Repository Link: <https://github.com/SamanEN/Operating-System-Lab-Projects>

Latest Commit Hash: <hash>

# مقدمه

## 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 label ها 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.

## 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

## آیا همه تله‌ها را می‌شود با سطح دسترسی DPL\_USER فعال نمود؟

خیر؛ اگر کاربر سعی کند تله‌ای دگیر 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 سواستفاده 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 می‌افتد.

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

# بخش سطح بالا و کنترل‌کننده زبان سی تله

## 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:

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

# 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.

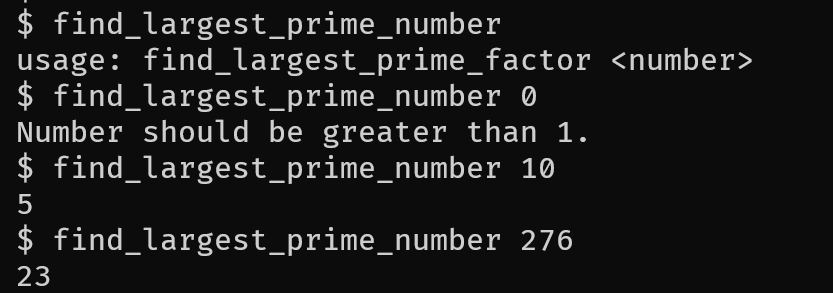
قسمت مهم این فایل این تابع است:

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 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 قبلی اش بر می‌گردانیم.

نمونه اجرای برنامه:



# پیاده‌سازی فراخوانی‌های سیستمی

## فراخوانی سیستمی تغییر سایز فایل

## فراخوانی سیستمی لیست پردازه‌های فراخواننده

## فراخوانی سیستمی گرفتن پردازه پدر

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

int get\_parent\_pid(void);

این تابع آرگومان ورودی ای 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

O

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.h 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;  
}

return largest\_prime\_factor(myproc()->tf->ebx);  
}

O

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.

#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 پردازۀ 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:

