آریا خلیق
۹۵۲۴۰۱۴
تمرین شماره ۶ OS!

۱)
در معماری ۳۲ بیتی هر فرایند کاربر یک فضای آدرس ۳۲ بیتی دارد که برابر با ۴گیگابایت آدرس مختللف برای هر فرایند می‌شود که نصف آن هم توسط سیستم‌عامل گرفته می‌شود.
زمانی که یک فرایند ایجاد می‌شود منطقاً می‌تواند از تمام ۲گیگابایت آدرس و خانه خود استفاده کند. این فضا به صفحه‌هایی با اندازه مساوی تقسیم می‌شود. که بعضی از آن‌ها به حافظه اصلی می‌آیند. سیستم‌عامل حافظه اصلی را region بندی می‌کند. هر منطقه می‌تواند سه حالت زیر را داشته باشد:
در دسترس:آدرس‌هایی که به وسیله فرایند استفاده نشده‌اند.
رزروشده: آدرس‌هایی که virtual memory manager برای یک فرایند دیگر رزرو کرده. (مثلاً ادامه stack یک برنامه)
commited: آدرس‌هایی که virtual memory manager برای استفاده فرایندها و صفحه‌های آن‌ها هستند. این صفحات می‌توانند بر روی دیسک یا حافظه مجازی قرار داشته باشند.

زمانی‌که یک فرایند وارد سیستم می‌شود چون فضای حافظه اصلی ما محدود است قسمت‌هایی از آن وارد حافظه اصلی می‌شود.
زمانی که حافظه اصلی خالی است صفحه‌ها به راحتی وارد حافظه اصلی می‌شوند (page fault رخ می‌دهد و یک صفحه از برنامه بدون بیرون انداخته شدن وارد سیستم میشود). وقتی که فضای حافظه اصلی محدودتر می‌شود صفحه‌هایی که اخیراً ارجاعی نداشته‌اند بیرون انداخته می‌شوند. حتی زمانی که حافظه اصلی خالی است ویندوز به فرایندهایی که حافظه زیادی گرفته‌اند نگاه کرده و صفحه‌هایی از آن‌ها که زیاد استفاده نشده‌اند و اخیراً ارجاعی نداشته‌اند را بیرون می‌اندازد. به این ترتیب زمانی که یک فرایند می‌خواهد وارد حافظه شود با مشکل حافظه برخورد نمی‌کند و سیستم‌عامل پیشگیرانه عمل می‌کند.

۲)
این الگوریتم، بهبود الگوریتم clock است که two-handed clock نامیده می‌شود. این الگوریتم از یک refrence bit استفاده می‌کند که زمانی که یک صفحه وارد می‌شود، به ۰ مقداردهی می‌شود و زمانی که یک بار برای خواندن/نوشتن به آن مراجعه می‌شود، ۱ می‌شود. عقربه fronthand در پیج‌های داخل فریم گذر می‌کند و بیت ارجاع آن‌ها را ۰ می‌کند. مدتی بعد عقربه backhand از روی پیج‌ها گذر کرده و بیت ارجاعشان را چک می‌کند. اگر ۱ باشد یعنی پس از گذر عقربه fronthand از روی آن حداقل یک‌بار ارجاع داشته است و کاری با آن نخواهیم داشت. اگر بیت ۰ باشد یعنی بعد از عقربه fronthand ارجاعی نداشته و کاندیدای خروج است و در یک لیست که کاندیداهای خروج در آن قرار دارند، گذاشته می‌شود.
دو فاکتور در عمل‌کرد این الگوریتم دخیل هستند:

- پریود عقربه‌ها چند دور در ثانیه است.
- فاصله عقربه fronthand با backhand چقدر است.

مورد اول می‌تواند بسته به شرایط تغییر کند. یعنی سریع‌تر یا کندتر شود. هرچقدر حافظه کمتری داشته باشیم سریع‌تر می‌شود تا حافظه بیش‌تری باز کند.
مورد دوم هم با مورد اول تغییر می‌کند و سرعشان بستگی به حافظه خالی دارد.

۳)
در داخل پردازنده پنتیوم دو نوع جدول وجود دارد:

- GDT : توصیف کننده محلی
- LDT : توصیف کننده جهانی

هر برنامه LDT خودش را دارد در حالی که یک GDT کلی وجود دارد.
X86 به تعداد ۱۶هزار سگمنت مجزا دارد و هر کدام از این سگمنت‌ها ۱ میلیارد کلمه ۳۲ بیتی دارند. در این پردازنده تعداد سگمنت‌هایمان کم است. ایده این است که تعداد کمی برنامه هستند که بیش از ۱۰۰۰ سگمنت می‌خواهند ولی برنامه‌های زیادی هستند که سگمنت‌هایی با سایز زیاد می‌خواهند.

LDT شامل کدهای برنامه، داده‌ها و پشته می‌باشد و segmentهای لوکال و مختص به برنامه را توصیف می‌کند ولی GDT، بخش‌های سیستم که شامل خود OS می‌شوند را نیز در خود نگه می‌دارد.

توصیف code segment پردازنده X86:
اگر Gbit برابر صفر باشد، limit فیلد به اندازه کل سگمنت خواهد بود که البته محدودیت ۱ مگابیتی دارد ولی اگر ۱ باشد، limit field اندازه سگمنت را مشخص می‌کند و آن ره به صفحه می‌دهد.
اگر سگمنت در حافظه باشد پنتیوم یک 32 bit base field را به آفست می‌دهد که این مقدار linear address نامیده می‌شود. Base field به هر سگمنت اجازه می‌دهد که یک نقطه شروع در 32bit linear address داشته باشند.
اگر صفحه‌بندی غیرفعال شود. Linear address به عنوان آدرس فیزیکی در نظر گرفته می‌شود و برای خواندن/نوشتن به حافظه اصلی فرستاده می‌شود.

۴)
page size به عنوان یک پارامتر توسط سیستم‌عامل انتخاب می‌شود. حتی اگر با سایز صفحه سخت‌افزار هم تفاوت داشته باشد.
بررسی فاکتور صفحه کوچک:
به طور متوسط نصف صفحه آخر خالی می‌ماند. این هدررفت internal fragmentation نامیده می‌شود. اگر n سگمنت داشته باشیم که هر کدام از آن‌ها سایز p داشته باشند، np/2 هدررفت حافظه در internal fragmentation خواهیم داشت.
اگر یک برنامه از ۳۲ فاز هر کدام با ۲ کیلوبایت حافظه مورد نیاز تشکیل شده باشد و اندازه صفحه ما ۶۴ کیلوبایت باشد، میزان حافظه بسیار زیادی هدر می‌کنیم در صورتی که هرچقدر کمتر و نزدیک به ۲ کیلوبایت باشد هدررفت حافظه کمتری خواهیم داشت.
در روی دیگر قضیه اگر اندازه صفحه کوچک داشته باشیم، جدول بزرگی برای ذخیره آن نیاز است. از طرفی دیسک‌ها هم‌زمان بسیاری صرف پیدا کردن و بردن head روی page می‌کنند که بیشتر از خواندن حجم پیج است پس اگر برای یک برنامه ۶۴ کیلوبایتی صفحه ۳۲ کیلو بایتی داشته باشیم 2x طول می‌کشد در صورتی که برای یک صفحه ۴ کیلوبایتی 16x طول می‌کشد.
اندازه کوچک صفحه هم باعث بزرگ‌تر شدن TLB می‌شود که خوب نیست.
به خاطر همین بده بستان سیستم عامل‌ها برای مثال از دو سایز صفحه مختلف استفاده می‌کنند، یکی برای کرنل و دیگری برای کاربر.
بیایید یک محاسبه کنیم:
اگر میانگین سایز هر فرایند s بایت و سایز صفحه p بایت باشد و هر رکورد صفحه e بایت داشته باشد، هر فرایند s/p صفحه و هر فرایند se/p بایت در page table می‌گیرد. هدررفت حافظه هم p/2 در قسمت internal fragmentation است که کلاً برابر se/p + p/2 می‌شود.
P اینجا متغیر است و از آن مشتق می‌گیریم و تساوی را برابر ۰ قرار می‌دهیم. در نهایت p برابر با رادیکال 2se خواهد بود که به عنوان یک نقطه optimum است.

