به نام خدا

پروژه پنجم آزمایشگاه سیستم‌عامل

(مدیریت حافظه در xv6)

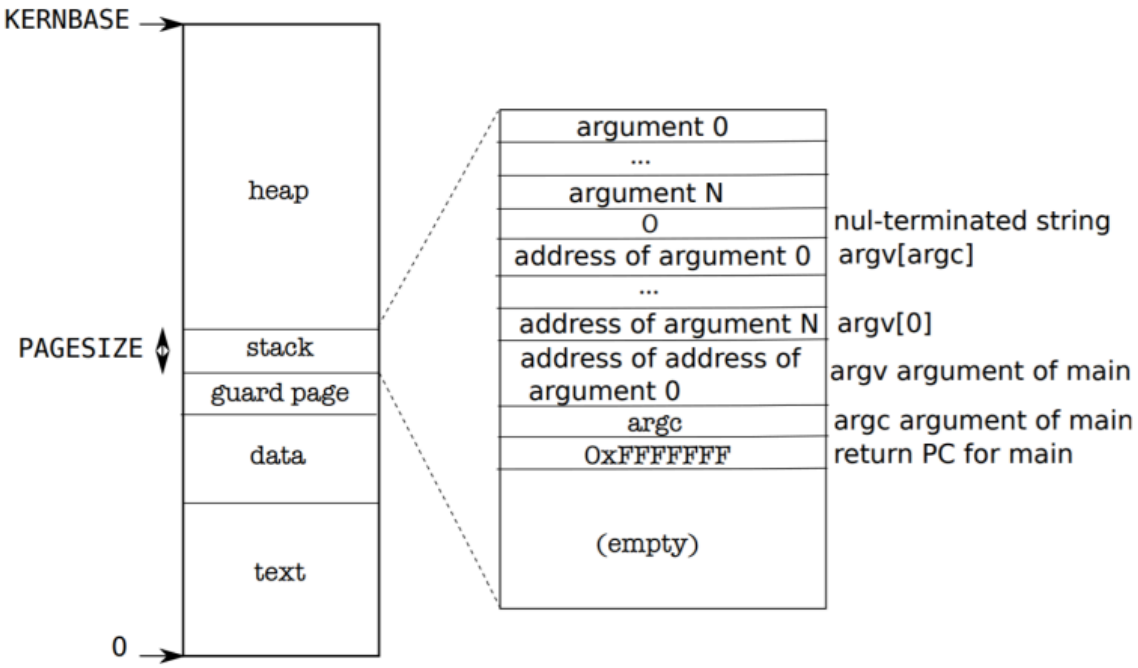
طراحان: سروش صادقیان، بهراد علمی

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

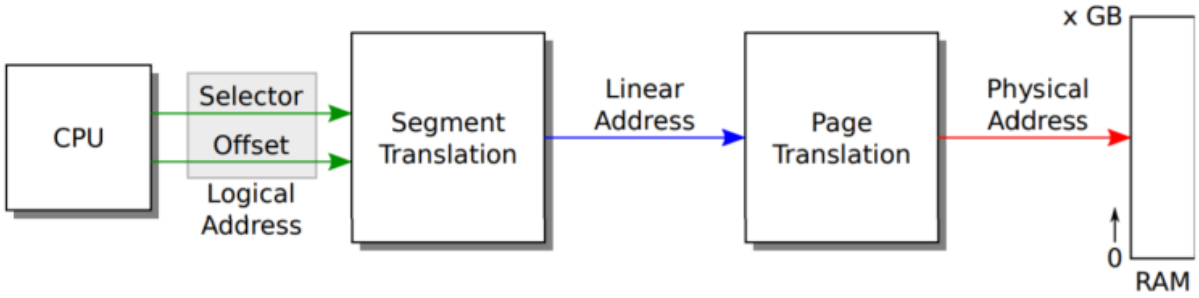
یک برنامه، حین اجرا تعامل‌های متعددی با حافظه دارد. دسترسی به متغیرهای ذخیره شده و فراخوانی توابع موجود در نقاط مختلف حافظه مواردی از این ارتباط‌ها می‌باشد. معمولاً کد منبع دارای آدرس نبوده و از نمادهایی برای ارجاع به متغیرها و توابع استفاده می‌شود. این نمادها توسط کامپایلر و پیونددهنده[1] به آدرس تبدیل خواهند 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:

[1] Linker
[2] Stack
[3] Heap



۱) راجع به مفهوم ناحیه مجازی[4] (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[5] در معماری x86 هیچ کدی (اعم از کد هسته یا کد برنامه سطح کاربر) دسترسی مستقیم به حافظه فیزیکی[6] نداشته و تمامی آدرس‌های برنامه از خطی[7] به مجازی[8] و سپس به فیزیکی تبدیل می‌شوند. این نگاشت در شکل زیر نشان داده شده است:



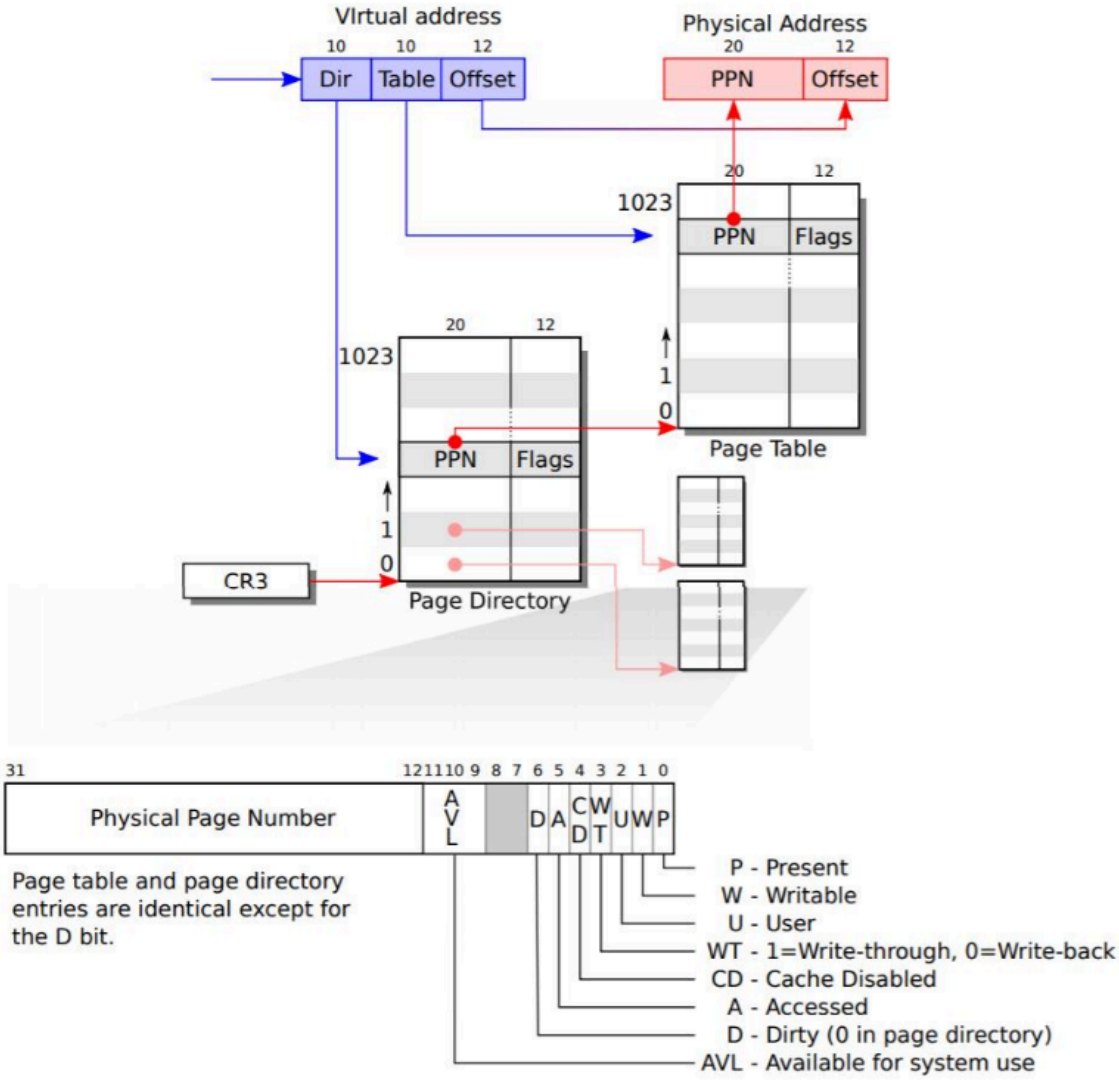
[4] Virtual Memory Area
[5] Protected Mode
[6] Physical Memory
[7] Linear
[8] Virtual

به همین منظور، هر برنامه یک جدول اختصاصی موسوم به جدول صفحه[9] داشته که در حین فرآیند تعویض متن[10] بارگذاری شده و تمامی دسترسی‌های حافظه (اعم از دسترسی به هسته یا سطح کاربر) توسط آن برنامه توسط این جدول مدیریت می‌شوند.
به علت عدم استفاده صریح از قطعه‌بندی در بسیاری از سیستم‌عامل‌های مبتنی بر این معماری، می‌توان فرض کرد برنامه‌ها از صفحه‌بندی[11] 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[12] 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 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 آدرس[13]) 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 کم استفاده (در جدول صفحه) و انتقال آن‌ها به دیسک، حافظه بیشتری در دسترس خواهد بود. این عمل جابه‌جایی حافظه[14] اطلاق می‌شود.

