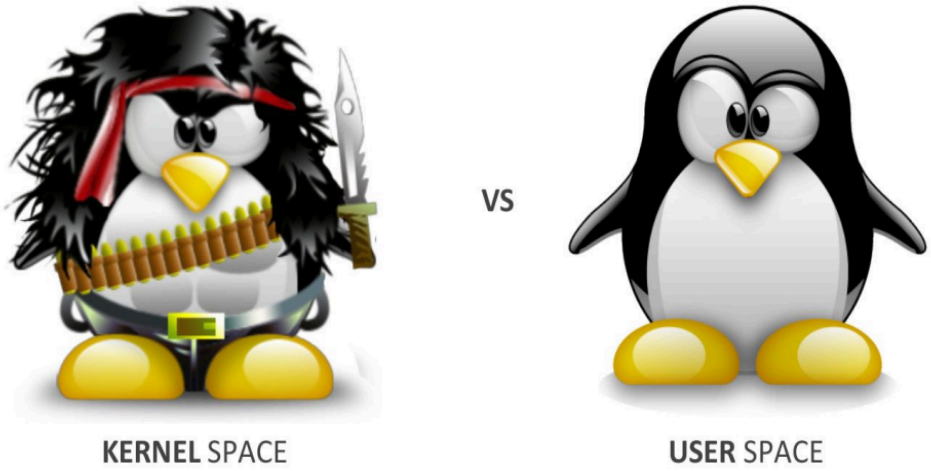
به نام خدا

آزمایشگاه سیستم عامل - پاییز ۱۴۰۴

پروژه دوم: فراخوانی سیستمی

طراحان: گلبو رشیدی، امیرارسلان شهبازی



اهداف پروژه

- آشنایی با سازوکار و چگونگی استفاده از فراخوانی‌های سیستمی[1] در هسته xv6
- آشنایی با پیاده‌سازی تعدادی از فراخوانی‌های 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 کامپیوتری ممکن است در آنِ واحد، چندین پردازه در انتظار سرویس دهی 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 لازم، اجرای آن دستور و نهایتاً تکرار این روند. 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. (به طور همگام با کد در حال اجرا رخ میدهد)
3) ممکن است برنامه نیاز به عملیات ممتاز داشته باشد. عملیاتی مانند دسترسی به اجزای سخت افزاری یا حالت ممتاز سیستم (مانند تغییر 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، 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، فری‌بی‌اس‌دی[11] و ویندوز 7 به ترتیب حدود ،300، 500 و 700 فراخوانی سیستمی وجود دارند که بسته به معماری پردازنده اندکی متفاوت خواهند 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 تغییر خواهد کرد[2]. به این ترتیب ممکن است کارایی اجرا به نصف کاهش یابد. غالباً عامل اصلی، سربار غیر مستقیم است. تعداد دستورالعمل اجرا شده به ازای هر

[8] Wrapper

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

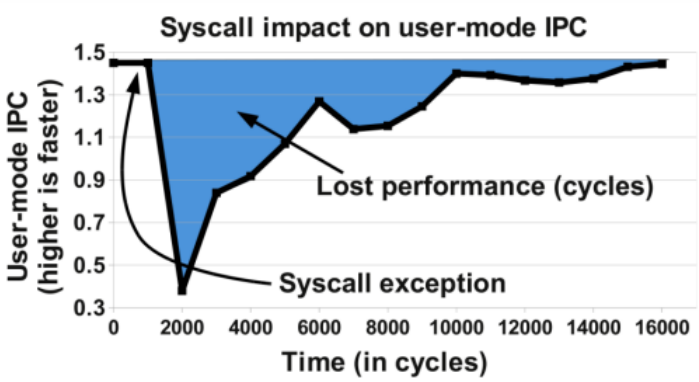
[10] Portability

[11] FreeBSD

[12] Caches

[13] Pipeline

سیکل[14] 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: مقایسه‌ای بین دستور int و دستورات جدید مانند sysenter یا sysexit در دو سیستم عامل مختلف (برای مثال Linux و xv6) انجام دهید. در این مقایسه به موارد زیر توجه کنید:

[14] Instruction per Cycle

[15] Virtual Dynamic Shared Objects (vDSO)

1. **نحوه پیاده‌سازی و عملکرد این دستورات:** چطور هرکدام از این دستورات به فراخوانی سیستم در سطح هسته پرداخته و از چه مکانیزم‌هایی استفاده می‌کنند؟

