به نام خدا

گزارش آزمایشگاه آزمایش دوم درس سیستم عامل

آیدین کاظمی: ۸۱۰۱۰۱۵۶۱ علی زیلوچی: ۸۱۰۱۰۱۵۶۰ بابک حسینی محتشم: ۸۱۰۱۰۱۴۰۸

مقدمه:

پرسش 1: کتابخانه‌های سطح کاربر در xv6، برای ایجاد ارتباط میان برنامه‌های کاربر و کرنل به کار می‌روند. این کتابخانه‌ها شامل توابعی هستند که از فراخوانی‌های سیستمی استفاده می‌کنند تا دسترسی به منابع سخت‌افزاری و نرم‌افزاری سیستم‌عامل ممکن شود. با تحلیل فایل‌های موجود در متغیر ULIB در xv6، توضیح دهید که چگونه این کتابخانه‌ها از فراخوانی‌های سیستمی بهره می‌برند؟ همچنین، دلایل استفاده از این فراخوانی‌ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه‌ها را شرح دهید.

پاسخ سوال ۱:

در ابتدا متغیر نام برده را داخل Makefile جانمایی میکنیم:

```
145
146 ULIB = ulib.o usys.o printf.o umalloc.o
147
```

که همانطور که مشاهده میشود از لینک شدن چهار آبجکت فایل تشکیل شده، که سورس هر کدام را بررسی میکنیم:

ulib.c

این کد شامل توابعی برای عملیات‌های پایه‌ای رشته و حافظه در کتابخانه سطح کاربر است که بیشتر آن‌ها به سیستم کال نیاز ندارند. توضیحات هر تابع:

- **strcpy**، **strcmp**، **strlen**: برای کپی، مقایسه، و اندازه‌گیری طول رشته‌ها استفاده می‌شوند.
- **memset**: آرایه‌ای از حافظه را با یک مقدار مشخص پر می‌کند.
- **strchr**: اولین حضور یک کاراکتر خاص در رشته را پیدا می‌کند.
- **gets**: با استفاده از سیستم کال read، ورودی را از کاربر می‌خواند و در بافر ذخیره می‌کند.

- **stat**: اطلاعات فایل را با سیستم کال‌های open، fstat و close به دست می‌آورد.
- **atoi** رشته‌ای از اعداد را به یک عدد صحیح تبدیل می‌کند.
- **memmove**: داده‌ها را از مبدا به مقصد کپی می‌کند.

usys.S

در ابتدای این فایل اسمبلی ماکرو SYSCALL وجود دارد که یک تابع تعریف میکند که با استفاده از نام هر سیستم کال، ابتدا شماره مربوط به آن را در رجیستر EAX ذخیره کرده (که این شماره ها در syscall.h ذخیره شده اند) و سپس int 64 را صدا میکند تا یک اینتراپت رخ دهد و trap اتفاق افتاده، با تغییر مود از user به kernel و درون کرنل مقدار eax خوانده میشود تا بفهمد کدام سیستم کال رخ داده و عملیات مورد نظر را انجام دهد. نهایتا با دستور ret به user مود برگشته و مقدار برگشتی داخل eax ذخیره شده است:

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

printf.c

این فایل شامل تعریف تابع printf میباشد، که عملیات چاپ کردن را با استفاده از دو تابع کمکی putc (نوشتن یک کاراکتر در فایل دیسکریپتور مقصد با سیستم کال write) و printint (فرمت دهی و نوشتن اعداد صحیح یا هگز با کمک تابع putc) انجام میدهد، به صورتی که یک رشته را دریافت کرده و مقدار متناسب را با این دو تابع در فایل دیسکریپتور مقصد مینویسد.

umalloc.c

این فایل شامل تعریف سه تابع morecore، malloc و free بر اساس کتاب کرنیگان و ریچی میباشد که تابع free وظیفه آزاد کردن حافظه تخصیص یافته و تابع malloc وظیفه تخصیص حافظه با استفاده از تابع morecore را دارد. تنها درون تابع morecore از سیستم کال استفاده شده (sbrk که فضای حافظه پراسس را بیشتر میکند).

دلایل استفاده از این فراخوانی ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه ها را شرح دهید:

از آنجایی که انجام بعضی از اعمال توسط کاربر ممکن است امنیت یا کارایی سیستم را به خطر بیندازد، و همچنین بعضی از دسترسی ها صرفا توسط سیستم عامل انجام میشود، وظیفه انجام برخی از کار ها به سیستم عامل سپرده شده و اینترفیس استفاده از آنها توسط سیستم کال ها پیاده سازی شده اند تا کاربر به شکل امنی بتواند از این خدمات سیستم استفاده کند. از آنجایی که بخش بزرگی از اینترفیس سیستم کال ها به طور کلی میان سیستم عامل های مختلف به صورت یکسان پیاده سازی شده اند، بخش هایی از برنامه که این سیستم کال ها را شامل میشوند قابل حمل و بقیه بخش های غیر مشترک غیر قابل حمل میباشند. همچنین استفاده از سیستم کال ها به دلیل تغییر مود از کاربر به کرنل و برخی اعمال مرتبط شامل سرباری جزئی میباشد.

