CSE332: Data Abstractions

Lecture 23: Data Races and Memory Reordering Deadlock Readers/Writer Locks Condition Variables

Dan Grossman

Spring 2012

Outline

Done:

- Programming with locks and critical sections
- Key guidelines and trade-offs

Now: The other basics an informed programmer needs to know

- Why you must avoid data races (memory reorderings)
- Another common error: Deadlock
- Other common facilities useful for shared-memory concurrency
  - Readers/writer locks
  - Condition variables, or, more generally, passive waiting

Motivating memory-model issues

Tricky and *surprisingly wrong* unsynchronized concurrent code

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

First understand why it looks like the assertion cannot fail:

- Easy case: call to **g** ends before any call to **f** starts
- Easy case: at least one call to **f** completes before call to **g** starts
- If calls to **f** and **g** *interleave*…

Interleavings

There is no interleaving of **f** and **g** where the assertion fails

- Proof #1: Exhaustively consider all possible orderings of access to shared memory (there are 6)
- Proof #2: If **!(b>=a)**, then **a==1** and **b==0**. But if **a==1**, then **y=1** happened before **a=y**. Because programs execute in order: **a=y** happened before **b=x** and **x=1** happened before **y=1**. So by transitivity, **b==1**. Contradiction.

Thread 1: **f**

```
x = 1;
y = 1;
```

Thread 2: **g**

```
int a = y;
int b = x;
assert(b >= a);
```


Wrong

However, the code has a *data race*

- Two actually
- Recall: data race: unsynchronized read/write or write/write of same location

If code has data races, you cannot reason about it with interleavings!

- That is simply the rules of Java (and C, C++, C#, …)
- (Else would slow down all programs just to “help” programs with data races, and that was deemed a bad engineering trade-off when designing the languages/compilers/hardware)
- So the assertion can fail

Recall Guideline #0: No data races

Why

For performance reasons, the compiler and the hardware often reorder memory operations

- Take a compiler or computer architecture course to learn why

Thread 1: **f**

```
x = 1;
y = 1;
```

Thread 2: **g**

```
int a = y;
int b = x;
assert(b >= a);
```

Of course, you cannot just let them reorder anything they want

- Each thread executes in order after all!
- Consider: **x=17; y=x;**

The grand compromise

The compiler/hardware will never perform a memory reordering that affects the result of a single-threaded program

The compiler/hardware will never perform a memory reordering that affects the result of a data-race-free multi-threaded program

So: If no interleaving of your program has a data race, then you can *forget about all this reordering nonsense:* the result will be equivalent to some interleaving

Your job: Avoid data races

Compiler/hardware job: Give illusion of interleaving *if you do your job*

Fixing our example

- Naturally, we can use synchronization to avoid data races
  - Then, indeed, the assertion cannot fail

```
class C {
  private int x = 0;
  private int y = 0;
  void f() {
    synchronized(this) { x = 1; }
    synchronized(this) { y = 1; }
  }
  void g() {
    int a, b;
    synchronized(this) { a = y; }
    synchronized(this) { b = x; }
    assert(b >= a);
  }
}
```


A second fix

- Java has `volatile` fields: accesses do not count as data races
- Implementation: slower than regular fields, faster than locks
- Really for experts: avoid them; use standard libraries instead
- And why do you need code like this anyway?

```
class C {
  private volatile int x = 0;
  private volatile int y = 0;
  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```


Code that is wrong

- Here is a more realistic example of code that is wrong
  - No *guarantee* Thread 2 will *ever* stop
  - But honestly it will "likely work in practice"

```
class C {
  boolean stop = false;
  void f() {
    while(!stop) {
      // draw a monster
    }
  }
  void g() {
    stop = didUserQuit();
  }
}
```

Thread 1: `f()`

Thread 2: `g()`

Outline

Done:

- Programming with locks and critical sections
- Key guidelines and trade-offs

Now: The other basics an informed programmer needs to know

- Why you must avoid data races (memory reorderings)
- Another common error: Deadlock
- Other common facilities useful for shared-memory concurrency
  - Readers/writer locks
  - Condition variables

Motivating Deadlock Issues

Consider a method to transfer money between bank accounts

