CS 370 – Project #1

Purpose: Become familiar with the **xv6** Teaching Operating System, shell organization, and system calls
Points: 200

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, 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.

- Install and become familiar with **xv6** Teaching Operating System
  - See installation instructions
  - Focus on shell organization

- Implement basic system calls
  - Time system call
  - Trace system call
  - Sysinfo system call

Submission
When complete, submit:

- A copy of the ***zipped xv6 folder*** (not qemu) via the class web page (assignment submission link) by class time.

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

Installing xv6

The following instructions walk you through the process of setting up **xv6** within Ubuntu.

1. Download and Install VirtualBox
    a. Use default/recommended RAM during installation.
    b. IMPORTANT: Allocate at least 20GB of disk space to the machine during set-up. Set up Ubuntu in VirtualBox. If you use the default 8GB setting, you are more likely to run out of space at some point.
    c. IMPORTANT: Allocate as many CPUs as possible, before installing Ubuntu in VirtualBox the amount of CPUs can be changed.
    d. Use all other recommendations during installation.

2. Download Ubuntu 20.04.1 LTS 64-bit
    a. Do ***not*** install higher version of Ubuntu

3. Open a terminal and execute the following commands:
    a. `sudo apt-get update`
    b. `sudo apt-get upgrade`
    c. `sudo apt-get install git zip build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu`
    d. `git clone https://github.com/unlv-os/xv6.git xv6`
        Note, this link may change, if the clone does not succeed, check the most current repository from MIT's official xv6 website
    e. `chmod 700 -R xv6`

You are now ready to run QEMU and xv6. To run **xv6** on QEMU execute the following commands in the terminal:

1. **cd xv6** (if not already there)
2. **make qemu** (this is how you start xv6)

You should get a **$** prompt which is the xv6 prompt.

For example, in xv6, the **ls** command should show:

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2170
cat            2 3 13604
echo           2 4 12616
forktest       2 5 8044
grep           2 6 15480
init           2 7 13196
kill           2 8 12668
ln             2 9 12564
ls             2 10 14752
mkdir          2 11 12748
rm             2 12 12724
sh             2 13 23212
stressfs       2 14 13396
usertests      2 15 56328
wc             2 16 14144
zombie         2 17 12388
console        3 18 0
$
```

Note: To close xv6 use **ctrl-a x**

Using GDB:

At some point, you may need to run xv6 on QEMU with GDB. Open two terminals as follows.

In terminal 1, execute the following commands:

1. **`cd xv6`** *(if not already in the xv6 directory)*
2. **`make qemu-gdb`**

In terminal 2, execute the following commands:

1. **`cd xv6`** *(if not already in the xv6 directory)*
2. **`gdb -iex "set auto-load safe-path <path-to-xv6-folder>" kernel`**

Helpful Information:

When attempting to exit xv6, use **`ctrl-a x`**
To quit GDB: **`q`**
To terminate an action being executed in GDB: **`ctrl-c`**
If your mouse and/or keyboard becomes trapped in QEMU, typing **ctrl-alt** will exit the mouse grab.

If you are debugging and you make a change to the code, make sure to close both QEMU and GDB. Otherwise they will get out of sync and cause problems.

To find the exact path required for the gdb command you can run gdb kernel in the second terminal. The path will be listed in the error provided.

Inserting a User Program Within xv6

Overview

One of the first steps in navigating **xv6** is learning to insert a user level C program into the system. This will eventually allow you to test and implement more advanced features. These features include additional system calls such as ***time***. The following tutorial demonstrates how to add a basic user program and display it at the shell prompt.

Tutorial

1. Create a new C program (**`hello.c`**) in your chosen text editor. The contents of the program can be as simple as that shown in Figure 1.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[])
{
    printf("Hello world, this is a sample xv6 user program\n");
    exit(0);
}
```

Figure 1: Sample **xv6** User Program (**hello.c**)

2. Save the program within the **xv6/user** folder.

3. Open the **xv6** Makefile from **xv6** directory. The **Makefile** should be edited to ensure that **`hello.c`** is properly added to the existing set of user programs (under UPROGS) to be compiled. This is done as shown in Figure 2.

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_hello\
```

Figure 2: Add **`hello`** to the List of User Programs

4. Open the terminal to compile the system and start up **xv6** on QEMU. The commands should be the following:

```
cd xv6                          (note, your xv6 directory might be different)
make clean
make qemu
```

If there were no compilation errors **xv6** should have been properly launched something like as shown in Figure 4.

```
ing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o user/zombie.o
 user/zombie.c
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_zombie user/zombie.o user/ul
ib.o user/usys.o user/printf.o user/umalloc.o
riscv64-linux-gnu-objdump -S user/_zombie > user/zombie.asm
riscv64-linux-gnu-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/zombie.
sym
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestand
ing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o user/hello.o
user/hello.c
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_hello user/hello.o user/ulib
.o user/usys.o user/printf.o user/umalloc.o
riscv64-linux-gnu-objdump -S user/_hello > user/hello.asm
riscv64-linux-gnu-objdump -t user/_hello | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/hello.sy
m
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_l
n user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_z
ombie user/_hello
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
balloc: first 618 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive
file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
```

Figure 4: Compilation and Start-Up of **xv6**

5. Use the ls command to verify that **`hello.c`** was added to the existing list of available user programs as shown in Figure 5. You can then type **hello** to view the output of the newly created test program.

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2059
cat            2 3 23896
echo           2 4 22720
forktest       2 5 13080
grep           2 6 27248
init           2 7 23824
kill           2 8 22696
ln             2 9 22648
ls             2 10 26128
mkdir          2 11 22792
rm             2 12 22784
sh             2 13 41656
stressfs       2 14 23800
usertests      2 15 153472
grind          2 16 37968
wc             2 17 25032
zombie         2 18 22184
hello          2 19 22344
console        3 20 0
$
```

