In His Name

Operating System Lab
First Project

Dorin Mosalman Yazdi
Marzieh Bagheri Nia
Kiavash Jamshidi

Fall 99

سوال یک

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

۱. **فایل ulib.c**:

- تابع strcmp: این تابع دو string را به عوان ورودی می‌گیرد و آن‌ها را با هم مقایسه می‌کند؛ فاقد فراخوانی سیستمی می‌باشد.
- تابع memmove: این تابع با شروع از source، به تعداد n (بایت) از رشته vsrc را در رشته vdst کپی می‌کند؛ فاقد فراخوانی سیستمی می‌باشد.
- تابع strcpy: این تابع یک string را در string دیگر کپی می‌کند؛ فاقد فراخوانی سیستمی می‌باشد.
- تابع strlen: سایز یک string را بر می‌گرداند؛ فاقد فراخوانی سیستمی می‌باشد.
- تابع strchr: این تابع یک char و یک *char به عنوان ورودی می‌گیرد؛ در صورتیکه char مورد نظر در string وجود داشته باشد (یک یا چندتا)، اشاره‌گر به اولین رخداد را بر می‌گرداند؛ فاقد فراخوانی سیستمی می‌باشد.
- تابع atoi: این تابع برای تبدیل *char به int استفاده می‌شود؛ فاقد فراخوانی سیستمی می‌باشد.
- تابع memset: برای آنکه تمامی کاراکترهای یک آرایه را به یک مقدار خاص initialize کنیم، از این تابع استفاده می‌کنیم؛ فاقد فراخوانی سیستمی می‌باشد.
- تابع stat: این تابع از فراخوانی‌های سیستمی open، close و fstate برای خواندن اطلاعات استفاده می‌کند؛ در واقعا اطلاعات را از فایل گرفته و در نهایت آن را close می‌کند.
- تابع gets: این تابع یک خط کامل (از ابتدای خط تا زمانی که به یکی از کاراکترهای \n و یا \r برسد) و یا به تعدادی که مشخص شده است (max) از ورودی buf می‌خواند؛ این تابع برای خواندن از console مورد استفاده قرار می‌گیرد و نیز در آن از فراخوانی سیستمی read استفاده می‌شود.

۲. **فایل printf.c**

- تابع puts: این تابع یک کاراکتر را بر روی file descriptor چاپ می‌کند و برای این کار از فراخوانی سیستمی write استفاده می‌کند.
- تابع printint: در این تابع یک عدد را در مبنای مورد نظر بر روی file descriptor نمایش می‌دهد و برای این کار از putc استفاده می‌کند.
- تابع printf: این تابع نیز یک رشته و یا عدد را به عنوان ورودی گرفته و آن را بر روی file descriptor چاپ می‌کند و برای این کار از putc استفاده می‌کند.

۳. **فایل umalloc.c**

- تابع free: از این تابع برای آزادسازی حافظه‌ای که توسط تابع malloc اشغال شده است، استفاده می‌کنیم؛ این تابع فاقد فراخوانی سیستمی می‌باشد.
- تابع morecore: از این تابع برای اشغال و تخصیص حافظه به تعداد مورد نظر (ورودی) استفاده می‌شود و برای این کار از فراخوانی سیستمی sbrk استفاده می‌کند.
- تابع malloc: از این تابع برای تخصیص حافظه به اشاره‌گری که به عنوان ورودی داده می‌شود استفاده می‌شود؛ این تابع فاقد فراخوانی سیستمی می‌باشد.

۴. **کتابخانه‌های C**

- Assert.h: این کتابخانه یک ماکرو به نام assert فراهم می‌کند؛ این ماکرو برای تایید assumptionهای برنامه و نیز چاپ کردن یک پیام تشخیصی در صورتی که assumptionها اشتباه بودند، به کار می‌رود.
- Ctype.h: این کتابخانه حاوی تعدادی از توابع کاربردی به منظور تست و نگاشت کاراکترها می‌باشد؛ تمامی این توابع یک عدد را به عنوان ورودی می‌گیرند؛ اگر این عدد برابر EOF و یا یک کاراکتر بدون علامت بود، توابع مقدار غیر 0 بر می‌گردانند در غیر اینصورت مقدار 0 را بر می‌گردانند.
- errno.h: این کتابخانه یک متغیر از نوع int به نام errno را تعریف می‌کند که توسط فراخوانی‌های سیستمی و نیز برخی توابع کتابخانه (در صورت بروز خطا) مقداردهی می‌شود تا بتواند نوع خطایی که رخ داده است را نشان دهد.

سوال دو

- استفاده از High-level IRQ flow exception handlers در موارد خاصی مانند خطاهای محاسباتی اعم از: تقسیم بر 0، overflow و ...
- استفاده از high level API
- استفاده از Chip level hardware interrupt برای رسیدگی کردن به مواردی مانند alarms و system and task counters برای مشخص کردن زمان توقف و ....

فراخوانی‌های سیستمی باعث می‌شوند برنامه وارد kernel mode شود که حضور موثر در آن نیازمند دسترسی‌های بالاتر (privilege) می‌باشد. پیاده‌سازی نادرست فراخوانی‌های سیستمی می‌تواند مشکلات زیادی را به وجود بیاورد و از امنیت سیستم بکاهد، وجود باگ در یک فراخوانی سیستمی می‌تواند باعث توقف OS شود و نیز عملیات فراخوانی‌های سیستمی به دلیل ارتباط مستقیمی که با HW دارند، می‌تواند بسیار پیچیده باشد؛ با توجه با آنچه که گفته شد، باید در پیاده‌سازی فراخوانی‌های سیستمی دقت کافی را بعمل آورد.

سوال سه

خیر! دستور int به فرایندهای user-level اجازه صدور signal interruptهای تعریف شده در table descriptor interrupt یا IDP را می‌دهد؛ بنابراین مقداردهی اولیه باید با دقت صورت گیرد تا از interrupt و exceptionهای غیرمجاز از سوی کاربر جلوگیری شود؛ این امر با تنظیم کردن مقدار DPL مربوط به descriptor gate یک interrupt و یا exception به عدد 0 انجام می‌شود. بنابراین متغیر control می‌تواند با مقایسه DPL با مقدار current-level privilege، تحرکات غیرمجاز را شناسایی کرده و یک protect general exception صادر کند. البته در برخی از موارد لازم است که یک فرایند user-level بتواند یک exception از پیش تعیین شده را صادر کند که این کار با تنظیم DPL به عدد ۳ امکان‌پذیر خواهد بود.

سوال چهار

هنگامی که یک Trap رخ می‌دهد (حال ممکن است دلایل مختلفی داشته باشد) یک System call صدا زده می‌شود؛ پیش از آنکه از user mode به kernel mode برویم، لازم است تا اطلاعاتی مانند ss(stack segment) و esp که حاوی اطلاعات stack کاربر هستند را به صورت trap-frame در kernel ذخیره کنیم تا در زمان بازگشت بتوانیم آن‌ها را بازیابی کنیم؛ اما هنگامی که در kernel mode هستیم اگر interrupt رخ دهد، نیازی به تغییر mode و دسترسی نیست بنابراین نیازی هم به ذخیره و بازیابی اطلاعات نداریم.

سوال پنج

در xv6 آرگومان‌ها به طور مستقیم به system callها فرستاده نمی‌شوند بلکه توسط توابعی مجزا خوانده شده و پس از بررسی اینکه حائز شرایط امنیتی سیستم هستند، در اختیار system callها قرار می‌گیرند. از جمله این توابع می‌توان به argint و argptr اشاره کرد:

Argint: Fetch the nth 32-bit system call argument.

Argptr: Fetch the nth word-sized system call argument as a pointer to a block of memory of size bytes. Check that the pointer lies within the process address space.

همانطور که در بالا به ساز و کار این توابع اشاره شد، برای آنکه اشاره‌گر مورد نظر در محدوده فضای حافظه فرایند در حال اجرا قرار بگیرد، باید در بازه معتبر (بازه‌ای که برای فرایند در نظر گرفته شده است) قرار داشته باشد و اگر که در این بازه نباشد، با برگرداندن ۱- (هشدار) یک segment trap ایجاد شده و فرایند kill می‌شود. تجاوز از این بازه معتبر امکان دسترسی یک پردازه به فضا یا اطلاعات حساس سخت‌افزار، سیستم عامل و حتی فرایندهای دیگر را ایجاد می‌کند و در واقع امکان نفوذ بدافزارها را نیز فراهم می‌کند؛ در فراخوانی سیستمی sys_read() نیز به همین شکل است و اگر بازه دسترسی فرایند بررسی و محدود نشود، ممکن است به داده‌هایی از سخت‌افزار یا فرایندهای دیگر دسترسی پیدا کرده و خرابکاری‌هایی را به صورت عمدی و یا غیرعمدی ایجاد نماید.

