به نام خدا

گزارش کار پروژه دوم آزمایشگاه سیستم عامل

" فراخوانی سیستمی"

گروه 5 :

- سینا طبسی 810199554

- سید حامد میرامیرخانی 810199500

- فاطمه محمدی 810199489

Repository Link: https://github.com/HamedMiramirkhani/OS_Lab_CA2

❖ سوالات

خطوط گفته شده در سوالات ۱و ۲ مربوط به کد خام xv6 میباشد

1) کتابخانه های(قاعدتا سطح کاربر، منظور فایل‌های تشکیل دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانی های سیستمی و علت این استفاده بررسی نمایید.

متغیر ULIB در Makefile شامل چهار object file میباشد(خط 146):

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

در ادامه به صورت جداگانه به بررسی سورس هرکدام فایل های نامبرده میپردازیم:

1. ulib.o

سورس: ulib.c

فایل ulib.c دارای توابع کمکی متنوع زیر میباشد:

strcpy, strcmp, strlen, memset, strchr, gets(line: 53), stat(line: 71), atoi, memmove

از هفت تابع بالا در دو تابع gets و stat، از فراخوانی های سیستمی استفاده شده است.

- **gets:**

در این تابع از تنها از یک سیستم کال read استفاده شده است (در خط 59):

cc = **read**(0, &c, 1);

که در حلقه for استفاده شده است تا ورودی را از stdin بخواند. (همانطور که میدانیم پارامتر اول فراخوانی read که 0 میباشد، ایدی مربوط به stdin است.)

- **stat:**

در این تابع از دو سیستم کال open و close به ترتیب در خطوط 76 , 80 استفاده شده است که به منظور بازکردن و بستن یک فایل میباشد.

همچنین در خط 79 ازسیستم کال fstat استفاده شده است که از طریق ان اطلاعات مربوط به فایل مورد نظر را دریافت کنیم.

```
fd = open(n, O_RDONLY);
if(fd < 0)
  return -1;
r = fstat(fd, st);
close(fd);
```

2. usys.o

سورس: usys.S

سورس usys.o، از نوع (.S) میباشد پس usys.o از کد اسمبلی تولید میشود.

به بررسی سورس میپردازیم:

در دو خط اول تنها کتابخانه های مورد نظر اضافه شده اند.

در خطوط 4 تا 9 که در ادامه امده است ماکرو زیر را داریم که به ازای هر system call، این ماکرو با اسم آن فراخوانی استفاده و اجرا میشود؛ که به این منظور توابعی در سطح user.h در سطح C دیکلر شده اند.

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

روال نحوه استفاده از این ماکرو را برای هر فراخوانی بررسی میکنیم(جهت بررسی فراخوانی های سیستمی - و تا حدودی علت که تقریبا در بالاتر توضیح اندکی داده شده است.):

a. میدانیم برای هر سیستم کال یک عدد نسبت داده شده است، ابتدا این عدد در رجیستر EAX ذخیره میشود.
b. سپس INT64 صدا زده میشود و در نتیجه یک وقفه رخ میدهد.
c. در نتیجه رخداد وقفه، تابع trap صدا زده میشود؛
d. با توجه به اینکه یک سیستم کالl میباشد، تابع syscall صدا زده میشود.
e. در تابع syscall، مقدار رجیستر EAX خوانده میشود، که متوجه شود کدام سیستم کال باید اجرا شود.

3. printf.o

سورس: printf.c

- **putc**

همانطور که در فایل دیده میشود، سه تابع در این فایل تعریف شده اند:

1. putc -> خط 5ام
```
static void
putc(int fd, char c)
{
  write(fd, &c, 1);
}
```
2. printint -> خط 12 ام
3. printf -> خط 40 ام

که در تابع putc یک کاراکتر با استفاده از سیستم کال write در فایل با fd مورد نظر مینویسد.
در تابع printinit و همچنین در تابع printf نیز از تابع putc استفاده شده است که از فراخوانی سیستمی مورد نظر استفاده میکند.

4. umalloc.o

سورس: umalloc.c

- **morecore**

این فایل همانطور که در ابتدا فایل ذکر شده است جهت memory allocator میباشد (کامنت خط 6) و در این فایل سه تابع زیر تعریف شده اند:

1. free -> خط 25ام
2. morecore -> خط 47 ام
```
static Header*
morecore(uint nu)
{
  char *p;
  Header *hp;

  if(nu < 4096)
    nu = 4096;
```

```
  p = sbrk(nu * sizeof(Header));
  if(p == (char*)-1)
    return 0;
  hp = (Header*)p;
  hp->s.size = nu;
  free((void*)(hp + 1));
  return freep;
}
```

3. malloc -> خط 64ام

همانطور که در کد moreore که آورده ایم، میبینیم از سیستم کال sbrk استفاده شده است که فضای پردازه را افزایش میدهد.
و همچنین در تابع malloc که برای تخصیص حافظه است از تابع کمکی morecore استفاده شده است.

2) دقت شود فراخوانی های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش ها را در لینوکس به اختصار توضیح دهید. میتوانید از مرجع[3] کمک بگیرید.

- **وقفه های سخت افزاری**

این وقفه ها از طریق سخت افزار ها رخ میدهند. برای مثال فشردن کلیدی در کیبورد و یا موس.

- **وقفه های نرم افزاری (Trap ها) که انواع مختلفی دارند:**

a. **فراخوانی سیستمی(System call)**

b. **استثناء (Exception):**

در زمان وقوع استثناء هایی(خطاهایی) همانند دسترسی بدون مجوز به حافظه و یا تقسیم بر صفر، دسترسی به kernel انجام میشود تا خطا رفع شود و و پس رفع خطا به سطح کاربر (user mode) برمیگردیم.

c. **سیگنال (Signal):**

سیگنال وقفه های نرم افزاری هستند که به یک برنامه ارسال میشوند تا نشان دهند اتفاق مهمی رخ داده است. درواقع سیگنال ها در زمان وقوع خطا توسط Shell و یا Terminal handler ایجاد میشوند تا باعث ایجاد وقفه شوند و یا میتوانند از یک فرآیند به فرآیند دیگر ارسال شوند.
سیگنالها در لینوکس انواع مختلف و متنوعی دارند که میتوان به SIGQUITو SIGTRAPو SIGKILL و SIGTERM و SIGTINIT نام برد.
(از دستور Kill برای ارسال سیگنال ها استفاده میشود.)

