CS 370 – Project #4, Part A

Purpose: Become familiar with operating system process scheduling and process context switching through xv6

Points: 200

Introduction:

Any operating system is likely to run with more processes than the computer has CPUs, so a plan is needed to time-share the CPUs among the processes. Ideally the sharing should be transparent to user processes.

Our first step will be to implment some additional system calls to support OS scheduler updates in subsequent parts of this assignment.

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 actions.

- Update the fork() system call
- Implement **ps** system call
- Implement **psm** system program

Submission

When complete, submit:

- A screen shot of the ps program output (PNG file) via the class web page (assignment submission link) by class time.
  - The full source code will be submitted in a subsequent Part.

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

Project Setup:

Project #4 will be done entirely in **xv6**. To get started with this project first follow the instructions below:

```
1. git clone https://github.com/unlv-cs/xv6.git project4
2. cd project4
3. git fetch
4. git checkout scheduler-s22
5. make clean
```

Note: **xv6** may not compile initially as source code is missing that you will have to add throughout this project.

xv6 Scheduling Background:

Every process in **xv6** is stored in a Process Control Block (PCB) which is defined as **`struct proc`** (**`kernel/proc.h`**). An array of **`struct proc`** of size **`NPROC`** (64) is defined near the top in **`kernel/proc.c`**. This array keeps track of all **xv6** processes; at most 64 processes can be in **xv6** at any given time. Each entry in the array is a different process and unused entries are marked with the process state **`UNUSED`**. For security purposes, all functions that can access the array of processes are implemented in **`kernel/proc.c`**. No other file gets access to the processes array. If you need to access the processes array, you must write a function in **`kernel/proc.c`**. You may create a system call that calls a function in **`kernel/proc.c`** (see fork/wait/exit system calls).

In **xv6**, a basic Round-Robin (RR) scheduler is implemented by default (**`kernel/proc.c`**). The RR scheduler assigns time slices to each process in equal portions. The process is interrupted if it is not completed in a given time slice and sent back to the queue where it gets the next time slice after all other processes in the queue get their allocated time slice. The default time slice in **xv6** is 100 ticks.

Switching from one process to another involves saving the old process's CPU registers, and restoring the previously-saved registers of the new process; the fact that the stack pointer and program counter are saved and restored means that the CPU will switch stacks and switch what code it is executing; this is called a ***context switch***.

The ***context switch*** function performs the saves and restores for a kernel process switch. This function doesn't directly know about processes; it just saves and restores register sets, called **`contexts`**. When it is time for a process to give up the CPU, the process's kernel thread calls the context switch to save its own context and return to the scheduler context. Each context is contained in a **`struct context`** (**`kernel/proc.h`**), itself contained in a process's **`struct proc`** or a CPU's **`struct cpu`**. Context switch takes two arguments: **`struct context *old`** and **`struct context *new`**. It saves the current registers in old, loads registers from new, and returns.

Update Process Control Block (PCB)
Update the Process Control Block (PCB) to add an additional field for a prority (uint64).

Update fork() System Call:
Update the existing `fork()` system call to set the priority field to to 2 as new processes are created.

Create New Structure, (ps_proc)
You will need to create a new structure **`struct ps_proc`** in **`kernel/psinfo.h`** (which you will need to create) to hold the applicable fields (name, state, pid, memory, and priority). Each can be an inetger, except the name which must be char array of size 16. For the state, you can use an integer value (1=used, 2=sleeping, 3=runnable, 4=running, and 5=zombie). Additionally, define MAX_PROCS and set to 15.

PS (Process Status) System Call:
The process status (**`ps`**) system call will be used to obtain information from from the kernal about current processes in the **xv6** system. We will be obtaining the **`name`**, **`pid`**, **`state`**, **`memory`** used (in bytes), and **`priority`** of processes whose current state is one of the following: **`USED`**, **`SLEEPING`**, **`RUNNABLE`**, **`RUNNING`** or **`ZOMBIE`**.

Your implementation of the **`ps`** system call will be located in **`kernel/proc.c`**. You will create a local array of of **`struct ps_proc`** (in **`kernel/psinfo.h`**). The array should be of size MAX_PROCS. You should initialze the values as appropriate (based on type). We will populate this array with applicable entries. To obtain the applicable entries, you should loop through the every process in the process table (NPROC entries) and return any processes that are ***not*** marked as UNUSED. The applicable fields (name, state, pid, memory, and priority) should be copied from the process control block to the next available entry in local the **`ps_proc`** array. When a valid entry is found, (i.e., an entry not marked as UNUSED), a lock should be obtained before copying information from the entry to ensure it does not change while we are copying the information. Once done, the lock should be released. The address of the user-space array of **`struct ps_proc`** (in **`kernel/psinfo.h`**) is passed from a user-level program to kernel space. We will **`copyout`** to correctly access/modify memory between user and kernel space which will copy data from the local array to the user-space array. If the copy fails, a -1 should be returned (indicating an error). You can use **`grep`** to find other examples using **`copyout`** correctly.

Since we are modifying **`proc.c`**, we will need to add **`ps`** there. Refer to the existing examples to see what must be done. Additionally, you will need to add **`ps`** to the **`sysproc.c`** file which will handle the argument address (using **`argaddr`**). You can find other examples in the **`sysproc.c`** file to see how this is done (i.e., **`wait()`**).

Apart from this, you will need to add the necessary stubs to fully implement the system call (similar to Project #1). This will include the **`user.h`**, **`usys.pl`**, **`syscall.h`**, and **`syscall.c`** files.

PSM (Process Status with Memory) System Program:

The process status (`psm`) system program will use the new `ps` system call and display information about current processes in the **xv6** system. We will be displaying the formatted `name`, `pid`, `state`, `memory` used (in bytes), and `priority` of processes.

Your implementation of `psm` system program will in user space. You will need to create and pass the address of an array of `struct ps_proc` (from `kernel/psinfo.h`) from our user-level program `psm` (`user/ps.c`) in order to pass information kernel space back to the user space system program.

You should call the `ps` system call (with the array) and verify there is no error. If there is no error, the `psm` system program should display a header as follows: "`Process Status (version: <yourFirstName>)`" which should include your first name (not the characters "youtFirstName"). The next line should be dashes, followed by the formatted outout; pid, state, priority, memory usage, and process name. The tab (\t) is used for alignment. See the example output for required formatting.

Modify the `Makefile` as necessary (also similar to Project #1).

Summary of Updates Files

As described, you will be either updating or creating the following files:

```
Makefile
user/user.h
user/usys.pl
kernel/syscall.h
kernel/syscall.c
kernel/sysproc.c
kernel/defs.h
kernel/psinfo.h
user/psm.c
```

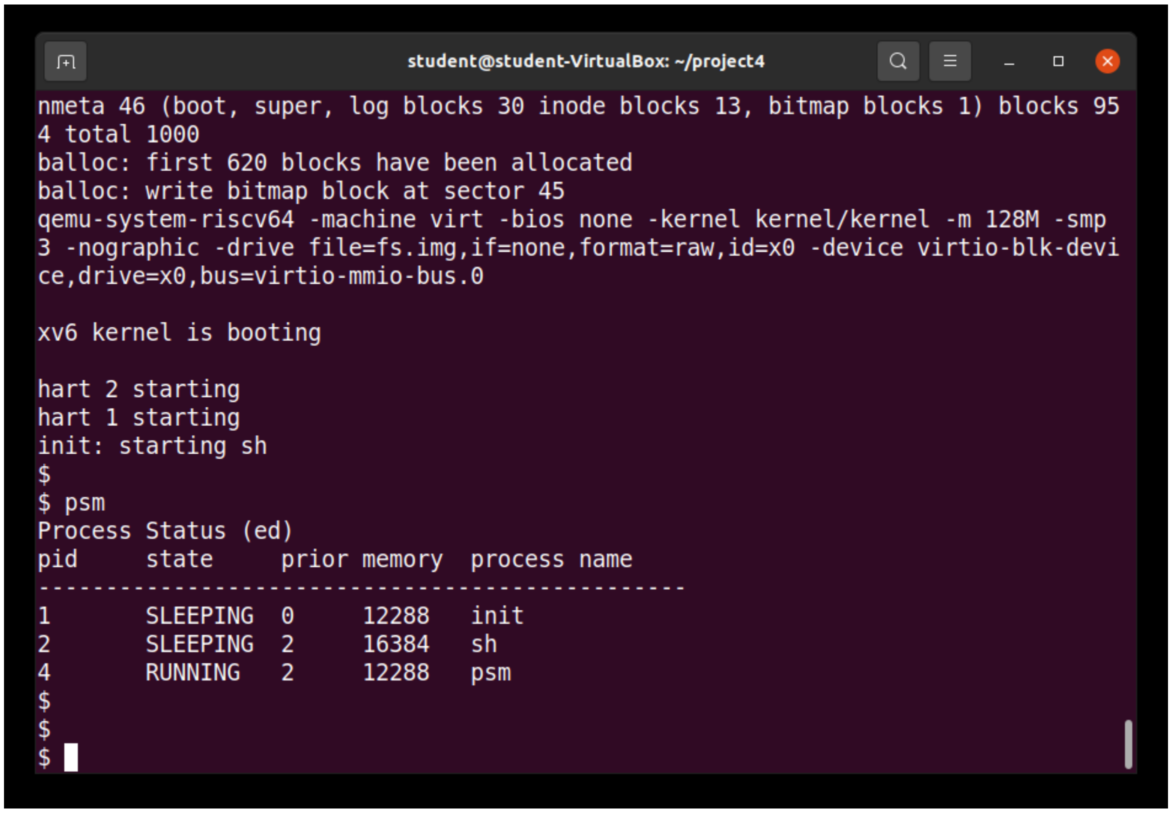
Pay close atteniton to user space vs kernel space.

Example Output

An example of the output is shown below.

```
$ psm
Process Status (ed)
pid     state     prior memory  process name
------------------------------------------------
1       SLEEPING  0     12288   init
2       SLEEPING  2     16384   sh
4       RUNNING   2     12288   psm
$
```

Here is a screen capture of the output for reference.



```
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 95
4 total 1000
balloc: first 620 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-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
$ psm
Process Status (ed)
pid     state     prior memory  process name
------------------------------------------------
1       SLEEPING  0     12288   init
2       SLEEPING  2     16384   sh
4       RUNNING   2     12288   psm
$
$
$
```