January 2024 CSE 314

Offline Assignment 3: xv6 – Scheduling

The default scheduling algorithm in xv6 works in a round robin fashion. In this assignment, you will be implementing an MLFQ (Multilevel Feedback Queue) scheduling algorithm with the aging mechanism. There will be two queues – one with lottery scheduling and the other working in a Round-Robin fashion.

Multilevel Feedback Queue

There are multiple queues (2 queues for this assignment), each having a prespecified scheduling algorithm (lottery scheduling for top queue, round-robin scheduling for bottom queue) and a limit of time slice (clock ticks). Processes are scheduled according to the following rules:

- A new process is always placed at the topmost queue (queue 0).
- 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 exits as usual.
    - if it consumes all the time slice allocated for that queue, it is preempted and inserted at the next lower level queue (from queue 0 to queue 1 in this task).
    - if it voluntarily gives up the control of the CPU before consuming all its time slices (due to waiting for I/O, or sleeping), it is moved to the immediate higher level queue, i.e., if a process voluntarily leaves CPU while it was in queue 1, it is inserted at queue 0.
- Processes that have been waiting too long in the lower-priority queue 1 will be promoted to the higher priority queue 0 to reduce the risk of being starved.

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. Each time a process is scheduled, 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.

Although calculating the probability requires a floating point division, you can get around the division like this: first calculate the sum of current tickets for all processes (say n). Then generate a random number $r = (0, n]$ (this must be uniformly generated over the interval, do **not** follow any particular distribution). Finally, keep track of the running sum of current tickets for each process, and the process responsible for causing the sum to exceed r will be the winning one.

Aging

Aging mechanism is a way to prevent process starvation by ensuring that processes that have been waiting for a long time in a lower-priority queue are promoted to a higher-priority queue. If a process is already in a higher priority queue, nothing needs to be done.

Note: Aging is not the same as priority boosting. Aging prevents starvation by gradually increasing the priority of *long-waiting processes*. Priority boosting is designed to restore fairness by *promoting all processes* to a higher priority at periodic intervals.

xv6 Scheduler

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 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** (think of this tick as a single iteration of the for loop in the scheduler function). This timer is maintained by the `ticks` variable in `kernel/trap.c`.

To implement the **Multi-Level Feedback Queue (MLFQ)**, processes must be scheduled for time slices that are multiples of timer ticks. For example, a process in the highest priority queue may have a time slice of 2 timer ticks, or 2 iterations of the scheduler loop.
In xv6, a context switch occurs with each timer interrupt. Suppose there are two processes, A and B, running at the highest priority level (queue 0), with each time slice being 2 timer ticks. If process A is scheduled first, it will **run for the full time slice** before process B gets its turn. **Although process A runs for 2 timer ticks, it yields control to the scheduler after each tick, and the scheduler will decide to resume A until its time slice is exhausted. This part is very important, so make sure you understand it properly.**

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 timer ticks, 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. When a process yields (gives up the CPU due to clock interrupt) or sleeps (gives up the CPU voluntarily), check its consumed time ticks and change its queue accordingly.

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 0) will be implementing lottery scheduling and the bottom one (queue 1) will implement the Round-Robin algorithm.**

- The time limits of the queues are defined using macros `TIME_LIMIT_0` and `TIME_LIMIT_1` having values 2 and 4, i.e., a process gets only 2 time slices while it is in the top queue and 4 time slices while in the bottom queue.
- The waiting threshold is defined using another macro `WAIT_THRESH` having value 6.
- For lottery scheduling, each process will have an initial ticket count of 10, defined by `DEFAULT_TICKETS`.

All the macros should be defined in `kernel/param.h`.

When a new process arrives, it should always start at the highest priority queue. If a process voluntarily gives up the CPU before its time slice is fully used at a certain priority level, it is promoted to the immediate higher priority level, if available. The next time the process is scheduled, it will resume from that priority level.

For this task, you don't need to implement the queues rigorously with enqueue / dequeue methods. Just assign queue numbers to each process, and based on that you can search for processes on queue 0 first, then on queue 1.

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_TICKETS` 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 waiting_time[NPROC];    // the time each process has spent waiting before being scheduled
  int running_time[NPROC]; // Number of times the process was scheduled before its time slice
was used
  int times_scheduled[NPROC];    // the total number of times this process was scheduled
  int tickets_original[NPROC]; // the number of tickets each process originally had
  int tickets_current[NPROC]; // the number of tickets each process currently has
  uint queue_ticks[NPROC][2];    // the total number of ticks each process has spent in each
queue
};
```

```
#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. Be sure to add the proper definitions in `kernel/defs.h`. If a function takes a structure as argument, you must specify the type at the beginning of `defs.h`

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. This can be done inside the `fork()` function in `kernel/proc.c`, when updating the state of the child process.

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.

For passing variables from the kernel to the userspace, you can check the `copyout()` function.

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 run a dummy loop and also have provisions to test forked processes. It's calling syntax should be like `dummyproc 43 100000`, where the first parameter is used to set the number of tickets, and the second parameter is used to control the number of iterations in the loop. The child process will similarly run a dummy loop like the parent process, except that it will sleep after certain iterations so as to simulate voluntarily giving up the CPU before its time slice is exhausted (this might be a bit tricky to test, see the sample logs for more information about this). 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:

```
$ dummyproc.c 10 1000000000 &; dummyproc.c 5 1500000000 &;
$ testprocinfo
```

PID	In Use	In Q	Waiting time	Running time	# Times Scheduled	Original Tickets	Current Tickets	q0	q1
1	0	0	0	1	34	10	9	344	0
2	0	0	0	0	30	9	8	343	0
41	1	0	0	0	1	8	7	0	0
21	1	0	8	1	12	5	4	91	34
23	1	1	3	0	20	10	8	63	61
28	1	1	0	0	12	5	3	79	40

Explanation:

- In Use: Whether this process is RUNNING or RUNNABLE
- In Q: The current queue of the process
- Waiting time: Ticks spent before being scheduled (should be incremented only when the process is RUNNABLE)
- Running time: Ticks spent before using up it's allocated time slice
- Times scheduled: Total number of times the process was scheduled by the CPU
- q0/1: Ticks spent in queue 0/1 (regardless of state)
- *You don't have to print processes with pid = 0*

Important: The max you can go with `uint32` in xv6 is `4294967295`. Make sure that the loop counter never exceeds this value, otherwise the counter will overflow and the loop will never exit.

Update: If you aren't able to fetch information for dummyproc from the process table (testprocinfo doesn't list the process), it's likely because dummyproc is ending too soon. You may want to run it for at least a billion iterations, or create a nested loop to increase the running time.

Logging steps

You may use a global variable like `print_logs` to turn on / off some helpful messages that will be printed to the console:

- When demoting a process to a lower level queue, print:
  `DEMO: Process 1 (sh) ran for 2 time ticks, demoted to queue 1`
- When promoting a process to a higher level queue because of the process voluntarily giving up the CPU (waiting for I/O or called sleep), print a similar message like this:
  `PROMO: Process 1 (sh) ran for 3 time ticks, promoted to queue 0`
  Try sleeping after various iterations to trigger this output. In my case, this triggered after sleeping every 10^8 iterations. The pattern for the child process should look something like this sample log.
- When promoting a process due to the aging mechanism, print:
  `BOOST: Process 1 (sh) waited for 6 ticks, promoted to queue 0`
- During lottery scheduling, print out the winning process like this:
  `LOTTERY: Process 1 (sh) won in queue 0 with tickets 13`

Use proper indentation / formatting to keep the logs separate from the actual console output as much as possible. The scheduler may intermix the logs with the console output, so keep the option to turn the logging on/off when needed. The **ANSI** color codes might be useful in this case.

Marks Distribution

Task	Marks
Random number generation	5
Argument passing in system calls 1. settickets: from user level to kernel level 5 2. getpinfo: from kernel level to user level 5	10
Implementing lottery scheduling	15
Implementing MLFQ	25
Aging mechanism	20
User-level programs for testing 1. Setting the tickets for both parent and child process 5 2. Simulating voluntary yielding by calling sleep 5 3. testprocinfo implementation 5	15
Proper logging at each step	10
Total	**100**
Bonus	5

Submission Guidelines

Failing to follow proper submission guidelines will result in a deduction of 10 marks.

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., 2005010).

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 --whitespace=fix <studentID>.patch
```

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

You are free to share ideas with your friends, but 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: November 02, 2024, 11:55 PM