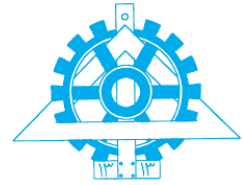




به نام خدا

آزمایشگاه سیستم‌عامل



## پروژه دوم: فراخوانی سیستمی

تاریخ تحویل: ۲۱ آبان



KERNEL SPACE



USER SPACE

### اهداف پروژه

- آشنایی با سازوکار و چگونگی صدازده شدن فراخوانی‌های سیستمی<sup>۱</sup> در هسته xv6
- آشنایی با پیاده‌سازی تعدادی فراخوانی سیستمی در هسته xv6
- ذخیره سازی اطلاعات فراخوانی‌های سیستمی
- آشنایی با نحوه ذخیره‌سازی پرده‌ها و ساختار داده‌های مربوط به آن

---

<sup>1</sup> System Call

## مقدمه

هر برنامه در حال اجرا یک پردازش<sup>۲</sup> 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 برنامه<sup>۳</sup> 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:<sup>۴</sup>

(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<sup>۵</sup> خواهد بود. وقفه به طور غیرهمگام با کد در حال اجرا رخ می‌دهد.

(۲) ممکن است یک استثناء<sup>۶</sup> 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 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 پردازش و بازیابی اطلاعات رویداد به وقوع

---

<sup>2</sup> Process

<sup>3</sup> Program Counter

<sup>۴</sup> در xv6 به تمامی این موارد trap گفته می‌شود. در حالی که در حقیقت در x86 نام‌های متفاوتی برای این گذارها به کار

می‌رود.

<sup>5</sup> Interrupt

<sup>6</sup> 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<sup>۷</sup> مانند توابع موجود در کتابخانه استاندارد C در لینوکس یعنی glibc صورت می‌پذیرد.<sup>۸</sup> به این ترتیب قابلیت حمل<sup>۹</sup> 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) استفاده شده در xv6 را از منظر استفاده از فراخوانی‌های سیستمی و علت این استفاده بررسی نمایید.

تعداد فراخوانی‌های سیستمی، وابسته به سیستم‌عامل و حتی معماری پردازنده است. به عنوان مثال در لینوکس، فری‌بی‌اس‌دی<sup>۱۰</sup> و ویندوز ۷ به ترتیب حدود ۳۰۰، ۵۰۰ و ۷۰۰ فراخوانی سیستمی وجود داشته که بسته به معماری پردازنده اندکی متفاوت خواهد بود [۱]. 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 نهان<sup>۱۱</sup> و خط لوله<sup>۱۲</sup> می‌باشد. به عنوان مثال، در یک فراخوانی سیستمی write() در لینوکس تا  $\frac{2}{3}$

<sup>۷</sup> Wrapper

<sup>۸</sup> در glibc، توابع پوشاننده غالباً دقیقاً نام و پارامترهایی مشابه فراخوانی‌های سیستمی دارند.

