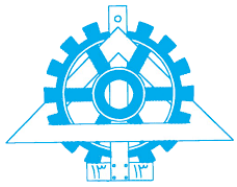
**به نام خدا**

آزمایشگاه سیستم‌عامل

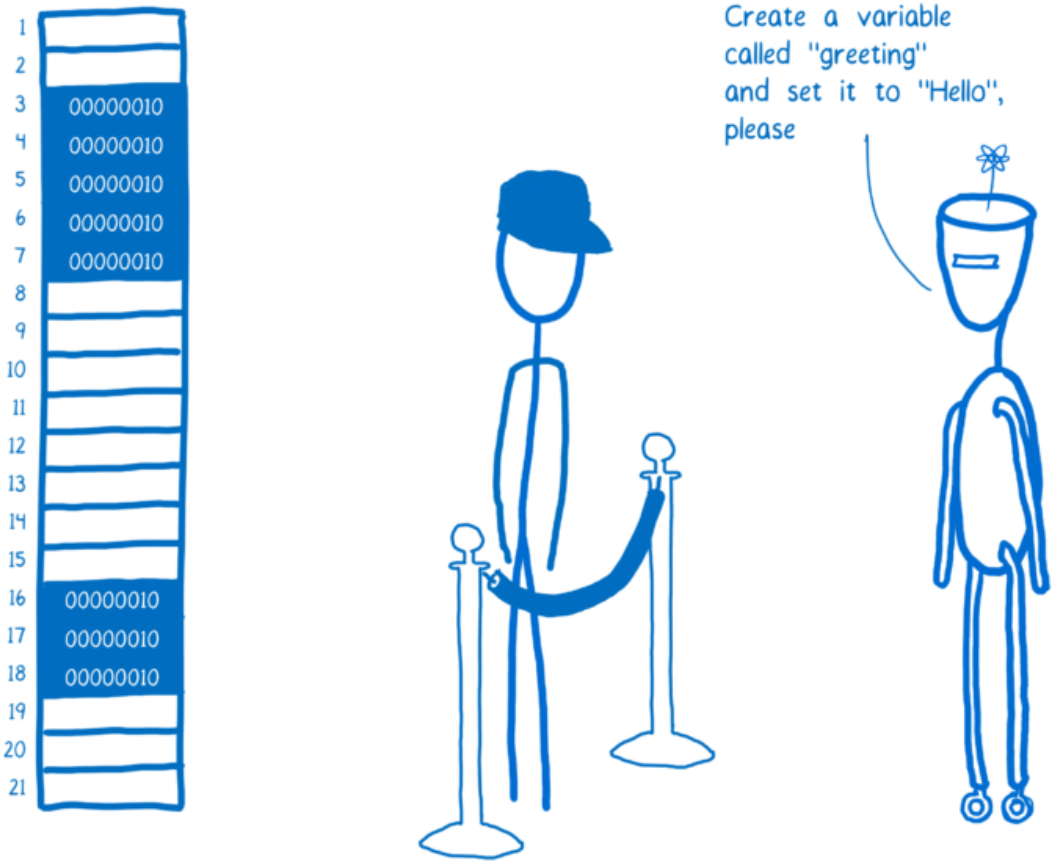


# پروژه پنجم: مدیریت حافظه

## (کپی در صورت نوشتن)

طراحان: مشکوة شریعت‌باقری ـ سید علی اخوانی

---



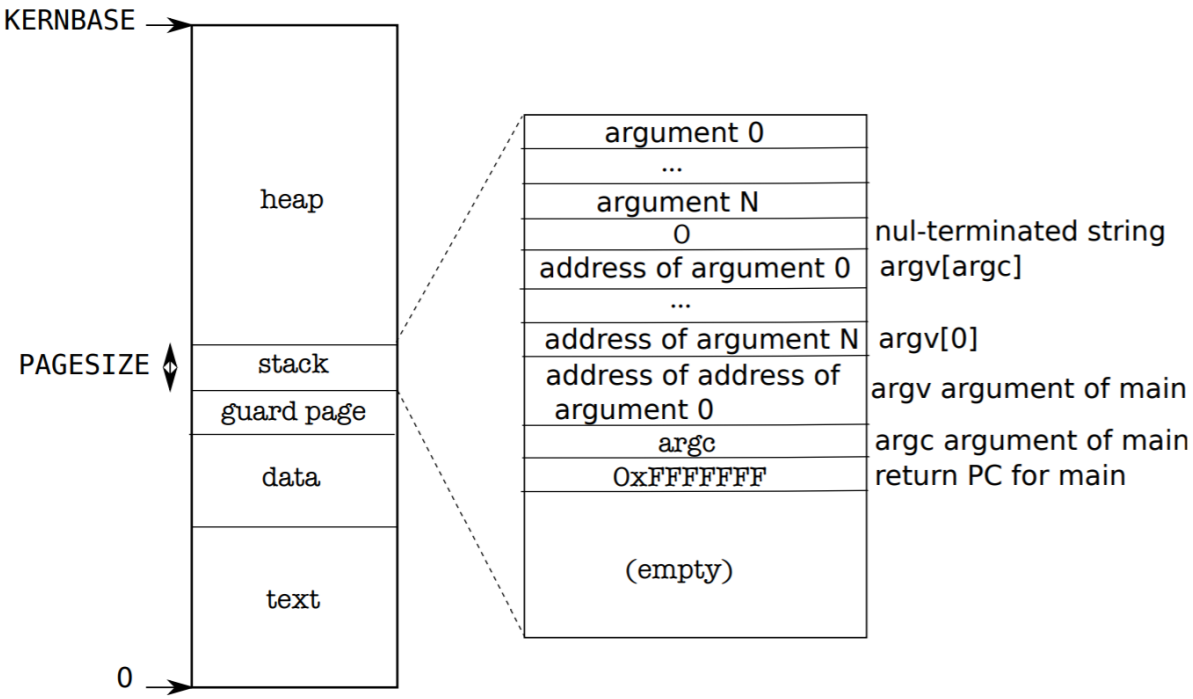

در این پروژه یکی از روش‌های کاهش سربار پردازشی و حافظه‌ای ایجاد پردازه فرزند در بسیاری از سیستم‌عامل‌ها به xv6 افزوده خواهد شد. در xv6، فراخوانی سیستمی sys_fork() وظیفه ایجاد پردازه فرزند را دارد. به این صورت که هر صفحه در حافظه برای پردازه پدر به صورت کامل برای پردازه فرزند

نیز کپی می‌شود. ولی در روش کپی در صورت نوشتن[1] (CoW) به جای کپی کردن کل صفحات پردازه پدر، از حافظه اشتراکی برای آن‌ها استفاده می‌شود. حافظه اشتراکی، حافظه‌ای است که هم پردازه پدر و هم پردازه‌های فرزند به صورت اشتراکی می‌توانند از آن استفاده کنند. با این تفاوت که عملیات کپی کردن حافظه تنها زمانی انجام می‌شود که هر کدام از پردازه‌های پدر یا فرزند بخواهند در هر صفحه حافظه تغییری ایجاد کنند. در ادامه ابتدا مدیریت حافظه به طور کلی در xv6 معرفی شده و در نهایت آزمایش شرح داده می‌شود.

