمان کمتری را برای اجرا می خواهد پس تا لحظه ی 5 اجرا می شود از 5 تا 7 زمان اجرای P1 می باشد.

$$\text{زمان انتظار: } P_0=0s, P_1=4s, P_2=2s$$

$$\text{میانگین زمان انتظار} = \frac{0+4+2}{3} = 2 \quad \text{میانگین زمان گردش کار} = \frac{(0+4) + (4+2) + (2+1)}{3} = \frac{13}{3}$$

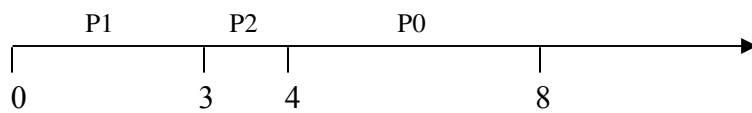
ج) در این شرایط در تمامی لحظه های اجرا اگر پردازش ای از راه برسد که زمان اجرای کمتر از زمان باقیمانده پردازش در حال اجرا باشد پردازش را در اختیار گرفته و شروع به اجرا می کند P0=3s, p1=0s, p2=1s : زمان انتظار



$$\text{میانگین زمان انتظار} = \frac{3+0+1}{3} = \frac{4}{3} \quad \text{میانگین زمان گردش کار} = \frac{(3+4) + (0+2) + (1+1)}{3} = \frac{11}{3}$$

مثال: مثال قبل را با جدول زیر حل کنید (برای قسمت ب و ج)

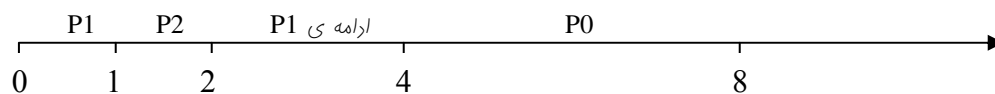
| پردازش | زمان رسیدن | زمان اجرا |
|--------|------------|-----------|
| P_0 | 0 | 4 |
| P_1 | 0 | 3 |
| P_2 | 1 | 1 |



ب) SJF

میانگین زمان گردش $ATT = \frac{(4+4) + (0+3) + (2+1)}{3} = \frac{14}{3}$

میانگین زمان انتظار $AWT = \frac{4+0+2}{3} = 2$



ج) SRT

$ATT = \frac{(4+4) + (1+3) + (0+1)}{3} = \frac{13}{3}$

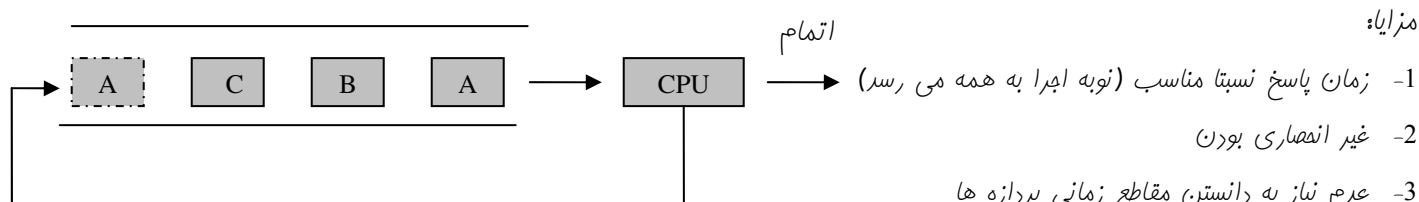
$AWT = \frac{4+1+0}{3} = \frac{5}{3}$

پایان جلسه دوم

پایه ی سوم:

الگوریتم نوبتی (Round Robin: RR): در این الگوریتم با در نظر گرفتن یک برهه زمانی (time slice) یا کوانتوم زمانی پردازش های موجود در صف آماده هر کدام به اندازه این برهه زمانی CPU را بدست آورده به طوری که اگر در برهه زمانی شان اجرای خود را به اتمام نرسانند مجدداً به انتهای صف منتقل می شوند.

- این الگوریتم بیشتر در روش های اشتراک زمانی (Time sharing) استفاده می شود.



معایب:

- نیاز به دقت در تنظیم برش زمانی
 - کاهش کارایی پردازنده به خاطر زمان های تعویض متن در برهه های زمانی خیلی کوتاه
 - شباهت به الگوریتم FCFS برای برش های زمانی طولانی
- کوانتوم زمانی بین 10 تا 100 میلی ثانیه است.

مثال: سه پردازش زیر را در نظر بگیرید، اگر از کوانتوم زمانی 4 میلی ثانیه استفاده شود، میانگین زمان انتظار و میانگین زمان گردش کار را بیابید. (هر سه در لحظه صفر وارد شده اند ولی الویت به ترتیب با شماره است)

نمودار گانت پردازش ها به صورت زیر خواهد بود.

| پردازش | زمان اجرا |
|--------|-----------|
| p_1 | 24 |
| p_2 | 3 |
| p_3 | 3 |

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| P1 | P2 | P3 | P1 | P1 | P1 | P1 | P1 | |
| 0 | 4 | 7 | 10 | 14 | 18 | 22 | 26 | 30 |

$$\text{میانگین زمان انتظار} = \frac{6+4+7}{3} = \frac{17}{3} \text{ ms}$$

$$\text{میانگین زمان گردش کار} = \frac{(6+24) + (4+3) + (7+3)}{3} = \frac{47}{3} \text{ ms}$$

مثال:

چهار پردازنده مطابق جدول زیر در سیستم وجود دارند. اگر از روش RR با برش زمانی 1ms استفاده شود و از سربار ناشی از تعویض متن فرایندها صرف نظر شود میانگین زمان انتظار پردازش ها چقدر است (فرض شود پردازش ای که از راه می رسد به ابتدای صف منتقل می شود)

| پردازش ها | زمان ورود | زمان پردازش |
|-----------|-----------|-------------|
| p_1 | 0 | 3 |
| p_2 | 1 | 5 |
| p_3 | 3 | 2 |
| p_4 | 9 | 2 |

| | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|----|
| P1 | P2 | P1 | P3 | P2 | P1 | P3 | P2 | P2 | P4 | P2 | P4 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$$\text{میانگین زمان انتظار} = \frac{3+5+2+1}{4} = \frac{11}{4} \text{ ms}$$

الگوریتم زمانبندی الویت دار

$$\text{الویت} = \frac{\text{زمان سرویس} + \text{زمان انتظار}}{\text{زمان سرویس}}$$

الویت با بالاترین نسبت پاسخ (Highest Responses Ratio Next -HRN)

الگوریتم های الویت دار با در نظر گرفتن یک الویت برای هر کدام از پردازش های موجود در سیستم ، پردازش ای که بالاترین الویت را داشته باشد اجرایش را به صورت انحصاری انجام می دهد ، الویت هائی که می توان در این الگوریتم در نظر گرفت عبارتند از

1- نسبت مقاطع زمانی ورودی یا خروجی به مقاطع زمانی پردازنده (CPU Burst به I/O Burst)

2- محدودیت زمانی ، نیاز های حافظه

3- تعداد فایل های باز شده

4- اهمیت پردازش و بخش ارائه دهنده کار

5- الویت نسبت پاسخ (فرمول بالائی)

مزایا:

1- طبق این الگوریتم پردازش های کوتاه در ابتدا الویت بالا پیدا می کنند ، چون زمان سرویس (زمان مورد نیاز جهت اجرا) کمتری نیاز دارند

2- پردازش هائی که مدت زیادی منتظر مانده اند نیز الویت بالا پیدا می کنند ، پس این شانس را پیدا می کنند که اجرا شوند.

معایب:

1- می بایست زمان سرویس را از قبل معلوم کرد

2- انحصاری است

مثال:

پردازش های زیر را در نظر بگیرید که همگی در لحظه ی صفر به ترتیب داده شده رسیده اند میانگین زمان انتظار را بیابید و جهت زمانبندی پردازش ها از الگوریتم الویت دار استفاده کنید و پردازش ای که عدد الویت آن کمتر است ، الویت بالاتری دارد.

حل: نمودار گانت به صورت زیر خواهد بود

| الویت | زمان اجرا | پردازش ها |
|-------|-----------|-----------|
| 3 | 10 | p_1 |
| 1 | 1 | p_2 |
| 3 | 2 | p_3 |
| 4 | 1 | p_4 |
| 2 | 5 | p_5 |

| | | | | |
|----|----|----|----|----|
| P2 | P5 | P1 | P3 | P4 |
| 0 | 1 | 6 | 16 | 18 |
| | | | | 19 |

$$\text{میانگین زمان انتظار} = \frac{6+0+16+18+1}{5} = \frac{41}{5} \text{ ms}$$

الگوریتم صف چند سطحی (Multi Level Queue: MLQ):

در الگوریتم صف چند سطحی پردازنده ها در صف های مختلف ، که هر صف الویت خاصی دارد قرار می گیرند و در صف های مختلف از الگوریتم های زمانبندی مختلفی استفاده می شود ، پارامتر هائی که در این الگوریتم می بایست مشخص نمود عبارتند از :

1- تعداد صف ها

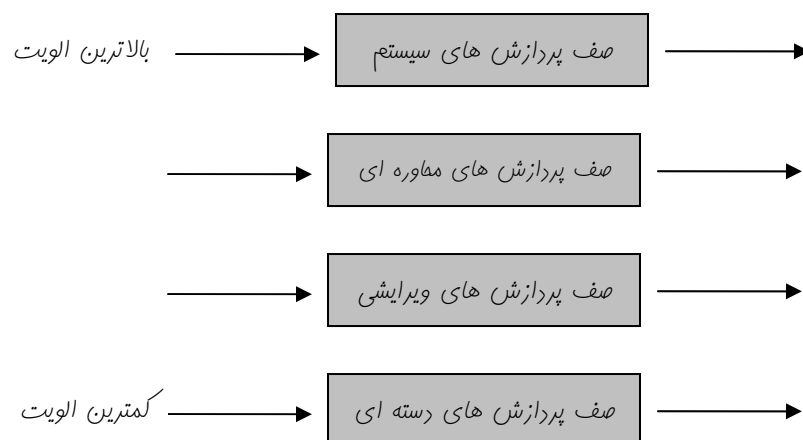
2- الگوریتم زمانبندی استفاده شده در هر صف

3- الویت صف ها نسبت به هم

• اگر در صفی با الویت بالا پردازنده ای وجود داشته باشد CPU در ابتدا پردازنده های آن صف را سرویس داده و در صورتی که صف های الویت بالاتر خالی شود به سراغ صف الویت پایین می رود.

• به عنوان مثال سیستمی می تواند 4

صف آماده با الویت های زیر داشته باشد



الگوریتم صف باز خورد چند سطحی (Multi Level Feedback Queue : MFQ)

این الگوریتم مانند الگوریتم MLQ می باشد با این تفاوت که در این الگوریتم امکان حرکت پردازنده ها بین صف های مختلف نیز وجود دارد ، در این الگوریتم علاوه بر مشخص نمودن پارامتر های MLQ موارد زیر نیز می بایست مشخص شود .

1- چه موقع یک پردازنده از صف بالا به صف پایین مهاجرت می کند

2- چه موقع یک پردازنده از صف پایین به صف بالا مهاجرت می کند

مثال:

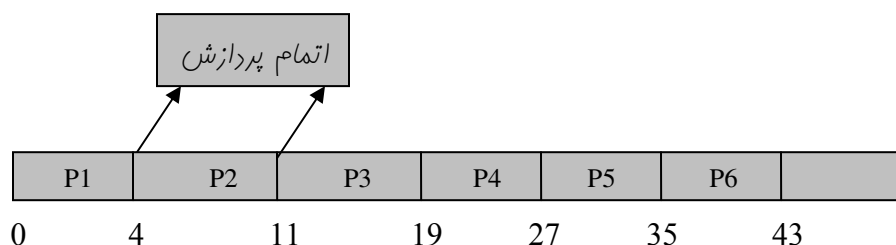
یک سیستم با الگوریتم MFQ یا MLF سه سطحی را در نظر بگیرید . اگر سطح اول از الگوریتم RR با کوانتوم زمانی 8 میکرو ثانیه و صف دوم از الگوریتم RR با کوانتوم زمانی 16 میکرو ثانیه و صف سوم با الگوریتم FCFS زمانبندی شوند . اگر 6 پردازنده همگی در زمان صفر با زمانهای اجرای 4 ، 7 ، 12 ، 20 ، 25 و 30 میکرو ثانیه وارد سیستم می شوند میانگین زمان برگشت و میانگین زمان انتظار

را در این سیستم بیابید (هر پردازش به معض پایان کوانتوم زمانی از صف بالاتر به صف پایین تر مهاجرت می کند) (کارشناسی ارشد - دولتی 80)

حل:

ابتدا برنامه ها در صف اول با $TS=8 \mu S$ قرار می گیرند و اگر در مدت 8 میکرو ثانیه تمام نشوند به صف دوم با $TS=16 \mu S$ منتقل می گردند باز هم اگر در این مدت 16 میکرو ثانیه تمام نشوند وارد صف سوم با الگوریتم FCFS می گردند. بنابراین نمودار زمانی این پردازش ها به صورت شکل زیر خواهد بود

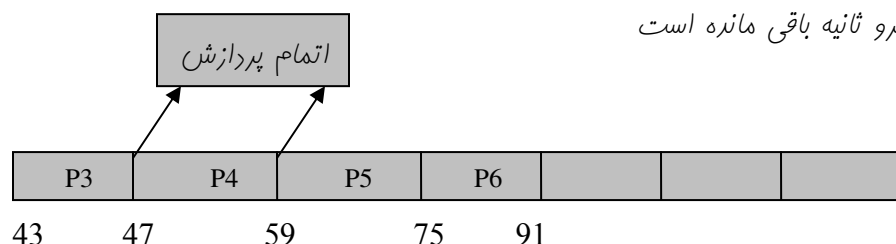
صف اول با $TS=8 \mu S$



پس از یکبار اجرای پرفشی P1 و P2 تمام می شوند در حالی که از برنامه های P3 ، P4 ، P5 و P6

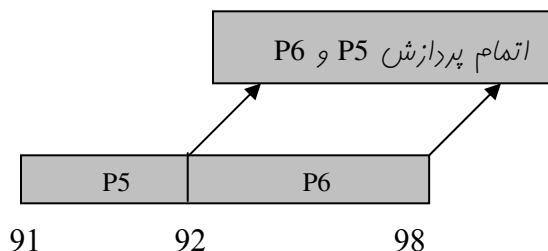
به ترتیب زمان های 4 ، 12 ، 17 و 22 میکرو ثانیه باقی مانده است

صف دوم با $TS=16 \mu s$



در انتهای مرحله ی دوم P3 و P4 تمام شده و از برنامه P5 به اندازه 1 و از برنامه P6 به اندازه 6 میکرو ثانیه باقی مانده است ، این دو پردازش در مرحله ی آخر به صورت FCFS زمانبندی می شوند.

صف سوم FCFS.



$$\text{میانگین زمان انتظار} = \frac{0 + 4 + 35 + 39 + 67 + 68}{6} = 35/5$$

$$\text{میانگین زمان برگشت} = \frac{(0 + 4) + (4 + 7) + (35 + 12) + (39 + 20) + (67 + 25) + (68 + 30)}{6} = 51/83$$

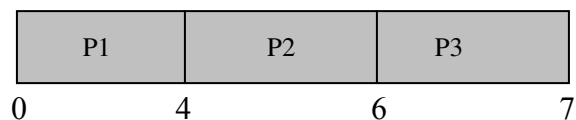
| | | | |
|-------|---|-------|---|
| p_3 | 1 | $t+3$ | 1 |
|-------|---|-------|---|

مثال: سه پردازش دسته ای (P_1 ، P_2 ، P_3 با زمان اجرا و زمان وارد شدن زیر را در نظر بگیرید متوسط زمان پاسکوئی را با تمامی الگوریتم ها به جز الگوریتم های MFQ و MLQ بیابید. (ارشد-83)

| | الویت | زمان وارد شدن | زمان اجرا |
|-------|-------|---------------|-----------|
| p_1 | 2 | t | 4 |
| p_2 | 0 | t | 2 |

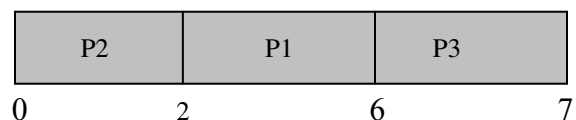
حل: t, را صفر در نظر می گیریم

الف) FCFS



$$AWT = \frac{(0+4) + (4+2) + (3+1)}{3} = \frac{14}{3}$$

ب) SJF



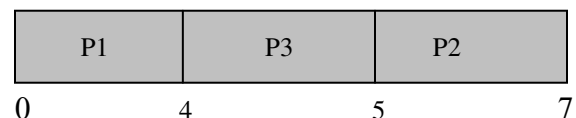
$$AWT = \frac{(2+4) + (0+2) + (3+1)}{3} = \frac{12}{3}$$

ج) SRT



$$AWT = \frac{(3+4) + (0+2) + (0+1)}{3} = \frac{10}{3}$$

د) الگوریتم نوبتی (Priority):



$$AWT = \frac{(0+4) + (5+2) + (1+1)}{3} = \frac{13}{3}$$

مثال:

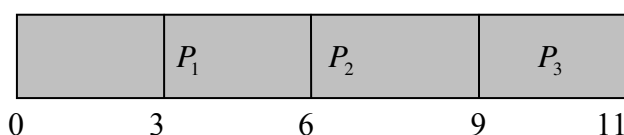
چهار پردازنده P_0 ، P_1 ، P_2 ، P_3 با مشخصات زیر در نظر بگیرید. میانگین زمان پاسخ دهی آنها را در الگوریتم های زمانبندی زیر بیابید. (فرض کنید تکه زمانی معادل یک واحد زمانی است. در مورد سیاست باگرددش نوبت پردازنده ای که وارد سیستم می شود در همان ابتدای ورودش اجرای آن آغاز می شود)

| نام پردازنده | زمان ورود به سیستم | زمان پردازش |
|--------------|--------------------|-------------|
| P_0 | 0 | 3 |
| P_1 | 1 | 3 |
| P_2 | 4 | 3 |
| P_3 | 6 | 2 |

الف) FCFS

ب) RR

حل: الف) FCFS



$$AWT = \frac{(0+3) + (2+3) + (2+3) + (3+2)}{4} = \frac{18}{4}$$

| | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| P_0 | P_1 | P_0 | P_1 | P_2 | P_0 | P_3 | P_1 | P_2 | P_3 | P_2 |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|

RR (ب)

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9$$

$$AWT = \frac{(3+10) + (4+3) + (4+3) + (2+2)}{4} = \frac{24}{4}$$

جلسه چهارم

رابطه بین پردازش‌ها (Interprocess Communication)

پردازش‌ها از جهت وابستگی به یکدیگر در دو دسته قرار می‌گیرند و توجه به این مسئله در نحوه فاصله آنها حائز اهمیت است

الف- پردازش‌های مستقل: هر پردازش‌ای که داده‌ای را (چه موقت و چه دائمی) با دیگر پردازش‌ها به اشتراک نگذارد مستقل است و یا به عبارت دیگر یک پردازش در صورتی مستقل از دیگر پردازش‌ها است که شرایط زیر را برآورده کند.

1- حالت آن به اشتراک‌گذاری نشده باشد.

2- نتیجه اجرای آن کاملاً مشخص باشد.

3- نتیجه اجرا در دفعات مختلف به ازای ورودی‌های یکسان، مساوی باشد.

4- توقف و شروع آن بدون تاثیر بر سایر پردازش‌ها امکان‌پذیر باشد.