پرسش 2: 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.

پاسخ سوال ۲:

برخی روش های دیگر به اختصار توضیح داده شده اند:

۱) Interrupt:

وقفه یا اینتراپت دارای دو نوع است: نرم افزاری (که به آن trap هم گفته میشود) و سخت افزاری. وقفه های سخت افزاری (که عموما توسط عملیات های ورودی خروجی ایجاد میشوند) عموما به صورت Asynchronous (بدون وابستگی به روند اجرای فعلی) انجام شده و توسط یک سخت افزار ایجاد میشوند (به طور مثال فشردن کیبورد، حرکت دادن موس یا اتمام آی او). وقفه های نرم افزاری (که سیستم کال ها نیز جزئی از آنها هستند) توسط یک نرم افزار و به صورت Synchronous تولید شده و به طور کلی سه

نوع سیستم کال، اکسپشن (که در هنگام بروز خطا مانند تقسیم بر ۰ رخ میدهند) و سیگنال (که برای آگاه سازی پراسس ها از اتفاقاتی خاص مانند SIGINT در لینوکس (اتمام پراسس با CTRL + C) یا SIGKILL برای اتمام فوری پراسس استفاده میشوند) را شامل میشوند.

۲) Pseudo-Filesystems

شامل فایل سیستم هایی مانند /proc و /sys میباشد که اطلاعاتی را در مورد تنظیمات و فرایند های کرنل را در اختیار کاربر قرار داده و کاربر میتواند با نوشتن یا خواندن از این فایل ها، اطلاعاتی در مورد سیستم به دست آورده یا تنظیمات را تغییر دهند.

۳) Netlink Sockets

یک وسیله ارتباط میان کاربر و کرنل مبتنی بر سوکت است که برای انتقال داده ها بین کرنل و برنامه کاربر استفاده میشود. معمولا برای شبکه بندی و مدیریت سیستم استفاده میشود.

سازوکار اجرای فراخوانی سیستمی در xv6

پرسش 3: آیا باقی تله‌ها را نمی‌توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟

پاسخ سوال ۳:

خیر. در سیستم عامل xv6، در صورتی که کاربر بخواهد تله دیگری را فعال کند خطای protection exception دریافت خواهد کرد، چرا که تله های سطح کرنل دارای سطح دسترسی پایین تری از سطح دسترسی کاربر هستند. از فواید این قاعده نیز میتوان به جلوگیری از دسترسی غیر مجاز به کرنل و بروز خطاهای احتمالی غیر عمدی (وجود باگ داخل برنامه) یا عمدی (سوء استفاده کاربر) اشاره کرد، که در صورت نبود این قاعده ممکن بود امنیت یا کارایی سیستم را به خطر بیندازند.

پرسش 4: در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟

پاسخ سوال ۴:

پراسس ها در سیستم عامل xv6 دارای دو نوع استک میباشند: استک کاربر و استک کرنل. هنگام تغییر مود از کاربر به هسته، مقادیر رجیستر های ss و esp که تا به حال به محتویات استک کاربر اشاره داشتند، دستخوش تغییر شده و به استک کرنل اشاره میکنند. حال از آنجایی که پس از اتمام کار کرنل باید به جای قبلی برنامه کاربر برگشته و اجرا را از سر بگیریم، برای جلوگیری از گم شدن محتویات فعلی esp و ss (و در نتیجه از دست رفتن مکان اجرای قبلی و عدم توانایی از سر گیری برنامه) این دو مقدار باید بر روی استک push شوند. حال در صورتی که تغییر مود نداشته باشیم، مقادیر esp و ss همچنان برای استک کاربر معتبر بوده و در نتیجه نیازی به ذخیره سازی دوباره آنها وجود ندارد.

پرسش 5: 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 ()read_sys اجرای سیستم را با مشکل روبرو سازد.

پاسخ سوال ۵:

توابع دسترسی به پارامتر های فراخوانی سیستمی عبارتند از argint،argptr و argstr که وظیفه هر کدام دریافت شماره آرگومان و مقصد برگرداندن آن، و گشتن داخل استک بر اساس آن شماره آرگومان، برای پیدا و قرار دادن آرگومان مد نظر در مقصد میباشد. هر کدام را به اختصار توضیح میدهیم:

۱) argint

این تابع وظیفه بازگرداندن یک مقدار صحیح از استک را دارد. در این تابع ابتدا آدرس دسترسی به آرگومان n ام محاسبه میشود، که از آنجایی که در سر استک آدرس بازگشت از تابع ذخیره میشود و پس از آن آرگومان ها به ترتیب از اولی تا آخری قرار دارند، باید به اندازه n + 1 خانه داخل استک جلو برود. حال از آنجایی که مقدار استک از حافظه بیشتر به کمتر پر میشود، یعنی باید داخل حافظه مربوط به استک به اندازه (n+1) * 4 بایت از نقطه سر استک جلو برویم، که این نقطه در رجیستر esp ذخیره شده است. بنابراین آدرس آرگومان مد نظر برابر با esp + 4 * (n+1) خواهد شد. در مرحله بعد این آدرس محاسبه

شده به تابع fetchint داده شده، و به عنوان آرگومان دیگر آن پوینتر ip داده میشود که به عنوان ورودی توسط این تابع دریافت شده. نهایتا fetchint در صورتی که آدرس حافظه نشان شده بالاتر یا مساوی رجیستر sz نباشد، آن را به عنوان پوینتر داخل ip قرار میدهد و موفقیت آمیز باز میگردد.

