COMP2021: Object-Oriented Programming

Concurrency

Dr. Max Yu PEI

Intended Learning Outcomes

- After the lecture, students should be able to
  - analyze the execution of programs with multiple threads,
  - define threads by either extending the Thread class or implementing the Runnable interface,
  - use synchronized methods or statements for mutual exclusive access of shared variables,
  - use wait(), notify(), and notifyAll() to coordinate multiple threads
  - manipulate thread objects through other methods defined in class Thread, and
  - understand the state transition of thread objects.

Concurrent Computing

- Applications designed as a collection of computational units that may execute in parallel
  - Logical vs. physical parallelism
  - Parallel vs. distributed

- What's concurrency good for?
  - Improved user experience
  - Better usage of resources
  - Higher performance

Processes and Threads

- Concurrency can have two levels of granularity, according to what is the unit of parallel computation
  - Processes
    - Abstraction of a running program, includes program counter, registers, variables, …
    - Different processes have <u>independent</u> address spaces
  - Threads
    - An independent thread of execution within a process, or a “lightweight process”
    - Threads within the same process <u>share</u> the address space

Java Threads

- Java's concurrency model is based on threads
- Threads are created by instantiating a Thread object
  - Each instance is associated with a class providing the code associated with the thread
  - Two ways to provide the class code
    - Write a class that inherits from class Thread
    - Write a class that implements interface Runnable

```
public class Thread extends Object implements Runnable {
  …
  public void run() {⋯}
  public void start() {⋯}
  …
  public static Thread currentThread() {⋯}
  public long getID() {⋯}
  public String getName() {⋯}
  public void setName(String name) {⋯}
  …
}
```

Extending Class Thread

- Override the run method to do the real work

```
public class DumbThread extends Thread {

   private Hero hero;

   public DumbThread(Hero hero) {
      this.hero = hero;  // now the thread shares a Hero object
   }                    // with its creator

   public void run() {
      System.out.println(“The health of hero is " + hero.health);
   }
}
```

Note aHero is shared between the main thread and the new DumbThread

What if we call t.run(), instead of t.start()?

```
Thread t = new DumbThread(aHero); // create a thread
t.start(); // Start thread t (start() calls run())
t.join();  // wait until thread t terminates
System.out.println("The thread has terminated");
```

Implementing interface Runnable

- Any implementation of Runnable must implement method run().

```
public class DumbThread implements Runnable {

   private Hero hero;

   public DumbThread(Hero hero) {
      this.hero = hero;  // now the thread shares a Hero object
   }                     // with its creator

   public void run() {
      System.out.println(“The health of hero is " + hero.health);
   }
}
```

This is more desirable than extending class Thread. Why?

```
Thread t = new Thread(new DumbThread(aHero)); // create a thread
t.start(); // start thread t
t.join();  // wait for thread t to terminate
System.out.println("The thread has terminated");
```

Coordination of Threads

The Need for Coordination

- Threads need to coordinate when accessing the shared memory to avoid <u>race conditions</u>
  - Inconsistent access to shared resources

```
// Shared memory
Hero h = ⋯; // h.health should never be negative

// thread 1
if(h.health > 0){
  h.health--;
}

// thread 2
h.health = 0;
```

What happens if thread 2 executes just after thread 1 has tested the if condition (but before the decrement)?

The Need for Coordination

- Coordination should guarantee mutual exclusion when accessing shared resources
  - a section of code that accesses some shared resource is called a critical region

```
// Shared memory
Hero h = …; // h.health should never be negative

// critical region for the shared resource in thread 1
if(h.health > 0){
  h.health--;
}

// critical region for the shared resource in thread 2
h.health = 0;
```

  - at any given time, no more than one thread should be in the critical region (w.r.t. a specific shared resource)

Coordination Mechanisms for Shared Memory

- Mutex (mutual exclusion object)
  - A mutex is a binary variable that can be locked by at most one thread at a time
    - **lock()**: locks the mutex (or, obtains the lock) if it is not locked yet; otherwise suspends the execution
    - **unlock()** : unlocks the mutex (or, releases the lock) so that other **lock()** operations may succeed

```
mutexForResourceA.lock();
// critical region for resource A in thread 1
mutexForResourceA.unlock();
```

```
mutexForResourceA.lock();
// critical region for resource A in thread 2
mutexForResourceA.unlock();
```

    - The **lock** and **unlock** operations are guaranteed to be non-interrupted

A thread may have multiple locks at certain point in time.

