گزارش کار تمرین کامپیوتری دوم

گروه 9

سجاد تقی زاده 810102425

دنیز مصطفی زادگان 810102603

محمد حسین شفیعی 810102470

هش اخرین کامیت: 09e6d83ecb2d4f063d830378de58346f2f585908

Q1:

کتابخانه‌های: سطح کاربر، موجود در دایرکتوری ULIB (مانند فایل‌های usys.S و ulib.c)، توابع پوشاننده‌ای را ارائه می‌دهند که این فراخوانی‌های سیستمی را به صورت انتزاعی مدیریت می‌کنند. توضیح دهید که این کتابخانه‌ها چگونه با استفاده از ماکروها و توابع پوشاننده، جزئیات فراخوانی‌های سیستمی(مانند شماره فراخوانی، آرگومان‌ها و بازگردانی مقادیر) را از برنامه‌نویس پنهان می‌کنند. همچنین،دلایل استفاده از این انتزاع در بهبود قابلیت حمل، افزایش امنیت و ساده‌سازی توسعه برنامه‌های کاربر را بیان نمایید.

چگونگی عملکرد

کتابخانه‌های ULIB یک Wrapper Layer بین برنامه کاربردی و هسته سیستم‌عامل ایجاد می‌کنند. این کار عمدتاً توسط فایل usys.S انجام می‌شود.

استفاده از ماکرو SYSCALL در usys.S:

فایل usys.S شامل ماکرویی به نام SYSCALL است.این ماکرو به صورت خودکار کد اسمبلی لازم برای هر فراخوانی سیستمی را تولید می‌کند.

وقتی شما یک فراخوانی سیستمی مانند fork را در usys.S تعریف می‌کنید، این ماکرو کد اسمبلی زیر را تولید می‌کند:

movl $SYS_fork, %eax: شماره فراخوانی سیستمی را در رجیستر eax قرار می‌دهد.

int $T_SYSCALL: دستورالعمل trap را اجرا می‌کند تا کنترل را به هسته منتقل کند.

پنهان‌سازی جزئیات:

شماره فراخوانی: برنامه‌نویس C فقط تابع fork() را صدا می‌زند. او هرگز نیازی به دانستن اینکه شماره fork مثلاً ۱ است و باید در eax قرار گیرد، ندارد. ماکرو SYSCALL این کار را مدیریت می‌کند.

مدیریت آرگومان‌ها: هنگامی که برنامه‌نویس تابعی مانند write را صدا می‌زند، کامپایلر C این آرگومان‌ها را طبق قراردادهای استاندارد فراخوانی C روی user stack قرار می‌دهد. هسته انتظار دارد آرگومان‌ها را از همین پشته بخواند تابع پوشاننده در usys.S اطمینان حاصل می‌کند که این انتقال به درستی انجام شده است.

نقش ulib.c:

این فایل توابع کتابخانه‌ای استاندارد C مانند malloc، printf و strcmp را فراهم می‌کند. این توابع، خودشان ممکن است از پوشاننده‌های usys.S استفاده کنند. برای مثال، malloc در نهایت فراخوانی سیستمی sbrk را برای دریافت حافظه بیشتر از هسته صدا می‌زند.

چرایی استفاده :

بهبود قابلیت حملPortability:

این مهم‌ترین دلیل است. کد برنامه کاربردی که به زبان C نوشته شده و از fork() و write() استفاده می‌کند، باید بدون تغییر روی معماری‌های مختلف سخت‌افزاری اجرا شود.
در xv6: از دستور int 64 استفاده می‌شود و آرگومان‌ها از پشته خوانده می‌شوند. اگر انتزاع وجود نداشت، برنامه‌نویس مجبور بود برای هر معماری، کد اسمبلی متفاوتی بنویسد. با وجود لایه ULIB، کافی است برنامه C خود را در برابر نسخه متناسب با آن معماری از ULIB کامپایل کند و خود usys.S جزئیات را مدیریت خواهد کرد.

ساده‌سازی توسعه برنامه:

برنامه‌نویسان می‌توانند در یک سطح بالاتر کار کنند و نیازی به درگیر شدن با جزئیات سطح پایین اسمبلی، مدیریت دستی رجیستر eax، یا دانستن شماره وقفه‌ها ندارند.این کار توسعه را بسیار سریع‌تر و خطایابی را آسان‌تر می‌کند.

افزایش امنیت :

اگرچه امنیت اصلی توسط هسته تأمین می‌شود، اما لایه انتزاعی به دو روش به امنیت کمک می‌کند:
کاهش خطای برنامه‌نویس: با فراهم کردن یک رابط استاندارد، از خطاهای سهوی برنامه‌نویسان در نوشتن کد اسمبلی برای ورود به هسته جلوگیری می‌کند.

اجرای یکنواخت: تضمین می‌کند که همه ورودی‌ها به هسته از یک کانال مشخص و استاندارد عبور می‌کنند، که این موضوع حسابرسی و اعمال سیاست‌های امنیتی در سطح هسته را آسان‌تر می‌کند.

Q2:

در دو سیستم عامل sysexit یا sysenter و دستورات جدید مانند int مقایسه‌ای بین دستور:
مختلف(برای مثال Linux و xv6)انجام دهید. در این مقایسه به موارد زیر توجه کنید:
.1نحوه پیاده‌سازی و عملکرد این دستورات :چطور هرکدام از این دستورات به فراخوانی سیستم درسطح هسته پرداخته و از چه مکانیزم‌هایی استفاده می‌کنند؟
.2کارایی :چگونه دستور int در مقایسه با دستورات جدیدتر مانند sysenter و sysexit از نظر عملکردو سربار پردازشی در سیستم‌های عامل مختلف عمل می‌کند؟
.3کاربردهای مختلف :در چه شرایطی یکی از این دستورات نسبت به دیگری برتری دارد و چرا؟

1 :نحوه پیاده‌سازی و عملکرد

int :

- عملکرد: این دستور یک وقفه نرم‌افزاری عمومی ایجاد می‌کند.
- پیاده‌سازی:

1. وقتی دستور int $T_SYSCALL که اجرا می‌شود، پردازنده فورا اجرای برنامه کاربر را متوقف می‌کند.
2. پردازنده به جدولی در حافظه به نام (Interrupt Descriptor Table) مراجعه می‌کند.
3. ورودی ۶۴اُم را پیدا می‌کند. این گیت حاوی آدرس روتین کنترل‌کننده وقفه در هسته و سطح دسترسی مورد نیاز است.
4. قبل از پرش به کد هسته، پردازنده به صورت خودکار اطلاعات وضعیت فعلی برنامه و در صورت تغییر سطح دسترسی، SS و ESP پشته کاربر را روی پشته هسته پوش می‌کند.
5. در نهایت، پردازنده به آدرس کنترل‌کننده در هسته می‌پرد و اجرای هسته آغاز می‌شود.

:sysenter / sysexit

- عملکرد: اینها دستوراتی تخصصی و بهینه‌سازی شده فقط برای فراخوانی سیستمی هستند، نه یک وقفه عمومی.
- پیاده‌سازی:

1. هسته لینوکس هنگام بوت شدن، یک سری رجیسترهای خاص پردازنده را تنظیم می‌کند که به sysenter می‌گویند آدرس دقیق روتین هسته، آدرس پشته هسته و سگمنت کد هسته کجاست.
2. وقتی کتابخانه glibc دستور sysenter را اجرا می‌کند، پردازنده بدون مراجعه به IDT و به صورت مستقیم:
   - سطح دسترسی را به هسته تغییر می‌دهد.
   - مقدار EIP (شمارنده برنامه) را با آدرس روتین هسته جایگزین می‌کند.
   - مقدار ESP (اشاره‌گر پشته) را با آدرس پشته هسته جایگزین می‌کند.
3. تفاوت کلیدی: این دستور به صورت خودکار هیچ چیزی را روی پشته *پوش* نمی‌کند. این وظیفه glibc است که قبل از اجرا، آدرس بازگشت و پشته کاربر را در رجیسترهای دیگری ذخیره کند تا هسته بتواند بعداً از آن‌ها برای بازگشت استفاده کند.
4. دستور sysexit نیز عملیات بازگشت سریع به فضای کاربر را به صورت اتمیک انجام می‌دهد.

۲. کارایی

:int

- **کا**رایی: پایین .
- دلیل سربار:

1. مراجعه به حافظه (IDT): پردازنده باید به IDT در حافظه اصلی مراجعه کند که می‌تواند منجر به *Cache Miss* شود.
2. عملیات پشته: پردازنده به صورت خودکار چندین عملیات push روی پشته هسته انجام می‌دهد که زمان‌بر است.
3. عمومی بودن: این یک سازوکار سنگین و عمومی برای مدیریت انواع اتفاقات است و برای فراخوانی سیستمی بهینه نیست.

