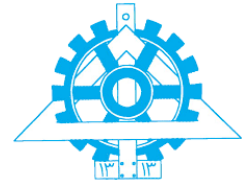




به نام خدا

آزمایشگاه سیستم عامل



پروژه دوم: فراخوانی سیستمی

تاریخ تحویل: ۲۰ فروردین



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 پردها و ساختار داده‌های مربوط به آن

¹ System Call

مقدمه

هر برنامه در حال اجرا یک پردازنده^۲ 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 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 غیر همگام 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 سیستم عامل باید به گونه‌ای باشد که مواردی از قبیل ذخیره‌سازی اطلاعات پردازنده و بازیابی اطلاعات رویداد به وقوع پیوسته مثل آرگومان‌ها را به صورت ایزوله شده 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، فراخوانی های سیستمی به طور غیرمستقیم و توسط توابع کتابخانه ای پوشاننده⁷ مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی glibc صورت می پذیرد.⁸ به این ترتیب قابلیت حمل⁹ برنامه های سطح کاربر افزایش می یابد. زیرا به عنوان مثال چنانچه در ادامه مشاهده خواهد شد، فراخوانی های سیستمی با شماره هایی مشخص می شوند که در معماری های مختلف، متفاوت است. توابع پوشاننده کتابخانه ای، این وابستگی ها را مدیریت می کنند. توابع پوشاننده xv6 در فایل usys.S توسط ماکروی SYSCALL تعریف شده اند.

(۱) کتابخانه های (قاعدتاً سطح کاربر، منظور فایل های تشکیل دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانی های سیستمی و علت این استفاده بررسی نمایید.

تعداد فراخوانی های سیستمی، وابسته به سیستم عامل و حتی معماری پردازنده است. به عنوان مثال در لینوکس، فری بی اس دی¹⁰ 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 سربارهایی دارد: (۱) سربار مستقیم که ناشی از تغییر مد اجرایی و انتقال به مد ممتاز بوده و (۲) سربار غیر مستقیم که ناشی از آلودگی ساختارهای پردازنده شامل انواع حافظه های نهان¹¹ و خط لوله¹² می باشد. 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، سربار غیر مستقیم است. تعداد دستورالعمل اجرا شده به ازای هر سیکل¹³ (IPC) هنگام اجرای یک فراخوانی سیستمی در بار کاری SPEC CPU 2006 روی پردازنده Core i7 اینتل در نمودار زیر نشان داده شده است [2].

