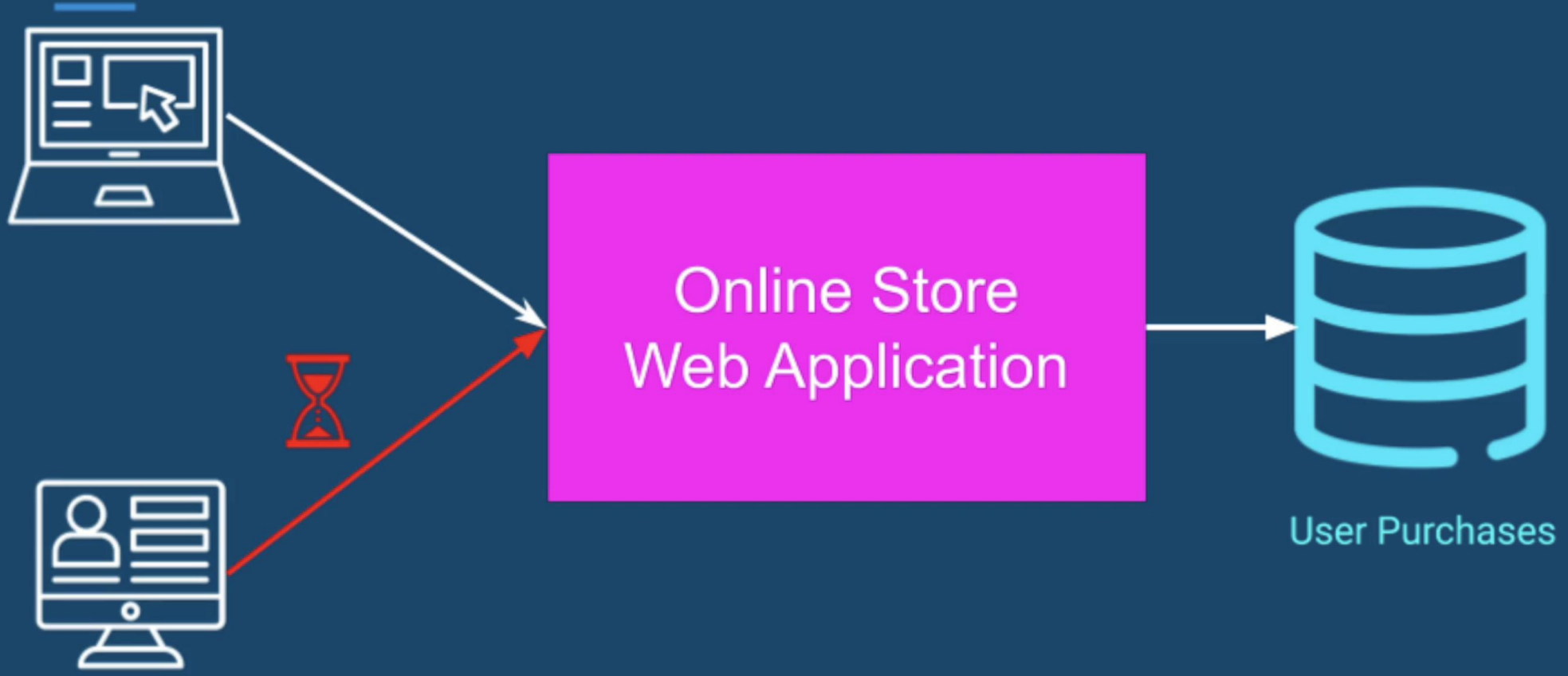
Motivation - Why we need Threads?

1. Responsiveness
2. Performance

1. Examples of Poor Responsiveness

- Waiting for Customer Support
- Late response from a person
- No feedback from an application

1. Responsiveness with a Single Thread



1. Responsiveness with a Multithreading

1. Responsiveness in User Interface

- Responsiveness is particularly critical in applications with a User Interface

1. Concurrency - Multitasking

- Responsiveness can be achieved by using multiple threads, with a separate thread for each task
- Generally very hard to achieve otherwise

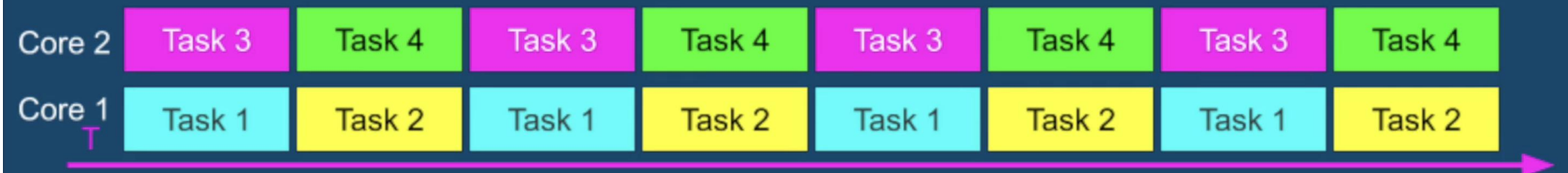
1. Concurrency - Multitasking

- Note : We don't need multiple cores to achieve *concurrency*

Single Core

2. Performance

- We can create an illusion of multiple tasks executing in parallel using just a single core
- With multiple cores we can truly run tasks completely in parallel



2. Performance - Impact

- Completing a complex task much faster
- Finish more work in the same period of time
- For high scale service -
  - Fewer machines
  - Less money spent on hardware
  - More money in your pocket

