

## 0 Contents

<b>1</b>	<b>Motivation</b>	<b>2</b>
1.1	Moore's Law	2
1.2	Hyperthreading vs Multiple Cores	2
1.3	Concurrent vs Parallel	2
1.3.1	Concurrency	2
1.3.2	Parallelism	2
<b>2</b>	<b>Parallel Programming in OS Space</b>	<b>2</b>
2.1	Processes and threads	2
2.2	Pros and Cons	2
2.3	User threads vs Kernel threads	3
2.4	Context Switch	3
2.5	Multi-Tasking	3
2.6	Thread States	3
2.7	Non-determinism	3
<b>3</b>	<b>Parallel Programming in Jafuck</b>	<b>3</b>
3.1	Thread Implementation	3
3.2	JVM Thread Model	3
3.3	JVM Termination	3
3.4	Thread in Java	3
3.5	Example for multithreading in java	4
3.6	Explicit/Sub-class Thread behavior	4
3.7	Thread join	4
3.8	Methods of a thread	4
<b>4</b>	<b>Thread Synchronization</b>	<b>4</b>
4.1	Race Condition	5
4.2	Critical Section	5
4.3	Synchronized methods in java	5
4.3.1	Synchronized Blocks	5
4.3.2	Multiple locks	5
4.3.3	wait()	5
4.3.4	Traps with Monitors	6
4.4	Spurious Wake-up	6
4.5	When a single notify is enough	6
4.6	Fairness	6
<b>5</b>	<b>Synchronization Methods</b>	<b>6</b>
5.1	Semaphore	6
5.1.1	Semaphore in java, geil	6
5.1.2	Types of semaphores	7
5.2	Bounded Buffer with Semaphores	7
5.2.1	Multi release/acquire	7
<b>6</b>	<b>Lock conditions</b>	<b>7</b>
6.0.1	Methods	8
6.1	Read-Write Locks	8
6.2	Countdown Latch	8
6.3	Cyclic Barrier	8
6.4	Exchanger	8
<b>7</b>	<b>Problems with concurrency</b>	<b>8</b>
7.1	Race Conditions	8
7.1.1	Data Races	8
7.2	Confinement / Thread Safety	8
7.2.1	Thread Confinement	8
7.2.2	Object Confinement	9
7.3	Thread Safety and Jafuck collections	9
7.4	Deadlocks	9
7.4.1	Live Locks	9
7.4.2	Dealing with Deadlocks	9
7.5	Starvation	9
7.6	Parallelism	9
<b>8</b>	<b>C#</b>	<b>9</b>
8.1	Basics	10
8.1.1	Spawning Threads	10
8.1.2	Monitor in C#	10
<b>9</b>	<b>Thread Pools</b>	<b>10</b>
9.1	Problem with Threads	10
9.2	Solution	10
9.3	Task Queue	10
9.3.1	Run To Completion	10
9.4	Thread Pools in Java	10
9.4.1	Thread Pool Instantiation	11
9.4.2	Functions	11
9.4.3	Parallel Counting Example	11
9.4.4	Full Task Implementation	11
9.5	Thread Pools in C#	11
9.5.1	Tasks in C#	12
9.5.2	Task with return Value	12
9.5.3	Nested Tasks	12

9.5.4	Parallelism without explicit tasks	12
9.5.5	Parallelism with explicit tasks	12
9.5.6	Data Parallel ForEach	12
9.5.7	Data Parallel For	12
9.5.8	Parallel Loop Partitioning	12
9.5.9	Parallel For with Thread-Local Variables	13
9.5.10	Parallel LinQ	13
9.5.11	Comparison to Java Stream API	13

## 1 Motivation

### 1.1 Moore's Law

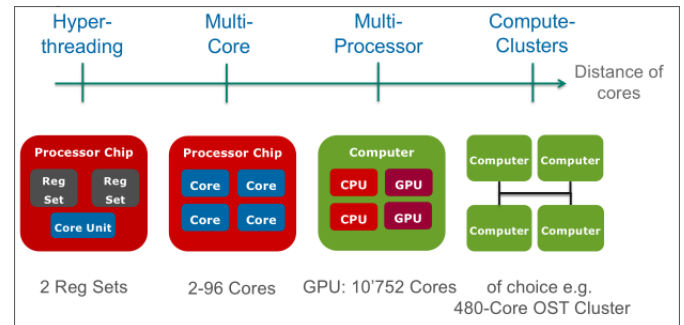
Until now the idea was that more transistors, more GHz, etc means that we can go faster and faster. However, the issue is that the increases are getting slower and slower, while the workloads are getting increasingly complex with clear need for *multithreading*.

### 1.2 Hyperthreading vs Multiple Cores

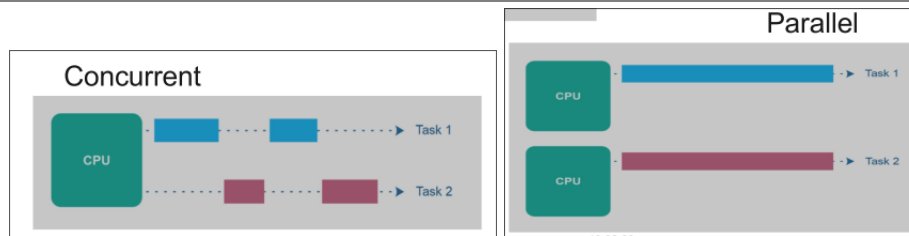
Hyperthreading creates the illusion of multiple cores via switching the context of the registers efficiently.

In other words, it just properly streamlines computation similar to JS await/async.

Note the different instruction set and the same compute core with hyperthreading!



