

گزارش آزمایشگاه سیستم عامل: پروژه شماره ۲

گروه ۲۲

پردیس زندکریمی - محمد امانلو - شهزاد ممیز
۸۱۰۱۰۱۰۸۱ - ۸۱۰۱۰۰۰۸۴ - ۸۱۰۱۰۰۲۷۲

Last commit code:

6d5dcf46c3900834051e226bdef9a3d361f0600a

Github repo:

https://github.com/MohammadAmanlou/OS_lab

۱. کتابخانه های (قاعدتاً سطح کاربر، منظور فایل های تشکیل دهنده متغیر **ULIB** در **Makefile** است) استفاده شده در **6xv** را از منظر استفاده از فراخوانی های سیستمی و علت این استفاده را بررسی نمایید.

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

همان طور که مشاهده می شود. متغیر **ULIB** در **makefile** از 4 object به نام های **ulib.o, usys.o, printf.o, umalloc.o** تشکیل شده است.

• ulib.o:

برای تشخیص اینکه این object دارای system call هست یا نه بایستی به source آن مراجعه کنیم، که در اینجا **ulib.c** را داریم. در **ulib.c** توابع زیر تعریف شده اند.

strcpy, strcmp, strlen, memset, strchr, gets, stat, atoi, memmove

از بین این توابع در **stat, gets** از فراخوانی های سیستمی استفاده شده است.

• **gets**: در این تابع از فراخوانی سیستمی **read** استفاده شده است.

```
cc=read(0, &c)
```

همان طور که می دانیم فراخوانی سیستمی **read** یک خط را از **STDIN** می خواند و آن را در **c** ذخیره می کند.

این خط در یک حلقه تکرار می‌شود و در هر مرحله یک خط از `stdin` می‌خواند.

- **stat**: در این تابع از فراخوانی های سیستمی `open` و `fstat` و `close` استفاده شده است.

در این تابع ابتدا با استفاده از `open` ، یک فایل باز می‌شود سپس با استفاده از `fstat` اطلاعات فایل مورد نظر را بدست می‌آوریم و در انتها با استفاده از `close` آن فایل را می‌بندیم.

- **usys.o**

سورس این فایل، فایل `usys.S` می باشد که این فایل شامل پوشاننده های سیستم کال ها، به زبان اسمبلی می باشد. در این فایل یک تابع به نام `SYSCALL(system)` وجود دارد که ابتدا مقدار `SYS_name` را در رجیستر `eax` می ریزد و سپس یک وقفه با کد `T_SYSCALL` تولید می کند.

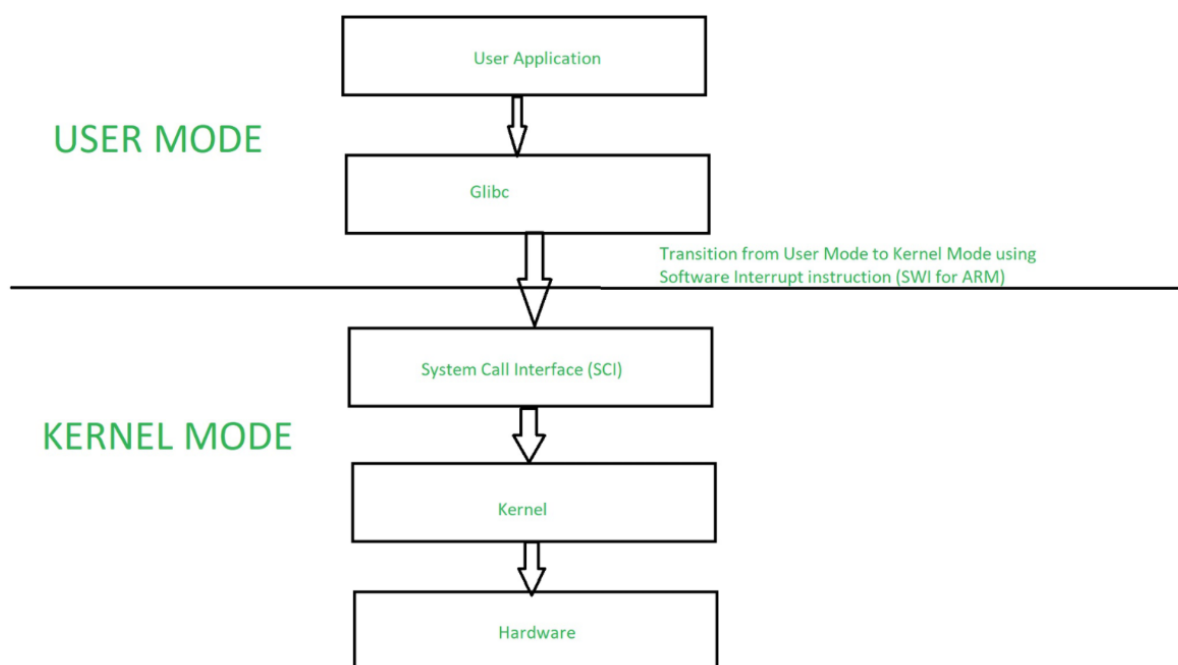
- **printf.o**

سورس این فایل، فایل `printf.c` می باشد. در این فایل 3 تابع `putc` و `printinit` و `printf` تعریف شده اند که در تابع `putc` از سیستم کال `write` استفاده شده است در واقع با استفاده از آن در فایل با `fd` مورد نظر می نویسد. در تو تابع دیگر از تابع `putc` استفاده شده است که از سیستم کال مورد نظر استفاده می کند.