ب- پردازش‌های همکاری‌کننده: اگر دو پردازش نسبت به هم یکی از شرایط فوق را نداشته باشد کوئیم پردازش‌ها نسبت به هم همکاری‌کننده به عبارت دیگر پردازش‌هایی هستند که شرایط زیر در مورد آنها صادق باشد.

1- حالت آن به اشتراک‌گذاری شود

2- نتیجه اجرا به دلیل وابستگی به سایر پردازش‌ها کاملاً مشخص نباشد

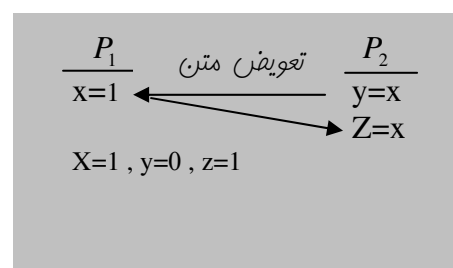
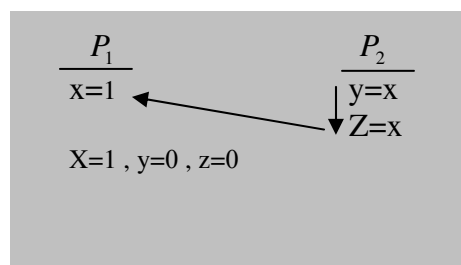
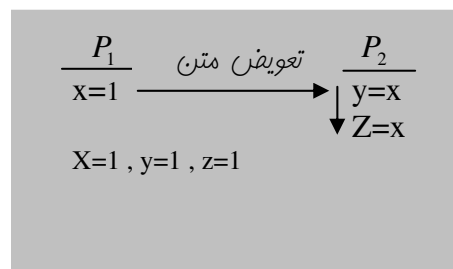
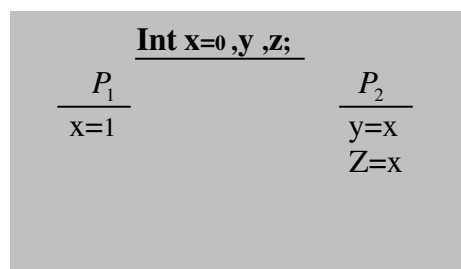
3- نتیجه اجرا برای ورودی‌های یکسان در دفعات مختلف، یکسان نباشد

منظور از اجرای Interleaved چیست؟

به این معناست که در بین اجرای یک پردازش در هر زمان امکان وقفه یا تعویض متن (Context Switching) به پردازش دیگر وجود دارد

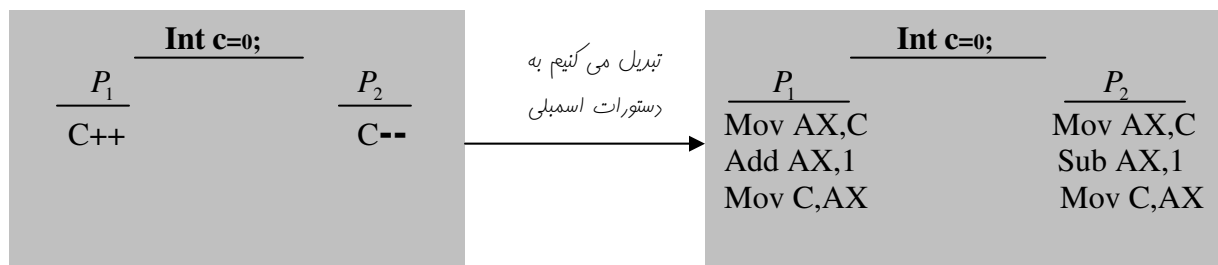
مثال: اگر دو پردازش P_1 و P_2 به صورت هم‌روند اجرا شوند مقادیر نهایی x ، y و z را بیابید

حل: در اجرای هم‌روند این دو پردازش سه ترتیب زیر امکان‌پذیر است

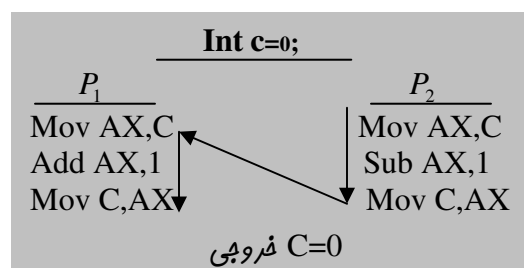


منظور از همگام سازی پردازش ها (Synchronization) چیست؟

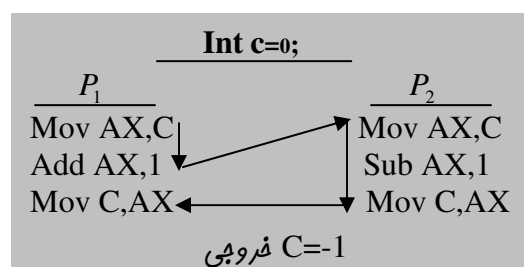
زمانی که پردازش ها به هم وابسته اند نتیجه اجرا به ترتیب اجرای دستورات برنامه بستگی دارد، بنا براین می بایست با توجه به نتایج مورد انتظار ترتیب اجرای دستورات پردازش ها را مشخص کرد که به این مسئله همگام سازی پردازش ها گویند
مثال: اگر دو پردازش P_1 و P_2 به صورت همروند اجرا شوند مقدار نهائی C بیاید.



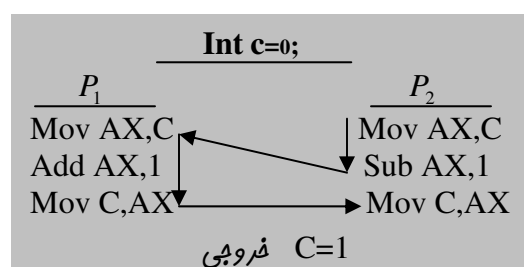
الف-



ب-



ج-



به علت اینکه هر دو پدازه از متغیر مشترک C استفاده کرده اند، مقدار نهائی C وابسته به ترتیب اجرای پدازه ها می باشد که در این مثال بسته به ترتیب اجرای دستورات C می تواند مقادیر 0, 1, -1 را اتخاذ نماید، ولی نتیجه مورد نظر صفر می باشد.

منظور از وضعیت مسابقه یا Race Condition چیست؟

وضعیت هائی که در آن فرایندهای متعددی، داده یکسانی را به طور همروند دستیابی و دستکاری می کنند و حاصل اجرا بستگی به ترتیب خاص دسترسی ها دارد، وضعیت مسابقه یا Race Condition گفته می شود.

نوامی بهرانی (Critical Section):

برای جلوگیری از شرایط رقابتی باید راهی را پیدا کنیم که از خواندن و نوشتن داده های مشترک، به طور همزمان، توسط بیش از یک پروسس جلوگیری به عمل آید. به عبارت دیگر ما به «انحصار متقابل» یا Mutual Exclusion نیاز داریم، به عبارتی دیگر اگر یکی از پدازه ها در حال استفاده از داده مشترک است باید مطمئن باشیم که دیگر پدازه ها، در آن زمان از انجام همان کار محروم می باشند.

بخشی از برنامه که به حافظه اشتراکی دسترسی دارد را قسمت یا ناحیه بهرانی (Critical Section) می نامیم. هر پدازه برای ورود به بخش بهرانی اش باید اجازه بگیرد. بخشی از کد پدازه که این اجازه گرفتن را پیاده سازی می کند بخش ورودی یا Entry section نام دارد.

بخش بهرانی می تواند با بخش خروجی یا exit section دنبال شود. این بخش خروجی کاری می کند که پدازه های دیگر بتوانند وارد ناحیه بهرانی شان بشوند. بقیه کد پردازش را بخش باقی مانده یا remainder section کوئیم. بنا براین سافتوئیر کلی پردازش ها به صورت زیر می باشد.

```
While(True){
```

```
    Entry section
```

```
    Critical_section();
```

```
    exit section
```

```
    Remainder_section();
```

```
}
```

باید جهت رفع مشکل وضعیت مسابقه چهار شرط زیر رعایت گردد تا یک راه حل خوب بدست آید

1- شرط انحصار متقابل (مانعه الجمعی Mutual Exclusion):

هنگامی که پردازشی در ناحیه بهرانی اش اجرا می گردد، هیچ پردازش دیگری نباید در ناحیه بهرانی حضور داشته باشد.

2- شرط پیشرفت یا پیشروی (Progress):

هنگامی که هیچ پردازشی در قسمت بهرانی در حال اجرا نباشد و تقاضاهائی برای ورود به بخش بهرانی وجود دارد، فقط پردازش هائی در تصمیم گیری برای ورود دقالت می کنند که هنوز به ناحیه بهرانی شان نرسیده باشند. به عبارت دیگر اگر پردازشی در قسمت باقی مانده (remainder) خود باشد، در تصمیم گیری اینکه که کدام پدازه وارد بخش بهرانی شود، شرکت داده نمی شود. به عبارت دیگر هیچ پردازشی نباید از بیرون ناحیه بهرانی خود امکان بلوکه کردن پردازش های دیگر را داشته باشد.

3- شرط انتظار مقید یا محدود (Bounded Waiting): یک برنامه منتظر ورود به ناهیه بهرانی، نباید به طور نامحدود در حالت انتظار باقی بماند

4- هر پردازشی با سرعت غیر صفر اجرا می شود ولی هیچ فرضی در مورد سرعت نسبی n پردازش و نیز تعداد CPU ها نمی کنیم

روش های جلوگیری از مسابقه (همزمانی پردازش ها)

1- غیر فعال ساختن وقفه ها؛

هر پردازش بلا فاصله پس از ورود به ناهیه بهرانی اش کلیه وقفه ها را از کار بیندازد و درست قبل از خروج از ناهیه بهرانی دوباره همه آنها را فعال کند. با خاموش ساختن وقفه ها CPU به هیچ عنوان نمی تواند از پردازشی به پردازش دیگر سوئیچ کند. پس پردازش می تواند بدون ترس از مداخله دیگر پردازش ها به دستکاری قسمت مشترک بپردازد.

این روش دو مشکل دارد

1- ممکن است کاربر وقفه ها را خاموش کند ولی دوباره آنها را فعال نسازد (یادش برود) بدین ترتیب سیستم از کار خواهد افتاد. پس اعطای قدرت غیر فعال ساختن وقفه ها به پردازش کاربر را عاقلانه نیست.

2- در سیستم های چند پردازنده ای غیر فعال ساختن وقفه ها، فقط در CPU ای اثر دارد که دستور از کار انداختن وقفه را اجرا می کند، بقیه CPU ها می توانند کار خودشان را ادامه داده و به حافظه مشترک دسترسی پیدا کنند.

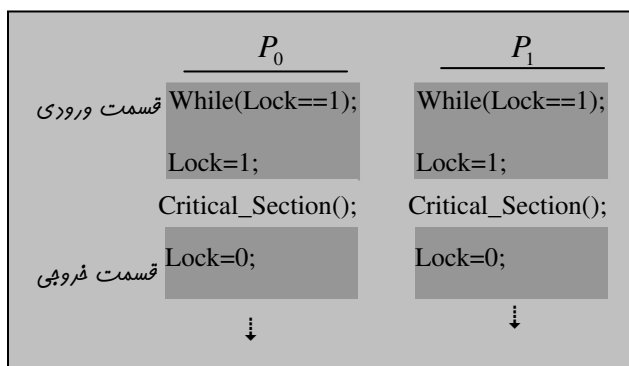
2- استفاده از متغیرهای قفل (Lock Variables):

فرض کنید یک متغیر قفل یکتا و مشترک با مقدار اولیه صفر وجود دارد (متغیر مثلا با نام Lock). هنگامی که پردازشی می خواهد وارد ناهیه بهرانی خود شود، ابتدا Lock را آزمایش می کند اگر $Lock=0$ بود آن را برابر 1 کرده و وارد ناهیه بهرانی می شود ولی اگر $Lock=1$ بود، باید در یک حلقه منتظر بماند تا Lock برابر صفر شود. بنا براین 0 به این معناست که هیچ پردازشی در ناهیه بهرانی نیست و 1 به این معناست که پردازشی در ناهیه بهرانی اش قرار دارد. کد زیر این روش را نشان می دهد، مقدار اولیه Lock برابر صفر است.

نکته: این روش شرط اصلی انحصار متقابل را ندارد

توضیح:

لمظه ای را تصور کنید که متغیر Lock برابر صفر است، پردازش P_0 در حلقه While متغیر Lock را چک می کند و چون برابر صفر است به سراغ خط بعدی می رود تا Lock را برابر 1 کند. ولی قبل از آنکه عدد 1 را در Lock بریزد، CPU به پردازش P_1 سوئیچ می کند. در این حال P_1 نیز متغیر Lock را برابر صفر می بیند و از حلقه While خارج می شود، آنگاه در ادامه Lock را برابر 1 کرده و وارد ناهیه بهرانی خود می شود. حال دوباره پردازنده به P_0 سوئیچ می کند، عدد 1 را در Lock ریخته و وارد ناهیه بهرانی P_0 می شود. یعنی هر دو پردازش P_0 و P_1 همزمان در ناهیه بهرانی می باشند!



3- روش تناوب قطعی (Strict Alternation):

در این روش از یک متغیر نوبت استفاده می شود طوری که اگر دو پردازش P_1 و P_2 داشته باشیم بعد از این که یکی از پردازش ها وارد ناهیه بهرانی شد، تا این که پردازش دیگر وارد ناهیه بهرانی نشود، این پردازش نمی تواند مجددا وارد ناهیه بهرانی بشود.

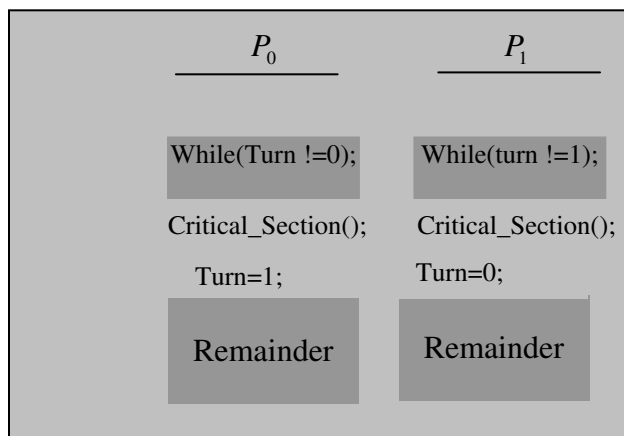
در این روش از یک متغیر نوبت استفاده می شود طوری که اگر دو پردازش P_1 و P_2 داشته باشیم بعد از این که یکی از پردازش ها وارد ناهیه بهرانی شد، تا این که پردازش دیگر وارد ناهیه بهرانی نشود، این پردازش نمی تواند مجددا وارد ناهیه بهرانی بشود.

```

Pi
Other = 1-i
While(Turn==Other);
Critical_Section();
Turn=Other;
Remainder Code

```

نکته: با این که این الگوریتم شرط انحصار متقابل را دارد ولی شرط پیشرفت در آن برقرار نیست



توضیح: حالتی را در نظر بگیرید که پردازنده P_0 وارد ناحیه بحرانی می شود که هنگام خروج از ناحیه بحرانی مقدار Turn را برابر 1 قرار می دهد و در این موقع پردازنده P_1 می تواند وارد ناحیه بحرانی شود. پردازنده P_1 پس از خروج از ناحیه بحرانی Turn را برابر صفر قرار می دهد، فرض کنید پردازنده P_1 در Remainder Code باشد اگر در همین اثنا پردازنده P_0 بخواهد وارد ناحیه بحرانی بشود چون Turn برابر صفر می باشد، وارد ناحیه بحرانی می شود این پردازنده پس از خروج از ناحیه بحرانی Turn را برابر 1 می کند. حال اگر قسمت Remainder Code مربوط به P_1 طولانی باشد، اگر P_0 بخواهد مجددا وارد

ناحیه بحرانی بشود، نمی تواند زیرا Turn برابر با 1 است، پس پردازنده P_1 که در قسمت Remainder Code خود قرار دارد در تصمیم گیری ورود به ناحیه بحرانی دقالت دارد و این یعنی نقض شرط پیشرفت

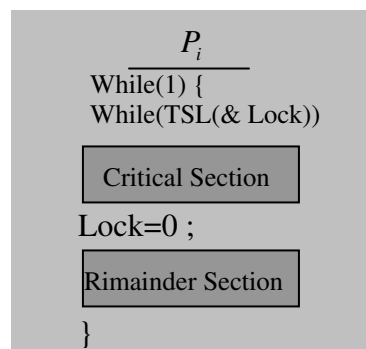
4- استفاده از دستور سفت افزاری (TSL):

دستورات اتمیک: دستوراتی هستند که تجزیه نا پذیرند (وقفه ناپذیر)

Mov Ax, Count اتمیک است

Count++ اتمیک نیست

در این روش از دستور اتمیک TSL استفاده می شود که طرز کار این دستور به صورت زیر است



پردازنده ها در هنگام ورود به ناحیه بحرانی دستور TSL را فراخوانی می کنند، که اگر مقدار Lock مخالف صفر باشد در حلقه گیر کرده و منتظر می مانند و اگر مقدار Lock صفر باشد، پردازنده ای که این دستور را اجرا کرده مقدار Lock را صفر می بیند و وارد ناحیه بحرانی می شود (در این لحظه مقدار Lock برابر 1 می شود)، اگر در همین عین پردازنده ی دیگری TSL را اجرا کند، چون مقدار Lock برابر 1 است نمی تواند وارد ناحیه بحرانی بشود، پردازنده ها پس از خروج از ناحیه بحرانی مقدار Lock را صفر می کنند تا دیگر پردازنده ها بتوانند وارد ناحیه بحرانی بشوند. (دقت شود که در اولین اجرا مقدار Lock برابر صفر است)

Tsl را می توانیم به این شکل بنویسیم

پایان جلسه چهارم

```
Tsl(x){
Int temp;
Temp=x;
x=1;
return (temp);
}
```

5- راه حل پیترسون :

```

int turn;
int flag[2]={0};
Enter(int pid)
{
    int other;
    turn=pid;
    flag[pid]=1;
    other=1-pid;
    while((turn=pid)&&(flag[other]== 1));
    .....
Exit(pid)
{
    flag[pid]=0;
}

```

توضیح: این الگوریتم از دو تابع `Enter()` و `Exit()` تشکیل یافته است. قبل از وارد شدن به نامیه بهرانی هر پردازش باید تابع `Enter()` را صدا زده و شماره پردازش خود را به عنوان آرگومان به آن بفرستد. این تابع باعث می شود که پردازش تا زمانی که بتواند بدون فطر وارد نامیه بهرانی شود، منتظر باقی بماند. پس از انجام کارهای نامیه بهرانی (دستکاری متغیرهای مشترک)، پردازش تابع `Exit()` را صدا می زند که نشان دهد کارش در نامیه بهرانی تمام شده و سایر پردازش ها در صورت نیاز می توانند وارد نامیه بهرانی بشوند. پردازش ها از دو متغیر مشترک `turn` و `flag` استفاده می کنند.

دو پردازش همکار P_0 و P_1 توابع فوق را به صورت زیر استفاده می کنند.

پردازش P_0
`Enter(0);`
`Critical_Section();`
`Exit(0);`

پردازش P_1
`Enter(1);`
`Critical_Section();`
`Exit(1);`

الف: این الگوریتم شرط انحصار متقابل را بر آورده می کند، زیرا فرض کنید هر دو پردازش باهم بخواهند وارد نامیه بهرانی بشوند، چون قبل از ورود به نامیه بهرانی حلقه `while()` را داریم، پردازش ای در حلقه می ماند که آخرین بار مقدار `turn` را تغییر داده باشد، و پردازش دیگر وارد حلقه می شود.

ب: شرط پیشرفت نیز برقرار است، زیرا هر پردازش به هنگام خروج از نامیه بهرانی، `flag` متناظر خود را صفر کرده و همین عمل اجازه ورود دیگر پردازش ها را با توجه به شرایط پردازش می دهد. به عبارتی پردازش ای که در قسمت R.C خود باشد در تصمیم گیری ورود دیگر پردازش ها به نامیه بهرانی دخالت ندارد.

از پنج راه حل گفته شده، دو راه حل آخر، که شرایط سه گانه را بر آورده می کنند، مشکل `Busy waiting` (انتظار مشغول) دارند، یعنی اینکه 1- یعنی اگر پروسسی بخواهد به نامیه بهرانی اش وارد شود ولی اجازه نداشته باشد، در یک حلقه بی کار می افتد، تا هنگامی که اجازه او صادر گردد. این روش باعث اتلاف وقت CPU می گردد.

2- فرض کنید دو پردازش یکی با الویت بالا و دیگری با الویت پایین در سیستم داشته باشیم، اگر پردازش الویت پایین در نامیه بهرانی باشد و پردازش الویت بالا از راه برسد و بخواهد وارد نامیه بهرانی بشود، داخل حلقه گیر می افتد. و چون الویت این پردازش بالاتر از الویت پردازش ای می باشد که در نامیه بهرانی است، بنابراین پردازش موجود در نامیه بهرانی هیچ فرصت اجرا (دست آوردن CPU) را ندارد، پس نمی تواند از نامیه بهرانی خارج شود و پردازش الویت بالا مدام در تست شرایط، مشغول می باشد (تا بی نهایت در حلقه دور می زند). پس هیچ کاری پیش نمی رود.

□ مشکل دیگر روش های گفته شده این است که قابل تعمیم به مسائل پیچیده نیستند.

6- سمافورها (Semaphores-راهنما ها)

سمافور x یک متغیر صحیح می باشد که جدای از مقدار دهی اولیه، فقط از طریق دو عمل اتمیک استاندارد زیر قابل دسترسی است

$$\text{wait}() \rightarrow p() - 1$$

$$\text{signal}() \rightarrow v() - 2$$

تعریف ساده p و v به صورت زیر است.

□ سمافور ها را می توان برای مساله بخش بهرانی n پردازش استفاده کرد.

فرض کنید دو پردازش p_1 و p_2 به شکل زیر اجرا شوند (مقدار اولیه سمافور 1 می باشد)

| p_1 | p_2 |
|---------|---------|
| $p(x);$ | $p(x);$ |
| $c++;$ | $c--;$ |
| $v(x);$ | $v(x);$ |

فرضا اگر ابتدا پردازش p_1 بخواهد اجرا شود با تابع p مقدار سمافور را صفر می کند و بعد وارد ناحیه بهرانی می شود (جهت دستکاری متغیر مشترک) و در انتها با دستور v مقدار سمافور را یک می کند، که قبل از یک شدن مقدار سمافور، اگر پردازش p_2 می خواست وارد ناحیه بهرانی شود در حلقه $\text{while}(x \leq 0)$ گیر می افتاد. (تا لحظه ای که پردازش p_1 دستور v را اجرا کند و دوباره مقدار سمافور یک شود)

□ اگر تعریف p و v به صورت داده شده باشد و سمافور x فقط در بردارنده یک مقدار باشد مشکل *Busy waiting* این جا نیز برقرار است. برای رفع این مشکل تعریف سمافور p و v را کاملتر می کنیم. در این حالت زمانی که پروسس، اجازه ورود به ناحیه بهرانی اش را ندارد بلوکه یا مسدود می شود (به جای آنکه در یک حلقه *while* پرخ بزند)، بدین ترتیب آن پروسس به حالت تعلیق می رود تا هنگامی که پروسس دیگری آن را بیدار (*wakeup*) کند □ سافتار واقعی سمافور، p و v به صورت زیر است.

Struct semaphore

```
{
List of process l;
Int value;
}
```

Semaphore x;

| $P(x)$ | $v(x)$ |
|---|--|
| $x.\text{value} - -;$ | $x.\text{value}++;$ |
| $\text{if}(x.\text{value} < 0) \text{ begin}$ | $\text{if}(x.\text{value} \leq 0) \text{ begin}$ |
| $\text{add process to } x.l;$ | $\text{remove a process } p \text{ from } x.l;$ |
| $\text{block the process};$ | $\text{wakeup}(p);$ |
| $\text{end};$ | $\text{end};$ |

توضیح: وقتی پردازشی عمل p را اجرا کرده و سمافور را غیر مثبت می یابد، باید صبر کند. اما به جای حلقه $while$ ، پردازش در صف انتظار مربوط به آن سمافور قرار داده می شود و حالت پردازش به $block$ تغییر می یابد. سپس کنترل به زمانبند CPU منتقل می گردد تا پردازش دیگری را برای اجرا، انتخاب کند. فرایندی که $block$ شده بر اثر اجرای عمل p توسط فرایند دیگر از سر گرفته می شود. این فرایند توسط یک عمل $wakeup$ بیدار شده و از حالت انتظار به حالت آماده می رود و در صف $Ready$ قرار داده می شود.

□ در ساختار جدید مقدار سمافور می تواند منفی باشد، که قدر مطلق این مقدار برابر تعداد پردازه هائی است که منتظر این سمافور هستند

□ اگر در هنگام اجرای دستور $wakeup(x)$ پردازش x در حالت بلوکه نباشد، این دستور هیچ عملی خاصی را انجام نمی دهد و اثری ندارد.

مسئله تولید کننده-مصرف کننده: دو پردازه تولید کننده و مصرف کننده وجود دارد که به طور همروند اجرا می شوند، تولید کننده یک سری اقلام اطلاعاتی را تولید می کند و در یک بافر که اندازه آن محدود است قرار می دهد. پردازه مصرف کننده این اقلام اطلاعاتی را از بافر برداشته و اجرا می کند.

نکته: دستورات $count++$ و $count--$ به صورت زیر عمل می کنند.

```
Int Buffer[max];
Int count=0;
Producer(){
While(1){
Produce an item;
Buffer[count]=item;
Count ++;
}
Concumer(){
Count--;
Item=Buffer[count];
Consum an item;
}
```

| <u>Count--</u> | <u>Count++</u> |
|----------------|----------------|
| Mov Ax,Count; | Mov Ax,Count; |
| Sub Ax,1; | Add Ax,1; |
| Mov Count,Ax; | Mov Count,Ax; |

□ محدودیت هائی که در این مسئله وجود دارند عبارتند از:

□ اگر بافر پر باشد و تولید کننده بخواهد یک قلم اطلاعاتی را در بافر قرار دهد، می بایست صبر کند تا بافر خالی شود.

□ اگر بافر خالی باشد و مصرف کننده بخواهد یک قلم اطلاعاتی را از بافر بردارد، می بایست صبر کند تا اطلاعاتی در بافر قرار داده شود.

□ مشکل دیگر مربوط به متغییر مشترک $count$ می باشد، به طور مثال فرض کنید $count = 3$ باشد، و پردازه مصرف کننده بخواهد یک

قلم اطلاعاتی را از بافر بردارد، در این صورت اگر این پردازه به دستور $Mov Ax,Count;$ برسد سه را در Ax قرار می دهد، حال قبل از اینکه مقدار تغییر یافته $count$ یعنی دو در Ax نوشته شود، $Context switching$ رخ می دهد و پردازه تولید کننده اجرا می شود. این

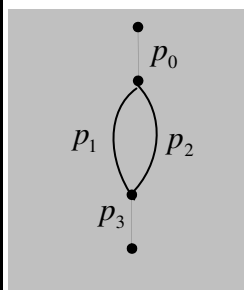
پردازه مقدار $count$ را سه می بیند، و بنا براین سه را در Ax قرار می دهد. اگر پردازه تولید کننده ابتدا، مقدار تغییر یافته $count$ را

بنویسد و بعد از آن پردازه مصرف کننده مقدار تغییر یافته $count$ را بنویسد، مقدار $count$ ، دو خواهد شد و در حالت بر عکس مقدار نهائی چهار خواهد شد. که هر دو حالت نادرست هستند، زیرا مقدار نهائی باید سه باشد، پس این مشکلات می بایست حل شوند. با استفاده از

سمافور ها می توانیم برنامه تولید کننده و مصرف کننده را به شکل زیر بنویسیم.

| | |
|----------------------------|---------------------|
| Semaphor mutex=1, Count=0; | Consumer(){ |
| Semaphor empty=max; | while(1){ |
| Semaphor full=0; | P(full); |
| Producer(){ | P(mutex); |
| While(1){ | Count - -; |
| Produce an item; | item=Buffer[count]; |
| P(empty); | V(mutex); |
| P(mutex); | V(empty); |
| Buffer[count]= item; | consum the item; |
| Count ++; | }} |
| V(mutex); | |

مثال: فرض کنید پردازش های p_0, p_1, p_2, p_3 می خواهند به صورت هم روند اجرا شوند. با استفاده از سمافور ها دستوراتی بنویسید، که هماهنگی بین پردازش ها را به صورت شکل فوق فراهم کنند



Semaphor $S_{12} = 0$ Semaphore $S_3 = 0$

Cobegin() {

$p_0(), p_1(), p_2(), p_3()$

}

$P_0() \{$

\vdots

$V(S_{12})$

$V(S_{12})$

$\}$

$P_1() \{$

$P(S_{12})$

\vdots

$V(S_3)$

$\}$

$P_2() \{$

$P(S_{12})$

\vdots

$V(S_3)$

$\}$

$P_3() \{$

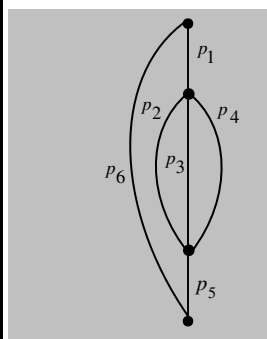
$P(S_3)$

$P(S_3)$

\vdots

$\}$

مثال: فرض کنید پردازش های $p_1, p_2, p_3, p_4, p_5, p_6$ می خواهند به صورت هم روند اجرا شوند. با استفاده از سمافور ها دستوراتی بنویسید، که هماهنگی بین پردازش ها را به صورت شکل فوق فراهم کنند



semaphor $s_1 = 0, s_{234} = 0;$

Cobegin() {

$p_1(), p_2(), p_3(), p_4(), p_5(), p_6()$

}

$p_1() \{$

\vdots

$V(s_1);$

$V(s_1);$

$V(s_1);$

$\}$

$p_2() \{$

$P(s_1)$

\vdots

$V(s_{234});$

$\}$

$p_3() \{$

$P(s_1)$

\vdots

$V(s_{234});$

$\}$

$p_4() \{$

$P(s_1)$

\vdots

$V(s_{234});$

$\}$

$p_5() \{$

$P(s_{234});$

$P(s_{234});$

$P(s_{234});$

\vdots

$\}$

$p_6() \{$

\vdots

$comands$

\vdots

$\}$

جلسه ششم

$Semaphor\ a = 1, b = 1;$

| | |
|----------|----------|
| $P_0 :$ | $P_1 :$ |
| $p(a);$ | $p(b);$ |
| $p(b);$ | $p(a);$ |
| \vdots | \vdots |
| $v(a);$ | $v(b);$ |
| $v(b);$ | $v(a);$ |

سمافورها می توانند سبب بن بست شوند دو پردازش P_0, P_1 را در نظر بگیرید.

فرض کنید P_0 دستور $p(a);$ را اجرا کرده باشد، در همین لحظه عمل تعویض متن رخ دهد و پردازش P_1 دستور $p(b);$ را اجرا کند، حال اگر پردازش P_1 دستور $p(a);$ را اجرا کند، یا پردازش P_0 دستور $p(b);$ را اجرا کند، هر دو بلاک می شوند (هر دو منتظر هم باقی می مانند)، در این حالت می گوئیم بین پردازش ها بن بست رخ داده است، پس سمافور ها می توانند سبب بن بست شوند.

7- مانیتور (monitor):

سمافور ها با همه مزایائی که دارند، پیچیده می باشند به عبارتی، در استفاده از سمافور ها می بایست خیلی دقت کرد، زیرا یک دستور v اضافی سبب مشکل می شود. مانیتور ها ابزارهای سطح بالائی هستند که این مشکل را ندارند، مانیتور یک سافتمان داده ای شامل توابع، متغیر های معمولی، متغیر های شرطی می باشند، که این متغیر های شرطی فقط از طریق دستورات $wait$ و $signal$ قابل دستیابی هستند، اگر پردازش ای در داخل مانیتور دستور $wait$ را صادر کند، آن پردازش در لیست پردازش های آن متغیر شرطی قرار گرفته، و بلاک می شود و پردازش های دیگر می توانند وارد مانیتور شوند، یکی از توابع داخل مانیتور را اجرا کنند. در هر لحظه فقط یک پردازش می تواند در داخل مانیتور باشد. یعنی فقط یک پردازش می تواند در هر آن در حال اجرای توابع داخل مانیتور باشد.

□ متغیر های شرطی، مقدار دهی نمی شوند و مقدار نمی گیرند.

□ دستور $signal$ سبب می شود اگر پردازش ای در صف بلاک آن متغیر شرطی باشد، بیدار شود و وارد مانیتور شده و اجرایش را از مملی که قبلا بلاک شده بود، ادامه دهد.

□ کنترل ورود به نامیه بهرانی را به هنگام استفاده از مانیتور، کامپایلر انجام می دهد، به طوری که کامپایلر اجازه نمی دهد بیش از یک پردازش در مانیتور اجرا شود، پس بهتر است نوامی بهرانی در مانیتور نوشته شود.

مسئله تولید کننده-مصرف کننده با استفاده از مانیتور

Monitor Tolid – Masraf

Condition full ,empty

| | | | |
|--------------------------|---------------------------|---------------------------------|-----------------------------------|
| $garardadan ()\{$ | $Bardash\ tan ()\{$ | $producer ()\{$ | $consumer ()\{$ |
| $if (count == max)$ | $if (count == 0)$ | $while (1)\{$ | $while (1)\{$ |
| $wait (full);$ | $wait (empty);$ | $produce\ an\ item ;$ | $Tolid - masraf .bardash\ tan();$ |
| $Buffer [count]=item ;$ | $count --;$ | $tolid - masraf .garardadan ()$ | $consum\ item ;$ |
| $count ++;$ | $item = Buffer [count];$ | \vdots | \vdots |
| $if (count == 1)$ | $if (count == max - 1)$ | $\}\}$ | $\}\}$ |
| $signal (empty)$ | $signal (full);$ | | |
| $\}$ | $\}$ | | |

□ مانیتور را می بایست حتماً کامپایلر پشتیبانی کند، ولی برای استفاده سمافور لزومی به پشتیبانی از طرف کامپایلر نیست، فقط کافی است سیستم عامل سمافور را پشتیبانی کند، که در این صورت می توان با استفاده از دستورات p, v پیاده سازی کرد.

بن بست پردازش ها (Process Deadlocks)

اگر مجموعه ای از پردازش ها در سیستم منتظر وقوع حادثه ای باشند که توسط دیگری انجام شود و هیچ کاری در سیستم پیش نرود، کوئیم سیستم دچار بن بست شده است.

شرایط وقوع بن بست: برای رخ دادن یک بن بست هر چهار شرط زیر باید برقرار باشند.

1- انحصار متقابل (Mutual Exclusion)

2- گرفتن و منتظر ماندن (Hold and wait)

3- عدم پس گرفتن (انحصاری بودن) (No preemption)

4- انتظار چرخشی (Circular Wait)

انحصار متقابل بدین معناست که منبع یا منابع در هر آن، فقط توسط یک پردازش قابل استفاده باشند.

گرفتن و منتظر ماندن یعنی پردازش یک سری منابع مورد نیازش را در اختیار گرفته، و منتظر منابعی است که در اختیار دیگر پردازش ها است عدم پس گرفتن بدین معناست که به اجبار نمی توان منبع یا منابعی را از پردازش پس گرفت.

انتظار چرخشی یعنی بایستی مجموعه ای از پردازش ها $\{p_0, p_1, \dots, p_n\}$ وجود داشته باشد به طوری که p_0 منتظر منبعی از p_1, p_1 منتظر منبع از p_2 و p_n منتظر منبعی از p_0 باشد.

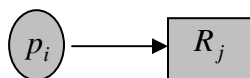
روش توصیف بن بست: استفاده از گراف تخصیص منابع (Resource Graph) $G(V, E) \leftarrow$

مستطیل $R =$ منابع R_1

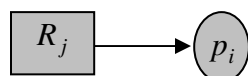
دایره $P =$ پردازش p_0

در گراف تخصیص منابع، گره ها از نوع منابع یا پردازش اند، که منابع با مستطیل نمایش داده می شوند و تعداد نمونه های آن با نقطه داخل آن مشخص می شوند و پردازش ها با دایره مشخص می شوند.

یال های گراف تخصیص منابع جهت دار می باشند که یا از پردازش به منابع می باشد (بدین معناست که پردازش p_i منتظر منبع R_j است) به این شکل.



یا از منابع به پردازش می باشد (بدین معناست که یک نمونه از منبع R_j در اختیار پردازش p_i می باشد)



به این شکل.

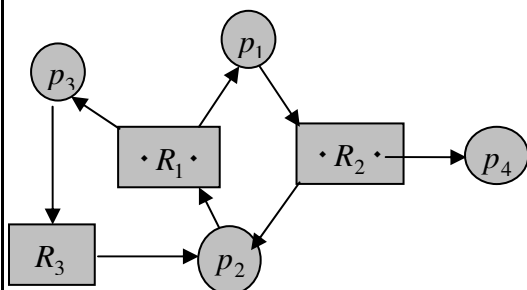
□ می توان اثبات کرد که اگر گراف دارای هیچ سیکلی (حلقه یا loop) نباشد، هیچ پردازشی در سیستم در بن بست نخواهد بود.

□ اگر گراف دارای سیکلی باشد، احتمال دارد بن بست وجود داشته باشد. پس وجود حلقه در گراف شرط لازم برای بن بست است و نه شرط کافی

□ اگر در گراف هر منبع دقیقاً یک نمونه داشته باشد، آنگاه اگر گراف حلقه داشته باشد، بدین معناست که حتماً بن بست رخ داده است، ولی اگر هر نوع منبع نمونه‌های متعددی داشته باشد، آنگاه حلقه الزاماً به معنای وقوع بن بست نیست

□ اگر در گراف حلقه‌ای وجود نداشته باشد، آنگاه سیستم در حالت بن بست نیست

مثال: آیا گراف زیر در بن بست قرار دارد.



