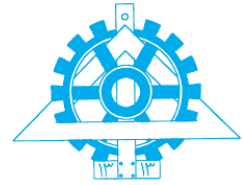




به نام خدا

آزمایشگاه سیستم‌عامل



پروژه دوم: فراخوانی سیستمی

تاریخ تحویل: ۲۳ آبان



KERNEL SPACE



USER SPACE

اهداف پروژه

- آشنایی با علت نیاز به فراخوانی سیستمی
- آشنایی با سازوکار و چگونگی صدازده شدن فراخوانی‌های سیستمی^۱ در هسته xv6
- آشنایی با افزودن فراخوانی‌های سیستمی در هسته xv6
- آشنایی با نحوه ذخیره‌سازی پرده‌ها و داده‌ساختارهای مربوط به آن

^۱ System Call

مقدمه

هر برنامه در حال اجرا یک پردازش^۲ نام دارد. به این ترتیب یک سیستم رایانه‌ای ممکن است در آن واحد، چندین پردازش در انتظار سرویس داشته باشد. هنگامی که یک پردازش در سیستم در حال اجرا است، پردازنده روال معمول پردازش را طی می‌کند: خواندن یک دستور، افزودن مقدار شمارنده برنامه^۳ به میزان یک واحد، اجرای دستور و نهایتاً تکرار حلقه. در یک سیستم رویدادهایی وجود دارند که باعث می‌شوند به جای اجرای دستور بعدی، کنترل از سطح کاربر به سطح هسته منتقل شود. به عبارت دیگر، هسته کنترل را در دست گرفته و به برنامه‌های سطح کاربر سرویس می‌دهد:^۴

(۱) ممکن است داده‌ای از دیسک دریافت شده باشد و به دلایلی لازم باشد بلافاصله آن داده از ثبات مربوطه در دیسک به حافظه منتقل گردد. انتقال جریان کنترل در این حالت، ناشی از وقفه^۵ خواهد بود. وقفه به طور غیرهمگام با کد در حال اجرا رخ می‌دهد.

(۲) ممکن است یک استثناء^۶ مانند تقسیم بر صفر رخ دهد. در این جا برنامه دارای یک دستور تقسیم بوده که عملوند مخرج آن مقدار صفر داشته و اجرای آن کنترل را به هسته می‌دهد.

(۳) ممکن است برنامه نیاز به عملیات ممتاز داشته باشد. عملیاتی مانند دسترسی به اجزای سخت‌افزاری یا حالت ممتاز سیستم (مانند محتوای ثبات‌های کنترلی) که تنها هسته اجازه دسترسی به آن‌ها را دارد. در این شرایط برنامه اقدام به فراخوانی **فراخوانی سیستمی** می‌کند. طراحی سیستم‌عامل باید به گونه‌ای باشد که مواردی از قبیل ذخیره‌سازی اطلاعات پردازش و بازیابی اطلاعات رویداد به وقوع

^۲ Process

^۳ Program Counter

^۴ در xv6 به تمامی این موارد trap گفته می‌شود. در حالی که در حقیقت در x86 نام‌های متفاوتی برای این گذارها به کار می‌رود.

^۵ Interrupt

^۶ Exception

پیوسته مثل آرگومان‌ها را به صورت ایزوله‌شده از سطح کاربر انجام دهد. در این پروژه، تمرکز بر روی فراخوانی سیستمی است.

در اکثریت قریب به اتفاق موارد، فراخوانی‌های سیستمی به طور غیرمستقیم و توسط توابع کتابخانه‌ای پوشاننده^۷ مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی `glibc` صورت می‌پذیرد.^۸ به این ترتیب قابلیت حمل^۹ برنامه‌های سطح کاربر افزایش می‌یابد. زیرا به عنوان مثال چنانچه در ادامه مشاهده خواهد شد، فراخوانی‌های سیستمی با شماره‌هایی مشخص می‌شوند که در معماری‌های مختلف، متفاوت است. توابع پوشاننده کتابخانه‌ای، این وابستگی‌ها را مدیریت می‌کنند. توابع پوشاننده `xv6` در فایل `usys.S` توسط ماکروی `SYSCALL` تعریف شده‌اند.

(۱) کتابخانه‌های (قاعدتاً سطح کاربر، منظور فایل‌های تشکیل‌دهنده متغیر `ULIB` در `Makefile` است) استفاده شده در `xv6` را از منظر استفاده از فراخوانی‌های سیستمی و علت این استفاده بررسی نمایید.

تعداد فراخوانی‌های سیستمی، وابسته به سیستم‌عامل و حتی معماری پردازنده است. به عنوان مثال در لینوکس، فری‌بی‌اس‌دی^{۱۰} و ویندوز ۷ به ترتیب حدود ۳۰۰، ۵۰۰ و ۷۰۰ فراخوانی سیستمی وجود داشته که بسته به معماری پردازنده اندکی متفاوت خواهد بود [۱]. در حالی که `xv6` تنها ۲۱ فراخوانی سیستمی دارد. لیست فراخوانی‌های سیستمی لینوکس را می‌توان در لینک زیر مشاهده نمود:

<https://filippo.io/linux-syscall-table/>

فراخوانی سیستمی سربارهایی دارد: (۱) سربار مستقیم که ناشی از تغییر مد اجرایی و انتقال به مد ممتاز بوده و (۲) سربار غیرمستقیم که ناشی از آلودگی ساختارهای پردازنده شامل انواع حافظه‌های