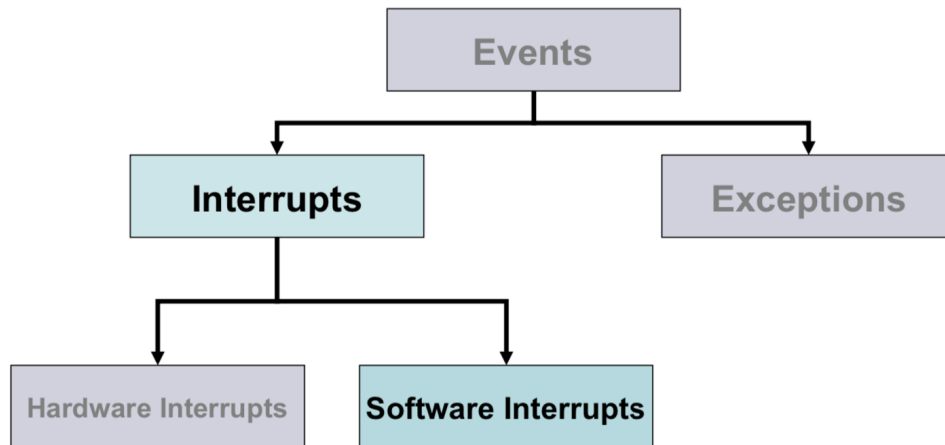
- **umalloc.o**

سورس این فایل `umalloc.c` می باشد. در این فایل توابع `malloc` و `free` و `morecore` تعریف شده اند. تابع `free` یک پوینتر را به عنوان ورودی می گیرد که این پوینتر به یک آدرس از حافظه اشاره می کند و سپس این بخش از حافظه را آزاد می کند و در آن از سیستم کالی استفاده نشده است. تابع `malloc` برای تخصیص حافظه ی پویا با اندازه ی مشخص استفاده می شود که مقدار خروجی آن یک اشاره گر `void` به حافظه ی تخصیص داده شده است. و در آن نیز از سیستم کالی استفاده نشده است. اما در تابع `morecore` از سیستم کال `sbrk` استفاده شده است که فضای پردازش را افزایش میدهد.

2. دقت شود فراخوانی های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روشها را در لینوکس به اختصار توضیح دهید.



به صورت کلی event ها سبب می شوند از سطح کاربر به کرنل برویم.



Event ها شامل interrupt و exception است.
 همچنین خود interrupt ها به دو دسته ی hardware interrupt و software interrupt تقسیم بندی می شوند.

همچنین دسترسی کاربر به هسته از روشهای زیر انجام می شود:

- Command line interface
- Batch interface
- Graghical user interface
- Touch screen interface

وقفه سخت افزاری: یک وقفه سخت افزاری توسط دستگاه های سخت افزاری خارجی برای درخواست توجه از CPU ایجاد می شود. این وقفه ها توسط رویدادها یا شرایط مختلف مانند ورودی کاربر، خطاهای سخت افزاری یا تکمیل عملیات I/O ایجاد می شوند. هنگامی که یک وقفه سخت افزاری رخ می دهد، CPU اجرای فعلی خود را به حالت تعلیق در می آورد و کنترل را به یک کنترل کننده وقفه از پیش تعریف شده منتقل می کند. سپس کنترل کننده وقفه اقدامات لازم را برای مدیریت وقفه انجام می دهد، مانند سرویس دستگاه یا انجام وظایف خاص مرتبط با وقفه.

وقفه نرم افزاری: وقفه نرم افزاری که به عنوان trap نیز شناخته می شود، یک وقفه عمدی است که توسط یک برنامه در حال اجرا یا توسط خود سیستم عامل ایجاد می شود. وقفه های نرم افزاری معمولاً برای درخواست خدمات از سیستم عامل یا انجام عملیات ممتازی که نیاز به انتقال به حالت هسته دارند استفاده می شوند. به عنوان مثال، یک برنامه ممکن است یک وقفه نرم

افزایی را برای درخواست عملیات ورودی/خروجی، تخصیص حافظه به صورت پویا، یا خاتمه اجرا راه اندازی کند. هنگامی که یک وقفه نرم افزاری رخ می دهد، CPU کنترل را به یک کنترل کننده وقفه یا روال خاصی که مسئول رسیدگی به سرویس درخواستی است، منتقل می کند.

به طور خلاصه، وقفه های سخت افزاری توسط دستگاه های سخت افزاری خارجی برای جلب توجه ایجاد می شوند، در حالی که وقفه های نرم افزاری عمداً توسط برنامه ها یا سیستم عامل CPU برای درخواست خدمات خاص یا انجام عملیات ممتاز ایجاد می شوند. هر دو نوع وقفه نقش مهمی در عملکرد کلی یک سیستم کامپیوتری دارند

همچنین سطح کاربر برای تعامل با هسته می تواند از File System Interface، Library API و Network Interface استفاده کند.

به غیر از روش system call، روش signal و سوکت نیز وجود دارند.

- روش سیگنال:

سیگنال یک پیام معمولاً کوتاه است که ممکن است به یک فرایند یا گروهی از فرایندها ارسال شوند. سیگنالهای استاندارد ارگومانها یا سایر اطلاعات را منتقل نمیکنند. تنها اطلاعات عددی است که سیگنال را شناسایی می کند. برای آگاه کردن یک فرایند از وقوع یک رویداد خاص استفاده می شود.

- روش سوکت:

برنامه هایی که در سطح کاربر هستند میتوانند از طریق سوکت از روی یک پورت میتوانند گوش دهند و پیام ردوبدل کنند.

3. آیا باقی تله ها را نمیتوان با سطح دسترسی DPL USER فعال نمود؟ چرا؟

خیر، این امر امکان پذیر نیست. همان طور که می دانیم در xv6 از 3 سطح کلی سیستم عامل ها سطح 0 و 3 که به ترتیب سطح kernel و user هستند موجود است. چون که سطح دسترسی USER_DPL یک سطح دسترسی کاربر یا در واقع همان سطح دسترسی 3 است، نباید امکان دسترسی به هسته و اجرای این تله ها را داشته باشد. اگر کاربر امکان اجرای این تله ها را داشت به راحتی می توانست به کرنل دسترسی داشته باشد. که با اصول protect کردن بخش kernel از بخش user در تناقض است.

در واقع اگر یک پردازنده بخواهد یک interrupt دیگری را فعال کند، kernel به آن اجازه این کار را نمی دهد چون که این موارد protected است. (به وکتور شماره 13 می رود) زیرا

ممکن است در برنامه سطح کاربر نیز یک مشکل وجود داشته باشد که آن به کل هسته تسری یابد یا اصلاً خود کاربر قصد حمله به هسته را داشته و با این روش می تواند به هسته حمله کرده و با در دسترس داشتن تمام دسترسی های `kernel mode` بتواند به هسته سیستم عامل آسیب بزند و هر بخشی را که از سخت افزار یا نرم افزار سیستم می خواهد مورد تهاجم قرار دهد.

4. در صورت تغییر سطح دسترسی، `ss` و `esp` روی پشته `Push` میشود. در غیراینصورت `Push` نمیشود. چرا؟

به طور کلی دو پشته داریم `user stack` و `kernel stack`. هنگامی که یک `trap` فعال شود و می خواهیم دسترسی را تغییر دهیم مثلاً از سطح کاربر به سطح کرنل برویم نمی توانیم از پشته قبل استفاده کنیم. بنابراین باید `ss` و `esp` روی پشته `push` شوند تا هنگام بازگشت بدانیم که آخرین دسترسی که انجام داده ایم چه بوده است و اطلاعات از دست نروند زیرا داشتن این اطلاعات ضروری است و بتوانیم ادامه روند اجرای دستورات را از سر بگیریم. ولی وقتی تغییر سطح دسترسی نداشته باشیم، نیازی به `push` آنها نیست زیرا همچنان با همان پشته کار میکنیم.

5. در مورد توابع دسترسی به پارامتر های فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در `Argptr()` بازه آدرس ها بررسی می گردد؟ تجاوز از بازه معتبر چه مشکل امنیتی ایجاد می کند؟ در صورت عدم بررسی بازه ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی `sys_read()` اجرای سیستم را با مشکل روبرو سازد.

توابع `argptr` و `argstr` و `argint` برای دسترسی به پارامتر های فراخوانی سیستمی تعریف شده اند. این توابع در صورت آرگومان غیر مجاز، مقدار -1 را برمی گردانند. `argint()`: این تابع دو ورودی دارد. ورودی اول، شماره پارامتر و ورودی دوم که به صورت `pass by reference` داده میشود یک متغیر با تایپ `int` است، که در واقع مقدار پارامتر در ورودی دوم ذخیره می شود. این تابع با صدا زدن تابع `fetchinit()` و دادن مقدار $(myproc()->tf->esp) + 4 + 4*n$ به عنوان ورودی اول آن، آدرس پارامتر خواسته شده را ساخته و محتوای آن را برمی گرداند. اگر این هدف با موفقیت انجام شود مقدار صفر و اگر این پارامتر وجود نداشته باشد و عملیات با موفقیت انجام نشود (آرگومان غیر مجاز) خروجی -1 برگردانده می شود.

`argptr()`: این تابع سه ورودی دارد. ورودی اول، شماره پارامتر خواسته شده است و ورودی دوم به صورت `pass by reference` و سائز پارامتر به عنوان ورودی سوم به تابع داده می شود که ورودی دوم محتوای پارامترهایی به شکل `pointer` مانند آرایه را در ورودی دوم ذخیره می کند. این تابع با صدا زدن تابع `argint()` آدرس خانه اول این پارامتر از جنس اشاره

گر را بر می گرداند. صورتی که این تابع با موفقیت عملیات را انجام دهد مقدار صفر و در غیر این صورت مقدار ۱- را برمی گرداند.

argstr(): به آن می‌گوییم چندمین آرگومان و این تابع پس از بررسی صحت، آن را در یک اشاره گر به **char *** قرار می‌دهد (رشته باید با 0 تمام شود یا **nul-terminate** باشد).

argfd(): این تابع در فایل **sysfile.c** تعریف شده و سه ورودی دارد. ورودی اول شماره پارامتر است. ورودی دوم آن اشاره گری به **file descriptor** و ورودی سوم آن اشاره گری به آدرس ساختمان داده **file** است که پس از انجام عملیات تابع مقدار دهی می‌شوند. این تابع در صورتی که با موفقیت عملیات را انجام دهد مقدار صفر و در صورت شکست مقدار ۱- را بر می‌گرداند.

