**آزمایشگاه سیستم عامل**

پروژه شماره 2

علی پادیاو – کسری حاجی حیدری – اولدوز نیساری

بهار 1402

Repository Link: <https://github.com/alumpish/OS-Lab-Projects>

Latest Commit Hash: 609bc6b3ca3756d057c115085606bcf7a3a5464a

**مقدمه**

1-

اگر در فایل‌های تشکیل دهنده متغیر ULIB دقت کنیم متوجه میشویم از فایل‌های زیر تشکیل شده است: ulib.o, usys.o, printf.o, umalloc.o که به تشریح آن‌ها می‌پردازیم:

ulib:

در ulib.c در تعدادی از توابع مثل strcpy,strlen,strcmp ,atoi از فراخوانی سیستمی استفاده نشده است، اما در تعدادی دیگر از توابع استفاده شده است:

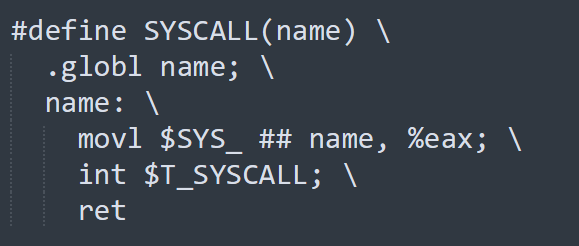
در stat از فراخوانی‌های سیستمی open , close برای باز و بسته کردن فایل‌ها استفاده می‌شود، هم چنین از فراخوانی سیستمی fsat برای به دست آوردن اطلاعات آن فایل استفاده می‌شود.

در gets از فراخوانی سیستمی read برای گرفتن ورودی استفاده شده است.

در memset از فراخوانی سیستمی stosb برای پر کردن حافظه استفاده شده است.

usys:

در این فایل اسم سیستم کال‌ها به صورت گلوبال نوشته شده است، آیدی‌های آن‌ها نیز در رجیستر eax نوشته شده است. در زمان فراخوانی هر یک از سیستم کال‌ها به define ای که در این فایل وجود دارد مراجعه می‌شود:



در این فایل همچنین لیستی از سیستم کال‌ها موجود است.

printf:

در تابع putc از فراخوانی سیستمی write برای نوشتن یک حرف کاراکتر در یک fd استفاده می‌شود.

umalloc:

در تابع morecore از فراخوانی سیستمی sbrk برای تغییر اندازه دیتا سگمنت و افزایش حافظه استفاده می‌شود.

2-

روش‌های دیگر دسترسی سطح کاربر به هسته لینوکس:

Pseudo-file-systems:

در حقیقت فایل‌هایی هستند که فایل‌های حقیقی ندارند بلکه تعدادی ورودی مجازی دارند که خود سیستم فایل آن‌ها را نگه می‌دارد. چون در حقیقیت یک رابط به هسته به ما می‌دهند نیاز دارند که به کرنل دسترسی داشته باشند.

Exception‌ها:

به طور کلی برای رفع خطاهایی که در interrupt‌های سخت افزاری یا نرم افزاری به وجود می‌آید دسترسی به کرنل انجام شود تا خطا‌ها بر طرف شوند.

( exception‌ها نوعی از interrupt‌های نرم افزاری است)

Socket :

برنامه‌های سطح کاربر که بر مبنای سوکت نوشته می‌شوند با قابلیت اتصال به سوکت و جا به جا کردن داده‌ها می‌توانند به کرنل دسترسی پیدا کنند.

3-

خیر. اگر باقی تله‌ها با همین حالت فعال شوند، در آن صورت دسترسی به هسته بسیار آسان می‌شود و امنیت دیگر تضمین نیست. به همین خاطر اگر کاربر سعی کند تله ای دیگر را فعال کند، xv6 جلوگیری می‌کند تا امنیت به خطر نیفتند و امکان سو استفاده و آسیب از بین برود.

**سازوکار اجرای فراخوانی سیستمی درxv6**

**بخش سخت افزاری واسمبلی**

4-

هنگامی که یک تله فعال میشود، سطح دسترسی از سطح دسترسی کاربر به هسته تغییر میکند و از پشته ی مختص به هسته استفاده میکند.(هر کدام از این دو حالت پشته مختص به خود را دارند) بعد از اینکه رسیدگی به تله به اتمام رسید باید به پشته کاربر بازگردیم. بدین منظور از دو رجسیتر esp و ss استفاده میکنیم که به پشته در حال استفاده اشاره میکند.برای اینکار زمانی که تله فعال میشود مقادیر این دو رجیستر که به پشته کاربر اشاره دارند را در پشته هسته ذخیره میکنیم و بعد از اتمام رسیدگی به تله مقادیر آن را بازیابی کرده و به پشته کاربر بازمیگردیم و برنامه کاربر از جای قبلی ادامه خواهد یافت.در صورتی که نیاز به تغییر سطح کاربر نباشد.

از آن جایی که همچنان با پشته قبلی کار میکنیم نیازی به ذخیره مقادیر از دو رجیستر نیست.

**بخش سطح بالا وکنترل کننده زبان سی تله**

5-

چهارتابع برای دسترسی به پارامترهای فراخوانی سیستمی تعریف شده که عبارتند از argint,argptr,argstr,argfd)) که در ادامه هر یک توضیح داده شده است:

argint: این تابع از رجیستر esp استفاده میکند تا n امین پارامتر را پیدا کند. این رجیستر به انتهای پشته و آدرس برگشت اشاره میکند و پارانتر های فراخوانی سیستمی بالای آن قرار دارند پس با فرمول زیر میتوان به nامین پارامتر دست یافت: esp + 4 + 4 \* n

پس از دست‌یابی به این پارامتر، آن را به همراه پوینتری به حافظه \*ip)) به تابع fetchint ارسال میشود و در صورت تایید ادرس فرستاده شده(در حافظه پردازه باشد) آن را در حافظه داده شده ذخیره می‌کند.

argptr: این تابع ابتدا با استفاده از تابع argint آدرس پوینتر مورد نظر را چک میکند و در صورتی که معتبر باشد آن را دخیره میکند.

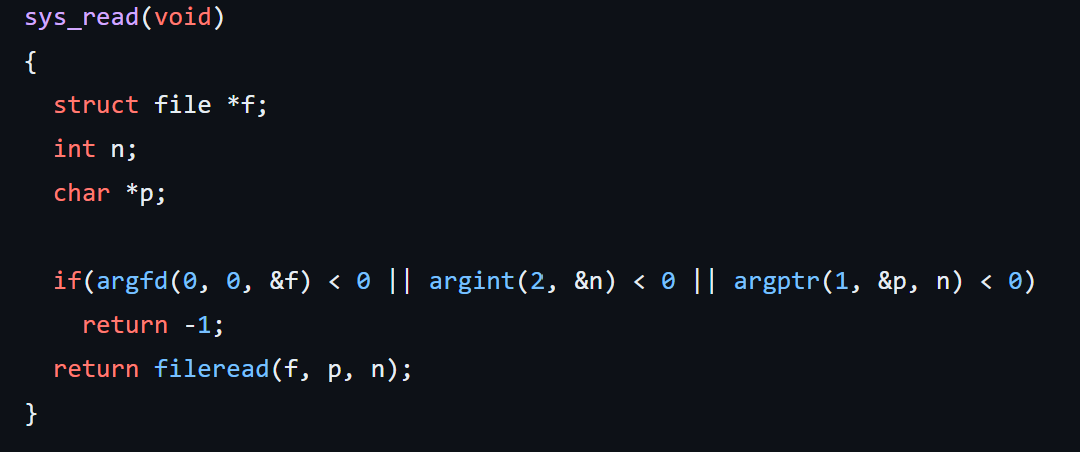
argstr: با استفاده از تابع argint ابتدای رشته را مشخص میکند و به تابع fetchstr میدهد و این تابع بررسی میکند که این آدرس در حافظه پردازه باشد و پس از تایید آن را ذخیره می‌کند.

argfd: با استفاده از تابع argint عدد filedescriptor را میگیرد و آن را چک میکند و اگر صحیح بود فایل متناظر با آن را بازمیگرداند.

تمامی این توابع بررسی می کنند که آدرس داده شده حتما در حافظه پردازه قرار گیرد که یک پردازه نتواند به حافظه پردازه دیگری دسترسی پیدا کند و از مشکلات امنیتی جلوگیری میکند، در نهایت در صورت وجود مشکل -1 را برمیگرداند.

تابع sys\_read به صورت زیر است که مربوط به تابع read() میباشد:

read(int fd, void\* buffer, int max)

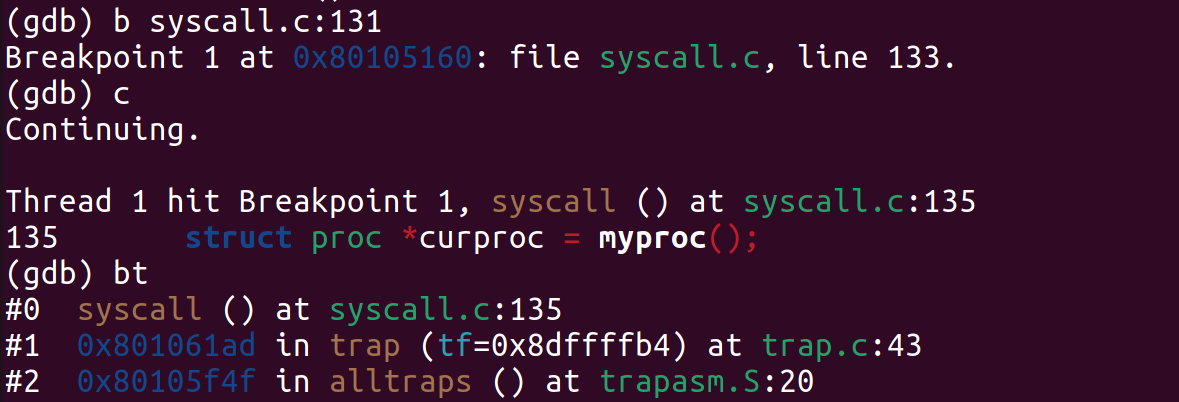


در ابتدا این تابع با استفاده از تابع argfd مقدار fd را دریافت میکند و درستی آن را بررسی میکند. در ادامه با استفاده از تابع argint پارامتر سوم را دریافت و در نهایت با استفاده از تابع argptrپارامتر دوم را دریافت و بررسی میکند که بافر داده شده از ابتدا تا اندازه ی max از حافظه ی پردازه باشد.

اگر این بررسی ها صورت نمیگرفت ممکن بود به علت عدم تطابق بافر و مقدار max هنگام خواندن و نوشتن در بافر ممکن بود از حافظه پردازه خارج شویم و به مشکل بربخوریم.

**بررسی گام‌های فراخوانی سیستمی در سطح کرنل توسط GDB:**

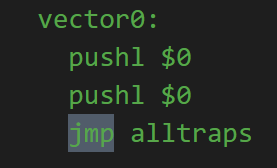
ابتدا بر روی خط 131 که ابتدای تابع syscall در فایل syscall.c است بریک پوینت قرار می‌دهیم. سپس اجرای برنامه را ادامه میدهیم تا روند اجرا در برخورد با بریک پوینت متوقف شود. در آن زمان از دستور bt استفاده می‌کنیم تا سابقه فراخوانی‌ها را ببینیم.

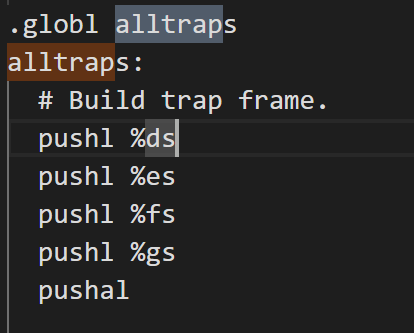


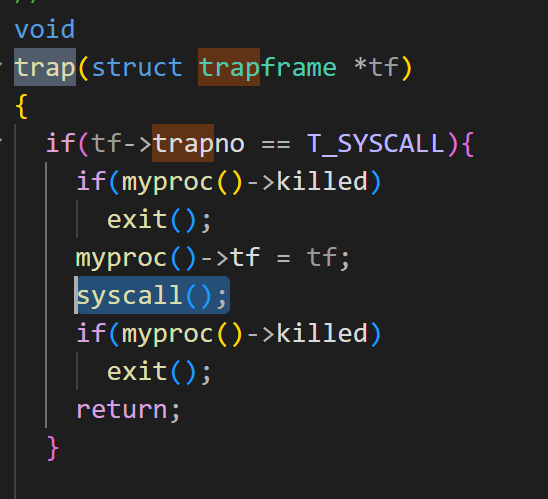
دستور bt مخفف backtrace است و زمانی که میخواهیم متوجه شویم که چه توابعی فراخوانی شده اند و به استک اضافه شده اند از آن استفاده می‌کنیم. این دستور فراخوانی‌ها را به ترتیب از درونی ترین فریم شروع می‌کند.

در خصوص توضیحات خروجی مان اتفاقی که می‌افتد به این صورت است که مطابق چیزی که در فایل usys.h که ابتدا شماره سیستم کال بر طبق چیزی که در syscall.h به آن اختصاص داده شده است در رجیستر eax نوشته می‌شود و سپس int $T\_SYSCALL; اجرا می‌شود. برای جستجو تعریف این خط وارد فایل vector.s میشود و در آنجا تابع alltraps که در trapasm.s است فراخوانی می‌شود و درون آن تابع هم traps که در trap.c است فراخوانی می‌شود.

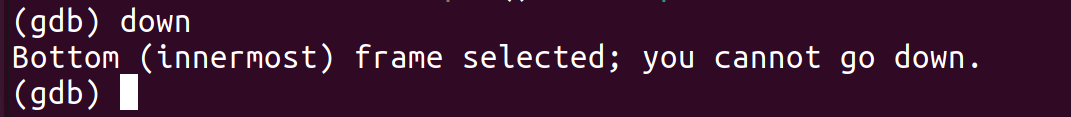
همان طور که می‌بینیم این روند فراخوانی از درونی ترین تا بیرونی ترین در عکس خروجی ما قابل دیدن است.





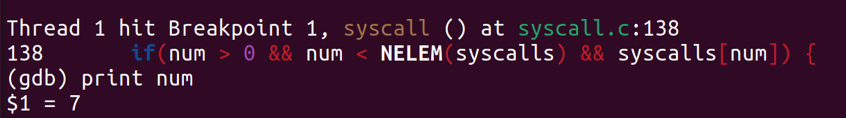


حال دستور down را که کار آن جا به کردن به سمت پایین استک است را اجرا می‌کنیم.

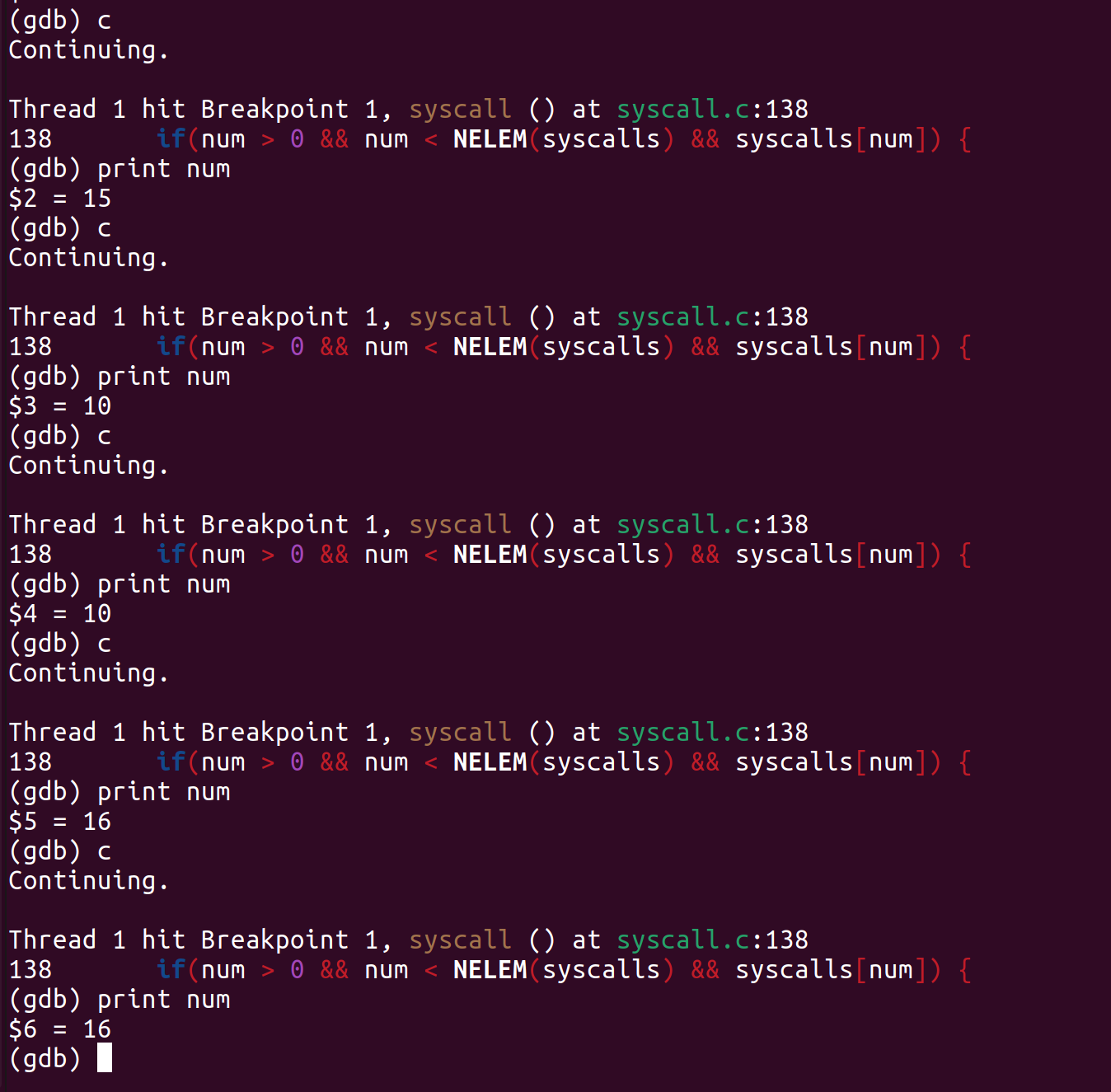


همان طور که مشاهده می‌کنیم چون در درونی ترین فریم هستیم، دستور اجرا نمی‌شود.

با قرار دادن یه بریک پوینت روی خط 138 مقدار eax را به دست می‌آوریم:



که همان طور که مشاهده می‌کنیم با مقدار get\_pid که 16 است برابر نیست. دلیل هم آن است که پیش از فراخوانی get\_pid فراخوانی‌های دیگری صدا زده می‌شوند. در نهایت هم با چند بار continue کردن مقدار رجیستر را می‌بینیم.



**ارسال آرگومان‌های فراخوانی‌های سیستمی**

برای اضافه کردن این فراخوانی سیستمی، در ابتدا تابع در دسترس کاربر را در user.h دیکلر می کنیم:

int find\_fibonacci\_number(void)

سپس تعریف این تابع را در usys.sانجام می دهیم:

SYSCALL(find\_fibonacci\_number)

حال باید عدد سیستم را در syscall.h اضافه کنیم:

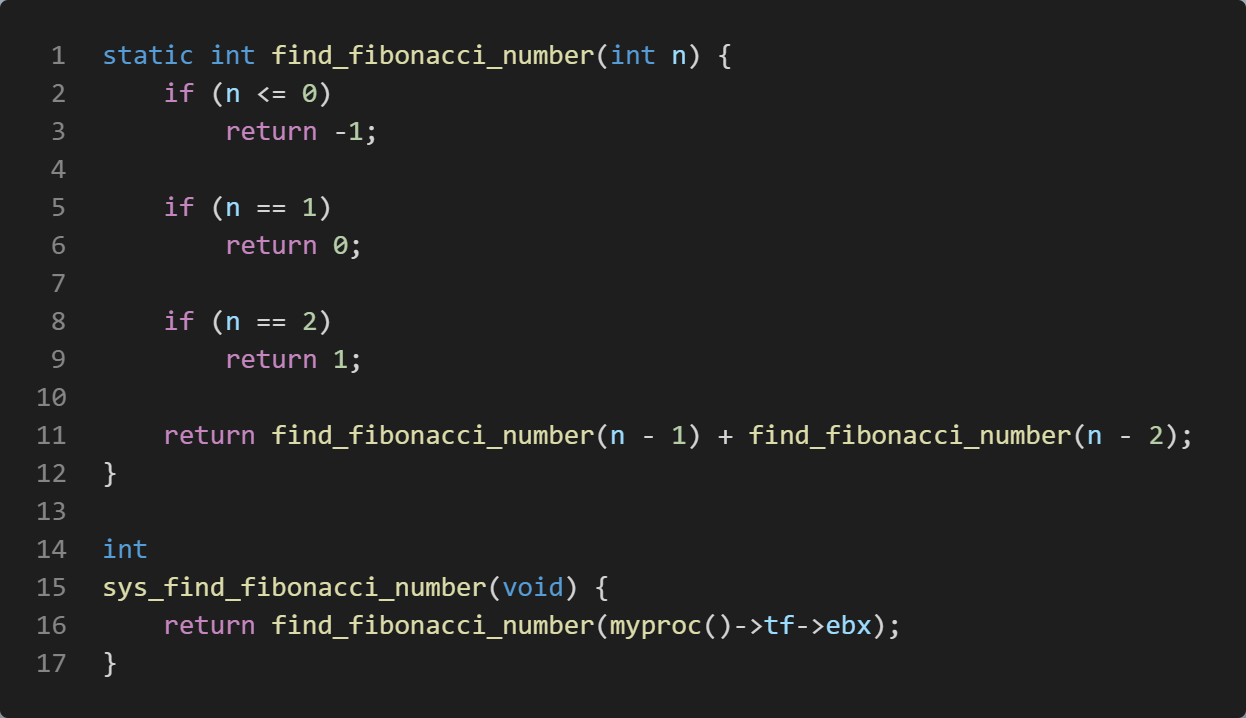
#define SYS\_find\_fibonacci\_number 26

حال درsyscall.c دیکلر تابع را قرار میدهیم و سپس آن را به آرایه مپ شماره سیستم کال به تابع اضافه می کنیم:

extern int sys\_find\_fibonacci\_number(void)

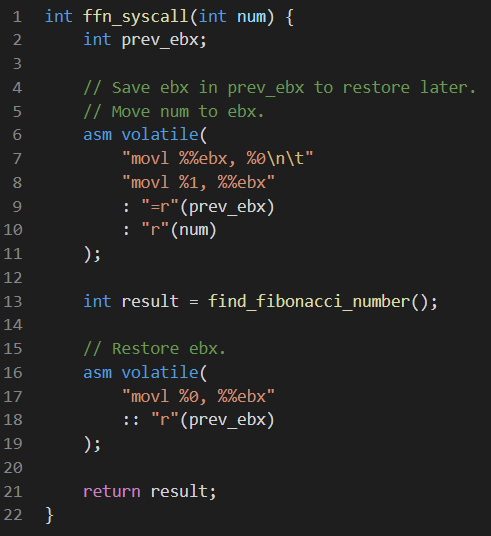
[SYS\_find\_fibonacci\_number] sys\_find\_fibonacci\_number

تعریف تابع را میتوان در یکی از فایل های sysproc.c یا sysfile.c قرار داد اما از آنجا که این تابع از نظر عملکردی ربطی به این دو فایل ندارد فایل جدیدی میسازیم و فایل تابع را داخل آن قرار میدهیم. sysutils.c))

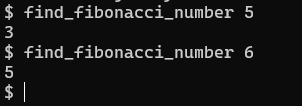


از آنجا که فایل جدیدی ساختیم باید آن را به متغیر OBJS اول Makefile اضافه کنیم. برای اجرای این فراخوانی سیستمی، یک برنامه سطح کاربر می سازیم. فایل find\_fibonacci\_number.c را ساخته و \_find\_fibonacci\_number را به متغیر UPROGS در Makefile اضافه می کنیم.

