CS 537 Spring 2020, Project 2b: xv6 Scheduler

FAQ

- NLAYER undeclared: pstat.h has been updated. If you want to use the old one add #define NLAYER 4 in either params.h or pstat.h
- pstat.h include error: Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=467) Q2 (https://piazza.com/class/k4bkdfbqqziw5?cid=532)
- Max wait time clarification Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=508)
- How to implement queus? Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=501)
- *boostproc* system return value Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=503)
- When should I update *ticks* for process? Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=497)
- How to implement *getprocinfo* sys call? Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=496), Q2 (https://piazza.com/class/k4bkdfbqqziw5?cid=504)
- *ticks* should be accumulated or cleared? accumulated for reporting Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=495),Q2 (https://piazza.com/class/k4bkdfbqqziw5?cid=491)
- When to reset *wait_ticks*? Q3 (https://piazza.com/class/k4bkdfbqqziw5?cid=512)Q2 (https://piazza.com/class/k4bkdfbqqziw5?cid=488), Q3 (https://piazza.com/class/k4bkdfbqqziw5?cid=476)
- time slice clarification:Q2 (https://piazza.com/class/k4bkdfbqqziw5?cid=524) Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=446)
- Processes with the same priority running order Q2 (https://piazza.com/class/k4bkdfbqqziw5?cid=520) Q1 (https://piazza.com/class/k4bkdfbqqziw5?cid=498)

Administrivia

- **Due Date** by Feb 24, 2020 at 10:00 PM
- Questions: We will be using Piazza for all questions.
- Collaboration: The assignment has to be done by yourself. Copying code (from others) is considered cheating. Read this (http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2018/dontcheat.html) for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- This project is to be done on the lab machines (https://csl.cs.wisc.edu/services/instructional-facilities), so you can learn more about programming in C on a typical UNIX-based platform (Linux).
- Please take the **quiz on Canvas** that will check if you have read the spec closely.

Objectives

- To understand code for performing context-switches in the xv6 kernel.
- To implement a basic MLFQ scheduler and Round-Robin scheduling method.
- To create system calls to extract process states and test the scheduler

Overview

In this project, you'll be implementing a simplified **multi-level feedback queue (MLFQ)** scheduler in xv6. Here is the overview of **MLFQ** scheduler in this assignment:

- **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**.

You will be implementing 2 system calls:

- `int getprocinfo(struct pstat *)` to extract process state to test the scheduler.
- `int boostproc(void)` which boosts the process **one** level in the MLFQ scheduler.

Reading:

- Chapter 5 (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf) in xv6 book.
- Discussion video from this year (https://mediaspace.wisc.edu/media/Shivaram+Venkataraman-+Psychology105+2.13.2020+5.30.24PM/0_90wjyp9d) and from last year (https://youtu.be/fJuljW4GjDw)

xv6 scheduler code details

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` (8 timer ticks or ~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 it's `time slice` is complete).

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` .

MLFQ implementation details

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

1. **Four** priority levels.
2. Number the queues from `3` for highest priority queue down to `0` for **lowest** priority queue.
3. Whenever the xv6 timer tick (10 ms) occurs, the highest priority ready process is scheduled to run.
4. The highest priority ready process is scheduled to run whenever the previously running process `exits` , `sleeps` , or otherwise `yields` the CPU.
5. The `time-slice` associated with priority queues are as follows:
    - Priority queue `3` : `8 timer ticks` (or ~80ms)
    - Priority queue `2` : `16 timer ticks`
    - Priority queue `1` : `32 timer ticks`
    - Priority queue `0` it executes the process until completion.
6. 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.
7. The round-robin slices differs for each queue and is as follows:
    - Priority queue `3` slice : `1 timer ticks`
    - Priority queue `2` slice : `2 timer ticks`
    - Priority queue `1` slice : `4 timer ticks`
    - Priority queue `0` slice : `64 timer ticks`
8. When a new process arrives, it should **start** at priority `3` (highest priority).
9. 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.
10. 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.
11. 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 the queue number `0` (lowest priority) consider the maximum wait time to be `6400` ms which equals to `640` **timer ticks**.
12. To make the scheduling behavior more visible you will be implementing a system call that boosts a process priority by **one** level (unless it is already at priority level `3`).

Create new system calls

You'll need to create two system call for this project:

1. `int getprocinfo(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 prioriti
es
  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-sp20/pstat.h). Do not change the names of the fields in `pstat.h` .

Action items to successfully add `pstat.h` :

- Copy the `pstat.h` file to `xv6/include` directory
- Add following line to the `user/user.h` file

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

- Include `pstat.h` in places needed. More hints here (https://piazza.com/class/k4bkdfbqqziw5?cid=467).

2. `int boostproc(void)`

This system call simply increases the priority of current process **one** level unless it is already in queue number `3` .

[EDITED] for simplicity this system call returns 0 unconditionally.

In reality existence of this system call makes the system susceptible to attack, where the process increases its priority right after it is demoted. But for the scope of this assignment it'll make it easier for you to test if things are working correctly!

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`. Type `cntl-a` followed by `x` to exit the emulation.

Code

We suggest that you start from the source code of xv6 at `~cs537-1/xv6-sp20`, instead of your own code from p1b as bugs may propagate and affect this project.

```
> cp -r ~cs537-1/xv6-sp20 ./
```

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 `getprocinfo` 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-sp20/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;
    }

    getprocinfo(&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
```

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.

Submitting Your Implementation

1. Please create a subdirectory ~cs537-1/handin/LOGIN/p2b/src

```
mkdir -p ~cs537-1/handin/LOGIN/p2b/src
```

To submit your solution, copy all of the xv6 files and directories with your changes into `~cs537-1/handin/<cs-login>/p2b/src`. 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/src/
```

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/src` directory. Having subdirectories in `<cs-login>/p2b/src` like `<cs-login>/p2b/src/xv6-sp19` or … is **not acceptable**.