

## گزارش پروژه (۱) آزمایشگاه سیستم عامل

نام استاد: دکتر کارگهی

امیرارسلان شهبازی ۸۱۰۱۰۱۴۵۱

محمدحسین مظهری ۸۱۰۱۰۱۵۲۰

محمد مهدی صمدی ۸۱۰۱۰۱۴۶۵

گروه ۱۱

لینک ریپوزیتوری گیتهاب: <https://github.com/AMIRSH1383/OS-SMS-Lab2.git>

(۱) کتابخانه های سطح کاربر در xv6، برای ایجاد ارتباط میان برنامه های کاربر و کرنل به کار میروند. این کتابخانه ها شامل توابعی هستند که از فراخوانی های سیستمی استفاده میکنند تا دسترسی به منابع سخت افزاری و نرم افزاری سیستم عامل ممکن شود. با تحلیل فایل های موجود در متغیر ULIB در xv6، توضیح دهید که چگونه این کتابخانه ها از فراخوانی های سیستمی بهره میبرند؟ همچنین، دالیل استفاده از این فراخوانی ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه ها را شرح دهید.

این کتابخانه ها به عنوان واسطی میان کاربر و کرنل رفتار میکنند. کتابخانه ها شامل توابعی هستند که از فراخوانی های سیستمی استفاده می کنند تا دسترسی به منابع سخت افزاری و نرم افزاری سیستم عامل ممکن شود.

### نحوه استفاده از فراخوانی های سیستمی

فراخوانی های سیستمی در xv6 به صورت تابع هایی در کتابخانه های سطح کاربر پیاده سازی شده اند. این فراخوانی ها توسط برنامه های کاربر فراخوانی می شوند و سپس به کرنل انتقال می یابند تا عملکردهای سطح پایین سیستم را اجرا کنند. در فایل های موجود در متغیر ULIB، توابعی مانند read، write، fork، exec و غیره تعریف شده اند که هر کدام به یک فراخوانی سیستمی معین متصل هستند.

### دلایل استفاده از فراخوانی های سیستمی

#### ۱. امنیت:

فراخوانی های سیستمی به کرنل اجازه می دهند که دسترسی به منابع حساس سیستم مانند حافظه، فایل ها، و دستگاه ها را کنترل کند. این امر مانع از دسترسی غیرمجاز و سوءاستفاده از منابع می شود.

#### ۲. مدیریت منابع:

فراخوانی های سیستمی امکان مدیریت موثر منابع سیستم را فراهم می کنند. به عنوان مثال، کرنل می تواند تعداد فرآیندهای همزمان را کنترل کرده و حافظه را بهینه تخصیص دهد.

#### ۳. قابلیت حمل:

با استفاده از فراخوانی‌های سیستمی، برنامه‌ها می‌توانند به شکلی مستقل از جزئیات سخت‌افزاری و نرم‌افزاری خاص اجرا شوند. این امر باعث می‌شود که برنامه‌ها بتوانند به راحتی به سیستم‌عامل‌ها و سخت‌افزارهای مختلف منتقل شوند.

## تأثیر فراخوانی‌های سیستمی بر عملکرد و قابلیت حمل

### ۱. عملکرد:

- استفاده از فراخوانی‌های سیستمی ممکن است مقداری سربار ایجاد کند، زیرا تغییر حالت بین کرنل و کاربر نیاز به زمان دارد. با این حال، استفاده بهینه از این فراخوانی‌ها می‌تواند به بهبود عملکرد کلی سیستم کمک کند.

### ۲. قابلیت حمل:

- فراخوانی‌های سیستمی یک واسط یکنواخت برای دسترسی به عملکردهای پایه سیستم‌عامل فراهم می‌کنند. این امر به برنامه‌ها اجازه می‌دهد که بدون نیاز به تغییرات بزرگ در کد، به راحتی بر روی سیستم‌عامل‌های مختلف اجرا شوند.

۲) فراخوانی‌های سیستمی تنها روش برای تعامل برنامه‌های 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.

علاوه بر فراخوانی‌های سیستمی، روش‌های دیگری نیز وجود دارند که برنامه‌های کاربر می‌توانند از طریق آنها به کرنل دسترسی داشته باشند. در ادامه، برخی از این روش‌ها را به اختصار توضیح می‌دهم:

### ۱. مکانیزم `/proc` و `/sys`

- **`/proc`:** یک سیستم فایل مجازی است که اطلاعات مربوط به فرآیندها و سیستم‌عامل را فراهم می‌کند. برنامه‌ها می‌توانند با خواندن یا نوشتن در فایل‌های موجود در `/proc`، اطلاعات مورد نیاز را بدست آورند یا تنظیمات سیستم را تغییر دهند.

○ مثال: خواندن اطلاعات فرآیندها از `/proc/pid/status`.

- **`/sys`:** یک سیستم فایل مجازی دیگر است که دسترسی به اطلاعات سخت‌افزاری و تنظیمات دستگاه‌ها را فراهم می‌کند.

○ مثال: دسترسی به تنظیمات دستگاه‌های USB از طریق `/sys/bus/usb`.

### ۲. نرم‌افزارهای کاربر-کرنل (Kernel-Userland Libraries)

- برخی کتابخانه‌های سطح کاربر به طور مستقیم با کرنل ارتباط برقرار می‌کنند تا عملکردهای خاص را فراهم کنند. به عنوان مثال، `libc` از فراخوانی‌های سیستمی استفاده می‌کند تا توابع سطح بالا مثل `malloc` و `free` را پیاده‌سازی کند.
- مثال: کتابخانه `libc` برای مدیریت حافظه.

### ۳. `IOCTL`

- مکانیزم **ioctl (Input/Output Control)** به برنامه‌های کاربر اجازه می‌دهد تا دستورات خاصی را به دستگاه‌های سخت‌افزاری ارسال کنند. این مکانیزم معمولاً برای دستگاه‌های خاص مثل ترمینال‌ها و دستگاه‌های شبکه استفاده می‌شود.

○ مثال: تنظیم سرعت بادگیر ترمینال با **ioctl** درایور ترمینال.

