سیستم عامل

نام نویسی:

جلسه ترم 6 | 02/12/24 | فصل اول

تمرین 3 | فعالیت ۱ | کوئیز ۱

پایان ترم 9 | فصل ۳ تا ۹

سیستم عامل: واسط نرم افزاری که ارتباط بین برنامه های کاربردی و سخت افزار را ایجاد می کند.

- سیستم سخت افزار را مدیریت می کند و آن را به برنامه های کاربردی تخصیص می دهد.
- کاربر با برنامه های کاربردی کار می کند، برنامه های کاربردی با سیستم عامل و سیستم عامل با سخت افزار کار می کند.
- برنامه های کاربردی برنامه هایی هستند که معمولاً روی سیستم عامل نصب می شوند مانند بازی های مختلف
- برنامه های سیستمی، برنامه هایی هستند که جزئی از سیستم عامل هستند مانند نوت پد

وظیفه سیستم عامل:

- از دید کاربر: انتظاری که کاربر از سیستم عامل دارد (کاربر ارباب بنده رعیت)

مانند: - اجرای برنامه ها

- سرعت اجرای برنامه بدون در نظر گرفتن بهره وری منبع

- مدیریت بهینه خطاها

• از دید سیستم: تخصیص منبع (OS ارباب، بقیه افراد رعیت)

چیزی که باعث میشه کامپیوتر کار کنه سیستم عامل رو اجرا کنه، دستورات ROM است.

• دستورات ROM، سیستم عامل را از هارد شناسایی می‌کند و هسته اصلی آن را به RAM منتقل می‌کند و بعد از آن OS می‌تواند اجرا شود.

ROM → OS → اجرا → برنامه‌های کاربری → RAM
↓
RAM

ویژگی‌های حافظه:

- سرعت ← برای حافظه‌های نزدیک به CPU مناسب است.
- حجم ← برای ذخیره سازی انبوه
- هزینه هر بیت
- پایداری

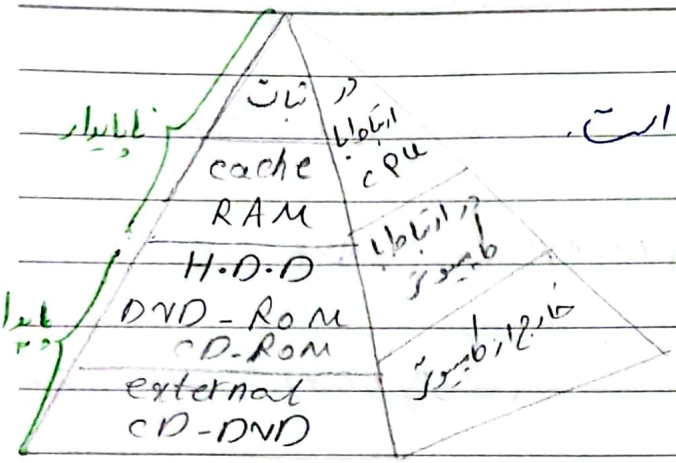
• علت تنوع حافظه چیست؟ چون حافظه‌ای نداریم که تمام ویژگی‌های خوب را کنار هم داشته باشد از انواع حافظه استفاده می‌کنیم

هرچقدر سرعت ↑، هزینه ↑

هرچقدر سرعت ↑، حجم ↓

هرچقدر حجم ↑، هزینه ↑

فرم حافظه



• ثبات سریع ترین، کم حجم ترین، گران ترین حافظه است.

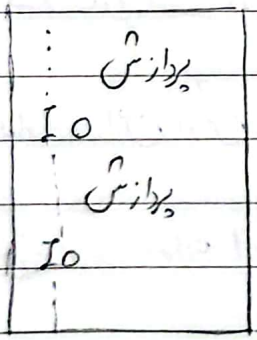
• cache و سه لایه است که هرکدام حجم و اندازه های مختلفی دارند. هرچه به CPU نزدیک تر باشد حجم آن کاهش پیدا می کند.

• هرچه به سمت پایین هرم حرکت می کنیم حجم حافظه ها افزایش پیدا می کند.

ساختار I.O

۲ روش برای انجام عملیات ورودی خروجی داریم:

۱- برنامه سازی شده



در این روش وقتی به دستور ورودی خروجی می رسیم CPU هیچ کار دیگری انجام نمی دهد و منتظر می ماند تا عملیات ورودی خروجی انجام شود

• این روش خوب نیست زیرا CPU هیچ کار دیگری انجام نمی دهد و همه برنامه ها منتظر ورودی خروجی می مانند

۲- مبتنی بر وقفه

در این روش وقتی CPU پردازش ها را انجام می دهد و به I.O می رسد منتظر نمی ماند و به سراغ پردازه دیگری می رود وقتی ورودی/خروجی تامین شد و اجرای دستورات پردازشی به CPU وقفه داده می شود و CPU به دستور I.O رسیدگی می کند

۲- دسترسی مستقیم به حافظه (DMA)

در این روش وقتی CPU به IO می رسد به DMA دستور می دهد و واحد DMA عملیات IO را انجام دهد اطلاعاتی که به DMA داده می شود عبارتند از:

۱- نوع عملیات (ورودی/خروجی)

۲- آدرس ذخیره سازی

۳- حجم داده ها ۴- آدرس ترمینال ها

CPU به ازای دستورات پردازشی سایر پردازنده ها می رود بعد از اینکه عملیات ورودی خروجی انجام شد و اطلاعات مورد نیاز CPU را در آدرس ذخیره کرد به CPU وقفه می خورد که به ادامه دستورات پردازشی برمیگردد

• تفاوت این روش با روش مبتنی بر وقفه این است که در روش مبتنی بر وقفه خود CPU وقتی ورودی برایه انجام میشد به اطلاعات را در آدرس مورد نظر copy می کرد اما در این روش عملیات copy هم به عهده DMA است

۱- اگر حجم IO کم باشد مبتنی بر وقفه بهتر است

۲- اگر حجم IO زیاد باشد DMA بهتر است.

وقفه Interrupt

به هر دلیلی که سیستم عامل بخواهد اختیار cpu را از دست بدهد و در اختیار برنامه دیگری قرار دهد Interrupt انجام می‌شود. به عبارت دیگر وقفه امکانی است که cpu را از سیستم عامل بگیرد و به جاهای مختلف منتقل کند.

معماری سیستم های کامپیوتری

۱- تک پردازنده ای: در سیستم کامپیوتری فقط یک cpu وجود دارد و آن وظیفه اجرای همه پردازش های کامپیوتر را به عهده داشت

• cpu خاص منظوره: فقط یک کار خاص انجام می‌دهد. برای اجرای پروژه های کاربردی تولید نشده

۲- چند پردازنده ای: سیستمی که چند پردازنده، چند منظوره داشته باشد که همه آنها پروژه های مربوط به کاربر را اجرا کنند.

مزایای سیستم چند پردازنده ای:

۱- بازده بالا:

سیستم با N cpu
سیستم با ۱ cpu
} بازده تقریبا کمتر از N برابر به زیرا زمانی صرف انجام کارهای جانبی می‌شود.

۲- صرفه اقتصادی :

به دلیل استفاده از منابع مشترک تهیه یک سیستم چهار پردازنده‌ای به صرفه‌تر از چهار سیستم تک پردازنده‌ای است.

۳- قابلیت اطمینان بالا :

در دسترس بودن سیستم را مشخص می‌کند.

با ایجاد مشکل برای یکی از cpu ها، تمام سیستم از کار نمی‌افتد و فقط باعث افت کارایی می‌شود.

سیستم چند پردازشی متقارن :

همه cpu ها یکسان هستند و وظیفه تقسیم کار به عهده کرنل است و برابری و برابری

سیستم چند پردازشی نامتقارن :

یک cpu وظیفه تقسیم کار به cpu های دیگر را دارد و یک cpu master و بقیه slave اند.

• در چند پردازشی نامتقارن اگر برای master مشکلی پیش آید، کل سیستم از کار می‌افتد و اگر cpu های دیگر از کار بیوفتند بازدهی کم می‌شود.

• در چند پردازشی متقارن برای هر کدام از cpu ها مشکلی پیش آید فقط بازدهی کم می‌شود.

• زمانی که بار پرداره روی سیستم بالا باشد معمولا از چند پردازشی نامتقارن می‌شود.

سیستم های خوشه ای (clustered system)

چندین سیستم از طریق شبکه محلی یا یک اتصال پر سرعت به یکدیگر متصل می شوند. و یک سیستم پر قدرت ایجاد می شود ← قابلیت اطمینان بالا

۱- خوشه ای نامتقارن: یک یا چند سیستم وجود دارد که به آن ها سیستم های Hot Standby Mode می گویند (آماده باش پر حرارت). این سیستم ها کاری انجام نمی دهند و تنها وظیفه دارند که در صورت بروز مشکل برای یکی از سیستم ها جایگزین آن شوند

• در پردازش های سنگین و مهم چندین سیستم را در حالت Hot Standby mode قرار می دهند تا مطمئن شوند مشکلی برای سیستم پیش نمی آید

۲- خوشه ای متقارن: همه سیستم ها در حال کار هستند و اگر مشکلی برای یکی از سیستم ها پیش بیاید وظایف آن بین بقیه سیستم ها تقسیم می شود ← نیاز به یک سیستم بالادری دارد که وظایف را تقسیم کند

سیستم های تک برنامه ای: سیستم عامل در هر لحظه بیش از یک برنامه را نمی تواند اجرا کند ← مانند ms-dos

چند برنامگی: امکان اجرای چند برنامه همزمان به صورت موازی اجرا شوند حتی اگر فقط یک cpu داشته باشیم.

• در چند برنامگی، چند برنامه در حافظه ram قرار دارد

RAM

OS
Job 1
Job 2
⋮
Job n

انبار کار

Job x

• برنامه‌ها برای اجرا مستقیماً از Hard به RAM منتقل نمی‌شوند

بلکه ابتدا در انباری به نام انبار کار قرار می‌گیرند

و بعد توسط OS برای اجرا انتخاب می‌شوند

• انبار کار همان حافظه مجازی است

درجه چند برنامگی: تعداد برنامه‌هایی که در حافظه RAM قرار می‌گیرند و به صورت موازی اجرا می‌شوند.

• به جابجایی برنامه‌ها بین انبار کار و RAM، Swapping گفته می‌شود.

• چند برنامگی توسط مفهومی به نام time sharing (اشتراک زمانی) مدیریت می‌شود.

برنامه‌های سیستمی: خود برنامه‌هایی که OS هستند و برای انجام بهتر وظایف باید دسترسی کامل به سخت افزار داشته باشند.

برنامه‌های کاربری: توسط برنامه نویس‌ها و برای منظور خاصی نوشته و فقط به همان منظور باید اجرا شود.

در سیستم عامل‌های قدیم دو مشکل اساسی وجود داشت:

۱- دسترسی نامحدود: برنامه‌ها دسترسی نامحدود داشتند.

۲- زمان اجرا: برنامه‌ها محدودیتی در زمان اجرا نداشتند.

• عملیات ضد دوگانه سعی در برطرف کردن این دو مشکل دارد

• به سخت افزارهای کامپیوتری یک بیت تحت عنوان بیت مد اضافه شده که مقدار صفر یا ۱ می گیرد.

اگر مقدار صفر داشته باشد: برنامه در حال اجرا برنامه سیستم است به آن مد هسته گفته می شود.

اگر مقدار ۱ داشته باشد: برنامه در حال اجرا برنامه کاربر است به آن مد کاربر گفته می شود.

دستورات ممتاز: دستورالعمل هایی که احتمال خرابی سیستم در آنها وجود دارد.

دستورالعمل های مخرب: دستورالعمل هایی که باعث خرابی سیستم می شوند.

دستورات مجاز: دستورالعمل هایی که ممتاز نباشند یعنی خود کاربر مجاز به اجرای آنها باشد

• برنامه های کاربری مجاز به اجرای دستورات ممتاز نیستند.

برنامه کاربری وقتی به دستورات ممتاز می رسد فراخوانی سیستمی انجام می دهد (System call)

• فراخوانی سیستمی، interface هست بین برنامه کاربری و کد که می تواند هر چیزی را از کد درخواست کند و در اختیار کاربر قرار دهد

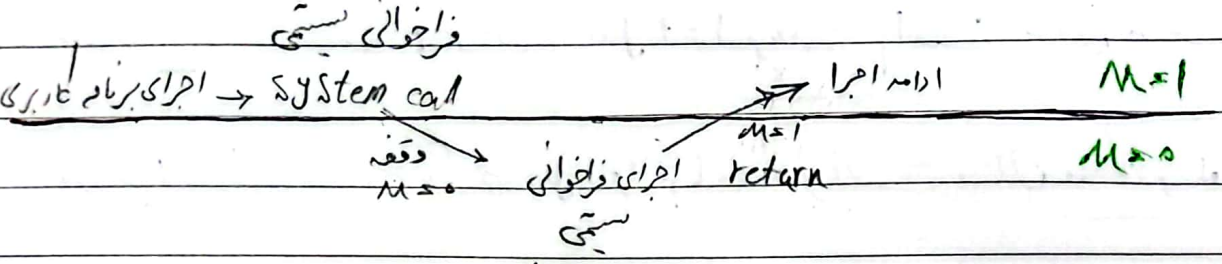
پس از فراخوانی سیستمی وقفه ای صادر می شود و cpu از برنامه کاربر گرفته می شود و به کد داده می شود

• اولین کاری که کد بعد از گرفتن cpu انجام می دهد تغییر بیت Mode به صفر است.

