Operating systems

Project2 report

رضا چهرقانی 810101401

امیر نداف فهمیده 810101540

مصطفی کرمانی‌نیا 810101575

مخزن گیتهاب این پروژه:

https://github.com/mostafa-kermaninia/OS_LAB_P2

آخرین کامیت:

caf21b737f86ed482b53dea1dd39a77d10079644

- پرسش 1) کتابخانه‌های سطح کاربر در 6**XV**، برای ایجاد ارتباط میان برنامه‌های کاربر و کرنل به کار می‌روند. این کتابخانه‌ها شامل توابعی هستند که از فراخوانی‌های سیستمی استفاده می‌کنند تا دسترسی به منابع سخت‌افزاری و نرم‌افزاری سیستم‌عامل ممکن شود. با تحلیل فایل‌های موجود در متغیر **ULIB** در 6**XV**، توضیح دهید که چگونه این کتابخانه‌ها از فراخوانی‌های سیستمی بهره می‌برند؟ همچنین، دلایل استفاده از این فراخوانی‌ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه‌ها را شرح دهید.

در 6xv، کتابخانه‌های کاربری که در پوشه ULIB قرار دارند، واسطی بین برنامه‌های کاربری و کرنل فراهم می‌کنند که شامل توابعی است که مستقیماً از System Callها استفاده می‌کنند. این توابع با فراخوانی‌های سیستمی، امکان دسترسی به منابع کرنل و در نهایت منابع سخت‌افزاری را به برنامه‌های کاربری می‌دهند. برای استفاده از System Call، برنامه کاربری ابتدا باید به کرنل سیگنالی برای ورود به "حالت کرنل" بدهد. در این حالت، کرنل به اجرای کد مربوط به System Call پرداخته و پس از اتمام، دوباره کنترل را به برنامه کاربری برمی‌گرداند.

دلایل و تأثیرات استفاده از فراخوانی‌های سیستمی

استفاده از فراخوانی‌های سیستمی از طریق کتابخانه‌های کاربر در 6xv دارای مزایای متعددی است:

- کتابخانه‌های کاربر پیچیدگی تعامل مستقیم با سخت‌افزار را پنهان می‌کنند و توسعه برنامه‌های کاربری را ساده‌تر می‌سازند.

- با محدود کردن دسترسی مستقیم به سخت‌افزار، کرنل می‌تواند سیاست‌های امنیتی را اعمال کرده و اطمینان حاصل کند که فرآیندها در محدوده‌های مجاز عمل می‌کنند.
- با ارائه مجموعه‌ای یکپارچه از فراخوانی‌های سیستمی، کتابخانه‌های کاربر در 6xv تعامل با کرنل را استاندارد کرده و برنامه‌ها را قابل حمل‌تر می‌سازند.
- فراخوانی‌های سیستمی امکان مدیریت خطا در سطح کرنل را فراهم می‌کنند، که می‌تواند با شکست‌هایی مثل کمبود حافظه یا خطاهای دسترسی به فایل به صورت کارآمدتری برخورد کند.

تأثیر بر قابلیت حمل و عملکرد

استفاده از فراخوانی‌های سیستمی از طریق کتابخانه‌های کاربر، با رعایت استانداردهای مشابه سیستم‌های شبه یونیکس، بر قابلیت حمل برنامه‌ها تأثیر مثبتی دارد و مهاجرت برنامه‌های 6xv به سایر سیستم‌های مبتنی بر یونیکس را تسهیل می‌کند. با این حال، انتقال بین فضای کاربر و فضای کرنل در طی فراخوانی‌های سیستمی باعث کاهش عملکرد می‌شود، زیرا نیاز به تغییر وضعیت یا همان context switch دارد. این کاهش عملکرد، بهای امنیت و پایداری است که مکانیزم‌های حفاظتی کرنل فراهم می‌کنند.

تحلیل فایل های موجود

متغیر ULIB (در makefile) :

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

این متغیر از 4 تا object file تشکیل می شود. حالا برای تشخیص system call های آنها باید به کد منبع آن ها مراجعه کنیم:

1. **ulib.c**

این فایل کد منبع ulib.o است و شامل توابع کمکی‌ای مانند strcpy، strcmp، strlen، memset، strchr، atoi، stat، gets و memmove است که به عنوان توابع سطح کاربر عمل می‌کنند (پس در h.user اظهار یا declare شده اند) می باشد. از این توابع، تنها دو تابع gets و stat از فراخوانی‌های سیستمی بهره می‌برند:

- تابع gets: این تابع برای خواندن ورودی از stdin، از فراخوانی سیستمی read استفاده می‌کند. هر کاراکتر از ورودی با استفاده از این فراخوانی در یک حلقه خوانده می‌شود و در متغیر یا آرایه مشخصی ذخیره می‌گردد.
- تابع stat: این تابع سه فراخوانی سیستمی fstat، open و close را برای دسترسی به فایل‌ها و خواندن اطلاعات متادیتای آن‌ها (مانند سایز فایل) استفاده می‌کند. ابتدا فایل مورد نظر با open باز می‌شود، سپس با fstat اطلاعات فایل به دست می‌آید و در نهایت با close فایل بسته می‌شود.

2. **usys.S**

این فایل یک کد اسمبلی است و کد منبع usys.o است و شامل پوشاننده‌های (wrappers) فراخوانی‌های سیستمی است. در این فایل، یک ماکرو به نام SYSCALL تعریف شده که وظیفه ایجاد این پوشاننده‌ها را بر عهده دارد:

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

این ماکرو به ازای هر فراخوانی سیستمی، نام آن را به صورت global اعلام کرده و شماره فراخوانی سیستمی مربوطه را در رجیستر eax قرار می‌دهد، سپس دستور int $T_SYSCALL را اجرا می‌کند ($T_SYSCALL برابر با 64 است زیرا شماره تله فراخوانی سیستمی 64 است و برنامه جهت فراخوانی سیستمی دستور int 64 را فراخوانی می کند) که باعث ایجاد یک وقفه نرم‌افزاری شده و کنترل را به کرنل منتقل می‌کند.

3. **printf.c**

فایل printf.c کد منبع printf.o و شامل توابعی مانند printint ،putc و printf است که برای چاپ کردن اطلاعات به خروجی مورد استفاده قرار می‌گیرند. تنها تابع putc از فراخوانی سیستمی استفاده می‌کند (در تو تابع دیگر از تابع putc استفاده شده است):

- تابع putc: این تابع از فراخوانی سیستمی write برای ارسال یک کاراکتر مشخص به fd مورد نظر (که معمولاً stdout است) استفاده می کند.

