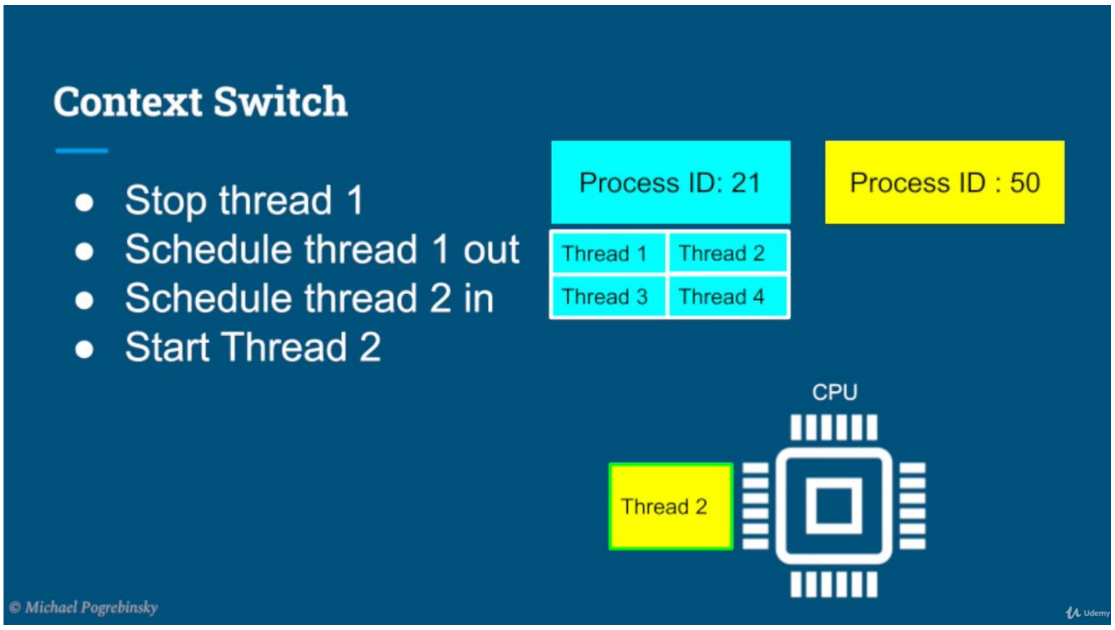
Introduction to Operating System

So what is a context switch?

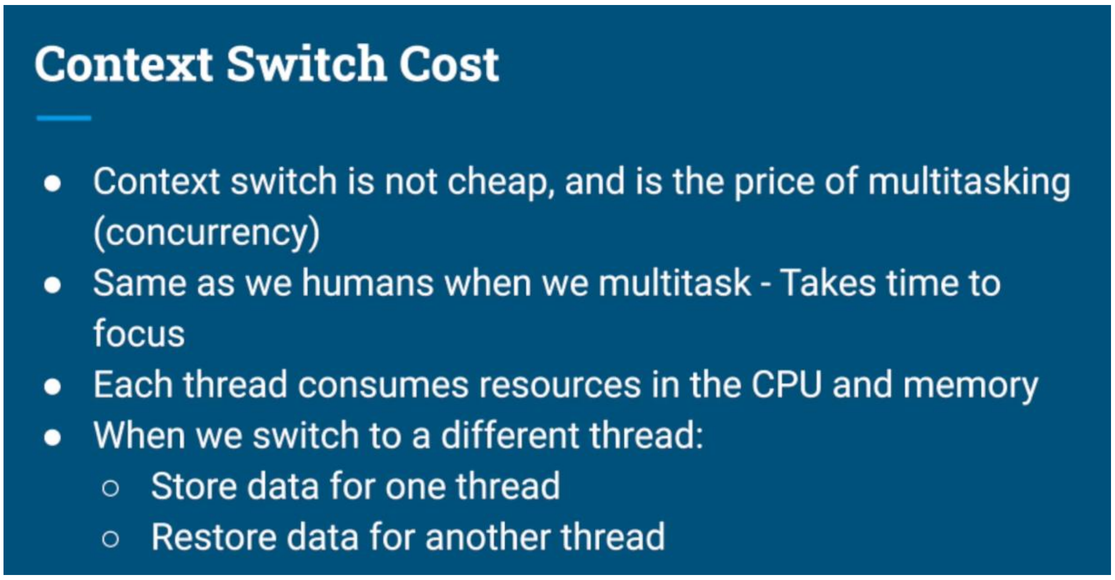
As we described in the previous lecture, each instance of an application that we run runs independently from other processes.
Normally they're way more processes than course.
Each process may have one or more threads and all these threads are competing with each other to be executed on the CPU.
Even if we have multiple cores, there are still way more threads than cores, So the operating system will have to run one thread, then stop it, run another thread, stop it.And so on

The act of stopping one thread, scheduling it out, scheduling in another thread and starting it is called a context switch.



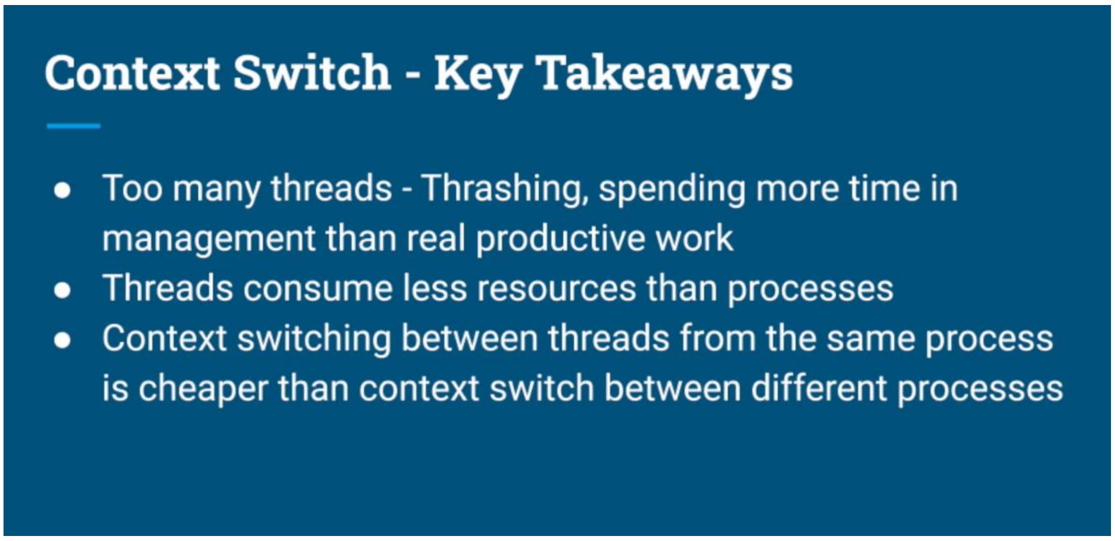
Context Switch

- Stop thread 1
- Schedule thread 1 out
- Schedule thread 2 in
- Start Thread 2



Context Switch Cost

- Context switch is not cheap, and is the price of multitasking (concurrency)
- Same as we humans when we multitask - Takes time to focus
- Each thread consumes resources in the CPU and memory
- When we switch to a different thread:
    - Store data for one thread
    - Restore data for another thread

