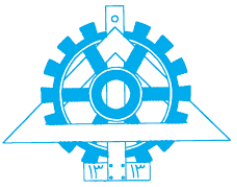
به نام خدا

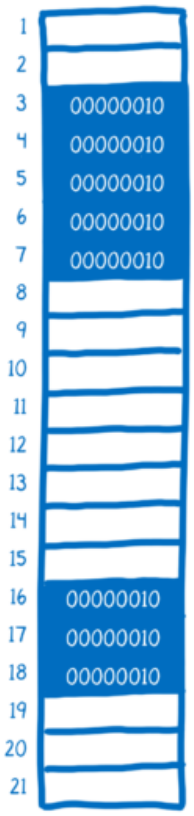
آزمایشگاه سیستم‌عامل

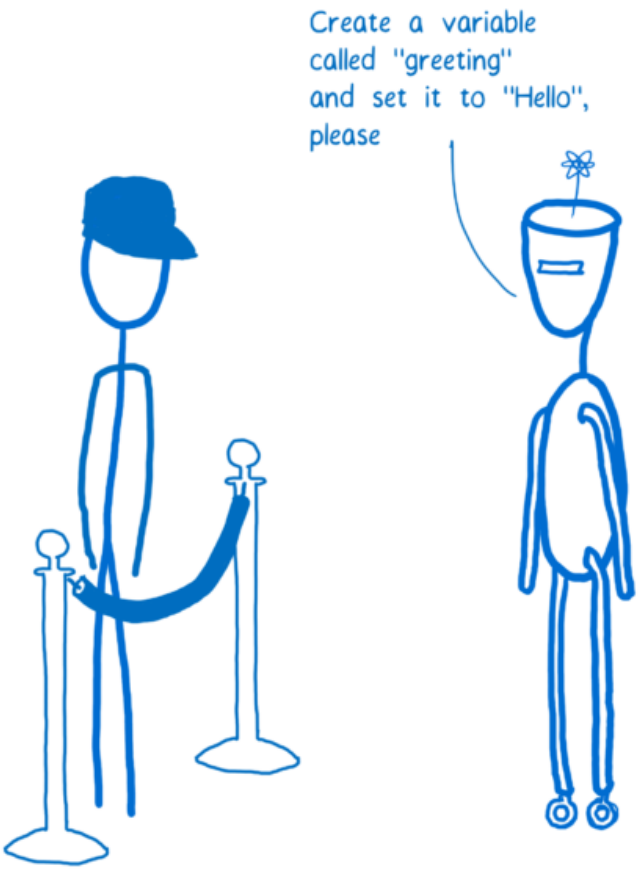
پروژه پنجم: مدیریت حافظه

(پیاده‌سازی mmap در xv6)

