به نام خدا

آزمایشگاه سیستم‌عامل

پروژه دوم: فراخوانی سیستمی

طراحان: صدف صادقیان-محمدهادی امیدی

تاریخ تحویل: ۱۶ فروردین

اهداف پروژه

- آشنایی با سازوکار و چگونگی صدازده 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[۲] 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[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 غیرهمگام با کد در حال اجرا رخ می‌دهد.

۲) ممکن است یک **استثنا**[۶] 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.

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 سیستم‌عامل باید به گونه‌ای باشد که مواردی از قبیل ذخیره‌سازی اطلاعات پردازه و بازیابی اطلاعات رویدادِ به وقوع

---

[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] Interrupt
[6] Exception

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 پوشاننده[۷] مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی glibc صورت می‌پذیرد.[۸] به این ترتیب قابلیت‌حمل[۹] 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.

۱) کتابخانه‌های (قاعدتاً سطح کاربر، منظور فایل‌های تشکیل‌دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانی‌های سیستمی و علت این استفاده بررسی نمایید.

سه کتابخانه استاندارد C به جز glibc را نام برده و کاربرد خاص آن‌ها را بیان نمایید.

تعداد فراخوانی‌های سیستمی، وابسته به سیستم‌عامل و حتی معماری پردازنده است. به عنوان مثال در لینوکس، فری‌بی‌اس‌دی[۱۰] 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 غیرمستقیم که ناشی از آلودگی ساختارهای پردازنده شامل انواع حافظه‌های

---

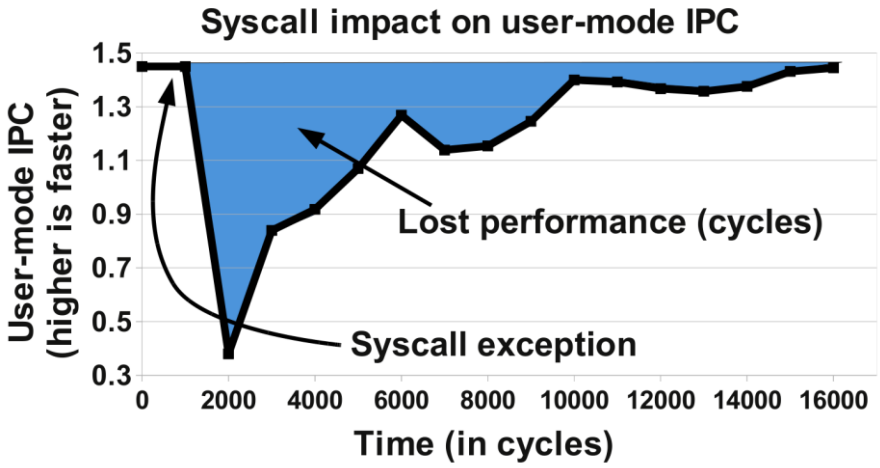
[7] Wrapper

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

[9] Portability

[10] FreeBSD

نهان[11] و خط لوله[12] می‌باشد. به عنوان مثال، در یک فراخوانی سیستمی write() در لینوکس تا $\frac{۲}{۳}$ حافظه نهان سطح یک داده خالی خواهد شد [2]. به این ترتیب ممکن است کارایی به نصف کاهش یابد. غالباً عامل اصلی، سربار غیرمستقیم است. تعداد دستورالعمل اجرا شده به ازای هر سیکل[13] (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].



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 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 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

[11] Caches
[12] Pipeline
[13] Instruction per Cycle

اشیای مجازی پویای مشترک[۱۴] (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.