```
class BankAccount {
  …
  synchronized void withdraw(int amt) {…}
  synchronized void deposit(int amt) {…}
  synchronized void transferTo(int amt,
                               BankAccount a) {
    this.withdraw(amt);
    a.deposit(amt);
  }
}
```

Notice during call to `a.deposit`, thread holds *two* locks

- Need to investigate when this may be a problem

The Deadlock

Suppose **x** and **y** are fields holding accounts

Thread 1: `x.transferTo(1,y)`	Thread 2: `y.transferTo(1,x)`
acquire lock for x	
do withdraw from x	
	acquire lock for y
	do withdraw from y
	block on lock for x
block on lock for y	

Time

Deadlock, in general

A deadlock occurs when there are threads **T1**, …, **Tn** such that:

- For **i**=1,..,n-1, **Ti** is waiting for a resource held by **T(i+1)**
- **Tn** is waiting for a resource held by **T1**

In other words, there is a cycle of waiting

- Can formalize as a graph of dependencies with cycles bad

Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise

Back to our example

Options for deadlock-proof transfer:

1. Make a smaller critical section: `transferTo` not synchronized
   - Exposes intermediate state after `withdraw` before `deposit`
   - May be okay, but exposes wrong total amount in bank
2. Coarsen lock granularity: one lock for all accounts allowing transfers between them
   - Works, but sacrifices concurrent deposits/withdrawals
3. Give every bank-account a unique number and always acquire locks in the same order
   - *Entire program* should obey this order to avoid cycles
   - Code acquiring only one lock can ignore the order

Ordering locks

```
class BankAccount {
  …
  private int acctNumber; // must be unique
  void transferTo(int amt, BankAccount a) {
    if(this.acctNumber < a.acctNumber)
       synchronized(this) {
       synchronized(a) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
    else
       synchronized(a) {
       synchronized(this) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
  }
}
```


Another example

From the Java standard library

```
class StringBuffer {
  private int count;
  private char[] value;
  …
  synchronized append(StringBuffer sb) {
    int len = sb.length();
    if(this.count + len > this.value.length)
      this.expand(…);
    sb.getChars(0,len,this.value,this.count);
  }
  synchronized getChars(int x, int, y,
                        char[] a, int z) {
    "copy this.value[x..y] into a starting at z"
  }
}
```


Two problems

Problem #1: Lock for `sb` is not held between calls to `sb.length` and `sb.getChars`

- So `sb` could get longer
- Would cause `append` to throw an `ArrayBoundsException`

Problem #2: Deadlock potential if two threads try to `append` in opposite directions, just like in the bank-account first example

Not easy to fix both problems without extra copying:

- Do not want unique ids on every `StringBuffer`
- Do not want one lock for all `StringBuffer` objects

Actual Java library: fixed neither (left code as is; changed javadoc)

- Up to clients to avoid such situations with own protocols

Perspective

- Code like account-transfer and string-buffer append are difficult to deal with for deadlock
- Easier case: different types of objects
  - Can document a fixed order among types
  - Example: "When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock"
- Easier case: objects are in an acyclic structure
  - Can use the data structure to determine a fixed order
  - Example: "If holding a tree node's lock, do not acquire other tree nodes' locks unless they are children in the tree"

Outline

Done:

- Programming with locks and critical sections
- Key guidelines and trade-offs

Now: The other basics an informed programmer needs to know

- Why you must avoid data races (memory reorderings)
- Another common error: Deadlock
- Other common facilities useful for shared-memory concurrency
  - Readers/writer locks
  - Condition variables

Reading vs. writing

Recall:
- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent read & write of same memory: Problem

So far:
- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:
- Could still allow multiple simultaneous readers!

Example

Consider a hashtable with one coarse-grained lock
- So only one thread can perform operations at a time

But suppose:
- There are many simultaneous `lookup` operations
- `insert` operations are very rare

Note: Important that `lookup` does not actually mutate shared memory, like a move-to-front list operation would

Readers/writer locks

A new synchronization ADT: The readers/writer lock

- A lock's states fall into three categories:
  - "not held"
  - "held for writing" by one thread
  - "held for reading" by *one or more* threads

$0 \le$ writers ≤ 1
$0 \le$ readers
writers*readers==0

- `new`: make a new lock, initially "not held"
- `acquire_write`: block if currently "held for reading" or "held for writing", else make "held for writing"
- `release_write`: make "not held"
- `acquire_read`: block if currently "held for writing", else make/keep "held for reading" and increment *readers count*
- `release_read`: decrement readers count, if 0, make "not held"

Pseudocode example (not Java)

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock for table
  RWLock lk = new RWLock();
  V lookup(K key) {
    int bucket = hasher(key);
    lk.acquire_read();
    … read array[bucket] …
    lk.release_read();
  }
  void insert(K key, V val) {
    int bucket = hasher(key);
    lk.acquire_write();
    … write array[bucket] …
    lk.release_write();
  }
}
```

Readers/writer lock details

- A readers/writer lock implementation ("not our problem") usually gives *priority* to writers:
  - Once a writer blocks, no readers *arriving later* will get the lock before the writer
  - Otherwise an **`insert`** could *starve*
- Re-entrant?
  - Mostly an orthogonal issue
  - But some libraries support *upgrading* from reader to writer
- Why not use readers/writer locks with more fine-grained locking, like on each bucket?
  - Not wrong, but likely not worth it due to low contention

In Java

Java's **`synchronized`** statement does not support readers/writer

Instead, library
`java.util.concurrent.locks.ReentrantReadWriteLock`

- Different interface: methods **`readLock`** and **`writeLock`** return objects that themselves have **`lock`** and **`unlock`** methods
- Does *not* have writer priority or reader-to-writer upgrading
  - Always read the documentation

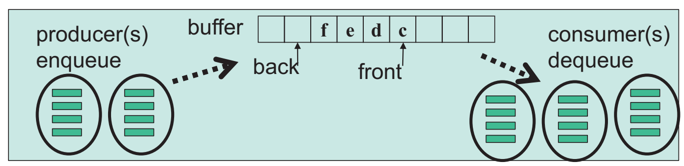
Outline

Done:
- Programming with locks and critical sections
- Key guidelines and trade-offs

Now: The other basics an informed programmer needs to know

- Why you must avoid data races (memory reorderings)
- Another common error: Deadlock
- Other common facilities useful for shared-memory concurrency
  - Readers/writer locks
  - Condition variables

Motivating Condition Variables



To motivate condition variables, consider the canonical example of a bounded buffer for sharing work among threads

Bounded buffer: A queue with a fixed size
- (Unbounded still needs a condition variable, but 1 instead of 2)

For sharing work – think an assembly line:
- Producer thread(s) do some work and enqueue result objects
- Consumer thread(s) dequeue objects and do next stage
- Must synchronize access to the queue

Code, attempt 1

```
class Buffer<E> {
  E[] array = (E[])new Object[SIZE];
  … // front, back fields, isEmpty, isFull methods
  synchronized void enqueue(E elt) {
    if(isFull())
      ???
    else
      … add to array and adjust back …
  }
  synchronized E dequeue()
    if(isEmpty())
      ???
    else
      … take from array and adjust front …
  }
}
```


Waiting

- **`enqueue`** to a full buffer should *not* raise an exception
  - Wait until there is room
- **`dequeue`** from an empty buffer should *not* raise an exception
  - Wait until there is data

Bad approach is to *spin* (wasted work and keep grabbing lock)

```
void enqueue(E elt) {
  while(true) {
    synchronized(this) {
      if(isFull()) continue;
      … add to array and adjust back …
      return;
}}}
// dequeue similar
```

What we want

- Better would be for a thread to *wait* until it can proceed
  - Be *notified* when it should try again
  - In the meantime, let other threads run
- Like locks, not something you can implement on your own
  - Language or library gives it to you, typically implemented with operating-system support
- An ADT that supports this: condition variable
  - Informs waiter(s) when the *condition* that causes it/them to wait has *varied*
- Terminology not completely standard; will mostly stick with Java

*Java approach: **not** quite right*

```
class Buffer<E> {
  …
  synchronized void enqueue(E elt) {
    if(isFull())
      this.wait(); // releases lock and waits
    add to array and adjust back
    if(buffer was empty)
      this.notify(); // wake somebody up
  }
  synchronized E dequeue() {
    if(isEmpty())
      this.wait(); // releases lock and waits
    take from array and adjust front
    if(buffer was full)
      this.notify(); // wake somebody up
  }
}
```


