Thread II

Slides courtesy of Dr. Nilanjan Banerjee

Race condition explained…

- What is the shared object in our example?

  the Bank object??

- Accessing shared objects is also termed as accessing “critical sections” of data

- Lets look at a statement in the critical section.

```
account[to] += amount;
```

Race condition explained..

- Lets decompose the previous statement into assembly and see what it constitutes..

- Load **accounts[to]** to some register
- Add **amount** to that register
- Copy back the result into the memory location

- The problem is that the above three statements are not **atomic (Atomic operation is something that cannot be preempted)**

- Lets see what can potentially happen when two threads simultaneously access this statement…

Race condition explained…

What has happened here?

Load X R1

Add Y Z R1

Store Y in Mem

Thread 1 executes the first two statements and is preempted…

Thread 2 executes all the three statements

Thread 1 returns and executes the third statement

Intuitively how would you solve this problem…

- First solution: allow only one thread to access the "critical section" or "shared object" at one time!!!! ☺

- Easiest way of doing this is using locks…

- java.util.concurrent provides the **Lock** interface and **ReentrantLock** implementation

```
Lock myLock = new ReentrantLock();
myLock.lock(); //acquire a lock..
try {
  critical section…..
} catch(Exception e) {}
myLock.unlock() //give up the lock
```

Properties of locks…

- Only one Thread can acquire a lock at one time
  - Suppose one Thread acquires the lock
  - Second Thread tries to acquire it
  - Second Thread blocks till first Thread releases the lock

- Make sure every lock is accompanied by an unlock statement
  - Else things will block forever ☹

- Reentrant means the same Thread can acquire the lock multiple times
- In our example every Bank object has a separate lock…
  - For different Bank objects the locks are completely different.. They do not collide…

Condition objects!

- Lets refine our Bank example
  - Add a condition that you can transfer data only if there is sufficient balance

Unsynchronized way of doing it

if(accounts[from] > amount) { carry out the transfer…}

Problem

If(accounts[from] > amount) {

Thread preempted…

Some other thread removes the money from the account

Thread scheduled again.. //now the operation is just WRONG!!!

}

Lets see if we can use locking to solve this

```
mylock.lock();

while(account[from] < amount){
 Thread.sleep(100);
}
//do the transfer…

mylock.unlock();
```

What is the problem here?

Only one Thread can acquire the lock..

This might lead to a deadlock…

Java's conditional variables

- Java provides **Conditional variables** with every Lock
- How do you use it::

Using a conditional (you can declare as many as you want)

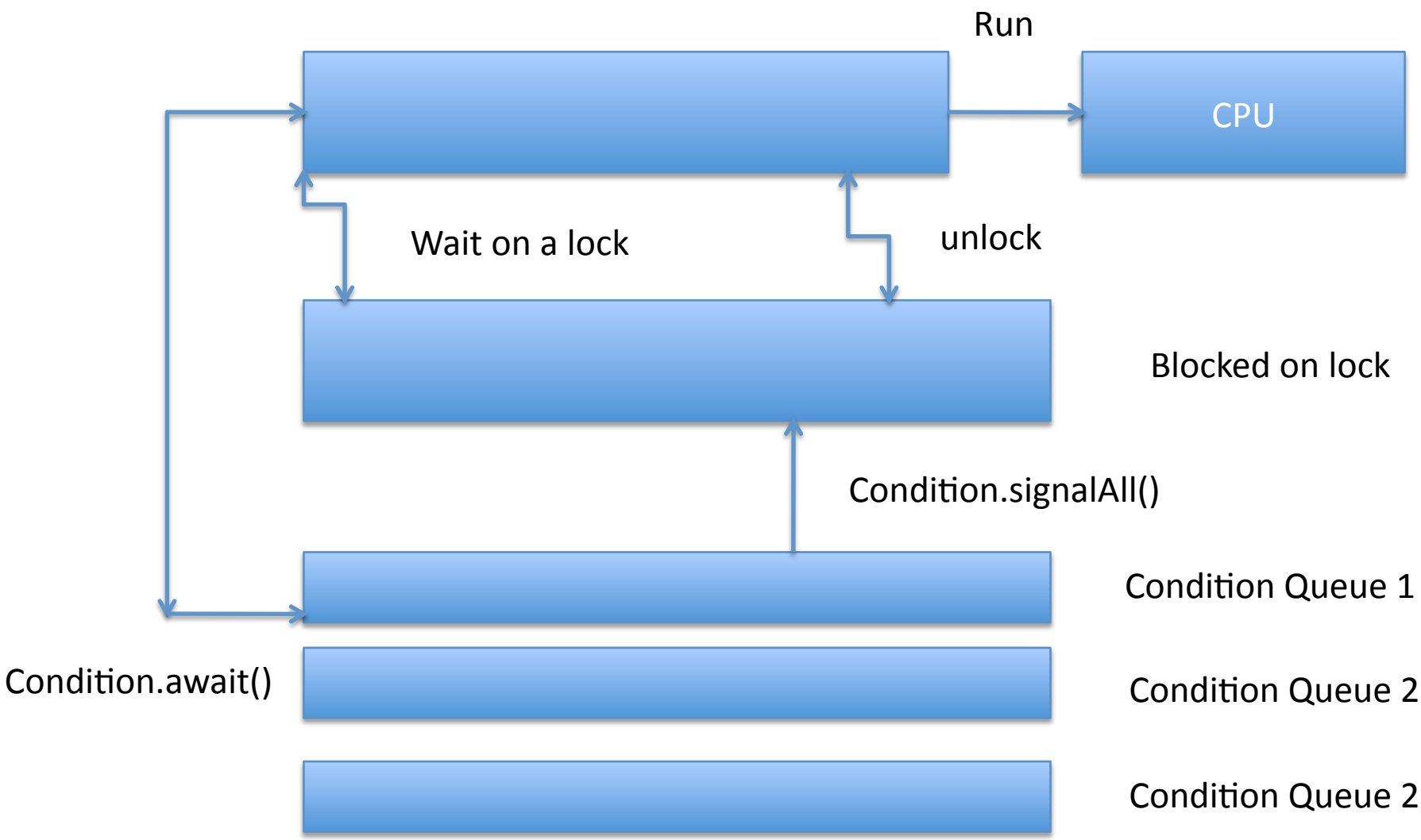
```
private Lock myLock = new ReentrantLock();
public Bank() { //constructor
private Condition sufficientFunds = myLock.newCondition();}
myLock.lock();

while(accounts[from] < amount) {
sufficientFunds.await(); //
}

//Transfer funds..
sufficientFunds.signalAll();
myLock.unlock();
```

Lets take a closer look at the await() and signalAll()

- sufficientFunds.await()
  - It pulls the Thread out of the run queue and places it in a condition queue (specific to that particular condition)
  - Releases the lock that the thread was holding
  - Consequently other Threads that are blocked on that lock can get the CPU!!!

- sufficientFunds.signalAll()
  - Signals all the Threads that are waiting on that particular condition
  - Indication that condition (sufficientFunds) in our case in now satisfied…
  - The Thread can be scheduled if it has the lock.

Difference between conditional wait and waiting on a lock



Summary of locking and conditional

- A lock protects a section of code, allowing only one thread to access the critical section at one time
- A lock manages threads that are trying to enter this “protected” code segment
- A lock can have more than one conditional objects associated with it
- Each conditional object manages threads that have entered a protected code section but cannot proceed due to a “condition”.

The synchronized keyword!

- Locks and Conditions provide you with fine-grained synchronization
  - However there is an easier way of synchronizing object if you are ready to sacrifice some of the flexibility

- The Java language provides a keyword “synchronized” that can be used to make a method thread-safe
  - Every Java object has an **intrinsic** lock
  - Calling a method synchronized uses that **intrinsic** lock
  - Lets take an example

```
//In Bank.java
public synchronized void transfer() {
… method body….
}.
```

.

Take a close look at the **synchronized** method

```
public synchronized void transfer() {
….method body…..
}
```

Equivalent to the following

```
public  void transfer() {
this.intrinsicLock.lock(); // acquire the intrinsic lock
….method body…..
this.intrinsicLock.unlock() //release the instrinsic lock
}
```

Conditionals for the intrinsic lock

Every intrinsic lock is associated with only **ONE** intrinsic conditional

```
public synchronized void transfer() {
while (accounts[from] < amount)  wait();
accounts[from] -= amount;
accounts[to] += amount;
notifyAll();
}
```

Equivalent to the following

```
public  void transfer() {
this.intrinsicLock.lock(); // acquire the intrinsic lock
While(accounts[from] < amount) this.intrinsicLock.getCondition().await();
.. Do stuff…
This.intrinsicLock.getCondition().signalAll()
this.intrinsicLock.unlock() //release the instrinsic lock
}
```

Limitations of using synchronized…

- One lock for the entire object
- One conditional that you can use…
- Pros: very clean code.. Just need to append the method name with the **synchronized** keyword

Concept of a monitor

- One big problem with locks is it is not a very object-oriented concept
- For years developers of thought of providing synchronization without explicit locks

- That is where the concept of a monitor comes into being
  - Monitor is a class that has only private elements
  - Every object has an implicit lock that is acquired when the a method is called and released when the method exits
  - Obj.method() – first acquire the lock and on returning release the lock
  - You can have as many conditionals as possible

How does Java's synchronization mechanism differ from monitors?

Use of **volatile**

- In some cases using locks might be too expensive

```
public synchronized boolean isDone() { return done;}
public synchronized void setDone() { done = true;}
private boolean done;
```

```
public boolean isDone() { return done;}
public void setDone() { done = true;}
private volatile boolean done;
```

There are some concurrent classes

- Java.util.concurrent.*
  - ConcurrentHashMap
  - ConcurrentSkipList
  - ConcurrentSkipListSet

Deadlock situation

- We have already seen one condition where threads might just be deadlocked due to use of locking and using await().

- Another situation due to multiple locks.

Thread 1	Thread 2
Lock1.lock()	Lock2.lock()
Lock2.lock()	Lock1.lock()
….do something…	….do something…
Lock2.unlock()	Lock1.unlock()
Lock1.unlock()	Lock2.unlock()