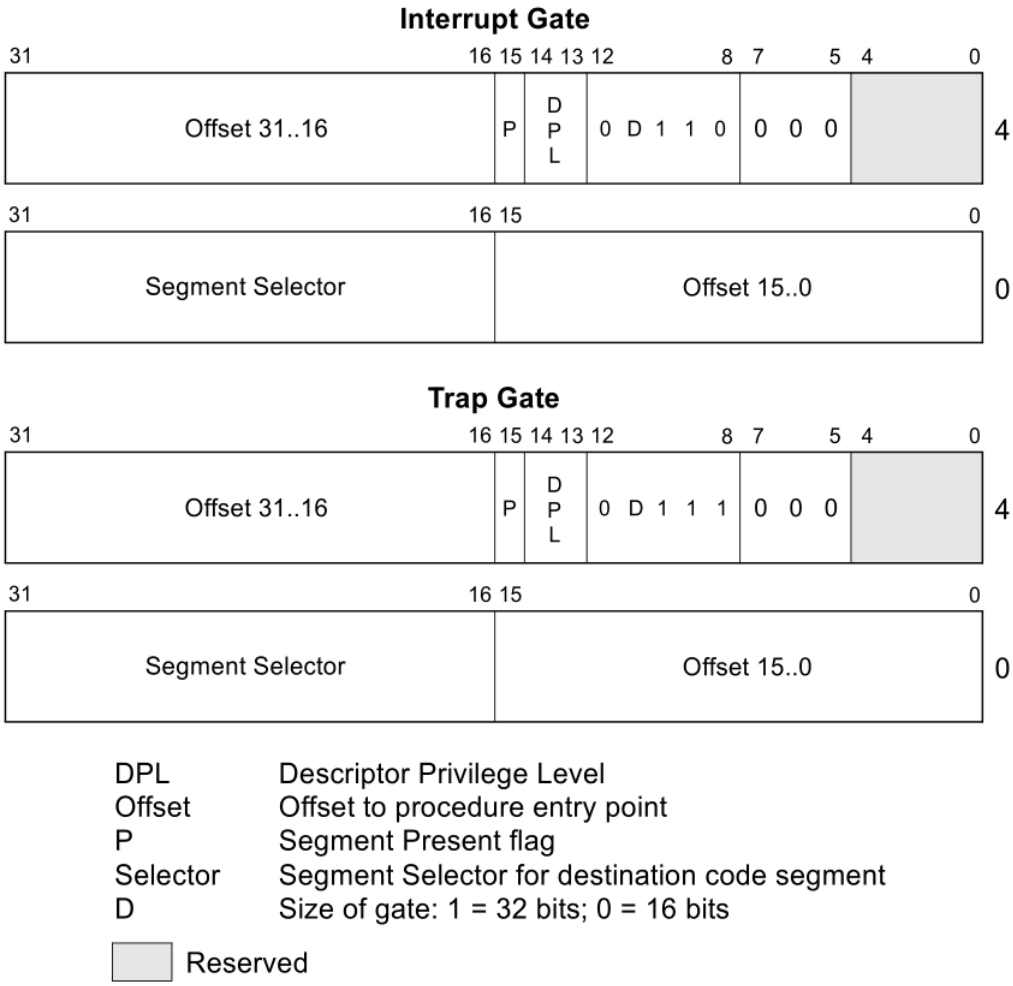
۲) دقت شود فراخوانی‌های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش‌ها را در لینوکس به اختصار توضیح دهید. می‌توانید از مرجع [3] کمک بگیرید.

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 Gateها در شکل زیر نشان داده شده است [4].

[14] Virtual Dynamic Shared Objects



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 تله[15] نسبت داده می‌شود. این Gateها 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 Gateها را می‌توان با بررسی پارامترهای ماکروی مقداردهنده به Gate مربوط به فراخوانی سیستمی بررسی نمود:

[15] Trap Number

پارامتر ۱: 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 Trap Gate 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 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.

