

آزمایشگاه سیستم عامل

تمرین کامپیوتری 2

اعضای گروه :

محمد سوری – 810100249

صفورا علوی پناه – 810100254

ریحانه حاجبی – 810100116

سوال یک: کتابخانه‌های (قاعداً سطح کاربر، منظور فایل‌های تشکیل دهنده متغیر **ULIB** در **Makefile** است) استفاده شده در **xv6** را از منظر استفاده از فراخوانی‌های سیستمی و علت این استفاده بررسی کنید.

در **makefile** چهار آبجکت فایل مربوط به **ULIB** داریم، که برنامه‌های سطح کاربر را پیاده‌سازی می‌کنند که یک **wrapper** برای فراخوانی‌های سیستمی است:

□ **Ulib**:

این کتابخانه شامل توابعی برای کار با استرینگ، آرایه‌ها و **i/o** است. این توابع در این فایل از فراخوانی‌های سیستمی استفاده می‌کنند:

۱. **Gets**: در این تابع از سیستم‌کال **read** استفاده می‌شود تا از **stdin** ورودی خوانده شود.

```
52 char*
53 gets(char *buf, int max)
54 {
55     int i, cc;
56     char c;
57
58     for(i=0; i+1 < max; ){
59         cc = read(0, &c, 1);
60         if(cc < 1)
61             break;
62         buf[i++] = c;
63         if(c == '\n' || c == '\r')
64             break;
65     }
66     buf[i] = '\0';
67     return buf;
68 }
```

۲. **Stat**: در این تابع از فراخوانی‌های **open** و **fstat** و **close** برای گرفتن اطلاعات یک فایل استفاده می‌شود.

```

70  int
71  stat(const char *n, struct stat *st)
72  {
73      int fd;
74      int r;
75
76      fd = open(n, O_RDONLY);
77      if(fd < 0)
78          return -1;
79      r = fstat(fd, st);
80      close(fd);
81      return r;
82  }

```

□ : Usys

در این فایل اینترایت هر سیستم‌کال وجود دارد:
این ماکرو سیستم‌کال را می‌گیرد و شماره را در رجیستر `eax` ذخیره می‌کند و سپس یک اینترایت سیستم‌کال اجرا می‌کند.

```

4  #define SYSCALL(name) \
5      .globl name; \
6      name: \
7          movl $SYS_ ## name, %eax; \
8          int $T_SYSCALL; \
9          ret

```

□ : Umaloc

اینجا توابع مربوط به مدیریت حافظه وجود دارد (`malloc` , `free` , ...)، و در آن از سیستم‌کال `sbrk` استفاده می‌شود.

```

46 static Header*
47 morecore(uint nu)
48 {
49     char *p;
50     Header *hp;
51
52     if(nu < 4096)
53         nu = 4096;
54     p = sbrk(nu * sizeof(Header));
55     if(p == (char*)-1)
56         return 0;
57     hp = (Header*)p;
58     hp->s.size = nu;
59     free((void*)(hp + 1));
60     return freep;
61 }

```

□ Printf : از فراخوانی سیستمی write استفاده می‌شود.

```

5 static void
6 putc(int fd, char c)
7 {
8     write(fd, &c, 1);
9 }

```

همچنین سه کتابخانه استاندارد C به جز glibc را نام برده و کاربرد خاص آن‌ها را بیان کنید.

- uClibc : برای embedded systems و مدیریت حافظه کاربرد دارد.
Embedded systems : به طور خاص برای سیستم‌های جاسازی شده و دیگر محیط‌های منابع محدود طراحی شده است که حافظه و قدرت پردازشی محدودی دارند. سعی می‌کند اندازه کوچکتری نسبت به کتابخانه‌های C دیگر ارائه دهد، که مناسب برای استفاده در دستگاه‌های با ذخیره‌سازی و حافظه محدود است.
- dietlibc : بر کاهش اندازه کد و مصرف حافظه تمرکز دارد در حالی که هنوز امکانات اصلی کتابخانه C را ارائه می‌دهد. این برای عملکرد و کارایی بالا هدف گذاری شده است و از تکنیک‌هایی مانند inline تابع و بهینه‌سازی برای معماری‌های خاص استفاده می‌کند.
- musl libc : سعی دارد بسیار کارآمد و سبک باشد و بر روی کم حجم بودن کد و بار زمان اجرایی تمرکز کند. بر امنیت و صحت در طراحی خود تأکید دارد و با توجه دقیق به سرریز شدن بافر و آسیب‌پذیری‌های دیگر،

طراحی می‌شود. این کتابخانه به طور معمول در سیستم‌های نهفته، توزیع‌های کم حجم (مانند Alpine Linux) و پروژه‌هایی که ابعاد کوچک و عملکرد مهم هستند، استفاده می‌شود.

سوال دو: انواع روش های دسترسی از سطح کاربر به سطح هسته را در لینوکس توضیح دهید.
دسترسی به سطح هسته از طریق اینترایت‌ها صورت می‌گیرد که اینترایت می‌تواند به دو صورت نرم افزاری و یا سخت افزاری باشد. اینترایت سخت افزاری همان طور که از اسمش پیداست، توسط سخت افزارها و دیوایس‌های I/O صورت می‌گیرد.
اینترایت نرم افزاری زمانی رخ می‌دهد که از در برنامه ی سطح کاربر از سیستم کالاستفاده کنیم. همچنین زمانی که در استثنایی مانند تقسیم بر صفر یا دسترسی غیر مجاز به حافظه رخ دهد هم اینترایت صورت می‌گیرد.
برنامه های سطح کاربر هم می‌توانند از طریق سیگنال ها با هم ارتباط داشته باشند که در اجرای آن ها نیز اینترایت نرم‌افزاری رخ می‌دهد.
همچنین با استفاده از `systems file pseudo` هم می‌توان به سطح هسته دسترسی پیدا کرد. در این روش به اطلاعات داده ساختارهای سطح هسته از طریق ساختارهای فایل مانند می‌تواند دسترسی پیدا کرد.

سوال سه: آیا باقی تله‌ها را نمی‌توان با سطح دسترسی `DPL_USER` فعال نمود؟ چرا؟
در `xv6` برای فعال کردن یک تله با سطح دسترسی `USER_DPL` برای تله دیگری باعث بروز یک `protection exception` می‌شود. این تدابیر امنیتی برای جلوگیری از مشکلات ممکن در برنامه های کاربری یا اقدامات خبیث اعمال شده است. اجازه دادن به کاربران برای اجرای تله ها با سطوح دسترسی بالاتر می‌تواند یک خطر جدی امنیتی ایجاد کند زیرا این اقدام ممکن است دسترسی غیرمجاز به هسته را فراهم کند و به تخریب کلی امنیت سیستم منجر شود. سخت افزار توسط معماری `x86` کنترل می‌شود و این سطوح دسترسی را اجرا میکند تا جدایی روشنی بین حالت کاربر و هسته حفظ شود و استثناء حفاظتی در صورت نقض این مراحل اجرایی بوجود آید.

سوال چهار : در صورت تغییر سطح `ss` دسترسی، و `esp` روی پشته `Push` می‌شود. در غیر این صورت `Push` نمی‌شود. چرا؟

وقتی که سطح دسترسی (`LevelPrivilege`) تغییر می‌کند، ممکن است مواردی مثل پشته نیاز به تغییر داشته باشند در معماری `ESP`، `x86` به عنوان اشاره گر به قسمت بالایی پشته استفاده می‌شود و `SS` نشان دهنده ی میزان دسترسی به پشته است. هنگامی که سطح دسترسی تغییر می‌کند (برای مثال، وارد

حالت کاربری می شویم یا از کد کاربر به کد سیستم عامل منتقل می شویم)، اطلاعات مربوط پشته نیاز به تغییر دارند. در صورت تغییر

سطح دسترسی، به ویژه هنگام رخ دادن یک trap، رجیسترهای “ss” و “esp” بر روی استک قرار داده می شوند.

این فرآیند برای تسهیل انتقال از استک کاربر به استک هسته حائز اهمیت است. دلیل این عمل مربوط به وجود دواستک است - استک کاربر و استک هسته. در زمان تغییر سطح دسترسی، به عنوان مثال در حین انتقال از حالت کاربر به حالت هسته، سیستم نیاز دارد که از استک هسته برای دسترسی به کد و ساختارهای داده ای که در دامنه هسته قرار دارند، استفاده کند. ابتدا، مقادیر فعلی “esp” و “ss” که به استک کاربر اشاره دارند، بر

روی استک ذخیره می شوند. این مقادیر ذخیره شده بعداً برای اشاره به استک هسته استفاده می شوند. این امکان را فراهم می کنند که کد هسته اجرا شود و به ساختارهای داده ای درون هسته دسترسی پیدا کند. پس از پردازش trap یا نقص، مقادیر قدیمی “esp” و “ss” بازیابی می شوند و این امکان را فراهم می کنند تا برنامه کاربر بی مشکل از جایی که متوقف شده بود ادامه یابد. این فرآیند اطمینان از صحت محیط اجرایی در طی انتقال

بین حالت های کاربر و هسته را فراهم می کند. مهم است که توجه داشته باشیم که اگر تغییری در سطح دسترسی رخ ندهد، به عبارت دیگر، اگر برنامه همچنان با همان استک کار کند، نیازی به ذخیره و بازیابی “esp” و “ss” وجود ندارد. این بهینه سازی جلوی انجام عملیات غیرضروری را هنگام عدم تغییر در سطح امتیاز می گیرد.

سوال پنج: در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در (argint) (به طور دقیقتر در (fetchint) بازه آدرس ها بررسی می گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ای ایجاد می کند؟

چهار تابع برای دسترسی به پارامترهای فراخوانی وجود دارند:

Argptr: در سیستم عامل xv6، تابع “argptr” برای بررسی صحت بازه آدرس های پارامترهای ارسالی به توابع فراخوانی سیستمی استفاده می شود. این تابع بررسی می کند که بازه آدرس ارائه شده در محدوده آدرس های قابل دسترس و معتبر در فضای آدرس کاربر است یا خیر. با انجام این بررسی، از وقوع آسیب پذیریه های امنیتی و دسترسی غیرمجاز به مناطق حافظه جلوگیری می شود.

argint: برای به دست آوردن یک عدد صحیح از فضای کاربری استفاده می شود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار آرگومان را از فضای

کاربری به فضای کرنل منتقل می کند. اگر عملیات با موفقیت انجام شود، مقدار 0 را برمی گرداند، در غیر این صورت ۱- برمی گرداند.

argstr: برای بازیابی یک رشته از فضای کاربری پردازش استفاده می شود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار رشته را از فضای کاربری به فضای کرنل کپی می کند. اگر عملیات با موفقیت انجام شود، مقدار 0 را برمی گرداند؛ در غیر این صورت ۱- برمی گرداند.

argfd: در xv6 برای بازیابی فایل دسکریپتور از فضای کاربری پردازش استفاده می شود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار فایل دسکریپتور را از فضای کاربری به فضای کرنل کپی می کند. در صورت موفقیت، مقدار 0 را برمی گرداند؛ در غیر این صورت ۱- برمی گرداند.

تمامی این توابع بررسی می کنند که آدرس داده شده حتما در حافظه پردازش قرار گیرد که یک پردازش نتواند به حافظه پردازش دیگری دسترسی پیدا کند زیرا این اتفاق ممکن است باعث مشکلات امنیتی در پردازش های دیگر شود. تجاوز از بازه معتبر آدرس در xv6 می تواند به نقض حفاظت حافظه، اجرای کد بد، و تخریب داده ها منجر شود. که این موضوع می تواند اجرای برنامه را دچار مشکل کند.

توابع دسترسی به پارامترهای فراخوانی سیستمی، مانند `argint` و `fetchint`، به طور کلی برای استخراج پارامترهایی که توسط برنامه های کاربردی بر روی سیستم ارسال می شوند، استفاده می شوند. این پارامترها ممکن است شامل آدرس های حافظه باشند که باید از سوی سیستم عامل خوانده شوند. وقتی که از این توابع استفاده می شود، بازه های آدرس های ارسالی بررسی می شوند تا اطمینان حاصل شود که دسترسی به منابع حافظه به طور قانونی و معتبر صورت می گیرد. این اقدام به کنترل دسترسی نامناسب به حافظه و جلوگیری از تجاوز به حافظه (مانند نوشتن یا خواندن از مناطقی که باید محافظت شوند) کمک می کند. در صورتی که تجاوزی از بازه معتبر اتفاق بیفتد، ممکن است به مشکلات امنیتی بزرگی منجر شود. به عنوان مثال، حملاتی مانند `buffer overflow` می توانند به وجود

ببایند که به مهاجم اجازه می‌دهد دسترسی به منابع سیستمی را بدست آورد و کنترل سیستم را در اختیار بگیرد. از این رو، بررسی صحیح بازه آدرس‌ها از اهمیت بالایی برخوردار است تا امنیت سیستم حفظ شود.

ارسال آرگومانهای فراخوانی های سیستمی

ابتدا برای اضافه کردن سیستم کال جدید مراحل زیر را انجام می‌دهیم:
تابع `count_num_of_digit` را در `user.h` تعریف می‌کنیم تا در سطح کاربر بتوان از آن استفاده کرد.

```
int count_num_of_digit
```

چون قرار است از طریق رجیستر ورودی را بخوانیم، پارامتری برای ورودی تابع نمی‌گذاریم.
سپس تعریف تابع را در `usys.S` می‌آوریم:

```
SYSCALL(count_num_of_digit)
```

این ماکرو ، عدد سیستم کال را در `eax` ذخیره می‌کند و یک اینترپت از نوع سیستم کال ایجاد می‌کند.
حال در فایل `syscall.h` شماره ی سیستم کال جدید را مشخص می‌کنیم:

```
define SYS_count_num_of_digit  
2
```

درنهایت در فایل `syscall.c` سیستم کال را به آرایه های سیستم کال ها اضافه می‌کنیم و تابع سیستم کال را نیز معرفی می‌کنیم:

```
extern int sys_count_num_of_digit(void);  
[SYS_count_num_of_digit] sys_count_num_of_digit,
```

پیاده سازی منطق آن را هم در `sysproc.c` انجام می‌دهیم و در تابع سیستم کال هم ورودی را از رجیستر `ebx` می‌خوانیم:


```

94
95 static int
96 count_num_of_digit(int n){
97     int count = 0;
98
99     do {
100         n /= 10;
101         ++count;
102     } while (n != 0);
103     return count;
104
105 }
106
107 int
108 sys_count_num_of_digit(void)
109 {
110     return count_num_of_digit(myproc()-> tf -> ebx);
111 }
112

```

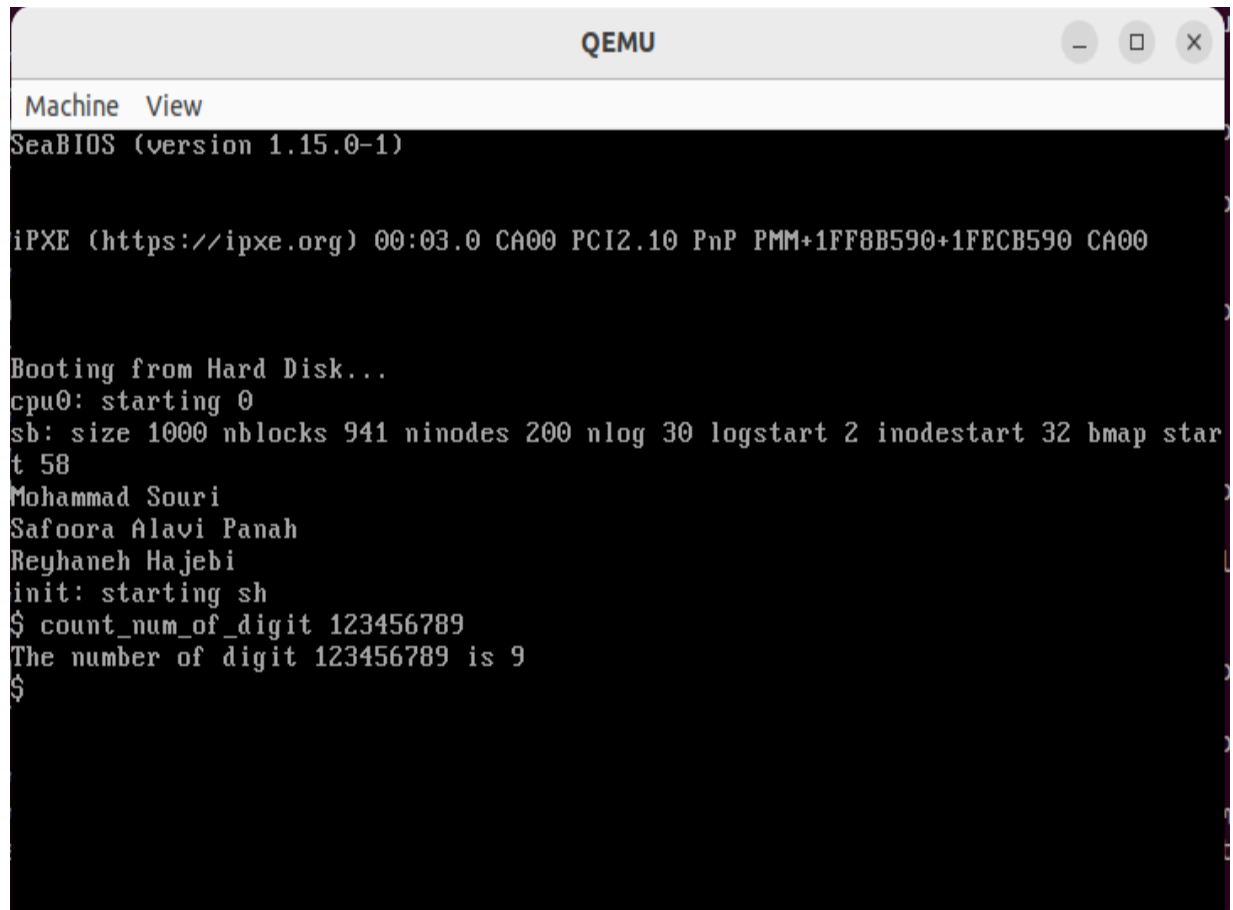
در برنامه ی سطح کاربر ابتدا باید ورودی را در رجیستر ebx ذخیره کنیم و همچنین مقدار اولیه را ذخیره میکنیم تا بتوانیم بعد از سیستم کال مقدار اولیه را به ebx برگردانیم:

```

C count_num_of_digit.c > count_num_of_digit_handler(int)
1  #include "types.h"
2  #include "user.h"
3  #include "stat.h"
4
5  int
6  count_num_of_digit_handler(int num){
7
8      int prev_ebx;
9
10     asm volatile(
11         "movl %%ebx, %0\n\t"
12         "movl %1, %%ebx"
13         : "=r"(prev_ebx)
14         : "r"(num)
15     );
16
17     int result = count_num_of_digit();
18
19     asm volatile(
20         "movl %0, %%ebx"
21         :: "r"(prev_ebx)
22     );
23
24     return result;
25 }
26
27 int
28 main(int argc, char* argv[]){
29     if (argc < 2) {
30         printf(1, "Usage: count_num_of_digit <num>\n");
31         exit();
32     }
33     int input = atoi(argv[1]);
34     int result = count_num_of_digit_handler(input);
35     /*if (result < 0){
36         printf(2, "number should be positive\n");
37         exit();
38     }*/
39     printf(1, "The number of digit %d is %d\n", input, result);
40     exit();
41 }

```

اجرای نهایی این دستور در سیستم عامل:



```
QEMU
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 start 58
Mohammad Souri
Safoora Alavi Panah
Reyhaneh Hajebi
init: starting sh
$ count_num_of_digit 123456789
The number of digit 123456789 is 9
$
```

Github link of project : [link](#)

Last commit message : final clean up