حل: خیر

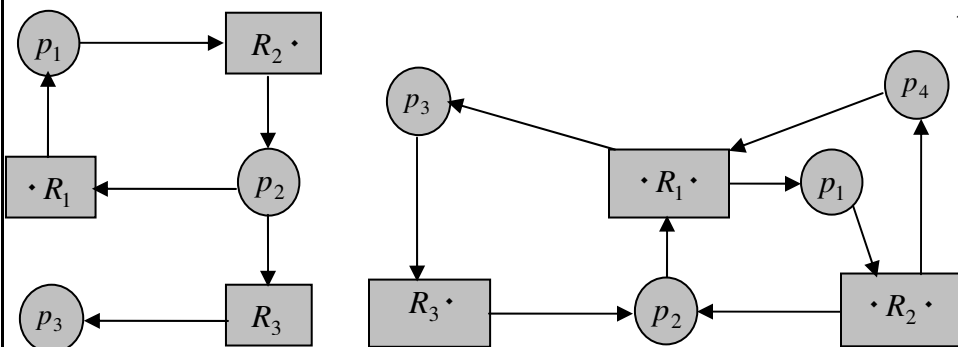
توضیح: در شکل گفته شده در گراف حلقه وجود دارد ولی سیستم (پردازه‌ها)

در بن بست نیستند، زیرا پردازه P_4 به منابع دیگری نیاز ندارد و بنابراین

اجرایش را به اتمام رسانده و منبع R_2 را، را می‌سازد با رها ساختن منبع R_2 بن بست شکسته می‌شود. منبع R_2 به پردازه P_1 تفصیل داده می‌شود پردازه P_1 تمام منابع مورد نیازش را در اختیار دارد، اجرایش را به اتمام رسانده و منابع را آزاد می‌سازد. در این حال منبع R_1 را به پردازه P_2 داده، پردازه P_2 نیز با در اختیار داشتن تمام منابع مورد نیازش، اجرایش را به پایان رسانده و تمام منابع را آزاد می‌سازد. که یکی از منابع R_3 است. منبع R_3 به پردازه P_3 تفصیل داده شده و این پردازه با در اختیار داشتن منابع اجرایش را به پایان می‌رساند.

مثال: آیا گراف‌های زیر در بن بست قرار دارند

حل: بله در هر دو بن بست وجود دارد.



فهرایند استفاده از منبع:

1- در خواست منبع (Request): اگر پردازه‌ای به منبعی نیاز داشته باشد، دستور سیستمی درخواست منبع را می‌دهد، که اگر آن منبع آزاد باشد، به پردازه درخواست‌کننده تفصیل داده می‌شود، و گرنه پردازه می‌بایست منتظر منبع باشد

2- بکارگیری منبع (allocation): در صورت وجود منبع درخواست شده، منبع به پردازه درخواست‌کننده توسط سیستم تفصیل داده می‌شود.

3- آزاد کردن منبع (Release): پردازه پس از استفاده از منبع تفصیل یافته، منبع را به سیستم برمی‌رداند.

□ در خواست منبع و آزاد کردن آن، توسط دستورات سیستم عامل (System Call) انجام می‌شود. مثلاً برای فایل دستورات open

و close و برای سمافور wait و signal انجام می‌شود. و همواره سیستم کلیه درخواست‌ها، تفصیل‌ها و آزاد شدن منابع را تحت نظر دارد

پایان جلسه ششم.

جلسه هفتم

روش های اداره بن بست (Dead lock handling)

4 روش برافورد با بن بست وجود دارد.

1- پیشگیری از بن بست: برای پیشگیری از بن بست می بایست کاری کرد که یکی از شرایط چهارگانه وقوع بن بست نقض شود. بدین ترتیب هرگز بن بست رخ نخواهد داد.

● **نقض انحصار متقابل:** اگر از منابعی به صورت اشتراکی استفاده کنیم بن بست هیچگاه رخ نمی دهد، نمونه آن فایل های فقط خواندنی هستند که می توانند همزمان توسط چندین پردازش استفاده شوند. البته این روش برای یک سری از منابع که ذاتا ماهیت غیر اشتراکی دارند قابل استفاده نیست مثل چاپگر. البته با *spool* کردن خروجی چاپگر چندین پروسس می توانند در یک زمان خروجی های خود را تولید کنند ولی تمام دستگاه ها را نمی توان *spool* کرد مثل جدول پروسس.

● **نقض گرفتن و منتظر ماندن:** برای نقض این شرط، باید مانع از ایجاد موقعیتی شد که در آن یک پردازش، منبعی را در اختیار گیرد و تقاضای منبع دیگری را نماید. رسیدن به این هدف با دو روش امکان پذیر است.

A. همه منابع مورد نیاز پردازش ها در ابتدای شروع اجرای پردازش اگر در دسترس باشند به آن اختصاص داده شوند وگرنه اختصاص داده نشوند به عبارتی یا همه منابع اختصاص داده شوند یا هیچکدام اختصاص داده نشوند. به عنوان مثال پردازش ای که می بایست داده ها را از نوار مغناطیس خوانده سپس آنها را بر روی دیسکی کپی کرده و آنها را مرتب کرده و چاپ کند، می بایست در ابتدا چاپگر را که در انتهای کارش نیاز دارد به دست آورد، پس از چاپگر به درستی استفاده نشده است، زیرا در همین حین که چاپگر بدون کار در اختیار این پردازش بوده، پردازش دیگری می توانست از این استفاده کند. پس عیب این پروتکل کاهش بهره وری سیستم است.

B. هر پردازش ای که یک سری منابع را در اختیار دارد، و طلب منابع دیگری می کند می بایست منابع در اختیارش را آزاد سازد، سپس تمام منابع را با هم تعویل بگیرد، مشکل این روش قحط زدگی (گرسنگی) است، یعنی پردازش ای که منابع متعددی نیاز دارد بایستی به طور نامعین در انتظار باشد چرا که به احتمال زیاد یکی از منابع مورد نیازش همواره توسط پردازش دیگری استفاده شده است.

● **نقض عدم پس گرفتن:** جهت تحقق این شرط دو راه حل وجود دارد.

A. یکی این است که اگر پردازش ای درخواست منابعی را دارد که آن منابع آزاد نباشند، تمام منابع در اختیار پردازش ی درخواست کننده پس گرفته شوند. B. راه حل دیگر آن است که بررسی شود که آیا پردازش های منتظر تمام منابع مورد نیاز پردازش کننده را دارند یا نه، که اگر داشته باشند منابع از پردازش های منتظر گرفته شده و به پردازش درخواست کننده انتساب یابد و اگر پردازش یا پردازش های منتظر، منابع مورد نیاز پردازش ی درخواست کننده را نداشته باشند، خود پردازش ی درخواست کننده باید منتظر بماند و ممکن است در حین انتظار منابع در اختیارش نیز از وی گرفته شوند و به پردازش های دیگری انتساب یابند.

● **نقض انتظار پرفشی:**

جهت نقض انتظار پرفشی می بایست منابع را شماره گذاری کرد به طوری که هر پردازش بتواند منابع را در جهت صعودی شماره هایشان درخواست کند. به عنوان مثال اگر پردازش ای منبع شماره 3 را در اختیار داشته باشد، منبع شماره 1 را نمی تواند درخواست کند، ولی می تواند منبع شماره 5 را درخواست کند. در این درخواست برای یک منبع می توان چندین نمونه از آن منبع را طلب کرد. شماره گذاری منابع می بایست بر اساس ترتیب نیاز به منابع صورت گیرد. به عنوان مثال، در مثال قبل می بایست شماره ی نوار مغناطیسی کمتر از شماره ی دیسک و شماره ی دیسک کمتر از شماره ی چاپگر باشد. جهت اثبات اینکه روش گفته شده انتظار پرفشی را نقض می کند از برهان خلف استفاده می کنیم. فرض کنیم با اعمال این

شرط باز هم انتظار پرفشی داشته باشیم. در این صورت اگر شماره منبع در اختیار پردازش $f(R_{i-1})$ باشد، این پردازش در صورتی منتظر P_i است که $f(R_{i-1}) < f(R_i)$ باشد و با تعمیم این رابطه به رابطه ی

$$f(R_0) < f(R_1) < \dots < f(R_n) < f(R_0) \quad f(R_n) < f(R_0)$$

نیست و برهان خلف اثبات می شود.

عیب روش پیشگیری از بن بست، بهره وری پایین منابع و کاهش توان عملیاتی سیستم می باشد.

2- اجتناب از بن بست:

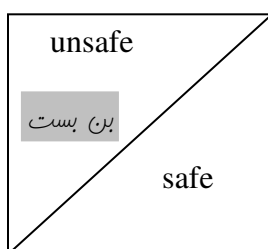
در این روش با توجه به اطلاعاتی نظیر حداقل نیاز پردازش ها به منابع و منابع تخصیص یافته به پردازش ها و موجودی، وقتی پردازش ای درخواست یک سری منابع را داشته باشد، اگر منابع موجود باشند، بررسی می کنیم که آیا با اجابت این درخواست سیستم به حالت امن می رود یا نه. اگر سیستم به حالت امن برود درخواست تخصیص داده می شود و گرنه از درخواست اجتناب می شود.

حالت امن حالتی است که پردازش ها می توانند به ترتیب خاص منابع مورد نیازشان را گرفته و اجرایشان را با موفقیت پشت سر بگذارند.

تعریف دیگر: حالتی است که در آن یک ترتیب امن (Safe sequence) از پردازش ها وجود داشته باشد.

تعریف دیگر: اگر سیستم بتواند منابع مورد درخواست را به ترتیبی تخصیص دهد که از بروز بن بست اجتناب شود کوئیم آن سیستم در حالت امن است.

Safe sequence (ترتیب امن) ترتیبی است از پردازش ها که می توانند با ترتیب معینی از تحویل گرفتن منابع، اجرایشان را خاتمه دهند.



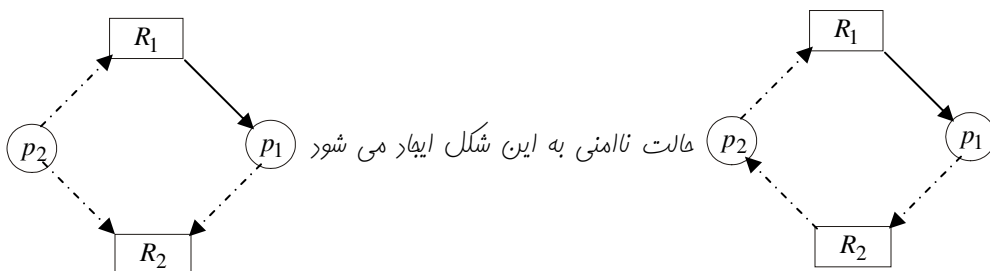
برای اجتناب از بن بست دو راه وجود دارد:

1- از هر منبع فقط یک نمونه وجود داشته باشد: از گراف شبیه گراف تخصیص منابع استفاده می شود.

2- از هر منبع بیش از یک نمونه وجود داشته باشد. از الگوریتم بانکداران استفاده می شود.

گراف تخصیص منابع: این گراف شبیه گراف تخصیص منابع است که علاوه بر یال های انتظار و تخصیص شامل یک یال ادعا (claim edge) می باشد که با نقطه چین نمایش داده می شود. به طوریکه R_i بدین معناست که ممکن است در آینده پردازش P_i درخواست منبع R_i را داشته باشد. هر یال تخصیص پس از پایان تخصیص تبدیل به یال ادعا می شود.

با توجه به شکل اگر P_2 در لحظه ی فعلی درخواست R_2 را داشته باشد، این درخواست به وی اعطا نمی شود، زیرا سیستم به حالت نا امن می رود به طوریکه ممکن است در آینده P_2 طلب R_1 و P_1 طلب R_2 را داشته باشد، که بن بست رخ می دهد



اگر در گراف بالای P_1 درخواست R_2 را داشته باشد، آیا این درخواست اجابت شود یا خیر؟

بله، اجابت می شود، زیرا در آینده ایبار حلقه غیر ممکن است.

الگوریتم بانکداران: این الگوریتم اولین بار توسط دیسترا ارائه شد، و به نام الگوریتم بانکدار معروف گردید، چرا که این الگوریتم شبیه، رفتار یک بانکدار شهر کوچک با مشتریانش طراحی شده است، یک بانکدار هرگز تمام سرمایه خودش را به مشتریان تفصیل نمی دهد و طوری عمل می کند که بتواند کلیه نیازهای مشتریانش را برآورده کند.

ساقتمان داده های مورد نیاز (n تعداد پردازش ها و m تعداد منابع)

1. $Max_{n \times m}$

$Max[i][j] = k$ بدین معناست که پردازش p_i ، جهت اجرای k نمونه از منبع R_j نیاز دارد

2. $Alocation_{n \times m}$

$Alocation[i][j] = k$ به این معناست که در حال حاضر k نمونه از منبع R_j در اختیار پردازش p_i ، می باشد.

3. $Need_{n \times m}$

$Need[i][j] = k$ به این معناست که پردازش p_i جهت ادامه کارش به k نمونه از منبع R_j نیاز دارد.

بدین معنی است که $Need[i][j] = Max[i][j] - Alocation[i][j]$ می باشد

بردار $Available$ (در دسترس) به طول m تعداد منابع آزاد از هر نوع را نشان می دهد. مثلا $Available[j] = k$ باشد یعنی از منبع نوع R_j به تعداد k نمونه در دسترس وجود دارد.

اگر $Request_i$ بردار درخواست منابع، پردازش p_i باشد الگوریتم به صورت زیر نوشته می شود.

1- اگر $Need_i < Request_i$ (یعنی سطر مربوط به پردازش p_i در ماتریس) باشد، این درخواست غیر قانونی است، و بررسی نمی شود (یعنی پایان الگوریتم)
 دو بردار x و y را به طول n در نظر بگیرید. می گوئیم $x \leq y$ اگر و فقط اگر $x[i] \leq y[i]$ باشد به ازاء $i = 1, 2, \dots, n$. مثلا
 $(0, 4, 2, 3) \leq (2, 8, 3, 5)$

2- اگر $Request_i > Available$ باشد بدین معناست که منابع کافی جهت تفصیل به پردازش p_i ، وجود ندارد، و p_i می بایست منتظر بماند.

3- وانمود می کنیم که منابع مورد درخواست پردازش p_i ، به وی اعطاء می شود، بنابراین ساقتمان داده ها به صورت زیر تغییر می یابند.

$Alocation_i += Request_i$

$Need_i = Request_i$

$Available_i = Request_i$

4- فراخوانی الگوریتم امنیت: اگر حالت سیستم ناامن باشد ساقتمان داده های تغییر یافته در قسمت 3 به حالت اول برگردانده می شود (اجتناب از بن بست)

5- پایان

⊗ مرتبه اجرای الگوریتم بانکدار $m \times n^2$ می باشد (n تعداد پردازش ها و m تعداد انواع منابع است)

الگوریتم امنیت

استفاده از بردارهای $work$ به طول m (متناظر با منابع) و $Finish$ به طول n (متناظر با پردازش ها)

1- $work = Available$ و برای $i = 1 \dots n$: $Finish[i] = False$

2- به ازای $i = 1, \dots, n$ (به ازای تمام مقایر پردازش ها) مقدار i را چنان بیابید که $Finish[i] = False$ و $Need_i \leq work$ اگر چنین ای پیدا نشد برو به مرحله ی 4

(در این مرحله به دنبال پردازش ای می گردیم که به پایان نرسیده باشد و با منابع موجود بتوان اجراش را به پایان رساند)

3- زمانی به مرحله سه می آییم که پردازش ای پیدا شود که اجراش تمام نشده و با استفاده از منابع موجود می تواند نیاز هایش را بر آورده کرده و

اجراش را به اتمام برساند بنابراین پس از اتمام اجراش منابع در اختیارش را آزاد می سازد. $work += Allocation$, $finish[i] = true$

4- اگر برای تمام i ها $Finish[i] = true$ ($i=1 \dots n$) باشد، بدین معناست که تمام پردازش ها می توانند با منابع موجود اجراشان را با موفقیت پشت

سر بگذرانند و سیستم در حالت امن قرار دارد ولی اگر حداقل یک i پیدا شود که $Finish[i] = false$ باشد بدین معناست که با منابع موجود این پردازش

نمی تواند اجراش را به اتمام برساند و سیستم در حالت ناامن قرار دارد.

مثال. فرض کنید سیستمی با چهار نوع منبع R_1, R_2, R_3, R_4 که هر کدام به ترتیب دارای 8, 5, 9, 7 نمونه می باشند، موجود باشد اگر در

این سیستم 5 پردازش p_1 الی p_5 را داشته باشیم، با توجه به ماتریس های max و $Allocation$ تشخیص دهید سیستم در حالت امن است یا

نه؟

| process | R_1 | R_2 | R_3 | R_4 |
|---------|-------|-------|-------|-------|
| p_1 | 3 | 2 | 1 | 4 |
| p_2 | 0 | 2 | 5 | 2 |
| p_3 | 5 | 1 | 0 | 5 |
| p_4 | 1 | 5 | 3 | 0 |
| p_5 | 3 | 0 | 3 | 3 |

max

| process | R_1 | R_2 | R_3 | R_4 |
|---------|-------|-------|-------|-------|
| p_1 | 2 | 0 | 1 | 1 |
| p_2 | 0 | 1 | 2 | 1 |
| p_3 | 4 | 0 | 0 | 3 |
| p_4 | 0 | 2 | 1 | 0 |
| p_5 | 1 | 0 | 3 | 3 |
| sum | 7 | 3 | 7 | 5 |

Allocation

حل. ابتدا ماتریس $Need$ را می سازیم با توجه به $Need = Max - Allocation$ خواهیم داشت .

| process | R_1 | R_2 | R_3 | R_4 |
|---------|-------|-------|-------|-------|
| p_1 | 1 | 2 | 0 | 3 |
| p_2 | 0 | 1 | 3 | 1 |
| p_3 | 1 | 1 | 0 | 2 |
| p_4 | 1 | 3 | 2 | 0 |
| p_5 | 2 | 0 | 0 | 3 |

Need

حال بردار $Available$ را با توجه به فرمول $sum -$ موجودی اولیه $Available =$ ایبار می کنیم

و بر دار $finish$ و $work$ نیز در ابتدا به شکل زیر می باشند.

| | R_1 | R_2 | R_3 | R_4 |
|--------|-------|-------|-------|-------|
| $work$ | 1 | 2 | 2 | 2 |

| | p_1 | p_2 | p_3 | p_4 | p_5 |
|----------|-------|-------|-------|-------|-------|
| $finish$ | f | f | f | f | f |

| | R_1 | R_2 | R_3 | R_4 |
|-------------|-------|-------|-------|-------|
| $Available$ | 1 | 2 | 2 | 2 |

پردازه هائی را که *false* هستند با *work* مقایسه می کنیم. (مقایسه سطرهای ماتریس *Need* با *work*). که در اولین مقایسه ملاحظه می شود که فقط پردازه p_3 کوچکتر است و می تواند اجرایش را با موفقیت انجام دهد و پس از اجرا منابع را برگرداند، کار را به همین ترتیب ادامه می دهیم.

| | | | | | | | | | | |
|------|-------|-------|-------|-------|--------|-------|-------|-------|-------|-------|
| | R_1 | R_2 | R_3 | R_4 | | P_1 | P_2 | P_3 | P_4 | P_5 |
| work | 5 | 2 | 2 | 5 | finish | f | f | T | f | f |
| | R_1 | R_2 | R_3 | R_4 | | P_1 | P_2 | P_3 | P_4 | P_5 |
| work | 7 | 2 | 3 | 6 | finish | T | f | T | f | f |
| | R_1 | R_2 | R_3 | R_4 | | P_1 | P_2 | P_3 | P_4 | P_5 |
| work | 7 | 3 | 5 | 7 | finish | T | T | T | f | f |
| | R_1 | R_2 | R_3 | R_4 | | P_1 | P_2 | P_3 | P_4 | P_5 |
| work | 7 | 5 | 6 | 7 | finish | T | T | T | T | f |
| | R_1 | R_2 | R_3 | R_4 | | P_1 | P_2 | P_3 | P_4 | P_5 |
| work | 8 | 5 | 9 | 7 | finish | T | T | T | T | T |

ملاحظه می شود که سیستم در حالت امن است و ترتیب اجرا به صورت P_3, P_1, P_2, P_4, P_5 می باشد در همین سؤال پردازه p_3 درخواست یک نمونه از منبع R_3 را داشته باشد آیا این درخواست پاسخ داده شود یا خیر؟

مثال. سیستمی با پنج فرایند p_0 الی p_4 و سه نوع منبع A, B, C را در نظر بگیرید منبع نوع A دارای 10 نمونه، منبع نوع B دارای 5 نمونه و منبع نوع C دارای 8 نمونه می باشد، فرض کنید در زمان t_0 وضعیت سیستم به صورت زیر باشد

| process | A | B | C |
|---------|---|---|---|
| p_0 | 7 | 5 | 3 |
| p_1 | 3 | 2 | 2 |
| p_2 | 9 | 0 | 2 |
| p_3 | 2 | 2 | 2 |
| p_4 | 4 | 3 | 3 |

max

| process | A | B | C |
|---------|---|---|---|
| p_0 | 0 | 1 | 0 |
| p_1 | 2 | 0 | 0 |
| p_2 | 3 | 0 | 2 |
| p_3 | 2 | 1 | 1 |
| p_4 | 0 | 0 | 2 |

Allocation

الف. ماتریس Need را بیابید

ب. آیا در لحظه فعلی سیستم در حالت امن است یا نه؟ اگر در حالت امن است توالی امن را بیابید

ج. فرض کنید در لحظه t_0 فرایند p_1 یک نمونه از منبع A و دو نمونه دیگر از منبع C درخواست کند آیا این درخواست فوراً پاسخ داده شود یا خیر.