:sysenter / sysexit

- کارایی: بسیار بالا .
- دلیل کارایی:

1. بدون مراجعه به حافظه: تنظیمات را از رجیسترهای پرسرعت داخلی MSR می‌خواند و نیازی به IDT نیست.
2. بدون عملیات پشته: هیچ push خودکاری روی حافظه انجام نمی‌دهد؛ تمام عملیات جابجایی در سطح رجیسترهای خود CPU رخ می‌دهد.
3. تخصصی بودن: این دستورات دقیقاً برای همین کار طراحی شده‌اند و تمام مراحل غیرضروری حذف شده‌اند. این همان دلیلی است که نمودار در پروژه شما نشان می‌دهد که IPC در لحظه فراخوانی به شدت افت می‌کند؛ sysenter این افت را به حداقل می‌رساند.

۳. کاربردهای مختلف

:int

- کاربرد: سادگی و یکپارچگی.
- برتری: در یک سیستم‌عامل آموزشی مانند xv6 که سادگی در آن اولویت دارد، استفاده از یک سازوکار واحد برای مدیریت هم فراخوانی‌های سیستمی، هم خطاها و هم وقفه‌های سخت‌افزاری بسیار منطقی است.
- ضعف: برای سیستم‌عامل‌های مدرن با حجم بالای I/O و فراخوانی‌های سیستمی به دلیل سربار بالا، اصلاً مناسب نیست.

:sysenter / sysexit

- کاربرد: فراخوانی‌های سیستمی پرسرعت و پرتکرار.
- برتری: در سیستم‌عامل‌های پربازده مانند لینوکس که در هر ثانیه هزاران فراخوانی سیستمی انجام می‌دهند، کاهش این سربار تفاوت کارایی چشمگیری ایجاد می‌کند.
- نکته: لینوکس از هر دو روش استفاده می‌کند! لینوکس از sysenter برای *فراخوانی‌های سیستمی* استفاده می‌کند، اما همچنان از سازوکار int و جدول IDT برای مدیریت *خطاها* و *وقفه‌های سخت‌افزاری* بهره می‌برد.

Q3:

با توجه به توضیحات ارائه شده درباره ی Descriptor Gateها در 6xv و نحوه ی تنظیم سطح دسترسی برای فراخوانی های سیستمی، چرا تنها فراخوانی سیستمی)که با Gate Trap پیاده سازی شده(با سطح دسترسی USER_DPL فعال میشود و سایر تله ها)مانند وقفه های سخت افزاری و استثناها(نمیتوانند از این سطح دسترسی بهره ببرند؟

تنها System Call است که باید از (User Mode) قابل اجرا باشد،
زیرا برنامه‌های کاربر نیاز دارند بتوانند به‌صورت امن و کنترل‌شده خدمات کرنل را درخواست کنند.

بنابراین Gate مربوط به system call با سطح دسترسی USER_DPL (یعنی DPL=3) تنظیم می‌شود.

اما وقفه‌های سخت‌افزاری (Interrupts) و استثناها (Exceptions)
توسط سخت‌افزار یا خود CPU تولید می‌شوند، نه توسط برنامه‌ی کاربر،
و اگر کاربر بتواند آن‌ها را مستقیم فعال کند، سیستم دچار نقض امنیت و پایداری می‌شود.
به همین دلیل، DPL آن‌ها برابر با ۰ (سطح کرنل) است.

سایر تله‌ها فقط در سطح کرنل ایجاد می‌شوند و DPL=0 دارند تا کاربر نتواند مستقیماً به کرنل وارد شود.

Q4:

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

در معماری x86 , هر Privilege Level می‌تواند پشته مخصوص خود را داشته باشد. ین پشته‌ها در TSS (Task State Segment)) تعریف می‌شوند.
وقتی دستور int n اجرا می‌شود:

1. CPU بررسی می‌کند آیا سطح دسترسی مقصد (در Gate Descriptor مربوطه) با سطح فعلی (CS.DPL یا CPL) متفاوت است یا نه.

2. اگر تفاوت وجود دارد → یعنی باید از پشته جدیدی استفاده شود.

   - CPU از TSS آدرس پشته‌ی مقصد (مثلاً پشته‌ی کرنل) را می‌گیرد.

   - مقدار ss و esp فعلی (که مربوط به user mode بودند) روی پشته‌ی جدید (kernel) ذخیره می‌شوند.

3. اگر تفاوت وجود ندارد → یعنی در همان سطح هستیم.

- CPU از همان پشته استفاده می‌کند.
- پس نیازی نیست ss و esp را دوباره push کند.

مثال :

در int $64 (فراخوانی سیستمی):

- برنامه در Ring 3 -> User Mode است.
- Gate مربوطه (T_SYSCALL) در Ring 0 -> Kernel Mode اجرا می‌شود.
- بنابراین CPU سطح دسترسی را از 3 به 0 تغییر می‌دهد → پس باید:
  - پشته‌ی جدید را از TSS بگیرد.
  - SS و ESP کاربر را روی پشته‌ی کرنل ذخیره کند.
  - بعد کنترل را به کرنل بدهد.

اما اگر وقفه‌ای یا trapی در داخل خود کرنل رخ دهد (یعنی از Ring 0 به Ring 0):

چون سطح دسترسی تغییری نکرده، همان پشته‌ی فعلی استفاده می‌شود
و بنابراین SS و ESP ذخیره نمی‌شوند.

نتیجه:

در دستور int n، پردازنده تنها زمانی مقادیر SS و ESP را روی پشته ذخیره می‌کند که انتقال کنترل بین دو سطح دسترسی مختلف انجام شود (مثلاً از سطح کاربر به هسته).

زیرا در این حالت باید پشته‌ی جدیدی در سطح مقصد فعال گردد و لازم است مقادیر پشته‌ی قبلی حفظ شوند تا پس از بازگشت (iret) بازیابی شوند.

در صورتی که سطح دسترسی تغییری نکند، پشته همان است و نیازی به ذخیره‌ی این مقادیر وجود ندارد.

:Q5

با توجه به توضیحاتی که در مورد نحوه قرارگیری پارامترهای فراخوانی سیستمی بر روی پشته و نحوه دسترسی به آنها از طریق توابعی مانند argint و argptr داده شده است، توضیح دهید که این توابع چه نقشی در بازیابی پارامترهای فراخوانی سیستمی دارند. بهخصوص، چرا تابع argptr باید بازههای آدرس ورودی را بررسی کند؟ در صورت عدم انجام این بررسیها، چه نوع مشکلات امنیتی)مانند دسترسی به حافظه خارج از محدوده مجاز(ممکن است ایجاد شود؟ به عنوان مثال، شرح دهید که چگونه نبود این بررسیها در فراخوانی سیستمی sys_read میتواند باعث بروز اختلال یا آسیب پذیری در سیستم شود. همچنین عنوان کنید که آیا حذف چک کردن بازه آدرس عملی و قابل پیاده سازی است یا خیر؟ یا ایا میتوان عملی بازه اشتباه وارد کنیم؟

توابعی مثل argint و argptr مسئول بازیابی و اعتبارسنجی پارامترهای فرستادهشده توسط برنامهٔ کاربر از روی پشتهٔ کاربر هستند.

argptr علاوه بر خواندن مقدار اشارهگر، بازهٔ آدرس (start … start+len-1) را هم چک میکند تا مطمئن شود اشارهگر به حافظهای در فضای کاربر و در محدودهٔ مجاز و قابلدسترس اشاره میکند.

اگر این چکها انجام نشود، کاربر میتواند آدرسهایی بدهد که باعث خواندن/نوشتن خارج از محدوده، خواندن حافظه کرنل، کرش کرنل و یا فرار از ایزولاسیون کاربر/کرنل (privilege escalation) شود.

چرا argptr باید بازهٔ آدرس را بررسی کند؟

جداسازی فضای آدرس: حافظهٔ کاربر و حافظهٔ کرنل باید ایزوله باشند. بدون بررسی، اشارهگر کاربر ممکن است به آدرسهای کرنل اشاره کند → کرنل ممکن است دادههای حساس را بازنویسی یا افشا کند.

محافظت از دادههای سایر پردازهها: بدون چک، برنامهٔ بدخواه میتواند حافظهٔ پردازهٔ دیگر را بخواند/بنویسد.

جلوگیری از integer overflow: اگر start + len با overflow بشود، چک سادهی start + len <= USERTOP اشتباه میشود—بنابراین چک باید overflow-safe باشد.

sys_read(fd, buf, n) کرنل بدون چک مستقیم read را اجرا میکند و دادهها را به آدرس buf می‌نویسد.:

حملات/خطاهایی که ممکن است رخ دهد:

1. نوشتن در حافظهٔ کرنل
   کاربر مقدار buf را طوری قرار میدهد که به داخل فضای کرنل اشاره کند (مثلاً آدرس ثابت یا با استفاده از باگ دیگری). sys_read دادهها را مستقیماً به آنجا مینویسد → ساختارهای کرنل (مثلاً

جدول صفحات، TCB، یا کد) خراب می‌شوند → امکان اجرای کد دلخواه در کرنل یا کرش کامل سیستم.

2. دسترسی خارج از محدودهٔ کاربر
   کاربر buf را طوری قرار می‌دهد که buf+len از حدود فضای کاربر فراتر برود (مثلاً buf نزدیک پایان و n بزرگ). نوشتن به آدرس‌های خارج از فضای کاربر → بازنویسی حافظهٔ پردازهٔ دیگر یا کرنل.

