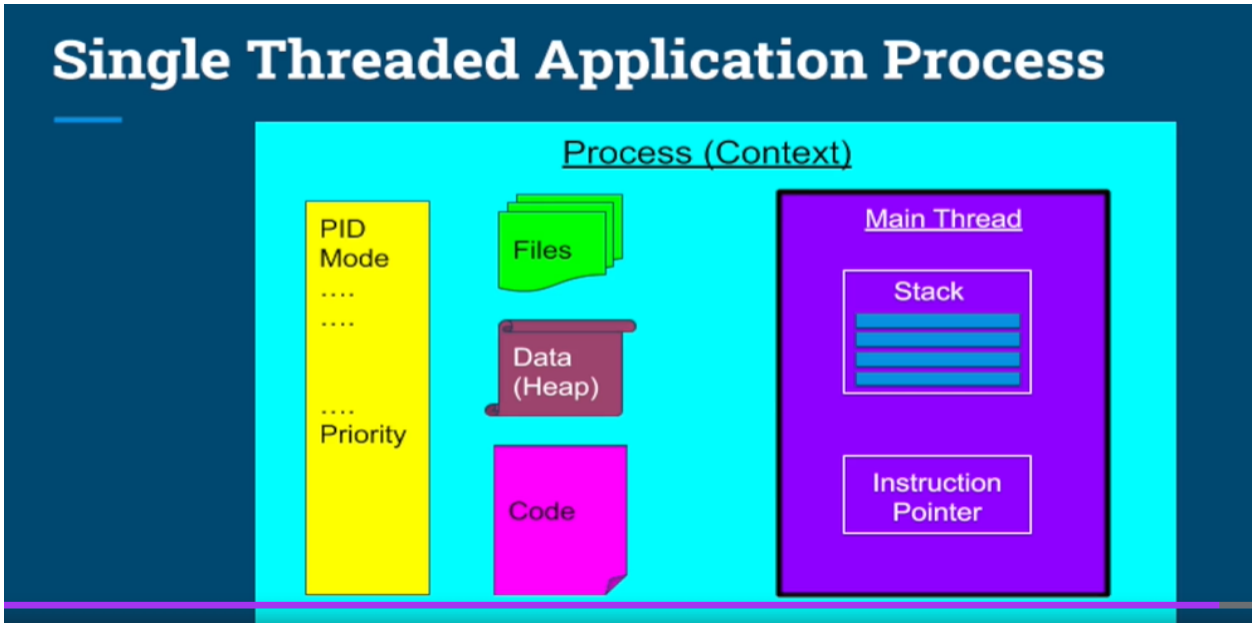
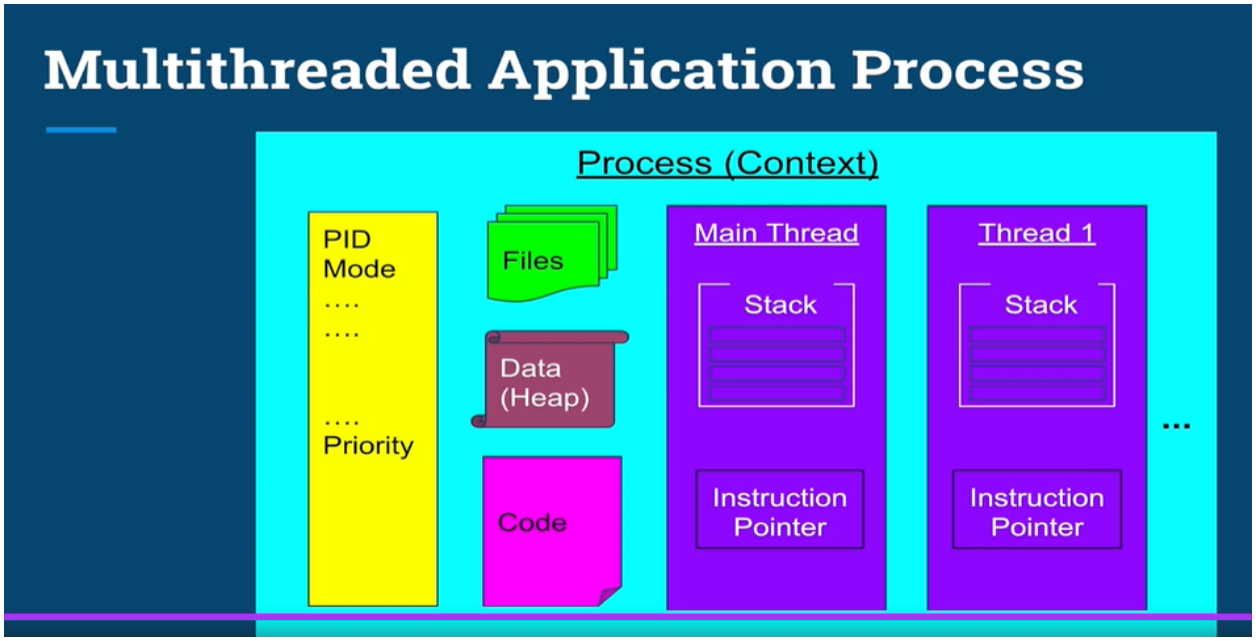
A few of the things that the process contains are the metadata, like the process ID, the files that the application opens for reading and writing the code, which is the program instructions that are going to be executed on the CPU, the heap, which contains all the data, our application needs. And finally, at least one thread called the main thread.
The thread contains two main things, the stack, and the instruction pointer.

