CS50300: Operating Systems
Lab2 Answers

Zichen Wang
wang4113@purdue.edu

September 26, 2018

1 Monitoring CPU usage and waiting time of processes

Please set 'XINUSCHED' in process.h to 0 (legacy mode) when testing.

1. We will spawn a new process myTestProcA() with higher priority from main() and monitor its gross CPU usage.

   At the beginning of myTestProcA(), we define a time stamp named *time_stamp* to record the approximate beginning time of a process. Then, we run a loop to consume some CPU cycles. Before myTestProcA() goes to sleep, we will print its gross CPU usage by $pgrosscpu + currproctime$. We also print a time by $clktimemilli - time\_stamp$. $pgrosscpu + currproctime$ and $clktimemilli - time\_stamp$ should be approximately equal. If they are not exactly equal, $pgrosscpu + currproctime$ is at most 1 ms larger than $clktimemilli - time\_stamp$ because of the context switching to myTestProcA() before we record the *time_stamp*.

   After that, myTestProcA() will go to sleep for 200 ms, and main() will continue to run. We will print the *pgrosscpu* in main(). The print in main() is typically not the same as the print in myTestProcA(), because kprintf() will consume 1 or more milliseconds to print this information and the scheduler will take several milliseconds to insert myTestProcA() into the sleeping list. Therefore, the print in main() should be larger than the print() in myTestProcA(). In my case, the difference is about 2 ms.

   After the myTestProcA() wakes up, it will continue to run a loop to consume some CPU cycles. Before the process ends, we print $pgrosscpu + currproctime$ and $clktimemilli - time\_stamp - 200$ to check whether they are approximately equal.

2. We will spawn two user processes to monitor waiting time. myTestProcB() will first run a loop, then sleep for 10 ms, and continue to run a loop. It will print time information before terminates. The first process spawned from main() has priority 19, and the second one has priority 18. main() will sleep after resuming these processes. Hence, we would like to see the tot waiting time of the second process should be approximately equal to the gross CPU usage of the first process. The tot waiting time of the first process should be zero.

The waiting count of the first process should be 2. One is after creating, and another is after waking up. The waiting count of the second process should be 3. We will see that when the first process is sleeping, the second process will be able to run. However, as the sleeping time is very short, the first process will grab the CPU after waking up since it has a higher priority. Thus, the second process will have to become ready state and the waiting count of the second process should be one more than the first process.

In my case, the tot waiting time of the second process is 1 millisecond larger than the gross CPU usage of the first process, since the gross CPU usage of the first process we see is not the tot running time of the first process. There are kprintf() and resource release between that print and the termination of the first process. The tot waiting time of the first process is 0.

2 Implementing XINU system calls via synchronous interrupt

Since the hardware will not disable the interrupt automatically when running a trap instruction, we should disable the interrupt manually in '_Xint33' just before calling 'getpid()' system call handler, and enable interrupt after returning from the handler.

3 Performance evaluation of R3

Parameters: $LOOP1 = 60$, $LOOP2 = 1480000$, $IOSLEEP = 30$.

Please set 'XINUSCHED' in process.h to 1 (R3 mode) when testing.

1. Scenario 1: create 8 app processes that are all CPU-bound. Table 1 shows the results.

PID	**proctype**	**clktimemilli (ms)**	**gross CPU usage (ms)**	**average waiting time (ms)**
4	0	17439	2005	172.839506
5	0	17447	2005	172.938271
6	0	17455	2005	173.37037
7	0	17463	2005	173.135802
8	0	17472	2006	173.234567
9	0	17480	2006	173.333333
10	0	17488	2006	173.432098
11	0	17496	2005	173.543209

Table 1: create 8 app processes that are all CPU-bound

2. Scenario 2: create 8 app processes that are all I/O-bound. Table 2 shows the results.
3. Scenario 3: create 4 app processes that are CPU-bound and 4 app processes that are I/O-bound. Table 3 shows the results.

In scenario 1, the CPU usage and average waiting time of all 8 processes are approximately the same, since all processes have the same code and the same unchanged priority. Moreover, the order

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
12	1	36441	1999	67.11976
19	1	36789	2006	71.627329
16	1	37069	2003	74.778481
18	1	37223	2003	72.981707
14	1	37290	2009	70.350877
15	1	37340	2006	75.49689
13	1	37540	2007	76.285714
17	1	37661	2010	74.698795

Table 2: create 8 app processes that are all I/O-bound

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
24	1	55988	1980	72.60402
25	1	56101	1981	72.326666
26	1	56172	1982	72.793333
27	1	56260	1982	73.380000
21	0	57378	2031	144.541666
23	0	57407	2031	144.843750
20	0	57530	2030	141.707070
22	0	57534	2030	141.747474

Table 3: create 4 app processes that are CPU-bound and 4 app processes that are I/O-bound

of termination is the same as the order of creation. It is because all processes are running orderly. When a process uses up its time slice, it will be put at the end of the ready list.

In scenario 2, the CPU usage and average waiting time of all 8 processes are also approximately the same, but the variance is a little bit larger than scenario 1. It is because the priority will increase when a process encounters sleepms(). When that process wakes up, it will grab the CPU from a process with lower priority. However, when a process uses up its time slice, its priority will decrease to normal. The change of priority will affect the waiting count and waiting time, and the order of termination is kind of random. It is really hard to figure out what happened exactly in this scenario, but the time is approximately the same under these parameters.

In scenario 3, as the specification said, the measured performance of 4 CPU-bound processes are approximately the same, and the same goes for the 4 I/O-bound processes. The gross CPU usage of the CPU-bound processes is higher than the I/O-bound processes, while the average waiting time is lower than the CPU-bound processes. The reason why the I/O-bound processes have lower average waiting time is that they will receive higher priority after sleeping a while. Moreover, the waiting count of the I/O-bound processes is more than the CPU-bound processes, since a I/O-bound process will become ready after waking up from sleep.

During the tuning of these parameters, I found that in scenario 2 if the running time of the inner loop is shorter than the time slice (25 ms) and the sleeping time is shorter than the running time of the inner loop, some I/O-bound processes may be starving. Let's consider a situation where the inner loop takes 20 ms and the sleeping time is only 1 ms. The first process will run the inner loop first and then go to sleep for 1 ms. Its priority will increase. The second process will run when the first process sleeps. However, after the first process wakes up, it will grab the CPU immediately and continue to run the next inner loop. The second process will have to stop because of lower priority. Hence, the second and other processes will be starving.

Therefore, my tuning strategy is letting the running time of inner loop slighter larger than the time slice (25 ms) and the sleeping time is approximately equal to the running time of inner loop.

4 Bonus problem

Every change that makes on legacy code has been provided with comments. The new codes added in the kernel system have comments specifying my user name, date and the purpose.