## Motivation for using threads:

A. Hide latency: some processes can wait data / sleep while others compute

   E.g. Network communication, file watcher, UI operation.

B. Use computational power available to solve faster a problem

## Parallelism at Hardware and OS level

a) Instruction level parallelism & Remember IP, call stack, assembler stuff
b) Difference between a process and a thread.
c) How different OS schedulers work :
    a. real time  OS vs windows case (why time sharing on windows ? what if single CPU code ?)
    b. Priorities, interrupt
    c. Check SimpleThread app code to see the way you can work with threads in Java Explanation on how instructions are executed on a multi – single CPU.

## Race conditions & Synchronization primitives

A. **Monitors** - Check the RaceStuff.java example
    a. Understand code and how shared memory is passed (how Counter object is shared between the threads)
    b. Run program and check counter. Try to remove "synchronized" word from uncommented Counter class to see the results.
    c. Switch to the commented Counter function to see a general synchronization method that doesn't affect entire function.

       Discussion about granularity level synchronization and parallelism, or “synchronized” per function vs “synchronized” per object.

B. **Waint/Notify/NotifyAll |  Barriers and Semaphores**
    a. Discuss about internal implementation and check the ProducerConsumer.java
    b. Check the BarrierExample.java then BarrierImpl.java solution for a hand-written implementation
    c. Semaphore and a hand-written implementation (SemaphoreHandWritten.txt)

**C. Locks**

**a.** Why Locks? Monitors (synchronized objects) present several disadvantages. The main one: you must work **on the same instance.** A better solution: Locks similar with mutex or critical sections in other languages.

**b.** Lock vs TryLock

**Lock**:

```
Lock l = ...;
   l.lock();
   try {
      // access the resource protected by this lock
   } finally {
      l.unlock();
   }
```

**TryLock**:

```
    Lock lock = ...;
    if (lock.tryLock()) {
       try {
          // manipulate protected state
       } finally {
          lock.unlock();
       }
    } else {
       // perform alternative actions
    }
```

c. Reentrant locks:

```
Lock l = new ReentrantLock()
```

- ➔ Adds fairness (check constructor). But this can get performance issues. Why ?
- ➔ l.lock with timeout

## General issues when working with threads: Deadlocks, Starvation, Priority Inversion

A. Deadlock: http://www.tutorialspoint.com/java/java_thread_deadlock.htm

App: See Deadlock.java (problem ) and Deadlock_solved.java (solution). If you run Deadlock.java a number of times you'll catch the deadlock for sure ☺.

How to debug in netbeans: Press Debug / Project then hit Pause button when it's blocked. Check the callstack (see picture DebugDeadlock.png in main folder).

- Starvation and livelock: https://avaldes.com/java-thread-starvation-livelock-with-examples/

- priority inversion: http://www.drdobbs.com/jvm/what-is-priority-inversion-and-how-do-yo/230600008

**D. Atomics**

A reference:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html

Check the example implemented in Histogram.java. Request:

* Generate an array of random numbers as described above
* Spawn P worker threads to solve this problem.
    - Idea 1: use a single global result histogram and atomic variables.
    - Idea 2: use local histograms (per worker) and combine them all in the end
* Measure execution time for serial vs parallel execution
* Try to use a Reduce algorithm:
https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html

# Applications

1) **Dinning Philosophers (solution in DinningPh.java)**

- N philosophers at a circular table, executes 2 operations in order: THINK then EAT.

- there are N forks, one between each pair of consecutive philosophers.

- to EAT a ph needs both forks (left and right fork. If ph has index 0 then he needs fork 0 and fork

TODO: Simulate 10 cycles of THINK+EAT for each ph.

Parallelism issues:

- need to ensure mutual exclusion of forks

- Can get a deadlock if all ph. get the fork on their left (one one could get the fork on the right anymore)

Implementation suggestions:

- Each ph. is a thread (class)

- To simulate THINK and EAT consider Thread.Sleep(random time) in interval [0,100] ms.

- Need a sempahore to control how many ph have started the cycle. Initially it should have value N-1 to avoid all N ph enter in cycle at once.

- Need binary semaphores for each fork.

- Have a class Table with static members that can be accessed by ph class: N, all semaphores.

**2) Readers and writers**

-> We have a shared object (let's say a database).

-> There are 2 actors: Readers - read only shared object and Writers - read/write modify the shared object

Conditions

-> Multiple readers can access the shared object at the same time

-> Only one writer at any point. When a writer gets access NO READERS should be active.

Obs: Use sempahores for impl.

Possible operations for actors (calls guaranteed in correct order):

a) Writers:

```
open(WRITE);
// read content
close(WRITE);
```

b) Readers:

```
open(READ);
// write content
close(READ);
```

Pair (numReaders, numWriters) can have values (0,0), (0,1), (n,0)

A state is represented by (numReaders, numWriters, semaphore) where sempahore is 1 if no one is accessing the resource, 0 otherwise.

Possible states: (0,0,1) , (1,0,0), (0,n,0).

Initial state is (0,0,1).

Implement 2 solutions:

A. Favor readers (solution in ReadersAndWriters)

B. Favor writers (solution in ReadersAndWriters_2)

C. Talk about starvation

4. **Merge app**

- Consider 2 arrays A, B with dimensions lA and lB.

- Target: if C = A U B, at the end of the program A should contain the smallest values while B the highest ones.

- The only acceptable operation is swapping a value between A and B.

Implement using a barrier. Check folder BarrierMerge for solution.

**5) ReadersAndWriters (no favors, simulation using a queue)**

(solution for a) in ReadersAndWriters_3, for b) in ReadersAndWriters_4)

a) Using the implementation for readers, writers and main from ReadersAndWriters folder, change the implementation of

Database class such that to be fair for both readers and writers.

Idea with "polling" thread:

- Create a queue of persons requesting access to the Database (each item would contain type of access - READ / WRITE - and an empty semaphore - 0 value initially)

- When a person wants access, it will get added to the queue and acquire the semaphore (which being 0, will block there).

- Database must be derived from a thread and have a run function executed by a separate thread.

This thread will always get the first element from the queue and call semaphore.release() (which will awake the waiting open operation)

if the operation is vaid: Eg. if READ then we must have no writers, if WRITE we must have no readers and writers, so just keep the counts..

- you will also need an internal lock for some operations...

b) Improve the algorithm implemented in a to use condition.

Lock r = new ReetrantLock(true);

Condition cond1 = r.newCondition();

// You can create how many condition you want...

When cond1.wait() is called, it will wait for a cond1.signal() or cond1.signalAll() to be called.

Notice: on wait() the lock r is released. Also, after there is a signal and we pass the wait() we re-aquire the lock automatically.

## Homework

1. Advanced Producer consumer problem:

   App 1: Implement producer - consumer paradigm
   - single item to produce / consume:
   http://www.tutorialspoint.com/javaexamples/thread_procon.htm
   - using an already syncronized queue (BlockingQueue), multiple consumers & items:

   https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

   Q: how do you think is BlockingQueue implemented ?
   See other concurrent collections here:
   https://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html

2. Task vs thread parallelism

   http://tutorials.jenkov.com/java-util-concurrent/executorservice.html
   http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor

3. Active object pattern

Original doc: http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf
Java: http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095

Implement an example !