- همچنین میتوان به **Pseudo-file system** ها اشاره کرد که در انها به دسترسی سطح هسته(kernel) نیاز است.

در واقع Pseudo-file system ها یک entry مجازی هستند، که این امکان را به یک application و یا ادمین میدهد که محتوای داده ساختارهای درون هسته را دریافت کنند به گونه ای که انگار محتوا روی یک فایل ذخیره شده است؛ پس به منظور دسترسی به داده ساختارهای هسته Pseudo file system ها نیز نیازمند دسترسی به سطح هسته می باشد.
از Pseudo file system های لینوکس میتوان به /sys و /proc و... اشاره کرد.

3) آیا همه تله ها را میشود با سطح دسترسی DPL_USER فعال نمود؟

خیر. در صورت تلاش کاربر برای فعال کردن trap ای دیگر، xv6 این اجازه را برای او صادر نمی کند و protection exception توسط کاربر دیده خواهد شد. این اتفاق به این دلیل است که امکان سوء استفاده توسط کاربر و یا وجود اشکال در برنامه کاربر وجود دارد. اگر این اجازه اجرای این trap ها برای کاربر وجود داشت، کاربر می توانست به هسته دسترسی داشته باشد. در این صورت امنیت سیستم به خطر می افتد.

4) چرا در صورت تغییر سطح دسترسی، ss و esp روی پشته push میشوند؟

به طور کلی User stack و Kernel stack وجود دارد. زمانی که یک trap فعال شده و تغییر سطح دسترسی ایجاد شود، برای دسترسی سیستم به کد و ساختار داده هسته، باید از stack هسته بهره ببرد. در نتیجه ابتدا باید esp و ss که به stack حال حاضر اشاره می کنند، ذخیره شوند. سپس با استفاده از این دو رجیستر (ss و esp) می توان به stack هسته اشاره کرد.

پس از رسیدگی به trap، مقدار دو رجیستر ss و esp به مقادیر اولیه خود برگشته و برنامه کاربر از محل وقوع trap به اجرا خود ادامه می دهد. حال اگر سطح دسترسی تغییر نیابد، نیازی به ذخیره دو رجیستر ss و esp نمی باشد. زیرا همچنان با همان stack در حال کار کردن می باشیم.

5) توضیح توابع دسترسی به پارامترهای فراخوانی سیستمی

توابع argint، argptr و argstr برای دسترسی به پارامترهای فراخوانی سیستمی تعریف شده اند. این توابع در صورت آرگومان غیر مجاز، مقدار -1 را برمی گردانند. توضیح این سه تابع در ادامه گفته شده است:

1) **تابع argint:** آدرس آرگومان nام ورودی در حافظه را محاسبه می کند. stack از آدرس بیشتر به کمتر پر شده و آخرین مقداری که در stack مورد نظر push می شود، آدرس نقطه return از تابع است و آرگومان های ورودی تابع قبل از آدرس نقطه return از تابع قرار می گیرند. همچنین آدرس ابتدای stack در رجیستر esp ذخیره می شود. در آخر، آدرس مورد نظر همراه پوینتر به حافظه برای مقدار int به تابع fetchint فرستاده می شود. در این تابع ، ابتدا بررسی می شود که آدرس ارسالی + 4 بایت در حافظه پردازه موجود باشد و اگر این مورد تایید شود، آرگومان دوم توسط آن مقدار دهی می شود.
2) **تابع argstr**: با کمک تابع argint، آدرس شروع رشته را مشخص کرده و بعد آن را به تابع fetchstr می فرستد. حال در تابع fetchstr ابتدا بررسی وجود آدرس داده شده در حافظه پردازه انجام می گیرد و بعد از آن در صورت تایید وجود ادرس، مقدار آرگومان دوم را برابر این آدرس قرار می دهد. در آخر، از ابتدای آدرس شروع کرده و پیمایش را انجام می دهد. اگر به NULL character برخورد کرد، طول رشته را return می کند. حال اگر به انتهای حافظه پردازه برسد و NULL character را مشاهده نکند، مقدار -1 را return می کند.
3) **تابع argptr:** به کمک تابع argint آدرس پوینتر را دریافت می کند.بعد از آن سایز پوینتر (آرگومان سوم) با کمک تابع argint گرفته و بررسی انجام می دهد که آیا پوینتر با سایز داده شده در حافظه پردازه می باشد و یا خیر. در صورت عدم وجود مشکل، آرگومان دوم مقدار دهی می شود.

در تمامی این توابع بررسی وجود آدرس داده شده در پردازه انجام می گیرد. این عمل به این خاطر صورت می گیرد که یک پردازه نتواند دسترسی به پردازه های دیگر داشته باشد. چون باعث ایجاد مشکل امنیتی و یا ایجاد مشکل در پردازه های دیگر شود.

میتوان برای درک این موضوع، می توان فراخوانی سیستمی sys_read را مورد بررسی قرار بدهیم. این فراخوانی سیستمی مربوط به تابع read می باشد. در این تابع، مقدار خوانده شده در آرگومان دوم آن که یک buffer است، قرار می گیرد. در آرگومان سوم آن، مقدار حداکثر تعداد بایت های که می خواند قرار دارد. اگر سیستم عامل قبل از خواندن این تعداد حداکثری بایت، به EOF رسید، عملیات خواندن از فایل خاتمه می یابد.

