Alexander M. Nicoara (A20301259)
CS450: Operating Systems
Programming Assignment 2
Professor Leung

Implementing *procState()*

In part 2 of this assignment, I was tasked with the objective of implementing a new xv6 system call *procState()*. This call prints a list of processes and some of their attributes (such as name, state, id, and memory) to the screen. It is called by the command $ ps in the shell (user level), and takes no arguments. In my implementation, the program (ps.c) makes a call to ps(), which generates an interrupt using the `int n` instruction, where n represents the system call number. Since 21 system calls already exist, *procState()* will have the number 22. After the trap is handled (we've entered the kernel), the syscall function (in syscall.c) will run and use the number n from the %ea register. The syscall function will call the appropriate routine (based on the value of n), which in our case is 22, and will call the sys_ps() function in sysproc.c. In sys_ps(), the function ps() will be called from proc.c and will execute, printing the process information for each process to the screen and then returning, which will then propagate the return in sys_ps() and send us back to trap function (which called syscall) which will also return and then bring us back to user mode.

Code Changes/Modifications

To create my system call, I first carefully analyzed other system calls (including read) to see how declarations were made and learn the flow/direction of how system calls get called and executed.

In **proc.c**

- Added (starting from *line 536* until *line 561*)
    - The reason why I am implementing my system call function here is because it has access to ptable, which contains a list of process attributes for each process that we need to print. This function loops through the ptable and prints a line displaying information (process name, state, id, memory) for each process.

```
int ps()
{
  struct proc *pr;
  sti(); // enable hardware interrupts

  acquire(&ptable.lock);
  cprintf("Name      State      ID      Memory \n"); // cprintf only prints %s, %d, %x, %p
  for(pr = ptable.proc; pr < &ptable.proc[NPROC]; pr++){
    // convert bytes to kilobytes
    int proc_mem = (int) (pr->sz);
    int memkb = proc_mem/1000; // part in front of decimal
    int mdec = proc_mem%1000; // remainder

    // print out a process state that matches the enum along with PID, Name, Memory in KB
    if(pr->state == SLEEPING){
      cprintf("%s  |   Sleeping    |  %d  |  %d.%d KBytes \n", pr->name, pr->pid, memkb, mdec);
    } else if(pr->state == RUNNING){
      cprintf("%s  |   Running     |  %d  |  %d.%d KBytes \n", pr->name, pr->pid, memkb, mdec);
    } else if(pr->state == RUNNABLE){
      cprintf("%s  |   Runnable    |  %d  |  %d.%d KBytes \n", pr->name, pr->pid, memkb, mdec);
    } else if(pr->state == ZOMBIE){
      cprintf("%s  |   Zombie      |  %d  |  %d.%d KBytes \n", pr->name, pr->pid, memkb, mdec);
    }
  }

  release(&ptable.lock);
  return 22;
}
```

Created a test program **ps.c**

- Added
  - Before adding the system call itself, I created a test program to make a call to the *ps()* function inside of proc.c

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int
main(int argc, char *argv[])
{
  ps();
  exit();
}
```

In **syscall.h**

- Added at *line 23*
  - We define our system call and assign an integer value (22), which represents the identification number which the kernel uses to know what call to make.

```
#define SYS_ps 22
```

In **defs.h**

- Added at *line 123*
  - Declaration of ps()

```
int ps(void);
```

In **user.h**

- Added at *line 26*
  - Declaration of ps()

```
int ps(void);
```

In **sysproc.c**

- Added at *line 93*
  - When sys_ps is called, execute ps() from proc.c and return 22 (the id of the sys_ps system call). Also display a comment (that you can uncomment) to show that the function was called when debugging/testing.

```
int
sys_ps(void)
{
   // cprintf("SYSPROC.C - inside sys_ps()\n");
   return ps();
}
```

In **usys.S**

- Added at *line 32*
  - Assembly declaration of ps

```
SYSCALL(ps)
```

In **syscall.c**

- Added at *line 106*
  - Extern allows sys_ps function to be visible from other files.

```
extern int sys_ps(void);
```

- Added (below) at *line 130*, moved closing bracket (originally line 130) down a line.
    - This associates our system call id to be associated with the function sys_ps in sysproc.c and indexed at 22 in the IDT.

```
[SYS_ps] sys_ps,
```

- Added (below) at *line 141*, moved everything below down a line.
    - A comment that can be uncommented to print whenever *syscall()* is called, this shows that a system call is being called.

```
// cprintf("SYSCALL.C - inside syscall()\n");
```

In **trap.c**

- Added (below) at *line 43*, moved everything below down a line.
    - A comment that can be uncommented to print whenever *trap()* is called, this shows that the trap handler works and is able to invoke a system call.

```
// cprintf("TRAP.C - inside trap()\n");
```

In **Makefile**

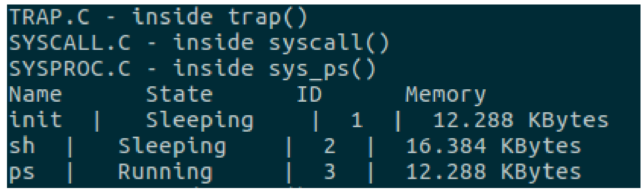
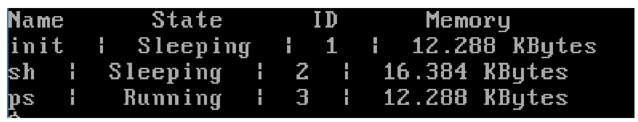
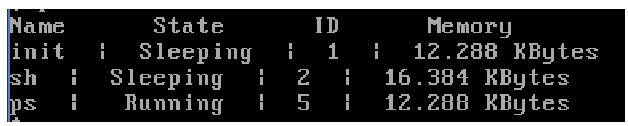
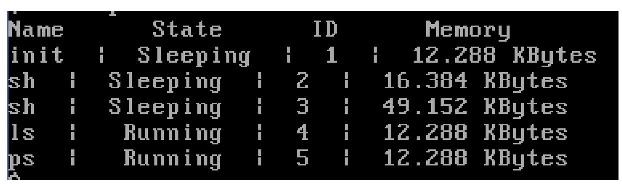
- Added at *line 176*, shifted below lines down by 1 line.

```
_ps\
```

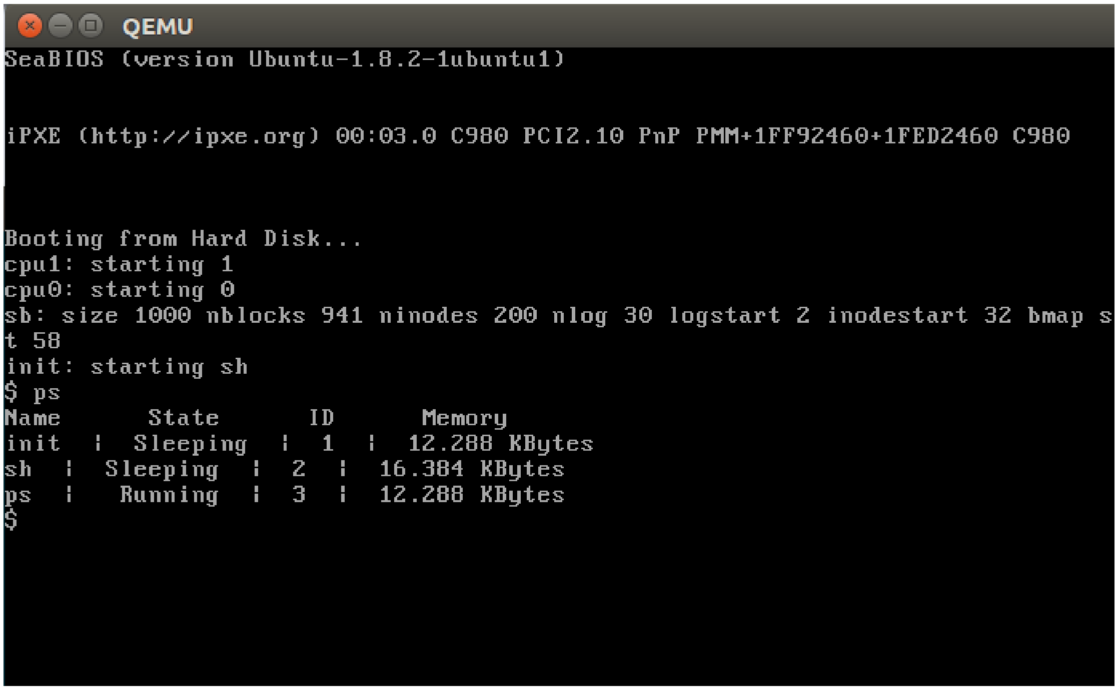
- Edited line 276, adding ps.c (the highlight represents the addition).

```
ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c ps.c zombie.c\
```

Test Cases

Input/Expected Result	Result
First do: syscall.c – uncomment line #141 sysproc.c – uncomment line #96 trap.c – uncomment line #43 Run: $ ps Expectation: We get print statements showing that some functions are being called up to the execution of *ps()* – This proves that my implementation is a system call.	
Run: $ ps Expectation: ps() runs and prints a list of all the processes and their attributes (name, state, id, memory)	
Open a fresh xv6 qemu window and run: $ ls then $ echo A then $ ps Expectation: The PID of **ps** is 5 (since it was 5th in sequence) and we won't see ls or echo because they terminated after execution.	
Run: $ ls | ps Expectation: We can see **ls** in the process table with the other processes, and **ls** is currently running. The pipe is not performing parallel execution, I used this to make the **ls** command execute before **ps**.	

Final Result


QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF92460+1FED2460 C980
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap s
t 58
init: starting sh
$ ps
Name State ID Memory
init ¦ Sleeping ¦ 1 ¦ 12.288 KBytes
sh ¦ Sleeping ¦ 2 ¦ 16.384 KBytes
ps ¦ Running ¦ 3 ¦ 12.288 KBytes
$