د. اگر پردازنده p_4 در خواست سه نمونه از منبع A و سه نمونه دیگر از منبع B را درخواست کند، آیا این درخواست می بایست برآورده شود یا خیر.

حل.

| process | A | B | C |
|---------|---|---|---|
| p_0 | 7 | 4 | 3 |
| p_1 | 1 | 2 | 2 |
| p_2 | 6 | 0 | 0 |
| p_3 | 0 | 1 | 1 |
| p_4 | 4 | 3 | 1 |

Need

الف. با توجه به $Need = \max - Allocation$ داریم

ب. ابتدا بردار Available را می سازیم

$$C \text{ (منابع موجود آزاد)} = 10 - (2 + 3 + 2) = 3 \quad B \text{ (منابع موجود آزاد)} = 5 - (1 + 1) = 3 \quad A \text{ (منابع موجود آزاد)} = 7 - (2 + 1 + 2) = 2$$

پس بردار منابع در دسترس (Available) به صورت (3,3,2) می باشد که با توجه به این بردار مسئله را حل می کنیم.

$$(3,3,2) \xrightarrow{p_1} (5,3,2) \xrightarrow{p_3} (7,4,3) \xrightarrow{p_4} (7,4,5) \xrightarrow{p_2} (10,4,7) \xrightarrow{p_0} (10,5,7)$$

همانطور که مشاهده می شود ترتیب اجراء p_1, p_3, p_4, p_2, p_0 یک ترتیب امن است. لذا سیستم در حالت امن قرار دارد

نکته: اگر در یک لحظه چندین پروسس شرط لازم را برای اجرا شدن داشته باشند، اهمیتی ندارد که کدام یک جهت تخصیص منابع انتخاب شود چرا که در هر حال آن پروسس پس از تخصیص منابع مورد نیاز و تمام شدن کل منابع خود را آزاد می کند.

ب. اگر درخواست فوق را برآورده سازیم ماتریس های Allocation و Need به صورت زیر در خواهند آمد

| process | A | B | C |
|---------|----------|---|----------|
| p_0 | 0 | 1 | 0 |
| p_1 | 3 | 0 | 2 |
| p_2 | 3 | 0 | 2 |
| p_3 | 2 | 1 | 1 |
| p_4 | 0 | 0 | 2 |

Allocation

| process | A | B | C |
|---------|----------|---|----------|
| p_0 | 7 | 4 | 3 |
| p_1 | 0 | 2 | 0 |
| p_2 | 6 | 0 | 0 |
| p_3 | 0 | 1 | 1 |
| p_4 | 4 | 3 | 1 |

Need

$$(2,3,0) \xrightarrow{p_1} (5,3,2) \xrightarrow{p_3} (7,4,3) \xrightarrow{p_4} (7,4,5) \xrightarrow{p_0} (7,5,5) \xrightarrow{p_2} (10,5,7)$$

همانطور که مشاهده می شود ترتیب اجراء $\langle p_1, p_3, p_4, p_0, p_2 \rangle$ یک ترتیب امن است. لذا اگر به درخواست های p_1 در لفظه t_0 پاسخ بگوئیم مطمئن هستیم که سیستم در حالت امن باقی خواهد ماند.

ب. اگر درخواست فوق را برآورده سازیم ماتریس های $Need$ و $Allocation$ به صورت زیر در خواهند آمد

| process | A | B | C |
|---------|----------|----------|---|
| p_0 | 0 | 1 | 0 |
| p_1 | 2 | 0 | 0 |
| p_2 | 3 | 0 | 2 |
| p_3 | 2 | 1 | 1 |
| p_4 | 3 | 3 | 2 |

Allocation

| process | A | B | C |
|---------|----------|----------|---|
| p_0 | 7 | 4 | 3 |
| p_1 | 1 | 2 | 2 |
| p_2 | 6 | 0 | 0 |
| p_3 | 0 | 1 | 1 |
| p_4 | 1 | 0 | 1 |

Need

(0,0,2) منابع آزاد

حال اگر منابع آزاد (بردار Available) را با هر سطر

ماتریس $Need$ مقایسه کنیم، هیچ کدام از سطرها

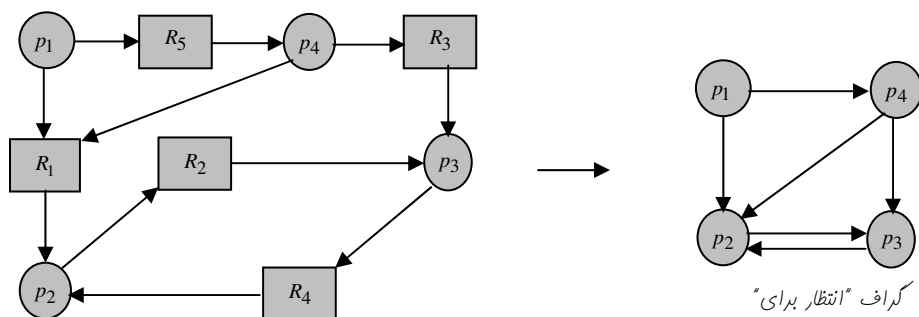
ماتریس $Need$ از بردار مذکور کمتر نیست لذا این

وضعیت نا امن می باشد

3- روش تشفیص بن بست و بازیافت سیستم :

تشفیص بن بست برای حالت یک نمونه از هر منبع

در این روش از روی کراف تفصیص منبع، کراف انتظار (wait-for graph) را بدست می آوریم. برای بدست آوردن کراف انتظار، کره های منبع را از کراف تفصیص حذف کرده و کمان های مناسبی را باهم ترکیب می کنیم. مثال. فرض کنید کراف تفصیص منابعی به شکل زیر باشد، اولاً کراف "انتظار برای" را رسم کنید ثانیاً مشفص نمائید که سیستم دچار بن بست شده است یا نه.



چون حلقه وجود دارد سیستم دچار بن بست شده است

کراف "انتظار برای"

کراف تفصیص منبع

یک پیکان از پردازش p_i به p_j در کراف انتظار، نمایانگر این است که پردازش p_i در انتظار پردازش p_j است تا منبعی را که p_i به آن نیاز دارد، آزاد کند. سیستم عامل، کراف انتظار را نگهداری کرده و مرتباً آن را بررسی می کند. اگر در کراف انتظار حلقه وجود داشته باشد، آنگاه سیستم به بن بست رسیده است. الگوریتم تشفیص بن بست در کراف از مرتبه $O(n^2)$ می باشد که n تعداد رئوس کراف است.

تشفیص بن بست برای حالت چند نمونه از هر منبع

روش کراف انتظار برای این حالت قابل استفاده نیست. در حالتی که هر منبع چند نمونه دارد می بایست از روشی شبیه بانگداران استفاده کنیم. در این روش به جای ماتریس Need از ماتریس $n \times m$ به نام Request استفاده می شود که نیازهای فعلی هر پردازش را نشان می دهد. اگر $Request[i][j] = k$ باشد یعنی پردازش p_i ، k نمونه بیشتر از منبع R_j را درخواست کرده است. الگوریتم این روش به شکل زیر خواهد بود.

- 1- ابتدا بردار $work$ را مساوی Available قرار می دهیم ($work = Available$) اگر به ازای $i = 0 \dots n$ ، $Allocation(i) \neq 0$ ، $finish[i] = fals$ و گرنه $finish[i] = True$
- 2- به ازای $i = 0 \dots n$ ، یک i چنان پیدا کنیم که $finish[i] = fals$ باشد و $Request[i] > work$ (سطر مربوط به پردازش p_i در ماتریس Request) باشد. اگر چنین i پیدا نشد برو به مرحله 4.

3- به ازای i پیدا شده در مرحله دو $finish[i]$ را برابر True قرار می دهیم و $work$ را با Allocation مربوطه جمع می کنیم ($work += Allocation$)، زیرا پردازشهای پیدا شده که با منابع موجود میتواند اجرایش را کامل کند و پس از فاصله اجراء منابع در اختیارش را آزاد سازد، دوباره برو به مرحله 2.

4- اگر پردازش p_i یافت شود که $finish[i] = false$ باشد، بدین معناست که p_i در بن بست قرار دارد. (سیستم دچار بن بست شده است)

□ پیچیدگی این الگوریتم $(m \times n)$ می باشد که m تعداد منابع است و n تعداد پردازش ها می باشد.

مثال. سیستمی با 5 پردازش $p_1 \dots p_5$ و سه نوع منبع A, B, C را در نظر می گیریم. A دارای 7 نمونه، منبع نوع B دارای 2 نمونه و C دارای 6 نمونه است. فرض کنید در زمان t_0 وضعیت سیستم به صورت زیر باشد آیا سیستم در بن بست قرار دارد یا خیر؟
ب. فرض کنید p_3 نمونه دیگری از منبع C را درخواست کند آیا سیستم دچار بن بست می شود یا خیر؟

وضعیت اولیه سیستم

| Process | A | B | C |
|---------|---|---|---|
| p_1 | 0 | 1 | 0 |
| p_2 | 2 | 0 | 0 |
| p_3 | 3 | 0 | 3 |
| p_4 | 2 | 1 | 1 |
| p_5 | 0 | 0 | 2 |

Allocation

| Process | A | B | C |
|---------|---|---|---|
| p_1 | 0 | 0 | 0 |
| p_2 | 2 | 0 | 2 |
| p_3 | 0 | 0 | 0 |
| p_4 | 1 | 0 | 0 |
| p_5 | 0 | 0 | 2 |

Request

حل: بردار Available برابر خواهد بود با: (0,0,0)، پس داریم

| Work | A | B | C |
|------|---|---|---|
| | 0 | 0 | 0 |

| Work | A | B | C |
|------|---|---|---|
| | 0 | 1 | 0 |

| Work | A | B | C |
|------|---|---|---|
| | 3 | 1 | 3 |

| Work | A | B | C |
|------|---|---|---|
| | 5 | 1 | 3 |

| Work | A | B | C |
|------|---|---|---|
| | 7 | 2 | 4 |

| Work | A | B | C |
|------|---|---|---|
| | 7 | 2 | 6 |

| finish | p_1 | p_2 | p_3 | p_4 | p_5 |
|--------|-------|-------|-------|-------|-------|
| | f | f | f | f | f |

| finish | p_1 | p_2 | p_3 | p_4 | p_5 |
|--------|-------|-------|-------|-------|-------|
| | T | f | f | f | f |

| finish | p_1 | p_2 | p_3 | p_4 | p_5 |
|--------|-------|-------|-------|-------|-------|
| | T | f | T | f | f |

| finish | p_1 | p_2 | p_3 | p_4 | p_5 |
|--------|-------|-------|-------|-------|-------|
| | T | T | T | f | f |

| finish | p_1 | p_2 | p_3 | p_4 | p_5 |
|--------|-------|-------|-------|-------|-------|
| | T | T | T | T | f |

| finish | p_1 | p_2 | p_3 | p_4 | p_5 |
|--------|-------|-------|-------|-------|-------|
| | T | T | T | T | T |

پس توالی $\langle p_1, p_3, p_2, p_4, p_5 \rangle$ امکان پذیر است و سیستم در بن بست قرار ندارد.

ب. در این صورت ماتریس Request به شکل زیر می باشد. هر که می توان منابع p_1 را باز پس گرفت، ولی این تعداد منابع موجود برای انجام تقاضاهای هیچ پردازشی کافی نخواهد بود. لذا بن بستی شامل پردازش های p_2, p_3, p_4, p_5 پدید می آید.

| Process | A | B | C |
|---------|---|---|---|
| p_1 | 0 | 0 | 0 |
| p_2 | 2 | 0 | 2 |
| p_3 | 0 | 0 | ① |
| p_4 | 1 | 0 | 0 |
| p_5 | 0 | 0 | 2 |

بن بست رخ می دهد $(0,0,0) \xrightarrow{p_1} (0,1,0)$ منابع

زمان فراخوانی الگوریتم تشفیص بن بست:

سؤال مهم این است که الگوریتم تشفیص بن بست را چه زمان هائی باید اجرا کنیم. در یک حالت حد، میتوان در هر بار درخواستی که سریعاً قابل اعطاء نمی باشد، این الگوریتم را اجرا کنیم. بدین ترتیب علاوه بر اینکه می توانیم مجموعه پردازش های موجود در بن بست را تشفیص دهیم، پردازش فاضی که باعث بن بست شده است نیز مشفیص می شود. ولی از طرف دیگر با توجه به زمانگیر بودن الگوریتم های تشفیص بن بست، اجرای مکرر آنها باعث کاهش کارائی سیستم می گردد. یک روش کم هزینه تر آن است که الگوریتم مذکور را با پریود کمتری اجرا کنیم. مثلاً در هر ساعت یکبار، یا هر بار که بهره وری CPU به زیر 40% برسد، یا هر بار که سیستم کم است. اگر این الگوریتم در زمانهای دلفواهی اجرا شود، ممکن است حلقه های بسیاری در گراف منبع وجود داشته باشند و بدین ترتیب در حالت کلی نمی توان گفت کدام پردازش باعث بن بست شده است.

ترمیم یا بازیافت سیستم:

یک روش برای ترمیم بن بست این است که بعد از اینکه بن بست تشفیص داده شد، به اپراتور خبر داده شود که سیستم دچار بن بست شده است که در این صورت اپراتور به صورت دستی سیستم را می بایست ترمیم کند. ولی اگر خود سیستم عامل بخواهد آن را باز یافت کند دو راه وجود دارد.

1- فائمه دادن به پردازش ها:

□ تمام پردازش های درگیر بن بست فائمه داده شوند، که این راه حل هر چند سربار ناشی از اجرای متعدد الگوریتم های تشفیص بن بست را ندارد ولی ممکن است سیستم متممل هزینه زیادی شود، چرا که ممکن است پردازش هائی دستورات زیادی را اجرا کرده باشند که بدین ترتیب می بایست دوباره از ابتدا اجرا شوند.

□ روش دوم این است که هر بار یک پردازش انتقاب شده و فائمه داده میشود، سپس الگوریتم کشف بن بست فراخوانی می شود. اگر سیستم دچار بن بست باشد، مجدداً پردازش دیگری انتقاب شده و فائمه داده می شود، تا جائی که سیستم از بن بست خارج شود. اشکال این روش سربار زمانی ناشی از الگوریتم کشف بن بست می باشد. انتقاب یک پردازش جهت فائمه دادن می بایست بر اساس هزینه کمینه باشد، که هزینه کمینه می تواند به فاکتور های زیر بستگی داشته باشد.

1- نوع پردازش (دسته ای یا همواره ای)

2- مدت زمانی که از اجرای پردازش سپری شده است

3- مدت زمانی که اجرای پردازش باقی مانده است

4- تعداد و نوع منابعی که در اختیار پردازش است

5- در صد استفاده پردازش از منابع

6- تعداد منابعی که برای کامل شدن نیاز دارد

2- پس گرفتن منابع:

در این روش منابعی از یک پردازش گرفته شده و در اختیار پردازش دیگری قرار داده می شود. برای این کار باید سه موضوع مشفیص گردد.

الف. انتقاب منبع و پردازش های مورد نظر

ب. بازگرداندن به عقب (Rollback) یعنی پردازشی که منبع او پس گرفته شده، باید به حالت امنی به عقب برگردانده شود، تا بعداً از آن حالت مجدداً اجرائش را از سر بگیرد.

ج. قسطی زدگی یعنی باید تضمین کرد که منابع همواره از یک پردازش فاض بازپس گرفته نشود، چرا که در آن صورت اجرای آن پردازش مرتباً به تعویق می افتد

4- **روش صرف نظر کردن از بن بست (الگوریتم استریخ ostrich) :** در این روش در واقع هیچ عملی در مقابل بن بست انجام داده نمی شود. در صورتی که بن بست منجر به از کار افتادن سیستم شود (Hang) آنگاه سیستم به صورت دستی ری ست (reset) می شود.

□ جالب است که بدانید در اکثر سیستم عامل های امروزی مثل unix از همین روش چهارم استفاده می شود، چرا که در این سیستم ها بن بست به ندرت رخ می دهد (مثلا سالی یک بار) لذا ارزانه تر آن است که به جای روش های پر هزینه پیشگیری، اجتناب و آشکار سازی کلا از این مشکل چشم پوشی کنیم.

ترکیب روش ها در اداره بن بست

در عمل هر یک از روش های اداره بن بست (پیشگیری، اجتناب، کشف) به تنهایی برای تمام انواع منابع مناسب نمی باشد، یک شیوه ترکیبی برای دسته های مختلف منابع مثلا می تواند به صورت زیر باشد.

- منابع داخلی سیستم مثل بلوک کنترل پردازش؛ پیشگیری از طریق ترتیب منابع
- منابع کار (گرداننده های دیسک، نوار، چاپگر و...) : اجتناب از بن بست، چون حداکثر نیاز از قبل مشخص است
- حافظه اصلی؛ پیش گیری از طریق پس دادن می تواند انجام پذیرد، چرا که به راحتی می توان حافظه اصلی را از پردازش ها پس گرفت؛ زیرا به محض کمبود حافظه یکسری از پردازش ها به حافظه جانبی منتقل می شوند.
- حافظه جابه جاپذیر (پشتیبان)؛ تفصیص از پیش، میتواند انجام پذیرد چرا که حداکثر نیاز های ذخیره سازی از قبل می تواند مشخص باشد.
- منابع پردازش؛ از طریق اجتناب

پایان جلسه هشتم

جلسه نهم

مدیریت حافظه (قطعه بندی و صفحه بندی):

□ یکی از مولفه های سیستم عامل مدیریت حافظه است.

وظایف مدیر حافظه:

- اداره کردن سلسله مراتب حافظه
- جلوگیری از تداخل برنامه های موجود در حافظه (به خصوص در محیط های چند برنامه گی)
- مدیریت حافظه مجازی

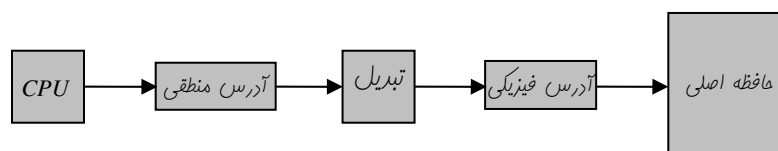
آدرس منطقی: آدرسی است که برنامه نویس در برنامه صادر میکند یا آدرسی که توسط CPU تولید میشود. مثلا آدرس منطقی 100 در که

`Mov Al, [100]`

آدرس فیزیکی: آدرس مشاهده شده توسط وایر حافظه (یعنی آنچه که در رجیستر آدرس حافظه بار میشود) را آدرس فیزیکی می نامند. آدرس مجازی: در حالتی که پیوند آدرس های حافظه در زمان اجرا باشد، به آدرس منطقی، آدرس مجازی (Virtual Addresss) میگویند.

تبدیل آدرس منطقی به آدرس فیزیکی:

به عمل تبدیل آدرس منطقی به آدرس فیزیکی نگاشت آدرس (mapping) گویند.



انواع انقیاد آدرس:

- 1- **زمان کامپایل:** اگر در موقع کامپایل معلوم باشد که برنامه در کجای حافظه قرار خواهد گرفت، در این صورت که مطلق میتواند تولید شود، یعنی آدرس های ذکر شده در برنامه هنگام بارشدن و یا هنگام اجرا تغییر نخواهد کرد و تصویر آینه وار برنامه در دیسک عینا به به حافظه آورده شده و اجرا می گردد. مثلا آدرس 100 ذکر شده در برنامه همان آدرس 100 مطلق حافظه RAM می باشد. برنامه های com تحت سیستم عامل DOS اینگونه هستند.
- 2- **زمان بار کردن:** اگر در زمان کامپایل معلوم نباشد که برنامه در کجای حافظه قرار خواهد گرفت، آنگاه کامپایلر بایستی که قابل جابه جایی (Relocatable) تولید کند. به عنوان مثال اگر در برنامه ای دستور `Mov Al [100]` را داشته باشیم و برنامه حاوی این دستور در زمان بار کردن در محل 300 حافظه قرار گیرد (شروع آدرس 300 باشد) در این صورت آدرس فیزیکی معادل آدرس 100 برابر 400 میشود.
- 3- **زمان اجرا:** اگر پردازش در حین زمان اجراش بتواند در حافظه جابه جا شود، آنگاه پیوند دادن بایستی تا زمان اجرا به تأخیر انداخته شود. برای این حالت نیاز به سخت افزار خاصی وجود دارد.

□ هم برای نگاشت زمان کامپایل و هم برای زمان نگاشت بار کردن مدیر حافظه ملزم به پشتیبانی سخت افزاری نیست ولی برای نگاشت زمان اجرا نیاز به پشتیبانی سخت افزاری است.

روش های مدیریت حافظه:

1- **تک برنامه گوی ساره:**

در این روش در هر لحظه فقط یک برنامه در حافظه اصلی قرار دارد، و برنامه ای که می بایست اجرا شود، نباید اندازه اش از حافظه اصلی بیشتر باشد اگر حافظه RAM به اندازه کافی در دسترس نباشد، برنامه اجرا نمی شود. سیستم عامل DOS اولیه این گونه بوده است.

| |
|------------------|
| برنامه های ROM |
| برنامه های کاربر |
| سیستم عامل |

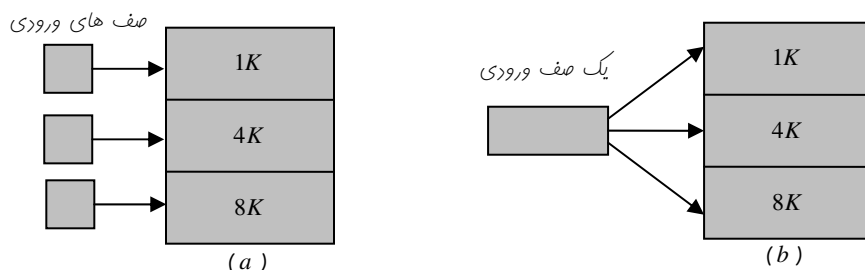
طرح حافظه در این سیستم عامل اولیه به این شکل بوه است

2- یک برنامه گنی با سیستم (overlay)

در این سیستم مدیریت، پردازش می تواند بزرگتر از حافظه اصلی باشد. در شیوه overlay (بایگلاشت) برنامه به بخش های مختلفی تقسیم شده و تنها آن داده ها و دستورالعمل هائی را در حافظه قرار می دهیم که در هر زمان مفروض مورد نیاز هستند و بقیه بخش ها در دیسک باقی می مانند. هنگامی که به بخش دیگری از آن برنامه نیاز داریم، قسمتی که مورد نیاز نیست از حافظه خارج شده و بخش مورد نیاز به حافظه آورده می شود. □ overlay به حمایت سفت افزاری ویژه ای نیاز ندارد. این تکنیک را خود برنامه نویس می بایست در برنامه پیاده سازی کند.

3- چند برنامه گنی با بخش بندی ثابت حافظه:

ساده ترین روش چند برنامه گنی این است که حافظه را به N قسمت تقسیم کنیم، اندازه هر قسمت می تواند با بخش های دیگر متفاوت باشد. این کار می تواند در هنگام شروع کار سیستم توسط سیستم عامل یا به صورت دستی توسط اپراتور انجام شود. وقتی یک کار وارد می شود در یک صف ورودی قرار می گیرد تا در کوچکترین بخش که مناسب آن است قرار داده شود. البته ممکن است آن بخش دقیقاً هم اندازه برنامه نبوده و بدین ترتیب مقداری از فضای آن از بین برود. در این روش می توان برای هر پارتیشن از یک صف مجزا استفاده کرد (شکل a) و یا اینکه فقط یک صف برای تمام پارتیشن ها داشت (شکل b).



4- چند برنامه گنی با جابه جایی (Swapping) - مبارله

در این روش در هر لحظه میتوان چندین پردازه داشت، ولی اگر پردازه ای می خواهد به حافظه اصلی بیاید، و حافظه خالی نباشد، به جای یکی از پردازه های موجود در حافظه اصلی قرار می گیرد، و پردازه ای که در حافظه اصلی بود به دیسک منتقل میشود. اشکالی که در این روش وجود دارد این است که به دلیل نقل انتقال پردازه مابین حافظه اصلی و دیسک زمانی طول می کشد.

$$CPU \text{ کارایی} = \frac{\text{زمان اجرای پردازه}}{\text{زمان جابه جایی} + \text{زمان اجرای کل زمان طی شده}}$$

5- چند برنامه گنی به صورت تفصیص همجواری:

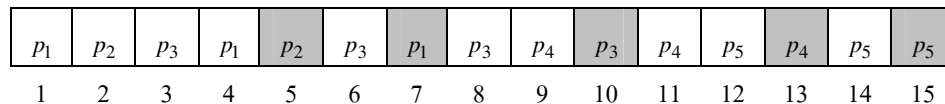
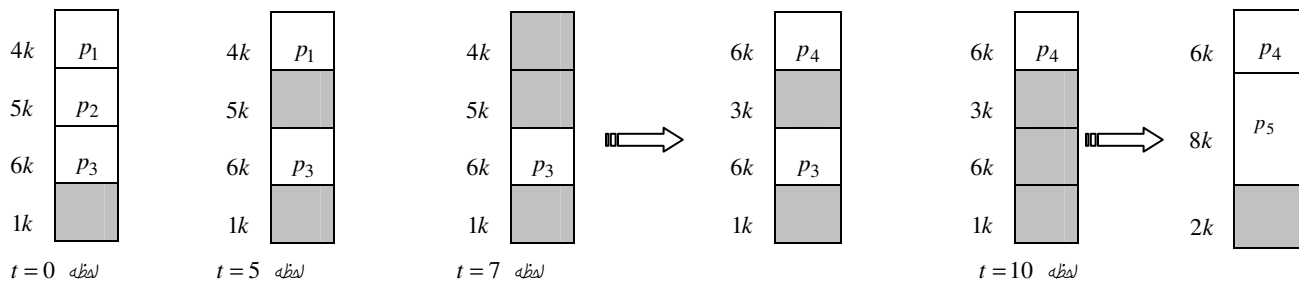
در این روش حافظه از قبل به اندازه های ثابت تقسیم نمی شود بلکه پردازه هائی که می بایست به حافظه آیند مطابق الگوریتم هائی یکی از فضا های آزاد حافظه را پیدا کرده و به طور کامل در آن قسمت قرار می گیرد.

مثال. فرض کنید اندازه حافظه ای 16k باشد اگر کار های زیر با توجه به الگوریتم FCFS توسط زمان بند بلند مدت جهت تبدیل شدن به پردازه ها انتاب شوند و در عین اجرای پردازه ها از زمانبندی RR با کوانتوم زمانی 1 استفاده شود. وضعیت حافظه را بعد از هر Load شدن برنامه جدید به حافظه مشفص نموده و زمان Load شدن پردازه ها را معین کنید (پردازه وارد شده به انتهای صف می رود).

| زمان اجرا | مقدار حافظه مورد نیاز | job |
|-----------|-----------------------|-------|
| 3 | 4k | j_1 |
| 2 | 5k | j_2 |
| 4 | 6k | j_3 |
| 3 | 6k | j_4 |
| 3 | 8k | j_5 |

ابتدا برنامه های p_1, p_2, p_3 در حافظه بار شده و شروع به اجرا میشوند. توجه کنید به علت نبود حافظه کافی

پردازه های p_4, p_5 در حافظه بار نمی شوند.



پارگی:

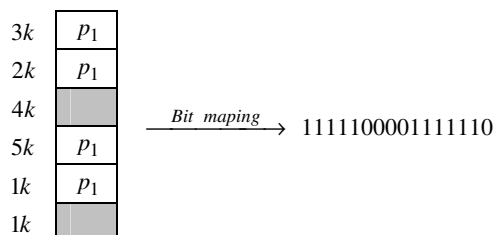
به صفه های حافظه که ظرفیت مجموع آنها زیاده ولی چون از هم دیگر فاصله دارند قابل استفاده نیستند را پارگی گوئیم که یک راه برای از بین بردن آنها فشرده سازی می باشد (کاری می کنیم که فضا های آزاد در کنار هم قرار گیرند یعنی برنامه ها را جابه جا کنیم)، فشرده سازی عملی زمانبر هست و از طرفی کدهائی را می توان جابه جا کرد که جابه جایی (Relocatable) باشند.

□ اگر زمانی پردازش ای مانند 4z بخواهد وارد حافظه اصلی شود یک فضای $7k$ خالی در ابتدای حافظه و یک فضای خالی $6.5k$ در محل دیگر حافظه باشد، در کدام محل قرار گیرد؟ جواب دادن به سؤال هائی از این قبیل به بحث الگوریتم های انتخاب بستگی دارد.

روش های مدیریت فضای آزاد

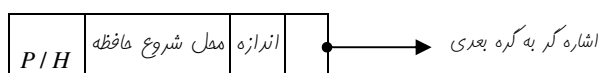
الف. روش نگاشت بیتی (Bit maps)

در این روش یک واحد مناسب در نظر گرفته می شود، و متناظر با این واحد یک بیت در نظر گرفته می شود، بنابراین با شروع از ابتدای حافظه هر واحدی از حافظه که پر باشد بیت متناظر آن 1 و اگر خالی باشد بیت متناظر آن صفر در نظر گرفته می شود. مشکل این روش این است که اگر پردازش به k واحد نیاز داشته باشد، می بایست در رشته بیتی حافظه به دنبال k صفر کنار هم باشیم. به عنوان مثال اگر واحد انتخابی $1k$ باشد. شکل زیر این موضوع را روشن می کند.



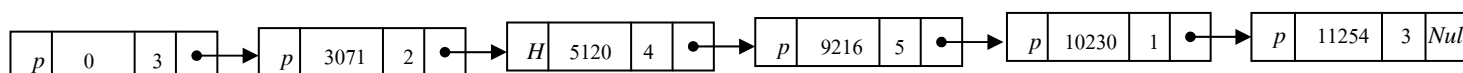
ب. روش لیست پیوندی (Linked list)

در روش لیست پیوندی با شروع از ابتدای حافظه به ازای هر پردازش یا فضای خالی یک گره ای با ساختار زیر در نظر گرفته می شود.



P : پردازش
 H : فضای آزاد

مثال قبلی به روش لیست پیوندی



الگوریتم های انتخاب ها:

1- اولین مناسب (frist fit)

وقتی پردازش می خواهد به حافظه load شود ابتدای لیست فضای آزاد را نگاه کرده و اولین فضای آزادی که اندازه اش بزرگتر یا مساوی اندازه پردازش باشد انتخاب شده و پردازش در آن محل قرار می گیرد.

□ مشکل تراکم پردازش ها در ابتدای حافظه است که روش بعدی سعی می کند این مشکل را برطرف کند.

2- مناسب بعدی (Next fit)

این روش مانند frist fit است با این تفاوت که جستجو از محلی در لیست آغاز می شود که آخرین بار تفصیص از آن محل صورت گرفته است. بدین ترتیب یکنواختی توزیع برنامه ها در سطح حافظه نسبت به روش قبلی بیشتر خواهد شد.

3- بهترین مناسب (Best fit)

در این روش کل لیست فضای آزاد جستجو شده و کوچکترین مغره که به اندازه کافی بزرگ است به پردازش تفصیص داد می شود. این روش باعث می شود که کوچکترین مغره بر اثر تفصیص باقی بماند. با این روش فضا های بزرگتر برای تقاضا های بیشتر حفظ می شوند. از آنجا که تمام لیست بلاک های آزاد باید بررسی شود، این تکنیک قدری زمانبر است.

پایان جلسه نهم

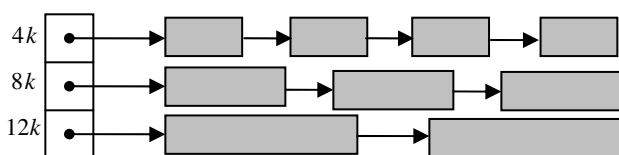
جلسه دهم:

4- بدترین مناسب (worst-fit):

در این الگوریتم بزرگترین مغره انتقاب شده و پردازش در آن قرار میگیرد. دلیل انتقاب بزرگترین مغره این است که از فضای باقی مانده دیگر پردازش ها می توانند استفاده کنند. ایراد این تکنیک این است که امکان دارد، تقاضاهای که نامیه بزرگی می خواهند، دیگر نتوانند برآورده شوند چرا که بلاک های بزرگ زودتر تفصیص یافته و کوچک می شوند.

5- سریعترین مناسب (Quick Fit):

در این الگوریتم لیستی از اندازه پردازش های متداول تهیه می شود و آرایه ای با n خانه در نظر گرفته می شود که هر خانه این آرایه شامل یک اشاره گر به ابتدای لیست یک فضای خالی به اندازه متداول است به عنوان مثال فضای های متداول می توانند $2k$, $4k$, $8k$, $12k$ و ... باشند که برای هر کدام یک خانه آرایه در نظر گرفته می شود. عیب این روش این است که اگر پروسسی فائمه یابدر باید فضای آزاد شده آن به لیست مناسب اضافه شود که این کار زمانبر می باشد.



6- الگوریتم رفاقتی (Buddy):

در این روش مغره ها (فضا های خالی) به صورت توان های 2 در نظر گرفته می شود. به عنوان مثال مغره هایی به اندازه $1k$, $2k$, $4k$, $8k$, $16k$, $32k$ و ... برای هر گروه یک لیست جداگانه در نظر گرفته می شود. بدین ترتیب جهت تفصیص یک بلاک تنها باید بلاک مورد نظر را از لیست مناسب خارج کرد. پس از تفصیص اگر فضای باقی مانده آن بلاک، توانی از 2 باشد در لیست مربوطه اش قرار می گیرد و در غیر این صورت به پندین بفش که اندازه هر کدام توانی از 2 می باشد تقسیم میشود. از طرف دیگر در این روش بلوک های کنار هم می توانند باهم ترکیب شده و بفش بزرگتری را پدید یآورند.

مثال. با قسمت هایی از حافظه به اندازه های $100k$, $200k$, $300k$, $426k$, $417k$, $112k$, $183k$ را چگونه در حافظه قرار می دهند و کدام روش از حافظه به طور بهینه استفاده می کند. ترتیب ورود پردازش ها را یک بار از راست به چپ و یک بار از چپ به راست بگیریم.

H : 100 k , 500 k , 200 k , 300 , 600 k
P : 212 k , 417 , 112 k , 426 k

لیست فضای آزاد، ترتیب پردازش ها از چپ به راست
لیست فضای آزاد، ترتیب پردازش ها از راست به چپ

الف. اولین مناسب: در حالت از چپ به راست

پردازش 426 باید منتظر بماند

لیست فضای آزاد، ترتیب پردازش ها از چپ به راست
لیست فضای آزاد، ترتیب پردازش ها از راست به چپ

ب. بهترین مناسب

لیست فضای آزاد، ترتیب پردازش ها از چپ به راست
لیست فضای آزاد، ترتیب پردازش ها از راست به چپ

ب. بدترین مناسب

روش بهترین مناسب از حافظه به صورت بهینه استفاده می کند. زیرا برای تمام پردازش ها فضای لازم را پیدا می کند و پارگی خارجی در آن حداقل است. مثال. در زیر بلوک های خالی حافظه به ترتیب از چپ به راست نشان داده شده اند، اگر درخواست های جدیدی برای چهار بلوک به اندازه $20k$, $30k$, $20k$, $35k$ به ترتیب از راست به چپ ذکر شده داده شود و از روش Next Fit استفاده شود و تفصیص از اول حافظه شروع شود، وضعیت حافظه را بعد از این تفصیص ها مشخص کنید.

