پروژه‌ی دوم سیستم عامل

روزبه بستان‌دوست - پویا نقوی - صادق حایری

گام اول:

- userها نمی توانند در سطح خود کارهایی از قبیل دسترسی به سخت افزار کامپیوتر، دسترسی به فایل ها و مدیریت آن ها، اجرا کردن برنامه، دسترسی به memory، عملیات های I\O، ساخت و ارتباط بین process ها، دسترسی به منابع، محاسبات و... را انجام دهند. این کارها در سطح کرنل انجام می پذیرند.

۲- فراخوانی های سیستم به ۵ دسته تقسیم بندی می شوند:

الف) Process control (کنترل پردازه ها): کارهایی از قبیل ایجاد کردن و خاتمه دادن processها، load کردن و اجرا کردن آن‌ها، گرفتن و یا setکردن خواص پردازه، اختصاص و یا آزاد کردن مموری و ...

ب) File management (مدیریت فایل ها): کارهایی از قبیل ایجاد ویا حذف کردن فایل ها، بازکردن یا بستن فایل ها، خواندن ویا نوشتن ویا تغییر موقعیت آن ها، گرفتن ویا setکردن خواص فایل ها و...

ج) Device management (مدیریت دستگاه ها): کارهایی از قبیل درخواست برای اتصال به deviceها ویا آزاد کردن آن ها، خواندن ویا نوشتن بر روی دیوایس ها، گرفتن ویا setکردن خواص دیوایس ها و...

د) Information maintenance (حفظ و نگهداری اطلاعات): کارهایی از قبیل گرفتن و یا setکردن زمان و تاریخ، گرفتن ویا setکردن دیتاهای سیستم و...

هـ) Communications (ارتباطات): کارهایی از قبیل ایجاد کردن ویا ازبین بردن ارتباط ها، ارسال ویا دریافت پیام ها، انتقال وضعیت اطلاعات و...

۳- از آنجایی که بیشتر برنامه ها و processها به دسترسی به منابع و مموری نیاز دارند، اگر دسترسی به حافظه از سطح user امکان پذیر بود، یعنی برنامه ها و processهای مختلف می توانستند به راحتی به تمام فضای حافظه دسترسی داشته باشند و در واقع می توانستند به راحتی دیتا های دیگر برنامه ها را از بین ببرند ویا با مقدارهای جدید جایگزین کنند. همان طور که می دانیم در یک سیستم تنها یک برنامه در حال اجرا نیست و تعداد متعدی برنامه و process ممکن است در حافظه قرار داشته باشند که در واقع مدیریت حافظه و منابع به این شکل امکان پذیر نبود. پس سطح دیگری به نام kernel mode اضافه شد.

۴- برای فرستادن پارامترها در سیستم کال آن ها را در ۶ رجسیتر قرار می دهیم.(ebx, ecx, edx, esi, edi, ebp). هر کدام از این رجیسترها ۳۲ بیتی هستند. یعنی به این نحو تنها ۶ پارامتر ۳۲ بیتی را می توانیم ارسال کنیم. اگر از ۳۲ بیت بیشتر بود می توانیم از ۲ تا رجیستر باهم برای یک پارامتر استفاده کنیم تا یک پارامتر ۶۴ بیتی را به عنوان مثال ارسال کنیم. هم چنین پارامترها نمی توانند از نوع floating point باشند چرا که سیستم عامل مقادیر رجیستر های floating-point را هنگامی که System Call در انحصار یک thread دیگر است را نگهداری می کند بنابراین System Call نمی تواند هیچ عملیات Floating-Point ای را استفاده کند.

۵- از آن جایی که سیستم کال بر روی استک خودش اجرا می شود پس تعداد پارامترهای ارسالی محدود است. برای ارسال بیش از ۶ پارامتر از استک استفاده می کنیم. و به خاطر این که دسترسی به استک برای سیستم کال تعریف نشده است، این قسمت باید به صورت فردی جداگانه نوشته شود.

۶- این تابع یک بلاک از داده ها را از فضای user به فضای kernel منتقل می کند. این تابع ۳ ورودی دارد. ۱. بافر مقصد یا همان کرنل. ۲. بافر مبدا یا همان user و ۳. طول به واحد بایت. در صورت موفقیت آمیز بودن این انتقال تابع مقدار صفر را برمی گرداند و در غیر این صورت مقداری غیر صفر به عنوان خروجی تابع درنظر گرفته می شود. (تعداد بایت داده ای که نتوانسته است، انتقال دهد)

```
copy_from_user( to, from, n );
```

این تابع با چک کردن قادر بودن برای خواندن از بافر user شروع به کار می کند و سپس مراحل کپی را طی می کند.
دلیل متفاوت بودن فضای آدرس کرنل و کاربر همان دلیل وجود ۲ سطح متفاوت کرنل و user است که در بالاتر ذکر شد. یعنی processای که در فضای آدرس user در حال اجرا است فقط به قسمت هایی از فضای مموری دسترسی دارد ولی process در فضای آدرس کرنل به تمام فضای مموری دسترسی دارد و هم چنین process در قسمت user به قسمت کرنل دسترسی ای ندارد.

۷-
۱) ioctl: برای اداره کردن یک سری از دیوایس ها توسط file descriptor استفاده می شود. یعنی ورودی و خروجی را کنترل می کند. یک تابعی که دیوایس و درخواست و تعدادی پارامتر دیگر می گیرد و به درخواست رسیدگی می کند. برای دیوایس هایی که در پلتفرم های خاصی هستند.

۲) procfs: یک فایل سیستم خاص برای سیستم عامل لینوکس است. یک ویرچوآل فایل سیستم که فقط در مموری قرار دارد. یک سری اطلاعات پروسس ها را به برنامه ها می دهند.

۳) Sysfs:یک ورچوآل فایل سیستم که در کرنل لینوکس ۲.۶ سرویس دهی می کند. اطلاعات دیواس ها و درایورها را از کرنل به user منتقل می کند.

۴) Netlink sockets: یک IPC خاص است که برای انتقال اطلاعات بین کرنل و user استفاده می شود. یک ارتباط دو طرفه ی کامل بین پروسس های user-space و کرنل.

۵) relay:یک نوع دیگر ویرچوآل فایل سیستم است که توسط کرنل پیاده سازی شده ولی باید در user نصب شود تا قابل دسترس باشد. استفاده از آن برای انتقال حجم زیاد داده و از فضای کرنل آسان تر است.

۶) debugfs:در فایل سیستم کرنل قرار دارد و برای قرار دادن ابزارهای دیباگینگ در جاهای خاص استفاده می شود. از procfs و sysfs هم آسان تر استفاده می شود. با اطلاعات دیباگینگ زمان توسعه را کاهش می دهد.

۷) Firmware loading.

۸- بخش مربوط به معماری سخت افزار سیستم در کرنل قرار دارد و handler function در source code کرنل قرار دارد. سیستم کال ها همان اینتراپت های نرم افزاری هستند. در کرنل یک سری توابع به نام handler function وجود دارند که هندل کردن این اینتراپت ها توسط آن ها صورت می گیرد. به ازای هر سیستم کال یک تابع برای آن وجود دارد. در کرنل لینوکس جدول خاصی به نام system call table وجود دارد که به صورت آرایه پیاده سازی شده است. وقتی سیستم کال به دنبال آدرس سیستم کال هندلر است، به سراغ این جدول می رویم. این جدول در arch/x۸۶/entry/syscall_۶۴.c قرار دارد(بستگی به معماری سیسستم دارد). این توابع تمام اطلاعات مربوطه که در رجیسترهای CPU هستند را در استک کرنل کپی می کنند. سپس توسط تابعENABLE_INTERRUPTS اینتراپت ها را دوباره اینیبل می کنیم. از شماره سیستم کال برای پیدا کردن سرویس روتین متناظر آن سیستم کال در جدول sys_call_table استفاده می شود. این سرویس روتین ها در کرنل قرار دارند.

۹- idtentry macro: برای آماده کردن شرایط قبل از وقوع اکسپشن استفاده می شود.
Interrupt macro: برای آماده کردن شرایط قبل از وقوع اینتراپت استفاده می شود
entry_SYSCALL_۶۴:.برای آماده کردن شرایط قبل از سیستم کال استفاده می شود

در واقع entry_SYSCALL_۶۴ درفایل اسمبلی arch/x۸۶/entry/entry_۶۴.S تعریف شده است و از ماکروی زیر آغاز می گردد: SWAPGS_UNSAFE_STACK
این ماکرو در هدر فایل arch/x۸۶/include/asm/irqflags.h تعریف شده است. این ماکرو توسط دستور زیر جایگزین می شود:

```
#define SWAPGS_UNSAFE_STACK    swapgs
```

سپس مقدارهای حال حاضر GS base register را با مقدار MS_KERNEL_GS_BASE جابه جا می کند. یعنی در واقع توسط این کار این مقدارها را به استک کرنل منتقل می کنیم. سپس ما پوینتر استک قبلی رو به rsp_scratch اشاره می دهیم و پوینتر استک فعلی را به بالای استکی که برای پروسسور فعلی هست اشاره می دهیم. سپس stack segment و پوینتر استک قبلی در استک فعلی پوش می شود.
سپس اینتراپت ها فعال می شوند. چرا که آن ها در هنگام entry و ذخیره شدن رجیسترهای خاص غیرفعال شده بودند. در هنگام فراخوانی سیستم کال متغیر ها حاوی اطلاعات زیر هستند:

- rax - contains system call number;
- rcx - contains return address to the user space;
- r11 - contains register flags;
- rdi - contains first argument of a system call handler;
- rsi - contains second argument of a system call handler;
- rdx - contains third argument of a system call handler;
- r10 - contains fourth argument of a system call handler;
- r8 - contains fifth argument of a system call handler;
- r9 - contains sixth argument of a system call handler;

این رجیسترها و باقی آرگومان ها در استک پوش می شوند. در نهایت اینستراکشن call صدا زده می شود.

```
call    *sys_call_table(, %rax, 8)
```

و سیستم کال متناظر را بدست می آورد.

Syscall( ):

Syscall یک تابع کتابخانه ای کوچک است که برای صدازدن system call هایی که دارای شماره(ثابت های نمادین برای شماره System Call را میتوانید در <sys/syscall.h> پیدا کنید) و آرگومان های مشخص هستند استفاده می شود.به کاربردن Syscall مفید است به عنوان مثال در مواقع صدا زدن system call هایی که wrapper function انها در کتابخانه C وجود ندارد،استفاده می شود.Syscall( ) همچنین رجیستر های پردازنده را قبل از صدازدن System Call ذخیره می کند و آن ها را دوباره هنگام برگشت از System Call بازیابی می کند و اگر خطایی رخ دهد، شناسه آن را که از طرف System Call آمده در (۳)erno ذخیره می کند.مقدار بازگشتی Syscall( ) توسط آن System Call ای که صدا زده شده است مشخص می شود ولی در کل مقدار برگشتی ۰ به معنای موفقیت و ۱- به معنای وقوع خطا است که شناسه آن هم در erno ذخیره می شود و یک مثال از استفاده از Syscall را در قطعه کد زیر مشاهده می کنید.

آن ها را دوباره هنگام برگشت از System Call بازیابی می کند و اگر خطایی رخ دهد، شناسه آن را که از طرف System Call آمده در (۳)erno ذخیره می کند.مقدار بازگشتی Syscall( ) توسط آن System Call ای که صدا زده شده است مشخص می شود ولی در کل مقدار برگشتی ۰ به معنای موفقیت و ۱- به معنای وقوع خطا است که شناسه آن هم در erno ذخیره می شود و یک مثال از استفاده از Syscall را در قطعه کد زیر مشاهده می کنید.

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <signal.h>

int
main(int argc, char *argv[])
{
    pid_t tid;

    tid = syscall(SYS_gettid);
    syscall(SYS_tgkill, getpid(), tid, SIGHUP);
}
```

قبل از syscall از دستور _syscallN استفاده میشد که مقدار N از ۰ تا ۶ تغییر می کند که تعداد پارامتر هایی می باشد که به آن پاس می دهیم.

منبع:http://man۷.org/linux/man-pages/man۲/syscall.۲.html

گام دوم:

```
365   545 common   hello_world      sys_hello_world
366   546 common   hello_there      sys_hello_there
367   547 common   hello_user       sys_hello_user
```

در فایل syscalls_۶۴.tbl سیستم‌کال‌های خودمون رو اضافه می‌کنیم.
شماره سیستم‌کال - ۳۲بیت یا ۶۴بیت بودن - اسم تابع - اسم سیستم‌کال

در فایل sys.c تعریف تابع را می‌نویسیم:

```
2180  SYSCALL_DEFINE1(hello_there, char *, name)
2181  {
2182    char buf[256];
2183    long copied = strncpy_from_user(buf, name, sizeof(buf));
2184    if (copied < 0 || copied == sizeof(buf))
2185      return -EFAULT;
2186    printk("hello \"%s\"\n!", buf);
2187    return 0;
2188  }
2189
2190  SYSCALL_DEFINE0(hello_user)
2191  {
2192    printk("hello user : \"%u\"\n!", get_current_user()->uid);
2193    return 0;
2194  }
2195
2196  SYSCALL_DEFINE0(hello_world)
2197  {
2198    printk("hello world\"\n!");
2199    return 0;
2200  }
```

تعداد آرگیومنت‌هایی که سیستم‌کال می‌گیرد و نوع‌شان.
برای کامپایل کردن و نصب کرنل دستورهای زیر را می‌زنیم:

```
make menuconfig (and save default config)
Make
Make modules_install
Make install
Update-grub
```

بعد از تنظیم گراب و آپدیت آن سیستم را ریستارت می‌کنیم و در قسمت advanced options کرنل ۳.۶ را انتخاب می‌کنیم.

(روی کرنل ۳.۶ lighdm کرش می‌کند و نمی‌توانیم با استفاده از gui وارد شویم ولی با tty مشکلی نداشت)

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>

/*
 * Put your syscall number here.
 */
#define SYS_stephen 545

int main(int argc, char **argv)
{
  if (argc <= 1) {
    printf("Must provide a string to give to system call.\n");
    return -1;
  }
  char *arg = argv[1];
  printf("Making system call with \"%s\".\n", arg);
  long res = syscall(SYS_stephen, arg);
  printf("System call returned %ld.\n", res);
  return res;
}
```

برای تست کد c کوچکی ران می‌کنیم:
که بعد از اجرا ۰ برمی‌گرداند به معنای اینکه برنامه درست کار کرده است.
برای دیدن خرجی printk باید لاگ‌های سیستم را ببینیم که با دستور زیر می‌توانیم خروجی سیستم‌کال‌های خودمان را مشاهده کنیم.

- خیر کار درستی نیست، چون ما برای هرکاری نیاز نیست کرنل را درگیر کنیم مثلا کار با متغیرها و انجام عملیاتی که انجام آنها نمی‌تواند برای برنامه‌های دیگر اشکال ایجاد کند را می‌توانیم خودمان انجام دهیم و سربار استفاده از سیستم‌کال‌ها می‌تواند زیاد باشد، همیچنین بعضی از کارها نمی‌توان در سطح کرنل انجام شوند و فقط در سطح یوزر تعریف می‌شوند.

گام سوم:

نام این فراخوانی چیست و چگونه strace را با آن می‌توان پیاده سازی کرد؟

ptrace

long int ptrace(enum __ptrace_request ***request***, pid_t ***pid***,
void * ***addr***, void * ***data***)

request: The value request determines what action needs to perform
pid: The PID of the process to be traced
addr: The address in the USER SPACE of the traced process
(1) to where the ***data*** may be written when instructed to do so, or
(2) from where a word is read and returned as the result of the ptrace system call

برای اینکه بتوانیم برنامه‌ای را با استفاده از این سیستم‌کال تریس کنیم ابتدا باید برنامه را بچه‌ی تریسر کنیم، برای این کار برنامه‌ای می‌نویسیم که یک فورک ایجاد کند سپس در قسمت بچه با درخواست PTRACE_TRACEME اطلاع می‌دهد که می‌خواد توسط پرنت دسترسی trace داشته باشد و سپس با دستور exec می‌توانیم برنامه مورد نظر خود را لود و اجرا کنیم.
در قسمت پدر منتظر می‌مانیم و هر لحظه که برنامه ما متوقف شود ما باخبر می‌شویم و می‌توانیم ببینیم چه اتفاقی افتاده است. (همچنین برای اینکه تنظیم کنیم در کجاها مارا باخبر کند، مثلا فقط برای سیستم‌کال‌ها می‌توانیم با PTRACE_SETOPTIONS این تنظیمات را انجام دهیم)

چرا فراخوانی‌های سیستمی به صورت مستقیم مورد استفاده قرار نمی‌گیرند؟

چون در این صورت کد فقط روی یک معماری خاص کار می‌کند و هر کد باید برای سیستم‌عامل‌های مختلف به صورت مجزا نوشته شوند و همچنین برای یک سیستم‌عامل هم بستگی به نوع معماری آن باید سیستم‌کال‌های خاصی به کار می‌رفت که در اینصورت برای هر سیستمی باید کد مجزایی نوشته می‌شد.
جدای از این مثلا برای malloc می‌بینیم که با توجه به مقدار مورد نیاز از سیستم‌کال‌های متفاوتی استفاده می‌شود که اگر بخواهیم از تابع‌ها استفاده نکنیم حجم کد ما بسیار زیاد می‌شود.

مسیرهای اجرایی دستور malloc کدام هستند؟

دستور malloc با توجه به مقدار فضایی که نیاز داریم دو مسیر را می‌تواند طی کند، مثلا اگه از آن بخواهیم فضای کمی (مثلا ۱۰۰بیت) را بگیرد، با استفاده از سیستم‌کال brk این‌کار را انجام می‌دهد که این دستور اندازه‌ی data segment حافظه را تغییر می‌دهد و با اینکار می‌تواند حافظه مورد نیاز را بگیرد.

```
brk(0)                                  = 0xffb000
brk(0x101c000)                          = 0x101c000
```

ولی اگر مقدار بیشتری حافظه بخواهیم این‌کار را با استفاده از سیستم‌کال mmap انجام می‌دهد که به این‌صورت عمل می‌کند که حافظه‌ای جدای برنامه در قسمت هیپ فضایی را برای برنامه ایجاد می‌کند.

```
mmap(NULL, 100003840, PROT_READ|PROT_WRITE,
 MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3b
31052000
```

کامپایل کردن برنامه با استفاده از لینک‌کردن glibc به صورت استاتیک:

برای اینکه بتوانیم کدهایمان به صورت استاتیک لینک شوند می‌توانیم کتاب‌خانه را به صورت دستی مانند روشی که گفته شد لینک‌کنیم ولی در قسمت کانفیگ کردن به مشکل برخوردیم!

ولی بجای این کار می‌توانیم از فلگ static- کامپایلرهای gcc استفاده کنیم، که با زدن این دستور کد ما به صورت استاتیک لینک می‌شود.

حال gdb را باز می‌کنیم و می‌توانیم با زدن دستور s به درون تابع‌های کتاب‌خانه نیز برویم و ببنیم کدام تابع‌ها اجرا می‌شوند.