حال به بررسی کد sys_read می پردازیم. همانطور که مشاهده می کنیم، کد آن در ادامه آورده شده است:

```
int
sys_read(void)
{
        struct file *f; int n;
        char *p;
        if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
                return -1;
        return fileread(f, p, n);
}
```

این تابع به کمک argfd مقدار file descriptor را می گیرد (توضیحات argfd در ادامه آورده شده است) و بعد از آن آرگومان سوم را با استفاده از تابع argint به دست آورده و در آخر با استفاده از تابع argptr ، این بررسی را انجام می دهد که آیا تمامی فضای آدرس دهی از ابتدا پوینتر به buffer تا انتهای آن در حافظه قرار می گیرد یا خیر.

در صورت عدم بررسی این مورد، امکان استفاده از تابع read با مقدار max بزرگ و برای فایلی بزرگ وجود داشت. در این صورت، در هنگام read از فایل و write در buffer، این امکان وجود داشت که سیستم عامل از حافظه پردازه خارج شود و در حافظه پردازه دیگری به نوشتن ادامه دهد که این مورد باعث ایجاد مشکل می شود. همچنین اگر مقدار max اگر از طول buffer بیشتر باشد اما حافظه پردازه بیرون نزند، باز ممکن است باعث overflow شدن buffer و ایجاد مشکل شود.

تابع argfd: با استفاده از تابع argint مقدار fd (آرگومان اول تابع read) را گرفته و اعتبار و درستی این fd را مورد بررسی قرار می دهد.

❖ بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

در ابتدای این قسمت، طبق شرح آزمایش، باید برنامه ای ساده در سطح کاربر نوشت. نام این برنامه pid بوده و با اجرای این برنامه وضعیت فعلی process id پردازه فعلی را چاپ می کند. کد آن به شکل زیر می باشد:

```c
#include "types.h"
#include "user.h"

int main(int argc, char* argv[]) {

    int pid = getpid();
    printf(1, "Process ID: %d\n", pid);
    exit();
}
```

حال با اجرای gdb mode، در خط 148 فایل syscall.c یک breakpoint قرار داده و مطابق دستورالعمل آزمایش، برنامه pid را اجرا میکنیم. با اجرای برنامه، برنامه در محل breakpoint متوقف می شود. حال اگر دستور bt را اجرا نماییم، خروجی به صورت زیر خواهد بود:

```
syscall.c
143       int pre_count = curproc->count_calls[num];
144       curproc->count_calls[num] = pre_count + 1;
145
146       if (num > 0 && num < NELEM(syscalls) && syscalls[num])
147       {
B+> 148     curproc->tf->eax = syscalls[num]();
149       }
150       else
151       {
152         cprintf("%d %s: unknown sys call %d\n",
153                 curproc->pid, curproc->name, num);
154         curproc->tf->eax = -1;
155       }
156     }
157
158
159
160
161
162
163
164
165
166
167
168
169
170
remote Thread 2 In: syscall
(gdb) bt
#0  syscall () at syscall.c:148
#1  0x80105915 in trap (tf=0x8dfbefb4) at trap.c:43
#2  0x8010572c in alltraps () at trapasm.S:20
#3  0x8dfbefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

همانطور که در تصویر بالا مشاهده می کنیم، با دستور bt می توانیم call stack برنامه را در همان لحظه مشاهده نماییم. با صدا زدن تابع، یک stack frame مخصوص آن تابع به وجود آمده که در آن اطلاعاتی نظیر آدرس بازگشت و متغیرهای محلی و .... قرار می گیرد. دستور bt، این stack frame ها را به ترتیب از داخلی ترین به بیرونی ترین frame را نشان می دهد.

در یک فراخوانی سیستمی، برای تعریف و اجرا، مراحل زیر طی می شود:

1) در فایل syscall.h برای system call یک عدد انتخاب می شود.
2) در فایل user.h شناسه system call نوشته می شود.
3) در فایل usys.s تعریف system call در زبان assembly انجام می گیرد.
4) در فایل vectors.s تعریف vector 64 انجام گرفته و با اجرای دستور int 64 در مرحله پیشین، وارد این قسمت می شویم. پس از push شدن مقدار 64، به فایل trapasm.s و بخش alltraps هدایت می شویم.

5) در این بخش که alltraps می باشد، با ساختن trap frame مربوطه و push کردن آن در استک، تابع trap در فایل trap.c را call می کند.
6) در این بخش، تابع trap پس از فهمیدن فراخوانی به عنوان system call، آرگومان تابع trap frame که در استک push شده است را به عنوان trap frame پردازه فعلی قرار داده و پس از آن تابع syscall را call می کند.
7) در این بخش (syscall در فایل syscall.c) ابتدا یک آرایه syscalls تعریف می شود و در آن شماره مربوط به system call را به تابع map کرده و پس از آن تابع syscall با خواندن آن شماره در eax در trap frame پردازه فعلی، تابع مربوطه را call می کند و خروجی آن را eax همان trap frame ذخیره میکند.

تصویر بالا، مراحل 5 تا 7 را نشان می دهند.

در تصویر زیر خروجی دو دستور down و up را مشاهده می کنیم:

```
trap.c
27      }
28
29      void
30      idtinit(void)
31      {
32        lidt(idt, sizeof(idt));
33      }
34
35      //PAGEBREAK: 41
36      void
37      trap(struct trapframe *tf)
38      {
39        if(tf->trapno == T_SYSCALL){
40          if(myproc()->killed)
41            exit();
42          myproc()->tf = tf;
> 43          syscall();
44          if(myproc()->killed)
45            exit();
46          return;
47        }
48
49        switch(tf->trapno){
50        case T_IRQ0 + IRQ_TIMER:
51          if(cpuid() == 0){
52            acquire(&tickslock);
53            ticks++;
54            wakeup(&ticks);
55            release(&tickslock);
56          }
57          lapiceoi();
58          break;
59        case T_IRQ0 + IRQ_IDE:
60          ideintr();
remote Thread 1 In: trap
(gdb) bt
#0  syscall () at syscall.c:148
#1  0x80105915 in trap (tf=0x8dffefb4) at trap.c:43
#2  0x8010572c in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1  0x80105915 in trap (tf=0x8dffefb4) at trap.c:43
(gdb)
```

در تصویر بالا، با اجرای دستور down به ارور بر می خوریم، زیرا در حال حاضر ما در داخلی ترین frame قرار داریم و نمی توانیم به frame داخلی تر برویم. با اجرای دستور up ، یک frame به عقب می رویم که فراخوانی تابع syscall در تابع trap می باشد.

در تصویر زیر که مربوط به چاپ کردن محتوای رجیستر eax می باشد، عدد 5 را مشاهده می کنیم. در صورتی که می دانیم شماره فراخوانی سیستمی getpid برابر 11 می باشد. خروجی به صورت زیر است:

```
(gdb) p num
$6 = 5
```

دلیل این تفاوت این است که قبل از رسیدن به فراخوانی سیستمی getpid، فراخوانی سیستمی دیگری، مانند read، اجرا می شود. با اجرای مکرر دستور c و خواند محتوای eax مراحل زیر به ترتیب طی می شوند:

1) شماره 5 (read) برای خواندن دستور تایپ شده در ترمینال به صورت کامل.
2) شماره 1 (fork) برای ایجاد پردازه جدید برای اجرای برنامه سطح کاربر.
3) شماره 3 (wait) برای انتظار برای پایان یافتن اجرای پردازه فرزند.
4) شماره 12(sbrk) برای تخصیص حافظه به پردازه ایجاد شده.
5) شماره 7 (exec) برای اجرای برنامه pid در پردازه ایجاد شده.
6) شماره 11(getpid) برای برنامه سطح کاربر ذکر شده است.

پس از تمامی این مراحل، تعدادی سیستم کال دیگر برای چاپ مقدار خروجی در ترمینال اجرا می شوند.
تصویر زیر مراحل گفته شده در بالا در خروجی ترمینال را نشان می دهد:

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$6 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$7 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$8 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 2, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$9 = 1
(gdb) c
Continuing.

Thread 1 hit Breakpoint 5, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$10 = 3
(gdb) c
Continuing.
[Switching to Thread 2]

Thread 2 hit Breakpoint 3, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$11 = 12
(gdb) c
Continuing.

Thread 2 hit Breakpoint 4, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$12 = 7
(gdb) c
Continuing.

Thread 2 hit Breakpoint 6, syscall () at syscall.c:148
148         curproc->tf->eax = syscalls[num]();
(gdb) p num
$13 = 11
```

در نهایت خروجی برنامه pid به صورت زیر خواهد بود:

```
************************
Group #5
Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
************************
$ pid
Process ID: 3
$
```

❖ تابع find_fibonacci_number

- لیست فایل های تغییر داده شده در هر مرحله:
  1. user.h
  2. syscall.h
  3. syscall.c
  4. usys.S
  5. Makefile
- لیست فایل های ایجاد شده در هر مرحله:
  1. sys_fibonacci.c
  2. test_find_fibonacci_number.c
- مراحل اضافه کردن این فراخوانی سیستمی:

1. ابتدا باید تابع را در user.h تعریف کنیم:

```
int find_fibonacci_number(void);
```

نکته: باتوجه به اینکه میخواهیم از رجیسترها برای پاس کردن ورودی ها و اطلاعات استفاده کنیم، همانطور که در تعریف مشاهده میشود از در تابع ورودی نمیگیریم(چرا که نمیخواهیم از استک استفاده کنیم.)

2. در این مرحله باید فراخوانی سیستمی را در سطح کرنل پیاده سازی کنیم :
   a. ابتدا باید در فایل syscall.h، شماره فراخوانی سیستمی اضافه میکنیم را اضافه کنیم،(با توجه به اینکه از شماره ۱ تا ۲۱ برای فراخوانی های سیستمی پیشفرض خود xv6 استفاده شده است به این فراخوانی شماره ۲۲ را نسبت می دهیم)

```
#define SYS_find_fibonacci_number 22
```

   b. در قدم بعدی تعریف تابع را در syscall.c در قسمت مربوطه اضافه کنیم

```
extern int sys_find_fibonacci_number(void);
```

   c. در اخرین قدم این مرحله باید فراخوانی سیستمی را در ارایه syscalls اضافه کنیم.

```
[SYS_find_fibonacci_number] sys_find_fibonacci_number,
```

3. در قدم بعدی باید این تابع را در usys.S تعریف کنیم.

```
SYSCALL(find_fibonacci_number)
```

همانطور که در بخش سوالات-سوال اول توضیح داده شده است در این فایل ماکرو (خط ۴ تا ۹)، خط اضافه شده را به کد زیر تبدیل میکند.

```
.globl  find_fibonacci_number\
name: \
  movl $SYS_ ## find_fibonacci_number, %eax; \
  int $T_SYSCALL; \
  ret
```

4. تعریف تابع را در فایلی جدید(test_sys_fibonacci.c) مینویسم:
   a. ابتدا تابع fibonacci_number را مینویسم که تنها وظیفه دارد با توجه به index داده شده مقدار fibonacci مربوطه را به عنوان خروجی بدهد (در واقع وظیفه این تابع تنها محاسبات میباشد):

```
static int fibonacci_number(int index) {
    int n1 = 0;
    int n2 = 1;
    int n3 = 0;
    if (index <= 0)
        return -1;
    else if (index == 1)
        return n1;
    else if (index == 2)
        return n2;
    else {
        for (int i = 2; i < index; i++) {
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;
        }
        return n3;
    }
}
```

همانطور که مشاهده می شود ابتدا چک میکنیم مقدار index ورودی معتبر باشد(عددی بزرگتر از 0) در صورت معتبر نبودن ورودی مقدار ۱- را برمیگردانیم و در غیر این صورت با توجه به مقدار ورودی، خروجی را محاسبه میکنیم.

   b. حال تابع اصلی sys_find_fibonacci_number را مینویسم که در این تابع تنها تابع fibonacci_number صدا زده میشود و آرگومان ورودی آن را همانطور که خواسته شده است با استفاده از regiter ebx پاس میدهد.

```
int sys_find_fibonacci_number(void) {
    return fibonacci_number(myproc()->tf->ebx);
}
```

c. در ادامه با توجه به اینکه یک فایل جدید ساخته ایم، باید به متغیر OBJS در Makefile، new_file_name.o که در این قسمت به صورت test_sys_fibonacci.o می باشد را اضافه کنیم.

```
30      test_sys_fibonacci.o\
```