شروع → 40k , 25k , 45k , 50k , 60k , 40k

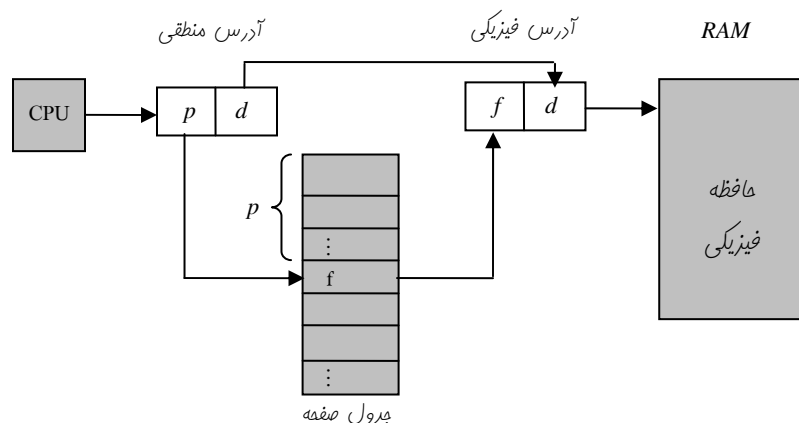
پواب: 20k , 25k , 15k , 30k , 25k , 40k

صفحه بندی (Paging):

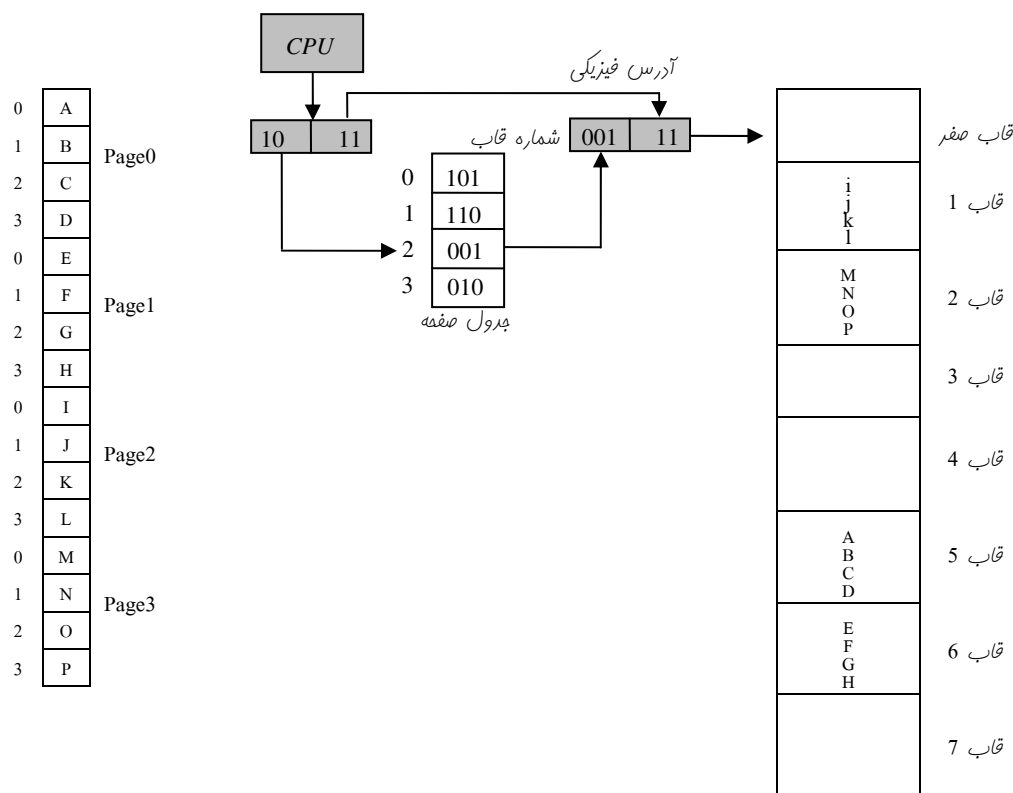
در روش صفحه بندی برنامه کاربر (فضای آدرس منطقی) به بخش هایی به اندازه ثابت به نام صفحه (Page) تقسیم می شود و حافظه فیزیکی به قسمت هایی به نام قاب (Frame) تقسیم می شود که اندازه هر فریم برابر اندازه Page می باشد. صفحه بندی این امکان را می دهد که قسمت های یک برنامه (Page های هر برنامه) در حافظه پراکنده باشند (لازم نیست مجاور هم باشند)

نمونه تبدیل آدرس منطقی به آدرس فیزیکی:

در این تکنیک هر آدرس تولید شده توسط CPU (یعنی آدرس منطقی) از دو بخش شماره صفحه (p) و افسست صفحه (d) تشکیل شده است. شماره صفحه به عنوان اندیس جدول صفحه (page table) استفاده می گردد. جدول صفحه شامل آدرس مبانی هر صفحه در آدرس فیزیکی RAM است. این آدرس مبنا با آدرس افسست منطقی ترکیب شده و آدرس فیزیکی نهائی را تشکیل می دهد. شکل زیر این موضوع را نشان می دهد.



مثال. فرض کنید تعداد صفحات $2^2 = 4$ عدد و تعداد قاب ها $2^3 = 8$ عدد باشد.



□ به هر کدام از سطرهای جدول صفحه یک مدخل یا Entry گویند. که هر مدخل شامل شماره یک فریم و اطلاعات دیگری از قبیل بیت معتبر (نامعتبر)، بیت Read، بیت write و بیت های دیگر می باشد.

$$\left[\begin{array}{c} \text{تعداد} \\ \log_2 \text{صفحات} \end{array} \right] = \text{تعداد بیت های مورد نیاز برای شماره جدول صفحه } (p)$$

حافظه منطقی \geq حافظه فیزیکی \Rightarrow تعداد page ها \geq تعداد فریم

$$\left[\begin{array}{c} \text{اندازه} \\ \log_2 \text{صفحات} \end{array} \right] = \text{تعداد بیت های مورد نیاز برای افست } (d)$$

□ در صفحه بندی اگر بیت valid صفر باشد بدین معناست که آدرس صفحه ذکر شده نامعتبر می باشد یا چنین صفحه ای وجود ندارد.

جلسه یازدهم و دوازدهم

مثال. اگر اندازه هر page ، 1k باشد تعداد خطوط آدرس منطقی را با مشخص کردن p و d و همینطور تعداد خطوط آدرس فیزیکی را با مشخص کردن f و d برست آورید. آدرس منطقی و آدرس فیزیکی صفحه شماره 1 و فانه شماره 20 را مشخص کنید.

| Page table | | RAM | |
|------------|-----|-----|-------|
| 00 | 000 | 000 | Page0 |
| 01 | 111 | 001 | |
| 10 | 011 | 010 | |
| 11 | 100 | 011 | Page2 |
| | | 100 | Page3 |
| | | 101 | |
| | | 110 | |
| | | 111 | Page1 |

حل.

| | | | |
|--------|-------|--------|-------|
| p | d | F | d |
| 10 بیت | 2 بیت | 10 بیت | 3 بیت |

$$d = \log_2^{page\ size} = \log_2^{1024} = 12$$

با توجه به شکل چهار صفحه داریم که دو بیت برای آدرس دهی آن کافی است پس برای آدرس دهی منطقی 12 بیت لازم است و

با توجه به شکل هشت قاب داریم که سه بیت برای آدرس دهی آن کافی است پس برای آدرس دهی فیزیکی 13 بیت لازم است

| | |
|-----|------------|
| 01 | 0000010100 |
| 111 | 0000010100 |

آدرس منطقی صفحه شماره 1 و فانه شماره 20

آدرس فیزیکی صفحه شماره 1 و فانه شماره 20

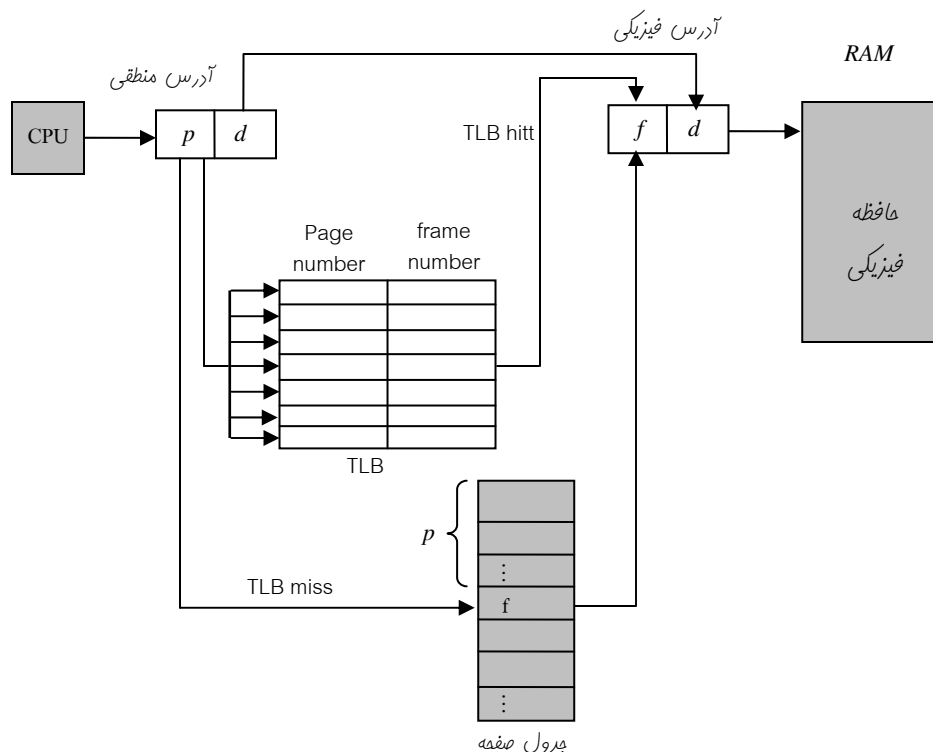
چون جدول صفحه در حافظه اصلی نگهداری میشود بنابراین هر مراجعه به حافظه تبدیل به دو مراجعه می شود. یک مراجعه به جدول صفحه جهت برست آوردن شماره frame و مراجعه دیگر جهت برست آوردن داده اصلی پس $t_{acc} = 2t_m$ که t_m زمان یک مراجعه می باشد. چون این روش زمانبر است معمولا جهت نگهداری جدول صفحه از بافر های دم دستی استفاده می شود (TLB) که از نوع حافظه شرکت پذیر می باشند. که به طور موازی می توان در آنها عمل جستجو را انجام داد.

روش های استفاده از TLB جهت نگهداری جدول صفحه

1- نگهداری کامل جدول صفحه در TLB

اشکال: اندازه جدول صفحه خیلی بزرگتر از TLB می باشد (اندازه TLB محدود می باشد زیرا گران است)

2- از جدول صفحه که در حافظه اصلی است به همراه TLB استفاده می شود که در این صورت جهت برست آوردن شماره frame متناظر با یک شماره صفحه ابتدا عمل جستجو در TLB انجام میشود. اگر شماره صفحه مورد نظر در TLB یافت نشود به جدول صفحه در حافظه اصلی مراجعه میشود و بعد از این عمل TLB بروز (update) می شود. شکل صفحه بعد این موضوع را نشان میدهد.



مثال.

اگر جستجوی TLB به اندازه 20 نانو ثانیه طول بکشد و دستیابی به حافظه اصلی نیز 100 نانو ثانیه زمان بخواهد، آنگاه زمان دستیابی به حافظه را موقعی که شماره قاب در TLB پیدا شود و هنگامی که در TLB پیدا نشود را بدست آورید.

حل: زمان دستیابی اگر در TLB پیدا شود. $20 + 100 = 120 \text{ ns}$

زمان دستیابی اگر در TLB پیدا نشود. $20 + 100 + 100 = 220 \text{ ns}$

تذکر: جدول TLB در واحد مدیریت حافظه یا MMU (memory Management unit) قرار دارد، خود MMU نیز در CPU قرار دارد.

h: احتمال وجود شماره صفحه در TLB

عدم وجود (عدم موفقیت): $1-h$

زمان دستیابی به حافظه اصلی: t_m

$$t_{acc} = h t_{TLB} + (1-h)(t_{TLB} + t_m) + t_m$$

مثال. یک سیستم صفحه بندی را در نظر بگیرید، جدول صفحه در حافظه اصلی است اگر زمان دستیابی به حافظه اصلی برابر 60 نانو ثانیه باشد و احتمال وجود شماره صفحه در TLB، 0.75 باشد و زمان دستیابی به TLB، 5 نانو ثانیه باشد، نسبت بهبود آدرس بر اثر TLB، در مقایسه با هنگامی که از TLB استفاده نمی شود چقدر است؟

$$h=0.75$$

$$1-h=0.25$$

$$t_m=60 \text{ ns}$$

$$t_{TLB}=5 \text{ ns}$$

زمان دستیابی به داده اصلی بدون استفاده از TLB $t_{acc} = 2t_m = 120 \text{ ns}$

زمان دستیابی به داده اصلی با استفاده از TLB $t_{acc} = 0.75(5) + (0.25)(5 + 60) + 60 = 80 \text{ ns}$

$$\text{نسبت بهبود آدرس} = \frac{120}{80}$$

مثال. در یک سیستم حافظه صفحه بندی با یک جدول صفحه حاوی 64 مدخل 11 بیتی (شامل یک بیت اعتبار/عدم اعتبار) و صفحه هایی به اندازه هر یک 512 بایت یک آدرس منطقی و یک آدرس فیزیکی چند بیت است؟

حل. چون جدول صفحه 64 مدخل و هر صفحه 512 بایت است بنابراین $2^{15} = 2^6 \times 512 = 64 \times 512$ = اندازه حافظه منطقی

پس آدرس منطقی 15 بیتی است، از طرفی چون هر مدخل جدول صفحه برای آدرس دهی 10 بیتی است (یک بیت برای عملیات کنترلی است) و هر آدرس موجود در هر مدخل (سطر) جدول صفحه به یک page با اندازه 512 بایت اشاره می کند پس $2^{19} = 2^{10} \times 512$ = اندازه حافظه فیزیکی لذا آدرس فیزیکی 19 بیتی است.

PTBR: همواره حاوی آدرس شروع جدول صفحه پردازنده در حال اجراست، هنگام *context swiching*، *PTBR* برابر آدرس شروع جدول صفحه پردازنده جدید میشود.

بعضی از مدیریت حافظه بایستی به صورت سفت افزاری باشد (پشتیبانی شود)

مثال. فرض کنید آدرس منطقی 32 بیتی و اندازه هر صفحه (قالب) نیز 1kB باشد در این صورت مطلوب است تعیین

الف. تعداد بیت های p و d

ب. اندازه جدول صفحه در صورتی که هر مدخل (Entitiy) جدول صفحه 8 بایت باشد.

حل. با توجه به این که اندازه هر صفحه یا قالب 1kB می باشد پس ($1k = 2^{10}$) 10 بیت برای قسمت آفست (d) لازم است. و با توجه به این که آدرس منطقی 32 بیتی می باشد، پس برای قسمت p، 22 بیت باقی می ماند.

| | |
|---|---|
| p | d |
|---|---|

10 بیت 22 بیت

برای حل قسمت ب ابتدا بایستی تعداد مدخل های جدول صفحه را بدست آوریم

$$2^{22} = 4M = \text{تعداد مدخل های جدول صفحه}$$

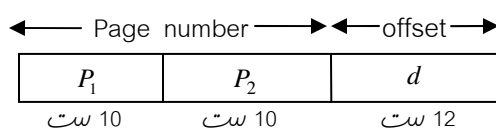
$$32M \text{ byte} = 4M \times 8 = \text{اندازه جدول صفحه}$$

□ چون اندازه جدول صفحه خیلی بزرگ می شود از روش صفحه بندی چند سطحی استفاده می شود به عبارتی برای جدول صفحه نیز جدول صفحه ایجاد می کنیم.

صفحه بندی چند سطحی

اغلب کامپیوترهای امروزی دارای فضای آدرس دهی منطقی بسیار بزرگی هستند (2^{32} تا 2^{64} فانه آدرس) در چنین سیستم هایی خود جدول بسیار بزرگ خواهد بود، برای رفع این مشکل می توان از تکنیک صفحه بندی دو سطحی استفاده کرد به گونه ای که در آن جدول صفحه، خود صفحه بندی شده باشد در واقع در این روش جدول صفحه به قطعات کوچک تقسیم شده و دیگر لازم نیست تمامی جدول صفحه در RAM نگهداری شوند به عبارت دیگر جدولی که به آنها فعلا نیاز نداریم به حافظه آورده نمی شوند

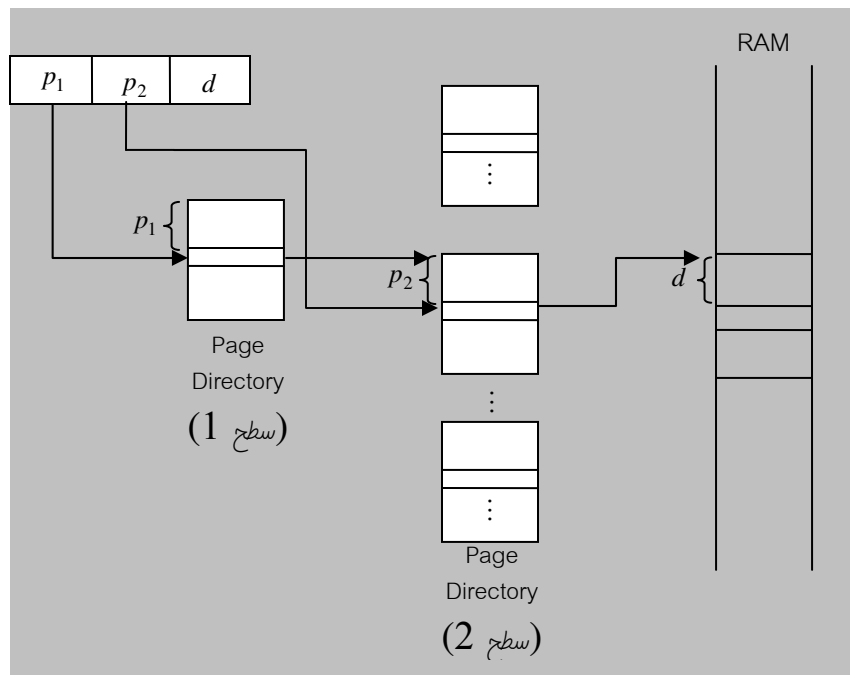
مثلا مدل آدرس دهی در اکثر پردازنده های 32 بیتی به صورت زیر است



چون آفست 12 بیتی است پس اندازه هر صفحه $4k = 2^{12}$ بوده و در اکثر

2^{20} صفحه وجود خواهد داشت ($2^{20} = 2^{12} \div 2^{32}$). هر مدخل p_1 آدرس شروع یک جدول صفحه را در سطح 2 می دهد و هر مدخل p_2 آدرس

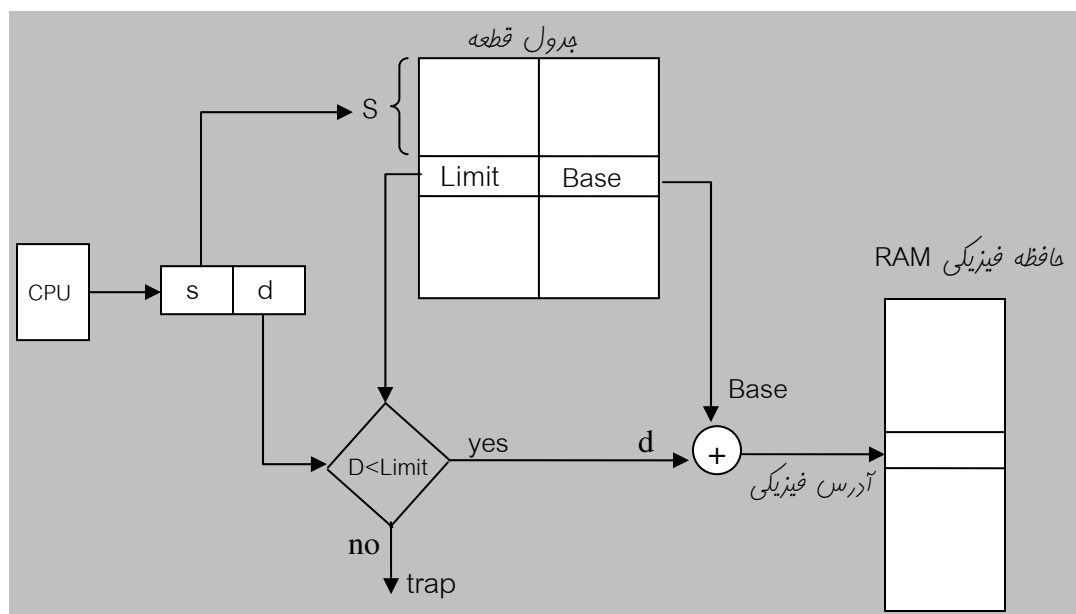
شروع یک page (یک frame) را می دهد و بیت های d یک آدرس را در داخل page (یا frame) مشخص می کند. شکل صفحه بعد نمونه این آدرس دهی را نشان می دهد.



برین ترتیب هر پردازش یک *page Directory* با $(2^{10} = 1k)$ هزار ورودی دارد که هر ورودی آن می تواند به یک *page Table* اشاره کند که آن نیز $(2^{10} = 1k)$ هزار ورودی دیگر دارد. برای هر پردازش همواره *page Directory* در حافظه قرار می گیرد ولی *page Table* های سطح 2، به تعداد لازم در حافظه قرار می گیرند.

