CS 537 Spring 2019, Project 2b: xv6 Scheduler

Change Log:

- Compile error with procstate not found (https://piazza.com/class/jql5yyu0x8a12k?cid=721) – solution: add this line to user/user.h

  ```
  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
  ```

FAQ

- Resources to understand MLFQ Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=741)
- How to implement queues? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=787)
- Should ticks and wait_ticks be reseted? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=739)Q2 (https://piazza.com/class/jql5yyu0x8a12k?cid=794)
- She we increase wait_ticks in non-RUNNING state? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=709)
- Where should I put pstat.h and how? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=710) Q2 (https://piazza.com/class/jql5yyu0x8a12k?cid=737)
- How to run the tests? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=714)
- When do we need to reset wait_tick? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=807)
- Error when using the pstat.h Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=716)
- **MLFQ Logic:**

  > 1. Arrival of a new process: Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=720) Q2 (https://piazza.com/class/jql5yyu0x8a12k?cid=782)
  > 2. Round robin in q3,q2 and q1: Q3 (https://piazza.com/class/jql5yyu0x8a12k?cid=783)
  > 3. queue0 FIFO clarification: Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=927)
  > 4. Sleep in the low priority queue: Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=925)
  > 5. Others: Q4 (https://piazza.com/class/jql5yyu0x8a12k?cid=772) Q5 (https://piazza.com/class/jql5yyu0x8a12k?cid=773) Q6 (https://piazza.com/class/jql5yyu0x8a12k?cid=785) Q7 (https://piazza.com/class/jql5yyu0x8a12k?cid=788)

- What do I need to add to proc struct? add things like priority, ticks, wait_ticks Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=725)
- ticks vs wait_ticks Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=726)
- Would would be expected when running ticks-test example? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=733)
- How to promote or demote a process? Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=749)
- NLAYER undeclared Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=763)

Objectives

- To understand code for performing context-switches in the xv6 kernel.
- To implement a basic MLFQ scheduler and FIFO scheduling method.
- To create system call that extract process states.

Overview

In this project, you'll be implementing a simplified **`multi-level feedback queue (MLFQ)`** scheduler in xv6. The basic idea is simple. Build an `MLFQ` scheduler with `four` priority queues; the `top` queue (numbered 3) has the `highest` priority and the `bottom` queue (numbered 0) 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. The time-slices for higher priorities will be shorter than lower priorities. The scheduling method in each of these queues will be `round-robin` , except the bottom queue which will be implemented as `FIFO` .

To make your life easier and our testing easier, you should run xv6 on only a ** **`single CPU`** ** . Make sure in your Makefile `CPUS := 1` .

Particularly useful for this project: Chapter 5 (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf) in xv6 book.

Project Details

You have two specific tasks for this part of the project. However, before starting these two tasks, you need first have a high-level understanding of how scheduler works in xv6.

Most of the code for the scheduler is quite localized and can be found in `kernel/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` , which is essentially `~10ms` , and you may assume that this timer tick is equivalent to a single iteration of the for loop in the `scheduler()` code. Why `10ms` ? This is based on the `timer interrupt frequency setup` in xv6 and you may find the code for it in `kernel/timer.c` . You can also find a code walkthrough in the discussion video (https://youtu.be/fJuljW4GjDw).

Now to implement `MLFQ` , you need to schedule the process for some time-slice, which is some multiple of timer ticks. For example, if a process is on the `highest priority level` , which has a time-slice of `8 timer tick` s, then you should schedule this process for `~80ms` , or equivalently, for `8 iterations` .

xv6 can perform a context-switch every time a timer interrupt occurs. For example, if there are 2 processes A and B that are running at the highest priority level (queue 3), and if the round-robin time slice for each process at level 3 (highest priority) is `8 timer ticks` , then if process A is chosen to be scheduled before B, A should run for a complete `time slice` (~80ms) before B can run. Note that even though process A runs for 8 timer ticks, every time a timer tick happens, process A will yield the CPU to the scheduler, and the scheduler will decide to run process A again (until its `time slice` is complete).

Implement **`MLFQ`**

Your `MLFQ` scheduler must follow these very precise rules:

- `Four` priority levels, numbered from `3` (**highest**) down to `0` (**lowest**).
- Whenever the xv6 10 ms timer tick occurs, the highest priority ready process is scheduled to run.
- The highest priority ready process is scheduled to run whenever the previously running process `exits` , `sleeps` , or otherwise `yields` the CPU.
- If there are more than one processes on the same priority level, then you scheduler should schedule all the processes at that particular level in a `round robin` fashion. Except for priority level `0` , which will be scheduled using `FIFO` basis.
- 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` , even if it did not start at the previous tick (Note that a `timer tick` is different than the `time-slice` .)
- The `time-slice` associated with priority `3` is `8 timer ticks` ; for priority `2` it is `16 timer ticks` ; for priority `1` it is `32 timer ticks` , and for priority `0` it executes the process until completion.
- When a new process arrives, it should `start` at priority `3` (highest priority).
- At priorities `3` , `2` , and `1` , after a process consumes its time-slice it should be downgraded `one` priority. At priority `0` , the process should be executed to completion.
- If a process voluntarily relinquishes the CPU before its time-slice expires at a particular priority level, its time-slice should not be reset; the next time that process is scheduled, it will continue to use the remainder of its existing time-slice at that priority level.
- To overcome the problem of starvation, we will implement a mechanism for priority boost. If a process has waited `10x` the time slice in its current priority level, it is raised to the next `higher` priority level at this time (unless it is already at priority level `3`). For priority `0` , which does not have a time slice, processes that have waited `500 ticks` should be raised to priority `1` .

Create new system calls

You'll need to create one system call for this project: `int getpinfo(struct pstat *)` Because your `MLFQ` implementations are all in the kernel level, you need to extract useful information for each process by creating this system call so as to better test whether your implementation works as expected.

To be more specific, this system call returns `0` on `success` and `-1` on failure. If success, some basic information about each process: its process ID, how many timer ticks have elapsed while running in each level, which queue it is currently placed on (3, 2, 1, or 0), and its current procstate (e.g., SLEEPING, RUNNABLE, or RUNNING) will be filled in the `pstat` structure as defined

```
struct pstat {
  int inuse[NPROC]; // whether this slot of the process table is in use (1 or 0)
  int pid[NPROC];   // PID of each process
  int priority[NPROC];  // current priority level of each process (0-3)
  enum procstate state[NPROC];  // current state (e.g., SLEEPING or RUNNABLE) of each process
  int ticks[NPROC][4];  // number of ticks each process has accumulated at each of 4 priorities
  int wait_ticks[NPROC][4]; // number of ticks each process has waited before being scheduled
};
```

The file can be seen pstat.h (http://pages.cs.wisc.edu/~shivaram/cs537-sp19/pstat.h). Do not change the names of the fields in `pstat.h` .

Tips

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 too much needs to be done; study its control flow and then try some small changes.

As part of the information that you track for each process, you will probably want to know its current priority level and the number of timer ticks it has left.

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.

You'll need to understand how to fill in the structure `pstat` in the kernel and pass the results to user space and how to pass the arguments from user space to the kernel. 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 `int argint(int n, int *ip)` in `syscall.c` (and related calls) to obtain a pointer that has been passed into the kernel.

To run the xv6 environment, use make `qemu-nox` . Doing so avoids the use of X windows and is generally fast and easy. However, quitting is not so easy; to quit, you have to know the shortcuts provided by the machine emulator, qemu. Type `control-a` followed by `x` to `exit` the emulation. There are a few other commands like this available; to see them, type `control-a` followed by an `h` .

Code

We suggest that you start from the source code of xv6 at `~cs537-1/xv6-sp19` , instead of your own code from p1b as bugs may propagate and affect this project.

```
> cp -r ~cs537-1/xv6-sp19 ./
```

Testing

Testing is critical. Testing your code to make sure it works is crucial. Writing testing scripts for xv6 is a good exercise in itself, however, it is a bit challenging. As you may have noticed from p1b, all the tests for xv6 are essentially user programs that execute at the user level.

Basic Test

You can test your `MLFQ` scheduler by writing workloads and instrumenting it with `getpinfo` systemcall. Following is an example of a user program which spins for a user input iterations. You can download this example here (http://pages.cs.wisc.edu/~shivaram/cs537-sp19/ticks-test.c).

