CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Spring 2007

Lecture 23— Concurrency Wrap-up; Introduction to C++

Where are we

Done:

- Basics of shared-memory multithreading
- Fork-join parallelism
- Critical sections via `atomic`
- Critical sections via careful use of locks (a.k.a. mutexes)

Doing:

- Pitfalls of using locks
- Other concurrency gotchas

Then:

- Some basics of C++ (2 lectures, need 20 for the whole language)

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

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:

```
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

Very challenging situation

My favorite example for ridiculing locks:

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”?

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

```
// potential deadlock
void xfer(int a,Acct other){
 synchronized(this) {
 synchronized(other) {
   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, e.g., Java's ConcurrentHashMap?

But they are increasingly important for efficient use of computing resources ("the multicore revolution").

Locks and shared-memory are (just) one common approach.

Learn about other useful synchronization mechanisms (e.g., condition variables) in CSE451.

C++

C++ is an enormous language:

- All of C
- Classes and objects (kind of like Java, some crucial differences)
- Many more little conveniences (I/O, new/delete, function overloading, pass-by-reference, bigger standard library)
- Namespaces (kind of like Java packages)
- Stuff we won't do: const, different kinds of casts, exceptions, templates, multiple inheritance, ...

We will focus on a couple themes rather than just a "big bag of new features to memorize"...

Our focus

OOP in a C-like language may help you understand C and Java better?

- We can put objects on the stack or the heap; an object is not a pointer to an object
- Still have to manage memory manually
- Still lots of ways to HCBWKMSCOD (hopefully crash, but who knows – might silently corrupt other data)
- Still distinguish header files from implementation files
- Allocation and initialization still separate concepts, but easier to "construct" and "destruct"
- Programmer has more control on how method-calls work (different defaults from Java)

Hello World

```
#include <iostream>
int main() {
// Use standard output stream cout
// and operator << to send "Hello World"
// and an end line to stdout
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

Differences from C: “new-style” headers (no .h), namespace access (::), I/O via stream operators, ...

Differences from Java: not everything is in a class, any code can go in any file, ...

Compiling

Need a different compiler than for C; use g++ on attu. Example:

```
g++ -Wall -o hello hello.cc
```

The `.cc` extension is a convention (just like `.c` for C), but less universal (also see `.cpp`, `.cxx`, `.C`).

Uses the C preprocessor (no change there).

Now: A few "niceties" before our real focus (classes and objects).

I/O

Operator << takes a "ostream" and (various things) and outputs it; returns the stream, which is why

`std::cout << 3 << "hi" << f(x) << '\n';` works

- Easier and safer than printf

Operator >> takes "istream" and (various things) and inputs into it.

- Easier and safer than scanf. Do *not* use pointers; e.g.,
  `int x; cin >> x;`

Can "think of" >> and << as keywords, but they are not:

- *Operator overloading* redefines them for different pairs of types.
  - In C they mean "left-shift" and "right-shift" (of bits); undefined for non-numeric types.
- Lack of address-of for input done with *call-by-reference* (later).

Namespaces

In C, all non-static functions in the program need different names

- Even operating systems with tens of millions of lines.

Namespaces (cf. Java packages) let you group top-level names:

- `namespace myspace { ... definitions ... }`
- Of course, then different namespaces can have the same function names and they are totally different functions.
- Can nest them
- Can reuse the same namespace in multiple places
  - Pariticularly common: in the `.h` and the `.cc`

For example, the whole C++ standard library is in namespace `std`.

To use a function/variable/etc. in another namespace, do `thespace::someFun()` (not . like in Java)

Using

To avoid having to write namespaces and `::` constantly, use a *using declaration*

Example:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello World" << endl;
  return 0;
}
```

Onto OOP

Consider the simple class defined in `Property.h`/`Property.cc`.

Like Java:

- Fields vs. methods, static vs. instance, constructors
- Method overloading (functions, operators, and constructors too)

Not quite like Java:

- access-modifiers (e.g., `private`) syntax and default
- declaration separate from implementation (like C)
- funny constructor syntax, default parameters (e.g., `... = 0`)

Nothing like Java:

- Objects vs. pointers to objects
- Destructors and copy-constructors
- virtual vs. non-virtual (to be discussed)