آزمایشگاه سیستم عامل

پروژه شماره 2

علی پادیاو – کسری حاجی حیدری – اولدوز نیساری

بهار 1402

Repository Link: https://github.com/alumpish/OS-Lab-Projects

Latest Commit Hash: 609bc6b3ca3756d057c115085606bcf7a3a5464a

مقدمه

-1

اگر در فایل‌های تشکیل دهنده متغیر ULIB دقت کنیم متوجه میشویم از فایل‌های زیر تشکیل شده است: ulib.o, usys.o, printf.o, umalloc.o که به تشریح آن‌ها می‌پردازیم:

:ulib

در ulib.c در تعدادی از توابع مثل atoi, strcpy,strlen,strcmp از فراخوانی سیستمی استفاده نشده است، اما در تعدادی دیگر از توابع استفاده شده است:

در stat از فراخوانی‌های سیستمی open , close برای باز و بسته کردن فایل‌ها استفاده می‌شود، هم چنین از فراخوانی سیستمی fsat برای به دست آوردن اطلاعات آن فایل استفاده می‌شود.

در gets از فراخوانی سیستمی read برای گرفتن ورودی استفاده شده است.

در memset از فراخوانی سیستمی stosb برای پر کردن حافظه استفاده شده است.

:usys

در این فایل اسم سیستم کال‌ها به صورت گلوبال نوشته شده است، آیدی‌های آن‌ها نیز در رجیستر eax نوشته شده است. در زمان فراخوانی هر یک از سیستم کال‌ها به define ای که در این فایل وجود دارد مراجعه می‌شود:

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

در این فایل همچنین لیستی از سیستم کال‌ها موجود است.

:printf

در تابع putc از فراخوانی سیستمی write برای نوشتن یک حرف کاراکتر در یک fd استفاده می‌شود.

:umalloc

در تابع morecore از فراخوانی سیستمی sbrk برای تغییر اندازه دیتا سگمنت و افزایش حافظه استفاده می‌شود.

-2

روش‌های دیگر دسترسی سطح کاربر به هسته لینوکس:

:Pseudo-file-systems

در حقیقت فایل‌هایی هستند که فایل‌های حقیقی ندارند بلکه تعدادی ورودی مجازی دارند که خود سیستم فایل آن‌ها را نگه می‌دارد. چون در حقیقیت یک رابط به هسته به ما می‌دهند نیاز دارند که به کرنل دسترسی داشته باشند.

:Exceptionها

به طور کلی برای رفع خطاهایی که در interruptهای سخت افزاری یا نرم افزاری به وجود می‌آید دسترسی به کرنل انجام شود تا خطاها بر طرف شوند.

(exceptionها نوعی از interruptهای نرم افزاری است)

:Socket

برنامه‌های سطح کاربر که بر مبنای سوکت نوشته می‌شوند با قابلیت اتصال به سوکت و جا به جا کردن داده‌ها می‌توانند به کرنل دسترسی پیدا کنند.

-3

خیر. اگر باقی تله‌ها با همین حالت فعال شوند، در آن صورت دسترسی به هسته بسیار آسان می‌شود و امنیت دیگر تضمین نیست. به همین خاطر اگر کاربر سعی کند تله ای دیگر را فعال کند، xv6 جلوگیری می‌کند تا امنیت به خطر نیفتند و امکان سو استفاده و آسیب از بین برود.

سازوکار اجرای فراخوانی سیستمی در xv6

بخش سخت افزاری واسمبلی

-4

هنگامی که یک تله فعال میشود، سطح دسترسی از سطح دسترسی کاربر به هسته تغییر میکند و از پشته ی مختص به هسته استفاده میکند.(هر کدام از این دو حالت پشته مختص به خود را دارند) بعد از اینکه رسیدگی به تله به اتمام رسید باید به پشته کاربر بازگردیم. بدین منظور از دو رجسیتر esp و ss استفاده میکنیم که به پشته در حال استفاده اشاره میکند.برای اینکار زمانی که تله فعال میشود مقادیر این دو رجیستر که به پشته کاربر اشاره دارند را در پشته هسته ذخیره میکنیم و بعد از اتمام رسیدگی به تله مقادیر آن را بازیابی کرده و به پشته کاربر بازمیگردیم و برنامه کاربر از جای قبلی ادامه خواهد یافت.در صورتی که نیاز به تغییر سطح کاربر نباشد.

از آن جایی که همچنان با پشته قبلی کار میکنیم نیازی به ذخیره مقادیر از دو رجیستر نیست.

بخش سطح بالا وکنترل کننده زبان سی تله

-5

چهارتابع برای دسترسی به پارامترهای فراخوانی سیستمی تعریف شده که عبارتند از (argint,argptr,argstr,argfd) که در ادامه هر یک توضیح داده شده است:

argint: این تابع از رجیستر esp استفاده میکند تا n امین پارامتر را پیدا کند. این رجیستر به انتهای پشته و آدرس برگشت اشاره میکند و پارانتر های فراخوانی سیستمی بالای آن قرار دارند پس با فرمول زیر میتوان به nامین پارامتر دست یافت: esp + 4 + 4 * n

پس از دست‌یابی به این پارامتر، آن را به همراه پوینتری به حافظه (ip*) به تابع fetchint ارسال میشود و در صورت تایید ادرس فرستاده شده(در حافظه پردازه باشد) آن را در حافظه داده شده ذخیره می‌کند.

argptr: این تابع ابتدا با استفاده از تابع argint آدرس پوینتر مورد نظر را چک میکند و در صورتی که معتبر باشد آن را دخیره میکند.

argstr: با استفاده از تابع argint ابتدای رشته را مشخص میکند و به تابع fetchstr میدهد و این تابع بررسی میکند که این آدرس در حافظه پردازه باشد و پس از تایید آن را ذخیره می‌کند.

argfd: با استفاده از تابع argint عدد filedescriptor را میگیرد و آن را چک میکند و اگر صحیح بود فایل متناظر با آن را بازمیگرداند.

تمامی این توابع بررسی می کنند که آدرس داده شده حتما در حافظه پردازه قرار گیرد که یک پردازه نتواند به حافظه پردازه دیگری دسترسی پیدا کند و از مشکلات امنیتی جلوگیری میکند، در نهایت در صورت وجود مشکل 1- را برمیگرداند.

تابع sys_read به صورت زیر است که مربوط به تابع read() میباشد:

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

```
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

در ابتدا این تابع با استفاده از تابع argfd مقدار fd را دریافت میکند و درستی آن را بررسی میکند. در ادامه با استفاده از تابع argint پارامتر سوم را دریافت و در نهایت با استفاده از تابع argptr پارامتر دوم را دریافت و بررسی میکند که بافر داده شده از ابتدا تا اندازه ی max از حافظه ی پردازه باشد.
اگر این بررسی ها صورت نمیگرفت ممکن بود به علت عدم تطابق بافر و مقدار max هنگام خواندن و نوشتن در بافر ممکن بود از حافظه پردازه خارج شویم و به مشکل بربخوریم.

بررسی گام‌های فراخوانی سیستمی در سطح کرنل توسط GDB:

ابتدا بر روی خط 131 که ابتدای تابع syscall در فایل syscall.c است بریک پوینت قرار می‌دهیم. سپس اجرای برنامه را ادامه میدهیم تا روند اجرا در برخورد با بریک پوینت متوقف شود. در آن زمان از دستور bt استفاده می‌کنیم تا سابقه فراخوانی‌ها را ببینیم.

```
(gdb) b syscall.c:131
Breakpoint 1 at 0x80105160: file syscall.c, line 133.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:135
#1  0x801061ad in trap (tf=0x8dffffb4) at trap.c:43
#2  0x80105f4f in alltraps () at trapasm.S:20
```

دستور bt مخفف backtrace است و زمانی که میخواهیم متوجه شویم که چه توابعی فراخوانی شده اند و به استک اضافه شده اند از آن استفاده می‌کنیم. این دستور فراخوانی‌ها را به ترتیب از درونی ترین فریم شروع می‌کند.

در خصوص توضیحات خروجی مان اتفاقی که می‌افتد به این صورت است که مطابق چیزی که در فایل usys.h که ابتدا شماره سیستم کال بر طبق چیزی که در syscall.h به آن اختصاص داده شده است در رجیستر eax نوشته می‌شود و سپس `int $T_SYSCALL;` اجرا می‌شود. برای جستجو تعریف این خط وارد فایل vector.s میشود و در آنجا تابع alltraps که در trapasm.s است فراخوانی می‌شود و درون آن تابع هم traps که در trap.c است فراخوانی می‌شود.

همان طور که می‌بینیم این روند فراخوانی از درونی ترین تا بیرونی ترین در عکس خروجی ما قابل دیدن است.

```
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal
```

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

حال دستور down را که کار آن جا به کردن به سمت پایین استک است را اجرا می‌کنیم.

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

همان طور که مشاهده می‌کنیم چون در درونی ترین فریم هستیم، دستور اجرا نمی‌شود.

با قرار دادن یه بریک پوینت روی خط 138 مقدار eax را به دست می‌آوریم:

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$1 = 7
```

که همان طور که مشاهده می‌کنیم با مقدار get_pid که 16 است برابر نیست. دلیل هم آن است که پیش از فراخوانی get_pid فراخوانی‌های دیگری صدا زده می‌شوند. در نهایت هم با چند بار continue کردن مقدار رجیستر را می‌بینیم.

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$2 = 15
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$3 = 10
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$4 = 10
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$5 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$6 = 16
(gdb)
```

ارسال آرگومان‌های فراخوانی‌های سیستمی

برای اضافه کردن این فراخوانی سیستمی، در ابتدا تابع در دسترس کاربر را در user.h دیکلر می کنیم:

```
int find_fibonacci_number(void)
```

سپس تعریف این تابع را در usys.s انجام می دهیم:

```
SYSCALL(find_fibonacci_number)
```

حال باید عدد سیستم را در syscall.h اضافه کنیم:

```
#define SYS_find_fibonacci_number 26
```

حال در syscall.c دیکلر تابع را قرار میدهیم و سپس آن را به آرایه مپ شماره سیستم کال به تابع اضافه می کنیم:

```
extern int sys_find_fibonacci_number(void)
[SYS_find_fibonacci_number] sys_find_fibonacci_number
```

تعریف تابع را میتوان در یکی از فایل های sysproc.c یا sysfile.c قرار داد اما از آنجا که این تابع از نظر عملکردی ربطی به این دو فایل ندارد فایل جدیدی میسازیم و فایل تابع را داخل آن قرار میدهیم. (sysutils.c)

```c
static int find_fibonacci_number(int n) {
    if (n <= 0)
        return -1;

    if (n == 1)
        return 0;

    if (n == 2)
        return 1;

    return find_fibonacci_number(n - 1) + find_fibonacci_number(n - 2);
}

int
sys_find_fibonacci_number(void) {
    return find_fibonacci_number(myproc()->tf->ebx);
}
```

از آنجا که فایل جدیدی ساختیم باید آن را به متغیر OBJS اول Makefile اضافه کنیم .برای اجرای این فراخوانی سیستمی، یک برنامه سطح کاربر می سازیم. فایل find_fibonacci_number.c را ساخته و _find_fibonacci_number را به متغیر UPROGS در Makefile اضافه می کنیم.

باید به صورت دستی آرگومان که عدد مورد نظر است را به رجیستر ebx بریزیم .برای این کار در ابتدا، مقدار کنونی رجیستر ebx را در متغیری ذخیره کرده و مقدار آرگومان را در آن میریزیم. سپس سیستم کال را انجام می دهیم و مقدار رجیستر را به حالت قبلی اش بر می گردانیم.

```c
int ffn_syscall(int num) {
    int prev_ebx;

    // Save ebx in prev_ebx to restore later.
    // Move num to ebx.
    asm volatile(
        "movl %%ebx, %0\n\t"
        "movl %1, %%ebx"
        : "=r"(prev_ebx)
        : "r"(num)
    );

    int result = find_fibonacci_number();

    // Restore ebx.
    asm volatile(
        "movl %0, %%ebx"
        :: "r"(prev_ebx)
    );

    return result;
}
```

خروجی به صورت زیر است.

```
$ find_fibonacci_number 5
3
$ find_fibonacci_number 6
5
$
```

پیاده‌سازی فراخوانی‌های سیستمی

نحوه اضافه کردن systemcall ها مثل سوال قبل است. بنابراین از توضیحات تکراری صرف نظر می‌کنیم.

1- `int find_most_callee(void)`

در فایل `syscall.c` آرایه‌ای برای شمارش سیستم‌کال‌ها اضافه کردیم و هربار که سیستم‌کالی فراخوانی می‌شود، این آرایه را آپدیت می‌کنیم.

```
int syscalls_count[25];

void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    syscalls_count[num]++;
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}

```

در فایل `sysproc.c` هم به صورت زیر عمل خواسته شده را انجام می‌دهیم.

```
extern int syscalls_count[25]; // assuming there are 25 system calls in xv6

int
sys_find_most_callee(void)
{
  int i, max_calls = 0, most_called_syscall = 0;
  for (i = 0; i < 25; i++) {
    if (syscalls_count[i] > max_calls) {
      max_calls = syscalls_count[i];
      most_called_syscall = i;
    }
  }
  return most_called_syscall;
}
```

برنامه سطح کاربر `find_most_calle.c` هم نوشتیم تا سیستم‌کال را تست کنیم.

