به نام خدا

آزمایشگاه سیستم عامل

پروژه دوم آزمایشگاه سیستم عامل

(فراخوانی سیستمی)

گروه ۶

اعضای گروه :

محمد سعادتی	**۸۱۰۱۹۸۴۱۰**
محمد عراقی	**۸۱۰۱۹۸۴۳۶**
سید عماد امامی	**۸۱۰۱۹۸۵۲۷**

مقدمه

۱) کتابخانه های (قاعدتاً سطح کاربر، منظور فایل های تشکیل دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانیهای سیستمی و علت این استفاده بررسی نمایید.

متغیر ULIB شامل چند فایل تشکیل دهنده به شرح زیر می باشد:

printf: برخی از توابع این کتابخانه به عنوان توابع کمکی برای چاپ در حالت های مختلف نوشته شده اند. در این بخش تنها تابع putc از فراخوانی سیستمی استفاده می کند.

در تابع putc از فراخوانی سیستمی write برای چاپ کردن استفاده می شود که برای آن fd و کاراکتر مورد نظر برای چاپ کردن انتخاب شده است.

ulib: برخی از توابع این کتابخانه برای کار با آرایه ی کاراکتر ها نوشته شده اند. فراخوانی های سیستمی برای این توابع وجود ندارد. ولی شامل سه تابع به نام stat، memset، gets می باشد که در آنها از فراخوانی سیستمی استفاده شده است.

gets تابعی است که با استفاده از read از ورودی می خواند. در تابع gets با اجرا یک حلقه ورودی گرفته می شود. در هر مرحله از ورودی گرفتن از فراخوانی سیستمی read استفاده می شود.

memset برای پر کردن حافظه با مقادیر دلخواه به کار برده می شود.

در تابع stat ابتدا از فراخوانی سیستمی open استفاده می شود که برای باز کردن یک فایل از این فراخوانی سیستمی استفاده می شود. در بخش بعد به کمک فراخوانی سیستمی fstat اطلاعات فایل مرتبط با file descriptor درخواستی داده می شود. در انتها به کمک فراخوانی سیستمی close فایل بسته می شود.

umalloc: برخی از توابع این کتابخانه برای اختصاص دادن حافظه استفاده شده اند. در این کتابخانه تابع morecore از فراخوانی سیستمی استفاده می کند.

تابع morecore برای افزایش حافظه استفاده می شود. در آن از فراخوانی سیستمی sbrk استفاده می شود. این فراخوانی سیستمی اندازه data segment را تغییر می دهد.

۲) دقت شود فراخوانی های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روشها را در لینوکس به اختصار توضیح دهید. میتوانید از مرجع [۳] کمک بگیرید.

Exceptions: در Exceptions ها دسترسی به kernel انجام می شود تا خطا رفع شود ، سپس دوباره به سطح کاربر بازگردانده می شویم.

Pseudo-file systems: در pseudo-file system ها مثل /dev، /sys، /proc دسترسی به کرنل نیاز است. این pseudo-file system ها مثل filesystem ها file به معنای فایلی که زمان ساخت، حجم و ... دارد، ندارند و شامل یک سری entry مجازی می باشند که می توانند محتوای داده ساختار های درون کرنل را طوری که انگار آن محتوا روی یک فایل ذخیره شده بودند، به یک اپلیکیشن یا ادمین تحویل دهند.

Socket based: در Socket based ها امکان گوش کردن بر روی socket و دریافت اطلاعات برای برنامه های سطح کاربر فراهم می شود.

سازوکار اجرای فراخوانی سیستمی در xv6

بخش سخت افزاری و اسمبلی

۳) آیا باقی تله ها را نمی توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟

خیر. xv6 اجازه فعال کردن یک interrupt دیگر را به یک پردازه نمی دهد و یک protection exception رخ می دهد و به vector شماره ۱۳ هدایت می شویم زیرا ممکن است در برنامه سطح کاربر باگی وجود داشته باشد، یا کاربر سوءاستفاده کند و امنیت سیستم به خطر بیفتد. یعنی سطح دسترسی DPL_USER سطح کاربر است و اگر بخواهیم باقی تله ها هم با همین سطح دسترسی فعال کنیم به راحتی می توان به kernel دسترسی داشت که در این صورت امنیت سیستم به خطر میفتد.

۴) در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟

در کل دو پشته یکی برای kernel و دیگری برای سطح کاربر وجود دارد. هنگامی که بخواهیم دسترسی را تغییر بدهیم، مثلا از kernel به سطح کاربر برویم، دیگر نمی توان از پشته قبلی استفاده کرد. پس باید ss و esp روی پشته push بشوند تا بتوان دوباره پس از بازگشت از سطح دسترسی دیگر از آنها استفاده کرد و اطلاعات از بین نروند. همچنین وقتی تغییر سطح دسترسی نداشته باشیم، نیازی به push کردن ss و esp نیست چون همچنان به همان پشته قبلی دسترسی داریم.

بخش سطح بالا و کنترلکننده زبان سی تله

۵) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr() بازه آدرسها بررسی میگردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی بازهها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی sys_read() اجرای سیستم را با مشکل روبرو سازد.

در این بخش سه تابع به نام های argptr،argstr ، argint وجود دارند که عملکرد هر کدام به شرح زیر است:

argint : برای گرفتن متغیر آدرس یک int را به آن می دهیم و شماره آرگومان آن را مشخص می کنیم.

argptr: آدرس یک پوینتر ، اندازه چیزی که خوانده می شود و شماره آرگومان را میگیرد و در آدرس پوینتر محتوای آرگومان را می ریزد.

argstr: دو پارامتر می گیرد اولی شماره آرگومان و دومی آدرس یک متغیر از نوع char* که در آن آرگومان ریخته می شود.

در صورتی که آدرس را چک نکنیم ممکن است خارج از محدوده مجاز باشد و به اطلاعات اشتباه دسترسی پیدا کنیم و مشکل در پردازش رخ بدهد.

۶) چگونه هنگام بازگشت به سطح کاربر، اجرا از همان خطی که متوقف شده بود، دوباره از سر گرفته میشود؟ فرایند را توضیح دهید.

در حین انتقال از user mode به kernel mode، پردازنده بین پشته per-process-user-stack و per-process-kernel-stack سوئیچ می کند. سپس selector بخش پشته(stack) user-per-process و stack pointer در kernel stack ذخیره می شود و سپس pointer دستورالعمل eip (آدرس برگشت در user mode) و سایر ثبات های سخت افزاری به استک kernel پوش(push) می شوند.

هنگامی که هسته باید به حالت کاربر(user mode) برگردد، کد trapret تمام مقادیر ذخیره شده در پشته هسته (kernel stack) را به رجیسترهای سخت افزاری بازمی گرداند. اما وقتی iret ، eip را از استک کرنل پاپ می کند، دستور بعدی که باید اجرا شود آدرس بازگشت در سطح کاربر (user mode) است. این فرآیند بدون پاپ کردن کامل سایر مقادیر استک کرنل اتفاق می افتد.

ارسال آرگومان های فراخوانی های سیستمی

سیستم کال calculate_sum_of_digits(int n):

برای اضافه کردن این سیستم کال به هسته، ابتدا باید شماره ی اختصاص داده شده به سیستم کال و امضای تابع را اضافه کنیم. برای اینکار در usys.S، defs.h ، syscall.c syscall.h،user.h تغییرات لازم را اعمال میکنیم. سپس برای بدنه ی سیستم کال باید در sysproc.c تابعی مانند بقیه ی توابع سیستم کال های نوشته شده بنویسیم طوری که ثبات ebx که ثبات پس از eax است را می خواند و تابع calculate_sum_of_digits را که در proc.c نوشته شده است با مقدار این ثبات صدا می زند.

```
int
sys_calculate_sum_of_digits(void) {
  int number = myproc()->tf->ebx;
  cprintf("KERNEL = SYS calculate sum of digits. Call for number %d\n", number);
  return calculate_sum_of_digits(number);
}
```

در proc.c تابع پیدا کردن مجموع ارقام ورودی به شرح زیر نوشته شده است:

```c
int
calculate_sum_of_digits(int number)
{
  int result = 0;
  while(number) {
    result += number % 10;
    number /= 10;
  }
  return result;
}
```

به منظور تست سیستم کال برای برنامه سطح کاربر قطعه کد sum_of_digits.c نوشته شده است که در آن پس از چک کردن تعداد ورودی ها، ثبات قبلی را ذخیره می کنیم سپس مقدار جدید را در ثبات قدیمی می نویسیم و تابع را صدا می زنیم و در نهایت مقدار اولیه ی ثبات را به آن برمی گردانیم.

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

int main(int argc, char *argv[]){
        if(argc < 2)
        {
                printf(2, "You must enter exactly one number!\n");
                exit();
        }
    else
    {
                int saved_ebx, number = atoi(argv[1]);
                asm volatile(
                        "movl %%ebx, %0;"
                        "movl %1, %%ebx;"
                        : "=r" (saved_ebx)
                        : "r"(number)
                );
                printf(1, "User: calculate_sum_of_digits() called for number: %d\n" , number);
                printf(1, "Sum Of Digits %d = %d\n" , number , calculate_sum_of_digits());
                asm("movl %0, %%ebx" : : "r"(saved_ebx));
                exit();
    }
    exit();
}
```

در پایان برای آن که این برنامه به درستی کامپایل شود باید این برنامه را به Makefile اضافه کنیم.

```
$ sum_of_digits 985
User: calculate_sum_of_digits() called for number: 985
KERNEL = SYS calculate sum of digits. Call for number 985
Sum Of Digits 985 = 22
```

پیاده سازی فراخوانی های سیستمی

۱. پیاده سازی فراخوانی سیستمی پیدا کردن آدرس سکتورهای فایل

سیستم کال void get_file_sectors(int fd, uint* addresses)

اطلاعات فراداده یک فایل (struct file) درون ساختار inode نگه‌داری می‌شود. بخشی ازین اطلاعات مربوط به آدرس سکتورهای مشخص شده برای فایل است که درون آرایه‌ای از اعداد مثبت (uint) نگه‌داری می‌شود.

با گرفتن یک fd ما ساختار فایل مربوط به آن را از طریق دستور argfd بدست می‌آوریم. این دستور در بین فایل‌های پردازه فعلی (myproc()->ofile) به دنبال فایلی با fd ذکر شده میگردد و در صورت یافتن آن را از طریق آرگومان پاس داده شده خروجی می‌دهد.

sysfile.c:

```
int
sys_get_file_sectors(void) {
  int fd;
  struct file *f;
  uint* addrs_starts;

  if(argfd(0, &fd, &f) < 0 || argptr(1, (void*)&addrs_starts, sizeof(*addrs_starts)) < 0)
    return -1;

  for(int i = 0; i < NDIRECT+1; i++)
    addrs_starts[i] = (int)f->ip->addrs[i];
  return 5;
}
```

حال اطلاعات inode یک فایل در ویژگی ip آن ذخیره شده و آدرس‌های سکتورها نیز در ویژگی addrs[i] از ip ذخیره شده است. پس ما آرگومان پاس داده شده (addrs_starts) را با addrsهای f->ip پر می‌کنیم.

```
$ sector_tester
KERNEL = SYS get file sectors. Call for fd 3
sector_file.txt:  65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 0, 0,
KERNEL = SYS get file sectors. Call for fd 4
sector_file1.txt:  76, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
KERNEL = SYS get file sectors. Call for fd 5
sector_file2.txt:  878, 879, 880, 881, 882, 883, 884, 0, 0, 0, 0, 0, 0,
KERNEL = SYS get file sectors. Call for fd 6
sector_file3.txt:  885, 886, 887, 888, 889, 890, 891, 0, 0, 0, 0, 0, 0,
$
```

۷) توضیح دهید عدم توالی داده های یک فایل روی دیسک، چگونه ممکن است در فرایند بازیابی آن پس از حذف مشکل ساز شود.

یک فایل در یکسری بلاک نگهداری میشود و دیتا های آن به این بلاک ها لینک شده است اما به این معنا نیست همه دیتا ها را بداند بلکه بلاک اول و محدوده را میداند (همانند آرایه) در نتیجه اگر توالی نداشته باشیم به مشکل بر می خوریم و ممکن است وارد دیتا های یک فایل دیگر شویم. سیستم فایل روی دیسک (به عنوان مثال، در سوپر بلوک) عدد inode فایلی را که تعداد لینک آن به صفر می رسد اما تعداد مراجع آن صفر نیست، ضبط می کند. اگر سیستم فایل زمانی فایل را حذف کند که تعداد مراجع آن به ۰ برسد، با حذف inode آن از لیست، لیست روی دیسک را به روز می کند. در بازیابی، سیستم فایل هر فایلی را در لیست آزاد می کند. در واقع اگر توالی وجود نداشته باشد ، log بطور کامل روی دیسک علامت گذاری نمی شود وضعیت دیسک به گونه ای خواهد بود که گویی چیزی روی دیسک نوشته نشده است و در هنگام بازیابی یا همه داده های یک فایل روی دیسک ظاهر می‌شوند یا هیچ‌کدام از آنها ظاهر نمی‌شوند و ممکن است اطلاعات فایل های دیگر دستخوش تغییر شود.

۲و۳ .پیاده سازی فراخوانی سیستمی گرفتن پردازه پدر و تغییر پدر یک پردازه

سیستم کال int get_parent_pid() و سیستم کال set_process_parent(int pid):

برای آنکه بتوان بین برنامه دیباگر و یک برنامه عادی تمایز قائل شد، فیلدهای isvirt و real_parent به استراکت proc اضافه می‌شوند. isvirt نشان دهنده دیباگر بودن یا نبودن پراسس هست و real_parent، پراسس پدر قبلی برنامه دیباگ شونده را ذخیره می‌کند تا در صورت صدا شدن سیستم کال get_parent_pid، بجای برگرداندن شناسه پراسس دیباگر، بتوانیم شناسه پراسس پدر اصلی را برگردانیم. دقت کنید که در این حین همیشه از نظر سیستم عامل پردازه پدر اصلی همان دیباگر است.

proc.h:

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int isvirt;                  // Is this a virtual process
  struct proc *real_parent;    // If this is a virtual process. Store the real parent.
};
```

get_parent_pid():

پدر پردازه فعلی را از myproc->parent بدست می‌آوریم و در متغیری مانند p ذخیره می‌کنیم. سپس تا وقتی که p یک برنامه دیباگر است (p (isvirt را به p->real_parent یا همان پدر حقیقی تغییر می‌دهیم.

proc.c:

```
int
get_parent_pid() {
  struct proc *p = myproc()->parent;
  cprintf("Process %d real parent is: %d\n", myproc()->pid, p->pid);
  while (p->isvirt) {
    p = p->real_parent;
  }
  return p->pid;
}
```

set_process_parent(int pid):

.اول از همه با جستجو در بین تمامی پردازه‌ها () پردازه با شناسه pid را میابیم فرض کنید این پردازه p باشد. پردازه فعلی را نیز از myproc بدست می‌آوریم. حال پدر فعلی p را به عنوان real_parent برای myproc ذخیره میکنیم. ضمنا myproc نیز مجازی می‌شود. (isvirt). در آخر پدر p برابر myproc می‌شود.

proc.c:

```
struct proc*
get_proc(int pid) {
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid)
      return p;
  }
  panic("There is no process with this pid.");
}

void
set_process_parent(int pid) {
  struct proc* p = get_proc(pid);
  struct proc* myp = myproc();
  myp->isvirt = 1;
  myp->real_parent = p->parent;
  p->parent = myproc();
  cprintf("proc %d parent changed to %d\n", p->pid, myp->pid);
}
```

خروجی استفاده از برنامه program و debugger (A و D):

```
$ program &
$ My pid: 4
Process 4 real parent is: 1
Parent pid: 1
debugger 4
Debugging pid 4
Debugger pid 5
Process 5 real parent is: 2
Debugger parent 2
KERNEL = SYS set process parent. Call for pid 4
proc 4 parent changed to 5
Parent changed.
My pid: 4
Process 4 real parent is: 5
Parent pid: 1
program exited
Debugger exited
$
```

debugger.c:

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

int main(int argc, char *argv[]){
    if(argc < 2){
        printf(2, "You must enter exactly 1 number!\n");
    }
    else
    {
        int saved_ebx, pid = atoi(argv[1]);
        asm volatile(
            "movl %%ebx, %0;"
            "movl %1, %%ebx;"
            : "=r" (saved_ebx)
            : "r"(pid)
        );
        // printf(1, "User: set_parent_process() called for pid: %d\n" , pid);
        printf(1, "Debugging pid %d\n" , pid);
        printf(1, "Debugger pid %d\n" , getpid());
        printf(1, "Debugger parent %d\n", get_parent_pid());
        asm("movl %0, %%ebx" : : "r"(saved_ebx));
        // exit();
        set_process_parent();

        printf(2, "Parent changed.\n");
        wait();
    }
    printf(2, "Debugger exited\n");
    exit();
}

```

program.c:

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

int main(int argc, char *argv[]){
    printf(1, "My pid: %d\n", getpid());
    printf(1, "Parent pid: %d\n", get_parent_pid());
    sleep(2000);
    printf(1, "My pid: %d\n", getpid());
    printf(1, "Parent pid: %d\n", get_parent_pid());
    printf(2, "program exited\n");
    exit();
}
```

تغییرات سایر فایل‌ها:

user.h:

```c
int calculate_sum_of_digits(void);
int get_file_sectors(int, void*);
int get_parent_pid(void);
void set_process_parent(void);
```

usys.S:

```
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(calculate_sum_of_digits)
SYSCALL(get_file_sectors)
SYSCALL(get_parent_pid)
SYSCALL(set_process_parent)
```

syscall.h:

```
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_calculate_sum_of_digits 22
#define SYS_get_file_sectors 23
#define SYS_get_parent_pid 24
#define SYS_set_process_parent 25
```

syscall.c:

```
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_calculate_sum_of_digits(void);
extern int sys_get_file_sectors(void);
extern int sys_get_parent_pid(void);
extern int sys_set_process_parent(void);
```

```
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_calculate_sum_of_digits] sys_calculate_sum_of_digits,
[SYS_get_file_sectors] sys_get_file_sectors,
[SYS_get_parent_pid] sys_get_parent_pid,
[SYS_set_process_parent] sys_set_process_parent,
};
```

sysproc.c:

```
int
sys_calculate_sum_of_digits(void) {
  int number = myproc()->tf->ebx;
  cprintf("KERNEL = SYS calculate sum of digits. Call for number %d\n", number);
  return calculate_sum_of_digits(number);
}

int
sys_get_parent_pid(void) {
  return get_parent_pid();
}

void
sys_set_process_parent(void) {
  int pid = myproc()->tf->ebx;
  cprintf("KERNEL = SYS set process parent. Call for pid %d\n", pid);
  return set_process_parent(pid);
}
```

defs.h:

```
C defs.h > calculate_sum_of_digits(int)
112   struct proc*     myproc();
113   void             pinit(void);
114   void             procdump(void);
115   void             scheduler(void) __attribute__((noreturn));
116   void             sched(void);
117   void             setproc(struct proc*);
118   void             sleep(void*, struct spinlock*);
119   void             userinit(void);
120   int              wait(void);
121   void             wakeup(void*);
122   void             yield(void);
123   int              calculate_sum_of_digits(int);
124   int              get_parent_pid();
125   void             set_process_parent(int);
126
```

sector_tester.c: (فایل‌های sector_file1.txt و sector_file2.txt به xv6 اضافه شده‌اند)

```
C sector_tester.c > print_sectors(char *)
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4   #include "fcntl.h"
5   #define BIG 3200
6
7
8   void make_file(char* path, int size) {
9       char big_data[BIG];
10
11      for(int i = 0; i < size; i++)
12          big_data[i] = 'a';
13
14      int fd = open(path, O_CREATE | O_RDWR);
15      write(fd, big_data, strlen(big_data));
16      close(fd);
17
18  }
19
20  void print_sectors(char* path) {
21      uint storing_addresses[13];
22      int fd = open(path, O_RDONLY);
23      get_file_sectors(fd, storing_addresses);
24
25      printf(2, "%s: ", path);
26      for(int i = 0; i < 13; i++)
27          printf(2, " %d,", storing_addresses[i]);
28      printf(1, "\n");
29  }
30
31  int main(int argc, char *argv[]){
32      make_file("sector_file2.txt", BIG);
33      make_file("sector_file3.txt", BIG - 512);
34
35      print_sectors("sector_file.txt");
36      print_sectors("sector_file1.txt");
37      print_sectors("sector_file2.txt");
38      print_sectors("sector_file3.txt");
39      exit();
40  }
41
```

خسته نباشید