۲) argptr

این تابع وظیفه بازگرداندن یک پوینتر سایز مشخص را دارد، که آدرس شروع آن در خانه n ام استک ذخیره شده است. در این تابع ابتدا با کمک argint آدرس شروع پوینتر مد نظر از داخل استک بازیابی میشود (که سناریو های شکست آن نیز در بخش قبل بررسی شد). سپس پارامتر های ورودی به تابع بررسی میشوند (مانند منفی نبودن سایز و داخل محدوده حافظه بودن پوینتر مشخص شده)، و در صورتی که مشکلی وجود نداشته باشد، مقدار باز گردانده شده توسط argint به عنوان پوینتر مد نظر قرار میگیرد.

۳) argstr

این تابع وظیفه بازیابی یک رشته را دارد. مانند تابع قبل، ابتدا با کمک argint آدرس شروع رشته را از داخل استک بازیابی میکند. سپس آدرس بازیابی شده را به تابع fetchstr پاس میدهد. این تابع نیز ابتدا بررسی میکند که آدرس داده شده در بازه حافظه پراسس باشد. در مرحله بعد مقدار پوینتر بیرونی را برابر آدرس شروع قرار داده و سپس از آدرس شروع یکی یکی جلو رفته و در صورت رسیدن به کاراکتر null، آدرس آن خانه را از خانه شروع کم کرده و به عنوان طول رشته باز میگرداند. در غیر این صورت و در صورت رسیدن به انتهای محدوده مشخص شده حافظه پراسس، مقدار -۱ باز میگرداند.

تمامی این توابع داخل محدوده بودن آدرس فراخوانی شده را بررسی میکنند، که در غیر این صورت بدیهتا مشکلاتی از قبیل نوشتن در حافظه پراسس دیگر و ایجاد مشکلات امنیتی (مانند تغییر سطح دسترسی) و نهایتا ایجاد خطا و خرابی کرنل به وجود می آیند.

سیستم کال sys_read

در این سیستم کال که از یک فایل دیسکریپتور، به مقدار مشخص خوانده و در بافر مشخص ذخیره میکند، هر سه عنصر صحت فایل دیسکریپتور (به عنوان آرگومان اول)، داخل محدوده حافظه بودن بافر (آرگومان سوم) و همچنین صحت طول مشخص شده (به عنوان آرگومان دوم) را بررسی میکند:

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

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

در صورتی که مشکلی وجود نداشته باشد، به اندازه مشخص شده از فایل دیسکریپتور خوانده و داخل بافر قرار میدهد (با کمک fileread، که در صورتی که پیش از رسیدن به سایز تعیین شده به EOF برسد جلوتر نخواهد رفت). حال در صورتی که این بررسی ها انجام نمیشد، ممکن بود طول خیلی بزرگی داده شود (که منجر به خارج شدن از فضای حافظه پراسس میشد) یا بافری خارج از محدوده حافظه پراسس داده شود که هر دو باعث نوشته شدن در حافظه پراسس دیگر و ایجاد مشکل میشدند.

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb:

برنامه سطح کاربری به نام test تشکیل می‌دهیم که در آن در صورت وارد کردن 0 از تابع getpid استفاده می‌کنیم. همچنین از این برنامه بعدا برای تست سیستم‌کال‌هایی که اضافه می‌کنیم استفاده می‌کنیم.

```
5   int
6   main(int argc, char *argv[]) {
7     int pid = getpid();
8     if (argc<2)
9     {
10      printf(2, "usage: test system_call...\n");
11      exit();
12    }
13    if (!strcmp(argv[1],"0"))
14      printf(1,"Process ID: %d\n", pid);
```

ابتدا در تابع syscall یک breakpoint می‌دهیم و پس از اجرای برنامه سطح کاربر به breakpoint می‌رسیم. سپس از دستور bt استفاده می‌کنیم.

```
(gdb) break syscall
Breakpoint 2 at 0x80106510: file syscall.c, line 135.
(gdb) continue
Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 2, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:135
#1  0x8010755d in trap (tf=0x8dffefb4) at trap.c:43
#2  0x801072ff in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

همان‌طور که می‌توان دید، این دستور، call stack برنامه را نشان می‌دهد. یعنی فراخوانی‌هایی که انجام شده تا به تابع فعلی برسیم. پس می‌توان دید که ابتدا به تابع alltraps می‌رویم. در این تابع مقادیر رجیسترها در استک push می‌شود سپس تابع trap صدا زده می‌شود. همچنین می‌توان تابع trapret را دید که احتمالا پس از اتمام trap صدا زده می‌شود و مقادیر رجیسترها را از استک pop می‌کند.

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

می‌توان دید که تابع trap در ابتدا بررسی می‌کند که trapnumber ذخیره شده در استک چند است و اگر برابر 64 بود که در xv6 برای فراخوانی‌های سیستمی است، تابع syscall را فراخوانی می‌کند.