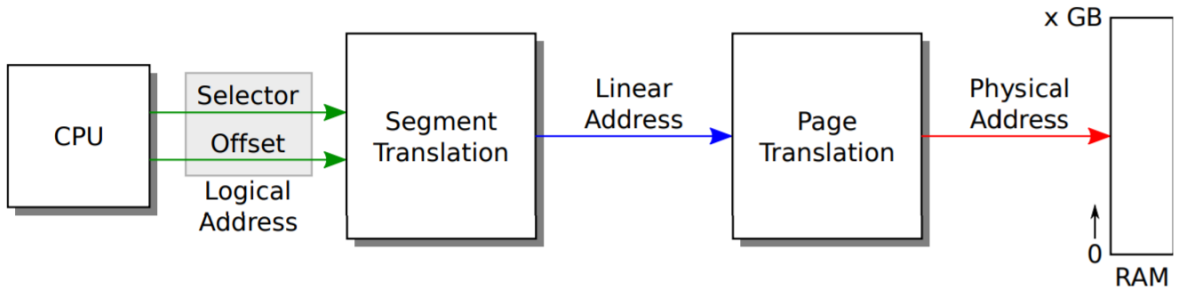
## مقدمه

یک برنامه، حین اجرا تعامل‌های متعددی با حافظه دارد. دسترسی به متغیرهای ذخیره شده و فراخوانی توابع موجود در نقاط مختلف حافظه مواردی از این ارتباط‌ها می‌باشد. معمولاً کد منبع دارای آدرس نبوده و از نمادها برای ارجاع به متغیرها و توابع استفاده می‌شود. این نمادها توسط کامپایلر و پیونددهنده[2] به آدرس تبدیل خواهد شد. حافظه یک برنامه سطح کاربر شامل بخش‌های مختلفی مانند کد، پشته[3] و هیپ[4] است. این ساختار برای یک برنامه در xv6 در شکل زیر نشان داده شده است.

[1] Copy-on-Write
[2] Linker
[3] Stack
[4] Heap




همان‌طور که در آزمایش یک ذکر شد، در مد محافظت‌شده[۵] در معماری x86 هیچ کدی (اعم از کد هسته یا کد برنامه سطح کاربر) دسترسی مستقیم به حافظه فیزیکی[۶] نداشته و تمامی آدرس‌های برنامه از خطی[۷] به مجازی[۸] و سپس به فیزیکی تبدیل می‌شوند. این نگاشت در شکل زیر نشان داده شده است.




به همین منظور، هر برنامه یک جدول اختصاصی موسوم به جدول صفحه[۹] داشته که در حین فرایند تعویض‌متن[۱۰] بارگذاری شده و تمامی دسترسی‌های حافظه (اعم از دسترسی به هسته یا سطح کاربر) توسط آن برنامه توسط این جدول مدیریت می‌شود.

---

5 Protected Mode
6 Physical Memory
7 Linear
8 Virtual
9 Page Table
10 Context Switch

به علت عدم استفاده صریح از قطعه‌بندی در بسیاری از سیستم‌عامل‌های مبتنی بر این معماری، می‌توان فرض کرد برنامه‌ها از صفحه‌بندی[۱۱] و لذا آدرس مجازی استفاده می‌کنند. علت استفاده از این روش مدیریت حافظه در درس تشریح شده است. به طور مختصر می‌توان سه علت عمده را برشمرد:

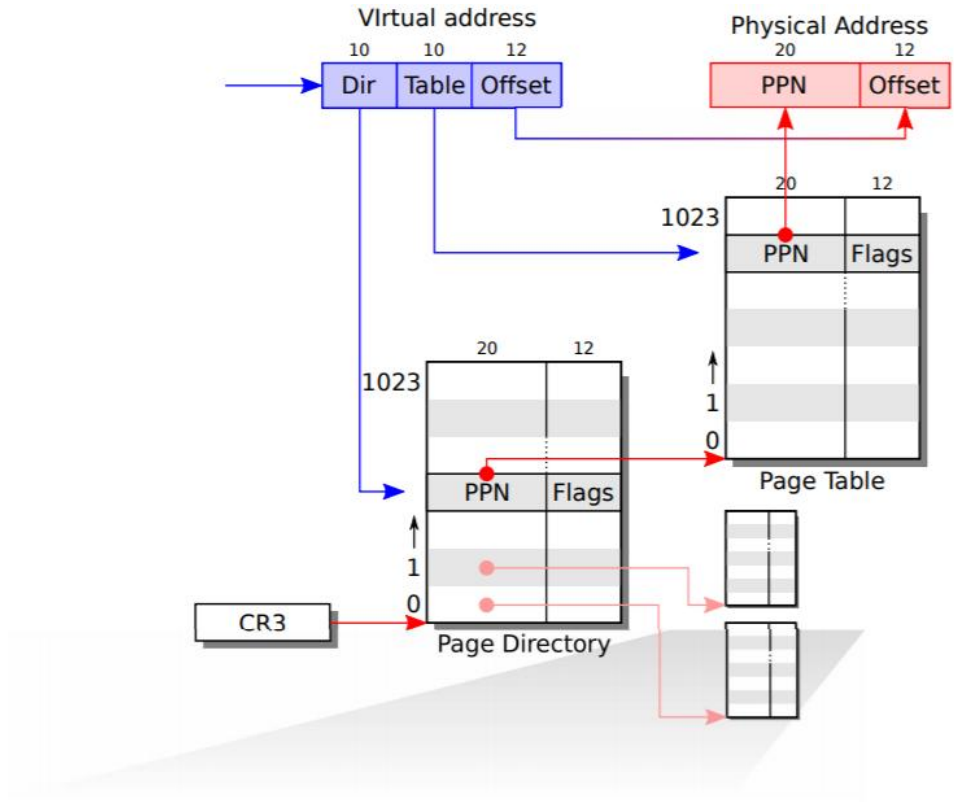
**۱) ایزوله‌سازی پردازه‌ها از یکدیگر و هسته از پردازه‌ها:** با اجرای پردازه‌ها در فضاهای آدرس[۱۲] مجزا، امکان دسترسی یک برنامه مخرب به حافظه برنامه‌های دیگر وجود ندارد. ضمن این که با اختصاص بخش مجزا و ممتازی از هر فضای آدرس به هسته امکان دسترسی محافظت‌نشده پردازه‌ها به هسته سلب می‌گردد.

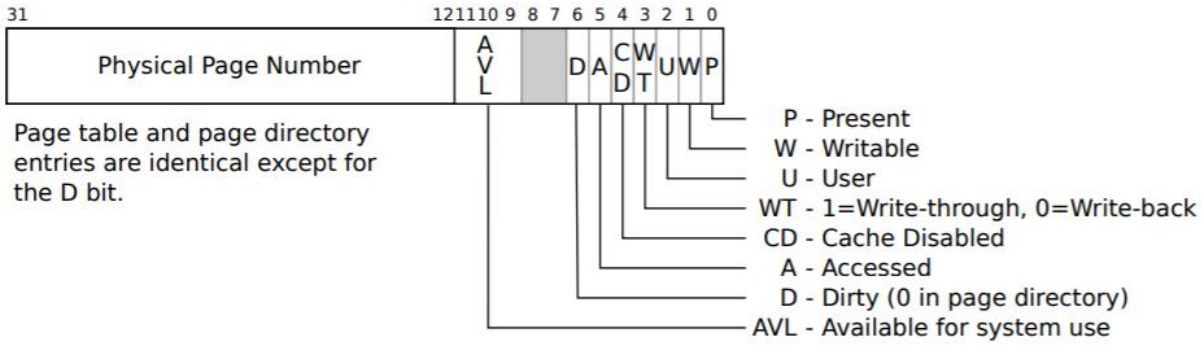
**۲) ساده‌سازی ABI سیستم‌عامل:** هر پردازه می‌تواند از یک فضای آدرس پیوسته (از آدرس مجازی صفر تا چهار گیگابیت در معماری x86) به طور اختصاصی استفاده نماید. به عنوان مثال کد یک برنامه در سیستم‌عامل لینوکس در معماری x86 همواره (در صورت عدم استفاده از تصادفی‌سازی چینش فضای آدرس[۱۳] (ASLR)) از آدرس 0x08048000 آغاز شده و نیاز به تغییر در آدرس‌های برنامه‌ها متناسب با وضعیت جاری تخصیص حافظه فیزیکی نمی‌باشد.

