به نام خدا

پروژه پنجم آزمایشگاه سیستم‌عامل

(مدیریت حافظه در xv6)

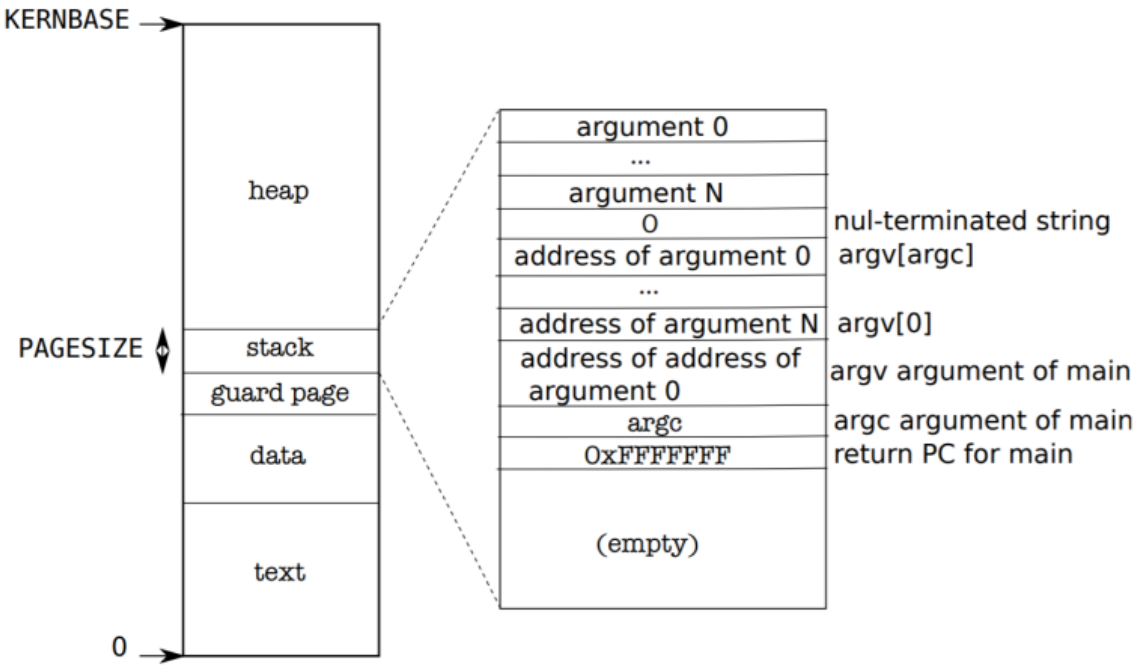
طراحان: محمدطاها فخاریان، سروش صادقیان

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

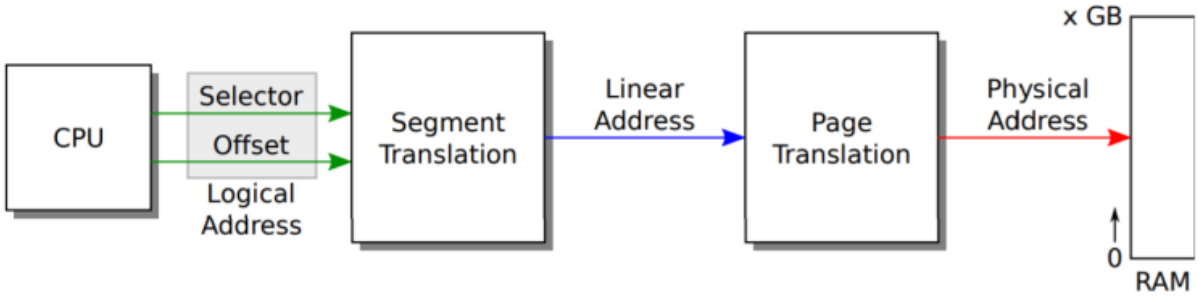
یک برنامه، حین اجرا تعامل‌های متعددی با حافظه دارد. دسترسی به متغیرهای ذخیره شده و فراخوانی توابع موجود در نقاط مختلف حافظه مواردی از این ارتباط‌ها می‌باشد. معمولاً کد منبع دارای آدرس نبوده و از نمادها برای ارجاع به متغیرها و توابع استفاده می‌شود. این نمادها توسط کامپایلر و پیونددهنده[1] به آدرس تبدیل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

[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:



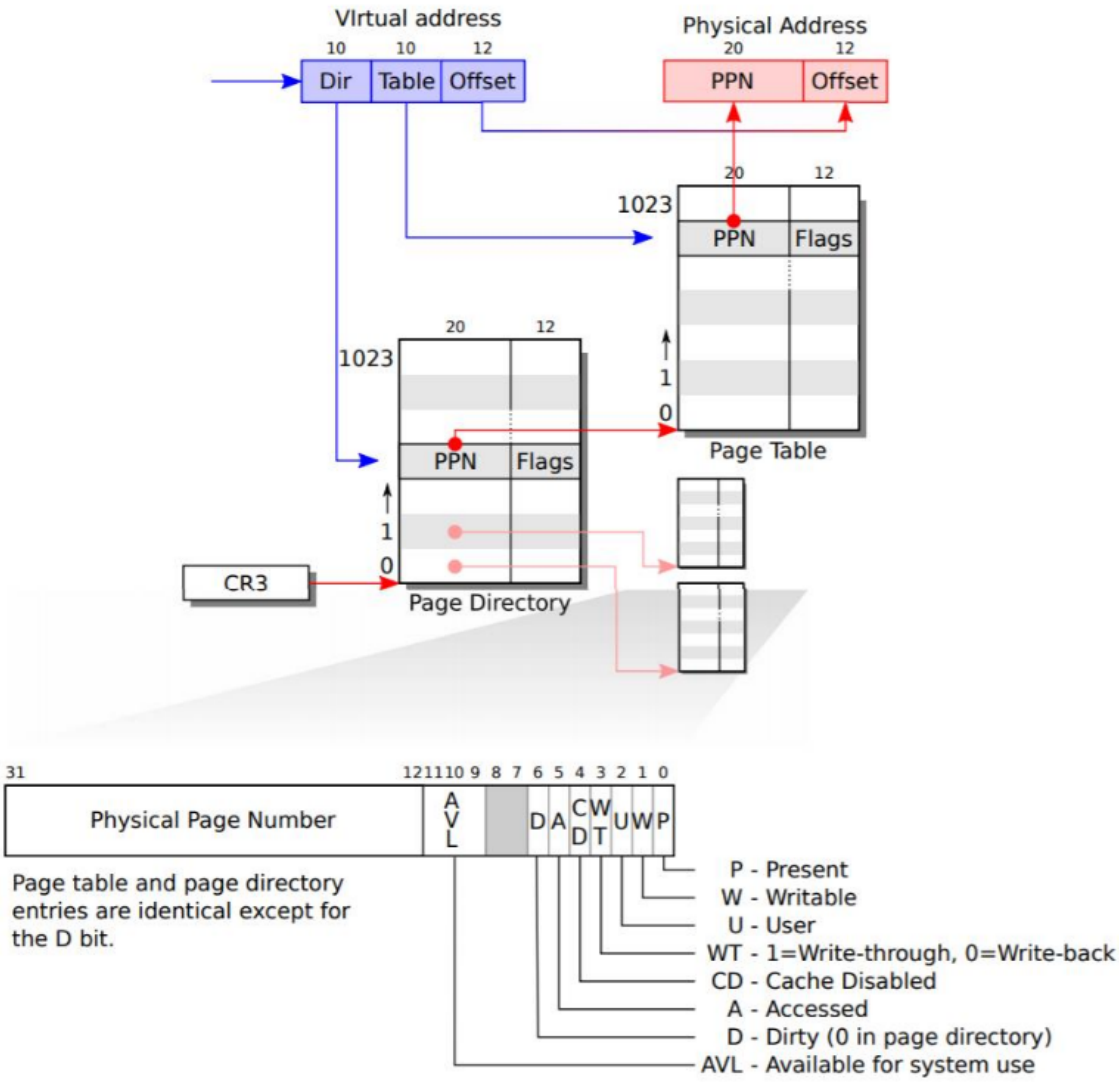
[4] Virtual Memory Area
[5] Protected Mode
[6] Physical Memory
[7] Linear
[8] Virtual

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 می‌شود.
به علیت 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] (ASLR) ازآدرس 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 حافظه[14] 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 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.

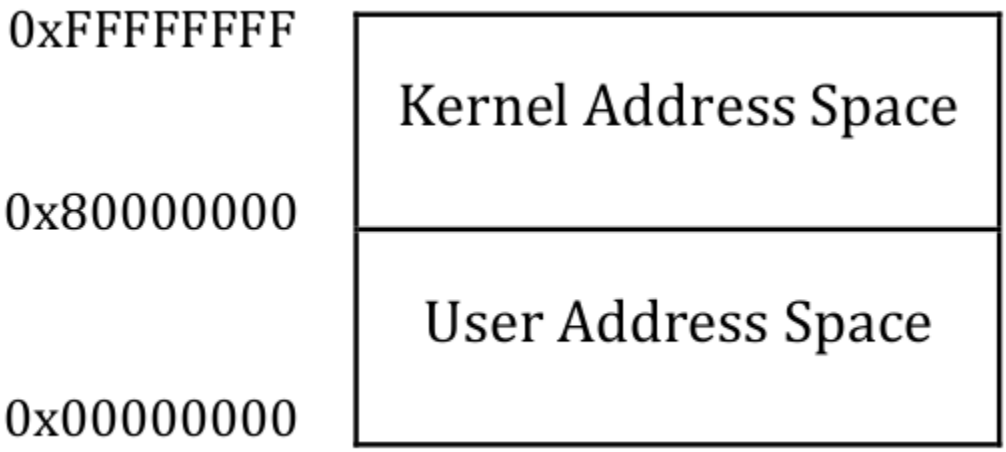
۲) چرا ساختار سلسله‌مراتبی منجر به کاهش مصرف حافظه می‌گردد؟
۳) محتوای هر بیت یک مدخل (۳۲ بیتی) در هر سطح چیست؟ چه تفاوتی میان آن‌ها وجود دارد؟

[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 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. [17] آدرس‌های کد پردازه‌ها و ریسه هسته آن‌ها توسط جدول صفحه‌ای کهاشاره‌گر 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] 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 هنگان 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 [18] 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. سپس با فراخوانی ()allocuvm فضای مورد نیاز برای کد و داده‌های برنامه جدید (خط ۶۶۵۱) و صفحه محافظ و

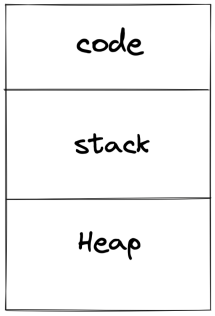
[18] Process Control Block

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).

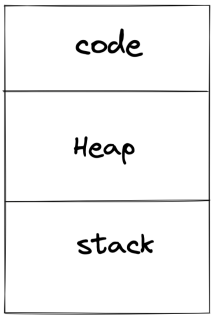
۸) توابع allocuvm و mappages که در ارتباط با حافظه‌ی مجازی هستند را توضیح دهید.
۹) شیوه‌ی بارگذاری[19] برنامه در حافظه توسط فراخوانی سیستمی exec را شرح دهید.

شرح پروژه

در این پروژه، شما فضای آدرس کد xv6 را تغییر می‌دهید. به صورت خاص، در xv6 فضای آدرس حافظه به این صورت است:



در این ساختار، فضای stack ثابت و به اندازه یک صفحه است. همچنین فضای heap نیز در صورت نیاز، تا انتهای فضای آدرس رشد می‌کند. حال می‌خواهیم در این پروژه، فضای آدرس را به گونه‌ای تغییر دهیم که شبیه به لینوکس شود:



در ساختار جدید، heap در صورت نیاز تا انتهای فضای آدرس رشد کرده و همچنین stack نیز می‌تواند در صورت نیاز، رو به بالا رشد کند و دیگر فضای آن ثابت نخواهد بود.

[19] Load

تغییر ساختار حافظه

وضعیت کلی فضای آدرس هسته xv6 را می‌توان در فایل memlayout.h رصد کرد؛ در این ساختار، حافظه کاربر از صفر شروع شده و تا مقدار KERNBASE رشد می‌کند. توجه داشته باشید که در این پروژه به هیچ عنوان فضای آدرس هسته تغییر نمی‌کند و تنها با فضای آدرس کاربر کار خواهیم داشت.
در حال حاضر، فضای حافظه برنامه به وسیله اینکه چطور برنامه بارگذاری شده و جدول صفحات آن پر می‌شود، مشخص می‌شود(تا به صفحات فیزیکی درست اشاره کنند). در xv6، این کار در فایل exec.c و به عنوان جزوی از فراخوانی سیستمی exec و با پشتیبانی از حافظه مجازی پیاده‌سازی شده در فایل vm.c، انجام شده است.
برای تغییر ساختار حافظه، نیاز است که بگونه‌ای کد exec را تغییر دهید که پس از آوردن برنامه به حافظه، فضای پشته را به نحوی مشخص کند که می‌خواهیم. توجه داشته باشید که برای این تغییر، احتمالاً نیاز خواهید داشت که اطلاعات بیشتری از حافظه را ذخیره کنید: در حال حاضر، تنها با استفاده از متغیر sz، می‌توان به انتهای فضای حافظه مجازی دسترسی داشت. در ساختار جدید، احتمالاً لازم است که به اطلاعات بیشتری مثل انتهای فضای heap و ابتدای فضای stack دسترسی داشته باشیم.

رشد فضای حافظه‌ی پشته

فضای حافظه پشته ثابت نیست و در صورت نیاز، می‌تواند رشد کند. هنگامی که فضای پشته رشد می‌کند، ممکن است به page fault برخورد کنیم؛ زیرا پشته می‌خواهد به صفحه‌ای دسترسی پیدا کند که هنوز map نشده است.
برای جلوگیری از این موضوع، ابتدا لازم است که یک شرط جدید به تله[20] اضافه کنید و پشتیبان[21] آن را به نحوی پیاده‌سازی کنید که بتواند از page fault جلوگیری کند. پیاده‌سازی به این گونه خواهد بود که ابتدا باید بررسی شود که علت به وجود آمدن تله چیست و اگر دلیل آن دسترسی به فضای حافظه‌ای بود که منجر به رشد پشته از جهت بالا می‌شد، فضای حافظه جدیدی اختصاص داده شده و map شود؛ در غیر این صورت می‌بایست از پشتیبان قبلی استفاده کنیم.

[20] Trap
[21] Handler

راهنمایی

۱- با بررسی فایل exec.c، متوجه می‌شوید که با فراخوانی allocuvm، دو صفحه، یکی برای پشته و دیگری برای یک صفحه محافظ که دقیقاً بعد از کد و داده قرار می‌گیرد، اختصاص داده می‌شود. شما باید به نحوی این قسمت از کد را تغییر دهید تا فضای پشته به جای جدیدی که می‌خواهیم، تغییر پیدا کند.

۲- احتمالاً نیاز خواهید داشت که یک اشاره‌گر به بالای پشته تعریف کنید تا بتوانید رشد آن را رصد و پشتیبانی کنید.

۳- با توجه به تغییرات داده شده در ساختار پشته، نیاز است که برخی از قسمت‌های فراخوانی‌های سیستمی در xv6 نیز تغییر کنند؛ در پیاده‌سازی برخی از این فراخوانی‌ها، برای بررسی پشته، آدرس با مقدار sz مقایسه می‌شود و احتمالاً نیاز است که این مقایسه‌ها با متغیرهای جدید انجام شود.

۴- پیاده‌سازی copyuvm هم نیاز به تغییر خواهد داشت. این تابع در فراخوانی سیستمی fork برای کپی فضای آدرس پردازه پدر استفاده می‌شود. در نسخه فعلی xv6، فرض می‌شود که پشته در ادامه کد آمده است؛ از این رو لازم است که پس از تغییرات در فضای آدرس، برای کپی فضای آدرس نیز تغییرات لازم در تابع copyuvm داده شود.

۵- برای بررسی اینکه چه آدرسی منجر به page fault شده است، می‌توانید از CR2 استفاده کنید. در xv6، برای خواندن محتوای این رجیستر، از تابع rcr2 استفاده می‌شود.