گزارش پروژه (1) آزمایشگاه سیستم عامل

نام استاد: دکتر کارگهی

امیرارسلان شهبازی 810101451

محمدحسین مظهری 810101520

محمدمهدی صمدی 810101465

گروه 11

لینک ریپوزیتوری گیتهاب: https://github.com/AMIRSH1383/OS-SMS-Lab2.git

1) کتابخانه های سطح کاربر در 6xv، برای ایجاد ارتباط میان برنامههای کاربر و کرنل به کار میروند. این کتابخانهها شامل توابعی هستند که از فراخوانیهای سیستمی استفاده میکنند تا دسترسی به منابع سختافزاری و نرمافزاری سیستمعامل ممکن شود. با تحلیل فایلهای موجود در متغیر ULIB در 6xv، توضیح دهید که چگونه این کتابخانهها از فراخوانیهای سیستمی بهره میبرند؟ همچنین، دالیل استفاده از این فراخوانیها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه ها را شرح دهید.

این کتابخانه ها به عنوان واسطی میان کاربر و کرنل رفتار میکنند. کتابخانه‌ها شامل توابعی هستند که از فراخوانی‌های سیستمی استفاده می‌کنند تا دسترسی به منابع سخت‌افزاری و نرم‌افزاری سیستم‌عامل ممکن شود.

**نحوه استفاده از فراخوانی‌های سیستمی**

فراخوانی‌های سیستمی در xv6 به صورت تابع‌هایی در کتابخانه‌های سطح کاربر پیاده‌سازی شده‌اند. این فراخوانی‌ها توسط برنامه‌های کاربر فراخوانی می‌شوند و سپس به کرنل انتقال می‌یابند تا عملکرد‌های سطح پایین سیستم را اجرا کنند. در فایل‌های موجود در متغیر ULIB، توابعی مانند read، write، fork، exec و غیره تعریف شده‌اند که هر کدام به یک فراخوانی سیستمی معین متصل هستند.

**دلایل استفاده از فراخوانی‌های سیستمی**

1. **امنیت**:

فراخوانی‌های سیستمی به کرنل اجازه می‌دهند که دسترسی به منابع حساس سیستم مانند حافظه، فایل‌ها، و دستگاه‌ها را کنترل کند. این امر مانع از دسترسی غیرمجاز و سوءاستفاده از منابع می‌شود.

1. **مدیریت منابع**:

فراخوانی‌های سیستمی امکان مدیریت موثر منابع سیستم را فراهم می‌کنند. به عنوان مثال، کرنل می‌تواند تعداد فرآیندهای همزمان را کنترل کرده و حافظه را بهینه تخصیص دهد.

1. **قابلیت حمل**:

با استفاده از فراخوانی‌های سیستمی، برنامه‌ها می‌توانند به شکلی مستقل از جزئیات سخت‌افزاری و نرم‌افزاری خاص اجرا شوند. این امر باعث می‌شود که برنامه‌ها بتوانند به راحتی به سیستم‌عامل‌ها و سخت‌افزارهای مختلف منتقل شوند.

**تأثیر فراخوانی‌های سیستمی بر عملکرد و قابلیت حمل**

1. **عملکرد**:
   * استفاده از فراخوانی‌های سیستمی ممکن است مقداری سربار ایجاد کند، زیرا تغییر حالت بین کرنل و کاربر نیاز به زمان دارد. با این حال، استفاده بهینه از این فراخوانی‌ها می‌تواند به بهبود عملکرد کلی سیستم کمک کند.
2. **قابلیت حمل**:
   * فراخوانی‌های سیستمی یک واسط یکنواخت برای دسترسی به عملکردهای پایه سیستم‌عامل فراهم می‌کنند. این امر به برنامه‌ها اجازه می‌دهد که بدون نیاز به تغییرات بزرگ در کد، به راحتی بر روی سیستم‌عامل‌های مختلف اجرا شوند.

2) فراخوانی های 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.

علاوه بر فراخوانی‌های سیستمی، روش‌های دیگری نیز وجود دارند که برنامه‌های کاربر می‌توانند از طریق آن‌ها به کرنل دسترسی داشته باشند. در ادامه، برخی از این روش‌ها را به اختصار توضیح می‌دهم:

**1. مکانیزم /proc و /sys**

* **/proc**: یک سیستم فایل مجازی است که اطلاعات مربوط به فرآیندها و سیستم‌عامل را فراهم می‌کند. برنامه‌ها می‌توانند با خواندن یا نوشتن در فایل‌های موجود در /proc، اطلاعات مورد نیاز را بدست آورند یا تنظیمات سیستم را تغییر دهند.
  + مثال: خواندن اطلاعات فرآیندها از /proc/pid/status.
* **/sys**: یک سیستم فایل مجازی دیگر است که دسترسی به اطلاعات سخت‌افزاری و تنظیمات دستگاه‌ها را فراهم می‌کند.
  + مثال: دسترسی به تنظیمات دستگاه‌های USB از طریق /sys/bus/usb.

**2. نرم‌افزارهای کاربر-کرنل (Kernel-Userland Libraries)**

* برخی کتابخانه‌های سطح کاربر به طور مستقیم با کرنل ارتباط برقرار می‌کنند تا عملکردهای خاص را فراهم کنند. به عنوان مثال، libc از فراخوانی‌های سیستمی استفاده می‌کند تا توابع سطح بالا مثل malloc و free را پیاده‌سازی کند.
  + مثال: کتابخانه libc برای مدیریت حافظه.

**3.IOCTL**

* مکانیزم ioctl (Input/Output Control) به برنامه‌های کاربر اجازه می‌دهد تا دستورات خاصی را به دستگاه‌های سخت‌افزاری ارسال کنند. این مکانیزم معمولاً برای دستگاه‌های خاص مثل ترمینال‌ها و دستگاه‌های شبکه استفاده می‌شود.
  + مثال: تنظیم سرعت بادگیر ترمینال با ioctl درایور ترمینال.

**4. Netlink Sockets**

* سوکت‌های Netlink واسطی بین کرنل و فضای کاربر فراهم می‌کنند که برای ارتباطات پیچیده‌تر مثل پیکربندی شبکه استفاده می‌شوند.
  + مثال: استفاده از Netlink برای مدیریت تنظیمات شبکه از طریق iproute2.

**5. Memory-Mapped I/O (mmap)**

* مکانیزم mmap به برنامه‌های کاربر اجازه می‌دهد تا بخشی از حافظه را مستقیماً به فضای آدرس خود نگاشت کنند. این روش معمولاً برای دسترسی به حافظه مشترک یا دستگاه‌های سخت‌افزاری استفاده می‌شود.
  + مثال: نگاشت فایل به حافظه برای دسترسی سریع‌تر به داده‌ها.

**6. Signaling**

* سیگنال‌ها مکانیزمی هستند که به کرنل اجازه می‌دهند تا رویدادهای خاصی را به برنامه‌های کاربر اطلاع دهند. برنامه‌ها می‌توانند سیگنال‌ها را پردازش کرده و اقدامات مناسب را انجام دهند.
  + مثال: دریافت سیگنال SIGINT برای خاتمه دادن به برنامه.

**7. پروتکل‌های خاص**

* برخی پروتکل‌های خاص مانند D-Bus به برنامه‌های کاربر اجازه می‌دهند تا به سرویس‌های کرنل دسترسی پیدا کنند و با آن‌ها ارتباط برقرار کنند.
  + مثال: استفاده از D-Bus برای ارتباط با مدیر جلسه کاربر (session manager).

هر یک از این روش‌ها دسترسی به عملکردهای خاصی از کرنل را فراهم می‌کنند و بسته به نیاز برنامه، می‌توانند به طور موثری مورد استفاده قرار گیرند.

پرسش 3 : آیا باقی تله ها را نمیتوان با سطح دسترسی DPL\_USER فعال نمود ؟ چرا ؟

خیر، این امر امکان پذیر نیست . همان طور که میدانیم در xv6 دو سطح kernel , user موجود هستند . چون سطح دسترسی DPL\_USER یک سطح دسترسی کاربر(سطح 3 ) است ، نباید امکان دسترسی به هسته و اجرای این تله ها را داشته باشد . اگر کاربر امکان اجرای این تله ها را داشته باشد ، به سادگی میتواند به kernel دسترسی داشته باشد . در این صورت protection نقض می شود .

در واقع اگر پردازنده بخواهد یک interrupt دیگر را فعال کند ، kernel اجازه ی این عمل را به او نمیدهد زیرا ممکن است در برنامه سطح کاربر مشکلی وجود داشته باشد و این مشکل به هسته منتقل شود(گسترش یابد و به هسته آسیب بزند) (protection ) یا خود کاربر قصد حمله به هسته را داشته باشد (security) که در صورت داشتن این دسترسی ها میتواند به هسته ی سیستم عامل آسیب بزند و هر بخشی از سخت افزار و نرم افزار را مورد تهاجم قرار دهد .

پرسش 4 : در صورت تغییر سطح دسترسی ، ss و esp روی پشته push میشود . در غیر اینصورت push نمیشود. چرا؟

دو پشته ی user stack و kernel stack موجود هستند . موقع فعال شدن یک trap ، میخوخاهیم دسترسی را تغییر دهیم (مثلا از سطح کاربر به سطح هسته برویم ) ، در این زمان نمیتوانیم از پشته قبلی استفاده کنیم . پس باید ss و esp روی پشته push شوند تا هنگام بازگشت آخرین دستوری که اجرا شده را بدانیم و انجام دستورات را از آنجا به بعد از سر بگیریم .