```c
int main(int argc, char *argv[])
{
    int pid = find_most_callee();
    printf(1, "Most called systemcall is: %d\n", pid);
    exit();
}
```

خروجی به صورت زیر است.

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group Members:
1- Ouldouz Neysari
2- Ali Padyav
3- Kasra Haji Heydari
$ find_most_calle
Most called systemcall is: 16
$ find_most_calle
Most called systemcall is: 16
$ find_most_calle
Most called systemcall is: 16
$
```

سیستم کال شماره 16 برای `write` است. علت این نتیجه این است که برای پرینت هر حرف در کنسول یکبار این سیستم‌کال فراخوانی میشود، برای همین در شروع کار بیشتر از همه استفاده شده است.

2- `int get_children_count(void)`

به استراکت `proc` در `proc.h` فیلد `children_count` را اضافه میکنیم.

```c
  int children_count;           // Count of childern
```

در فایل `proc.c` و در تابع `allocproc` که یک پردازه را میسازد مقدار `children_count` را برابر `0` قرار میدهیم.

```c
  p->children_count = 0
```

در فایل sysproc.c هم به صورت زیر عمل خواسته شده را انجام می‌دهیم.

```
int
sys_get_children_count(void)
{
  return myproc()->children_count;
}
```

برنامه سطح کاربر get_children_count.c را نوشته و سیستم‌کال را تست می‌کنیم.

```
int main(void)
{
    int pid;
    int i;

    for (i = 0; i < 3; i++)
    {
        pid = fork();
        if (pid == 0)
        {
            // child process
            sleep(50);
            exit();
        }
    }

    printf(1, "Parent process (pid %d) has %d children.\n", getpid(), get_children_count());

    for (int i = 0; i < 3; ++i)
        wait();

    exit();
}
```

خروجی به صورت زیر است.

```
$ get_children_count
Parent process (pid 9) has 3 children.
$
```

3- int kill_first_child(void)

ابتدا به استراکت proc فیلد فرزندان پردازه را اضافه می‌کنیم.

```
struct proc *children[NPROC];   // array of child processes
```

در فایل sysproc.c تابع kfc که در proc.c هست را کال می‌کنیم.

```
int
sys_kill_first_child(void)
{
  return kfc();
}
```

در این تابع بین فرزندان process فعلی می‌گردیم و آنیکه کمترین pid دارد را kill می‌کنیم.

```
int
kfc()
{
  struct proc *p  = myproc();
  acquire(&ptable.lock);
  int lowest_pid = INT_MAX;
  int found = 0;
  for(int i = 0; i < p->children_count; i++){
    if (p->children[i]->pid < lowest_pid){
      lowest_pid = p->children[i]->pid;
      found = 1;
    }
  }
  release(&ptable.lock);

  if (found)
    return kill(lowest_pid);

  return -1;
}
```

در تابع exit که در proc.c هست هم تکه کد زیر را اضافه کردیم تا وقتی process ای به پایان می‌رسد، آرایه فرزندانِ پدر این process را آپدیت می‌کنیم.

```
// update parent's children array
for(int i = 0; i < curproc->parent->children_count; i++){
  if(curproc->parent->children[i] == curproc){
    curproc->parent->children[i] = curproc->parent->children[curproc->parent->children_count - 1];
    curproc->parent->children[curproc->parent->children_count - 1] = 0;
    curproc->parent->children_count--;
    break;
  }
}
```

در نهایت با برنامه سطح کاربر kill_first_child.c سیستم‌کال را تست می‌کنیم.

```
int main(void)
{
    int pid;

    pid = fork();
    if (pid == 0)
      sleep(100);
    else if (pid > 0) {
      printf(1, "Parent process (pid %d) has %d children.\n", getpid(), get_children_count());
      kill_first_child();
      sleep(50);
      printf(1, "After killing first child parent process (pid %d) has %d children.\n", getpid(), get_children_count());
      wait();
    }
    else
      printf(1, "fork failed\n");

    exit();
}
```

خروجی به صورت زیر است.

```
$ kill_first_child
Parent process (pid 6) has 1 children.
After killing first child parent process (pid 6) has 0 children.
$
```

در پردازه پدر ابتدا kill_first_child( ) را کال می‌کنیم. برای اینکه نتیجه را بتوان نشان داد اندکی صبر می‌کنیم و بعد دوباره children_count را نشان می‌دهیم. چون kill کردن یک پردازه مقداری زمان می‌خواهد.