5. در این مرحله به منظور ایجاد امکان اجرای این فراخوانی، یک برنامه در سطح کاربر باید بنویسیم. به این منظور فایل جدیدی (find_fibonacci_number.c) ایجاد میکنیم.

a. در main ابتدا چک میکنیم که آرگومان ورودی داده شده است یا خیر و سپس در صورت داده شدن آرگومان ورودی لازم از تابع کمکی call_find_fibonacci_number استفاده میکنیم که ساختار آن در ادامه توضیح داده شده است و سپس مقدار خروجی را ذخیره کرده و با توجه به اینکه ایا آرگومان به درستی داده شده است و یا خیر پیام مناسب را به عنوان خروجی پرینت میکنیم.

```c
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf(2, "Use the function as: find_fibonacci_number<number>\n");
        exit();
    }

    int index_fibonacci_num = atoi(argv[1]);
    // Save input in ebx (use register for save input)
    int fibonacci_num = call_find_fibonacci_number(index_fibonacci_num);
    if (fibonacci_num == -1)
        printf(2, "Number should be greater than 0 for example: 1, 2, 3, ...\n");
    else
        printf(1, "%d\n", fibonacci_num);

    exit();
}
```

b. همانطور که گفتیم از تابع کمکی call_find_fibonacci_number استفاده می کنیم که دراین تابع ابتدا مقدار رجیستر ebx را ذخیره میکنیم تا پس از اجرای عملیات و بدست آوردن مقدار فیبوناچی خواسته شده مقدار را بازگردانیم.
در قدم بعدی پس از ذخیره مقدار قبلی، ورودی داده شده توسط کاربر را در رجیستر ebx ذخیره میکنیم و تابع find_fibonacci_number را صدا میزنیم و مقدار خروجی را ذخیره میکنیم و سپس قبل از return کردن نتیجه مقدار ebx را به حالت قبلی برمی گردانیم.

```c
int call_find_fibonacci_number(int number) {
    int previous_ebx;

    //First Save current ebx in previous_ebx
    //Save number in ebx
    asm volatile(
        "movl %%ebx, %0\n\t"
        "movl %1, %%ebx"
        : "=r"(previous_ebx)
        : "r"(number)
    );

    //Save output of find_fibbonacci_number before restore ebx
    int fibonacci_num = find_fibonacci_number();

    // Restore last version of ebx
    asm volatile(
        "movl %0, %%ebx"
        :: "r"(previous_ebx)
    );

    return fibonacci_num;
}
```

c. در آخر باید به متغیر UPROGS در Makefile نام فایل را اضافه کنیم.

```
188         _test_find_fibonacci_number\
```

- تست و اجرای فراخوانی سیستمی اضافه شده:

1. ابتدا خروجی برای مقدار های اولیه ۱ و ۲ میسنجیم که به درستی ابتدا خروجی ۰ و سپس ۱ میگیریم.
2. سپس خروجی را برای یک مقدار معتبر میسنجیم، همانطور که میدانیم برای ۱۰ باید خروجی ۳۴ باشد که به درستی کار میکند.
3. سپس برای دو مقدار نامعتبر 0 , کمتر از صفر (۱۰-) میسنجیم که همانطور انتظار میرود برنامه به ما اخطار مناسب میدهد.
4. در آخر برنامه را بدون ورودی اجرا میکنیم که باز هم همانطور انتظار میرود برنامه به ما اخطار مناسب با این رفتار را نشان میدهد.

```
**************************
$ test_find_fibonacci_nuber 1
0
$ test_find_fibonacci_nuber 2
1
$ test_find_fibonacci_nuber 10
34
$ test_find_fibonacci_nuber 0
Number should be greater than 0 for example: 1, 2, 3, ...
$ test_find_fibonacci_nuber -10
Number should be greater than 0 for example: 1, 2, 3, ...
$ test_find_fibonacci_nuber
Use the function as: find_fibonacci_number<number>
$
```

❖ تابع find_most_caller

- لیست فایل های تغییر داده شده در هر مرحله:
  1. user.h
  2. syscall.h
  3. syscall.c
  4. usys.S
  5. proc.h
  6. proc.c
  7. Makefile
- لیست فایل های ایجاد شده در هر مرحله:
  1. sys_most_caller.c
  2. test_find_most_caller.c
- مراحل اضافه کردن این فراخوانی سیستمی:

1. ابتدا باید تابع را در user.h تعریف کنیم:

```
27  int find_most_caller(void);
```

2. در این مرحله باید فراخوانی سیستمی را در سطح کرنل پیاده سازی کنیم :

a. ابتدا باید در فایل syscall.h، شماره فراخوانی سیستمی اضافه میکنیم را اضافه کنیم،(با توجه به اینکه از شماره ۱ تا ۲۱ برای فراخوانی های سیستمی پیشفرض خود xv6 استفاده شده است و شماره ۲۲ نیز برای فراخوانی find_most_caller استفاده شده است به این فراخوانی شماره ۲۳ را نسبت می دهیم)

```
24  #define SYS_find_most_caller 23
```

b. در قدم بعدی تعریف تابع را در syscall.c در قسمت مربوطه اضافه کنیم

```
103  extern int sys_find_most_caller(void);
```

c. در اخرین قدم این مرحله باید فراخوانی سیستمی را در آرایه syscalls اضافه کنیم.

```
131      [SYS_find_most_caller]   sys_find_most_caller,
```

3. در قدم بعدی باید این تابع را در usys.S تعریف کنیم.

```
33  SYSCALL(find_most_caller)
```

همانطور که در بخش سوالات-سوال اول توضیح داده شده است در این فایل ماکرو (خط ۴ تا ۹)، خط اضافه شده را به کد زیر تبدیل میکند:

```
.globl  find_most_caller\
name: \
  movl $SYS_ ## find_most_caller, %eax; \
  int $T_SYSCALL; \
  ret
```

4. در این مرحله نیاز داریم یک آرایه تعریف کنیم که همواره تعداد دفعات فراخوانی شدن هر یک از فراخوانی های سیستمی را ذخیر کند،به این منظور در struct proc که در فایل proc.h تعریف شده است یک آرایه count_calls اضافه میکنیم.

