دانشگاه آزاد اسلامی واحد تهران شمال

دانشکده فنی و مهندسی

رشته:

مهندسی کامپیوتر

عنوان جزوه:

سیستم عامل

استاد:

جناب آقای راشدی اشرفی

کد جزوه:

۲۲۰

چاپ و تکثیر
انتشارات دانشکده فنی و مهندسی
شرکت سپهر تکثیر تابان

فصل اول

مفاهیم و تعاریف اولیه ی سیستم عامل

هدف: سیستم عامل چگونه به وجود آمد و چه کاری را انجام می دهد؟

- ✓ ۱- مقدمه — Introduction
- ✓ ۲- ساختمان سیستم کامپیوتر — Computer System Structure
- ✓ ۳- ساختمان سیستم عامل — Operating System Structure
- ✓ ۴- مدیریت پردازش — Process Management
- ✓ ۵- هماهنگی پردازش — Process Coordination
- ۶- وقفه یا بن بست — Deadlocks
- ✓ ۷- مدیریت حافظه — Memory Management
- ✓ ۸- حافظه مجازی — Virtual Memory
- ✓ ۹- مدیریت حافظه — Memory Management
- ۱۰- سیستم فایل ها — File Systems
- ✓ ۱۱- محافظت — Protection

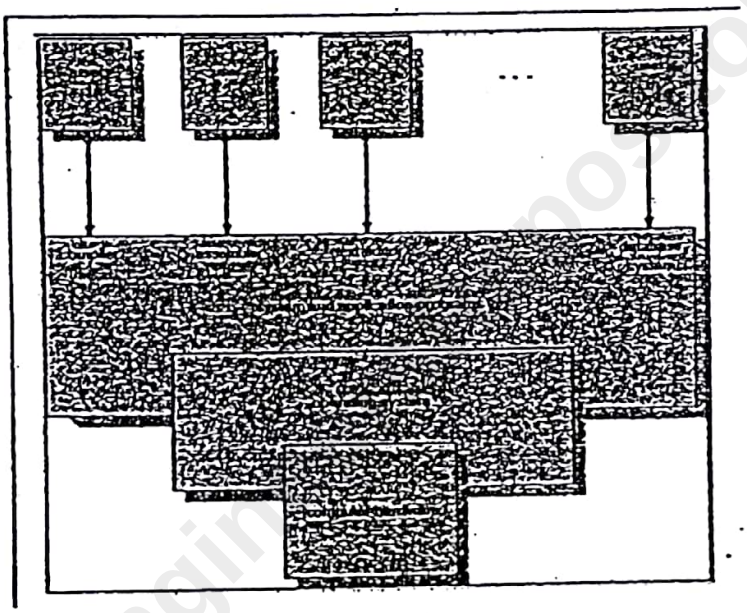
اهداف: راحتی کاربر - بهینه سازی استفاده از منابع

تعریف سیستم عامل Operating system definition

۱- رابط بین سخت افزار کامپیوتر با Application program و User می باشد.

۲- از ابتدای شروع کار کامپیوتر تا پایان وجود دارد مثلاً زبان بیسیک را صدا می زنیم ولی OS اینطور نیست و داخل حافظه نمی باشد.

۳- سیستم عامل به طور هماهنگ سخت افزار کامپیوتر را مدیریت می کند و هماهنگ کننده کنترل از میان برنامه های مختلف مثلاً پرینت کردن می باشد.



هدف سیستم عامل

۱- استفاده از کامپیوتر را آسان می نماید (چرا؟) چون اگر ماشین را روشن نمایم و OS نباشد هیچ عملی نمی توانیم انجام دهیم و روی صفحه ی مانیتور چیزی نمی آید.

۲- استفاده ی بهینه از سخت افزار کامپیوتر

استفاده جهت ارسال

هر سیستم از چهار قسمت تشکیل شده که عبارتند از:

سخت افزار	Hardware
سیستم عامل	Operating system
برنامه های کاربردی	Application programs
کاربر	User

سیستم عامل مدیریت کامپیوتر را به عهده دارد و هیچ خروجی ندارد و برنامه یا نرم افزاری است که از شروع کار با کامپیوتر تا پایان کار آن در حا اجرا است و می توانیم تعریف های دیگری نیز داشته باشیم از جمله سیستم عامل تخصیص دهنده ی منابع یا Resource allocator می باشد که این منابع عبارتند از:

زمان پردازنده	CPU time
فضای حافظه	Memory space
فضای ذخیره ی فایل	File storage pace
وسایل ورودی خروجی	I/O devices

هدف اصلی سیستم های کامپیوتری اجرای برنامه ی کاربر (User) و سهولت حل مسائل کاربر می باشد. از آنجا که استفاده از سخت افزار ساده نیست لذا برنامه های کاربردی یا Application programs ایجاد شدند. این برنامه های مختلف کاربردی احتیاج به عملیات مشترک مثل کنترل کردن Input/output دارند و توابع مشترک کنترل کردن (Control programs) و اختصاص دادن منابع (Resource allocating) تواماً باعث شده است که یک نرم افزار به نام سیستم عامل یا OS بوجود آید.

۲-۱ سیستم های اولیه Early systems

در اوایل سال های ۱۳۵۰ فقط سخت افزار کامپیوتر وجود داشت. کامپیوترهای اولیه ماشین های بسیار بزرگی بودند (از نظر فیزیکی) که از Console (مانیتور کنترل اجرای دستگاه) به اجرا در می آمدند. برنامه نویس هم برنامه را می نوشت و هم به عنوان اپراتور مستقیماً با کنسول کار می کرد. برنامه نویس نخست برنامه را بصورت دستی در حافظه کامپیوتر load می کرد که برای این کار از کلید های روی دستگاه که هر دستور را در یک زمان در حافظه کامپیوتر load می کرد، استفاده می نمود.

سپس دکمه های (buttons) مناسب زده می شد تا مجموعه ای از آدرس های شروع و اجرای عملیات برنامه شروع شود. درحالیکه برنامه اجرا می شد برنامه نویس/ اپراتور می توانست نظاره گر اجرای برنامه باشد و محتویات حافظه و رجیسترها را آزمایش یا تست نموده و لذا برنامه را غلط گیری کند(Interactive Hands on) ولی در این روش اشکالات زیادی وجود داشت. اگر فردی برای یک ساعت وقت کامپیوتر را اجاره می کرد ممکن بود در طول یک ساعت به طور کامل اجرا نشود، لذا از آنجا که ممکن بود برنامه نویس بعدی وقت کامپیوتر را اجاره کرده باشد، برنامه نویس اول مجبور بود برنامه اش را قطع نماید و این کار هم هزینه ی زیادی در برداشت و هم وقت زیادی تلف می شد زیرا برنامه نویس اول مجبور بود در زمان دیگری دوباره load نمودن برنامه را از اول شروع نماید.

حالت دیگر این بود که برنامه قبل از زمان مورد نظر به اتمام برسد که در این صورت برنامه نویس اجاره ی بیشتری را پرداخت کرده بود. به تدریج نرم افزار ها و سخت افزار های جدید مثل کارت خوان یا Card reader، پرینترهای خطی، نوارمغناطیسی Magnetic tapes، اسمبلر linker-loader طراحی شدند تا مسایل برنامه نویسی را ساده کنند. کتابخانه ها با library ها از توابع مشترک بوجود آمدند که این توابع مشترک می توانستند داخل برنامه های جدید کپی شوند بدون اینکه نیاز باشد دوباره type شوند. Routineهایی که برای خواندن از I/O وجود داشتند خیلی اهمیت داشتند چون هر دستگاه I/O مشخصات مخصوص به خود را داشت و به همین خاطر نیاز به برنامه نویسی بسیار دقیقی بود لذا برای هر دستگاه I/O یک سری روتین منحصر به فرد نوشته شد که این روتین ها به device divers معروف بودند هر D.D می دانست که از بافرها، پرچم ها (Flags)، رجیستر ها، بیت های کنترل (Control bits) برای دستگاه بخصوص بایستی استفاده شود. یعنی هر دستگاه یک driver مخصوص به خود را داشت. مثلاً یک عمل ساده مثل خواندن یک کاراکتر از دستگاه نوارخوان (Tape reader) ممکن است عملیت پیچیده ای را درگیر نماید. به همین خاطر device divers می توانند به سادگی مورد استفاده قرار بگیرند بدون اینکه نیازی باشد هر دفعه کدهای لازم باز نویسی شوند. سپس کامپایلر فرترن و کوبول و بقیه ی زبان ها ظاهر شدند و برنامه نویسی را خیلی ساده تر کردند ولی عملیات کامپیوتری مشکل تر شد. برای مثال برنامه ی فرترن برای اجرا باید مراحل زیر را طی کند:

- نخست برنامه نویس لازم بود کامپایلر فرترن را در کامپیوتر load کند که معمولاً کامپایلر ها روی نوارهای مغناطیسی نگهداری می شدند. سپس لازم بود که نوار کامپایلر مورد نظر روی دستگاه Tape نصب شود و برنامه از طریق کارت خوان خوانده شود و روی نوار مغناطیسی دیگری نوشته شود و کامپایلر فرترن، زبان اسمبلی را تولید کند و سپس برنامه باید اسمبل می شد. Outputهای اسمبلر می بایست با روتین های کتابخانه ای که ایجاد شده بودند لینک می شدند و در پایان کار، کد باینری برنامه برای اجرا آماده می شد تا object code بوجود آید. سپس در حافظه load واجرا می شد. لذا در کلیه ی مراحلی که ذکر شد مقدار بسیار زیادی وقت صرف setup و آماده کردن دستگاه می شد و در هر مرحله امکان داشت به جهت خطایی (error)، اجرای برنامه دوباره از اول شروع شود.

۱) کامپایلر فرترن ← load در کامپیوتر
2) نوار کامپایلر ← نصب روی دستگاه Tape
3) خوانده شدن برنامه از طریق کارت خوان و نوشته شدن بر نوار
4) تولید زبان اسمبلی
5) ...
6) آماده شدن کد باینری برای اجرا

Simple Monitor ۱-۳

زمان Setup نمودن برای اجرای هر Job مساله ی بسیار بزرگی بود، در مدت زمانی که نوارهای مغناطیسی روی دستگاه قرار می گرفتند و برنامه نویس روی کنسول کار می کرد، CPU بی کار بود. به خاطر داشته باشید در اوایل به وجود آمدن کامپیوترها تعداد کمی کامپیوتر وجود داشت و بسیار گران قیمت بودند. (علاوه بر هزینه برق و تبرد کردن دستگاه) از طرف دیگر، کامپیوترها دستگاه با ارزشی بودند که صاحبانشان می خواستند از آنها تا حد امکان استفاده کنند تا از سرمایه گذاری خود استفاده مادی ببرند، لذا برای استفاده بهتر از کامپیوتر ها دو راه حل وجود داشت:

۱- اپراتورهای حرفه ای کامپیوتر استخدام شوند.

دیگر نیاز نبود برنامه نویس روی دستگاه کامپیوتر شخصاً کار نماید. به محض این که یک Job تمام می شد، اپراتور می توانست برنامه یا Job دیگری را شروع نماید، پس وقت بیکاری دستگاه کمتر شد. از آنجا که اپراتور تجربه بیشتری پیدا می کرد (از لحاظ نصب نوارها روی دستگاه) لذا زمان setup کاهش پیدا می کرد. همچنین برنامه نویس فقط یک توضیح کوتاه در مورد اجرای Job خود به اپراتور ارائه می داد. البته اپراتور نمی توانست یک برنامه را خودش غلط گیری کند زیرا برنامه را نمی فهمید و اطلاعی از برنامه نویسی نداشت. لذا در حالیکه یک برنامه error داشت یک dump غلط از آن تهیه می شد. این عمل اجازه می داد که اپراتورJob بعدی را شروع نماید ولی برنامه نویس با مساله مشکل تر debugging مواجه می شد.

۲- Job هایی که احتیاجات مشترک داشتند در یک گروه، batch یا دسته بندی می شدند.

به عنوان مثال اگر اپراتور یک Job فرترن، یک Job کوبل و یک Job فرترن دیگری را دریافت می کرد. اگر او به ترتیبی که Job ها را دریافت می کرد آنها را اجرا می نمود می بایست اول کامپیوتر را برای فرترن setup می کرد، سپس برای کوبل و در پایان مجدداً برای فرترن. لذا اگر دو برنامه ی فرترن را به عنوان batch یا دسته اجرا می کرد، نیاز بود که یک بار برای فرترن setup نماید. در این صورت زمان setup کمتر می شد و این تغییرات اپراتور را از User جدا می ساخت.

اما همچنان مشکل وجود داشت. به عنوان مثال زمانی که یک Job متوقف می شد، اپراتور می بایست از طریق مشاهده کنسول مشخص می کرد که چرا برنامه متوقف شده است (خاتمه معمولی normal است و یا برنامه error دارد) و اگر احتیاج بود dump از برنامه بردارد در این صورت در زمانی که انتقال از یک Job به Job دیگر صورت می گرفت باز CPU بیکار می ماند.

Automatic Job Sequencing (A.J.B)

سبب ردیف کردن Job ها می شود.

برای غالب شدن به مشکلات فوق A.J.B به وجود آمد. این تکنیک در واقع اولین سیستم عامل بنیار ابتدایی بود که اختراع گردید. آنچه که مورد نظر بود یک پروسیجر یا روشی برای انتقال کنترل اتوماتیک از یک Job به Job های دیگر بود. برای این منظور یک برنامه ی کوتاه به نام Resident Monitor (R.M) نوشته شد که در این برنامه همیشه حافظه ای (Memory) وجود داشت که عمل آن باعث می شد کنترل به برنامه بعد انتقال یابد. به محض اینکه Job یا برنامه اول تمام می شد، دوباره کنترل به Resident Monitor می رفت. این بود که کنترل بطور اتوماتیک از یک Jobبه Job دیگر به صورت ردیفی انتقال پیدا می کرد

اما چطور R.M می فهمد که کدام Job می بایست اجرا شود؟ در روش قبل به اپراتور یک توضیح کوتاه داده می شد که کدام برنامه یا data باید اجرا شود. ولی در A.J.B این اطلاعات مستقیماً برای Resident Monitor (R.M) تهیه می شد. لذا برای انجام این کار Control Card ها معرفی شدند. نحوه ی عمل آنها بسیار ساده است. یعنی علاوه بر داده ها و برنامه ها، کارت های مخصوصی را داخل برنامه (که همان control card ها هستند) وارد کنیم. که کار آنها راهنمایی R.M می باشد و مشخص می کند که کدام برنامه می بایست اجرا شود به عنوان مثال کاربران ممکن است بخواهند که یکی از سه برنامه ی زیر را اجرا کنند:

کامپایلر فرترن FTN

اسمبلر ASM

اجرا کردن برنامه RUN

لذا ما می توانیم از یک کنترل کارت جداگانه برای هر کدام از این برنامه ها استفاده کنیم.

اجرا کردن کامپایلر فرترن $FTN

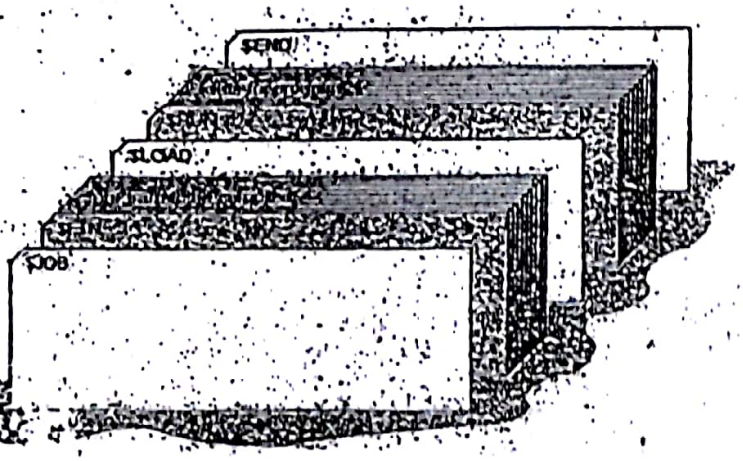
اجرا کردن اسمبلر $ASM

اجرا کردن برنامه (User را مثلاً) $RUN

در واقع این کارت های کنترل به Resident Monitor (R.M) می گویند که کدام برنامه را اجرا کند. همچنین نیاز است که دو کارت اضافی کنترل دیگر برای تعریف حدود هر Job استفاده کنیم که این ها عبارتند از:

$JOB که ابتدای Job را مشخص می کند.

$END که انتهای Job را مشخص می کند.



یک نمونه برای سیستم فوق بصورت روبه رو نشان داده می شود:

•روی هر کارت یک خط دستور نوشته می شود و هر (!) نشان دهنده یک کارت می...

اجزای تشکیل دهنده ی Resident Monitor عبارتند از:

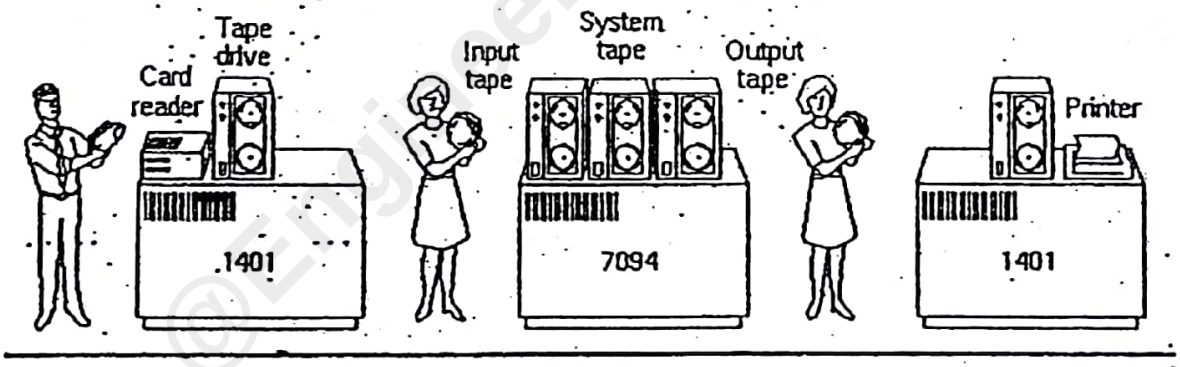
۱) Loader ← لود کردن برنامه های سیستم و کاربری کاربر به داخل حافظه
کنترل I/O ها

۲) Job Sequencing ← ردیف کامل هایی که راکد از طریق کنترل کارت مشخص شده است، نگه داری می کند.

۳) Control Card Interpreter : مسئول خواندن و اجرای دستورات روی کارت ها در هنگام اجرای برنامه است.

مهم ترین قسمت R.M Control Card Interpreter (CCI) است که مسئول خواندن و اجرای دستورات روی کارت ها در هنگام اجرای برنامه است و CCI در فواصل زمانی، loader را برای load نمودن System Programs (برنامه های سیستم) و برنامه های Application User Programs به داخل حافظه (Memory) درگیر می کند. همچنین CCI به loader نیاز دارد که I/O را کنترل کند. لذا Resident Monitor دارای یک مجموعه از Device Driver ها برای دستگاه های I/O نیز می باشد و A.J.S ردیف Job ها را که از طریق کنترل کارت مشخص شده است نگهداری می نماید. پس زمانی که کارت کنترل مشخص می کند که یک برنامه می بایست اجرا شود، R.M برنامه را به داخل حافظه load نموده و کنترل را به آن انتقال می دهد و زمانی که اجرای برنامه تمام شد، کنترل را به R.M انتقال داده و در این حالت کارت کنترل بعدی را خوانده و برنامه مربوطه را load می کند و به همین ترتیب عملیات ادامه می یابد.

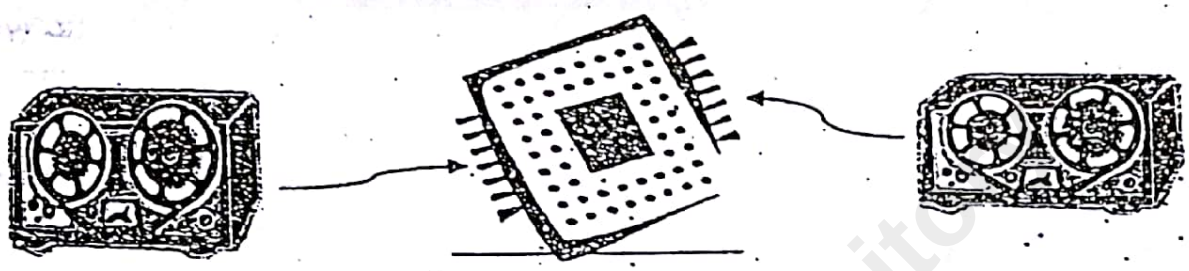
۱-۴ Off-line Operation:



کامپیوترها دارای دستگاه های I/O هستند که سرعت این دستگاه های مکانیکی بسیار آهسته تر از دستگاه های الکترونیکی (CPU) هستند، حتی یک CPU بسیار کند می تواند میلیون ها دستور را در یک ثانیه انجام دهد از طرف دیگر یک دستگاه کارت خوان بسیار سریع جداکثر ممکن است هزار کارت را در دقیقه بخواند. لذا اختلاف سرعت بین دستگاه های I/O و CPU همیشه وجود داشته است. به این معنی که کند بودن دستگاه های I/O مساوی با این است که اغلب، CPU به علت بی کاری باید منتظر I/O بماند. به عنوان مثال یک کامپایلر قادر است ۳۰۰ کارت یا بیشتر را در هر ثانیه پردازش نماید. از طرف دیگر یک کارت خوان بسیار سریع نادر است فقط ۱۲۰۰ کارت را در دقیقه بخواند و این به این معنی است که کامپایل نمودن یک برنامه ۱۲۰۰ کارتو فقط ۴ ثانیه از وقت CPU را لازم دارد. اما برای خواندن همین تعداد کارت، کارت خوان نیازیه ۶۰ ثانیه دارد. لذا CPU د

۶۰ ثانیه فقط ۴ ثانیه مشغول بوده و در ۵۶ ثانیه دیگر بیکار و منتظر خواهد ماند که این حالت برای عملیات خروجی هم صادق است. اگرچه در طول زمان، پیشرفت تکنولوژی باعث بیشتر شدن سرعت دستگاه های I/O شده است اما سرعت CPU بیشتر نیز شده است. یک راه حل کلی برای حل مشکل فوق جایگزین نمودن کارت خوان های بسیار کند و چاپگرهای خطی با نوار مغناطیسی بود. در این حالت به جای اینکه CPU مستقیماً از کارت خوان ها و چاپگرهای خطی به صورت OFF-LINE انجام وظیفه می کردند.

Buffering and Spooling ۱-۵



بافرینگ یعنی Overlap کردن CPU و دستگاه های I/O مانند Card reader که سرعتش نصف CPU است. پس هر دو Card reader با CPU تنظیم می شوند.

در Spooling از دیسک به عنوان یک بافر سریع استفاده می شود و یک بافر بزرگ برای I/O در نظر گرفته می شود. دستاورد واقعی در عملیات OFF-Line امکان استفاده از چند Card Reader-to-tape و Tape-to-printer برای یک CPU بود.

اگر یک CPU بتواند دو برابر سرعت یک کارت خوان Input را پردازش نماید. پس دو کارت خوان می توانند به صورت همزمان کار کنند تا به اندازه ی کافی نوار تهیه نمایند و CPU را مشغول نگه دارد. پس عملیات پردازش OFF-line اجازه می دهد که CPU و I/O با یکدیگر هماهنگ شوند. (Overlap) اگر ما مایل باشیم که همین هماهنگی را با یک دستگاه ورودی انجام دهیم می بایست بین دستگاه های I/O و CPU روش دیگری را در پیش گیریم تا اجازه ی جلسازی مشابه را بدهیم.

Buffering ۱.۵.۱:

بافرینگ یک روش Overlap یا هماهنگ نمودن I/O یک Job با محاسبات همان Job که روش بسیار ساده ای است:

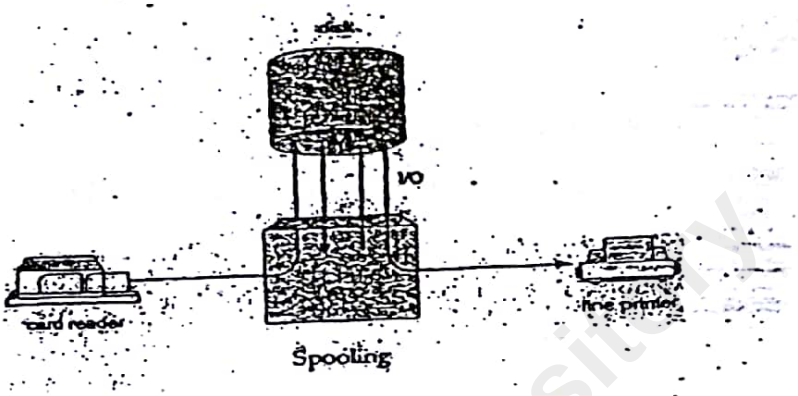
بعد از اینکه داده ها خوانده شدند و CPU عملیات روی آنها را شروع می کند به دستگاه Input دستور داده می شود تا بلافاصله Input بعدی را نیز بخواند لذا هم CPU و هم دستگاه ورودی هر دو مشغول اجرا یا کار هستند. لذا زمانی که CPU برای پردازش داده بعدی آماده است، دستگاه ورودی می بایست کار خواندن رکورد بعدی را تمام کرده باشد که در این حالت CPU می تواند پردازش رکورد بعدی را شروع نماید، در حالی که دستگاه ورودی خواندن رکورد بعدی را شروع کرده باشد.

اما در عمل buffering به ندرت می توان هم CPU و هم I/O را در همه ی زمان ها مشغول نگه داشت. به خاطر اینکه یا CPU یا I/O کارشان را زودتر تمام می کنند. چنانچه CPU زودتر پردازش را به اتمام برساند، باید منتظر Input بعدی باشد و نمی

تواند تا زمانی که رکورد بعدی در حافظه موجود نباشد کاری انجام دهد و بالعکس، اگر Input کارش را زودتر تمام کند بایستی منتظر CPU بماند یا ممکن است شروع به خواندن رکورد بعدی نماید.

بافرهایی که رکوردها را نگه می دارند اغلب به اندازه ی کافی بزرگ هستند تا چندین رکورد را نگهداری نمایند. در این صورت یک دستگاه ورودی می تواند چندین رکورد را جلوتر از اجرای CPU بخواند و ذخیره نماید.

Spooling ۱.۵.۲:



اگرچه با اجرای عملیات OFF-Line برای Job ها مدت زمان زیادی ادامه داشت و به سرعت در سیستم های قبلی جایگزین شد، سیستم های دیسک بطور گسترده تر بوجود آمدند و عمل OFF-Line را بسیار توسعه دادند. مشکل استفاده از سیستم های Tape یا نوار این بود که کارت خوان نمی توانست روی یک انتهای نوار بنویسد، در حالیکه CPU از قسمت دیگر آن رکورد را بخواند؛ ولی سیستم های دیسک این مشکل را نداشتند به این معنی که بعد از یک قسمت از دیسک به قسمت دیگر به سرعت انتقال داده می شدند و نیز بعداً می توانستند به سرعت به نقطه ای که CPU در روی دیسک احتیاج دارد رجوع کند تا data بعدی را بخواند. پس به سرعت انتقال داده می شدند که این فرم از پردازش Spooling نامیده می شود. به شرح شکل فوق.

در واقع spooling بطور اساسی از دیسک به عنوان یک بافر بسیار بزرگ برای خواندن استفاده می شود. (تا آنجا که ممکن است زودتر از دستگاه های ورودی و برای ذخیره کردن فایل های خروجی تا زمانی که دستگاه های خروجی قادر باشند آنها را بپذیرند)

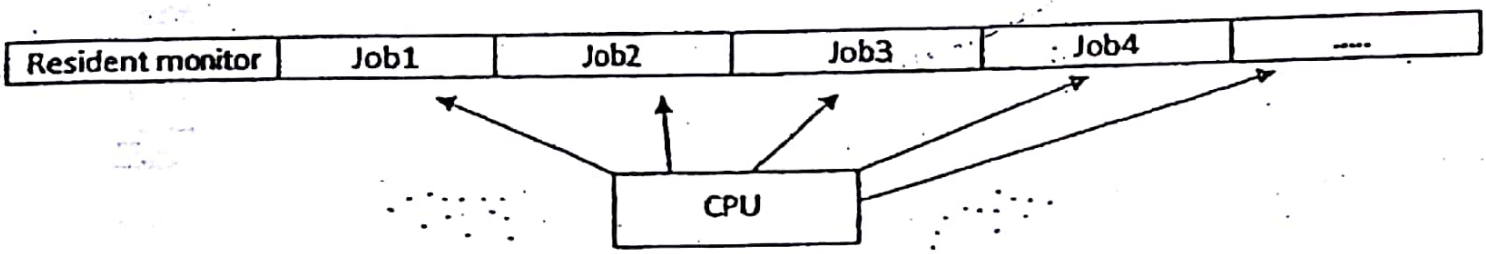
اسپولینگ، I/O یک Job را با محاسبات Job های دیگر هماهنگ می کند و همچنین Spoolingیک مزیت بسیار مفید دارد. به صورتی که با مقداری هزینه استفاده از فضای خالی دیسک و تعداد کمی جدول، CPU می تواند هماهنگی لازم را برای محاسبات یک Job و I/O ی Job های دیگر برقرار نماید. لذا Spooling ساختمان داده بسیار مهمی را به نام Job pool معرفی می نماید. معنی دقیق تر Spooling یعنی اجرای چند Job که همگی از روی دیسک خوانده شده اند و برای اجرا آماده می باشد و یک Job pool این اجازه را به سیستم عامل می دهد که کدام Job را برای اجرا انتخاب نماید، بطوریکه کارایی CPU را افزایش دهد.

در فصل ۴ در این مورد مفصلاً بحث خواهد شد.

در اینجا چند زمینه مهم که در Spooling استفاده می شود. مختصراً شرح داده می شود که عبارتند از:

Time sharing, Multi Tasking, Multi Programming

۶.۱ Multi Programming:

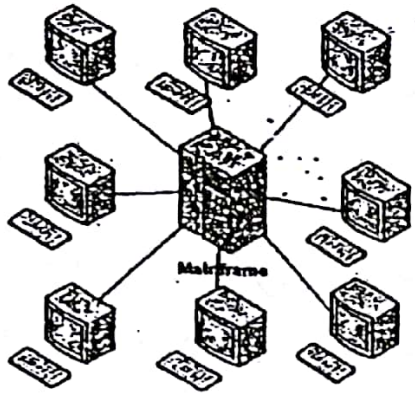


مهم ترین استفاده از CPU Scheduling توانایی در Multi Programming است.

نمی تواند هم CPU وهم دستگاه های I/O را مشغول نگه دارد . از طرف دیگر Multi Programming کارایی CPU را توسط دسته بندی نمودن Job ها افزایش می دهد، بطوریکه CPU همیشه کاری برای انجام دادن دارد. M.P به شرح زیر است:

سیستم عامل یک Job را از Job pool را انتخاب نموده و شروع به اجرا می نماید بالاخره Job فوق مجبور می شود تا برای یک Task یا موضوعی منتظر بماند (مثل نصب یک نوار مغناطیسی روی دستگاه نوار یا یک دستور که می بایست در Keyboard تایپ شود) که در این صورت سیستم عملا به سادگی شروع به اجرای Job دیگری می کند. وقتی که این Job هم لازم شد که برای یک عمل دیگر منتظر بماند Job CPU به Job دیگری Switch یا انتقال می نماید... در آخر کار Job اول انتظارش به سرآمده (یعنی خواندن یا هر کار دیگری تمام شده) و CPU را به خودش بر می گرداند، لذا تا زمانی که Job هایی وجود دارد که اجرا شوند CPU هیچ وقت بی کار نخواهد ماند. سیستم عامل Multi Programming نسبتاً پیچیده است به این دلیل که چند Job پیوسته در حال آمادگی برای اجرا هستند، سیستم باید همه ی آنها را نگهداری نماید. لذا سیستم عامل احتیاجی به فرم هایی از مدیریت حافظه را دارد که در فصل های ۷ و ۸ در این مورد بحث می شود. به علاوه اگر چند Job آماده باشند تا در یک زمان اجرا شوند، سیستم عامل می بایست از میان آن ها یکی را انتخاب نماید. این تصمیم به نام CPU Scheduling یا Job نامیده می شود که در فصل ۴ در مورد آن بحث می شود. بالاخره چندین Job که همزمان می توانند اجرا شوند، بعضی محدودیت ها را در تاثیر پذیری متقابل بوجود می آورند که می بایست در مرحله های مختلف سیستم عامل محدود شوند که شامل Process scheduling و disk space management و Memory management هستند.

۷.۱ Multi tasking or Time sharing:



Time-sharing یا Multitasking بسط منطقی از Multi Programming است. یک سیستم عامل Batch (دسته بندی)، Multi Programming است. معمولاً یک رشته از Job های جداگانه را مثلاً از کارت خوان می خواند و هر کدام از این Job های کنترل ، کارتهای مختلفی دارند که طرز عمل آن Job را تعریف می کنند و خروجی اش معمولاً چاپ می شود. یک سیستم batch ، در زمانی که Job در حین اجراست، کمبود Interaction یا تاثیر متقابل بین User و Job دارد. چند اشکال با سیستم batch از دیدگاه یک برنامه نویس یا User عبارتند از:

۱) از آنجا که User نمی تواند با Job اش در حالی که در حال اجراست، تماس متقابل داشته باشد؛ User می بایست کنترل کارتها را طوری تهیه نماید یا setup کند که مشکل برطرف شود و یا همه ی حالت ها را در نظر بگیرد.

۲) زمانی که لازم است برنامه ای error گیری یا debug شود برنامه نویس می بایست از dump برنامه برای error استفاده نماید. یعنی برنامه نویس نمی تواند در حین اجرای برنامه رفتار آن را کنترل نماید. لذا Turn around های طولانی برای اجرای برنامه همیشه وجود خواهد داشت، در صورتی که سیستم Interactive یا Hands-on یک ارتباط on-line بین User ها و سیستم کامپیوتر تهیه می کند.

