به نام خدا

پروژه پنجم آزمایشگاه سیستم‌عامل

(آشنایی با حافظه مجازی در xv6)

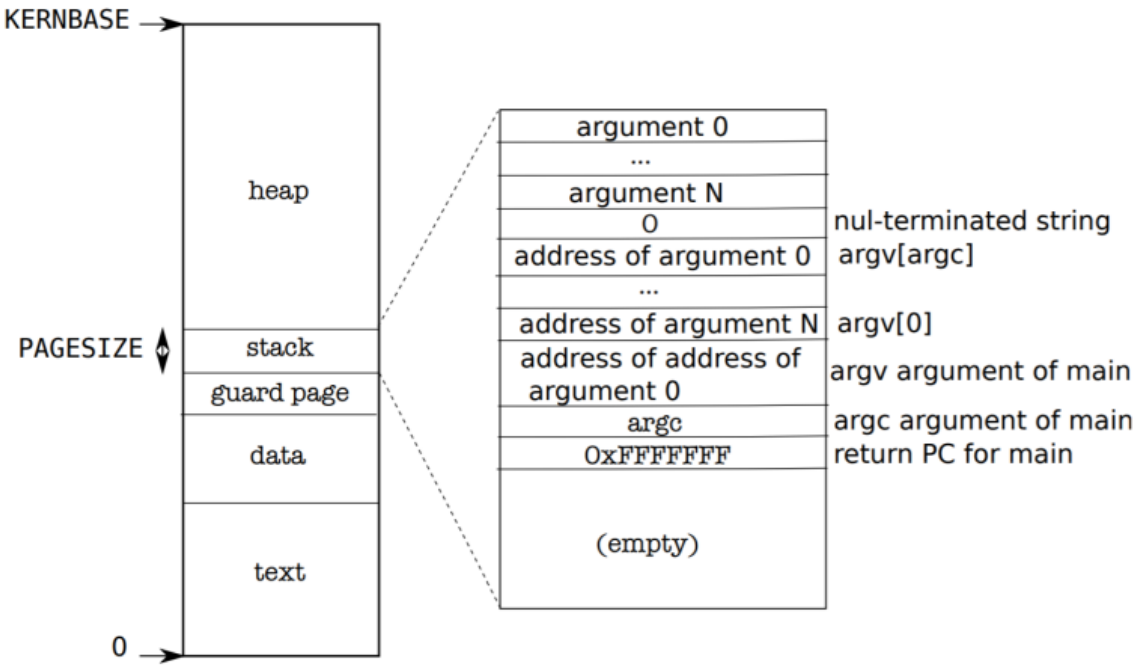
طراحان: علی زارع، آتیه آرمین، دانشور امراللهی

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

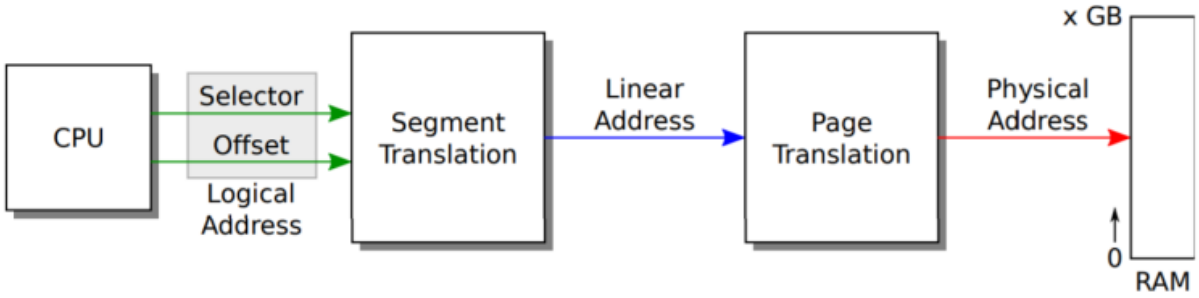
یک برنامه، حین اجرا تعامل‌های متعددی با حافظه دارد. دسترسی به متغیرهای ذخیره شده و فراخوانی توابع موجود در نقاط مختلف حافظه مواردی از این ارتباط‌ها می‌باشد. معمولا کد منبع دارای آدرس نبوده و از نمادها برای ارجاع به متغیرها و توابع استفاده می‌شود. این نمادها توسط کامپایلر و پیونددهنده[1] به آدرس تبدیل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