3. صفحهٔ نقشه‌نشده → page fault در کرنل
   کرنل تلاش به نوشتن در صفحه‌ای می‌کند که نقشه نشده است؛ اگر این fault در کرنل به درستی هندل نشود، منجر به kernel panic می‌شود.

4. آسیب‌پذیری اطلاعاتی (Information Leak)
   اگر تابع خواندن داده‌ای از دستگاه یا فایل باشد، و آدرس کاربر به آدرس کرنل اشاره کند، ممکن است داده‌های کرنل به برنامهٔ کاربر نشت کند (خواندن کرنل). یا بالعکس، کاربر بتواند مقادیری را که نباید تغییر کند بازنویسی کند.

حذف کامل این چک‌ها عملاً غیرقابل‌پذیر و ناایمن است. نتیجهٔ حذف: کرنل آسیب‌پذیر، ناپایدار، و مستعد افشا/تخریب حافظهٔ حساس.

پیاده‌سازی امن باید :

چک محدودهٔ آدرس کاربر (start و end) با حفاظت در برابر overflow

استفاده از توابع copyin/copyout که page faults را امن مدیریت می‌کنند،

در صورت امکان، کاهش زمان بین validate و use یا قفل‌گذاری مناسب برای جلوگیری از TOCTOU باشد.

Q6 :

در صورتی که کاربر تغییراتی به صورت دستی در رجیسترهای بالا ایجاد کند، چه مشکلاتی به همراه خواهد داشت؟

1) نقض قرارداد ABI و خراب‌شدن وضعیت برنامه

در معماری‌ای که ABI مشخص می‌کند برخی ثبات‌ها باید محفوظ بمانند (مثلاً ebx, esi, edi, ebp در System V i386)، اگر کاربر این ثبات‌ها را تغییر دهد:

- wrappers یا کتابخانه‌ها که انتظار داشتند این ثبات‌ها بعد از syscall دست‌نخورده بمانند، پس از بازگشت مقدارهای نادرست می‌بینند در نتیجه متغیرها و اشاره‌گرها به اشتباه می‌روند سپس کرش یا رفتار نامناسب رخ می‌دهد..

2) تغییر شمارهٔ syscall (رجیستر eax)

کاربر می‌تواند eax را به شمارهٔ نامعتبر یا دلخواه بگذارد:

- اگر شماره نامعتبر باشد، کرنل معمولاً آن را تشخیص داده و -1 برمی‌گرداند یا پیغام خطا می‌دهد.
- اگر کرنل نقص خطوط بررسی داشته باشد، ممکن است تابع نامناسبی اجرا شود و موجب رفتار ناخواسته گردد.

3) تغییر رجیستر هایی که حاوی آدرس‌اند (ebx, ecx, edx یا آرگومان‌های روی پشته)

اگر اشاره‌گرها به آدرس‌های نامعتبر یا به آدرس‌های فضای کرنل اشاره کنند:

- در کرنل ناسالم یا چک‌نشده: نوشتن/خواندن به آدرس کرنل می‌تواند ساختارهای کرنل را خراب کند یا منجر به escalation شود.

4) تغییر ESP / تغییر پشته

اگر کاربر قبل از int مقدار esp را تغییر دهد یا stack را دستکاری کند:

- CPU هنگام push مقادیر مربوط به انتقال سطح (SS, ESP, EIP, CS, EFLAGS) و سپس alltraps انتظار ساختاری مشخص روی پشته دارد. اگر ESP به آدرسی نامناسب اشاره کند، این push ها ممکن است روی صفحات نقشه‌نشده صورت گیرد → page fault در حالتی که در کرنل رخ دهد

خطرناک است و ممکن است باعث kernel panic شود.

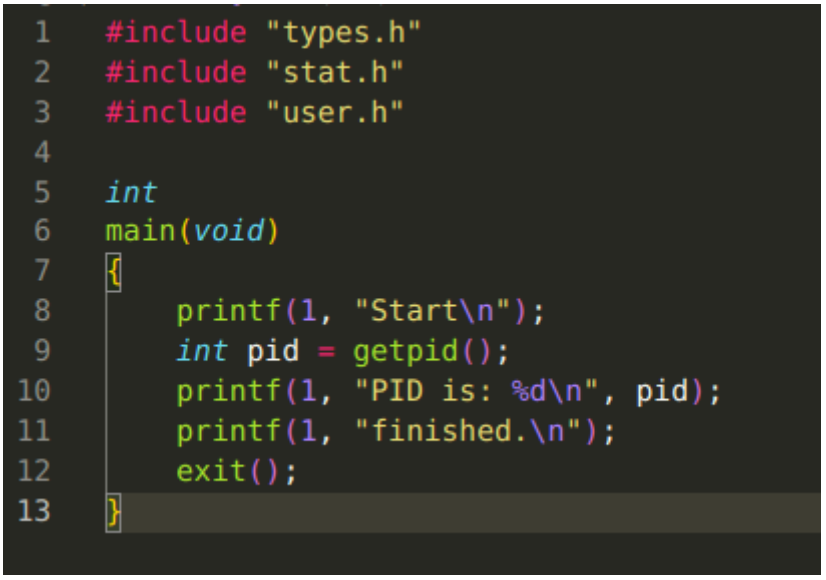
- همچنین می‌توان رفتار بازگشت (iret) را خراب کرد و اجرای برنامه را غیرقابل پیش‌بینی کرد.

5) تغییر رجیسترهای سیستمی/سگمنت‌ها

دست‌کاری CS, SS, یا ثبات‌های سگمنت می‌تواند منجر به General Protection Fault یا سایر exceptions شود. اینها معمولاً در سطح کاربر محدودند ولی تغییرات نامناسب می‌تواند فورساً فرآیند را بکشد یا سیستم را ناپایدار کند.

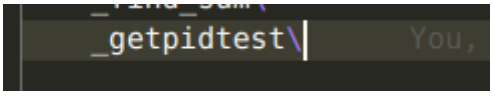
بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

کد برنامه سمت کاربر:



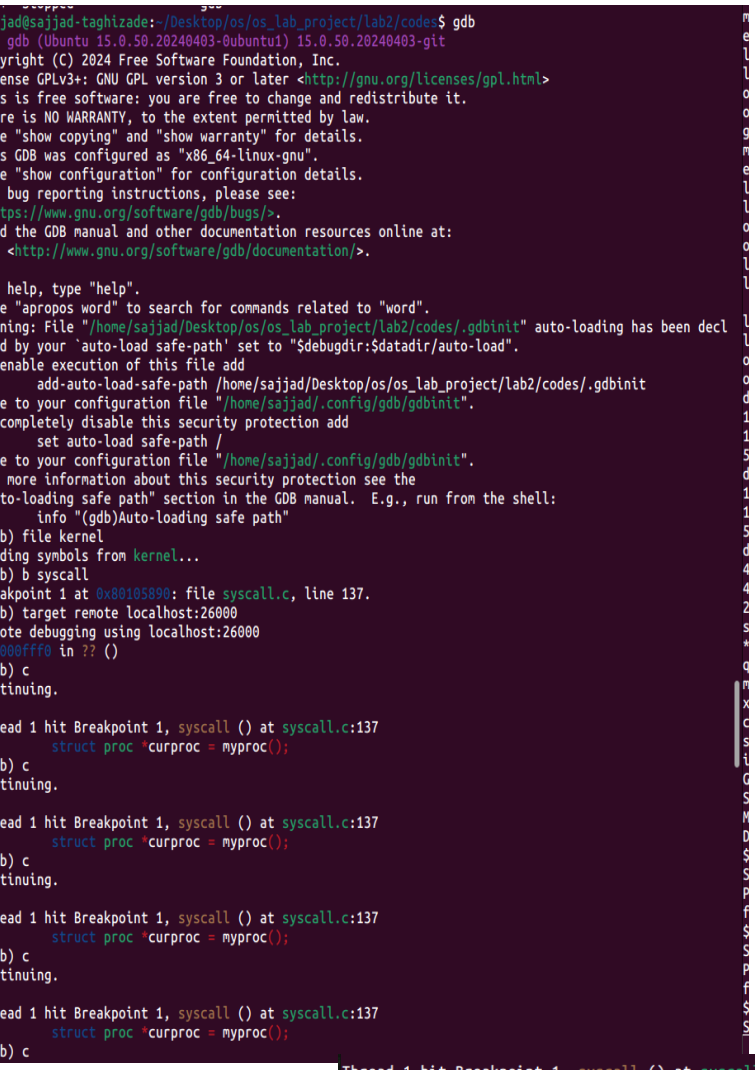
```c
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(void)
{
    printf(1, "Start\n");
    int pid = getpid();
    printf(1, "PID is: %d\n", pid);
    printf(1, "finished.\n");
    exit();
}
```

