Operating systems

**Project2 report**

**رضا چهرقانی 810101401**

**امیر نداف فهمیده 810101540**

**مصطفی کرمانی‌نیا 810101575**

**مخزن گيتهاب اين پروژه‌:**

[**https://github.com/mostafa-kermaninia/OS\_LAB\_P2**](https://github.com/mostafa-kermaninia/OS_LAB_P2)

**آخرین کامیت:**

horizontal line

* **پرسش 1) کتابخانه‌های سطح کاربر در XV6، 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 XV6، 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 xv6، 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 xv6 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 xv6 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 xv6 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) :



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:

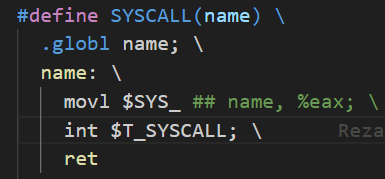
1. **ulib.c**

این فایل کد منبع ulib.o است و شامل توابع کمکی‌ای مانند strcpy، strcmp، strlen، memset، strchr، gets، stat، atoi 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 open، fstat 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.

1. **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:



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 64 int را فراخوانی می کند) که باعث ایجاد یک وقفه نرم‌افزاری شده و کنترل را به کرنل منتقل می‌کند.

1. **printf.c**

فایل printf.c کد منبع printf.o و شامل توابعی مانند putc، printint 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.

1. **umalloc.c**

این فایل کد منبع umalloc.o است و شامل توابعی مانند malloc، free 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.

1. **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.

1. **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.

1. **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 xv6، امکان اجرای تله‌های دیگر توسط سطح دسترسی کاربر (DPL\_USER) وجود ندارد و اگر یک پردازه بخواهد interrupt دیگری را فعال کند، xv6 به آن این اجازه را نمی دهد و با یک استثنای protection exception مواجه می شوند. در xv6، 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 x86، 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 می‌کند.
* **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 x86، 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 argint، argptr و argstr و argfd در xv6 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.

### 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:

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.

### 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.

### 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.

### تابع (argfd(int n, int \*pfd, struct file \*\*pf

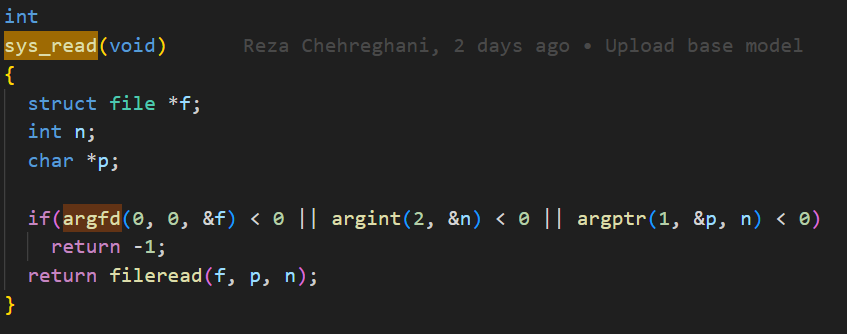
تابع argfd برای دریافت آرگومان‌های نوع فایل دیسکریپتور (file descriptor) در یک فراخوانی سیستمی طراحی شده است. این تابع سه ورودی دارد: شماره پارامتر مورد نظر (مثلاً n) که مشخص می‌کند کدام پارامتر فایل دیسکریپتور مورد نظر است، و اشاره‌گری به یک متغیر int که مقدار فایل دیسکریپتور در آن ذخیره خواهد شد، و 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()->ofile[fd] 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:



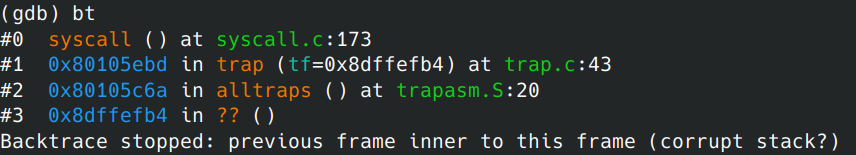
در این کد، ابتدا argfd مقدار file descriptor را دریافت و بررسی می‌کند. سپس argint مقدار n (حداکثر بایت‌هایی که باید خوانده شود) را دریافت می‌کند. در نهایت، argptr به کار می‌رود تا اطمینان حاصل شود که محدوده آدرسی که p (بافر) به آن اشاره می‌کند در حافظه مجاز پردازه قرار دارد و می‌توان به تعداد n بایت 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.

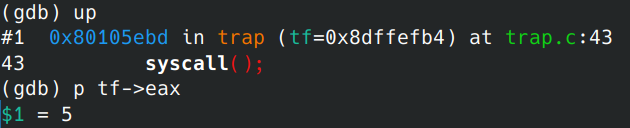
* **بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb**

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.



* 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 tf=0x8dffefb4 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 0x8dffefb4 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.



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.



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