به نام خدا

آزمایشگاه سیستم عامل

پروژه دوم: فراخوانی سیستمی

طراحان: محمد جواد بشارتی، علی عطاءاللهی

اهداف پروژه

- آشنایی با ساز و کار و چگونگی صدا زده شدن فراخوانی‌های سیستمی[1] در هسته xv6
- آشنایی با پیاده‌سازی تعدادی فراخوانی سیستمی در هسته xv6
- ذخیره‌سازی اطلاعات فراخوانی‌های سیستمی
- آشنایی با نحوه ذخیره‌سازی پردازه‌ها و ساختار داده‌های مربوط

[1] system call

مقدمه

هربرنامه در حال اجرا یک پردازه[2] نام دارد. به این ترتیب یک سیستم رایانه‌ای ممکن است در آن واحد، چندین پردازه در انتظار سرویس داشته باشد. هنگامی که یک پردازه، در سیستم، در حال اجرا است، پردازنده روال معمول پردازش را طی می کند: خواندن یک دستور، افزودن مقدار شمارنده برنامه[3] به اندازه یک واحد، اجرای دستور و نهایتاً تکرار حلقه. در یک سیستم رویداد هایی 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 برنامه های سطح کاربر سرویس می‌دهد[4]:

1) 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[5] خواهد بود. وقفه به طور غیر همگام با کد در حال اجرا رخ می دهد.
2) ممکن است یک استثنا[6] 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 می دهد.
3) ممکن است برنامه نیاز به عملیات ممتاز داشته باشد. عملیاتی مانند دسترسی به اجزای سخت افزاری یا حالت ممتاز سیستم (مانند محتوای ثبات های کنترلی) که تنها هسته اجازه دسترسی به آن ها را دارد.در این شرایط برنامه اقدام به فراخوانی فراخوانی سیستمی می‌کند. طراحی سیستم عامل باید به گونه‌ای باشد که

---

[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

مواردی از قبیل ذخیره سازی اطلاعات پردازه و بازیابی اطلاعات رویدادِ به وقوع پیوسته مثل آرگومان‌ها را به صورت ایزوله شده از سطح کاربر انجام دهد. در این پروژه، تمرکز بر روی فراخوانی سیستمی است. در اغلب موارد، فراخوانی های سیستمی به طور غیرمستقیم و توسط توابع کتابخانه‌ای پوشاننده[7] مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی glibc صورت می‌پذیرد.[8] به این ترتیب قابلیت حمل[9] برنامه‌های سطح کاربر افزایش می‌یابد. زیرا به عنوان مثال چنان چه 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.

1) کتابخانه‌های (قاعدتا سطح کاربر، منظور فایل‌های تشکیل دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانی‌های سیستمی و علت این استفاده بررسی کنید. هم‌چنین سه کتابخانه استاندارد C به جز glibc را نام برده و کاربرد خاص آن‌ها را بیان کنید.

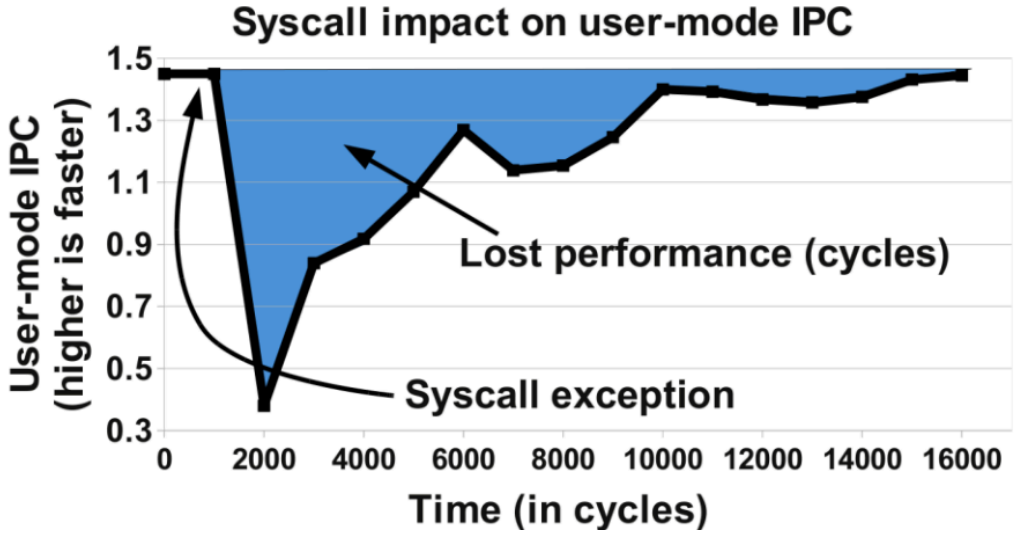
تعداد فراخوانی‌های سیستمی، وابسته به سیستم عامل و حتی معماری پردازنده است. به عنوان مثال در لینوکس، FreeBSD و ویندوز ۷ به ترتیب حدود ۳۰۰، ۵۰۰ و ۷۰۰ فراخوانی سیستمی وجود دارد که بسته به معماری پردازنده اندکی متفاوت خواهد بود. در حالی که xv6 تنها ۲۱ فراخوانی سیستمی دارد.