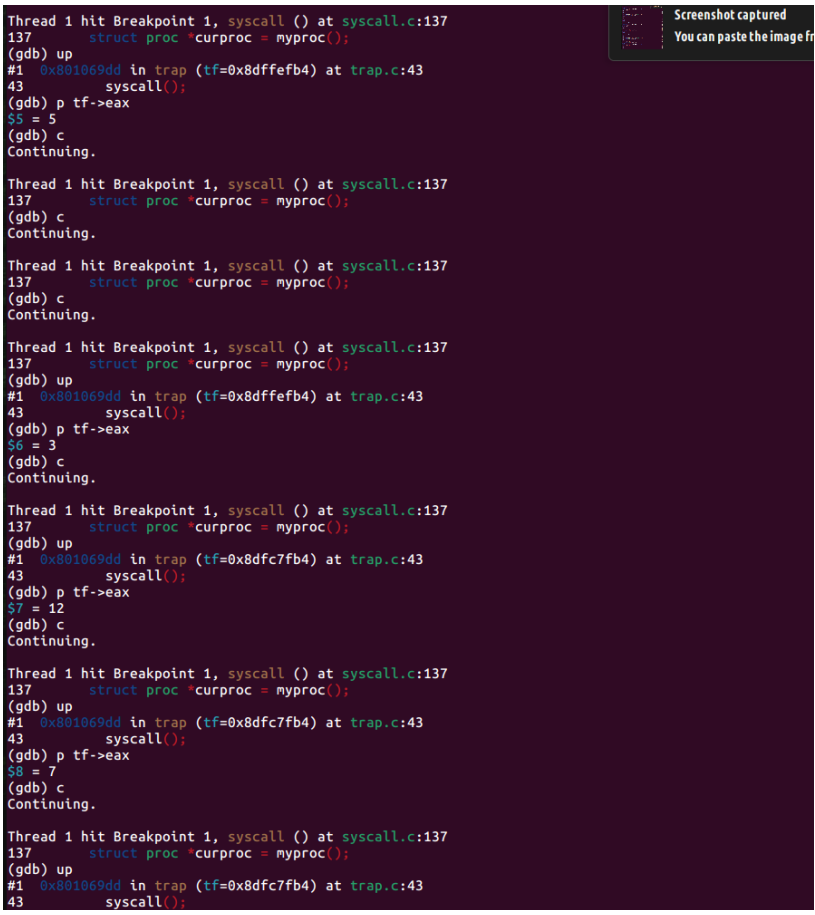
قرار دادن در میک فایل:

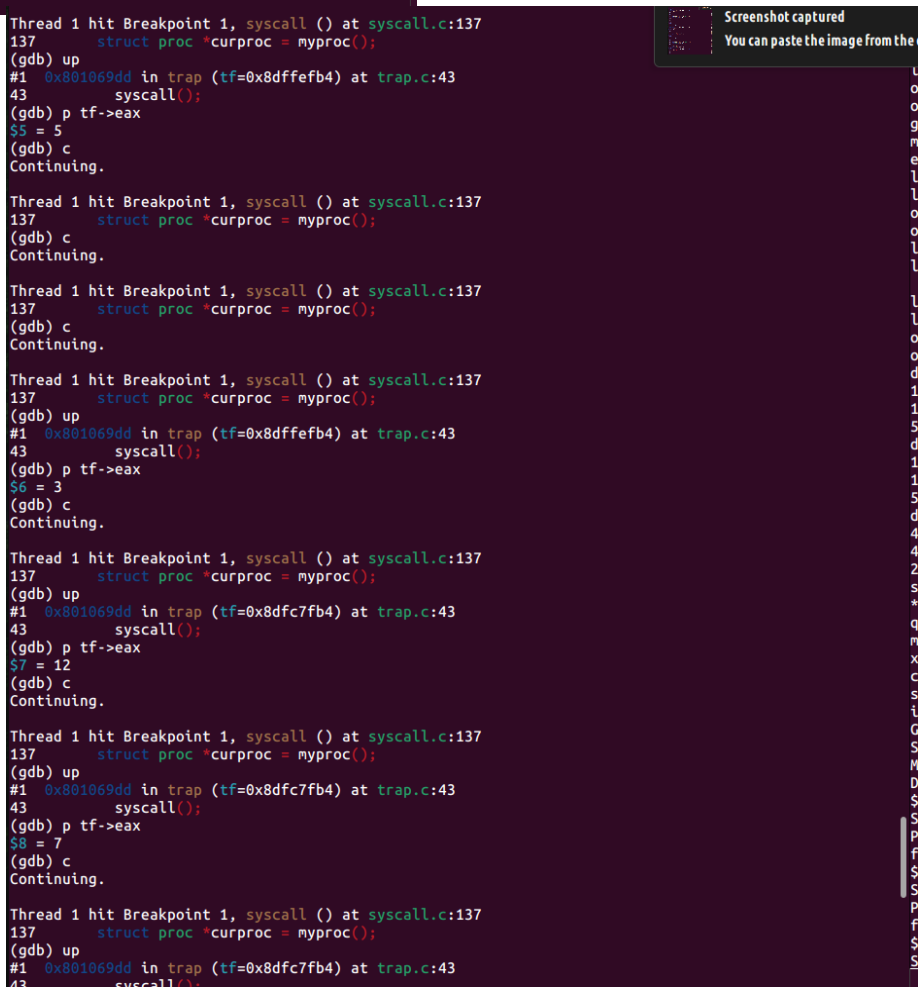


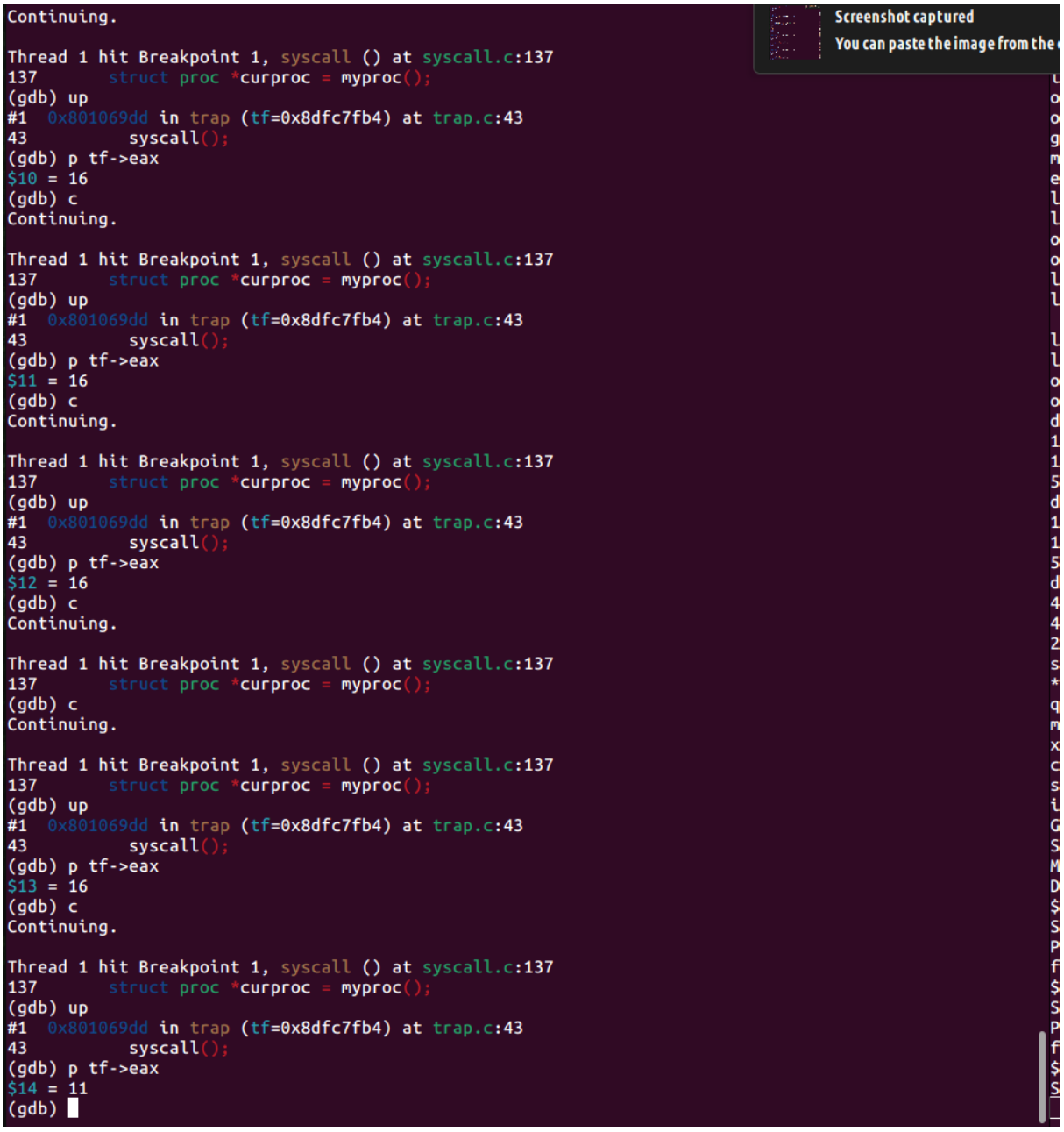
```
    _getpidtest\
```

بخش gdb:


jad@sajjad-taghizade:~/Desktop/os/os_lab_project/lab2/codes$ 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:
<http://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/sajjad/Desktop/os/os_lab_project/lab2/codes/.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/sajjad/Desktop/os/os_lab_project/lab2/codes/.gdbinit
line to your configuration file "/home/sajjad/.config/gdb/gdbinit".
To completely disable this security protection add
set auto-load safe-path /
line to your configuration file "/home/sajjad/.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:
info "(gdb)Auto-loading safe path"
(gdb) file kernel
Reading symbols from kernel...
(gdb) b syscall
Breakpoint 1 at 0x80105890: file syscall.c, line 137.
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x000fff0 in ?? ()
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) c


Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dffefb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$5 = 5
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dffefb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$6 = 3
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dfc7fb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$7 = 12
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dfc7fb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$8 = 7
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dfc7fb4) at trap.c:43
43 syscall();


Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dffefb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$5 = 5
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dffefb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$6 = 3
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dfc7fb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$7 = 12
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dfc7fb4) at trap.c:43
43 syscall();
(gdb) p tf->eax
$8 = 7
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137 struct proc *curproc = myproc();
(gdb) up
#1 0x801069dd in trap (tf=0x8dfc7fb4) at trap.c:43



تحلیل روند:

دستور up یا down و بررسی eax

توضیح کاربرد up/down: دستور up ما را یک فریم در پشته بالا می‌برد (به تابعی که ما را صدا زده). ما از syscall (فریم ۰) به trap (فریم ۱) رفتیم تا به آرگومان آن، یعنی tf ، دسترسی پیدا کنیم.

محتوای رجیستر eax (اولین توقف‌ها): پس از اجرای getpidtest، اولین توقف‌ها مقادیر زیر را در eax نشان دادند :

$1 = 7: این SYS_exec است (فراخوانی exec توسط شل برای اجرای getpidtest).
$2 = 5: این SYS_open است (مربوط به اولین printf در برنامه ما).
$9 = 16: این SYS_write است (مربوط به اولین printf در برنامه ما).

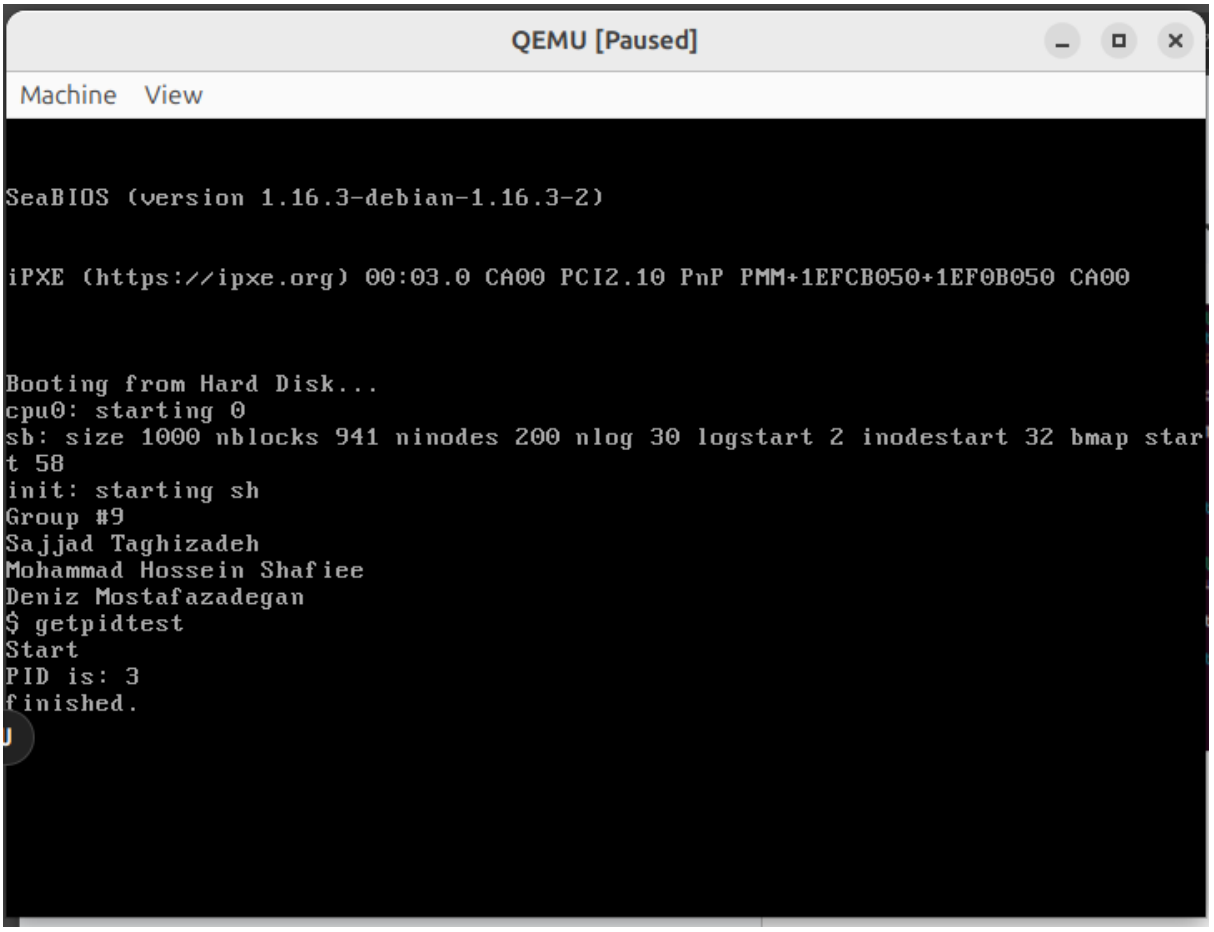
آیا این مقدار برابر با شماره getpid است؟ خیر
علت: برنامه تستی ما قبل از فراخوانی getpid()، ابتدا تابع printf را اجرا می‌کند. تابع printf برای نوشتن روی کنسول، خود نیازمند اجرای فراخوانی‌های سیستمی (open و write) است. بنابراین، GDB ابتدا روی این فراخوانی‌ها متوقف می‌شود، نه روی getpid.

اجرای c تا رسیدن به getpid :با اجرای مکرر دستور c ، از فراخوانی‌های سیستمی اولیه عبور کردیم تا به getpid برسیم.

تحلیل لاگ:
$9 = 16: این SYS_write مربوط به printf اول ("...Calling getpid()") است.
$14 = 11: این SYS_getpid است. این توقف دقیقاً مربوط به اجرای خط pid = getpid(); در کد ما است.

این ردیابی به نشان می‌دهد که شماره فراخوانی سیستمی getpid (که ۱۱ است) قبل از printf دوم و بعد از printf اول، توسط برنامه کاربر در رجیستر eax قرار داده شده و به هسته منتقل شده است.

خروجی در شل xv6:



۱. دستور bt : این دستور پشته فراخوانی را نشان می‌دهد و می‌گوید که چطور به اینجا رسیده‌ایم.

تحلیل خروجی bt:

```
(gdb) bt
#0  syscall () at syscall.c:135
#1  0x8010689d in trap (tf=0x8dffffb4) at trap.c:43
#2  0x8010664f in alltraps () at trapasm.S:20
#3  0x8dffffb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

- **bt چیست؟** bt مخفف **Backtrace** است. این دستور به ما نشان می‌دهد که کدام تابع، تابع فعلی را فراخوانی کرده، و کدام تابع، آن تابع را فراخوانی کرده این زنجیره، مسیر رسیدن به نقطه فعلی را نشان می‌دهد.
- **تحلیل:**

فریم 0 (#0): ما در syscall هستیم

فریم 1 (#1): تابع syscall توسط trap فراخوانی شده است. این منطقی است، زیرا trap کنترل‌کننده مرکزی تمام تله‌ها است و با بررسی شماره تله، تشخیص داده که این یک فراخوانی سیستمی است.

فریم 2 (#2): تابع trap توسط alltraps فراخوانی شده است. alltraps کد اسمبلی است که تمام رجیسترهای برنامه کاربر را روی پشته هسته ذخیره می‌کند تا trap frame ساخته شود.

قبل از آن: قبل از alltraps، کد اسمبلی vector64 اجرا شده که آن هم در نتیجه اجرای دستور int 64 توسط برنامه سطح کاربر فعال شده است.

هدف اجرای این کار:

به طور خلاصه، هدف این بود که:

1. مسیر اجرا را با استفاده از bt ، دیدیم که اجرای یک فراخوانی سیستمی یک پرش ساده به یک تابع نیست، بلکه یک زنجیره پیچیده است:
    - از برنامه کاربر
    - به بردار وقفه در اسمبلی
    - به alltraps برای ذخیره رجیسترها
    - به trap کنترل‌کننده C مرکزی
    - و در نهایت به syscall

2. مکانیسم انتقال داده را : نشان داده شد که برنامه سطح کاربر، شماره فراخوانی سیستمی را در رجیستر eax قرار می‌دهد. شما این عدد را در tf->eax دیدیم.

پایان بخش های تحلیلی

ارسال آرگومان های فراخوانی های سیستمی:

برای اعمال این بخش ابتدا یک در فایل syscall.h یک شماره جدید اضافه کردیم (23):