### 1.3 Concurrent vs Parallel



#### 1.3.1 Concurrency

Concurrency has the goal of *simpler programs*, it does this by offering *simultaneous or interleaved(time shared)* execution that accesses shared resources.

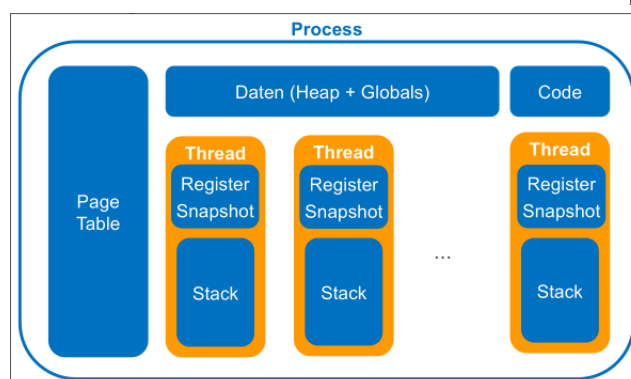
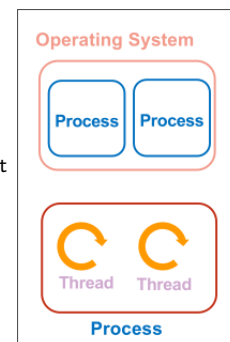
#### 1.3.2 Parallelism

Parallelism has the goal of *faster programs*, it does this by decomposition of a program into *several sub-programs*, which can run in parallel on multiple processors or cores.

## 2 Parallel Programming in OS Space

### 2.1 Processes and threads

A process is the instance of a program, while a thread is a subpart of a program, which will then have different callstacks. Meaning that each thread will have it's own callstack but share the same heap!



### 2.2 Pros and Cons

Cons:

- Interprocess Communication
- Process Management via system calls
- Memory isolation

Pros:

- Process isolation
- Responsiveness

## 2.3 User threads vs Kernel threads

User level threads is a so-called *green thread*, it can't offer true parallelism and is only scheduled by a runtime library or a virtual machine.

Kernel level thread is the true form of multithreading. *native threads* It offers context switching via *SW interrupt*.

## 2.4 Context Switch

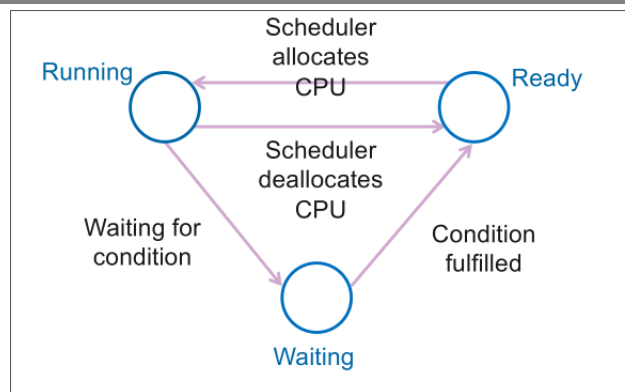
- **Synchronous**
  - Thread waiting for condition
  - queues itself as waiting and gives processor free
  - locks processor during usage
- **Asynchronous**
  - after some defined time the thread should release the processor
  - prevent a thread from permanently occupying the processor (solves locks)

## 2.5 Multi-Tasking

- **Cooperative**
  - threads must explicitly initiate context switches synchronously at the scheduler at intervals
  - scheduler cannot interrupt running thread
- **Preemptive**
  - scheduler can interrupt the running thread asynchronously via timer interrupt
  - Time-Sliced scheduling: each thread has the processor for maximum time interval

Preemptive is used for the most part!!

## 2.6 Thread States



## 2.7 Non-determinism

When using multi-threading, you can't be sure which thread will be used to complete a task first, meaning that if you print 2 different statements in 2 threads, then you can't be sure of the order in which these statements will be printed.

## 3 Parallel Programming in Jafuck

### 3.1 Thread Implementation

Java implements its own threads which are then linked to the kernel threads.



### 3.2 JVM Thread Model

Java is a single process system -> JVM as one process on the operating system.

JVM creates a thread at startup which calls `main()`.

You are then free to call/create more threads as the programmer.

### 3.3 JVM Termination

- The JVM runs as long as threads are running, the main function doesn't matter in this case
  - The only exception are so called *daemon threads*, these are threads like the garbage collector, which ofc needs to be ignored for the jvm to EVER end.
  - You can exit manually via `System.exit()` or `Runtime.exit()`
- Note that this means uncontrolled termination of all threads. This can lead to *undefined behavior*.

### 3.4 Thread in Java

A thread in java takes a so called "runnable target", this is an interface that simply defines the type of behavior that can be run inside of a thread. For example the thread might run something like a lambda.

```

public class Thread implements Runnable {
    private Runnable target;
    public synchronized void start() {
        if (threadStatus != 0)
            throw new IllegalThreadStateException();
        group.add(this);
        boolean started = false;
    }
    public void run() {
        if (target != null) {
            target.run();
        }
    }
    public Thread() {
        this(null, null, "Thread-" + nextThreadNum(), 0);
    }
    public Thread(Runnable target) {
        this(null, target, "Thread-" + nextThreadNum(), 0);
    }
}

```

```
}
}
```

Note that should a thread cause an exception, the other threads will continue to run!

### 3.5 Example for multithreading in java

```
public class MultiThreadTest {
    public static void main(String[] args) {
        var a = new Thread(() -> multiPrint("A"));
        var b = new Thread(() -> multiPrint("B"));
        a.start();
        b.start();
        System.out.println("main finished");
    }
    static void multiPrint(String label) {
        for (int i = 0; i < 10; i++) {
            System.out.println(label + ": " + i);
        }
    }
}
```

Lambdas as well as method references implement the runnable interface

### 3.6 Explicit/Sub-class Thread behavior

You can explicitly set the behavior of a thread via overriding the run method of the runnable interface.

Note that this means you will implement your own thread!

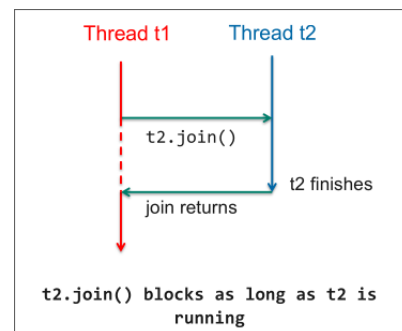
```
class SimpleLogic implements Runnable {
    @Override
    public void run() {
        // thread behavior
    }
}
var myThread = new Thread(new SimpleLogic());
myThread.start();
```

If you simply with to extend it you can extend the thread class instead:

```
class SimpleThread extends Thread {
    @Override
    public void run() {
        // thread behavior
    }
}
var myThread = new SimpleThread();
myThread.start();
```

### 3.7 Thread join

This is used when you specifically want another thread to be blocked while another one is running, because you might have a usecase where that one thread needs to fulfill their job.



```
var a = new Thread(() -> multiPrint("A"));
var b = new Thread(() -> multiPrint("B"));
System.out.println("Threads start");
a.start();
b.start();
a.join();
b.join();
System.out.println("Threads joined");
```

### 3.8 Methods of a thread

- **thread.sleep(milliseconds)**  
waits until time has elapsed before becoming ready again -> wait until it is scheduled again to run
- **thread.yield()**  
thread releases processor and will be ready to be used again -> wait until it is scheduled again to run  
For newer systems where preemptive multi-tasking is used, there is no need for yield, as allocation is time based either way!
- **thread.interrupt()**  
Used for cooperative canceling.  
This is an indication to the thread that it should stop current operation and do something else.  
This can be used by the programmer to decide how the thread will respond to an interrupt, however it is common for the thread to terminate on interrupt.


## 4 Thread Synchronization

This is the try to get a synchronized order of running threads.

In other words, we sacrifice some performance for the sake of deterministic running of threads.

## 4.1 Race Condition

When 2 threads try to access, more specifically *write* to the same variable at the same time, you might lose one of these writes if their write overlaps.  
This is yet another reason to just use rust, as this is handled via the borrow rules!  
*Only 1 mutable reference at a time, OR, unlimited immutable references.*

Thread 1	Balance	Thread 2
read balance => reg (reg == 0)	0	
	0	read balance => reg (reg == 0)
reg = reg + amount (reg == 100)	0	
	0	reg = reg + amount (reg == 50)
write reg => balance	100	
	50	write reg => balance
 Expected: 150		

## 4.2 Critical Section

This is a certain part of a job that must be done in sequence, for example a *transaction of a bank*.

The solution might seem easy, why not just create a locked variable, however, if you do this then you will run into the same problem, When exactly will the lock be written? The same race condition can apply here.

## 4.3 Synchronized methods in java

In java you can mark a method as synchronized to make sure that you will not run into race conditions.

```
class BankAccount {
    private int balance = 0;
    // function that may not be "truly" multithreaded
    public synchronized void deposit(int amount) {
        this.balance += amount;
    }
}
```

When using these methods, make sure that *ALL* methods that need to be synchronized also have this flag.  
In other words if you have a withdraw function and a deposit method, make sure both are marked as synchronized

### 4.3.1 Synchronized Blocks

You can also just specify a block to be synchronized.

In this case you need to pass a class to be locked!

```
class BankAccount {
    private int balance = 0;
    public void deposit(int amount) {
        synchronized(this) {
            // specify the this object to be locked
            this.balance += amount;
        }
        System.out.println("Deposit done");
    }
}
```

Similar, you can make locks on both *Classes and Objects*.

```
class Test {
    synchronized void f() { ... } // object lock
    static synchronized void g() { ... } // class lock
}

class Test {
    void f() { // object lock
        synchronized(this) { ... }
    }
    static void g() { // class lock
        synchronized(Test.class) { ... }
    }
}
```

### 4.3.2 Multiple locks

When a method is recursive and synchronized, the method will have multiple locks, of these, it needs to release all of them at the end for the next thread to acquire the locks.

In other words, the locks are like refcounting.

```
synchronized void limitedDeposit(int amount) {
    if (amount + balance <= limit) {
        deposit(amount);
    }
}

synchronized void deposit(int amount) { ... }
// will free the locks once all of the calls are done.
```

### 4.3.3 wait()

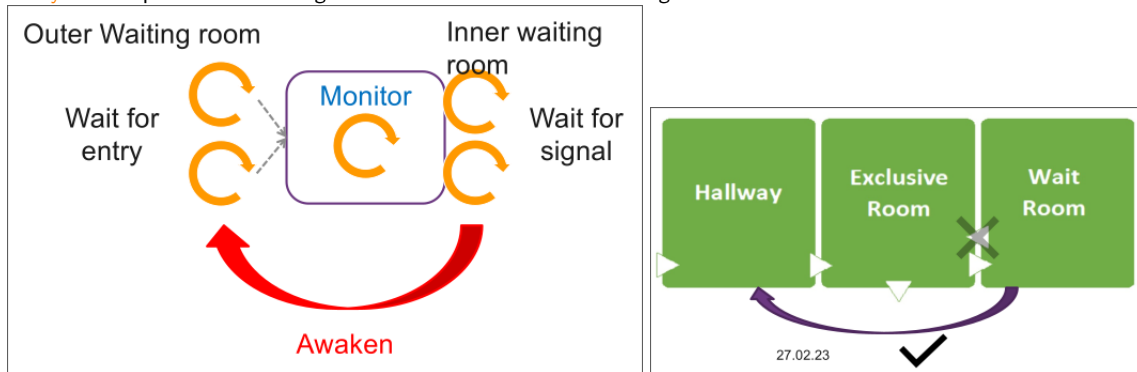
The wait method will release the lock and regain it after the thread has received a *wake up call*.

```
class BankAccount {
    private int balance = 0;
    public synchronized void withdraw(int amount) throws InterruptedException {
        while (amount > balance) {
            wait(); // release lock and go into waiting mode
        }
        balance -= amount;
    }
}

public synchronized void deposit(int amount) {
    balance += amount;
    notifyAll(); // wakeup all waiting threads!
}
```

As you can see, if a thread goes into waiting mode and does not receive a wakeup call, it will not start again.  
This can cause *deadlocks*, as you might have multiple threads waiting for something to happen, but nothing ever will.

- **notifyAll** wakes up *all* waiting threads in the inner waiting room
- **notify** wakes up a *random* waiting thread which is in the inner waiting room



In other words, `wait()` will put the thread in the hallway, the operating system will then put the thread into the exclusive room after it is scheduled. From here it can be awoken early with `notifyALL`, or it can wait until it is in the proper waiting room, where it can be called with `notify()`.

#### 4.3.4 Traps with Monitors

##### Trap 1: IF

```
public synchronized void put(T x) throws InterruptedException {
    if (queue.size() == capacity) {
        wait(); // await non-full
    } // oops we only waited once, if the capacity is still to small we now have a problem!
    queue.add(x);
    notify(); // signal non-empty
}
// instead!
while (!condition) {
    wait(); // this makes sure that we will wait even after the thread gets another lock!
}
```

##### Trap 2: Overtaking

Here the issue is that if you use `notify()`, you are going to call a random thread, in case you need serialization, you will not be achieving this with `notify()`, instead use `notifyALL()`!

```
// ...
queue.add(x);
notify(); // signal non-empty
```

##### Trap 3: Eternal Wait

Here the issue is that with a single `notify()`, you might awaken a thread that will then also wait for another condition, which will not notify any other thread, meaning that there is no further notify chaining.

You will now wait for something that will never be done, good job :)

#### 4.4 Spurious Wake-up

This is a OS specific feature, for example for POSIX operating systems.

Here the thread can *spuriously awaken without a specific notify*, this is something that you as a developer have to guard against!

#### 4.5 When a single notify is enough

```
public class Turnstile {
    private boolean open = false;
    public synchronized void pass()
        throws InterruptedException {
        while (!open) { wait(); }
        open = false;
    }
    public synchronized void open() {
        open = true;
        notify(); // or notifyAll() ?
    }
}
```

Here the idea is that if one passes, only one new thread can be reactivated. 1 in, 1 out.

This means that you can't notify everyone -> would unnecessarily wake up other threads.

The other condition that needs to be true is that every waiting thread is waiting for the same condition!

- 1 in, 1 out
- same condition for every thread

#### 4.6 Fairness

When calling `notify`, we have no guarantee of order which thread will be woken up. This is called the problem of *fairness*. `notifyAll` helps this by waking up EVERY thread.

In short, with java monitor, you either have the *fairness problem with notify*, or *inefficiency with notifyAll*!

### 5 Synchronization Methods

#### 5.1 Semaphore

- Essentially a counter for resources
- acquiring resource -> -1
- releasing resource -> +1
- wait for resource if necessary
- relatively low level and efficient! -> no `notifyAll`!!

##### 5.1.1 Semaphore in java, geil

- `acquire()` => `P()`  
if not available, wait until another thread releases
- `release()` => `V()`

You can always call `release`! This means you can create fake resources, it is your responsibility to make sure to not call `release` in this case!

```

public class Semaphore {
    private int value;
    public Semaphore(int initial) {
        value = initial;
    }
    public synchronized void acquire() throws InterruptedException {
        while (value <= 0) { wait(); }
        value--;
    }
    public synchronized void release() {
        value++;
        notify();
        here
    }
}

```

### 5.1.2 Types of semaphores

- **General Semaphore -> from 0 to N**
  - new Semaphore(N)
  - Up to N threads are allowed to acquire at the same time.
  - for limited resources/quotas etc
- **Binary Semaphore -> 0 or 1**
  - new Semaphore(1)
  - For mutual exclusion (1 = open, 0 = closed)
- **Fair Semaphore**
  - New Semaphore(N, true)
  - uses FIFO for fairness
  - the longest waiting thread gets resource
  - slower than other variants
  - the true or false flag defines if the semaphore is fair -> flag

### 5.2 Bounded Buffer with Semaphores

```

class BoundedBuffer<T> {
    private Queue<T> queue = new LinkedList<>();
    private Semaphore upperLimit = new Semaphore(Capacity, true); // lowerbound of buffer
    private Semaphore lowerLimit = new Semaphore(0, true); // upperbound of buffer
    private Semaphore mutex = new Semaphore(1, true); // mutex to ensure no race conditions
    public void put(T item) throws InterruptedException {
        upperLimit.acquire();
        mutex.acquire(); queue.add(item); mutex.release();
        lowerLimit.release();
    }
    public T get() throws InterruptedException {
        lowerLimit.acquire();
        mutex.acquire(); T item = queue.remove(); mutex.release();
        upperLimit.release();
        return item;
    }
}

```

#### 5.2.1 Multi release/acquire

You can release or acquire multiple resources at once with *release(n)* or *acquire(n)*, with n being a random number.

### 6 Lock conditions

Before we only worked with one monitor, however that is not always ideal. Usually we want to work with many waiting rooms, monitors etc. In this case using synchronized can be very harsh on performance, instead we can use *multiple waiting rooms and locks*.

```

Lock monitor = new ReentrantLock(true);
Condition nonFull = monitor.newCondition();
Condition nonEmpty = monitor.newCondition();

```

Here we have a nonFull queue and a nonEmpty queue, as well as a lock to ensure no race conditions! -> mutex

Here is how you would then go ahead to use this:

```

class BoundedBuffer<T> {
    private Queue<T> queue = new LinkedList<>();
    private Lock monitor = new ReentrantLock(true);
    private Condition nonFull = monitor.newCondition();
    private Condition nonEmpty = monitor.newCondition();
    public void put(T item) throws InterruptedException {
        monitor.lock();
        try {
            while (queue.size() == Capacity) { nonFull.await(); }
            queue.add(item);
            nonEmpty.signal(); Loop to avoid spurious wakeup
        } finally {
            monitor.unlock();
        }
    }
    public T get() throws InterruptedException {
        monitor.lock();
        try {
            while (queue.size() == 0) {
                nonEmpty.await();
            }
            T item = queue.remove();
            nonFull.signal();
            return item;
        } finally {
            monitor.unlock();
        }
    }
}

```

### 6.0.1 Methods

- `room.signal()` -> same as `notify()`
- `room.signalAll()` -> same as `notifyAll()`
- `room.await()` -> same as `wait()`  
Just like `wait()` `room.await()` throws that annoying exception...
- `lock()/unlock()` -> same as synchronized blocks

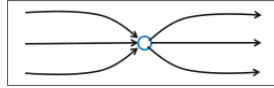
### 6.1 Read-Write Locks

Just like rust does by default, the idea is that you can read as long as you don't write, meaning you can either have *unlimited reads OR one write*.

### 6.2 Countdown Latch

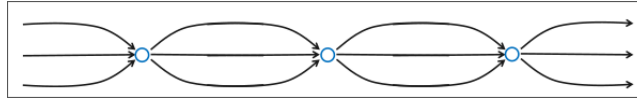
This is simply condition where you would need *more than 1 thread to be done with their execution*.

For example, in order for an F1 race to be over, you need all cars over the line. This can easily be done by using a counter, each car will decrement the counter, and the race is over when the count is 0.



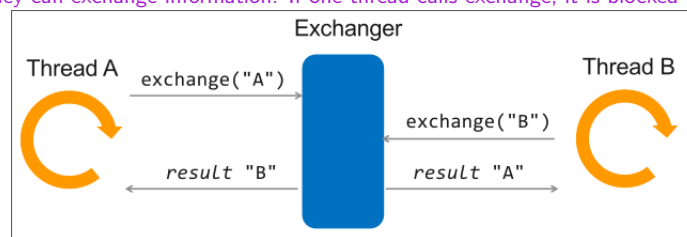
### 6.3 Cyclic Barrier

This is essentially a *reusable countdown latch*, when the count is 0 at the end of the race, it will be reset for the new race.



### 6.4 Exchanger

When 2 threads stop simultaneously, they can exchange information. If one thread calls exchange, it is blocked until the other thread also calls exchange.



Example:

```
var exchanger = new Exchanger<Integer>();
for (int count = 0; count < 2; count++) {
    new Thread(() -> {
        for (int in = 0; in < 5; in++) {
            try {
                int out = exchanger.exchange(in);
                System.out.println(
                    Thread.currentThread().getName() + " got " + out);
            } catch (InterruptedException e) { }
        }
    }).start();
}
```

## 7 Problems with concurrency

### 7.1 Race Conditions

This is nondeterministic behavior.

Eg. wrong order of execution -> addition before multiplication.

This can occur with, OR without data races.

An example of a race condition even with synchronized is this:

```
class BankAccount {
    int balance = 0;
    synchronized int getBalance() { return balance; }
    synchronized void setBalance(int x) { balance = x; }
} // problem: we can use getBalance and setBalance in the wrong order, even though
// the updates themselves are now protected, meaning there is no write or read at the same time!
// Rust also can't solve this issue, this is a programmer issue!!
```

#### 7.1.1 Data Races

Eg. two threads that want to access the same variable/object at the same time.

In this case there is no method that guarantees any sort of serialization, instead it is random.

Note, for a data race to occur, there needs to be *at least one write access with either one or more additional write OR read access*, this is exactly what the rust borrow checker enforces!

	Race Condition	no Race Condition
Data Race	Erroneous behaviour	Program works correctly, but formally incorrect
No Data Race	Erroneous behaviour	Correct behaviour

### 7.2 Confinement / Thread Safety

This limits the availability of objects/variables to something specific, eg. specific thread, or only inside synchronized functions/objects.

Thread safety simply means the elimination of data races. Note, race conditions can still occur!

#### 7.2.1 Thread Confinement

This ensures that the object is only accessible via a reference of a specific thread.




### 7.2.2 Object Confinement

This ensures that the object is only accessible inside of other synchronized objects.

```
class ProductDatabase {
    private final HashMap<String, Product> productMap = new HashMap<>();
    public synchronized void addProduct(String name, String details) {
        productMap.put(name, new Product(details));
    }
    public synchronized String getProductDetails(String name) {
        return productMap.get(name).getDetails();
    }
    public synchronized void notifySale(String name) {
        productMap.get(name).increaseSales();
    }
}
```

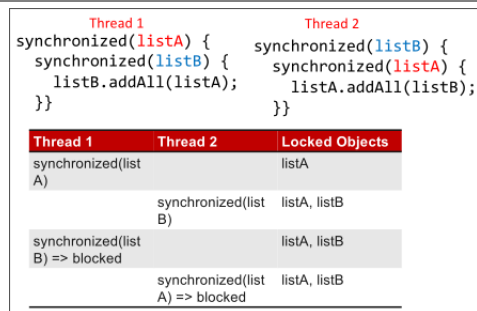
### 7.3 Thread Safety and Jafuck collections

Version	Beispiele	Thread-sicher
Old Java 1.0 Collections	Vector, Stack, Hashtable	yes
Modern Collections (java.util, Java > 1.0)	HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap	NO 
Concurrent Collections (java.util.concurrent)	ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList, ...	yes

In general, you always need to make sure that everything you use is thread safe, in general, older collections are thread safe with more modern ones not being thread safe for some reason.

Probably because jafuck is a horrible language and no longer has any right to exist.

### 7.4 Deadlocks



Deadlocks can also happen with synchronized, as the thread itself will be locked when used:

```
class BankAccount {
    private int balance;
    public synchronized void transfer(BankAccount to, int amount) {
        balance -= amount;
        to.deposit(amount);
    }
    public synchronized void deposit(int amount) {
        balance += amount;
    }
}
a.transfer(b, 20); // acquires lock to b
b.transfer(a, 50); // oops....
```

#### 7.4.1 Live Locks

This is a special case of deadlocks, where the cpu is actually still working, eg. it is in a while loop waiting for a lock to be released.

However, that lock will never be released, as two threads have blocked each other!

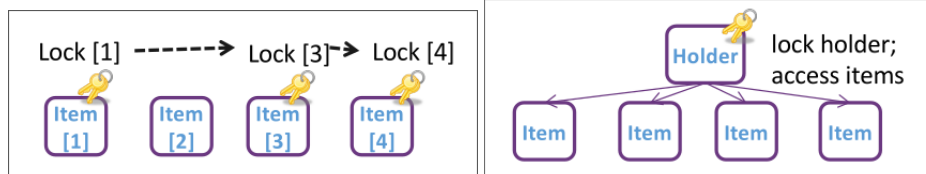
This is even worse than deadlocks, as this means the cpu will waste power on garbage.

#### 7.4.2 Dealing with Deadlocks

You could lock linearly, meaning that you will start with 1 lock and expand as needed.

Other threads can't start locking, as the base is already locked.

Note that depending on how deep the base lock is, you can call it a "granular lock" -> lock at root



### 7.5 Starvation

This happens when the lock avoidance leads to certain threads not being able to do their job.

Eg the submissive thread that always releases its lock as soon as someone else wants it will never actually be able to finish its task, meaning that you might end up with a problematic state that blocks or even crashes your program.

The best way to deal with this is using something like semaphores that will enforce fairness, but always keep in mind that this will be *at the cost of performance*.

### 7.6 Parallelism

In the end what we want is a program that has *no race conditions, no deadlocks and no starvation, with still decent performance*.

## 8.1 Basics

### 8.1.1 Spawning Threads

```
var myThread = new Thread(() => {
    for (int i = 0; i < 100; i++) {
        Console.WriteLine("MyThread step {0}", i);
    }
});
myThread.Start();
myThread.Join();
```

### 8.1.2 Monitor in C#

```
class BankAccount {
    private decimal balance;
    public void Withdraw(decimal amount) {
        lock (syncObject) {
            while (amount > balance) {
                Monitor.Wait(syncObject);
            }
            balance -= amount;
        }
    }
    public void Deposit(decimal amount) {
        lock (syncObject) {
            balance += amount;
            Monitor.PulseAll(syncObject);
        }
    }
}
```

## 9 Thread Pools

### 9.1 Problem with Threads

- scheduling needs time
- thread might awake only for it to wait again since condition is not met
- limited in numbers
- every thread needs a stack for itself
- preemptive full register-backup for each thread

### 9.2 Solution

Instead of using *all threads all the time*, we only use some of them. (at least as many as physical cores)

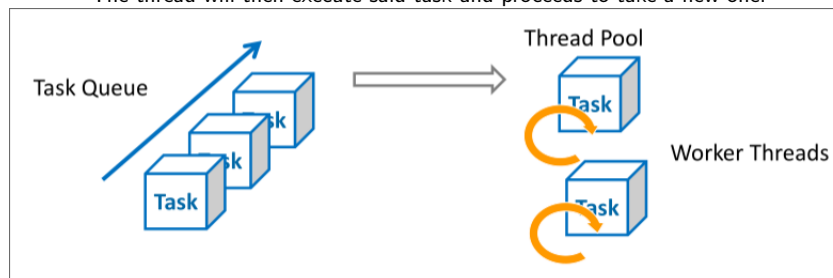
Then we define the functionality in question as *task* instead.

These tasks can then be parallelized, or not, depending on the functionality in question.

### 9.3 Task Queue

Tasks are handled as a queue, from which threads can take individual tasks from.

The thread will then execute said task and proceeds to take a new one.

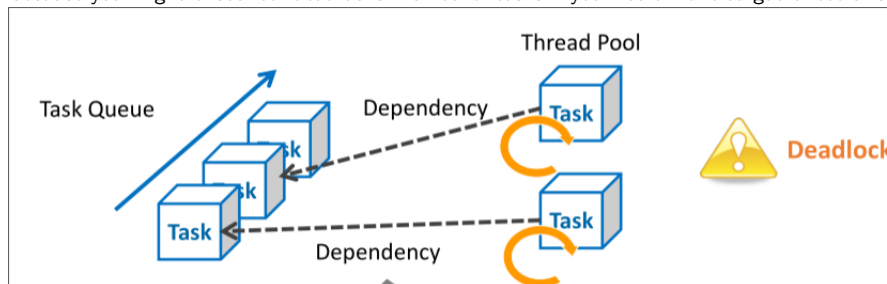


Benefits:

- scalable -> more threads or cores automatically means more performance
- easily made to run in parallel -> multi CPU structures

#### 9.3.1 Run To Completion

As already stated above, each task needs to be run to completion, however *this does not apply to nested subtasks*, this is because you might encounter deadlocks with other tasks if you would want to guarantee this as well.



### 9.4 Thread Pools in Java

```
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.submit(() -> {
    // this is the task to execute
    int value = 50;
    // do something with value
    return value;
});
```

Essentially a promise in js, you don't know if it is done or not.

To check use the following:

```
Future<T> future = threadPool.submit(yourtask); // launch task
T result = future.get(); // blocks until result is here
```

Note, unlike js, the `get()` function will *block the thread* until the result is here!!

#### 9.4.1 Thread Pool Instantiation

```
// explicit pool
var threadPool = new ForkJoinPool();
int result = threadPool.invoke(new CountTask(2, N));

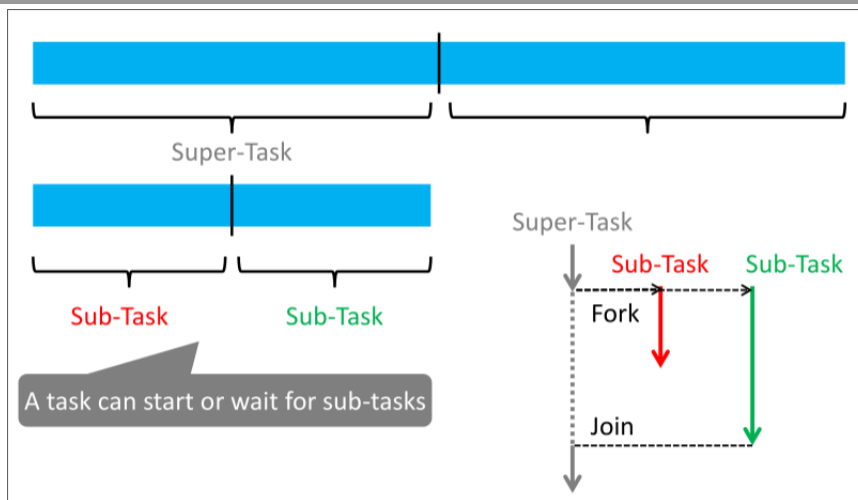
// using default pool
int result = new CountTask(2, N).invoke();
```

Important, when using the default pool, it will not always use all processors, as it does internal optimization.

#### 9.4.2 Functions

- **RecursiveAction**  
like `RecursiveTask<Void>` -> simply no return value
- **invokeAll()**  
start and wait for sub-tasks -> `invokeAll(task1, task2)`
- **submit()**  
create a new task
- **get()**  
block until result is here
- **invoke()**  
run task
- **cancel()**  
cancel task
- **fork()**  
split task into subtasks
- **join()** wait for the other task to finish after own execution has stopped

#### 9.4.3 Parallel Counting Example



```
class CountTask extends RecursiveTask<Integer> {
    // Constructor
    @Override
    protected Integer compute() {
        // if no or single element => return result
        // split into two parts
        var left = new CountTask(lower, middle);
        var right = new CountTask(middle, upper);
        left.fork(); right.fork();
        return right.join() + left.join();
    }
}
```

#### 9.4.4 Full Task Implementation

```
class CountTask extends RecursiveTask<Integer> {
    private final int lower, upper;
    public CountTask(int lower, int upper) {
        this.lower = lower; this.upper = upper;
    }
    protected Integer compute() {
        if (upper - lower > THRESHOLD) {
            // parallel count
            int middle = (lower + upper) / 2;
            var left = new CountTask(lower, middle);
            var right = new CountTask(middle, upper);
            left.fork(); right.fork();
            return right.join() + left.join();
        } else {
            // sequential count
            int count = 0;
            for (int number = lower; number < upper; number++) {
                if (isPrime(number)) { count++; }
            }
            return count;
        }
    }
}
```

#### 9.5 Thread Pools in C#

- efficient thread pool by default
- Task Parallization: use tasks explicitly
- Data Parallization: use parallel statements and queries using tasks explicitly
- Asynchronous Programming

### 9.5.1 Tasks in C#

```
Task task = Task.Run(() => {
    // task implementation
});
// perform other activity
task.Wait(); // blocking just like in jafuck
```

### 9.5.2 Task with return Value

```
Task<int> task = Task.Run(() => {
    int total = // some calculation
    return total;
});
// no wait/get etc necessary!
Console.Write(task.Result); // result of task
```

### 9.5.3 Nested Tasks

```
var task = Task.Run(() => {
    // unlike java you don't have to specifically say that you want to wait for the task
    // instead you can just use it in the return.
    var left = Task.Run(() => Count(leftPart));
    var right = Task.Run(() => Count(rightPart));
    return left.Result + right.Result;
});
static Task<int> Count(part) {
    // do something
}
```

### 9.5.4 Parallelism without explicit tasks

Some things can be done in parallel without specifying it.

```
void MergeSort(l, r) {
    long m = (l + r)/2;
    MergeSort(l, m);
    MergeSort(m, r);
    Merge(l, m, r);
}

void Convert(IList list) {
    foreach (File file in list) {
        Convert(file);
    }
}
```

### 9.5.5 Parallelism with explicit tasks

```
Parallel.Invoke(
    () => MergeSort(l, m),
    () => MergeSort(m, r)
);
```

Here the continuation is blocked until *both threads have stopped executing!*

### 9.5.6 Data Parallel ForEach

```
Parallel.ForEach(list,
    file => Convert(file)
);
```

Queue a task for each item in the foreach sequence. This blocks by default until all tasks are done!

