



به نام خدا



آزمایشگاه سیستم عامل - بهار ۱۴۰۴

پروژه دوم: فراخوانی سیستمی

طراحان: گلبو رشیدی، مهدی نوری



KERNEL SPACE

VS



USER SPACE

اهداف پروژه

- آشنایی با سازوکار و چگونگی صدا زده شدن فراخوانی‌های سیستمی¹ در هسته xv6
- آشنایی با پیاده‌سازی تعدادی فراخوانی سیستمی در هسته xv6
- ذخیره سازی اطلاعات فراخوانی‌های سیستمی
- آشنایی با نحوه ذخیره‌سازی پردازش‌ها و ساختار داده‌های مربوط به آن

¹ System Call

مقدمه

هر برنامه در حال اجرا یک پردازش² 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 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

- 1) ممکن است داده‌ای از دیسک دریافت شده باشد و به دلایلی لازم باشد بلافاصله آن داده از ثبات⁵ مربوطه در دیسک به حافظه منتقل گردد. انتقال جریان کنترل در این حالت، ناشی از وقفه⁶ خواهد بود. وقفه به طور غیر همگام با کد در حال اجرا رخ میدهد.
- 2) ممکن است یک استثنا⁷ 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 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.

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 پوشاننده⁸ مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی glibc⁹ صورت می‌پذیرد. به این ترتیب قابلیت‌حمل¹⁰ 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، موجود در دایرکتوری ULIB (مانند فایل‌های usys.S و ulib.c)، توابع پوشاننده‌ای را ارائه می‌دهند که این فراخوانی‌های سیستمی را به صورت انتزاعی مدیریت می‌کنند. توضیح دهید که این کتابخانه‌ها چگونه با استفاده از ماکروها و توابع پوشاننده، جزئیات فراخوانی‌های سیستمی (مانند شماره فراخوانی، آرگومان‌ها و بازگردانی مقادیر) را از برنامه‌نویس پنهان می‌کنند. همچنین، دلایل استفاده از این انتزاع در بهبود قابلیت حمل، افزایش امنیت و ساده‌سازی توسعه برنامه‌های کاربر را بیان نمایید.

تعداد فراخوانی‌های سیستمی، وابسته به سیستم‌عامل و حتی معماری پردازنده است. به عنوان مثال در لینوکس، فری‌بی‌اس‌دی¹¹ 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 نهان¹² و خط لوله¹³ 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 سیکل¹⁴

⁸ Wrapper

⁹ در glibc، توابع پوشاننده غالباً نام و پارامترهایی مشابه فراخوانی‌های سیستمی دارند.

¹⁰ Portability

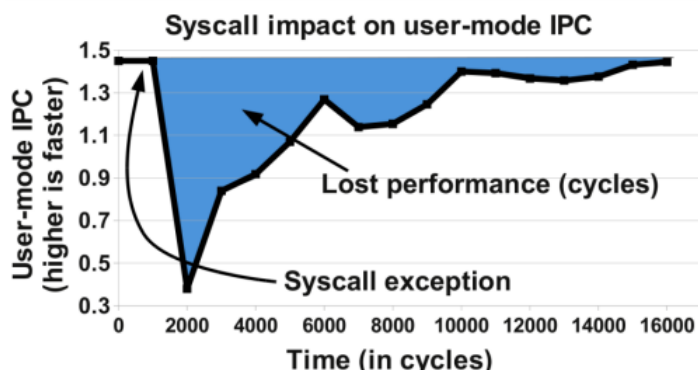
¹¹ FreeBSD

¹² Caches

¹³ Pipeline

¹⁴ Instruction per Cycle

هنگام اجرای یک فراخوانی سیستمی در بار کاری SPEC CPU 2006 روی پردازنده Intel Core i7 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 مشترک¹⁵ 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.

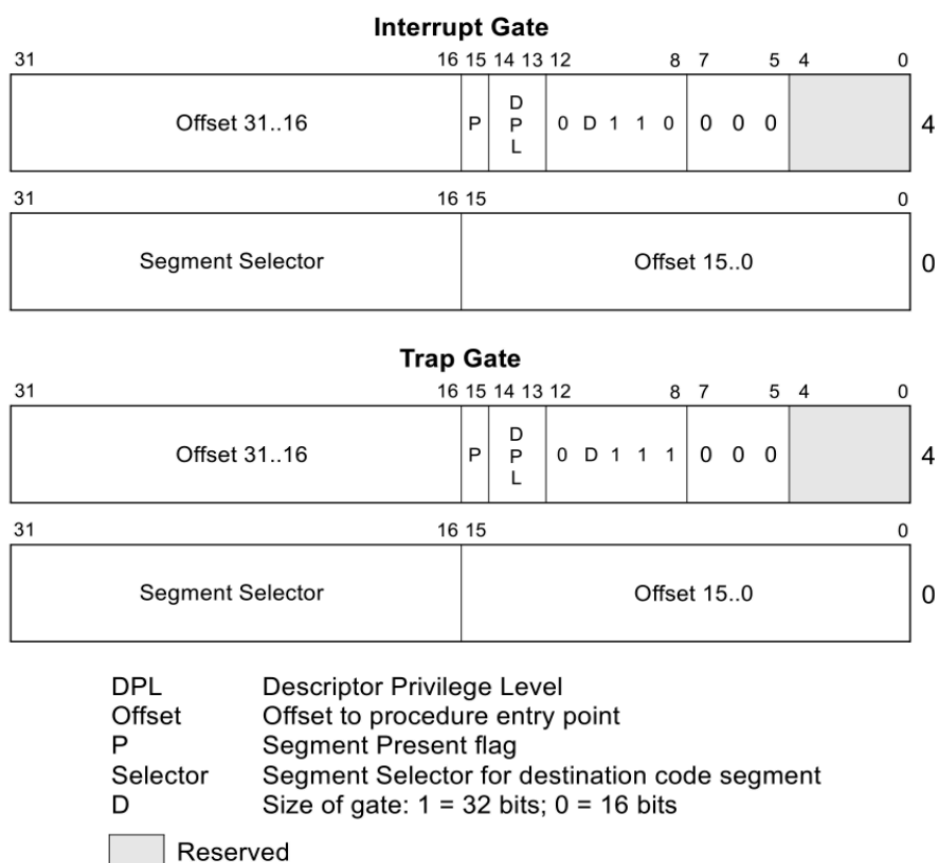
پرسش 2: فراخوانی‌های سیستمی تنها روش برای تعامل برنامه‌های کاربر با کرنل نیستند. چه روش‌های دیگری در لینوکس وجود دارند که برنامه‌های سطح کاربر می‌توانند از طریق آن‌ها به کرنل دسترسی داشته باشند؟ هر یک از این روش‌ها را به اختصار توضیح دهید.

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 تله¹⁶ 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: 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 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 <3 KCODE_SEG به ماکرو ارسال شده است.

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

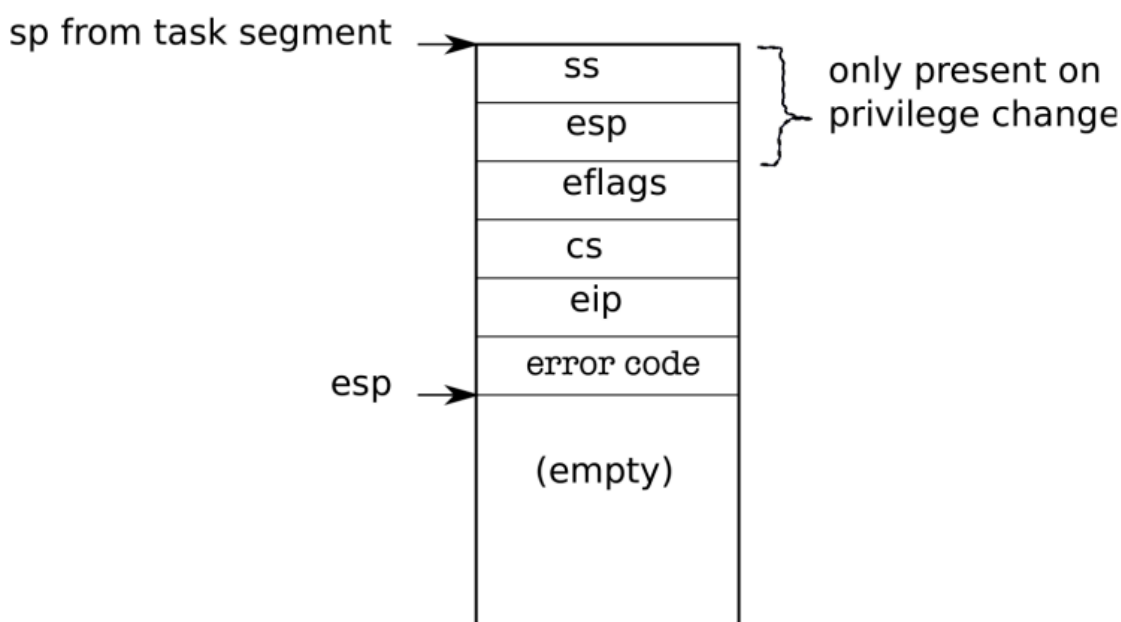
پرسش 3: با توجه به توضیحات ارائه‌شده درباره‌ی Gate Descriptor ها در xv6 و نحوه‌ی تنظیم سطح دسترسی برای فراخوانی‌های سیستمی، چرا تنها فراخوانی سیستمی (که با Trap Gate پیاده‌سازی شده) با سطح دسترسی DPL_USER فعال می‌شود و سایر تله‌ها (مانند وقفه‌های سخت‌افزاری و استثناها) نمی‌توانند از این سطح دسترسی بهره ببرند؟

