

گزارش پروژه دوم آزمایشگاه سیستم عامل

پریا پاسه‌ورز 810101393

کوثر شیری جعفرزاده 810101456

پریسا یحیی‌پور فتیده 810101551

لینک گیت‌هاب: <https://github.com/parisa-yahyapour/Operating-System-Lab-01-F03>

هش آخرین کامیت: 4a2f2dde05cc3654602c8d34e1216dc15048dbcf

مقدمه

پرسش 1

کتابخانه‌های سطح کاربر در V6 برای ایجاد ارتباط میان برنامه‌های کاربر و کرنل به کار می‌روند این کتابخانه‌ها شامل توابعی هستند که از فراخوانی‌های سیستمی استفاده می‌کنند تا دسترسی به منابع سخت افزاری و نرم افزاری سیستم عامل ممکن شود. با تحلیل فایل‌های موجود در متغیر ULIB در XV6 توضیح دهید که چگونه این کتابخانه‌ها از فراخوانی‌های سیستمی بهره می‌برند؟ همچنین دلایل استفاده از این فراخوانی‌ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه‌ها را شرح دهید.

تغییر ULIB داخل فایل MakeFile و به شکل زیر تعریف می‌شود:

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

این متغیر مسیری به سمت کتابخانه‌های سطح کاربر را نشان می‌دهد که برنامه‌ها می‌توانند به کمک آنها link شوند.

ulib.c

از توابع موجود در این فایل برای پیاده‌سازی system call و wrapperها استفاده می‌شود.

Library functions

این توابع در واقع همان توابع پیاده‌سازی شده در کتابخانه c هستند که مستقیماً در فایل ulib.c قرار داده شده اند، بدون اینکه نیازی به دسترسی به kernel باشد. آنها عملکردهای مورد نیاز روی stringها را پیاده‌سازی می‌کنند.

کد:

```
char*
strcpy(char *s, const char *t)
{
    char *os;

    os = s;
    while((*s++ = *t++) != 0)
        ;
    return os;
}
```

تابع strcpy یک رشته را در دیگری کپی می‌کند.

کد:

```
int
strcmp(const char *p, const char *q)
{
    while(*p && *p == *q)
        p++, q++;
    return (uchar)*p - (uchar)*q;
}
```

تابع strcmp دو رشته را با هم مقایسه می‌کند.

کد:

```
uint
strlen(const char *s)
{
    int n;

    for(n = 0; s[n]; n++)
        ;
    return n;
}
```

تابع strlen طول یک رشته را بر می‌گرداند.

کد:

```
void*
memset(void *dst, int c, uint n)
{
    stosb(dst, c, n);
    return dst;
}
```

تابع memset از یک نقطه مشخص در حافظه شروع می‌کند و به تعداد n بایت از مقدار c را در آن می‌نویسد.
کد:

```
char*
strchr(const char *s, char c)
{
    for(; *s; s++)
        if(*s == c)
            return (char*)s;
    return 0;
}
```

تابع strchr یک Pointer به نقطه‌ای از رشته که اولین بار یک کاراکتر خاص در آن مشاهده شده است، برمی‌گرداند.
کد:

```
int
atoi(const char *s)
{
    int n;

    n = 0;
    while('0' <= *s && *s <= '9')
        n = n*10 + *s++ - '0';
    return n;
}
```

تابع atoi یک رشته از آن می‌گیرد و مقدار Integer متناظر با آن را بر می‌گرداند.

کد:

```
void*
memmove(void *vdst, const void *vsrc, int n)
{
    char *dst;
    const char *src;

    dst = vdst;
    src = vsrc;
    while(n-- > 0)
        *dst++ = *src++;
    return vdst;
}
```

تابع memmove از یک نقطه از حافظه به تعداد مشخصی بایت را به نقطه دیگری از حافظه کپی می‌کند.

System call Wrappers

از این wrapper-ها برای ارتباط برقرار کردن ارتباط با kernel استفاده می‌شود.

کد:

```
char*
gets(char *buf, int max)
{
    int i, cc;
    char c;

    for(i=0; i+1 < max; ){
        cc = read(0, &c, 1);
        if(cc < 1)
            break;
        buf[i++] = c;
        if(c == '\n' || c == '\r')
            break;
    }
    buf[i] = '\0';
    return buf;
}
```

تابع gets ورودی را از روی console می‌خواند و این کار را با کمک read system call انجام می‌دهد که با kernel ارتباط برقرار می‌کند تا ورودی کاربر را دریافت کند.

کد:

```
int
stat(const char *n, struct stat *st)
{
    int fd;
    int r;

    fd = open(n, O_RDONLY);
    if(fd < 0)
        return -1;
    r = fstat(fd, st);
    close(fd);
    return r;
}
```

تابع stat با کمک سیستم‌کال‌های open, close, fstat از جمله یک فایل را باز می‌کند، اطلاعات موجود در آن را دریافت می‌کند و در نهایت فایل را می‌بندد.

```
#include "syscall.h"
#include "traps.h"

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

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
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)
```

در این فایل system call-های پایه با استفاده از کد Assembly پیاده‌سازی شده‌اند. هر system call با استفاده از یک macro تعریف شده است که کد assembly متناظر با آن را می‌سازد.

.globl name

یک label می‌سازد که برای سایر فایل‌ها قابل دسترس باشد و بتوانند از این system call استفاده کنند.

```
movl $SYS_ ## name, %eax;
```

شماره system call را درون رجیستر eax% لود می‌کند. این شماره‌ها در فایل syscall.h ذخیره شده‌اند و برای Kernel مشخص می‌کنند که چه system call ای صدا زده شده‌است.

```
int $T_SYSCALL;
```

یک software interrupt را trigger می‌کند تا از user mode به kernel mode، switch کنیم. این interrupt (\$T_SYSCALL) توسط kernel هندل می‌شود و به واسطه آن محتوای register خوانده می‌شود.

```
ret
```

پس از به اتمام رسیدن system call، اجرا به user mode باز می‌گردد و محتوای مورد نظر در eax% ذخیره شده است.

printf.c

این فایل پیاده‌سازی‌های اولیه مورد نیاز برای تابع printf را انجام می‌دهد و با استفاده از آن می‌توان محتوای مورد نظر را با formatهای مختلف چاپ کرد.
کد:

```
static void
putc(int fd, char c)
{
    write(fd, &c, 1);
}
```

با استفاده از تابع putc می‌توان می‌توان یک کاراکتر را در file descriptor مورد نظر نوشت. برای این کار از write با استفاده می‌شود.

کد:

```
static void
printint(int fd, int xx, int base, int sgn)
{
    static char digits[] = "0123456789ABCDEF";
    char buf[16];
    int i, neg;
    uint x;

    neg = 0;
    if(sgn && xx < 0){
        neg = 1;
        x = -xx;
    } else {
        x = xx;
    }

    i = 0;
    do{
        buf[i++] = digits[x % base];
    }while((x /= base) != 0);
    if(neg)
        buf[i++] = '-';

    while(--i >= 0)
        putc(fd, buf[i]);
}
```

این تابع اعداد را format می‌کند به نحوی که بتوان با استفاده از putc آن را پرینت کرد. عدد می‌تواند integer یا hexadecimal باشد.

کد:

```
// Print to the given fd. Only understands %d, %x, %p, %s.
void
printf(int fd, const char *fmt, ...)
{
    char *s;
    int c, i, state;
    uint *ap;

    state = 0;
    ap = (uint*)(void*)&fmt + 1;
    for(i = 0; fmt[i]; i++){
        c = fmt[i] & 0xff;
        if(state == 0){
            if(c == '%'){
                state = '%';
            } else {
                putc(fd, c);
            }
        } else if(state == '%'){
            if(c == 'd'){
                printint(fd, *ap, 10, 1);
                ap++;
            } else if(c == 'x' || c == 'p'){
                printint(fd, *ap, 16, 0);
                ap++;
            } else if(c == 's'){
                s = (char*)*ap;
                ap++;
                if(s == 0)
                    s = "(null)";
                while(*s != 0){
                    putc(fd, *s);
                    s++;
                }
            } else if(c == 'c'){
                putc(fd, *ap);
                ap++;
            } else if(c == '%'){
                putc(fd, c);
            } else {
                // Unknown % sequence. Print it to draw attention.
                putc(fd, '%');
                putc(fd, c);
            }
            state = 0;
        }
    }
}
```

این تابع یک رشته از حروف را می‌گیرد و با کمک توابع `putc` و `printint` آن را کاراکتر به کاراکتر می‌نویسد.

umalloc.c

با استفاده از این فایل یک Memory allocator پیاده‌سازی می‌شود که درخواست‌های اختصاص یافتن و آزاد کردن حافظه را handle می‌کند. این ساختار بر اساس traditional Kernighan and Ritchie allocator design پیاده‌سازی شده است.
کد:

```
static Header*
morecore(uint nu)
{
    char *p;
    Header *hp;

    if(nu < 4096)
        nu = 4096;
    p = sbrk(nu * sizeof(Header));
    if(p == (char*)-1)
        return 0;
    hp = (Header*)p;
    hp->s.size = nu;
    free((void*)(hp + 1));
    return freep;
}
```

در تابع morecore از sbrk system call استفاده می‌شود. تابع morecore زمانی صدا زده می‌شود که memory pool متعلق به allocator برای یک Memory allocation جدید جای کافی نداشته باشد. sbrk system call فضای heap برنامه را با افزایش program break (انتهای data segment) گسترش می‌دهد و فضای حافظه بیشتری برای اختصاص یافتن خواهیم داشت.

sbrk(nu * sizeof(Header))

nu یونیت از حافظه را از سیستم عامل درخواست می‌کند، که هر unit معادل header struct است.

کد:

```
void*
malloc(uint nbytes)
{
    Header *p, *prevp;
    uint nunits;

    nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
    if((prevp = freep) == 0){
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
        if(p->s.size >= nunits){
            if(p->s.size == nunits)
                prevp->s.ptr = p->s.ptr;
            else {
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void*)(p + 1);
        }
        if(p == freep)
            if((p = morecore(nunits)) == 0)
                return 0;
    }
}
```

این تابع با استفاده از morecore حافظه درخواست می‌کند.