باید به صورت دستی آرگومان که عدد مورد نظر است را به رجیستر ebx بریزیم. برای این کار در ابتدا، مقدار کنونی رجیستر ebx را در متغیری ذخیره کرده و مقدار آرگومان را در آن میریزیم. سپس سیستم کال را انجام می دهیم و مقدار رجیستر را به حالت قبلی اش بر می گردانیم.



خروجی به صورت زیر است.

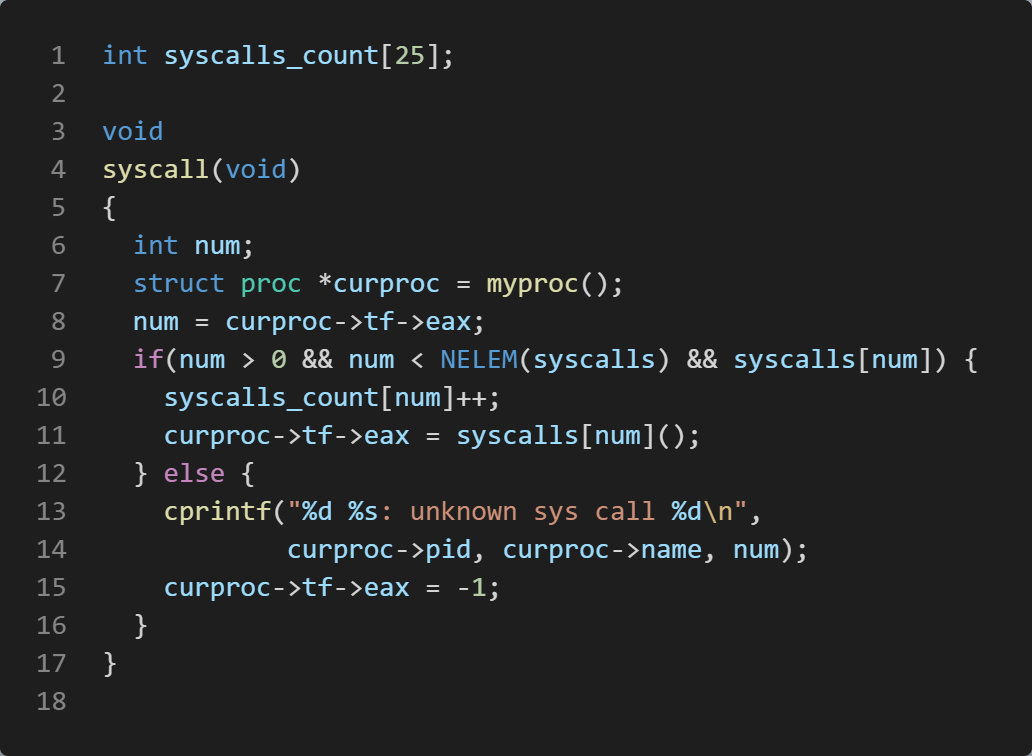


**پیاده‌سازی فراخوانی‌های سیستمی**

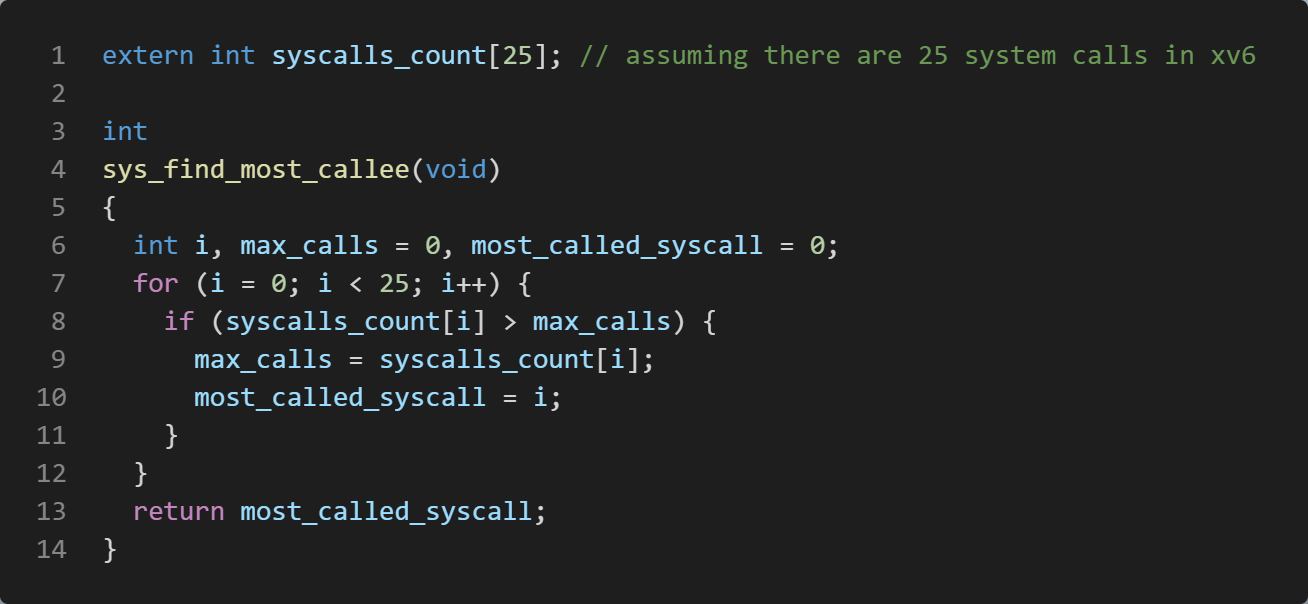
نحوه اضافه کردن systemcall ها مثل سوال قبل است. بنابراین از توضیحات تکراری صرف نظر می‌کنیم.‌

1. int find\_most\_callee(void)

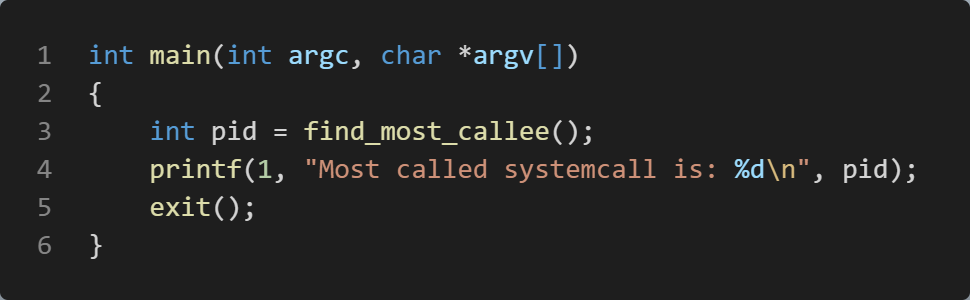
در فایل syscall.c آرایه‌ای برای شمارش سیستم‌کال‌ها اضافه کردیم و هربار که سیستم‌کالی فراخوانی می‌شود، این آرایه را آپدیت می‌کنیم.



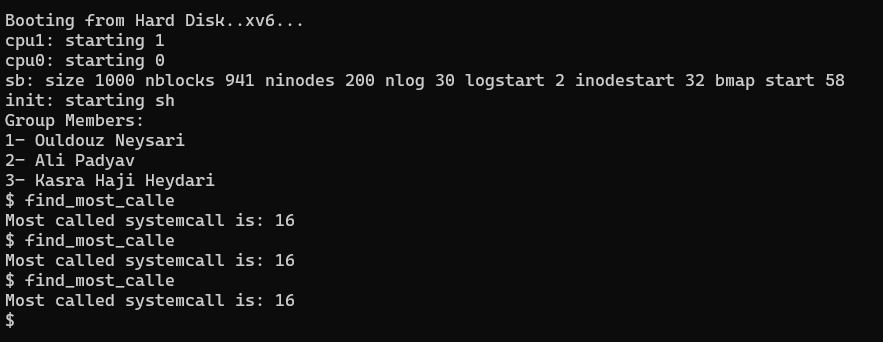
در فایل sysproc.c هم به صورت زیر عمل خواسته شده را انجام می‌دهیم.



برنامه سطح کاربر find\_most\_calle.c هم نوشتیم تا سیستم‌کال را تست کنیم.



خروجی به صورت زیر است.



سیستم کال شماره 16 برای write است. علت این نتیجه این است که برای پرینت هر حرف در کنسول یکبار این سیستم‌کال فراخوانی میشود، برای همین در شروع کار بیشتر از همه استفاده شده است.

1. int get\_children\_count(void)

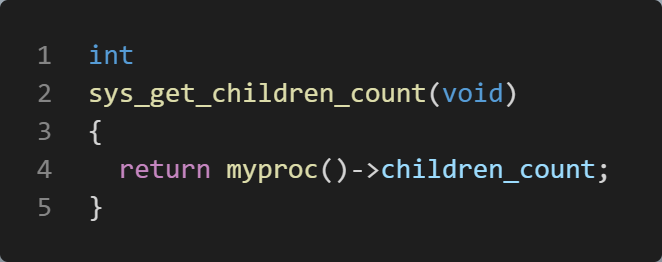
به استراکت proc در proc.h فیلد children\_count را اضافه میکنیم.

  int children\_count;          // Count of childern

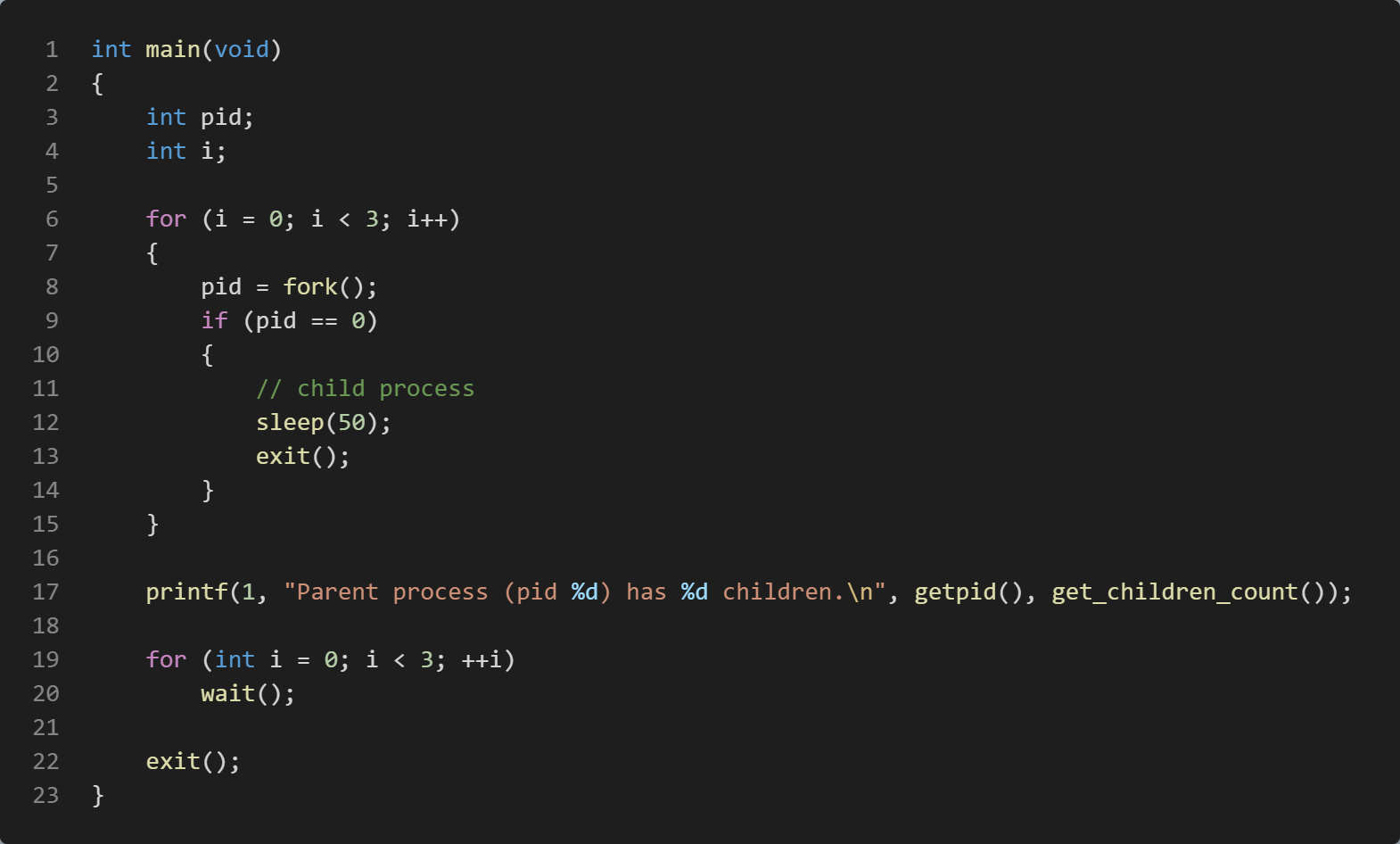
در فایل proc.c و در تابع allocproc که یک پردازه را میسازد مقدار children\_count را برابر 0 قرار میدهیم.

  p->children\_count = 0

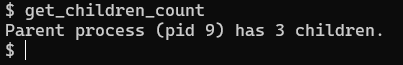
در فایل sysproc.c هم به صورت زیر عمل خواسته شده را انجام می‌دهیم.



برنامه سطح کاربر get\_children\_count.c را نوشته و سیستم‌کال را تست می‌کنیم.



خروجی به صورت زیر است.

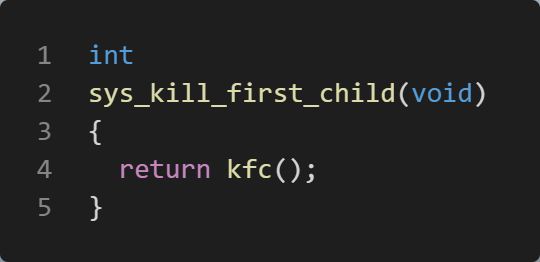


1. int kill\_first\_child(void)

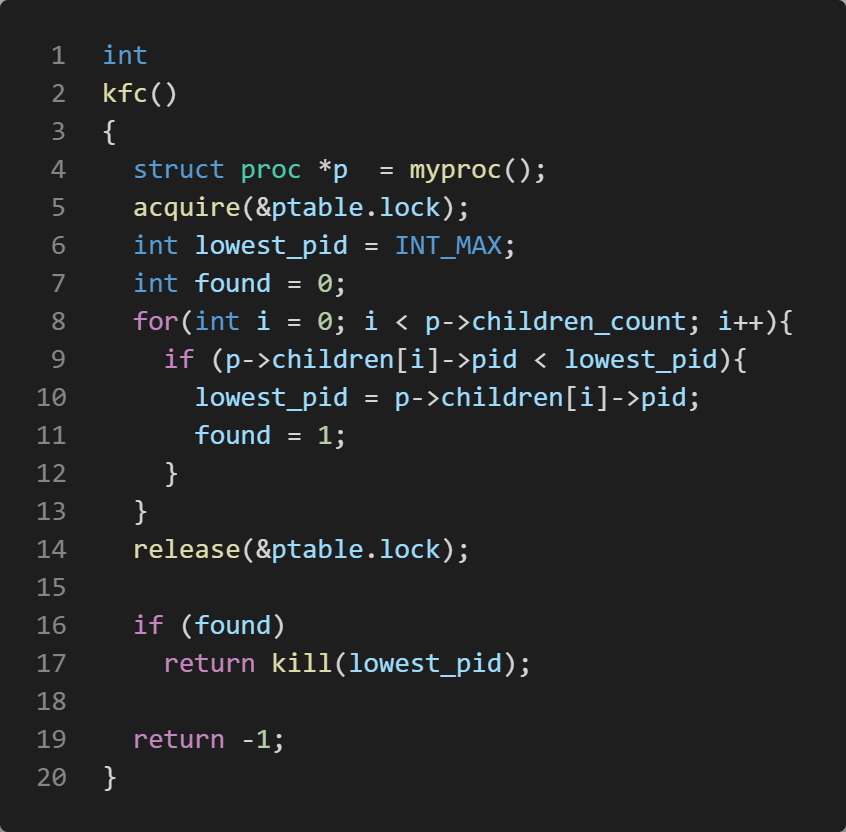
ابتدا به استراکت proc فیلد فرزندان پردازه را اضافه می‌‌کنیم.

  struct proc \*children[NPROC];   // array of child processes

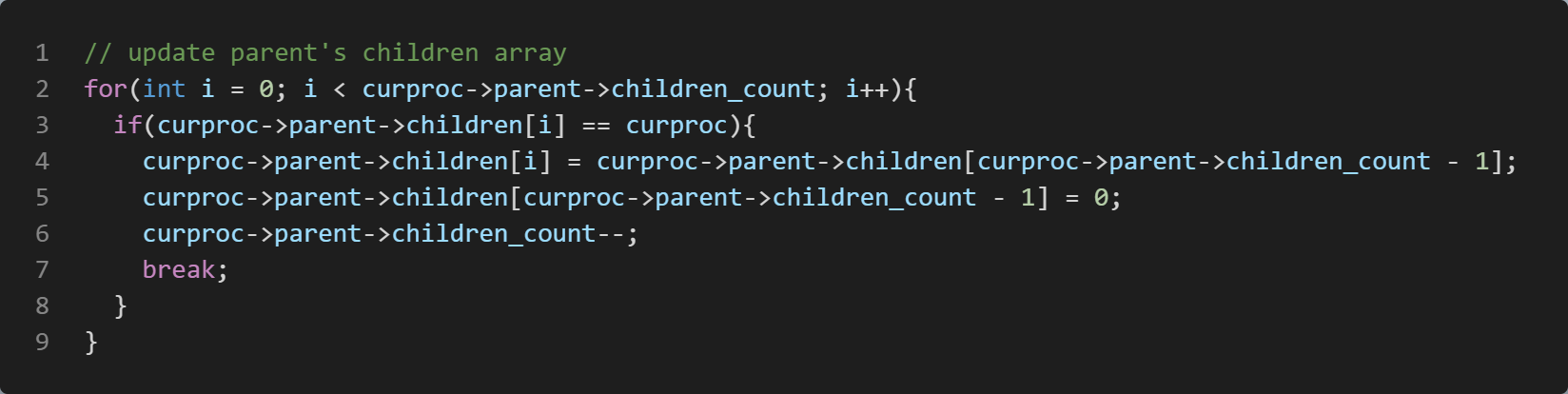
در فایل sysproc.c تابع kfc که در proc.c هست را کال می‌کنیم.



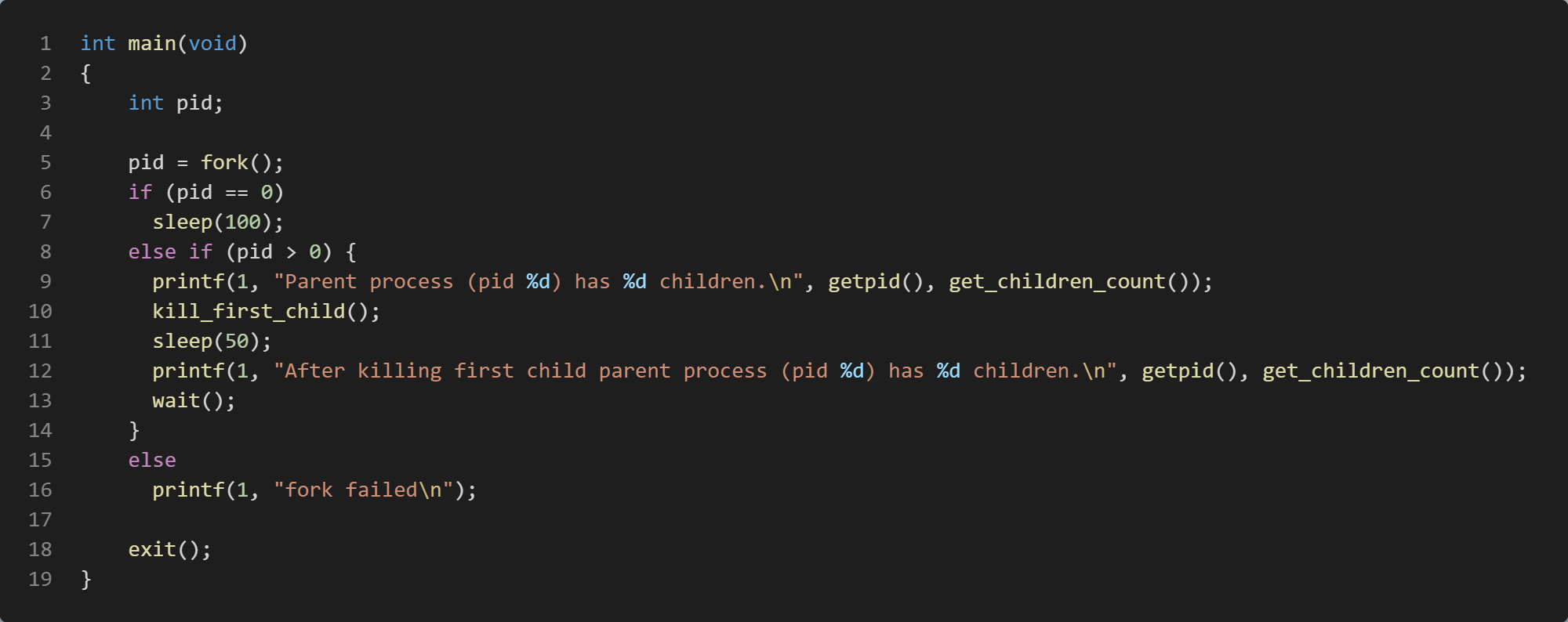
در این تابع بین فرزندان process فعلی می‌گردیم و آنیکه کمترین pid دارد را kill می‌کنیم.



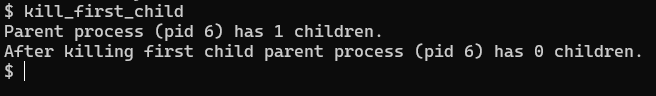
در تابع exit که در proc.c هست هم تکه کد زیر را اضافه کردیم تا وقتی process ای به پایان می‌رسد، آرایه فرزندانِ پدر این process را آپدیت می‌کنیم.



در نهایت با برنامه سطح کاربر kill\_first\_child.c سیستم‌کال را تست می‌کنیم.



خروجی به صورت زیر است.



در پردازه پدر ابتدا kill\_first\_child() را کال می‌کنیم. برای اینکه نتیجه را بتوان نشان داد اندکی صبر می‌کنیم و بعد دوباره children\_count را نشان می‌دهیم. چون kill کردن یک پردازه مقداری زمان می‌خواهد.