به نام خدا

مهدی خصالی: 81010134 علی داداشی: 810100138 محمدمهدی داورزنی: 810100140

پرسش 1:

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

متغیر ULIB متشکل از چهار آبجکت فایل است که در بالا مشاهده میکنید این کتابخانه‌ها توابع و سیستم کالهای پایه در xv6 را در خود دارند.

ulib.o: در این کتابخانه توابعی برای فرایندهای I/O همچون read, write, open, close وجود دارد که در سطح کاربر اجرا میشوند. همچنین توابعی مانند fork, signal, exit, kill هم در این کتابخانه قرار دارند.

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

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

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

پرسش 2:

- Procfs, sysfs, and similar mechanisms. This includes /dev entries as well, and all the methods in which kernel space exposes a file in user space (/proc, /dev, etc. entries are basically files exposed from the kernel space).
- Socket based mechanisms. Netlink is a type of socket, which is meant specially for communication between user space and kernel space.
- System calls.
- Upcalls. The kernel executes a code in user space. For example spawning a new process.
- mmap - Memory mapping a region of kernel memory to user space. This allows both the kernel, and the user space to read/write to the same memory area.

پرسش 3:

خیر. اگر یک پردازه بخواهد interrupt دیگری را فعال کند، xv6 به آن این اجازه را نمی‌دهد و با یک protection exception مواجه می‌شوند که به vector شماره ۱۳ می‌روند. زیرا ممکن است در برنامه سطح کاربر باگی وجود داشته باشد، یا کاربر سوءاستفاده کند و امنیت سیستم به خطر بیفتد.

سطح دسترسی USER_DPL سطح کاربر است و اگر کاربر امکان اجرای این تله‌ها را داشت به راحتی می‌توانست به kernel دسترسی داشته باشد و امنیت سیستم به خطر می‌افتاد.

پرسش 4:

رجیسترهای 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 رجیسترها به استک نیست زیرا وضعیت استک ثابت میماند.

پرسش 5:

در صورت عدم بررسی بازه ها در این تابع مثالی بزنید که در آن، فراخوانی سیستمی sys_read() اجرای سیستم را با مشکل رو به رو سازد.

تابع هایی که برای دسترسی به فراخوانی های سیستمی تعریف شده‌اند عبارت‌اند از: **argint**, **argptr**, **argstr**

هر تابع در صورت آرگومان غیرمجاز مقدار 1- را برمی‌گرداند.

تابع argint: در این تابع ابتدا آدرس آرگومان N-ام محاسبه می‌شود. همانطور که در می‌دانیم و در صورت پروژه ذکر شده پشته از آدرس بیشتر به کمتر رشد می‌کند و از آرگومان آخر به اول در آن ذخیره می‌شود و در آخر آدرس نقطه بازگشت در سر پشته قرار خواهد گرفت. از آنجا که آدرس پشته در رجیستر Esp ذخیره می‌شود میتوانیم آدرس آرگومان n-ام را با استفاده از رابطه ptr= esp + 4 + 4*n به دست بیاوریم.

در انتها این آدرس همراه اشاره‌گر به حافظه مدنظر برای مقدار int به تابع fetchint ارسال می‌شود. این تابع ابتدا بررسی کرده که آیا آدرس ارسالی 4 بایت در حافظه پردازه باشد و در اینصورت آرگومان دوم را مقدار دهی می‌کند.

تابع argstr: این تابع با استفاده از تابع argint آدرس مربوط به ابتدای رشته را مشخص کند و بعد این مقدار را به تابع fetchstr می‌دهد. در این تابع ابتدا بررسی می‌شود که آیا آدرس داده شده در حافظه پردازه وجود دارد و سپس مقدار آرگومان دوم را برابر این اشاره‌گر قرار میدهد، سپس از ابتدای پوینتر شروع به پیمایش کرده و اگر به کاراکتر نال رسید طول رشته را برمی‌گرداند و در غیر این صورت اگر به انتهای به حافظه رسید 1- را برمی‌گرداند

تابع argptr: این تابع با استفاده از تابع argint آدرس اشاره‌گر را دریافت کرده و سپس آرگومان سوم که سایز پوینتر است را نیز با استفاده از argint دریافت می‌کند و بررسی می‌کند که اشاره‌گر با آن سایز در حافظه پردازه موجود باشد و اگر مشکلی وجود نداشت سپس آرگومان دوم را مقداردهی می‌کند.

همه توابع بررسی میکنند که آدرس حتما در حافظه پردازه وجود داشته باشد تا یک پردازه نتواند به حافظه یک پردازه دیگر دسترسی پیدا کند زیرا در غیر اینصورت باعث مشکلات امنیتی یا باگ در دیگر پردازه ها می‌شود زیرا اگر در حین دسترسی حافظه ای از پردازه دیگر را تغییر دهد که آن پردازنده به آن نیاز دارد در روند آن پردازنده مشکلات ذکر شده به وجود خواهد آمد.

بررسی گام‌های اجرای فراخوانی سیستمی در سطح کرنل توسط GDB

ابتدا برنامه سطح کاربر زیر را ایجاد میکنیم که PID را چاپ میکند.

Pid.c > main(void)

```c
#include "types.h"
#include "user.h"

int main(void)
{
    int CurrentProcessId = getpid();
    printf(1, "current process ID : %d\n", CurrentProcessId);
    exit();
}
```

سپس xv6 با دستور make کامپایل نموده و با make qemu-gdb اجرا میکنیم. همزمان در ترمینالی دیگر دستور gdb را اجرا کرده و با دستور target remote و قرار دادن localhost:26000 آنرا به qemu متصل میکنیم. با دستور break syscall روی تابع ()syscall بریک پوینت میگذاریم و سپس C را اجرا میکنیم.

```
ali@ali-VMware-Virtual-Platform:~/Desktop/xv6-public$ gdb
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 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/ali/Desktop/xv6-public/.gdbinit" auto-loading has been declined by you
r `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/ali/Desktop/xv6-public/.gdbinit
line to your configuration file "/home/ali/.config/gdb/gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/ali/.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) target remote localhost:26000
Remote debugging using localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000fff0 in ?? ()
(gdb) file kernel
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from kernel...
(gdb) c
Continuing.
```

حال نوبت اجرای دستور Pid است، با اجرای آن بریک پوینت هیت میشود.

```
init: staali@ali-VMware-Virtual-Platform:~/Desktop/xv6-public$ make qemu-gdb
*** 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
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ Pid
current process ID : 3
$ Pid
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:133
133     {
(gdb) bt
#0  syscall () at syscall.c:133
#1  0x80105b41 in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80105883 in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

با اجرای دستور bt کال استک این بریک پوینت نشان داده میشود. از دستور up استفاده میکنیم تا به محل قرارگیری آبجکت tf در کال استک برویم. با دستور print tf.eax محتوای رجیستر eax نمایش داده میشود که در بین تمام بریک پوینتهایی که فعال میشوند یکی از آنها مقدار pidای است که برای ما نمایش داده میشود. بقیه مربوط به processهای سطح کرنل هستند. لذا نتیجه میشود که PID در eax ذخیره میشود.

افزودن چند سیستم کال به کرنل xv6

برای افزودن هر سیستم کال باید یک شماره جدید به آن منسب کرد. فراخوانی‌های سیستمی و مکانیزم پاس دادن توابع را تعیین نمودو در آخر توسط برنامه سطح کاربر به آن دسترسی پیدا کرد.

برخی تغییرات و کدهای اضافه شده برای دو سیستم کال `Int sort_syscalls(int pid)` و `Int get_most_invoked_syscall(int pid)` و `Int list_all_processes()` به صورت زیر است.

```
int sort_syscalls(int pid){
  struct proc *p;
  int i;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid){
      int syscall_invoke[NUMBER_OF_SYSCALLS];
      for (i = 0; i < NUMBER_OF_SYSCALLS; i++)
      {
        syscall_invoke[i] = p->invoke_count[i];
      }
      for (i = 0; i < NUMBER_OF_SYSCALLS-1; i++)
      {
        cprintf("Syscall %d invoked %d times\n", i+1, syscall_invoke[i]);
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

