CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 24: Locks and Deadlocks

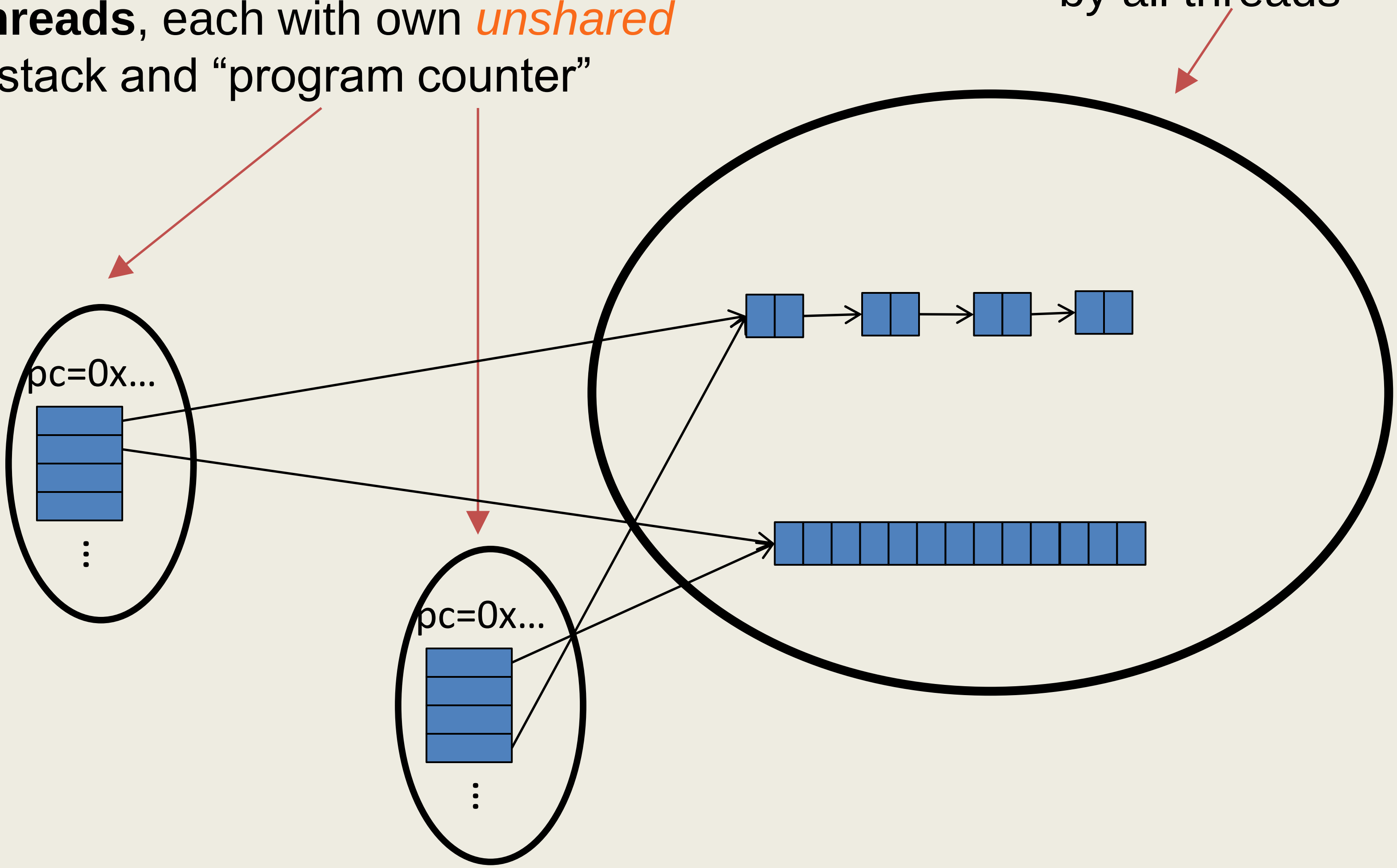
Announcements

- Starting Wednesday – Graph algorithms
  - Readings: Weiss, chapter 9
- Today – 2nd lecture on locks and concurrency

Really sharing memory between Threads

Heap for all objects and static fields, *shared* by all threads

2 Threads, each with own *unshared* call stack and "program counter"



Banking

- Two threads both trying to **`withdraw(100)`** from the same account:
- Assume initial **`balance`** 150

```
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);
  }
  … // other operations like deposit, etc.
}
```

Thread 1

```
x.withdraw(100);
```

Thread 2

```
x.withdraw(100);
```

Race Conditions

A race condition: program executes inconsistently due to ordering of threads

Write-write

T1: a = 0;

T2: a = 1;

Write-read

T1: a = 0;

T2: b = a;

Read-read (not a problem)

T1: b = a;

T2: c = a;

Corrected Version

```
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 …
     setBalance(b – amount);
  }
  // deposit would also use synchronized
}
```

`synchronized` provides a re-entrant lock for each bank account

Locking in Java

Low level locks (available in multiple versions):

```
Lock lk = new Lock();
lk.acquire();
lk.release();
```

Standard Java Locking:

```
Object lk = new Object();
synchronized (lk){
        do stuff;
}
```

Java short cuts:

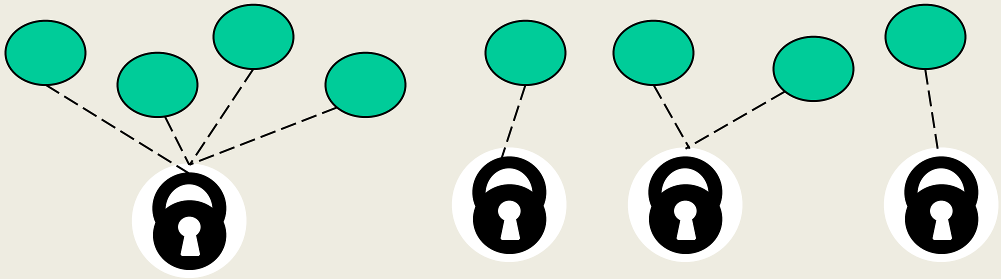
```
synchronized (this) { do stuff;}
synchronized { do stuff; }
synchronized Method { }
```

Locking Guidelines

- Correctness
- Consistency: make it well-defined
- Granularity: coarse to fine
- Critical Sections: make them small, atomic
- Leverage libraries

Consistent Locking

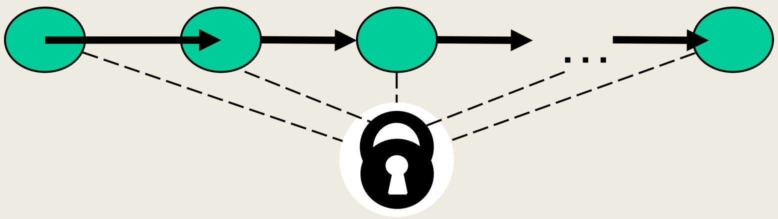
- Clear mapping of locks to resources
  - followed by all methods
  - clearly documented
  - same lock can guard multiple resources



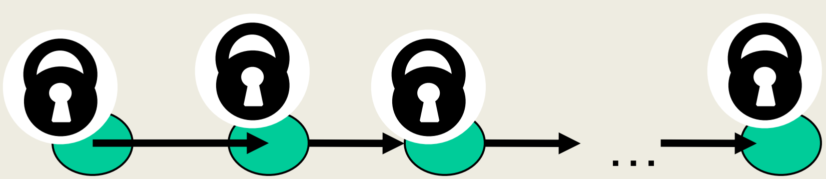
  - what's a resource? Conceptual:
    - object
    - field
    - data structure (e.g., linked list, hash table)

Lock Granularity

- Coarse grained: fewer locks, more objects per lock
  - e.g., one lock for entire data structure (e.g., linked list)



  - advantage:
  - disadvantage:
- Fine grained: more locks, fewer objects per lock
  - e.g., one lock for each item in the linked list



Lock Granularity

- Example: hashtable with separate chaining
  - coarse grained: one lock for whole table
  - fine grained: one lock for each bucket

- Which supports more concurrency for **`insert`** and **`lookup`**?

- Which makes implementing **`resize`** easier?

- Suppose hashtable maintains a **`numElements`** field. Which locking approach is better?

Critical Sections

- Critical sections:
  - how much code executes while you hold the lock?
  - want critical sections to be short
  - make them “atomic”: think about smallest sequence of operations that have to occur at once (without data races, interleavings)

Critical Sections

- Suppose we want to change a value in a hash table
  - assume one lock for the entire table
  - computing the new value takes a long time (“expensive”)

```
synchronized(lock) {
  v1 = table.lookup(k);
  v2 = expensive(v1);
  table.remove(k);
  table.insert(k,v2);
}
```

Critical Sections

- Suppose we want to change a value in the hash table
  - assume one lock for the entire table
  - computing the new value takes a long time (“expensive”)
  - will this work?

```
synchronized(lock) {
  v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lock) {
  table.remove(k);
  table.insert(k,v2);
}
```

Critical Sections

- Suppose we want to change a value in the hash table
  - assume one lock for the entire table
  - computing the new value takes a long time (“expensive”)
  - convoluted fix:

```
done = false;
while(!done) {
  synchronized(lock) {
    v1 = table.lookup(k);
  }
  v2 = expensive(v1);
  synchronized(lock) {
    if(table.lookup(k)==v1) {
      done = true; // I can exit the loop!
      table.remove(k);
      table.insert(k,v2);
}}}
```

Another Bank Operation

- Consider transferring money:

```
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);
  }
}
```

- What can go wrong?

Deadlock

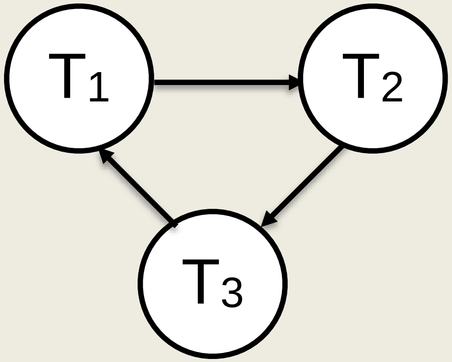
- **`x`** and **`y`** are two different accounts

Time ↓

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

Deadlock = Cycles

- Multiple threads depending on each other in a cycle



  - T2 has lock that T1 needs
  - T3 has lock that T2 needs
  - T1 has lock that T3 needs

- Solution?

How to Fix Deadlock?

- In Banking example

```
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);
  }
}
```

How to Fix Deadlock?

- Separate withdraw from deposit

```
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);
  }
}
```

- Problems?

Possible Solutions

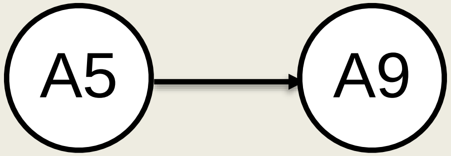
1. **`transferTo`** not synchronized
   - exposes intermediate state after **`withdraw`** before **`deposit`**
   - may be okay here, but exposes wrong total amount in bank

2. Coarsen lock granularity: one lock for all accounts works, but sacrifices concurrent deposits/withdrawals

3. Give every bank-account a unique ID and always acquire locks in the same ID order
   - *Entire program* should obey this order to avoid cycles

Ordering Accounts

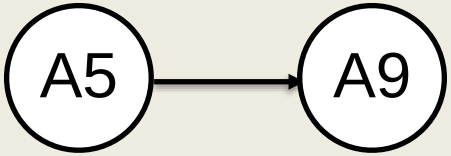
- Transfer from bank account 5 to account 9



1. lock A5
2. lock A9
3. withdraw from A5
4. deposit to A9

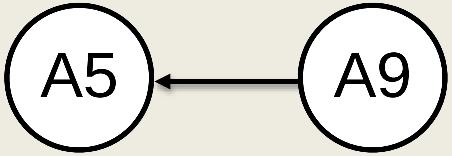
Ordering Accounts

- Transfer from bank account 5 to account 9



1. lock A5
2. lock A9
3. withdraw from A5
4. deposit to A9

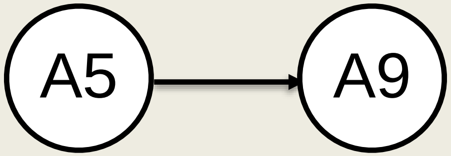
Transfer from bank account 9 to account 5



1. lock
2. lock
3. withdraw from
4. deposit to

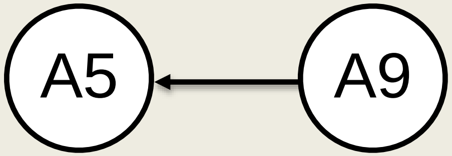
Ordering Accounts

- Transfer from bank account 5 to account 9



1. lock A5
2. lock A9
3. withdraw from A5
4. deposit to A9

Transfer from bank account 9 to account 5



1. lock
2. lock
3. withdraw from
4. deposit to

No interleavings will produce deadlock!

- T1 cannot block on A9 until it has A5
- T2 cannot acquire A9 until it has A5

Banking Without Deadlocks

```
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);
       }}
  }
}
```

Lock Ordering

- Useful in many situations
  - e.g., when moving an item from work queue A to B, need to acquire locks in a particular order

- Doesn't always work
  - not all objects can be naturally ordered
  - Java StringBuffer append is subject to deadlocks
    - thread 1: append string A onto string B
    - thread 2: append string B onto string A

Locking a Hashtable

- Consider a hashtable with
  - many simultaneous **`lookup`** operations
  - rare **`insert`** operations

- What’s the right locking strategy?

Read vs. Write Locks

- Recall race conditions
  - two simultaneous write to same location
  - one write, one simultaneous read
- But two simultaneous reads OK
- Synchronize is too strict
  - blocks simultaneous reads

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

- **`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"

$0 \leq$ writers ≤ 1
$0 \leq$ readers
writers==0 || readers==0

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

Concurrency Summary

- Parallelism is powerful, but introduces new concurrency issues:
  - Data races
  - Interleaving
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion
- Guidelines for correct use help avoid common pitfalls