[7] wrapper

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

[9] portability

فراخوانی سیستمی سربارهایی دارد: ۱- سربار مستقیم که ناشی از تغییر مد اجرایی و انتقال به حالت ممتاز است و ۲- سربار غیر مستقیم که ناشی از پر کردن ساختارهای پردازنده شامل انواع حافظه‌های نهان[10] و خط لوله[11] با مقادیر جدید است. به عنوان مثال، در یک فراخوانی سیستمی write() در لینوکس تا $\frac{2}{3}$ حافظه نهان سطح یک داده خالی خواهد شد. 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[12] (IPC) هنگام اجرای یک فراخوانی سیستمی در بار کاری SPEC CPU 2006 روی پردازه Core i7 اینتل در نمودار زیر نشان داده شده است:



[10] caches

[11] pipeline

[12] instruction per cycle

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 می رود 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 فراخوانی‌های‌سیستمی (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 روش های دیگری مانند اشیای مجازی پویای مشترک[13] (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 غیر مستقیم و توسط کتابخانه gclib صورت می‌پذیرد. در ادامه ساز و کار اجرای فراخوانی سیستمی در xv6 مرور خواهد شد.

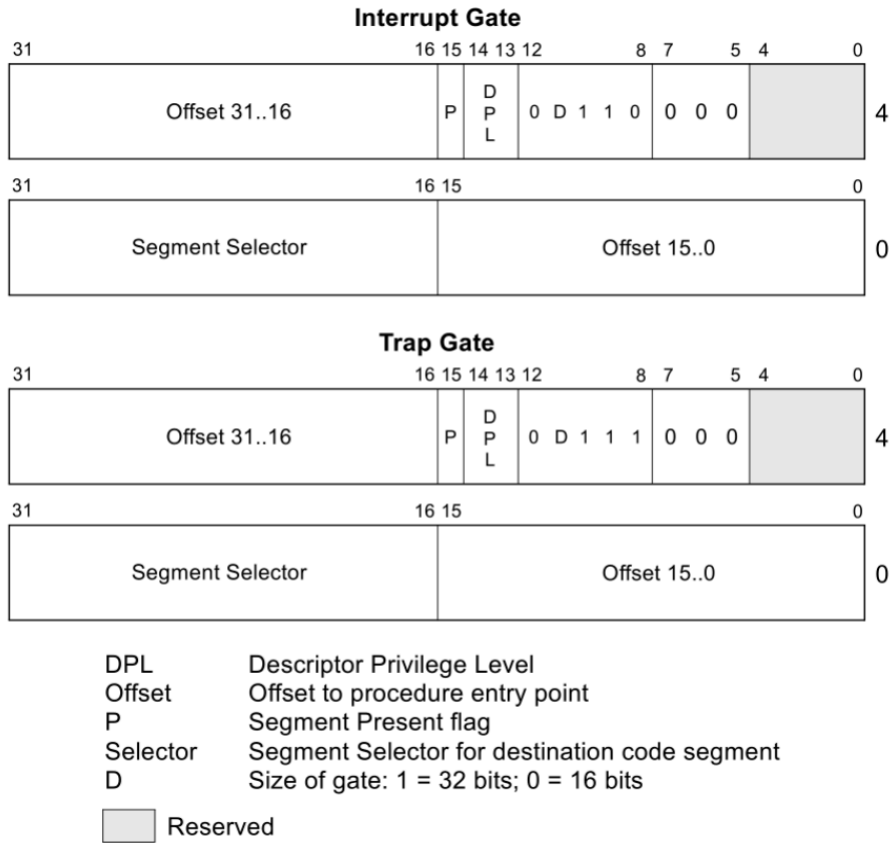
۲) دقت شود فراخوانی‌های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش‌ها را در لینوکس به اختصار توضیح دهید.

