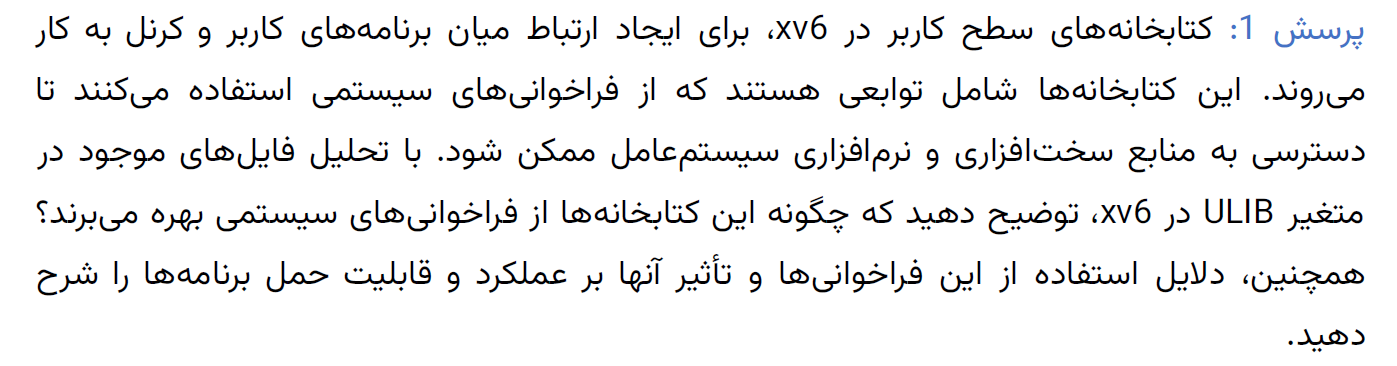
به نام خدا

گزارش آزمایشگاه آزمایش دوم درس سیستم عامل

آیدین کاظمی: 810101561 علی زیلوچی: 810101560 بابک حسینی محتشم: 810101408

مقدمه:



پاسخ سوال 1:

در ابتدا متغیر نام برده را داخل Makefile جانمایی میکنیم:

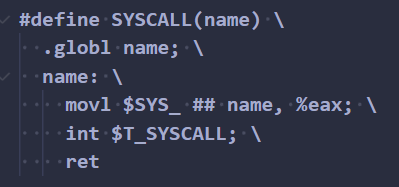


که همانطور که مشاهده میشود از لینک شدن چهار آبجکت فایل تشکیل شده، که سورس هر کدام را بررسی میکنیم:

این کد شامل توابعی برای عملیات‌های پایه‌ای رشته و حافظه در کتابخانه سطح کاربر است که بیشتر آن‌ها به سیستم کال نیاز ندارند. توضیحات هر تابع:

* **strcpy، strcmp، strlen:** برای کپی، مقایسه، و اندازه‌گیری طول رشته‌ها استفاده می‌شوند.
* **memset** : آرایه‌ای از حافظه را با یک مقدار مشخص پر می‌کند.
* **strchr** : اولین حضور یک کاراکتر خاص در رشته را پیدا می‌کند.
* **gets** : با استفاده از سیستم کال read، ورودی را از کاربر می‌خواند و در بافر ذخیره می‌کند.
* **stat** : اطلاعات فایل را با سیستم کال‌های open، fstat و close به دست می‌آورد.
* **atoi** رشته‌ای از اعداد را به یک عدد صحیح تبدیل می‌کند.
* **memmove** : داده‌ها را از مبدا به مقصد کپی می‌کند.



در ابتدای این فایل اسمبلی ماکرو SYSCALL وجود دارد که یک تابع تعریف میکند که با استفاده از نام هر سیستم کال، ابتدا شماره مربوط به آن را در رجیستر EAX ذخیره کرده (که این شماره ها در syscall.h ذخیره شده اند) و سپس int 64 را صدا میکند تا یک اینتراپت رخ دهد و trap اتفاق افتاده، با تغییر مود از user به kernel و درون کرنل مقدار eax خوانده میشود تا بفهمد کدام سیستم کال رخ داده و عملیات مورد نظر را انجام دهد. نهایتا با دستور ret به user مود برگشته و مقدار برگشتی داخل eax ذخیره شده است:



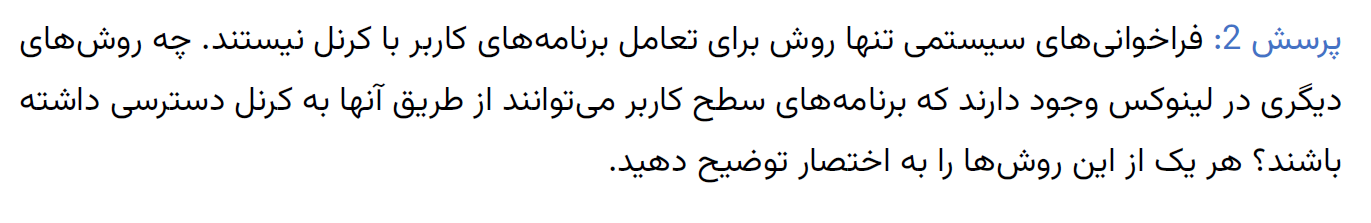
این فایل شامل تعریف تابع printf میباشد، که عملیات چاپ کردن را با استفاده از دو تابع کمکی putc (نوشتن یک کاراکتر در فایل دیسکریپتور مقصد با سیستم کال write) و printint (فرمت دهی و نوشتن اعداد صحیح یا هگز با کمک تابع putc) انجام میدهد، به صورتی که یک رشته را دریافت کرده و مقدار متناسب را با این دو تابع در فایل دیسکریپتور مقصد مینویسد.



این فایل شامل تعریف سه تابع morecore، malloc و free بر اساس کتاب کرنیگان و ریچی میباشد که تابع free وظیفه آزاد کردن حافظه تخصیص یافته و تابع malloc وظیفه تخصیص حافظه با استفاده از تابع morecore را دارد. تنها درون تابع morecore از سیستم کال استفاده شده (sbrk که فضای حافظه پراسس را بیشتر میکند).

دلایل استفاده از این فراخوانی ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه ها را شرح دهید:

از آنجایی که انجام بعضی از اعمال توسط کاربر ممکن است امنیت یا کارایی سیستم را به خطر بیندازد، و همچنین بعضی از دسترسی ها صرفا توسط سیستم عامل انجام میشود، وظیفه انجام برخی از کار ها به سیستم عامل سپرده شده و اینترفیس استفاده از آنها توسط سیستم کال ها پیاده سازی شده اند تا کاربر به شکل امنی بتواند از این خدمات سیستم استفاده کند. از آنجایی که بخش بزرگی از اینترفیس سیستم کال ها به طور کلی میان سیستم عامل های مختلف به صورت یکسان پیاده سازی شده اند، بخش هایی از برنامه که این سیستم کال ها را شامل میشوند قابل حمل و بقیه بخش های غیر مشترک غیر قابل حمل میباشند. همچنین استفاده از سیستم کال ها به دلیل تغییر مود از کاربر به کرنل و برخی اعمال مرتبط شامل سرباری جزئی میباشد.



پاسخ سوال 2:

برخی روش های دیگر به اختصار توضیح داده شده اند:

1. Interrupt:

وقفه یا اینتراپت دارای دو نوع است: نرم افزاری (که به آن trap هم گفته میشود) و سخت افزاری. وقفه های سخت افزاری (که عموما توسط عملیات های ورودی خروجی ایجاد میشوند) عموما به صورت Asynchronous (بدون وابستگی به روند اجرای فعلی) انجام شده و توسط یک سخت افزار ایجاد میشوند (به طور مثال فشردن کیبورد، حرکت دادن موس یا اتمام آی او). وقفه های نرم افزاری (که سیستم کال ها نیز جزئی از آنها هستند) توسط یک نرم افزار و به صورت Synchronous تولید شده و به طور کلی سه نوع سیستم کال، اکسپشن (که در هنگام بروز خطا مانند تقسیم بر 0 رخ میدهند) و سیگنال (که برای آگاه سازی پراسس ها از اتفاقاتی خاص مانند SIGINT در لینوکس (اتمام پراسس با CTRL + C) یا SIGKILL برای اتمام فوری پراسس استفاده میشوند) را شامل میشوند.

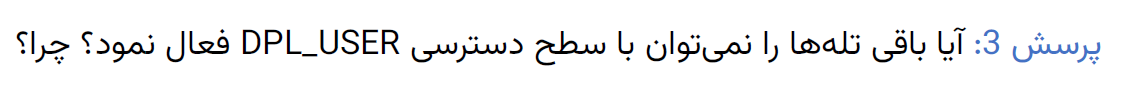
1. Pseudo-Filesystems

شامل فایل سیستم هایی مانند proc/ و sys/ میباشد که اطلاعاتی را در مورد تنظیمات و فرایند های کرنل را در اختیار کاربر قرار داده و کاربر میتواند با نوشتن یا خواندن از این فایل ها، اطلاعاتی در مورد سیستم به دست آورده یا تنظیمات را تغییر دهند.

1. Netlink Sockets

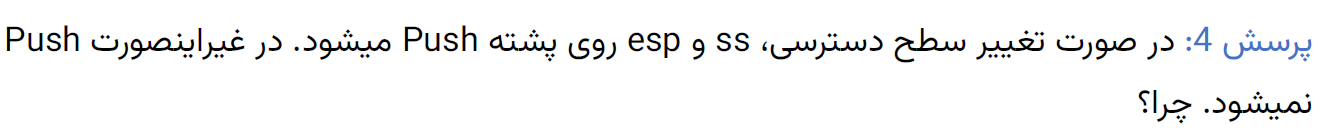
یک وسیله ارتباط میان کاربر و کرنل مبتنی بر سوکت است که برای انتقال داده ها بین کرنل و برنامه کاربر استفاده میشود. معمولا برای شبکه بندی و مدیریت سیستم استفاده میشود.

سازوکار اجرای فراخوانی سیستمی در xv6



پاسخ سوال 3:

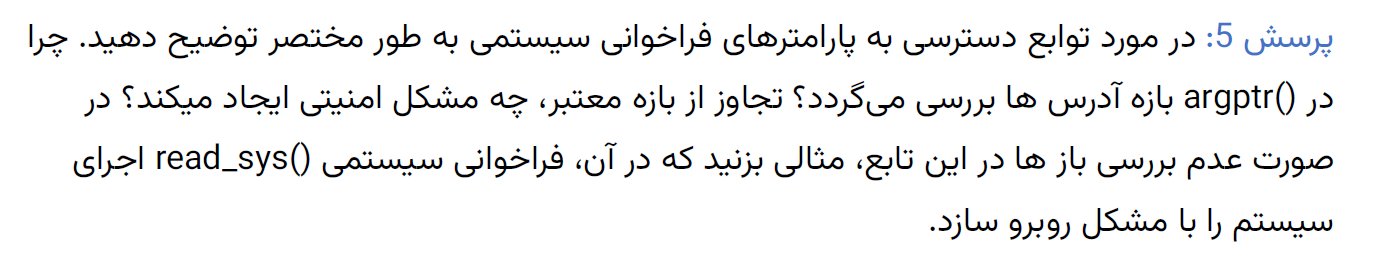
خیر. در سیستم عامل xv6، در صورتی که کاربر بخواهد تله دیگری را فعال کند خطای protection exception دریافت خواهد کرد، چرا که تله های سطح کرنل دارای سطح دسترسی پایین تری از سطح دسترسی کاربر هستند. از فواید این قاعده نیز میتوان به جلوگیری از دسترسی غیر مجاز به کرنل و بروز خطاهای احتمالی غیر عمدی (وجود باگ داخل برنامه) یا عمدی (سوء استفاده کاربر) اشاره کرد، که در صورت نبود این قاعده ممکن بود امنیت یا کارایی سیستم را به خطر بیندازند.



4:

پاسخ سوال 4:

پراسس ها در سیستم عامل xv6 دارای دو نوع استک میباشند: استک کاربر و استک کرنل. هنگام تغییر مود از کاربر به هسته، مقادیر رجیستر های ss و esp که تا به حال به محتویات استک کاربر اشاره داشتند، دستخوش تغییر شده و به استک کرنل اشاره میکنند. حال از آنجایی که پس از اتمام کار کرنل باید به جای قبلی برنامه کاربر برگشته و اجرا را از سر بگیریم، برای جلوگیری از گم شدن محتویات فعلی esp و ss (و در نتیجه از دست رفتن مکان اجرای قبلی و عدم توانایی از سر گیری برنامه) این دو مقدار باید بر روی استک push شوند. حال در صورتی که تغییر مود نداشته باشیم، مقادیر esp و ss همچنان برای استک کاربر معتبر بوده و در نتیجه نیازی به ذخیره سازی دوباره آنها وجود ندارد.



پاسخ سوال 5:

توابع دسترسی به پارامتر های فراخوانی سیستمی عبارتند از argint، argptrو argstr که وظیفه هر کدام دریافت شماره آرگومان و مقصد برگرداندن آن، و گشتن داخل استک بر اساس آن شماره آرگومان، برای پیدا و قرار دادن آرگومان مد نظر در مقصد میباشد. هر کدام را به اختصار توضیح میدهیم:

1. argint

این تابع وظیفه بازگرداندن یک مقدار صحیح از استک را دارد. در این تابع ابتدا آدرس دسترسی به آرگومان n ام محاسبه میشود، که از آنجایی که در سر استک آدرس بازگشت از تابع ذخیره میشود و پس از آن آرگومان ها به ترتیب از اولی تا آخری قرار دارند، باید به اندازه n + 1 خانه داخل استک جلو برود. حال از آنجایی که مقدار استک از حافظه بیشتر به کمتر پر میشود، یعنی باید داخل حافظه مربوط به استک به اندازه 4 \* (n+1) بایت از نقطه سر استک جلو برویم، که این نقطه در رجیستر esp ذخیره شده است. بنابراین آدرس آرگومان مد نظر برابر با esp + 4 \* (n+1) خواهد شد. در مرحله بعد این آدرس محاسبه شده به تابع fetchint داده شده، و به عنوان آرگومان دیگر آن پوینتر ip داده میشود که به عنوان ورودی توسط این تابع دریافت شده. نهایتا fetchint در صورتی که آدرس حافظه نشان شده بالاتر یا مساوی رجیستر sz نباشد، آن را به عنوان پوینتر داخل ip قرار میدهد و موفقیت آمیز باز میگردد.