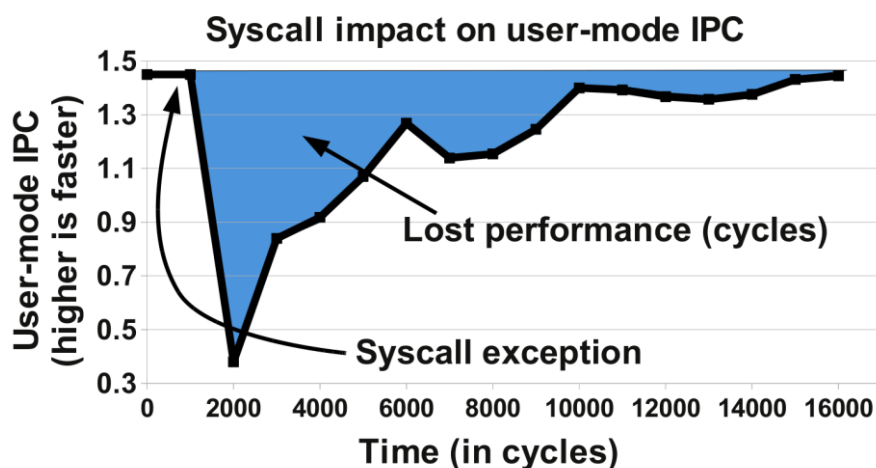
<sup>۹</sup> Portability

<sup>۱۰</sup> FreeBSD

<sup>۱۱</sup> Caches

<sup>۱۲</sup> Pipeline

حافظه نهان سطح یک داده خالی خواهد شد [۲]. به این ترتیب ممکن است کارایی به نصف کاهش یابد. غالباً عامل اصلی، سربرار غیرمستقیم است. تعداد دستورالعمل اجرا شده به ازای هر سیکل<sup>۱۳</sup> (IPC) هنگام اجرای یک فراخوانی سیستمی در بار کاری SPEC CPU 2006 روی پردازنده Core i7 اینتل در نمودار زیر نشان داده شده است [۲].



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، سربرار مستقیم آن است. در پردازنده‌های مدرن‌تر x86 دستورهای اسمبلی جدیدی با سربرار 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 سربرار مستقیم آن‌ها بر برنامه زیاد خواهد بود. در این موارد می‌توان از روش‌های دیگری مانند اشیای مجازی پویای مشترک<sup>۱۴</sup> (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

<sup>13</sup> Instruction per Cycle

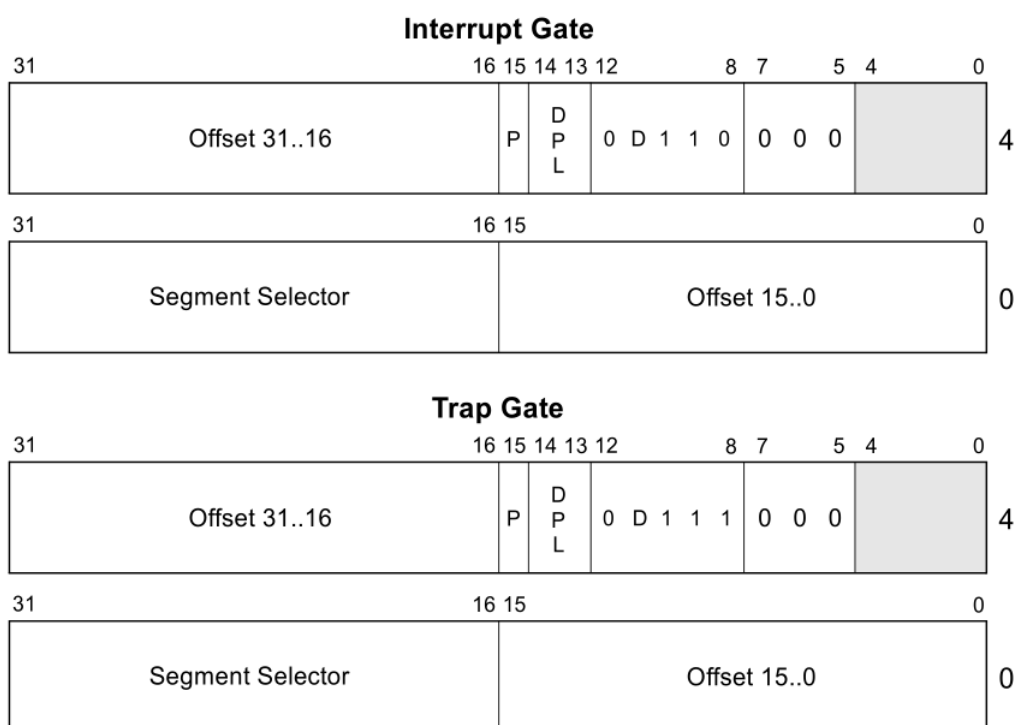
<sup>14</sup> Virtual Dynamic Shared Objects

نمی‌پذیرد. این دسترسی نیز به طور غیرمستقیم و توسط کتابخانه glibc صورت می‌پذیرد. در ادامه سازوکار اجرای فراخوانی سیستمی در xv6 مرور خواهد شد.

## 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 این جا 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 [۳].