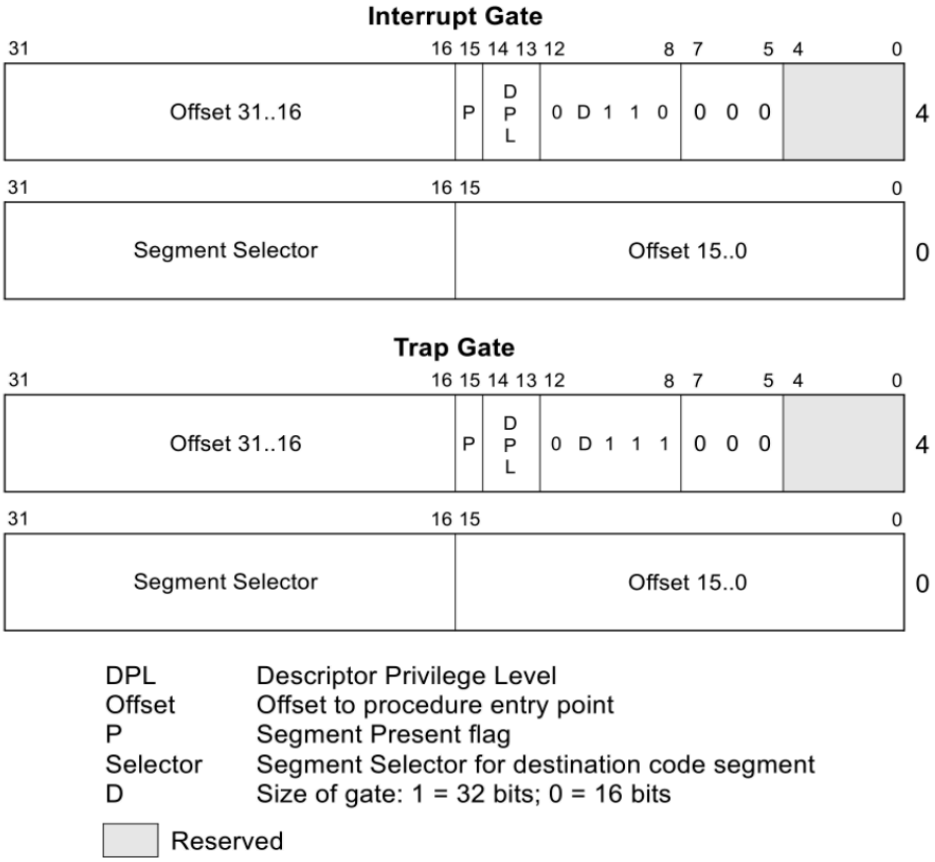
2. **کارایی:** چگونه دستور int در مقایسه با دستورات جدیدتر مانند sysenter و sysexit از نظر عملکرد و سربار پردازشی در سیستم‌های عامل مختلف عمل می‌کند؟

3. **کاربردهای مختلف:** در چه شرایطی یکی از این دستورات نسبت به دیگری برتری دارد و چرا؟

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:

[16] Trap Number

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 vector[T_SYSCALL] است. این نیز بر اساس شماره تله‌ها شاخص‌گذاری شده است.

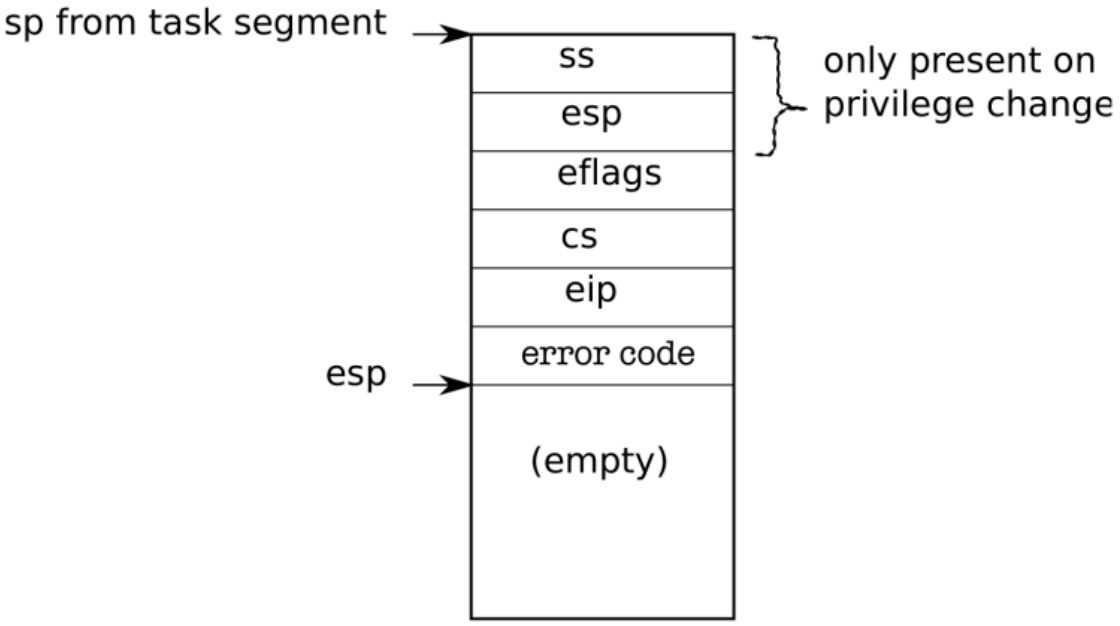
پارامتر ۵: سطح دسترسی مجاز برای اجرای این تله، DPL_USER است. زیرا فراخوانی سیستمی توسط (قطعه) کد سطح کاربر فراخوانی می‌گردد.