نکته: سایز آرایه را یکی بیشتر از فراخوانی ها می گیریم چرا که شماره گذاری های فراخوانی ها از شماره ۱ شروع میشود، و ما نیز از خانه شماره 0 ام ارایه استفاده نمیکنیم.

نکته ۲: همچنین هربار که یک فراخوانی سیستمی برای xv6 تعریف میکنیم و اضافه میکنیم باید سایز این ارایه افزایش پیدا کند و با توجه به اینکه در این پروژه میدانیم قرار است دو فراخوانی دیگر اضافه شود سایز آرایه را برابر با ۲۶ می گیریم.

```
int count_calls[26];          // for find_most_caller
```

5. پس از تعریف آرایه در proc.h باید آن را مقدار دهی اولیه کنیم، به این منظور کد زیر را در فایل proc.c در تابع userinit قرار میدهیم:

```
//init the count_calls[]
p->count_calls[0] = 0;
for(int i = 1; i < sizeof(p->count_calls) / sizeof(p->count_calls[0]); i++) {
  p->count_calls[i] = 0;
}
```

6. در این مرحله باید هربار که یک فراخوانی سیستمی داریم ارائه count_calls را آپدیت کنیم،به این منظور قطعه کد زیر را در تابع syscall در فایل syscall.c قرار میدهیم، در این صورت هربار که یک فراخوانی سیستمی داریم با توجه به شماره فراخوانی سیستمی مقدار خانه با index برابر با شماره فراخوانی سیستمی یکی بیشتر میشود.

```
//update count_calls:
int pre_count = curproc->count_calls[num];
curproc->count_calls[num] = pre_count + 1;
```

7. تعریف تابع را در فایلی جدید(sys_most_caller.c) مینویسیم:

a. ابتدا تابع find_index_maximum را مینویسم که وظیفه دارد با توجه به آرایه ورودی مقدار maximum در آرایه را بدست بیاورد و سپس مقدار index را برگرداند.

```
static int find_index_maximum(int *counts, int num) {
    int maximum = 0;
    int index_max = 0;
    for (int i = 1; i < num; i++) {
        //cprintf("%d: %d\n",i , counts[i]);
        if (counts[i] >= maximum) {
            maximum = counts[i];
            index_max = i;
        }
    }
    return index_max;
}
```

b. حال تابع اصلی sys_find_most_caller را مینویسم که در این تابع تنها تابع find_index_maximum صدا زده میشود و آرگومان ورودی آن را همان آرایه ای که از قبل تعریف کرده ایم، به همراه طول آرایه میدهد.

```
int sys_find_most_callee(void) {
    return find_index_maximum(myproc()->count_calls, sizeof(myproc()->count_calls) / sizeof(myproc()->count_calls[0]));
}
```

c. در ادامه با توجه به اینکه یک فایل جدید ساخته ایم، باید به متغیر OBJS در Makefile، new_file_name.o که در این قسمت به صورت test_sys_most_caller.o میباشد را اضافه کنیم.

```
31        test_sys_most_caller.o\
```

8. در این مرحله به منظور ایجاد امکان اجرای این فراخوانی، یک برنامه در سطح کاربر باید بنویسیم. به این منظور فایل جدیدی (test_find_most_caller.c) ایجاد میکنیم.
   a. در این فایل کافی است در main، تابع find_most_caller را صدا زده و نتیجه را پرینت کنیم.

```
5   int main(int argc, char* argv[]) {
6       int most = find_most_caller();
7       printf(1, "%d\n", most);
8       exit();
9   }
```

   b. در آخر باید به متغیر UPROGS در Makefile نام فایل را اضافه کنیم.

```
189         _test_find_most_caller\
```

- تست و اجرای فراخوانی سیستمی اضافه شده:

فراخوانی را دوبار اجرا میکنیم که نتایج به صورت زیر می باشد:

```
**************************
$ test_find_most_caller
23
$ test_find_most_caller
16
$
```

در اولین فراخوانی مقدار ۲۳ چاپ میشود که شماره مربوط به فراخوانی سیستمی find_most_caller است که صدا زده ایم و واضح است تا به حال تنها این فراخوانی را خودمان دستی صدا زده ایم البته تعدادی فراخوانی نیز برای اجرای این فراخوانی نیز صدا زده شده اند(یا همراه وقبل از این فراخوانی و یا مانند فراخوانی exit پس از اتمام این فراخوانی صدا زده شده اند)که در کد ما چون به صورتی نوشته شده است که درصورت مساوی بودن تعداد فراخوانی کردن دو یا بیشتر فراخوانی سیستمی با بیشترین تعداد، شماره فراخوانی با شماره بیشتر را برگرداند، و البته اگر تساوی را در خط فایل sys_most_caller.c برداریم نتیجه اولین بار فراخوانی برابر با ۷ میشود و نه ۲۳ که مربوط به فراخوانی سیستمی exec میباشد که دلیل این موضوع در ادامه آمده است.

