More tools for concurrent programming

Concurrent programming

- Concurrent programming = programming with multiple threads or processes
- As we have seen, it's easy to make mistakes. Therefore, computer scientists have developed "standard solutions" for many typical situations
- Note: All the examples we see on the next slides could be also implemented with `synchronized`/wait()/notify()

Read/Write locks

- A ReadWriteLock is a lock where multiple threads can read data at the same time if there is no thread writing data

```
class ReadWriteDictionary<E> {
    private final HashMap<String, E> m = new HashMap<String, E>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public E get(String key) {
        r.lock();
        try {
            return m.get(key);
        }
        finally {
          r.unlock();
        }
    }

    public E put(String key, E value) {
        w.lock();
        try {
            return m.put(key, value);
        }
        finally {
            w.unlock();
        }
    }
}
```

To read from the hashmap, the thread needs a read-lock

Many threads are allowed to get a read-lock at the same time.
However, if one thread gets a write-lock, all other threads must wait.

To write to the hashmap, the threads needs a write-lock. The thread has to wait until all read-locks are released.

Semaphore

- A Semaphore is like a lock. However, n threads are allowed to enter a semaphore at the same time. ($n = 1 \rightarrow$ normal lock)

```
public class SemaphoreTest {
    private LinkedList<Hammer> hammers = new LinkedList<Hammer>(
                            Arrays.asList(new Hammer(), new Hammer(), new Hammer()));
    private final Semaphore available = new Semaphore(3, true);

    public Hammer getHammer() throws InterruptedException {
        available.acquire();
        synchronized(hammers) {
            return hammers.remove();
        }
    }

    public void giveBackHammer(Hammer hammer) {
        synchronized(hammers) {
            hammers.add(hammer);
        }
        available.release();
    }
}
```

A semaphore with three permits ($n = 3$)

Ask a permit from the semaphore. If there is no permit available, the thread waits

Give permit back

Barriers

- Let's imagine you have 1000 ideas for a Christmas present for your friend
- You only want to buy one present. You don't want to necessarily buy the cheapest one, but it must cost less than 100 Euros.
  - For each idea X, you can buy it on Amazon or on eBay. Of course, you want to buy at the shop where X is the cheapest.
- Idea for the implementation:
  1. Take first idea. Check prices on Amazon and on eBay. Stop the search and buy it if the price is less than 100 Euros.
  2. Take second idea. Check prices on ....
  3. ...
- To be fast, we want to do the search on Amazon and eBay in parallel

Barriers (2)

- Algorithm:

  For each present idea $X \in \{present\ ideas\}$:

  1. Thread t1 finds price A of X on Amazon
  2. Thread t2 finds price B of X on eBay
  3. Wait until both threads have found the price
  4. Stop if $\min(A, B) \leq 100$

  (Steps 1 and 2: in parallel)

- Step 3 is called a *barrier*: The algorithm can only continue if both threads have finished
- This could be implemented with a for-loop and Futures. But then we have to create new futures for every present idea.
  - An algorithm where we have to repeatedly wait for n threads to finish can be implemented with java.util.concurrent.CyclicBarrier
  - (If we want to wait only once for n threads, then we can use java.util.concurrent.CountDownLatch)

Implementation with CyclicBarrier

```
public class ChristmasPresents {
    CyclicBarrier barrier;
    int amazonPrice;
    int ebayPrice;
    boolean presentFound = false;
    int i = 0;
    String[] presentIdeas=new String[] { "Idea 1", "Idea 2","..."};

    class AmazonSearch implements Runnable {  // similar code for EbaySearch
        public void run() {
            while(!presentFound) {
                amazonPrice = findAmazonPrice(presentIdeas[i]);
                try {
                    barrier.await();
                } catch (InterruptedException ex) {
                    return;
                } catch (BrokenBarrierException ex) {
                    return;
                }
            }
        }
    }

    public void findBestPresent() {
        barrier = new CyclicBarrier(2, ()->{
            if(amazonPrice<=100 || ebayPrice<=100)
                presentFound=true;
            else
                i++;  // check next present
        });
        Thread t1=new Thread(new AmazonSearch());
        Thread t2=new Thread(new EbaySearch());
        t1.start(); t2.start();
        t1.join(); t2.join();
    }
}
```

