EEE3535-02 **Fall 2024**

Assignment 3: Scheduling

Due: Sunday, Nov. 3, 2024, 11:59PM

1 Introduction

- The objective of this assignment is to modify the process scheduler of xv6-riscv to work as a stride scheduler.
- To begin the assignment, go to the `xv6-riscv/` directory, download `sched.sh`, and run the script to update xv6-riscv.

```
$ cd xv6-riscv/
$ wget https://casl.yonsei.ac.kr/teaching/eee3535/sched.sh
$ chmod +x sched.sh
$ ./sched.sh
```

- To check if the update was successful, build xv6-riscv and type `sdbg on`. The program is supposed to turn the process scheduler's debugging mode on but does nothing at this moment. Terminate xv6-riscv by pressing `ctrl+x` and then `x`.

```
$ make qemu

...

qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1
-nographic -global virtio-mmio.force-legacy=false -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
$ sdbg on
$
```

- The following describes the implementation of the default round-robin scheduler in xv6-riscv.
- In `kernel/proc.cc`, processes are statically allocated in an array with `NPROC = 64`. This means that xv6-riscv can schedule only up to 64 processes at a time.

```
struct proc proc[NPROC];
```

- Process entries with the `UNUSED` state (e.g., `proc[i].state == UNUSED`) in the array indicate that they are invalid. Other entries in `SLEEPING`, `RUNNABLE`, `RUNNING`, or `ZOMBIE` states represent active processes.
- The `scheduler()` function in `kernel/proc.c` handles process scheduling in xv6-riscv 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);
```

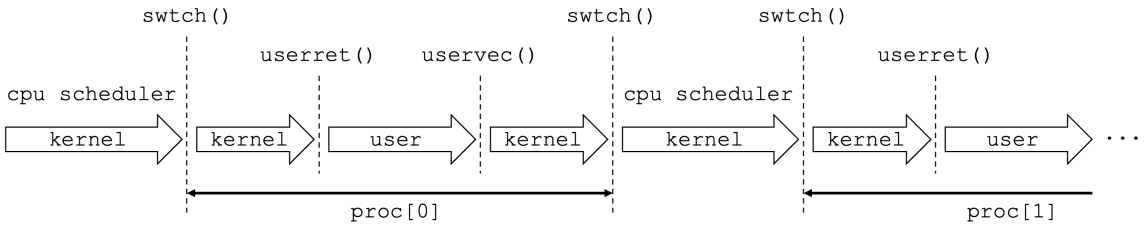

Figure 1: A CPU execution flow in xv6-riscv. The CPU scheduler selects the next process and calls `swtch()` to context-switch to the selected process. The process runs in the kernel mode using its `kstack`. Once kernel-side operations are done, `userret()` restores the context of the user program from the process's `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 the CPU scheduler. The scheduling algorithm selects the next process to run. The rest of the timeline repeats the same steps.

```
      if(p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

- The `scheduler()` function linearly scans the `proc[]` array until it finds a process whose state is `RUNNABLE`. The processing becomes `RUNNING`, and the kernel context-switches to the user process.
- Upon returning from the user process back to the `scheduler()` function, the next line of the code (i.e., `c->proc = 0`) is executed. This happens when the user process was interrupted.
- The scheduler finds the next `RUNNABLE` process in the `proc[]` array. When the end of the array is reached, it starts over in an infinite loop, implementing the round-robin scheduling policy.
- Figure 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 described above.
- The scheduler is executed when `sched()` is called. The `sched()` function is called in three places: `sleep()`, `exit()`, and `yield()` in `kernel/proc.c`.
  - `sleep()` makes the caller process `SLEEPING` (e.g., `wait()` syscall), and `exit()` makes the caller process `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 `usertrap()` or `kerneltrap()` in `kernel/trap.c` on a timer interrupt. It changes the process state from `RUNNING` to `RUNNABLE` and calls `sched()` for process scheduling.

2 Implementation

- **Ticket management**
  - When a process is first created, it is assigned five tickets. The process initialization takes place in the `allocproc()` function of `kernel/proc.c`.
  - To prioritize interactive jobs over long-running ones, a process earns an extra ticket for every 5th syscall of any kind.

- If a process occupies an entire time slice (i.e., an interval between two timer interrupts) without making any syscalls, it loses a ticket.
- To keep track of whether a process occupies a whole time slice, a pointer (or a flag) is needed to mark which process is scheduled immediately after `yield()` and if the same process is still there at the next `yield()`.
- If the process makes a syscall, the pointer (or the flag) must be cleared to indicate that it is not the case of occupying the whole time slice without syscalls.
- The maximum number of tickets per process is 10, and the minimum is 1. The ticket limits prevent processes from monopolizing the CPU or starving for scheduling.

• **Striding**

- The stride scheduler compares the current pass of all runnable processes and selects the one with the lowest pass. If multiple processes have the same lowest passes, any one of them can be chosen for scheduling.
- The pass of the selected process is updated by adding its stride to it. The stride of the process is calculated by dividing a large number by its ticket count.
- Since the number of tickets can only vary between 1 and 10, the least common multiple (LCM) of 1, 2, 3, $\cdots$ 10 can be easily found.

• **Pass**

- Simply adding strides to processes' passes makes them grow unboundedly. Consequently, old processes suffer from starvation due to large passes, while new processes dominate the CPU time.
- To prevent the problem, the pass of a selected process (i.e., the minimum-pass process) can be subtracted from that of all runnable processes.
- This will limit the range of processes' passes, solving the starvation problem and a value overflow issue.

• **Scheduling information**

- To validate the implementation, print the scheduling result of each process at the end of its execution.
- As processes are terminated in the `exit()` function of `kernel/proc.c`, this must be a good place to print the result.
- Print the stride scheduling result in the following format. Do not print other debugging messages.

```
$ sdbg on
pid = 3 (sdbg): final tickets = 5, scheduling rate = 100%

$ cat README | grep xv6 | wc
pid = 5 (cat): final tickets = 8, scheduling rate = 86%
pid = 7 (grep): final tickets = 8, scheduling rate = 78%
...
```

- The example shows that each line prints a process's PID, name, final tickets, and scheduling rate.
- Final tickets indicate the number of tickets that the process has at the end of its execution.
- The scheduling rate means how many times the process was chosen for scheduling as the minimum-ticket process when it is runnable (i.e., # of times chosen for scheduling / # of times the process was runnable).
- A user program and a syscall, both named `sdbg`, have been added to the skeleton code. Their implementations are complete, so you do not have to work on them.
- When `sdbg on` is typed in the shell, the syscall sets `sdbg = 1`, which is a globally defined flag in `kernel/proc.c`.
- On the contrary, typing `sdbg off` clears the flag.
- Using this, make the xv6-riscv kernel print the scheduling information only when the flag is on. Turning on or off the flag applies globally, not per process.

3 Validation

- Your assignment will be graded based on the following five cases, but they are not the exact and only ones that will be on the test. Your code should be able to handle other similar common-sense situations.

1. **Turning on and off `sdbg`**
   - Turning on the debugging mode immediately prints the result of the `sdbg` process itself.
   - Turning off the debugging mode silences the result.

```
$ sdbg on
pid = 3 (sdbg): final tickets = 5, scheduling rate = 100%
$ sdbg off
$
```

2. **Process with many syscalls**
   - Turn the debugging mode back on.
   - The `ls` command is busy with many syscalls to read file stats and print the information. This process is likely to reach the maximum ticket count.

```
$ sdbg on
pid = 5 (sdbg): final tickets = 5, scheduling rate = 100%
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2305
cat            2 3 32944
echo           2 4 31824
forktest       2 5 16728
grep           2 6 36376
init           2 7 32280
kill           2 8 31736
ln             2 9 31560
ls             2 10 34888
mkdir          2 11 31800
rm             2 12 31792
sdbg           2 13 32048
sh             2 14 54368
stressfs       2 15 32664
usertests      2 16 180416
grind          2 17 47544
wc             2 18 33864
whoami         2 19 31344
zombie         2 20 31160
console        3 21 0
pid = 6 (ls): final tickets = 10, scheduling rate = 100%
```

3. **Short concurrent processes**
   - With the debugging mode still on, pipe several commands as follows.
   - The short processes with syscalls are likely to earn extra tickets and contend for scheduling, resulting in scheduling rates of less than 100%.

```
$ cat README | grep xv6 | wc
pid = 8 (cat): final tickets = 8, scheduling rate = 85%
pid = 10 (grep): final tickets = 8, scheduling rate = 78%
5 54 319
pid = 11 (wc): final tickets = 8, scheduling rate = 25%
pid = 9 (sh): final tickets = 7, scheduling rate = 60%
pid = 7 (sh): final tickets = 6, scheduling rate = 75%
```

4. **Concurrent processes with different execution times**
   - `forktest` creates as many processes as possible until the `fork()` syscall fails.
   - The `forktest long` command gradually increases the runtime of child processes as it calls `fork()`.
   - The first set of processes ends with five tickets because these processes immediately terminate after they are created.
   - However, the last set of processes with longer execution times should end with only one ticket, as they lose tickets for occupying time slices without syscalls.

```
$ forktest long
fork test
pid = 13 (forktest): final tickets = 5, scheduling rate = 100%
pid = 14 (forktest): final tickets = 5, scheduling rate = 50%
pid = 15 (forktest): final tickets = 5, scheduling rate = 33%
...
pid = 71 (forktest): final tickets = 1, scheduling rate = 21%
pid = 72 (forktest): final tickets = 1, scheduling rate = 30%
pid = 73 (forktest): final tickets = 1, scheduling rate = 23%
fork test OK
pid = 12 (forktest): final tickets = 10, scheduling rate = 30%
```

5. **`usertests`**
   - The `usertests` program runs various test cases to stress the xv6-riscv kernel.
   - As the final confirmation of the stride scheduler implementation, run this program with the debugging mode turned off.
   - You must see `ALL TESTS PASSED` at the end.

```
$ sdbg off
$ usertests -q
usertests starting
test copyin: OK
test copyout: OK
test copyinstr1: OK
...
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
```

4 Submission

- In the `xv6-riscv/` directory, run the `tar.sh` script to create a tar file named after your student ID (e.g., `2024143535`).

```
$ ./tar.sh
$ ls
2024143535.tar  kernel  LICENSE  Makefile  mkfs  README  tar.sh  user
```

- Upload this tar file (e.g., `2024143535.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 extra rules without notice for the fair evaluation of students' efforts.

  -5 points: The tar file includes extra 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.

-5 points: Do not print unasked debugging messages.

-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 "F" 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 agreement is reached between you and the TA, escalate the case to the instructor to review your assignment. Refer to the course website for the contact information of the TA and the instructor: `https://casl.yonsei.ac.kr/eee3535`
- Arguing for partial credits for no valid reasons will be considered as a cheating attempt and lose the assignment score.