CDP 2013

Threads in Java
Threads: reminder

- Basic sequential unit of execution.

- Multiple threads run in the context of single process:
  - all threads within a process share common memory space (but not their stacks)
  - access to common memory space requires synchronization.
Threads: reminder (cont.)

• Threads are scheduled and executed independently

• On single-CPU systems:
  – threads are multiplexed on the same processor (pseudo-parallel execution)
  – good for I/O stalls etc.

• On multi-CPU (multi-core) systems:
  – threads are truly run in parallel
Threads in Java

- JVM allows to create multiple Java threads
  - note: standard JVM may run only a single Java process, but any number of threads

- JVM internally implements the mapping of Java threads onto real OS threads
  - Implementation is platform-dependent (and thread behavior)
  - You cannot assume anything about thread scheduling
Question

- Thread A { while (1) print “A” }
- Thread B { while (1) print “B” }
- We invoke them simultaneously.
- What is the output?
Threads in Java (cont.)

- Thread is a runnable object.
  - Regular object
    - Instance of a class
    - Has various methods and member variables
    - Should be instantiated with constructor
  - Has a method which is invoked in a separate JVM thread (note: not invoked automatically upon object instantiation)
Writing threads
(Extending Thread class)

class WriteThread extends Thread {
    char letter;
    boolean stop = false;
    WriteThread(char letter) {
        this.letter = letter;
    }
    public void halt() { stop = true; }
    @Override public void run() {
        while (!stop) { System.out.print(letter); }
    }
}

WriteThread a = new WriteThread('a');
WriteThread b = new WriteThread('b');
a.start(); b.start();
// do some other stuff
a.halt(); b.halt(); // are both guaranteed to be stopped?
a.start(); // ERROR – Illegal thread state exception: can be 'start'-ed only once
class WriteObject implements Runnable {
    char letter;
    boolean stop = false;
    WriteObject(char letter) {
        this.letter = letter;
    }
    public void halt() { stop = true; }
    @Override public void run() {
        while (!stop) {
            System.out.print(letter);
            Thread.yield();
        }
    }
}

WriteObject a = new WriteObject('a');
WriteObject b = new WriteObject('b');

new Thread(a).start(); new Thread(b).start();

.... // do some other stuff
a.halt(); b.halt();
So, which way is better?

- Implementing Runnable:
  - Does not occupy the single extension slot.
  - Allows running the same object in multiple threads.
  - Separates executed class logic from Thread class logic – cleaner design.

- Extending Thread:
  - Slightly less code...

- A matter of design.

- No performance difference.
A note about Runnable.run()

• It is a regular Java method, which is set as the thread's starting point ("main").
• Needed because you can't pass pointers to functions in Java (unless using reflection).
• Calling run() directly executes it in the context of the calling thread.
• Thread.start goes to the JVM, creates a new thread, and passes the Runnable to be executed. The new thread executes the given Runnable's run().
Thread properties

- Constructor called

- Name
- ID
- Priority
- State
- isDaemon

The run method terminates
public class Counter extends Thread {
    public int count = 0;
    public void increase() {
        count++;
    }
    @Override public void run() {
        for (int i = 0; i < 20; i++) increase();
    }
}

main()....
Counter c = new Counter();
c.start();
for (int i = 0; i < 20; i++) c.increase();
c.join();
print("Result is " + c.count);
And the output is...

- Anything between 2 and 40.
- The ++ operator is not atomic, and is equivalent to:
  - Read $count$ into $temp$.
  - Add 1 to $temp$.
  - Write $temp$ to $count$.
  - Interleaving can happen at each stage.
- So how would you get 2?
  - T1 reads $count$ (0) and is then preempted.
  - T2 does 19 increments.
  - T1 writes 1 to $count$.
  - T2 reads $count$ (1).
  - T1 does 19 increments and ends.
  - T2 writes 2 to $count$ and ends.
Synchronization in Java

• Critical sections – code segments allowing access to shared data from multiple threads

• Access to shared data requires mutual exclusion to ensure atomic operations

• Basic synchronization mechanism in Java is monitor.
  – java.util.concurrent library provides many other synchronization primitives
Monitors

- Each object in Java has an associated monitor (coming from Object – the root of all classes).
- The monitor protects the object's state, namely its fields.
- Lightweight mutex – does not cross process boundaries.
- Full blown synchronization object usually created upon contention. Till then, possibly just a few flags (JVM implementation dependent).
public class Counter extends Thread {
    public int count = 0;
    public void increase() {
        synchronized(this) {
            count++;
        }
    }
    ...
What exactly happens here?

Object C

Try to acquire

Failure: Thread blocked

Unblocked but not scheduled. **Fairness not ensured!!**
What is actually synchronized?

