CSC501 Spring 2009

Lab 1: Process Scheduling

Due: Feb. 10th 2009, 11:45PM

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

1. 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 lab0-spring09.tgz you have downloaded and compiled by following the lab setup guide. But you need to rename the whole directory to csc501-lab1.

This lab is divided into two parts. In the first part you will fix some flaws that are in *chprio()*. In the second part 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.

Starvation is produced in Xinu when we have two or more processes eligible for execution that have different priorities. The scheduling invariant in Xinu assumes that at any time, the highest priority process eligible for CPU service is executing, with round-robin scheduling for processes of equal priority. Under this scheduling policy, the processes with the highest priority will always be executing. As a result, all the processes with lower priority will never get CPU time. You may test this condition by running *test1.c* and *test2.c* .

Part 1: Run the test programs *test3.c*, and *test4.c*. Check if the function *chprio()* executes correctly. Fix any bug(s) that you find and verify that the test files run correctly with your modifications.

Part 2: Now you will be implementing two new scheduling policies that will avoid *starvation*.

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.

The second scheduling policy is a **proportional sharing policy.** First the policy will be explained and then we will give pointers as to how to adapt this to XINU.

Every process has a scheduling parameter called the *rate.* For a process i let us denote the *rate* as Ri. Every process *i* has a *priority value* Pi. Initially, all the processes start with a *priority value 0* (Pi = 0). Whenever a rescheduling occurs, the *priority value* Pi of the **currently** running process is updated as follows,

Pi : = Pi + (t x 100 / Ri).

where **t** is the CPU ticks consumed by the process since the last update of Pi. Ri is a percentage value and takes values between 0 and 100.

Now the scheduler schedules a runnable process with the *smallest* Pi. Whenever a process is scheduled the first time or is scheduled after blocking, its Pi value is updated as follows,

Pi : = max (Pi, T).

where T is the number of CPU ticks that have elapsed since the start of the system. A safe assumption of the number of the CPU ticks would be number of QUANTAMs (as defined in Xinu), that have elapsed so far since the start of the system.

Intuitively, you can think of this policy as one that gives the processes some guarentees about their CPU share. If a process has a rate Ri, then it is guarenteed atleast Ri percent of CPU time, provided the sum of all Ri values is less than 100. As the CPU usage of a process increases, its Pi value also increases depending on its Ri. If you have a large Ri, then your Pi increases more slowly and hence giving you a larger share of the CPU. Thus, Ri can be considered as a *share* of the CPU for the process *i.* The second formula can be intuitively understood from the following example.

Consider two processes A, B starting at time 0 and running continuously with rate 50 and 40 respectively. Let us say that another process C is created after 100,000 ticks with rate 10. C will start executing with a 0 priority and hence will hog the CPU for a very long time and processes A, B have to wait for long to get the CPU back and would not enjoy their share of the CPU till all the Pi values level off On the other hand, if the process C starts with a priority 100,000 instead of 0, then it will run only for a short amount of time before yielding the CPU back to A and B.

According to this policy, the processes are scheduled in increasing order of their priority (ie) the process with the lowest priority value Pi will be scheduled first. But, Xinu works in exactly the opposite way (ie) the process with the highest priority is scheduled. In order to overcome this mismatch, we can maintain an internal variable Pi for every process which will contain the *priority value.* Let us denote the Xinu process priority of a process *i* to be PRIOi. Then PRIOi can be calculated as **PRIOi = MAXINT - Pi.** As Pi increases, PRIOi decreases and the process will get lesser share of CPU.

To summarize, at every reschedule operation the scheduler does the following,

* the Pi value of the current process is modified and its PRIOi value updated.
* the scheduler chooses for execution the process with the highest priority, choosing from the processes in the ready list *and* the current process.

Also, whenever a process is scheduled for the first time or immediately after blocking, then the Pi value is changed as indicated above and the PRIOi is updated to reflect the change. (You, have to identify when and how to change Pi)

The two scheduling policies should work simultaneously in Xinu. The scheduling policies are on a per process basis. Whenever a process is created in Xinu, it should have either of the two policies. This can be implemented by adding an additional parameter to the Create system call. The new prototype of the create system call would be,

SYSCALL create (int *procaddr, int ssize, int priority, int spolicy, char *name, int nargs, long args);

The parameter spolicy would be either RANDOMSCHED or PROPORTIONALSHARE. If otherwise, SYSERR should be returned. In the case of RANDOMSCHED, the parameter priority would indicate the Xinu process priority value. But in the case of proportional share it would indicate the 'rate' of the process. The internal priority value P should be made zero and hence the Xinu process priority value should be made MAXINT.

The following macros are defined in proc.h

```
#define RANDOMSCHED          1
#define PROPORTIONALSHARE    2
```

To make the two scheduling policies work simultaneously, implement the following in *resched()* :

- If there are processes with RANDOMSCHED policy in the ready list, then a random number between 0 and 99 will be generated. If it is less than a variable *rsfrequency* (that you will define), then a process of policy of RANDOMSCHED will be chosen as described above. Otherwise a process of policy PROPORTIONAL SHARE will be chosen. The variable *rsfrequency* will be set using the procedure *sfrequency(rsfrequency)* that you will also define. *rsfrequency* will have the default value of 50.

Note, that the behavior r of chprio(pid, priority) will also change depending upon the scheduling policy of the process. If the process is a randomly scheduled process, then the parameter priority is the usual Xinu process priority. In the case of a proportional share process, the parameter priority represents the 'rate' of the process.

---

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: 50% to P1, 30% to P2, and 20% 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. We want to get the same effect as that of a proportional share scheduler with the Random scheduler for a given set of processes and their 'rate's. Can we? Explain.
4. Describe the way each of the schedulers affects the NULL process.
   For the proportional share scheduler, try the following scenario. Create two processes A and B with rates 0.7 and 0.3 respectively. Both processes should basically be continuously computing (except as noted below) and periodically printing some output to the console so that you can track their progress. Let both processes run for 20 seconds. At 20 seconds, let A sleep for 60 seconds, while allowing B to continue running. When A wakes up at 80 seconds, what happens? What does that tell you about the fairness of the scheduler, and why might it cause problems for certain applications? How might you adapt the algorithm to solve the problem?

If you have any questions, post them to the class **message board.** ***Please avoid mailing your questions to TAs email accounts.***

Turn-in Instructions

Electronic turn-in instructions (**Make sure that your code compiles and that you have turned off all your debugging output!):**

i) go to the csc501-lab1/compile directory and do "make clean".

ii) 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

iii) 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.

iv) 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

You can write code in main.c to test your procedures, but please note that when we test your programs we will replace the main.c file! Therefore, do not put any functionality in the main.c file.

Back to the CSC501 web page