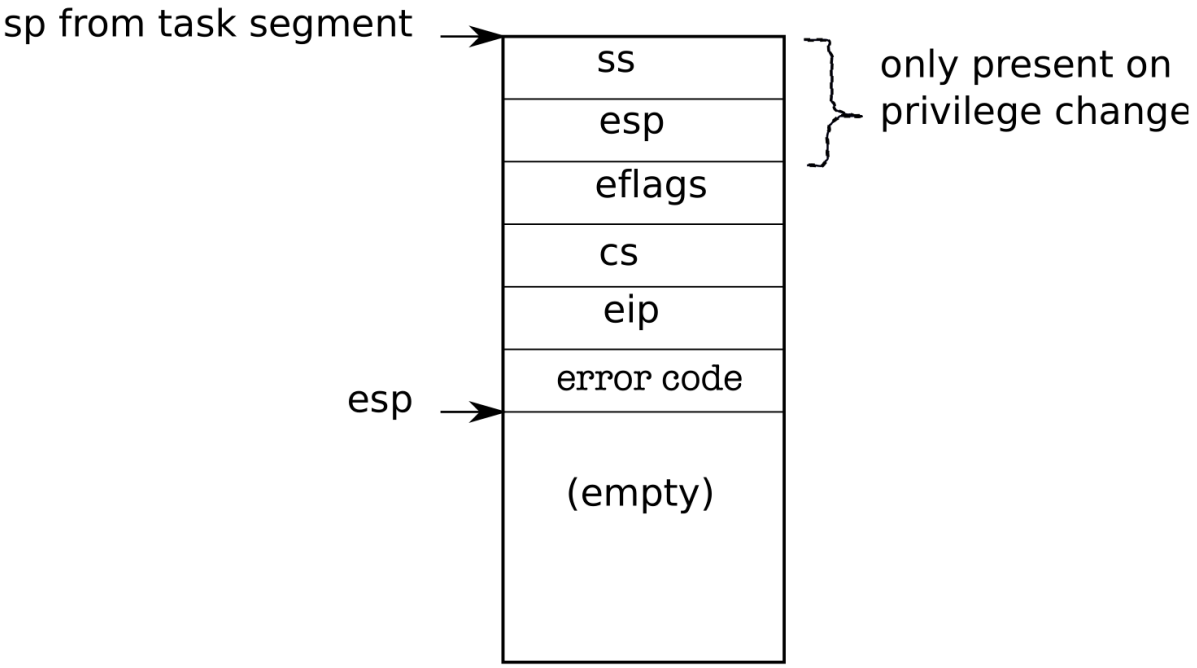
پارامتر ۵: سطح دسترسی مجاز برای اجرای این تله. DPL_USER است. زیرا فراخوانی سیستمی توسط (قطعه) کد سطح کاربر فراخوانی می‌گردد.

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.

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 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[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 گامِ int، بردار تله یا همان کد کنترل‌کننده مربوط به فراخوانی سیستمی اجرا می‌گردد که در شکل زیر نشان داده شده است.

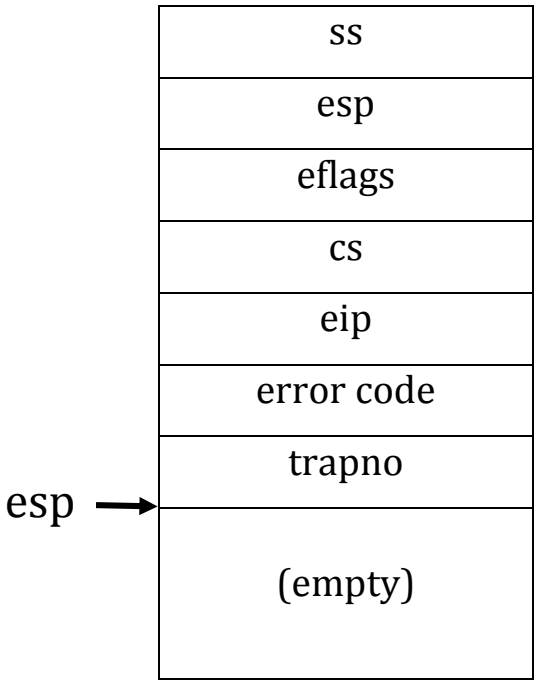
```
.globl vector64
```

[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 switchuvm() برای اجرا هنگام وقوع تله در آن برنامه آماده می‌گردد.