طراحان: شایان حمیدی، آرین حدادی



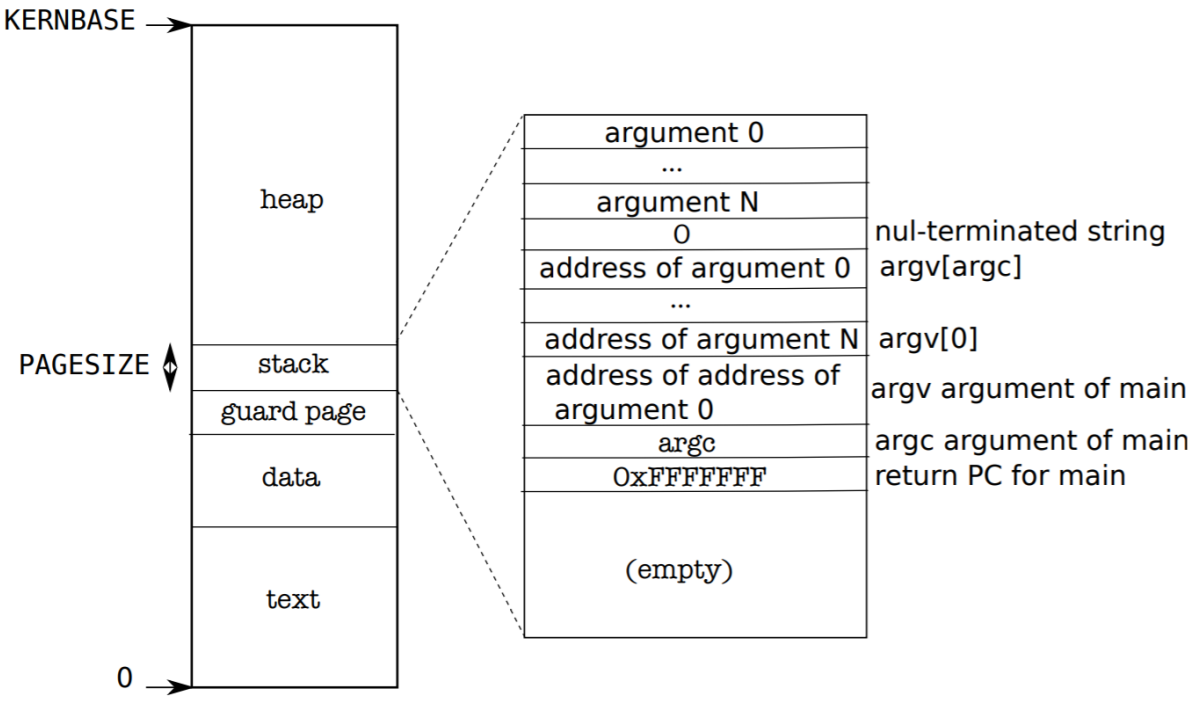




در این پروژه شیوه مدیریت حافظه در سیستم‌عامل xv6 بررسی شده و قابلیت‌هایی به آن افزوده خواهد شد. در ادامه ابتدا مدیریت حافظه به طور کلی در xv6 معرفی شده و در نهایت صورت آزمایش شرح داده خواهد شد.

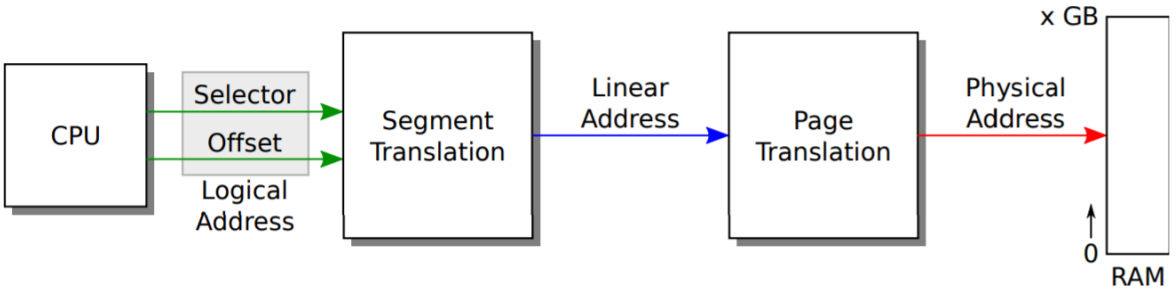
مقدمه

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[1] به آدرس تبدیل خواهد شد. حافظه یک برنامه سطح کاربر شامل بخش‌های مختلفی مانند کد، پشته[2] و هیپ[3] است. این ساختار برای یک برنامه در xv6 در شکل زیر نشان داده شده است.



همان‌طور که در آزمایش یک ذکر شد، در مد محافظت‌شده[4] در معماری x86 هیچ کدی (اعم از کد هسته یا کد برنامه سطح کاربر) دسترسی مستقیم به حافظه فیزیکی[5] نداشته و تمامی آدرس‌های برنامه از خطی[6] به مجازی[7] و سپس به فیزیکی تبدیل می‌شوند. این نگاشت در شکل زیر نشان داده شده است.

[1] Linker
[2] Stack
[3] Heap
[4] Protected Mode
[5] Physical Memory
[6] Linear
[7] Virtual



به همین منظور، هر برنامه یک جدول اختصاصی موسوم به جدول صفحه[8] داشته که در حین فرایند تعویض‌متن[9] بارگذاری شده و تمامی دسترسی‌های حافظه (اعم از دسترسی به هسته یا سطح کاربر) توسط آن برنامه توسط این جدول مدیریت می‌شود.