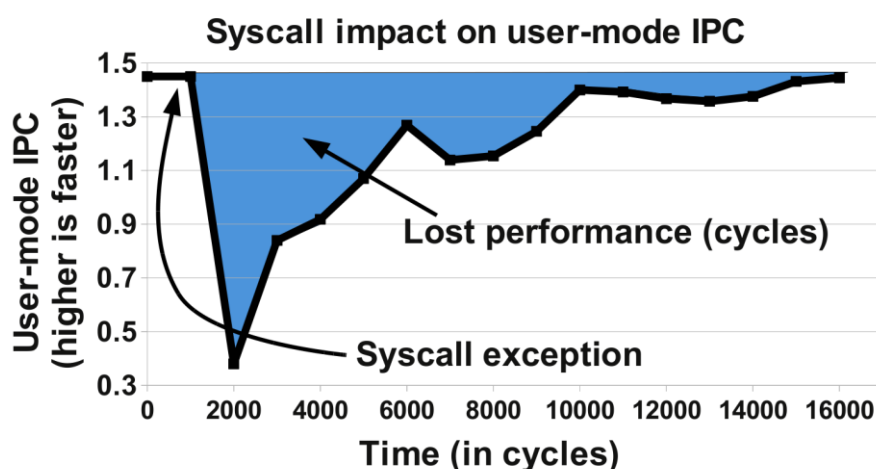
^۷ Wrapper

^۸ در `glibc`، توابع پوشاننده غالباً دقیقاً نام و پارامترهایی مشابه فراخوانی‌های سیستمی دارند.

^۹ Portability

^{۱۰} FreeBSD

نهان^{۱۱} و خط لوله^{۱۲} می‌باشد. به عنوان مثال، در یک فراخوانی سیستمی `write()` در لینوکس تا $\frac{2}{3}$ حافظه نهان سطح یک داده خالی خواهد شد [۲]. به این ترتیب ممکن است کارایی به نصف کاهش یابد. غالباً عامل اصلی، سربار غیرمستقیم است. تعداد دستورالعمل اجرا شده به ازای هر سیکل^{۱۳} (IPC) هنگام اجرای یک فراخوانی سیستمی در بار کاری SPEC CPU 2006 روی پردازنده Core i7 اینتل در نمودار زیر نشان داده شده است [۲].



مشاهده می‌شود که در لحظه‌ای IPC به کمتر از ۰,۴ رسیده است. روش‌های مختلفی برای فراخوانی سیستمی در پردازنده‌های x86 استفاده می‌گردد. روش قدیمی که در xv6 به کار می‌رود استفاده از دستور اسمبلی `int` است. مشکل اساسی این روش، سربار مستقیم آن است. در پردازنده‌های مدرن‌تر x86 دستورهای اسمبلی جدیدی با سربار انتقال کمتر مانند `sysenter/sysexit` ارائه شده است. در لینوکس، `glibc` در صورت پشتیبانی پردازنده، از این دستورها استفاده می‌کند. برخی فراخوانی‌های سیستمی (مانند `gettimeofday()` در لینوکس) فرکانس دسترسی بالا و پردازش کمی در هسته دارند. لذا سربار مستقیم آن‌ها بر برنامه زیاد خواهد بود. در این موارد می‌توان از روش‌های دیگری مانند

¹¹ Caches

¹² Pipeline

¹³ Instruction per Cycle

اشیای مجازی پویای مشترک^{۱۴} (vDSO) در لینوکس بهره برد. به این ترتیب که هسته، پیاده‌سازی فراخوانی‌های سیستمی را در فضای آدرس سطح کاربر نگاشت داده و تغییر مد به مد هسته صورت نمی‌پذیرد. این دسترسی نیز به طور غیرمستقیم و توسط کتابخانه glibc صورت می‌پذیرد. در ادامه سازوکار اجرای فراخوانی سیستمی در xv6 مرور خواهد شد.

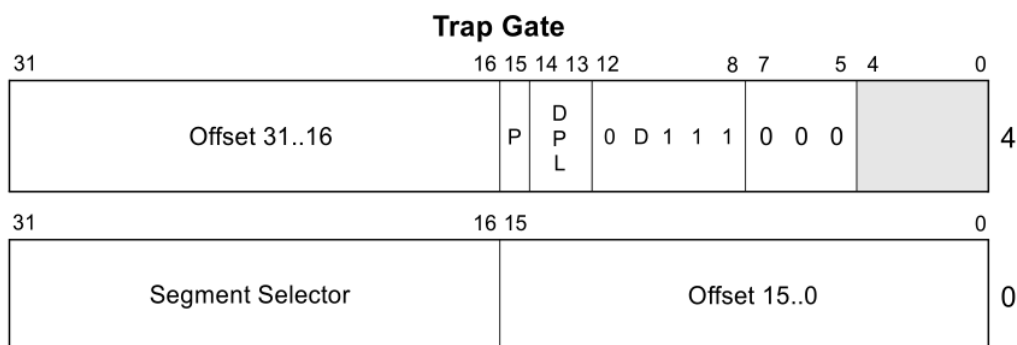
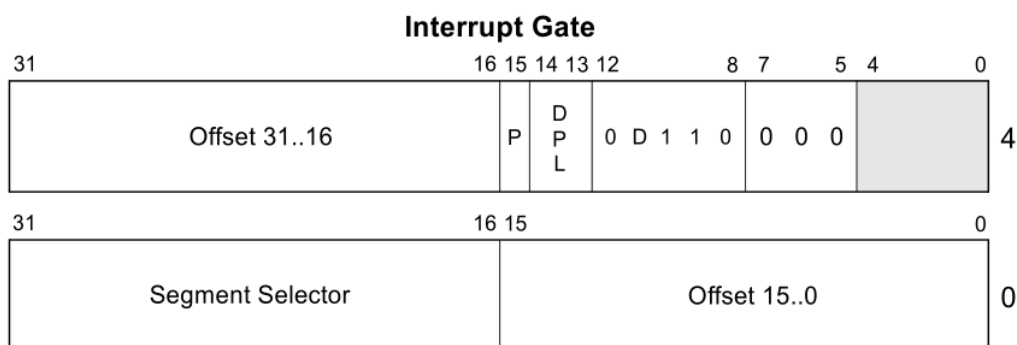
(۲) دقت شود فراخوانی‌های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش‌ها را در لینوکس به اختصار توضیح دهید. می‌توانید از مرجع [۳] کمک بگیرید.