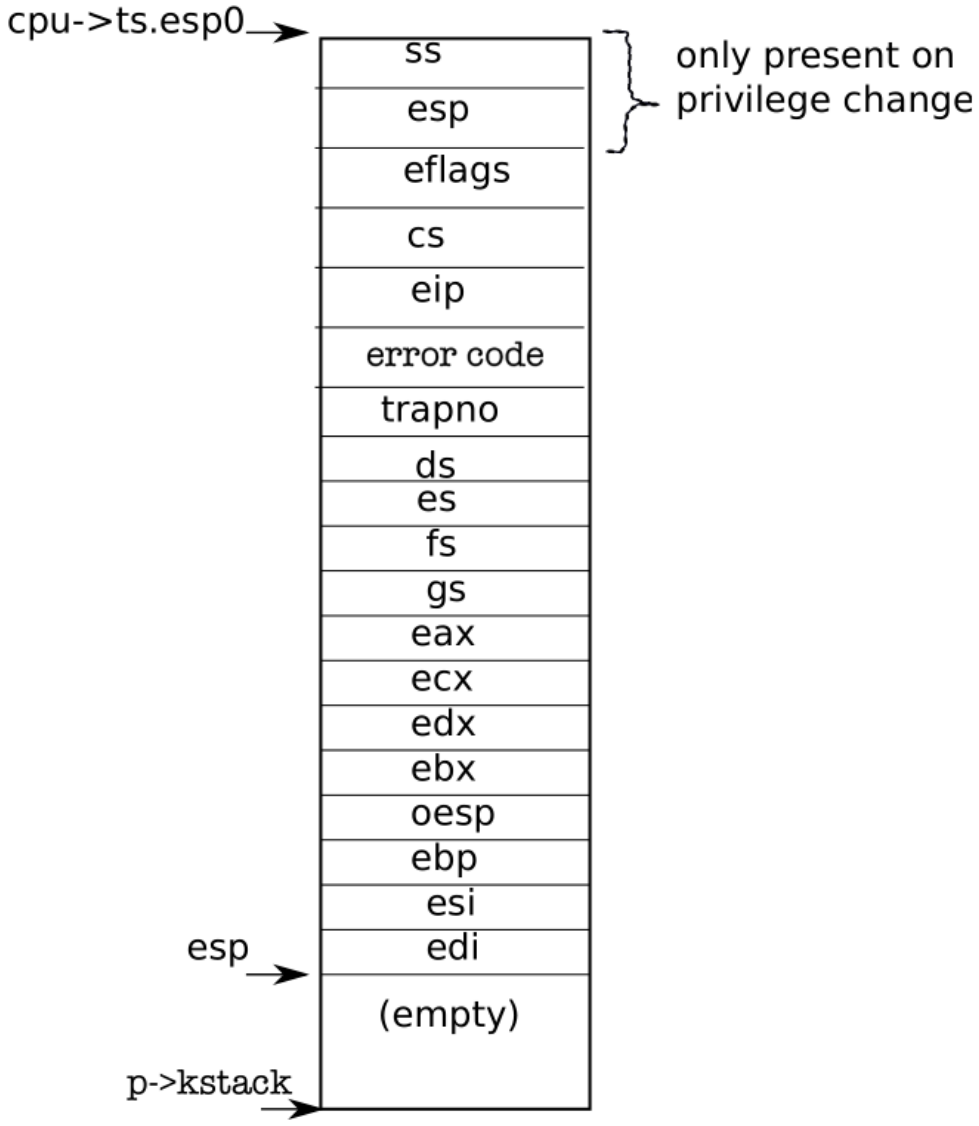
```
vector64:
    pushl $0
    pushl $64
    jmp alltraps
```

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، 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 ۳۳۱۸) 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 فراخوانی‌های سیستمی، متنوع بوده و هر یک دارای شماره‌ای منحصربه‌فرد است. این شماره‌ها در فایل syscall.h به فراخوانی‌های سیستمی نگاشت داده شده‌اند (خط ۳۵۰۰). تابع syscall() ابتدا وجود فراخوانی سیستمی فراخوانی شده را بررسی نموده و در صورت وجود پیاده‌سازی، آن را از جدول فراخوانی‌های سیستمی اجرا می‌کند. جدول فراخوانی‌های سیستمی، آرایه‌ای از اشاره‌گرها 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 فراخوانی‌شده 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() 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 argint() (به طور دقیق‌تر در fetchint()) بازه آدرس‌ها بررسی میگردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد می‌کند؟

