Concurrency

Data Structures and Parallelism

Announcements

Today is “recommended deadline” for exercises 8 & 9

- We’re not grading them until the real deadline, but we recommend you finish them by about today.
- They are going to be the most relevant to P3.

P3 checkpoint on Monday.

Sharing Resources

So far we've been writing parallel algorithms that don't share resources.

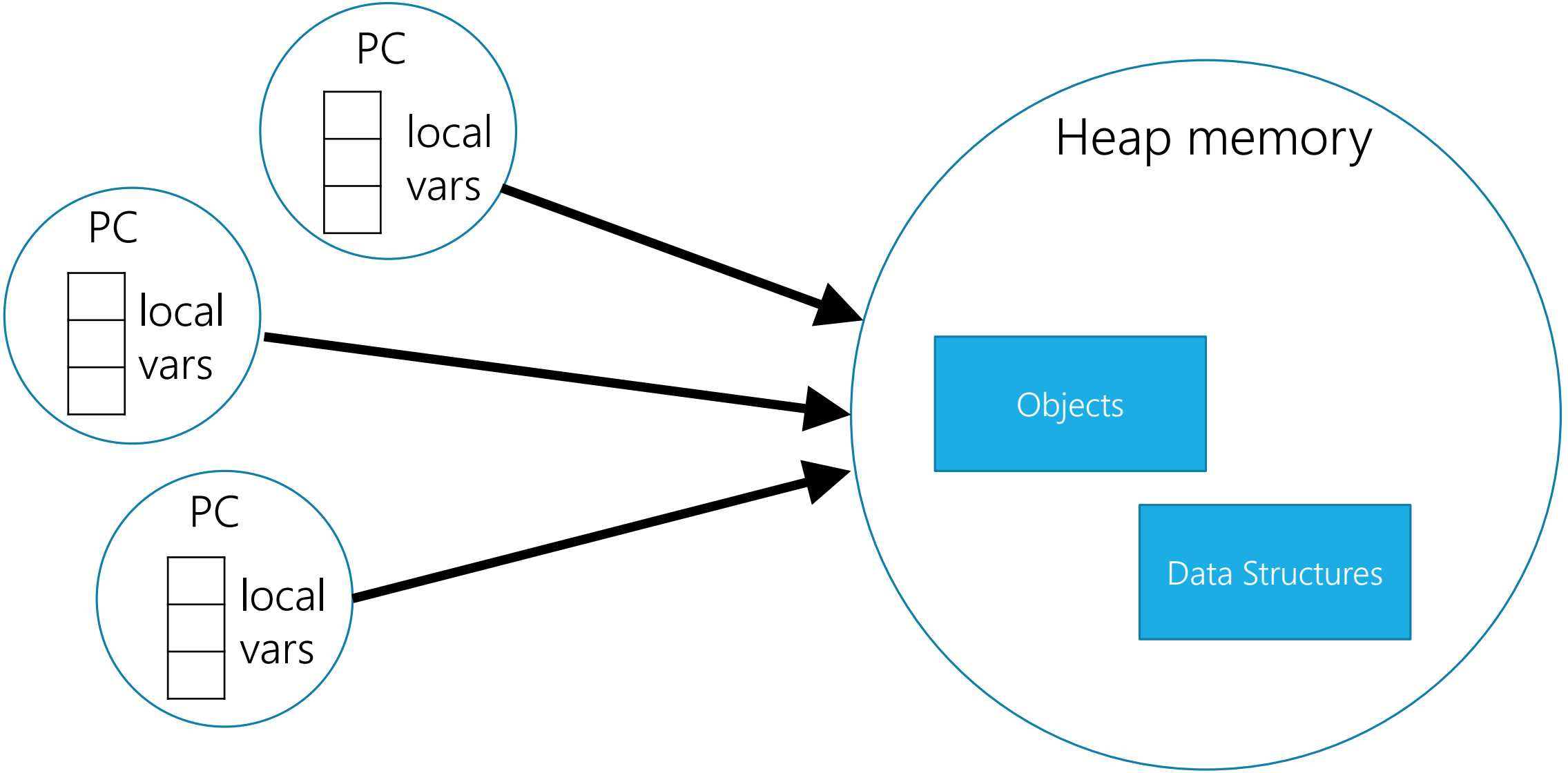
Fork-join algorithms all had a simple structure

- Each thread had memory only it accesses.
- Results of one thread not accessed until joined.
- The **structure** of the code ensured sharing didn't go wrong.
  - No index accessed by more than one thread

Can't use the same strategy when memory overlaps

Thread doing independent tasks on same resources.

Parallel Code



Why Concurrency?

If we're not using them to solve the same big problem, why threads?

Code responsiveness

- One thread responds to GUI, another does big computations

Processor utilization

- If a thread needs to go to disk, can throw another thread on while it waits.

Failure isolation

- Don't want one exception to crash the whole program.

Concurrency

Different threads might access the same resources

In unpredictable orders or even simultaneously

Simultaneous access is rare

- Makes testing very difficult
- Instead we'll be disciplined when writing the code.

In this class, we'll focus on code idioms that are known to work.

Won't talk about java specifics – there are some details in the Grossman notes.

Sharing a Queue

Two threads both want to insert into a queue.

Each has its own program counter, they can each be running different parts of the code simultaneously.

They can arbitrarily “interrupt” each other.

Any possible combination of code from the two threads is allowed, as long as each thread still operates in serial order

What could we consider acceptable outcomes?

What can go wrong?

Bad Interleaving

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

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

Bad Interleaving

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

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

Bad Interleaving

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

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

Good Interleaving

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

```
Enqueue(x){
  if(back==null){
    back=new Node(x);
    front=back;
  }
  else{                          4
    back.next=new Node(x);       5
    back=back.next;              6
}
```

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

```
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);
}
```

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

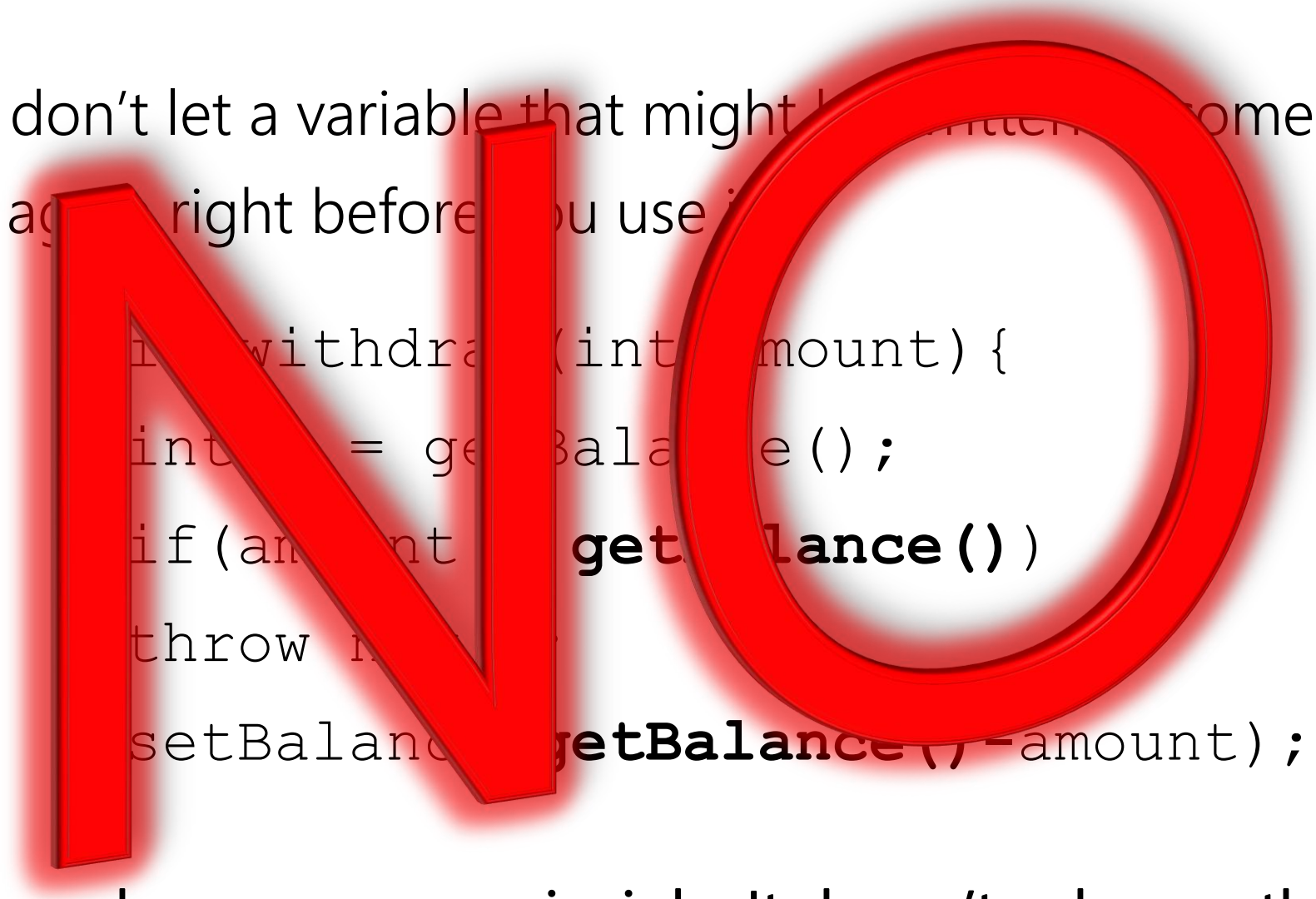
```
void withdraw(int amount){
  int b = getBalance();
  if(amount > getBalance())
  throw new …;
  setBalance(getBalance()-amount);
}
```

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] getB[illegible]lance())
    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);
}
```

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

1 2 3 4 5 6 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.

```
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 a real object)

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

```
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();
  }
```

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)?

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

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

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.

Multiple Locks

What happens when you need to acquire more than one lock?

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

Oh No!

```
void transferTo(…){
  1 this.lk.acquire();
    a.lk.acquire(); 3
    this.withdraw(amt);
    a.deposit(amt);
    a.lk.release();
    this.lk.release();
 }
```

```
void transferTo(…){
    this.lk.acquire(); 2
  4 a.lk.acquire();
    this.withdraw(amt);
    a.deposit(amt);
    a.lk.release();
    this.lk.release();
 }
```

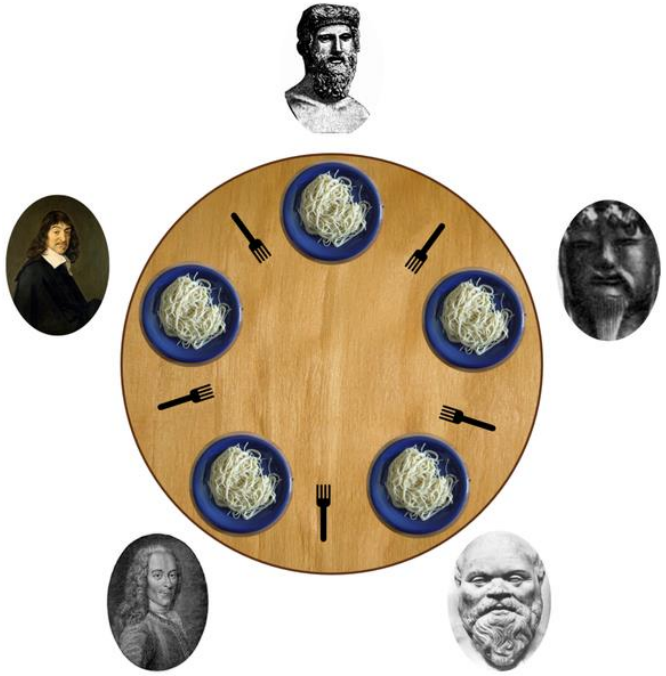
Ex: The Dining Philosophers

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

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 order.

- E.g. in order of account number.

More options – take Operating Systems!

“Conventional Wisdom”

There are three types of memory

1. Thread local (each thread has its own copy)
2. Immutable (no thread overwrites that memory location)
3. 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**.

- When the system's behavior becomes dependent on uncontrollable timing

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.

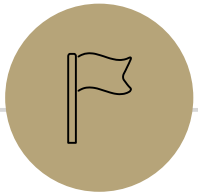
There’s probably a library that does what you need.

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 Example

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

`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.

Simpler Example

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