CSE 410 - Computer Systems
Homework 6

Assigned: Friday, May 21, 2004

Due: Friday, May 28, 2004
At the start of class

Name: Solution DWJ 28-May

Student #: _____

1. On most modern systems, there are usually several "ready queues" and several "wait queues." None of the tasks (threads) on any of these queues is actually executing instructions. For the following questions, describe a set of circumstances that would cause the given scenario to take place. There are many such possibilities.

a. Describe a set of circumstances under which a task would move from a wait queue to a ready queue. Include a description of the particular wait queue before the transition, the event that triggers the transition and a description of the particular ready queue after the transition.

One very common example is a thread that is doing disk I/O. The thread is placed on the disk I/O wait queue when it initiates a disk read. The event that triggers the transition is completion of the read (all data is in memory) and the thread is moved to the ready queue that is appropriate for its priority. It may or may not run immediately, depending on the priorities of any other ready threads.

b. Describe a set of circumstances under which a task would move from a ready queue to being the running task, executing instructions on a CPU. Include a description of a particular ready queue before the transition, the event that triggers the transition, and a description of how the scheduler picks this particular task to run next.

A task moves from ready to running when it is the first thread in the priority queue with the highest priority that has any threads on it.

For example, the priority 16 ready queue might have two threads on it, both of which recently completed a disk read. Assume the running thread is also priority 16. When the running thread has run for the length of time defined by its quantum, it goes to the end of the P16 queue, and the thread at the beginning of the queue is picked by the scheduler to run next.

2. For scheduling purposes, a useful characterization of tasks is often that they are "I/O bound" or "CPU bound".

a. Give an example of a desktop PC task that is likely to be I/O bound often. Why does this task fit this description?

A task that reads data from any I/O continuously is likely to be I/O bound because if the amount of actual work done per read is low. Most of the time the task will be waiting for the I/O to complete and will not actually doing anything.

Most spreadsheet applications would be an example of this. Most of their time is spent waiting for the user to type something. The amount of CPU time actually doing anything is usually fairly low.

b. Give an example of a desktop PC task that is likely to be CPU bound for relatively long periods. Why does this task fit this description?

A task that does little I/O relative to long periods of calculation is CPU bound. One example is a spreadsheet application that performs very large "what if" - style analysis of a large (but in-memory) data set. The calculation period could consume a significant amount of time, while requiring little or no I/O.

3. The lecture dated May 17 is about scheduling on Windows 2000. On slide 9 there is a description of theWin2K priority structure. Refer to that page for information on process priority classes.

On the next page of this homework there is a snapshot of a perfmon4 graph showing the folding@home calculation process running at the same time as a virus scanner program. The dotted line that bounces around 70 is the percent CPU being used by thread 4 of the scan32 (virus scanner) process. The heavy line that bounces around 25 is the percent CPU being used by thread 3 of the Fah_core (folding@home) process. The horizontal line at 8 is the current priority for thread 4 of scan32, and the horizontal line at 1 is the current priority of thread 3 of Fah_core.

a. What is the most likely process priority class for scan32? Why?

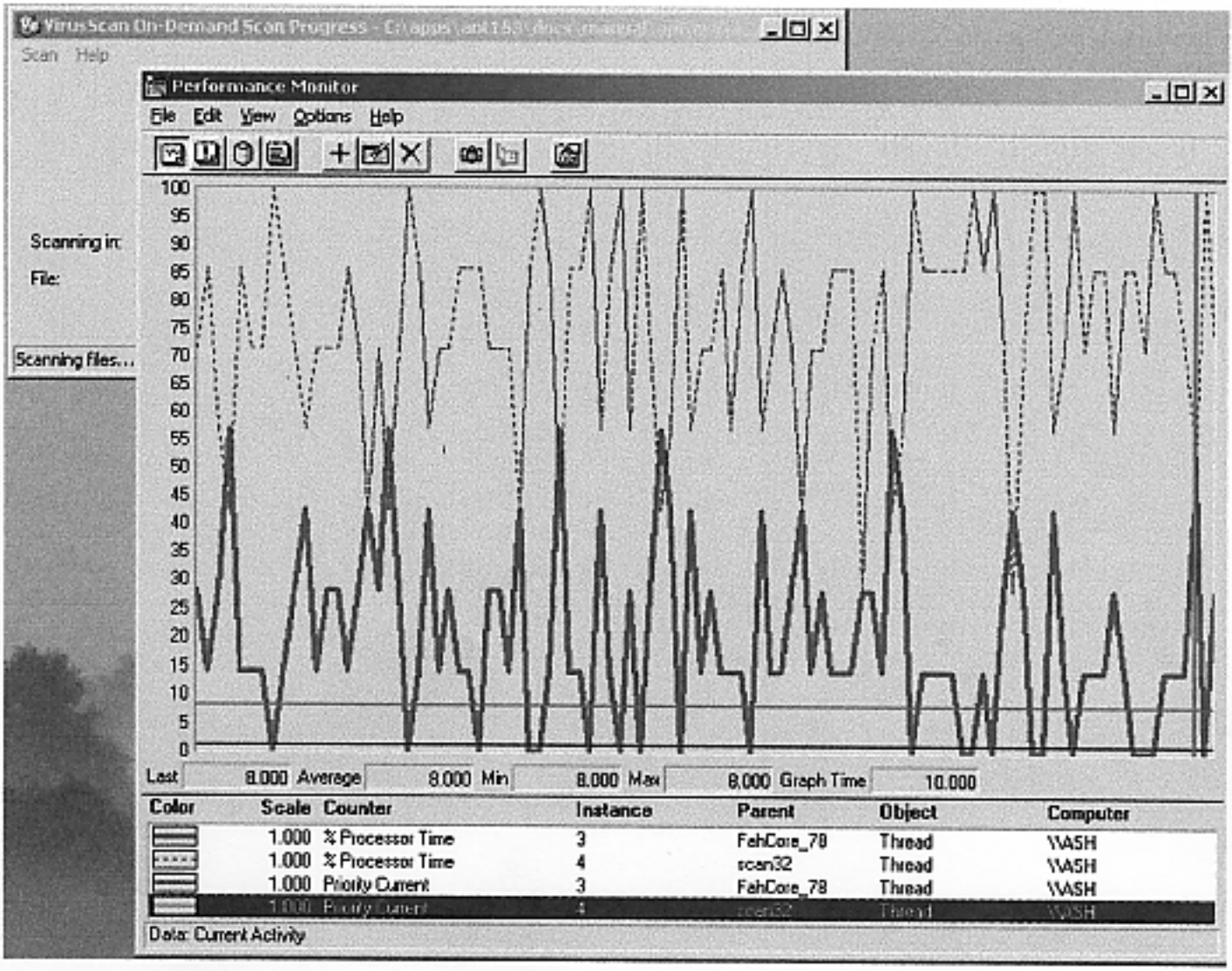
Normal. 8 is the normal relative thread priority for the Normal process priority class.

b. It is true that the Fah_core thread is CPU-bound. In fact, if there is no other activity on the machine, it will absorb 100% of the available CPU time. Would you say that the scan32 thread is also CPU-bound, or is it IO-bound? Why?

It's most likely I/O bound because FAH is getting a significant amount of time to run. Since scan32 is higher priority, it would get all the CPU time it wanted if it were CPU. It doesn't use it all, therefore it's probably waiting for disk reads much of the time.

c. The priority of the Fah_core thread never gets to be higher than the priority of the scan32 thread, and yet the Fah_core thread gets significant amounts of CPU time. Using the terminology of the running thread, wait queues, and ready queues, explain how this could happen.

scan32 is probably spending much of its time on the disk wait queue, waiting for a disk read to complete. While scan32 is waiting, the Fah thread comes off the low priority ready queue and becomes the running thread.


Virus Scan On-Demand Scan Progress
Scan Help
Scanning in:
File:
Scanning files...
Performance Monitor
File Edit View Options Help
100 95 90 85 80 75 70 65 60 55 50 45 40 35 30 25 20 15 10 5 0
Last 8.000 Average 8.000 Min 8.000 Max 8.000 Graph Time 10.000
Color Scale Counter Instance Parent Object Computer
1.000 % Processor Time 3 FehCore_78 Thread \\ASH
1.000 % Processor Time 4 scan32 Thread \\ASH
1.000 Priority Current 3 FehCore_78 Thread \\ASH
1.000 Priority Current 4 scan32 Thread \\ASH
Data: Current Activity

4. Consider the slides in the May 17 lecture that describe various scheduling scenarios (slides 18 to 20). Note that the thread that comes off the ready queue to be run is always the one at the head of the queue. For this question, the term "scheduling events" means statements like "move to wait queue for device read", "quantum exhausted", "preempt by higher priority task" and so on.

a. Are the threads in these examples "dynamic" threads or "real time" threads?

dynamic

b. Starting from the situation shown in slide 19, where thread B is running, describe a set of scheduling events under which thread A will get a chance to run.

B has priority 12, waiting thread A has priority 14. When the wait condition for B is satisfied (for example, a disk read is completed) then A becomes ready and because it is higher priority than B, it preempts B and is made the running thread.

c. Starting from the situation shown in slide 20, where thread C has just started running, describe a set of scheduling events under which thread E will eventually get to run.

The only way E will get a chance to run is if B, C, and D all block or terminate, or if the priority of E is boosted higher than 12.

One scenario is that each of B, C, and D all do disk reads in rapid succession. They will all move to the disk wait queue, leaving E the first candidate to run on the highest priority ready queue. The OS will schedule E to run

5. Coordinated access to shared state variables is the key issue in synchronization of multiple threads.

a. On a preemptively scheduled system, is the following `if` statement an "atomic operation", or is it possible for a thread to be interrupted while executing the statement?

```
if (j < k) j++;
```

No, not atomic. There are several assembly language instructions needed to implement this, and they could be interrupted at any time.

b. Is the "`j++`" portion of the statement an "atomic operation" or is it possible for the thread to be interrupted while executing the statement?

No, probably not atomic on most machines, although some machines might implement an atomic increment.

c. Assuming that j and k are shared state variables, show how you would use a lock with this code to prevent corruption due to a context swap at an inconvenient moment. You can use the same notation conventions that I used in the Synchronization lectures, namely `lock->acquire()` and `lock->release()`. This is not a trick question; the answer is simple.

```
lock -> acquire();
if (j < k) j++;
lock -> release();
```

6. Consider the lecture "Synchronization Part 2". On slide 12, there is an example of how Push() and Pop() functions can be implemented to provide coordinated, multi-thread access to a single shared stack.

a. Notice that the Push procedure uses `condition->signal(lock)` to alert any waiting thread that there is something on the stack. If that call were replaced with `condition->broadcast(lock)`, do you think that the procedures would still work correctly in a multi-threaded application?

Yes

b. Describe what would happen when a broadcast took place. If you think it won't work correctly, describe an example where it fails. If you think it will work correctly, describe why it won't fail.

When the broadcast happens, all the threads that are in the condition→wait() state are made ready to run. One of them will be given the lock, all the rest of the threads will be blocked waiting for the lock. The thread with the lock will execute the rest of the Pop code, eventually releasing the lock. One of the blocked threads will be given the lock, it will run, but it will find in the "while" statement that the stack is now empty again. So it will go back into the wait() mode, releasing the lock. Another blocked thread will get the lock and repeat the process, and so on until all the threads have had a chance to look and have gone back to sleep with wait().