اما هنگامی که تغییری در سطح دسترسی رخ ندهد ، نیازی به push کردن آنها نیست زیرا همچنان با همان پشته ی قبلی مشغول به کار هستیم .

سوال 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.

این بخش سه تابع دارد.

argint : برای آن شماره آرگومان مشخص می شود و آدرس یک int را برای گرفتن متغیر به آن داده می شود.

argstr : دو پارامتر می گیرد. اولی شماره آرگومان است و دومی آدرس یک متغیر از نوع char \*است که در آن آرگومان ریخته می شود.

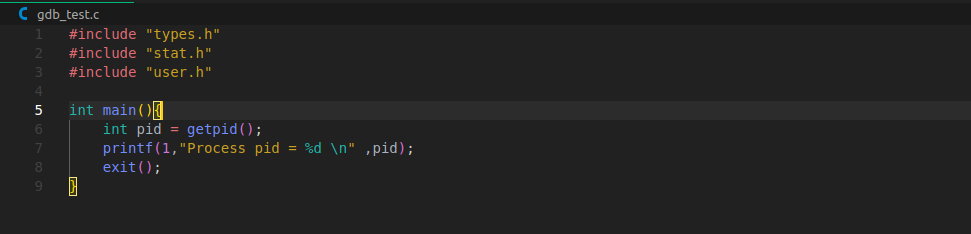
argptr : شماره آرگومان، آدرس یک پوینتر و اندازه چیزی که خوانده می شود را می گیرد و در آدرس پوینتر، محتوای آرگومان ریخته می شود.

اگر بازه آدرس ها بررسی نگردد ممکن است تجاوز از بازه ی معتبر پیش بیاید. خارج شدن از بازه معتبر می تواند مشکلاتی را برای ما به وجود بیاورد. مثلا ممکن است بخواهیم چیزی را با sys\_read بخوانیم اما اگر آدرسی که می خواهیم از آن بخوانیم از بازه معتبر خارج باشد و ما اعتبار بازه را چک نکرده باشیم، اطلاعات اشتباه را ممکن است بخوانیم و روند برنامه را بهم بریزیم.

**بررسی گام های اجرای فراخوانی سیستم در سطح کرنل توسط gdb :**

ابتدا برنامه ی سطح کاربر را مینویسیم و آن را به makefile اضافه می کنیم .

**gdb\_test.c**

****

سپس دستور make qemu-gdb را داخل دایرکتوری ای که OS در آن قرار دارد در ترمینال وارد می کنیم

در ترمینالی دیگر دستور kernel gdb را میزنیم .

سپس عبارت target remote tcp::26000 را مینویسیم .

سپس c را مینویسیم ( continue )

سپس cntrl c را میفشاریم .

پس از آن break syscall را می نویسیم .( break point قرار داده می شود )

سپس c را مینویسیم .

متوجه می شویم که برنامه در syscall متوقف می شود .

بعد bt را می نویسیم .

دستور bt یا همان backtrace ، stack call برنامه در این لحظه را نشان می دهد . هر تابع که فراخوانی می شود یک frame stack مخصوص به خودش به آن اختصاص داده می شود که متغیر های محلی و آدرس بازگشت و .. در آن نگهداری می شود .

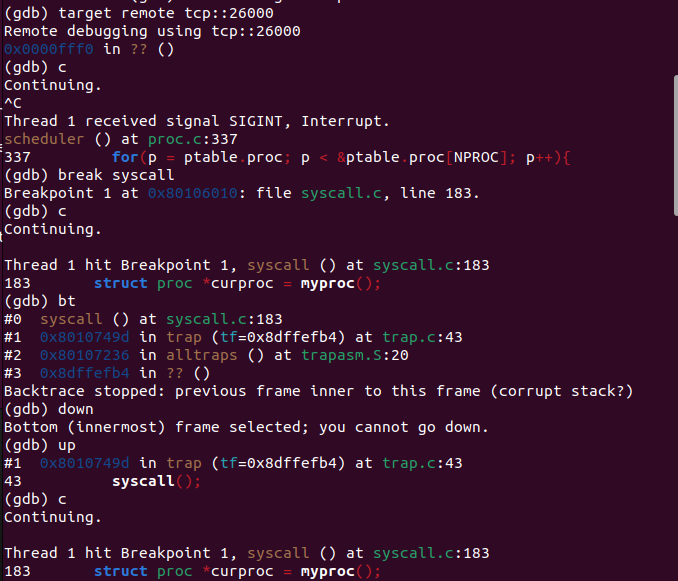
خروجی bt در هر خط یک frame stack را نشان می دهد که به ترتیب از درونی ترین frame هر یک را بیان می کنیم:

1. alltraps : در تابع در ابتدا trapframe مربوط به این trap ایجاد شده و در استک قرار میگیرد . سپس تابع trap() فراخوانی میشود . ( این تابع در trap.c است .)
2. trap : در این تابع ابتدا بررسی میشود که trap number متناظر با چه وقفه ای است . پس از آنکه معلوم شد که از نوع system call است ، trapframe مربوط به پردازه ی فعلی را برابر با trapframe قرار داده شده در استک می کند و تابع syscall() را صدا میزند .
3. syscall : این تابع eax را از trapframe پردازه فعلی میخواند (این مقدار برابر است با system call مورد نظر ) .

سپس با استفاده از syscalls[num] تابع مربوط به آن سیستم کال را فرا میخواند و خروجی آن را در eax در trapframe پردازه فعلی ذخیره می کند . (آرایه syscalls در ابتدای فایل syscalls.c قرار دارد که شماره هر سیستم کال را به تابع متناظر آن مپ می کند . )

بعد down را مینویسیم . چون در درونی ترین لایه هستیم ، با زدن این دستور به error زیر دچار میشویم (down ما را به سیستم کال قبلی ای که مارا فراخوانی کرده است میبرد ولی چون اینجا در درونی ترین لایه هستیم ، به ارور می خوریم . )

بعد up را مینویسیم .

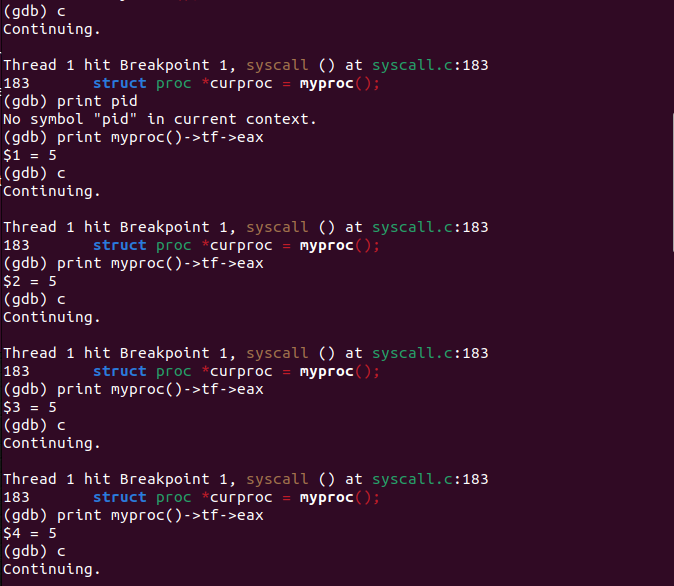


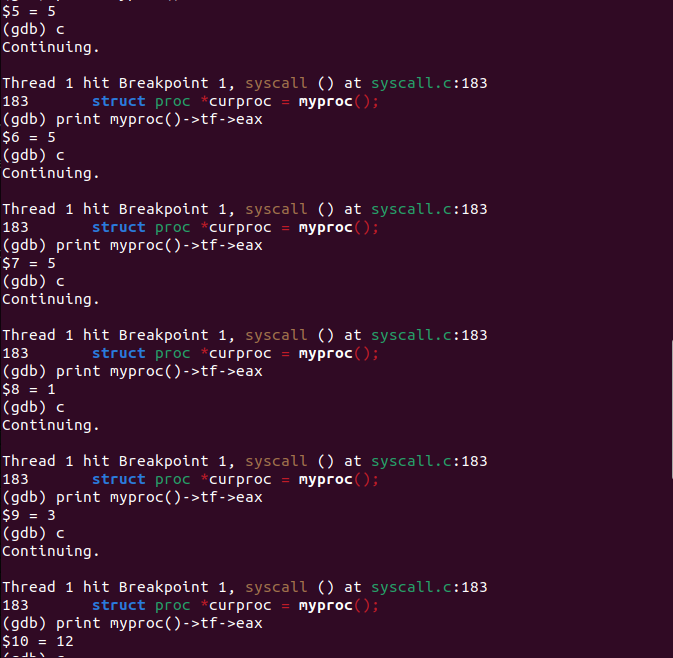
سپس چند باری c و سپس print myproc()->tf->eax را مینویسیم . (چاپ محتوای eax که در tf است )

تا جایی این کار را ادامه میدهیم که محتوای رجیستر eax برابر با شماره سیستم کال getpid() شود .

همانطور که مشاهده می شود مقدار رجیستر با شماره سیستم کال getpid() مطابقت ندارد .