1. argptr

این تابع وظیفه بازگرداندن یک پوینتر سایز مشخص را دارد، که آدرس شروع آن در خانه n ام استک ذخیره شده است. در این تابع ابتدا با کمک argint آدرس شروع پوینتر مد نظر از داخل استک بازیابی میشود (که سناریو های شکست آن نیز در بخش قبل بررسی شد). سپس پارامتر های ورودی به تابع بررسی میشوند (مانند منفی نبودن سایز و داخل محدوده حافظه بودن پوینتر مشخص شده)، و در صورتی که مشکلی وجود نداشته باشد، مقدار باز گردانده شده توسط argint به عنوان پوینتر مد نظر قرار میگیرد.

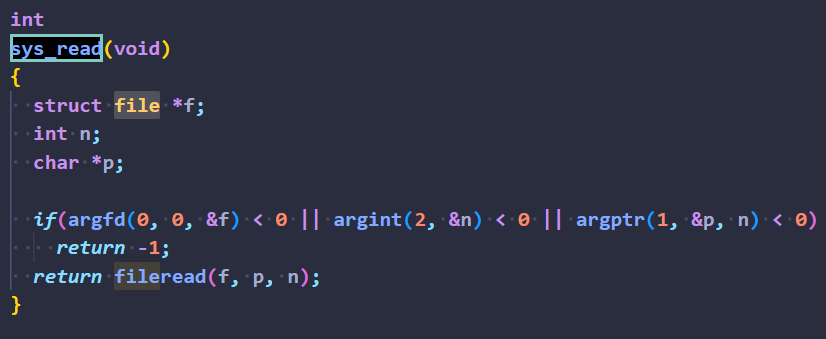
1. argstr

این تابع وظیفه بازیابی یک رشته را دارد. مانند تابع قبل، ابتدا با کمک argint آدرس شروع رشته را از داخل استک بازیابی میکند. سپس آدرس بازیابی شده را به تابع fetchstr پاس میدهد. این تابع نیز ابتدا بررسی میکند که آدرس داده شده در بازه حافظه پراسس باشد. در مرحله بعد مقدار پوینتر بیرونی را برابر آدرس شروع قرار داده و سپس از آدرس شروع یکی یکی جلو رفته و در صورت رسیدن به کاراکتر null، آدرس آن خانه را از خانه شروع کم کرده و به عنوان طول رشته باز میگرداند. در غیر این صورت و در صورت رسیدن به انتهای محدوده مشخص شده حافظه پراسس، مقدار -1 باز میگرداند.

تمامی این توابع داخل محدوده بودن آدرس فراخوانی شده را بررسی میکنند، که در غیر این صورت بدیهتا مشکلاتی از قبیل نوشتن در حافظه پراسس دیگر و ایجاد مشکلات امنیتی (مانند تغییر سطح دسترسی) و نهایتا ایجاد خطا و خرابی کرنل به وجود می آیند.

سیستم کال sys\_read

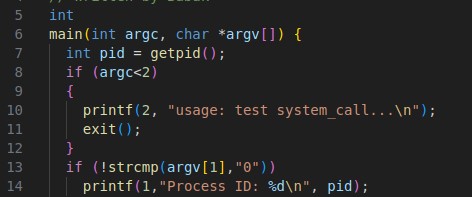
در این سیستم کال که از یک فایل دیسکریپتور، به مقدار مشخص خوانده و در بافر مشخص ذخیره میکند، هر سه عنصر صحت فایل دیسکریپتور (به عنوان آرگومان اول)، داخل محدوده حافظه بودن بافر (آرگومان سوم) و همچنین صحت طول مشخص شده (به عنوان آرگومان دوم) را بررسی میکند:



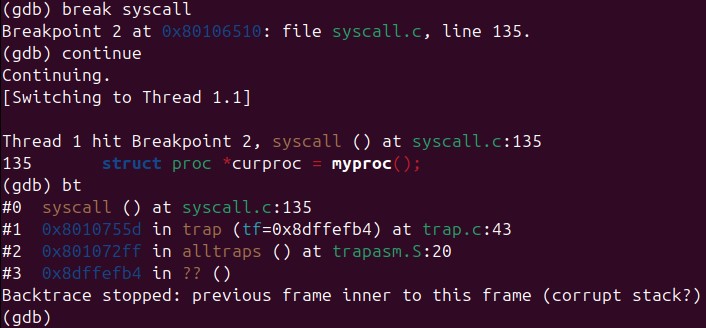
در صورتی که مشکلی وجود نداشته باشد، به اندازه مشخص شده از فایل دیسکریپتور خوانده و داخل بافر قرار میدهد (با کمک fileread، که در صورتی که پیش از رسیدن به سایز تعیین شده به EOF برسد جلوتر نخواهد رفت). حال در صورتی که این بررسی ها انجام نمیشد، ممکن بود طول خیلی بزرگی داده شود (که منجر به خارج شدن از فضای حافظه پراسس میشد) یا بافری خارج از محدوده حافظه پراسس داده شود که هر دو باعث نوشته شدن در حافظه پراسس دیگر و ایجاد مشکل میشدند.

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسطgdb :

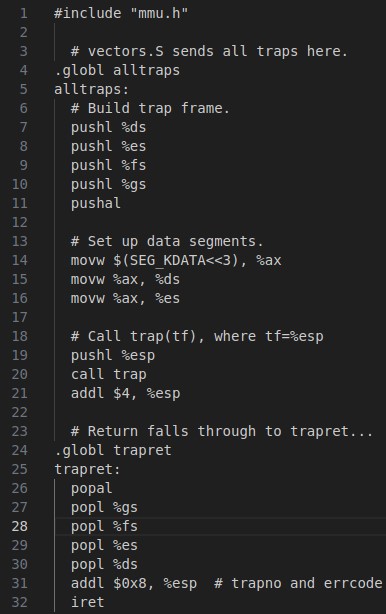
برنامه سطح کاربری به نام test تشکیل می‌دهیم که در آن در صورت وارد کردن 0 از تابع getpid استفاده می‌کنیم. همچنین از این برنامه بعدا برای تست سیستم‌کال‌هایی که اضافه می‌کنیم استفاده می‌کنیم.



