CSC501 Spring 2014
PA 1: Process Management and Scheduling
Due: 2/4, 11:59pm

1. Objectives

The objectives of this lab are to get familiar with the concepts of process management like process monitoring, 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), cpu-clock&ticks and quantum related(clkinit.c, clkint.S),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 PA0 lab setup guide. But you need to rename the whole directory to csc501-lab1.

In this Lab you will implement a ps like command and 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) Implement a function similar to linux's ps shell command

The function should list all the processes that are alive in the system. For each process, you need to output the key information associated with the process including *process name*, *PID*, *priority*, *process execution status* (running, waiting, etc), *the number of cpu ticks the process has used for execution*, *the stack size of the process*. You need to sort all the processes using the descending order of CPU ticks. If two processes have the same CPU ticks, order them based on the descending order of PID. For process's status, it should be one of the status listed in "proc.h"; for the actual printed info, you could just print "PRCURR", "PRREADY".
The function's signature should be "void ps()".
The first line of the output should print "ProcessName PID Priority Status Ticks StackSize". The following lines should output the information of all the processes with one line per process. The fields should be separated by spaces.

2) Random Priority Scheduling

In this problem, you are required to implement a simple random priority scheduling algorithm. We maintain three queues in the system. Processes whose priorities are between 99-66 are placed in the first queue; processes whose priorities are between 65-33 are placed in the second queue; processes whose priorities are 32-0 are placed in the third queue.
When needing to choose a process to be executed, your program should generate a random number and select which process to run based on the value of the random number. Your random selection algorithm should satisfy the following rule: the first queue has a 50% chance to be selected; the second queue has a 30% change to be selected; the third queue has a 20% chance to be selected. Processes in every queue are ordered by their arrival time. Newly arrived process is placed at the end of the queue. After a process finishes its quantum (in this problem, please use the default quantum value in Xinu), put it at the end of the queue. If a process just finishes its quantum, and a new process arrives at the same time in the same queue, insert the new process first, then put the process which just finishes its quantum at the end of the queue.

Generally, the procedure of selecting a process is illustrated as:

a) generate a random value
b) select a queue based on the random value
c) take a process from the head of the queue to execute

In your implementation you may use the srand(seed) and rand() functions, as implemented in ./lib/libxc/rand.c.

3) 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 (see http://oreilly.com/catalog/linuxkernel/chapter/ch10.html). 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 (conditions like, change from another scheduling algorithm), 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 priority 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)

Note:
If you need a header file, please rename it to lab1.h

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. testmain1.c is a simple sample test case for your program.
The first part is to test the ps function. The output is something like:

Name	PID	Priority	Status	Ticks	StackSize
main	49	20	PRCURR	70	4096
prnull	0	0	PRREADY	10	0

The second part is to test random-priority-scheduling. The output is something like:

```
Start... A
Start... B
Start... C
Test1 RESULT: A = 245803, B = 360302, C = 579612 (2 : 3 : 5)
```

The third part is for the linux-like-scheduling. Process A, B, C and the main function will have a loop to print a character. The output should be something like:

```
MCCCCCCCCCCCCCCBBBBBBBBBMMMACCCCCCCCCCCCCCCBBBBBBBMMMACCCCCCCCCCCCCCBBBBBBBMMMACCCCCCCCCCCCCCBBBBBBBMMM
```

Note for the output:
1) For the ps function, the cpu-tick doesn't have to be the same as the sample.
Also, if you create more processes to test your code, because while you sort the processes according to their cpu ticks, the sorting function might be preempted by other process whose cpu-ticks will increase, the output might not be strictly descendingly ordered by cpu ticks. This is a acceptable. As long as the processes are GENERALLY ordered by cpu ticks, it's fine.
2) For the random-priority scheduling, your output doesn't have to be the same as the sample. As long as the percentage is generally 2:3:5, it's fine.
3) For the linux-like scheduling, your output doesn't have to be the same as the sample. The key idea behind this output is that, process with higher priority will be executed more frequently and finish early.

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. 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, go to `/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 the directory.
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.