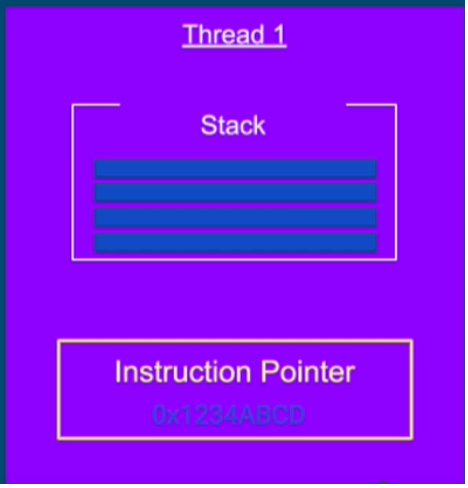


In a multithreaded obligation. Each thread comes with its own stack and its own instruction pointer, but all the rest of the components in the process are shared by all threats.

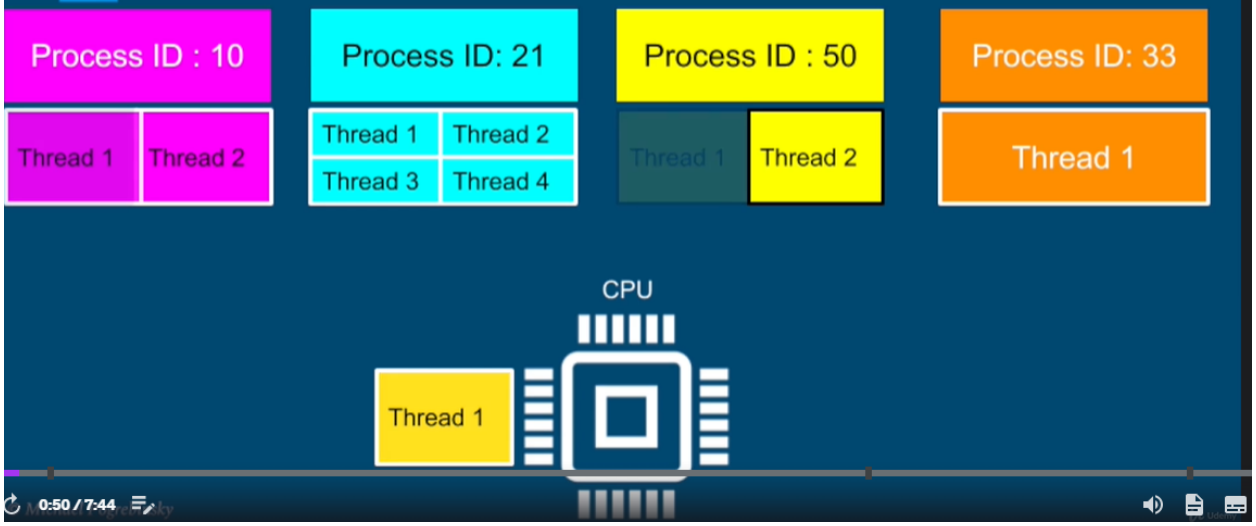


What the thread contains

- Stack - Region in memory, where local variables are stored, and passed into functions
- Instruction Pointer - Address of the next instruction to execute

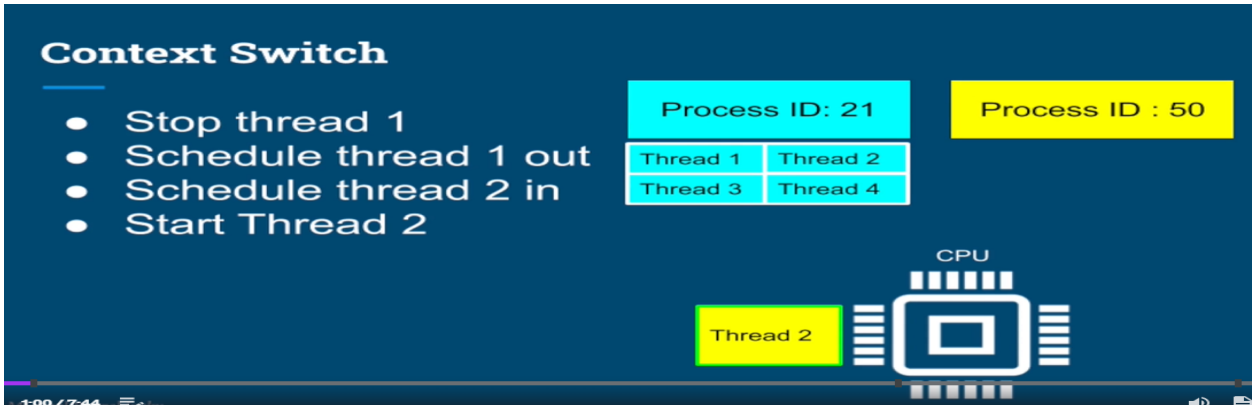


Context Switch



Context Switch

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

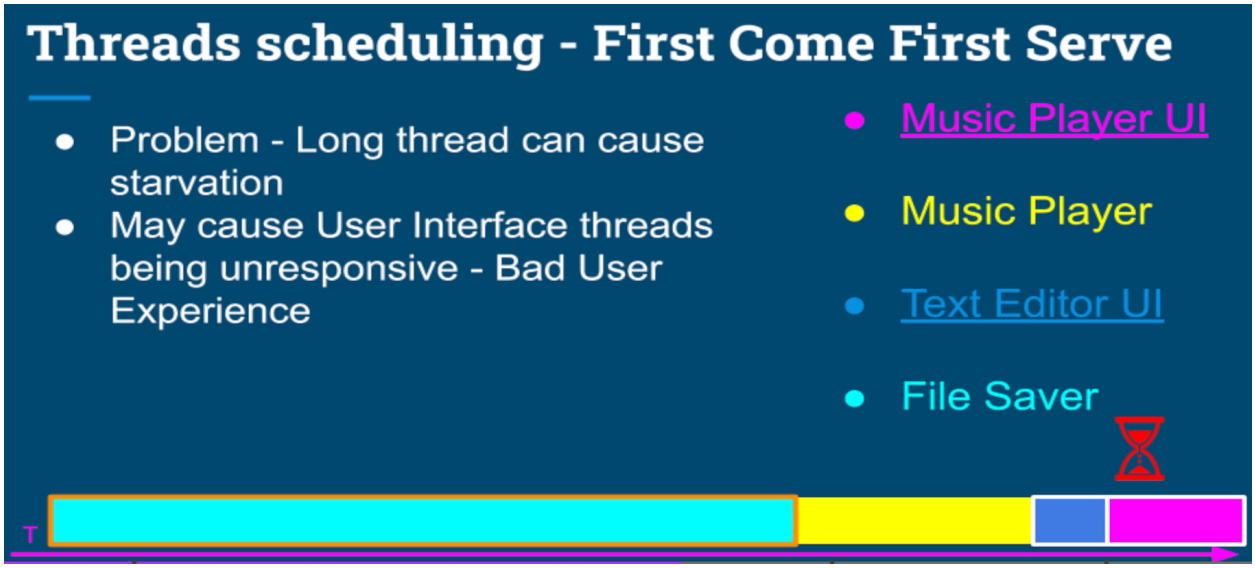


Context Switch Cost

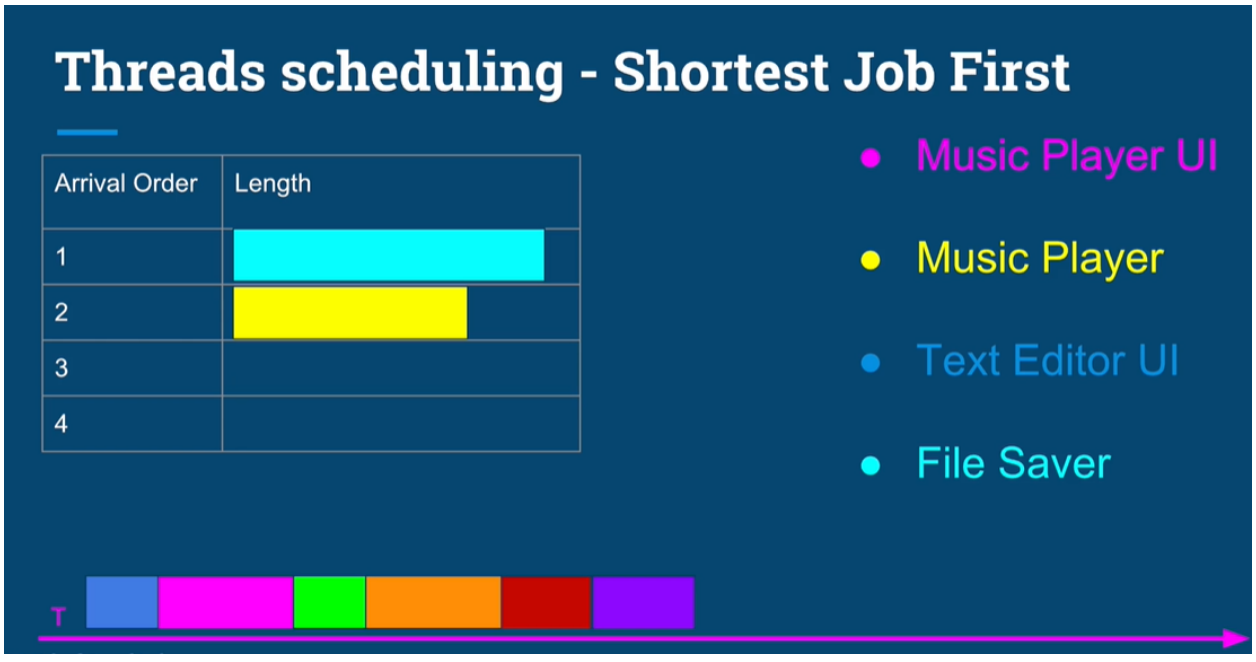
- Context switch is not cheap, and is the price of multitasking (concurrency)
- Same as we humans when we multitask - Takes time to focus
- Each thread consumes resources in the CPU and memory
- When we switch to a different thread:
  - Store data for one thread
  - Restore data for another thread