سوال شش

برای دسترسی به آدرس فراخوانی سیستمی sys_getpid به فایل kernel.asm که توسط دستور objdump (در makefile) تولید می‌شود مراجعه کرده و در آنجا آدرس این فراخوانی را می‌یابیم که برابر است با: 0x80105bd0
سپس مطابق آنچه که در صورت سوال گفته شد، آن را به یک اشاره‌گر به تابع از نوع تابع int(void) نسبت داده و این اشاره‌گر را فراخوانی می‌کنیم؛ در ادامه تصویر برنامه سطح کاربر که در فایل direct_syscall.c قرار دارد، نشان داده شده است:

File Edit Selection Find View Goto Tools Project Preferences Help

direct_syscall.c

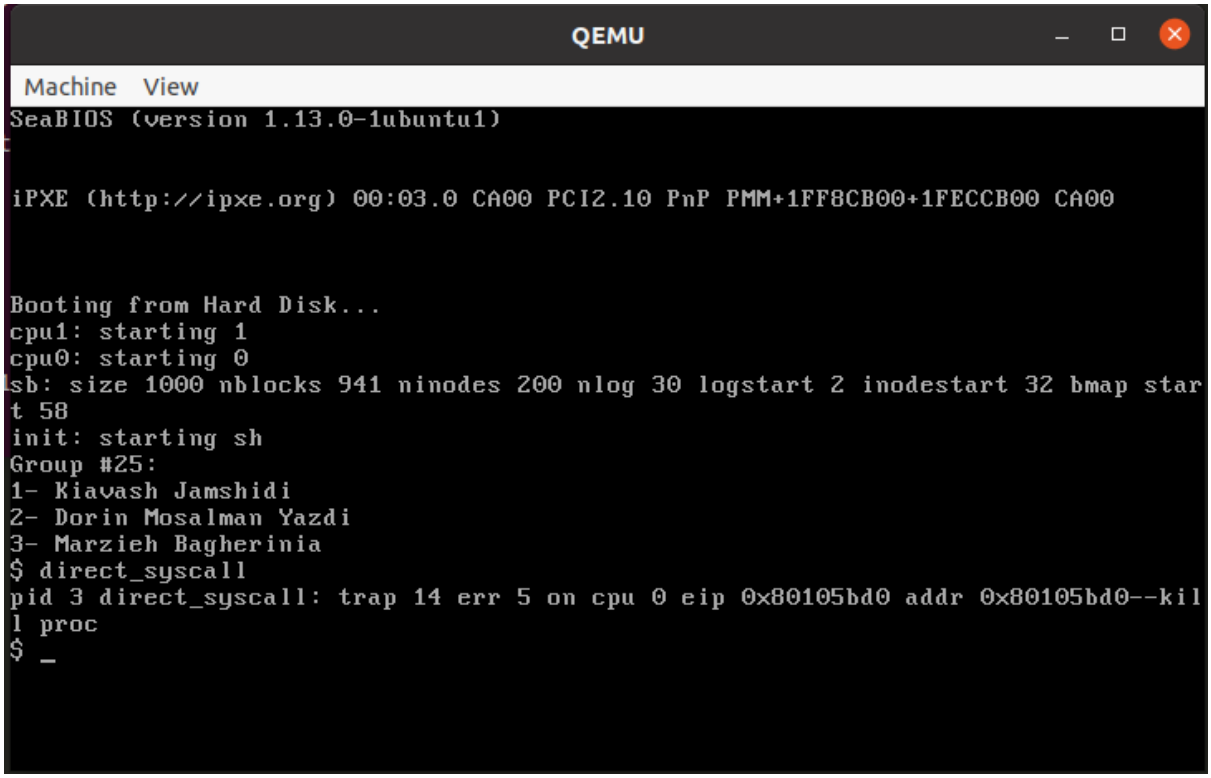
```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main()
{
    int (*fun_ptr)(void) = (int(*)(void)) 0x80105bd0;

    (*fun_ptr)();

    return 0;
}
```

در صورت اجرای این برنامه، پیام خطایی که در تصویر زیر نشان داده شده، نمایش داده می‌شود که در ادامه به تفسیر آن خواهیم پرداخت:



دسترسی مستقیم به syscall ها یک user misbehaved تلقی می‌شود و این کار مختص kernel mode است و عملا نباید از user mode یا user space از آن استفاده کرد؛ بنابراین در صورتی که این user misbehaved اتفاق بیوفتد، سخت‌افزار دستور را اجرا نکرده و آن را به صورت یک interrurpt نرم‌افزاری یا همان trap (که در این case خاص از نوع ۱۴ است) برای OS معرفی می‌کند و در نهایت با رخداد این trap، فرایندی که درحال اجرای این دستور بوده kill می‌شود.

