Operating System
CS 370
Spring 2019

Project 2
Time System Call

Due Date: 11:59 PM February 11th, Monday 2018
Must be submitted before deadline

Total Points: 50

Graduate Assistant: Pradip Maharjan
Office Hours: MoWe 12 PM - 2 PM
Teaching Assistant: Luis Maya-Aranda
Office Hours: MoWe: 1 PM - 4 PM
TuTh: 10 AM - 12 PM

Course Web Page: http://osserver.cs.unlv.edu/moodle/

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

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. Although **xv6** lacks the full functionality of a modern OS, it is based on Sixth Edition Unix (also called V6) and is very similar in design and structure.

Resources:

The following resources provide more in-depth information regarding **xv6**. They include the **xv6** reference book, source code (pdf), and a tutorial on running and debugging.

1. **xv6** Reference Book:	https://pdos.csail.mit.edu/6.828/2016/xv6/book-rev9.pdf
2. **xv6** Source Code PDF:	https://pdos.csail.mit.edu/6.828/2016/xv6/xv6-rev9.pdf
3. Running and Debugging Tutorial:	http://zoo.cs.yale.edu/classes/cs422/2010/lec/l2-hw

Project:

Complete the following steps.

- Implement basic **time** system call
    - See time instructions
- Implement time system call with a **-s** parameter (to show time in seconds)

Submission

When complete, submit:

- Zip the entire **xv6** folder and submit (not qemu).
- A copy of the ***zipped xv6 folder*** via the class web page (assignment submission link) by 11:59 pm 11th February, 2019.

We will check for code similarities using MOSS. Please do not download code from the internet or distribute solutions anywhere including Github, Bitbucket and any other public code publishing platforms.

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

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 simplified version of the UNIX time command. This command measures how long it takes for some other command 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 of this assignment is not to implement a very difficult system call, but to become familiar with **xv6**'s files and environment.

Go through chapter 1 in xv6 reference book to have basic understanding of system calls in xv6.

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 other programs and it must be available in kernel space so has permission to access resources that prohibited from user mode.

<u>User Mode Files</u>
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.

1. **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.)

☐ TO DO: 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.

2. **usys.S**

This is an assembly file. This is where the system call is implemented. It is a very simple file and you will see that there is one macro declared:

```
01   #define SYSCALL(name) \
02     .globl name; \
03     name: \
04       movl $SYS_ ## name, %eax; \
05       int $T_SYSCALL; \
06       ret
```

A brief walk-through of the macro:

Line 01: Macro definition, it takes one argument called name.
Line 02: The argument is used to define a global variable with that name.
Line 03: A label with the name.
Line 04: Then you're going to move something, a number, into the register eax. This indicates what system call you are calling. The number being pushed into eax is the number that was declared in **syscall.h** that corresponds to the system call. Syscall.h will be discussed later.
Line 05: Then you will interrupt the kernel and pass a number, which indicates the corresponding interrupt is a system call.
Line 06: Then you will return.

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

☐ TO DO: Edit **usys.S** to add a line for the time system call.

Hint: Look at how other system calls are declared and follow the pattern.

3. **time.c**

This file is the program what will implement the time system call. This in user space and therefore is written in C. Implementing the time system requires you to use other **xv6** system calls, in particular uptime, fork, exec, and wait. You can find a detailed explanation of these in Chapter 0 of the **xv6** textbook.

You are responsible to 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 the corresponding error message.
- Fork a child process, the parent process is responsible to measure the time it takes for the child process to finish executing.
  - Get the current time using uptime.
  - Fork and then wait for the child process to terminate.
  - Then when wait returns in the parent process, get the current time again and calculate the difference.
- Return an error message if fork is unsuccessful.
- The child process is responsible for executing the command the user wishes to time using the exec command. You must determine how you will pass the arguments from the command line.
  - The child is responsible to return an error message is execution fails.

☐ TO DO: Create **time.c** and implement the time system call.

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

1. **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.

☐ TO DO: Edit **syscall.h** to add a line for the time system call.

Hint: Look at how other system calls are declared and follow the pattern.

2. **syscall.c**

This file has two sections where the system call must be added. Starting at line 80, is a portion where all the functions for the system calls are defined within the kernel.

☐ TO DO: Edit **syscall.c** to add a line for the time system call.

Hint: 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.

☐ TO DO: Edit **syscall.c** to add a line in the function pointers array for the time system call.

Hint: 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.

```
void
syscall(void)
{
  int num;
  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) &&
syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
```

A brief walk-through of the function:

- First it gets the number of the system call to from eax.
  - The number was stored in eax when the macro declared in usys.S got executed.
- Then some basic error checking:
  - Checks if the number is greater than 0;
  - Checks if the number less than the total number of system call in the system;
  - Checks if the system call we are calling it not null.
- If the system call passes, it call that function and then the function will return a value in eax. In **xv6** the convention is to return the value of a function in the eax register.
- If it fails, it returns an error and returns a -1 into eax.

