

CS 370 – Project #2

Purpose: Become familiar with the **xv6** Teaching Operating System, shell organization, and system calls
Points: 100

Introduction

The **xv6** OS is an educational OS designed by MIT for use as a teaching tool to give students hands-on experience with topics such as shell organization, virtual memory, CPU scheduling, and file systems.

Resources

The following resources provide more in depth information regarding **xv6**. They include the **xv6** reference book, tools for installing, and guidance to better understand **xv6**.

1. **xv6** Reference Book: <https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf>
2. Lab Tools Guide: <https://pdos.csail.mit.edu/6.828/2020/tools.html>
3. Lab Guidance: <https://pdos.csail.mit.edu/6.828/2020/labs/guidance.html>

Project

Complete the following steps.

- Implement a new system call:
`getreadcount()`
- Implement a user level C program, `getreadcount.c`, to call the new system call.

Submission

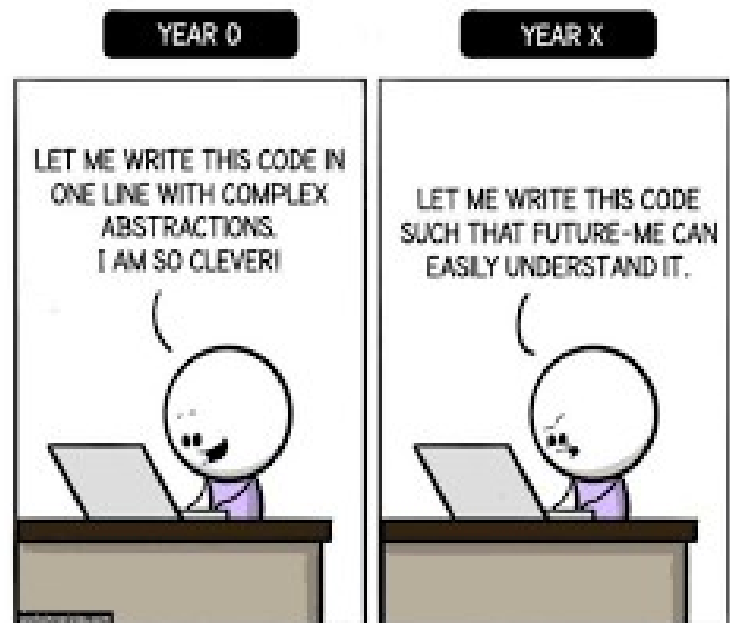
When complete, submit:

- A copy of the *zipped xv6 folder* (not qemu) via the class web page (assignment submission link). After committing (`git commit`) your final changes, clean and create a zip of the entire project:

```
xv6 $ git commit
xv6 $ make clean
xv6 $ cd ..
$ zip project2.zip -r xv6
```

- Submit the `project2.zip` via Canvas.

Submissions received after the due date/time will not be accepted.



System Calls

Implement the following system call.

getreadcount() → Returns number of times the **read()** system call has been invoked since the system was booted.

You can find an explanation of the system call process in Chapter 1 Operating System Interfaces of the **xv6** textbook. To simplify the coding, we will not acquire a lock.

Setting up a System Call

A system call must be both available in the user space and kernel space. It must be available in user space so that it can be used by other programs and it must be available in kernel space so it has permission to access resources that are prohibited from user mode.

User Mode Files (xv6/user)

When we call a system call, we are actually invoking a trap in the kernel. However, we wish to avoid writing assembly code or traps in our programs. So wrappers, written in a high-level language, will “wrap” around the assembly code which will allow us to use them as functions.

FILE: **user.h**

Listed in this file are all the functions that are available in user space as a wrapper for the system calls. The functions are written in C and it looks very similar to a function prototype. It is important to make sure that the arguments declared with this function matches the arguments when the function is defined (they are defined in **sysproc.c** which will be discussed later.)

- ACTION: Edit **user.h** to add a line for the new system call. Hint; Look at how other system calls are declared and follow the pattern that best suits each new system call.

FILE: **usys.pl**

This is a Perl file. This is where the system call is actually performed via Perl to RISC-V assembly.

```
9  sub entry {
10      my $name = shift;
11      print ".global $name\n";
12      print "${name}:\n";
13      print " li a7, SYS_${name}\n";
14      print " ecall\n";
15      print " ret\n";
16 }
```

A brief walk-through of the code:

- Line 11: Emit **\$name** (system call) to symbol table (scope GLOBAL).
- Line 12: A label with the **\$name**.
- Line 13: Load immediate (system call number) which specifies the operation the application is requesting into register a7.
- Line 14: System calls are called executive calls (**ecall**). Perform **ecall** (safe transfer of control to OS). Exception handler saves temp regs, saves **ra**, and etc.
- Line 15: Return.

The code that follows after this is repeatedly calling for all the different system calls.

- ACTION: Edit **usys.pl** to add a line for the new system calls. Hint: Look at how other system calls are declared and follow the pattern.

Kernel Mode Files (xv6/kernel)

The files below are where the system call is made available to the kernel.

FILE: **syscall.h**

This file is the interface between the kernel space and user space. From the user space you invoke a system call and then the kernel can refer to this to know where to find the system call. It is a simple file that defines a system call to a corresponding number, which will be later used as an index for an array of function pointer.

- ACTION: Edit **syscall.h** to add a line for the new system calls.

FILE: **syscall.c**

This file has two (2) sections where the system call must be added. Starting around line 83, is a portion where all the functions for the system calls are defined within the kernel.

- ACTION: Edit **syscall.c** to add a line for the new system calls (two places each). Look at how other system calls are declared and follow the pattern.
- ACTION: Add a variable for the global read counter.

FILE: **sysproc.c**

This is where we are defining all of the functions in kernel space. Here you will find how the other system calls are defined. Because the new system calls are very simple they may only one line. However, it still needs to be declared in this file.

- ACTION: Edit **sysproc.c** to add a function for the new system call. Do **not** overthink the function. You should look at the other system calls definitions to derive your solution.
- ACTION: Add the variable definition from **syscall.c** with an extern statement so it can be access from this file.

User-Level - Test Program

Implement and test the user level program to call the new program to call a series of system calls to display information about the current process.

FILE: **user/getreadcount.c**

Edit the Makefile to add your new user -level user program so that it will be compiled along with the rest of the user programs. This is very similar to how we added our “hello” user program. You only have to do this once for each new user program created. As before, you can build **xv6** as follows:

```
make clean
make qemu
```

Once working, the user program should look similar to the following:

```
$ getreadcount
Read Count: 47
$
```

The actual count displayed will vary between systems.

Summary

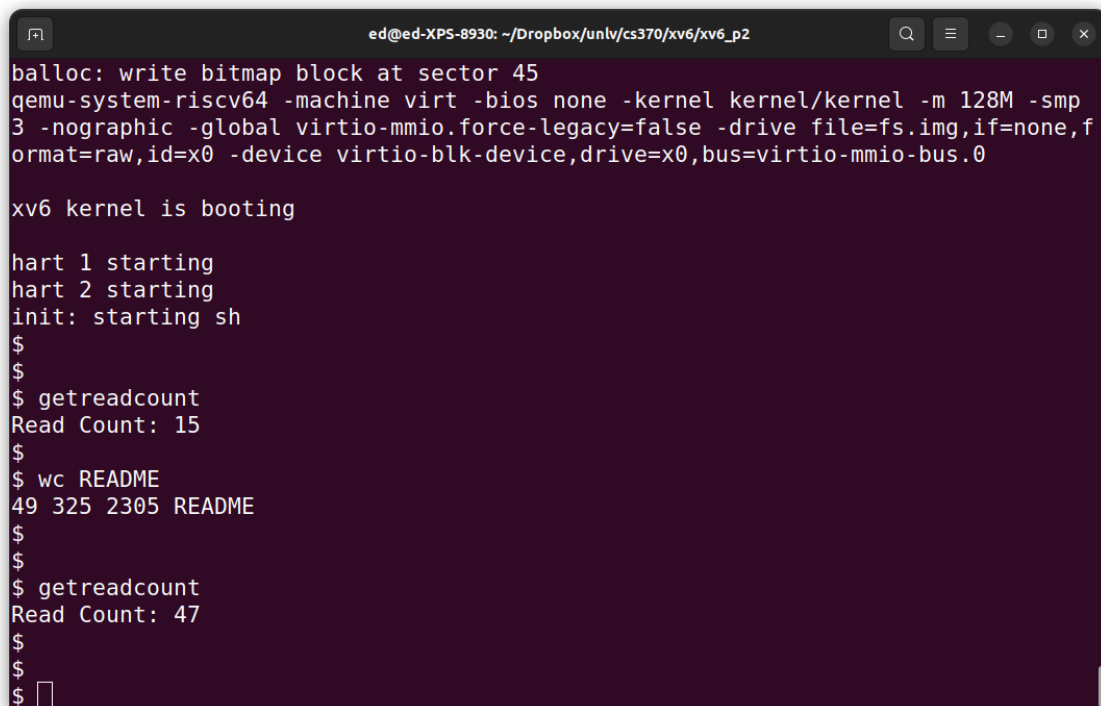
In order to define your own system call in **xv6**, you need to make changes to 5 files:

```
xv6/Makefile
xv6/user/user.h
xv6/user/usys.pl
xv6/kernel/syscall.h
xv6/kernel/syscall.c
xv6/kernel/sysproc.c
```

In addition, you will need to create the file **xv6/user/getreadcount.c**.

Example:

Below is an example of the output of the **getreadcount** test program.



```
ed@ed-XPS-8930: ~/Dropbox/unlv/cs370/xv6/xv6_p2
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
$
$ getreadcount
Read Count: 15
$
$ wc README
49 325 2305 README
$
$
$ getreadcount
Read Count: 47
$
$
$
```