از آنجا که توابع printf و printint در نهایت putc را فراخوانی می‌کنند، به‌طور غیرمستقیم از فراخوانی سیستمی write استفاده می‌کنند تا اطلاعات را به خروجی ارسال کنند.

4. **umalloc.c**

این فایل کد منبع umalloc.o است و شامل توابعی مانند free ،malloc و morecore است. تابع malloc برای تخصیص حافظه پویا و free برای آزاد کردن آن به کار می‌روند و هیچکدام مستقیماً از فراخوانی‌های سیستمی استفاده نمی‌کنند. (تابع free یک پوینتر را به عنوان ورودی می گیرد که این پوینتر به یک آدرس از حافظه اشاره می کند و سپس این بخش از حافظه را آزاد می کند و تابع malloc برای تخصیص حافظه ی پویا با اندازه ی مشخص استفاده می شود که مقدار خروجی آن یک اشاره گر void به حافظه ی تخصیص داده شده است.) اما تابع morecore که برای افزایش فضای حافظه مورد نیاز فرآیند استفاده می‌شود، از فراخوانی سیستمی sbrk بهره می‌برد تا فضای حافظه پردازه را به میزان دلخواه افزایش دهد.

- 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.

به دسته‌بندی و توضیح هر یک از این روش‌ها می‌پردازیم:

1. وقفه‌ها (**Interrupts**)

وقفه‌ها سیگنال‌هایی هستند که برای درخواست سرویس از کرنل استفاده می‌شوند و شامل دو دسته کلی هستند:

- وقفه‌های سخت‌افزاری (Hardware Interrupts): این وقفه‌ها به صورت آسنکرون (بدون وابستگی به جریان اجرای فعلی برنامه) توسط سخت‌افزارهای خارجی مانند صفحه کلید، موس یا کارت شبکه تولید می‌شوند و بدون وابستگی به جریان فعلی برنامه‌ها، CPU را برای انجام عملیات خاص فرا می‌خوانند. به عنوان مثال، فشردن کلید، حرکت موس یا دریافت بسته از شبکه، وقفه‌ای را ایجاد می‌کند که CPU را به یک روال سرویس‌دهی وقفه (ISR) هدایت می‌کند تا نیاز دستگاه برطرف شود. پس از اتمام این فرآیند، کنترل به برنامه قبلی باز می‌گردد.
- وقفه‌های نرم‌افزاری (Software Interrupts) یا Trapها: این وقفه‌ها توسط خود برنامه‌ها تولید می‌شوند و بصورت سنکرون اجرا می شوند و اغلب برای درخواست سرویس از سیستم‌عامل یا انجام عملیات خاص استفاده می‌شوند. نمونه‌هایی از این وقفه‌ها عبارتند از:
  - System Calls: فراخوانی‌های سیستمی که مستقیماً دستورات کرنل را فرا می‌خوانند.
  - Exceptions: استثناهایی که توسط CPU هنگام بروز خطاهایی مانند تقسیم بر صفر یا دسترسی غیرمجاز به حافظه ایجاد می‌شوند.
  - Signals: سیگنال‌ها پیام‌هایی هستند که از طریق کرنل برای آگاه‌سازی فرآیندها از رویدادهای خاص ارسال می‌شوند؛ مانند SIGKILL برای خاتمه فوری فرآیند و SIGINT برای متوقف کردن فرآیند با ترکیب Ctrl + C یا SIGTERM برای ارسال یک سیگنال پایان به یک فرایند استفاده می شود.

2. سیستم فایل‌های مجازی (**Pseudo-filesystems**)

سیستم فایل‌های proc/ و sys/ اینترفیس‌های مجازی هستند که داده‌های ساختارهای کرنل را در دسترس برنامه‌های کاربر قرار می‌دهند. پس، استفاده از این فایل سیستم ها نیز ، نیازمند دسترسی به هسته است.

- proc/: این فایل‌سیستم دسترسی به اطلاعات سیستمی و وضعیت پردازه‌ها را فراهم می‌کند. هر پردازه یک دایرکتوری مختص خود در proc/ دارد که شامل اطلاعاتی مانند شناسه، وضعیت و حافظه مصرفی آن است.
- sys/: این فایل‌سیستم اطلاعات و تنظیمات مربوط به سخت‌افزار و درایورها را نمایش می‌دهد. به کمک این سیستم فایل، می‌توان تنظیمات سخت‌افزاری سیستم را در زمان اجرا تغییر داد.

3. **Netlink Sockets**

سوکت‌های Netlink امکان ارتباط بین کرنل و برنامه‌های سطح کاربر را فراهم می‌کنند و به‌ویژه در پیکربندی و مدیریت شبکه و مانیتورینگ رویدادهای شبکه به کار می‌روند. این سوکت‌ها برای ارسال پیام‌های دوجهته بین پردازه‌ها و کرنل طراحی شده‌اند.

4. واسط‌های شبکه و کتابخانه‌های سطح کاربر

واسط شبکه (Network Interface): برنامه‌ها می‌توانند از طریق سوکت‌ها با کرنل و دیگر پردازه‌ها ارتباط برقرار کنند. این روش برای ارتباطات شبکه‌ای و انتقال داده‌ها از طریق TCP/IP کاربرد دارد.

کتابخانه‌های سطح کاربر (User-space Libraries): بسیاری از توابع پیچیده مانند توابع شبکه یا مدیریت دستگاه‌ها از طریق کتابخانه‌هایی مانند libc ارائه می‌شوند که توابع پیچیده کرنل را در قالبی ساده برای برنامه‌های کاربر فراهم می‌کنند.

این روش‌ها مکمل فراخوانی‌های سیستمی هستند و با ارائه یک لایه انتزاعی و واسط‌های متنوع، امکان تعامل امن و کارآمد با کرنل را برای برنامه‌های کاربر فراهم می‌کنند. این تنوع ابزارها، توسعه‌دهندگان را قادر می‌سازد تا نیازهای خاص هر برنامه را به بهترین شکل برطرف کنند و در عین حال امنیت و پایداری سیستم را حفظ نمایند.

