Lecture 06: Java Threads

Computer Science Department, University of Crete

Multicore Processor Programming

Based on slides by J. Foster, M. Hicks, D. Holmes, and D. Lea

Pratikakis (CSD)

Java Threads

What is a thread?

- Intutively/conceptually:
 - One of possibly many parallel computations occuring within a process
- Implementation:
 - It is a program counter and a stack
 - Heap and static areas are shared among all threads in a process
- All programs have at least one thread (main())

Thread Implementation

A program counter and a stack

- Stack pointer and program counter saved in memory when thread is not running
- Contained in hardware registers (esp, eip) of a core while the thread is running

Tradeoffs involved

- Threads can increase performance
 - Create parallelism on multiprocessors
 - Intuitive way to get concurrent I/O and computation
- Natural fit for some programming paradigms
 - Event processing
 - Simulations
- Tradeoff: increased complexity
 - Need to think about safety, liveness, composability
 - Shared heap, complex interleavings
- Higher resource usage
 - Oversubscription

Thread Programming Model

- Threads exist in many languages
 - C, C++, C#, Java, Smalltalk, Objective Caml, F#, ...
- In many languages (e.g., C, C++) threads a an add-on library
 - Not a part of the language specification
 - See also related paper: "Threads Cannot be Implemented as a Library" posted on website
- Java threads are part of the language specification
 - For more, read paper "The Java Memory Model" for Monday

Java Threads

- Every application has at least one thread, main
 - Started by the JVM to run the application's main() method
- main() thread can create more threads
 - Explicitly: using the Thread class
 - Implicitly: calling libraries that use threads
 - * RMI, Applets, Swing/AWT, ...

Java Threads as Objects

- Java is Object Oriented
 - Uses OO model to express threads too
 - Most OO languages
- To create a Java Thread:
 - Instantiate a Thread object
 - * An object of class Thread or any subclass of Thread
 - Invoke the object's start() method
 - ★ That will create a new execution thread
 - The new thread will start executing the object's run() method
 - Execution will proceed concurrently with the "parent" thread
 - The new thread terminates when it's run() method completes

Running Example: Alarms

• Goal: let's set alarms to be triggered in the future

- Input: time t in seconds, a message m to be printed
- Result: will see message m printed after t seconds

Example: Synchronous Alarms

```
while (true) {
 System.out.print("Alarm> ");
 // read user input
 String line = b.readLine();
 parseInput(line); // sets timeout
 // wait (seconds)
 try {
   Thread.sleep(timeout * 1000);
 } catch (InterruptedException e) { }
 System.out.println("(" + timeout + ") " + msg);
```

Make it threaded (1)

```
public class AlarmThread extends Thread {
 private String msg = null;
 private int timeout = 0;
 public AlarmThread(String msg, int time) {
   this.msg = msg;
   this.timeout = time:
 }
 public void run() {
   trv {
    Thread.sleep(timeout * 1000);
   } catch (InterruptedException e) { }
   System.out.println("(" + timeout + ") " + msg);
```

Make it threaded (2)

```
while (true) {
 System.out.print("Alarm> ");
 // read user input
 String line = b.readLine();
 parseInput(line); // sets timeout
 if (m != null) {
   // start alarm thread
   Thread t = new AlarmThread(msg, timeout);
   t.start();
...
```

Alternative: The Runnable Interface

- Extending Thread prohibits a different parent
- Instead, implement interface Runnable
 - Declares that the class has a void run() method
- Construct a Thread from a Runnable
 - > Constructor Thread(Runnable target)
 - > Constructor Thread(Runnable target, String name)

Example, revisited (1)

```
public class AlarmRunnable implements Runnable {
 private String msg = null;
 private int timeout = 0;
 public AlarmRunnable(String msg, int time) {
   this.msg = msg;
   this.timeout = time:
 }
 public void run() {
   trv {
    Thread.sleep(timeout * 1000);
   } catch (InterruptedException e) { }
   System.out.println("(" + timeout + ") " + msg);
```