DPL      Descriptor Privilege Level  
 Offset    Offset to procedure entry point  
 P          Segment Present flag  
 Selector   Segment Selector for destination code segment  
 D          Size of gate: 1 = 32 bits; 0 = 16 bits

Reserved

این ساختارها در xv6 در قالب یک ساختار هشت بایتی موسوم به struct gatedesc تعریف شده‌اند (خط ۹۰۰). 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<sup>۱۵</sup> 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:

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.

(۲) آیا باقی تله‌ها را نمی‌توان با سطح دسترسی DPL\_USER فعال نمود؟ چرا؟

---

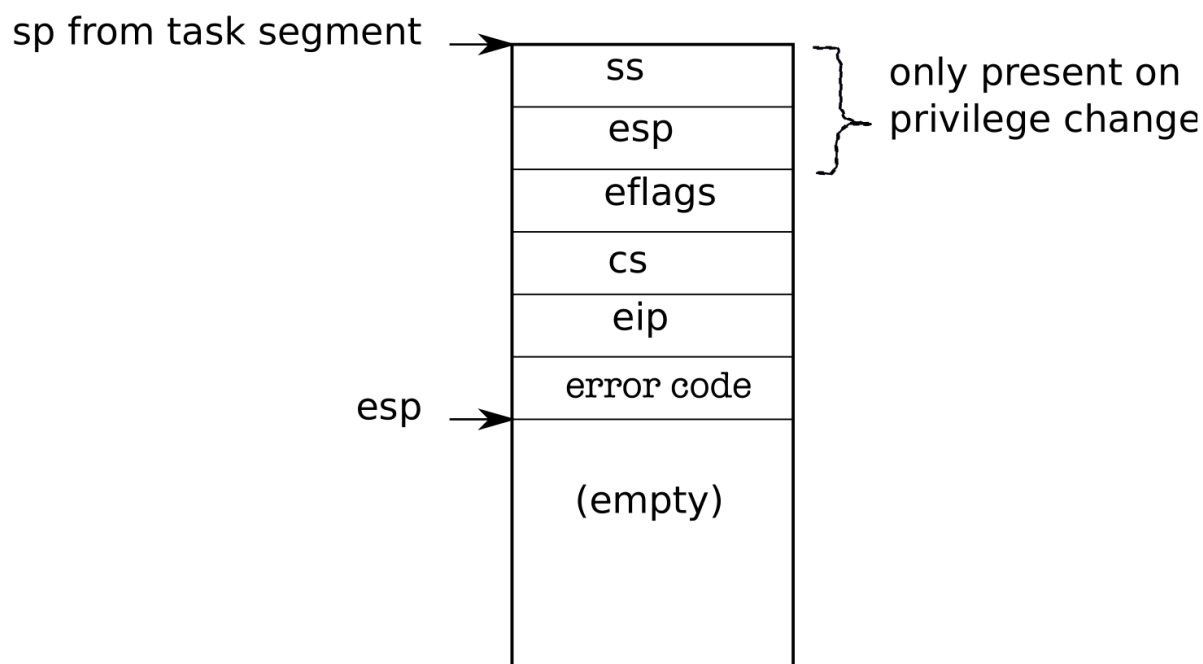
<sup>15</sup> Trap Number

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 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<sup>۱۶</sup> که در فصل مدیریت حافظه معرفی می‌گردد)، کد خطا نیز در انتها روی پشته قرار داده می‌شود. حالت پشته (سطح هسته<sup>۱۷</sup>) 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.

<sup>16</sup> Page Fault

<sup>۱۷</sup> 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 `switchvm()` برای اجرا هنگام وقوع تله در آن برنامه آماده می‌گردد.



۳) 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 کنترل‌کننده مربوط به فراخوانی سیستمی اجرا می‌گردد که در شکل زیر نشان داده شده است.

