

به نام خدا



دانشگاه تهران  
پردیس دانشکده‌های فنی  
دانشکده برق و کامپیوتر



## سیستم عامل

### گزارش پروژه آزمایشگاه شماره ۲

محمدامین باقرشاهی – ۸۱۰۱۹۷۴۶۴

سجاد علی‌زاده – ۸۱۰۱۹۷۵۴۷

سارینا همدانی – ۸۱۰۱۹۷۶۰۶

## پرسش اول

همانطور که ذکر شد برخی از برنامه‌ها نیاز به عملیات ممتاز دارند. کتابخانه‌های ذکر شده از system call های زیر استفاده می‌کنند:

محل کد	نام system call	توضیح
ulib.c:59	read()	برای خواندن یک کاراکتر از ورودی استاندارد استفاده شده است
ulib.c:76	open()	برای به دست آوردن file descriptor از روی نام فایل استفاده شده است
ulib.c:79	fstat()	برای به دست آوردن اطلاعات در مورد file descriptor داده شده استفاده شده است
ulib.c:80	close()	برای بستن file descriptor باز شده از سوی کرنل استفاده شده است
umalloc.c:54	sbrk()	برای دریافت حافظه استفاده شده است
printf.c:8	write()	برای نوشتن یک کاراکتر بر روی خروجی استاندارد استفاده شده است

## پرسش دوم

علاوه بر system call ها دو روش دیگر برای دسترسی سطح کاربر به هسته وجود دارد.

### فایل سیستم های مجازی:

به منظور تبادل داده بین فضای کاربر و فضای کرنل لینوکس چندین فایل سیستم بر پایه RAM را فراهم می‌کند. خود این اینترفیس‌ها مبتنی بر فایل‌ها هستند. معمولاً یک فایل تنها یک مقدار را نمایندگی میکند اما ممکن است مجموعه‌ای از مقادیر را نمایندگی کند. فضای کاربر میتواند با استفاده از توابع استاندارد read() و write() به این مقادیر دسترسی داشته باشند. فایل سیستم‌های sys و prog نمونه‌هایی از این فایل سیستم‌ها هستند.

### مکانیزم های مبتنی بر سوکت:

سوکت به کرنل اجازه می‌دهد تا اعلان(notification)هایی به سطح کاربر ارسال کند. این برخلاف مکانیزم فایل سیستم است. جایی که کرنل میتواند یک فایل را تغییر دهد ولی برنامه کاربر تا وقتی دسترسی به آن فایل را انتخاب نکند از تغییرات آگاهی ندارد. مکانیزم های مبتنی بر سوکت به برنامه‌ها

اجازه می‌دهد تا بر یک سوکت گوش کنند و کرنل در هر زمانی میتواند برای آنها پیام ارسال کند. این اتفاق منجر به مکانیزم ارتباطی میشود که در آن فضای هسته و فضای کرنل دارای پارنترهای یکسان هستند.

### پرسش سوم

خیر. زیرا سطح دسترسی کاربر پایین‌ترین سطح دسترسی است و برای فعال کردن trap باید سطح دسترسی بالاتری داشته باشیم. اگر در غیر این صورت می‌بود میتوانستیم هر عملیات فضای کرنل را در فضای کاربر انجام دهیم که این ناقض سیاستهای حفاظت است.

### پرسش چهارم

وضعیت process در حال اجرا ذخیره می‌گردد تا بتوان در سطح کاربر این وضعیت را بازیابی نمود. اگر تغییر سطح دسترسی نداشته باشیم این بازیابی بدون نیاز به ss و esp در stack انجام خواهد شد. اما اگر تغییر سطح دسترسی داشته باشیم هنگام تغییر سطح حفاظت از فضای کاربر به فضای هسته، هسته نباید از stack پرده کاربر استفاده کند زیرا ممکن است valid نباشد و یا حتی process کاربر مخرب باشد و یا حاوی خطا باشد. در اصل در این حالت نیاز داریم این مقادیر نیز روی stack قرار گیرند تا در فضای کاربر مشکلی برای دسترسی به آنها نداشته باشیم.

### **int argint(int, int\*)**

پارامتر اول شماره آرگومانی است که می‌خواهیم از stack بخوانیم و پارامتر دوم آدرس یک متغیر integer که آرگومان خوانده شده از stack در آن ریخته می‌شود. اگر این عملیات موفقیت آمیز نبود منفی یک به عنوان مقدار بازگشتی return می‌شود.

### **int argstr(int, char\*\*)**

پارامتر اول مانند حالت قبل شماره آرگومانی است که می‌خواهیم از stack بخوانیم و پارامتر دوم آدرس یک char\* است که آرگومان خوانده شده در آن ذخیره می‌شود.

### **int argptr(int, char\*\*, int)**

پارامتر اول شماره آرگومانی است که می‌خواهیم از stack بخوانیم و پارامتر دوم یک pointer است که آرگومان خوانده شده در آن ریخته می‌شود و پارامتر سوم اندازه چیزی است که از استک خوانده می‌شود.

### **دلیل بررسی بازه آدرس‌ها**

زیرا اگر آدرس داده شده از محدوده process داده شده فراتر رود ممکن است اطلاعاتی به اشتباه از سایر process‌ها fetch کند که بدیهی است باعث مشکلاتی می‌شود. یکی از مشکلاتی که ممکن است رخ دهد system call‌های ناخواسته است. در مورد sys\_read ممکن است خواندن از file descriptor اشتباه صورت گیرد زیرا از اطلاعات process دیگری استفاده می‌شود و در این صورت سیستم با مشکل مواجه خواهد شد.

## پرسش شش و هفت

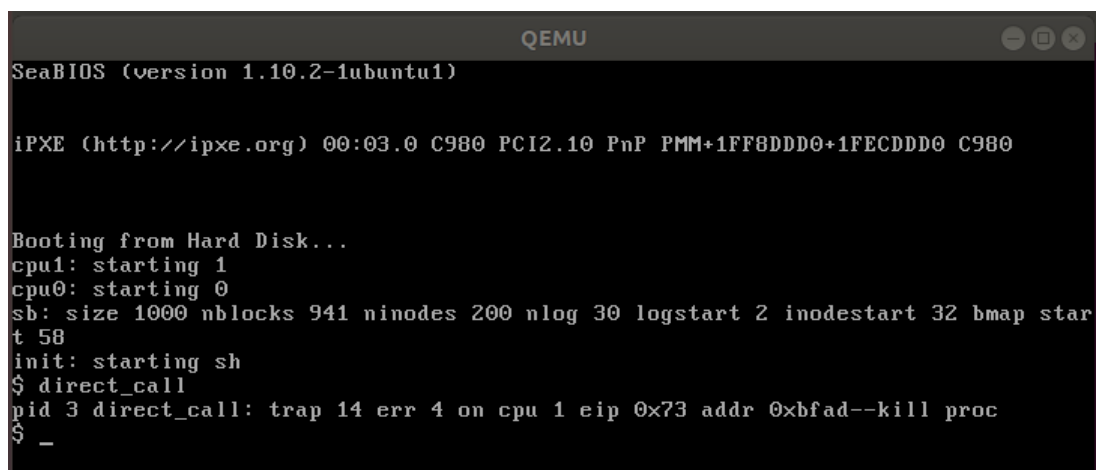
ابتدا با استفاده از دستور objdump و آپشن -d آدرس فراخوانی سیستمی sys\_getpid() را به دست می‌آوریم که نتیجه در پایین قابل مشاهده است:

```
00000070 <sys_getpid>:
70: 55          push    %ebp
71: 89 e5       mov     %esp,%ebp
73: 83 ec 08    sub     $0x8,%esp
76: e8 fc ff ff call    77 <sys_getpid+0x7>
7b: 8b 40 10    mov     0x10(%eax),%eax
7e: c9         leave
7f: c3         ret
```

سپس در یک فایل به نام direct\_call.c یک برنامه سطح کاربر نوشته که کارکرد آن مطابق دستورالعمل آنست که یک اشاره‌گر به تابع تعریف کنیم که با نوع پارامترها و مقدار بازگشتی sys\_getpid() مطابقت داشته باشد و مقدار آن را برابر آدرس فراخوانی سیستمی در کد هسته قرار دهیم. به عبارتی یک اشاره‌گر با تابع با پارامتر خالی (void) و مقدار بازگشتی از نوع int تعریف کرده و مقدار آن را پس از casting برابر 0x00000070 یعنی آدرس sys\_getpid() در کد هسته قرار می‌دهیم.

۶) سپس این اشاره‌گر را فراخوانی می‌کنیم. پس از ایجاد تغییرات لازم در فایل makefile دستور مربوط به این برنامه سطح کاربر (به نام direct\_call) را فراخوانی می‌کنیم. با خطایی با محتوای زیر مواجه می‌شویم:

pid 3 direct\_call: trap 14 err 4 on cpu 1 eip 0x73 addr 0xbfad--kill proc



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8DDDD+1FECDDDD C980

Booting from Hard Disk...
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
$ direct_call
pid 3 direct_call: trap 14 err 4 on cpu 1 eip 0x73 addr 0xbfad--kill proc
$ _
```

با مراجعه به فایل trap.c متوجه می‌شویم که علت بروز این خطا رفتار نادرستی است که از یک پردازنده در فضای کاربر سر بزند. درواقع در این حالت، کاربر دست به کاری زده که از حوزه‌ی کاری او

خارج بوده است. به عنوان مثال، ما در این برنامه، در حالتی که در فضای کاربر قرار داشتیم یک فراخوانی سیستمی را که ملزم به اجرا در مد هسته است را فراخوانی کردیم و در نتیجه یک `trap` رخ می‌دهد که در اینجا علت این `trap`، بروز خطا بوده است. با مراجعه به کد مربوطه شماره `trap` و خطا همچنین محتوای کنونی اشاره‌گر دستورالعمل از `trapframe` استخراج شده و نمایش داده می‌شود.

۷) هنگام فراخوانی یک فراخوانی سیستمی، حالت کاربری تغییر یافته و به یک روتین در قسمت مربوط به هسته در حافظه پرش می‌کند. هسته باید از صحت پارامترهای فراخوانی سیستمی اطمینان حاصل کند و اجازه‌ی دسترسی پردازشی کنونی را نیز چک کند. این کارها باید در حالت کاربری ممتاز انجام شوند و نمی‌توان مانند یک برنامه سطح کاربر و همانند یک تابع ساده آن‌ها را فراخوانی کرد.

در واقع، توابع کتابخانه‌ای همچون `getpid()` به عنوان یک `wrapper` عمل می‌کند که چندین گام ابتدایی را اجرا می‌کند و سپس دستور `trap` مربوط به یک فراخوانی سیستمی مانند `sys_getpid()` را اجرایی می‌کند.

## اضافه کردن `reverse_number`

برای رسیدن به این هدف نیاز است چندین گام را طی کنیم.

گام اول: فایل `syscall.h` شامل اعدادی است که برای دسترسی به سیستم‌کال‌ها از آن استفاده می‌شود از آنجایی که سیستم عامل `xv6` شش بیست و یک سیستم کال از پیش تعریف شده دارد عدد ۲۵ را برای سیستم کال جدید در نظر میگیریم.

```
// 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
#define SYS_get_children 22
#define SYS_get_grandchildren 23
#define SYS_trace_syscalls 24
#define SYS_reverse_number 25
```

گام دوم: در فایل syscall.c دو تغییر ایجاد میکنیم که در شکلهای زیر قابل مشاهده است.

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_get_children(void);
extern int sys_get_grandchildren(void);
extern int sys_trace_syscalls(void);
extern int sys_reverse_number(void);
```



```
static int (*syscalls[])(void) = {
[SYS_fork]      sys_fork,
[SYS_exit]      sys_exit,
[SYS_wait]      sys_wait,
[SYS_pipe]      sys_pipe,
[SYS_read]      sys_read,
[SYS_kill]      sys_kill,
[SYS_exec]      sys_exec,
[SYS_fstat]     sys_fstat,
[SYS_chdir]     sys_chdir,
[SYS_dup]       sys_dup,
[SYS_getpid]    sys_getpid,
[SYS_sbrk]      sys_sbrk,
[SYS_sleep]     sys_sleep,
[SYS_uptime]    sys_uptime,
[SYS_open]      sys_open,
[SYS_write]     sys_write,
[SYS_mknod]     sys_mknod,
[SYS_unlink]    sys_unlink,
[SYS_link]      sys_link,
[SYS_mkdir]     sys_mkdir,
[SYS_close]     sys_close,
[SYS_get_children] sys_get_children,
[SYS_get_grandchildren] sys_get_grandchildren,
[SYS_trace_syscalls] sys_trace_syscalls,
[SYS_reverse_number] sys_reverse_number,
};
```

همانطور که مشاهده میشود دو خط

`extern in sys_reverse_number(void)` و `sys_reverse_number [SYS_reverse_number]` به این

فایل اضافه شده است. این فایل در اصل شماره سیستم کال را به فرایند متناظر آن تصویر میکند.

گام سوم: در فایل `ulib.c` تابعی تعریف میکنیم که ابتدا محتوای رجیستر `edx` را در یک متغیر ذخیره کند و بعد از آن آرگومان خود را در این رجیستر قرار دهد. بعد از آن شماره سیستم کال `reverse_number` که ۲۵ است را در رجیستر مخصوص شماره سیستم کال یا `eax` قرار دهد و سپس با دستور `int 64` این سیستم کال را اجرا نماید. پس از آن مقدار رجیستر `edx` بازنشانی میشود

```
int
reverse_number(int num)
{
    uint temp;
    asm("mov %%edx, %0" : "=r" (temp));
    asm("mov %0, %%edx" : : "r" (num));
    asm("mov $25, %%eax" :);
    asm("int $64");
    asm("mov %0, %%edx" : : "r" (temp));
    return 0;
}
```

گام چهارم: در فایل `sysproc.c` پیاده سازی این سیستم کال را انجام می‌دهیم به این صورت که در ابتدا عدد از روی رجیستر `edx` خوانده میشود و سپس عملیات لازم بر روی آن انجام میشود. توجه کنید که چاپ کردن نتیجه در سطح کرنل انجام میشود.

```
int
sys_reverse_number(void)
{
    int num = myproc()->tf->edx;
    while (num != 0)
    {
        cprintf("%d", num % 10);
        num = num / 10;
    }
    cprintf("\n");
    return 0;
}
```

گام آخر:

برای تست کردن این سیستم کال دستور `reverse_number number` به سیستم عامل اضافه شده است که آرگومان ورودی را برعکس چاپ میکند.

## فراخوانی سیستمی trace\_syscalls

```
10 struct {
11     struct spinlock lock;
12     struct proc proc[NPROC];
13     int trace_syscalls_state;
14     ptable;
15 }
```

برای پیاده سازی این فراخوانی سیستمی یک trace\_syscalls\_state در ptable نگه می‌داریم که ۱ بودن آن به معنی این است که اطلاعات مربوط به تعداد فراخوانی های سیستمی پردازش ها باید update و چاپ شود.

اطلاعات مربوط به تعداد فراخوانی های سیستمی صدا زده شده برای هر پردازش در یک struct به نام tr\_syscalls نگه داری می شود. این struct در فایل proc.h اضافه شده است. در proc که اطلاعات مربوط به هر پردازش نگه داری می شود یک struct tr\_syscalls نگه داری می کنیم.

```
39 struct tr_syscalls {
40     int fork_number;
41     int exit_number;
42     int wait_number;
43     int pipe_number;
44     int read_number;
45     int kill_number;
46     int exec_number;
47     int fstat_number;
48     int chdir_number;
49     int dup_number;
50     int getpid_number;
51     int sbrk_number;
52     int sleep_number;
53     int uptime_number;
54     int open_number;
55     int write_number;
56     int mknod_number;
57     int unlink_number;
58     int link_number;
59     int mkdir_number;
60     int close_number;
61     int get_children_number;
62     int get_grandchildren_number;
63     int trace_syscalls_number;
64     int reverse_number_number;
65 }
```

```
69 // Per-process state
70 struct proc {
71     uint sz; // Size of process memory (bytes)
72     pde_t* pgdir; // Page table
73     char *kstack; // Bottom of kernel stack for this
74     enum procstate state; // Process state
75     int pid; // Process ID
76     struct proc *parent; // Parent process
77     struct trapframe *tf; // Trap frame for current syscall
78     struct context *context; // switch() here to run process
79     void *chan; // If non-zero, sleeping on chan
80     int killed; // If non-zero, have been killed
81     struct file *ofile[NOFILE]; // Open files
82     struct inode *cwd; // Current directory
83     char name[16]; // Process name (debugging)
84     struct tr_syscalls syscalls; // Traced syscalls
85 };
86
```

```
10 int
11 main(void)
12 {
13     int pid, wpid;
14
15     if(open("console", O_RDWR) < 0){
16         mknod("console", 1, 1);
17         open("console", O_RDWR);
18     }
19     dup(0); // stdout
20     dup(0); // stderr
21
22     for(;;){
23         if(fork() == 0)
24         {
25             trace_syscalls(0);
26             while(1);
27             exit();
28         }
29         else
30         {
31             printf(1, "init: starting sh\n");
32             printf(1, "Group Members:\n");
33             printf(1, "Sajjad Alizadeh\n");
34             printf(1, "Amin Bagher Shahi\n");
35             printf(1, "Sarina Hamedani\n");
36             pid = fork();
37             if(pid < 0){
38                 printf(1, "init: fork failed\n");
39                 exit();
40             }
41             if(pid == 0){
42                 exec("sh", argv);
43                 printf(1, "init: exec sh failed\n");
44                 exit();
45             }
46             while((wpid=wait()) >= 0 && wpid != pid)
47                 printf(1, "zombie\n");
48         }
49     }
50 }
```

در ابتدا هنگامی که سیستم boot می شود پس از ایجاد پردازش Init پردازش دیگری ایجاد می شود که همیشه فعال است و در صورتی که state برابر ۱ باشد به چاپ اطلاعات جمع شده می پردازد. درواقع در این پردازش یکبار فراخوانی سیستمی trace\_syscalls را با مقدار ۰ صدا می زند(در ابتدا state برابر ۰ است) و در این سیستم کال در حلقه بی نهایت می افتد. شکل روبرو init.c را نشان می دهد:

هنگامی که پردازش های دیگر فراخوانی سیستمی `trace_syscalls` را صدا بزنند پس از خواندن ورودی از `stack` در صورتی که ورودی ۰ باشد مقدار `trace_syscalls_state` در `ptable` به ۰ تغییر می کند. این کار توسط تابع `set_trace_syscalls_state` انجام می شود که در `proc.c` تعریف شده است. همچنین اطلاعات جمع شده درمورد تعداد فراخوانی های سیستمی پردازش ها پاک می شود. این کار توسط تابع `reset_trace_syscalls` انجام می شود که در فایل `proc.c` تعریف شده است.

```

// Set trace
void
set_trace_syscalls_state(int state)
{
    acquire(&ptable.lock);
    ptable.trace_syscalls_state = state;
    release(&ptable.lock);
}

```

```

725 void
726 reset_trace_syscalls()
727 {
728     cprintf("----- \n");
729     cprintf("Resetting traced syscalls.\n");
730     cprintf("-----\n");
731     acquire(&ptable.lock);
732     for (int i = 0; i < NPROC; i++)
733     {
734         ptable.proc[i].syscalls.fork_number = 0;
735         ptable.proc[i].syscalls.exit_number = 0;
736         ptable.proc[i].syscalls.wait_number = 0;
737         ptable.proc[i].syscalls.pipe_number = 0;
738         ptable.proc[i].syscalls.read_number = 0;
739         ptable.proc[i].syscalls.kill_number = 0;
740         ptable.proc[i].syscalls.exec_number = 0;
741         ptable.proc[i].syscalls.fstat_number = 0;
742         ptable.proc[i].syscalls.chdir_number = 0;
743         ptable.proc[i].syscalls.dup_number = 0;
744         ptable.proc[i].syscalls.getpid_number = 0;
745         ptable.proc[i].syscalls.sbrk_number = 0;
746         ptable.proc[i].syscalls.sleep_number = 0;
747         ptable.proc[i].syscalls.uptime_number = 0;
748         ptable.proc[i].syscalls.open_number = 0;
749         ptable.proc[i].syscalls.write_number = 0;
750         ptable.proc[i].syscalls.mknod_number = 0;
751         ptable.proc[i].syscalls.unlink_number = 0;
752         ptable.proc[i].syscalls.link_number = 0;
753         ptable.proc[i].syscalls.mkdir_number = 0;
754         ptable.proc[i].syscalls.close_number = 0;
755         ptable.proc[i].syscalls.get_children_number = 0;
756         ptable.proc[i].syscalls.get_grandchildren_number = 0;
757         ptable.proc[i].syscalls.trace_syscalls_number = 0;
758         ptable.proc[i].syscalls.reverse_number_number = 0;
759     }
760     release(&ptable.lock);
761 }

```

در صورتی که ورودی ۱ باشد مقدار `state` در `ptable` به ۱ تغییر می کند. این کار توسط تابع `set_trace_syscalls_state` انجام می شود که در `proc.c` تعریف شده است. با ۱ شدن `state` هر زمانی که یک فراخوانی سیستمی در پردازش های مختلف صدا زده شود `struct tr_syscalls` مربوط به آن پردازش به روز می شود. این کار توسط تابع `update_syscalls` انجام می شود که در تابع `syscall` که در `syscall.c` قرار دارد صدا زده شده است. چرا که هرزمان که یک فراخوانی سیستمی صدا زده می شود در صورت ۱ بودن `state` اطلاعات باید به روز شوند. تعریف تابع `update_syscalls` در `proc.c` قرار دارد.

```

//////////////////////////////////// Set trace
void
set_trace_syscalls_state(int state)
{
    acquire(&ptable.lock);
    ptable.trace_syscalls_state = state;
    release(&ptable.lock);
}

```

```

139 void
140 syscall(void)
141 {
142     int num;
143     struct proc *curproc = myproc();
144
145     num = curproc->tf->eax;
146     update_syscalls(num);
147     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
148         curproc->tf->eax = syscalls[num]();
149     } else {
150         cprintf("%d %s: unknown sys call %d\n",
151             curproc->pid, curproc->name, num);
152         curproc->tf->eax = -1;
153     }
154 }
155

```

```

//////////////////////////////////// Update syscalls
void
update_syscalls(int syscall_num)
{
    if (ptable.trace_syscalls_state == 1)
    {
        acquire(&ptable.lock);
        switch (syscall_num)
        {
            case 1:
                myproc()->syscalls.fork_number += 1;
                break;
            case 2:
                myproc()->syscalls.exit_number += 1;
                break;
            case 3:
                myproc()->syscalls.wait_number += 1;
                break;
            case 4:
                myproc()->syscalls.pipe_number += 1;
                break;
            case 5:
                myproc()->syscalls.read_number += 1;
                break;
            case 6:
                myproc()->syscalls.kill_number += 1;
                break;
            case 7:
                myproc()->syscalls.exec_number += 1;
                break;
            case 8:
                myproc()->syscalls.fstat_number += 1;
                break;
            case 9:
                myproc()->syscalls.chdir_number += 1;
                break;
            case 10:
                myproc()->syscalls.dup_number += 1;
                break;
            case 11:
                myproc()->syscalls.getpid_number += 1;
                break;
            case 12:
                myproc()->syscalls.sbrk_number += 1;
                break;
        }
    }
}

```

## تغییرات دیگر:

Declaration تمام توابعی که در proc.c تعریف شده اند در defs.h است.

عدد اختصاص داده شده به این فراخوانی سیستمی در فایل syscall.h مشخص شده است. شکل:

```

25 #define SYS_trace_syscalls 24

```

خط زیر در فایل syscall.c اضافه می شود تا تعریف تابع handler را در کرنل و شل متصل کند. شکل:

```

108 extern int sys_trace_syscalls(void);

```

همچنین خط زیر در فایل syscall.c در بردار فراخوانی سیستمی مشخص می کند که چه handler ای باید برای فراخوانی سیستمی مورد نظر باید اجرا شود.

```
135 [SYS_trace_syscalls] sys_trace_syscalls,
```

خط زیر در فایل `usys.S` اضافه می شود و فراخوانی ای که کاربر انجام می دهد و فراخوانی سیستمی را بهم متصل می کند.

```
34 SYSCALL(trace_syscalls)
```

خط زیر در فایل `user.h` اضافه می شود و `interface` تابعی که کاربر می تواند صدا بزند را مشخص می کند.

```
30 int trace_syscalls(int);
```

تست:

برای تست این فراخوانی سیستمی فایل `trace.c` اضافه شده است. با اجرای `trace on` و `trace off` به ترتیب فراخوانی سیستمی `trace_syscalls` با ورودی ۱ و ۰ صدا زده می شوند و `update` شدن و چاپ اطلاعات را فعال می کنند. با اجرای `trace` فراخوانی های سیستمی `write` و `getpid` و `reverse` و `get_children` و `fork` و ... و همچنین دستور `ls` اجرا می شود و در صورتی که چاپ اطلاعات فعال باشد می توان نتیجه آن را مشاهده کرد.

```
24 int
25 main(int argc, char *argv[])
26 {
27     if (argc == 1)
28         test(argv);
29     else if (argc == 2)
30     {
31         if (!strcmp(argv[1], "on"))
32             trace_syscalls(1);
33         else if (!strcmp(argv[1], "off"))
34             trace_syscalls(0);
35     }
36     exit();
37 }
```

## فراخوانی سیستمی `get_children`

این فراخوانی سیستمی فرزندان پردازنده دارای `pid` ای که در ورودی گرفته ایم را نشان می دهد. در `handler` این فراخوانی سیستمی (`sys_get_children`) پس از خواندن ورودی از `stack` تابع `children` صدا زده می شود که آرایه ای از فرزندان پردازنده مورد نظر را برمی گرداند. در ادامه توسط یک

حلقه محتویات این آرایه در یک عدد قرار داده می شود. برای مثال اگر آرایه {4, 5} باشد مقدار 54 بدست می آید و این عدد به عنوان خروجی برگردانده می شود.

```
94 ////////////////////////////////////////////////// Get children handler
95 int
96 sys_get_children(void)
97 {
98     int pid;
99     if(argint(0, &pid) < 0)
100         return -1;
101
102     // Array of children's pids
103     int* children_array = children(pid);
104
105     // Number made by concating all pids
106
107     int output_number = 0;
108     int i = 0, modulus = 1, temp_pid = 0;
109     while (children_array[i])
110     {
111         output_number += modulus * children_array[i];
112         temp_pid = children_array[i];
113         while (temp_pid != 0)
114         {
115             temp_pid /= 10;
116             modulus *= 10;
117         }
118         i++;
119     }
120     print_trace_syscalls();
121     return output_number;
122 }
```

تابع `children` که در فایل `proc.c` تعریف شده است بر روی همه پردازش های موجود حلقه می زند و هرکدام که پدرش پردازنده ورودی باشد را در لیست اضافه می کند و در نهایت این لیست را برمیگرداند.

```
38 //////////////////////////////////////////////////
39 int*
40 children(int pid)
41 {
42     int index = 0;
43     static int children[NPROC];
44     for (int i = 0; i < NPROC; i++)
45     {
46         children[i] = 0;
47     }
48
49     acquire(&ptable.lock);
50     for (int i = 0; i < NPROC; i++)
51     {
52         if(ptable.proc[i].parent->pid == pid)
53         {
54             children[index] = ptable.proc[i].pid;
55             index++;
56         }
57     }
58     release(&ptable.lock);
59     return children;
60 }
61 //////////////////////////////////////////////////
```



## تغییرات دیگر:

Declaration تمام توابعی که در proc.c تعریف شده اند در defs.h است.

عدد اختصاص داده شده به این فراخوانی سیستمی در فایل syscall.h مشخص شده است. شکل:

```
23 | #define SYS_get_children 22
```

خط زیر در فایل syscall.c اضافه می شود تا تعریف تابع handler را در کرنل و شل متصل کند.

```
106 | extern int sys_get_children(void);
```

همچنین خط زیر در فایل syscall.c در بردار فراخوانی سیستمی مشخص می کند که چه handler ای باید برای فراخوانی سیستمی مورد نظر باید اجرا شود.

```
133 | [SYS_get_children] sys_get_children,
```

خط زیر در فایل usys.S اضافه می شود و فراخوانی ای که کاربر انجام می دهد و فراخوانی سیستمی را بهم متصل می کند.

```
32 | SYSCALL(get_children)
```

خط زیر در فایل user.h اضافه می شود و interface تابعی که کاربر می تواند صدا بزند را مشخص می کند.

```
28 | int get_children(int);
```

## فراخوانی سیستمی get\_grandchildren

این فراخوانی سیستمی فرزندان و نوادگان پردازنده دارای pid ای که در ورودی گرفته ایم را نشان می دهد. در handler این فراخوانی سیستمی (sys\_get\_grandchildren) پس از خواندن ورودی از stack تابع grandchildren صدا زده می شود که آرایه ای از فرزندان و نوادگان پردازنده مورد نظر را برمی گرداند. در ادامه توسط یک حلقه محتویات این آرایه در یک عدد قرار داده می شود. برای مثال اگر آرایه {4, 5, 6, 7} باشد مقدار 7654 بدست می آید و این عدد به عنوان خروجی برگردانده می شود. شکل:



```

127 // Get grandchildren number
128 int
129 sys_get_grandchildren(void)
130 {
131     int pid;
132     if(argint(0, &pid) < 0)
133         return -1;
134
135     // Array of grandchildren's pids
136     int* children_array = grandchildren(pid);
137
138     // Number made by concating all pids
139     int output_number = 0;
140     int i = 0, modulus = 1, temp_pid = 0;
141     while (children_array[i])
142     {
143         output_number += modulus * children_array[i];
144         temp_pid = children_array[i];
145         while (temp_pid != 0)
146         {
147             temp_pid /= 10;
148             modulus *= 10;
149         }
150         i++;
151     }
152
153     return output_number;
154 }

```

تابع grandchildren که در فایل proc.c تعریف شده است. در این تابع از یک صف استفاده می شود که در ابتدا پرده و ورودی در ابتدای آن قرار دارد. سپس توسط یک حلقه روی این صف حرکت می کنیم و در هر لحظه فرزندان پرده ای که روی آن هستیم را به انتهای صف اضافه می کنیم. و در واقع آن را expand می کنیم. برای اضافه کردن فرزندان یک پرده از تابع children که در قسمت قبل توضیح

داده شد استفاده می کنیم.

```

565 // Grandchildren
566 int*
567 grandchildren(int pid)
568 {
569     static int grandchildren[NPROC];
570     for (int i = 0; i < NPROC; i++)
571     {
572         grandchildren[i] = 0;
573     }
574     // Expanding root
575     int* curr_children = children(pid);
576     int j = 0, index = 0;
577     while (curr_children[j])
578     {
579         grandchildren[index] = curr_children[j];
580         index++;
581         j++;
582     }
583
584     int i = 0;
585     while (grandchildren[i])
586     {
587         curr_children = children(grandchildren[i]);
588         j = 0;
589         while (curr_children[j])
590         {
591             grandchildren[index] = curr_children[j];
592             index++;
593             j++;
594         }
595         i++;
596     }
597
598     return grandchildren;
599 }

```

در واقع کاری شبیه به الگوریتم BFS انجام می شود به جز اینکه پرده های expand شده از صف حذف نمی شوند چرا که در نهایت تمام فرزندان و نوادگان باید در صف باشند. شکل:

### تغییرات دیگر:

Declaration تمام توابعی که در proc.c تعریف شده اند در defs.h است.

عدد اختصاص داده شده به این فراخوانی سیستمی در فایل syscall.h مشخص شده است.

```
24 #define SYS_get_grandchildren 23
```

خط زیر در فایل syscall.c اضافه می شود تا تعریف تابع handler را در کرنل و شل متصل کند.

```
107 extern int sys_get_grandchildren(void);
```

همچنین خط زیر در فایل syscall.c در بردار فراخوانی سیستمی مشخص می کند که چه handler ای باید برای فراخوانی سیستمی مورد نظر باید اجرا شود.

```
134 [SYS_get_grandchildren] sys_get_grandchildren,
```

خط زیر در فایل usys.S اضافه می شود و فراخوانی ای که کاربر انجام می دهد و فراخوانی سیستمی را بهم متصل می کند.

```
33 SYSCALL(get_grandchildren)
```

خط زیر در فایل user.h اضافه می شود و interface تابعی که کاربر می تواند صدا بزند را مشخص می کند.

```
29 int get_grandchildren(int);
```

### تست:

برای تست فراخوانی سیستمی get\_grandchildren و get\_children فایل children\_test.c اضافه شده است. در این فایل ابتدا با استفاده از fork یک پرده فرزند ایجاد می شود. در این پرده ۳ فرزند دیگر با استفاده از fork ایجاد می شوند و در نهایت با استفاده از فراخوانی های سیستمی get\_grandchildren و get\_children فرزندان و نوادگان پرده های ایجاد شده نشان داده می شود.

```

5 void
6 test()
7 {
8     if(fork() == 0) // child
9     {
10         if(fork() == 0 || fork() == 0 || fork() == 0) // child
11         {
12             sleep(100);
13         }
14         else // parent
15         {
16             int children2 = get_children(getpid());
17             printf(1, "Children of root children: %d\n", children2);
18             wait();
19             wait();
20             wait();
21         }
22         sleep(200);
23     }
24     else // parent
25     {
26         sleep(200);
27         int children1 = get_children(getpid());
28         int grandchildren1 = get_grandchildren(getpid());
29         printf(1, "Root children: %d\n", children1);
30         printf(1, "Root grandchildren: %d\n", grandchildren1);
31         wait();
32     }
33 }

```