Example, revisited (2)

```
while (true) {
 System.out.print("Alarm> ");
 // read user input
 String line = b.readLine();
 parseInput(line); // sets timeout
  if (m != null) {
   // start alarm thread
   Thread t = new Thread(new AlarmRunnable(msg, timeout));
   t.start();
}
...
```

Passing parameters

- run() does not take parameters
- To "pass parameters" to the new thread store them as private fields
 - In the extended class
 - In the Runnable object
 - E.g., the timeout and msg private fields of the AlarmThread class

Concurrency

- A *concurrent* program is one that has multiple threads active at the same time
 - It may run on one CPU
 - ★ The CPU alternates between threads
 - ★ Thread scheduler decides details
 - Context-switching may happen at any time
 - It may be run in *parallel* on a multicore machine
 - ★ Each CPU core runs a thread
 - May run more than one thread per CPU core
 - * Threads may resume on the same or on different CPU core
 - ★ Scheduling policy may differ by JVM

Concurrency and Shared Data

• Concurrency is easy if threads do not interact

- Each thread does its own thing, uses its own objects
- Typically, threads need to communicate with each other
- Communication by sharing data
 - Many threads can access the heap simultaneously
 - Communication via writing and reading the same objects
 - Writes and reads may interleave arbitrarily
 - Hardware may reorder instructions, messages
 - Scheduler may interleave threads
 - ★ Compiler may reorder code
 - May get problems if we are not careful!

Data Race Example

```
public class Example extends Thread {
 private static int counter = 0; // shared state
 public void run() {
   int y = counter;
   counter = y + 1;
 }
 public static void main(String args[]) {
   Thread t1 = new Example();
   Thread t2 = new Example();
   t1.start();
   t2.start();
```

What happens?

- Different schedules lead to different results
 - This is a Data Race or Race Condition
- A thread is preempted in the middle of an operation
- Or, parallel instructions from the other thread run in between its instructions
- Reading and writing counter was supposed to be *atomic*
 - Atomic (conceptually): to appear instantaneous
 - To happen with no interference from other threads
 - In atomic code, thread t1 should "see" no values written by thread t2 and vice versa
- These bugs can be extremely hard to reproduce
- So, hard to debug
- Depends on timing of scheduler, or hardware

Question

If, instead of

int y = counter; counter = y + 1;

we had written

counter++;

- Would the result be different?
- Answer: NO
- Do not trust your intuition on whether an instruction is atomic or not
- May be on some machines, not on others

Synchronization

- Refers to mechanisms that control the execution order of operations accross threads
- Conceptually:
 - Threads produce executions with all possible interleavings, timings
 - Some such executions are correct, some are incorrect
 - Synchronization mechanisms remove incorrect executions by restricting interleavings
- Different languages use different mechanisms to synchronize threads
- Java has several such mechanisms
- We will look at locks first

Java Locks

```
interface Lock {
    void lock();
    void unlock();
    ....
}
class ReentrantLock implements Lock { ... }
```

• Only one thread can hold a lock at any time

 Other threads that try to acquire the same lock will block (or become suspended) until the lock becomes available

• Reentrant lock: can be re-acquired by the same thread

- As many times as desired
- No other thread may acquire the lock until it has been released the same number of times it was acquired
- Hence, re-entry (needs re-exit)

Avoiding Interference: Synchronization

```
public class Example extends Thread {
    private static int counter = 0;
    static Lock lock = new ReentrantLock();

    public void run() {
        lock.lock();
        int y = counter;
        counter = y + 1;
        lock.unlock();
    }
    ...
}
```

Different locks do not interact

```
static int counter = 0;
static Lock I = new ReentrantLock();
static Lock m = new ReentrantLock();
public void inc1() {
 l.lock();
 counter++:
 I.unlock();
}
public void inc2() {
 m.lock();
 counter++:
 m.unlock();
}
```

- This program has a race condition
- Threads only block if they try to acquire a lock held by another thread

Pratikakis (CSD)

Question

static int counter = 0; static int x = 0;

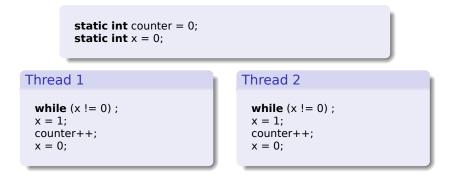
Thread 1

while (x != 0); x = 1; counter++; x = 0;

Thread 2

while (x != 0); x = 1; counter++; x = 0;

Question



- Threads may be interrupted after the while but before writing to x
- Both would think they hold the lock!
- This is busy waiting: consumes lots of processor cycles

Reentrant Lock Example

```
static int c = 0;
static Lock l =
    new ReentrantLock();
void inc() {
    l.lock();
    c++;
    l.unlock();
}
```

```
void returnAndInc() {
    int temp;
    l.lock();
    temp = c;
    inc();
    l.unlock();
```

 Reentrancy is useful because each method can acquire/release locks as it needs

- No need to worry about whether callers already hold locks
- Keeps code simpler, readable

Deadlock

- Deadlock occurs when no thread can run because all threads are waiting for a lock
- No thread runs, so no thread can release any lock to enable another to run

Lock I = **new** ReentrantLock(); Lock m = **new** ReentrantLock();

Thread 1	Thread 2
l.lock();	m.lock();
m.lock();	l.lock();
m.unlock();	l.unlock();
l.unlock();	m.unlock();

Pratikakis	(CSD)

Deadlock, cont.

- Some schedules work fine
 - Thread 1 runs to completion, then thread 2
- What if...
 - Thread 1 acquires 1
 - Thread 2 acquires m
- Deadlock:
 - Thread 1 is trying to acquire m
 - Thread 2 is trying to acquire 1
 - Neither can, because the other thread has it

The wait graph

- The wait graph
 - Each thread is a node
 - Each lock is a node
 - Draw edge l to Thread1 if it has the lock
 - Draw edge Thread1 to m when it tries to acquire the lock
 - The wait graph captures a single point in the execution
- Deadlock occurs when there is a cycle
- Program has deadlock if any execution can produce a cycle
- Difficult to reproduce, difficult to debug

Another Deadlock Example

- Lock l not released along all possible execution paths
- File exception may leave lock acquired by the thread
 - Likely to cause deadlock later
 - Even more difficult to debug, deadlock will appear in possibly unrelated point in the execution

Solution: use "finally"

```
static Lock I = new ReentrantLock();
void f() throws Exception {
  I.lock();
 try {
   FileInputStream f = new FileInputStream("file.txt");
   // do something with f
   f.close();
  }
 finally {
   // this code is executed always,
   // regardless of how we exit the try block
   I.unlock();
```

Synchronized blocks

- This pattern is very common
 - Acquire a lock, do something, release the lock under any circumstances (e.g., finally)
- Java has a special language construct for this pattern
 - synchronized (obj) { body }
 - Every Java object has an implicit associated lock
 - Obtain the lock associated with obj
 - Execute body
 - Release the lock when the syntactic scope is exited
 - ★ Even in the case of exception or explicit return

Example

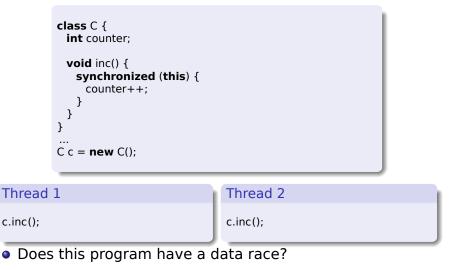
```
static Object o = new Object();
void f() throws Exception {
    synchronized (o) {
        FileInputStream f = new FileInputStream("file.txt");
        // do something with f
        f.close();
    }
}
```

- Lock associated with object o acquired before body is executed
 - Released when exiting the block scope, even when exception is thrown

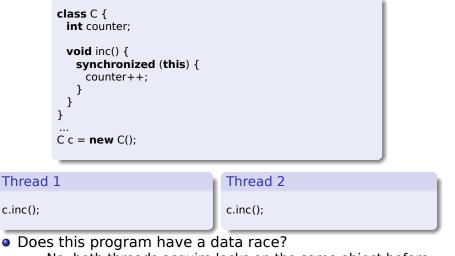
Object locks

- An object and its associated lock are different!
- Holding the lock does not stop anyone else from accessing that object, calling methods, etc.

Example (1)



Example (1)

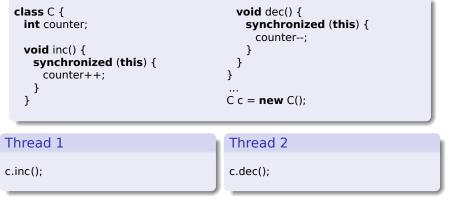


 No, both threads acquire locks on the same object before accessing the shared data

Pratikakis (CSD)

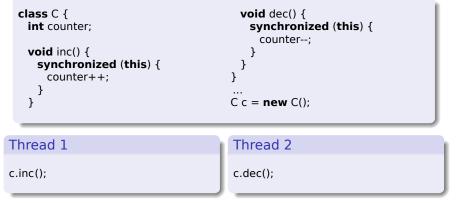
Java Threads

Example (2)



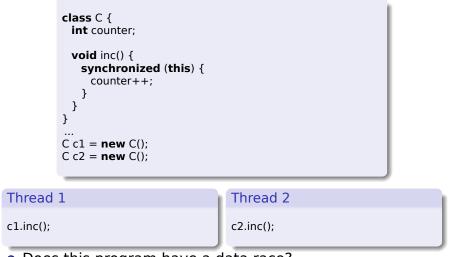
Does this program have a data race?

Example (2)



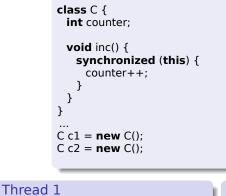
- Does this program have a data race?
 - No, both threads acquire locks on the same object before accessing the shared data

Example (3)



Does this program have a data race?

Example (3)



c1.inc();

Thread 2

c2.inc();

- Does this program have a data race?
 - No, threads acquire different locks, but they write to different objects

Pratikakis (CSD)

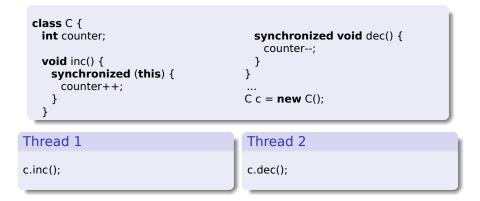
Java Threads

Synchronized Methods

- Mark a method as synchronized
 - The same as synchronizing on this in the body of the method
 - Easier way to express the same pattern
- The following programs are the same:

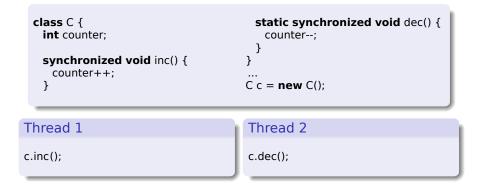
```
class C {
    int counter;
    void inc() {
        synchronized (this) {
            counter++;
        }
     }
    }
}
class C {
    int counter;
    synchronized void inc() {
            counter++;
        }
    }
}
```

Synchronized methods: Example



Synchronized static methods

- Warning: Static methods lock class object!
 - There is no this to lock



Thread Scheduling

- When multiple threads share a CPU core
 - When should the current thread stop running?
 - What thread should run next?
- A thread can voluntarily yield() the CPU core
 - Call to yield() may be ignored
- Preemptive schedulers
 - Can de-schedule a running thread at any time
 - Not all JVMs use pre-emptive schedulers
 - A thread stuck in a loop may never yield automatically
 - Sometimes good to yield() manually inside loops
- Threads are de-scheduled when they block

Lock, I/O, etc.

Thread Lifecycle

Running thread goes through several different phases

- New: Created but not yet started
- Runnable: Currently running or able to run on a free CPU core
- Blocked: Waiting for I/O, lock, or other synchronization operation
- Sleeping: Paused for a user-specified interval
- Terminated: Completed, not running

Which Thread Runs Next?

- Look at all runnable threads
 - Any thread just became unblocked?
 - ★ A lock was released
 - ★ I/O became available
 - ★ Finished sleeping
- Pick a thread and run it
 - Can try to influence priority with setPriority(int)
 - Higher priority value gets preference
 - Probably no need to set priority

Interesting Thread Methods

- void join() throws InterruptedException
 - Waits for a thread to finish
- static void yield()
 - Current thread gives up the CPU core
- static void sleep(long milliseconds) throws InterruptedException
 - Current thread sleeps for the given time
- static Thread currentThread()
 - Returns the Thread object of the currently executing thread

Example: Alarm

```
while (true) {
 System.out.print("Alarm> ");
 // read user input
 String line = b.readLine();
 parseInput(line); // sets timeout
 // wait (seconds) asynchronously
 if (msg != null) {
   // start alarm thread
   Thread t = new AlarmThread(msg, timeout);
   t.start();
   // wait for the thread to complete
   t.join();
...
```

Daemon Threads

void setDaemon(boolean on)

- Marks thread as a daemon thread
- Must be set before thread started
- By default, each new thread acquires the status of the thread that spawned it
- Program execution terminates when no threads left running
 - Except daemon threads

Key Ideas

• Multiple threads running simultaneously

- Either truly in multiple CPU cores
- Or scheduled on a single processor
 - A running thread can be pre-empted at any time
- Or a combination of these
- Threads can share data
 - In Java, only fields can be shared
 - Need to prevent interference
 - ★ Good practice 1: Hold a lock when accessing shared data
 - Good practice 2: Do not release the lock until shared data is in a valid state
 - Overuse of synchronization can create deadlocks
 - * Rule of thumb: No deadlock if only one lock acquired at a time

Producer – Consumer Design Pattern

• Suppose two threads communicate with a shared variable

- E.g., some kind of buffer holding messages
- One thread produces input to the buffer
- One thread consumes data from the buffer
- How do we implement this?
 - ★ Use condition variables

Conditions

```
interface Lock { Condition newCondition(); ... }
interface Condition {
    void await();
    void signalAll();
    ...
}
```

- Condition created using a Lock object
- await() called with lock acquired
 - Releases the lock
 - ★ But not any other locks held by this thread
 - Adds this thread to wait set for the lock
 - Blocks the thread
- signalAll() called with lock acquired
 - Resumes all threads on lock's wait set
 - Those threads try to reacquire lock before continuing
 - ★ Only one will succeed
 - If acquiring thread had blocked in await() it continues with lock acquired

Pratikakis (CSD)

Example: Producer – Consumer

Lock lock = ReentrantLock(); Condition ready = lock.newCondition(); **boolean** valueReady = **false**; Object value;

Thread 1

```
void produce(Object o) {
    lock.lock();
    while (valueReady)
    ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}
```

Thread 2

```
Object consume() {
lock.lock();
while (!valueReady)
ready.await();
Object o = value;
valueReady = false;
ready.signalAll();
lock.unlock();
```

Prefer this design pattern

- This is the right solution to the problem
 - It may be tempting to try to use locks directly
 - Very hard to get right
 - Problems with other implementations often very subtle
 - * E.g., double-checked locking is broken

Example: BROKEN code (1)

Lock lock = **new** ReentrantLock(); **boolean** valueReady = **false**; Object value;

Thread 1

```
void produce(Object o) {
    lock.lock();
    while (valueReady);
    value = o;
    valueReady = true;
    lock.unlock();
}
```

Thread 2

```
Object consume() {
    lock.lock();
    while (!valueReady);
    Object o = value;
    valueReady = false;
    lock.unlock();
}
```

- This code is broken
- Deadlock: threads wait while holding the lock, no progress

Example: BROKEN code (2)

Lock lock = **new** ReentrantLock(); **boolean** valueReady = **false**; Object value;

Thread 1

```
void produce(Object o) {
    while (valueReady);
    lock.lock();
    value = o;
    valueReady = true;
    lock.unlock();
}
```

Thread 2

Object consume() {
 while (!valueReady);
 lock.lock();
 Object o = value;
 valueReady = false;
 lock.unlock();
}

- This code is broken, too
- Data Race: valueReady accessed without holding the lock

Pratikakis (CSD)

Java Threads

Example: BROKEN code (3)

Lock lock = **new** ReentrantLock(); Condition ready = lock.newCondition(); **boolean** valueReady = **false**; Object value;

Thread 1

```
void produce(Object o) {
    lock.lock();
    if (valueReady) ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}
```

Thread 2

```
Object consume() {
lock.lock();
if (!valueReady) ready.await();
Object o = value;
valueReady = false;
ready.signalAll();
lock.unlock();
}
```

- This code is broken, too!
- Correctness: What if there are multiple producers and consumers?

Pratikakis (CSD)

The Condition Interface

```
interface Condition {
   void await();
   boolean await(long time, TimeUnit unit);
   void signal();
   void signalAll();
   ...
}
```

await(t, u) waits for time t and then gives up

- Boolean result: false if the waiting time detectably elapsed before return from the method, else true
- signal() wakes up only one waiting thread
 - Tricky to get right
 - Have all waiting threads be equal, handle exceptions correctly
 - Highly recommended to use signalAll()

Issues with await and signalAll

await() must be in a loop

- Do not assume that when it returns, the condition holds
- Maybe many threads "consume" the condition
- Avoid holding other locks when waiting
 - await() only gives up locks on the object you wait on
- Cannot have a Condition object on two locks
- Can have two Condition objects on the same lock

Blocking Queues

Interface for Producer-Consumer pattern

```
interface Queue<E> extends Collection<E> {
    boolean offer(E x); // produce
    // waits for queue to have capacity
    E remove(); // consume
    // waits for queue to become non-empty
    ...
}
```

Two useful implementations

- LinkedBlockingQueue (FIFO, may be bounded)
- ArrayBlockingQueue (FIFO, bounded)
- A few more, look up in documentation

Wait and NotifyAll (1)

- Old synchronization (Java 1.4)
- In Java 1.4, use synchronized keyword on an object to acquire lock
 - Objects have an associated lock
 - Objects have an associated wait set

Wait and NotifyAll (2)

• o.wait()

- Must hold the lock associated with o (inside synchronized block)
- Releases the lock
 - ★ No other locks
- Adds the thread to the wait set of the lock
- Blocks the thread
- On return, the lock will again be acquired

o.notifyAll()

- Must hold the lock associated with o
- Resumes all threads in the wait set of o
- These threads will try to reacquire lock before continuing (e.g., before wait() returns)

Producer – Consumer in Java 1.4

```
public class ProducerConsumer {
 private boolean valueReady = false;
 private Object value;
 synchronized void produce(Object o) {
   while (valueReady) wait();
   value = o;
   valueReady = true;
   notifyAll();
 }
 synchronized Object consume() {
   while (!valueReady) wait();
   valueReady = false;
   Object o = value;
   notifyAll();
   return o:
```

InterruptedException

- Exception thrown if certain concurrency operations are interrupted
 - wait(), await(), sleep(), join(), and lockInterruptibly()
 - Also thrown if one of these is called with interrupt flag set
- The exception is *not* thrown when blocked on Java 1.4 lock or on I/O

```
class Object {
        void wait() throws InterruptedException;
      interface Lock {
        void lock();
        void lockInterruptibly() throws InterruptedException;
      interface Condition {
        void await() throws InterruptedException;
        void signalAll();
Pratikakis (CSD)
                                Java Threads
```