**۳) استفاده از جابه‌جایی حافظه:** با علامت‌گذاری برخی از صفحه‌های کم‌استفاده (در جدول صفحه) و انتقال آن‌ها به دیسک، حافظه فیزیکی بیشتری در دسترس خواهد بود. به این عمل جابه‌جایی حافظه[۱۴] اطلاق می‌شود.

ساختار جدول صفحه در معماری x86 (در حالت بدون گسترش آدرس فیزیکی[۱۵] (PAE) و گسترش اندازه صفحه[۱۶] (PSE)) در شکل زیر نشان داده شده است.

11 Paging
12 Address Spaces
13 Address Space Layout Randomization
14 Memory Swapping
15 Physical Address Extension
16 Page Size Extension




هر آدرس مجازی توسط اطلاعات این جدول به آدرس فیزیکی تبدیل می‌شود. این فرایند، سخت‌افزاری بوده و سیستم‌عامل به طور غیرمستقیم با پر کردن جدول، نگاشت را صورت می‌دهد. جدول صفحه دارای سلسله‌مراتب دوسطحی بوده که به ترتیب Page Directory و Page Table نام دارند. هدف از ساختار سلسله‌مراتبی کاهش مصرف حافظه است.

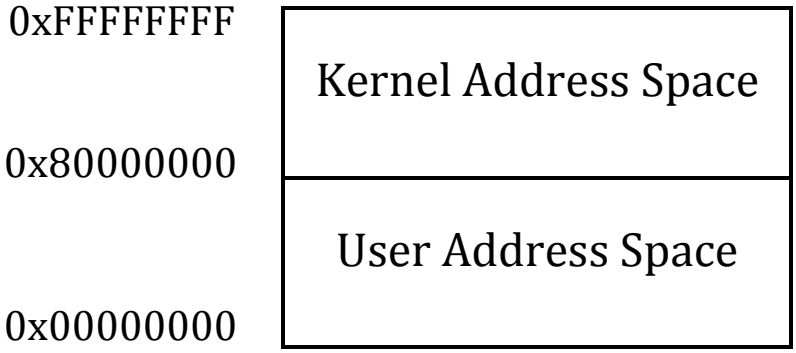
۱) چرا ساختار سلسله‌مراتبی منجر به کاهش مصرف حافظه می‌گردد؟

۲) محتوای هر بیت یک مدخل (۳۲ بیتی) در هر سطح چیست؟ چه تفاوتی میان آن‌ها وجود دارد؟

# مدیریت حافظه در xv6

## ساختار فضای آدرس در xv6

در xv6 نیز مد اصلی اجرای پردازنده، مد حفاظت‌شده و سازوکار اصلی مدیریت حافظه صفحه‌بندی است. به این ترتیب نیاز خواهد بود که پیش از اجرای هر کد، جدول صفحه آن در دسترس پردازنده قرار گیرد. کدهای اجرایی در xv6 شامل کد پردازه‌ها (کد سطح کاربر) و ریسه هسته متناظر با آن‌ها و کدی است که در آزمایش یک، کد مدیریت‌کننده نام‌گذاری شد.[۱۷] آدرس‌های کد پردازه‌ها و ریسه هسته آن‌ها توسط جدول صفحه‌ای که اشاره‌گر به ابتدای Page Directory آن در فیلد pgdir از ساختار proc (خط ۲۳۳۹) قرار دارد، نگاشت داده می‌شود. نمای کلی ساختار حافظه مجازی متناظر با جدول صفحه این دسته در شکل زیر نشان داده شده است.




دوگیگابایت پایین جدول صفحه مربوط به اجزای مختلف حافظه سطح کاربر پردازه است. دو گیگابایت بالای جدول صفحه مربوط به اجزای ریسه هسته پردازه بوده و در تمامی پردازه‌ها یکسان است. آدرس تمامی متغیرهایی که در هسته تخصیص داده می‌شوند در این بازه قرار می‌گیرد. جدول صفحه کد مدیریت‌کننده هسته، دو گیگابایت پایینی را نداشته (نگاشتی در این بازه ندارد) و دو گیگابایت بالای آن

[۱۷] بحث مربوط به پس از اتمام فرایند بوت است. به عنوان مثال، در بخشی از بوت، از صفحات چهار مگابایتی استفاده شد که از آن صرفنظر شده است.

