OPERATING SYSTEM LAB

SESSION 2

SeyedMehdi HajiSeyedHossein	810100118
Alireza Hosseini	810100125
AmirAli Shahriary	810100173

سـوال ۱ : کتابخانه های (قاعدتاً سـطح کاربر، منظور فایلهای تشـکیل دهنده متغیر **ULIB** در **Makefile** اسـت) استفاده شده در **xv6** را از منظر استفاده از فراخوانی های سیستمی و علت این استفاده بررسی نمایید.

```
146 ULIB = ulib.o usys.o printf.o umalloc.o
147
148 _%: %.o $(ULIB)
149     $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
150     $(OBJDUMP) -S $@ > $*.asm
151     $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym
```

متغییر **ULIB** شامل چندین فایل .**o** است :

- کتابخانه ulib : شامل چندین تابع هست که بیشتر برای کار با آرایه ها هستند ولی فقط چند تابع آن از سیستم کال استفاده میکنند . مانند تابع memset , stats , gets .
- تابع memset : برای ست کردن یک مقدار مشخص در آرایه که از سیستم کال stosb استفاده میکند.

```
36  void*
37  memset(void *dst, int c, uint n)
38  {
39    stosb(dst, c, n);
40    return dst;
41  }
```

- تابع stats : که از سیستم کال هایی چون open , close , fstat برای باز کردن فایل دیسکریپتور و خواندن آن فایل دیسکریپتور استفاده میکنیم .

```
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  }
```

- تابع gets : که از stdin یک چیزی خوانده میشود با استفاده از فایل دیسکریپتور 0 و در بافر ذخیره میکند .

```
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  }
69
```

- Usys.o : که شامل یک ماکرو هست که به هر سیستم کال یک آیدی اختصاص میدهد .

```
4   #define SYSCALL(name) \
5     .globl name; \
6     name: \
7       movl $SYS_ ## name, %eax; \
8       int $T_SYSCALL; \
9       ret
10
```

- printf.O : این فایل در ابتدا یک تابع به نام PUTC تعریف کرده که در آن از سیستم کال write استفاده میکند بدین صورت که یک فایل دیسکریپتور میگیرد و سپس در آن مینویسید .
- حالا در دو تابع دیگر این فایل printf , printint از تابع putc استفاده میکنند

```
5   static void
6   putc(int fd, char c)
7   {
8     write(fd, &c, 1);
9   }
10
```

- Umalloc.o : دارای چندین تابع هست که فقط یکی از آنها از سیستم کالی به نامه SBRK استفاده میکنند . در کل این توابع بیشتر برای اختصاص دادن حافظه هستند .
- تابع morecore وظیفه ی افزایش حافظه را دارد که با سیستم کال sbrk ، دیتا سگمنت را افزایش میدهد .
- تابع malloc از تابع morecore استفاده میکند.

```
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  }
```

سوال ۲ : فراخوانی های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روشها را در لینوکس به اختصار توضیح دهید.

Socket communication

در این روش برنامه سطح کاربر با استفاده از socket descriptor به یک سوکت گوش میدهد تا اطلاعات لازم را از کرنل دریافت کند .

Exceptions

در این موارد هم وقتی خطایی مثلا مثل divide by 0 رخ میدهد برنامه به دست کرنل سپرده میشود تا آن را هندل کند . و سپس آن را به حالت user mode برمیگرداند .

Pseudo fileSystem (virtual fileSystem)

در این روش انگار کرنل برخی از اطلاعات را در یکسری filesystem به اشتراک میگذارد .

{Since everything is a file is the general philosophy of Linux. Working on that premise there are filesystems which expose some of the kernel resources over the file interface.}

این فایل سیستم ها لزوما فایل نیستند میتوانند یک سری entry مجازی باشند که اطلاعات را مشابه فایل در اختیار برنامه های سطح کاربر قرار میدهد.

که این filesystem ها شامل :

/dev

/sys

/proc

Network filesystem(NFS)

مثلا به عنوان مثلا در : proc/

/proc/cpuinfo شامل اطلاعاتی از cpu مانند هسته ها و تعداد انها و ... هست.

/proc/meminfo شامل اطلاعاتی در باره ی حافظه فیزیکی است.

proc/intruppts شامل اطلاعاتی درباره ی وقفه ها و هندلر های مخصوص انهاست .

سوال ۳ : آیا باقی تله‌ها را نمی‌توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟
قطعا خیر .
چون اگر میخواستیم باقی trap ها با همین سطح دسترسی فعال بشن راحت میتونستند که به kernel با privilage mode دسترسی داشته باشند و آن وقت کرنل در خطر بود .
همچنین اگر یک prosecc بخواهد intrrypt دیگری را هم قعال کند سیستم عامل نباید این اجازه را صادر کند چون در این صورت اگر مثلا در برنامه سطح کاربر باگی داشته باشیم اون وقت کرنل در خطر است .
در واقع وقتی پروسس بخواهد intrrupt دیگری را فعال کند سیستم عامل با یک استثنایی به اسم protextion execption آن را جواب میدهد.
سوال ۴ : در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟
به طور کلی دو پشته یکی برای سطح کاربر و دیگری برای kernel موجود است. همچنین رجیستر ss(stack segment) برای نگه داشتن مقدار segment ایست که درآنجا برنامه تغییر مد داده شده و رجیستر ESP(extended Stack Pointer) برای نگه داشتن آدرس تاپ استک است در جایی که برنامه تغییر مد داده شده .
هنگامی که میخواهیم دسترسی را تغییر بدهیم، مثال از سطح کاربر به kernel برویم، دیگر نمیتوانیم از پشته قبلی استفاده کنیم.
پس باید ss و esp روی پشته push بشوند تا بتوان دوباره پس از بازگشت از سطح دسترسی دیگر از آنها استفاده کرد و اطلاعات از دست نروند.

از طرفی وقتی تغییر سطح دسترسی نداشته باشیم، نیازی به push کردن ss و esp نیست. چون به همان پشته هنوز دسترسی داریم.
همچنین ALLtraps باقی ثبات هارا در استک پوش میکند تا بعد از اتمام تغییر سطح دسترسی و بازگشت به مد قبلی همه رجیستر ها قابل بازیابی باشند .

سوال ۵ : در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr() بازه آدرسها بررسی میگردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی بازهها در این تابع، مثالی بزنید که در آن، فراخوانی سیستم sys_read() اجرای سیستم را با مشکل روبرو سازد

همه ی توابع argptr , argstr , argint در فایل syscall.c قابل بررسی اند .
این توابع میتوانند به پارامتر های پاس داده شده به سیستم کال های در سطح کرنل دسترسی داشته باشند و انها را مشخص کنند . این عمل از روی استکی که در صورت پروژه مطرح شده صورت میگیرد :

esp+12	c
esp+8	b
esp+4	a
esp	Ret Addr

```
10  // User code makes a system call with INT T_SYSCALL.
11  // System call number in %eax.
12  // Arguments on the stack, from the user call to the C
13  // library system call function. The saved user %esp points
14  // to a saved program counter, and then the first argument.
15
16  // Fetch the int at addr from the current process.
17  int
18  fetchint(uint addr, int *ip)
19  {
20    struct proc *curproc = myproc();
21
22    if(addr >= curproc->sz || addr+4 > curproc->sz)
23      return -1;
24    *ip = *(int*)(addr);
25    return 0;
26  }
27
```

```
28  // Fetch the nul-terminated string at addr from the current process.
29  // Doesn't actually copy the string - just sets *pp to point at it.
30  // Returns length of string, not including nul.
31  int
32  fetchstr(uint addr, char **pp)
33  {
34    char *s, *ep;
35    struct proc *curproc = myproc();
36
37    if(addr >= curproc->sz)
38      return -1;
39    *pp = (char*)addr;
40    ep = (char*)curproc->sz;
41    for(s = *pp; s < ep; s++){
42      if(*s == 0)
43        return s - *pp;
44    }
45    return -1;
46  }
47
```

- Argint:

این تابع شماره اینکه چندمین آرگون را میخواهیم میگیرد و همچنین یک آدرس میگیرد و مقدار n امین آرگومان ورودی سیستم کال را از روی استک در آن آدرس میدهد .

```
47
48    // Fetch the nth 32-bit system call argument.
49    int
50  argint(int n, int *ip)
51    {
52      return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
53    }
54
```

- Argstr:

این تابع ایندکس آرگومان ورودی ای که مربوط به سیستم کال موردنظر است را به همراه یک پوینتر به یک متغییر از نوع پوینتر به کاراکتر میگرد(یعنی انگار پوینتری به ارایه ای از کاراکتر ها در واقع) و مقدار nامین پارامتر ان سیستم کال که روی استک است را در این متغییر ذخیره میکند.

```
72  // Fetch the nth word-sized system call argument as a string pointer.
73  // Check that the pointer is valid and the string is nul-terminated.
74  // (There is no shared writable memory, so the string can't change
75  // between this check and being used by the kernel.)
76  int
77  argstr(int n, char **pp)
78  {
79    int addr;
80    if(argint(n, &addr) < 0)
81      return -1;
82    return fetchstr(addr, pp);
83  }
```

- Argptr:

این تابع ایندکس آرگومان ورودی ای که مربوط به سیستم کال موردنظر است را به همراه آدرس یک پوینتر و همچنین سایز مقدار آرگومانی که میخواهیم بخوانیم را میگیرد و مقدار آن آرگومان ورودی را در ان قسمت از حافظه که به همراه سایز آن مشخص کرده ایم میریزد .

```
55  // Fetch the nth word-sized system call argument as a pointer
56  // to a block of memory of size bytes.  Check that the pointer
57  // lies within the process address space.
58  int
59  argptr(int n, char **pp, int size)
60  {
61    int i;
62    struct proc *curproc = myproc();
63
64    if(argint(n, &i) < 0)
65      return -1;
66    if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
67      return -1;
68    *pp = (char*)i;
69    return 0;
70  }
```

نکته : در صورتی که چک کردن محدوده حافظه ها در تابع argptr صورت نگیرد ممکن است که در یک آدرسی که مجاز به نوشتن نیستیم بگوییم که بیا در این آدرس مقدار آرگون ورودی فلان سیستم کال را قرار بده .

نکته ۲ : در لینوکس برخلاف xv6 پارامترهای فراخوانی سیستمی در ثبات منتقل میگردند.

یعنی در لینوکس در سطح اسمبلی، ابتدا توابع wrapper پارامترها را در پشته منتقل نموده و سپس پیش از فراخوانی فراخوانی سیستمی، این پارامترها ضمن جلوگیری از از دست رفتن محتوای register ها درآنها کپی میگردند .

بررسی گامهای اجرای فراخوانی سیستمی در سطح کرنل توسط gdb :

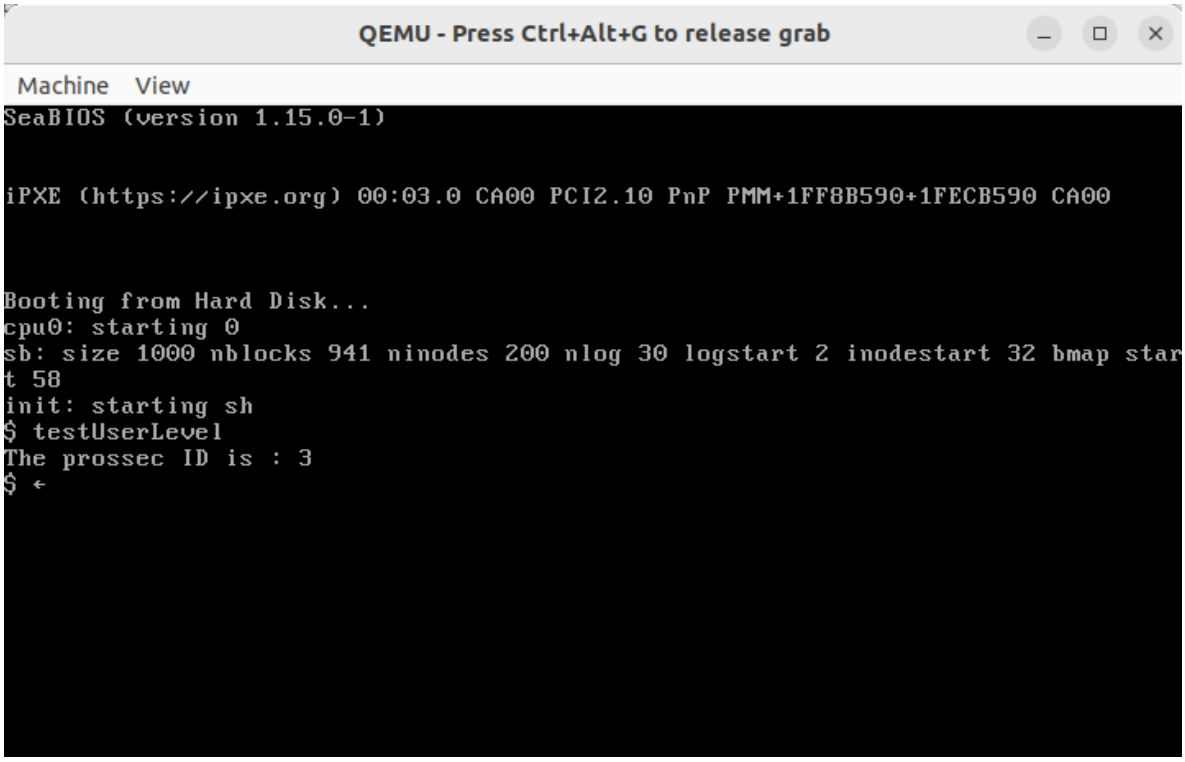
برای بررسی روند اجرای یک سیستم کال (در اینجا getpid) برنامه سطح کاربری نوشتیم .

testUserLevel.c

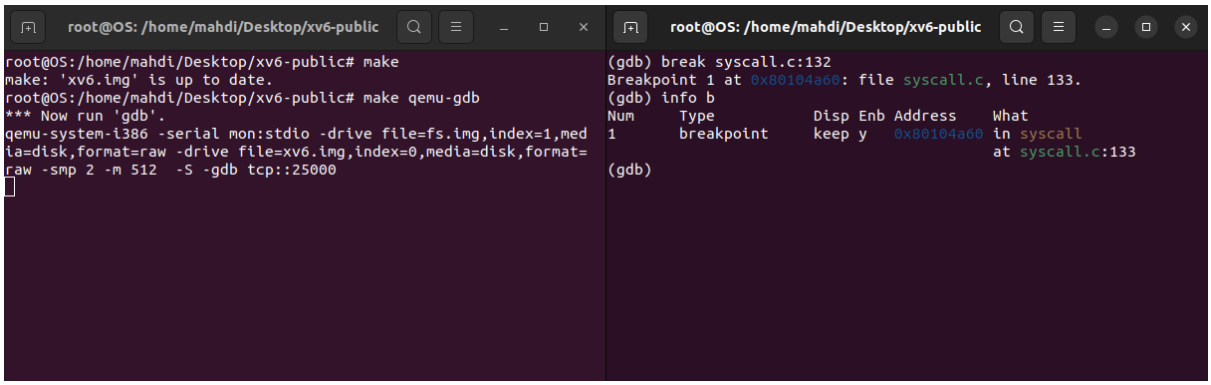
```c
#include "types.h" // for basic types like int , char , ...
#include "stat.h"  // fro file status constatnts .
#include "user.h" // for user-level program systemcalls and definitiosn



int main (){
    int thePID = getpid(); // getpid system call

    printf(1 ,"The prossec ID is : %d\n" , thePID); // this write the pid in stdout (1<-fd)
    exit();


}
```

و سپس آن را به makefile در متغییر های UPROGRS که برای برنامه های سطح کاربر هستن اضافه میکنیم.

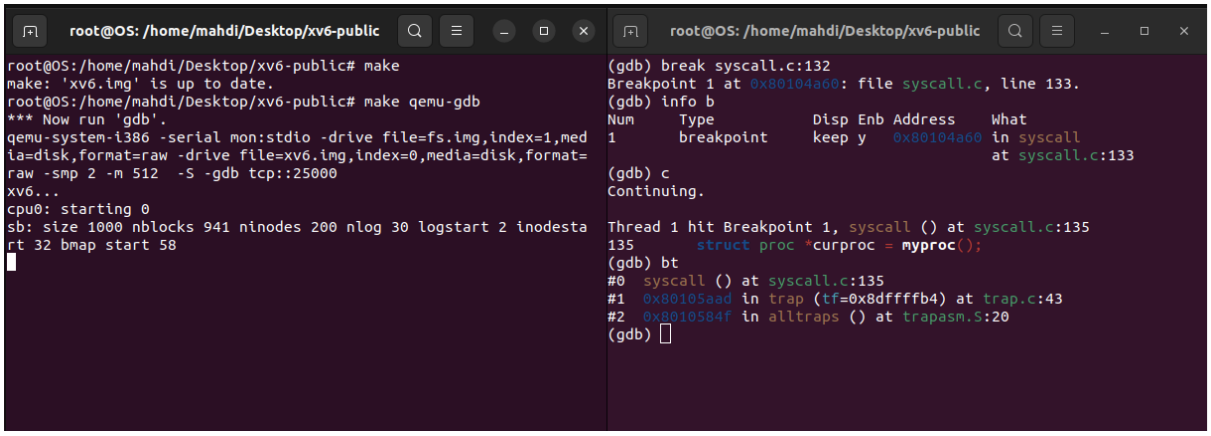
```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_testUserLevel\

```

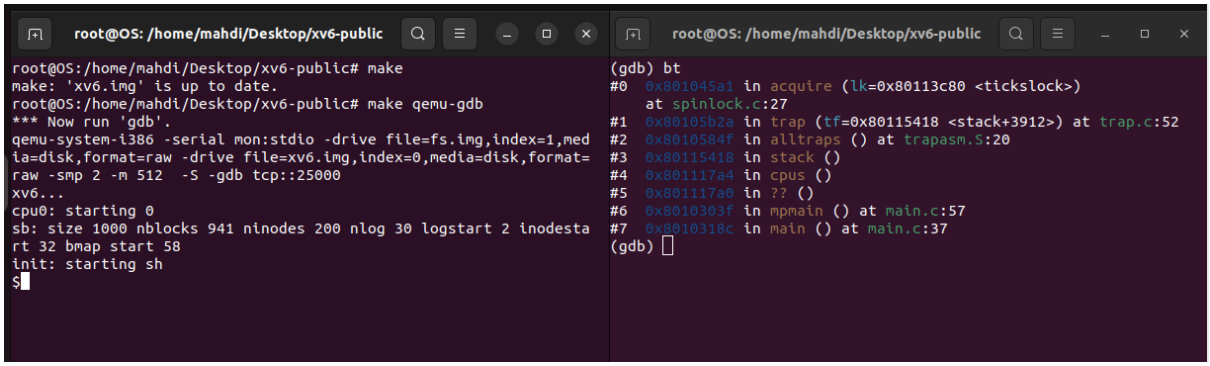


سپس breakpoint ای در gdb در ابتدای تابع syscall قرار میدهیم .



حالا ادامه میدهیم تا به breakpoint برسیم .

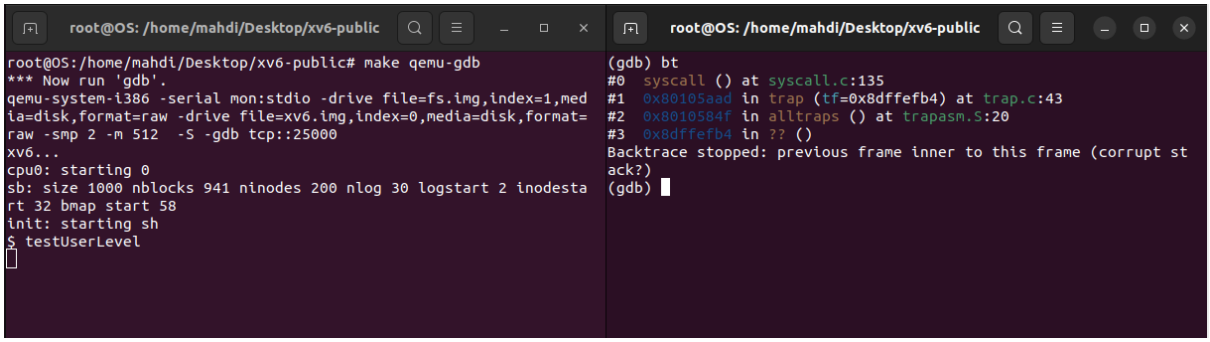




در ابتدای اجرای سیستم برای انیکه در ترمینال مقدار init : statring sh را بنویسد چندین بار به بریک پونیت گذاشته شده برخورد میکنیم . (چون در واقع همش در حال صدا زدن سیستم کال write هستیم)

حالا میریم سراغ اینکه برنامه testUserLevel را اجرا کنیم . پس از اجرای این دستور باز هم به bp برمیخوریم حالا به اجرای دستور bt یا همان backtracing میپردازیم که وظیفه این دستور را در ادامه شرح میدهیم :

دستور backtrace یک stack trace از تمام فانکشن هایی که تا به اینجا صدا زده شده اند را به ما نشان میدهد . همچنین با این دستور میتوان دید که این فانشکن ها هر کدام در کجای برنامه کال شده اند .



این stack trace نشان میدهد که برای اینکه برنامه testUserLevel اجرا شود باید به دستورهای trap و alltrapsدر فایل زیر دسترسی داشته باشد . درواقع این دستور یک trap frame برای دسترسی به کرنل مود ایجاد میکند که این تمام کار های ذکر شده در صورت پروژه از جمله ذخیره استک ها و همچنین رجیستر ها را انجام میدهد . و یک trap برای دسترسی به privilege mode ایجاد میکند. (تصویر صفحه بعد)

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret

```

در واقع trap frame یک استراکت درون فایل x86.h است که اطلاعات رجیستر ها را در خود قبل از عوض کردن مود نگه میدارد:

```
150  struct trapframe {
151    // registers as pushed by pusha
152    uint edi;
153    uint esi;
154    uint ebp;
155    uint oesp;      // useless & ignored
156    uint ebx;
157    uint edx;
158    uint ecx;
159    uint eax;
160
161    // rest of trap frame
162    ushort gs;
163    ushort padding1;
164    ushort fs;
165    ushort padding2;
166    ushort es;
167    ushort padding3;
168    ushort ds;
169    ushort padding4;
170    uint trapno;
171
172    // below here defined by x86 hardware
173    uint err;
174    uint eip;
175    ushort cs;
176    ushort padding5;
177    uint eflags;
178
179    // below here only when crossing rings, such as from user to kernel
180    uint esp;
181    ushort ss;
182    ushort padding6;
183  };
184
```

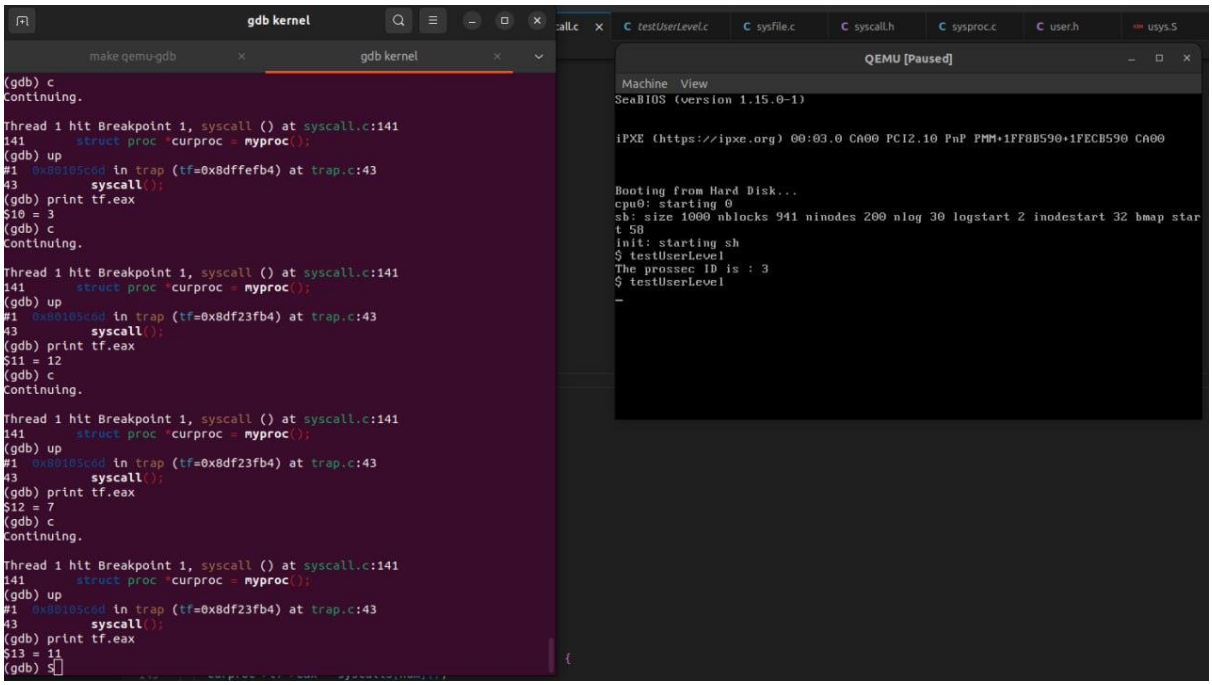
حالا طبق صورت پروژه ابتدا یک برنامه سطح کاربر مینویسیم که از سیستم کال getPID استفاده کند . و گام به گام با breakpoint ای که در قسمت syscall گذاشته ایم پیش میرویم .

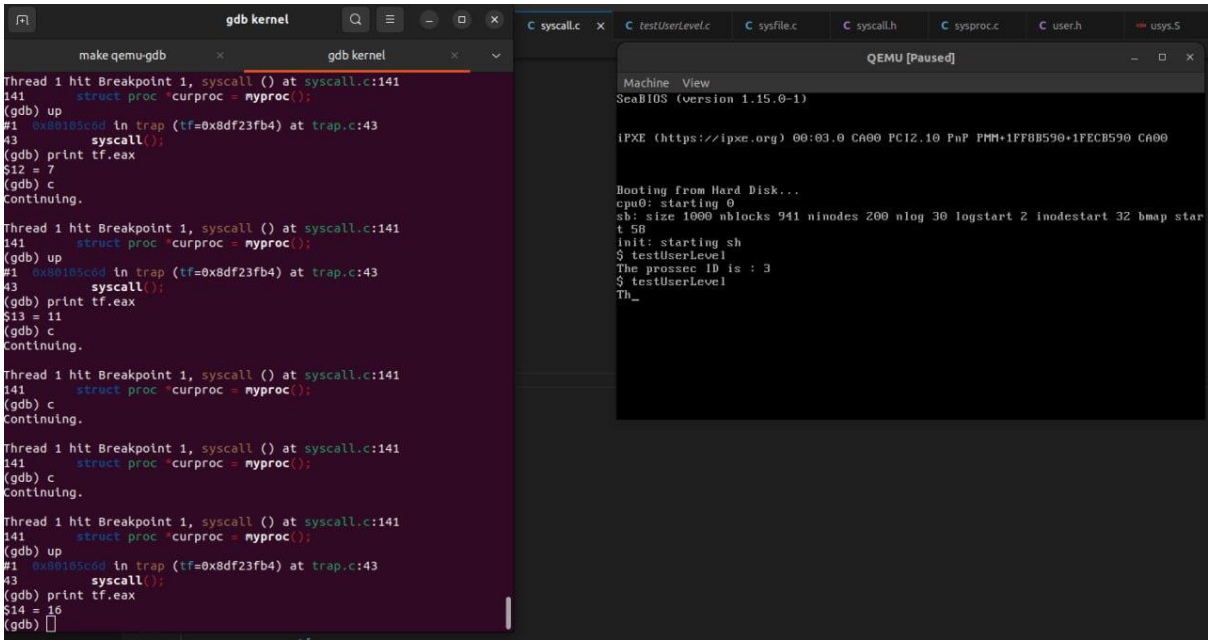
در ابتدا trapFram.eax که نگه دارنده شماره ی مربوط به سیستم کال صدا زده شده است دارای مقدار ۵ را داراست که بدین معنی است که درحال خواندن از ورودی با سیستم کال read است .

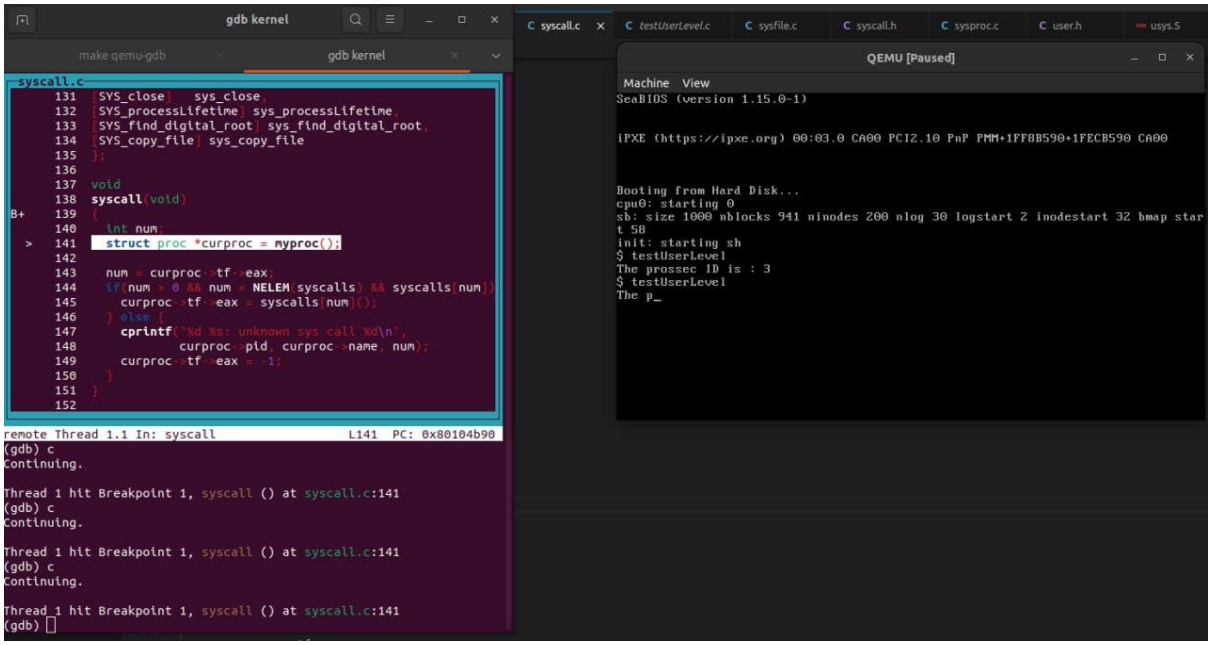
بعد سیستم کال مربوط به fork با شماره ی ۱ و سپس سیستم کال exec با شماره ۷ و سپس سیستم کال wait باشماره ۳ و پس از آن سیستم کال srbk با شماره ۱۲ ، و در نهایت سیستم کال مورد نظر ما یعنی getpid باشماره ی ۱۱ اجرا میشود . در نهایت پس آن سیستم کال exit اجرا میشود .

و پس از آن به چند مقدار که برای write کردن نتیجه روی ترمینال اختصاص دارد مقدار 16 یعنی سیستم کال write درون متیغیر trapfram.eax ذخیره میگردد .

```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```




gdb kernel
make qemu-gdb
gdb kernel
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141 struct proc *curproc = myproc();
(gdb) up
#1 0x80105c6d in trap (tf=0x8df23fb4) at trap.c:43
43 syscall();
(gdb) print tf.eax
$12 = 7
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141 struct proc *curproc = myproc();
(gdb) up
#1 0x80105c6d in trap (tf=0x8df23fb4) at trap.c:43
43 syscall();
(gdb) print tf.eax
$13 = 11
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141 struct proc *curproc = myproc();
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141 struct proc *curproc = myproc();
(gdb) up
#1 0x80105c6d in trap (tf=0x8df23fb4) at trap.c:43
43 syscall();
(gdb) print tf.eax
$14 = 16
(gdb)
syscall.c
testUserLevel.c
sysfile.c
syscall.h
sysproc.c
user.h
usys.S
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 start 58
init: starting sh
$ testUserLevel
The prossec ID is : 3
$ testUserLevel
Th_


gdb kernel
make qemu-gdb
gdb kernel
syscall.c
131 [SYS_close] sys_close,
132 [SYS_processLifetime] sys_processLifetime,
133 [SYS_find_digital_root] sys_find_digital_root,
134 [SYS_copy_file] sys_copy_file
135 };
136
137 void
138 syscall(void)
139 {
140 int num;
141 struct proc *curproc = myproc();
142
143 num = curproc->tf->eax;
144 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
145 curproc->tf->eax = syscalls[num]();
146 } else {
147 cprintf("%d %s: unknown sys call %d\n",
148 curproc->pid, curproc->name, num);
149 curproc->tf->eax = -1;
150 }
151 }
152
remote Thread 1.1 In: syscall L141 PC: 0x80104b90
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
(gdb)
testUserLevel.c
sysfile.c
syscall.h
sysproc.c
user.h
usys.S
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 start 58
init: starting sh
$ testUserLevel
The prossec ID is : 3
$ testUserLevel
The p_