# بسم الله الرحمن الرحیم

# گزارش پروژه دوم آزمایشگاه سیستم عامل


## دکتر کارگهی

## دکتر زحمت‌کش

## پوریا مهدیان ۸۱۰۱۰۱۵۳۰

## محمدطاها مجلسی کوپائی ۸۱۰۱۰۱۵۰۴

## علیرضا کریمی ۸۱۰۱۰۱۴۹۲


## آبان ۱۴۰۳

لینک گیت هاب:

https://github.com/tahamajs/xv6-Modified_OS_Lab_Projects/tree/03950814ccd182af8aaca96eec2f4489b283b6df/Phase2

# پاسخ به سوالات

## سوال ۱: کتابخانه‌های سطح کاربر در xv6، برای ایجاد ارتباط میان برنامه‌های کاربر و کرنل به کار می‌روند. این کتابخانه‌ها شامل توابعی هستند که از فراخوانی‌های سیستمی استفاده می‌کنند تا دسترسی به منابع سخت‌افزاری و نرم افزاری سیستم عامل ممکن شود. با تحلیل فایل‌های موجود در متغیر ULIB در xv6، توضیح دهید که چگونه این کتابخانه‌ها از فراخوانی‌های سیستمی بهره می‌برند؟ همچنین، دلایل استفاده از این فراخوانی‌ها و تاثیر آنها بر عملکرد و قابلیت حمل برنامه‌ها را شرح دهید.

توی سیستم‌عامل xv6، کتابخونه‌های سطح کاربر مثل `ULIB` یه رابط بین برنامه‌های کاربر و کرنل ایجاد می‌کنن. این کتابخونه‌ها توابعی دارن که بهشون میگیم "Wrapper" که کارشون اینه که سیستم‌کال‌ها رو طوری ارائه بدن که برنامه‌های کاربر بتونن بدون دسترسی مستقیم به منابع سخت‌افزاری یا نرم‌افزاری، کارهایی مثل مدیریت فایل یا ارتباط بین فرآیندها رو انجام بدن.

### چجوری `ULIB` از سیستم‌کال‌ها استفاده می‌کنه

فایل‌هایی که توی `ULIB` هستن، مثل `ulib.c` و `usys.S`، توابعی رو دارن که سیستم‌کال‌ها رو پوشش می‌دن و باعث می‌شن برنامه‌های کاربر راحت‌تر بتونن به این سیستم‌کال‌ها دسترسی داشته باشن. برای مثال، توابعی مثل `write` که برای نوشتن توی فایل هست، از `syscall` استفاده می‌کنه تا درخواستش به کرنل فرستاده بشه. این‌جوری `write` خودش مستقیم به کرنل وصل نمیشه، بلکه از `syscall` کمک می‌گیره تا کرنل کار رو انجام بده.

### مثال ساده از تابع `write` در `ulib.c`

در `ulib.c`، تابع `write` یک نمونه ساده از یک "Wrapper" هست. کد این تابع به شکل زیره:

```
int write(int fd,const void *buf,int n){return syscall(SYS_write,fd,buf,n);}z
```

اینجا `write` به عنوان یه واسط عمل می‌کنه و به جای اینکه مستقیم به کرنل وصل بشه، از `syscall` استفاده می‌کنه. `SYS_write` هم شناسه‌ی سیستم‌کال `write` هست که کرنل از طریق اون متوجه می‌شه که باید عملیات نوشتن رو انجام بده.

## مزایای این کار چیه؟

1. **انتزاع و قابلیت حمل**: `ULIB` باعث میشه که برنامه‌ها نیازی به دونستن جزئیات کرنل نداشته باشن. اگه xv6 بخواد روی یه سیستم جدید پورت بشه، تنها چیزی که باید تغییر کنه پیاده‌سازی `syscall` هست و برنامه‌های کاربر همونطور باقی می‌مونن.
2. **ایمنی و نگهداری**: این روش تضمین می‌کنه که همه دسترسی‌ها به کرنل از طریق یه روش استاندارد انجام بشه. اگه تغییری در امنیت یا مدیریت خطا نیاز باشه، این تغییرات می‌تونه فقط تو توابع `ULIB` اعمال بشه، بدون اینکه برنامه‌های کاربر تغییر کنن.

## چرا از سیستم‌کال‌ها استفاده می‌کنیم؟

- **انتزاع**: برنامه‌ها نیازی ندارن بدونن کرنل دقیقاً چطوری کار می‌کنه.
- **قابلیت حمل**: برنامه‌ها می‌تونن روی نسخه‌های مختلف xv6 بدون تغییر اجرا بشن.
- **کارایی**: `ULIB` کمک می‌کنه که کدهای کاربر مرتب‌تر و ساده‌تر بشن و از تکرار کدها جلوگیری بشه.

## نکته‌ای درباره پیاده‌سازی `write`

تابع `write` توی سیستم‌های مختلف یه تابع رایجه، اما پیاده‌سازی اون ممکنه بسته به سخت‌افزار یا کرنل فرق کنه. `ULIB` این تفاوت‌ها رو پنهان می‌کنه و باعث میشه برنامه‌ها بدون نیاز به تغییر، روی سیستم‌های مختلف هم کار کنن.

به‌طور کلی، کتابخانه‌های سطح کاربر `ULIB` تو xv6 به برنامه‌ها کمک می‌کنن که به راحتی و با امنیت بیشتری از منابع سیستم‌عامل استفاده کنن و همزمان قابل‌حمل‌تر و پایدارتر باشن.

## سوال ۲: فراخوانی‌های سیستمی تنها روش برای تعامل برنامه‌های کاربر با کرنل نیستند. چه روش‌های دیگری در لینوکس وجود دارند که برنامه‌های سطح کاربر می‌توانند از طریق آنها به کرنل دسترسی داشته باشند؟ هر یک از این روش‌ها را به اختصار توضیح دهید.