قطعه بندی (segmentation)

حافظه اصلی یا فیزیکی به صورت یک آرایه خطی از بایت ها می باشد به عبارتی هر خانه در حافظه یک آدرس دارد، در روش صفحه بندی پردازش ها بر اساس محدودیت فیزیکی یعنی اندازه هر صفحه تقسیم می شوند ولی در قطعه بندی خود کاربر برنامه نویس می تواند برنامه اش را به صورت منطقی تقسیم کند که به هر یک از این قسمت ها یک *segment* گویند هر قطعه یک آدرس شروع و یک طول دارد، آدرسی که کاربر در سطح منطقی می دهد شامل دو جزء است. یکی شماره قطعه و دیگری فاصله (آفست) خانه مورد نظر از اول آن قطعه است، آدرس دو بعدی استفاده شده توسط کاربر در سطح منطقی می بایست توسط یک نگاشت به آدرس یک بعدی فیزیکی تبدیل شود. این نگاشت به وسیله جدول قطعه انجام می پذیرد، جدول قطعه از دو ستون اصلی تشکیل شده است، یکی آدرس پایه قطعه (Base) و دیگری طول یا حد قطعه (Limit). قسمت Base حاوی آدرس فیزیکی در RAM می باشد که قطعه از آنجا شروع می شود، شکل زیر نحوه تبدیل آدرس منطقی را به آدرس فیزیکی در این سیستم نشان میدهد.



آدرس منطقی از دو جزء شماره قطعه (s) و آفست درون آن قطعه (d) تشکیل یافته است.

□ یکی از مزایای قطعه بندی اشتراک است.

□ در صفحه بندی اندازه صفحات برابر است ولی در قطعه بندی لزومی ندارد که اندازه صفحات یکی باشد

مثال. فرض کنید در یک شبکه کامپیوتری چندین کاربر به صورت همزمان نیاز به اجرای برنامه word دارند به جای این که هر برنامه word که برای همه مشترک است برای هر کاربر به صورت مجزا در حافظه load شود، فقط یک بار در حافظه load شده و تمامی کاربران به صورت اشتراکی از آن استفاده می کنند و هر کاربر قطعه خاص خود را دارد.

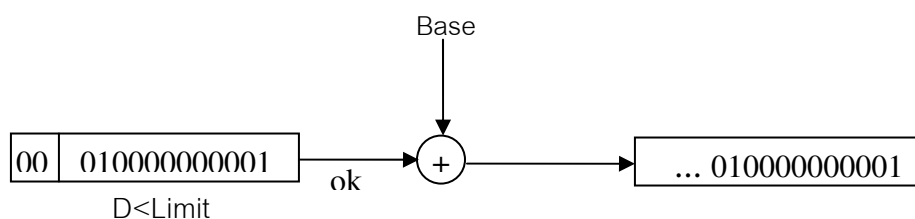
مثال. فرض کنید پردازش ای دارای سه قطعه می باشد به طوریکه قطعه اول 2k، قطعه دوم 3k و قطعه سوم 4k می باشد مطلوب است الف. تعیین ورودیهای جدول قطعه

| | base | Limit | V |
|----|------|-------------------|---|
| 00 | 0000 | 2×2^{10} | 1 |
| 01 | 3000 | 3×2^{10} | 1 |
| 10 | 2000 | 4×2^{10} | 1 |
| 11 | | | 0 |

| | RAM |
|------|----------|
| 0000 | قطعه ی 0 |
| 2000 | قطعه ی 2 |
| 3000 | قطعه ی 1 |
| | |

| S | d |
|---|----|
| | |
| 2 | 12 |

ب. اگر برنامه نویس در قطعه صفر دستور push با آدرس 1025 را داشته باشد آدرس فیزیکی متناظر با این آدرس را بیابید.



نکته: اگر برنامه نویس دستور pop فانه 2048 را در قطعه ی صفر بدهد چون آدرس صادر شده در قطعه صفر وجود ندارد بنابر این وقفه رخ میدهد

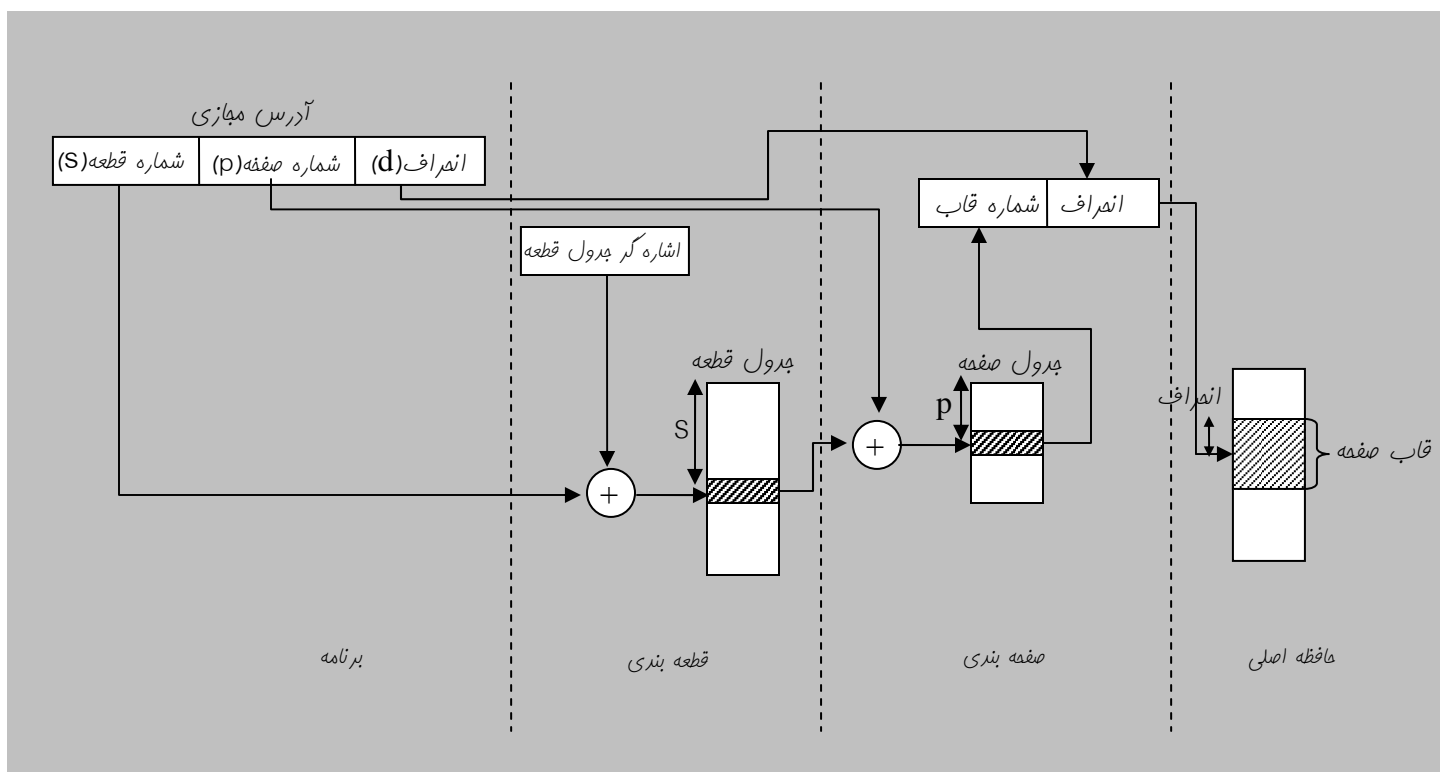
ج. اگر برنامه نویس دستور `Mov AX, [100]` را صادر کند، آدرس موثر مطلق (فیزیکی) را بیابید

ترکیب قطعه بندی و صفحه بندی

هم صفحه بندی و هم قطعه بندی نقاط قوت مخصوص به خود را دارند برای ترکیب نقاط قوت هر دو، این دو روش را ترکیب کرده و با هم استفاده می کنیم، در یک سیستم ترکیبی صفحه بندی / قطعه بندی فضای آدرس کاربر تحت نظر برنامه ساز به تعدادی قطعه تقسیم میشود، هر قطعه به نوبه خود به تعدادی صفحه به اندازه ثابت تقسیم می گردد. به ازای هر فرایند یک جدول قطعه و به ازای هر قطعه یک جدول صفحه ایجاد می گردد، تعداد درایه های هر جدول صفحه بستگی به اندازه قطعه مربوطه دارد. همانند صفحه بندی آخرین صفحه هر قطعه معمولاً پر نمی شود و به ازای هر قطعه به طور میانگین به اندازه نصف صفحه تکه تکه شدن داخلی داریم. از دید برنامه ساز، آدرس

منطقی همپنان شامل شماره قطعه و انحراف در قطعه است. از دید سیستم این انحراف در قطعه، به صورت یک شماره صفحه و انحراف در صفحه دیده می شود.

هنگامی که فرایندی در حال اجراست یک ثابت آدرس شروع جدول قطعه آن فرایند را نگه می دارد. پردازنده از شماره قطعه ای که در آدرس منطقی است به عنوان شاخص به جدول قطعه استفاده کرده و جدول صفحه را برای قطعه مذکور پیدا می کند، سپس بخش شماره صفحه آدرس منطقی به عنوان شاخص جدول صفحه به کار رفته و شماره قاب مربوطه بدست می آید. این شماره قاب با بخش انحراف آدرس منطقی ترکیب شده و آدرس فیزیکی مورد نظر نتیجه میشود. شکل زیر نحوه ترجمه آدرس در یک سیستم قطعه بندی/صفحه بندی را نشان می دهد



برای مثال سیستم با آدرس منطقی 34 بیتی را در نظر بگیرید که در آن شماره قطعه 18 بیتی و انحراف 16 بیتی است، به کارگیری مدیریت حافظه قطعه بندی در این سیستم به خاطر اندازه بزرگ قطعه که می تواند تا 2^{16} باشد مشکلاتی را که در فوق متذکر شدیم در بر می گیرد. بنابراین سیستم از مدیریت حافظه قطعه بندی/صفحه بندی استفاده می نماید. نگرش این مدیریت در این سیستم بدین گونه است که انحراف قطعه به دو بخش 6 بیتی شماره صفحه و 10 بیتی انحراف صفحه تجزیه می شود. جدول صفحه برای هر قطعه می تواند حداکثر از 2^6 درایه برخوردار باشد و هر فرایند حداکثر می تواند تا 2^{18} قطعه را دربرگیرد.

| | | |
|-------------------------|--------------------|---------------------|
| انحراف از صفحه (10 بیت) | شماره صفحه (6 بیت) | شماره قطعه (18 بیت) |
|-------------------------|--------------------|---------------------|

مهمترین مزیت صفحه بندی:

پارگی فاربی ندارد

در قطعه بندی پارگی فاربی داریم زیرا اندازه قطعات مساوی نیست و قطعه ای بلااستفاده ای وجود دارد که پراکنده بودن و همجواری نیستند بنابراین نمی توان از آنها برای یک قطعه استفاده کرد.

مهمترین مزیت قطعه بندی

اشتراک

اشتراک در صفحه بندی نسبت به قطعه بندی ناپذیر است و یا اصلا وجود ندارد.

جدول زیر مقایسه ای بین روش های مدیریت حافظه را نشان می دهد.

| آیا کل برنامه در حافظه اصلی کنار هم قرار می گیرد | جهت اجرا کل برنامه در حافظه می باشد یا نه | چند برنامه | |
|--|---|------------|--------------------------------|
| کنار هم | کل برنامه | یک | تک برنامه کی ساده |
| کنار هم | لزومی ندارد | یک | تک برنامه کی با overlay |
| کنار هم | کل برنامه | چند | چند برنامه کی با swapping |
| کنار هم | کل برنامه | چند | چند برنامه کی به صورت همجواری |
| کنار هم | کل برنامه | چند | چند برنامه کی به صورت بخش بندی |
| لزومی ندارد | کل برنامه | چند | صفحه بندی |
| لزومی ندارد | کل برنامه | چند | قطعه بندی |

جلسه دوازدهم و چهاردهم

حافظه مجازی:

تکنیکی است که اجازه می دهد بدون این که کل برنامه در حافظه اصلی قرار گیرد اجرا شود. **مزیت مهم** این روش آن است که اجازه می دهد برنامه بزرگتر از حافظه اصلی باشد. علاوه بر این در این روش کاربرد حافظه را به صورت آرایه ای فوق العاده بزرگ و یکنواخت می بیند. به عبارتی کاربرد (برنامه نویس) خود را درگیر حافظه فیزیکی نمی کند و فرض می کند که بی نهایت حافظه در اختیار دارد و برنامه نویسی ساده می شود. **مزیت دیگر** این است که چون فقط صفات مورد نیاز از دیسک به حافظه اصلی منتقل می شوند بنابراین زمان کمتری صرف عمل ۱/۰ می شود از این رو زمان پردازش نیز سریعتر می شود.

بهرت پیاده سازی حافظه مجازی دو روش وجود دارد.

- ترکیبی از صفحه بندی و swapping

- ترکیبی از قطعه بندی و swapping

به عبارتی حافظه مجازی به صورت صفحه بندی نیازی پیاده سازی می شود. در این روش مطابق اصل مملی بودن مراجعه به حافظه صفحه یا صفحاتی را که مورد نیاز باشند به حافظه اصلی آورده می شوند و هر وقت به صفحه ای نیاز باشد ولی در حافظه اصلی نباشد آن صفحه مطابق روشی که بعدا خواهیم گفت به حافظه اصلی Load می شود.

حافظه فیزیکی به یک سری قسمت های مساوی به نام Frame یا قاب تقسیم می شود. و برنامه کاربرد نیز به قسمت هایی به اندازه هر Frame به نام page یا صفحه تقسیم می شود.

□ اگر در ابتدا هیچ صفحه ای در حافظه اصلی نباشد و به ممض نیاز، به حافظه اصلی آورده شود به این روش **صفحه بندی نیازی ممض** گویند. در این روش هرگز صفحه ای تا وقتی که مورد نیاز نباشد وارد حافظه نمی شود.

کارایی صفحه بندی نیازی

اگر t_m زمان دستیابی به حافظه، p احتمال وقوع فضای صفحه (احتمال این که صفحه در حافظه اصلی نباشد) و t_f زمان لازم برای سرویس دهی به فضای صفحه باشد، آنگاه زمان دسترسی موثر (t_{eff}) از فرمول زیر مناسبه می شود.

$$t_{eff} = (1 - p)t_m + p \times t_f$$

در اکثر سیستم ها t_m بین 10 تا 200 نانو ثانیه می باشد.

انتساب قاب ها یا frame ها به پردازه ها

- **انتساب مساوی.** در این روش قاب های حافظه اصلی بین پردازه ها به صورت مساوی تقسیم می شود به عنوان مثال اگر حافظه اصلی دارای 12 قاب باشد و 5 پردازه بخواهند به طور هم زمان اجرا شوند به هر کدام از پردازه ها دو قاب تفصیص می یابد و دو قاب باقی مانده به عنوان قاب های بافر در نظر گرفته می شود. پس اگر n قاب داشته باشیم معمولا k تا از این n قاب را به عنوان بافر در نظر گرفته و $n-k$ قاب باقی مانده را مابین m پردازه تقسیم می کنیم.

- **انتساب متناسب.** در این روش تعدا قاب ها متناسب با اندازه پردازه ها به پردازه ها تفصیص می یابد به عنوان مثال اگر حافظه اصلی دارای 12 قاب باشد و دو قاب از آنها به عنوان بافر در نظر گرفته شود در صورتی که اندازه هر قاب $1k$ باشد و در

سیستم سه پدازه $5k$, $3k$, $2k$ به طور همزمان در حال اجرا باشند به پدازه $5k$, $5k$ ، به پدازه $3k$ ، $3k$ و به پدازه $2k$ ، $2k$ قابل در نظر گرفته می شود.

هیچ کدام از دو روش قبلی الویت را در نظر نمی گیرند می توان کاری کرد که به پدازه ها به نسبت الویتشان قاب تفصیل داد.

در هر مراجعه به حافظه اعمال زیر رخ می دهد

در روش مدیریت حافظه مبارزی به آدرس منطقی آدرس مبارزی می‌گویند اگر بیت ۷ یک باشد بدین معناست که page مورد نیاز در حافظه اصلی موجود است اگر صفر باشد دو برداشت از آن وجود دارد.

1- چنین page ی اصلا وجود ندارد یعنی دسترسی غیر مجاز است به عنوان مثال پرتال شما دارای سه page است ولی شما مراجعه به page چهارم را می دهید که این دسترسی غیر مجاز است.

2- چنین page ی وجود دارد ولی در حافظه اصلی نیست.

در روش حافظه میازی به طریق صفحه بندی نیازی، در هر مرحله به حافظه مراحل زیر دنبال می شود.

1- ابتدا به ساختار جدول صفحه مراجعه می شود اگر صفحه مورد درخواست در حافظه اصلی باشد شماره قلاب از ساختار جدول صفحه بدست آمده بنابراین آدرس فیزیکی آماده می شود.

2- اگر بیت $v=0$ باشد و صفحه مراجعه شده در پرده وجود نداشته باشد، دسترس غیر مفاز است و پایان پرده است

3- اگر بیت $v=0$ باشد ولی صفحه مورد نظر جزء پیردازه باشد در این صورت فضای فقدان صفحه رخ داده و بنابراین

الف. در حافظه اصلی قلاب خالی وجود داشته باشد، صفحه مورد نظر از رسانه جانبی به حافظه اصلی منتقل شده و جدول صفحه update می شود.

ب. در حافظه اصلی قاب فالی وجود ندارد از این رو یکی از صفات موجود در حافظه اصلی مطابق الگوریتم های جایگزینی به رسانه جانبی منتقل شده و صفحه مورد نیاز از رسانه جانبی به حافظه اصلی منتقل می شود.

برای این که انتقال صفه مورد نیاز از حافظه جانبی به حافظه اصلی منتظر ثالی شدن قاب حافظه اصلی نباشد، صفه مورد نیاز از حافظه جانبی به یکی از قاب های بافر منتقل شده و بنابراین عمل ورودی و خروجی سریعتر انجام میشود. سپس بعد از انتقال قاب شایسته ی خروج از حافظه اصلی به حافظه جانبی، قاب ثالی شده به لیست قاب های بافر اضافه می شود.

الگوریتم های جایگزینی صفت

مساله الگوریتم های جایگزینی صفحه یعنی هنگامی که نیاز است صفحه ای را از حافظه خارج کنیم کدام یک را انتخاب کنیم تا کارایی حداکثر شود. در زیر الگوریتم های متعددی را برای جایگزینی صفحه شرح می دهیم. ولی به طور کلی الگوریتمی بهتر است که تعداد نقض صفحه های (page fault rate) آن کمتر باشد بدیهی است که هر چه قدر تعداد فریم های آزاد بیشتر باشد، تعداد خطا های صفحه ای کاهش می یابد.

1- الگوریتم FIFO : ساده ترین روش جایگزینی صفحه است. در این روش صفحه ای برای خروج انتخاب می شود که زودتر از بقیه صفات وارد حافظه اصلی شده باشد

مثال. مراجعات (از چپ به راست) 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4 را با سه قاب آزاد در نظر بگیرید در روش جایگزینی FIFO چند نقض صفحه رخ می دهد.

| تقاضا | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 |
|----------|---|---|---|---|---|---|---|---|---|---|---|---|
| قاب 1 | 0 | 0 | 0 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
| قاب 2 | | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 |
| قاب 3 | | | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 3 |
| نقض صفحه | * | * | * | * | * | * | * | | | * | * | |

ملاحظه می شود که در این مثال 9 بار فضای نقض صفحه رخ می دهد.

ناهنجاری بلیدی: اگر در الگوریتم جایگزینی با افزایش تعداد قاب ها (Frame) تعداد فضای صفحه افزایش یابد کوئیم آن الگوریتم دارای ناهنجاری بلیدی است که الگوریتم FIFO یکی از آنها می باشد.

ادامه دارد.....