Administrivia

• HW7 due tomorrow night, 11 pm
  – (+ late days if you still have any and want to use them)

• Friday office hours moved to 12:30-1:20 (right after class) in CSE 021 because of scheduling conflicts

• Course evals – reissued Monday to get instructor name (i.e., who to blame) right. Please fill out this week.
Concurrency

• 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 preempt to provide responsiveness or mask I/O latency
  – true parallelism: more than one CPU (e.g., most consumer machines have 2-4, your laptop has ?, …)
• 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
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”
Threads in C and Java

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, 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: two threads using information computed by the other
• Example: failure-isolation – each “transaction” in its own thread so if a problem just “kill that transaction”
• 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 network (or other slow thing that can happen in parallel)
• Use more than one CPU at the same time
  – The way computers get faster these days
  – 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 (i.e., wait until all done)
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”
Less structure

- Often you have a bunch of threads running at once and they might need the same mutable (writable) 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)
The issue

```
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”
Program 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 thread 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:
  
  atomic { if(a->balance < amt) return 1; }
  atomic { a->balance -= amt; }

• This code (skeleton) is probably wrong because the critical section is too big:
  – Assume other guy does not compute until data is set
    
    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
• 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
Lock basics

A lock is \textit{acquired} and \textit{released} by a thread

- At most one thread “holds it” at any moment
- Acquiring it “blocks” until the holder releases it and the blocked thread acquires it
  - Many threads might be waiting; one will “win”
  - The lock-implementer avoids race conditions on the lock-acquire
- So to keep two things from happening at the same time, surround them with the same lock-acquire/lock-release
Locks in C/Java

• C: Need to initialize and destroy mutexes (a synonym for locks)
  – The joys of C
• An initialized (pointer to a) mutex can be locked or unlocked via library function calls
• Java: A synchronized statement is an acquire/release
  – Any object can serve as a lock
  – Lock is released on any control-transfer out of the block (return, break, exception, ...)
  – “Synchronized methods” just save keystrokes
Choosing how to lock

- Now we know what locks are (how to make them, what acquiring/releasing means), but programming with them correctly and efficiently is difficult...
  - As before, if critical sections are too small we have races; if too big we may not communicate enough to get our work done efficiently
  - But now, if two “synchronized blocks” grab different locks, they can be interleaved even if they access the same memory
    - A “data race”
  - Also, a lock-acquire blocks until a lock is available and only the current-holder can release it
    - Can have “deadlock” ...

Deadlock

Object a;
Object b;

void m1() {
    synchronized a {
        synchronized b {
            synchronized b {
                synchronized a {
                    ...
                }
            }
        }
    }
}

void m2() {
    synchronized b {
        synchronized a {
            ...
        }
    }
}

• A cycle of threads waiting on locks means none will ever run again!
• Avoidance: All code acquires locks in the same order (very hard to do). Ad hoc: Don’t hold onto locks too long or while calling into unknown code
• Recovery: detect deadlocks, kill off and rerun one of the processes (databases)
Rules of thumb

• Any one of the following are sufficient for avoiding races:
  – Keep data thread-local (an object is reachable, or at least only accessed by, one thread)
  – Keep data read-only (do not assign to object fields after an object’s constructor)
  – Use locks consistently (all accesses to an object are made while holding a particular lock)
  – Use a partial-order to avoid deadlock (over-simple example: do not hold multiple locks at once?)
• These are tough invariants to get right, but that’s the price of multithreaded programming today
• But... one way to do all the above is to have “one lock for all shared data” and that is inefficient...
False sharing

- “False sharing” refers to not allowing separate things to happen in parallel. Example:

  ```java
  synchronized x {
    synchronized x {
      ++y;
      ++z;
    }
  }
  ```

- More realistic example: one lock for all bank accounts rather than one for each account

- On the other hand, acquiring/releasing locks is not so cheap, so “locking more with the same lock” can improve performance

- This is the “locking granularity” question
  - Coarser vs. finer granularity
What about this?

- If each bank account has its own lock, how do you write a “transfer” method such that no other thread can see the “wrong total balance”?

```java
// race (not data race)  // potential deadlock
void xfer(int a, Acct other) {
    synchronized(this) {
        balance += a;
        other.balance -= a;
    }
}
```

- The problem is there is no relative order among accounts, so “inverse transfers” could deadlock
A final gotcha

• You would naturally assume that all memory accesses happen in “some consistent order” that is “determined by the code”
• Unfortunately, compilers and chips are often allowed to cheat (reorder)! The assertion in the right thread may fail!
  initially flag==false
  data = 42;   while(!flag) { } 
  flag = true;   assert(data==42);
• To disallow reordering the programmer must:
  – Use lock acquires (no reordering across them), or
  – Declare flag to be volatile (for experts, not us)
Conclusion

• Threads make a lot of otherwise-correct approaches incorrect
  – Writing “thread-safe” libraries can be excruciating
  – Use an expert implementation if you can, e.g., Java’s ConcurrentHashMap & others
• But they are increasingly important for efficient use of computing resources
• Locks and shared-memory are (just) one common approach