### 1. سیستم‌فایل‌های /proc و /sys

- **/proc:**
  - این سیستم‌فایل مجازی شامل اطلاعاتی در مورد وضعیت کرنل و فرآیندهای در حال اجرا در سیستم است. این فایل‌ها در واقع به برنامه‌های کاربر اجازه می‌دهند که به اطلاعات داخلی کرنل دسترسی پیدا کنند، بدون اینکه نیازی به فراخوانی مستقیم سیستم کال‌ها داشته باشند.
  - برخی از فایل‌های رایج در `/proc` عبارتند از:
    - `/proc/cpuinfo`: اطلاعات مربوط به پردازنده (مثل تعداد هسته‌ها، مدل، سرعت کلاک و ...).
    - `/proc/meminfo`: اطلاعات مربوط به حافظه سیستم، شامل حافظه آزاد، استفاده‌شده و ... .
    - `/proc/[pid]/status`: وضعیت فرآیندها، مانند میزان حافظه مصرفی و وضعیت اجرایی.
  - برنامه‌ها می‌تونن با استفاده از توابع فایل استاندارد مثل `read`، `open` و `write` به این فایل‌ها دسترسی پیدا کنند.
- **/sys:**
  - این سیستم‌فایل برای دسترسی به اطلاعات سخت‌افزاری و پارامترهای سیستم طراحی شده. در این دایرکتوری، اطلاعاتی در مورد دستگاه‌ها، درایورهای سخت‌افزاری و تنظیمات کرنل وجود داره.
  - به عنوان مثال:
    - `/sys/class/net`: اطلاعات مربوط به رابط‌های شبکه.
    - `/sys/block/sda`: اطلاعات مربوط به دستگاه ذخیره‌سازی.
  - برنامه‌ها می‌تونن از این دایرکتوری برای انجام تغییرات در پارامترهای دستگاه‌ها، مثل تغییر سرعت پردازنده یا تغییر تنظیمات دستگاه‌های ورودی/خروجی، استفاده کنن.

### 2. ioctl (کنترل ورودی/خروجی)

- `ioctl` یه سیستم‌کال است که به برنامه‌های کاربر اجازه می‌ده که دستورهای خاصی به درایورهای دستگاه ارسال کنند. این سیستم‌کال می‌تواند برای انجام عملیات‌هایی استفاده شود که نمی‌توان آن‌ها را با توابع ساده خواندن و نوشتن انجام داد.
- مثلاً می‌تونید از `ioctl` برای تنظیم ویژگی‌های یک دستگاه ورودی مانند ماوس یا کیبورد استفاده کنید یا حتی می‌تونید برای پیکربندی پارامترهای سخت‌افزاری به کار ببرید.
- یکی از استفاده‌های معمول `ioctl` در برنامه‌های شبکه است، جایی که برای پیکربندی رابط‌های شبکه یا مدیریت جداول مسیریابی از آن استفاده می‌شود.

## 3. سوکت‌ها (Netlink و Unix Domain Sockets)

- **Netlink Sockets**:
  - این سوکت‌ها به برنامه‌های کاربر این امکان رو می‌دهند که با ماژول‌های کرنل ارتباط برقرار کنند، معمولاً برای مسایل مربوط به شبکه و نظارت بر سیستم. این سوکت‌ها به ویژه در زمان نیاز به ارتباط با کرنل برای مدیریت تنظیمات شبکه یا تعامل با ماژول‌های امنیتی مفید هستن.
  - برای مثال، می‌توانید از Netlink برای پیکربندی رابط‌های شبکه، تغییر آدرس IP یا مسیریابی استفاده کنید.
  - از Netlink برای ارسال پیام‌ها به کرنل و دریافت اطلاعات سیستم مانند وضعیت شبکه یا آمار پردازش بسته‌ها می‌توان استفاده کرد.
- **Unix Domain Sockets**:
  - این سوکت‌ها برای ارتباط بین فرآیندهای مختلف درون یک سیستم استفاده می‌شوند. برخلاف سوکت‌های شبکه‌ای که برای ارتباط بین سیستم‌های مختلف طراحی شده‌اند، Unix Domain Sockets فقط در یک سیستم واحد عمل می‌کنند.
  - این سوکت‌ها برای IPC (ارتباط بین فرآیندها) مورد استفاده قرار می‌گیرند. به عنوان مثال، فرآیندهایی که نیاز به ارسال داده‌های سریع به یکدیگر دارند، می‌توانند از این سوکت‌ها استفاده کنند.

## 4. حافظه مشترک (Shmget، mmap)

- حافظه مشترک یک روش کارآمد برای اشتراک‌گذاری داده‌ها بین فرآیندهای مختلف است. این روش از طریق تخصیص نواحی خاصی از حافظه به فرآیندها این امکان رو فراهم می‌کنه که داده‌ها رو مستقیماً به اشتراک بذارند بدون اینکه نیاز به کپی کردن داده‌ها بین فرآیندها باشه.
- **shmget**: این سیستم‌کال برای ایجاد یا دسترسی به یک بخش حافظه مشترک استفاده می‌شود.
- **mmap**: این سیستم‌کال به فرآیندها اجازه می‌دهد که یک منطقه از حافظه (مثل بافر دستگاه یا بخش‌های خاص حافظه فیزیکی) رو به فضای کاربری خودشون مپ کنند. این روش می‌تواند برای انتقال داده‌ها با سرعت بالا، به خصوص برای پردازش‌های ویدئویی یا صوتی، مفید باشد.

## 5. سیگنال‌ها

- سیگنال‌ها یک روش برای اطلاع‌رسانی کرنل به برنامه‌های کاربر درباره رویدادهای مختلف، مثل خطاها یا درخواست‌های خاص هستند.
- به عنوان مثال:
  - `SIGSEGV`: خطای دسترسی به حافظه (Segmentation Fault).
  - `SIGTERM`: درخواست خاتمه به یک فرآیند.
  - `SIGINT`: قطع فرآیند به واسطه سیگنال Ctrl+C.
- برنامه‌ها می‌تونن از سیگنال‌ها برای واکنش به رویدادهای خاص استفاده کنن، مثلاً وقتی که یه سیگنال خاص به فرآیند ارسال میشه، برنامه می‌تونه یک تابع خاص (handler) رو اجرا کنه که رفتار مشخصی رو انجام بده.

## 6. فایل‌های دستگاه

- فایل‌های دستگاه در دایرکتوری `dev/` قرار دارند و نمایانگر دستگاه‌های سخت‌افزاری سیستم هستند. برنامه‌ها می‌توانند با استفاده از توابع فایل معمول (مثل `read`، `open` و `write`) با این فایل‌ها تعامل داشته باشند تا به دستگاه‌ها دسترسی پیدا کنند.
- به عنوان مثال:
  - `dev/sda/`: فایل مربوط به دیسک‌های سخت.
  - `dev/tty/`: فایل مربوط به ترمینال.
- برنامه‌ها می‌تونن با باز کردن این فایل‌ها و انجام عملیات‌های ورودی/خروجی روی اون‌ها، به دستگاه‌های سخت‌افزاری دسترسی پیدا کنن.

