COMP31212: Concurrency
Practical Java Concurrency

Giles Reger
Outline

The Problem

Running tasks

Blocking Queues

Concurrent structures
Abstraction

- Last time we looked at the Thread and low level synchronization
- This time we consider concurrency at a slightly higher level of abstraction
  - Concurrent data structures
  - Executor services
Data parallelism versus Task parallelism

- Two ways to think about parallelism
  - Task parallelism
    - Split problem into multiple tasks that can run in parallel.
  - Data parallelism
    - Perform the same task on different pieces of data.

- In the real world you will often meet data parallelism, which we discuss here.
Matrix Multiplication

- We should all be familiar with the idea of matrix multiplication
- The idea is to create a task to compute each entry in $AB$

\[
A = \begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1m} \\
A_{21} & A_{22} & \cdots & A_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
A_{n1} & A_{n2} & \cdots & A_{nm}
\end{pmatrix}, \quad B = \begin{pmatrix}
B_{11} & B_{12} & \cdots & B_{1p} \\
B_{21} & B_{22} & \cdots & B_{2p} \\
\vdots & \vdots & \ddots & \vdots \\
B_{m1} & B_{m2} & \cdots & B_{mp}
\end{pmatrix}
\]

\[
AB = \begin{pmatrix}
(AB)_{11} & (AB)_{12} & \cdots & (AB)_{1p} \\
(AB)_{21} & (AB)_{22} & \cdots & (AB)_{2p} \\
\vdots & \vdots & \ddots & \vdots \\
(AB)_{n1} & (AB)_{n2} & \cdots & (AB)_{np}
\end{pmatrix}
\]

\[
(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}
\]
Executor

- We can separate how our tasks are executed from what is executed

- Using `java.util.concurrent.Executor` interface

- Executors manage the execution of runnable tasks

  ```java
  Executor exec = ...;
  exec.execute(runnable);
  ```
The Task

private static class Task implements Runnable {
    private int row, column;
    private int[][] A, B, C;
    private CountDownLatch latch;

    public Task(int r, int col, int[][] a, int[][] b, int[][] c, CountDownLatch lat) {
        row = r; column = col;
        A = a; B = b; C = c;
        latch = lat;
    }

    public void run() {
        for (int k = 0; k < A[row].length; k++) {
            C[row][column] += A[row][k] * B[k][column];
        }
        latch.countDown();
    }
}
private static class Task implements Runnable {
    private int row, column;
    private int[][] A, B, C;
    private CountDownLatch latch;

    public Task(int r, int col, int[][] a, int[][] b, int[][] c, CountDownLatch lat) {
        row = r; column = col;
        A = a; B = b; C = c;
        latch = lat;
    }

    public void run() {
        for (int k = 0; k < A[row].length; k++) {
            C[row][column] += A[row][k] * B[k][column];
        }
        latch.countDown();
    }
}

The Task
private static class Task implements Runnable {
    private int row, column;
    private int[][] A, B, C;
    private CountDownLatch latch;

    public Task(int r, int col, int[][] a, int[][] b, int[][] c, CountDownLatch lat) {
        row = r; column = col;
        A = a; B = b; C = c;
        latch = lat;
    }

    public void run() {
        for (int k = 0; k < A[row].length; k++) {
            C[row][column] += A[row][k] * B[k][column];
        }
        latch.countDown();
    }
}
private static class Task implements Runnable {
    private int row, column;
    private int[][] A, B, C;
    private CountDownLatch latch;

    public Task(int r, int col, int[][] a, int[][] b,
                int[][] c, CountDownLatch lat) {
        row = r; column = col;
        A = a; B = b; C = c;
        latch = lat;
    }

    public void run(){
        for (int k = 0; k < A[row].length; k++) {
            C[row][column] += A[row][k] * B[k][column];
        }
        latch.countDown();
    }
}

The Task
private static Executor exec;
private static BlockingQueue<Runnable> work;
static{
    work = new LinkedBlockingQueue<Runnable>();
    exec = Executors.newFixedThreadPool(4);
}

public static int[][] multiply(int[][] a, int[][] b)
    throws InterruptedException {
    int rows = a.length; int cols = b[0].length;
    int [][] c = new int [rows][cols];