[13] Virtual Dynamic Shared Objects

ساز و کار اجرای فراخوانی سیستمی در xv6

بخش سخت افزاری 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.



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[14] نسبت داده می‌شود. این Gate ها 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 Gate ها را می‌توان با بررسی پارامترهای ماکروی

مقدار دهنده به Gate مربوط به فراخوانی سیستمی بررسی نمود:

پارامتر ۱: idt[T_SYSCALL] محتوای Gate مربوط به فراخوانی سیستمی را نگه می‌دارد. آرایه idt (خط۳۳۶۱) 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.

[14] Trap Number

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؟

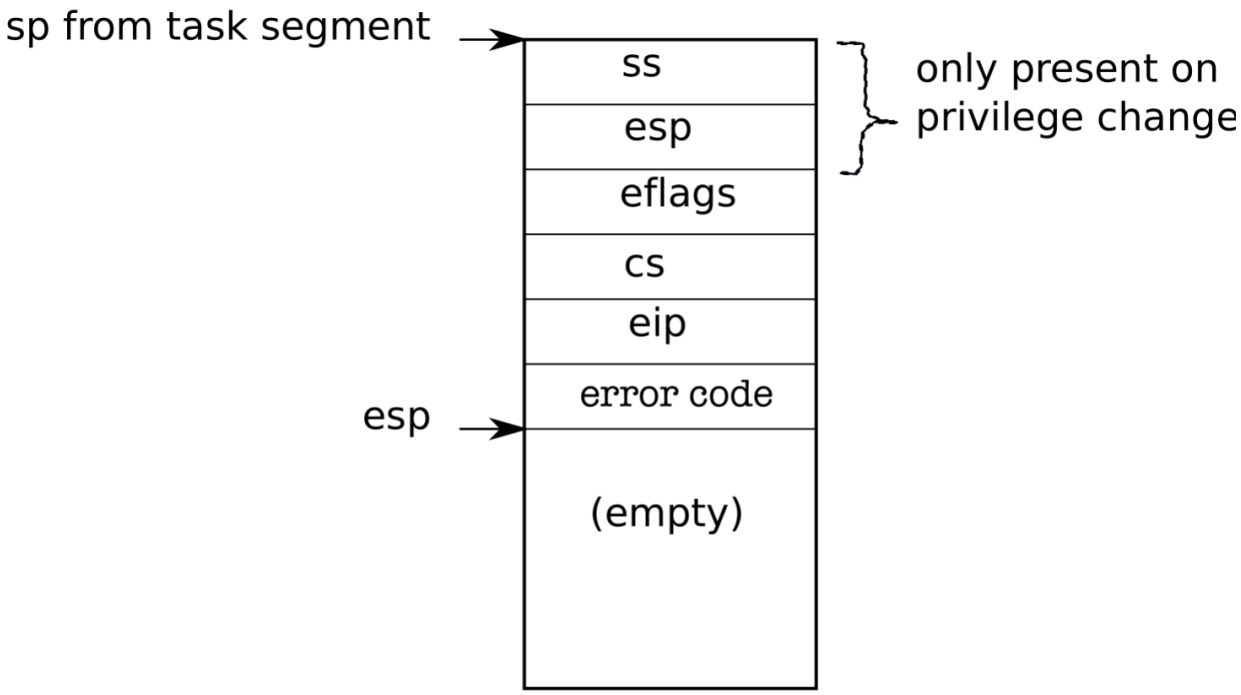
به این ترتیب برای تمامی تله‌های idt مربوطه ایجاد می‌گردد. به عبارت دیگر پس از اجرای ()tvinit آرایه idt به طور کامل مقداردهی شده است. حال باید هر هسته پردازنده بتواند از اطلاعات idt استفاده کند تا بداند هنگام اجرای هر تله چه کد مدیریتی باید اجرا شود. بدین منظور تابع()idtinit در انتهای راه‌اندازی اولیه 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> است. 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[15] که در فصل مدیریت حافظه معرفی می‌گردد)، کد خطا نیز در انتها روی پشته قرار داده می‌شود. حالت پشته (سطح هسته[16]) 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

[15] page fault

[16] دقت شود با توجه به اینکه قرار است تله در هسته مدیریت شود، 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 می‌شود.

با Push کردن کاهش می‌یابد.



