

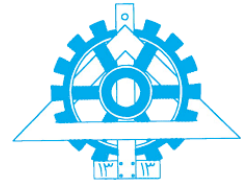


به نام خدا

آزمایشگاه سیستم عامل

پروژه دوم: فراخوانی سیستمی

تاریخ تحویل: یکشنبه 21 آبان 1402



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 xv6
- آشنایی با افزودن فراخوانی‌های سیستمی در هسته xv6
- آشنایی با نحوه ذخیره‌سازی پردازنده‌ها و ساختار داده‌های مربوط به آن

مقدمه

هر برنامه در حال اجرا یک پردازنده² نام دارد. به این ترتیب یک سیستم رایانه‌ای ممکن است در آن واحد، چندین پردازنده در انتظار سرویس داشته باشد. هنگامی که یک پردازنده در سیستم در حال

¹ System Call

² Process

اجرا است، پردازنده روال معمول پردازش را طی می‌کند: خواندن یک دستور، افزودن مقدار شمارنده برنامه³ 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:⁴

(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 وقفه⁵ خواهد بود. وقفه به طور غیرهمگام با کد در حال اجرا رخ می‌دهد.

(۲) ممکن است یک استثنا⁶ 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 سیستم عامل 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.

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

³ Program Counter

⁴ در xv6 به تمامی این موارد trap گفته می‌شود. در حالی که در حقیقت در x86 نام‌های متفاوتی برای این گذارها به کار می‌رود.

⁵ Interrupt

⁶ Exception

⁷ Wrapper

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

⁹ Portability

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 xv6 تنها ۲۱ فراخوانی سیستمی دارد.

فراخوانی سیستمی سربراهایی دارد: (۱) سربرار مستقیم که ناشی از تغییر مد اجرایی و انتقال به مد ممتاز بوده و (۲) سربرار 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 $\frac{2}{3}$ حافظه نهان سطح یک داده خالی خواهد شد [2]. به این ترتیب ممکن است کارایی به

نصف کاهش یابد. غالباً عامل اصلی، سربرار غیرمستقیم است. تعداد دستورالعمل اجرا شده به

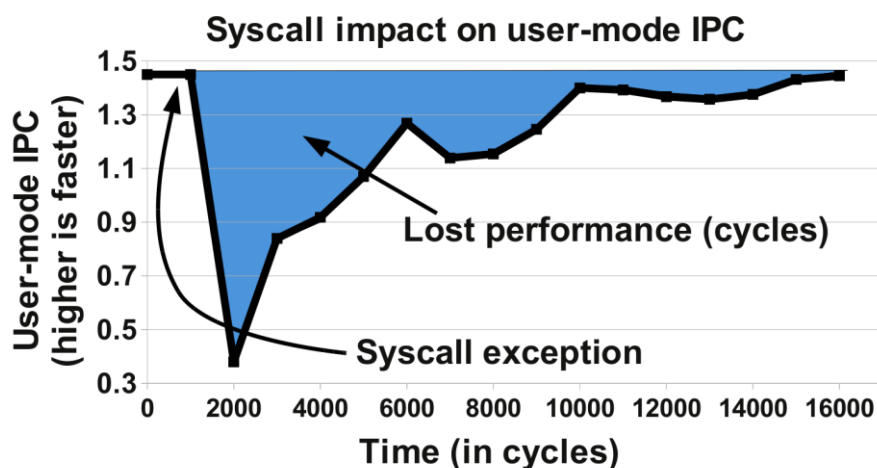
ازای هر سیکل¹³ (IPC) هنگام اجرای یک فراخوانی سیستمی در بار کاری 2006 SPEC CPU روی پردازنده Core i7 اینتل در نمودار زیر نشان داده شده است [2].

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 x86 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 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 اشیای مجازی پویای مشترک¹⁴ (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.

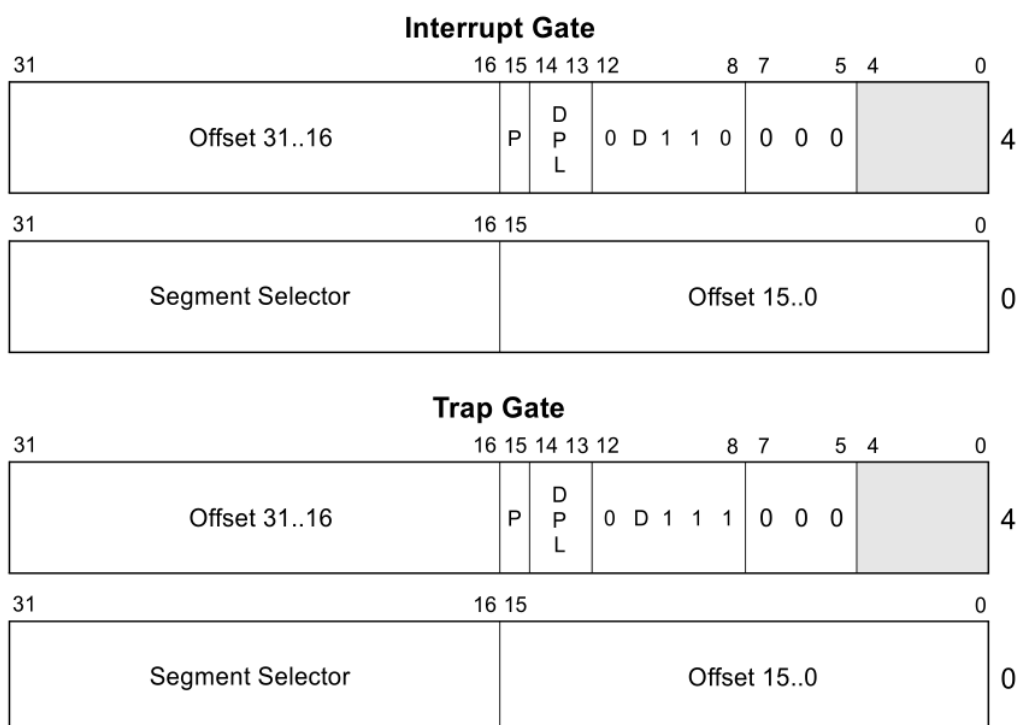
(۲) دقت شود فراخوانی‌های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش‌ها را در لینوکس به اختصار توضیح دهید. می‌توانید از مرجع [3] کمک بگیرید.

¹⁴ Virtual Dynamic Shared Objects

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 این‌جا 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 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
<div style="display: inline-block; width: 20px; height: 10px; background-color: #cccccc; border: 1px solid black;"></div>	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 تله¹⁵ 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، هر یک 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. 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.

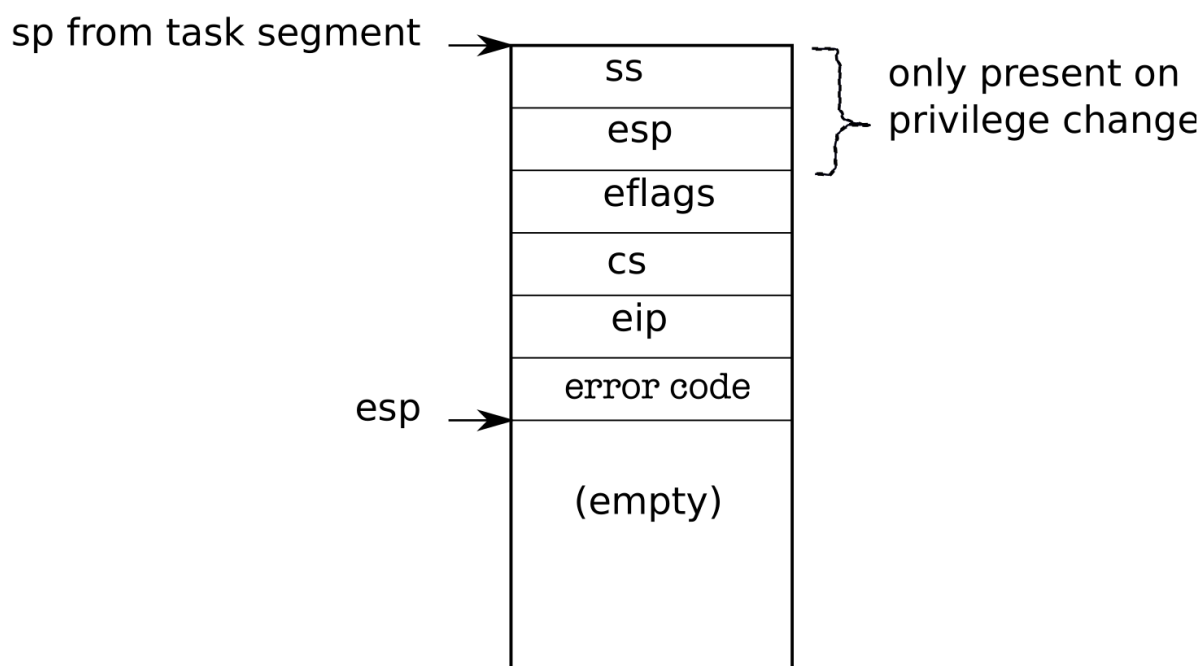
پارامتر ۵: سطح دسترسی مجاز برای اجرای این تله. `DPL_USER` است. زیرا فراخوانی سیستمی توسط (قطعه) کد سطح کاربر فراخوانی می‌گردد.

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.

¹⁵ Trap Number

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 64` 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.

^{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.

.globl vector64

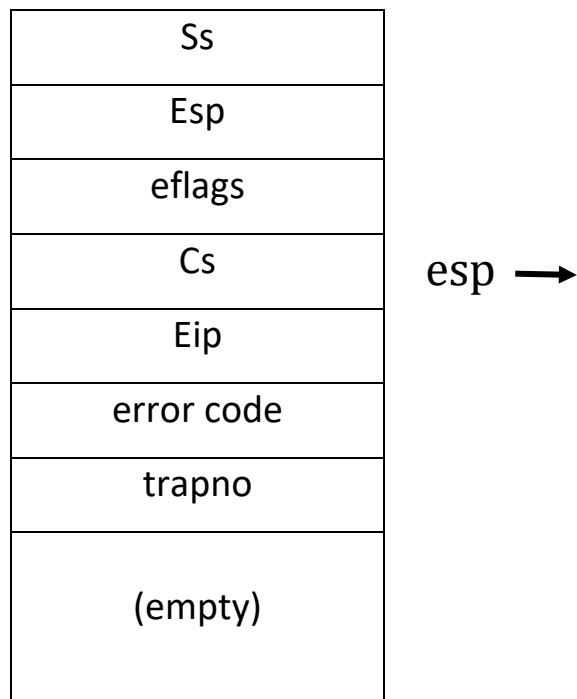
vector64:

pushl \$0

pushl \$64

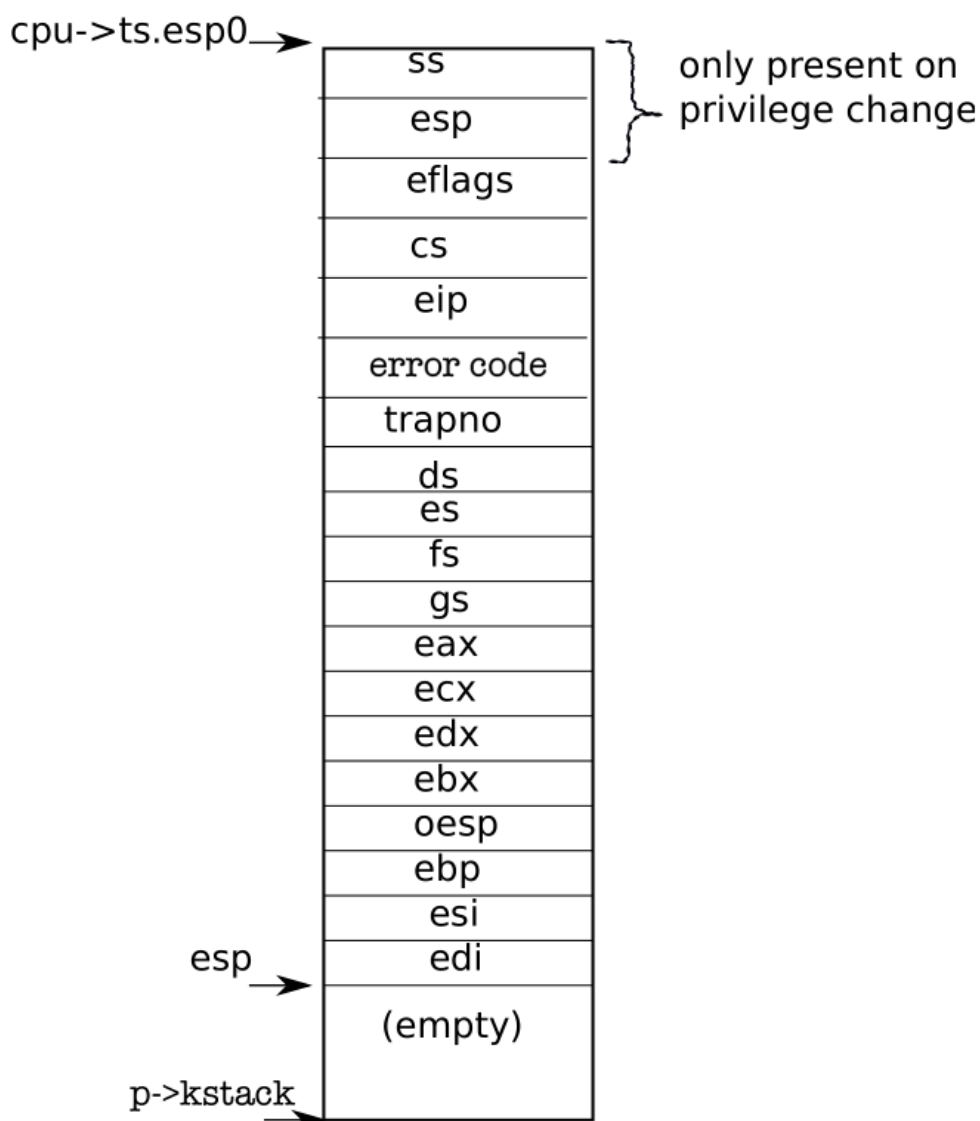
jmp alltraps

در اینجا 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 می یابد. حالت پشته، پیش از اجرای کد alltraps در شکل زیر نشان داده شده است.



alltraps باقی ثبات ها را Push می کند. 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 اشاره گر به بالای پشته (esp) که در اینجا اشاره گر به قاب تله است روی پشته قرار داده شده (خط ۳۳۱۸) و تابع trap() فراخوانی می شود. این معادل اسمبلی این است که اشاره گر 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 کنترل کننده زبان سی تله

تابع `trap` ابتدا نوع تله را با بررسی مقدار شماره تله چک می کند (خط ۳۴۰۳). با توجه به این که فراخوانی سیستمی رخ داده است تابع `syscall` اجرا می شود. پیش تر ذکر شد فراخوانی های سیستمی، متنوع بوده و هر یک دارای شماره ای منحصر به فرد است. این شماره ها در فایل `syscall.h` به فراخوانی های سیستمی نگاشت داده شده اند (خط ۳۵۰۰). تابع `syscall()` ابتدا وجود فراخوانی سیستمی فراخوانی شده را بررسی نموده و در صورت وجود پیاده سازی، آن را از جدول فراخوانی های سیستمی اجرا می کند. جدول فراخوانی های سیستمی، آرایه ای از اشاره گر ها

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 `8+esp` 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 واسطه باینری برنامه‌های کاربردی (`ABI`¹⁸) یک سیستم عامل روی یک معماری پردازنده است. به عنوان مثال در سیستم عامل لینوکس در معماری xv6، 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 فراخوانی فراخوانی‌های 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 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 (API²¹) وجود دارد که شامل مجموعه‌ای از تعاریف توابع (نه پیاده‌سازی) در سطح زبان برنامه‌نویسی بوده که واسط قابل‌حمل سیستم‌عامل (POSIX²²) نمونه‌ای از آن است. پشتیبانی توابع کتابخانه‌ای سیستم‌عامل‌ها از این تعاریف، قابلیت‌حمل 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 خواهیم کرد. ابتدا یک برنامه ساده سطح کاربر بنویسید که بتوان از طریق آن، فراخوانی‌های سیستمی `getpid()` در xv6 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،

²¹ Application Programming Interface

²² Portable Operating System Interface

²³ توابع پوشاننده فراخوانی‌های سیستمی بخشی از POSIX هستند.