## 7. eBPF (فیلتر بسته برکلی گسترش‌یافته)

- eBPF یک فناوری قدرتمند است که به برنامه‌های کاربر این امکان رو می‌ده که برنامه‌های خود رو در کرنل اجرا کنن، اما به صورت امن و محدودشده. این فناوری به برنامه‌های کاربر این امکان رو می‌ده که فیلترهای شبکه، نظارت بر عملکرد سیستم یا ردیابی رو در سطح کرنل انجام بدن، بدون اینکه نیاز به تغییرات در کد کرنل باشه.
- eBPF از برنامه‌های کاربر برای انجام عملیات‌هایی مثل تجزیه و تحلیل ترافیک شبکه، شبیه‌سازی بسته‌ها یا نظارت بر سیستم‌های ورودی/خروجی استفاده می‌کنه. این رویکرد به کاهش هزینه‌های پردازشی و افزایش امنیت کمک می‌کنه.

## سوال ۳:

**دسترسی انتخابی به تله‌ها**: نه همه تله‌ها و وقفه‌ها باید از حالت کاربر قابل دسترس باشند. برای مثال:

- **سیستم کال‌ها** (مثل `T_SYSCALL`) نقاط ورود کنترل‌شده‌ای به هسته هستند که به برنامه‌های کاربر این امکان را می‌دهند که درخواست‌هایی مثل دسترسی به فایل‌ها یا مدیریت حافظه کنند، با انجام بررسی‌های لازم.
- **وقفه‌های سخت‌افزاری یا همون interrupt ها**(مثل وقفه‌های مربوط به دیسک‌ها، تایمرها یا واسط‌های شبکه) معمولاً محدود به هسته هستند چون نیاز به تعامل مستقیم با سخت‌افزار دارند یا ممکنه باعث مختل شدن عملکردهای دیگر سیستم بشوند.
- **خطاها یا همون fault ها** (مثل `T_PGFLT` برای خطاهای صفحه) هم مختص هسته هستند چون مدیریت حافظه مجازی به‌طور ایمن فقط توسط هسته انجام می‌شود.

**نمی‌توان تمامی تله‌ها (traps) را با سطح دسترسی `DPL_USER` فعال کرد.** این محدودیت به دلیل مسائل امنیتی و پایداری سیستم است. در سیستم‌عامل‌ها، مد کاربر و مد هسته به‌طور جداگانه از هم عمل می‌کنند تا از خرابکاری و دسترسی غیرمجاز جلوگیری شود.

1. **جداسازی مد کاربر و هسته**: در مد کاربر، برنامه‌ها دسترسی محدودی دارند و نمی‌توانند به منابع حساس مثل سخت‌افزار یا مدیریت حافظه دسترسی پیدا کنند. در حالی که مد هسته دسترسی کامل به این منابع دارد.
2. **تله‌های خاص برای دسترسی محدود**: تله‌هایی مثل `T_SYSCALL` که برای سیستم کال‌ها (system calls) استفاده می‌شوند، تنها نقاط کنترل‌شده‌ای هستند که به برنامه‌های کاربر اجازه می‌دهند درخواست‌هایی از هسته داشته باشند. دیگر تله‌ها که برای پردازش وقایع مانند خطاها یا وقفه‌های سخت‌افزاری استفاده می‌شوند، فقط به هسته مربوط هستند.
3. **دلیل محدودیت دسترسی در `DPL_USER`**: اگر تمام تله‌ها برای مد کاربر فعال بودند، برنامه‌های کاربر می‌توانستند به طور مستقیم به سخت‌افزار دسترسی پیدا کنند یا هسته سیستم را مختل کنند. به همین دلیل فقط `T_SYSCALL` با `DPL_USER` فعال است تا به برنامه‌های کاربر اجازه دهد از طریق سیستم کال‌ها به هسته دسترسی پیدا کنند و درخواست‌های خود را انجام دهند.

## سوال ۴: در صورت تغییر سطح دسترسی، ss و esp روی پشته Push می‌شود. در غیراین‌صورت Push نمی‌شود. چرا؟

### 1. وقفه‌ها، تله‌ها و پشته

- وقتی یک وقفه یا تله رخ می‌دهد، پردازنده به‌طور خودکار اطلاعات مهمی را در پشته ذخیره می‌کند تا پس از پردازش وقفه، بتواند به حالت قبل بازگردد.
- این اطلاعات معمولاً شامل مقادیر رجیسترهای `cs` (Code Segment)، `eip` (Instruction Pointer)، `eflags` و در صورت لزوم کد خطا است. این رجیسترها همیشه روی پشته قرار می‌گیرند، صرف‌نظر از سطح دسترسی.

### 2. سطوح دسترسی و مدیریت پشته

- پردازنده x86 دارای چندین سطح دسترسی است که از سطح 0 (حالت هسته، بالاترین سطح دسترسی) تا سطح 3 (حالت کاربر، پایین‌ترین سطح دسترسی) متغیر است. این ساختار به سیستم کمک می‌کند تا از دسترسی غیرمجاز و خطرات امنیتی جلوگیری کند.
- هر سطح دسترسی معمولاً پشته خاص خود را دارد. برای مثال، هسته سیستم یک پشته خاص برای انجام وظایف سطح سیستمی دارد، در حالی که هر فرآیند در حالت کاربر دارای پشته‌ای جداگانه برای اجرای خود است.

### 3. چرا تنها زمانی که سطح دسترسی تغییر می‌کند، `ss` و `esp` ذخیره می‌شوند؟

- زمانی که یک تغییر سطح دسترسی رخ می‌دهد (مثلاً تغییر از حالت کاربر به حالت هسته)، پردازنده نیاز دارد تا به پشته هسته سوئیچ کند. این تغییر ضروری است زیرا هسته باید از پشته‌ای جداگانه نسبت به حالت کاربر استفاده کند تا از مشکلات امنیتی و احتمال خرابی داده‌ها جلوگیری شود.
- برای انجام این سوئیچ، پردازنده مقادیر فعلی `ss` و `esp` (که مربوط به پشته حالت کاربر هستند) را ذخیره می‌کند تا بعداً زمانی که به حالت کاربر بازمی‌گردد، این مقادیر را بازیابی کرده و به همان نقطه‌ای که وقفه در آنجا رخ داده بود، بازگردد.

بدون ذخیره مقادیر `ss` و `esp` در هنگام تغییر سطح دسترسی، پردازنده قادر نخواهد بود موقعیت اصلی پشته حالت کاربر را دنبال کند و برگشت به فرآیند کاربر به درستی انجام نخواهد شد.

### 4. چرا همیشه `ss` و `esp` ذخیره نمی‌شوند؟

- در صورتی که تغییری در سطح دسترسی ایجاد نشود (مثلاً در حالتی که یک وقفه در حالت هسته رخ دهد)، پردازنده از پشته هسته موجود استفاده می‌کند و نیازی به ذخیره‌سازی `ss` و `esp` نیست زیرا پشته همان است و عوض نشده است.
- ذخیره‌سازی مکرر `ss` و `esp` می‌تواند بار اضافی بر پشته وارد کند و فضای بیشتری را برای ذخیره‌سازی اشغال کند، بدون آنکه لزومی به آن باشد.

### 5. توضیح اضافی در مورد `ss` و `esp` در معماری x86

- **`ss` (Stack Segment)**: این رجیستر به **سگمنت پشته** اشاره می‌کند. `ss` اطلاعاتی در مورد سگمنت حافظه‌ای که به‌عنوان پشته استفاده می‌شود، نگه می‌دارد. در حالت محافظت شده x86، حافظه به سگمنت‌ها تقسیم می‌شود و selector `ss` مربوط به سگمنت پشته فعلی را نگه می‌دارد. در حالت کاربر، `ss` به سگمنت پشته کاربر اشاره دارد و در حالت هسته به سگمنت پشته هسته اشاره می‌کند.
- **`esp` (Extended Stack Pointer)**: این رجیستر **موقعیت فعلی بالای پشته** را نشان می‌دهد. `esp` در واقع اشاره‌گر به بالای پشته است و هنگامی که داده‌ای به پشته اضافه می‌شود، مقدار آن کاهش می‌یابد و وقتی داده‌ای از پشته برداشته می‌شود، افزایش می‌یابد. این رجیستر تعیین می‌کند که در کدام نقطه از پشته داده‌ها قرار خواهند گرفت.

رجیستر `ss:esp` آدرس کامل بالای پشته را نشان می‌دهد، به‌طوری‌که `ss` پایه سگمنت پشته را تعیین می‌کند و `esp` آفست (فاصله) را فراهم می‌کند.

## سوال ۵: 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 ()read_sys اجرای سیستم را با مشکل روبرو سازد.

### 1. توابعی که پارامترهای فراخوانی سیستم رو دریافت می‌کنند.

این توابع توسط هسته سیستم برای گرفتن اطلاعاتی استفاده می‌شه که از طرف برنامه‌های کاربری به فراخوانی‌های سیستم ارسال شده. وقتی یه برنامه کاربری یه فراخوانی سیستم رو انجام می‌ده، پارامترهایی که می‌ده توی حافظه کاربر قرار دارن. هسته با استفاده از این توابع (مثل `argint`، `()argptr` و ...)

این پارامترها رو از حافظه کاربر به‌طور امن دریافت می‌کنه و از این که به آدرس‌های درست و معتبر دسترسی پیدا کنه، مطمئن می‌شه.

### 2. چرا ()argptr آدرس‌ها رو چک می‌کنه؟

argptr() آدرس‌ها رو چک می‌کنه تا مطمئن بشه که آدرس‌هایی که از فضای کاربر ارسال می‌شن در بازه‌ای هستن که هسته می‌تونه بهشون دسترسی داشته باشه. این کار برای اینه که هسته از دسترسی به قسمت‌های غیرمجاز یا خطرناک حافظه جلوگیری کنه. اگه آدرس‌ها خارج از بازه‌ای باشن که کاربر می‌تونه بهش دسترسی داشته باشه، هسته از دسترسی به این آدرس‌ها جلوگیری می‌کنه تا از کرش سیستم یا مشکلات دیگه جلوگیری بشه.

### 3. چرا اگر آدرس‌ها از محدوده مجاز فراتر برن، مشکل امنیتی پیش میاد؟

اگه آدرسی که کاربر می‌ده از محدوده مجاز بیرون بره (مثلاً به جای حافظه کاربر به حافظه هسته اشاره کنه)، ممکنه یه مهاجم از این ضعف استفاده کنه و بتونه به داده‌های حساس هسته دسترسی پیدا کنه یا اونا رو تغییر بده. این می‌تونه منجر به آسیب‌پذیری‌های امنیتی بشه که ممکنه به کنترل غیرمجاز سیستم یا خرابی داده‌ها ختم بشه.

### 4. یه مثال از اینکه چرا read_sys ممکنه مشکل‌ساز بشه اگه آدرس‌ها چک نشه.

فرض کن یه کاربر یه اشاره‌گر اشتباه به ()read_sys می‌ده که به یه بخش حساس از حافظه هسته اشاره می‌کنه. اگه ()argptr چک نکنه که این آدرس در محدوده‌ی مجاز قرار داره یا نه، ممکنه ()read_sys داده‌ها رو توی حافظه هسته بنویسه به جای حافظه کاربر. این می‌تونه باعث بشه که داده‌های مهم هسته خراب بشه و سیستم دچار مشکل بشه.

## بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb:

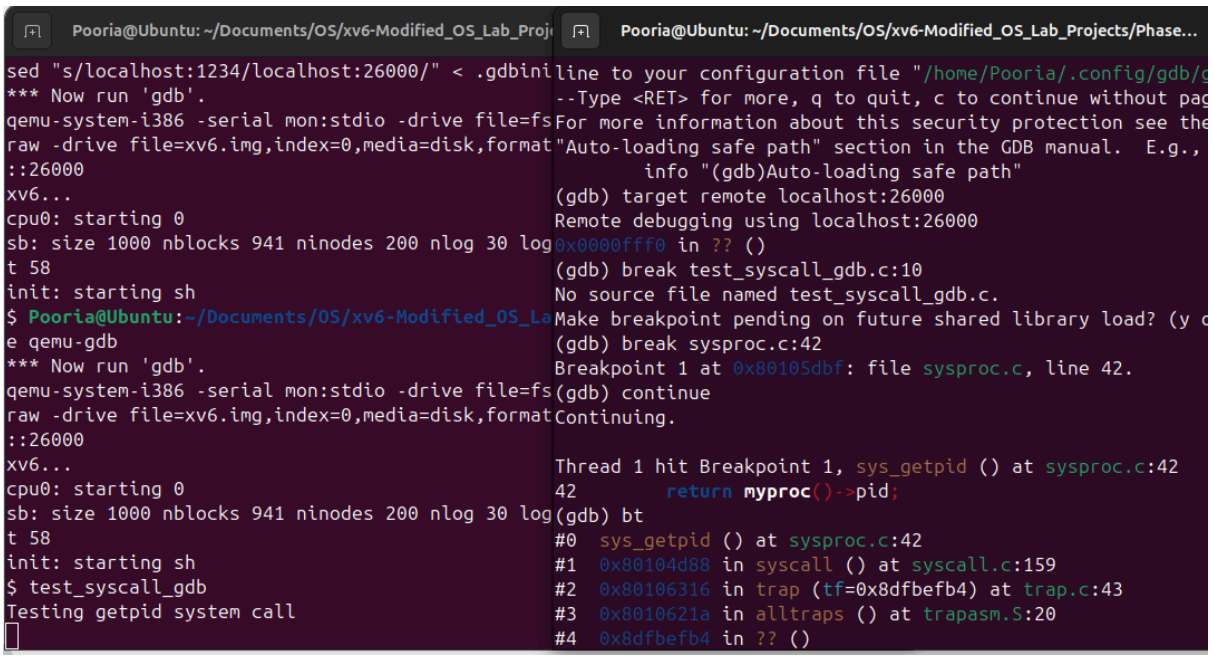
برای بررسی روند فراخوانی های سیستمی برنامه کوچک زیر نوشته و استفاده می‌شود که در آن به غیر فراخوانی هایی که برای نوشتن پیام در ترمینال صورت میگیرد تنها یک فراخوانی ()getpid صورت گرفته و بعد از آن نیز Id مورد نظر در ترمینال چاپ میشود.

```c
// test_syscall_gdb.c

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

int main(void) {
    // Make some system calls
    printf(1, "Testing getpid system call\n");
    int pid = getpid();
    printf(1,"Process ID: %d\n",pid);
    exit();
}
```

در ادامه سیستم عامل را با gdb اجرا می‌کنیم و بر روی ابتدا تابع syscall که در ابتدای تمامی فراخوانی های سیستمی اجرا می‌شود یک breakpoint قرار می‌دهیم. پس از اجرای برنامه سطح کاربر اضافه شده با دستور test_syscall_gdb به breakpoint مورد نظر میرسیم.اما اگر مقدار eax را بررسی کنیم با مقدار pid برابر نیست.علت آن است که همچنان به فراخوانی سیستمی مورد نظرمان نرسیده‌ایم و با چند بار تکرار دستور c یا همان continue به نقطه مورد نظرمان می‌رسیم.دستور bt درgdb یا همان (backtrace) توالی فراخوانی های تابع را نشان می دهد که به نقطه فعلی در نقطه break برنامه منتهی می شود که برای اشکال زدایی برای درک جریان اجرا و شناسایی مکان هایی که ممکن است کارها اشتباه انجام شود مفید است.خروجی این دستور در تصویر زیر آمده است که روند اجرای برنامه های مختلف تا رسیدن به نقطه فعلی را نشان می‌دهد:

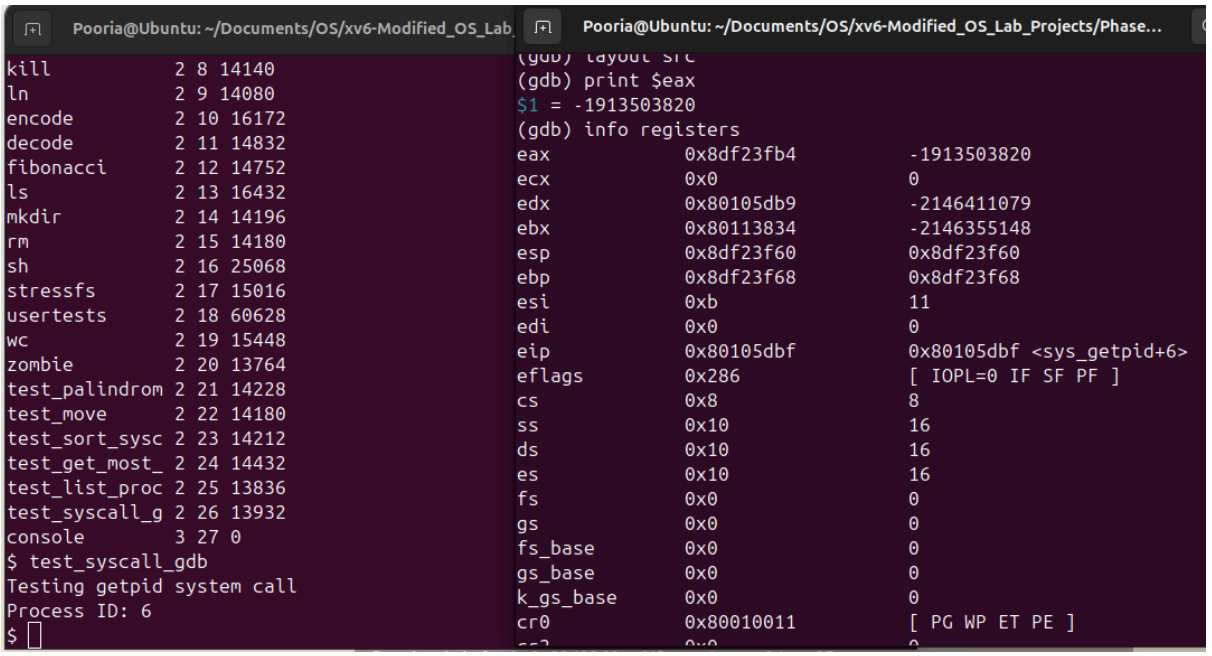



در این حالت با استفاده از دستور down میتوانیم در پشته تماس به اندازه یک فریم به پایینتر منتقل شویم.با استفاده از کلید های میان بر Ctrl+A + Ctrl+X و دستور layout split در پنل باز شده میتوانیم همزمان کد های اسمبلی و دستور متناظر با آن در فایل هایمان را مشاهده کنیم.با استفاده از دو دستور si (پرش به اندازه یک دستور ماشین) و s (پرش به اندازه یک دستور در کد مورد نظر) آنقدری جلو میرویم تا به دستور بعد از دریافت pid برسیم.در این حالت اگر محتوای رجیستر eax را مشاهده کنیم همان مقدار pid برایمان نمایش داده می‌شود.

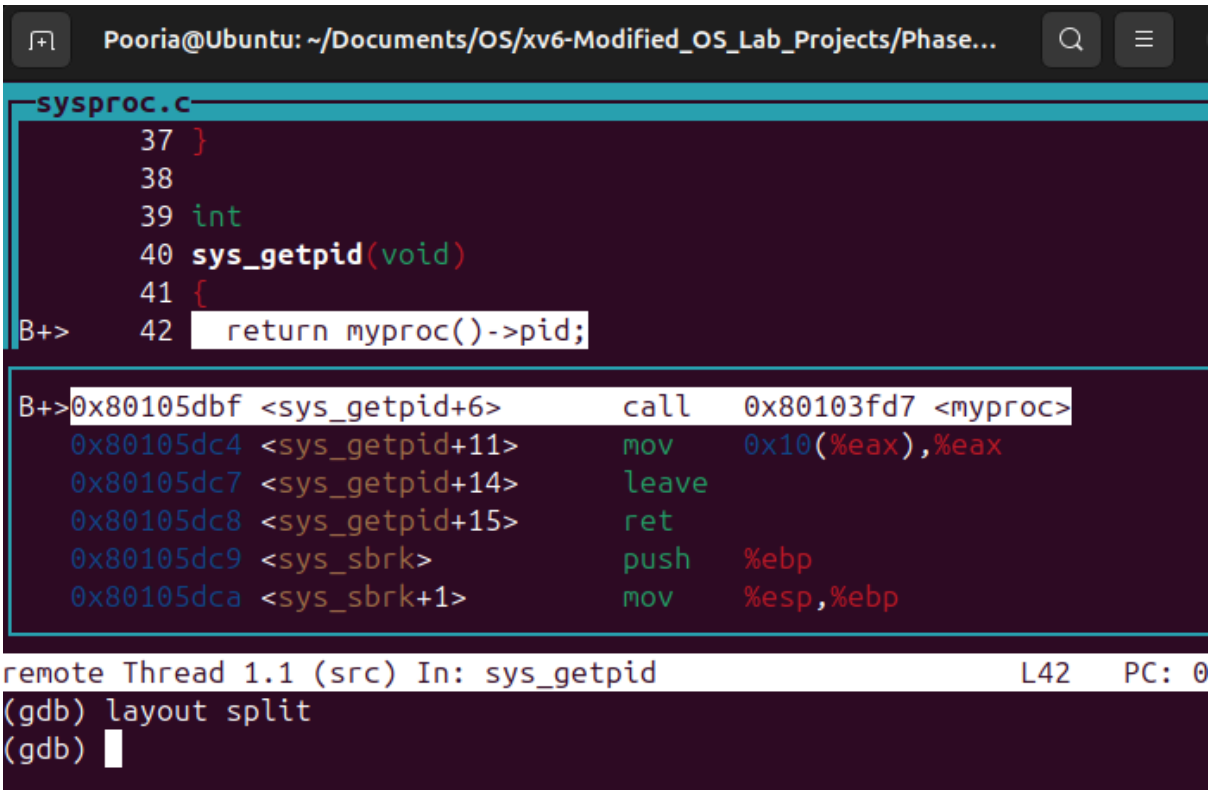
Breakpointبر روی خط159 کد زیر قرار داده شده است :

```
150  // syscall.c
151  int syscall_counts[MAX_SYSCALLS] = {0};  // Define the array with the new name
152
153  void syscall(void) {
154      int num;
155      struct proc *curproc = myproc();
156
157      num = curproc->tf->eax;
158      if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
159          curproc->tf->eax = syscalls[num]();
160          curproc->syscalls[num]++;  // Increment the count of the specific system call for the process
161          curproc->syscall_count++;  // Increment the total number of system calls for the process
162          syscall_counts[num]++;  //Increment the total number of system call usage count
163      } else {
164          // Handle invalid syscall number
165          curproc->tf->eax = -1;
166      }
167  }
```

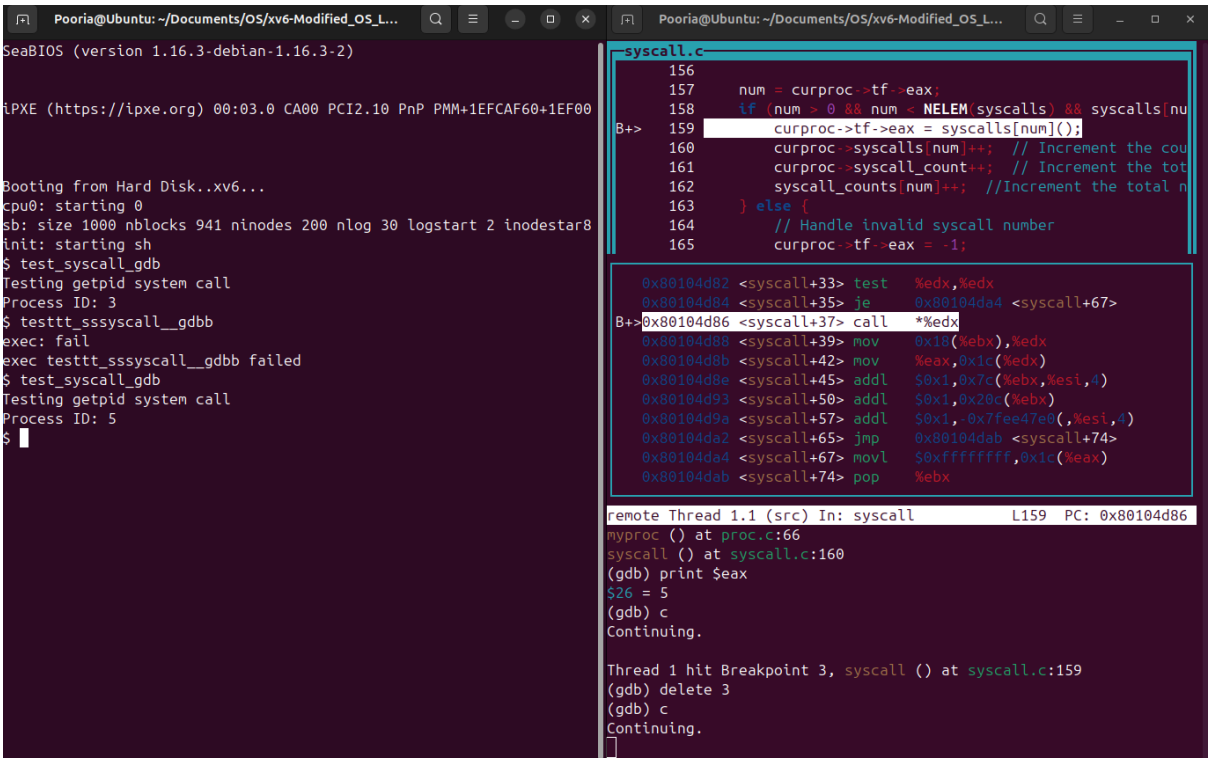
پس از اولین برخورد به breakpoint مقدار eax به صورت زیر است.