Coordination Mechanisms for Shared Memory

- Monitor = Mutex + Condition variable
  - Mutex
  - Condition variable: to coordinate threads protected by the corresponding mutex
    - A condition variable is basically a container of threads that are waiting for a certain condition
    - A thread may not be able to proceed because it needs some other thread's work. Then it can wait and yield control to other threads.
    - When a thread performs an action that some other threads may be waiting for, it can signal it and wake them up (interrupting their waiting)

Java Object and Monitor

- Each object in Java is associated with a monitor
  - monitorenter/monitorexit
    - Bytecode instructions that operate on a Java object
    - Obtains/Releases the lock on the object
  - aObject.wait(): to make the current thread wait on aObject.
    - suspend and unlock all objects until some thread does a notify or notifyAll on aObject
  - aObject.notify(): notify one thread waiting on aObject
    - resume one suspended thread (chosen nondeterministically)
  - aObject.notifyAll(): notify all threads waiting on aObject
    - resume all suspended threads

wait(), notify(), and notifyAll() all operate on the shared aObject, which implies
1. they should only be invoked if the thread has already the lock on aObject;
2. the thread first has to obtain all its previous locks before it can continue with the execution after wait()

Synchronized Blocks

- Synchronized blocks (a.k.a. synchronized statements) support synchronization based on monitor.

```
// hero must be accessed in mutual exclusion
private Hero hero;
public void decreaseHealth() {
  // critical region
  synchronized(hero) {  // hero.monitorenter()
    if (hero.getHealth() > 0) {
      hero.decreaseHealth(1);
    }
  }                     // hero.monitorexit()
}
public void increaseHealth() {
  // critical region
  synchronized(hero) {  // hero.monitorenter()
    hero.increaseHealth(1);
  }                     // hero.monitorexit()
}
```

Synchronized Methods

- Methods can also be synchronized
  - it is as if the method body is a synchronized(this) block

```
// hero must be accessed in mutual exclusion
private Hero hero;
public synchronized void decreaseHealth(){ // this.monitorenter()
  // critical region
  if (hero.getHealth() > 0) {
    hero.decreaseHealth(1);
  }
}                                        // this.monitorexit()
public synchronized void increaseHealth(){ … }
```

  - Equivalent to

```
public void decreaseHealth(){
  synchronized(this){
    if (hero.getHealth() > 0) { hero.decreaseHealth(1); }
  }
}
…
```

Synchronized Static Methods

- Static methods can be synchronized too
  - The monitor associated with the class is used for locking

```
class Test{

  public static synchronized void test(){ // Test.class.monitorenter()
    …
  }                                      // Test.class.monitorexit()

}
```

  - Equivalent to

```
class Test{
  public static void test(){
    synchronized (Test.class){ … }
  }
}
```

The Producer-Consumer Example

- Two kinds of threads, the Producer and the Consumer, work concurrently on a shared Buffer of bounded size

  - A Producer puts new messages in the buffer
    - if the buffer is full, the Producer must wait until the Consumer takes some messages
    - the Producer also signals the last message

  - A Consumer takes messages from the buffer
    - if the buffer is empty, the Consumer must wait until the Producer puts some new messages
    - the Consumer terminates after the last message

```
class Buffer{
    private List<String> storage;
    private int capacity;
    Buffer(int capacity){
        if(capacity <= 0) throw new IllegalArgumentException();
        storage = new LinkedList<>();
        this.capacity = capacity;
    }
    boolean isEmpty(){ return storage.isEmpty(); }
    boolean isFull() { return storage.size() == capacity; }
    String getElement(){
        if(isEmpty()) throw new IllegalStateException();
        return storage.get(0);
    }
    void popElement(){
        if(isEmpty()) throw new IllegalStateException();
        storage.remove(0);
    }
    void addElement(String s){
        if(isFull()) throw new IllegalStateException();
        storage.add(s);
    }
}
```

```java
class Producer extends Thread{
  private Buffer buffer;
  Producer(Buffer buffer){
    this.buffer = buffer;
  }

  public void run(){
    int i = 20;
    while(i >= 0){
      try {
        synchronized (buffer) {
          if (buffer.isFull())
            buffer.wait();
          buffer.addElement(i+"");
          buffer.notify();
        }
      }
      catch(InterruptedException e){}
      finally { i--; }
    }
  }
}
```

```java
class Consumer extends Thread{
  private Buffer buffer;
  Consumer(Buffer buffer){
    this.buffer = buffer;
  }
  public void run(){
    String ele = "";
    while(!ele.equals("0")){
      try {
        synchronized (buffer) {
          if (buffer.isEmpty())
            buffer.wait();
          ele = buffer.getElement();
          buffer.popElement();
          buffer.notify();
          System.out.println(ele);
        }
      }
      catch(InterruptedException e){}
    }
  }
}
```

Running The Threads

- One Producer + One Consumer

```
Buffer buffer = new Buffer(10);
Thread producer1 = new Producer(buffer);
Thread consumer1 = new Consumer(buffer);
producer1.start(); consumer1.start();
producer1.join();  consumer1.join();
```

- One Producer + Two Consumers

```
Buffer buffer = new Buffer(10);
Thread producer1 = new Producer(buffer);
Thread consumer1 = new Consumer(buffer);
Thread consumer2 = new Consumer(buffer);
producer1.start(); consumer1.start(); consumer2.start();
producer1.join();  consumer1.join();  consumer2.join();
```

An IllegalStateException may be thrown from Buffer.getElement().
Because the call to notify() by a Consumer may wake up the other Consumer, instead of the Producer.

Multiple Conditions

- What if we want to have two kinds of Consumers, one to consume Strings of even integers and the other those of odd integers?
  - Do we need to have two condition variables, one for each type of Strings?
    - Complicated and not scalable!
  - How to achieve the same effect using only one condition variable?
    - Use a single condition variable for a group of conditions
    - Call notifyAll() instead of notify()
    - Check the condition again after waking up from each wait()

!! This is how wait(), notify(), and notifyAll() are typically used !!

```java
class Producer extends Thread{
  private Buffer buffer;
  Producer(Buffer buffer){
    this.buffer = buffer;
  }

  public void run(){
    int i = 20;
    while(i >= 0){
      try {
        synchronized (buffer) {
          while (buffer.isFull())
            buffer.wait();
          buffer.addElement(i+"");
          buffer.notifyAll();
        }
      }
      catch(InterruptedException e){}
      finally { i--; }
    }
  }
}
```

```java
class Consumer extends Thread{
  private Buffer buffer;
  Consumer(Buffer buffer){
    this.buffer = buffer;
  }
  public void run(){
    String ele = "";
    while(!ele.equals("0")
         && !ele.equals("1")){
      try {
        synchronized (buffer) {
          while (buffer.isEmpty())
            buffer.wait();
          ele = buffer.getElement();
          buffer.notifyAll();
          buffer.popElement();
          System.out.println(ele);
        }
      }
      catch(InterruptedException e){}
    }
  }
}
```

One Producer + Two Consumers

A Better Design: Thread-Safe Buffer

```
class Buffer{
    private List<String> storage;   private int capacity;
    Buffer(int capacity){ … }
    synchronized boolean isEmpty(){ return storage.isEmpty(); }
    synchronized boolean isFull(){ return storage.size() == capacity; }
    synchronized String getElement() throws InterruptedException{
        while (isEmpty()) wait();
        return storage.get(0);
    }
    synchronized void popElement() throws InterruptedException{
        while (isEmpty()) wait();
        storage.remove(0);
        notifyAll();
    }
    synchronized void addElement(String s) throws InterruptedException{
        while (isFull()) wait();
        storage.add(s);
        notifyAll();
    }
}
```