⁷ Wrapper

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

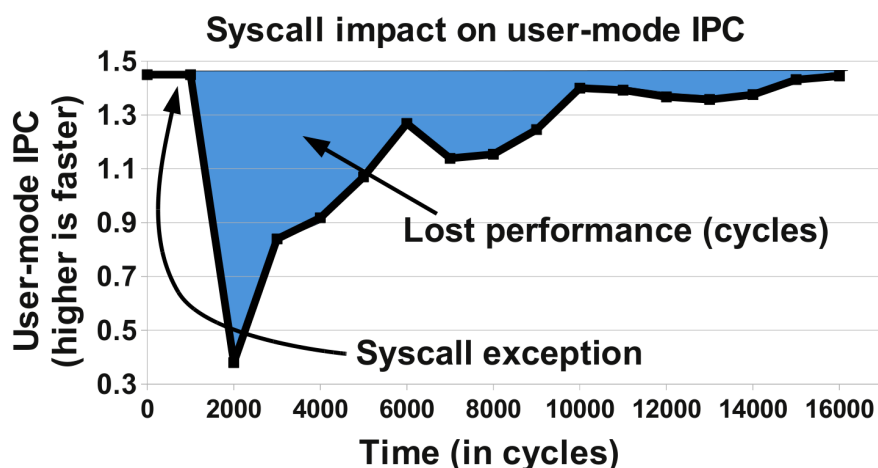
⁹ Portability

¹⁰ FreeBSD

¹¹ Caches

¹² Pipeline

¹³ Instruction per Cycle



مشاهده می شود که در لحظه ای IPC به کمتر از ۰.۴ رسیده است. روش های مختلفی برای فراخوانی سیستمی در پردازنده های x86 استفاده می گردد. روش قدیمی که در xv6 به کار می رود استفاده از دستور اسمبلی `int` است. مشکل اساسی این روش، سربار مستقیم آن است. در پردازنده های مدرن تر 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 می کند. برخی فراخوانی های سیستمی (مانند `gettimeofday()` در لینوکس) فرکانس دسترسی بالا و پردازش کمی در هسته دارند. لذا سربار مستقیم آن ها بر برنامه زیاد خواهد بود. در این موارد می توان از روش های دیگری مانند اشیای مجازی پویای مشترک ¹⁴ (`vDSO`) در لینوکس بهره برد. به این ترتیب که هسته، پیاده سازی فراخوانی های سیستمی را در فضای آدرس سطح کاربر نگاشت داده و تغییر مد به مد هسته صورت نمی پذیرد. این دسترسی نیز به طور غیرمستقیم و توسط کتابخانه `glibc` صورت می پذیرد. در ادامه سازوکار اجرای فراخوانی سیستمی در xv6 مرور خواهد شد.

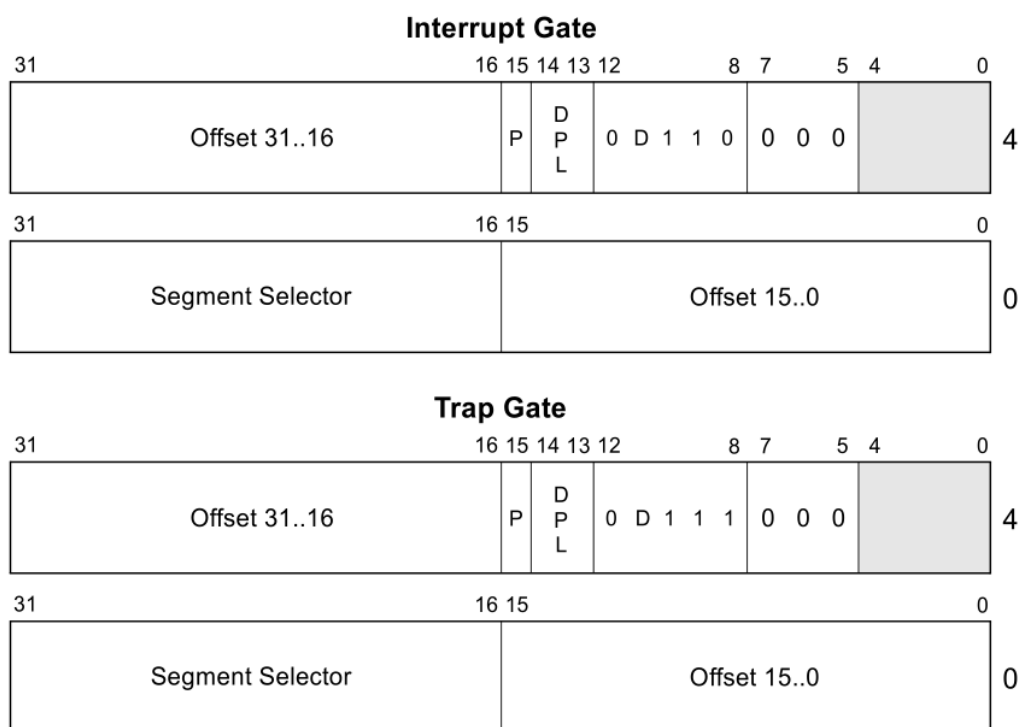
(۲) دقت شود فراخوانی های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش ها را در لینوکس به اختصار توضیح دهید. می توانید از مرجع [3] کمک بگیرید.

