

CS 370 – Project #1, Part B

Purpose: Become familiar with the **xv6** Teaching Operating System, shell organization, and system calls
Points: 50

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.

- Complete Project #1, Part A
 - Install an Ubuntu development environment
 - Become familiar with the **xv6** Teaching Operating System
- Implement a simplified version of the UNIX `time`¹ command (reporting ticks²)
 - *Note*, to make it easier, this time call will provide a program execution time in ticks instead of real, user and sys times. See Example Submission section for an example.

Submission

When complete, submit:

- A screen shot of the time program output (PNG file) via the class web page (assignment submission link) by class time.
 - *Note*, In Ubuntu there is a pre-installed program named screenshot that will be useful to capture the terminal window.

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

¹ For more information, refer to: [https://en.wikipedia.org/wiki/Time_\(Unix\)](https://en.wikipedia.org/wiki/Time_(Unix))

² A tick is a fairly arbitrary unit of time in xv6, determined by how often a hardware timer generates interrupts.

Time System Call

A system call is how a program requests service from an operating system's kernel. System calls may include hardware related services, creation and execution of new processes and process scheduling. It is also possible for a system call to be made only from user space processes.

In this assignment you will implement a system call to get the current time, and a simplified version of the UNIX time command. This command measures how long it takes for some other command to execute. For example, if we run `time ls` the time command will run `ls`, then when that's done tells us how long it took to execute. Usually this is measured in seconds, but for this project time is measured in ticks (100 ticks equals to one second of real time).

The objective is not to implement a very difficult system call, but to become familiar with **xv6**'s files and environment.

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 time system call. Hint; Look at how other system calls are declared and follow the pattern that best suits the time 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 time system call. Hint: Look at how other system calls are declared and follow the pattern.

FILE: **user/time.c**

This file is the implementation of the simplified time command from bash. This is in user space and therefore is written in C. Implementing the program requires you to use the system call added in the earlier section and other **xv6** system calls, in particular **fork()**, **exec()**, and **wait()**. You can find a detailed explanation of these in Chapter 1 Operating System Interfaces of the **xv6** textbook.

ACTION: You must create a new file called **time.c** and create a program that will:

- Accepts arguments from the command line interface. You are expected to do the necessary error checking with appropriate error messages.
- Fork a child process. The parent process is responsible for measuring the time it takes for the child to finish executing. A outline of the step involved is as follows:
 - Get the current time using *uptime*.
 - Fork() and then wait for the child process to terminate.
 - When the wait() call returns to the parent process get the current time again and calculate the difference.
- You should return an error message if the fork() is unsuccessful.
- The child process is responsible for:
 - Executing the command as per the user provided command line arguments via the exec() call.
 - You must determine how you will pass the arguments from the command line.
 - The child should return an error message if execution fails.

Hint: Code **time.c** after you've finished making the system call accessible in the kernel and editing the **Makefile**. This way you can test and debug as you program.

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 time system call.

FILE: **syscall.c**

This file has two sections where the system call must be added. Starting at line 86, 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 time system call. Look at how other system calls are declared and follow the pattern.

The second part is an array of function pointers. A function pointer in C, is when you can pass a name of a function as an argument to another function. The array is indexed using the values defined in **syscall.h**. Each element of the array points to a function, which is invoked.

- ACTION: Edit **syscall.c** to add a line in the function pointers array for the time system call. Look at how other system calls are declared and follow the pattern. Before you add the line for the time system call, notice that the last system call has a comma. That is intentional and you must follow the pattern when you add the time system call.

At the end of the array of function pointers there is the how **xv6** implements system calls.

```
132 void
133 syscall(void)
134 {
135     int num;
136     struct proc *p = myproc();
137
138     num = p->trapframe->a7;
139     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140         p->trapframe->a0 = syscalls[num]();
141     } else {
142         printf("%d %s: unknown sys call %d\n",
143             p->pid, p->name, num);
144         p->trapframe->a0 = -1;
145     }
146 }
```

A brief walk-through of the function:

1. Creates a variable called **p** which is the current **RUNNING** process (Line 136).
2. Get the number of the system call from register **a7** (Line 138).
 - a. The number was stored in **a7** when the system call declared in **usys.pl** got executed.
3. Then some basic error checking (Line 139):
 - a. Checks if the number is greater than 0;
 - b. Checks if the number less than the total number of system call in the system;
 - c. Checks if the system call we are calling is not null.
4. If the system call passes, it calls that function and then the function will return a value in **a0**. In **xv6** the convention is to return the value of a function in the **a0** register (Line 140).
5. If it fails, it returns an error and returns a -1 into **a0** (Line 144).

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 time system call is very simple, it actually does not need to get any information from the process. However, it still needs to be declared in this file.

- ACTION: Edit **sysproc.c** to add a function for the time system call. Do **not** overthink this function. You should look at the other system call definitions to derive your solution.

