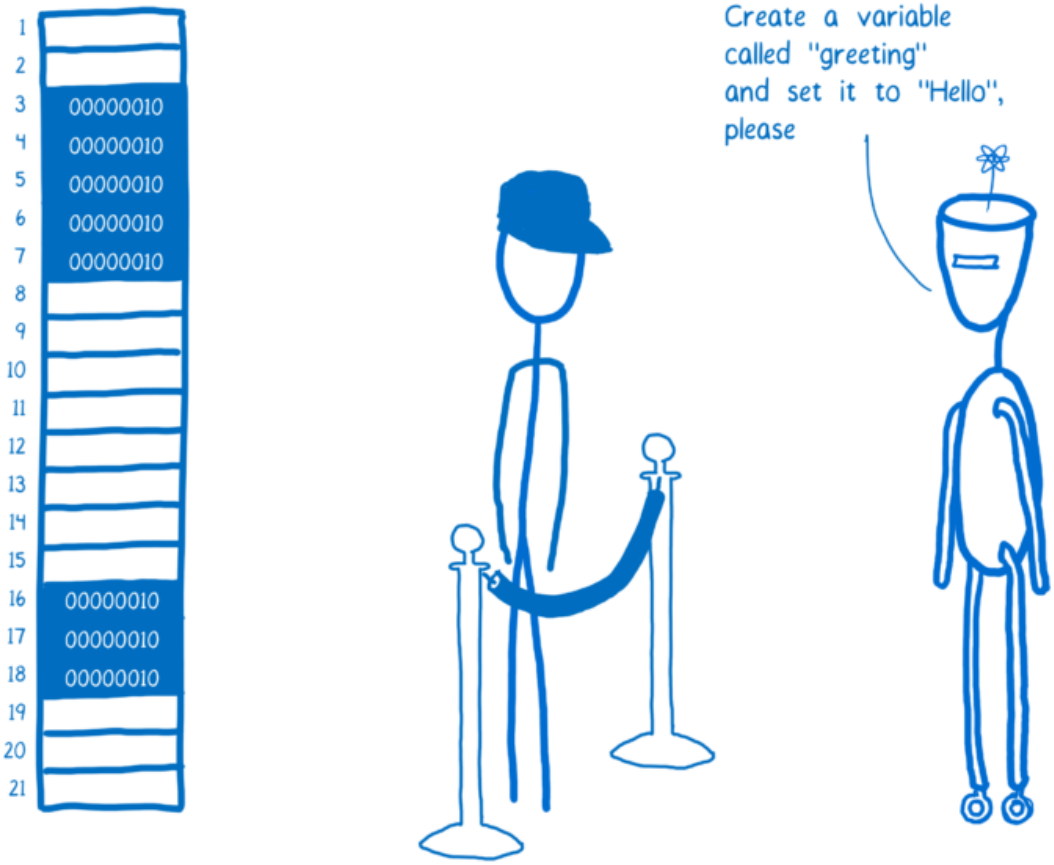
به نام خدا

آزمایشگاه سیستم‌عامل

پروژه پنجم: مدیریت حافظه

(آشنایی با حافظه مجازی در xv6)