سازوکار اجرای فراخوانی سیستمی در xv6

بخش سخت‌افزاری و اسمبلی

جهت فراخوانی سیستمی در xv6 از روش قدیمی پردازنده‌های x86 استفاده می‌شود. در این روش، دسترسی به کد دارای سطح دسترسی ممتاز (در این جا کد هسته) مبتنی بر مجموعه توصیف‌گرهایی موسوم به Gate Descriptor است. چهار نوع Gate Descriptor وجود دارد که xv6 تنها از Trap Gate و Interrupt Gate استفاده می‌کند. ساختار این Gate‌ها در شکل زیر نشان داده شده است [۴].

¹⁴ Virtual Dynamic Shared Objects



DPL Descriptor Privilege Level
 Offset Offset to procedure entry point
 P Segment Present flag
 Selector Segment Selector for destination code segment
 D Size of gate: 1 = 32 bits; 0 = 16 bits
 Reserved

این ساختارها در xv6 در قالب یک ساختار هشت بایتی موسوم به struct gatedesc تعریف شده‌اند (خط ۸۵۵). به ازای هر انتقال به هسته (فراخوانی سیستمی و هر یک از انواع وقفه‌های سخت‌افزاری و استثناها) یک Gate در حافظه تعریف شده و یک شماره تله^{۱۵} نسبت داده می‌شود. این Gate‌ها توسط تابع tvinit() در حین بوت (خط ۱۲۲۹) مقداردهی می‌گردند. Interrupt Gate اجازه وقوع وقفه در پردازنده حین کنترل وقفه را نمی‌دهد. در حالی که Trap Gate این‌گونه نیست. لذا برای فراخوانی سیستمی از Trap Gate استفاده می‌شود تا وقفه که اولویت بیشتری دارد، همواره قابل سرویس‌دهی باشد (خط ۳۳۷۳). عملکرد Gate‌ها را می‌توان با بررسی پارامترهای ماکروی مقداردهنده به Gate مربوط به فراخوانی سیستمی بررسی نمود:

^{۱۵} Trap Number

پارامتر ۱: `idt[T_SYSCALL]` محتوای Gate مربوط به فراخوانی سیستمی را نگه می‌دارد. آرایه `idt` (خط ۳۳۶۱) بر اساس شماره تله‌ها اندیس‌گذاری شده است. پارامترهای بعدی، هر یک بخشی از `idt[T_SYSCALL]` را پر می‌کنند.

پارامتر ۲: تعیین نوع Gate که در این جا Trap Gate بوده و لذا مقدار یک دارد.

پارامتر ۳: نوع قطعه کدی که بلافاصله پس از اتمام عملیات تغییر مد پردازنده اجرا می‌گردد. کد کنترل‌کننده فراخوانی سیستمی در مد هسته اجرا خواهد شد. لذا مقدار `SEG_KCODE << 3` به ماکرو ارسال شده است.

پارامتر ۴: محل دقیق کد در هسته که `vectors[T_SYSCALL]` است. این نیز بر اساس شماره تله‌ها شاخص‌گذاری شده است.

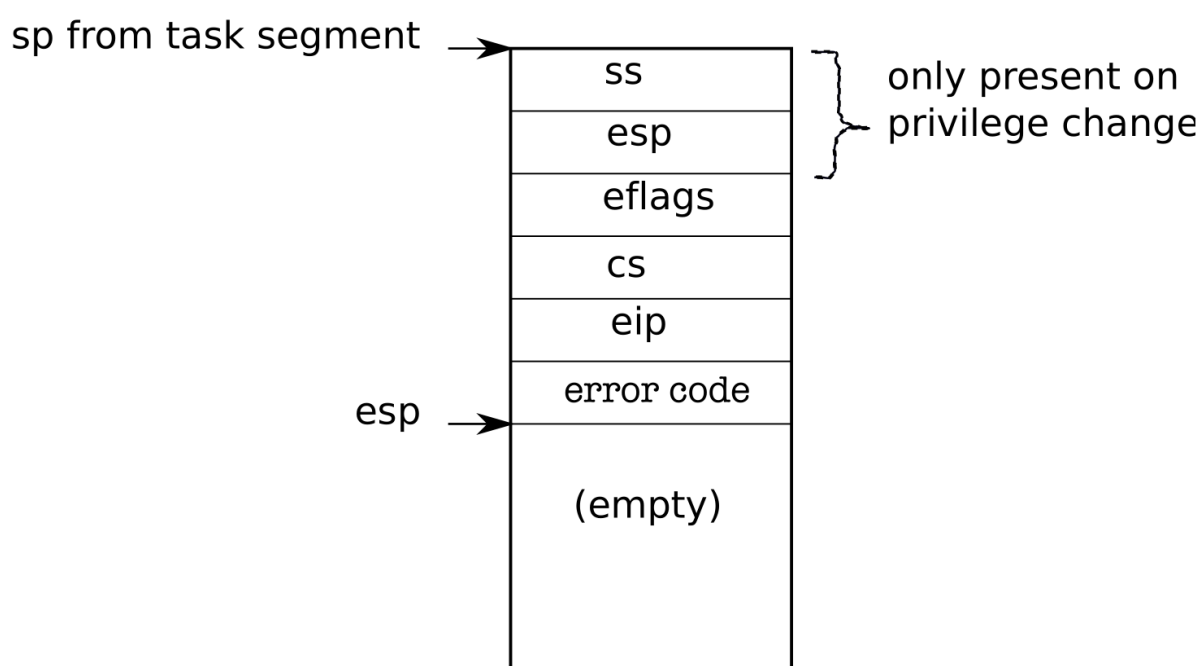
پارامتر ۵: سطح دسترسی مجاز برای اجرای این تله. `DPL_USER` است. زیرا فراخوانی سیستمی توسط (قطعه) کد سطح کاربر فراخوانی می‌گردد.