¹⁴ Virtual Dynamic Shared Objects

سازوکار اجرای فراخوانی سیستمی در xv6

بخش سخت افزاری و اسمبلی

جهت فراخوانی سیستمی در xv6 از روش قدیمی پردازنده های x86 استفاده می شود. در این روش، دسترسی به کد دارای سطح دسترسی ممتاز (در این جا کد هسته) مبتنی بر مجموعه توصیف گرایبی موسوم به Gate Descriptor است. چهار نوع Gate Descriptor وجود دارد که xv6 تنها از Trap Gate و Interrupt Gate استفاده می کند. 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 xv6 در قالب یک ساختار هشت بایتی موسوم به struct gatedesc تعریف شده اند (خط ۸۵۵). به ازای هر انتقال به هسته (فراخوانی سیستمی و هر یک از انواع وقفه های سخت افزاری و استثناها) یک Gate در حافظه تعریف شده و

یک شماره تله¹⁵ نسبت داده می شود. این Gate ها توسط تابع `tvinit()` در حین بوت (خط ۱۲۲۹) مقداردهی می گردند. Interrupt Gate اجازه وقوع وقفه در پردازنده حین کنترل وقفه را نمی دهد. در حالی که Trap Gate این گونه نیست. لذا برای فراخوانی سیستمی از Trap Gate استفاده می شود تا وقفه که اولویت بیشتری دارد، همواره قابل سرویس دهی باشد (خط ۳۳۷۳). عملکرد Gate ها را می توان با بررسی پارامترهای ماکروی مقدار دهنده به Gate مربوط به فراخوانی سیستمی بررسی نمود:

پارامتر ۱: `T_SYSCALL[idt]` محتوای Gate مربوط به فراخوانی سیستمی را نگه می دارد. آرایه `idt` (خط ۳۳۶۱) بر اساس شماره تله ها اندیس گذاری شده است. پارامترهای بعدی، هر یک بخشی از `T_SYSCALL[idt]` را پر می کنند.

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 می گردد. کد کنترل کننده 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 تله ها شاخص گذاری شده است.

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

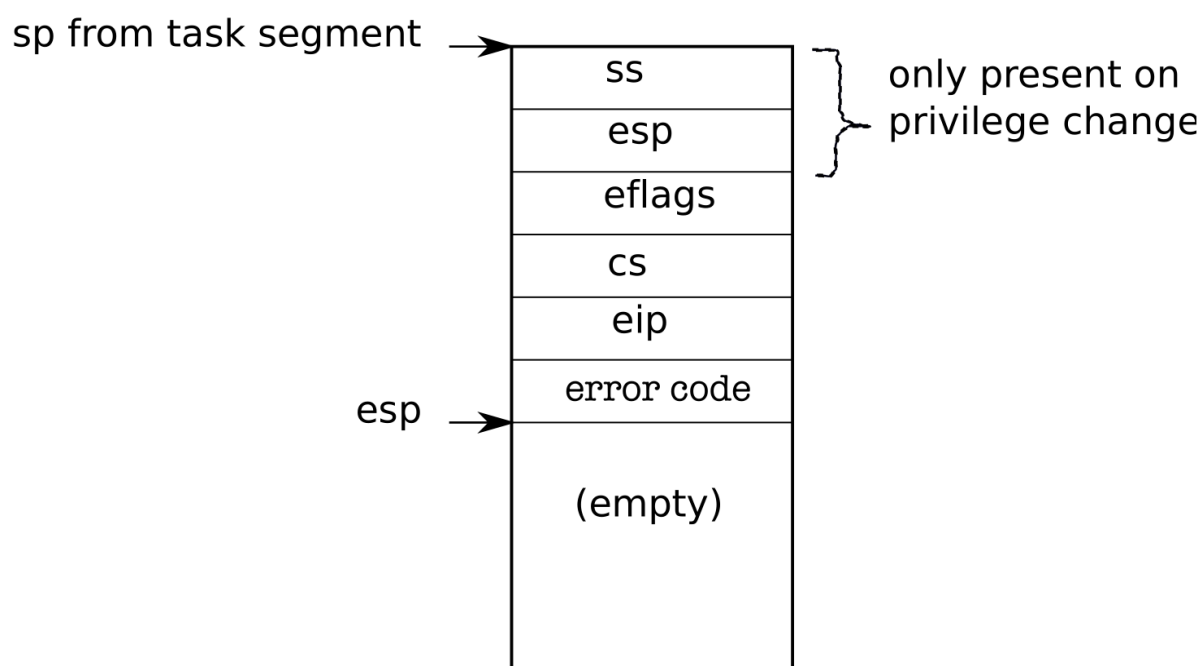
۳) آیا باقی تله ها را نمی توان با سطح دسترسی `DPL_USER` فعال نمود؟ چرا؟

به این ترتیب برای تمامی تله ها `idt` مربوطه ایجاد می گردد. 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 راه اندازی اولیه هر هسته پردازنده، اجرا شده و اشاره گر به جدول `idt` را در ثبات مربوطه در هر هسته بارگذاری می نماید. از این به بعد امکان سرویس دهی به تله ها فراهم است. یعنی پردازنده می داند برای هر تله چه کدی را فراخوانی کند.

