Lecture 14
Concurrent Programming

17th October 2002

Review

Covered: 3 steps in handling every kernel entry: fate of current thread, choose next thread, switch threads. Fate of current thread depends on the operation it is performing – return to ready queue, join some wait queue. How do we choose the next thread: FIFO queueing (covered last time), priority scheduling, dynamic priority scheduling, weighted fair scheduling this time.

Strict Priority Scheduling

In strict priority scheduling, each thread is given a priority from some ordered set. (Often lower numbers are treated as higher priorities, but that is system dependent.) In a strict priority scheduler the dispatcher chooses the next task from the highest priority non-empty ready queue. Thus, the dispatcher ensures

priority-dispatching-property (UP version) the current running thread has priority at least as high as any runnable thread whenever the current thread is not running in the threads-kernel.

Multiprocessor: with n processors there will be n running threads. The dispatcher tries to ensure

priority-dispatching-property (MP version) no runnable (but not running) thread has priority higher than the lowest priority running thread.

On a uniprocessor, notice that a high priority thread will exclude all lower priority threads, but the same is not true on a multiprocessor. Also, on a uniprocessor, the priority-dispatching-property always holds (it's an invariant). But on a multi-processor the property is only a desired state: because processors schedule themselves and it takes time for rescheduling to happen, the priority dispatching property is not invariant on multiprocessors.

Example:

```
Running task priorities: 6, 4
The priority 6 running task makes a priority 5 task ready (how?). Then until the other
```

Lesson: *thou shalt not rely on the strict priority property on multiprocessors.*

Priority Inversions

Now consider the interaction of priorities with synchronization. Suppose a low priority thread holds a resource needed by a high priority thread. Clearly, the high priority thread will not be able to proceed until the low priority thread releases the resource. This is called a *priority inversion.* In a system with only two threads this is not a problem. In a system with additional threads at priority above that of the low priority thread it can be a *big* problem. In fact a thread at an intermediate priority level can cause the high priority thread to *never* make any progress. This is known as an *unbounded priority inversion.*

Curing priority inversion problems in strict priority systems: a straightforward solution to priority inversions is to implement *priority inheritance:* when a thread waits on a lock, the priority of the current holder of the lock is temporarily boosted to that of the waiting process.

Consider: how well does priority inheritance work for semaphores used in the passing-the-baton scenario? How well does it work for threads that wait on condition variables?

Example: Mars lander

The Mars pathfinder lander suffered a classic case of unbounded priority inversion soon after its landing. After the mission controllers discovered that the antenna system had much more bandwidth than anticipated they devoted some of the additional bandwidth to sending medium-priority data. This prevented a low-priority thread from releasing a semaphore needed by a high-priority thread in a timely fashion. Another high priority thread noticed that the first had not completed in time and repeatedly forced a system reset. *Handout Wilner's description.*

Interaction of time slicing and Yield with priorities

Last time in class we mentioned difficulties that arise if rogrammers have to include code to voluntarily switch tasks (yield). Time slicing (interrupting a running thread and forcing a switch to another ready thread) is a generally preferable model to the voluntary yield model.

With strict priority scheduling neither time slicing nor voluntary yielding is adequate to ensure progress if priority inversions occur. Also, neither can arrange for a thread to reliably receive a few timeslices now and then.

Ensuring progress in a strict priority scheduler by making it "less strict". DirectedYield

An interesting primitive to include in a threads-kernel is DirectedYield(tid). DirectedYield switches to the named thread regardless of which thread the priority computation would designate as being next. The beneficiary thread runs either to the end of the timeslice or until it blocks. A high priority thread that can inspect the ready queues can occasionally "donate" some cycles to each thread, thus eliminating unbounded priority inversion.

No work has been done on quantifying the length of priority inversions under such a scheme. To me this trick feels too much like recovering from missing notifys by adding timeouts to be very satisfying. Nevertheless, it allowed us to fix a real problem in the GV system when the resources weren't available to fix the application code. We made a simple change to the thread kernel code to solve the problem.

Dynamic Priority Scheduling

Often used in time-sharing systems. Not so common in user-space thread libraries. In this scheme priorities are treated as advice. The system gives more resources to high priority threads than to low priority ones. Threads priorities are changed by the system depending on their resource use. Changes are typically intended to give "good performance" to "interactive threads". I'm not aware of a good characterization of the performance one can expect from these schemes, but they have evolved to the point of being common and useful.

Some systems such as NT and Solaris use a combined dynamic priority and strict priority system, with dynamic priority threads receiving service only after the needs of strict priority threads have been met.

Weighted Scheduling

Weighted scheduling attempts to reflect the importance of different threads by saying what fraction of the processor (or other resource) should be devoted to each thread. In weighted scheduling the goal is to give every thread as much processor time as it needs. This is unlike priority scheduling which equates "higher-priority" with "ability to exclude others".

Example: threads have weights 6, 3, 1: the fraction of the processor received by each thread is $w_i / \sum w_j$. In our example, the first thread receives 60%, the second 30% and the third 10%. Research topic: characterize weighted scheduling when threads are interdependent (interact through synchronization primitives).

Implementations: randomized, weighted perfect shuffle, Bresenham line drawing.

Randomized: pick a random number r in $[0..\sum w_j - 1]$. If $r < w_1$ pick thread 1, if $w_1 \leq r < w_2$ pick thread 2, etc. Advantage: no state is kept about history of scheduling decisions, but choosing random numbers may be expensive. Adapts easily to changing weights or changing ready lists.

Perfect shuffle: for weights summing to 2^n, at each scheduling step increment an n-bit counter. Use its bit-wise reversal in place of r in the above formula. This approach requires minor adjustment if the weights don't sum to a power of two. Requires n bits of state. Reasonableness across weight and ready-list changes must be established.

Line drawing: a fast simple algorithm for drawing optimal approximations to straight lines on pixel-based screens. For a fixed weight set for *N threads* we are trying to draw a pixelated line in N space. Requires state proportional to the number of threads (given the amount of other space that threads require this is probably not unreasonable). Again, adaptation to changing weights and ready lists requires work.

Relationship to WFQ in networking

Hierarchical schedulers

Real-time scheduling

Maybe later (a *very non-realtime* statement!)

Homework

Try to create an unbounded priority inversion in Java. Test your code. What can you conclude about the scheduler in the JVM that you used on the basis of your test?