یک سیستم عامل Time sharing از CPU Scheduling و Multiprogramming استفاده می کند تا برای هر User یک قسمت کوچک از کامپیوتر تهیه نماید که به صورت زیر عمل می شود:

هر User یک برنامه جداگانه در حافظه یا Memory دارد. زمانی که یک برنامه بایست اجرا شود فقط برای یک زمان کوتاه وقت CPU به آن داده می شود که یا در این فاصله کوتاه آن برنامه تمام می شود و یا آنکه احتیاج به اجرای یک I/O خواهد داشت. I/O ممکن است Interactive باشد، یعنی اینکه خروجی می بایست روی مانیتور (CRT) چاپ شود و Input از keyboard داخل شود و دراین مدت زمان که نیاز به I/O است، CPU بیکار ننشسته و سیستم عامل به سرعت CPU را به برنامه ی دیگری

انتقال می دهد از آنجا که سیستم به سرعت از یک کاربر یا User به User های دیگری انتقال پیدا می کند این مدت بسیار کوتاه توسط کاربران قابل احساس و درک نیست.

ایده ی Time sharing در اوایل 1960 بوجود آمد. اما از آنجا که این سیستم ها بسیار گران قیمت و مشکل بودند تا سال های ۱۹۷۰ آنها نتوانستند عمومیت پیدا کنند تا اینکه در اوایل دهه ی ۱۹۷۰ محبوبیت و شهرت سیستم های Time sharing بیشتر پدیدار شد و همچنین محققان کوشیدند تا سیستم های Multi Programming و Time sharing را در یک سیستم ترکیب کنند به عنوان مثال IBM OS/360 به سیستم TOS یا Time Sharing Uptime تغییر یافت. امروزه بیشتر سیستم ها هم Multiprogramming و هم Timesharingرا تهیه می کنند.

سیستم عامل های Time sharing بسیار پیچیده هستند، آنها می بایست مکانیزمی را تهیه کنند تا امکان اجرای همزمان برنامه ها را ایجاد کنند. (فصل های ۴ و ۵) و نیز مانند Multiprogramming چند Job را می بایست همزمان در حافظه نگهداری نمایند یعنی به مدیریت حافظه و حفاظت با Protection و CPU Scheduling و همچنین یک زمان پاسخگویی منطقی نیاز دارند. Time slice و Job ها ممکن است که مجبور باشند حافظه اصلی را به دیسک swap نماید لذا می بایست Disk Management نیز استفاده شود (فصل ۹) و همچنین Timesharing می بایست یک سیستم فایل On-line را نیز تهیه نماید. (فصل ۱۰). لازم به تذکر است که Multi Programming و Timesharing فایل های مرکزی و اصلی سیستم عامل های مدرن هستند

۸.۱ سیستم های توزیع شده Distributed Systems

یکی از آخرین تحولات در سیستم های کامپیوتری این است که محاسبات را در میان چند پردازش کننده ی فیزیکی توزیع می کند دو طرح برای ساخت چنین سیستم هایی وجود دارد:

۱- Tightly Coupled System

پردازش کننده یا پروسسور، حافظه و ساعت را به طور اشتراکی مورد استفاده قرار می دهند و در آن ارتباطات یا Communication از طریق پرت اشتراکی انجام می شود.

۲- Loosely Coupled System

پردازش کننده ، حافظه یا کلاک را شریک نمی کند و هر پردازش کننده حافظه داخلی خودش را دارد و پردازش کننده ها با یکدیگر از طریق خط های مختلف ارتباطی مثل خط تلفن ارتباط برقرار می کنند. پردازش کننده ها در یک سیستم توزیع شده (Distributed) در اندازه و عمل با یکدیگر فرق دارند. ممکن است آنها دارای Microprocessor ، Minicomputer ها و Mainframe ها باشند.

مزیت های سیستم های توزیع شده (Distributed) عبارتند از:

۱- مشارکت منابع یا Resource ها:

در صورتیکه چند سایت مختلف به یکدیگر متصل شوند یک User در یک سایت می تواند از منابع موجود در سایت های دیگر استفاده نماید. به عنوان مثال پرینتر لیزری اگر در سایت A باشد، می تواند در سایت B مورد استفاده قرار بگیرد.

۲- بالارفتن سرعت محاسبات:

اگر یک محاسبه ی بخصوص بتواند به تعدای محاسبات کوچکتر تقسیم شود، سیستم های Distributed این اجازه را می دهد که آن محاسبات را هم زمان در سایت های مختلف اجرا گردند.

۳- اعتماد:

به این معنی است که اگر یک سایت distributed دچار مشکل شده و از کار بیافتد می توان از بقیه ی سایت ها برای اجرای عملیات سایت از کار افتاده استفاده نمود.

۴- ارتباطات Communications:

از آنجا که سایت ها توسط شبکه ی ارتباطی با هم در ارتباط هستند، لذا داری این خاصیت هستند که اطلاعات را با یکدیگر مبادله نمایند. مثل انتقال اطلاعات و یا برقراری ارتباط از طریق ایمیل.

I Love You

1.9 Real Time System:

یک فرم دیگری از سیستم عامل، سیستم های Real Time است.

R.T.S اغلب به عنوان دستگاه کنترل در دستگاه ها و سیستم‌های خاصی به کار برده می شوند سیستم هایی که آزمایشات علمی را کنترل می کنند، سیستم های تصویری پزشکی، سیستم های کنترل صنعتی، سیستم های Real Time هستند. همچنین کنترل وسایل خانگی مثل Microwave و غیره نیز از جمله هستند.

یک سیستم عامل Real Time محدودیت های زمانی ثابت تعریف شده ای دارد، یعنی پردازش می بایست بین محدودیت های تعریف شده انجام شود، در غیر این صورت سیستم از کار خواهد افتاد. به عنوان مثال برای بازوی یک ربات این مهم است که در زمان خاصی و یا مدت زمان محدودی کاری را روی ماشین انجام دهد و گرنه آن سیستم از کار خواهد افتاد. این محدودیت زمانی باعث می شود که سیستم های Real Time دارای امکانات خوبی باشند، یعنی هیچ نوع حافظه مجازی (دیسک) وجود ندارد و اگر وجود دارد ، محدود هستند. ولی در عوض داده ها (data) در حافظه های دائمی مثل (ROM) نوشته می شوند که محتوایش با قطع برق هم از بین نمی رود.

تکلیف:

۱- فایده ی اصلی Multi programming چیست؟

۲- چرا برای Multi Programming، Spooling لازم است؟ آیا برای سیستم های Time Sharing هم لازم است؟

۳- در محیط Multi Programming و Time sharing چند User همزمان می توانند از سیستم استفاده کنند؟ آیا این حالت منجر به مشکلات امنیتی مختلفی خواهد شد؟

الف) این مشکلات کدامند؟

ب) آیا می توانید همان درجه ی اطمینانی را که در کامپیوتر های اولیه داریم در Time sharing هم داشته باشیم یا نه؟

(فصل دوم)

ساختمان سیستم کامپیوتر

قبل از اینکه بتوانیم جزئیات عملیات سیستم را کشف کنیم، احتیاج است که یک دانش کلی نسبت به ساختمان سیستم کامپیوتر داشته باشیم. در این فصل چندین قسمت مجزا از این ساختمان ارائه می شود تا اطلاعات کلی جانبی خود را تکمیل کنیم(از آنجا که سیستم های Multi Programming و Time sharing از overlap نمودن CPU و عملیات I/O استفاده می کنند، ساختمان دو ماشین استاندارد را که برای انجام دادن این Overlap به کار می روند را مورد بررسی قرار می دهیم:

۲.۱ Interrupted based system:

برای اینکه CPU و I/O، Overlap شوند، می بایست دارای مکانیزمی باشند که (هماهنگ desynchoronization و دوباره هماهنگ Resynchronization) که برای هماهنگ و موزون نمودن هریک از متدهای زیر مورد استفاده قرار گیرد. بعضی از سیستم ها هردوی آنها را به کار می برند.

۱- Interrupt- driven Data Transfer

۲- DMA (Direct Memory Access) Data Transfer

اما در سیستم های قدیمی این Data Transfer یا انتقال داده های تحت کنترل CPU مجبور بودند هم انتقال اطلاعات را انجام دهند و هم آنها را تحت نظر داشته باشند.(به عنوان مثال مراحلی که برای چاپ اطلاعات از حافظه نیاز است عبارتند از:

۱- CPU می بایست چک کند که آیا پرینتر برای چاپ کارکتر بعدی آماده است یا خیر.

۲- اگر پرینتر آماده نبود به مرحله یک باز گردد.

۳- اگر پرینتر آماده است، چک کند که آیا کاراکتر دیگر برای چاپ موجود است یا خیر.

۴- اگر کارکتر دیگری وجود دارد به مرحله ی ۱ باز گردد.

۵- اگر کارکتر دیگری وجود ندارد عملیات چاپ پایان یافته است.

این متد over lap نمودن، busy waiting نامیده می شود. یعنی اینکه CPU می بایست همیشه در حال چک کردن I/O باشد، بنابرین Busy است. در حالیکه برای I/O نیز منتظر است تا تمام شود

شکل بالای صفحه ۲۱

اگر چه در تئوری این درست که CPU می تواند پردازش های دیگری را انجام بدهد و برگردد و کاراکتر بعدی را چاپ نماید اما در عمل همیشه این کار امکان پذیر نیست.

Interrupt Based I/O بهترین راه حل مشکل فوق است که ساختمان کلی آن در شکل زیر کشیده شده است.

شکل پائین صفحه ی ۲۱

هردستگاه Controller داری مقداری Buffer و یک مجموعه از رجیسترهای مخصوص است. دستگاه Controller مسئول انتقال اطلاعات بین دستگاه های جانبی و CPU است که بافر داخلی اش را کنترل می کند. آغاز یک عمل I/O بدین صورت است که CPU برای شروع I/O رجیستر مناسب را در دستگاه کنترولر load می کند و سپس عملیات نرمال خود را از سر می گیرد. دستگاه کنترولر به نوبه ی خود محتویات این رجیسترها را تست می کند تا مشخص کند چه عملی را می بایست انجام دهد. به عنوان مثال اگر بفهمد که یک درخواست خواندن از ورودی انجام شده است، آن وقت کنترولر شروع به انتقال data از دستگاه بافر داخلی اش می کند. به محض اینکه انتقال اطلاعات کامل شد دستگاه کنترولر به CPU اطلاع می دهد که عملیاتش پایان یافته است. این اطلاع توسط Interrupt انجام می شود. وقتی که CPU، Interrupt را دریافت کرد هرکاری را که مشغول انجام آن است متوقف می کند و بلافاصله اجرای دستورات را از یک محل ثابت بنام Interrupt Service Routine (I.S.R) ادامه می دهد. این برنامه معمولاً از آدرس شروع، آغاز می شود. اما در حالی که ISR برای Interrupt در نظر گرفته است، اطلاعات از بافر

داخلی دستگاه کنترل به حافظه اصلی انتقال می دهد و زمانی که انتقال انجام شد CPU می تواند محاسبات قطع شده را از سر بگیرد. به این طریق دستگاه های CPU و I/O می توانند هم زمان کار خود را انجام دهند.

Interrupt مهم ترین قسمت ساختمان یک کامپیوتر است. همه ی طرح های کامپیوتر یک مکانیزم Interrupt مخصوص به خود دارند ولی چند کار در آن ها مشترک است:

۱- Interrupt می بایست کنترل را به ISR مشترک در همه ی Interrupt ها انتقال دهد و این عمل با رزرو و ذخیره نمودن یک مجموعه از مکان ها یا محل هایی در حافظه پائین (Low Memory) انجام می شود.

۲- ساختمان Interrupt همچنین می بایست آدرس دستورات Interrupt (یا قطع شده) را ذخیره کند. بیشتر طرح های قدیمی به سادگی آدرس Interrupt را در یک مکان ثابت یا در مکان هایی که توسط شماره دستگاه Index (اندیس) شده بودند، ذخیره می کردند ولی بیشتر آرشیتکت های سال های اخیر آدرس برگشت (Return Address) را روی Stack سیستم ذخیره می کنند. بعد از اینکه Interrupt سرویس دهی شد یک (Jump Back) پرش به عقب به جایی که آدرس برگشتی ذخیره شده است، صورت می گیرد تا محاسبات قطع شده را از سر بگیرد. مثل اینکه Interrupt اصلاً اتفاق نیافتاده است.

معمولاً سایر وقفه ها در زمان پردازش وقفه، ممنوع می شوند و بعد از اتمام کار وقفه ی اول دوباره مجاز می گردند. اگر وقفه ی تو در تو مجاز باشد البته نیازی به این عمل نداریم. در معماری وقفه پیشرفته، وقفه ها دارای طرح اولویت هستند که بر حسب اهمیت نسبی آنها در نظر گرفته شده اند. وقفه ی با حق تقدم بالاتر می تواند هر زمان رخ دهد، ولی معمولاً وقفه های پائین پوشانیده می شوند و یا ممنوع می شوند تا وقفه های ناخواسته و غیر ضروری رخ ندهد.

۲.۲ I/O Structure

اگر هیچ Jobی برای اجرا موجود نباشد نه دستگاه I/O سرویس می دهد و نه User ها به کسی جواب می دهند و سیستم عامل منتظر خواهد ماند تا کاری درخواست شود. شروع کار سیستم عامل تقریباً همیشه توسط یک Interrupt و یا توسط Trap (اجرای غیرعادی) است. (که یک Trap یک انتزاع نرم افزاری است که توسط یک Error (تقسیم بر صفر) و یا توسط یک درخواست مخصوص برنامه ایجاد می شود ~~ایجاد می شود~~). یک سیستم عامل یک Interrupt Driver (دستگاه انقطاع) است. طبیعی است Interrupt Driver یک سیستم عامل است که ساختمان کلی یک سیستم را تعریف می کند و زمانی که یک Interrupt و یا یک Trap اتفاق می افتد، Hardware کنترلر را به OS انتقال می دهد:

۱- OS حالت CPU را با ذخیره نمودن رجیستر ها و کانتر برنامه، حتماً ذخیره می کند.

۲- می بایست مشخص کند که چه نوعی از Interrupt اتفاق افتاده است. این تشخیص ممکن است نیاز به Polling داشته باشد. یعنی جستجوی همه ی دستگاه های I/O تا کشف شود که کدام سرویس درخواست شده و یا ممکن است که یک نتیجه ی طبیعی از سیستم انقطاع باشد. (Interrupt Vector) یا (Interrupt Array) که برای هر نوع از Interrupt بخش های مجزا از کدها در سیستم عامل وجود دارد که مشخص می کند چه عملی باید انجام داد.

زمانی که برنامه ای نیاز به خواندن و یا نوشتن دارد دو حالت اتفاق می افتد:

۱- حالت اول: برنامه شروع به Input و Output می نماید و منتظر می ماند تا I/O تمام شود و سپس کنترل به برنامه برگردد.

۲- حالت دوم: برنامه شروع به خواندن و نوشتن می نماید و کنترل به خود برنامه بر می گردد بدون اینکه منتظر بماند تا I/O تمام شود.

در دوحالت ما نیاز به دستور Wait داریم که CPU را منتظر نگه دارد تا I/O تمام شود. در بعضی از سیستم ها منتظر نگهداشتن با دستور Wait انجام می شود ولی در بعضی سیستم های دیگر از Loop بسته ای به نام Wait loop استفاده می شود. بصورت زیر:

Jump loop :(لوپ انتظار)

مسلم است که دستور Wait بهتر است. به خاطر اینکه در هر زمان یک I/O برجسته است و لذا CPU می داند که Interrupt کی اتفاق خواهد افتاد و از چه دستگاهی خواهد بود. ولی در حالت دوم ما نیاز به جدول حالت- وسیله (Device Status Table) داریم که در آن مشخصات up to date هر دستگاه I/O در آن مشخص شده است و یک ورودی به ازای هر وسیله ی I/O دارد.

۲.۳ مدهای سیستم عامل Dual-mode Operation:

برای اینکه عملیات یک سیستم کامپیوتری به صورت صحیح و درست انجام شود و برنامه های User نتوانند اخلالی در کار سیستم بوجود آورند (مثل دسترسی به مکان های حافظه ای که در اختیار OS است) و یا اینکه برنامه کاربری نتواند به محل های حافظه ای کاربران دیگر و نیز داده های کاربران دیگر دسترسی داشته باشد، به سخت افزار یک bit به نام Mode bit اضافه نمودند که این بیت دو حالت صفر و یک را می تواند داشته باشد. مد صفر Monitor mode و مد یک User mode نامیده می شود. زمانی که یک برنامه اجرا می شود، در مد یک شروع به کار می کند، به محض مواجه شدن با دستوری که به عملکرد سیستم ضربه می زند (مثل خواندن و یا نوشتن، دسترسی به محل های حافظه که در اختیار OS و یا در اختیار دیگر کاربران است و loop ها) بلافاصله به مانیتور مد انتقال پیدا می کنند. در حالت Monitor mode، OS وارد عمل شده و کنترل برنامه را به عهده می گیرد. به محض اینکه آن عمل انجام گرفت دوباره به User Mode انتقال پیدا نموده و جریان طبیعی برنامه ادامه پیدا می کند.

دستوراتی که به اجرای صحیح سیستم ضربه می زند، بنام دستورات Privilege ممتاز یا شاخص نامیده می شود.

۲.۴ محافظت سخت افزاری Hardware Protection

برای اینکه سیستم عامل بطور صحیح اجرایش را ادامه دهد، باید دستوراتی که به OS ضربه می زنند را کشف نمائیم و این کشف می تواند از طریق سخت افزاری اعمال گردد. یکی از مواردی که می تواند به سیستم عامل صدمه وارد کند دسترسی به مکان های غیر مجاز از حافظه سیستم عامل و یا برنامه های دیگر است. برای جلوگیری از دسترسی غیر مجاز می توان دو رجیستر Base و Limit را برای Job های مختلف در نظر گرفت. مثلاً Job1 می تواند به محل هایی از حافظه ۱۰۰۰ تا ۱۵۰۰ در تماس باشد که در این صورت رجیستر Base آن برابر ۱۰۰۰ و Limit برابر ۵۰۰ خواهد بود که این مقدار Base Register و Limit Register توسط سیستم عامل Load می شود که این محافظت می تواند از طریق یعنی CPU اعمال شود به شرح شکل زیر:

شکل صفحه ی ۲۶ بالای صفحه

یکی دیگر از محافظت هایی که یک OS باید انجام دهد، جلوگیری از در Loop افتادن برنامه ها (در نتیجه CPU) می باشد. برای حل مشکل فوق از یک Timer استفاده می شود. خود Timer از Clock و Counter تشکیل شده است. چنانچه تصمیم گرفته شد که برنامه ها بتوانند مثلاً ۵ دقیقه از وقت CPU را به خود اختصاص دهند، در این صورت کانتر به عدد ۵*۶۰ یا ۳۰۰ ثانیه set خواهد شد و هر زمانی که Clock یک تیک (Tic) می کند یک واحد از Counter کم می شود و به محض اینکه Counter مساوی صفر شد CPU خود به خود از آن برنامه گرفته خواهد شد.

مواد دیگر استفاده از تایمر می تواند به صورت Time slice باشد. با توجه به اینکه در سیستم های Time sharing به هر Job یک زمان مشخص به نام Time slice اجازه داده می شود تا CPU را در اختیار خود بگیرد و به محض اینکه Time slice سپری شد، خود به خود CPU از آن Job گرفته می شود و به Job بعدی داده خواهد شد آن هم به اندازه ی یک Time slice و ...

System Clock –Home work

۱) کدام یک از دستورات زیر Privilege هستند؟

A) Set Value for timer
B) Turn off interrupt ✓
C) Read the Clock
D) Clear Memory
E) Switch from User mode to monitor mode ✓

۲) توضیح دهید که نیاز برای کنترل کارت ها منجر به این شد که جدا سازی مد های User و monitor به وجود یاید

۳) بعضی از سیستم های اولیه OS را محافظت می کردند توسط جایگزین نمودن آنها در حافظه که برنامه های کاربران و خود OS نمی توانست آن را تغییر دهد، دو مشکلی که می تواند به خاطر طراحی فوق بوجود آید را تعریف کنید.

۴) بیشتر کامپیوتر ها Dual Operation را سخت افزاری تهیه یا پشتیبانی نمی کنند آیا می توان یک سیستم عامل مطمئن با توجه به مطالب فوق در این سیستم ها بوجود آورد؟ توضیح دهید که هم امکان دارد و هم امکان ندارد.

۲.۵ معماری کلی سیستم General System Architecture:

تمایل به کم نمودن زمان Setup و توسعه سیستم کامپیوتر منجر به Batching of Job و Buffering و Spooling و بالاخره به Multi Programming و Time sharing شد.

اشتراک زمانی مستقیماً به تغییر بنیادی کامپیوتر اثر گذاشت و اجازه داد که سیستم عامل کنترل کامپوتر را به دست گیرد. اگر بخواهیم عملیات صحیح و ادامه دار را تهیه کنیم، کنترل بایستی باقی بماند یا طرح اجرای مد دوگانه مفهوم دستورات Privilege را که فقط در Monitor mode پشتیبانی می کند می تواند اجرا شود.

دستورات I/O و دستورات تغییر دهنده ی رجیستر های مدیریت حافظه یا تایمر همگی دستورات ممتاز در Monitor mode هستند.

همانطور که می توانید تصور کنید چندین دستور دیگر هم به عنوان Privilege کلاس بندی شده اند. به عنوان مثال دستور Halt، یک برنامه User هرگز نبایستی قادر باشد که computer را Halt کند. دستورالعمل های مجاز یا غیر مجاز نمودن وقفه ها هم ممتاز هستند به این دلیل که عمل درست تایمر و I/O بستگی به توانایی پاسخگویی صحیح به وقفه دارند.

ستور سوئیچ از مُد کاربر به مُد مانیتور ممتاز است و هر دستور تغییر بیت مُد، هم ممتاز محسوب می شود. از آنجایی که دستورات I/O ممتاز هستند و تنها توسط OS می توانند اجرا شوند. پس یک برنامه چگونه I/O User را انجام می دهد.

پاسخ این است که کاربر از مانیتور درخواست می کند تا به نام او عمل I/O را انجام دهد. پس راه حل این مساله این است: از آنجا که فقط Monitor Mode می تواند I/O را انجام بدهد User بایستی سوال کند که آیا Monitor، I/O را از طرف User انجام بدهد؟

چنین درخواستی بنام فراخوانی سیستم (یا بنام فراخوانی تابع مانیتور تا صدا زدن تابع OS) شناخته شده است. احضار فراخوانی سیستم به شیوه های گوناگون بر حسب قابلیت عمل پردازشگر مورد نظر انجام می گیرد. معمولاً فراخوانی سیستم به شکل یک تله ی OS در محل ویژه ای در بردار وقفه، صورت می گیرد. این تله می تواند توسط دستورالعمل Trap اجرا گردد.

وقتی که یک system call اجرا می شود، از دید سخت افزار همانند وقفه ی نرم افزاری تعبیر می شود. کنترل از طریف بردار وقفه به روتین سرویس در سیستم عامل می رسد و بیت مُد به بیت مانیتور مُد set می شود. روال سرویس دهنده ی فراخوانی سیستم یک قسمت از OS است.

Monitor دستورات انقطاع را امتحان می کند تا مشخص کند که system call اتفاق افتاده است. نوع سرویس درخواستی را مشخص می کند. اطلاعات اضافی که احتیاج است برای درخواست ممکن از طریق register ها و یا memory عبور کند (با Pointer ها محل های حافظه عبور کند در Register ها).

Monitor درخواست را اجرا می کند و کنترل را به دستورات بعد از system call بر می گرداند. درنتیجه برای تمام I/O یک برنامه User یک System Call اجرا کند تا درخواست را اجرا می کند و کنترل را به دستور بعد از system call می گرداند.

OS که در Monitor mode اجرا می شود چک می کند که درخواست valid است یا نه و اگر valid است درخواست I/O از OS به User بر می گردد.

ashrafi.0521o@gmail.com

هسته عامل: ۱- مدیریت پردازش ۲- هماهنگی پردازشی ۳- مدیریت حافظه ۴- مدیریت حافظه کنترلی (دستی) ۵- مدیریت فایل ۶- سیستم I/O ۷- حافظت ۸- System commander (command interpreter)

فصل سوم

ساختمان سیستم عامل

یک سیستم عامل محیطی را آماده می کند که در آن برنامه ها اجرا شوند. برای ایجاد چنین محیطی OS را بطور منطقی به قسمت های کوچکتری تقسیم نموده و یک ارتباط خوب تعریف شده ای را بین برنامه ها و قسمت ها ایجاد می کنیم. بطور کلی سیستم های عامل در سازمان دهی و آرایش فرق های زیادی با هم دارند

۳.۱ اجزای سیستم (سیستم عامل)

یک سیستم به بزرگی و پیچیدگی سیستم عامل را فقط با تقسیم بندی آن به قسمت های کوچکتر می توان تشخیص داد و هر کدام از این قسمت های کوچک می بایست مشخصاً تعریف شده باشند. یعنی ورودی ها (Input) و خروجی ها و عملکرد (Functions) تعریف شده باشد. مسلم است که همه ساختمان های OS یکسان نیستند ولی دارای اجزای مشترک هستند که در بخش های ۳.۱.۱ تا ۳.۱.۸ شرح داده شده اند.

۳.۱.۱ مدیریت پردازش Process Management

CPU برنامه های زیادی را اجرا می کند. اگر چه وظیفه ی اصلی اش اجرای برنامه های کاربران است ولی CPU برای فعالیت های دیگر سیستم نیز مورد احتیاج می باشد. این فعالیت های Process (پردازش) نامیده می شوند. به عنوان مثال یک برنامه ی در حال اجرا یک پردازش است. یک سیستم Task هر کاری که CPU نیاز دارد یک Task است مانند خواندن یا Spooling یک پردازش است. همچنین یک Multi program Job یا یک برنامه Time share یک پردازش می باشد. اما مفهوم پردازش (process) در واقع کلی تر است. همانطور که در فصل ۴ خواهیم دید این امکان وجود دارد که System Call هایی تهیه شوند که به پردازش ها اجازه دهند تا پردازش های دیگری را نیز به طور همزمان اجرا کند. در کل یک Process نیاز به منابع دارد از جمله CPU time، حافظه، فایل ها و دستگاه های I/O تا Task خودش را به انجام برساند.

ما تاکید می کنیم که یک برنامه خودش به تنهایی یک Process نیست. یک برنامه یک عنصر غیر فعال (passive) است مانند محتویات یک فایل که روی دیسک ذخیره شده است. در حالیکه یک پردازش یک عنصر Active می باشد با یک Program counter که دستور بعدی را که می بایست اجرا شود مشخص می کند. اجرای یک پردازش می بایست بطور پیوسته اجرا شود یعنی اینکه در هر نقطه از زمان حداکثر یک دستور در حال اجرا باشد(از طرف پردازش). Process یک واحد کار در یک سیستم

محسوب می شود. یک سیستم عبارت است از مجموعه ای از پردازش ها که بعضی از آنها پردازش های سیستم عامل هستند همه ی این پردازش ها این توانایی را دارند که به طور همزمان توسط استفاده مشترک از CPU اجرا شوند.

سیستم عامل برای فعالیت های زیر در رابطه با مدیریت پردازش مسئول است:

✓ الف) ایجاد و حذف پردازش های سیستم و پردازش های کاربران

✓ ب) به تعویق در آوردن و آزاد نمودن Process ها

✓ ج) تهیه ی مکانیزمی برای اجرای همزمان پردازش ها

✓ د) تهیه ی مکانیزمی برای ارتباط بین پردازش ها

✓ ه) تهیه ی مکانیزمی برای برطرف نمودن بن بست یا deadlocks

تکنیک های مدیریت پردازش در فصل های ۴ و ۵ و ۶ مفصلاً بحث خواهد شد.

۳.۱.۲ مدیریت حافظه Memory Management:

حافظه کامپیوتر مرکز اصلی عملیات کامپیوتر های مدرن می باشد. همانطور که می دانید حافظه یک آرایه ی بزرگ از کلمات (word) یا بایت ها می باشد که هر کدام آدرس مخصوص به خود را دارند و همچنین می دانید حافظه یک محل ذخیره ی اطلاعات است که به طور صریح قابل دسترسی هستند (توسط CPU و دستگاه های I/O). CPU دستورات را از حافظه اصلی در زمان اجرای برنامه می خواند و همچنین می تواند اطلاعات را بر روی حافظه اصلی بنویسد. یعنی حافظه اصلی تنها حافظه ای است CPU قادر است به طور مستقیم آدرس دهی نماید.

برای بهبود بخشیدن به کارآیی CPU و سرعت جواب دادن کامپیوتر به کاربران، ما می بایستی چندین برنامه را در حافظه نگهداری نمائیم. طرح های مختلفی برای مدیریت حافظه وجود دارد و موثر بودن الگوریتم ها و طرح ها به فاکتورهای مختلفی به خصوص به سخت افزار آن سیستم بستگی دارد. سیستم عامل برای فعالیت های زیر در ارتباط با مدیریت حافظه مسئول است:

الف) بداند که کدام قسمت از حافظه استفاده شده است و توسط کدام برنامه

ب) زمانی که فضای حافظه آزاد است تصمیم بگیرد که کدام پردازش بایستی در حافظه بارگیری شود.

ج) تخصیص دادن و آزاد نمودن حافظه، زمانی که حافظه نیاز است.

۳.۱.۳ مدیریت حافظه دوم Secondary Memory Management:

برنامه ها برای اجرا نیاز به داده ها دارند که این داده ها بایستی در حافظه اصلی موجود باشند تا CPU بتواند در هنگام اجرا از آنها استفاده نماید. از آنجا که حافظه های اصلی بسیار کوچک تر هستند تا همه ی برنامه ها و داده ها را خودشان نگهداری کنند، لذا سیستم کامپیوتر بایستی حافظه دوم را تهیه نماید به این منظور که حافظه اصلی را پشتیبانی نماید.

در بیشتر سیستم های کامپیوتری مدرن از دیسک به عنوان دستگاه Storage (انباره online) هم برای برنامه و هم برای داده استفاده می کنند.

بیشتر برنامه ها شامل کامپایلر ها، اسمبلرها، روش های sort، editor ها (text editor) و... هستند که روی دیسک ذخیره می شوند تا در زمان مناسب داخل حافظه اصلی load گردند یعنی از دیسک هم به عنوان مبداء و هم به عنوان مقصد پردازش ها استفاده می شود. پس مدیریت صحیح انباره دیسک برای سیستم کامپیوتر از اهمیت بسیار بالایی برخوردار است.

OS برای فعالیت های زیر در ارتباط با مدیریت دیسک SMM (Secondary Memory Management) مسئول است:

الف) مدیریت فضای خالی	free space management
ب) اختصاص دادن انباره	Storage allocation
ج) زمان بندی دیسک	Disk Scheduling

۳.۱.۴ I/O System Management:

یکی از اهداف OS این است که سختی ها یا مشکلات دستگاه های سخت افزاری بخصوص دستگاه I/O را از کاربر پنهان نماید. زیر سیستم های I/O عبارتند از:

الف) مولفه مدیریت حافظه شامل بافرگیری، نهایی کردن و اسپولینگ	A buffer-Caching System
ب) واسطه ی درایو وسیله عمومی	A General Device Driver Interface
ج) درایورهای دستگاه های سخت افزاری	Drivers for Specific Hardware Devices

فقط درایور دستگاه خصوصیات و مشکلات دستگاه ویژه ای را که به آن نسبت داده شده است می دانند همانطور که در فصل ۲ در مورد ابنکه چطور Device Driver ها در ایجاد سیستم های I/O مفید مورد استفاده قرار می گیرند بحث کردیم در فصل ۹ چگونگی استفاده از یک دستگاه بخصوص را که به طور موثر مدیریت می شوند را مورد بررسی قرار می دهیم

۳.۱.۵ مدیریت فایل ها File management:

مدیریت فایل یکی از مهمترین قسمت های یک OS است. کامپیوتر های می توانند اطلاعات را در چندین فرم مختلف به طور فیزیکی ذخیره نمایند. دیسک مغناطیسی، نوار مغناطیسی و CD ها بیشترین فرم های متداول هستند. هر کدام از این دستگاه ها مختصات وسازمان فیزیکی مربوط به خود را دارند. لذا برای راحتی استفاده از کامپیوتر OS ذخیره سازی اطلاعات را طور منطقی همشکل انجام می دهد. یعنی اینکه OS خصوصیات فیزیکی دستگاه های ذخیره سازی اش را مطلق سازی می کند (Abstraction) تا یک واحد ذخیره سازی منطقی یعنی فایل را تعریف نماید.

سیستم عامل در مورد فعالیت ها ی زیردر ارتباط با مدیریت فایل مسئول است

الف) ایجاد و حذف فایل ها

ب) ایجاد و حذف دایرکتوری ها

ج) پشتیبانی و حمایت از اعمال فایل ها و دایرکتوری ها (از قبیل حذف و اضافه و...)

د) نگاشت نمودن (Map) به حافظه ی دوم و یا دیسک یعنی تعیین فایل های ورودی و خروجی در زمان اجرای برنامه

ه) پشتیبانی گرفتن (Backup) از فایل ها روی دستگاه های ذخیره ی با دوام مثل نوار

۳.۱.۶ محافظت Protection:

پردازش های مختلف در یک سیستم باید در مقابل فعالیت های دیگر سیستم محافظت شود. برای این منظور مکانیزم های تهیه شده است تا اطمینان دهد که فایل ها، قطعات حافظه، CPU و منابع دیگر فقط توسط آن پردازش هایی اجرای می شوند که مجوز صحیح را از OS بدست آورده باشند. به عنوان مثال سخت افزار آدرس بندی حافظه (Memory Addressing) اطمینان می دهد یک Process فقط در فضای آدرس متعلق به خودش می تواند اجرا شود و تیز تایمر ضمانت می کند که هیچ پردازشی نمی تواند کنترل CPU را برای ابد به دست آورد و بالاخره User ها در مورد دستورات Privilege نمی توانند به دستگاه های I/O دسترسی مستقیم داشته باشند، مگر اینکه OS اجرای آن را به عهده بگیرد و چک کند که آیا دستورات مجاز است یا نه. لذا یک سیستم محافظت شده، وسیله ای را تهیه می کند تا استفاده مجاز و غیر مجاز را تشخیص دهد و بر علیه استفاده ی غیر مجاز از خود دفاع کند که در فصل ۱۱ مورد بحث قرار می گیرد.