• اگر اول بیت مد تغییر کند و [illegible] دیگر cpu دست برنامه کاربری است

و برنامه کاربری هر کاری می تواند انجام دهد.

پس از اجرای فراخوانی سیستمی، نیتی به کاربر برگردانده می شود اما قبل از برگرداندن cpu به کاربر M=1 قرار می دهد.



• مشکل دسترسی با استفاده از بیت ممتاز حل شد اما مشکل زمان همچنان حل نشده بود

• برای حل مشکل اجرای بینهایت از تایمر استفاده می شود ← دسترسی به این تایمر جزو دستورات ممتاز به حساب می آید و برنامه کاربری نمی تواند به آن مقدار بدهند.

• سخت افزاری تحت عنوان تایمر به معماری کامپیوتر افزوده شد که برنامه ها حداکثر یک زمان معین cpu را در اختیار داشته باشند و پیش از آن زمان cpu به آنها داده نشود.

• تایمر یک مقدار اولیه دارد، وقتی به صفر رسید وقفه صادر می شود و این وقفه باعث می شود cpu از برنامه کاربری گرفته شود حتی اگر برنامه کاربری هنوز تمام نشده باشد cpu گرفته می شود و به برنامه بعدی داده می شود ← با استفاده از تایمر جلوی در اختیار گرفتن cpu توسط یک برنامه گرفته می شود

• اگر یک برنامه دستورات پردازشی زیادی داشته باشد ممکن است در loop های مختلف cpu کندی به آن برسد.

تایمر ← کاهش ← وقفه

• برنامه کاربری حق تغییر بیت مد و تایمر را ندارد.

برای این دستورات فراخوانی سیستمی هم وجود ندارد.

فصل دوم — ساختارهای سیستم عامل

سرویس‌های OS

۱- رابط کاربر (user interface)

• رابط کاربرهای مختلفی وجود دارد:

۱- رابط خط فرمان CLI ۲- Graf user interface GUI

۲- اجرای برنامه ← مهم‌ترین وظیفه OS

۳- عملیات IO

۴- سیستم فایل (FS) ← امکان مدیریت فایل

۵- ارتباطات ← که باید امکان برقراری ارتباط بین چند پردازه را فراهم کند.

• ارتباطات ← دو صورت ۱- ارسال پیغام ۲- حافظه مشترک می‌تواند ایجاد شود.

۶- تشخیص خطا

• خطا می‌تواند نرم‌افزاری یا سخت‌افزاری باشد.

نرم‌افزاری ← دسترسی غیرمجاز به حافظه، تقسیم بر صفر، سرریز در محاسبات

۷- تخصیص منبع ← باید request ها و release ها را مدیریت کند.

۸- حسابداری: تعداد، میزان و زمان استفاده پردازه ها از منابع

۹- حفاظت و امنیت

مفسر فرمان:

۱- بخش اصلی در هسته قرار گیرد

۲- برنامه مجزا در برنامه های سیستمی

- بعضی مفسرها هستند که چند نمونه مفسرهای فرمان دارند به اینها پوسته (Shell) می گویند که هرکدام به منظور خاصی مورد استفاده قرار می گیرند

مفسر فرمان CLI به دو شکل می تواند پیاده سازی شود:

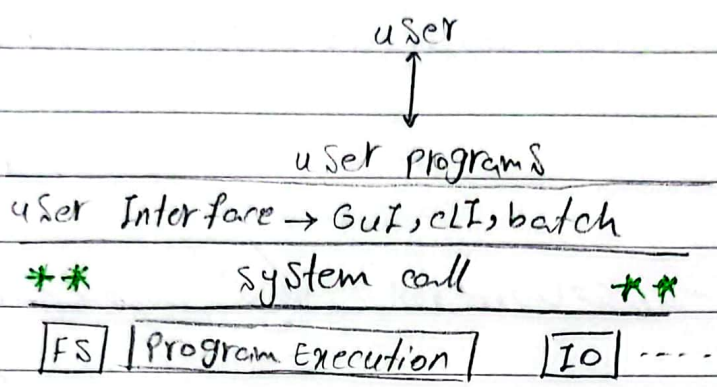
۱- دستورات را به صورت Hard code در خود داشته باشد

- دستورات Hard code، دستوراتی هستند که وقتی برنامه ساخته شد دیگر قابل تغییر نباشد

۲- دستورات به صورت فایل های مجزا در کنار مفسر قرار می گیرد

- اگر مفسر به صورت فایل باشد انعطاف پذیری بالایی دارد و دستورات به راحتی کم و زیاد می شود و قابل توسعه است اما اگر به صورت Hard code باشد برای اضافه کردن دستور باید نسخه قبلی مفسر پاک شود و دوباره جدید ترکیب شود اما سرعت بیشتری دارد

واسط کاربر بین برنامه‌های کاربری و سرویس‌های سیستم عامل، فراخوانی‌های سیستمی هستند.



API: Application programming Interface

رابط برنامه‌نویسی کاربردی

API: مجموعه‌ای از توابع + پارامترهای ورودی آن‌ها + نوع خروجی آن‌ها

انواع فراخوانی سیستمی:

۱- کنترل برنامه

۲- مدیریت فایل

۳- مدیریت دستگاه

۴- نگهداری اطلاعات

۵- ارتباطات

ساختار سیستم عامل :

۱. ساختار ساده : مربوط به کد هایی که برای برنامه و هدف خاصی نوشته شده و قرار بوده که بشن مانند MS-DOS

• سیستم ها با این ساختار به صورت کوچک، ساده و محدود ساخته می شوند.

• سطوح مختلف تفکیک نشده هستند.

بررسی ساختار MS-DOS :

برنامه های کاربردی (Application programs)

این برنامه ها از برنامه های خود سیستم عامل استفاده می کنند

برنامه های سیستمی (resident system program)

برنامه های کاربردی و سیستمی با استفاده از درایورها از سخت افزار استفاده می کنند

درایورها (MS-DOS device Drivers)

سخت افزار (Hard core)

* همه سطوح به طور مستقیم با هم در ارتباط بودند

بررسی ساختار یونیکس اولیه:

تفکیک لایه‌ها به نسبت بهتر بود اما باز هم مدیریتی بین لایه‌ها وجود نداشت و برنامه‌های کاربردی به تمام قسمت‌ها دسترسی داشتند.

کاربران — کاربران با برنامه‌های مختلف به سخت افزار

پوسته‌ها و مفسرهای فرمان — دسترسی داشتند و تمامی سرویس‌های سیستم عامل را

کامپایلرها و مفسران — یک لایه کنار هم پیاده سازی شدند.

فراخوانی سیستمی (SCI)

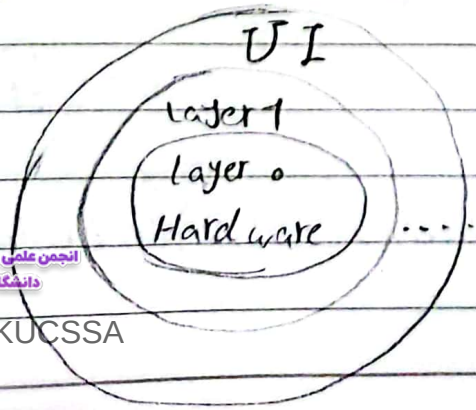
FS، زمانبندی CPU، ... } سرویس‌های OS

مدیریت حافظه، I/O، ...

سخت افزار

۲- ساختار لایه به لایه:

سیستم عامل به صورت لایه‌های مجزا پیاده سازی شود. درونی‌ترین لایه را لایه سخت افزار در نظر می‌گیرند. هر لایه با interface به لایه پایینی در اختیارش قرار داده فقط می‌تواند از لایه پایینی استفاده کند.



مزیت ها:

۱- عیب یابی و رفع خطای ساده

۲- امنیت بالا - به کد هر لایه برای لایه های دیگر مخفی است

ایرادات:

۱- تفکیک لایه ها از هم کار ساده ای نیست.

۲- سرعت پایین

۳- ریز هسته:

مولفه های غیر ضروری از هسته خارج و به عنوان برنامه های سیستمی مورد استفاده قرار می گیرند.

ارتباط بین برنامه های سیستمی و هسته از طریق ارسال پیغام صورت می گیرد.

مزیت ها:

۱- پیاده سازی ساده به دلیل حجم کم هسته

۲- سادگی در توسعه

۳- امنیت به دلیل مراجعه کم به هسته سیستم عامل

ایراد:

۱- چون ارسال پیغام کند است و ارتباط از طریق ارسال پیغام انجام می شود باعث کندی سیستم می شود

۲- چون همه قسمت های مهم در ریزهسته قرار گرفتند اگر ریزهسته از کار بیوفتد کل سیستم از کار میوفتد.

۴- ماژول ها:

مانند ساختار ریزهسته قسمت های اصلی را در ریزهسته در نظر گرفته اما قسمت های دیگر را مانند عنوان ماژول هایی در نظر گرفته که در صورت نیاز در حافظه لود شوند و از آنها استفاده شود.

این باعث می شود سرعت سیستم عامل افزایش پیدا کند.

• ارتباط بین ماژول ها از طریق حافظه مشترک ایجاد می شود.

۵- ترکیبی:

ترکیبی از ساختار لایه به لایه و ریزهسته

یک لایه ساختار ریزهسته و لایه های بالایی شامل برنامه های کاربردی

فصل سوم پردازه ها

واحد انجام کار cpu، پردازه (process) است.

پردازه: یک برنامه در حال اجرا است.

تفاوت بین برنامه های کاربردی و process:

برنامه های کاربردی برنامه هایی هستند که روی سیستم نصب می شوند و موجودیت منفعل دارند و

به صورت پیش فرض در حال اجرا نیستند اما وقتی به هر طریق اجرا می شوند تبدیل به یک موجودیت

فصل سوم — پردازه ها

واحد انجام کار cpu، پردازه (process) است.

پردازه: یک برنامه در حال اجرا است.

تفاوت بین برنامه‌های کاربردی و process:

برنامه‌های کاربردی برنامه‌هایی هستند که روی سیستم نصب می‌شوند و موجودیت منفعل دارند و

به صورت پیش فرض در حال اجرا نیستند اما وقتی به هر طریق اجرا می‌شوند تبدیل به یک موجودیت

فعال می‌شوند.

برنامه کاربردی ← موجودیت منفعل

پردازه ← موجودیت فعال

• پردازه از یک شمارنده برنامه و قسمت های زیر تشکیل شده

stack	← ذخیره متغیرهای محلی و پارامترهای تابع
	← پشته از بالا به پایین و heap از پایین به بالا فضای حافظه را پر می‌کنند
heap	← تخصیص حافظه پویا
data	← ذخیره عملوندهای مورد نیاز برنامه
text	← شمارنده برنامه، شماره دستوری که در این قسمت نوشته شده را نگه می‌دارد.

حالات پردازه‌ها: وضعیت‌ها و ویژگی‌هایی که پردازه‌ها در حالت اجرا با آن مواجه می‌شوند

۱- حالت شروع: در این حالت پردازه ایجاد می‌شود. (new)

۲- حالت آماده (ready): پردازه یا پردازه‌هایی که در حالت آماده هستند همه منابع لازم را در اختیار دارند و فقط منتظر cpu هستند.

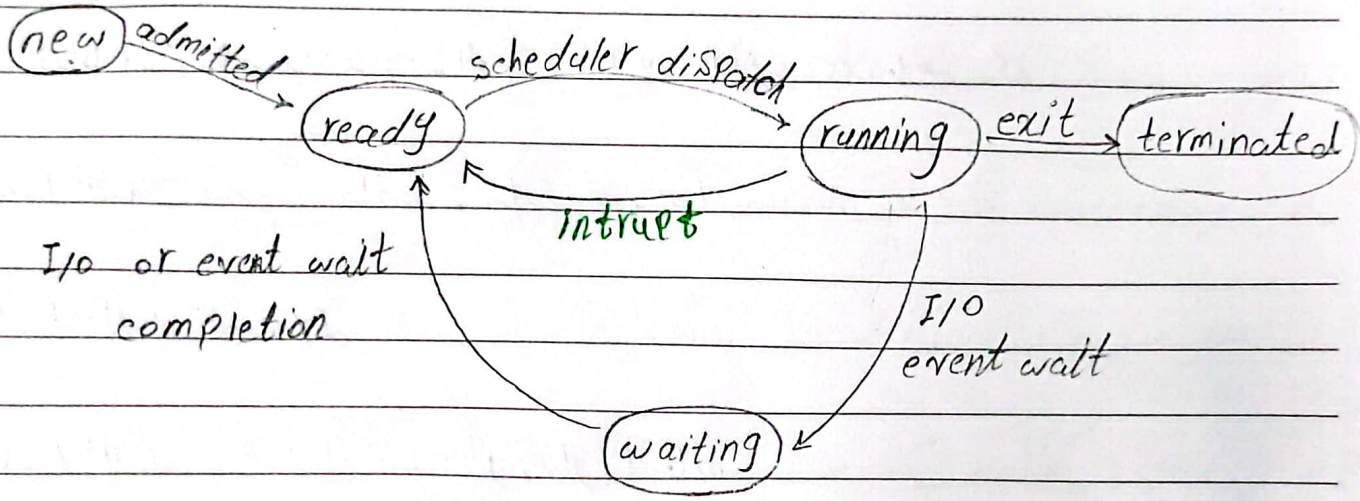
۳- حالت اجرا: پردازه‌ای که دستورات آن توسط cpu در حال اجرا است. (running)

۴- حالت انتظار (waiting): پردازه‌ای که در حال اجرا است، وقتی به دستوری می‌رسد که اجرای آن نمی‌تواند ادامه پیدا کند از حالت اجرا خارج می‌شود و به حالت انتظار می‌رود.

