CSE332: Data Abstractions

Lecture 25: Deadlocks and Additional Concurrency Issues

Tyler Robison

Summer 2010

Where we are

- We’ve covered basic concurrency, then some odds and ends:
  - Readers/writer locks
  - Condition variables
- There are a couple more common issues we need to hit:
  - Deadlocks: Very common and very bad
  - Additional problems that pop up due to concurrency

A New Concurrency Issue: Deadlocks

- So far our bank account operations have been limited to one account
- Now consider a *transfer* method between accounts
- As always, we'd like to synchronize access (one lock per account for a fine-grained locking scheme)

```
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 ***2*** locks; first it's own then the destination account's (both due to `synchronized`)

The Deadlock

For simplicity, suppose **`x`** and **`y`** are static fields holding accounts

What happens if symmetric transfers occur simultaneously between accounts x & y?

Thread 1: **`x.transferTo(1,y)`**

Thread 2: **`y.transferTo(1,x)`**

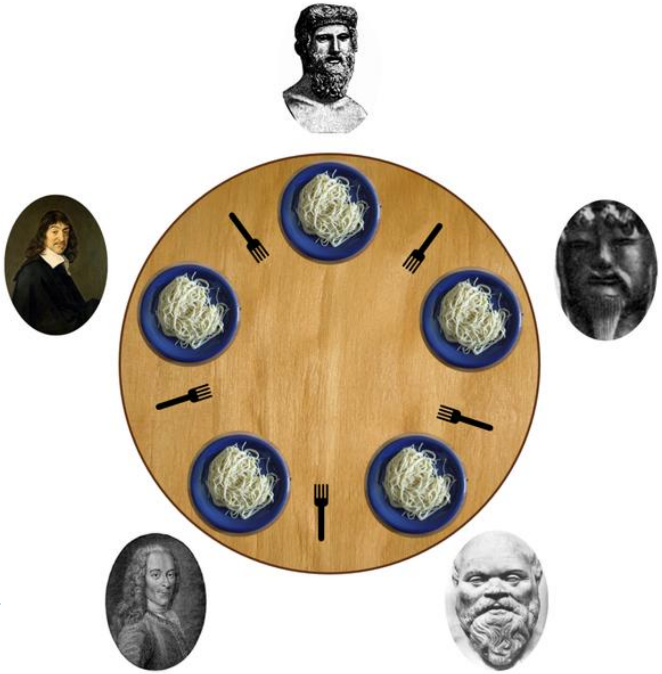
Time

Thread 1	Thread 2
acquire lock for x	
do withdraw from y	
	acquire lock for y
	do withdraw from x
	block on lock for x
block on lock for y	

Deadlock: Each thread is waiting for the other's lock

Ex: The Dining Philosophers

- 5 philosophers go out to dinner together at an Italian restaurant
- Sit at a round table; one fork per setting
- When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork, then eats
- ‘Locking’ for each fork results in a ***deadlock***



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

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 work out okay here, but would break other functionality
     - If we were to get the total $ in all accounts at this point, it would be wrong

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 is fine though

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

Problem #2: The 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`**

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

Perspective

- Code like account-transfer and string-buffer append are difficult to deal with for reasons of 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”

Motivating memory-model issues

Tricky and *surprisingly wrong* unsynchronized concurrent code; the assert below *should* never be capable of failing

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

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

It *seems* like it could never fail, despite how it interleaves:

- x and y are initialized to 0 when the object is constructed; no concurrent on the object possible there
- x and y can only change when f() is called; first x changes, then y changes
- g() get's y's value, then x's
- For the assert to fail, yy's value needs to be greater than xx's

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

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

```
int yy = y;
int xx = x;
assert(xx >= yy);
```

- Proof #2: Assume `!(xx>=yy)`; then `yy==1` and `xx==0`
  - But if `yy==1`, then `yy=y` happened after `y=1`
  - Since programs execute in order, `xx=x` happened after `yy=y` and `x=1` happened before `y=1`
  - So by transitivity, `xx==1`. Contradiction.

For yy=1, the yy assignment must happen *after* the y assignment

Thread 1: **`f`**

```
x = 1;

y = 1;
```

Thread 2: **`g`**

```
int yy = y;

int xx = x;

assert(xx >= yy);
```

Data Race => Wrong

However, the code has a *data race*

- Two actually; potentially simultaneous access to x & y
- Recall: data race = unsynchronized read/write or write/write of same location = bad

If your code has data races, you can't reason about it with interleavings

- Even if there are no possible bad interleaving, *your program can still break*

Data Race => Wrong

How?!?

- Optimizations do weird things:
  - Reorder instructions
  - Maintain thread-local copies of shared memory, and don’t update them immediately when changed
  - Optimizations occur both in compiler and hardware

Why?!?

- In a word, ‘speed’
- Can get great time savings this way; otherwise would sacrifice these to support the questionable practice of data races

- Will *not* rearrange insturctions when sequential dependencies come into play; ex: Consider: `x=17; y=x;`
- Regarding updating of shared-memory between threads, there are ways to force updates

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 interleaving (illusion) *if you do your job*

Fixing our example

- Naturally, we can use synchronization to avoid data races
  - Correct ordering now guaranteed because no data races
    - Compiler knows it's not allowed to reorder these in strange ways
  - Now the assertion can't fail

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

A second fix: `volatile`

- Java has `volatile` fields: accesses don't count as data races
  - Accesses will be ordered correctly
  - Updates shared correctly between threads
- Implementation: slower than regular fields, faster than locks
- Really for experts: generally avoid using it; use standard libraries instead
- If you do plan to use `volatile`, look up Java's documentation of it first

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

Code that's wrong

- Here is a more realistic example of code that's wrong
  - Realistic because *I wrote it*, and not with the intention of it being wrong…
  - Data race on `stop`; change made to `stop` in one thread not guaranteed to be updated to others (for reasons of optimization)
  - No *guarantee* Thread 2 will *ever* stop; even after `stop=true` in Thread 1
  - Would "probably work" despite being *wrong*

```
class C {
  boolean stop = false;
  void f() {
    while(!stop) {
      // do something…
    }
  }
  void g() {
    stop = didUserQuit();
  }
}
```

Thread 1: `f()`

Thread 2: `g()`

Fixes: synchronize access or make it volatile

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)
- New performance issues pop up as well:
  - Critical sections too large; covers expensive computation
  - Locks too coarse-grained; loses benefit of concurrent access
- Guidelines for correct use help avoid common pitfalls; stick to them