    CountDownLatch latch = new CountDownLatch(rows*cols);
    for(int row=0;row<rows;row++)
        for(int col=0;col<cols;col++)
            exec.execute(new Task(row,col,a,b,c,latch));
    latch.await();
    return c;
}
Creating tasks

private static Executor exec;
private static BlockingQueue<Runnable> work;
static{
    work = new LinkedBlockingQueue<Runnable>();
    exec = Executors.newFixedThreadPool(4);
}

public static int[][] multiply(int[][] a, int[][] b)
    throws InterruptedException {
    int rows = a.length; int cols = b[0].length;
    int[][] c = new int[rows][cols];

    CountDownLatch latch = new CountDownLatch(rows*cols);
    for(int row=0;row<rows;row++)
        for(int col=0;col<cols;col++)
            exec.execute(new Task(row,col,a,b,c,latch));
    latch.await();
    return c;
}
Creating tasks

private static Executor exec;
private static BlockingQueue<Runnable> work;
static{
    work = new LinkedBlockingQueue<Runnable>();
    exec = Executors.newFixedThreadPool(4);
}

public static int[][] multiply(int[][] a, int[][] b)
throws InterruptedException {
    int rows = a.length; int cols = b[0].length;
    int [][] c = new int [rows][cols];

    CountDownLatch latch = new CountDownLatch(rows*cols);
    for(int row=0;row<rows;row++)
        for(int col=0;col<cols;col++)
            exec.execute(new Task(row,col,a,b,c,latch));
    latch.await();
    return c;
}
Creating tasks

private static Executor exec;
private static BlockingQueue<Runnable> work;
static{
    work = new LinkedBlockingQueue<Runnable>();
    exec = Executors.newFixedThreadPool(4);
}

public static int[][] multiply(int[][] a, int[][] b)
    throws InterruptedException {
    int rows = a.length; int cols = b[0].length;
    int [][] c = new int [rows][cols];

    CountDownLatch latch = new CountDownLatch(rows*cols);
    for(int row=0;row<rows;row++)
        for(int col=0;col<cols;col++)
            exec.execute(new Task(row,col,a,b,c,latch));
    latch.await();
    return c;
}
Creating tasks

```java
private static Executor exec;
private static BlockingQueue<Runnable> work;
static{
    work = new LinkedBlockingQueue<Runnable>();
    exec = Executors.newFixedThreadPool(4);
}

public static int[][] multiply(int[][] a, int[][] b)
    throws InterruptedException {
    int rows = a.length; int cols = b[0].length;
    int [][] c = new int [rows][cols];

    CountDownLatch latch = new CountDownLatch(rows*cols);
    for(int row=0;row<rows;row++)
        for(int col=0;col<cols;col++)
            exec.execute(new Task(row,col,a,b,c,latch));
    latch.await();
    return c;
}
```
Creating tasks

private static Executor exec;
private static BlockingQueue<Runnable> work;
static{
    work = new LinkedBlockingQueue<Runnable>();
    exec = Executors.newFixedThreadPool(4);
}

public static int[][] multiply(int[][] a, int[][] b)
        throws InterruptedException {
    int rows = a.length; int cols = b[0].length;
    int[][] c = new int [rows][cols];

    CountDownLatch latch = new CountDownLatch(rows*cols);
    for(int row=0;row<rows;row++)
        for(int col=0;col<cols;col++)
            exec.execute(new Task(row,col,a,b,c,latch));
    latch.await();
    return c;
}
CountDownLatch

- We use CountDownLatch to synchronize all threads
- The idea
  - Set it to \textit{counter}
  - Each thread decrements counter
  - Wait for it to reach zero
- java.util.concurrent contains other low-level synchronization tools - semaphores, barriers etc.
- Useful for getting threads to agree
Thread Pool

- We can now choose how our tasks will be executed
- The most common choice is a *thread pool*
- Conceptually this maintains a *pool* of threads and uses them to execute submitted tasks
- A thread pool has a *work list*
- Abstractly, each thread executes
  ```java
  while(true){
    Runnable task = workQueue.take();
    task.run();
  }
  ```
- Different implementations manage threads differently
Illustrating a thread pool
Illustrating a thread pool
Illustrating a thread pool
Illustrating a thread pool
Illustrating a thread pool
Illustrating a thread pool
Illustrating a thread pool

Thread Pool Executor

Task Queue

maxPoolSize

corePoolSize

Task

Thread

Task

Thread

Task

Thread
Illustrating a thread pool
Illustrating a thread pool
Illustrating a thread pool
Illustrating a thread pool

Thread Pool Executor

Task Queue

maxPoolSize

corePoolSize

Thread

Thread
Creating a Thread Pool

- The library provides ThreadPoolExecutor with this constructor:

  ```java
  public ThreadPoolExecutor(
      int corePoolSize, int maxPoolSize,
      long keepAlive, TimeUnit unit,
      BlockingQueue<Runnable> workQueue,
      RejectedExecutionHandler handler);
  ```

- `corePoolSize` and `maxPoolSize` - min and max number of threads to use
- `keepAlive` and `unit` - how long an unused thread should be kept
- `workQueue` - queue of runnable tasks to run
- `handler` - for the case where task cannot be accepted
Prepared thread pools

- Choosing values to create a thread pool can be complicated
- We can use prepared setups
- `java.util.concurrent.Executors` allows you to create
  - `newFixedThreadPool` - using a fixed number of threads
  - `newCachedThreadPool` - unbounded, but keeps minimum alive
  - `newSingleThreadedExecutor` - using a single thread
  - `newScheduledThreadPool` - fixed size, with scheduling

- A thread pool with scheduling allows us to run tasks after some delay or periodically
Blocking Queue

- All of the threads share the same work queue
- Therefore, this needs to be able to handle concurrent access

- A blocking queue is a queue that
  - Synchronizes queue operations
  - *blocks* a thread if it tries to
    - put to a full queue
    - take from an empty queue
  - includes *non-blocking* offer and poll
How it works

- Actually more complicated than this:

```java
private Object lock;

dpublic void put (E e) throws InterruptedException {
    synchronized(lock){
        while(is_full()){ lock.wait(); }
        private_put(e);
        notifyAll(); // now nonempty
    }
}

dpublic E take() throws InterruptedException{
    synchronized(lock){
        while(is_empty()){ lock.wait(); }
        E result = private_take();
        notifyAll(); // now nonfull
        return result;
    }
}
```
How it works

- Actually more complicated than this:

```java
private Object lock;

public void put (E e) throws InterruptedException {
    synchronized(lock){
        while(is_full()){
            lock.wait();
        }
        private_put(e);
        notifyAll(); // now nonempty
    }
}

public E take() throws InterruptedException{
    synchronized(lock){
        while(is_empty()){
            lock.wait();
        }
        E result = private_take();
        notifyAll(); // now nonfull
        return result;
    }
}
```
How it works

- Actually more complicated than this:

```java
private Object lock;

public void put (E e) throws InterruptedException {
    synchronized(lock){
        while(is_full()){ lock.wait(); }
        private_put(e);
        notifyAll(); // now nonempty
    }
}

public E take() throws InterruptedException{
    synchronized(lock){
        while(is_empty()){ lock.wait(); }
        E result = private_take();
        notifyAll(); // now nonfull
        return result;
    }
}
```
How it works

- Actually more complicated than this:

```java
private Object lock;

public void put (E e) throws InterruptedException {
    synchronized(lock){
        while(is_full()){
            lock.wait();
        }
        private_put(e);
        notifyAll(); // now nonempty
    }
}

public E take() throws InterruptedException{
    synchronized(lock){
        while(is_empty()){
            lock.wait();
        }
        E result = private_take();
        notifyAll(); // now nonfull
        return result;
    }
}
```
How it works

- Actually more complicated than this:

```java
private Object lock;

public void put (E e) throws InterruptedException {
    synchronized(lock){
        while(is_full()){
            lock.wait();
        }
        private_put(e);
        notifyAll(); // now nonempty
    }
}

public E take() throws InterruptedException{
    synchronized(lock){
        while(is_empty()){
            lock.wait();
        }
        E result = private_take();
        notifyAll(); // now nonfull
        return result;
    }
}
```
Java Monitors: An Overview

- In HotSpot the Entry Set is a List
- In HotSpot `notify` adds to the EntryList (since 1.6)
How it works

- Actually more complicated than this:

```java
public boolean offer(E e) {
    if(is_full()) return false;
    synchronized(lock) { private_put(e); }
    return true;
}

public E poll() {
    if(is_empty()) return null;
    synchronized(lock) { return private_take(); }
}
```
How it works

- Actually more complicated than this:

  ```java
  public boolean offer(E e) {
      if(is_full()) return false;
      synchronized(lock){ private_put(e); } 
      return true;
  }

  public E poll() {
      if(is_empty()) return null;
      synchronized(lock) { return private_take(); }
  }
  ```
Updating how threads run

- Previously we said the threads run like this

```java
run()
    {
        while(true)
            {
                Runnable task = workQueue.take();
                task.run();
            }
    }
```
Updating how threads run

- Previously we said the threads run like this
- We can now refine this (but it’s still more complicated)

```java
class RunnableTask implements Runnable {
    int id;
    public RunnableTask(int id) {
        this.id = id;
    }
    @Override
    public void run() {
        try {
            System.out.println(id);
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```java
public class WorkQueueExample {
    public static void main(String[] args) {
        BlockingQueue<RunnableTask> workQueue = new ArrayBlockingQueue<>(10);
        for (int i = 0; i < 10; i++) {
            workQueue.offer(new RunnableTask(i));
        }
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    RunnableTask task = workQueue.poll();
                    if (task != null) {
                        task.run();
                    } else {
                        handleNoWork();
                        break;
                    }
                }
            }
        }).start();
    }
}
```
Updating how threads run

• Previously we said the threads run like this
• We can now refine this (but it’s still more complicated)

```java
run(){
    Runnable task = workQueue.poll();
    while(task!=null) {
        task.run();
        task = workQueue.poll();
    }
    handleNoWork();
}
```

```java
execute(Runnable task){
    if(!workQueue.offer(task))
        reject(task);
}
```
• Notice that we have *two conditions* - queue is full or empty

• i.e. a thread is *waiting* to and a *take* occurs it will be woken up by *notifyAll* in *take*

• Why is *notifyAll* needed instead of *notify*?

• What is the advantage of locking on a private lock?

• *BlockingQueue* also has versions of *offer* and *poll* that use *timeouts*
Understanding how it runs

- This machine has 8 cores on two quad core chips
- 32 KB L1 cache, 256 KB L2 cache
- 64-byte cache line (16 integers)

- Effects - cache locality, cache misses, cache sharing
Issue with our example

• All threads are using the same array data structures
• This means that we will have lots of cache invalidations, causing cache misses
• There are lots of complicated solutions to this (loop tiling)
• Need to consider where objects are placed in memory
Other concurrent data structures

- We finish by considering other notions of concurrent data structure (if we have time)
- These are very important as threads communicate via *shared memory*
- They typically offer a level of abstraction that means we don’t need to think about synchronization
The importance of synchronization

If we do not synchronize collections

- ConcurrentModificationException
- Unexpected behaviour causing
  - Incorrect results
  - Divergence (hanging) - problem in HashMap

If we synchronize collections badly

- Deadlock
- Dramatic loss of efficiency
- Failure to actually stop the above
An example: HashMap

```java
public class HashMapFail{
    static Map<String,Integer> map = new HashMap<>();
    public static void main(String[] args) throws Exception {
        Worker one = new Worker(0,10000);
        Worker two = new Worker(100000,10000);
        one.start(); two.start();
        one.join(); two.join();
        System.out.println(map.size()); //should be 20000
    }
    
    private static class Worker extends Thread{
        final int start; final int step;
        public Worker(int sta, int ste){ start=sta;step=ste;}
        public void run(){
            for(int i=start;i<start+step;i++)
                map.put(""+i,i);
        }
    }
}
```
An example: HashMap

```java
public class HashMapFail{
    static Map<String,Integer> map = new HashMap<>();
    public static void main(String[] args) throws Exception {
        Worker one = new Worker(0,10000);
        Worker two = new Worker(100000,10000);
        one.start(); two.start();
        one.join(); two.join();
        System.out.println(map.size()); //should be 20000
    }
    private static class Worker extends Thread{
        final int start; final int step;
        public Worker(int sta, int ste){ start=sta;step=ste;}
        public void run(){
            for(int i=start;i<start+step;i++)
                map.put(""+i,i);
        }
    }
}
```
Other Concurrent Datastructures

- Make use of the built in data structures - only build your own if you *need* to

- `java.util.Collections.synchronizedCollection` makes all methods synchronized

- `BlockingQueue` - `LinkedBlockingQueue`, `ArrayBlockingQueue`, ...

- `ConcurrentHashMap` - many readers, adjustable writers

- `CopyOnWriteArrayList` - keeps a local copy, for many reads/few writes