زیرا وقتی برنامه سطح کاربر را اجرا میکنیم ، ابتدا چند عملیات read کردن (که شماره آن 5 است ) انجام میشود (7 مرتبه این اتفاق رخ میدهد . )

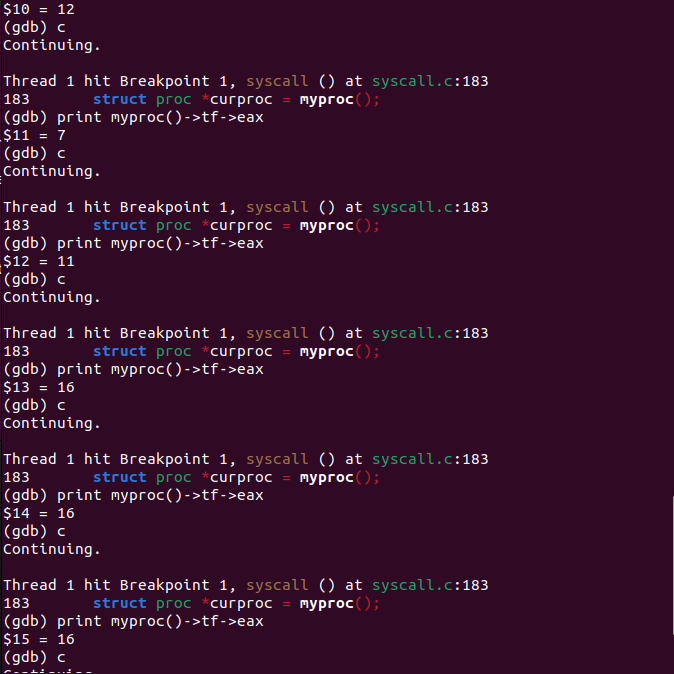




عدد 1 مربوط به سیستم کال fork است که جهت ایجاد پردازه جدید برای برنامه سطح کاربر صدا زده میشود.

عدد 3 مربوط به سیستم کال wait است که توسط پردازه ی پدر صدا زده میشود که تا پایان کار پردازه فرزند صبر میکند .

عدد 12 مربوط به سیستم کال sbrk است که این سیستم کال به پردازه ایجاد شده ، حافظه اختصاص میدهد.

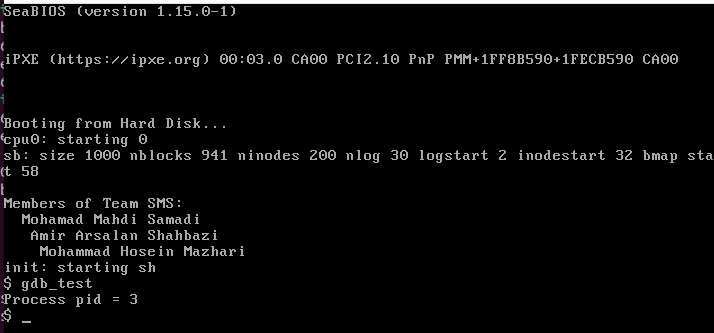


عدد 7 مربوط به سیستم کال exec است که برای اجرای برنامه pid در پردازه ایجاد شده استفاده میشود .

عدد 11 مربوط به سیستم کال getpid() است که انتظار آن را داشتیم .

عدد 16 مربوط به سیستم کال write است که در نهایت خروجی مورد نظر را برای کاربر مینویسد .

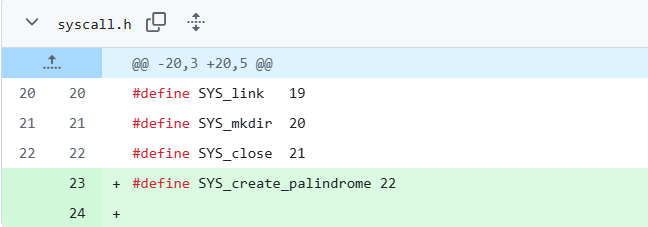
**نتیجه**

****

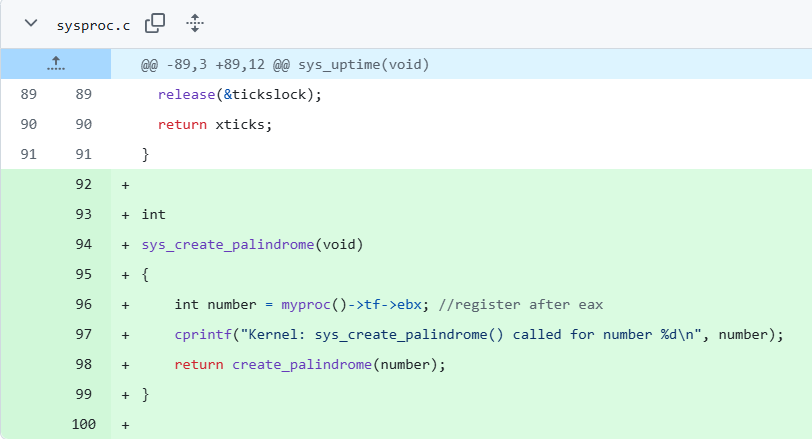
**ارسال آرگومان‌های فراخوانی سیستمی**

فراخوانی سیستمی create palindrome

در این دستور ما می بایست یک ورودی از کاربر گرفته و پالیندروم شده آن را تولید و به کاربر نشان دهیم.

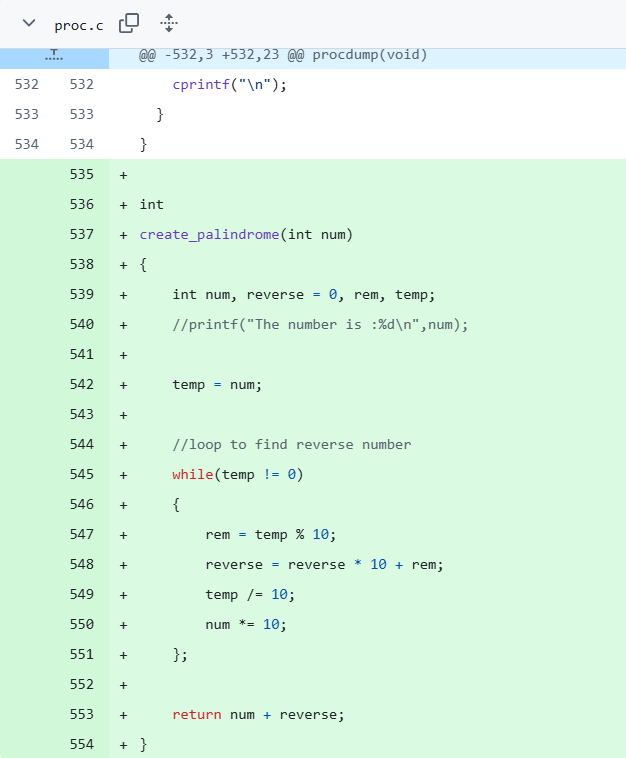


ابتدا در فایل بالا یک سیستم کال جدید به نام پالیندروم و شماره 22 ایجاد می کنیم.

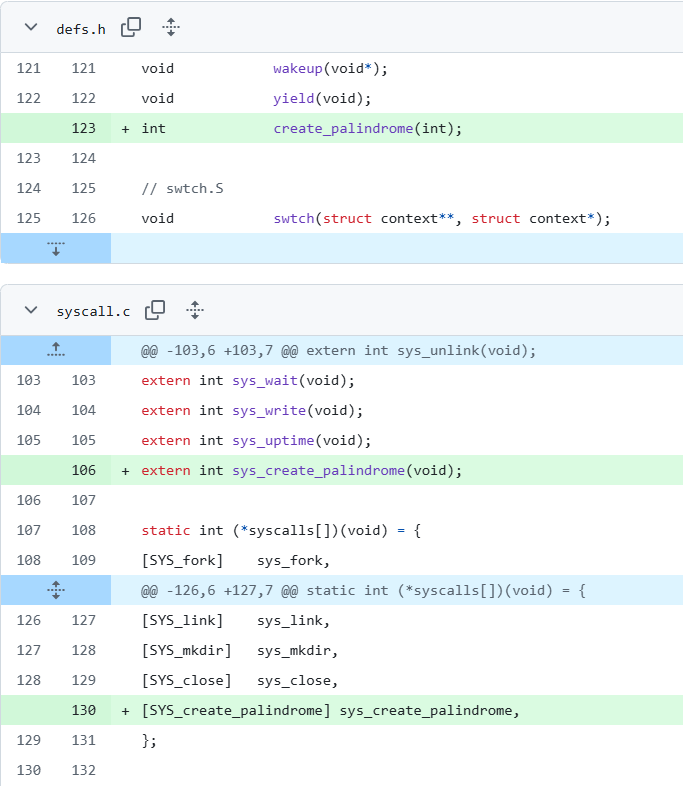


سپس در فایل sysproc.c یک تابع جدید برای سیستم کال ساخت پالیندروم تشکیل می دهیم. در این تابع با توجه به رفرنس و تابعی که برای سطح کاربر می زنیم، برای گرفتن آرگومان تابع می بایست رجیستر ebx را هدف قرار دهیم.

سپس می گوییم که کرنل این تابع را فراخوانی کرده است و در انتها هم تابع پالیندرومی که بعدا خواهیم نوشت و مسئول محاسبات خواهد بود را صدا می کنیم.



در فایل proc.c تابع محاسبه پالیندروم را می نویسیم. روال اینطور است که عددها را هر سری به توانهای 10 تقسیم می کنیم تا هر رقم در بیاید و در نهایت به صورت برعکس شده به رشته عددی خود اضافه می کنیم.