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

¹⁶ Trap Number

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 `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 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 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.

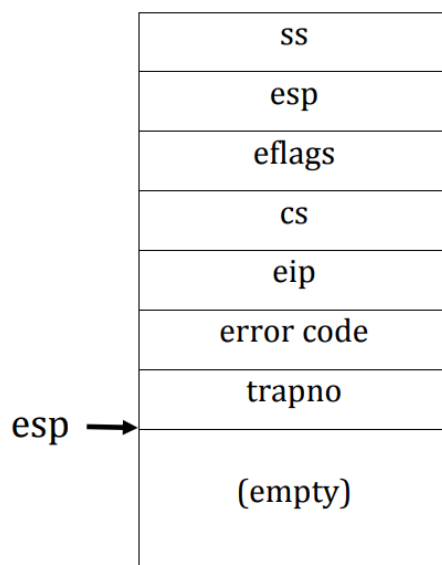


پرسش 4: در صورت تغییر سطح دسترسی، `ss` و `esp` روی پشته `Push` میشود. در غیراینصورت `Push` نمیشود. چرا؟

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

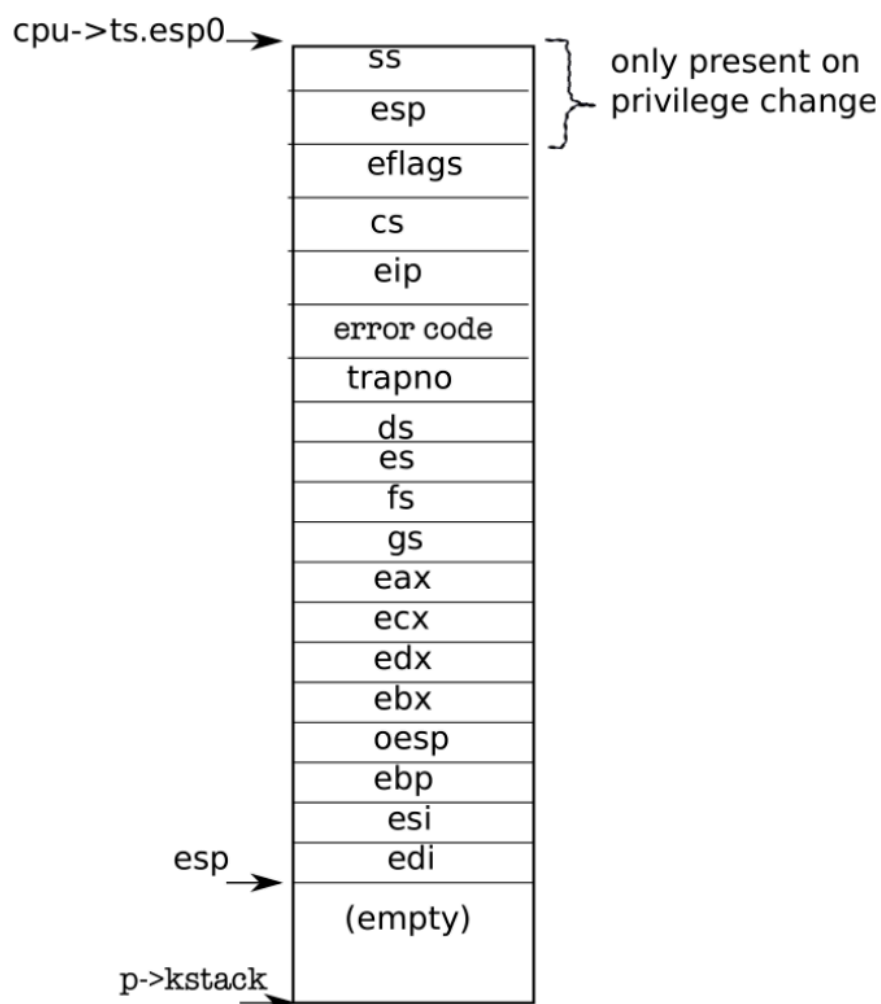
```
.globl vector64
vector64:
pushl $0
pushl $64
jmp alltraps
```