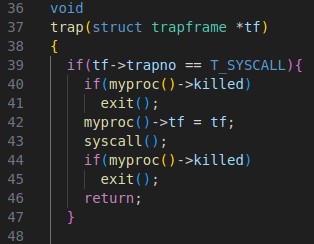
ابتدا در تابع syscall یک breakpoint می‌دهیم و پس از اجرای برنامه سطح کاربر به breakpoint می‌رسیم. سپس از دستور bt استفاده می‌کنیم.



همان‌طور که می‌توان دید، این دستور، call stack برنامه را نشان می‌دهد. یعنی فراخوانی‌هایی که انجام شده تا به تابع فعلی برسیم. پس می‌توان دید که ابتدا به تابع alltraps می‌رویم. در این تابع مقادیر رجیسترها در استک push می‌شود سپس تابع trap صدا زده می‌شود. همچنین می‌توان تابع trapret را دید که احتمالا پس از اتمام trap صدا زده می‌شود و مقادیر رجیسترها را از استک pop می‌کند.



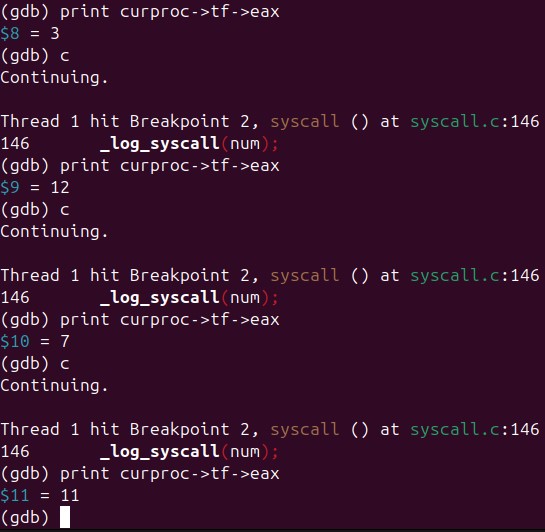
می‌توان دید که تابع trap در ابتدا بررسی می‌کند که trapnumber ذخیره شده در استک چند است و اگر برابر 64 بود که در xv6 برای فراخوانی‌های سیستمی است، تابع syscall را فراخوانی می‌کند.



با دو دستور up و down می‌توان بین توابعی که فراخوانی کردیم بالا و پایین برویم و چون الان در آخرین تابع فراخوانی شده هستیم دستور down کار خاصی انجام نمی‌دهد.

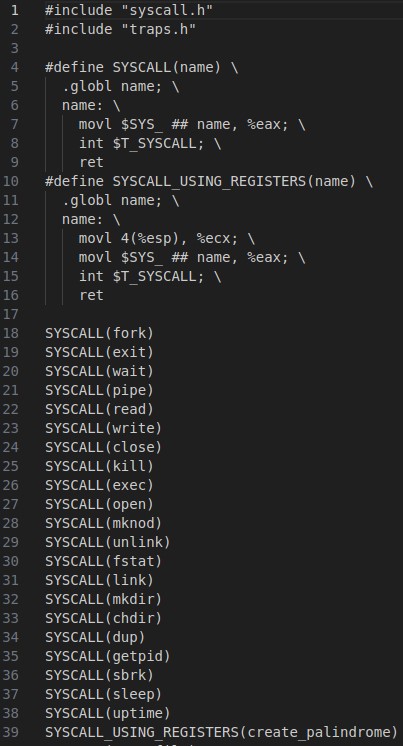


با استفاده از دستور c محتویات رجیستر eax که شماره سیستم‌کال در آن ذخیره شده را چندین بار میخوانیم. در ابتدا چند بار از سیستم‌کال read استفاده می‌شود که احتمالا هدف خواندن از ترمینال است. سپس یک بار سیستم‌کال fork اجرا می‌شود چون باید پردازه جدیدی ایجاد شود تا برنامه test را اجرا کند. سپس پردازه والد با سیستم‌کال wait منتظر اتمام کار پردازه جدید می‌شود. سپس از سیستم‌کال sbrk استفاده می‌شود. این سیستم‌کال مقدار فضای پردازه را تغییر می‌دهد پس احتمالا از آن برای تغییر فضای حافظه پردازه جدید استفاده می‌شود. سپس با سیستم‌کال exec کد مربوط به test در این پردازه کپی و اجرا می‌شود. در نهایت هم از سیستم‌کال getpid استفاده می‌شود.

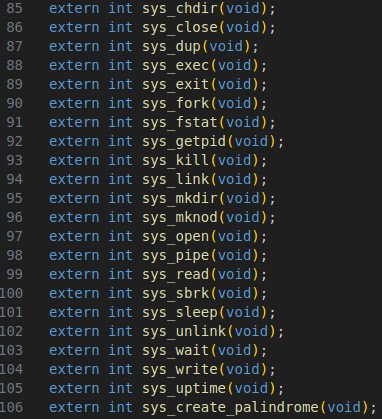


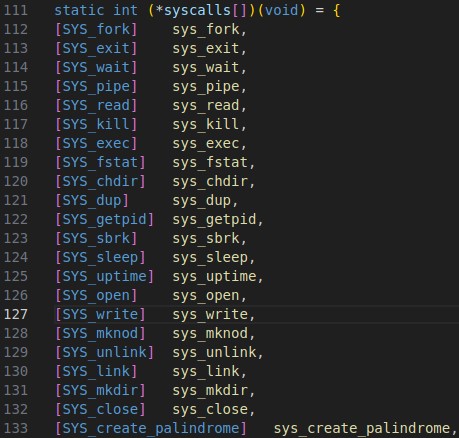
ارسال آرگومان های فراخوانی های سیستمی:

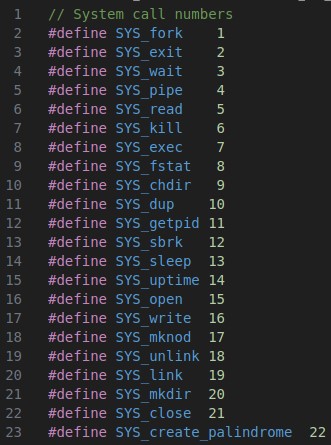
با بررسی فایل usys.s متوجه شدیم که در این فایل، ماکروی SYSCALL تابع پوشاننده سیستم‌کال‌های xv6 است. این ماکرو ابتدا شماره سیستم‌کال را در رجیستر eax می‌ریزد و سپس دستور interrupt می‌دهد. ما در این بخش برای ارسال آرگومان از طریق ثبات، ماکروی جدیدی به نام SYSCALL\_USING\_REGISTERS تعریف کردیم. چون پس از فراخوانی توابع در اسمبلی، آرگومان در استک ذخیره می‌شود، و می‌دانیم محتوای خانه اول استک آدرس بازگشت است پس در خانه دوم استک عددی که می‌خواهیم پالیندرومش را حساب کنیم قرار دارد پس محتوای خانه دوم استک را که در آدرس esp+4 قرار گرفته را در رجیستر ecx می‌ریزیم و سپس مشابه ماکروی SYSCALL عمل می‌کنیم. دلیل اینکه در رجیستر ecx می‌ریزم هم این است که این ثبات جزو ثبات‌های caller-saved registers است پس لزومی ندارد که مقدار آن پس از فراخوانی توابع تغییر کند و به همین دلیل می‌توان از آن به عنوان ثبات موقتی استفاده کرد.