دقیقاً شبیه به پردازه‌ها خواهد بود. زیرا این کد، همواره در هسته اجرا شده و پس از بوت غالباً در اوقات بی‌کاری سیستم اجرا می‌شود.

## کد مربوط به ایجاد فضاهای آدرس در xv6

فضای آدرس کد مدیریت‌کننده هسته در حین بوت، در تابع main() ایجاد می‌شود. به این ترتیب که تابع kvmalloc() فراخوانی شده (خط ۱۲۲۰) و به دنبال آن تابع setupkvm() متغیر سراسری kpgdir را مقداردهی می‌نماید (خط ۱۸۴۲). به طور کلی هر زمان نیاز به مقداردهی ساختار فضای آدرس هسته باشد، از setupkvm() استفاده خواهد شد. با بررسی تابع setupkvm() (خط ۱۸۱۸) می‌توان دریافت که در این تابع، ساختار فضای آدرس هسته بر اساس محتوای آرایه kmap (خط ۱۸۰۹) چیده می‌شود.

۲) تابع kalloc() چه نوع حافظه‌ای تخصیص می‌دهد؟ (فیزیکی یا مجازی)

۳) تابع mappages() چه کاربردی دارد؟

فضای آدرس مجازی نخستین برنامه سطح کاربر (initcode) نیز در تابع main() ایجاد می‌گردد. به طور دقیق‌تر تابع userinit() (خط ۱۲۳۵) فراخوانی شده و توسط آن ابتدا نیمه هسته فضای آدرس با اجرای تابع setupkvm() (خط ۲۵۲۸) مقداردهی خواهد شد. نیمه سطح کاربر نیز توسط تابع inituvm() ایجاد شده تا کد برنامه نگاشت داده شود. فضای آدرس باقی پردازه‌ها در ادامه اجرای سیستم توسط توابع fork() یا exec() مقداردهی می‌شود. به این ترتیب که هنگام ایجاد پردازه فرزند توسط fork() با فراخوانی تابع copyuvm() (خط ۲۵۹۲) فضای آدرس نیمه هسته ایجاد شده (خط ۲۰۴۲) و سپس فضای آدرس نیمه کاربر از والد کپی می‌شود. این کپی با کمک تابع walkpgdir() (خط ۲۰۴۵) صورت می‌پذیرد.

۴) راجع به تابع walkpgdir() توضیح دهید. این تابع چه عمل سخت‌افزاری را شبیه‌سازی می‌کند؟

وظیفه تابع exec() اجرای یک برنامه جدید در ساختار بلوک کنترلی پردازه[۱۸] (PCB) یک پردازه موجود است. معمولاً پس از ایجاد فرزند توسط fork() فراخوانده شده و کد، داده‌های ایستا، پشته و هیپ برنامه جدید را در فضای آدرس فرزند ایجاد می‌نماید. بدین ترتیب با اعمال تغییراتی در فضای آدرس موجود، امکان اجرای یک برنامه جدید فراهم می‌شود. روش متداول Shell در سیستم‌عامل‌های مبتنی بر یونیکس از جمله xv6 برای اجرای برنامه‌های جدید مبتنی بر exec() است. Shell پس از دریافت ورودی و فراخوانی fork1() تابع runcmd() را برای اجرای دستور ورودی، فراخوانی می‌کند (خط ۸۷۲۴). این تابع نیز در نهایت تابع exec() را فراخوانی می‌کند (خط ۸۶۲۶). چنان‌چه در آزمایش یک مشاهده شد، خود Shell نیز در حین بوت با فراخوانی فراخوانی سیستمی sys_exec() (خط ۸۴۱۴) و به دنبال آن exec() ایجاد شده و فضای آدرسش به جای فضای آدرس نخستین پردازه (initcode) چیده می‌شود. در پیاده‌سازی exec() مشابه قبل setupkvm() فراخوانی شده (خط ۶۶۳۷) تا فضای آدرس هسته تعیین گردد. سپس با فراخوانی allocuvm() فضای مورد نیاز برای کد و داده‌های برنامه جدید (خط ۶۶۵۱) و صفحه محافظ و پشته (خط ۶۶۶۵) تخصیص داده می‌شود. دقت شود تا این مرحله تنها تخصیص صفحه صورت گرفته و باید این فضاها در ادامه توسط توابع مناسب با داده‌های مورد نظر پر شود (به ترتیب خطوط ۶۶۵۵ و ۶۶۸۶).