۳) آیا باقی تله‌ها را نمی‌توان با سطح دسترسی `DPL_USER` فعال نمود؟ چرا؟

به این ترتیب برای تمامی تله‌ها `idt` مربوطه ایجاد می‌گردد. به عبارت دیگر پس از اجرای `tvinit()` آرایه `idt` به طور کامل مقداردهی شده است. حال باید هر هسته پردازنده بتواند از اطلاعات `idt` استفاده کند تا بداند هنگام اجرای هر تله چه کد مدیریتی باید اجرا شود. بدین منظور تابع `idtinit()` در انتهای راه‌اندازی اولیه هر هسته پردازنده، اجرا شده و اشاره‌گر به جدول `idt` را در ثبات مربوطه در هر هسته بارگذاری می‌نماید. از این به بعد امکان سرویس‌دهی به تله‌ها فراهم است. یعنی پردازنده می‌داند برای هر تله چه کدی را فراخوانی کند.

یکی از راه‌های فعال‌سازی هر تله استفاده از دستور `int <trap no>` می‌باشد. لذا با توجه به این که شماره تله فراخوانی سیستمی ۶۴ است (خط ۳۲۲۶)، کافی است برنامه، جهت فراخوانی فراخوانی سیستمی دستور `int ۶۴` را فراخوانی کند. `int` یک دستورالعمل پیچیده در پردازنده x86 (یک

پردازنده CISC است. ابتدا باید وضعیت پردازنده در حال اجرا ذخیره شود تا بتوان پس از فراخوانی سیستمی وضعیت را در سطح کاربر بازیابی نمود. اگر تله ناشی از خطا باشد (مانند خطای نقص صفحه^{۱۶} که در فصل مدیریت حافظه معرفی می‌گردد)، کد خطا نیز در انتها روی پشته قرار داده می‌شود. حالت پشته (سطح هسته^{۱۷}) پس از اتمام عملیات سخت‌افزاری مربوط به دستور `int` (مستقل از نوع تله با فرض `Push` شدن کد خطا توسط پردازنده) در شکل زیر نشان داده شده است. دقت شود مقدار `esp` با `Push` کردن کاهش می‌یابد.



۴) در صورت تغییر سطح دسترسی، `ss` و `esp` روی پشته `Push` می‌شود. در غیراینصورت `Push` نمی‌شود. چرا؟

در آخرین گام `int`، بردار تله یا همان کد کنترل‌کننده مربوط به فراخوانی سیستمی اجرا می‌گردد که در شکل زیر نشان داده شده است.

`globl vector64`

^{۱۶} Page Fault

^{۱۷} دقت شود با توجه به اینکه قرار است تله در هسته مدیریت گردد، پشته سطح هسته نیاز است. این پشته پیش از اجرای هر برنامه سطح کاربر، توسط تابع `switchvm()` برای اجرا هنگام وقوع تله در آن برنامه آماده می‌گردد.

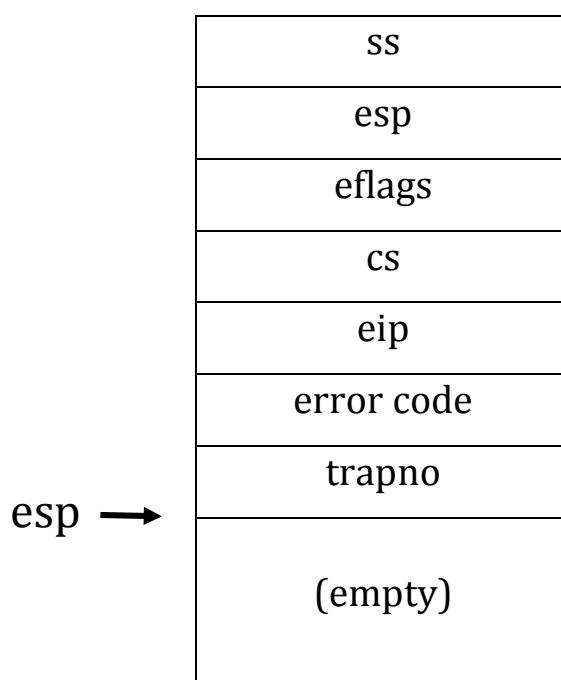
vector64:

pushl \$0

pushl \$64

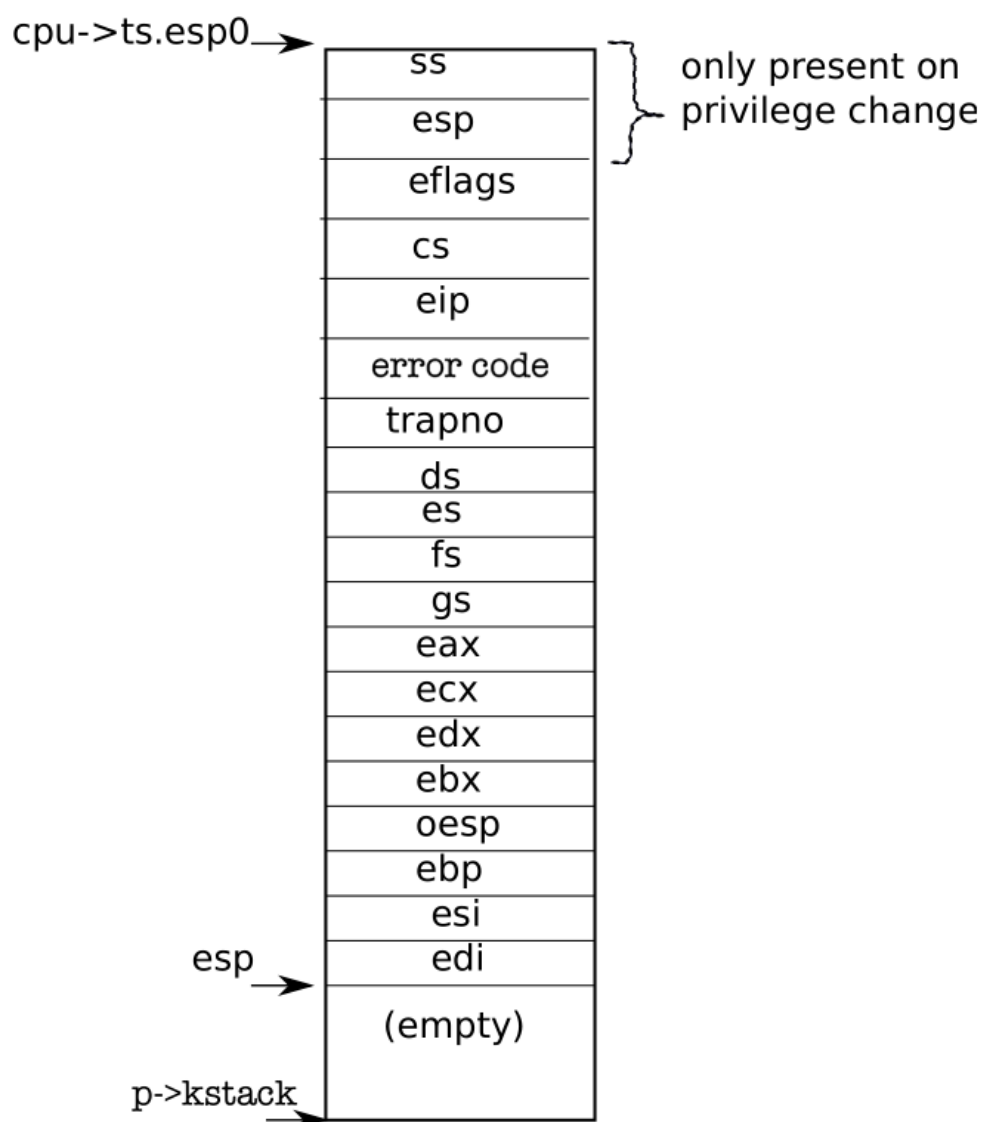
jmp alltraps

در این جا ابتدا یک کد خطای بی‌اثر صفر و سپس شماره تله روی پشته قرار داده شده است. در انتها اجرا از کد اسمبلی alltraps ادامه می‌یابد. حالت پشته، پیش از اجرای کد alltraps در شکل زیر نشان داده شده است.



alltraps باقی‌ثبات‌ها را Push می‌کند. به این ترتیب تمامی وضعیت برنامه سطح کاربر پیش از فراخوانی سیستمی ذخیره شده و قابل بازیابی است. شماره فراخوانی سیستمی و پارامترهای آن نیز در این وضعیت ذخیره شده، حضور دارند. این اطلاعات موجود در پشته، همان قاب تله هستند که در پروژه قبل مشابه آن برای برنامه initcode.S ساخته شده بود. حال اشاره‌گر به بالای پشته (esp) که در این جا اشاره‌گر به قاب تله است روی پشته قرار داده شده (خط ۳۳۱۸) و تابع trap() فراخوانی

می‌شود. این معادل اسمبلی این است که اشاره‌گر به قاب تله به عنوان پارامتر به `trap()` ارسال شود. حالت پشته پیش از اجرای `trap()` در شکل زیر نشان داده شده است.



بخش سطح بالا و کنترل‌کننده زبان سی تله

تابع `trap()` ابتدا نوع تله را با بررسی مقدار شماره تله چک می‌کند (خط ۳۴۰۳). با توجه به این که فراخوانی سیستمی رخ داده است تابع `syscall()` اجرا می‌شود. پیش‌تر ذکر شد فراخوانی‌های سیستمی، متنوع بوده و هر یک دارای شماره‌ای منحصر به فرد است. این شماره‌ها در فایل `syscall.h`

به فراخوانی‌های سیستمی نگاشت داده شده‌اند (خط ۳۵۰۰). تابع `syscall()` ابتدا وجود فراخوانی سیستمی فراخوانی شده را بررسی نموده و در صورت وجود پیاده‌سازی، آن را از جدول فراخوانی‌های سیستمی اجرا می‌کند. جدول فراخوانی‌های سیستمی، آرایه‌ای از اشاره‌گرها به توابع است که در فایل `syscall.c` قرار دارد (خط ۳۶۷۲). هر کدام از فراخوانی‌های سیستمی، خود، وظیفه دریافت پارامتر را دارند. ابتدا مختصری راجع به فراخوانی توابع در سطح زبان اسمبلی توضیح داده خواهد شد. فراخوانی توابع در کد اسمبلی شامل دو بخش زیر است:

(گام ۱) ایجاد لیستی از پارامترها بر روی پشته. دقت شود پشته از آدرس بزرگتر به آدرس کوچکتر پر می‌شود.

ترتیب `Push` شدن روی پشته: ابتدا پارامتر آخر، سپس پارامتر یکی مانده به آخر و در نهایت پارامتر نخست.

مثلاً برای تابع `f(a,b,c)` کد اسمبلی کامپایل شده منجر به چنین وضعیتی در پشته سطح کاربر می‌شود:

esp+8	C
esp+4	B
esp	A

(گام ۲) فراخوانی دستور اسمبلی معادل `call` که منجر به `Push` شدن محتوای کنونی اشاره‌گر دستورالعمل (`eip`) بر روی پشته می‌گردد. محتوای کنونی مربوط به اولین دستورالعمل بعد از تابع فراخوانی شده است. به این ترتیب پس از اتمام اجرای تابع، آدرس دستورالعمل بعدی که باید اجرا شود روی پشته موجود خواهد بود.

