«گزارش کار پروژه دوم»

«آزمایشگاه سیستم عامل»

کسری کاشانی 810101490

البرز محمودیان 810101514

نرگس بابالار 810101557

- مقدمه

پرسش 1- **کتابخانه های سطح کاربر در xv6، برای ایجاد ارتباط میان برنامه های کاربر و کرنل به کار میروند. این کتابخانه ها شامل توابعی هستند که از فراخوانی های سیستمی استفاده میکنند تا دسترسی به منابع سخت افزاری و نرمافزاری سیستم عامل ممکن شود. با تحلیل فایلهای موجود در متغیر ULIB در xv6، توضیح دهید که چگونه این کتابخانه ها از فراخوانی های سیستمی بهره میبرند؟ همچنین، دلیل استفاده از این فراخوانی ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه ها را شرح دهید.**

در متغیر ULIB، مجموعه‌ای از فایل‌های کتابخانه‌ای وجود دارد که توابعی ارائه می‌دهند که از فراخوانی‌های سیستمی(System Calls) استفاده می‌کنند. فراخوانی سیستمی مکانیزمی است که برنامه‌های کاربر را قادر می‌سازد با کرنل ارتباط برقرار کنند. کتابخانه‌های سطح کاربر در xv 6 معمولاً توابعی را شامل می‌شوند که از فراخوانی‌های سیستمی خاصی بهره می‌برند.

برای مثال:

- **فراخوانی‌های مدیریت فایل** :توابعی مانند open(), read(), write(), وclose()، که دسترسی به فایل‌ها را فراهم می‌کنند.
- **فراخوانی‌های مدیریت پردازش** :توابعی مانند fork(), exec(), وwait()که مدیریت پردازش‌ها را انجام می‌دهند.
- **فراخوانی‌های مدیریت حافظه** :توابعی مانند mmap() یا munmap() برای دسترسی به حافظه.
- **فراخوانی‌های مدیریت زمان‌بندی و سیگنال‌ها** :توابعی که زمان‌بندی یا مدیریت سیگنال‌ها را ممکن می‌سازند.

این توابع، هنگام فراخوانی توسط برنامه کاربر، در نهایت به فراخوانی‌های سیستمی منجر می‌شوند که توسط کرنل پردازش می‌شوند و نتایج را به برنامه کاربر بازمی‌گردانند.

دلایل استفاده از فراخوانی‌های سیستمی:

- **امنیت و حفاظت** : برنامه‌های کاربر نباید مستقیماً به سخت‌افزار یا حافظه کرنل دسترسی داشته باشند. استفاده از فراخوانی‌های سیستمی موجب می‌شود که عملیات حساس، نظیر دسترسی به حافظه یا فایل‌های سیستم، تحت کنترل کرنل انجام شود.

- **ایجاد انتزاع** :فراخوانی‌های سیستمی باعث می‌شوند که جزئیات سطح پایین مربوط به دسترسی و مدیریت منابع برای برنامه‌های کاربر مخفی بماند. به‌این‌ترتیب، برنامه‌ها می‌توانند از توابع سطح بالا استفاده کنند بدون اینکه نگران پیچیدگی‌های ارتباط با سخت‌افزار یا سیستم‌عامل باشند.
- **مدیریت منابع** :کرنل وظیفه مدیریت منابع سیستم مانندCPU ، حافظه و دستگاه‌های ورودی و خروجی را دارد. با استفاده از فراخوانی‌های سیستمی، برنامه‌ها می‌توانند به شکلی استاندارد و کنترل‌شده از این منابع استفاده کنند.

تأثیر فراخوانی‌های سیستمی بر عملکرد و قابلیت حمل برنامه‌ها:

- **عملکرد** :هر فراخوانی سیستمی یک تغییر حالت از فضای کاربر به فضای کرنل نیاز دارد که می‌تواند زمان‌بر باشد و باعث تأخیر شود. این تغییر حالت به‌خصوص در سیستم‌هایی با تعداد بالای فراخوانی‌های سیستمی، می‌تواند منجر به کاهش عملکرد برنامه‌ها شود. با این‌حال، کتابخانه‌های سطح کاربر در xv 6ممکن است از کش یا سایر تکنیک‌ها برای کاهش تعداد فراخوانی‌های سیستمی غیرضروری بهره ببرند، که به بهبود عملکرد کمک می‌کند.
- **قابلیت حمل** :استفاده از فراخوانی‌های سیستمی در برنامه‌های کاربر باعث می‌شود که برنامه‌ها بهAPI های سیستم‌عامل وابسته شوند. این امر می‌تواند موجب محدودیت‌هایی در قابلیت حمل (portability) برنامه‌ها شود، زیرا فراخوانی‌های سیستمی معمولاً به سیستم‌عامل خاصی وابسته هستند و برای اجرا در سیستم‌عامل‌های دیگر نیاز به تغییر دارند. برای کاهش این وابستگی، بسیاری از سیستم‌عامل‌ها، از جمله xv6، کتابخانه‌های استانداردی مانند POSIX را فراهم می‌کنند که به صورت انتزاعی عمل می‌کنند و قابلیت حمل برنامه‌ها را بهبود می‌بخشند.
در نهایت، کتابخانه‌های سطح کاربر در xv 6از فراخوانی‌های سیستمی به‌عنوان پل ارتباطی میان برنامه‌های کاربر و کرنل استفاده می‌کنند. این فراخوانی‌ها امنیت، حفاظت و مدیریت بهینه منابع را فراهم کرده و دسترسی برنامه‌های کاربر به منابع سیستم را استاندارد می‌سازند. اما به‌کارگیری آن‌ها می‌تواند تأثیرات منفی بر عملکرد و قابلیت حمل برنامه‌ها داشته باشد که با استفاده از تکنیک‌های بهینه‌سازی و استانداردسازی تا حدودی قابل حل است.

پرسش 2- **فراخوانی های سیستمی تنها روش برای تعامل برنامه های کاربر با کرنل نیستند. چه روشهای دیگری در لینوکس وجود دارند که برنامه های سطح کاربر میتوانند از طریق آنها به کرنل دسترسی داشته باشند؟ هر یک از این روشها را به اختصار توضیح دهید.**

فایل‌های دستگاه (Device Files) در /dev

در لینوکس، دستگاه‌های سخت‌افزاری مانند دیسک‌ها، پرینترها، و کارت‌های شبکه به صورت فایل‌های ویژه‌ای در دایرکتوری /dev نمایش داده می‌شوند. برنامه‌های کاربر می‌توانند با خواندن یا نوشتن به این فایل‌ها به دستگاه‌های مربوطه دسترسی پیدا کنند.
مثال :فایل /dev/sda به عنوان دیسک اصلی سیستم عمل می‌کند و می‌توان برای خواندن یا نوشتن داده‌ها مستقیماً با آن ارتباط برقرار کرد. این دسترسی معمولاً از طریق توابع سطح بالای کتابخانه‌ای مانند open(), read(), و write() امکان‌پذیر است، اما به کرنل وابسته است که عملیات فیزیکی را انجام دهد.

سوکت‌های Netlink

سوکت‌های Netlink مکانیزمی برای ارتباط بین کرنل و برنامه‌های کاربر در لینوکس هستند. این سوکت‌ها به برنامه‌های سطح کاربر اجازه می‌دهند تا اطلاعات را با کرنل به اشتراک بگذارند یا پارامترهای شبکه‌ای را تنظیم کنند.

کاربردها: Netlinkبیشتر در مدیریت شبکه به کار می‌رود. به عنوان مثال، برای تنظیمات مسیریابی شبکه یا تغییر جدول‌های فایروال، برنامه‌ها می‌توانند از سوکت‌های Netlinkاستفاده کنند.

اینترفیس ioctl

ioctl (Input/Output Control) یک مکانیزم برای ارسال دستورات خاص از برنامه‌های کاربر به درایورهای دستگاه است. این مکانیزم به برنامه‌های کاربر اجازه می‌دهد که عملیات پیچیده‌ای را بر روی دستگاه‌های خاص انجام دهند که ممکن است فراتر از خواندن یا نوشتن ساده باشد.

مثال :یک برنامه می‌تواند از ioctl برای تغییر تنظیمات دستگاهی مانند کارت گرافیک استفاده کند، یا پارامترهایی مانند وضوح تصویر را تنظیم کند.

Epoll و Inotify

- epoll : یک مکانیزم برای مدیریت چندین توصیف‌گر فایل (file descriptors) است که به برنامه‌های کاربر اجازه می‌دهد رویدادهای ورودی/خروجی (I/O) را به شکل کارآمدی مدیریت کنند. این روش برای برنامه‌های سرور که با تعداد زیادی ارتباط کار می‌کنند (مانند وب‌سرورها) بسیار مناسب است و در مقایسه با روش‌های سنتی کارایی بیشتری دارد.
- inotify : یک اینترفیس برای نظارت بر تغییرات فایل‌ها و دایرکتوری‌ها است. برنامه‌ها می‌توانند از inotify برای دریافت اطلاعیه در مورد تغییرات فایل‌ها یا دایرکتوری‌های خاص استفاده کنند. این مکانیزم برای برنامه‌هایی که نیاز دارند در لحظه تغییرات سیستم فایل را رصد کنند، کاربردی است.