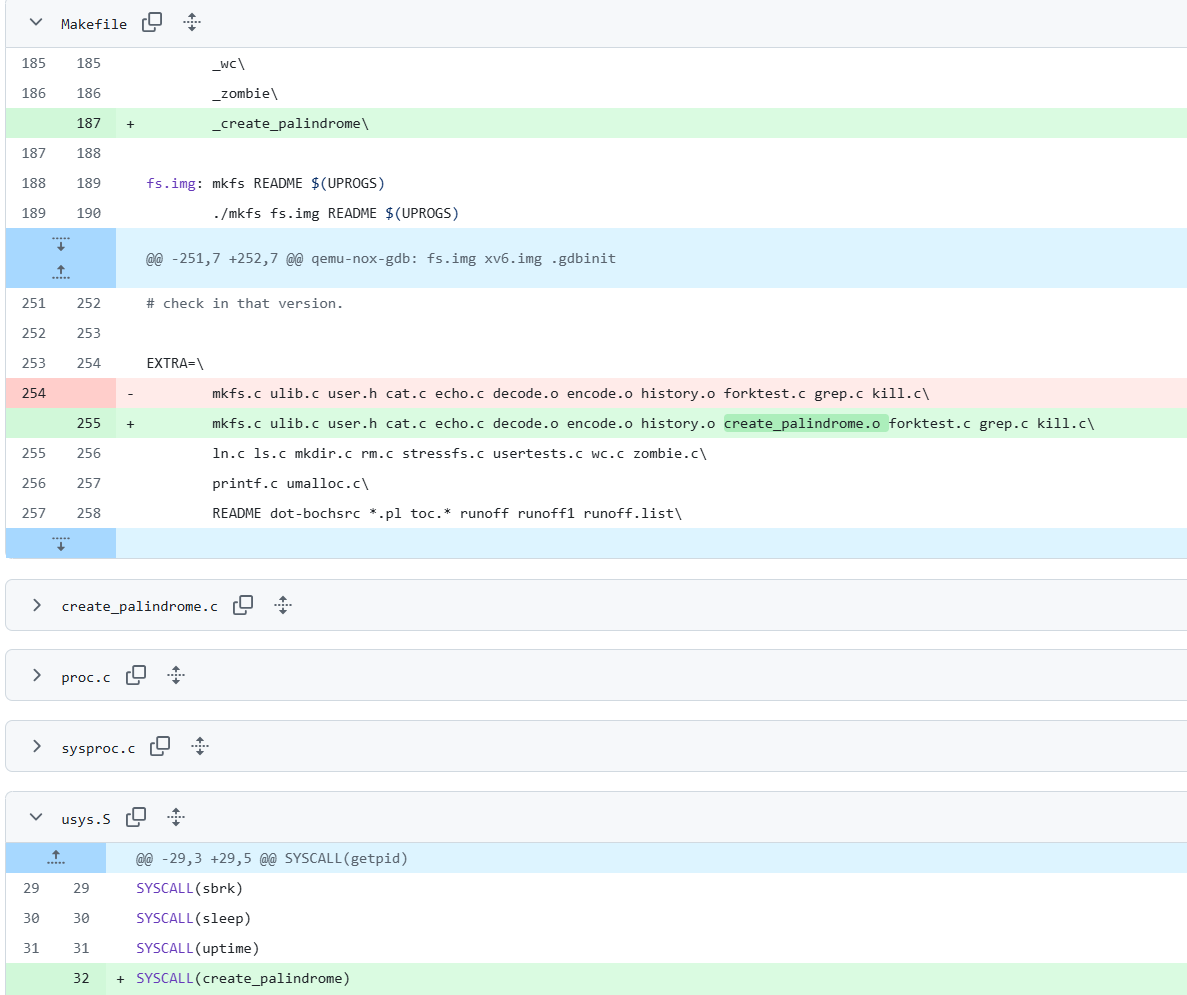
یکسری دیفاین کردن ها که باقی مانده بود را در فایل های بالا با توجه به اینکه فراخوانی جدید می سازیم، اضافه می کنیم.



در گام بعد یک فایل جدید به نام پالیندروم می سازیم که برنامه سطح کاربر خواهد بود.

در این برنامه در صورت وارد کردن صحیح دستور توسط کاربر، ابتدا آرگومان ذخیره شده را در رجیستر برگزیده ذخیره می کنیم.

سپس تابع create\_palindrome را صدا می زنیم تا نتیجه برایمان محاسبه شود. در نهایت هم محتویات رجیستری که انتخاب کرده بودیم را بازسازی می کنیم.



در پایان هم به میک فایل و فایل usys.S ملزومات اجرای این سیستم کال را اضافه کرده و ایجاد این فراخوانی را به پایان می رسانیم.

**پیاده سازی فراخوانی های سیستمی**

**1) پیاده سازی فراخوانی سیستمی انتقال فایل**

جهت اضافه کردن این فراخوانی سیستمی ، باید فایل های مختلفی را تغییر داد که در ادامه به آن اشاره می کنیم

user.h

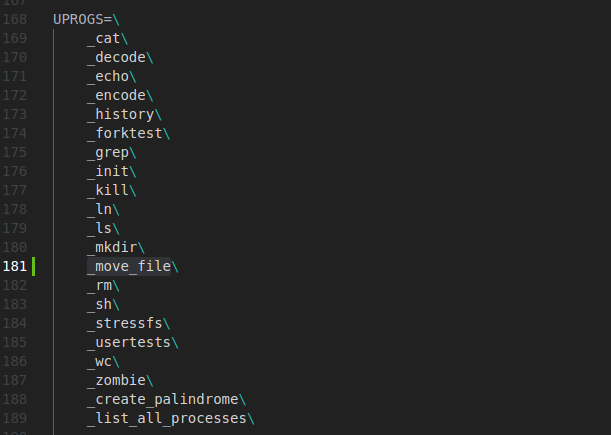
****

تابع را در دسترس کاربر قرار می دهیم .

defs.h

****

Makefile(UPROGS)

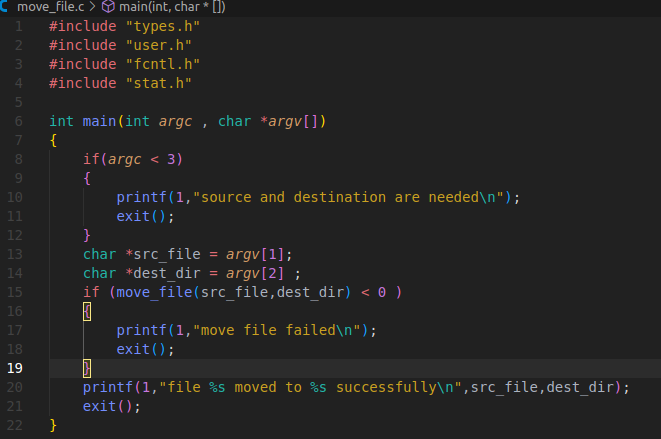
****

Makefile(EXTRA)

****

move\_file.c

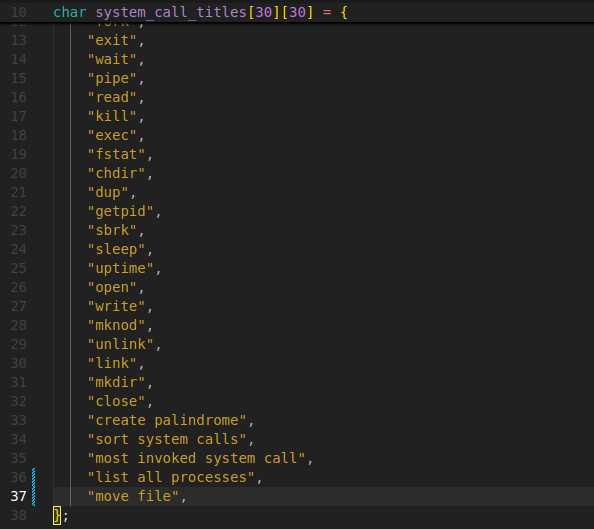
تست با برنامه ی سطح کاربر :