در فایل syscall.c لیست توابع سیستم‌کال‌ها قرار گرفته پس sys\_create\_palindrome هم به این تابع اضافه می‌کنیم.

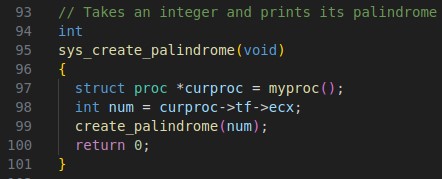


همچنین شماره سیستم‌کال جدید را به فایل syscall.h اضافه می‌کنیم.

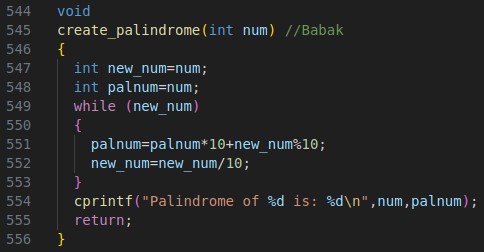


در فایل user.h هم این تابع این سیستم‌کال را declare می‌کنیم تا بتوان از آن در برنامه‌های سطح کار هم استفاده کرد. 

در فایل sysproc.c هم تابع sys\_create\_palindrome را اضافه کردیم که وظیفه آن مشابه وظیفه اکثر توابع این فایل این است که آگومان را بگیرد و آن را به تابع دیگری که create\_palindrome در فایل proc.c است بدهد. البته چون آرگومان را در رجیستر ecx ریختیم، آرگومان را از این رجیستر پردازه بر‌می‌داریم.

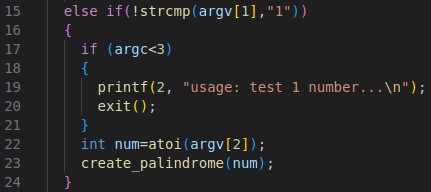


در نهایت منطق را در تابع create\_palindrome در فایل proc.c اضافه می‌کنیم که ابتدا پالیندروم عدد آرگومان ورودی را تشکیل داده و سپس چاپ می‌کنیم.



حال برای تست، برنامه سطح کاربر را تغییر داده و سیستم‌کال را تست می‌کنیم.

بخش اضافه شده به برنامه سطح کاربر:



تست سیستم‌کال: 

پیاده‌سازی فراخوانی‌های سیستمی:

.1 پیاده سازی فراخوانی سیستمی انتقال فایل:

برای پیاده‌سازی بقیه سیستم‌کال‌ها هم با کمی تغییرات مشابه قبل عمل می‌کنیم. در سیستم‌کال‌های بعدی چون از استک برای ارسال آرگومان‌ها استفاده می‌کنیم پس از همان ماکروی SYSCALL استفاده می‌کنیم.

اضافه کردن به usys.s: 

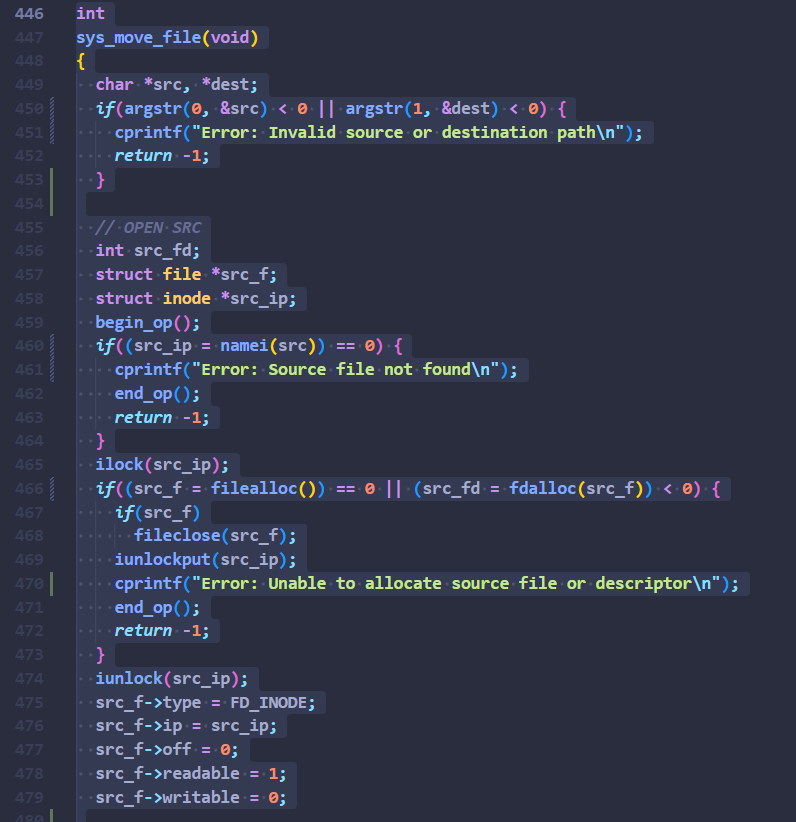
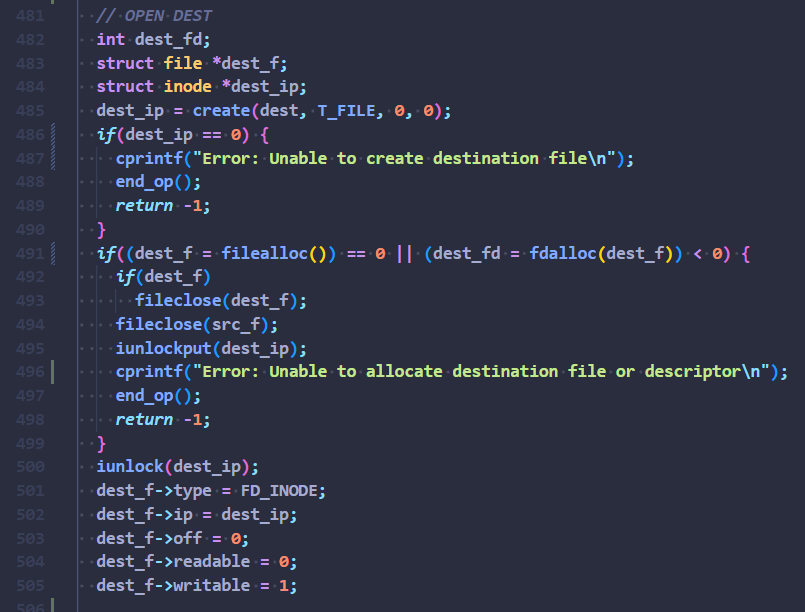
اضافه کردن به syscall.h: 

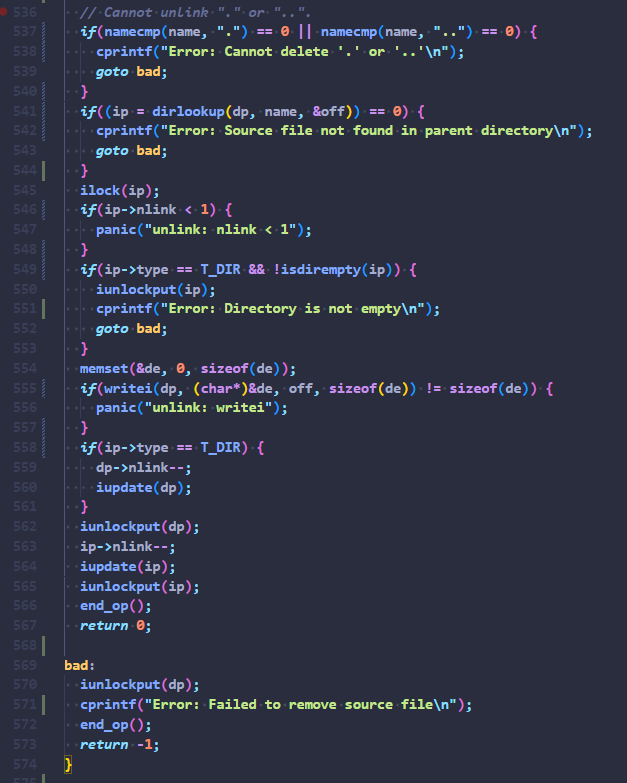
اضافه کردن به syscall.c: 

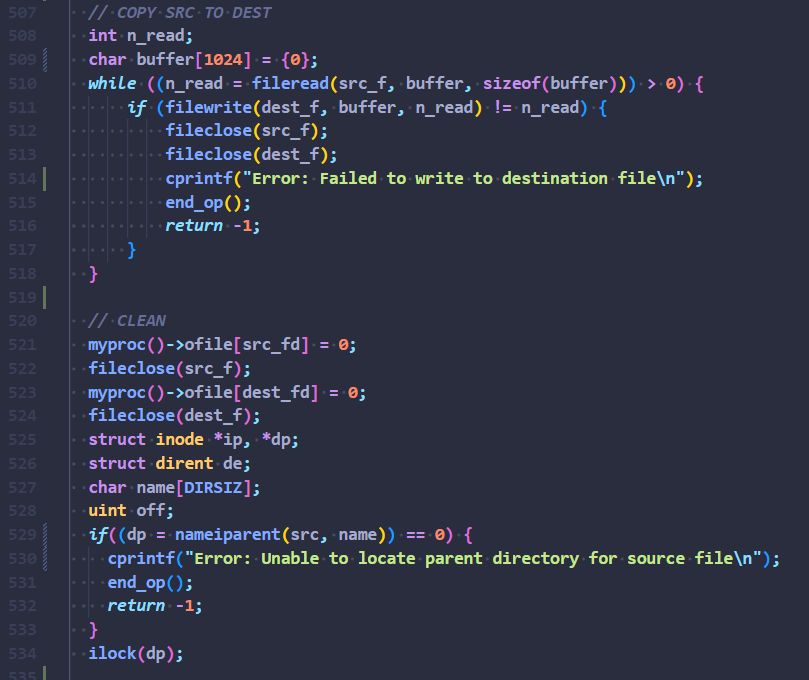


چون سیستم‌کال‌های مربوط به پردازه در sys\_proc.c و سیستم‌کال‌های مربوط به فایل در sysfile.c قرار دارند، پس این سیستم‌کال را در sysfile.c تعریف می‌کنیم. اکثر توابع این فایل هم آرگومان‌ها را از استک می‌گیریند و هم منطق را پیاده می‌کنند پس ما هم همین کار را می‌کنیم. چون سیستم‌کال جز بنیادین دستورات سیستم‌عامل است پس تصمیم گرفتیم از سیستم‌کالی در پیاده‌سازی move\_file استفاده نکنیم پس با کمک پیاده‌سازی بقیه سیستم‌کال‌ها، سیستم‌کال move\_file را پیاده‌سازی می‌کنیم. به صورت کلی این تابع فایل مبدا و فایل جدیدی در پوشه مقصد باز می‌کند و فایل مبدا را در فایل مقصد کپی می‌کند و فایل مبدا را حذف می‌کند.

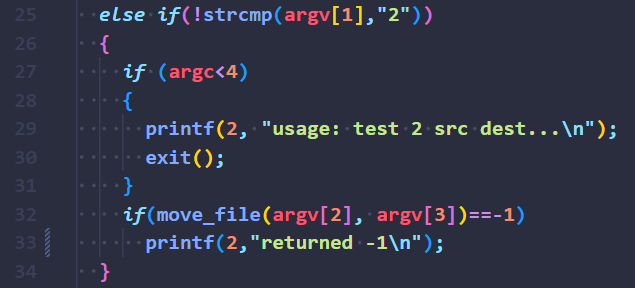
کد در فایل sysfile.c:



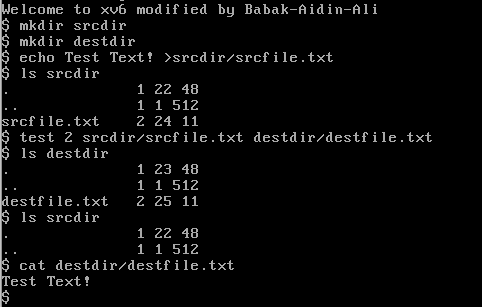




حال این سیستم‌کال را در برنامه سطح کاربر به کار می‌بریم:



و تست می‌کنیم:



.2 پیاده سازی فراخوانی سیستمی مرتب سازی فراخوانی های یک پردازه:

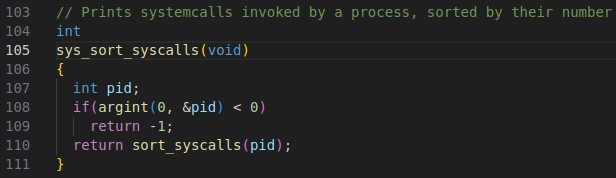
اضافه کردن به usys.s: 

اضافه کردن به syscall.h: 

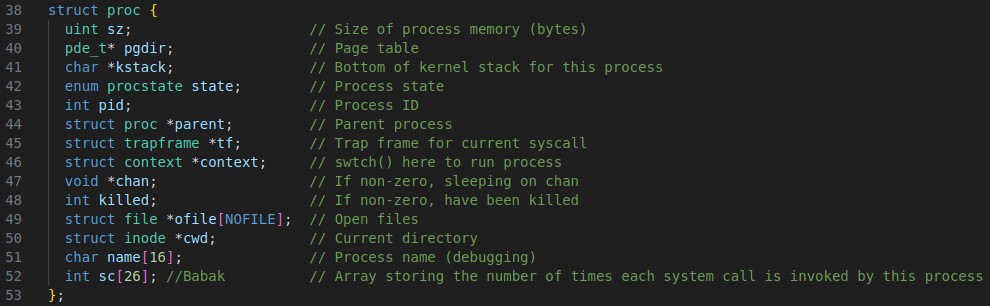
اضافه کردن به syscall.c: 



مشابه سیستم‌کال پالیندروم، این سیستم‌کال هم به sys\_proc.c اضافه می‌کنیم ولی در اینجا میخواهی آرگومان را از استک بخوانیم پس با استفاده از argint شماره پردازه را دریافت می‌کنیم.

