# Summary of chapter 4 – Concurrency and threads:

**What is the chapter about?**

The operating system can take advantage of doing multiple activities at once. Instead of doing one and one activity sequential it would be much more useful doing multiple activities at the same time. This is called **concurrency**, which refers to multiple activities that can happen at the same time.

We use threads as an abstraction to help bridge the gap (the gap referrers to the fact that humans are not very good at doing multiple thing at once).

**Why do we need concurrency? What profits from concurrency?**

* Networks/servers
  + A server need to handle multiple connections simultaneously.
  + Several users might want to connect to the server.
    - Therefore, a server would benefit from concurrency by being able to do multiple stuff at once.
* Parallel programs
  + A way to achieve better performance
  + Parallel programs need to be able to map work onto multiple processors to get the performance benefits of multicore architectures.
* Programs with user interfaces
  + To achieve user responsiveness while doing computation
  + The OS needs to provide good responsiveness to the user. At the same time it can handle different tasks in the background.
* Network and disk bound programs
  + Disks, and different IO is extremely slow compared to the processor. There is therefore an latency.
  + With concurrency we can hide the disk/IO latency.

**What is a thread?**

The definition of a thread is: *a single execution sequence that represents a separately schedulable task*. A process can have several threads, but a thread can only be part of one process.

**A single execution sequence** is the familiar programming model where everything goes in sequence.

**A separately schedulable task** means that the OS can run or suspend a thread at any time. So it can be *scheduled.*

**Why do we want to use threads?**

1. Program structure: Expressing logically concurrent task.
   1. The use of threads simplifies concurrent code. In addition, the threads let you express an application’s natural concurrency by writing each concurrent task as a separate thread.
2. Responsiveness: Shifting work to run in the background
   1. By having tasks running in background we improve responsiveness.
   2. The user interface remain responsive, even though the program may be very complex.
3. Performance: Exploiting multiple processors.
   1. The operating system can make threads run on a multiprocessor with them working parallel. They can do the same amount of money in less time.
4. Performance: Managing I/O devices
   1. While the processor is waiting for I/O-devices, it can run other tasks while waiting. The I/O is very slow, so in the meantime the processor can do a lot of work.

**How can we run, suspend and resume threads?**

It is important to mention that threads provide the illusion of infinite amount of processors. This is obviously not the case in reality. We have a specific amount of processors. So, how does the operating system provide this illusion? *By running, suspending and resuming threads*.

The OS use a **thread** **scheduler**. The thread scheduler can switch between threads that are running, and threads that are ready. We have a queue for threads that are ready to run.

Switching a thread is transparent to change the code being executed within each thread.

**Why do we run, suspend and/or resume threads?**

As mentioned in the question above, we use the thread scheduler because it switches between threads that are ready and threads that are running. This gives the illusion of infinite amount of processors. It is the thread scheduler that handles all the threads, and when they run etc.

**Why do we say that each thread run on a dedicated virtual processor with unpredictable and variable speed?**

Because it is the thread scheduler that handles all the threads. So you cannot necessarily say when the thread is going to be run. It is up to the thread scheduler. In other word, the threads run on a virtual processor with unpredictable and variable speed.

A programmer should make no assumptions about the behaviour of the thread scheduler.

Maskingenerert alternativ tekst:
Programmer Abstraetion 
Thresds 
Proeassors 
S 
Physieal Reality 
s 
2. 
S 
Hunning 
Tnreads 
Ready 
Threads 

To the left we can se how the threads think the reality is. They think they run on a dedicated virtual processor. Whereas, in reality, we only have two threads running, and the others are in the queue of ready threads.

**How do we create, and use threads? How does the thread API look like?**

We have many different functions related to threads. The most simple, but yet effective are:

* Void create\_thread(thread, func, arg)
  + Creates a new thread to run the fun with arguments arg.
* Thread\_yield()
  + The yield function makes the thread give up the processor voluntarily.
  + It gives away the execution to another thread.
  + The scheduler can resume the thread whenever it wants to.
  + Preemptive model (?)
* Thread\_join(thread)
  + If the thread is parent, wait for forked thread to exit, then return
  + It is similar to the wait() function in UNIX. The wait function waits for the child process to be executed before the parent starts executing.
  + In main, if thread\_join is called the main will wait for the thread to exit, then return.
  + Thread\_join might only be called **once** for each thread.
* Thread\_exit
  + Finishes the thread, then stores the value ret in the current thread’s data structure.
  + If a thread has been waiting after calling join, resume that thread.

**How can a thread create a new thread to perform work?**

A thread can create a child which can perform work for the parent. When the thread creates the child it uses the fork() function. Then the parent will wait for the their results. When waiting for the result, the parent uses the join. As mentioned, the join will wait for the forked thread to exit.

The data between the processes (?) can only be shared before fork or after join. This ensures independence.