برابر با شماره فراخوانی سیستمی `getpid()` می‌باشد؟ علت را در گزارش کار توضیح دهید.

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

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

ارسال آرگومان‌های فراخوانی‌های سیستمی

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

● `int find_digital_root(int n)`

در این قسمت به جای بازیابی آرگومان‌ها به روش معمول، از ثبات‌ها استفاده می‌کنیم. در این فراخوانی، ریشه دیجیتالی عدد ورودی را محاسبه کنید. برای مثال در صورتی که عدد ورودی 284 باشد، شما باید عدد 5 را در خروجی چاپ کنید.

عدد `n` باید مثبت باشد و در غیر این صورت فراخوانی سیستمی عددی منفی برگردانده و برنامه سطح کاربر این را به کاربر اطلاع می‌دهد.

توضیح: ریشه دیجیتالی یک عدد، حاصل جمع مکرر ارقام آن عدد است تا اینکه حاصل به یک تک رقم برسد. در مثال گفته شده، در ابتدا $2+8+4=14$ می‌شود و سپس $1+4=5$ ریشه دیجیتالی عدد است.

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 با پیاده‌سازی فراخوانی‌های سیستمی، اضافه‌کردن آن‌ها به هسته xv6 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 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 پردازش‌ها در هسته

پردازش‌ها در سیستم عامل xv6 پس از درخواست یک پردازش دیگر توسط هسته ساخته می‌شوند. در این صورت هسته نیاز دارد تا اولین پردازش را خودش اجرا کند. هسته xv6 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 ذخیره کردن اطلاعات بیشتر، می‌توان داده‌ها را به این ساختار داده اضافه کرد.

1. پیاده‌سازی فراخوانی سیستمی کپی کردن فایل

در این قسمت فراخوانی سیستمی طراحی کنید که دو نام می‌گیرد و فایل با نام اول را کپی کرده و با نام دوم ذخیره می‌کند.

- `int copy_file(const char* src, const char* dest)`

²⁴ PID

برای تست این فراخوانی سیستمی، یک برنامه‌ی سطح کاربر بنویسید و فراخوانی سیستمی گفته شده را فراخوانی کنید و نتیجه را نشان دهید. عمل کپی کردن باید به طور کامل در کرنل و نه در برنامه سطح کاربر انجام شود.

برای تولید فایل، می‌توانید از دستور echo استفاده کنید و یا فایل README را کپی کنید. این فراخوانی سیستمی در صورت موفقیت 0 و در غیر این صورت 1- ریترن می‌کند. در صورت خطا در برنامه سطح کاربر به کاربر اطلاع داده می‌شود.

توجه کنید که یک فایل نباید به خودش کپی شود و در این صورت 1- ریترن می‌شود. اگر نام دوم داده شده از قبل وجود داشته باشد، می‌توانید به دلخواه یا خطا گرفته و یا آن را override کنید.

2. پیاده‌سازی فراخوانی سیستمی تعداد uncle های پردازش

در این قسمت، فراخوانی سیستمی را طراحی کنید که تعداد uncle های یک پردازش را برگرداند. منظور از uncle یک پردازش، sibling های پردازش پدر آن پردازش می‌باشد.

● int get_uncle_count(int)

برای تست، برنامه‌ای در سطح کاربر بنویسید که با استفاده از fork() سه فرزند ساخته شود و سپس برای یک کدام از آن‌ها یک فرزند ایجاد شود. در نهایت get_uncle_count() را برای آن فراخوانی کنید و خروجی را نمایش دهید.

3. پیاده‌سازی فراخوانی سیستمی طول عمر پردازش

در این قسمت، مدت زمان زندگی یک پردازش از زمان به وجود آمدن تا زمان صدا کردن این فراخوانی سیستمی، محاسبه شود.

● int get_process_lifetime(int)

برای تست این فراخوانی سیستمی، برنامه‌ای در سطح کاربر بنویسید که با استفاده از fork() یک پردازش فرزند ایجاد شود و پس از 10 ثانیه از بین برود. طول عمر پردازش فرزند و 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 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.