در فراخوانی سیستمی **sys_read** اگر این بازه ها چک نشوند برای مثال ممکن است محتوای فایلی که میخوانیم در قسمت نادرستی که مخصوص این پردازش نیست ذخیره شود و بنابراین هم در انجام عملیات پردازش فعلی به مشکل میخوریم و هم ممکن است در ایجاد عملیات دیگر پردازش ها ایجاد مشکل کند و بخشی از اطلاعات حافظه از دست برود و یا در **trap** بیفتیم.

بررسی گامهای اجرای فراخوانی سیستمی در سطح کرنل توسط **gdb**

پس از **clean** کردن با استفاده از **make** سپس دستور **make qemu-gdb** را اجرا می‌کنیم. پس از آنکه به طور کامل فایل ها کامپایل شدند در ترمینال دیگری **gdb** را اجرا میکنیم. سپس یک برنامه سطح کاربر که از **getpid** استفاده می‌کند را به نام **gdb_test** ایجاد میکنیم. محتویات این فایل میتواند مشابه کد زیر باشد:

```
C gdb_test.c > main(int, char *[])
1  #include "types.h"
2  #include "user.h"
3
4  int main(int argc , char * argv[]) {
5      int pid = getpid();
6      printf(1 , "Process pid = %d \n" , pid);
7      exit();
8
9  }
```

سپس در فایل `syscall.c` یک breakpoint در ابتدای تابع `syscall` ایجاد میکنیم. که می تواند مشابه دستورات زیر صورت بپذیرد.

```
type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel...
warning: File "/home/mohammad/OS_lab/OS_lab/xv6-public/.gdbinit" auto-loading has been declined by your 'auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
  add-auto-load-safe-path /home/mohammad/OS_lab/OS_lab/xv6-public/.gdbinit
line to your configuration file "/home/mohammad/.gdbinit".
To completely disable this security protection add
  set auto-load safe-path /
line to your configuration file "/home/mohammad/.gdbinit".
For more information about this security protection see the
--Type <RET> for more, q to quit, c to continue without paging--
"Auto-loading safe path" section in the GDB manual. E.g., run from the shell:
  info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
xv6:fire in ? ()
(gdb) b syscall.c:144
Breakpoint 1 at 0x801050c2: file syscall.c, line 145.
(gdb) r
The "remote" target does not support "run". Try "help target" or "continue".
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145      num = curproc->tf->eax;
(gdb) █
```

سپس می بایست در بخش دستورات `gdb` دستور `bt` را وارد کنیم. با وارد کردن این دستور لیستی از تمام فراخوانی های تابع که به نقطه فعلی اجرا رسیده اند را نمایش می دهد.

```
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x80105aad in trap (tf=0x8dffffb4) at trap.c:43
#2  0x8010584f in alltraps () at trapasm.S:20
(gdb) █
```

پس از صدا زده شدن این دستور محتویات `call stack` نمایش داده می شود. `Call stack` در واقع یک `stack` است که برای ذخیره سازی سیر اجرای برنامه برای اینکه بتوانیم سیر اجرای درست برنامه را پیگیری کنیم ایجاد میشود. در واقع سیر توابع صدا زده شده را در خود دارد. وقتی یک تابع صدا زده می شود در واقع یک بلوک حافظه به نام `stack frame` ایجاد می شود. که در آن اطلاعات تابع مانند متغیر های محلی، آدرس بازگشت و ... موجود است. وقتی تابع فراخوانی می شود این `stack frame` ایجاد شده در بالای `call stack` اضافه می شود. با زدن دستور `bt` محتویات `call stack` نمایش داده می شود. همان طور که در تصویر بالا قابل مشاهده است، معمولاً در اجرای `bt` محتویاتی مانند نام توابع، نام آرگومان ها، آدرس آن در `source` و در `memory` نمایش داده می شود. برای اینکه بفهمیم چرا وقتی در تابع `syscall` از breakpoint استفاده می شود خروجی مطابق تصویر بالا می شود ابتدا باید بدانیم، در `xv6` مکانیزم تعریف و اجرای `system call` ها به چه نحوی است.

1. برای هر سیستم کال یک عدد و شناسه در نظر گرفته می شود که در فایل های `syscall.h` و `user.h` قرار دارند. و هر سیستم کال در واقع به این شناسه شناخته می شود. در صورتی که

یک سیستم کال به برنامه بخواهیم اضافه کنیم باید ابتدا آن را در لیست سیستم کال ها اضافه کنیم. و به آن شناسه مربوطه را تخصیص دهیم.

2. سیستم کال ها به زبان اسمبلی تعریف می شوند که در فایل `usys.s` قرار دارند. (در واقع اینجا منظور تعریف خود سیستم کال در ارتباط با سخت افزار است)

3. در این فایل (`usys.s`) در `xv6` رجیستر `eax` معمولاً برای ذخیره مقدار بازگشتی یک سیستم کال استفاده میشود. هنگامی که یک سیستم کال فراخوانی می شود، معمولاً شماره مورد نظر سیستم کال قبل از اجرای دستور `int 64` در رجیستر `eax` قرار میگیرد. پس از اجرای سیستم کال، مقدار یا وضعیت حاصل عموماً در رجیستر `eax` ذخیره میشود تا در پردازش یا بازیابی بیشتر توسط کد فراخواننده استفاده شود. و در نهایت `int64` صدا می شود.

4. با اجرای دستور `int 64` در مرحله قبل، وارد بخش `vector64` میشویم که بعد از `push` شدن مقدار آن در فایل `vector.s` به `alltraps` پرش (`jump`) می کنیم. که در فایل `trapsasm.s` قرار دارد. حال `alltraps` فعالیت خود را آغاز می کند.

