January 2023 CSE 314

Offline Assignment 3: xv6 – Scheduling

In this assignment, you will be implementing a new scheduler for the xv6 operating system. This scheduler will implement an MLFQ (Multilevel Feedback Queue) scheduling algorithm. There will be two queues – one implementing lottery scheduling and the other working in a Round-Robin fashion.

Multilevel Feedback Queue

There are multiple queues, each having a prespecified scheduling algorithm and a limit of time slices (clock ticks). Usually the time limit continues to increase from top to bottom. Processes are scheduled according to the following rules.

- A new process is always inserted at the end (tail) of the topmost queue (queue 1).
- The scheduler searches the queues starting from the topmost one for scheduling processes.
    - When the scheduler finds a non-empty queue, it schedules a process according to the specified algorithm of that queue.
    - After the process leaves the CPU (either voluntarily or being preempted), the scheduler starts searching again from the topmost queue.
- When a process is assigned to the CPU,
    - if it is completed within the time limit of its queue, it leaves the system.
    - if it voluntarily relinquishes control of the CPU, it is inserted at the tail of the immediate higher level queue, i.e., if a process voluntarily leaves CPU while it was in queue 2, it is inserted at the tail of queue 1.
    - if it consumes all the time slices, it is preempted and inserted at the end of the next lower level queue.
- After a certain time interval, all the processes are brought to the topmost queue, which is known as priority boosting.

Lottery Scheduling

The basic idea behind lottery scheduling is quite simple. Each process is assigned a fixed (integer) number of tickets. A process is probabilistically assigned a time slice based on its number of tickets. More specifically, if there are n processes $p_1, p_2, \dots, p_n$ and they have $t_1, t_2, \dots, t_n$ tickets respectively at any time, then the probability of a process p_i being scheduled at that time is $\frac{t_i}{\sum_{j=1}^{n} t_j}$. Basically, you need to sample t_i based on the probability distribution derived from the ticket counts. After each time slice, t_i will be reduced by 1 (i.e., the process has used that ticket). Hence, the probabilities need to be recomputed each time using the updated ticket counts. All the processes are reinitialized with their original ticket count once the ticket counts of **all runnable** processes become 0.

Specifications

For this assignment, we will initially assume that there is only one CPU. This can be implemented by setting `CPUS = 1` in the `Makefile` of xv6.

As we have only two queues, the top one (queue 1) will be implementing lottery scheduling and the bottom one (queue 2) will implement the Round-Robin algorithm. The time limits of the queues are defined using macros `TIME_LIMIT_1` and `TIME_LIMIT_2` having values 1 and 2, i.e., a process gets only one time

slice while it is in the top queue and 2 time slices while in the bottom queue. The priority boosting time interval is defined using another macro `BOOST_INTERVAL` having value 64. All the macros should be defined in `kernel/param.h`.

Also, you may assume that the total number of processes will be small enough so that you can check each of them for their queue number instead of implementing the queues rigorously.

Scheduling Algorithm

You need to implement the scheduling algorithm mostly in the `kernel/proc.c` file. Try to understand the default Round-Robin algorithm implemented inside the `scheduler()` function. It basically loops over all the active processes and runs the first one which is in `runnable` state (states of processes are defined in `kernel/proc.h`) in a Round-Robin format.

For the assignment, you may add additional variables in `struct proc` defined in `kernel/proc.h` to store necessary information like which queue the process is currently in, its original ticket count, current ticket count, consumed time slots (both total and in current turn) etc. When a process is created, these variables should be initialized and then updated when needed.

Inside the `scheduler()` function, you need to schedule the processes according to the specifications. After a process returns, check its consumed time slices and change its queue accordingly. To find the consumed time slices, you may update all the running processes after each clock interrupt. The `clockintr()` function defined in `kernel/trap.c` might seem useful.

System Calls

You will need two system calls for your implementation:

`settickets`

The first system call is `int settickets(int number)`, which sets the number of tickets of the calling process. By default, each process should get a default number of tickets; calling this routine makes it such that a process can change the number of tickets it receives and thus receives a different 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, in which case the default number of tickets are allocated). The default number of tickets will be equal to `DEFAULT_TICKET_COUNT` macro (keep its value as 10) defined in `kernel/param.h`.

`getpinfo`

The second system call is `int getpinfo(struct pstat *)`. This routine returns some information about all active processes, including their PIDs, statuses, which queue each one is currently in, how many time slices each has been scheduled to run etc. You can use this system call to build a variant of the linux command line program `ps`, which can then be called to see what is going on. The structure `pstat` is defined below; note that you cannot change this structure and must use it exactly as is. When the `getpinfo()` routine is called, the `pstat` structure should be updated with the necessary values. The routine should return 0 if successful and -1 otherwise (if, for example, a bad or `NULL` pointer is passed into the kernel).

You will need to understand how to fill in the structure `pstat` in the kernel and pass the results to the userspace. The structure should look like what you see below. You need to add a file named `pstat.h` to the xv6 kernel directory.

```
#ifndef _PSTAT_H_
#define _PSTAT_H_
#include "param.h"

struct pstat {
  int pid[NPROC]; // the process ID of each process
  int inuse[NPROC]; // whether this slot of the process table is being used (1 or 0)
  int inQ[NPROC]; // which queue the process is currently in
  int tickets_original[NPROC]; // the number of tickets each process originally had
  int tickets_current[NPROC]; // the number of tickets each process currently has
  int time_slices[NPROC]; // the number of time slices each process has been scheduled
};

#endif // _PSTAT_H_
```

Files

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 (maybe from the xv6 book) and then try some small changes.

Ticket Assignment

You will need to assign tickets to a process when it is created. Specifically, you will need to ensure that a child inherits the same number of tickets as its parent. Thus, if the parent originally had 17 tickets and calls `fork()` to create a child process, the child should also get 17 tickets initially.

Argument Passing

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 `argaddr()` (and related calls) to obtain a pointer that has been passed into the kernel. Note how careful the kernel is with pointers passed from the userspace – they are a security threat(!) and thus must be checked very carefully before usage.

Random Number Generation

You will also need to figure out how to generate (pseudo)random numbers in the kernel; you can implement your own random number generator or use any off-the-shelf implementation from the web. You must make sure that the random number generator uses a deterministic seed (so that the results will be reproducible) and is implemented as a kernel-level module.

Bonus Task

Make necessary changes so that the scheduler works for multiple CPUs.

Testing

You need to write two user-level programs, `dummyproc.c` and `testprocinfo.c` to test out the ticket assignment and scheduling, respectively. The `dummyproc.c` program should also have provisions to test forked processes. Its calling syntax should be like `dummyproc 43` where the parameter is used to set the number of tickets. The `testprocinfo.c` program should update the `pstat` structure and then print its contents in a nice format. Its calling syntax should be like `testprocinfo` as it should not need any parameter.

Executing multiple instances of `dummyproc.c` (by appending `&;` after each call) followed by a single instance `testprocinfo.c` should print the relevant statistics defined in the `pstat` structure like below.

```
PID     |     In Use     |     inQ    |   Original Tickets | Current Tickets | Time Slices
1               1               1               370             193             942
2               0               2               110             4               1027
3               1               1               280             159             2028
4               1               1               410             395             1489
5               1               2               190             147             296
```

Marks Distribution

Task	Marks
Random number generation	5
Argument passing	10
Implementing lottery scheduling	30
Implementing MLFQ	40
User-level programs for testing	15
Bonus	35

Submission Guidelines

Start with a fresh copy of xv6 from the original repository. Make necessary changes for this assignment. Like the previous assignment, you will submit just the patch file.

Don't commit. Modify and create files that you need to. Then create a patch using the following command:

```
git add --all
git diff HEAD > <studentID>.patch
```

where studentID is your own seven-digit student ID (e.g., 1905000).

Just submit the patch file, do not zip it. In the lab, during evaluation, we will start with a fresh copy of xv6 and apply your patch using the command:

```
git apply <studentID>.patch
```

Make sure to test your patch file after submission in the same way we will run it during the evaluation.

Please DO NOT COPY solutions from anywhere (your friends, seniors, internet, etc.). Any form of plagiarism (irrespective of source or destination) will result in getting –100% marks in this assignment. It is your responsibility to protect your code.

Deadline: July 24, 2023, Monday, 11:55 PM