

دانشگاه تهران دانشکده مهندسی برق و کامپیوتر ۱۴۰۱-۱۴۰۲ آزمایشگاه سیستم عامل بهار
استاد: مهدی کارگهی
نام اعضای گروه: آریان رجبی قلعه، امیر محمد خدایی، امیرحسین کیخسروی

۱- متغیر ULIB از چند فایل تشکیل میشود که عبارتند از :
ulib در تابع gets از فراخوانی سیستمی read برای خواندن ورودی استفاده میکند و در تابع stat برای بازکردن و بستن فایل از دو فراخوانی سیستمی open و close استفاده میکند
printf در تابع putc از فراخوانی سیستمی write برای چاپ یک کاراکتر استفاده میکند
umalloc در این فایل به طور مستقیم از فراخوانی سیستمی morecore به منظور تخصیص حافظه استفاده میشود

۲- در اینجا سه مورد را بازگو میکنم :
سیستم‌های غیر واقعی (pseudo file systems) : فایل سیستم‌هایی هستند که به صورت مجازی در لایه‌ی سیستم‌عامل ایجاد می‌شوند و دسترسی به آن‌ها از طریق درایوهای واقعی انجام می‌شود. این فایل سیستم‌ها به کاربران اجازه می‌دهند تا به منابع دیگر سیستم عامل مانند پردازش‌ها، شبکه، حافظه و ... دسترسی پیدا کنند .
خطاها (Exception): هنگامیکه Exception رخ دهد دسترسی به هسته انجام میشود تا خطا رفع شود و پس از آن به سطح کاربر باز میگردد
socket based: برنامه های سطح کاربر در این حالت اطلاعات را از طریق سوکت دریافت میکند و مستقیماً با هسته ارتباط میگیرد .

۳- خیر ؛ با سطح دسترسی DLP_USER نمی توان تمامی تله ها را فعال کرد. چراکه برخی از تله ها نیاز به دسترسی بالاتری دارند و برای اجرای آنها نیاز به سطح دسترسی بیشتری است. همچنین، تله هایی که به صورت خودکار و بدون دسترسی کاربر اجرا می شوند نیز نمی توانند توسط سطح دسترسی DLP_USER محدود شوند. به علاوه، اگر کاربران با سطح دسترسی DLP_USER به منابع حساس دسترسی پیدا کنند، ممکن است خطرات امنیتی افزایش یابد و به منابع حساس آسیب وارد شود .

۴- در کل دو پشته داریم یکی برای سطح کاربر و دیگری برای سطح هسته. وقتی سطح دسترسی تغییر پیدا کند مثلاً از سطح کاربر به هسته تغییر یابد دیگر نمیتوان از پشته قبلی استفاده کرد .
لذا باید ss و esp روی پشته push بشوند تا برای بازگشت به آن سطح اطلاعات از بین نرود و مجدداً بتوان از آن استفاده کرد. به همین ترتیب وقتی سطح دسترسی ثابت بماند به همان پشته دسترسی لازم وجود داشته و نیازی به push کردن ss و esp نیست .

۵- argint ، argstr و argptr از توابع مورد استفاده در برنامه‌نویسی سیستم عامل هستند که برای دریافت آرگومان‌های فراخوانی سیستم استفاده می‌شوند .

argint : برای دریافت یک عدد صحیح به عنوان آرگومان استفاده می‌شود .
argstr : برای دریافت یک رشته به عنوان آرگومان استفاده می‌شود .
argptr : برای دریافت یک اشاره‌گر به عنوان آرگومان استفاده می‌شود .

بررسی بازه آدرس‌ها در تابع `argptr` از تجاوز از حافظه جلوگیری می‌کند. اگر این بررسی انجام نشود، برنامه ممکن است به طور ناخواسته به یک آدرس غیرمعتبر دسترسی پیدا کند و این باعث بروز خطای `segmentation fault` می‌شود و اجرای سیستم را مختل می‌کند.

به عنوان مثال، فرض کنید که یک برنامه کاربردی `sys_read` را فراخوانی می‌کند و پارامترهای آن به شکل زیر است:

```
int fd = 0; // stdin
void *buf = malloc(1024);
size_t count = 1024;
```

اگر برنامه‌نویس از تابع `argptr` برای بررسی صحت و درستی آدرس پوینتر به `buf` استفاده نکند و به جای آن، مستقیماً آدرس `buf` به `sys_read` منتقل کند، برنامه ممکن است به یک آدرس غیرمعتبر دسترسی پیدا کند و خطای `segmentation fault` را تجربه کند. این مشکل می‌تواند با استفاده از تابع `argptr` برای بررسی صحت آدرس پوینتر به `buf`، جلوگیری شود.

بررسی گام‌های فراخوانی‌های سیستمی در سطح کرنل توسط `gdb`

نحوه افزودن یک برنامه سطح کاربر: فایل `C`. برنامه را در پوشه قرار می‌دهیم، در داخل `makefile` نام فایل را به متغیر `EXTRA` اضافه می‌کنیم، و سپس در `UPROGS` نام را بدون پسوند فایل و با افزودن `_` به ابتدای آن اضافه می‌کنیم.

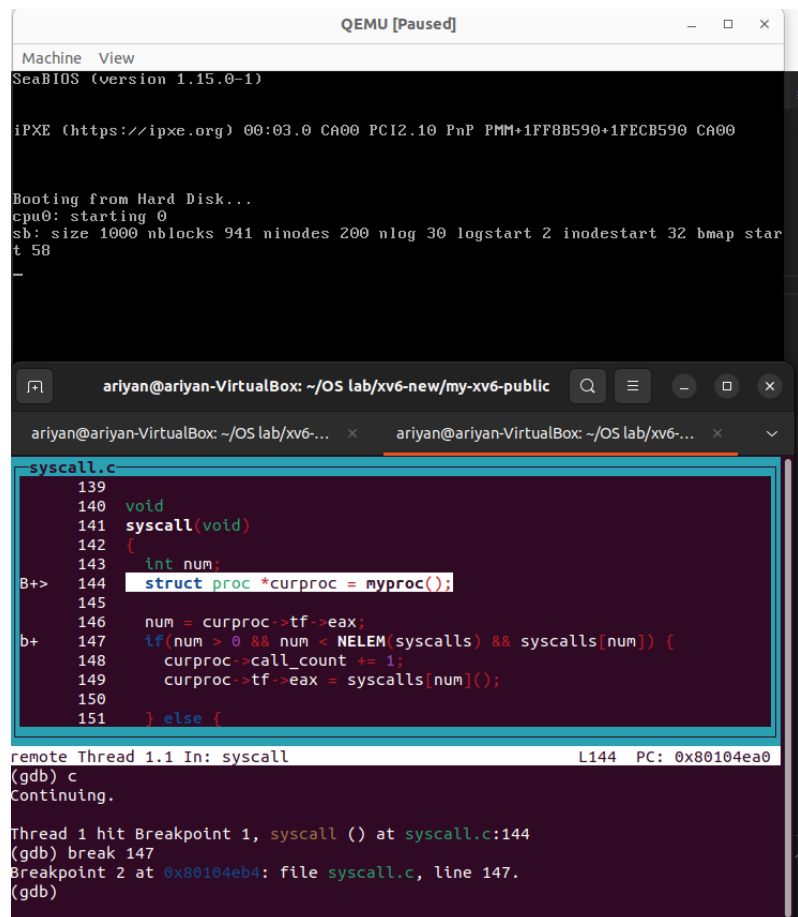
```
255 EXTRA=\
256     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
257     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
258     fibo.c\
259     most.c\
260     alive.c killfirst.c\
261     task1.c\
262     printf.c umalloc.c\
263     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
264     .gdbinit.tmpl gdbutil\
265
```

```
168 UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184     _fibo\
185     _most\
186     _alive\
187     _killfirst\
188     _task1\
189
```

نام برنامه سطح کاربر task1.c

```
C task1.c > main(int, char *[])
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char *argv[]) {
6      getpid();
7      exit();
8      return 0;
9  }
```

ایجاد break point: در همان بوت شدن سیستم چندین بار به این نقطه برخورد میکنیم (یک breakpoint هم در خط ۱۴۷ برای نمایش بهتر محتوای رجیستر اضافه کرده ایم).



The screenshot shows a QEMU [Paused] window at the top with the SeaBIOS (version 1.15.0-1) boot screen. Below it is a terminal window showing the execution of a program. The GDB debugger window is open, displaying the source code of syscall.c. A breakpoint has been set at line 147, and the debugger shows that Thread 1 hit Breakpoint 1 at syscall.c:147. The GDB prompt is (gdb).

```
QEMU [Paused]
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
--

ariyan@ariyan-VirtualBox: ~/OS lab/xv6-new/my-xv6-public
ariyan@ariyan-VirtualBox: ~/OS lab/xv6-... x ariyan@ariyan-VirtualBox: ~/OS lab/xv6-... x
syscall.c
139
140 void
141 syscall(void)
142 {
143     int num;
144     struct proc *curproc = myproc();
145
146     num = curproc->tf->eax;
147     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
148         curproc->call_count += 1;
149         curproc->tf->eax = syscalls[num]();
150     } else {
151         remote Thread 1.1 In: syscall L144 PC: 0x80104ea0
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:144
(gdb) break 147
Breakpoint 2 at 0x80104eb4: file syscall.c, line 147.
(gdb)
```

دستور bt یا همان back trace در واقع stack فراخوانی های انجام شده در کد تا رسیدن به break point را نشان میدهد. (اگر کد های trap.c و trapasm.S را بررسی کنیم به روند آن پی میبریم، البته فایل S. اسمبلی می باشد)

```
remote Thread 1.1 In: syscall
Thread 1 hit Breakpoint 1, syscall () at syscall.c:144
(gdb) break 147
Breakpoint 2 at 0x80104eb4: file syscall.c, line 147.
(gdb) bt
#0  syscall () at syscall.c:144
#1  0x80105f7d in trap (tf=0x8dffffb4) at trap.c:43
#2  0x80105d1e in alltraps () at trapasm.S:20
(gdb)
```

اجرای down و چاپ محتوای eax: چون در انتهای استک هستیم نمیتوانیم پایین برویم، دستور down به نقطه پایینی استک می رود (مخالف دستور up)، محتوای رجیستر eax عدد و آدرسی نا معلوم هست

```
#2  0x80105d1e in alltraps () at trapasm.S:20
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) i r eax
eax          0x80112d54          -2146357932
(gdb)
```

اما اگر در خط ۱۴۷ اقدام به چاپ eax بکنیم:

```
Thread 1 hit Breakpoint 2, syscall () at syscall.c:147
(gdb) i r eax
eax          0x7              7
(gdb)
```

عدد ۷ که همان معادل sys_exec هست چاپ می شود (برای چاپ رجیستر میتوانستیم داخل کد هم با cprintf اقدام کنیم که ما از break point جدید استفاده کردیم)، در ادامه دستور C را آنقدر اجرا میکنیم که qemu از ما ورودی بخواهد و در این مرحله task1 را اجرا میکنیم سپس آنقدر C را اجرا میکنیم که محتوای رجیستر برابر ۱۲ (شماره getpid) شود.

```
ariyan@ariyan-VirtualBox: ~/OS lab/xv6-new/my-xv6-public
ariyan@ariyan-VirtualBox: ~/OS lab/xv6... x ariyan@ariyan-VirtualBox: ~/OS lab/xv6... x
syscall.c
142 {
143     int num;
144     struct proc *curproc = myproc();
145
146     num = curproc->tf->eax;
147     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
148         curproc->call_count += 1;
149         curproc->tf->eax = syscalls[num]();
150     } else {
151         cprintf("%d %s: unknown sys call %d\n",
152             curproc->pid, curproc->name, num);
153         curproc->tf->eax = -1;
154     }
155 }

remote Thread 1.1 In: syscall
eax          0x3              3
(gdb) c
Continuing.

Thread 1 hit Breakpoint 3, syscall () at syscall.c:147
(gdb) i r eax
eax          0xc              12
(gdb)
```

ارسال آرگومان های سیستمی (فیبوناچی)

افزودن یک سیستم کال به xv6:

در `syscall.h` باید شماره ای به سیستم کال اختصاص داد، سپس در `syscall.c` امضای تابع را اضافه میکنیم و داخل آرایه `*syscall[]` این سیستم کال را مطابق بقیه اضافه میکنیم، در مرحله بعد داخل `sysproc.c` تابع سیستم کال را پیاده میکنیم (ما در فایل های خود، تابع اصلی را در `proc.c` تعریف کرده ایم (و امضای آن در `defs.h`) و سپس در `sysproc` آن را صدا زده ایم (برای راحتی استفاده از `ptable`)

```
22 #define SYS_close 21
23 #define SYS_fibo 22
24 #define SYS_most 23
25 #define SYS_alive 24
26 #define SYS_killfirst 25

105 extern int sys_uptime(void);
106 extern int sys_fibo(void);
107 extern int sys_most(void);
108 extern int sys_alive(void);
109 extern int sys_killfirst(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_fibo] sys_fibo,
134     [SYS_most] sys_most,
135     [SYS_alive] sys_alive,
136     [SYS_killfirst] sys_killfirst,
137 };
138
```

```

92
93 int sys_fibo(void){
94     //rsi and rdi commonly used to pass 2 arguments to system calls
95     //rax: system calls return their results in this register
96     //rbx,rcx,rdx: various use cases, can be used to pass system call arguments
97     int n = myproc()->tf->ebx;
98     cprintf("sys_fibo: %d\n",n);
99     return find_fibonacci_number(n);
100 }
101
102 int sys_most(void){
103     cprintf("sys_most called\n");
104     return find_most_callee();
105 }
106
107 int sys_alive(void){
108     cprintf("sys_alive called\n");
109     return get_alive_children_count();
110 }
111
112 int sys_killfirst(void){
113     cprintf("sys_killfirst called\n");
114     return kill_first_child_process();
115 }

```

```

537 //lab2
538
539 int find_fibonacci_number (int n){
540     if(n==0||n==1){
541         return n;
542     }
543     return find_fibonacci_number(n-1) + find_fibonacci_number(n-2);
544 }
545

```

```

188
189 //lab2
190 int find_fibonacci_number (int n);
191 int find_most_callee(void);
192 int get_alive_children_count(void);
193 int kill_first_child_process(void);
194

```

در ادامه در `usys.S` مطابق الگوی موجود سیستم کال را اضافه میکنیم و در مرحله نهایی در `user.h` مطابق الگوی اسم فایل یک امضای تابع سیستم کال را اضافه میکنیم.

```

31 SYSCALL(uptime)
32 SYSCALL(fibo)
33 SYSCALL(most)
34 SYSCALL(alive)
35 SYSCALL(killfirst)
36

```

```

26 int fibo(void); //void?
27 int most(void);
28 int alive(void);
29 int killfirst(void);
30

```

نحوه ارسال و دریافت ورودی به کمک رجیسترها: در برنامه سطح کاربر (fibo.c) آرگومان را از command line میگیریم (argv[]) سپس به کمک asm دستور اسمبلی اجرا میکنیم که محتوای کنونی ebx را در prev_ebx ذخیره کند و n را جای آن قرار دهد. (دو دستور اسمبلی inline)

در داخل تابع sys_fibo به کمک myproc() و tf مقدار رجیستر ebx را میگیریم و تابع فیبوناچی را اجرا میکنیم. بعد برگشت به برنامه سطح کاربر، با کد اسمبلی مقدار prev_ebx را در داخل ebx قرار می دهیم (بازیابی)

```

C fibo.c > main(int, char *[])
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main (int argc, char *argv[]) {
6      int prev_ebx=0, n = atoi(argv[1]); //get argument from command line
7      asm volatile(
8          "movl %%ebx, %0;" // store current ebx in prev_ebx
9          "movl %1, %%ebx;" // put n in ebx
10         : "=r" (prev_ebx) //output
11         : "r"(n) //input
12         );
13     printf(2, "fibo called from user side: %d, result: %d\n", n, fibo());
14     asm("movl %0, %%ebx" : : "r"(prev_ebx)); // restore ebx
15     exit();
16

```

۱- پر استفاده ترین فراخوانی سیستمی

تمام مراحل (افزودن برنامه سطح کاربر و فراخوانی سیستمی) مشابه قبل انجام میشوند (فایل most.c برنامه سطح کاربر)، اینبار در proc.h به ساختار proc عدد call_amount را اضافه میکنیم، سپس در proc.c در allocproc مقدار صفر را به آن میدهیم (مقدار اولیه)، هر موقع تابع syscall در syscall.c صدا زده شد، به call_count یکی اضافه میکنیم.

در پیاده سازی اصلی تابع در proc.c جدول ptable را acquire میکنیم و با iterate کردن آن، پردازش با بیشترین مقدار call_count را برمیگردانیم (pid). باید release در جای مناسبی انجام شود.

```
545
546 int find_most_callee(void){
547     int max_pid = -1,max_count = 0;
548     struct proc *p;
549     acquire(&ptable.lock);
550     for(p = ptable.proc;p<&ptable.proc[NPROC];p++){
551         //cprintf("%d %d",p->call_count,p->pid);
552         if(p->call_count > max_count){
553             max_count = p->call_count;
554             max_pid = p->pid;
555         }
556     }
557     release(&ptable.lock);
558
559     cprintf("killed: %d\n",kill_first_child_process());
560     cprintf("\n");
561     return max_pid;
562 }
```

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

۲- تعداد فرزندان پردازش کنونی

تمام مراحل (افزودن برنامه سطح کاربر و فراخوانی سیستمی) مشابه قبل انجام میشوند (فایل alive.c برنامه سطح کاربر)، مشابه تابع قبلی ptable را پیمایش میکنیم و تعداد فرزندی که زنده باشند (killed == 0) را می شماریم. (فرزندان مستقیم)


```

563
564 int get_alive_children_count(void){
565     int count = 0;
566     struct proc *current_process= myproc(),*p;
567     acquire(&ptable.lock);
568     for(p = ptable.proc;p<&ptable.proc[NPROC];p++){
569         if(p->parent->pid == current_process->pid && p->killed ==0){//direct children,
570             count++;
571         }
572     }
573     release(&ptable.lock);
574
575     return count;
576 }

```

```

1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main (int argc,char *argv[]){
6      for(int i = 0;i<3;i++){
7          int pid = fork();
8          if(pid == 0){
9              exit();//child process
10         }else if (pid<0){
11             printf(1,"Fork Failed!\n");
12         }
13     }
14
15     printf(1, "alive children count:%d\n",alive());
16     for(int i = 0;i<3;i++){
17         wait();//wait for children to exit (kill zombies)
18     }
19     exit();
20
21 }

```

۳- کشتن اولین فرزند پردازش کنونی

تمام مراحل (افزودن برنامه سطح کاربر و فراخوانی سیستمی) مشابه قبل انجام میشوند (فایل killfirst.c برنامه سطح کاربر)، مشابه تابع های قبلی ptable را پیمایش میکنیم و وقتی به اولین فرزند رسیدیم تابع kill را روی آن صدا میزنیم، جدول را release کرده و pid فرزند کشته شده را برمیگردانیم (با توجه به مقدار دهی اولیه، در صورت نبود فرزند 1- برگردانده می شود)

```

7
8 int kill_first_child_process(void){
9     struct proc *current_process= myproc(),*p;
10    acquire(&ptable.lock);
11    for(p = ptable.proc;p<&ptable.proc[NPROC];p++){
12        if(p->parent->pid == current_process->pid){//direct children, not decendants
13            release(&ptable.lock);
14            kill(p->pid);
15            return p->pid;
16        }
17    }
18    release(&ptable.lock);
19    return -1;//no children
20 }

```

```

C killfirst.c > main(int, char *[])
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main (int argc, char *argv[]) {
6      int pid = fork();
7      if(pid == 0){
8          sleep(10);//sleep 3 seconds
9
10         exit();//child process
11     } else if (pid < 0){
12         printf(0, "Fork Failed!\n");
13         exit();
14     }
15     printf(1, "(before) children count: %d\n", alive());
16     int killed = killfirst();
17     if(killed == -1){
18         printf(0, "No children!\n");
19     } else {
20         printf(1, "killed pid: %d\n", killed);
21     }
22     printf(1, "(after) children count: %d\n", alive());
23     wait();
24     exit();
25 }

```