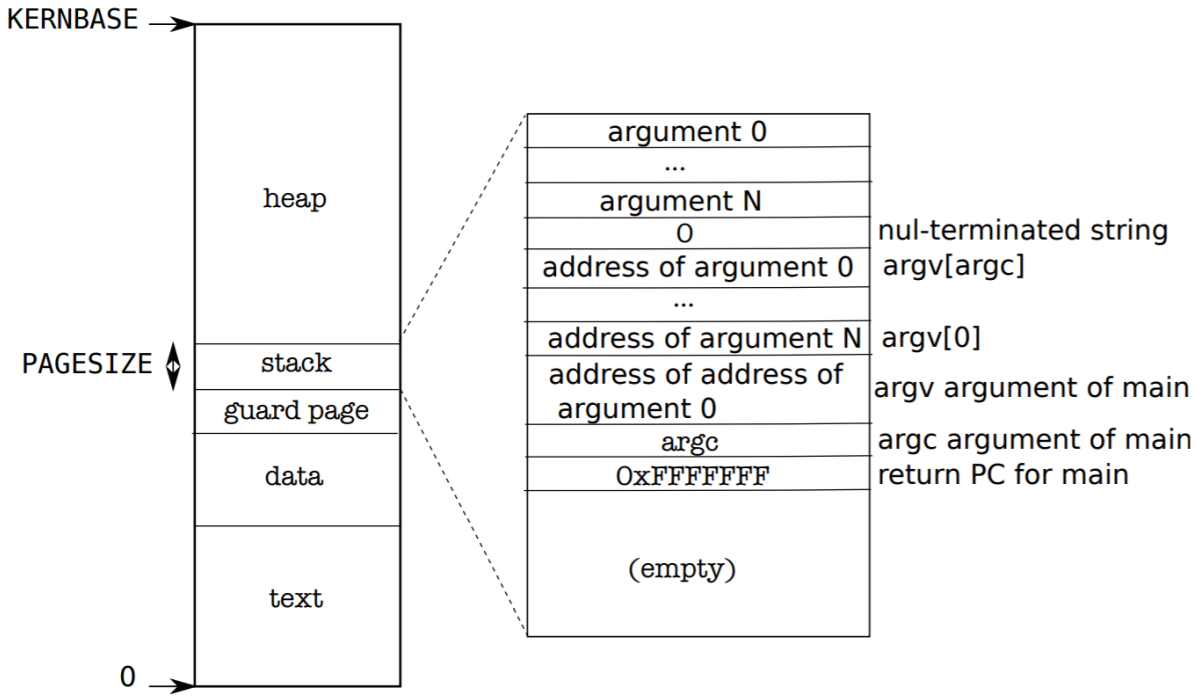
طراحان: آیلین جمالی، محمدعلی توفیقی



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[۱] به آدرس تبدیل خواهد شد. حافظه یک برنامه سطح کاربر شامل بخش‌های مختلفی مانند کد، پشته[۲] و است. این ساختار برای یک برنامه در xv6 در شکل زیر نشان داده شده است.

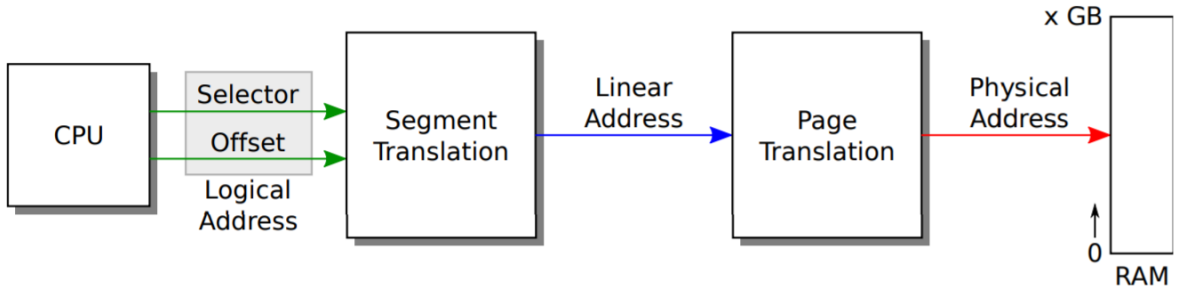


۱) ساختار حافظه مجازی (مشابه شکل بالا) یک برنامه در لینوکس در معماری x86 (۳۲ بیتی) را نشان دهید. (راهنمایی: می‌توانید به منبع [۱] رجوع کنید.)

همان‌طور که در آزمایش یک ذکر شد، در مد محافظت‌شده[۴] در معماری x86 هیچ کدی (اعم از کد هسته یا کد برنامه سطح کاربر) دسترسی مستقیم به حافظه فیزیکی[۵] نداشته و تمامی آدرس‌های برنامه

[1] Linker
[2] Stack
[3] Heap
[4] Protected Mode
[5] Physical Memory

از خطی[1] به مجازی[2] 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[3] داشته که در حین فرایند تعویض‌متن[4] 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[5] 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[6] 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

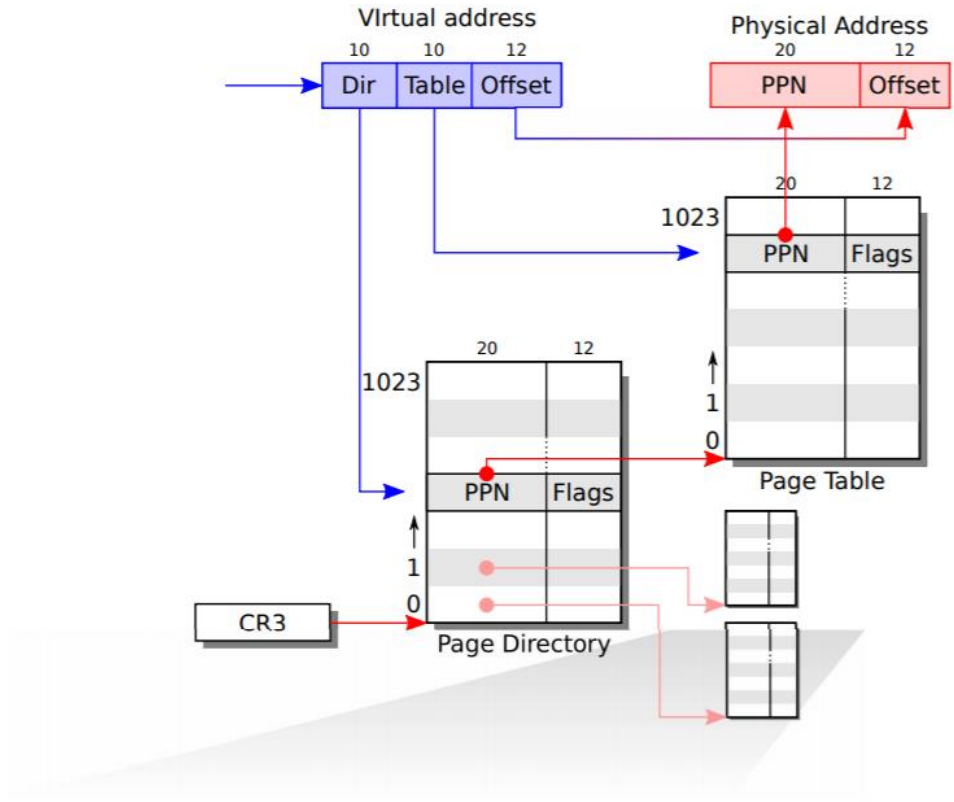
[1] Linear
[2] Virtual
[3] Page Table
[4] Context Switch
[5] Paging
[6] Address Spaces

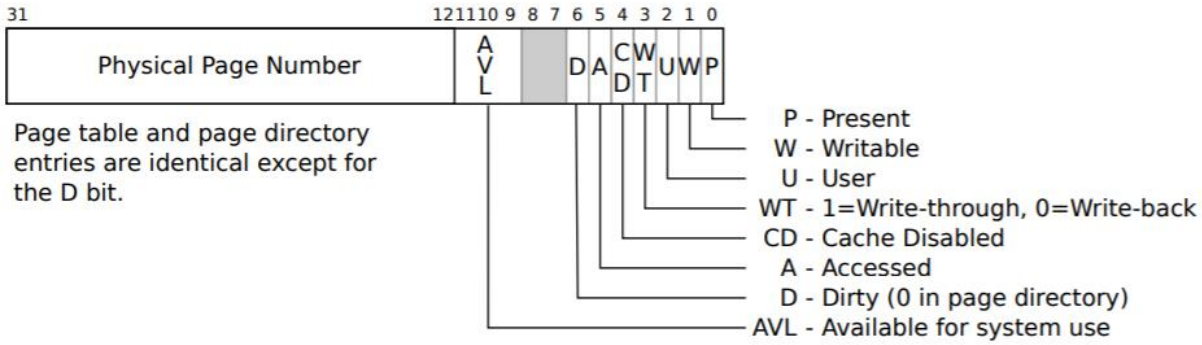
آدرس[۱] (ASLR)) از آدرس 0x08048000 آغاز شده و نیاز به تغییر در آدرس‌های برنامه‌ها متناسب با وضعیت جاری تخصیص حافظه فیزیکی نمی‌باشد.

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[۲] اطلاق می‌شود.

ساختار جدول صفحه در معماری x86 (در حالت بدون گسترش آدرس فیزیکی[۳] (PAE) و گسترش اندازه صفحه[۴] (PSE)) در شکل زیر نشان داده شده است.

[1] Address Space Layout Randomization
[2] Memory Swapping
[3] Physical Address Extension
[4] 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.

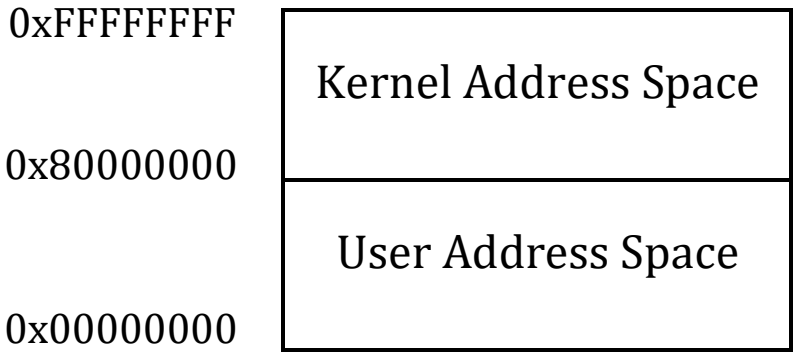
۲) چرا ساختار سلسله‌مراتبی منجر به کاهش مصرف حافظه می‌گردد؟

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

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.[۱] 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

[۱] 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.

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 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.

۶) 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[1] (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).

[1] Process Control Block

شرح آزمایش

در بخش پیاده‌سازی این پروژه از آزمایشگاه، درباره مدیریت حافظه مجازی در xv6 مطالبی ارائه شده و چند ویژگی هسته‌های امروزی سیستم‌عامل‌ها به xv6 افزوده خواهد شد. در بخش اول، بروز خطا هنگام دسترسی به اشاره‌گر NULL و در بخش دوم، حافظه مشترک در هسته سیستم‌عامل xv6 پیاده‌سازی می‌گردد. پیش از انجام این فاز، توصیه می‌شود ویدیوی توجیهی پروژه را مشاهده و فصل دوم از کتاب xv6 را مطالعه کنید.

بخش اول: بروز خطا هنگام دسترسی به اشاره‌گر NULL

یک اشاره‌گر NULL، اشاره‌گری است که به یک نقطه غیرمعتبر از حافظه[۱] اشاره می‌کند. دسترسی به این خانه از حافظه توسط کد برنامه، در سیستم‌عامل‌ها معمولاً غیر مجاز است. همگی با خطاهای دسترسی به اشاره‌گر NULL در سیستم‌عامل‌های خود آشنا هستیم. اما در هسته xv6، در حال حاضر چنین خطایی رخ نمی‌دهد و در صورتی از این خانه بخوانیم، در واقع نخستین دستورات کد برنامه‌مان (بخش متن[۲]) را خوانده‌ایم.

برای درک بیشتر این مسئله، یک برنامه سمت کاربر بنویسید و در آن از خانه صفر حافظه بخوانید. نام این برنامه را nullread.c بگذارید.

توجه کنید که ممکن است هنگام اجرای این برنامه به تله ناشی از کد عملیاتی غیرمجاز[۳] برخورد کنید. این رفتار به این علت است که در نتیجه بهینه‌سازی هنگام کامپایل، کامپایلر شما دسترسی‌های NULL را شناسایی کرده و با توجه به این که دسترسی به این خانه در اغلب سیستم‌ها منجر به رفتار تعریف‌نشده[۴]

---

[۱] در اغلب پیاده‌سازی زبان‌ها مقدار اشاره‌گر NULL صفر است.

