PProg: Important things to know

Creating a Thread

```
public class Useless extends Thread {
    int i;
    Useless (int i) {
        this.i = i;
    }
    public void run() {
        System.out.println("Thread says hi" + i);
        System.out.println("Thread says bye" + i);
    }
}


public class M {
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            Thread t = new Useless(i+ 1);
            t.start(); //Important: you cannot use t.run() --> t.run doesn't
create a new Thread
        }
    }
}
```

Important: you can only create a Thread, when you use start(). Thread, run method has no return value and no argument

Joining threads:

```
public class Useless extends Thread {
    int i;
    Useless (int i) {
        this.i = i;
    }
    public void run() {
        System.out.println("Thread says hi" + i);
        System.out.println("Thread says bye" + i);
    }
}


public class M {
    public static void main(String[] args) {
        Thread[] threads = new Thread[20];
```

```
        for (int i = 0; i < 20; i++) {
            Thread t = new Useless(i+ 1);
            t.start();
            threads[i] = t;
        }
        for (int i = 0; i < 20; i++) {
            try { //need catchblock around join
                threads[i].join();
            } catch (InterruptedException e) {
               //Some catch block
            }
        }
        System.out.println("All done.");
    }
}
```

With "join", you wait until every thread is done

Thread states

If we want to be able to talk about the effects of different thread operations, we need some notion of thread states. In short, a Java thread typically goes through the following states:

- **Non-Existing:** Before the thread is created, this is where it is. We don't know too much about this place, as it's not actually on our plane of reality, but it's somewhere out there.
- **New:** Once the Thread object is created, the thread enters the new state.
- **Runnable:** Once we call start() on the new thread object, it becomes eligible for execution and the system can start scheduling the thread as it wishes.
- **Blocked:** When the thread attempts to acquire a lock, it goes into a blocked state until it has obtained the lock, upon which it returns to a runnable state. In addition, calling the join() method will also transfer a thread into a blocked state.
- **Waiting:** The thread can call wait() to go into a waiting state. It'll return to a runnable state once another thread calls notify() or notifyAll() and the thread is removed from the waiting queue.
- **Terminated:** At any point during execution we can use interrupt() to signal the thread to stop its execution. It will then transfer to a terminated state. Note that when the thread is in a runnable state, it needs to check whether its interrupted flag is set itself, it won't transfer to the terminated state automatically. Of course, exiting the run method is equivalent to entering a terminated state. Once the garbage collector realizes that the thread has been terminated and is no longer reachable, it will garbage collect the thread and return it to a non-existing state, completing the cycle

Data Races:

A data race is a specific kind of race condition that is better described as a **simultaneous access error**, although nobody uses that term. There are two kinds of data races:

- When one thread might read an object field at the same moment that another thread writes the same field.
- When one thread might write an object field at the same moment that another thread also writes the same field.

```
class C {
    private int x = 0;
    private int y = 0;

    void f() {
        x = 1; //line A
        y = 1; //line B
    }
    void g() {
        int a = y; //line C
        int b = x; //line D
        assert (b >= a);
    }
}
```

Code has data races, but it doesn't occur --> proof by contradiction

Important to remember:

Speedup:

$S_p := \frac{T_1}{T_P}$

Where T_1 is the sequentail time (Time with one processor) and T_P with the time with P processors.

Reasons why program is nevertheless slower: _ *Additional overheads caused by inter-thread dependencies, creating threads, communicating between them and memory-hierarchy issues can greatly limit the speedup we gain from adding more processors.*

Amdahl:

- Fixed workload and upper bound on the speedup achievable when increasing the number of processors at our disposal.

Let f denote the non-parallelizable, serial fraction of the total work done in a program and P the number of processors at our disposal. Then, the following inequality holds:

$S_P <= \frac{1}{f + \frac{1 - f}{P}}$

If P is infinity then:

$S_\infty <= \frac{1}{f}$

How to derive it:

$T = T_s + T_p$ Where as T is the total time, T_s the sequential time and T_p the parallel time. $T_p = \frac{T_s}{p}$ If you have more than one processor you can rewrite the function to:

$T = T_s + \frac{T_p}{p}$

$Sp = \frac{W_{seq} + W_{par}}{W_{seq} + \frac{W_{par}}{p}}$

We know that $W_{seq} + W_{par} = 1$ So we can rewrite the function to:

$\frac{1}{W_{seq} + \frac{W_{par}}{p}} = \frac{1}{f + {\frac{1-f}{p}}}$

Gustafson

- We increase the problem size as we improve the resources at our disposal. We consider the time interval to be fixed and look at the problem size.

Let f denote the non-parallelizable, serial fraction of the total work done in the program and P the number of processors at our disposal. Then, we get:

$S_P = f + P(1 -f) = P - f(P - 1)$

Workspan

$T_x <= T_\infty + \frac{T_1 - T_\infty} {x}$

In a graph:

Work: all jobs summed up Span: Longest critical path

Divide and Conquer with ExecutorService

```
class MaxTask implements Callable  {
    //Runnable would work as well, but then you don't have a return value,
callable does have one
    int l;
    int h;
    int[] arr;
    ExecutorService ex;

    üublioc MaxTask(ExecutorService ex, int lo, int hi, int[] arr) {
       this...
    }
    public Integer call() throws Exception {
        //Check base case
        int size = h - l;
        if (size == 1) {
            return arr[l];
        } //split work
        int mid = size / 2;
        MaxTask m1 = new MaxTask(ex, l, l + mid, arr);
        MaxTask m2 = new MaxTask(ex, l + mid, h, arr);
        //Start subtasks
        Future<Integer> f1 = ex.submit(m1);
        Future<Integer> f2 = ex.submit(m2);
        //Combine results
        try {
            return Math.max(f1.get(), f2.get());
        } catch(Exception e) {
            return 0;
```

```
        }
    }
}

public static void main(String[] args)
int[] arr = new int[] {15, 7, 9, 8, 4, 22, 42, 13};
ExecutorService ex = Executors.newFixedThreadPool(8); //Attention, has to be
minimum number of thread which are needed --> otherwise you have an endless loop
MaxTask top = new MaxTask(ex, 0, arr.length, arr);
Future<Integer> max = ex.submit(top);
try {
    System.out.println(max.get());
} catch(Exception e){
    //somethinbg
}
ex.shutdown();
```

To avoid "knowing" how many threads you need, you can use Recursive Task:

```
class MaxForkJoin extends RecursiveTask<Integer>  {
    int l;
    int h;
    int[] arr;

    public MaxForkJoin(int lo, int hi, int[] arr) {
       this...
    }
    public Integer compute() {
        //Check base case
        int size = h - l;
        if (size == 1) {
            return arr[l];
        } //split work
        int mid = size / 2;
        MaxForkJoin m1 = new MaxForkJoin(l, l + mid, arr);
        MaxForkJoin m2 = new MaxForkJoin(l + mid, h, arr);
        //Run subtasks
        m1.fork();
        int max2 = m2.compute();
        int max1 = m1.join();
        //Combine results
        return Math.max(f1.get(), f2.get());
    }
}

public static void main(String[] args)
int[] arr = new int[] {15, 7, 9, 8, 4, 22, 42, 13};
MaxForkJoin top = new MaxTask(0, arr.length, arr);
ForkJoinPool jfp = new ForkJoinPool();
int res = fjp.invoke(tp);
System.out.println(res);
```

Note the following similarities: We use the library as follows:

- Instead of extending Thread, we extend RecursiveTask (with return value) or RecursiveAction (without return value)
- Instead of overriding run, we override compute
- Instead of calling start, we call fork
- Instead of a topmost call to run, we create a ForkJoinPool and call invoke Also, note that in the case of RecursiveTask<T>, join now returns a result.

Throughput

$Throughput \approx \frac{1}{max(computationtime(stages))}$

Latency

Time to perform a single computation, including wait time resulting from resource dependencies.

A pipeline is **balanced** if the latency remains constant over time.

MPI

Synchronous, asynchronous, blocking, non-blocking

- Synchronous + blocking: try to call somebody until he answers.
- Synchronous + non-blocking: try to call, if the other person does not pick up I do something else.
- Asynchronous + blocking: wait until your crush texts you back.
- Asynchronous + non-blocking: send an E-Mail and continue working until you get a response.

In the actor model, messages are sent in an **asynchronous, non-blocking fashion**. --> The sender places the message into the buffer of the receiver and continues execution. In contrast, when the sender sends **syncrhonous-messages**, it blocks until the message has been received.

MPI collects processes into groups, where each group can have multiple **colors**. A group paired with its color uniquely identifies a communicator. Initially, all processes are collected in the same group and communicator MPI_- COMM_WORLD. Within each communicator, a process is assigned a unique identifier, called the **rank**.

```
public void Send(
    Object buf, //Ptr to data to be sent
    int offset ,
    int count, //number of items to be sent
    Datatype datatype, //datatype of items
    int dest, //destination process id
    int tag, //data id tag
    ) ;

public void Recv(
    Object buf,
    int offset ,
    int count, // Number of items to be received
    Datatype datatype, // Datatype of items
```

```
    int dest, // Source process id
    int tag, //Data id tag
    ) ;
```

Transactional Memory

Definition 3.7.1 (Transactional Memory) Transactional Memory is a programming model whereby loads and stores on a particular thread can be grouped into transactions. The read set and write set of a transaction are the set of addresses read from and written to respectively during the transaction. A data conflict occurs in a transaction if another processor reads or writes a value from the transaction's write set, or writes to an address in the transaction's read set. Data conflicts cause the transaction to abort, and all instructions executed since the start of the transaction (and all changes to the write set) to be discarded.

Transactions run in *isolation*: while a transaction is running, effects from other transactions are not observed. A good analogy is the one of a snapshot: transactional memory works as if transaction takes a snapshot of the global state when it begins and then operates on that snapshot.

Important Code

Barrier

```
public class Barrier {
  private Semaphore mutex;
  private Semaphore barrier1;
  private Semaphore barrier2;

  private volatile int count = 0;
  private final int n;

  Barrier(int n) {
    mutex = new Semaphore(1);
    barrier1 = new Semaphore(0);
    barrier2 = new Semaphore(1);

    this.count = 0;
    this.n = n;
  }

  void await() throws InterruptedException {
    mutex.acquire();
    ++count;
    if (count == n) {
      barrier2.acquire();
      barrier1.release();
    }
    mutex.release();

    barrier1.acquire();
    barrier1.release();
```

```
    mutex.acquire();
    --count;
    if (count == 0) {
      barrier1.acquire();
      barrier2.release();
    }
    mutex.release();

    barrier2.acquire();
    barrier2.release();
  }
}
```

Semaphore

```
public class Semaphore {
  private volatile int count;
  private Object monitor = new Object();

  public Semaphore(int count) {
    this.count = count;
  }

  public void acquire() throws InterruptedException {
    synchronized(monitor) {
      while (count <= 0)
        monitor.wait();
      --count;
    }
  }

  public void release() {
    synchronized(monitor) {
      ++count;
      monitor.notify();
    }
  }
}
```

PetersonLock

```
class PetersonLock{
    volatile boolean flag[] = new boolean[2];
    // Note: the volatile keyword refers to the reference, not the array contents
    // This example may still work in practice
    // It is recommended to instead use Java's AtomicInteger and
AtomicIntegerArray
    volatile int victim;
```

```
    public void Acquire(int id) {
        flag[id] = true;
        victim = id;
        while (flag[1-id] && victim == id);
        }

    public void Release(int id){
        flag[id] = false;
    }
}
```

Filterlock

```
int[] level(#threads), int[] victim(#threads)
lock(me) {
for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (exists(k != me): level[k] >= i && victim[i] == m){};
    }
}
unlock(me) {
level[me] = 0;
}
```

Filterlock is not fair

BakeryLock

```
class BakeryLock{
    AtomicIntegerArray flag; // there is no AtomicBooleanArray
    AtomicIntegerArray label;
    final int n;
    BakeryLock(int n) {
        this.n = n;
        flag = new AtomicIntegerArray(n);
        label = new AtomicIntegerArray(n);
    }
    int MaxLabel() {
        int max = label.get(0);
        for (int i = 1; i<n; ++i)
            max = Math.max(max, label.get(i));
        return max;
    }
    boolean Conflict(int me) {
        for (int i = 0; i < n; ++i)
            if (i != me && flag.get(i) != 0) {
                int diff = label.get(i) - label.get(me);
```

```
                if (diff < 0 || diff == 0 && i < me)
                    return true;
            }
        return false;
    }
    public void Acquire(int me) {
        flag.set(me, 1);
        label.set(me, MaxLabel() + 1);
        while(Conflict(me));
        }

    public void Release(int me) {
        flag.set(me, 0);
    }
}
```

TAS

```
boolean TAS(memref s) {
    if (mem[s] == 0) {
        mem[s] = 1;
        return true;
    } else {
        return false
    }
}

Init(lock) {
    lock = 0;
    Acquire(lock) {
        while (!TAS(lock)); //wait
    }
    Release(lock) {
        lock = 0;
    }
}
```

CAS

```
    int CAS(memref a, int old, int newValue){
        oldVal = mem[a];
        if (old == oldVal){
            mem[a] = newValue
        }
        return oldVal;
    }

    Init (lock) {
        lock = 0;
```

```
        Acquire (lock) {
            while (CAS(lock, 0, 1) != 0); //wait
        }
        Release(lock) {
            CAS(lock, 1, 0); //ignore result
        }
    }
```

TASLock

```
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

    public void lock() {
        while (state.getAndSet(true)){}
    }

    public void unlock() {
        state.set(false);
    }
}
```

TATASLock

```
public class TATASLock implemnts Lock{
    AtomicBoolean state = new AtomicBoolean(false);

    public void lock() {
        do {
            while(state.get()) {}

        } while (!state.compareAndSet(false, true))
    }
}
```