

اعضا: سید احمد رکنی حسینی ۸۱۰۱۰۰۱۵۴ محمدعلی شاهین فر ۸۱۰۱۰۰۱۶۹ امیرحسین راحتی ۸۱۰۱۰۰۱۴۴

سوال اول :

این کتابخانه ها توابع و سیستم کال های پایه در xv6 را در خود دارند

ulib.o: در این کتابخانه توابعی I/O مثل read , write , open , close وجود دارد که در سطح یوزر اجرا میشوند. همچنین توابعی مثل fork , signal , exit , kill هم وجود دارند.

usys.o: توابع در این قسمت برای ارتباط بین سطح یوزر با کرنل برای انجام سیستم کال استفاده میشود . توابع این فایل به عنوان کنترل کننده سیستم کال ها در نظر گرفته میشوند

Printf.o: توابعی برای نوشتن در کنسول ارائه میدهد که توسط یوزر قابل دسترسی و اجرا هستند.

umalloc.o: توابع مربوط به تخصیص حافظه در این بخش وجود دارد و امکان مدیریت حافظه را ارائه میدهد. برخی توابع آن , free , malloc realloc هستند.

سوال دوم :

Linux has some API we can use for access include system calls, pseudo-files, and libc functions. for example, every ubuntu installation requires 224 system calls, 204 ioctl,fcntl and prctl codes and hundreds of pseudo files. system Library APIs: programmers program to more user friendly API in libc and other library. libc exporting an API like POSIX as well as the primary way application developers interact with the OS kernel pseudo files: in the /proc, /dev and /sys. these are called pseudo-file systems because they are not backed by disk, but rather export the contents of kernel data structures to an application or administrator as if they were stored in a file

سوال سوم :

in the XV6 operating system, the DPL_USER access level refers to user mode, which is a lower privilege level compared to kernel mode. When a trap occurs in user mode, the processor switches to kernel mode to handle the trap. However, the XV6 kernel does not allow traps to be enabled while in user mode.

Enabling traps in user could potentially lead to security vulnerabilities or instability in the system. By restricting trap handling to kernel mode, XV6 ensures that critical operations and privileged instructions are only executed by the kernel, preventing unauthorized access or misuse.

So, to maintain the system's security and reliability, XV6 restricts the ability to enable traps in DPL_USER access level.

سوال چهارم :

رجیستر های ss (stack segment) و esp (stack pointer) زمانی به استک پوش میشوند که تغییر سطح دسترسی از یوزر به کرنل داشته باشیم. این مقادیر برای این است که قبلی حالت استک یوزر حفظ شود و بعدا بتوانیم آنرا بازیابی کنیم. اگر تغییر دسترسی نداشته باشیم نیازی به پوش کردن این رجیستر ها به استک نیست زیرا وضعیت استک ثابت میماند.

سوال پنجم :

functions accesses the parameters with some pointer math. they access the trap frame struct of the process, containing the user-space registers of the system call that save parameters starting at the esp register. it adds for to skip empty word on the stack.

two checks occur during a call to code argptr. first, the user stack pointer is checked during the fetching of the argument, then the argument, itself a user pointer, is checked.

بررسی اجرای فراخوانی سیستمی در سطح کرنل توسط GDB

ابتدا برنامه سطح کاربر زیر را ایجاد میکنیم که `process id` را چاپ میکند

```
pid.c - final
pid.c
proc.c
trap.c

h x86_64 > trapframe > ds
148 // Layout of the trap frame
149 // hardware and by trapasm.S
150 struct trapframe {
151 // registers as pushed by
152 uint edi;
153 uint esi;
154 uint ebp;
155 uint oesp; // useless
156 uint ebx;
157 uint edx;
158 uint ecx;
159 uint eax;
160

pid.c > main(void)
1 #include "types.h"
2 #include "user.h"
3
4 int main(void)
5 {
6 int my_pid = getpid();
7 printf(1, "process ID: %d\n", my_pid);
8 exit();
9 }
```

سپس `xv6` را با کامند `make qemu-gdb` کامپایل و اجرا میکنیم. در ترمینالی دیگر `gdb` را به `qemu` متصل میکنیم و سطح دسترسی `gdb` باید کرنل تعریف شود.

```
QEMU [Paused]
Machine View
Guest has not initialized the display (yet).

amir@amir: ~/Desktop/final
amir@amir:~/Desktop/final$ gdb
GNU gdb (Ubuntu 12.1-0ubuntu1-22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
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".
warning: File "/home/amir/Desktop/final/.gdbinit" auto-loading has been declined by your 'auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
add-auto-load-safe-path /home/amir/Desktop/final/.gdbinit
line to your configuration file "/home/amir/.config/gdb/gdbinit".
To completely disable this security protection add
set auto-load safe-path /
line to your configuration file "/home/amir/.config/gdb/gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual. E.g., run from the shell:
info "(gdb)Auto-loading safe path"
(gdb)

amir@amir:~/Desktop/final$ make qemu-gdb
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp::26000
```

سپس روی `syscall` با دستور `break syscall` توقف قرار میدهم و با دستور `continue` هردفعه از آن عبور میکنیم. زمانی که دستور `pid` را اجرا میکنیم `breakpoint` هایی که قرار دادیم هیت میشوند. در این بین با دستور `bt` میتوان لیست `backtrace` ها را در استک فراخوانی دید. از دستور `up` استفاده میکنیم تا به محل قرار گیری آبجکت `tf` در استک فراخوانی برویم. با دستور `print tf.eax` محتوای رجیستر `eax` نمایش داده میشود که در بین تمام `breakpoint` هایی که فعال میشوند یکی از آن ها مقدار `pid` ای است که برای ما نمایش داده میشود. بقیه مربوط به `process` های سطح کرنل هستند.

```
QEMU [Paused] - Press Ctrl+Alt+G to release grab
Machine View
SeaBIOS (version 1.15.0-1)
IPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 Fw 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
Amirhossein Rahati
Seyyed$ Ahmad RokniHosseini
MuhammadAli Shahinfar
pid

xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Amirhossein Rahati
Seyyed$ Ahmad RokniHosseini
MuhammadAli Shahinfar
pid

ia=di Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
smp 2 138 struct proc *curproc = myproc();
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138 struct proc *curproc = myproc();
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138 struct proc *curproc = myproc();
(gdb) bt
#0 syscall () at syscall.c:138
#1 0x8010628d in trap (tf=0x8dfffefb4) at trap.c:43
#2 0x8010602f in alltraps () at trapasm.S:20
#3 0x8dfffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) print tf.eax
No symbol "tf" in current context.
(gdb)
```

```
amir@amir: ~/Desktop/final
syscall.c
132 };
133
134 void
135 syscall(void)
136 {
137     int num;
138     struct proc *curproc = myproc();
139
140     num = curproc->tf->eax;
141     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
142         curproc->tf->eax = syscalls[num]();
143     } else {
144         cprintf("%d %s: unknown sys call %d\n",
145             curproc->pid, curproc->name, num);
146     }
147 }
148
remote Thread 1.1 In: syscall L138 PC: 0x80105230
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) c
Continuing.

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

تصویری از هیت شدن breakpoint ها در
حالت layout src که محل فعال شدن
breakpoint را نشان میدهد

نتیجه میشود که مقدار process id هر
process حین اجرا در رجیستر eax ذخیره
میشود.

افزودن چند سیستم کال به کرنل xv6

برای افزودن هر سیستم کال ، باید یک شماره جدید به آن نسبت داد ، فراخوانی های سیستمی را و مکانیزم پاس
دادن توابع را تعیین نمود و در آخر توسط برنامه سطح کاربر به آن دسترسی پیدا کرد.

برخی تعییرات و کد های اضافه شده برای دو سیستم کال : `get_uncle_count()` , `get_process_lifetime()`

```
C syscall.c C proc.c 6 X
C proc.c > get_uncle_count(int)
536 int
537 get_uncle_count(int pid)
538 {
539     struct proc *p;
540     struct proc *grand_parent;
541     int count = 0;
542
543     //acquire(&ptable.lock);
544     for(p= ptable.proc;p< &ptable.proc[NPROC];p++){
545         if(p->pid == pid){
546             grand_parent = p->parent->parent;
547             for(p = ptable.proc;p<&ptable.proc[NPROC];p++){
548                 if(p->parent->pid == grand_parent->pid){
549                     count++;
550                 }
551             }
552         }
553     }
554     return --count;
555 }
```

ابتدا توابع و سیستم کال ها در فایل `proc.c`
تعریف و پیاده سازی میشوند.
سپس در فایل `syscall.c` باید هرکدام از آن
ها تعریف و به کرنل شناسانده شوند

```
C syscall.c C proc.c 6 X
C proc.c > get_process_lifetime(int)
556 int
557 get_process_lifetime(int pid){
558     struct proc *p;
559     for(p= ptable.proc; p< &ptable.proc[NPROC];p++){
560         if(p->pid == pid){
561             return ticks - p->ptime;
562         }
563     }
564     return -1;
565 }
```

در فایل `syscall.h` و `user.h` هر سیستم کال را تعریف و میکنیم و به سطح یوزر و کرنل می شناسانیم.

C `syscall.c 9+` h `syscall.h` X

h `syscall.h` > ...

```
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_get_uncle_count 22
24 #define SYS_get_process_lifetime 23
```

C `syscall.c 9+` h `user.h` 1 X

h `user.h` > ...

```
26 int get_uncle_count(int);
27 int get_process_lifetime(int);
```

C `syscall.c 9+` X C `proc.c 6`

C `syscall.c` > [0] syscalls

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

سیستم کال های اضافه شده را به فایل `sysproc.c` هم اضافه میکنیم. این فایل در واقع محل شناسایی سیستم کال ها و هندل کردن آن ها در سطح یوزر است.

فایل `user.h` هم نقشی مشابه دارد همچنین سیستم کال ها باید در فایل `proc.c` هم تعریف شوند که درواقع برای این است که آنها به عنوان یک `process` تعریف شوند.

نهایتا دو برنامه سطح کاربر برای تست و اجرای سیستم کال ها اضافه میکنیم.

C `syscall.c 9+` C `sysproc.c` X

C `sysproc.c` > ...

```
92 int
93 sys_get_uncle_count(void)
94 {
95     int pid;
96     if(argint(0, &pid) < 0)
97         return -1;
98     int result = get_uncle_count(pid);
99     return result;
100 }
```

C `syscall.c 9+` X C `sysproc.c` X

C `sysproc.c` > ...

```
102 int
103 sys_get_process_lifetime(void)
104 {
105     int pid;
106     if(argint(0, &pid) < 0)
107         return -1;
108     int result = get_process_lifetime(pid);
109     return result;
110 }
```

C `get_process_lifetime_test.c` X

C `get_process_lifetime_test.c` > main(int, char * [])

```
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4 #include "fcntl.h"
5
6 int main(int argc, char* argv[]) {
7     uint pid = fork();
8     if(pid == 0) {
9         sleep(1000);
10        printf(1, "%d\n", get_process_lifetime(getpid()));
11        exit();
12    }
13    else {
14        wait();
15        printf(1, "%d\n", get_process_lifetime(getpid()));
16        exit();
17    }
18 }
```

C `get_uncle_count_test.c` X

C `get_uncle_count_test.c` > main(int, char * [])

```
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4 #include "fcntl.h"
5 int main(int argc, char* argv[]) {
6     uint pid1 = 1, pid2 = 1, pid3 = 1, pid4 = 1;
7     pid1 = fork();
8     if(pid1 == 0) {
9         sleep(100);
10    }
11    else {
12        pid2 = fork();
13        if(pid2 == 0) {
14            sleep(100);
15        }
16    }
```