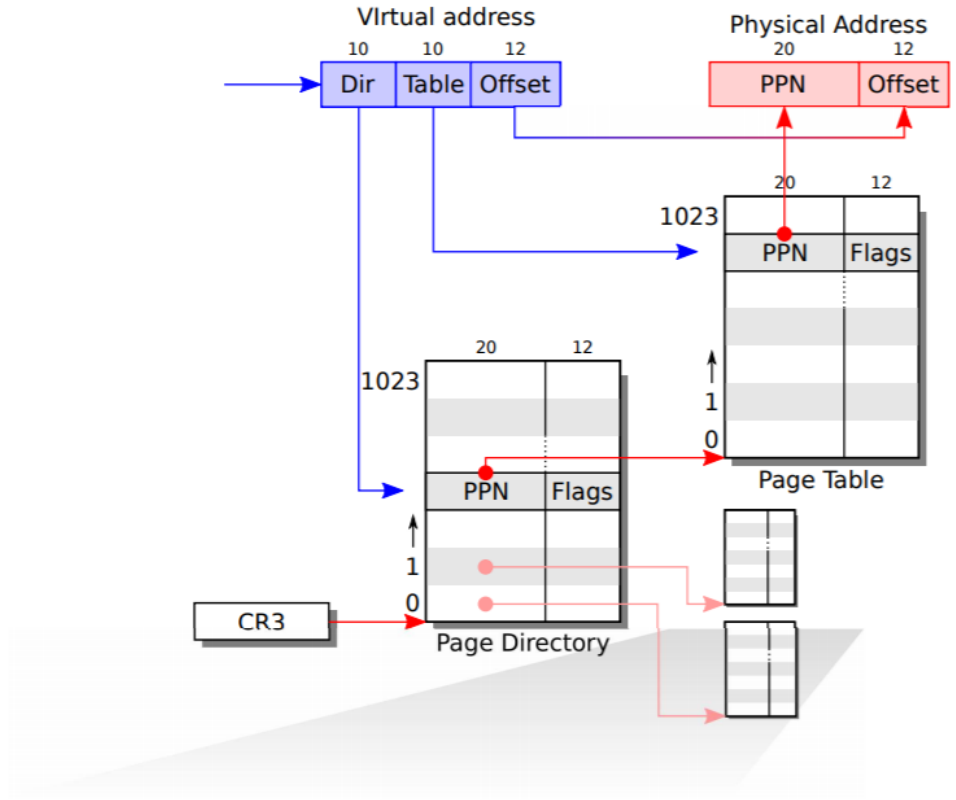
به علت عدم استفاده صریح از قطعه‌بندی در بسیاری از سیستم‌عامل‌های مبتنی بر این معماری، می‌توان فرض کرد برنامه‌ها از صفحه‌بندی[10] و لذا آدرس مجازی استفاده می‌کنند. علت استفاده از این روش مدیریت حافظه در درس تشریح شده است. به طور مختصر می‌توان سه علت عمده را برشمرد:

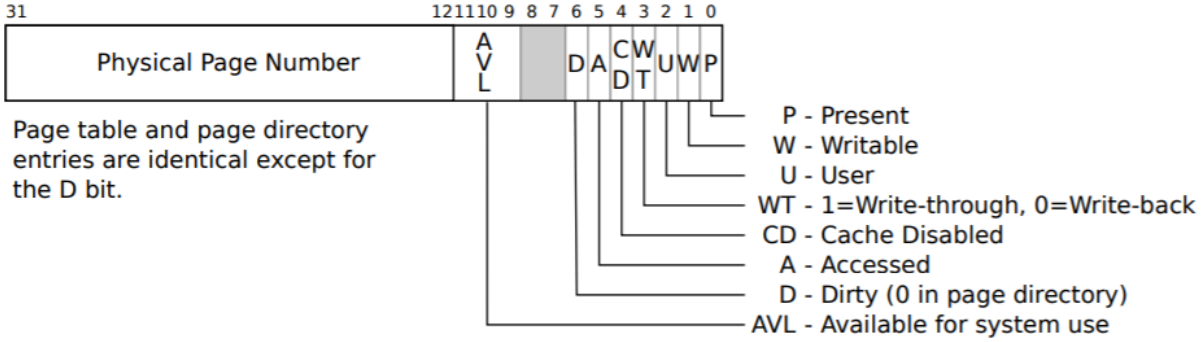
۱) **ایزوله‌سازی پردازه‌ها از یکدیگر و هسته از پردازه‌ها:** با اجرای پردازه‌ها در فضاهای آدرس[11] مجزا، امکان دسترسی یک برنامه مخرب به حافظه برنامه‌های دیگر وجود ندارد. ضمن این که با اختصاص بخش مجزا و ممتازی از هر فضای آدرس به هسته امکان دسترسی محافظت‌نشده پردازه‌ها به هسته سلب می‌گردد.