5. `Alltraps` ابتدا `trap frame` را می سازد و آن را در استک `push` می کند. و سپس تابع `trap` را که در فایل `trap.c` قرار دارد صدا می زند. که این تابع با عدد 64 متوجه ایجاد یک `interrupt` می شود. که در واقع یعنی یک سیستم کال به وجود آمده است.

6. سپس تابع `trap`، آن `trap frame` را که از قبل در استک پوش شده بود بعنوان `trap frame` برای `process` فعلی قرار می دهد و تابع `syscall` را صدا میکند. که با `breakpoint` مواجه می شود.

7. همان طور که در تصویر نیز قابل مشاهده است هنوز تابع `syscall` کامل اجرا نشده است و از طریق `trap` و `alltrap` صدا زده شده است که به ترتیب قرار دارند.

عملیات بعدی که باید در این بخش انجام شود این است که دستور `down` را در بخش خط فرمان `gdb` اجرا کنیم. این دستور ما را به `stack frame` بالا تر در `bt` می برد در واقع ما را تابعی که دیرتر صدا زده شده یا همان `callee` می برد. که در اینجا چون `syscall` تابعی را صدا زده است پس `call` `stack` دارای هیچ `stack frame` در این جهت نیست.

```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb)
```

دستور بعدی دستور `up` است که دقیقاً مخالف دستور `down` است و اینجا چون به یک دستور عقب تر می رویم به دستور `trap` می رسیم.

```
#3  0x80106361 in trap (tf=0x1010101) at trap.c:43
43      syscall();
(gdb)
```

همانطور که پیشتر اشاره شد، رجیستر `eax` برای ذخیره سازی شماره سیستم کال به کار می رود. شماره دستور `getpaid` برابر 11 است، اما با اجرای دستور زیر به مقدار 11 نمی رسیم، زیرا قبل از سیستم کال مورد نظر، فراخوانی های دیگری هم انجام شده است.

```
Terminal
(gdb) print myproc()->tf->eax
$6 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$7 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$8 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$9 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$10 = 1
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$11 = 3
(gdb) c
Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$12 = 12
(gdb) c
Continuing.
```

```

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$2 = 1
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$3 = 12
(gdb) c
Continuing.

Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$4 = 7
(gdb) c
Continuing.

Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$5 = 11
(gdb) c
Continuing.

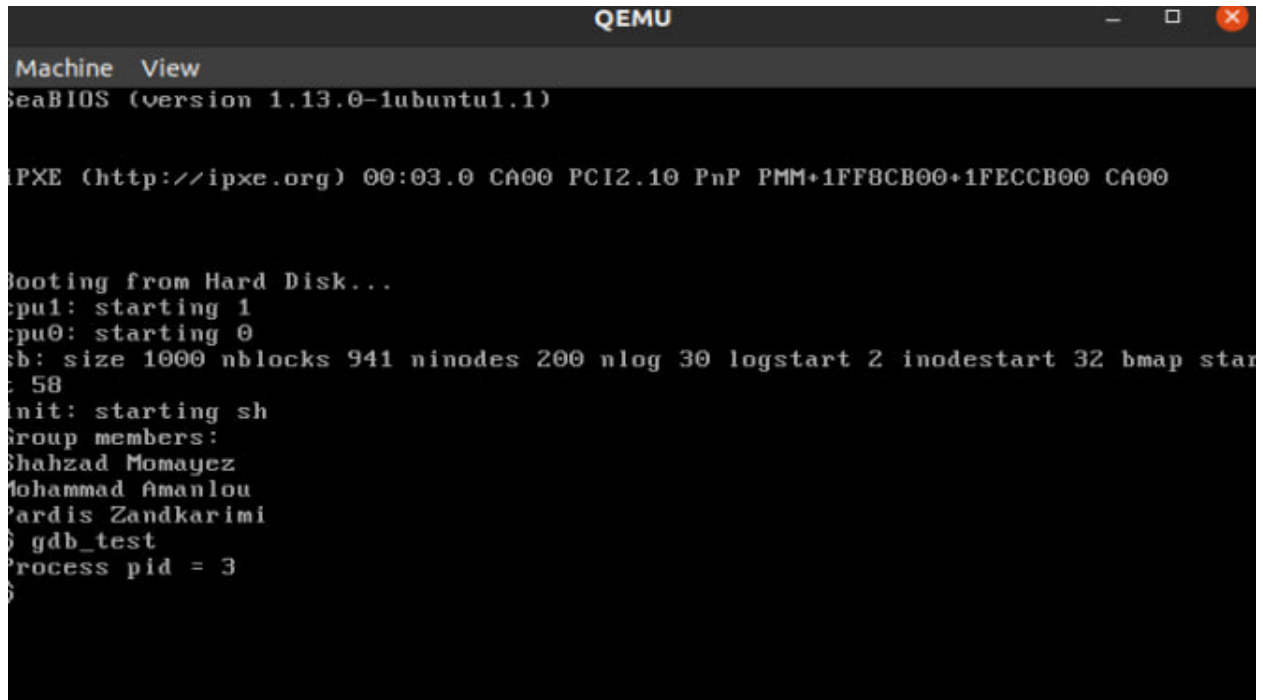
Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$6 = 16
(gdb) c

```

در ذیل معنای هر یک از کد ها نوشته شده است:

عملکرد	نام فراخوانی سیستمی	کد فراخوانی سیستمی
خواندن از ترمینال	read	5
ایجاد پردازش جدید برای اجرای برنامه سطح کاربر	fork	1
انتظار برای اتمام پردازش فرزند	wait	3
تخصیص حافظه به پردازش ایجاد شده	sbrk	12
قرار دادن برنامه سطح کاربر در حافظه پردازش ایجاد شده	exec	7
شماره پردازش فعلی را بر می گردانند.	getpid	11
خروجی برنامه سطح کاربر را روی ترمینال چاپ می کند	write	16

در نهایت نیز با اجرای دستور gdb_test که یک pid را get می کند به خروجی زیر می رسیم.



```
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group members:
Shahzad Momayez
Mohammad Amanlou
Pardis Zandkarimi
$ gdb_test
Process pid = 3
$
```

ارسال آرگومان‌های فراخوانی‌های سیستمی

ابتدا در فایل `syscall.h` ، شماره 22 را به این سیستم کال اختصاص می‌دهیم:

```
23  #define SYS_find_digital_root 22
```

همچنین در `user.h`, `usys.S`, `syscall.c` و `defs.h` نام و declaration این تابع را اضافه می‌کنیم که تغییرات هر کدام را میتوان مشاهده کرد:

اضافه کردن prototype تابع سیستم کال در `syscall.c`:

```
106  extern int sys_find_digital_root(void);
```

اضافه کردن سیستم کال به آرایه `syscalls` در `syscall.c`:

```
133  [SYS_find_digital_root] sys_find_digital_root,
```

تعریف تابع در `usys.S`:

```
SYSCALL(find_digital_root)
```

اضافه کردن prototype تابع سیستم کال در `user.h`:

```
int find_digital_root(void);
```

اضافه کردن تابع در `defs.h`:

```
125  int find_digital_root(int);
```

در فایل `proc.c` ، بدنه‌ی تابع `int find_digital_root(int n)` را تعریف می‌کنیم که منطق این برنامه در آن قرار دارد و به ازای گرفتن عدد `n`، ریشه دیجیتال آن را برمی‌گرداند:

```

606 int
607 find_digital_root(int n){
608     while(n>9){
609         int sum = 0 ;
610         while(n > 0){
611             int digit = n%10;
612             sum += digit;
613             n = n/10;
614         }
615         n = sum;
616     }
617
618     return n;
619 }

```

در فایل sysproc.c تابع `int sys_find_digital_root(void)` را تعریف می‌کنیم در این تابع، تابع `find_digital_root` که در فایل `proc.c` قرار دارد و در کرنل است و آرگومان ورودی را با استفاده از رجیستر `ebx` پاس می‌دهد صدا زده میشود.

```

int
sys_find_digital_root(void)
{
    int n = myproc()->tf->ebx;
    cprintf("KERNEL: sys_find_digital_root(%d)\n", n);
    return find_digital_root(n);
}

```

در انتها برای تست کردن فراخوانی سیستمی ایجاد شده یک فایل تست `digital_root_testing.c` میسازیم برای تغییر نکردن محتوای رجیستر استفاده شده در این فراخوانی، در فایل تست، مقدار آن را در یک متغیر در سطح حافظه نگه داشته، سپس مقدار مورد نظر را در آن قرار میدهم، فراخوانی

سیستمی را انجام میدهیم و در آخر، مقدار قبلی رجیستر استفاده شده را در آن مینویسیم:

```
C digital_root_testing.c
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char *argv[])
6  {
7      if(argc != 2)
8      {
9          if(argc < 2)
10             printf(2, "Error: you didn't enter the number!\n");
11             else if(argc > 2)
12                 printf(2, "Error: Too many arguments!\n");
13             exit();
14         }
15     else
16     {
17
18         int last_ebx_value;
19         int number = atoi(argv[1]);
20
21         asm volatile(
22             "movl %%ebx, %0;" // last_ebx_value = ebx
23             "movl %1, %%ebx;" // ebx = number
24             : "=r" (last_ebx_value)
25             : "r"(number)
26         );
27         printf(1, "USER: find_digital_root() is called for n = %d\n" , number);
28         int answer = find_digital_root();
29         printf(1, "digital root of number %d is: %d\n" , number , answer);
30
31         asm("movl %0, %%ebx"
32             :
33             : "r"(last_ebx_value)
34             );
35     }
36     exit();
37 }
38
```

همچنین این فایل را به makefile در قسمت های EXTRA و UPROGS اضافه میکنیم:

```
printf.c umalloc.c digital_root_testing.c test_copy_file.c\
```

186

_digital_root_testing\

با اجرا کردن این برنامه به خروجی زیر میرسیم:

```

init: starting sh
Group members:
Shahzad Momayez
Mohammad Amanlou
Pardis Zandkarimi
$ digital_root_testing 457
USER: find_digital_root() is called for n = 457
KERNEL: sys_find_digital_root(457)
digital root of number 457 is: 7
$ _

```

پیاده سازی فراخوانی های سیستمی

1. پیاده سازی فراخوانی سیستمی کپی کردن فایل

ابتدا در فایل `syscall.h` ، شماره 24 را به این سیستم کال اختصاص میدهیم:

```
25 #define SYS_copy_file 24
```

همچنین در `user.h`، `usys.S`، `syscall.c` نام و `declaration` این تابع را اضافه میکنیم که تغییرات هر کدام را میتوان مشاهده کرد:

اضافه کردن `prototype` تابع سیستم کال در `syscall.c`:

```
108 extern int sys_copy_file(void);
```

اضافه کردن سیستم کال به آرایه `syscalls` در `syscall.c`:

```
135 [SYS_copy_file] sys_copy_file,
```

تعریف تابع در `usys.S`:

```
34 SYSCALL(copy_file)
```

اضافه کردن `prototype` تابع سیستم کال در `user.h`:

```
28 int copy_file(const char*, const char*);
```


در تعریف این تابع باید دقت داشت که در صورت وجود فایل مقصد از قبل برنامه باید با ارور مواجه شود.

