EEE3535-01 **Fall 2023**

Assignment 3: Scheduling

Due: Sunday, Nov. 5, 2023, 11:59PM

1 Introduction

- The objective of this assignment is to replace the round-robin process scheduler of xv6-riscv with a fair-share scheduling algorithm.
- The `scheduler()` function in `kernel/proc.c` handles process scheduling as follows.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

- In xv6-riscv, processes are allocated in the static array defined as `struct proc proc[NPROC]` in `proc.c`.
- `scheduler()` linearly searches the `proc[]` array until it finds a process whose state is `RUNNABLE`. The process becomes `RUNNING`, and the kernel context-switches to the user process.
- Returning to the next code line (i.e., `c->proc = 0`) means that the user process was interrupted.
- The scheduler finds the next `RUNNABLE` process in the `proc[]` array. Reaching the end of the array starts over in the infinite loop, implementing a round-robin scheduling policy.
- Fig. 1 outlines the sequence of process scheduling and context switching in xv6-riscv, where the CPU scheduler refers to a kernel process running the `scheduler()` function above in the infinite loop.
- The scheduler is called in when `sched()` is called. The `sched()` function is called in three places; `sleep()`, `exit()`, and `yield()` in `proc.c`.
  - `sleep()` makes the caller process `SLEEPING` (e.g., `wait()` syscall), and `exit()` makes the caller `ZOMBIE` when the process terminates. In both cases, the process is no longer runnable, and the scheduler is brought in by `sched()` to find the next runnable process.
  - `yield()` is called by `kerneltrap()` in `kernel/trap.c` on a timer interrupt. It changes the process state from `RUNNING` to `RUNNABLE` and calls `sched()` for processing scheduling.

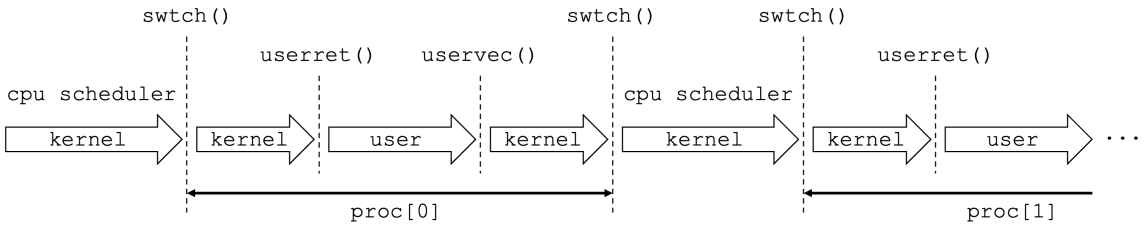

Figure 1: Timeline of the CPU execution flow. The CPU scheduler selects the next process and calls `swtch()` to context-switch to the selected process. The process runs in the kernel mode using its `kstack`. Once kernel-side operations are done, `userret()` restores the context of the user program from the process' `trapframe`. Then, the user program runs. When the user program is interrupted, the `trampoline` routine executes `uservec()` to save the context of the user program and jumps to `usertrap()`. Necessary kernel functions are executed for the process using its `kstack`. If the process calls `sched()`, the `swtch()` routine swaps registers between the process and CPU scheduler. The scheduling algorithm selects the next process to run. The rest of the timeline repeats the steps.

2 Implementation

- To start the assignment, go to `xv6-riscv` directory, download `sched.sh`, and run the script to update xv6-riscv. This assignment has no new programs or syscalls.

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

- **Ticket management:** Every process is assigned 20 tickets when it is created. If a process gives up the CPU before the end of the time slice, it earns one ticket. If a process uses up the entire time slice, it loses a ticket. The maximum ticket count is 100, and the minimum is 5.
- **Random number generation:** Use `rand()` in `kernel/random.c` to generate a random unsigned integer.
- **Drawing a winner:** Fair-share scheduling uses a random number to determine the next process. Refer to the pseudo-code example in the lecture slide to implement the scheduling algorithm in `scheduler()`.
- **Execution result:** For every process, print its scheduling stats when the process terminates in `exit()` in `kernel/proc.c`. The following example shows that `wc` has `pid = 3`, and its final ticket count is 28. It won random drawing 100% of the time for being the only runnable process. The winning rate is calculated as the number of times the process won the drawing over the number of times it was runnable and competed for scheduling.

```
$ wc README
49 325 2305 README
pid = 3: tickets = 28, winning rate = 100%
```

3 Validation

- Your assignment will be graded with the following tests.
  1. **Single interactive process:** The process is expected to earn several tickets by making system calls.

```
$ cat README
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,

...

pid = 3: tickets = 28, winning rate = 100%
$
```

2. **Single CPU-intensive process:** `prime` implements a slow search algorithm to find the N^{th} prime number. The process is expected to lose tickets for large N.

```
$ prime 20000
20000th prime number = 224737
pid = 4: tickets = 5, winning rate = 100%
$
```

3. **Several interactive processes:** Piping several commands will fight for scheduling.

```
$ cat README | grep xv6 | wc
pid = 6: tickets = 37, winning rate = 78%
pid = 8: tickets = 30, winning rate = 45%
5 54 319
pid = 9: tickets = 28, winning rate = 56%
pid = 7: tickets = 21, winning rate = 66%
pid = 5: tickets = 22, winning rate = 75%
$
```

4. **Many short processes:** `forktest` keeps forking until `fork()` fails. Processes are expected to have strong competition.

```
$ forktest
fork test
pid = 23: tickets = 20, winning rate = 100%
pid = 40: tickets = 20, winning rate = 50%
pid = 45: tickets = 20, winning rate = 33%

...

$
```

5. **Everything all at once:** `usertests` creates thousands of threads with different runtime behaviors. The program must finish with `ALL TESTS PASSED`. This program runs for several minutes.

```
$ usertests
usertests starting
pid = 73: tickets = 59, winning rate = 65%
test copyin: pid = 74: tickets = 82, winning rate = 100%
OK
test copyout: pid = 75: tickets = 21, winning rate = 100%
OK

...

ALL TESTS PASSED
pid = 72: tickets = 100, winning rate = 87%
$
```

4 Submission

- In the `xv6-riscv/` directory, execute the `tar.sh` script to create a tar file named after your student ID (e.g., `2023143535`).

```
$ ./tar.sh
$ ls
2023143535.tar  kernel  LICENSE  Makefile  mkfs  README  tar.sh  user
```

- Upload the tar file (e.g., `2023143535.tar`) on LearnUs. Do not rename the file.

5 Grading Rules

- The following is the general guideline for grading. A 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; a grader may add a few extra rules without notice for a fair evaluation of students' efforts.

  -5 points: The submitted tar file includes redundant tags such as a student name, `hw3`, etc.

  -5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

  -6 points each: The validation section has five test cases. Each failed test will lose 6 points.

  -30 points: No or late submission.

  Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get Fs for the final grade.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If no agreements are 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 the TA and instructor: `https://icsl.yonsei.ac.kr/eee3535`

- Arguing for partial credits for no valid reasons will be regarded as a cheating attempt; such a student will lose the assignment scores.