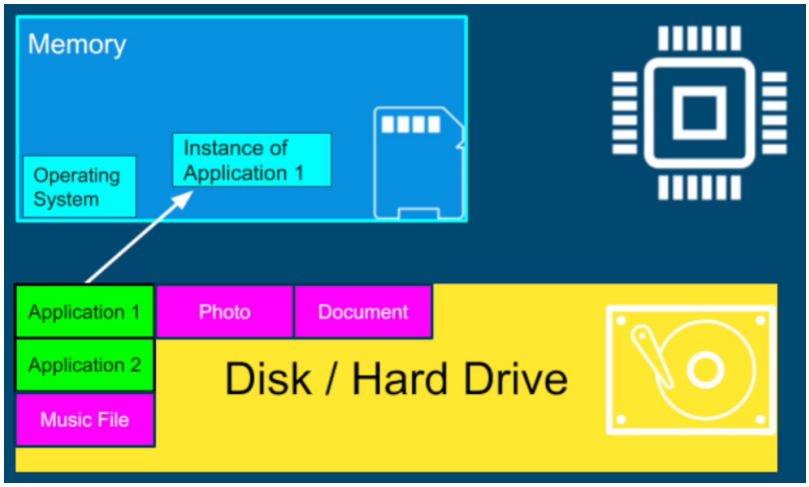
Concurrency - Multitasking

- We don't need multiple cores to achieve concurrency

Why do we need threads

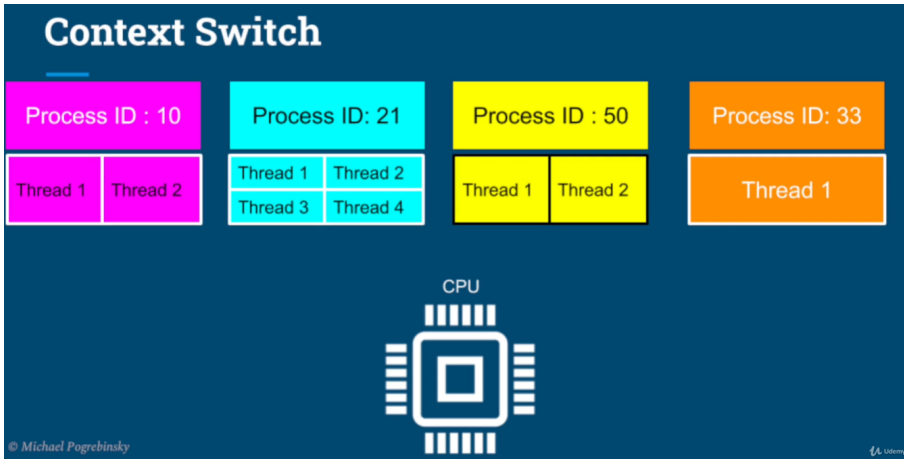
1. Responsiveness (concurrency)
2. Performance(parallelism). Complete complex tasks in a shorter time.

What Threads are and where they live ?



Threads independently contain two main things (1) Stack (2) Instruction pointer. Everything else like the [(A) heap (B) Code (C) processes open files (d) processes metadata] is shared.

Context Switch - when a process is stopped and a new process starts. A context switch is not cheap and is the price of multitasking (concurrency)



Section 2 - Thread Creation

When to multi-thread

- Tasks share a lot of data
- Threads are much faster to create and destroy

When to multi-process

- Security and stability are of higher importance
- Tasks are unrelated to each other

How does the Operating System design what thread to schedule?

- (●) The Operating System maintains a dynamic priority for each thread to prioritize interactive threads and to avoid starvation for any particular thread in the system.

How to create a thread in Java ?

(1) Implement the runnable interface and pass a new Thread object
(2) Extend Thread class and create an object of that class

Question 1:

What does this code do?

```
1  Thread thread = new Thread(new Runnable() {
2    @Override
3    public void run() {
4       System.out.println("Executing from a new thread");
5    }
6  });
```

- ( ) The code simply prints out `Executing from a new thread` to the screen
- ( ) The code starts a new thread and executes the code inside `run()`
- ( ) The code is creating a new thread which **if** started (by calling the `thread.start()`, will execute the code inside the `run()` method on a new thread.

