CSCI350 Project 3

Due: April 1st by 11:59pm

Tasks

1. Implement a basic non-preemptive MFQ (Multilevel Feedback Queue) scheduler with an aging mechanism to prevent starvation.
2. Add a new system call to get information about a running process
3. Design tests to demonstrate the correctness of your scheduler.
4. Show how process behavior interacts with the scheduler by creating timeline graphs.

Overview

Clone your Project 3 repo to your 32 bit Ubuntu VM.

In this project, you'll implement a simplified Multi-level Feedback Queue (MFQ) scheduler in xv6.

Start by reading Chapter 5 of the xv6 book. This is a very good guide to the details of how scheduling and context switches are implemented in xv6. The default and only scheduling policy in xv6 is **round-robin**. Every runnable process gets an equal CPU time-slice, regardless of priority.

You will build an MFQ scheduler with three priority queues: the top queue (numbered 0) has the highest priority and the bottom queue (numbered 2) has the lowest priority. When a process uses up its time-slice (counted as a number of ticks), it should be downgraded to the next (lower) priority level.

Details

You need to understand how scheduler works in xv6. Most of the code for the scheduler is quite localized and can be found in **proc.c**, where you should first look at the routine `scheduler()`. It's essentially looping forever and for each iteration, it looks for a runnable process across the `ptable`. If there are multiple runnable processes, it will select one according to some policy. The vanilla xv6 does no fancy things about the scheduler; it simply schedules processes for each iteration in a round-robin fashion. For example, if there are three processes A, B and C, then the pattern under the vanilla round-robin scheduler will be `A B C A B C ...`, where each letter represents a process scheduled within a **timer tick**.

~~To change the scheduler, not too much needs to be done; study its control flow and then try some small changes~~

Another relevant file is **trap.c**. The function `trap(struct trapframe *tf)` handles all the interrupts. The vanilla xv6 simply forces a process to yield at each timer interrupt (tick). You may want to do more than that to implement MLFQ.

Tips for Project 3: (read at the end again)

- Many students find this project to be the most difficult one all semester. *Start early.*
- You may find it helpful to include a "position of last element in queue" variable and a counter for ticks
- Remember that once a process has finished running, it should be removed from the queue!
- In part 4, some simple I/O and CPU-intensive tasks are printing and adding, respectively

Task 1: Implement MFQ

To make your life easier and our testing easier, you should run xv6 on only a single CPU (the default is two). To do this, in your Makefile, replace `CPUS := 2` with **`CPUS := 1.`**

It is much easier to deal with fixed-sized arrays in xv6 than linked-lists. For simplicity, we recommend that you use arrays to represent each priority level (queue). For example, define the following variables to represent the three priority queues:

```
struct proc* q0[NPROC];
struct proc* q1[NPROC];
struct proc* q2[NPROC];
```

Your MFQ scheduler must follow these precise rules:

1. It must have three priority levels, numbered from 0 (highest) down to 2 (lowest).
2. *On every xv6 timer tick* the highest priority *RUNNABLE* process is scheduled to run.
3. If there are more than one process on the same priority level, then your scheduler should schedule all the processes at that particular level in a round robin fashion.
4. The time-slice for priority 0 should be 1 timer tick. The times-slice for priority 1 is 2 timer ticks; for priority 2, it is 8 timer ticks.
5. When a timer tick occurs, whichever process was currently using the CPU should be considered to have used up an entire timer tick's worth of CPU. That is, if a process is running and its total ticks (since it was created) is *n-1* before it yields the CPU, the *n*th tick—the one that was about to occur before the process yielded the CPU—should not count towards the total ticks or the current time-slice ticks. (Yes, a process could game the scheduler by repeatedly relinquishing the CPU just before the timer tick occurs, therefore preventing its time-slice from ever being exhausted; ignore this!)
6. When a new process arrives, it should be placed at the end the priority 0 queue (highest priority).
7. At priorities 0, and 1, after a process consumes its time-slice it should be downgraded one priority level. Whenever a process is moved to a lower priority level, it should be placed at the end of the queue.
8. If a process wakes up after voluntarily giving up the CPU (e.g., by performing I/O or sleeping before the time slice is up) it stays at the same priority level.
   - Place this process at the end of its queue; it should not preempt a process with the same priority.
9. Your scheduler **should never preempt** a lower priority process if a higher priority process is available to run.
10. You need to implement the priority boosting mechanism to avoid starvation, which will be used to increase a priority of a process that has not been scheduled in a while. The goal is to avoid starvation, which happens when a process never receives CPU time because higher-priority processes keep arriving.
    - After a *RUNNABLE* process has been waiting in the lowest priority queue for 50 ticks or more, move the process to the end of the highest priority queue. This method of priority boosting is called aging.