یکی از راه های فعال سازی هر تله استفاده از دستور `<trap no < int` می باشد. 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`

¹⁵ Trap Number

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 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

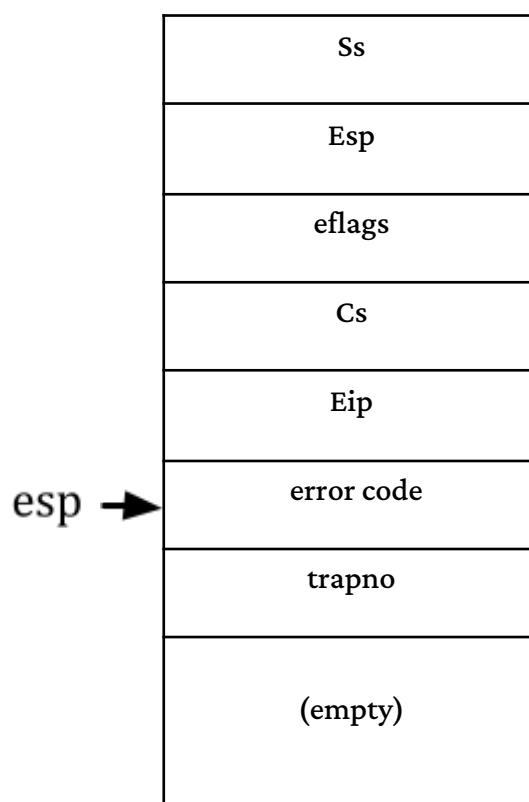
¹⁶ Page Fault

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.

