

CS 351

Design of Large Programs

Threads and Concurrency

October 20, 2021

Concurrency in Java

- Java has basic concurrency support built into the language.
- Also has high-level APIs available in `java.util.concurrent` package

Processes vs Threads

- Processes
 - Self-contained execution environment
 - Each process has own memory space
 - Communicate with other processes through interprocess communication (pipes, sockets, files, etc.)
- Threads
 - Creating new thread requires fewer resources than an new process.
 - Threads within same process share process's resources.
 - Threads have shared heap, but separate stacks.

Thread objects

- Each thread is associated with an instance of `Thread`
- Two ways to create a new thread:
 - Subclass `Thread`
 - Implement `Runnable` interface and pass `Runnable` object to `Thread` constructor.
- In both cases, you'll implement `run` method to contain the code to be executed on the thread.
- Implementing `Runnable` is the more flexible approach.

Thread.sleep method

- The sleep method causes the current thread to suspend execution for a specified number of milliseconds.
- Sleep time is not guaranteed to be precise.
(Limits of OS)
- Sleep period may be terminated by an interrupt.

Thread methods

If I have initialized a `Thread` named `myThread`

- `myThread.start()` – starts running `myThread`
- `myThread.join()` – Pauses current thread until `myThread` terminates

Synchronized Methods

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() { c++; }  
    public synchronized int value() { return c; }  
}
```

- It is not possible for two invocations of synchronized methods on the same object to interleave.
- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread is done with the object.

Synchronized Blocks

```
private Point p;
...
public void copyP(Point destination){
    synchronized(p) {
        destination.x = p.x;
        destination.y = p.y;
    }
}

public void addP()(int n) {
    synchronized(p) {
        p.x += n;
        p.y += n;
    }
}
```

Both synchronized blocks obtain lock on member variable `p`

Synchronized Methods vs Blocks

- Making a method synchronized is equivalent to wrapping method body in a `synchronized(this)` block.
- Synchronized blocks are more complicated, but offer finer grained synchronization than synchronized methods.

Liveness Problems

- Deadlock – Threads are blocked forever waiting for each other.
- Starvation – Thread cannot gain access to shared resources held by other “greedy” threads.
- Livelock – Threads too busy responding to each other to actually make progress.

FibThreads: Worker (1/1)

```
public static class Worker extends Thread {
    private final String name;
    private long step = 0;
    private int x = 0;
    private int y = 1;
    private int z;
    private boolean keepGoing = true;

    public Worker(String name) {
        this.name = name;
        z = x + y;
    }

    private synchronized void update() {
        step++;
        if (z < 0) {
            // restart after overflow
            x = 0;
            y = 1;
        } else {
            x = y;
            y = z;
        }
        z = x + y;
    }
}
```

FibThreads: Worker (2/2)

```
public void quit() {
    keepGoing = false;
}

@Override
public void run() {
    while(keepGoing) {
        update();
    }
    System.out.println(name +
        " stopping at step " + step);
}

@Override
public synchronized String toString() {
    return name + " step " + step +
        ", x = " + x + ", y = " + y + ", z = " + z;
}
}
```

FibThreads: main

```
public static void main(String[] args) throws InterruptedException {  
    Worker[] workers = new Worker[]{ new Worker("A"), new Worker("B") };  
    for(Worker worker : workers) { worker.start(); }  
    for(int i = 0; i < 10; ++i) {  
        System.out.println("i = " + i);  
        for(Worker worker : workers) {  
            System.out.println(worker);  
        }  
        Thread.sleep(1000); // Take a short nap  
    }  
    for(Worker worker : workers) { worker.quit(); }  
    for(Worker worker : workers) {  
        // wait until this thread has finished.  
        worker.join();  
    }  
    System.out.println("All workers are done. Goodbye.");  
}
```

FibThreads: Why synchronized?

- update and toString methods are synchronized
- What might happen if we didn't?

Producer/Consumer Pattern

- Producers and consumers run concurrently.
- Producer produces values and places them in a shared queue.
- Consumer removes values from queue and processes them.
- May be multiple producers, consumers, queues.

Producer/Consumer with BlockingQueue

- Could implement with any queue type and careful use of synchronized blocks, but there is an easier way.
- The `java.util.concurrent.BlockingQueue` interface extends `java.util.Queue` with methods that make current thread wait if necessary.
 - put – add item to queue (wait if no room)
 - take – remove next item from queue (wait if empty)

ProducerConsumer: Producer

```
public static class Producer implements Runnable {
    private final String name;
    private final BlockingQueue<Integer> queue;

    public Producer(String name, BlockingQueue<Integer> queue) {
        this.name = name;
        this.queue = queue;
    }

    @Override
    public void run() {
        System.out.println("Start " + name);
        try {
            for(int i = 0; i < 20; i++) {
                System.out.println(name + " produces " + i);
                queue.put(i);
            }
            System.out.println(name + " is done producing");
        } catch (InterruptedException e) {
            System.out.println(name + " was interrupted");
        }
    }
}
```

ProducerConsumer: Consumer

```
public static class Consumer implements Runnable {
    private final String name;
    private final BlockingQueue<Integer> queue;

    public Consumer(String name, BlockingQueue<Integer> queue) {
        this.name = name;
        this.queue = queue;
    }

    @Override
    public void run() {
        System.out.println("Start " + name);
        try {
            while(true) {
                int value = queue.take();
                System.out.println(name + " consumes " + value);
            }
        } catch (InterruptedException e) {
            System.out.print(name + " was interrupted");
        }
    }
}
```

ProducerConsumer: main

```
public static void main(String[] args)
    throws InterruptedException {

    BlockingQueue<Integer> sharedQueue =
        new LinkedBlockingQueue<>();

    Thread prodThread =
        new Thread(new Producer("P", sharedQueue));
    Thread consThread =
        new Thread(new Consumer("C", sharedQueue));

    prodThread.start();
    consThread.start();
}
```