۵) داده‌ساختار kmem در فایل kalloc.c چه کاربردی دارد؟ xv6 چگونه صفحات آزاد را ردیابی و نگهداری می‌کند؟

[18] Process Control Block

## شرح آزمایش

در انتهای این پروژه شما باید قابلیت کپی در صورت نوشتن را به تابع fork() عادی موجود در xv6 اضافه نمایید. توجه داشته باشید که این فرایند مستلزم تغییر در چندین فایل از xv6 و افزودن توابع جدیدی به آن‌ها است. مراحل اضافه کردن این ویژگی به xv6 در بخش‌های ذیل شرح داده شده است.

### بررسی تعداد صفحات خالی

یک فراخوانی سیستمی با نام getNumFreePages() اضافه کنید که تعداد صفحات خالی موجود در سیستم را محاسبه نموده و نمایش دهد. از این تابع در سایر بخش‌ها و همچنین برای اشکال‌زدایی استفاده کنید تا متوجه شوید که چه زمانی یک صفحه مورد استفاده قرار می‌گیرد.

### نگه‌داری تعداد ارجاع به صفحه

در xv6 اجازه‌ی به‌اشتراک‌گذاری قاب صفحه فیزیکی[۱۹] را نداریم. جهت افزودن این قابلیت باید به نحوی مقدار تعداد ارجاع به هر صفحه نگه‌داری شود. در ابتدا باید با نحوه تخصیص صفحات فیزیکی آشنا شوید. برای این کار فایل kalloc.c را مورد بررسی قرار دهید.

در این پروژه، صفحه اشتراکی در ابتدا از نوع فقط‌خواندنی خواهد بود. نگه‌داری تعداد ارجاعات به آن صفحه باعث می‌شود که بفهمید یک صفحه فقط‌خواندنی مربوط به فرآیند CoW بوده یا مربوط به یک صفحه فقط‌خواندنی غیراشتراکی برای بخش‌های دیگر سیستم‌عامل است.

برای نگه‌داری تعداد پردازه‌هایی که به یک صفحه دسترسی دارند باید تعداد ارجاعات را به struct مربوط به صفحات اضافه کنید. این مقدار هنگامی که در هنگام تخصیص صفحه، یک خواهد بود. هم‌چنین تابعی کمکی برای افزایش و کاهش تعداد ارجاعات بنویسید. این کار مستلزم استفاده مناسب از قفل‌ها و

[19] Physical Page Frame

دستورالعمل‌های اتمیک است. تعداد ارجاعات به یک صفحه با ایجاد هر فرزند، یک واحد افزوده می‌شود. هم‌چنین با اتمام اشاره یک پردازه به صفحه، یک واحد از این مقدار کاسته خواهد شد.