- پرسش 3) آیا باقی تله‌ها را نمی‌توان در سطح **DPL_USER** فعال نمود؟ چرا؟

خیر، در سیستم عامل 6xv، امکان اجرای تله‌های دیگر توسط سطح دسترسی کاربر (DPL_USER) وجود ندارد و اگر یک پردازه بخواهد interrupt دیگری را فعال کند، 6xv به آن این اجازه را نمی دهد و با یک استثنای protection exception مواجه می شوند. در 6xv، توابع مرتبط با مدیریت اینتراپت‌ها و تله‌ها از طریق ماکروی

SETGATE در IDT (جدول توصیفگر اینتراپت) پیکربندی می‌شوند و تله‌های سیستمی با DPL_USER تنظیم می‌شوند تا از طریق آن‌ها، تنها فراخوانی‌های سیستمی (که کنترل شده‌اند) توسط سطح کاربر فعال شوند. در صورتی که یک پردازه کاربر تلاش کند تله‌ای خارج از این محدوده را فعال کند، سیستم با یک استثنای حفاظتی (protection exception) به این عمل پاسخ می‌دهد و پردازش به وکتور شماره ۱۳ هدایت می‌شود. این سازوکار مانع از اجرای تله‌های غیرمجاز می‌شود و تنها به تله‌های مجاز اجازه اجرا می‌دهد.. در ادامه، توضیحاتی از دلایل این محدودیت و مکانیزم‌های پشت آن می نویسیم:

- محدودیت سطح دسترسی: در معماری 86x، برای دسترسی به تله‌ها، سطح اولویت جاری (CPL) پردازه باید برابر یا کمتر از سطح دسترسی توصیفگر تله (DPL) باشد. تله‌های سطح کرنل دارای DPL سطح 0 هستند و فقط پردازه‌هایی با این سطح دسترسی امکان فعال‌سازی آن‌ها را دارند. از سوی دیگر، تله‌های سطح کاربر که برای فراخوانی‌های سیستمی استفاده می‌شوند، تنها به DPL_USER (سطح 3) محدود شده‌اند.
- جلوگیری از دسترسی غیرمجاز به کرنل: یکی از مهم‌ترین دلایل محدود کردن دسترسی به تله‌ها جلوگیری از سوءاستفاده‌های احتمالی است. به عنوان مثال، اگر به یک برنامه کاربر اجازه داده شود که تله‌های کرنل را بدون محدودیت فعال کند، این برنامه می‌تواند از این قابلیت برای اجرای کد مخرب در سطح کرنل و دسترسی به تمامی منابع و امکانات سیستم استفاده کند. چنین سوءاستفاده‌ای می‌تواند به تغییرات غیرمجاز در داده‌ها و دسترسی به منابع حساس سیستم منجر شود که امنیت سیستم عامل را به طور جدی به خطر می‌اندازد.
- محافظت در برابر خطاهای کاربر: اگر برنامه‌های کاربر به تله‌های کرنل دسترسی داشتند، هر باگ در کد کاربر می‌توانست به کل سیستم آسیب برساند. سیستم عامل با محدود کردن دسترسی‌ها از تأثیر احتمالی این مشکلات بر پایداری و امنیت کرنل جلوگیری می کند.

- پرسش 4) در صورت تغییر سطح دسترسی، **ss** و **esp** روی پشته **Push** می‌شود. در غیر این‌صورت **Push** نمی‌شود. چرا؟

در سیستم‌های مبتنی بر معماری 86x، هنگامی که تله‌ای فعال می‌شود و تغییر سطح دسترسی از کاربر به کرنل رخ می‌دهد، ذخیره ی ss و esp روی پشته ضروری است. دلیل این امر موارد زیر است:

- استفاده از پشته مجزا برای کرنل: هر پردازه دارای دو پشته است؛ پشته کاربر و پشته کرنل. هنگامی که پردازه از سطح کاربر به سطح کرنل منتقل می‌شود (مثلاً در زمان یک فراخوانی سیستمی یا وقفه)، باید از پشته‌ای استفاده شود که فقط برای عملیات کرنل اختصاص داده شده است. این جداسازی، ایمنی اطلاعات پردازه در سطح کاربر و ثبات کرنل را تضمین می‌کند.
- ذخیره مقادیر SS و ESP: برای انتقال به پشته کرنل، مقادیر ثبات‌های SS (به معنی Stack Segment و مربوط به بخش پشته) و ESP (به معنی Extended Stack Pointer و اشاره‌گر به بالای پشته) که در حال حاضر به پشته کاربر اشاره می‌کنند، روی پشته جدید (پشته کرنل) ذخیره می‌شوند. این ذخیره‌سازی برای بازگشت به حالت قبلی پس از اتمام عملیات کرنل ضروری است؛ زیرا بدون آن، پردازنده نمی‌تواند به درستی به پشته کاربر بازگردد.
- عدم نیاز به ذخیره‌سازی در صورت عدم تغییر سطح دسترسی: اگر تله در همان سطح دسترسی فعلی اجرا شود (مثلاً اگر در سطح کرنل فعال شده و نیازی به تغییر سطح نباشد)، نیازی به ذخیره SS و ESP نیست؛ زیرا پردازنده همچنان از همان پشته قبلی استفاده می‌کند و تغییری در دسترسی و پشته مورد استفاده رخ نمی‌دهد.

(نکته ی تقریبا اضافی اما مهم) دلایل استفاده از دو پشته (پشته کاربر و پشته کرنل):

1. حفظ وضعیت پردازه: پشته کاربر شامل اطلاعات وضعیت پردازه قبل از تغییر سطح دسترسی است؛ ذخیره SS و ESP کمک می‌کند تا وضعیت پشته کاربر در بازگشت به آن بازیابی شود.
2. حفاظت از پشته کاربر: پشته کاربر ممکن است حاوی اطلاعات حساسی باشد؛ استفاده از پشته کرنل از افشای اطلاعات کاربر توسط عملیات کرنل جلوگیری می‌کند.
3. جلوگیری از سرریز پشته: پشته کرنل از احتمال سرریز پشته کاربر در عملیات کرنل و خرابی سیستم جلوگیری می‌کند.
4. جداسازی سطح دسترسی: استفاده از پشته‌های جداگانه به حفظ امنیت و پایداری سیستم کمک کرده و جدایی سطوح دسترسی را تقویت می‌کند.

- پرسش 5) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در **argptr**() بازه آدرس‌ها بررسی می‌گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد می‌کند؟ در

صورت عدم بررسی بازه ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی **read_sys**() اجرای سیستم را با مشکل روبرو سازد.

توابع argptr ،argint و argstr و argfd در 6xv برای دسترسی به پارامترهای فراخوانی‌های سیستمی استفاده می‌شوند که از فضای کاربر به فضای کرنل منتقل می‌شوند. این توابع وظیفه دارند تا مقدار یا آدرس آرگومان‌ها را از استک پروسه‌ی فراخوانی شده دریافت کنند و قبل از استفاده، بررسی کنند که این آدرس‌ها در محدوده مجاز حافظه پروسه قرار داشته باشند. این بررسی‌ها به دلایل امنیتی و جلوگیری از دسترسی غیرمجاز به حافظه سایر پردازه‌ها ضروری هستند.

