# Why Threads?

- On single or multiprocessor machines
  - Allow progress when a process is blocked – eg for I/O
  - Convenient programming model
  - Multitasking
- On multiprocessor machines
  - Speedup through parallel processing

# What is a Java Thread?

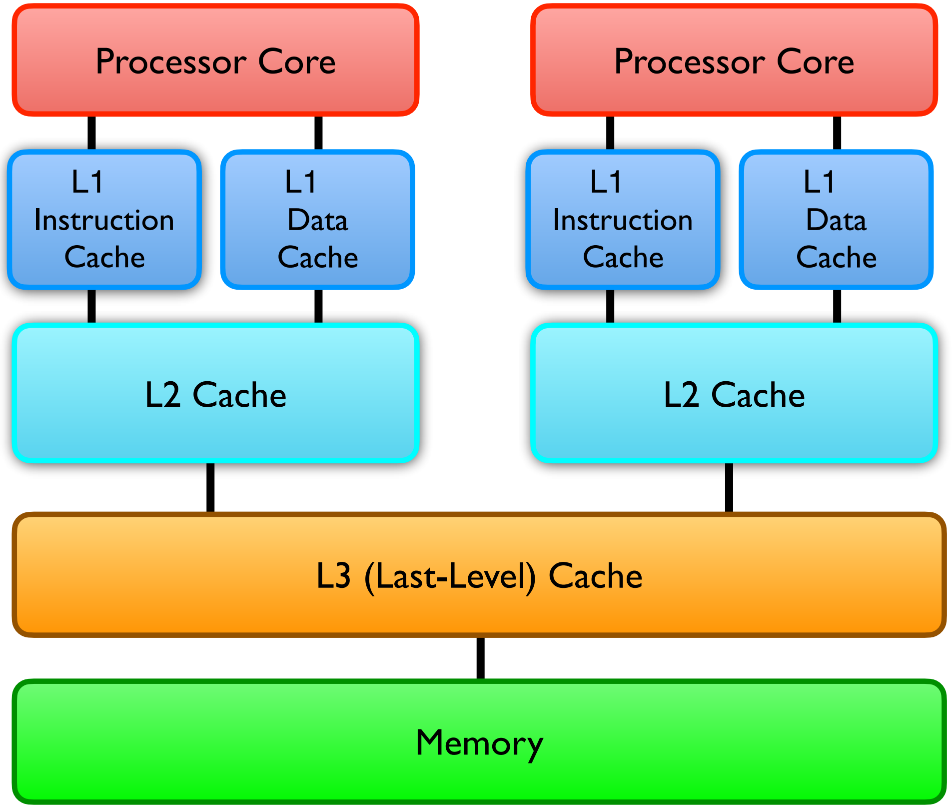
- Conceptually, like a one-shot machine
  - its program is a `Runnable`:

```
public interface Runnable {
   void run();
}
```

- A thread dies when its `run` method terminates
- An application terminates when its last non-daemon thread terminates
- Threads co-operate through shared stored data

# Why is Multi-Thread Programming So Hard?

- Modern computer architectures are all about having lots of cores running as fast as possible



# java.util.concurrent

- Higher-level utilities that abstract away from the difficult detail of threads:
  - Executor Framework
  - Synchronizers
  - Concurrent Collections
  - Atomic Variables

# Possible `Executor` Implementations

Abstract model of execution context, supports different possibilities:

- In-thread

```
public class InThread implements Executor {
  public void execute( Runnable task ) {
    task.run();
  }
}
```

- Thread-per-task

```
public class ThreadPerTask implements Executor {
  public void execute( Runnable task ) {
    new Thread(task).start();
  }
}
```

- Thread-pool…

# `Executors` – Class supplying `Executor` implementations

- `Executors.newFixedThreadPool()`
  - and `Executors.newSingleThreadExecutor()`
  - fixed set of n threads operating off unbounded queue
  - new threads created to replace threads that crash
- `Executors.newCachedThreadPool()`
  - unlimited size thread pool, new threads created on demand
  - threads unused for 60 seconds terminated and removed
- `Executors.newScheduledThreadPool()`
  - and `Executors.newSingleThreadExecutor()`
  - allows tasks to be scheduled for one-off or repeated execution
- `Executors.newWorkStealingPool()`

# `Executors` – Class supplying `Executor` implementations

`Executors` factories actually produce *`ExecutorService`* implementations

- `ExecutorService` extends `Executor` interface
  - methods to manage termination and to track progress of submitted tasks:

```
interface ExecutorService extends Executor {
  void shutdown();
  boolean isShutdown();
  boolean isTerminated();
  <T> Future<T> submit(Callable<T> task);
  ...
}
```

# Callable<V> and Future<V>

- Callable<V> is like a Runnable that can return a value and throw exceptions

```
interface Callable<V> {
  V call() throws Exception
}
```

- Future<V> represents the progress of an asynchronous computation

```
interface Future<V> {
  boolean cancel(...);
  isCancelled();
  get(...);
  isDone();
}
```

# Concurrency Exercise 1 – Executor Framework

- Exercise documentation in lab docs folder
- Starting code in `concurrency.DirectoryTreeLister`

# Synchronizers

- `Semaphore`
- `CountDownLatch`
- Also
  - `CyclicBarrier`
  - `Phaser`
  - `Exchanger`
  - `FutureTask`

# Semaphore

- Manages a set of permits
  - used to control the number of activities accessing a resource
  - think of a nightclub bouncer!
- Acquire a permit with `acquire()` or `tryAcquire()`
- Release one or more with `release()`
- Other capabilities: `availablePermits(), reducePermits()`
- Basic construct: permits can be lost!

# Semaphore throttling task submission

```
public class BoundedExecutor {
    private final Executor exec;
    private final Semaphore semaphore;

    public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
    }

    public void submitTask(final Runnable command) throws InterruptedException {
        semaphore.acquire();
        try {
            exec.execute(new Runnable() {
                public void run() {
                    try {
                        command.run();
                    } finally {
                        semaphore.release();
                    }
                }
            });
        } catch (RejectedExecutionException e) {
            semaphore.release();
        }
    }
}
```

# `CountDownLatch`

- A *latch* (a one-way gate)
  - Allows one or more threads to wait until a set of events is complete
  - For example, players in a multi-player game can’t start until everyone is ready
  - Latch is initialised with a count
  - Threads that call `await()` are blocked until the latch opens
  - Latch opens after `countDown()` has been called enough
  - Subsequent calls to `await()` don’t block

# CountDownLatch — use in timing test

```
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task) throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) {}
                }
            };
            t.start();
        }
        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end - start;
    }
}
```

# Thread-Safe Collections

Picking a collection:

- If the collection is not shared, non-thread-safe collections are great!
  - `ArrayList, LinkedList, HashMap, TreeMap`
- If the collection is shared but access is not too frequent, use the standard synchronized collections
  - Benefit: ease of use, low memory footprint
- If the collection is shared and frequently accessed by multiple threads, use a concurrent collection
  - `ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentLinkedQueue,` etc.
  - Benefit: can be used by lots of threads without too much blocking
  - Disadvantage: might use a lot more memory

# Synchronized vs Concurrent Collections

- Synchronized collections
  - All methods synchronize on the same lock
  - They hold the lock for the entire operation, even time consuming ones - e.g. `List.contains`
- Concurrent collections can offer dramatic scalability improvements with little risk!
  - At the cost of using more memory
- Most important concurrent collections are `ConcurrentHashMap` and some `BlockingQueue` implementations

# ConcurrentHashMap

- Can be used as a drop-in thread-safe replacement for `HashMap`
- Implementations atomically execute methods that require check-then-act
  - `putIfAbsent`
  - `computeIfAbsent, computeIfPresent`
  - `merge`
  - `replace`

# BlockingQueue

- Blocking queues are under the hood of every producer-consumer app
  - that is, nearly all concurrent apps
- Blocking queues convey tasks – or data items to be processed – between the processes of a workflow system
  - smoothing out spikes in workload
- Consumers needing a work item should block until one is available
  - that is, the queue is non-empty
- Producers needing to provide a work item should wait until downstream processes are ready for one
  - that is, the queue is non-full

# BlockingQueue

```
interface BlockingQueue<E> {
  void put(E e);
  E take();
  boolean offer(E e, long timeout, TimeUnit unit);
  E poll(long timeout, TimeUnit unit);
}
```

- Most important implementations are `ArrayBlockingQueue`, `LinkedBlockingQueue`
  - Only LBQ can be unbounded
  - Both are FIFO structures
- Other implementations include `PriorityBlockingQueue`, `DelayQueue`, `SynchronousQueue`

# Optimistic Locking

- Conventional exclusive locking is *pessimistic*
  - Assumes that if you don't guard your valuables, they'll be "rearranged"
  - By someone else!
- With *optimistic* locking you hope everything will be all right
  - If it isn't, your operation failed to finish without interference
  - You can just try it again
- Optimistic locking relies on hardware support – often a compare-and-swap (CAS) instruction
- Can be much more efficient than pessimistic locking, especially when contention isn't high

# How CAS works

```
public class SimulatedCAS {
    private int value;
    public synchronized int get() { return value; }
    public synchronized int compareAndSwap(int expectedValue,
                                           int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue)
            value = newValue;
        return oldValue;
    }
}
```

```
SimulatedCAS cas = ...
valueToWorkOn = cas.get();
newValue = doLongRunningOperation(valueToWorkOn);
cas.compareAndSwap(valueToWorkOn, newValue))
```

# AtomicInteger

- So-called because its operations are atomic
- Pseudo-code for `addAndGet`:

```
public final int addAndGet(int delta) {
  for (;;) {
    int current = get();
    int next = current + delta;
    if (compareAndSwap(current, next) == current)
      return next;
  }
}
```

# Atomics as “Better Volatiles”

- Atomic variables are thread-safe without synchronisation
  - Values stored internally as volatile fields
  - Same visibility semantics
  - Little reason now to use volatile directly

# Types of Atomic classes

- The following types have atomics built in
  - `AtomicBoolean`
  - `AtomicInteger`
    - Use for `int`, `short`, `byte` and `float` (use `Float.floatToIntBits(float)`)
- `AtomicLong`
    - Use for `long` and `double`
- `AtomicReference`
- There are also atomic array classes
    - Necessary as you can never make values of an array volatile!
  - `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray`

# LongAdder

- Fast counter when we have high contention
  - Stripes the values into different cells to reduce contention

```
public class BankAccount {
  private final LongAdder balance = new LongAdder();
  public BankAccount(long balance) {
    this.balance.add(balance);
  }
  public void deposit(long amount) {
    balance.add(amount);
  }
  public void withdraw(long amount) {
    deposit(-amount);
  }
  public long getBalance() {
    return balance.longValue();
  }
}
```

# Concurrency Exercise 2

- Exercise documentation:
- Starting code: