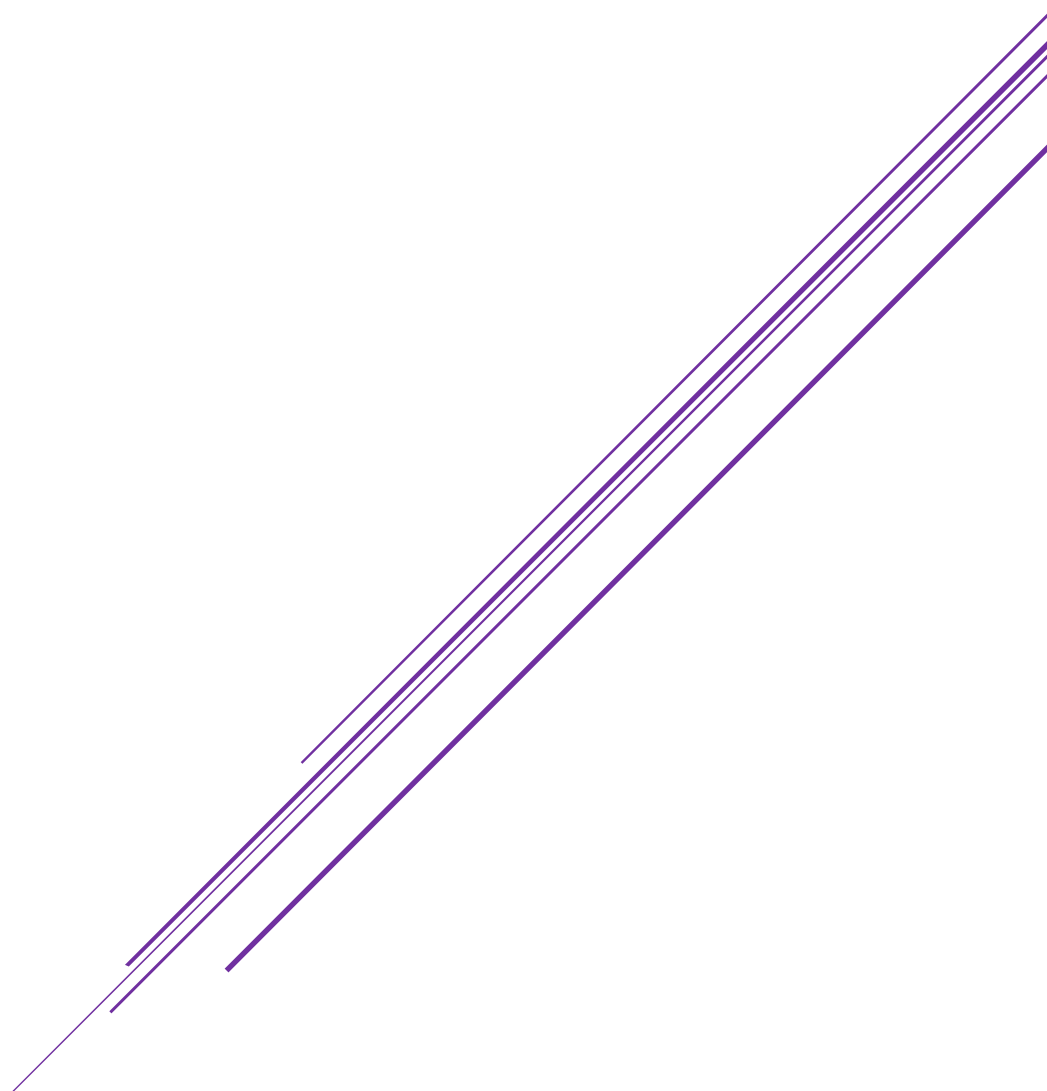


آزمایشگاه سیستم عامل

تمرین کامپیوتری ۲



اعضای گروه :

علی حمزه پور - ۸۱۰۱۰۰۱۲۹

نرگس سادات سیدحائری - ۸۱۰۱۰۰۱۶۵

مینا شیرازی - ۸۱۰۱۰۰۲۵۰

سوال یک: کتابخانه‌های سطح کاربر استفاده‌شده در xv6 را از منظر استفاده از فراخوانی‌های سیستمی و علت استفاده از آن‌ها را بررسی کنید.

در makefile چهار آبجکت فایل مربوط به ULIB داریم که کتابخانه‌های سطح کاربر هستند. این کتابخانه‌ها مسئولیت پیاده‌سازی فانکشن‌های متداولی که در برنامه‌های سطح کاربر را دارند و در حقیقت یک wrapper برای فراخوانی‌های سیستمی‌ست. ۴ فایل مربوط به ULIB به شکل زیر هستند:

• Ulib:

این کتابخانه شامل توابعی برای کار با استرینگ، آرایه‌ها و i/o است. این توابع در این فایل از فراخوانی‌های سیستمی استفاده می‌کنند:

۱. gets: در این تابع از سیستم‌کال read استفاده می‌شود تا از stdin ورودی خوانده شود.

```
char*
gets(char *buf, int max)
{
    int i, cc;
    char c;

    for(i=0; i+1 < max; ){
        cc = read(0, &c, 1);
        if(cc < 1)
            break;
        buf[i++] = c;
        if(c == '\n' || c == '\r')
            break;
    }
    buf[i] = '\0';
    return buf;
}
```

۲. stat: در این تابع از فراخوانی‌های open و fstat و close برای گرفتن اطلاعات یک فایل استفاده می‌شود.

```
int
stat(const char *n, struct stat *st)
{
    int fd;
    int r;

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

• **:usys**

در این فایل، برای هر سیستم‌کال کد اسمبلی برای ساخت اینتراپت آن سیستم‌کال وجود دارد. یک ماکروی SYSCALL تعریف شده که به شکل زیر است:

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

این ماکرو اسم سیستم‌کال رو می‌گیرد و شماره‌ی آن را به رجیستر `eax` منتقل می‌کند، سپس یک اینتراپت از نوع سیستم‌کال اجرا می‌کند.

• **:printf**

در این فایل از فراخوانی سیستمی `write` برای پیاده‌سازی توابع آن استفاده می‌شود.