• با اتمام حالت انتظار به حالت آماده برمی‌گردد.

۵- حالت خاتمه (terminated): دستورات برنامه بعد از اجرای مختلف به پایان رسید، در حالت خاتمه قرار می‌گیرد.

نمودار تغییر حالت



intruet: پردازه بدون دلیل و میدار انتظار از حالت اجرا خارج شده است.

• هر پردازه ای وقتی از حالت اجرا خارج می‌شود باید نتایج آن ذخیره شود و قبل از اجرا باید داده‌های قبلی خود را بارگذاری کند.

بلاک کنترل پردازه

process control Block → (PCB)

PCB: فضایی که به هر پردازه اختصاص داده می‌شود برای ذخیره داده‌های مربوط به آن پردازه

داده‌هایی که در PCB ذخیره می‌شود:

۱- حالت پردازه

۲- شمارنده برنامه (PC)

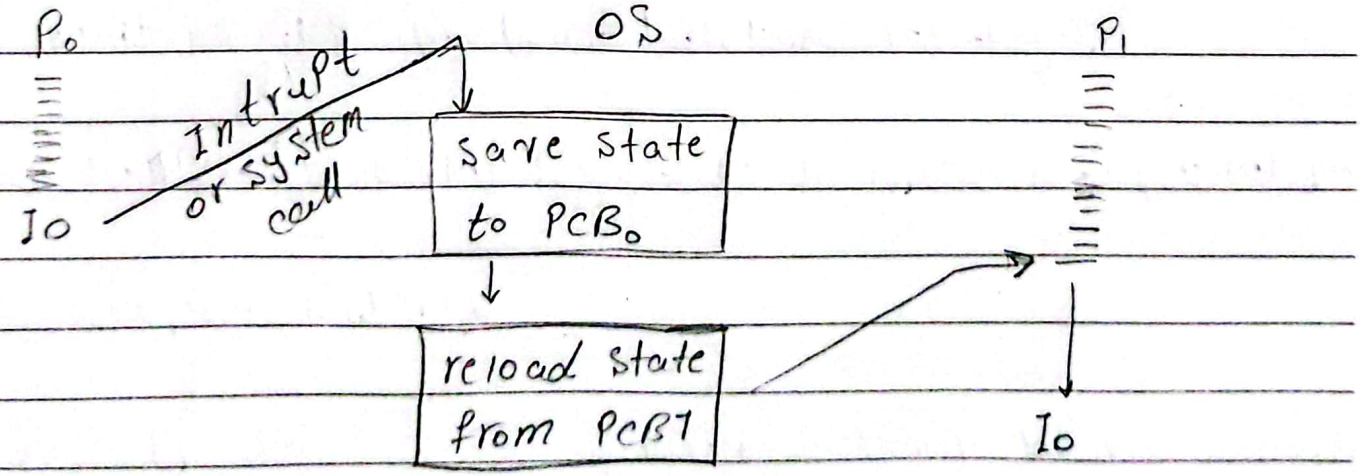
۳- ثبات‌های CPU

۴- اطلاعات زمانبندی CPU: اولویت پردازه - پارامترهای زمانبندی

۵- اطلاعات مدیریت حافظه: ثبات حد و مبنا

۶- اطلاعات I/O

۷- اطلاعات حسابداری: میزان زمان استفاده



صف آماده ← لیست پیوندی از PCB ها (در حافظه اصلی قرار دارد)

• هرجا که چند پردازه در حالت آماده باشند صف آماده داریم.

صف کار (Job Queue): صفی برای انتظار برای ورود به حافظه اصلی

← در حافظه مجازی قرار دارد.

زمانبند ← وظیفه مدیریت صف ها را دارد.

• از بین پردازه هایی که در صف کار آماده ورود به حافظه اصلی هستند چندین پردازه را انتخاب و به حافظه اصلی منتقل می کند و پردازه ها در صف آماده قرار می گیرند. ← تخصیص حافظه به پردازه ها (زمانبند کار)

• از بین پردازه هایی که در صف آماده هستند، یک پردازه به حالت اجرا می رود. (زمانبند کوتاه مدت)

درجه چندبرنامگی: تعداد process هایی که برای حافظه اصلی انتخاب می شوند (به صف آماده منتقل می شوند) درجه چندبرنامگی گفته می شود.

• وظیفه زمانبند بلندمدت مشخص کردن درجه چندبرنامگی است.

• به زمانبند cpu، زمانبند کوتاه مدت گفته می شود زیرا در بازه زمانی خیلی کم نیاز است اجرا شود و پردازه های بعدی را مشخص کند ← به صورت میانگین هر ۱۰۰ میلی ثانیه یکبار زمانبندی انجام می دهد.

• به زمانبند کار، زمانبند بلندمدت گفته می شود زیرا فرکانس اجرایی کمی دارد و معمولاً چند دقیقه یکبار اجرا می شود.

زمانبند میان مدت: وظیفه آن مبادله (swapping) است.

• به انتقال برنامه از حافظه اصلی به حافظه مجازی ← swap out

• به انتقال برنامه جدید به حافظه اصلی ← swap in

• وظیفه دیگر زمانبند میان مدت مشخص کردن ترکیب پردازه‌ها است.

پردازه‌ها:

I/O Bound1: پردازه‌هایی که تعداد دستورات ورودی خروجی زیادی دارند.

cpu Bound2: پردازه‌هایی که دستورات پردازشی زیادی دارند.

• هر پردازه دنباله‌ای از دستورات پردازشی و دستورات ورودی/خروجی است.

• اگر نسبت تعداد I/O Bound و cpu Bound به هم بخورد، زمانبند میان مدت Swapping این نسبت را دوباره برقرار می‌کند.

* عملیات روی پردازه‌ها

1- ایجاد پردازه ← یک پردازه وقتی می‌خواهد ساخته شود یا با دستور new یک پردازه جدید ایجاد می‌شود

یا فراخوانی سیستمی با نام fork یک از شعاب صورت می‌گیرد.

• همیشه یک پردازه توسط یک پردازه دیگر ساخته می‌شود

به پردازه‌ای که پردازه ایجاد می‌کند، پردازه والد و به پردازه‌هایی که توسط پردازه والد ایجاد شده‌اند

• برنامه‌های کاربری هم می‌توانند process ایجاد کنند.

• برای پردازه‌ها می‌توان ساختار درختی رسم کرد و برای هر پردازه یک pid در نظر گرفت

ایجاد فرزند جدید:

۱- پردازه فرزند کپی والد باشد.

۲- پردازه فرزند پردازه‌ای متفاوت باشد.

فرزند در مقابل والد به ۲ روش می‌تواند اجرا شود:

۱- پردازه والد منتظر خاتمه فرزند شود. ← معمولاً وقتی والد به نتایج پردازه فرزند نیاز دارد.

۲- پردازه والد همروند با فرزند به اجرای خود ادامه دهد.

۲- خاتمه پردازه

دلایل خاتمه پردازه:

۱- کار محوله به پردازه فرزند انجام شود.

۲- فرزند منابعی بیش از حد مجاز استفاده کرده باشد.

• در این حالت پردازه فرزند قبل از خاتمه دستورات بسته می‌شود.

۳- خاتمه والد

• در این موارد خاتمه آبشاری داریم یعنی هر والد که حذف شود به صورت سلسله مراتبی فرزندانش

ارتباط پردازه ها

پردازه مستقل : پردازه‌ای که نه از پردازه های دیگر اثر بگیرد و نه روی پردازه های دیگر اثر بگذارد.

پردازه های همکار : پردازه هایی که روی هم اثر می گذارند.

• برای اجرای پردازه های همکار نیاز به ارتباط بین پردازه هاست.

دلایل ارتباط بین پردازه ها :

۱_ اشتراک اطلاعات

۲_ تسریع محاسبات

۳_ پیمانه ای ساختن (Modularity)

۴_ راحتی کاربر

ارتباط بین پردازه ها به ۲ روش امکان پذیر است :

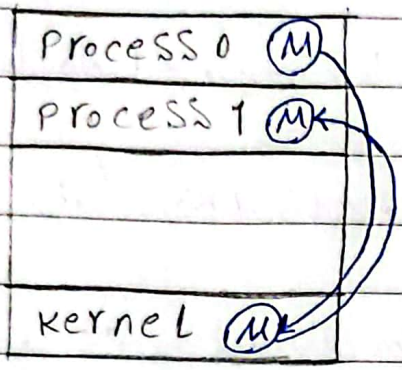
۱_ ارسال پیغام ← حجم داده های مشترک کم است.

حجم کم داده ها به عنوان یک پیغام از process مبدا به process مقصد ارسال می شود و معمولاً از هسته سیستم عامل استفاده می شود.

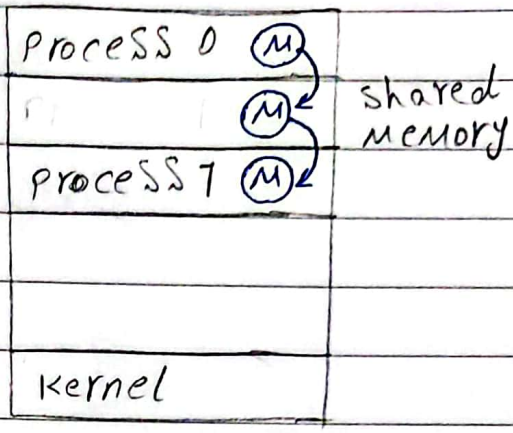
• چون هسته سیستم عامل درگیر است این ارسال کند است.

۲- حافظه مشترک ← حجم داده‌ی مشترک زیاد است.

بین پردازه‌ها یک حافظه مشترک ایجاد می‌شود که وظیفه هسته سیستم عامل است.



ارسال پیغام



مثال: مسأله تولید کننده - مصرف کننده

یک بری پردازه تولید کننده داریم که item ها را تولید می‌کند و در حافظه مشترک قرار می‌دهد

این حافظه مشترک تحت عنوان بافر در نظر گرفته می‌شود (حافظه مشترک ← بافر)

پس پردازه‌های مصرف کننده item ها را از حافظه مشترک برمی‌دارند و استفاده می‌کنند

مانند: کامپایلر و اسمبلر ← کامپایلر کد را تولید می‌کند و در بافر قرار می‌دهد و اسمبلر مصرف کننده است.

• مصرف کننده‌ها item را از بافر حذف می‌کنند و باعث تغییر در بافر می‌شوند.

پردازه‌های همکار در این مثال:

- تولید کننده‌ها با هم
- مصرف کننده‌ها با هم
- تولید کننده‌ها مصرف کننده‌ها با هم

• زیرا روی یک بافر مشترک عملیات انجام می‌دهند و عملیات‌شان روی هم تاثیر می‌گذارد.

اندازه بافر:

محدود: باید محدودیتی در تولید item در بافر وجود داشته باشد و این محدودیت معمولاً توسط کد باید مدیریت شود

• اگر برنامه کاربردی باشد باید توسط برنامه نویس مدیریت شود.

نامحدود: محدودیتی در تولید item وجود ندارد.

شبه کد مدیریت حافظه مشترک

```
typedef struct {  } item;
item Buffer[n];
int in=0 , out=0;
```

n اندازه بافر است

in اندیس اولین جای خالی در بافر و out اندیس item معرفی توسط معرف کننده

اگر in=out باشد بافر خالی است

شبه کد تولید کننده

آیتم بعدی که تولید می‌شود در بافر قرار می‌گیرد

```
item nextItem;
while (true) {
    // produce an item
    nextItem = new Item( );
    while ((in+1)%n == out)
    ; // do nothing: wait
    Buffer[in] = nextItem;
    in = (in+1) % n;
}
```

تا زمانی که بافر پر باشد

in++ به صورت چرخشی انجام می‌شود

تک مصرف کننده

```
item nextConsumed;
while(True) {
    while (in==out)
        ;//do nothing :wait
    nextConsumed=Buffer[out];
    out=(out+1)%n;
    // consumed the item
}
```

آیتمی در بافر برای مصرف هست یا نه

• اگر تولید کننده ی وظایف کار کند و مصرف کننده ی بیکار باشد، بافر پر می شود و تولید کننده ی هم بعد از چندبار تولید متوقف می شود

• اگر مصرف کننده ی کار کند و تولید کننده ی بیکار باشد، بافر خالی می شود و مصرف کننده ی منتظر می ماند.