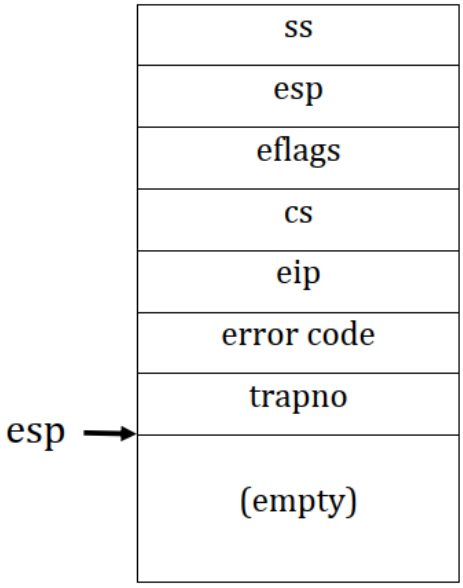
۴) در صورت تغییر سطح دسترسی، ss و esp روی پشته Push می‌شود. در غیراینصورت Push نمی‌شود. چرا؟

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

vector64:

    pushl $0
```

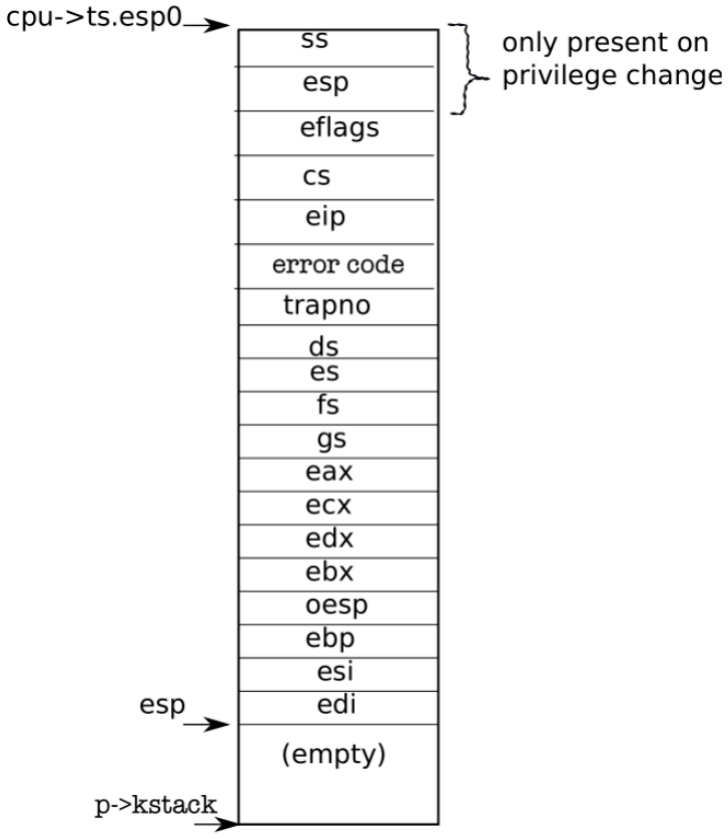
```
pushl $64

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 منحصربه‌فردند. 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. 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 امنیتی‌ای ایجاد می‌کند؟

شیوه فراخوانی فراخوانی‌های سیستمی جزئی از واسط دودویی برنامه‌های کاربردی[17] (ABI) یک سیستم‌عامل روی یک معماری پردازنده است. به عنوان مثال در سیستم‌عامل لینوکس در معماری x86، پارامترهای فراخوانی سیستمی به

[17] Application Binary Interface

ترتیب در ثبات‌های ebx، ecx، edx، esi، edi، ebp قرار داده می‌شوند.[18] ضمن این که طبق این ABI، نباید مقادیر ثبات‌های ebx، esi، edi، 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 کرد[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 می‌شوند. 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 می‌شوند. جهت آشنایی با پارامترهای فراخوانی‌های سیستمی در هسته لینوکس ۲.۶.۳۰.۵.۴ می‌توان به آدرس زیر مراجعه کرد:

http://syscalls.kernelgrok.com/

در هنگام تحویل سؤالاتی از سازوکار فراخوانی سیستمی پرسیده می‌شود. دقت شود در مقابل ABI، مفهومی تحت عنوان واسط برنامه نویسی برنامه کاربردی[20] (API) وجود دارد که شامل مجموعه‌ای از تعاریف توابع (نه پیاده‌سازی) در سطح زبان برنامه‌نویسی بوده که واسط قابل‌حمل سیستم‌عامل[21] (POSIX) نمونه‌ای از آن است. پشتیبانی توابع

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

[19] مسیرها مربوط به glibc-2.26 است.

