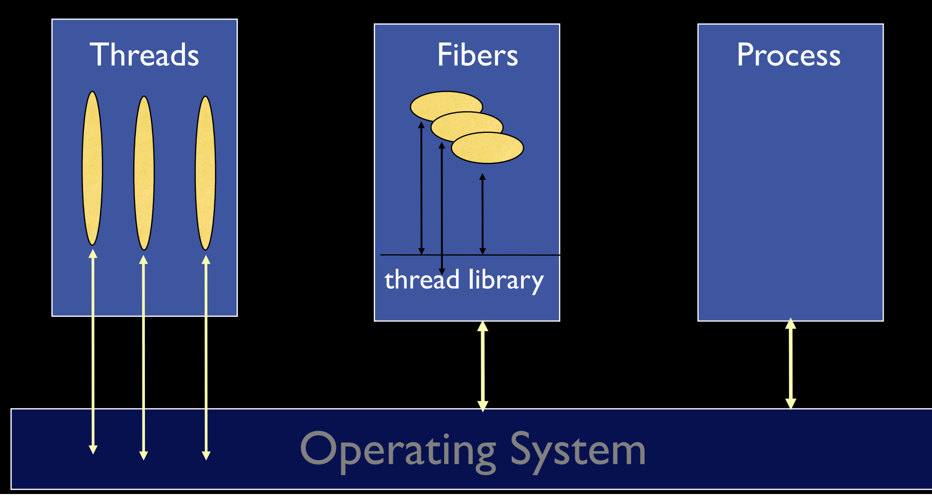
Concurrent Programming in Java

Concurrency Models I

- Heavyweight tasks execute in their own address space
- Lightweight tasks run in the same address space
- A task is disjoint if it does not communicate with or affect the execution of any other task



Concurrency Models II

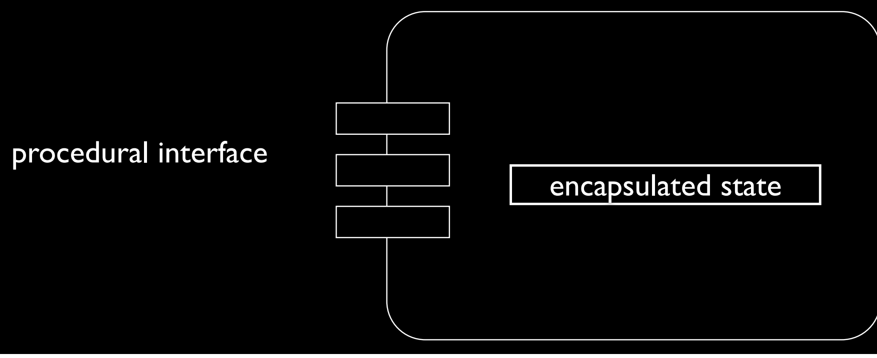
- Java supports threads
  - Threads execute within a single JVM
  - **Native threads** map a single Java thread to an OS thread
  - **Green threads** adopt the thread library approach
  - **M-on-N** threads are a mixture of the above (M green threads scheduled on N native threads)
  - On a multiprocessor, native threads are needed to get true parallelism

Concurrency Models III

- There are various ways in which concurrency can be introduced
  - API for explicit thread creation or thread forking
  - high-level language construct, e.g. PAR (occam), tasks (Ada), or processes (Modula)
- Integration with object-oriented programming, various models:
  - asynchronous method calls
  - early return from methods
  - futures
  - active objects
- Java adopts the active object approach

Concurrency Models IV

- Communication and Synchronization
  - approaches broadly classified as shared-variable or message passing
  - many different models, a popular one is a monitor
  - a monitor can be considered as an object where each of its operation executes in **mutual exclusion**



Concurrency Models V

- **Condition Synchronization**
  - expresses a constraint on the ordering of execution of operations, e.g., data cannot be taken from a buffer until data has been put in it
- Monitors provide condition variables with three operations which can be called when the lock is held
  - `wait`: an unconditional suspension of the calling thread (the thread is placed on a queue associated with the condition variable)
  - `notify`: one thread is taken from the queue and re-scheduled for execution (it must reclaim the lock first)
  - `notifyAll`: all suspended threads are re-scheduled
  - `notify` and `notifyAll` have no effect if no threads are suspended on the condition variable