کد:

```
void
free(void *ap)
{
    Header *bp, *p;

    bp = (Header*)ap - 1;
    for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;
    if(bp + bp->s.size == p->s.ptr){
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if(p + p->s.size == bp){
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

این تابع با جابه‌جایی pointerها، فضای خالی موجود در حافظه را مدیریت می‌کند.

دلیل استفاده از این فراخوانی‌های سیستمی چیست؟

این system call ها به برنامه‌ها اجازه می‌دهند عملیات هایی مثل file access, memory management, process control به شکل امنی انجام پذیرند. دسترسی مستقیم به سخت‌افزار از طریق برنامه‌های سطح کاربر می‌تواند امنیت سیستم را دچار اشکال کند.

همچنین اینکه kernel برنامه‌های سطح پایین مثل خواندن از دیسک، نوشتن در فایل و schedule کردن process ها را انجام می‌دهد. System call ها در واقع یک interface برای کاربران برای دسترسی به این خدمات پیاده‌سازی می‌کنند.

بعضی operation ها مثل دسترسی به منبع سخت‌افزاری فقط در سطح kernel قابل انجام هستند و استفاده از system call ها به ما این اطمینان را می‌دهند که فقط kernel قابلیت انجام این عملیات‌ها را دارد.

تاثیر این فراخوانی‌ها بر عملکرد و قابلیت حمل برنامه‌ها چیست؟

System call ها به ما اجازه می‌دهند روی platform های مختلف از قابلیت‌های آنها استفاده کنیم، چون بخش ارتباط با hardware استانداردسازی شده است. اما اگر یک system call به طور خاص مثلا برای سیستم عامل xv6 پیاده‌سازی شده باشد، نمی‌توان از آن برای سیستم‌عامل‌های دیگر استفاده کرد.

از طرف دیگر استفاده از system call ها می‌تواند روی عملکرد سیستم تاثیر منفی داشته باشد، چون باید مدام بین user mode و kernel mode جابجا شویم اما در سیستم‌عامل‌های بزرگ عمل context switching، overhead کمی دارد و تاثیر قابل توجهی نخواهد داشت.

پرسش 2

فراخوانی های سیستمی تنها روش برای تعامل برنامه های کاربر با کرنل نیستند. چه روشهای دیگری در لینوکس وجود دارند که برنامههای سطح کاربر میتوانند از طریق آنها به کرنل دسترسی داشته باشند؟ هر یک از این روشها را به اختصار توضیح دهید.

می توانیم وقایع رخ داده در سیستم را به صورت زیر دسته بندی کنیم:

1. Exception

2. Interrupt

a. S.W interrupt

b. H.W interrupt

به طور کلی دسترسی به هسته با یک interrupt (وقفه) رخ می دهد.

وقفه سخت افزاری: این وقفه توسط دستگاه های سخت افزاری خارجی به صورت آسنکرون تولید می شود که به طور مثال می تواند برای ورودی کاربر، تکمیل عملیات I/O .. باشد. در حالتی که چنین وقفه ای رخ می دهد CPU استیت فعلی خود را ذخیره می کند و کنترل را به interrupt service routine منتقل می کند بعد از انجام کارهای مربوطه برای رفع interrupt کنترل به CPU باز می گردد.

وقفه نرم افزاری (trap): این وقفه توسط برنامه به صورت سنکرون ایجاد می شود. این وقفه ها معمولا به دلیل درخواست انجام یک کار توسط سیستم عامل ایجاد می شوند که به طور مثال می توان به درخواست یک حافظه اشتراکی ، خاتمه دادن به یک برنامه، بازکردن یک فایل و .. باشد.

از انواع trap ها می توان به :

1. Signal:

سیگنال های مختلفی در لینوکس نظیر SIGINT برای ایجاد یک وقفه و SIGKILL برای پایان دادن به یک

وقفه وجود دارد

2. System call

که به آن اشاره شده است.

از روش های دیگر می توان اشاره کرد(library API, file system interface, network interface):

1. Socket

کاربر می تواند از این طریق پیغام های خود را مبادله کند.

2. Netlink Socket

این دسته از socket ها برای ارتباط بین user-space and kernel مورد استفاده قرار می گیرند که معمولا برای network-related tasks استفاده می شوند

3. در لینوکس Pseudo-file-systems نیز وجود دارد همانند /dev استفاده از این فایل سیستم ها نیز

نیازمند دسترسی به هسته است.

4. exception ها نیز در صورت رخ دادن وقایعی همچون تقسیم بر صفر، دسترسی به حافظه ممنوعه و ... رخ می دهد که باعث می شود به kernel برویم

سازوکار اجرای فراخوانی سیستمی در XV6

بخش سخت افزاری و اسمبلی

پرسش 3

آیا باقی تله ها را نمیتوان با سطح دسترسی USER_DPL فعال نمود؟ چرا؟

خیر چنین کاری امکان ندارد. سطح ذکر شده سطح دسترسی کاربر است و در این سطح نباید اجازه دسترسی به هسته سیستم عامل (کرنل) و اجرای تله ها را داشت، در صورت انجام این عمل protection exception فعال می شود.

لازم به ذکر است که در صورتی که این اجازه داده میشد به protection kernel ایراد وارد می شد.

از دلایلی که این موضوع در انحصار سطح kernel است می توان به :

1. یک برنامه قصد سواستفاده از هسته را دارد و با این روش کنترل کل هسته سیستم عامل را در اختیار می گیرد
2. یک برنامه مخرب در چنین حالتی می تواند با آسیب به هسته تمام سطوح H.W & S.W آسیب وارد کند.
3. اگر برنامه کاربر دچار ایراد باشد در حالتی که در سطح kernel اجرا می شود این مشکل می تواند به تمام سطح هسته گسترش یابد.

پرسش 4

در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیراینصورت Push نمی شود. چرا؟

به طور کلی دو نوع پشته در سیستم عامل xv6 وجود دارند که عبارتند از پشته کاربر و پشته هسته که از اسم آنها مشخص است که چه کسی از آنها استفاده می کند. در صورتی که نیاز به تغییر دسترسی داشته باشیم (مثلا اجرای یک فراخوانی سیستمی) می دانیم که باید وضعیت پردازش فعلی ذخیره شود تا پس از برگشت به حالت کاربر بتوان اطلاعات آن را بازیابی کرد. esp و ss فعلی اشاره به پشته کاربر دارند. پس از تغییر حالت این دو متغیر به پشته هسته اشاره خواهند کرد. بنابراین پس از پایان کار در حالت هسته باید به حالت کاربر برگردیم و کار را از سر بگیریم. برای این برگشت باید مقدار esp و ss هم برگردانده شوند. بنابراین باید در جایی ذخیره شوند. پس آنها را push می کنیم. پس در صورتی که تغییر دسترسی صورت نگیرد همچنان داریم با پشته کاربر کار می کنیم و نیازی به ذخیره سازی و push کردن برای بازیابی آنها نیست.

بخش سطح بالا و کنترل کننده زبان سی تله

پرسش 5

در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در `argptr` بازه آدرس ها بررسی می گردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی بازه ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی `sys read` اجرای سیستم را با مشکل روبرو سازد.

برای دسترسی به آرگومان های فراخوانی های سیستمی در صورتی که از پشته برای انتقال آرگومان استفاده کنیم، باید از توابعی کمک بگیریم که این آرگومان ها را از پشته بخوانند و به محلی که می خواهیم انتقال دهند. این توابع شماره آرگومان (n) را می گیرند و با توجه به نوع آن آرگومان، در مقصد قرار می دهند. در ادامه به بررسی هر کدام از این توابع کمی می پردازیم.

1. `argint`:

`argint` تابع `fetchint` را فراخوانی می کند تا مقدار موجود در آن آدرس را از حافظه کاربر بخواند و درون `ip*` بنویسد. `fetchint` می تواند به سادگی آدرس را به یک اشاره گر تبدیل کند، زیرا کاربر و هسته از یک جدول صفحات مشترک استفاده می کنند، اما هسته باید اطمینان حاصل کند که این اشاره گر در محدوده بخش کاربری فضای آدرس قرار دارد. هسته سخت افزار جدول صفحات را طوری تنظیم کرده است که اطمینان حاصل شود فرآیند نمی تواند به حافظه خارج از محدوده حافظه خصوصی خود دسترسی داشته باشد: اگر یک برنامه کاربری تلاش کند که به حافظه ای با آدرسی برابر یا بیشتر از `p->sz` دسترسی پیدا کند، پردازنده باعث ایجاد خطای تقسیم بندی (`segmentation trap`) می شود و تله ی خطا (`trap`) باعث خاتمه ی فرآیند خواهد شد، همان طور که در بالا توضیح داده شد. با این حال، هسته می تواند به هر آدرسی که کاربر ممکن است ارسال کرده باشد دسترسی پیدا کند، بنابراین باید صریحاً بررسی کند که آدرس زیر مقدار `p->sz` باشد.

2. `argptr`:

تابع `argptr` برای دریافت آرگومان های فراخوان سیستمی که به داده هایی در فضای حافظه کاربر اشاره می کنند، استفاده می شود. این تابع بررسی می کند که آیا این اشاره گر معتبر است و آیا در محدوده ای از حافظه قرار دارد که کاربر اجازه دسترسی به آن را دارد یا خیر. این بررسی برای اطمینان از امنیت سیستم ضروری است، چرا که هسته سیستم باید از دسترسی کاربر به حافظه خارج از محدوده خود جلوگیری کند. در طول یک فراخوانی به `argptr`، دو مرحله بررسی انجام می شود. ابتدا، هنگام دریافت آرگومان، اشاره گر پشته ی کاربر بررسی می شود. سپس، خود آرگومان که یک اشاره گر به فضای کاربر است، بررسی می گردد.

3. `argstr`:

برای دریافت آرگومان‌های فراخوان سیستمی که به رشته‌های کاراکتری (رشته‌های NUL-terminated) در فضای کاربر اشاره می‌کنند، استفاده می‌شود. این تابع بررسی می‌کند که آیا اشاره‌گر معتبر است و آیا رشته به‌طور کامل در محدوده حافظه کاربر قرار دارد. این کار برای امنیت و جلوگیری از دسترسی به آدرس‌های نامعتبر ضروری است.

4. `argfd`:

تابع `argfd` برای دریافت فایل دیسکریپتور (شماره فایل) از آرگومان‌های یک فراخوان سیستمی استفاده می‌شود. این تابع علاوه بر دریافت دیسکریپتور، بررسی می‌کند که آیا دیسکریپتور معتبر است یا خیر و در صورت معتبر بودن، ساختار `struct file` متناظر با آن دیسکریپتور را برمی‌گرداند.

تمام این توابع بررسی می‌کنند که آدرس در فضای مجاز پردازش باشند. در غیر این صورت، ممکن است مشکلات مختلفی برای سیستم عامل پیش بیاید که در ادامه به آنها می‌پردازیم.

1. دسترسی به حافظه سایر پردازش‌ها: این موضوع باعث می‌شود داده سایر پردازش‌ها امنیت نداشته باشند و عوض کردن این داده‌ها توسط پردازش دیگر باعث اختلال در روند پردازش دوم می‌شود.

2. خراب کردن حافظه هسته: در صورتی که یک پردازش وارد فضای در دسترس هسته شود می‌تواند باعث ایجاد اختلال در سیستم عامل شود.

3. افزایش سطح دسترسی: یک پردازش با دسترسی به حافظه هسته می‌تواند سطح دسترسی خودش را افزایش دهد که باعث می‌شود به فعالیت‌هایی که تنها در حالت هسته قابل اجرا هستند، دسترسی داشته باشد و در روند فعالیت‌های سیستم اختلال ایجاد کند.

در تابع `argptr` در `xv6`، بازه‌ی آدرس‌ها به منظور اطمینان از معتبر بودن ناحیه حافظه‌ای که یک فرآیند درخواست دسترسی به آن را دارد، بررسی می‌شود. این تابع به سیستم‌عامل کمک می‌کند تا از تجاوز فرآیند به حافظه‌ای که مجاز به دسترسی به آن نیست، جلوگیری کند. در واقع، `argptr` از بروز خطاها و آسیب‌پذیری‌های امنیتی به دلیل دسترسی غیرمجاز به حافظه جلوگیری می‌کند.

تابع `sys_read` در واقع فراخوانی سیستمی مربوط به تابع `read` است. این تابع یک `file descriptor` را به همراه یک بافر و حداکثر تعداد بایت‌هایی که باید بخواند، می‌گیرد. این تابع در فایل `sysfile.c` قرار دارد چون یک فراخوانی سیستمی مربوط به `file management` می‌باشد. برای بررسی عملکرد به کد آن دقت می‌کنیم:


```

70 int sys_read(void)
71 {
72     struct file *f;
73     int n;
74     char *p;
75
76     if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
77         return -1;
78     return fileread(f, p, n);
79 }
80

```

در خط 76 تعدادی شرط وجود دارد. شرط آخر که با استفاده از تابع `argptr` انجام شده، در واقع بررسی می کند که فضای آدرس از ابتدای بافر تا انتهای آن در فضای آدرس دهی پردازش قرار بگیرد. اگر بدون بررسی بازه حافظه اقدام به خواندن کنیم، ممکن است پوینتر به ابتدای بافر و حداکثر تعداد بایت ها به گونه ای باشد که از حافظه پردازش فعلی خارج شویم و از حافظه ای که مربوط به این پردازش نیست بخوانیم یا در شرایط دیگر بنویسیم که باعث رخ دادن مشکلات امنیتی که توضیح دادیم، می شود.

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

یک برنامه ساده سطح کاربر بنویسید که بتوان از طریق آن، فراخوانی های سیستمی `getpid` در 6xv را اجرا کرد. ابتدا برنامه زیر را در قالب یک فایل جدید c می نویسیم. Process id با استفاده از `system call` `getpid()` به دست می آید.
کد:

```

Lab02 > C pid.c > ...
1  #include "user.h"
2  #include "types.h"
3
4  int main(int argc, char* argv[])
5  {
6      int pid = getpid();
7      printf(1, "Process ID: %d,\n", pid);
8      exit();
9  }

```

سپس آن را به makefile اضافه می‌کنیم:

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _pid\
```

برای مشاهده نحوه کارکرد این برنامه سطح کاربر، آن را در qemu اجرا می‌کنیم:

```
QEMU
Machine  View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ pid
Process ID: 3,
$ pid
Process ID: 4,
$ pid
Process ID: 5,
$ _
```

یک breakpoint در ابتدای تابع syscall قرار دهید. حال برنامه سطح کاربر نوشته شده را اجرا کنید. زمانی که به نقطه توقف برخورد کرد، دستور bt را در gdb اجرا کنید.

Breakpoint را روی خط 134 ام از فایل syscall.c قرار می‌دهیم. پس از برخورد به Breakpoint، دستور bt را اجرا می‌کنیم.

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) break syscall.c:134
Breakpoint 1 at 0x80104a60: file syscall.c, line 134.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:135
#1  0x80105aad in trap (tf=0x8dffffb4) at trap.c:43
#2  0x8010584f in alltraps () at trapasm.S:20
(gdb) █
```

دستور bt در واقع به معنای backtrace است. وقتی این دستور را اجرا می‌کنیم، call stack در لحظه‌ای که برنامه متوقف شده‌است، نمایش داده می‌شود. این call stack، از function call-هایی تشکیل شده است که پشت سر هم صدا زده شده‌اند تا به breakpoint برخورد کرده‌ایم. زمانی که یک user program نیاز به برقراری ارتباط با kernel دارد، از یک system call استفاده می‌کند. System call-ها معمولاً از طریق یک trap به CPU اطلاع داده می‌شوند که نیاز است سطح کنترل از user mode به kernel mode برود.

#2: alltraps at trapasm.S:20

Trap ایجاد شده توسط یک تابع assembly به نام alltraps دریافت می‌شود که در فایل trapasm.s تعریف شده‌است. زمانی که alltraps صدا زده می‌شود:

1. وضعیت کنونی registerهای CPU درون kernel stack ذخیره می‌شود. این قدم ضروری است، چرا که وقتی دوباره از kernel mode به user mode باز می‌گردیم، باید بتوانیم وضعیت سابق را بازیابی کنیم.

2. CPU به kernel mode می‌رود.

3. تابع trap صدا زده می‌شود که انواع trap-ها از جمله system call ها را مدیریت می‌کند.

#1: trap at trap.c:43

وقتی کنترل از alltraps به trap منتقل می‌شود، این تابع باید دلیل رخداد trap را متوجه شود. اول trap frame بررسی می‌شود تا بفهمیم trap ناشی از یک system call بوده یا Interrupt دیگری رخ داده است. در این نمونه چون user program یک system call را فراخوانی کرده است، syscall توسط trap فراخوانی می‌شود تا آن را مدیریت کند.

#0: syscall() at syscall.c:135

در تابع process، syscall، کنونی با کمک myproc () ارزیابی می‌شود. این تابع، proc structure مربوط به process ای را بر می‌گرداند که system call را صدا زده است. این ساختار شامل اطلاعاتی از process شامل حافظه و registerهای آن است.

شماره system call در eax register ذخیره شده است. با استفاده از unique number مربوط به handler، system call مناسب با آن فراخوانی می‌شود. پس از اتمام handle کردن system call، کنترل به user program باز می‌گردد.

حال دستور down را در gdb اجرا کنید.

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

دستور down به ما اجازه می‌دهد که از روی frame کنونی روی stack، به یک frame نزدیک‌تر به ته stack برویم. به عبارت دیگر، با این دستور یکی در function call-ها به جلو حرکت می‌کنیم. به طور معمول، current frame ما جایی است که برنامه در آن متوقف شده است. چون در اینجا روی نقطه توقف هستیم، پس جایی برای جلوتر رفتن وجود ندارد و پیام بالا هنگام صدا زدن تابع چاپ می‌شود.

محتوای رجیستر eax را که در tf می‌باشد چاپ کنید.

محتوای آن برابر 3 نخواهد بود، چرا که قبل از اجرای system call، getpid()، های دیگری نیز اجرا شده‌اند.

آنقدر دستور continue را اجرا می‌کنیم تا process id موجود در eax برابر با 3 شود:

```
(gdb) print myproc()->tf->eax
$1 = 7
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$2 = 15
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$3 = 10
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$4 = 10
```

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$5 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$6 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$7 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$8 = 16
```

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$9 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$10 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$11 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$12 = 16
```

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$13 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$14 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$15 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$16 = 16
```

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$17 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$18 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$19 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$20 = 16
```

```

(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$21 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$22 = 16
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$23 = 1
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135      struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$24 = 3

```

همانطور که می بینیم، محتوی id process هم سه است:

```

QEMU - Press Ctrl+Alt+G to release grab
Machine  View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
pid
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ Process ID: 3,
$

```


ارسال آرگومان های فراخوانی های سیستمی

در قدم اول باید در فایل syscall.h یک سیستم کال با شماره 22 به آن اختصاص می دهیم:

```
#define SYS_create_palindrome 22
```

حال برای اضافه کردن این فراخوانی سیستمی، در ابتدا تابع در دسترس کاربر را در user.h می کنیم تا با استفاده از آن به سیستم کال مدنظر متصل شود:

```
void create_palindrome(void);
```

نکته مهم این است که چون قرار است پارامترها را برحسب رجیستر ها پاس دهیم باید ورودی این تابع را void تعریف کنیم.

در قدم بعدی باید prototype مربوط به تابع سیستم کال را در فایل syscall.c اضافه کنیم:

```
extern int sys_create_palindrome(void);
```

همچنین باید آن را به آرایی syscalls اضافه کنیم:

```
[SYS_create_palindrome] sys_create_palindrome,
```

حال نیاز است تا تابع را در defs.h نیز تعریف کنیم:

```
void create_palindrome(int);
```

در فایل usys.s نیز تعریف تابع را اضافه می کنیم:

```
SYSCALL(create_palindrome)
```

حال در قدم بعدی در فایل proc.c بدنه ی تابع create_palindrome را تعریف می کنیم که با گرفتن عدد n پالیندروم آن را تولید کرده و نتیجه را بر روی سطح هسته چاپ می کند:

```

void create_palindrome(int num)
{
    int reversed = 0, remainder;
    int original = num;
    while (num != 0) {
        remainder = num % 10;
        reversed = reversed * 10 + remainder;
        num /= 10;
    }
    if (reversed==0)
    {
        cprintf("palindrome :%d00%d\n",original,reversed);
    }
    else
    {
        cprintf("palindrome :%d%d\n",original,reversed);
    }
}

```

حال باید تابع sys_create_palindrome را تعریف کنیم این تابع آرگومان ذخیره شده در رجیستر ebx را در سطح کرنل خوانده و آن را به تابع create_palindrome پاس می دهد تا محاسبات را انجام دهد:

```

int sys_create_palindrome(void)
{
    int n=myproc()->tf->ebx;
    create_palindrome(n);
    return 0;
}

```

برای تست کردن این بخش فایل create_palindrome.c را می نویسیم، در آن ابتدا مقدار قبلی رجیستر ebx را ذخیره کرده و در قدم بعدی مقدار وارد شده در کامند لاین را در رجیستر قرار می دهیم و پس از اتمام سیستم کال مقدار قبلی را به درون رجیستر باز می گردانیم.

```

C create_palindrome.c > main(int, char * [])
1  ✓ #include "types.h"
2    #include "syscall.h"
3    #include "user.h"
4  ✓ int main(int argc, char *argv[])
5      {
6          int number = atoi(argv[1]);
7          int last_value;
8          asm volatile(
9              "movl %%ebx, %0;"
10             "movl %1, %%ebx;"
11             : "=r"(last_value)
12             : "r"(number));
13          create_palindrome();
14          asm("movl %0, %%ebx"
15              :
16              : "r"(last_value));
17          exit();
18      }

```

حال برای تست این تابع را اجرا می کنیم:

```

$ create_palindrome 2003121
palindrome :20031211213002
$

```

نکته:

در سیستم عامل xv6 شماره ی سیستم کال در رجیستر eax قرار دارد و اولین آرگومان جهت پاس دادن به سیستم کال در ebx قرار می گیرد به همین دلیل ما نیز پارامتر دریافت شده از کامند لاین را درون این رجیستر قرار دادیم تا بتوانیم در سیستم کال مدنظر این مقدار را از رجیستر دریافت کرده و آن را به تابع پاس دهیم.

پیاده سازی فراخوانی های سیستمی

1 پیاده سازی فراخوانی سیستمی انتقال فایل

ابتدا در فایل syscall.h یک شماره به این فراخوانی جدید اختصاص می دهیم:

```
#define SYS_move_file 23
```

سپس یک declaration از تابع سطح کاربر این دستور در فایل user.h انجام می دهیم:

```
int move_file(const char *source_path, const char *destination_path);
```

پارامتر های این تابع توسط Stack باید پاس داده شوند و پارامتر ها در عکس بالا مشخص است. سپس prototype مربوط به تابع فراخوانی سیستمی جدید را در فایل syscall.c اضافه کنیم:

```
extern int sys_move_file(void);
```

و آن را به آراییه syscalls هم اضافه می کنیم:

```
[SYS_move_file] sys_move_file,
```

در فایل usys.s هم که مربوط به سطح اسمبلی است، فراخوانی سیستمی جدید را اضافه می کنیم:

```
SYSCALL(move_file)  
0 references
```

با توجه به اینکه این فراخوانی سیستمی جدید مربوط به file management است باید تابع فراخوانی سیستمی آن در sysfile.c باشد

```

459 √ int sys_move_file(void)
460 {
461     char *source_path;
462     char *destination_path;
463
464 √ if (argstr(0, &source_path) < 0 || argstr(1, &destination_path) < 0)
465 {
466     cprintf("Error at the beginning\n");
467     return -1;
468 }
469 begin_op();
470
471 struct inode *source_inode = namei(source_path);
472 √ if (source_inode == 0)
473 {
474     cprintf("The source file does not exists!\n");
475     return -1;
476 }
477 ilock(source_inode);
478 √ if (source_inode->type != T_FILE)
479 {
480     cprintf("The chosen one is not a file\n");
481     iunlockput(source_inode);
482     return -1;
483 }
484 char address[512];

```

```

485     for (int i = 0; i < strlen(destination_path); i++)
486     {
487         address[i] = destination_path[i];
488     }
489     address[strlen(destination_path)] = '/';
490     int length = strlen(destination_path) + 1;
491     for (int i = 0; i < strlen(source_path); i++)
492     {
493         address[length] = source_path[i];
494         length++;
495     }
496     address[length] = '\0';
497     char *result = address;
498     struct inode *destination_inode = create(result, T_FILE, 0, 0);
499     if (destination_inode == 0)
500     {
501         cprintf("Destination directory does not exists!\n");
502         iunlockput(source_inode);
503         return -1;
504     }
505     int num_bytes;
506     char buffer[512];
507     struct file *source_file = filealloc();
508     struct file *destination_file = filealloc();
509     if (!source_file || !destination_file)
510     {

```

```

511     cprintf("Error in copying file!\n");
512     iunlockput(source_inode);
513     iunlockput(destination_inode);
514     return -1;
515 }
516 source_file->ip = source_inode;
517 destination_file->ip = destination_inode;
518 while ((num_bytes = readi(source_inode, buffer, destination_file->off, sizeof(buffer))) > 0)
519 {
520     if (writei(destination_inode, buffer, destination_file->off, num_bytes) != num_bytes)
521     {
522         cprintf("Not able to write in destination\n");
523         iunlockput(source_inode);
524         iunlockput(destination_inode);
525         return -1;
526     }
527     destination_file->off += num_bytes;
528 }
529 iunlockput(source_inode);
530 iunlockput(destination_inode);
531 if (argstr(0, &source_path) < 0 || sys_unlink() < 0)
532 {
533     cprintf("The source file can't be deleted!\n");
534     return -1;
535 }

```

```

536     end_op();
537
538     return 0;
539 }

```

فرایند به این شکل است که ابتدا آرگومان ها را از استک دریافت می کنیم. سپس تلاش می کنیم تا برای فایل مبدا یک inode بسازیم که یک داده ساختار برای ذخیره اطلاعات مربوط به فایل است. در صورتی که موفقیت آمیز نباشد، به این معناست که فایل مبدا وجود خارجی ندارد و باید ارور بدهیم. همچنین type این فایل هم بررسی می شود تا در صورت اشتباه بودن خطا دهد. سپس باید آدرس مقصد را بسازیم. با توجه به اینکه فقط نام فایل مبدا و دایرکتوری مقصد را داریم، آدرس به شکل:

<directory name>/<source file name>

می باشد. پس از ساخت آدرس مقصد یک داده ساختار inode برای آن ایجاد می کنیم و در صورتی که مقدار برگشتی صفر باشد به این معناست که دایرکتوری ما وجود خارجی ندارد. بنابراین باید 1- برگردانیم. سپس یک بافر ایجاد می کنیم تا به کمک آن فرایند خواندن و نوشتن را انجام دهیم که در یک حلقه while قابل انجام است. پس از اتمام فرایند نوشتن در فایل جدید، فایل اصلی مبدا را حذف می کنیم. برنامه سطح کاربر test_move_file نام دارد. که باید آن را به makefile هم اضافه کنیم.

```

_test_move_file\

```

```

int main(int argc, char const *argv[])
{
    const char *source_file = argv[1];
    const char *destination_directory = argv[2];
    if (move_file(source_file, destination_directory) == 0)
    {
        printf(1, "success\n");
    }
    else
    {
        printf(1, "Moving file fails!\n");
    }
    exit();
    return 0;
}

```

در این برنامه آرگومان ها را از خط فرمان می گیریم و تابع مربوط به انتقال فایل را صدا میزنیم تا به کمک فراخوانی سیستمی که تعریف کردیم فایل را انتقال دهد.

برای تست ابتدا در ترمینال qemu یکبار تمام فایل های موجود را می بینیم:

```

README          2 2 2286
cat              2 3 16480
echo            2 4 15332
forktest        2 5 9648
grep            2 6 18700
init            2 7 15920
kill            2 8 15364
ln              2 9 15220
ls              2 10 17844
mkdir           2 11 15460
rm              2 12 15440
sh              2 13 28084
stressfs        2 14 16352
usertests       2 15 67460
wc              2 16 17216
zombie          2 17 15032
pid             2 18 15172
system_call_co  2 19 15700
create_palindr  2 20 15284
test_move_file  2 21 15416
testsort        2 22 15176
test_most_invo  2 23 15544
console         3 24 0
$

```

سپس فایل txt را به کمک دستور echo می سازیم و دوباره لیست را می بینیم تا مطمئن شویم ساخته شده است:

```
console 3 21 0
$ echo "group18" > s.txt
$ _
```

```
cat          2 3 16480
echo         2 4 15332
forktest     2 5 9648
grep         2 6 18700
init         2 7 15920
kill         2 8 15364
ln           2 9 15220
ls           2 10 17844
mkdir        2 11 15460
rm           2 12 15440
sh           2 13 28084
stressfs     2 14 16352
usertests    2 15 67460
wc           2 16 17216
zombie       2 17 15032
pid          2 18 15172
system_call_co 2 19 15700
create_palindr 2 20 15284
test_move_file 2 21 15416
testsort     2 22 15176
test_most_invo 2 23 15544
console      3 24 0
s.txt        2 25 10
$ _
```

حال یک دایرکتوری جدید می سازیم و مجددا لیست را چک می کنیم:

```
$ mkdir d
$
```



```
echo          2 4 15332
forktest      2 5 9648
grep          2 6 18700
init          2 7 15920
kill          2 8 15364
ln            2 9 15220
ls            2 10 17844
mkdir         2 11 15460
rm            2 12 15440
sh            2 13 28084
stressfs      2 14 16352
usertests     2 15 67460
wc            2 16 17216
zombie        2 17 15032
pid           2 18 15172
system_call_co 2 19 15700
create_palindr 2 20 15284
test_move_file 2 21 15416
testsort      2 22 15176
test_most_invo 2 23 15544
console       3 24 0
s.txt         2 25 10
d             1 26 32
$ _
```

سپس انتقال را انجام می دهیم:

```
$ test_move_file s.txt d
success
$ _
```

سپس لیست را دوباره چک می کنیم تا ببینیم فایل مبدا حذف شده باشد و داخل دایرکتوری هم چک می کنیم تا فایل جدید را ببینیم:

```

cat          2 3 16480
echo         2 4 15332
forktest     2 5 9648
grep         2 6 18700
init         2 7 15920
kill         2 8 15364
ln           2 9 15220
ls           2 10 17844
mkdir        2 11 15460
rm           2 12 15440
sh           2 13 28084
stressfs     2 14 16352
usertests    2 15 67460
wc           2 16 17216
zombie       2 17 15032
pid          2 18 15172
system_call_co 2 19 15700
create_palindr 2 20 15284
test_move_file 2 21 15416
testsort     2 22 15176
test_most_invo 2 23 15544
console      3 24 0
d            1 26 48
$ _

```

```

$ cat d/s.txt
"group18"
$ _

```

فایل با موفقیت منتقل شده است. حال خطاهای عدم وجود فایل مبدا و دایرکتوری مقصد هم تست می‌کنیم. در حال حاضر چون s.txt دیگر در ریشه وجود ندارد، اگر دوباره دستور انتقال را صدا بزنیم خطا می‌دهد:

```

$ test_move_file s.txt d
The source file does not exists!
Moving file fails!
$ _

```

حال دوباره فایل جدید می‌سازیم اما این بار به یک دایرکتوری که وجود ندارد انتقال می‌دهیم:

```

$ echo "18" > s.txt
$

```

```

$ test_move_file s.txt fi
Destination directory does not exists!
Moving file fails!
$

```

همان طور که انتظار می رفت خطا می دهد چون دایرکتوری fi وجود ندارد.

2 پیاده سازی فراخوانی سیستمی مرتب سازی فراخوانی های یک پردازش

ابتدا شماره system call را به فایل syscall.h اضافه می کنیم:

```
#define SYS_sort_syscalls 24
```

سپس باید آن را به لیست توابع موجود در فایل syscall.c نیز اضافه کنیم:

```
extern int sys_sort_syscalls(void);
```

و در لیست سیستم کال های موجود در syscall.c نیز اضافه شود:

```
[SYS_sort_syscalls] sys_sort_syscalls,
```

و تابع syscall در syscall.c نیز باید مشابه روبه رو تغییر کند. در این قسمت به ازای هر process، هر وقت system call جدیدی صدا زده شود که تاکنون صدا زده نشده است، شماره آن ذخیره می شود.

```

void syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        curproc->system_call_count++;
        int is_unique = 1;

        for (int i = 0; i < curproc->unique_syscalls_count; i++)
        {
            if (curproc->syscalls[i] == num)
            {
                is_unique = 0;
                break;
            }
        }
        // If unique, add to syscalls list
        if (is_unique && curproc->unique_syscalls_count < MAX_SYSCALLS)
        {
            curproc->syscalls[curproc->unique_syscalls_count++] = num;
        }

        curproc->tf->eax = syscalls[num]();
    }
    else
    {
        cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}

```

پیاده‌سازی سیستم‌کال داخل فایل sysproc.c انجام می‌گیرد. ابتدا process مورد نظر با توجه به process id آن یافت می‌شود (اگر یافت نشد، پیغام مناسبی چاپ می‌شود) و تابع sort روی سیستم‌کال‌های آن process صدا زده شده و نهایتاً به ترتیب id، چاپ می‌شوند.