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[12] (ASLR)) از آدرس 0x08048000 آغاز شده و نیاز به تغییر در آدرس‌های برنامه‌ها متناسب با وضعیت جاری تخصیص حافظه فیزیکی نمی‌باشد.

[8] Page Table
[9] Context Switch
[10] Paging
[11] Address Spaces
[12] Address Space Layout Randomization

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[13] اطلاق می‌شود. ساختار جدول صفحه در معماری x86 (در حالت بدون گسترش آدرس فیزیکی[14] (PAE) و گسترش اندازه صفحه[15] (PSE)) در شکل زیر نشان داده شده است.





[13] Memory Swapping
[14] Physical Address Extension
[15] 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 دو سطحی 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) روشی برای تخمین فرکانس دسترسی به صفحه‌های حافظه به کمک بیت‌های مدخل جدول صفحه[16] ارائه دهید. (راهنمایی: می‌توان از شیوه بهره‌گیری از بیت Accessed در لینوکس الگوبرداری کرد.)

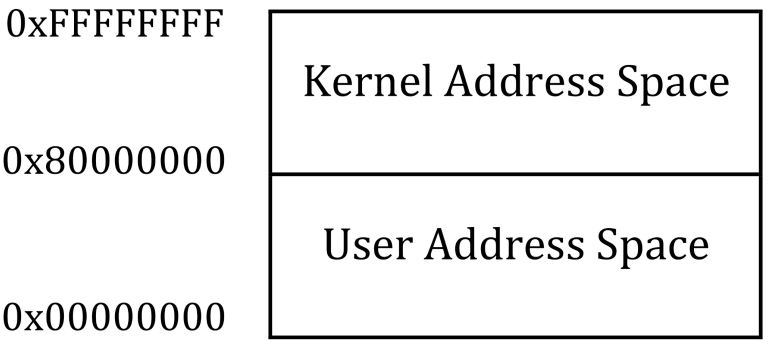
مدیریت حافظه در xv6

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.[17] 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.

[16] Page Table Entry

[17] 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 صرف نظر شده است.



دو گیگابایت 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 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.

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

۴) 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.

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

وظیفه تابع exec() اجرای یک برنامه جدید در ساختار بلوک کنترل پردازه[18] (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

[18] Process Control Block

صفحه صورت گرفته و باید این فضاها در ادامه توسط توابع مناسب با داده‌های مورد نظر پر شود (به ترتیب خطوط ۶۶۵۵ و ۶۶۸۶).

شرح آزمایش

در سیستم‌عامل‌های سازگار با پازیکس[19]، یک فراخوانی سیستمی وجود دارد تا بتوان دستگاه‌های ورودی/خروجی[20] (I/O) یا فایل‌ها را در حافظه مجازی برنامه‌ها نگاشت تا واسطه دسترسی به آن‌ها همانند خواندن و نوشتن در خانه‌های حافظه باشد. این فراخوانی سیستمی mmap نام دارد. در این تمرین قصد داریم فایل‌های نگاشته‌شده به حافظه[21] را با استفاده از mmap پیاده‌سازی کنیم. خواندن و نوشتن در فایل‌ها با استفاده از mmap امکانات زیادی به ما می‌دهد. (هرچند پیاده‌سازی ما در این تمرین تمام این امکانات را به ما نمی‌دهد.)

۶) دو نقص نگاشت فایل در حافظه نسبت به خواندن عادی فایل‌ها را بیان کنید.

روش مورد استفاده جهت نگاشت فایل‌ها در حافظه به این شکل است که با فراخوانی mmap، به اندازه مورد نیاز، حافظه مجازی اختصاص داده می‌شود اما داده‌ای از دیسک خوانده نشده و حافظه فیزیکی تخصیص نمی‌یابد؛ بلکه تنها درخواست نگاشت در PCB پردازه ذخیره می‌شود. سپس هنگام دسترسی به حافظه توسط پردازه، تله خطای صفحه رخ می‌دهد و در این هنگام محتوای صفحه مورد نظر از فایل خوانده شده، یک فریم حافظه فیزیکی برای آن اختصاص داده شده و به حافظه مجازی نگاشته می‌شود. به این نوع پر کردن فضای آدرس مجازی صفحه‌بندی حین تقاضا[22] (یا بارگذاری تنبل[23]) می‌گویند که امکان خواندن فایل‌های بزرگ‌تر از حافظه فیزیکی را ممکن کرده و برای استفاده‌هایی که تنها نقاط محدودی از فایل مورد دسترسی واقع می‌شوند نیز مفید است. به این ترتیب، خواندن و نوشتن محتویات فایل توسط پردازه به شکل دسترسی به یک بلوک حافظه قابل انجام است.