### مدیریت نقص صفحه

فایل trap.c را مطالعه کنید. در این فایل شماره تله‌ها مانند T_SYSCALL (که در پروژه دو معرفی شد) وجود دارد. در تابع trap() یک کنترل‌کننده برای T_PGFLT ایجاد کنید. وقتی که یک پردازه والد یا یک فرزند اقدام به نوشتن در یک صفحه فقط‌خواندنی نماید، یک نقص صفحه[۲۰] رخ خواهد داد. در این بخش باید یک trap handler اضافه کنید تا این نقص صفحه را مدیریت کند. یعنی باید قابلیت بررسی کردن نوشتن در یک صفحه CoW را نیز به آن اضافه کنید. هم‌چنین در نوشتن تست‌های خود، یک تست برای این بخش بنویسید که بخواهد به یک خانه نامعتبر از حافظه دسترسی پیدا کند و مطمئن شوید که هندلر شما برای نابودی آن پردازه صدا زده می‌شود. به طور خاص باید در بخشی از حافظه کد برنامه بنویسید.

### پیاده‌سازی CoW Fork

در این‌جا به سراغ تابع fork() می‌رویم. این تابع در فایل proc.c تعریف شده است. همان‌طور که ذکر شد در فرآیند fork از تابع copyuvm() بهره برده می‌شود که بخش مهمی از عملیات این تابع را انجام می‌دهد. این تابع در فایل vm.c قرار دارد. در این پروژه باید یک تابع با نام cowuvm() پیاده‌سازی شود که مشابه copyuvm() است. اما برای پیاده‌سازی قابلیت کپی هنگام نوشتن ایجاد می‌شود و با copyuvm() تفاوت‌هایی دارد. پیش از این در fork() تمام صفحه‌ها کپی می‌شد. اما در CoW Fork باید در حین Fork دو عمل صورت گیرد:

۱- صفحه مورد نظر فقط‌خواندنی شده و صفحه پردازه والد برای فرزند کپی نشود.

[20] Page Fault

۲- با نوشتن بر روی حافظه فقط‌خواندنی نقص صفحه رخ خواهد داد. زیرا هنگامی که یکی از پردازه‌های والد یا فرزند قصد نوشتن در حافظه اشتراکی دارند، باید از آن صفحات کپی مجزایی گرفته شده تا هر پردازه کپی اختصاصی خود را داشته باشد (همان عملی که تا پیش از این در copyuvm() صورت می‌گرفت).

۶) کدام بخش‌های فضای آدرس سطح کاربر نیاز به کپی مجزا ندارند؟ برای سطح هسته نیز توضیح دهید.

## آزمون

در این قسمت شما باید بتوانید تمام آزمون‌ها را با موفقیت پشت سر بگذارید. در این بخش حداقل سه آزمون برای fork بنویسید (به جز آزمونی که برای نقص صفحه‌ها نوشتید). به آزمون‌هایی که هوشمندانه باشد و موارد خاصی از fork را اجرا کند، نمره اضافی تعلق می‌گیرد.

## سایر نکات

- کدهای شما باید به زبان C بوده و و نام‌گذاری فایل‌ها و توابع مانند الگوهای مذکور باشد.
- کیفیت کد و Code Style شما در نمره‌دهی موثر است. بنابراین توابع خود را به شکل مناسب اضافه کنید. هم‌چنین تغییرات خود را به نحوی انجام دهید که کد نهایی همچنان کد تمیزی باشد.
- در نهایت کدهای خود را در یک فایل zip با نام‌گذاری مشابه الگوی مقابل آپلود کنید. (stdNum1_stdNum2_stdNum3.zip)
- همه اعضای گروه باید به پروژه بارگذاری شده توسط گروه خود مسلط بوده و لزوماً نمره افراد یک گروه با یکدیگر برابر نخواهد بود.
- در صورت مشاهده هرگونه مشابهت بین کدها یا گزارش دو یا چند گروه، نمره صفر به همه آن‌ها تعلق می‌گیرد.
- پاسخ تمامی سؤالات را در کوتاه‌ترین اندازه ممکن در گزارش خود بیاورید.
- فصول ۱ و ۲ کتاب xv6 می‌تواند مفید باشد.
- هر گونه سؤال در مورد پروژه را فقط از طریق فروم درس مطرح نمایید.

موفق باشید