```

int sys_sort_syscalls(void) {
    int pid;
    if (argint(0, &pid) < 0) return -1;

    struct proc *p = findproc(pid);
    if (!p) // Process not found
    {
        cprintf("Process not found!\n");
        return -1;
    }
    // Sort system calls for this process
    sort_syscalls(p->syscalls, p->unique_syscalls_count);

    // Print the sorted system calls
    for (int i = 0; i < p->unique_syscalls_count; i++) {
        cprintf("Syscall %d\n", p->syscalls[i]);
    }
    return 0;
}

```

Sort شدن به روش زیر انجام می‌گیرد (در فایل proc.c):

```

// Sort syscalls by ID
void sort_syscalls(int syscalls[MAX_SYSCALLS], int count) {
    for (int i = 0; i < count - 1; i++) {
        for (int j = i + 1; j < count; j++) {
            if (syscalls[i] > syscalls[j]) {
                int temp = syscalls[i];
                syscalls[i] = syscalls[j];
                syscalls[j] = temp;
            }
        }
    }
}

```

تابع findproc نیز به فایل proc.c اضافه می‌شود که یک Process خاص را در Ptable با توجه به ID پیدا کرده و برمی‌گرداند:

```

struct proc* findproc(int pid) {
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) return p;
    }
    return 0;
}

```

به proc struct موجود در proc.h نیز دو field دیگر اضافه می‌شود، یکی برای نگه داشتن id سیستم‌کال‌ها و یکی برای نگه داشتن تعداد system call‌های یونیک:

```

// 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
    int system_call_count; // Process systemcalls
    int syscalls[MAX_SYSCALLS]; // Array to store unique syscall IDs
    int unique_syscalls_count; // Number of unique system calls made
    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)
};

```

تعریف تابع findproc و sort_syscalls نیز بایستی به این فایل اضافه شود:

```

struct proc *findproc(int pid);
void sort_syscalls(int syscalls[MAX_SYSCALLS], int count);

```

تعریف تابع sort_syscalls را به user.h اضافه می‌کنیم:

```

int sort_syscalls(int pid);

```

نهایتاً system call مربوطه را باید به usys.s اضافه کنیم:

```

SYSCALL(sort_syscalls)

```

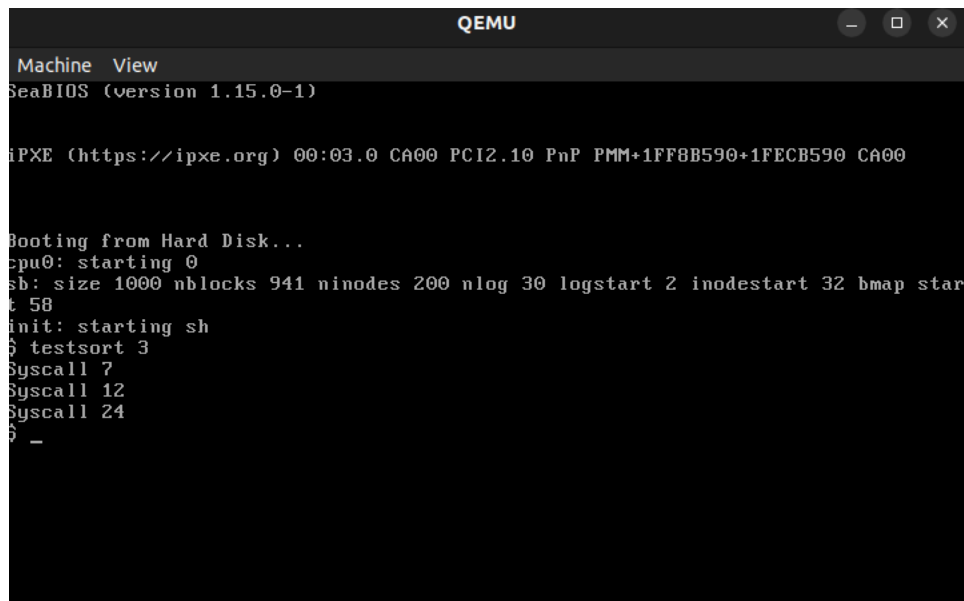
حال باید یک برنامه تست بنویسیم تا عملکرد را چک کنیم:

```
1 #include "types.h"
2 #include "user.h"
3
4 int main(int argc, char *argv[]) {
5     int pid = atoi(argv[1]);
6     sort_syscalls(pid);
7     exit();
8 }
```

و این برنامه تست را به MakeFile اضافه کنیم:

```
testsort\
```

نهایتاً می‌توانیم اجرای برنامه را تست کنیم:



The screenshot shows a QEMU window titled "QEMU" with a terminal interface. The terminal output is as follows:

```
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 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
$ testsort 3
Syscall 7
Syscall 12
Syscall 24
$ -
```

همانطور که می‌بینیم، در process شماره سه، سه system call فراخوانی شده است، با شماره 7 و 12 و 24.

- System call 7: exec (برنامه جدید را با برنامه قبلی جایگزین می‌کند)
- System call 12: sbrk (فضای حافظه برنامه (هیپ) را مدیریت می‌کند)
- System call 24: sort_sys_calls (همان سیستم کالی است که خودمان پیاده‌سازی کرده‌ایم)

3 پیاده سازی فراخوانی سیستمی برگرداندن بیشترین فراخوانی سیستم برای یک فرآیند خاص

ابتدا در فایل syscall.h یک شماره به این فراخوانی جدید اختصاص می دهیم:

```
#define SYS_get_most_invoked_syscall 25
```

سپس یک declaration از تابع سطح کاربر این دستور در فایل user.h انجام می دهیم:

```
int get_most_invoked_syscall(int pid);
```

پارامتر های این تابع توسط Stack باید پاس داده شوند و پارامتر ها در عکس بالا مشخص است. سپس prototype مربوط به تابع فراخوانی سیستمی جدید را در فایل syscall.c اضافه کنیم:

```
extern int sys_get_most_invoked_syscall(void);
```

و آن را به آرایه syscalls هم اضافه می کنیم:

```
[SYS_get_most_invoked_syscall] sys_get_most_invoked_syscall,
```

در فایل usys.s هم که مربوط به سطح اسمبلی است، فراخوانی سیستمی جدید را اضافه می کنیم:

```
0 references  
SYSCALL(get_most_invoked_syscall)
```

با توجه به اینکه این فراخوانی سیستمی جدید مربوط به process management است باید تابع فراخوانی سیستمی آن در sysproc.c باشد و تابع سطح کاربر آن در proc.c. به داده ساختار مربوط به هر process یک آرایه int به اندازه حداکثر تعداد فراخوانی سیستمی که سیستم می تواند داشته باشد می سازیم.


```

struct proc
{
    uint sz;                // Siz
    pde_t *pgdir;          // Pag
    char *kstack;          // Bot
    enum procstate state;  // Pro
    int pid;               // Pro
    int history[NPROC];
    int system_call_count;
    int syscalls[MAX_SYSCALLS];
    int unique_syscalls_count;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];
};

```

زمانی که یک process ساخته می شود این آرایه را با صفر مقدار دهی اولیه می کنیم و هربار که یک فراخوانی سیستمی در رخ داد در process مربوط خانه این آرایه را بهلاوه یک می کنیم.

```

138 void syscall(void)
139 {
140     int num;
141     struct proc *curproc = myproc();
142
143     num = curproc->tf->eax;
144     if (num > 0 && num < NELEM(syscalls) && syscalls[num])
145     {
146         curproc->system_call_count++;
147         curproc->history[num]++;
148         int is_unique = 1;
149
150         for (int i = 0; i < curproc->unique_syscalls_count; i++)
151         {
152             if (curproc->syscalls[i] == num)
153             {
154                 is_unique = 0;
155                 break;
156             }
157         }
158         // If unique, add to syscalls list
159         if (is_unique && curproc->unique_syscalls_count < MAX_SYSCALLS)
160         {
161             curproc->syscalls[curproc->unique_syscalls_count++] = num;
162         }
163
164         curproc->tf->eax = syscalls[num]();
165     }
166     else
167     {
168         cprintf("%d %s: unknown sys call %d\n",
169             curproc->pid, curproc->name, num);
170         curproc->tf->eax = -1;
171     }

```

در فایل proc.c ابتدا می بینیم آیا این pid معتبر است یا خیر. در صورت معتبر بودن در آرایه اختصاصی که برای این دستور ساخته ایم می گردیم تا ماکزیمم را پیدا کنیم. پس از پیدا کردن شماره آن سیستم کال که همان ایندکس آرایه است بر می گردانیم و سایر اطلاعات را هم چاپ می کنیم.

```

int get_most_invoked_syscall(int pid)
{
    struct proc *p = find_process_by_pid(pid);
    if (p == NULL)
    {
        cprintf("The process does not exists in ptable!\n");
        return -1;
    }
    int empty = 1;
    int max_index = -1;
    int max_num = 0;
    for (int i = 0; i < NPROC; i++)
    {
        if (p->history[i] > max_num)
        {
            empty = 0;
            max_num = p->history[i];
            max_index = i;
        }
    }
    if (empty)
    {
        cprintf("There is no system call in this process\n");
        return -1;
    }
    cprintf("Number of repeat: %d\nName of the system call: %s\n", max_num, syscall_names[max_index]);
    return max_index;
}

```

در فایل sysproc.c بودن آرگومان را بررسی کرده و در نهایت تابع قبلی را صدا می زنیم:

```

int sys_get_most_invoked_syscall(void)
{
    int pid;
    if (argint(0, &pid) < 0)
    {
        cprintf("Invalid input\n");
        return -1;
    }
    int result = get_most_invoked_syscall(pid);
    return result;
}

```

برای تست برنامه فایل test_most_invoked.c را می سازیم و به makefile اضافه می کنیم:

```
_test_most_invoked\
```

در فایل تست داریم:

```
int main(int argc, char const *argv[])
{
    int pid = atoi(argv[1]);
    int result = get_most_invoked_syscall(pid);
    if (result == -1)
    {
        printf(1, "The request failed!\n");
    }
    else
    {
        printf(1, "The most invoked system call in process with PID: %d\nThe system call code: %d\n", pid, result);
    }
    exit();
    return 0;
}
```

در واقع تابع را صدا کردیم و نتیجه آن را چاپ کردیم.

حال یک بار این تست را اجرا می کنیم:

```
$ test_most_invoked 1
Number of repeat: 18
Name of the system call: write
The most invoked system call in process with PID: 1
The system call code: 16
$ _
```

در واقع در xv6 دو process با pid-های 1 و 2 همواره در حال اجرا هستند که 1 همان init است و تشخیص داده شده که بیشترین سیستم کال آن مربوط به write است.

حال یک pid نامعتبر می دهیم:

```
$ test_most_invoked 576495876
The process does not exists in ptable!
The request failed!
$
```

همان طور که انتظار می رفت هیچ process-ای با این آیدی نداریم.

4 پیاده سازی فراخوانی سیستمی لیست کردن پردازش ها

در ابتدا باید در فایل syscall.h یک عدد به این سیستم کال اختصاص دهیم:

```
#define SYS_list_all_processes 26
```

در قدم بعدی prototype تابع سیستم کال را در فایل syscall.c را اضافه می کنیم:

```
extern int sys_list_all_processes(void);
```

علاوه بر آن در آرایه syscalls نیز آن را اضافه می کنیم:

```
[SYS_list_all_processes] sys_list_all_processes,
```

باید تابعی را که در قرار است توسط کاربر صدا زده شود را در user.h اضافه کنیم:

```
int list_all_processes(void);
```

حالا باید تابع را در usys.s نیز اضافه کنیم:

```
SYSCALL(list_all_processes)
```

برای اینکه تعداد فراخوانی های یک پردازش را بدانیم داخل فایل proc.h و در struct proc متغیر جدیدی را به نام system_call_count اضافه می کنیم:

```
int system_call_count;
```

این متغیر باید در هنگام ایجاد یک process مقدار اولیه صفر را بگیرد پس در همان فایل proc.c و تابع allocproc باید این مقداردهی اولیه را انجام دهیم:

```

static struct proc*
allocproc(void)
{
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;

    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->system_call_count=0;
    p->unique_syscalls_count=0;
    memset(p->syscalls, -1, sizeof(p->syscalls));

```

در قدم بعدی نیاز است تا هربار سیستم کالی فراخوانی شد به روزرسانی شود پس در فایل syscall.c در تابع syscall هرزمان که سیستم کال فراخوانی شده وجود داشت، تعداد سیستم کال های مربوط به این process را افزایش دهد.

```

void syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        curproc->system_call_count++;
    }
}

```

تابع `list_all_processes` به صورت زیر در فایل `proc.c` تعریف شده است:

```
int list_all_processes(void)
{
    struct proc *p;
    cprintf("Print Info\n");
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state == RUNNING)
        {
            cprintf("Name: %s-PID: %d-number of system calls %d \n", p->name, p->pid, p->system_call_count);
        }
    }
    return 0;
}
```

در این تابع تمامی `process` های سیستم که در حال اجرا هستند (یعنی در استیت `running` قرار دارند) اطلاعات زیر برای آن ها پرینت میشود:

PID- Name- Number of system calls

تابع `sys_list_all_processes` به صورت زیر تعریف شده است که در درون خود تابع `list_all_processes` را فراخوانی کند:

```
int sys_list_all_processes(void)
{
    cprintf("Enter kernel\n");
    list_all_processes();
    return 0;
}
```

در نهایت برای تست عملکرد این تابع از فایل تست زیر استفاده می کنیم:

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

void child_process(int id) {
    exit();
}

int main() {
    int pid;
    pid = fork();
    if (pid < 0) {
        printf(1, "Fork failed\n");
        exit();
    } else if (pid == 0) {
        child_process(pid);
    }
    sleep(5);
    list_all_processes(); // This should print the PIDs and syscall counts
    wait();
    exit(); // Exit the parent process
}
```

در این حالت فایل تست یک child ایجاد می کنیم و چون هردو با هم به ادامه برنامه می پردازند در if-else به child خاتمه می دهیم و در نهایت سیستم کال sleep را فراخوانی می کنیم این به دلیل این است که می خواهیم ببینیم آیا این دو سیستم کال در تعداد نهایی افزوده می شوند یا خیر.

```
init: starting sh
$ system_call_count
Enter kernel
Print Info
Name: system_call_cou-PID: 3-number of system calls 5
$ _
```