[19] POSIX
[20] Input/Output
[21] Memory-Mapped Files
[22] Demand Paging
[23] Lazy Loading

برای انجام این تمرین، ابتدا یک فراخوانی سیستمی برای یافتن تعداد صفحه‌های فیزیکی در دسترس سیستم خواهیم نوشت و سپس فراخوانی سیستمی mmap را پیاده‌سازی خواهیم کرد. پیش از آغاز انجام این تمرین، توصیه می‌شود فصل دوم کتاب xv6 را مطالعه نمایید.

۱. فراخوانی سیستمی get_free_pages_count

وظیفه این فراخوانی سیستمی محاسبه و برگرداندن تعداد صفحه‌های آزاد موجود از حافظه فیزیکی است. از این فراخوانی سیستمی برای عیب‌یابی برنامه خود نیز می‌توانید استفاده کنید.

```
int get_free_pages_count();
```

نکات:

- ساختمان داده مربوط به فریم فیزیکی (kmem) در فایل kalloc.c آمده است. لیست فریم‌های خالی با استفاده از یک لیست پیوندی پیاده شده است. برای درک بهتر، می‌توانید توابع kalloc و kfree را در این فایل مطالعه کنید.
- برای نوشتن این فراخوانی سیستمی، از آن‌جا که ساختمان داده‌ها و توابع مربوط به حافظه فیزیکی در فایل kalloc.c وجود دارند، توصیه می‌شود بدنه مربوط به آن را به عنوان تابعی کمکی در این فایل بنویسید و در تعریف فراخوانی سیستمی خود این تابع را صدا بزنید.