شیوه فراخوانی فراخوانی‌های سیستمی جزئی از واسط باینری برنامه‌های کاربردی[۱۸] (ABI) یک سیستم‌عامل روی یک معماری پردازنده است. به عنوان مثال در سیستم‌عامل لینوکس در معماری x86، پارامترهای فراخوانی سیستمی به ترتیب در ثبات‌های ebx، ecx، edx، esi، edi و ebp قرار داده می‌شوند.[۱۹] 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.[۲۰]

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. جهت آشنایی با پارامترهای فراخوانی‌های سیستمی در هسته لینوکس ۲,۶,۳۵,۴ می‌توان به آدرس زیر مراجعه نمود:

http://syscalls.kernelgrok.com/

[18] Application Binary Interface

[19] فرض این است که حداکثر شش پارامتر ارسال می‌گردد.

[20] مسیرها مربوط به glibc-2.26 است.

در هنگام تحویل سوالاتی از سازوکار فراخوانی سیستمی پرسیده می‌شود. دقت شود در مقابل ABI، مفهومی تحت عنوان واسط برنامه‌نویسی برنامه کاربردی[۲۱] (API) وجود دارد که شامل مجموعه‌ای از تعاریف توابع (نه پیاده‌سازی) در سطح **زبان برنامه‌نویسی** بوده که واسط قابل‌حمل سیستم‌عامل[۲۲] (POSIX) نمونه‌ای از آن است. پشتیبانی توابع کتابخانه‌ای سیستم‌عامل‌ها از این تعاریف، قابلیت‌حمل برنامه‌ها را افزایش می‌دهد.[۲۳] 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.

ارسال آرگومان‌های فراخوانی‌های سیستمی

تا این‌جای کار با نحوه ارسال آرگومان‌های فراخوانی‌های سیستمی در سیستم‌عامل 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.

- SYS_count_num_of_digits(int num)

در این فراخوانی، تعداد رقم‌های عدد ورودی محاسبه شده و در سطح هسته چاپ می‌شود. 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 دست‌نخورده بماند. تمامی مراحل کار باید در گزارش کار همراه با فایل‌هایی که آپلود می‌کنید موجود باشند.

---

[21] Application Programming Interface

[22] Portable Operating System Interface

[23] توابع پوشاننده فراخوانی‌های سیستمی بخشی از POSIX هستند.

پیاده‌سازی فراخوانی‌های سیستمی

در این آزمایش ابتدا با پیاده‌سازی چند فراخوانی سیستمی، اضافه‌کردن آن‌ها به هسته xv6 را فرا می‌گیرید. در این فراخوانی‌ها که در ادامه توضیح داده می‌شوند، پردازش‌هایی در سطح هسته روی اطلاعات پردازه‌ها و وضعیت سیستم صورت گرفته 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 یک فراخوان سیستمی

برای انجام این کار لینک و مستندات زیادی در اینترنت و منابع دیگر موجود است. شما باید چند فایل را برای اضافه کردن فراخوانی 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. بعد از پیاده‌سازی فراخوانی‌های سیستمی خواسته شده، لازم است تا پارامترهای ورودی آن‌ها را بازیابی کنید. در فایل sysproc.c توابعی برای این کار وجود دارند که می‌توانید از آن ها استفاده کنید. گزارشی که ارائه می‌دهید باید شامل تمامی مراحل اضافه کردن فراخوانی 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 ایجاد 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[۲۴] 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. در قسمت بعدی از شما خواسته می‌شود تا اطلاعات فراخوانی‌های سیستمی صدازده‌شده توسط هر پردازه و داده‌های آن‌ها را ذخیره کنید. برای این کار باید ساختار داده مناسبی طراحی کنید که بتواند این اطلاعات را در هر پردازه مدیریت کند.

پیاده‌سازی فراخوانی‌های سیستمی

در این قسمت قصد داریم تا با استفاده از چند فراخوانی سیستمی روند فراخوانی فراخوانی‌های سیستمی توسط پردازه‌ها را بررسی کنیم و اطلاعات مورد استفاده توسط آن‌ها را نمایش دهیم. هدف از این بخش آشنایی با بخش‌های مختلف عملکرد فراخوانی‌های سیستمی است.