****

syscall.c

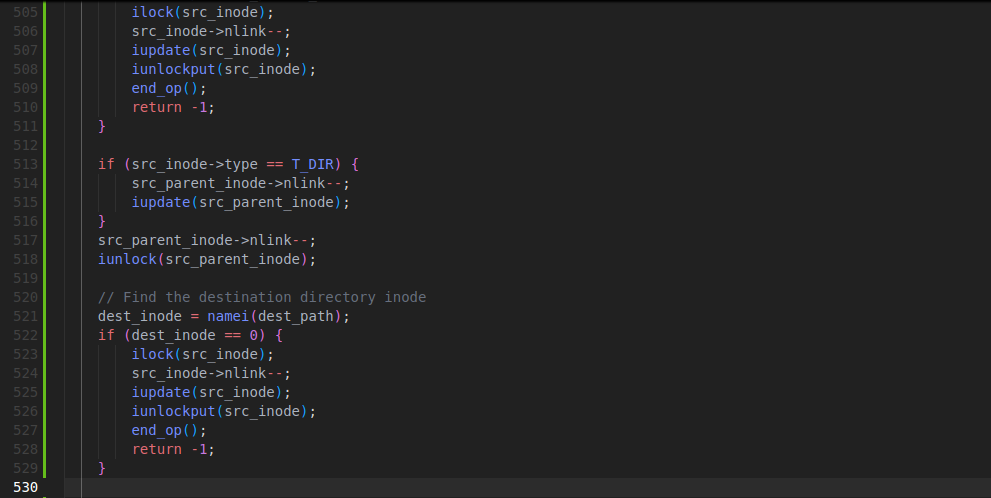
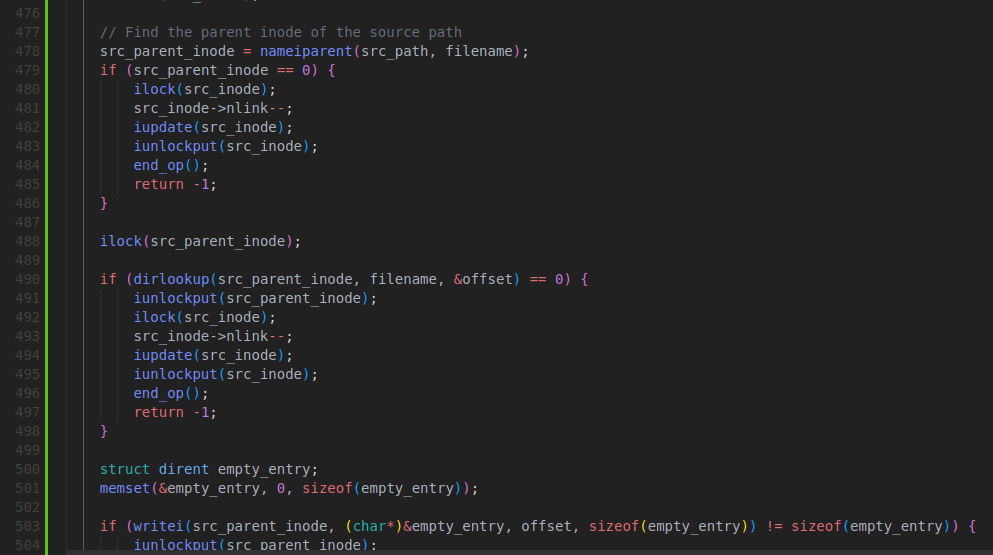
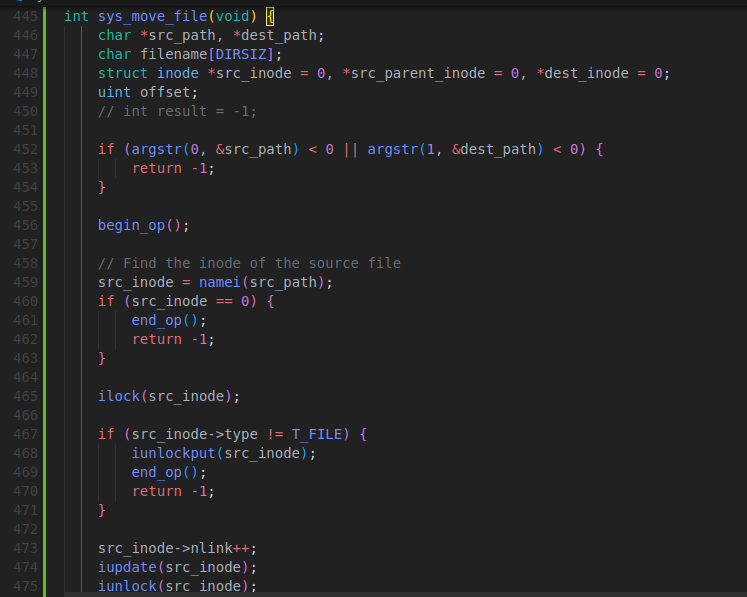
****

****

****

sysfile.c

اضافه کردن کد قابلیت خواسته شده (انتقال فایل ) به این فایل

****

usys.s

****

تابع move\_file به SYS\_move\_file مپ می شود

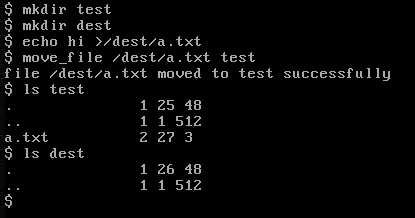
SYS\_move\_file عدد سیستم کال است که در syscall.h تعریف میشود.

syscall.h

****

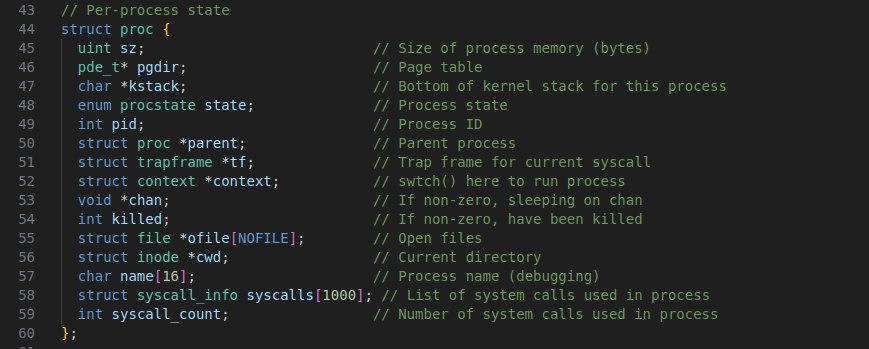
نتیجه

ایجاد دو دایرکتوری و ایجاد فایل در دایرکتوری dest و انتقال آن به دایرکتوری test

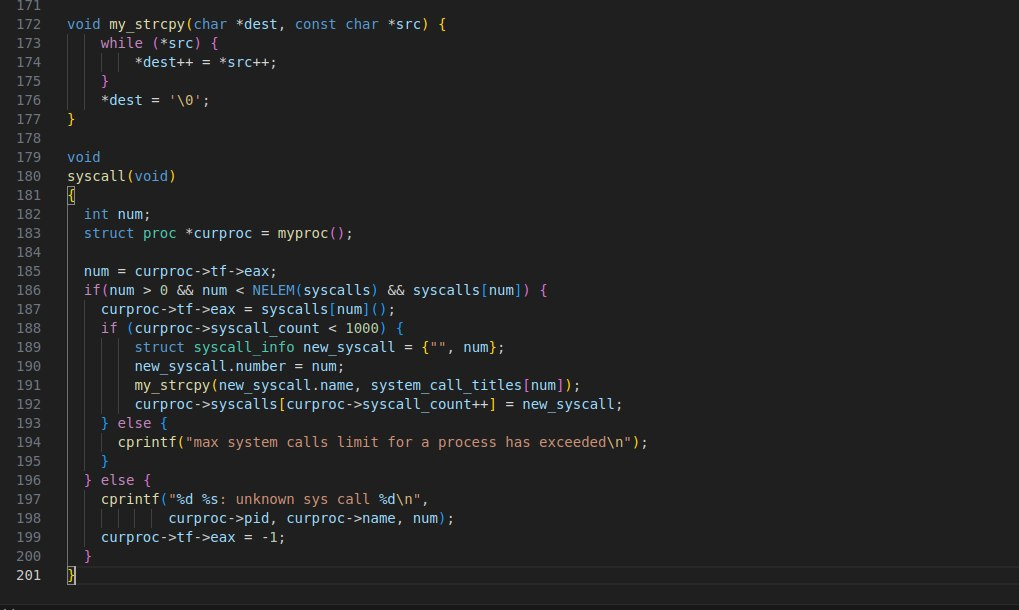
****

**2و3)**

برای سیستم کال های sort\_syscalls و get\_most\_invoked نیاز داریم که تمام سیستم کال های اجرا شده در یک پراسس را داشته باشیم. به این منظور در struct proc که اطلاعات پراسس در حال اجرا را نگه میدارد دو فیلد اضافه کرده ایم. Syscalls آرایه ای از سیستم کال ها است (syscall\_info حاوی اسم و شماره سیستم کال است) و syscall\_count تعداد سیستم کال را نشان می دهد.

****

در فایل کد syscall.c هم تغییراتی اعمال کردیم. بعد از فراخوانی هر سیستم کال دو فیلد گفته شده در استراکت آپدیت میشوند. تابع my\_strcpy برای کپی کردن نام سیستم کال ها استفاده میشود.

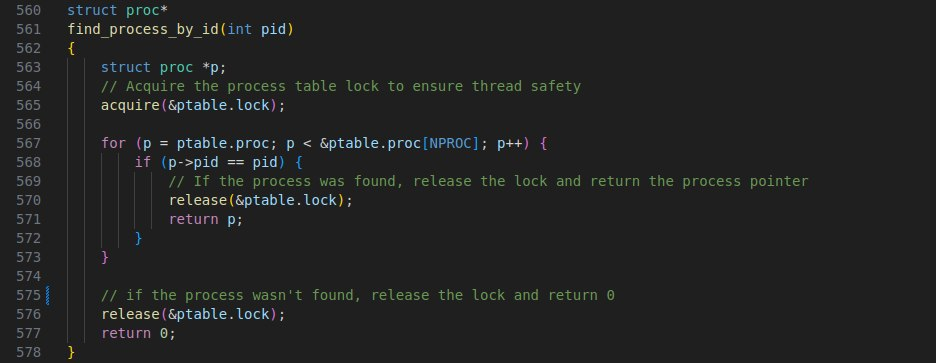
****

برای دسترسی به نام سیستم کال ها با استفاده از شماره آن ها آرایه ای از استرینگ تعریف شده که در هدر syscall\_titles.h قرار دارد تا در فایل های مختلف استفاده شود.

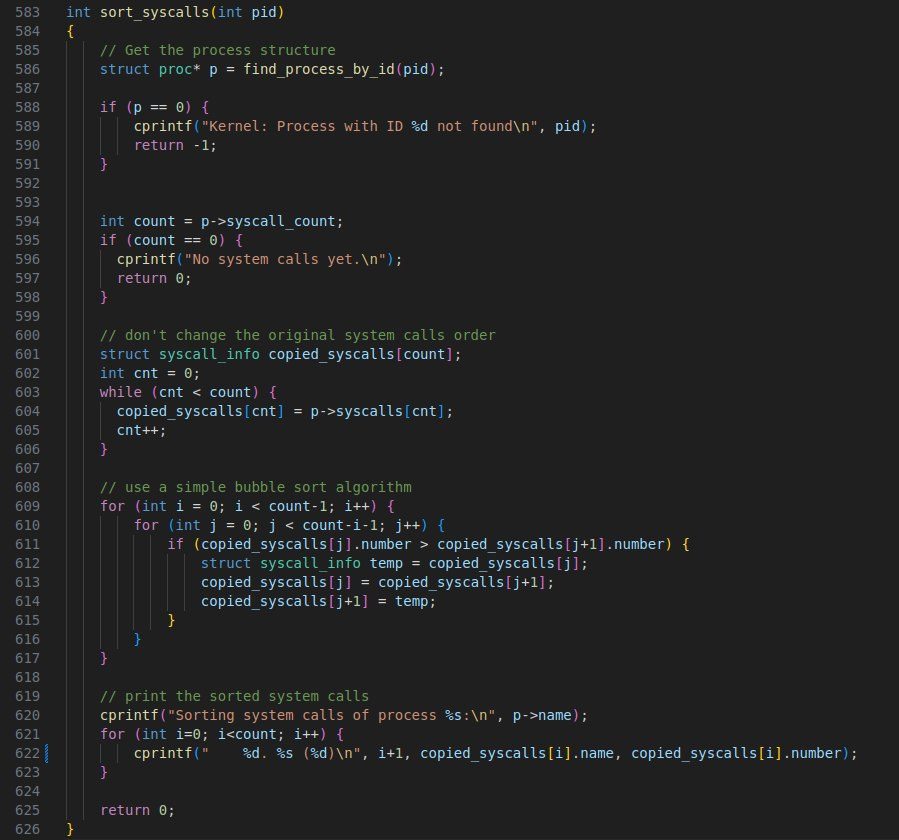
****

تعدادی تابع در فایل proc.c نیاز اضافه شده اند که به توضیح آن ها میپردازیم.

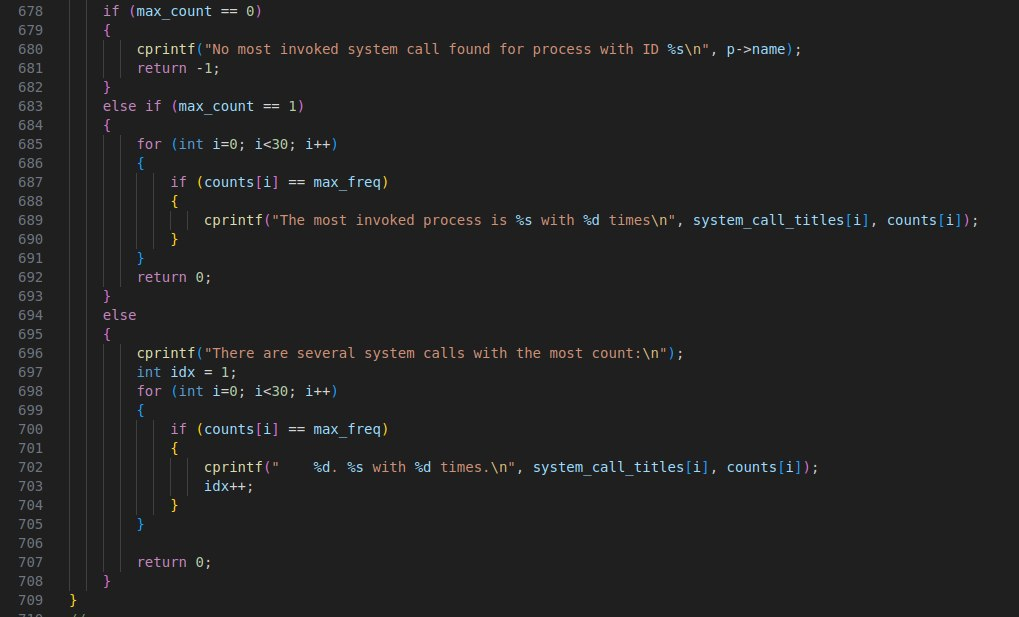
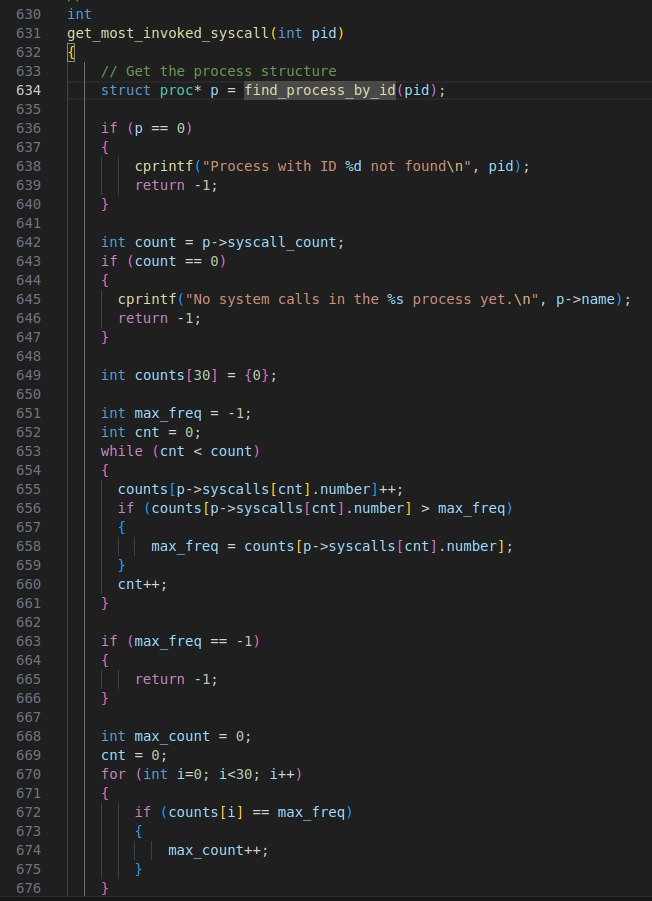
این تابع بر اساس آیدی داده شده به جستجوی پراسس در page table میگردد. اگر پیدا نکند صفر برمیگرداند و در غیر این صورت اشاره گر به استراکت آن پراسس.

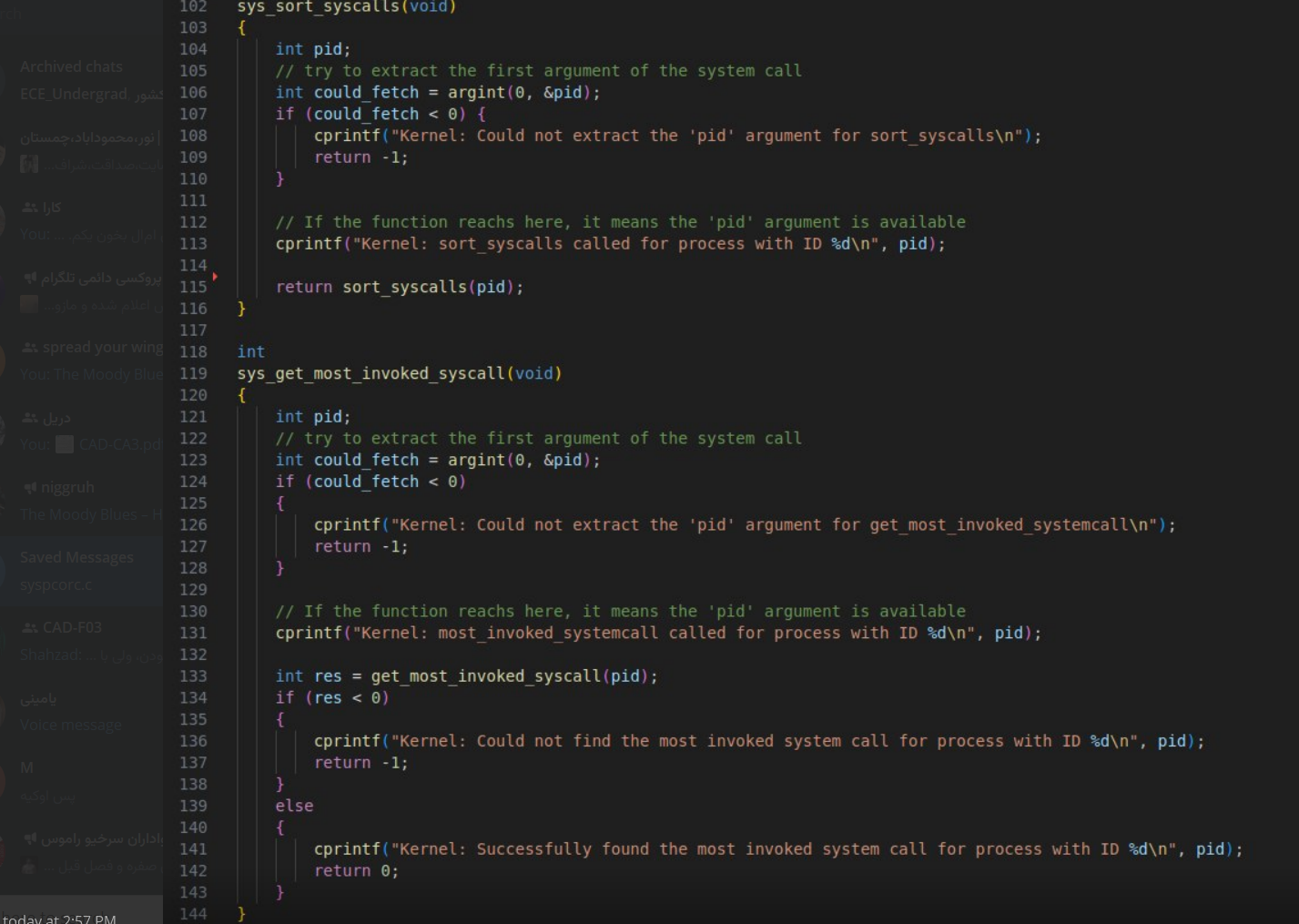
****

این تابع سیستم کال های یک پراسس را بر اساس شماره شان مرتب میکند (صعودی). در ابتدا بر اساس آیدی پراسس را پیدا کرده و اگر وجود نداشت پیغام مناسبی را چاپ میکند. در غیر این صورت اطلاعات سیستم کال ها رادر آرایه جدیدی میریزد. برای این که ترتیب اصلی سیستم کال ها را عوض نکنیم. سپس با استفاده از الگوریتم بابل سورت به مرتب سازی میپردازد در نهایت نیز پراسس های مرتب شده با نام و شماره شان چاپ می شوند.