۲. فراخوانی سیستمی mmap

این فراخوانی سیستمی در لینوکس دارای امضای زیر است:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, int offset);
```

قرار است این فراخوانی سیستمی را با امکانات محدودی در xv6 پیاده سازی کنیم. مقدار خروجی این فراخوانی سیستمی آدرسی به آغاز حافظه مجازی نگاشته شده است. توضیحات و فرضیات موجود برای هر کدام از آرگومان‌ها نیز در ادامه آمده است:

- addr: اگر برابر با صفر (NULL) باشد، هسته تصمیم می‌گیرد که نگاشت را در چه آدرسی انجام دهد. در غیر این صورت، هسته سعی می‌کند نگاشت را در این آدرس یا در نزدیکی این آدرس انجام دهد. **در این تمرین، می‌توانید فرض کنید که addr همیشه NULL است و در پیاده سازی سمت هسته، آدرس انتخاب می‌شود. فضای آدرس پردازه‌های سمت کاربر از صفر تا 0x80000000 است و ما از وسط این بازه (آدرس 0x40000000 به بعد) برای نگاشت فایل‌ها در حافظه استفاده خواهیم کرد.**
- length: نشان‌دهنده تعداد بایت‌هایی از فایل است که باید نگاشت شود. توجه کنید که مقدار آن می‌تواند مضربی از PGSIZE نباشد. در این صورت، نیاز است تمام صفحه برای این نگاشت مورد استفاده قرار گیرد و نگاشت بعدی از آغاز صفحه بعدی انجام شود. برای تبدیل یک آدرس به آدرس ابتدای صفحه بعدی، می‌توانید از ماکروی PGROUNDUP استفاده کنید.
- prot: نشان‌دهنده بیت‌های حفاظتی از حافظه نگاشته شده است. **برای این تمرین، فرض کنید این آرگومان تنها مقادیر PROT_READ را می‌تواند بگیرد.** برای درک بهتر چگونگی تنظیم بیت‌های حفاظتی، به موارد استفاده تابع mappages در فایل vm.c دقت کنید.
- flags: مشخص می‌کند که تغییرات روی حافظه نگاشته شده چگونه انتشار می‌یابند. برای مثال، سه نوع زیر از مقادیر پرکاربرد آن هستند:
  - MAP_SHARED: استفاده از این پرچم یعنی تغییرات روی خانه‌های حافظه مرتبط با حافظه mmap شده باید در نهایت روی فایل مدنظر قابل مشاهده باشند.

- MAP_PRIVATE: نشانگر این است که تغییراتی که یک پردازه روی حافظه mmap شده می‌دهد، تنها مربوط به خودش است و روی فایل اعمال نمی‌شود. **در این تمرین، کافی است تنها این پرچم را پیاده سازی کنید.**
- MAP_ANONYMOUS: در صورتی از این پرچم استفاده می‌شود که از mmap فقط برای اختصاص حافظه (مانند malloc) استفاده شود. در این حالت، آرگومان fd نادیده گرفته می‌شود.

- fd: توصیف‌گر پرونده مربوط به این فایل است. در نتیجه نیاز است پیش از فراخوانی mmap، فایل مدنظر با استفاده از فراخوانی سیستمی open باز شده باشد.
- offset: نشان‌دهنده فاصله از آغاز فایل است تا از آن نقطه از فایل، نگاشت صورت گیرد. **در این تمرین، همواره مقدار offset برابر صفر است و نگاشت از آغاز فایل صورت می‌گیرد.**

نکات:

- همانطور که در ابتدا اشاره شد، هنگام فراخوانی mmap، در عمل فایلی از دیسک خوانده نشده و حافظه فیزیکی‌ای نیز تخصیص داده نمی‌شود و این کار به صورت تنبل انجام می‌گیرد. در نتیجه، آدرس بازگردانده شده از mmap، آدرسی بدون نگاشت است که دسترسی به آن منجر به تله خطای صفحه می‌شود. اختصاص حافظه فیزیکی و خواندن از فایل باید در هندلر مربوط به این تله (در فایل trap.c) انجام شود. تله خطای صفحه، تله شماره ۱۴ است و با نماد T_PGFLT مشخص شده است.
- برای پیدا کردن آدرسی که منجر به تله شده است، می‌توانید از تابع rcr2 استفاده کنید.
- توجه کنید که ممکن است تله خطای صفحه به علت موارد دیگری به جز دسترسی به حافظه mmap شده نیز رخ دهد؛ پس روی آدرس منجر به خطا اعتبارسنجی‌های لازم را انجام دهید.

- برای اینکه حالت MAP_PRIVATE را ایجاد کنید باید از روش کپی‌حین‌نوشتن[24] (COW) استفاده کنید. به این صورت که وقتی که می‌خواهیم روی حافظه نگاشت شده تغییری ایجاد کنیم، داده‌های صفحه قدیمی در یک صفحه دیگر کپی شوند و تغییرات عملا روی آن صفحه جدید اعمال شود. از این بعد نوشتن‌ها به صورت محلی (در این صفحه جدید) انجام می‌شوند.
- هنگامی که یک پردازه نابود شده و یا به طور کلی کار آن به پایان می‌رسد، تمامی فضای آدرس آن آزاد می‌شود (فراخوانی deallocuvm() در تابع freevm()).
- برای پیدا کردن مدخل جدول صفحه متناظر با یک آدرس مجازی، از تابع walkpgdir() استفاده کنید.