• پس که باید مدیریت کند تولید کننده ی به اندازه تولید و مصرف کننده ی به اندازه مصرف کند.

روش ارسال پیغام ← برای پردازه هایی که در یک سیستم نیستند

مبدأ: send(message) مقصد: receive(message)

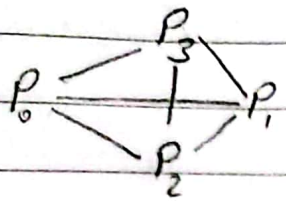
با توجه به خط ارتباطی برقرار شده روش های مختلفی برای ارسال پیغام وجود دارد:

۱- ارتباط مستقیم یا غیر مستقیم

ارتباط مستقیم:

۱- بین هر دو پردازه همکار یک خط ارتباطی ایجاد می شود

درگاه ارتباط مستقیم



۲- هر خط ارتباطی تنها متعلق به دو پردازه است.

۳- بین هر دو پردازه دقیقاً یک خط ارتباطی وجود دارد.

ارتباط غیر مستقیم : صندوق پستی یا درگاه ها (port)

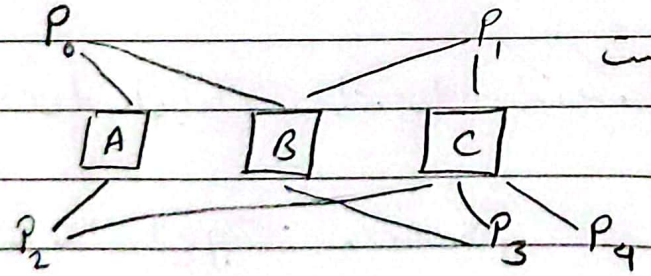
- مبدأ پیغام را در صندوق پستی قرار می دهد و مقصد پیغام را از صندوق پستی برمی دارد.

مبدأ: send(A, Message) مقصد: receive(A, Message)

۱- در صورتی یک خط ارتباطی بین دو پردازه وجود دارد که بین آنها حداقل یک port مشترک وجود داشته باشد.

۲- یک خط ارتباطی می تواند به بیش از دو پردازه متعلق باشد.

۳- بین دو پردازه می تواند خطوط ارتباطی مختلفی باشد.



- بین P_1 و P_2 هم از طریق A و هم از طریق C ارتباط هست

- بین P_0 و P_4 ارتباطی وجود ندارد

ارسال به دو صورت انجام می شود:

۱- مسدود شونده (هماهنگام) : فرستنده، خط ارتباطی را تا رسیدن پیغام به گیرنده مسدود می کند.

۲- بی انسداد (ناهماهنگام) : فرستنده، فقط send message انجام می دهد.

دریافت :

۱- مسدود شونده (همگام) : خط ارتباطی مسدود می شود تا زمانی که پیغام دریافت شود.

۲- بی انسداد (ناهمگام) : می تواند بارها فراخوانی سیستمی receive انجام دهد و چک کند پیغامی هست یا نه.

ملاقات : زمانی که ارسال و دریافت هردو مسدود شونده باشند.

• در ارتباط غیر مستقیم معمولاً ارسال و دریافت پیغام به صورت بی انسداد انجام می شود.

بافرسازی :

۱- ظرفیت صفر : فضایی برای ذخیره پیغام منتظر نداریم ← ارسال مسدود شونده

• معمولاً مربوط به ارتباطات مستقیم است.

۲- ظرفیت محدود : چند پیغام به صورت محدود به صورت منتظر در خط ارتباطی نگه می داریم.

← ارسال و دریافت بی انسداد ← اگر ظرفیت پر شد فرستنده باید مسدود شود.

• معمولاً زمانی استفاده می شود که فرستنده بخواهد مطمئن شود پیغام به دست گیرنده می رسد.

۳- ظرفیت نامحدود : اگر فضای ذخیره سازی نسبت به اندازه پیغام زیاد باشد می گوییم ظرفیت نامحدود داریم.

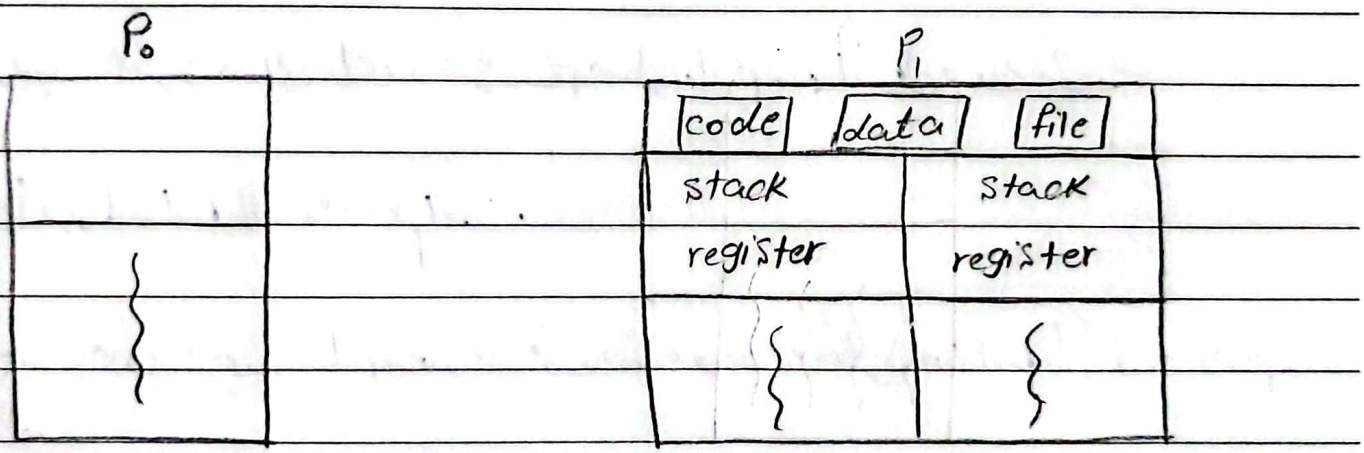
فصل ۴

Threads نخ ها

نخ: دنباله‌ای از دستورات که توسط یک پردازه انجام می‌شود را نخ می‌گویند.

• اگر یک پردازه از زمان شروع تا پایان برنامه یک رشته نخ محاسباتی داشته باشد به آن single Thread گویند.

• یک پردازه، چندین رشته نخ محاسباتی داشته باشد که این رشته‌ها روند اجرای محاسبات را مشخص می‌کنند و این روندها می‌توانند به صورت موازی اجرا شوند ← multi Thread



Single Thread multi Thread

• برای هر رشته نخ، نیاز به ثبات‌ها و پشته جدا وجود دارد. اما کد، داده و فایل بین رشته نخ‌ها یکسان است.

• در OS، multi Thread باید نخ‌ها را مدیریت کند.

انگیزه چندنخی :

۱- تقسیم یک پردازه به قسمت های مختلف

۲- انجام کارهای مشابه

مزیت ها :

۱- پاسخگو بودن ← اگر برای یک Thread مشکلی پیش بیاید بقیه Thread ها به کار خود ادامه می دهند

۲- اشتراک منبع ← code ، data ، file به صورت مشترک استفاده می شوند

۳- صرفه اقتصادی

۴- توسعه پذیری

• برنامه نویس ها در ساخت و اجرای Thread ها بسیار موثر هستند.

نکاتی که برنامه نویس ها باید توجه کنند :

۱- امکان تقسیم فعالیت وجود دارد یا خیر

۲- تعادل بین قسمت ها

۳- شکستن داده ها ← امکان تقسیم داده ها بین Thread ها وجود دارد یا نه

۴- وابستگی داده ها ← ناسازگاری داده ای به وجود نیاید

۵- آزمون و عیب یابی ← در multi Thread دشوار است.

با توجه به اینکه پشتیبانی نخ به چه صورت است، ۲ نوع نخ داریم:

۱- نخ های کاربر: پشتیبانی نخ ها در سطح کاربر و بدون پشتیبانی هسته انجام می شود

۲- نخ های هسته: پشتیبانی در سطح هسته انجام می شود درواقع نخ های هسته به صورت مستقیم توسط os مدیریت می شود

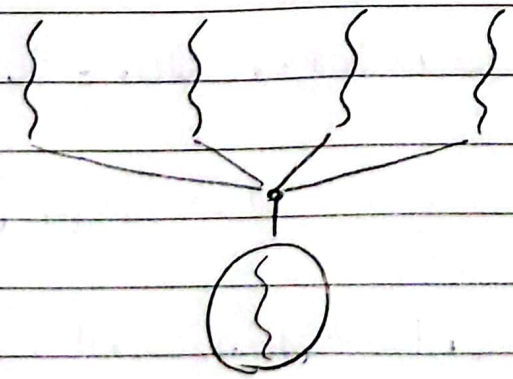
- اگر برنامه کاربری بدون فراخوانی سیستمی انجام شود، نخ های کاربری توانند برنامه را اجرا کنند.

اگر نیاز به فراخوانی سیستمی شود، چون فراخوانی سیستمی توسط os انجام می شود، نخ های کاربر باید فراخوانی را به نخ های هسته ارجاع دهند. ← نخ های کاربر و هسته باید با هم ارتباط داشته باشند.

مدل های چند نخی:

۱- مدل چند به یک: چندین نخ کاربر به یک نخ هسته نگاشت می شوند و توسط آن مدیریت می شوند

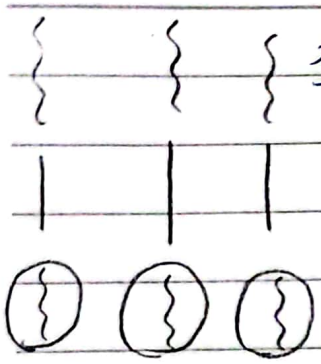
مشکلات:



- اگر مشکلی برای نخ هسته به وجود بیاید برای همه ی نخ های کاربر مشکل به وجود می آید.

- نخ هایی که با یک هسته مدیریت می شوند نمی توانند به صورت موازی در هسته های مختلف محاسباتی اجرا شوند.

۲- مدل یک به یک : هر نخ سطح کاربر به یک نخ سطح هسته نگاشت شود



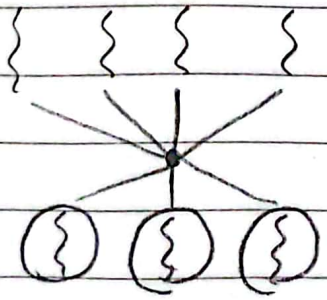
• مشکلات مدل قبلی را ندارد.

مشکل این مدل :

چون به ازای هر نخ کاربر یک نخ هسته وجود دارد، تعداد نخ های سطح هسته افزایش پیدا می کند و مدیریت آن توسط که سخت تر می شود و سیستم کند می شود

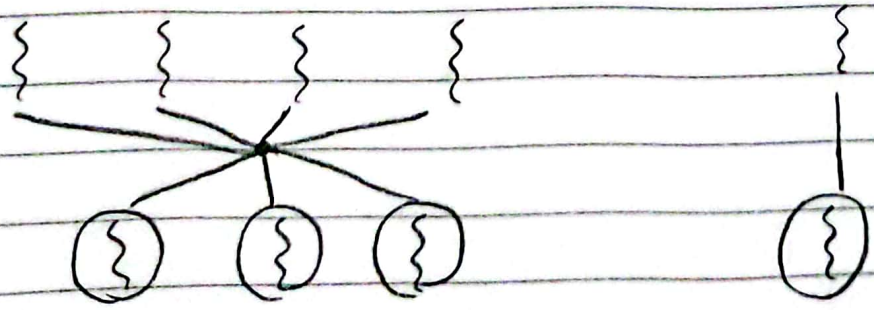
۳- مدل چند به چند : چندین نخ کاربر به تعداد کمتر یا مساوی نخ هسته نگاشت می شوند.

• مشکلات دو مدل قبل را ندارد



۴- مدل دو سطحی : ترکیبی از مدل چند به چند و یک به یک است.

• چندین نخ کاربر به کمتر مساوی نخ هسته نگاشت می شوند و برخی نخ های کاربر که نخ های مهمی هستند (اولویت بالا و اهمیت زیادی دارند) به یک نخ سطح هسته نگاشت می شوند.



Subject :   Date :

فصل پنجم: جلسه یازدهم:

یک process با دستورات پردازشی شروع می‌شود و با همان هم خاتمه می‌یابد.

P
{ دستورات پردازشی
IO
{ دستورات پردازشی → Cpu Burst   انفجار پردازنده
IO → IO Burst
{ دستورات پردازشی }

زمان‌بندی پیش گرفتنی: ۱. از حالت اجرا به انتظار ← زمان‌بند بعد از این حالت باید پردازنده دیگری را از صف آماده انتخاب کند در حالت اجرا ببرد.