Key ideas

- Java weirdness: every object "is" a condition variable (and a lock)
  - other languages/libraries often make them separate
- `wait`:
  - "register" running thread as interested in being woken up
  - then atomically: release the lock and block
  - when execution resumes, *thread again holds the lock*
- `notify`:
  - pick one waiting thread and wake it up
  - no guarantee woken up thread runs next, just that it is no longer blocked on the *condition* – now waiting for the *lock*
  - if no thread is waiting, then do nothing

Bug #1

```
synchronized void enqueue(E elt){
  if(isFull())
    this.wait();
  add to array and adjust back
  …
}
```

Between the time a thread is notified and it re-acquires the lock, the condition can become false again!

Time ↓

Thread 1 (enqueue)	Thread 2 (dequeue)	Thread 3 (enqueue)
`if(isFull())` `this.wait();`		
	`take from array` `if(was full)` `this.notify();`	
		`make full again`
`add to array`		

Bug fix #1

```
synchronized void enqueue(E elt) {
  while(isFull())
    this.wait();
  …
}
synchronized E dequeue() {
  while(isEmpty())
    this.wait();
  …
}
```

Guideline: *Always* re-check the condition after re-gaining the lock
- In fact, for obscure reasons, Java is technically allowed to notify a thread *spuriously* (i.e., for no reason)

Bug #2

- If multiple threads are waiting, we wake up only one
  - Sure only one can do work *now*, but can't forget the others!

Time ↓

Thread 1 (enqueue)	Thread 2 (enqueue)	Thread 3 (dequeues)
`while(isFull())` `this.wait();`	`while(isFull())` `this.wait();`	
		`// dequeue #1` `if(buffer was full)` `this.notify();`
…	…	`// dequeue #2` `if(buffer was full)` `this.notify();`

Bug fix #2

```
synchronized void enqueue(E elt) {
  …
  if(buffer was empty)
    this.notifyAll(); // wake everybody up
}
synchronized E dequeue() {
  …
  if(buffer was full)
    this.notifyAll(); // wake everybody up
}
```

`notifyAll` wakes up all current waiters on the condition variable

Guideline: If in any doubt, use **`notifyAll`**

- Wasteful waking is better than never waking up

- So why does **`notify`** exist?
  - Well, it is faster when correct…

Alternate approach

- An alternative is to call **`notify`** (not **`notifyAll`**) on every **`enqueue`** / **`dequeue`**, not just when the buffer was empty / full
  - Easy: just remove the **`if`** statement
- Alas, makes our code subtly wrong since it is technically possible that an **`enqueue`** and a **`dequeue`** are both waiting
  - See notes for the step-by-step details of how this can happen
- Works fine if buffer is unbounded since then only dequeuers wait

Alternate approach fixed

- The alternate approach works if the enqueuers and dequeuers wait on *different* condition variables
  - But for mutual exclusion both condition variables must be associated with the same lock
- Java's "everything is a lock / condition variable" does not support this: each condition variable is associated with itself
- Instead, Java has classes in **`java.util.concurrent.locks`** for when you want multiple conditions with one lock
  - **`class ReentrantLock`** has a method **`newCondition`** that returns a new **`Condition`** object associate with the lock
  - See the documentation if curious

Last condition-variable comments

- **`notify/notifyAll`** often called **`signal/broadcast`**, also called **`pulse/pulseAll`**
- Condition variables are subtle and harder to use than locks
- But when you need them, you need them
  - Spinning and other work-arounds do not work well
- Fortunately, like most things in a data-structures course, the common use-cases are provided in libraries written by experts
  - Example: **`java.util.concurrent.ArrayBlockingQueue<E>`**
  - All uses of condition variables hidden in the library; client just calls **`put`** and **`take`**

Concurrency summary

- Access to shared resources introduces new kinds of bugs
  - Data races
  - Critical sections too small
  - Critical sections use wrong locks
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion (common, various flavors)
  - Condition variables for signaling others (less common)
- Guidelines for correct use help avoid common pitfalls
- Not clear shared-memory is worth the pain
  - But other models (e.g., message passing) not a panacea