#### ۴. Netlink Sockets

- سوکت‌های **Netlink** واسطی بین کرنل و فضای کاربر فراهم می‌کنند که برای ارتباطات پیچیده‌تر مثل پیکربندی شبکه استفاده می‌شوند.

○ مثال: استفاده از **Netlink** برای مدیریت تنظیمات شبکه از طریق **iproute2**.

#### ۵. Memory-Mapped I/O (mmap)

- مکانیزم **mmap** به برنامه‌های کاربر اجازه می‌دهد تا بخشی از حافظه را مستقیماً به فضای آدرس خود نگاشت کنند. این روش معمولاً برای دسترسی به حافظه مشترک یا دستگاه‌های سخت‌افزاری استفاده می‌شود.

○ مثال: نگاشت فایل به حافظه برای دسترسی سریع‌تر به داده‌ها.

#### ۶. Signaling

- سیگنال‌ها مکانیزمی هستند که به کرنل اجازه می‌دهند تا رویدادهای خاصی را به برنامه‌های کاربر اطلاع دهند. برنامه‌ها می‌توانند سیگنال‌ها را پردازش کرده و اقدامات مناسب را انجام دهند.

○ مثال: دریافت سیگنال **SIGINT** برای خاتمه دادن به برنامه.

#### ۷. پروتکل‌های خاص

- برخی پروتکل‌های خاص مانند **D-Bus** به برنامه‌های کاربر اجازه می‌دهند تا به سرویس‌های کرنل دسترسی پیدا کنند و با آن‌ها ارتباط برقرار کنند.

○ مثال: استفاده از **D-Bus** برای ارتباط با مدیر جلسه کاربر (session manager).

هر یک از این روش‌ها دسترسی به عملکردهای خاصی از کرنل را فراهم می‌کنند و بسته به نیاز برنامه، می‌توانند به طور موثری مورد استفاده قرار گیرند.

**پرسش ۳: آیا باقی تله‌ها را نمیتوان با سطح دسترسی **DPL\_USER** فعال نمود؟ چرا؟**

خیر، این امر امکان پذیر نیست. همان طور که میدانیم در **xv6** دو سطح **user** , **kernel** موجود هستند. چون سطح دسترسی **DPL\_USER** یک سطح دسترسی کاربر (سطح ۳) است، نباید امکان دسترسی به هسته و اجرای این تله‌ها را داشته باشد. اگر کاربر امکان اجرای این تله‌ها را داشته باشد، به سادگی میتواند به **kernel** دسترسی داشته باشد. در این صورت **protection** نقض می‌شود.

در واقع اگر پردازنده بخواهد یک **interrupt** دیگر را فعال کند، **kernel** اجازه ی این عمل را به او نمیدهد زیرا ممکن است در برنامه سطح کاربر مشکلی وجود داشته باشد و این مشکل به هسته منتقل شود (گسترش یابد و به هسته آسیب بزند) (**protection**) یا خود کاربر

قصد حمله به هسته را داشته باشد (security) که در صورت داشتن این دسترسی ها میتواند به هسته ی سیستم عامل آسیب بزند و هر بخشی از سخت افزار و نرم افزار را مورد تهاجم قرار دهد .

**پرسش ۴ :** در صورت تغییر سطح دسترسی ، **ss** و **esp** روی پشته **push** میشود . در غیر اینصورت **push** نمیشود. چرا؟

دو پشته ی **user stack** و **kernel stack** موجود هستند . موقع فعال شدن یک **trap** ، میخواهیم دسترسی را تغییر دهیم (مثلا از سطح کاربر به سطح هسته برویم ) ، در این زمان نمیتوانیم از پشته قبلی استفاده کنیم . پس باید **ss** و **esp** روی پشته **push** شوند تا هنگام بازگشت آخرین دستوری که اجرا شده را بدانیم و انجام دستورات را از آنجا به بعد از سر بگیریم .

اما هنگامی که تغییری در سطح دسترسی رخ ندهد ، نیازی به **push** کردن آنها نیست زیرا همچنان با همان پشته ی قبلی مشغول به کار هستیم .

**سوال ۵)** در مورد توابع دسترسی به پارامترهای 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 باز ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی (**sys\_read**) اجرای سیستم را با مشکل روبرو سازد.

این بخش سه تابع دارد.

**argint** : برای آن شماره آرگومان مشخص می شود و آدرس یک **int** را برای گرفتن متغیر به آن داده می شود.

**argstr** : دو پارامتر می گیرد. اولی شماره آرگومان است و دومی آدرس یک متغیر از نوع **char \*** است که در آن آرگومان ریخته می شود.

**argptr** : شماره آرگومان، آدرس یک پوینتر و اندازه چیزی که خوانده می شود را می گیرد و در آدرس پوینتر، محتوای آرگومان ریخته می شود.

اگر بازه آدرس ها بررسی نگردد ممکن است تجاوز از بازه ی معتبر پیش بیاید. خارج شدن از بازه معتبر می تواند مشکلاتی را برای ما به وجود بیاورد. مثلا ممکن است بخواهیم چیزی را با **sys\_read** بخوانیم اما اگر آدرسی که می خواهیم از آن بخوانیم از بازه معتبر خارج باشد و ما اعتبار بازه را چک نکرده باشیم، اطلاعات اشتباه را ممکن است بخوانیم و روند برنامه را بهم بریزیم.

**بررسی گام های اجرای فراخوانی سیستم در سطح کرنل توسط gdb :**

ابتدا برنامه ی سطح کاربر را مینویسیم و آن را به **makefile** اضافه می کنیم .

**gdb\_test.c**

```
C gdb_test.c
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int main()
6 {
7     int pid = getpid();
8     printf(1, "Process pid = %d \n", pid);
9     exit();
10 }
```

سپس دستور **make qemu-gdb** را داخل دایرکتوری ای که **OS** در آن قرار دارد در ترمینال وارد می کنیم

در ترمینالی دیگر دستور kernel gdb را میزنیم .

سپس عبارت target remote tcp::26000 را مینویسیم .

سپس c را مینویسیم ( continue )

سپس c cntrl را میفشاریم .

پس از آن break syscall را می نویسیم . ( break point قرار داده می شود )

سپس c را مینویسیم .

متوجه می شویم که برنامه در syscall متوقف می شود .

بعد bt را می نویسیم .

دستور bt یا همان backtrace ، stack call برنامه در این لحظه را نشان می دهد . هر تابع که فراخوانی می شود یک frame stack مخصوص به خودش به آن اختصاص داده می شود که متغیر های محلی و آدرس بازگشت و .. در آن نگهداری می شود .

خروجی bt در هر خط یک frame stack را نشان می دهد که به ترتیب از درونی ترین frame هر یک را بیان می کنیم:

(۱) alltraps : در تابع در ابتدا trapframe مربوط به این trap ایجاد شده و در استک قرار میگیرد . سپس تابع trap() فراخوانی میشود . ( این تابع در trap.c است . )

(۲) trap : در این تابع ابتدا بررسی میشود که trap number متناظر با چه وقفه ای است . پس از آنکه معلوم شد که از نوع system call است ، trapframe مربوط به پردازش ی فعلی را برابر با trapframe قرار داده شده در استک می کند و تابع syscall() را صدا میزند .

(۳) syscall : این تابع eax را از trapframe پردازش فعلی میخواند (این مقدار برابر است با system call مورد نظر ) .

سپس با استفاده از syscalls[num] تابع مربوط به آن سیستم کال را فرا میخواند و خروجی آن را در eax در trapframe پردازش فعلی ذخیره می کند . (آرایه syscalls در ابتدای فایل syscalls.c قرار دارد که شماره هر سیستم کال را به تابع متناظر آن مپ می کند . )

بعد down را مینویسیم . چون در درونی ترین لایه هستیم ، با زدن این دستور به error زیر دچار میشویم ( down ما را به سیستم کال قبلی ای که مارا فراخوانی کرده است میبرد ولی چون اینجا در درونی ترین لایه هستیم ، به ارور می خوریم . )

بعد up را مینویسیم .

```

(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x00000fff0 in ?? ()
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
scheduler () at proc.c:337
337     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
(gdb) break syscall
Breakpoint 1 at 0x80106010: file syscall.c, line 183.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183     struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:183
#1  0x8010749d in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80107236 in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1  0x8010749d in trap (tf=0x8dffefb4) at trap.c:43
43     syscall();
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183     struct proc *curproc = myproc();

```

سپس چند باری C و سپس `print myproc()->tf->eax` را مینویسیم. (چاپ محتوای `eax` که در `tf` است)

تا جایی این کار را ادامه میدهیم که محتوای رجیستر `eax` برابر با شماره سیستم کال `getpid()` شود.

همانطور که مشاهده می شود مقدار رجیستر با شماره سیستم کال `getpid()` مطابقت ندارد.

زیرا وقتی برنامه سطح کاربر را اجرا میکنیم، ابتدا چند عملیات `read` کردن (که شماره آن ۵ است) انجام میشود (۷ مرتبه این اتفاق رخ میدهد).

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print pid
No symbol "pid" in current context.
(gdb) print myproc()->tf->eax
$1 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$2 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$3 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$4 = 5
(gdb) c
Continuing.
```

```

$5 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$6 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$7 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$8 = 1
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$9 = 3
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$10 = 12
(gdb) c
Continuing.

```

عدد ۱ مربوط به سیستم کال fork است که جهت ایجاد پردازش جدید برای برنامه سطح کاربر صدا زده میشود.

عدد ۳ مربوط به سیستم کال wait است که توسط پردازش ی پدر صدا زده میشود که تا پایان کار پردازش فرزند صبر میکند .

عدد ۱۲ مربوط به سیستم کال sbrk است که این سیستم کال به پردازش ایجاد شده ، حافظه اختصاص میدهد.



```

$10 = 12
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$11 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$12 = 11
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$13 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$14 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:183
183      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$15 = 16
(gdb) c
Continuing.

```

عدد ۷ مربوط به سیستم کال exec است که برای اجرای برنامه pid در پردازش ایجاد شده استفاده میشود .

عدد ۱۱ مربوط به سیستم کال getpid() است که انتظار آن را داشتیم .

عدد ۱۶ مربوط به سیستم کال write است که در نهایت خروجی مورد نظر را برای کاربر مینویسد .

## نتیجه

```
SeaBIOS (version 1.15.0-1)
t
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 sta
t 58
Members of Team SMS:
  Mohamad Mahdi Samadi
  Amir Arsalan Shahbazi
  Mohammad Hosein Mazhari
init: starting sh
$ gdb_test
Process pid = 3
$ _
```

## ارسال آرگومان‌های فراخوانی سیستمی

فراخوانی سیستمی create palindrome

در این دستور ما می‌بایست یک ورودی از کاربر گرفته و پالیندروم شده آن را تولید و به کاربر نشان دهیم.

syscall.h		
↑	@@ -20,3 +20,5 @@	
20	20	#define SYS_link 19
21	21	#define SYS_mkdir 20
22	22	#define SYS_close 21
23	+	#define SYS_create_palindrome 22
24	+	

ابتدا در فایل بالا یک سیستم کال جدید به نام پالیندروم و شماره ۲۲ ایجاد می‌کنیم.

```
sysproc.c @@ -89,3 +89,12 @@ sys_uptime(void)
89      89      release(&tickslock);
90      90      return xticks;
91      91      }
92      +
93      + int
94      + sys_create_palindrome(void)
95      + {
96      +     int number = myproc()->tf->ebx; //register after eax
97      +     cprintf("Kernel: sys_create_palindrome() called for number %d\n", number);
98      +     return create_palindrome(number);
99      + }
100     +
```

سپس در فایل sysproc.c یک تابع جدید برای سیستم کال ساخت پالیندروم تشکیل می دهیم. در این تابع با توجه به رفرنس و تابعی که برای سطح کاربر می زنیم، برای گرفتن آرگومان تابع می بایست رجیستر ebx را هدف قرار دهیم.



سپس می گوییم که کرنل این تابع را فراخوانی کرده است و در انتها هم تابع پالیندرومی که بعدا خواهیم نوشت و مسئول محاسبات خواهد بود را صدا می کنیم.