پرسش 3: با توجه به توضیحات ارائه‌شده درباره‌ی Gate Descriptorها 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 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 `in <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.

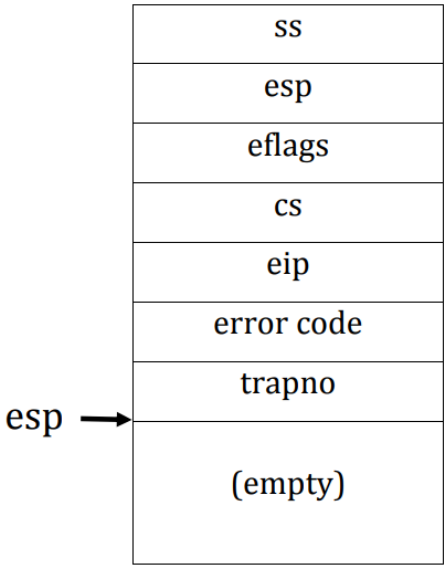


پرسش 4: در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟

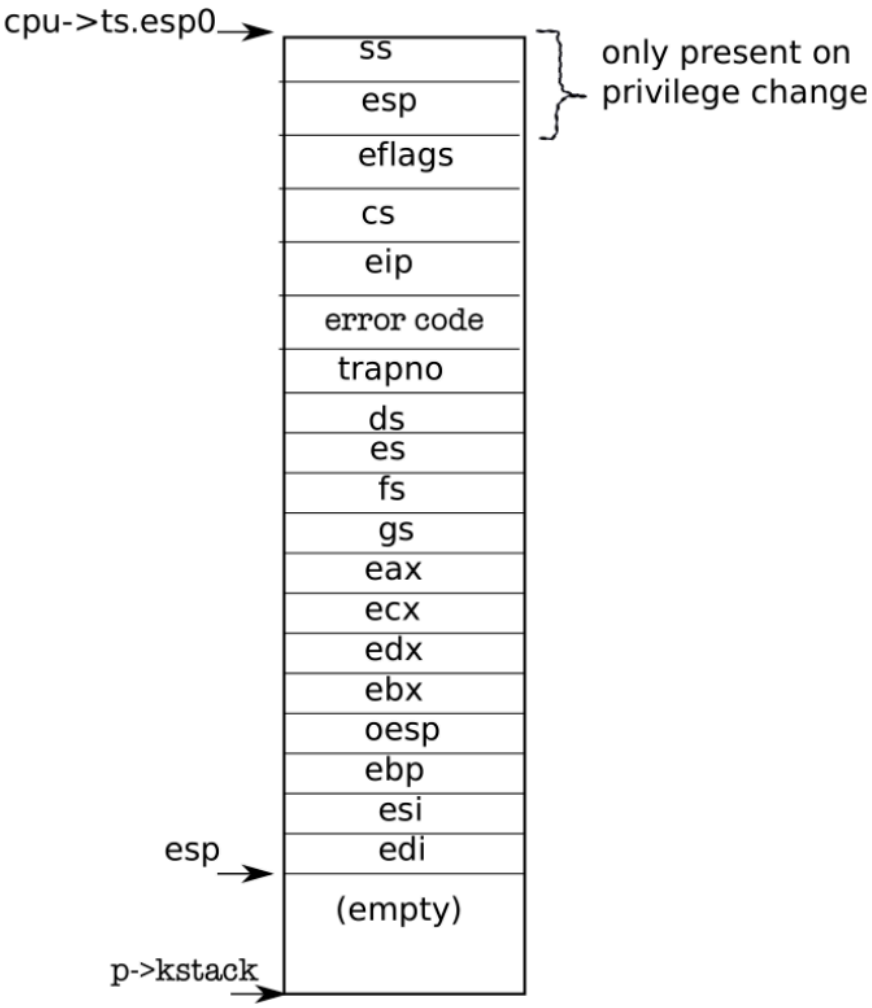
در آخرین گامِ int، بردار تله یا همان کد کنترل کننده مربوط به فراخوانی سیستمی اجرا می گردد که در شکل زیر نشان داده شده است.

```
.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 C 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 اجرا 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: 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. به‌خصوص، چرا تابع argptr باید بازه‌های آدرس ورودی را بررسی کند؟ در صورت عدم انجام این بررسی‌ها، 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 بررسی‌ها در فراخوانی سیستمی read_sys می‌تواند باعث بروز اختلال یا آسیب‌پذیری در سیستم شود. همچنین عنوان کنید که آیا حذف چک کردن بازه آدرس عملی و قابل پیاده‌سازی است یا خیر؟ یا ایا میتوان عملی بازه اشتباه وارد کنیم؟

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 x86 پارامترهای فراخوانی سیستمی به ترتیب در ثباتهای edi, esi, edx, ecx, ebx و ebp قرار داده 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

[17] Application Binary Interface

[18] فرض این است که حداکثر ۶ پارامتر ارسال می‌شود.

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.

پرسش 6: در صورتی که کاربر تغییراتی به صورت دستی در رجیسترهای بالا ایجاد کند، چه مشکلاتی به همراه خواهد داشت؟

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.

[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 طریق پشته)، قصد داریم این کار را مستقیماً از طریق ثبات‌ها (Registers) انجام دهیم.

فراخوانی سیستمی ساده‌ی زیر را که در آن دو آرگومان ورودی از نوع int وجود دارد، پیاده‌سازی کنید:

```
int simple_arithmetic_syscall(int a, int b)
```

شرح عملکرد:

شما باید سازوکار دریافت آرگومان در هسته را طوری تغییر دهید که این دو مقدار را مستقیماً از ثبات‌های مشخصی (مثلاً EBX برای a و ECX برای b) بخواند و نه از پشته‌ی پردازه.

منطق تابع:

این فراخوانی سیستمی باید محاسبه‌ی ساده‌ی (a+b) * (a-b) را انجام داده و نتیجه‌ی نهایی را برگرداند.

همچنین، برای اهداف اشکال‌زدایی، مقادیر ورودی و نتیجه‌ی محاسبه را در سطح هسته (با cprintf) چاپ کنید.

مثال: اگر ورودی‌ها (a = 5, b = 3) باشند:

- نتیجه‌ی محاسبه: (5+3) * (5-3) = 8 * 2 = 16
- خروجی چاپ‌شده در هسته: Calc: (5+3)*(5-3) = 16

- مقدار بازگشتی فراخوانی: 16

تمام مراحل کار، از جمله تغییرات مربوط به خواندن آرگومان‌ها از ثبات‌ها و پیاده‌سازی منطق تابع، باید 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.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 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 واحد

خود[20] که توسط آن شناخته می‌شود، پردازه 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 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 ایجاد نسخه کپی فایل

این فراخوانی سیستم برای کپی کردن یک فایل به نام ورودی (src_file) و ذخیره آن به نام جدید در همان مسیر طراحی شده است. در این سیستم‌کال، فایل مبدا خوانده می‌شود و یک کپی از آن در همان مسیر با پسوند copy_ ایجاد خواهد شد.

```
int make_duplicate(const char* src_fil)
```

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

عمل Duplicate باید به طور کامل در هسته سیستم‌عامل انجام شود و نه در برنامه‌های سطح کاربر. برای ایجاد فایل، می‌توانید از دستور echo استفاده کنید.این سیستم‌کال در صورت موفقیت مقدار **0** و در صورت بروز خطا مقدار **1** را برمی‌گرداند. در صورت بروز خطا در سطح کاربر، پیام مناسبی برای اطلاع رسانی به کاربر نمایش داده می‌شود. همچنین، اگر فایل مبدا وجود نداشته باشد، سیستم‌کال باید مقدار -1 را برگرداند.

2. پیاده سازی فراخوانی سیستمی بدست آوردن اعضای خانواده یک پردازه

این فراخوانی سیستم با گرفتن یک PID به عنوان ورودی، پدر (والد)، بچه‌ها و برادرها (پردازه‌هایی که والد مشترک دارند) پردازه ورودی را نمایش می‌دهد.

[20] PID