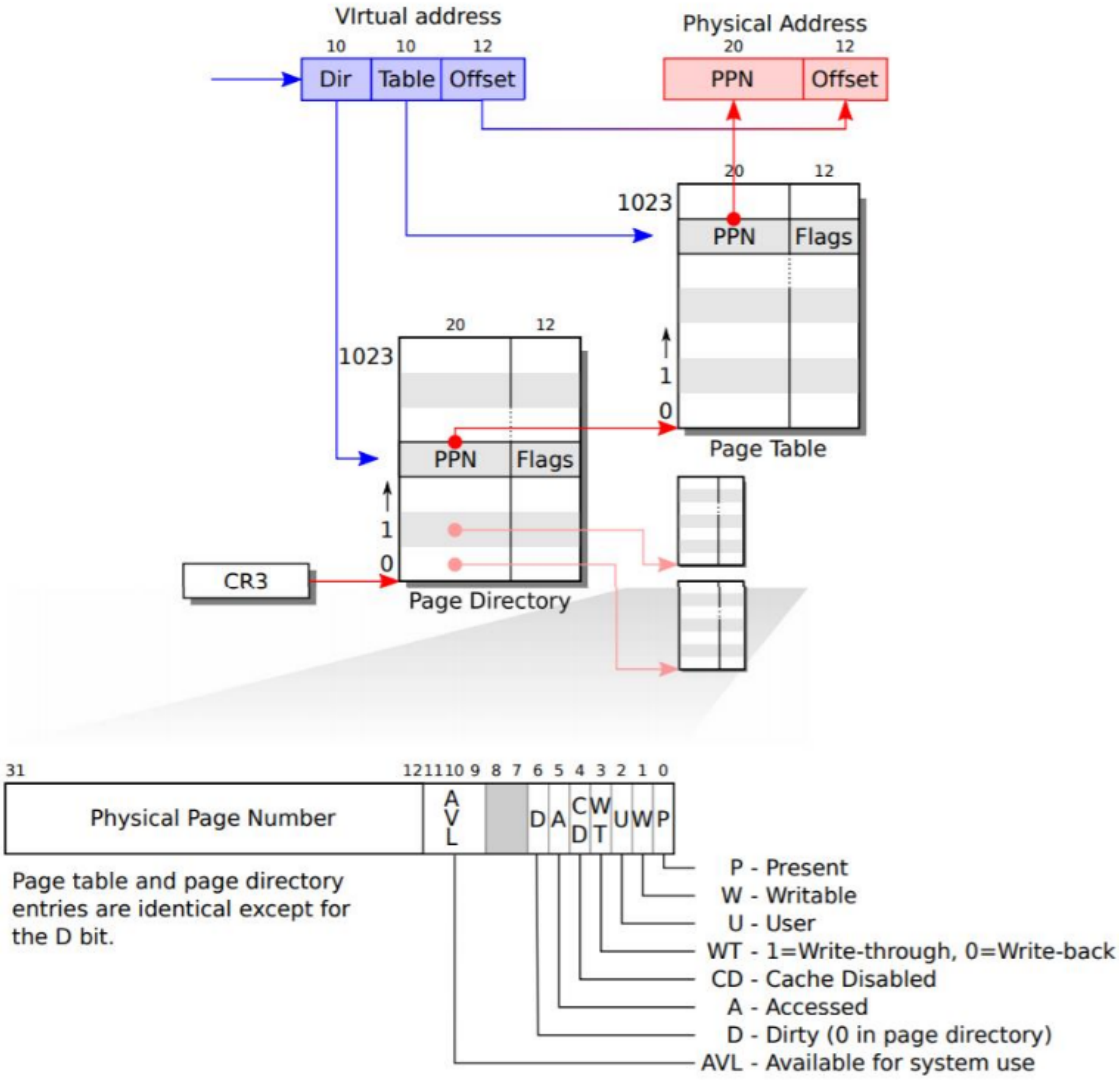
[4] Virtual Memory Area
[5] Protected Mode
[6] Physical Memory
[7] Linear
[8] Virtual

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 طو 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 گسترس 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.