۵)
هر سگمنت لزوماً سایز یکسانی ندارد. ارجاع به حافظه از دو بخش شماره سگمنت و offset تشکیل می‌شود.
سگمنتیشن این فواید را برای ما دارد:
۱- اگر برنامه‌نویس نداند که یک ساختمان دادا چقدر قرار است بزرگ شود، آن را در یک سگمنت تعریف می‌کنیم و سیستم‌عامل سگمنت را بزرگ‌تر یا کوچک‌تر می‌کند.
۲- به برنامه‌ها اجازه می‌دهد که به صورت مستقل کامپایل یا تغییر بکنند.
حال به حافظه مجازی می‌رسیم:
در این حالت حافظه اصلی مانند حالت صفحه بندی از قبل تکه‌تکه و فریم فریم نمی‌شود. خود برنامه‌نویس مسئول سگمنتیشن است.
Internal fragmentation نداریم چون موقعی که برنامه و کدها تمام می‌شوند، دیگر آن segment را هم می‌بندیم ولی بعد از رفت‌وامدهای متوالی external fragmentation رخ می‌دهد. مانند صفحه بندی هم یک جدول داریم که آدرس سگمنت و طول آن را نشان می‌دهد.
سیستم‌عامل باید لیست جاهایی که خالی است و یک سگمنت می‌تواند در آن قرار بگیرد را نگه دارد. برای دسترسی به یک خانه حافظه باید شماره سگمنت و offset آن خط در آن سگمنت را داشته باشیم که با تبدیل و جمع آن‌ها به هدف مورد نظر برسیم.
در حافظه مجازی زمانی‌که به یک شگمنت نیاز داشته باشیم و در حافظه نباشد آن را به حافظه دعوت می‌کنیم.

۶)
فایده این کار چیست:
یک کد یکسان که مثلاً ۱۰ کاربر با هم آن را اجرا می‌کنند را بین آن‌ها تقسیم می‌کنیم و همه از یکی استفاده می‌کنند مانند استفاده ۲۰ نفر از gedit در یک سیستم‌عامل (فقط بخش کد آن)
در این صوت هر فرایند از بخش کد به صورت مشترک استفاده خواهد کرد ولی بخش داده‌های آن و رجیسترهای آن متفاوت خواهد بود. Page table هر کاربر به بخش‌های کد یکسان که قبلاً بود شده است اشاره خواهد کرد.
برای این که بتوان از این قابلیت استفاده کرد بخش کد باید read-only باشد و سیستم‌عامل باید این محافظت را انجام دهد.
در حالت قطعه‌بندی نیز برای اشتراک گذاری هم تقریباً مشابه تکنیک بالا است و segment table به اشتراک گذاشته شده و این کار برای کدها اتفاق می‌افتد و داده‌ها حتماً در یک سگمنت جدا و برای هر کاربر به صورت جداگانه ذخیره می‌شوند.
برای محافظت از دسترسی‌های گوناگون از روش حلقه بندی استفاده می‌کنیم که حلقه‌های با شماره کمتر را حلقه و قسمت پایینی و حلقه با شماره بالاتر را حلقه خارجی می‌نامیم و در نظر می‌گیریم. هر برنامه می‌تواند به داده‌های حلقه خود و شماره بالاتر دسترسی داشته باشد و از حلقه خود یا شماره‌های پایین‌تر سرویس بگیرد.

۷)
عدد ۳۲ بیت و ۶۴ بیت به ما می‌گوید به چند خانه حافظه از طریق CPU می‌توان دسترسی داشت که ر ۳۲ بیتی ۴ گیگابایت است که می‌تواند تنها تا ۴ گیگابایت رم را ساپورت کند ولی در ۶۴ بیتی ۲ به توان ۶۴ آدرس را.
زمانی که ساپورت ما از حافظه اصلی بالاتر می‌رود و رم ما هم بیشتر از ۴ گیگابایت است مثلاً در حالت صفحه‌بندی می‌توانیم تعداد صفحات بسیار بیشتری و به اندازه رم وارد حافظه اصلی کنیم.
زمانی که حافظه اصلی ما کمتر از ۴ گیگابایت است از نظر مدیریت حافظه تفاوت زیادی نخواهیم داشت.