One Producer + Two Consumers

```
Buffer buffer = new Buffer(10);
Thread producer1 = new Producer(buffer);
Thread consumer1 = new EvenConsumer(buffer);
Thread consumer2 = new OddConsumer(buffer);
producer1.start(); consumer1.start(); consumer2.start();
producer1.join();  consumer1.join();  consumer2.join();
```

```
class Producer extends Thread{
  private Buffer buffer;
  Producer(Buffer buffer){ this.buffer = buffer; }
  public void run(){
    int i = 20;
    while(i >= 0){
      try { buffer.addElement(i+""); }
      catch (InterruptedException e){}
      finally { i--; }
    }
  }
}
```

```
abstract class Consumer extends Thread{
 private Buffer buffer;
 Consumer(Buffer buffer){ this.buffer = buffer; }
 public void run(){
  boolean shouldStop = false;
  while(!shouldStop){
   try {
    synchronized (buffer) {
     String ele = buffer.getElement();
     if(shouldConsume(ele)){
      buffer.popElement();
      System.out.println(ele);
      shouldStop = shouldStop(ele);
     }
    }
   }
   catch(InterruptedException e){}
  }
 }
 public abstract boolean shouldConsume(String element);
 public abstract boolean shouldStop(String element);
}
```

```
class EvenConsumer extends Consumer{
    EvenConsumer(Buffer buffer){ super(buffer); }
    public boolean shouldConsume(String element){
        return element != null && Integer.valueOf(element) % 2 == 0;
    }
    public boolean shouldStop(String element){
        return element.equals("0");
    }
}

class OddConsumer extends Consumer{
    OddConsumer(Buffer buffer){ super(buffer); }
    public boolean shouldConsume(String element){
        return element != null && Integer.valueOf(element) % 2 == 1;
    }
    public boolean shouldStop(String element){
        return element.equals("1");
    }
}
```

Limitations of Monitors

- Lack of concurrency
  - Synchronized block/statement
- Deadlock occurs when two or more threads block each other, and none can make progress

```
class Test extends Thread{
  Object m1, m2;
  public Test(Object m1, Object m2){
    this.m1 = m1; this.m2 = m2;
  }
  public void run(){
    synchronized(m1){
      synchronized(m2){
        ...
      }
    }
  }
}
```

```
Object m1 = new Object();
Object m2 = new Object();
Test t1 = new Test(m1, m2);
Test t2 = new Test(m2, m1);
t1.start();
t2.start();
t1.join();
t2.join();
```

Be careful with the order in which you acquire locks.

Putting a Thread to Sleep

- The sleep(int t) static method suspends the thread in which it is invoked for t milliseconds, or until an interrupt is received

```
Thread.sleep(2000); // suspend the thread currently
                    // running for 2 seconds
```

  - sleep throws an InterruptedException if an interrupt occurs
  - even if no interrupt occurs, the timing may be more or less precise according to the real-time guarantees of the running JVM

Thread Termination

- A thread terminates
  - when it completes the execution of its run method either normally or as the result of an unhandled exception
  - upon a call to its `stop` method (deprecated)
    - the `run` method is stopped, and the thread class cleans up before terminating the thread
    - directly releasing the locks may leave objects in inconsistent states
    - the thread object is now eligible for garbage collection.
  - if its `destroy` method is called (deprecated)
    - terminates the thread without any cleanup
    - not releasing the locks may cause deadlocks in the future

Cancelling a Thread: Interrupt

- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else.
  - It is often used to terminate a thread.
    - If a thread t1 is sleeping or waiting when t1.interrupt() is called, then the sleeping or waiting immediately returns with an exception
    - If thread t1 is doing other operations, calling t1.interrupt() will set an internal flag of t1 to true; t1.interrupted() can be used to check the value of that flag.

