CSE 303: Concepts and Tools for Software Development

Hal Perkins
Spring 2008
Lecture 22—Shared-Memory Concurrency
Concurrent Computation

Computation where “multiple things happen at the same time” is inherently more complicated than sequential computation.

- Entirely new kinds of bugs and obligations

Two forms of concurrency:

- time-slicing: only one computation at a time but pre-empt to provide responsiveness or mask I/O latency.

- true parallelism: more than one CPU (e.g., the lab machines have two, the attu machines have 4, ...)

No problem unless the different computations need to communicate or use the same resources.
Example: Processes

The O/S runs multiple processes “at once”.

Why? (Convenience, efficient use of resources, performance)

No problem: keep their address-spaces separate.

But they do communicate/share via files (and pipes).

Things can go wrong, e.g., a race condition:

```
echo "hi" > someFile
foo=`cat someFile`
# assume foo holds the string hi??
```

The O/S provides synchronization mechanisms to avoid this

- See CSE451; we will focus on intraprocess concurrency.
The Old Story

We said a running Java or C program had code, a heap, global variables, a stack, and “what is executing right now” (in assembly, a program counter).

C, Java support parallelism similarly (other languages can be different):

- One pile of code, global variables, and heap.
- Multiple “stack + program counter”s — called threads
- Threads can be pre-empted whenever by a scheduler
- Threads can communicate (or mess each other up) via shared memory.
- Various synchronization mechanisms control what thread interleavings are possible.
  - “Do not do your thing until I am done with my thing”
Basics

C: The POSIX Threads (pthreads) library

• `#include <pthread.h>`

• Link with `-lpthread`

• `pthread_create` takes a function pointer and an argument for it; runs it as a separate thread.

• Many types, functions, and macros for threads, locks, etc.

Java: Built into the language

• Subclass `java.lang.Thread` overriding `run`

• Create a `Thread` object and call its `start` method

• Any object can “be synchronized on” (later)
Why do this?

- Convenient structure of code
  - Example: 2 threads using information computed by the other
  - Example: Failure-isolation – each “file request” in its own thread so if a problem just “kill that request”.
  - Example: Fairness – one slow computation only takes some of the CPU time without your own complicated timer code. Avoids starvation.

- Performance
  - Run other threads while one is reading/writing to disk (or other slow thing that can happen in parallel)
  - Use more than one CPU at the same time
    * The way computers will get faster over the next 10 years
    * So no parallelism means no faster.
Simple synchronization

If one thread did nothing of interest to any other thread, why is it running?

So threads have to communicate and coordinate.

- Use each others' results; avoid messing up each other's computation.

Simplest two ways not to mess each other up (don’t underestimate!):

1. Do not access the same memory.
2. Do not mutate shared memory.

Next simplest: One thread does not run until/unless another thread is done

- Called a join
Using Parallel Threads

- A common pattern for expensive computations:
  - Split the work
  - Join on all the helper threads
  - Called fork-join parallelism

- To avoid bottlenecks, each thread should have about the same amount of work (load-balancing)
  - Performance depends on number of CPUs available and will typically be less than “perfect speedup”

- C vs. Java (specific to threads)
  - Java takes an OO approach (shared data via fields of Thread)
  - Java separates creating the Thread-object and creating the running-thread
Less structure

Often you have a bunch of threads running at once and they *might* need the same mutable memory at the same time but *probably not*.

Want to be *correct* without sacrificing parallelism.

Example: A bunch of threads processing bank transactions:

- withdraw, deposit, transfer, currentBalance, ...
- chance of two threads accessing the same account at the same time very low, but not zero.
- want *mutual exclusion* (a way to keep each other out of the way when there is *contention*)

Another example: Parallel search through an arbitrary graph
The Issue

```c
struct Acct { int balance; /* ... other fields ... */ };

int withdraw(struct Acct * a, int amt) {
    if(a->balance < amt) return 1; // 1==failure
    a->balance -= amt;
    return 0; // 0==success
}
```

This code is correct in a sequential program.

It may have a race condition in a concurrent program, allowing a negative balance.

Discovering this bug is very hard with testing since the interleaving has to be “just wrong”.
Programmers must indicate what must *appear to happen all-at-once*.

```c
int withdraw(struct Acct * a, int amt) {
    atomic {
        if(a->balance < amt) return 1; // 1==failure
        a->balance -= amt;
    }
    return 0; // 0==success
}
```

Reasons not to do “too much” in an atomic:

- **Correctness**: If another threads needs an intermediate result to compute something you need, must “expose” it.
- **Performance**: Parallel threads must access disjoint memory
  - Actually read/read conflicts can happen in parallel
Getting it “just right”

This code is probably wrong because critical sections too small:

```c
atomic { if(a->balance < amt) return 1; }
atomic { a->balance -= amt; }
```

This code (skeleton) is probably wrong because critical section too big:

- Assume other guy does not compute until the data is set.

```c
atomic {
    data_for_other_guy = 42; // set some global
    ans = wait_for_other_guy_to_compute();
    return ans;
}
```
So far

Shared-memory concurrency where multiple threads might access the same mutable data at the same time is tricky

- Must get size of critical sections just right

It’s worse because

- atomic does not yet exist in languages like C and Java
- (Major thread of programming language research at UW.)

Instead programmers must use locks (a.k.a. mutexes) or other mechanisms, usually to get the behavior of critical sections

- But misuse of locks will violate the “all-at-once” property
- Or lead to other bugs we haven’t seen yet