# CS/COE 1550 LAB 3
# PRIORITY SCHEDULING FOR XV6[1]

## 1. OVERVIEW

In this lab, you will implement a priority-based scheduler for xv6. To get started, download a new copy of the xv6 source code. You'll do two things in this lab:

1. You'll replace xv6's current round-robin scheduler with a priority-based scheduler.
2. You'll add a new syscall for a process to set its own priority.

## 2. PART 1: PRIORITY-BASED SCHEDULER FOR XV6

In the first part, you will replace the round-robin scheduler for xv6 with a priority-based scheduler. The valid priority for a process is in the range of 0 to 200, inclusive. The smaller value represents the higher priority. For example, a process with a priority of 0 has the highest priority, while a process with a priority of 200 has the lowest priority. The default priority for a process is 50. A priority-based scheduler always selects the process with the highest priority for execution. If there are multiple processes with the same highest priority, the scheduler uses round-robin to execute them in turn to avoid starvation. For example, if process A, B, C, D, E have the priority of 30, 30, 30, 40, 50, respectively, the scheduler should execute A, B, and C first in a round-robin fashion, then execute D, and execute E at last.

For this part, you will need to modify proc.h and proc.c. The change to proc.h is simple: just add an integer field called priority to struct proc. The changes to proc.c are more complicated. You first need to add a line of code in the allocproc function to set the default priority for a process to 50.

Xv6's scheduler is implemented in the scheduler function in proc.c. The scheduler function is called by the mpmain function in main.c as the last step of initialization. This function will never return. It loops forever to schedule the next available process for execution. If you are curious about how it works, read Chapter 5 of the xv6 book available on CourseWeb.

In this part, you need to replace the scheduler function with your implementation of a priority-based scheduler. The major difference between your scheduler and the original one lies in how the next process is selected. Your scheduler loops through all the processes to find a process with the highest priority (instead of locating the next runnable process). If there are multiple processes with the same priority, it schedules them in turn (round-robin). One way to do that is to save the last scheduled process and start from it to loop through all the processes.

---

[1] Based on http://www.cs.fsu.edu/~zwang/files/cop4610/Spring2014/project2.pdf

## 3. PART 2: ADD A SYSCALL TO SET PRIORITY

The first part adds support of the priority-based scheduling. However, all the processes still have the same priority (50, the default priority). In the second part, you will add a new syscall (setpriority) for the process to change its priority. The syscall changes the current process's priority and returns the old priority. If the new priority is lower than the old priority (i.e., the value of new priority is larger), the syscall will call yield to reschedule.

In this part, you will need to change user.h, usys.S, syscall.h, syscall.c, and sysproc.c. Review Lab 1 to refresh the steps to add a new syscall. Here is a summary of what to do in each file:

- syscall.h: add a new definition for SYS_setpriority.
- user.h: declare the function for user-space applications to access the syscall by adding: int setpriority(int);
- usys.S: implement the setpriority function by making a syscall to the kernel.
- syscall.c: add the handler for SYS_setpriority to the syscalls table using this declaration: extern int sys_setpriority(void);
- sysproc.c: implement the syscall handler sys_setpriority. In this function, you need to check that the new priority is valid (in the range of [0, 200]), update the process's priority, and, if the new priority is larger than the old priority, call yield to reschedule. You can use the proc pointer to access the process control block of the current process.

## 4. BONUS (2 POINTS)

Note that (a) in XV6, the scheduler has hard affinity, and (b) the highest process is running in each core. If the scheduler preempts that process and there are no other processes of the same priority or higher priority, it'd be wasteful to preempt and resume the running process. To optimize the scheduler, the scheduler can avoid "yield"ing, and just continues to execute that highest priority process.

- Use a flag to check whether we need to yield before we call yield() in trap.c
- Because of hard affinity and multicore systems, you will need one flag per core; cpuid() return the CPU id where the code is running.
- Do you need to introduce a mutex for this flag?

## 5. DELIVERABLES

Submit to GradeScope your modified source code of xv6 as a gzip compressed tarball. Include in the tarball all files necessary for a successful build! Your submission will be graded by compiling and running it and reviewing the source code.

- Please make sure your source code can compile. Absolutely no credit if it does not compile.
- Please don't include the binary files. Do a **make clean** before submission.
- Assuming that you are inside the xv6 source code folder (named xv6-public), type the following on linux.cs.pitt.edu:

- make clean
- cd ..
- tar cfz USERNAME_lab3.tar.gz xv6-public

- Then upload the gzipped tarball