```
proc.c @@ -532,3 +532,23 @@ procdump(void)
532 532      cprintf("\n");
533 533  }
534 534  }

535 +
536 + int
537 + create_palindrome(int num)
538 + {
539 +     int num, reverse = 0, rem, temp;
540 +     //printf("The number is :%d\n",num);
541 +
542 +     temp = num;
543 +
544 +     //loop to find reverse number
545 +     while(temp != 0)
546 +     {
547 +         rem = temp % 10;
548 +         reverse = reverse * 10 + rem;
549 +         temp /= 10;
550 +         num *= 10;
551 +     };
552 +
553 +     return num + reverse;
554 + }
```

در فایل proc.c تابع محاسبه پالیندروم را می نویسیم. روال اینطور است که عددها را هر سری به توانهای ۱۰ تقسیم می کنیم تا هر رقم در بیاید و در نهایت به صورت برعکس شده به رشته عددی خود اضافه می کنیم.

▼ defs.h  			
121	121	void	wakeup(void*);
122	122	void	yield(void);
	123	+ int	create_palindrome(int);
123	124		
124	125	// swtch.S	
125	126	void	swtch(struct context**, struct context*);
..... ↓			

▼ syscall.c  			
		..... ↑	@@ -103,6 +103,7 @@ extern int sys_unlink(void);
103	103	extern	int sys_wait(void);
104	104	extern	int sys_write(void);
105	105	extern	int sys_uptime(void);
	106	+ extern	int sys_create_palindrome(void);
106	107		
107	108	static	int (*syscalls[])(void) = {
108	109	[SYS_fork]	sys_fork,
		..... ↑	@@ -126,6 +127,7 @@ static int (*syscalls[])(void) = {
126	127	[SYS_link]	sys_link,
127	128	[SYS_mkdir]	sys_mkdir,
128	129	[SYS_close]	sys_close,
	130	+ [SYS_create_palindrome]	sys_create_palindrome,
129	131		};
130	132		

یکسری دیفاین کردن ها که باقی مانده بود را در فایل های بالا با توجه به اینکه فراخوانی جدید می سازیم، اضافه می کنیم.

```
create_palindrome.c
*** @@ -0,0 +1,28 @@
1 + #include "types.h"
2 + #include "stat.h"
3 + #include "user.h"
4 +
5 + int main(int argc, char *argv[]){
6 +     if(argc < 2){
7 +         printf(2, "You must enter exactly 1 number!\n");
8 +         exit();
9 +     }
10 +    else
11 +    {
12 +        // We will use ebx register for storing input number
13 +        int saved_ebx, number = atoi(argv[1]);
14 +        //
15 +        asm volatile(
16 +            "movl %%ebx, %0;" // saved_ebx = ebx
17 +            "movl %1, %%ebx;" // ebx = number
18 +            : "=r" (saved_ebx)
19 +            : "r"(number)
20 +        );
21 +        printf(1, "User: create palindrome() called for number: %d\n", number);
22 +        printf(1, "Palindrome of %d is: %d\n", number, create_palindrome());
23 +        asm("movl %0, %%ebx" : : "r"(saved_ebx)); // ebx = saved_ebx -> restore
24 +        exit();
25 +    }
26 +
27 +    exit();
28 + }
```

در گام بعد یک فایل جدید به نام پالیندروم می سازیم که برنامه سطح کاربر خواهد بود.

در این برنامه در صورت وارد کردن صحیح دستور توسط کاربر، ابتدا آرگومان ذخیره شده را در رجیستر برگزیده ذخیره می کنیم.

سپس تابع `create_palindrome` را صدا می زنیم تا نتیجه برایمان محاسبه شود. در نهایت هم محتویات رجیستری که انتخاب کرده بودیم را بازسازی می کنیم.

```
▼ Makefile
185 185      _wc\
186 186      _zombie\
187 +      _create_palindrome\
187 188
188 189      fs.img: mkfs README $(UPROGS)
189 190      ./mkfs fs.img README $(UPROGS)
189 190
189 190      @@ -251,7 +252,7 @@ qemu-nox-gdb: fs.img xv6.img .gdbinit
189 190
251 252      # check in that version.
252 253
253 254      EXTRA=\
254 -      mkfs.c ulib.c user.h cat.c echo.c decode.o encode.o history.o forktest.c grep.c kill.c\
255 +      mkfs.c ulib.c user.h cat.c echo.c decode.o encode.o history.o create_palindrome.o forktest.c grep.c kill.c\
255 256      ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
256 257      printf.c umalloc.c\
257 258      README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
257 258
257 258
> create_palindrome.c
> proc.c
> sysproc.c
▼ usys.S
29 29      SYSCALL(sbrk)
30 30      SYSCALL(sleep)
31 31      SYSCALL(uptime)
32 + SYSCALL(create_palindrome)
```

در پایان هم به میک فایل و فایل usys.S ملزومات اجرای این سیستم کال را اضافه کرده و ایجاد این فراخوانی را به پایان می رسانیم.

## پیاده سازی فراخوانی های سیستمی

### (۱) پیاده سازی فراخوانی سیستمی انتقال فایل

جهت اضافه کردن این فراخوانی سیستمی ، باید فایل های مختلفی را تغییر داد که در ادامه به آن اشاره می کنیم

user.h

```
29 int move_file(char*,char*);
30
31
```

تابع را در دسترس کاربر قرار می دهیم .

defs.h

```

127 void list_all_processes(int);
128 int move_file(char*,char*);
129
130 // switch

```

Makefile(UPROGS)

```

167
168 UPROGS=\
169     _cat\
170     _decode\
171     _echo\
172     _encode\
173     _history\
174     _forktest\
175     _grep\
176     _init\
177     _kill\
178     _ln\
179     _ls\
180     _mkdir\
181     _move_file\
182     _rm\
183     _sh\
184     _stressfs\
185     _usertests\
186     _wc\
187     _zombie\
188     _create_palindrome\
189     _list_all_processes\
190

```

Makefile(EXTRA)

