**Homework 3**

**Part 1**

All system calls are stored as pointers to functions in the syscalls array. The first thing the syscall() function does is to get the index of the function pointer. The mapping from index to system call can be found in syscall.h. Therefore to print the system call you must switch on the index selected. Unfortunately, I couldn't think of a more elegant way to do this. I couldn't find a way to retrieve the name of the system call without hardcoding it. GDB displays the name but that's because it looks up the pointer in the symbol table.

To get the return code of the system call you must look in the eax member of the Trapframe struct after the call returns. Note that when this code runs, the output of our debugging is mixed with the actual results of the the system calls. For example the write() system call will output characters to the console. I'm including below the finished code of the syscall() functions:

void

syscall(void)

{

int num;

num = proc->tf->eax;

if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

proc->tf->eax = syscalls[num]();

switch (num) {

case SYS\_fork:

cprintf("fork -> ");

break;

case SYS\_exit:

cprintf("exit -> ");

break;

case SYS\_wait:

cprintf("wait -> ");

break;

case SYS\_pipe:

cprintf("pipe -> ");

break;

case SYS\_read:

cprintf("read -> ");

break;

case SYS\_kill:

cprintf("kill -> ");

break;

case SYS\_exec:

cprintf("exec -> ");

break;

case SYS\_fstat:

cprintf("fstat -> ");

break;

case SYS\_chdir:

cprintf("chdir -> ");

break;

case SYS\_dup:

cprintf("dup -> ");

break;

case SYS\_getpid:

cprintf("getpid -> ");

break;

case SYS\_sbrk:

cprintf("sbrk -> ");

break;

case SYS\_sleep:

cprintf("sleep -> ");

break;

case SYS\_uptime:

cprintf("uptime -> ");

break;

case SYS\_open:

cprintf("open -> ");

break;

case SYS\_write:

cprintf("write -> ");

break;

case SYS\_mknod:

cprintf("mknod -> ");

break;

case SYS\_unlink:

cprintf("unlink -> ");

break;

case SYS\_link:

cprintf("link -> ");

break;

case SYS\_mkdir:

cprintf("mkdir -> ");

break;

case SYS\_close:

cprintf("close -> ");

break;

default:

panic("should never get here\n");

}

cprintf("%d\n", proc->tf->eax);

} else {

cprintf("%d %s: unknown sys call %d\n",

proc->pid, proc->name, num);

proc->tf->eax = -1;

}

}

**Part 2**

The file date.c contains the userpace code that calls the date() system call. That file contains a main() method because presumably it gets compiled into its own binary and executes in its own process via fork() (I presume thats's what happens after I type date in the xv6 prompt). The interface to the date() system call is defined in user.h: int date(struct rtcdate \*r). Note that the userspace code in date.c calls the date() function which is not the actual system call implementation. If userspace code was able to call the implementation directly then that would be a violation of the isolation between the kernel and userspace.

The actual implementation of the system call can be found in sysproc.c, under the sys\_date() function. How do we get there from the call to date() in date.c? When date.c is linked, date() points to a piece of assembly code that is defined in usys.S. That assembly code stores the index to sys\_date() in the eax registry. That's the index into the syscalls array defined in 'syscall.c'. In addition, that piece of assembly also issues an interrupt that forces us to switch over to kernel mode and start executing with different permissions. The interrupt handler eventually lands in syscall(), which as we saw in Part 1 selects the function to call based on the index we just stored in eax. Here's what sys\_date() looks like:

int

sys\_date(void)

{

struct rtcdate \*r;

if (argptr(0, (char \*\*) &r, sizeof(struct rtcdate)) < 0)

return -1;

cmostime(r);

return 0;

}

Note that arguments to the function are retrieved using the arg\*() functions in syscall.c.