Answer: C

Section 3 - Thread Coordination

<Thread stopping >

Why stop a thread ?

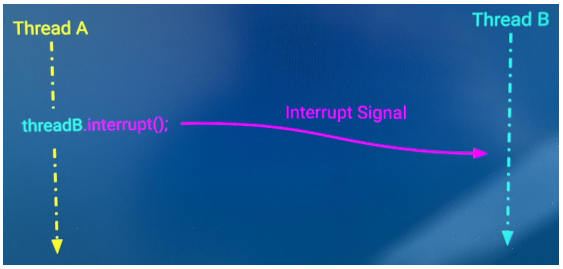
(1) Threads consume resources even when not doing anything. When a thread is running it is using cpu time.
(2) If a thread is misbehaving, we want to stop it.
(3) By default, the application will not stop as long as one thread is still running

How to stop a thread method 1: call Thread.interrupt()

Let Thread_A represent the main thread and Thread_B represent another task like sleep. If we call Thread_B.interrupt() before the main thread ends, Thread_B will stop.

When can we interrupt a thread ?

(1) If the threads code is handling the interrupt signal explicitly
(2) If the thread is executing a method that throws an InterruptedExecution



How to stop a thread method 2: set Daemon first before calling Thread.interrupt() later

Scenario 1 - background tasks that should not block our application from terminating. For example, say we have a text editor open with an auto save running on a background thread. We do not want the background autosave thread preventing us from closing our application.

Scenario 2 - code in a worker thread is not under our control and we do not want it to block our application from terminating. For example, a 3rd party dependency that is being called that does not handle the thread interrupt signal.

Question 1:

What does this code do?

```
1  Thread thread = new Thread(new Runnable() {
2    @Override
3    public void run() {
4       System.out.println("Executing from a new thread");
5    }
6  });
```

- ○ The code simply prints out `Executing from a new thread` to the screen
- ○ The code starts a new thread and executes the code inside `run()`
- ○ The code is creating a new thread which **if** started (by calling the `thread.start()`, will execute the code inside the `run()` method on a new thread.

Answer: C

<Thread joining>

Intro - Threads run independently. When multiple threads are running there is no way to determine who will finish first. Given thread A and thread B, they can finish in any order, finishing concurrently, or running in parallel. **But what happens when one thread depends on another ?** A naive solution would be to check if threadDepend checks on threadInpendent in a while loop. It's inefficient because threadDepend will burn CPU cycles checking if threadIndependent is done. Thread joining will be the more efficient way of handling this.

Section 4 - Performance Optimization

How is performance defined in different applications ?

(1) Latency. Ex.) speed trading system. The faster the transaction is the faster the app.
(2) Precision / Accuracy. Ex.) Framerate of a film being served is what user wants
(3) Throughput. Ex.) amount of data that can be fed into a ML model in a certain amount of time

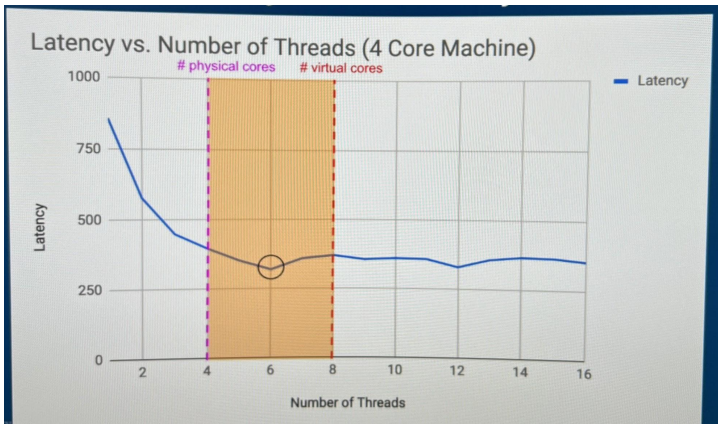
take away is that performance can be measured on a infinite number of characteristics

