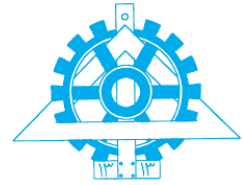




به نام خدا

آزمایشگاه سیستم عامل



پروژه دوم: فراخوانی سیستمی

تاریخ تحویل: ۲۲ آبان



KERNEL SPACE



USER SPACE

اهداف پروژه

- آشنایی با علت نیاز به فراخوانی سیستمی
- آشنایی با سازوکار و چگونگی صدا زده شدن فراخوانی‌های سیستمی¹ در هسته 6xv
- آشنایی با افزودن فراخوانی‌های سیستمی در هسته 6xv
- آشنایی با نحوه ذخیره‌سازی پردها و ساختار داده‌های مربوط به آن

¹ 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 استثنا^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 پیوسته مثل آرگومان‌ها را به صورت ایزوله‌شده از سطح کاربر انجام دهد. در این پروژه، تمرکز بر روی فراخوانی سیستمی است.

^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

در اکثریت قریب به اتفاق موارد، فراخوانی‌های سیستمی به طور غیرمستقیم و توسط توابع کتابخانه‌ای پوشاننده⁷ مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی glibc صورت می‌پذیرد.⁸ به این ترتیب قابلیت حمل⁹ 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 6xv 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 6xv 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، فری‌بی‌اس‌دی¹⁰ و ویندوز ۷ به ترتیب حدود ۳۰۰، ۵۰۰ و ۷۰۰ فراخوانی سیستمی وجود داشته که بسته به معماری پردازنده اندکی متفاوت خواهد بود [1]. در حالی که 6xv تنها ۲۱ فراخوانی سیستمی دارد.

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 نهان¹¹ و خط لوله¹² می‌باشد. به عنوان مثال، در یک فراخوانی سیستمی write() در لینوکس تا ۲ حافظه نهان سطح یک داده خالی خواهد شد [2]. به این ترتیب ممکن است کارایی به نصف کاهش یابد. غالباً عامل اصلی، سربار غیرمستقیم است. تعداد دستورالعمل اجرا شده به ازای هر سیکل¹³ (IPC) هنگام اجرای یک فراخوانی سیستمی در بار کاری 2006 SPEC CPU روی پردازنده 7Core i اینتل در نمودار زیر نشان داده شده است [2].

⁷ Wrapper

⁸ در glibc، توابع پوشاننده غالباً دقیقاً نام و پارامترهایی مشابه فراخوانی‌های سیستمی دارند.

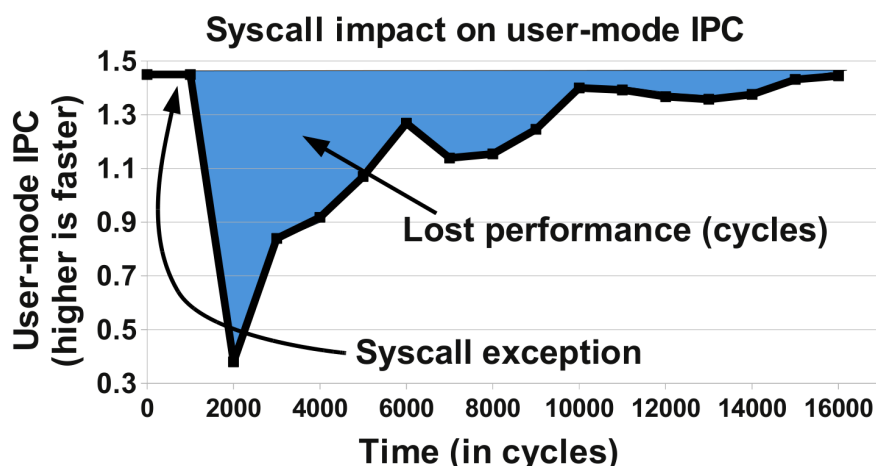
⁹ Portability

¹⁰ FreeBSD

¹¹ Caches

¹² Pipeline

¹³ Instruction per Cycle



مشاهده می‌شود که در لحظه‌ای IPC به کمتر از ۰.۴ رسیده است. روش‌های مختلفی برای فراخوانی سیستمی در پردازنده‌های 86x استفاده می‌گردد. روش قدیمی که در 6xv به کار می‌رود استفاده از دستور اسمبلی `int` است. مشکل اساسی این روش، سربار مستقیم آن است. در پردازنده‌های مدرن‌تر 86x 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 اشیای مجازی پویای مشترک (`vDSO`¹⁴) 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 6xv مرور خواهد شد.

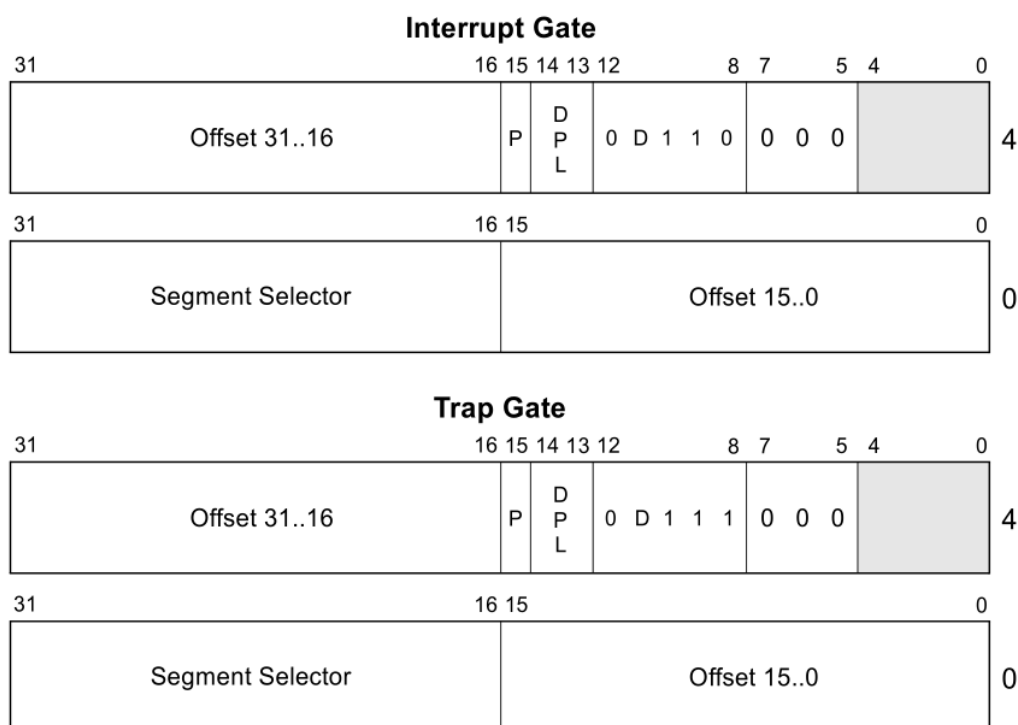
۲) دقت شود فراخوانی‌های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش‌ها را در لینوکس به اختصار توضیح دهید. می‌توانید از مرجع [3] کمک بگیرید.

¹⁴ Virtual Dynamic Shared Objects

سازوکار اجرای فراخوانی سیستمی در 6xv

بخش سخت‌افزاری و اسمبلی

جهت فراخوانی سیستمی در 6xv از روش قدیمی پردازنده‌های 86x 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 تنها از Trap Gate و Interrupt Gate استفاده می‌کند. ساختار این Gate ها در شکل زیر نشان داده شده است [4].



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 6xv 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 Gate ها 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 Gate ها را می‌توان با بررسی پارامترهای ماکروی مقداردهنده به Gate مربوط به فراخوانی سیستمی بررسی نمود:

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

پارامتر ۲: تعیین نوع Gate که در این‌جا `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.

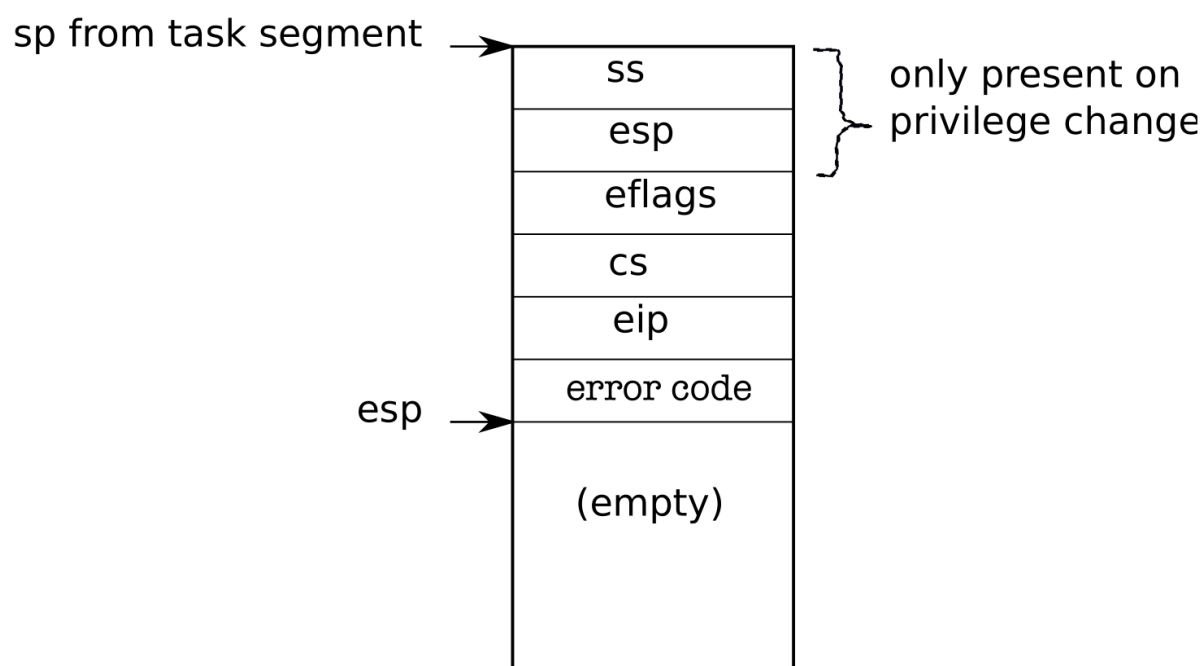
۳) آیا باقی تله‌ها را نمی‌توان با سطح دسترسی `DPL_USER` فعال نمود؟ چرا؟

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 `<trap no>int` می‌باشد. لذا با توجه به این که شماره تله فراخوانی سیستمی ۶۴ است (خط ۳۲۲۶)، کافی است برنامه، جهت فراخوانی فراخوانی سیستمی دستور `int 64` را فراخوانی کند. `int` یک

¹⁵ Trap Number

دستورالعمل پیچیده در پردازنده 86x (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 صفحه¹⁶ که در فصل مدیریت حافظه معرفی می‌گردد)، کد خطا نیز در انتها روی پشته قرار داده می‌شود. حالت پشته (سطح هسته¹⁷) 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

vector64:

pushl \$0

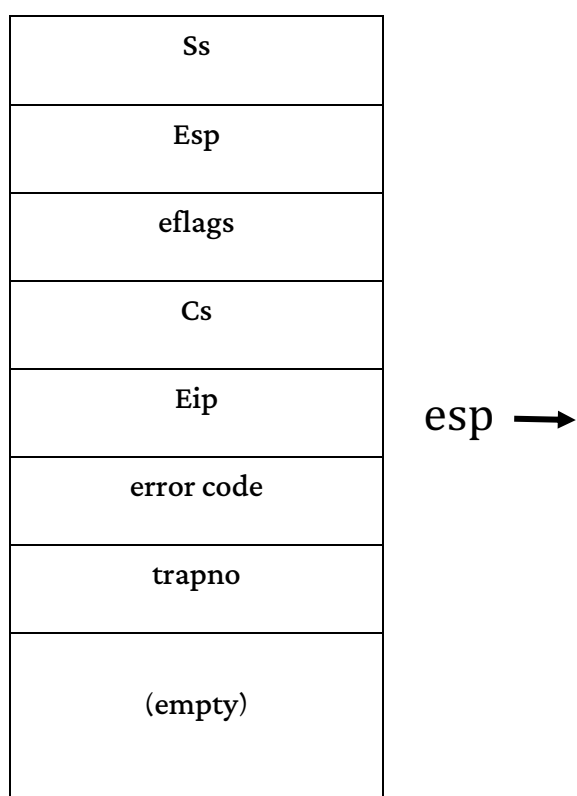
¹⁶ Page Fault

¹⁷ 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.

`pushl $64`

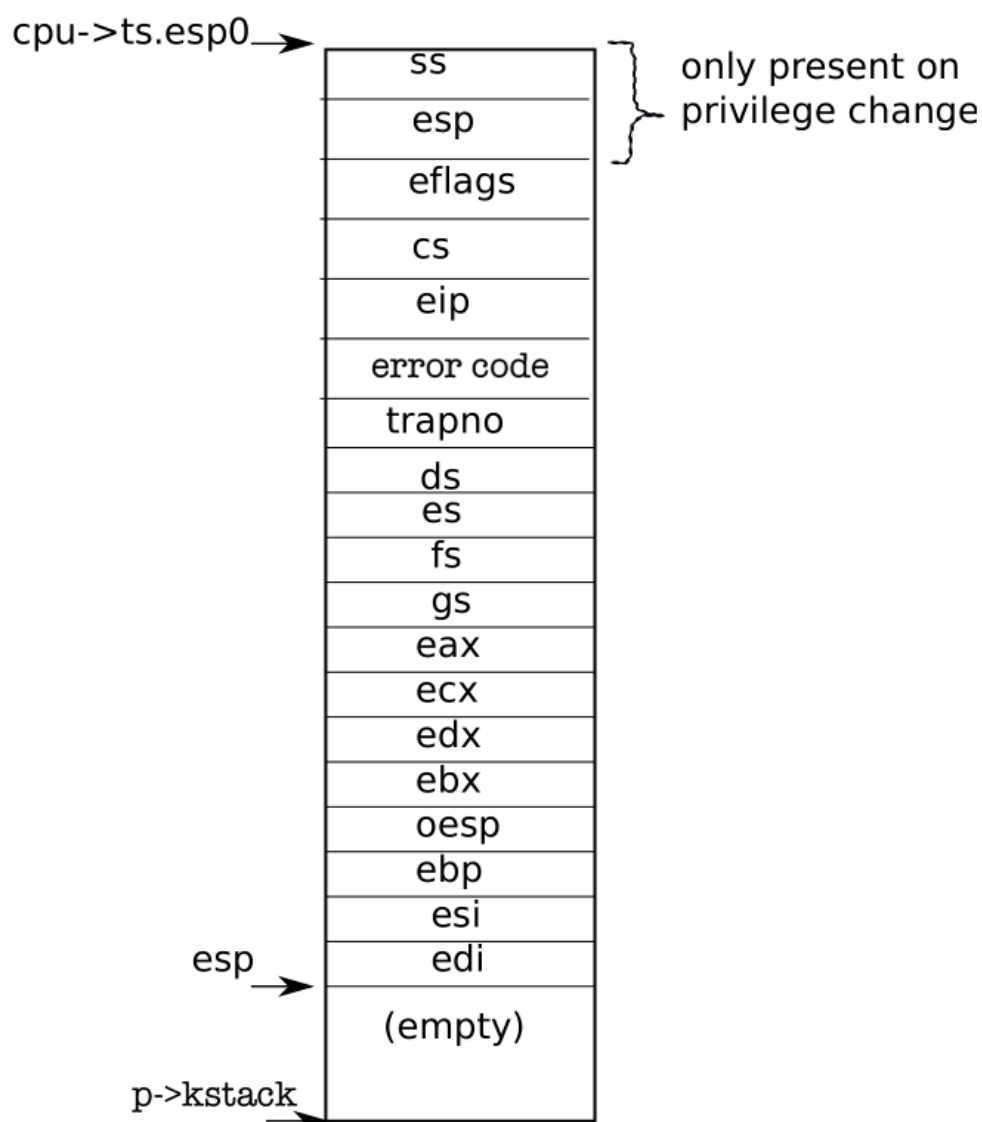
`jmp alltraps`

در این جا ابتدا یک کد خطای بی اثر 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 ۳۴۳). 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 پارامتری که 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 6xv 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 `$init` و `$argv` 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 6xv 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 فراخوانی‌های فراخوانی‌های سیستمی جزئی از واسطه باینری برنامه‌های کاربردی (¹⁸ABI) یک سیستم‌عامل روی یک معماری پردازنده است. به عنوان مثال در سیستم‌عامل لینوکس در معماری 86x، پارامترهای فراخوانی سیستمی به ترتیب در ثبات‌های `ebx`، `ecx`، `edx`، `esi`، `edi` و `ebp` قرار داده می‌شوند.¹⁹ 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 فراخوانی‌های فراخوانی سیستمی را می‌توان در فایل‌های زیر از کد منبع `glibc` مشاهده نمود.²⁰

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

¹⁸ Application Binary Interface

¹⁹ فرض این است که حداکثر شش پارامتر ارسال می‌گردد.

²⁰ مسیرها مربوط به `glibc-2.26` است.

به این ترتیب در لینوکس برخلاف 6xv پارامترهای فراخوانی سیستمی در ثبات منتقل می‌گردند. یعنی در لینوکس در سطح اسمبلی، ابتدا توابع پوشاننده پارامترها را در پشت‌پشته 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 کاربردی²¹ (API) وجود دارد که شامل مجموعه‌ای از تعاریف توابع (نه پیاده‌سازی) در سطح زبان برنامه‌نویسی بوده که واسط قابل حمل سیستم‌عامل²² (POSIX) نمونه‌ای از آن است. پشتیبانی توابع کتابخانه‌ای سیستم‌عامل‌ها از این تعاریف، قابلیت حمل برنامه‌ها را افزایش می‌دهد.²³ مثلاً امکان کامپایل یک برنامه روی لینوکس و iOS فراهم خواهد شد. جهت آشنایی بیشتر با POSIX و پیاده‌سازی آن در سیستم‌عامل‌های لینوکس، اندروید و iOS می‌توان به مرجع [5] مراجعه نمود.