[2] Text Section
[3] Illegal Opcode
[4] Undefined Behavior

می‌گردد، یک دستور نامعتبر را به جای خواندن از NULL جایگزین کرده است[۱]. برای حل این مسئله، می‌توانید در Makefile پروژه، O2- موجود در CFLAGS را با O1- جایگزین نمایید.

در حال حاضر، فضای آدرس برنامه‌ها، از خانه صفر توسط کد برنامه اشغال شده و سایر بخش‌های آن نیز به ترتیب در آدرس‌های پس از آن آمده‌اند. اگر برنامه جدیدتان را همراه با xv6 کامپایل کنید، قطعه‌های[۲] آن مشابه خروجی زیر خواهد بود:

```
$ readelf --segments _nullread

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000080 0x00000000 0x00000000 0x009e8 0x009f4 RWE 0x10
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text.rodata.eh_frame.bss
   01
```

Segmentها اجزایی از فضای آدرس هستند که حین اجرا Sectionها را در برمی‌گیرند. مثلاً در خروجی بالا، یک قطعه از نوع LOAD داریم که آدرس مجازی آغاز آن 0x0 است و شامل بخش‌های text (همان کد برنامه)، rodata، eh_frame و bss خواهد بود. در این بخش قصد داریم تغییراتی ایجاد کنیم که دسترسی به خانه صفر توسط کد برنامه، غیرمجاز باشد و منجر به خطای صفحه[۳] شود. به همین منظور، نیاز داریم تا خانه شروع حافظه مجازی برنامه‌ها را به جای صفر، به جلوتر ببریم و صفحه‌ای از حافظه که خانه صفر در آن قرار دارد (اولین صفحه از حافظه مجازی برنامه) را به جایی نگاشت ندهیم. باتوجه به نیازمندی‌های بخش دوم پروژه، برنامه را به چهار صفحه جلوتر خواهیم برد.

---

[۱] برای اطلاعات بیشتر به این لینک مراجعه کنید.

[2] Segments
[3] Page Fault

- به Makefile پروژه نگاه کنید. چه بخشی از آن باید تغییر کند تا نقطه شروع متن برنامه‌ها به جای خانه صفر، به خانه 0x4000 (چهار صفحه جلوتر) برود؟

پس از اعمال تغییرات شما، فرمت فایل اجرایی این برنامه مشابه خروجی زیر خواهد بود:

```
$ readelf --segments _nullread

Elf file type is EXEC (Executable file)
Entry point 0x4000
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000080 0x00004000 0x00004000 0x009e8 0x009f4 RWE 0x10
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text.rodata.eh_frame.bss
   01
```

- در سمت کد هسته، به پیاده‌سازی تابع exec() در فایل exec.c دقت کنید. در این تابع، یک برنامه اجرایی از فایل خوانده شده و هر قطعه از آن در حافظه مجازی بارگذاری می‌شود. در این تابع نقطه آغاز حافظه مجازی همواره از خانه صفر در نظر گرفته شده است. برای این که نخستین آدرس مجازی نگاشته شده را به چهار صفحه جلوتر ببریم، چه قسمتی از این تابع را باید تغییر دهیم؟
- هنگام اجرای fork() نیز، برای ساختن حافظه پردازه فرزند، حافظه پردازه والد کپی می‌شود (copyuvm()). این تابع به چه تغییری نیاز دارد؟
- نخستین برنامه اجرا شده (پردازه initcode) توسط خود هسته بارگذاری می‌شود (تابع userinit()). فراموش نکنید که صفحات این پردازه را نیز اصلاح کنید.

توجه کنید که سؤالاتی که حین توضیحات این بخش مطرح شد صرفاً جهت راهنمایی شما برای حل مسئله است و نیازی به آوردن پاسخ آن‌ها در گزارش نیست.