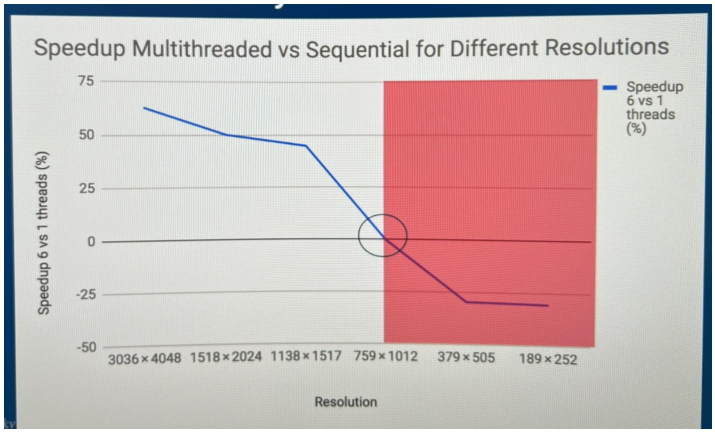
Latency - the time to completion of a task. Measured in time units. Low latency is good.
Throughput - the amount of tasks completed in a given period. Measured in tasks/time unit.

<***optimizing for latency***>

ex.) image processing program. We want to recolor some components (let's say white flowers) in our image (to purple flowers).

Method 1 - use a single thread to go from the top left corner to the bottom right.
Method 2 - use multiple threads. Separate the image into multiple vertical slices. Process each in parallel.