همچنین همانطور که در صورت پروژه گفته شده است در صورت موفقیت فراخوانی سیستمی عدد 0 و در غیر اینصورت مقدار منفی یک را برمیگرداند و به همین دلیل مقدار بازگشتی آن را `int` در نظر گرفتیم.

نکته دیگری که باید رعایت شود این است که در پیاده سازی این مورد حق استفاده از دستوراتی مانند `close`، `write`، `open`، `read` را نداریم و باید از دستورات مربوط به هسته مانند `create` و `namei` استفاده می کنیم.

سپس سیستم کال را چون مرتبط با کار با فایل است در `sysfile.c` مطابق ذیل تعریف میکنیم.

```

446 int
447 sys_copy_file(void) {
448     char* src_path;
449     char* dst_path;
450
451     struct inode* ip_dst;
452     struct inode* ip_src;
453     int bytesRead;
454     char buf[1024];
455
456     if (argstr(0, &src_path) < 0 || argstr(1, &dst_path) < 0)
457         return -1;
458     begin_op();
459
460     if ((ip_src = namei(src_path)) == 0) { // Check if source file exists
461         end_op();
462         return -1;
463     }
464
465     ip_dst = namei(dst_path);
466     if (ip_dst > 0) { // Check if destination file already exists
467         end_op();
468         return -1;
469     }
470     ip_dst = create(dst_path, T_FILE, 0, 0);
471
472     int bytesWrote = 0;
473     int readOffset = 0;
474     int writeOffset = 0;
475     ilock(ip_src);
476     while ((bytesRead = readi(ip_src, buf, readOffset, sizeof(buf))) > 0) {
477
478         ilock(ip_dst);
479         while ((bytesRead = readi(ip_src, buf, readOffset, sizeof(buf))) > 0) {
480             readOffset += bytesRead;
481             if ((bytesWrote = writei(ip_dst, buf, writeOffset, bytesRead)) <= 0)
482                 return -1;
483             writeOffset += bytesWrote;
484         }
485
486         iunlock(ip_src);
487         iunlock(ip_dst);
488         end_op();
489
490         return 0;
491     }
492 }

```

در نهایت فایل زیر را به جهت تست کردن سیستم کال ایجاد شده مینویسیم و آن را در برنامه های سطح کاربر در میک فایل نیز اضافه میکنیم.

```

C test_copy_file.c > main(int, char * [])
1
2  #include "types.h"
3  #include "stat.h"
4  #include "user.h"
5  #include "fcntl.h"
6  int main(int argc, char *argv[])
7  {
8
9
10     if(argc != 3)
11     {
12         if(argc < 3)
13             printf(2, "Error: you didn't enter 2 files!\n");
14         else if(argc > 3)
15             printf(2, "Error: Too many arguments!\n");
16     }
17     exit();
18
19
20     int a = copy_file(argv[1], argv[2]);
21     if (a == -1) {
22         printf(2, "Error: File already exists!\n");
23     }
24
25     exit();
26 }

```

نمونه خروجی نیز به نحو زیر خواهد بود:

```

Group members:
Shahzad Momayez
Mohammad Amanlou
Pardis Zandkarimi
$ test_copy_file README a
Error: File already exists!
$ test_copy_file README n
$ cat n
NOTE: we have stopped maintaining the x86 version of xv6, and switched
our efforts to the RISC-V version
(https://github.com/mit-pdos/xv6-riscv.git)

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,

```

2. پیاده سازی فراخوانی سیستمی تعداد **uncle** های پردازش

مشابه موارد ذکر شده تغییرات را اعمال میکنیم. در فایل `syscall.h` شماره 25 را به این سیستم کال اختصاص میدهیم. و تغییرات مورد نیاز را نیز در فایل های `user.h`, `usys.S`, `syscall.c` و `defs.h` اعمال میکنیم. فایل `proc.c` تابع مورد نیاز را تعریف میکنیم و آرگومان های مورد نیاز آن را از طریق فایل `sysproc.c` پیدا کرده و به آن پاس میدهیم. در تابع مذکور پردازش پدربزرگ پردازش فعلی را پیدا کرده و با استفاده از `iterate` کردن در `ptable` پردازش های `uncle` را پیدا میکنیم.

تعریف این تابع به صورت زیر است:

```

int get_uncle_count(int pid){
    struct proc *p;
    struct proc *p_parent;
    struct proc *p_grandParent = 0;
    acquire(&ptable.lock);
    if(pid < 0 || pid >= NPROC){
        return -1;
    }
    int count = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        count ++;
        if((p->pid) == pid){
            p_parent = p->parent;
            if(p_parent != 0){
                p_grandParent = p_parent->parent;
                if(p_grandParent == 0){
                    cprintf("grandparent is zero.");
                    return -1;
                }
            }
            else{
                cprintf("parent is zero.");
                return -1;
            }
        }
        if(count > 80){
            break;
        }
    }
    int siblings = 0;
    for(struct proc *i = ptable.proc; i < &ptable.proc[NPROC]; i++){
        if(i->parent == p_grandParent){
            siblings++;
        }
    }
    release(&ptable.lock);
}

```

همچنین برنامه سطح کاربری برای این کار در نظر گرفته شده است که در آن سه پردازش c1, c2, c3 به پردازش فعلی fork میشوند و از درون c1 هم پردازش جدیدی fork میشود. Pid این پردازش را به تابع پاس میدهیم و pid پردازش فعلی را به عنوان نتیجه خواهیم داشت.

```
int main(){

    int c1, c2, c3;

    int grandchild;

    c1 = fork();

    if(c1 == 0){

        printf(1, "c1 forked. \n");

        sleep(100);

        grandchild = fork();

        if(grandchild == 0){

            printf(1,"grandparent forked\n");

            int pid = getpid();

            int uncles = get_uncle_count(pid);

            printf(1, "uncles of c1: %d\n", uncles);

            exit();

        }

        wait();

        exit();

    }

    c2 = fork();

    if(c2 == 0){

        printf(1, "c2 forked. \n");

        sleep(200);
```

```

        exit();

    }

    c3 = fork();

    if(c3 == 0){

        printf(1, "c3 forked. \n");

        sleep(300);

        exit();

    }

    wait();

    wait();

    wait();

    exit();

}

```

در نهایت نیز خروجی مطابق زیر خواهد بود:

```

Group members:
Shahzad Momayez
Mohammad Amanlou
Pardis Zandkarimi
$ get_uncle_test
c1 forked.
c2 forked
c3 forked.
d1.
grandparent forked
uncles of c1: 3
$

```

3. پیاده سازی فراخوانی سیستمی طول عمر پردازش

ابتدا در فایل `syscall.h` ، شماره 23 را به این سیستم کال اختصاص میدهیم:

```
24 #define SYS_get_process_lifetime 23
```

همچنین در user.h, usys.S, syscall.c و defs.h نام و declaration این تابع را اضافه میکنیم که تغییرات هر کدام را میتوان مشاهده کرد:

اضافه کردن prototype تابع سیستم کال در syscall.c:

```
107 extern int sys_get_process_lifetime(void);
```

اضافه کردن سیستم کال به آرایه syscalls در syscall.c:

```
[SYS_get_process_lifetime] sys_get_process_lifetime,  
[SYS_sys_file] sys_sys_file,
```

تعریف تابع در usys.S:

```
33 SYSCALL(get_process_lifetime)
```

اضافه کردن prototype تابع سیستم کال در user.h:

```
27 int get_process_lifetime(void);
```

اضافه کردن تابع در defs.h:

```
107 int get_process_lifetime(void);
```


همچنین برای تست کردن این سیستم کال یک تابع به صورت زیر تعریف میکنیم:

```
int main(int argc, char *argv[]) {
    int pid = fork();
    printf(1, "pid is %d \n", getpid());
    if(pid < 0){
        printf(1, "Fork failed.\n");
    }
    else if(pid == 0){
        printf(1, "pid of child is %d \n", getpid());
        sleep(1000);
        exit();
    }
    else{
        wait();

        int lifetime = get_process_lifetime();
        if(lifetime >= 0){
            printf(1, "child process lifetime: %d ticks\n", lifetime);
        }
        else{
            printf(1, "error getting process lifetime.\n");
        }

        int parent_lifetime = get_process_lifetime();
        printf(1, "parent lifetime is %d \n", parent_lifetime);
    }

    exit();
}
```

در این تابع یک پردازش فرزند با استفاده از دستور `fork` ساخته میشود. در این پردازش تنها دستور `sleep` استفاده میشود تا مقدار زمان حضور آن مشخص باشد. سپس با استفاده از سیستم کال پیاده سازی شده اختلاف زمان بین پردازش فعلی و پردازش فرزند سنجیده میشود. مقدار آن از آرگومان `sleep` گرفته شده در آرگومان `sleep` است.

برای پیاده سازی این سیستم کال مقدار جدیدی را به استراکت `proc` با نام `start_time` ایجاد میکنیم که زمان ورود آن پردازش به سیستم را در خود ذخیره خواهد کرد. سپس در تابع `alloc` که در آن پردازش `allocate` میشود خط زیر را اضافه میکنیم:

```
p->start_time = ticks;
|
```

سپس تابع مورد نظرمان در `proc.c` را به صورت زیر پیاده سازی میکنیم:

```

int
get_process_lifetime(void){
    return sys_uptime() - myproc()->start_time ;
}

```

در این تابع از سیستم کال پیاده سازی شده در خود سیستم استفاده میکنیم که زمان فعلی پردازنده را نشان میدهد که در صورت تفریق کردن آن از `start_time` که کاربرد آن پیش تر گفته شد میتوانیم زمانی که پردازنده ذکر شده حضور دارد را خروجی دهیم.

شکل زیر نحوه اضافه شدن `start_time` در استراکت گفته شده در فایل `proc.h` را نشان میدهد:

```

2      char name[16];           // Process name (debugging)
3      uint start_time;         //Process start tick

```

و در نهایت خروجی مطابق زیر خواهد بود :

```

$ cat 58
$ ./init: starting sh
Group members:
Shahzad Momayez
Mohammad Amanlou
Pardis Zandkarimi
$ get_process_lifetime_test
pid is 4
pidpid of is 3
child is 4
child process lifetime: 1014 ticks
parent lifetime is 1017
$

```