کتابخانه‌های Shared Memory و IPC (ارتباط بین پردازشی)

سیستم‌عامل لینوکس از چندین روش ارتباط بین پردازشی (IPC) مانند حافظه مشترک (Shared Memory)، پیام‌رسانی(Message Queues) ، و لوله‌های نام‌گذاری‌شده (Named Pipes)پشتیبانی می‌کند. این مکانیزم‌ها به برنامه‌های مختلف اجازه می‌دهند تا بدون نیاز به فراخوانی‌های مستقیم سیستمی با یکدیگر و حتی با کرنل ارتباط برقرار کنند.

- **مثال** :در حافظه مشترک، دو برنامه می‌توانند به بخشی از حافظه که بین آن‌ها مشترک است، دسترسی داشته باشند. این روش کارایی بالایی دارد زیرا برنامه‌ها به‌طور مستقیم به حافظه دسترسی پیدا می‌کنند.

فایل‌های مجازی در سیستم‌فایل proc/و sys/

سیستم‌فایل‌های proc/و sys/از مکانیزم‌های مجازی هستند که اطلاعات مربوط به وضعیت کرنل، پردازش‌ها، و تنظیمات سیستم را به صورت فایل‌های مجازی ارائه می‌دهند. این سیستم‌ها به برنامه‌های کاربر اجازه می‌دهند بدون نیاز به فراخوانی‌های سیستمی به داده‌های سیستم دسترسی پیدا کنند و برخی پارامترها را پیکربندی کنند.

- **proc/**این سیستم‌فایل اطلاعاتی مانند پردازش‌های جاری، وضعیت حافظه، و اطلاعات CPU را در اختیار می‌گذارد. به عنوان مثال، فایل /proc/cpuinfo اطلاعات مربوط به پردازنده را ارائه می‌دهد.

- **/sys**: سیستم‌فایل /sys اطلاعات و تنظیمات مربوط به دستگاه‌ها و درایورها را مدیریت می‌کند. به عنوان مثال، برای تغییر برخی پارامترهای دستگاه‌ها، می‌توان از فایل‌های موجود در /sys استفاده کرد.

این روش‌ها هر یک مزایای خاص خود را دارند و به برنامه‌های کاربر اجازه می‌دهند که با کرنل تعامل کنند و از امکانات سیستم‌عامل استفاده کنند. هرچند فراخوانی‌های سیستمی همچنان مهم‌ترین مکانیزم برای تعامل مستقیم با کرنل هستند، این روش‌های جایگزین، انعطاف بیشتری برای برنامه‌ها و بهینه‌سازی عملکرد در برخی از سناریوها فراهم می‌کنند.

پرسش 3- **آیا باقی تله ها را نمیتوان با سطح دسترسی USER_DPL فعال نمود؟ چرا؟**

خیر، سایر تله‌ها (مانند تله‌های مرتبط با وقفه‌ها و استثناها) را نمی‌توان با سطح دسترسی USER_DPL فعال کرد. دلایل این محدودیت به امنیت، پایداری سیستم و مدیریت منابع برمی‌گردد. در ادامه، به‌طور کامل به دلایل این محدودیت می‌پردازیم:

امنیت سیستم‌عامل

تله‌ها به کرنل اجازه می‌دهند تا به صورت مستقیم با سخت‌افزار و منابع حساس سیستم تعامل داشته باشد. اگر برنامه‌های کاربر بتوانند با سطح دسترسی USER_DPL تله‌هایی مثل وقفه‌های سخت‌افزاری یا استثناهای خاص را فعال کنند، ممکن است از این دسترسی برای ایجاد تغییرات در وضعیت سخت‌افزار یا منابع سیستم‌عامل استفاده کنند. این تغییرات می‌توانند منجر به دسترسی غیرمجاز به داده‌های حساس، خرابی در حافظه، یا کنترل منابع سیستم شوند و امنیت سیستم‌عامل را به خطر بیندازند. محدود کردن دسترسی به تله‌ها در سطح KERNEL_DPL به محافظت از این منابع و جلوگیری از سوءاستفاده کمک می‌کند.

پایداری و ثبات سیستم‌عامل

تله‌ها و وقفه‌ها، به‌ویژه آن‌هایی که مستقیماً به مدیریت منابع سیستم مانند پردازش‌ها، حافظه و ورودی/خروجی‌ها (I/O) مربوط می‌شوند، برای کنترل داخلی سیستم‌عامل استفاده می‌شوند. دسترسی برنامه‌های کاربر به این تله‌ها می‌تواند عملکرد سیستم را ناپایدار کند، زیرا برنامه‌های کاربر ممکن است بدون توجه به وضعیت کلی سیستم، این تله‌ها را فعال کنند. به عنوان مثال، وقفه‌های تایمر و I/O نیاز به کنترل دقیق کرنل دارند تا از تداخل در اجرای فرآیندها و دسترسی به منابع جلوگیری شود. با محدود کردن

دسترسی به این تله‌ها، سیستم‌عامل می‌تواند اطمینان حاصل کند که فقط کرنل، که آگاهی کامل از وضعیت سیستم دارد، می‌تواند به این وقفه‌ها پاسخ دهد.

مدیریت کارآمد منابع و افزایش بهره‌وری

وقتی تله‌های سیستم مانند وقفه‌های ورودی/خروجی(I/O) ، مدیریت حافظه، و تایمرها به طور انحصاری توسط کرنل کنترل شوند، سیستم‌عامل می‌تواند منابع را به شکل بهینه‌تری مدیریت کند. به‌عنوان مثال، مدیریت وقفه‌های سخت‌افزاری به صورت مستقیم بر عملکرد دستگاه‌های متصل و مدیریت حافظه اثر می‌گذارد و نیاز به برنامه‌ریزی دقیق دارد. اگر برنامه‌های کاربر بدون نظارت کرنل به این وقفه‌ها دسترسی داشته باشند، ممکن است منابع سخت‌افزاری مانند CPU و حافظه به صورت غیر مؤثر و ناکارآمد استفاده شوند. به این ترتیب، محدود کردن دسترسی به این تله‌ها باعث بهبود کارایی و مدیریت بهینه منابع می‌شود.

جلوگیری از سوءاستفاده و آسیب‌های احتمالی

اگر برنامه‌های کاربر بتوانند تله‌هایی مانند وقفه‌های سخت‌افزاری یا استثناهای کرنل را فعال کنند، ممکن است از این امکان برای ایجاد حملات یا سوءاستفاده‌های امنیتی استفاده کنند. به‌عنوان مثال، دسترسی آزاد به تله‌ها می‌تواند باعث شود که برنامه‌های کاربر به‌طور مداوم وقفه‌ها را فعال کنند و منابع سیستم را به شدت مصرف کنند. این امر می‌تواند منجر به کاهش کارایی سیستم، توقف سرویس‌های حیاتی و حتی قفل شدن سیستم شود. در نتیجه، محدود کردن دسترسی به تله‌ها به سطح کرنل (KERNEL_DPL) باعث می‌شود که برنامه‌های کاربر نتوانند از این تله‌ها برای آسیب رساندن به سیستم یا سایر برنامه‌ها استفاده کنند.

این محدودیت‌ها تضمین می‌کنند که کرنل، که سطح بالاتری از کنترل و دسترسی به منابع دارد، به‌تنهایی مسئول مدیریت تله‌های سیستم‌عامل باشد. با این روش، سیستم‌عامل می‌تواند به شکلی پایدار و امن عمل کند و از بروز مشکلات ناشی از دخالت برنامه‌های کاربر در فرآیندهای حساس جلوگیری شود.

پرسش 4- در صورت تغییر سطح دسترسی، ssو esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟

در صورت تغییر سطح دسترسی، مانند زمانی که یک تله یا فراخوانی سیستمی از سطح کاربر (User Mode) به سطح کرنل (Kernel Mode) رخ می‌دهد، رجیسترهای SS (Stack Segment) و

ESP (Extended Stack Pointer)روی پشته کرنل Pushمی‌شوند.

اگر تغییری در سطح دسترسی رخ ندهد، این مقادیر Pushنمی‌شوند. زیرا:

- **عدم نیاز به ذخیره‌سازی SSو ESPدر صورت ثابت بودن سطح دسترسی**

در صورتی که تله یا فراخوانی سیستمی در همان سطح دسترسی (مثلاً از کرنل به کرنل) رخ دهد و نیازی به تغییر سطح نباشد، نیازی به تغییر پشته نیست. پردازنده به‌طور طبیعی به اجرای کد در همان سطح دسترسی ادامه می‌دهد و همان پشته در حال استفاده باقی می‌ماند. از این رو، نیازی به ذخیره SSو ESPنیست و سیستم با صرفه‌جویی در فضای پشته، عملکرد بهینه‌تری خواهد داشت.

