

STUDENT NAME: Nikhil Sarika

## CS 450 Fall 2021 Assignment #02

Due: Monday, October 11th, 11:59 PM

### Problem 1:

usys.S has the assembly code defined to execute the system calls.

```
vagrant@ubuntu-xenial: ~/xv6
1 #include "syscall.h"
2 #include "traps.h"
3
4 #define SYSCALL(name) \
5     .globl name; \
6     name: \
7         movl $SYS_ ## name, %eax; \
8         int $T_SYSCALL; \
9         ret
10
```

The assembly code has a function defined called SYSTEM which takes in an argument called name to execute the specific system call.

For the close function the line number 7 is executed as

```
movl $SYS_close, %eax;
```

A "syscall.h" file is defined with all the system call number for all the system calls.

```
#define SYS_close 21
#define SYS_countTraps 22
~
```

For close 21 is allocated as the system call number.

As close() is a system call and the user doesn't have the privilege to execute it. The system calls are executed by the kernel with the help of Traps.

In X86 the traps are executed with help of an assembly instruction called "int". The trap function passes.

As we can see in the first image the int instruction is executed in line number 8.

During the execution of this "int" instruction the cpu context is saved on the kernel stack.

The "alltraps" assembly function saves the data within the registers and other data that is being used by the current program.

After saving the data, the alltraps function transfers the control to trap(tf) which is implemented in trap.c file.

```
36 void
37 trap(struct trapframe *tf)
38 {
39     if(tf->trapno == T_SYSCALL){
40         // int trapNumber = myproc()->tf->trapno;
41         // cprintf("This is a trap: %d!\n", trapNumber);
42         if(myproc()->killed)
43             exit();
44         myproc()->tf = tf;
45         syscall();
46         if(myproc()->killed)
47             exit();
48         return;
49     }
50 }
```

The trap(tf) function checks for the trapno that is saved in the trapframe struct.

The trap numbers are saved in the traps.h header file

```
#define T_SYSCALL      64      // system call
#define T_DEFAULT      500     // catchall
```

The trap number for system calls is defined as 64.

If the trapno is a match in line 39, The syscall() function in line 45 is executed.

Systemcall() function is defined in the syscall.c file.

```
133 void
134 syscall(void)
135 {
136     int num;
137     struct proc *curproc = myproc();
138     // cprintf("inside the syscall method");
139     num = curproc->tf->eax;
140     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
141         curproc->tf->eax = syscalls[num]();
142     } else {
143         cprintf("%d %s: unknown sys call %d\n",
144             curproc->pid, curproc->name, num);
145         curproc->tf->eax = -1;
146     }
147 }
```

The line 139 fetches the eax register value which is stored on the kernel stack.

As 64 is a valid value for the system call the syscall for close is executed.

The close system call is located in the sysfile.c folder.

```

93 int
94 sys_close(void)
95 {
96     int fd;
97     struct file *f;
98
99     if(argfd(0, &fd, &f) < 0)
100         return -1;
101     myproc()->ofile[fd] = 0;
102     fileclose(f);
103     return 0;
104 }

```

The sys\_close function executed the the close syscall.

The sys\_close validates the argument with the help of argfd() function defined in the samefile.

As we have passed an invalid value to argfd this method would return -1. The returned value is used to set the eax register in the syscall()'s method line number 141.

The last line in the usys.S's method takes this eax register value and returns it to the user. As -1 indicates a error response. The ret instruction returns the value and does the necessary error handling.

## Problem 2:

### Steps to Implement a SystemCall:

1. Modified user.h file to include method signature i.e the method definition and the return type.

```
int countTraps(void);
```

2. Modified syscall.h file to include the system call number. For countTraps() system call we have incuded a system call number 22

```
#define SYS_countTraps 22
```

3. Modified syscall.c file to include countTraps() in the array that stores all the system call numbers.

```

extern int sys_countTraps(void);

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

```

4. Modified `usys.S` assembly file to invoke the system call for our `countTraps()`

```
SYSCALL(countTraps)
```

5. Modified the `sysproc.c` file to implement our `countTraps()` system call.

```

int
sys_countTraps(void) {
    cprintf("Hello from syscall!\n");
    // struct proc *curProc = myproc();
    // int trapArray[27] = curProc->trapCounts;
    int i;
    for(i = 0; i < 600; i++) {
        if(myproc()->trapCounts[i] != 0) {
            cprintf("Trap No: %d! and Count of Trap: %d\n", i, myproc()->trapCounts[i]);
        }
    }
    // for(i = 0; i < 600; i++) {
    //     myproc()->trapCounts[i] = 0;
    // }
    return 0;
}

```

6. Modified `proc.h` to edit the `proc` struct to include a custom array that holds the count of all the traps that occurred in a user process.

```

// 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
    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 trapCounts[600];
};

```

7. Modified trap.c file to increment the trapCounts array that we have included in the proc struct. The entries are updated based on the trapno that is part of the trapframe. Line 45 is added to increment the entries in trapcounts array which represents the number of traps.

```

36 void
37 trap(struct trapframe *tf)
38 {
39     //acquire(&tickslock);
40     //myproc()->trapCounts[myproc()->tf->trapno]++;
41     //release(&tickslock);
42     if(tf->trapno == T_SYSCALL){
43         // int trapNumber = myproc()->tf->trapno;
44         // cprintf("This is a trap: %d!\n", trapNumber);
45         myproc()->trapCounts[myproc()->tf->trapno]++;
46         if(myproc()->killed)
47             exit();
48         myproc()->tf = tf;
49         syscall();
50         if(myproc()->killed)
51             exit();
52         return;
53     }

```

8. Modified the proc.c file to edit the allproc() method. The countTraps array is reset to default values before allocation of memory for each user process. The values of countTraps are set to zero before each allocation.

```

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++;

    release(&ptable.lock);

    // Allocate kernel stack.
    if((p->kstack = kalloc()) == 0){
        p->state = UNUSED;
        return 0;
    }
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;


    int i;
    for(i = 0; i < 600; i++){
        p->trapCounts[i] = 0;
    }

    return p;
}

```

Testing of Implementation:

1. Implemented mycode.c file to test the count of traps.

 vagrant@ubuntu-xenial: ~/xv6

```

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

int
main(int argc, char *argv[])
{
    getpid();
    // getpid();
    countTraps();
    exit();
}

```

a. Included two system calls to get the count of number of traps in the program. The program is included in the make file.

Test 1 output:

```

vagrant@ubuntu-xenial:~/xv6$ make qemu-nox
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.
.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' >
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0249222 s, 205 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00108146 s, 473 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
335+1 records in
335+1 records out
171624 bytes (172 kB, 168 KiB) copied, 0.00133803 s, 128 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodes
init: starting sh
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 13376
echo       2 4 12452
forktest  2 5 8172
grep       2 6 15204
init       2 7 13044
kill       2 8 12492
ln         2 9 12404
ls         2 10 14620
mkdir      2 11 12520
rm         2 12 12496
sh         2 13 23136
stressfs   2 14 13172
usertests  2 15 56056
wc         2 16 14032
zombie     2 17 12228
mycode     2 18 12296
console    3 19 0
$ mycode
Hello from syscall!
Trap No: 64 and Count of Trap: 4
Total Trap Count : 4

```

2. Modified the mycode.c file to include fork() system call. The test case is similar to the first one except that the fork starts a child process.



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

int
main(int argc, char *argv[])
{
    getpid();
    int pid = fork();
    if (pid == 0) {
        printf(1, "%s", "inside child process\n");
    } else {
        printf(1, "%s", "inside parent prcoess\n");
    }

    countTraps();
    // printf(1, "%d", err);
    // printf(stderr, "%s", "Error from kernel");
    exit();
}
```

Test Case 2 Ouput:

```
vagrant@ubuntu-xenial:~/xv6$ make qemu-nox
gcc -fno-pic -static -fno-builtin -fno-stri
ld -m elf_i386 -N -e main -Ttext 0 -o _m
objdump -S _mycode > mycode.asm
objdump -t _mycode | sed '1,/SYMBOL TABLE/d
./mkfs fs.img README _cat _echo _forktest _
nmeta 59 (boot, super, log blocks 30 inode
balloc: first 593 blocks have been allocate
balloc: write bitmap block at sector 58
qemu-system-i386 -nographic -drive file=fs.
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog
init: starting sh
$ mycode
inside child process
hello from syscall!
Trap No: 64 and Count of Trap: 22
Total Trap Count : 22
inside parent procoess
hello from syscall!
Trap No: 64 and Count of Trap: 27
Total Trap Count : 27
```

The second test case has additional traps that are a result of the `fork()` system call.



