More Concurrency

CSE 332 25Sp
Lecture 23

Announcements

	Monday	Tuesday	Wed	Thursday	Friday
This Week	Ex 11 (parallel prog) out				TODAY Ex 10 (F-J prog) due Ex 12 (concurrency, GS) out
Next Week	Veteran's Day (no class) OH cancelled or zoom	Ex 11 due			Ex 12 due

Optional readings (Grossman) covers next few weeks of parallelism and concurrency

Bad Interleaving

```
Enqueue(x){
1  if(back==null){
3    back=new Node(x);
5    front=back;
   }
   else{
     back.next=new Node(x);
     back=back.next;
}
```

```
Enqueue(x){
   if(back==null){
2    back=new Node(x);
4    front=back;
6  }
   else{
     back.next=new Node(x);
     back=back.next;
}
```

One Example

```
class BankAccount{
    private int balance=0;
    int getBalance() {return balance;}
    void setBalance(int x) {balance = x;}
    void withdraw(int amount){
        int b = getBalance();
        if(amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b-amount);
    }
…
}
```

Bad Interleavings

Suppose the account has `balance` of 150.

Two threads run: one withdrawing 100, another withdrawing 75.

Find a bad interleaving – what can go wrong?

Bad Interleaving

balance: 150

100

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > b)
  throw new …;
  setBalance(b-amount);
}
```

75

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > b)
  throw new …;
  setBalance(b-amount);
}
```

Bad Interleaving

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > b)
    throw new …;
  setBalance(b-amount);
}
```

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > b)
    throw new …;
  setBalance(b-amount);
}
```

1 2 6

3 4 5

Bad Interleavings

What's the problem?

We stored the result of `balance` locally, but another thread overwrote it after we stored it.

The value became stale.

A Principle

Principle: don't let a variable that might be written become stale.

Ask for it again right before you use it

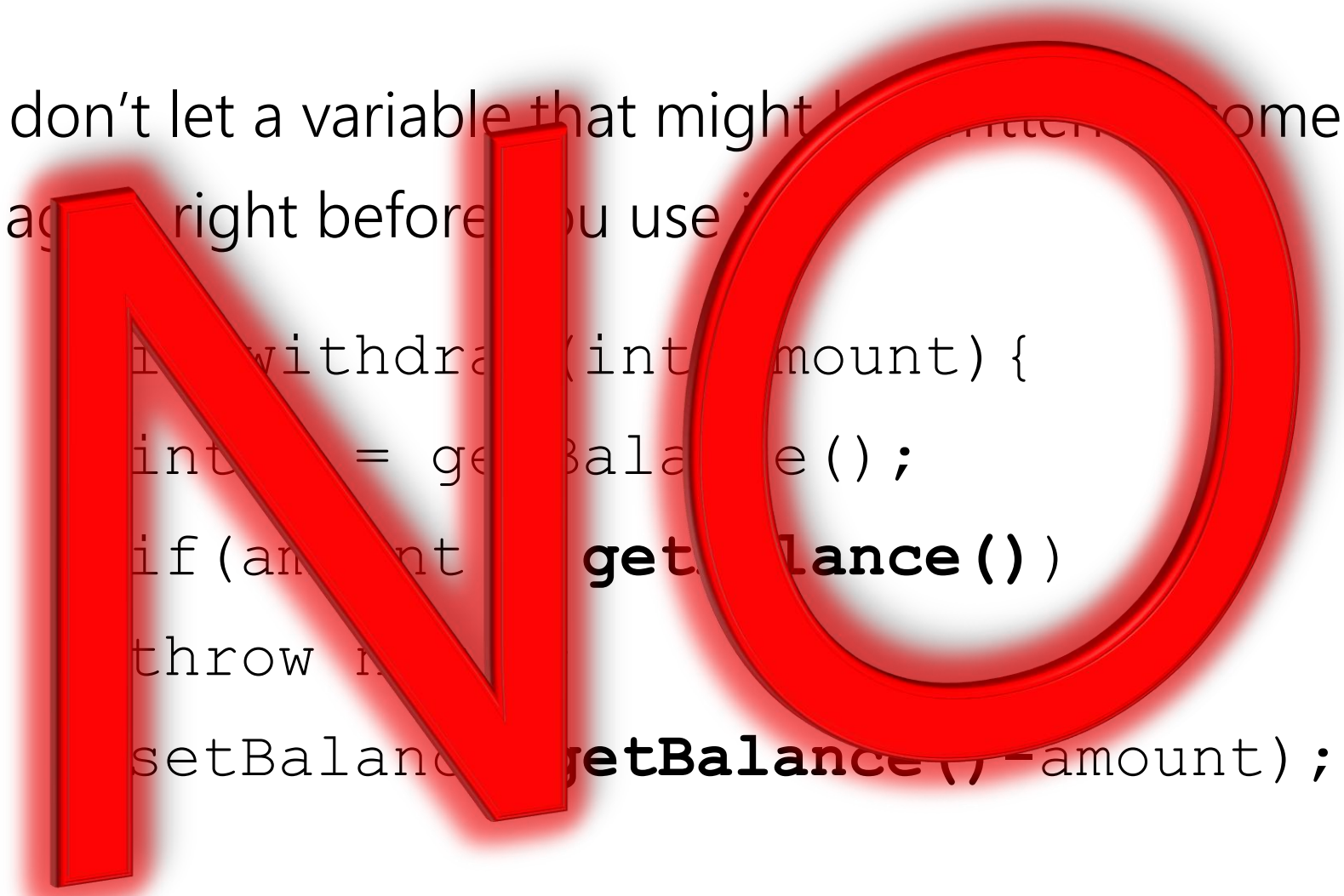
```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
  throw new …;
  setBalance(getBalance()-amount);
}
```

