

CS 4375 Fall 2025

Homework 3: Priority-based Scheduler for xv6

100 points

MD Armanuzzaman

Dude Date: October 27, 2025 (11.59 PM))

For this assignment, you will implement a priority-based scheduler for xv6. xv6 currently uses a round-robin scheduler. You should keep the round-robin scheduler and add your priority-based scheduler so that one or the other may be used by setting the appropriate values in `param.h` to select the scheduler at compile-time. For process effective priorities, use a range of integers from 0 to 99, with processes allowed to have priorities 0 to 49 and effective priorities 0 to 99. The `init` process should start running with priority 0. A child process should inherit its parent's priority. A process should be able to query its priority by using the `getpriority()` system call and set its priority using the `setpriority()` system call. Your priority scheduler should always select a process with the highest priority as the next process to run. Your priority scheduler should increase the effective priority of a process by using **aging**. The “age” of a process will be determined by how long it has been waiting in the ready queue. Rather than maintaining an explicit age field in `struct proc`, it will be more efficient to store the time at which the process became ready to run, say in a field called `readytime`, and calculate the effective priority from the priority and `readytime` fields. You may decide on your own aging policy. A simple example could be the following:

```
#define MAXEFPRIORITY 99
#define AGING_DIV 25

effective_priority = min(MAXEFPRIORITY, priority + (currtime -
    readytime)/AGING_DIV)
```

Here, you can increase/decrease the `AGING_DIV` value if make the aging process faster or slower.

Important note: For this homework and the last one, we want to use just one CPU so that all processes are scheduled on the same CPU. To make this happen, in the Makefile, change `CPUS := 3` to `CPUS := 1`.

Pull the **initHW3** branch from the instructor's xv6 repository. Add instructors repository as upstream:

```
$git remote add upstream git@github.com:Tomal-kuet/xv6-riscv-labs.git
```

Verify upstream:

```
$git remote -v
origin  git@github.com:yourUSER/yourPRIVATerepo.git (fetch)
origin  git@github.com:yourUSER/yourPRIVATerepo.git (push)
upstream git@github.com:Tomal-kuet/xv6-riscv-labs.git (fetch)
upstream git@github.com:Tomal-kuet/xv6-riscv-labs.git (push)
```

And then create a hw3 branch with:

```
$git fetch upstream initHW3
$git checkout -b hw3 upstream/initHW3
```

By doing this, you will have a `getprocs()` system call and a `ps` command already implemented for you.

You can also (optional) merge in your hw2 branch so that you have the `time` command as well. If you already branched hw2 from **initHW3**, then you can just branch your **hw3** from your **hw2**. However you must add the `matmul` user program from the last homework that will be used in test programs here.

Task 1. Keeping track of and modifying priority (25 pts)

Add a priority field to `struct proc` and implement the `getpriority()` and `setpriority()` system calls. Modify the provided `getprocs()` system call and `ps` program to also retrieve and print the priority field. **Hint:** It may be beneficial to modify the `struct pstat` to get priority alongside other information for each process. Write a test program and use your modified `ps` command to check that your new system calls work and show the results. Your test program should `fork` a child to test whether parent priority is inherited by the child process and `exec ps` in the child to show all current process info.

Sample output with test (e.g., `task1.c` setting priority to 17, all other initialized to 0):

```
$ task1
parent initial priority: 0
parent after set: 17
child inherited priority: 17

Exec ps to print all process info
```

pid	state	size	ppid	priority	name
1	sleeping	12288	0	0	init
2	sleeping	16384	1	0	sh
3	sleeping	12288	2	17	task1
4	running	12288	3	17	ps

Note: A user process should not be allowed to set priority of a user process that is out of the valid range (0 to 49).

Note: Of course you need add task1 user program to your xv6.

Task 2. Implement and merge the priority scheduler (25 pts)

Add constants to `param.h` to choose between scheduling policies at compile time. Implement a priority-based scheduler that selects the highest priority process to run next. If there is a tie, select any highest priority process. You are given `task2.c` test programs to verify that your priority scheduler works as expected and show the results.

```
$ task2
Child A ran (priority = 40, CPU ticks = 101)
Child B ran (priority = 39, CPU ticks = 101)//C doesn't exist
Child C ran (priority = 40, CPU ticks = 101)//C created
Child A ran (priority = 40, CPU ticks = 102)
Child C ran (priority = 40, CPU ticks = 102)//C before B
Child B ran (priority = 39, CPU ticks = 102)
Child A ran (priority = 40, CPU ticks = 103)
Child C ran (priority = 40, CPU ticks = 103)//C before B
Child B ran (priority = 39, CPU ticks = 103)
Child A ran (priority = 40, CPU ticks = 104)
Child C ran (priority = 40, CPU ticks = 104)//C before B
Child B ran (priority = 39, CPU ticks = 104)
Child A ran (priority = 40, CPU ticks = 105)
Child C ran (priority = 40, CPU ticks = 105)//C before B
Child B ran (priority = 39, CPU ticks = 105)
```

Note that, in `task2.c` both child **A** and **C** have a priority of **40**, while child **B** has a priority of **39** (lower). All three child give up the CPU with `sleep(1)` in the loop. If your priority scheduling is implemented correctly, after the first three prints (first round) child **C** will always execute earlier than child **B**. For the first round child **C** was not created yet, and hence child **B** got the CPU as there was no other processes to execute.

Task 3. Keeping track of a process age (25 pts)

Add a `readytime` field to `struct proc` and initialize it to the current time whenever the process's state gets changed from another state to `RUNNABLE`. Add the `readytime` field to `struct pstat` in `kernel/pstat.h` and modify the `ps` command to print the process's age if the process state is `RUNNABLE`, where the age is the current time minus `readytime`. Write test cases to verify your implementation.

Task 4. Implement and merge aging policy to the priority scheduler (25 pts)

Decide on an aging policy and add aging to your priority scheduler. Evaluate your implementation based on the given test programs to verify that your aging policy works as expected and show the results.

There are two user utility test programs provided for this task: 1) **aging.c** and 2) **pexec.c**. Of course you need to add these programs to your xv6 to test.

Without proper aging policy the **aging** user program will be hogged in the infinite loop in the **hog()** function. **Write on your report why?**

With proper aging policy implemented the **aging** user program will execute the lower priority forked child at some point. **Write on your report why?**

Sample output: aging.c

```
[LOW] first scheduled at tick 3505 (wait = 25 ticks, base = 0)
```

Sample output: pexec.c pexec is a user program that takes a priority, a command, and the command's arguments as inputs and runs the command at that priority.

Your output need not look exactly like this:

```
$ pexec 0 matmul 50 & matmul 100 &
$ pexec 10 ps
pid  state      size    ppid    priority  age    name
1    sleeping    12288   0        0         N/A    init
2    sleeping    16384   1        0         N/A    sh
7    runnable    12288   5        0         0       matmul
6    runnable    12288   1        0         10      matmul
5    sleeping    12288   1        0         N/A     pexec
8    sleeping    12288   2        10        N/A     pexec
9    running     12288   8        10        N/A     ps

$ pexec 10 ps
pid  state      size    ppid    priority  age    name
1    sleeping    12288   0        0         N/A    init
2    sleeping    16384   1        0         N/A    sh
7    runnable    12288   5        0         0       matmul
6    runnable    12288   1        0         21      matmul
5    sleeping    12288   1        0         N/A     pexec
10   sleeping    12288   2        10        N/A     pexec
11   running     12288   10       10        N/A     ps

$ pexec 10 ps
Time: 256 ticks //first matmul finishes thanks to aging
```

pid	state	size	ppid	priority	age	name
1	sleeping	12288	0	0	N/A	init
2	sleeping	16384	1	0	N/A	sh
7	zombie	12288	5	0	N/A	matmul
6	runnable	12288	1	0	0	matmul
5	runnable	12288	1	0	18	pexec
12	sleeping	12288	2	10	N/A	pexec
13	running	12288	12	10	N/A	ps
\$ pexec 10 ps						
pid	state	size	ppid	priority	age	name
1	sleeping	12288	0	0	N/A	init
2	sleeping	16384	1	0	N/A	sh
14	sleeping	12288	2	10	N/A	pexec
6	runnable	12288	1	0	18	matmul
15	running	12288	14	10	N/A	ps
\$ Time: 745 ticks //second matmul finishes thanks to aging						
\$ pexec 10 ps						
pid	state	size	ppid	priority	age	name
1	sleeping	12288	0	0	N/A	init
2	sleeping	16384	1	0	N/A	sh
27	sleeping	12288	2	10	N/A	pexec
28	running	12288	27	10	N/A	ps

Turn-in procedure and Grading

Please use the accompanying template for your report. Convert your report to PDF format and push your hw3 branch with your code and report to your xv6 GitHub repository by the due date. Also, turn in the assignment on Teams with the URL of your GitHub repo by the due date. Give access to your repo to the instructor and TA.

This assignment is worth 100 points. The breakdown of points is given in the task descriptions above. The points for each task will be evaluated based on the correctness of your code, proper coding style and adequate comments, and the section of your report for that task.

You may discuss the assignment with other students, but do not share your code. Your code and lab report must be your own original work. Any resources you use should be credited in your report. If we suspect that code has been copied from an online website or GitHub repo, from a book, or from another student, or generated using AI, we will turn the matter over to the Office of Student Conduct for adjudication.