۲. از حالت اجرا به آماده

۳. حالت انتظار به آماده

۴. خاتمه پردازنده ← زمان‌بند بعد از این حالت باید پردازنده دیگری را از صف آماده انتخاب کند و به حالت اجرا ببرد.

اگر فقط در حالت ۴ و ۱ زمان‌بندی انجام شود ← زمان‌بندی تعاونی است ← غیر قابل پیش گیر

۲،۳ ← زمان‌بندی پیش گیر است

وقتی که یک process به cpu داده می‌شود تا زمانی که در دور CPU Burst آن تمام نشود آن را رها نمی‌کند.
اما اگر سیستم‌عامل بتواند وسط CPU Burst یک process، cpu را از آن بگیرد پس آن زمان‌بندی پیش گرفتنی می‌گویند (اینجا به راهکاری برای دسترسی به داده‌های مشترک داشته باشیم)

در داده‌های مشترک زمان‌بندی‌های پیش گیرنده، ویژگی‌های مشترکی دارند

Subject : Date :

معیارهای زمان‌بندی: ۱. بهره‌وری پردازنده ← میزان استفاده از cpu (چند درصد زمان مشغول باشد ↓ است) ↑ ۴۰-۹۰٪

۲. بازده ← تعداد پردازه‌های تکمیل شده در واحد زمان ↑

۳. زمان گردش ← فاصله زمانی بین شروع تا تکمیل یک پردازه ← میانگین زمان گردش ↓

زمان گردش = در صف آماده + I/O انتظار + اجرا + حافظه

✓ — × — × — ×

۴. زمان انتظار ← جمع تمام زمان‌های پردازه در صف آماده * مهم‌ترین

↓ میانگین ← " " " منتظر cpu است.

۵. زمان پاسخ ← زمان از لحظه شروع پردازه تا رسیدن به اولین پاسخ ← میانگین ↓

سرعت دریافت پاسخ

جلسه دوازدهم: FCFS ① SJF ② SRT ③ PA ④ RR ⑤ MLQ ⑥

MLFQ ⑦

الگوریتم‌های زمان‌بندی (زمان‌بند cpu)

اگر چندین پردازه در صف آماده داشته باشیم، چطور الگوریتمی یکی از پردازه‌ها را انتخاب کند و به cpu تخصیص دهد.

۱. الگوریتم FCFS "first come first service"

تعاریفی است یعنی تا زمانی که ...

CPU Burst آن تمام نشود cpu را رها نمی‌کند.

Process	Cpu Burst
P_1	24
P_2	3
P_3	3

FIFO

P_1	P_2	P_3
0 – 24	24 – 27	27 – 30

$$\text{میانگین زمان انتظار} = \frac{0+24+27}{3} = 17$$

خوب یا بد بودن الگوریتم بستگی به این دارد که پردازه‌ها در صف چجوری قرار گرفتن، اگر اول پردازه‌های طولانی در صف باشند و بعد پردازه‌های کوتاه میانگین زمان انتظار بالا می‌رود ولی اگر برعکس باشد میانگین ...

در انتخاب کردن نیازی به CPU Burst ها ندارد فقط به کدام زودتر در کدام زودتر آمده بستگی دارد

اشکال: قحطی زدگی پیش می‌آید برای پردازه‌های با طول زیاد. با CpuBurst های کم متقاضی، پردازه‌های با طول بلند و طولانی باید منتظر بمانند.

۲. SJF Shortest job first

از بین تمام پردازه‌هایی که در صف آماده هستند آن را انتخاب کن که Cpu Burst کمتری دارد نسبت به بقیه. برای انتخاب پردازه‌ها نیاز به Cpu Burst دارد.

Process	Cpu Burst
P_1	6
P_2	8
P_3	7
P_4	3

یک پردازه CpuBurst بعدی آن چقدر خواهد بود → پیش‌بینی / برآورد

اگر چند پردازه با CpuBurst یکسان وجود داشت، اولویت انتخاب با پردازه‌ای است که زودتر آمده باشد. تساوی است.

P4	P1	P3	P2
0	3	9	16 → 24

ثابت شده که الگوریتم SJF بهترین الگوریتم است که کمترین میانگین زمان انتظار را حاصل می‌کند.

$$\text{میانگین زمان انتظار} = \frac{3+16+9+0}{4} =$$

P_i	P_i
Cpu Burst	CB_1
IO Burst	CB_2
Cpu Burst	CB_{n-1}
IO Burst	CB_n

هر چه α به یک نزدیک‌تر باشد تخمین حاصل به زمان واقعی قبلی نزدیک‌تر است و هر چه α به صفر نزدیک‌تر باشد برآورد به حدس دوره قبلی نزدیک‌تر است.

CB_n زمان برآورد: $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ (t: واقعی، τ: پیش‌بینی؛ t_{n-1}: زمان واقعی قبلی، τ_{n-1}: حدس دوره قبلی) $\xrightarrow{\alpha=1/2} \frac{\tau_{n-1}+t_{n-1}}{2}$, $0<\alpha<1$

$$\tau_n = \alpha t_{n-1} + (1-\alpha)\alpha t_{n-2} + (1-\alpha)^2\alpha t_{n-3} + (1-\alpha)^3\alpha t_{n-4} + \dots + (1-\alpha)^n \tau_0$$

Subject :   Date :

۳. SRT "Shortest remaining time"

پیشگیرانه

از بین پردازه‌هایی که در صف هستند آنکه زمان باقیمانده کمتری دارد آن زودتر اجرا شود

Process	CB	ET
P1	5	10
P2	7	0
P3	3	6
P4	3	0
P5	1	1

P5	P4	P2	→	P1	P3	P2

P4	P5	P4	P2	P3	P2	P2	P1

ET ← 0   1   2   4   6   9   10   14   19

زمان خاتمه - جمع زمان ورود - Cpu Burst = زمان انتظار

از SJB کمتر است (میانگین زمان انتظار)

$$\text{میانگین زمان انتظار} = \frac{4+7+0+1+0}{5}$$

جلسه سیزدهم :

SjF → شکل بالا

P4	P5	P2	P3	P1

0   3   4   11   14   19

اول ET بعد به اولویت

۴. PA "الگوریتم اولویت"

Process	CB	ET	Priority	زمان انتظار
~~P0~~	2 ~~3~~	0	2	5
~~P1~~	2	6	1	2
P2	4	10	2	0
P3	9	2	3	12
~~P4~~	3 ~~8~~	0	1	0

برای زمان دهی آن پردازه‌هایی که در صف هستند
از بین تمام پردازه‌ها آنکه اولویت بیشتری دارد را
انتخاب کند (عدد اولویت کمتر)
اولویت یکسان ← آنکه زودتر آمده

گرسنگی (قحطی زدگی / انسداد نامعین): پردازه‌هایی با اولویت کم در
الگوریتم‌های اولویت دچار گرسنگی می‌شوند.

سالخوردگی (Aging) : با توجه به جبران

نمودار گانت (شکل بالا) :

P4	P0	P1	P2	P3

0   5   8   10   14   23

زمانی که پردازه در صف آماده منتظر می‌ماند کم‌کم اولویت را افزایش می‌دهند.
افزایش اولویت پردازه‌ای که در اثر سالخوردگی به عقب رانده شود دچار گرسنگی نمی‌شوند.

Subject : | P4 | P₀ | Date :

$\rightarrow$ ~~P₀~~ ~~P₄~~ P₃ ~~P₁~~ ~~P₂~~

(پیشنهاد امثال قبل) :

P4	P4	P0	P1	P0	P2	P3

0 2 5 6 8 10 14 23

معیارهای اولویت پردازه :

۱. داخلی : CB ↓ ، میزان حافظه ↓ ، تعداد فایل های باز ↑ ، تعداد منابع ↑ ، $\frac{IOB}{CB}$ «اگر بیشتر از یک باشد یعنی پردازش I/O است اولویت بالاتری دارد.»

۲. خارجی : نوع ، هزینه ، اهمیت ،

* SJF ← اولویت ایستا ، CB کمتر = عدد اولویت

* SRT ← اولویت پویا ، زمان باقی مانده کمتر = عدد اولویت

FIFO

۵. RR «نوبت چرخشی» Round Robin

یک برش زمانی وجود دارد «Time Quantum» ← q میزان S انجام کردن Process ها

به این ترتیب پردازه هایی که CB کمتری دارند در تعداد کمتر اجرا می شوند و پردازه هایی که CB بالایی دارند منتظر می مانند تا بارها و بارها اجرا شوند.

مزیت : دچار گرسنگی نمی شوند (به همه پردازه ها به یک چشم نگاه می کند)

اگر q خیلی بزرگ باشد ← FCFS (صف اولویت / صف عادی FIFO)

اگر q خیلی کوچک باشد ← CPU sharing ← بهره وری سیستم ↓ می شود زمانی صرف context switch می شود (از حالت اجرا متوقف می شود و یک پردازه دیگر به حالت اجرا می رود).

جلسه چهاردهم :

	PROCESS	CB	ET	FR*	q=4
2	P0	2	14	1	
2	P1	6	0	5	
11	P2	9	7	2	
1	P3	10	[illegible]	8	
	P4	2	0	0	

P1	P4	P1	P2	P3	P0	P2	P3	P2	P3

0 4 6 7 8 12 16 18 22 26 27 29

$P_1 = 7$, $P_3 = 26$

Subject : Date :

۶. زمان بندی صف چند سطحی «MLQ»

چند صف که هر کدام الگوریتم مختص به نوع و میزان CB، اندازه‌ی صف‌ها و ...)

مهم‌ترین

۱. تعداد صف‌ها ۲. الگوریتم زمان‌بندی هر صف ۳. اولویت هر صف

کاربری	محاوره‌ای	پردازه‌های سیستمی

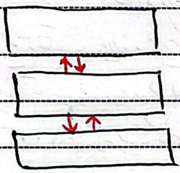
پردازه‌ها دائما بررسی می‌شوند.

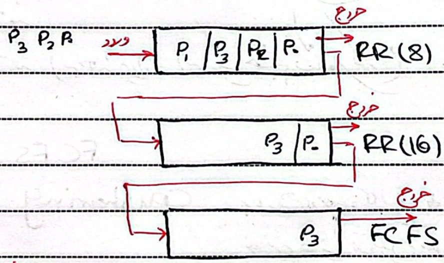
۷. زمان‌بندی صف با بازخورد چند سطحی «MLFQ»

پردازه‌ها می‌توانند در صف‌ها جابجا شوند.

۱. تعداد صف ۲. الگوریتم زمان‌بندی هر صف ۳. اولویت هر صف ۴. نحوه ارتقا پردازه‌ها

۵. نحوه تنزل (کاهش) پردازه





Process	CB	ET	زمان انتظار
P_0	2 ~~16~~	0	18
P_1	4	10	12
P_2	6	0	8
P_3	22 6 ~~30~~	0	20

P_0	P_2	P_3	P_1	P_0	P_3	P_3

0 8 14 22 26 28 44 50

Subject :	Date :

جلسه پانزدهم :

تعادل بار : ← چند پردازنده‌ای ← چند پردازنده بتوانند رشته‌ها و فرایندهای مختلف را اجرا کنند.

اینکه چه پردازنده‌ای به چه پردازنده‌ای اختصاص داده شود ← توسط OS

پشته Core

← خویشاوندی : چند پردازنده داشته باشیم که هر کدام با شمارنده‌ی رشته‌های مشترک کار کنند ← هر یک

← خویشاوندی پردازنده به پردازنده : که بهتر است یک پردازنده تا آخر اجرا توسط یک پردازنده اجرا شود و به پردازنده دیگری منتقل نشود

تعادل بار : پردازنده‌های مختلف کارهای به اندازه یکسانی برای انجام داشته باشند.

راهکارها : جابجایی : پردازه‌ای به صورت دوره‌ای اجرا شود و چک کند اندازه صف‌ها یکسان است؟ ← بسته

جابجا کشیدن : هر کدام از صف‌ها خالی شد پردازه‌ها از صف شلوغ به صف خالی منتقل می‌شوند.

فصل ششم : همگامی پردازه‌ها

به پردازه‌هایی که روی هم اثر بگذارند پردازه‌های همکار می‌گویند ← به بررسی داده‌ی مشترک (تولیدکننده - مصرف‌کننده)

اگر پردازه‌ها به صورت موازی اجرا شوند و هم‌زمان روی این داده‌های مشترک تأثیر بگذارند این تأثیر ممکن است باعث ناسازگاری شود و داده‌ی نامعتبر تولید کند.

شرط پر بودن $in == out$

خالی بودن $(in+1) \% n == out$

} یکی از خانه‌ها همیشه خالی است

Subject :　　　　　　　　　　　　　　　　　　　　　　　　　　　Date :

تولید کننده:

```
while (true) {
    // item = new item ();
    while (Counter == n)        پر بودن
        ;  // do nothing
    buffer [in] = item ;
    in = (in +1) % n ;
    Counter ++ ; }
```

هر دو تا تولید کننده همکار هستند زیرا روی متغیرهای مشترک کار می‌کنند.

