آزمایشگاه سیستم عامل

گزارش کار پروژه دوم

اعضای گروه:

امیرحسین عارف زاده	810101604
مهدی نایینی	810101536
کیارش خراسانی	810101413

Repository : https://github.com/Amir-rfz/OS-Lab

Latest Commit : b989ca9024d6ffc66be91a1b0ce4a8b4ead41098

مقدمه

1. با تحلیل فایلهای موجود در متغیر ULIB در 6xv، توضیح دهید که چگونه این کتابخانه ها از فراخوانی های سیستمی بهره میبرند؟ همچنین، دلیل استفاده از این فراخوانی ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه ها را شرح دهید.

همان طور که در عکس قابل مشاهده است سورس فایل از ۴ فایل object تشکیل شده است.

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

- **Ulib.c**

در این فایل توابع زیر تعریف شده اند:

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

که تنها در توابع stat , gets از فراخوانی های سیستمی استفاده شده است.

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 در این تابع از فراخوانی سیستمی read در کد استفاده شده است که هدف آن خواندن از ورودی استاندارد است.

stat: هدف این تابع این است که به متادیتا یک فایل دسترسی پیدا کنیم. فراخوانی های سیستمی open و fstat و close در آن استفاده شده که هدف آن باز کردن فایل (open) بررسی اطلاعات موجود در فایل (fstat) و بستن آن (close) است.

- **usys.S**

این فایل حاوی کد اسمبلی است که با استفاده از آن usys.o ساخته میشود.

همان طور که در تصویر مشاهده میکنید:

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

ابتدا تمامی فراخوانی ها به صورت SYSCALL (name) ساخته میشود و در واقع برای فراخوانی آن ها از این خط که در بالای کد defined شده استفاده میشود. در کد قابل مشاهده است که این فراخوانی ها گلوبال تعریف شده و آیدی این فراخوانی در رجیستر eax نوشته می شود. از آنجایی که شماره فراخوانی سیستمی 64 است مقدار $T_SYSCALL برابر با ۶۴ خواهد بود به همین دلیل در هنگام ایجاد software interrupt رخ میدهد.

- **printf.c**

در این فایل توابع putc, printint, printf تعریف شده اند و تنها تابع putc از فراخوانی سیستمی(فراخوانی سیستمی write) استفاده می کند چون می خواهد روی کنسول کاراکتر چاپ کند و مجبور است برای دسترسی به fd مورد نیاز به kernel mode برود.

- **umalloc.c**

در این فایل توابع malloc و freeو morecore تعریف شده اند.

در free و malloc که همانطور که واضح است مسئولیت تخصیص و آزاد سازی حافظه را دارا هستند فراخوانی سیستمی استفاده نشده ولی در تابع morecore از فراخوانی سیستمی sbrk به جهت افزایش حافظه پردازه استفاده میشود.

به همین دلیل تابع malloc از morecore استفاده میکند تا بتواند حافظه پردازه را افزایش دهد.

2. فراخوانی های سیستمی تنها روش برای تعامل برنامه های کاربر با کرنل نیستند. چه روشهای دیگری در لینوکس وجود دارند که برنامههای سطح کاربر میتوانند از طریق آنها به کرنل دسترسی داشته باشند؟ هر یک از این روشها را به اختصار توضیح دهید.

یکی از روش های دسترسی به هسته ایجاد interrupt است که دو نوع نرم افزاری و یا سخت افزاری دارد.
نوع سخت افزاری معمولا توسط دستگاه های I/O فرستاده می شوند و توسط دستگاه های I/O مثل کیبورد یا موس دریافت میشود(مثلا ctr+c) چون توسط کاربر در هر لحظه میتوانند اتفاق بیفتند از نوع asynchronous هستند.
دسته دیگر interrupt های نرم افزاری هستند که به trap نیز معروفند و به صورت synchronous توسط خود برنامه ایجاد میشوند.
این interrupt ها انواع مختلفی دارند:

System calls: مثل read,write, open, close که به طور کامل در صورت پروژه توضیح داده شده
Exceptions: به طور مثال میتوان به تقسیم به صفر یا پوینتر به آدرس حافظه غیر مجاز اشاره کرد.
Signals: به طور مثال SIGKILL که برای قطع اجرای برنامه به صورت ناگهانی و در مواردی گرفتن رشته برنامه توسط کرنل، یا SIGINT که برای متوقف کردن یک برنامه با استفاده از کلیدهای Ctrl+C استفاده می‌شود. SIGKILL و SIGTERM که برای ارسال سیگنال پایان به یک برنامه استفاده می‌شود.

Memory Mapped I/O: در این روش آدرس hardware به فضای آدرس کاربر مپ می شود که باعث ایجاد توانایی دسترسی مستقیم برنامه سطح کاربر به hardware میشود. این روش معمولا برای درایور های دستگاه و دسترسی های سطح پایین سخت افزار استفاده میشود.

Netlink sockets: میتوان سوکت هایی را برای ارتباط کرنل با کاربر ایجاد کرد.

Shared memory: با این روش میتوان یک memory segment را با کرنل یا پردازه های دیگر به اشتراک گذاشت.

Proc file system: برنامه سطح کاربر میتواند اطلاعات مربوط به کرنل را از آنجا دریافت کند و عملیات های خاصی را انجام دهد.

Kernel Modules:برنامه های سطح کاربر میتوانند آن ها را لود و با آنها ارتباط برقرار کنند. Kernel modules درواقع بخش هایی از کد هایی هستند که میتوانند به صورت دینامیک در کرنل اضافه شوند.

سازوکار اجرای فراخوانی سیستمی در 6xv

3. آیا باقی تله ها را نمی‌توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟

خیر. سیستم عامل اجازه نمیدهد و protection exception ایجاد میشود.

دلیل این موضوع سه چیز است:

۱. امنیت: اگر کاربر بتواند به تمامی تله ها (به طور مثال تله های مربوط به سخت افزار یا دستورات ویژه) دسترسی پیدا کند این امکان ایجاد میشود که کاربر یک کد malicious را اجرا کند و عملکرد سیستم را با دسترسی به سخت افزار یا آسیب رساندن به اطلاعات دستگاه مختل کند.

۲. پایداری: تله های سطح کرنل به طور معمول برای رسیدگی به عملیات های سطح پایین سیستم هستند که برای عملکرد پایدار سیستم عامل لازم هستند. اگر برنامه های سطح کاربر بتوانند با آن ها مداخله کنند میتواند باعث ناپایداری سیستم و یا حتی crash کردن آن شود.

محافظت از منابع: منابع مهم سیستم مثل مموری یا درایور های سیستم باید از دسترسی نامجاز محافظت شوند. به همین دلیل تله های سطح کرنل ساخته شده اند که فقط کرنل بتواند این منابع را مدیریت کند.

به همین دلیل برای بعضی دسترسی ها به DPL_KERNEL نیاز است.

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

برای هر پردازه دو پشته کاربر و هسته وجود دارد که وقتی پردازه در کرنل mode می رود وارد پشته هسته میشود. اگر یک تله فعال شود ما نیاز پیدا میکنیم از پشته هسته استفاده کنیم زیرا وارد kernel mode میشوید اما در انجام متغیر های کاربر و شماره پردازه ای که در حال اجرا بود گم میشود به همین دلیل لازم است آنها را به ترتیب در متغیر ss و eps ذخیره و در پشته آن ها را push کنیم.

به طبع وقتی رسیدگی به تله تمام شود پردازه دوباره به سطح کاربر برمیگردد و می توانیم مقدار متغیر های محلی و آدرس برگشت را از متغیر های ss و eps بازیابی کنیم.

با توجه به توضیحات واضح است هنگام تغییر سطح دسترسی نیاز به ذخیره این متغیر ها در پشته داریم و در غیر این صورت نیازی نیست.

بخش سطح بالا و کنترل کننده زبان سی تله

5. در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در ()argptr بازه آدرس ها بررسی می گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی باز ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی ()sys_read اجرای سیستم را با مشکل روبرو سازد.

توابع دسترسی به فراخوانی های سیستمی عبارت اند از argptr, argstr, argint.

argint(n, &ip): آرگومان n ام را به عنوان integer، به تابع fetchint ارسال میکند تا fetch شود.اگر ورودی تابع نادرست باشد 1- را برمیگرداند. کاربرد آن برای وقتی است که systemcall یک آرگومان integer نیاز دارد. مکان این آرگومان در پشته با توجه به قسمت های قبل و صورت پروژه ip = eps + 4 + 4*n است.
این تابع ابتدا چک میکند که آیا آدرس ارسال شده ۴ بایت در بازه حافظه پردازه است یا خیر، اگر نبود ارور برمیگرداند و در غیر این صورت آرگومان دوم تابع مقدار دهی میشود.

argstr(n, &pp): آرگومان n ام را به عنوان string، که یک پوینتر به آرایه از از character ها است به تابع fetchstr ارسال میکند تا fetch شود.اگر ورودی تابع نادرست باشد 1- را برمیگرداند. کاربرد آن برای وقتی است که systemcall یک آرگومان string نیاز دارد. مکان این آرگومان در پشته با توجه به قسمت های قبل و صورت پروژه pp = eps + 4 + 4*n است.
این تابع ابتدا چک میکند که آیا آدرس ارسال شده در بازه حافظه پردازه است یا خیر، اگر نبود ارور برمیگرداند و در غیر این صورت آرگومان دوم تابع برابر اشاره گر می شود و تا دیدن کاراکتر نال که نشان گر پایان رشته است پیش می رود اگر آن را دید که طول رشته برگردانده میشود در صورتی که به انتهای حافظه رسیدیم ولی رشته تمام نشد، عملیات ناموفق بوده و ۱- برگردانده میشود.

argptr(n, &app, size): آرگومان n ام را به عنوان پوینتر fetch میکند و سپس چک میکند که آیا سایز پوینتر در بازه حافظه برنامه است یا خیر اگر بود آرگومان مقدار دهی میشود و در غیر این صورت ۱- برگردانده میشود. اگر ورودی نادرست باشد ۱- برگردانده میشود.

همانطور که قابل مشاهده است تمامی این توابع ابتدا چک میکنند آدرس گفته شده در بازه حافظه پردازه باشد تا پردازه ای نتواند به حافظه پردازه دیگری دسترسی پیدا کند و اطلاعات آن را خراب کند.
مشکل امنیتی ای که میتواند ایجاد کند این است که اگر ورودی تابع طولش بیش از حد مجاز باشد و این چک انجام نشود در اطلاعات پردازه های دیگر تداخل ایجاد می شود که میتواند به پردازه های اصلی سیستم آسیب برساند.

مثالی از read_sys() که به دلیل استفاده نادرست به پردازه های دیگر آسیب خواهد زد.

```
int
sys_read(void) {
    int fd;
    char *buf;
    int nbytes;

    argint(0, &fd);
    argptr(1, &buf, nbytes);
    argint(2, &nbytes);

    return read(fd, buf, nbytes);
}

```

همانطور که در عکس قابل مشاهده است بدون چک کردن memory range ها تابع های arg صدا شده که در صورتی که طول buf بیش از اندازه حافظه پردازه باشد می تواند حافظه برنامه های دیگر را خراب کند.
یا اگر یک attacker آدرس پونتری به حافظه کرنل را بدهد مشکلات امنیتی زیادی مثل خواندن اطلاعات غیر مجاز یا خراب کردن حافظه کرنل ایجاد میشود.
به همین دلیل باید به صورت زیر چک شود که range های حافظه درست است.

```
int
sys_read(void) {
    int fd;
    char *buf;
    int nbytes;

    if (argint(0, &fd) < 0) return -1;
    if (argint(2, &nbytes) < 0) return -1;
    if (argptr(1, &buf, nbytes) < 0) return;

    return read(fd, buf, nbytes);
}

```

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

ابتدا برنامه سطح کاربر گفته شده به نام pid را مینویسیم و به Makefile اضافه میکنیم که این برنامه شماره پردازه فعلی را با استفاده از سیستم کال ()getpid چاپ می کند.

```c
pid.c > ...
#include "types.h"
#include "user.h"
int main(int argc, char* argv[]) {
    int pid = getpid();
    printf(1, "current Process ID: %d\n", pid);
    exit();
}
```

حال سیستم عامل را بالا میاوریم. سپس یک point break در ابتدای تابع syscall (خط 140)می گذاریم و برنامه سطح کاربر را اجرا میکنیم. پس از آنکه به point break رسید، دستور bt را اجرا میکنیم که تصویر خروجی به شکل زیر است:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137       struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:137
#1  0x80106f8d in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80106d24 in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

دستور bt که مخفف backtrace است call stack برنامه در لحظه کنونی را نشان می دهد. هر تابع که صدا زده می شود یک stack frame 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 stack frame را نشان می دهد که به ترتیب از درونی ترین frame شروع به توضیح دادن میکنیم:

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

- Trap : در این تابع ابتدا بررسی میشود که number trap 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.

- 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] syscall 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 c.syscalls تعریف شده که شماره هر سیستم کال را به تابع مربوط به آن مپ میکند.)

از آنجایی که در حال حاضر در درونی ترین الیه 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.
(gdb)
```

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 نمی باشد. 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 2, syscall () at syscall.c:140
140       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$1 = 5
(gdb)
```

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 2, syscall () at syscall.c:140
140       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$4 = 1
(gdb)
```

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

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, syscall () at syscall.c:140
140       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$5 = 3
(gdb)
```

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 2, syscall () at syscall.c:140
140       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$6 = 12
(gdb)
```

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:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, syscall () at syscall.c:140
140       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$7 = 7
(gdb)
```

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

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, syscall () at syscall.c:140
140       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$8 = 11
(gdb)
```

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 2, syscall () at syscall.c:140
140       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$9 = 16
(gdb)
```

خروجی دستور سطح کاربر اجرا شده:

```
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ pid
current Process ID: 3
$ _
```

ارسال آرگومان‌های فراخوانی‌های سیستمی

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

```
int create_palindrome(void)
```

این ورودی تابع باید در اصل int باشد ولی از آنجا که می خواهیم آرگومانها را با استفاده از رجیسترها پاس بدهیم، پس در خود تابع ورودی ای نمی گیریم و از argint استفاده نمی کنیم.

حال در ادامه تعریف این تابع را در usys.S انجام می دهیم:

```
SYSCALL(create_palindrome)
```

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

```
#define SYS_create_palindrome 22
```

حال باید تابع در سطح کرنل را اضافه کنیم. ابتدا تعریف تابع را در syscall.c می نویسیم.

```
extern int sys_create_palindrome(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 pointer 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_create_palindrome] sys_create_palindrome;
```

در فایل proc.c بدنه‌ی تابع int create_palindrome(int n) را تعریف می‌کنیم که منطق این برنامه در آن قرار دارد و به ازای گرفتن یک عدد، palindrome آن را چاپ می کند.

```
void create_palindrome(int num) {
  int temp = num;
  int answer = num;
  while (temp != 0) {
    answer = (answer * 10) + (temp % 10);
    temp /= 10;
  }
  cprintf("%d\n", answer);
}
```

سپس در فایل sysproc.c بدنه‌ی تابع int sys_create_palindrome(void) را تعریف می‌کنیم در این تابع، فقط تابع create_palindrome که در کرنل است را صدا می زنیم و آرگومان ورودی را با استفاده از رجیستر ebx پاس می‌دهیم.

```
int sys_create_palindrome(void)
{
  int number = myproc()->tf->ebx;
  cprintf("Kernel: Executing palindrome generation system call with input: %d\n", number);
  create_palindrome(number);

  return 0;
}
```

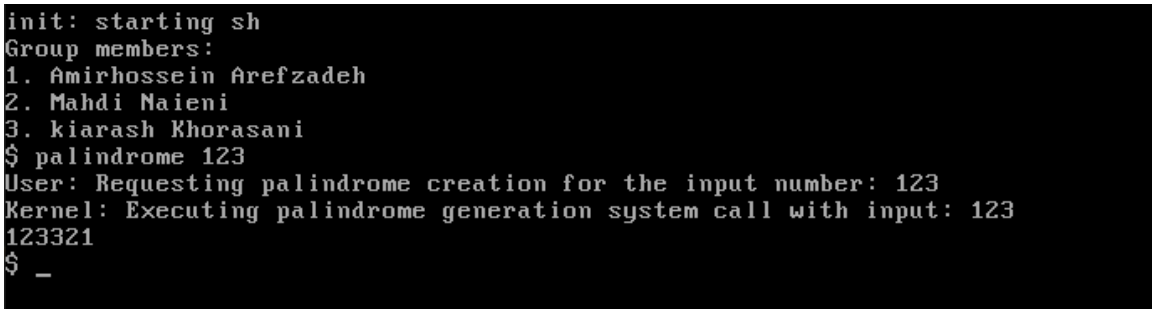
تعریف تابع در defs.h:

int create_palindrome (int);

حال برای برنامه ی سطح کاربر نیاز است تا سیستم کال امتحان شود. برای این کار فایل create_palindrome.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 در makefile در قسمت های UPROGS و EXTRA اضافه می کنیم تا کاربر بتواند آن را اجرا کند:

```
#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 {
            printf(2, "Error: Too many arguments!\n");
        }
        exit();
    }

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

    asm volatile(
        "movl %%ebx, %0;"
        "movl %1, %%ebx;"
        : "=r" (last_ebx_value)
        : "r" (number)
    );

    printf(1, "User: Requesting palindrome creation for the input number: %d\n", number);
    create_palindrome();

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

    exit();
}
```

با اجرای این برنامه به خروجی زیر می‌رسیم:



پیاده سازی فراخوانی های سیستمی

1. پیاده سازی فراخوانی سیستمی انتقال فایل

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

```
#define SYS_move_file 23
```

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

```
int move_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(move_file)
```

سپس declaration تابع را در فایل syscall.c مینویسیم:

```
extern int sys_move_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_move_file] sys_move_file,
```

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.

نکته: 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 .

سپس کافیت فایل مبدا را حذف کنیم

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

تابع انتقال فایل :

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

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

  char *filename = src;
  for (char *p = src; *p; p++) {
    if (*p == '/')
      filename = p + 1;
  }

  char dst[512];
  safestrcpy(dst, dest_dir, sizeof(dst));
  int len = strlen(dst);
  if (dst[len - 1] != '/')
    safestrcpy(dst + len, "/", sizeof(dst) - len);
  safestrcpy(dst + strlen(dst), filename, sizeof(dst) - strlen(dst));

  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, sizeof(buffer))) > 0)
{
  if (filewrite(fdst, buffer, n) != n)
  {
    end_op();
    return -1;
  }
}


fileclose(fsrc);
fileclose(fdst);

remove_moved_file(src);

end_op();

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

تابع حذف فایل موجود:

```
int
remove_moved_file(char* path)
{
  struct inode *ip, *dp;
  struct dirent de;
  char name[DIRSIZ];
  uint off;

  begin_op();
  if((dp = nameiparent(path, name)) == 0){
    end_op();
    return -1;
  }

  ilock(dp);

  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
    goto bad;

  if((ip = dirlookup(dp, name, &off)) == 0)
    goto bad;
  ilock(ip);

  if(ip->nlink < 1)
    panic("unlink: nlink < 1");
  if(ip->type == T_DIR && !isdirempty(ip)){
    iunlockput(ip);
    goto bad;
  }

  memset(&de, 0, sizeof(de));
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("unlink: writei");
  if(ip->type == T_DIR){
    dp->nlink--;
    iupdate(dp);
  }
  iunlockput(dp);

  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);

  end_op();

  return 0;

bad:
  iunlockput(dp);
  end_op();
  return -1;
}
```

نوشتن برنامه سطح کاربر برای تست کردن این فراخوانی سیستمی:

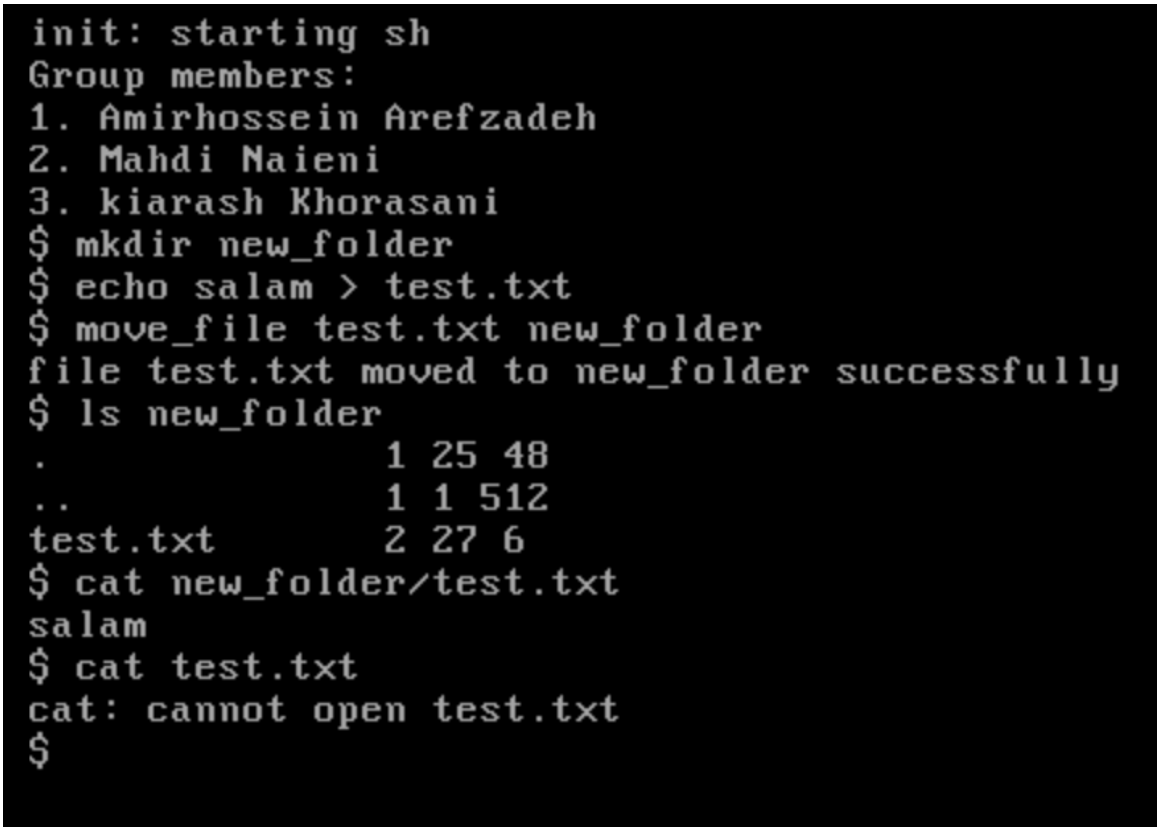
```c
#include "types.h"
#include "user.h"
#include "fcntl.h"
#include "stat.h"

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 (move_file(src,dst) < 0)
    {
        printf(1, "move failed\n");
        exit();
    }

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

سپس این برنامه سطح کاربر را به UPROGS و EXTRA در Makefile اضافه می‌کنیم.

```
move_file.c\
```

```
_move_file\
```

نمونه اجرای برنامه :



2. پیاده سازی فراخوانی سیستمی مرتب سازی فراخوانی های یک پردازه

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

```
#define SYS_sort_syscalls 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 sys_sort_syscalls(void);
```

```
[SYS_sort_syscalls] sys_sort_syscalls,
```

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

```
int sort_syscalls(int);
```

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

```
SYSCALL(sort_syscalls)
```

برای مرتب سازی فراخوانی های یک پردازه کاری که میکنیم این است در ابتدا یک struct جدید به نام syscall_info تعریف کرده که شامل شماره آن سیستم کال به همراه تعداد تکرار های آن است.
حال کافیت در proc struct لیستی از این syscall_info ها را نگه داریم که در آن ها مقدار تعداد آن ها را با MAX SYSCALLS + 1 مقدار دهی اولیه میکنیم .
سپس هربار که یک سیستم کال فرستاده میشود چک میکنیم که آیا تعداد تکرار این سیستم کال در آن پردازه MAX SYSCALLS + 1 هست یا نه اگر MAX SYSCALLS + 1 باشد به این معنی است که این سیستم کال جدید است پس آن را پس آن را به struct آن پردازه اضافه میکنیم و اگه تعداد آن MAX SYSCALLS + 1 نبود به تعداد آن یکی اضافه میکنیم
در نهایت یک تابع مینویسیم که با استفاده از bubble sort فراخوانی های سیستمی بر اساس شماره مرتب کند.

```
#define MAX_SYSCALLS 50
```

```
struct syscall_info {
    int number;         // System call number
    int count;          // Count of how many times this syscall was made
    const char* name;   // Name of syscall
};
```

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:

```
// Initialize syscall_data to zero
for (int i = 0; i < MAX_SYSCALLS; i++) {
  p->syscall_data[i].number = MAX_SYSCALLS + 1;
  p->syscall_data[i].count = 0;
}
```

```
int
sort_syscalls(int pid)
{
    struct proc *p;
    int i, j, flag = 0, index= 1;
    struct syscall_info temp;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == pid) {
        for (i = 0; i < MAX_SYSCALLS - 1; i++) {
          for (j = 0; j < MAX_SYSCALLS - i - 1; j++) {
            if (p->syscall_data[j].number > p->syscall_data[j + 1].number) {
              temp = p->syscall_data[j];
              p->syscall_data[j] = p->syscall_data[j + 1];
              p->syscall_data[j + 1] = temp;
            }
          }
        }
        release(&ptable.lock);
        flag = 1;
      }
    }
    if (flag) {
      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
          for (i = 0; i < MAX_SYSCALLS - 1; i++) {
            if (p->syscall_data[i].count != 0) {
              cprintf("Syscall #%d: Name = %s | Number = %d | Usage Count = %d\n",
                      index++, p->syscall_data[i].name, p->syscall_data[i].number, p->syscall_data[i].count);
            }
          }
          return 0;
        }
      }
    }
    release(&ptable.lock);
    cprintf("Process with PID %d not found\n", pid);
    return -1;
}
```

تعریف تابع در defs.h:

```
int             sort_syscalls(int);
```

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

```
int
sys_sort_syscalls(void)
{
  int pid;
  if (argint(0, &pid) < 0)
    return -1;
  int a = sort_syscalls(pid);
  return a;
}
```

همچنین نیاز است که در فایل syscall.c ما هر فراخوانی جدیدی که اومد رو ذخیره کنیم.

```
int record_syscall(struct proc *p, int num) {
    for (int i = 0; i < MAX_SYSCALLS; i++) {
        if (p->syscall_data[i].number == num) {
            p->syscall_data[i].count++;
            return 1;
        }
        if (p->syscall_data[i].count == 0) {
            break;
        }
    }

    for (int i = 0; i < MAX_SYSCALLS; i++) {
        if (p->syscall_data[i].count == 0) {
            p->syscall_data[i].number = num;
            p->syscall_data[i].count = 1;
            p->syscall_data[i].name = syscall_names[num-1];
            return 0;
        }
    }

    cprintf("Error: syscall_data array full for PID %d\n", p->pid);
    return -1;
}
```

نوشتن برنامه سطح کاربر برای تست کردن این فراخوانی سیستمی:

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

int main(int argc, char *argv[])
{
  if (argc < 2) {
    printf(2, "Usage: sort_syscall <pid>\n");
    exit();
  }

  int pid = atoi(argv[1]);
  if (sort_syscalls(pid) < 0) {
    printf(2, "Failed to sort syscalls for PID %d\n", pid);
  } else {
    printf(1, "Syscalls sorted for PID %d\n", pid);
  }

  exit();
}
```

سپس این برنامه سطح کاربر را به UPROGS و EXTRA در Makefile اضافه می‌کنیم.

```
sort_syscall.c\
```

```
_sort_syscall\
```

نمونه اجرای برنامه :

```
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ sort_syscall 2
Syscall #1: Name = fork ¦ Number = 1 ¦ Usage Count = 1
Syscall #2: Name = wait ¦ Number = 3 ¦ Usage Count = 1
Syscall #3: Name = read ¦ Number = 5 ¦ Usage Count = 15
Syscall #4: Name = exec ¦ Number = 7 ¦ Usage Count = 1
Syscall #5: Name = open ¦ Number = 15 ¦ Usage Count = 1
Syscall #6: Name = write ¦ Number = 16 ¦ Usage Count = 2
Syscall #7: Name = close ¦ Number = 21 ¦ Usage Count = 1
Syscalls sorted for PID 2
$ sort_syscall 2
Syscall #1: Name = fork ¦ Number = 1 ¦ Usage Count = 2
Syscall #2: Name = wait ¦ Number = 3 ¦ Usage Count = 2
Syscall #3: Name = read ¦ Number = 5 ¦ Usage Count = 30
Syscall #4: Name = exec ¦ Number = 7 ¦ Usage Count = 1
Syscall #5: Name = open ¦ Number = 15 ¦ Usage Count = 1
Syscall #6: Name = write ¦ Number = 16 ¦ Usage Count = 4
Syscall #7: Name = close ¦ Number = 21 ¦ Usage Count = 1
Syscalls sorted for PID 2
$ _
```

همانطور که مشاهده میکنیم پس از دوبار فراخوانی sort_syscall فراخوانی read افزایش یافته که نشان دهنده خوانده شدن فراخوانی در دستور بعدی است.

3 .پیاده سازی فراخوانی سیستمی برگرداندن بیشترین فراخوانی سیستم برای یک فرآیند خاص

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

SYS_get_most_invoked_syscall 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_most_invoked_syscall(void);

[SYS_get_most_invoked_syscall] sys_get_most_invoked_syscall,

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

int get_most_invoked_syscall(int);

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

SYSCALL(get_most_invoked_syscall);

برای مرتب سازی فراخوانی های یک پردازه کاری که میکنیم این است در ابتدا یک struct جدید به نام syscall_info تعریف کرده که شامل شماره آن سیستم کال به همراه تعداد تکرار های آن است.
حال کافیت در proc struct لیستی از این syscall_info ها را نگه داریم که در آن ها مقدار تعداد آن ها را با MAX SYSCALLS + 1 مقدار دهی اولیه میکنیم .
سپس هربار که یک سیستم کال فرستاده میشود چک میکنیم که آیا تعداد تکرار این سیستم کال در آن پردازه MAX SYSCALLS + 1 هست یا نه اگر MAX SYSCALLS + 1 باشد به این معنی است که این سیستم کال جدید است پس آن را پس آن را به struct آن پردازه اضافه میکنیم و اگه تعداد آن MAX SYSCALLS + 1 نبود به تعداد آن یکی اضافه میکنیم
در نهایت یک تابع مینویسیم که سیستم کالی را که بیشترین فراخوانی های سیستمی را داشته، برگرداند.

```
#define MAX_SYSCALLS 50
```

```
struct syscall_info {
    int number;         // System call number
    int count;          // Count of how many times this syscall was made
    const char* name;   // Name of syscall
};
```

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:

```
// Initialize syscall_data to zero
for (int i = 0; i < MAX_SYSCALLS; i++) {
  p->syscall_data[i].number = MAX_SYSCALLS + 1;
  p->syscall_data[i].count = 0;
}
```

```
int
get_most_invoked_syscall(int pid)
{
  struct proc *p;
  int i, flag = 0, max= 0, found_index= 0;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->pid == pid) {
      for (i = 0; i < MAX_SYSCALLS - 1; i++) {
        if (p->syscall_data[i].count > max) {
          max = p->syscall_data[i].count;
          found_index = i;
        }
      }
      release(&ptable.lock);
      flag = 1;
    }
  }
  if (flag) {
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == pid) {
        if (max > 0) {
          cprintf("Most invoked syscall: Name = %s | Number = %d | Usage Count = %d\n",
                  p->syscall_data[found_index].name, p->syscall_data[found_index].number, p->syscall_data[found_index].count);
          return p->syscall_data[found_index].number;
        }
        else {
          cprintf("No system calls have been invoked");
          return -1;
        }
      }
    }
  }
  release(&ptable.lock);
  cprintf("Process with PID %d not found\n", pid);
  return -1;
}
```

تعریف تابع در defs.h:

```
int get_most_invoked_syscall(int);
```

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

```
int
sys_get_most_invoked_syscall(void)
{
  int pid;
  if (argint(0, &pid) < 0)
    return -1;
  int output = get_most_invoked_syscall(pid);
  return output;
}
```

همچنین نیاز است که در فایل syscall.c ما هر فراخوانی جدیدی که اومد رو ذخیره کنیم.

```
int record_syscall(struct proc *p, int num) {
    for (int i = 0; i < MAX_SYSCALLS; i++) {
        if (p->syscall_data[i].number == num) {
            p->syscall_data[i].count++;
            return 1;
        }
        if (p->syscall_data[i].count == 0) {
            break;
        }
    }

    for (int i = 0; i < MAX_SYSCALLS; i++) {
        if (p->syscall_data[i].count == 0) {
            p->syscall_data[i].number = num;
            p->syscall_data[i].count = 1;
            p->syscall_data[i].name = syscall_names[num-1];
            return 0;
        }
    }

    cprintf("Error: syscall_data array full for PID %d\n", p->pid);
    return -1;
}
```

نوشتن برنامه سطح کاربر برای تست کردن این فراخوانی سیستمی:

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

int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf(2, "Usage: get_most_syscall <pid>\n");
        exit();
    }
    int pid = atoi(argv[1]);
    if (get_most_invoked_syscall(pid) < 0) {
        printf(2, "Failed to get most invoked syscall for PID %d\n", pid);
    } else {
        exit();
    }
    exit();
}
```

سپس این برنامه سطح کاربر را به UPROGS و EXTRA در Makefile اضافه می‌کنیم.

get_most_syscall.c\
_get_most_syscall\

نمونه اجرای برنامه :

```
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ sort_syscall 2
Syscall #1: Name = fork ¦ Number = 1 ¦ Usage Count = 1
Syscall #2: Name = wait ¦ Number = 3 ¦ Usage Count = 1
Syscall #3: Name = read ¦ Number = 5 ¦ Usage Count = 15
Syscall #4: Name = exec ¦ Number = 7 ¦ Usage Count = 1
Syscall #5: Name = open ¦ Number = 15 ¦ Usage Count = 1
Syscall #6: Name = write ¦ Number = 16 ¦ Usage Count = 2
Syscall #7: Name = close ¦ Number = 21 ¦ Usage Count = 1
Syscalls sorted for PID 2
$ get_most_syscall 2
Most invoked syscall: Name = read ¦ Number = 5 ¦ Usage Count = 34
$
```

همانطور که مشاهده میکنیم پس از فراخوانی get_most_syscall فراخوانی read بیشترین فراخوانی را داشته است.

4 . پیاده سازی فراخوانی سیستمی لیست کردن پردازه ها

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

SYS_list_all_processesl 26

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_list_all_processes(void);

[SYS_list_all_processes] sys_list_all_processes,

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

int list_all_processes(void);

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

SYSCALL(list_all_processes);

برای مرتب سازی فراخوانی های یک پردازه کاری که میکنیم این است در ابتدا یک struct جدید به نام syscall_info تعریف کرده که شامل شماره آن سیستم کال به همراه تعداد تکرار های آن است.
حال کافیت در proc struct لیستی از این syscall_info ها را نگه داریم که در آن ها مقدار تعداد آن ها را با MAX SYSCALLS + 1 مقدار دهی اولیه میکنیم .
سپس هربار که یک سیستم کال فرستاده میشود چک میکنیم که آیا تعداد تکرار این سیستم کال در آن پردازه MAX SYSCALLS + 1 هست یا نه اگر MAX SYSCALLS + 1 باشد به این معنی است که این سیستم کال جدید است پس آن را پس آن را به struct آن پردازه اضافه میکنیم و اگه تعداد آن MAX SYSCALLS + 1 نبود به تعداد آن یکی اضافه میکنیم
در نهایت یک تابع مینویسیم که سیستم کالی را که بیشترین فراخوانی های سیستمی را داشته، برگرداند.

```
#define MAX_SYSCALLS 50
```

```
struct syscall_info {
    int number;         // System call number
    int count;          // Count of how many times this syscall was made
    const char* name;   // Name of syscall
};
```

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:

```
// Initialize syscall_data to zero
for (int i = 0; i < MAX_SYSCALLS; i++) {
  p->syscall_data[i].number = MAX_SYSCALLS + 1;
  p->syscall_data[i].count = 0;
}
```

```
int
list_all_processes(void)
{
  struct proc *p;
  int flag=0, idx=1;

  acquire(&ptable.lock);
  for (idx=1, p = ptable.proc; p < &ptable.proc[NPROC]; p++, idx++) {
      if (p->pid != 0) {
        cprintf("Process #%d: Pid = %d | Syscall Count = %d\n", idx, p->pid, p->syscall_counts);
        flag = 1;
      }
  }
  if (flag) {
    release(&ptable.lock);
    return 0;
  }
  release(&ptable.lock);
  return -1;
}
```

تعریف تابع در defs.h:

intlist_all_processes(void);

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

```
int
sys_list_all_processes(void)
{
  int output = list_all_processes();
  return output;
}
```

همچنین نیاز است که در فایل syscall.c ما هر فراخوانی جدیدی که اومد رو ذخیره کنیم.

```
int record_syscall(struct proc *p, int num) {
    for (int i = 0; i < MAX_SYSCALLS; i++) {
        if (p->syscall_data[i].number == num) {
            p->syscall_data[i].count++;
            p->syscall_counts++;
            return 1;
        }
        if (p->syscall_data[i].count == 0) {
            break;
        }
    }

    for (int i = 0; i < MAX_SYSCALLS; i++) {
        if (p->syscall_data[i].count == 0) {
            p->syscall_data[i].number = num;
            p->syscall_data[i].count = 1;
            p->syscall_data[i].name = syscall_names[num-1];
            p->syscall_counts++;
            return 0;
        }
    }

    cprintf("Error: syscall_data array full for PID %d\n", p->pid);
    return -1;
}
```

نوشتن برنامه سطح کاربر برای تست کردن این فراخوانی سیستمی:

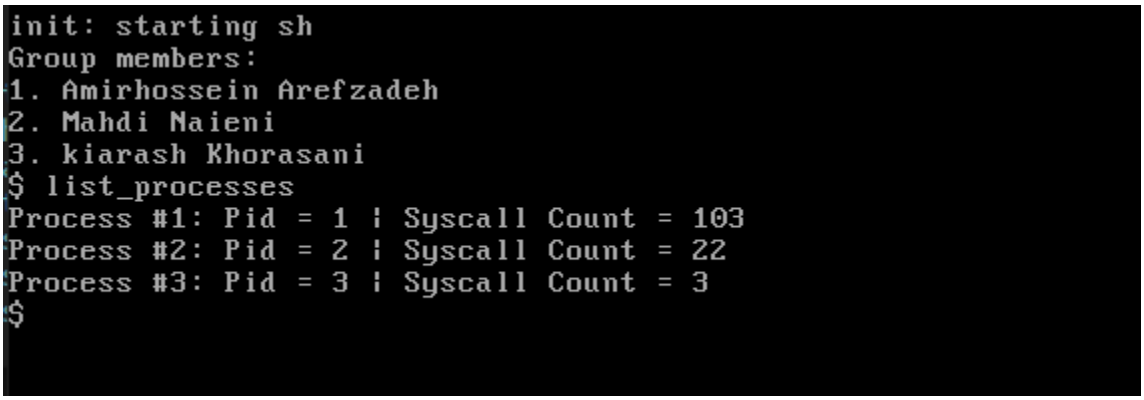
```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[])
{
    if (argc >= 2) {
        printf(2, "Usage: list_processes\n");
        exit();
    }
    if (list_all_processes() < 0) {
        printf(2, "Failed to list all processes\n");
    }
    exit();
}
```

سپس این برنامه سطح کاربر را به UPROGS و EXTRA در Makefile اضافه می‌کنیم.

list_all_processes.c\

_list_all_processes\

نمونه اجرای برنامه :



همانطور که مشاهده میکنیم پس از فراخوانی list_processes لیست process ها را مشاهده می کنیم.