آزمایشگاه سیستم عامل

پروژه دو

اعضای گروه:

پارسا علیزاده - 810101572

نیلوفر مرتضوی - 220701096

محمدرضا خالصی - 810101580

Repository: https://github.com/pars1383/OS-xv6-public

Latest Commit: 8fb6faabb65df74e01bed01a8ad66354969f8dea

1. **کتابخانه های سطح کاربر، موجود در دایرکتوری ULIB(مانند فالهای S.usys و c.ulib)، توابع وشانندهای را ارائه می دهند که ان فراخوانی های سیستی را به صورت انتزاعی مدریت می کنند. توضح دهید که این کابخانه ها چگونه با استفاده از ماکروها و توابع وشاننده، جزئیات فراخوانیهای سیستی)مانند شماره فراخوانی، آرگومانها و بازگردانی مقادر(را از برنامهنویس پنهان یکنند. همچنن، دالل استفاده از ان انزاع در بهبود قابلیت حمل، افزایش امنیت و ساده سازی توسعه برنامههای کاربر را بیان نماید.**

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 **xv6** معمولاً توابعی را شامل می‌شوند که از فراخوانی‌های سیستمی خاصی بهره می‌برند.

برای مثال:

- **فراخوانی‌های مدیریت فایل:** توابعی مانند close ،()write ،()read ،()open که دسترسی به فایل‌ها را فراهم می‌کنند.
- **فراخوانی‌های مدیریت پردازش:** توابعی مانند wait ،()exec ،()fork 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 محافظت:** 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. بنابراین، 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 مانند CPU، 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 دارد که می‌تواند زمان‌بر باشد و باعث تأخیر شود. این تأخیر بسته به‌خصوص در سیستم‌هایی با تعداد بالای فراخوانی‌های سیستمی، می‌تواند منجر به کاهش عملکرد برنامه‌ها شود. با این‌حال، تکنیک‌هایی مانند کش در CPU ممکن است این تأخیر را بهبود دهند و با کاهش تعداد فراخوانی‌های سیستمی غیرضروری بهره‌وری و عملکرد برنامه را بهبود بخشند.

- **قابلیت حمل:** استفاده از فراخوانی‌های سیستمی در برنامه‌های کاربر باعث می‌شود که برنامه‌ها به APIهای سیستم‌عامل وابسته شوند. این امر می‌تواند موجب محدودیت‌هایی در **قابلیت حمل (portability)** برنامه‌ها شود، زیرا فراخوانی‌های سیستمی ممکن است در سیستم‌عامل‌های مختلف متفاوت باشند و برای اجرا در سیستم‌عامل 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 **xv6** از فراخوانی‌های سیستمی به‌عنوان پل ارتباطی میان برنامه‌های کاربر و کرنل استفاده می‌کنند. این فراخوانی‌ها امنیت، محافظت، و مدیریت بهینه منابع را فراهم کرده و دسترسی برنامه‌های کاربر به منابع سیستم را استاندارد می‌سازند. اما بکارگیری آن‌ها می‌تواند تأثیراتی بر عملکرد و قابلیت حمل برنامه‌ها داشته باشد که با استفاده از تکنیک‌های بهینه‌سازی و استانداردسازی تا حدودی قابل حل است.

2. **فراخوانیهای سیستمی تنها روش برای تعامل برنامه های کاربر با کرنل نیستند. چه روشهای دیگری در لینوکس وجود دارند که برنامه های سطح کاربر می توانند از طرق آنها به کرنل دسترسی داشته باشند؟ هر یک از این روشها را به اختصار توضح دهید.**

فایل‌های دستگاهی (/dev/ (DEVICE FILES/

در یک سیستم‌عامل مبتنی بر یونیکس، فایل‌های دستگاهی در دایرکتوری /dev/ قرار دارند و برای دسترسی به سخت‌افزارها و منابع سیستمی استفاده می‌شوند. این فایل‌ها به دو نوع اصلی تقسیم می‌شوند: فایل‌های بلوکی (block devices) مانند دیسک‌ها (/dev/sda) و فایل‌های کاراکتری (character devices) مانند ترمینال‌ها (/dev/tty). برای دسترسی به این فایل‌ها می‌توان از دستوراتی مانند (open)، (read)، (write) استفاده کرد.

سیستم Netlink

Netlink یک سوکت برای ارتباط بین فضای کاربر (user space) و هسته (kernel) در سیستم‌عامل لینوکس است. این سیستم برای انتقال پیام‌ها بین فرآیندهای کاربر و هسته یا بین خود فرآیندهای کاربر استفاده می‌شود. Netlink جایگزین مکانیزم‌های قدیمی‌تر مانند ioctl است و از پروتکل‌های مختلفی مانند NETLINK_FIREWALL، NETLINK_ROUTE و غیره پشتیبانی می‌کند.

(ioctl (Input/Output Control

ioctl یک فراخوان سیستمی است که برای کنترل دستگاه‌ها یا انجام عملیات خاص روی فایل‌های دستگاهی استفاده می‌شود. این فراخوان به برنامه‌نویسان اجازه می‌دهد تا دستورات خاصی را به درایورهای دستگاه ارسال کنند. با این حال، استفاده بیش از حد از ioctl ممکن است کد را پیچیده‌تر کند و توصیه می‌شود در صورت امکان از مکانیزم‌های جایگزین مانند sysfs استفاده شود.

Inotify Epoll

- **epoll: یک کتابخانه جهت کنترل تعداد زیادی فایل ((file descriptors**

I/O) .۴۱) ورودی/خروجی در لینوکس برای مدیریت تعداد زیادی فایل در یک زمان استفاده می‌شود. این کتابخانه به شما این امکان را می‌دهد که چندین فایل را به صورت همزمان مانیتور کنید و در صورت

تغییر در وضعیت هر یک از آن‌ها (مثلاً خواندن یا نوشتن) به شما اطلاع دهد. این ابزار در برنامه‌هایی که نیاز به مدیریت تعداد زیادی فایل دارند، بسیار مفید است.

- **inotify: یک مکانیزم برای مانیتور کردن تغییرات در فایل‌ها و دایرکتوری‌ها**

با استفاده از inotify می‌توانید تغییرات در فایل‌ها و دایرکتوری‌ها را در زمان واقعی مانیتور کنید. این ابزار به شما این امکان را می‌دهد که به محض تغییر در یک فایل یا دایرکتوری، از آن مطلع شوید. این قابلیت برای برنامه‌هایی که نیاز به نظارت بر تغییرات فایل‌ها دارند، بسیار کاربردی است.

کاتکشن‌ها با Shared Memory و IPC (ارتباط بین پروسس‌ها)

IPC (Inter-Process Communication):

ارتباط بین پروسس‌ها با استفاده از مکانیزم‌های مختلف مانند (Shared Memory) حافظه مشترک، (Message Queues) صف‌های پیام، (Named Pipes) پایپ‌های نام‌گذاری شده و ... انجام می‌شود. این روش‌ها به پروسس‌ها این امکان را می‌دهند که با یکدیگر ارتباط برقرار کنند و اطلاعات را به اشتراک بگذارند. این ابزارها برای هماهنگی بین پروسس‌ها و انتقال داده‌ها بین آن‌ها بسیار مهم هستند.

- **حالت‌های مختلف که می‌توان از آن‌ها استفاده کرد:**

مانیتور کردن یک فایل یا دایرکتوری با استفاده از روش‌های مختلف که در بالا ذکر شد، می‌تواند به شما کمک کند تا تغییرات را به صورت همزمان مشاهده کنید. این روش‌ها به شما این امکان را می‌دهند که به صورت کارآمد و سریع از تغییرات مطلع شوید.

فایل‌های سیستمی مرتبط با /sys و /proc

۱. اطلاعات سیستمی که در /sys و /proc ذخیره می‌شوند، می‌توانند به شما کمک کنند تا اطلاعات بیشتری در مورد پروسس‌ها، دایرکتوری‌ها و فایل‌های سیستمی به دست آورید. این اطلاعات می‌توانند شامل وضعیت پروسس‌ها، منابع استفاده شده و اطلاعات دیگر باشند. این داده‌ها برای مدیریت سیستم و عیب‌یابی بسیار مفید هستند.

- **/proc:**

اطلاعات در مورد پروسس‌های در حال اجرا در /proc ذخیره می‌شوند. این دایرکتوری شامل فایل‌هایی است که اطلاعات مفیدی در مورد پروسس‌ها ارائه می‌دهند.

مثال: اطلاعات در مورد CPU در فایل /proc/cpuinfo ذخیره می‌شود. این فایل شامل جزئیاتی در مورد پردازنده سیستم است.

- **/sys: فایل‌های سیستمی /sys، تنظیمات و اطلاعات در مورد سخت‌افزار و درایورها را در خود دارند.**

این اطلاعات می‌توانند شامل تنظیمات سخت‌افزاری، وضعیت درایورها و اطلاعات دیگر باشند که برای مدیریت سیستم و عیب‌یابی بسیار مفید هستند. این داده‌ها به شما کمک می‌کنند تا درک بهتری از وضعیت سیستم و سخت‌افزارهای آن داشته باشید.

مثال: اطلاعات سیستمی در دایرکتوری /sys/class ذخیره می‌شوند. این دایرکتوری شامل اطلاعاتی در مورد کلاس‌های مختلف سخت‌افزاری است.

3. **با توجه به توضیحات ارائه‌شده درباره‌ی Gate Descriptorها در xv6 و نحوه‌ی تنظیم سطح دسترسی برای فراخوانی‌های سیستمی، چرا تنها فراخوانی سیستمی (که با Trap Gate پیاده‌سازی شده) با سطح دسترسی DPL_USER فعال می‌شود و سایر تله‌ها (مانند وقفه‌های سخت‌افزاری و استثناها) نمی‌توانند از این سطح دسترسی بهره ببرند؟**

در سیستم‌عامل xv6 و به‌طور کلی در معماری x86، Gate Descriptorها مانند Interrupt Gate، Trap Gate و Call Gate برای انتقال کنترل از سطح پایین‌تر (مثل user mode) به سطح بالاتر (مثل kernel mode) استفاده می‌شوند.

در این میان، سطح دسترسی (Descriptor Privilege Level یا DPL) در هر Gate مشخص می‌کند که کدام سطح از کد (Ring) می‌تواند از آن Gate استفاده کند.

1. فراخوانی‌های سیستمی (system calls) باید از حالت کاربر (Ring 3) در دسترس باشند، چون این تنها راه امن برای درخواست خدمات از کرنل است. بنابراین، Trap Gate مربوط به system call دارای DPL = 3 است تا برنامه‌های user بتوانند آن را فراخوانی کنند.

2. سایر تله‌ها و وقفه‌ها مثل وقفه‌های سخت‌افزاری یا page fault و ... نباید از سوی user مستقیماً فراخوانی شوند، چون این کار می‌تواند امنیت سیستم را به خطر بیندازد. این تله‌ها باید فقط از سوی سخت‌افزار یا کرنل (با سطح دسترسی بالا، مثلاً Ring 0) فعال شوند، به همین دلیل DPL آن‌ها برابر 0 تنظیم می‌شود.

3. بنابراین، اگرچه تمام این تله‌ها توسط Gate Descriptorها تعریف می‌شوند، ولی تنها Trap Gate مربوط به system call اجازه دسترسی از Ring 3 را دارد (چون DPL آن برابر 3 است)، در حالی که بقیه گیت‌ها دارای DPL=0 هستند و فقط از درون کرنل یا توسط سخت‌افزار قابل استفاده‌اند.

4. **در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غراینصورت Push نمیشود. چرا؟**

در معماری‌های پردازنده مانند x86، وقتی سطح دسترسی (Privilege Level) تغییر می‌کند (مثلاً از حالت کاربر به حالت کرنل یا بالعکس)، برای حفظ حالت قبلی و امکان بازگشت به آن، اطلاعاتی مثل Segment Selector) **SS** برای پشته) و Stack Pointer) **ESP**) روی پشته (Stack) Push می‌شوند. این کار به دلایل زیر انجام می‌شود:

1. **حفظ حالت قبلی**: تغییر سطح دسترسی معمولاً به معنای تغییر کنتکست اجرایی است (مثلاً از Ring 3 به Ring 0). برای بازگشت به حالت قبلی بدون از دست دادن اطلاعات پشته، SS و ESP ذخیره می‌شوند.

2. **مدیریت پشته‌های مختلف**: در سطوح دسترسی مختلف (مثلاً کاربر و کرنل)، ممکن است از پشته‌های متفاوتی استفاده شود. SS و ESP فعلی Push می‌شوند تا سیستم بتواند به پشته مربوط به سطح دسترسی جدید سوئیچ کند و بعداً به پشته قبلی برگردد.
3. **عدم نیاز در صورت عدم تغییر سطح دسترسی**: اگر سطح دسترسی تغییر نکند (مثلاً در یک فراخوانی درون همان Ring)، نیازی به تغییر پشته نیست و SS و ESP فعلی همچنان معتبر هستند. بنابراین، Push کردن آن‌ها ضروری نیست و برای بهینه‌سازی عملکرد، این کار انجام نمی‌شود.