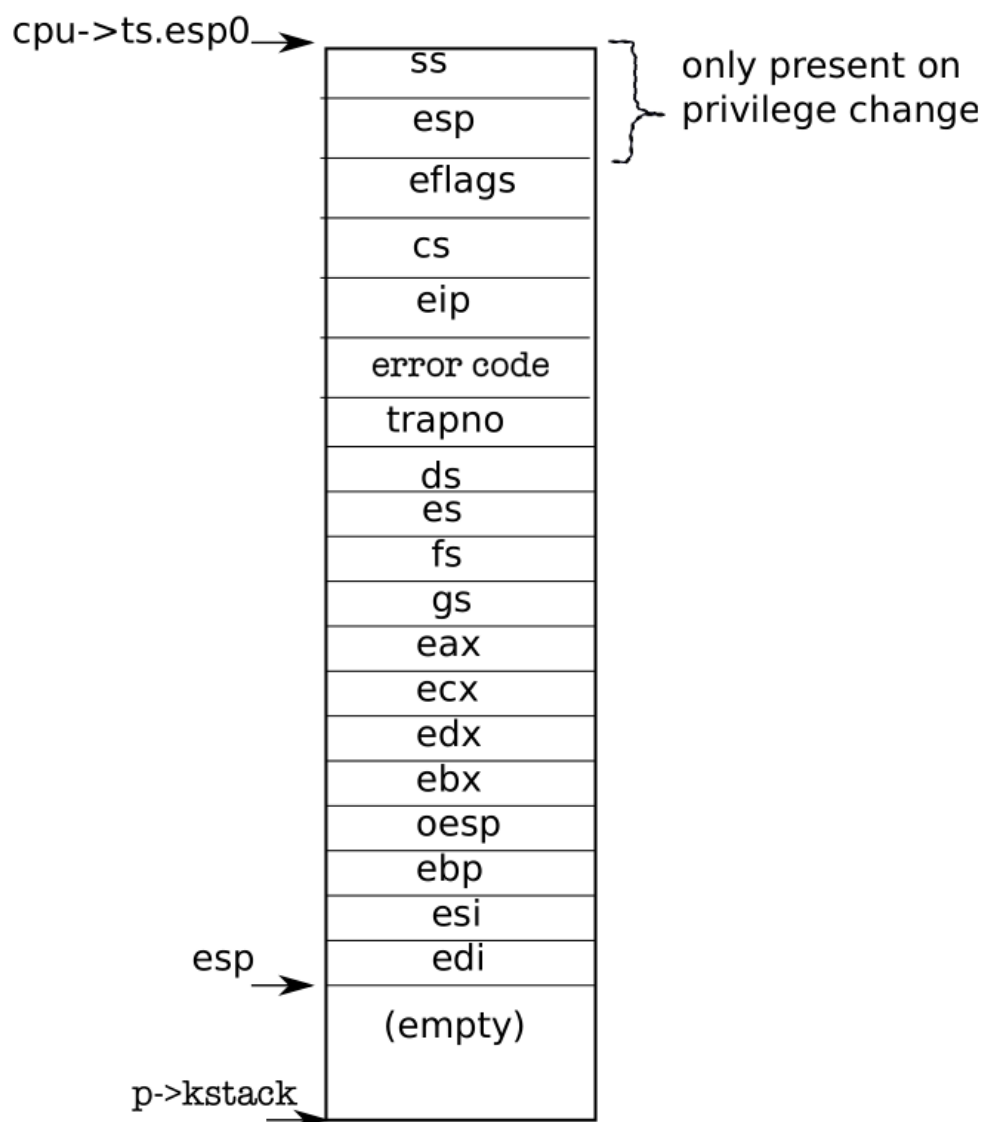
`pushl $64`

`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 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 این جا 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

$esp+8$ 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()` می تواند 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 می گردد.

(۵) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در `argptr()` بازه آدرس ها بررسی می گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد می کند؟ در صورت عدم بررسی بازه ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی `sys_read()` اجرای سیستم را با مشکل روبرو سازد.

شیوه فراخوانی های فراخوانی سیستمی جزئی از واسطه باینری برنامه های کاربردی¹⁸ (ABI) یک سیستم عامل روی یک معماری پردازنده است. به عنوان مثال در سیستم عامل لینوکس در معماری `x86`، پارامترهای فراخوانی سیستمی به ترتیب در ثبات های `ebx`، `ecx`، `edx`، `esi`، `edi` و `ebp` قرار داده می شوند.¹⁹ ضمن این که طبق این ABI، نباید مقادیر ثبات های `ebx`، `esi`، `edi` و `ebp` پس از فراخوانی تغییر کنند. لذا باید مقادیر این ثبات ها 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 های سیستمی را می توان در فایل های زیر از کد منبع `glibc` مشاهده نمود.²⁰

`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 می گردند. 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 ثبات ها، در آن ها کپی می گردند.

در هنگام تحویل سوالاتی از سازوکار فراخوانی سیستمی پرسیده می شود. دقت شود در مقابل ABI، مفهومی تحت عنوان واسطه برنامه نویسی برنامه کاربردی²¹ (API) وجود دارد که شامل مجموعه ای از تعاریف توابع (نه پیاده سازی) در سطح زبان

¹⁸ Application Binary Interface

¹⁹ فرض این است که حداکثر شش پارامتر ارسال می گردد.

²⁰ مسیرها مربوط به `glibc-2.26` است.

²¹ Application Programming Interface

برنامه نویسی بوده که واسط قابل حمل سیستم عامل²² (POSIX) نمونه ای از آن است. پشتیبانی توابع کتابخانه ای سیستم عامل ها از این تعاریف، قابلیت حمل برنامه ها را افزایش می دهد.²³ مثلاً امکان کامپایل یک برنامه روی لینوکس و iOS فراهم خواهد شد. جهت آشنایی بیشتر با POSIX و پیاده سازی آن در سیستم عامل های لینوکس، اندروید و iOS می توان به مرجع [5] مراجعه نمود.