نکات:

- پس از اعمال این تغییرات، اجرای nullread و به طور کلی دسترسی به خانه صفر حافظه باید منجر به وقوع تله شماره ۱۴ (خطای صفحه) و در نتیجه کشته شدن پردازه گردد.
- در این مرحله، چهار صفحه ابتدایی فضای آدرس مجازی یک پردازه هرگز نباید نگاشتی داشته باشد.

راهنمایی‌ها:

- در Makefile، فایل اجرایی نهایی برنامه‌های سمت کاربر با پیش‌وند _ (Underscore) مشخص شده‌اند (مثلاً _echo). از این پیش‌وند برای یافتن دستوراتی که هنگام کامپایلِ آن‌ها اجرا می‌شود استفاده کنید.
- تنظیم حافظه مجازی مورد استفاده برنامه‌ها در مرحله پیوند[1] انجام می‌شود. پیونددهنده[2] مورد استفاده هنگام کامپایل سیستم‌عامل، در Makefile با متغیر LD مشخص شده است. از موارد استفاده این متغیر برای پیدا کردن دستور مربوط به پیوند استفاده کنید.
- همان‌طور که در بخش توضیحات نیز اشاره شد، ترتیب پر شدن فضای آدرس پردازه‌ها در xv6، از آدرس کمتر آغاز شده و به ترتیب با کد و داده‌های برنامه (که از فایل اجرایی خوانده می‌شود)، یک صفحه خالی نگاشته نشده به عنوان محافظ، پشته[3] (با سایز ثابت یک صفحه)، و هیپ (با سایز متغیر) اشغال شده‌است. برای درک بهتر نحوه پر شدن فضای آدرس برنامه‌ها، این پیاده‌سازی را در تابع exec() بررسی کنید.

[1] Link
[2] Linker
[3] Stack

بخش دوم: حافظه مشترک[1]

یکی از روش‌های رایجی که در سیستم‌عامل‌ها برای ارتباط پردازه‌ها با یک‌دیگر[2] وجود دارد، استفاده از حافظه مشترک است؛ اما در حال حاضر در xv6 مکانیزمی برای این مدل ارتباط وجود ندارد. در این بخش از پروژه، قصد داریم ویژگی ساده شده‌ای از این قابلیت را به xv6 اضافه کنیم.

برای این منظور، یک فراخوانی سیستمی با نام shmget() اضافه می‌کنیم که شناسه یک صفحه مشترک را به عنوان پارامتر دریافت کرده و پس از نگاشت آن به حافظه مجازی پردازه، آدرس مجازی متناظر با آن را برمی‌گرداند.

امضای این فراخوانی سیستمی به صورت زیر خواهد بود:

```
void* shmget(int shared_page_id)
```

در این بخش، جهت سهولت فرض می‌شود سه صفحه با شناسه‌های یک تا سه می‌توانند میان پردازه‌ها به اشتراک گذاشته شوند. این صفحات فیزیکی به ترتیب در صفحه‌های دوم تا چهارم حافظه مجازی پردازه (که در بخش قبلی آن‌ها را خالی کردیم) نگاشته می‌شوند. برای مثال، اگر پردازه‌ای shmget(2) را صدا کند، باید دومین صفحه فیزیکی مشترک، به سومین صفحه در آدرس مجازی این پردازه نگاشته شود. در نتیجه، فضای آدرس مجازی آن به این صورت تغییر خواهد کرد:

بازه آدرس مجازی	در ابتدا	پس از فراخوانی shmget(2)
0x4000 - KERNBASE	کد، پشته، هیپ	کد، پشته، هیپ
0x3000 - 0x4000	نگاشته نشده	نگاشته نشده
0x2000 - 0x3000	نگاشته نشده	دومین صفحه فیزیکی مشترک
0x1000 - 0x2000	نگاشته نشده	نگاشته نشده
0x0000 - 0x1000	نگاشته نشده	نگاشته نشده

[1] Shared Memory
[2] Inter-Process Communication

نکات:

- صفحه‌های مشترک مختص یک پردازه و فرزندهای آن نیستند و هر پردازه‌ای در هر زمان می‌تواند با فراخوانی shmget(n) در اشتراک صفحه مشترک nاُم سهیم شود.
- نیازی به در نظر گرفتن موارد مربوط به همگام‌سازی[1] در ساختمان داده‌ای که برای ذخیره حافظه مشترک استفاده می‌کنید نیست.
- در صورتی که هنگام اجرای این فراخوانی سیستمی مشکلی رخ داد و یا شناسه دریافت شده خارج از بازه یک تا سه بود، اشاره‌گر NULL را برگردانید.
- در یک پردازه، ممکن است shmget() با یک شناسه چندبار فراخوانی شود. لذا تنها در صورتی نگاشت را انجام دهید که صفحه مجازی مورد نظر قبلاً نگاشته نشده باشد.
- هنگام اجرای fork()، پردازه فرزند نیز باید تمامی صفحه‌های مشترکی که پردازه والد برای خود نگاشته را به همراه داشته باشد.
- دسترسی به آدرس‌های مجازی که هنوز نگاشته نشده‌اند باید منجر به وقوع تله خطای صفحه و کشتن پردازه شود (در صورتی که پیاده‌سازی صحیحی از این بخش داشته باشید، این تله باید بدون انجام تغییری از جانب شما رخ دهد).

راهنمایی‌ها:

- برای نگهداری حافظه فیزیکی مشترک، می‌توانید از یک متغیر سراسری استفاده کنید.
- برای اختصاص صفحه فیزیکی و نگاشت آن به جدول صفحه پردازه، از پیاده‌سازی تابع allocuvm() کمک بگیرید.

[1] Synchronization

- برای پیدا کردن مدخل جدول صفحه متناظر با یک آدرس مجازی، از تابع walkpgdir() استفاده کنید.
- هنگامی که یک پردازه نابود شده و یا به طور کلی کار آن به پایان می‌رسد، تمامی فضای آدرس آن آزاد می‌شود (تابع freevm()). با وجود صفحه‌های مشترک، دقت کنید که صفحات فیزیکی مربوط به این صفحات آزاد نشوند.
- هنگام پیاده‌سازی فراخوانی سیستمی، برای دسترسی راحت‌تر به توابع مربوط به حافظه مجازی، بهتر است منطق آن را در یک تابع کمکی در vm.c بنویسید و در فراخوانی سیستمی خود آن را صدا کنید.
- هر جا که جدول صفحه یک پردازه را تغییر دادید، باید سخت‌افزار را از این تغییر آگاه کنید. برای این منظور، باید رجیستر CR3 را با جدول صفحه جدید پردازه بروزرسانی کنید (از تابع lcr3() استفاده کنید).

سایر نکات

- کدهای شما باید به زبان C بوده و و نام‌گذاری فایل‌ها و توابع مانند الگوهای مذکور باشد.
- جهت آزمون صحت عملکرد پیاده‌سازی، برای هر بخش یک برنامه سمت کاربر بنویسید که عملکرد تغییرات اعمال‌شده را در شرایط گوناگون مورد بررسی قرار دهد. هنگام تحویل پروژه، صحت پیاده‌سازی شما مقابل برنامه‌های سمت کاربر دیگری نیز سنجیده خواهد شد.
- برای تحویل پروژه، یک مخزن خصوصی در سایت GitLab ایجاد نموده و کد هر دو بخش را در یک شاخه[1] آن Push کنید. سپس اکانت UT_OS_TA را با دسترسی Maintainer به مخزن خود اضافه کنید. کافی است در محل بارگذاری در سایت درس، آدرس مخزن، شناسه آخرین Commit و گزارش پروژه را بارگذاری نمایید.
- 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.
- هر گونه سؤال در مورد پروژه را فقط از طریق فروم درس مطرح نمایید.

موفق باشید

[1] Branch

مراجع

[1] Wolfgang Mauerer. 2008. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK.