Alexander M. Nicoara (A20301259)
CS450: Operating Systems
Programming Assignment 2
Professor Leung

## Introduction

In part 1 of this assignment, the objective was to trace the xv6 system call *read().*This system call has the arguments: int fd (the file descriptor), void *buf (the buffer), and size_t n (number of bytes to read). The *read()* system call reads from the file until n bytes is reached, then it returns either the number of bytes read or 0 (indicating end of the file). Upon an unsuccessful execution, -1 is returned.

## Tracing *read()*

*read()* is first called from the user level through a program that executes the function *read(fd,10,n).* Next, the usys.S puts the *read()* system call id (22 from syscall.h) into the eax register and then generates an interrupt with int that is a system call.

**in usys.S :: line 0**

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

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

vector.sS pushes the trap number onto the stack and calls alltraps to context save.

**in vector.sS :: line 0**

```
.globl vector64
vector64
  pushl $64
  jmp alltraps
```

In trapasm.S, the trap frame is built, data segments are pushed onto the stack, descriptors changed for memory access, and *trap(tf)* in trap.c is called.

**in trapasm.S :: line 0**

```
#include "mmu.h"

# vectors.S sends all traps here.
.globl alltraps
alltraps:
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  pushl %esp
  call trap
  addl $4, %esp
```

When we execute the function *trap()*, we check to see if the trap number references a system call. If it does (in our case), we then check to see if *myproc()* (which returns the current process) is killed. If it's not, then we set current process trapframe equal to the function's trapframe parameter. We next call the function *syscall()*, which calls syscall() inside of syscall.c

**in trap.c :: line 36**

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
```

In the function syscall(), we obtain the current process from the cpu by calling myproc(). We obtain the value of the eax register and store it into num. If num is within bounds of the of the possible system call numbers, then call the function inside the syscalls array with index num and store the return value in the register eax.

**in syscall.c :: line 131**

```
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]();
```

Since num = 22 in our case, *sys_read()* will be the function called. sys_read() is in sysfile.c and checks to see if the parameters are valid for execution (in our case they aren't since fd isn't valid). The first condition of the if statement tries to fetch the descriptor and the struct file.

**in sysfile.c :: line 69**

```
int
sys_read(void)
```

```
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 ||
```

*argfd()* is called as a result and is passed 0, 0, and the address location of f. In *argfd(),* we encounter an if statement condition which is the return value of the function *argint()*. In the condition, argint is called with n and &f, which returns the return value of the function fetchint, which ends up being 0. However, since fd doesn't contain a valid file, an exception is thrown and we return -1.

**in sysfile.c :: line 21**

```
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;

  if(argint(n, &fd) < 0)
```

**in syscall.c :: line 48**

```
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
```

**in syscall.c :: line 17**

```
int
fetchint(uint addr, int *ip)
{
  struct proc *curproc = myproc();

  if(addr >= curproc->sz || addr+4 > curproc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
```

**Return to argint() in sysfile.c**
**Return to argfd() in sysfile.c**

```
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
```

**Return to sys_read() in sysfile.c**

Since *argfd()* returns -1 and the conditional state only needs 1 true to be true, -1 is returned and we jump back to *syscall()* and then back to *trap()*.

```
  argint(2, &n) < 0 ||

  argptr(1, &p, n) < 0)
```

```
    return -1;

..

  Return to syscall() in syscall.c
  End syscall() in syscall.c
}
..
  Return to trap() in trap.c

    if(myproc()->killed)
      exit();
    return;
  }
```

Return back to trapasm.S and pop registers to restore original context. Then add trap number and error code together and returns.

**Return to line 23 in trapasm.S**

```
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

We keep returning from functions and eventually get back to user mode.

**Return to usys.S :: line 9**

```
  ret
```