سوال هفت

سیستم‌عامل‌ها باید بالاترین سطح آسایش ممکن و حداکثر ثبات و امنیت را در اختیار کاربران قرار دهند. به همین دلیل است که توسعه‌دهندگان سیستم‌عامل سعی دارند خطر عوارض احتمالی سیستم را در نتیجه سهل‌انگاری ناخواسته یا حملات هدفمند از خارج، تا حد ممکن پایین آورند. یکی از مهمترین اقداماتی که برای این منظور انجام شده، جداسازی دقیق هسته سیستم عامل (kernel) و برنامه‌های کاربردی یا فرایندهای کاربر است. در نتیجه باید استفاده از function callها را کنار گذاشته و از یک روش جایگزین برای آن استفاده نماییم؛ نتیجه این امر این است که برنامه ها و فرایندهایی که به سیستم تعلق ندارند، دسترسی مستقیمی به CPU و حافظه ندارند و در عوض به فراخوانی‌های سیستمی متکی هستند. در سیستم‌عامل‌های مدرن، فراخوانی سیستمی در صورتی استفاده می‌شود که یک کاربر یا پردازش کاربر نیاز به انتقال و خواندن اطلاعات به / از سخت افزار، سایر فرایندها یا خود kernel داشته باشد. بنابراین در این case خاص، می‌دانیم getpid() فقط از یک مکانِ حافظه داده‌ای را می‌خواند اما ممکن است سیستم‌عامل بخواهد در آن مکان چیزی بنویسد و با توجه به دسترسی به آن نقطه‌ی حافظه ممکن است داده‌ی پرتی را در حافظه بنویسد. همچنین ممکن است فرایندهای قبلی نیز داده‌های پرتی را در حافظه بنویسند به همین علت دستور باید از mode user جدا باشد و برنامه برای دسترسی به آن حافظه از طریق kernel اقدام کند. به همین علت این دستور به صورت system call پیاده‌سازی شده است.

پیاده‌سازی فراخوانی‌های سیستمی

- **مراحل اضافه کردن فراخوانی‌های سیستمی**

۱. ابتدا نام system callها را درون فایل syscall.h قرار داده و یک شناسه یکتا را برای آن تعریف می‌کنیم:

```
#define SYS_close  21
#define SYS_get_parent_id 22
#define SYS_get_children 23
#define SYS_get_family 24
#define SYS_trace_syscalls 25
#define SYS_print_syscalls_handler 26
#define SYS_reverse_number 27
```

۲. این system callها را درون فایل syscall.c، extern می‌کنیم:

```
extern int sys_uptime(void);
extern int sys_get_parent_id(void);
extern int sys_get_children(void);
extern int sys_get_family(void);
extern int sys_reverse_number(void);
extern int sys_trace_syscalls(void);
extern int sys_print_syscalls_handler(void);
```

۳. در جدول maping که در فایل syscall.c قرار دارد، اشاره‌گری به systemcall ها اضافه می‌کنیم:

```
[SYS_close]   sys_close,
[SYS_get_parent_id]   sys_get_parent_id,
[SYS_get_children]    sys_get_children,
[SYS_get_family]      sys_get_family,
[SYS_reverse_number]  sys_reverse_number,
[SYS_trace_syscalls]  sys_trace_syscalls,
[SYS_print_syscalls_handler] sys_print_syscalls_handler
};
```

۴. بدنه systemcallها را در فایل sysproc.c اضافه کرده و در آن‌ها می‌توانیم از توابع تعریف شده در proc.c که همان نمادهای قابل استفاده از این systemcallها در برنامه‌های سطح کاربر هستند، استفاده کنیم:

```
int
sys_get_family(void)
{
  int pid;

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

int
sys_trace_syscalls(void)
{
  int status;
  if(argint(0, &status) < 0)
    return -1;
  return trace_syscalls(status);
}


void
sys_print_syscalls_handler(void)
{
  return print_syscalls_handler();
}
```

```
int
sys_reverse_number(void) {
    int inputNumber;
    int sum=0;
    asm ("movl %%edi, %0;"
                    : "=r"(inputNumber)
                    :
                    : "%edi");

    for(;inputNumber != 0;inputNumber /= 10){
        sum = sum*10 + inputNumber % 10;
    }
    return(sum);
}

int
sys_get_parent_id(void)
{
  return get_parent_id();
}

int
sys_get_children(void)
{
  int pid;

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

۵. Prototypeهای فراخوانی‌های سیستمی مذکور را در فایل defs.h (به منظور خوانایی بیشتر در قسمت proc.c) اضافه می‌کنیم:

```
void            yield(void);
int             get_parent_id(void);
int             get_children(int);
int             get_family(int);
int             trace_syscalls(int);
void            print_syscalls_handler(void);
```

۶. Prototypeهای فراخوانی‌های سیستمی مذکور را در فایل user.h اضافه می‌کنیم تا برنامه‌های سطح کاربر بتوانند از این فراخوانی‌های استفاده نمایند:

```
int uptime(void);
int get_parent_id(void);
int get_children(int);
int get_family(int);
int reverse_number(void);
int trace_syscalls(int);
void print_syscalls_handler(void);
```

۷. به منظور از بین بردن پیچیدگی‌های استفاده از یک system call و برقراری ارتباط با سخت‌افزار، یک تابع پوشاننده (با استفاده از ماکروی SYSCALL) برای هر system call در فایل usys.S ایجاد می‌کنیم؛ این توابع وابستگی‌ها را مدیریت می‌کنند:

```
SYSCALL(uptime)
SYSCALL(get_parent_id)
SYSCALL(get_children)
SYSCALL(get_family)
SYSCALL(reverse_number)
SYSCALL(trace_syscalls)
SYSCALL(print_syscalls_handler)
```

۸. پیاده‌سازی این prototypeها در فایل proc.c آمده است که در ادامه در هر بخش به آنها خواهیم پرداخت.

- **ارسال آرگومان‌های فراخوانی‌های سیستمی**

برای پیاده‌سازی تابعی که قابلیت معکوس کردن عدد ورودی را داشته باشد، باید system call با نام sys_reverse_number را پیاده‌سازی کنیم؛ این system call از رجیستر عدد ورودی را می‌خواند و اعمال لازم را بر روی آن انجام می‌دهد تا معکوس شود؛ پس از آن، در برنامه سطح کاربری که نوشتیم، این عدد از رجیستر خوانده شده و چاپ می‌شود.

پیاده‌سازی system call ذکر شده در فایل sysproc.c قرار دارد و در ادامه نیز تصویر آن قرار داده شده است:

```
int
sys_reverse_number(void) {
    int inputNumber;
    int sum=0;
    asm ("movl %%edi, %0;"
                    : "=r"(inputNumber)
                    :
                    : "%edi");

    for(;inputNumber != 0;inputNumber /= 10){
        sum = sum*10 + inputNumber % 10;
    }
    return(sum);
}
```

برنامه سطح کاربر مذکور نیز در فایل reverse_number.c موجود است و نیز تصویر آن در ادامه قابل مشاهده است:

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

int main(int argc,char* argv[])
{
    int inputNumber;
    int tmp;
    asm ("movl %%edi, %0;"
                    : "=r" (tmp)
                    :
                    : "%edi");
    inputNumber = atoi(argv[1]);
    asm ("movl %0, %%edi;"
                    :
                    : "r" (inputNumber)
                    : "%edi");
    printf(1,"The result is %d\n",reverse_number());
    asm ("movl %0, %%edi;"
                    :
                    : "r" (tmp)
                    : "%edi");
    exit();
}
```

- **ردگیری تعداد فراخوانی‌های سیستمی فراخوانی شده**

برای پیاده‌سازی این فراخوانی سیستمی ابتدا باید ساختار تعریف processها را به گونه‌ای تغییر دهیم که زمانی که هر process یک system call را فراخوانی می‌کند، اطلاعات آن‌ها را ذخیره کند.