اضافه کردن به sys\_proc.c: 

برای پیاده‌سازی این بخش نیاز بود ساختار داده‌ای در struct پردازه اضافه کنیم. ما آرایه 26 تایی در فایل proc.h به ساختارداده پردازه اضافه کردیم که تعداد استفاده از هر سیستم‌کال توسط این پردازه را در آن نگه می‌داریم. در اصل بدین صورت از روش counting sort استفاده کردیم و sort با پیچیدگی بهینه o(n) انجام می‌شود.

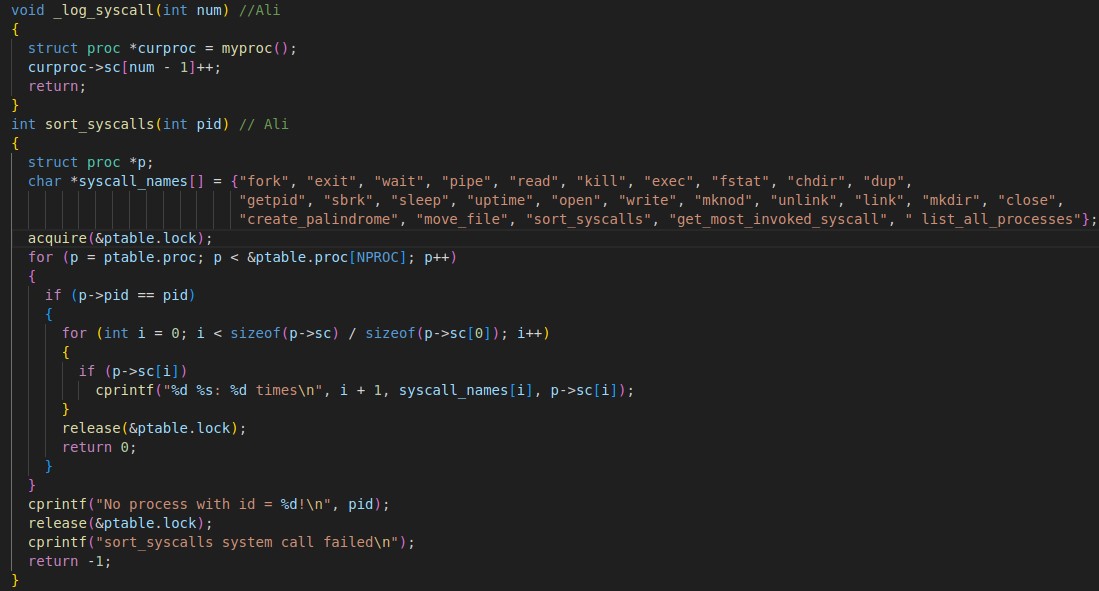


دو جا در فایل proc.c تمام خانه‌های آرایه sc را صفر می‌کنیم. یک بار در تابع allocproc که باعث می‌شود وقتی پردازه جدید ایجاد می‌شود، مقادیر آرایه sc آن صفر باشد و بار دیگر در تابع wait که اگر پردازه‌ای در استیت zombie یافت شد، تمام متغیرهای آن از جمله sc صفر می‌شود.

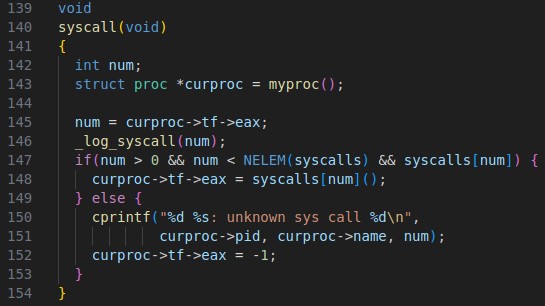


تابع sort\_syscalls تمام پردازه‌ها را بررسی می‌کند تا پردازه مورد نظر را پیدا کند و سپس به ازای هر سیستم‌کالی که آن پردازه فراخوانده، تعداد دفعات فراخوانی را چاپ می‌کند.

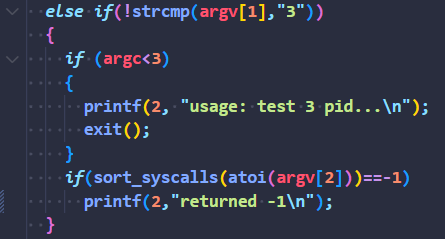
اضافه کردن به proc.c:



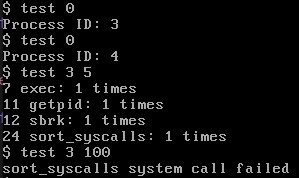
تابع \_log\_syscall یکی به تعداد فراخوانی سیستم‌کال مشخصی در پردازه فعلی اضافه می‌کند. وقتی که پس از فراخوانی سیستمی به تابع syscalls رفتیم این تابع را صدا می‌کنیم.



برنامه سطح کاربر را تغییر می‌دهیم تا از این سیستم‌کال استفاده کنیم.



تست سیستم‌کال:



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

اضافه کردن به usys.s: 

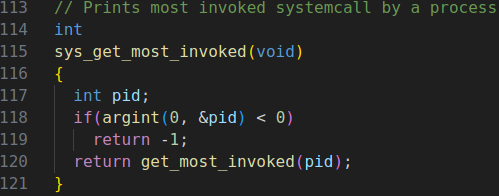
اضافه کردن به syscall.h: 

اضافه کردن به syscall.c: 