**How is the thread data structure?**

The thread scheduler switches between different threads, and can afterwards resume suspended threads. For this to work, we need to save the state and different information about each thread. So we have an per-thread state, but also a state that is shared among threads.

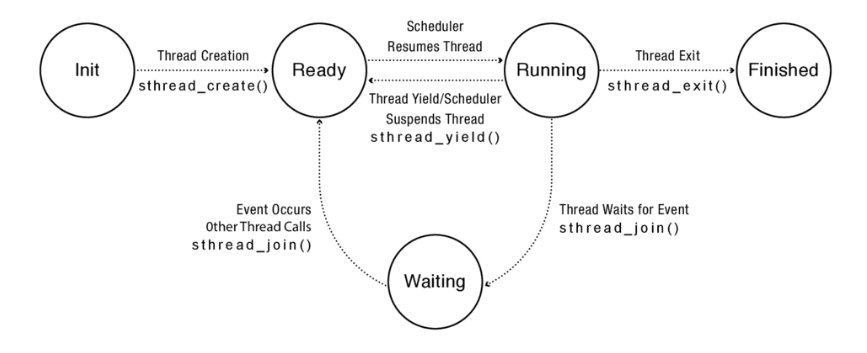
The data structure that saves the information about each thread is called *the thread control block* (TCB). The TCB holds two types of information about the threads:

* The state of the computation being performed by the thread
  + In order to being able to create, and start and stop threads, the OS must save the current state of the thread’ computation: a pointer to the stack and a copy of it’s processors registers.
* Metadata about the thread that is used to manage the thread
  + Thread ID, scheduling priority and status (waiting, running, init)

We also have a **shared state** between threads running in the same process or within the operating system kernel.

For example, all *program* *code* is shared by all threads in a process. Furthermore, all statically allocated global variables and dynamically allocated heap variables are stored in the shared state.

**How is the life cycle of a thread?**



The figure showing the lifecycle of a thread. From the initialization to it is finished

As the picture is showing, a thread can have different **states**.

In the **init** state, the thread is initialized, and being allocated and initialized per-thread data structures. Once that is done, the thread’s state is putted into the **ready** state. Here, it will be in the ready queue and waiting for the thread scheduler to run it.

If the thread is in the state **running**¸ it means it will be run on the processor. The registers values are now stored on the processor rather than in the TCB.

It is important to mention that a **running** thread can transition to the ready state in two ways:

* Thread\_yield (the thread will give away the execution to another thread), or the scheduler suspends thread.
* The scheduler moves the running thread to the ready queue. Often because a higher priority thread occurred.

After running the thread will **exit,** and is then in the state **finished**.

***I don’t really understand what he is trying to say in the slide “Implementing threads: Roadmap” and the start of the subchapter 4.6***

**How do we implement the threads?**

We then need to create the thread. We then use the function thread\_create(func, args). The thread will be created in the following order:

1. Allocate thread control block (TCB)
2. Allocate a stack to the thread
3. Build a stack frame for base of stack. The stack frame is the stub which is a dummy function for the function we want to run. So the thread starts in the stub rather than in func. Without the stub, if the func returns instead of calling thread\_exit, the func would return to whatever random location lying on top of the stack. So instead, the func will return to the stub, which then calls the thread\_exit.
4. Put func, args, which are the parameters, on the stack
5. Put the thread on the ready list, which signals that it is ready to run and will wait in line.
6. Will run sometimes later (maybe right away) depending on the scheduler.

**How can we switch between threads that are running and threads that are ready?**

The mechanism supporting the switch is called a thread context switch. What it basically does it that is suspends one running thread, and resumes another thread that are ready.

It is important that we save the state of the thread that were running because it may be resumed afterwards. Therefore, the switch saves the current running thread’s registers to the TCB and stack, and then restores the registers of the thread that is now going to run.

**What triggers a thread context switch? Why does it happen?**

We distinguish between two main types; voluntary and involuntary.

Voluntary is the cooperative context switch. This is by either the thread\_yield or the thread\_join function. The thread\_yield send an interrupt so the scheduler runs and we can run a different thread. The thread is giving up the processor voluntarily.

Voluntary is the preemptive context switch. A thread will stop running due to an interrupt/exception or that some other thread has a higher priority. If there is an interrupt or processor exception the interrupt handler will be inwoken and the state of the thread will be saved and then another thread will be run.

**How does voluntary thread context switch happen?**

As mentioned, a voluntary thread context switch is the cooperative thread context switch. When this occurs, the registers of the running thread will be saved to its TCB. Then the thread\_switch will switch to the stack of the new thread, restores the state of the thread and return to the program counter lying on the stack of the new thread.

**How does an involuntary thread context switch in kernel happen?**

Similar to when a processor exception or interrupt happen to a process. The state of the thread is saved. Then the kernel handler runs and will take care of the interrupt or the processor exception.