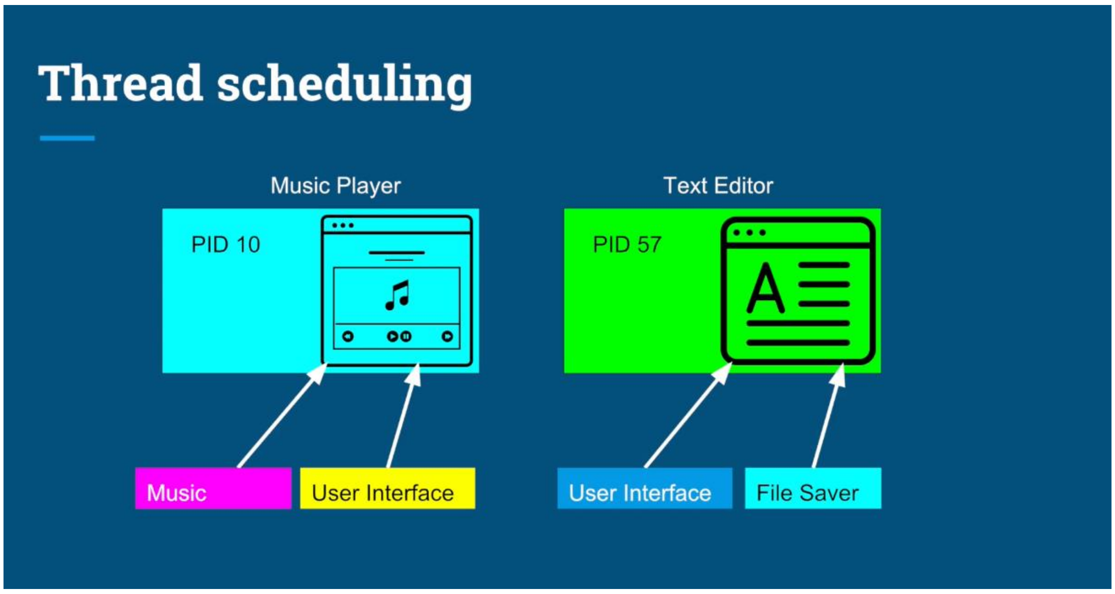


Context Switch - Key Takeaways

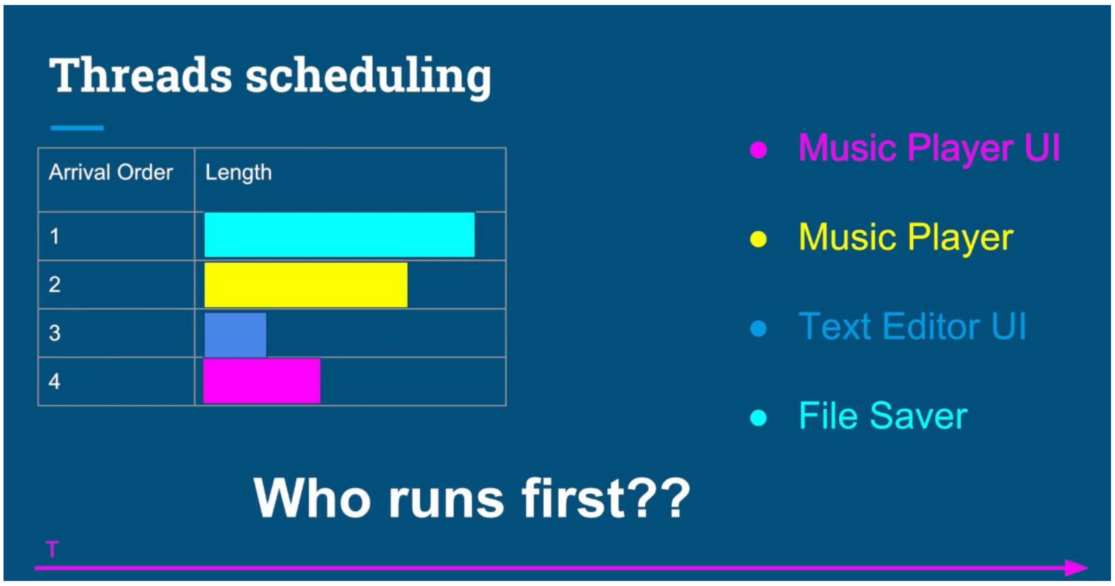
- Too many threads - Thrashing, spending more time in management than real productive work
- Threads consume less resources than processes
- Context switching between threads from the same process is cheaper than context switch between different processes

How the operating system decides when to run which thread and when to perform a context switch ?

