Repository Link: https://github.com/SamanEN/Operating-System-Lab-Projects
Latest Commit Hash: ef63db9e7cbc72d1dceb9643540cea60f79d3a4b

مقدمه

1. بررسی استفاده از فراخوانی‌های سیستمی در کتابخانه‌ها (متغیر ULIB)

متغیر ULIB در Makefile متشکل از 4 آبجکت فایل به صورت زیر است:

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

سورس هر کدام از فایل‌ها را بررسی می‌کنیم:

- ulib.c : این فایل شامل توابع کمکی:

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

که در user.h دیکلر شده‌اند می‌باشد. از بین اینها، در دو تابع gets و stat از فراخوانی سیستمی استفاده شده است.

در تابع gets، از تابع سیستمی read استفاده شده است چون قرار است که یک خط از stdin را بخواند.

در تابع stat، از توابع سیستمی open, fstat, close استفاده شده است. به ترتیب، ابتدا با open فایلی باز می‌کند، سپس با fstat، metadata آن فایل (فیلدهای struct fstat مانند سایز فایل) را می‌گیرد. در نهایت با استفاده از close فایل را می‌بندد.

- usys.S : اینجا usys.o با استفاده از کد اسمبلی تولید می‌شود. در ابتدای این فایل یک ماکرو داریم:

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

سپس به ازای هر سیستم کال، این ماکرو با name آن استفاده می‌شود. مثلا برای SYSCALL(read) داریم:

```
.globl read;
read:
  movl $SYS_read, %eax;  # SYS_read (syscall.h) == 5
  int $T_SYSCALL;        # T_SYSCALL (traps.h) == 64
  ret
```

که همان اینستراکشن‌های لازم برای یک سیستم کال است.

برای فراخوانی این labelها در سطح سی، توابعی در user.h دیکلر شده‌اند.

ابتدا عدد سیستم کال به رجیستر EAX ریخته می‌شود و سپس INT 64 زده می‌شود. پس یک interrupt رخ داده و تابع trap صدا می‌شود و از آنجا که یک system call است، به تابع syscall می‌رود. در آنجا مقدار EAX خوانده شده و می‌فهمد کدام سیستم کال را باید اجرا کند که اینجا تابع sys_read است.

- printf.c : در این فایل تابع printf تعریف شده است که دیکلر آن در user.h است.

در این فایل دو تابع کمکی static به نام‌های putc و printint هم وجود دارد که printf و printint در نهایت putc را صدا می‌زنند.

تابع putc یک کاراکتر را با استفاده از تابع سیستمی write پرینت می‌کند.

- umalloc.c : در این فایل تابع malloc تعریف شده است که دیکلر آن در user.h است.

این تابع برای تخصیص حافظه استفاده می‌شود و در نهایت با استفاده از سیستم کال `sbrk` فضای پردازه را افزایش می‌دهد.

2. انواع روش دسترسی سطح کاربر به هسته در لینوکس

دسترسی به سطح هسته با رخ دادن یک interrupt انجام می‌پذیرد. Interruptها می‌توانند نرم‌افزاری و یا سخت‌افزاری باشند که به interruptهای نرم‌افزاری trap نیز گفته می‌شود.

- Interruptهای سخت‌افزاری: اینگونه interruptها از طریق سخت‌افزارها (معمولا I/O) رخ می‌دهند و به صورت asynchronous هم اجرا می‌شوند. برای مثال زمانی که در کیبورد کلیدی را فشار می‌دهیم، موس را حرکت می‌دهیم و یا یک packet از طریق شبکه به ما می‌رسد، یک interrupt رخ می‌دهد.
- Interruptهای نرم‌افزاری (trap): این interruptها توسط یک برنامه و به صورت synchronous رخ می‌دهند. trapها انواع مختلفی دارند:
  - System Call: فراخوانی‌های سیستمی که پیش‌تر در مورد آن‌ها صحبت شده است.
  - Exception: استثناها نظیر تقسیم بر 0 و یا دسترسی بدون مجوز به حافظه.
  - Signal: سیگنال‌ها در لینکوس انواع مختلفی دارند که پرکاربردترین آن‌ها عبارتند از SIGINT و SIGKILL و SIGTERM.

در لینکوس تعدادی Pseudo-File-System نظیر proc/، dev/ و sys/ وجود دارد که یک اینترفیس برای ساختار داده‌های هسته در اختیار ما قرار می‌دهد. در نتیجه، استفاده از این فایل سیستم‌ها نیز، نیازمند دسترسی به هسته است.

سازوکار اجرای فراخوانی سیستمی

بخش سخت افزاری و اسمبلی

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

خیر؛ اگر کاربر سعی کند تله‌ای دیگر را فعال کند، xv6 به او اجازه نداده و کاربر استثناء protection exception را خواهد دید. دلیل این موضوع این است که ممکن است در برنامه کاربر مشکلی وجود داشته باشد و یا کاربر قصد سؤاستفاده داشته باشد. اگر کاربر امکان اجرای این تله‌ها را داشت به راحتی می‌توانست به هسته دسترسی داشته باشد که در نتیجه آن امنیت سیستم به خطر می‌افتاد.

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

در کل دو پشته کاربر[1] و هسته[2] داریم. هنگامی که یک تله فعال می‌شود و در نتیجه آن تغییر سطح دسترسی صورت می‌گیرد، برای آنکه سیستم بتواند به کد و ساختارهای داده هسته دسترسی یابد، باید از پشته هسته استفاده کند. بنابراین ابتدا باید esp و ss که به پشته فعلی اشاره دارند ذخیره شوند؛ پس از آن این دو رجیستر

[1] User stack

[2] Kernel stack

برای اشاره به پشته هسته استفاده خواهند شد. بعد از اتمام رسیدگی به تله، مقادیر قدیمی این دو رجیستر بازیابی شده و برنامه کاربر از همان جای قبلی ادامه خواهد یافت.
در صورتی که سطح دسترسی تغییر نیابد، از آنجا که همچنان با همان پشته قدیمی کار می‌کنیم، نیازی به ذخیره این دو رجیستر نخواهیم داشت.

بخش سطح بالا و کنترل‌کننده زبان سی تله

5. توضیح توابع دسترسی به پارامترهای فراخوانی سیستمی

توابع `argint`، `argptr` و `argstr` برای دسترسی به پارامترهای فراخوانی سیستمی تعریف شده‌اند که در بخش زیر هر یک به اختصار توضیح داده شده است. لازم به ذکر است که تمامی این توابع در مقابل آرگومان غیرمجاز، مقدار 1- را بازمی‌گردانند.

- تابع `argint`: این تابع ابتدا آدرس آرگومان n-ام ورودی در حافظه را محاسبه می‌کند. می‌دانیم استک از آدرس بیشتر به سمت آدرس کمتر پر می‌شود و همچنین آدرس نقطه بازگشت از تابع، آخرین مقداری است که در استک پوش شده است و آرگومان‌های ورودی تابع قبل از آن قرار گرفته‌اند. از طرفی، آدرس سر استک در رجیستر esp ذخیره شده است. پس می‌توان گفت آدرس آرگومان n-ام ورودی تابع از رابطه زیر بدست می‌آید:

$$ptr = esp + 4 + 4 \times n$$

در نهایت این آدرس به همراه پوینتر به حافظه مد نظر برای مقدار int به تابع `fetchint` ارسال می‌شود. این تابع ابتدا بررسی می‌کند آدرس ارسالی + 4 بایت (اندازه int) در حافظه پردازه باشد و در صورت تایید، آرگومان دوم را مقداردهی می‌کند.
- تابع `argptr`: این تابع ابتدا به کمک تابع `argint` آدرس پوینتر موردنظر را دریافت می‌کند. سپس، آرگومان سوم که سایز پوینتر است را نیز به کمک تابع `argint` دریافت می‌کند و بررسی می‌کند که پوینتر با سایز داده شده در حافظه پردازه قرار داشته باشد. در نهایت، اگر مشکلی وجود نداشت، آرگومان دوم را مقداردهی می‌کند.
- تابع `argstr`: این تابع ابتدا به کمک تابع `argint`، آدرس ابتدای رشته را مشخص می‌کند و سپس این مقدار را به تابع `fetchstr` پاس می‌دهد. این تابع ابتدا بررسی می‌کند آدرس داده شده در حافظه پردازه باشد و سپس، مقدار آرگومان دوم را برابر با این پوینتر قرار می‌دهد. در نهایت، از ابتدای پوینتر شروع به پیمایش می‌کند و در صورت رسیدن به کاراکتر نال ('\0')، طول رشته و در صورت رسیدن به انتهای حافظه پردازه، مقدار 1- را برمی‌گرداند.

تمامی این توابع بررسی می‌کنند که آدرس داده شده حتما در حافظه پردازه قرار گیرد که یک پردازه نتواند به حافظه پردازه دیگری دسترسی پیدا کند زیرا این اتفاق ممکن است باعث مشکلات امنیتی و یا باگ در پردازه‌های دیگر شود.
برای مثال می‌توانیم فراخوانی سیستمی `sys_read` را بررسی کنیم. این فراخوانی سیستمی مربوط به تابع `read` است:

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

در این تابع آرگومان دوم بافری است که مقدار خوانده شده در آن قرار می‌گیرد و آرگومان سوم برابر است با حداکثر تعداد بایت‌هایی که قرار است خوانده شود. در صورتی که سیستم عامل پیش از خواندن این تعداد

بایت به EOF برسد، عملیات خواندن از فایل را پایان می‌دهد. تابع sys_read به صورت زیر تعریف شده است:

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

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

تابع ذکر شده ابتدا به کمک تابع argfd (این تابع ابتدا با استفاده از تابع argint مقدار fd که آرگومان اول تابع read است را دریافت می‌کند و معتبر بودن این file descriptor را بررسی می‌کند) مقدار file descriptor را دریافت می‌کند، سپس ابتدا آرگومان سوم (max) را به کمک تابع argint دریافت می‌کند و در نهایت به کمک تابع argptr بررسی می‌کند کل فضای آدرس‌دهی از ابتدای پوینتر به بافر (آرگومان دوم) تا انتهای آن (به طول max)، در حافظه پردازه قرار گیرد.

اگر این بررسی انجام نمی‌شد، ممکن بود در یک برنامه از تابع read با مقدار max بزرگ و برای فایلی بزرگ استفاده شود. در این صورت، هنگام خواندن از فایل و نوشتن در بافر، سیستم عامل از حافظه پردازه خارج می‌شد و در حافظه پردازه دیگری شروع به نوشتن می‌کرد که این مورد ممکن است باعث رخ دادن مشکلات بسیار زیادی شود. البته لازم به ذکر است که اگر مقدار max از طول بافر بیشتر باشد ولی از حافظه پردازه بیرون نزند، همچنان می‌تواند باعث overflow شدن بافر و درنتیجه ایجاد باگ در پردازه شود.

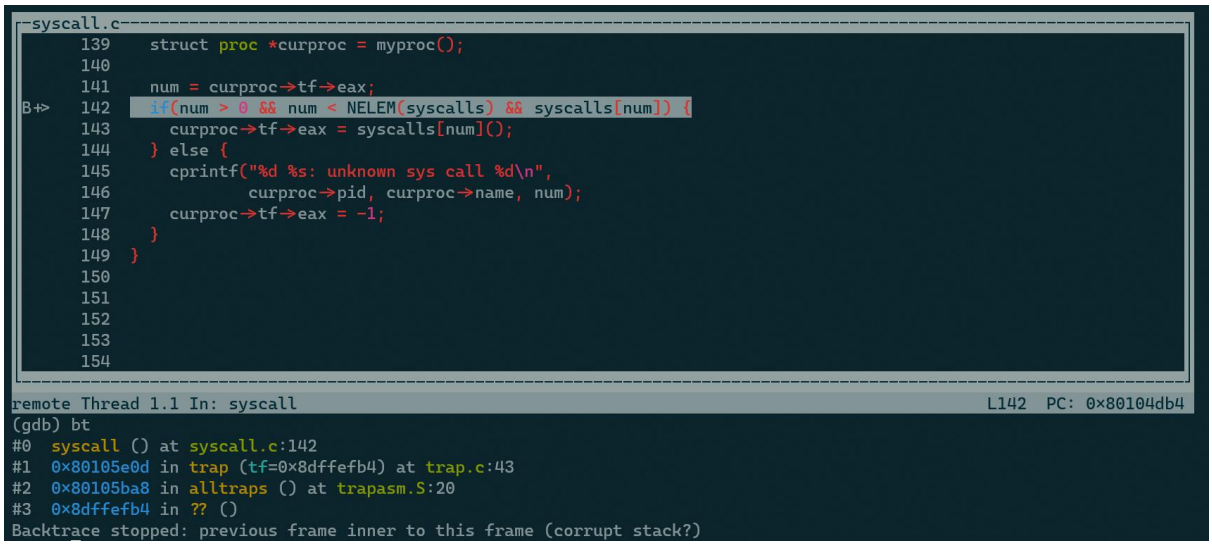
بررسی گام‌های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

ابتدا یک برنامه سطح کاربر به شکل زیر و به نام pid نوشته شد که اجرای برنامه، pid پردازه فعلی را به می‌دهد.

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

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

پس از بالا آمدن سیستم عامل، یک breakpoint در خط 142 فایل syscall.c قرار داده شد. با اجرای برنامه pid، دیباگر در خط ذکر شده متوقف می‌شود. در نهایت، با اجرای دستور bt (backtrace)، به خروجی زیر می‌رسیم:



دستور `bt` در واقع call stack برنامه در لحظه کنونی را نشان می‌دهد. هر تابعی که صدا زده می‌شود یک stack frame مخصوص به خودش را می‌گیرد که متغیرهای محلی و آدرس بازگشت و غیره در آن قرار دارند. خروجی این دستور در هر خط یک stack frame را نشان می‌دهد که به ترتیب از درونی‌ترین frame که در آن قرار داریم شروع می‌شود.

یک فراخوانی سیستمی برای تعریف و اجرا مراحل زیر را طی می‌کند:

1. در فایل syscall.h یک عدد برای سیستم کال مورد نظر انتخاب شده است.
2. در فایل user.h شناسه سیستم کال مورد نظر نوشته شده است.
3. در فایل usys.S تعریف سیستم کال در زبان اسمبلی انجام می‌شود (ابتدا شماره سیستم کال در رجیستر eax قرار می‌گیرد و سپس دستور `int 64` اجرا می‌شود).
4. تعریف `vector64` در فایل vectors.S انجام شده است که با اجرای دستور `int 64` در مرحله قبل، وارد این بخش می‌شویم. در نهایت پس از push شدن مقدار 64، به بخش `alltraps` در فایل trapasm.S هدایت می‌شویم.
5. بخش `alltraps` ابتدا trap frame مربوطه را می‌سازد و آن را در استک push می‌کند. سپس تابع trap در فایل trap.c را فراخوانی می‌کند.
6. تابع `trap` پس از اینکه متوجه می‌شود فراخوانی مربوط به یک سیستم کال است، trap frame پوش شده در استک (آرگومان تابع) را به عنوان trap frame پردازه فعلی قرار می‌دهد و سپس تابع `syscall` را صدا می‌زند.
7. تابع `syscall` در فایل syscall.c قرار دارد. در این فایل ابتدا یک آرایه syscalls تعریف شده که شماره مربوط به سیستم کال را به تابع آن مپ می‌کند. تابع `syscall` نیز پس از خواندن شماره سیستم کال که در فیلد eax در trap frame پردازه فعلی قرار دارد، تابع مربوط به آن را صدا می‌زند و خروجی این تابع را در فیلد eax در trap frame پردازه فعلی ذخیره می‌کند.

همانطور که در تصویر دیده شد، خروجی دستور bt مراحل 5 تا 7 را نشان می‌دهد (تا پیش از مرحله 5 هیچ فراخوانی تابعی وجود نداشته و در نتیجه در call stack نیز داده‌ای وجود ندارد).

احتمالا منظور از استفاده از دستور `down`، در واقع دستور `up` بوده است زیرا زمانی که در داخلی‌ترین frame قرار داریم، در صورت استفاده از دستور `down` به ارور زیر برمی‌خوریم:

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

با استفاده از دستور up می‌توانیم به یک frame (تابع) عقب‌تر بازگردیم که در اینجا نقطه فراخوانی تابع syscall در تابع trap فایل trap.c مد نظر است:

```
-trap.c----------------------------------------------------------------------
      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
      49    switch(tf->trapno){
      50    case T_IRQ0 + IRQ_TIMER:
      51      if(cpuid() == 0){
remote Thread 1.1 In: trap                                 L43   PC: 0x80105e0d
(gdb) bt
#0  syscall () at syscall.c:142
#1  0x80105e0d in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80105ba8 in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) up
#1  0x80105e0d in trap (tf=0x8dffefb4) at trap.c:43
```

می‌دانیم شماره فراخوانی سیستمی getpid برابر با 11 است. با خواندن محتوای رجیستر eax به مقدار 5 می‌رسیم که شماره سیستم کال مد نظر ما نیست:

```
(gdb) print myproc()->tf->eax
$1 = 5
```

دلیل این اتفاق این است که پیش از رسیدن به فراخوانی سیستمی getpid، فراخوانی‌های سیستمی دیگری نظیر read (برای خواندن دستور تایپ شده در ترمینال) اجرا می‌شوند. با اجرای چندباره دستور c (continue) و خواندن محتوای رجیستر eax، موارد زیر به ترتیب طی می‌شوند:

1- سیستم کال شماره 5 (read): این سیستم کال چندین بار اجرا می‌شود تا دستور تایپ شده در ترمینال به طور کامل خوانده شود.
2- سیستم کال شماره 1 (fork): این سیستم کال برای ایجاد پردازه جدید جهت اجرای برنامه سطح کاربر اجرا می‌شود.
3- سیستم کال شماره 12 (sbrk): این سیستم کال جهت تخصیص حافظه به پردازه ایجاد شده اجرا می‌شود.
4- سیستم کال شماره 7 (exec): این سیستم کال برای اجرای برنامه pid در پردازه ایجاد شده اجرا می‌شود.
5- سیستم کال شماره 3 (wait): این سیستم کال در پردازه پدر اجرا می‌شود و هدف آن، انتظار برای پایان یافتن اجرای پردازه فرزند (pid) است.
6- سیستم کال شماره 11 (getpid): این سیستم کال مربوط به برنامه سطح کاربر ذکر شده است.

پس از این مرحله، تعدای سیستم کال دیگر جهت چاپ خروجی برنامه در ترمینال اجرا می‌شود.

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$1 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$2 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$3 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$4 = 1
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$5 = 12
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$6 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$7 = 3
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) p num
$8 = 11
```

در نهایت خروجی برنامه pid به شکل زیر خواهد بود:

```
init: starting sh
Group 1:
- Saman Eslami Nazari : 810199375
- Pasha Barahimi      : 810199385
- Misagh Mohaghegh    : 810199484
$ pid
Process ID: 3
```

find_largest_prime_factor

برای اضافه کردن این فراخوانی سیستمی، در ابتدا تابع در دسترس کاربر را در user.h دیکلر می‌کنیم:

```
int find_largest_prime_factor(void);
```

این تابع باید در اصل ورودی int بگیرد ولی از آنجا که می‌خواهیم آرگومان‌ها را با استفاده از رجیسترها پاس بدهیم در خود تابع ورودی‌ای نمی‌گیریم و از استک استفاده نمی‌کنیم.

حال تعریف این تابع را در usys.S انجام می‌دهیم:

```
SYSCALL(find_largest_prime_factor)
```

ماکرو SYSCALL این خط را به کد زیر تبدیل می‌کند:

```
.globl find_largest_prime_factor;
find_largest_prime_factor:
  movl $SYS_find_largest_prime_factor, %eax;
  int $T_SYSCALL;
  ret
```

اینجا SYS_find_largest_prime_factor عدد سیستم کال می‌باشد که در syscall.h باید اضافه کنیم:

```
#define SYS_find_largest_prime_factor 22
```

حال باید تابع در سطح کرنل را پیاده‌سازی کنیم. ابتدا دیکلر تابع را در syscall.c می‌نویسیم و سپس آن را به آرایه مپ شماره سیستم کال به تابع اضافه می‌کنیم:

```
extern int sys_find_largest_prime_factor(void);
```

و در انتهای آرایه syscalls:

```
[SYS_find_largest_prime_factor] sys_find_largest_prime_factor,
```

تعریف تابع‌های سیستمی در فایل‌های sysproc.c و sysfile.c بنا بر عملکرد دستور قرار گرفته‌اند. از آنجا که این تابع ربطی به آن دو ندارد، آن را در فایل جدید sysutils.c قرار می‌دهیم:

```
int sys_find_largest_prime_factor(void) {
    return largest_prime_factor(myproc()->tf->ebx);
}
```

این تابع مقدار رجیستر ebx که به عنوان رجیستر آرگومان اول انتخاب کرده‌ایم را گرفته و به تابع استتیک `largest_prime_factor` که در همین فایل تعریف شده و محاسبات ریاضی برای به دست آوردن بزرگ‌ترین مقسوم‌علیه اول را انجام می‌دهد، داده می‌شود. در صورتی که عدد از 1 کمتر باشد تابع 1- ریترن می‌کند.

از آنجا که فایل جدیدی ساختیم باید `sysutils.o` را به متغیر OBJS اول Makefile اضافه کنیم.

حال برای تست و اجرای این فراخوانی سیستمی، یک برنامه سطح کاربر می‌سازیم که در آرگومان کامندلاین عددی را می‌گیرد و بزرگ‌ترین مقسوم‌علیه اول آن را پرینت می‌کند. فایل find_largest_prime_factor.c را ساخته و `_find_largest_prime_factor` را به متغیر UPROGS در Makefile اضافه می‌کنیم.

قسمت مهم این فایل این تابع است:

```
int flpf_syscall(int num) {
    int prev_ebx;

    // Save ebx in prev_ebx to restore later.
    // Move num to ebx.
    asm volatile(
        "movl %%ebx, %0\n\t"
        "movl %1, %%ebx"
        : "=r"(prev_ebx)
        : "r"(num)
    );

    int result = find_largest_prime_factor();

    // Restore ebx.
    asm volatile(
        "movl %0, %%ebx"
        :: "r"(prev_ebx)
    );

    return result;
}
```

برای اجرای سیستم کال `find_largest_prime_factor` که در user.h دیکلر کرده‌ایم، باید به صورت دستی، آرگومان که عدد مورد نظر است را به رجیستر EBX بریزیم.
برای این کار از GCC Extended Inline Assembly استفاده می‌کنیم و در ابتدا، مقدار کنونی رجیستر EBX را در متغیری ذخیره کرده و مقدار آرگومان را در آن می‌ریزیم. سپس سیستم کال را انجام می‌دهیم و مقدار رجیستر را به حالت قبلی‌اش بر می‌گردانیم.
نمونه اجرای برنامه:

```
$ find_largest_prime_number
usage: find_largest_prime_factor <number>
$ find_largest_prime_number 0
Number should be greater than 1.
$ find_largest_prime_number 10
5
$ find_largest_prime_number 276
23
```

پیاده‌سازی فراخوانی‌های سیستمی

1. فراخوانی سیستمی تغییر سایز فایل

ابتدا شناسه فراخوانی سیستمی را به فایل user.h اضافه می‌کنیم:

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

لازم به ذکر است که خروجی تابع به جای void، int در نظر گرفته شده تا در صورت وجود ارور متوجه شویم.
سپس برای این فراخوانی سیستمی شماره 23 را در فایل syscall.h در نظر می‌گیریم:

```
#define SYS_change_file_size 23
```

حال تعریف تابع را در فایل usys.S و به کمک ماکرو SYSCALL انجام می‌دهیم:

```
SYSCALL(change_file_size)
```

ماکرو ذکر شده تبدیل به کد زیر می‌شود:

```
.globl change_file_size;
get_parent_pid:
  movl $SYS_change_file_size, %eax;
  int $T_SYSCALL;
  ret
```

حال باید این فراخوانی سیستمی را در سطح هسته تعریف کنیم. ابتدا شناسه تابع را در فایل syscall.c اضافه می‌کنیم:

```
extern int sys_change_file_size(void);
```

حال باید شماره فراخوانی سیستمی را به این تابع مپ کنیم. برای این کار، خط زیر را به تعریف آرایه syscalls اضافه می‌کنیم:

```
[SYS_change_file_size] sys_change_file_size,
```

برای تعریف این تابع، از فایل sysfile.c استفاده می‌کنیم زیرا فراخوانی سیستمی ذکر شده مربوط به فایل است. ابتدا تابع `sys_change_file_size` را در این فایل می‌نویسیم:

```
int
sys_change_file_size(void)
{
  char *path;
  int n, r;
  struct file *f;
  struct inode *ip;

  if(argstr(0, &path) < 0 || argint(1, &n) < 0)
    return -1;

  // some commands to open file

  r = filechangesize(f, n);
  fileclose(f);
  return r;
}
```

سپس تابع `filechangesize` را در فایل file.c تعریف کرده و شناسه آن را در فایل defs.h وارد می‌کنیم:

```
int
filechangesize(struct file *f, int n)
{
  int r;

  if(f->writable == 0)
    return -1;
  if(f->type == FD_INODE){
    begin_op();
    ilock(f->ip);
    if ((r = changesize(f->ip, n)) > 0)
      f->off = r;
    iunlock(f->ip);
    end_op();
    return r;
  }
  panic("filechangesize");
}
```

در نهایت تابع `changesize` را در fs.c تعریف می‌کنیم و شناسه آن را نیز فایل defs.h وارد می‌کنیم:

```
int
changesize(struct inode *ip, uint size)
{
  uint n, off, tot, m;
  struct buf *bp;

  if (ip->type != T_FILE)
    return -1;
  if (size > ip->size) {
    n = size - ip->size;
    off = ip->size;
  } else {
    n = ip->size - size;
    off = size;
  }
  for(tot=0; tot<n; tot+=m, off+=m){
      bp = bread(ip->dev, bmap(ip, off/BSIZE));
      m = min(n - tot, BSIZE - off%BSIZE);
      memset(bp->data + off%BSIZE, 0, m);
      log_write(bp);
      brelse(bp);
  }
  ip->size = size;
  iupdate(ip);
  return size;
}
```

برای تست کردن این فراخوانی سیستمی یک برنامه سطح کاربر به نام change_file_size می‌نویسیم و آن را به متغیر UPROGS در Makefile نیز اضافه می‌کنیم:

```
int main(int argc, char* argv[]) {
    if (argc < 3) {
        printf(1, "change_file_size: 2 args required\n");
        exit();
    }
    char *path = argv[1];
    int size = atoi(argv[2]);
    if (size < 0) {
        printf(1, "change_file_size: invalid size %d\n", size);
        exit();
    }

    int prevSize = fileSize(path);
    if (change_file_size(path, size) < 0) {
        printf(1, "change_file_size: cannot change file size\n");
        exit();
    }
    int newSize = fileSize(path);

    printf(1, "File size changed from %d to %d\n", prevSize,
newSize);
    exit();
}
```

خروجی این برنامه در تصویر زیر مشخص شده است:

```
init: starting sh
Group 1:
- Saman Eslami Nazari : 810199375
- Pasha Barahimi      : 810199385
- Misagh Mohaghegh    : 810199484
$ echo test0test > test.txt
$ ls test.txt
test.txt        2 24 10
$ change_file_size test.txt 5
File size changed from 10 to 5
$ ls test.txt
test.txt        2 24 5
$ cat test.txt
test0$ change_file_size test.txt 15
File size changed from 5 to 15
$ ls test.txt
test.txt        2 24 15
$ cat test.txt
test0$ change_file_size test.txt 0
File size changed from 15 to 0
$ ls test.txt
test.txt        2 24 0
$ cat test.txt
$
```

2. فراخوانی سیستمی لیست پردازه‌های فراخواننده

در ابتدا، برای اضافه کردن این فراخوانی سیستمی جدید، تابع را در فایل user.h دیکلر می‌کنیم تا در دسترس کاربر قرار گیرد:

```
void get_callers(int);
```

پس از آن تعریف این تابع را در usys.S به صورت زیر انجام می‌دهیم:

```
SYSCALL(get_callers)
```

این ماکرو، خط فوق را به کد زیر تبدیل می‌کند:

```
.globl get_callers;
get_callers:
  movl $SYS_get_callers, %eax;
  int $T_SYSCALL;
  ret
```

در تکه کد بالا مقدار `SYS_get_callers` در رجیستر eax قرار می‌گیرد. این مقدار باید در فایل syscall.h دیفاین شود:

```
#define SYS_get_callers 24
```

سپس دیکلر تابع `sys_get_callers` را در فایل syscalls.c انجام می‌دهیم و پوینتر این تابع را به عدد دیفاین شده مپ می‌کنیم:

```
extern int sys_get_callers(void);
[SYS_get_callers] sys_get_callers
```

در ادامه به پیاده‌سازی این تابع می‌پردازیم؛ ابتدا باید یک تاریخچه از تمام فراخوانی‌های سیستمی توسط پردازه‌ها نگه داریم. برای این کار آرایه `p_hist` در فایل proc.c تعبیه شده است. اضافه کردن فراخوانی‌های جدید به صورت دایره‌وار صورت می‌گیرد و در صورت پر شدن آرایه، از ابتدای آن دوباره شروع به نوشتن می‌کنیم. اضافه شدن فراخوانی‌ها به تاریخچه دستورات توسط تابع `push_p_hist` انجام می‌گیرد:

```
void
push_p_hist(int pid, int syscall_number) {
  int cur_size = p_hist[syscall_number].size % PROC_HIST_SIZE;
  p_hist[syscall_number].pids[cur_size] = pid;
  ++(p_hist[syscall_number].size);
}
```

این تابع توسط تابع `syscall`، هر بار پس از آنکه یک فراخوانی سیستمی صورت می‌گیرد، صدا زده می‌شود. بنابراین فراخوانی این تابع را به `syscall` نیز اضافه کردیم. پس از آن تابع `get_callers` در فایل proc.c را تعریف کردیم که وظیفه آن چاپ تاریخچه مربوط به فراخوانی سیستمی خواسته شده است:

```
void
get_callers(int syscall_number) {
  int limit = p_hist[syscall_number].size;

  if(limit == 0) {
    cprintf("No process has called system call number %d.\n",
syscall_number);
    return;
  }

  int i = (limit > PROC_HIST_SIZE) ? limit % PROC_HIST_SIZE : 0;
  limit %= PROC_HIST_SIZE;
  while(1) {
    cprintf("%d", p_hist[syscall_number].pids[i]);
    i = (i + 1) % PROC_HIST_SIZE;
    if(i == limit) break;
    cprintf(", ");
  }
  cprintf("\n");
}
```

این تابع در `sys_get_callers` صدا زده می‌شود:

```
int
sys_get_callers(void) {
  int sys_call_number;
  if(argint(0, &sys_call_number) < 0)
    return -1;

  get_callers(sys_call_number);
  return 0;
}
```

برای تست کردن این فراخوانی سیستمی جدید، یک برنامه سطح کاربر ایجاد می‌کنیم که در آن برای سه فراخوانی سیستمی `write`، `wait` و `fork` تاریخچه را چاپ می‌کنیم. این برنامه در فایل get_callers_test.c و به صورت زیر می‌باشد:

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

int main(int argc, char* argv[]) {
    printf(1, "Testing SYS_write:\n");
    get_callers(SYS_write);

    printf(1, "Testing SYS_fork:\n");
    get_callers(SYS_fork);

    printf(1, "Testing SYS_wait:\n");
    get_callers(SYS_wait);

    exit();
}
```

خروجی این برنامه در صورتی که به عنوان اولین دستور پس از اجرای سیستم عامل استفاده شود به صورت زیر خواهد بود:

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group 1:
- Saman Eslami Nazari : 810199375
- Pasha Barahimi      : 810199385
- Misagh Mohaghegh    : 810199484
$ get_callers_test
Testing SYS_write:
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
Testing SYS_fork:
1, 2
Testing SYS_wait:
1, 2
$ 
```

دلیل خروجی بخش `SYS_fork` این است که ابتدا پردازه `init` با شماره 1 شروع به کار می‌کند. پس از آن در فایل init.c یک پردازه `fork()` شده که در نهایت همان shell خواهد بود. شماره پردازه shell برابر 2 می‌باشد. سپس برنامه get_callers_test توسط shell با استفاده از فراخوانی سیستمی `fork()` ایجاد می‌شود که دلیل وجود پردازه 2 در لیست `SYS_fork` همین است.

3. فراخوانی سیستمی گرفتن پردازه پدر

برای اضافه کردن این فراخوانی سیستمی، در ابتدا تابع در دسترس کاربر را در user.h دیکلر می‌کنیم:

```c
int get_parent_pid(void);
```

این تابع آرگومان ورودی‌ای نمی‌گیرد. حال تعریف این تابع را در usys.S انجام می‌دهیم:

```
SYSCALL(get_parent_pid)
```

ماکرو SYSCALL این خط را به کد زیر تبدیل می‌کند:

```
.globl get_parent_pid;
get_parent_pid:
  movl $SYS_get_parent_pid, %eax;
  int $T_SYSCALL;
  ret
```

اینجا SYS_get_parent_pid عدد سیستم کال می‌باشد که در syscall.h باید اضافه کنیم:

```
#define SYS_get_parent_pid 25
```

حال باید تابع در سطح کرنل را پیاده‌سازی کنیم. ابتدا دیکلر تابع را در syscall.c می‌نویسیم و سپس آن را به آرایه مپ شماره سیستم کال به تابع اضافه می‌کنیم:

```
extern int sys_get_parent_pid(void);
```

و در انتهای آرایه syscalls:

```
[SYS_get_parent_pid] sys_get_parent_pid,
```

تعریف تابع‌های سیستمی در فایل‌های sysproc.c و sysfile.c بنا بر عملکرد دستور قرار گرفته‌اند. از آنجا که این تابع به process ربط دارد، آن را در فایل sysproc.c قرار می‌دهیم:

```
int
sys_get_parent_pid(void)
{
  return myproc()->parent->pid;
}
```

این تابع فیلد parent از پردازه کنونی (که در `struct proc` ذخیره شده است و تایپ اش نیز همین است) را گرفته و فیلد pid آن را ریترن می‌کند.

حال برای تست و اجرای این فراخوانی سیستمی، یک برنامه سطح کاربر می‌سازیم که سه نسل پردازه می‌سازد و برای پردازه‌های دوم و سوم، PID پدرشان را پرینت می‌کند. فایل get_parent_pid_test.c را ساخته و `_get_parent_pid_test` را به متغیر UPROGS در Makefile اضافه می‌کنیم.

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

void third() {
    printf(1, "3rd Process:\n  PID: %d\n  Parent: %d\n", getpid(),
get_parent_pid());
    exit();
}

void second() {
    printf(1, "2nd Process:\n  PID: %d\n  Parent: %d\n", getpid(),
get_parent_pid());
    int forkpid = fork();
    if (forkpid > 0) {
        wait();
    }
    else if (forkpid == 0) {
        third();
    }
    else {
        printf(2, "Failed to create 3rd process.\n");
    }
    exit();
}

int main(int argc, char* argv[]) {
    int forkpid = fork();
    if (forkpid > 0) {
        wait();
    }
    else if (forkpid == 0) {
        second();
    }
    else {
        printf(2, "Failed to create 2nd process.\n");
    }
    exit();
}
```

با استفاده از تابع سیستمی `fork`، یک بار در پردازه اول و باری در پردازه دوم، سه نسل پردازه تولید می‌کنیم. پس از هر `fork` در پدر، از تابع `wait` استفاده می‌کنیم که تا خروج پردازهٔ فرزند صبر کند. در فرزندان با استفاده از سیستم کال‌های `getpid` و `get_parent_pid` که پیاده‌سازی کردیم مقادیر را پرینت می‌کنیم.
نمونه اجرای برنامه:

```
$ get_parent_pid_test
2nd Process:
  PID: 5
  Parent: 4
3rd Process:
  PID: 6
  Parent: 5
```