CMSC 330: Organization of Programming Languages

Threads

Synchronization

- Refers to mechanisms allowing a programmer to control the execution order of some operations across different threads in a concurrent program.
- Different languages have adopted different mechanisms to allow the programmer to synchronize threads.
- Java has several mechanisms; we'll look at locks first.

Locks (Java 1.5)

```
interface Lock {
  void lock();
  void unlock();
  ... /* Some more stuff, also */
}
class ReentrantLock implements Lock { ... }
```

- Only one thread can hold a lock at once
  - Other threads that try to acquire it *block* (or become suspended) until the lock becomes available
- *Reentrant lock* can be reacquired by same thread
  - As many times as desired
  - No other thread may acquire a lock until has been released same number of times it has been acquired

Avoiding Interference: Synchronization

```
public class Example extends Thread {
  private static int cnt = 0;
  static Lock lock = new ReentrantLock();
  public void run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
    }
  }
  ...
}
```

Lock, *for protecting the shared state*

Acquires *the lock; Only succeeds if not held by another thread*

Releases *the lock*

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

Shared state cnt = 0

T1 acquires the lock

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

Shared state cnt = 0

y = 0

T1 reads cnt into y

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

y = 0

Shared state cnt = 0

T1 is preempted. T2 attempts to acquire the lock but fails because it's held by T1, so it blocks

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

y = 0

Shared state **cnt = 1**

T1 runs, assigning to cnt

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

y = 0

Shared state cnt = 1

T1 releases the lock and terminates

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

y = 0

Shared state cnt = 1

T2 now can acquire the lock.

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

y = 0

Shared state cnt = 1

T2 reads cnt into y.

y = 1

Applying Synchronization

```
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

y = 0

Shared state **cnt = 2**

T2 assigns cnt, then releases the lock

y = 1

Different Locks Don't Interact

```
static int cnt = 0;
static Lock l =
    new ReentrantLock();
static Lock m =
    new ReentrantLock();

void inc() {
  l.lock();
  cnt++;
  l.unlock();
}
```

```
void inc() {
  m.lock();
  cnt++;
  m.unlock();
}
```

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

Reentrant Lock Example

```
static int cnt = 0;
static Lock l =
    new ReentrantLock();

void inc() {
  l.lock();
  cnt++;
  l.unlock();
}
```

```
void returnAndInc() {
  int temp;

  l.lock();
  temp = cnt;
  inc();
  l.unlock();
}
```

- Reentrancy is useful because each method can acquire/release locks as necessary
  - No need to worry about whether callers have locks
  - Discourages complicated coding practices

Deadlock

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

This code can deadlock…
-- when will it work?
-- when will it deadlock?

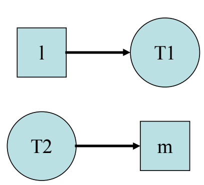
```
Lock l = new ReentrantLock();
Lock m = new ReentrantLock();
```

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

Deadlock (cont'd)

- Some schedules work fine
  - Thread 1 runs to completion, then thread 2
- But what if...
  - Thread 1 acquires lock l
  - The scheduler switches to thread 2
  - Thread 2 acquires lock m
- Deadlock!
  - Thread 1 is trying to acquire m
  - Thread 2 is trying to acquire l
  - And neither can, because the other thread has it

Wait Graphs

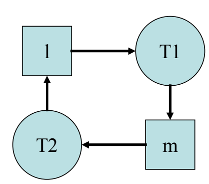


Thread T1 holds lock l

Thread T2 attempting to acquire lock m

Deadlock occurs when there is a cycle in the graph

Wait Graph Example



T1 holds lock on l
T2 holds lock on m
T1 is trying to acquire a lock on m
T2 is trying to acquire a lock on l

Another Case of Deadlock

```
static Lock l = new ReentrantLock();

void f () throws Exception {
  l.lock();
  FileInputStream f =
     new FileInputStream("file.txt");
  // Do something with f
  f.close();
  l.unlock();
}
```

- l not released if exception thrown
  - Likely to cause deadlock some time later

Solution: Use Finally

```
static Lock l = new ReentrantLock();

void f () throws Exception {
  l.lock();
  try {
    FileInputStream f =
       new FileInputStream("file.txt");
    // Do something with f
    f.close();
  }
  finally {
    // This code executed no matter how we
    // exit the try block
    l.unlock();
  }
}
```


Synchronized

- This pattern is really common
  - Acquire lock, do something, release lock under any circumstances after we're done
    - Even if exception was raised etc.
- Java has a language construct for this
  - `synchronized (obj) { body }`
    - Every Java object has an implicit associated lock
  - Obtains the lock associated with `obj`
  - Executes ***body***
  - Release lock when scope is exited
    - Even in cases of exception or method return

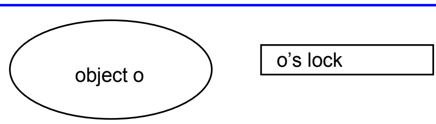
Example

```
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 o acquired before body executed
  - Released even if exception thrown

Discussion



- An object and its associated lock are different!
  - Holding the lock on an object does not affect what you can do with that object in any way
  - Ex:

```
synchronized(o) { ... }   // acquires lock named o
o.f ();                   // someone else can call o's methods
o.x = 3;                  // someone else can read and write o's fields
```


Example: Synchronizing on this

```
class C {
   int cnt;

   void inc() {
     synchronized (this) {
       cnt++;
     }
   }
}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.inc();
```

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

Example: Synchronizing on this (cont'd)

```
class C {
   int cnt;

   void inc() {
     synchronized (this) {
       cnt++;
     }
   }

   void dec() {
     synchronized (this) {
       cnt--;
     }
   }
}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.dec();
```

- Data race?
  - No, threads acquire locks on the same object before they access shared data

Example: Synchronizing on this (cont'd)

```
class C {
   int cnt;

   void inc() {
     synchronized (this) {
       cnt++;
     }
   }
}
```

```
C c1 = new C();
C c2 = new C();
```

```
Thread 1
c1.inc();
```

```
Thread 2
c2.inc();
```

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

Synchronized Methods

- Marking method as synchronized same as synchronizing on this in body of the method
  - The following two programs are the same

```
class C {
   int cnt;

   void inc() {
     synchronized (this) {
       cnt++;
     }
   }
}
```

```
class C {
   int cnt;

   synchronized void inc(){
     cnt++;
   }
}
```


Synchronized Methods (cont'd)

```
class C {
   int cnt;

   void inc() {
     synchronized (this) {
       cnt++;
     }
   }

   synchronized void dec() {
     cnt--;
   }
}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.dec();
```

- Data race?
  - No, both acquire same lock

Synchronized Static Methods

- Warning: Static methods lock class object
  - There's no this object to lock

```
class C {
   static int cnt;

   void inc() {
     synchronized (this) {
       cnt++;
     }
   }

   static synchronized void dec() {
     cnt--;
}  }
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
C.dec();
```


What can be synchronized?

- code blocks
- methods
  - subclasses do not inherit synchronized keyword
  - interface methods cannot be declared synchronized
- NOT fields
  - but you could write synchronized accessor methods
- NOT constructors
  - but you could include synchronized code blocks
- objects in an array

Thread Scheduling

- When multiple threads share a CPU...
  - When should the current thread stop running?
  - What thread should run next?
- A thread can voluntarily yield() the CPU
  - Call to yield may be ignored; don't depend on it
- *Preemptive schedulers* can de-schedule the current thread at any time
  - Not all JVMs use preemptive scheduling, so a thread stuck in a loop may *never* yield by itself. Therefore, put yield() into loops
- Threads are de-scheduled whenever they block (e.g., on a lock or on I/O) or go to sleep

Thread Lifecycle

- While a thread executes, it goes through a number of different phases
  - **New**: created but not yet started
  - **Runnable**: is running, or can run on a free CPU
  - **Blocked**: waiting for I/O or on a lock
  - **Sleeping**: paused for a user-specified interval
  - **Terminated**: completed

Which Thread to Run Next?

- Look at all runnable threads
  - A good choice to run is one that just became unblocked because
    - A lock was released
    - I/O became available
    - It finished sleeping, etc.
- Pick a thread and start running it
  - Can try to influence this with setPriority(int)
  - Higher-priority threads get preference
  - But you probably don't need to do this

Some Thread Methods

- void interrupt()
  - Interrupts the thread
- void join() throws InterruptedException
  - Waits for a thread to die/finish
- static void yield()
  - Current thread gives up the CPU
- static void sleep(long milliseconds) throws InterruptedException
  - Current thread sleeps for the given time
- static Thread currentThread()
  - Get Thread object for currently executing thread

Example: Threaded, Sync Alarm

```
while (true) {
  System.out.print("Alarm> ");

  // read user input
  String line = b.readLine();
  parseInput(line);

  // wait (in secs) asynchronously
  if (m != null) {
    // start alarm thread
    Thread t = new AlarmThread(m,tm);
    t.start();
    // wait for the thread to complete
    t.join();
  }
}
```


Daemon Threads

- Definition: Threads which run unattended and perform periodic functions, generally associated with system maintenance.
- void setDaemon(boolean on)
  - Marks thread as a daemon thread
  - Must be set before thread started
- By default, thread acquires status of thread that spawned it
- Program execution terminates when no threads running except daemons

Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
    - A running thread can be pre-empted at any time
- Threads can share data
  - In Java, only fields can be shared
  - Need to prevent interference
    - Rule of thumb 1: You must hold a lock when accessing shared data
    - Rule of thumb 2: You must not release a lock until shared data is in a valid state
  - Overuse use of synchronization can create deadlock
    - Rule of thumb: No deadlock if only one lock

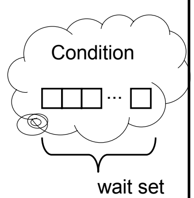
Producer/Consumer Design

- Suppose we are communicating with a shared variable
  - E.g., some kind of a 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 (Java 1.5)

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

- Condition created from a Lock
- await called with lock held
  - Releases the lock
    - But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread
- signalAll called with lock held
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don't need to do it explicitly)



Producer/Consumer Example

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

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

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


Use This Design

- This is the right solution to the problem
  - Tempting to try to just use locks directly
  - Very hard to get right
  - Problems with other approaches often very subtle
    - E.g., double-checked locking is broken

Broken Producer/Consumer Example

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

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

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

Threads wait with lock held – no way to make progress

Broken Producer/Consumer Example

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

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

valueReady accessed without a lock held – race condition

Broken Producer/Consumer Example

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

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

what if there are multiple producers or consumers?

More on the Condition Interface

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

- away(t, u) waits for time t and then gives up
  - Result indicates whether woken by signal or timeout
- signal() wakes up only *one* waiting thread
  - Tricky to use correctly
    - Have all waiters be equal, handle exceptions correctly
  - Highly recommended to just use signalAll()

Await and SignalAll Gotcha's

- await *must* be in a loop
  - Don't assume that when wait returns conditions are met
- Avoid holding other locks when waiting
  - await only gives up locks on the object you wait on

Blocking Queues in Java 1.5

- Interface for producer/consumer pattern

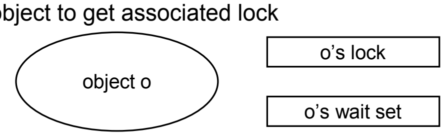
```
interface Queue<E> extends Collection<E> {
 boolean offer(E x);  /* produce */
   /* waits for queue to have capacity */

 E remove();          /* consume */
   /* waits for queue to become non-empty */
 ... }
```

- Two handy implementations
  - LinkedBlockingQueue (FIFO, may be bounded)
  - ArrayBlockingQueue (FIFO, bounded)
  - (plus a couple more)

Wait and NotifyAll (Java 1.4)

- Recall that in Java 1.4, use synchronize on object to get associated lock



- Objects also have an associated wait set

Wait and NotifyAll (cont'd)

- o.wait()
  - Must hold lock associated with o
  - Release that lock
    - And no other locks
  - Adds this thread to wait set for lock
  - Blocks the thread
- o.notifyAll()
  - Must hold lock associated with o
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don't need to do it explicitly)

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;
  }
}
```


Thread Cancellation

- Example scenarios: want to cancel thread
  - Whose processing the user no longer needs (i.e., she has hit the "cancel" button)
  - That computes a partial result and other threads have encountered errors, … etc.
- Java used to have Thread.kill()
  - But it and Thread.stop() are deprecated
  - Use Thread.interrupt() instead

Thread.interrupt()

- Tries to wake up a thread
  - Sets the thread's interrupted flag
  - Flag can be tested by calling
    - interrupted() method
      - Clears the interrupt flag
    - isInterrupted() method
      - Does not clear the interrupt flag
- Won't disturb the thread if it is working
  - Not asynchronous!

Cancellation Example

```
public class CancellableReader extends Thread {
  private FileInputStream dataFile;
  public void run() {
    try {
      while (!Thread.interrupted()) {
        try {
          int c = dataFile.read();
          if (c == -1) break;
          else process(c);
        } catch (IOException ex) { break; }
      }
    } finally { // cleanup here }
  }
}
```

This could acquire locks, be on a wait set, etc.

What if the thread is blocked on a lock or wait set, or sleeping when interrupted?

InterruptedException

- Exception thrown if interrupted on certain ops
  - wait, await, sleep, join, and lockInterruptibly
  - Also thrown if call one of these with interrupt flag set
- *Not thrown* when blocked on 1.4 lock or I/O

```
class Object {
  void wait() throws IE;
  ... }
interface Lock {
  void lock();
  void lockInterruptibly() throws IE;
  ... }
interface Condition {
  void await() throws IE;
  void signalAll();
  ... }
```

Responses to Interruption

- Early Return
  - Clean up and exit without producing errors
  - May require rollback or recovery
  - Callers can poll cancellation status to find out why an action was not carried out
- Continuation (i.e., ignore interruption)
  - When it is too dangerous to stop
  - When partial actions cannot be backed out
  - When it doesn't matter

Responses to Interruption (cont'd)

- Re-throw InterruptedException
  - When callers must be alerted on method return
- Throw a general failure exception
  - When interruption is a reason method may fail
- In general
  - Must reset invariants before cancelling
  - E.g., close file descriptors, notify other waiters, etc.

Handling InterruptedException

```
synchronized (this) {
  while (!ready) {
    try { wait(); }
    catch (InterruptedException e) {
      // make shared state acceptable
      notifyAll();
      // cancel processing
      return;
    }
    // do whatever
  }
}
```

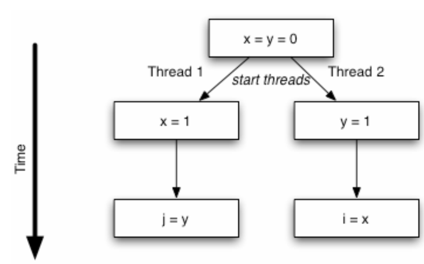

Why No Thread.kill()?

- What if the thread is holding a lock when it is killed? The system could
  - Free the lock, but the data structure it is protecting might be now inconsistent
  - Keep the lock, but this could lead to deadlock
- A thread needs to perform its own cleanup
  - Use InterruptedException and isInterrupted() to discover when it should cancel

Aspects of Synchronization

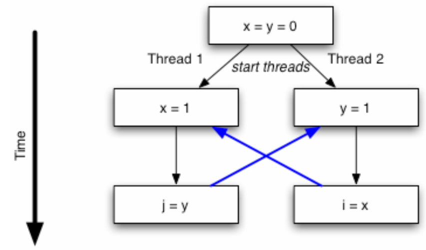
- Atomicity
  - Locking to obtain mutual exclusion
  - What we most often think about
- Visibility
  - Ensuring that changes to object fields made in one thread are seen in other threads
- Ordering
  - Ensuring that you aren't surprised by the order in which statements are executed

Quiz



- Can this result in i=0 and j=0?

Doesn't Seem Possible...

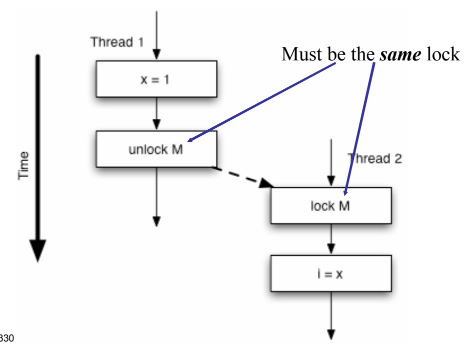


- But this can happen!

How Can This Happen?

- Compiler can reorder statements
  – Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized in global memory

When Are Actions Visible?



Forcing Visibility of Actions

- All writes from thread that holds lock M are visible to next thread that acquires lock M
  – Must be the same lock
- Use synchronization to enforce **visibility** and **ordering**
  – As well as mutual exclusion

Volatile Fields

- If you are going to access a shared field without using synchronization
  – It needs to be volatile
- If you don't try to be too clever
  – Declaring it volatile just works
- Example uses
  – A one-writer/many-reader value
    - Simple control flags:
      – volatile boolean done = false;
  – Keeping track of a "recent value" of something

Misusing Volatile

- Incrementing a volatile field doesn't work
  – In general, writes to a volatile field that depend on the previous value of that field don't work
- A volatile reference to an object isn't the same as having the fields of that object be volatile
  – No way to make elements of an array volatile
- Can't keep two volatile fields in sync
- Don't use for this course

Guidelines for Programming w/Threads

- Synchronize access to shared data
- Don't hold multiple locks at a time
  - Could cause deadlock
- Hold a lock for as little time as possible
  - Reduces blocking waiting for locks
- While holding a lock, don't call a method you don't understand
  - E.g., a method provided by someone else, especially if you can't be sure what it locks
  - Corollary: document which locks a method acquires