



به نام خدا



پروژه پنجم آزمایشگاه سیستم عامل

(مدیریت حافظه در xv6)

طراحان: محمدطاها فخاریان، پوریا تاجمحرابی

در این پروژه شیوه مدیریت حافظه در سیستم عامل 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.

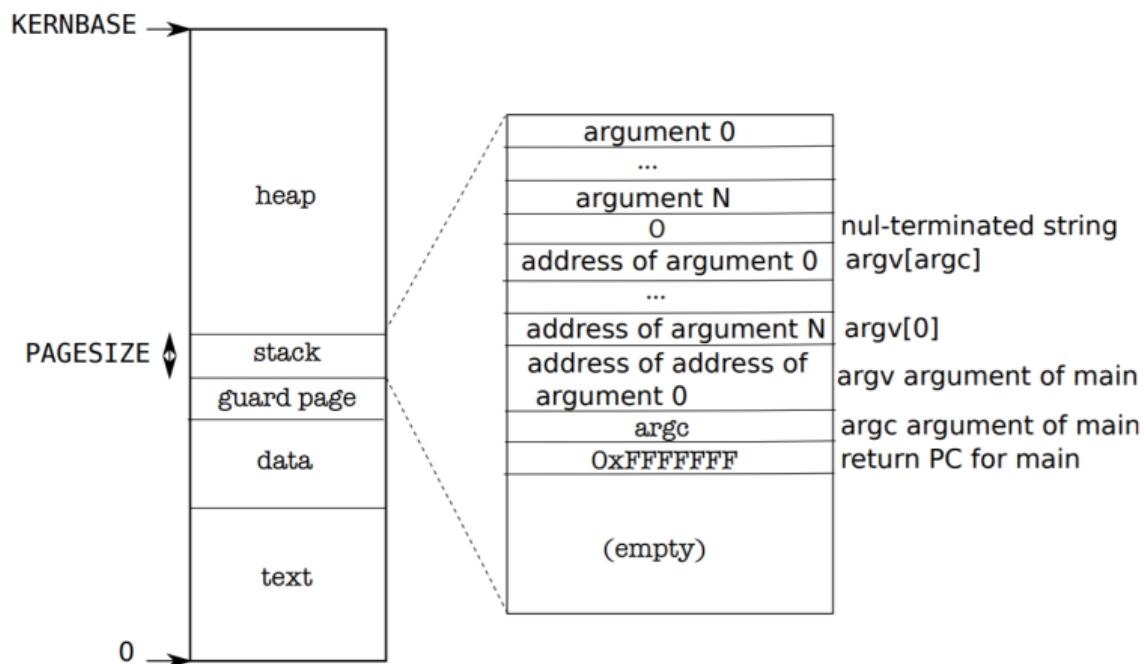
مقدمه

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 پیونددهنده¹ به آدرس تبدیل خواهد شد. حافظه یک برنامه سطح کاربر شامل بخش‌های مختلفی مانند کد، پشته² و هیپ³ است. این ساختار برای یک برنامه در xv6 در شکل زیر نشان داده شده است:

¹ Linker

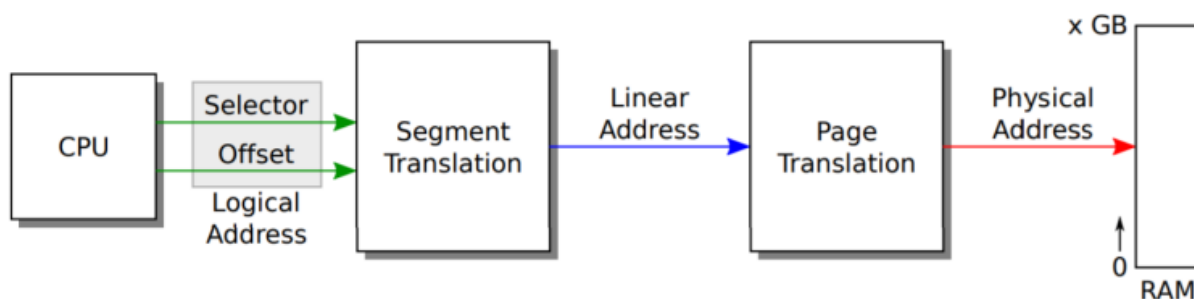
² Stack

³ Heap



(۱) راجع به مفهوم ناحیه مجازی^۴ (VMA) در لینوکس به طور مختصر توضیح داده و آن را با xv6 مقایسه کنید.

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:



^۴ Virtual Memory Area

^۵ Protected Mode

^۶ Physical Memory

^۷ Linear

^۸ Virtual

به همین منظور، هر برنامه یک جدول اختصاصی موسوم به جدول صفحه⁹ داشته که در حین فرآیند تعویض متن¹⁰ بارگذاری شده و تمامی دسترسی‌های حافظه (اعم از دسترسی به هسته یا سطح کاربر) توسط آن برنامه توسط این جدول مدیریت می‌شود.