- برای اختصاص صفحه فیزیکی و نگاشت آن به جدول صفحه پردازه، از پیاده‌سازی تابع allocuvm() کمک بگیرید.
- در استاندارد پازیکس آمده است که توصیف‌گر پرونده مربوط به فایل می‌تواند پس از فراخوانی mmap بسته شود و نباید نگاشت را با مشکل روبرو کند. به همین دلیل، هنگام فراخوانی mmap مقدار ref اشاره‌گر struct file مربوطه را یک واحد اضافه کنید که مانع از بین رفتن شیء مربوط به آن شوید (در نتیجه نیاز است هنگام پایان یافتن کار پردازه نیز، این مقدار را یک واحد کاهش دهید). برای این منظور توابع filedup() و fileclose() را بررسی کنید.
- دقت کنید که هنگام Fork، پردازه فرزند نیز باید نگاشت‌های پردازه والد را داشته باشد. در این هنگام، فراموش نکنید که مقدار ref مربوط به اشاره‌گر struct file مربوطه را نیز یک واحد اضافه کنید.
- برای خواندن بخشی از فایل می‌توانید از تابع fileread() در فایل file.c استفاده کنید. توجه کنید که برای فایل‌های موجود در فایل‌سیستم (که در این تمرین با آن‌ها کار می‌کنیم)، مقدار

[24] Copy-on-Write

type در struct file برابر با FD_INODE است. برای نوشتن در فایل نیز تابع filewrite() را بررسی کنید.

- ممکن است mmap در یک پردازه بیش از یک بار فراخوانی شود؛ پس برای این که آدرس‌ها و نگاشت‌های قبلی خراب نشوند، نیاز است اطلاعات مورد نیاز را در struct proc نگه‌داری کنید. نگاشت‌ها را از آدرس 0x40000000 آغاز کنید و صفحه به صفحه جلو بروید (می‌توانید فرض کنید فضای هیپ پردازه به این خانه از حافظه نمی‌رسد).
- فرض کنید mmap بیش از یکبار با یک توصیف‌گر پرونده فراخوانی نمی‌شود.
- فرض کنید هر پردازه حداکثر می‌تواند هشت فایل نگاشته شده در حافظه داشته باشد.
- در صورت بروز هرگونه خطا و عدم دسترسی کافی به صفحات حافظه، مقدار صفر (NULL) را به عنوان آدرس خروجی بازگردانید.

بخش امتیازی:

می‌دانیم پس از عملیات Fork، فضای حافظه پردازه والد برای فرزندان کپی می‌شود، لذا تغییرات محلی والد برای فرزندان هم اعمال می‌شود. فلسفه استفاده از پرچم MAP_PRIVATE در قسمت قبل آن است که هر پردازه برای خود یک کپی اختصاصی داشته باشد و این کپی با هیچ پردازه دیگری حتی پردازه‌های فرزند به اشتراک گذاشته نشود. لذا برای این بخش پیاده‌سازی خود را به گونه‌ای تغییر دهید که پس از عملیات Fork تغییرات محلی پردازه والد برای فرزندان اعمال نشود.در کلام دیگر، تغییرات قبل این عملیات برای فرزندان قابل دسترسی است ولی از این به بعد پردازه والد و فرزند از هم مستقل هستند. **برای تست کد خود، برنامه ای بنویسید که پردازه والد یک فایل را نگاشت داده باشد و سپس تعدادی فرزند تولید کند. سپس فرزندان و والد شروع به تغییر داده‌ها کنند ولی این تغییرات برای پردازه‌های فرزند و والد محلی است و هیچ تاثیری روی هم ندارند.** این تست را هم به صورت برنامه کاربر در یک فایل C کنار پروژه خود ارائه دهید.

سایر نکات:

- کدهای شما باید به زبان C بوده و و نام‌گذاری توابع مانند الگوهای مذکور باشد.
- جهت آزمون صحت عملکرد پیاده‌سازی، برای هر بخش یک برنامه سمت کاربر بنویسید هنگام تحویل پروژه، صحت پیاده‌سازی شما مقابل برنامه‌های سمت کاربر دیگری نیز سنجیده خواهد شد. برای تست mmap، چند پردازه را از طریق Fork بسازید. سپس این پردازه‌ها فایل را نگاشت دهند و حافظه را بخوانند یا بنویسند. در نهایت هر پردازه باید تغییرات خود را (که نوشته است) به طور محلی ببیند اما در فایل نباید تغییری ایجاد شده باشد.

به طور دقیق‌تر، یک فایل شامل اعداد تصادفی از ۱ تا ۱۰۰ ایجاد کنید. سپس سه پردازه ایجاد نمایید. یکی فقط فایل را بخواند. دومی، اعداد را به توان دو رسانده و در همان حافظه رونویسی کرده و سومی، آن‌ها را به توان سه رسانده و رونویسی کند. در نهایت هر یک کل آرایه را چاپ نماید.

- 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 شباهت بین کدها یا گزارش دو یا چند گروه، نمره صفر به همه آن‌ها تعلق می‌گیرد.
- پاسخ تمامی سوالات را در کوتاه‌ترین اندازه ممکن در گزارش خود بیاورید.
- فصول یک و دو از کتاب xv6 می‌تواند مفید باشد.
- هرگونه سوال در مورد پروژه را فقط از طریق فروم درس مطرح نمایید.