- **سوئیچ به پشته کرنل در صورت تغییر سطح دسترسی**

زمانی که پردازنده از سطح کاربر به سطح کرنل منتقل می‌شود، نیاز به استفاده از پشته‌ای امن و مخصوص به کرنل دارد، زیرا پشته کاربر در سطحی پایین‌تر و با محدودیت‌های امنیتی متفاوتی قرار دارد. برای اینکه پردازنده بتواند در بازگشت به سطح کاربر به پشته قبلی بازگردد، باید SSو ESPرا، که نشان‌دهنده‌ی پشته کاربر هستند، روی پشته کرنل Pushکند. این کار تضمین می‌کند که پردازنده بتواند پس از اتمام عملیات کرنل، به‌درستی به سطح کاربر و پشته آن بازگردد.

- **بهینه‌سازی و جلوگیری از مصرف غیرضروری پشته**

هر Pushاضافی باعث مصرف فضای پشته و تأخیر در اجرای کد می‌شود. اگر تغییر سطح دسترسی وجود نداشته باشد، این Pushها غیرضروری خواهند بود و فقط باعث افزایش مصرف منابع می‌شوند. با عدم Pushکردن SSو ESPدر این شرایط، سیستم بهینه‌تر عمل می‌کند و اجرای تله‌ها سریع‌تر می‌شود.

بنابراین SSو ESPتنها زمانی روی پشته ذخیره می‌شوند که انتقال از سطح کاربر به سطح کرنل صورت گیرد. در این حالت، این مقادیر برای بازگشت به سطح کاربر نیاز هستند. اما اگر تغییر سطح دسترسی رخ ندهد، نیازی به ذخیره این مقادیر نیست، چون پشته در حال استفاده ثابت می‌ماند و عملکرد سیستم بهینه‌تر می‌شود.

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

مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی باز ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی ()sys_read اجرای سیستم را با مشکل روبرو سازد.

در سیستم‌عامل xv6، توابع دسترسی به پارامترهای فراخوانی سیستمی (مانند argintو argptr)به کرنل کمک می‌کنند تا به پارامترهای ارسال‌شده توسط برنامه کاربر در فراخوانی سیستمی دسترسی داشته باشد. این توابع پارامترها را از پشته سطح کاربر می‌خوانند و آن‌ها را برای استفاده در کرنل آماده می‌کنند. به عنوان مثال، تابع sys_exec برای اجرای یک برنامه جدید از این توابع استفاده می‌کند تا پارامترهای argvو initرا از پشته برداشته و به کرنل منتقل کند.

()Argint :این تابع دو ورودی دارد. ورودی اول، شماره پارامتر و ورودی دوم که به صورت reference by passداده میشود یک متغیر با تایپ int است، که در واقع مقدار پارامتر در ورودی دوم ذخیره می شود. این تابع با صدا زدن تابع() fetchinit و

دادن مقدارn*4+4+(myproc()->tf->esp)به عنوان ورودی اول آن، آدرس پارامتر خواسته شده را ساخته و محتوای آن را برمیگرداند. اگر این هدف با موفقیت انجام شود مقدار صفر و اگر این پارامتر وجود نداشته باشد و عملیات با موفقیت انجام نشود)آرگومان غیر مجاز(خروجی -۱ برگردانده می شود)

()Argptr :این تابع سه ورودی دارد. ورودی اول، شماره پارامتر خواسته شده است و ورودی دوم به صورت reference by pass و سایز پارامتر به عنوان ورودی سوم به تابع داده می شود که ورودی دوم محتوای پارامترهایی به شکل pointer مانند آرایه را در ورودی دوم ذخیره می کند. این تابع با صدا زدن تابع() argint آدرس خانه اول این پارامتر از جنس اشاره گر را بر می گرداند. در صورتی که این تابع با موفقیت عملیات را انجام دهد مقدار صفر و در غیر این صورت مقدار -۱ را برمی گرداند.

دلیل بررسی بازه آدرس‌ها در argptr

تابع argptrوظیفه دارد که آدرس ارسال‌شده از برنامه کاربر را بررسی کند و اطمینان یابد که این آدرس در فضای معتبر حافظه کاربر قرار دارد. این بررسی ضروری است زیرا اگر آدرس غیرمعتبری (خارج از محدوده فضای کاربر یا متعلق به فضای کرنل) به تابع ارسال شود و این بررسی انجام نشود، مشکلات جدی امنیتی و پایداری در سیستم به وجود می‌آید.

مشکلات امنیتی ناشی از تجاوز از بازه معتبر آدرس‌ها

تجاوز از بازه معتبر آدرس‌ها می‌تواند مشکلات امنیتی زیر را ایجاد کند:

1. **دسترسی به داده‌های حساس کرنل** :اگر آدرس ارسال‌شده توسط برنامه کاربر خارج از محدوده مجاز باشد و به بخش‌های حساس حافظه کرنل اشاره کند، کاربر می‌تواند به اطلاعات کرنل دسترسی پیدا کند یا آن‌ها را تغییر دهد.
2. **خرابی و عدم پایداری سیستم** :ارسال آدرس‌های نامعتبر می‌تواند باعث خطاهای دسترسی به حافظه (segmentation fault) شود که منجر به کرش کردن برنامه یا حتی کل سیستم‌عامل می‌شود.
3. **امکان حملات امنیتی** :برنامه‌های مخرب می‌توانند با ارسال آدرس‌های خاص، به داده‌ها و کدهای حساس دسترسی پیدا کنند و حملاتی مانند افزایش سطح دسترسی (Privilege Escalation)یا دستکاری داده‌های کرنل را انجام دهند.

مثال مشکل در sys_readدر صورت عدم بررسی بازه در argptr

فرض کنید تابع sys_readبرای خواندن داده از یک فایل، به آدرس بافری که توسط برنامه کاربر ارسال شده نیاز دارد و این آدرس از طریق argptr به دست می‌آید. اگر argptrبازه آدرس‌ها را بررسی نکند و یک برنامه مخرب، آدرس غیرمعتبری خارج از فضای کاربر (مثلاً یک آدرس کرنل) را به sys_readبفرستد، چند سناریوی زیر ممکن است رخ دهد:

- **نشت اطلاعات کرنل** :اگر sys_readتلاش کند که داده‌ها را به آدرس غیرمعتبر ارسال کند، ممکن است بخش‌هایی از حافظه کرنل به برنامه کاربر نشت پیدا کند و برنامه بتواند اطلاعات حساس را دریافت کند.
- **کرش سیستم** :اگر آدرس غیرمعتبر به فضای نامعتبر حافظه اشاره کند، دسترسی sys_readبه این آدرس باعث ایجاد یک خطای دسترسی به حافظه (segmentation fault)می‌شود که می‌تواند باعث کرش سیستم‌عامل شود و در نتیجه پایداری سیستم به خطر بیافتد.

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

پس از make clean کردن، دستور make qemu-gdb را در ترمینال زده و یک ترمینال دیگر باز می کنیم. در این ترمینال جدید، gdb را می زنیم و فایل kernel را می دهیم. سپس با دستور b syscall یک breakpoint برای تابع syscall در فایل syscall.h ایجاد میکنیم. حال برنامه سطح کاربر زیر را اجرا می کنیم:

```c
// gdb_test.c > main()
#include "types.h"
#include "user.h"

int main()
{
    int pid = getpid();

    printf(1, "The process id is: %d\n", pid);

    exit();
}
```

با زدن دستور bt، تمام فراخوانی های تابع مورد نظر تا رسیدن به breakpoint نمایش داده می شوند. در واقع محتویات داخل call stack نمایش داده می شود که برای نمایش روند اجرای برنامه استفاده می شود. می دانیم که هنگام فراخوانی شدن یک تابع، یک frame stack حاوی اطلاعات آن تابع مانند return address بر روی call stack پوش می شود و و با دستور bt محتویات آن stack نمایش داده می شود. در واقع پس از اجرا شدن دستور int64، مقدارش در vector64 در فایل vector.s اضافه می شود و سپس به alltraps در فایل trapsasm.s می رود و frame trap را ساخته و آن را در آن stack پوش می کند.

```
Kasra@Kasra:~/Desktop/xv6-public-master$ gdb
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: File "/home/Kasra/Desktop/xv6-public-master/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/Kasra/Desktop/xv6-public-master/.gdbinit
line to your configuration file "/home/Kasra/.config/gdb/gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/Kasra/.config/gdb/gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
--Type <RET> for more, q to quit, c to continue without paging--c
        info "(gdb)Auto-loading safe path"
(gdb) file kernel
Reading symbols from kernel...
(gdb) target remote:26000
Remote debugging using :26000
0x0000fff0 in ?? ()
(gdb) b syscall
Breakpoint 1 at 0x80105b10: file syscall.c, line 145.
(gdb) r
The "remote" target does not support "run".  Try "help target" or "continue".
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       struct proc *curproc = myproc();
(gdb) bt
```

حال دستور down را اجرا می کنیم تا به frame stack بالاتر در bt برویم و به تابع calee برسیم که قبل تر صدا زده شده است. اما تابع syscall در خود هیچ تابع دیگری را صدا نزده و لذا calee نداریم و چیزی در frame stack نمایش داده نخواهد شد.

```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
```

سپس با زدن دستور up به یک frame stack پایین تر در bt می رویم و به trap می رسیم.

```
(gdb) up
#1  0x80106d7d in trap (tf=0x8dffffb4) at trap.c:43
43            syscall();
```

حال محتوای رجیستر eax در tf را با دستور print myproc()->tf->eax چاپ می کنیم. این رجیستر حاوی شماره ی سیستم کال صدا زده شده ی فعلی می باشد.

