EEE3535-01 **Fall 2020**

Assignment 4: Process Scheduling

Due: Monday, Nov. 23, 2020, 11:59PM

1 Introduction

- The goal of this assignment is to implement a scheduling policy using multiple process queues in xv6-riscv.
- The scheduling policy to implement in this assignment is analogous to multi-level feedback queue (MLFQ) in that processes hop around multiple queues based on their execution behaviors.
- However, detailed scheduling methods are completely different from the MLFQ. Read instructions carefully to implement the described scheduling policy in this assignment.
- Before starting the assignment, go to the `xv6-riscv/` directory, download the following script file, and execute it to update xv6-riscv. The script makes some minor changes to xv6-riscv for this assignment.

```
$ cd xv6-riscv/
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/sched.sh
$ chmod +x sched.sh
$ ./sched.sh
```

- Launch xv6-riscv, and try executing the `whoami` program. It should print out your student ID and name followed by the process execution information. Details of the message will be explained later.

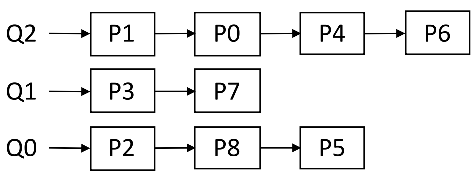
```
$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1
-nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,
drive=x0,bus=virtio-mmio-bus.0

EEE3535 Operating Systems: booting xv6-riscv kernel
EEE3535 Operating Systems: starting sh
$ whoami
Student ID:   2020140000
Student name: William Song
whoami (pid=3): Q2(0%): Q1(0%): Q0(0%)
$ QEMU: Terminated
```

- If xv6-riscv runs fine, terminate it by pressing `Ctrl+a` and then `x`.

2 Scheduling Policy

- The following describes the details of scheduling policy you will have to implement in the xv6-riscv kernel.
- **Scheduler organization**
  - The scheduling policy uses three process queues named `Q2`, `Q1`, and `Q0`. Each queue is implemented as a linked list where processes are connected in a chain via pointers.
  - An example in the figure below shows that `Q2` has four processes (i.e., `P1`, `P0`, `P4`, and `P6`), and `Q1` and `Q0` have two and three processes, respectively.
  - The `Q2` is used for scheduling interactive processes, and the `Q1` is for compute-intensive ones. The `Q0` stores non-runnable processes whose states are `sleep` or `zombie`.
  - Process states are defined as `enum procstate` in the `kernel/proc.h` file of xv6-riscv.



- **Scheduling policy**
  - A new process is first placed in the `Q2` by assuming that it would be an interactive one.
  - If a process invokes a system call or voluntarily gives up the CPU during a timer interrupt interval, the process stays in the `Q2`. But, if a process occupies a whole time slice, then it relocates to the `Q1`.
  - When a process is enqueued, it is placed at the end of list.
  - If a process buffered in the `Q1` makes a system call or voluntarily gives up the CPU, it immediately moves to the `Q2`. Otherwise, it continues to stay in the `Q1`.
  - When a process changes its state to a non-runnable state (i.e., `sleep` or `zombie`), the process is placed in the `Q0`. Thus, scheduling is made only among processes in the `Q2` or `Q1` because those in the `Q0` are not runnable processes.
  - Since the `Q2` includes interactive jobs, processes in this queue should have higher priority for scheduling than those in the `Q1`.
  - However, executing only `Q2` processes makes those in the `Q1` starve for scheduling. Although the starvation problem can be alleviated by introducing a priority boosting method, we will not exploit the priority boosting in this scheduling policy.
  - Instead, processes in the `Q2` and `Q1` are scheduled as follows. To simplify explanations, let us assume that processes do not change their states or move between queues.
  - Processes in the `Q2` are sequentially executed in the order they are queued, i.e., `P1`, `P0`, `P4`, and then `P6`. If the scheduler reaches the end of `Q2` process chain (i.e., linked list), it lets one process in the `Q1` run in the next time slice. Since the `P3` is the head of `Q1` list, this process is executed.
  - After the `P3`, the scheduler again runs processes in the `Q2` from `P1` to `P6`. When the scheduler reaches the end of `Q2` process chain again, it triggers the next process in the `Q1` to run, i.e., `P7`. When the `Q1` has reached the end of list after the `P7`, it loops back to the head of queue. The `P3` will be the next process to run when it becomes the `Q1`'s turn to go.
  - In reality, processes may dynamically hop around different queues and change their states, and thus linked-list pointers of process queues must be carefully handled to avoid erroneous operations.

3 Implementation

- To implement the described scheduling policy in **2 Scheduling Policy** using multiple process queues, you first need to understand how the xv6-riscv kernel handles processes.
- All process-related operations in xv6-riscv occur in `kernel/proc.c`, so the `proc.h` and `proc.c` files are the only ones you will have to read to understand how process scheduling in xv6-riscv works.
- **Process creation**
  - The first process of xv6-riscv is created during boot time in the `userinit()` function in `proc.c`. The first process (i.e., `initproc`) forks a child process and makes it run a shell program named `sh`.
  - Every new process except for the `initproc` is created via the `fork()` function in `proc.c`. The `fork()` calls `allocproc()` to create and initialize a new process.
  - Technically processes are not created in the xv6-riscv kernel. They are statically allocated as as `struct proc proc[NPROC]` near the beginning of `proc.c`, where `NPROC = 64`. It means that xv6-riscv can handle only up to 64 processes at a time.
  - Elements of the statically-allocated `proc` array are used for storing the information of active processes. If an array entry has `UNUSED` state, it does not contain a valid process; a new process later can be allocated to this entry. If the entry has other states (i.e., `SLEEPING`, `RUNNABLE`, `RUNNING`, or `ZOMBIE`), it includes an active process.
  - When a process is completed, it dies in the `freeproc()` function. The xv6-riscv kernel sets the corresponding entry of `proc` array as `UNUSED` to indicate that the entry is available for a new process.
- **Default xv6-riscv process scheduling**
  - xv6-riscv runs processes in round-robin manner by simply scanning through the `proc` array. Whenever it finds a `RUNNABLE` process while walking through the array, it is executed.

- If an array entry is `UNUSED`, there is no valid process in it. If the entry has a `SLEEPING` or `ZOMBIE` process, the non-`RUNNABLE` process is simply skipped. The scheduler continues to scan the array until it finds a `RUNNABLE` process.
- Process scheduling is implemented in the `scheduler()` function in `proc.c`. It repeats in an indefinite `for(;;)` loop until xv6-riscv is forcefully closed (i.e., pressing `Ctrl+a,x`).
- In this assignment, you will have to modify the default round-robin scheduler to implement multi-list queue scheduler described in **2 Scheduling Policy**.

- **Timer interrupt**
  - When a timer interrupt occurs, it falls to the `yield()` function in `proc.c`. This function makes the interrupted process give up the CPU and performs context switching.
  - The `yield()` function may be a good place to check if the interrupted process has occupied the entire time slice and thus needs to move to the `Q1` if it was previously in the `Q2`.
  - The `yield()` function is called either from `usertrap()` or `kerneltrap()` in `trap.c`. Note from the code that a timer interrupt is called only when the CPU is running a user process. If the CPU is idle or executing the kernel thread, the timer interrupt is silenced.
  - Thus, when xv6-riscv runs short programs such as `echo`, `ls`, and `wc`, you may find that timer interrupts occur infrequently and aperiodically instead of having 1ms periodic intervals because these programs rarely occupy the CPU for extensive period of time.
  - The interval of timer interrupt is initialized in the `timerinit()` function in `start.c`.
  - The default xv6-riscv defines `interval = 1000000`, and it says in a comment that it is about 1/10th of a second. It means the default time unit is 100ns.
  - You should notice in your xv6-riscv copy that the `interval = 10000`, which indicates the timer interrupt interval is set to 1ms in this assignment.
  - The wall-clock time in the xv6-riscv kernel can be probed by reading the `ticks` variable. This variable increments every 1ms in the `clockintr()` function in `trap.c` regardless of xv6-riscv kernel activities or timer interrupts.

- **System call**
  - Upon a system call, the xv6-riscv kernel immediately serves the request. It does not perform context switching after the system call but returns to the user process.
  - Since a process is supposed to move to the `Q2` if it is not already there, the `syscall()` function in `syscall.c` may be a good place to move the process.

- **Termination**
  - A process terminates in the `freeproc()` function in `proc.c`.
  - When the process ends, print out how much portion of execution time the process has spent on each queue. For instance, executing a pipe command, `cat README | grep xv6`, may print out the following result.

```
$ cat README | grep xv6
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
The code in the files that constitute xv6 is
(kaashoek,rtm@mit.edu). The main purpose of xv6 is as a teaching
cat (pid=4): Q2(83%): Q1(4%): Q0(13%)
grep (pid=5): Q2(66%): Q1(13%): Q0(21%)
sh (pid=3): Q2(28%): Q1(0%): Q0(72%)
```

  - The example tells us that there were three processes involved in the execution of pipe command, i.e., `cat`, `grep`, and `sh` with `pid` of 4, 5, and 3, respectively.
  - The `cat` process spent 83% of its execution time (since the process was created in `allocproc()` until it was terminated in `freeproc()`) in the `Q2`, 4% in the `Q1`, and 13% in the `Q0`.
  - The execution statistics of `grep` and `sh` processes can be interpreted in the similar way.

- Your skeleton code should print the occupation statistics with all zero percents since the multi-queue scheduling scheme is not implemented.
- Implement the multi-queue scheduling algorithm and make every process print out the execution information at the end.

4 Submission

- In the `xv6-riscv/` directory, execute the `tar.sh` script. This script will compress the current `xv6-riscv/` directory into a tar file named after your student ID (e.g., `2020142020.tar`).
- Upload the created tar file on YSCEC. Do not rename the tar file by adding your name, project4, etc.

5 Grading Rules

- The following is a general guideline for grading the assignment. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and a grader may add a few extra rules for fair evaluation of students' efforts.

  -5 points: The tar file is renamed and includes some other tags such as a student name.

  -5 points: Program codes do not have sufficient comments. Comments in the baseline xv6-riscv do not count. You must make an effort to clearly explain what each part of your implementation intends to do.

  -10 points: An xv6-riscv scheduler fails to implement multi-queue structures for process scheduling.

  -10 points: The scheduler fails to move processes between link-list queues on system calls or timer interrupts based on the described scheduling policy.

  -10 points: The scheduler fails to execute processes in the described order, i.e., `Q2` processes and then one `Q1` process, and so forth.

  -5 points: Executions of processes do not print out reasonable runtime statistics. The irrational results possibly mean incorrect implementation of scheduling algorithm.

  -30 points: No or late submission.

  F grade: A submitted code is copied from someone else. All students involved in the incident will be penalized and given F for final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, discuss your concerns with the TA. Always be courteous when contacting the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee3535`
- Begging for partial credits for no viable reasons will be treated as a cheating attempt, and thus such a student will lose all scores for the assignment.