.globl vector64

vector64:

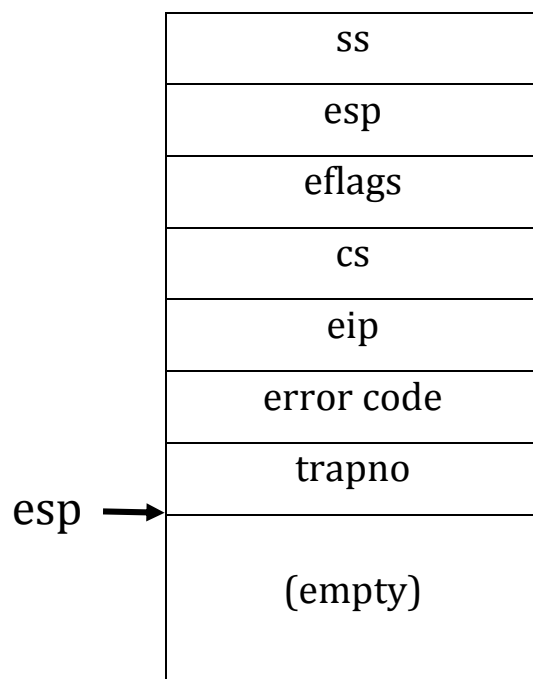
pushl \$0

pushl \$64

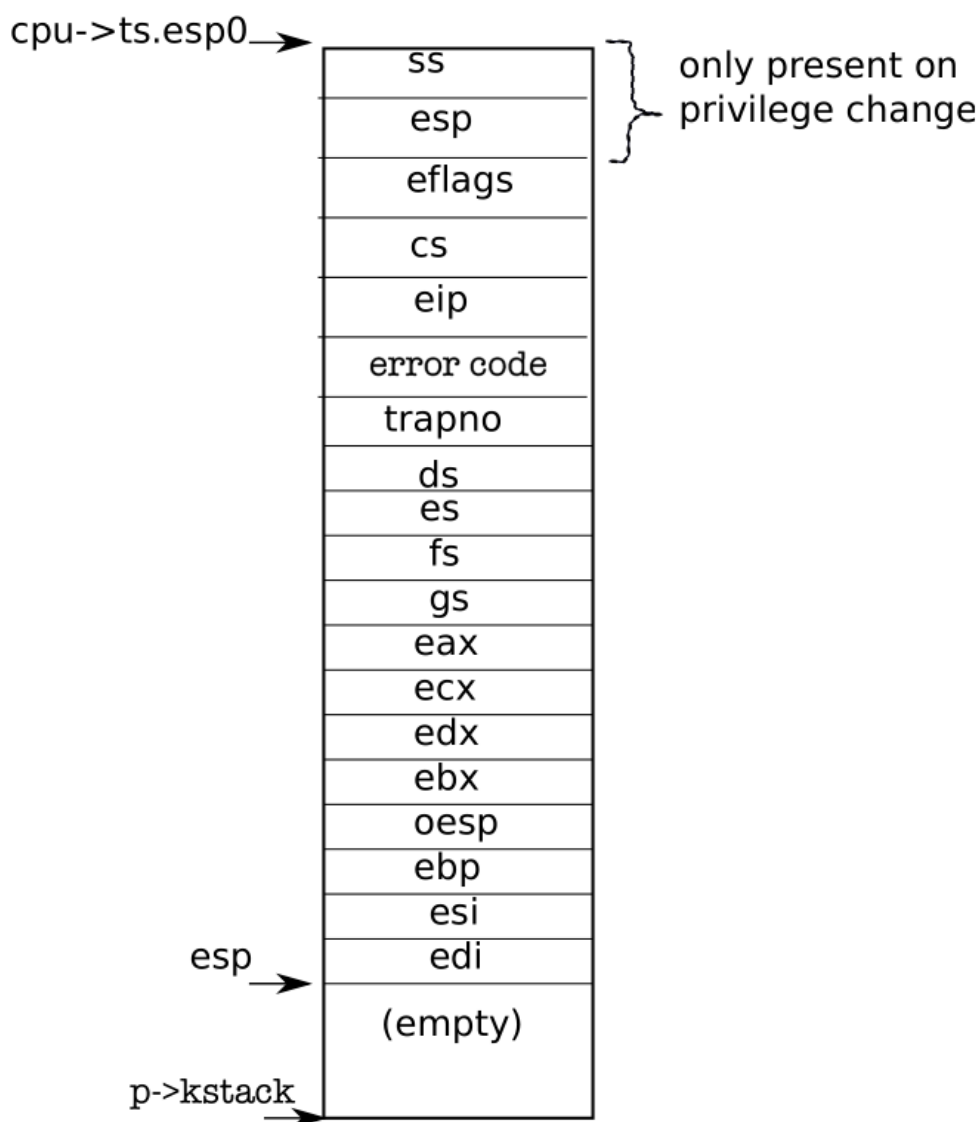
jmp alltraps

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 این جا 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 کنترل کننده زبان سی تله

تابع `trap0` 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 منحصر به فرد 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 فراخوانی شده 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.

