0 Contents

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
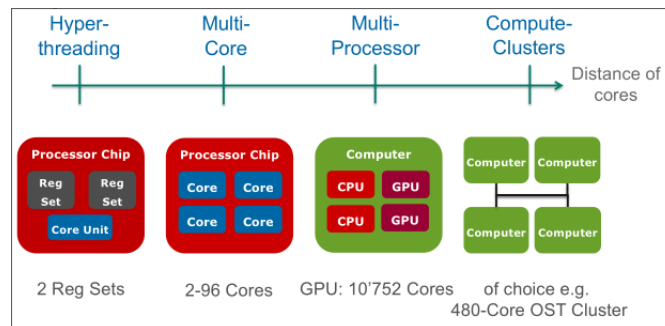
Hyperthreading creates the illusion of multiple cores via switching the context of the registers efficiently.
In other words, it just properly streamlines computation similar to JS await/async.
Note the different instruction set and the same compute core with hyperthreading!

1.3 Concurrent vs Parallel
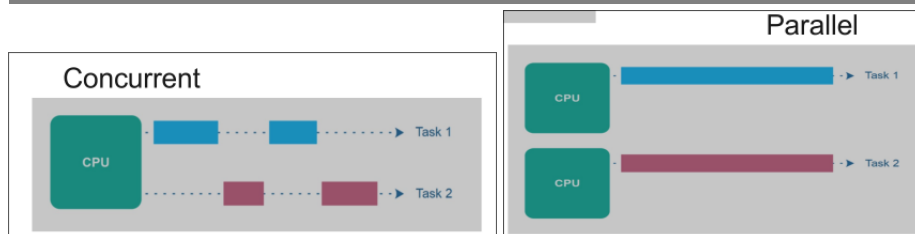
1.3.1 Concurrency

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

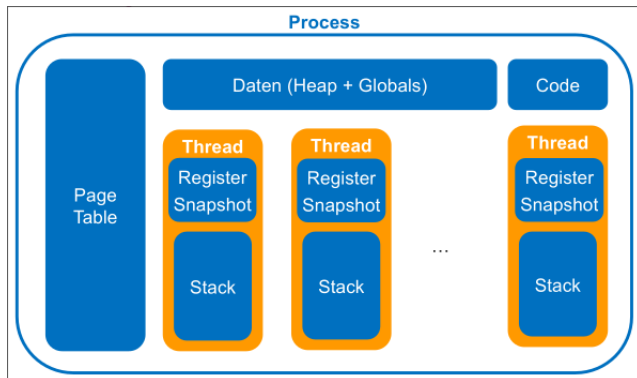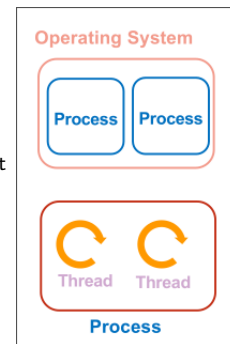### 2.1 Processes and threads

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
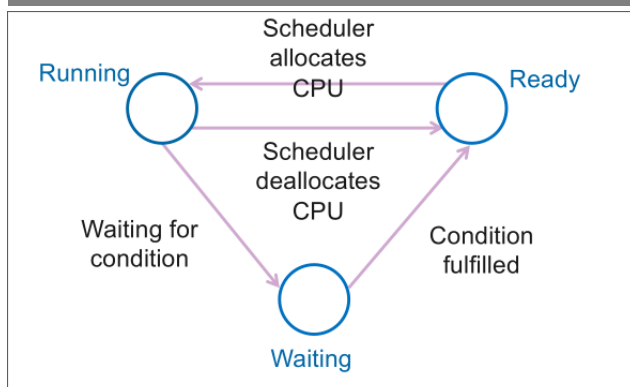
2.7 Non-determinism

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.
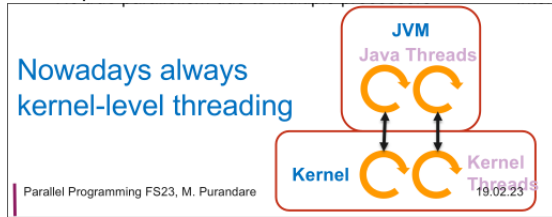
3.2 JVM Thread Model

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.

```java
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

```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!

```java
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:

```java
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.
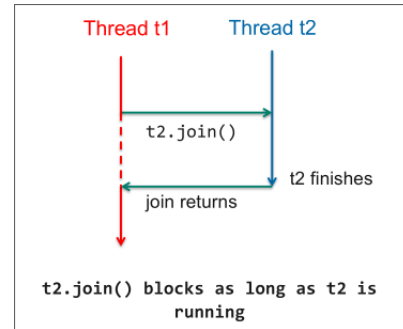

```
var a = new Thread(() -> multiPrint("A"));
var b = new Thread(() -> multiPrint("B"));
System.out.println("Threads start");
a.start();
b.start();
a.join();
b.join();
System.out.println("Threads joined");
```

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 if 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 synchronied 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*.

```java
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 aquire the locks.
In other words, the locks are like refcounting.

```java
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*.

```java
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!
}
```
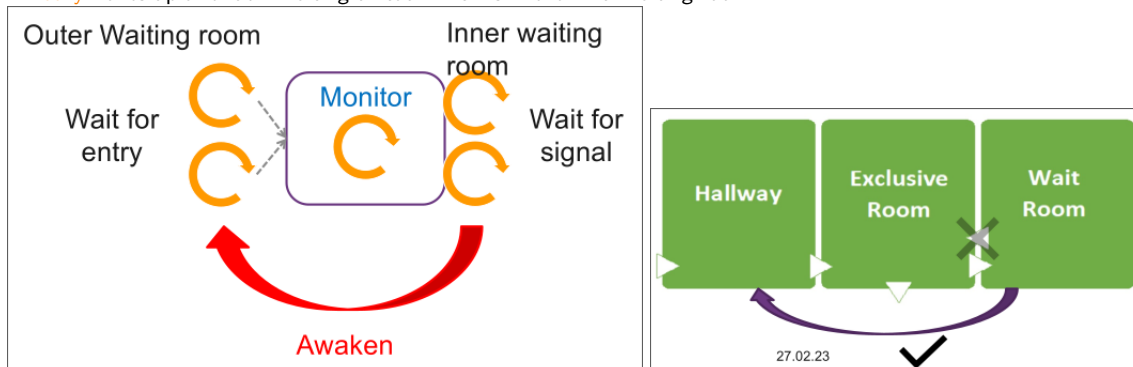
As you can see, if a thread goes into waiting mode and does not receive a wakeup call, it will not start again.
This can cause *deadlocks*, as you might have multiple threads waiting for something to happen, but nothing ever will.
- notifyAll wakes up *all* waiting threads *in the inner waiting room*
- 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

```java
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()!*

```java
// ...
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!