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

در این قسمت با توجه به توضیحاتی که تا الان داده شده است، قسمتی از روند اجرای یک سیستم کال را در سطح هسته بررسی خواهیم کرد. ابتدا یک برنامه ساده سطح کاربر بنویسید که بتوان از طریق آن، فراخوانی های سیستمی (`getpid()` در `xv6` را اجرا کرد. یک نقطه توقف (`breakpoint`) در ابتدای تابع `syscall` قرار دهید. حال برنامه سطح کاربر نوشته شده 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` می باشد، چاپ کنید. آیا مقداری که مشاهده می کنید، برابر با شماره فراخوانی سیستمی (`getpid()` می باشد؟ 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 داشته باشد.

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

²² Portable Operating System Interface

²³ توابع پوشاننده فراخوانی های سیستمی بخشی از POSIX هستند.

ارسال آرگومان‌های فراخوانی‌های سیستمی

تا این‌جا 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 را که در آن تنها یک آرگومان ورودی از نوع `int` وجود دارد پیاده‌سازی کنید.

● `int find_fibonacci_number(int n)`

در این قسمت به جای بازیابی آرگومان‌ها به روش معمول، از ثبات‌ها استفاده می‌کنیم. در این فراخوانی، `n` امین عدد در دنباله فیبوناچی را محاسبه کنید. برای مثال در صورتی که عدد ورودی 10 باشد، شما باید عدد 34 را در خروجی چاپ کنید. یادآوری:

Fibonacci series = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

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 `user.h`، `syscall.h`، `syscall.c` 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 پرده‌ها در هسته

پرده‌ها در سیستم‌عامل xv6 پس از درخواست یک پرده دیگر توسط هسته ساخته می‌شوند. در این صورت هسته نیاز دارد تا اولین پرده را خودش اجرا کند. هسته xv6 برای نگهداری هر پرده یک ساختار داده ساده دارد که در یک لیست مدیریت می‌شود. هر پرده اطلاعاتی از قبیل شناسه واحد خود²⁴ که توسط آن شناخته می‌شود، پرده 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 پر استفاده‌ترین فراخوانی سیستمی

در این قسمت فراخوانی سیستمی طراحی کنید که شماره فراخوانی سیستمی‌ای که بیشتر از سایر فراخوانی‌های سیستمی استفاده شده است را برگرداند.

● `int find_most_callee(void)`

برای تست این فراخوانی سیستمی برنامه‌ی سطح کاربر بنویسید و فراخوانی سیستمی گفته شده را فراخوانی کنید و نتیجه را چاپ کنید.

توجه: خروجی حاصل را از این فراخوانی را در گزارش کار توجیه کنید.

²⁴ PID

2. پیاده‌سازی فراخوانی سیستمی تعداد فرزندان پردازش کنونی

در این قسمت، فراخوانی سیستمی را طراحی کنید که تعداد فرزندان زنده پردازش کنونی را برگرداند.

- `int get_alive_children_count(void)`

برای تست، برنامه‌ای در سطح کاربر بنویسید که با استفاده از `fork()` سه پردازش فرزند ایجاد کنید و در پردازش پدر

`get_alive_children_count()` را فراخوانی کنید.

3. پیاده‌سازی فراخوانی سیستمی کشتن اولین فرزند پردازش کنونی

در این قسمت، اولین پردازش فرزند پردازش کنونی را بکشید.

- `int kill_first_child_process(void)`

برای تست این فراخوانی سیستمی، برنامه‌ای در سطح کاربر بنویسید که با استفاده از `fork()` یک پردازش فرزند ایجاد کنید و

قبل از از بین رفتن پردازش فرزند (که از `sleep()` می‌توانید کمک بگیرید) پردازش پدر، فرزند مذکور را کشته و قبل و بعد کشتن

با استفاده از فراخوانی سیستمی قسمت قبل تعداد فرزندان زنده چاپ شود.

توجه: خروجی حاصل را از این فراخوانی را در گزارش کار توجیه کنید و درباره نحوه ایجاد خروجی این تست در گزارش

کار بنویسید.

نکاتی در رابطه با فراخوانی های سیستمی

- برای این که بتوانید فراخوانی های سیستمی خود را تست کنید لازم است که یک برنامه سطح کاربر بنویسید و در آن فراخوانی ها 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 فراخوانی ها، پیغام های 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 شباهت بین کدها یا گزارش دو گروه، به هر دو گروه نمره ۰ تعلق می گیرد.
- فصل سه کتاب xv6 می توان کمک کننده 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.