مثلاً برای فراخوانی تابع قبلی پس از اجرای دستورالعمل معادل `call` وضعیت پشته به صورت زیر خواهد بود:

esp+12	c
esp+8	b
esp+4	a
esp	Ret Addr

در داخل تابع `f()` نیز می‌توان با استفاده از اشاره‌گر ابتدای پشته به پارامترها دسترسی داشت. مثلاً

برای دسترسی به `b` می‌توان از `esp+8` استفاده نمود. البته این‌ها تنها تا زمانی معتبر خواهند بود

که تابع `f()` تغییری در محتوای پشته ایجاد نکرده باشد.

در فراخوانی سیستمی در `xv6` نیز به همین ترتیب پیش از فراخوانی سیستمی پارامترها روی پشته

سطح کاربر قرار داده شده‌اند. به عنوان مثال چنان‌چه در پروژه یک آزمایشگاه دیده شد، برای فراخوانی

سیستمی `sys_exec()` دو پارامتر `$argv` و `$init` و آدرس برگشتی صفر به ترتیب روی پشته قرار

داده شدند (خطوط ۸۴۱۰ تا ۸۴۱۲). سپس شماره فراخوانی سیستمی که در `SYS_exec` قرار دارد

در ثبات `eax` نوشته شده و `int $T_SYSCALL` جهت اجرای تله فراخوانی سیستمی اجرا شد.

`sys_exec()` می‌تواند مشابه آن‌چه در مورد تابع `f()` ذکر شد به پارامترهای فراخوانی سیستمی

دسترسی پیدا کند. به این منظور در `xv6` توابعی مانند `argint()` و `argptr()` ارائه شده است. پس

از دسترسی فراخوانی سیستمی به پارامترهای مورد نظر، امکان اجرای آن فراهم می‌گردد.

(۵) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در

`argptr()` بازه آدرس‌ها بررسی می‌گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد می‌کند؟ در

صورت عدم بررسی بازه‌ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی `sys_read()` اجرای

سیستم را با مشکل روبرو سازد.

شیوه فراخوانی فراخوانی‌های سیستمی جزئی از واسط باینری برنامه‌های کاربردی ^{۱۸} (ABI) یک سیستم‌عامل روی یک معماری پردازنده است. به عنوان مثال در سیستم‌عامل لینوکس در معماری x86، پارامترهای فراخوانی سیستمی به ترتیب در ثبات‌های ebx، ecx، edx، esi، edi و ebp قرار داده می‌شوند.^{۱۹} ضمن این که طبق این ABI، نباید مقادیر ثبات‌های ebx، esi، edi و ebp پس از فراخوانی تغییر کنند. لذا باید مقادیر این ثبات‌ها پیش از فراخوانی فراخوانی سیستمی در مکانی ذخیره شده و پس از اتمام آن بازیابی گردند تا ABI محقق شود. این اطلاعات و شیوه فراخوانی فراخوانی‌های سیستمی را می‌توان در فایل‌های زیر از کد منبع glibc مشاهده نمود.^{۲۰}

`sysdeps/unix/sysv/linux/i386/syscall.S`

`sysdeps/unix/sysv/linux/i386/sysdep.h`

به این ترتیب در لینوکس برخلاف xv6 پارامترهای فراخوانی سیستمی در ثبات منتقل می‌گردند. یعنی در لینوکس در سطح اسمبلی، ابتدا توابع پوشاننده پارامترها را در پشته منتقل نموده و سپس پیش از فراخوانی فراخوانی سیستمی، این پارامترها ضمن جلوگیری از دست رفتن محتوای ثبات‌ها، در آن‌ها کپی می‌گردند.

در هنگام تحویل سوالاتی از سازوکار فراخوانی سیستمی پرسیده می‌شود. دقت شود در مقابل ABI، مفهومی تحت عنوان واسط برنامه‌نویسی برنامه کاربردی (API)^{۲۱} وجود دارد که شامل مجموعه‌ای از تعاریف توابع (نه پیاده‌سازی) در سطح زبان برنامه‌نویسی بوده که واسط قابل حمل سیستم‌عامل^{۲۲} (POSIX) نمونه‌ای از آن است. پشتیبانی توابع کتابخانه‌ای سیستم‌عامل‌ها از این تعاریف، قابلیت حمل برنامه‌ها را افزایش می‌دهد.^{۲۳} مثلاً امکان کامپایل یک برنامه روی لینوکس و iOS فراهم خواهد شد.

¹⁸ Application Binary Interface

^{۱۹} فرض این است که حداکثر شش پارامتر ارسال می‌گردد.

^{۲۰} مسیرها مربوط به glibc-2.26 است.

²¹ Application Programming Interface

²² Portable Operating System Interface

^{۲۳} توابع پوشاننده فراخوانی‌های سیستمی بخشی از POSIX هستند.

جهت آشنایی بیشتر با POSIX و پیاده‌سازی آن در سیستم‌عامل‌های لینوکس، اندروید و iOS می‌توان به مرجع [۵] مراجعه نمود.

۶) چگونه هنگام بازگشت به سطح کاربر، اجرا از همان خطی که متوقف شده بود، دوباره از سر گرفته می‌شود؟ فرایند را توضیح دهید.

ارسال آرگومان‌های فراخوانی‌های سیستمی

تا این‌جا کار با نحوه ارسال آرگومان‌های فراخوانی‌های سیستمی در سیستم‌عامل xv6 آشنا شدید. در این قسمت به جای بازیابی آرگومان‌ها به روش معمول، از ثبات‌ها استفاده می‌کنیم. فراخوانی سیستمی زیر را که در آن تنها یک آرگومان ورودی از نوع `int` وجود دارد پیاده‌سازی کنید.

• `calculate_sum_of_digits(int n)`

در این قسمت به جای بازیابی آرگومان‌ها به روش معمول، از ثبات‌ها استفاده می‌کنیم. در این فراخوانی، باید مجموع ارقام ورودی را محاسبه کنید. برای مثال در صورتی که عدد ورودی ۴۹۵ باشد، شما باید عدد ۱۸ را در خروجی چاپ کنید.

