

# CS 1217 Spring 2023

## Lab 2

Total Points : 125

Due - Friday, March 31, 11:59 pm

### Lab Instructions

1. **Only one** person of a team (of 2) should submit the assignment. This should be the same person who had submitted assignments 1 and 2.
2. This assignment constitutes **6% of the total course grade**.
3. Please refer to the class webpage for notes on late assignment submission policy. ***No extensions, other than the ones allowed by class policy, will be granted.***
4. **Important:** The assignment submission should also contain a pdf file listing the contributions of individual team members.

### Instructions

1. **GitHub Classroom Link** - <https://classroom.github.com/a/4GIMCq6m>
2. GitHub Classroom will prompt you for fresh team names, **please use the following format: OLDTEAMNAME-1**
  - a. If your original team was called **xyz**, your new team name should be **xyz-1**
3. Please make sure that you are forming teams with the same team members as before, if there is a mishap on your end it will affect your grades

**Important note:** Make sure that this is cloned in a new folder.

To compile the project, `cd` into the `xv6-public` directory where you have cloned your project and execute `make qemu-nox` to compile xv6 and boot it within qemu. These are the same set of instructions that you have used for Assignment 3 earlier in the course. In addition, the challenge problem for adding a system call to the existing code base will also come in handy in this assignment.

In this Lab, you will learn about the CPU scheduler implementation in xv6. Once you have cloned the xv6 repository, start browsing through the code in `proc.h` and `proc.c`, where the bulk of the scheduler code is located. You'd be well advised to browse through the various data structure definitions in `proc.h`; understanding them and their usefulness will help carry out these exercises better.



### Question 1 [5+5+5+5]



Locate the various states that an xv6 process can be in. How many of them are there? Why are these many states needed? Finally, name the variable in the `proc` data structure that keeps track of the process state?



Hint : Look for enum definitions in `proc.h`, and the answer will reveal itself.

Once done, start browsing through the code for `proc.c`. It is highly recommended that you browse `proc.c` and understand how various states of the process are being used to make different kinds of decisions.

Aside: The scheduler code also has a number of things called locks that are acquired and released. For the purposes of this assignment, you do not have to worry about what they are, except for the fact that these are essential for making sure that a data structure is being modified by only one process at a time. We will discuss more on locks when we talk about synchronization and concurrency in the second half of this course.

Next, let us start looking at the scheduler code.

Line 323 in `proc.c` provides a definition of a function called `scheduler()`. Read through the code for the function and understand what is happening. Based on your understanding, answer the following questions.

### Question 2 [10+5+5]



Line 329 has a `for` loop that looks like an infinite loop. Explain why such a loop is required based on the code inside the `for` loop. Then, look at the code between lines 342 - 344. This will provide you some understanding of the default scheduling algorithm that is being implemented in xv6, which is one of the algorithms that we have discussed in class. Name the algorithm. Support your claims by pointing to the relevant sections of code, along with line numbers.

Hint: Almost all of what you are looking for is in `proc.c/proc.h`.

### Question 3 [5+5]

Next, look at the code for `fork()`, starting at line 181. What is the state of the newly created child process (as a result of the `fork()` system call) set to? Why is that the case?



### Question 4 [5]

Recall the concept of re-parenthood of child processes that we have talked about in class. In the public implementation of xv6, which is the process to which the **abandoned** child processes (i.e., the processes whose parents have exited) are assigned?

Hint: Return back to the process state enum structure and figure out which processes would fit the bill of “abandoned” and work your way backwards from there.

Next, we will move to the more interesting, implementation based parts of the assignment.

## Part 1: MLFQ Scheduler for xv6 [5+5+10+10]

*Implement a version of the Multi-Level Feedback Queue Scheduling algorithm as described below.*

One of the first things that you'll have to do is to implement two system calls as below. The utility of these function calls will be evident as you read through the description.

### Process Priorities and System Calls

The function prototypes are

```
int setpriority (int pid , int priority);  
int getpriority (int pid);
```



priority ranges from 0 to MAXPRIORITY.

The `setpriority()` system call will set the priority for an active process with PID `pid` to priority and resetting the budget to a default value (use something like a `#define DEFAULT_BUDGET` at the correct place in `proc.h`). Your code should return an error if the values for `pid` or `priority` are not correct. This means that the system call is required to enforce bounds for checking on the priority value; you should not assume that a user program only passes correct values.

Below is a simple way for a process to set its own priority:

```
myPriority = setpriority (getpid() , newPriority);
```

where `myPriority` is the return code from the system call and `newPriority` is an integer. If the value for any of the priorities is invalid, you should leave the original priority for the process unchanged and return an error. Likewise, the budget for a process should not be changed unless the priority is correct.


For the `getpriority()` system call, the return value is the priority of the process or -1 on error. `getpriority()` should return the priority for the process that matches the PID provided and is not in the `UNUSED` state. If there is no process with the provided PID, or the process with the provided PID is in the `UNUSED` state, it is an error.

Here is a simple way for a process to get its own priority:

```
myPriority = getpriority ( getpid( ));
```

where `myPriority` is the return code from the system call.

## Process Priority

You should `#define` a constant `MAXPRIORITY` at the correct location in `proc.h`. `MAXPRIORITY` is a number  $\geq 0$ . For each process, there should be a priority field in process' state, that will dictate the ready list to which the process belongs when it is in the `RUNNABLE` state. You will need to track a process' priority while it is **not** `RUNNABLE` so that you know where to put it if it later becomes `RUNNABLE` again. Upon allocation, each process will have the same initial (default) priority value, the highest priority. The process priority value can be changed during process execution via the `setpriority()` system call. 

## Periodic Priority Adjustment

In order to avoid starvation among processes, you will need to implement a promotion strategy. The strategy is to periodically increase the priority of all active processes by one priority level; that is, the priority of all processes in the `RUNNABLE`, `SLEEPING`, and `RUNNING` states will periodically be adjusted. You should use the following approach:

1. Add a new field to the `ptable` structure. Make it an unsigned integer (`uint`) and call it `PromoteAtTime`. The value stored will be the ticks value at which promotion will occur. This value is the same for all processes so you should put it in the `ptable` structure not each process. While there is an overflow issue here, you will ignore it for this project. You'll need to initialize `PromoteAtTime` to `ticks + TICKS_TO_PROMOTE` in `userinit()`, because the scheduler will expect `PromoteAtTime` to be set to some sane value as soon as it starts running.
2. Create a constant `#define TICKS_TO_PROMOTE XXX`, where `XXX` is the number of ticks that will elapse before all the priorities are adjusted. Each time that the routine `scheduler()` runs, check to see if it is time to adjust priorities.
3. When the value of ticks reaches, or exceeds, `PromoteAtTime`:
  - a. Adjust the priority value for active non-zombie processes to the next higher priority if not already at the highest priority
  - b. Change the priority queue for a process as appropriate. Put any adjusted processes on the back of the new queue. Do not move processes for which the priority was not adjusted
  - c. Set the value for `PromoteAtTime` to `ticks + TICKS_TO_PROMOTE`

## MLFQ to Implement

The algorithm for MLFQ will be a modified version of the MLFQ algorithm discussed in class. In this algorithm, you will utilize a time budget, rather than basing priority assignment on the fraction of a time slice used. Each time that the MLFQ algorithm runs, the process at the front of the highest priority non empty queue will be selected to run. This means that each time the algorithm looks for a new process, the algorithm must start by checking the highest priority queue and only checking a lower queue if no higher priority jobs are available.