به علیت عدم استفاده صریح از قطعه‌بندی در بسیاری از سیستم‌عامل‌های مبتنی بر این معماری، می‌توان فرض کرد برنامه‌ها از صفحه‌بندی¹¹ 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 پردازنده‌ها از یکدیگر و هسته از پردازنده‌ها: با اجرای پردازنده‌ها در فضاهای آدرس¹² 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 پردازنده‌ها به هسته سلب می‌گردد.

(۲) ساده‌سازی ABI سیستم‌عامل: هر پردازنده می‌تواند از یک فضای آدرس پیوسته (از آدرس مجازی صفر تا چهار گیگابایت 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 آدرس¹³ (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 صفحه‌ها کم استفاده (در جدول صفحه) و انتقال آن‌ها به دیسک، حافظه بیشتری در دسترس خواهد بود. به این عمل جابه‌جایی حافظه¹⁴ اطلاق می‌شود.

⁹ Page Table

¹⁰ Context Switch

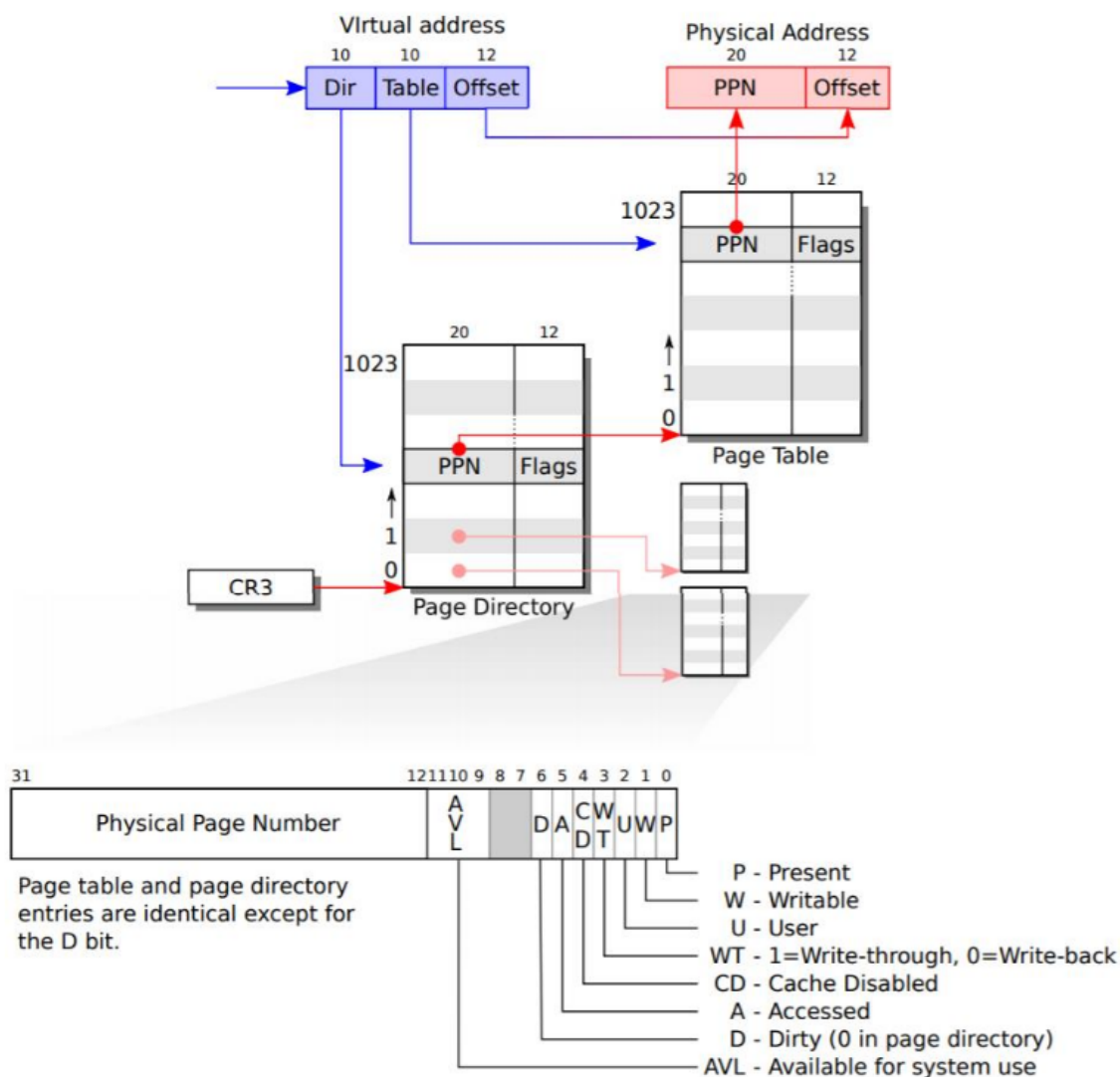
¹¹ Paging

¹² Address Spaces

¹³ Address Space Layout Randomization

¹⁴ Memory Swapping

ساختار جدول صفحه در معماری x86 (در حالت بدون گسترش آدرس فیزیکی¹⁵ (PAE) و گسترش اندازه صفحه¹⁶ (PSE) در شکل زیر نشان داده شده است.



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

(۲) چرا ساختار سلسله‌مراتبی منجر به کاهش مصرف حافظه می‌گردد؟

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

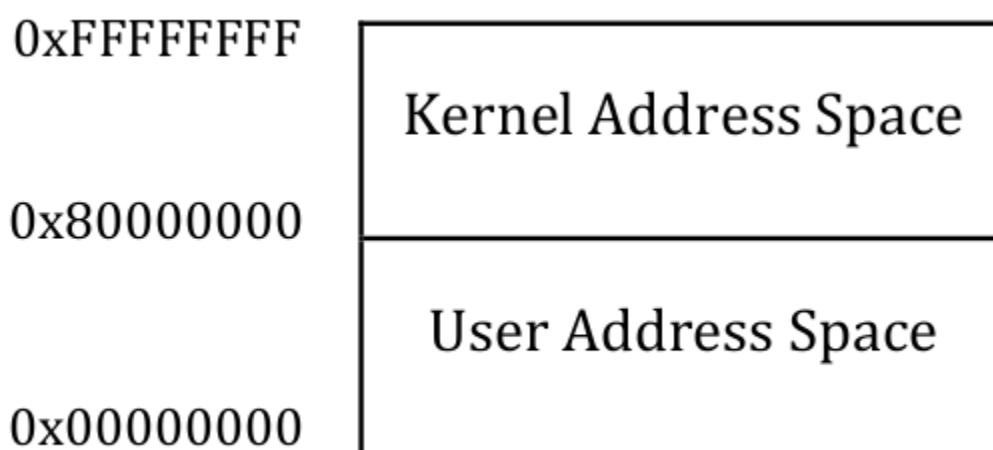
¹⁵ Physical Address Extension

¹⁶ Page Size Extension

مدیریت حافظه در xv6

ساختار فضای آدرس در xv6