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

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
    curproc->syscalls_count[num - 1] ++;  //increase sys count for proc
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

تابع syscall() که هنگام فراخوانی هر system call صدا می‌شود، تابع مربوط به system call مورد نظر را اجرا می‌کند. در این تابع می‌توانیم پردازه فعلی را گرفته و syscall_count[] آن را افزایش دهیم. سپس هنگام فراخوانی سیستم کال مربوطه، اطلاعات آن را پرینت کند. ساختار داده ptable که در مد کرنل می‌توان از آن استفاده کرد، پوینتری به تمام پردازه‌ها دارد و با استفاده از آن، تابع پرینتِ تعداد system callهای فراخوانی شده به شکل زیر پیاده‌سازی می‌شود.

```
void
print_syscalls(){
  for(int i = 0; i < NPROC; i++){
    struct proc* p = &ptable.proc[i];
    if(p->pid != 0){
      cprintf("proc name: %s, proc id: %d\n", p->name, p->pid);

      for(int j = 0; j < NSYSCALLS; j++){
        if(p->syscalls_count[j] != 0)
          cprintf("    -%s: %d\n", sys_names[j+1], p->syscalls_count[j]);
      }
    }
  }

}
```

سیستم‌کال trace_syscalls (status) باید شمارش تمام سیستم‌کال‌ها را صفر کرده و به تابع print_handler() بگوید که هر ۵ ثانیه یک بار اطلاعات لازم را پرینت کند. این تابع به صورت زیر پیاده‌سازی شده است:

```
int
trace_syscalls(int status)
{
  for(int i = 0; i < NPROC; i++){
    struct proc* p = &ptable.proc[i];
    p->sys_count_stat = status;
  }
  global_status = status;
  if(status)
  {
    cprintf("**trace system calls start\n");
    ptable.syscount_status = status;
    acquire(&ptable.lock);
    for(int i = 0; i < NPROC; i++){
      struct proc* p = &ptable.proc[i];
      for(int j = 0; j < NSYSCALLS; j++){
        p->syscalls_count[j] = 0;
      }
    }
    release(&ptable.lock);
  }
  else
  {
    cprintf("**trace system calls end\n");
  }
  for(int i = 0; i < NPROC; i++){
    struct proc* p = &ptable.proc[i];
    if(strncmp(p->name, PRINT_HANDLER, sizeof(p->name)) == 0){
      return p->sys_count_stat;
    }
  }
  return 4;
}
```

برای اینکه به صورت اتوماتیک هر ۵ ثانیه یک بار اطلاعات پردازه‌ها پرینت شود، از ابتدای بوت شدن برنامه باید پروسه‌ای در حال اجرا باشد و در یک لوپ شرایط لازمه برای پرینت داده‌ها را چک کند. ابتدا چیزی پرینت نشده و زمانی که status مقدار یک می‌گیرد شروع به شمارش زمان کرده و ۵ ثانیه بعد تابع print_syscalls() فراخوانی می‌شود. همچنین در صورت تغییر مقدار آن به صفر از لوپ خارج شده و دوباره منتظر یک شدن آن می‌شود. این تابع به صورت زیر پیاده‌سازی شده است:

```
void
print_syscalls_handler()
{
  uint startticks;
  uint currticks;
  int count_stat = 0;
  struct proc* currproc = myproc();
  currproc->sys_count_stat = 0;
  safestrcpy(currproc->name, PRINT_HANDLER, sizeof(PRINT_HANDLER));

  for(;;)
  {
    for(int i = 0; i < NPROC; i++){
      struct proc* p = &ptable.proc[i];
      if(strncmp(p->name, PRINT_HANDLER, sizeof(p->name)) == 0){
        count_stat = p->sys_count_stat;
        if(count_stat)
        break;
      }
    }
    if(global_status == 1)
    {
      cprintf("start printing syscall trace");
      acquire(&tickslock);
      startticks = ticks;
      release(&tickslock);
      for(;;)
      {

        if(global_status == 0){
          break;
        }
        currticks = ticks;
        while(currticks - startticks < 1000)
        {
          acquire(&tickslock);
          currticks = ticks;
          release(&tickslock);
        }
        if(global_status == 1){
          print_syscalls();
          startticks = currticks;
        }
      }
    }
  }
}
```

برای فراخوانی سیستم‌کال‌ها باید در فایل sysproc.c توابع مورد نظر هر سیستم کال صدا زده بشود. این توابع به صورت زیر پیاده‌سازی شده‌اند:

```
int
sys_trace_syscalls(void)
{
  int status;
  if(argint(0, &status) < 0)
    return -1;
  return trace_syscalls(status);
}


void
sys_print_syscalls_handler(void)
{
  return print_syscalls_handler();
}
```