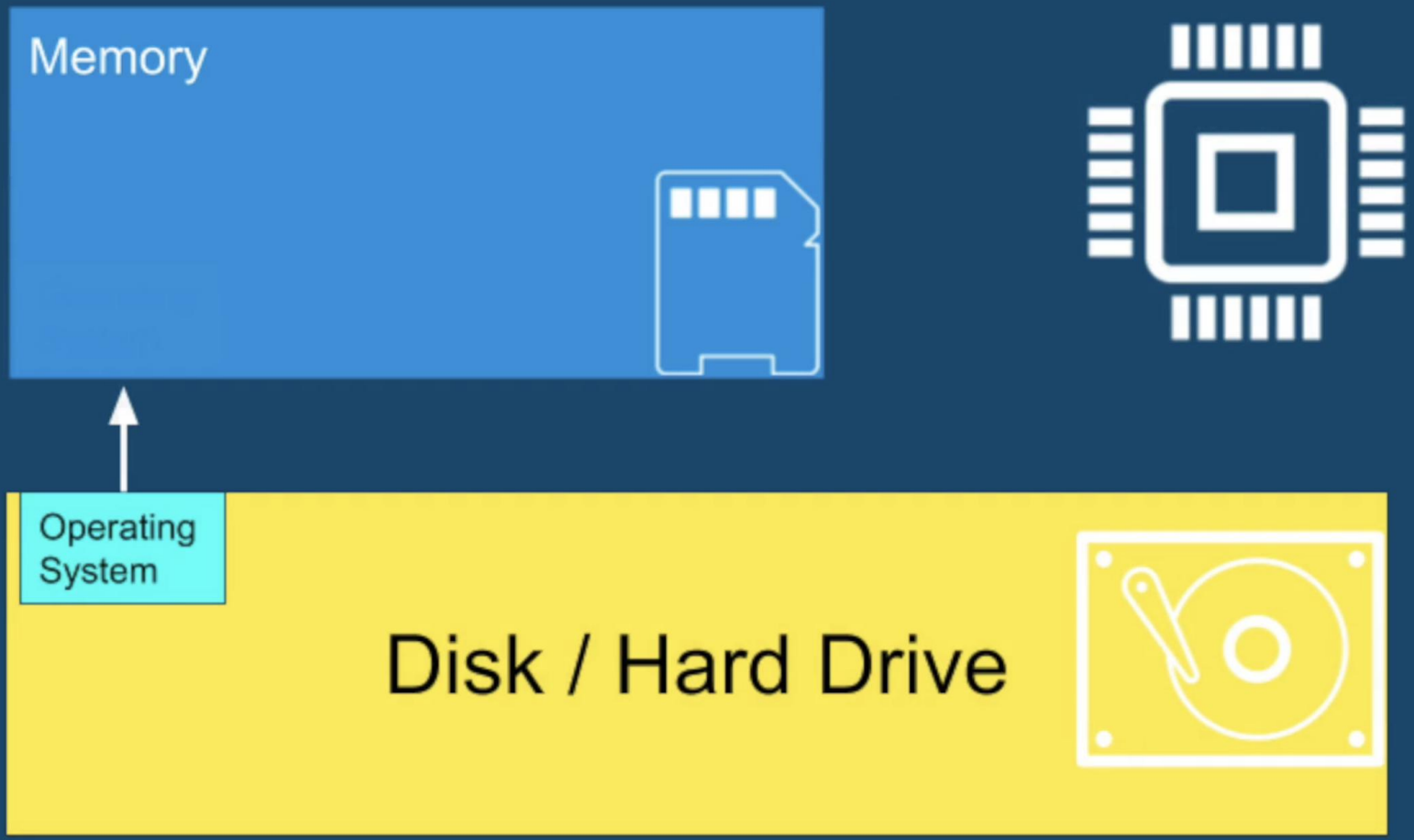
Multithreading Caveat

- Multithreaded programming is fundamentally different from single threaded programming
- In this course we will lay the groundwork, and learn all the tools to become a successful multithreaded programming developer

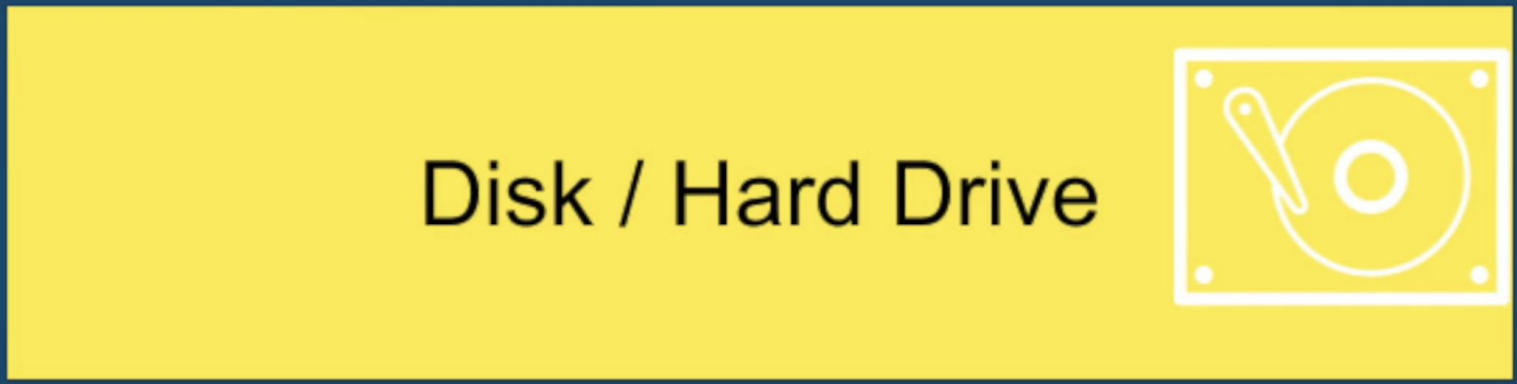
What we learn in this lecture

- Motivation - Concurrency and Parallelism
- What threads are- Introduction to OS