۳.۱.۷ Networking:

۱- Distributed System یا سیستم توزیع شده، مجموعه ای از پردازش ها است که حافظه، وسایل جانبی یا ساعت را به طور مشترک مورد استفاده قرار نمی دهند و در عوض هر پردازنده (Processor) در CPU حافظه داخلی و ساعت خودش را دارد و این پردازنده ها از طریق خطوط ارتباطی مختلف با هم ارتباط دارند مثل گذرگاه ها یا خط های تلفن. همانطور که می دانید پردازنده ها در یک سیستم گسترده (Distributed) از نظر اندازه و عمل تغییر می کنند و آنها در سیستم از طریق Communication Network (شبکه ارتباطی) به یکدیگر متصل هستند. هم چنان که می دانید یک سیستم Distributed دسترسی کاربران را به منابع مختلف فراهم می کند و این دسترسی به منابع مشترک سرعت محاسبات و همچنین میزان اعتماد را افزایش می دهد. معمولاً OS دسترسی به شبکه را به عنوان یک نوع از دسترسی به فایل به جزئیات شبکه ای تعمیم می دهد.

۳.۱.۸ مفسر دستورات سیستم Command Interpreter System : با همین سیستم ها کار می کنند.

یکی از مهم ترین برنامه های سیستم، مفسر دستورات (C.I) است. بعضی از OS ها مخصوصاً آن دسته که روی Micro Computer هستند C.I را در هسته ی مرکزی یا Kernel مرکزی دایر کرده اند.

ولی بیشتر OS های Mainframe، CI را به عنوان یک برنامه مخصوص در نظر گرفته اند که در آغاز یک یا برقرار شدن اولین ارتباط اجرا می شود. مثل Unix & MS DOS. بیشتر مفسر های قوی و پیچیده در Unix در قسمت Shell سیستم وجود دارد و عملکرد آنها کاملاً ساده است.

مثال) حکم فرمان بعدی را خوانده و آن را اجرا کن

Go the next command and statement and execute it

احکام فرمان خود به ایجاد پردازش و مدیریت آن، اعمال I/O، مدیریت ذخیره ی ثانویه، مدیریت حافظه اصلی، دستیابی سیستم فایل، حفاظت و اعمال شبکه می پردازد.

یک سیستم عامل محیطی برای اجرای برنامه ها فراهم می کند. همچنین OS بعضی از سرویس ها را برای برنامه ها و برای کاربران (Users) تهیه می نماید. باید توجه داشت که این سرویس ها از یک OS به OS دیگر فرق خواهند داشت. اما بعضی سرویس های مشابه در سیستم عامل ها می توانند به صورت زیر باشند. با توجه به اینکه سرویس ها برای راحتی برنامه نویسان تهیه شده اند می توانند نوشتن برنامه ها را برای کاربران ساده کنند.

الف) اجرای برنامه

سیستم می بایست قادر باشد تا یک برنامه را به داخل حافظه (بازیابی) نموده و آن را اجرا نماید و باید قادر باشد تا اجرایش را به چه به صورت نرمال (Normal) و چه به صورت غیر نرمال (Error) خاتمه دهد.

ب) عملیات I/O

یک برنامه در حال اجرا ممکن است به I/O احتیاج داشته باشد و I/O ممکن است یک فایل یا یک دستگاه I/O را در بر گیرد برای دستگاه بخصوص توابع خاصی ممکن است لازم باشد (مثل Rewind یا برگشت دستگاه نوار یا خالی نمودن Screen) از آنجا که کاربر نمی تواند عملیات I/O را مستقیماً انجام دهد (برای حصول بازدهی و حفاظت) این وظیفه ی OS است که این سرویس ها را تهیه نماید.

ج) دستکاری فایل ها

این واضح است که برنامه ها نیاز دارند از روی فایل ها (Read) خوانده و یا روی فایل ها نوشته (Write) و یا فایل جدیدی را ایجاد (Create a new file) کنند و یا فایل ها را حذف کنند (Delete File) و این وظیفه ی OS است که این امکانات را فراهم نماید.

د) ارتباطات

دو شرایط بخصوصی یک پردازش (Process) نیاز دارد تا اطلاعاتش را با Process های دیگر تبادل نماید. دو روش اصلی برای اجرای این چنین ارتباطات وجود دارد. ۱- ارتباط بین پردازش های در حال اجرا روی یک سیستم اتفاق می افتد ۲- بین پردازش های در حال اجرا در سیستم های کامپیوتری متفاوت که توسط شبکه ی ارتباطی به هم متصل شده اند می باشد. و این ارتباط ممکن است از طریق حافظه اشتراکی و یا توسط تبادل پیغام ها انجام گیرد و OS این چنین سرویس هایی را نیز تهیه می کند.

ه) کشف خطاهای Error

OS می بایست پیوسته از error ها یا خطاهایی که ممکن است در کامپیوتر اتفاق بیافتد آگاه باشد. این خطاها ممکن است در سخت افزار CPU و حافظه اتفاق بیافتد و یا در دستگاه های I/O. لذا OS برای هر نوع error سرویسی را تهیه می کند که آن سرویس می تواند عکس العمل مناسب را در برابر خطاها نشان دهد.

سرویس های دیگری وجود دارد که برای اطمینان دادن به عملیات کارآمد سیستم برای خودش به اجرا در می آید:

1. تخصیص منابع
2. حسابداری
3. محافظت

الف) تخصیص منابع

وقتی که چندین User با Job های مختلف هم زمان اجرای می شوند منابع می بایست به هرکدام از آنها تخصیص داده شود و این منابع توسط OS برای استفاده مشترک از طرف Job ها مدیریت می شوند.

ب) حسابداری Accounting

مشخص می کند که کدام User چه مقدار و چه نوع از منابع سیستم کامپیوتر را مورد استفاده قرار داده است. ثبت این رکورد ها ممکن است جهت استفاده ی حسابداری و یا برای استفاده ی آماری باشد. پس از سرویس های دیگر OS تهیه امکاناتی برای ثبت رکوردهای حسابداری می باشد.

ج) محافظت Protection

صاحبان اطلاعات ذخیره شده در یک سیستم کامپیوتری چندیدن نفرند. باید استفاده از آن اطلاعات را توسط دیگران کنترل نمایند. در واقع محافظت همه ی پارامترهایی را که به System Call داده می شود چک می نماید تا مطمئن گردد دسترسی به منابع با مجوز قبلی صورت گرفته باشد و کنترل شده باشد. همچنین محافظت سیستم از خارج یا بیرون نیز مهم است. چنین ایمنی با محافظت با استفاده از Password یا کلمه ی عبور بخصوص به اجرا در می آید تا اجازه دسترسی به منابع را بدهد.

۳.۳ فراخوانی های سیستم System Call

سیستم call ها واسطه ای (interface) بین یک برنامه در حال اجرا و OS هستند. این فراخوانی های سیستم کلاً به صورت دستورات اسمبلی موجود هستند و در کتابچه ی راهنمای زبان اسمبلی موجود هستند. در بعضی سیستم ها به System Call اجازه داده می شود تا مستقیماً از برنامه به زبان سطح بالا صدا زده شود. در چنین حالت ها System Call ها به صورت توابع از قبل تعریف شده و یا همان Call subroutine ها وجود دارند. چندین زبان برنامه نویسی مثل C جایگزین زبان اسمبلی برای برنامه نویس های سیستم شده اند. این زبان ها به System Call ها اجازه می دهند تا مستقیماً فراخوانده شوند و همچنین بعضی سیستم های دیگر مانند پاسکال و فرترن دارای چنین امکاناتی هستند. به عنوان مثال اینکه چطور System Call ها مورد استفاده قرار می گیرند نوشتن برنامه ساده زیر را در نظر بگیرید که عبارتست از خواندن data از یک فایل و کپی نمودن آن به فایل دیگر. اولین چیزی که برنامه نیاز دارد نام دو فایل است. یعنی فایل ورودی و فایل خروجی برنامه. می تواند از کاربر نام دو فایل را سوال نماید. در یک سیستم interactive این عمل نیاز به یک سری از System Call ها دارد. یعنی اول نوشتن یک پیغام درخواست نام فایل روی صفحه و سپس تایپ نمودن نام فایل ها از طریق صفحه کلید (Keyboard) و...

کاربران هیچ یک از جزئیات فوق را نمی بینند و سیستم های پشتیبانی Runtime یک میانجی بسیار ساده تر را برای زبان های برنامکه نویسی تهیه می کنند. به عنوان مثال یک جمله ی write در پاسکال و یا فرترن به صورت یک System Call به Routine پشتیبانی Runtime کامپایل یا ترجمه می شود تا System Call مورد نیاز را را صادر نماید و در انتها کنترل برنامه به User برخواهد گشت. در نتیجه بیشترین جزئیات ارتباط با سیستم از برنامه نویسان پنهان می شود که توسط کامپایلر و Routine های پشتیبانی Runtime انجام می گردد. System Call ها به روش های مختلف اتفاق می افتند که بستگی کامل به کامپیوتر دارد.

سه روش کلی برای انتقال دادن پارامترها به سیستم عامل مورد استفاده قرار می گیرند. ساده ترین روش این است که پارامترها را با استفاده از رجیستر ها منتقل کرد و در بعضی حالات زمانی که پارامترهای بیشتری از رجیسترها ممکن است وجود داشته باشد به صورت یک بلاک و یا جدول در حافظه ذخیره می شوند و آدرس این بلاک یا جدول به عنوان پارامتر به یک رجیستر انتقال داده می شود و همچنین ممکن است پارامترها داخل stack توسط برنامه push شده و توسط POP ،OS شوند. بیشتر OS ها روش های بلاک یا Stack را ترجیح می دهند چون آنها نامحدود و یا طول پارامترهای منتقل شونده را محدود نمی کنند. System Call ها به ۵ قسمت عمده تقسیم می شوند:

۱. Process Control system call
۲. File manipulation System Call
۳. Device manipulation System Call
۴. Information Statue System Call
۵. Communication System Call

که در بخش های ۳.۳.۱ الی ۳.۳.۵ به طور مختصر نوع System Call ها را بحث خواهیم نمود. البته بحث ما ممکن است مقداری سطحی به نظر برسد و مفهوم نباشد زیرا اکثر این فراخوانی نهای قبلی، مفاهیم و توابع مورد بحث در فصل های بعد توضیح داده می شوند. شکل زیر نوع System Call هایی را که به طور نرمال توسط OS تهیه می شوند، نشان می دهد.

1. Process Control System Call
   - End, Abort
   - Load, Execute
   - Create Process, Terminate Process
   - Get Process Attributes, Set Process Attributes
   - Wait for time
   - Wait Event Signal Event
   - Allocate and free Memory
2. File Manipulation System Call
   - Create file, Delete file
   - Open, Close
   - Read, Write, Reposition
   - Get file attributes, set file attributes

3. Device Manipulation System Call
    - Request Device, Release Device
    - Read, Write, Reposition
    - Get device attributes, Set device attributes
    - Logically attach or detach devices
4. Information maintenance system call
    - Get time or date, Set time or date
    - File or device attributes
5. Communication system call
    - Create, delete communications connectives
    - Send receive message
    - Transfer status information
    - Attach or detach, remove device

۳.۳.۱ Process and Job Control:

یک برنامه ی اجرایی احتیاج دارد تا قادر باشد که اجرایش را چه به صورت نرمال(end) و چه به صورت غیرنرمال (abort) پایان دهد و اگر یک System Call صادر شود و اجرای برنامه را بصورت غیر نرمال خاتمه دهد یا اگر یک برنامه داری مشکلی شود و باعث error trap شود، یک روتوشت از حافظه گرفته می شود و پیغام خطا صادر می شود. تحت شرایط غیرنرمال سیستم عامل باید کنترل را به مفسر دستورات (command interpreter) انتقال دهد آنگاه command interpreter به سادگی یا دستور بعدی ادامه می دهد و فرض خواهد کرد که User پیغام مناسبی را در جواب هر خطا صادر نموده است. در یک سیستم دسته ای CI معمولاً تمام Job های بعدی را شروع می کند. بعضی از سیستم ها به کارت های کنترل اجازه می دهند تا در صورتی که error اتفاق بیافتد، عملیات بازیافت بخصوصی را انجام دهد. اگر یک برنامه یک اشتباه را در هنگام اجرایش کشف کند و بخواد تا به طور غیرنرمال خاتمه پیدا کند، ممکن است بخواهد تا سطح خطا را تعریف کند. بیشتر اشتباهات جدی توسط پارامترهای اشتباه سطح بالا نشان داده می شوند، پس ممکن است که اتمام نرمال و غیرنرمال توسط تعریف نرمال به عنوان خطا در سطح 0 تعبیر شوند. مفسر زبان یا برنامه ی بعدی، می تواند سطح خطا را برای تعیین اتوماتیک عمل بعدی، به؟؟؟؟؟.

یک Process یا یک Job در حال اجرا ممکن است بخواهد که برنامه ی دیگری را load واجرا کند. مفسر فرمان این اجازه را می دهد که برنامه ای را اجرا نماید. توسط یک دستور User (کاربر)، کلیک نمودن ماوس و یا دستور Batch. یک سوال جالب این است که وقتی برنامه ی load شده خاتمه می یابد کنترل به کجا بر می گردد؟ این سوال بستگی دارد به اینکه برنامه ی موجود اجرایش تمام شده است یا ذخیره شده است یا اجازه دارد که به اجرایش ادامه دهد. اگر کنترل به برنامه ی موجود برگردد تا وقتی که برنامه خاتمه پیدا می کند باید تصویر حافظه ی برنامه ی موجود را ذخیره کنیم و لذا بطور موثر مکانیزمی ایجاد نموده ایم که مالتی پروگرام هستند. غالباً یک فراخوانی سیستم ویژه ی این کار موجود است. (Create process or submit Job) اگر کار یا پردازش جدیدی ایجاد نمائیم یا حتی شاید مجموعه ای از کارها یا پردازش ها، ما باید قادر باشیم تا اجرای آن ها را کنترل کنیم.

خالی

عامل Multi tasking است) مفسر فرمان می تواند اجرایش را ادامه دهد در حالی که برنامه ی دیگر در حال اجرا شدن است. برای شروع شدن یک Process جدید Shell یک System Call را اجرا می کند. سپس برنامه ی انتخاب شده داخل حافظه Load می شود. بستگی به روشی که دستور صادر شده است دارد که shell یا منتظر اتمام Process می ماند و یا Process را در پس زمینه اجرا می کند. در حال دوم Shell دستور دیگری را درخواست می کند. وقتی که یک Process در Background اجرا می شود، نمی تواند مستقیماً از user توسط صفحه کلید ورودی دریافت کند اما از آنجا که shell همچنان انتظار دریافت Input را دارد، I/O از طریق فایل ها انجام می شود. در همین حال User آزاد است تا از shell بخواهد که برنامه ی دیگر را اجرا کند، پیشرفت Process در حال اجرا را مشاهد کند و حق تقدم برنامه را تغییر دهد و .... وقتی که Process انجام شد یک Systemcall را اجرا می کند تا متوقف شود و کد وضعیت 0 و یا کد خطای غیر صفر را به برنامه ی صداکننده اش بر می گرداند. این کد وضعیت در اختیار پوسته و یا سایر برنامه ها قرار می گیرد.

۳.۳.۲ پردازش فایل File Manipulation:

سیستم فایل در فصل ۱۵ با جزئیات مفصل تر بحث خواهد شد. اگرچه ما می توانیم چندین System Call معمولی را در رابطه با سیستم فایل ها شناسایی کنیم اما نخست باید بتوانیم فایل ها را ایجاد (create) و یا حذف (delete) نمائیم. هر یک از آنها نام فایل و شاید بعضی صفات ویژه ی فایل را لازم دارند. وقتی که فایل ایجاد شد، لازم است آن را Open نموده و مورد استفاده قرار داد.

همچنین ممکن است ما Read, write یا Reposition (مثلاً برگرداندن به انتهای فایل) نمائیم و بالاخره نیاز داریم که فایل را Close کنیم تا بگوییم دیگر به آن احتیاج نداریم.

اگر یک ساختمان dir برای سازماندهی فایل داشته باشیم، به مجموعه ای از اعمال یکسان برای فهرست ها نیازمندیم. به علاوه برای فایل ها و دایرکتوری ها ما احتیاج داریم که مقدار صفت های مختلف را شناسایی کنیم و یا آنها را تغییر دهیم. صفات ویژه شامل نام فایل، نوع فایل، کد محافظت، اطلاعات حسابداری و غیره هستند. دو System Call برای این عمل مورد نیاز است:

- Set File Attributes
- Get File Attributes

۳.۳.۳ سیستم مدیریت دستگاه Device Management:

یک برنامه ی در حال اجرا ممکن است به منابع اضافی احتیاج داشته باشد تا به اجرایش ادامه دهد. منابع اضافی مثل حافظه بیشتر، دستگاه های نوار، دسترسی به فایل و غیره هستند. اگر منابع موجود باشند، می توانند استفاده شوند و کنترل می تواند به برنامه ی User برگردد و برنامه ادامه یابد. در غیر این صورت برنامه باید منتظر بماند تا منابع کافی در اختیار قرار بگیرند.

فایل ها می توانند به عنوان دستگاه های مجازی یا مطلق (Abstract) در نظر گرفته شوند، لذا بیشتر System Call های فایل ها برای دستگاه ها نیز مورد نیاز است. اگر چندین User وجود داشته باشند، اول دستگاه مورد نظر را Request کنیم تا از استفاده ی انحصاری آن اطمینان حاصل کنیم و در خاتمه دستگاه را Release کنیم. این شبیه به توابع System Call های Open/Close فایل است. به محض اینکه دستگاه درخواست و به ما اختصاص داده شد، ما می توانیم Read/Write و احتمالاً Reposition کنیم. همانند فایل های معمولی در حقیقت شباهت بین دستگاه های I/O و فایل های آن قدر زیاد است که بیشتر سیستم های عامل نظیر Unix و MSDOS آنها را ادغام نموده، در یک ساختمان ترکیب File/Device قرار داده اند. در این حالت دستگاه های I/O توسط نام فایل های مخصوص شناسایی می شوند.

۳.۳.۴ نگهداری اطلاعات Information Maintenance:

بیشتر System Call ها به منظور انتقال اطلاعات بین برنامه User و سیستم عامل وجود دارند. به عنوان مثال بیشتر سیستم ها برای برگرداندن زمان جاری و تاریخ یک System Call دارند.

بعضی دیگر از System Call ها ممکن است اطلاعاتی در مورد سیستم برگردانند. مثلاً تعداد User های در حال کار، شماره Version سیستم عامل، مقدار حافظه آزاد یا فضای دیسک و غیره.

به علاوه سیستم عامل اطلاعات درباره ی تمام Job ها و Process ها را نگهداری می کند و System Call هایی وجود دارد که آنها را reset می کند. در بخش ۴.۱.۳ بحث خواهیم کرد که چه اطلاعاتی به طور نرمال نگهداری می شوند.

۳.۳.۵ ارتباط Communication:

دو مدل ارتباطی مشترک وجود دارد. در مدل عبور پیغام از طریق وسیله ی ارتباطی فرا پردازشی که توسط سیستم عامل تهیه می شود، اطلاعات مبادله می شوند. قبل از اینکه ارتباط انجام شود یک اتصال باید برقرار شود. نام بقیه ی ارتباط دهندگان می بایست شناخته شده باشد که می تواند یک Process دیگر روی همان CPU باشد یا یک Process روی یک کامپیوتر دیگر در شبکه ی ارتباطی باشد. هر کامپیوتر در شبکه یک نام میزبان دارد که توسط آن شناخته می شود. بطور مشابه هر Process یک Process name دارد که به یک شناسه ی معادل ترجمه می شود که توسط آن سیستم عامل می تواند به آن رجوع کند. Get Host ID و همچنین Get Process ID، System Call هایی هستند که عمل ترجمه را انجام می دهند و به Open connection و Close Connection بخصوص برحسب مدل ارتباطی سیستم عبور داده می شوند. Recipient Process (دریافت کننده) معمولاً بایستی آمادگی خود را برای ارتباط از طریق SC یک Accept Connection Call اعلام نماید. اکثر Process هایی که اتصالات دریافت کننده خواهد بود شبح های منظوره هستند و جزو برنامه های سیستمی می باشند. آنها S.C Wait For Connection را اجرا می کنند و وقتی که اتصال برقرار می شود به جریان می افتند و بیدار می شوند. منبع ارتباط معروف به مشتری و دریافت کننده.

Daemon، معروف به server سپس پیغام ها را توسط System Call های read message و write message مبادله می کنند و close connection call به این ارتباط خاتمه می دهد.

در مدل حافظه مشترک،Process ها System Call های Map Memory را به کار می برند تا دسترسی به مناطقی از حافظه که مربوط به Process های دیگر است را بدست آورند. به خاطر بیاورید که معمولاً سیستم عامل سعی می کند یک پردازش را از دسترسی به حافظه ی پردازش منع کند. Shared Memory نیاز دارد که یا چند Process موافق باشند که این محدودیت را حذف کنند. سپس آنها توسط خواندن و نوشتن data در محیط مشترک،اطلاعات را مبادله می کنند. فرم data و محل آنها توسط این Process ها معین می شوند و تحت کنترل سیستم عامل نیستند. همچنین process ها مسئولند اطمینان دهند که همه در یک محل همزمان نمی نویسند. در مورد این چنین مکانیزمی در فصل ۵ بحث خواهد شد.

هر دو این روش ها در سیستم عامل وجود دارند و بعضی از سیستم ها هر دو را Implement می کنند. مدل Message passing زمانی که مقدار کمی Data نیاز است که مبادله شوند مفید هستند. زیرا لازم نیست از هیچ تناقضی اجتناب شود. همچنین ارتباط فراکامپیوتری راحت تر پیاده سازی می شود. در عوض مدل shared memory بالاترین سرعت و راحتی ارتباط را اجازه می دهد. زیرا می تواند در سرعت های حافظه انجام شود اما مشکلاتی در ارتباط با سنکرون کردن و محافظت دارد.

۳.۴ برنامه های سیستم System Programs

مزیت دیگر یک OS مدرن وجود مجموعه ای از برنامه های سیستم است. برنامه های سیستم یک محیط بسیار راحت را برای اجرا و توسعه برنامه تهیه می نماید. آنها می توانند به چندین گروه تقسیم شوند:

الف) دستکاری فایل: این برنامه ها برای ایجاد، حذف، کپی، دوباره نام گذاری کردن rename، چاپ یا print، لیست گیری و کلاً برای تغییرات فایل ها و دایرکتوریها به کار می روند.

ب) اطلاعات وضعیت سیستم: بعضی از برنامه ها در سیستم عامل، تاریخ، زمان، مقدار حافظه موجود در مقدار فضای خالی دیسک، تعداد کاربران و یا اطلاعات مشابه را سوال می کنند. این اطلاعات سپس از طرف OS جواب داده می شوند که روی ترمینال یا روی فایل و یا دستگاه های دیگر چاپ می شوند.

ج) اصلاح فایل File modification (تغیر فایل ها): چندین ویرایشگر در سیستم عامل ممکن است وجود داشته باشند که برای ایجاد و تغییر محتویات فایل که روی دیسک یا نوار ذخیره شده اند به کار می روند.

د) پشتیبانی از زبان های برنامه نویسی: کامپایلر ها، اسمبلرها و مفسر ها برای زبان های برنامه نویسی معمولی مثل فرترن، کوبول، بیسیک، C و غیره برای سیستم عامل تهیه می شوند.

ه) بارگیری و اجرای برنامه ها: زمانی که یک برنامه کامپال شد بایستی در حافظه load شود و بتواند به اجرا در بیاید. لذا سیستم بایستی loader ها و Linkage Editor ها را تهیه نماید.

و) ارتباطات: این برنامه مکانیزمی برای ایجاد و ارتباط بین process ها و user های مختلف کامپیوتر تهیه می نماید. آنها به user اجازه می دهند تا پیغام را به ترمینال های user دیگر بفرستند . برای پیغام های زیادتر از پست الکترونیک email استفاده کرده را از یک ماشین به ماشی دیگر از راه دور انتقال می دهند.

ز) برنامه های کاربردی Application Programming: بیشتر سیستم عامل ها به برنامه های مجهز هستند تا عملیات مشترک را به انجام برساند و به کاربران کمک نمایند. این برنامه ها شامل (compiler compilers) (مراحل یک کامپایلر را کامپایل می کند) ،Text formatter، سیستم های پایگاه داده و صفحات گسترده (Spread sheets) و غیره می باشند. شاید مهم ترین برنامه سیستم برای یک سیستم عامل Command Interpreter (مفسر دستور) است که عمل اصلی آن این است که دستور بعدی را که user داده است را گرفته و آن را اجرا نماید. بیشتر دستورهایی که در این سطح داده می شوند، فایل ها را دستکاری می نمایند (Manipulate) یعنی ایجاد، حذف، لیست، چاپ، کپی، اجرا و غیره. روش کلی برای پیاده سازی این دستورات وجود دارد که در قسمت شرح سرویس های سیستم عامل به آن می پردازیم.

۳.۵ ساختمان سیستم عامل System Structure

یک سیستم به بزرگی و پیچدگی سیستم عامل می بایست، با دقت ساخته شود. یک راه معمولی این است که Task یا وظایف آن را به قسمت های کوچتر تقسیم نمود و هر کدام از این قسمت ها می بایست ورودی و خروجی مشخص داشته باشند. ما تا کنون به طور خلاصه اجزای مشترک سیستم عامل را تعریف نمودیم. در این قسمت روشی را که این اجزا به همدیگر متصل می شوند و یک Kernel یا هسته را به وجود می آورند، توضیح می دهیم.

۳.۵.۱ ساختمان داده

شکل صفحه ۵۱

سیستم های تجارتی بسیار زیادی وجود دارد که ساختمان خوب تعریف شده ای ندارند. بسیاری از این سیستم عامل های تخست به عنوان OS کوچک، ساده و محدود ایجاد می شوند و بعد فراتر از میدان عمل خود رشد پیدا می کنند. به عنوان مثال MS DOS پرفروش ترین OS ماکرو کامپیوتر ها در اصل توسط چند نفر طراحی شد که هیچ اعتقادی نداشتند که روزی بسیار مشهور خواهند شد و به این خاطر نوشته شد که بیشترین عملکرد را در حداقل فضای خالی نیاز می باشد. لذا با دقت به قسمت های کوچکتر تقسیم بندی نشده

است. شکل بالای صفحه، ساختمان آن را نشان می دهد اگر چه MS DOS دارای مقداری Structure یا ساختمان است ولی مباحثی آن یعنی سطوح علمی آن به خوبی جدا نشده اند به عنوان مثال Application program می توانند مستقیماً به Routine ها Rom Bios DD دسترسی داشته باشند به این معنی که مستقیماً نمی توانند روی دستگاه های دیسک و Disk play بنویسند البته MS DOS توسط سخت افزاری که روی آن اجرا می شود محدود گشته است یعنی پردازنده ی Intel 8086 . چون این پردازنده که DOS بر روی آن نوشته شده فاقد مُد دوگانه و حفاظت سخت افزاری بود، طراحان چاره ای جز در دسترس گذاشتن سخت افزار نداشتند.

۳.۵.۲ روش لایه ای شکل در صفحه ۶۱

شکل صفحه ۵۲

با داشتن پشتیبانی درست سخت افزاری، سیستم عامل می تواند به قسمت های کوچکتر شکسته شود و تقسیم بندی شود. این مطلب اجازه می دهد که سیستم عامل کنترل بسیار بیشتری روی کامپیوتر و Application هایی که از آن استفاده می کنند اعمال نماید. همچنین به طراحان Implementers آزادی عملی بیشتری می دهد تا بتوانند قسمت های داخلی سیستم را در صورت نیاز تغییر دهند. لذا modularization یا قسمت بندی (تقسیم بندی سیستم) می تواند به روش های مختلفی انجام شود. متداولترین روش، روش لایه ای است که عبارت است از شکستن سیستم عامل به چند لایه (سطح یا Level) که هر کدام از روی لایه های پائین تر ساخته شده است. آخرین لایه یعنی لایه ی صفر سخت افزاری است و بالاترین لایه، لایه ی Application Program یا user خواهد بود. مهم ترین فایده روش لایه ای Modularity یا خاصیت پیمانه ای است. یعنی لایه ها طوری انتخاب می شوند که هر کدام فقط عملیات و سرویس های لایه های سطح پائین تر از خود را بکار ببرند. این روش می تواند غلط گیری و صحت سیستم را راحت تر نمایند. به این معنی که اولین سطح می تواند غلط گیری شود، بدون اینکه نگران بقیه سیستم باشد. زیرا طبق تعریف هر لایه فقط سخت افزار پایه را برای تحقق پذیری اهدافش بکار می برد و به محض اینکه سطح اول غلط گیری شد؛ عملکرد آن می تواند صحیح فرض شود در حالی که سطح دوم در حال اجراست و به همین ترتیب چنانچه یک اشتباه در طی غلط گیری یک سطح بخصوص بوجود بیاید، معلوم است که اشتباه در آن سطح رخ داده است. به این خاطر که سطوح زیرین قبلاً غلط گیری شده اند. هر لایه با استفاده از آن عملیاتی که توسط لایه های سطح پائین تر تهیه شده اند، اجرا می شود. همچنین هر لایه نیاز ندارد که بداند چطور این عملیات پیاده سازی شده اند. بلکه فقط احتیاج دارد که بداند آن اعمال چه می کنند. اولین سیستمی که با استفاده از روش لایه ای طراحی شد، سیستم THE است که در شش (6) لایه تعریف شده است و در شکل صفحه قبل نشان داده شده است. مشکل اساسی روش لایه ای عبارت است از تعریف مناسب لایه ها یا سطوح. از آنجا که یک سطح یا لایه تنها می تواند لایه های سطح پائین تر را بکار ببرد، برنامه ریزی دقیقی لازم است. به عنوان مثال Device Driver ها برای پشتیبانی فضای خالی روی دیسک بایستی در سطح پائین تری از روال های مدیریت حافظه قرار گیرد زیرا مدیریت حافظه نیازمند به کارگیری فضای ذخیره پشتیبان می باشد.

ممکن است موارد دیگر خیلی آشکار نباشد، مثلاً دستگاه دیسک بطور نرمال بایستی از زمان بندی CPU scheduler بالاتر باشد، به خاطر این که دستگاه دیسک یا Disk Driver ممکن است برای I/O منتظر بماند و در این حال CPU می تواند دوباره برنامه ریزی شود و پردازش دیگری را اجرا نماید.

۳۶ ماشین مجازی Virtual Machine

توضیحات صفحه 28

بطور مفهومی یک سیستم کامپیوتر از لایه هایی ساخته شده است. سخت افزار در تمام این سیستم ها در پائین ترین سطح قرار دارد. Kernel که در سطح بعدی اجرا شده دستورات سخت افزاری را بکار می برد تا یک مجموعه از System call ها را برای استفاده توسط لایه های بالاتر ایجاد کند. برنامه های سیستم بالای kernel قادر هستند که هم از system call ها و هم از دستورات سخت افزار کند و به نوعی بین این دو فرقی قائل نمی شوند. لذا اگر چه بطرز مختلف آنها قابل دسترسی هستند، اما هر دو عملیاتی را فراهم می سازند که برنامه بتواند برای ایجاد توابع پیشرفته تر به کار گیرد. برنامه های سیستم، به نوبت با سخت افزار و یا system call ها گویی که آنها در سطح یکسان هستند، رفتار می کند. بعضی از سیستم ها حتی از عمل قراتر رفته، به برنامه های سیستمی اجازه می دهند بسادگی توسط برنامه های Application صدا زده شوند. اگر چه برنامه های سیستم در سطحی بالاتر از سطح سایر روتین ها قرار دارند اما برنامه های کابردی می توانند همه چیز را در زیر سطح خود در سلسله مراتب ببینند گویی که برنامه های سیستمی بخشی از خود ماشین هستند. این دستاورد لایه ای در نتیجه منطقی خود به مفهوم یک ماشین مجازی Virtual machine فرض می شود. سیستم عامل VM برای سیستم IBM بهترین مثال یک Virtual machine است، از آنجا که IBM آغازگر کار در این زمینه است.

استفاده از زمان بندی با CPU، CPU Scheduling (فصل ۴) و تکنیک های Virtual machine (فصل ۸) یک سیستم عامل می تواند تصویری (illusion) از چندین Process چندگانه را که هریک بر روی پردازنده در حافظه مجازی خود در حال اجراست، ایجاد کند.

البته به طور نرمال، Process حالت های دیگری هم دارد. مثل system call ها و سیستم فایل، که توسط سخت افزار Base– تهیه نمی شوند. در طرف دیگر سیستم ماشین مجازی هیچ تابع اضافی را تهیه نمی کند. اما به جای آن واسطه ای را که معادل سخت افزار زیرین مورد نظر ماست فراهم می کند. به هر پردازش یک کپی مجازی از کامپیوتر مورد نظر داده می شود.

منابع کامپیوتری فیزیکی در ایجاد Virtual machine، CPU Scheduling زمان بندی CPU مشترک می شوند می تواند بکار رود تا CPU را بطور مشترک مورد استفاده قرار بدهند و تظاهری را که User ها Process خودش را دارند، ایجاد کند. Spooling و سیستم فایل می تواند کارت خوان های مجازی و چاپگرهای خطی را تهیه کند. یک ترمینال اشتراک زمانی Time-Sharing نرمال عمل عملکرد کنسول اپراتور را تهیه می کند.

یک مشکل اساسی سیستم ها دیسک است. فرض کنید که ماشین فیزیکی سه تا دستگاه Disk دارد اما می خواد هفت ماشین مجازی را پشتیبانی کند. واضح است که نمی تواند یک Disk Drive را به هر ماشین مجازی تخصیص بدهد. بخاطر بیاورید که نرم افزار ماشین مجازی خودش احتیاج به فضای خالی دیسک بسیار دارد تا حافظه مجازی و Spooling را تهیه کند. راه حل این است که دیسک های مجازی به نام minidisk نامیده گردند. هر سیستم minidisk را به اندازه شیارهای مورد نیازش بر روی دیسک فیزیکی ایجاد می نماید. آشکار است که مجموع سایز های همه ی مینی دیسک ها بایستی از مقدار واقعی فضای خالی فیزیکی موجود کمتر باشد. لذا به کاربران ماشین مجازی خودشان عرضه می شود. پس آنها می توانند هر نرم افزاری از بسته نرم افزارهای را که روی ماشین اصلی وجود دارند اجرا کنند. برای سیستم Virtual Machine از IBM یک کاربر معمولاً CMS را اجرا می کند که یک سیستم عامل Interactive تک کاربر است.