مصرف کننده:

```
while (true) {
    while (Counter == 0)        خالی بودن
        ; // do nothing
    item = buffer [out] ;
    out = (out +1) % n;
    Counter -- ;
        // use item }
```

هر دو تا مصرف کننده هم همکار هستند.

```
5 Counter ++ ;                    Counter -- ;
5 R1 = Counter 5                  5 R2 = Counter 5
→ 6 R1 = R1 + 1  ✗                → ✗ 4 R2 = R2 - 1
6 6 Counter = R1 6                Counter = R2
                                  4      4
P0                                P1
```

اگر چنین اجرای P_0 به آن وقفه بخورد و بعد P_1 اجرا شود همان متغیرها را می‌بیند که اشتباه می‌شود و Counter=4 در حالیکه $Counter_0=6$ است

Counter = 5 → Counter = 6

Counter = 6

Counter = 4

اگر چنین اجرای P_1 به آن وقفه بخورد و بعد P_0 اجرا شود Counter=6 در حالیکه $Counter_1=4$ است

اسم این عمل شرایط رقابتی است که چند پردازه‌ی همکار روی داده‌های مشترک همزمان کار می‌کنند

Subject :   Date :

شرایط مسابقه: چندین پردازه همزمان میخواهند داده مشترکی را تغییر دهند

مسئله بحث: یک ناحیه مشترکی وجود دارد که توسط چندین process همزمان مورد دسترسی و تغییر قرار میگیرد

ناحیه بحرانی: قسمتی از کد پردازه که داده مشترک را تغییر میدهد.

وقتی یک پردازه در ناحیه بحرانی قرار دارد OS نباید اجازه بدهد پردازه های دیگری در ناحیه بحرانی قرار بگیرند

فرض کنیم هر پردازه درون یک حلقه مدام به ناحیه بحرانی وارد و از آن خارج میشود
که پردازه را به ۴ قسمت تقسیم کنیم

```
do{
    Entry Section          ناحیه ورود → کدهایی که برای ورود به ناحیه بحرانی لازم است
    Critical Section       ناحیه بحرانی
    Exit Section           ناحیه خروج
    remainder Section      ناحیه باقی مانده → ناحیه ای که به داده مشترک کاری ندارد
}while(true)
```

ناحیه ورود: پردازه چک کند آیا حق ورود به ناحیه بحرانی را دارد یا نه؟ و اگر حق ورود ندارد باید منتظر بماند
قبل از ورود به ناحیه بحرانی باید چک شود که میتواند وارد ناحیه بحرانی شود یا نه

۱. بررسی خالی بودن ناحیه بحرانی   ۲. اگر ناحیه بحرانی خالی است به اطلاع بقیه برساند که من در ناحیه بحرانی هستم

ناحیه خروج: به اطلاع بقیه برساند به پردازه هایی که میخواهند وارد ناحیه بحرانی شوند من آمدم بیرون و الان شما میتوانید
وارد ناحیه بحرانی شوید

همه پردازه های مشترک باید این ۴ قسمت را داشته باشند.

Subject :   Date :

مسئله ناحیه بحرانی:

یک ناحیه‌ی بحرانی وجود دارد که پردازه‌ها همگی با هم نباید در ناحیه بحرانی قرار بگیرند

راهکارهایی باید ارائه شود برای مدیریت پردازه‌های همگام و این راهکارها باید ۳ شرط را داشته باشند:

۱. انحصار متقابل: حداکثر یک پردازه باید در ناحیه بحرانی قرار بگیرد

۲. پیشروی: اگر ناحیه بحرانی خالی باشد فقط process هایی می‌توانند در ورود (یا عدم ورود) به ناحیه بحرانی اثر بگذارند که در بخش باقی‌مانده‌ی کدشون نباشند.

۳. انتظار محدود: تعداد دفعات انتظار در ناحیه ورود محدود باشد (متناهی باشد)

راه حل «Peterson»:

برای ۲ پردازه - حل مسئله ناحیه بحرانی $P_i , P_j \quad j = 1 - i$

```
int turn;                  // نوبت
boolean flag[2];           // هر پردازه‌ای که به ناحیه‌ی بحرانی رفت flag آن true می‌شود
do{                        // پردازه‌ی Pi
   flag[i] = true
ES turn = j ;              // تعارف می‌کند. چون می‌گوید اگر j process خواست وارد شود، همین الان جا بدهد
   while (flag[j] && turn == j)
      ; // do nothing
   CS                      // ناحیه بحرانی
   flag[i] = false         // ناحیه خروج
   RS                      // ناحیه باقی‌مانده
} while (true);
```

برای کد پردازه‌ی P_j کافی است جای i و j عوض شود

وقتی می‌گیم P_i و P_j همروند اجرا می‌شوند یعنی دستورات آنها به هر ترتیبی ممکن است اجرا می‌شوند.

Subject :   Date :

جلسه هفدهم :

بررسی شروط :

۱. انحصار متقابل : متغیر turn یه مقدار را دارد پس هر دو با هم نمی‌توانند وارد ناحیه بحرانی شوند.

۲. انتظار محدود : هر پردازه هنگام ورود به ناحیه بحرانی اول نوبت را به پردازه دیگری می‌دهد.

۳. پیشرفت : پردازه‌ها در RS تاثیری در ورود یا خروج یکدیگر ندارند.

مشکلات :

۱. فقط برای ۲ پردازه است.

۲. پشتیبانی سخت افزاری ندارد.

راه حل‌ها :

۱. سخت افزاری : خود سخت افزار ناحیه بحرانی را کنترل می‌کند.

۲. نرم افزاری :

در هر دو راه حل سخت افزاری و نرم افزاری و عیش در ورود باید قفلی در اختیار گرفته شود و بعد وارد ناحیه بحرانی شود و در هنگام خروج قفل آزاد شود

```
do {
acquire lock
   CS
release lock
   RS
} while(true);
```

Subject : Date :

دستورات سخت افزاری پیاده سازی ورود به ناحیه بحرانی

Test and Set

Swap

• Test And Set

```
boolean testAndSet (boolean *target)
{
    boolean returnValue = *target;
    *target = true;
    return returnValue; }
```

مقدار target را برمی‌گرداند.

مقدار true را در target قرار می‌دهد.

```
P_i:
do {
    while (testAndSet(&lock))
        ; // do nothing
    CS
    lock = false;
    RS
}
```

① lock را true کرده ← ورود به ناحیه بحرانی

② مقدار false را برگردانده ← وارد ناحیه بحرانی شود

آزاد کردن قفل

چون دستور TestAndSet مقدار lock را بررسی کرده lock را هم‌زمان وارد ناحیه بحرانی شده است.

- Swap

```
void Swap(booleam *a, booleam *b)
{
  booleam temp=*a;
   *a = *b;
   *b= temp; }
```

```
Pi
do{
  key=true;
  while(key==true)
     Swap(&lock, &key);
  CS
  lock=false;
  RS
}
```

متغیر local ، p_i «مشترک نیست».

جای مقدار lock, key عوض می‌شود «بدون وقفه».

: Swap

۱. خودش را وارد ناحیه بحرانی می‌کند با false کردن key

۲. اجازه ورود به بقیه نمی‌دهد با true کردن lock

شرط انحصار متقابل رعایت می‌شود

جلسه هجدهم:

در هیچ یک از دو دستور شرط انتظار محدود رعایت نمی‌شود برای حل این مشکل به یک سری متغیرهای کمکی نیاز داریم (waiting)

اگر waiting[i]=T یعنی پردازه‌ای منتظر ورود به ناحیه بحرانی است.

Subject :   Date :

P.

```
do{
    waiting[i] = true;
    key = true;
    while(waiting[i] && key)
        key = testandset(&lock);
    waiting[i] = false
        CS
    j = (i+1) % n;
    *while(j != i && !waiting(j))
        j = (j+1) % n;
    if (j == i)
    lock = false
    else
    waiting[j] = false
    RS
} while(true)
```

(ES: بخش ورود — Exit: بخش خروج)

- `waiting[i] = true;` ← به بقیه پردازه‌ها گفته منتظر ورود به ناحیه بحرانی است
- `key = true;` ← متغیر local
- `while(waiting[i] && key)` ← زمانی متغیر key، false می‌شود که lock، false باشد
- `j = (i+1) % n;` ← آیا process دیگری وجود دارد که waiting آن true باشد یا خیر؟ اگر هست waiting آن را false کنیم و به آن اجازه دهیم که از ناحیه ورود رد شود و وارد ناحیه بحرانی شود
- `if (j == i)` ← process منتظر ورود به ناحیه بحرانی نداریم
- `lock = false` ← آزاد کردن قفل
- * یا waiting آن true باشد یا i = j شود
- `waiting[j] = false` ← پردازه j که منتظر ورود به ناحیه بحرانی است.

بدترین حالت یک پردازه n-1 بار در حلقه while منتظر بماند که شرط انتظار محدود برقرار است.

سمافور "Semaphore"

یکی از ابزارهای نرم‌افزاری برای مدیریت ناحیه بحرانی

متغیری از نوع صحیح که جدای از مقداردهی اولیه‌اش تنها با دو دستور wait, Signal مقدار آن تغییر می‌کند.

```
wait(S){
    while(S <= 0)
        ; //do nothing
    S--; }

Signal(S){
    S++; }
```

Subject :   Date :

استفاده برای مدیریت ناحیه بحرانی :

```
Pi        Semaphore S=1 ;
do{
wait(S);
CS ;
Signal(S);
RS; }
```

هر process ای برای ورود به ناحیه بحرانی wait(S) می‌کند

تا زمانی که pi در ناحیه بحرانی است Semaphore=0 است

چون Semaphore مقدار دودویی دارد شرط انحصار متقابل برقرار می‌شود

دیگر شرط انتظار محدود رعایت نمی‌شود

انواع Semaphore:

دودویی: ۰ ، ۱ ← معمولاً برای مدیریت ناحیه بحرانی از آن استفاده می‌شود به آن mutex می‌گویند

شمارشی: اگر مقدار اولیه Semaphore بیشتر از ۱ باشد، شمارشی است در مدیریت منابع کاربرد دارد.

```
S=4
wait(S)
use resource
release resorce
Signal(S);
```

حداکثر ۴ برنامه می‌توانند از این منبع استفاده کنند

مدیریت منابع با Semaphore شمارشی با دارای ۴ نمونه

مقدار اولیه Semaphore = تعداد منبع

یکی دیگر از کاربرد Semaphore دودویی در همگام‌سازی برنامه است ← اگر بخواهیم یک دستور حتماً قبل از یک دستور دیگر اجرا شود.

```
Semaphore x=0

1  S1 ;          | wait(x);  3
2  Signal(x);    | S2 ;      4
```

S1 حتماً قبل از S2 اجرا می‌شود

جلسه نوزدهم :

انتظار مشغول "Busy waiting"

پردازه به اندازه برش زمانی خود cpu را میگیرد در حالیکه کاری انجام نمی‌دهد و منتظر است.

حل مشکل انتظار مشغول :

```
typedef struct{
  int value;
  struct process *list;
 } semaphore
  wait(semaphore *S){
  S -> value --;
  if(S -> value < 0){
     add this process to S -> list
   block();
  }
}
signal(semaphore *S){
  S -> value ++;
  if(S -> value <= 0){
   remove a process p from S -> list
  wakeup(p); }
}
```

مقدار semaphore را مشخص می‌کند.

waiting list هم لیست پیوندی از پردازه‌ها

پردازه به لیست پردازه‌های منتظر اضافه می‌شود تا در برش زمانی اجرا شود.

اگر کوچکتر از صفر بود یعنی در لیست پردازه‌های منتظر سیگنال هستند.

P_i و P_j را wake up میکند و به حالت آماده می‌برد و بعد wait را اجرا میکند و به ناحیه بحرانی می‌رود.

مقدار semaphore
- + : چند process در لیست است.
- − : n تا process در لیست هست.
- 0 : دیگر هیچ process در لیست نیست.

مشکل انتظار مشغول با block کردن و مشکل انتظار محدود با wake up کردن حل می‌شود.

مشکلات Semaphore :

۱. بن بست (Dead lock) و قحطی زدگی (Starvation)

قحطی زدگی: پردازه‌ای در queue سمافور wait و چندین بار نادیده گرفته می‌شود و در نهایت برای همیشه در حالت block باشد.

بن بست: یعنی پردازه هیچ گاه از حالت block خارج نشوند.

۲. وارونگی اولویت

یک پردازه با اولویت بالا منتظر پردازه با اولویت پایین باشد.

OS وقتی وارونگی اولویت رخ بدهد به وسیله ارث بری اولویت حل می‌کند.

۳. ارث بری اولویت

اگر یک پردازه منابع دیگر پردازه با اولویت بالاتر را در اختیار دارد، اولویت را هم از پردازه با اولویت بالاتر ارث ببرد تا پردازه‌های با اولویت پایین‌تر نتوانند cpu را از آن بگیرند.

مسائل کلاسیک همگامی پردازه‌ها: ۱. بافر محدود ۲. خواننده‌ها - نویسنده‌ها ۳. فیلسوف‌های خوش خوراک

۱. مسئله بافر محدود: تولیدکننده و مصرف کننده — empty = n (اندازه بافر) , full = 0

```
do {
    item = new Item();
    wait(empty);        -> n-1, n-2, ...
    wait(mutex);
    buffer[count++] = item;
    signal(mutex);
    signal(full);
} while(1)
```

تولید کننده، آیتم تولید می‌کند و در بافر قرار می‌دهد.