```
36  void
37  trap(struct trapframe *tf)
38  {
39    if(tf->trapno == T_SYSCALL){
40      if(myproc()->killed)
41        exit();
42      myproc()->tf = tf;
43      syscall();
44      if(myproc()->killed)
45        exit();
46      return;
47    }
48
```

با دو دستور up و down می‌توان بین توابعی که فراخوانی کردیم بالا و پایین برویم و چون الان در آخرین تابع فراخوانی شده هستیم دستور down کار خاصی انجام نمی‌دهد.

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

با استفاده از دستور c محتویات رجیستر eax که شماره سیستم‌کال در آن ذخیره شده را چندین بار میخوانیم. در ابتدا چند بار از سیستم‌کال read استفاده می‌شود که احتمالا هدف خواندن از ترمینال است. سپس یک بار سیستم‌کال fork اجرا می‌شود چون باید پردازه جدیدی ایجاد شود تا برنامه test را اجرا کند. سپس پردازه والد با سیستم‌کال wait منتظر اتمام کار پردازه جدید می‌شود. سپس از سیستم‌کال sbrk استفاده می‌شود. این سیستم‌کال مقدار فضای پردازه را تغییر می‌دهد پس احتمالا از آن برای تغییر فضای حافظه پردازه جدید استفاده می‌شود. سپس با سیستم‌کال exec کد مربوط به test در این پردازه کپی و اجرا می‌شود. در نهایت هم از سیستم‌کال getpid استفاده می‌شود.

```
(gdb) print curproc->tf->eax
$8 = 3
(gdb) c
Continuing.

Thread 1 hit Breakpoint 2, syscall () at syscall.c:146
146         _log_syscall(num);
(gdb) print curproc->tf->eax
$9 = 12
(gdb) c
Continuing.

Thread 1 hit Breakpoint 2, syscall () at syscall.c:146
146         _log_syscall(num);
(gdb) print curproc->tf->eax
$10 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 2, syscall () at syscall.c:146
146         _log_syscall(num);
(gdb) print curproc->tf->eax
$11 = 11
(gdb)
```

ارسال آرگومان های فراخوانی های سیستمی:

با بررسی فایل usys.s متوجه شدیم که در این فایل، ماکروی SYSCALL تابع پوشاننده سیستم‌کال‌های xv6 است. این ماکرو ابتدا شماره سیستم‌کال را در رجیستر eax می‌ریزد و سپس دستور interrupt می‌دهد. ما در این بخش برای ارسال آرگومان از طریق ثبات، ماکروی جدیدی به نام SYSCALL_USING_REGISTERS تعریف کردیم. چون پس از فراخوانی توابع در اسمبلی، آرگومان در استک ذخیره می‌شود، و می‌دانیم محتوای خانه اول استک آدرس بازگشت است پس در خانه دوم استک عددی که می‌خواهیم پالیندرومش را حساب کنیم قرار دارد پس محتوای خانه دوم استک را که در آدرس esp+4 قرار گرفته را در رجیستر ecx می‌ریزیم و سپس مشابه ماکروی SYSCALL عمل می‌کنیم. دلیل اینکه در رجیستر ecx می‌ریزم هم این است که این ثبات جزو ثبات‌های caller-saved registers است پس لزومی ندارد که مقدار آن پس از فراخوانی توابع تغییر کند و به همین دلیل می‌توان از آن به عنوان ثبات موقتی استفاده کرد.

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
#define SYSCALL_USING_REGISTERS(name) \
  .globl name; \
  name: \
    movl 4(%esp), %ecx; \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL_USING_REGISTERS(create_palindrome)
```

