**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؟**

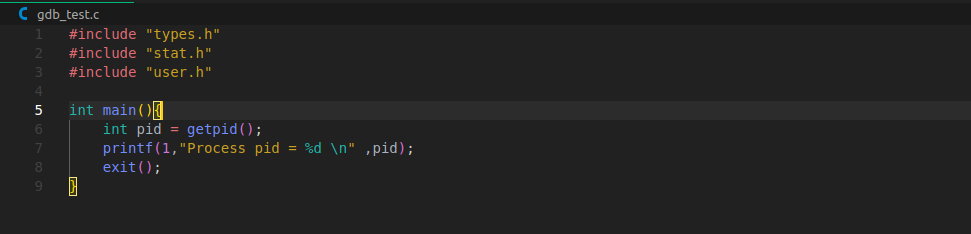
دو پشته ی user stack و kernel stack موجود هستند . موقع فعال شدن یک trap ، میخوخاهیم دسترسی را تغییر دهیم (مثلا از سطح کاربر به سطح هسته برویم ) ، در این زمان نمیتوانیم از پشته قبلی استفاده کنیم . پس باید ss و esp روی پشته push شوند تا هنگام بازگشت آخرین دستوری که اجرا شده را بدانیم و انجام دستورات را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

**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 .

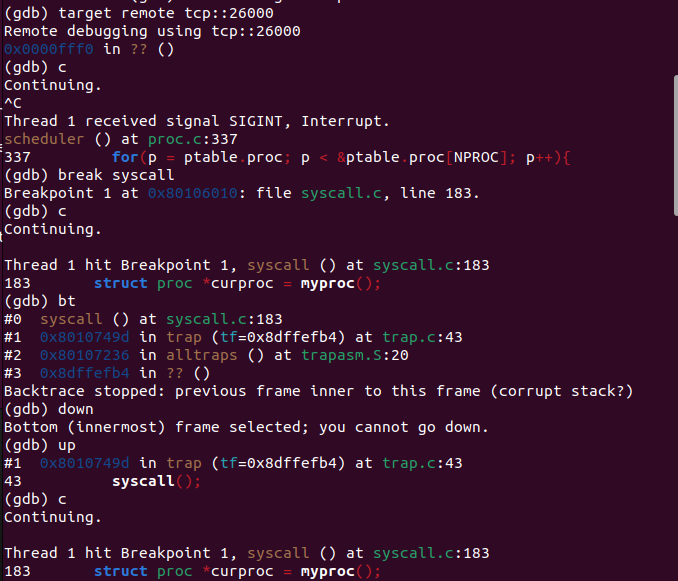
خروجی bt در هر خط یک frame stack را نشان می دهد که به ترتیب از درونی ترین frame هر یک را بیان می کنیم:

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 .