Approach:

1. Each process is assigned its own budget. This budget must be initialized to the default value whenever a new process is allocated.
2. Each priority level has an associated FIFO queue. Each queue is serviced in a round robin fashion.
3. A newly created process is inserted at the end (tail) of the highest priority FIFO queue when it is moved from the `EMBRYO` to the `RUNNABLE` state.
4. At some point the process reaches the head of its queue and is assigned to a CPU. The system already records the time at which the process entered the CPU in the process structure.
5. If the process exits before the time slice expires, it leaves the system.
6. When a process is removed from the CPU (i.e. transitions out of the `RUNNING` state), the budget is updated according to this formula:  $\text{budget} = \text{budget} - (\text{time\_out} - \text{time\_in})$ 
  - a. If  $\text{budget} \leq 0$  then the process will be demoted and placed at the tail of the next lower priority queue when it again reaches the `RUNNABLE` state. The budget value will be reset to the default.
  - b. If the budget is not expired, the process will not be demoted and will be placed at the tail of the appropriate queue when it again reaches the `RUNNABLE` state.
7. Periodically a promotion timer will expire. The expiration of this timer will cause each process to be promoted one priority level. Promoted processes are placed at the tail of the new queue. On promotion, set the process budget to the default value.

## Part 2 : Lottery Scheduler for xv6 [10+10+10]

*Implement a version of the Lottery Scheduling algorithm as described below.*

The basic idea is simple: assign each running process a slice of the processor based on the number of tickets it has; the more tickets a process has, the more it runs. Each time slice, a randomized lottery determines the winner of the lottery; that winning process is the one that runs for that time slice.

You'll need two new system calls to implement this scheduler. The first is `int settickets(int number)`, which sets the number of tickets of the calling process. By default, each process should get one ticket; calling this routine makes it such that a process can raise the number of tickets it receives, and thus receive a higher proportion of CPU cycles. This routine should return 0 if successful, and -1 otherwise (if, for example, the caller passes in a number less than one).

The second is `int getpinfo(struct pstat *)`. This routine returns some information about all running processes, including how many times each has been chosen to run and the process ID of each. You can use this system call to build a variant of the command line program `ps`, which can then be called to see what is going on. The structure `pstat` is defined below; note,

you cannot change this structure, and must use it exactly as is. This routine should return 0 if successful, and -1 otherwise (if, for example, a bad or NULL pointer is passed into the kernel).

Most of the code for the scheduler is quite localized and can be found in `proc.c`; the associated header file, `proc.h` is also quite useful to examine. To change the scheduler, not much needs to be done; study its control flow and then try some small changes.

You'll need to assign tickets to a process when it is created. Specifically, you'll need to make sure a child process *inherits* the same number of tickets as its parents. Thus, if the parent has 10 tickets, and calls `fork()` to create a child process, the child should also get 10 tickets.

You'll also need to figure out how to generate random numbers in the kernel; some searching should lead you to a simple pseudo-random number generator, which you can then include in the kernel and use as appropriate.

Finally, you'll need to understand how to fill in the structure `pstat` in the kernel and pass the results to user space. The structure should look like what you see here, in a file you'll have to include called `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 tickets[NPROC]; // the number of tickets this process has
    int pid[NPROC]; // the PID of each process
    int ticks[NPROC]; // the number of ticks each process has accumulated
};

#endif // _PSTAT_H_
```

Good examples of how to pass arguments into the kernel are found in existing system calls. In particular, follow the path of `read()`, which will lead you to `sys_read()`, which will show you how to use `argptr()` (and related calls) to obtain a pointer that has been passed into the kernel. Note how careful the kernel is with pointers passed from user space -- they are a security threat(!), and thus must be checked very carefully before usage.

### Part 3: Comparing Schedulers [10]

*Compare the performance of default, MLFQ and Lottery schedulers, by writing at least two interesting test cases. A test case, in this context would be a C program that can create processes for the xv6 scheduler to run, via use of system calls like `fork()`. You are required to submit the test cases with your assignment submission.*

