

# بسم الله الرحمن الرحيم

## پروژه دوم درس آزمایشگاه سیستم عامل دکتر کارگهی

محمد امین توانایی - ۸۱۰۱۰۱۳۹۶

سید علی تهامی - ۸۱۰۱۰۱۳۹۷

مهدی وجهی - ۸۱۰۱۰۱۵۵۸

:Question 1

The ULIB variable contains files that implement the wrapper functions that are widely used in user programs and include such system calls, so that writing them as system calls in the user program is very complicated.

Some of these functions are like converting a string to a number or allocating dynamic memory or writing on the screen. Also, this variable contains a file that provides different system calls to the user so that he can use them while writing the program. Also this Programs can provide more understandable feedback to the user when there is an error.

If the implementation of the system calls are different in different systems, by providing a higher single layer such as wrapper functions, you can make yourself independent from the system and the libraries manage this difference internally.

Also, these libraries can have a significant impact on the performance of the program. Usually, overhead is imposed due to the addition of a new layer, but on the other hand, they can be implemented in a way that optimally uses different system calls and increases performance.

:Question 2

1. Virtual system call: Not every request really needs a context switch and for commonly used functions where shared data is accessed there are two mechanisms. One is the legacy vsyscall memory mapping which contains simple pseudo-syscall functions like `.gettimeofday()` and a shared memory region which holds the data to return.

2. Signals : A process can register a handler function to replace the default action to e.g. handle a Ctrl-C SIGABRT on the terminal. The handler is asynchronously invoked independently from the normal program execution. A signal mask can block signals during handler execution.

3. Sockets : Messages through sockets offer flexibility and portability compared to direct syscalls or shared memory. Concerning sockets there are various families available and it is to be distinguished between datagram and stream mode which gives a guaranteed ordering without the notion of packages.

## سوال ۳

خیر، سیستم عامل برای محافظت و امنیت از اجرایی تمامی سیستم کال ها با سطح دسترسی یوزر مخالفت می کند و اکسپشن مناسبی را برمی گرداند زیرا در صورت عدم وجود این موضوع می تواند مشکلاتی برای سیستم عامل، خود برنامه و سایر برنامه ها ایجاد کند.

## سوال ۴

وقتی می خواهیم سطح دسترسی را تغییر بدیم لازم است استک پوینتر را ذخیره کنیم که وقتی به برنامه بر می گردیم سر استک را بدانیم و همچنین لازم است سطح برنامه را هم ذخیره کنیم که در هنگام برگشت مجدد به همان مد برگردیم و واضح است وقتی این کار رو نمی کنیم لازم نیست این اطلاعات را ذخیره کنیم.

## سوال ۵

تابع برای استخراج آرگومان ها به صورت زیر است:

- argint(): برای استخراج یک عدد صحیح.
- argptr(): برای استخراج یک اشاره گر و بررسی بازه آدرس.
- argstr(): برای استخراج یک رشته و بررسی بازه آدرس.

هر کدام از این موارد بررسی می کنند که آدرس درخواستی در فضای حافظه باشد و دسترسی به آن مجاز باشد سپس مقادیر را در متغیر پاس داده شده قرار می دهد. برای argstr هم بازه درخواستی بررسی می شود که از محدوده برنامه خارج نشود. این موضوع از آن جهت می تواند خطرناک باشد که ما وقتی این تابع را صدا می کنیم در مد کرنل هستیم و اگر آدرس در محدوده برنامه هم نباشد اجازه دسترسی داریم و این موضوع می تواند مشکلات حفاظتی و امنیتی ایجاد کند. مثلاً در تابع sys\_read اگر این موضوع کنترل نشود کاربر می تواند با دادن مقدار بزرگی به عنوان طول رشته مقادیر موجود در باقی خانه های حافظه که مربوط به آن برنامه نیست را هم بخواند.

## اضافه کردن سیستم کال ها

1. Move\_file

a. اضافه کردن عدد مربوط به سیستم کال در فایل `syscall.h`

```
...
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_move_file 22
24
```

b. متصل کردن سیستم کال به تابع سیستم کال مدنظر در `syscall.c`

```
85 extern int sys_chdir(void);
86 extern int sys_close(void);
87 extern int sys_dup(void);
88 extern int sys_exec(void);
89 extern int sys_exit(void);
90 extern int sys_fork(void);
91 extern int sys_fstat(void);
92 extern int sys_getpid(void);
93 extern int sys_kill(void);
94 extern int sys_link(void);
95 extern int sys_mkdir(void);
96 extern int sys_mknod(void);
97 extern int sys_open(void);
98 extern int sys_pipe(void);
99 extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_move_file(void);
107
108 static int (*syscalls[])(void) = {
109     [SYS_fork] sys_fork,
110     [SYS_exit] sys_exit,
111     [SYS_wait] sys_wait,
112     [SYS_pipe] sys_pipe,
113     [SYS_read] sys_read,
114     [SYS_kill] sys_kill,
115     [SYS_exec] sys_exec,
116     [SYS_fstat] sys_fstat,
117     [SYS_chdir] sys_chdir,
118     [SYS_dup] sys_dup,
119     [SYS_getpid] sys_getpid,
120     [SYS_sbrk] sys_sbrk,
121     [SYS_sleep] sys_sleep,
122     [SYS_uptime] sys_uptime,
123     [SYS_open] sys_open,
124     [SYS_write] sys_write,
125     [SYS_mknod] sys_mknod,
126     [SYS_unlink] sys_unlink,
127     [SYS_link] sys_link,
128     [SYS_mkdir] sys_mkdir,
129     [SYS_close] sys_close,
130     [SYS_move_file] sys_move_file,
131 }
```

c. اضافه کردن تابع مربوط استفاده یوزر در user.h

```

100, 3 hours ago | 7 authors (pat and others)
1 struct stat;
2 struct rtcdat;
3
4 // system calls
5 int fork(void);
6 int exit(void) __attribute__((noreturn));
7 int wait(void);
8 int pipe(int*);
9 int write(int, const void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(const char*, int);
15 int mknod(const char*, short, short);
16 int unlink(const char*);
17 int fstat(int fd, struct stat*);
18 int link(const char*, const char*);
19 int mkdir(const char*);
20 int chdir(const char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int move_file(char*,char*);
27
28 // ulib.c
29 int stat(const char*, struct stat*);
30 char* strcpy(char*, const char*);
31 void *memmove(void*, const void*, int);
32 char* strchr(const char*, char c);
33 int strcmp(const char*, const char*);
34 void printf(int, const char*, ...);
35 char* gets(char*, int max);
36 uint strlen(const char*);
37 void* memset(void*, int, uint);
38 void* malloc(uint);
39 void free(void*);
40 int atoi(const char*);
41

```

d. تنظیم usys.S برای هماهنگی بین سیستم کال بدون آرگومان سطح سیستم و تابع سطح یوزر

```

You, 5 hours ago | 3 authors (rsc and others)
1  #include "syscall.h"
2  #include "traps.h"
3
4  #define SYSCALL(name) \
5      .globl name; \
6      name: \
7          movl $SYS_ ## name, %eax; \
8          int $T_SYSCALL; \
9          ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
17 SYSCALL(close)
18 SYSCALL(kill)
19 SYSCALL(exec)
20 SYSCALL(open)
21 SYSCALL(mknod)
22 SYSCALL(unlink)
23 SYSCALL(fstat)
24 SYSCALL(link)
25 SYSCALL(mkdir)
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL[move_file]
33

```

e. اضافه کردن تابع چک کننده عملکرد سیستم کال

```

1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #include "fcntl.h"
5
6
7  int main(int argc, char *argv[]) {
8
9
10     if (move_file(argv[1], argv[2]) < 0) {
11         printf(2, "move: failed to mov\n");
12     } else {
13         printf(1, "File moved successfully from\n");
14     }
15     exit();
16
17 }

```

f. اضافه کردن تعریق تابع `sys_move_file` در `sysfile.c`

برای این کار لازم است تا آییی و نود فایل مبدا را پیدا و دایرکتوری پدر آن را معلوم کنیم سپس لینک بین این فایل و پدرش را قطع کرده و آن را به پدر جدیدش لینک کنیم.

2. `Sort_syscalls`

a. مراحل a, b, c, d, e را همانند `move_file` انجام می‌دهیم.

b. برای اینکه بتوانیم تعداد فراخوانی های سیستم کال ها را برای یک پردازش بدانیم لازم است یک آرایه داشته باشیم که ایندکس آن شماره سیستم کال و مقدار آن تعداد استفاده از آن پردازش است. این آرایه را به استراکچر پردازش در فایل `proc.h` اضافه می‌کنیم.

```

C proch > ...
28 struct context {
29     uint esp;
30     uint eip;
31 };
32
33 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
34
35 // Per-process state
36 You, 2 hours ago | 6 authors (kolya and others)
37 struct proc {
38     uint sz; // Size of process memory (bytes)
39     pde_t* pgdir; // Page table
40     char *kstack; // Bottom of kernel stack for this process
41     enum procstate state; // Process state
42     int pid; // Process ID
43     struct proc *parent; // Parent process
44     struct trapframe *tf; // Trap frame for current syscall
45     struct context *context; // switch() here to run process
46     void *chan; // If non-zero, sleeping on chan
47     int killed; // If non-zero, have been killed
48     struct file *ofile[NOFILE]; // Open files
49     struct inode *cwd; // Current directory
50     char name[16]; // Process name (debugging)
51     int syscall_count[NUM_SYSCALLS];
52 };
53
54 int sort_syscalls(int pid);
55
56 // Process memory is laid out contiguously, low addresses first:
57 // text
58 // original data and bss
59 // fixed-size stack
60 // expandable heap
61
62

```

c. همچنین هر زمان که پردازش ای بوجود آمد باید این آرایه مقدار دهی اولیه شود این و هر بار که سیستم کالی صدا زده شد مقدار مربوط به خانه آن یکی اضافه شود. مقدار دهی اولیه در فایل

proc.c و آپدیت آن در فایل syscall.c انجام شده است.

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

    num = curproc->tf->eax;
    if (num > 0 && num < NUM_SYSCALLS) {
        curproc->syscall_count[num-1]++;

        if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
            curproc->tf->eax = syscalls[num]();
        } else {
            cprintf("%d %s: unknown sys call %d\n",
                    curproc->pid, curproc->name, num);
            curproc->tf->eax = -1;
        }
    }
}
```

d. خود کد سیستم کال نیز به این صورت عمل میکند که صرفا این آرایه را پرینت میکند.



## بررسی فراخوانی سیستمی با gdb

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

int main(int argc, char* argv[]) {
    int pid = getpid();
    printf(1, "Process ID: %d\n", pid);
    exit();
}
```

```
mvajjhi@mahdi-laptop xv6-public/ca2 (master) » 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

To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/home/mvajjhi/.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) break syscall
Breakpoint 1 at 0x80104a40: file syscall.c, line 135.
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint     keep y   0x80104a40 in syscall at syscall.c:135
(gdb)
```