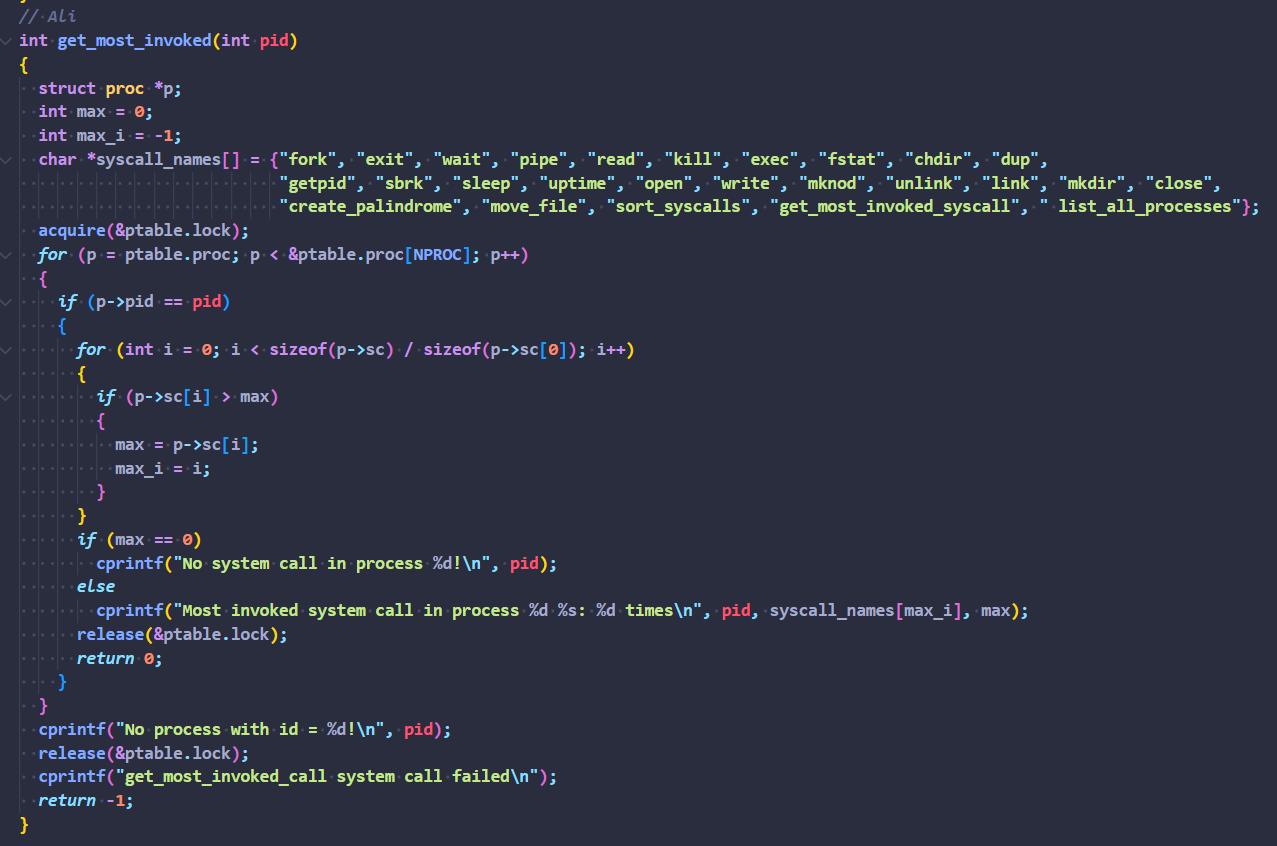
مشابه بخش قبل به sys\_proc.c اضافه کرده و ورودی را دریافت می‌کنیم.

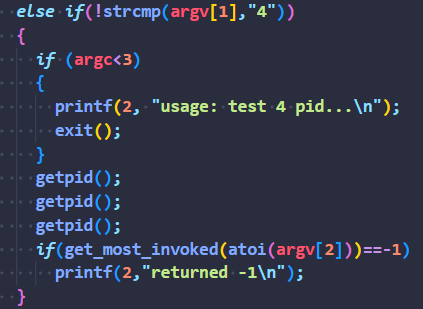
اضافه کردن به sys\_proc.c:

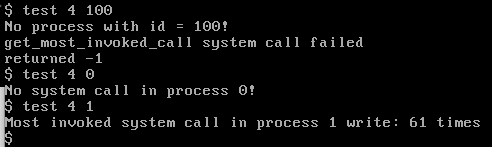


تابع get\_most\_invoked تمام پردازه‌ها را بررسی می‌کند تا پردازه مورد نظر را پیدا کند؛ سپس تعداد همه سیستم‌کالهایی که آن پردازه فراخوانده بررسی کرده تا بیشترین را بیابد، در نهایت فراخوانی مورد نظر را با تعداد دفعات فراخوانی ها را چاپ می‌کند. همچنین این تابع در صورت پیدا نشدن پردازه مورد نظر و یا فراخوانی نشدن سیستم‌کالی از آن پردازه شرایط را به کاربر اطلاع می‌دهد.

اضافه کردن به proc.c:



برنامه سطح کاربر را تغییر می‌دهیم تا از این سیستم‌کال استفاده کنیم.

تست سیستم‌کال:

4. پیاده سازی فراخوانی سیستمی لیست کردن پردازه ها:

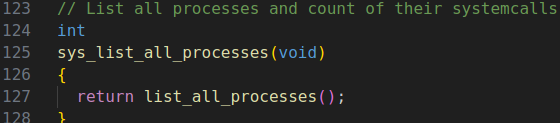
اضافه کردن به usys.s: 

اضافه کردن به syscall.h: 

اضافه کردن به syscall.c: 

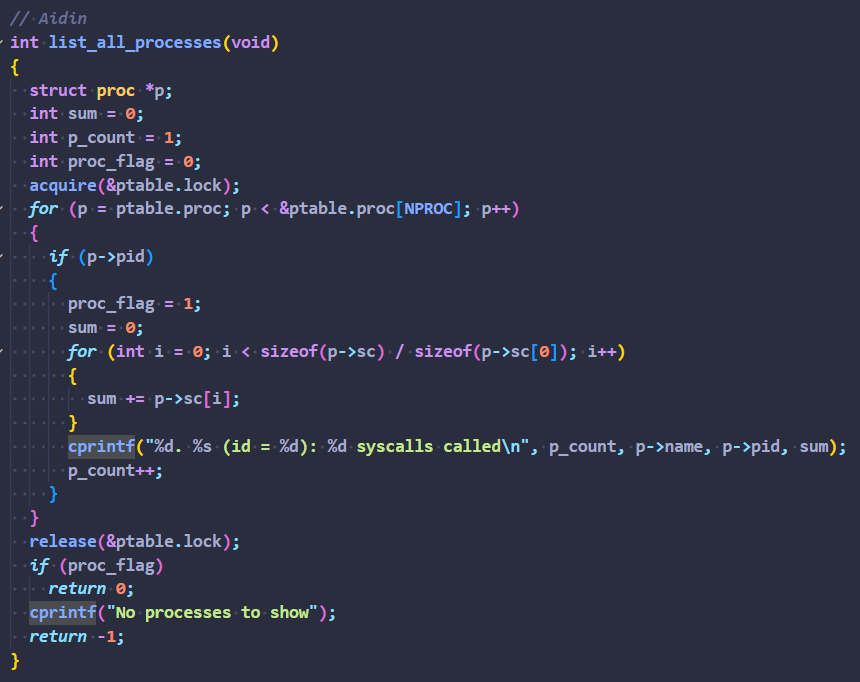


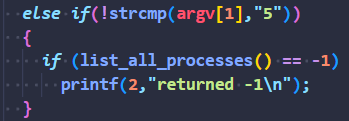
اضافه کردن به sys\_proc.c:



تابع list\_all\_processes برای تمام پردازه‌های موجود تعداد همه سیستم‌کالهایی که آن پردازه فراخوانده را جمع کرده، سپس تعداد دفعات فراخوانی ها را به همراه مشخصات پردازه چاپ می‌کند.

اضافه کردن به proc.c:



برنامه سطح کاربر را تغییر می‌دهیم تا از این سیستم‌کال استفاده کنیم.

تست سیستم‌کال:

