Operating System

CA2 Report

Sadra Madayeni AVVAL 810102564

AmirHossein Alikhani 810102479

Hasti Abolhasani 810102378

- بخش ۱: ارسال آرگومان از طریق ثبات (Register)

در این بخش، ما یک فراخوانی سیستمی استاندارد را طوری تغییر دادیم که آرگومان‌های خود را به جای Stack، مستقیماً از ثبات‌های (Registers) پردازنده بخواند.

ثبت استاندارد: ما تابع را مانند هر فراخوانی سیستمی دیگری در `syscall.h` (اختصاص شماره) و `syscall.c` (ثبت تابع) و `user.h` (اعلان تابع) ثبت کردیم.

پیاده‌سازی هسته (در `sysproc.c`):

- در تابع `sys_simple_arithmetic_syscall`، به جای استفاده از `argint()`، مستقیماً به **قاب تله (Trap Frame)** پردازه فعلی دسترسی پیدا کردیم.

این فراخوانی سیستمی باید محاسبهی سادهی (b+a) * (b-a)را انجام داده و نتیجه ی نهایی را برگرداند.

مشکل اینجا بود که ما نمیتونستیم این تابع رو به صورت عادی در C صدا بزنیم و هرکاری کردم کامپایل نشد در نتیجه تصمیم گرفتیم از کد های اسمبلی درون خطی استفاده کنیم.

درون کد اسمبلی، قبل از اجرای تله (`int $64`):

- شماره فراخوانی سیستمی (`..._SYS`) را در ثبات `EAX` قرار دادیم.
- آرگومان اول (a) را در ثبات `EBX` قرار دادیم.
- آرگومان دوم (b) را در ثبات `ECX` قرار دادیم.

وقتی برنامه کاربر دستور `int $64` را اجرا می‌کند، سخت‌افزار بلافاصله تمام ثبات‌های سطح کاربر (شامل `EAX`, `EBX`, `ECX` و ...) را در مکانی امن در حافظه هسته به نام **قاب تله (Trap Frame)** (که با `myproc()->tf` به آن دسترسی داریم) ذخیره می‌کند.

سپس کنترل به تابع syscall ما در هسته منتقل می‌شود. تابع ما (sys_simple_arithmetic_syscall) به سادگی به این قاب تله نگاه می‌کند، مقادیر ذخیره شده در tf->ebx و tf->ecx را می‌خواند، محاسبه (a-b) *(a+b) را انجام می‌دهد و نتیجه را برمی‌گرداند.

```c
int
main(void)
{
  int a = 5;
  int b = 3;
  int result;

  printf(1, "Testing register-based system call with a=%d, b=%d\n", a, b);

__asm__ __volatile__(
        "movl %1, %%eax\n\t"
        "movl %2, %%ebx\n\t"
        "movl %3, %%ecx\n\t"
        "int $64\n\t"
        "movl %%eax, %0"
        : "=r" (result)
        : "i" (SYS_simple_arithmetic_syscall),
          "r" (a),
          "r" (b)
        : "%eax", "%ebx", "%ecx"
      );

  printf(1, "System call returned: %d\n", result);



  exit();
}
```

- بخش 2 Make Duplicate

هدف این فراخوانی سیستمی، همانطور که در پروژه خواسته شده بود، ایجاد یک کپی کامل از یک فایل ورودی در همان مسیر

و با پسوند `.copy` بود. تمام این عملیات باید به طور کامل در **سطح هسته (Kernel)** انجام می‌شد.

1. نام فایل مبدأ را از کاربر بگیرد.
2. نام فایل مقصد را بسازد (مثلاً `file.txt + .copy`).
3. فایل مبدأ را برای خواندن باز کند.
4. فایل مقصد را ایجاد کند (و اگر وجود داشت، محتوای آن را پاک کند).
5. محتوای مبدأ را بلاک به بلاک بخواند و در مقصد بنویسد.
6. تمام منابع (مانند `inode`ها و حافظه) را آزاد کند.

- منطق اصلی

یک حلقه (`while(off < file_size)`) اجرا کردیم تا کل فایل را بخوانیم.

با اسنفاده از bmap شماره ی بلاک فیزیکی دیسک رو پیدا کردیم

با bread آن بلاک را در یک بافر میخونیم (موقت)

با memmove داده هارا از بافر موقت به بافر اصلی منتقل میکنیم.

و در نهایت بافر موقت را آزاد میکنیم.

از تابع( )writei استفاده کردیم. ما دریافتیم که writei (برخلاف readi) زمانی که با یک بافر kalloc شده در کرنل کار می‌کند، امن است و فایل‌سیستم را خراب نمی‌کند.

در انتهای تابع (و در صورت بروز هرگونه خطا)، هر دو inode را با () iunlockput آزاد کردیم.

تراکنش را با ()end_op بستیم.

حافظه تخصیص داده شده را با (kfree(buf آزاد کردیم.

```
int
main(int argc, char *argv[])
{
  if(argc != 2){
    printf(1, "Usage: makeduptest <filename>\n");
    exit();
  }

  printf(1, "Attempting to duplicate %s...\n", argv[1]);

  if(make_duplicate(argv[1]) == 0){
    printf(1, "SUCCESS: Created %s.copy\n", argv[1]);
  } else {
    printf(1, "ERROR: Failed to duplicate file.\n"); //
  }

  exit();
}
```

Show_Procces_Family

دسترسی به جدول پردازه‌ها (`ptable`) بود که به صورت خصوصی (private) در فایل `proc.c` تعریف شده بود.

تابع `sys_show_process_family` شماره `pid` را با تابعی به نام (argint(0, &pid دریافت میکند

قفل جدول پردازه‌ها را با (`acquire(&ptable.lock` به دست می‌آورد تا از تغییر جدول در حین خواندن جلوگیری کند.

پیدا کردن پردازه: در `ptable.proc` حلقه می‌زند تا پردازه‌ای (`p`) که `p->pid == pid` است را پیدا کند.

چاپ والد: `p->parent->pid` را چاپ می‌کند.

چاپ فرزندان: دوباره در جدول حلقه می‌زند و هر پردازه‌ای (`np`) که `np->parent == p` باشد را چاپ می‌کند.

چاپ برادران: برای بار سوم در جدول حلقه می‌زند و هر پردازه‌ای (np) که `np->parent == p->parent && np != p` باشد را چاپ می‌کند.

در نهایت، قفل را با `release(&ptable.lock)` آزاد می‌کند.

```
int
main(int argc, char *argv[])
{
  if(argc != 2){
    printf(1, "Usage: pidtest <pid>\n");
    exit();
  }

  int pid = atoi(argv[1]); // "2" تبدیل می‌کند 2 را به عدد

  printf(1, "--- Calling show_process_family(%d) ---\n", pid);

  if(show_process_family(pid) < 0){
    printf(1, "--- Error: Process %d not found ---\n", pid);
  } else {
  }

  exit();
}
```

Grep_Syscall

آماده‌سازی: تابع `bmap` (در `fs.c`) را از `static` به عمومی تغییر دادیم تا بتوانیم از آن استفاده کنیم. یک ماکروی `min` تعریف کردیم.

دریافت آرگومان‌ها: `keyword`, `filename`, `user_buffer` را دریافت کردیم.

تخصیص حافظه: یک صفحه حافظه در هسته با `buf = kalloc()` تخصیص دادیم.

باز کردن فایل: کل عملیات را در `begin_op()`/`end_op()` پیچیدیم. `inode` فایل (`ip`) را با `namei()` گرفتیم و آن را `ilock(ip)` کردیم.

حلقه خواندن دستی: یک حلقه

`while(off < ip->size)` اجرا کردیم:

با `bmap(ip)` شماره بلاک داده را پیدا کردیم.

با `bread` بلاک را از دیسک خواندیم.

با `memmove(...)` داده‌ها را از بافر موقت `bread` به بافر اصلی `kalloc` خود کپی کردیم.

و تقریبا مشابه کار هایی که در داپلیکیت کردن کردیم

در بافر `kalloc` خود (که اکنون حاوی کل فایل بود) خط به خط با تابع `simplestr` (که خودمان نوشتیم) جستجو کردیم.

کپی به کاربر: پس از پیدا کردن خط، با `copyout()` آن را به صورت امن از بافر هسته (`buf`) به `user_buffer` منتقل کردیم.

با `kfree(buf)` حافظه را آزاد کردیم.

```c
int
main(int argc, char *argv[])
{
  char buffer[512];
  int len;

  if(argc != 3){
    printf(1, "Usage: grep_test <keyword> <filename>\n");
    exit();
  }

  char *keyword = argv[1];
  char *filename = argv[2];

  printf(1, "Testing grep_syscall: searching for '%s' in '%s'\n", keyword, filename);

  len = grep_syscall(keyword, filename, buffer, sizeof(buffer));

  if (len < 0) {
    printf(1, "Grep ERROR: Keyword not found or file error.\n");
  } else {
    printf(1, "Grep SUCCESS: Found line (len %d): '%s'\n", len, buffer);
  }

  exit();
}
```

- (تابع `set_priority_syscall`)

اصلاح ساختار پردازه (در `proc.h`):

- یک فیلد جدید به نام `int priority` به `struct proc` اضافه کردیم.

در `allocproc` (در `proc.c`)، به تمام پردازه‌های جدید اولویت پیش‌فرض `p->priority = 1;` دادیم

تابع `sys_set_priority_syscall` (در `sysproc.c`) را نوشتیم که با `argint` یک `pid` و `priority` می‌گیرد، `ptable.lock` را قفل می‌کند، پردازه را پیدا کرده و فیلد `p->priority` آن را به‌روزرسانی می‌کند.

تابع `scheduler` را به طور کامل بازنویسی کردیم.

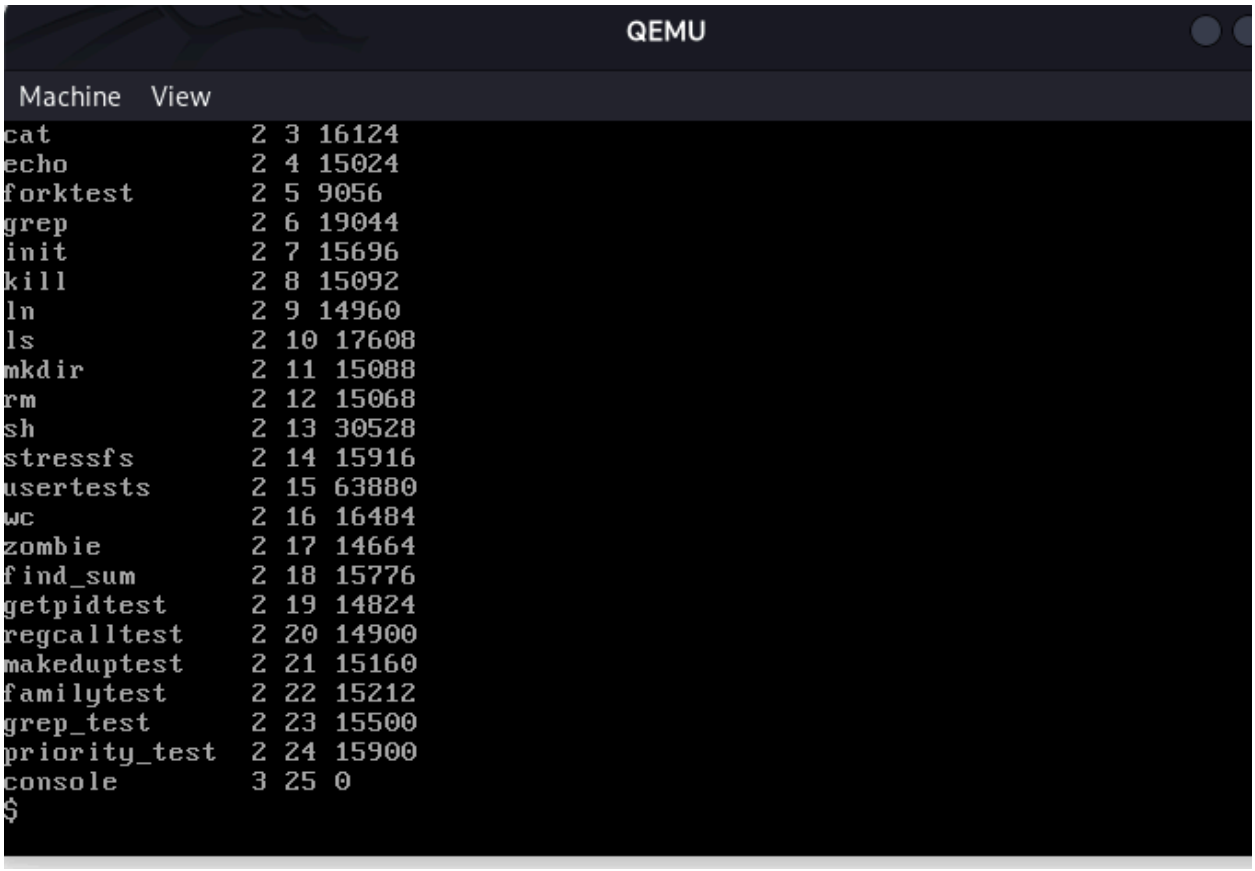
منطق قدیمی: اولین پردازه `RUNNABLE` را پیدا می‌کرد و اجرا می‌کرد.

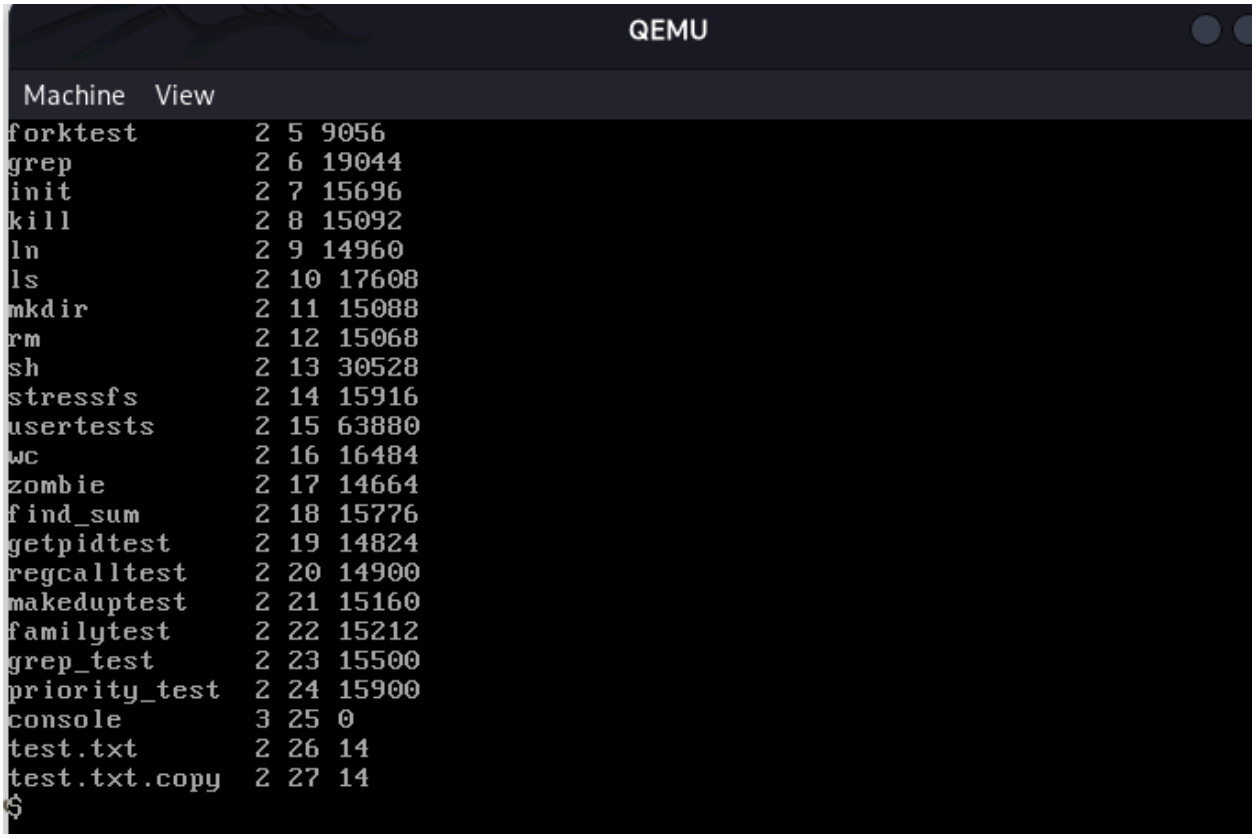
منطق جدید:

- یک متغیر `best_p = 0` (بهترین پردازه) و
- `best_priority` = 4 (عددی بالاتر از بدترین اولویت ما یعنی 2) تعریف کردیم.
- در جدول پردازه‌ها (`ptable.proc`) **به طور کامل** حلقه زدیم.
- اگر پردازه‌ای `RUNNABLE` بود و اولویت آن (`p->priority`) از `best_priority` کمتر بود، آن را به عنوان `best_p` جدید انتخاب می‌کردیم.
- *پس از اتمام حلقه*، پردازه‌ای را که در `best_p` ذخیره شده بود، اجرا می‌کردیم.

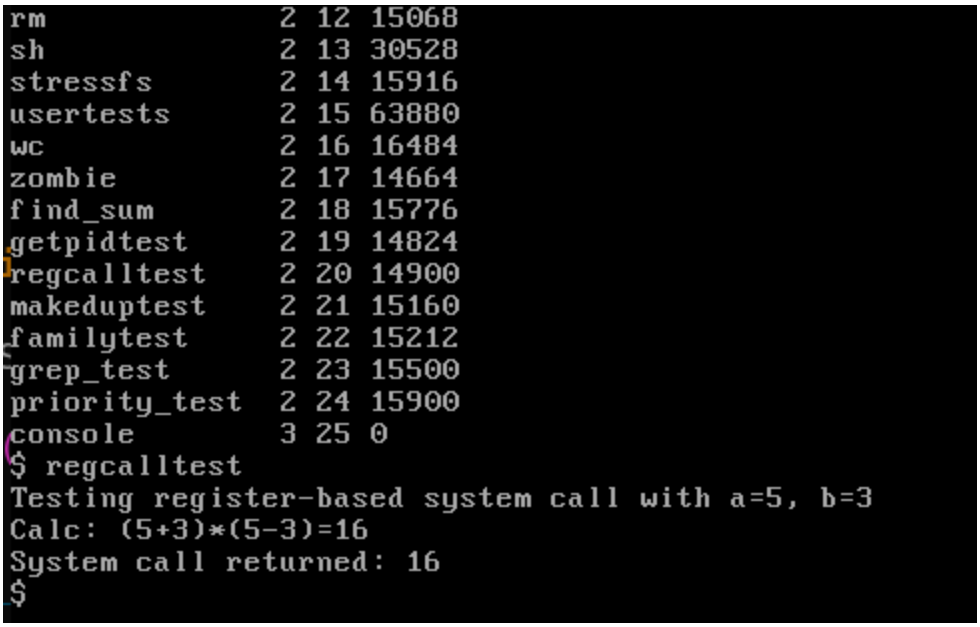
تست کارکرد برنامه

با duplicate شروع میکنیم

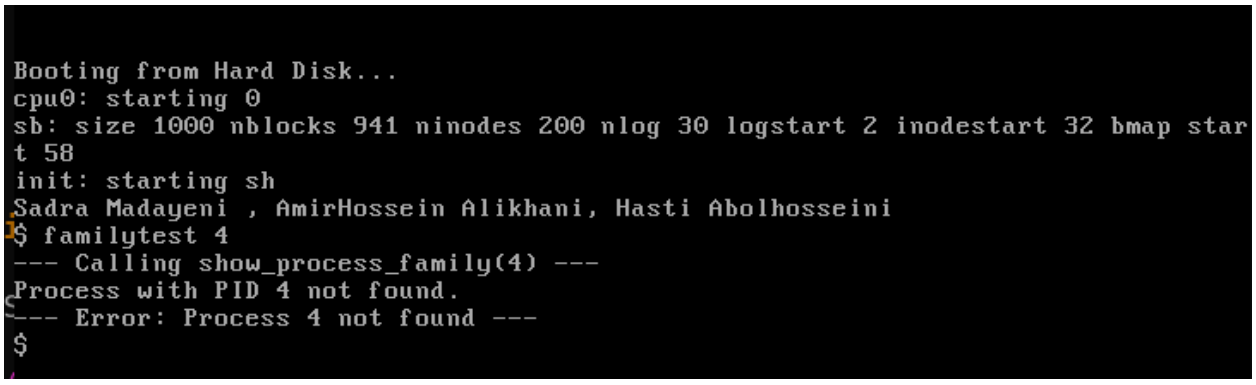



QEMU
Machine View
forktest 2 5 9056
grep 2 6 19044
init 2 7 15696
kill 2 8 15092
ln 2 9 14960
ls 2 10 17608
mkdir 2 11 15088
rm 2 12 15068
sh 2 13 30528
stressfs 2 14 15916
usertests 2 15 63880
wc 2 16 16484
zombie 2 17 14664
find_sum 2 18 15776
getpidtest 2 19 14824
regcalltest 2 20 14900
makeduptest 2 21 15160
familytest 2 22 15212
grep_test 2 23 15500
priority_test 2 24 15900
console 3 25 0
test.txt 2 26 14
test.txt.copy 2 27 14
$

Simple_arithmetic_syscall



Family test



```
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
Sadra Madayeni , AmirHossein Alikhani, Hasti Abolhosseini
$ familytest
Usage: pidtest <pid>
$ familytest 2
--- Calling show_process_family(2) ---
My id: 2, My parent id: 1
Children of process 2:
Child pid: 4
Siblings of process 2:
No siblings found.
$
```

Grep_test

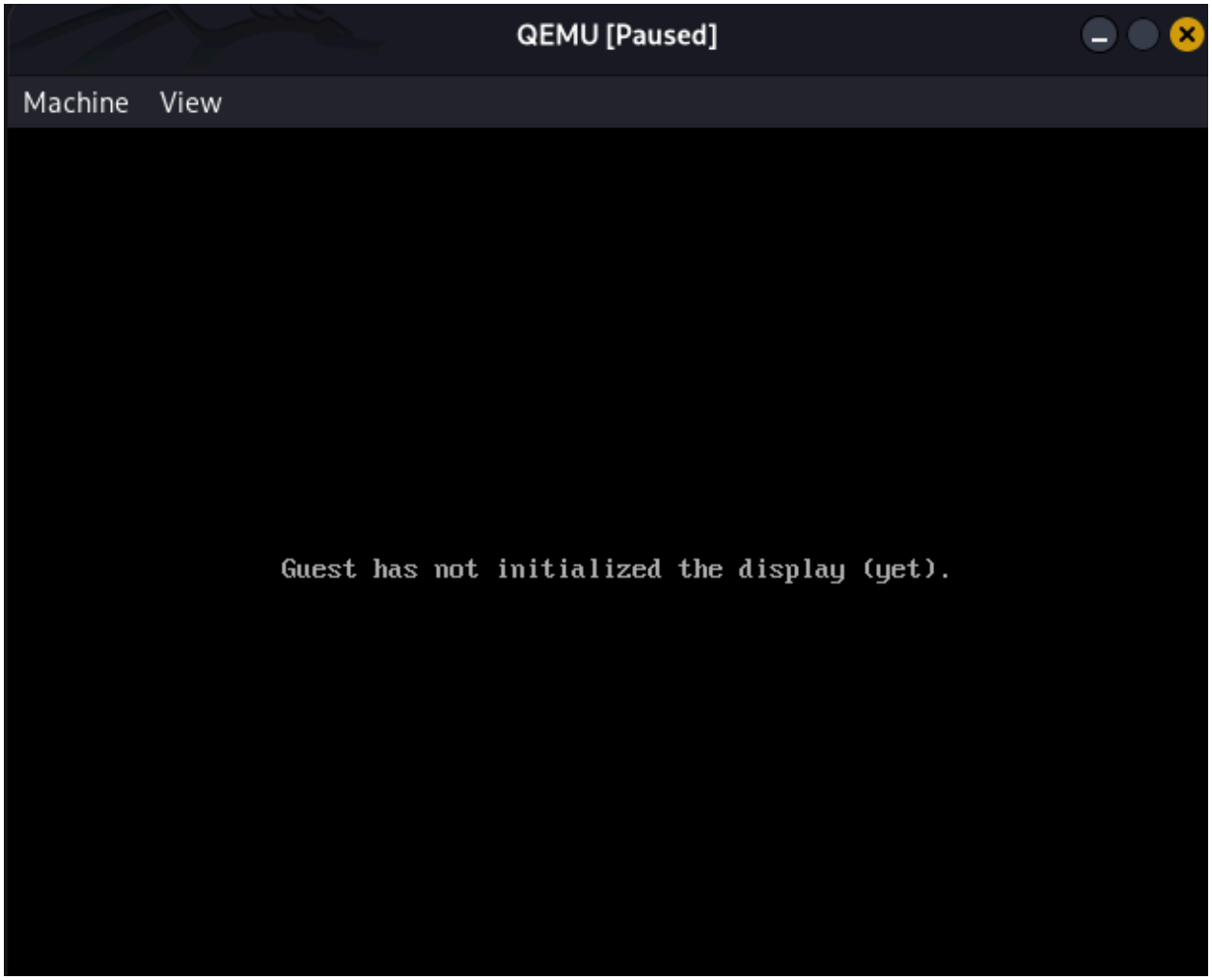
برای تست این فهمیدم که اکوی خود کمو هم خرابه و باید با استفاده از کت یه فایل بسازیم و grep رو تست کنیم

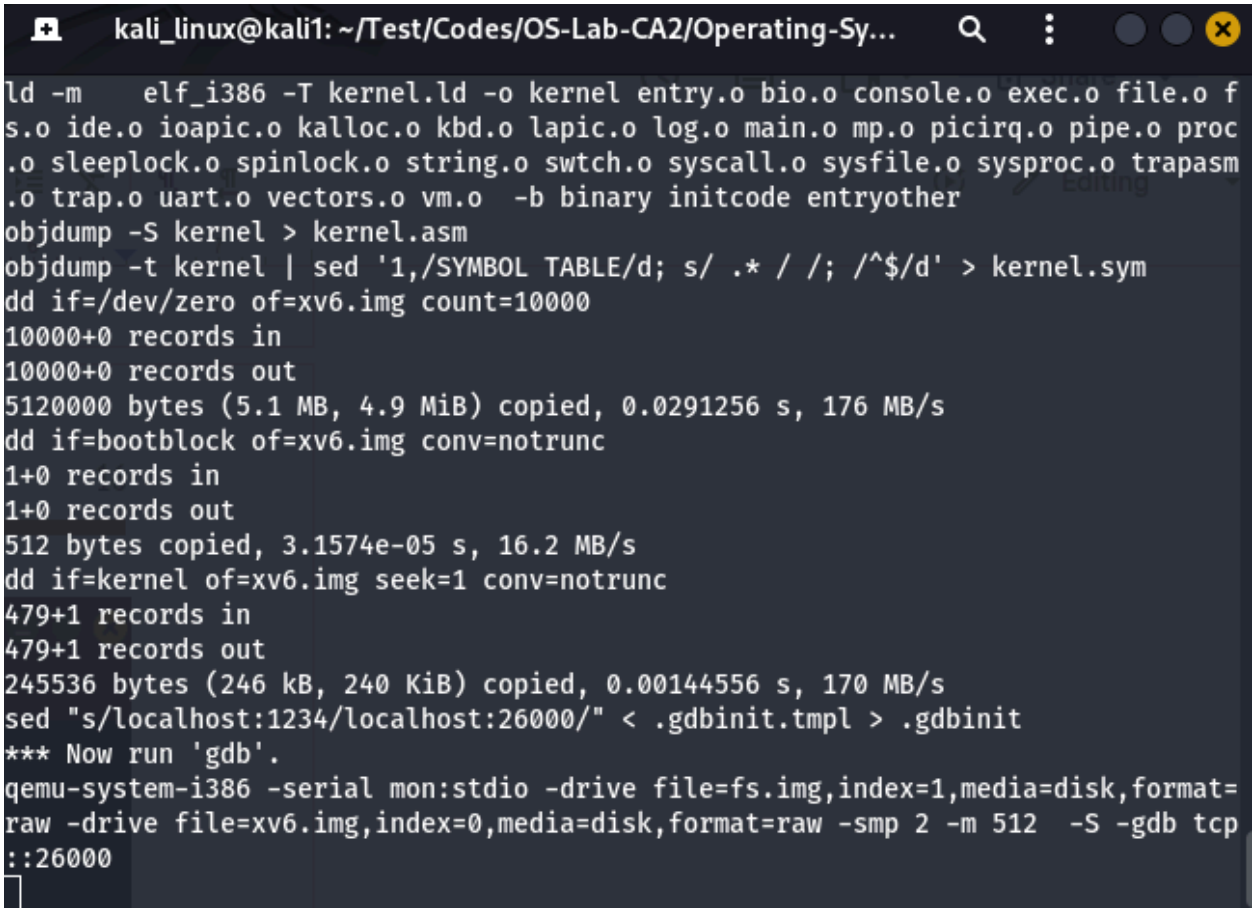
```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodesta
t 58
init: starting sh
Sadra Madayeni , AmirHossein Alikhani, Hasti Abolhosseini
$ cat os.txt
hello
this is a test message
i am writing
report
$ _
```

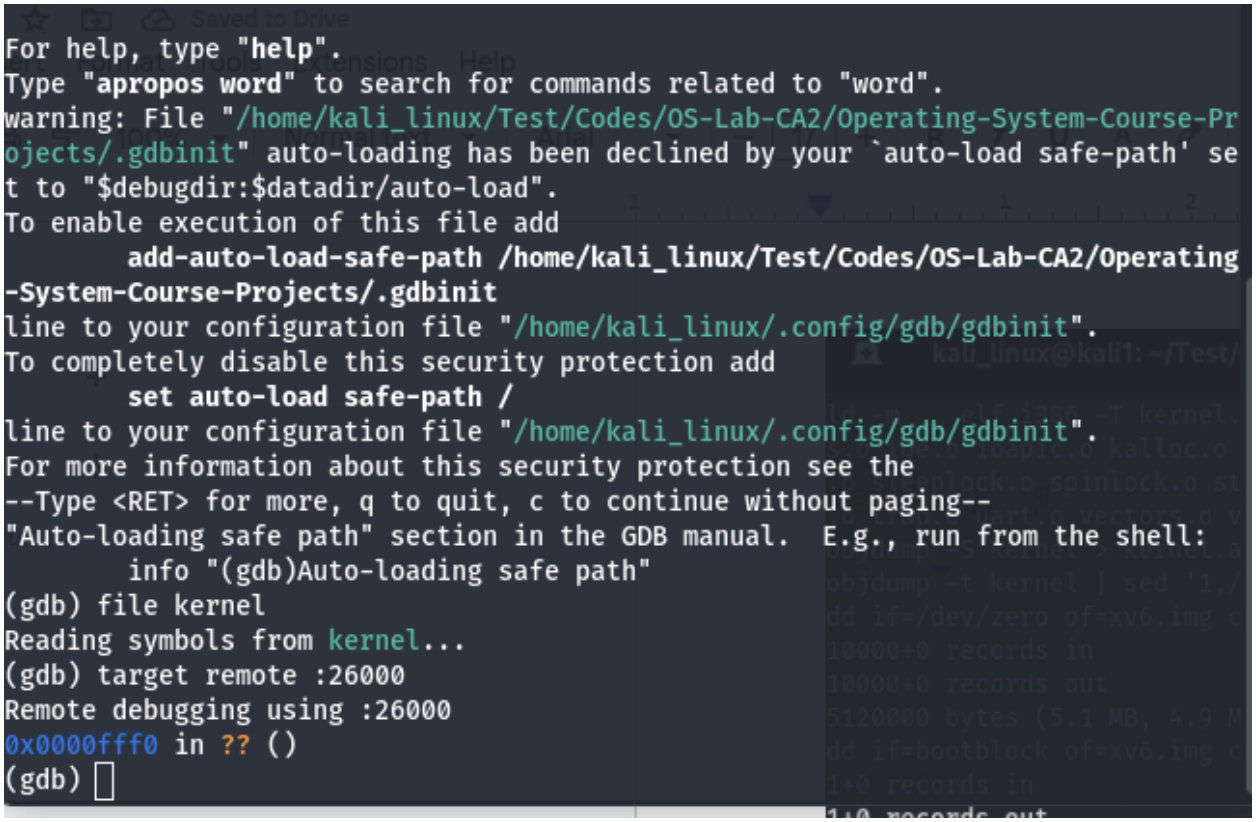
```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodesta
t 58
init: starting sh
Sadra Madayeni , AmirHossein Alikhani, Hasti Abolhosseini
$ cat os.txt
hello
this is a test message
i am writing
report
$ grep_test test os.txt
Testing grep_syscall: searching for 'test' in 'os.txt'
Grep SUCCESS: Found line (len 22): 'this is a test message'
$ _
```

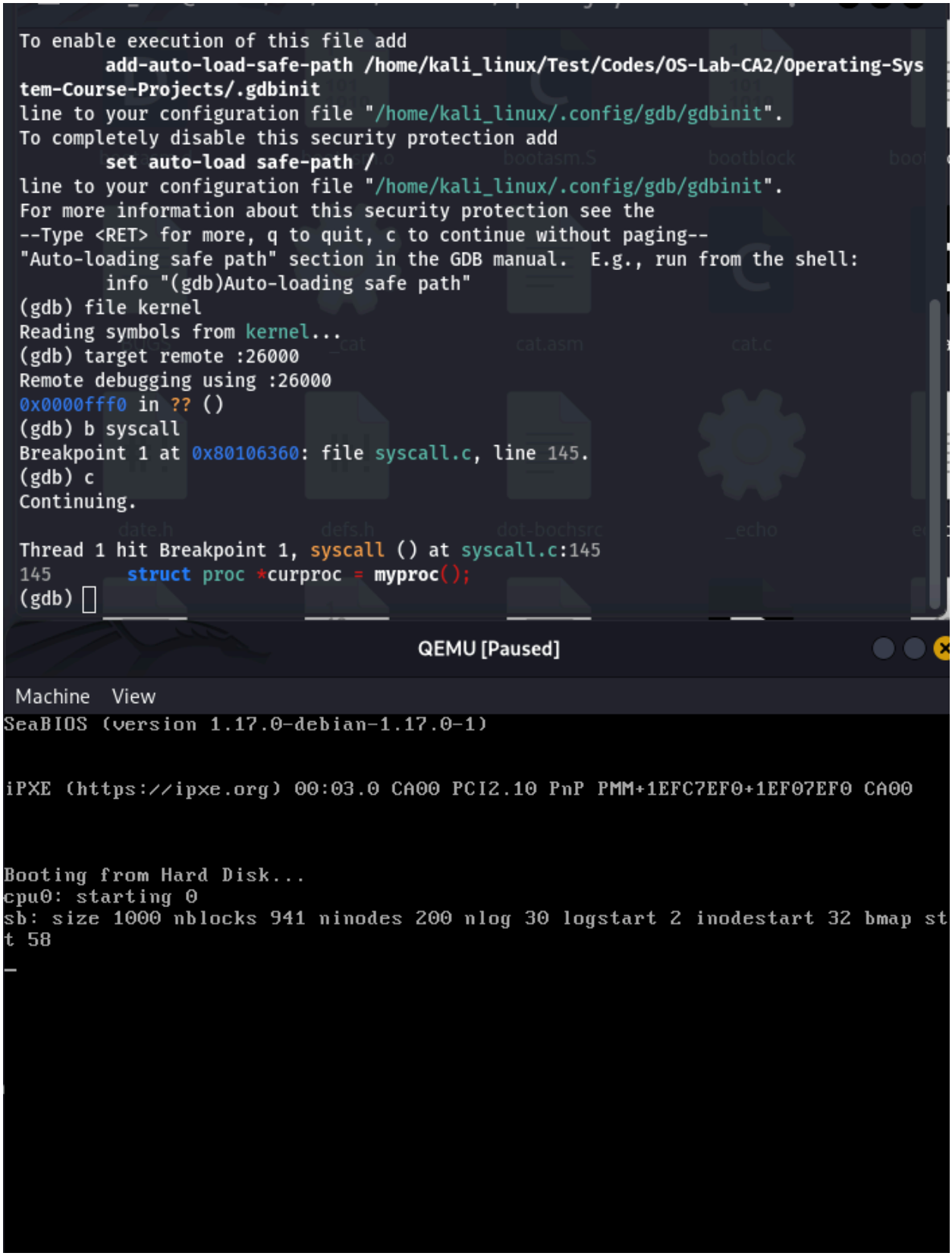
Priority test

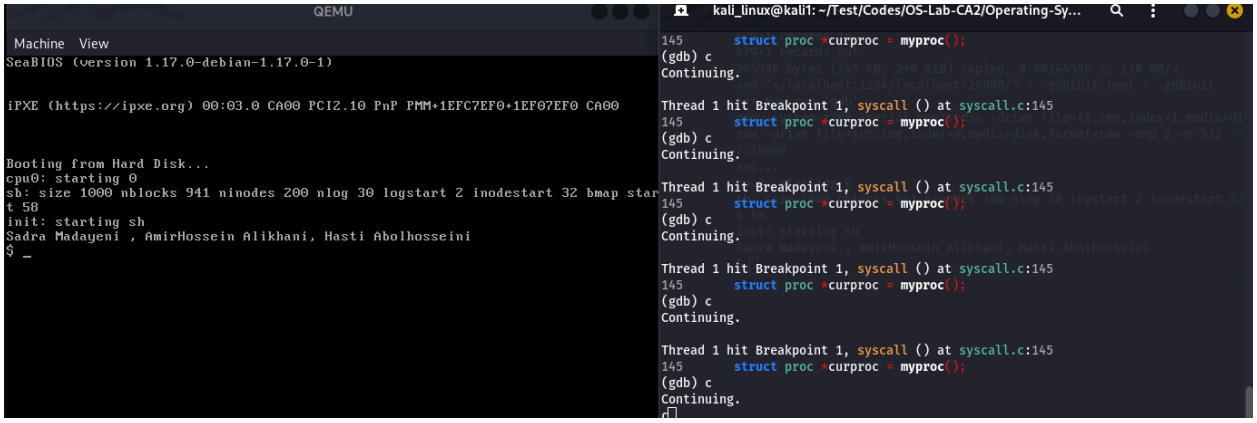
```
$ priority_test
Starting priority test...
Parent waiting for children...
Process Child 2 (High Priority) still running...
Process Child 2 (High Priority) still running...
Process Child 2 (High Priority) still running...
Process Child 2 (High Priority) still running...
--- Process Child 2 (High Priority) FINISHED ---
Process Child 1 (Low Priority) still running...
Process Child 1 (Low Priority) still running...
Process Child 1 (Low Priority) still running...
Process Child 1 (Low Priority) still running...
--- Process Child 1 (Low Priority) FINISHED ---
Priority test finished.
```


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

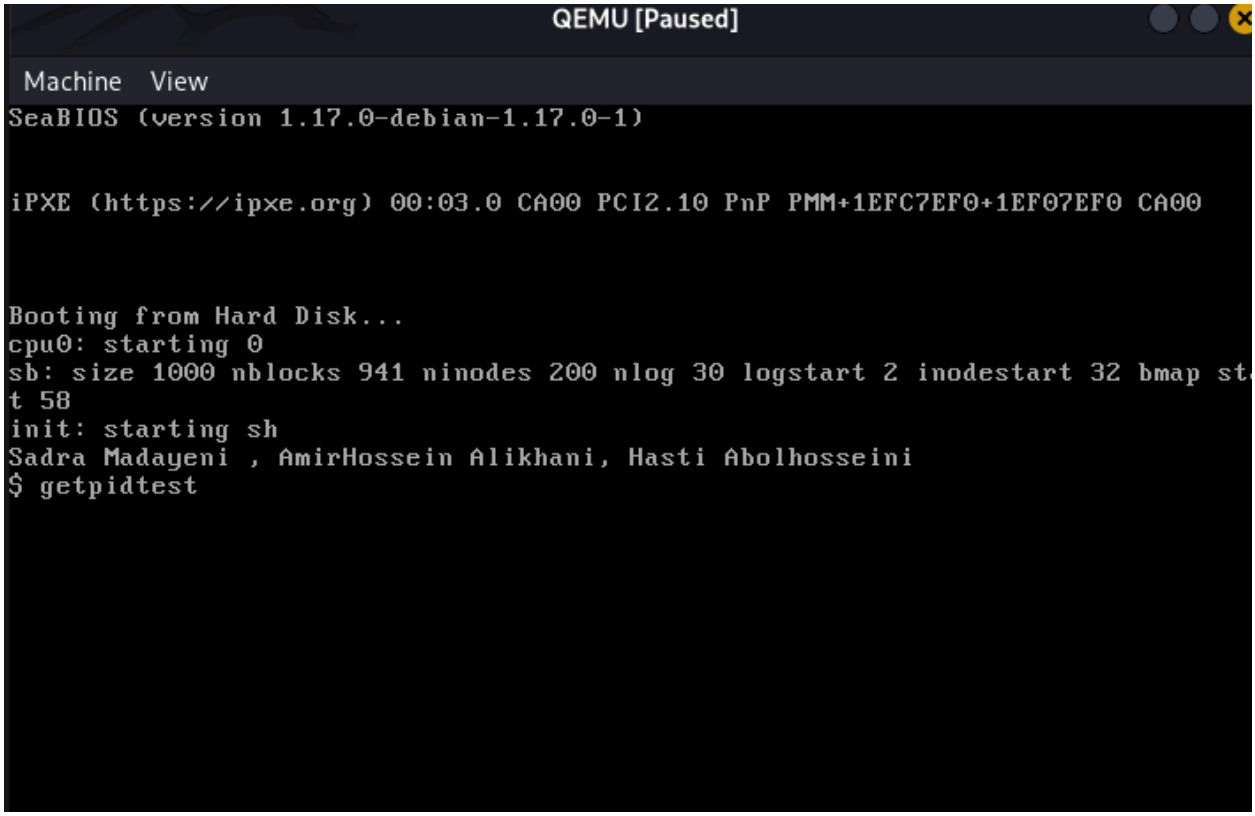

kali_linux@kali1: ~/Test/Codes/OS-Lab-CA2/Operating-Sy...
ld -m   elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0291256 s, 176 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 3.1574e-05 s, 16.2 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
479+1 records in
479+1 records out
245536 bytes (246 kB, 240 KiB) copied, 0.00144556 s, 170 MB/s
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp::26000


For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: File "/home/kali_linux/Test/Codes/OS-Lab-CA2/Operating-System-Course-Projects/.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/kali_linux/Test/Codes/OS-Lab-CA2/Operating-System-Course-Projects/.gdbinit
line to your configuration file "/home/kali_linux/.config/gdb/gdbinit".
To completely disable this security protection add
set auto-load safe-path /
line to your configuration file "/home/kali_linux/.config/gdb/gdbinit".
For more information about this security protection see the
--Type <RET> for more, q to quit, c to continue without paging--
"Auto-loading safe path" section in the GDB manual. E.g., run from the shell:
info "(gdb)Auto-loading safe path"
(gdb) file kernel
Reading symbols from kernel...
(gdb) target remote :26000
Remote debugging using :26000
0x0000fff0 in ?? ()
(gdb)


To enable execution of this file add
add-auto-load-safe-path /home/kali_linux/Test/Codes/OS-Lab-CA2/Operating-System-Course-Projects/.gdbinit
line to your configuration file "/home/kali_linux/.config/gdb/gdbinit".
To completely disable this security protection add
set auto-load safe-path /
line to your configuration file "/home/kali_linux/.config/gdb/gdbinit".
For more information about this security protection see the
--Type <RET> for more, q to quit, c to continue without paging--
"Auto-loading safe path" section in the GDB manual. E.g., run from the shell:
info "(gdb)Auto-loading safe path"
(gdb) file kernel
Reading symbols from kernel...
(gdb) target remote :26000
Remote debugging using :26000
0x0000fff0 in ?? ()
(gdb) b syscall
Breakpoint 1 at 0x80106360: file syscall.c, line 145.
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145 struct proc *curproc = myproc();
(gdb)
QEMU [Paused]
Machine View
SeaBIOS (version 1.17.0-debian-1.17.0-1)
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFC7EF0+1EF07EF0 CA00
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58



بعد از حدود 40 بار زدن دستور سی تازه برامون سیستم عامل کمو از حالت توقف بیرون میاد



بلافاصله پس از زدن Enter، پنجره QEMU **دوباره "Paused" می‌شود**.

حالا با استفاده از دستور bt میتونیم استک فراخوانی رو ببینیم

`syscall` در بالا، `trap` در زیر آن، و `alltraps` در زیر `trap`

```
(gdb) bt
#0  syscall () at syscall.c:145
#1  0x80107b6d in trap (tf=0x8dffefb4) at trap.c:43
#2  0x801078cc in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) p tf->eax
```

```
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) p tf->eax
No symbol "tf" in current context.
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb)
```

دستور down اجرا نمیشود چون در پایین ترین حالت هستیم

باید درستور up باشد

```
(gdb) up
#1  0x80107b6d in trap (tf=0x8dffefb4) at trap.c:43
43          syscall();
(gdb) p tf->eax
$1 = 5
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       struct proc *curproc = myproc();
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       struct proc *curproc = myproc();
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       struct proc *curproc = myproc();
(gdb) p tf->eax
No symbol "tf" in current context.
(gdb) up
#1  0x80107b6d in trap (tf=0x8dfbefb4) at trap.c:43
43          syscall();
(gdb) p tf->eax
$3 = 16
(gdb)
```

```
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_simple_arithmetic_syscall 22
#define SYS_make_duplicate 23
#define SYS_show_process_family 24
#define SYS_grep_syscall 25
#define SYS_set_priority_syscall 26
```

کتابخانه های سطح کاربر مثل usys.S و ulib.c با بهره گیری از ماکرو ها و توابع پوشاننده باعث می شوند تا برنامه نویس اطلاعی از جزییات فراخوانی سیستمی نداشته باشد.
فایل usys.S شامل ماکرو کلیدی SYSCALL این مارکو برای هر فراخوانی سیستمی یک تابع اسمبلی تولید میکند.

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

همانطور که در تصویر بالا مشخص است این ماکرو ابتدا شماره فراخوانی را در رجیستر %eax قرار میدهد این شماره ها در فایل syscall.h قرار گرفتن و سپس با دستور int $T_SYSCALL یک وقفه نرم افزاری تولید میکند که به سبب این وقفه کنترل از کاربر به هسته منتقل می شود. و دستور ret مقدار بازگشتی از فراخوانی را بر می گرداند.
در فایل ulib.c تعریف توابع پوشاننده قرار گرفته است. و برنامه نویس میتواند از مانند توابع معمولی استفاده کند.
در واقع برنامه نویس نیاز به دانستن شماره فراخوانی ها نیست و و این که آرگومان ها باید در کدام رجیستر ذخیره بشوند. از طرفی برنامه نویس نیاز ندارد در رجیستر خاصی یه دنبال مقدار بازگشتی باشد و توابع مقدار بازگشتی را به صورت یک مقدار به آن بر می گردانند.
استفاده از این کتابخانه ها مزایایی به همراه دارد:
۱. با توجه به اینکه شماره فراخوانی سیستمی و نحوه انتقال آرگومان ها د رمعماری ها و سیتستم عامل های مختلف متفاوت است وجود این کتابخانه ها قابلیت حمل را افزایش میدهد. به این صورت که این توابع مستقل از معماری هستند بنابراین اگر برنامه نویس بخواهد کد را در سیستم عامل و معماری دیگری اجرا کند تنها کافی است که کتابخانه متناسب با آن را استفاده کند و نیازی به تغییر کد ندارد.
۲. اگر برنامه نویس به صورت مستقیم به فراخوانی های سیستمی دسترسی داشته باشد ممکن هست که آرگومان ها را در رجیستر های نادرستی ذخیره کند. یا به بخش هایی دسترسی پیدا کند که باعث آسیب پذیری های امنیتی بشه.
۳. از طرفی دیگر نوشتن کد اسمبلی برای یک فراخوانی سیستمی پیچیده و مستعد خطا هست اما کار با کتابخانه ها بسیار ساده تر هست پس باعث ساده سازی پیاده سازی میشود.
پرسش ۲:
دستور های int , sysenter/sysexit مکانیزم هایی برای انتقال کنترل از فضای کاربر به هسته هستند تا فراخوانی سیستمی انجام بشود.

در سیستم عامل 6xv وقتی یک برنامه می خواهد با هسته ارتباط برقرار کند از دستور int استفاده میکند . وقتی برنامه دستور

int $T_SYSCALL را اجرا میکند پردازنده رجیستر های مهم مثل eip, cs, eflags, esp, ss را روی استک هسته ذخیره میکنه و در ادامه به IDT نگاه میکنه وgate مربوط به T_SYSCALL را پیدا می‌کند، و به آدرس handler پرش می‌کند و سپس رجیستر های کاربر در ساختار trapframe ذخیره می‌شوند تا هسته بتواند از آن‌ها استفاده کند سپس تابع trap اجرا می شود و بررسی می کند که trapno == T_SYSCALL است. سپس تابع syscall شماره syscall را از %eax می‌خواند و handler مربوطه را اجرا می‌کند.در نهایت، دستور iret وضعیت ذخیره‌شده را بازیابی می‌کند و برنامه کاربر از جایی که متوقف شده بود ادامه می‌یابد.

این مکانیزم زمان بر است چرا که پردازنده باید چندین رجیستر را روی پشته push و pop کند که این فرآیند شامل چندین دستورالعمل است که هر کدام چند سیکل کلاک طول می‌کشد از طرفی دیگر پردازنده باید به جدول IDT نگاه کند تا آدرس handler را پیدا کند، که شامل دسترسی به حافظه است همچنین در alltraps، رجیسترهای اضافی ذخیره می‌شوند، که باز هم سیکل‌های بیشتری مصرف می‌کند در 6xv، این فرآیند معمولاً ۲۰۰ تا ۳۰۰ سیکل کلاک طول می‌کشد. برای مقایسه، هر سیکل

کلاک در یک پردازنده مدرن حدود ۰٫۳ نانوثانیه است، پس ۲۰۰ سیکل حدود ۶۰ نانوثانیه زمان می‌برد. این برای یک سیستم آموزشی مثل 6xv قابل قبول است، اما در سیستم‌های واقعی می‌تواند مشکل‌ساز باشد.

وقتی یک فراخوانی سیستمی رخ می‌دهد، هسته ممکن است داده‌های جدیدی را به cache بیاورد. وقتی کنترل به هسته منتقل می‌شود، داده‌های برنامه کاربر که در cache بودند ممکن است با داده‌های هسته جایگزین شوند که زمان‌بر است.

اما با این صورت ما همچنان از int استفاده میکنیم چرا که 6xv برای آموزش مفاهیم سیستم‌عامل طراحی شده و هدفش بهینه‌سازی عملکرد نیست. پیاده‌سازی sysenter نیاز به تنظیم MSRs و مدیریت پیچیده‌تر پشته دارد، که کد را پیچیده‌تر می‌کند همچنین int در تمام پردازنده‌های 86x کار می‌کند، اما sysenter فقط در پردازنده‌های جدیدتر پشتیبانی می‌شود از طرفی 6xv برای workloads سبک طراحی شده نه برای سناریوهای سنگین مثل سرورها. سربار ۲۰۰-۳۰۰ سیکل در این محیط قابل چشم‌پوشی است.

اگر **sysenter** در **6xv** پیاده شود سربار مستقیم می‌تواند به کمتر از ۱۰۰ سیکل کاهش یابد، چون نیازی به IDT و ذخیره کامل رجیسترها نیست اما این کار نیاز به بازنویسی بخش‌هایی از 6xv (مثل usys.S برای استفاده از sysenter به جای int) و تنظیم MSRs در بوت سیستم دارد.

در سیستم عامل لینوکس، وقتی یک برنامه می‌خواهد با هسته ارتباط برقرار کند، بسته به معماری پردازنده ، از دستورات مدرن مانند sysenter/sysexit یا syscall/sysret استفاده می‌کند. روش قدیمی 80int $0x هنوز برای سازگاری پشتیبانی می‌شود، اما کمتر استفاده می‌شود زیرا کندتر است.

وقتی برنامه دستور syscall را اجرا می‌کند پردازنده رجیسترهای مهم مانند rcx, rip، را ذخیره می‌کند و کنترل را مستقیماً به هسته منتقل می‌کند این کار بدون نیاز به جدول IDT انجام می‌شود، زیرا syscall از رجیسترهای خاص MSR استفاده می‌کند. پردازنده سطح privilege را از کاربر به ring 0 تغییر می‌دهد و به handler ورود syscall پرش می‌کند.

در هسته، وضعیت کاربر در ساختار pt_regs ذخیره می‌شود تا هسته بتواند آرگومان‌ها را بخواند. سپس، هسته شماره syscall را از رجیستر %rax می‌خواند و بررسی‌های امنیتی انجام می‌دهد. بعد از آن، از جدول sys_call_tableو handler مربوطه را پیدا کرده و اجرا می‌کند. در نهایت، دستور sysret وضعیت ذخیره‌شده را بازیابی می‌کند و برنامه کاربر از دستور بعدی ادامه می‌یابد. این مکانیزم از vDSOct برای برخی syscallهای ساده استفاده می‌کند تا بدون ورود کامل به هسته اجرا شوند.

این مکانیزم نسبتاً سریع است، اما همچنان سربار دارد. سربار مستقیم ناشی از تغییر حالت privilege، ذخیره و بازیابی رجیسترها، و بررسی‌های امنیتی هسته است. این فرآیند شامل دستورالعمل‌هایی است که هر کدام چند سیکل کلاک طول می‌کشد، به علاوه دسترسی به MSRها و جدول syscall. در لینوکس مدرن روی 86x-64، این سربار معمولاً ۱۰۰ تا ۳۰۰ سیکل کلاک طول می‌کشد. برای مقایسه، هر سیکل کلاک در یک پردازنده مدرن حدود ۰٫۲۵-۰٫۳ نانوثانیه است، پس ۲۰۰ سیکل حدود ۵۰-۶۰ نانوثانیه زمان می‌برد. این سربار برای سیستم‌های واقعی مثل سرورها قابل مدیریت است، اما در workloads سنگین می‌تواند bottleneck باشد.

وقتی یک فراخوانی سیستمی رخ می‌دهد، هسته ممکن است داده‌های جدیدی را به cache بیاورد. وقتی کنترل به هسته منتقل می‌شود، داده‌های برنامه کاربر که در cache بودند ممکن است با داده‌های هسته جایگزین شوند، که باعث آلودگی cache می‌شود. همچنین، TLB ممکن است flush شود یا branch prediction پردازنده مختل شود، که دسترسی بعدی به داده‌ها را کندتر می‌کند . لینوکس با تکنیک‌هایی مثل vDSO و KPTI این سربارها را کاهش می‌دهد، اما همچنان وجود دارد.

با این حال، لینوکس از syscall/sysret استفاده می‌کند چرا که سریع‌تر از روش قدیمی int است و برای عملکرد بالا در سیستم‌های واقعی طراحی شده. پیاده‌سازی syscall نیاز به تنظیم MSRها در زمان بوت هسته دارد، اما این کار کد را پیچیده‌تر نمی‌کند زیرا لینوکس از ابتدا برای بهینه‌سازی عملکرد ساخته شده. همچنین، syscall در تمام پردازنده‌های مدرن 86x-64 پشتیبانی می‌شود و سازگاری بالایی دارد. لینوکس برای workloads سنگین طراحی شده، پس سربار ۱۰۰-۳۰۰ سیکل قابل قبول است و با mitigationهای امنیتی تعدیل می‌شود.

اگر int $0x80 در لینوکس استفاده شود، سربار مستقیم می‌تواند به ۲۰۰-۵۰۰ سیکل افزایش یابد، چون نیاز به IDT و ذخیره کامل‌تر وضعیت دارد. اما لینوکس به سمت syscall/sysret رفته تا سربار را کاهش دهد، و این کار نیاز به بازنویسی wrapperهای glibc و تنظیم MSRها در هسته دارد. در نسخه‌های مدرن لینوکس ، حتی برای syscallهای ساده، از تکنیک‌های اضافی مثل user-space mapping برای کاهش ورود به هسته استفاده می‌شود.

پرسش ۳:

در معماری xv6 برای اجرای فراخوانی‌های سیستمی از مکانیزم قدیمی پردازنده‌های x86 و به‌طور مشخص از Descriptor Gateها استفاده می‌شود. هر ورود کنترل‌شده به هسته، چه در قالب فراخوانی سیستمی و چه در قالب وقفه‌های سخت‌افزاری و استثناها، از طریق یک Gate در جدول توصیفگر وقفه‌ها (IDT) انجام می‌گیرد. نکته‌ی مهم این است که سطح دسترسی (DPL) تعریف‌شده برای هر Gate تعیین می‌کند که کدام سطح از کد (کاربر یا هسته) مجاز است به‌صورت نرم‌افزاری و عمدی آن Gate را فعال کند. در این میان، تنها فراخوانی سیستمی است که در xv6 با استفاده از Trap Gate و با سطح دسترسی USER_DPL مقداردهی می‌شود، در حالی که سایر تله‌ها مانند وقفه‌های سخت‌افزاری و اغلب استثناها، چنین سطح دسترسی‌ای را در اختیار ندارند.

فراخوانی سیستمی تنها مسیر رسمی و کنترل‌شده‌ای است که از طریق آن کد سطح کاربر (با CPL=3) می‌تواند آگاهانه وارد هسته شود. این کار معمولاً با اجرای دستور نرم‌افزاری `int` با شماره تله‌ی مربوط به SYSCALL انجام می‌شود. مطابق قواعد سخت‌افزاری x86، اجرای دستور `int n` از سطح کاربر تنها زمانی مجاز است که سطح دسترسی Gate متناظر در IDT، یعنی DPL، بزرگ‌تر یا مساوی سطح فعلی اجرای برنامه (CPL) باشد. بنابراین اگر قرار است برنامه‌ی کاربر بتواند Gate مربوط به فراخوانی سیستمی را فعال کند، باید DPL آن Gate برابر USER_DPL (یعنی سطح ۳) تنظیم شود. این تنظیم باعث می‌شود که تنها از طریق همین تله‌ی مشخص، انتقال از مد کاربر به مد هسته صورت گیرد و هسته بتواند به‌صورت متمرکز بر روی این نقطه‌ی ورود، کنترل‌های لازم (مانند بررسی شماره‌ی فراخوانی سیستمی، آرگومان‌ها، و محدودیت‌های امنیتی) را اعمال کند.

در مقابل، سایر تله‌ها از جمله وقفه‌های سخت‌افزاری و بسیاری از استثناها، برای این طراحی نشده‌اند که مستقیماً و به‌صورت عمدی از طرف برنامه‌ی کاربر فراخوانی شوند. وقفه‌های سخت‌افزاری توسط کنترلر وقفه یا سخت‌افزار تولید می‌شوند و استثناهایی مثل تقسیم بر صفر یا خطای دسترسی به حافظه نیز مستقیماً توسط خود پردازنده و در پاسخ به یک خطای واقعی در اجرا ایجاد می‌گردند، نه با دستور `int` از طرف برنامه‌ی کاربر. اگر برای این Gateها نیز سطح دسترسی USER_DPL در نظر گرفته شود، برنامه‌ی کاربر می‌تواند با اجرای دستورات `int` مختلف، مستقیماً وارد روتین‌های وقفه‌ها و استثناهای هسته شود؛ برای مثال، بدون وقوع واقعی آن وقفه یا استثنا، به کد رسیدگی به وقفه‌ی تایمر یا سایر وقفه‌ها بپرد. این موضوع عملاً به معنای فراهم شدن یک مسیر ورود غیرمجاز و کنترل‌نشده به بخش‌هایی از کد هسته است که برای ورود از سطح کاربر طراحی نشده‌اند و می‌تواند باعث دور زدن مکانیزم فراخوانی سیستمی، ایجاد رخنه‌های امنیتی، و حتی از کار انداختن سیستم شود. به همین دلیل، این Gateها معمولاً با DPL=0 (سطح هسته) مقداردهی می‌شوند تا فقط کدی که خودش در مد هسته اجرا می‌شود بتواند آن‌ها را به‌صورت نرم‌افزاری فعال کند و در عین حال، سخت‌افزار و خود CPU همچنان بتوانند در صورت نیاز آن‌ها را بدون توجه به DPL فراخوانی کنند.

در نتیجه، تنها تله‌ای که در xv6 با سطح دسترسی USER_DPL فعال می‌شود، تله‌ی مربوط به فراخوانی سیستمی است که با Trap Gate پیاده‌سازی شده است. این طراحی باعث می‌شود یک نقطه‌ی ورود مشخص، امن و تحت‌کنترل برای انتقال از مد کاربر به مد هسته وجود داشته باشد، در حالی که سایر تله‌ها (وقفه‌های سخت‌افزاری و استثناها) صرفاً توسط سخت‌افزار یا خود هسته فعال می‌شوند و برنامه‌ی کاربر امکان استفاده مستقیم از آن‌ها را ندارد. به این ترتیب، جداسازی سطوح کاربر و هسته حفظ شده و امنیت سیستم عامل تضمین می‌گردد.

پرسش ۴:

در هنگام اجرای دستور `int` در معماری x86، پردازنده باید یک قاب تله (Trap Frame) روی پشته ایجاد کند تا پس از اتمام رسیدگی به تله، بتواند با دستور `iret` دقیقاً به همان نقطه‌ی قبلی اجرا بازگردد. نحوه‌ی تشکیل این قاب تله به این بستگی دارد که آیا با ورود به تله، سطح دسترسی پردازنده (Privilege Level) تغییر می‌کند یا خیر. در صورتی که اجرای تله موجب تغییر سطح دسترسی شود، مانند حالتی که یک فراخوانی سیستمی از مد کاربر (3 ring) به مد هسته (0 ring) انجام می‌گیرد، پردازنده ناچار

است علاوه بر پرچم‌ها و ثبات‌های مربوط به کد، اطلاعات مربوط به پشته‌ی قبلی را نیز ذخیره کند. در این حالت، ابتدا پردازنده با توجه به اطلاعات موجود در TSS به پشته‌ی هسته سوییچ کرده و سپس مقادیر ثبات‌های SS و ESP مربوط به سطح قبلی (مثلاً سطح کاربر) را روی پشته‌ی جدید (پشته‌ی هسته) Push می‌کند. این کار به این دلیل انجام می‌شود که پس از پایان رسیدگی به تله، سیستم بتواند دقیقاً به همان پشته‌ی قبلی در سطح کاربر بازگردد و اجرای برنامه بدون اشکال ادامه یابد. در واقع، چون در این حالت سوییچ واقعی بین دو پشته‌ی مجزا (پشته‌ی کاربر و پشته‌ی هسته) رخ می‌دهد، ذخیره‌ی آدرس کامل پشته‌ی قبلی، یعنی ترکیب SS:ESP، برای امکان بازگردانی صحیح و امن حالت قبل ضروری است.

در مقابل، اگر تله‌ای در همان سطح دسترسی جاری رخ دهد و تغییری در سطح دسترسی صورت نگیرد، مانند وقوع یک وقفه یا استثنا در حالی که پردازنده از قبل در مد هسته (0 ring) قرار دارد، دیگر نیازی به ذخیره‌سازی ثبات‌های SS و ESP وجود ندارد. در این وضعیت، پردازنده از همان پشته و همان سگمنت پشته‌ی فعلی استفاده می‌کند و عملیات Push صرفاً روی همین پشته ادامه می‌یابد. بنابراین، تنها مقادیری مانند EIP، CS، EFLAGS و در صورت وجود، کد خطا روی پشته قرار می‌گیرند و قاب تله بر همین اساس تشکیل می‌شود. از آن‌جا که در این حالت، سگمنت پشته و اشاره‌گر پشته تغییر نکرده‌اند، ذخیره‌ی جداگانه‌ی SS و ESP ضرورتی ندارد و بازگشت با دستور iret بدون نیاز به برگرداندن این دو ثبات نیز به‌درستی انجام می‌پذیرد. به بیان دیگر، در نبود تغییر سطح دسترسی، پشته‌ی قبل و بعد از تله یکی است و پردازنده نیازی به نگه‌داشتن اطلاعات اضافه برای سوییچ بین دو پشته‌ی متفاوت ندارد.

در نتیجه، می‌توان گفت که Push شدن یا نشدن SS و ESP روی پشته کاملاً به تغییر سطح دسترسی وابسته است. اگر با ورود به تله، سطح دسترسی پردازنده تغییر کند، برای امکان بازگشت دقیق به سطح قبلی، ثبات‌های SS و ESP نیز روی پشته ذخیره می‌شوند؛ اما اگر سطح دسترسی ثابت بماند، نیازی به ذخیره‌ی این دو ثبات نیست و پردازنده تنها با ذخیره‌ی مقادیر لازم برای ادامه و بازگشت اجرای برنامه (مانند EIP، CS و EFLAGS) قاب تله را ایجاد می‌کند. این رفتار موجب کارایی بیشتر و در عین حال حفظ درستی و امنیت در مکانیزم مدیریت تله‌ها در پردازنده می‌گردد.

پرسش ۵:
در فرآیند اجرای فراخوانی‌های سیستمی در xv6، پس از وقوع تله‌ی فراخوانی سیستمی و تشکیل قاب تله بر روی پشته، کنترل به تابع سطح بالای C یعنی trap و سپس به تابع syscall منتقل می‌شود. در این مرحله، تمام وضعیت برنامه‌ی کاربر از جمله شماره‌ی فراخوانی سیستمی و پارامترهای آن روی پشته یا در ثبات‌ها موجود است، اما این اطلاعات به‌صورت خام و در قالب مقادیر عددی و آدرس‌ها قرار دارند و باید به شکلی ایمن و ساخت‌یافته بازیابی شوند. توابعی مانند argint و argptr در اینجا نقش کلیدی در استخراج این پارامترها از پشته و تبدیل آن‌ها به مقادیری قابل استفاده در هسته ایفا می‌کنند.

تابع argint برای بازیابی آرگومان‌هایی استفاده می‌شود که از نوع عدد صحیح (مثلاً اندازه‌ی بافر، شماره‌ی فایل، شماره‌ی فریم و غیره) هستند. این تابع با توجه به ترتیب قراردادن پارامترها روی پشته در کد سطح کاربر، و با استفاده از اطلاعات موجود در قاب تله (مانند مقدار esp ذخیره‌شده)، موقعیت دقیق پارامتر مورد نظر را روی پشته پیدا کرده و مقدار آن را به‌صورت یک عدد صحیح در اختیار کد هسته قرار می‌دهد. به این ترتیب، هسته بدون دسترسی مستقیم و کور به پشته‌ی کاربر، از طریق یک واسط تعریف‌شده و مشخص، مقدار پارامترهای عددی را دریافت می‌کند.

در مقابل، تابع argptr علاوه بر بازیابی مقدار عددی آرگومان، آن را به‌عنوان یک آدرس (Pointer) در نظر می‌گیرد و باید اطمینان حاصل کند که این آدرس و بازه‌ی حافظه‌ی متناظر با آن، در محدوده‌ی مجاز فضای آدرس‌دهی فرایند کاربر قرار دارد. به عبارت دیگر، argptr نه تنها مقدار آدرس را از روی پشته می‌خواند، بلکه بررسی می‌کند که:

- آدرس شروع بافر داخل فضای آدرس‌دهی معتبر فرایند کاربر باشد،
- جمع آدرس شروع و اندازه‌ی داده (طول بافر) از انتهای حافظه‌ی مجاز فرایند (sz) فراتر نرود،

- بافر روی ناحیه‌ای قرار نگیرد که متعلق به هسته یا خارج از فضای مجاز کاربر است.

علت ضرورت این بررسی آن است که هسته در فراخوانی‌های سیستمی‌ای مانند `sys_read`، بر اساس این آدرس‌ها داده را به حافظه‌ای که کاربر مشخص کرده کپی می‌کند. در `sys_read`، کاربر سه پارامتر مهم ارسال می‌کند: شماره‌ی فایل، آدرس بافر، و طول بافر. اگر تابع `argptr` بازه‌ی آدرس ورودی را کنترل نکند، برنامه‌ی کاربر (اعم از خرابکار یا دارای باگ) می‌تواند:

- آدرسی خارج از فضای مجاز خود، یا حتی آدرسی در محدوده‌ی حافظه‌ی هسته، به‌عنوان بافر مقصد معرفی کند؛
- طول بافر را به‌گونه‌ای انتخاب کند که جمع آدرس و طول، از مرز حافظه‌ی مجاز عبور کرده و باعث دسترسی خارج از محدوده شود.

در چنین شرایطی، اگر هسته بدون بررسی، داده‌ای را در آن آدرس بنویسد، ممکن است:

- بخشی از حافظه‌ی هسته بازنویسی شود و منجر به خرابی هسته (کرش) یا رفتار غیرقابل پیش‌بینی گردد؛
- داده‌های حساس هسته یا فرایندهای دیگر افشا یا دستکاری شوند؛
- یک برنامه‌ی خرابکار با طراحی دقیق آدرس و طول، از این رفتار برای ایجاد آسیب‌پذیری امنیتی و ارتقای سطح دسترسی استفاده کند.

به‌طور خاص، در `sys_read` اگر بافر اشاره‌گر به ناحیه‌ای خارج از فضای کاربر باشد، و هسته بدون چک کردن، داده را در آن محل کپی کند، این امکان وجود دارد که حافظه‌ی هسته یا حافظه‌ی فرایند دیگری ناخواسته تغییر کند. این وضعیت می‌تواند هم باعث اختلال در عملکرد سیستم (مثلاً کرش ناگهانی) شود و هم زمینه‌ساز حملاتی باشد که از طریق آن کد کاربر کنترل بخش‌هایی از حافظه‌ی حساس را به‌دست می‌آورد.

از این رو، بررسی بازه‌ی آدرس در تابع `argptr` نه یک کار اضافی، بلکه جزء جدایی‌ناپذیر طراحی امن فراخوانی‌های سیستمی است. حذف این چک‌ها از نظر عملی و امنیتی قابل قبول نیست؛ زیرا سیستم‌عامل نمی‌تواند به صحت و حسن‌نیت برنامه‌های کاربر اعتماد کند و همواره این احتمال وجود دارد که پارامترهای اشتباه (چه به‌صورت ناخواسته و چه عمدی) به هسته ارسال شوند. بنابراین، هم امکان وارد کردن بازه‌ی غلط توسط برنامه‌ی کاربر همیشه وجود دارد، و هم هسته موظف است با استفاده از توابعی مانند **`argptr`** این خطاها یا حملات احتمالی را قبل از دسترسی به حافظه تشخیص و متوقف کند.

در نتیجه، نقش اصلی `argint` و `argptr` در بازیابی پارامترهای فراخوانی سیستمی، ایجاد یک لایه‌ی واسط امن بین فضای آدرس‌دهی کاربر و هسته است. `argint` مقادیر عددی را به‌صورت ساخت‌یافته و منظم استخراج می‌کند و `argptr` با کنترل بازه‌ی آدرس، تضمین می‌نماید که هسته فقط به حافظه‌ی مجاز و متعلق به همان فرایند کاربر دسترسی پیدا کند. بدون این بررسی‌ها، فراخوانی‌هایی مانند `sys_read` می‌توانند به نقطه‌ی شروعی برای اختلال، کرش یا آسیب‌پذیری‌های جدی امنیتی تبدیل شوند و جداسازی بین فضای کاربر و هسته عملاً نقض گردد.

پرسش ۶:
طبق ABI لینوکس در معماری x86، پارامترهای فراخوانی سیستمی در ثبات‌هایی مانند `edi`، `esi`، `edx`، `ecx`، `ebx` و `ebp` قرار می‌گیرند. در عین حال، برخی از این ثبات‌ها (مانند `edi`، `esi`، `ebx` و `ebp`) جزء ثبات‌های موسوم به **callee-saved** هستند؛ یعنی طبق قرارداد ABI، بعد از بازگشت از فراخوانی (اعم از فراخوانی تابع معمولی یا تابع پوشاننده‌ی فراخوانی سیستمی)،

مقدار آن‌ها نباید از دید برنامه‌ی سطح بالا تغییر کرده باشد. به همین دلیل، در کدهای glibc قبل از انجام فراخوانی سیستمی، مقادیر این ثبات‌ها در جایی (معمولاً روی پشته) ذخیره شده و پس از انجام syscall دوباره بازیابی می‌شوند تا قرارداد ABI نقض نشود و برنامه بتواند به اجرای عادی خود ادامه دهد.

اگر برنامه‌نویس به‌صورت دستی و خارج از چارچوب معمول (برای مثال با نوشتن اسمبلی درون‌خطی بدون رعایت قواعد) مقادیر این ثبات‌ها را تغییر دهد، چند دسته مشکل به‌وجود می‌آید. از یک سو، پارامترهای فراخوانی سیستمی ممکن است دیگر مطابق انتظار نباشند؛ یعنی کرنل هنگام اجرای syscall مقادیر اشتباه را به‌عنوان شماره‌ی فایل، آدرس بافر، طول داده، یا سایر پارامترها دریافت می‌کند. نتیجه‌ی این وضعیت می‌تواند بروز خطاهایی مانند بازگشت کدهای خطا، دسترسی نامعتبر به حافظه، و رفتارهای غیرقابل پیش‌بینی در سطح سیستم‌عامل باشد. برای مثال، اگر به‌صورت نادرست آدرس بافر یا طول آن در ثبات‌های مربوطه تغییر داده شود، ممکن است فراخوانی سیستمی `read` یا `write` با آدرس یا طولی اجرا شود که با چیزی که برنامه‌ی کاربر در سطح C انتظار دارد سازگار نیست؛ در نتیجه داده در محل نامناسبی نوشته شده یا خطاهای جدی در اجرا رخ می‌دهد.

از سوی دیگر، تغییر دستی این ثبات‌ها می‌تواند قرارداد **ABI** بین کامپایلر، کتابخانه‌ی استاندارد و برنامه را نقض کند. کامپایلر ممکن است متغیرهای محلی، اشاره‌گرهای مهم یا حتی آدرس فریم فعلی تابع را در ثبات‌هایی مانند `edi`، `esi`، `ebx` یا `ebp` نگه دارد و فرض کند که این ثبات‌ها طبق قرارداد تا پایان تابع دست‌نخورده باقی می‌مانند. اگر برنامه‌نویس در اسمبلی، بدون اطلاع دادن به کامپایلر و بدون ذخیره و بازیابی مقادیر قبلی، این ثبات‌ها را تغییر دهد، متغیرهای داخلی برنامه و ساختار پشته دچار خرابی می‌شوند و این موضوع می‌تواند منجر به کرش برنامه، دسترسی به حافظه‌ی نامعتبر، نتایج محاسباتی غلط یا حتی آسیب‌پذیری‌های امنیتی شود. به بیان دیگر، سخت‌افزار جلوی تغییر این ثبات‌ها را نمی‌گیرد، اما نتیجه‌ی چنین تغییراتی نقض قرارداد **ABI** و بروز رفتار نامعین (**undefined behavior**) در سطح برنامه است که از دید طراحی سیستم ناامن و نامطلوب محسوب می‌شود.

بنابراین، اگرچه از لحاظ فنی برنامه‌ی کاربر می‌تواند رجیسترها را تغییر دهد، این کار تنها زمانی مجاز و قابل اتکا است که یا در چهارچوب ABI و قراردادهای کامپایلر انجام شود (مثلاً با استفاده از توابع پوشاننده‌ی استاندارد، یا اسمبلی درون‌خطی همراه با اعلان درست ثبات‌های تغییریافته برای کامپایلر)، یا این‌که برنامه‌نویس به‌طور کامل مسئولیت پیامدهای نقض ABI را بپذیرد. در غیر این صورت، تغییرات دستی در رجیسترهای مذکور می‌تواند باعث ارسال پارامترهای نادرست به فراخوانی‌های سیستمی، خرابی وضعیت داخلی برنامه و ایجاد اختلال یا آسیب‌پذیری در سیستم گردد.