Parallel Computing

Prof. Marco Bertini
Shared memory: Java threads
Introduction

• Java provides built-in multithreading
  • Low level primitives:
    • Class Thread / Interface Runnable
  • High level framework:
    • Java Concurrency Utilities
Low level primitives
Class Thread

- Class Thread constructors
  public Thread(String threadName)
  public Thread()

- Code for thread in thread’s run method

- Method sleep makes thread inactive

- Method yield hints the scheduler that the thread is willing to yield its current use of processor

- Method interrupt interrupts a running thread

- Method isAlive checks status of a thread

- Method setName sets a thread’s name

- Method join waits for thread to finish and continues from current thread
Thread states

- Born state
  - Thread was just created
- Ready state
  - Thread’s start method invoked
  - Thread can now execute
- Running state
  - Thread is assigned a processor and running
- Dead state
  - Thread has completed or exited
  - Eventually disposed of by system
public class ThreadHelloWorld {

    public static void main(String[] args) throws InterruptedException {
        Thread myThread = new Thread() {
            public void run() {
                System.out.println("Hello from new thread");
            }
        };
        myThread.start();
        Thread.yield(); // gives the thread a chance to run first
        System.out.println("Hello from main thread");
        myThread.join();
    }
}

The order of the output will change…
Remind: make no assumptions regarding execution order
Thread synchronization

- Java uses monitors for thread synchronization

- The synchronized keyword uses the lock that is built into every Java Object

  - Every synchronized method of an object has a monitor

    - we can synchronize any statement acquiring the lock on an object

  - One thread inside a synchronized method at a time

  - All other threads block until method finishes

  - Next highest priority thread runs when method finishes
Thread synchronization

public synchronized void method() {
    // statements
}

• is syntactic sugar for

public void method() {
    synchronized(this) {
        // statements
    }
}

synchronizing statements allows a finer granularity in parallelism
public class RaceCondition {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            private int count = 0;

            public void increment() {
                ++count;
            }

            public int getCount() {
                return count;
            }
        }

        final Counter counter = new Counter();

        class CountingThread extends Thread {
            public void run() {
                System.out.println(counter.getCount());
            }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        ReadingThread t3 = new ReadingThread();
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
        System.out.println(counter.getCount());
    }
}
public class RaceCondition {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            private int count = 0;
            public synchronized void increment() {
                ++count;
            }
            public synchronized int getCount() { return count; }
        }
        final Counter counter = new Counter();
        class CountingThread extends Thread {
            public void run() {
                System.out.println(counter.getCount());
            }
        }
        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        ReadingThread t3 = new ReadingThread();
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
        System.out.println(counter.getCount());
    }
}

public class RaceCondition {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            private int count = 0;
            public synchronized void increment() {
                ++count;
            }
            public synchronized int getCount() { return count; }
        }
        final Counter counter = new Counter();
        class CountingThread extends Thread {
            public void run() {
                System.out.println(counter.getCount());
            }
        }
        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        ReadingThread t3 = new ReadingThread();
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
        System.out.println(counter.getCount());
    }
}

Sync example
import java.util.Random;
class Philosopher extends Thread {
    private Chopstick first, second;
    private Random random;
    private int thinkCount;

    public Philosopher(Chopstick left, Chopstick right) {
        if (left.getId() < right.getId()) {
            first = left; second = right;
        } else {
            first = right; second = left;
        }
        random = new Random();
    }
    public void run() {
        try {
            while (true) {
                ++thinkCount;
                if (thinkCount % 10 == 0)
                    System.out.println("Philosopher " + this + " has thought " + thinkCount + " times");
                Thread.sleep(random.nextInt(1000)); // Think for a while
                synchronized(first) { // Grab first chopstick
                    synchronized(second) { // Grab second chopstick
                        Thread.sleep(random.nextInt(1000)); // Eat for a while
                    }
                }
            }
        } catch (InterruptedException e) {}}
}

