CSC501 Spring 2018

PA 1: Process Scheduling

Due: Feburary 12 2018, 11:59 pm

1. Objectives

The objectives of this lab are to get familiar with the concepts of process management like process priorities, scheduling and context switching.

2. Readings

The xinu source code (in sys/), especially those related to process creation (create.c,), scheduling (resched.c, resume.c suspend.c), termination (kill.c), priority change (chprio.c), as well as other related utility programs (e.g., ready.c) and system initialization code (initialize.c) etc.

3. What to do

You will also be using the `csc501-lab0.tgz` you have downloaded and compiled by following the lab setup guide. But you need to rename the whole directory to csc501-lab1.

In this Lab you will implement two new scheduling policies that will avoid *starvation* between processes. At the end of this lab you will be able to explain the advantages and disadvantages of the two new scheduling policies.

The default scheduler in Xinu will schedule the process with the higher priority. Starvation is produced in Xinu when there are two or more processes eligible for execution that have different priorities. The higher priority process gets to execute first which results in lower priority processes never getting any CPU time unless the higher priority process ends.

The two scheduling policies that you need to implement, as described below, should address this problem. Note that for each of them, you need to consider how to handle the Null process, so that this process is selected to run when and only when there are no other ready processes.

For both scheduling policies, a valid process priority value is an integer between 0 to 99, where 99 is the highest priority.

1) Random Scheduler

The first scheduling policy is a **random scheduler**. This scheduler will choose at random a process eligible for execution. The probability of choosing a specific process will be proportional to its priority. For example, assume that there are three processes eligible for execution P1, P2, and P3 (each of them is either in the ready list or is the current process) with priorities 30, 20, and 10. The *total-sum* of those priorities is 60. The goal of the random scheduler is to allocate the CPU with a probability of 30/60 to P1, 20/60 to P2, and 10/60 to P3.

To accomplish this, each time a rescheduling happens, the scheduler will generate a random number between 0 and *total-sum* minus 1 (0 to 59 in this case). If the random number is less than the *priority* of the first process in the list of processes eligible for execution, then the scheduler schedules this process for execution. Otherwise, the random number is decreased by *priority* units and it goes to the next process in the list and so on. For example assume that the random number generated was 38, and the first process in the list is P1. As

38 is not less than 30, we decrease the number 38 by 30 and we go to the next process in the list. As 8 is less than 20, then the scheduler will choose P2 for execution. In your implementation you may use the *srand()* and *rand()* functions, as implemented in ./lib/libxc/rand.c.

2) Linux-like Scheduler (based loosely on the 2.2 Linux kernel)

This scheduling algorithm tries to loosely emulate the Linux scheduler in 2.2 kernel. In this assignment, we consider all the processes "conventional processes" and uses the policies regarding `SCHED_OTHER` specified in the above document. With this algorithm, the CPU time is divided into *epochs*. In each epoch, every process has a specified time quantum, whose duration is computed at the beginning of the epoch. An epoch will end when all the **runnable** processes have used up their quantum. If a process has used up its quantum, it will not be scheduled until the next epoch starts, but a process can be selected many times during the epoch if it has not used up its quantum.

When a new epoch starts, the scheduler will recalculate the time quantum of **all** processes (including blocked ones). This way, a blocked process will start in the epoch when it becomes runnable again. New processes created in the middle of an epoch will wait till the next epoch, however. For a process that has never executed or has exhausted its time quantum in the previous epoch, its new quantum value is set to its process priority (i.e., `quantum = priority`). A quantum of 10 allows a process to execute for 10 ticks (10 timer interrupts) within an epoch. For a process that did not get to use up its previously assigned quantum, we allow part of the unused quantum to be carried over to the new epoch. Suppose for each process, a variable `counter` describes how many ticks are left from its quantum, then at the beginning of the next epoch, `quantum = floor(counter/2) + priority`. For example, a counter of 5 and a priority of 10 will produce a new quantum value of 12.

During each epoch, runnable processes are scheduled according to their *goodness*. For processes that have used up their quantum, their goodness value is 0. For other runnable processes, their goodness value is set considering both their priority and the amount of quantum allocation left: `goodness = counter + priority`. Again, round-robin is used among processes with equal goodness.

The priority can be changed by explicitly specifying the priority of the process during the `create()` system call or through the `chprio()` function. Priority changes made in the middle of an epoch, however, will only take effect in the next epoch.

An example of how processes should be scheduled under this scheduler is as follows:

If there are processes P1,P2,P3 with time quantum 10,20,15 then the epoch would be equal to 10+20+15=45 and the possible schedule (with quantum duration specified in the braces) can be: P2(20), P3(15), P1(10), P2(20) ,P3(15), P1(10) but not: P2(20), P3(15), P2(20), P1(10)

Other implementation details:

1. `void setschedclass(int sched_class)`
   This function should change the scheduling type to either of the supplied RANDOMSCHED or LINUXSCHED

2. `int getschedclass()`
   This function should return the scheduling class which should be either RANDOMSCHED or LINUXSCHED

3. Each of the scheduling class should be defined as constants
   ```
   #define RANDOMSCHED 1
   #define LINUXSCHED 2
   ```

4. Some of source files of interest are: create.c, resched.c, resume.c, suspend.c, ready.c, proc.h, kernel.h etc.

5. Test files test1.c, test2.c, test3.c and test4.c demonstrates how to create processes and call `chprio()` to

change the priorities.

6. You can use testmain.c as your test case or you can create your own. The goal is to illustrate and understand the difference of these two scheduling policies through some examples.

4. Additional Questions

Write your answers to the following questions in a file named Lab1Answers.txt(in simple text). Please place this file in the sys/ directory and turn it in, along with the above programming assignment.

1. What are the advantages and disadvantages of each of the two scheduling policies? Also, give the advantages and disadvantages of the round robin scheduling policy originally implemented in Xinu.
2. Assume that there are three processes P1, P2, P3 that will run forever without blocking. We want to allocate the following CPU times to each of them: 20% to P1, 30% to P2, and 50% to P3. Which priorities do we need to set to P1, P2, and P3 to accomplish this if we want to use the Random scheduler? Assume that those processes are the only ones running in the machine. Could you give a generalization of your result to *n* processes? Explain.
3. Describe the way each of the schedulers affects the NULL process.

Turn-in Instructions

Electronic turn-in instructions:

1. go to the `csc501-lab1/compile` directory and do `make clean`.
2. go to the directory of which your `csc501-lab1` directory is a subdirectory (NOTE: please do not rename `csc501-lab1`, or any of its subdirectories.)

   e.g., if `/home/csc501/csc501-lab1` is your directory structure, goto `/homes/csc501/`

3. create a subdirectory TMP (under the directory `csc501-lab1`) and copy all the files you have modified/written, both .c files and .h files into thedirectory.
4. compress the `csc501-lab1` directory into a tgz file and use Wolfware's Submit Assignment facility. Please only upload one tgz file.

   ```
   tar czf csc501-lab1.tgz csc501-lab1
   ```

ALL debugging output should be turned off before you submit your code.

Back to the CSC501 web page