```
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_make_duplicate 22
#define SYS_simplearith 23
```

و در ارایه syscalls[] اضافه کردیم :

```
static int (*syscalls[])(void) = {
[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_make_duplicate] sys_make_duplicate,
[SYS_simplearith] sys_simplearith,
};
```

سپس تابع sys_simplearith با مشخصات داده شده رو در فایل syscall.c اضافه کردیم :

```
int
sys_simplearith(void)
{
  struct proc *p = myproc();

  int a = p->tf->ebx;
  int b = p->tf->ecx;

  int result = (a + b) * (a - b);

  cprintf("Calc: (%d+%d)*(%d-%d) = %d\n", a, b, a, b, result);

  return result;
}
```

تا این خواسته هارا انجام دهد. بدلیل یک دستور جدید باید برای اینکه از استک نخونه ورودی هارو یک wrapper در سطح کاربر اضافه میکردیم :

```
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(make_duplicate)

.globl simplearith
simplearith:
    pushl %ebx
    movl 8(%esp), %ebx
    movl 12(%esp), %ecx
    movl $SYS_simplearith, %eax
    int $T_SYSCALL
    popl %ebx
    ret
```

با انجام این تغییرات این دستور ما به سیستم کال اضافه شد.

نتیجه خروجی در ترمینال qemu:

```
$ test_simplearith
Calc: (5+3)*(5-3) = 16
user: simple_arith(5,3) returned 16
$ _
```

فراخوانی‌های سیستمی:

1. پیاده سازی فراخوانی سیستمی ایجاد نسخه کپی فایل:

مراحل انجام کار:

اضافه کردن شماره سیستم کال به syscall.h:

```
#define SYS_make_duplicate 22
```

معرفی تابع به هسته با این ۲ خط در syscall.c:

```
extern int sys_make_duplicate(void);
```

```
[SYS_make_duplicate] sys_make_duplicate,
```

پیاده سازی منطق duplicate در فایل sysfile.c:

```
int sys_make_duplicate(void)
{
  char *src_path;
  char dst_path[MAXPATH];
  struct inode *src_inode, *dst_inode;
  char buf[BSIZE];
  uint offset;
  int n;

  argstr(0, &src_path);

  safestrcpy(dst_path, src_path, MAXPATH);

  uint len = strlen(dst_path);

  safestrcpy(dst_path + len, "_copy", MAXPATH - len);

  if ((src_inode = namei(src_path)) == 0)
  {
    cprintf("ERROR: source file not found\n");
    return -1;
  }

  ilock(src_inode);

  begin_op();
  dst_inode = create(dst_path, T_FILE, 0, 0);

  end_op();

  offset = 0;

  while ((n = readi(src_inode, buf, offset, BSIZE)) > 0)
  {

    begin_op();
    writei(dst_inode, buf, offset, n);
    end_op();
    offset += n;
  }

  iunlockput(src_inode);
  iunlockput(dst_inode);

  cprintf("make_duplicate: Success(in sysytem call)\n");
  return 0;
```

بخش های سمت کاربر:

پروتوتایپ تابع را برای برنامه‌های C در فایل user.h:

```
int make_duplicate(const char*);
```

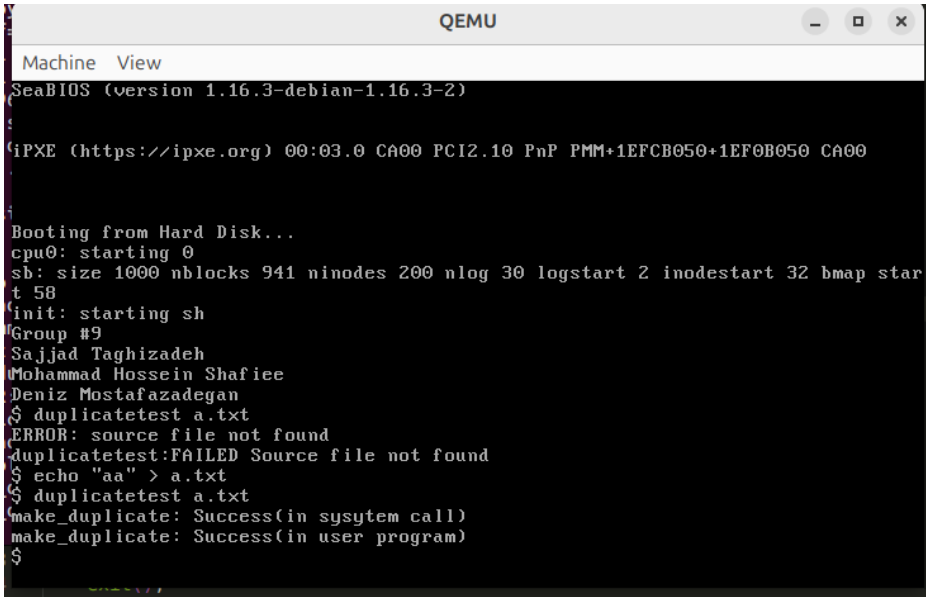
wrapper اسمبلی در usys.s:

```
SYSCALL(make_duplicate)
```

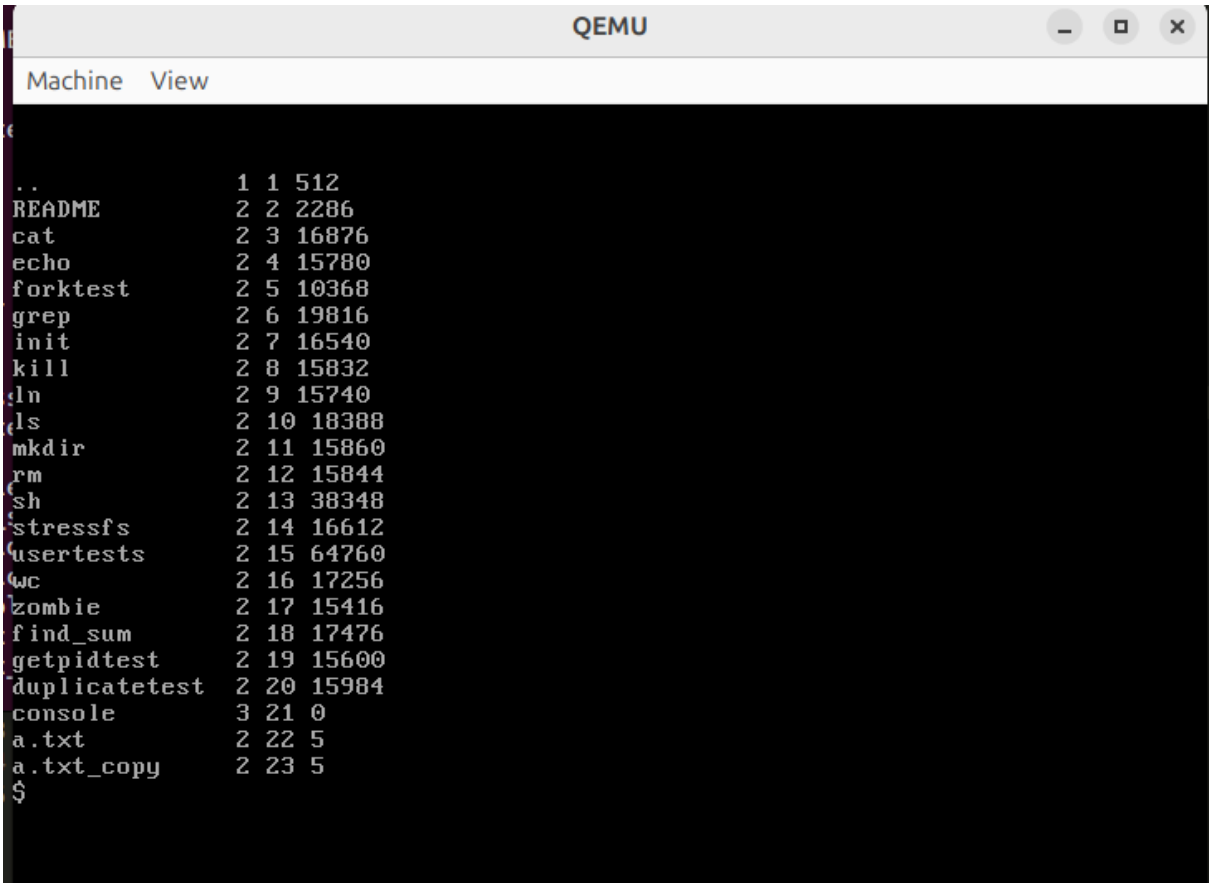
ساخت فایل تست برای تست سیستم کال:

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

int main(int argc, char *argv[])
{

    char *filename = argv[1];
    int ans = make_duplicate(filename);

    if (ans == 0)
    {
        printf(1, "make_duplicate: Success(in user program)\n");
    }
    else if (ans == -1)
    {
        printf(1, "duplicatetest:FAILED Source file not found \n");
    }
    else
    {
        printf(1, "duplicatetest: FAILED\n");
    }

    exit();
}
```