Task 2: Create a new system call

For this project, you need to create a new system call to help you debug the scheduler and to help you obtain information about how multiple processes are scheduled over time.

`int getpinfo(int pid).`

This routine takes as input an ID of a process (pid) and prints some basic scheduling information about the process to the terminal: the ticks at which it is scheduled, how long it runs before it gives up the CPU, what priority it has each time it is scheduled, etc. This function returns -1 in case of error and 0 in case of success. A process should call this function at the very end of its execution to get a full view of its scheduling record.

You'll need to fill in all the pieces of information somehow in the kernel and print the results for the user. You have done this in project1. You may want to stare at the routines like `int argint(int n, int *ip)` in **syscall.c** for some hints

To get the information mentioned above, there are several changes that we suggest you make to xv6.

1. You'll probably need to define a structure in the kernel to record scheduling information for a process at each tick. This will give you the necessary information to plot the graphs in part 3.

Create a file named **pstat.h** that contains the following information.

```
#ifndef _PSTAT_H_
#define _PSTAT_H_
#define NTICKS 500
#define NSCHEDSTATS 1500

/*
 * responsible for recording the scheduling state per process at a particular tick
 * e.g. a process can have an array of sched_stat_t's, with each of them holding the info
 * of a scheduling round of the process
 */
struct sched_stat_t
{
  int start_tick;      //the number of ticks when this process is scheduled
  int duration;        //number of ticks the process is running before it gives up the CPU
  int priority;        //the priority of the process when it's scheduled

  //you may add more fields for debugging purposes
};

#endif
```

2. You may also want to add some new fields in the proc struct as described below. These fields are not particularly useful for plotting the graphs in part 3 but can be used for debugging your MLFQ scheduler.

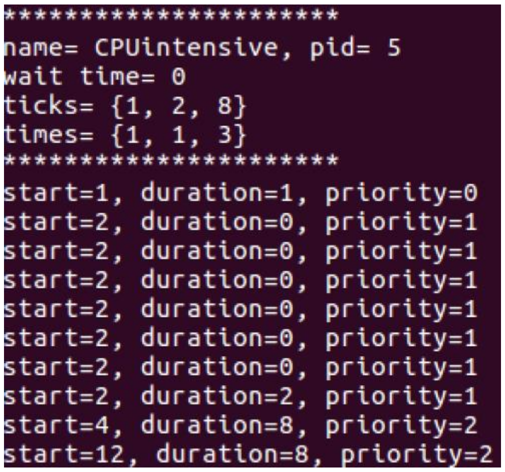
```
int times[3]                // number of times each process was scheduled at each of 3
                            // priority queues

int ticks[3]                // number of ticks each process used the last time it was
                            // scheduled in each priority queue
                            // cannot be greater than the time-slice for each queue

uint wait_time;             // number of ticks each RUNNABLE process waited in the lowest
                            // priority queue
```

3. If you're using the number of ticks as the index to an array, please be careful that this number may exceed `NTICK` after a while and overflow your array (this may happen without any errors being emitted), leading to nonsensical results. You probably want to figure out a way to allow a process to start a tick variable itself, and to allow other processes to join this process's "timeline" by sharing that tick variable (you can do so through system calls).

For your reference, `getpinfo()` would produce an output similar to this: (your formatting does not need to match that of the reference output)



```
**********************
name= CPUintensive, pid= 5
wait time= 0
ticks= {1, 2, 8}
times= {1, 1, 3}
**********************
start=1, duration=1, priority=0
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=2, priority=1
start=4, duration=8, priority=2
start=12, duration=8, priority=2
```

Task 3: Create Tests

In addition to implementing the MFQ scheduler, you must design three user-level test programs to show that that your scheduler works correctly.