The objects act as mutexes
Sync using objects

import java.util.Random;
class Philosopher extends Thread {
    private Chopstick first, second;
    private Random random;
    private int thinkCount;
    public Philosopher(Chopstick left, Chopstick right) {
        if(left.getId() < right.getId()) {
            first = left; second = right;
        } else {
            first = right; second = left;
        }
        random = new Random();
    }
    public void run() {
        try {
            while(true) {
                ++thinkCount;
                if (thinkCount % 10 == 0)
                    System.out.println("Philosopher " + this + " has thought " + thinkCount + " times");
                Thread.sleep(random.nextInt(1000));     // Think for a while
                synchronized(first) {                   // Grab first chopstick
                    synchronized(second) {                // Grab second chopstick
                        Thread.sleep(random.nextInt(1000)); // Eat for a while
                    }
                }
            }
        } catch(InterruptedException e) {}  
    }
}

class Chopstick {
    private int id;
    public Chopstick(int id) { this.id = id; }
    public int getId() { return id; }
}

class Philosopher {
    private Chopstick first, second;
    private Random random;
    private int thinkCount;
    public Philosopher(Chopstick left, Chopstick right) {
        if(left.getId() < right.getId()) {
            first = left; second = right;
        } else {
            first = right; second = left;
        }
        random = new Random();
    }
    public void run() {
        try {
            while(true) {
                ++thinkCount;
                if (thinkCount % 10 == 0)
                    System.out.println("Philosopher " + this + " has thought " + thinkCount + " times");
                Thread.sleep(random.nextInt(1000));     // Think for a while
                synchronized(first) {                   // Grab first chopstick
                    synchronized(second) {                // Grab second chopstick
                        Thread.sleep(random.nextInt(1000)); // Eat for a while
                    }
                }
            }
        } catch(InterruptedException e) {}  
    }
}

public static void main(String[] args) throws InterruptedException {
    Philosopher[] philosophers = new Philosopher[5];
    Chopstick[] chopsticks = new Chopstick[5];
    for (int i = 0; i < 5; ++i)
        chopsticks[i] = new Chopstick(i);
    for (int i = 0; i < 5; ++i) {
        philosophers[i] = new Philosopher(chopsticks[i], chopsticks[(i + 1) % 5]);
        philosophers[i].start();
    }
    for (int i = 0; i < 5; ++i)
        philosophers[i].join();
}

The objects act as mutexes
import java.util.Random;

class Philosopher extends Thread {
    private Chopstick first, second;
    private Random random;
    private int thinkCount;

    public Philosopher(Chopstick left, Chopstick right) {
        if(left.getId() < right.getId()) {
            first = left; second = right;
        } else {
            first = right; second = left;
        }
        random = new Random();
    }

    public void run() {
        try {
            while(true) {
                ++thinkCount;
                if (thinkCount % 10 == 0)
                    System.out.println("Philosopher " + this + " has thought " + thinkCount + " times");
                Thread.sleep(random.nextInt(1000));     // Think for a while
                synchronized(first) {                   // Grab first chopstick
                    synchronized(second) {                // Grab second chopstick
                        Thread.sleep(random.nextInt(1000)); // Eat for a while
                    }
                }
            }
        } catch(InterruptedException e) {};
    }
}

class Chopstick {
    private int id;
    public Chopstick(int id) { this.id = id; }
    public int getId() { return id; }
}

public static void main(String[] args) throws InterruptedException {
    Philosopher[] philosophers = new Philosopher[5];
    Chopstick[] chopsticks = new Chopstick[5];

    for (int i = 0; i < 5; ++i)
        chopsticks[i] = new Chopstick(i);
    for (int i = 0; i < 5; ++i) {
        philosophers[i] = new Philosopher(chopsticks[i], chopsticks[(i + 1) % 5]);
        philosophers[i].start();
    }
    for (int i = 0; i < 5; ++i)
        philosophers[i].join();
}

This is the solution provided by Dijkstra: relative ordering of resources and ordered acquisition
Alien methods

- A synchronized method should not call a method it knows nothing about - an \textit{alien method} - since it may acquire a second lock without respecting the correct order, thus risking deadlock.