Figure 5: Successful Sample Program Output

Committing Changes

The files you will need for this and subsequent lab assignments are distributed using the Git version control system. To learn more about Git, take a look at the Git user's manual, or, you may find this CS-oriented overview of Git useful. Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the project exercises, and want to checkpoint your progress, you can commit your changes by running:

```
$ git add .
$ git commit -am 'my solution for hello xv6'
Created commit 60d2135: my solution for hello xv6
 1 files changed, 1 insertions(+), 0 deletions(-)
$
```

System Calls

To start the this part of the project, switch to the syscall branch (make sure you're in the xv6 folder):

```
git fetch
git checkout syscall
make clean
```

If you run **`make grade`**, you will see that the grading script cannot exec **`time`**, **`trace`** and **`sysinfotest`**. Your job is to add the necessary system calls and stubs to make them work.

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[1] 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.

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

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.

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.)

- 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.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.

- 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.

3. **`time.c`**

This file is the program that will implement the time system call. This is 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 1 Operating System Interfaces 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 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.

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.

- Edit **`syscall.h`** to add a line for the time system call.

2. **`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.

- 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.

- 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).

3. **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.

- Edit **sysproc.c** to add a function for the time system call. Do ***not*** overthink this function. It is an empty function and it returns an integer, very similar to an empty main function in C. You should look at the other system call definitions to derive your solution.

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
$
```

System call tracing

In this section you will add a system call tracing feature that may help you when debugging later labs. You'll create a new **trace** system call that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the fork system call, a program calls **`trace(1 << SYS_fork)`**, where **`SYS_fork`** is a syscall number from **`kernel/syscall.h`**.

You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments. The **`trace`** system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.

We have provided a **`trace`** user-level program that runs another program with tracing enabled (see **`user/trace.c`**).

When you're done, you should see output like this:

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$
$ grep hello README
$
```

In the first example above, trace invokes grep tracing just the read system call. The 32 is **`1<<SYS_read`**. In the second example, trace runs grep while tracing all system calls; the 2147583647 has all 31 low bits set. In the third example, the program isn't traced, so no trace output is printed. Your solution is correct if your program behaves as shown above (though the process IDs may be different).

Some hints:

- Add **$U/_trace** to **UPROGS** in **`Makefile`**
- Run **`make qemu`** and you will see that the compiler cannot compile **`user/trace.c`**, because the user-space stubs for the system call don't exist yet: add a prototype for the system call to **`user/user.h`**, a stub to **`user/usys.pl`**, and a syscall number to **`kernel/syscall.h`**. The **`Makefile`** invokes the perl script **`user/usys.pl`**, which produces **`user/usys.S`**, the actual system call stubs, which use the RISC-V ecall instruction to transition to the kernel. Once you fix the compilation issues, run **`trace 32 grep hello README`**; it will fail because you haven't implemented the system call in the kernel yet.

- Add a **sys_trace()** function in **kernel/sysproc.c** that implements the new system call by remembering its argument in a new variable in the proc structure (see **kernel/proc.h**). The functions to retrieve system call arguments from user space are in **kernel/syscall.c**, and you can see examples of their use in **kernel/sysproc.c**.
- Modify **fork()** (see **kernel/proc.c**) to copy the trace mask from the parent to the child process.
- Modify the **syscall()** function in **kernel/syscall.c** to print the trace output.
    - ***You will need to add an array of syscall names to index into.***

Sysinfo

sysinfo collects information about the running system. The system call takes one argument: a pointer to a struct sysinfo (see **kernel/sysinfo.h**). The kernel should fill out the fields of this struct: the **freemem** field should be set to the number of bytes of free memory, and the **nproc** field should be set to the number of processes whose state is not **UNUSED**. We provide a test program **sysinfotest**; you pass this part of the project if it prints "**sysinfotest: OK**".

Some hints:

- Add **$U/_sysinfotest** to **UPROGS** in **Makefile**
- Run **make qemu** and **user/sysinfotest.c** will fail to compile. Add the system call **sysinfo**, following the same steps as in the previous system calls. To declare the prototype for **sysinfo()** in **user/user.h** you need predeclare the existence of **struct sysinfo**:

```
struct sysinfo;

int sysinfo(struct sysinfo *);
```

- Once you fix the compilation issues, run **sysinfotest**; it will fail because you haven't implemented the system call in the kernel yet**.**
- The **sys_sysinfo()** function in **kernel/sysproc.c** needs to copy a **struct sysinfo** back to user space; see **sys_fstat()** (**kernel/sysfile.c**) and **filestat()** (**kernel/file.c**) for examples of how to do that using **copyout()**.
- To collect the amount of free memory, add a function to **kernel/kalloc.c**
- To collect the number of processes, add a function to **kernel/proc.c**

Grading and hand-in procedure

You can run **make grade** to test your solutions with the grading program. The TAs will use the same grading program to assign your project submission a grade.

After committing your final changes to the project, type **make zip** to compress your project and then submit the **project1.zip** via Canvas.