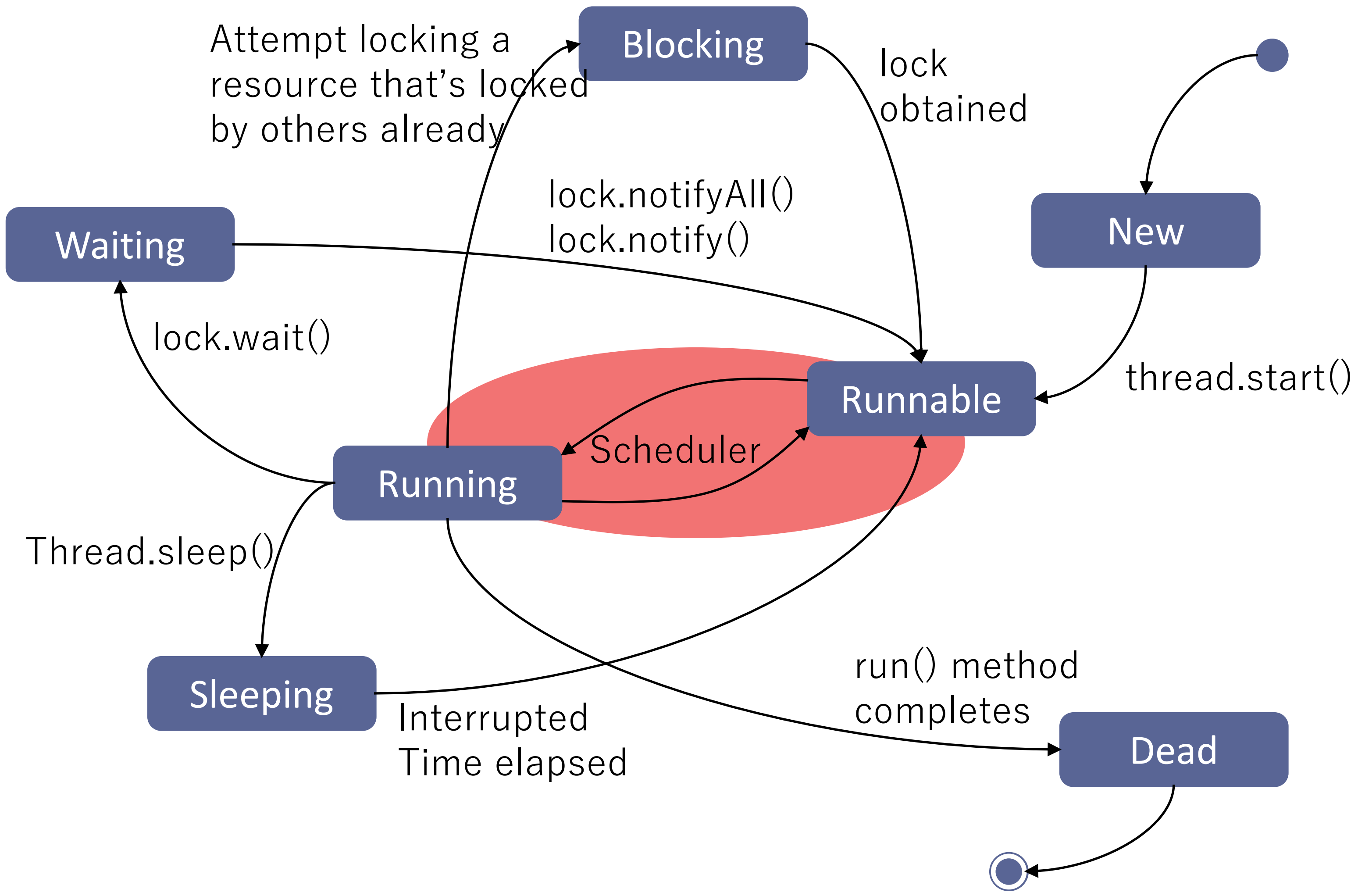
```
// in your controller thread
if(shouldCancel()){ workerThread.interrupt(); }
```

```
// in your worker thread
```

```
while(!interrupted()){
  // do some work
}
```

```
try{ … if(…) o.wait(); … }
catch(InterruptedException e){ … }
```

Thread States



User Threads and Daemon Threads

- Daemon threads provide general services and typically never terminate
  - When all user threads have terminated, daemon threads can also be terminated, and the main program terminates
    - JVM does not wait for daemon threads to terminate
      - finally blocks are not executed
      - stacks are not unwound - the JVM just exits.
  - Creation
    - When a new thread is created it inherits the daemon status of its parent.
    - Method `setDaemon()` must be called, if necessary, before the thread is started

Summary

- Threads and Processes
- Thread and Runnable
- Synchronized Methods and Statements
- wait(), notify(), and notifyAll()
- Thread manipulation
- State transition of threads

End