Kinds of Synchronization

- Cooperative
  - Task A must wait for task B to complete some activity before A can continue executing, e.g., producer-consumer
  - `wait/notify`
- Competitive
  - Two or more tasks must use a resource that cannot be simultaneously accessed, e.g., a shared counter
  - `synchronized`

Liveness and Deadlock

- **Liveness** is a characteristic that a program may or may not have
  - In sequential code, it means the program will eventually complete
  - In a concurrent environment, a task can easily lose its liveness
  - If all tasks lose their liveness, it is called **deadlock**

Concurrency in the real world...

- Concurrency is a also a source of problems
  - Windows 2000: Concurrency errors most common defects among detectable errors
  - Windows 2000: Incorrect synchronization and protocol errors most common coding errors
  - Windows 2003: Synchronization errors second only to buffer overruns
  - Race conditions create security vulnerabilities
  - Concurrent programs are hard to test because they are non-deterministic
    - *Bugs are hard to reproduce because there are exponentially many possible interleavings that often only manifest on deployed*

Data Races

- A *race condition* occurs if two threads access a shared variable at the same time and at least one of the accesses is a write
- Consider the following program when multiple threads are call **`deposit()`** in parallel:

```
class Account {
  private int bal;

  void deposit(int n) {
     int j = bal;
     bal = j + n;
  }
}
```

Lock-based Synchronization

- Monitors must be used to protect every shared location access

```
a = x.a; b = x.b;   ||   x.a = 1; x.b = 2;
```

- Locks must be held before every read/write to a shared location

```
  synchronized(x) {a = x.a; b = x.b;}
||
  synchronized(x) {x.a
```

- When multiple locks are used, a lock acquisition protocol must be adhered to by the application to avoid deadlocks

```
   synchronized(x){ synchronized(y) {...}}
||
   synchronized(y){ synchronized(x) {...}}
```


Puzzlers #1

- How many different values can there be for local variables a and b?

```
a = x.a; b = x.b;   ||    x.a = 1; x.b = 2;
```

Puzzlers #2

- Do the following programs have data races? Are they non-deterministic?

```
long a = x.a;        ||   long b = x.a;
```

```
x.i=1;int i=x.i;     ||   x.i=1;int i=x.i;
```

```
             x.a = MAX_LONG

x.a = 0          ||  long a = x.a;  x.a = (a==MAX_LONG)? 0 : a;
```


Puzzlers #3

- Does the following program have data race?

```
interface INC { void inc(); }

class Int implements INC { int i;  void inc() { i++; } }

class SyncInt implements INC {
   INC val;
   SyncInt(INC v){ val = v; }
   synchronized inc() { val.inc(); }
}

...
INC i = new Int();
INC[] arr=new INC[]{new SyncInt(i),new SyncInt(i),new SyncInt(i)};


arr[0].inc();   ||   arr[1].inc();   ||   arr[2].inc();
```

Puzzlers #3

- Does the following program have concurrency problem?

```
class LL {
     int i;
     LL next;
     LL(LL n,int v){next=n;i=v;}
     synchronized swap() {
        synchronized(next) { int t=next.i; next.i= i; i=t; }
     }
}
```

```
LL a=new LL(null,0), b=new LL(a,1); a.next=b;

      a.swap();   ||  b.swap();
```


ECS 2009

Puzzlers #3

- A fix?

```
class LL {
     static int K;
     private final int id = K++;
     int i;
     LL next;
     LL(LL n,int v){next=n;i=v;}
     private void _swap() {int t=next.i; next.i= i; i=t;}
     void swap() {
        if (this.id > next.id)
          synchronized (this) {synchronized(next) { _swap(); }
        else
          synchronized (next) {synchronized(this) { _swap(); }
     }
}
```

Thursday, August 27, 2009

Puzzlers #3

- A fix?

```
class LL {
    static int K;
    private final int id;
    int i;LL next;

    LL(LL n,int v){
       next=n;i=v;
       synchronized (LL.class) { id = K++; }
    }

    private void _swap() {int t=next.i; next.i= i; i=t;}
    void swap() {
       if (this.id > next.id)
         synchronized (this) {synchronized(next) { _swap(); }
       else
         synchronized (next) {synchronized(this) { _swap(); }
    }

}
```


Puzzlers #4

- The following idioms occurs frequently in library code. Why is it?

```
final class SyncTree {

   Node left, right;

   private final Object lock = new Object();

   public balance() {

       synchronized (lock) {

           ....
```

Puzzlers #5

- Is this a correct solution to synchronization problems?

```
class Big {
    static final Object Lock = new Object();
}


class LL {
     int i;
     LL next;
     LL(LL n,int v){next=n;i=v;}
     void swap() {
        synchronized(Big.Lock) { int t=next.i; next.i= i; i=t; }
     }
 }



    LL a=new LL(null,0), b=new LL(a,1); a.next=b;

          a.swap();    ||  b.swap();
```


ECS 2009

Puzzlers #6

- Does the following program have a data race? A concurrency problem?

```
 class Big { static final Object Lock = new Object(); }


class LL {
     int i;  LL next; LL(LL n,int v){next=n;i=v;}


     void swap() {
       int t = 0;
       synchronized(Big.Lock) { t=next.i; }
       synchronized(Big.Lock) { next.i= i; }
       synchronized(Big.Lock) { i=t; }
     }
 }
```

Thursday, August 27, 2009

Lessons

- To reason about concurrency one must understand all interleaving of operations performed by each thread
- Not all high-level commands are atomic
- Aliasing makes it hard to determine which values are shared, thus one may fail to acquire the right lock (or locks)
- Lock acquisition protocols must be followed to avoid deadlocks
- Library classes must protect themselves from clients by implementing their own synchronization
- Oversynchronization is always safe but decreases concurrency (possibly to the point of making it meaningless?)
- Definition of data race too low level to catch all errors

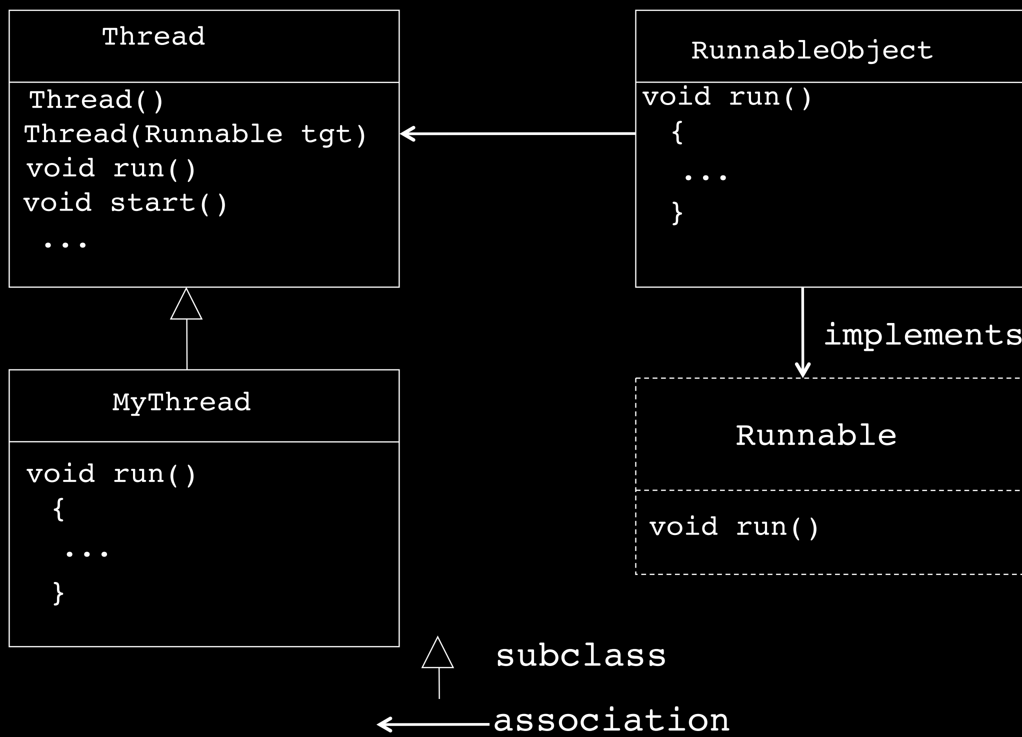
Concurrency in Java

- Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads are created
- However to avoid all threads extending `Thread`, it also has a standard interface

```
public interface Runnable {
  public void run();
}
```

- Hence, a class which wishes to express concurrent execution implements this interface
- Threads do not begin their execution until `start` is called

Threads in Java



ECS 2009

Communication in Java

- Via reading and writing to data encapsulated in shared objects protected by monitors
- Every object is implicitly derived from the `Object` class which defines a mutual exclusion lock
- Methods in a class can be labeled as **`synchronized`**, this means that they can only be executed if the lock can be acquired (this happens automatically)
- The lock can also be acquired via a **synchronized statement** which names the object
- A thread can **`wait`** and **`notify`** on a single anonymous condition variable

Thursday, August 27, 2009

The Thread Class

```
public class Thread implements Runnable {
  public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target, String name);
  public Thread(Runnable target, String name,
                long stackSize);
  public void run();
  public void start();
  ...
```


ECS 2009

Thread Identification

- The identity of the currently running thread can be found using the `currentThread` method
- This has a static modifier, which means the method can always be called using the `Thread` class

```
public class Thread implements Runnable {
  ...
  public static Thread currentThread();
```

Thursday, August 27, 2009

A Thread Terminates:

- when it completes execution of its `run` method either normally or as the result of an unhandled exception
- via a call to its `stop` method — the `run` method is stopped and the thread class cleans up before terminating the thread (releases locks and executes any finally clauses)
  - the thread object is now eligible for garbage collection.
  - `stop` is inherently unsafe as it releases locks on objects and can leave data in inconsistent states; (deprecated; should not be used)
- via a call to its `destroy` method — `destroy` terminates the thread without any cleanup (deprecated)

ECS 2009

Daemon Threads

- Threads can be of two types: user threads or daemon threads
- Daemon threads provide general services and never terminate
- When all user threads have terminated, daemon threads will be terminated by the virtual machine on shutdown
- The `setDaemon` method must be called before calling `start`

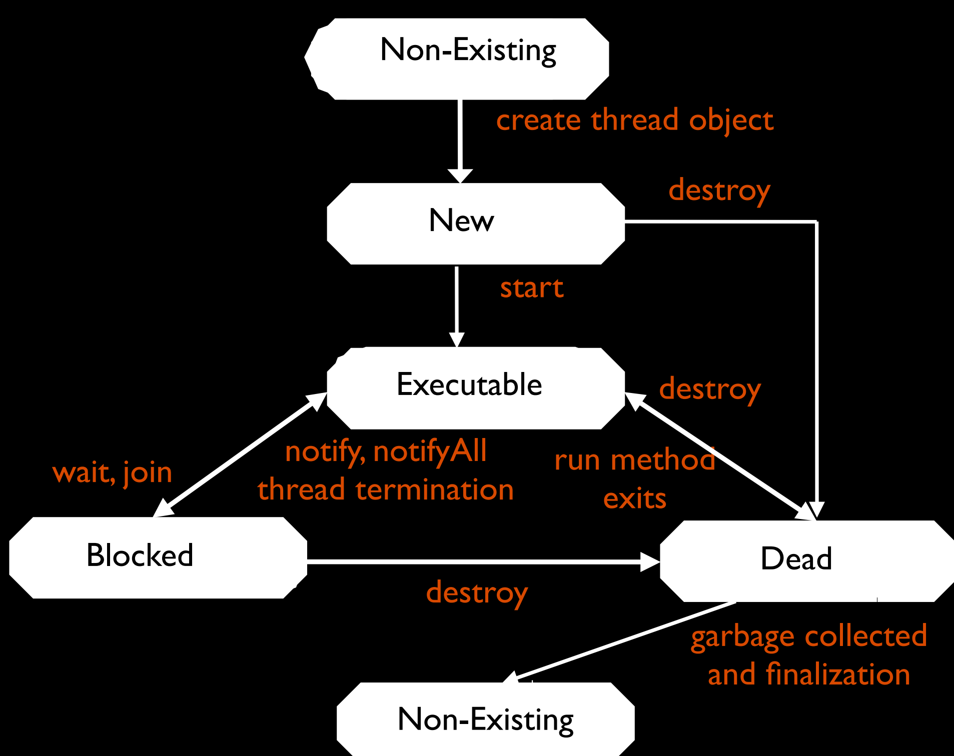
```
public class Thread implements Runnable {
  public void destroy();        // DEPRECATED
  public final boolean isDaemon();
  public final void setDaemon();
  public final void stop();   // DEPRECATED
```

Joining

- One thread can wait (with or without a timeout) for another thread (the target) to terminate by issuing the `join` method call on the target's thread object
- The `isAlive` method allows a thread to determine if the target thread has terminated

```
public class Thread implements Runnable {
  public final native boolean isAlive();
  public final void join() throws InterruptedException;
  public final void join(long ms) throws InterruptedExcept
  public final void join(long millis, int nanos) throws In
```


Java Thread States



Summary

- A thread is created when a `Thread` object is created
- At this point, the thread is not executable, it is in the `new` state
- Once `start` has been called, the thread is eligible for execution
- If the thread calls `wait` on an Object, or calls `join` on another thread object, the thread becomes blocked and no longer eligible
- It becomes executable if an associated `notify` is called by another thread, or if the target thread of the `join` is dead
- it enters the dead state, if the `run` method exits (normally or unhandled exception) or because `destroy` has been called. In the latter case, the thread will not execute any `finally` clauses; it may leave other objects locked

Synchronized Methods

- A mutual exclusion lock is associated with each object.
  It can't be accessed directly by the application but is affected by
  - the method modifier synchronized
  - block synchronization
- When a method is labeled as `synchronized`, the method can only execute once the system has the lock
- Hence, synchronized methods have mutually exclusive access to the data encapsulated by the object, if that data is only accessed by other synchronized methods
- Non-synchronized methods do not require the lock and, therefore, can be called at any time

Example of Synchronized Methods

```
class SharedInteger {
  private int data;
  SharedInteger(int val) { data = val; }
  synchronized int read() { return data;}
  synchronized void write(int val) { data = val; }
  synchronized void incrementBy(int by) { data += by; }
}

SharedInteger shi = new SharedInteger(42);
```


Block Synchronization

- A mechanism where a block can be labeled as synchronized
- The `synchronized` keyword takes as a parameter an object whose lock the system needs to obtain before it can continue
- Synchronized methods are effectively implementable as

```
public int read() {
  synchronized(this) {
      return theData;
  }
}
```

- `this` is the Java mechanism for obtaining the current object

Warning

- In its full generality, **synchronized blocks** can undermine one of the advantages of monitor-like mechanisms, that of encapsulating synchronization constraints associate with an object into a single place in the program
- This is because it isn't possible to understand the synchronization associated with an object by just looking at the object itself when other objects can name that object in a synchronized statement
- However with careful use, this facility allows more expressive synchronization constraints to be programmed

Accessing Synchronized Data

- Consider a simple class which implement a two-dimensional coordinate that is to be shared between two or more threads
- This class encapsulates two integers
- Writing is simple, the `write` method can be synchronized
- The constructor can be assumed not to have any synchronization

```
class Coord {
  Coord(int x,int x) { x_=x; y_=y; }
  synchronized void write(int x,int y) {x_=x; y_=y;}
  private int x_, y_;
}
```