۱. پیاده سازی Alarm

اولین فراخوانی سیستمی که پیاده‌سازی می‌کنید به صورت زیر است:

- void sys_set_alarm(int msecs)

این فراخوانی سیستمی به این صورت عمل می‌کند که از زمان فراخوانی به میزان زمان ارسالی منتظر بوده و سپس به کاربر هشدار خواهد داد. بدین ترتیب پردازه از فاصله زمانی، مطلع خواهد شد. برای پیاده سازی این بخش باید یک پردازه همواره در حال اجرا داشته باشید و با آن این مسئله را رسیدگی کنید. دقت کنید که در این پیاده سازی، اگر هشداری تنظیم شود اما پیش از اتمام مهلت زمانی، هشدار دیگری ایجاد شود، سیستم‌عامل باید به آخرین آن‌ها رسیدگی کند. همچنین در صورتی که عددی کوچکتر یا مساوی صفر به عنوان آرگومان این فراخوانی سیستمی باید با پیغام مناسب به کاربر اطلاع داده شود. جهت آزمودن این فراخوانی سیستمی، باید هر سه سناریو بررسی شود. (راهنمایی: برای زمان از متغیر ticks (خط ۳۳۶۴) در xv6 استفاده کنید.

[24] PID

هر ۱۰۰ تیک حدود ۱ ثانیه است. نیازی به مدیریت سرریز[۲۵] در این متغیر را ندارید. فرض کنید همواره رو به بالا حرکت می‌کند.)

امتیازی: این روش برای بازه‌های خیلی کوچک (حتی مستقل از دقت ساعت سیستم) نادقیق است. علت چیست؟ آن را دقیق‌تر پیاده‌سازی نمایید. (راهنمایی: از روتین وقفه ساعت سیستم استفاده نمایید.)

۲. ذخیره و نمایش اطلاعاتی از فراخوانی‌های سیستمی هر پردازه

در این مرحله ابتدا باید سیستم عامل xv6 را به گونه‌ای تغییر دهید که از ابتدای اجرای سیستم‌عامل، فراخوانی‌های سیستمی فراخوانی شده را به همراه خروجی آن‌ها برای هر پردازه نگه‌داری نماید. (راهنمایی: جهت نگهداری اطلاعات باید یک ساختار داده در داده‌ساختار proc ایجاد نموده و اطلاعات را در آن نگهداری کنید. سپس با فراخوانی زیر اطلاعات ذخیره شده را چاپ نمایید.)

- int sys_print_syscalls()

پس از اجرای این فراخوانی، تمامی اطلاعاتی را که تا به اینجا ذخیره شده باید در قالب زیر نمایش داده شود.

```
Proc <proc id 1>:
        -<syscall name1>: <return value>
        -<syscall name2>: <return value>
                ...
Proc <proc id 2>:
                ...
```

۳. کار با مقادیر ثبات‌ها در قاب تله

- void set_edx(int value)

در این فراخوانی سیستمی یک مقدار به عنوان ورودی گرفته می‌شود و مقدار رجیستر edx در پردازه به این مقدار تغییر پیدا می‌کند.

[25] Overflow

- void read_registers()

این فراخوانی سیستمی مقدار تمامی ثبات‌های پردازه را به عنوان خروجی چاپ می‌کند.

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 ان 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 فایل‌سیستم xv6 اضافه شود.
- برای ردیابی روال فراخوانی‌ها، پیغام‌های مناسبی در نقاط مناسب چاپ کنید.
- برای نمایش اطلاعات در سطح هسته از cprintf() استفاده کنید.

سایر نکات

- برای تحویل پروژه ابتدا یک مخزن خصوصی در سایت GitLab ایجاد نموده و سپس پروژه خود را در آن Push کنید. سپس اکانت UT_OS_TA را با دسترسی Maintainer به مخزن خود اضافه کنید. کافی است در محل بارگذاری در سایت درس، آدرس مخزن، شناسه آخرین Commit و گزارش پروژه را 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 هر گونه 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

مراجع

[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.