

گزارش کار پروژه دوم آزمایشگاه سیستم عامل

گروه ۱۳:

ارشیا ابوالقاسم - ۸۱۰۱۹۹۳۱۹

سیده زینبیش بیمن - ۸۱۰۱۹۹۵۹۷

محمد جواد بشارت - ۸۱۰۱۹۹۳۸۶

آدرس مخزن github:

<https://github.com/JavadBesharati/OS-lab-projects-UT-Spring-2023>

شناسه آخرین commit:

9bcd22c3a5e0c522d642e3466250e60059a98018

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

یک برنامه سطح کاربر به نام `proc_id.c` نوشته ایم که این برنامه `id` پردازش فعلی را به ما خواهد داد:

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

بعد از نوشتن برنامه سطح کاربر، هم `xv6` و هم `gdb` را در (حالت `kernel`) اجرا می کنیم. سپس در `gdb` برنامه را `continue` می کنیم تا برنامه کمی جلو برود و بتوانیم برنامه سطح کاربر را اجرا نماییم. سپس در `gdb` کنترل `C` می زنیم. حال یک `breakpoint` در خط 138 فایل `syscall.c` اضافه می کنیم. سپس دوباره برنامه را `continue` کرده حال روی `qemu` برنامه `proc_id` را اجرا می نماییم و خواهیم دید که `breakpoint` ای که گذاشتیم، `hit` شده است. تصاویر زیر مراحل مذکور را در `gdb` و `qemu` نشان می دهند:

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000ffff in ?? ()
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48      for (i = 0; i < ncpu; ++i) {
(gdb) b syscall.c:138
Breakpoint 1 at 0x80104f14: file syscall.c, line 138.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) □
```

```
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
```

```
Group #13:
```

```
1. Arshia
```

```
2. Zeinab
```

```
3. Javad
```

```
$ proc_id
```

```
-
```

حال با فشردن همزمان `ctrl+x+a` به حالت TUI در `gdb` می‌رویم و دستور `bt` را همانطور که خواسته شده وارد می‌کنیم که تصویر آن در صفحه بعد آمده است.

توضیح دستور `bt`: این دستور مخفف `backtrace` است و سلسله توابعی که فراخوانی شده‌اند و به استک اضافه شده‌اند را نشان می‌دهد. به طور خلاصه می‌توان گفت این دستور نشان می‌دهد که برنامه چطور به جایی که اکنون در آن قرار دارد، رسیده است. برای آنکه بهتر متوجه خروجی این دستور شویم، لازم است تا مراحل تعریف و اجرای یک فراخوانی سیستمی را بدانیم. که در ادامه این مراحل را توضیح می‌دهیم:

- ۱- در فایل `syscall.h` یک عدد برای فراخوانی سیستمی مورد نظر انتخاب شده است.
- ۲- شناسه فراخوانی سیستمی مورد نظر در فایل `user.h` نوشته شده است.
- ۳- در فایل `usys.S` تعریف فراخوانی سیستمی به اسمبلی انجام می‌شود.
- ۴- `vector64` در فایل `vectors.S` تعریف شده است. که می‌توان با اجرای دستور `int 64` در مرحله قبل، وارد این بخش شد. بعد از `push` شدن مقدار 64، به بخش `alltraps` در فایل `trapasm.S` خواهیم رفت.
- ۵- بخش `alltraps` ابتدا `trap frame` مربوطه را خواهد ساخت و آن را در استک `push` خواهد کرد. سپس تابع `trap` در فایل `trap.c` را فراخوانی خواهد کرد.
- ۶- تابع `trap` بعد از آنکه می‌فهمد فراخوانی مربوط به یک `system call` است، `trap frame` ای که در استک پوش شده است را به عنوان `trap frame` پردازش فعلی قرار خواهد داد و تابع `syscall` را فرا می‌خواند.

- ۷- تابع syscall در فایل syscall.c قرار دارد. این تابع پس از خواندن شماره فراخوانی سیستمی که در فیلد eax در trap frame پردازش فعلی قرار دارد، تابع مربوط به آن را فرا می‌خواند و خروجی این تابع را در فیلد eax در trap frame پردازش فعلی ذخیره می‌کند.

```

syscall.c
127 [SYS_mkdir] sys_mkdir,
128 [SYS_close] sys_close,
129 };
130
131 void
132 syscall(void)
133 {
134     int num;
135     struct proc *curproc = myproc();
136
137     num = curproc->tf->eax;
B+> 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

remote Thread 1.1 In: syscall
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x80105f4d in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80105cef in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) 

```

همانطور که در تصویر فوق مشاهده می‌شود، خروجی دستور bt مراحل ۵ تا ۷ تعریف و اجرای یک فراخوانی سیستمی را نمایش می‌دهد.

در صورت استفاده از دستور down به خطای زیر می‌خوریم که علت آن این است که در داخلی‌ترین frame قرار داریم:

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

پس از دستور up استفاده می‌کنیم و به یک frame عقب‌تر می‌رویم:

```
trap.c
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);
```

remote Thread 1.1 In: trap

(gdb) bt

#0 syscall () at syscall.c:138

#1 0x80105f4d in trap (tf=0x8dffefb4) at trap.c:43

#2 0x80105cef 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 0x80105f4d in trap (tf=0x8dffefb4) at trap.c:43

شماره فراخوانی سیستمی getpid برابر با ۱۱ است. حال اگر محتوای ثبات eax را بخوانیم، می‌بینیم که مقدار آن ۵ است و با شماره فراخوانی سیستمی مورد نظر ما متفاوت است:

```
#1 0x80105f4d in trap (tf=0x8dffffb4) at trap.c:43
(gdb) print myproc()->tf->eax
$1 = 5
(gdb) □
```

علت این اتفاق این است که قبل از رسیدن به فراخوانی سیستمی getpid فراخوانی های سیستمی دیگری هم رخ می‌دهند. در صورتی که برنامه را چندین مرحله continue کرده و هر بار محتوای ثبات eax را چک کنیم، این فراخوانی ها را خواهیم دید:

- فراخوانی سیستمی شماره ۵ (read): این فراخوانی سیستمی برای خواندن دستور تایپ شده است و آن را کاراکتر به کاراکتر می‌خواند.
- فراخوانی سیستمی شماره ۱ (fork): این فراخوانی سیستمی برای ایجاد پردازش جدید جهت اجرای برنامه سطح کاربر اجرا می‌شود.
- فراخوانی سیستمی شماره ۱۲ (sbrk): این فراخوانی سیستمی جهت تخصیص حافظه به پردازش ایجاد شده اجرا می‌شود.
- فراخوانی سیستمی شماره ۷ (exec): این فراخوانی سیستمی برای اجرای برنامه سطح کاربر در پردازش ایجاد شده اجرا می‌شود.
- فراخوانی سیستمی شماره ۳ (wait): این فراخوانی سیستمی در پردازش پدر اجرا می‌شود و تا اتمام اجرای پردازش فرزند صبر می‌کند.
- فراخوانی سیستمی شماره ۱۱ (getpid): این فراخوانی سیستمی مربوط به برنامه سطح کاربر است.
- فراخوانی سیستمی شماره ۱۶ (write): این فراخوانی سیستمی خروجی برنامه سطح کاربر را کاراکتر به کاراکتر می‌نویسد.

(شماره تمامی فراخوانی های سیستمی در فایل syscall.h موجود است.)

در تصاویر صفحه بعد روند اجرا شدن فراخوانی های سیستمی تا چاپ شدن خروجی برنامه سطح کاربر قابل مشاهده است:

چون اسم برنامه سطح کاربر (proc_id) از ۷ کاراکتر تشکیل شده پس انتظار داریم ۷ بار فراخوانی read (شماره ۵) رخ دهد ولی چون یک کاراکتر هم زمان رسیدن به breakpoint خوانده شده پس ۶ بار این فراخوانی رخ خواهد داد:

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$2 = 5
(gdb) □
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$7 = 5
(gdb) □
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$8 = 1
(gdb) □
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$9 = 3
(gdb) □
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$10 = 12
(gdb) █
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$11 = 7
(gdb) █
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$12 = 11
(gdb) █
```

بعد از این مرحله به تعداد کاراکتر های خروجی برنامه سطح کاربر، فراخوانی سیستمی write (شماره ۱۶) اجرا خواهد شد. در نهایت خروجی برنامه سطح کاربر به شکل زیر است:

```
Group #13:
1. Arshia
2. Zeinab
3. Javad
$ proc_id
Process ID is: 3_
```


ارسال آرگومان‌های فراخوانی‌های سیستمی

- در فایل `syscall.h` به فراخوانی سیستمی `sys_find_fibonacci_number` شماره 22 را اختصاص می‌دهیم.
- در فایل `syscall.c` این تابع را به لیست توابع فراخوانی‌های سیستمی اضافه می‌کنیم.
- تعریف این تابع را در فایل `sysproc.c` می‌آوریم.
- به فایل `user.h` تابع سطح بالاتر `find_fibonacci_number` را اضافه می‌کنیم و نحوه فراخوانی آن را مشخص می‌کنیم.
- نام این تابع را به فایل `usys.S` اضافه می‌کنیم و در فایل `defs.h` امضای تابع را تعریف می‌کنیم.
- در فایل `proc.c` تعریف تابع را انجام می‌دهیم.

اعمال این تغییرات در تصاویر زیر آمده است:

```
1 // System call numbers
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
5 #define SYS_pipe    4
6 #define SYS_read    5
7 #define SYS_kill    6
8 #define SYS_exec    7
9 #define SYS_fstat    8
10 #define SYS_chdir   9
11 #define SYS_dup     10
12 #define SYS_getpid  11
13 #define SYS_sbrk    12
14 #define SYS_sleep   13
15 #define SYS_uptime  14
16 #define SYS_open    15
17 #define SYS_write   16
18 #define SYS_mknod   17
19 #define SYS_unlink  18
20 #define SYS_link     19
21 #define SYS_mkdir    20
22 #define SYS_close    21
23 #define SYS_find_fibonacci_number 22
24
```

فایل `syscall.h`:



فایل :syscall.c

```
96 extern int sys_mknod(void);
97 extern int sys_open(void);
98 extern int sys_pipe(void);
99 extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_find_fibonacci_number(void);
```

```
123 [SYS_open]      sys_open,
124 [SYS_write]     sys_write,
125 [SYS_mknod]     sys_mknod,
126 [SYS_unlink]    sys_unlink,
127 [SYS_link]      sys_link,
128 [SYS_mkdir]     sys_mkdir,
129 [SYS_close]     sys_close,
130 [SYS_find_fibonacci_number] sys_find_fibonacci_number,
131 };
```

فایل :sysproc.c

```
93 // system call to find the nth fibonacci number:
94
95 int sys_find_fibonacci_number(void){
96     int n = myproc()->tf->ebx;
97     cprintf("Kernel: sys_find_fibonacci_number(%d) is called\n", n);
98     cprintf("        now calling find_fibonacci_number(%d)\n", n);
99     return find_fibonacci_number(n);
100 }
```

```
19 int mkdir(const char*);
20 int chdir(const char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int find_fibonacci_number(void);
```

فایل :user.h

```
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(find_fibonacci_number)
33
```

فایل :usys.S

```
115 void scheduler(void) __attribute__((noreturn));
116 void sched(void);
117 void setproc(struct proc*);
118 void sleep(void*, struct spinlock*);
119 void userinit(void);
120 int wait(void);
121 void wakeup(void*);
122 void yield(void);
123 int find_fibonacci_number(int);
```

فایل :defs.h

```

536 // find the nth fibonacci number:
537 int find_fibonacci_number(int n) {
538     if(n > 1 && n < 4) {
539         return 1;
540     }
541     else if(n == 1) {
542         return 0;
543     }
544     else if (n < 1) {
545         return -1;
546     }
547     else{
548         return find_fibonacci_number(n - 1) + find_fibonacci_number(n - 2);
549     }
550 }

```

در ادامه یک برنامه سطح کاربر به نام `test_find_fibonacci_number` برای تست فراخوانی سیستمی‌ای که نوشته‌ایم می‌نویسیم تا ببینیم برنامه به درستی کار می‌کند یا خیر. کد مربوط به این برنامه سطح کاربر در صفحه بعد آمده است و طبعاً چون برنامه سطح کاربر است، برای کامپایل شدنش بایستی به قسمت EXTRA و UPROGS در Makefile نام این برنامه را اضافه کنیم.

```

1#include "types.h"
2#include "fcntl.h"
3#include "user.h"
4
5int main(int argc, char* argv[]){
6    write(1, "testing find_fibonacci_number system call started\n", 50);
7
8    if(argc != 2){
9        write(1, "Invalid input. Please enter just a single integer.\n", 51);
10       exit();
11    }
12
13    int n = atoi(argv[1]), prev_ebx;
14
15    asm volatile(
16        "movl %%ebx, %0;"
17        "movl %1, %%ebx;"
18        : "=r" (prev_ebx)
19        : "r"(n)
20        );
21
22    printf(1, "calling find_fibonacci_number(%d)\n", n);
23
24    int answer = find_fibonacci_number();
25
26    asm volatile(
27        "movl %0, %%ebx;"
28        : : "r"(prev_ebx)
29        );
30
31    if(answer == -1){
32        printf(1, "find_fibonacci_number failed\n");
33        printf(1, "Please check if you entered an integer smaller than 1\n");
34        exit();
35    }
36
37    printf(1, "find_fibonacci_number(%d) = %d\n", n, answer);
38    exit();
39}

```



```
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
```

```
Group #13:
```

```
1. Arshia
```

```
2. Zeinab
```

```
3. Javad
```

```
$ test_find_fibonacci_number 10
```

```
testing find_fibonacci_number system call started
```

```
calling find_fibonacci_number(10)
```

```
Kernel: sys_find_fibonacci_number(10) is called
```

```
now calling find_fibonacci_number(10)
```

```
find_fibonacci_number(10) = 34
```

```
$ _
```

پیاده سازی فراخوانی سیستمی پر استفاده ترین فراخوانی سیستمی

- در فایل `syscall.h` به فراخوانی سیستمی `sys_find_most_callee` شماره 23 را اختصاص می‌دهیم.
- در فایل `syscall.c` این تابع را به لیست توابع فراخوانی های سیستمی اضافه می‌کنیم.
- تعریف این تابع را در فایل `sysproc.c` می‌آوریم.
- به فایل `user.h` تابع سطح بالاتر `find_most_callee` را اضافه می‌کنیم و نحوه فراخوانی آن را مشخص می‌کنیم.
- نام این تابع را به فایل `usys.S` اضافه می‌کنیم و در فایل `defs.h` امضای تابع را تعریف می‌کنیم.
- در فایل `proc.c` تعریف تابع را انجام می‌دهیم.
- یک برنامه سطح کاربر برای تست کردن این فراخوانی سیستمی با نام `test_find_most_callee` می‌نویسیم
- به قسمت `EXTRA` و `UPROGS` در `Makefile` نام برنامه سطح کاربر را اضافه می‌کنیم تا این برنامه هم کامپایل شود.
- برای یافتن پر استفاده ترین فراخوانی سیستمی یک آرایه به نام `syscalls_count` در فایل `proc.c` تعریف می‌کنیم.
- طول این آرایه را 30 در نظر می‌گیریم و هر بار که یک فراخوانی سیستمی داشتیم، مقدار خانه‌ای از آرایه که اندیس آن

متناظر شماره فراخوانی سیستمی است، 1 واحد زیاد می‌کنیم. در انتها بین عناصر آرایه ماکسیم می‌گیریم و پاسخ را اعلام می‌کنیم.

در ادامه تصاویر مربوط به پیاده سازی مراحل گفته شده آمده‌اند:

```
C syscall.h > ...
1  // System call numbers
2  #define SYS_fork    1
3  #define SYS_exit    2
4  #define SYS_wait    3
5  #define SYS_pipe    4
6  #define SYS_read    5
7  #define SYS_kill    6
8  #define SYS_exec    7
9  #define SYS_fstat    8
10 #define SYS_chdir    9
11 #define SYS_dup     10
12 #define SYS_getpid  11
13 #define SYS_sbrk    12
14 #define SYS_sleep   13
15 #define SYS_uptime  14
16 #define SYS_open    15
17 #define SYS_write   16
18 #define SYS_mknod   17
19 #define SYS_unlink  18
20 #define SYS_link    19
21 #define SYS_mkdir   20
22 #define SYS_close   21
23 #define SYS_find_fibonacci_number 22
24 #define SYS_find_most_callee 23
25
```

```
C syscall.c > [?] syscalls
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_find_fibonacci_number(void);
107 extern int sys_find_most_callee(void);
```

C syscall.c > ...

```
129 [SYS_mkdir] sys_mkdir,  
130 [SYS_close] sys_close,  
131 [SYS_find_fibonacci_number] sys_find_fibonacci_number,  
132 [SYS_find_most_callee] sys_find_most_callee,  
133 };
```


برای آن که بتوانیم هر بار یک فراخوانی سیستمی استفاده می‌شود، تعداد دفعات استفاده شدنش را به روز نماییم، از تابع `update_syscalls_count` استفاده کرده‌ایم و از این تابع بایستی در تابع `syscall` فایل `syscall.c` استفاده کرد:

C syscall.c > ...

```
136 void  
137 syscall(void)  
138 {  
139     int num;  
140     struct proc *curproc = myproc();  
141  
142     num = curproc->tf->eax;  
143     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {  
144         // increase the count of how many times a system call, has been called by one.  
145         update_syscalls_count(num);  
146         curproc->tf->eax = syscalls[num]();  
147     } else {  
148         cprintf("%d %s: unknown sys call %d\n",  
149             curproc->pid, curproc->name, num);  
150         curproc->tf->eax = -1;  
151     }  
152 }  
153
```

C sysproc.c > ...

```
101
102 // system call to find the most used system call:
103
104 int sys_find_most_callee(void){
105     cprintf("Kernel: sys_find_most_callee is called\n");
106     cprintf("        now calling find_most_callee\n");
107     return find_most_callee();
108 }
109 |
```

C user.h >  atoi(const char *)

```
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int find_fibonacci_number(void);
27 int find_most_callee(void);
```

ASM usys.S

```
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(find_fibonacci_number)
33 SYSCALL(find_most_callee)
34 |
```

C defs.h > ...

```
119 void userinit(void);
120 int wait(void);
121 void wakeup(void*);
122 void yield(void);
123 int find_fibonacci_number(int);
124 void update_syscalls_count(int);
125 int find_most_callee(void);
126 |
```

```
C proc.h > NUM_OF_SYSCALLS
58 // expandable heap
59
60 #define NUM_OF_SYSCALLS 30
```

```
C proc.c > ...
551
552 // an array to keep how many times each system call called
553 int syscalls_count[NUM_OF_SYSCALLS] = {0};
554
555 void update_syscalls_count(int num){
556     syscalls_count[num - 1] = syscalls_count[num - 1] + 1;
557 }
558
559 int find_most_callee(void){
560     int most_used_sys_call = 0;
561     int max_called = -1;
562
563     for(int i = 0; i < NUM_OF_SYSCALLS; i++){
564         if(syscalls_count[i] > max_called){
565             most_used_sys_call = i + 1;
566             max_called = syscalls_count[i];
567         }
568     }
569
570     return most_used_sys_call;
571 }
572
```

```

C test_find_most_callee.c > ...
1  #include "types.h"
2  #include "fcntl.h"
3  #include "user.h"
4  #include "syscall.h"
5
6  int main(int argc, char* argv[]){
7      printf(1, "calling find_most_callee\n");
8      int most_used_sys_call = find_most_callee();
9      printf("Most used system call number is: %d\n", most_used_sys_call);
10     exit();
11 }
12

```

حال در صورت اجرای برنامه سطح کاربر بعد از بالا آمدن xv6 خروجی به شکل زیر خواهد شد:

```

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
Group #13:
1. Arshia
2. Zeinab
3. Javad
$ test_find_most_callee
calling find_most_callee
Kernel: sys_find_most_callee is called
      now calling find_most_callee
Most used system call number is: 16
$ _

```

همانطور که در تصویر فوق مشاهده می‌شود شماره پر استفاده ترین فراخوانی سیستمی برابر 16 است که مربوط به فراخوانی سیستمی write می‌شود. از آنجا که فراخوانی سیستمی write برای نوشتن در خروجی به کار می‌رود و بجز برنامه سطح کاربر برنامه خاص دیگری را اجرا نکرده‌ایم، دیگر فراخوانی های سیستمی ای که ممکن است رخ داده باشند، read و exec اند. شاید فراخوانی های سیستمی دیگری هم باشند که اطلاع دقیقی از این که رخ داده‌اند یا خیر نداریم. ولی پر استفاده ترین آن‌ها تا این لحظه write بوده. چرا که در خروجی چیز های زیادی نوشته شده است و برای نوشتن هر کاراکتر، این فراخوانی سیستمی یک بار صدا زده می‌شود. پس منطقیست که همین فراخوانی سیستمی تا این لحظه پر کاربرد ترین باشد.

همانطور که در روند اضافه کردن دو systemcall قبلی مشاهده شد، مراحل تغییر فایل‌های syscall.h, syscall.c, user.h, usys.S, defs.h و Makefile کاملاً مشابه است و فقط در برخی جزئیات کوچک با یکدیگر متفاوتند (مانند نام توابع اضافه شده و ...). و به نوعی فقط تعریف توابع در sysproc.c, proc.c و تعریف برنامه تست مهم است. پس برای دو فراخوانی سیستمی باقی مانده فقط فایل‌های proc.c, sysproc.c و برنامه سطح کاربر و نمونه‌ای از اجرای شان را نشان می‌دهیم و در باقی فایل‌ها اگر تغییر خیلی متفاوتی وجود داشت، نشان می‌دهیم.

پیاده سازی فراخوانی سیستمی تعداد فرزندان پردازش کنونی

برای پیاده‌سازی این فراخوانی سیستمی ابتدا id پردازش فعلی را با استفاده از تابع myproc به دست می‌آوریم. سپس جدول پردازش‌ها را با استفاده از تابع acquire تا پایان کار این تابع فیکس می‌کنیم و بعد از اتمام کار این تابع، جدول پردازش‌ها را release می‌کنیم. در ادامه در یک حلقه for به سادگی چک می‌کنیم که اگر id پدر پردازش فرزند با id پردازش فعلی یکسان بود، تعداد فرزندان پردازش فعلی یکی زیاد شود.

نکته‌ای که در فایل تست سطح کاربر مهم است این است که تا زمانی که تمام پردازش‌های فرزند نمرده‌اند، پردازش پدر را نگه داریم که برای این کار از توابع wait و sleep استفاده می‌کنیم. (دقت شود در صورتی که این کار را نکنیم، ممکن است تعداد فرزندان پردازش کنونی به درستی نمایش داده شود چون ممکن است پردازش پدر قبل از پردازش فرزند بمیرد. همچنین ممکن است در ترمینال کلمه zombie را مشاهده کنیم که به این علت است که هر پردازش فرزند قبل از این که بمیرد، تبدیل به zombie می‌شود و باعث اجرای فراخوانی سیستمی zombie خواهد شد برای اطلاعات بیشتر در این مورد می‌توانید به این [لینک](#) مراجعه نمایید.)

در ادامه تصاویر مربوط به فایل‌های sysproc.c, proc.c، برنامه تست سطح کاربر و نمونه‌ای از اجرای فراخوانی سیستمی خواسته شده آورده شده است:

```
C sysproc.c > ...
109
110 // system call to get children count of current process:
111
112 int sys_get_children_count(void){
113     cprintf("Kernel: sys_get_children_count is called\n");
114     cprintf("        now calling get_children_count\n");
115     return get_children_count();
116 }
```


C proc.c > ...

```
573 // a function to return children count of current process:
574 int get_children_count(void){
575     int pid = myproc()->pid;
576     struct proc *p;
577     int children_count = 0;
578
579     acquire(&ptable.lock);
580
581     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
582         if(p->parent->pid == pid){
583             children_count++;
584         }
585     }
586
587     release(&ptable.lock);
588
589     return children_count;
590 }
```

```

C test_get_children_count.c > ...
1  #include "types.h"
2  #include "fcntl.h"
3  #include "user.h"
4  #include "syscall.h"
5
6  int main(int argc, char* argv[]){
7      printf(1, "calling get_children_count\n");
8
9      int c1 = fork();
10     int c2 = fork();
11     int c3 = fork();
12
13     if(c1 > 0 && c2 > 0 && c3 > 0){
14         printf(1, "Children count of parent is: %d\n", get_children_count());
15     }
16     // wait for all children to die :(
17     while(wait() != -1) {}
18     // give time to parent to reach wait clause
19     sleep(1);
20     exit();
21 }
22

```

Group #13:

1. Arshia

2. Zeinab

3. Javad

\$ test_get_children_count

calling get_children_count

Kernel: sys_get_children_count is called

now calling get_children_count

Children count of parent is: 3

\$ _

پیاده سازی فراخوانی سیستمی کشتن اولین فرزند پردازش کنونی

برای پیاده سازی این فراخوانی سیستمی دوباره `id` پردازش فعلی را با استفاده از تابع `myproc` به دست می آوریم. سپس در یک حلقه `for` تا یافتن اولین فرزند پردازش فعلی در جدول پردازش ها جست و جو می کنیم و با یافتن اولین فرزند با استفاده از تابع `kill` که `id` یک پردازش را می گیرد و آن را می کشد، `id` این پردازش را به تابع `kill` می دهیم. در مورد تابع سطح کاربر هم تنها نکته مهم مشابه با قسمت قبلی استفاده از `wait` و `sleep` است تا پردازش پدر تا کشته شدن فرزندش صبر کند.

در ادامه تصاویر مربوط به فایل های `sysproc.c`, `proc.c`، برنامه تست سطح کاربر و نمونه ای از اجرای فراخوانی سیستمی خواسته شده آورده شده است:

```
C sysproc.c > ...
118 // system call to kill the first child of a process:
119
120 int sys_kill_first_child_process(void){
121     cprintf("Kernel: sys_kill_first_child_process is called\n");
122     cprintf("        now calling kill_first_child_process\n");
123     return kill_first_child_process();
124 }
125
```

```
C proc.c > ...
591
592 // a function to kill the first child of current process:
593
594 int kill_first_child_process(void){
595     int pid = myproc()->pid;
596     struct proc *p;
597
598     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
599         if(p->parent->pid == pid){
600             kill(p->pid);
601             // If child killed successfully, return 1
602             return 1;
603         }
604     }
605     // If parent has no child, return -1
606     return -1;
607 }
608
```

```

C test_kill_first_child_process.c > ...
1  #include "types.h"
2  #include "fcntl.h"
3  #include "user.h"
4  #include "syscall.h"
5
6  int main(int argc, char* argv[]){
7      printf(1, "calling kill_first_child_process\n");
8
9      int c1 = fork();
10
11     if(c1 > 0){
12         printf(1, "Children count before any child is killed: %d\n", get_children_count());
13         int kill_result = kill_first_child_process();
14         while(wait() != -1) {}
15         sleep(1);
16         if(kill_result == 1){
17             printf(1, "First child was killed successfully.\n");
18             printf(1, "Children count after first child was killed: %d\n", get_children_count());
19             exit();
20         }
21         else{
22             printf(1, "Parent has no child.\n");
23             exit();
24         }
25     }
26     else{
27         printf(1, "Error in fork\n");
28         exit();
29     }
30 }
31 |

```

Group #13:

1. Arshia
2. Zeinab
3. Javad

```

$ test_kill_first_child_process
calling kill_first_child_process
Kernel: sys_get_children_count is called
      now calling get_children_count
Children count before any child is killed: 1
Kernel: sys_kill_first_child_process is called
      now calling kill_first_child_process
First child was killed successfully.
Kernel: sys_get_children_count is called
      now calling get_children_count
Children count after first child was killed: 0
$

```

همانطور که در تصویر مقابل مشاهده می‌شود، ابتدا تابع تست کشتن اولین فرزند پردازش فعلی صدا می‌شود ولی چون داخل آن از تابع تعداد فرزندان پردازش فعلی استفاده کرده‌ایم، این تابع هم صدا زده می‌شود و چون با fork یک پردازش ایجاد کرده‌ایم، پردازش فعلی یک فرزند خواهد داشت در ادامه تابع کشتن اولین فرزند پردازش فعلی صدا می‌شود. فرزند پردازش فعلی کشته می‌شود و دوباره تابع تعداد فرزندان پردازش فعلی صدا می‌شود و این

بار چون پردازش فعلی فرزندش کشته شده است، خروجی این تابع صفر می‌شود.

پاسخ به سؤالات تشریحی:

۱. کتابخانه‌های هر یک از فایل‌های تشکیل‌دهنده متغیر ULIB را به صورت مجزا بررسی می‌کنیم:

• **Ulib**: تعدادی از توابع این کتابخانه به منظور کار با آرایه‌ی کاراکترها نوشته شده‌اند. این توابع فرخوانی‌های سیستمی ندارند. اما در سه تابع `stat`, `memset`, `gets` فراخوانی سیستمی استفاده شده است.

• **Memset**: برای پر کردن حافظه با یک مقدار خاص استفاده می‌شود و با دادن یک اشاره‌گر به حافظه‌ای که قرار است پر شود، مقداری مشخص را به آن تخصیص می‌دهد.

• **Gets**: یک حلقه اجرا و ورودی گرفته می‌شود. در هر مرحله از انجام این کار از فراخوانی سیستمی `read` استفاده شده است.

• **Stat**: در این تابع، ابتدا از فراخوانی سیستمی `open` استفاده شده است که برای باز کردن یک فایل استفاده می‌شود. در بخش بعد به کمک فراخوانی سیستمی `fstat` اطلاعات فایل مرتبط با `file descriptor` درخواستی داده می‌شود و در انتها به کمک فراخوانی سیستمی `close`، فایل بسته می‌شود.

• **Printf**: تعدادی از توابع این کتابخانه به منظور توابع کمکی برای چاپ در حالت‌های مختلف نوشته شده‌اند. تنها تابعی که در این بخش از فراخوانی سیستمی استفاده می‌کند، تابع `printf` و `putc` است.

• **Putc**: از فراخوانی سیستمی `write` برای چاپ کردن استفاده می‌کند و در هر فراخوانی، فقط یک کاراکتر در `fd` مربوطه می‌نویسد.

• **Printint**: مقادیر عددی را چاپ می‌کند و می‌تواند عدد صحیح ورودی را با توجه به مبنای داده‌شده تفسیر و چاپ کند.

• **Umalloc**: شامل تعدادی توابع کتابخانه‌ای به منظور اختصاص دادن حافظه یا آزاد کردن آن می‌باشد. در این بخش تنها تابع `morecore` از فراخوانی سیستمی استفاده می‌کند.

• **Morecore**: این تابع برای تخصیص حافظه استفاده می‌شود و در نهایت با استفاده از سیستم `sbrk` فضای پردازش را افزایش می‌دهد.

۲. انواع روش‌های فراخوانی‌های سیستمی در لینوکس:

- **Hardware Interrupt**: از طریق سخت‌افزارها (معمولا I/O) رخ می‌دهند و به صورت asynchronous هم اجرا می‌شوند. برای مثال زمانی که در کیبورد کلیدی را فشار می‌دهیم، موس را حرکت می‌دهیم یا یک packet از طریق شبکه به ما می‌رسد، یک interrupt رخ می‌دهد.
- **Software Interrupt – trap**: توسط یک برنامه و به صورت synchronous رخ می‌دهند و انواع مختلفی دارند:
 - **System Call**: فراخوانی‌های سیستمی که پیشتر در مورد آنها صحبت شده است.
 - **Exception**: هنگامی که خطاهای محاسباتی مانند تقسیم بر صفر رخ می‌دهد، به مود کرنل رفته تا خطا را رفع کند و سپس به مود یوزر باز می‌گردد.
 - **Signal**: سیگنال‌ها در لینوکس انواع مختلفی دارند، که پرکاربردترین آنها عبارتند از SIGKILL و SIGINT و SIGTERM.
- **Pseudo-file system**: اطلاعاتی را به عنوان فایل در اختیار کاربران قرار می‌دهند، اما در واقع به عنوان رابط برنامه‌نویسی بین سیستم‌عامل و برنامه‌ها عمل می‌کنند. در واقع، این فایل سیستم‌ها فقط به صورت مجازی و برای ارائه اطلاعات به کاربران ایجاد شده‌اند و به طور مستقیم به فایل‌های فیزیکی سیستم دسترسی ندارند.
 - **/proc**: اطلاعاتی درباره فرآیندها، منابع سیستم، حافظه و شبکه را ارائه می‌دهد.
 - **/sys**: به برنامه‌ها اجازه می‌دهد تا با هسته سیستم‌عامل ارتباط برقرار کنند و اطلاعاتی درباره تنظیمات سیستم و دستگاه‌های سخت‌افزاری را در اختیار بگذارند.
 - **/dev**: به برنامه‌ها اجازه می‌دهد با دستگاه‌های سخت‌افزاری مثل دستگاه‌های دیسک، پرینتر و دیگر دستگاه‌ها ارتباط برقرار کنند.
 - **/tmp**: به برنامه‌ها اجازه می‌دهد تا فایل‌های موقتی را برای اجرای عملیات‌های خاص در زمان اجرا ایجاد کنند.

۳. خیر، چرا که امنیت سیستم را به خطر می‌اندازد. دسترسی DPL_USER سطح کاربر است و امکان فعال‌سازی سایر تله‌ها را ندارد. اگر چنین نباشد، کاربر می‌تواند در هسته اختلال ایجاد کند. 6xv برای جلوگیری از چنین مواردی، پس از تغییر پرده از مود کاربر به هسته، آرگومان آن را چک و تایید می‌کند.

۴. وقتی که سطح دسترسی در حال تغییر است، ممکن است که مجموعه‌ای از دستورات بر اساس سطح دسترسی قبلی اجرا شده باشند و سپس بخواهیم به سطح دسترسی جدید دسترسی پیدا کنیم. بنابراین، برای انتقال کنترل به کد با سطح دسترسی جدید، آدرس بازگشتی (return address) و همچنین مقدار ثابتی که برای ESP ذخیره شده است (که به عنوان آدرس بازگشتی برای بازگشت به سطح دسترسی قبلی استفاده می‌شود) باید از طریق پشته push شوند تا در صورت نیاز بتوان به آنها دسترسی پیدا کرد. به این ترتیب، در صورت بازگشت به سطح دسترسی قبلی، با استفاده از آدرس بازگشتی و ESP ذخیره شده، به مکان مناسب برای ادامه اجرای دستورات پیشین باز خواهیم گشت.

۵.

• در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید.

توابع دسترسی به پارامترهای فراخوانی سیستمی در واقع به کد درون هسته سیستم عامل کمک می‌کنند تا به پارامترهای مربوط به سیستم فراخوانی داده شده توسط یک فرآیند دسترسی پیدا کنند.

این توابع از داده‌های موجود در پشته کرنل استفاده می‌کنند تا پارامترهای فراخوانی شده را بازیابی کنند. به عنوان مثال،

توابع دسترسی به پارامترهای فراخوانی سیستمی می‌توانند با استفاده از مکانیزم‌های مربوط به پشته کرنل، به

پارامترهایی مانند شماره سیستمی که فراخوانی شده است، آدرس بافر داده‌های ورودی، اندازه بافر، و غیره دسترسی پیدا کنند.

در واقع، این توابع از پشته کرنل استفاده می‌کنند تا داده‌های فراخوانی شده را بازیابی کنند، سپس آنها را برای استفاده

در فرآیند‌های بعدی درون هسته سیستم عامل به کار می‌برند. این توابع از پارامترهای فراخوانی سیستمی استفاده می‌کنند تا با کد درون هسته سیستم عامل تعامل کنند و عملیات مورد نظر را انجام دهند.

• چرا در `argptr()` بازه آدرس‌ها بررسی می‌گردد؟

تابع `argptr()` در واقع برای بررسی صحت بافر ورودی در فراخوانی سیستمی استفاده می‌شود. در این تابع، با استفاده از بافر ورودی، آدرس، و اندازه، بازه‌ی حافظه مورد نظر برای دسترسی به داده‌ها بررسی می‌گردد. این بررسی به دلیل این است که برنامه‌نویسان باید از دسترسی به مناطق حافظه‌ای که به آن‌ها دسترسی ندارند، جلوگیری کنند. اگر بافر ورودی نامعتبر باشد و یا بازه آدرس‌هایی که برای دسترسی به داده‌ها استفاده می‌شود صحیح نباشد، ممکن است که برنامه با خطای جدی مواجه شود یا امنیت سیستم عامل به خطر بیفتد. بنابراین، با استفاده از بررسی بازه آدرس‌ها، از این خطرات جلوگیری می‌شود و از صحت و اعتبار بافر ورودی اطمینان حاصل می‌شود.

• تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد می‌کند؟

در این نوع آسیب‌پذیری، اطلاعات بیشتری از بافر مورد نظر در حافظه خوانده می‌شود یا در آن نوشته می‌شود تا آنچه که اندازه‌ی بافر معین شده است. این مسأله ممکن است باعث دسترسی غیرمجاز به حافظه شود که در نتیجه می‌تواند باعث تهدید امنیتی و حتی کنترل کامپیوتر شود. تجاوز از بازه معتبر می‌تواند توسط هر نوع برنامه‌ای با بافر داده‌ای که به‌روزرسانی نشده است، به‌وجود بیاید. برای مثال، یک برنامه‌ی نوشته شده با زبان C که از توابع مانند `strcpy()` برای کپی رشته‌ها استفاده می‌کند، ممکن است در معرض این نوع حملات قرار گیرد. اگر طول رشته‌ی منبع از اندازه‌ی بافر مقصد بیشتر باشد، داده‌های بعدی در حافظه، شامل داده‌های مهم مانند شناسه‌ی کاربر و رمز عبور، به دسترس قرار می‌گیرند. در این صورت، مهاجم می‌تواند به اطلاعات محرمانه‌ی برنامه دسترسی پیدا کند و از آن‌ها بهره‌برد. بنابراین، تجاوز از بازه معتبر می‌تواند باعث افزایش خطر امنیتی در برنامه‌های کامپیوتری شود و برای پیشگیری از آن باید از روش‌هایی مانند بررسی صحت بافرها و حفاظت از حافظه‌ی کامپیوتر در برابر این نوع حملات استفاده کرد. • در صورت عدم بررسی در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی `sys_read()` اجرای سیستم را با مشکل روبرو سازد.

اگر در تابع سفارشی با نام `my_syscall` بدون بررسی بازه آدرس‌ها، مستقیماً به تابع سیستمی `sys_read` فراخوانی شود، ممکن است در صورت ورود بازه آدرس غیرمعتبر به `sys_read`، این تابع اجازه دهد تا داده‌های حساس در حافظه مورد دسترسی قرار گیرند. به عبارت دیگر، فرد مهاجم با استفاده از یک بازه آدرس غیرمعتبر، می‌تواند داده‌های

محرمانه را از حافظه سیستم بخواند و در صورت نیاز آن‌ها را تغییر دهد. بنابراین، اجرای تابع `sys_read` بدون بررسی بازه آدرس‌ها، می‌تواند امنیت سیستم را به خطر بیاندازد.