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145         struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$87 = 1
(gdb) c
Continuing.
```

همانطور که مشاهده می شود، مقدار ذخیره شده در این رجیستر برابر با 7 است که با شماره فراخوانی سیستمی ()fork یعنی 1 برابر نیست. علت این است که قبل از رسیدن به دستور ()getpid، فراخوانی سیستمی های متعدد دیگری نیز صورت گرفته اند. لذا چندین بار با دستور c، continue می کنیم و دوباره محتوای eax را چاپ می کنیم تا آنکه به دستور ()fork با شماره فراخوانی سیستمی 1 برسیم.

```
(gdb) up
#1  0x80106d7d in trap (tf=0x8dffffb4) at trap.c:43
43            syscall();
(gdb) print myproc()->tf->eax
$1 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145         struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$2 = 15
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145         struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$3 = 10
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145         struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$4 = 10
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145         struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$5 = 16
(gdb) c
Continuing.
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145         struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$87 = 1
(gdb) c
Continuing.
```

چند فراخوانی سیستمی دیگر در این حین صورت گرفته اند. از جمله سیستم کال exec() با شماره ی 7، سیستم کال open با شماره ی 15، سیستم کال dup() با شماره ی 10، سیستم کال write با شماره ی 16 و در نهایت سیستم کال fork() با شماره ی 1.

- پیاده سازی فراخوانی های سیستمی

ابتدا در فایل syscall.h به ازای اضافه شدن هر سیستم کال جدید یک عدد به ان اختصاص می دهیم که مثلا اعداد 1 تا 21 از قبل وجود داشتند پس ما به ازای اضافه شدن سیستم کال های {create_palindrome, move_file, sort_syscalls, get_most_invoked_syscall, list_all_processes} اعداد 22 تا 26 را به انها اختصاص می دهیم.

syscall.h > SYS_list_all_processes

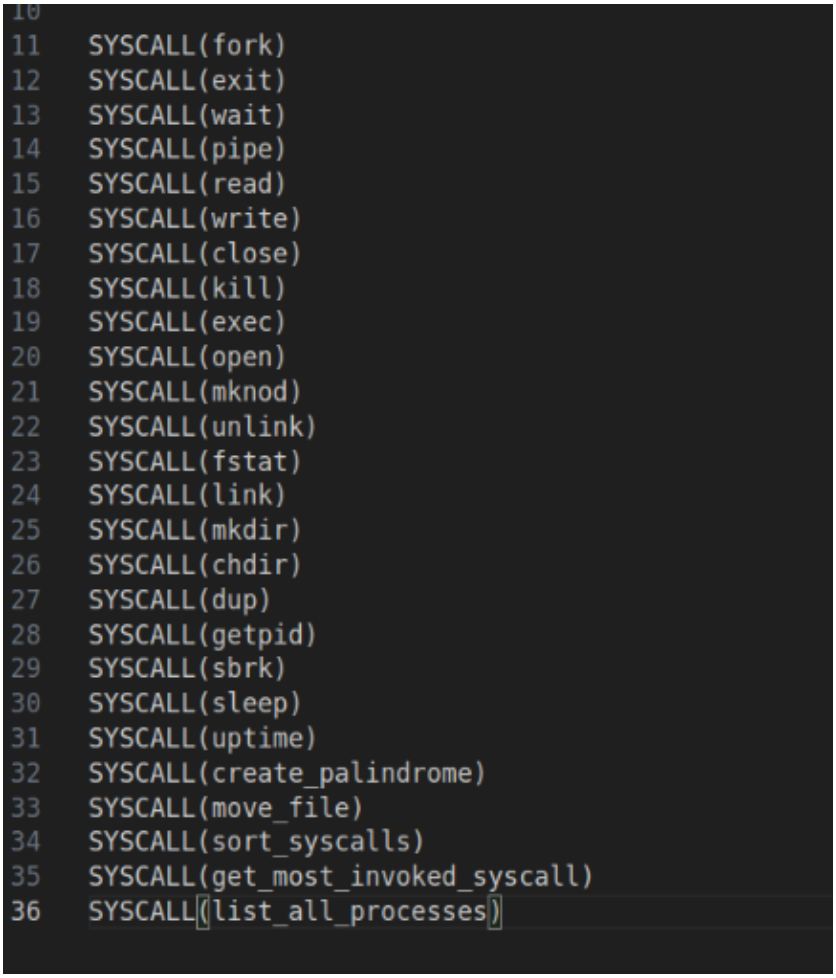
```
1   // System call numbers
2   #define SYS_fork    1
3   #define SYS_exit    2
4   #define SYS_wait    3
5   #define SYS_pipe    4
6   #define SYS_read    5
7   #define SYS_kill    6
8   #define SYS_exec    7
9   #define SYS_fstat   8
10  #define SYS_chdir   9
11  #define SYS_dup     10
12  #define SYS_getpid  11
13  #define SYS_sbrk    12
14  #define SYS_sleep   13
15  #define SYS_uptime  14
16  #define SYS_open    15
17  #define SYS_write   16
18  #define SYS_mknod   17
19  #define SYS_unlink  18
20  #define SYS_link    19
21  #define SYS_mkdir   20
22  #define SYS_close   21
23  #define SYS_create_palindrome 22
24  #define SYS_move_file 23
25  #define SYS_sort_syscalls 24
26  #define SYS_get_most_invoked_syscall 25
27  #define SYS_list_all_processes 26
```

سپس در فایل syscall.c ، prototype هر سیستم کال را طبق عکس زیر در زیر سیستم کال های قبلی به کد خود اضافه می کنیم.

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_create_palindrome(void);
extern int sys_move_file(void);
extern int sys_sort_syscalls(void);
extern int sys_get_most_invoked_syscall(void);
extern int sys_list_all_processes(void);
```

همچنین به ازای اضافه شدن هر سیستم کال جدید ان سیستم کال را به ارایه ی syscalls در فایل syscall.c اضافه می کنیم:

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[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,
};
```



حال به ازای اضافه کردن هر سیستم کال جدید که که باید در فایل proc.c قرار گیرد ، در فایل defs.h نیز ان را اضافه می کنیم (حواسمان باشد که در این قسمت تنها سیستم کال هایی را که باید در فایل proc.c قرار گیرند را در این قسمت اضافه می کنیم . مثلا سیستم کال move_file که مربوط به فایل ها است و در proc.c نباید باشد را در این قسمت اضافه نمی کنیم)

```
104  //PAGEBREAK: 16
105  // proc.c
106  int             cpuid(void);
107  void            exit(void);
108  int             fork(void);
109  int             growproc(int);
110  int             kill(int);
111  struct cpu*     mycpu(void);
112  struct proc*    myproc();
113  void            pinit(void);
114  void            procdump(void);
115  void            scheduler(void) __attribute__((noreturn));
116  void            sched(void);
117  void            setproc(struct proc*);
118  void            sleep(void*, struct spinlock*);
119  void            userinit(void);
120  int             wait(void);
121  void            wakeup(void*);
122  void            yield(void);
123  int             create_palindrome(int);
124  int             sort_syscalls(int);
125  int             get_most_invoked_syscall(int);
126  int             list_all_processes(void);
127  struct proc*    get_proc_by_pid(int);
128
```

سپس به ازای هر سیستم کال جدید prototype ان را در فایل user.h مانند عکس زیر اضافه می کنیم تا user قابلیت استفاده از این ها را داشته باشد:

```
C user.h > ...
 1   struct stat;
 2   struct rtcdate;
 3
 4   // system calls
 5   int fork(void);
 6   int exit(void) __attribute__((noreturn));
 7   int wait(void);
 8   int pipe(int*);
 9   int write(int, const void*, int);
10   int read(int, void*, int);
11   int close(int);
12   int kill(int);
13   int exec(char*, char**);
14   int open(const char*, int);
15   int mknod(const char*, short, short);
16   int unlink(const char*);
17   int fstat(int fd, struct stat*);
18   int link(const char*, const char*);
19   int mkdir(const char*);
20   int chdir(const char*);
21   int dup(int);
22   int getpid(void);
23   char* sbrk(int);
24   int sleep(int);
25   int uptime(void);
26   int create_palindrome(int);
27   int move_file(const char*, const char*);
28   int sort_syscalls(int);
29   int get_most_invoked_syscall(int);
30   int list_all_processes(void);
31
```

حال به ازای هر سیستم کال نیز در ادامه یک فایل تست می نویسیم که بتوانیم هر سیستم کال را با کد تست ان که یک فایل c است تست کنیم. یعنی 5 تا فایل جدید با نام های زیر به folder خود اضافه می کنیم:

- Create_palindrome_test.c, move_file_test.c, sort_syscalls_test.c, get_most_invoked_syscall_test.c, list_all_processes_test.c

پس باید این 5 فایل را در makefile خود در قسمت های extra و uprogs اضافه کنیم همچنین یک فایل نیز برای تست gbd زده ایم (gbd_test.c) که این فایل را نیز در این دو قسمت اضافه می کنیم.

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c gdb_test.c create_palindrome_test.c move_file_test.c sort_syscalls_test.c get_most_invoked_syscall_test.c list_all_processes_test.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

Makefile