در فایل syscall.c لیست توابع سیستم‌کال‌ها قرار گرفته پس sys_create_palindrome هم به این تابع اضافه می‌کنیم.

```
111  static int (*syscalls[])(void) = {
112  [SYS_fork]    sys_fork,
113  [SYS_exit]    sys_exit,
114  [SYS_wait]    sys_wait,
115  [SYS_pipe]    sys_pipe,
116  [SYS_read]    sys_read,
117  [SYS_kill]    sys_kill,
118  [SYS_exec]    sys_exec,
119  [SYS_fstat]   sys_fstat,
120  [SYS_chdir]   sys_chdir,
121  [SYS_dup]     sys_dup,
122  [SYS_getpid]  sys_getpid,
123  [SYS_sbrk]    sys_sbrk,
124  [SYS_sleep]   sys_sleep,
125  [SYS_uptime]  sys_uptime,
126  [SYS_open]    sys_open,
127  [SYS_write]   sys_write,
128  [SYS_mknod]   sys_mknod,
129  [SYS_unlink]  sys_unlink,
130  [SYS_link]    sys_link,
131  [SYS_mkdir]   sys_mkdir,
132  [SYS_close]   sys_close,
133  [SYS_create_palindrome]   sys_create_palindrome,
```

```
85   extern int sys_chdir(void);
86   extern int sys_close(void);
87   extern int sys_dup(void);
88   extern int sys_exec(void);
89   extern int sys_exit(void);
90   extern int sys_fork(void);
91   extern int sys_fstat(void);
92   extern int sys_getpid(void);
93   extern int sys_kill(void);
94   extern int sys_link(void);
95   extern int sys_mkdir(void);
96   extern int sys_mknod(void);
97   extern int sys_open(void);
98   extern int sys_pipe(void);
99   extern int sys_read(void);
100  extern int sys_sbrk(void);
101  extern int sys_sleep(void);
102  extern int sys_unlink(void);
103  extern int sys_wait(void);
104  extern int sys_write(void);
105  extern int sys_uptime(void);
106  extern int sys_create_palindrome(void);
```

همچنین شماره سیستم‌کال جدید را به فایل syscall.h اضافه می‌کنیم.

```
1   // System call numbers
2   #define SYS_fork    1
3   #define SYS_exit    2
4   #define SYS_wait    3
5   #define SYS_pipe    4
6   #define SYS_read    5
7   #define SYS_kill    6
8   #define SYS_exec    7
9   #define SYS_fstat   8
10  #define SYS_chdir   9
11  #define SYS_dup    10
12  #define SYS_getpid 11
13  #define SYS_sbrk   12
14  #define SYS_sleep  13
15  #define SYS_uptime 14
16  #define SYS_open   15
17  #define SYS_write  16
18  #define SYS_mknod  17
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_create_palindrome  22
```

در فایل user.h هم این تابع این سیستم‌کال را declare می‌کنیم تا بتوان از آن در برنامه‌های سطح کار هم استفاده کرد.

```
27   // added system calls
28   void create_palindrome(int);
```

در فایل sysproc.c هم تابع sys_create_palindrome را اضافه کردیم که وظیفه آن مشابه وظیفه اکثر توابع این فایل این است که آگومان را بگیرد و آن را به تابع دیگری که create_palindrome در فایل proc.c است بدهد. البته چون آرگومان را در رجیستر ecx ریختیم، آرگومان را از این رجیستر پردازه برمی‌داریم.

```
93   // Takes an integer and prints its palindrome
94   int
95   sys_create_palindrome(void)
96   {
97     struct proc *curproc = myproc();
98     int num = curproc->tf->ecx;
99     create_palindrome(num);
100    return 0;
101  }
```

در نهایت منطق را در تابع create_palindrome در فایل proc.c اضافه می‌کنیم که ابتدا پالیندروم عدد آرگومان ورودی را تشکیل داده و سپس چاپ می‌کنیم.

```
544  void
545  create_palindrome(int num) //Babak
546  {
547    int new_num=num;
548    int palnum=num;
549    while (new_num)
550    {
551      palnum=palnum*10+new_num%10;
552      new_num=new_num/10;
553    }
554    cprintf("Palindrome of %d is: %d\n",num,palnum);
555    return;
556  }
```

حال برای تست، برنامه سطح کاربر را تغییر داده و سیستم‌کال را تست می‌کنیم.

بخش اضافه شده به برنامه سطح کاربر:

```
15    else if(!strcmp(argv[1],"1"))
16    {
17      if (argc<3)
18      {
19        printf(2, "usage: test 1 number...\n");
20        exit();
21      }
22      int num=atoi(argv[2]);
23      create_palindrome(num);
24    }
```

تست سیستم‌کال:

```
$ test 1 738
Palindrome of 738 is: 738837
$
```

پیاده‌سازی فراخوانی‌های سیستمی:

1. پیاده سازی فراخوانی سیستمی انتقال فایل:

برای پیاده‌سازی بقیه سیستم‌کال‌ها هم با کمی تغییرات مشابه قبل عمل می‌کنیم. در سیستم‌کال‌های بعدی چون از استک برای ارسال آرگومان‌ها استفاده می‌کنیم پس از همان ماکروی SYSCALL استفاده می‌کنیم.

اضافه کردن به usys.s:

```
40    SYSCALL(move_file)
```

اضافه کردن به syscall.h:

```
24    #define SYS_move_file  23
```

اضافه کردن به syscall.c:

```
107   extern int sys_move_file(void);
```

```
134   [SYS_move_file]   sys_move_file,
```

چون سیستم‌کال‌های مربوط به پردازه در sys_proc.c و سیستم‌کال‌های مربوط به فایل در sysfile.c قرار دارند، پس این سیستم‌کال را در sysfile.c تعریف می‌کنیم. اکثر توابع این فایل هم آرگومان‌ها را از استک می‌گیرند و هم منطق را پیاده می‌کنند پس ما هم همین کار را می‌کنیم. چون سیستم‌کال جز بنیادین دستورات سیستم‌عامل است پس تصمیم گرفتیم از سیستم‌کالی در پیاده‌سازی move_file استفاده نکنیم پس با کمک پیاده‌سازی بقیه سیستم‌کال‌ها، سیستم‌کال move_file را پیاده‌سازی می‌کنیم. به صورت کلی این تابع فایل مبدا و فایل جدیدی در پوشه مقصد باز می‌کند و فایل مبدا را در فایل مقصد کپی می‌کند و فایل مبدا را حذف می‌کند.

کد در فایل sysfile.c:

```c
int
sys_move_file(void)
{
  char *src, *dest;
  if(argstr(0, &src) < 0 || argstr(1, &dest) < 0) {
    cprintf("Error: Invalid source or destination path\n");
    return -1;
  }

  // OPEN SRC
  int src_fd;
  struct file *src_f;
  struct inode *src_ip;
  begin_op();
  if((src_ip = namei(src)) == 0) {
    cprintf("Error: Source file not found\n");
    end_op();
    return -1;
  }
  ilock(src_ip);
  if((src_f = filealloc()) == 0 || (src_fd = fdalloc(src_f)) < 0) {
    if(src_f)
      fileclose(src_f);
    iunlockput(src_ip);
    cprintf("Error: Unable to allocate source file or descriptor\n");
    end_op();
    return -1;
  }
  iunlock(src_ip);
  src_f->type = FD_INODE;
  src_f->ip = src_ip;
  src_f->off = 0;
  src_f->readable = 1;
  src_f->writable = 0;
```

```c
  // OPEN DEST
  int dest_fd;
  struct file *dest_f;
  struct inode *dest_ip;
  dest_ip = create(dest, T_FILE, 0, 0);
  if(dest_ip == 0) {
    cprintf("Error: Unable to create destination file\n");
    end_op();
    return -1;
  }
  if((dest_f = filealloc()) == 0 || (dest_fd = fdalloc(dest_f)) < 0) {
    if(dest_f)
      fileclose(dest_f);
    fileclose(src_f);
    iunlockput(dest_ip);
    cprintf("Error: Unable to allocate destination file or descriptor\n");
    end_op();
    return -1;
  }
  iunlock(dest_ip);
  dest_f->type = FD_INODE;
  dest_f->ip = dest_ip;
  dest_f->off = 0;
  dest_f->readable = 0;
  dest_f->writable = 1;
```

```c
  // COPY SRC TO DEST
  int n_read;
  char buffer[1024] = {0};
  while ((n_read = fileread(src_f, buffer, sizeof(buffer))) > 0) {
      if (filewrite(dest_f, buffer, n_read) != n_read) {
          fileclose(src_f);
          fileclose(dest_f);
          cprintf("Error: Failed to write to destination file\n");
          end_op();
          return -1;
      }
  }

  // CLEAN
  myproc()->ofile[src_fd] = 0;
  fileclose(src_f);
  myproc()->ofile[dest_fd] = 0;
  fileclose(dest_f);
  struct inode *ip, *dp;
  struct dirent de;
  char name[DIRSIZ];
  uint off;
  if((dp = nameiparent(src, name)) == 0) {
    cprintf("Error: Unable to locate parent directory for source file\n");
    end_op();
    return -1;
  }
  ilock(dp);
```

```c
  // Cannot unlink "." or "..".
  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0) {
    cprintf("Error: Cannot delete '.' or '..'\n");
    goto bad;
  }
  if((ip = dirlookup(dp, name, &off)) == 0) {
    cprintf("Error: Source file not found in parent directory\n");
    goto bad;
  }
  ilock(ip);
  if(ip->nlink < 1) {
    panic("unlink: nlink < 1");
  }
  if(ip->type == T_DIR && !isdirempty(ip)) {
    iunlockput(ip);
    cprintf("Error: Directory is not empty\n");
    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);
  cprintf("Error: Failed to remove source file\n");
  end_op();
  return -1;
}
```

حال این سیستم‌کال را در برنامه سطح کاربر به کار می‌بریم:

```
25    else if(!strcmp(argv[1],"2"))
26    {
27      if (argc<4)
28      {
29        printf(2, "usage: test 2 src dest...\n");
30        exit();
31      }
32      if(move_file(argv[2], argv[3])==-1)
33        printf(2,"returned -1\n");
34    }
```

و تست می‌کنیم:

```
Welcome to xv6 modified by Babak-Aidin-Ali
$ mkdir srcdir
$ mkdir destdir
$ echo Test Text! >srcdir/srcfile.txt
$ ls srcdir
.              1 22 48
..             1 1 512
srcfile.txt    2 24 11
$ test 2 srcdir/srcfile.txt destdir/destfile.txt
$ ls destdir
.              1 23 48
..             1 1 512
destfile.txt   2 25 11
$ ls srcdir
.              1 22 48
..             1 1 512
$ cat destdir/destfile.txt
Test Text!
$
```

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

اضافه کردن به usys.s:

```
41    SYSCALL(sort_syscalls)
```

اضافه کردن به syscall.h:

```
25    #define SYS_sort_syscalls 24
```

اضافه کردن به syscall.c:

```
108   extern int sys_sort_syscalls(void);
```

```
135   [SYS_sort_syscalls]   sys_sort_syscalls,
```

مشابه سیستم‌کال پالیندروم، این سیستم‌کال هم به sys_proc.c اضافه می‌کنیم ولی در اینجا میخواهی آرگومان را از استک بخوانیم پس با استفاده از argint شماره پردازه را دریافت می‌کنیم.

اضافه کردن به sys_proc.c:

```
103  // Prints systemcalls invoked by a process, sorted by their number
104  int
105  sys_sort_syscalls(void)
106  {
107    int pid;
108    if(argint(0, &pid) < 0)
109      return -1;
110    return sort_syscalls(pid);
111  }
```

برای پیاده‌سازی این بخش نیاز بود ساختار داده‌ای در struct پردازه اضافه کنیم. ما آرایه 26 تایی در فایل proc.h به ساختارداده پردازه اضافه کردیم که تعداد استفاده از هر سیستم‌کال توسط این پردازه را در آن نگه می‌داریم. در اصل بدین صورت از روش counting sort استفاده کردیم و sort با پیچیدگی بهینه o(n) انجام می‌شود.

```
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52    int sc[26]; //Babak          // Array storing the number of times each system call is invoked by this process
53  };
```

دو جا در فایل proc.c تمام خانه‌های آرایه sc را صفر می‌کنیم. یک بار در تابع allocproc که باعث می‌شود وقتی پردازه جدید ایجاد می‌شود، مقادیر آرایه sc آن صفر باشد و بار دیگر در تابع wait که اگر پردازه‌ای در استیت zombie یافت شد، تمام متغیرهای آن از جمله sc صفر می‌شود.

```
  // Clear the system call history of the child.
  for (int i = 0; i < sizeof(np->sc)/sizeof(np->sc[0]); i++)
    np->sc[i]=0;
```

تابع sort_syscalls تمام پردازه‌ها را بررسی می‌کند تا پردازه مورد نظر را پیدا کند و سپس به ازای هر سیستم‌کالی که آن پردازه فراخوانده، تعداد دفعات فراخوانی را چاپ می‌کند.

اضافه کردن به proc.c:

```c
void _log_syscall(int num) //Ali
{
  struct proc *curproc = myproc();
  curproc->sc[num - 1]++;
  return;
}
int sort_syscalls(int pid) // Ali
{
  struct proc *p;
  char *syscall_names[] = {"fork", "exit", "wait", "pipe", "read", "kill", "exec", "fstat", "chdir", "dup",
                           "getpid", "sbrk", "sleep", "uptime", "open", "write", "mknod", "unlink", "link", "mkdir", "close",
                           "create_palindrome", "move_file", "sort_syscalls", "get_most_invoked_syscall", " list_all_processes"};
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      for (int i = 0; i < sizeof(p->sc) / sizeof(p->sc[0]); i++)
      {
        if (p->sc[i])
          cprintf("%d %s: %d times\n", i + 1, syscall_names[i], p->sc[i]);
      }
      release(&ptable.lock);
      return 0;
    }
  }
  cprintf("No process with id = %d!\n", pid);
  release(&ptable.lock);
  cprintf("sort_syscalls system call failed\n");
  return -1;
}
```

تابع log_syscall_ یکی به تعداد فراخوانی سیستم‌کال مشخصی در پردازه فعلی اضافه می‌کند. وقتی که پس از فراخوانی سیستمی به تابع syscalls رفتیم این تابع را صدا می‌کنیم.

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

  num = curproc->tf->eax;
  _log_syscall(num);
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

برنامه سطح کاربر را تغییر می‌دهیم تا از این سیستم‌کال استفاده کنیم.

```
else if(!strcmp(argv[1],"3"))
{
  if (argc<3)
  {
    printf(2, "usage: test 3 pid...\n");
    exit();
  }
  if(sort_syscalls(atoi(argv[2]))==-1)
    printf(2,"returned -1\n");
}
```

تست سیستم‌کال:

```
$ test 0
Process ID: 3
$ test 0
Process ID: 4
$ test 3 5
7 exec: 1 times
11 getpid: 1 times
12 sbrk: 1 times
24 sort_syscalls: 1 times
$ test 3 100
sort_syscalls system call failed
```

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

اضافه کردن به usys.s:

```
42  SYSCALL(get_most_invoked)
```

اضافه کردن به syscall.h:

```
26  #define SYS_get_most_invoked 25
```

اضافه کردن به syscall.c:

```
109  extern int sys_get_most_invoked(void);
```

```
137      [SYS_get_most_invoked] sys_get_most_invoked,
```

مشابه بخش قبل به sys_proc.c اضافه کرده و ورودی را دریافت می‌کنیم.

اضافه کردن به sys_proc.c:

```
113  // Prints most invoked systemcall by a process
114  int
115  sys_get_most_invoked(void)
116  {
117    int pid;
118    if(argint(0, &pid) < 0)
119      return -1;
120    return get_most_invoked(pid);
121  }
```

تابع get_most_invoked تمام پردازه‌ها را بررسی می‌کند تا پردازه مورد نظر را پیدا کند؛ سپس تعداد همه سیستم‌کالهایی که آن پردازه فراخوانده بررسی کرده تا بیشترین را بیابد، در نهایت فراخوانی مورد نظر را با تعداد دفعات فراخوانی ها را چاپ می‌کند. همچنین این تابع در صورت پیدا نشدن پردازه مورد نظر و یا فراخوانی نشدن سیستم‌کالی از آن پردازه شرایط را به کاربر اطلاع می‌دهد.

اضافه کردن به proc.c:

```
// Ali
int get_most_invoked(int pid)
{
  struct proc *p;
  int max = 0;
  int max_i = -1;
  char *syscall_names[] = {"fork", "exit", "wait", "pipe", "read", "kill", "exec", "fstat", "chdir", "dup",
                           "getpid", "sbrk", "sleep", "uptime", "open", "write", "mknod", "unlink", "link", "mkdir", "close",
                           "create_palindrome", "move_file", "sort_syscalls", "get_most_invoked_syscall", " list_all_processes"};
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      for (int i = 0; i < sizeof(p->sc) / sizeof(p->sc[0]); i++)
      {
        if (p->sc[i] > max)
        {
          max = p->sc[i];
          max_i = i;
        }
      }
      if (max == 0)
        cprintf("No system call in process %d!\n", pid);
      else
        cprintf("Most invoked system call in process %d %s: %d times\n", pid, syscall_names[max_i], max);
      release(&ptable.lock);
      return 0;
    }
  }
  cprintf("No process with id = %d!\n", pid);
  release(&ptable.lock);
  cprintf("get_most_invoked_call system call failed\n");
  return -1;
}
```

برنامه سطح کاربر را تغییر می‌دهیم تا از این سیستم‌کال استفاده کنیم.

```
else if(!strcmp(argv[1],"4"))
{
  if (argc<3)
  {
    printf(2, "usage: test 4 pid...\n");
    exit();
  }
  getpid();
  getpid();
  getpid();
  if(get_most_invoked(atoi(argv[2]))==-1)
    printf(2,"returned -1\n");
}
```

تست سیستم‌کال:

```
$ test 4 100
No process with id = 100!
get_most_invoked_call system call failed
returned -1
$ test 4 0
No system call in process 0!
$ test 4 1
Most invoked system call in process 1 write: 61 times
$
```

4. پیاده سازی فراخوانی سیستمی لیست کردن پردازه ها:

اضافه کردن به usys.s:

```
43  SYSCALL(list_all_processes)
```

اضافه کردن به syscall.h:

```
27  #define SYS_list_all_processes 26
```

اضافه کردن به syscall.c:

```
110  extern int sys_list_all_processes(void);
```

```
138    [SYS_list_all_processes] sys_list_all_processes,
```

اضافه کردن به sys_proc.c:

```
123  // List all processes and count of their systemcalls
124  int
125  sys_list_all_processes(void)
126  {
127    return list_all_processes();
128  }
```

تابع list_all_processes برای تمام پردازه‌های موجود تعداد همه سیستم‌کالهایی که آن پردازه فراخوانده را جمع کرده، سپس تعداد دفعات فراخوانی ها را به همراه مشخصات پردازه چاپ می‌کند.

اضافه کردن به proc.c:

```c
// Aidin
int list_all_processes(void)
{
  struct proc *p;
  int sum = 0;
  int p_count = 1;
  int proc_flag = 0;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid)
    {
      proc_flag = 1;
      sum = 0;
      for (int i = 0; i < sizeof(p->sc) / sizeof(p->sc[0]); i++)
      {
        sum += p->sc[i];
      }
      cprintf("%d. %s (id = %d): %d syscalls called\n", p_count, p->name, p->pid, sum);
      p_count++;
    }
  }
  release(&ptable.lock);
  if (proc_flag)
    return 0;
  cprintf("No processes to show");
  return -1;
}
```

برنامه سطح کاربر را تغییر می‌دهیم تا از این سیستم‌کال استفاده کنیم.

```c
else if(!strcmp(argv[1],"5"))
{
    if (list_all_processes() == -1)
        printf(2,"returned -1\n");
}
```

تست سیستم‌کال:

```
Welcome to xv6 modified by Babak-Aidin-Ali
$ test 5
1. init (id = 1): 69 syscalls called
2. sh (id = 2): 14 syscalls called
3. test (id = 3): 4 syscalls called
$ _
```