Writing tests for xv6 is a good exercise in itself, however, it is a bit challenging. This is a good chance to practice creating test cases and think comprehensively!

Your tests for xv6 are essentially user programs that execute at the user level. Create three user level programs (named `test1.c`, `test2.c` and `test3.c`) with different workloads to demonstrate behavior of your scheduler. In particular,

- **`test1.c`** should implement a mixed workload that includes an IO intensive process and a CPU intensive process.
- **`test2.c`** should implement a workload that demonstrates how priority boost can improve performance of a long running CPU bound process.
- **`test3.c`** should implement a workload that demonstrates how MFQ scheduler can be gamed. A process manages to stay in the highest priority queue by yielding the CPU before its time slice expires. Hint: you can create a loop that iteratively calls `sleep()` to relinquish the processor before the time interrupt occurs. The number of ticks will not be updated, and the process will stay in the highest priority queue.

Your tests should compile and run. The programs should print out the process statistics by calling `getpinfo()`.

Task 4: Make graphs

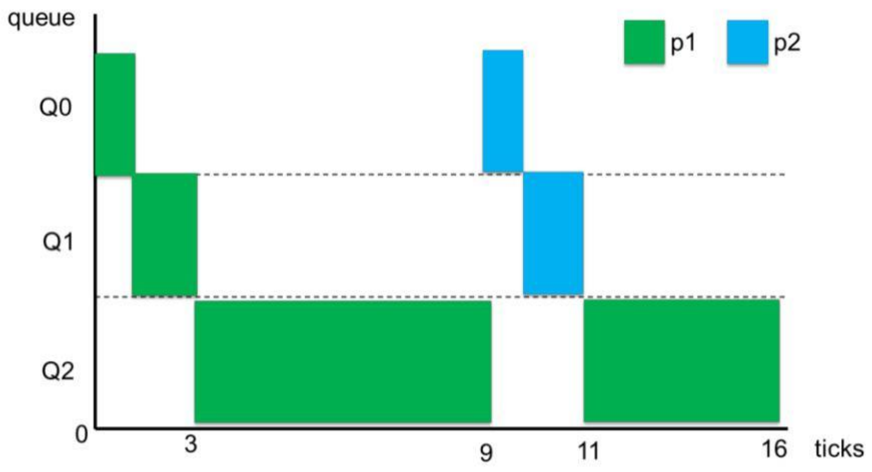
You should make three graphs that show timelines of the processes in each of your three tests, including which queue each process is on, and how much CPU time (ticks) they received.

You can use whatever graphing tool you would like ("gnuplot" is a fine, basic choice). When you have your graphs, please create a .pdf versions of them and place it in a file named **graph1.pdf**, **graph2.pdf** and **graph3.pdf**.

Please describe the workload that you ran to create your graph and explain why your graph shows the results that it does. Create a separate file **workload.pdf** or **workload.txt** (if you use plain text). These are the only formats and filenames you should use.

To obtain the info for your graph, you should use the `getpinfo()` system call. Use the graphs to prove to us that your scheduler is working as desired.

Example graph:



Grading and Submission

You need to submit your code, design document, tests, graphs and workload description files.

The total number of the points for this project is 100.

Your report (see SubmissionInstructions document in Resources for details) is worth 10 points.

- Describe your changes to the xv6 code
    - Name the files your modified and briefly describe the changes
        - Clearly explain the changes you made as a part of Task1 to implement:
            1. MFQ policies
            2. Priority boost mechanism
            3. Workload description for your test programs
    - Please don't modify any files you don't need to! You'll make grading harder for us if you do.
- Your code must match the general code style of xv6

Correct implementation of the MFQ: 20 points

Correct implementation of the priority boost: 10 points

Correct implementation of the tests: 5 points each (15 points total). No points if the tests do not compile.

Graphs, 5 points each (15 points total).

Free Response Questions - pdf file including questions is in Black Board: 15 points

FRQ should be submitted to BB, and each student should answer FRQ individually.

Detailed rubric will be provided again on Piazza.