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

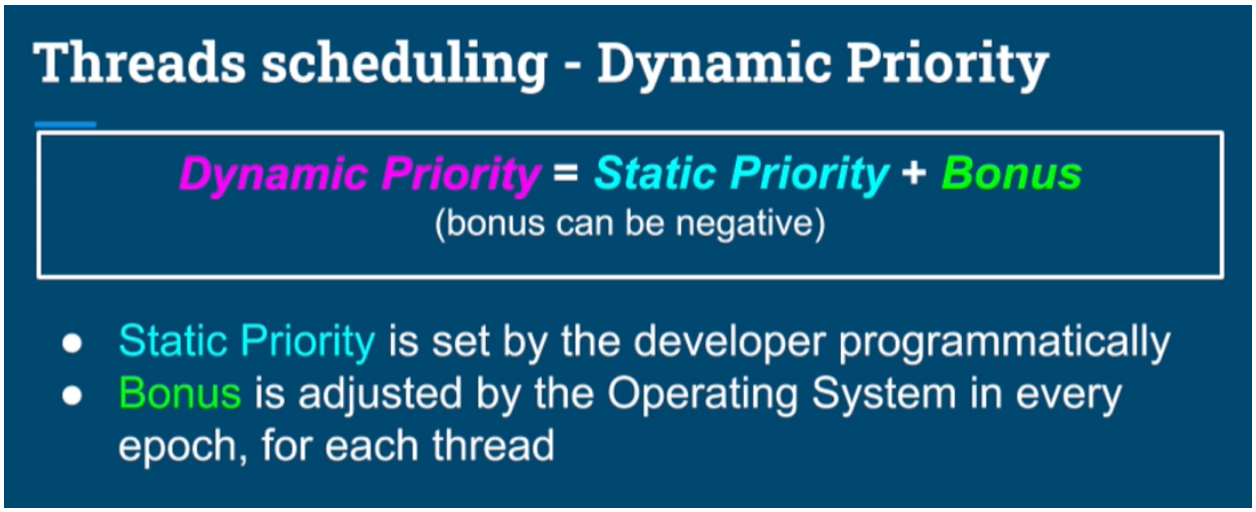


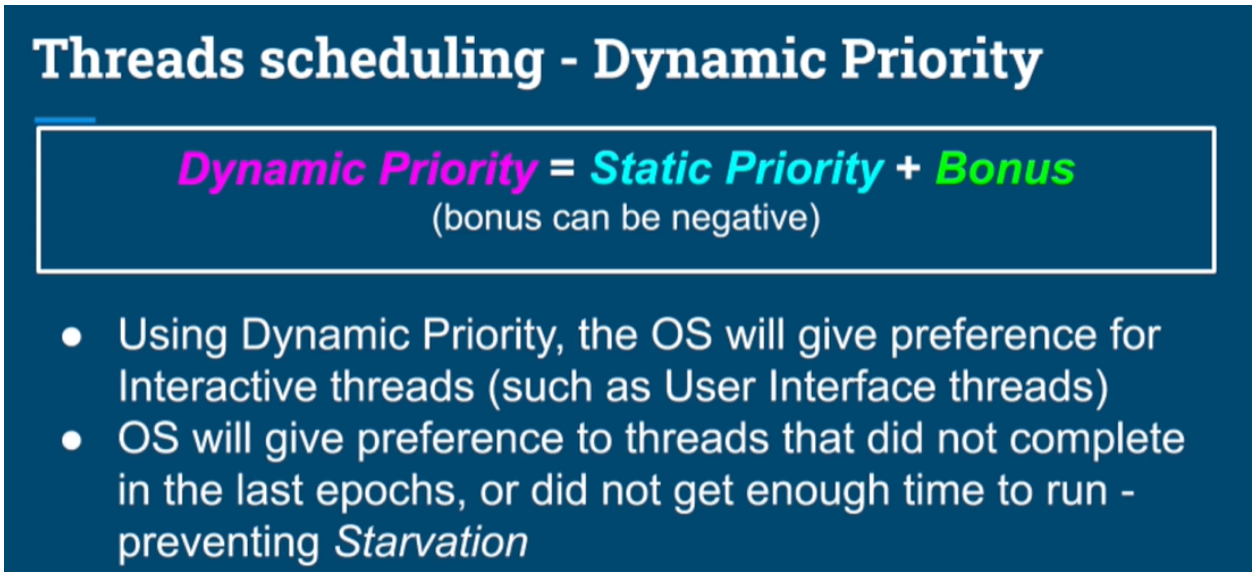
The obvious problem with this approach is if a very long thread arrives first, it can cause what's called starvation for other threads. This is a particularly big problem for you.
I threads this will make our applications unresponsive and our users will have a terrible experience.



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.

In general, 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. The decision on how to allocate the time for each thread is based on a dynamic priority. The operating system maintains for each thread.





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)

When to prefer Multi-Process Architecture

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

Multiple threads in a single process share:

- **The Heap**
- **Code**
- **The process's open files**
- **The process's metadata**