Lecture 9: Concurrency Conclusion

9.0 Main Points:

Summarize concurrency section

Illustrate drawbacks to threads

9.1 Concurrency Summary

Physical Hardware	Programming Abstraction
single CPU, interrupts, test&set	sequential execution, infinite # of CPUs, semaphores and monitors

Every major operating system built since 1985 has provided threads -- Mach, OS/2, NT (Microsoft), Solaris (new OS from SUN), OSF (DEC Alphas). Why? Because makes it a lot easier to write concurrent programs, from Web servers, to databases, to embedded systems.

So does this mean you should all go out and use threads?

9.2 Cautionary Tales

Illustrate why an abstraction doesn't always work the way you want it to.

9.2.1 Therac-25

Concurrency bug in radiation theraphy machine, documented case where software bug killed several people. Caused by race condition between user entering command and hardware reading the entered value.

9.2.2 Eraser

Eraser is an automatic tool to detect race conditions in concurrent programs. Simple check: are variables protected by the same lock everywhere in the program? Clearly, won't catch all bugs!

Used on two sets of programs:

Assignments in CSE 451 (previous project) -- 90% of all students had race condition in at least one submitted assignment.

Production software written by PhD operating systems researchers (search engine, file server, source code management system). Each had at least one race condition, some multiple.

Background: uniprocessors provide sequential ordering -- all memory operations appear to the processor to be done in order. But most multiprocessors implement weak ordering:

a) all synchronization operations are done in the same order at every CPU
b) all reads and writes are done everywhere before next synch operation
c) reads and writes between synch instructions can be done in any order

Is this policy OK for critical sections?

Thread A	Thread B
`Lock`	`Lock`
`Write x = 1;`	`If (x == 1) read y`
`Write y;`	`unlock`
`unlock`	

Is it ok for the code below?

```
if (p->val == 0) {  // check if uninitialized
     lock
     if (p->val == 0) {
          initialize p->ptr
          p->val = 1;
     }
     unlock;
}
use p->ptr
```

9.2.3 OS/2

Microsoft OS/2 (around 1988): initially, a spectacular failure. Since then, IBM has completely re-written it from scratch. (Another spectacular failure, but for a different reason.) Used threads for everything -- window systems, communication between programs, etc. Threads are a good idea, right?

Thus, system created lots of threads, but few actually running at any one time -- most waiting around for user to type in a window, or for a network packet to arrive, etc.

Scenario: Might have 100 threads, but just a few at any one time on the ready queue. And each thread needs its own execution stack, say, 10 KB, whether it is runnable or waiting.

Result: system needs an extra**1 MB** of memory, mostly consumed by waiting threads. 1 MB of memory cost $200 in 1988 (albeit $2 today).

Moral: threads are cheap, but they're not free.

9.2.4 Event Loops

If threads are dangerous and slow, why use them? Are there any alternatives?

Event loops: suppose you structured your program as a giant event loop. When a event arrives (e.g., disk interrupt, network packet, keystroke, etc.), code processes event and decides what to do next. It runs the operation to completion, then waits for next event.

Analogy between iteration and recursion. You can manage the stack yourself, and recode any recursive program into an iterative one.

Advantages: completely deterministic! If events arrive in the same order, behavior of the program will be exactly the same.

Disadvantages: have to completely restructure your code around the event loop, and even so, won't work on a multiprocessor.

9.2.5 Threads and Multiprocessors

How do you write a parallel program?

One idea: Just split program up into threads, so that things can run in parallel.

Example: Matrix multiply

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
     for (k = 0; k < N; k++)
        C[i][j] += A[i][k] * B[j][k];
```

Create a thread for every iteration of inner loop (each i, j, k)? Each one can run in parallel, using a lock to protect access to each element in C[i][j].

Would work, but wouldn't be efficient.

Instead: group iterations (e.g., thread for each i, j) so that each thread does a fair amount of work. One benefit, no lock on C[i][j]