```
static void
putc(int fd, char c)
{
    write(fd, &c, 1);
}
```

• :umalloc

در این فایل پیاده‌سازی توابع مربوط به کنترل حافظه مانند malloc و free انجام شده‌است که در آن از سیستم‌کال sbrk استفاده می‌شود.

```
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;
}
```

سوال دو: انواع روش‌های دسترسی از سطح کاربر به سطح هسته را در لینوکس توضیح دهید.

دسترسی به سطح هسته از طریق اینترپت‌ها صورت می‌گیرد که اینترپت می‌تواند نرم‌افزاری و یا سخت‌افزاری باشد.

اینترپت سخت‌افزاری همان‌طور که از اسمش پیداست، توسط سخت‌افزارها و دیوایس‌های I/O صورت می‌گیرد. اینترپت نرم‌افزاری زمانی رخ می‌دهد که از در برنامه‌ی سطح کاربر از سیستم‌کال استفاده کنیم. همچنین زمانی که در استثنایی مانند تقسیم بر صفر یا دسترسی غیر مجاز به حافظه رخ دهد هم اینترپت صورت می‌گیرد. برنامه‌های سطح کاربر هم می‌توانند از طریق سیگنال‌ها با هم ارتباط داشته‌باشند که در اجرای آن‌ها نیز اینترپت نرم‌افزاری رخ می‌دهد.

همچنین با استفاده از pseudo file systems هم می‌توان به سطح هسته دسترسی پیدا کرد. در این روش به اطلاعات داده‌ساختارهای سطح هسته از طریق ساختارهای فایل‌مانند می‌توان دسترسی پیدا کرد.

سوال سه: آیا همه تله‌ها را می‌شود با سطح دسترسی DPL_USER فعال نمود؟

در xV6، تلاش برای فعال کردن یک تله با سطح دسترسی USER_DPL برای تله دیگری باعث بروز یک protection exception می‌شود. این تدابیر امنیتی برای جلوگیری از مشکلات ممکن در برنامه‌های کاربری یا اقدامات خبیث اعمال شده است. اجازه دادن به کاربران برای اجرای تله‌ها با سطوح دسترسی بالاتر می‌تواند یک خطر جدی امنیتی ایجاد کند زیرا این اقدام ممکن است دسترسی غیرمجاز به هسته را فراهم کند و به تخریب کلی امنیت سیستم منجر شود. سخت‌افزار توسط معماری x86 کنترل می‌شود و این سطوح دسترسی را اجرا می‌کند تا جدایی روشنی بین حالت کاربر و هسته حفظ شود و استثناء حفاظتی در صورت نقض این مراحل اجرایی بوجود آید.

سوال چهار: در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیر اینصورت Push نمیشود. چرا؟

وقتی که سطح دسترسی (Privilege Level) تغییر می‌کند، ممکن است مواردی مثل پشته نیاز به تغییر داشته باشند. در معماری ESP، x86 به عنوان اشاره‌گر به قسمت بالایی پشته استفاده می‌شود و SS نشان‌دهنده‌ی میزان دسترسی به پشته است. هنگامی که سطح دسترسی تغییر می‌کند (برای مثال، وارد حالت کاربری می‌شویم یا از کد کاربر به کد سیستم عامل منتقل می‌شویم)، اطلاعات مربوط به پشته نیاز به تغییر دارند. در صورت تغییر سطح دسترسی، به ویژه هنگام رخ دادن یک trap، رجیسترهای "ss" و "esp" بر روی استک قرار داده می‌شوند. این فرآیند برای تسهیل انتقال از استک کاربر به استک هسته حائز اهمیت است. دلیل این عمل مربوط به وجود دو استک است - استک کاربر و استک هسته. در زمان تغییر سطح دسترسی، به عنوان مثال در حین انتقال از

حالت کاربر به حالت هسته، سیستم نیاز دارد که از استک هسته برای دسترسی به کد و ساختارهای داده‌ای که در دامنه هسته قرار دارند، استفاده کند. ابتدا، مقادیر فعلی "esp" و "ss" که به استک کاربر اشاره دارند، بر روی استک ذخیره می‌شوند. این مقادیر ذخیره شده بعداً برای اشاره به استک هسته استفاده می‌شوند و این امکان را فراهم می‌کنند که کد هسته اجرا شود و به ساختارهای داده‌ای درون هسته دسترسی پیدا کند. پس از پردازش trap یا نقص، مقادیر قدیمی "esp" و "ss" بازیابی می‌شوند و این امکان را فراهم می‌کنند تا برنامه کاربر بی‌مشکل از جایی که متوقف شده بود ادامه یابد. این فرآیند اطمینان از صحت محیط اجرایی در طی انتقال بین حالت‌های کاربر و هسته را فراهم می‌کند. مهم است که توجه داشته باشیم که اگر تغییری در سطح دسترسی رخ ندهد، به عبارت دیگر، اگر برنامه همچنان با همان استک کار کند، نیازی به ذخیره و بازیابی "esp" و "ss" وجود ندارد. این بهینه‌سازی جلوی انجام عملیات غیرضروری را هنگام عدم تغییر در سطح امتیاز می‌گیرد.

سوال پنج: در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr() بازه آدرسها بررسی می‌گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ 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.

چهار تابع برای دسترسی به پارامترهای فراخوانی وجود دارند:

۱. **argptr**: در سیستم عامل xv6، تابع "argptr" برای بررسی صحت بازه آدرس‌های پارامترهای ارسالی به توابع فراخوانی سیستمی استفاده می‌شود. این تابع بررسی می‌کند که بازه آدرس ارائه شده در محدوده آدرس‌های قابل دسترس و معتبر در فضای آدرس کاربر است یا خیر. با انجام این بررسی، از وقوع آسیب‌پذیری‌های امنیتی و دسترسی غیرمجاز به مناطق حافظه جلوگیری می‌شود.
۲. **argint**: برای به دست آوردن یک عدد صحیح از فضای کاربری استفاده می‌شود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار آرگومان را از فضای کاربری به فضای کرنل منتقل می‌کند. اگر عملیات با موفقیت انجام شود، مقدار ۰ را برمی‌گرداند؛ در غیر این صورت، ۱- برمی‌گرداند.
۳. **argstr**: برای بازیابی یک رشته از فضای کاربری پردازش استفاده می‌شود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار رشته را از فضای کاربری به فضای کرنل کپی می‌کند. اگر عملیات با موفقیت انجام شود، مقدار ۰ را برمی‌گرداند؛ در غیر این صورت، ۱- برمی‌گرداند.
۴. **argfd**: در xv6 برای بازیابی فایل دسکریپتور از فضای کاربری پردازش استفاده می‌شود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار فایل دسکریپتور را از فضای کاربری به فضای کرنل کپی می‌کند. در صورت موفقیت، مقدار ۰ را برمی‌گرداند؛ در غیر این صورت، ۱- برمی‌گرداند.

تمامی این توابع بررسی می کنند که آدرس داده شده حتما در حافظه پردازش قرار گیرد که یک پردازش نتواند به حافظه پردازش دیگری دسترسی پیدا کند زیرا این اتفاق ممکن است باعث مشکلات امنیتی در پردازش های دیگر شود.

تجاوز از بازه معتبر آدرس در x86 می تواند به نقض حفاظت حافظه، اجرای کد بد، و تخریب داده ها منجر شود. که این موضوع می تواند اجرای برنامه را دچار مشکل کند.

برای مثال می توانیم فراخوانی سیستمی sys_read را بررسی کنیم. این فراخوانی سیستمی مربوط به تابع read است:

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

در این تابع آرگومان دوم بافری است که مقدار خوانده شده در آن قرار می گیرد و آرگومان سوم برابر است با حداکثر تعداد بایت هایی که قرار است خوانده شود. در صورتی که سیستم عامل پیش از خواندن این تعداد بایت به EOF برسد، عملیات خواندن از فایل را پایان می دهد. تابع sys_read به صورت زیر تعریف 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);
}
```

تابع مذکور در ابتدا با استفاده از تابع "argint"، مقدار "fd" که آرگومان اول تابع "read" است را دریافت می کند و معتبر بودن این شماره فایل را بررسی می کند. سپس با استفاده از تابع "argint"، مقدار "max" (آرگومان سوم) را دریافت می کند. در نهایت، با استفاده از تابع "argptr"، بررسی می کند که کل فضای آدرس دهی از ابتدای پوینتر به بافر (آرگومان دوم) تا انتهای آن (به طول "max")، در حافظه پردازش قرار گیرد.

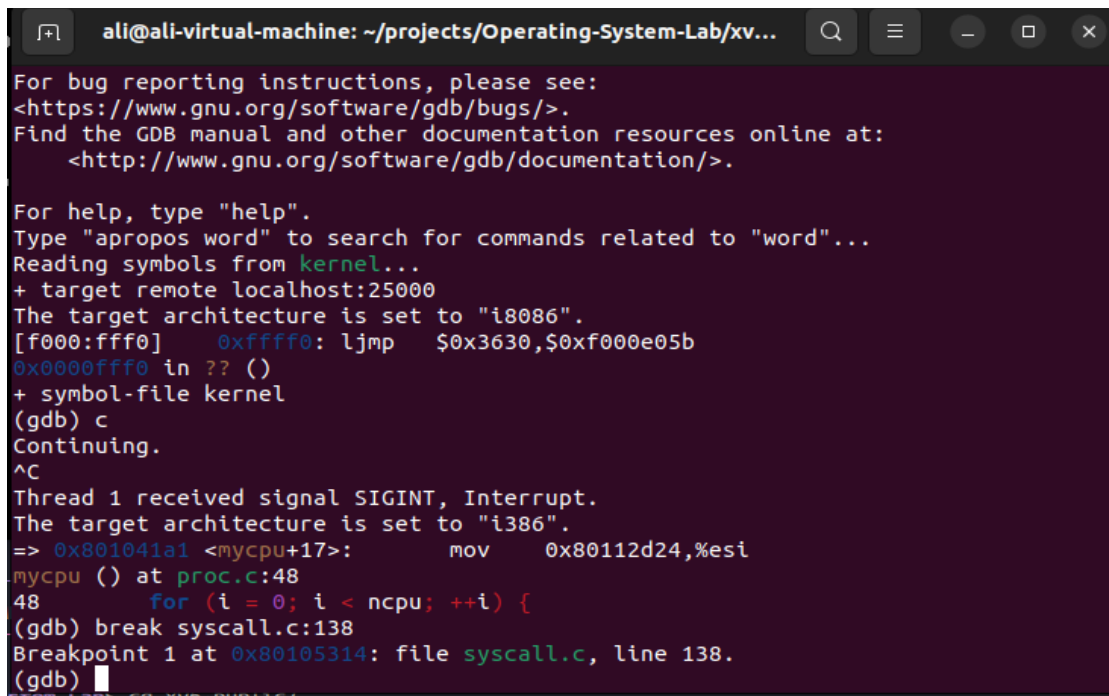
اگر این بررسی انجام نمی شد، ممکن بود در یک برنامه از تابع "read" با مقدار "max" بزرگ و برای فایلی بزرگ استفاده شود. در این صورت، هنگام خواندن از فایل و نوشتن در بافر، سیستم عامل ممکن است از حافظه پردازش خارج شود و در حافظه پردازش دیگری شروع به نوشتن کند. این ممکن است باعث رخ دادن مشکلات بسیار زیادی شود. البته، در صورتی که مقدار "max" از طول بافر بیشتر باشد ولی از حافظه پردازش خارج نشود، همچنان می تواند باعث بروز خطاهای overflow در بافر و در نتیجه خطا در پردازش شود.

بررسی گام‌های اجرای فراخوانی سیستمی در سطح هسته توسط gdb

ابتدا برنامه‌ای در سطح کاربر به اسم getpidtest.c می‌نویسیم که pid پردازشی فعلی را با استفاده از سیستم‌کال getpid پیدا کند و نمایش دهد:

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

سپس سیستم‌عامل را در حالت gdb اجرا می‌کنیم و gdb را به آن متصل می‌کنیم. بعد از بوت شدن سیستم‌عامل یک برک‌پوینت در فایل syscall.c و در خط ۱۳۸ می‌گذاریم تا بتوانیم شماره‌ی سیستم‌کال را در آن نقطه ببینیم:



```
ali@ali-virtual-machine: ~/projects/Operating-System-Lab/xv...
For bug reporting instructions, please see:
<https://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...
+ target remote localhost:25000
The target architecture is set to "i8086".
[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
The target architecture is set to "i386".
=> 0x801041a1 <mycpu+17>:      mov     0x80112d24,%esi
mycpu () at proc.c:48
48      for (i = 0; i < ncpu; ++i) {
(gdb) break syscall.c:138
Breakpoint 1 at 0x80105314: file syscall.c, line 138.
(gdb)
```


بعد از انجام این کار `continue` می‌کنیم و در کنسول سیستم‌عامل دستور `getpidtest` را می‌نویسیم تا برنامه اجرا شود. در این زمان در برگ‌پوینت متوقف می‌شویم. همانطور که خواسته‌شده دستور `bt` را اجرا می‌کنیم. دستور `bt` (که اختصار یافته‌ی `backtrace` است) توالی توابع فراخوانده‌شده را نشان می‌دهد (قبل از این کار دستور `layout src` را هم استفاده کرده بودیم تا نقطه‌ای از کد که روی آن هستیم را هم نشان دهد):

```

syscall.c
135 struct proc *curproc = myproc();
136
137 num = curproc->tf->eax;
138 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num]();
140 } else {
141     cprintf("%d %s: unknown sys call %d\n",
142         curproc->pid, curproc->name, num);
143     curproc->tf->eax = -1;
144 }
145 }
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168

remote Thread 1.1 in: syscall
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x00100040 in trap (tf=0x0dffe4b4) at trap.c:43
#2  0x001000ef in alltraps () at trapasm.S:20
#3  0x0dffe4b4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) s

```

همانطور که می‌بینید این توالی توابع صدا زده شده تا به برگ‌پوینت رسیدیم:

۱. **تابع `alltraps`:** این تابع `trap frame` مربوط به پردازش را می‌سازد که در آن اطلاعات مربوط به پردازش ذخیره می‌شود که بعد از رسیدگی به `trap`، بتوانیم به روال اجرا برگردیم. سپس این تابع، `trap` را صدا می‌زند.

۲. **تابع `trap`:** نوع `trap` را شناسایی می‌کند و `handler` مربوطه را صدا می‌زند که در این مثال، `syscall` را صدا می‌زند.

۳. **تابع `syscall`:** این تابع شماره‌ی سیستم‌کال را چک می‌کند و تابع مربوط به آن سیستم‌کال را صدا می‌زند تا به آن سیستم‌کال رسیدگی شود.

برای جابه‌جا شدن بین توالی توابع در `bt` از دستور `down` و `up` می‌توانیم استفاده کنیم. دستور `down` به تابع درونی‌تر در استک می‌رود و دستور `up` به تابع بیرونی‌تر منتقل می‌شود.

در این لحظه اگر از دستور down استفاده کنیم، چون از قبل در درونی‌ترین تابع صدا زده شده هستیم، به تابع دیگری منتقل نمی‌شویم و gdb این موضوع را به ما اطلاع می‌دهد:

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

برای رفتن به تابع syscall می‌توانیم از دستور up استفاده 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();
61     }
62 }

remote Thread 1.1 In: trap
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x8010634d in trap (tf=0x8dffe4b4) at trap.c:43
#2  0x801060ef in alltraps () at trapasm.S:20
#3  0x8dffe4b4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) up
#1  0x8010634d in trap (tf=0x8dffe4b4) at trap.c:43
(gdb)
```

در این نقطه برای نمایش محتویات رجیستر eax از دستور print استفاده می‌کنیم:

```
(gdb) print myproc()->tf->eax
$2 = 5
(gdb)
```

در رجیستر `eax`، شماره‌ی سیستم‌کال صدازده‌شده قرار دارد و همانطور که مشاهده می‌کنید مقدار فعلی آن با شماره‌ی سیستم‌کال `getpid` (که ۱۱ است متفاوت است). دلیل آن این است که کنسول اول با استفاده از سیستم‌کال `read` (که شماره‌ی آن همین ۵ است) می‌خواهد ورودی را بخواند. چندین بار `continue` می‌کنیم و شماره‌ی سیستم‌کال را بررسی می‌کنیم تا زمانی‌که به سیستم‌کال شماره‌ی ۱۱ برسیم:

```
(gdb) p num
$3 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>: lea -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$4 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>: lea -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$5 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>: lea -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$6 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>: lea -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$7 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>: lea -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$8 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>: lea -0x1(%eax),%edx
```

```

(gdb) p num
$9 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$10 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$11 = 5
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$12 = 1
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$13 = 3
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$14 = 12
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138

```

```

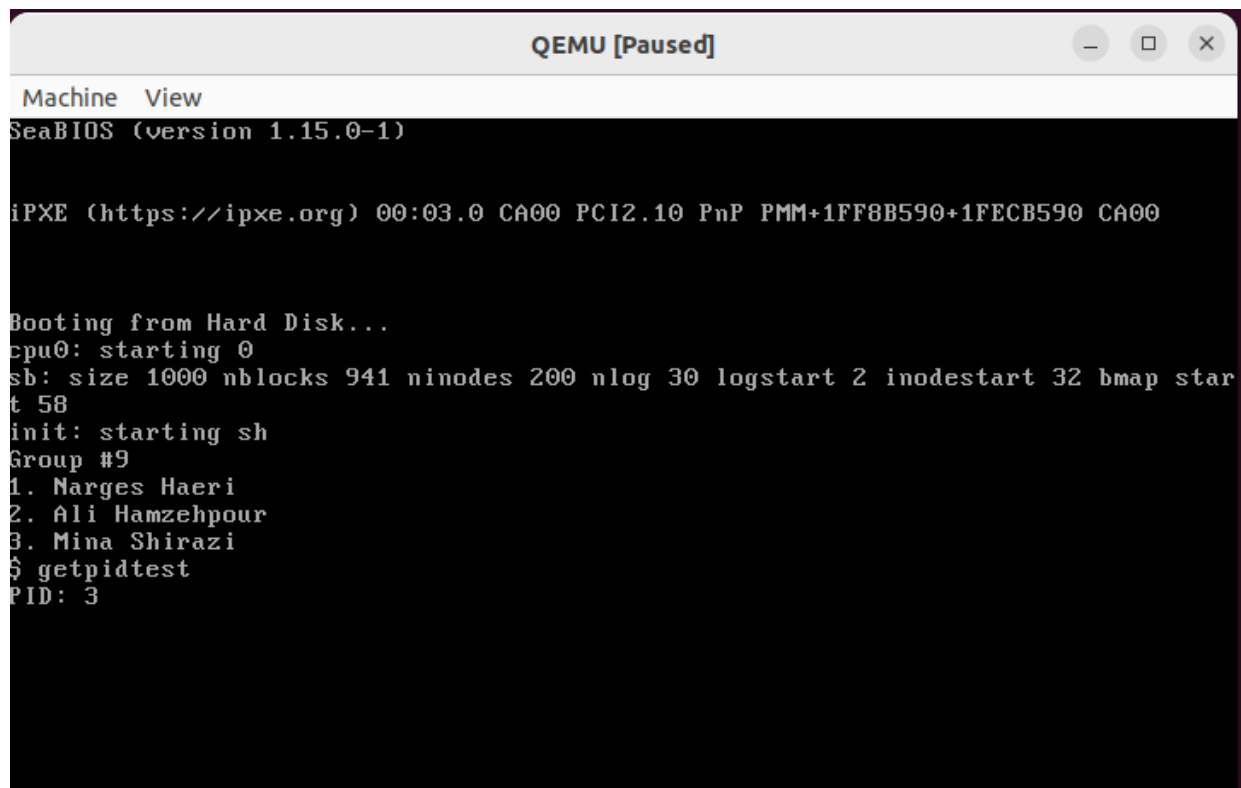
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$15 = 7
(gdb) c
Continuing.
=> 0x80105314 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$16 = 11

```

همانطور که در عکس مشخص است، ابتدا چندین بار سیستم کال read (شماره‌ی ۵) صدا زده می‌شود. سپس سیستم کال fork (شماره‌ی ۱) صدا زده می‌شود تا پردازشی جدیدی برای برنامه‌ی سطح کاربر درست شود. در ادامه سیستم کال wait (شماره‌ی ۳) آن پردازشی که fork را انجام داده بود، برای تمام شدن پردازشی فرزند صدا زده می‌شود. بعد سیستم کال sbrk (شماره‌ی ۱۲) را می‌بینیم، که برای تخصیص حافظه به پردازش است. سپس سیستم کال exec (شماره‌ی ۷) صدا زده می‌شود تا برنامه‌ی سطح کاربر در پردازش اجرا شود. در نهایت می‌بینیم که سیستم کال getpid صدا زده می‌شود و با دستور continue خروجی برنامه را در کنسول می‌بینیم:



```

Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ getpidtest
PID: 3

```

ارسال آرگومانهای فراخوانی‌های سیستمی

ابتدا برای اضافه کردن سیستم کال جدید مراحل زیر را انجام می‌دهیم:
تابع find_digital_root را در user.h تعریف می‌کنیم تا در سطح کاربر بتوان از آن استفاده کرد.

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

چون قرار است از طریق رجیستر ورودی را بخوانیم، پارامتری برای ورودی تابع نمی‌گذاریم.

سپس تعریف تابع را در usys.S میاوریم:

SYSCALL(find_digital_root)

این ماکرو همانطور که در سوالات توضیح دادیم، عدد سیستم‌کال را در eax ذخیره می‌کند و یک اینتراپت از نوع سیستم‌کال ایجاد می‌کند.

حالا در فایل syscall.h شماره‌ی سیستم‌کال جدید را مشخص می‌کنیم:

```
#define SYS_find_digital_root ۲۲
```

در نهایت در فایل syscall.c سیستم‌کال را به آرایه‌های سیستم‌کال‌ها اضافه می‌کنیم و تابع سیستم‌کال را نیز معرفی می‌کنیم:

```
[SYS_get_uncle_count] sys_get_uncle_count,  
extern int sys_get_uncle_count(void);
```

پیاده‌سازی منطق آن را هم در sysproc.c انجام می‌دهیم و در تابع سیستم‌کال هم ورودی را از رجیستر ebx می‌خوانیم:

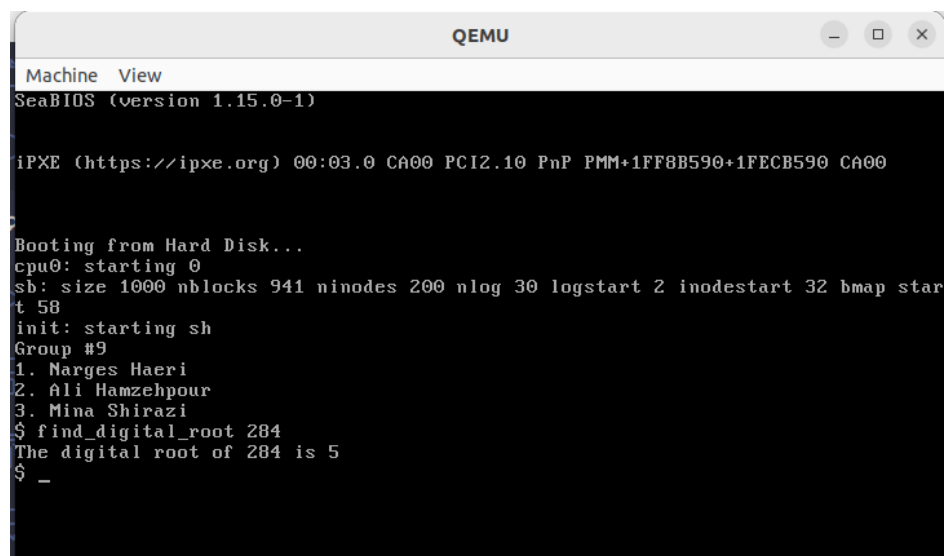
```
int  
sys_find_digital_root(void){  
    return find_digital_root(myproc()->tf->ebx);  
}
```

```
static int  
find_digital_root(int n){  
    if (n <= 0)  
        return -1;  
    while (n > 9){  
        int new_n = 0;  
        while (n != 0){  
            new_n += n % 10;  
            n /= 10;  
        }  
        n = new_n;  
    }  
    return n;  
}
```

در برنامه‌ی سطح کاربر ابتدا باید ورودی را در رجیستر ebx ذخیره کنیم و مقدار اولیه آن را هم جایی ذخیره کنیم تا بعد از انجام سیستم‌کال، بتوانیم مقدار اولیه را به رجیستر ebx برگردانیم:

```
1 #include "types.h"
2 #include "user.h"
3
4 int
5 find_digital_root_handler(int num){
6     int prev_ebx;
7
8     asm volatile(
9         "movl %%ebx, %0\n\t"
10        "movl %1, %%ebx"
11        : "=r"(prev_ebx)
12        : "r"(num)
13    );
14
15    int result = find_digital_root();
16
17    asm volatile(
18        "movl %0, %%ebx"
19        :: "r"(prev_ebx)
20    );
21
22    return result;
23 }
24
25 int
26 main(int argc, char* argv[]){
27     if (argc < 2) {
28         printf(1, "Usage: find_digital_root <num>\n");
29         exit();
30     }
31     int input = atoi(argv[1]);
32     int result = find_digital_root_handler(input);
33     if (result < 0){
34         printf(2, "number should be positive\n");
35         exit();
36     }
37     printf(1, "The digital root of %d is %d\n", input, result);
38     exit();
39 }
```

اجرای این دستور در سیستم‌عامل به این شکل می‌شود:



```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzhepour
3. Mina Shirazi
$ find_digital_root 284
The digital root of 284 is 5
$ _
```

پیاده سازی فراخوانی های سیستمی

۱. پیاده سازی فراخوانی سیستمی طول عمر پردازش:

در این قسمت، مدت زمان زندگی یک پردازش از زمان به وجود آمدن تا زمان صدا کردن این فراخوانی سیستمی `get_process_lifetime(int)` محاسبه کردیم.

- در مرحله اول شناسه فراخوانی سیستمی را به فایل `user.h` اضافه می کنیم:

```
xv6-public > C user.h
27 int copy_file(const char* src, const char* de
28 int get_uncle_count(int pid);
29 int get_process_lifetime(int pid);
30
```

- سپس برای این فراخوانی سیستمی شماره ۲۵ را در فایل `syscall.h` به آن اختصاص می دهیم:

```
xv6-public > C syscall.h
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_find_digital_root 22
24 #define SYS_copy_file 23
25 #define SYS_get_uncle_count 24
26 #define SYS_get_process_lifetime 25
27
```

تعریف تابع را در فایل `usys.S` و به کمک ماکرو `SYSCALL` انجام می دهیم که ابتدا در رجیستر `eax` شماره سیستم کال را قرار می دهد و با استفاده از دستور `int $T_SYSCALL` یک `software interrupt` از نوع `int` درست می کند. حال باید این فراخوانی سیستمی را در سطح هسته تعریف کنیم و برای این عمل:

```
xv6-public > C usys.S
1 #include "syscall.h"
2 #include "traps.h"
3
4 #define SYSCALL(name) \
5     .globl name; \
6     name: \
7     movl $SYS_ ## name, %eax; \
8     int $T_SYSCALL; \
9     ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
17 SYSCALL(close)
18 SYSCALL(kill)
19 SYSCALL(exec)
20 SYSCALL(open)
21 SYSCALL(mknod)
22 SYSCALL(unlink)
23 SYSCALL(fstat)
24 SYSCALL(link)
25 SYSCALL(mkdir)
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(find_digital_root)
33 SYSCALL(copy_file)
34 SYSCALL(get_uncle_count)
35 SYSCALL(get_process_lifetime)
```


شناسه تابع را در فایل syscall.c اضافه کنیم و سپس باید شماره فراخوانی سیستمی را به این تابع مپ کنیم و به تعریف آرایه syscalls اضافه می‌کنیم:

```

104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_find_digital_root(void);
107 extern int sys_copy_file(void);
108 extern int sys_get_uncle_count(void);
109 extern int sys_get_process_lifetime(void);
110
111 static int (*syscalls[])(void) = {
112     [SYS_fork] sys_fork,
113     [SYS_exit] sys_exit,
114     [SYS_wait] sys_wait,
115     [SYS_pipe] sys_pipe,
116     [SYS_read] sys_read,
117     [SYS_kill] sys_kill,
118     [SYS_exec] sys_exec,
119     [SYS_fstat] sys_fstat,
120     [SYS_chdir] sys_chdir,
121     [SYS_dup] sys_dup,
122     [SYS_getpid] sys_getpid,
123     [SYS_sbrk] sys_sbrk,
124     [SYS_sleep] sys_sleep,
125     [SYS_uptime] sys_uptime,
126     [SYS_open] sys_open,
127     [SYS_write] sys_write,
128     [SYS_mknod] sys_mknod,
129     [SYS_unlink] sys_unlink,
130     [SYS_link] sys_link,
131     [SYS_mkdir] sys_mkdir,
132     [SYS_close] sys_close,
133     [SYS_find_digital_root] sys_find_digital_root,
134     [SYS_copy_file] sys_copy_file,
135     [SYS_get_uncle_count] sys_get_uncle_count,
136     [SYS_get_process_lifetime] sys_get_process_lifetime,
137     0,
138 }

```

در ادامه به پیاده‌سازی این تابع می‌پردازیم؛ برای پیاده‌سازی ما باید از ticks استفاده کنیم. در xv6، هسته یک تایمر را پیکربندی می‌کند تا به صورت دوره‌ای به تولید نقاط زمانی یا تایمرهای موقعیت زمانی بپردازد. هر بار که تایمر یک نقطه زمانی تولید می‌کند، به عنوان یک "tick" نامگذاری معمولاً می‌شود. هر زمان یک tick اتفاق می‌افتد، کنترل به رویینی سرویس اینترپت (ISR) مرتبط با اینترپت تایمر منتقل می‌شود. این روند با وظیفه‌های مربوط به این اینترپت مانند به‌روزرسانی زمان سیستم یا اتخاذ تصمیمات در مورد زمان‌بندی فرایندها سر و کار دارد. در اصل ticks تعداد tickهایی است که سیستم عامل تا الان انجام داده. ما در استراکت proc در proc.h یک start_time از جنس uint اضافه می‌کنیم و در allocproc که یک نمونه جدید از استراکت proc درست می‌کند آن را set کردیم بدین صورت که چون در هر ثانیه می‌دانیم ۱۰۰ تیک می‌خورد پس start_time به این صورت می‌شود. ما در ابتدای این فایل TICK_PER_SECOND را دیفاین کردیم و معادل ۱۰۰ قرار دادیم :

```

5 static struct proc*
6 allocproc(void)
7 {
8     struct proc *p;
9     char *sp;
10
11     acquire(&ptable.lock);
12
13     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
14         if(p->state == UNUSED)
15             goto found;
16
17     release(&ptable.lock);
18     return 0;
19
20 found:
21     p->state = EMBRYO;
22     p->pid = nextpid++;
23     p->start_time = ticks/TICKS_PER_SECOND;
24 }

```

در انتهای همین فایل تابع اصلی `find_process_lifetime` را بدین صورت تعریف می‌کنیم در این تابع در ابتدا پردازش مورد نظر پیدا می‌کنیم و بعد زمان فعلی را هم تعریف می‌کنیم چیزی که این تابع برمیگرداند اختلاف زمان بین شروع پردازش و زمان الان هست که نشان دهنده طول زندگی مطلوب است :

```
int
find_process_lifetime(int pid){
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            break;
        }
    }
    int current_time = ticks / TICKS_PER_SECOND;
    return (current_time - p->start_time);
}
```

در نهایت برای تست و اجرای این فراخوانی سیستمی، یک برنامه سطح کاربر می‌سازیم برای اینکه بتوانیم تست کنیم که درست کار می‌کند یا نه ابتدا `sleep` را برای مدت ۱۰ ثانیه ست می‌کنیم و بعد تابع را صدا می‌زنیم.

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

int main(int argc, char* argv[]){
    if (fork() == 0){
        sleep(1000);
        printf(1, "Child Process lifetime: %d\n", get_process_lifetime(getpid()));
    }
    else {
        wait();
        sleep(200);
        printf(1, "Parent Process lifetime: %d\n", get_process_lifetime(getpid()));
    }
    exit();
}
```

```
$ test_find_process_lifetime
Child Process lifetime: 10
Parent Process lifetime: 12
$
```

۲. فراخوانی سیستمی کپی کردن فایل

عملیات های مربوط به اضافه کردن سیستم‌کال را مانند شماره ۱ انجام می‌دهیم. حال برای پیاده‌سازی این سیستم‌کال ابتدا `inode` مربوط به فایل `src` را با استفاده از دستور `namei` ذخیره می‌کنیم و `inode` مربوط به `dest` را با استفاده از دستور `create` می‌سازیم.

حالا در یک حلقه به نوبت از inode مبدا می‌خوانیم و در inode مقصد می‌نویسیم. برای خواندن از دستور read و برای نوشتن از دستور write استفاده می‌کنیم. در پایان سائز inode مقصد را برابر inode مبدا قرار می‌دهیم.

```
int
sys_copy_file(void){
    char *src, *dest;

    if(argstr(0, &src) < 0 || argstr(1, &dest) < 0)
        return -1;

    begin_op();
    struct inode *src_ip, *dest_ip;
    src_ip = namei(src);
    if (src_ip == 0){
        end_op();
        return -1;
    }

    ilock(src_ip);

    dest_ip = create(dest, T_FILE, 0, 0);

    if(dest_ip == 0){
        iunlock(src_ip);
        end_op();
        return -1;
    }

    int src_size = src_ip->size;
    char* buf = kalloc();
```

```

for (int cur_off = 0; cur_off < src_size; cur_off += BSIZE) {
    int diff = src_size - cur_off;
    int read_size = (diff > BSIZE) ? BSIZE : diff;
    int read_result = readi(src_ip, buf, cur_off, read_size);
    if (read_result < 0){
        iunlockput(dest_ip);
        iunlock(src_ip);
        end_op();
        return -1;
    }
    int write_result = writei(dest_ip, buf, cur_off, read_size);
    if (write_result < 0){
        iunlockput(dest_ip);
        iunlock(src_ip);
        end_op();
        return -1;
    }
}
dest_ip->size = src_size;
iupdate(dest_ip);
iunlock(dest_ip);
iunlock(src_ip);
end_op();
kfree(buf);

return 0;
}

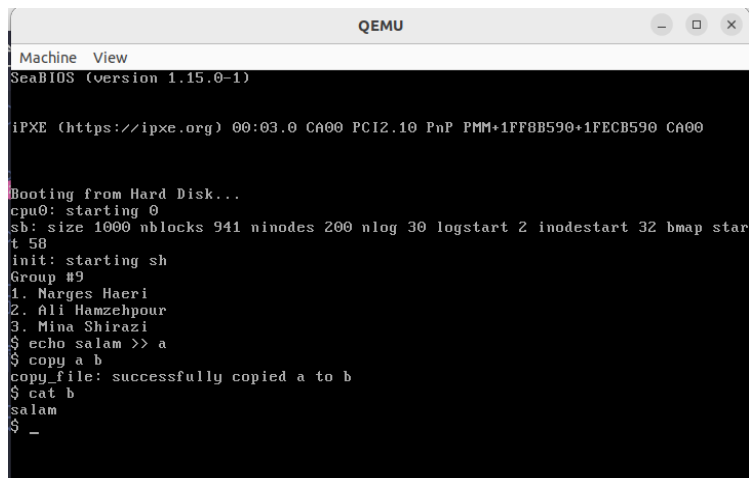
```

برنامه‌ی سطح کاربر به شکل زیر می‌شود:

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

int main(int argc, char* argv[]){
    if (argc < ۳){
        printf(\\, "Usage: copy_file <src> <dst>\\n");
        exit();
    }
    char* src = argv[1];
    char* dest = argv[۲];
    if (copy_file(src, dest) < ۰){
        printf(\\, "copy_file: failed to copy %s to %s\\n", src, dest);
        exit();
    }
    printf(\\, "copy_file: successfully copied %s to %s\\n", src, dest);
    exit();
}
```

اجرای سیستم‌کال در برنامه‌ی سطح کاربر به شکل زیر می‌شود:



```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ echo salam >> a
$ copy a b
copy_file: successfully copied a to b
$ cat b
salam
$ _
```

۳. پیاده سازی فراخوانی سیستمی تعداد uncle های پردازش

عملیات های مربوط به اضافه کردن سیستم کال را مانند شماره ۱ انجام می دهیم. حال برای پیاده سازی این فراخوانی، ابتدا تابع زیر را به proc.c اضافه می کنیم و هدر آن را در defs.h قرار می دهیم:

```
int
uncle_count(int pid)
{
    struct proc *p;
    int num_of_uncles = 0;

    acquire(&ptable.lock);

    int grand_father_pid = -1;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            grand_father_pid = p->parent->parent->pid;
        }
    }
    if (grand_father_pid < 0)
        return -1;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->parent->pid == grand_father_pid)
            num_of_uncles++;

    release(&ptable.lock);
    return num_of_uncles - 1;
}
```

در این تابع ابتدا با استفاده از شماره شناسه پردازش ، شماره شناسه پدر بزرگ پردازش را پیدا می کنیم سپس بر روی تمامی پردازش ها یک حلقه می زنیم و اگر شماره شناسه پدر پردازش در حال پیمایش با شماره شناسه پدر بزرگ پردازش ما یکسان بود این پردازش عمومی پردازش ما می باشد، بنابراین متغیر شمارش uncle ها را که در ابتدا به مقدار صفر مقدار دهی شده بود، یکی اضافه می کنیم.

حال این تابع را در تابع sys_get_uncle_count که در فایل sysproc.c تعریف شده است صدا می زنیم:

```
int
sys_get_uncle_count(void){
    int pid;
    if (argint(0, &pid) < 0)
        return -1;
    return uncle_count(pid);
}
```

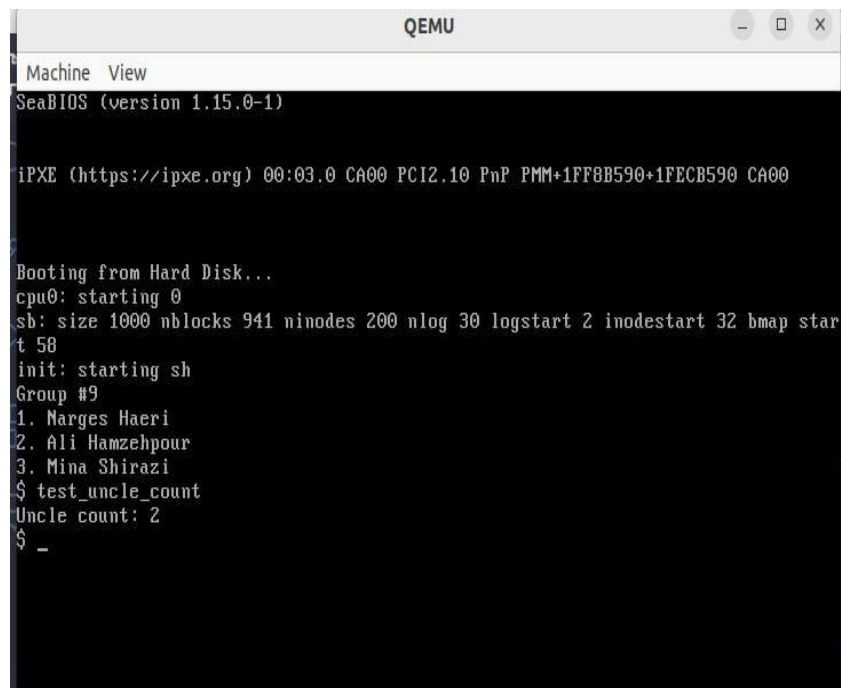
حال برای استفاده از این فراخوانی برنامه سطح کاربر زیر را تعریف می کنیم:

```

1 #include "types.h"
2 #include "user.h"
3
4 int main(int argc, char* argv[]){
5     if (fork() == 0){
6         sleep(10);
7     }
8     else if (fork() == 0){
9         sleep(10);
10    }
11    else if (fork() == 0){
12        if (fork() == 0){
13            printf(1, "Uncle count: %d\n", get_uncle_count(getpid()));
14        }
15        else{
16            wait();
17        }
18    }
19 }
20
21 else {
22     wait();
23     wait();
24     wait();
25 }
26 exit();
27 }

```

در این برنامه با استفاده از `fork()` سه فرزند ساخته می شود و سپس برای یک کدام از آن ها یک فرزند دیگر ایجاد می شود. از تابع `sleep` برای جلوگیری از خروج پردازش های `uncle` استفاده شده است. در نهایت `get_uncle_count` را برای پردازش فرزند فراخوانی می کنیم و خروجی به صورت زیر خواهد بود:



```

QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzeshpour
3. Mina Shirazi
$ test_uncle_count
Uncle count: 2
$ _

```