- Solution: reduce synchronization to statements and do not call the alien method in that synchronized section.
class Downloader extends Thread {

    private InputStream in;
    private OutputStream out;
    private ArrayList<ProgressListener> listeners;

    public Downloader(URL url, String outputFilename) throws IOException {
        in = url.openConnection().getInputStream();
        out = new FileOutputStream(outputFilename);
        listeners = new ArrayList<ProgressListener>;
    }

    public synchronized void addListener(ProgressListener listener) {
        listeners.add(listener);
    }

    public synchronized void removeListener(ProgressListener listener) {
        listeners.remove(listener);
    }

    private synchronized void updateProgress(int n) {
        for (ProgressListener listener:
            listeners)
            listener.onProgress(n);
    }

    public void run() {
        int n = 0, total = 0;
        byte[] buffer = new byte[1024];
        try {
            while((n = in.read(buffer)) != -1) {
                out.write(buffer, 0, n);
                total += n;
                updateProgress(total);
            }
            out.flush();
        } catch (IOException e) { }
    }
}
class Downloader extends Thread {
    private InputStream in;
    private OutputStream out;
    private ArrayList<ProgressListener> listeners;
    
    public Downloader(URL url, String outputFileFilename) throws IOException {
        in = url.openConnection().getInputStream();
        out = new FileOutputStream(outputFilename);
        listeners = new ArrayList<ProgressListener>();
    }
    
    public synchronized void addListener(ProgressListener listener) {
        listeners.add(listener);
    }
    
    public synchronized void removeListener(ProgressListener listener) {
        listeners.remove(listener);
    }
    
    private synchronized void updateProgress(int n) {
        for (ProgressListener listener: listeners)
            listener.onProgress(n);
    }
    
    public void run() {
        int n = 0, total = 0;
        byte[] buffer = new byte[1024];
        try {
            while((n = in.read(buffer)) != -1) {
                out.write(buffer, 0, n);
                total += n;
                updateProgress(total);
            }
        } catch (IOException e) {
        } finally {
            out.flush();
        }
    }
}

The methods of the Subject are synchronized but the notification method class an alien method in the observer
class Downloader extends Thread {
    private InputStream in;
    private OutputStream out;
    private ArrayList<ProgressListener> listeners;
    public Downloader(URL url, String outputFilename) throws IOException {
        in = url.openConnection().getInputStream();
        out = new FileOutputStream(outputFilename);
        listeners = new ArrayList<ProgressListener>();
    }
    public synchronized void addListener(ProgressListener listener) {
        listeners.add(listener);
    }
    public synchronized void removeListener(ProgressListener listener) {
        listeners.remove(listener);
    }
    private synchronized void updateProgress(int n) {
        for (ProgressListener listener: listeners)
            listener.onProgress(n);
    }
    public void run() {
        int n = 0, total = 0;
        byte[] buffer = new byte[1024];
        try {
            while((n = in.read(buffer)) != -1) {
                out.write(buffer, 0, n);
                total += n;
                updateProgress(total);
            }
            out.flush();
        } catch (IOException e) { }
    }
}

public synchronized void removeListener(ProgressListener listener) {
    listeners.remove(listener);
}

private synchronized void updateProgress(int n) {
    ArrayList<ProgressListener> listenersCopy;
    synchronized(this) {
        listenersCopy = (ArrayList<ProgressListener>)listeners.clone();
    }
    for (ProgressListener listener: listenersCopy)
        listener.onProgress(n);
}

private void updateProgress(int n) {
    ArrayList<ProgressListener> listenersCopy;
    synchronized(this) {
        listenersCopy = (ArrayList<ProgressListener>)listeners.clone();
    }
    for (ProgressListener listener: listenersCopy)
        listener.onProgress(n);
}
wait/notify

- The Object class provides other means to synchronize threads, acting as monitors of a queue whose access is controlled by wait/notify methods:

  - `public final void wait() throws InterruptedException`
  - `public final void wait(long timeout, int nanos) throws InterruptedException`
    - Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method.
    - The current thread must own this object's monitor. The thread releases ownership of this monitor.
  - `public final void notify()`
  - `public final void notifyAll()`
    - Wakes up a single thread that is waiting on this object's monitor / wake up all threads waiting.
    - The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
wait/notify

- A thread can call `wait()` on an object that has locked:
  - the lock is released
  - the thread goes into waiting state

- Other threads may obtain that released lock, then they perform the required operations and call:
  - `notify()` to awaken a waiting thread
  - `notifyAll()` to awaken all the threads waiting the object
  - The awakened threads have to acquire the lock
  - notifications are not cumulated
wait/notify