[9] Page Table
[10] Context Switch
[11] Paging
[12] Address Spaces
[13] Address Space Layout Randomization (ASLR)
[14] Memory Swapping

ساختار جدول صفحه در معماری x86 در حالت بدون گسترش آدرس فیزیکی[15] (PAE) و گسترش اندازه صفحه[16] (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؟

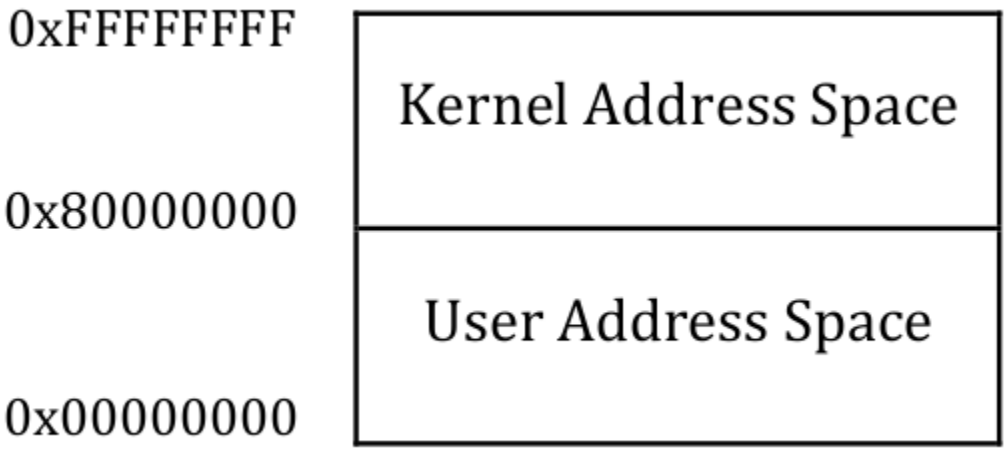
[15] Physical Address Extension
[16] Page Size Extension

۳) محتوای هر بیت یک مدخل[17] (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 محافظت‌شده و ساز و کار 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. [18] 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 این دسته در شکل زیر نشان داده 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 دقیقا شبیه به پردازه‌ها خواهد بود. زیرا این کد، همواره در هسته اجرا شده و پس از بوت غالبا، در اوقات بیکاری سیستم اجرا می‌شود.

---

[17] Entry

[18] 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 باقی پردازه‌ها در ادامه اجرای سیستم توسط توابع ()fork یا ()exec مقداردهی می‌شوند. 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.

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 پردازه [19] یک پردازه موجود است. معمولا 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 سیستم‌های‌های 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 پردازه (initcode) چیده می‌شود. در پیاده‌سازی ()exec مشابه قبل ()setupkvm فراخوانی شده (خط ۶۶۳۷) تا فضای‌آدرس هسته تعیین گردد.

[19] Process Control Block

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).

۸) توابع allocuvm و mappages که در ارتباط با حافظه‌ی مجازی هستند را توضیح دهید.
۹) شیوه‌ی بارگذاری[20] برنامه در حافظه توسط فراخوانی سیستمی exec را شرح دهید.

شرح پروژه

نمایش اطلاعات درباره‌ی حافظه:

در این بخش فراخوانی‌های سیستمی[21] زیر را پیاده می‌کنید:

- printvir باید تعداد صفحات مجازی/منطقی در بخش کاربری فضای آدرس پردازه را برگرداند. برای این کار می‌توانید از جدول صفحه پردازه برای شمارش تعداد ورودی‌های جدول صفحه‌ای که آدرس فیزیکی معتبر دارند، استفاده کنید.
- printphy باید تعداد صفحات فیزیکی در بخش کاربری فضای آدرس پردازه را برگرداند. برای شمارش باید از جدول صفحات استفاده کنید و page table هایی که آدرس فیزیکی معتبری دارند را بشمارید.

چون xv6 از demand paging استفاده نمی‌کند، می‌توان انتظار داشت که تعداد صفحات مجازی و فیزیکی در ابتدا یکسان باشند. با این حال، در بخش بعدی از شما خواسته شده تا این ویژگی را تغییر دهید.

[20] Load
[21] System Call

نگاشت حافظه:

در این بخش، نسخه ساده‌ای از فراخوانی سیستمی mapex را در xv6 پیاده‌سازی می‌کنید. فراخوانی سیستمی mapex دارای یک آرگومان است: تعداد بایت‌هایی که به فضای آدرس پردازه اضافه می‌شوند. می‌توانید فرض کنید که تعداد بایت‌ها عددی مثبت و برابر با مضربی از اندازه صفحه است.

در صورتی که ورودی داده شده به فراخوانی سیستمی معتبر باشد، فضای آدرس مجازی پردازه به میزان بایت‌های مشخص شده افزایش پیدا می‌کند و آدرس مجازی اولین حافظه جدید برگردانده می‌شود. توجه کنید که صفحات مجازی جدید باید در انتهای حافظه برنامه فعلی اضافه شوند و اندازه برنامه را متناسب با آن افزایش دهند. با این حال، فراخوانی سیستمی نباید هیچ حافظه فیزیکی‌ای متناظر با صفحات مجازی جدید اختصاص دهد، زیرا این حافظه به شکل تقاضا محور اختصاص داده شده. همچنین در صورتی که ورودی فراخوانی سیستمی نامعتبر بود، مقدار 0 برگردانده می‌شود.

می‌توانید از فراخوانی‌های سیستمی قسمت قبل برای چاپ شمارش صفحات استفاده کرده تا پیاده‌سازی خود را تایید کنید. پس از فراخوانی سیستمی mapex و پیش از هر دسترسی به صفحات حافظه مپ‌شده، شما باید فقط افزایش تعداد صفحات مجازی پردازه را ببینید، نه تعداد صفحات فیزیکی.

اختصاص دادن حافظه فیزیکی برای یک صفحه مجازی مپ‌شده تنها در حین دسترسی به آن صفحه، توسط کاربر و به شکل تقاضا محور، انجام می‌شود. زمانی که کاربر به یک صفحه حافظه مپ‌شده دسترسی پیدا کند، یک خطای صفحه[22] رخ خواهد داد و اختصاص یافتن حافظه فیزیکی تنها به عنوان قسمتی از عملیات رسیدگی به خطای صفحه انجام می‌شود. علاوه بر این، اگر بیش از یک صفحه برای حافظه مجازی پردازه مپ شده باشد، حافظه فیزیکی تنها برای صفحاتی که به آن‌ها دسترسی صورت گرفته اختصاص می‌یابد و نه برای تمامی صفحات موجود در ناحیه حافظه مپ‌شده. می‌توانید از شمارش صفحات مجازی/فیزیکی برای تایید اینکه صفحات فیزیکی فقط به صورت تقاضا محور اختصاص داده شده‌اند، استفاده کنید.

در نهایت برنامه‌ای ساده برای نشان دادن صحت درستی تغییرات خود بنویسید.

[22] Page Fault

راهنمایی‌ها:

- پیاده‌سازی فراخوانی سیستم sbrk را درک کنید. فراخوانی سیستم mapex شما از منطق مشابهی پیروی خواهد کرد. در sbrk، فضای آدرس مجازی افزایش یافته و حافظه فیزیکی نیز در همان فراخوانی سیستمی اختصاص داده می‌شود. پیاده‌سازی sbrk، تابع growproc را فرا می‌خواند که به نوبه خود تابع allocuvm را برای اختصاص حافظه فیزیکی فرا می‌خواند. برای پیاده‌سازی mapex خود، شما فقط باید فضای آدرس مجازی را در پیاده‌سازی فراخوانی سیستم افزایش دهید، و حافظه فیزیکی باید در طی خطای صفحه اختصاص داده شود. شما می‌توانید allocuvm را فرا بخوانید (یا تابع مشابه دیگری بنویسید) تا در هنگام خطای صفحه، بتوانید حافظه فیزیکی اختصاص دهید.

- نسخه اصلی xv6 خطای تله صفحه را مدیریت نمی‌کند. شما باید خودتان خطای تله صفحه را در trap.c مدیریت کنید، تا حافظه را بر اساس تقاضا برای صفحه‌ای که باعث خطای صفحه شده است، اختصاص دهد. می‌توانید بررسی کنید که آیا یک تله، خطای صفحه است یا خیر (با بررسی اینکه آیا tf->trapno برابر با T_PGFLT است). به آرگومان‌های جملات cprintf در trap.c نگاه کنید تا متوجه شوید چگونه می‌توان آدرس مجازی که باعث خطای صفحه شده است را پیدا کرد.

- از PGROUNDDOWN برای گرد کردن آدرس مجازی که منجر به خطای صفحه شده استفاده کنید تا شروع مرز صفحه را به شما بدهد.

- هر زمان که جدول صفحات پردازه تغییر می‌کند، switchuvm را فراخوانی کنید تا رجیستر CR3 و TLB به‌روزرسانی شوند. این به‌روزرسانی جدول صفحات به پردازه اجازه می‌دهد تا پس از مدیریت صحیح خطای صفحه، بتواند اجرای خود را از سر بگیرد.