[20] Application Programming Interface

[21] Portable Operating System Interface

کتابخانه‌ای سیستم‌عامل‌ها از این تعاریف، قابلیت حمل برنامه‌ها را افزایش می‌دهد.[22] مثلاً امکان کامپایل یک برنامه روی لینوکس و iOS فراهم خواهد شد.

ارسال آرگومان‌های فراخوان‌های 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، ازثبات‌ها استفاده میکنیم. فراخوانی سیستمی زیر را که در آن تنها یک آرگومان ورودی از نوع int وجود دارد پیاده‌سازی کنید.

- SYS_count_num_of_digits(int n)

در این فراخوانی، تعداد رقم‌های عدد ورودی محاسبه‌شده و در سطح هسته چاپ می‌شود. دقت داشته باشید که از ثبات برای ذخیره آرگومان استفاده میکنیم نه برای آدرس محل قرارگیری آن. ضمن این که پس از اجرای فراخوانی، باید مقدار ثبات دست نخورده بماند.تمامی مراحل کار باید در گزارش کار همراه با فایل‌هایی که آپلود می‌کنید موجود باشند.

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 بر پردازه‌های موجود در هسته و فراخوانی‌های سیستمی صدا زده شده توسط آنها انجام می‌شود که از سطح کاربر قابل انجام نیست.

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

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 اضافه‌کردن فراخوانی سیستمی در xv6 تغییر دهید. برای اینکه با این فایل‌ها بیشتر آشنا شوید، پیاده‌سازی فراخوانی‌های سیستمی موجود را در xv6 مطالعه کنید. این فایل‌ها شامل syscall.c، syscall.h، user.h و ... است. بعد از پیاده‌سازی فراخوانی‌های سیستمی خواسته شده، لازم است تا پارامترهای ورودی آنها را بازیابی کنید. درفایل sysproc.c توابعی برای اینکار وجود دارند که می‌توانید از آنها استفاده کنید. گزارشی که ارائه می‌دهید باید شامل تمامی مراحل اضافه کردن فراخوانی سیستمی و همین‌طور مستندات خواسته شده در مراحل بعد باشد.

نحوه ذخیره اطلاعات پردازه‌ها در هسته

پردازه‌ها در سیستم عامل xv6 پس از درخواست 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 که توسط آن شناخته می‌شود، پردازه والد و غیره را در ساختار خود دارد. برای ذخیره کردن اطلاعات بیشتر، می‌توان داده‌ها را به این ساختار داده اضافه کرد.

پیاده‌سازی فراخوانی‌های سیستمی

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

اضافه‌کردن متغیر محیطی PATH

برای اجرا شدن یک دستور، ابتدا فایل باینری آن دستور در دایرکتوری کار فعلی جست‌وجو شده و سپس اجرا می‌شود. اگر فایل خواسته شده در دایرکتوری کار فعلی وجود نداشته باشد، یک پیام خطا چاپ می‌شود.

هدف در این قسمت اضافه کردن متغیر محلی PATH است. PATH یک متغیر محیطی است که لیستی از دایرکتوری‌هایی که در آن‌ها فایل‌های اجرایی دستورات وجود دارند را مشخص می‌کند. در صورتی که فایل اجرایی دستور تایپ شده در دایرکتوری کار فعلی وجود نداشته باشد، فایل اجرایی در بقیه دایرکتوری‌های مشخص شده در PATH جست‌وجو می‌شود و در صورتی خطا چاپ می‌شود که فایل اجرایی در دایرکتوری کار فعلی و هیچ کدام از دایرکتوری‌های لیست شده در متغیر PATH نباشد.

در این قسمت باید فراخوانی سیستمی‌ای برای مقداردهی به PATH پیاده‌سازی شود. فراخوانی سیستمی‌ای پیاده‌سازی کنید که با گرفتن دستوری مشابه با مثال زیر از کاربر متغیر PATH را مقداردهی کند. دقت کنید برای لیست دایرکتوری‌ها از ":" به عنوان جداکننده استفاده شده است.

- set PATH /:bin:

در مثال بالا دایرکتوری‌های root و bin در لیست دایرکتوری‌های PATH اضافه شده‌اند.

*راهنمایی :

- می‌توانید فرض کنید در بیشترین حالت 10 دایرکتوری در این متغیر قرار می‌گیرد.
- متغیر PATH را می‌توانید در فایل proc.h تعریف کنید.