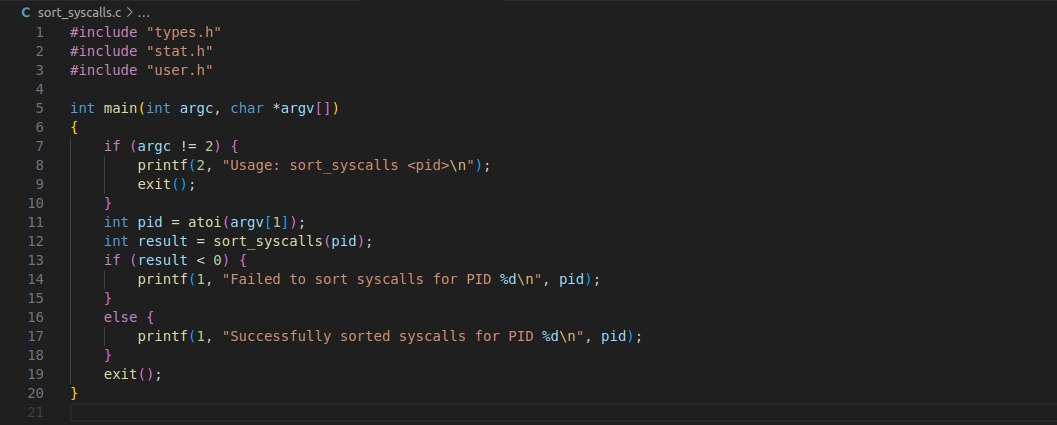
****

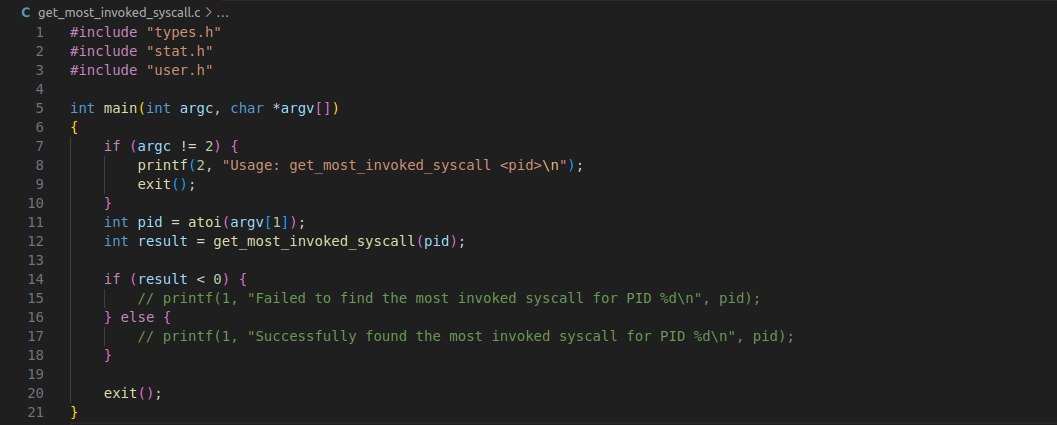
این تابع برای پیدا کردن سیستم کال با بیشترین تعداد تکرار است. ابتدا به مانند تابع قبل پراسس را پیدا میکند. سپس آرایه ای برای شمردن سیستم کال ها ساخته میشود و با پیشمایش روی سیستم کال ها آرایه آپدیت میشود. همچنین متغیر max\_freq نیز. در ادامه دو حالت داریم. تنها یک سیستم کال بیشترین تعداد تکرار را داشته باشد (اکثر مواقع) و اینکه چند سیستم کال این خاصیت را داشته باشند. هر دو حالت هندل شده است. در انتها نیز موارد خواسته شده چاپ میشوند (اسم و تعداد تکرار)

****دو تابع به sysproc.c نیز اضافه شدند که برای اجرای دو سیستم کال هستند. بعد از ارور هندلینگ تابع هرکدام کال شده است.

****

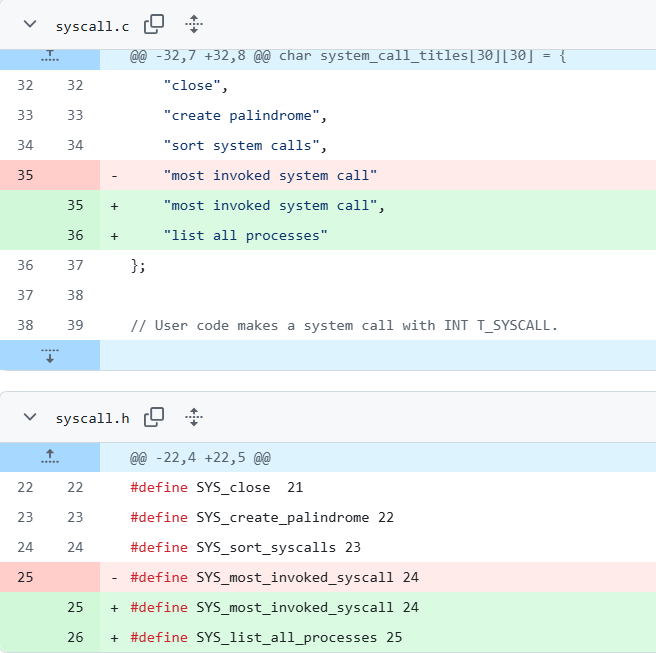
حالا که پیاده سازی کامل شده است میتوانیم تست کنیم. دو فایل sort\_syscalls.c و get\_most\_invoked\_syscall.c نوشته شده اند.

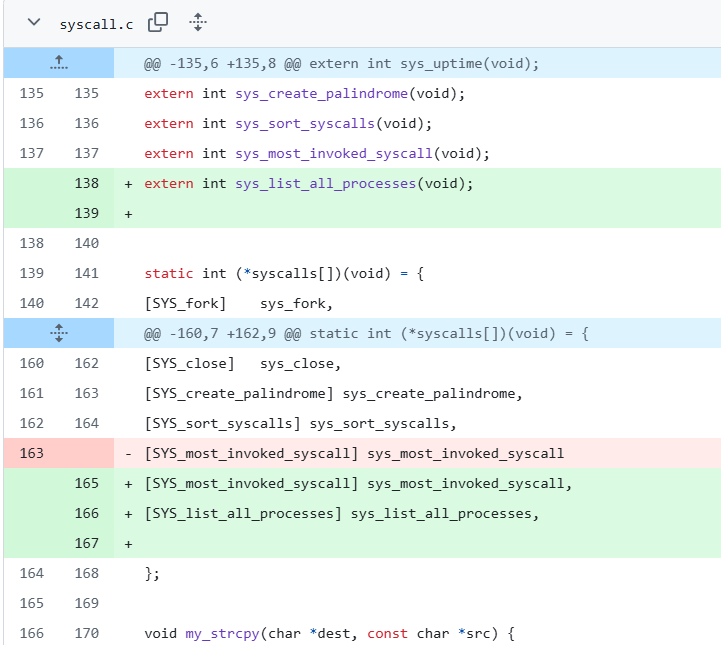
****

****

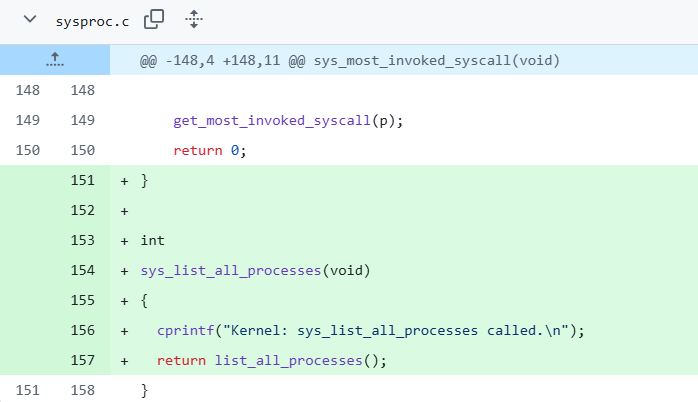
**4) پیاده سازی فراخوانی سیستمی لیست کردن پردازه ها**

در این بخش ما به کمک برخی ویژگی هایی که در بخش های قبل اضافه شده و برخی ویژگی هایی که خودمان اضافه می کنیم، این قابلیت را به سیستم می دهیم.

