آزمایشگاه سیستم عامل

پروژه یک

اعضای گروه:

الهه خداوردی - 810100132

فرشته باقری - 810100089

عاطفه میرزاخانی - 810100220

Repository: https://github.com/elahekhodaverdi/Operating-System-Lab-Projects

Latest Commit: 1a2eaa462b8a891f7d125770cdcb1b4dda9bf063

فراخوانی سیستمی

1. **کتابخانه های (قاعدتا سطح کاربر، منظور فایل های تشکیل دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانی های سیستمی و علت این استفاده بررسی نمایید.**

متغیر ULIB در Makefile از 4 آبجکت فایل تشکیل شده است که عبارت اند از:
Ulib.o, usys.o, printf.o,umalloc.o

سورس هر کدام از آبجکت ها را بررسی می کنیم.

- **Ulib.c**

این فایل شامل توابع زیر می‌باشد:
strcpy, strcmp, strlen, memset, strchr, gets, stat, atoi, memmove

از بین این توابع در stat , gets از فراخوانی های سیستمی استفاده شده است.
gets: در این تابع از فراخوانی سیستمی read استفاده شده است در قسمتی از کد داریم :

```
cc=read(0, &c, 1);
```

این خط در یک حلقه تکرار می‌شود و در هر مرحله یک خط از stdin می‌خواند.
stat: در این تابع از فراخوانی های سیستمی open و fstat و close استفاده شده است.
در این تابع ابتدا با استفاده از open ، یک فایل باز می‌شود سپس با استفاده از fstat اطلاعات فایل مورد نظر را بدست می‌آوریم و در انتها با استفاده از close آن فایل را می‌بندیم.

- **usys.S**

از آنجایی که به پسوند این فایل (.S) است پس usys.o با استفاده از کد اسمبلی ساخته میشود.

تمامی فراخوانی های سیستمی به صورت SYSCALL(name_of_syscall) نوشته شده اند و با فراخوانی هریک از این فراخوانی های سیستمی به #define که در خط 4 این فایل قرار دارد مراجعه می‌شود؛ در اینجا نام فراخوانی سیستمی به صورت global نوشته می شود و همچنین شماره‌ی آیدی این فراخوانی سیستمی در رجیستر eax نوشته می‌شود؛$T_SYSCALL نیز برابر با 64 است زیرا شماره تله فراخوانی سیستمی 64 است و برنامه جهت فراخوانی سیستمی دستور int 64 را فراخوانی می‌کند؛ در واقع هنگامی که interrupt software اتفاق افتد، این اعمال انجام می‌شوند.

ماکروی ابتدای فایل:

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

برای مثال اگر فراخوانی سیستمی ما fork باشد داریم:

```
#define SYSCALL(fork) \
  .globl fork; \
  fork: \
    movl $SYS_ ## fork, %eax; \     #SYS_fork(syscall.h) == 1
    int $T_SYSCALL; \               # T_SYSCALL (traps.h) == 64
    ret
```

- **printf.c**

در این فایل توابع putc, printint, printf وجود دارند که تنها تابعی که در این بخش از فراخوانی سیستمی استفاده می‌کند، تابع putc است.

در تابع putc از فراخوانی سیستمی write به منظور چاپ کردن استفاده شده است. که برای آن fd موردنظر جهت چاپ کردن و کاراکتر مورد نظر انتخاب شده است.

- **umalloc.c**

در این فایل توابع malloc و freeو morecore تعریف شده اند که free و malloc کار تخصیص و آزاد کردن حافظه را به عهده دارند و در آنها از فراخوانی‌های سیستمی استفاده نشده است.

در تابع morecore از فراخوانی سیستمی sbrk استفاده شده است که این فراخوانی سیستمی، حافظه پردازه را گسترش می دهد.

2. **در سطح کاربر میتوانیم با ایجاد interrupt ها به هسته دسترسی داشته باشیم. البته این دسترسی به این معنا نیست که کاربر(user program) می تواند تغییرات دلخواهش را انجام دهد و با فعال کردن یک وقفه در واقع یک درخواست به سیستم عامل می فرستد و OS خودش وقفه را handle می کند.**

وقفه ها می توانند سخت افزاری یا نرم افزاری باشند.

Interrupt های سخت افزاری توسط سخت افزار ها (معمولا I/O) فرستاده می شوند و asynchronous هستند ، به این معنا که آنها بدون وابستگی به جریان اجرای فعلی برنامه رخ می‌دهند. interruptهای سخت‌افزاری توسط رویدادهای خارجی، مانند ورودی صفحه کلید، حرکت موس یا رسیدن بسته‌های شبکه، فعال می‌شوند. این رویدادها می‌توانند در هر زمانی رخ دهند، حتی زمانی که برنامه به طور فعال منتظر آنها نیست.

به interrupt های نرم افزاری trap نیز می گویند. Trap ها توسط خود برنامه ایجاد و به صورت synchronous اجرا می شوند.

این وقفه ها انواع مختلفی دارند:

1. **System calls:** مانند read,write, open, close
2. **Exeptions:** تقسیم به صفر یا پوینتر به آدرس حافظه غیر مجاز
3. **Signals:** برای مثال از SIGINT که برای متوقف کردن یک برنامه با استفاده از کلیدهای Ctrl+C استفاده می‌شود، SIGKILL که برای قطع ناگهانی اجرای یک برنامه استفاده می‌شود و SIGTERM که برای ارسال یک سیگنال پایان به یک برنامه استفاده می‌شود.

همچنین سطح کاربر برای تعامل با هسته می تواند از Library API، File System Interface و Network Interface استفاده کند.

3. **آیا باقی تله‌ها را نمی‌توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟**
خیر. اگر یک پردازه بخواهد interrupt دیگری را فعال کند، 6xv 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.

4. **در صورت تغییر سطح دسترسی، ss و eps روی پشته push می‌شوند.در غیر این صورت push نمی‌شوند. چرا؟**

می‌دانیم که برای هر پردازه دو پشته کاربر و هسته داریم که هرکدام مربوط به حالت دسترسی پردازه می‌شوند. وقتی یک تله فعال می‌شود و دسترسی تغییر پیدا میکند برای سیستم از پشته ی هسته استفاده کند و به همین علت ما باید esp و ss که به پشته کاربر(فعلی) اشاره دارند را در جایی ذخیره کنیم زیرا پس از آن به پشته ی هسته اشاره خواهند کرد. برای ذخیره این دو آن‌ها را در پشته push می‌کنیم تا وقتی که رسیدگی به تله تمام شد و بایستی دسترسی پردازه دوباره به سطح کاربر بازمیگشت آن دو را بازیابی کنیم.
با توجه به صحبت های بالا زمانی که سطح دسترسی پردازه تغییر نکند مقادیر آن‌ها تغییری نمی‌کند و نیازی به ذخیره سازی آن ها در پشته نخواهیم داشت.

بخش سطح بالا و کنترل کننده زبان سی تله

5. **در مورد توابع دسترسی به پارامتر های فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr() بازه آدرس ها بررسی می‌گردد؟ تجاوز از بازه معتبر چه مشکل امنیتی ایجاد می‌کند؟**

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.

برای مثال میتوانیم فراخوانی سیستمی sys_kill را عنوان کنیم.

```
int
sys_kill(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}
```

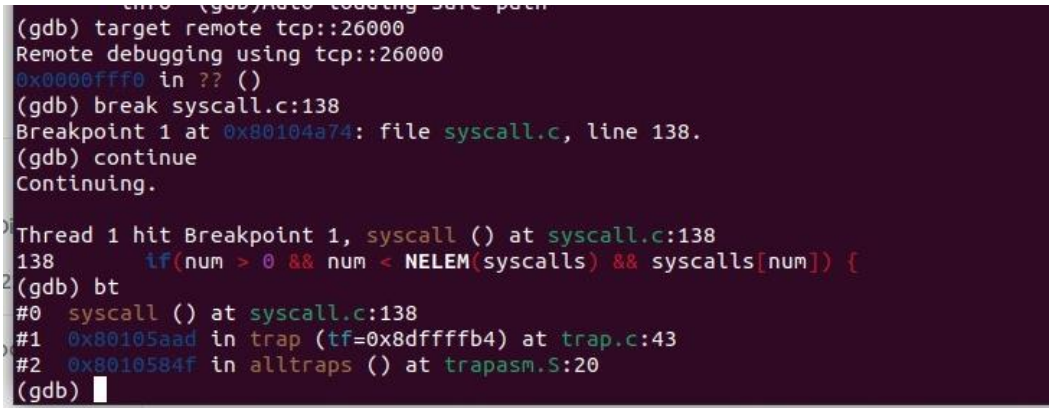
بررسی گامهای اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

یک برنامه سطح کاربر به نام pid نوشته شده و به Makefile اضافه شده که شماره پردازه فعلی را با استفاده از سیستم کال ()getpid چاپ می کند.

xv6-public-master > C pid.c

```
1  #include "types.h"
2  #include "user.h"
3
4  int main(int argc, char* argv[]) {
5      int pid = getpid();
6      printf(1, "Process ID: %d\n", pid);
7      exit();
8  }
```

بعد از اتصال gdb به سیستم عامل یک breakpoint در خط 138 فایل syscall.c قرار می دهیم. با اجرای برنامه سطح کاربر، gdb در خط ذکر شده متوقف می شود و دستور bt را اجرا می کنیم:



دستور "bt" لیستی از تمام فراخوانی های تابع که به نقطه فعلی اجرا منجر شده‌اند، نمایش می‌دهد.

Call stack یک پشته است که برای ذخیره و پیگیری سیر اجرای برنامه (توابع صدا زده شده) است. وقتی یک تابع صدا زده می شود برنامه یک بلوک حافظه به نام stack frame برای آن allocate می کند که در آن متغیرهای محلی تابع، پارامترها، آدرس بازگشت و سایر اطلاعات مورد نیاز وجود دارد. با فراخوانی تابع، استک فریم آن در بالای کال استک push می شود و بعد از اتمام اجرا نیز پاپ می شود و به تابع فراخواننده برمی گردیم. دستور bt در هر لحظه تصویر call stack را چاپ می کند که از درونی ترین تابع به ترتیب چاپ می شوند.

این لیست معمولاً شامل اطلاعات زیر برای هر تابع است:

1. نام توابع: نام توابع در تماس‌های تابع را نشان می‌دهد، از تابعی که در حال حاضر در بالای لیست اجرا می‌شود شروع کرده و به تابع اولیه که فراخوانی شده است برمی‌گردد.

2. آرگومان‌ها

3. آدرس source code

4. آدرس‌های حافظه: آدرس حافظه دستوری که تابع از آنجا فراخوانی شده است.

تحلیل خروجی:

در 6xv مکانیزم تعریف و اجرای فراخوان های سیستمی به صورت زیر است:

برای هر سیستم کال یک عدد و شناسه در نظر گرفته می شود که در فایل های Syscall.h و user.h هستند.

سیستم کال ها به زبان اسمبلی تعریف می شوند که در فایل usys.s هستند.

در 6xv، رجیستر eax معمولاً برای ذخیره مقدار بازگشتی یک سیستم کال استفاده می‌شود. هنگامی که یک سیستم کال فراخوانی می‌شود، معمولاً شماره مورد نظر سیستم کال قبل از اجرای دستور 64 int در رجیستر eax قرار می‌گیرد. پس از اجرای سیستم کال، مقدار یا وضعیت حاصل عموماً در رجیستر eax ذخیره می‌شود تا برای پردازش یا بازیابی بیشتر توسط کد فراخواننده استفاده شود.

با اجرای دستور 64 int در مرحله قبل، وارد بخش 64vector می شویم و بعد از پوش شدن مقدار 64 به alltraps می رود.

Alltraps ابتدا trap frame را می سازد و آن را در Stack پوش می کند و سپس تابع trap را صدا می زند.

سپس تابع trap، trap frame را به عنوان ترپ فریم پردازه فعلی قرار می دهد و تابع syscall را صدا می زند. (جایی که breakpoint داریم)

دستور bt تصویر call stack را در این لحظه نشان می دهد (syscall هنوز اجرا نشده است) که نشان می دهد alltraps، trap را صدا زده است.

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

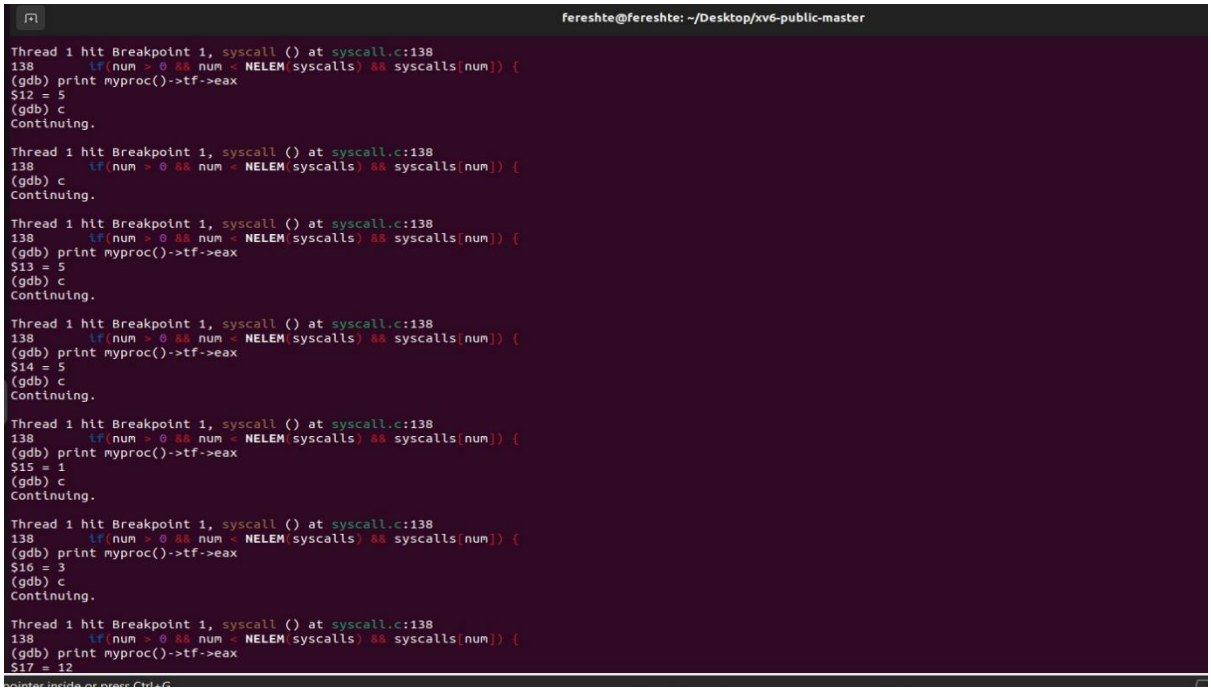
دستور downبه stack frame بالاتر (تابعی که دیرتر صدا زده شده) می رویم که در اینجا trap هیچ تابعی را صدا نزده است پس call stack داده ی دیگری ندارد.

با دستور up می توانیم به یک stack frame عقب تر برگردیم.

```
(gdb) up
#1  0x80105aad in trap (tf=0x8dffffb4) at trap.c:43
43          syscall();
(gdb)
```

همانطور که پیش تر اشاره شد رجیستر eax برای ذخیره سازی شماره سیستم کال به کار می رود. شماره دستور getpaid() برابر 11 است ولی با اجرای دستور زیر به مقدار 11 نمی رسیم چون قبل از سیستم کال مورد نظر

فراخوانی های دیگری هم انجام می شود:



read => 5 : خواندن از ترمینال

fork => 1 : ایجاد پردازه جدید برای اجرای برنامه سطح کاربر

wait => 3 : انتظار برای اتمام پردازه فرزند

sbrk => 12 : تخصیص حافظه به پردازه ایجاد شده

exec => 7 : گذاشتن کد برنامه سطح کاربر در حافظه پردازه ایجاد شده

getpid => 11 : شماره پردازه فعلی را بر می گرداند

write => 16 : چند بار اجرا می شود تا خروجی برنامه سطح کاربر روی ترمینال چاپ شود.