دقت داشته باشید که از ثبات برای ذخیره مقدار آرگومان استفاده می‌کنیم نه برای آدرس محل قرارگیری آن. ضمن این که پس از اجرای فراخوانی، باید مقدار ثبات دست نخورده باقی بماند.

پیاده‌سازی فراخوانی‌های سیستمی

در این آزمایش ابتدا با پیاده‌سازی یک فراخوانی سیستمی، اضافه‌کردن آن‌ها به هسته xv6 را فرا می‌گیرید. در این فراخوانی که در ادامه توضیح داده می‌شود، پردازش‌هایی بر پردازش‌های موجود در

هسته و فراخوانی‌های سیستمی صدا زده شده توسط آن‌ها انجام می‌شود که از سطح کاربر قابل انجام نیست. شما باید اطلاعات فراخوانی‌های سیستمی مختلفی که توسط پردازنده‌ها صدا زده می‌شوند را ذخیره کنید و روی آن‌ها عملیاتی انجام دهید. تمامی مراحل کار باید در گزارش کار همراه با فایل‌هایی که آپلود می‌کنید موجود باشند.

نحوه اضافه کردن فراخوانی‌های سیستمی

برای انجام این کار لینک و مستندات زیادی در اینترنت و منابع دیگر موجود است. شما باید چند فایل را برای اضافه کردن فراخوانی‌های سیستمی در xv6 تغییر دهید. برای این که با این فایل‌ها بیشتر آشنا شوید، پیاده‌سازی فراخوانی‌های سیستمی موجود را در xv6 مطالعه کنید. این فایل‌ها شامل `syscall.c`، `syscall.h`، `user.h` و ... است. گزارشی که ارائه می‌دهید باید شامل تمامی مراحل اضافه کردن فراخوانی‌های سیستمی و همین‌طور مستندات خواسته‌شده در مراحل بعد باشد.

نحوه ذخیره اطلاعات پردازنده‌ها در هسته

پردازنده‌ها در سیستم‌عامل xv6 پس از درخواست یک پردازنده دیگر توسط هسته ساخته می‌شوند. در این صورت هسته نیاز دارد تا اولین پردازنده را خودش اجرا کند. هسته xv6 برای نگهداری هر پردازنده یک ساختار داده ساده دارد که در یک لیست مدیریت می‌شود. هر پردازنده اطلاعاتی از قبیل شناسه واحد خود^{۲۴} که توسط آن شناخته می‌شود، پردازنده والد و غیره را در ساختار خود دارد. برای ذخیره کردن اطلاعات بیشتر، می‌توان داده‌ها را به این ساختار داده اضافه کرد.

²⁴ PID

۱. پیاده‌سازی فراخوانی سیستمی پیدا کردن آدرس سکتورهای فایل

فایل سیستم، بخشی از سیستم‌عامل است که وظیفه ذخیره‌سازی داده‌های کاربر روی ادوات ذخیره‌سازی^{۲۵} مانند دیسک سخت^{۲۶} و ارائه واسطه‌هایی مناسب برای دسترسی به این داده‌ها در قالب فایل‌ها را دارد. در این راستا هر فایل سیستم از انتزاعات خاصی جهت مدیریت داده‌ها بهره می‌گیرد. به طور کلی، اطلاعات ذخیره‌سازی، شامل داده‌ها و فراداده‌ها^{۲۷} هستند. در فایل سیستم‌های مبتنی بر یونیکس^{۲۸}، فراداده‌های مربوط به ساختار فایل سیستم در واحدی تحت عنوان ابربلوک^{۲۹} نگه‌داری می‌گردد. برخی از این اطلاعات شامل نوع فایل سیستم، اندازه آن و اطلاعاتی همچون محل فراداده‌های خاص هر فایل می‌باشد.

فراداده خاص هر فایل در قالب Inode ذخیره شده و داده‌های هر فایل نیز در قالب بلوک‌ها قرار می‌گیرند. از جمله مهمترین فراداده‌های خاص هر فایل نگاشت بلوک‌ها به ساختارهای سطح دیسک است. بلوک‌ها واحدهای ذخیره‌سازی داده‌های اصلی فایل (محتوای فایل) از منظر سیستم‌عامل هستند. در سوی دیگر، سکتورها^{۳۰} قرار دارند که واحدهای تخصیص در سخت‌افزار دیسک می‌باشند. اندازه بلوک‌ها مضربی از اندازه سکتورها بوده و در xv6 اندازه این دو واحد با یکدیگر برابر ۵۱۲ بایت است. بلوک‌های فایل، توسط فراداده‌های آن به سکتورهای روی دیسک نگاشت می‌شوند. مثلاً بلوک یک فایل ممکن است در xv6 حاوی ۵۱۲ بایت داده ابتدایی فایل بوده و به یکی از سکتورهای دیسک تناظر داده می‌شود.

²⁵ Storage Devices

²⁶ Hard Disk Drive

²⁷ Metadata

²⁸ UNIX

²⁹ Superblock

³⁰ Sectors

در این بخش لازم است داده‌ساختارهای شرح داده شده را در xv6 بیاپید و فراخوانی سیستمی زیر را طراحی کنید که بعد از باز کردن فایل، توصیف‌کننده فایل^{۳۱} (FD) را به عنوان ورودی دریافت نموده و آدرس سکتورهای فایل را به سطح کاربر برگردانده و چاپ کند. سکتورهای داده فایل را از منظر توالی، با بلوک‌ها مقایسه نمایید. (راهنمایی: جهت انتقال آدرس‌ها از فراخوانی‌های سیستمی مشابه، الگو بگیرید).

```
void get_file_sectors(int fd, <parameter storing returned sector
addresses>);
```

