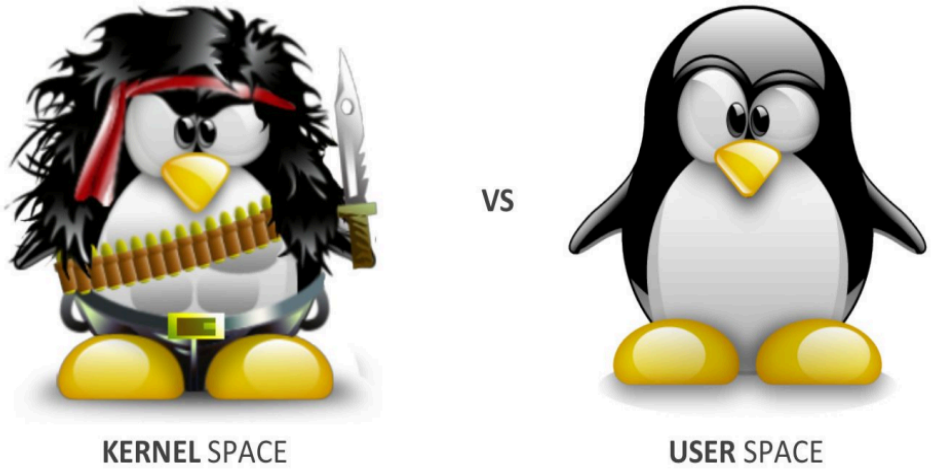
به نام خدا

آزمایشگاه سیستم عامل - پاییز ۱۴۰۳

پروژه دوم: فراخوانی سیستمی

طراحان: گلبو رشیدی، مهدی نوری



اهداف پروژه

- آشنایی با سازوکار و چگونگی صدا زده شدن فراخوانی‌های سیستمی[1] در هسته xv6
- آشنایی با پیاده‌سازی تعدادی فراخوانی سیستمی در هسته xv6
- ذخیره سازی اطلاعات فراخوانی‌های سیستمی
- آشنایی با نحوه ذخیره‌سازی پردازه‌ها و ساختارداده‌های مربوط به آن

[1] System Call

مقدمه

هر برنامه در حال اجرا یک پردازه[2] نام دارد. به این ترتیب یک سیستم رایانه‌ای ممکن است در آنِ واحد، چندین پردازه در انتظار سرویس داشته باشد. هنگامی که یک پردازه در سیستم در حال اجرا است، پردازنده روال معمول پردازش را طی میکند: خواندن یک دستور، افزودن مقدار شمارنده برنامه[3] 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]

1) ممکن است داده‌ای از دیسک دریافت شده باشد و به دلایلی لازم باشد بلافاصله آن داده از ثبات[5] مربوطه در دیسک به حافظه منتقل گردد. انتقال جریان کنترل در این حالت، ناشی از وقفه[6] خواهد بود. وقفه به طور غیر همگام با کد در حال اجرا رخ میدهد.
2) ممکن است یک استثنا[7] مانند تقسیم بر صفر رخ دهد. در اینجا برنامه دارای یک دستور تقسیم بوده که عملوند مخرج آن مقدار صفر داشته و اجرای آن کنترل را به هسته میدهد.
3) ممکن است برنامه نیاز به عملیات ممتاز داشته باشد. عملیاتی مانند دسترسی به اجزای سخت افزاری 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 پردازه و بازیابی اطلاعات رویدادِ به وقوع پیوسته مثل آرگومان‌ها را به صورت ایزوله‌شده از سطح کاربر انجام دهد. در این پروژه، تمرکز بر روی فراخوانی سیستمی است.

---

[2] Process

[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] Register

[6] Interrupt

[7] Exception

در اکثریت قریب به اتفاق موارد، فراخوانی‌های سیستمی به طور غیرمستقیم و توسط توابع کتابخانه‌ای پوشاننده[8] مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی glibc صورت می‌پذیرد.[9] به این ترتیب قابلیت‌حمل[10] 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.

پرسش 1: کتابخانه‌های سطح کاربر در xv6، برای ایجاد ارتباط میان برنامه‌های کاربر و کرنل به کار می‌روند. این کتابخانه‌ها شامل توابعی هستند که از فراخوانی‌های سیستمی استفاده می‌کنند تا دسترسی به منابع سخت‌افزاری و نرم‌افزاری سیستم‌عامل ممکن شود. با تحلیل فایل‌های موجود در متغیر ULIB در xv6، توضیح دهید که چگونه این کتابخانه‌ها از فراخوانی‌های سیستمی بهره می‌برند؟ همچنین، دلایل استفاده از این فراخوانی‌ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه‌ها را شرح دهید.

تعداد فراخوانی‌های سیستمی، وابسته به سیستم‌عامل و حتی معماری پردازنده است. به عنوان مثال در لینوکس، فری‌بی‌اس‌دی[11] و ویندوز 7 به ترتیب حدود ،300، 500 و 700 فراخوانی سیستمی وجود داشته که بسته به معماری پردازنده اندکی متفاوت خواهد بود [1]. در حالی که xv6 تنها 21 فراخوانی سیستمی دارد.

فراخوانی سیستمی سربارهایی دارد: 1) سربار مستقیم که ناشی از تغییر مد اجرایی و انتقال به مد ممتاز بوده و 2) سربار غیر مستقیم که ناشی از آلودگی ساختارهای پردازنده شامل انواع حافظه‌های نهان[12] و خط لوله[13] 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، سربار غیر مستقیم است. تعداد دستورالعمل اجرا شده به ازای هر سیکل[14]

[8] Wrapper

[9] در glibc، توابع پوشاننده غالبا نام و پارامترهایی مشابه فراخوانی‌های سیستمی دارند.

[10] Portability

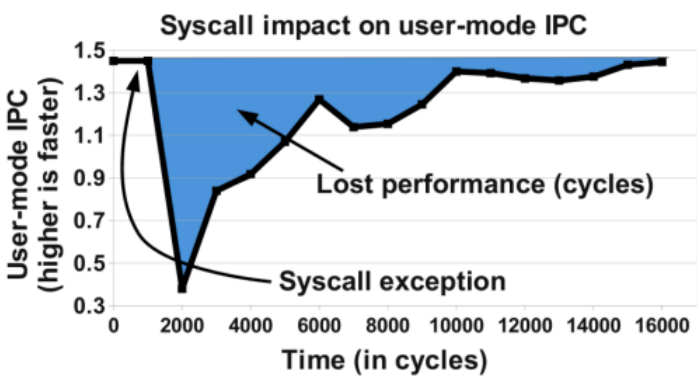
[11] FreeBSD

[12] Caches

[13] Pipeline

[14] Instruction per Cycle

هنگام اجرای یک فراخوانی سیستمی در بار کاری SPEC CPU 2006 روی پردازنده Intel Core i7 در نمودار زیر نشان داده شده است [2].



مشاهده می‌شود که در لحظه‌ای IPC به کمتر از 0.4 رسیده است. روش‌های مختلفی برای فراخوانی سیستمی در پردازنده های 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 sysexit/sysenter 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 مجزای پویای مشترک[15] 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.

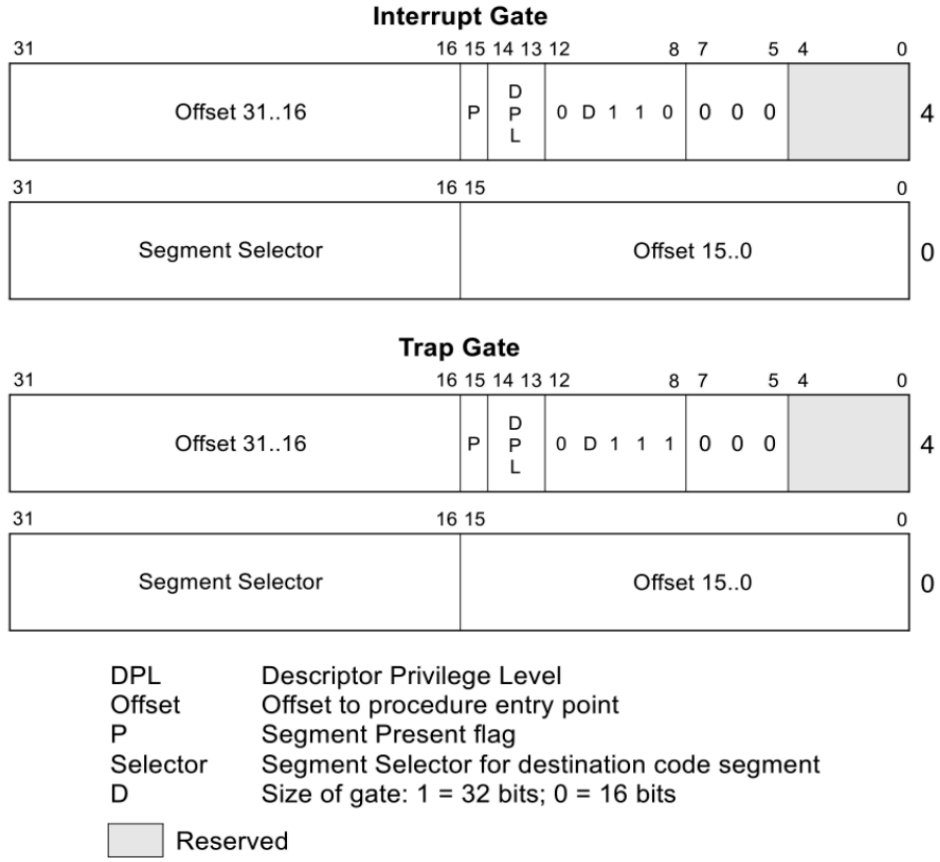
پرسش 2: فراخوانی‌های سیستمی تنها روش برای تعامل برنامه‌های کاربر با کرنل نیستند. چه روش‌های دیگری در لینوکس وجود دارند که برنامه‌های سطح کاربر می‌توانند از طریق آنها به کرنل دسترسی داشته باشند؟ هر یک از این روش‌ها را به اختصار توضیح دهید.