What threads are and where they live

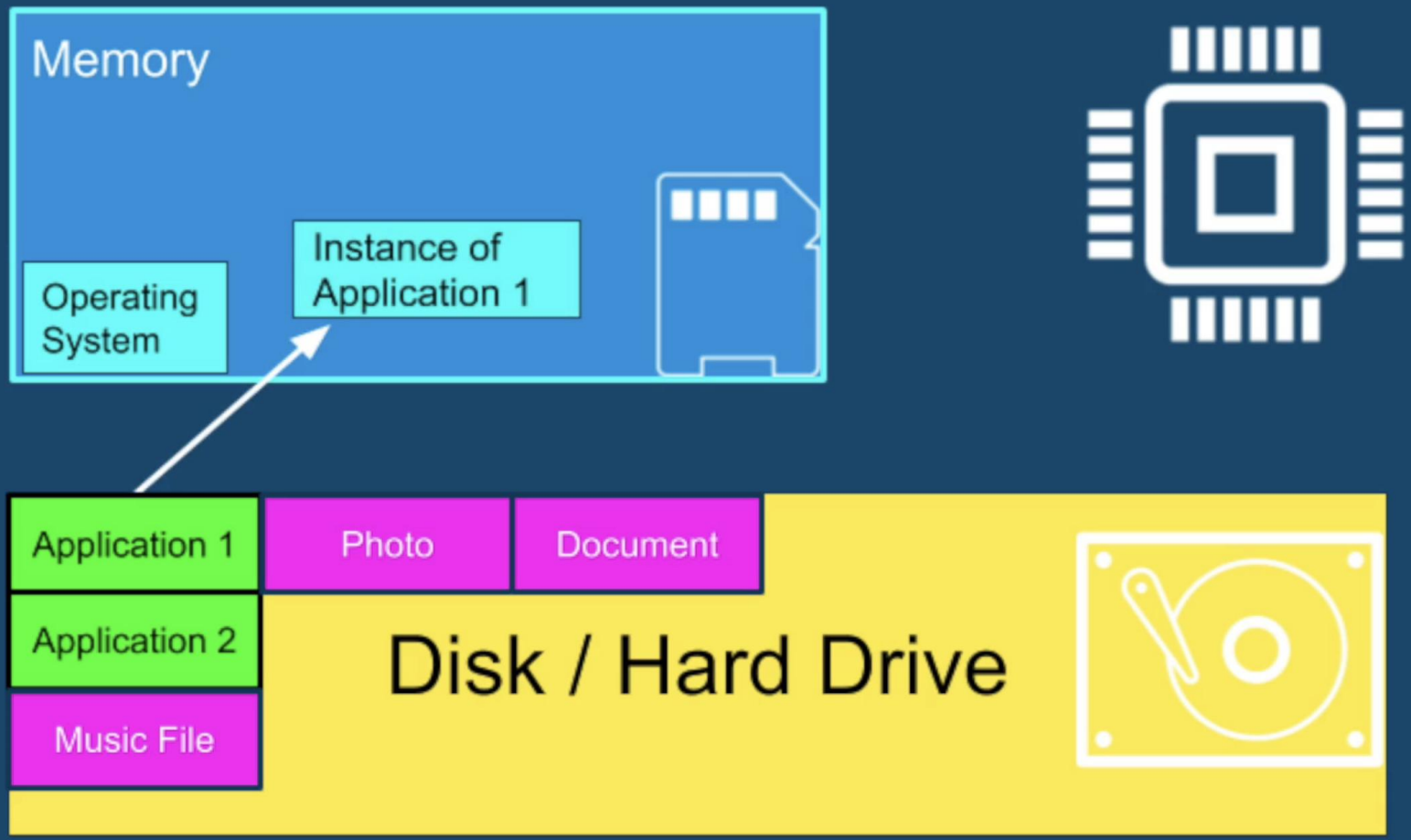


What threads are and where they live

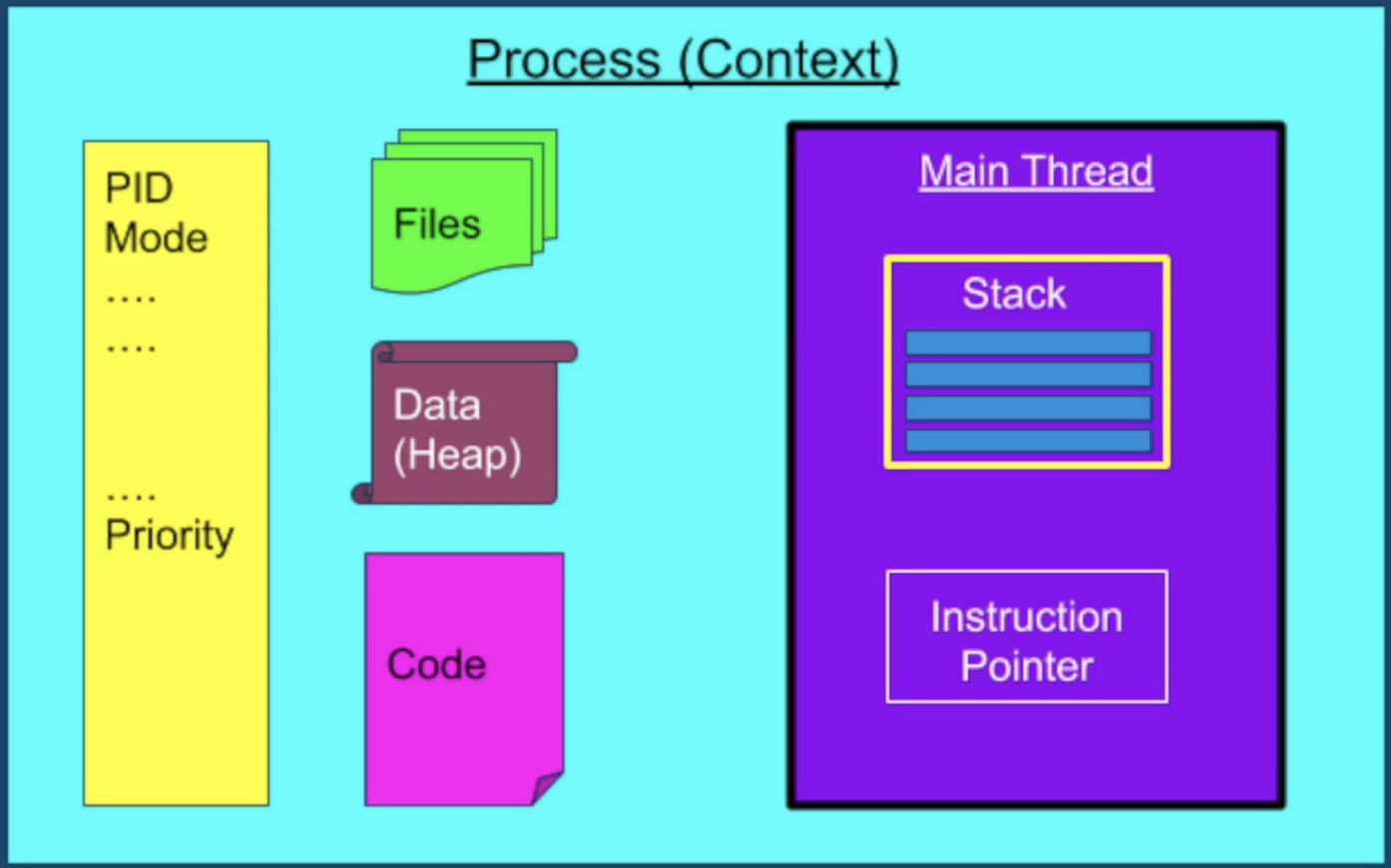


What threads are and where they live

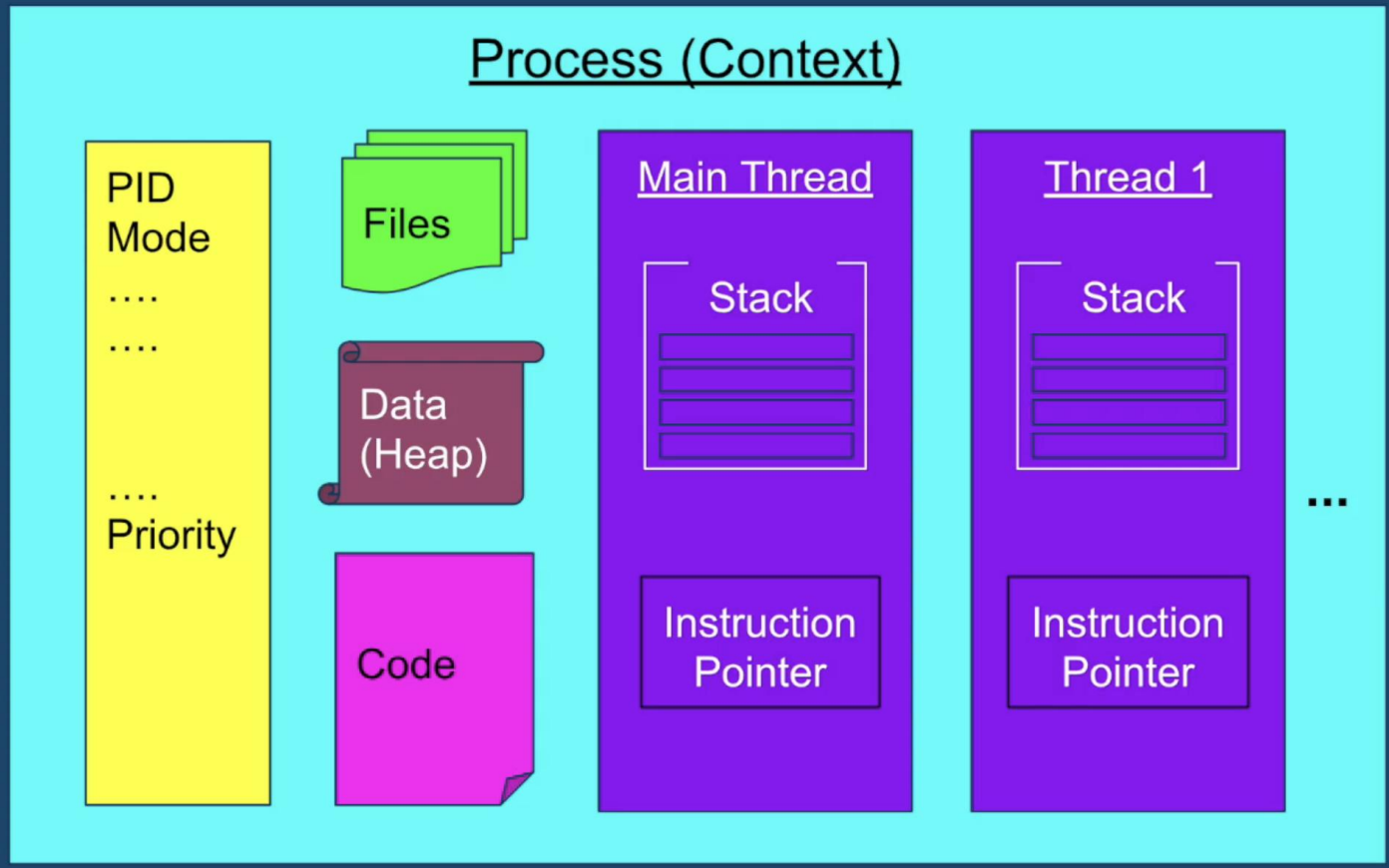
What threads are and where they live



Single Threaded Application Process

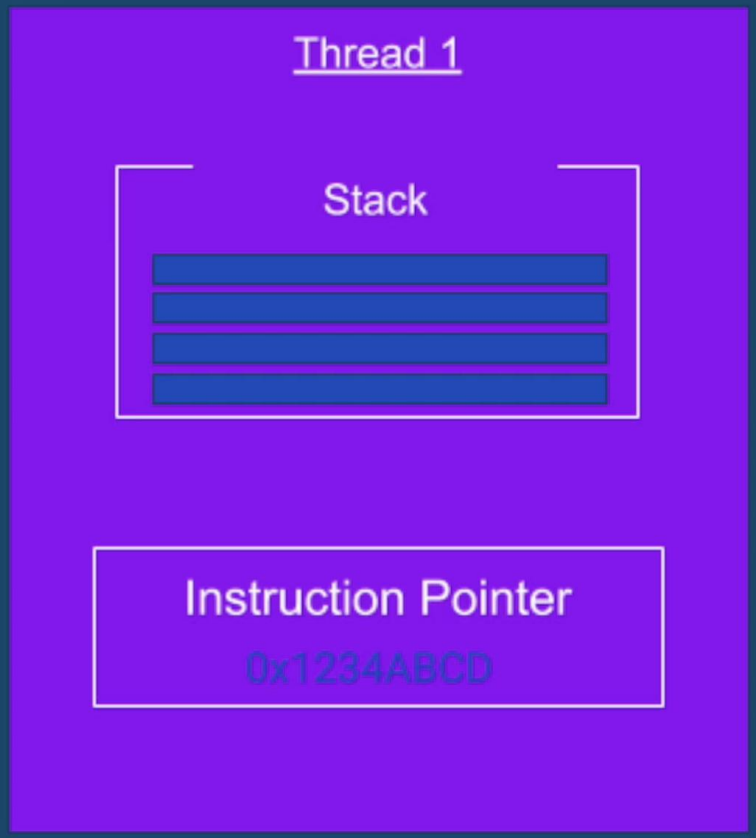


Multithreaded Application Process



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



Summary

- Motivation for multithreading
  - Responsiveness achieved by *concurrency*
  - Performance achieved by parallelism
- Threads are and what they contain
  - Stack
  - Instruction pointer
- What threads share
  - Files
  - Heap
  - Code