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

int
main(int argc, char *argv[])
{
    struct pstat st;

    if(argc != 2){
        printf(1, "usage: mytest counter");
        exit();
    }

    int i, x, l, j;
    int mypid = getpid();

    for(i = 1; i < atoi(argv[1]); i++){
        x = x + i;
    }

    getpinfo(&st);
    for (j = 0; j < NPROC; j++) {
        if (st.inuse[j] && st.pid[j] >= 3 && st.pid[j] == mypid) {
            for (l = 3; l >= 0; l--) {
                printf(1, "level:%d \t ticks-used:%d\n", l, st.ticks[j][l]);
            }
        }
    }

    exit();
    return 0;
}
```

If you run `ticks-test 10000000` the expected output is something like below:

```
level:3      ticks-used:8
level:2      ticks-used:16
level:1      ticks-used:32
level:0      ticks-used:160
```

The ticks used on the last level will be somewhat unpredictable and it may vary. However, on most machines, we should be able to see that the ticks used at levels `3`, `2`, and `1` as `8`, `16` and `32` respectively. As there are no other programs are running, there won't be any boost for this program after it reaches bottom queue (numbered 0) (It won't wait for 500 ticks). If you invoke with small counter value such as 10 (`ticks-test 10`), then the output should be like this:

```
level:3      ticks-used:1
level:2      ticks-used:0
level:1      ticks-used:0
level:0      ticks-used:0
```

Mulitple Jobs Test

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "pstat.h"
#define check(exp, msg) if(exp) {} else {       \
  printf(1, "%s:%d check (" #exp ") failed: %s\n", __FILE__, __LINE__, msg); \
  exit();}

int pow2[] = {80000000, 32, 16, 8};

int workload(int n) {
  int i, j = 0;
  for(i = 0; i < n; i++)
    j += i * j + 1;
  return j;
}

int
main(int argc, char *argv[])
{
  struct pstat st;

  sleep(10);

  int i, j, k, count = 0;
  for (i = 0; i <= 60; i++) {
    if (fork() == 0) {
      workload(4000000 * (i + 1));
      if (i == NPROC - 4) {
        sleep(100);
        check(getpinfo(&st) == 0, "getpinfo");

        // See what's going on...

        for(k = 0; k < NPROC; k++) {
          if (st.inuse[k]) {
            int m;
            printf(1, "pid: %d\n", st.pid[k]);
            if (st.pid[k] > 3) {
                check(st.ticks[k][3] > 0, "Every process at the highest level should use at least 1 t
imer tick");
            }
            for (m = 3; m >= 0; m--) {
              printf(1, "\t level %d ticks used %d\n", m, st.ticks[k][m]);
            }
          }
        }
        for(k = 0; k < NPROC; k++) {
          if (st.inuse[k]) {
            count++;
            check(st.priority[k] <= 3 && st.priority[k] >= 0, "Priority should be 3, 2, 1 or 0");
            for (j = 3; j > st.priority[k]; j--) {
              if (st.ticks[k][j] != pow2[j]) {
                printf(1, "#ticks at this level should be %d, \
                    when the priority of the process is %d. But got %d\n",
                    pow2[j], st.priority[k], st.ticks[k][j]);
                exit();
              }
            }
            if (st.ticks[k][j] > pow2[j]) {
              printf(1, "#ticks at level %d is %d, which exceeds the maximum #ticks %d allowed\n", j,
st.ticks[k][j], pow2[j]);
              exit();
            }
          }
        }
        check(count == NPROC, "Should have 64 processes currently in used in the process table.");
        printf(1, "TEST PASSED");
      }
    } else {
      wait();
      break;
    }
  }

  exit();
}
```

Note: The exact number of ticks may vary across machines; however, if `TEST PASSED` is printed at the end of the program's execution, then the scheduler has passed the test.

Write Your Own Tests

These tests are by no means exhaustive. It is important *(and a good practice)* to write your own test programs. Try different tests which will exercise the functionality of the scheduler by using `fork()` , `sleep(n)` , and other methods.

Due Date

This project is due on **February, 27th, 2019 at 11:59PM**.

Submitting Your Implementation

To submit your solution, copy all of the xv6 files and directories with your changes into `~cs537-1/handin/<cs-login>/p2b/` . One way to do this is to navigate to your solution's working directory and execute the following command:

```
cp -r . ~cs537-1/handin/<cs-login>/p2b/
```

Consider the following when you submit your project:

- If you use any slip days you have to create a `SLIP_DAYS` file in the `/<cs-login>/p2b/` directory otherwise we use the submission on the due date.
- Do not remove the `README` file in the main directory
- Do not change the permissions
- Do not remove the other files that you are not modifying
- Do not modify or remove the Makefile
- Your files should be directly copied to `~cs537-1/handin/<cs-login>/p2b/` directory. Having subdirectories in `<cs-login>/p2b/` like `<cs-login>/p2b/xv6-sp19` or … is **not acceptable**.