Imagine we're doing a homework using our favorite text editor while listening to our favorite music in the background. So we have two processes, the text editor and the music player. For simplicity, say our music player has two threads.
One is loading the music from the file and planning through the speakers.
And the other thread is the UI thread that shows us the progress of the track and the response to mouseclicks on the play and stop buttons.
The text editor, which also has two threads.
One is again, a UI thread showing us what we already typed and also
response to keyboard and mouse events.
And the other thread runs every two seconds and saves our current work to a file.



So for simplicity, we also have one core and we have four threads.
We need to decide how to schedule on that one core.
So assuming the arrival or their, of their tasks and their corresponding
length of execution is given how does the operating system decide who runs first?



Threads scheduling

Arrival Order	Length
1	🟦
2	🟨
3	🟦
4	🟪

- Music Player UI
- Music Player
- Text Editor UI
- File Saver

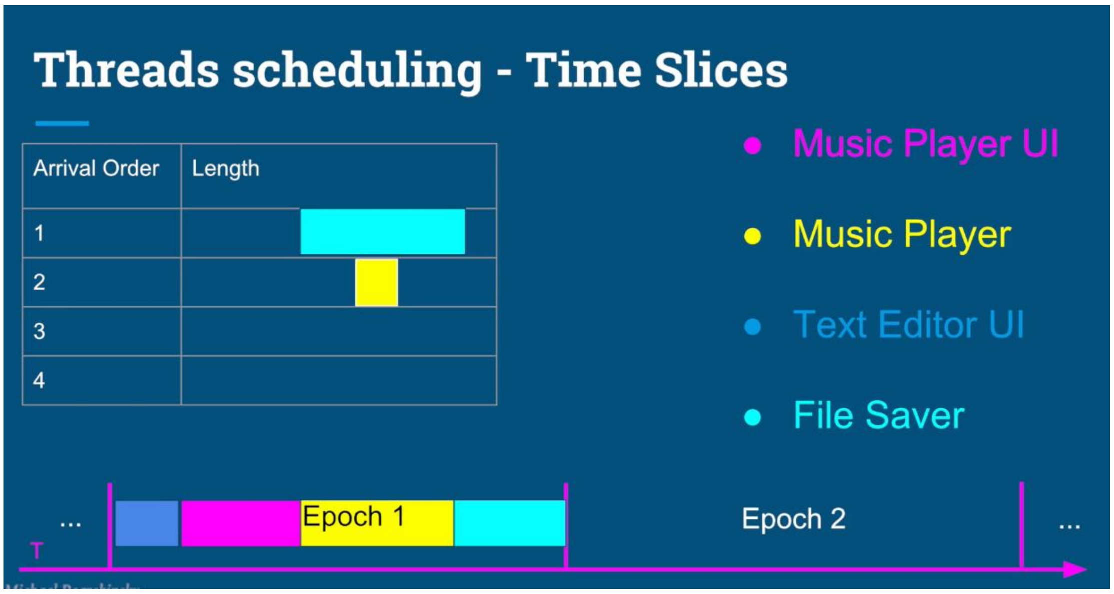
Who runs first??

So how about we simply schedule the tasks on the first come first serve basis.
That sounds fair.
Whoever came first should be executed first.
So we scheduled the file saver thread first, then the music player thread,
the text editor, UI thread, and the music player, UI thread in the end.

The obvious problem with this approach is if a very long thread
arrives first, it can cause what's called starvation for other threads.
This is a particularly big problem for you.

I threads this will make our applications unresponsive and our
users will have a terrible experience.
A common fact that you might've already noticed in the table is
that UI threads are usually shorter.
Normally they simply respond to an input from a user and update the screen.
So how about we schedule a shorter job first in this case, our scheduling
sequence would look like this.
However, this has an opposite problem.
There are user related events coming to our system all the time.
So if we keep scheduling the shortest job first, all the time, the longer tasks that involve computations will never be executed.
Okay.
So now after we ran those thought experiments and understand what kind
of trade offs and challenges the operating system needs to deal with
when fairly allocating the CPU time among threads, let's learn how itreally works in most operating systems.

