Mini-project 2 (baseline) – Fall 2025

Part A: Implementing the ps command in xv6

Part B: Evaluating the fairness of Round Robin in XV6

Submission deadline: Tuesday, October 14, at 11:59 pm.
Presentation: Tuesday, October 14, at class time.
Deliverables: zip file with xv6, and a report (follow the same format as mini project1).

Here, I am giving you the more important elements. Your job is to work with the details (externing functions, passing variables from one file to another by updating defs.h, checking the header files needed, etc.).

Part A: Warming up: Implementing the command ps in XV6

Description:

Adding a missing piece of XV6. Every Linux distribution includes the command **ps** which can tell you the current running process in the system.

If you type **man ps** in your Linux terminal you will obtain a description of the command and all the options about using it.

```
[ljaimes@ember ~]$ man ps
PS(1)                                                  User Commands

NAME
       ps - report a snapshot of the current processes.

SYNOPSIS
       ps [options]

DESCRIPTION
       ps displays information about a selection of the active processes.  If you want a repetitive update of the selection and the displayed information, use top(1) instead.

       This version of ps accepts several kinds of options:

       1   UNIX options, which may be grouped and must be preceded by a dash.
       2   BSD options, which may be grouped and must not be used with a dash.
       3   GNU long options, which are preceded by two dashes.
```

```
EXAMPLES
       To see every process on the system using standard syntax:
          ps -e
          ps -ef
          ps -eF
          ps -ely

       To see every process on the system using BSD syntax:
          ps ax
          ps axu

       To print a process tree:
          ps -ejH
          ps axjf

       To get info about threads:
          ps -eLf
          ps axms
```

```
[ljaimes@ember ~]$ ps
  PID TTY          TIME CMD
11614 pts/0    00:00:00 bash
11661 pts/0    00:00:00 ps
[ljaimes@ember ~]$
```

Objective:

To modify the `xv6` kernel to access and print the details of various process-specific variables such as process ID, state, priority, memory size, and other relevant information. This project will help students understand process management and manipulation in an OS, as well as how system calls work in `xv6`.

Key Concepts Covered:

- Process control in `xv6`
- Kernel modifications
- System calls
- Process table (`proc` array)
- C programming and OS internals

Methodology:

Step 1: Understanding the Process Table (`proc` structure)

In the 2012 version of xv6, the proc structure can be found in kernel/proc.h. This structure holds information about all processes, including:

- `pid`: Process ID
- `state`: Process state
- `sz`: Memory size of the process
- `name`: Name of the process
- `parent`: Pointer to the parent process
- `kstack`: Kernel stack of the process

These fields are useful for retrieving and printing process-related information.

Because here we need a bigger modification of the kernel, we are going to split the implementation of the system call into two components: implementation and prototype.

The implementation will take place in kernel/proc.c and the prototype in kernel/sysproc

Here, the implementation of **ps** in kernel/proc.c (further explanations in class)

```
249 int ps(void)
250 {
251   struct proc *p;
252   char *state;
253
254   cprintf("PID\tState\t\tMemory Size\tProcess Name\n");
255
256   acquire(&ptable.lock);
257   for(p=ptable.proc; p < &ptable.proc[NPROC]; p++) {
258          if(p->state == UNUSED) {
259                  continue;
260             }
261
262        if(p->state==SLEEPING) {
263                state="SLEEPING";
264        } else if (p->state==RUNNING){
265                state="RUNNING";
266        } else if (p->state==ZOMBIE){
267                state="ZOMBIE";
268        } else {
269                state="OTHER";
270        }
271    cprintf("PID: %d\tState: %s\tMemory Size: %d\tProcess Name: %s\n", p->pid, state, p->sz, p->name
);
272   }
273   release(&ptable.lock);
274
275
276  return 0;
```

Figure 1. ps() system call

Here in line 251 create a pointer that can point to elements of the structure proc. The proc structure is defined in kernel/proc.h, the structure proc contains the variables of a process, including pid, state, name, etc. In line 257 we loop on the elements of ptable. ptable is defined at the beginning of kernel/proc.c and corresponds to an array to store all the processes of the system, this array has NPROC slots, NPROC constant is defined in param.h. This loop will print any process whose state is different of unused

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

Figure 2 *ptable* (already made in kernel/proc.c)

