سهیل حاجیان منش 810100119
امیرعلی رحیمی 810100146
مهدی نوری 810100231

github repo : https://github.com/MahdiNoori2003/OS-Lab-2
last commit : `86ab1983a715d4cf470fe3c199cc5c26ba7cec45`

۱.بررسی استفاده از فراخوانی های سیستمی در کتابخانه های استفاده شده در xv6:

متغیر ulib در makefile شامل چهار آبجکت فایل است که هر کدام سورس مربوط به خود را دارند که در ادامه بررسی میکنیم.
آبجکت فایل ها:

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

- **ulib.c:**

این فایل شامل توابع strcpy,strcmp,strlen,memset,strchr,gets,stat,atoi,memmove است که در user.h تعریف شده اند. تنها در توابع gets و stat از فراخوانی های سیستمی استفاده شده است که در زیر شرح داده شده است:
stat: هدف این تابع این است که به متادیتا یک فایل دسترسی پیدا کنیم. برای این منظور ابتدا با سیستم کال open فایل مورد نظر را باز میکنیم. سپس با سیستم کال fstat متادیتا آن فایل را میخوانیم و در نهایت با سیستم کال close فایل را میبندیم.
gets: از آنجایی که این تابع برای خواندن ورودی که یک عملیات IO است نیاز به اجرا در حالت کرنل دارد که برای این منظور از سیستم کال read در آن استفاده شده است.

- **printf.c:**

این فایل شامل توابع putc,printint,printf میشود.
در تابع putc با استفاده از سیستم کال write یک کاراکتر را روی stdout مینویسیم. در توابع printint و printf هم از putc استفاده شده است پس بطور غیرمستقیم آنها هم از سیستم کال write استفاده میکنند.

- **umalloc.c:**

این فایل شامل سه تابع malloc , free , morecore میباشد که برای تخصیص یا ازاد کردن حافظه استفاده می شوند. در تابع morecore با استفاده از سیستم کال sbrk فضای پردازه را افزایش میدهد.در تابع malloc هم از morecore به همین منظور استفاده شده است.

- **usys.c:**

ابتدا usys.o توسط کد اسمبلی زیر ساخته میشود:

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

در اینجا با صدا کردن ماکرو SYSCALL نام سیستم کال مورد نظر در محل name قرار میگیرد.سپس توسط کد اسمبلی movel شماره متناظر با این سیستم کال در رجیستر EAX ذخیره میشود. سپس با دستور اسمبلی int یک interrupt صادر میشود و به تابع trap می رویم و در آن جا چون یک system call است تابع syscall صدا زده میشود و در آنجا با توجه به مقدار EAX متوجه میشود کدام سیستم کال باید صدا زده شود.

2. انواع روش های دسترسی از سطح کاربر به کرنل در لینوکس

در لینوکس برای دسترسی به کرنل نیازمند یک Interrupt هستیم. وقفه ها میتوانند سخت افزار و یا نرم افزاری باشند.

وقفه سخت افزاری: این وقفه ها معمولا توسط سخت افزار ها بطور مثال ورودی ها و خروجی ها رخ میدهند و به صورت asynchronous هم اجرا میشوند. (مانند فشردن کلید کیبورد یا حرکت دادن موس)

وقفه نرم افزاری: به وقفه های نرم افزاری trap گفته میشود که بطور کلی سه نوع است:

System call.1: فراخوانی های سیستمی که مفصل توضیح داده شده است.

Exception.2: استثنا ها که بطور مثال میتوان به دسترسی غیرمجاز به حافظه یا تقسیم بر صفر اشاره کرد.

Signal.3: سیگنال ها در لینوکس انواع مختلفی دارند که پرکاربردترین آن ها عبارتند از SIGINT و SIGKILL و SIGTERM.

3.آیا همه تله ها را میشود با سطح دسترسی DPL_USER فعال نمود؟

خیر.همه تله ها را نمی توان در این سطح اجرا کرد، به خصوص آنهایی که به دلایل امنیتی یا یکپارچگی سیستم(security or system integrity) به امتیازات بالاتری(higher privileges) نیاز دارند.اگر کاربر سعی کند تله ای دیگر را فعال کند که در سطح دسترسی او نیست با استثنا(protection exception) روبرو خواهد شد.مزیت این کار این است که از مختل شدن امنیت سیستم جلوگیری میکند و کاربر نتواند به راحتی به هسته دسترسی داشته باشد.

4.چرا در صورت تغییر سطح دسترسی، ss و esp روی پشته push میشوند؟

برای هر سطح دسترسی یک پشته داریم.یک پشته برای سطح کاربر() و یک پشته برای سطح هسته().در هنگام تغییر سطح دسترسی (بطور مثال در اینجا فعال شدن یک تله،سیستم برای اینکه بتواند به کد های هسته دسترسی پیدا کند باید از پشته هسته استفاده کند.)این دو رجیستر ss و esp مورد استفاده قرار میگیرند.در مثال گفته شده ابتدا ss و esp که به پشته فعلی اشاره دارند ذخیره میشوند. بعد از اتمام رسیدگی به تله مقادیر قدیمی این دو رشته بازیابی شده و برنامه از همان جای قبلی در سطح کاربر ادامه می یابد.
همانطور که گفته شد این دو رجیستر تنها در هنگام تغییر سطح دسترسی کاربرد دارند در غیر اینصورت نیازی به استفاده از آن ها نیست.

5.توضیح توابع دسترسی به پارامتر های فراخوانی سیستمی

برای دسترسی به پارامتر های فراخوانی سیستمی سه تابع وجود دارد که هر کدام به شرح است:

- ***argint:***

این تابع دو آرگومان به عنوان ورودی میگیرد.ورودی اول شماره آرگومان ورودی در تابع است. و ورودی دوم پوینتر به حافظه مدنظر است. سپس آدرس آرگومان n ام تابع را در استک محاسبه میکند.طبق تعریفات گفته شده استک از آدرس بیشتر به کمتر پر میشود و آدرس سر استک که در رجیستر esp ذخیره شده است به همراه آدرس نقطه فراخوانی تابع برای بازگشت از تابع آخرین مقدار هایی هستند که در استک پوش شده اند. پس آدرس آرگومان n ام تابع بصورت زیر حساب شده است:

addr=esp+4+4*n

سپس این آدرس را به همراه پوینتر به حافظه که ورودی دوم تابع بود به تابع fetchint پاس میدهد.وظیفه این تابع این است که بررسی کند آدرس ارسالی بعلاوه ۴ بایت در حافظه پردازه باشد.در این صورت پوینتر به حافظه که ورودی دوم بود را مقدار دهی میکند.

- ***argptr:***

این تابع به کمک تابع argint آدرس پوینتر مورد نظر را دریافت میکند و سپس به کمک همین تابع سایز بافر مورد نظر را نیز دریافت میکند و سپس بررسی میکند که پوینتر با این سایز در حافظه پردازه وجود دارد یا خیر. در صورت وجود، آن را مقداردهی میکند.

- ***argstr:***

این تابع به کمک argint آدرس ابتدای رشته را مشخص میکند و سپس این مقدار و پوینتر به حافظه مورد نظر را به تابع fetchstr پاس میدهد. در این تابع هم دوباره چک میکند آدرس داده شده در حافظه پردازه باشد و در اینصورت آن را با مقدار پوینتر مقداردهی میکند.

همان که در توضیحات گفته شد تمامی این توابع بررسی میکنند که میزان حافظه داده شده حتما در حافظه پردازه مورد نظر قرار گیرد. چون در غیر اینصورت ممکن است از حافظه پردازه دیگر استفاده کند که باعث ایجاد مشکلات زیادی در پردازنده میشود.

تابع *sys_read* که فراخوانی سیستمی مربوط به تابع *read* است بصورت زیر تعریف شده است:

```
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);
}
```

خود تابع read نیز بصورت زیر تعریف شده است:

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

تابع *sys_read* در if سه کار را انجام میدهد. به کمک تابع argfd مقدار fd را دریافت میکند. (آرگومان اول) به کمک تابع argint مقدار آرگومان سوم که ماکزیمم سایز ورودی است را دریافت میکندو به کمک argptr بررسی میکند که کل فضای آدرس دهی از ابتدا تا ماکزیمم در حافظه پردازه قرار میگیرد یا خیر. اگر این مورد بررسی نشود ممکن است مقدار ماکزیمم خیلی زیاد باشد و از یک فایل بزرگ در حال خواندن باشیم حافظه پردازه فعلی جوابگو نباشد و سیستم عامل باقی موارد را در در حافظه پردازه دیگری شروع به نوشتن میکند.طبیعتا این کار میتواند باعث بروز مشکلات زیادی اعم از ایجاد باگ یا مشکلات امنیتی بشود. البته در موارد هم مقداری ماکزیمم از حافظه بافر بیشتر است که در این موارد هم سرریز حافظه رخ میدهد که نوعی باگ است.

بررسی گام های اجرای سیستم کال در سطح کرنل در gdb

ابتدا برنامه سطح کاربر گفته شده را مینویسیم و آن را مشابه پروژه ۱، به Makefile اضافه میکنیم:

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


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

حال سیستم عامل را بالا میاوریم. سپس یک break point در ابتدای تابع syscall می گذاریم و برنامه سطح کاربر را اجرا میکنیم. پس از آنکه به break point رسید، دستور bt را اجرا میکنیم که تصویر خروجی به شکل زیر است:

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x80105fbd in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80105d5f in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
```

دستور bt که مخفف backtrace است stack call برنامه در لحظه کنونی را نشان می دهد. هر تابع که صدا زده می شود یک frame stack مخصوص به خودش را می گیرد که متغیرهای محلی و

آدرس بازگشت و غیره در آن ذخیره می شود . خروجی این دستور در هر خط یک frame stack را نشان می دهد که به ترتیب از درونی ترین frame شروع به توضیح دادن میکنیم:

۱. alltraps: در تابع ابتدا trapframe مربوط به این trap ساخته میشود و در استک پوش میشود. سپس تابع ()trap را فرا میخواند(همانطور که در تصویر هم مشاهده میشود این تابع در trap.c قرار دارد)

۲. trap: در این تابع ابتدا بررسی میشود که trap number داده شده مربوط به چه وقفه ای است. بعد از اینکه مشخص شد از نوع سیستم کال میباشد، trapframe مربوط به پردازه ی فعلی را برابر trapframe ای که در استک پوش شده بود قرار داده و تابع ()syscall را صدا میکند.

۳. syscall: در این تابع، eax را از trapframe پردازه ی فعلی میخواند که این مقدار برابر شماره سیستم کال مورد نظر میباشد. حال با استفاده از [num]syscalls تابع مربوط به آن سیستم کال را فرا میخواند و خروجی آن را در eax در trapframe پردازه فعلی ذخیره میکند. (آرایه syscalls در ابتدای فایل syscalls.c تعریف شده که شماره هر سیستم کال را به تابع مربوط به آن مپ میکند.)

از آنجایی که در حال حاضر در درونی ترین لایه frame قرار داریم، نیاز به وارد کردن دستور down که به یک لایه درونی تر میرود، نداریم. لذا با وارد کردن down به ارور زیر بر میخوریم:

```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
```

میدانیم که شماره ی سیستم کال getpid برابر ۱۱ میباشد.

حال مقدار num که مقدار رجیستر eax در آن ریخته شده است را پرینت میکنیم. میبینیم که مقدار آن برابر ۵ است و برابر ۱۱ نمیباشد. علت آن است که ابتدا باید تعدادی سیستم کال read استفاده کرد تا دستور را از ورودی خواند و در ادامه چند پردازه دیگر که لازم است قبل از اجرای getpid، اجرا شوند اجرا میشوند.(از آنجایی که طول getpid برابر ۶ است، ۶ بار سیستم کال read استفاده میشود.)

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$2 = 5
```

مقدار بعدی num، برابر ۱ میشود که مربوط به سیستم کال fork است که برای ایجاد پردازه جدید برای برنامه سطح کاربر صدا میشود:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$7 = 1
```

مقدار بعدی num، برابر ۳ میشود که مربوط به سیستم کال wait است که این سیستم کال در پردازه ی پدر صدا میشود که منتظر میماند تا کار پردازنده ی فرزند تمام شود:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$8 = 3
```

مقدار بعدی num، برابر ۱۲ میشود که مربوط به سیستم کال sbrk است که این سیستم کال به پردازه ایجاد شده، حافظه اختصاص میدهد:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$9 = 12
```

مقدار بعدی num، برابر ۷ میشود که مربوط به سیستم کال exec است که برای اجرای برنامه ی pid در پردازه ی ایجاد شده استفاده میشود:

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$10 = 7
(gdb) continue
Continuing.
```

در نهایت، مقدار num، برابر ۱۱ میشود که مربوط به getpid است که انتظار آن را داشتیم.

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$11 = 11
```

در آخر، تعدادی write که مقدار num آن برابر ۱۶ میباشد اجرا میشود که خروجی مورد نظر را برای کاربر چاپ کند.

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$12 = 16
```

خروجی دستور سطح کاربر اجرا شده:

```
$ getpid
Current process id is: 3
```

ارسال آرگومان های فراخوانی های سیستمی

find digital root

برای اضافه کردن این system call، در ابتدا تابع در دسترس کاربر را در user.h قرار می دهیم:

```
int find_digital_root(void);
```

این ورودی تابع باید در اصل int باشد ولی از آنجا که می خواهیم آرگومانها را با استفاده از رجیسترها پاس بدهیم در خود تابع ورودی ای نمیگیریم و از stack استفاده نمی کنیم.
حال در ادامه تعریف این تابع را در usys.S انجام می دهیم:

```
SYSCALL(find_digital_root)
```

این ماکرو تابع find_digital_root را به SYS_find_digital_root مپ میکند.
اینجا SYS_find_digital_root عدد سیستم کال است که باید در syscall.h اضافه کنیم :

```
#define SYS_find_digital_root 22
```

حال باید تابع در سطح کرنل را اضافه کنیم. ابتدا تعریف تابع را در syscall.c می نویسیم و سپس آن را به آرایه مپ شماره system call به تابع اضافه می کنیم:

```
extern int sys_find_digital_root(void);
```

```
[SYS_find_digital_root] sys_find_digital_root,
```

حال تعریف این system call را در فایل find_digital_root_util.c اضافه میکنیم:

```
static int get_digital_root(int n)
{
    if (n < 0)
    {
        return -1;
    }

    while (n > 9)
    {
        int sum = 0;
        while (n != 0)
        {
            sum += n % 10;
            n /= 10;
        }
        n = sum;
    }
    return n;
}

int sys_find_digital_root(void)
{
    int result = get_digital_root(myproc()->tf->ebx);
    return result;
}
```

در اینجا همانطور که مشخص است مقدار رجیستر ebx را به عنوان ورودی به تابع میدهیم تا محاسبات روی آن انجام شود. این محاسبات به عهده تابع get_digital_root است.

از آنجا که فایل جدیدی ساختیم باید find_digital_root_util.o را به متغیر OBJS داخل Makefile اضافه کنیم:

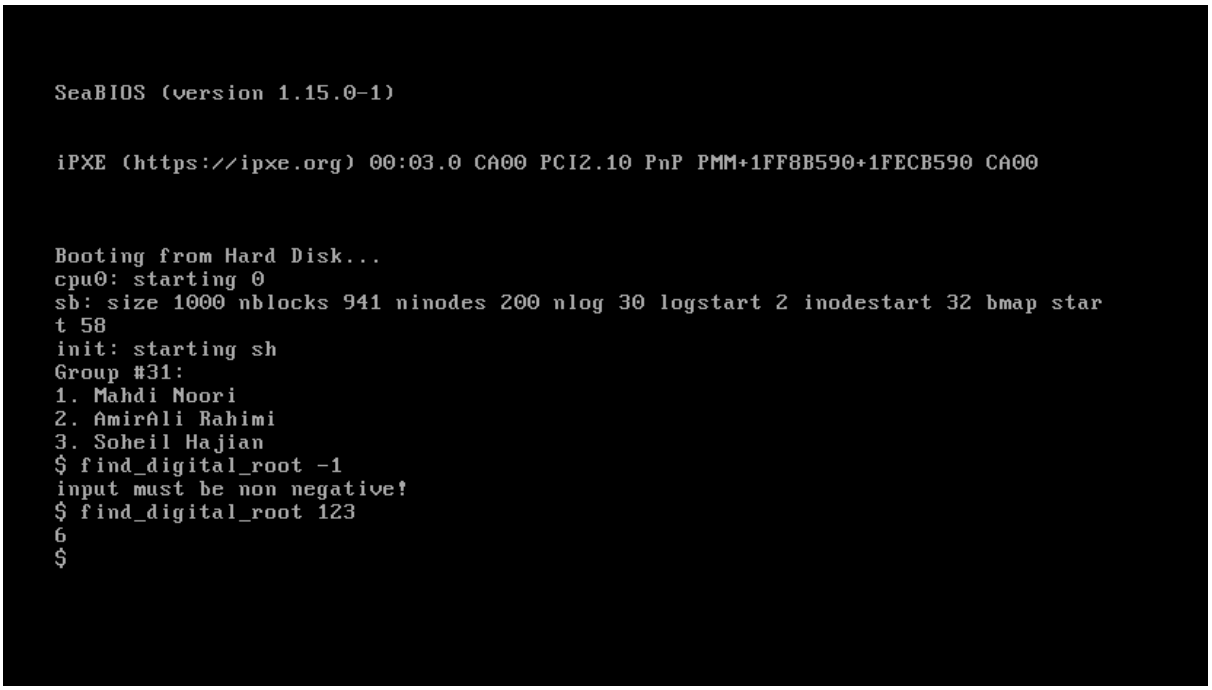
```
    uart.o\
    vectors.o\
    vm.o\
    find_digital_root_util.o\
# Cross-compiling (e.g., on Mac OS X)
# TOOLPREFIX = i386-jos-elf
```

در ادامه برای تست تابع find_digital_root.c را ایجاد می کنیم که حاوی یک برنامه سطح کاربر است . این برنامه عدد مدنظر را ورودی گرفته و خروجی مطلوب را به نمایش می گذارد. توجه شود که در ابتدا ما یک بخش کد assembly ایجاد کردیم که آرگومان ورودی را در رجیستر ebx قرار میدهد و مقدار قبلی ebx را تا انتهای عملیات ذخیره میکند.

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

#define INVALID_INPUT_ERROR "input must be non negative!\n"
#define FEW_ARGUMANTS_ERROR "few arguements.needed one integer provided zero\n"

int find_digital_root_syscall(int n)
{
    int prev_ebx;
    asm volatile(
        "movl %%ebx, %0\n\t"
        "movl %1, %%ebx"
        : "=r"(prev_ebx)
        : "r"(n));
    int result = find_digital_root();
    asm volatile(
        "movl %0, %%ebx" ::"r"(prev_ebx));
    return result;
}
```

```c
int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf(2, FEW_ARGUMANTS_ERROR);
        exit();
    }

    int n = (*argv[1] == '-') ? -atoi(argv[1] + 1) : atoi(argv[1]);

    int result = find_digital_root_syscall(n);
    if (result == -1)
    {
        printf(2, INVALID_INPUT_ERROR);
    }
    else
    {
        printf(1, "%d\n", result);
    }

    exit();
}
```

نمونه اجرای برنامه:



پیاده سازی فراخوانی های سیستمی

copy_file

برای اضافه کردن دستور copy_file به سیستم کال مانند بخش قبل عمل میکنیم که تصاویر آن در زیر قابل مشاهدست :

```
int copy_file(char*,char*);
```

(user.h)

```
SYSCALL(copy_file)
```

(usys.S)

```
#define SYS_copy_file 25
```

(syscall.h)

```
extern int sys_copy_file(void);
```

```
[SYS_copy_file] sys_copy_file,
```

(syscall.c)

برای پیاده سازی به این صورت عمل میکنیم که ابتدا فایل مبدا را باز کرده و در ادامه فایل مقصد را ایجاد میکنیم . حال یک بافر به اندازه ۱۰۲۴ در نظر میگیریم. در ادامه از فایل مبدا به اندازه بلوک های ۱۰۲۴ تایی خوانده و در فایل مقصد می نویسیم.

توجه شود که در انتها یک \n اضافه می شود که فرمت خروجی فایل صحیح باشد

برای باز کردن فایل مبدا از **namei** استفاده میکنیم که به ما **inode** فایل مبدا را خروجی میدهد.
برای ایجاد فایل مقصد از **create** استفاده می کنیم که **inode** مقصد را خروجی میدهد و در ادامه آبجکت را فایل برای هر دو آنها ایجاد می کنیم .

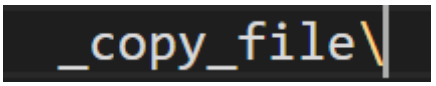
توجه شود که تعریف تابع در sysfile.c قرار دارد چون این عملیات جزو بخش file به حساب می آید.

```
int sys_copy_file(void)
{
  char *src, *dst;
  struct file *fsrc, *fdst;
  struct inode *ipsrc, *ipdst;
  int n;

  if (argstr(0, &src) < 0 || argstr(1, &dst) < 0)
    return -1;

  begin_op();

  if ((ipsrc = namei(src)) == 0)
  {
    end_op();
    return -1;
  }

  if ((ipdst = namei(dst)) != 0)
  {
    end_op();
    return -1;
  }
```

```
ipdst = create(dst, T_FILE, 0, 0);
if (ipdst == 0)
{
  end_op();
  return -1;
}
if ((fdst = filealloc()) == 0)
{
  iunlockput(ipdst);
  end_op();
  return -1;
}

iunlock(ipdst);

if ((fsrc = filealloc()) == 0)
{
  end_op();
  return -1;
}

fsrc->type = FD_INODE;
fsrc->ip = ipsrc;
fsrc->off = 0;
fsrc->readable = 1;
fsrc->writable = 0;
```

```
fdst->type = FD_INODE;
fdst->ip = ipdst;
fdst->off = 0;
fdst->readable = 1;
fdst->writable = 1;

char buffer[1024];
while ((n = fileread(fsrc, buffer, 1024)) > 0)
{
  if (filewrite(fdst, buffer, n) != n)
    break;
}

if (n > 0)
{
  filewrite(fdst, "\n", strlen("\n"));
}

fileclose(fsrc);
fileclose(fdst);

end_op();

return n < 0 ? -1 : 0;
}
```

برای اجرا نیز فایل copy_file.c را ایجاد میکنیم و **_copy_file** را به بخش UPROGS فایل Make file اضافه می کنیم.



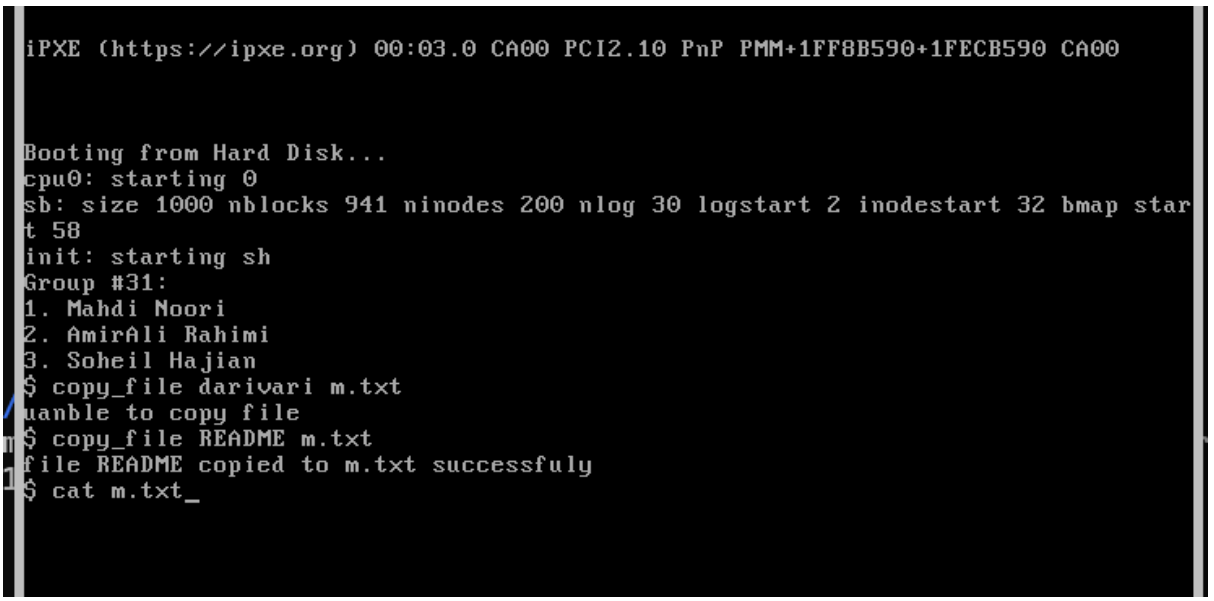
(Makefile)

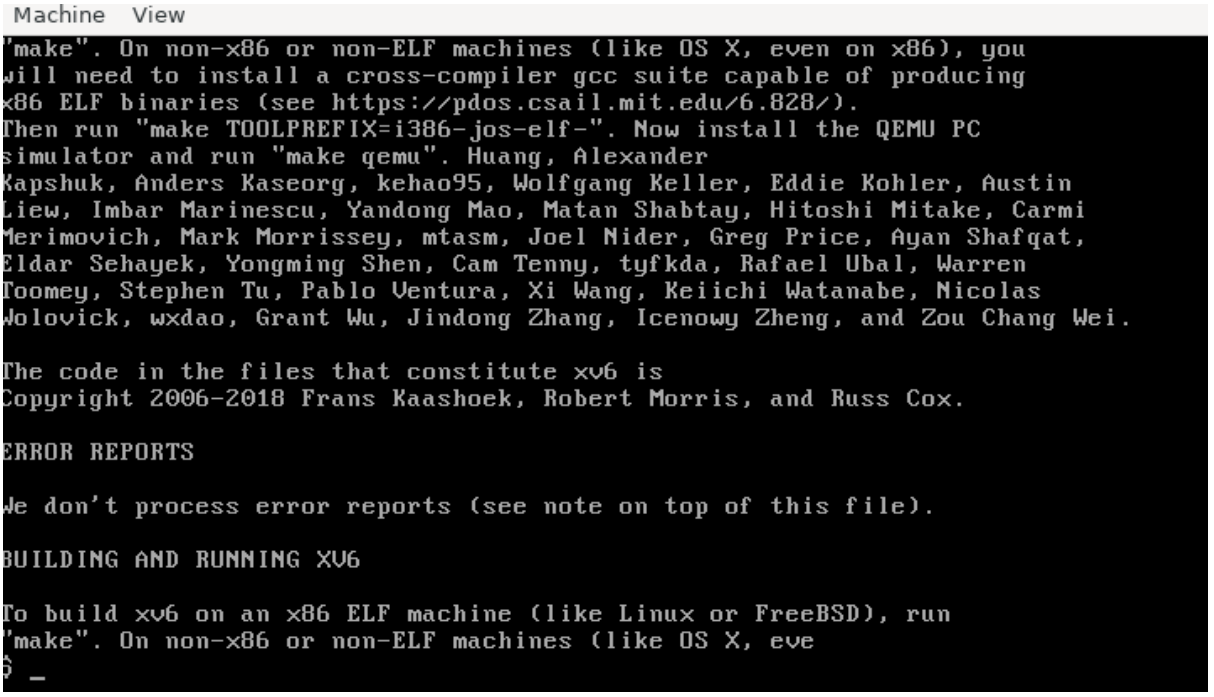
```c
#include "types.h"
#include "user.h"
#include "fcntl.h"
#include "stat.h"

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf(1, "2 args required. provided %d\n", argc);
        exit();
    }
    char *src = argv[1];
    char *dst = argv[2];

    if (copy_file(src, dst) < 0)
    {
        printf(1, "uanble to copy file\n");
        exit();
    }

    printf(1, "file %s copied to %s successfuly\n", src, dst);
    exit();
}
```

(copy_file.c)

توجه شود که در صورتی که فایل مبدا موجود نباشد یا اگر فایل مقصد موجود باشد عملیات ارور میدهد .

نمونه اجرای برنامه :





get_uncle_count

برای اضافه کردن این دستور مانند قبل عمل می کنیم و قطعه کد های زیر را به فایل های مربوطه اضافه می کنیم :

```
int get_uncle_count(int);
```

(user.h)

```
SYSCALL(get_uncle_count)
```

(usys.S)

```
#define SYS_get_uncle_count 23
```

(syscall.h)

```
extern int sys_get_uncle_count(void);
```

```
[SYS_get_uncle_count] sys_get_uncle_count,
```

(syscall.c)

در ادامه تعریف تابع sys_get_uncle_count را در فایل **sysproc.c** قرار میدهیم :

```
int sys_get_uncle_count(void)
{
  int pid;
  if (argint(0, &pid) < 0)
    return -1;
  return count_uncles(pid);
}
```

حال برای بازگرداندن تعداد عمو ها ابتدا process را از **ptable** می یابیم. میدانیم که پدر هر process ای که عموی process مد نظر ما باشد **pid** یکسانی با پدربزرگ process مدنظر دارد پس تمام process های **ptable** را جستجو می کنیم.

توجه شود که بررسی میکنیم که پراسس مشغول باشد و در استیت UNUSED نباشد .

این تعداد را میشماریم و مساوی count می گذاریم. مشخصا یکی از آنها پدر process است پس count منهای یک را باز می گردانیم.

```
int count_uncles(int pid)
{
  int p_index = -1;
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].pid == pid)
    {
      p_index = i;
      break;
    }
  }
  if (p_index == -1)
  {
    return -1;
  }
```

```
  int count = 0;
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[p_index].parent->parent->pid == ptable.proc[i].parent->pid &&
        ptable.proc[i].parent->state != UNUSED)
    {
      count++;
    }
  }

  if (count == 0)
  {
    return -1;
  }
  else
  {
    return count - 1;
  }
}
```

(proc.c)

در ادامه برای تست برنامه کاربر زیر را می نویسیم و به **UPROGS** فایل **Makefile** اضافه میکنیم:

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

void test()
{
    int forkpid = fork();
    if (forkpid > 0)
    {
        wait();
    }
    else if (forkpid == 0)
    {
        printf(1, "the number of uncles is %d\n", get_uncle_count(getpid()));
    }
    else
    {
        printf(2, "Failed to create 3rd process.\n");
    }
    exit();
}
```

```c
int main(int argc, char *argv[])
{
    int pid1 = fork();

    if (pid1 < 0)
    {
        printf(2, "Fork failed\n");
        exit();
    }

    if (pid1 == 0)
    {
        sleep(50);
    }
    else
    {
        int pid2 = fork();

        if (pid2 < 0)
        {
            printf(2, "Fork failed\n");
            exit();
        }

        if (pid2 == 0)
        {
            sleep(50);
        }
        else
```

```
        if (pid2 == 0)
        {
            sleep(50);
        }
        else
        {

            int pid3 = fork();

            if (pid3 < 0)
            {
                printf(2, "Fork failed\n");
                exit();
            }

            if (pid3 == 0)
            {
                test();
            }
        }
        wait();
        wait();
        wait();
    }
    exit();
}
```

(get_uncle_count_test.c)

```
_get_uncle_count_test\
```

(Makefile)

نمونه خروجی برنامه :

```
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 star
t 58
init: starting sh
Group #31:
1. Mahdi Noori
2. AmirAli Rahimi
3. Soheil Hajian
$ get_uncle_count_test
the number of uncles is 2
$ _
```

get_process_life_time

برای اضافه کردن تعریف این system call مانند قبل عمل می کنیم و هر کدام از کد های زیر را به فایل های مربوطه اضافه می کنیم:

```
int get_process_lifetime(int);
```

(user.h)

```
SYSCALL(get_process_lifetime)
```

(usys.S)

```
#define SYS_get_process_lifetime 24
```

(syscall.h)

```
extern int sys_get_process_lifetime(void);
```

```
[SYS_get_process_lifetime] sys_get_process_lifetime,
```

(syscall.c)

تعریف تابع سیستمی را در فایل sysproc.c وارد می کنیم.

```
int sys_get_process_lifetime(void)
{
  int pid;
  if (argint(0, &pid) < 0)
    return -1;
  return calc_process_lifetime(pid);
}
```

مشخصا تابع را برای process که pid آن را داریم کال می کنیم.

برای پیاده سازی این دستور به استراکچر **proc** مقدار **init_time** را اضافه میکنیم . این متغیر زمان ایجاد شدن این process را نگه میدارد.

```
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
  char name[16];               // Process name (debugging)
  struct rtcdate init_time;
  uint children_count;
};
```

در ادامه تابع **fork** را به گونه ای تغییر میدهیم که به ازای هر بار ایجاد شدن process با تابع **cmostime** مقدار فعلی زمان را در آن ذخیره کند که برابر زمان ایجاد process است.

```
  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);
  cmostime(&np->init_time);
  curproc->children_count++;
  return pid;
}
```

حال برای محاسبه زمان زنده بودن process از تابع زیر استفاده می کنیم.

توجه شود که در اینجا زمان کنونی سیستم و ایجاد process به ثانیه تبدیل می شود و در ادامه از هم کم می شوند .

```
int calc_process_lifetime(int pid)
{
  int p_index = -1;
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].pid == pid)
    {
      p_index = i;
      break;
    }
  }
  if (p_index == -1)
  {
    return -1;
  }
  return calc_diff_till_now(ptable.proc[p_index].init_time);
}
```

```
static int calc_diff_till_now(struct rtcdate input_time)
{
  struct rtcdate now;
  cmostime(&now);
  uint start_seconds = input_time.hour * 3600 + input_time.minute * 60 + input_time.second;
  uint end_seconds = now.hour * 3600 + now.minute * 60 + now.second;

  uint start_timestamp = input_time.year * 31536000 + input_time.month * 2592000 + input_time.day * 86400 +
  uint end_timestamp = now.year * 31536000 + now.month * 2592000 + now.day * 86400 + end_seconds;

  uint difference = end_timestamp - start_timestamp;
  return difference;
}
```

در ادامه برای تست برنامه کاربر زیر را مینویسیم و به UPROGS فایل Makefile اضافه می کنیم :

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

void test()
{
    int forkpid = fork();
    if (forkpid > 0)
    {
        wait();
    }
    else if (forkpid == 0)
    {

        while (1)
        {
            if (get_process_lifetime(getpid()) >= 10)
            {
                printf(1, "%d\n", forkpid);
                printf(1, "child process lifetime is %d\n", get_process_lifetime(getpid()));
                kill(forkpid);
                break;
            }
        }
        exit();
    }
    else
    {
        printf(2, "Failed to create process.\n");
    }
}
```

```c
int main(int argc, char *argv[])
{
    int forkpid = fork();
    if (forkpid > 0)
    {
        wait();
    }
    else if (forkpid == 0)
    {
        test();
        sleep(500);

        printf(1, "parent process lifetime is %d\n", get_process_lifetime(getpid()));
        exit();
    }
    else
    {
        printf(2, "Failed to create process.\n");
    }
    exit();
}
```

(get_process_lifetime_test.c)

```
_get_process_lifetime_test\
```

(Makefile)

خروجی به صورت زیر است :