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 create the file **xv6/user/time.c**.

Compiling Your New System Call

Edit the **Makefile** to add your time 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. *Note*, the location you place the command, time in this example, is the order it will be displayed. 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
```

Testing Time System Call

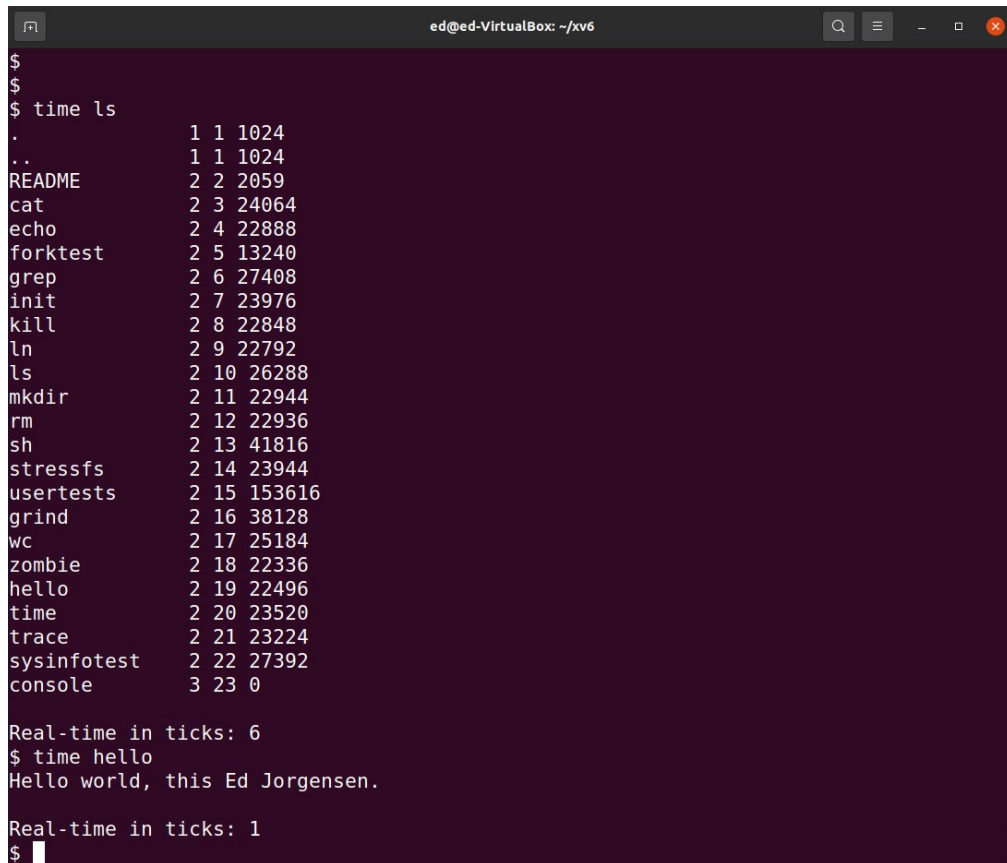
Once you are done with your program, you can test it and execute **make qemu**. Type in a simple **time ls** command and see if the system call is working as intended.

```
$ time ls
.          1 1 1024
..         1 1 1024
README    2 2 2059
cat        2 3 24008
echo       2 4 22832
forktest   2 5 13184
grep       2 6 27352
init       2 7 23920
kill       2 8 22808
ln         2 9 22752
ls         2 10 26232
mkdir      2 11 22904
rm         2 12 22888
sh         2 13 41760
stressfs   2 14 23904
usertests  2 15 153568
grind      2 16 38064
wc         2 17 25144
zombie     2 18 22296
time       2 19 25160
trace      2 20 23192
sysinfotest 2 21 27336
console    3 22 0
Real-time in ticks: 12
$
```

Example Submission

The final submission should include a capture of current window (via built-in screenshot utility “Grab current window”).

As noted, your name should be included in the hello world output. The following is an example of the PNG file that should be submitted.



```
ed@ed-VirtualBox: ~/xv6
$
$
$ time ls
.          1 1 1024
..         1 1 1024
README    2 2 2059
cat       2 3 24064
echo      2 4 22888
forktest  2 5 13240
grep      2 6 27408
init      2 7 23976
kill      2 8 22848
ln        2 9 22792
ls        2 10 26288
mkdir     2 11 22944
rm        2 12 22936
sh        2 13 41816
stressfs  2 14 23944
usertests 2 15 153616
grind     2 16 38128
wc        2 17 25184
zombie    2 18 22336
hello     2 19 22496
time      2 20 23520
trace     2 21 23224
sysinfotest 2 22 27392
console   3 23 0

Real-time in ticks: 6
$ time hello
Hello world, this Ed Jorgensen.

Real-time in ticks: 1
$
```