### 9.5.7 Data Parallel For

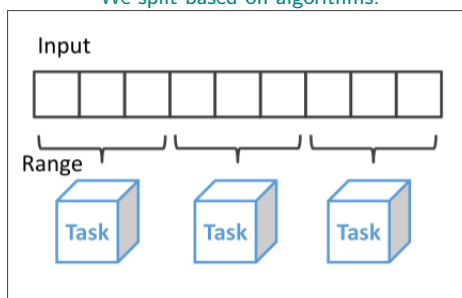
```
for (i = 0; i < array.Length; i++) {
    DoComputation(array[i]);
} // sequential
Parallel.For(0, array.Length,
    i => DoComputation(array[i])
); // parallel, only works in case iterations are independent!
```

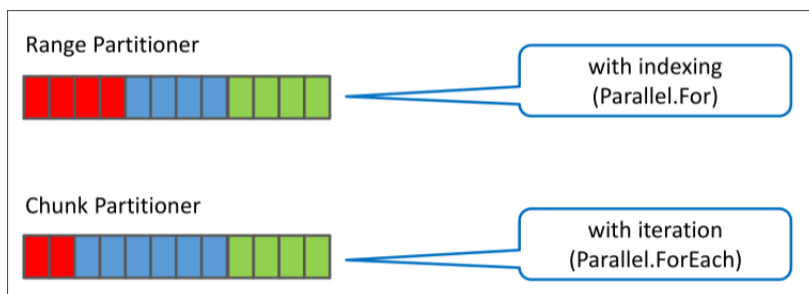
Please note, for the second variant the iterations need to be independent.

### 9.5.8 Parallel Loop Partitioning

When you iterate through an array, you might want to split the iterations to multiple threads. However, making each iteration run on a new thread is really inefficient, so what can we do?

We split based on algorithms:





### 9.5.9 Parallel For with Thread-Local Variables

```
long total = 0;
Parallel.For<long>(
    0, array.Length,
    () => 0,
    (i, _, subtotal) => {
        subtotal += array[i];
        return subtotal;
    },
    subtotal => Interlocked.Add(ref total, subtotal)
);
```

### 9.5.10 Parallel LinQ

- Parallelization of language-integrated query (sql like)
- Data Parallelism
- Similar to Java Stream API

Examples:

```
// Get ISBN of all books with the title "Concurrency"
bookCollection
.Where(book => book.Title.Contains("Concurrency"))
.Select(book => book.ISBN)

// Map each number in a list to a bool value (true = prime number)
inputList
.Select(number => IsPrime(number))
```

You can also make this work with a sql like query

```
// Get ISBN of all books with title "Concurrency"
from book in bookCollection
where book.Title.Contains("Concurrency")
select book.ISBN

// Map each number in a list to a bool value (true = prime number)
from number in inputList
select IsPrime(number)
```

If you want to use this with *parallelization*:

```
// Get ISBN of all books with title "Concurrency"
from book in bookCollection.AsParallel()
where book.Title.Contains("Concurrency")
select book.ISBN
Results in random order

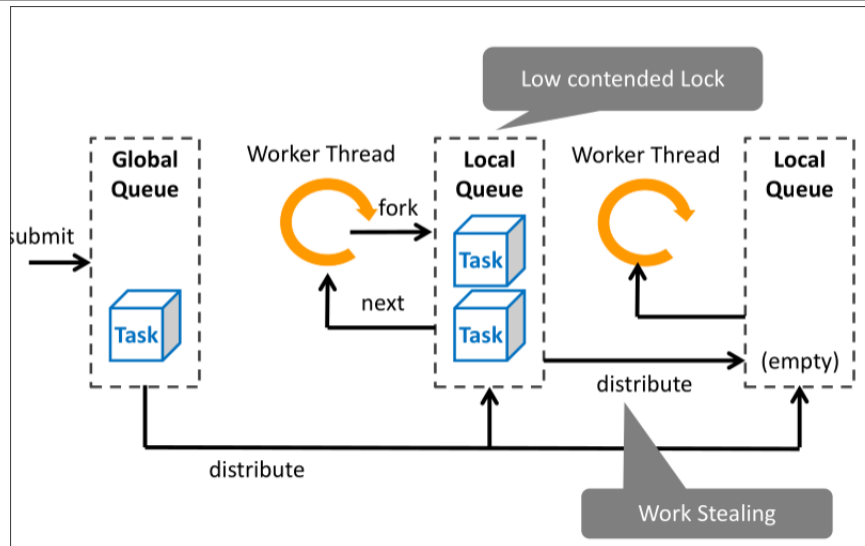
// Map each number in a list to a bool value (true = prime number)
from number in inputList.AsParallel().AsOrdered()
select IsPrime(number)
// as ordered makes sure it is still serialized! Otherwise the return order is weird
```

### 9.5.11 Comparison to Java Stream API

```
// Get ISBN of all books with title "Concurrency"
// Allow unordered explicitly (by default ordered in java)
bookCollection.parallelStream().unordered()
.filter(book -> book.getTitle().contains("Concurrency"))
.map(book -> book.getISBN());

// Map each number in a list to a bool value (true = prime number)
inputList.parallelStream()
.map(number -> isPrime(number));
```

## 9.5.12 Work Stealing Thread Pool



A thread will take multiple tasks from the global pool and then start working on them.

Should one thread be done with all tasks and there is still work to be done on other threads, the now afk thread will steal a task from the other threads.

## 9.5.13 Thread Injection

This is the idea that a controller will automatically add threads to certain tasks (*at runtime!*) if it sees a benefit in terms of runtime.

It does this with a *Hill Climbing Algorithm*, by measuring throughput.

In general this does solve deadlocks, but it is slower than regular thread pools. (deadlocks still possible with `ThreadPool.SetMaxThreads()`)