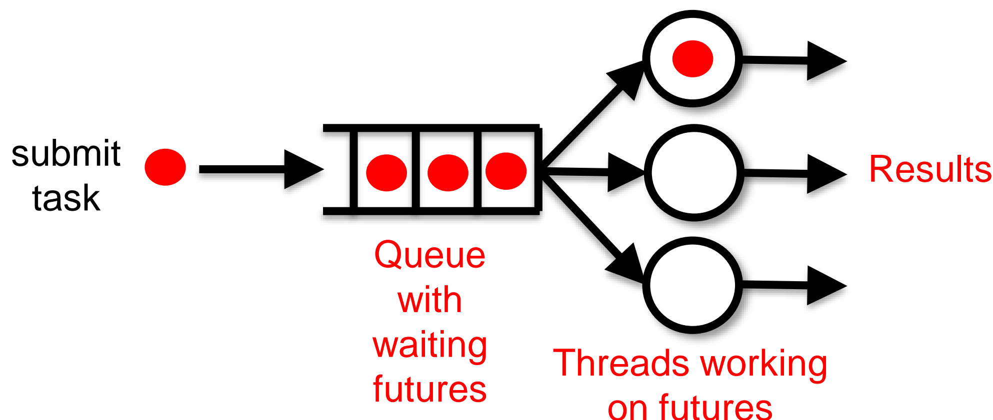
await() blocks the thread until both threads have called await()

This is a barrier to synchronize two threads

This lambda expression is executed everytime both threads have called await().

Futures and Threadpools

- We have already seen how to use Futures and Threadpools
- Internally, this is implemented in the Java library like this:

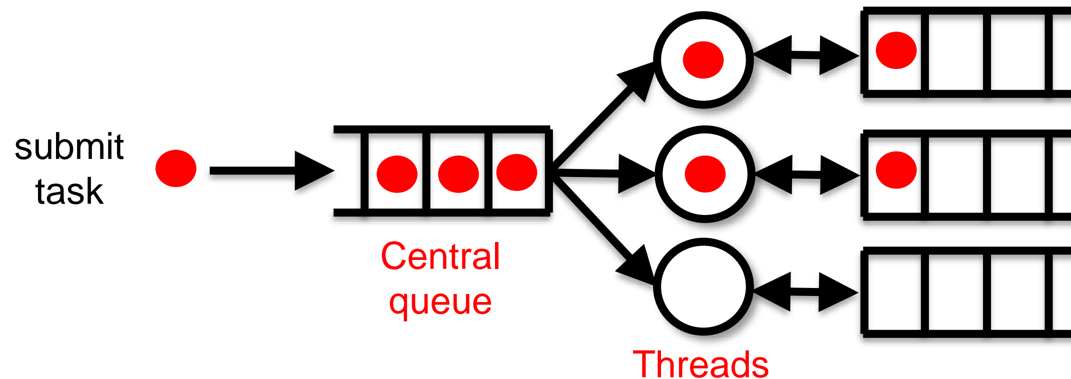


- This is easy to implement but not very efficient if you have an algorithm where the futures can create new futures
  - Example: A recursive sorting algorithm like Quicksort. A future divides the list in two and creates two futures for each part of the list

→ Lot of waiting for synchronization at the add/remove methods of the queue

java.util.concurrent.ForkJoinPool

- In a ForkJoinPool, every thread in the threadpool has its own queue



- There is still a central queue
- When a thread creates a new task it is placed in its own queue
- When a threads looks for a task to execute it will
  - first, look in its own queue
  - second, look in the queues of other threads (this is called “stealing”)
  - finally, look in the central queue

How to use ForkJoinPool

- Tasks in a ForkJoinPool are of type ForkJoinTask<R> where R is the type of the result of computation (ForkJoinTask is a subclass of Future)
- In the exercise in Java, you will work with a special subclass of ForkJoinTask: the RecursiveAction class. It's a task without result (void)
- Example:
  - We want to increment all elements of an array by 1
  - We first create a task to increment all elements from 0 to length-1 and give it to the ForkJoinPool:

```
int[] array = new int[]{ 1,2,3,4,5,6,7,8,9,10,11,12};

ForkJoinPool pool = new ForkJoinPool(3);

pool.invoke(new IncrementTask(array,0,array.length));
```

a pool with three threads

invoke() submits the task to the pool and waits until the task has finished

How to use ForkJoinPool (2)

- Here is the implementation of the task to increment the elements [lo..hi] of an array:

```
class IncrementTask extends RecursiveAction {
    final int[] array;
    final int lo, hi;

    IncrementTask(int[] array, int lo, int hi) {
        this.array = array; this.lo = lo; this.hi = hi;
    }

    @Override
    public void compute() {
        if (hi - lo < 5) {
            for (int i = lo; i < hi; ++i)
                array[i]++;
        }
        else {
            int mid = (lo + hi)/2;
            invokeAll(new IncrementTask(array, lo, mid),
                      new IncrementTask(array, mid, hi));
        }
    }
}
```

the compute() method defines the work to do for the RecursiveAction

If the task is very small (less than 5 elements to increment), we do it here

invokeAll() submits new tasks and waits until they have finished

Here we create two new tasks for the elements [lo,mid] and [mid,hi]