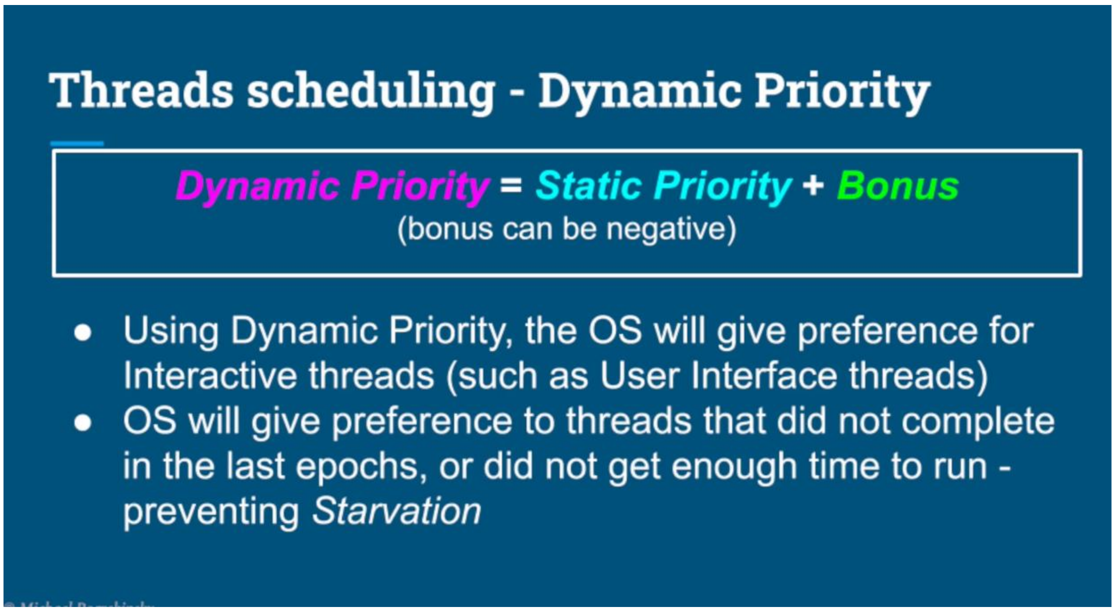
the operating system divides the time into moderately sized pieces called epochs.
In each epoch, the operating system allocates a different time slice for each thread. Notice that not all the threads get to run or complete in each epoch.



Threads scheduling - Time Slices

Arrival Order	Length
1	
2	
3	
4	

- Music Player UI
- Music Player
- Text Editor UI
- File Saver



Threads scheduling - Dynamic Priority

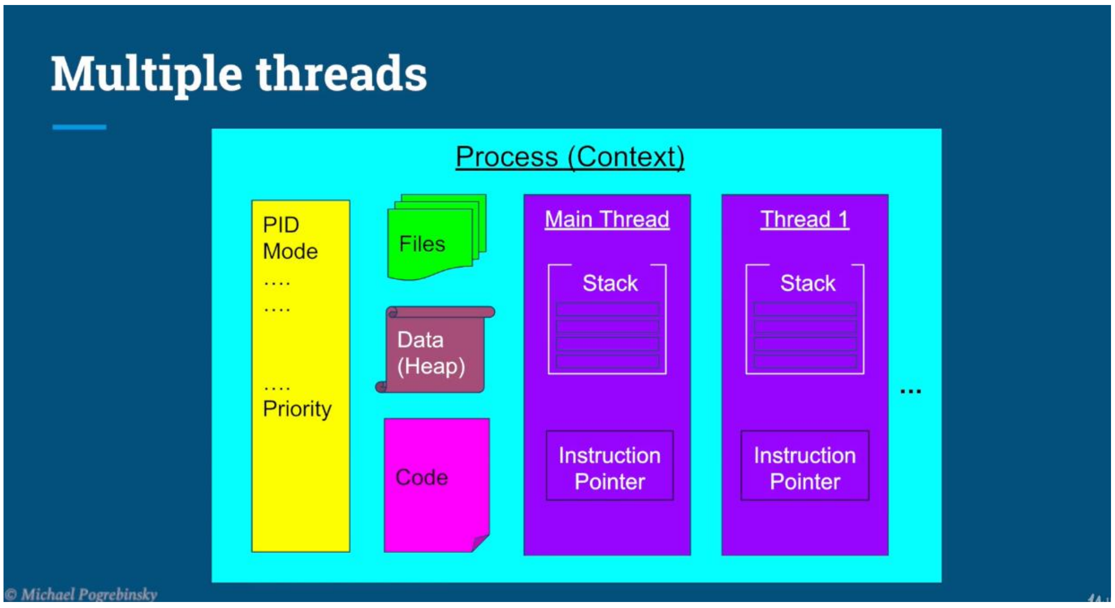
Dynamic Priority* = *Static Priority* + *Bonus
(bonus can be negative)