۸)
فرض کنیم که یک فرایند که در حافظه اصلی نیست ریستارت می‌شود. این برنامه مثلاً ۲۰ صفحه دارد که به آن نیاز داریم و در دیسک هستند. برای آوردن آن‌ها باید برای هر کدام حتماً یک page fault بخورد تا آن را وارد حافظه اصلی کنیم.
پیش‌صفحه‌بندی این مشکل را رفع می‌کند.
ایده این است که در اول کار تمامی‌صفحه‌هایی که لازم است را لود کنیم. برای پیاده‌سازی یک working set برای هر فرایند در نظر می‌گیریم. Working set یک لیست از صفحه‌هایی است که در آن فرایند با آن‌ها کار داریم. قبل از آن که فرایند را متوقف کنیم این working set را که شامل شماره صفحه‌هایی است که فرایند با آن‌ها کار دارد را ذخیره می‌کنیم و زمانی که خواستیم دوباره فرایند را ادامه دهیم قبل از آن صفحه‌های داخل working set را لود می‌کنیم.
چه زمانی مقرون به صرفه‌تر است؟
این روش زمانی به‌صرفه‌تر است که صفحه‌هایی که وارد حافظه اصلی می‌کنیم دوباره استفاده شوند.
اگر a را تعداد درصد صفحه‌های استفاده شده و صفحه‌هایی که لود شده‌اند بگیریم هرچقدر a نزدیک به ۱ باشد روش prepaging بهتر جواب داده و هرچقدر نزدیک به ۰ باشد به درد نخور بوده است.

۹)
الگوریتم:
از یک شمارنده برای هر کدام از صفحه‌ها استفاده می‌کنیم. زمانی که یک ارجاع به صفحه داریم و آن صفحه داخل فریم‌های حافظه اصلی موجود است، شمارنده‌ها را برای تمامی صفحه‌ها یکی افزایش می‌دهیم ولی همان صفحه را که به آن مراجعه شده است ۰ می‌کنیم. در این روش هرچقدر مقدار شمارنده بیشتر باشد برای شمارنده بدتر است و در معرض حذف است. زمانی که یک صفحه وارد شد مقدار شمارنده آن برابر با ۰ قرار می‌گیرد. در صورت تساوی شمارنده‌ها از FIFO استفاده می‌کنیم.
ایده این روش این است که اخیراً ارجاع داده شده‌ها را با کمتر بودن شمارنده آن‌ها مشخص می‌کند با امید به اینکه ارجاع‌های اخیر دوباره هم ارجاع داده می‌شوند.
مزیت نسبت به روش aging:
تعداد بیت‌های کمتری مورد نیاز است و به طور کلی چون تفاوت بین انتخاب ۰۰۱۱۱۱۱ با ۰۱۰۰۱۱۱۱ مشخص و بدیهی نبود با حذف تعداد بیت‌های بیشتر سعی کرده دومی را به اولی ترجیح دهد و اولی را هم به خیلی‌های دیگر ترجیح خواهد داد.

مزایای کلی:
- حافظه کمتری آشغال می‌شود.
- نوعی Aging را پیاده‌سازی می‌کند.
- پیاده‌سازی راحتی دارد.
- Overflow را در روش‌های شمارنده دار مدیریت می‌کند.

معایب:
- شاید ما در روش Aging، صفحه ۰۱۰۰۱۱۱۱ را بیرون انداختیم (ترجیح دادیم بیرون بیاندازیم) ولی این روش آن را نگه می‌دارد.

پیاده‌سازی آن هم به این صورت است که مثلاً یک شمارنده ۴ بیتی برای آن مشخص می‌کنیم و زمانی که این شمارنده به عدد ۱۵ دسیمال رسید دیگر به جای overflow، شمارنده را افزایش نمی‌دهیم و کل روش با یک شمارنده مدیریت می‌شود.