همانطور که مشاهده می کنید با دستور مربوط به دیباگ برنامه را اجرا کردیم سپس در جی دی بی به آن وصل شدیم برنامه را ادامه دادیم تا تنظیمات اولیه انجام شود و سپس برنامه را متوقف کردیم و یک نقطه توقف در syscall تنظیم کردیم. سپس برنامه ی get را اجرا کردیم و برنامه در syscall متوقف شد.

```
syscall.c
123 [SYS_open] sys_open,
124 [SYS_write] sys_write,
125 [SYS_mknod] sys_mknod,
126 [SYS_unlink] sys_unlink,
127 [SYS_link] sys_link,
128 [SYS_mkdir] sys_mkdir,
129 [SYS_close] sys_close,
130 [SYS_move_file] sys_move_file,
131 ];
132
133 void
134 syscall(void)

Thread 1.1 (src) In: syscall L135 PC: 0x80104a40
(gdb) bt
#0  syscall () at syscall.c:135
#1  0x80105b51 in trap (tf=0x8dfffefb4) at trap.c:43
#2  0x80105893 in alltraps () at trapasm.S:20
#3  0x8dfffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

همانطور که مشاهده می کنید پشته صدا شدن تابع را نشان می دهد که تابع فعلی در بالای استک و برنامه سطح کاربر get در آخر پشته قرار دارد. همچنین می بینیم که آدرس دستوری که از آن پرش شده و تابع جدیدی صدا شده نمایش داده می شود همچنین می دانیم که در این استک متغیرهای محلی، آرگومان ها و موارد لازم برای اجرا تابع در زمان برگشت ذخیره می شود. همانطور هم که در توضیحات فایل پروژه داشتم ابتدا trapasm اجرا شده و بعد به وسیله آن ابتدا trap frame ساخته می شود و سپس آدرس برگشت روی استک قرار می

گیرد و تابع trap صدا می شود. همچنین در این عملیات سطح دسترسی هم به حالت هسته می رود. سپس در این تابع فریم trap بررسی می شود و سپس syscall فراخوانی می شود. سپس در این تابع بردار فراخوانی های سیستم موجود است و با پیدا کردن فراخوانی مربوطه آن را صدا می کند.

```

trap.c
38 {
39     tf(tf->trapno == T_SYSCALL){
40         tf(myproc()->killed)
41         exit();
42         myproc()->tf = tf;
43     }
44     sysctl();
45     tf(myproc()->killed)
46     exit();
47     return;
48 }
49 switch(tf->trapno){

```

```

remote Thread 1.1 (src) In: trap L43 PC: 0x80105b51
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1 0x80105b51 in trap (tf=0x8dffe4b4) at trap.c:43
(gdb) p tf.eax
$4 = 5
(gdb)

```

طبق خواسته سوال دستور down را اجرا می کنیم و همانطور که مشاهده می کنید اجرا نمی شود. این دستور در واقع برای پیمایش استک تابع هست و مثلاً اگر در حال مشاهده تابع خانه ۱ استک باشیم با اجرای این دستور به تابع خانه ۰ استک می رویم همچنین دستور دیگری هم هست به نام up که دقیقاً برعکس آن کار می کند و در اینجا باید از آن استفاده کنیم و وقتی به تابع trap رفتیم با استفاده از دستور چاپ مقدار tf.eax را نمایش می دهیم که برابر با ۵ است.

```

$12 = 16 (gdb)
$11 = 11 (gdb)
$10 = 7 (gdb)
$9 = 12 (gdb)
$8 = 3 (gdb)
$7 = 1 (gdb)
$6 = 5 (gdb)
$5 = 5 (gdb)

$ get
Process ID: 3
$27 = 2 (gdb)
$26 = 16 (gdb)
$15 = 16 (gdb)
$14 = 16 (gdb)
$13 = 16 (gdb)

```

```

// System call numbers (syscall.h)
#define SYS_fork 1
#define SYS_exit 2
#define SYS_wait 3
#define SYS_read 5
#define SYS_exec 7
#define SYS_getpid 11
#define SYS_sbrk 12
#define SYS_write 16

```

شماره فراخوانی و مقدار آن در هر تصویر مشخص هست و مقادیر تعریف شده هم قرار داده شده. دلیل این موضوع که در ابتدا ۱۱ نیست واضح است بعد از دریافت فرمان ما سیستم باید ابتدا ورودی را بخواند و سپس برنامه را اجرا کند و منتظر خروجی آن بماند و سپس برنامه حافظه مورد نیاز رو میگیرد و فایل آن اجرا می شود. در اینجاست که برنامه id خود را می گیرد و در قسمت های بعد هم خروجی را چاپ می کند و در آخر خارج می شود.