خروجی برنامه pid به صورت زیر است:

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
$ pid
Process ID: 3
```

ارسال آرگومان‌های فراخوانی‌های سیستمی

ابتدا در فایل syscall.h شماره سیستم کال جدیدمان را تعریف می‌کنیم:

```
#define SYS_find_digital_root 22
```

در فایل syscall.c، prototype تابع این سیستم کال را اضافه می کنیم:

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

در فایل syscall.c یک پوینتر به سیستم کال ها اضافه می کنیم. در این فایل یک آرایه از pointer function ها داریم که به واسطه ی اعداد نسبت داده شده به سیستم کال ها، یک پوینتر به سیستم کال ها تعریف می کند. با این کار هر زمان که یک سیستم کال را با شماره متناظر آن صدا کنیم، تابعی که آن سیستم کال به آن اشاره می کند را صدا خواهیم کرد.

```
[SYS_find_digital_root] sys_find_digital_root,
```

در فایل proc.c بدنه‌ی تابع int find_digital_root(int n) را تعریف می‌کنیم که منطق این برنامه در آن قرار دارد و به ازای گرفتن عدد n، ریشه دیجیتال آن را برمی‌گرداند:

```c
int
find_digital_root(int n) {
    while (n >= 10) {
        int sum = 0;
        while (n > 0) {
            sum += n % 10;
            n /= 10;
        }
        n = sum;
    }
    return n;
}
```

سپس این تابع را در usys.S تعریف می‌کنیم:

```
SYSCALL(find_digital_root)
```

سپس در فایل sysproc.c بدنه‌ی تابع int sys_find_digital_root(void) را تعریف می‌کنیم در این تابع، فقط تابع find_digital_root که در کرنل است و آرگومان ورودی را با استفاده از رجیستر ebx پاس می‌دهد.

```
int sys_find_digital_root(void)
{
  int number = myproc()->tf->ebx;
  cprintf("KERNEL: sys_find_digital_root() is called for n = %d\n", number);
  return find_digital_root(number);
}
```

در فایل user.h باید prototype تابع را بنویسیم که برنامه سطح کاربر به واسطه صدا کردن آن ، به سیستم کال مورد نظر متصل می شود.

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

تعریف تابع در defs.h:

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

حال برای برنامه ی سطح کاربر نیاز است تا سیستم کال امتحان شود. برای این کار فایل digital_root.c نوشته شده است که در آن در ابتدا تعداد ورودی‌ها چک میشود ، ثبات قبلی را ذخیره کرده، مقدار جدید را در ثبات قدیمی نوشته، تابع را صدا زده و در انتها مقدار اولیه‌ی ثبات را به آن برمی‌گرداند. و آن را به makefile در قسمت های UPROGS و EXTRA اضافه می کنیم تا کاربر بتواند آن را اجرا کند:

digital_root.c

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

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        if(argc < 2)
            printf(2, "Error: you didn't enter the number!\n");
        else if(argc > 2)
            printf(2, "Error: Too many arguments!\n");
        exit();
    }
    else
    {

        int last_ebx_value;
        int number = atoi(argv[1]);

        asm volatile(
            "movl %%ebx, %0;" // last_ebx_value = ebx
            "movl %1, %%ebx;" // ebx = number
            : "=r" (last_ebx_value)
            : "r"(number)
        );
        printf(1, "USER: find_digital_root() is called for n = %d\n" , number);
        int answer = find_digital_root();
        printf(1, "digital root of number %d is: %d\n" , number , answer);

        asm("movl %0, %%ebx"
            :
            : "r"(last_ebx_value)
        );
    }
    exit();
}
```

با اجرای این برنامه به خروجی زیر می‌رسیم:

```
init: starting sh
Group #29:
Elaheh Khodaverdi
Fereshteh Bagheri
Atefeh Mirzakhani
$ digital_root 127
USER: find_digital_root() is called for n = 127
KERNEL: sys_find_digital_root() is called for n = 127
digital root of number 127 is: 1
$
```

پیاده سازی فراخوانی های سیستمی

1. پیاده سازی فراخوانی سیستمی کپی کردن فایل

در ابتدا باید عنوان کرد که در صورت وجود فایل مقصد از قبل برنامه با ارور مواجه می‌شود.

ابتدا شماره سیستم کال مورد نظر را در syscall.h تعریف می‌کنیم:

```
#define SYS_copy_file 23
```

سپس شناسه فراخوانی سیستمی را در فایل user.h قرار می‌دهیم:

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

همانطور که در صورت پروژه گفته شده است در صورت موفقیت فراخوانی سیستمی عدد 0 و در غیر اینصورت منفی یک را برمی‌گرداند و به همین دلیل مقدار بازگشتی آن را int در نظر گرفتیم.

سپس تعریف این تابع را در فایل usys.s به شکل زیر قرار می‌دهیم:

```
SYSCALL(copy_file)
```

سپس declaration تابع را در فایل syscall.c مینویسیم:

```
Extern int sys_copy_file(void);
```

سپس شماره مربوط به فراخوانی سیستمی در سطح هسته را به این تابع مپ میکنیم. برای این کار کافیست در فایل syscall.c در آرایه syscalls شناسه فراخوانی و تابع مربوطه را به صورت زیر اضافه کنیم:

```
[SYS_copy_file] sys_copy_file
```

حال از آنجا که این تابع مربوط به فراخوانی های مربوط به فایل هاست تعریف این تابع را در فایل sysfile.c قرار می‌دهیم:

```c
int sys_copy_file(void)
{
  char *path_src;
  char *path_des;
  struct file *f_src;
  struct file *f_des;
  struct inode *ip_src;
  struct inode *ip_des;
  if (argstr(0, &path_src) < 0 || argstr(1, &path_des) < 0)
    return -1;

  begin_op();
  ip_src = namei(path_src);
  ip_des = namei(path_des);
  if (ip_src == 0 || ip_des != 0 || (ip_src->type != T_FILE) || (ip_src == ip_des))
  {
    end_op();
    return -1;
  }
  ilock(ip_src);
  ip_des = create(path_des, T_FILE, 0, 0);

  if ((f_src = filealloc()) == 0 || (f_des = filealloc()) == 0)
  {
    iunlockput(ip_des);
    iunlockput(ip_src);
    end_op();
    return -1;
  }
  iunlock(ip_des);
  iunlock(ip_src);
  end_op();
  f_src->type = FD_INODE;
  f_src->ip = ip_src;
  f_src->off = 0;
  f_src->readable = 1;

  f_des->type = FD_INODE;
  f_des->ip = ip_des;
  f_des->off = 0;
  f_des->writable = 1;
  int r = copyfile(f_src, f_des);
  fileclose(f_src);
  fileclose(f_des);
  return r;
}
```

سپس declaration تابع copyfile را در فایل defs.h قرار داده (چون تابع مربوط به کار با فایل است) و definition آن را در فایل file.c می‌نویسیم.

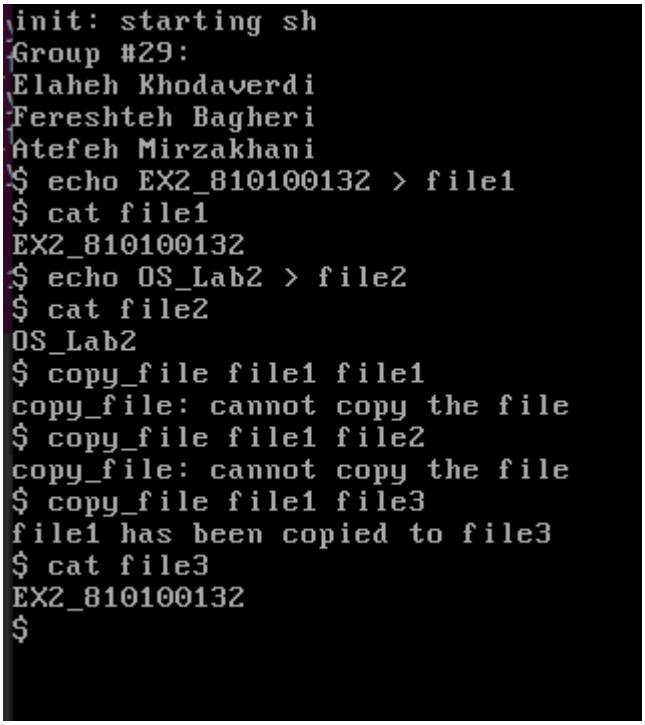
```c
int copyfile(struct file *src, struct file *dst)
{
  int r1, r2;
  if (src->readable == 0 || dst->writable == 0)
    return -1;
  if (src->type == FD_INODE && dst->type == FD_INODE)
  {
    begin_op();
    ilock(src->ip);
    ilock(dst->ip);

    char buf[256];
    while ((r1 = readi(src->ip, buf, src->off, 256)) > 0)
    {
      src->off += r1;
      if ((r2 = writei(dst->ip, buf, dst->off, r1)) > 0)
        dst->off += r2;
      if (r2 < 0)
        break;
      if (r2 != r1)
        panic("short filewrite");
    }
    iunlock(src->ip);
    iunlock(dst->ip);
    end_op();
    return dst->off == src->off ? 0 : -1;
  }
  panic("copyfile");
}
```

برای تست کردن این فراخوانی سیستمی یک برنامه سطح کاربر نوشته و سپس آن را به UPROGS و EXTRA در Makefile اضافه می‌کنیم.

```c
#include "types.h"
#include "user.h"
#include "fcntl.h"
#include "stat.h"
int main(int argc, char * argv[]){
    if (argc < 3){
        printf(1, "copy_file: 2 args required\n");
        exit();
    }
    char * src = argv[1];
    char * des = argv[2];
    if (copy_file(src,des) < 0){
        printf(1,"copy_file: cannot copy the file\n");
        exit();
    }
    printf(1,"%s has been copied to %s\n", src, des);
    exit();
}
```

مثال:



2. پیاده سازی فراخوانی سیستمی تعداد uncle های پردازه

شماره سیستم کال مورد نظر را در syscall.h تعریف می کنیم:

```
#define SYS_get_uncle_count 24
```

حال declaration تابع را در syscall.c می نویسیم و آن را به آرایه مپ شماره سیستم کال به تابعش اضافه می کنیم:

```
extern int get_uncle_count(void);
```

تابع قابل دسترسی توسط کاربر را در فایل user.h تعریف می کنیم:

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

این تابع را در فایل usys.S نیز تعریف می کنیم:

```
SYSCALL(get_uncle_count)
```

برای پیدا کردن عمو های یک پردازه، ابتدا پدر بزرگ آن را پیدا می کنیم بعد روی همه پردازه ها پیمایش می کنیم و چک می کنیم که آیا پدر آنها با پدر بزرگ گفته شده برابر است یا نه. برای این کار باید به ptable که در فایل proc.c به صورت زیر تعریف شده است دسترسی داشته باشیم:
در این فایل proc یک آرایه از تمام پردازه ها است.

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

برای استفاده از ptable تابع زیر را در فایل proc.c می نویسیم و آن را در defs.h نیز تعریف می کنیم تا در فایل sysproc.c قابل دسترسی باشد:

```
int get_child_count(void){
  struct proc* curr = myproc();
  struct proc* grand_parent = curr->parent->parent;
  int child_count = 0;
   for (int i=0; i < NPROC; i++) {
    if (ptable.proc[i].parent == grand_parent) {
      child_count++;
    }
   }
   return child_count;
}
```

تعریف تابع در defs.h:

```
int         get_child_count(void);
```

از آنجایی که سیستم کال get_uncle_count به پردازه ها مربوط است تعریف آن را در sysproc.c می نویسیم:

```
int
sys_get_uncle_count(void)
{
  int child_count = get_child_count();
  return child_count-1;
}
```

برای نوشتن تست یک برنامه سطح کاربر می نویسیم که یک پردازنده سه پردازه فرزند fork می کند و یکی از آنها نیز یک فرزند ایجاد می کند و سیستم کال اضافه شده برای آن صدا زده می شود. برای اینکار تابع را در فایل get_uncle_count_test.c می نویسیم و آن را به makefile هم اضافه می کنیم تا کاربر بتواند آن را اجرا کند:

xv6-public-master > C get_uncle_count_test.c

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


int main() {
    int child1, child2, child3;
    int grandchild;

    child1 = fork();
    if (child1 == 0) {
        printf(1, "Child 1 forked successfully (PID: %d)\n", getpid());
        sleep(100);
        grandchild = fork();
        if (grandchild == 0) {
            printf(1, "Grandchild process forked successfully (PID: %d)\n", getpid());
            int uncle_count = get_uncle_count();
            printf(1, "Uncle count: %d\n", uncle_count);
            exit();
        }
        wait();
        exit();
    }

    child2 = fork();
    if (child2 == 0) {
        printf(1, "Child 2 process forked successfully (PID: %d)\n", getpid());
        sleep(200);
        exit();
    }

    child3 = fork();
    if (child3 == 0) {
        printf(1, "Child 3 process forked successfully (PID: %d)\n", getpid());
        sleep(300);
        exit();
    }
    wait();
    wait();
    wait();
    exit();
}

```

با اجرای این برنامه به خروجی زیر می رسیم:

```
init: starting sh
Group #29:
Elaheh Khodaverdi
Fereshteh Bagheri
Atefeh Mirzakhani
$ get_uncle_count_test
Child 1 forked successfully (PID: 4)
Child 2 process forked successfully (PID: 5)
Child 3 process forked successfully (PID: 6)
Grandchild process forked successfully (PID: 7)
Uncle count: 2
```

3 .پیاده سازی فراخوانی سیستمی طول عمر پردازه

شماره سیستم کال مورد نظر را در syscall.h تعریف می کنیم:

```
#define SYS_get_process_lifetime 25
```

حال declaration تابع را در syscall.c می نویسیم و آن را به آرایه مپ شماره سیستم کال به تابعش اضافه می کنیم:

```
extern int sys_get_process_lifetime(void);
[SYS_get_process_lifetime] sys_get_process_lifetime,
```

تابع قابل دسترسی توسط کاربر را در فایل user.h تعریف می کنیم:

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

این تابع را در فایل usys.S نیز تعریف می کنیم:

```
SYSCALL (get_process_lifetime)
```

برای محاسبه مدت زمان زندگی یک پردازه از زمان به وجود آمدن تا زمان صدا کردن این فراخوانی سیستمی نیاز به زمان ایجاد آن پردازه داریم برای این منظور متغیر creation_time را به struct proc در فایل proc.h اضافه می‌کنیم سپس در فایل proc.c در تابع fork این متغیر را مقداردهی میکنیم:

```
np->creation_time=ticks;
```

که یک تیک (tick) یک واحد زمانی است که توسط تایمر سخت افزاری روی CPU تعریف می شود.

در فایل proc.c تابع int get_process_lifetime(void) را به صورت زیر پیاده‌سازی کردیم:

```
int
get_process_lifetime(void) {
  return (sys_uptime() - myproc()->creation_time);
}
```

که در آن تفاوت مقدار زمان حال و زمان ایجاد پردازه مورد نظر که همان طول عمر پردازه می‌باشد را برمی گردانیم.

و برای پیاده‌سازی این تابع sys_uptime را در proc.h صدا میکنیم:

```
extern int sys_uptime(void);
```

همچنین تابع را در defs.h تعریف میکنیم:

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

در فایل sysproc.c نیز بدنه‌ی تابع int sys_get_process_lifetime(void قرار دارد که در آن تابع get_process_lifetime() که در کرنل قرار دارد، فراخوانی می‌شود:

```
int sys_get_process_lifetime(void) {
  return get_process_lifetime();
}
```

برای تست این سیستم کال یک برنامه سطح کاربر می نویسیم که یک پردازه یک پردازه فرزند fork می کند و فرزند 10 ثانیه صبر می‌کند ((sleep(10) و پردازه والد صبر میکند تا پردازه فرزند exit شود و طول عمر آنها را بعد از sleep و wait چاپ می کنیم برای این کار تابع را در فایل get_process_lifetime.c می نویسیم و آن را به makefile در قسمت های UPROGS و EXTRA اضافه می کنیم تا کاربر بتواند آن را اجرا کند:

get_process_lifetime.c

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

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

    if (pid < 0) {
        printf(1, "Fork failed\n");
        exit();
    }
    else if (pid == 0) {
        sleep(1000);
        printf(1, "Child process with PID: %d has lifetime: %d deciseconds\n",getpid(), get_process_lifetime());
        exit();
    }
    else {
        wait();
        printf(1, "Parent process with PID: %d has lifetime: %d deciseconds\n",getpid(), get_process_lifetime());
        exit();
    }
}
```

با اجرای این برنامه به خروجی زیر می‌رسیم:

```
init: starting sh
Group #29:
Elaheh Khodaverdi
Fereshteh Bagheri
Atefeh Mirzakhani
$ get_process_lifetime
Child process with PID: 4 has lifetime: 1000 deciseconds
Parent process with PID: 3 has lifetime: 1005 deciseconds
$ _
```