پنل کاربری دوگانه به صورت زیر است:



در تصویر بعدی نشان داده شده که همانطور که در انتهای برنامه سطح کاربر pid برابر با 5 چاپ شده در gdb نیز مقدار رجیستر eax برابر 5 است.



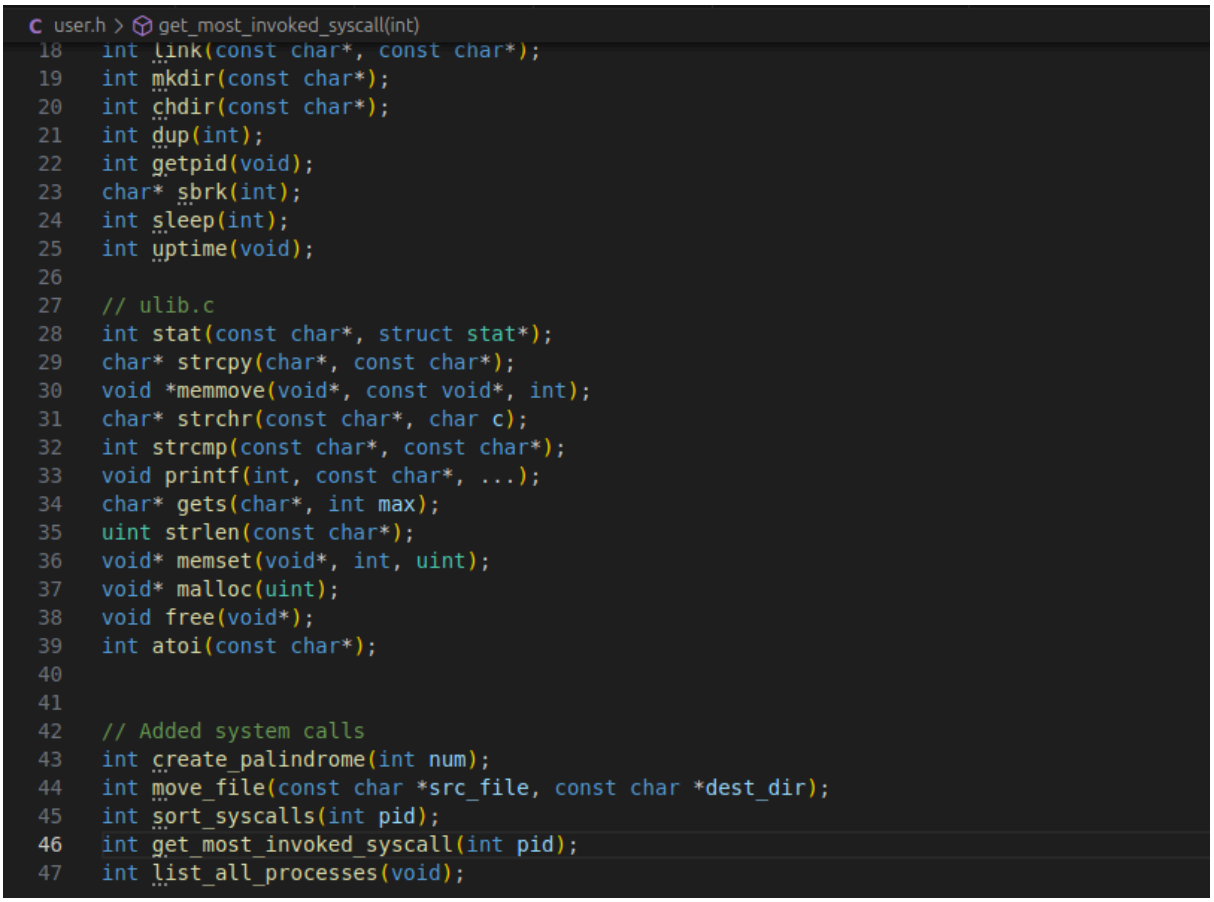
## اضافه کردن فراخوانی سیستمی جدید:

برای اضافه کردن فراخوانی سیستمی جدید باید مراحل زیر را انجام دهیم:

1)ابتدا در فایل "syscall.h" ؛ id مربوط به فراخوانی جدید را تعریف می‌کنیم.

```c
// System call numbers
#define SYS_fork    1
#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_create_palindrome 22
#define SYS_move_file 23
#define SYS_sort_syscalls 24
#define SYS_get_most_invoked_syscall 25
#define SYS_list_all_processes 26
```

2)در فایل "user.h" فراخوانی جدید را همراه با نوع آرگومان های ورودی و خروجی فراخوانی declare میکنیم تا برنامه های سطح کاربر بتوانند فراخوانی انجام بدهند.



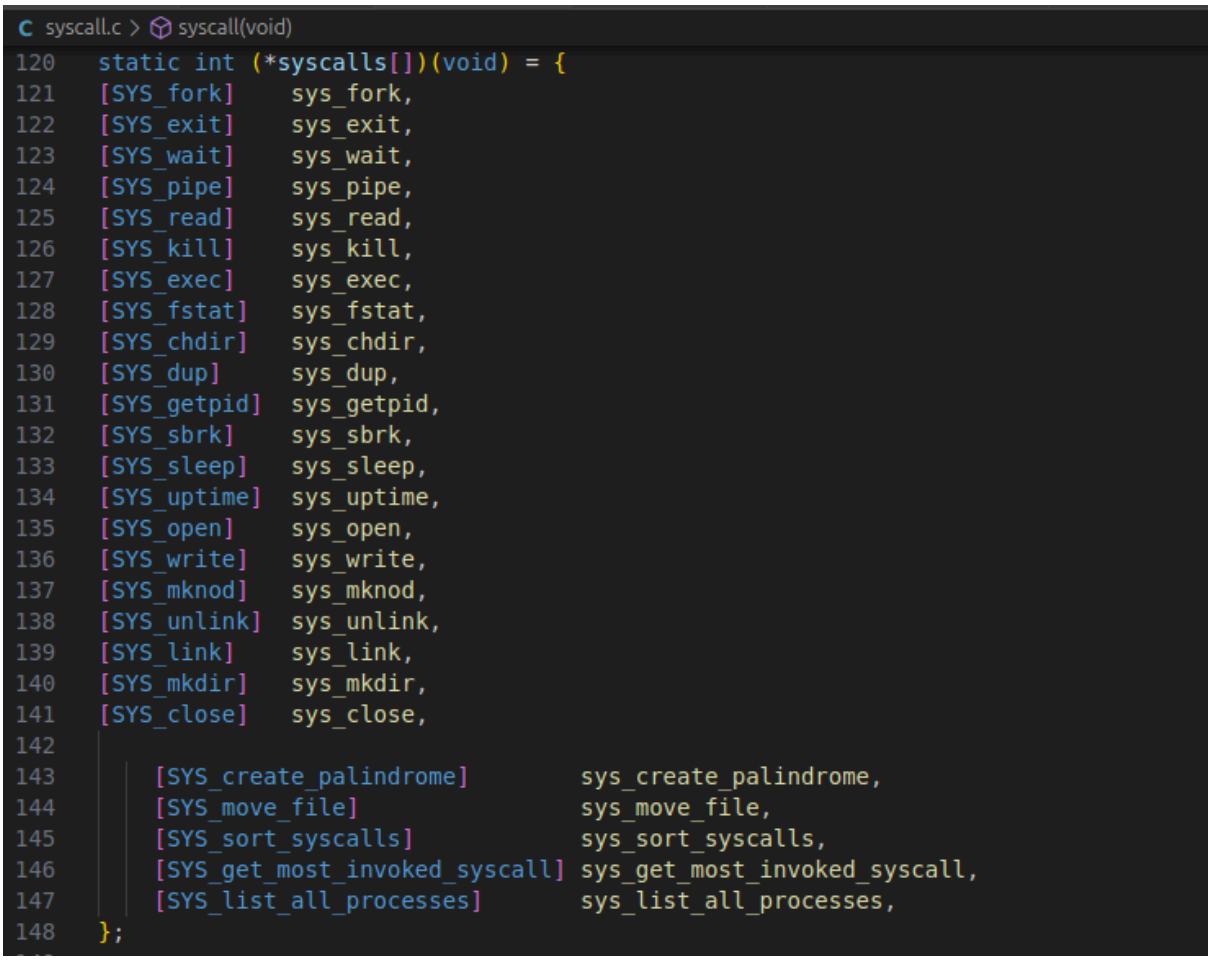
3)پیاده سازی فراخوانی سیستمی در کرنل:تابعی که هنگام فراخوانی سیستمی صدا زده میشود را در فایل "sysproc.c" تعریف میکنیم(definition تابع).در تصویر زیر پیاده سازی فراخوانی create_palindrome آمده است.درا ین تابع ابتدا عدد مورد نظر کاربر توسط argint دریافت و در num ذخیره میشود.argint همچنین اینکه ورودی به تابع داده شده یا نه را بررسی میکند و در صورت عدم وارد شدن ورودی از تابع خارج شده و -1 را برمیگرداند.در صورتی که عدد ورودی صفر باشد خود صفر به عنوان خروجی برگردانده میشود.علت جدا کردن آن به عنوان یک حالت خاص آن است که برای به دست آوردن عبارت برعکس شده همواره ورودی را بر 10 تقسیم میکنیم و باقی مانده آن را درنظر میگیریم که درصورت صفر بودن تابع درست عمل نخواهد کرد.

درحلقه while بعدی تا رسیدن به صفر متغیر temp که در ابتدا برابر عددمان است را بر 10 تقسیم کرده و باقی مانده آن را به عنوان رقم عدد برعکس شده (reversed) ذخیره میکنیم و این کار را تا صفر شدن temp ادامه میدهیم.در این مرحله عدد و برعکس آن را پشت یکدیگر چاپ میکنیم.علت این کار این است که عدد palindrome محاسبه شده میتواند مقادیر بسیار زیادی داشته باشد و overflow رخ بدهد.اما با استفاده از این روش اعداد با تعداد ارقام خیلی بیشتر را نیز میتوان به صورت palindrome شده نوشت.

sysproc.c > sys_create_palindrome(void)

```
93  int
94  sys_create_palindrome(void)
95  {
96      int num;
97      if (argint(0, &num) < 0)
98          return -1;
99      if(num == 0){
100       cprintf("Palindrome: %d\n",num);
101       return 0;
102     }
103     int reversed = 0, temp = num;
104     while (temp != 0) {
105         reversed = reversed * 10 + temp % 10;
106         temp /= 10;
107     }
108     cprintf("Palindrome: %d%d\n", num,reversed);
109
110     return 0;
111 }
```

4)فایل "syscall.c" را تغییر داده و فراخوانی جدید را به لیست فراخوانی های موجود در system call table اضافه میکنیم.



5)در "syscall.c" تابع مربوط به فراخوانی سیستمی را declare میکنیم.

```c
// syscall.c > syscall(void)
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"
// int syscalls[MAX_SYSCALLS] = {0};  // Initialize with default values if needed


// syscall.c

// Existing includes and definitions...

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);
```

6)فایل "usys.s" را تغییر میدهیم تا یک رابط برای برنامه های سطح کاربر برای صدا زدن فراخوانی جدید ایجاد کنیم.

```asm
usys.S
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)


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

7)در این مرحله یک برنامه سطح کاربر برای تست کردن عملکرد فراخوانی سیستمی می‌نویسیم و در آن فراخوانی مورد نظرمان را صدا میزنیم.در تصویر زیر برنامه create_palindrome آمده است.

در کد داده شده ابتدا بررسی میشود که ورودی به تابع داده شده و آیا به تعداد مناسب ورودی داده شده است یا خیر.به این علت که تابع atoi خروجی صفر را برای رشته های غیر عددی میدهد برای تمایز میان ورودی عدد 0 و رشته حرفی بررسی میکنیم که اولین کاراکتر ورودی چیست.برای رشته حروف پیغامی در ترمینال چاپ میشود که نحوه استفاده از برنامه را توضیح میدهد و برای ورودی های معتبر تابع مربوط به فراخوانی سیستمی صدا زده میشود.

test_palindrome.c > ...

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

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

    int num = atoi(argv[1]);
    if(num == 0 && argv[1][0] != '0'){
        printf(2, "Usage: test_palindrome <number>\n");
        exit();
    }
    create_palindrome(num);

    exit();
}
```

8) در آخر "Makefile" را ویرایش میکنیم:نام برنامه سطح کاربر اضافه شده را در بخش UPROGS اضافه میکنیم تا بتوانیم در ترمینال آن را صدا زده و استفاده کنیم.