```
163  # that disk image changes after first build are persistent until clean.  More
164  # details:
165  # http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
166  .PRECIOUS: %.o
167
168  UPROGS=\
169      _cat\
170      _echo\
171      _forktest\
172      _grep\
173      _init\
174      _kill\
175      _ln\
176      _ls\
177      _mkdir\
178      _rm\
179      _sh\
180      _stressfs\
181      _usertests\
182      _wc\
183      _zombie\
184      _encode\
185      _decode\
186      _gdb_test\
187      _create_palindrome_test\
188      _move_file_test\
189      _sort_syscalls_test\
190      _get_most_invoked_syscall_test\
191      _list_all_processes_test\
192
```

در xv6 ، فراخوانی‌های سیستمی (system calls) بر اساس نوع عملکردشان در فایل‌های مختلفی نگهداری می شوند .دو فایل اصلی برای فراخوانی‌های سیستمی در xv6 عبارتند از sysproc.c و sysfile.c هرکدام از این فایل‌ها برای مدیریت نوع خاصی از عملکردها استفاده می‌شوند.

فایل sysproc.cمخصوص فراخوانی‌های سیستمی مربوط به کنترل و مدیریت فرآیندها است. این شامل عملیات‌هایی می‌شود که فرآیندها را ایجاد، مدیریت یا به‌نوعی با آن‌ها تعامل می‌کنند.

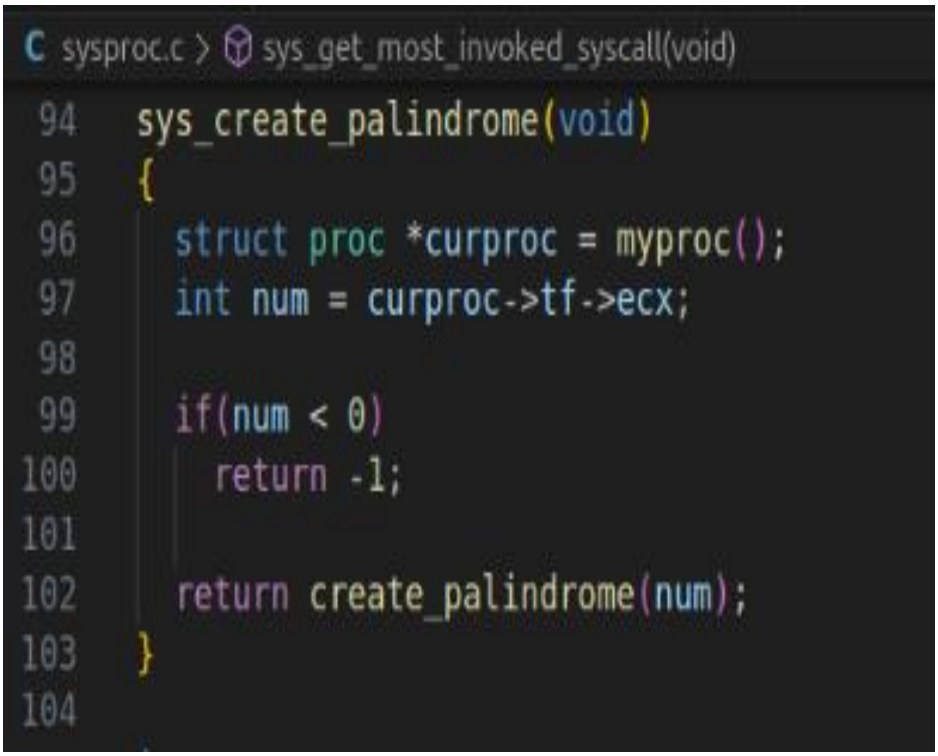
فایل sysfile.c برای عملیات فایل و دایرکتوری است. این فایل شامل تعاملات با سیستم فایل می‌شود، از جمله خواندن، نوشتن، ایجاد، حذف و لینک دهی فایل‌ها.

اگر یک فراخوانی سیستمی فرآیندی را مدیریت می‌کند، حافظه فرآیند را تغییر می‌دهد باید در sysproc.c قرار گیرد مانند سیستم کال های create_palindrome و sort_syscalls و get_most_invoked_syscalls و list_all_procceses

اگر یک فراخوانی سیستمی با فایل‌ها، دایرکتوری‌ها، دسکریپتورهای فایل، یا inodeها (ساختارهای داده‌ای که فایل‌ها را در سیستم فایل نشان می‌دهند) تعامل دارد، باید در sysfile.c قرار گیرد مانند سیستم کال move_file

- ارسال آرگومان های فراخوانی های سیستمی

حال در کد زیر در فایل sysproc.c هنگامی که این سیستم کال create_palindrome رخ می دهد این تابع فراخوانی می شود. Process ی که این سیستم کال را صدا می زند ، عددی که می خواهد palindrome ان را دریافت کند را در رجیستر ecx قرار می دهد و این تابع که در sysproc.c است ابتدا عدد داده شده را در خط 97 از روی این رجیستر برمی دارد و تابع create_palindrome را که در فایل proc.c است را فراخوانی می کند و این عدد را به عنوان ورودی به این تابع پاس می دهد. درواقع کاربرد رجیستر ecx برای پاس دادن ارگومان تابع به سیستم کال مربوطه است .



حال در فایل proc.c به ازای عدد گرفته شده ابتدا ان را به یک ارایه ای از char تبدیل می کنیم که معکوس عدد داده شده است مثلا عدد 1234 را به {‘1’, ‘2’, ‘3’, ‘4’} تبدیل می کند و در num_str ذخیره می کند. حال در ارایه ی دیگری به نام palindrome_str که طول ان دو برابر ارایه ی قبلی است حاصل را ذخیره می کنیم. به اینگونه که در نیمه ی اول برعکس شده ی num_str و در نیمه ی دوم ان خود num_str را قرار می دهیم به اینگونه که palindrome_str به صورت {‘1’, ‘2’, ‘3’, ‘4’, ‘4’, ‘3’, ‘2’, ‘1’} در خواهد امد و در نهایت ان را در قالب یک string برای کاربر نشان می دهیم.

```c
int
create_palindrome(int num)
{
  int number_of_digits = 0, temp_num = num;
  char num_str[128], palindrome_str[256];

  if(num == 0)
  {
    num_str[number_of_digits] = (num % 10) + '0';
    number_of_digits ++;
  }

  while(num > 0)
  {
    num_str[number_of_digits] = (num % 10) + '0';
    number_of_digits ++;
    num = num / 10;
  }

  num_str[number_of_digits] = '\0';

  for(int i = 0; i < number_of_digits; i++)
  {
    palindrome_str[i] = num_str[number_of_digits - i - 1];
    palindrome_str[number_of_digits + i] = num_str[i];
  }

  palindrome_str[2 * number_of_digits] = '\0';

  cprintf("Palindrome of %d is %s\n", temp_num, palindrome_str);

  return 0;
}
```

حال یک کد c می زنیم تا این سیستم کال را در ان تست کنیم به صورتی که کاربر عدد خود را به عنوان ارگومان به برنامه می دهد و برنامه، palindrome_str درست شده را برای کاربر در سطح کرنل چاپ می کند.