جهت فراخوانی این فرایندها و تست صحت درستی برنامه به یک برنامه سطح کاربر نیاز داریم؛ set_syscount_status(int) به این منظور به دستورهای سطح کاربر (در فایل set_syscount_status.c) اضافه شده است که با گرفتن ورودی status تابع trace_syscalls(status) را فراخوانی می‌کند. برنامه مذکور و نتیجه آن در ادامه دیده می‌شود:

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

int
main(int argc, char *argv[])
{
    int status = atoi(argv[1]);
    int result = trace_syscalls(status);
    printf(2, "trace syscalls result: %d\n", result);
    exit();
}
```

پیاده‌سازی فراخوانی سیستمی نمایش فرزندان پردازه

برای پیاده‌سازی این فراخوانی سیستمی، ابتدا باید system call را پیاده کنیم که با استفاده از ساختار proc (structure)، pid پدر فرایند در حال اجرا را برگرداند. پیاده‌سازی این system call، در فایل sysproc.c قرار دارد و در ادامه نیز تصویر آن قابل مشاهده است:

```
int
sys_get_parent_id(void)
{
  return get_parent_id();
}
```

پیاده‌سازی prototype این فراخوانی با نام get_parent_id() در فایل proc.c قرار دارد و در ادامه نیز تصویر آن قابل مشاهده است:

```
int
get_parent_id()
{
  struct proc *p = myproc();
  return p->parent->pid;
}
```

حال به پیاده‌سازی فراخوانی سیستمی get_children می‌پردازیم؛ ابتدا از یک تابع کمکی به نام get_process_children را پیاده می‌کنیم؛ این تابع به ازای هر فرایند با پیمایش بر روی ptable، بچه‌های آن پردازه را به صورت یک عدد (رشته‌ای) برمی‌گرداند. دو تابع کمکی دیگر با نام‌های pow (برای عملیات توان رسانی) و نیز num_of_degits (برای شمردن تعداد دیجیت‌های یک رشته عددی) وجود دارند که پیاده‌سازی آنها در ادامه آمده است؛ با استفاده از این توابع کمکی، فراخوانی سیستمی sys_get_children پیاده‌سازی می‌شود:
پیاده‌سازی این system call، در فایل sysproc.c قرار دارد و در ادامه نیز تصویر آن قابل مشاهده است:

```
int
sys_get_children(void)
{
  int pid;

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

پیاده‌سازی prototype این فراخوانی با نام ()get_children در فایل proc.c قرار دارد و در ادامه نیز تصویر آن قابل مشاهده است:

```
int
get_children(int pid)
{

  int all_children = get_process_children(pid);
  int num_of_all_children = num_of_degits(all_children);
  int final = all_children;

  if(num_of_all_children == 1 && all_children == 0)
  {
      return 0;
  }

  return final;
}
```

پیاده‌سازی توابع کمکیِ get_process_children، num_of_degits و pow نیز در فایل proc.c قرار دارند و در ادامه نیز تصویر آن قابل مشاهده است:

```c
int
get_process_children(int pid)
{
  struct proc *p;
  int children =0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->parent->pid == pid) {
      children = children*10 +p->pid;
    }
  }

  release(&ptable.lock);
  return children;
}

int num_of_degits(int num)
{
  int digits = 0;
  while(num > 0)
  {
    num = num/10;
    digits++;
  }
  return digits;
}

int pow(int ten, int n)
{
  for(int i=0;i<n-1;i++)
  {
    ten = ten * ten;
  }
  return ten;
}
```

حال برای راستی‌آزمایی system call اضافه شده، یک برنامه سطح کاربر را در فایلی با نام get_children.c قرار می‌دهیم و آن را اجرا می‌کنیم؛ تصویر برنامه مذکور در ادامه قرار داده شده است:

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

int main(int argc, char *argv[]) {
        int pid = getpid();
        int res;

        if(fork() == 0) {
                if(fork() != 0) {
                        wait();
                } else {
                        sleep(100);
                        printf(2, "child[1] --> pid = %d and parent's pid = %d\n", getpid(), get_parent_id());
                }