Implementing the prototype in kernel/sysproc.c

```
int
sys_ps(void)
{
  return ps();
}
```

Because you want to use ps() in sysproc.c, but you implemented in proc.c you have to register ps(void) in defs.h you can use (similar than in miniproject 1 with counterB, and C)

Tester: Here is the tester, call it user/pstester.c, don't forget to register the tester in makefile.mk

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

int main(void){
        ps();
        exit();
}
```

Expected output

```
$ pstester
PID     State           Memory Size     Process Name
PID: 1  State: SLEEPING Memory Size: 8192       Process Name: init
PID: 2  State: SLEEPING Memory Size: 12288      Process Name: sh
PID: 3  State: RUNNING  Memory Size: 8192       Process Name: pstester
```

Remember, everything here is based on system calls, follow the table provided in Mini project 1 to register your system call. Don't forget to extern your system call in syscall.c

Part B: Experiment

Here, we will create an experiment. We would like to know how many times each process is running on the CPU. For this purpose, we will create five children (5 processes), and we will measure the number of times each process is scheduled to run on the CPU. Given that the scheduler is round-robin. We expect that all the processes will run a similar number of times on the CPU. Because, we already know of to get the information for each process in Part A, all that we need to do is to store this information in a table.

Here are the elements of the experiment.

1. A table is given to you to store the information of the five processes.

In use	pid	ticks	size

The same table is in code (below). Call the table pstat.h and place in include/pstat.h

```
#ifndef _PSTAT_H_
#define _PSTAT_H_

#include "param.h"

struct pstat {
        int inuse[NPROC];   // whether this slot of the process table is in use (1 or 0)
        int pid[NPROC];     // the PID of each process
        int ticks[NPROC];   // the number of ticks each process has accumulated
        int size[NPROC]; // number of bytes of the process
};

#endif  // _PSTAT_H_
```

The time and order for processing running in the CPU is determined by the scheduler.

Steps

1. This step is crucial
   a. Place the pstat.h in your *include* folder.

Re compile the kernel to check if everything works well.

   b. Use this two testers : childrenTester.c

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

int main(int argc, char *argv[])
{
  int i;
  int rc;



  printf(1, "Parent PID: %d\n", getpid());
  for (i = 1; i <= 3; i++)
  {
    rc = fork();
    if (!rc)
    {
      //Child process
      printf(1,
             "Child %d PID: %d\n",
             i,
             getpid());
      for(;;){}
    }
  }
  exit();
}
```

acccessTable.c

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

int main(int argc, char *argv[])
{

  struct pstat mypointer;

  mypointer.inuse[0]=5;
  printf(1, "Use: %d\n", mypointer.inuse[0]);


  exit();
}
```

Make sure you update defs.h to be able to use the pstat table

```
#ifndef _DEFS_H_
#define _DEFS_H_

struct buf;
struct context;
struct file;
struct inode;
struct pipe;
struct proc;
struct spinlock;
struct stat;
struct pstat;
```

LETS PLAY WITH THIS BEFORE CONTINUE

2. you will modify the process structure (kernel/proc.h) and add an extra feature, the new feature, call this new feature numTicks, this variable will count the number of time a process is schedule in the CPU. As shown in the next figure

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  int numTicks;                  // Counts the number of times a process has been scheduled.
};
```

Kernel/proc.h struct proc

3. You will need to update the variable numTicks for each process every time that process is schedule in the CPU (works as counter for each process). Let's start by set numTickts = 0 for every process in the system. Do that in kernel/proc.c, by adding p->numTicks = 0 in function allocproc(void)

```
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->numTicks = 0;
  release(&ptable.lock);
```

4. Update numTicks every time a process is scheduled in the CPU. To do that, check the scheduler function in kernel/proc.c and update the variable numTicks in the function void scheduler (void) as shown in the next figure.

```
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;

      p->numTicks += 1; //update the number of time schedule in cpu

      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

5. Create a system call to store the information of the five processes in the table. The name of the system call is getpinfo in kernel/proc.c. Here, the code

```
int getpinfo(struct pstat* pInfo)
{
        struct proc *p;
        int i = 0;
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state == ZOMBIE || p->state == EMBRYO){
                        continue;
                }
                if(p->state == UNUSED){
                        pInfo->inuse[i] = 0;
                }      PROCESS VARIABLES
                else{
                        pInfo->inuse[i] = 1;
TABLE PSTAT     }
                pInfo->pid[i] = p->pid;      PROCESS VARIABLES
                pInfo->ticks[i] = p-> ?;
                pInfo->size[i] = p-> ?
                i++;
        }
        release(&ptable.lock);
        return 0;
}
```

6. The prototype the system calls in kernel/sysproc.c

```
int sys_getpinfo(void)
{
        struct pstat *pInfo;
        if(argptr(0, (void *)&pInfo, sizeof(*pInfo)) < 0){
                return -1;
        }
        if(pInfo == NULL){
                return -1;
        }
        getpinfo(pInfo);                                //
        return 0;
}
```

7. Create a tester (childrentester.c) for creating the 3 or 4 children process.

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

int main(int argc, char *argv[])
{
  int i;
  int rc;
  printf(1, "Parent PID: %d\n", getpid());
  for (i = 1; i <= 3; i++)
  {
    rc = fork(); //create 3 children
    if (!rc)
    {
      //Child process
      printf(1, "Child %d PID: %d\n", i, getpid());
      for(;;){}
    }
  }
  exit();
}
```

8. Create a tester (store.c) for storing the information of each child (in-use, pid, ticks, size) in the table. This tester just calls the system call getpinfo. Don't forget to pass in a in the command line store a

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "pstat.h"

//Outputs results of getpinfo in a readable way.
//Options:
//a: Abbreviate. Do not print unused processes.
int main(int argc, char *argv[])
{
  int i;
  struct pstat proc_info;
  int abbreviate = 0;
  for (i = 0; i < argc; i++)
  {
    if (*argv[i] == 'a')
      abbreviate = 1;
  }
  getpinfo(&proc_info);
  for (i = 0; i < NPROC; i++)
  {
    if (abbreviate && !proc_info.inuse[i])
      continue;
    printf(1,
           "Use: %d   Size: %d   PID: %d   Ticks: %d\n",
           proc_info.inuse[i],
           proc_info.size[i],
           proc_info.pid[i],
           proc_info.ticks[i]);
  }
  exit();
}
```

Run the experiment by calling:

childrentester //to create the processes

```
$ childrentester
Parent PID: 3
Child 1 PID: 4
Child 2 PID: 5
Child 3 PID: 6
```

store a //to store and print the elements of the table pstat which contains the information of the processes.

```
store a
Use: 1   size: 8192   PID: 1   Ticks: 15
Use: 1   size: 12288   PID: 2   Ticks: 25
Use: 1   size: 8192   PID: 8   Ticks: 7
Use: 1   size: 8192   PID: 5   Ticks: 191
Use: 1   size: 8192   PID: 6   Ticks: 191
Use: 1   size: 8192   PID: 7   Ticks: 191
```

9. The final step collects the data manually from the store.c tester program output and creates a table in Excel to show the number of times each process was scheduled to run in the CPU tester. Here the ticks, and memory size columns are fake as well as the graph the graph too.

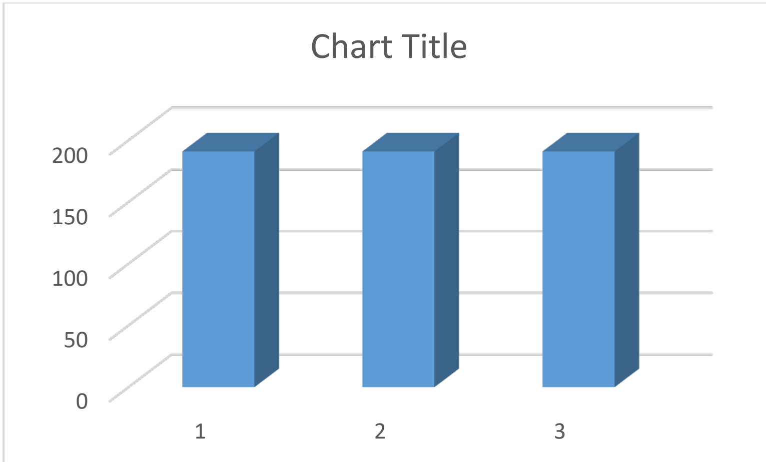
Chart Title
200
150
100
50
0
1
2
3