[15] Virtual Dynamic Shared Objects (vDSO)

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].



این ساختارها در xv6 در قالب یک ساختار هشت بایتی موسوم به struct gatedesc تعریف شده‌اند (خط 855). 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[16] نسبت داده میشود. این Gate ها توسط تابع ()tvinit در حین بوت (خط 1229) 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 3373). عملکرد Gate ها را می‌توان با بررسی پارامترهای ماکروی مقدار دهنده به Gate مربوط به فراخوانی سیستمی بررسی نمود:

پارامتر 1: [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 اینجا Gate Trap 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 KCODE_SEG<<3 به ماکرو ارسال شده است.

پارامتر ۴: محل دقیق کد در هسته که [T_SYSCALL]vector 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.

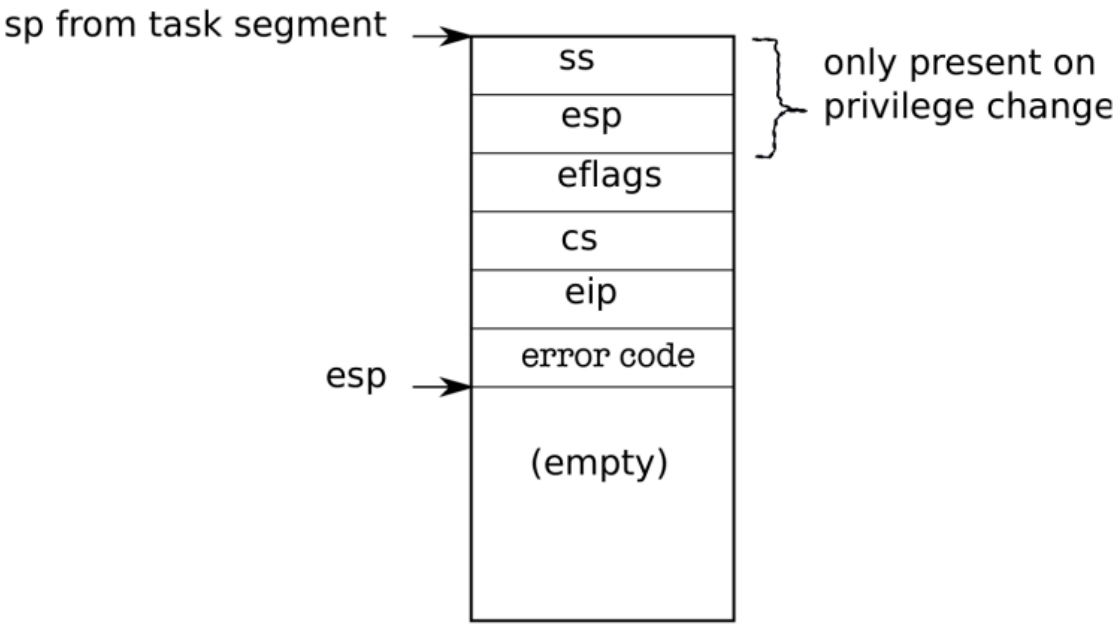
پرسش 3: آیا باقی تله‌ها را نمی‌توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟

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

[16] Trap Number

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 `init <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 پردازه 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 که در فصل مدیریت حافظه معرفی می‌گردد)، کد خطا نیز در انتها روی پشته قرار داده می‌شود. پس از اتمام عملیات سخت‌افزاری مربوط به تله، حالت پشته (سطح هسته) در 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.