اگر بافر پر شد منتظر می‌ماند. — چک می‌کند ناحیه بحرانی خالی باشد.

نقض انحصار متقابل (حاشیه، کنار خطوط wait(mutex) تا signal(mutex))

آیتم را در بافر قرار می‌دهد.

مصرف کننده از buffer، item برمی‌دارد.

Subject : Date :

```
do{
  wait(full)
  wait(mutex)
  item = buffer[count--];
  Signal(mutex)
  Signal(empty);}
  while(true)
```

مصرف کننده

مصرف کننده منتظر بافر خالی است و چیزی برای مصرف وجود ندارد.

چون mutex مشترک است تعیین می‌کند که

متغیر count، buffer همزمان توسط

چند پردازه تغییر نکند.

یک آیتم از بافر حذف شده، یک خانه خالی اضافه شود و تولیدکننده‌ای که

منتظر بوده است از wait(empty) رها می‌شود.

جلسه بیستم :

۲. مسئله خوانندگان نویسندگان : پردازه‌های همگامی که تعدادی از آنها فقط فایل را می‌خوانند و تعدادی دیگر روی فایل تغییر ایجاد می‌کنند.

```
do{
  wait(wrt);
  //CS: write the hile
  Signal(wrt);
  RS
}while(true);
```

نویسندگان:

برای آنکه تعیین کنیم دو نویسنده فایل را تغییر ندهند

از سمافور دودویی استفاده می‌کنیم.

rd Count ← تعداد خواننده‌های در حال حاضر را حساب می‌کند.

Subject : Date :

خواننده‌ها:

```
do {
    wait(mutex)
    rd Count ++;
    if(rd Count ==1)
        wait(wrt);
    signal(mutex)
    //CS: read file
    wait(mutex);
    rd Count --;
    if(rd Count==0)
        signal(wrt);
    signal(mutex);
}while(true);
```

سمافور دودویی mutex برای متغیر rd Count استفاده می‌شود (جلوگیری کردن متغیر ++)

جلوگیری می‌کند هر زمان دو پردازه rd count را اضافه کنند.

اولین خواننده‌ای است که فایل را می‌خواند

زیرا wrt=1 باشد به نویسنده اجازه ورود نمی‌دهد یعنی wrt==0 باشد

خواننده‌ها همزمان نویسنده نباشند. ناحیه بحرانی بین خواننده و نویسنده

تغییر متغیر rd count باید با سمافور mutex باشد.

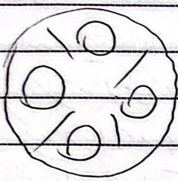
آخرین خواننده در حال ترک فایل است و به نویسنده اجازه دسترسی به فایل می‌دهد

شرایطی که توسط الگو تضمین می‌شود:

1. چند نویسنده فایل را تغییر نمی‌دهند.
2. تا زمانی که فایل در حال نوشته شدن است هیچ خواننده‌ای فایل را نمی‌تواند بخواند.
3. تا زمانی که حداقل یک خواننده در حال خواندن فایل است هیچ نویسنده‌ای امکان تغییر فایل ندارد.
4. متغیر rd Count در هر لحظه تنها توسط یک خواننده تغییر می‌کند.

Subject : Date :

۳. مسئله فیلسوفان خوش خوراک:



فیلسوفان: فکر کردن / خوردن

از چوب های مشترک استفاده می کنند.

Chopstick ← ناحیه بحرانی

```
do{
  wait(chopstick[i]);
  wait(chopstick[(i+1) % n]);
   //eat
  signal (chopstick[i]);
  signal (chopstick[(i+1)%n]);
   // think
 } while (true)
```

فیلسوف P از چوب i و i+1 استفاده می کند.

بدترین حالت زمانی است که همه فیلسوف ها چوب خود را بردارند که به بن بست می رسند ← راه حل ها:

۱. حداکثر ۴ فیلسوف حق ورود به غذاخوری را داشته باشند ← حداقل به ۱ نفر ۲ چوب می رسد پس غذا را می خورد و چوب ها را آزاد می کند ← سایرین شانس

۲. تست آزاد بودن هر دو چوب قبل از wait کردن آنها

۳. تناوب اضافی: فیلسوف شماره زوج اول i و سپس i+1 را wait کند و فیلسوف شماره فرد اول i+1 و سپس i را wait کند.

Subject :   Date :

فصل هفتم: بن‌بست ۴

جلسه بیست و یکم:

بن‌بست: اگر پردازه‌ای که منتظر است هیچ وقت نتواند از حالت انتظار خارج شود چون منابعی که درخواست کرده در اختیار پردازه‌های منتظر دیگر است.

مفاهیم:

منبع:
- تک نمونه‌ای: وقتی در سیستم که یک پردازنده‌ای فقط یک cpu داریم / پرینتر / CD-ROM
- چند نمونه‌ای: با فرض اینکه از یک منبع چند نمونه داریم.   تولیدکننده - مصرف‌کننده:

منبع:
- سخت‌افزاری: cpu ، printer
- نرم‌افزاری (منطقی): سمافورها، فایل‌ها

مراحل استفاده از منبع:

۱. درخواست منبع ← باز کردن لیست   request , wait , open

۲. استفاده از منبع   use

۳. رهاسازی منبع   release , signal , close

جدول منابع: در واقع جدولی از os است که مشخص می‌کند هر منبعی که در os وجود دارد آزاد است یا نه

Subject : Date :

شرایط بن بست (لازم و کافی نیست و همه ۴ شرط باید اتفاق بیافتند)

۱. انحصار متقابل: از هر منبع حداکثر یک پردازه می تواند استفاده کند. unshareable

۲. گرفتن و منتظر ماندن: پردازه ها حداقل یک منبع گرفته باشند و حداقل منتظر یک منبع دیگر باشند.

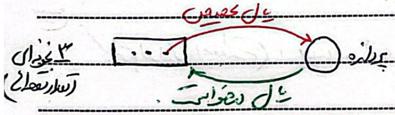
۳. نبود پس گرفتن: آزادسازی منبع فقط توسط خود پردازه انجام شود.

۴. انتظار چرخشی: زمانی بن بست داریم که پردازه ها به صورت چرخشی منتظر هم باشند.

$P_1, P_2, \dots, P_n$

گراف تخصیص منبع:

این گراف مشخص می کند از هر منبع چند نمونه داریم، هر نمونه در اختیار کدام پردازه ها است. و پردازه ها به چه منابعی نیاز دارند.



این گراف حالت فعلی سیستم را نشان می دهد و مدام در حال تغییر است.

در گراف های چند نمونه ای نمی توان بن بست را تشخیص داد ← وجود دور شرط لازم است اما کافی نیست. در گراف تک نمونه ای اگر دور وجود داشته باشد شرط لازم و کافی برای وجود بن بست است.

روش های برخورد با بن بست:

Prevent — الف) پیشگیری از بن بست

avoid — ۱. شرایطی را فراهم کنیم که سیستم دچار بن بست نشود ← ب) اجتناب از بن بست

۲. تشخیص و حل

۳. هیچ کاری انجام ندهیم

Subject : Date :

دلایل برای لزوم:

۱. احتمال وقوع بن‌بست خیلی پایین است.

۲. مدیریت و حل بن‌بست هم پیچیده است و هم باعث کاهش بازدهی می‌شود

۳. خود برنامه‌نویس که مدیریت کند.

الف) پیشگیری از بن‌بست:
- ۱. نقض انحصار متقابل
- ۲. نقض گرفتن و منتظر ماندن
- ۳. نقض نبود پس‌گیری
- ۴. نقض انتظار چرخشی

اگر یکی از ۴ شرط بن‌بست نقض شود بن‌بست اتفاق نمی‌افتد.

ب) اجتناب از بن‌بست

که به کمک اطلاعات اضافه‌ای که پردازه‌ها و منابع دارند تعیین می‌کنند اگر درخواست پردازه‌ها برآورده شود هیچگاه بن‌بست اتفاق نمی‌افتد.

پیشگیری از بن‌بست: **جلسه بیست دوم**

۱. نقض انحصار متقابل: منابع تا حد امکان به صورت مشترک استفاده شوند.

روش قطعی برای پیشگیری از بن‌بست نیست زیرا برخی منابع ذاتاً قابل اشتراک نیستند ← سمافورها، صف منبع

۲. نقض گرفتن و منتظر ماندن:

پروتکل ۱: هر پردازه قبل اجرا تمام منابع مورد نیاز خود را دریافت کند ← نقض منتظر ماندن

پروتکل ۲: تنها در صورتی به پردازه اجازه می‌دهیم منبع (جدید) درخواست کند که هیچ منبعی در اختیار نداشته باشد یا تمام منابع قبلی را آزاد کرده باشد ← نقض گرفتن

Subject : Date :

ایرادات:

۱. پروتکل ۱ پردازه‌ها را دچار گرسنگی می‌کند.

۲. در پروتکل ۲ منابع همزمان قابل استفاده نیستند.

۳. در هر دو پروتکل بازدهی و بهره‌وری شدیداً کاهش پیدا می‌کند.

۳. نقض نبود پس‌گیری:

پروتکل ۱: اگر یک پردازه به منبعی نیاز داشت که آزاد نبود تمام منابع از پردازه پس گرفته شود.

پروتکل ۲: پردازه P منبع A را نیاز دارد که در اختیار Q است:

الف) اگر Q در حال اجرا است ← P منتظر بماند.

ب) اگر Q در حالت انتظار است ← منبع A از Q به P داده شود.

ایرادات:

۱. در پروتکل ۱ ممکن است منابع غیرقابل بازپس‌گیری شوند.

۲. در هر دو پروتکل گرسنگی داریم.

۳. کاهش بازدهی و بهره‌وری

۴. نقض انتظار چرخشی: (شماره‌ای برای تخصیص شماره به هر منبع استفاده کنیم)

پروتکل ۱: پردازه‌ها تنها می‌توانند منابع را به ترتیب افزایش شماره درخواست کنند.

۱	۲	۳		n →	X
A	B	C	...	F	A

پروتکل ۲: اگر پردازه‌ای P منبع i را درخواست کند باید ابتدا تمام منابع ≥ i را آزاد کند.

Subject : Date :

ایرادات:

۱. ممکن است منابع غیرقابل برگشت باشند.

۲. در شرایطی اشتراکی.

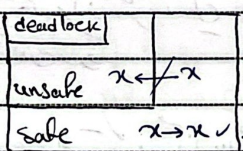
اجتناب از بن‌بست: ۱. گراف تخصیص منبع (یک نمونه‌ای) ۲. الگوریتم بانکدار

حالت امن: سیستم در حالت امن است اگر دنباله امنی از پردازه‌ها موجود باشد.

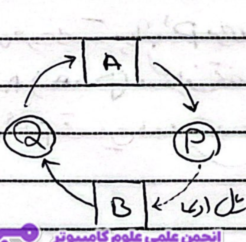
دنباله امن: دنباله $P_1, P_2, ..., P_n$ امن است اگر هر پردازه P_i با استفاده از منابع آزاد و منابعی که در اختیار P_j $(j<i)$ است بتواند خاتمه یابد.

حالت ناامن: اگر حالت سیستم امن نباشد.

حالت ناامن لزوماً بن‌بست نیست، هر بن‌بست زیرمجموعه‌ای از حالت ناامن است.



حالت سیستم ممکن است با انجام هر تخصیص منبع تغییر کند.
اگر حالت از امن به امن تغییر پیدا کرد، تخصیص منبع انجام می‌شود و اگر از حالت امن به ناامن تغییر پیدا کرد تخصیص منبع انجام نمی‌شود (حتی اگر منبع آزاد باشد) و برای جلوگیری از بن‌بست از آن جلوگیری می‌کنیم.



یال ادعا: پردازه در آینده به منبع نیاز دارد.

P, Q

Q, P

چون P و Q با منابع آزاد تمام نمی‌شوند حالت ناامن است.

Subject : Date :

جلسه بیست و سوم :

مثال :

حالت امن است؟

باید ببینیم می‌توانیم دنباله امنی پیدا کنیم

A → 12 تعداد منابع

	max	Allocate
P_0	10	5
P_1	4	2
P_2	9	2
		9 (مجموع)

P_1 , P_0 , P_2

A → 12 − 9 = 3 (آزاد) → (P_1 اجرا شود) → 5 → (P_0 اجرا شود) → 10

الگوریتم گراف تخصیص منبع :

با یال ادعا مشخص می‌کنیم پردازه‌ها در آینده به چه منابعی نیاز دارند. وقتی یک یال ادعا تبدیل به یال درخواست شد الگوریتم چک می‌کند اگر با تبدیل یال درخواست به تخصیص دور به وجود آید، این کار را انجام نمی‌دهد تا حالت سیستم ناامن نشود.