ابتدا توابع در فایل proc.c تعریف و پیاده سازی میشوند. سپس در فایل syscall.c باید هر کدام از آنها تعریف و به کرنل شناسانده شوند. در فایل syscall.h و user.h هر سیستم کال را تعریف میکنیم و به سطح یوزر و کرنل میشناسانیم.

```c
int
get_most_invoked_syscall(int pid)
{
  struct proc *p;
  int max = -1;
  int most_invoked_syscall = -1;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      // Find syscall with maximum count
      for(int i = 0; i < NUMBER_OF_SYSCALLS; i++)
      {
        if(p->invoke_count[i] > max)
        {
          max = p->invoke_count[i];
          most_invoked_syscall = i;
        }
      }
      break;
    }
  }
  release(&ptable.lock);
  return most_invoked_syscall;
}
```

```c
int
list_all_processes(void)
{
  struct proc *p;

  acquire(&ptable.lock);
  cprintf("Process ID\tTotal Syscall Invokes\n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state != UNUSED)
    {
      int total_syscall_invokes = 0;
      for(int i = 0; i < NUMBER_OF_SYSCALLS; i++)
      {
        total_syscall_invokes += p->invoke_count[i];
      }
      cprintf("%d       \t%d\n", p->pid, total_syscall_invokes);
    }
  }
  release(&ptable.lock);
  return 0;
}
```

سیستم کالهای اضافه شده را به فایل sysproc.c هم اضافه میکنیم. این فایل در واقع محل شناسایی سیستم کالها و هندل کردن آنها در سطح یوزر است.

فایل user.h هم نقش مشابهی دارد، همچنین سیستم کالها باید در فایل proc.c هم تعریف شوند که در واقع برای این است که آنها به عنوان یک پروسس تعریف شوند.

نهایتا چند برنامه نیز برای تست این سیستم کالها اضافه میکنیم.

```c
int sys_move_file(void) {
  char *oldpath, *newpath;
  char name[DIRSIZ];
  struct inode *ip, *dp,*np;
  uint off;


  if (argstr(0, &oldpath) < 0 || argstr(1, &newpath) < 0)
    return -1;

  begin_op();

  if ((ip = namei(oldpath)) == 0) {
    end_op();
    return -1;
  }

  ilock(ip);
```

mvfile.c > main(int, char * [])

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"


int main(int argc, char *argv[])
{
  if (move_file(argv[1], argv[2]) == -1)
  {
    printf(2, "move: Error moving file\n");
  }
  else
  {
    printf(1, "File moved successfully from\n");
  }
    exit();
}
```

gmis.c > main(int, char * [])

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"


int main(int argc, char *argv[])
{
    int pid = getpid();
    printf(1, "Current PID: %d\n", pid);
    int most_invoked = get_most_invoked_syscall(pid);
    printf(1, "Most invoked syscall : %d\n", most_invoked);
    exit();
}
```

listallproc.c > main(int, char * [])

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"


int main(int argc, char *argv[])
{
    list_all_processes();
    exit();
}
```

syscallsort.c > main(int, char * [])

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc,char* argv[])
{
    if(argc != 2)
    {
        printf(2, "Error Usage\n");
    }
    int pid = atoi(argv[1]);

    if(sort_syscalls(pid) == -1){
        printf(2, "process not found\n");
    }
    exit();
}
```

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _Pid\
    _mvfile\
    _gmis\
    _listallproc\
    _syscallsort\
```

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    Pid.c mvfile.c gmis.c listallproc.c syscallsort.c\
    printf.c umalloc.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

```c
62
63  int sort_syscalls(int pid);
64  int get_most_invoked_syscall(int pid);
65  int list_all_processes(void);
66
```

```c
23  char* sbrk(int);
24  int sleep(int);
25  int uptime(void);
26  int move_file(char*,char*);
27  int sort_syscalls(int);
28  int get_most_invoked_syscall(int pid);
29  int list_all_processes(void);
```

```c
[SYS_close]   sys_close,
[SYS_move_file] sys_move_file,
[SYS_sort_syscalls] sys_sort_syscalls,
[SYS_get_most_invoked_syscall]  sys_get_most_invoked_syscall,
[SYS_list_all_processes]        sys_list_all_processes,
};
```

```c
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->invoke_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;
  }
}
```

```c
22  #define SYS_close  21
23  #define SYS_move_file 22
24  #define SYS_sort_syscalls 23
25  #define SYS_get_most_invoked_syscall  24
26  #define SYS_list_all_processes        25
```