What we learn in this lecture

- Context switch
- Thread scheduling
- Threads vs Processes

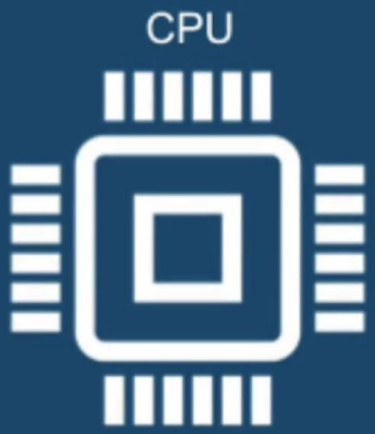
Context Switch

Process ID : 10

Process ID: 21

Process ID : 50

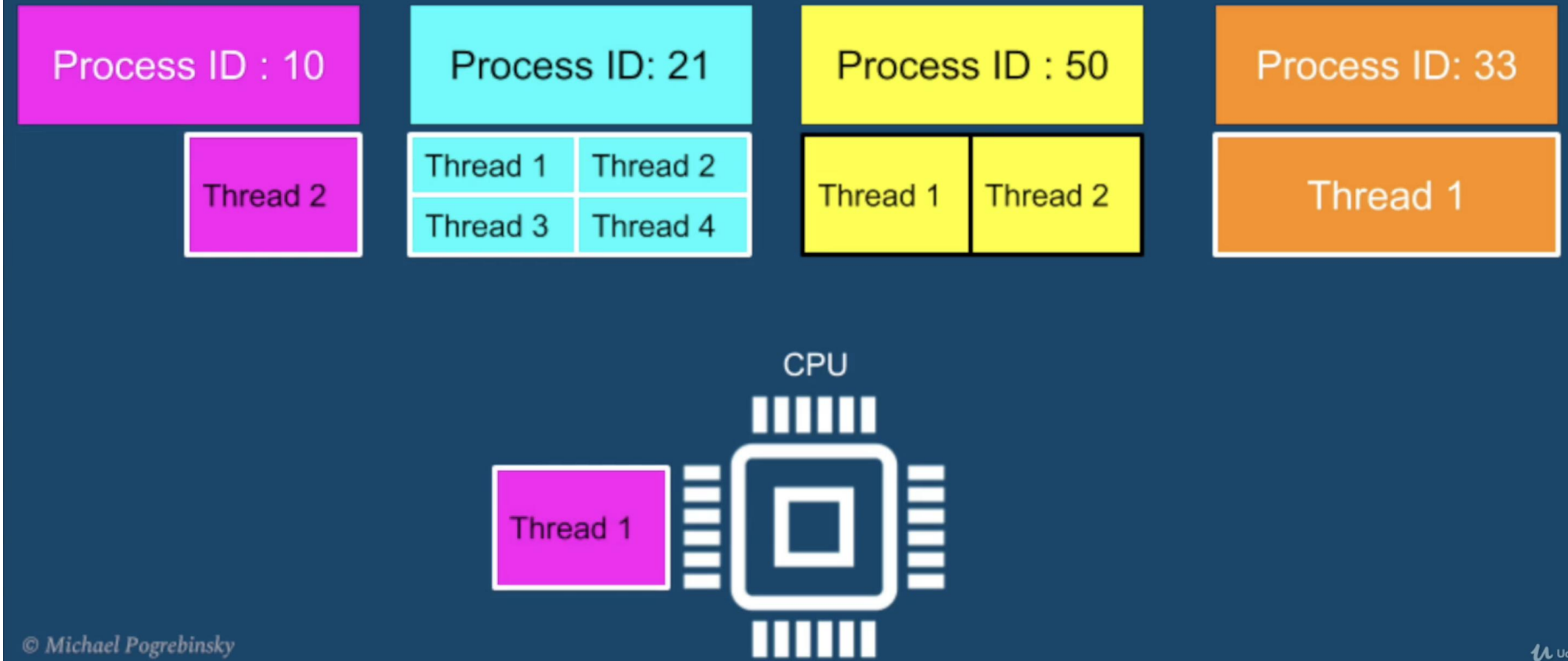
Process ID: 33



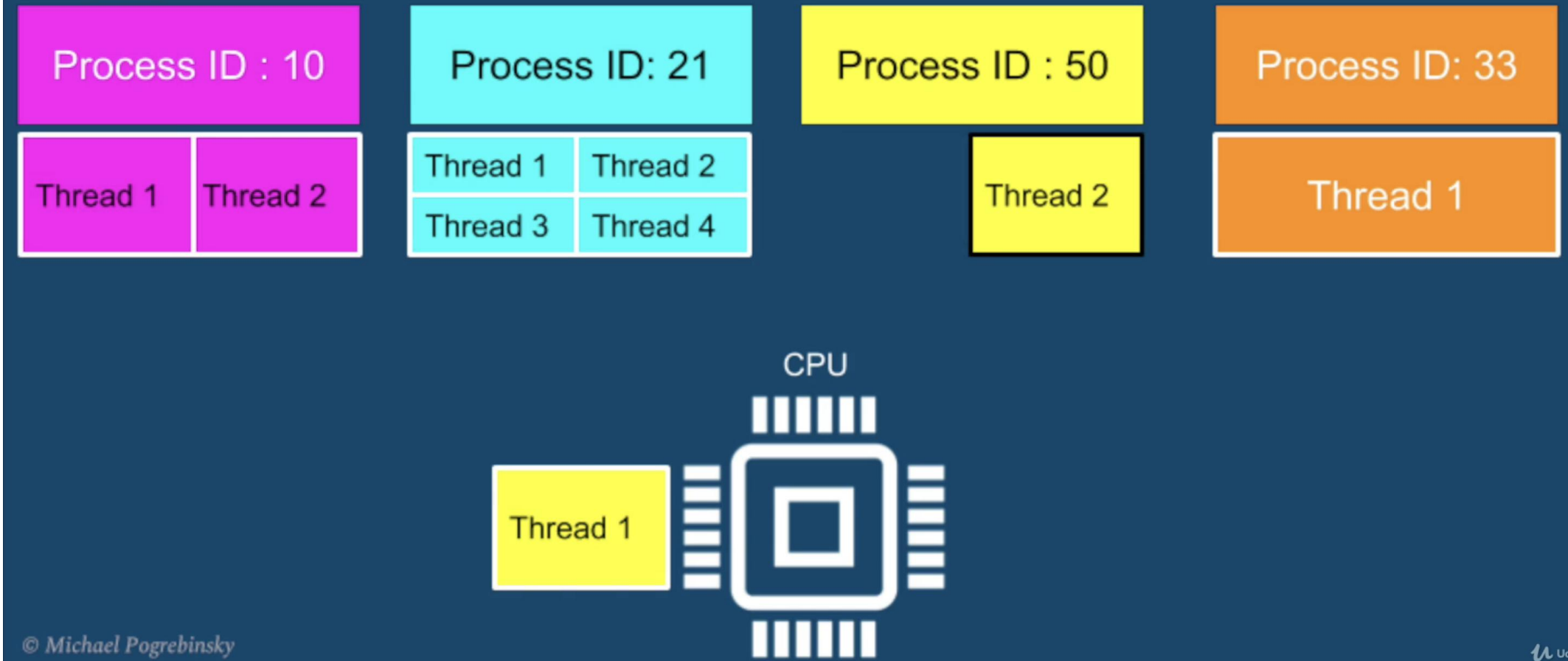
Context Switch

Process ID : 10	Process ID: 21	Process ID : 50	Process ID: 33
Thread 1	Thread 2	Thread 1	Thread 2	Thread 1	Thread 2	Thread 1
	Thread 3	Thread 4			

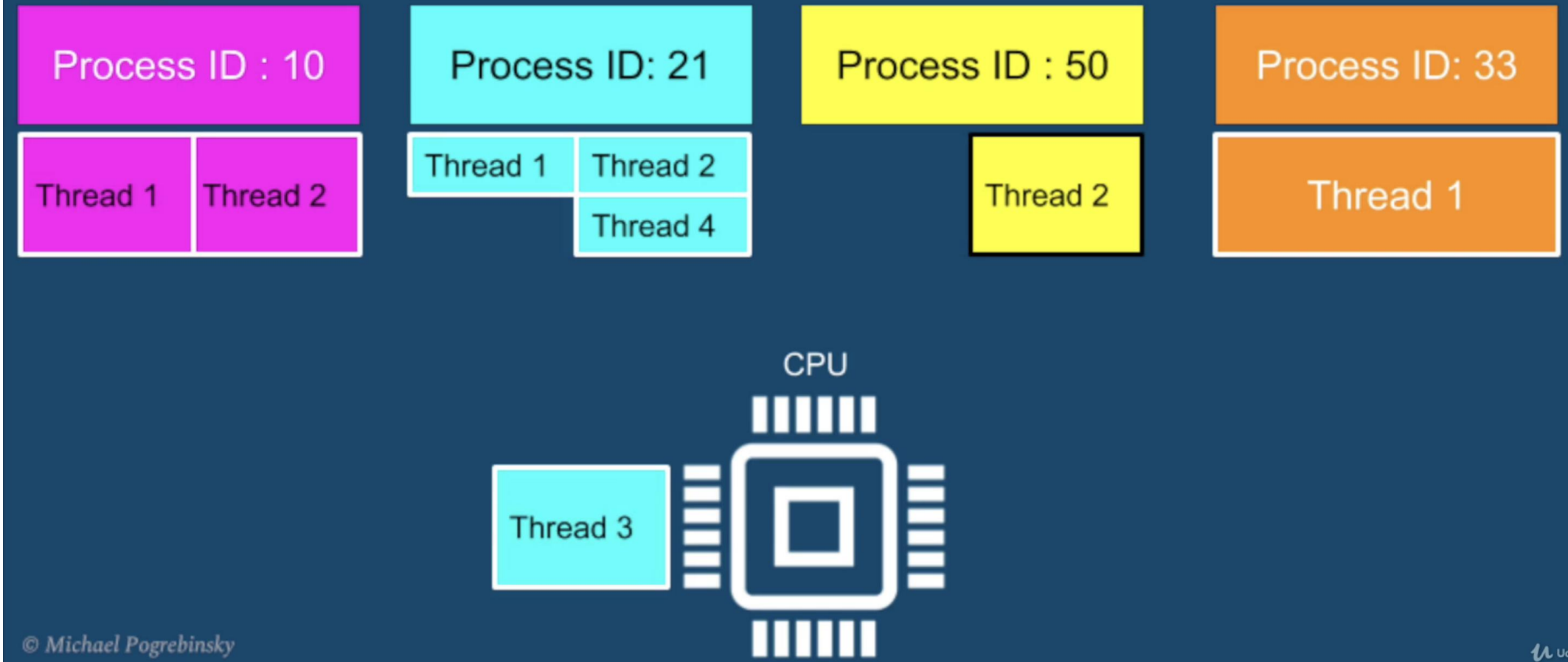
Context Switch



Context Switch

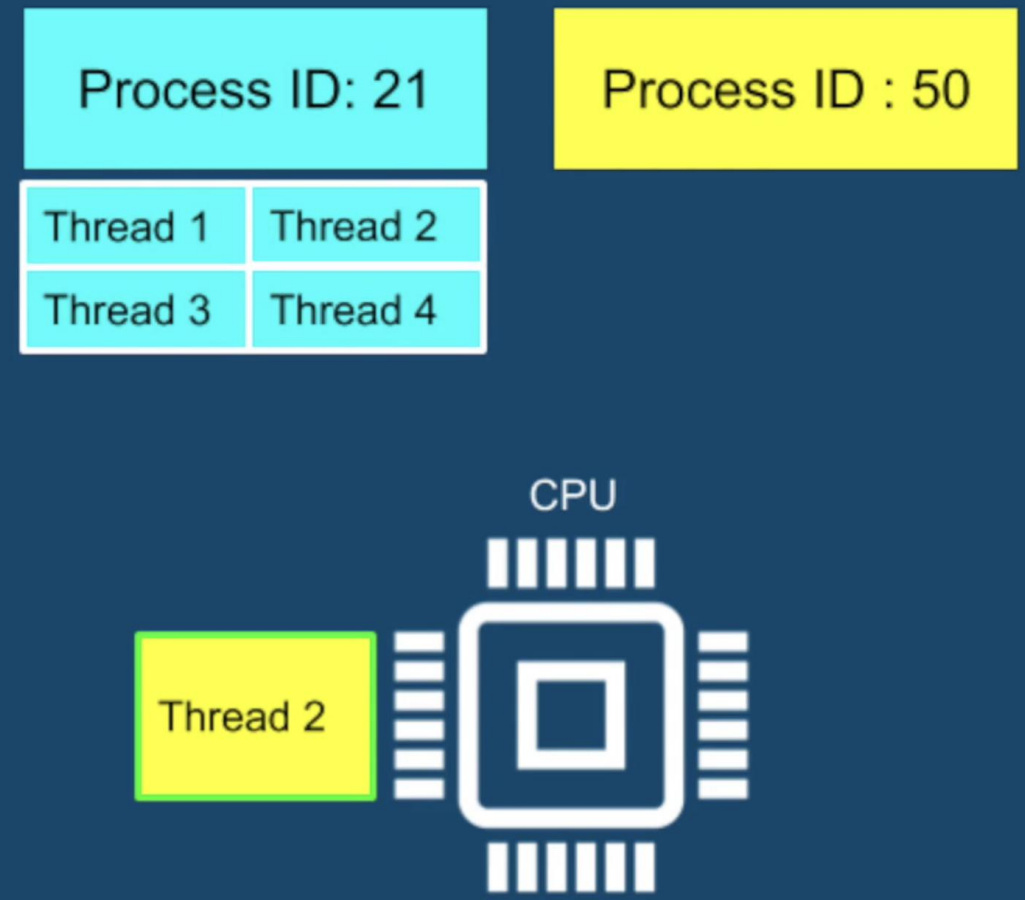


Context Switch



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

What we learn in this lecture

- Context switch
- Thread scheduling
- Threads vs Processes

Thread scheduling

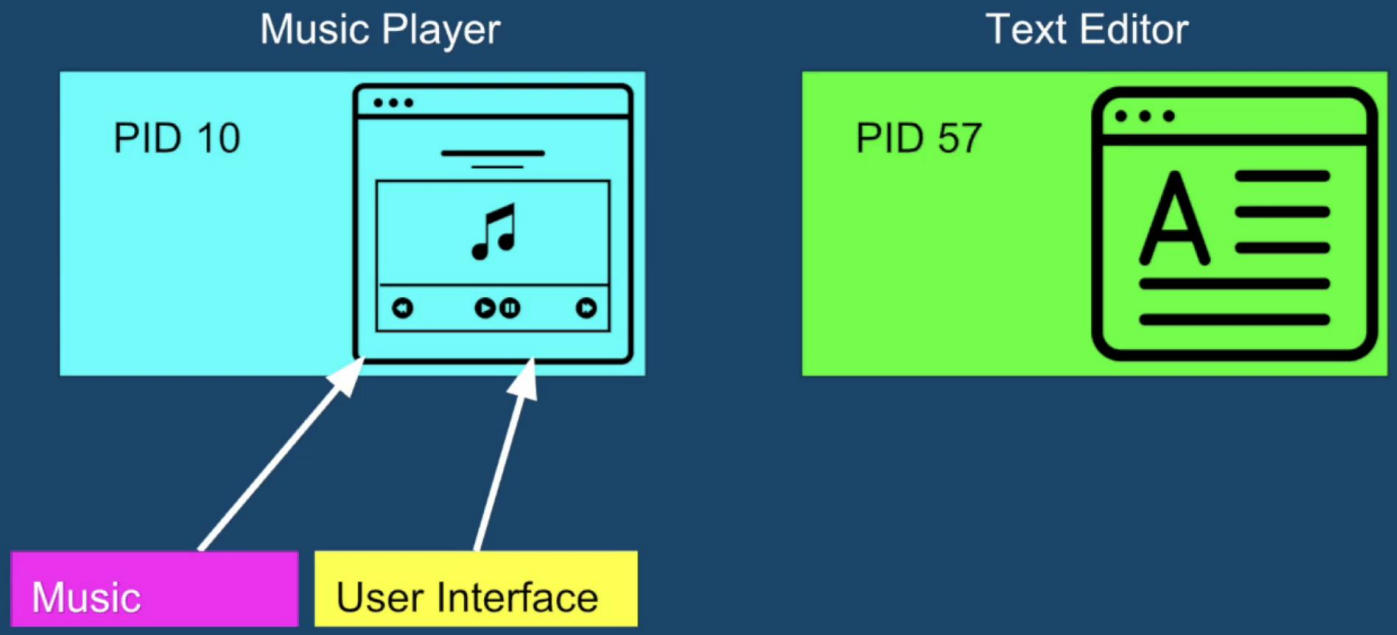
Music Player

PID 10

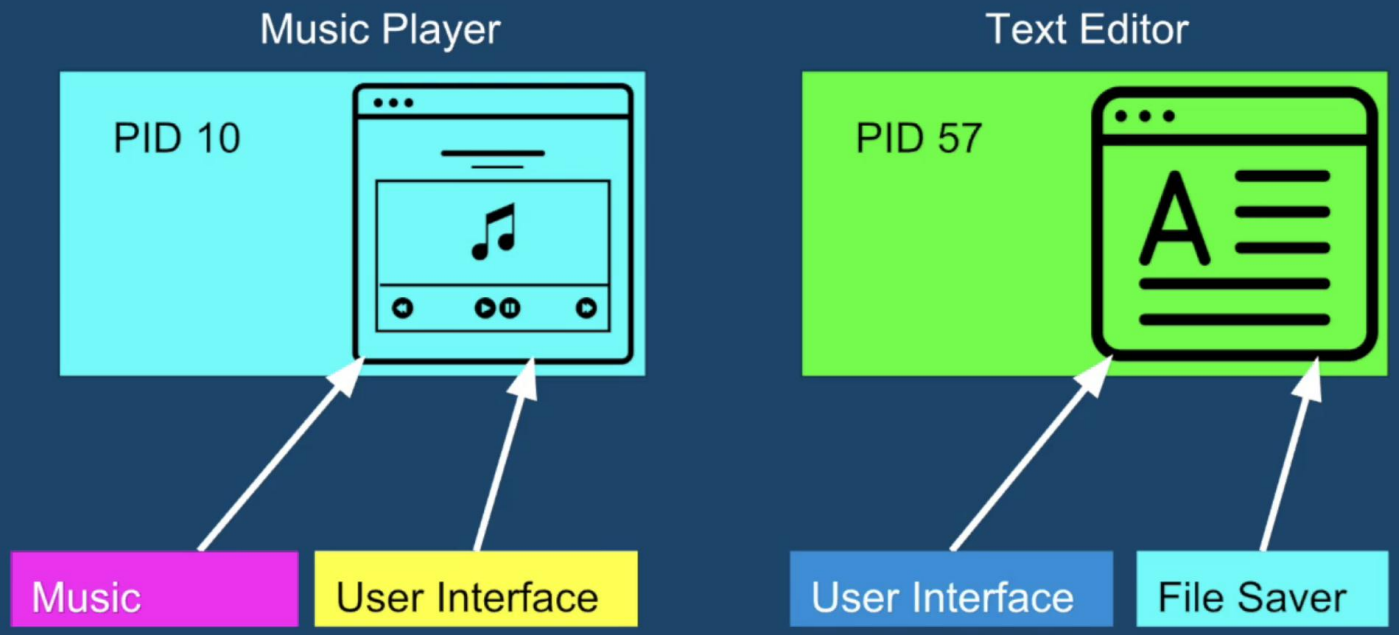
Text Editor

PID 57

Thread scheduling

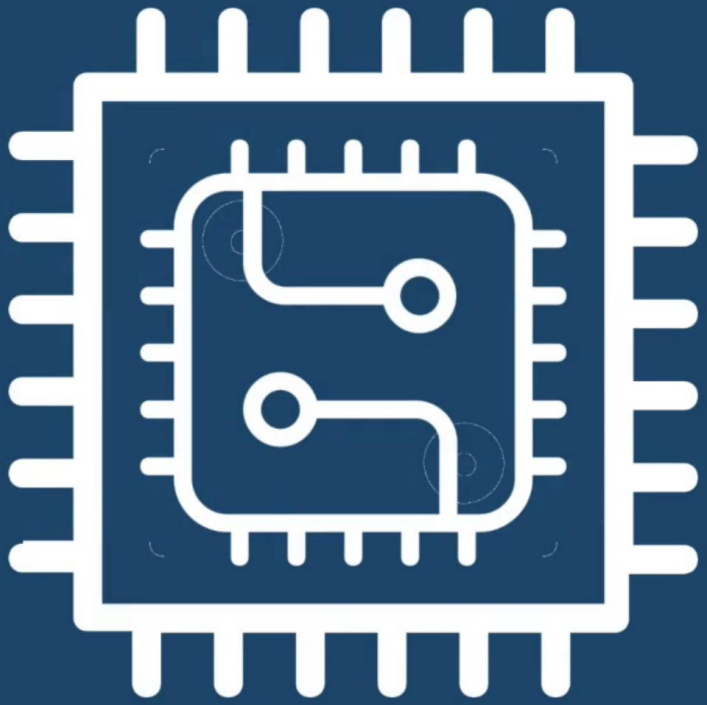


Thread scheduling



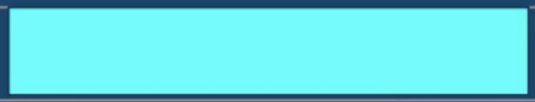
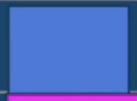
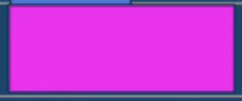
Threads scheduling

CPU - Single Core



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

Threads scheduling

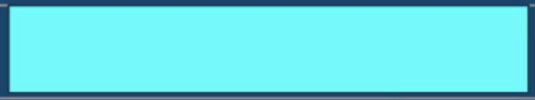
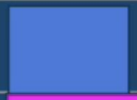
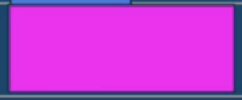