به ازای هر درخواست اگر منبع آزاد باشد تخصیص بلافاصله صورت نمی‌گیرد، چک می‌شود با در نظر گرفتن یال‌های ادعا دور به وجود می‌آید یا نه. اگر تشکیل دور می‌داد حتی با وجود آزاد بودن منبع آن را تخصیص نمی‌دهد.

به ازای هر درخواست پردازه P برای منبع A اگر تبدیل یال درخواست P → A (P به A) به یال تخصیص A → P (A به P) در گراف تخصیص منبع با در نظر گرفتن یال‌های ادعا دور ایجاد نشود منبع A به پردازه P تخصیص داده می‌شود، در غیر این صورت P منتظر A می‌ماند.

Subject :   Date :

الگوریتم بانکدار دارد: • الگوریتم ایمنی

• الگوریتم درخواست منبع

n : تعداد پردازه‌ها   m = تعداد منابع

max → n×m ماتریس   max[i][j] = k P_i به حداکثر k تا از منبع P_j نیاز دارد

Allocation → n×m ماتریس   Alloc[i][j] = k P_i k تا از منبع P_j دارد

need → n×m ماتریس   need = max − Allocation

Available → آرایه m تایی   Available[j] = k   k تا از منبع P_j آزاد است

• الگوریتم ایمنی: چک می‌کند حالت سیستم امن است یا نه:

آرایه n تایی

۱. مقداردهی اولیه: یعنی هیچکدام از پردازه‌ها تمام نشده   Finished ← False , work ← Available

۲. i را پیدا کن که: دنبال پردازه‌ای می‌گردد که بتواند نیازش را تامین کند

Finished[i] == False
and
need[i] ≤ work

آرایه m تایی ← آرایه m تایی

اگر i پیدا نشد برو به ۴

۳. finished[i] = true
work += Allocation[i]
برو به مرحله ۲

۴. اگر برای تمام i ها finished[i] = true باشد، حالت سیستم امن است در غیر این صورت اگر i وجود داشته باشد که finished(i) = false باشد حالت سیستم ناامن است.

جلسه بیست و چهارم:

• الگوریتم درخواست

Request → آرایه n×m

Request[i][j] → k : پردازه i ام k نمونه از منبع j ام درخواست کرده

الگوریتم:

۱. اگر $Request_i \le need_i$ برو به ۲ در غیر این صورت خطا

۲. اگر $Request \le Available$ برو به ۳ در غیر این صورت ← انتظار

اگر یکی از درایه‌ها بزرگتر باشد انتظار

۳. تغییرات زیر را انجام بده:

$Allocation_i \mathrel{+}= Request_i$

$Available \mathrel{-}= Request_i$

$need_i \mathrel{-}= Request_i$

الگوریتم ایمنی را اجرا می‌کنیم

اگر true بود ← منابع مورد نیاز پردازه i ام به پردازه تخصیص داده می‌شود.

اگر false بود ← منابع را تخصیص نده و تغییرات را برگردان

مثال: n=5 , m=3

	Allocation			max			need = max − Allocation		
	A	B	C	A	B	C	A	B	C
P_0	۰	۱	۰	۷	۵	۳	۷	۴	۳
✓ P_1	۲	۰	۰	۳	۲	۲	۱	۲	۲
P_2	۳	۰	۲	۹	۰	۲	۶	۰	۰
P_3	[illegible]	۱	۱	۲	۲	۲	۰	۱	۱
P_4	۰	۰	۲	۴	۳	۳	۴	۳	۱

Subject : Date :

Available A B C Finished = (F,F,F,F,F)
۳ ۳ ۲

از منبع B که ۵ تا داریم (۳ تا آزاد + ۲ تا تخصیص یافته)
از منبع C که ۷ تا (۲ تا آزاد + ۵ تا تخصیص یافته)

work (3,3,2) need[i] ≤ work need[i] ≤ work →
work += Allocation[i] → work (5,3,2) → Finished = (F,T,F,F,F)

work (3,3,2) P_1→ (5,3,2) P_3→ (7,4,3) P_4→ (7,4,5) P_0→ (7,5,5) P_2→

و بنابراین این $P_1 P_3 P_4 P_0 P_2$ پاسخ ما است.

- زمانی که سیستم پردازه P_i درخواست منابع را داد: request (A, B, C)
برای اینکه سیستم آیا باید به درخواست جواب بدهیم یا نه

request$_i$ ≤ need$_i$
request$_i$ ≤ Available

تغییرات را اعمال کنیم و الگوریتم ایمنی را روی این کنیم

need[i] ≤ work

request باید از need کمتر باشد و کمتر است و مثبت است باید درخواست از منابع آزاد هم کمتر باشد که هست

	Allocation			max			need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
P_1	2	3	0	3	2	2	1	0	0
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1
P_4	0	0	2	4	3	3	4	3	1

Available A B C
2 3 0

جلسه بیست و پنجم :

ایراد الگوریتم بانکدار :

منبع آزاد داریم اما چون ممکن است به بن بست برسیم خیلی از حالت های سیستم را از دست می دهیم ← کاهش کارایی

و بهره وری

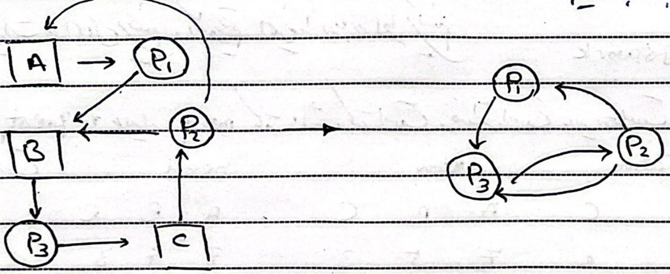
۲. تشخیص و حل

روش های تشخیص : ۱. الگوریتم تشخیص در منابع تک نمونه ای : گراف انتظار کشی

۲. الگوریتم تشخیص بن بست در منابع چند نمونه ای

گراف انتظار کشی : همان گراف تخصیص منبع است که گره منابع از آن حذف شده و نشان می دهیم هر پردازه منتظر کدام پردازه است.

اگر دور به وجود آمد، حتماً بن بست داریم



$P_i \square P_0$

سیکل به طول ۲

منابع چند نمونه‌ای

Allocation   Request   Available.

1. $Work \leftarrow Available$

اگر $Allocation \neq 0$ باشد برای i ها $finished[i] = F$ (منبع داشته)

اگر $Allocation = 0$ باشد برای i ها $finished[i] = T$ (زیرا فرایندی که هیچ منبعی در اختیار نگرفته نمی‌تواند باعث بن‌بست شود پس آن را بررسی نمی‌کنیم)

2. اندیس i را پیدا کن که $finished[i] = F$ & $Request_i \leq Work$

اگر پیدا نشد برو به 4 در غیر این صورت برو به 3

3. $finished[i] = T$, $Work += Allocation_i$;

4. اگر به ازای تمام i ها $finished[i] = T$ باشد یعنی سیستم امن است و در غیر این صورت بن‌بست پیش آمده

زمان اجرای الگوریتم تشخیص:

1. زمانی که درخواست بلافاصله پاسخ داده نشود.

2. بازه زمانی مشخص

3. اگر بهره‌وری CPU کمتر از یک درصد شد.

Subject :   Date :

روش‌های حل: ۱. ختم پردازه ← تمام پردازه‌های بن‌بست
یکی یکی
۲. پس گرفتن منبع

• ختم پردازه: چه پردازه‌ای خاتمه یابد؟
۱. اولویت کمتر
۲. مدت محاسبات کمتر
۳. تعداد منابع تخصیص یافته کمتر
۴. تعداد منابع مورد نیاز بیشتر
۵. نوع پردازه ← تعاملی، محاوره‌ای یا دسته‌ای

• پس گرفتن منبع: ۱. انتخاب قربانی
۲. عقب‌گرد ← منبع قابل پس گرفتن باشد ← printer عقب‌گرد نیست.
۳. قحطی زدگی ← همیشه یک پردازه قربانی نشود

فصل ۸: حافظه اصلی

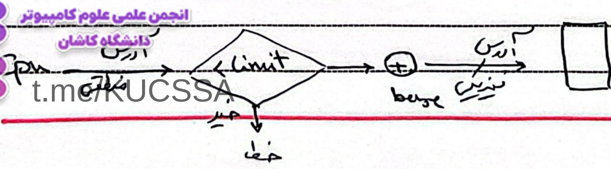
آدرس فیزیکی: آدرس واقعی که به خانه‌ای از RAM اشاره دارد.
آدرس منطقی (مجازی): آدرسی که cpu برای واکشی و تنفیذ درخواست می‌کند
هر پردازه‌ای که برای اجرا وارد می‌شود، مقدار حد و پایه خود را در ثبات base , limit ذخیره می‌کند

ثبات پایه: آدرس شروع پردازه base
ثبات حد: اندازه پردازه limit

base: 1024
limit: 300

1024
300 ←

آدرس فیزیکی = آدرس منطقی + base ← پس چک می‌شود از limit بیشتر نشود.



تخصیص حافظه:

- همجوار: تمامی قسمت های مربوط به حافظه یک پردازه در حافظه اصلی کنار هم قرار میگیرند.
- غیر همجوار: قسمت های مختلف پردازه در مکان های مختلفی از حافظه قرار میگیرند.

• تخصیص همجوار:

الف) پارتیشن بندی ثابت:

حافظه به اندازه های ثابت پارتیشن بندی می شود و پردازه ها در پارتیشن قرار میگیرند.
و اگر اندازه یک پردازه بیش از یک پارتیشن باشد در ۲ پارتیشن قرار میگیرد.

ایراد: حافظه به بخش هایی که قابل استفاده نیست تقسیم می شود و حافظه به صورت بهینه استفاده نمی شود.
internal fragmentation

ب) پارتیشن بندی متغیر:

پارتیشن بندی ای وجود ندارد و به هر پردازه حافظه به نسبت اندازه پردازه تعلق میگیرد.

ایراد: پردازه ها پس از اجرا حافظه را خالی میکنند و حافظه پر از حفره هایی میشود که برای هیچ پردازه ای قابل استفاده نیست. external fragmentation

قانون ۵۰٪: اگر n بلاک تخصیص حافظه داشته باشیم n کز آنها بدون استفاده است وجود داریم $\frac{1}{3}$ حافظه بلا استفاده است.

راه حل:

۱. تخصیص غیر همجوار حافظه
۲. فشرده سازی: همه پردازه ها به سمت بالا شیفت پیدا کنند و حفره ها کنار هم قرار بگیرند.

Subject : Date :

• روش های تخصیص پیوسته

۱. first fit : حفره ها را از اول چک کن. اولین حفره ای که اندازه اش بزرگتر از اندازه برنامه باشد برنامه را در آن حفره قرار بده.

۲. next fit : برنامه جدید حفره ها را از جایی که برنامه قبلی در حفره واقع شده چک کند و اگر پیدا نکرد به اول برگردد.

۳. Best fit : همه حفره ها را چک کن و کوچکترین حفره ای که اندازه آن بزرگتر از برنامه است را انتخاب کن.

۴. worst fit : از بین تمام حفره ها بزرگترین حفره ای که اندازه آن بزرگتر از حافظه است را پیدا کن.

مشکل ۳،۴ بررسی کل حافظه است.

مشکل ۱،۲،۳ ـــ external fragmentation

• تخصیص غیر همجوار:

الف) صفحه بندی (Paging):

برنامه به صفحه ها، حافظه به تکه ها (frame) تقسیم شود ← اندازه page ها، frame ها یکی است.

هر page یک frame قرار می گیرد ← ترتیب و کنار هم بودن اهمیتی ندارد.

جدولی را نشان می دهد page در کدام frame قرار گرفته.

• با استفاده از page Table می توانیم آدرس منطقی را به فیزیکی تبدیل کنیم.

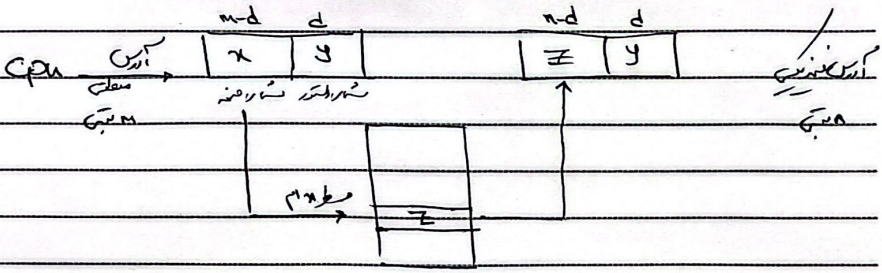
0	1
1	
2	

→ page 0 در frame 1 قرار گرفته.

اندازه هر صفحه = اندازه هر قاب = 2^d

Subject :

Date :



تعداد صفحات 2^{m-d} تعداد ردیف 2^{n-d}

اندازه هر ردیف 2^d اندازه جدول صفحه $(n-d) \times 2^{m-d}$

اندازه آدرس حافظه مجازی m بیت ← فضای آدرس مجازی 2^m بایت

اندازه آدرس فیزیکی n بیت ← فضای آدرس فیزیکی 2^n (حافظه اصلی)

باید بین page table و این حافظه‌ای دیگر را برقرار کنیم