1. تابع **argint(int n, int *ip)**

این تابع برای دریافت آرگومان‌های عددی از نوع int طراحی شده است. این تابع ابتدا آدرس آرگومان n-ام را با استفاده از رجیستر ESP محاسبه می‌کند. از آنجا که در استک، آرگومان‌های ورودی از بالای استک (آدرس بیشتر) به پایین (آدرس کمتر) ذخیره می‌شوند، و آدرس بازگشت آخرین مقدار ذخیره‌شده در استک است و آرگومان های ورودی تابع قبل از آن است، و آدرس سر استک هم در رجیستر ESP است، برای یافتن آدرس n-امین آرگومان، رابطه زیر استفاده می‌شود: $ptr = ESP + 4 + n * 4$

پس از به‌دست‌آوردن آدرس، این آدرس همراه اشاره گر به حافظه مدنظر برای مقدار int به تابع fetchint ارسال می شود. پس تابع fetchint برای دسترسی به مقدار موجود در آن آدرس فراخوانی می‌شود و بررسی می‌کند که آیا این آدرس معتبر است یا خیر (آیا آدرس ارسالی + 4 بایت که اندازه ی int است، در حافظه پردازه است یا نه). در صورت موفقیت، مقدار int آرگومان در متغیر مرجع داده‌شده در آرگومان دوم ذخیره می‌شود؛ در غیر این صورت، مقدار 1- بازمی‌گردد.

2. تابع **argptr(int n, char **pp, int size)**

تابع argptr برای دریافت آرگومان‌هایی از نوع اشاره‌گر (پوینتر) طراحی شده است که به بلاک حافظه‌ای از یک اندازه مشخص اشاره می‌کنند. این تابع سه آرگومان می‌گیرد: شماره آرگومان مورد نظر، اشاره‌گر به یک اشاره‌گر (char** pp) که در آن آدرس حافظه ذخیره می‌شود، و اندازه بلاک حافظه به بایت که باید دسترسی‌پذیر باشد. حالا ابتدا از argint برای دریافت مقدار آدرس پوینتر استفاده می‌شود. این آدرس ممکن است به داده‌ای در فضای کاربر اشاره کند که باید با دقت بررسی شود. سپس، بازه آدرس بررسی می‌شود تا اطمینان حاصل شود که کل بلاک

حافظه (از آدرس شروع تا آدرس شروع + size) در محدوده حافظه پردازه قرار دارد. در صورت معتبر بودن آدرس، اشاره‌گر به این آدرس در pp ذخیره می‌شود و مقدار صفر بازمی‌گردد. اگر این شرایط برقرار نباشند، مقدار 1- به عنوان خطا بازمی‌گردد.

3. تابع **argstr(int n, char **pp)**

تابع argstr برای دریافت آرگومان‌هایی از نوع رشته (string) طراحی شده است. این تابع دو آرگومان می‌گیرد: شماره آرگومان مورد نظر و اشاره‌گری به یک اشاره‌گر *char که آدرس رشته در آن ذخیره خواهد شد. حالا ابتدا، با استفاده از argint، آدرس ابتدای رشته دریافت می‌شود. سپس این آدرس به تابع fetchstr ارسال می‌شود. تابع fetchstr وظیفه دارد که بررسی کند آیا این آدرس در محدوده حافظه پردازه قرار دارد و رشته با یک کاراکتر نال (\0) خاتمه یافته است یا خیر.

مراحل انجام کار در fetchstr : ابتدا آدرس شروع رشته بررسی می‌شود که آیا در محدوده حافظه پردازه قرار دارد یا خیر. سپس از آدرس داده‌شده به سمت پایین پیمایش می‌شود و به دنبال کاراکتر نال می‌گردد. اگر کاراکتر نال یافت شود، رشته به‌درستی شناسایی شده و مقدار اشاره‌گر به رشته در pp ذخیره می‌شود و طول رشته (بدون نال) بازگردانده می‌شود .اما در صورتی که کاراکتر نال یافت نشود و به انتهای محدوده حافظه پردازه برسد، مقدار 1- بازگردانده می‌شود تا نشان دهد رشته نال-ترمینیتد نبوده و یا آدرس معتبر نیست.

4. تابع **argfd(int n, int *pfd, struct file **pf)**

تابع argfd برای دریافت آرگومان‌های نوع فایل دیسکریپتور (file descriptor) در یک فراخوانی سیستمی طراحی شده است. این تابع سه ورودی دارد: شماره پارامتر مورد نظر (مثلاً n) که مشخص می‌کند کدام پارامتر فایل دیسکریپتور مورد نظر است، و اشاره‌گری به یک متغیر int که مقدار فایل دیسکریپتور در آن ذخیره خواهد شد، و اشاره‌گری به یک اشاره‌گر از نوع *struct file که به فایل مرتبط با فایل دیسکریپتور اشاره می‌کند. عملکرد این تابع به این صورت است که ابتدا argint برای دریافت مقدار فایل دیسکریپتور از استک فراخوانی می‌شود. اگر argint نتواند مقدار صحیحی را برگرداند (به عنوان مثال اگر پارامتر n خارج از محدوده باشد)، تابع argfd با مقدار 1- به عنوان خطا پایان می‌یابد. سپس مقدار فایل دیسکریپتور بررسی می‌شود: این مقدار نباید کمتر از صفر باشد، همچنین نباید از حداکثر تعداد فایل‌های مجاز (NOFILE) بیشتر باشد، و در نهایت، چک می‌شود که آیا این فایل دیسکریپتور به یک فایل باز در جدول فایل‌های باز پردازه اشاره می‌کند یا خیر. برای این کار، از -()myproc

[fd]ofile< استفاده می‌شود تا اطمینان حاصل شود فایل دیسکریپتور به یک فایل معتبر اشاره می‌کند. در صورتی که تمام شرایط برقرار باشند و فایل دیسکریپتور معتبر باشد، تابع مراحل زیر را انجام می‌دهد: اگر اشاره‌گر pfd معتبر باشد، مقدار فایل دیسکریپتور (fd) در آن ذخیره می‌شود، و اگر اشاره‌گر pf معتبر باشد، اشاره‌گر به فایل (f) در آن قرار داده می‌شود. در نهایت، اگر تمام بررسی‌ها موفقیت‌آمیز باشند، تابع مقدار صفر را به‌عنوان نتیجه موفقیت‌آمیز بودن عملیات بازمی‌گرداند. اما اگر هر کدام از این بررسی‌ها به دلیل نامعتبر بودن فایل دیسکریپتور شکست بخورند، تابع 1- را برمی‌گرداند.