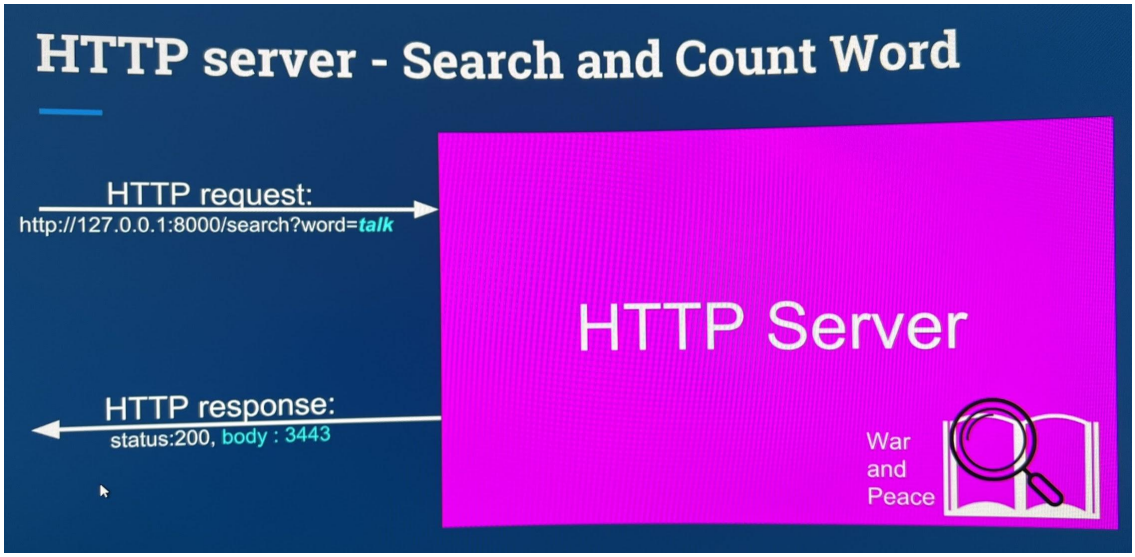


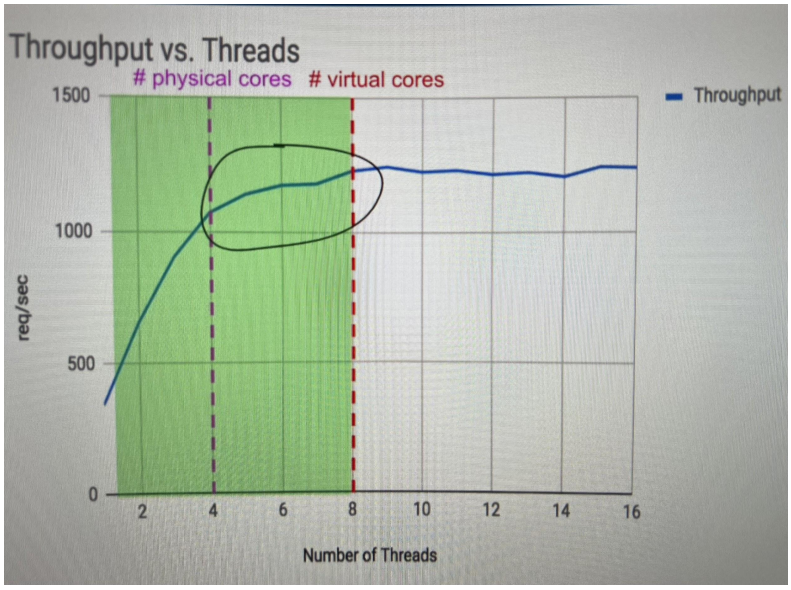
<***optimizing for throughput***>
Throughput - the # of tasks completed in a given period. Measured in tasks/time unit.

When to optimize for throughput ? Whenever given a program has concurrent tasks and we want to perform as many tasks as possible as fast as possible.

Thread Pooling - creating the threads once and reusing them for future tasks.

Example. Build an HTTP server. Input request will be word in book. Worker thread will search for word count in the book and return a response.





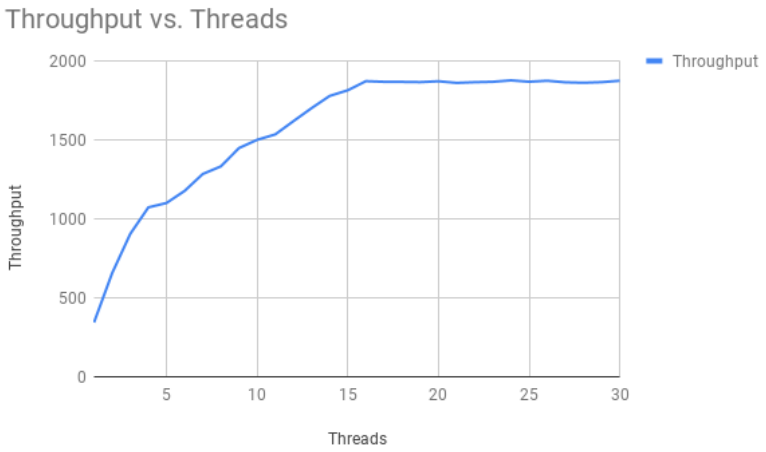
Question 1:

The following graph summarizes the throughput measurements made on an HTTP server, running on a single machine.

Handling of the requests is delegated to a fixed-size pool of threads.

Each request is handled by a single thread from the pool by performing a **non blocking** task, and sending an HTTP response to the user.

The measurements are showing the throughput of the server as a function of the size of the thread pool (number of threads in the pool).



What do you think is the number of cores on that particular machine?

Question 2:

We are running an HTTP server on a single machine.

Handling of the HTTP requests is delegated to a fixed-size pool of threads.

Each request is handled by a single thread from the pool by performing a **blocking** call to an external database which may take a **variable duration**, depending on many factors.

After the response comes from the database, the server thread sends an HTTP response to the user.

Assuming we have a 64 core machine.

What would be the optimal thread pool size to serve the HTTP request?

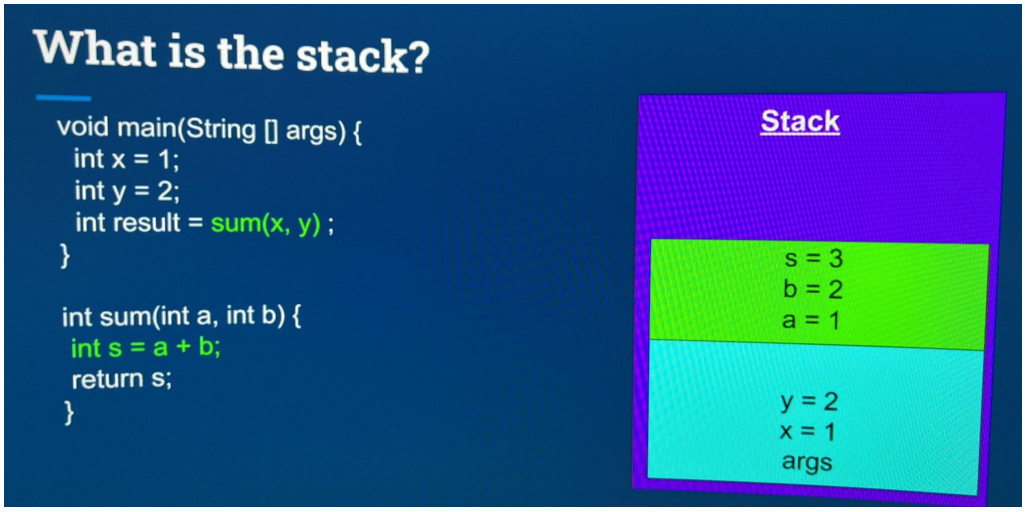
- ○ 16
- ○ 64
- ○ 128
- ◉ There is no way of knowing, since the tasks include a blocking call. But definitely more than 64

Answer 1: 16

Section 5: Data sharing between threads

Stack - memory region where (1) methods are called (2) arguments are passed (3) local variables are stored

State of each threads execution = stack + instruction pointer



Heap - shared memory region that belongs to the process (1) all objects (anything created with the new) (2) members of classes (3) static variables

REDO resource sharing lecture

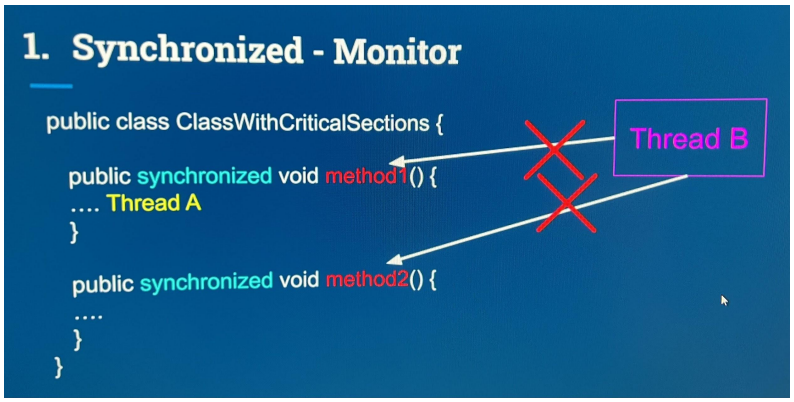
Section 6: The concurrency challenges and solutions

<***Critical Section & Synchronization***>

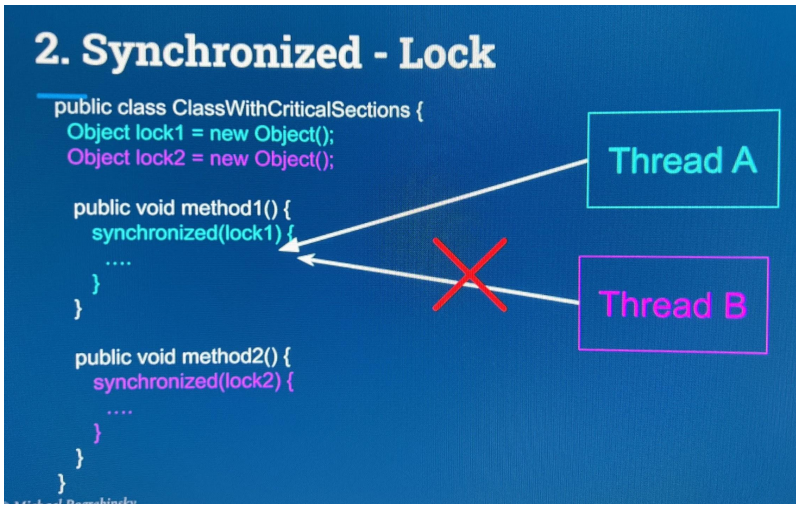
Critical Section - area of code that if two threads access will cause issues

Solutions (in Java)

1. **Synchronized - monitor.**
2. **Synchronized - lock.** Preferred method because we can lock only the critical sections instead of an entire function. More flexible and granular but also more verbose.



If even one method is being executed, all other synchronized methods of the same object become inaccessible to other threads



<***Atomic operations, volatile & metrics*** >

Atomic Operations

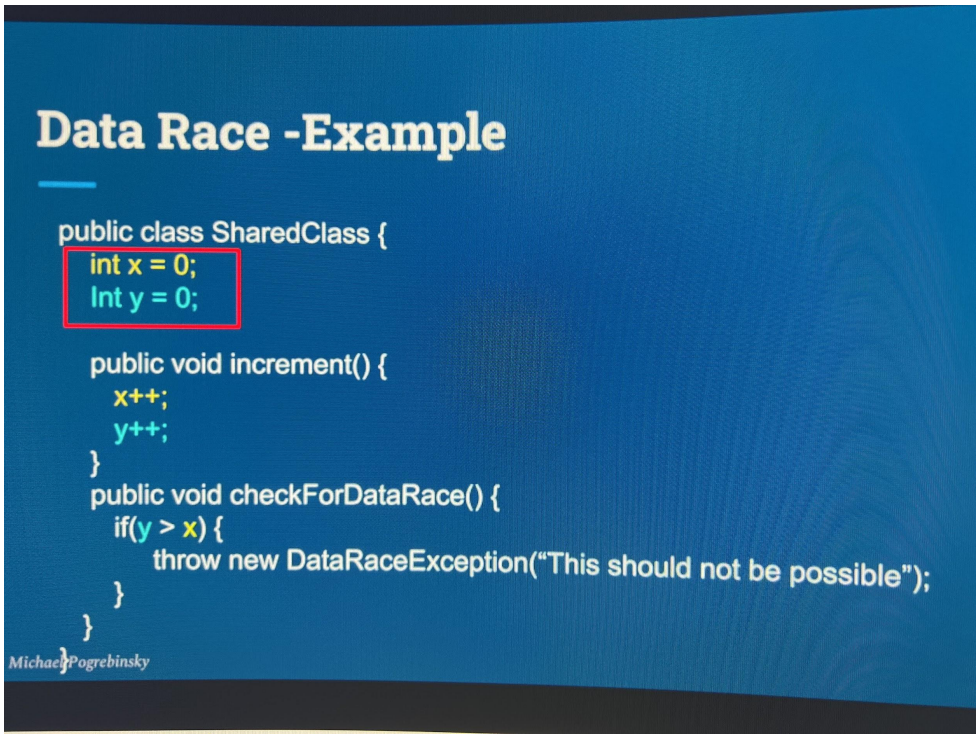
- All reference assignments
- All assignments to primitive types are safe EXCEPT long and double. Must use volatile keyword before defining double or long

<***Race Conditions & Data Races*** >

Race Condition require:

(1) Multiple threads sharing a resource
(2) At least one thread modifying the resources

Data Race Problem -
Compiler and CPU may execute the instructions out of order to optimize performance and utilization.



Data Race Solutions - Establish a happens before semantics by one of these methods

(1) Synchronization of methods which modify shared variables. But has a performance penalty.
(2) Declaration of shared variables with the volatile keyword. Has to be used carefully.

simply, only allow one thread into the critical section

<***Locking Strategies and Deadlocks*** >
Fine grained locking vs coarse grained locking.

Coarse grained locking - We only have a single thread to worry about. Makes life simple but not always the best strategy.

Fine grain locking - better parallelism but leads to deadlocking.

Section 7: Advanced Locking

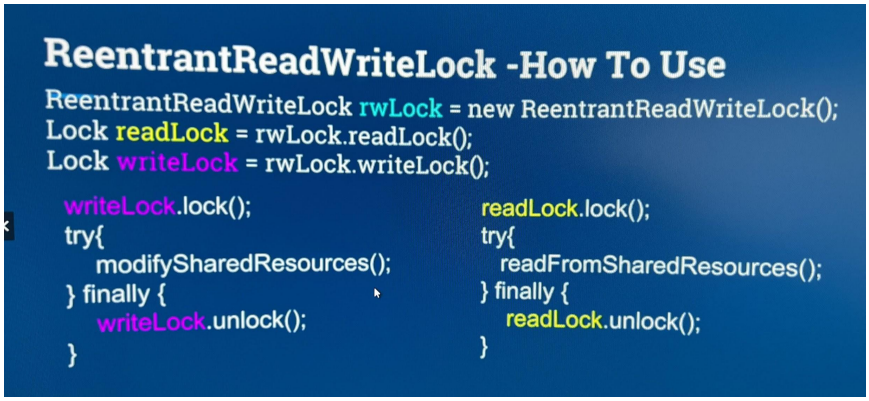
<***tryLock***>
An issue that arises from locking is that if an exception is thrown while the thread is locked, we may never get to unlock that thread. To prevent this from happening, we create a try finally block with the lock and unlock code.

<***LockInterruptibly***>
Use cases
(1) Watchdog for deadlock detection and recovery
(2) Waking up threads to do clean and close the application

Real time applications where suspending a thread on a lock() method is unacceptable.
(1) video/image processing
(2) High speed/low latency trading systems
(3) User interface applications

<***Read Write Lock & Database implementation***>

- Allows multiple threads to get into a read lock.
- If a write lock is acquired, no thread can acquire a read lock
- It at least one thread acquired a read lock, no thread can acquire a write lock



Faster way to lock resources!

Section 8: Inter-Thread Communication

<***Semaphore***>
A semaphore can be used to restrict the number of “users” to a particular resource or a group of resources. A lock can be thought of as a special case of a semaphore in which only one is allowed.

Difference from lock
(1) Semaphore doesn't have a notion of owner thread
(2) Many threads can acquire a permit
(3) The same thread acquire the semaphore multiple times

```
public static class Barrier {
    private final int numberOfWorkers;
    private Semaphore semaphore = new Semaphore( //** blank 1 **/);
    private int counter = //** blank 2 **/;
    private Lock lock = new ReentrantLock();

    public Barrier(int numberOfWorkers) {
        this.numberOfWorkers = numberOfWorkers;
    }

    public void barrier() {
        lock.lock();
        boolean isLastWorker = false;
        try {
            counter++;

            if (counter == numberOfWorkers) {
                isLastWorker = true;
            }
        } finally {
            lock.unlock();
        }

        if (isLastWorker) {
            semaphore.release(//** blank 3 **/);
        } else {
            try {
                semaphore.acquire();
            } catch (InterruptedException e) {
            }
        }
    }
```

```
blank 1 : 0
blank 2 : 0
blank 3 : numberOfWorkers
```

```
blank 1 : 0
blank 2 : 0
blank 3 : numberOfWorkers - 1
```

```
blank 1 : numberOfWorkers
blank 2 : 0
blank 3 : numberOfWorkers
```

we initialize the semaphore to 0, to make sure every thread that tries to acquire the semaphore gets blocked. And the last thread to get to the barrier, releases the semaphore numberOfWorkers - 1 since "numberOfWorkers - 1" threads are blocked on the semaphore.

<***Condition Variable***>

Functions

(1) Wait. causes the current thread to wait until another thread wakes it up.

(2) Notify. Wakes up a single thread.

(3) notify all. Wakes up all threads.

Good example in lecture 28

Section 9: Lock-Free Algorithms, Data Structures & Techniques

Problems with multithreading

- Deadlocks are generally unrecoverable
- Slow critical sections
- Priority inversion

AtomicInteger. Does not require methods to have synchronized keyword.

Building our own lock free data structure

All you do is wrap the type in an AtomicReference. You then use the compareAndSet operation to change the values

```
1   public class Metric {
2       private AtomicLong count = new AtomicLong(0);
3       private AtomicLong sum = new AtomicLong(0);
4
5       public void addSample(long sample) {
6           sum.addAndGet(sample);
7           count.incrementAndGet();
8       }
9
10      public double getAverage() {
11          double average = (double)sum.get()/count.get();
12          reset();
13          return average;
14      }
15
16      private void reset() {
17          count.set(0);
18          sum.set(0);
19      }
20  }
```

Is the class thread safe now?

- ○ Yes, now it's thread-safe
- ◉ No, it is still not thread safe

Some operations became thread safe, like sum.addAndGet(sample) or count.incrementAndGet();. However when we read or modify 2 variables, even if each individual operation on each variable is atomic, the aggregate operation is no