• A thread can call wait() on an object that has locked:
  - the lock is released
  - the thread goes into waiting state
  - Other threads may obtain that released lock, then they perform:
    • notify() to awaken a waiting thread
    • notifyAll() to awaken all the threads waiting the object
  - The awakened threads have to acquire the lock
  - notifications are not cumulated

Producer

synchronized void put() {
  while buffer=full
    wait()
  Put in buffer
  notify()
}

Consumer

synchronized void get() {
  while buffer=empty
    wait()
  Get from buffer
  notify()
}
public class Monitor {
    private boolean full = false;
    private boolean stop = false;
    private String buffer;

    synchronized void send(String msg) {
        if (full) {
            try {
                wait(); // if full wait until
                // it becomes empty
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // if empty becomes full
        // and receive the msg
        full = true;
        notify();
        buffer = msg;
    }

    synchronized void endMessages() {
        stop = true; // no more messages
        // from the producer
    }

    synchronized String receive() {
        if (!full) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        full = false;
        notify();
        return buffer;
    }

    synchronized boolean isEndCommunications() {
        return stop & !full; // true if there
        // are no more messages to consume and
        // the producer said it was going to stop
    }
}

Use this monitor to communicate between threads
High level framework
Beyond intrinsic locks

- Locking on an object with synchronized, as with low level APIs has some limitations:
  - lock acquisition and release are only in the same method or start/end of statements
  - a thread may have to wait a long time before acquiring it
  - no timeout while waiting for the lock
java.util.concurrent.locks.ReentrantLock

- It is a more powerful alternative to intrinsic lock:
  - can be created with a fairness parameter to give precedence to threads that have been waiting a long time
  - lower throughput but less variances in time to obtain locks
  - can be acquired and released in different methods
  - interruptible lock waits that support time-out
  - immediate acquisition of lock, independently of how many other threads were waiting for it
java.util.concurrent.locks.ReentrantLock

- It is a more powerful alternative to intrinsic lock:

  Lock lock = new ReentrantLock();
  lock.lock();
  try {
    // use shared resources
  } finally {
    lock.unlock();
  }

  Use the finally to be sure to release the lock!

- can be created with a fairness parameter to give precedence to threads that have been waiting a long time
- lower throughput but less variances in time to obtain locks
- can be acquired and released in different methods
- interruptible lock waits that support time-out
- immediate acquisition of lock, independently of how many other threads were waiting for it
java.util.concurrent.locks.ReentrantLock

- It is a more powerful alternative to intrinsic locks:
  - can be created with a fairness parameter to give precedence to threads that have been waiting a long time
  - lower throughput but less variance in time to obtain locks
  - can be acquired and released in different methods
  - interruptible lock waits that support time-out
  - immediate acquisition of lock, independently of how many other threads were waiting for it

```java
class ReentrantLockExample {
    private ReentrantLock lock;

    public void foo() {
        lock.lock();
        // use shared resources
        lock.unlock();
    }

    public void bar() {
        lock.unlock();
    }
}
```

Use the finally to be sure to release the lock!
java.util.concurrent.locks.ReentrantLock

- It is a more powerful alternative to intrinsic locks:
  - can be created with a fairness parameter to give precedence to threads that have been waiting a long time
  - lower throughput but less variance in time to obtain locks
  - can be acquired and released in different methods
  - interruptible lock waits that support time-out
  - immediate acquisition of lock, independently of how many other threads were waiting for it

```java
private ReentrantLock lock;

public void foo() {
    lock.lock();
    ...
}
public void bar() {
    lock.unlock();
    ...
}
```

Use the finally to be sure to release the lock!
java.util.concurrent.locks.ReentrantLock

- It is a more powerful alternative to intrinsic lock:
  - can be created with a fairness parameter to give precedence to threads that have been waiting a long time
  - lower throughput but less variance in time to obtain locks
  - can be acquired and released in different methods
  - interruptible lock waits that support time-out
  - immediate acquisition of lock, independently of how many other threads were waiting for it

```java
private ReentrantLock lock;

public void foo() {
    lock.lock();
    ...
    lock.lock();
    ...
}

public void bar() {
    ...
    lock.unlock();
    ...
}

Use the finally to be sure to release the lock!

```
Interruptible locking

• A lock due to intrinsic locking is not interruptible (i.e. Thread interrupt method does not stop it)
  • therefore a deadlock can be stopped only by killing the JVM!

• ReentrantLock is interruptible
Interruptible locking

- A lock due to intrinsic locking is not interruptible (i.e. Thread interrupt method does not stop it).
- Therefore a deadlock can be stopped only by killing the JVM!
- ReentrantLock is interruptible.

```java
final Object lock1 = new Object();
final Object lock2 = new Object();

Thread t1 = new Thread() {
    public void run() {
        try {
            synchronized(lock1) {
                Thread.sleep(1000);
                synchronized(lock2) {}
            }
        } catch (InterruptedException e) {
            System.out.println("t1 interrupted");
        }
    }
};

If another thread acquires o1 and o2 in the opposite order we have a deadlock!
Interruptible locking

```java
final ReentrantLock lock1 = new ReentrantLock();
final ReentrantLock lock2 = new ReentrantLock();

Thread t1 = new Thread() {
    public void run() {
        try {
            lock1.lockInterruptibly();
            Thread.sleep(1000);
            lock2.lockInterruptibly();
        } catch (InterruptedException e) {
            System.out.println("t1 interrupted");
        }
    }
};

a t1.interrupt() now stops the deadlock (if another thread acquires the two locks with a different order...)
```
Interruptible locking

```java
final ReentrantLock lock1 = new ReentrantLock();
final ReentrantLock lock2 = new ReentrantLock();

Thread t1 = new Thread() {
    public void run() {
        try {
            lock1.lockInterruptibly();
            Thread.sleep(1000);
            lock2.lockInterruptibly();
        } catch (InterruptedException e) {
            System.out.println("t1 interrupted");
        }
    }
};
```

Note: reentrant is a term that indicates a block of code that can be entered by another actor before an earlier invocation has finished, without affecting the path that the first actor would have taken through the code. That is, it is possible to re-enter the code while it's already running and still produce correct results. E.g. some code that can be interrupted in the middle of its execution and then safely called again.
tryLock and livelocks

• It may be tempting to use a tryLock with timeout to solve a deadlock, since there’s no need to acquire resources in the required order but…

• … we are not avoiding deadlock, just recovering from them

• … we are risking a livelock: if multiple threads timeout at the same time they may have immediately another deadlock. Threads are not really progressing, unless we randomize the timeout.
Hand-over-hand locking

- It’s a fine-grained locking, where multiple locks are used to lock the smallest possible part of a data structure that the current thread needs to operate on.

- As we acquire new locks we unlock the older ones.

- Can be implemented with ReentrantLock, that allows to lock/unlock whenever we need.
Concurrent Linked List

- Locking the whole method that inserts/searches an element does not scale: access becomes too much sequential… we need fine grained lock.

- Solution: lock only the position we are examining for the insertion: hand-over-hand lock.

- Each node needs a ReentrantLock

To insert a node, we need to lock the two nodes on either side of the point we’re going to insert. We start by locking the first two nodes of the list. If this isn’t the right place to insert the new node, we unlock the first node and lock the third… This continues until we find the appropriate place, insert the new node, and finally unlock the nodes on either side.
Concurrent Sorted List: example

public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value)
                    return;
            }
            finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    }
    finally {
        current.lock.unlock();
    }
    current = next;
    next = current.next;

    try {
        if (next == tail ||
            next.value < value) {
            Node node = new
            Node(value, current, next);
            next.prev = node;
            current.next = node;
        }
    }
    finally {
        next.lock.unlock();
    }
}
public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;
    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                        Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        current.lock.unlock();
    }
}
Concurrent Sorted List: example

```java
public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                        Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        next.lock.unlock();
    }
}
```
Concurrent Sorted List: example

```java
public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                        Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        next.lock.unlock();
    }
}
```

Lock next node
public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                        Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        current.lock.unlock();
    }
}
Concurrent Sorted List: example

public void insert(int value) {
    Node current = head;
    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                        Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        current.lock.unlock();
    }
    current = next;
    next = current.next;
}
Concurrent Sorted List: example

```java
public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                        Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        next.lock.unlock();
        current.lock.unlock();
        current = next;
        next = current.next;
    }
}
```
Concurrent Sorted List: example

```java
public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        try {
            if (next == tail ||
                next.value < value) {
                Node node = new
                Node(value, current, next);
                next.prev = node;
                current.next = node;
        } finally {
            current.lock.unlock();
        }
        current = next;
        next = current.next;
    } finally {
        next.lock.unlock();
    }

    return;
}
} finally {
    current.lock.unlock();
}
```
public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                      Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        current.lock.unlock();
    }
}

public void insert(int value) {
    Node current = head;
    current.lock.lock();
    Node next = current.next;

    try {
        while (true) {
            next.lock.lock();
            try {
                if (next == tail ||
                    next.value < value) {
                    Node node = new
                      Node(value, current, next);
                    next.prev = node;
                    current.next = node;
                    return;
                }
            } finally {
                current.lock.unlock();
            }
            current = next;
            next = current.next;
        }
    } finally {
        current.lock.unlock();
    }
}
Concurrent Sorted List: example

public int size() {
    Node current = tail;
    int count = 0;

    while (current.prev != head) {
        ReentrantLock lock = current.lock;
        lock.lock();
        try {
            ++count;
            current = current.prev;
        } finally {
            lock.unlock();
        }
    }

    return count;
}
Semaphore

- java.util.concurrent.Semaphore provides counting semaphores. The number of threads that can get a permit to access a critical section is decided in the initialization.

  - initializing to one creates a binary semaphore

```java
static int counter = 0;
static Semaphore semaphore = new Semaphore(1);

public static void incrementCounter() {
    try {
        semaphore.acquire();
        counter++;
        semaphore.release();
    } catch (InterruptedException ex) {
    }
}
```
Condition variables

- To use a condition variable effectively, we need to follow a very specific pattern:

```java
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
    while (! condition_is_true)
        condition.await();
    // use shared resources
} finally {
    lock.unlock();
}
```
Condition variables

• To use a condition variable effectively, we need to follow a very specific pattern:

```java
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
    while (! condition_is_true)
        condition.await();
    // use shared resources
} finally {
    lock.unlock();
}
```

• A condition variable is associated with a lock, and a thread must hold that lock before being able to wait on the condition.

• Once it holds the lock, it checks to see if the condition that it's interested in is already true.

• If it is, then it continues with whatever it wants to do and unlocks the lock.
Condition variables

• To use a condition variable effectively, we need to follow a very specific pattern:

```java
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
    while (! condition_is_true)
        condition.await();
    // use shared resources
} finally {
    lock.unlock();
}
```
Condition variables

• To use a condition variable effectively, we need to follow a very specific pattern:

```java
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
    while (! condition_is_true)
        condition.await();
    // use shared resources
} finally {
    lock.unlock();
}
```

• If, however, the condition is not true, it calls `await()`, which atomically unlocks the lock and blocks on the condition variable.

• When another thread calls `signal()` or `signalAll()` to indicate that the condition might now be true, `await()` unblocks and automatically reacquires the lock. When `await()` returns, it only indicates that the condition might be true. This is why `await()` is called within a loop—we need to go back, recheck whether the condition is true, and potentially block on `await()` again if necessary.
Atomic variables

- java.util.concurrent.atomic provides a set of types that can be accessed and modified atomically, without need of synchronization or locks.
  - we can not miss locks
  - we can not have a deadlock, since there are no locks
  - we can implement non-blocking, lock-free algorithms
- Example:
  ```java
  final AtomicInteger counter = new AtomicInteger();
  ```
  is a good substitute for the counter class that required synchronized methods
Atomic variables

- `java.util.concurrent.atomic` provides a set of types that can be accessed and modified atomically, without need of synchronization or locks.
  - we can not miss locks
  - we can not have a deadlock, since there are no locks
  - we can implement non-blocking, lock-free algorithms
  - Example:
    ```java
    final AtomicInteger counter = new AtomicInteger();
    ```
    is a good substitute for the counter class that required synchronized methods

Atomic types provide methods such as `set/get, compareAndSet, getAndSet`
Executors and Thread pools

• In small applications is OK to manually create and start threads, but in more complex cases it is better to encapsulate the creation in an Executor:
  
  • it separates thread management and creation from the rest of the application.

  • Threads are created from pools of threads, avoiding the creation of an excessive number of threads, thus keeping low the overhead
Executors

• An executor will take a runnable object and execute it:
  
  • `e.execute(r);`

  • versus

  • `(new Thread(r)).start();`

• an important difference is that the executor will likely take an already existing thread to assign the runnable object to it.

• The pool of thread may have a fixed or a dynamic size
public class EchoServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(4567);
        int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;
        ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);
        while (true) {
            Socket socket = server.accept();
            executor.execute(new ConnectionHandler(socket));
        }
    }

    public class ConnectionHandler implements Runnable {
        InputStream in;
        OutputStream out;
        public void run() {
            try {
                int n;
                byte[] buffer = new byte[1024];
                while ((n = in.read(buffer)) != -1) {
                    out.write(buffer, 0, n);
                    out.flush();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

class Executors: example

public static void main(String[] args) throws IOException {
    public class EchoServer {
        public static void main(String[] args) throws IOException {
            ServerSocket server = new ServerSocket(4567);
            int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;
            ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);
            while (true) {
                Socket socket = server.accept();
                executor.execute(new ConnectionHandler(socket));
            }
        }
    }
}

public class ConnectionHandler implements Runnable {
    InputStream in;
    OutputStream out;
    public void run() {
        try {
            int n;
            byte[] buffer = new byte[1024];
            while ((n = in.read(buffer)) != -1) {
                out.write(buffer, 0, n);
                out.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
public class EchoServer {

    public static void main(String[] args) throws IOException {
        class ConnectionHandler implements Runnable {
            InputStream in; OutputStream out;
            ConnectionHandler(Socket socket) throws IOException {
                in = socket.getInputStream();
                out = socket.getOutputStream();
            }

            public void run() {
                try {
                    int n;
                    byte[] buffer = new byte[1024];
                    while((n = in.read(buffer)) != -1) {
                        out.write(buffer, 0, n);
                        out.flush();
                    }
                } catch (IOException e) {}
            }
        }

        ServerSocket server = new ServerSocket(4567);
        int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;
        ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);
        while (true) {
            Socket socket = server.accept();
            executor.execute(new ConnectionHandler(socket));
        }
    }
}

The old approach would have been:
while (true) {
    Socket socket = server.accept();
    Thread handler = new Thread(new ConnectionHandler(socket));
    handler.start();
}
public class EchoServer {

    public static void main(String[] args) throws IOException {
        class ConnectionHandler implements Runnable {
            InputStream in; OutputStream out;
            ConnectionHandler(Socket socket) throws IOException {
                in = socket.getInputStream();
                out = socket.getOutputStream();
            }

            public void run() {
                try {
                    int n;
                    byte[] buffer = new byte[1024];
                    while((n = in.read(buffer)) != -1) {
                        out.write(buffer, 0, n);
                        out.flush();
                    }
                } catch (IOException e) {}
            }
        }

        ServerSocket server = new ServerSocket(4567);
        int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;
        ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);
        while (true) {
            Socket socket = server.accept();
            executor.execute(new ConnectionHandler(socket));
        }
    }
}
public class EchoServer {

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

class ConnectionHandler implements Runnable {

    InputStream in; OutputStream out;
    ConnectionHandler(Socket socket) throws IOException {
        in = socket.getInputStream();
        out = socket.getOutputStream();
    }

    public void run() {
        try {
            int n;
            byte[] buffer = new byte[1024];
            while ((n = in.read(buffer)) != -1) {
                out.write(buffer, 0, n);
                out.flush();
            }
        } catch (IOException e) {} // catch exceptions
    }
}

    ServerSocket server = new ServerSocket(4567); // create a server socket
    int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2; // get number of processors
    ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize); // create executor service
    while (true) {
        Socket socket = server.accept(); // accept a client connection
        executor.execute(new ConnectionHandler(socket)); // execute the handler
    }
}

How many threads?

A good rule of thumb is that for computation-intensive tasks, you probably want to have approximately the same number of threads as available cores. Larger numbers are appropriate for I/O-intensive tasks.
Synchronized Collections

• It is possible to transform a thread-unsafe collection (e.g. ArrayList) into a thread-safe version using wrappers like:
  public static <T> Collection<T> synchronizedCollection (Collection<T> c)

• If all access is performed using the returned collection, then serial access through synchronized code is guaranteed.
  E.g.: List list = Collections.synchronizedList(new ArrayList());
Concurrent Collections

- The java.util.concurrent package includes several data structures design for concurrent access:
  - BlockingQueue defines a FIFO structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
  - ConcurrentMap is a Map with atomic operations. The standard implementation is ConcurrentHashMap, which is a concurrent analog of HashMap.
  - ConcurrentNavigableMap is a sort of ConcurrentMap that supports approximate matches. The standard implementation is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.
Copy-on-write Collections

• Copy-on-write is a strategy to manage local identical copies of some information that occasionally is modified by some task. Each task receives a pointer to the data and a local copy is created only when new data is written. Other tasks do not see the modified data.

• CopyOnWriteArrayList<E> is a thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

• There is also CopyOnWriteArraySet<E>
Copy-on-write Collections

Reconsider the alien method example using a copy-on-write data structure:

```java
private CopyOnWriteArrayList<ProgressListener> listeners;

public void addListener(ProgressListener listener) {
    listeners.add(listener);
}

public void removeListener(ProgressListener listener) {
    listeners.remove(listener);
}

private void updateProgress(int n) {
    for (ProgressListener listener: listeners)
        listener.onProgress(n);
}
```
Producer/Consumer: example

class FrameExtractor implements Runnable {
    private BlockingQueue<FFMpegFrame> queue;
    private FFMpegVideo video;

    public FrameExtractor(String videoName,
                            BlockingQueue<FFMpegFrame> queue) {
        video.open(videoName);
        this.queue = queue;
    }

    public void run() {
        try {
            FFMpegFrame frame = video.getNextFrame();
            queue.put(frame);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
Producer/Consumer: example

class FrameProcessor implements Runnable {
    private BlockingQueue<FFMpegFrame> queue;
    private ConcurrentHashMap<Integer, VisualFeature> results;
    private VisualFeatureProcessor processor = new CNNProcessor();

    public FrameProcessor(BlockingQueue<FFMpegFrame> queue, ConcurrentHashMap<Integer, VisualFeatures> results) {
        this.queue = queue;
        this.results = results;
    }

    public void run() {
        try {
            while(true) {
                FFMpegFrame frame = queue.take();
                if (frame.isNull())
                    break;

                VisualFeature features = processor.process(frame);
                results.put(frame.getFrameNumber(), features);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
ArrayBlockingQueue<FFMpegFrame> queue = new ArrayBlockingQueue<FFMpegFrame>(100);
ConcurrentHashMap<Integer, VisualFeature> results = new ConcurrentHashMap<Integer, VisualFeature>();

Thread consumer = new Thread(new FrameProcessor(queue, results));
Thread producer = new Thread(new FrameExtractor(queue));

consumer.start();
producer.start();
producer.join();
queue.put(new VideoFrame(Null)); // signal end
// of processing
consumer.join();
ArrayBlockingQueue<FFMpegFrame> queue = new ArrayBlockingQueue<FFMpegFrame>(100);
ConcurrentHashMap<Integer, VisualFeature> results = new ConcurrentHashMap<Integer, VisualFeature>();

ExecutorService executor = Executors.newCachedThreadPool();
for (int i = 0; i < NUM_CONSUMERS; ++i)
    executor.execute(new FrameProcessor(queue, results));
Thread producer = new Thread(new FrameExtractor(queue));

producer.start();
producer.join();
for (int i = 0; i < NUM_CONSUMERS; ++i)
    queue.put(new VideoFrame(Null));
executor.shutdown();
executor.awaitTermination(10L, TimeUnit.MINUTES);
Producer/Consumer: example

• Further speedup can be reached by changing the consumers:

  • instead of updating a shared concurrent hash map they can update a local hash map then merge the results to the global map at the end of the while(true) loop.

  • reducing access to shared variables increases parallelism…
Books

• Parallel Programming for Multicore and Cluster Systems, Thomas Dauber and Gudula Rünger, Springer - Chapt. 6

• Principles of Parallel Programming, Calvin Lyn and Lawrence Snyder, Pearson - Chapt. 6