اهمیت بررسی بازه آدرس‌ها در **argptr**

تابع argptr به این دلیل بازه آدرس‌ها را بررسی می‌کند که اطمینان حاصل کند دسترسی به حافظه تنها در محدوده مجاز پروسه فعلی انجام می‌شود. در صورت عدم بررسی بازه آدرس، پردازه ممکن است به حافظه پروسه‌های دیگر یا حتی کرنل دسترسی پیدا کند که می‌تواند به مشکلات امنیتی منجر شود؛ مانند دسترسی غیرمجاز به داده‌های حساس پردازه‌های دیگر و توانایی افشا یا دستکاری آن ها یا ایجاد اختلال در اجرای سایر پردازه‌ها یا کرش کردن کرنل و ناپایداری آن.

مثال از فراخوانی سیستمی **sys_read**

در فراخوانی سیستمی sys_read، آرگومان‌ها به‌صورت زیر دریافت و بررسی می‌شوند:

```
int
sys_read(void)      Reza Chehreghani, 2 days ago • Upload base model
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

در این کد، ابتدا argfd مقدار file descriptor را دریافت و بررسی می‌کند. سپس argint مقدار n (حداکثر بایت‌هایی که باید خوانده شود) را دریافت می‌کند. در نهایت، argptr به کار می‌رود تا اطمینان حاصل شود که

محدوده آدرسی که p (بافر) به آن اشاره می‌کند در حافظه مجاز پردازه قرار دارد و می‌توان به تعداد n بایت از آن دسترسی داشت.

اگر این بررسی بازه آدرس انجام نشود و برنامه‌ای تابع read (این فراخوانی مربوط به تابع read است) را با یک مقدار max بسیار بزرگ فراخوانی کند، سیستم‌عامل هنگام خواندن داده‌ها از فایل و نوشتن آن‌ها در بافر p از محدوده حافظه پردازه خارج شده و ممکن است داده‌ها را در حافظه سایر پردازه‌ها یا بخش‌های حساس کرنل ذخیره کند. این امر می‌تواند مشکلات گفته شده را بوجود آورد یا حتی منجر به سرریز شدن بافر یا ایجاد trap شود.

بنابراین، بررسی بازه آدرس در argptr در sys_read و توابع مشابه، از دسترسی غیرمجاز، افشای داده‌ها و ناپایداری سیستم جلوگیری کرده و امنیت سیستم را تضمین می‌کند.

- بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط **gdb**

دستور bt در GDB برای نمایش پشته (Backtrace) استفاده می‌شود. این دستور تمامی فریم‌های تابع در حال اجرا را لیست می‌کند، از جمله آدرس‌های بازگشت، نام توابع، و محل (فایل و خط) مربوطه.

```
(gdb) bt
#0  syscall () at syscall.c:173
#1  0x80105ebd in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80105c6a in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

- Frame 0: تابع فعلی که در آن اجرا متوقف شده، syscall است. این تابع در فایل syscall.c و خط 173 قرار دارد.
- Frame 1: این تابع، trap، با آرگومان 4tf=0x8dffefb فراخوانی شده است. در فایل trap.c و خط 43 قرار دارد. این نشان می‌دهد که خطای مربوطه در حین اجرای یک تله (trap) رخ داده است.
- Frame 2: تابع alltraps در خط 20 از فایل اسمبلی trapasm.S قرار دارد. این معمولاً بخشی از مدیریت استثناهای پایین‌سطح سیستم‌عامل است.
- Frame 3: یک فریم ناشناخته است که در آدرس حافظه 4x8dffefb0 قرار دارد. GDB قادر به شناسایی این بخش از کد یا ارتباط آن با کدی خاص نیست.
- خط آخر: این پیام نشان می‌دهد که پشته ممکن است خراب شده باشد (Corrupt Stack).

دستور down برای حرکت در فریم‌های پشته (Stack Frames) استفاده می‌شود. وقتی یک باگ یا توقف در تابعی رخ می‌دهد، GDB اطلاعات مربوط به فریم فعلی و فریم‌های بالا را در پشته نشان می‌دهد. وقتی در فریم‌های بالاتر پشته (مثل تابع فراخواننده) هستیم و می‌خواهیم به فریم پایین‌تر (توابع فرزند یا توابعی که مستقیماً اجرا شده‌اند) حرکت کنیم، از down استفاده می‌کنیم. هر بار اجرای دستور ما را یک فریم پایین‌تر می‌برد.

```
(gdb) up
#1  0x80105ebd in trap (tf=0x8dffefb4) at trap.c:43
43              syscall();
(gdb) p tf->eax
$1 = 5
```

در اینجا با دستور up به فریم ۱ می رویم. مقدار رجیستر eax برابر ۵ می باشد. اما شماره فراخوانی سیستمی getpid() برابر ۱۱ می‌باشد که برابر نمی‌باشند. دلیل آن است که برای اجرای یک برنامه سطح کاربر تعدادی فراخوانی سیستمی قبل و بعد آن اجرا می‌شوند. مثلا برای اجرای کامند ورودی، پردازه sh موظف است تا با فراخوانی سیستمی fork یک پردازه جدید بسازد و سپس خود پردازه جدید با فراخوانی سیستمی exec برنامه را لود می‌کند و در اتمام کارش با فراخوانی سیستمی exit به کار خودش پایان می‌دهد. همچنین پس از اتمام اجرای پردازه جدید، دوباره پردازه sh با فراخوانی سیستمی علامت $ را در صفحه چاپ کرده و با یک فراخوانی سیستمی دیگر منتظر کامند جدید می‌ماند.

```
(gdb) p tf->eax
$8 = 11
```

همانطور که مشاهده می‌کنید مقدار رجیستر eax در چاپ هشتم برابر با شماره فراخوانی سیستمی getpid() می‌باشد.