A Principle

Principle: don't let a variable that might [illegible] come stable.

Ask for it ag[illegible] right before [illegible]u use i[illegible]

```
[illegible] withdra[illegible](int [illegible]mount){
  int [illegible] = ge[illegible]Bala[illegible]e();
  if(am[illegible]nt [illegible] getBalance())
    throw [illegible]
  setBalanc[illegible]getBalance()-amount);
```



That's not a real concurrency principle. It doesn't solve anything.

Bad Interleaving

There's still a bad interleaving. Find one.

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
  throw new …;
  setBalance(
    getBalance()-amount);
}
```

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
  throw new …;
  setBalance(
    getBalance()-amount);
}
```

Bad Interleaving

There's still a bad interleaving. Find one.

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
    throw new …;
  setBalance(
      getBalance()-amount);
}
```

1 2 5 7

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
    throw new …;
  setBalance(
      getBalance()-amount);
}
```

3 4 6 8

Bad Interleaving

There's still a bad interleaving. Find one.

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
    throw new …;
  setBalance(
      getBalance()-amount);
}
```

1 2 5 6

```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
    throw new …;
  setBalance(
      getBalance()-amount);
}
```

3 4 7 8

In this version, we can have negative balances without throwing the exception!

A Real Principle

Mutual Exclusion (aka Mutex, aka Locks)

Rewrite our methods so only one thread can use a resource at a time
- All other threads must wait.

We need to identify the "critical section"
- Portion of the code only a single thread can execute at once.

This MUST be done by the programmer. You, the programmer, know what is "correct" for an interleaving and what isn't. The compiler doesn't.

BankAccount v.2

```
class BankAccount{
    private int balance=0;
    private boolean busy = false;
    void withdraw(int amount){
        while(busy){ /* spin wait */ }
        busy = true;
        int b = getBalance();
        if(amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b-amount);
        busy = false;
    }
…
}
```

Does this code work?

Locks

We can still have a bad interleaving.

If two threads see `busy==False` and get past the loop simultaneously.

We need a single operation that
- Checks if `busy` is `False`
- AND sets it to `True` if it is
- Where no other thread can interrupt us.

An operation is **atomic** if no other threads can interrupt it/interleave with it.

Locks

There's no regular java command to do that.

We need a new library

`Lock` (not the real Java class, but will let us understand the principles)

`acquire()` – blocks if lock is unavailable. When lock becomes available, one thread only gets lock.

`release()` – allow another thread to acquire lock.

Need OS level support to implement.

Take an operating systems course to learn more.

Locks

Acq lock in setBalance/getBalance?