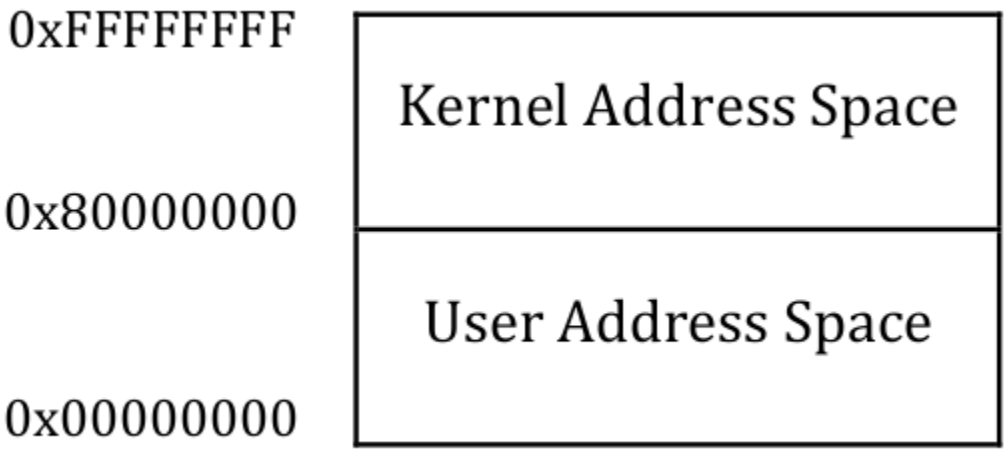
۲) چرا ساختار سلسله‌مراتبی منجر به کاهش مصرف حافظه می‌گردد؟
۳) محتوای هر بیت یک مدخل (۳۲ بیتی) در هر سطح چیست؟ چه تفاوتی میان آن‌ها وجود دارد؟

[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

[18] Process Control Block

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).

شرح پروژه

در این پروژه، شما کد xv6 را به صورتی تغییر می‌دهید که یک قابلیت از سیستم‌عامل مدرن را دارا شود: هنگامی که برنامه شما یک null pointer را dereference[19] کرد، یک استثنا[20] را raise کند. همچنین بایستی قابلیت تغییر سطوح محافظت‌سازی[21] برخی از صفحات[22] در فضای آدرس[23] یک پردازه را تغییر دهید.

Null-Pointer Dereference

در سیستم‌عامل xv6، سیستم VM از یک جدول صفحه[24] ساده دو سطحی استفاده می‌کند. در ساختار فعلی آن، کد کاربر در ابتدای فضای آدرس load می‌شود. بنابراین اگر یک اشاره‌گر null را dereference کنید، خطای illegal opcode خواهید دید. شما بایستی از این trap جلوگیری کرده و trap type شماره 14 که page fault است نشان داده شود.

بنابراین بخش اولی که بایستی انجام دهید **نوشتن برنامه‌ای است که یک null-pointer را dereference کند. آن را هم در لینوکس هم در xv6 اجرا کنید و تفاوت آن‌ها را مشاهده کنید.**

ابتدا باید متوجه شوید که xv6 یک جدول صفحه را به چه شکل تشکیل می‌دهد. هدف این پروژه بیشتر متوجه شدن کد است و نیازی نیست حجم زیادی کد جدید بنویسید.
پیشنهاد می‌شود ابتدا با مطالعه در مورد exec() متوجه شوید که فضای آدرس چگونه با کد پر می‌شود و initialize می‌شود.