```c
C create_palindrome_test.c > main(int, char * [])
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4
5   int main(int argc, char *argv[])
6   {
7       if(argc != 2)
8       {
9           printf(2, "ERROR: Too many arguments!\n");
10          exit();
11      }
12
13      int num = atoi(argv[1]);
14
15      int result = create_palindrome(num);
16
17      if(result == -1)
18      {
19          printf(2, "ERROR: Create palindrome failed!\n");
20          exit();
21      }
22
23      exit();
24  }
```

```
$ create_palindrome_test 12345
Palindrome of 12345 is 1234554321
$
```

1- پیاده سازی فراخوانی سیستمی انتقال فایل

سیستم کال move_file را چون مرتبط با کار با فایل است در sysfile.c مطابق زیر تعریف میکنیم:

```
int
sys_move_file(void)
{
    struct dirent de;
    struct inode *src_inode, *dest_inode;
    char *src_file, *dest_dir;
    char filename[DIRSIZ];

    // Get the source and destination paths from user space
    if(argstr(0, &src_file) < 0 || argstr(1, &dest_dir) < 0)
        return -1;

    // Start a file system transaction
    begin_op();

    // Check if the source file exists
    if((src_inode = namei(src_file)) == 0)
    {
        end_op();
        return -1;
    }

    ilock(src_inode);

    // Extract filename from the source path
    safestrcpy(filename, src_file, DIRSIZ);

    // Open the destination directory
    if((dest_inode = namei(dest_dir)) == 0)
    {
        iunlockput(src_inode);
        end_op();
        return -1;
    }

    ilock(dest_inode);

    // Link the file to the destination directory
    if(dirlink(dest_inode, filename, src_inode->inum) < 0)
    {
        iunlockput(dest_inode);
        iunlockput(src_inode);
        end_op();
        return -1;
    }

    struct inode *dp_old = nameiparent(src_file, filename);
    if (dp_old == 0)
    {
      iunlockput(src_inode);
      iunlockput(dest_inode);
      end_op();
      return -1;
    }

    uint offset;
    struct inode *ip = dirlookup(dp_old, filename, &offset);
    if (ip == 0)
    {
      iunlockput(src_inode);
      iunlockput(dest_inode);
      end_op();
      return -1;
    }

    memset(&de, 0, sizeof(de));
    ilock(dp_old);
    if (writei(dp_old, (char*)&de, offset, sizeof(de)) != sizeof(de))
    {
      iunlockput(dp_old);
      iunlockput(src_inode);
      iunlockput(dest_inode);
      end_op();
      return -1;
    }

    iunlockput(dp_old);
    iunlockput(dest_inode);
    iunlockput(src_inode);
    end_op();

    return 0; // Successfully moved the file
}
```

این کد تابعی به نام sys_move_file را پیاده‌سازی می‌کند که در سطح سیستم‌عامل به عنوان یک فراخوانی سیستمی برای جابجایی فایل‌ها از یک مسیر مبدا به یک مسیر مقصد استفاده می‌شود.

توضیحاتی راجع به کد مربوط به move_file:

Struct inode *src_inode, *dest_inode :

اشاره‌گرهایی به ساختار inode هستند که به ترتیب فایل مبدا و دایرکتوری مقصد را نشان می‌دهند.

char * sre_file, *dest_dir :

اشاره‌گرهایی هستند برای مسیر فایل مبدا و مسیر دایرکتوری مقصد.

argstr(1, &dest_dir) argstr(0, &src_file) :

سپس توسط این دو تابع ارگومان های مبدا و مقصد را از کاربر می گیریم

namei(src_file) :

تابعی که inode مربوط به مسیر فایل مبدا را جستجو و بازمی‌گرداند در واقع وجود فایل در مبدا را بررسی می کند

ilock(src_inode) :

قفل inode فایل مبدا را برای جلوگیری از دسترسی همزمان می‌بندد.

سپس دو مراحل ilock(dest_inode) و namei(dest_dir) را نیز برای مقصد نیز انجام می دهیم.

dirlink(dest_inode, filename, src_inode->inum) :

سپس فایل را با استفاده از inode فایل مبدا به دایرکتوری مقصد لینک می‌کنیم

inode nameiparent(src_file, filename) :

دایرکتوری parent فایل مبدا را جستجو و برمی‌گرداند

dirlookup(dp_old, filename, &offset) :

فایل مبدا را در دایرکتوری پدرش پیدا می‌کنیم و موقعیت آن را در متغیر offsetذخیره می‌کند.

: memset(&de, 0, sizeof(de))

محتویات de را پاک می‌کند تا فایل مبدا را از دایرکتوری parent حذف کند.

: writei(dp_old, (char*)&de, offset, sizeof(de))

ورودی خالی را به موقعیت offset در دایرکتوری parent می‌نویسیم و بدین ترتیب فایل مبدا حذف می‌شود.

: iunlockput(dp_old), iunlockput(dest_inode), iunlockput(src_inode)

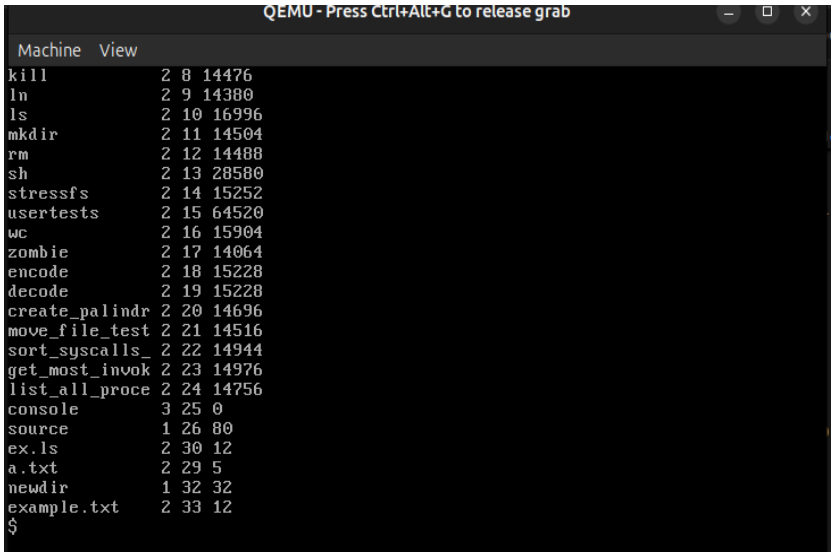
قفل فایل ها را آزاد می کنیم و اشاره‌گرها را رها می کنیم.

در نهایت فایل زیر را به جهت تست کردن سیستم کال ایجاد شده مینویسیم که دو ارگومان به عنوان اسم فایل مبدا و ادرس مقصد را می گیرد :

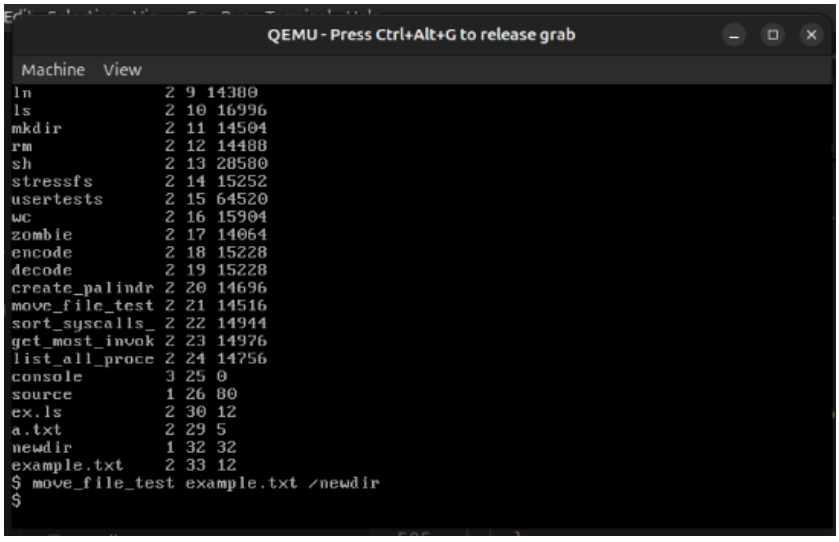
C move_file_test.c > main(int, char * [])

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

int main(int argc, char *argv[])
{
    if(argc != 3)
    {
        printf(2, "ERROR: Too many arguments!\n");
        exit();
    }

    int result = move_file(argv[1], argv[2]);

    if(result == -1)
    {
        printf(2, "ERROR: File already exists!\n");
        exit();
    }

    exit();
}
```

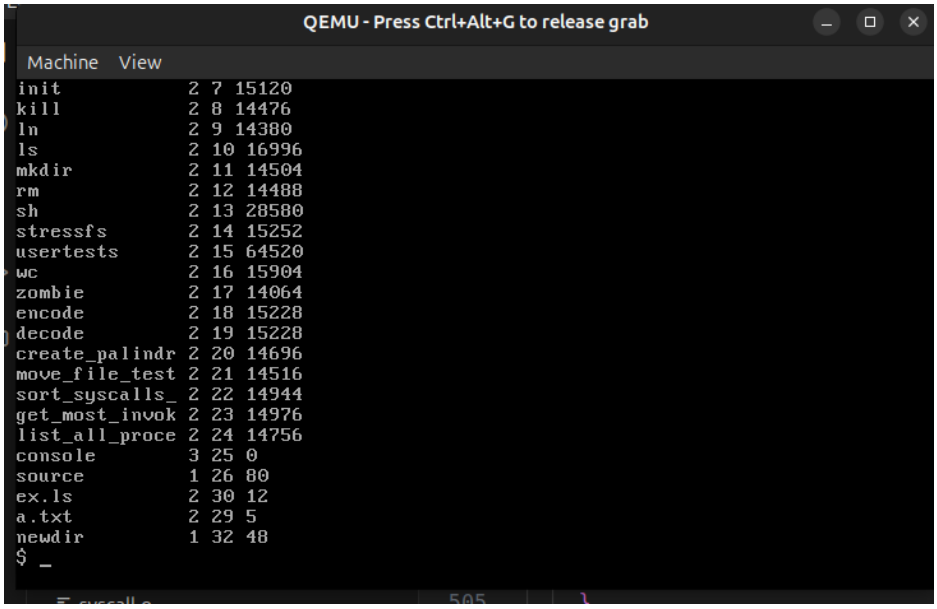
حال برای تست این سیستم کال ابتدا یک دیرکتوری جدید newdir را با mkdir و سپس فایل example.txt با محتوای HelloWorld! را با echo میسازیم:



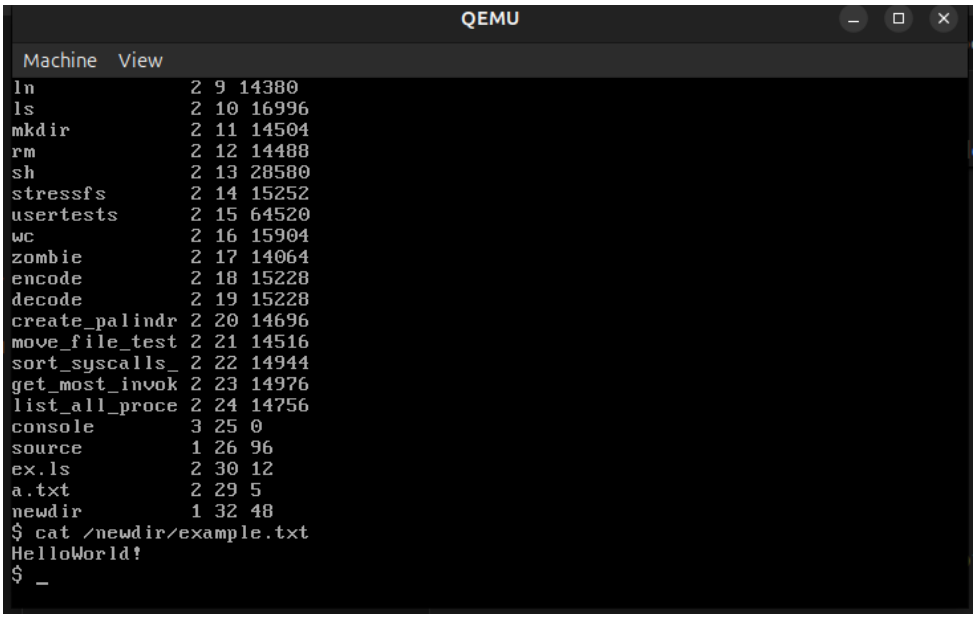
سپس با تابع move_file فایل مورد نظر را به مقصد ایجاد کرده منتقل میکنیم:



مشاهده میشود که فایل موردنظر جا به جا شده و دیگر در محل اولش نیست و از آنجا خارج شده است:



مشاهده میشود که فایل به دیرکتوری موردنظر منتقل شده و محتویات آن هم تغییر نکرده و عیناً منتقل شده است. برای دیدن محتوای آن مجددا از دستور cat استفاده میکنیم:



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

ابتدا در کد زیر در فایل proc.h به struct proc یک attribute جدید که یک ارایه ای از int ها است را با نام used_syscalls می سازیم. سایز این ارایه به تعداد سیستم کال های سیستم عامل یعنی 26 است که در خط 37، define شده است. هر process در این ارایه تعداد هر سیستم کال را نگه می دارد. مثلا در used_syscalls[0] تعداد استفاده از سیستم کال شماره 1 در اینproccess است.

```
36
37  #define MAX_SYSCALLS 26  // Define a reasonable maximum for tracked system calls.
38
39  // Per-process state
40  struct proc {
41    uint sz;                     // Size of process memory (bytes)
42    pde_t* pgdir;                // Page table
43    char *kstack;                // Bottom of kernel stack for this process
44    enum procstate state;        // Process state
45    int pid;                     // Process ID
46    struct proc *parent;         // Parent process
47    struct trapframe *tf;        // Trap frame for current syscall
48    struct context *context;     // swtch() here to run process
49    void *chan;                  // If non-zero, sleeping on chan
50    int killed;                  // If non-zero, have been killed
51    struct file *ofile[NOFILE];  // Open files
52    struct inode *cwd;           // Current directory
53    char name[16];               // Process name (debugging)
54    int used_syscalls[MAX_SYSCALLS];
55  };
56
```

حال در فایل syscalls.c به ازای هر سیستم کال که در process رخ می دهد، در ارایه ی used_syscalls ان process، به تعداد سیستم کال رخ داده شده یکی اضافه می کنیم.

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

  num = curproc->tf->eax;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    curproc->used_syscalls[num - 1]++;

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

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

همچنین در ابتدا در فایل proc.c ، به ازای ساخته شدن هر process جدید باید تمام ایندکس های ارایه ی used_syscalls را صفر بگزاریم. که یعنی در ابتدا تمام 26 سیستم کال تا کنون صدا زده نشده اند. که در خط 93 کد زیر قابل مشاهده است.

```
C proc.c > allocproc(void)
68  //PAGEBREAK: 32
73  static struct proc*
74  allocproc(void)
75  {
76    struct proc *p;
77    char *sp;
78
79    acquire(&ptable.lock);
80
81    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
82      if(p->state == UNUSED)
83        goto found;
84
85    release(&ptable.lock);
86    return 0;
87
88  found:
89    p->state = EMBRYO;
90    p->pid = nextpid++;
91
92    for(int i = 0; i < MAX_SYSCALLS; i++){
93      p->used_syscalls[i] = 0;
94    }
95
96    release(&ptable.lock);
97
98    // Allocate kernel stack.
99    if((p->kstack = kalloc()) == 0){
100     p->state = UNUSED;
101     return 0;
102   }
103   sp = p->kstack + KSTACKSIZE;
104
105   // Leave room for trap frame.
106   sp -= sizeof *p->tf;
107   p->tf = (struct trapframe*)sp;
108
109   // Set up new context to start executing at forkret,
```

حال در کد زیر در فایل sysproc.c هنگامی که این سیستم کال sort_syscalls رخ می دهد این تابع فراخوانی می شود که سپس این تابع id پروسس مربوطه را که این سیستم کال را فراخوانی کرده است از روی استک بر میدارد و تابع sort_syscalls را که در فایل proc.c است را فراخوانی می کند .

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

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

  return sort_syscalls(pid);
}
```

تابع زیر در فایل proc.c است که یک id می گیرد و process مربوطه را return می کند.

```
struct proc*
get_proc_by_pid(int pid)
{
    struct proc *p;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            release(&ptable.lock);
            return p;
        }
    }
    release(&ptable.lock);

    return 0;  // Return 0 if the process with the given PID was not found
}
```

حال در تابع زیر در فایل proc.c به ازای process پیدا شده روی ارایه ی used_syscalls اش پیمایش می کنیم و تعداد استفاده شده از هر سیستم کال برای ان process را به همراه شماره ی ان سیستم کال چاپ می کنیم.

```
int
sort_syscalls(int pid)
{
  struct proc *p = get_proc_by_pid(pid);

  if(p == 0)
    return -1;

  for(int i = 0; i < MAX_SYSCALLS; i++){
    cprintf("\tSystem call number %d: %d usage\n", i + 1, p->used_syscalls[i]);
  }

  return 0;
}
```

حال یک فایل c می زنیم که در ان این سیستم کال را تست می کنیم . به این صورت که تعدادی سیستم کال را صدا می زنیم (اعم از سیستم کال create_palindrome که قبلا به سیستم عامل خود اضافه کردیم) و سپس سیستم کال sort_syscalls را صدا می زنیم تا تعداد فراخوانی هر سیستم کال را در این برنامه به ما بگوید.

```
C sort_syscalls_test.c > main(int, char * [])
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4
5   int main(int argc, char *argv[])
6   {
7       if(argc != 1)
8       {
9           printf(2, "ERROR: Too many arguments!\n");
10          exit();
11      }
12
13      int pid = getpid();
14
15      if(fork() == 0)
16      {
17          printf(1, "Child process\n");
18          exit();
19      }
20      else
21      {
22          printf(1, "Parent process\n");
23          wait();
24      }
25
26      sleep(1);
27
28      int palindrome = create_palindrome(123);
29
30      int result = sort_syscalls(pid);
31
32      if(result == -1)
33      {
34          printf(2, "ERROR: Sort system calls failed!\n");
35          exit();
36      }
37
38      exit();
39  }
```

در عکس زیر تعداد هر سیستم کال از بین 26 سیستم کال داخل سیستم عامل که در این process فراخوانی شده اند قابل مشاهده است.

```
Group members:
 Kasra Kashani
 Albroz Mahmoudian
 Narges Babalar
$ sort_syscalls_test
Parent process
Child process
Palindrome of 12216 is 1221661221
        System call number 1: 1 usage
        System call number 2: 0 usage
        System call number 3: 1 usage
        System call number 4: 0 usage
        System call number 5: 0 usage
        System call number 6: 0 usage
        System call number 7: 1 usage
        System call number 8: 0 usage
        System call number 9: 0 usage
        System call number 10: 0 usage
        System call number 11: 1 usage
        System call number 12: 1 usage
        System call number 13: 1 usage
        System call number 14: 0 usage
        System call number 15: 0 usage
        System call number 16: 15 usage
        System call number 17: 0 usage
        System call number 18: 0 usage
        System call number 19: 0 usage
        System call number 20: 0 usage
        System call number 21: 0 usage
        System call number 22: 1 usage
        System call number 23: 0 usage
        System call number 24: 1 usage
        System call number 25: 0 usage
        System call number 26: 0 usage
$
```

مثلا با توجه به عکس زیر قابل مشاهده است که سیستم کال شماره 22 که همان create_palindrome است یکبار و خود سیستم کال sort_syscalls نیز یکبار صدا زده شده اند. یا مثلا سیستم کال شماره 16 که همان write است 15 بار صدا زده شده است.