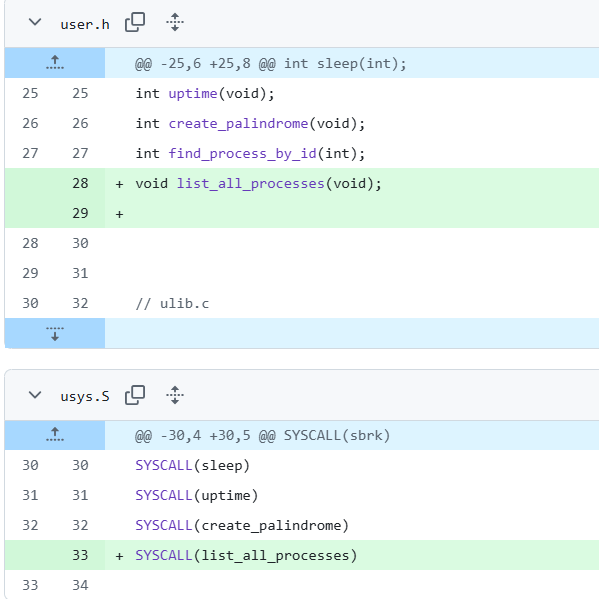




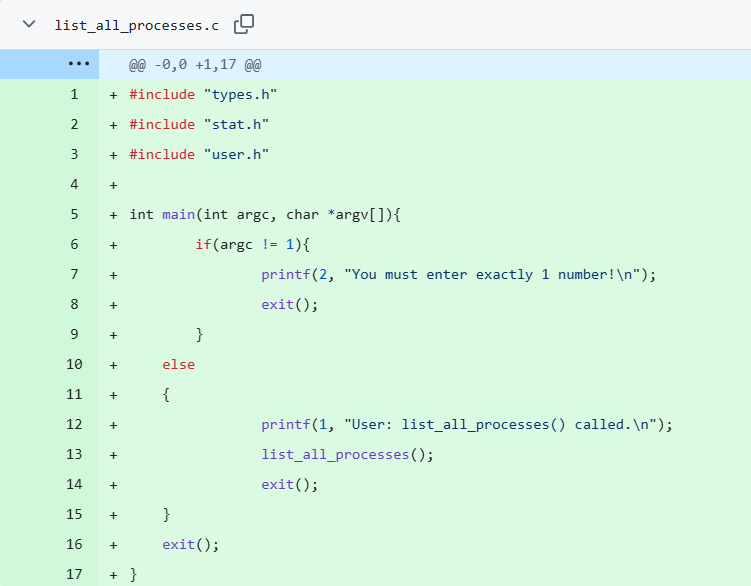
مجدد مانند موارد قبلی یک شماره به سیستم کال جدید خود نسبت می دهیم و آن را تعریف می کنیم. و همچنین در فایل هایی که نیاز به تعریف این سیستم کال جدید است آن را تعریف می کنیم.



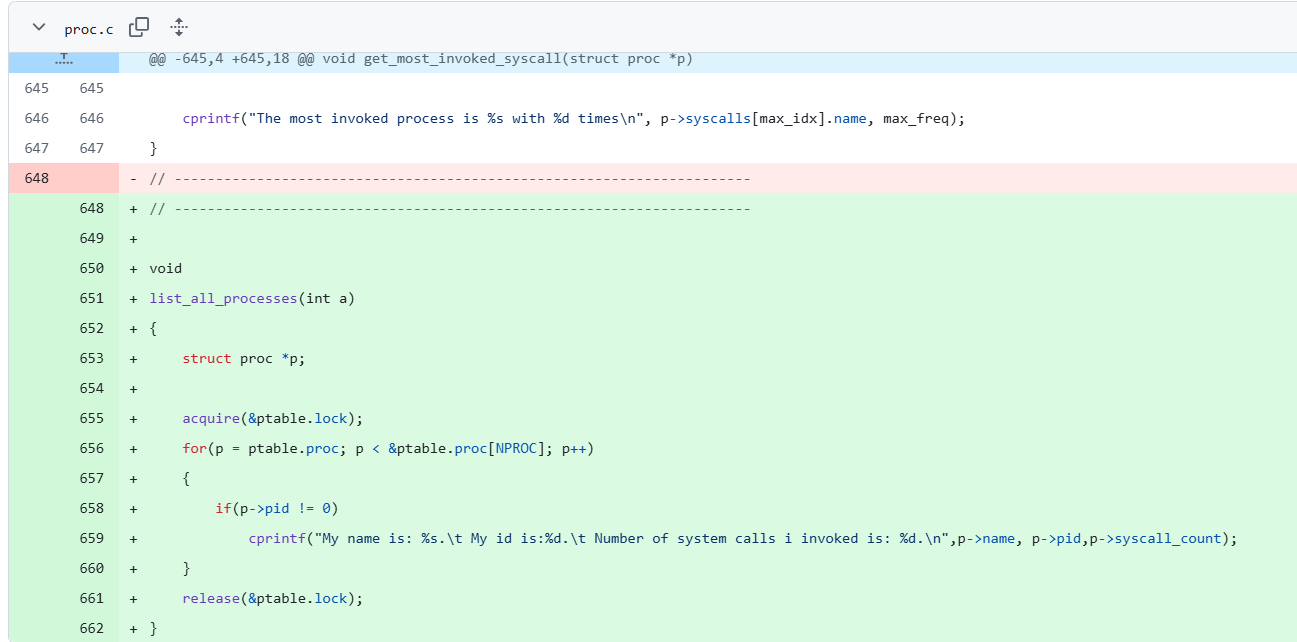
تابع مربوط به بخش هسته را در فایل بالا اضافه می کنیم. این تابع هم مثل موارد قبلی یک تابعی که قرار است محاسبات ما را انجام دهد صدا می زند.



در تابع usys.S نام سیتسم کال جدید را به سیستم کال های گذشته اضافه می کنیم. همچنین در هدر یوزر تابع برای لیست کردن پراسس ها در سطح کاربر را دیکلیر می کنیم.

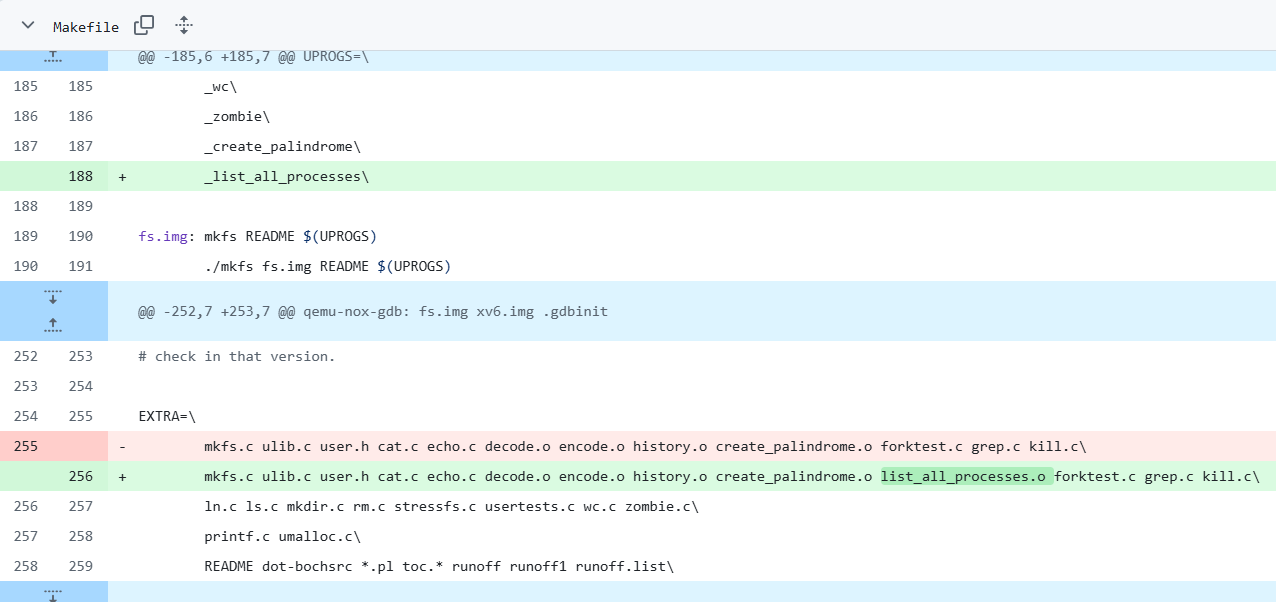


یک فایل جدید درست می کنیم که این فایل حکم سطح کاربر را دارد. در این بخش ابتدا مشخص می کنیم که کاربر درست ورودی وارد کرده باشد و سپس تابع مربوط به لیست کردن همه پراسس ها را اجرا می کنیم.

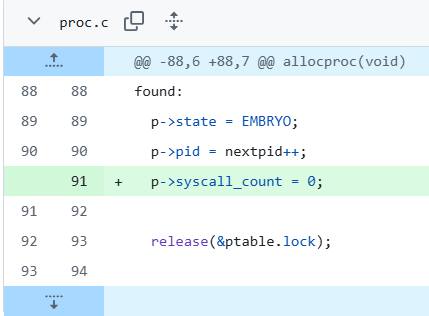


در فایل proc.c تعریف این تابع را قرار می دهیم. برای لیست کردن تمام پراسس های فعلی ما از پیج تیبل که پراسس های هر لحظه را دارد استفاده می کنیم. به این صورت که به کمک پارامتری که در بخش های قبل تعریف شده بود و تعداد سیستم کال های یک پراسس را محاسبه می کرد می توانیم به جواب برسیم.

روند کار اینطور است که روی پیج تیبل پیمایش انجام می دهیم و به ازای هر پراسسی که وجود داشت تعداد سیستم کال هایی که انجام داده است را به کاربر نشان می دهیم.



مجددا تغییرات میک فایل را انجام می دهیم و سیستم کال جدید را به آن اضافه می کنیم.



بازگرداندن تعداد سیستم کال پراسس بخشی حیاتی است به این دلیل که در صورت اشتباه در این بخش، نتیجه های بدست آمده اشتباه خواهد شد. پس مهم است که هنگامی که یک پراسس قرار است به چیزی تعلق گیرد تعداد سیستم کال هایش به صفر برگردانده شود.