در اینجا ابتدا یک کد خطای 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 و حضور دارند. این اطلاعات موجود در پشته همان قاب تله هستند که در پروژه قبل مشابه آن برای برنامه `S.initcode` ساخته شده بود. حال اشاره گر 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 می کند. 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 می شود.

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 $f(a,b,c)$ می شود:

| | |
|-------|---|
| 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 می گردد. 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. $exec_sys()$ 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.

پرسش 5: با توجه به توضیحاتی که در مورد نحوه قرارگیری پارامترهای فراخوانی سیستمی بر روی پشته و نحوه دسترسی به آنها از طریق توابعی مانند $argint$ و $argptr$ داده شده است، توضیح دهید که این توابع چه نقشی در بازیابی پارامترهای فراخوانی سیستمی دارند. به خصوص، چرا تابع $argptr$ باید بازه های آدرس ورودی را بررسی کند؟ در صورت عدم انجام این بررسی ها، چه نوع مشکلات امنیتی (مانند دسترسی به حافظه خارج از محدوده مجاز) ممکن است ایجاد شود؟ به عنوان مثال، شرح دهید که چگونه نبود این بررسی ها در فراخوانی سیستمی $read_sys$ می تواند باعث بروز اختلال یا آسیب پذیری در سیستم شود.

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 ثابتهای ebx, ecx, edx, esi, edi و ebp قرار داده می شوند¹⁸. ضمن این که طبق این ABI، نباید مقادیر ثابتهای edi, esi, ebx و ebp پس از فراخوانی تغییر کنند. لذا باید مقادیر این ثابتها 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 میتوان در فایل های زیر از کد منبع glibc مشاهده نمود¹⁹.

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 ثابت 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 ثابتها، در آنها کپی می گردند.

در هنگام تحویل سوالاتی از سازوکار فراخوانی سیستمی پرسیده می شود.

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.

¹⁷ Application Binary Interface

¹⁸ فرض این است که حداکثر ۶ پارامتر ارسال می شود.

¹⁹ مسیرها مربوط به glibc-2.26 است.

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.

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 اینجا 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 next_palindrome(int num)
```

در این فراخوانی، عدد palindrome بعد از عدد داده شده محاسبه و در سطح هسته چاپ می‌شود. به عنوان مثال، اگر عدد ورودی 123 باشد، فراخوانی سیستم 131 را تولید می‌کند و نتیجه را در سطح هسته چاپ می‌کند. اگر عدد ورودی خودش palindrome بود آنگاه فراخوانی سیستمی همان عدد را در سطح هسته چاپ می‌کند.

تمام مراحل کار باید در گزارش آورده شود.

پیاده‌سازی فراخوانی های سیستمی

در این آزمایش با پیاده‌سازی فراخوانی های سیستمی، اضافه کردن آن‌ها به هسته xv6 را فرا می‌گیرید. در این فراخوانی‌ها که در ادامه توضیح داده می‌شود، پردازش‌هایی بر پردازش‌های موجود در هسته و فراخوانی های سیستمی صدا زده شده توسط آنها انجام می‌شود که از سطح کاربر قابل انجام نیست. شما باید اطلاعات فراخوانی های سیستمی مختلفی که توسط پردازش‌ها 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 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 خود²⁰ 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.

حال قصد داریم تا با استفاده از چند فراخوانی سیستمی روند صدا زده شدن فراخوانی‌های سیستمی توسط پردازش‌ها 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 logs

در این فراخوانی نیاز است که شما چندین فرآیند را پیاده سازی کنید.

²⁰ PID

- ساخت یک کاربر

در ابتدا فراخوانی سیستمی `make_user` را پیاده سازی کنید :