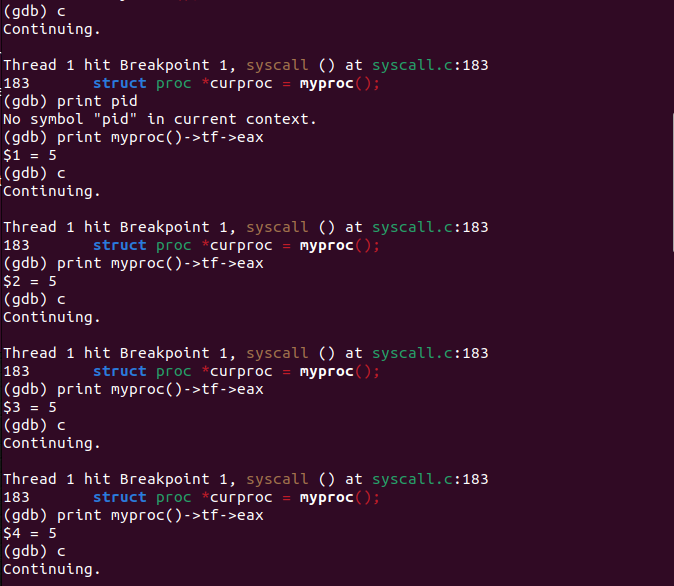


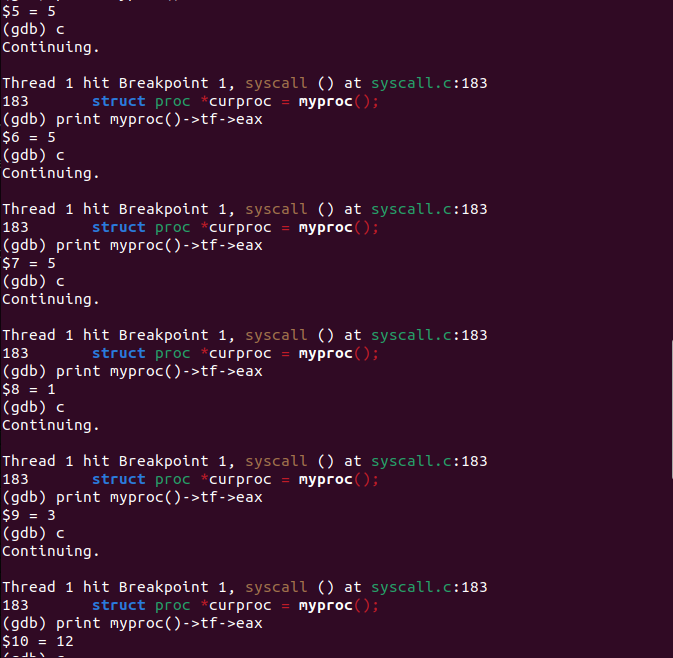
سپس چند باری c و سپس print myproc()->tf->eax را مینویسیم . (چاپ محتوای eax که در tf است )

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 . )

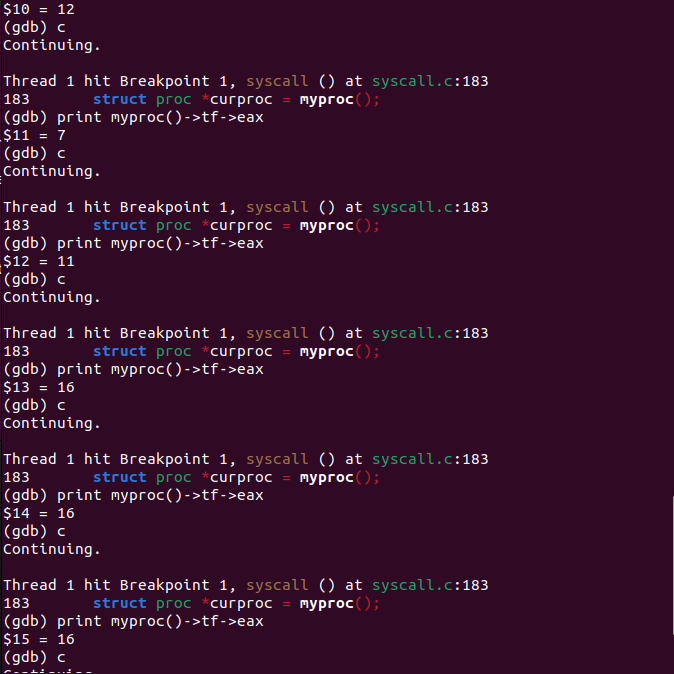




عدد 1 مربوط به سیستم کال fork است که جهت ایجاد پردازه جدید برای برنامه سطح کاربر صدا زده میشود.

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.