3. **sysproc.c**

This is where we are defining all of the function 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.

☐ TO DO: Edit **sysproc.c** to add a function for the time system call.

Hint: Do not over think this function. It is an empty function and it returns an integer, very similar to an empty main function in C. You may look at the other system call definitions to derive your solution.

Compiling Your New System Call

Change the Makefile to add your time program so that it will be compiled along with the rest of the user programs the when you run the command **make qemu**.

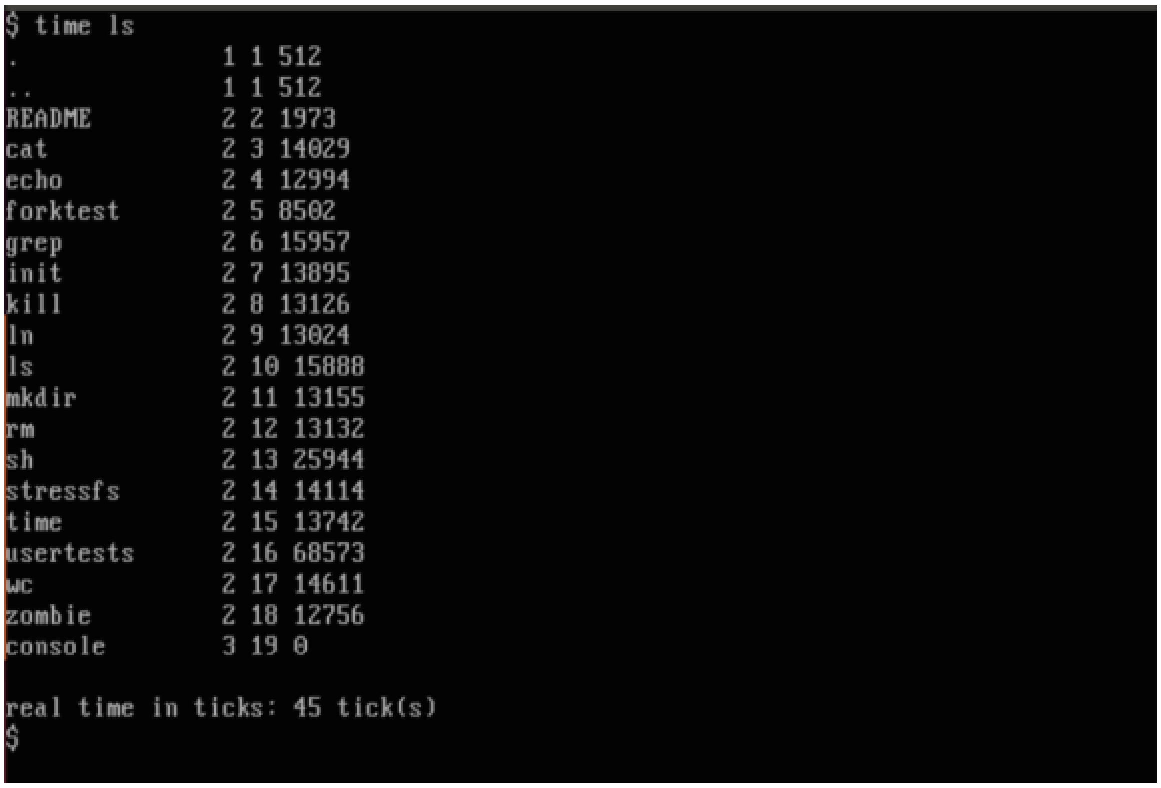
```
UPROGS=\
               ...
               _stressfs\
               _time\      //Add this line
               _usertests\
```

☐ TO DO: Edit **Makefile** to add the time program under UPROGS (line 159).

You only have to do this once for each new user program created.

Testing Time System Call

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



```
$ time ls
.              1 1 512
..             1 1 512
README         2 2 1973
cat            2 3 14029
echo           2 4 12994
forktest       2 5 8502
grep           2 6 15957
init           2 7 13895
kill           2 8 13126
ln             2 9 13024
ls             2 10 15888
mkdir          2 11 13155
rm             2 12 13132
sh             2 13 25944
stressfs       2 14 14114
time           2 15 13742
usertests      2 16 68573
wc             2 17 14611
zombie         2 18 12756
console        3 19 0

real time in ticks: 45 tick(s)
$
```

Further instructions

1. If command is "time ls", it should display in number of ticks
2. If command is "time -s ls", it should display time in seconds (floating point value). (float or double don't work in xv6).
   Hint: 100 ticks = 1 second

Note: List of xv6 files that need to be updated to add a system call in this assignment

1. user.h
2. usys.s
3. syscall.h
4. syscall.c
5. sysproc.c
6. makefile