

1. استفاده از فراخوانی های سیستمی در کتابخانه ها

در فایل makefile متغیری به نام ULIB وجود دارد که شامل 4 شیء¹ است:

ulib.o usys.o printf.o umalloc.o

به اختصار به توضیح سورس کد هر کدام می پردازیم:

(1) Ulib.c : درین فایل تعریف یک سری توابع کاربردی آورده شده است مانند:

strcpy , strcmp , strlen , memset , strchr , gets , stat , atoi , memmove

که تمامی شان در user.h اظهار² شده اند. از بین این توابع در gets , stat از فراخوانی سیستمی استفاده شده است:

در gets از سیستم کال read برای خواندن یک خط از stdin استفاده شده است.

در stat نیز از توابع open , fstat , close به ترتیب برای باز کردن یک فایل , گرفتن فیلد های struct fstat³ و در نهایت بستن فایل استفاده میشود

(2) Usys.s : درین جا فایل آبجکت مربوط به usys با کد اسمبلی ساخته می شود که ابتدای آن یک ماکرو است:

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

عدد مربوط به هر سیستم کال در رجیستر eax ذخیره میشود و سپس INT 64 کال شده و یک وقفه رخ می دهد. درینجا به تله افتاده و تابع trap متوجه میشود عامل یک سیستم کال است لذا از eax عدد را خوانده و می فهمد کدام سیستم کال است. برای هر سیستم کال یک ماکرو با نام همان داریم. مثلاً:

```
.globl wait; \
wait: \
    movl $SYS_wait, %eax; \
    int $T_SYSCALL; \
    ret
```

¹ object

² declare

³ در فایل هدر یوزر دیکلر شده است

3) **Printf.c** : درین فایل تابع **printf** که مثل همه توابع قبل در **user.h** دیکلر شده تعریف شده است. درین فایل دو تابع به نام های **printint** , **putc** وجود دارد که نهایتا دو تابع **printf** و **printint** تابع **putc** را صدا می زنند که آنهم با تابع سیستمی **write** یک کاراکتر را پرینت می کند.

4) **Umalloc.c** : اینجا تابع **malloc** که طبعاً در **user.h** دیکلر شده تعریف شده است.

این تابع نیز برای اختصاص دادن حافظه استفاده میشود و نهایتاً با تابع سیستمی **sbrk** حافظه پروسس را افزایش می دهد

2. انواع دسترسی سطح کاربر به سطح هسته در لینوکس

دسترسی به سطح هسته اصولاً با **interrupt** صورت میگیرد که خود شامل دو نوع است:

اول : **interrupt** های سخت افزاری که غالباً از طریق **I/O** رخ میدهند و بصورت سنکرون اجرا میشوند مثلاً فشردن دکمه کیبرد یا حرکت موس

دوم : **interrupt** های نرم افزاری که بصورت آسنکرون رخ میدهند و در اصطلاح به آنها **trap** گفته میشود که خود سه نوع دارند:

✓ فراخوانی های سیستمی

✓ استثناها مانند تقسیم بر صفر یا دسترسی بدون مجوز به حافظه

✓ سیگنال ها مانند **SIGINIT** , **SIGKILL**

در لینوکس تعدادی **Pseudo-File-System** مانند **sys** , **dev** , **proc** وجود دارند که یک رابط برای ساختار های سطح هسته هستند که استفاده از آنها نیازمند مجوز دسترسی به سطح هسته است.

ساز و کار فراخوانی سیستمی

3. آیا امکان فعال کردن همه تله ها با سطح دسترسی **DPL_USER** وجود دارد؟

مطلقاً خیر! زیرا اگر کاربر بخواهد تله دیگری فعال کند سیستم عامل **xv6** این اجازه را به او نخواهد داد. منطق این حفاظت آن است که ممکن است برنامه کاربر در جایی مشکل داشته باشد و فعال کردن تله دیگر مشکل را ممکن است پیچیده تر کند. ضمن اینکه اگر او امکان اجرای تله ها را داشته باشد می تواند به سطح هسته دسترسی پیدا کند که این امنیت سیستم را به خطر می اندازد

4. علت ذخیره شدن ss , esp روی پشته در صورت تغییر سطح دسترسی چیست؟

می دانیم در هر سطح یک پشته مجزا لازم است. یعنی یک پشته کاربر و یک پشته هسته داریم. وقتی یک تله فعال میشود و مجبور به تغییر مود و رفتن به هسته هستیم باید از پشته هسته استفاده کنیم. لذا ابتدا باید ss , esp که به پشته کنونی ما اشاره دارند را ذخیره کنیم تا بتوانیم پس از پایان سیستم کال مربوط به تله در هسته مجدد به سطح کاربر برگشته و اجرای پروسس مد نظر را از جای قبلی ادامه دهیم. اگر چنانچه تغییر سطح نداشته باشیم دیگر نیازی به ذخیره این دو رجیستر نخواهد بود.

توابع دسترسی به پارامترهای سیستم کال

3 تابع برای دسترسی به پارامترهای سیستم کال وجود دارد که هر یک را به اختصار توضیح میدهیم:

(a) تابع **argint**: این تابع آدرس آرگومان n ام ورودی در حافظه را محاسبه میکند. استک از آدرس بزرگ به کوچک پر میشود و آخرین خانه آن محل بازگشت پس از اتمام تابع است که در **esp** ذخیره میشود. پس برای داشتن آدرس آرگومان n ام از رابطه زیر داریم:

$$Adr_n = (4 * n) + esp$$

این آدرس به همراه اشاره گر به حافظه مد نظر برای **int** به تابع **fetchint** داده میشود که چک کند آدرس ارسالی 4 بایت یعنی به اندازه یک عدد صحیح در حافظه پروسس باشد و اگر مشکلی نبود آرگومان دوم را مقدار دهی میکند.

(b) تابع **argptr**: این تابع نیز با کمک تابع **argint** آدرس اشاره گر مد نظر را گرفته و آرگومان سوم که سائز اشاره گریست را دریافت کرده و بررسی میکند که اشاره گر با آن سائز در حافظه پروسس باشد و اگر بود آرگومان دوم را مقدار دهی میکند.

(c) تابع **argstr**: این تابع با کمک تابع **argint** آدرس اول رشته را حساب کرده و آن را به **fetchstr** پاس میدهد. این تابع مشابه **fetchint** بررسی میکند که آدرس داده شده به اندازه رشته در حافظه پروسس کوجود باشد و اگر بود آرگومان دوم را با مقدار این اشاره گر مقداردهی میکند. نهایتاً از ابتدای اشاره گر پیمایش کرده و در صورت رسیدن به کاراکتر '\0' که بیانگر انتهای رشته است طول آن را بر میگردد و در غیر این صورت مقدار (-1) بر میگردد

تمامی این توابع موجود بودن آدرس داده شده در حافظه پروسس را بررسی میکنند تا مطمئن شوند پروسس ها به حافظه هم دسترسی نداشته باشند و برنامه به باگ یا مشکلات امنیتی نخورد.

فراخوانی سیستمی **sys_read** مربوط به سیستم کال **read** است:

read(int fd , void* buffer , int max)

آرگومان دوم بافری است که مقدار **read** در آن قرار می گیرد و سومی بیشینه تعداد بایت هایی است که خوانده میشود.

چنانچه قبل از رسیدن به این تعداد بایت به پایان فایل برسیم سیستم عامل به طور خودکار عملیات خواندن را پایان میدهد.

نگاهی به این سیستم کال بیندازیم:

```
int
sys_read(void)
{
    struct file *f;
    int n;
    char *p;

    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n)
    < 0)
        return -1;
    return fileread(f, p, n);
}
```

ابتدا با کمک تابع `argfd` که خود با کمک `argint` مقدار `fd` که آرگومان اول تابع `read` است را دریافت کرده و اعتبارسنجی میکند) مقدار `file` descriptor را دریافت کرده و سپس ابتدا آرگومان سوم (`max`) را به کمک تابع `argint` دریافت کرده و نهایتاً به کمک تابع `argptr` بررسی میکند که فضای آدرس دهی از ابتدای بافر تا انتهای آن (به طول آرگومان دوم) در حافظه پروسس قرار داشته باشد. ممکن بود برای نوشتن از فایلی حجیم و `max` بزرگی استفاده شود که در بافر یک پروسس نمی گنجد. درین صورت سیستم عامل مجبور بود ادامه نوشتن و خواندن را در بافر یک پروسس دیگر انجام دهد که این امر مشکلات و باگ های زیادی دارد. لذا مجبوریم تا این بازه را در تابع `argptr` قبل از شروع عملیات خواندن بررسی کنیم.

بررسی گامهای اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

یک برنامه ساده مینویسیم که از طریق آن شناسه یک پروسس را دریافت کنیم:

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

پس از بوت سیستم عامل یک breakpoint در خط 137 فایل `syscall.c` می گذاریم. این باعث میشود برنامه در خط مورد نظر متوقف شود. مطابق شکل 1 که در صفحه بعدی آورده شده است.

سپس در آخر با دستور `bt` به خروجی مطابق شکل 2 می رسیم.

```
syscall.c
134 struct proc *curproc = myproc();
135
136 num = curproc->tf->eax;
B+> 137 if (num > 0 && num < NELEM(syscalls) && syscalls[num])
138 {
139     curproc->tf->eax = syscalls[num]();
140 }
141 else
142 {
143     cprintf("%d %s: unknown sys call %d\n",
144         curproc->pid, curproc->name, num);
145     curproc->tf->eax = -1;
146 }
147 }
148
149
150
151
152
153
154
155
156
157
158
159
```

شکل 1

```
Process Thread 1:1 in: syscall
(gdb) bt
#0  syscall () at syscall.c:137
#1  0x8010656d in trap (tf=0x8dfbefb4) at trap.c:43
#2  0x8010630a in alltraps () at trapasm.S:20
#3  0x8dfbefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

شکل 2

هر تابعی که صدا زده شود یک stack frame اختصاصی دارد که محل نگهداری متغیرهای محلی و آدرس بازگشت تابع و باقی اطلاعات است. در اصل خروجی دستور backtrace در هر خط یک stackframe است که به ترتیب از درونی ترین frame که در آن قرار داریم شروع به نمایش می کند.

مراحل تعریف و اجرای یک سیستم کال

- (1) در فایل `syscall.h` یک عدد برای سیستم کال جدید در نظر گرفته میشود.
- (2) در فایل `user.h` نام آن سیستم کال را دیکلر میکنیم.
- (3) در فایل `usys.S` تعریف اسمبلی سیستم کال انجام میشود (با استفاده از ماکرویی که قبل تر توضیح داده شد)
- (4) پس از اجرای دستور `INT 64` در ماکروی قبلی وارد بخش `vector 64` در فایل `vectors.S` می شویم که نهایتا با `push` شدن مقدار `64` به بخش `alltraps` در فایل `trapasm.S` می رویم.

```
vector64:
    pushl $0
    pushl $64
    jmp alltraps
```

- (5) درین بخش ابتدا `trap frame` مربوط به این بخش را ساخته و پس از `push` کردن آن در استک تابع `trap` در فایل `trap.c` فراخوانی میشود.

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal
```

- (6) تابع `trap` پس از فهمیدن اینکه فراخوانی توسط سیستم کال بوده است `trap frame` پوش شده در استک را بعنوان `trap frame` پروسس حال حاضر قرار می دهد و نهایتا تابع `syscall` صدا زده میشود.

```
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
}
```

(7) در فایل `syscall.c` یک آرایه از سیستم کال ها موجودست که شماره هر سیستم کال را به تابع آن نگاشت میکند. بخشی ازین آرایه به شکل زیر است:

```
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,
```

(8) پس از خواندن شدن شماره سیستم کال که در متغیر `eax` فریم تله پروسس قرار دارد تابع `syscall` تابع مربوط به آن شماره را صدا میزند و خروجی آن را بجای شماره سیستم کال در `eax` پروسس ذخیره میکند⁴ (دقت شود که `eax` فیلد مربوط به `trap frame` یک پروسس است)

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

    num = curproc->tf->eax;
    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;
    }
}
```

خط ششم تصویر⁴

در تصویر مربوط به خروجی دستور `bt` (شکل 2 صفحه 5) دیدیم که خروجی مربوط به مراحل 5 تا 7 نمایش داده شد. چرا که اولین فریم را در مرحله 5 و در فایل `trapasm.S` تولید کردیم.

وقتی در داخلی ترین `frame` قرار داریم استفاده از دستور `down` بی معنی است و دچار ارور خواهد شد.

استفاده از دستور `up` ما را به یک فریم عقب تر برمیگرداند که ابتدای `syscall` در فایل `trap.c` خواهد بود:

```
31 {
32     lidt(idt, sizeof(idt));
33 }
34
35 //PAGEBREAK: 41
36 void
37 trap(struct trapframe *tf)
38 {
39     if(tf->trapno == T_SYSCALL){
40         if(myproc()->killed)
41             exit();
42         myproc()->tf = tf;
43         > syscall();
44         if(myproc()->killed)
45             exit();
46         return;
47     }
48
49     switch(tf->trapno){
50     case T_IRQ0 + IRQ_TIMER:
51         if(cpuid() == 0){
52             acquire(&tickslock);
53             ticks++;
54             wakeup(&ticks);
55             release(&tickslock);
56         }
```

در فایل `syscall.h` شماره سیستم کال (`getpid()` برابر 11 است. ولی با خواندن مقدار رجیستر `eax` مربوط به فریم تله مقدار 5 برمیگردد که مد نظر ما نیست.

علت این است که قبل از اجرای سیستم کال `pid` سیستم کال های دیگری فراخوانی و اجرا میشوند.

این سیستم کال ها به ترتیب در صفحه بعد آورده میشوند.


```
(gdb) i lo
num = 5
curproc = 0x80113a10 <ptable+176>
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137      if (num > 0 && num < NELEM(syscalls) && syscalls[num])
(gdb) i lo
num = 1
curproc = 0x80113a10 <ptable+176>
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137      if (num > 0 && num < NELEM(syscalls) && syscalls[num])
(gdb) i lo
num = 3
curproc = 0x80113a10 <ptable+176>
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137      if (num > 0 && num < NELEM(syscalls) && syscalls[num])
(gdb) i lo
num = 12
curproc = 0x80113a8c <ptable+300>
(gdb) █
```

```
(gdb) i lo
num = 7
curproc = 0x80113a8c <ptable+300>
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137      if (num > 0 && num < NELEM(syscalls) && syscalls[num])
(gdb) i lo
num = 11
curproc = 0x80113a8c <ptable+300>
(gdb) █
```

سیستم کال های فراخوانی شده به ترتیب:

- (1) سیستم کال read برای خواندن از ورودی با شماره 5 اجرا میشود.(به تعداد کاراکتر های ورودی فراخوانی میشود)
- (2) سیستم کال fork با شماره 1 برای ایجاد یک پروسس جدید استفاده میشود.
- (3) سیستم کال wait با شماره 3 فراخوانی شده و پروسس پدر منتظر می ماند تا فرآیند خواندن فرزندش به پایان برسد.
- (4) سپس سیستم کال sbrk با شماره 12 برای تخصیص حافظه برای ورودی خوانده شده صدا زده میشود.
- (5) سپس سیستم کال exec با شماره 7 به معنای پایان یافتن اجرای پروسه فرزند صدا زده میشود.
- (6) نهایتا سیستم کال getpid با شماره 11 برای دریافت شناسه پروسس پدر صدا زده میشود.

Find_digital_root

ابتدا مراحل ذکر شده در بخش ایجاد سیستم کال را اجرا میکنیم. ابتدا دیکلر در `user.h` و اختصاص عدد در `syscall.h` که در تصاویر زیر آورده شده است:

```
#define SYS_find_digital_root 22
#define SYS_copy_file 23
#define SYS_get_uncle_count 24 // system calls
#define SYS_lifetime 25 int find_digital_root(void);
```

سپس در فایل `syscall.c` توابع را در آرایه استاتیک مربوط به ماکروها اضافه میکنیم تا بعدا شماره هر تابع با سیستم کال مورد نظرش مپ شود.

```
[SYS_find_digital_root] sys_find_digital_root, extern int sys_find_digital_root(void);
[SYS_copy_file] sys_copy_file, extern int sys_copy_file(void);
[SYS_get_uncle_count] sys_get_uncle_count, extern int sys_get_uncle_count(void);
[SYS_lifetime] sys_lifetime, extern int sys_lifetime(void);
```

در فایل `utils.c` تعریف این تابع آورده شده است. نکته در ورودی این تابع است که ظاهرا باید عدد باشد اما از نوع `void` است که دلیل آن در ادامه آورده شده است:

```
// return digital root of number given
int sys_find_digital_root(void)
{
    int n = myproc()->tf->ebx;
    int res = 0;
    while (n > 0)
    {
        res += n % 10;
        n /= 10;
        if (res > 9)
            res -= 9;
    }
    return res;
}
```

چون قرار است آرگومان تابع را از رجیستر بخوانیم و از استک استفاده نشود لذا آرگومان تابع از نوع `int` نخواهد بود.

در صفحه بعد تصویری از کار با رجیستر را میبینیم که قرار است مقدار آرگومان قبلی در متغیر `prev` ذخیره شده و آرگومان جدید را در `ebx` ذخیره کنیم.

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

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf(2, "not good args : %d!\n", argc);
        exit();
    }

    int prev;

    asm volatile(
        "movl %%ebx, %0\n\t"
        "movl %1, %%ebx"
        : "=r"(prev)
        : "r"(atoi(argv[1])));

    int res = find_digital_root();

    asm volatile(
        "movl %0, %%ebx" :: "r"(prev));

    printf(1, "FDR : %d\n", res);

    exit();
}
```

حال برای تست این تابع در محیط ترمینال این تابع را در فرمت کامند به همراه ورودی آن وارد کرده و خروجی را در خط بعدی میگیریم. برای اینکه این تابع کامند شود نیز آن را در لیست UPROGS در فایل makefile اضافه میکنیم:

```
UPROGS=\
_cat\
_echo\
_forktest\
_grep\
_init\
_kill\
_ln\
_ls\
_strdiff\
_test_fdr\
_get_pid\
_mkdir\
_rm\
_sh\
_stressfs\
_usertests\
_wc\
_zombie\
_user_get_uncle_count\
_user_lifetime\
```

```
Enter starting shell
Group #8 :
1. Mohammad Ali Ghahari
2. Nima Tajik
3. Mohammad Sadeghi
$ test_fdr 234
FDR : 9
$ -
```

Int copy_file

نحوه اضافه کردن این سیستم کال نیز مانند بخش قبل است.

ابتدا باید فایل مبدأ موجود با سیستم کال `open` و مود `READONLY` باز شود چون قرار نیست چیزی درون آن نوشته شود. همچنین فایل مقصد نیز مود `WRONLY` دارد تا درون آن امکان نوشتن باشد.

```
// return 0 on success and -1 on error
int sys_copy_file(void)
{
    char *src, *dest;

    if (argstr(0, &src) < 0 || argstr(1, &dest) < 0)
    {
        // cprintf("fail to read input!\n");
        return -1;
    }

    if (strlen(src) == strlen(dest) && strncmp(src, dest, strlen(src)) == 0)
    {
        // cprintf("can't copy to src!\n");
        return -1;
    }

    if (_open(dest, O_WRONLY) >= 0)
    {
        // cprintf("file already exist!\n");
        return -1;
    }
}
```

قبل از `open` کردن فایل ها نیز یک سری بررسی انجام میشود از نظر اینکه فایل ها موجود باشند و یکی نباشند و مشکلی از جهت خواندن در فایل ها نباشد.

ضمن اینکه در فایل مقصد امکان نوشتن وجود داشته باشد.

در قبال تمام این خطا ها تابع منفی 1 را برخواهد گرداند.

بعد از آن در تصویر صفحه بعد عملیات کپی متن را انجام می دهیم:

```
int srcfd = _open(src, O_RDONLY);
int dstfd = _open(dest, O_CREATE | O_WRONLY);
if (srcfd < 0 || srcfd >= NOFILE || dstfd >= NOFILE || dstfd < 0)
{
    // cprintf("can't open files.\n");
    return -1;
}

int r = 0, w = 0;
char buf[512];
while ((r = fileread(myproc()->ofile[srcfd], buf, sizeof(buf))) > 0)
{
    buf[r] = 0;
    w = filewrite(myproc()->ofile[dstfd], buf, r);
    if (w < r || w < 0 || r < sizeof(buf))
        break;
}
if (r < 0 || w < 0 || w < r)
{
    // cprintf("cp: error copying %s to %s\n", src, dest);
    return -1;
}
if (_close(srcfd) < 0 || _close(dstfd) < 0)
{
    // cprintf("close fail!\n");
    return -1;
}
return 0;
```

با متغیرهای r , w عملیات خواندن و نوشتن را کنترل میکنیم. در حلقه `while` با تابع `fileread`, `filewrite` که در سیستم کال های مربوط به فایل مورد استفاده قرار می گیرند از فایل مبدا روی فایل مقصد می نویسیم و در انتها باز هم چک میکنیم که مشکلی در نوشتن و خواندن نباشد. برای مثال حجم خوانده شده و نوشته شده بیشتر یا کمتر از هم نباشند. برای بررسی درستی نیز آن را مشابه تابع قسمت قبل در `makefile` کامند کرده و دو ورودی به آن می دهیم که فرمت ورودی دادن و نمونه خروجی در صفحه بعد مشاهده میشود:

```
sh          2 17 28572
stressfs    2 18 15444
usertests   2 19 62944
wc          2 20 15972
zombie      2 21 14092
console     3 22 0
$ test_copy README ali.txt
$ cat ali.txt
```

پس از اجرا محتوای فایل README در فایل ali.txt کپی میشود.

```
Froehlich, Yakir Goaron, Shivam Handa, Bryan Henry, Jim Huang, Alexander  
Kapshuk, Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Austin  
Liew, Imbar Marinescu, Yandong Mao, Matan Shabtay, Hitoshi Mitake, Carmi  
Merimovich, Mark Morrissey, mtasm, Joel Nider, Greg Price, Ayan Shafqat,  
Eldar Sehayek, Yongming Shen, Cam Tenny, tyfkda, Rafael Ubal, Warren  
Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas  
Wolovick, wxdao, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.
```

```
The code in the files that constitute xv6 is  
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.
```

ERROR REPORTS

```
We don't process error reports (see note on top of this file).
```

BUILDING AND RUNNING XV6

```
To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run  
"make". On non-x86 or non-ELF machines (like OS X, even on x86), you  
will need to install a cross-compiler gcc suite capable of producing  
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).  
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC  
simulator and run "make qemu".  
$
```

و ضمناً این فایل به همراه طول آن و شماره آن در لیست فایل ها اضافه خواهد شد:

```
usertests      2 19 62944  
wc              2 20 15972  
zombie         2 21 14092  
console        3 22 0  
ali.txt        2 23 2286  
$
```


Int get_uncle_count

یک struct گلوبال بنام ptable برای دسترسی به تمام پروسس ها در فایل proc.c موجودست لذا بدنه این تابع را درین فایل پیاده سازی میکنیم.

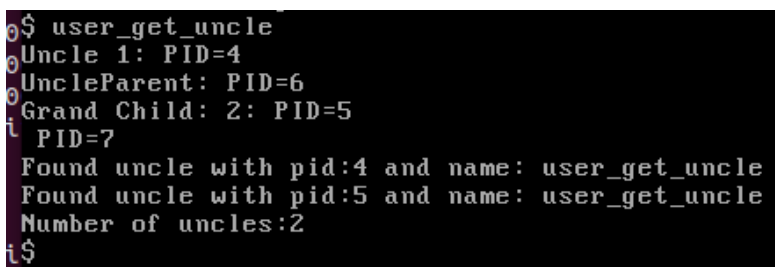
```
int sys_get_uncle_count(void)
{
    acquire(&ptable.lock);
    int count = 0;
    struct proc *my_proc = myproc(); // Get the current process
    struct proc *curr_proc;

    for (curr_proc = ptable.proc; curr_proc < &ptable.proc[NPROC]; curr_proc++)
    {
        if (curr_proc->state == UNUSED || curr_proc->state == EMBRYO || curr_proc->pid == my_proc->pid || curr_proc->pid == my
            continue;

        if (curr_proc->parent && curr_proc->parent->pid == my_proc->parent->parent->pid)
        {
            cprintf("Found uncle with pid:%d and name: %s\n", curr_proc->pid, curr_proc->name);
            count++;
        }
    }
    release(&ptable.lock);

    return count;
}
```

درین تابع ابتدا فرایند های معمول در توابع مربوط به پروسس طی میشود. سپس در حلقه روی پروسس های معتبر چک میکنیم و اگر پدر یک پروسس شناسه اش با پدر بزرگ پروسس فعلی یکی باشد نتیجه میگیریم آن پروسس عمومی پروسس فعلی است و شناسه آن را پرینت میکنیم و به تعداد عمو های پروسس فعلی می افزاییم و نهایتا پس از اتمام بررسی تمام پروسس ها تعداد عموها چاپ میشود.



```
$ user_get_uncle
Uncle 1: PID=4
UncleParent: PID=6
Grand Child: 2: PID=5
PID=7
Found uncle with pid:4 and name: user_get_uncle
Found uncle with pid:5 and name: user_get_uncle
Number of uncles:2
$
```

برای تست نیز 4 پروسس در فایل user_get_uncle_count.c ساخته و تابع get_uncle را در درونی ترین پروسس (نوه) صدا میزنیم که سورس آن در repository مربوطه در گیت موجود است.

get_proccess_lifetime

بدنه این تابع درون فایل sysproc.c نوشته شده شده است:

```
uint sys_lifetime(void){
    uint xticks;

    acquire(&tickslock);
    xticks = ticks;
    release(&tickslock);

    struct proc *my_proc = myproc(); // Get the current process

    cprintf("[sys_lifetime] Current time is: %d this app creation time is: %d\n",xticks,my_proc->xticks);
    return (xticks-my_proc->xticks)/100;
}
```

همانطور که در شکل مشخص است ابتدا پروسس فعلی را ذخیره و lock میکنیم.(با استفاده از تابع release و متغیر گلوبال tickslock که در طول برنامه ثابت است) سپس با استفاده از فیلد xticks که در بدنه ساختار proc تعریف شده است زمان به وجود آمدن آن را ذخیره می کنیم. ساختار تغییر یافته پروسس:

```
// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    uint xticks; // time of creation
    char name[16]; // Process name (debugging)
};
```

مطابق شکل فیلد xticks یک عدد است که نگهدارنده زمانی است که یک پروسس هم زمان با fork ایجاد میشود.

سپس در هر لحظه خواستیم عمر پروسس در حال اجرا را ببینیم با استفاده از اختلاف xticks آن لحظه اش با لحظه ایجاد شدنش این زمان بدست میاید(تقسیم بر 100 نهایی برای تبدیل کردن به ثانیه می باشد)

برای تست درستی این برنامه نیز در پروسس پدر یک پروسس فرزند ایجاد می کنیم و بلافاصله زمان بوجود آمدنش را ذخیره میکنیم.

سپس 10 ثانیه صبر میکنیم و بصورت موازی عمر هر دو پروسس محاسبه میشود. ولی چون پروسس پدر یک wait کوچک برای اتمام پروسس فرزند دارد طول عمرش کمی بیشتر است که چون تقسیم بر 100 میشود خیلی محسوس نیست.

```
int main(void)
{
    int pid;
    pid=fork();

    if(pid==0){
        //child process
        printf(2,"Child lifetime before sleep is: %d\n",lifetime());
        sleep(1000);// sleep for 10 seconds
        printf(2,"Child lifetime after sleep is: %d\n",lifetime());

    } else{
        wait(); // after child process finished
        printf(2,"Parent lifetime is: %d\n",lifetime());
    }

    exit();
}
```

خروجی نمونه برنامه بصورت زیر خواهد بود:

```
$ user_lifetime
[sys_lifetime] Current time is: 4332 this app creation time is: 4332
Child lifetime before sleep is: 0
[sys_lifetime] Current time is: 5333 this app creation time is: 4332
Child lifetime after sleep is: 10
[sys_lifetime] Current time is: 5333 this app creation time is: 4331
Parent lifetime is: 10
$
```