        } else {
                if(fork() !=0 ) {
                        wait();
                } else {
                        printf(2, "child[2] --> pid = %d and parent's pid = %d\n", getpid(), get_parent_id());
                        sleep(200);
                }
                wait();
        }

        if(getpid() == pid + 3) {
                res = get_children(pid - 1);
                printf(2, "Parent with %d pid's children are: %d \n", pid - 1, res);

                res = get_children(pid);
                printf(2, "Parent with %d pid's children are: %d \n", pid, res);

                res = get_children(pid + 1);
                printf(2, "Parent with %d pid's children are: %d \n", pid + 1, res);

                res = get_children(pid + 2);
                printf(2, "Parent with %d pid's children are: %d \n", pid + 2, res);

                res = get_children(pid + 3);
                printf(2, "Parent with %d pid's children are: %d \n", pid + 3, res);
        }

        exit();
}
```

همانطور که در صورت سوال خواسته شده است، با ایجاد تغییراتی کوچک در system call ِget_children، بتوانیم pid بچه‌ها، نوه‌ها، فرزندان نوه‌ها، نوه‌های نوه‌ها و ... یک پردازه را نیز نمایش دهیم. به این منظور تنها با ایجاد تغییراتی در تابع get_children، این کار را انجام می‌دهیم؛ پس از آنکه با پیمایش بر روی ptable فرزندان یک پردازه را به دست آوردیم، به ازای هر بچه، بچه‌های آن را نیز با استفاده از همان تابع کمکی get_process_children به دست آورده و این کار را آنقدر ادامه می‌دهیم تا دیگر بچه‌ای وجود نداشته باشد. فراخوانی سیستمی جدید را sys_get_family می‌نامیم؛ پیاده‌سازی این system call، در فایل sysproc.c قرار دارد و در ادامه نیز تصویر آن قابل مشاهده است:

```c
int
sys_get_family(void)
{
  int pid;

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

پیاده‌سازی prototype این فراخوانی با نام ()get_family در فایل proc.c قرار دارد و در ادامه نیز تصویر آن قابل مشاهده است:

```
int
get_family(int pid)
{
  int cur_pid;
  int num_of_cur_children;
  int cur_children;
  int all_children = get_process_children(pid);
  int num_of_all_children = num_of_degits(all_children);
  int final = all_children;
  int i=0;

  if(num_of_all_children == 1 && all_children == 0)
  {
      return 0;
  }
  while(i <= num_of_all_children)
  {
    if(all_children < 10){
      cur_pid = all_children;
    }
    else
    {
      cur_pid = all_children / pow(10,(num_of_all_children-1));
    }

    cur_children = get_process_children(cur_pid);
    num_of_cur_children = num_of_degits(cur_children);

    if(cur_children != 0)
    {
      final = final* pow(10,num_of_cur_children) + cur_children;
      all_children = all_children* pow(10,num_of_cur_children) + cur_children;
       num_of_all_children = num_of_all_children + num_of_cur_children;
      i = i+1;

    }
    else
    {
      i = i+1;
      num_of_cur_children = 0;
    }
    all_children = all_children % pow(10,num_of_all_children-1);
    num_of_all_children = num_of_all_children  - 1;
  }
  return final;
}
```

حال برای راستی‌آزمایی system call اضافه شده، یک برنامه سطح کاربر را در فایلی با نام **get_family.c** قرار می‌دهیم و آن را اجرا می‌کنیم؛ تصویر برنامه مذکور در ادامه قرار داده شده است:

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

int main(int argc, char *argv[]) {
        int pid = getpid();
        if(fork() == 0) {
                if(fork() != 0) {
                        wait();
                } else {
                        sleep(100);
                        printf(2, "child[1] --> pid = %d and parent's pid = %d\n", getpid(), get_parent_id());
                }
        }
        else {
                if(fork() != 0) {
                        wait();
                } else {
                        printf(2, "child[2] --> pid = %d and parent's pid = %d\n", getpid(), get_parent_id());
                        sleep(200);
                }
                wait();
        }

        if(getpid() == pid + 3) {
                res = get_family(pid - 1);
                printf(2, "Parent with %d pid's family are: %d \n", pid - 1, res);

                res = get_family(pid);
                printf(2, "Parent with %d pid's family are: %d \n", pid, res);

                res = get_family(pid + 1);
                printf(2, "Parent with %d pid's family are: %d \n", pid + 1, res);

                res = get_family(pid + 2);
                printf(2, "Parent with %d pid's family are: %d \n", pid + 2, res);

                res = get_family(pid + 3);
                printf(2, "Parent with %d pid's family are: %d \n", pid + 3, res);
        }

        exit();
}
```

آدرس مخزن Gitlab:

https://gitlab.com/dmosalman/operatingsystemlab2