این رفتار معمولاً توسط سخت‌افزار و مکانیزم‌های مدیریت وقفه یا فراخوانی سیستم (مانند IDT یا TSS) کنترل می‌شود.

5. با توجه به توضیحاتی که در مورد نحوه قرارگری پارامترهای فراخوانی سیستمی بر روی پشه و نحوه دسترسی به آنها از طرق توابعی مانند argint و argptr داده شده است، توضح دهید که این توابع چه نقشی در بازیای پارامترهای فراخوانی سیستی دارند. به خصوص، چرا تابع argptr باید بازه های آدرس ورودی را بررسی کند؟ در صورت عدم انجام ان بررسی ها، چه نوع مشکالت امنیتی (مانند دسترسی به حافظه خارج از محدوده مجاز) ممکن است ایجاد شود؟ به عنوان مثال، شرح دهید که چگونه نبود ان بررسیها در فراخوانی سیستی sys_read یتواند باعث بروز اخالل یا آسیب پذری در سیستم شود.

قش توابع argint و argptr در بازیابی پارامترهای فراخوانی سیستمی

در سیستم‌عامل‌هایی مانند xv6 یا دیگر سیستم‌های مبتنی بر معماری یونیکس، فراخوانی‌های سیستمی (System Calls) از طریق انتقال پارامترها بین فضای کاربر (User Space) و فضای کرنل (Kernel Space) انجام می‌شوند. این پارامترها معمولاً روی پشته (Stack) یا در رجیسترها قرار می‌گیرند. برای دسترسی ایمن و صحیح به این پارامترها در کرنل، توابعی مانند argint و argptr استفاده می‌شوند:

- **argint**: این تابع برای بازیابی پارامترهای عددی (Integer) از فراخوانی سیستمی استفاده می‌شود. به عنوان مثال، در فراخوانی سیستمی مانند sys_read(fd, buf, n)، پارامتر fd (شماره فایل) یا n (تعداد بایت‌ها) به‌صورت عدد صحیح هستند. argint مقدار این پارامترها را از پشته یا رجیسترها می‌خواند و به

کرنل تحویل می‌دهد. این تابع معمولاً ساده است، زیرا فقط یک مقدار عددی را کپی می‌کند و نیازی به بررسی پیچیده ندارد.

- **argptr**: این تابع برای بازیابی پارامترهایی که به صورت اشاره‌گر (Pointer) به فضای کاربر هستند، استفاده می‌شود. مثلاً در sys_read(fd, buf, n)، پارامتر buf یک اشاره‌گر به بافری در فضای کاربر است که داده‌ها باید در آن نوشته شوند. argptr اطمینان می‌دهد که اشاره‌گر معتبر است و به آدرسی در فضای کاربر اشاره می‌کند، نه فضای کرنل یا محدوده‌های غیرمجاز. این تابع معمولاً آدرس اشاره‌گر و اندازه داده‌ای که باید از آن خوانده یا نوشته شود را بررسی می‌کند.

۲. چرا argptr باید بازه‌های آدرس ورودی را بررسی کند؟

argptr وظیفه دارد اطمینان حاصل کند که اشاره‌گری که از فضای کاربر دریافت شده، به محدوده‌ای از حافظه اشاره می‌کند که:

- **در فضای کاربر است**: آدرس باید در محدوده‌ای باشد که برای فرآیند کاربر تخصیص داده شده (معمولاً آدرس‌های پایین‌تر از یک حد مشخص، مثلاً در x86 آدرس‌های زیر KERNBASE).
- **معتبر و قابل دسترسی است**: آدرس باید در محدوده حافظه تخصیص‌یافته به فرآیند باشد و نه در محدوده‌های آزاد، محافظت‌شده یا متعلق به فرآیندهای دیگر.
- **اندازه‌اش مجاز است**: اگر اشاره‌گر به یک بافر با اندازه مشخص (مثلاً n بایت) اشاره می‌کند، کل محدوده آدرس (از buf تا buf + n) باید معتبر باشد.

این بررسی‌ها به دلایل زیر ضروری هستند:

- **جلوگیری از دسترسی غیرمجاز به حافظه کرنل**: اگر یک فرآیند کاربر بتواند اشاره‌گری به فضای کرنل (مانند ساختارهای داده کرنل) ارائه دهد، ممکن است کرنل به‌طور ناخواسته داده‌های حساس را بخواند یا تغییر دهد.
- **جلوگیری از دسترسی به حافظه غیرمجاز فرآیندهای دیگر**: بدون بررسی، یک فرآیند مخرب می‌تواند به حافظه متعلق به فرآیندهای دیگر دسترسی پیدا کند.

- **جلوگیری از خطاهای حافظه**: دسترسی به آدرس‌های نامعتبر (مثلاً خارج از محدوده تخصیص‌یافته) می‌تواند باعث خطاهای صفحه (Page Fault) یا کرش کرنل شود.

۳. مشکلات امنیتی در صورت عدم بررسی بازه‌های آدرس

اگر argptr بازه‌های آدرس را بررسی نکند، مشکلات امنیتی متعددی ممکن است رخ دهد:

- **دسترسی به حافظه خارج از محدوده مجاز**: یک فرآیند مخرب می‌تواند اشاره‌گری به فضای کرنل یا حافظه دیگر فرآیندها ارائه دهد و داده‌های حساس را بخواند یا تغییر دهد.
- **آسیب‌پذیری‌های سرریز بافر (Buffer Overflow)**: بدون بررسی اندازه، کرنل ممکن است داده‌هایی را در آدرس‌های غیرمجاز بنویسد و باعث بازنویسی حافظه شود.
- **فاش شدن اطلاعات (Information Disclosure)**: خواندن داده‌ها از آدرس‌های غیرمجاز ممکن است اطلاعات حساس کرنل یا فرآیندهای دیگر را فاش کند.
- **کرش سیستم**: دسترسی به آدرس‌های نامعتبر می‌تواند باعث خطاهای سخت‌افزاری (مانند Page Fault) شود که کرنل را ناپایدار یا متوقف کند.

۴. مثال: آسیب‌پذیری در فراخوانی سیستمی sys_read

فراخوانی سیستمی sys_read(fd, buf, n) برای خواندن n بایت داده از فایل با شناسه fd و نوشتن آن‌ها در بافر buf در فضای کاربر استفاده می‌شود. فرض کنید argptr بررسی‌های لازم را برای اشاره‌گر buf و اندازه n انجام ندهد. در این صورت، سناریوهای زیر ممکن است رخ دهند:

- **سناریوی ۱: دسترسی به حافظه کرنل**
  - یک فرآیند مخرب اشاره‌گری به فضای کرنل (مثلاً آدرس یک ساختار داده حساس مانند جدول فرآیندها) ارائه می‌دهد.
  - کرنل بدون بررسی، داده‌های فایل را در این آدرس می‌نویسد.
  - **پیامد**: این کار می‌تواند ساختارهای داده کرنل را خراب کند، باعث کرش سیستم شود یا به فرآیند مخرب اجازه دهد کنترل سیستم را به دست گیرد (مثلاً با تغییر جدول فرآیندها).

- **سناریوی ۲: فاش شدن اطلاعات**
  - فرآیند مخرب اشاره‌گری به آدرسی در فضای کاربر اما متعلق به فرآیند دیگری ارائه می‌دهد.
  - کرنل داده‌های فایل را در این آدرس می‌نویسد یا داده‌های موجود در آن آدرس را می‌خواند.
  - **پیامد:** اطلاعات حساس فرآیند دیگر (مانند رمزها یا کلیدهای رمزنگاری) فاش می‌شود.
- **سناریوی ۳: سرریز حافظه**
  - فرآیند مخرب یک اشاره‌گر معتبر اما با اندازه n بسیار بزرگ ارائه می‌دهد که خارج از محدوده حافظه تخصیص‌یافته است.
  - کرنل بدون بررسی، سعی می‌کند داده‌ها را در آدرس‌های غیرمجاز بنویسد.
  - **پیامد:** این می‌تواند باعث بازنویسی حافظه، خراب شدن داده‌های دیگر یا ایجاد خطای صفحه شود که سیستم را ناپایدار می‌کند.
- **سناریوی ۴: کرش سیستم**
  - فرآیند مخرب اشاره‌گری به آدرس نامعتبر (مثلاً 0x0 یا آدرسی خارج از محدوده) ارائه می‌دهد.
  - کرنل بدون بررسی، سعی می‌کند در این آدرس بنویسد.
  - **پیامد:** این کار باعث خطای صفحه (Segmentation Fault) در کرنل می‌شود که می‌تواند کل سیستم را متوقف کند.

بررسی گام های اجرای فراخوانی سیستی در سطح کرنل توسط gdb

پس از clean make و make gemu-gdb کردن، برای اجرا با نقاط کلیدی که در آن وارد می‌شوید، gdb را می زنیم و فایل kernel را به ان می دهیم با b syscall یک breakpoint قرار میدهیم

با زدن دستور bt، تمام فراخوانی‌های تابع مورد نظر تا رسیدن به breakpoint نمایش داده می‌شود. در واقع محتویات داخل stack نمایش داده می‌شود که برای نمایش روند اجرای برنامه استفاده می‌شود. می‌دانیم که هنگام فراخوانی شدن یک تابع، یک frame stack حاوی اطلاعات آن تابع مانند return address بر روی call stack پوش می‌شود و با دستور bt محتویات آن stack نمایش داده می‌شود. در واقع پس از اجرا شدن دستور

int64، مقدارش در vector64 در فایل vectors.s اضافه می‌شود و سپس به alltraps در فایل trapsasm.s می‌رود و frame trap ساخته و آن را در آن stack پوش می‌کند.

```
(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 می‌کنیم تا به `frame stack` بالاتر در `bt` برویم و به تابع `callee` برسیم که قبل‌تر صدا زده شده است. اما تابع `syscall` در خود هیچ تابع دیگری را صدا نزده و لذا `callee` 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 frams stack پایین تر 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 دستورprint 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
```

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
```

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 ۱۰، سیستم کال ()write با شماره ی 16 و در نهایت سیستم کال ()fork با شماره ی 1.

در xv6، تیک‌ها معمولاً هر 10 میلی‌ثانیه (100 تیک در ثانیه) رخ می‌دهند، اما این مقدار به تنظیمات تایمر سخت‌افزاری و بار سیستم بستگی دارد. اگر تیک‌ها دقیقاً با ساعت واقعی همگام نباشند، زمان انتظار واقعی ممکن است کمی بیشتر یا کمتر از مقدار مورد انتظار باشد.

1. **زمان‌بندی RTC**:

فراخوانی cmostime زمان ساعت واقعی (RTC) را می‌خواند که ممکن است با سرعت تیک‌های سیستم کاملاً همگام نباشد. RTC معمولاً دقت ثانیه‌ای دارد، در حالی که تیک‌ها در مقیاس میلی‌ثانیه هستند.

2. **تأخیرهای سیستمی**:

هنگام اجرای yield()، فرآیند ممکن است به دلیل زمان‌بندی سایر فرآیندها یا وقفه‌ها کمی بیشتر منتظر بماند. این باعث می‌شود زمان واقعی سپری‌شده بیشتر از مقدار تئوری (100 / ticks ثانیه) باشد.

3. **رزولوشن پایین RTC**:

چون RTC زمان را در قالب ثانیه برمی‌گرداند، اگر انتظار ما کمتر از یک ثانیه باشد (مثلاً 50 تیک ≈ 0.5 ثانیه)، ممکن است تفاوت زمانی گزارش‌شده صفر یا یک ثانیه باشد که دقت پایینی دارد.

پیاده سازی فراخوانی سیستمی logs:

```
#define SYS_make_user_syscall 23
#define SYS_login_syscall     24
#define SYS_logout_syscall    25
#define SYS_get_logs_syscall  26
```

در کد بالا شماره سیستم کال ها برای logs مشاهده می شود.

```
$ make_user 1 pass1
User created with UID 1
$ make_user 2 pass2
User created with UID 2
$ login 1 pass1
Logged in as UID 1
$ login 2 pass1
Another user (UID 1) is already logged in
$ test_palindrome 131
131
$ get_logs
Logs for UID 1 (PID 8):
22
$ logout 1
UID 1 logged out
UID 1 logged out
$ get_logs
Logs for all users (not logged in):
UID 1:
22
26
```

با فراخوانی دستورات مورد نظر که در شکل بالا تست شده است میتوان از سیستم کال های مربوطه در سطح کاربر استفاده کرد.

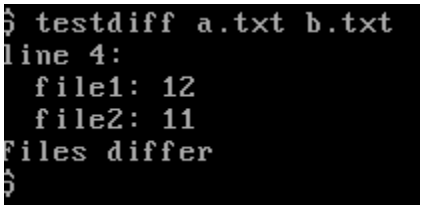
پالیندروم:

این سیستم کال با شماره 22 ثبت شده است.

کارکرد این دستور نیز در تصویر بالا قابل مشاهده است.

Diff system call:

دستور تست این بخش به شکل کامند testdiff که به شکل زیر با دو فایل تکست مورد نظر صدا زده میشود:



در صورت یکی بودن فایل ها "files are identical" و در صورت متفاوت بودن آنها مانند شکل خطوط تفاوت مشخص میشوند.