ECE459: Programming for Performance Winter 2011

Lecture 10 — February 3, 2011

Patrick Lam *version 1*

I talked about using `wait()` and `notify()/notifyAll()` in Java in Lecture 9, off the top of my head. I also said that I'd post a calculation of the maximum number of useful threads. Here it is.

Tip for Implementing Thread Pools: wait/notify

Although the Java `ThreadPoolExecutor` does this all for you (and more efficiently than you'd tend to implement it), we're here to learn about how things work. So I'll explain how thread pool implementations work, at a conceptual level. I'm going to use Java for my examples, which I'm most familiar with, but you can do the same thing in GLib and pthreads[1].

Thread pools typically use a work queue, shared between sources of work (producers) and doers of work (consumers). Idle threads wait for work to appear in the queue.

Worker Thread Implementation

Assume that you have a queue `queue` visible to all. Then, a typical worker thread would look like this:

```
public void run() {
  while (true) { // or other condition
    synchronized (queue) { // grab lock on the queue
      while (queue.isEmpty()) // always while, not if
        queue.wait();
      Work w = queue.removeFirst();
    }
    // do the work on w.
  }
}
```

In words: we have an infinite loop (or a loop which runs until a control task tells the thread to stop). Inside the loop, we grab a lock on the shared work queue, and wait for the queue to become non-empty—note that non-emptiness is the *condition* we're waiting on.

Note that you are always supposed to put the check for the condition inside a `while` loop: 1) the code putting tasks in the queue may use `notifyAll()`, and 1a) it might be notifying about some event that you don't care about; or 1b) another thread might have consumed the work before you got to it; or 2) there might be a spurious wakeup[2].

[1] `http://stackoverflow.com/questions/2085511/wait-and-notify-in-c-c-shared-object`.

[2] Thanks to Scott Gerstmann: `http://download.oracle.com/javase/6/docs/api/java/lang/Object.html#wait()`.

wait(). The `wait()` call temporarily relinquishes the lock while waiting for some other thread to `notify()/notifyAll()` on the lock. After a `notify()`, it waits for its turn to grab the lock, and returns to the caller, again holding the lock.

Finally, the worker code thread calls `removeFirst()` to remove a work item from the queue, releases the lock, and processes the work item.

Adding items to the work queue

Conversely, to add an item to the work queue, you need to grab the lock, add an item, and `notify()` or `notifyAll()` the waiting threads.

```
synchronized (queue) {
  queue.addLast(w);
  queue.notifyAll();
}
```

It's always safe to choose `notifyAll()`, but calling `notify()` is more efficient (fewer wake-ups) if you know that every worker thread can respond to the event. If you have heterogeneous threads waiting for different conditions, use `notifyAll()`.

Control Logic

Java's thread pool implementation contains a minimum and maximum number of threads to keep around. You can change these numbers as appropriate, and the system will spawn or terminate threads to make sure that the right number of threads exist. Any thread pool implementation also would contain similar control logic, either in the main thread or a separate control thread.

Calculating The Maximum Useful Number of Threads

I refused to do this calculation on-the-fly, since I thought I'd mess it up. Following page 97 of Gove, let's say that you have a program with serial time proportion T_s and parallel time proportion $T_p = 1 - T_s$. The total runtime T with N threads would be:

$$T = T_s + \frac{T_p}{N}.$$

Let's say that you have a tolerance t. Then an acceptable runtime T' would be:

$$T' = T_s(1 + t),$$

assuming that you don't speed up the T_s part at all and speed up the T_p part just up to the tolerance. Equating T and T', and solving for N:

$$\begin{aligned} T_s(1+t) &= \left(T_s + \frac{T_p}{N}\right); \text{ and} \\ N &= \frac{T_p}{T_s t} = \frac{T_p}{(1-T_p)t}. \end{aligned}$$