```
class BankAccount{
 private int balance = 0;
 private Lock lk = new Lock();
 …
  void withdraw(int amount){
    lk.acquire(); //might block
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
    lk.release();
  }
```

Bug in withdraw

Using Locks

Questions:

What is the critical section (i.e., the part of the code protected by the lock)?

How many locks should we have

- One per `BankAccount` object?
- Two per `BankAccount` object (one in `withdraw` and a different lock in `deposit`)?
- One (static) one for the entire class (shared by all `BankAccount` objects)?

Using Locks

How many locks?

Different locks for withdraw and deposit will lead to bad interleavings.
- The shared resource is `balance` not the methods themselves.

One `lock` for the whole class isn't wrong…but it is a bad design decision.

Only one thread anywhere can do any withdraw/deposit operation; No matter how many bank accounts there are.

There's a tradeoff in how granular you make critical sections:
- Bigger: easier to rule out errors, but fewer threads can work at once.

Using Locks

More Questions:

There is a subtle bug in `withdraw()`, what is it?

Do we need locks for

- `getBalance()`?
- `setBalance()`?
- For the purposes of this question, assume those methods are public.

Using Locks

Bug in `withdraw`:

- When you throw an exception, you still hold onto the lock!

You could release the `lock` before throwing the exception.

Or use `try{} finally{}` blocks

```
try{ critical section }
finally{ lk.release()}
```

Re-entrant Locks

Do we need to lock `setBalance()`?

If it's public, yes.

But now we have a problem:

`withdraw` will acquire the `lock`,

Then call `setBalance`…

Which needs the same `lock`

Re-entrant Locks

Our locks need to be **re-entrant**.

That is, the `lock` isn't held by a single method call

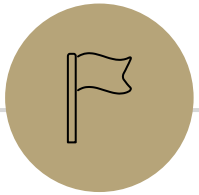
But rather by a thread.

- Execution can **re-enter** another critical section, while holding the same lock.

Lock needs to know which `release` call is the "real" release, and which one is just the end of an inner method call.

Intuition: have a counter. Increment it when you "re-acquire" the lock, decrement when you release. Until releasing on 0 then really release.

Take an operating systems course to learn more.



Some Java Notes

Real Java locks

A re-entrant lock is available in:

`java.util.concurrent.locks.ReentrantLock`

Methods are `lock()` and `unlock()`

synchronized

Java has built-in support for reentrant locks with the keyword `synchronized`

```
synchronized (expression) {
 //Critical section here
 }
```

- Expression must evaluate to an object.
  - Every object “is a lock” in java
  - Lock is acquired at the opening brace and released at the matching closing brace.
  - Released even if control leaves due to throw/return/etc.

`synchronized`

If your whole method is a critical section

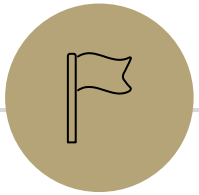
And the object you want for your lock is `this`

You can change the method header to include `synchronized`.

E.g. `private synchronized void getBalance()`

Equivalent of having

`synchronized(this){ }` around entire method body.



Deadlock

Multiple Locks

What happens when you need to acquire more than one lock? What new thing could go wrong with this code?