Shared Coordinate

- How to read the value of the coordinates?
- Methods return a single value, parameters are passed by value
- Consequently, it is not possible to have a single read method which returns both the **x** and the **y** values
- If two synchronized functions are used, `readX` and `readY`, the value of the coordinate can be written between calls. The result will be an inconsistent value of the coordinate

Solution I

- Return a new coordinate, which can be accessed freely

```
class Coord {
  synchronized Coord read() { return new Coord(x, y); }
  int readX() {return x;}    int readY() { return y; }
}
```

- The result is only a snapshot of the shared `Coord`, which might be changed by another thread right after the **read** has returned
- The individual field values will be consistent
- Once the coordinate has been used, it can be discarded and made available for garbage collection
- If efficiency is a concern, avoiding unnecessary object creation and garbage collection is appropriate

Solution 2

- Assume the client thread will use synchronized blocks to obtain atomicity

```
class Coord {
   ...
   synchronized void write(int x, int x) { … }

   int readX() { return x; } // not synchronized
   int readY() { return y; } // not synchronized
}
Coord p = new Coord(0,0);


synchronized(p) {
   Coord p2 = new Coord(p.readX(), p.readY());
}
```


Waiting and Notifying

- To obtain conditional synchronization requires the methods provided in the predefined object class
- These methods require the current thread to hold the object lock
- If called without the lock, the unchecked exception **`IllegalMonitorStateException`** is thrown
- The **`wait`** method always blocks the calling thread and releases the lock associated with the object

```
class Object {
  final void notify();
  final void notifyAll();

  final void wait()throws InterruptedException;
  final void wait(long millis) throws InterruptedException;
  final void wait(long millis, int nanos)throws InterruptedE
```

Important Notes

- The **`notify`** method wakes up one waiting thread; the one woken is not defined by the Java language
- **`notify`** does not release the lock; hence the woken thread must wait until it can obtain the lock before proceeding
- To wake up all waiting threads requires use of **`notifyAll`**
- If no thread is waiting, then **`notify`**/**`notifyAll`** are no-ops

Thread Interruption

- A waiting thread can also be awoken if it is interrupted by another thread
- In this case the **`InterruptedException`** is thrown

Condition Variables I

- There are no explicit condition variables in Java
- When a thread is awoken, it cannot assume that its condition is true, as all threads are potentially awoken irrespective of what conditions they were waiting on
- For some algorithms this limitation is not a problem, as the conditions are mutually exclusive,
- E.g., the bounded buffer traditionally has two condition variables: BufferNotFull and BufferNotEmpty
- If a thread is waiting for one condition, no other thread can be waiting for the other condition
- One would expect that the thread can assume that when it wakes, the buffer is in the appropriate state

Condition Variables II

- This is not always the case; Java makes no guarantee that a thread woken from a wait will gain immediate access to lock
- Another thread could call the put method, find that the buffer has space and inserted data into the buffer
- When the woken thread eventually gains access to the lock, the buffer will again be full
- Hence, it is usual for threads to re-evaluate their guards

Bounded Buffer

```
class BoundedBuffer  {
  private final int buffer[];
  private int first, last, numberInBuffer;
  private final int size;
  BoundedBuffer(int len){ buffer = new int[size = len]; }

  synchronized void put(int i)throws InterruptedException{
      while (numberInBuffer == size) wait();
      numberInBuffer++;
      buffer[last = (last+1)%size] = i;
      notifyAll();
  }

  synchronized int get()throws InterruptedException {
      while (numberInBuffer == 0) wait();
      numberInBuffer--;
      notifyAll();
      return buffer[first = (first+1)%size];
  }
}
```


Class Exercise

- How would you implement a semaphore using Java?

Summary I

- Errors in communication and synchronization cause working programs to suddenly suffer from deadlock or livelock
- The Java model revolves around controlled access to shared data using a monitor-like facility
- The monitor is represented as an object with synchronized methods and statements providing mutual exclusion
- Condition synchronization is given by the wait and notify method
- True monitor condition variables are not directly supported by the language and have to be programmed explicitly