```
int show_process_family(int pid)
```

pid شناسه پردازه‌ای که می‌خواهیم روابط آن را مشاهده کنیم. این پارامتر به عنوان ورودی به سیستم‌کال داده می‌شود.

خروجی مورد انتظار:
سیستم‌کال اطلاعات زیر را به کاربر نمایش می‌دهد:

1. **پدر (والد)** پردازه ورودی.
2. **بچه‌ها** پردازه ورودی.
3. **برادرها** (پردازه‌هایی که والد مشترک دارند).

خروجی نمونه به صورت زیر خواهد بود:

My id: 10, My parent id: 5
Children of process 10:
Child pid: 8
Child pid: 9
Siblings of process 10:
Sibling pid: 6
Sibling pid: 7

توجه:
اگر پردازه با PID داده شده پیدا نشود، سیستم‌کال مقدار **-1** را برمی‌گرداند.
در صورت موفقیت، سیستم‌کال مقدار **0** را برمی‌گرداند و اطلاعات پردازه‌ها را نمایش می‌دهد.
در صورتی که پردازه‌ای بچه یا برادر نداشته باشد، پیام مناسب به کاربر نمایش داده می‌شود.
این سیستم‌کال در هسته سیستم‌عامل اجرا می‌شود و اطلاعات پردازه‌ها را مستقیماً از جدول پردازه‌ها می‌خواند.

3. پیاده‌سازی فراخوانی سیستمی جستجوی محتوا در فایل

این فراخوانی ترکیبی از کار با فایل‌سیستم، مدیریت حافظه و رشته ها در سطح هسته است. هدف این تمرین، پیاده‌سازی یک نسخه‌ی ساده‌شده از ابزار grep در سطح هسته است. این فراخوانی یک کلیدواژه و یک نام فایل را دریافت کرده و اولین خطی از فایل را که حاوی آن کلیدواژه است، به فضای کاربر برمی‌گرداند.

```
int grep_syscall(const char* keyword, const char* filename, char*
user_buffer, int buffer_size)
```

این فراخوانی یک کلیدواژه، نام فایل، یک بافر در فضای کاربر و اندازه‌ی آن بافر را دریافت می‌کند. دقت کنید کل عملیات باید در سطح هسته انجام شود. شما باید محتوای فایل را درون هسته بخوانید. این کار نیازمند مدیریت دقیق حافظه‌ی هسته (تخصیص و آزادسازی حافظه‌ی موقت) است.

در صورت موفقیت (یافتن و کپی کردن خط)، تابع باید طول خط کپی‌شده را برگرداند. در صورت شکست (مانند یافت نشدن فایل یا keyword)، تابع باید 1- را برگرداند.

برای تست فراخوانی سیستمی خود، یک برنامه‌ی سطح کاربر (grep_test.c) بنویسید. اطمینان از اینکه فراخوانی شما به درستی فایل‌ها را جستجو کرده و نتایج را به فضای کاربر برمی‌گرداند.

الزامات پیاده‌سازی:

1. برنامه‌ی شما باید بتواند grep_syscall را با یک فایل مشخص (مثلاً README یا فایلی که خودتان با echo ایجاد کرده‌اید)، یک کلیدواژه، و یک بافر برای دریافت نتیجه، فراخوانی کند.
2. برنامه باید رشته‌ی بازگشتی (که اکنون در بافر سطح کاربر قرار دارد) را در ترمینال چاپ کند.
3. شما باید هم حالت موفقیت (جستجوی کلمه‌ای که وجود دارد) و هم حالت شکست (جستجوی کلمه‌ای که وجود ندارد یا جستجو در فایلی که موجود نیست) را تست کنید.

راهنمایی:

1. باز کردن فایل در سطح هسته: فایل شما نمی‌توانید از فراخوانی sys_open استفاده کنید. باید با استفاده از توابعی مانند namei مستقیماً inode فایل را پیدا کنید.

2. در هسته به کتابخانه‌های استاندارد مانند string.h یا stdio.h دسترسی ندارید. شما باید توابع مورد نیاز برای جستجوی رشته و تشخیص کاراکتر خط جدید (\n) را خودتان پیاده‌سازی کنید.

3. به محض پیدا کردن اولین خطی که حاوی keyword است، آن خط را در user_buffer کپی کنید. user_buffer یک اشاره‌گر به بافری در فضای حافظه‌ی سطح کاربر است.

4. فراموش نکنید که حافظه‌ی تخصیص‌داده‌شده با kalloc را در نهایت با kfree آزاد کنید.

4. پیاده‌سازی فراخوانی سیستمی تنظیم اولویت

این فراخوانی سیستمی، یک نسخه‌ی ساده‌شده از دستور nice در لینوکس را پیاده‌سازی می‌کند. هدف، تغییر اولویت زمان‌بندی (Scheduling Priority) یک پردازه‌ی در حال اجرا است.