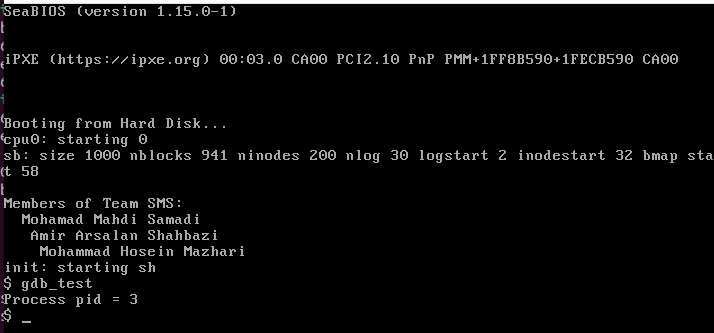


عدد 7 مربوط به سیستم کال exec است که برای اجرای برنامه pid در پردازه ایجاد شده استفاده میشود .

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**

****

**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 میکنیم

user.h

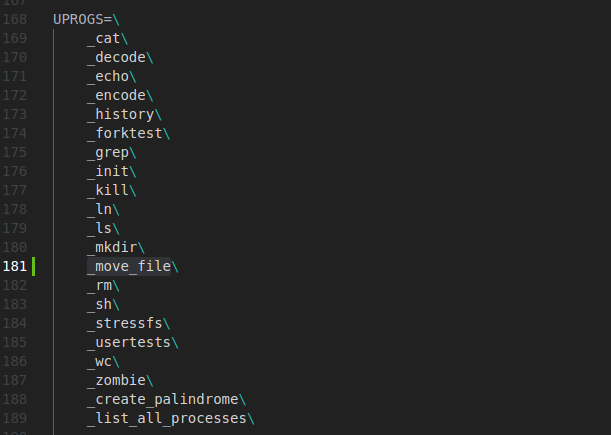


تابع را در دسترس کاربر قرار می دهیم .

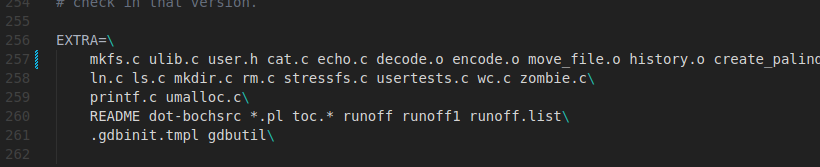
**defs.h**

****

**Makefile(UPROGS)**

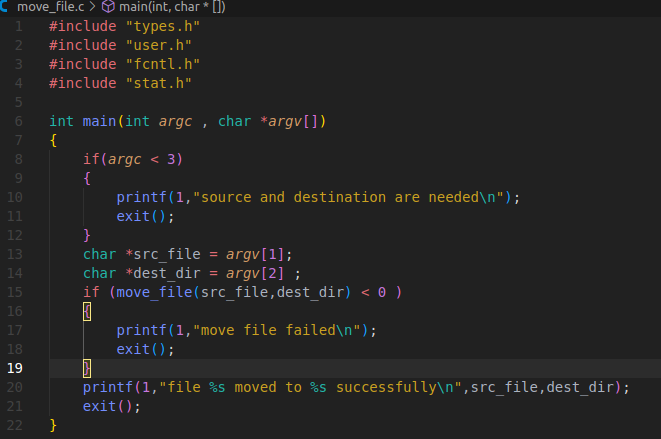
****

**Makefile(EXTRA)**

****

**move\_file.c**

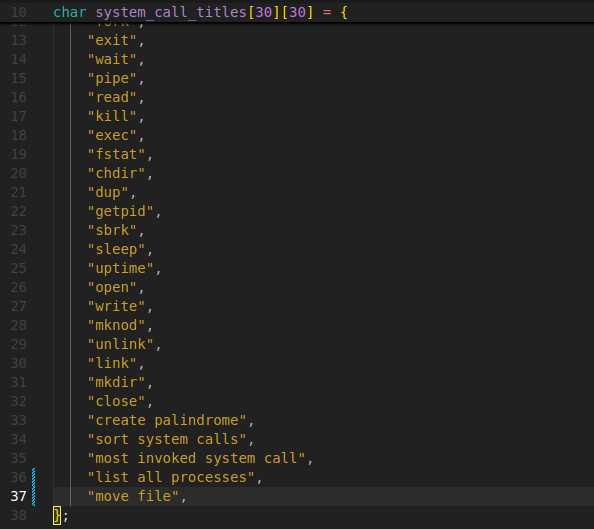
تست با برنامه ی سطح کاربر :