---

[19] دسترسی به داده/مقدار در حافظه که اشاره‌گر به آن اشاره می‌کند:
https://stackoverflow.com/questions/4955198/what-does-dereferencing-a-pointer-mean
[20] Exception
[21] Protection Levels
[22] Pages
[23] Address Space
[24] Page Table

همچنین پیشنهاد می‌شود در مورد ()fork آن بخشی که فضای آدرس پردازه فرزند [25] ساخته می‌شود (کپی کردن فضای پردازه والد) را مطالعه کنید. آیا تغییری در این بخش باید اعمال شود؟
در ادامه پیشنهاد می‌شود نگاهی به سراسر کد xv6 بیندازید تا جاهایی که فرضیاتی راجع به فضای آدرس در نظر گرفته شده را مشاهده کنید. به این فکر کنید که وقتی یک پارامتر (مثل اشاره‌گر) به کرنل پاس داده می‌شود، کرنل چگونه تضمین می‌کند این اشاره‌گر به جای بامعنی اشاره می‌کند؟

راهنمایی: نگاهی به makefile سیستم‌عامل xv6 بیندازید. در آنجا برنامه‌های کاربر طوری کامپایل می‌شوند که نقطه آغازین آن‌ها [26] (محل اولین دستور) صفر باشد. اگر شما xv6 را طوری تغییر می‌دهید که صفحه اول نامعتبر باشد، بنابراین نقطه آغازین آن‌ها نیز بایستی نقطه دیگری باشد (مثل صفحه بعدی، یا 0x1000). بنابراین در makefile بایستی تغییراتی اعمال شود.

کد Read-only

در اکثر سیستم‌عامل‌ها، کدها به جای read-write به صورت read-only هستند. اما در xv6 اینگونه نیست و یک برنامه می‌تواند به اشتباه متن خودش را بازنویسی[27] کند. می‌توانید این موضوع را امتحان کنید!

در این بخش پروژه بایستی **بیت‌های محافظت[28] بخشی از جدول صفحه را طوری تغییر دهید که read-only شوند.** در نتیجه، امکان write کردن مجدد روی سورس برنامه‌ها وجود نخواهد داشت.

برای انجام این بخش بایستی دو تا سیستم‌کال

```
int mprotect(void *addr, int len)
```

9

```
int munprotect(void *addr, int len)
```

به xv6 اضافه کنید.

[25] Child Process
[26] Entry Point
[27] Overwrite
[28] Protection Bits

فراخوانی ()mprotect باید بیت‌های محافظت صفحاتی که در بازه addr تا addr+len قرار دارند را به حالت read-only تغییر دهد. بنابراین بعد از اجرای این سیستم‌کال، برنامه باید بتواند همچنان صفحات این ناحیه را بخواند اما اگر write صورت گیرد، بایستی یک تله[29] اتفاق بیفتد و پردازه کشته[30] شود.

فراخوانی ()munprotect بایستی برعکس سیستم‌کال ()mprotect عمل کند. یعنی بایستی ناحیه مورد نظر را به read-write تبدیل کند.

همچنین توجه کنید که تغییرات ذکر شده بایستی هنگام ()fork نیز اعمال شوند. یعنی اگر یک پردازه روی بخشی از صفحاتش سیستم‌کال ()mprotect را صدا زده باشد، سیستم‌عامل بایستی محافظت‌های read-only را برای پردازه فرزند نیز کپی کند.

حالت‌های خطا: اگر addr به ناحیه‌ای اشاره می‌کرد که بخشی از فضای آدرس نبود یا اگر len کمتر مساوی صفر بود، بایستی 1- بازگردانده[31] شود و هیچ تغییری اعمال نشود. در صورت موفقت‌آمیز بودن بایستی 0 بازگردانده شود.

راهنمایی: بعد از اعمال تغییر در یک مدخل[32] در جدول صفحه، بایستی اطمینان حاصل شود که سخت‌افزار از این تغییر آگاه است. در 32bit-xv6 این اتفاق با آپدیت کردن رجیستر CR3 اتفاق میفتد (page-table base register). هنگامی که سخت‌افزار می‌بیند مقدار جدیدی روی این رجیستر نوشته شده است، تضمین می‌کند که بروزرسانی[33]های PTE در دسترسی‌های بعدی درنظر گرفته خواهند شد. برای این بخش بایستی از تابع ()lcr3 استفاده کنید.
همچنین پیشنهاد می‌شود برای آشنایی بیشتر با PTE_W فصل سوم book-rev11 که در صفحه درس قرار گرفته‌است را مطالعه کنید.

[29] Trap
[30] Kill
[31] Return
[32] Entry
[33] Update