```
C syscall.h > SYS_list_all_processes
1   // System call numbers
2   #define SYS_fork    1
3   #define SYS_exit    2
4   #define SYS_wait    3
5   #define SYS_pipe    4
6   #define SYS_read    5
7   #define SYS_kill    6
8   #define SYS_exec    7
9   #define SYS_fstat   8
10  #define SYS_chdir   9
11  #define SYS_dup     10
12  #define SYS_getpid  11
13  #define SYS_sbrk    12
14  #define SYS_sleep   13
15  #define SYS_uptime  14
16  #define SYS_open    15
17  #define SYS_write   16
18  #define SYS_mknod   17
19  #define SYS_unlink  18
20  #define SYS_link    19
21  #define SYS_mkdir   20
22  #define SYS_close   21
23  #define SYS_create_palindrome 22
24  #define SYS_move_file 23
25  #define SYS_sort_syscalls 24
26  #define SYS_get_most_invoked_syscall 25
27  #define SYS_list_all_processes 26
```

3- پیاده سازی فراخوانی سیستمی برگرداندن بیشترین فراخوانی سیستم برای یک فرآیند خاص

در کد زیر در فایل sysproc.c هنگامی که این سیستم کال get_most_invoked_syscalls رخ می دهد این تابع فراخوانی می شود که سپس این تابع id پروسس مربوطه را که این سیستم کال را فراخوانی کرده است از روی استک بر میدارد و تابع get_most_invoked_syscalls را که در فایل proc.c است را فراخوانی می کند .

```
int
sys_get_most_invoked_syscall(void)
{
  int pid;
  if (argint(0, &pid) < 0)
    return -1;

  return get_most_invoked_syscall(pid);
}
```

حال در تابع زیر در فایل proc.c به ازای process پیدا شده روی ارایه ی used_syscalls اش پیمایش می کنیم و بین تمام سیستم کال های استفاده شده در این سیستم ان سیستم کالی که بیشتر از همه رخ داده است را چاپ می کنیم.

```
int
get_most_invoked_syscall(int pid)
{
  int max_count = 0, index_max = 0;
  struct proc *p = get_proc_by_pid(pid);

  if(p == 0)
    return -1;

  for(int i = 0; i < MAX_SYSCALLS; i++){
    if(p->used_syscalls[i] > max_count){
      max_count = p->used_syscalls[i];
      index_max = i;
    }
  }
```

حال یک فایل c می زنیم که در ان این سیستم کال را تست می کنیم . به این صورت که تعدادی سیستم کال را صدا می زنیم (اعم از سیستم کال create_palindrome که قبلا به سیستم عامل خود اضافه کردیم) و سپس سیستم کال get_most_invoked_syscalls را صدا می زنیم تا سیستم کالی که بیشترین فراخوانی را داشته است را به ما بگوید.

get_most_invoked_syscall_test.c > main(int, char * [])

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

int main(int argc, char *argv[])
{
    if(argc != 1)
    {
        printf(2, "ERROR: Too many arguments!\n");
        exit();
    }

    int pid = getpid();

    if(fork() == 0)
    {
        printf(1, "Child process\n");
        exit();
    }
    else
    {
        printf(1, "Parent process\n");
        wait();
    }

    sleep(1);

    int palindrome = create_palindrome(123);

    int result = get_most_invoked_syscall(pid);

    if(result == -1)
    {
        printf(2, "ERROR: Sort system calls failed!\n");
        exit();
    }

    exit();
}
```

در عکس زیر قابل مشاهده است که بنا به انتظار سیستم کال 16 که همان write بود با 15 تا استفاده بیشترین سیستم کال مورد استفاده در این process است.

```
$ get_most_invoked_syscall_test
Parent process
Child process
Palindrome of 12200 is 1220000221
The most invoked system call is system call number 16 with 15 usage
$
```

4- پیاده سازی فراخوانی سیستمی لیست کردن پردازه ها

در کد زیر در فایل sysproc.c هنگامی که سیستم کال list_all_processes رخ می دهد این تابع فراخوانی می شود که سپس این تابع، تابع list_all_processes را که در فایل proc.c است را فراخوانی می کند .

```
159  int
160  sys_list_all_processes(void)
161  {
162    return list_all_processes();
163  }
```

تابع زیر را در فایل proc.c پیاده سازی کرده ایم که به ازای هر pid ، process مربوطه را پیدا می کند و روی ارایه ی used_syscalls اش پیمایش می کند و مجموع تمام سیستم کال های مورد استفاده در این process به صورت یک int بر می گرداند.

```
int
get_total_syscalls(int pid)
{
  struct proc *p = get_proc_by_pid(pid);
  int num_of_syscalls = 0;

  if(p == 0)
    return -1;

  for(int i = 0; i < MAX_SYSCALLS; i++){
    num_of_syscalls += p->used_syscalls[i];
  }

  return num_of_syscalls;
}
```

حال در تابع زیر در فایل proc.c روی تمام process ها پیمایش می کنیم و به ازای process هایی که در حال اجرا هستند (یعنی process هایی که state انها UNUSED نباشد) ، مجموع تمام سیستم کال های رخ داده شده در این process به همراه نام و id این process را چاپ می کنیم.

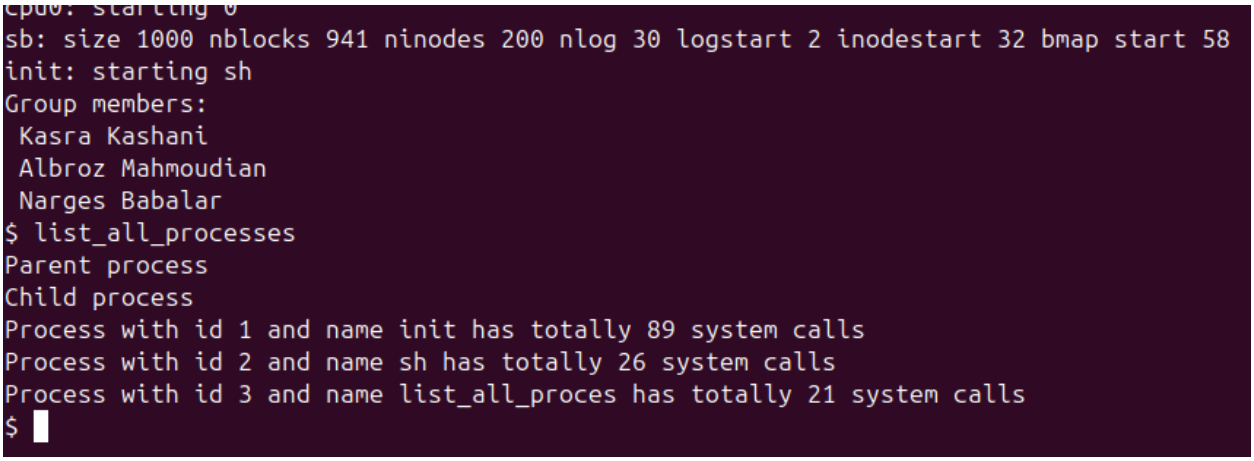
```
643
644  int
645  list_all_processes(void)
646  {
647    int flag = 0;
648
649    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
650      if (p->state != UNUSED) {  // Check if the process is active
651        cprintf("Process with id %d and name %s has totally %d system calls\n", p->pid, p->name, get_total_syscalls(p->pid));
652        flag = 1;
653      }
654    }
655
656    if(!flag)
657      return -1;
658
659    return 0;
660  }
```

حال یک فایل c می زنیم که در ان این سیستم کال را تست می کنیم :

list_all_processes_test.c > main(int, char * [])

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

int main(int argc, char *argv[])
{
    if(argc != 1)
    {
        printf(2, "ERROR: Too many arguments!\n");
        exit();
    }

    if(fork() == 0)
    {
        printf(1, "Child process\n");
        exit();
    }
    else
    {
        printf(1, "Parent process\n");
        wait();
    }

    sleep(1);

    int result = list_all_processes();

    if(result == -1)
    {
        printf(2, "ERROR: Sort system calls failed!\n");
        exit();
    }

    exit();
}
```

که خروجی در صورت فراخوانی این سیستم کال به صورت زیر خواهد بود:



به صورتی که پراسس init مربوط به کار های اولیه و بالا آمدن و بوت شدن سیستم و ترمینال، پراسس sh مربوط به خود shell و عملیات ها و I/O های داخل ترمینال، و پراسس list_all_processes نیز مربوط به همین برنامه ی سطح کاربر در حال اجرا می باشد.