در xv6 نیز مد اصلی اجرای پردازنده، مد محافظت شده و ساز و کار اصلی مدیریت حافظه صفحه بندی 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 پردازنده ها (کد سطح کاربر) و ریشه هسته متناظر با آن ها و کدی است که در آزمایش یک، کد مدیریت کننده نام گذاری شد.¹⁷ آدرس های کد پردازنده ها و ریشه هسته آن ها توسط جدول صفحه ای که اشاره گر به ابتدای Page Directory آن در فیلد pgdir از ساختار proc (خط ۲۳۳۹) قرار دارد، نگاشت داده می شود. نمای کلی ساختار حافظه مجازی متناظر با جدول این دسته در شکل زیر نشان داده شده است:



دو گیگابایت پایین جدول صفحه مربوط به اجزای مختلف حافظه سطح کاربر پردازنده است. دو گیگابایت بالای جدول صفحه مربوط به اجزای ریشه هسته پردازنده بوده و در تمامی پردازنده ها یکسان است. آدرس تمامی متغیرهایی که در هسته تخصیص داده می شوند در این بازه قرار می گیرد. جدول صفحه کد مدیریت کننده هسته، دو گیگابایت پایینی را نداشته (نگاشتی در این بازه ندارد) و دو گیگابایت بالای آن دقیقاً شبیه به پردازنده ها خواهد بود. زیرا این کد، همواره در هسته اجرا شده و پس از بوت غالباً، در اوقات بیکاری سیستم اجرا می شود.

¹⁷ بحث مربوط به پس از اتمام فرآیند بوت است. به عنوان مثال، در بخشی از بوت، از صفحات چهار مگابایتی استفاده شد که از آن صرف نظر شده است.

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 `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.

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

۵) 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 `setupkvm()` (خط ۲۵۲۸) مقداردهی خواهد شد. نیمه سطح کاربر نیز توسط تابع `inituvm()` ایجاد شده تا کد برنامه نگاشت داده شود. فضای آدرس باقی‌پرده‌ها در ادامه اجرای سیستم توسط توابع `fork()` یا `exec()` مقداردهی می‌شوند. به این ترتیب که هنگام ایجاد پرده 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.

