آزمایشگاه سیستم عامل

پروژه دوم

اعضای گروه:
علیرضا کامکار 810100202
پرهام بدو 810100244
علی علم طلب 810100189

https://github.com/alirezakamkar/OS_LAB.git

مقدمه

پرسش 1: کتابخانه‌های سطح کاربر، موجود در دایرکتوری ULIB (مانند فایل‌های usys.S و ulib.c) توابع پوشاننده‌ای را ارائه می‌دهند که این فراخوانی‌های سیستمی را به صورت انتزاعی مدیریت می‌کنند.
توضیح دهید که این کتابخانه‌ها چگونه با استفاده از ماکروها و توابع پوشاننده، جزئیات فراخوانی‌های سیستمی (مانند شماره فراخوانی، آرگومان‌ها و بازگردانی مقادیر) را از برنامه‌نویس پنهان می‌کنند.
همچنین، دلیل استفاده از این انتزاع در بهبود قابلیت حمل، افزایش امنیت و ساده‌سازی توسعه برنامه‌های کاربر را بیان نمایید.

نحوه پنهان‌سازی جزئیات فراخوانی‌های سیستمی توسط ULIB:

کتابخانه‌های سطح کاربر (مانند `USys`، `SYS` و `ULIB`) با استفاده از ماکروها و توابع پوشاننده (Wrapper Functions) جزئیات فراخوانی‌های سیستمی را پنهان می‌کنند:

انتزاع شماره فراخوانی سیستمی:
به‌جای استفاده مستقیم از شماره‌های فراخوانی (مثال: `int 64`) در کد برنامه، برنامه‌نویس از توابع توصیفی مانند `fork()` یا `write()` استفاده می‌کند. این توابع پوشاننده، شماره فراخوانی سیستمی مربوطه (مثال: `SYS_write`) را در ثبات `eax` قرار داده و دستور `int 64` را اجرا می‌کنند.

مدیریت آرگومان‌ها:

توابع پوشاننده، آرگومان‌های دریافتی (مثال: `count`، `buf` برای `write`) را در ثبات‌های مشخص (مانند `edx`، `ecx`، `ebx`) قرار می‌دهند تا هسته بتواند آن‌ها را بخواند.

پردازش مقادیر بازگشتی:

پس از اجرای فراخوانی سیستمی، نتیجه در ثبات `eax` قرار می‌گیرد. کتابخانه سطح کاربر این مقدار را بررسی کرده و در صورت نیاز (مثال: خطا)، آن را به `errno` تبدیل می‌کند.

مزایای استفاده از این انتزاع:

قابلیت حمل (Portability): برنامه‌های کاربردی بدون وابستگی به شماره‌های فراخوانی یا معماری پردازنده (مثال: x86 در مقابل ARM) نوشته می‌شوند. تغییرات در هسته (مثال: به‌روزرسانی شماره فراخوانی) تنها نیاز به اصلاح کتابخانه دارد.

امنیت:توابع پوشاننده می‌توانند پیش از فراخوانی سیستمی، اعتبارسنجی آرگومان‌ها (مثال: بررسی محدوده حافظه `buf`) را انجام دهند تا از آسیب‌پذیری‌هایی مانند دسترسی به حافظه غیرمجاز جلوگیری شود.

ساده‌سازی توسعه: برنامه‌نویسان نیازی به یادگیری جزئیات سطح پایین (مثل اسمبلر یا ثبات‌ها) ندارند و می‌توانند از توابع استاندارد (مانند `write()` / `read()`) استفاده کنند.

پرسش 2: مقایسه‌ای بین دستور int و دستورات جدید مانند sysenter یا sysexit در دو سیستم عامل مختلف (برای مثال Linux) انجام دهید.

نحوهٔ پیاده‌سازی و عملکرد این دستورات:

دستور `int`: دستور `int` در پردازنده‌های x86 برای ایجاد وقفه نرم‌افزاری و انتقال از حالت کاربر به حالت هسته استفاده می‌شود. پس از اجرای این دستور، پردازنده به پردازش‌های مربوط به فراخوانی سیستمی در سطح هسته می‌پردازد.

دستور `sysenter` و `sysexit`: این دستورات در پردازنده‌های مدرن به منظور بهینه‌سازی عملکرد فراخوانی‌های سیستمی استفاده می‌شوند. `sysenter` برای ورود سریع‌تر به حالت هسته و `sysexit` برای بازگشت سریع‌تر از حالت هسته به حالت کاربر طراحی شده‌اند. این دستورات به پردازنده‌ها اجازه

می‌دهند که سریع‌تر از حالت کاربر به حالت هسته منتقل شوند و برعکس.

کارایی:

دستور `int`: به دلیل سربار زیاد در انتقال به حالت هسته (به‌ویژه ذخیره و بازیابی ثبات‌ها)، کارایی نسبتا کمتری دارد. این دستور در پردازنده‌های قدیمی‌تر استفاده می‌شود.

دستورات `sysenter` و `sysexit`: این دستورات در پردازنده‌های مدرن طراحی شده‌اند و کارایی بالاتری دارند. آن‌ها به طور خاص برای کاهش سربار و بهبود سرعت انتقال به حالت هسته و بازگشت به حالت کاربر طراحی شده‌اند.

کاربردهای مختلف:

دستور `int`: معمولاً در سیستم‌های قدیمی‌تر یا در مواردی که نیازی به بهینه‌سازی‌های جدیدتر نیست، از دستور `int` استفاده می‌شود.

دستورات `sysenter` و `sysexit`: این دستورات در سیستم‌های مدرن‌تر مانند Linux و در پردازنده‌های Intel و AMD استفاده می‌شوند تا به بهینه‌سازی عملکرد و کاهش سربار در فراخوانی‌های سیستمی کمک کنند.

مقایسه در xv6 و Linux:

در **xv6**، که یک نسخه ساده‌شده از سیستم‌عامل یونیکس است، معمولاً از دستور `int` برای ایجاد وقفه و فراخوانی سیستمی استفاده می‌شود. این سیستم‌عامل از پردازنده‌های x86 استفاده می‌کند.

در **Linux**، پردازنده‌های مدرن به‌طور معمول از دستورات `sysenter` و `sysexit` برای بهبود کارایی فراخوانی‌های سیستمی استفاده می‌کنند.

دستورات جدیدتر مانند `sysenter` و `sysexit` نسبت به `int` عملکرد بهتری دارند زیرا آن‌ها سربار کمتری ایجاد می‌کنند و سرعت فراخوانی‌های سیستمی را افزایش می‌دهند. `int` بیشتر در پردازنده‌های قدیمی و سیستم‌های ساده‌تر (مانند xv6) استفاده می‌شود.

ساز و کار اجرای فراخوانی سیستمی در xv6

پرسش 3: با توجه به توضیحات ارائه شده درباره ی xv6 و نحوه ی تنظیم سطح دسترسی برای فراخوانی‌های سیستمی، چرا تنها فراخوانی سیستمی (که با Trap Gate یا پیداسازی شده) با سطح دسترسی DPL_USER فعال می‌شود و سایر تله‌ها (مانند وقفه‌های سخت‌افزاری و استثناها) نمی‌توانند از این سطح دسترسی بهره برد؟

در xv6، تلاش برای فعال کردن یک تله با سطح دسترسی USER_DPL برای تله دیگری باعث بروز یک protection exception می‌شود. این تدابیر امنیتی برای جلوگیری از مشکلات ممکن در برنامه های کاربری یا اقدامات خبیث اعمال شده است. اجازه دادن به کاربران برای اجرای تله‌ها با سطوح دسترسی بالاتر می‌تواند یک خطر جدی امنیتی ایجاد کند زیرا این اقدام ممکن است دسترسی غیرمجاز به هسته را فراهم کند و به تخریب کلی امنیت سیستم منجر شود. سخت‌افزار توسط معماری x86 کنترل می‌شود و این سطوح دسترسی را اجرا می‌کند تا جدایی روشنی بین حالت کاربر و هسته حفظ شود و استثناء حفاظتی در صورت نقض این مرزها بوجود آید.

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

به طور کلی دو پشته داریم: stack kernel و stack user. هنگامی که یک trap فعال شود و می خواهیم دسترسی را تغییر دهیم مثلاً از سطح کاربر به سطح کرنل برویم نمی توانیم از پشته قبل استفاده کنیم. بنابراین باید esp و ss روی پشته push شوند تا هنگام بازگشت بدانیم که آخرین دستوری که انجام داده ایم چه بوده است و اطلاعات از دست نروند زیرا داشتن این اطلاعات ضروری است و بتوانیم ادامه روند اجرای دستورات را از سر بگیریم. ولی وقتی تغییر سطح دسترسی نداشته باشیم، نیازی به push آنها نیست زیرا همچنان با همان پشته کار میکنیم.

پرسش 5:با توجه به توضیحاتی که در مورد نحوه قرارگیری پارامترهای فراخوانی سیستمی بر روی پشته و نحوه دسترسی به آن‌ها از طریق توابعی مانند argptr داده شده است، توضیح دهید که این توابع چه نقشی در بازیابی پارامترهای فراخوانی سیستمی دارند. به خصوص، چرا تابع argptr باید بازه‌های آدرس ورودی را بررسی کند؟ در صورت عدم انجام این بررسی‌ها، چه نوع مشکلات امنیتی (مانند دسترسی به حافظه خارج از محدوده مجاز)

ممکن است ایجاد شود؟ به عنوان مثال، شرح دهید که چگونه نبود این بررسی‌ها در فراخوانی سیستمی read_sys، تواند باعث بروز اختلال یا آسیب‌پذیري در سیستم شود.

چهار تابع برای دسترسی به پارامترهای فراخوانی وجود دارند:

تابع argptr: در سیستم عامل xv6 برای بررسی صحت بازه آدرس های پارامترهای ارسالی به توابع فراخوانی سیستمی استفاده میشود. این تابع بررسی می‌کند که بازه آدرس ارائه شده در محدوده آدرس های قابل دسترس و معتبر در فضای آدرس کاربر است یا خیر. با انجام این بررسی، از وقوع آسیب پذیری های امنیتی و دسترسی غیرمجاز به مناطق حافظه جلوگیری می‌شود.

تابع argint: برای به دست آوردن یک عدد صحیح از فضای کاربری استفاده میشود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار آرگومان را از فضای کاربری به فضای کرنل منتقل میکند. اگر عملیات با موفقیت انجام شود، مقدار 0 را برمی‌گرداند؛ در غیر این صورت، 1- برمی‌گرداند.

تابع argstr: برای بازیابی یک رشته از فضای کاربری پردازه استفاده میشود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار رشته را از فضای کاربری به فضای کرنل کپی میکند. اگر عملیات با موفقیت انجام شود، مقدار 0 را برمی‌گرداند؛ در غیر این صورت -1 برمی‌گرداند.

تابع argfd: برای بازیابی فایل دسکریپتور از فضای کاربری پردازه استفاده میشود. این تابع آدرس مجازی آرگومان را محاسبه کرده، دسترسی به حافظه را بررسی کرده و مقدار فایل دسکریپتور را از فضای کاربری به فضای کرنل کپی می‌کند. در صورت موفقیت، مقدار 0 را برمی‌گرداند؛ در غیر این صورت -1 برمی‌گرداند.

همان که در توضیحات گفته شد تمامی این توابع بررسی میکنند که میزان حافظه داده شده حتماً در حافظه پردازه مورد نظر قرار گیرد. چون در غیر اینصورت ممکن است از حافظه پردازه دیگر استفاده کند که باعث ایجاد مشکلات زیادی در پردازنده میشود. تابع sys_read که فراخوانی سیستمی مربوط به تابع read است 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);
}
```

خود تابع read نیز به صورت زیر تعریف شده است:

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

تابع sys_read در if سه کار را انجام میدهد. به کمک تابع argfd مقدار fd را دریافت میکند. (آرگومان اول) به کمک تابع argint مقدار آرگومان سوم که ماکزیمم سایز ورودی است را دریافت میکند و به کمک argptr بررسی میکند که کل فضای آدرس دهی از ابتدا تا ماکزیمم در حافظه پردازه قرار میگیرد یا خیر. اگر این مورد بررسی نشود ممکن است مقدار ماکزیمم خیلی زیاد باشد و از یک فایل بزرگ در حال خواندن باشیم حافظه پردازه فعلی جوابگو نباشد و سیستم عامل باقی موارد را در در حافظه پردازه دیگری شروع به نوشتن میکند. طبیعتا این کار می تواند باعث بروز مشکلات زیادی اعم از ایجاد باگ یا مشکلات امنیتی بشود. البته در موارد هم مقداری ماکزیمم از حافظه بافر بیشتر است که در این موارد هم سرریز حافظه رخ میدهد که نوعی باگ است.

پرسش 6 مربوط به **تغییرات دستی در رجیسترها و ایجاد مشکلات در فراخوانی‌های سیستمی** است. این تغییرات می‌تواند موجب بروز مشکلاتی در هنگام فراخوانی‌های سیستمی شود.

پرسش 6: در صورتی که کاربر تغییراتی به صورت دستی در رجیسترها یا دستورهای بالا ایجاد کند، چه مشکلاتی به همراه خواهد داشت؟

1.مشکلات با وضعیت رجیسترها:

از دست رفتن اطلاعات مهم: اگر کاربر به طور دستی مقادیر رجیسترها را تغییر دهد، ممکن است داده‌های مهمی که برای انجام فراخوانی‌های سیستمی ضروری هستند، تغییر یابند. این می‌تواند منجر به خرابی در عملیات و عدم دقت در فراخوانی‌های سیستمی شود.

عدم هماهنگی با کد هسته: بسیاری از فراخوانی‌های سیستمی به مقادیر مشخصی از رجیسترها وابسته‌اند تا اطلاعات لازم را برای اجرای عملیات دریافت کنند. تغییر دستی این مقادیر می‌تواند باعث شود که کد هسته نتواند به درستی پردازش‌های خود را انجام دهد.

2.ایجاد خطاهای غیرمنتظره:

عدم تطابق پارامترها: بسیاری از فراخوانی‌های سیستمی نیاز دارند که پارامترها از طریق رجیسترها به هسته منتقل شوند. تغییر دستی این مقادیر می‌تواند باعث شود که مقادیر نادرستی به هسته ارسال شوند و این باعث بروز خطاهای غیرمنتظره در سیستم خواهد شد.

مشکلات در بازگشت از فراخوانی‌های سیستمی: اگر مقادیر رجیسترها به درستی تنظیم نشوند، ممکن است پردازنده نتواند به درستی به وضعیت قبلی خود بازگشته و فرآیند اجرای برنامه مختل شود.

3.برهم خوردن امنیت و پایداری سیستم:

حمله‌های احتمالی: تغییرات دستی در رجیسترها ممکن است به نادرستی برخی از محدودیت‌های امنیتی سیستم را دور بزند و به اجرای کدهای غیرمجاز یا مخرب منجر شود.

عدم پایداری سیستم: در صورتی که تغییرات به درستی پیاده‌سازی نشده یا به درستی در نظر گرفته نشوند، ممکن است پایداری سیستم به خطر بیفتد و فرآیندهای هسته به درستی اجرا نشوند.

4.تداخل در فرآیندهای همزمان:

مسائل هماهنگی پردازش‌ها: در صورتی که تغییرات دستی در رجیسترها باعث شود که مقادیر اشتباهی به فراخوانی‌های سیستمی ارسال شوند، این می‌تواند باعث شود که پردازش‌های مختلف به اشتباه با هم تداخل کنند و مشکلاتی در زمان‌بندی یا مدیریت منابع ایجاد شود.

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

یک برنامه سطح کاربر به نام IDProc.c نوشتیم که ID پردازنده فعلی را به ما بدهد.

IDProc.c

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

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

هم xv6 و هم gdb را در حالت کرنل اجرا می کنیم و continue می کنیم تا برنامه پیش برود و بتوانیم برنامه سطح کاربر را اجرا نماییم سپس ctrl+c می زنیم و در خط 138 syscall.c یک breakpoint قرار داده و

سپس دوباره continue می کنیم حال در qemu باز شده IDProc را اجرا می کنیم و ملاحظه میکنیم که breakpoint قرار داده شده hit شده است تصاویر به شرح زیر است:

```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000fff0 in ?? ()
(gdb) c
Continuing.
^C
Thread 2 received signal SIGINT, Interrupt.
[Switching to Thread 1.2]
0x000fd0a9 in ?? ()
(gdb) b syscall.c:138
Breakpoint 1 at 0x80104aa4: file syscall.c, line 138.
(gdb) c
Continuing.

[Switching to Thread 1.1]

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb)
Continuing.
```

حال با فشردن همزمان ctrl+x+a به حالت TUI در gdb میرویم و دستور bt را همانطور که خواسته شده وارد میکنیم.

دستور bt چیست؟ این دستور مخفف backtrace است و سلسله توابعی که فراخوانی شده اند و به استک اضافه شده اند را نشان میدهد. به طور کلی میتوان گفت این دستور نشان میدهد که برنامه چطور به جایی که اکنون در آن قرار دارد، رسیده است. برای آنکه بهتر متوجه خروجی این دستور شویم، مراحل تعریف و اجرای یک فراخوانی سیستمی را مرور می کنیم:

1. در فایل syscall.h یک عدد برای فراخوانی سیستمی مورد نظر انتخاب شده است.

2. شناسه فراخوانی سیستمی مورد نظر در فایل user.h نوشته شده است.

3. در فایل usys.S تعریف فراخوانی سیستمی به اسمبلی انجام میشود.

4. vector64 در فایل vectors.S تعریف شده است. که میتوان با اجرای دستور int64 در مرحله قبل، وارد این بخش شد. بعد از push شدن مقدار 64، به بخش alltraps در فایل trapasm.S خواهیم رفت.

5. بخش alltraps ابتدا trap frame مربوطه را خواهد ساخت و آن را در استک push خواهد کرد. سپس تابع trap در فایل trap.c را فراخوانی خواهد کرد.

6.تابع trap بعد از آنکه می‌فهمد فراخوانی مربوط به یک system call است، این trap frame ای که در استک پوش شده است را به عنوان trap frame پردازه فعلی قرار خواهد داد و تابع syscall را فرا می‌خواند.

7. تابع syscall در فایل syscall.c قرار دارد. این تابع پس از خواندن شماره فراخوانی سیستمی که در فیلد eax در trap frame پردازه فعلی قرار دارد، تابع مربوط به آن را فرا می‌خواند و خروجی این تابع را در فیلد eax در trap frame پردازه فعلی ذخیره میکند.

```
syscall.c
      133 {
      134   int num;
      135   struct proc *curproc = myproc();
      136
      137   num = curproc->tf->eax;
B+>   138   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
      139     curproc->tf->eax = syscalls[num]();
      140   } else {
      141     cprintf("%d %s: unknown sys call %d\n",
      142             curproc->pid, curproc->name, num);
      143     curproc->tf->eax = -1;
      144   }
      145 }
```

```
remote Thread 1.1 (src) In: syscall                    L138  PC: 0x80104aa4
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x80105a9d in trap (tf=0x8dffefb4) at trap.c:43
#2  0x8010584f in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

همانطور که در تصویر فوق مشاهده میشود، خروجی دستور bt مراحل 5 تا 7 توضیح بالا و اجرای یک فراخوانی سیستمی را نمایش میدهد.

در صورت استفاده از دستور down به این خطا برمیخوریم که علتش این است که در داخلی ترین frame قرار داریم:

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

حال از دستور up استفاده میکنیم و به یک frame عقب تر میرویم:

```
trap.c
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){
52       acquire(&tickslock);
53       ticks++;
54       wakeup(&ticks);
55       release(&tickslock);
56     }
57     lapiceoi();
58     break;
remote Thread 1.1 (src) In: trap                    L43   PC: 0x80105a9d
#2  0x8010584f 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  0x80105a9d in trap (tf=0x8dffefb4) at trap.c:43
(gdb)
```

میدانیم که شماره فراخوانی سیستمی getpid برابر با ۱۱ است. حال اگر محتوای ثبات eax را بخوانیم،

میبینیم که مقدار آن ۵ است و با شماره فراخوانی سیستمی مورد نظر ما تفاوت دارد:

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

علت این اتفاق این است که قبل از رسیدن به فراخوانی سیستمی getpid، فراخوانی های سیستمی دیگری هم رخ میدهند. در صورتی که برنامه را چندین مرحله continue کرده و هر بار محتوای ثبات eax را چک کنیم، این فراخوانی ها را میبینیم.

فراخوانی سیستمی شماره 5 (read): این فراخوانی سیستمی برای خواندن دستور تایپ شده است و آن را کاراکتر به کاراکتر می‌خواند.

فراخوانی سیستمی شماره 1 (fork): این فراخوانی سیستمی برای ایجاد پردازه جدید جهت اجرای برنامه سطح کاربر اجرا می‌شود.

فراخوانی سیستمی شماره 12 (sbrk): این فراخوانی سیستمی جهت تخصیص حافظه به پردازه ایجاد شده اجرا می‌شود.

فراخوانی سیستمی شماره 7 (exec): این فراخوانی سیستمی برای اجرای برنامه سطح کاربر در پردازه ایجاد شده اجرا می‌شود.

فراخوانی سیستمی شماره 3 (wait): این فراخوانی سیستمی در پردازه پدر اجرا می‌شود و تا اتمام اجرای پردازه فرزند صبر می‌کند.

فراخوانی سیستمی شماره 11 (getpid): این فراخوانی سیستمی مربوط به برنامه سطح کاربر است.

فراخوانی سیستمی شماره 16 (write): این فراخوانی سیستمی خروجی برنامه سطح کاربر را کاراکتر به کاراکتر می‌نویسد.

شماره تمامی فراخوانی های سیستمی در فایل syscall.h موجود است.

در ادامه روند اجرا شدن فراخوانی های سیستمی تا چاپ شدن خروجی برنامه سطح کاربر قابل مشاهده است:

چون اسم برنامه سطح کاربر (IDProc) از 6 کاراکتر تشکیل شده پس انتظار داریم 6 بار فراخوانی (read = 5) رخ دهد ولی چون یک کاراکتر هم زمان رسیدن به breakpoint خوانده شده پس 5 بار این فراخوانی رخ خواهد داد:

```
#0  syscall () at syscall.c:138
(gdb) p num
$2 = 5
(gdb)
```

```
$6 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$7 = 16
(gdb)
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$8 = 1
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$9 = 3
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$10 = 12
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$11 = 7
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$12 = 11
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) p num
$13 = 16
```

و در ادامه فراخوانی سیستمی 16 اجرا خواهد شد که همان write خواهد بود و در ادامه خروجی ما خواهد بود:

```
QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 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
init: starting sh
Alireza Kamkar
Parham Badv
Ali Elm talab
$ IDProc
Process ID is: 3
$
```

ارسال آرگومان های فراخوانی های سیستمی:

اضافه کردن شماره فراخوانی سیستمی در فایل syscall.h:

```
#define SYS_simple_arithmetic 22
```

اضافه کردن پروتوتایپ تابع برای فضای کاربر در فایل user.h:

```
int simple_arithmetic(int, int);
```

اضافه کردن تابع پوشاننده در کتابخانه کاربر در فایل usys.S:

```
SYSCALL(simple_arithmetic)
```

ثبت فراخوانی سیستمی در جدول syscalls در فایل syscall.c:

```
[SYS_simple_arithmetic] sys_simple_arithmetic,
```

```
extern int sys_simple_arithmetic(void);
```

پیاده‌سازی تابع sys_simple_arithmetic (خواندن از ثبات‌ها) در فایل sysproc.c:

```
int
sys_simple_arithmetic(void)
{
  int a, b;

  struct proc *curproc = myproc();

  a = curproc->tf->edi;
  b = curproc->tf->esi;

  int result = (a + b) * (a - b);
  cprintf("%d\n",result);
  cprintf("simple_arithmetic: a=%d, b=%d, result=%d\n", a, b, result);

  return result;
}
```

ایجاد یک برنامه تستی در فایل simple_arithmetic_test.c و اضافه کردن در MAKEFILE:

simple_arithmetic_test.c

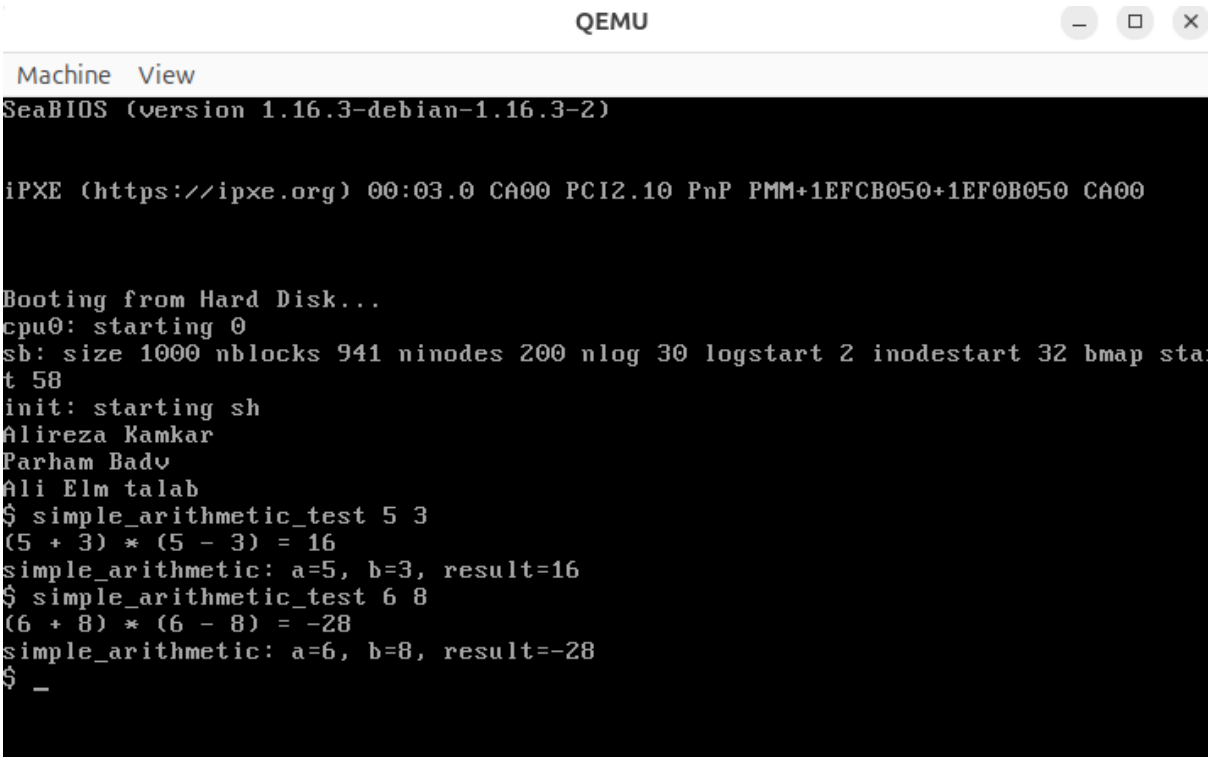
```c
#include "types.h"
#include "user.h"
#include "stat.h"

int call_simple_arithmetic(int a, int b) {
  int result;
  asm volatile(
    "movl %1, %%edi\n\t"
    "movl %2, %%esi\n\t"
    "movl $22, %%eax\n\t"
    "int $64\n\t"
    "movl %%eax, %0"
    : "=r" (result)
    : "r" (a), "r" (b)
    : "%eax", "%edi", "%esi"
  );
  return result;
}

int main(int argc, char *argv[]) {
  if (argc != 3) {
    printf(2, "Usage: %s <a> <b>\n", argv[0]);
    exit();
  }

  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  printf(1, "(%d + %d) * (%d - %d) = ", a, b, a, b);
  int result = call_simple_arithmetic(a, b);
  exit();
}
```

خروجی:



پیاده سازی فراخوانی های سیستمی

1.پیاده سازی فراخوانی سیستمی ایجاد نسخه کپی فایل:

اضافه کردن شماره فراخوانی سیستمی در فایل syscall.h:

```
#define SYS_make_duplicate 23
```

ثبت در جدول syscalls در فایل syscall.c:

```c
extern int sys_uptime(void);
extern int sys_simple_arithmetic(void);
extern int sys_make_duplicate(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_simple_arithmetic] sys_simple_arithmetic,
[SYS_make_duplicate] sys_make_duplicate,
};
```

در فایل sysfile.c تابع زیر را اضافه میکنیم:

```c
int
sys_make_duplicate(void)
{
  char *src_path;
  char dest_path[512];
  struct inode *ip_src, *ip_dest;
  int n;
  uint off;

  if(argstr(0, &src_path) < 0)
    return -1;

  if((ip_src = namei(src_path)) == 0) {
    cprintf("make_duplicate: source file '%s' not found\n", src_path);
    return -1;
  }

  ilock(ip_src);

  if(ip_src->type != T_FILE) {
    iunlockput(ip_src);
    cprintf("make_duplicate: source is not a file\n");
    return 1;
  }

  safestrcpy(dest_path, src_path, sizeof(dest_path));
  safestrcpy(dest_path + strlen(dest_path), ".copy",
             sizeof(dest_path) - strlen(dest_path));

  cprintf("make_duplicate: copying %s to %s\n", src_path, dest_path);

  begin_op();
  ip_dest = create(dest_path, T_FILE, 0, 0);
  if(ip_dest == 0) {
    end_op();
    iunlockput(ip_src);
    cprintf("make_duplicate: failed to create destination file\n");
    return 1;
  }

```

```
486    ilock(ip_dest);
487
488    char buf[512];
489    off = 0;
490    for(;;) {
491      n = readi(ip_src, buf, off, sizeof(buf));
492      if(n <= 0)
493        break;
494
495      if(writei(ip_dest, buf, off, n) != n) {
496        cprintf("make_duplicate: write error\n");
497        iunlockput(ip_dest);
498        iunlockput(ip_src);
499        end_op();
500        return 1;
501      }
502      off += n;
503    }
504
505    iunlockput(ip_src);
506    iunlockput(ip_dest);
507    end_op();
508
509    cprintf("make_duplicate: copy completed successfully\n");
510    return 0;
511  }
```

مرحله ۱: اعتبارسنجی ورودی‌ها

ابتدا مسیر فایل مبدأ از آرگومان‌های فراخوانی خوانده می‌شود

وجود فایل مبدأ در سیستم فایل بررسی می‌گردد

در صورت عدم وجود فایل مبدأ، خطای مناسب بازگردانده می‌شود

مرحله ۲: بررسی مجوزها و نوع فایل

قفل inode فایل مبدأ برای دسترسی انحصاری گرفته می‌شود

بررسی می‌شود که منبع واقعاً یک فایل معمولی باشد و نه دایرکتوری یا دستگاه

مرحله ۳: ایجاد فایل مقصد

مسیر فایل مقصد با افزودن پسوند .copy به نام فایل مبدأ ساخته می‌شود

یک تراکنش فایل‌سیستم جدید آغاز می‌شود

فایل مقصد با همان مشخصات فایل مبدأ ایجاد می‌گردد

مرحله ۴: کپی کردن محتوا

داده‌ها از فایل مبدأ به صورت بلوک‌های ۵۱۲ بایتی خوانده می‌شوند

هر بلوک خوانده شده بلافاصله در فایل مقصد نوشته می‌شود

این فرآیند تا انتهای فایل مبدأ ادامه می‌یابد

مرحله ۵: پایانی‌سازی

قفل‌های inode آزاد می‌شوند

تراکنش فایل‌سیستم تکمیل می‌گردد

پیام موفقیت ثبت شده و نتیجه مناسب بازگردانده می‌شود

در صورت عدم وجود فایل مبدأ، مقدار ۱- بازگردانده می‌شود

در صورت شکست در ایجاد فایل مقصد یا نوشتن داده، مقدار ۱ بازگشت داده می‌شود

تنها در صورت موفقیت کامل، مقدار ۰ بازمی‌گردد

از توابع ایمن برای کپی رشته‌ها استفاده می‌شود تا از overflow جلوگیری شود

مکانیزم قفل‌گذاری از دسترسی همزمان جلوگیری می‌کند

اضافه کردن تابع در فایل user.h:

```
int make_duplicate(const char*);
```

در فایل usys.S:

```
SYSCALL(make_duplicate)
```

نوشتن برنامه تست duplicate_test.c:

duplicate_test.c

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

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf(2, "Usage: %s <filename>\n", argv[0]);
    exit();
  }

  int result = make_duplicate(argv[1]);

  if (result == 0) {
    printf(1, "File duplicated successfully: %s.copy created\n", argv[1]);
  } else if (result == -1) {
    printf(2, "Error: Source file '%s' not found\n", argv[1]);
  } else {
    printf(2, "Error: Duplication failed\n");
  }

  exit();
}
```

اجرا:

QEMU

Machine View

```
SeaBIOS (version 1.16.3-debian-1.16.3-2)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Alireza Kamkar
Farham Badv
Ali Elm talab
$ echo "salam" > a.txt
$ duplicate_test a.txt
make_duplicate: copying a.txt to a.txt.copy
cat a.txt.copy
salam
```

2.پیاده سازی فراخوانی سیستمی بدست آوردن اعضای خانواده یک پردازه

عملیات های مربوط به اضافه کردن سیستم کال را مانند شماره 1 انجام می دهیم.

در فایل proc.c تابع زیر را اضافه کنید:

```c
int
sys_show_process_family(void)
{
  int pid;
  struct proc *p;
  struct proc *current_proc;
  struct proc *parent;
  int found = 0;

  if(argint(0, &pid) < 0)
    return -1;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid && p->state != UNUSED) {
      current_proc = p;
      found = 1;
      break;
    }
  }

  if(!found) {
    release(&ptable.lock);
    cprintf("show_process_family: process with PID %d not found\n", pid);
    return -1;
  }

  cprintf("My id: %d, My parent id: %d\n",
          current_proc->pid,
          current_proc->parent ? current_proc->parent->pid : -1);

  cprintf("Children of process %d:\n", pid);
  int has_children = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->parent == current_proc && p->state != UNUSED) {
      cprintf("Child pid: %d\n", p->pid);
      has_children = 1;
    }
```

```c
  }
  if(!has_children) {
    cprintf("No children\n");
  }

  cprintf("Siblings of process %d:\n", pid);
  int has_siblings = 0;
  parent = current_proc->parent;
  if(parent) {
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if(p->parent == parent && p != current_proc && p->state != UNUSED) {
        cprintf("Sibling pid: %d\n", p->pid);
        has_siblings = 1;
      }
    }
  }
  if(!has_siblings) {
    cprintf("No siblings\n");
  }

  release(&ptable.lock);
  return 0;
}
```

شناسه پردازه (PID) از پارامترهای ورودی خوانده می‌شود - صحت و اعتبار PID ورودی بررسی می‌گردد

با قفل کردن جدول پردازه‌ها، جستجو برای یافتن پردازه با PID مشخص شده آغاز می‌شود - تنها پردازه‌های فعال (با وضعیت غیر از UNUSED) در نظر گرفته می‌شوند - در صورت عدم یافتن پردازه مورد نظر، پیام خطای مناسب نمایش داده می‌شود

سپس شناسه پردازه جاری و شناسه پردازه والد آن نمایش داده می‌شود - در صورت عدم وجود والد (مانند پردازه init)، مقدار -1 نمایش می‌یابد

در ادامه تمام پردازه‌های موجود در سیستم بررسی می‌شوند - پردازه‌هایی که parent آنها به پردازه جاری باشد، به عنوان فرزندان شناسایی می‌شوند - لیست کامل شناسه‌های فرزندان نمایش داده می‌شود - در صورت عدم وجود فرزند، پیام "No children" نمایش می‌یابد

سپس پردازه‌هایی که parent مشترک با پردازه جاری دارند، به عنوان برادر شناسایی می‌شوند - پردازه جاری از لیست برادران حذف می‌شود تا خودش نمایش داده نشود - لیست کامل شناسه‌های برادران نمایش داده می‌شود - در صورت عدم وجود برادر، پیام "No siblings" نمایش می‌یابد

سپس قفل جدول پردازه‌ها آزاد می‌شود و کد موفقیت (۰) بازگردانده می‌شود.

نوشتن برنامه تست family_test.c:

family_test.c

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

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf(2, "Usage: %s <pid>\n", argv[0]);
    exit();
  }

  int pid = atoi(argv[1]);
  int result = show_process_family(pid);

  if (result == -1) {
    printf(2, "Error: Process with PID %d not found\n", pid);
  }

  exit();
}
```

اجرا:

QEMU

Machine View

```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00


Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Alireza Kamkar
Parham Badv
Ali Elm talab
$ family_test 1
My id: 1, My parent id: -1
Children of process 1:
Child pid: 2
Siblings of process 1:
No siblings
$ family_test 999
show_process_family: process with PID 999 not found
Error: Process with PID 999 not found
$
```

3.پیاده سازی فراخوانی سیستمی جستجوی محتوا در فایل

عملیات های مربوط به اضافه کردن سیستم کال را مانند شماره 1 انجام می دهیم.

پیاده سازی تابع در sysfile.c:

```c
int
sys_grep_search(void)
{
  char *keyword, *filename, *user_buffer;
  int buffer_size;
  struct inode *ip;
  char *kernel_buffer;
  char *line_start, *line_end;
  int found = 0;
  int result = -1;

  if(argstr(0, &keyword) < 0 || argstr(1, &filename) < 0 ||
     argstr(2, &user_buffer) < 0 || argint(3, &buffer_size) < 0)
    return -1;

  if(buffer_size <= 0)
    return -1;

  if((ip = namei(filename)) == 0) {
    cprintf("grep_syscall: file '%s' not found\n", filename);
    return -1;
  }

  ilock(ip);

  if(ip->type != T_FILE) {
    iunlockput(ip);
    cprintf("grep_syscall: '%s' is not a file\n", filename);
    return -1;
  }

  kernel_buffer = kalloc();
  if(kernel_buffer == 0) {
    iunlockput(ip);
    cprintf("grep_syscall: memory allocation failed\n");
    return -1;
  }

  readi(ip, kernel_buffer, 0, ip->size);

  line_start = kernel_buffer;
```

```c
  for(char *p = kernel_buffer; p < kernel_buffer + ip->size; p++) {
    if(*p == '\n' || p == kernel_buffer + ip->size - 1) {
      line_end = p;

      char *temp = line_start;
      while(temp <= line_end) {
        char *key_temp = keyword;
        char *search_temp = temp;

        while(*key_temp && search_temp <= line_end && *key_temp == *search_temp) {
          key_temp++;
          search_temp++;
        }

        if(*key_temp == '\0') {
          found = 1;
          break;
        }

        temp++;
      }

      if(found) {
        int line_length = line_end - line_start + 1;
        if(line_length > buffer_size - 1)
          line_length = buffer_size - 1;

        if(copyout(myproc()->pgdir, (uint)user_buffer, line_start, line_length) < 0) {
          result = -1;
        } else {
          if(line_length < buffer_size)
              copyout(myproc()->pgdir, (uint)user_buffer + line_length, "\0", 1);
          result = line_length;
        }
        break;
      }

      line_start = p + 1;
    }
  }
```

```c

  kfree(kernel_buffer);
  iunlockput(ip);

  if(!found) {
    cprintf("grep_syscall: keyword '%s' not found in file '%s'\n", keyword, filename);
    return -1;
  }

  return result;
}
```

فرآیند جستجو برای یافتن یک کلمه خاص در یک فایل متنی، با دریافت چهار پارامتر ورودی شروع می‌شود: کلیدواژه، نام فایل، بافر خروجی و اندازه بافر. در ابتدا، سیستم ورودی‌ها را بررسی می‌کند تا مطمئن شود که اندازه بافر معتبر است و مقادیر ورودی درست وارد شده‌اند. سپس، فایل مورد نظر جستجو می‌شود. اگر فایل وجود نداشته باشد یا نوع آن مناسب نباشد، سیستم خطای مناسب را بازمی‌گرداند.

در ادامه، یک بافر در حافظه برای ذخیره محتوای فایل ایجاد می‌شود و تمام داده‌های فایل به این حافظه موقت منتقل می‌گردد. سپس، محتویات فایل به خطوط مختلف تقسیم می‌شود و هر خط به صورت

جداگانه بررسی می‌شود تا ببینیم آیا شامل کلیدواژه مورد نظر است یا خیر. جستجو به صورت دقیق و حرف به حرف انجام می‌شود و زمانی که اولین خط حاوی کلیدواژه پیدا شد، جستجو متوقف می‌شود.

خط پیدا شده به بافر مشخص شده توسط کاربر کپی می‌شود. در صورتی که طول این خط بیشتر از اندازه بافر باشد، آن را کوتاه کرده و یک علامت پایان‌دهنده به انتهای آن اضافه می‌شود تا مطابق با اندازه بافر باشد.

در مرحله آخر، حافظه موقت آزاد می‌شود و فایل سیستم قفل‌ها را رها می‌کند. این فرآیند با رعایت مسائل امنیتی و مدیریت صحیح خطاها انجام می‌شود. به این معنی که ابتدا بررسی می‌شود که فایل وجود دارد، سپس نوع فایل تأیید می‌شود تا از دسترسی به منابع غیرمجاز جلوگیری شود. همچنین، حافظه به درستی مدیریت می‌شود و از سرریز بافر جلوگیری می‌شود. برای دسترسی امن به فایل‌ها از مکانیزم قفل‌گذاری استفاده می‌شود.

در نهایت، اگر خطی با کلیدواژه پیدا شود، طول آن به بایت به عنوان نتیجه باز می‌گردد و اگر مشکلی وجود داشته باشد، عدد (۱-) به عنوان خطا باز می‌گردد.

نوشتن برنامه تست grep_test.c:

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

int main(int argc, char *argv[]) {
  char buffer[512];
  int result;

  if (argc != 3) {
    printf(2, "Usage: %s <filename> <keyword>\n", argv[0]);
    exit();
  }

  result = grep_search(argv[2], argv[1], buffer, sizeof(buffer));

  if (result > 0) {
    printf(1, "Found line (%d bytes): %s\n", result, buffer);
  } else {
    printf(2, "Error: Keyword '%s' not found in file '%s'\n", argv[2], argv[1]);
  }

  exit();
}
```

اجرا:

QEMU

Machine View

```
SeaBIOS (version 1.16.3-debian-1.16.3-2)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 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
init: starting sh
Alireza Kamkar
Parham Badv
Ali Elm talab
$ grep_test README xv6
Found line (71 bytes): NOTE: we have stopped maintaining the x86 version of xv6
 and switched

$ grep_test README "nonexistentword"
grep_syscall: keyword '"nonexistentword"' not found in file 'README'
Error: Keyword '"nonexistentword"' not found in file 'README'
$
```