در دومین بار صدا زدن این فراخوانی (چه با وجود تساوی و یا بدون وجود تساوی)خروجی برای با شماره ۱۶ میباشد که مربوط به فراخوانی write است و دلیل این موضوع نیز در ادامه آمده است.
در انتها برای شفاف سازی بهتر درستی عملکرد این تابع از تابع کمکی cprintf استفاده می کنیم. در همان تابع find_index_maximum این تابع را استفاده کرده و مقدار تمام خانه های آرایه را قرار میدهیم (این خط کامنت شده است) و به این صورت به ترتیب برای دو بار اجرا خروجی های زیر را میگیریم که میتوان با توجه خروجی ها صحت دو موضوع را بررسی کنیم:

1. شمارش درست تعداد بارهای فراخوانی
2. عملکرد درست تابع find_index_maximum

صحت عملکرد مورد دوم: کافی است بزرگترین مقدار(با بیشترین ایندکس)را یافته و و ایندکس را ه مقدار خروجی مقایسه کنیم که درصورت مقایسه کاملا واضح است که هر دو یکسانند و تابع به درستی کار میکند. (نتایج در عکس هایی که در ادامه آمده اند.)
صحت عملکرد مورد اول: برای چند ایندکس که مقدارشان صفر نیست، تعداد بار های اجرا را بررسی میکنیم و با خروجی مقایسه می کنیم: (عکس ها در آخر توضیحات آمده اند.)

- اجرا اول:

7: exec:

- همانطور که میدانیم exec برای run کردن یک executable file استفاده میشود که در هنگام استفاده از فراخوانی سیستمی find_most_caller، از exec نیز استفاده میشود و در نتیجه تعداد این فراخوانی با هربار صدا زدن find_most_caller اضافه میشود.

12: sbrk:

- تعداد این فراخوانی نیز با هربار صدا زدن یک فراخوانی افزایش پیدا میکند چرا که برای هر بار اجرای یک برنامه لازم است sbrk، حافظه فیزیکی را allocate کندو به ادرس مجازی برنامه map کند.

23: find_most_caller:

- این فراخوانی را خودمان صدا زده ایم و واضح است چرا تعداد اجرای آن افزایش یافته است.

- اجرا دوم:

2: exit:

- دلیل افزایش تعداد دفعات فراخوانی exit به این دلیل است که پس از یک بار صدا زدن find_most_caller در مرحله قبل پس از اتمام کار از فراخوانی exit استفاده کرده ایم تا برنامه را terminate کنیم.

7: exec: بحث شده است.

12: sbrk: بحث شده است.

16: write: با توجه به استفاده از printf استفاده شده در مرحله قبل و غیره قابل توجیه است.

23: find_most_caller: بحث شده است.

* پس توانستیم صحت عملکرد این تابع را نشان دهیم.

اجرا اول / اجرا دوم

```
$ test_find_most_caller
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 1
8: 0
9: 0
10: 0
11: 0
12: 1
13: 0
14: 0
15: 0
16: 0
17: 0
18: 0
19: 0
20: 0
21: 0
22: 0
23: 1
24: 0
25: 0
23
$
```

```
$ test_find_most_caller
1: 0
2: 1
3: 0
4: 0
5: 0
6: 0
7: 2
8: 0
9: 0
10: 0
11: 0
12: 2
13: 0
14: 0
15: 0
16: 3
17: 0
18: 0
19: 0
20: 0
21: 0
22: 0
23: 2
24: 0
25: 0
16
$
```

❖ تابع get_alive_childrn_count

- لیست فایل های تغییر داده شده در هر مرحله:

user.h.1
syscall.h.2
syscall.c.3
usys.S.4
proc.h.5
proc.c.6

Makefile.7

- لیست فایل های ایجاد شده در هر مرحله:

1. test_get_alive_children_count.c

- مراحل اضافه کردن این فراخوانی سیستمی:

1. نگاشت شماره فراخوانی در فایل syscall.h(با توجه به توضیحات داده شده برای پیاده سازی سیستم کالی قبلی شماره 24 به این سیستم کال اختصاص می یابد.)

```
25  #define SYS_get_alive_children_count  24
```

2. تعریف تابع در فایل syscall.c

```
104  extern int sys_get_alive_children_count(void);
```

3. اضافه کردن به آرایه syscalls در فایل syscall.c

```
132    [SYS_get_alive_children_count] sys_get_alive_children_count,
```

4. اضافه کردن تابع در فایل user.h

```
28  int get_alive_children_count(int);
```

5. اضافه کردن تابع فراخواننده در فایل sysproc.c

```
94  int
95  sys_get_alive_children_count(void)
96  {
97    int pid;
98    if(argint(0, &pid) < 0)
99      return -1;
100   return get_alive_children_count(pid);
101 }
```

6. پیاده سازی تابع در فایل proc.c

```
543 get_alive_children_count(int parent_pid)
544 {
545   struct proc *p;
546   int alive_children_count = 0;
547
548   acquire(&ptable.lock);
549   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
550   {
551     if (p->parent->pid == parent_pid && p->killed == 0)
552       alive_children_count += 1;
553   }
554   release(&ptable.lock);
555
556   return alive_children_count;
557 }
```

..

7. تعریف تابع مربوطه در فایل usys.S

```
34  SYSCALL(get_alive_children_count)
```

- تست و اجرای فراخوانی سیستمی اضافه شده:

1. برای تست کردن این سیستم کال فایل test_get_alive_children_count.c را کنار کدها قرار دادیم. روند اجرای این فایل تست به وسیله log های مختلفی که در حین اجرا چاپ می شود مشخص است.

```c
#include "types.h"
#include "stat.h"
#include "user.h"

void
print_curr_info(char* name, int curr_pid, int flag)
{
  if(flag)
  {
    printf(1, name);
    printf(1, " = %d\n", curr_pid);
  }
  printf(1,"current info: \n");
  printf(1,"~~ current PID = %d\n", getpid());
  printf(1, "~~ alive_children_count[%d] = %d \n", \
  getpid(), get_alive_children_count(getpid()));
  printf(1, "=======================================\n");
}

int
main(int argc, char *argv[])
{
  printf(1, "test of <<get_alive_children_count>> syscall\n\n");

  print_curr_info("Parent", getpid(), 1);
  printf(1, "call A=fork()\n");
  int A = fork();
  if(A)// if A = 0 => getpid() = child but if A > 0 => getpid() = parent
  {
    print_curr_info("A", A, 1);
    printf(1, "call B=fork()\n");
    int B = fork();
    if(B)
    {
      print_curr_info("B", B, 1);
      printf(1, "call C=fork()\n");
      int C = fork();
      if(C)
      {
        print_curr_info("C", C, 1);
      }
    }
  }
  while (wait() != -1);
  exit();
}
```

2. در makefile در قسمت UPROGS این فایل تعریف میشود.

```
188     _test_get_alive_children_count\
```

3. خروجی اجرای تست این سیستم کال را در ادامه مشاهده میکنید.

```
************************
Group #5
Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
************************
$ test_get_alive_children_count
test of <<get_alive_children_count>> syscall

Parent = 3
current info:
~~ current PID = 3
~~ alive_children_count[3] = 0
=======================================
call A=fork()
A = 4
current info:
~~ current PID = 3
~~ alive_children_count[3] = 1
=======================================
call B=fork()
B = 5
current info:
~~ current PID = 3
~~ alive_children_count[3] = 2
=======================================
call C=fork()
C = 6
current info:
~~ current PID = 3
~~ alive_children_count[3] = 3
=======================================
$
```

❖ تابع kill_first_child_process

- لیست فایل های تغییر داده شده در هر مرحله:
  1. user.h
  2. syscall.h
  3. syscall.c
  4. usys.S
  5. proc.h
  6. proc.c
  7. Makefile
- لیست فایل های ایجاد شده در هر مرحله:
  1. test_kill_first_child_process.c
- مراحل اضافه کردن این فراخوانی سیستمی:
  1. نگاشت شماره فراخوانی در فایل syscall.h(با توجه به توضیحات داده شده برای پیاده سازی سیستم کالی قبلی شماره 25 به این سیستم کال اختصاص می یابد.)

```
26   #define SYS_kill_first_child_process  25
```

2. تعریف تابع در فایل syscall.c

```
105   extern int sys_kill_first_child_process(void);
```

3. اضافه کردن به آرایه syscalls در فایل syscall.c

```
133     [SYS_kill_first_child_process] sys_kill_first_child_process,
```

4. اضافه کردن تابع در فایل user.h

```
105   extern int sys_kill_first_child_process(void);
```

5. اضافه کردن تابع فراخواننده در فایل sysproc.c

```
104   sys_kill_first_child_process(void)
105   {
106     int pid;
107     if(argint(0, &pid) < 0)
108       return -1;
109
110     kill_first_child_process(pid);
111     return 1;
112   }
```

6. پیاده سازی تابع در فایل proc.c

```
560   kill_first_child_process(int parent_pid)
561   {
562     struct proc *p;
563     cprintf("{\n");
564     // acquire(&ptable.lock);
565     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
566     {
567       if (p->parent->pid == parent_pid && p->killed==0)
568       {
569         cprintf("   first alive child PID = %d\n", p->pid);
570         cprintf("   kill(%d)\n", p->pid);
571         kill(p->pid);
572         break;
573       }
574     }
575     // release(&ptable.lock);
576     cprintf("}\n");
577   }
```

7. تعریف تابع مربوطه در فایل usys.S

```
35   SYSCALL(kill_first_child_process)
```

- تست و اجرای فراخوانی سیستمی اضافه شده:

1. برای تست کردن این سیستم کال فایل test_kill_first_child_process.c را کنار کدها قرار دادیم.
روند اجرای این فایل تست به وسیله log های مختلفی که در حین اجرا چاپ می شود مشخص است.

```c
#include "types.h"
#include "stat.h"
#include "user.h"

void
print_curr_info(char* name, int curr_pid, int flag)
{
  if(flag)
  {
    printf(1, name);
    printf(1, " = %d\n", curr_pid);
  }
  printf(1,"current info: \n");
  printf(1,"~~ current PID = %d\n", getpid());
  printf(1, "~~ alive_children_count[%d] = %d \n", \
  getpid(), get_alive_children_count(getpid()));
  printf(1, "=======================================\n");
}

int
main(int argc, char *argv[])
{
  printf(1, "test of <<kill_first_child_process>> syscall\n\n");

  int parent_PID = getpid();
  print_curr_info("Parent", parent_PID, 1);
  printf(1, "call A=fork()\n");
  int A = fork();
  if(A) // if A = 0 => getpid() = child but if A > 0 => getpid() = parent
  {
    sleep(10);
    print_curr_info("A", A, 1);
    printf(1, "call B=fork()\n");
    int B = fork();
    if(B)
    {
      sleep(10);
      print_curr_info("B", B, 1);
      printf(1, "call C=fork()\n");
      int C = fork();
```

```
41          if (C)
42          {
43            sleep(10);
44            print_curr_info("C", C, 1);
45            printf(1, "parent(PID = %d)'s children count = %d\n",
46            parent_PID, get_alive_children_count(parent_PID));
47            printf(1, "call kill_first_alive_child_process(%d)\n", parent_PID);
48            kill_first_child_process(parent_PID);
49            printf(1, "parent(PID = %d)'s children count = %d\n",
50            parent_PID, get_alive_children_count(parent_PID));
51            print_curr_info("C", C, 0);
52          }
53        }
54      }
55      while (wait() != -1);
56      exit();
57    }
58
```

2. در makefile در قسمت UPROGS این فایل تعریف میشود.

```
189        _test_kill_first_child_process\
```

3. خروجی اجرای تست این سیستم کال را در ادامه مشاهده میکنید.

```
************************
Group #5
Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
************************
$ test_kill_first_child_process
test of <<kill_first_child_process>> syscall

Parent = 3
current info:
~~ current PID = 3
~~ alive_children_count[3] = 0
=====================================
call A=fork()
A = 4
current info:
~~ current PID = 3
~~ alive_children_count[3] = 1
=====================================
call B=fork()
B = 5
current info:
~~ current PID = 3
~~ alive_children_count[3] = 2
=====================================
call C=fork()
C = 6
current info:
~~ current PID = 3
~~ alive_children_count[3] = 3
=====================================
parent(PID = 3)'s children count = 3
call kill_first_alive_child_process(3)
{
   first alive child PID = 4
   kill(4)
}
parent(PID = 3)'s children count = 2
current info:
~~ current PID = 3
~~ alive_children_count[3] = 2
=====================================
$
```