```

254 # Check in that version.
255
256 EXTRA=\
257     mkfs.c ulib.c user.h cat.c echo.c decode.o encode.o move_file.o history.o create_palindrome.o
258     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
259     printf.c umalloc.c\
260     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
261     .gdbinit.tmpl gdbutil\
262

```

move\_file.c

تست با برنامه ی سطح کاربر :



```

move_file.c > main(int, char *[])
1  #include "types.h"
2  #include "user.h"
3  #include "fcntl.h"
4  #include "stat.h"
5
6  int main(int argc , char *argv[])
7  {
8      if(argc < 3)
9      {
10         printf(1,"source and destination are needed\n");
11         exit();
12     }
13     char *src_file = argv[1];
14     char *dest_dir = argv[2] ;
15     if (move_file(src_file,dest_dir) < 0 )
16     {
17         printf(1,"move file failed\n");
18         exit();
19     }
20     printf(1,"file %s moved to %s successfully\n",src_file,dest_dir);
21     exit();
22 }

```

syscall.c

```

167 [SYS_move_file] sys_move_file ,
168 };

```

```

139 extern int sys_list_all_processes(void);
140 extern int sys_move_file(void);

```

```

10 char system_call_titles[30][30] = {
11     "fork",
12     "exit",
13     "wait",
14     "pipe",
15     "read",
16     "kill",
17     "exec",
18     "fstat",
19     "chdir",
20     "dup",
21     "getpid",
22     "sbrk",
23     "sleep",
24     "uptime",
25     "open",
26     "write",
27     "mknod",
28     "unlink",
29     "link",
30     "mkdir",
31     "close",
32     "create palindrome",
33     "sort system calls",
34     "most invoked system call",
35     "list all processes",
36     "move file",
37 };
38

```

sysfile.c

اضافه کردن کد قابلیت خواسته شده (انتقال فایل ) به این فایل

```

445 int sys_move_file(void) {
446     char *src_path, *dest_path;
447     char filename[DIRSIZ];
448     struct inode *src_inode = 0, *src_parent_inode = 0, *dest_inode = 0;
449     uint offset;
450     // int result = -1;
451
452     if (argstr(0, &src_path) < 0 || argstr(1, &dest_path) < 0) {
453         return -1;
454     }
455
456     begin_op();
457
458     // Find the inode of the source file
459     src_inode = namei(src_path);
460     if (src_inode == 0) {
461         end_op();
462         return -1;
463     }
464
465     ilock(src_inode);
466
467     if (src_inode->type != T_FILE) {
468         iunlockput(src_inode);
469         end_op();
470         return -1;
471     }
472
473     src_inode->nlink++;
474     iupdate(src_inode);
475     iunlock(src_inode);
476
477     // Find the parent inode of the source path
478     src_parent_inode = nameiparent(src_path, filename);
479     if (src_parent_inode == 0) {
480         ilock(src_inode);
481         src_inode->nlink--;
482         iupdate(src_inode);
483         iunlockput(src_inode);
484         end_op();
485         return -1;
486     }
487
488     ilock(src_parent_inode);
489
490     if (dirlookup(src_parent_inode, filename, &offset) == 0) {
491         iunlockput(src_parent_inode);
492         ilock(src_inode);
493         src_inode->nlink--;
494         iupdate(src_inode);
495         iunlockput(src_inode);
496         end_op();
497         return -1;
498     }
499
500     struct dirent empty_entry;
501     memset(&empty_entry, 0, sizeof(empty_entry));
502
503     if (writei(src_parent_inode, (char*)&empty_entry, offset, sizeof(empty_entry)) != sizeof(empty_entry)) {
504         iunlockput(src_parent_inode);

```

```

505     ilock(src_inode);
506     src_inode->nlink--;
507     iupdate(src_inode);
508     iunlockput(src_inode);
509     end_op();
510     return -1;
511 }
512
513 if (src_inode->type == T_DIR) {
514     src_parent_inode->nlink--;
515     iupdate(src_parent_inode);
516 }
517 src_parent_inode->nlink--;
518 iunlock(src_parent_inode);
519
520 // Find the destination directory inode
521 dest_inode = namei(dest_path);
522 if (dest_inode == 0) {
523     ilock(src_inode);
524     src_inode->nlink--;
525     iupdate(src_inode);
526     iunlockput(src_inode);
527     end_op();
528     return -1;
529 }
530
531 ilock(dest_inode);
532 if (dest_inode->type != T_DIR) {
533     iunlockput(dest_inode);
534     ilock(src_inode);
535     src_inode->nlink--;
536     iupdate(src_inode);
537     iunlockput(src_inode);
538     end_op();
539     return -1;
540 }
541
542 if (dirlookup(dest_inode, filename, &offset) != 0) {
543     iunlockput(dest_inode);
544     ilock(src_inode);
545     src_inode->nlink--;
546     iupdate(src_inode);
547     iunlockput(src_inode);
548     end_op();
549     return -1;
550 }
551
552 if (dirlink(dest_inode, filename, src_inode->inum) < 0) {
553     iunlockput(dest_inode);
554     ilock(src_inode);
555     src_inode->nlink--;
556     iupdate(src_inode);
557     iunlockput(src_inode);
558     end_op();
559     return -1;
560 }
561
562 iunlockput(dest_inode);
563
564 ilock(src_inode);
565 src_inode->nlink--;
566 iupdate(src_inode);
567 iunlockput(src_inode);
568
569 end_op();
570 return 0;
571

```


## 34 SYSCALL(move\_file)

تابع `move_file` به `SYS_move_file` مپ می شود

`SYS_move_file` عدد سیستم کال است که در `syscall.h` تعریف میشود.

`syscall.h`

```
27 | #define SYS_move_file 26
```

 Do you  
extensi

نتیجه

ایجاد دو دایرکتوری و ایجاد فایل در دایرکتوری `dest` و انتقال آن به دایرکتوری `test`

```
$ mkdir test
$ mkdir dest
$ echo hi >/dest/a.txt
$ move_file /dest/a.txt test
file /dest/a.txt moved to test successfully
$ ls test
.          1 25 48
..         1 1 512
a.txt     2 27 3
$ ls dest
.          1 26 48
..         1 1 512
$
```

(۳۰۲)

برای سیستم کال های `sort_syscalls` و `get_most_invoked` نیاز داریم که تمام سیستم کال های اجرا شده در یک پراسس را داشته باشیم. به این منظور در `struct proc` اطلاعات پراسس در حال اجرا را نگه میدارد دو فیلد اضافه کرده ایم. `Syscalls` آرایه ای از سیستم کال ها است (`syscall_info` حاوی اسم و شماره سیستم کال است) و `syscall_count` تعداد سیستم کال را نشان می دهد.

```
43 // Per-process state
44 struct proc {
45     uint sz; // Size of process memory (bytes)
46     pde_t* pgdir; // Page table
47     char *kstack; // Bottom of kernel stack for this process
48     enum procstate state; // Process state
49     int pid; // Process ID
50     struct proc *parent; // Parent process
51     struct trapframe *tf; // Trap frame for current syscall
52     struct context *context; // switch() here to run process
53     void *chan; // If non-zero, sleeping on chan
54     int killed; // If non-zero, have been killed
55     struct file *ofile[NOFILE]; // Open files
56     struct inode *cwd; // Current directory
57     char name[16]; // Process name (debugging)
58     struct syscall_info syscalls[1000]; // List of system calls used in process
59     int syscall_count; // Number of system calls used in process
60 };
```

در فایل کد syscall.c هم تغییراتی اعمال کردیم. بعد از فراخوانی هر سیستم کال دو فیلد گفته شده در استراکت آپدیت میشوند. تابع my\_strcpy برای کپی کردن نام سیستم کال ها استفاده میشود.

```
171
172 void my_strcpy(char *dest, const char *src) {
173     while (*src) {
174         *dest++ = *src++;
175     }
176     *dest = '\0';
177 }
178
179 void
180 syscall(void)
181 {
182     int num;
183     struct proc *curproc = myproc();
184
185     num = curproc->tf->eax;
186     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
187         curproc->tf->eax = syscalls[num]();
188         if (curproc->syscall_count < 1000) {
189             struct syscall_info new_syscall = {"", num};
190             new_syscall.number = num;
191             my_strcpy(new_syscall.name, system_call_titles[num]);
192             curproc->syscalls[curproc->syscall_count++] = new_syscall;
193         } else {
194             cprintf("max system calls limit for a process has exceeded\n");
195         }
196     } else {
197         cprintf("%d %s: unknown sys call %d\n",
198             curproc->pid, curproc->name, num);
199         curproc->tf->eax = -1;
200     }
201 }
```

برای دسترسی به نام سیستم کال ها با استفاده از شماره آن ها آرایه ای از استرینگ تعریف شده که در هدر syscall\_titles.h قرار دارد تا در فایل های مختلف استفاده شود.

```

10
11 char system_call_titles[30][30] = {
12     "not a valid syscall",
13     "fork",
14     "exit",
15     "wait",
16     "pipe",
17     "read",
18     "kill",
19     "exec",
20     "fstat",
21     "chdir",
22     "dup",
23     "getpid",
24     "sbrk",
25     "sleep",
26     "uptime",
27     "open",
28     "write",
29     "mknod",
30     "unlink",
31     "link",
32     "mkdir",
33     "close",
34     "create palindrome",
35     "sort system calls",
36     "get most invoked system call",
37     "list all processes",
38     "move file",
39 };
40

```

تعدادی تابع در فایل `proc.c` نیاز اضافه شده اند که به توضیح آن ها میپردازیم.

این تابع بر اساس آیدی داده شده به جستجوی پراسس در `page table` میگردد. اگر پیدا نکند صفر برمیگرداند و در غیر این صورت اشاره گر به استراکت آن پراسس.

```

560 struct proc*
561 find_process_by_id(int pid)
562 {
563     struct proc *p;
564     // Acquire the process table lock to ensure thread safety
565     acquire(&ptable.lock);
566
567     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
568         if (p->pid == pid) {
569             // If the process was found, release the lock and return the process pointer
570             release(&ptable.lock);
571             return p;
572         }
573     }
574
575     // if the process wasn't found, release the lock and return 0
576     release(&ptable.lock);
577     return 0;
578 }

```

این تابع سیستم کال های یک پراسس را بر اساس شماره شان مرتب میکند (صعودی). در ابتدا بر اساس آیدی پراسس را پیدا کرده و اگر وجود نداشت پیغام مناسبی را چاپ میکند. در غیر این صورت اطلاعات سیستم کال ها را در آرایه جدیدی میریزد. برای این که ترتیب اصلی سیستم کال ها را عوض نکنیم. سپس با استفاده از الگوریتم بابل سورت به مرتب سازی میپردازد در نهایت نیز پراسس های مرتب شده با نام و شماره شان چاپ می شوند.

```

583 int sort_syscalls(int pid)
584 {
585     // Get the process structure
586     struct proc* p = find_process_by_id(pid);
587
588     if (p == 0) {
589         cprintf("Kernel: Process with ID %d not found\n", pid);
590         return -1;
591     }
592
593
594     int count = p->syscall_count;
595     if (count == 0) {
596         cprintf("No system calls yet.\n");
597         return 0;
598     }
599
600     // don't change the original system calls order
601     struct syscall_info copied_syscalls[count];
602     int cnt = 0;
603     while (cnt < count) {
604         copied_syscalls[cnt] = p->syscalls[cnt];
605         cnt++;
606     }
607
608     // use a simple bubble sort algorithm
609     for (int i = 0; i < count-1; i++) {
610         for (int j = 0; j < count-i-1; j++) {
611             if (copied_syscalls[j].number > copied_syscalls[j+1].number) {
612                 struct syscall_info temp = copied_syscalls[j];
613                 copied_syscalls[j] = copied_syscalls[j+1];
614                 copied_syscalls[j+1] = temp;
615             }
616         }
617     }
618
619     // print the sorted system calls
620     cprintf("Sorting system calls of process %s:\n", p->name);
621     for (int i=0; i<count; i++) {
622         cprintf("    %d. %s (%d)\n", i+1, copied_syscalls[i].name, copied_syscalls[i].number);
623     }
624
625     return 0;
626 }

```

این تابع برای پیدا کردن سیستم کال با بیشترین تعداد تکرار است. ابتدا به مانند تابع قبل پراسس را پیدا میکند. سپس آرایه ای برای شمردن سیستم کال ها ساخته میشود و با پیشمایش روی سیستم کال ها آرایه آپدیت میشود. همچنین متغیر `max_freq` نیز. در ادامه دو حالت داریم. تنها یک سیستم کال بیشترین تعداد تکرار را داشته باشد (اکثر مواقع) و اینکه چند سیستم کال این خاصیت را داشته باشند. هر دو حالت هندل شده است. در انتها نیز موارد خواسته شده چاپ میشوند (اسم و تعداد تکرار)



```

630 int
631 get_most_invoked_syscall(int pid)
632 {
633     // Get the process structure
634     struct proc* p = find_process_by_id(pid);
635
636     if (p == 0)
637     {
638         cprintf("Process with ID %d not found\n", pid);
639         return -1;
640     }
641
642     int count = p->syscall_count;
643     if (count == 0)
644     {
645         cprintf("No system calls in the %s process yet.\n", p->name);
646         return -1;
647     }
648
649     int counts[30] = {0};
650
651     int max_freq = -1;
652     int cnt = 0;
653     while (cnt < count)
654     {
655         counts[p->syscalls[cnt].number]++;
656         if (counts[p->syscalls[cnt].number] > max_freq)
657         {
658             max_freq = counts[p->syscalls[cnt].number];
659         }
660         cnt++;
661     }
662
663     if (max_freq == -1)
664     {
665         return -1;
666     }
667
668     int max_count = 0;
669     cnt = 0;
670     for (int i=0; i<30; i++)
671     {
672         if (counts[i] == max_freq)
673         {
674             max_count++;
675         }
676     }

```

```

678     if (max_count == 0)
679     {
680         cprintf("No most invoked system call found for process with ID %s\n", p->name);
681         return -1;
682     }
683     else if (max_count == 1)
684     {
685         for (int i=0; i<30; i++)
686         {
687             if (counts[i] == max_freq)
688             {
689                 cprintf("The most invoked process is %s with %d times\n", system_call_titles[i], counts[i]);
690             }
691         }
692         return 0;
693     }
694     else
695     {
696         cprintf("There are several system calls with the most count:\n");
697         int idx = 1;
698         for (int i=0; i<30; i++)
699         {
700             if (counts[i] == max_freq)
701             {
702                 cprintf("    %d. %s with %d times.\n", system_call_titles[i], counts[i]);
703                 idx++;
704             }
705         }
706         return 0;
707     }
708 }
709 }

```

دو تابع به sysproc.c نیز اضافه شدند که برای اجرای دو سیستم کال هستند. بعد از ارور هندلینگ تابع هرکدام کال شده است.

```

102 sys_sort_syscalls(void)
103 {
104     int pid;
105     // try to extract the first argument of the system call
106     int could_fetch = argint(0, &pid);
107     if (could_fetch < 0) {
108         cprintf("Kernel: Could not extract the 'pid' argument for sort_syscalls\n");
109         return -1;
110     }
111
112     // If the function reaches here, it means the 'pid' argument is available
113     cprintf("Kernel: sort_syscalls called for process with ID %d\n", pid);
114
115     return sort_syscalls(pid);
116 }
117
118 int
119 sys_get_most_invoked_syscall(void)
120 {
121     int pid;
122     // try to extract the first argument of the system call
123     int could_fetch = argint(0, &pid);
124     if (could_fetch < 0)
125     {
126         cprintf("Kernel: Could not extract the 'pid' argument for get_most_invoked_systemcall\n");
127         return -1;
128     }
129
130     // If the function reaches here, it means the 'pid' argument is available
131     cprintf("Kernel: most_invoked_systemcall called for process with ID %d\n", pid);
132
133     int res = get_most_invoked_syscall(pid);
134     if (res < 0)
135     {
136         cprintf("Kernel: Could not find the most invoked system call for process with ID %d\n", pid);
137         return -1;
138     }
139     else
140     {
141         cprintf("Kernel: Successfully found the most invoked system call for process with ID %d\n", pid);
142         return 0;
143     }
144 }

```

today at 2:57 PM

حالا که پیاده سازی کامل شده است میتوانیم تست کنیم. دو فایل `sort_syscalls.c` و `get_most_invoked_syscall.c` نوشته شده اند.

```
C sort_syscalls.c > ...
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char *argv[])
6  {
7      if (argc != 2) {
8          printf(2, "Usage: sort_syscalls <pid>\n");
9          exit();
10     }
11     int pid = atoi(argv[1]);
12     int result = sort_syscalls(pid);
13     if (result < 0) {
14         printf(1, "Failed to sort syscalls for PID %d\n", pid);
15     }
16     else {
17         printf(1, "Successfully sorted syscalls for PID %d\n", pid);
18     }
19     exit();
20 }
21
```

```
C get_most_invoked_syscall.c > ...
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char *argv[])
6  {
7      if (argc != 2) {
8          printf(2, "Usage: get_most_invoked_syscall <pid>\n");
9          exit();
10     }
11     int pid = atoi(argv[1]);
12     int result = get_most_invoked_syscall(pid);
13
14     if (result < 0) {
15         // printf(1, "Failed to find the most invoked syscall for PID %d\n", pid);
16     } else {
17         // printf(1, "Successfully found the most invoked syscall for PID %d\n", pid);
18     }
19
20     exit();
21 }
```

#### ۴) پیاده سازی فراخوانی سیستمی لیست کردن پردازش ها

در این بخش ما به کمک برخی ویژگی هایی که در بخش های قبل اضافه شده و برخی ویژگی هایی که خودمان اضافه می کنیم، این قابلیت را به سیستم می دهیم.

▼ syscall.c

...	@@ -32,7 +32,8 @@	char system_call_titles[30][30] = {
32	32	"close",
33	33	"create palindrome",
34	34	"sort system calls",
35	-	"most invoked system call"
	35 +	"most invoked system call",
	36 +	"list all processes"
36	37	};
37	38	
38	39	// User code makes a system call with INT T_SYSCALL.

↓

▼ syscall.h

↑	...	@@ -22,4 +22,5 @@
22	22	#define SYS_close 21
23	23	#define SYS_create_palindrome 22
24	24	#define SYS_sort_syscalls 23
25	-	#define SYS_most_invoked_syscall 24
	25 +	#define SYS_most_invoked_syscall 24
	26 +	#define SYS_list_all_processes 25

syscall.c		
↑		@@ -135,6 +135,8 @@ extern int sys_uptime(void);
135	135	extern int sys_create_palindrome(void);
136	136	extern int sys_sort_syscalls(void);
137	137	extern int sys_most_invoked_syscall(void);
	138	+ extern int sys_list_all_processes(void);
	139	+
138	140	
139	141	static int (*syscalls[])(void) = {
140	142	[SYS_fork] sys_fork,
↕		@@ -160,7 +162,9 @@ static int (*syscalls[])(void) = {
160	162	[SYS_close] sys_close,
161	163	[SYS_create_palindrome] sys_create_palindrome,
162	164	[SYS_sort_syscalls] sys_sort_syscalls,
163		- [SYS_most_invoked_syscall] sys_most_invoked_syscall
	165	+ [SYS_most_invoked_syscall] sys_most_invoked_syscall,
	166	+ [SYS_list_all_processes] sys_list_all_processes,
	167	+
164	168	};
165	169	
166	170	void my_strcpy(char *dest, const char *src) {

مجدد مانند موارد قبلی یک شماره به سیستم کال جدید خود نسبت می دهیم و آن را تعریف می کنیم. و همچنین در فایل هایی که نیاز به تعریف این سیستم کال جدید است آن را تعریف می کنیم.

▼ sysproc.c

↑	@@ -148,4 +148,11 @@ sys_most_invoked_syscall(void)
148	148
149	149       get_most_invoked_syscall(p);
150	150       return 0;
151	+ }
152	+
153	+ int
154	+ sys_list_all_processes(void)
155	+ {
156	+     cprintf("Kernel: sys_list_all_processes called.\n");
157	+     return list_all_processes();
151	158     }

تابع مربوط به بخش هسته را در فایل بالا اضافه می کنیم. این تابع هم مثل موارد قبلی یک تابعی که قرار است محاسبات ما را انجام دهد صدا می زند.



▼ list\_all\_processes.c

```
... @@ -0,0 +1,17 @@
1 + #include "types.h"
2 + #include "stat.h"
3 + #include "user.h"
4 +
5 + int main(int argc, char *argv[]){
6 +     if(argc != 1){
7 +         printf(2, "You must enter exactly 1 number!\n");
8 +         exit();
9 +     }
10 +    else
11 +    {
12 +        printf(1, "User: list_all_processes() called.\n");
13 +        list_all_processes();
14 +        exit();
15 +    }
16 +    exit();
17 + }
```

یک فایل جدید درست می کنیم که این فایل حکم سطح کاربر را دارد. در این بخش ابتدا مشخص می کنیم که کاربر درست ورودی وارد کرده باشد و سپس تابع مربوط به لیست کردن همه پراسس ها را اجرا می کنیم.



```
proc.c @@ -645,4 +645,18 @@ void get_most_invoked_syscall(struct proc *p)
645 645
646 646     printf("The most invoked process is %s with %d times\n", p->syscalls[max_idx].name, max_freq);
647 647 }
648 - // -----
648 + // -----
649 +
650 + void
651 + list_all_processes(int a)
652 + {
653 +     struct proc *p;
654 +
655 +     acquire(&ptable.lock);
656 +     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
657 +     {
658 +         if(p->pid != 0)
659 +             printf("My name is: %s.\t My id is:%d.\t Number of system calls i invoked is: %d.\n",p->name, p->pid,p->syscall_count);
660 +     }
661 +     release(&ptable.lock);
662 + }
```

در فایل `proc.c` تعریف این تابع را قرار می دهیم. برای لیست کردن تمام پراسس های فعلی ما از پیچ تیبیل که پراسس های هر لحظه را دارد استفاده می کنیم. به این صورت که به کمک پارامتری که در بخش های قبل تعریف شده بود و تعداد سیستم کال های یک پراسس را محاسبه می کرد می توانیم به جواب برسیم.

روند کار اینطور است که روی پیچ تیبیل پیمایش انجام می دهیم و به ازای هر پراسسی که وجود داشت تعداد سیستم کال هایی که انجام داده است را به کاربر نشان می دهیم.

```
Makefile @@ -185,6 +185,7 @@ UPROGS=\
185 185     _wc\
186 186     _zombie\
187 187     _create_palindrome\
188 +     _list_all_processes\
188 189
189 190 fs.img: mkfs README $(UPROGS)
190 191     ./mkfs fs.img README $(UPROGS)
192
193 @@ -252,7 +253,7 @@ qemu-nox-gdb: fs.img xv6.img .gdbinit
252 253 # check in that version.
253 254
254 255 EXTRA=\
255 -     mkfs.c ulib.c user.h cat.c echo.c decode.o encode.o history.o create_palindrome.o forktest.c grep.c kill.c\
256 +     mkfs.c ulib.c user.h cat.c echo.c decode.o encode.o history.o create_palindrome.o list_all_processes.o forktest.c grep.c kill.c\
256 257     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
257 258     printf.c umalloc.c\
258 259     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\

```

مجددا تغییرات میک فایل را انجام می دهیم و سیستم کال جدید را به آن اضافه می کنیم.

proc.c		
↑...	@@ -88,6 +88,7 @@ allocproc(void)	
88	88	found:
89	89	p->state = EMBRYO;
90	90	p->pid = nextpid++;
91	+	p->syscall_count = 0;
91	92	
92	93	release(&ptable.lock);
93	94	
↓...		

بازگرداندن تعداد سیستم کال پراسس بخشی حیاتی است به این دلیل که در صورت اشتباه در این بخش، نتیجه های بدست آمده اشتباه خواهد شد. پس مهم است که هنگامی که یک پراسس قرار است به چیزی تعلق گیرد تعداد سیستم کال هایش به صفر برگردانده شود.