بررسی گام‌های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

در این قسمت با توجه به توضیحاتی که تا الان داده شده‌است، قسمتی از روند اجرای یک سیستم‌کال را در سطح هسته بررسی خواهیم‌کرد. ابتدا یک برنامه ساده سطح کاربر بنویسید که بتوان از طریق آن، فراخوانی‌های سیستمی (`getpid()` در 6xv را اجرا کرد. یک نقطه توقف (breakpoint) در ابتدای تابع `syscall` قرار دهید. حال برنامه سطح کاربر نوشته‌شده را اجرا کنید. زمانی که به نقطه توقف برخورد کرد، دستور `bt` را در `gdb` اجرا کنید. توضیح کاربرد این دستور، تصویر خروجی آن و تحلیل کامل تصویر خروجی را در گزارش کار ثبت کنید.

حال دستور `down` (توضیح کارکرد این دستور را نیز در گزارش ذکر کنید) را در `gdb` اجرا کنید. محتوای رجیستر `eax` را که در `tf` می‌باشد، چاپ کنید. آیا مقداری که مشاهده می‌کنید، برابر با شماره فراخوانی سیستمی (`getpid()` می‌باشد؟ علت را در گزارش کار توضیح دهید.

²¹ Application Programming Interface

²² Portable Operating System Interface

²³ توابع پوشاننده فراخوانی‌های سیستمی بخشی از POSIX هستند.

چند بار دستور c را در gdb اجرا کنید تا در نهایت، محتوای رجیستر eax، شماره فراخوانی سیستمی (getpid()) را در خود داشته باشد.

دقت کنید می‌توانید در ابتدا دستور layout src را اجرا کنید تا کد c در ترمینال gdb نشان داده شود و شاید در تحلیل مراحل، کمکتان کند.

ارسال آرگومان‌های فراخوانی‌های سیستمی

تا این‌جا کار با نحوه ارسال آرگومان‌های فراخوانی‌های سیستمی در سیستم‌عامل 6xv 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.