نرم افزار ماشین مجازی به چند برنامه ای کردن ماشین های مجازی چندگانه بر روی یک ماشین فیزیکی مربوط می شود. اما احتیاج ندارد که هر نرم افزار حمایت شده ی کاربر را مد نظر قرار دهد. این مساله (Arrangemnet) ممکن است یک قسمت بندی مفید از مساله طراحی یک سیستم Interactive چند کاربره به دو قسمت کوچکتر تهیه کند.

اگر چه تئوری ماشین مجازی مفید است اما پیاده سازی آن مشکل است، تلاش های زیادی لازم است تا یک کپی (Dupilcate) دقیق (Exact) از ماشین اصلی تهیه کند. بخاطر بیاورید که ماشین اصلی دارای دو حالت است، User Mode و Monitor mode

نرم افزار ماشین مجازی می تواند در Monitor mode اجرا شود، زیرا که آن یک سیستم عامل است. خود ماشین مجازی می تواند در User mode اجرا شود. از آنجا که ماشین فیزیکی دارای دو mode است، لذا ماشین مجازی هم باید دو mode داشته باشد. در نتیجه ما بایستی یک مد کاربری مجازی (Virtual user mode) و یک Monitor mode داشته باشیم که هر کدام از آنها در مد کاربری فیزیکی می توانند اجرا شوند. آن عملیاتی که در یک ماشین مجازی سبب انتقال از user mode به Monitor mode می شود (مثل یک systemcall یا اجرای دستورات previlage) همچنین بایستی سبب شود که یک انتقال از مد کاربر به مد مانیتور در یک ماشین مجازی صورت گیرد.

این انتقال می تواند به طور کلی نسبتاً ساده و آسان انجام شود. به عنوان مثال وقتی که فراخوانی سیستم در مد کاربری مجازی توسط برنامه ای که روی ماشین مجازی اجرا می شود صادر شود سبب انتقال به مانیتور ماشین مجازی بر روی ماشین حقیقی خواهد شد. وقتی ماشین مجازی، کنترل را به دست آورد می تواند مقادیر رجیستر ها و شمارنده ی برنامه را برای ماشین مجازی عوض نماید تا اثر فراخوانی سیستم شبیه سازی شود. سپس می تواند ماشین مجازی را دو باره راه اندازی نماید. با توجه به اینکه اکنون در مد مانیتور مجازی است، سپس اگر ماشین مجازی شروع به خواندن از کارت خوان مجازی اش نماید در واقع یک دستور العمل I/O ممتاز را اجرا کرده است. از آنجا که ماشین مجازی در مد user فیزیکی اجرای می شود این دستور در مانیتور ماشین مجازی Trap خواهد شد. سپس مانیتور ماشین مجازی باید تا اثر دستور I/O را شبیه سازی کند

در ابتدا فایل Spool شده را که کارت خوان مجاز را پیاده سازی می کند پیدا می کند پس آن خواندن کارت خوان مجازی را به یک خواندن از روی فایل Spool Disk شده منتقل می دهد

تصویر کارت Card Image مجازی بعدی را به حافظه بعدی را به حافظه مجازی از ماشین مجازی انتقال می دهد

بالاخره می تواند ماشین مجازی را دوباره راه اندازی کند حالت ماشین مجازی طوری تغییر پیدا کرده است که گویی دقیقاً یک عمل خواندن از دستگاه کارت خوان حقیقی برای ماشین حقیقی که در مود مانیتور حقیقی در حال اجرا است انجام گرفته است

البته اختلاف اصلی Time است در حالی که یک I/O واقعی ممکن است در ۱۰۰ میلی ثانیه انجام شود I/O مجازی ممکن است زمان کمتری (از آنجا که آن Spool است) یا بیشتر (بخاطر این که تعبیر شده است) طول بکشد بعلاوه CPU ما بین ماشین های مجازی بسیاری به حالت چند برنامه ای در حال اجراست که سبب کم شدن سرعت ماشین های مجازی از راه های غیر قابل پیش بینی می شود در نهایت ممکن است لازم باشد که برای داشتن یک ماشین مجازی درست کلیه دستورها شبیه سازی شوند VM برای ماشین های IBM کار می کند زیرا که دستورات نرمال برای ماشین های مجازی می تواند مستقیماً روی سخت افزار اجرا شود فقط دستورات Previlage (برای I/O حتماً لازم است) بایستی شبیه سازی شود که از این جهت خیلی آهسته تر اجرا خواهد شد

مفهوم ماشین مجازی چندین مزیت دارد توجه کنید که در این مود محافظت کامل برای منابع مختلف سیستم وجود دارد هر ماشین مجازی کاملاً از بقیه ماشین های مجازی جدا سازی شده – لذا هیچ مسئله امنیتی وجود نخواهد داشت از طرف دیگر هیچ اشتراک مستقیم وجود نخواهد داشت برای فراهم نمودن اشتراک پذیری منابع دو روش پیاده سازی شده اند اول ممکن است که Mini Disk را به اشتراک بگذارند این طرح بعد از دیسک تقسیم شده فیزیکی مدل شده اما توسط نرم افزار پیاده سازی شده است با این تکنیک فایل ها می توانند مورد استفاده اشتراکی قرار گیرند دوم اینکه ممکن است که یک شبکه از ماشین های مجازی تعریف شود که هر کدام می تواند اطلاعات را روی شبکه ارتباطی مجازی بفرستند دوباره شبکه بعد از شبکه های ارتباطی فیزیکی مدل شده است اما توسط نرم افزار و پیاده سازی implement شده اند

همچنین سیستم ماشین مجازی یک وسیله نقلیه کامل برای توسعه و تحقیق سیستم های عامل است به طور نرمال تغییر یک سیستم عامل موضوع مشکلی است از آنجا که سیستم های عامل برنامه های بزرگ و پیچیده هستند خیلی مشکل است که مطمئن باشیم یک تغییر در یک نقطه سبب نخواهد شد که خطاهای مبهم در سایر نقاط ایجاد گردد این حالت بخاطر توانایی و قدرت سیستم عامل می تواند خطرناک باشد از آنجا که سیستم عامل در مود مانیتور اجرا می شود یک تغییر اشتباه در یک Pointer می تواند سبب اشتباهی شود که تمام سیستم فایل را ویران کند لذا لازم است که تمام تغییرات سیستم عامل به دقت آزمایش شوند اما سیستم عامل روی تمام ماشین اجرا و کنترل می شود بنابراین سیستم جاری در حالی که تغییرات در حال ساخته شدن است بایستی متوقف شده و به کار برده نشود.

این زمان عمده به نام زمان توسعه سیستم نامیده می شود چون در حین این زمان Time سیستم عامل در اختیار کاربران قرار نمی گیرد زمان System Developement اغلب در آخر شب ها و روزهای تعطیل برنامه ریزی می شود

یک سیستم ماشین مجازی‌می توانند بیشتر مشکلات را حذف کندبرنامه نویس های سیستم‌ماشین مجازی خودشان را در اختیار دارند و توسعه سیستم بر روی این ماشین های مجازی رخ می دهد به جای آنکه بر روی ماشین حقیقی به انجام رسدعمل نرمال سیستم به ندرت به خاطر توسعه سیستم قطع می شود

۳.۷. طراحی و پیاده سازی سیستم System Design & Implementation

در این قسمت ما در مشکلات طراحی و تکمیل نمودن یک سیستم بحث خواهیم کردالبته هیچ راه حل کاملی برای مشکلات طراحی وجود ندارد،اما راه حل هایی وجود دارد که با موفقیت روبرو است.

۳.۷.۱ اهداف طراحی Design Goals

اولین مسئله در طراحی یک سیستم این است که هدف ها و مشخصات سیستم تعریف شده باشدر بالاترین سطح،انتخاب نوع سخت افزار و نوع سیستم در طرح یک سیستم تاثیر بسیار زیادی دارد: دسته ای Batch، اشتراک زمانی Time-Shared، تک کاربره Single-User، چند کاربره Multi-User، توزیع شده Distributed، بی درنگ Real-Time یا همه منظوره General-Purpose.

بعد از این بالاترین سطح طرح،از طرف دیگرمشخص کردن نیازها ممکن است بسیار سخت تر باشد. نیازها به طور عمده می توانند به ۲ گروه تقسیم شوند: هدف های User و هدف های سیستم.

کاربران در یک سیستم بعضی نیازهای آشکار را طالب هستند: سیستم بایستی برای استفاده راحت باشدقابل اعتماد باشدایمن و سریع باشد.البته این مشخصات در طراحی یک سیستم خیلی مفید نیستند زیرا که هیچ توافق عمومی که چطور به این هدفها رسید،وجود ندارد. ...

یک سری نیازهای مشابه هم توسط کسانی که بایستی سیستم را طراحی،ایجاد،نگهداری و اجرا کندمی توانند تعریف شودسیستم عامل بایستی برای طراحی ساده،قابل اجرا و قابل توسعه باشدبایستی انعطاف پذیر،قابل اعتماد،بدون اشتباه و کارآمد باشدبار دیگر لازم به تذکر است که اینتیازها خیلی مبهم هستند و راه حل کلی وجود ندارد.

هیچ راه حل واحدی برای (Requirement) مشکل تعریف نیازهای یک سیستم عامل وجود ندارد.دامنه وسیعی از سیستم ها نشان می دهد که نیازهای مختلف می توانند منجر به راه حل های بسیار گوناگونی برای محیط های مختلف شود.به عنوان مثال،سیستم عامل MS-DOS،یک سیستم تک کاربره برای میکروکامپیوترهابایستی کاملاً با سیستم هایی مثل MVS،فرق داشته باشد.سیستم عامل چند دسترسی،چند [illegible] برای مین فریم های IBM تماس های بسیار دارد.مشخصات و طراحی یک

سیستم عامل یک موضوع بسیار ابتکاری است.هیچ کتاب درسی محض Mere Text Book نمی تواند این مشکل را حل کند.گر چه بعضی اصول کلی وجود دارد که پیشنهادی هستند.

مهندسی نرم افزار Software Engineering یک میدان (زمینه) کلی برای این اصول پیشنهادی است.بعضی ایده ها از این Field مخصوصاً قابل اجرا در سیستم عامل است..

۳.۱.۲ مکانیزم ها و روش ها Mechanisms & Policies

یک اصل مهم جداسازی مکانیزم و روش است.مکانیزم مشخص می کند که چطور(How) کاری انجام شود.بر خلاف آن روش تصمیم می گیرد که چه (What) چه بایستی انجام شود.

به عنوان مثال- یک مکانیزم برای اطمینان از حفاظت از CPU استفاده از Timer است (بخش ۲.۳.) تصمیم اینکه برای چه مدت Timer برای یک کاربر بخصوص در نظر گرفته می شود یک روش تصمیم گیری است.

جداسازی مکانیزم و روش برای انعطاف پذیری بسیار مهم است.روش ها ممکن است از جائی به جای دیگر و یا از زمانی به دیگر تغییر پیدا کنند.در بدترین حالت هر تغییر در روش،یک تغییر در مکانیزم زیرین (Underlying) نیاز خواهد داشت.

مکانیزم کلی مطلوب خواهد بود که اگر یک تغییر در روش ایجاد شود نیاز به دوباره تعریف نمودن فقز بعضی از پارامترهای سیستم داشته باشد.به عنوان مثال،اگر در یک سیستم کامپیوتر یک روش تصمیم ساخته شود که برنامه ها I/O-Intensive حق تقدم بر CPU-Intensive داشته باشد،بعد روش مخالف آن بایستی به سادگی در بعضی دیگر از سیستم کامپیوتر سازمان دهی (Institute) شود.اگر مکانیزم به طور صحیح جداسازی شده باشد و مستقل از روش باشد.

تصمیم گیری های روش برای اختصاص دادن همه منابع و وسائل برنامه ریزی خیلی مهم هستند.هر جا که لازم است تصمیم گرفته شود که آیا منبع اختصاص داده نشود و یا شود،یک روش تصمیم گیری بایستی ایجاد شود.هر گاه سؤال این باشد چگونه به جای چه،این مکانیزم است که باید مشخص شود.

۳.۷.۲ پیاده سازی Implementation

هرگاه که یک سیستم طراحی شد،آن بایستی Implement شود.به طور مرسوم،سیستم عامل به زبان Assembly نوشته می شود.اگر چه به طور کلی دیگر این مطلب حقیقت ندارد.سیستم عامل می تواند به زبان های سطح بالاتر هم نوشته شود.اولین سیستمی که به زبان اسمبلی نوشته نشد احتمالاً Master Control Program اختصاراً MCP برای کامپیوترهای Burroughs بود.MCP به زبان های مختلف (Multics ALGOL (Invariant of ALGOLها که در MIT بوجود آمد به زبان PL/1 نوشته شده

بود.سیستم عامل Unixو OS/1 به زبان C نوشته شده بودند.تنها ۹۰۰ خط از کد یونیکس اصلی به زبان اسمبلی بوده است که اکثر آن هم مربوط به زمان بند و گرداننده های دستگاه است.

سیستم عامل Primos برای کامپیوترهای Prime به زبان های مختلف Fortran نوشته شدسیستم عامل Solo به زبان Concurrent Pascal. نوشته شده است.

مزیت استفاده از زبان های سطح بابلاتر و یا حداقل یک زبان پیاده سازی سیستم برای انجام گرفتن تمهلات سیستم های عامل شبیه زمانی است که زبان برای استفاده برنامه کاربردی به کار می رود.

۱- می تواند سریع تر و جمع و جورتر نوشته شودبرای فهمیدن و غلط گیری کردن آسان تر است و تنها Disadvantageهای اظهار شده،کم شدن سرعت و نیاز به انبار (حافظه) بیشتر هستنداگر چه هیچ کامپوتری به طور همانگ نمی تواند کد بهتری از یک برنامه نویس متخصص زبان اسمبلی تولید نماید،یک کامپایلر اغلب می تواند کدی که یک برنامه نویس معمولی زبان اسمبلی می تواند بنویسد،تولید کند.

۲- جایگزین کردن یک کامپایلر بهتر به طور یکنواخت کد ایجاد شده را برای کل سیستم عامل با دوباره کامپایل نمودن ساده می تواند توسعه دهد.

۳- بالاخره اگر آن به یک زبان سطح بالاتر نوشته شود یک سیستم عامل بسیار ساده خواهد بود قابلیت حمل آن (portability)- که روی سخت افزارهای دیگر اجرا شود- ساده تر خواهد بود.به عنوان مثال،MS-DOS به زبان اسمبلی Intel 8088 نوشته شده است،در نتیجه فقط با CPUهای خانواده Intel کار می کندسیستم عامل Unix که بیشتر به زبان C نوشته شده است،با مقدار زیادی CPUهای مختلف کار می کند که شامل Intel،Motorolla،Sun SPARC و MIPS

می شود.

همچنین مثل سیستم‌های دیگر،توسعه اجراهای کلی سیستم عامل – به احتمال زیاد نتیجه ساختمان Data بهتر و الگوریتم ها است تا کد تمیز نوشتن. بعلاوه،گر چه سیستم های عامل سیستم های خیلی وسیع هستند،فقط مقدار کمی از کد به اجرای بهتر بستگی دارد،مدیریت حافظه و برنامه ریزی کننده CPU به احتمال قوی روتین های اصلی برای اجرای بهتر سیستم هستندبعد از اینکه یک سیستم نوشته شد و به طور درست کار کرد،روتین های گلوگاه Bottleneck می تواند مشخص شود و سپس با روتین های معادل زبان اسمبلی جایگزین شوند.

برای اینکه Bottleneck [illegible] داده شود – ما بایستی قادر باشیم تا اجرای سیستم را زیر نظر داشته باشیم.کد بایستی اضافه شود تا محاسبه کند و [illegible] سیستم را نشان بدهددر تعدادی از سیستم هاسیستم عامل این مطلب را توسط تولید کردن

لیست های ردیابی (Trace Listing) رفتار سیستم‌انجام می دهد. تمام وقایع(event) جالب بر حسب زمان و پارامترهای مهم شان گزارش می شدند و بر روی فایل نوشته می شوند. بعدا یک برنام تحلیلگر می تواند فایل گزارش را پردازش کند تا طرز کار سیستم را مشخص سازد و bottleneck را و کمبود کارائی را مشخص سازد. همین ردیابی می تواند همچنین به عنوان ورودی برای یک شبیه سازی از یک سیستم توسعه یافته پیشنهادی اجرا شود. ردیابی ها همچنین برای پیدا کردن اشتباه در رفتار سیستم عامل می تواند مفید باشد.

یک امکان دیگر آن است که معیارهای کارایی را بی درنگ محاسبه کرده و به نمایش درآوریم. این روش ممکن است به اپراتورهای سیستم اجازه دهد که با طرز رفتار سیستم بیشتر آشنا بشوند و عملیات سیستم را بی درنگ تحول بدهند.

۳۸ تولید سیستم generation:

این امکان وجود دارد که یک سیستم عامل را مخصوصا برای یک کامپیوتر طراحی و کد گذاری و اجرا و پیاده سازی کرد اما به طوز عمومی تر سیستم عامل برای اجرا بر روی هر نوع کامپیوتر در سایتهای مختلف با پیکر بندی جانبی گوناگون طراحی می شوند. بعدا سیستم بایستی برای هر سایت ویژه پیکر بندی شده یا تولید شود .این عمل به نام system generation معروف است.

سیستم عامل معمولا روی نوار یا دیسک توزیع می شوند . برای ایجاد کردن سیستم ها از برنامه ی به خصوصی استفاده می شود.

برنامه sys_genاز روی فایل می خواند و یا از اپراتور برای اطلاعات مربوط به سخت افزار سوال می کند.

الف)از چه cpuهایی استفاده می شوند؟ چه انتخاب هایی نصب install شود. مثل اعداد اعشاری یا کسری.برای سیستم که دارای چند cpuاست و هر cpu بایستی تعریف گردد.

ب)چقدر حافظه موجود است؟ بعضی از سیستم ها مقدار حافظه را خودشان با مراجعه به محل های حافظه بعد از رسیدن به انتها با ایجاد شدن illegal address تهیه می کنند. این روش تعریف می کند که آدرس قانونی آخر را و نیز مقدار حافظه موجود را تعریف می کند.

ج)چه وسایلی در دسترسند؟ سیستم نیاز دارد که بداند چطور هر دستگاه را آدرس بندی کند(شماره دستگاه) شماره انقطاع دستگاه نوع و مدل دستگاه و هر مختصات مخصوص دستگاه را.

د)چه قسمت هایی از سیستم عامل نیاز است واز چه مقادیری از پارامترها استفاده شود. ممکن است شامل: چه مقدار از باقرها ودر چه اندازه ای بایستی استفاده شود. چه نوع الگوریتم زمان بندی cpuبر چه تعداد از process بایستی پشتیبانی شود.

همین که اطلاعات تعریف شده اند از آنها به چند طریق می توان استفاده نمود:

۱)در یک حالت ممکن است از آنها استفاده شود تا یک کپی از کد اولیه از سیستم عامل را تغییر دهد و سپس سیستم عامل کاملا ترجمه compileمی شود: تعریف dataها و مقادیر اولیه اعداد و ثابت ها همراه با کامپایل شرطی یک خروجی وروژن مقصد سیستم عامل را تولید می نماید که برای سیستم توصیف شده مناسب است.

۲)در یک سطح کمتر مناسب at slightly less tailored level تعریف سیستم ممکن است سبب ایجاد جدولها و انتخاب مدول هایی از کتابخانه از قبل کامپایل شده شود. این moduleها با همدیگر link خواهد شد تا سیستم عامل ایجاد شده را فرم دهد. انتخاب اجازه خواهد داد که کتابخانه شامل device driverبرای همه دستگاه های پشتیبانی شده شود. اما فقط آنهایی که واقعا مورد نیاز هستند در سیستم عاملlink می شوند. از انجایی که سیستم دوباره کامپایل نخواهد شد. ایجاد کردن سیستم سریعتر خواهد بود اما ممکن است منجر به سیستمی بسیار کلی تر از انکه مورد نیاز است شود.

۳)در حالت های افراطی دیگرممکن است سیستمی ساخته شود که کاملا جدول باشد. تمام کد همیشه یک قسمت از سیستم خواهد بود و انتخاب در زمان اجرا اتفاق خواهد افتاد نه در زمان کامپایل کردن و نهlink شدن _ تولید سیستم به سادگی مرکب از ایجاد جداول مناسب برای توصیف سیستم خواهد بود.

اختلاف اصلی در بین این روش ها اندازه و عمومیت سیستم تولید شده و نیز سادگی تغییرات در موقع تغییر پیکر بندی سخت افزار می باشد.

عملیاتی که توسط لایه های سطح پائین تر تهیه شده اند. همچنین هر لایه نیاز ندارد که بداند چطور این عملیات پیاده سازی شده اند.

آن فقط احتیاج دارد که بداند چه عملیاتی آن ها انجام می دهند.

اولین سیستمی که طراحی شد با استفاده از روش لایه ای سیستم the است که شش لایه تعریف شده است که در شکل صفحه ی قبل نشان داده شده است. مشکل اساسی روش لایه ای عبارت است از تعریف لایه ها یا سطوح از آنجایی که یک سطح یا لایه تنها می تواند لایه های سطح پایین تر را بکار ببرد ، برنامه ریزی با دقت لازم است به عنوان مثال device driver ها برای پشتیبانی فضای خالی روی دیسک بایستی در یک سطحی باشد پایین تر از سطح مدیریت حافظه ، به این خاطر که حافظه مجازی نیاز دارد که دیسک را بکار ببرد. ممکن است موارد دیگر خیلی آشکار نباشد.مثلا دستگاه دیسک ممکن است برای I\O انتظر بماند و در این حال CPU می تواند دوباره برنامه ریزی شود و پردازش دیگری اجرا نماید.

فصل چهارم

مدیریت پردازش

... واحد کاری در سیستم است هر OS عبارت است در مجموعه ای از پردازش ها که پردازش های OS کد سیستم ...
... و پردازش های USER کد کاربر را اجرا می کنند. همه ی این PROCESS ها می توانند همزمان اجرا شوند که
... توانند بین آنها (مالتی پلکس) به طور اشتراکی مورد استفاده قرار بگیرد.

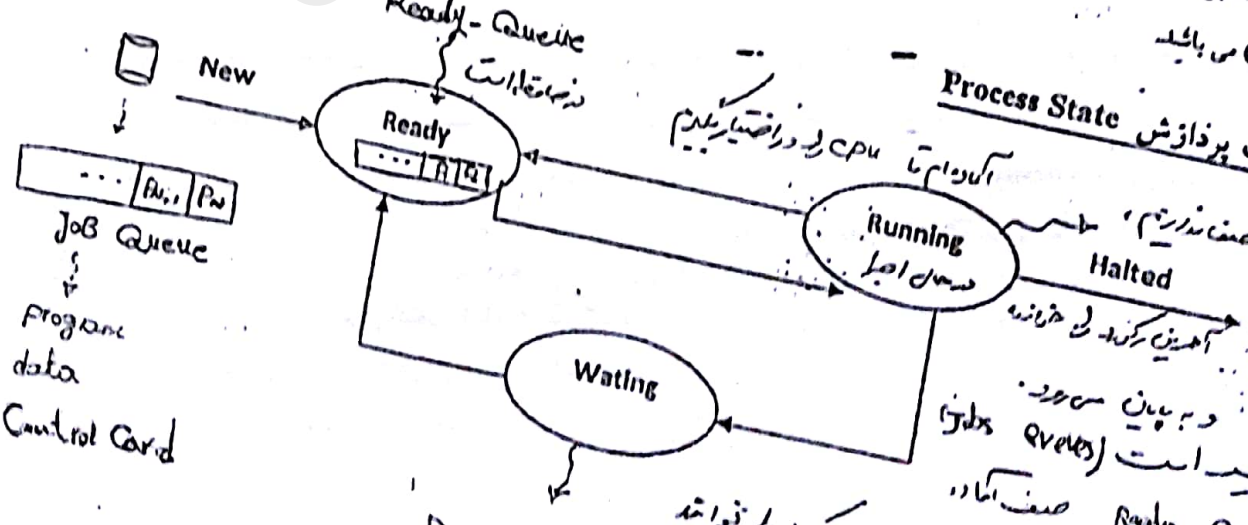
PROCESS CONCEPT پردازش

... بحث کردن از OS این است که فعالیتهای CPU را چه بنامیم، یک سیستم JOB،BATCH ما را اجرا می کند. در حالی
... سیستم TIME SHARING برنامه های USER یا TASK را اجرا می کند... OS ممکن است احتیاج داشته باشد تا فعالیتهای
... خودش را پشتیبانی نماید. مثل SPOOLING را ایس می شود مشاهده کرد همه ی فعالیتها شبیه به هم هستند لذا ما می
... آنها را پردازش نمایم. توجه کنید که کلمات JOB،PROCESS را تقریبا می شود به جای همدیگر به کار برد اما ما
... PROC را ترجیح می دهیم.

۲.۰ sequential process پردازش ردیفی

... طور غیر رسمی یک پردازش ردیفی برنامه ای است که در حال اجرا باشد. اجرای یک process بایستی بصورت ردیفی یا
... شت سر هم اجرا شود، یعنی اینکه در هر لحظه از زمان حداکثر یک دستور از طرف پردازش اجرا می شود. یک پردازش عبارت
... است از اجرای کد برنامه به علاوه فعالیت جاری سیستم است و آن شامل Stack، داده های موقت، بخش داده ما که شامل متغیرهای
... Global(جهانی) می باشد

۲.۱.۲ حالت پردازش Process State



همچنانکه پردازش در حال اجراست State یا حالت آن مرتبا تغییر می کند. Stateای که پردازش (حالت) فعالیت جاری آن را تعریف می کند. هر پردازش ردیفی ممکن است در یکی از حات های سه گانه بالا باشد که در آن Running به معنای پردازشی است که در حال اجراست . حالت Wating یعنی پردازش منتظر است واقعه ای اتفاق بیفتد .

حالت Ready پردازش منتظر است تا به یک Processor تخصیص داده شود.

این مهم است که بدانیم که در هر لحظه از زمان یک پردازش می تواند در حالت Running باشد. ولی چندین پردازش ممکن است در حالت Ready و Wating باشند.

Process Control Block (PBC) ۴.۱.۳

هر پردازش توسط یک PBC مشخص می شود. یک PBC یک بلاک از داده ها و یا یک رکورد شامل تعدادی یا مقادیری از اطلاعات مربوط به یک پردازش مشخص می باشد که این اطلاعات می توانند قسمت های مختلفی را در برگیرند، از جمله حالت که ممکن است Wating و Running و New و Halted یا Ready باشد.

Program Counter که آدرس دستورالعمل بعدی را که برای این پردازش بایستی اجرا شود را مشخص می کند.

Pointer State
Process Number
Program Counter
Register
Memory Limites
List of open file

CPU Rejister که شامل Accumulator ها ، Index Rejister ها و Stack Pointerها و هم چنین اطلاعات مربوط به Condition Code (c.c) (برنامه وقتی اجرا می شوند شرایطی به اسم C.C را مشخص می کند. مثلا برنامه ای که Error پیدا کرده کدهایی که سیستم عامل برای پردازش می دهد می شود C.C)

این رجیسترها همزاه با Program Counter اطلاعات مربوط به حالت پردازش را که بایستی ذخیره شوند در زمانیکه یک Intrupt اتفاق می افتد بایستی ذخیره شود که بعدا اجازه دهد تا بعد از بر طرف نمودن Intrupt پردازش صحیح ادامه پیدا کند.

PCB0 OS PCB1

و هم چنین PCB شامل اطلاعات زمانبندی CPU این اطلاعات شامل حق تقدم یک پردازش نشانگر به صف زمانبندی و بقیه پارامترهای مربوط به زمانبندی می باشد.

اطلاعات مربوط به Memory یا حافظه اصلی

این اطلاعات شامل Page Tableها و limit Register

اطلاعات حسابداری

این اطلاعات شامل زمان بندی واقعی استفاده از CPU ، شماره job پردازش شماره حساب.

اطلاعات مرتبط به وضعیت I/O

این اطلاعات شامل درخواست های I/O دستگاه های I/O که به این پردازش اختصاص داده شده است و هم چنین یک لیست از فایل های Open و ... به طور کلی PCB ها یک منبع از هر گونه اطلاعات پردازش ها را مشخص و معین می کنند

۴.۲ Processهای هماهنگ یا همزمان

در یک سیستم پردازش های همزمان می توانند اجرا شوند یعنی که پردازش های بسیاری از یک CPU استفاده مشترک می کنند چند دلیل برای اجرای همزمان پردازش ها وجود دارد.

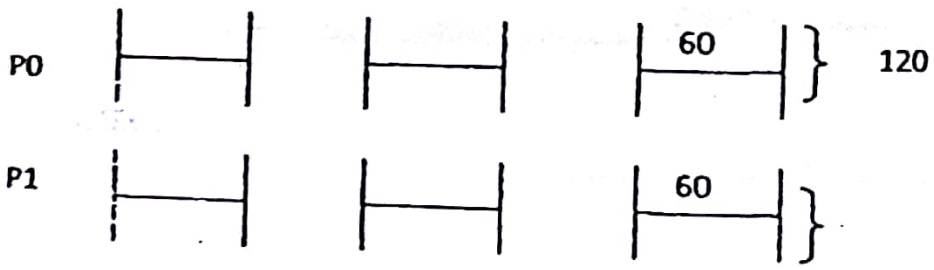
اشراک منابع فیزیکی

به خاطر اینکه سخت افزارهای کامپیوترها محدود هستند لذا ممکن است مجبور باشیم که از آنها به طور مشترک در یک محیط چند کاربره استفاده کنیم.

اشتراک منابع منطقی

از آنجا که چندین کاربر ممکن است نیاز به مقداری اطلاعات مشابه داشته باشند به عنوان مثال یک فایل مشترک لذا ما بایستی دسترسی همزمان به این منابع را تهیه کنیم. در فصل ۵ در این باره صحبت خواهیم کرد

۴.۳ مفهوم زمان بندی



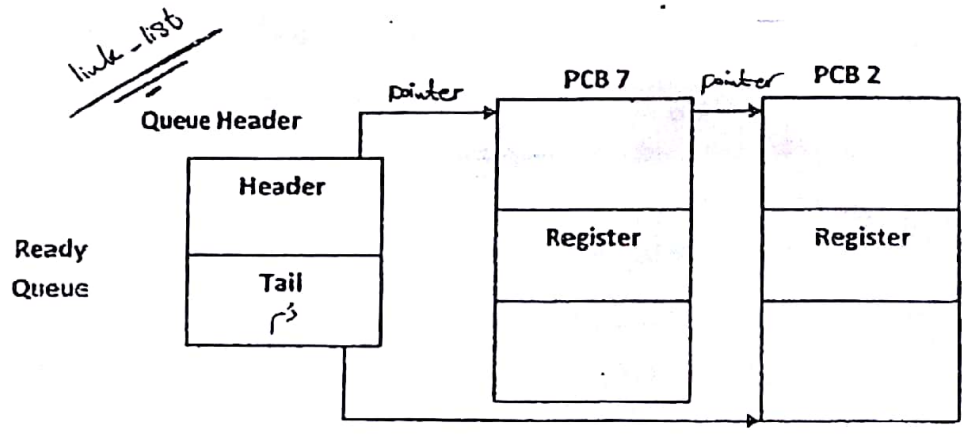
هدف Multi Programming این است که چندین پردازش را همزمان اجرا نماید تا اینکه کارایی CPU را به حداکثر برساند. ایده Multi Programming نسبتا ساده است یعنی اینکه یک پردازش اجرا می شود تا اینکه آن بایستی منتظر مثلا تکمیل شدن درخواست I/O شود. در یک سیستم کامپیوتر ساده CPU بیکار خواهد نشست.