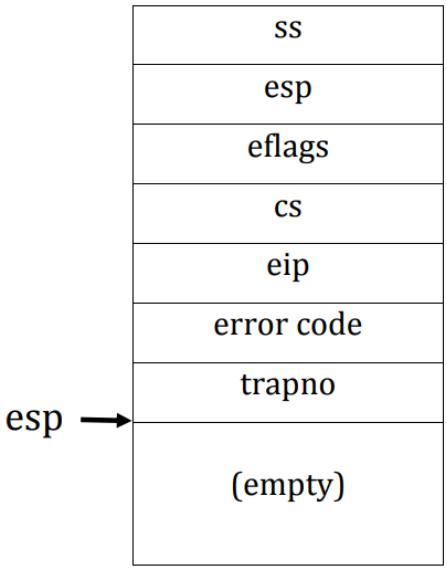


پرسش 4: در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟

در آخرین گامِ int، بردار تله یا همان کد کنترل کننده مربوط به فراخوانی سیستمی اجرا می گردد که در شکل زیر نشان داده شده است.

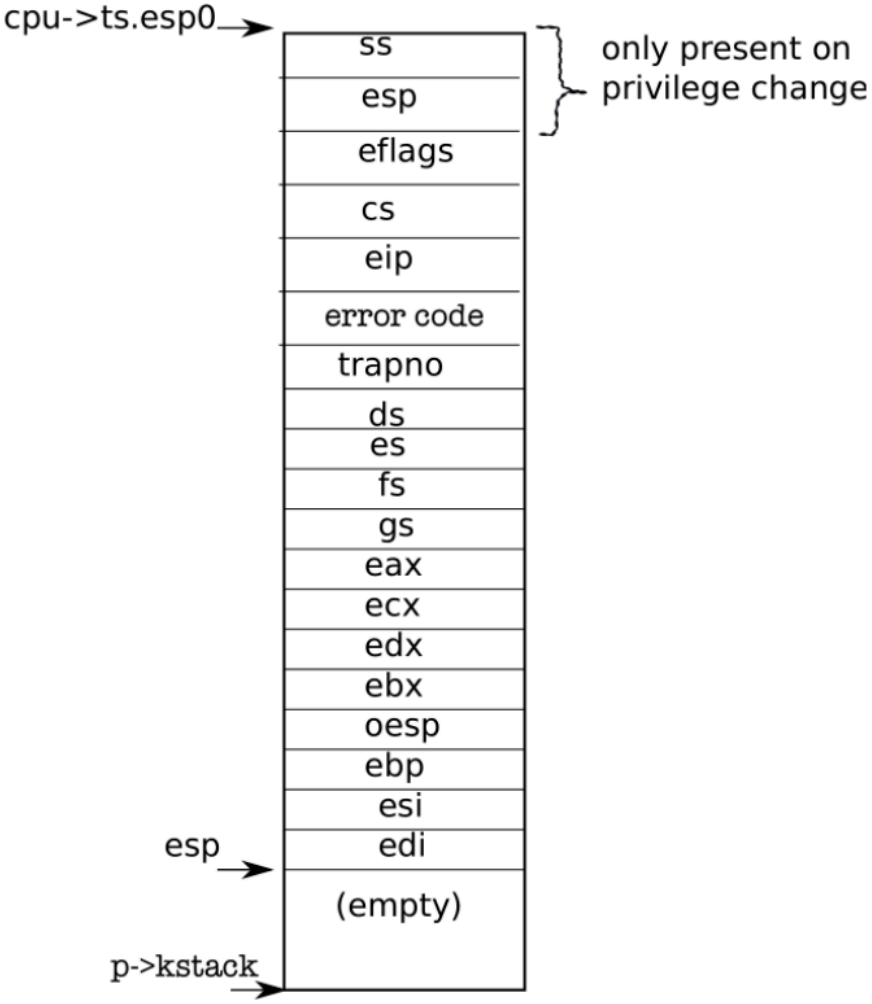
```
.globl vector64
vector64:
pushl $0
pushl $64
jmp alltraps
```

در اینجا 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` باقی ثبات‌ها 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 و 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 اینجا اشاره‌گر به قاب تله است، روی پشته قرار داده شده (خط ۳۳۲۸) 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 فراخوانی های سیستمی، متنوع بوده و هر یک دارای شمارهای منحصر به فرد است. این شماره ها در فایل syscall.h به فراخوانی های سیستمی نگاشت داده شدهاند (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 می‌کند. جدول فراخوانی های سیستمی، آرایه ای از اشاره گرها به توابع است که در فایل syscall.c قرار دارد (خط ۳۶۷۲). هر کدام از فراخوانی های 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.به طور مثال کد اسمبلی کامپایل شده منجر به چنین وضعیتی در پشته سطح کاربر برای تابع f(a,b,c) می‌شود:

esp+8	C
esp+4	B
esp	A

(گام ۲) فراخوانی دستور اسمبلی معادل call که منجر به Push شدن محتوای کنونی اشاره گر 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. به طور مثال 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 می توان با استفاده از اشاره گر ابتدای پشته به پارامترها دسترسی داشت. به طور مثال برای دسترسی به b می توان 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 شده اند. به عنوان مثال چنانچه در پروژه یک آزمایشگاه دیده شد، برای فراخوانی سیستمی exec_sys() 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. exec_sys() 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.

پرسش 5: در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr() بازه آدرس ها بررسی می‌گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی باز ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی read_sys() اجرای سیستم را با مشکل روبرو سازد.

شیوه فراخوانی فراخوانی های سیستمی جزئی از واسط باینری برنامه های کاربردی (ABI[17]) سیستم عامل روی یک معماری پردازنده است. به عنوان مثال در سیستم عامل لینوکس در معماری xv6 پارامترهای فراخوانی سیستمی به ترتیب در ثباتهای edi, esi, edx, ecx, ebx و ebp قرار داده

[17] Application Binary Interface

می‌شوند[18]. ضمن این که طبق این ABI، نباید مقادیر ثباتهای edi, esi, ebx 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 فراخوانی های سیستمی را میتوان در فایلهای زیر از کد منبع glibc مشاهده نمود[19].

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 از دست رفتن محتوای ثباتها، در آنها کپی میگردند.

در هنگام تحویل سوالاتی از سازوکار فراخوانی سیستمی پرسیده می‌شود.

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 خواهید 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 می‌باشد، چاپ کنید. آیا مقداری که مشاهده می‌کنید، برابر با شماره فراخوانی سیستمی ()getpid می‌باشد؟ علت را در گزارش کار توضیح دهید.

[18] فرض این است که حداکثر ۶ پارامتر ارسال می‌شود.
[19] مسیرها مربوط به glibc-2.26 است.

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

ارسال آرگومان های فراخوانی های سیستمی

تا اینجای کار با نحوه ارسال آرگومان های فراخوانی های 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 پیاده سازی کنید.

```
Void create_palindrome(int num)
```

در این فراخوانی، palindrome عدد داده شده محاسبه و در سطح هسته چاپ می‌شود. با اضافه کردن عکس عدد ورودی به خودش، یک palindrome تشکیل می‌شود. به عنوان مثال، اگر شماره ورودی 123 باشد، فراخوانی سیستم 123321 را تولید می کند و نتیجه را در سطح هسته چاپ می کند. تمام مراحل کار باید به همراه فایل هایی که آپلود می کنید در گزارش آورده شود.

پیاده‌سازی فراخوانی‌های سیستمی

در این آزمایش با پیاده‌سازی فراخوانی‌های سیستمی، اضافه کردن آن‌ها به هسته xv6 را فرا می‌گیرید. در این فراخوانی‌ها که در ادامه توضیح داده می‌شود، پردازش‌هایی بر پردازه‌های 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 پردازه‌ها 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 user.h ،syscall.c ،syscall.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 پردازه‌ها در هسته

پردازه‌ها در سیستم‌عامل xv6 پس از درخواستِ یک پردازه دیگر توسط هسته ساخته می‌شوند. در این صورت هسته نیاز دارد تا اولین پردازه را خودش اجرا کند. هسته xv6 برای نگهداری هر پردازه یک ساختار داده ساده دارد که در یک لیست مدیریت می‌شود. هر پردازه اطلاعاتی از قبیل شناسه که توسط آن شناخته می‌شود، پردازه والد و غیره را در ساختار خود دارد. برای ذخیره‌کردن 26 واحد خود اطلاعات بیشتر، میتوان داده‌ها را به این ساختار داده اضافه کرد.

حال قصد داریم تا با استفاده از چند فراخوانی سیستمی روند صدا زده شدن فراخوانی‌های سیستمی توسط پردازه‌ها را بررسی کنیم و اطلاعات استفاده‌شده و دستکاری‌شده توسط آن‌ها را نمایش دهیم. هدف از این بخش آشنایی با بخش‌های مختلف عملکرد فراخوانی‌های سیستمی است:

1. پیاده سازی فراخوانی سیستمی انتقال فایل

در این قسمت شما فراخوانی سیستمی را طراحی می‌کنید که نام یک فایل را گرفته و به محل آدرس مقصد منتقل می‌کند. نام فایل مقصد با مبدا یکی خواهد بود.

```
Int move_file(const char* src_file, const char* dest_dir)
```

برای تست این فراخوانی سیستمی، یک برنامه‌ی سطح کاربر بنویسید و فراخوانی سیستمی گفته شده را فراخوانی کنید و نتیجه را نشان دهید.

عمل انتقال باید به طور کامل در کرنل و نه در برنامه سطح کاربر انجام شود. برای تولید فایل، میتوانید از دستور echo استفاده کنید. همچنین برای ساخت دایرکتوری مقصد می‌توانید از دستور mkdir بهره ببرید. این فراخوانی سیستمی در صورت موفقیت 0 و در غیر این صورت 1- ریترن میکند. در صورت خطا در برنامه سطح کاربر با پیام مناسب به کاربر اطلاع داده می‌شود. در صورتی که فایل مبدا و یا دایرکتوری مقصد وجود نداشت باید 1- ریترن شود.

توجه شود که فایل مبدا بعد از انتقال باید حذف شود.

2. پیاده سازی فراخوانی سیستمی مرتب سازی فراخوانی های یک پردازه

در این تابع فراخوانی های سیستمی بر اساس شماره مرتب می‌شوند.

```
Int sort_syscalls(int pid)
```

برای تست این فراخوانی سیستمی، یک برنامه‌ی سطح کاربر بنویسید و فراخوانی سیستمی گفته شده را فراخوانی کنید و نتیجه را نشان دهید.

اگر پردازه‌ای با این شناسه پیدا نشود،تابع باید 1- برگرداند و در غیر این صورت 0 را بازگرداند. در صورت نبود پردازه پیغام مناسبی چاپ شود تا کاربر مطلع شود. دقت داشته باشید که این مرتب سازی باید به وسیله جابجا کردن خانه های ساختار داده طراحی شده خودتان انجام شود.

3. پیاده‌سازی فراخوانی سیستمی برگرداندن بیشترین فراخوانی سیستم برای یک فرآیند خاص

در این قسمت شما فراخوانی سیستمی را پیاده‌سازی می‌کنید که برای یک پردازه خاص (pid)، فراخوانی سیستمی را برمی‌گرداند که بیشتر فراخوانی شده است.

```
Int get_most_invoked_syscall(int pid)
```

پس از فراخوانی، باید نام فراخوانی سیستمی موردنظر و تعداد دفعاتی که فراخوانی شده است در خروجی نشان داده شود. اگر هیچ فراخوانی سیستمی فراخوانی نشده است یا اگر پردازه وجود نداشته باشد، باید یک پیام مناسب چاپ شود.

4. پیاده‌سازی فراخوانی سیستمی لیست کردن پردازه ها

این فراخوانی، تمام پردازه‌های در حال اجرا را به همراه pid آن‌ها و تعداد فراخوانی‌های سیستمی که هر کدام فراخوانی کرده‌اند را لیست می‌کند.

```
Int list_all_processes()
```

سایر نکات

- در محل بارگذاری در سایت درس، فایل‌ها و کدهای مورد نیاز به همراه گزارش پروژه را بارگذاری نمایید. بهتر است یک مخزن خصوصی در سایت GitHub ایجاد نموده و به افراد گروه خود دسترسی دهید تا بتوانید مشارکت راحت تری داشته باشید.
- پاسخ به سوالات را در گزارش 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 0 تعلق میگیرد.
- سوالات را در کوتاه‌ترین اندازه ممکن پاسخ دهید.

موفق باشید