```
Int make_user_syscall(int user_id, const char* password)
```

در این فراخوانی ما اقدام به ساخت یک کاربر می‌کنیم. توجه شود که در اینجا مقدار `user_id` باید یکتا باشد.

در اجرای این فراخوانی اگر `user_id` داده شده یکتا نباشد خروجی ما ۱- خواهد بود و در غیر این صورت ۰ را خواهیم داشت.

- ورود و خروج

```
Int login_syscall(int user_id, const char* password)
```

```
Int logout_syscall()
```

در این بخش در حالت `login` وجود `user_id` خاص بررسی شده و در صورت تطابق رمز عبور ها کاربر وارد می‌شود. اگر `login` موفقیت آمیز بود خروجی ۰ و در غیر این صورت ۱- باز می‌گردد.

در حالت `logout` اگر کاربری وارد شده بود آن را خارج کرده و ۰ باز می‌گردانیم و در غیر این صورت ۱- بر می‌گردد.

- دستور logs

در این دستور اگر کاربری وارد شده باشد لیستی از سیستم کال هایی که فراخوانی کرده نمایش داده می‌شود (نمایش شماره سیستم کال کفایت می‌کند). حال اگر کاربری وارد نشده بود لاگ به ازای تمام کاربران نشان داده می‌شود.

```
Int get_logs_syscall()
```

2. پیاده سازی فراخوانی سیستمی diff

در این دستور با گرفتن نام دو فایل متنی آن دو را خط به خط مقایسه کرده و در صورت وجود تفاوت در هر خط در خروجی با استفاده از `cprintf` خطوطی که تفاوت در آن ها دیده می شود را نمایش می دهیم.

```
Int diff_syscall(const char* text_file, const char* text_file)
```

در صورتی که دو فایل یکسان باشند خروجی تابع ۰ و در غیر این صورت ۱- خواهد بود.
توجه شود که در این سوال ابتدا `fd` متناظر با فایل ها را یافته و سپس با استفاده از آنها به کار خود ادامه دهید.

3. پیاده سازی فراخوانی سیستمی خواباندن پردازش

در این قسمت سیستم کالی را طراحی کنید که پردازش به مدت زمان مشخصی که از ورودی میگیرد صبر کند و به اصطلاح بخوابد.

```
Int set_sleep_syscall(int ticks)
```

دقت کنید که این کار را بدون استفاده از فراخوانی `sleep` انجام دهید. در صورت مشاهده نمره‌ای به آن تعلق نمیگیرد.

همچنین برنامه‌ی سطح کاربری بنویسید که ابتدا ساعت سیستم را بخواند، سپس فراخوانی سیستمی گفته شده را با مقدار زمان مشخصی صدا بزند و بعد از اتمام آن، دوباره ساعت سیستم را بخواند و تفاوت بین این دو ساعت گرفته شده را چاپ کند.
راهنمایی: برای پیاده سازی این سیستم کال می‌توانید از `ticks` سیستم عامل استفاده کنید. برای خواندن ساعت سیستم نیز می‌توانید از فراخوانی سیستمی `cmostime` استفاده کنید. دقت کنید که این اختلاف ساعت با اختلاف ساعت سیستم مقدار اندکی متفاوت است. علت آن را در گزارش کار توضیح دهید.

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 راحت‌تری داشته باشید.
- پاسخ به سوالات را در گزارش خود بیاورید و در کوتاه‌ترین اندازه ممکن پاسخ دهید.
- همه افراد باید به پروژه آپلود شده توسط گروه 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 و حتی نمره منفی تعلق خواهد گرفت.

موفق باشید