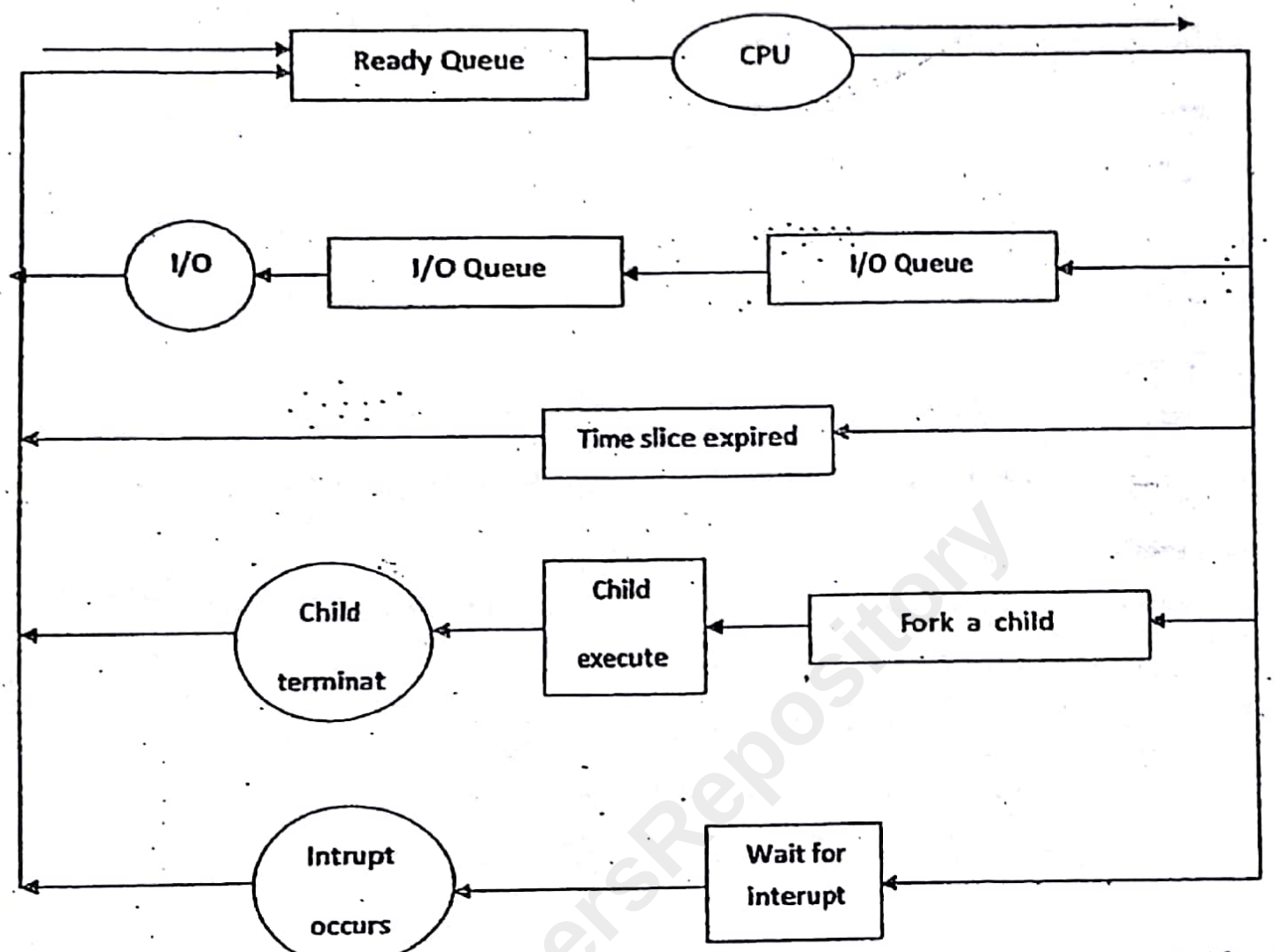
در طول این زمان انتظار CPU کار مفیدی انجام نخواهد داد ولی با M.P می توانیم چندین پردازش را در یک زمان در حافظه نگهداری کنیم و زمانیکه یک پردازش برای واقعه ای منتظر می ماند O.S، CPU را از آن پردازش آزاد نموده و آنرا به پردازش دیگری اختصاص می دهد. (مثال بالا)

مزیت های مهم Multi Programmin بالا بردن کارایی CPU و افزایش Through Put می باشد. (Through Put یعنی مقدار کاری که در یک فاصله زمانی داده شده انجام می شود. مثلا ۱۱ پردازش در یک ساعت)

۴.۳.۱ صف های زمانبندی

همچنانکه پردازش ها داخل سیستم می شوند آنها در Process Queue/ job Queue قرار داده می شوند این صف عبارت است از تمام پردازش هایی که روی دیسک قرار دارند که منتظر هستند تا به آنها حافظه اصلی تخصیص داده شود . پردازش هایی که در حافظه اصلی قرار دارند و آماده هستند که اجرا شوند در یک صف به نام Ready Queue یا (صف آمادگی) نگهداری می شوند.این صف ها به طور کلی لیست های پیوندی یا Link List می باشند که شامل یک عنوان Ready Queue . (Header) شامل Pointer ها خواهد بود که به اولین و آخرین PCB در صف اشاره خواهد کرد.صف های دیگری نیز در سیستم وجود دارد زمانیکه به یک پردازش CPU اختصاص داده می شود و آن به مدت کمی اجرا خواهد شد و سر انجام یا کارش تمام می شود و یا برای انجام شدن یک واقعه به خصوص مثل اتمام درخواست یک I/O منتظر خواهد ماند در چنین حالتی در دستگاه درخواست شونده ممکن است یک دستگاه نوار یا یک دستگاه مشترک مثل دیسک و یا دستگاه های پرینتر و غیره باشد لیست پردازش هایی که برای یک دستگاه به خصوص I/O منتظر هستند Device نامیده می شوند. شکل صف Ready Queue و صف های دستگاههای I/O مختلف را به نمایش می گذارد.





یک شکل کلی از زمانبندی پردازش ها به نام Queue Diagram منحنی صف بندی یا دیاگرام صف بندی می باشد شبیه شکل بالا که هر مستطیل در آن یک صف را نمایش می دهد ~~که در آن یک صف را نمایش می دهد~~ که در آن دو نوع از صف ها نشانه داده شده اند صف Ready Queue و یک مجموعه آغاز صف های دستگاه ها و دایره ها منابعی هستند که به صف ها سرویس می دهند و فلش ها جریان پردازش را در سیستم مشخص می کنند . یک پردازش در آغاز در صف Ready گذاشته می شود به محض به دست آوردن CPU به آن پردازش یکی از حالت های زیر ممکن است اتفاق بیفتد:

۱) پردازش ممکن است که یک درخواست I/O را بنماید سپس در یک صف I/O قرار بگیرد.

۲) پردازش ممکن است اجبارا به خاطر تمام شدن time slice در سیستم های Multi tasking ، CPU را از دست بدهد که در انتهای صف Ready Queue قرار بگیرد.

۳) این پردازش می تواند توسط System Call Fork یک پردازش جدید را Call نماید و تا خاتمه یافتن آن پردازش منتظر بماند.

۴) پردازش ممکن است به طور اجباری cpu را از دست بدهد یعنی در نتیجه ایجاد یک intrupt که دوباره به انتهای صف Ready Queu برگردانده می شود.

۴.۳.۲ زمانبندی کننده ها Schedulers

long term scheduling

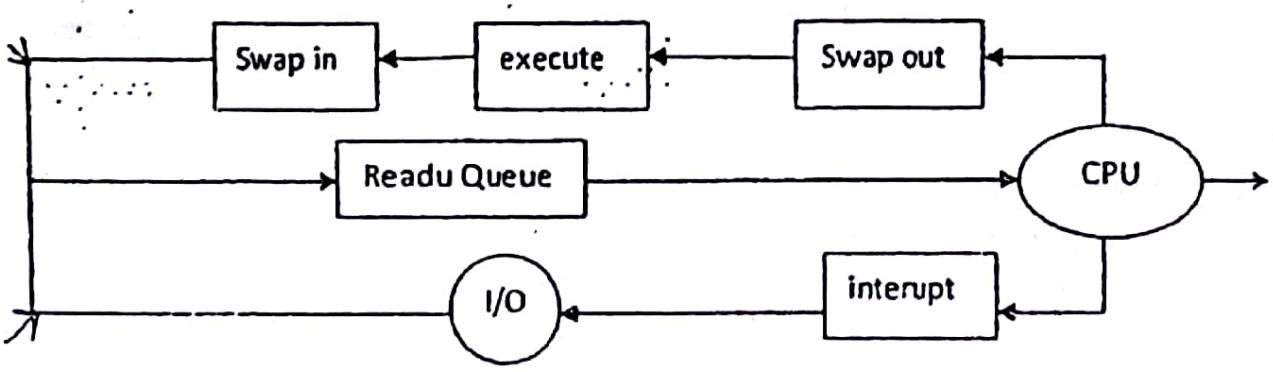
یک پردازش بین صفحه های زمانبندی مختلف در سرتاسر زمان اجرایش گردش خواهد کرد. OS بایستی از این صفحه پردازش هایی را به طریقی انتخاب کند. انتخاب پردازش ها توسط زمانبندی کننده مربوطه انجام پذیرد:

Long Term Scheduler پردازش ها از Job Pool انتخاب نمودهو آنها را داخل حافظه Load می کند تا آنها را اجرا نماید.

Short Term Scheduler از بین این پردازش هایی که آماده برای اجرا هستند (Ready Queue) یکی از این پردازش ها را انتخاب نموده و CPU را به آن اختصاص می دهد. فرق اصلی این دو زمانبندی کننده کثرت اجرایی آنها است یعنی Long Term Scheduler بسیار کمتر از Short Term Scheduler عمل می نماید یعنی Long Term Scheduler زمانیکه اجرای یک Job به اتمام رسیده باشد آن وقت Job جدید را به داخل حافظه Ready Queue خواهد فرستاد در صورتیکه Short Term به دفعات بسیار زیاد برای یک پردازش اتفاق می افتد.

Long Term Scheduler وظیفه انتخاب Jobها را به عهده دارد به این معنی که همانطور که می دانیم Jobها بر دو گونه اند: ۱.CPU bounded ۲.I/O bounded چنانچه Long Term Scheduler همه Jobهای CPU bounded را برای اجرا به صف Ready Queue انتقال دهد در نتیجه CPU و در واقع Ready Queue همیشه پرقرار خواهد بود و در عوض صف های I/O خالی خواهند ماند و اگر Long Term Scheduler همه Jobهای I/O bounded را برای اجرا انتخاب نماید در این صورت CPU بیکار خواهد ماند و صف های I/O همواره پر خواهد ماند این هم باز بر خلاف سیستم Multi programming است پس بایستی یک تعادلی را Long Term Scheduler در انتخاب Jobها برای اجرا رعیت نماید یعنی Jobهای I/O bounded و CPU bounded باشد.

بعضی از OSها مثل Time sharing ممکن است یک سطح میانجی یا سطح میانه (واسطه) از برنامه ریزی را به نام Medium Term Scheduler را معرفی نماید یعنی تنها بایستی یک پردازش را از حافظه (Ready Queue) موقتا بیرون برده تا سیستم بهتر بتواند کارش را انجام دهد. Swap Out و بعدا پردازش را به داخل حافظه برگرداند تا اجرایش را ادامه دهد(Swap in) و طرح کلی آن Swapping این عمل زمانی اتفاق می افتد که مثلا یک پردازش حافظه بسیار زیادی را اشغال نموده است که در نتیجه اجرای آن سیستم با کمبود حافظه مواجه شده است و یا اینکه بشود ترکیب پردازش ها را توسعه داد که از مواقعی است که به Swapping نیاز می باشد.



۴.۴ زمانبندی CPU

Scheduling یا زمانبندی کردن عمل اصلی یک OS است.

تقریبا تمام منابع یک کامپیوتر قبل از اینکه مورد استفاده قرار گیرند زمانبندی می شوند. از آنجا که CPU یکی از منابع اصلی کامپیوتر است لذا زمانبندی آن در طراحی سیستم عامل اهمیت بسیاری دارد.

۴.۴.۱ CPU burst و I/O burst

پردازش را می توان به مجموعه ای از CPU burst و I/O burst تعریف نمود یعنی اجرای هر پردازش با CPU burst شروع شده و سپس توسط I/O burst دنبال می شود که بعدا توسط CPU burst دیگری دنبال می شود و سپس I/O burst دیگر و ... بالاخره آخرین دستور CPU burst پایان پیدا کرده که توسط یک System call به اجرایش خاتمه خواهد داد.

۴.۴.۲ زمان بندی کننده CPU (CPU Scheduler)

Short term scheduling

هر وقت CPU بیکار می شود سیستم بایستی پردازش ها را از Ready Queue انتخاب کرده و اجرا نماید.انتخاب پردازش ها توسط Short Term Scheduler انجام می گیرد. CPU Scheduler از بین پردازش هایی که در حافظه هستند و آماده برای اجرا می باشند یکی را انتخاب کرده و CPU را به آن اختصاص می دهد توجه کنید که Ready Queue لازم نیست که به صورت صف FIFO (First in First out) باشد همانطور که بعدا خواهیم داد الگوریتم های مختلفی برای زمانبندی کردن CPU وجود دارد یعنی یک Ready Queue می تواند به عنوان صف FIFO یا صف تقدم دار (Priority) و یا یک درخت (tree) و یا هم چنین می تواند یک لیست پیوندی غیر مرتب باشد.

۴.۴.۳ Scheduling

زمانبندی کردن CPU تحت شرایط زیر ممکن است انجام شود:

۱.وقتی که یک پردازش در حالت running به حالت waiting انتقال پیدا می کند.

۲.وقتی که یک پردازش از حالت running به حالت ready انتقال پیدا می کند.

۳.وقتی که پردازش از حالت waiting به حالت ready انتقال پیدا می کند.

۴.وقتی که یک پردازش به صورت normal خاتمه پیدا می کند.

برای شرایط ۲و۳ بایستی CPU زمانبندی شود و در شرایط ۱و۴ زمانبندی CPU هیچ عملی را انجام نخواهد داد.

به یک پردازش اختصاص داده می شود و آن پردازش CPU یک non preemptive تحت زمانبندی انتقال waiting را تا زمیکه اجرای خود را تمام نکند آزاد نمی ماند به استثنا اینکه به حالت CPU انتقل پیدا کند.

۴.۴.۴ Context Switch

انتقال CPU به پردازش دیگر (یا تخصیص CPU به پردازش دیگر) نیاز دارد که پردازش قدیمی ذخیره شود و حالت ذخیره شده برای پردازش جدید Load شود.

این Task به نام Context نامیده می شود زمان Context S در ماشین های مختلف فرق می کند و به سرعت حافظه و تعداد رجیستر ها بستگی دارد.

۴.۴.۵ Dispatcher – اعزام کننده

یک عضو دیگر که در عمل زمان بندی درگیر است Dispatcher می باشد آن یک module یا برنامه است که کنترل CPU را به پردازش هایی که توسط Short Term Scheduler انتخاب شده است را می دهد این عمل را درگیر می کند:

۱) Context Switch

۲) تبدیل Monitor Mode به User Mode

۳) پریدن به محل صحیح در برنامه User تا برنامه را دوباره شروع کند.

۴.۵ الگوریتم های زمانبندی کردن

زمان بندی CPU در مورد این که به کدام پردازش در صف CPU- Ready تخصیص داده شود را بحث و بررسی می نماید.

تعداد الگوریتم های مختلف زمانبندی CPU وجود دارد که ما در این بخش چند تا از آنها را تعریف خواهیم نمود. معیارهای مختلفی برای مقایسه الگوریتم های زمانبندی CPU پیشنهاد شده است که این معیارها شامل:

۱. کارایی CPU (CPU Utilitation) : یعنی اینکه CPU را تا آنجا که ممکن است بایستی مشغول سازیم.

۲. through put سیستم: یعنی تعداد پردازش هایی که در واحد زمان تکمیل(انجام) می شود.

۳. turn around time : یعنی فاصله زمانی که از موقع دادن پردازش به سیستم شروع شده تا کامل شدن اجرای پرداش ادامه می یابد به نام T.A.T نامیده می شود.

۴.waiting time : الگوریتم زمان بندی CPU کلا بر مقدار زمانیکه در طول آن زمان یک پردازش اجرا می شود و یا input و out put را انجام دهد تاثیر ندارد فقط بر مقدار زمانیکه پردازش وقت صرف می کند در حالیکه در Ready Queue است تاثیر دارد ... باید کم باشد

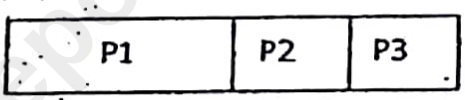
۵.Response time : در یک سیستم time sharing interative مدت زمانی که از فرمان دادن پردازش به سیستم تا اولین جواب که از طرف سیستم داده می شود Response time نامیده می شود ... باید کم باشد

باید توجه داشت که ما بایستی کارایی CPU و through put سیستم را به حداکثر برسانیم و در عوض tum around time و waiting time و Response time را به حداقل برسانیم.

حال الگوریتم های زمانبندی CPU را بحث خواهیم نمود که عبارتند از:

۴.۵.۱ الگوریتم FCFS (First Come First Service)

Process – name	CPU - burst
P1	24
P2	3
P3	3



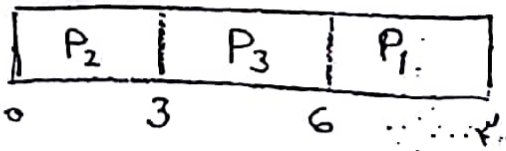
اگر پردازش ها دارای CPU burst های بالا باشد پس انتظار برای Jobها بالا می رود پس این مناسب نیست.

Gant - chart.

A.W.T= $\frac{0+24+27}{3}$

حال اگر اول به ترتیب p2– p3 – p1 وارد شود

Process – name	CPU - burst
p2	3
P3	3
p1	24



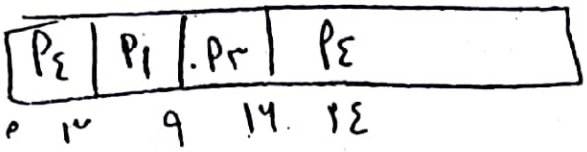
A.W.T=(0+3+6)/3

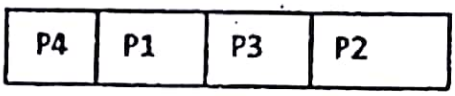
الگوریتم های FCFS ، non preemtiv است.

۴.۵.۲ الگوریتم های Shortest Job First (S.J.F)

Jobهایی را که CPU – burst شان کمتر است اول اجرا می شود

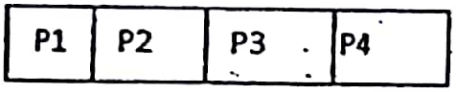
Process – name	CPU - burst
p1	6
P2	8
p3	7
P4	3





A.w.T = (3+16+9+0) / 4= 7

اگر بر اساس FCFS:



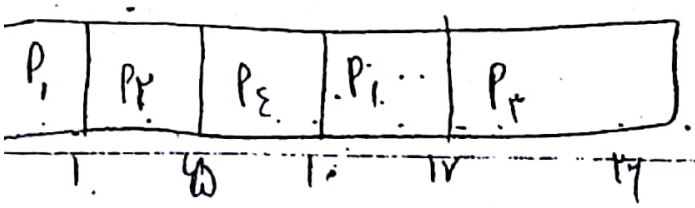
A.w.T = (0+6+14+21) / 4= 10.25

بهتر است از S.J.F استفاده کنیم تا FCFS.

SJF یک روش بهینه است ولی یک مشکل دارد چون از ابتدا نمی دانیم که چه زمانی به هر کدام اختصاص داده شده است پس از روش SJF نمی توانیم استفاده کنیم. باید زمان اجرا را بدانیم SJF هم Preemtive است هم non preemptive.

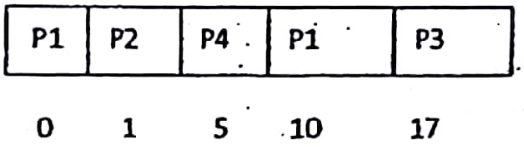
مثال: اگر پردازش های ما به شکل زیر وارد Ready Queue شوند:

Process name	arrival time	CPU burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5



اگر Preemtive در نظر بگیریم

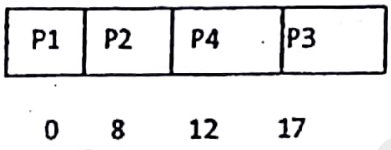
SJF(P)

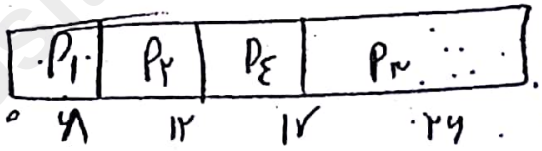


A.W.T = [(10-1) + (1-1) +(17-2) +(12-3)] /4 = 6.5

اگر non preemptive در نظر بگیریم:

SJF(NP)

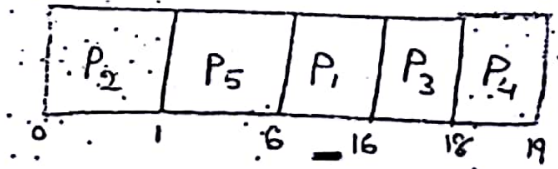




A.W.T = [(0-0) +(8-1) + (17-2) + (12-3)] / 4 = 7.75

۴.۵.۳ الگوریتم Priority «اولویت دار، حق تقدم دار»

Process name	CPU burst	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2



A.W.T = (6 + 0 + 16 + 18 +1)/ 5 = 8.2

حق تقدم حالت کلی SJF است.

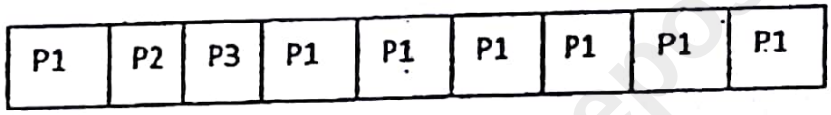
الگوریتم حق تقدم هم Preemtive هم non preemptive است. ممکن است بعضی پردازش ها چون حق تقدم پایین دارند باقی بمانند و از اجرا محروم شوند یا block شود. برای رفع این مشکل : (Priority مشکل Starvation یا indefinitive blocking دارد) هر مدت زمانی (مثلا یک ربع ، یک ثانیه و..) از حق تقدم کم می کنیم که به این عمل aging می گویند. یعنی من Priority را پایین می آوریم و مثلا بعد از ۱۰ ساعت Priority داخل سیستم نمی ماند.

۴.۵.۲ الگوریتم Round Robin

برای سیستم های Multi tasking و Time sharing استفاده می شود. مثلا اگر پردازشی به صورت زیر داشته باشیم:

Process – name	CPU - burst
p1	24
P2	3
p3	3

اگر Time Slice = 4 (Time Quantum)

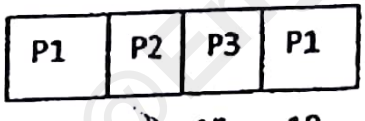


0 4 7 10 14 18 22 24 26 30

A.W.T=[(10-4)+4+7]/3 = 5.66

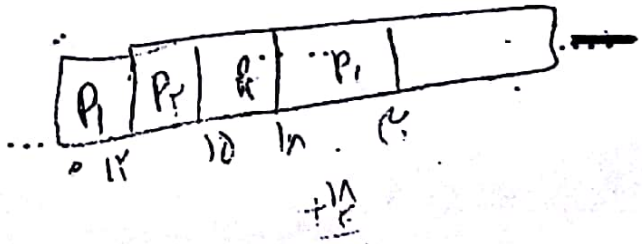
اگر Time Slice =12:

(Context Switch کم می شود.)

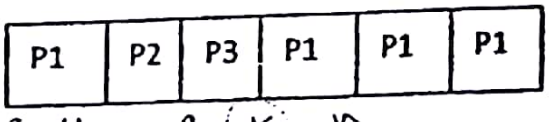


0 12 15 18 30

A.W.T=[(18-12)+15+18]/3 = 13



اگر Time Slice= 6:



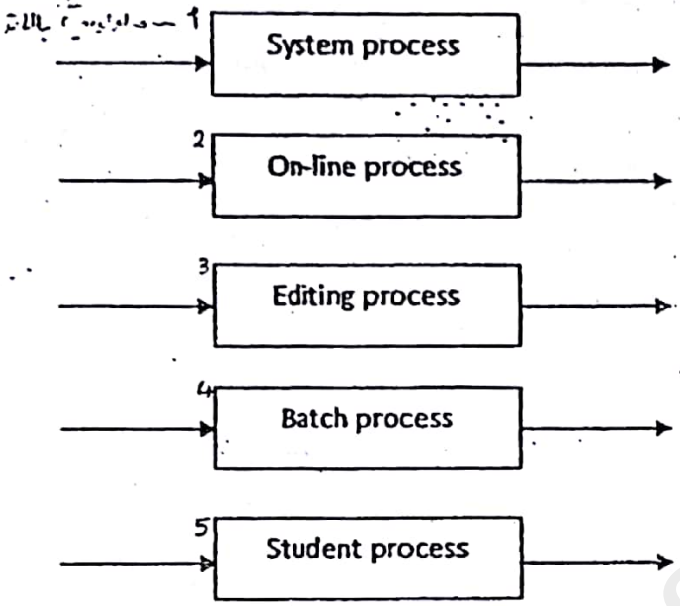
0 6 9 12 18 24 30

A.W.T=[(12-6)+6+9]/3 = 7

در انتخاب time دقت شود که نه آنقدر زیاد که FCFS شود و آنقدر کم که Context Switch شود.

۴.۵.۵ زمانبندی Multi level Queue

از همان ابتدا پردازش ها را به صف های مختلفی تخصیص می دهند مثل on-line proc ess قرار گرفته اند

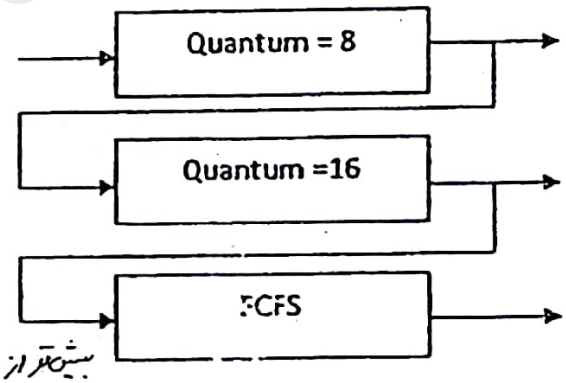


وقتی سیستم اول مثلا مشخص می کند که سیستم پراسس است اجرا می کند و تا اخر انجام می دهد چه ۱ ثانیه چه ۱۰ ثانیه و سپس به صف بعدی و ـبه علاوه بایستی زمانبدی مابین صف ها نیز باشد که معمولا به صورت زمانبندی پس گرفتنی با برتری ثابت پیاده سازی می شود.

در اینجا امکان انتقال پرداش از صف به صفی دیگر نیست و هر کدام در جای خود انجام می شود.

۴.۵.۶ الگوریتم Multi level feed back Queue

بیشتر سیستم های استفاده کننده از Multi level feed back Queue اتصال بین صف ها را مجاز می دانند.



هر پردازشی که می آید وارد صف اول می شود cpu ، ۸ ثانیه وقت می دهد مارش را تمام کند بیرون رود. اگر تمام نکرد به صف بعدی منتقل می کند و در نهایت process هایی به صف ۲ می رسد که زمانشان از ۸ ثانیه و کمتر از۲۴=(۱۶+۸) است. صف دوم وقتی تمام می شود که صف اول تمام شده باشد و به همین ترتیب صف های دیگر. در اینجا Over head کمتر و نیاز به تخمین نداریم. در اینجا می توان priority داد.

فصل پنجم

هماهنگی پردازش (Process Cooperation)

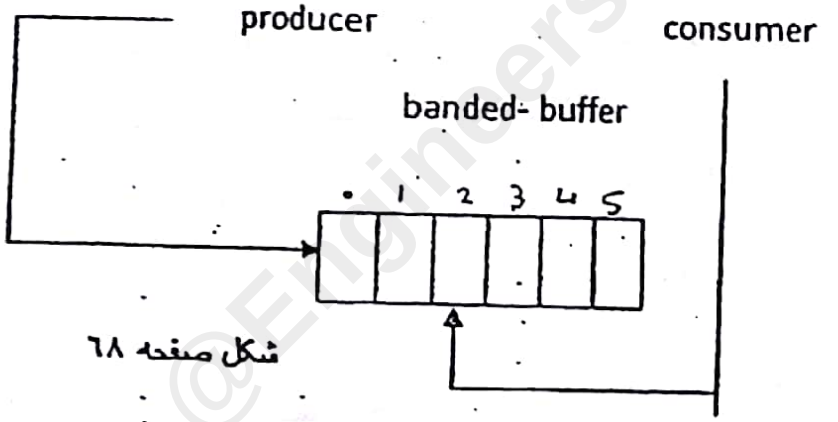
هماهنگی پردازش مبنای سیستم های عامل Multi programming است . در این فصل ما مسئله هماهنگی را با جزئیات بیشتر خواهیم نمود. سیستم های هماهنگ عبارت است از یک مجموعه از پردازش ها .

پردازش های سیستم عامل: کد سیستم را اجرا می کند.

پردازش های کاربر: کد کاربر را اجرا می کند.

تمام این پردازش ها می توانند هم زمان اجرا شوند.

۵.۱ مقدمه



شکل صفحه ٦٨

تغییرهای زیر را برای buffer تعریف می کنیم:

```
Var n

Type item = ...;

Var buffer : arrey[0...n-1] of item

Var in,out: 0 ... n-1
```

?

In: متغیری که خانه بعدی که بایستی پر شود را نشان می دهد.

Out: اولین محل پر شده از buffer را اشاره می کند.

In=out بافر داخلی

In+1 mod n = out بافر پر است.

Counter=0 متغیر مشترک است.

Procedure
Repeat
.....
Procedure an item in next p
.....
While counter = n do no-op
Buffer [In]:= next p
In =in+1 mod n;
Counter = counter +1;
Until false

Consumer
Repeat
While counter=0 do no-op
Next C= buffer [out]
Out=out+1 mod n;
Couner = counter -1;
.....
Consume the item in next ;
Until False;

Register 1 = counter	Register2=counter	
T0:producer execute	register1=counter +1	counter=6
T1:consumer execute	register 2=counter-1	counter=4

counter=4 و counter=6 به تناقض می خوریم.

زمانی که متغیرهای مشترک داریم نباید اجازه بدهیم که پردازش های مختلف متغیرهای مشترک را همزمان تغییر بدهند.(Critial Section)

۵.۲ مسئله بخش بحرانی Critial Section

ملاحظه کنید یک سیستم را که از n پردازش (p0,p1,...,pn) تشکیل شده است. هر پردازش یک بخش از کل به نام Critial Section دارد که در ان پردازش ممکن است مقدار متغیر مشترک را تغیی دهد و یا یک جدول را به روز در آورد و یا اینکه یک رکورد از یک فایل را بخواند و یا بنویسد(خواندن مشکل ندارد ولی نوشتن مشکل دارد.)

مسئله مهم این است که وقتی یک پردازش بخش بحرانی خود را اجرا می کند بقیه پردازش ها اجازه نداشته باشند که بخش بحرانی خود را اجرا کند.و بدین خاطر مسئله بخش بحرانی طراحی شده است تا پردازش ها بتوانند با یکدیگر هماهنگی صحیح و درست داشته باشند.یک راه حل برای مسئله بخش بحرانی این است که سه مورد زیر در آن رعایت شود.

۱.انحصار دو طرفه Mutual Exclusion (ME)

اگر پردازشی در حال اجرای بخش بحرانی خود است بقیه پردازش ها نتوانند بخش بحرانی خود را اجرا کنن.

۲.پیشرفت یا جلو پردن Progress

اگر هیچ پرداشی در حال اجرای بخش بحرانی خود نباشد ولی بعضی از پردازش ها وجود دارند که می خواهند به بخش بحرانی خود وارد شوند. پس فقط آن پردازش هایی که در حال اجرای بخش باقی مانده خود نیستند می توانند در تصمیم گیری اینکه داخل بخش بحرانی خود شوند شرکت نمایند. (حق تقدم با پردازشی است که از بخشش خیر باقی مانده. وارد بخش حرانی شود.)

۳.مدت انتظار محدود bounded waiting (B.W)

بایستی یک محدودیت روی تعداد دفعاتی که بقیه پردازشها مجازند تا داخل بخش بحرانی خود شوند اعمال می گردد یعنی بعد از اینکه یک پردازش درخواست ورود به بخش بحرانی خود را نموده و قبل از اینکه به آن پردازش پاسخ مثبت داده شود.

در بخش های بعدی راه حل مسئله بخش بحاتی را که سه شرط فوق را براورده سازد را مطالعه می کنیم.توجه کنید که وقتی ما یک الگوریتم را ارائه می دهیم فقط متغیرهایی را که برای منظور هماهنگی به کار می رود را تعریف می کنیم و هم چنین فقط یک پردازش به نام p1 را به طور نمونه توصیف می کنیم که ساختمان کلی آن به صورت زیر است.

	0	1
Flag[i]	i	F
Flag[j]	j	F

flag array [0...1] of boolean

flag [i] = true

while flag [j] do no - op

Critical Section C.S

flag [i] = false

remender Section

Until false

الگوریتم دوم:

Flag[j] = T

while (flag[i]) Do no-p

C_S

Flag[j] = F

الگوریتم سوم:

Var flag: array [0 ..... 1] of boolean

i	F
j	F

Turn = T

Turn : 0....1

repeat

flag [i] = true

turn = j

while (flag [j] and turn = j) do no op

Critical Section C_S

flag [i] = false

remainder Section

Until false

Flag[i] = true

Turn = j

while [Flag[j] & Turn = j] Do-p

C_S

Flag[j] = false

j = 1 ⟵ i = 0

j = 0 ⟵ i = 1

در این بخش ما ترجمه خودمان را به الگوریتمهایی در مورد دو پردازش که در یک زمان است مطرح می کنیم . این پردازشها p0 , p1 شماره گذاری شده اند و برای راحتی کار وقتی پردازش Pi را داریم می دهیم پردازش دیگر بنام Pj نامگذاری شده است یعنی j = 1 - i

repeat

while turn ≠ I do no - op

Critical Section

turn = j

Remainder Section.

Until

false ;

متغیر turn را با یک آرایه عوض می کنیم .

turn متغیر مشترکی است که در هر دو پردازش بکار میرود .

Flag [0] در آغاز False است	True	True
Flag [1]	False	False

var flag : array [0....1] of bolean

repeat

توسط تاکید نمودن ایده های آلگوریتمهای ۱ و ۲ ما میتوانیم راه حل صحیح مسئله بخش بحرانی را

بدست آوریم بطوریکه هر سه مورد بخش بحرانی رعایت می گردد . این پردازشها دو متغیر زیر را بطور

اشتراکی مورد استفاده قرار می دهند .

در آغاز Flg 1 = Flg 0 = False میباشد و مقدار num صفر و یک است که فرق نمی کند که در آغاز

صفر باشد یا یک .

ساختمان پردازش Pi در شکل صفحه قبل نشان داده شده است . برای داخل شدن به بخش بحرانی

پردازش pi نخست flag [I] = true می نماید و نیز بیان می کند که اگر مناسب باشد نوبت پردازش

دیگر است منظور j است که به بخش بحرانی خود وارد شود . (num = j) اگر هر دو پردازش سعی

کنند که وارد بخش بحرانی خود در یک زمان شوند ؛ از آنجا که num نمی تواند در یک زمان هم مقدار I

هم مقدار j را داشته باشد لذا مقدار آخری num تصمیم می گیرد که کدام یک از این دو پردازش بعی

تواند وارد بخش بحرانی خود شود . برای اثبات ۱ می گوئیم که هر Pi وارد بخش بحرانی خود می شود

فقط اگر flag [j] = false باشد و یا اینکه num = I باشد . همچنین اگر هر دو پردازش بتوانند در

یک زمان در بخش بحرانی خود اجرا شوند یعنی flag [I] = flag [0] = true در نتیجه مشاهدات .

فوق تلویحاً می گویند که P0 و P1 نمی توانند بطور موفقیت آمیز جملات While خودشان را همزمان

اجرا کنند . از آنجا که مقدار num می تواند صفر باشد و یا یک و نه هر دو پس حالت یک رعایت

میشود .

برای اثبات موارد ۲ و ۳ توجه کنیم که Pi نمی تواند وارد بخش بحرانی خود شود اگر شرط num = j

و flag [j] = True درست باشد . لذا اگر Pj آماده نباشد تا وارد بخش بحرانی خود شود یعنی Flag

اکمار دوطرفه

False = flg [j] پس Pi می تواند وارد بخش بحرانی خود شود . اگر Pj دارای True = flg [j] باشد

و همچنین در حال اجرای جمله while خودش باشد پس turn = i است یا turn = j خواهد بود .

اگر turn = i باشد پس pi می تواند وارد بخش بحرانی خود بشود و اگر turn = j باشد Pj وارد

بخش بحرانی خود می شود .

اگر چه به محض اینکه Pj از بخش بحرانی خود خارج شود flg [j] = false خواهد شد که اجازه

خواهد داد Pi وارد بخش بحرانی خود شود . اگر Pj . true = flg [j] شاید آن پایستی هم چنین turn

= i شاید بنابراین از آنجا که Pi می تواند مقدار متغیر turn را تغییر دهد در هنگامی که جمله while

در حال اجرا است لذا Pi می تواند وارد بخش بحرانی خود شود (progress بعد از حد اکثر یک ورود

به بخش بحرانی توسط Pj (bounded waiting)

5 . 2 . 2 راه حل برای چند پردازش

سه الگوریتم را برای مسئله بخش بحرانی برای دو پردازش بیان کردیم حال در الگوریتم مختلف

برای حل مسئله بخش بحرانی را برای n پردازش بیان می کنیم

الگوریتم ۴ : ساختمان داده های مشترک عبارتند از :

```
Var flag: array [ 0.. n - 1] of ( idele, want-- in, in-cs)

turn: 0 . . n - 1
```

تمام عناصر flag مقدار اولیه idele را خود خواهند داشت .

و مقدار اولیه turn یکی از اعداد (0,1,... , n-1) خواهد بود با توجه به اینکه در هر لحظه از زمان

turn دارای فقط یک مقدار خواهد بود نه بیشتر ساختمان Pi در شکل زیر نشان داده شده است.

```
Var j : 0 . . n;
repeat
    repeat flag [ j ] = want - in
           j: = turn
    while j = I
    do if flag [ j ] = idle
           then j : = turn
           else j := j + 1 Mod n;
    flag [ j ] : = in - cs;
    j = 0;
    while ( j < n ) and ( j = I or flag [ j ] = in - cs ) do j : = j + 1;
    turn : = I;
    Critical Section
    j: = turn + 1 mod n;
    while ( flag [ j ] = idle ) do j: = j + 1 mod n;
    turn : = j ;
    flag [I] : = idle
    Reminder Section
until false;
```

برای اثبات مورد ۱ توجه کنید که پردازش P به بخش بحرانی اش وارد می شود فقط اگر flag [j]

مخالف in - cs باشد (در بخش critical section) برای همه j = i باشد

از آنجا که فقط Pi می تواند in - cs = flag [i] شاید و همچنین flag [j], Pi را تست میکند فقط زمانی که in - cs = flag [I] است پس مورد یک رعایت شده است.

برای اثبات مورد ۲ ما مشاهده می کنیم که مقدار turn می تواند تغییر یابد وقتی که یک پردازش وارد بخش بحرانی خود می شود و وقتی که از بخش بحرانی خود خارج می شود بنابراین اگر هیچ پردازشی بخش بحران خود را اجرا نکند و یا از آن خارج نشود مقدار turn ثابت می ماند لذا اولین پردازش از ترتیب دورانی (Turn, turn+1, ..., n - 1, 0, ..., turn - 1) وارد بخش بحرانی خود می شود.

برای اثبات مورد ۳ مشاهده میکنیم که وقتی که یک پردازش بخش بحرانی خود را ترک می کند بایستی مشخص کند به عنوان تنها خانه‌نین اولین پردازش از ترتیب دوران (turn+1,...,n-1,0,..., turn) با مشخص سازد که هر پردازش که می خواهد وارد بخش بحرانی خود شود در طول مدت n-1 دفعه انجام شود.

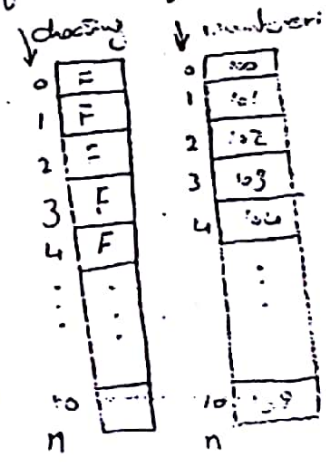
5- الگوریتم Bakery برای n پردازش

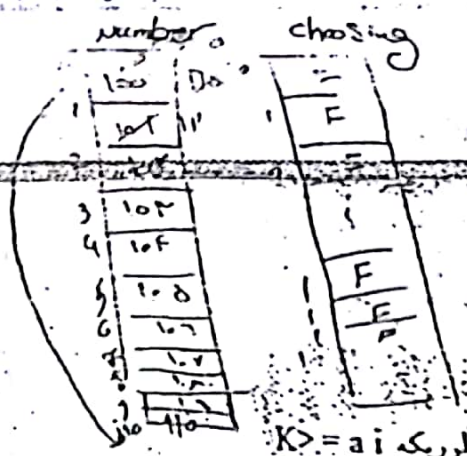
یک روش دیگر برای مسأله بخش بحرانی الگوریتم Bakery است به این ترتیب که در هنگام ورود به فروشگاه هر مشتری یک شماره را دریافت میکند به مشتری که دارای کمترین شماره است خدمت سرویس داده میشود ساختمان داده های مشترک الگوریتم عبارت است از:

```
var   choosing : array [0 . . . n-1] of boolean;
      number   : array [0 . . . n-1] of integer;
```

در آغاز آرایه choosing دارای مقدارهای اولیه fals و آرایه number دارای مقادیر اولیه صفر خواهد بود برای راحتی کار تو فرمول زیر را تعریف می کنیم که

$(a,b) < (c,d)$ اگر کل آرایه پردازد، محدودیت $(n-1)$ است:





$(a,b) > (c,d) = \begin{cases} a > c \\ a = c \Rightarrow b > d \end{cases}$

اگر $a<c$ باشد و اگر $a=c$ است $b<d$ باشد

$K > = a_i$ عددی نام K خواهد بود به طوریکه $K = \text{Max}(a_0, a_1, \ldots, a_{n-1})$

$i = 0,1,2,\ldots,n-1$ برای

P_i ساختار پردازش

repeat

```
choosing [I] =true
number [I] = Max (number [0], number [1],..., number [n-1]÷1;
choosing [I] = false
for j: 0 to n-1
do begin
        while choosing [j] do No-Op;
        while number [j]≠0
        and ( number [j], j ) < ( number [I], I ) do no-op;
end;
```

Critical Section

```
number [I] : = 0;
```

Reminder Section

Until false

برای اینکه ثابت کنیم که الگوریتم Bakery درست است ما نخست نیاز داریم که نشان دهیم که اگر P_i در بخش بحرانی خود است و $PK\ (K = i)$ یک شماره آن را انتخاب نموده است یعنی $number[k] = 0$

پس بایستی $number[I], I < number[k], k$ باشد.

با استفاده از نتیجه فوق حال ساده است که نشان دهیم M.E رعایت شده است.

Function ـــــ چند ورودی و یک خروجی

Procedure ـــــ می‌تواند ورودی نداشته یا یکی یا چندتا داشته باشد و خروجی هم می‌تواند نداشته باشد یا چندتا داشته باشد.

در حقیقت ملاحظه کنید Pi در بخش بحرانی خود است و PK سعی کند تا وارد بخش بحرانی خود شود وقتی که پردازش PK جملات while دوم برای i = j را اجرا میکند می فهمد که (number[i],i)<number [k], k;o= number[i] بنابراین loop در جمله while دوم ادامه پیدا میکند تا اینکه Pi از بخش بحرانی خود خارج شود. برای اینکه نشان دهیم که Progrss و b.waiting رعایت شده اند کافی است که مشاهده کنیم که پردازش ها داخل بخش بحرانی خود می شوند بر مبنای first serve, first come

3.3 syncronization hardware

در این بخش ... ات سخت افزاری ساده ای را که بر سیستم های مختلف موجود هستند را ارائه می دهیم و نشان می دهیم که چطور آنها می‌توانند به طور موثر مورد استفاده قرار گیرند. برای حل مسائل بحرانی.

مساله بخش بحرانی می تواند به سادگی حل شود اگر ما بتوانیم اجازه ندهیم که intrruppt ها در حالیکه یک متغیر مشترک در حال تغییرات است اتفاق بیافتد بنابراین ... راه حل همیشه fisibel نیست (شدنی نیست).

بنابراین بیشتر ماشین ها دستورات به خصوص را تهیه می کنند که به ما اجازه می دهد به طور ساده محتویات یک word را تغییر داده و یا محتویات دو word را با هم عوض کنیم در این تست مفاهیم کلی این نوع دستورات را به طور عام مورد مطالعه قرار میدهیم.

دستور Test and Set که به روش زیر تعریف می گردد

function test and set (var target : boolean) : boolean

```
begin
        test and set : = target
        target : = true
end;
```

اگر یک ماشین دستور test and set را پشتیبان نباید ما می توانیم Mutual exclusion را

توسط معرفی کردن یک متغیر boolean به نام lock را که در آغاز مقدار اولیه fals را داراست پیاده

سازی می نمائیم. ساختمان برداری Pi در شکل زیر نشان داده شده است.

```
repeat
    while test - and - set (lock) do No-Op;
        Critical Section
    lock : = false
    Reminder Section
Until false;
```

الگوریتم پیاده سازی mutual excution با استفاده test and set

5. 4 Semaphore

راه حلهای پیشنهاد شده برای حل مسئله بخش بحرانی برای عمومیت دادن آن به مسائل پیچیده تر به سادگی انجام نمی گیرد.

برای غلبه به این مسئله ما می توانیم از یک وسیله هماهنگ کننده بنام Semaphore استفاده کنیم.

یک (S.) یک متغیر صحیح است که به غیر از مقدار اولیه گرفتن فقط از طریق دو عمل استاندارد

یعنی wait و سیگنال قابل دسترسی است. تعریف کلاسیک wait و سیگنال به شرح زیر است

```
wait (S) : While S≤ 0 do no - op
S: = S -1
```

Signal(Sync)

Signal (S): S:=S+1

5 : 4 . 1 چگونگی استفاده از سمافور

سمافور می تواند برای حل مسئله بخش بحرانی n پردازش مورد استفاده قرار گیرد یعنی n پردازش

یک سمافور مشترک بنام Mutex که به آن مقدار اولیه ۱ داده شده است را بطور اشتراکی مورد استفاده

قرار می دهند Mutex = 1 هر پردازش Pi بطریق زیر سازماندهی می شوند.

repeat

wait (mutex) ;

Critical Section

Signal (Mutex) ;

Remainder Section.

Until false

mutex برای حل مسئله بحرانی بکار رفته .

مورد استفاده دیگر سمافور برای حل سایتل Syncronization به عنوان مثال دو پردازش که همزمان

در حال اجرای P1 با جمله S1 و P2 با جمله S2 و فرض کنید که S2 بایستی بعد از اینکه اجرای

S1 کامل شد اجرا شود . ما می توانیم این طرح را به سادگی پیاده سازی کنیم با اجازه دادن به P1

و P2 که یک سمافور مشترک بنام SYNC را که دارای مقدار اولیه صفر است را به صورت زیر بکارد

استفاده قرار دهد.

Sync = 0

S1 ;

Signal (Sync) ;

برای پردازش P1

---

wait (Sync) ;

S2 ;

برای پردازش P 2

در پردازش P 2 در آنجا که در آغاز Sync مقدار صفر را دارد P2 ، S2 را فقط زمانی اجرا می کند که P1 سیگنال (Sync) را اجرا کرده باشد.

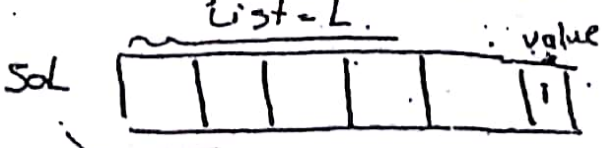
5.4.2 پیاده سازی سمافور

عیب اصلی در راه حلهای مربوط به بخش بحرانی که در بخش 5.2 و نیز در تعریف کلاسیک سمافور آمده است این است که همه آنها نیاز به busy waiting دارند یعنی اینکه زمانیکه یک پردازش در زمان بحرانی خود است بقیه پردازشها که سعی کنند تا وارد بخش بحرانی خود شوند بایستی بطور مدام در بخش ابتری خود یعنی Loop باقی بمانند برای غلبه شدن به این مشکل ما می توانیم عملیات Wait و سیگنال را مقداری تغییر دهیم که در آن پردازش به جای busy waiting می تواند خودش را Block نماید. عمل بلاک به این معنی است که یک پردازش را داخل صف waiting یا سمافور مربوط قرار می دهد و حالت پردازش به حالت waiting تغییر دهد بعد از اینکار به CPU Schaduler انتقال پیدا نموده تا پردازش دیگری را برای اجرا انتخاب نماید.

برای پیاده سازی سمافوردر طراحی فوق ما سمافوررا به عنوان یک رکورد تعریف می کنیم یعنی

```
Type Sema Phare = record
Value : integer;
L: List of Process;
end ;
```

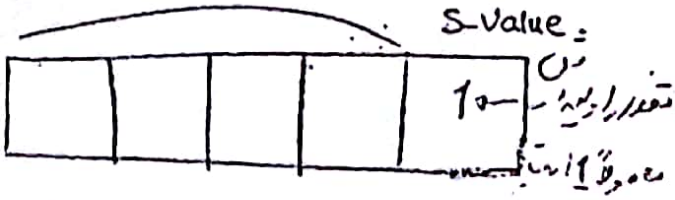
سمافور مدرن کلاسیک
سمافور را به عنوان یک رکورد تعریف کردیم.

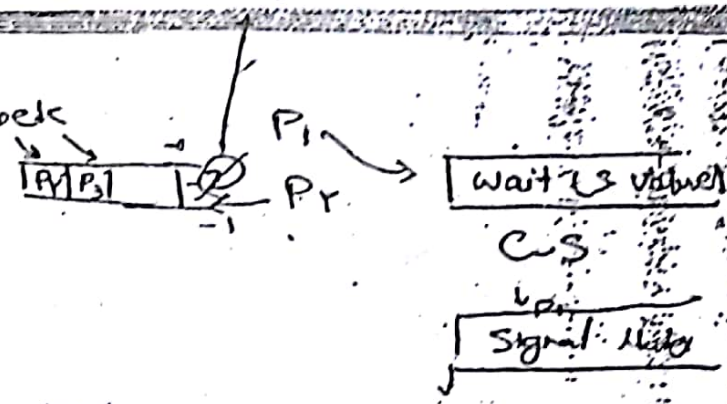


حال عملیات سمافوربطریق زیر تعریف می شود.

```
Wait( S) : S.value := S.Valve -1 ;
if S.value < 0 Then
begin
```

سمافور مدرن :





add this process to S.L

block ;

end ;

Signal (S) : S. Value = S. value + 1

if S.Valuck < = 0 Then

begin

remove a process P from s.L ;

Wake up (P) ;

end ;

بایستی یک پردازش که بلاک شده و منتظر است (با یک سمافور بنام S) توسط اجرای یک عمل سیگنال

توسط بقیه پردازشها دوباره اجرا شود توسط یک عمل Wait up که حالت پردازش را از waiting به

حالت ready تغییر می دهد و سپس پردازش به صف Ready Queue فرستاده می شود تا در صورت

امکان CPU را در اختیار گیرد.

هر سمافور دارای یک مقدار صحیح و یک لیست از پردازشها خواهد بود وقتی که یک پردازش بایستی

منتظر بماند آن پردازش به صف پردازشهای دیگر اضافه می شود . یک عمل سیگنال یک پردازش را از

لیست پردازشهای در حال انتظار کم نموده و آنرا فعال می سازد : Wait up

شکل شده سمافورها این است که آنها بایستی به صورت یکتا در نظر گرفته شوند یعنی اینکه ما بایستی

تضمین کنیم که دو پردازش نتوانند عملیات Signal Wait را روی یک سمافور و مشابه و در یک زمان

مشابه اجرا کنند . که می توان آنرا بطریق زیر حل کرد یعنی اینکه ابتدائیات ها یا انتظاعها را در طول

عملیات Wait و Signal ممنوع کنیم .

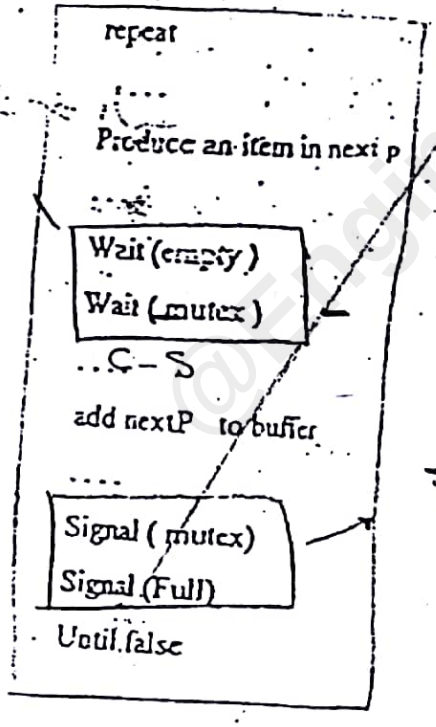
5.5 مسائل کلاسیک هماهنگ سازی

در این بخش ما تعدادی مسائل مختلف هماهنگ سازی را ارائه می دهیم که بسیار مهم هستند زیرا آنها مثالهایی برای یک مجموعه وسیع از مسائل Concurency - Control هستند لذا از سمافورها برای حل آنها می توان کمک گرفت .

5.5.1 مسئله Bounded - buffer

در اینجا ما ساختمان کلی B.B را ارائه می دهیم و فرض می کنیم که Pool عبارت است از n باتر است و هر کدام ظرفیت نگهداری یک آیتم را دارند و از سمافور Mutex جهت دسترسی به باتر استفاده می کنیم که در آغاز مقدار اولیه 1 را دارا باشد و نیز از دو سمافور دیگر به اسم empty - Full که به ترتیب تعداد باترهای خالی و پر را می شمارند . سمافور empty دارای مقدار اولیه n و سمافور Full دارای مقدار اولیه صفر را خواهد داشت با توجه به مطالب فوق حال می توانیم ساختمان پردازش Producer تولید کننده و ساختمان پردازش Consumer یا مصرف کننده را بنویسیم .

Full = 0
empty = n
mutex = 1



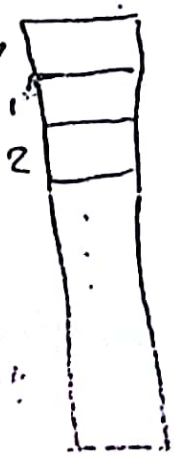
Consumer

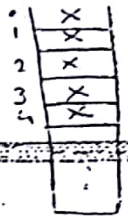
mutex = 1
empty = n Full = 0

n-1
1-1=0

signal(S): S = S+1

1-1=0





Full = 5
empty = n-5

```
repet
    Wait (Full)
    Wait ( mutex)
    ---------
    remove an item from buffer to next C
    ---------
    Signal ( mutex) ;
    Signal ( empty) ;
    ---------
    Consume The Item in next C
    .....
```

mutex ۱ بوده که می شود 0

n = empty n تا خالی داریم

0 = Full یعنی 0 تا پر داریم

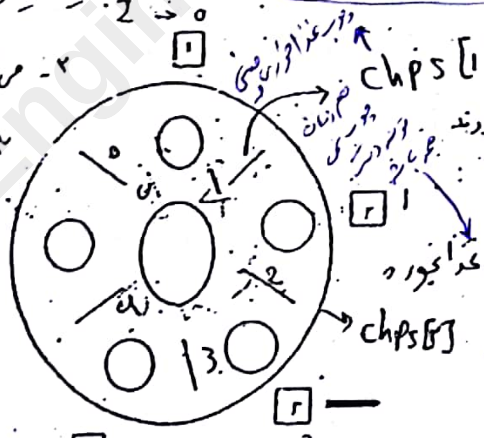
این برنامه مشکل هم زمانی پیش می‌یاد و توش
۳ تا راه حل واسش داریم
۱-۱) chps آزاد کنیم — فیلسوف‌ها I
۱-۲) یک فیلسوف کم کنیم
۲- می‌تونیم محدودیت بذاریم
۳- فرد دست‌ها بتواست چپ و زوج‌ها دست راست رو بردارن

5.5.2 Dining - Philosephers

بگیم

پنج فیلسوف هستند می خواهند شام بخورند

با میزبان برنامه‌نویس
در هر وقت فیلسوف‌ها می‌توانند غذا بخورند
۲) فیلسوف همسایه وقتی یکیشون می‌خواد غذا بخوره اون یکی نمی‌تونه بخوره — گرسنه
از سمافور استفاده می‌کنیم



پنج فیلسوف زندگیشان را به فکر کردن و خوردن می گذرانند. فیلسوفها یک میز دایره ای را که دارای 5 صندلی است و هر صندلی متعلق به یک فیلسوف است مورد استفاده قرار می دهند. در مرکز این میز یک کاسه برنج قرار دارد و 5 Chop Sticks روی میز قرار دارد. وقتی یک فیلسوف فکر می کند او نمی

تواند با دیگر فیلسوفها تماس داشته باشد هرچند مدت یک فیلسوف گرسنه شده و سعی می کند که دو

عدد از Chop Stick را که نزدیک به او هستند(یعنی Chop Stick در کنار دست راست و چپ او

هستند) را بردارد و هر فیلسوف می تواند فقط یک Chop Stick را هر دفعه بردارد آشکار است که او

نمی تواند Chop Stick ای را که در دست فیلسوف دیگر است را بردارد زمانیکه یک فیلسوف گرسنه

است و هر دو Chop Stick را همزمان در دست دارد . آنوقت می تواند بخورد بدون آنکه آنها را رها

کند. وقتیکه خوردنش تمام شد او هر دو Chop Stick را روی میز گذاشته و شروع به فکر کردن می

نماید . این مسئله به خاطر اهمیت علمی آن یک مثال بسیار خوب از یک کلاس بزرگ از مسائل

Conciising Control است یک راه حل ساده این است که هر Chop Stick را توسط یک سماتور ارائه

داد . یعنی یک فیلسوف سعی می کند که یک Chop Stick را توسط اجرای عمل Wait روی آن سماتور

در دست بگیرد . و همچنین Chop Stick خود را توسط اجرای یک عمل سیگنال روی سماتور مناسب

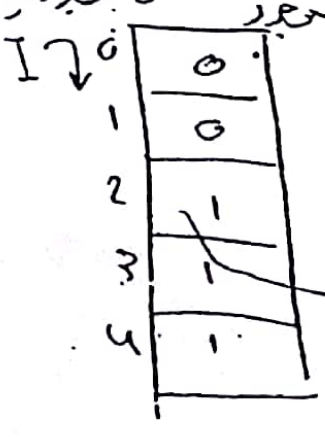
آزاد می نماید .

ساختمان داده های مشترک عبارت است از

```
Var chopstick : array [0. . 4] of Semaphore
```

در آغاز هر Chop Stick مقدار اولیه ۱ را خواهند داشت. ساختمان فیلسوف بشرح زیر می باشد.

```
:Repeat
wait ( Chop Stick [ I ]);
Wait ( Chop Stick [ I + I mod 5 ];
. . . . . .
eat
. . . . . .
Signal ( Chopstick [i])
Signal ( Chopstick [i+] Mod 5 ] ;
```



Think

Until false

اگر چه این الگوریتم تضمین می کند که دو همسایه فیلسوف یا دو فیلسوف مجاور همزمان غذا نمی توانند

در حال خوردن باشند ولی احتمال وجود deadlock این تست وجود دارد به این خاطر که اگر 5

فیلسوف همزمان گرسنه شوند و هر کدامشان Chop Stick سمت چپ خود را بردارد و در این صورت

همه عناصر chop stick ها مساوی صفر خواهند شد و وقتیکه هر فیلسوف سعی کند که Chop

Stick سمت راست خود را بردارد او برای همیشه منتظر خواهد ماند.

چند راه حل برای مسئله بن بست در زیر لیست می شود تا اطمینان دهد که بن بستها اتفاق نیفتد:

۱- اجازه دادن حداکثر به 4 فیلسوف که همزمان بنشینند

۲- اجازه دادن به یک فیلسوف که Chop Stick خودش را هنگامی بردارد که هر دوی آنها موجود

باشند.

۳- به فیلسوفهای فرد اجازه داد اول Chop Stick سمت چپ خود را بردارند و سپس Chop Stick

سمت راست خود را و برای فیلسوفهای زوج برعکس عمل نماید.

* ص 354 (ابتدای) چند صفحه جلوتر است.

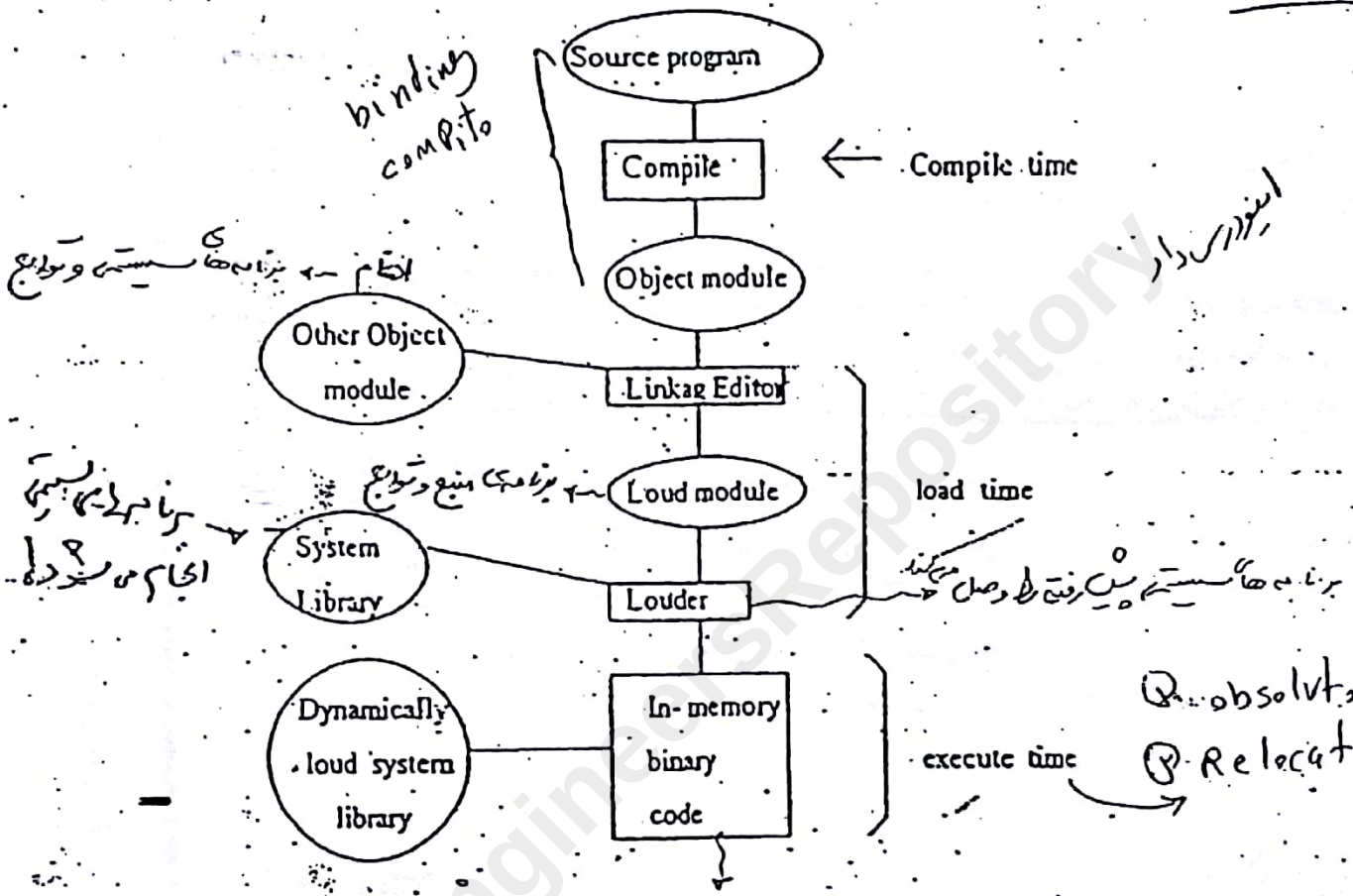
برای استفاده بهتر در فضای حافظه می توان از Dynamic Loading بهره برد به این معنی که تا

زمانیکه یک روتین یا تابع یا سایروتین Call شود در حافظه Load نشود

7.1.3 Dynamic Linking اتصال پویا

بیشتر OS ها Static Linking را پشتیبانی می نمایند یعنی اینکه System Library و Object

Moudule مرتبط Loader داخل برنامه اجرایی ترکیب می شود.



دل در dynamic linking که شبیه dynamic loading است یعنی به جای اینکه loading تا زمان

اجرا عقب بیفتد linking به عقب می افتد

Relocatable : قابل جابجایی در زمان اجرا

obsolete : مطلق

چگونه در حافظه صرفه جویی کنیم
1) Dynamic loading
2) Dynamic linking
3) overlay

اگر یک برنامه 200K را بخواهیم اجرا کنیم و حافظه 150K خالی داشته باشد 200K را باید تقسیم کرد و توسط Over lays و سپس اجرا کرد.

Over lays :

7.1.4

از آنجا که میدانیم بایستی تمام فضای آدرس منطقی یك پردازش در حافظه فیزیکی باشد، قبل از اینکه بتوان آن پردازش را اجرا کرد ولی چنانچه فضای آدرس منطقی بیشتر از فضای حافظه فیزیکی تخصیص داده شده باشد می توان از تکنیکی بنام Over lays استفاده نمود. ایدهٔ Over lays به این صورت است که فقط آن دستورات و داده هائی که در یك زمان مورد احتیاج هستند داخل حافظه نمود. وقتی که به بقیه دستورات احتیاج شد بتوان آنرا در فضایی از حافظه که قبلاً توسط دستوراتی که دیگر مورد احتیاج نیست قرار داشتند جایگزین نمود به عنوان مثال ملاحظه کنید یك برنامه اسمبلر دو مرحله ای را که در مرحله اول یا Pass 1 یك جدول Symbol Table می سازد یا در Pass 2 کد زبان ماشین را تولید می کند و فرض کنید که اندازه های این اجزاء به صورت زیر باشند.

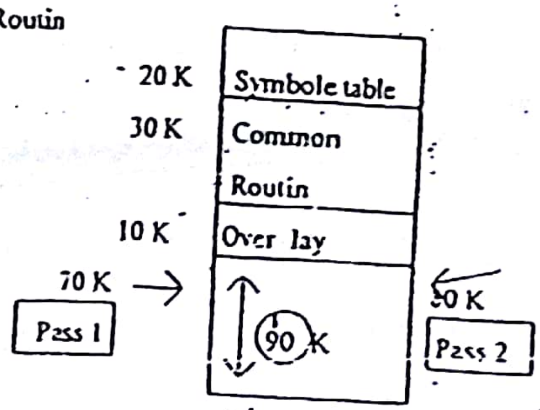
Pass 1	70 k
Pass 2	80 k
Symbol	20 k
Common Routin	30 k

pass1 , pass2 نیازی ندارند همزمان در حافظه باشند پس بنابراین دو overlay تعریف میکنیم
A جدول علائم، روتین های مشترک و pass1
B " " pass2

ولی حافظه فیزیکی موجود 150 k می باشد. از آنجا که Pass 1 , 2 مستقل از هم می باشند لذا نیاز نیست که همزمان در حافظه باشند لذا ما دو تا Over lays بصورت زیر تعریف می کنیم.

A : Pass 1 , Symbol table , Common Routin

B : Pass 2 , Symbol table , Common Routin



بزرگی اندازه برنامه میتواند بزرگتر از حافظه تخصیص یافته به آن باشد [illegible] ... over lay برای انجام ایده آن این است که تنها آن داده ها و دستورالعمل هایی در حافظه قرار دهیم که در هر زمان معین مورد نیاز باشند وقتی دستورات [illegible]

segmentation را عنوان کردیم

امتحان تا آخر Paging است

(فصل هفتم)

مهمترین فضا را اشغال می‌کند

مدیریت حافظه Memory Management

Address binding
Compile
Load
Execute

همانطور که در فصل 4 دیدیم که چگونه CPU می تواند توسط یکسری از پردازشها مورد استفاده قرار گیرد و همچنین می دانیم که برای اجرای Time Sharing , Muti programing بایستی چندین پردازش را در حافظه نگه داریم لذا بایستی حافظه را بصورت اشتراکی مورد استفاده قرار گیرد در این فصل روشهای مختلف مدیریت حافظه را از فرم ابتدائی تا Segmentation Paging را مورد بحث و بررسی قرار میدهیم .

7.1 مقدمه

بطور کلی حافظه یک آرایه بزرگ از Word یا byte است که هر کدام آدرس مخصوص خود را دارد و CPU از حافظه هم به عنوان ذخیره سازی و هم به عنوان بازیابی استفاده می کند .

1.1.7 Address Binding . محدود کردن

معمولاً یک پردازش روی یک Disk به صورت فایل اجرایی باینری قرار دارد و مجموعه ای از این پردازشها که روی دیسک منتظر هستند تا به داخل حافظه اورده شوند یک input Queue را تشکیل می دهند یک روش معمول این است که یکی از پردازشهای داخل input Queue انتخاب شده و داخل

حافظه Load گردد همانطور که پردازش اجرا می شود و یا داده ها و دستورات در حافظه تماس دارد و

در نهایت پردازش خاتمه یافته و حافظه استفاده شده برای آن پردازش آزاد می گردد .

آدرسها در برنامه منبع یا Source بطور کلی سمبلیک هستند مثل متغیر i که کامپایلر این آدرسهای

سمبلیک را bind (محدود می کند) می کند به آدرسهای Relocatable مثل 3 بایت از شروع ماجول

پردازش سپس loader به نوبت این آدرسهای قابل جابجایی . R به آدرسهای مطلق (مثل حافظه شماره

7050) یا Absolut یار می کند بطور کلی binding دستورات و data ها به آدرسهای حافظه می

تواند در یکی از مراحل زیر انجام گیرد .

۱- Compile Time

اگر پردازش در زمان کامپایل حافظه قرار گیرد پس کد مطلق یا Absolut تولید می شود .

۲- Load time

اگر پردازش در زمان کامپایل در حافظه قرار نگیرد پس کامپایلر بایستی کد Relocatable ایجاد نماید

در اینصورت binding نهایی تا زمان Load به تعویق می افتد پس اگر آدرس شروع تغییر نماید .

فقط کد را دوباره Reload می نمائیم تا تغییرات به حساب آید .

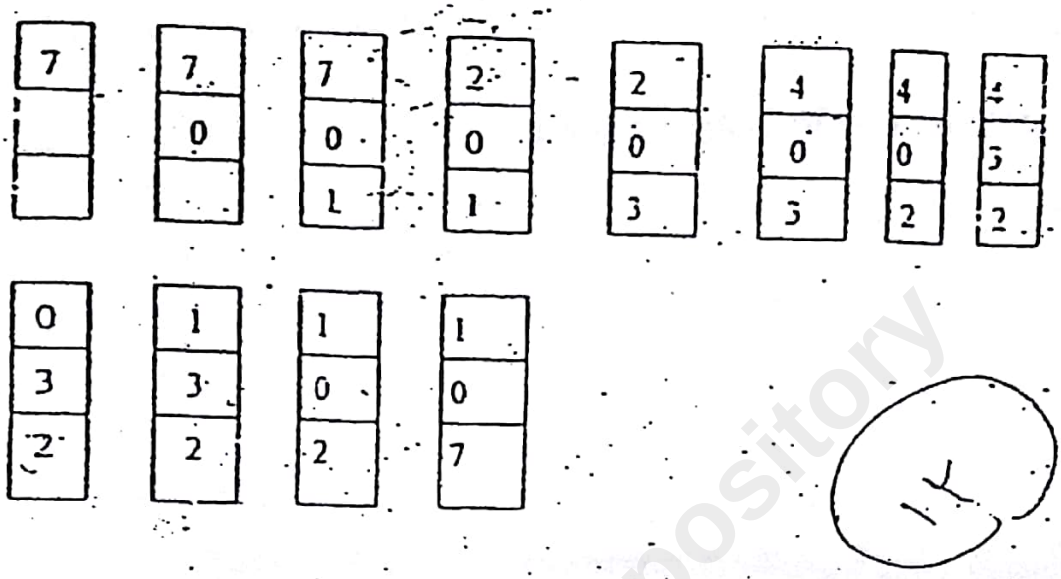
۳- Execution Time

اگر بتوان یک پردازش را در زمان اجرایش از یک بخش از حافظه به بخش دیگری انتقال داد پس

binding تا زمان Run یا اجرا به تأخیر می افتد در این مورد سخت افزار مخصوصی بایستی موجود

باشد تا بتوان آنرا اجرا نمود .

7.1.2 Dynamic Loading



طریق Plemeniation برای LRV

یک Counter به هر Page در هر Page table یک رجیستری که زمان استفاده از آن Page را نشان میدهد الحاق می شود و به CPU نیز یک کلاک یا کانتر اضافه می شود . به کلاک Memory refrence یک واحد اضافه می شود و وقتی که یک مراجعه به یک Page صورت می گیرد . لذا محتویات رجیستر کلاک به رجیستر Page استفاده شده کپی می شود .

در اینصورت همیشه تا زمان آخرین مراجعه به Page را داریم . لذا Page ای را که کمترین زمان را دارد باید Replace باشد .

پیاده سازی LRV

در این روش یک Stack از شماره Page ها نگهداری می شود و وقتی که به یک Page مراجعه صورت می گیرد در O.S بزرگ از آن روش استفاده می شود .

شماره اش به بالای Stack منتقل می شود . پس آخرین شماره موجود در ته

خواهد برد که بایستی از حافظه فیزیکی خارج شود. List -Recent- use و Stack

(۱) تعریف سیستم عامل و اهداف سیستم - چه ترکیبی به وجود آمدن که کارت های کنترل ، اولین سیستم عامل
ابتدایی و بافرینگ - spooling مزایای spooling ، وسیله های سخت افزاری انقلاب در سیستم
کامپیوتر شد - Multi tasking و Multipro -

(۲) مسائل امنیتی در سیستم عامل مطرح (۲ قسمت) . حافظه تحت اجرای - حافظه کمک ابزار
- Dual Mode - Timer

(۳) اجزای سیستم عامل (هشت قسمت) - مدیریت حافظه های جانبی و سرویس سیستم عامل در مقابل فایل
Sys call ، برنامه های سرویس که سیستم virtual machine

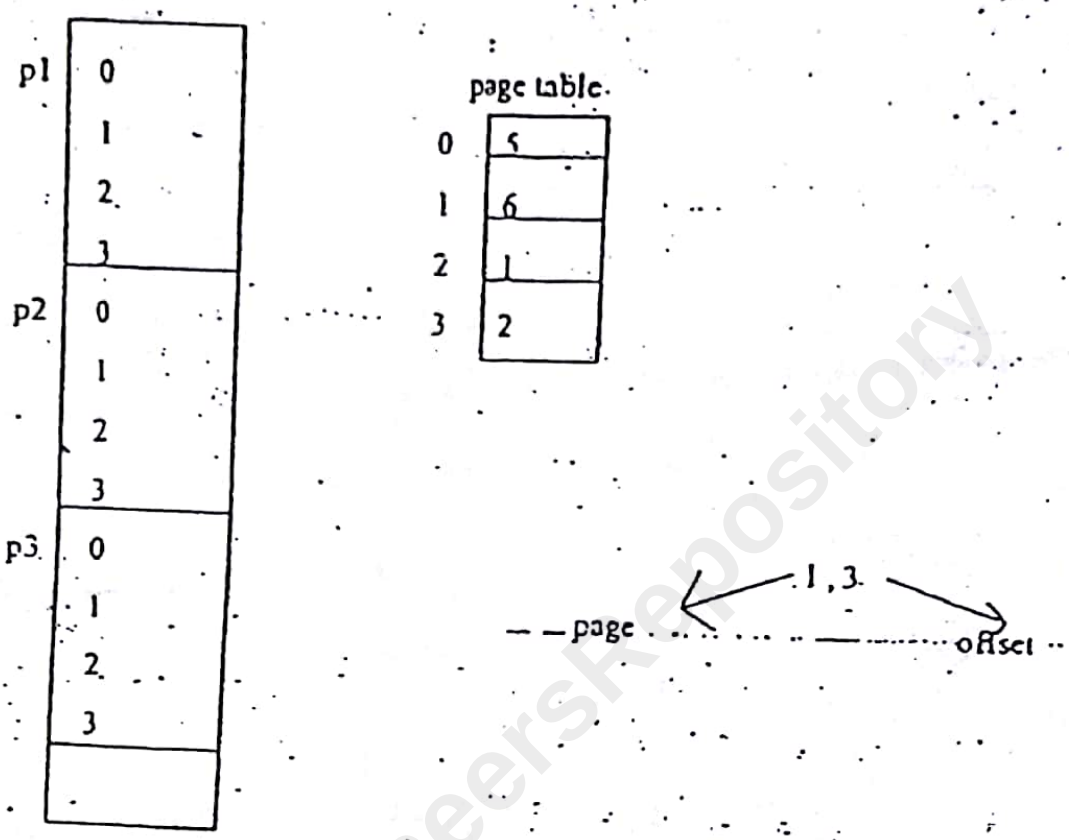
(۴) مدیریت پردازش - حالت های پردازش ، صف های پردازش ، structure صف ها
زمان بندی کننده ها ، PCB (process control block) ، الگوریتم های پردازش ، FCFS ، preemptive
SJF - RR ، معیار زمان انتظار و بیابید بهترین الگوریتم preemptive و non-preemptive

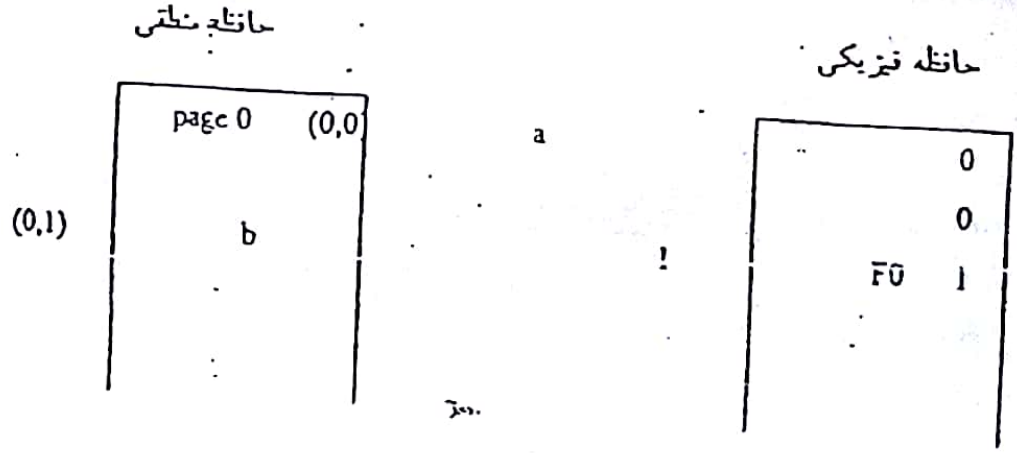
(۵) مسائل همزمانی ، الگوریتم ها ، customer ، الگوریتم اول برای بخش بحرانی پردازش ،
semaphore - حل مشکلات از سمافور ، تعریف سمافور و ...
...

(۶) مدیریت حافظه ، مدیریت آدرس حافظه swapping | Single partition APP ، Multi partition APP
paging ، compact ، segment + virtual memory ... الگوریتم ...

یا استفاده از اندازه Page که 4 word و یک حافظه فیزیکی که از 32 word (8 Frame) تشکیل شده است می توانیم نشان دهیم که چطور از حافظه منطقی می توانیم به حافظه فیزیکی برسیم؟

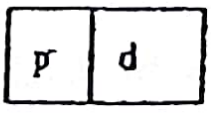
Frame و Page باید یک اندازه باشد.

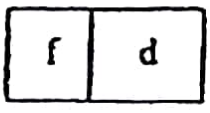




	(0,2)	c
	(0,3)	d
page 1	(1,0)	e
	(1,1)	f
	(1,2)	g
	(1,3)	h
page 2	(2,0)	I
	(2,1)	j
	(2,2)	k
	(2,3)	l
page3	(3,0)	m
	(3,1)	o
	(3,2)	p
	(3,3)	q

	2	2
	3	3
	4	4
		i
F1	5	j
	6	k
	7	l
	8	m
F2	9	n
	10	o
	11	p
		12
F3		13
		14
		15
		16
F4		17
		18
		19
F5	20	a
	21	b
	22	c
	23	d.
	24	e
F6	25	f
	26	g
	27	h
		28





7.5.2 پیاده سازی Page-Table

Page Table.

به سه طریق انجام می شود

۱ - در ساده ترین حالت اگر تعداد عناصر Page-table بطور قابل قبول کوچک باشد (حد اکثر 256 عنصر) از رجیستر ها می توان استفاده نمود .

۲ - چنانچه اندازه Page table خیلی بزرگ باشد (مثلا 1000000 عنصر) می توان p.t را در حافظه اصلی نگهداری نمود که یک PTBR یا Page table back register به اول Page table اشاره می کند . مشکل اصلی پیاده سازی Page table از طریق حافظه اصلی . زمان لازم برای دسترسی به یک محل حافظه و پردازش کاربر است .

۳ - راه حل استاندارد برای پیاده سازی Page table استفاده از یک سخت افزار حافظه کوچک بنام Associative Register است که شامل دو قسمت است :

(۱ - یک کلید ۲ - یک مقدار) وقتیکه یک Associative Register یک کلید را معرفی می کند هم زمان با تمام کلید ها مقایسه شده و چنانچه کلید مربوطه پیدا شود مقدار مربوطه مشخص خواهد شد . این جستجو بسیار سریع است Associative Register بطریق زیر توسط Page table مورد استفاده قرار میگیرد .

Associative Register شامل چندین Page table خواهد بود . وقتی که یک آدرس منطقی توسط CPU تولید می شود شماره Page آن در اختیار یک مجموعه از associative reg. ها قرار می گیرد که شامل page ها و شماره Frame مربوطه هستند اگر شماره page در associative reg. پیدا شد شماره frame مربوطه بلافاصله برای دسترسی به حافظه مورد نظر مورد استفاده قرار می گیرد . اگر شماره Page در Associative Register پیدا نشود یک مراجعه حافظه با reference به Page

table بایستی یافت شود . وقتی که شماره Frame بدست آید ما می توانیم از آن استفاده کنیم تا به حافظه دسترسی یابیم . به علاوه شماره Page و شماره Frame را به Associative Register اضافه می کنیم تا بتوانیم با سرعت بالاتر در مراجعه بعدی به آن دسترسی پیدا کنیم .

Fit Ratio :

در صدی از دفعات که از طریق Associative Register اطلاعات page را پیدا میکنیم

7 . 5 . 3 Shared Pages :

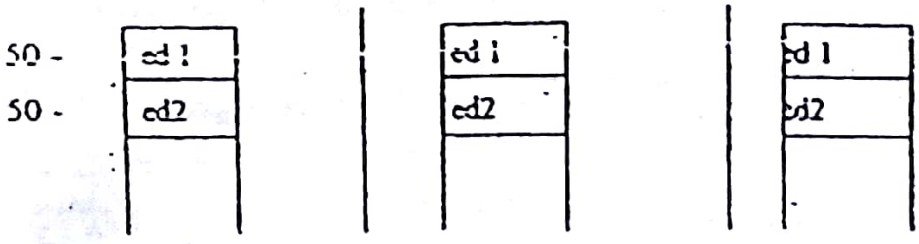
وضعیت دیگر Paging امکان مشترک شدن کد های کلی است . این مطلب مخصوصاً در مورد محیط Time sharing بسیار مهم است .

(text - Editor)= > 100* 200 = 20000 k = 2000 x 2 * 10

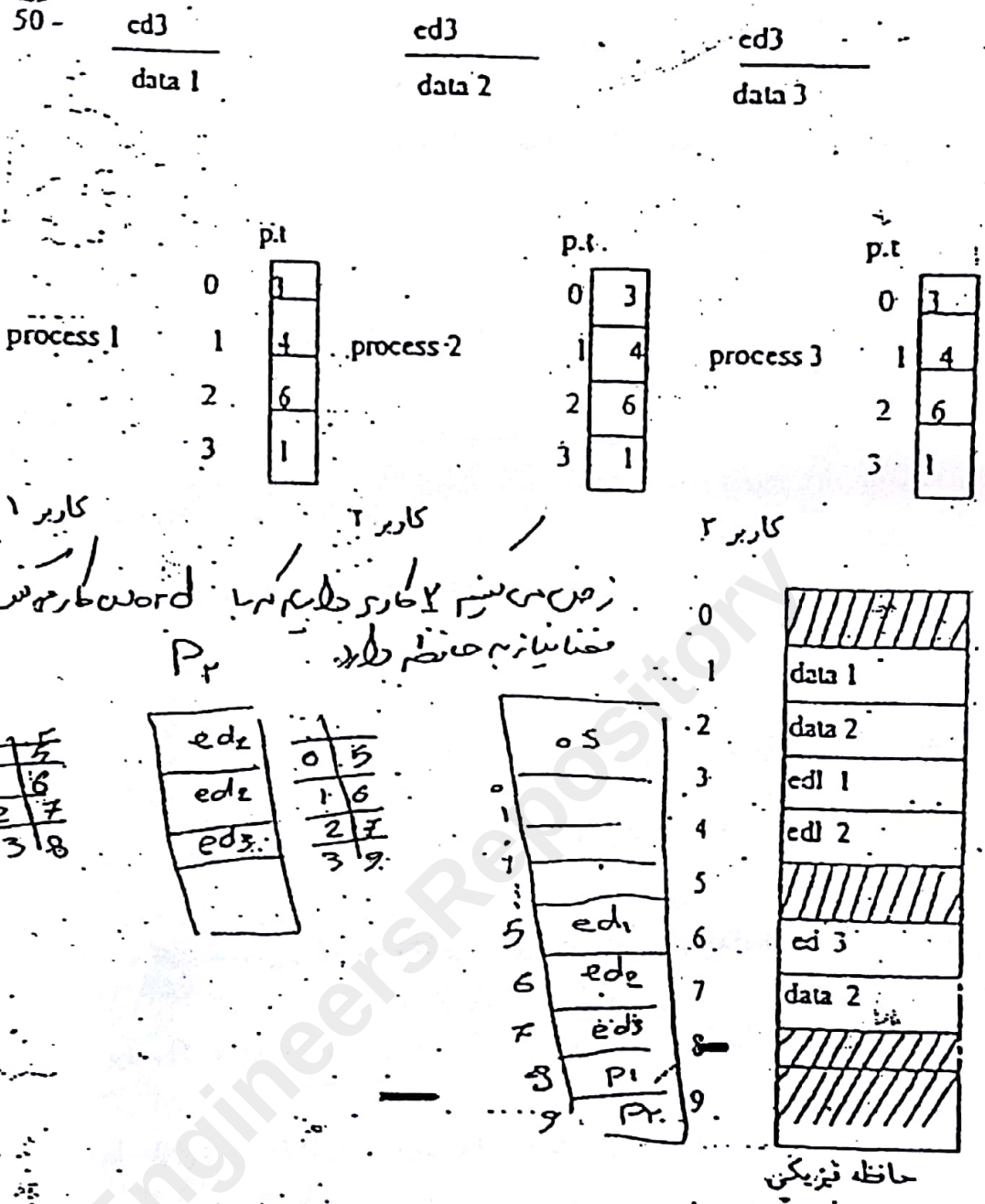
اگر ۱۰۰ ذخیره هر کدام ۲۰۰ - ۵۰ نیاز باشد

از حافظه باید مصرف شود text editor یک کپی بیشتر نمی گذارد و برای بقیه از Sharing استفاده می کند . [illegible] ملاحظه کنید سیستمی را که به ۴۰ کاربر سرویس می دهد و هر کدام از آنها Text Editor را بکار می برند . اگر Text Editor شامل 150 k کد و 50k فضای data باشد پس ما نیاز به 8000 k حافظه داریم تا به ۴۰ کاربر را سرویس دهد . ولی اگر کد (150 + 50 = 200 * 40 = 8000 k) باشد می تواند بطور مشترک مورد استفاده قرار بگیرد

یعنی بصورت زیر:



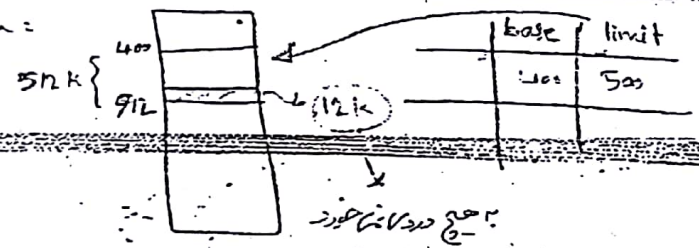
50 -



هر کدام Page table خاص خود را دارد که با Text Editor کار می کند.

Paging این خاصیت را دارد که اجازه میدهد برنامه را تقسیم بندی کنیم. ما از فضای حافظه استفاده نمی کنیم بلکه از Copy آنها استفاده می کنیم که Reentrant کدی است که از آن طریق می توان وارد برنامه شد که Reentrant کدی است که خودش را نمی تواند تغییر دهد و از هر نقطه ای می شود وارد برداشش که به صورت برداشش کپی از رجیستر های خودش و حافظه داده های مربوطه به خودش را دارا می باشد. هم چنین فقط یک کپی از Text Editor نیاز است که در حافظه فیزیکی

Internal Fragmentation :



نگهداری شود. لذا چنانچه ما جهت سرویس دادن به ۲۰ کاربر فقط نیاز است که یک کپی از Text Editor یعنی 40,150 k کپی از 50 k از فضای data را برای هر User می باشد. لذا کل فضای مورد نیاز 2150 k بجای 8000 k می باشد که یک ذخیره سازی یا صرفه جوئی بسیار زیاد در حافظه است.

7.7 Segmentation : مدیریت حافظه در سیستم های segment

در Segmentation یک فضای آدرس منطقی یک مجموعه از Segment ها است.

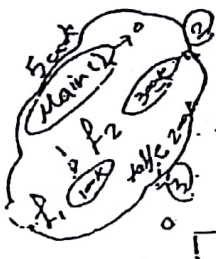
آدرسها هم نام Segment و هم offset داخل Segmentation را مشخص می کند. لذا کاربر هر آدرس را با دو مقدار مشخص می سازد :

۱ - یک نام Segment ۲ - Offset

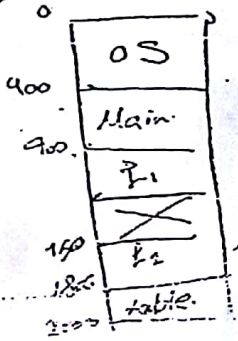
برای سادگی پیاده سازی Segment ها شماره گذاری شده و توسط شماره به آنها مراجعه می شود.

7.7.1 Segmentation : سخت افزار Hard ware

اگر چه کاربر می تواند به Object ها در برنامه آدرسی از آرایه دو بعدی را در نظر بگیرد. ول حافظه فیزیکی واقعی یک رشته از Word های آرایه یک بعدی است. لذا ما بایستی یک پیاده سازی را معرفی کنیم که آرایه دو بعدی را که از طرف کاربر معرفی می شود را به یک آدرس فیزیکی آرایه ای یک بعدی تبدیل نمائیم. این تبدیل یا Mapping با استفاده از Segment table ساخته می شود.



شکل زیر چگونگی استفاده از Segment table را نشان میدهد که در آن یک آدرس منطقی از دو

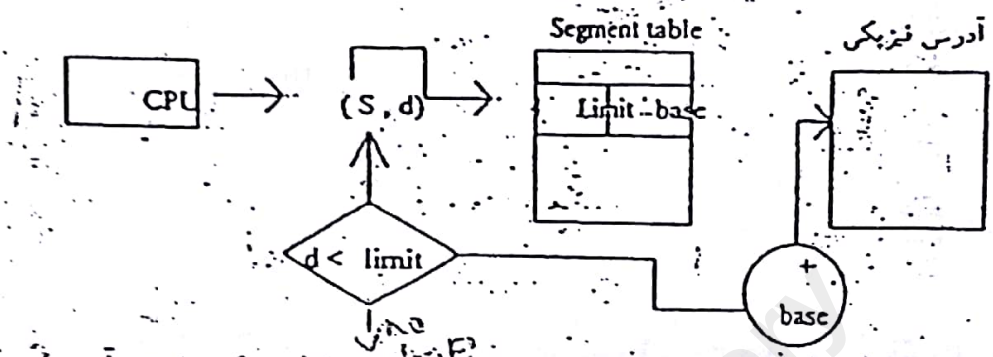


جدول segment table ساخته شده :

segment	base	limit
0	400	500
1	900	700
2	150	300
3	1800	200

عیب : [illegible] Page segmentation از این مشکل

۱ - شماره Segment (S) ۲ - یک Offset داخل Segment (d)



(اگر کمتر از Limit باشد . d را با base جمع میکند تا مقدار حافظه فیزیکی بدست آید .)

تبدیل آدرس لاجیکی از طریق Segment به آدرس فیزیکی

شماره Segment به عنوان اندیس Segment table بکار می رود . هر عنصر در Segment یک Segment base و یک Seg limit است . Ofiset مربوط به آدرس منطقی باید بین Seg . Limit و ۰ باشد چنانچه نباشد یک trap در O.S انجام خواهد گرفت .

(فصل هشتم)

Virtual Memory

تکنیکی است که اجرای یک پردازشی را که کاملاً نمی تواند در حافظه استقرار یابد اجازه می دهد

فایده اصلی این تکنیک آن است که برنامه ها می توانند از حافظه فیزیکی بزرگتری برخوردار باشند.

البته تجربه نشان داده است که در بیشتر موارد احتیاج نیست که همه برنامه ها در حافظه باشد از جمله :

۱ - برنامه ها برای کد های عیب یابی هستند که همیشه این کد ها مورد استفاده قرار نمی گیرند.

۲ - آرایه ها ، جداول و لیستها همیشه بیشتر از موارد مورد نیاز تعریف می شوند . لذا ضروری نیست

که همه خانه های حافظه برای آنها در نظر گرفته شود . پس از اینکه بتوان برنامه ای را اجرا نمود

قسمتی از آن در حافظه وجود دارد که این فرایندی دارد از جمله :

۱ - برنامه هایی که دیگر توسط مقدار حافظه محدود نمی شوند .

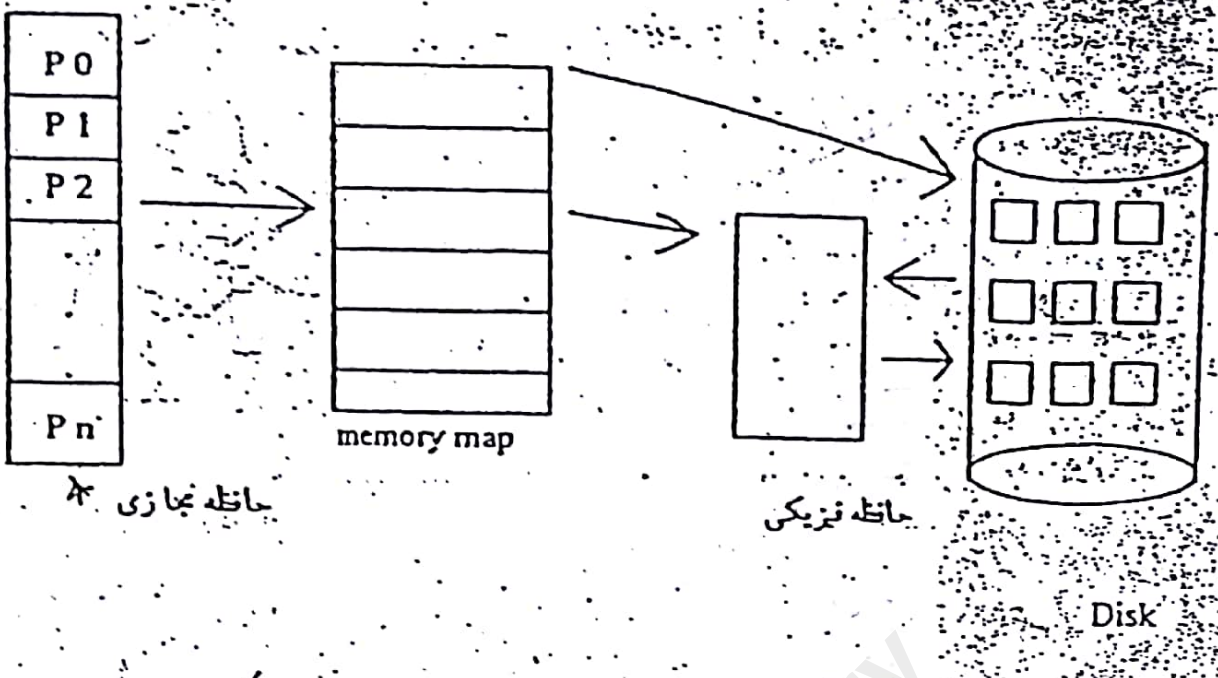
۲ - هر برنامه میتواند حافظه کمتری را اشغال نماید . لذا درجه Multi programing بالا می رود . در نتیجه بهره وری از CPU بیشتر می شود

۳ - I/O کمتری برای Load شدن و یا Swap شدن مورد نیاز است . لذا برنامه ها سریعتر انجام

می شود . لذا اجرای یک برنامه که کاملاً در حافظه فیزیکی نباشد ، هم برای کامپیوتر مفید است هم

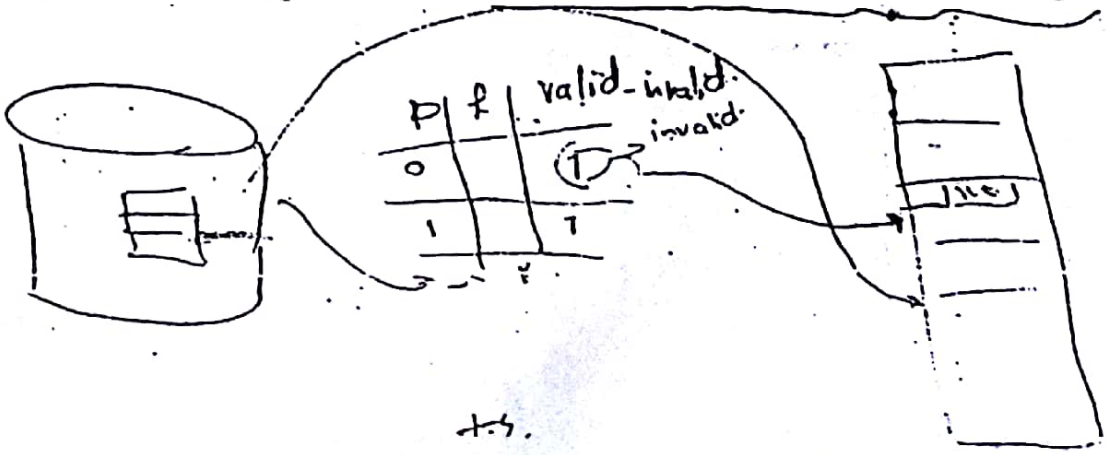
برای User . در واقع Virtual Mem جدا سازی حافظه منطقی کاربر از حافظه فیزیکی است که در

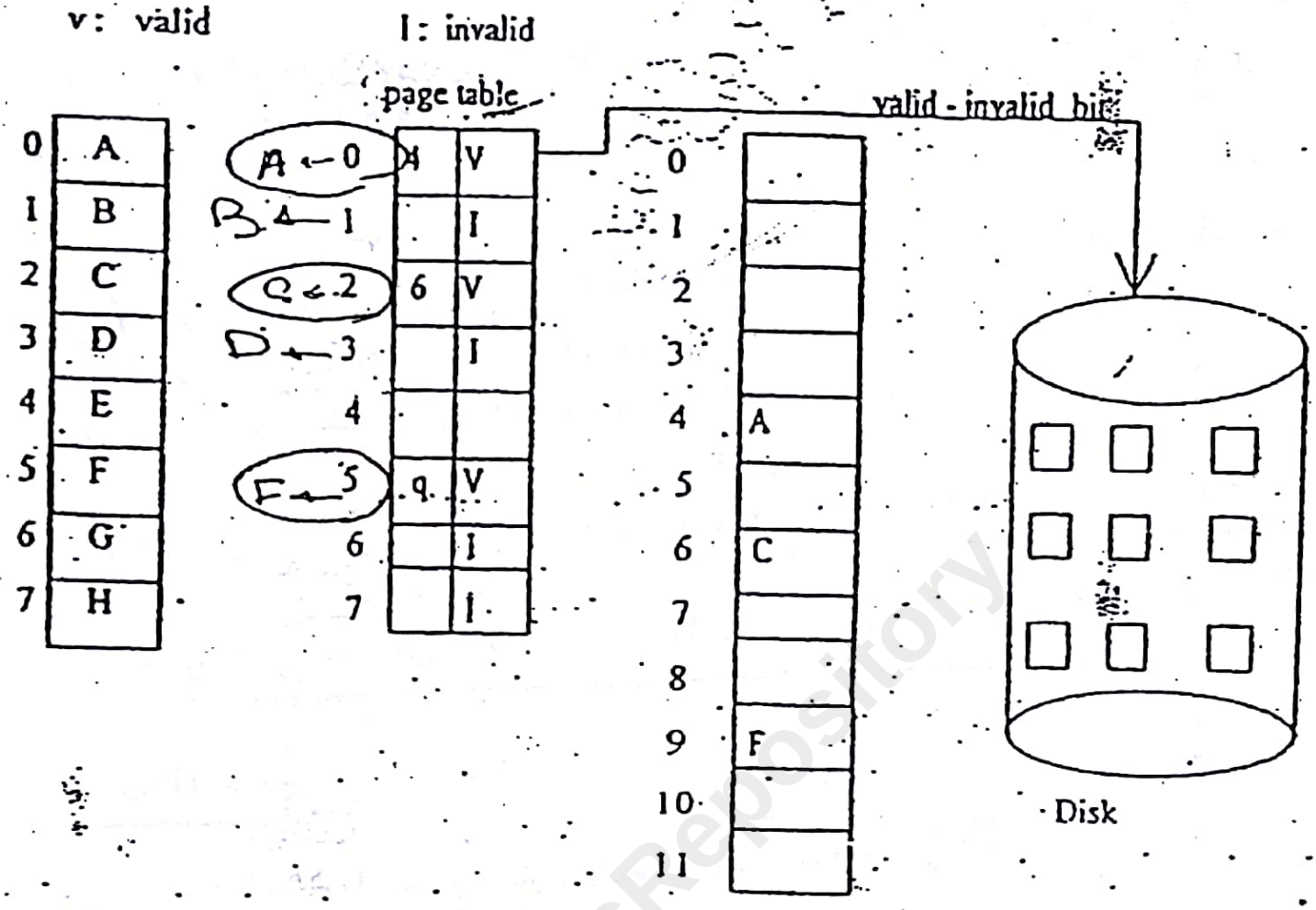
شکل زیر نشان داده شده است .



Demand Paging: (صفحه بندی سازی)

یک D.P نیز Paging sys با swaping پردازشها روی دیسک قرار می گیرند. وقتیکه یک پردازش بایستی اجرا شود ما آنرا داخل حافظه Swap می کنیم اما بجای اینکه همه پردازش را داخل حافظه Swap کنیم در تکنیکی بنام Lazy Swaper استفاده می کنیم. یعنی این که یک Page را داخل حافظه Swap نمی کنیم مگر آنکه آن Page مورد نیاز باشد. طرح Demand page احتیاج به پشتیبان سخت افزاری دارد. لذا یک بیت به هر عنصر Page table بنام Valid - invalid bit اضافه می شود. وقتی بیت Valid است یعنی Page مربوطه در داخل حافظه است و وقتی Invalid است یعنی page مربوطه روی دیسک است.





حافظه فیزیکی

اجرای این پردازش بصورت نرمال ادامه پیدا می کند. ولی چنانچه Page درخواست شده در حافظه

نباشد Page fault Trap اتفاق خواهد افتاد. در واقع O.S. بر آوردن Page بداخل حافظه از

تخنین استفاده می کند. لذا مراحل زیر جهت آوردن اطلاعات بداخل حافظه فیزیکی انجام می شود.

۱ - نخست جدول Page table را چک می کنیم تا بفهمیم که مراجعه Valid است یا Invalid.