۴) 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 واسطه باینری برنامه‌های کاربردی<sup>۱۸</sup> (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.<sup>۱۹</sup> 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

<sup>18</sup> Application Binary Interface

<sup>19</sup> فرض این است که حداکثر شش پارامتر ارسال می‌گردد.

ذخیره شده و پس از اتمام آن بازیابی کردند تا ABI محقق شود. این اطلاعات و شیوه فراخوانی فراخوانی‌های سیستمی را می‌توان در فایل‌های زیر از کد منبع glibc مشاهده نمود.<sup>۲۰</sup>

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 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/>

در هنگام تحویل سوالاتی سازوکار فراخوانی سیستمی پرسیده می‌شود.

## ارسال آرگومان‌های فراخوانی‌های سیستمی

تا این‌جا 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. فراخوان سیستمی زیر را که در آن تنها یک آرگومان ورودی از نوع int وجود دارد پیاده سازی کنید.

### • SYS\_inc\_num(int num)

در این فراخوانی، به مقدار داده شده در ورودی یکی اضافه می‌شود 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

<sup>۲۰</sup>مسیرها مربوط به glibc-2.26 است.

کار باید در گزارش کار همراه با فایل‌هایی که آپلود می‌کنید موجود باشند.

### پیاده‌سازی فراخوانی‌های سیستمی

در این آزمایش ابتدا با پیاده‌سازی چند فراخوان سیستمی، اضافه کردن آن‌ها به هسته xv6 را فرا می‌گیرید. در این فراخوانی‌ها که در ادامه توضیح داده می‌شوند، پردازش‌هایی بر پردازش‌های موجود در هسته و فراخوانی‌های سیستمی صدازده شده توسط آن‌ها انجام می‌شود که از سطح کاربر قابل انجام نیست. شما باید اطلاعات فراخوانی‌های 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 یک فراخوان سیستمی

برای انجام این کار لینک و مستندات زیادی در اینترنت و منابع دیگر موجود است. شما باید به جست‌وجوی روش درست انجام این کار بپردازید و پس از آزمودن روش، مستند مناسبی از نحوه اضافه کردن فراخوان سیستمی را در گزارش خود بیاورید. گزارش شما باید شامل تمامی مراحل اضافه کردن فراخوان سیستمی و همین‌طور مستندات خواسته شده در مراحل بعد باشد.

### نحوه ذخیره اطلاعات پردازش‌ها در هسته

پردازش‌ها در سیستم‌عامل 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 خود<sup>۲۱</sup> که توسط آن شناخته می‌شود، پردازش والد و غیره را در ساختار خود دارد. برای ذخیره کردن اطلاعات بیشتر، می‌توان داده‌ها را به این ساختار داده اضافه کرد. در قسمت بعدی از شما خواسته

---

<sup>21</sup> Pid

می‌شود تا اطلاعات فراخوانی‌های سیستمی صدا زده شده توسط هر پردازش و داده‌های آن‌ها را ذخیره کنید. برای این کار باید ساختار داده مناسبی طراحی کنید که بتواند این اطلاعات را در هر پردازش مدیریت کند.

### پیاده‌سازی فراخوانی‌های سیستمی

در این قسمت قصد داریم تا با استفاده از چند فراخوانی سیستمی اطلاعات مربوط به سیستم‌کال‌های هر پردازش را مدیریت کنیم. هر پردازش در یک سیستم‌عامل با تعدادی فراخوانی سیستمی سروکار دارد. همان‌طور که گفته شد، شما باید اطلاعات مربوط به هر فراخوانی سیستمی صدا زده شده توسط هر پردازش را ذخیره کنید و روی آن‌ها پردازش‌هایی انجام دهید. اطلاعات خواسته شده شامل شماره فراخوانی سیستمی، نام آن، لیستی از نوع آرگومان‌های مربوطه و مقدار آن‌ها، زمان صدا شدن و تعداد بار فراخوانی این سیستم‌کال است.