- Using Dynamic Priority, the OS will give preference for Interactive threads (such as User Interface threads)
- OS will give preference to threads that did not complete in the last epochs, or did not get enough time to run - preventing *Starvation*

The decision on how to allocate the time for each thread is based on a dynamic priority the operating system maintains for each thread.
The static priority is set by the developer ahead of time.
And the bonus is adjusted by the operating system.
In every epoch,for each thread, this way the operating system will give preference

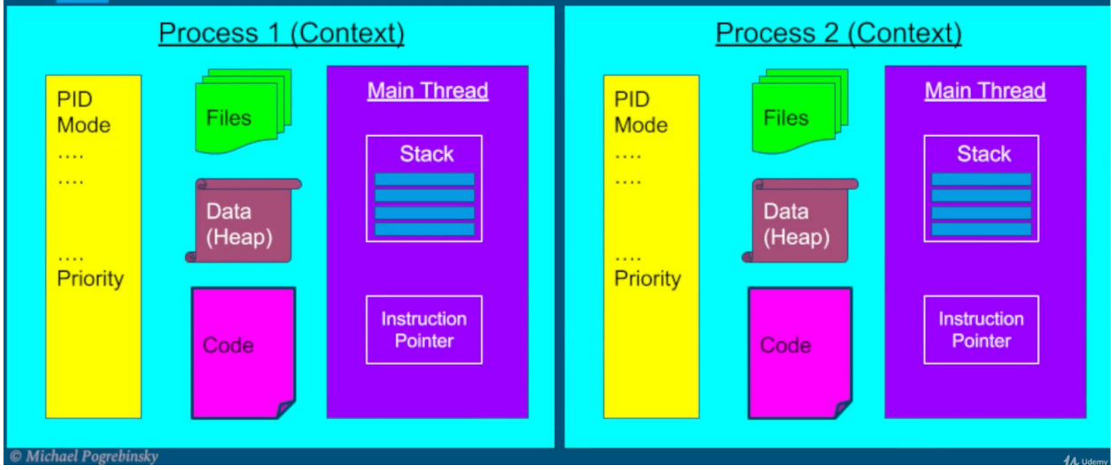
to interactive and real time threads that need more immediate attention.
And in the same time, it will give preference to computational threads
that did not complete, or did not get enough time to run in previous epoch to prevent starvation.

When to use multiple threads in a single program? And when to simply create a new program and run in a different process?