- ارسال آرگومان های فراخوانی های سیستمی

برای این بخش ما برای ارسال آرگومان ها از ثبات ها استفاده می‌کنیم به این صورت که در برنامه سطح کاربر آرگومان را در یکی از ثبات ها قرار داده و در تابع مربوطه برای سیستم کال، آرگومان را از آن ثبات دریافت می‌کنیم. در فایل sysproc.c یک تابع به صورت زیر داریم که مقدار عدد را از رجیستر ebx می‌گیرد و به تابع سازنده پالیندروم که در فایل proc.c وجود دارد پاس می‌دهد. (نحوه ساختن یک سیستم کال در بولت بعدی توضیح داده شده است)

```
86
87  int sys_create_palindrome(void)
88  {
89    int num;
90    struct proc *p = myproc();
91    num = p->tf->ebx;
92    create_palindrome(num);
93    return 0;
94  }
```

تابع create_palindrome در فایل proc.c به صورت زیر پالیندروم عدد را می‌سازد.

```
542  void
543  create_palindrome(int num){
544    int ans = num;
545    while (num != 0)
546    {
547      ans = ans * 10 + num % 10;
548      num /= 10;
549    }
550    cprintf("%d\n", ans);
551  }
```

دقت شود که definition این تابع در فایل def.h قرار دارد.

برای تست این برنامه هم برنامه ای در سطح کاربر می‌نویسیم که کاربر با وارد کردن دستور پالیندروم و یک عدد به دنبال اون مثل palindrome 123 پالیندروم عدد مورد نظر را بگیرد.

همانطور که می‌بینید این برنامه در فایل palindrome.c به این صورت نوشته شده است که با استفاده از تابع 1create_palindrome با استفاده از کد اسمبلی که در آن قرار دارد محتوای رجیستر ebx را در perv_val می‌ریزد و مقدار num را در این رجیستر قرار می‌دهد. سپس سیستم کال مربوطه را صدا می‌زند و بعد از اتمام سیستم کال، مقدار رجیستر ebx را به مقدار قبلی خود برمی‌گرداند.

- نحوه اضافه کردن فراخوانی‌های سیستمی

برای اضافه کردن فراخوانی سیستمی به سیستم عامل، باید چند جا آن‌ها را اضافه کنیم:

- user.h

```
void create_palindrome(void);
int move_file(const char *, const char *);
int sort_syscalls(int);
int list_all_processes(void);
int get_most_invoked_syscall(int);
```

- syscall.h

```
#define SYS_create_palindrome 22
#define SYS_move_file 23
#define SYS_sort_syscalls 24
#define SYS_get_most_invoked_syscall 25
#define SYS_list_all_processes 26
```

- syscall.c

```
extern int sys_create_palindrome(void);
extern int sys_move_file(void);
extern int sys_sort_syscalls(void);
extern int sys_list_all_processes(void);
extern int sys_get_most_invoked_syscall(void);

static int (*syscalls[])(void) = {
…
[SYS_create_palindrome]   sys_create_palindrome,
[SYS_move_file]           sys_move_file,
[SYS_sort_syscalls]       sys_sort_syscalls,
[SYS_get_most_invoked_syscall]
sys_get_most_invoked_syscall,
[SYS_list_all_processes] sys_list_all_processes,
};
```

- usys.S

```
SYSCALL(create_palindrome)
SYSCALL(move_file)
SYSCALL(sort_syscalls)
SYSCALL(get_most_invoked_syscall)
SYSCALL(list_all_processes)
```

1. پیاده سازی فراخوانی سیستمی انتقال فایل

تابع sys_move_file را در فایل sysfile.c که برای فراخوانی‌های سیستمی File-system می‌باشد تعریف می‌کنیم.

در ابتدا متغیرهای محلی مورد نیاز را اضافه می‌کنیم.

```
int sys_move_file(void)
{
 struct inode *ip, *dp_new, *dp_old;
 struct dirent de;
 char name[DIRSIZ], *src_file, *dest_dir;
 uint off;
```

برای خواندن آرگومان‌ها از استک یعنی فایل منبع و آدرس مقصد، از تابع argstr استفاده می‌کنیم که آرگومان فراخوانی سیستمی به اندازه کلمه n را به عنوان نشانگر رشته‌ای واکشی می‌کند.

```
 // Fetch arguments from user space
 if (argstr(0, &src_file) < 0 || argstr(1, &dest_dir) < 0)
   return -1;
```

در ابتدا هر فراخوانی سیستمی File_system باید تابع begin_op را صدا بزنیم.

```
 begin_op();
```

ما برای آنکه بتوانیم یک فایل را انتقال دهیم یعنی از یک دایرکتوری حذف و به یک دایرکتوری دیگر ببریم، نیازی نداریم در فایل اصلی تغییری ایجاد کنیم و فقط کافی است تا دایرکتوری جدید را به فایل لینک کنیم و لینک دایرکتوری فعلی به فایل را حذف کنیم. برای این کار ما نیاز به دایرکتوری فعلی فایل داریم. بنابراین از تابع nameiparent استفاده می‌کنیم تا inode مربوط به آن را به ما بدهد. این تابع همچنین رشته name را نیز پر می‌کند که اسم فایل می‌باشد.

```
 // Look up source directory
 if ((dp_old = nameiparent(src_file, name)) == 0){
   end_op();
   return -1;
 }
```

حال برای آنکه بتوانیم فایل را از دایرکتوری مبدا حذف کنیم، باید بدانیم که فایل در کدام entry آن قرار دارد. برای این کار از تابع dirlookup استفاده می‌کنیم که مقدار off را برای ما پر می‌کند. همچنین این تابع، inode مربوط به فایل را نیز می‌دهد.

```
 // Look up source file
 if((ip = dirlookup(dp_old, name, &off)) == 0){
   end_op();
   return -1;
 }
```

حال که inode دایرکتوری مبدا و فایل را داریم، فقط inode دایرکتوری مقصد می‌ماند که آن را به کمک تابع namei بدست می‌آوریم که با گرفتن آدرس یک فایل به صورت رشته، inode آن را برمیگرداند.

```
// Look up destination directory
if((dp_new = namei(dest_dir)) == 0){
  end_op();
  return -1;
}
```

حال که همه inode های لازم را داریم. به سراغ انجام عملیات انتقال می‌رویم. اول دایرکتوری مقصد را به فایل لینک می‌کنیم. این کار را با تابع dirlink انجام می‌دهیم که اسم فایل و شماره inode آن و دایرکتوری مقصد را به عنوان آرگومان می‌گیرد. البته قبل چک می‌کنیم که روی یک device قرار داشته باشند. نکته‌ای که وجود دارد آن است که برای انجام عملیات روی یک inode باید آن را در حین عملیات lock کنیم. که در ابتدا با تابع ilock دایرکتوری مقصد را قفل کرده و در انتها هم با iunlockput قفل را باز کرده و ارجاع به آن inode در حافظه را رها می‌کنیم.

```
// Link the file in the destination directory
ilock(dp_new);
if(dp_new->dev != ip->dev || dirlink(dp_new, name, ip->inum) < 0){
  iunlockput(dp_new);
  end_op();
  return -1;
}
iunlockput(dp_new);
```

حال باید لینک دایرکتوری فعلی فایل را حذف کنیم. از آنجایی که تابع آماده برای dirunlink نداریم، باید به صورت دستی این کار را انجام دهیم. بنابراین یک فایل از ساختار dirent که برای entry ها در دایرکتوری استفاده می‌شود درست می‌کنیم و تمام خانه‌های آن را برابر صفر می‌کنیم. سپس این dirent را در محل off دایرکتوری مبدا می‌نویسیم. با این کار در واقع لینک را حذف کرده‌ایم. با حذف این لینک در واقع این گونه به نظر می‌رسد که فایل از دایرکتوری مبدا حذف شده است.

```
// Unlink the file from its original location
memset(&de, 0, sizeof(de));
ilock(dp_old);
if(writei(dp_old, (char*)&de, off, sizeof(de)) != sizeof(de)){
  iunlockput(dp_old);
  end_op();
  return -1;
}
iunlockput(dp_old);
```

2. پیاده سازی فراخوانی سیستمی مرتب سازی فراخوانی های یک پردازه

برای پیاده سازی این تابع، ابتدا باید تغییراتی ایجاد کنیم تا فراخوانی‌های سیستمی یک پردازه را ذخیره کند. بنابراین به ساختار struct proc چند متغیر اضافه می‌کنیم. از آرایه به طول ۶۴ برای نگه داری فراخوانی‌های سیستمی استفاده می‌کنیم. یعنی حداکثر ۶۴ تا فراخوانی سیستمی را می‌توانیم بشماریم.

```
#define MAX_SYSCALLS 64  // Maximum number of distinct system calls to track
struct proc {
 ...
 int syscalls_count;                // count number of system calls
 int syscall_num[MAX_SYSCALLS];     // system calls number
 char *syscall_name[MAX_SYSCALLS];  // system calls name
};
```

هنگام مقداردهی اولیه یک پردازه، تعداد فراخوانی‌های سیستمی را برابر صفر قرار می‌دهیم.

```
 // Initialize number of system calls
 p->syscalls_count = 0;
```

برای چاپ نام یک فراخوانی سیستمی، یک آرایه از اسامی فراخوانی‌های سیستمی تشکیل می‌دهیم تا اسامی را نگه دارد و به ازای هر فراخوانی سیستمی، نام آن را هم در ساختار proc ذخیره می‌کنیم.

```
static char *syscall_names[] = {
 [SYS_fork]    "fork",
 [SYS_exit]    "exit",
 [SYS_wait]    "wait",
 [SYS_pipe]    "pipe",
 [SYS_read]    "read",
 [SYS_kill]    "kill",
 [SYS_exec]    "exec",
 [SYS_fstat]   "fstat",
 [SYS_chdir]   "chdir",
 [SYS_dup]     "dup",
 [SYS_getpid]  "getpid",
 [SYS_sbrk]    "sbrk",
 [SYS_sleep]   "sleep",
 [SYS_uptime]  "uptime",
 [SYS_open]    "open",
 [SYS_write]   "write",
 [SYS_mknod]   "mknod",
 [SYS_unlink]  "unlink",
 [SYS_link]    "link",
 [SYS_mkdir]   "mkdir",
 [SYS_close]   "close",
 [SYS_create_palindrome]         "create_palindrome",
 [SYS_move_file]                 "move_file",
```

```
 [SYS_sort_syscalls]              "sort_syscalls",
 [SYS_get_most_invoked_syscall]
"get_mosttt_invoked_syscall",
 [SYS_list_all_processes]         "list_all_processes",
};
```

برای ذخیره سازی فراخوانی‌های سیستمی، قبل از اجرای تابع آن، به شرط آنکه تعداد فراخوانی‌های سیستمی بیشتر از ۶۴ نشود، شماره و نام آن را در انتهای آرایه اضافه می‌کنیم.

```
void
syscall(void)
{
 int num;
 struct proc *curproc = myproc();

 num = curproc->tf->eax;
 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
   // Track the system call
   if (curproc->syscalls_count < MAX_SYSCALLS) {
     curproc->syscall_num[curproc->syscalls_count] = num;
     curproc->syscall_name[curproc->syscalls_count] = syscall_names[num];
     curproc->syscalls_count++;
   }

   curproc->tf->eax = syscalls[num]();

 } else {
   cprintf("%d %s: unknown sys call %d\n",
           curproc->pid, curproc->name, num);
   curproc->tf->eax = -1;
 }
}
```

حال برای فراخوانی سیستمی sort_syscalls، تابع sys_sort_syscalls را در فایل sysproc.c که مربوط به فراخوانی‌های سیستمی پردازه‌ها می‌باشد، تعریف می‌کنیم.این تابع آرگومان ورودی یعنی pid را از استک خوانده و تابع sort_syscalls را صدا می‌زند که در فایل proc.c تعریف شده است. دلیل این کار این است که اطلاعات پردازه‌ها در متغیر ptable قرار دارد که این متغیر فقط در فایل proc.c در دسترس می‌باشد.

```
int sys_sort_syscalls(void) {
 int pid;

 if (argint(0, &pid) < 0)
   return -1;
 return sort_syscalls(pid);
}
```

در تابع sort_syscalls، ابتدا پردازه‌ها را با تابع acquire قفل می‌کنیم. سپس به دنبال pid مورد نظر می‌گردیم. حال که پردازه را پیدا کردیم، آرایه‌ای که فراخوانی‌های سیستمی را در آن ذخیره می‌کنیم را به کمک bubble sort مرتب می‌کنیم. سپس آن‌ها را چاپ می‌کنیم. البته چک می‌کنیم که اگر از یک فراخوانی سیستمی چند بار استفاده شده است، آن را فقط یک بار چاپ کنیم.

```
int
sort_syscalls(int pid)
{
 int i, j, tmp_num, last_one = 0;
 char *tmp_name;
 struct proc *p;

 // Find the process with the given PID
 acquire(&ptable.lock);
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
   if(p->pid == pid){
     // Sort system calls
     for(i = 0; i < p->syscalls_count-1; i++){
       for(j = 0; j < p->syscalls_count-i-1; j++){
         if(p->syscall_num[j] > p->syscall_num[j+1]){
           tmp_num = p->syscall_num[j];
           tmp_name = p->syscall_name[j];

           p->syscall_num[j] = p->syscall_num[j+1];
           p->syscall_name[j] = p->syscall_name[j+1];

           p->syscall_num[j+1] = tmp_num;
           p->syscall_name[j+1] = tmp_name;
         }
       }
     }

     cprintf("System calls for process %d:\n", pid);
     for(i = 0; i < p->syscalls_count-1; i++)
       if(p->syscall_num[i] != last_one){
         cprintf("Syscall #%d: %s\n", p->syscall_num[i], p->syscall_name[i]);
         last_one = p->syscall_num[i];
       }

     release(&ptable.lock);
     return 0;
   }
 }
 release(&ptable.lock);
 return -1;
}
```

3. پیاده‌سازی فراخوانی سیستمی برگرداندن بیشترین فراخوانی سیستم برای یک فرآیند خاص

برای پیاده سازی این تابع، ابتدا باید تغییراتی ایجاد کنیم تا تعداد هر فراخوانی سیستمی برای یک پردازه ذخیره شود. بنابراین به ساختار struct proc یک متغیر اضافه می‌کنیم. از یک آرایه برای این کار استفاده می‌کنیم

که در ایندکس i ام آن تعداد فراخوانی های سیستم کال شماره i ذخیره شده است. بنابراین برای تمام سیستم کال ها، تعداد فراخوانی آن ها را داریم.

```
int syscall_invokes[MAX_SYSCALLS];    //Array to count each syscall
```

همچنین برای هر پروسس که شروع می‌شود، در تابع allocproc این آرایه را به صفر اینیشیالایز می‌کنیم.

```
allocproc(void)

// Initialize number of each system call
memset(p->syscall_invokes, 0, sizeof(p->syscall_invokes));
```

حال تابع sys_get_most_invoked_syscall را به صورت زیر در sysproc.c تعریف می‌کنیم تا بعد از دریافت آرگومان pid به وسیله argint، تابع get_most_invoked_syscall را که در proc.c است صدا می‌زنیم.

```
103
104  int sys_get_most_invoked_syscall(){
105    int pid;
106    if (argint(0, &pid) < 0)
107      return -1;
108    return get_most_invoked_syscall(pid)
109  }
110
```

تابع get_most_invoked_syscall در proc.c به صورت زیر تعریف شده است.

```
595  int
596  get_most_invoked_syscall(int pid){
597    char *syscall_name = "";
598    int syscall_invokes = 0;
599    int syscall_num = 0;
600    struct proc *p;
601    acquire(&ptable.lock);
602    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
603      if(p->pid == pid){
604        for(int i =0; i< p->syscalls_count;i++){
605          if(syscall_invokes <= p->syscall_invokes[p->syscall_num[i]]){
606
607              syscall_invokes = p->syscall_invokes[p->syscall_num[i]];
608              syscall_name = p->syscall_name[i];
609              syscall_num = p->syscall_num[i];
610          }
611        }
612        if (syscall_invokes > 0){
613          cprintf("Most invoked syscall for process %d is %s with %d invokes\n",
614            p->pid, syscall_name, syscall_invokes);
615          release(&ptable.lock);
616          return syscall_num;
617        }
618      }
619    }
620    release(&ptable.lock);
621    return -1;
622  }
623
```

این تابع ابتدا در ptable به دنبال پروسس با آیدی مورد نظر می‌گردد. سپس زمانی که آن را یافت، بر روی سیستم کال های آن یک حلقه می‌زند و برای هر کدام اگر تعداد invoke های آن بیشتر از syscall_invokes بود، آن را به عنوان بیشترین فراخوانی سیستم برای یک فرآیند خاص انتخاب می‌کند. سپس اگر این فراخوانی سیستمی را یافت آیدی پروسس، نام و تعداد فراخوانی های آن سیستم کال را پرینت می‌کند و شماره آن سیستم کال را برمی‌گرداند. در غیر این صورت منفی یک برمی‌گرداند.

برای تست این فراخوانی سیستمی، برنامه سطح کاربر زیر را نوشتیم که ابتدا چک می‌کند طرز صدا زدن این برنامه درست است یا خیر سپس pid نوشته شده توسط کاربر را ذخیره می‌کند و سیستم کال را با آن pid فراخوانی می‌کند. اگر این سیستم کال نتواند به هر دلیلی این بیشترین فراخوانی سیستم برای یک فرآیند خاص را برگرداند، ارور مورد نظر نمایش داده می‌شود.

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf(2, "Usage: most_invoked_syscall <pid>\n");
    exit();
  }

  int pid = atoi(argv[1]);

  if (get_most_invoked_syscall(pid) < 0) {
    printf(2, "Error: Could not print most invoked system call for pid %d\n", pid);
  }

  exit();
}
```

4. پیاده‌سازی فراخوانی سیستمی لیست کردن پردازه ها

اول تابع sys_list_all_processes را به صورت زیر در فایل sysproc.c تعریف می‌کنیم.

```c
int sys_list_all_processes(void){
  return list_all_processes();
}
```

این تابع، تابع list_all_processes در فایل proc.c را فراخوانی می‌کند.

در فایل proc.c ، آیدی ها و در واقع تمام اطلاعات پروسس ها در ptable ذخیره شده اند. با استفاده از قطعه کد زیر در ptable را ابتدا lock می‌کنیم و سپس به دنبال برنامه های در حال اجرا می‌گردیم و طبق خواسته دستور پروژه، pid و تعداد سیستم کال های آن ها را که در p->syscalls_count قرار دارد ، پرینت می‌کنیم. در آخر هم چک میکنیم که اگر پروسس در حال اجرایی پیدا نکردیم -1 در غیر این صورت 0 برگردانیم (که کلا نیازی نبود)

```
619
620  int
621  list_all_processes(){
622    int p_count = 0;
623    struct proc *p;
624
625    acquire(&ptable.lock);
626    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
627      if (p->state == RUNNING){
628          p_count++;
629          cprintf("Process %d with %d Syscalls\n", p->pid, p->syscalls_count);
630        }
631      }
632    release(&ptable.lock);
633    return (p_count == 0) ? -1 : 0;
634  }
635
```

برای تست این سیستم کال یک برنامه سطح کاربر نوشتیم که صرفا در آن این سیستم کال صدا زده می‌شود. کد این برنامه به صورت زیر است.

```
C list_all_processes.c > main(int, char * [])
1   #include "types.h"
2   #include "user.h"
3
4   int main(int argc, char *argv[])
5   {
6       list_all_processes();
7       exit();
8   }
```