```
void transferTo(int amt, BankAccount a){
     this.lk.acquire();
     a.lk.acquire();
     this.withdraw(amt);
     a.deposit(amt);
     a.lk.release();
     this.lk.release();
 }
```

Multiple Locks

THREAD 1, FROM ACCT1 TO ACCT2

```
void transferTo(…){
    this.lk.acquire();
    a.lk.acquire();
    this.withdraw(amt);
    a.deposit(amt);
    a.lk.release();
    this.lk.release();
}
```

1

blocks

THREAD 2, FROM ACCT2 TO ACCT1

```
void transferTo(…){
    this.lk.acquire();
    a.lk.acquire();
    this.withdraw(amt);
    a.deposit(amt);
    a.lk.release();
    this.lk.release();
}
```

2

blocks

UH-OH!
Thread 1 needs Thread 2 to let go, but
Thread 2 needs Thread 1 to let go

Deadlock

Deadlock occurs when we have a cycle of dependencies

i.e. we have threads $T_1, \ldots, T_n$ such that

thread T_i is waiting for a resource held by T_{i+1} and

T_n is waiting for a resource held by T_1.

How can we set up our program so this doesn't happen?

Deadlock Solutions

Smaller critical section:

- Acquire bank account 1's lock, withdraw, release that lock
- Acquire bank account 2's lock, deposit, release that lock

Maybe ok here, but exposes wrong total amount in bank while blocking.

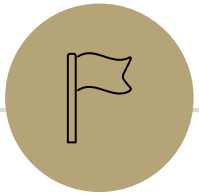
Coarsen the lock granularity

- One lock for all accounts.
- Probably too coarse for good behavior

All methods acquiring multiple locks acquire them in the same order.

- E.g. in order of account number.

More options – take Operating Systems!



Conventional Wisdom

Conventional Wisdom

There are three types of memory

Thread local (each thread has its own copy)

Immutable (no thread overwrites that memory location)

Shared and mutable
- Synchronization needed to control access.

Whenever possible make memory of type 1 or 2.

If you can minimize/eliminate side-effects in your code, you can make more memory type 2.

Conventional Wisdom

Consistent locking:

Every location that reads or writes a shared resource has a lock.

Even if you can't think of a bad interleaving, better safe than sorry.

When deciding how big to make a critical section:

Start coarse grained, and move finer if you really need to improve performance.

Conventional Wisdom

Avoid expensive computations or I/O in critical sections.

If possible release the lock, do the long computation, and reacquire the lock.

Just make sure you haven't introduced a race condition.

Think in terms of what operations need to be atomic.

i.e. consider atomicity first, then think about where the locks go.

Conventional Wisdom

Don’t write your own.

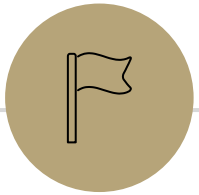
There’s probably a library that does what you need.

Use it.

There are thread-safe libraries like `ConcurrentHashMap`.

No need to do it yourself when experts already did it

- and probably did it better.



Some Java Notes

Real Java locks

A re-entrant lock is available in java.util

java.util.concurrent.locks.ReentrantLock

Methods are `lock()` and `unlock()`

`synchronized`

Java has built-in support for reentrant locks with the keyword `synchronized`

```
synchronized (expression) {
 Critical section
 }
```

- Expression must evaluate to an object.
  - Every object “is a lock” in java
  - Lock is acquired at the opening brace and released at the matching closing brace.
  - Released even if control leaves due to throw/return/etc.

`synchronized`

If your whole method is a critical section

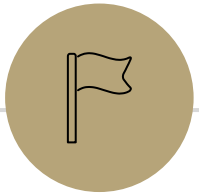
And the object you want for your lock is `this`

You can change the method header to include `synchronized`.

E.g. `private synchronized void getBalance()`

Equivalent of having

`synchronized(this){ }` around entire method body.



Data Race

A distinction

A **Race Condition** is an error in parallel code – it's an error where the output depends on the order of execution of the threads (who wins the race to be executed).

We'll divide into two types

A **data race** is an error where **at (potentially) the same time:**

1. Two threads are writing the same variable.

2. One thread writes to a variable while another is reading it.

A **Bad interleaving**

Is when incorrect behavior (as defined by the user) could result from a particular sequential execution order.

Huh?

Consider this code…

```
class Stack<E>{
  private E[] array = (E[])new Object[SIZE];
  private int index = -1;
  synchronized public Boolean isEmpty() { return index==-1;}
  synchronized public void push(E val) {array[++index]=val;}
  synchronized public E pop() {
    if(isEmpty()) { throw new StackEmptyException(); }
    return array[index--];
  }
  public E peek() { E ans = pop(); push(ans); return ans; }
}
```

What's the problem?

That `peek` is, uh, interesting..

Certainly would work as sequential code (albeit probably bad style).

But with multiple threads…?

Well, there aren't any **data races.** The calls to `push` and `pop` are synchronized. We only ever have one thread touching any data (the underlying array or index variable) at a time.

But it certainly isn't correct! Peek has an intermediate state that (if exposed during a **bad interleaving**) leads to incorrect behavior.

Bad Interleaving 1

THREAD A (PEEK)

```
2  E ans = pop();
4  push(ans);
5  return ans;
```

THREAD B (PUSH + ISEMPTY)

```
1  push(x);
3  boolean b = isEmpty();
```

Logical expectation: If we push (and haven't popped) then the stack is not empty.

This is a bad interleaving, without a data race.

Bad Interleaving 2

THREAD A (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

THREAD B (TWO PUSHES)

```
push(x);
push(y);
```

Logical expectation: Pushed values go in LIFO order

Bad Interleaving 2

THREAD A (PEEK)

```
2  E ans = pop();
4  push(ans);
5  return ans;
```

THREAD B (TWO PUSHES)

```
1  push(x);
3  push(y);
```

Logical expectation: Pushed values go in LIFO order

This is a bad interleaving, without a data race.
Notice, this interleaving would be fine if we just had a generic list!

Bad Interleaving 3

THREAD A (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

THREAD B (TWO PUSHES, POP)

```
push(x);
push(y);
E e = pop();
```

Logical expectation: Popped values come in LIFO order

Bad Interleaving 3

THREAD A (PEEK)

```
3  E ans = pop();
5  push(ans);
6  return ans;
```

THREAD B (TWO PUSHES, POP)

```
1  push(x);
2  push(y);
4  E e = pop();
```

Logical expectation: Popped values come in LIFO order

Bad Interleaving 4

THREAD A (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

THREAD B (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

Logical expectation: Peek on a non-empty heap does not throw an exception

Bad Interleaving 4

THREAD A (PEEK)

```
2 E ans = pop();
3 push(ans);
4 return ans;
```

THREAD B (PEEK)

```
1 E ans = pop();
5 push(ans);
6 return ans;
```

Logical expectation: Peek on a non-empty heap does not throw an exception

The Fix: Don’t allow interleaving!

Peek needs synchronization

Enlarge the critical section to be the whole method.

Ensures no one is even looking at the stack until we push back the element we popped.

The Wrong “fix”: read-only interleavings

Problem so far: peek does writes, creating an intermediate state.

It’s tempting to try to fix the code like this:

`peek`, if “normally” implemented (or `isEmpty`) doesn’t actually write anything. Maybe we can skip the synchronization on those.

Do NOT remove the synchronization. You will create a data race!

`isEmpty` reads the same data `peek` writes (e.g., the `index` variable). That is **definitionally** a data race

We Can’t Keep Getting Away With It

It might look like `isEmpty` or `peek` not being synchronized won’t lead to errors.

After all, that push is just one line! I can’t figure out how to make bad interleavings.

Don’t think just because you can’t figure out a bad interleaving that a data race won’t be a problem.

1. “single steps” aren’t always single steps.

2. A data race is an error **by definition**. The compiler does a lot of optimizations (including reordering sequential code) **assuming** you don’t have data races. If you have them, your code may execute wrong. And good luck figuring that bug out..(see Grossman 7.2)

Summary

Two kinds of race conditions:

Data race (a thread potentially reads while another writes, or two potentially write simultaneously)

Bad Interleaving: exposes intermediate state to other threads, leads to behavior **we** find incorrect.

Data races are never acceptable, even if you can't find a bad interleaving.