- برای افزودن یک فراخوانی سیستمی جدید بایستی چندین فایل از قبیل syscall.h، syscall.c، sysproc.c، user.h، usys.S و ... را تغییر دهید.

در ادامه باید تابع exec در فایل exec.c را طوری تغییر دهید که زمان اجرای دستور علاوه بر دایرکتوری کار فعلی سایر دایرکتوری‌های اضافه شده به متغیر PATH را نیز برای یافتن فایل اجرایی جستجو کند.

خواباندن پردازه

در این قسمت فراخوانی سیستمی‌ای طراحی کنید که پردازه به مدت زمان مشخصی که از ورودی می‌گیرد صبر کند و به اصطلاح بخوابد. 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 پیاده‌سازی فراخوانی سیستمی، می‌توانید از واحد زمانی سیستم عامل[23] (در xv6 تعداد tick ها از زمان آغاز به کار سیستم عامل، توسط هسته نگه‌داری می‌شود و با استفاده از فراخوانی سیستمی uptime قابل دسترس است) استفاده کنید.

[23] ticks

- 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.

گرفتن پردازه‌های فرزند یک پردازه

در این قسمت لازم است فراخوانی‌های سیستمی زیر را پیاده‌سازی کنید.

- get_parent_id

این فراخوانی سیستمی pid پدر پردازه‌ی فعلی را برمی‌گرداند.

- get_children

این فراخوانی سیستمی با گرفتن یک pid به عنوان ورودی، pid فرزندان آن پردازه را برمی‌گرداند. زمانی که پردازه بیشتر از یک فرزند داشت، شماره‌ی پردازه‌های آنها را به صورت یک عدد چند رقمی برگردانید. به این صورت که فرض کنید که پردازه‌ی فعلی شما دو فرزند با شماره‌پردازه‌های 4 و 5 دارد. خروجی فراخوانی سیستمی نوشته شده عدد 45 یا 54 است. (ترتیب مهم نیست)

برای تست این فراخوانی‌های سیستمی برنامه‌ای در سطح کاربر بنویسید و با استفاده از fork تعداد پردازه فرزند ایجاد کنید و برای هر پردازه pid پردازه، pid پدر پردازه و خروجی فراخوانی سیستمی get_children با pid پدر پردازه به عنوان ورودی را چاپ کنید.

همچنین برای سادگی کار، دربرنامه‌ی سطح کاربر خود برای تست کردن تعداد پردازه‌هایی که می‌سازید را طوری در نظر بگیرید که شماره‌ی پردازه‌ها یک رقمی باشند.

امتیازی

فراخوانی سیستمی بالا را طوری پیاده‌سازی کنید که علاوه بر نشان‌دادن شماره پردازه‌های فرزند، شماره پردازه‌های نوه‌ها، فرزندان نوه‌ها، نوه‌های نوه‌ها و... را نیز نشان دهد. برای تست این قسمت نیز برنامه‌ی سطح کاربری بنویسید که درستی عملکرد پیاده‌سازی شده را نشان دهد.

سایر نکات:

- برای این که بتوانید فراخوانی‌های سیستمی خود را تست کنید لازم است که یک برنامه‌سطح کاربر بنویسید و در آن 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 فایل سیستم xv6 اضافه شود.
- برای ردیابی روال فراخوانی‌ها، پیغام‌های مناسبی در جاهای مناسب چاپ کنید.
- برای نمایش اطلاعات در سطح هسته از cprintf() استفاده کنید.
- از لاگ‌های مناسب در پیاده سازی استفاده نمایید تا تست و اشکال‌زدایی کد ساده‌تر شود. واضح است که استفاده بیش از حد از آن‌ها باعث سردرگمی خواهد شد.

- کدهای خود را مشابه پروژه‌های پیشین در Github یا Gitlab بارگذاری نموده و آدرس مخزن، شناسه آخرین Commit و گزارش پروژه را در سایت بارگذاری نمایید.
- همه افراد باید به پروژه مسلط باشند و نمره تمامی اعضای گروه لزوما یکسان نخواهد بود.
- در صورت تشخیص تقلب، نمره هر دو گروه صفر در نظر گرفته خواهد شد.
- بخش امتیازی ۱۰ نمره دارد و با احتساب بخش امتیازی نمره پروژه از ۱۱۰ است.
- هرگونه سوال در مورد پروژه را از طریق ایمیل‌های طراحان می‌توانید مطرح نمایید.

aliataollahi40@gmail.com

javad.besharati78@gmail.com

موفق باشید