● int find_largest_prime_factor(int n)

در این قسمت به جای بازیابی آرگومان‌ها به روش معمول، از ثبات‌ها استفاده می‌کنیم. در این فراخوانی، بزرگترین مقسوم علیه اول عدد n را محاسبه کنید. برای مثال در صورتی که عدد ورودی 276 باشد، شما باید عدد 23 را در خروجی چاپ کنید.

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

پیاده‌سازی فراخوانی‌های سیستمی

در این آزمایش با پیاده‌سازی فراخوانی‌های سیستمی، اضافه‌کردن آن‌ها به هسته 6xv را فرا می‌گیرید. در این فراخوانی‌ها که در ادامه توضیح داده می‌شود، پردازش‌هایی انجام می‌شود که از سطح کاربر قابل انجام نیست. 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 6xv تغییر دهید. برای این که با این فایل‌ها بیشتر آشنا شوید، پیاده‌سازی فراخوانی‌های سیستمی موجود را در 6xv 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 پردازش‌ها در هسته

پردازش‌ها در سیستم‌عامل 6xv پس از درخواست یک پردازش دیگر توسط هسته ساخته می‌شوند. در این صورت هسته نیاز دارد تا اولین پردازش را خودش اجرا کند. هسته 6xv برای نگه‌داری هر پردازش یک ساختار داده ساده دارد که در یک لیست مدیریت می‌شود. هر پردازش اطلاعاتی از قبیل شناسه واحد خود²⁴ که توسط آن شناخته می‌شود، پردازش والد و غیره را در ساختار خود دارد. برای ذخیره‌کردن اطلاعات بیشتر، می‌توان داده‌ها را به این ساختار داده اضافه کرد.

1. پیاده‌سازی فراخوانی سیستمی تغییر سائز فایل

در این قسمت فراخوانی سیستمی طراحی کنید که مسیر یک فایل و طول مورد نظر برای آن را به عنوان ورودی می‌گیرد و سپس فایل هدف را به اندازه طول خواسته شده می‌رساند.

● `void change_file_size(const char *path, int length)`

اگر طول فایل از طول نهایی داده شده کمتر باشد، فایل طولانی شده و قسمت اضافه با بایت نال ('0\') پر می‌شود.

²⁴ PID

اگر طول فایل از طول نهایی داده شده بیشتر باشد، طول فایل را با حذف از انتهای فایل کاهش دهید.

برای تست این فراخوانی سیستمی برنامه‌ی سطح کاربر بنویسید و فراخوانی سیستمی گفته شده را برای یک فایل مشخص ابتدا با طول کمتر از طول واقعی فایل و سپس با طول بیشتر از طول واقعی فایل فراخوانی کنید.

2. پیاده‌سازی فراخوانی سیستمی لیست پدازه‌های فراخواننده

در این قسمت فراخوانی سیستمی طراحی کنید که شماره یک فراخوانی سیستمی را به عنوان ورودی بگیرد و سپس شماره‌ی تمام پدازهایی که از آن فراخوانی سیستمی خاص استفاده کرده‌اند را چاپ کند. (شماره پدازه‌ها را با , از یکدیگر جدا کنید.)

- void get_callers(int syscall_number)

همچنین برنامه‌ای در سطح کاربر بنویسید و فراخوانی سیستمی گفته شده را با مقدارهای SYS_fork, SYS_write,

SYS_wait صدا بزنید و مقادیر به دست آمده را چاپ کنید.

توجه: خروجی حاصل از فراخوانی با مقدار SYS_fork را توجیه کنید.

3. پیاده‌سازی فراخوانی سیستمی گرفتن پدازه پدر

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

- int get_parent_pid(void)

در ادامه برنامه‌ی آزمونی برای این فراخوانی سیستمی بنویسید که در آن تا ۳ نسل پدازه به وجود آید. سپس در پدازه‌های نسل دوم و سوم با استفاده از فراخوانی سیستمی که در بخش قبل طراحی کردید، PID پدر را نمایش دهید.

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 6xv 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 6xv می‌توان کمک‌کننده باشد.
- هر گونه سوال در مورد پروژه را فقط از طریق فروم درس مطرح کنید.

موفق باشید

مراجع

[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.