فراخوانی‌های سیستمی که برای مدیریت این اطلاعات پیاده‌سازی می‌کنید از این قرار می‌باشند:

- `SYS_invoked_syscalls(int pid)`

این تابع با گرفتن شناسه یک پردازش، تمام فراخوانی‌های سیستمی‌ای که این پردازش صدا زده است را با اطلاعات کاملشان نمایش می‌دهد. خروجی تابع از نوع `void` است.

در صورتی که پردازش‌های با این شناسه پیدا نشود، باید پیغام مناسبی چاپ شود.

- `SYS_sort_syscalls(int pid)`

در این تابع فراخوانی‌های سیستمی بر اساس شماره مرتب می‌شوند. مطابق قبل اگر پردازش‌های با این شناسه پیدا نشود، باید پیغام مناسبی چاپ شود. دقت داشته باشید که این مرتب‌سازی باید به وسیله جابجا کردن خانه‌های ساختار داده طراحی شده خودتان انجام شود و جابجا کردن فیلدهای داده مد نظر نیست.

- `SYS_get_count(int pid, int sysnum)`

این تابع نیز با اطلاعات فراخوانی‌های سیستمی هر پردازش سروکار دارد. با گرفتن یک شماره فراخوانی سیستمی، تعداد دفعاتی که فراخوانی شده است را برمی‌گرداند. بدیهی است که اگر این پردازش فراخوانی سیستمی مربوطه را فراخوانی نکرده است باید پیغام مناسبی چاپ شود.

- `SYS_log_syscalls()`

پس از فراخوانی این تابع، تمام فراخوانی‌های سیستمی که تا به الان توسط پردازش‌های مختلف سیستم‌عامل فراخوانی شده‌اند به ترتیب زمان فراخوانی نمایش داده می‌شوند. اطلاعاتی که باید نمایش دهید شامل نام فراخوان سیستمی، زمان فراخوانی آن و شناسه پردازش است که آن را صدا زده است. برای این کار لازم است که پردازش‌های در حال اجرای سیستم را در محل مناسبی نگه دارید. همچنین جهت بررسی درستی عملکرد خود با استفاده از چند برنامه سطح کاربر مختلف و فرستادن فراخوانی سیستمی در زمان‌های متفاوت رکوردی از سیستم‌کال‌های صدا زده شده را نمایش دهید.

### نکاتی در رابطه با فراخوانی‌های سیستمی

- برای توابع خود یک برنامه سطح کاربر بنویسید تا از صحت کارکرد آن‌ها اطمینان حاصل کنید.
- برای ردیابی روال فراخوانی‌ها، پیغام‌های مناسبی در موقعیت‌های مناسب چاپ کنید.
- برای گرفتن زمان فراخوانی یک سیستم‌کال لازم است تا از تابع کمکی `cmostime()` با آرگومان موردنیاز برای گرفتن ساعت سیستم استفاده کنید.
- برای نمایش اطلاعات در سطح هسته از `cprintf()` استفاده کنید.

### خطایابی - امتیازی



هسته xv6 یک روند کنترلی خاص به نام control-p را پشتیبانی می‌کند که امکان نمایش اطلاعات پردازنده‌ها را روی صفحه کنسول فراهم می‌کند و از آن به منظور ابزار خطایابی استفاده می‌شود. با اضافه کردن ویژگی‌های جدید خواسته شده از شما به ساختار پردازنده xv6، این روند کنترلی را تغییر دهید تا اطلاعات جدید را نیز نمایش دهد. در این قسمت تنها کافایت تعداد فراخوانی‌های سیستمی صدا زده شده توسط هر پردازنده را به اطلاعات قبلی جهت نمایش اضافه کنید و اسم هدر آن را count بگذارید.

## سایر نکات

- فقط بخش‌های تغییر یافته هسته را به همراه گزارش خود آپلود کنید.
- 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

- [1] "System Call." [Online]. Available:  
[https://en.wikipedia.org/wiki/System\\_call](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] "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide," 2015.