۲ - اگر Invalid بود پردازش را یا خاتمه می دهیم و یا اینکه Page مربوطه را بداخل حافظه انتقال

می دهیم بطریق زیر:

۳ - یک Frame آزاد را پیدا می کنیم.

۴ - Page مربوط را داخل Frame, Load می کنیم.

۵ - وقتیکه کار فوق تمام شد جدول داخلی را تغییر می دهیم که نشان دهیم Pager داخلی حافظه

است.

۶ - پس دستورات را از اول شروع (اجرای دستوری را که بر عکس آدرس دهی بر صفحه نامعتبر را داشت و منجر به وقفه شده بود را مجدداً) از سر می گیرد. در مرحله سوم ممکن هیچ frame آزادی در حافظه اصلی وجود نداشته باشد در این حالت یکی از صفحات استفاده شده را بر دیسک منتقل می‌کنیم تا فضای خالی ایجاد شود

در نتیجه در یک حالت افراطی ما می توانیم یک پردازش را شروع به اجرا کنیم بدون اینکه هیچ صفحه‌ای از آن در حافظه باشد (صفحه بندی نیاز محض)

Pageای از آن در حافظه باشد. این روش بنام Pure demand paging نام دارد. یعنی یک

Page را داخل حافظه نکن، مگر اینکه در خواست شده باشد و سخت افزار آن مانند Demand (page replacement)

Page است یعنی

۱ - Page table: این جدول دارای یک بیت Valid، Invalid می باشد

۲ - حافظه دوم (دیسک): دیسک فضای مورد نیاز برای اجرای Swapping

:Page - replacement { FIFO, optimal, last Recently

اگر ما درجه Multi program را بالا ببریم برای اینکه پردازشهای بیشتری را به حافظه بیاوریم

به این عمل جایگزین کردن Page یا Over allocating Memory (تخصیص بیش از حد حافظه)

گویند.

Over allocating Memory: بشرح زیر انجام می شود:

- در حالیکه یک پردازش کاربر در حال اجرا است یك Page fault (صفحه ای که در حافظه فیزیکی نیست) و Page fault trap اجرا می شود، سیستم عامل مشخص می کند که Page مربوطه روی دیسک موجود است یا خیر اما در می یابد که هیچ Frame ای آزاد یا خالی پیدا نمی کند یعنی تمام حافظه فیزیکی مورد استفاده قرار گرفته است

(OS) چند راه حل برای این موضوع دارد :

۱ - می تواند پردازش را خاتمه یافته تلقی کند .

۲ - یک پردازش دیگر را Swap out نماید یعنی درجه Multi programing را کاهش دهد

۳ - از Page - replacement استفاده نماید . در اینجا حالت سوم را توضیح می دهیم :

اگر هیچ Frame ای آزاد نباشد یک Frame ای را انتخاب می کنیم که در حال حاضر مورد استفاده قرار نگرفته لذا ما میتوانیم یک frame را انتخاب نموده وآنرا روی دیسک بنویسیم یعنی Frame Page table ، از مربوطه را Invalid نموده و مشخص می کنیم که آن Page دیگری حافظه وجود ندارد . مثلا 9 مربوط به F است ، F را از حافظه فیزیکی پاک و در دیسک می گذاریم و Page دیگری از پردازش دیگری که لازم است به جای آن می گذاریم .

Page- replacement : اساس demand- paging است یعنی آن جدایی بین حافظه منطقی و حافظه فیزیکی را کامل می کند

انگوریتمهای Page-replacement

الگوریتمهای مختلفی وجود دارد که بهترین آن الگوریتمی است که کمترین Page fault را داشته باشد

برای ارزشیابی یک الگوریتم Page.replasement آنرا با یک رشته از مراجعات به حافظه اجرا می کنیم

و تعداد page Fault را محاسبه می کنیم که به این رشته Reference-Stering می گویند برای اینکه تعداد داده ها را برای چک نمودن الگوریتم کم کنیم دو مطلب زیر را در نظر می گیریم .

۱ - برای یک Page-Size (اندازه Page) داده شده ما فقط شماره Page را در نظر میگیریم نه آدرس آنرا .

۲- اگر ما یک مراجعه به Page داشته باشیم پس هر مراجعه بعدی که بلافاصله انجام شود هرگز موجب Page Fault نخواهد شد.

به عنوان مثال اگر ما اجرای پردازش به خصوص را تعقیب نمائیم ممکن است رشته آدرسهای زیر را ثبت کنیم:

0100 , 0432 , 0101 , 0102 , 405

اگر هر صفحه 100 بایت باشد می توانیم reference String (مراجعه به رشته) را به اعداد زیر تبدیل کنیم:

از 0-99 ---- 0

تا 100 ‑ 199 —— 1

تا 299 —— 2

توجه: هر قدر که تعداد Frame ها بیشتر باشد تعداد Page fault ها کمتر است. برای توضیح دادن الگوریتم های زیر از reference string زیر استفاده می کنیم:

اعداد از 0 تا 7 که بزرگتر نداریم پس نیاز به ۸ فریم داریم

7 , 0 , 1 , 2 , 0 , 3 , 0 , 4 , 2 , 3 , 0 , 3 , 2 , 1 , 2 , 0 , 1 , 7 , 0 , 1

و تعداد Frame ها را برای تمام الگوریتم ها ۳ در نظر می گیریم.

۱- الگوریتم FIFO : فرض: هر page، 100k است

با فرض اینکه ۳ فریم داریم و خالی

یعنی قرار است یک Replace Page شود = قدیمی ترین آنها را در نظر می گیریم

چون جزء وجود دارد جایگزین نمی شود، قبلاً هم وجود داشت

page fault به اینها می گویند

0	7	7	7	2	2	2	4	4
1	-	0	0	0	3	3	3	2
2			1	1	1	0	0	0

* اعدادی که می خواهند وارد شوند باید چک شود که قبلاً وجود داشته یا نه و اگر وجود داشته باشد در Frame فعلی دیگر جایگزین نمی شود

0, 2, 3 را باید جایگزین کنیم، هر چون 2 و 3 را داریم جایگزین نمی شود

ازآنجا که page fault داریم و باید FIFO طبق victim

4	0	0	0	7	7	7
2	2	1	1	1	0	0
3	3	3	2	2	2	1

Memory Reference

مثلاً 7، 0، 1، 2، 3 و ...

100K 0 → 99 → 0 page

100 → 199 → 1

200 → 299 → 2

تا 15 page fault اتفاق افتاده ولی در واقع 12 تا است و ما 15 می گیریم چون اولی ها از ابتدا وجود نداشته

page ... valid / invalid ... هستند

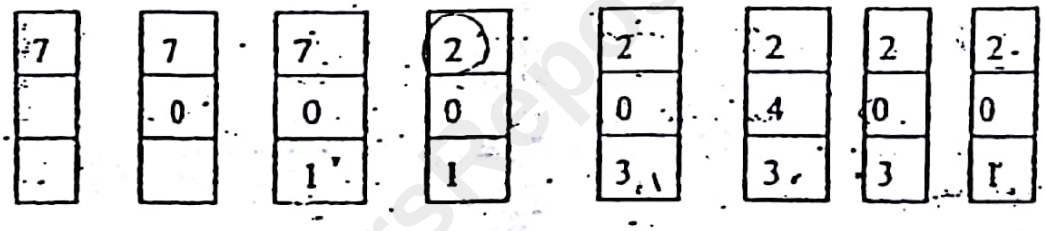
15 تا Page fault اتفاق می افتد . مشکل را داریم می کنیم اول کسانی آمده خارج می شود .

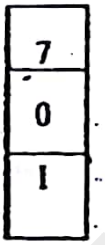
۲ – آلگوریتم Optimal : بهینه ترین حالت که جهت اجرایی ندارد تا اجرا نکنیم بهینه مشخص نمی شود

کمترین Page fault را خواهد داشت .

طرز کار : جایگزین کن Page ای را که برای مدت طولانی مورد استفاده قرار نخواهد گرفت در

تهیه . در این جا ۹ تا Page fault داریم .





۳ – آلگوریتم LRU یا Least Recently Used

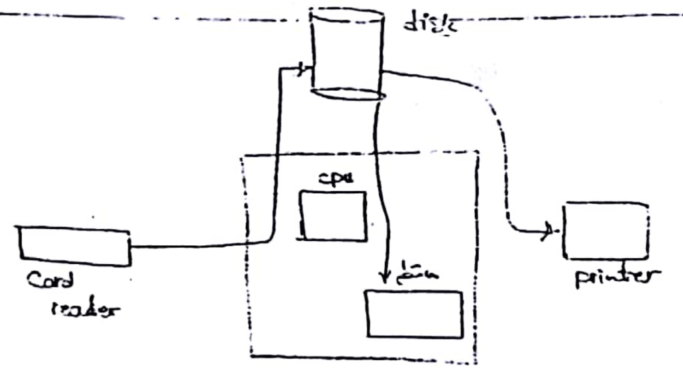
مهترین آلگوریتم است چون بیشتر سیستمهای بزرگ از آن استفاده می کنند .

"یعنی اینکه اخیرا کمتر مورد استفاده قرار گرفته"

و در اینجا 12 - Page fault داریم .

ضمیمه

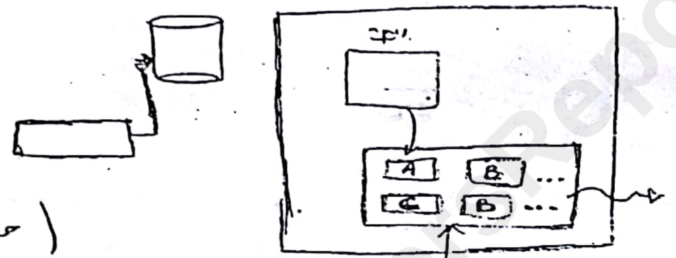
Spooling :



+ Spooling : overlap کردن I/O در یک job با محاسبات یک job دیگر

در صورت کلی { 1) شاهی
2) تجاری

Multi programming : اجرای همزمان چندین job



یک کار بسته شود . cpu در حداقل زمان بیکار بماند .
الگوریتم‌ها که برنامه‌ها در آن قرار دارند .
هر چه تعداد jobها بیشتر باشد ، زمان بیکاری cpu کاهش پیدا می‌کند .

(جلسه دوم)

بانک خرید و فروش فروشگاه

multiProgram = اجرای همزمان چندین برنامه

فرق با Pc = cpu ، multi user ، بیت ، intractive ، ارتباط متقابل ، online

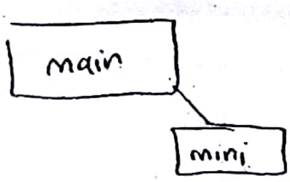
مولتی پروگرمینگ آن لاین نیست . offlin ← کارنامه

اگر multi Programing ، on line باشد مولتی tasking گفته می‌شود .

Timesharing نام دیگر

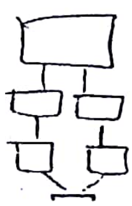
Time shar ← یعنی time ، تقسیم کردن cpu

multi tasking ، ادامه multi Programing است .



اشتراک ندارند

اولین نوع Losely وابستگی ندارد
مثال internet



به هم اشتراک دارند

دومین نوع Tightly
وابستگی زیاد

← اولی

ساختار کامپیوتر :

۱) interrupt-base data transfer

۲) مد دوتایی Dual Mode

۳) Hardward protection

Busy waiting : باعث می‌شود که cpu که کارش محاسبات است، کارایی‌اش پایین‌تر شود.

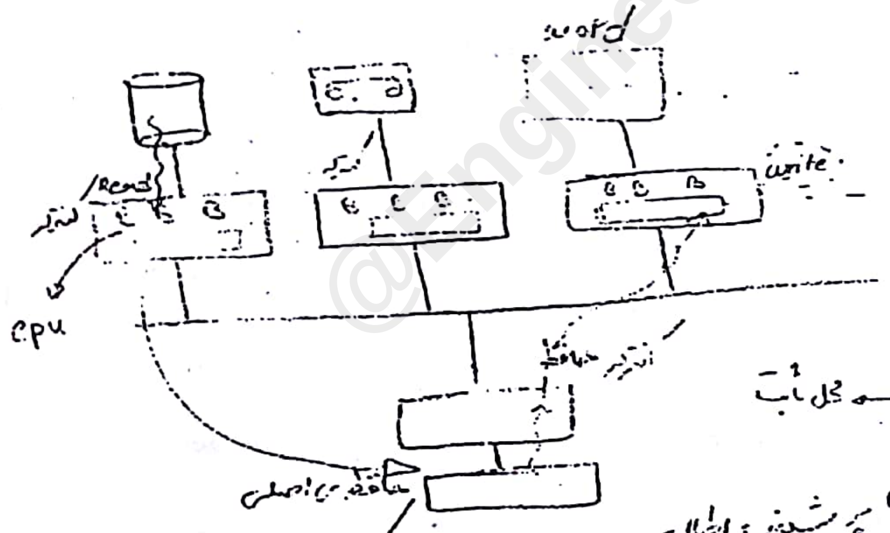
P1 — Read ... write

P2 — Read ... write

مشکل: cpu مشغول انتظار است و cpu باید به کارهای دیگر هم برسد.

interrupt-base : هر دستگاه سخت‌افزاری یک کنترلر وصل می‌شود و آن دارای رجیسترها می‌باشد. (دیسک، فلاپی و ...)

کنترلر ← register ، buffer



ISR : Interrupt service routine

داده از دستگاه وارد بافرها می‌شود، وقتی بافرها پر شدند اطلاعات به CPU می‌دهند ... بافر پر شده است، CPU کارش را انجام می‌دهد.

در نتیجه busy waiting را برداشتند که منتظر تمام شدن IO نشود.

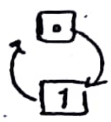
* مواردی که نیاز به محافظت دارند:

1) دستوراتی که به عملکرد صحیح سیستم لطمه می‌زنند ← Dual mode

2) جلوگیری از دسترسی به مکان‌های غیرمجاز حافظه ← base & limit register

3) جلوگیری از به loop افتادن ← timer

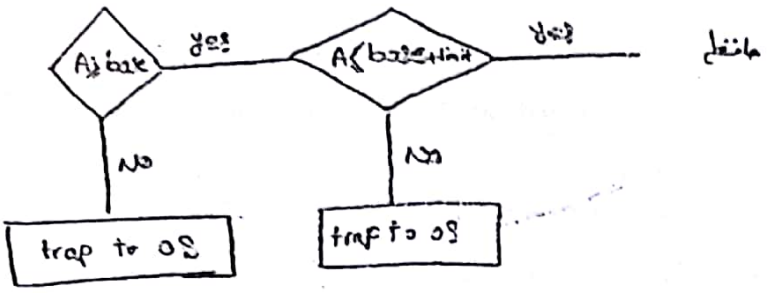
اگر برنامه، داده‌های مربوط به دیگری را بخواند، دستور trap to OS (تله سیستم عامل) صادر می‌شود.



اگر هر برنامه داده‌ی مربوط به خودش را بخواند و همه چیز درست باشد، mode تغییر می‌کند.
سیستم کار خودش را درست انجام می‌دهد mode از 0 به 1
برای رکورد بعدی 1 به 0 و دوباره تکرار می‌شود.

استفاده از این روش هزاران نفر می‌توانند با هم کار کنند، بدون اینکه اطلاعات هم را بخوانند.

trap یک وقفه نخت افزاری است.



Counter : صفر برند ، یعنی برنامه در loop است . چون اگر به صفر میرسد یعنی برنامه کارش را انجام داده و تمام شده است.

timer → Counter=300 / clock : ساعت

یک ساعت 1s است.

می ممکن است ، برنامه در loop نباشد ، اما تعداد دستورها بیشتری نیاز باشد که 300 s برای آن کم است . بنابراین بین تر Counter را بیشتر کنند.

اجزای سیستم عامل :

- مدیریت پردازش
- مدیریت حافظه (اصلی) پردازش
- مدیریت حافظه
- مدیریت فایل
- سیستم I/O
- محافظت
- شبکه
- مفسر دستور

مثل : پرینتر spooling — برای اولویت بالاتری نسبت به سیستم

✓ process → سیستم
→ user — برای اجرای برنامه‌های کاربران — کاربر

پردازش‌هایی که برنامه خودشان برای انجام کردن شدیدا (اغلب) سروکار دارد (پردازش‌های سیستمی می‌نویسند).

برنامه باید به حافظه برود و cpu به آن تعلق بگیرد ؛ پردازش می‌شود.

پردازش active یعنی برنامه به حافظه برود.

زای سیستم عامل : مدیریت پردازش

مدیریت حافظه

مدیریت حافظه دوم (کمکی)

I/O sys managment

مدیریت فایل

حفاظت

سیستم

مفسر دستور

پردازش : برنامه‌ی در حال اجرا با Program Counter را گویند که نشان دهنده دستور بعدی است که باید توسط cpu اجرا شود.

برنامه‌ای که منابع را مصرف می‌کند پردازش گویند. منابع مثل : حافظه، cpu، ...

اگر داده‌ای روی دیسک باشد پردازش نیست چون که منبع مصرف می‌کند و نه فعال است.

پردازش:
- سیستمی : مثل compiler، spooling
- غیرسیستمی : هر چیزی که user به کار ببرد و سیستمی نباشد. مثل معدل گیری

... : با مقدار کم بیشترین output را داشته باشیم. در مورد کامپیوتر یعنی بیشترین پردازش را در حافظه‌ای کم جا بدهیم.

... تشکیل می‌شود از آرایه‌ی یک بعدی که شامل خانه‌هایی است.

... به خودی خود وجود خارجی ندارد. فایل یک نگاه منطقی به داده‌ها و برنامه‌هاست.

دستگاه نیاز به برنامه‌ی راه‌انداز دارد.

Device Driv:
- General Purpose : برای مصارف عمومی
- Specific Device : دستگاه‌های خاص هستند مثل ploter ها

حفاظت:
- خارجی : کسانی که از بیرون می‌خواهند وارد main frame شوند باید username, pass داشته باشند
- داخلی : برای حفاظت از Dual Mode، برای از sleep از Timer استفاده می‌شود.

مفسر دستور:
- در حافظه قرار نمی‌گیرد.
- در حافظه قرار می‌گیرد.

ص 38

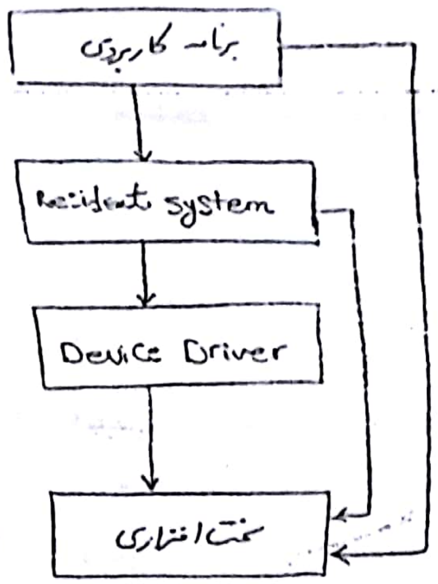
ماشین‌های مجازی { virtual window ... منابع را شناسایی می‌کند و ما میزانی که می‌خواهیم استفاده کنیم را به آن می‌دهیم ...
... OS با آن مشخصات برای ما جدا از OS اصلی درست می‌کند

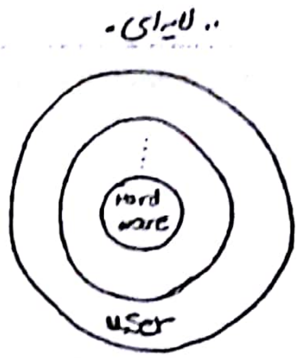
host vm

نتیجه: virtualization باعث می‌شود که پشتیبانی بالا برود و چندین OS را در کنار هم داشته باشیم.
وابستگی به سخت‌افزار را بسیار کم می‌کند و قابلیت حمل دارد.

SVm → Server virtual Machine
PVm → process virtual Machine

37 برای مث





6) Layer5 Application program (user)
↓
5) Layer4 Buffering for input & output
↓
4) Layer3 operator Console Device Driver ← I/O مثل OC دستگاه ورودی
↓
3) Layer2 Memory management
↓
2) Layer1 Cpu scheduling
↓
1) Layer0 hard ware

output لایه‌ی زیرین ، و inputs لایه‌ی بالایی است.

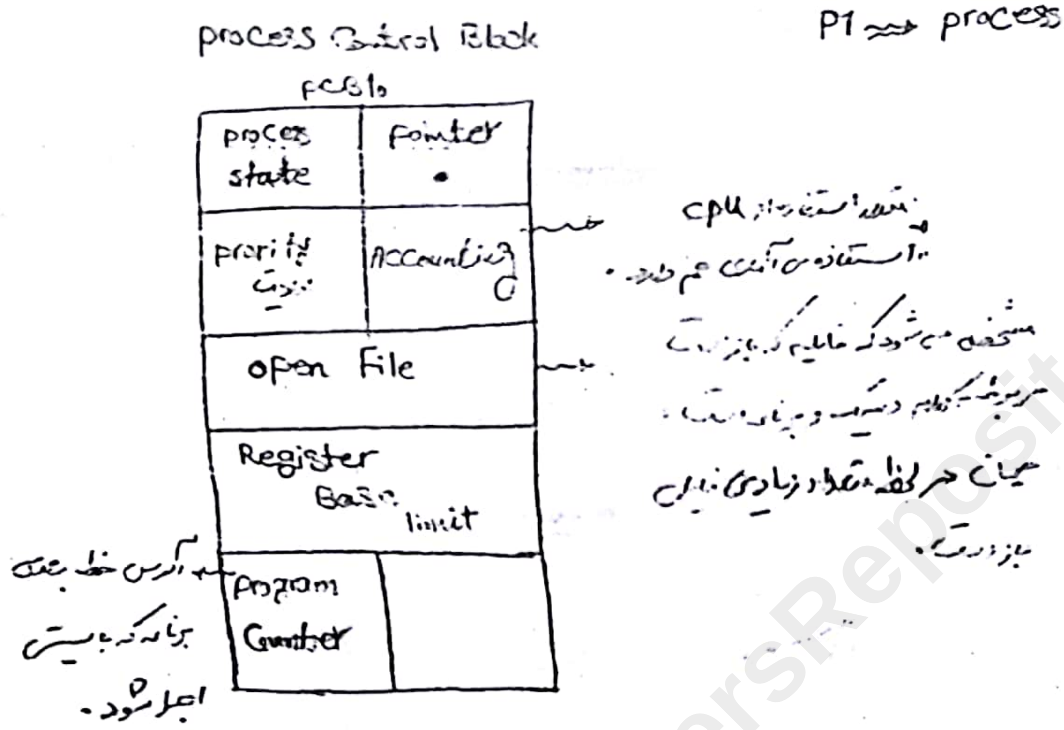
لایه‌ی 3 و 4 را با هم ترکیب کردند و نام آن شد ← MICRO Code
↓
چون اگر این‌ها جابه‌جا می‌شدند ایجاد مشکل می‌کردند.

MVS و Vm هر دو سیستم عامل‌اند.

MVS ← سه لایه‌ای کار می‌کند.

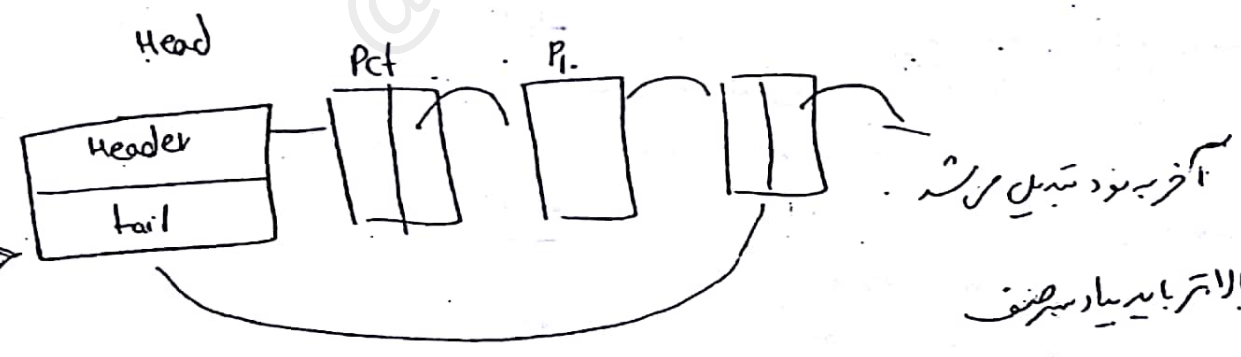
Vm ← نوشتن برنامه‌اش با MVS فرض می‌کنیم تشکیل شده مثلاً از 100 تا

P1 ⇝ process Control Block!



Prossese Control Block = PCB

* حجم) بر اساس صف = زمانی که حق تقدم بالاست و از link list استفاده می‌کنیم



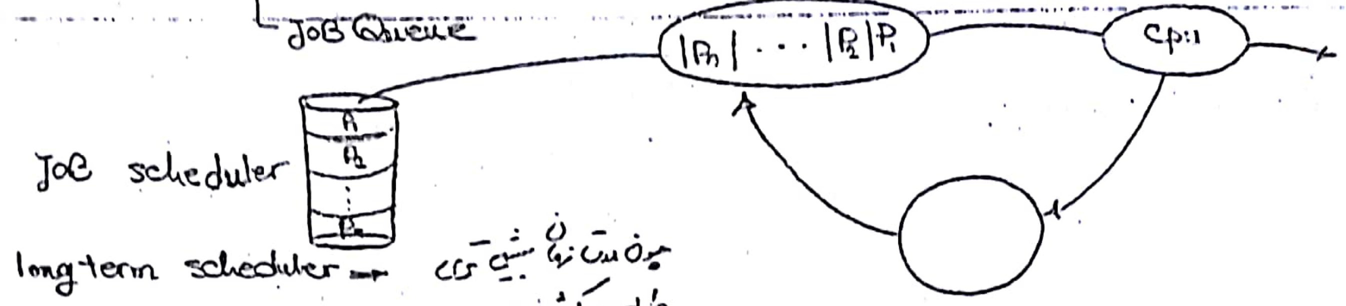
P_1 حق تقدم بالاتر باید بیاد سر صف

زمان بندی کردن :

زمان بندی کوتاه مدت ← Short term scheduling

برنامه‌هایی که چون به حافظه جا نشده در صف قرار گرفته اند.

وارد حافظه شده اند



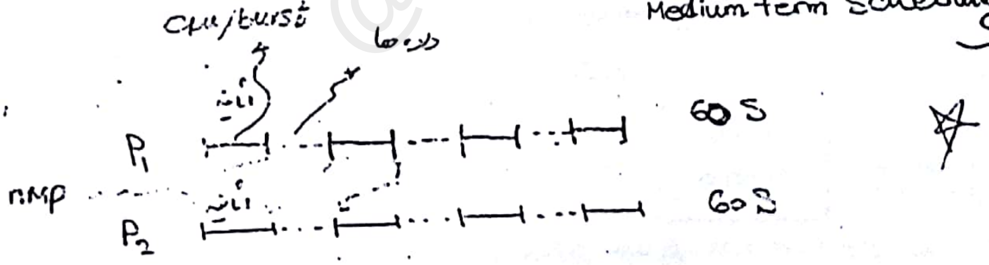
برنامه‌ها:
- علمی ← بیشترین درگیری برای CPU را دارند. محاسبات زیاد و I/O کم. مثل محاسبات پل ← مهندسین
- تجاری ← محاسبات کم و I/O زیاد. مثل حقوق و دستمزد ← کوبول - پاسکال

JOB scheduler باید بیشتر برنامه‌های تجاری و علمی را تلفیق کند تا از پایین آمدن کارایی CPU جلوگیری کند. و بدین ترتیب CPU و دستگاه‌های I/O مشغول نگه داشته می‌شوند.

زمان بندی میان مدت ← Medium term scheduling ← Multi tasking

زمان بندی کوتاه مدت ← Short / long term scheduling ← Multi programming

برنامه‌هایی که در Multi tasking باعث کند شدن اجرا می‌شود، آن را از CPU به دیسک می‌برد تا کارایی سیستم افزایش پیدا کند. ← وظیفه‌ی Medium term scheduling

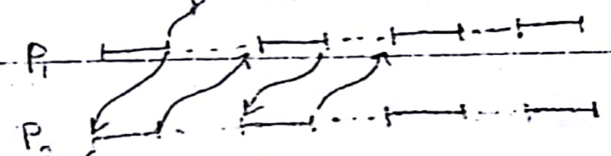


Multiprogramming ⇒ 120 S

none Multiprogramming ⇒ 61 S ← زمانبندی بهتر، چون بیکاری CPU کم شده و چون الان هر ثانیه برنامه‌ی دیگر اجرا شد.

هدف زمانبندی: طرح برنامه‌ریزی کنیم که CPU بیشترین کارایی را داشته باشد.

1- Context switch:



2- dispatch = اعزام کردن → cpu را در اختیار برنامه می‌گذارد

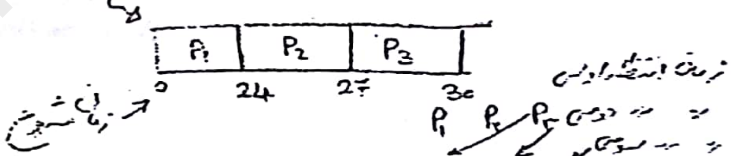
dispatch {
1) Context switch → در mode سیستم
2) switch to user mode
3) jump to program Counter

الگوریتم First Come First service:

I/O - burst : کار نمی‌کنیم، چون مربوط به سخت افزار است.

پردازش	cpu burst
P_1	24
P_2	3
P_3	3

Gantt chart



$$\text{average waiting time} = awt = \frac{0+24+27}{3} = \frac{51}{3} = 17$$

برای کامپیوتر فرقی نمی‌کند ولیکن برای برنامه‌هایی که کوتاه‌اند اجرا شوند، تفاوت است. چون خیلی طول نمی‌کشد.

* این الگوریتم بستگی به شرایط دارد که خوب یا بد باشد.

به ترتیب باشد ← ۳+۰+۶

۳
۳
۲٤

اگر P_1 آخر باشد

$$awt = \frac{0+3+6}{3} = \frac{9}{3} = 3$$

$\log ex$ = این به درد ما نمی‌خورد

= shift

بهترین حالت اینست فرض کنیم می‌توانستیم زمان را محاسبه کنیم

که این کار ممکن نیست و (پیش‌بینی می‌شد)

الگوریتم shortest job First :

« نمی‌تواند مورد استفاده قرار گیرد »

SJF { preemptive / Nonpreemtive

پردازش	cpu burst	
P_1	6	②
P_2	8	④
P_3	7	③
P_4	3	①

$$\frac{\overset{P_1}{0} + \overset{P_2}{3} + \overset{P_3}{9} + \overset{P_4}{14}}{4} = 7$$

SJF از کمترین شروع می‌کنیم

SJF Pr

SJF N Preem / None

	cpu	arrival time
P_1	8 (8-1=7)	0
P_2	4 (4-1=3)	1
P_3	9	2
P_4	5	3

Gant chart SJF

P_1	P_2	P_4	P_1	P_3
0–1	1–5	5–10	10–17	17–21

$$awt = \frac{\overset{P_1}{(10-1)} + \overset{P_2}{(1-1)} + \overset{P_3}{(17-2)} + \overset{P_4}{(5-3)}}{4} = 6.5$$

1 = NoN Pre

P_1	8	0
P_2	4	1
P_3	9	2
P_4	5	3

P_1	P_2	P_4	P_3
0–8	8–12	12–17	17–21

زمان پیش‌بینی ...

$$\frac{\overset{P_1}{0} + \overset{P_2}{(8-1)} + \overset{P_3}{(17-2)} + \overset{P_4}{(12-3)}}{4} = 7.72$$

همیشه P بهتر است از NP

P_1 اولی → 0 تا 1

P_2 = 4 → 1+4=5

P_4 — 5 → 5+5=10

P_1 = 7 → 10+7=17

P_3 = 17+9=26

P_1	P_2	P_4	P_1	P_3
0–1	1–5	5–10	10–17	17–26

P: ((10-0-1)+(1-1))

برای awt ← عدد کل P_1 ← 10 یکبار

در کل عدد قبلی هر کدام را بیاریم مثلاً P_4 ← 5 و آن را از arrival time کم کنیم

SJF (P):

پردازش	cpu burst	arrival time
P_1	8 - ۱ = ۷	۰
P_2	۴ - ۱ = ۳	۱
P_3	9	۲
P_4	5 - ۱ = ۴	3

P_2

P_1	P_2			P_4	P_1	P_3
0 – 1	1 – 2	2 – 3	3 – 4	5 – 10	10 – 17	17 – 26

P_1 ، P_2 زرگ‌تر می‌شوند.

$$\text{Average wait time} = \frac{\overset{P_1}{(0-0)+(10-1)} + \overset{P_2}{(1-1)} + \overset{P_3}{(17-2)} + \overset{P_4}{(5-3)}}{4} = \frac{26}{4}$$

✓ اگر پردازش‌ها باهم [illegible] برسند، [illegible] بنابراین همه حالت‌ها preemptive بهتر است.

↓

یک حالت آرمانی است.

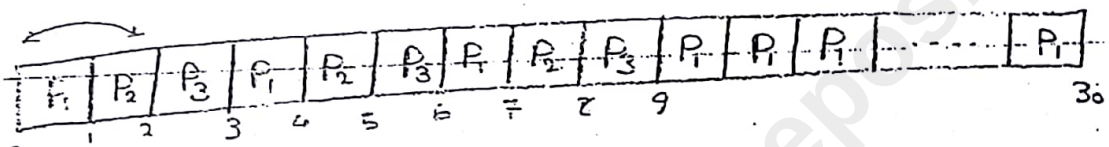
* Starvation ← پردازشی که بیش از حد در صف منتظر است تا به آن رسیدگی شود چون اولویت پایینی دارد.

↓

aging (بالا بردن سن)

« هر یک 15 دقیقه ⟸ 1 واحد به اولویت اضافه شود ← در مثال ما عددش کمتر می‌شود.

اگر Time Slice=1 :



$$a.w.T = \frac{(1+2+2) + (2+2+2) + (6+2+2+2)}{3} = \frac{17}{3} = 5.6$$

(P_2 P_3 P_1)

در اینجا awT ! برابر شد با زمانی که TimeSlice=4 Context switch اتفاق می‌افتد.

↳ P_2 کردن load و P_1 کردن save ← «در قسمت اول»

«پردازش قبلی save و پردازش بعدی load شود»

حل کلی