****

**syscall.c**

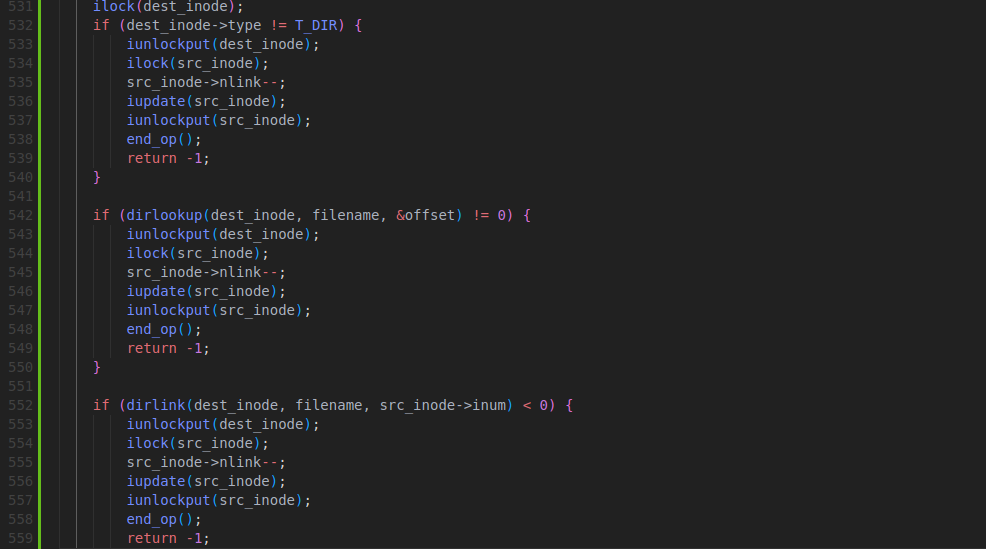
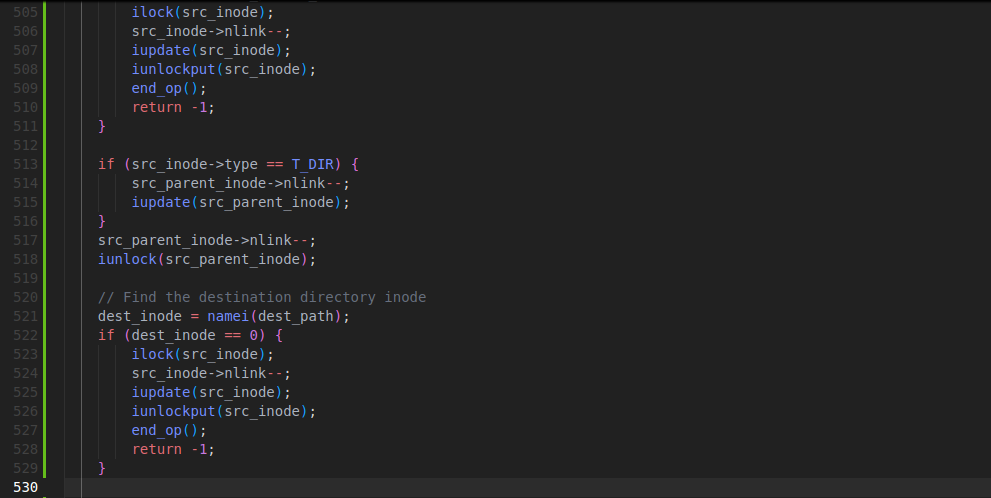
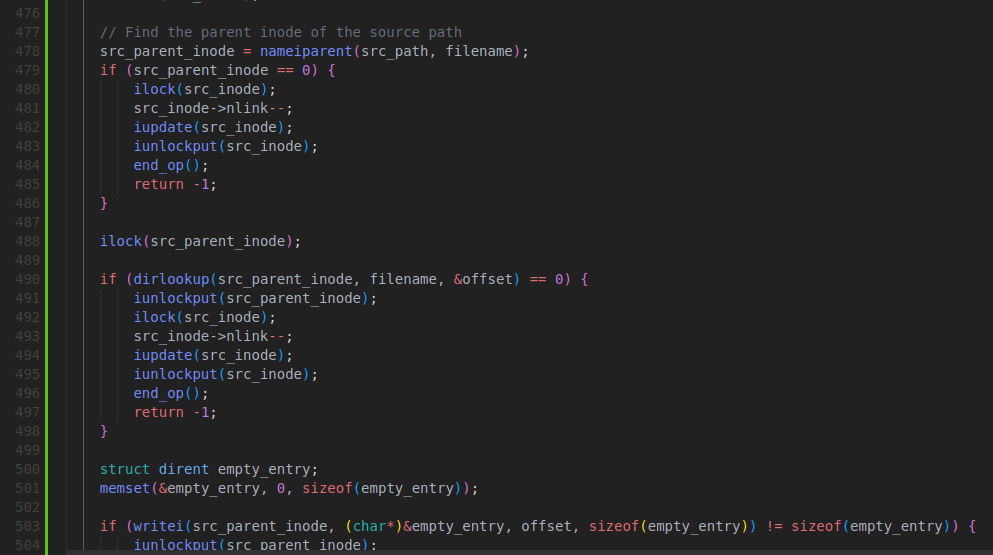
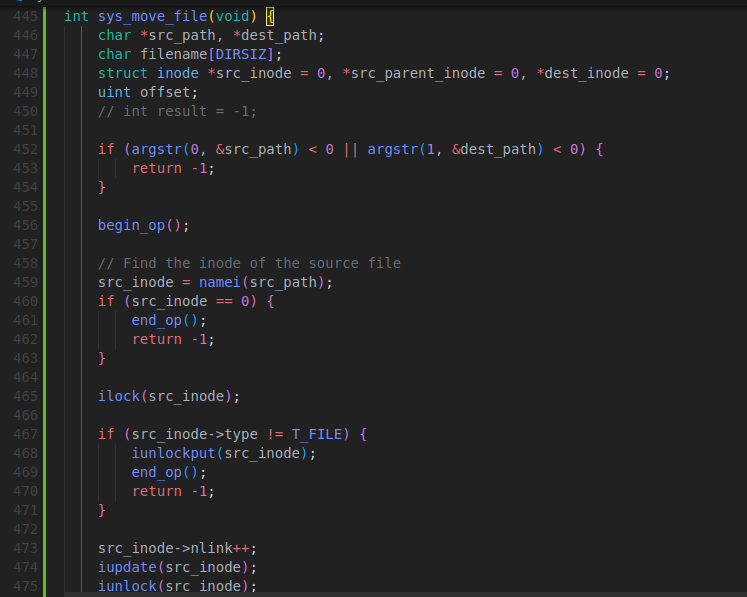
****

****

****

**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

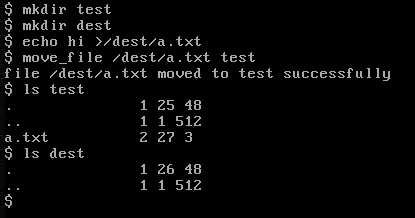
SYS\_move\_file عدد سیستم کال است که در syscall.h تعریف میشود.

**syscall.h**

****

**نتیجه**

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

****