تست اجرای درست و هندل خطای نبودن یک فایل و درخواست برای کپی کردنش:



فایل ها بعد اجرای دستور ls:



2. پیاده سازی فراخوانی سیستمی بدست آوردن اعضای خانواده یک پردازه

مانند بقیه بخش ها در اینجا هم ابتدا باید سیستم کال جدید را تعریف کنیم.

syscall.h:

```
#define SYS_show_process_family 24
```

user.h:

```
int show_process_family(int);
```

usys.S:

```
SYSCALL(show_process_family)
```

syscall.c:

```
extern int sys_show_process_family(void);
```

همچنین در همین فایل باید مقداری که در فایل syscall.h تعریف کردیم را با تابع سیستم کال مورد نظر مپ کنیم:

```
[SYS_show_process_family] sys_show_process_family,
```

sysproc.c:

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

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

  return show_process_family(pid);
}
```

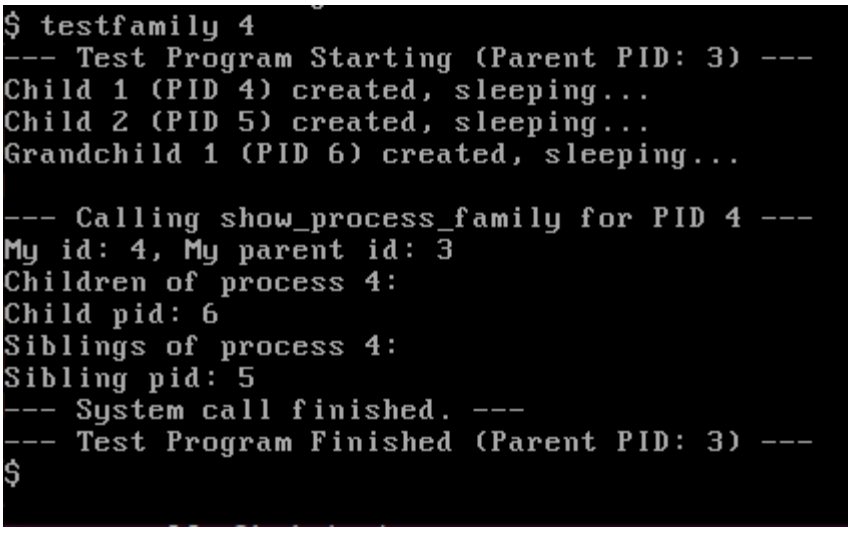
این تابعی است که وقتی سیستم کال صدا زده می‌شود، اجرا می‌شود. اما در این فایل محدودیتی داشتیم و نمیتوانستیم از ptable استفاده کنیم. بنابراین یک تابع جدید در فایل proc.c ایجاد میکنیم و آن را در تابع این سیستم کال صدا می‌زنیم و مقادیر مورد نیاز را به آن پاس می‌دهیم. در اینجا ابتدا pid را با استفاده از تابع argint میخوانیم و آن را به تابع show_process_family در فایل proc.c پاس می‌دهیم.

این تابع در فایل proc.c ابتدا پردازه با آیدی مورد نظر را پیدا می‌کند و سپس بعد از پیدا کردن پرنت آن siblings های آن را پیدا کرده. همچنین با استفاده از پردازه مورد نظر بچه های آن پردازه را نیز پیدا می‌کنیم.

در نهایت هم یک فایل تست اضافه کردیم تا بتوان صحت این سیستم کال را بررسی کنیم.

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

int
main(int argc, char *argv[])
{
  int pid_child1, pid_child2, pid_grandchild1;
  int parent_pid = getpid();
  int pid_to_test;

  printf(1, "--- Test Program Starting (Parent PID: %d) ---\n", parent_pid);

  pid_child1 = fork();
  if(pid_child1 == 0){
    pid_grandchild1 = fork();
    if (pid_grandchild1 == 0) {
      printf(1, "Grandchild 1 (PID %d) created, sleeping...\n", getpid());
      sleep(300);
      exit();
    }
    printf(1, "Child 1 (PID %d) created, sleeping...\n", getpid());
    sleep(300);
    wait();
    exit();
  }

  pid_child2 = fork();
  if(pid_child2 == 0){
    printf(1, "Child 2 (PID %d) created, sleeping...\n", getpid());
    sleep(300);
    exit();
  }

  sleep(20);

  if(argc < 2){
    printf(1, "\n--- No PID passed to test. ---\n");
    printf(1, "Created sleeping process family:\n");
    printf(1, "  Child 1 PID: %d\n", pid_child1);
    printf(1, "  Child 2 PID: %d\n", pid_child2);
    printf(1, "  (Grandchild was created by Child 1)\n");
    printf(1, "\nRun again with a PID to test, e.g.: testfamily %d\n", parent_pid);
  } else {
    pid_to_test = atoi(argv[1]);
    printf(1, "\n--- Calling show_process_family for PID %d ---\n", pid_to_test);

    show_process_family(pid_to_test);

    printf(1, "--- System call finished. ---\n");
  }
  wait();
  wait();

  printf(1, "--- Test Program Finished (Parent PID: %d) ---\n", parent_pid);
  exit();
}
```

در اینجا اول پردازه اصلی یک فرزند می‌سازد و آن فرزند یک فرزند دیگر می‌سازد. سپس پردازنده اصلی یک فرزند دوم نیز می‌سازد. حال با کمک تابع sleep می‌توان این پردازه ها را به مدت دلخواه نگه داشت تا بتوان سیستم کال را روی آن‌ها تست کرد.



همانطور که در این تصویر مشخص است، پردازه با شماره 4، 3 parent و 6 child دارد. همچنین یک sibling با آیدی ۵ نیز دارد.

3. پیاده سازی فراخوانی سیستمی جستجوی محتوا در فایل

در مرحله اول تعاریف را در syscall وارد کردیم:

```
[SYS_grep_syscall]   sys_grep_syscall,
```

```
#define SYS_grep_syscall 26
```

```
extern int sys_grep_syscall(void);
```

تابع این سیستم کال رو در فایل sysfile اضافه کردیم + تابع پیش نیازش :

طبق محدودیت‌های پروژه، در هسته به کتابخانه استاندارد string.h دسترسی وجود ندارد. بنابراین یک تابع کمکی ساده برای جستجوی زیررشته در همان فایل sysfile.c پیاده‌سازی شد

```
static int
contains_substring(const char *haystack, const char *needle)
{
  if (*needle == 0)
    return 1;
  for (; *haystack; haystack++)
  {
    const char *h = haystack;
    const char *n = needle;
    while (*h && *n && *h == *n)
    {
      h++;
      n++;
    }
    if (*n == 0)
      return 1;
  }
  return 0;
}
```

در wrapper اضافه کردیم:

```
SYSCALL(grep_syscall)
```

تعریف در user.h :

```
int grep_syscall(const char* keyword, const char* filename, char* user_buffer,
    int buffer_size);
```

ایجاد برنامه تستی grep_test.c

برای تست این فراخوانی سیستمی، یک برنامه سطح کاربر نوشته شد که سه چیز کلیدی را بررسی می‌کند:

1. **موفقیت:** جستجوی کلمه‌ای (xv6) که در فایل (README) وجود دارد.
2. **شکست (کلیدواژه):** جستجوی کلمه‌ای که وجود ندارد.
3. **شکست (فایل):** جستجو در فایلی که وجود ندارد.

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

int
main(void)
{
  char buf[1024];
  int len;

  printf(1, "--- grep_syscall Test ---\n");

  printf(1, "Test 1: Searching for 'xv6' in 'README' (should succeed)\n");

  len = grep_syscall("xv6", "README", buf, sizeof(buf));

  if (len > 0) {
    printf(1, "SUCCESS: Found line (len %d): %s\n", len, buf);
  } else {
    printf(1, "FAILED: Word 'xv6' not found (ret=%d)\n", len);
  }

  printf(1, "\nTest 2: Searching for 'nonexistent_word' in 'README' (should fail)\n");

  len = grep_syscall("nonexistent_word", "README", buf, sizeof(buf));

  if (len == -1) {
    printf(1, "SUCCESS: Syscall correctly returned %d (word not found)\n", len);
  } else {
    printf(1, "FAILED: Syscall returned %d, but expected -1\n", len);
  }

  printf(1, "\nTest 3: Searching in 'nonexistent_file' (should fail)\n");

  len = grep_syscall("xv6", "nonexistent_file", buf, sizeof(buf));

  if (len == -1) {
    printf(1, "SUCCESS: Syscall correctly returned %d (file not found)\n", len);
  } else {
    printf(1, "FAILED: Syscall returned %d, but expected -1\n", len);
  }

  printf(1, "--- grep_syscall Test Complete ---\n");
  exit();
}
```

4 .پیاده‌سازی فراخوانی سیستمی تنظیم اولویت

مراحل انجام کار:

اضافه کردن شماره سیستم کال به syscall.h:

```
#define SYS_set_priority_syscall 25
```

معرفی تابع به هسته با این ۲ خط در syscall.c:

```
[SYS_set_priority_syscall] sys_set_priority_syscall,
```

```
extern int sys_set_priority_syscall(void);
```

بخش های سمت کاربر:

پروتوتایپ تابع را برای برنامه‌های C در فایل user.h:

```
int set_priority_syscall(int pid, int priority);
```

wrapper اسمبلی در usys.s:

```
SYSCALL(set_priority_syscall)
```

یک فیلد جدید به نام int priority; به ساختار struct proc در فایل kernel/proc.h اضافه شد.

```
You, 3 hours ago | 1 author (You)
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];


  int priority;       You, 3 ho

};
```

در تابع allocproc() (واقع در proc.c)، به تمام پردازه‌هایی که تازه ایجاد می‌شوند، یک اولویت پیش‌فرض (سطح عادی= ۱) اختصاص داده شد تا رفتار سیستم در حالت عادی حفظ شود.

```
p->priority = 1;
```

پیاده‌سازی تابع sys_set_priority_syscall در proc.c:

تابع ابتدا آرگومان‌های pid و priority را با استفاده از argint() از پشته کاربر می‌خواند.

بررسی می‌کند که اولویت در بازه مجاز تعریف‌شده توسط پروژه (۰=بالا، ۱=عادی، ۲=پایین) باشد.

با گرفتن قفل ptable.lock، جدول پردازه‌ها را جستجو می‌کند.

در صورت یافتن pid مورد نظر، فیلد p->priority آن را به‌روزرسانی می‌کند.

در صورت موفقیت 0 و در صورت یافت نشدن PID، مقدار -1 را برمی‌گرداند.

```
int
sys_set_priority_syscall(void)
{
  int pid, priority;
  struct proc *p;

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


  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->priority = priority;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);

  cprintf("set_priority: PID %d not found\n", pid);
  return -1;
}
```

تغییر منطق زمان‌بند:

منطق قبلی (Round-Robin ساده) که اولین پردازه RUNNABLE را انتخاب می‌کرد، حذف شد.

منطق جدید در یک حلقه for(;;)، ابتدا کل جدول پردازه‌ها (ptable) را برای یافتن یک پردازه RUNNABLE با اولویت بالا (۰) جستجو می‌کند.

اگر پردازه‌ای با اولویت بالا پیدا نشد، *دوباره* جدول را از ابتدا برای یافتن پردازه‌های با اولویت عادی (۱) جستجو می‌کند.

اگر باز هم پیدا نشد، برای اولویت پایین (۲) جستجو می‌کند.

به محض یافتن اولین پردازه RUNNABLE در بالاترین اولویت موجود، آن را برای اجرا انتخاب کرده (goto run_process) و با swtch به آن منتقل می‌شود.

```
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    sti();

    struct proc *best_p = 0;
    int current_priority;

    acquire(&ptable.lock);
    for(current_priority = 0; current_priority <= 2; current_priority++){

      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE || p->priority != current_priority)
          continue;

        best_p = p;
        goto run_process;
      }
    }
    release(&ptable.lock);
    continue;

run_process:
    p = best_p;

    // Switch to chosen process.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    c->proc = 0;

    release(&ptable.lock);

  }
}
```

ایجاد برنامه تستی:

برنامه والد دو پردازه فرزند با fork() ایجاد می‌کند.

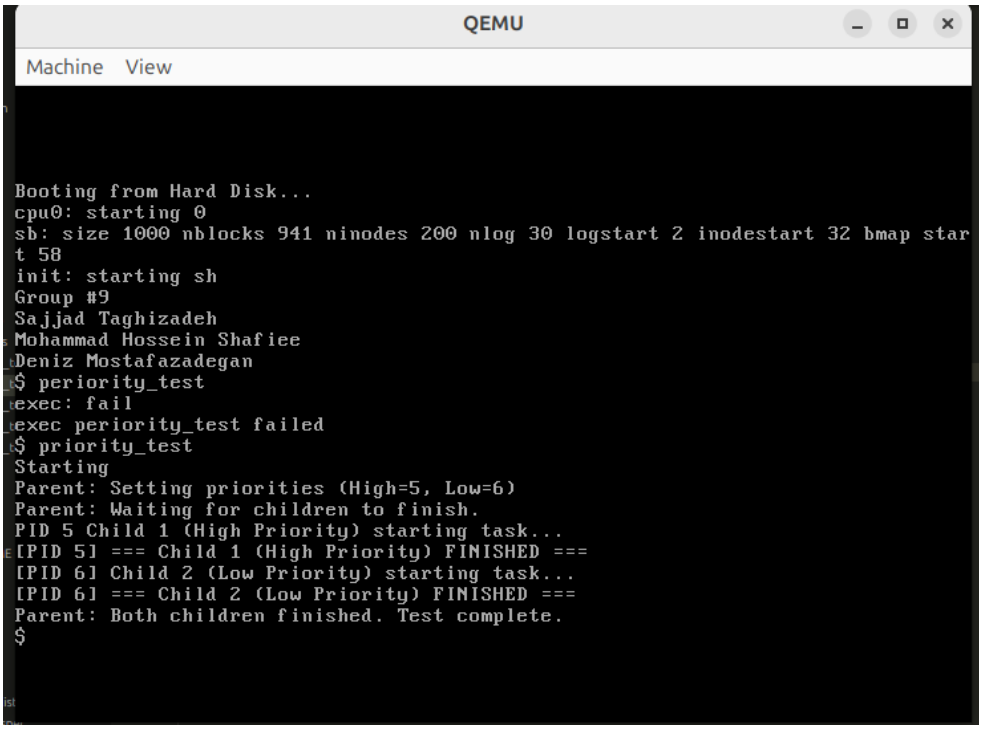
هر دو فرزند یک کار محاسباتی سنگین (CPU-intensive) و طولانی را اجرا می‌کنند.

والد بلافاصله پس از ایجاد فرزندان، با استفاده از set_priority_syscall، اولویت یک فرزند را "بالا" (۰) و دیگری را "پایین" (۲) تنظیم می‌کند.

نتیجه مشاهده شده: فرزندی که اولویت بالا داشت، تمام زمان CPU را در اختیار گرفت و کار محاسباتی خود را به طور کامل تمام کرد . تنها پس از اتمام کار او، زمان‌بند به فرزند با اولویت پایین اجازه اجرا داد. این خروجی، موفقیت‌آمیز بودن پیاده‌سازی زمان‌بند اولویت‌بندی شده را اثبات کرد.

```c
void cpu_intensive_task() {
  volatile unsigned long long i;
  for(i = 0; i < 500000000; i++) {
    asm("nop");
  }
}

int
main(void)
{
  int pid1, pid2;

  printf(1, "Starting\n");

  pid1 = fork();
  if(pid1 == 0) {      You, 3 hours ago • add setpriority
    printf(1, "PID %d Child 1 (High Priority) starting task...\n", getpid());
    cpu_intensive_task();
    printf(1, "[PID %d] === Child 1 (High Priority) FINISHED ===\n", getpid());
    exit();
  }

  pid2 = fork();
  if(pid2 == 0) {
    printf(1, "[PID %d] Child 2 (Low Priority) starting task...\n", getpid());
    cpu_intensive_task();
    printf(1, "[PID %d] === Child 2 (Low Priority) FINISHED ===\n", getpid());
    exit();
  }

  printf(1, "Parent: Setting priorities (High=%d, Low=%d)\n", pid1, pid2);

  set_priority_syscall(pid1, 0);
  set_priority_syscall(pid2, 2);

  printf(1, "Parent: Waiting for children to finish.\n");

  wait();
  wait();

  printf(1, "Parent: Both children finished. Test complete.\n");
  exit();
}
```

نتیجه تست در ترمینال qemu:



اجرای priority_test

1. Parent: Setting priorities (High=5, Low=6):
   - برنامه والد شما اجرا شده و دو فرزند با PIDهای ۵ و ۶ ایجاد کرده.
   - بلافاصله، اولویت PID 5 رو به "بالا" (High) و اولویت PID 6 رو به "پایین" (Low) تغییر داده.
2. [PID 5] === Child 1 (High Priority) FINISHED ===:
   - زمان‌بند تشخیص داده که یک پردازه با اولویت بالا (PID 5) آماده اجراست.
   - به همین دلیل، *تمام* زمان CPU رو به PID 5 اختصاص داده تا جایی که این پردازه کل کار محاسباتی سنگینش رو کامل تمام کرده.
3. [PID 6] === Child 2 (Low Priority) FINISHED ===:
   - *تنها پس از اتمام* کار پردازه با اولویت بالا، زمان‌بند به سراغ پردازه با اولویت پایین (PID 6) رفته و به اون اجازه داده تا کار محاسباتیش رو شروع و تمام کنه.

نتیجه‌گیری: این خروجی به نشون می‌ده که سیستم دیگه Round-Robin (نوبتی) کار نمی‌کنه، بلکه یک زمان‌بندی اولویت‌بندی شده (Priority-based) واقعی رو پیاده‌سازی کرده که در اون، پردازه‌های با اولویت بالا، پردازه‌های با اولویت پایین رو starve می‌دن.