۷) 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 پرده^{۱۸} یک پرده 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 سیستم‌های 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 سیستمی `sys_exec()` (خط ۸۴۱۴) و به دنبال آن `exec()` ایجاد شده و فضای آدرسش به جای فضای آدرس نخستین پرده (`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

¹⁸ Process Control Block

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).

۸) توابع `allocuvm` و `mappages` که در ارتباط با حافظه‌ی مجازی هستند را توضیح دهید.

۹) شیوه‌ی بارگذاری^{۱۹} برنامه در حافظه توسط فراخوانی سیستمی `exec` را شرح دهید.

شرح پروژه

در این پروژه، می‌خواهیم قابلیت فضای حافظه اشتراکی^{۲۰} را به هسته `xv6` اضافه کنیم. با استفاده از این قابلیت، دو پردازنده این امکان را خواهند داشت که به یک حافظه مشترک دسترسی داشته باشند. برای پیاده‌سازی این قابلیت، نیاز است که دو فراخوانی سیستمی به `xv6` اضافه شوند که در ادامه به توضیح آن‌ها می‌پردازیم. قبل از آن، نیاز است که برای مدیریت فضای حافظه اشتراکی، یک جدول به این منظور به `xv6` اضافه کنید. این استراکت احتمالاً باید شامل این موارد باشد:

- تعدادی صفحه اشتراکی که هر صفحه شامل این ویژگی‌هاست:
 - آیدی صفحه
 - پوینتری به فریم فیزیکی شروع صفحه
 - تعداد رفرنس‌هایی که به آن صفحه دسترسی دارند
- یک قفل برای مدیریت دسترسی به جدول؛ برای وقتی که چند پردازنده بخواهند به جدول دسترسی داشته باشند

open_sharedmem

در این فراخوانی سیستمی، آیدی یک بخش از حافظه گرفته شده و چک می‌شود که برای این آیدی، قبلاً حافظه گرفته شده است یا نه. اگر گرفته نشده باشد، نیاز است که اختصاص پیدا کرده و تبدیل حافظه مجازی به فیزیکی انجام شود و این اطلاعات به جدول حافظه اشتراکی اضافه شود. توجه داشته باشید که نیاز است که حتماً به درستی از قفل موجود در جدول استفاده کنید تا داده‌ها و حافظه‌ها به درستی در جدول اضافه شده و تغییر داده شوند. مدیریت تعداد رفرنس‌ها هم باید به درستی انجام شود. برای تبدیل حافظه‌ها از مجازی به فیزیکی می‌توانید از `mappages` استفاده کنید. پوینتر به شروع حافظه اشتراکی به عنوان خروجی این فراخوانی سیستمی بازگردانده می‌شود.

¹⁹ Load

²⁰ Memory Sharing

close_sharedmem

در این فراخوانی سیستمی، آیدی یک حافظه اشتراکی گرفته می‌شود و چک می‌شود که در جدول وجود دارد یا خیر؛ در صورت وجود، یکی از تعداد رفرنس‌ها کم شده و در صورتی که تعداد رفرنس‌ها به صفر برسد، نیاز است که آن حافظه اشتراکی از جدول حذف شود. توجه داشته باشید که با توجه به اینکه هنوز در جدول تبدیل حافظه مجازی به فیزیکی، آن حافظه وجود داشته و معتبر هست، نیازی به آزادسازی حافظه اشتراکی نخواهد بود.

برنامه آزمون

بعد از این که این دو فراخوانی سیستمی را به xv6 اضافه کردید، زمان آزمون قابلیت حافظه اشتراکی فرا رسیده است! یک برنامه سطح کاربر بنویسید که پردازش اول یک حافظه اشتراکی بگیرد و مقدار آن فریم را صفر کند. سپس تعدادی پردازش فرزند ایجاد کند. هر پردازش فرزند به آن حافظه اشتراکی دسترسی پیدا کرده و مقدار حافظه را یکی زیاد کنند. تنها در صورتی مقدار حافظه‌ها به درستی تغییر می‌کنند که به درستی از قفل استفاده شود (می‌توانید یک بار از قفل استفاده نکنید و خروجی دوبار اجرا را باهم مقایسه کنید). پس از اضافه کردن این مقدار، حافظه اشتراکی خود را بسته و پس از اتمام کار تمامی پردازش‌های فرزند، پردازش اصلی مقدار حافظه را گزارش می‌کند. در صورتی که به درستی قابلیت را پیاده‌سازی کرده باشید، این مقدار باید برابر با تعداد پردازش‌های فرزند باشد. خروجی برنامه آزمون و نحوه پیاده‌سازی فراخوانی‌های سیستمی را حتماً در گزارش خود ذکر کنید.