VS

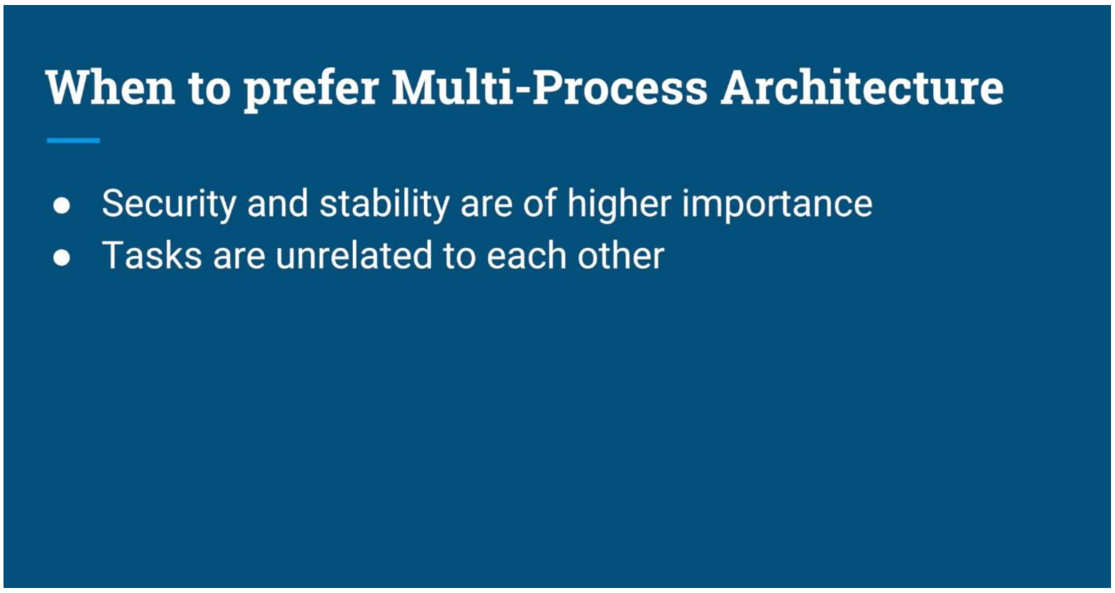
Multiple Processes



When to prefer Multithreaded Architecture

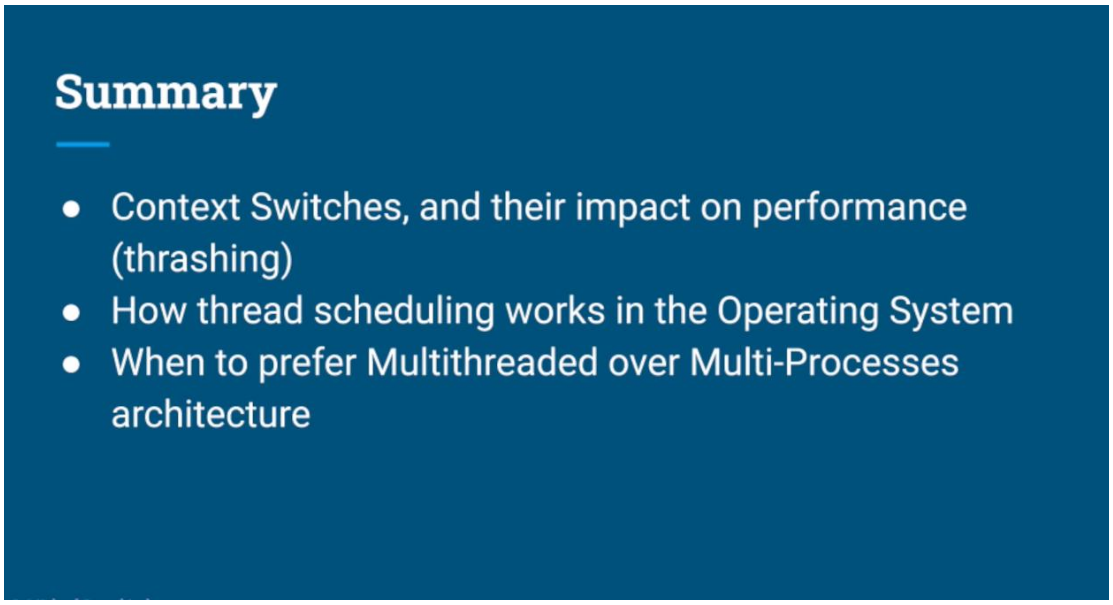
- Prefer if the tasks share a lot of data
- Threads are much faster to create and destroy
- Switching between threads of the same process is faster (shorter context switches)

VS



When to prefer Multi-Process Architecture

- Security and stability are of higher importance
- Tasks are unrelated to each other



Summary

- Context Switches, and their impact on performance (thrashing)
- How thread scheduling works in the Operating System
- When to prefer Multithreaded over Multi-Processes architecture