- All accesses to the same monitor.
- All *synchronized* methods of an object are mutually excluded.
- Why? Because *we synchronize data access, and not code execution*!
- You want to protect the object's state, and it could be accessed by any method.
- If different methods access different fields object level synchronization might be too coarse – use the fields themselves as synchronization object.
Synchronizing static methods?

```java
public class Counter extends Thread {
    private static Object synchronizer = new Object();
    private static int count = 0;
    public static void increase() {
        synchronized (synchronizer) {
            count++;
        }
    }
}
```

```java
public class Counter extends Thread {
    public int getCount() {
        return counter;
    }
    public int setCount(int val) {
        counter = val;
    }
    public static int synchGetCounter(Counter c) {
        synchronized (c) {
            return c.getCount();
        }
    }
    public static void synchSetCounter(Counter c, int k) {
        synchronized (c) {
            c.setCount(k);
        }
    }
}
```
public class Counter extends Thread {
    int count = 0;
    public synchronized void setCounter(int k) {...}
    public synchronized int testAndSet(int k) {
        int old = count;
        if (count > k) {
            setCounter(k);
        }
        return old;
    }
    ...
}

Is it a deadlock? The lock on this has been already acquired?
Producer-consumer

• Scenario:
  – Producer thread generates objects
    ● E.g. read data from the disk (usually slow) while the consumer processes it
  – Consumer thread processes them
    ● E.g. processes the data while the producer reads more from the disk
  – How to coordinate these threads?
Threads communicate via queue

```java
public class Producer extends Thread {
    Queue newWork;
    public Producer(Queue work) {
        this.newWork = work;
    }
    @Override public void run() {
        for (int i = 0; i < 10; i++) {
            Work w = readFromDisk();
            newWork.enqueue(w);
        }
    }
}

public class Consumer extends Thread{
    Queue toDo;
    public Consumer(Queue toDo) {
        this.toDo = toDo;
    }
    @Override public void run() {
        for (int i = 0; i < 10; i++) {
            Work w = toDo.getNext();
            process(w);
        }
    }
}

main(){
    Queue q = new Queue();
    new Producer(q).start();
    new Consumer(q).start();
}
```
Queue implementation
Try 1

```java
public class Queue {
    Work myWork = null;  // actually set to null by default
    public Work getNext() {
        Work tmp = myWork;
        myWork = null;
        return tmp;
    }
    public void enqueue(Work w) {
        myWork = w;
    }
}
```

- Problems?
Queue implementation
Try 2

public class Queue {
    Work myWork = null;
    public synchronized Work getNext() {
        Work tmp = myWork;
        myWork = null;
        return tmp;
    }
    public synchronized void enqueue(Work w) {
        myWork = w;
    }
}

- Problems?
public class Queue {

    Work myWork = null;
    public synchronized Work getNext() {
        while (myWork == null) {};  
        Work tmp = myWork;
        myWork = null;
        return tmp;
    }

    public synchronized void enqueue(Work w) {
        while (myWork != null) {};  

        myWork = w;
    }
}

• Problems?
public class Queue {
    Work myWork = null;
    public synchronized Work getNext() {
        while (myWork == null) {
           wait();
        }
        Work tmp = myWork;
        myWork = null;
        notify();
        return tmp;
    }
    public synchronized void enqueue(Work w) {
        while (myWork != null) {
            wait();
        }
        myWork = w;
        notify();
    }
}
Waits and notifications

- Notifications are not accumulated – if no thread is waiting, notification goes to waste.
- The notified thread is chosen arbitrarily – no predefined criteria (like time waiting).
- A notified thread will try to acquire the monitor again – other threads might get a hold of it first, even if they weren't waiting.
- Spurious wakeups are possible according to the Java Standard, and will happen on Posix-based systems!
- Waits and notifications – only within synchronized block.
- `wait()` and `notify()` are methods – they affect the monitor of the object they belong to.
Try #4 problem:

- Two consumers and one producer.
- C1 calls getNext() and waits.
- C2 calls getNext() and waits.
- P calls enqueue, and wakes C1.
- P calls enqueue again, and waits.
- C1 gets work, and wakes C2.
- C2 sees myWork == null and waits.
- C1 calls getNext() and waits.
- Now all threads are waiting – deadlock.
- Only way out is if P gets a spurious wakeup...
public class Queue {
    Work myWork = null;
    public synchronized Work getNext() {
        while (myWork == null) {
            wait();
        }
        Work tmp = myWork;
        myWork = null;
        notifyAll();
        return tmp;
    }
    public synchronized void enqueue(Work w) {
        while (myWork != null) {
            wait();
        }
        myWork = w;
        notifyAll();
    }
}

All threads waiting on this object are woken up

Actually, a single notifyAll() is sufficient in this example.