به منظور آزمودن این فراخوانی سیستمی چندین فایل را به فایل سیستم xv6 اضافه کنید. سپس در برنامه کاربر خود فراخوانی سیستمی را فراخوانی نموده و آدرس سکتورها را دریافت نمایید. توجه داشته باشید هر فایل باید آن قدر بزرگ بوده که شامل چندین بلوک شود. (۷ توضیح دهید عدم توالی داده‌های یک فایل روی دیسک، چگونه ممکن است در فرایند بازیابی آن پس از حذف مشکل ساز شود. (امتیازی: ۵ نمره)

۲. پیاده‌سازی فراخوانی سیستمی گرفتن پردازنده پدر

در این بخش لازم است تا یک فراخوانی سیستمی طراحی کنید که PID پردازنده پدر پردازنده فعلی را بازگرداند.

```
int get_parent_pid();
```

³¹ File Descriptor

۳. پیاده‌سازی فراخوانی سیستمی تغییر پدر یک پردازش

در لینوکس فراخوانی سیستمی `ptrace()` ارائه راهکاری است که توسط آن یک پردازش می‌تواند اجرای پردازش دیگر را مشاهده و کنترل کرده و نیز به حافظه و ثبات‌های آن دسترسی پیدا کند. یکی از روش‌های پیاده‌سازی اشکال‌زدها^{۳۲} استفاده از این فراخوانی سیستمی است. اشکال‌زدها می‌تواند با فراخوانی `sys_ptrace()` والد یک پردازش شده تا از طریق سیگنال‌ها^{۳۳} بتواند اجرای آن را کنترل کرده و به حالت اجرایی آن دسترسی داشته باشد.

در این قسمت باید فراخوانی سیستمی زیر را طراحی کنید که با دریافت PID یک پردازش، پردازش فعلی را والد نموده و فرم ساده اشکال‌زدایی را (در حد تغییر پردازش والد برنامه تحت اشکال‌زدایی) شبیه‌سازی نماید. یعنی اگر پردازش A در سیستم فرزند پردازش B باشد، پردازش اشکال‌زدایی مانند D ایجاد شود. سپس با دریافت شناسه پردازش A، والد این پردازش شود در عین حال باید بتوان والد مورد انتظار پردازش را تشخیص داد تا اگر والد پردازش A در حالت اجرای عادی تأثیری در منطق برنامه دارد، در اجرای تحت اشکال‌زدها مشکلی ایجاد نشود.

```
set_process_parent(int pid);
```

یعنی باید هنگامی که D والد A شده، در صورت فراخوانی `get_parent_pid()` همچنان شناسه پردازش B برگردانده شود.

همچنین نشان دهید که تغییر والد هم به درستی صورت گرفته و D واقعاً والد A شده است. یک سناریوی پیشنهادی آزمون بخش مربوط به دید فرزند از والد (نه بخش اطمینان از والد سیستمی A که D است) به شرح زیر است:

³² Debuggers

³³ Signals

دو برنامه آزمون بنویسید:

برنامه نخست، معادل A بوده و دومی معادل D باشد. این دو باید همروند اجرا شوند. ابتدا A اجرا شده و PID خود و والد خود را چاپ می‌کند. سپس ابتدا فراخوانی سیستمی `sys_sleep()` اجرا شده و به میزان کافی تأخیر ایجاد شود. باید فراخوانی A در پوسته، با قرار دادن & پس از نام آن صورت گیرد تا در پس‌زمینه اجرا شده و امکان اجرای D در همان حین فراهم گردد.

D زمانی که A که در حال اجرای `sys_sleep()` است، اجرا می‌شود. PID چاپ شده توسط A را به عنوان ورودی دریافت نموده و والد آن می‌شود. پس از مدتی که A در پس‌زمینه بیدار شد، مجدداً PID والد خود را چاپ می‌کند. باید این مقدار، با وجود اتصال پردازش D که شبیه‌ساز اشکال‌زدا است، از دید A درست چاپ شود.

نتیجه را در گزارش کار خود ذکر کنید.

نکاتی در رابطه با فراخوانی‌های سیستمی

- برای این که بتوانید فراخوانی‌های سیستمی خود را تست کنید لازم است که یک برنامه سطح کاربر بنویسید و در آن فراخوانی‌ها را صدا بزنید. برای این که بتوانید برنامه سطح کاربر خود را درون Shell اجرا کنید، باید تغییرات مناسبی را روی Makefile انجام دهید تا برنامه جدید کامپایل شود و به فایل سیستم xv6 اضافه شود.
- برای ردیابی روال فراخوانی‌ها، پیغام‌های مناسبی در جاهای مناسب چاپ کنید.
- برای نمایش اطلاعات در سطح هسته از `cprintf()` استفاده کنید.

سایر نکات

- آدرس مخزن و شناسه آخرین تغییر خود را در محل بارگذاری در سایت درس، بارگذاری نمایید.
- تمام مراحل کار را در گزارش کار خود بیاورید.
- همه افراد باید به پروژه آپلود شده توسط گروه خود مسلط باشند و لزوماً نمره افراد یک گروه با یکدیگر برابر نیست.
- در صورت مشاهده هرگونه مشابهت بین کدها یا گزارش دو گروه، به هر دو گروه نمره ۰ تعلق می‌گیرد.
- فصل سه کتاب xv6 می‌توان کمک‌کننده باشد.
- هر گونه سوال در مورد پروژه را فقط از طریق فروم درس مطرح کنید.

موفق باشید

- [1] "System Call." [Online]. Available:
https://en.wikipedia.org/wiki/System_call.
- [2] L. Soares and M. Stumm, "FlexSC: Flexible System Call Scheduling with Exception-less System Calls," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 33–46.
- [3] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "A Study of Modern Linux API Usage and Compatibility: What to Support when You're Supporting," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, p. 16:1--16:16.
- [4] "Intel{®} 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide," 2015.
- [5] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, p. 19:1--19:17.