```
int set_priority_syscall(int pid, int priority)
```

این فراخوانی یک pid (شناسه‌ی پردازه‌ی هدف) و یک priority (سطح اولویت جدید) را دریافت می‌کند. شما باید یک قرارداد برای سطوح اولویت تعریف کنید (مثلاً: 0 = بالا، 1 = عادی، 2 = پایین). شما باید راهی برای ذخیره‌سازی اطلاعات اولویت هر پردازه در سطح هسته پیدا کنید.

فراخوانی سیستمی باید در صورت موفقیت 0 و در صورت یافت نشدن pid، مقدار 1- را برگرداند.

برای تست این فراخوانی سیستمی، شما باید یک برنامه‌ی سطح کاربر (priority_test.c) بنویسید که به طور ملموس **تأثیر** تغییر اولویت را نشان دهد. هدف این برنامه، اثبات این است که زمان‌بند شما به درستی پردازه‌های با اولویت بالاتر را به پردازه‌های با اولویت پایین‌تر ترجیح می‌دهد.

الزامات پیاده‌سازی:

1. برنامه‌ی شما باید *حداقل* دو پردازه‌ی فرزند با fork() ایجاد کند.
2. این فرزندان باید یک کار محاسباتی سنگین (CPU-intensive) را اجرا کنند.
3. پردازه‌ی والد باید پس از ایجاد فرزندان، از فراخوانی سیستمی تنظیم_اولویت استفاده کند تا اولویت این دو فرزند را به سطوح متفاوت (مثلاً یکی "بالا" و دیگری "پایین") تنظیم کند.
4. هر فرزند باید پس از اتمام کار محاسباتی خود، پیامی مبنی بر اتمام کار چاپ کند.

راهنمایی:

- یک کار محاسباتی سنگین، کاری است که CPU را مشغول نگه می‌دارد (مانند یک حلقه‌ی تکرار بسیار بزرگ).

سایر نکات

- پروژه خود را در یک مخزن خصوص در Github یا Gitlab پیش برده و در نهایت یک نفر از اعضای گروه کدها را به همراه پوشه git. و فایل pdf گزارش کار زیپ کرده و در سامانه با فرمت OS-Lab2-<SID1>-<SID2>-<SID3>.zip آپلود نمایید.
- رعایت نکردن مورد فوق کسر نمره را به همراه خواهد داشت.
- به تمامی سوالات پاسخ داده و پاسخ‌ها و نتایج را در گزارش کار خود قید کنید.
- بخش خوبی از نمرۀ شما را پاسخ دهی به سوالات مطرح شده تشکیل میدهد که به شما در درک نحوۀ کارکرد 6xv کمک میکند.
- سوالات را به طور خالصه پاسخ دهید.
- در پیاده‌سازی خود در خصوص مواردی که ذکر نشده‌اند میتوانید فرضی منطقی و بر اساس کارکرد shell لینوکس خود در نظر بگیرید.
- تمامی مواردی که در جلسه توجیهی، گروه تیمز و فروم درس مطرح می شوند، جزئی از پروژه خواهند بود. در صورت وجود هرگونه سوال یا ابهام میتوانید با ایمیل دستیاران مربوطه یا گروه تیمز درس در ارتباط باشید.
- همه اعضای گروه باید به تمام بخش های پروژه آپلود شده توسط گروه، چه کد و چه سوالات مسلط بوده و هنگام تحویل از تمامی اعضا و از تمامی قسمتها به صورت تصادفی سوال پرسیده خواهد شد.

- در صورت مشاهده هرگونه شباهت بین کدها یا گزارش‌ها میان گروه‌ها در این ترم یا ترم‌های گذشته، مطابق با سیاست‌های درس برخورد خواهد شد.

موفق باشید