Arrival Order	Length
1	
2	
3	
4	

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

Who runs first??

T

Threads scheduling - First Come First Serve

Arrival Order	Length
1	
2	
3	
4	

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

T

Threads scheduling - First Come First Serve

Arrival Order	Length
1	
2	
3	
4	

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

Threads scheduling - First Come First Serve

- Problem - Long thread can cause starvation

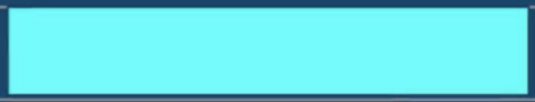
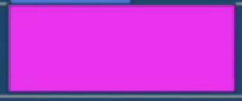
- Music Player UI
- Music Player
- Text Editor UI
- File Saver

Threads scheduling - First Come First Serve

- Problem - Long thread can cause starvation
- May cause User Interface threads being unresponsive - Bad User Experience

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

Threads scheduling - Shortest Job First

Arrival Order	Length
1	
2	
3	
4	

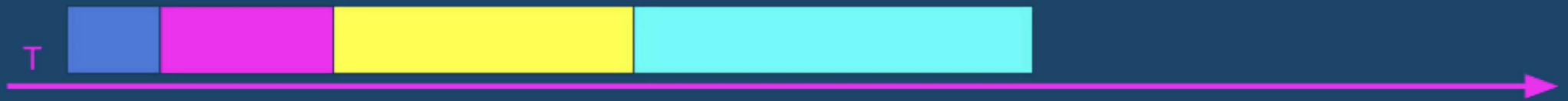
- Music Player UI
- Music Player
- Text Editor UI
- File Saver

T

Threads scheduling - Shortest Job First

Arrival Order	Length
1	
2	
3	
4	

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



Threads scheduling - Shortest Job First

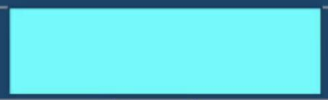
Arrival Order	Length
1	
2	
3	
4	

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

Thread Scheduling

- Now we understand the tradeoffs and challenges with scheduling threads
- Let’s learn how it really works

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)

- Static Priority is set by the developer programmatically
- Bonus is adjusted by the Operating System in every epoch, for each thread

Threads scheduling - Dynamic Priority

Dynamic Priority **=** ***Static Priority*** **+** ***Bonus***

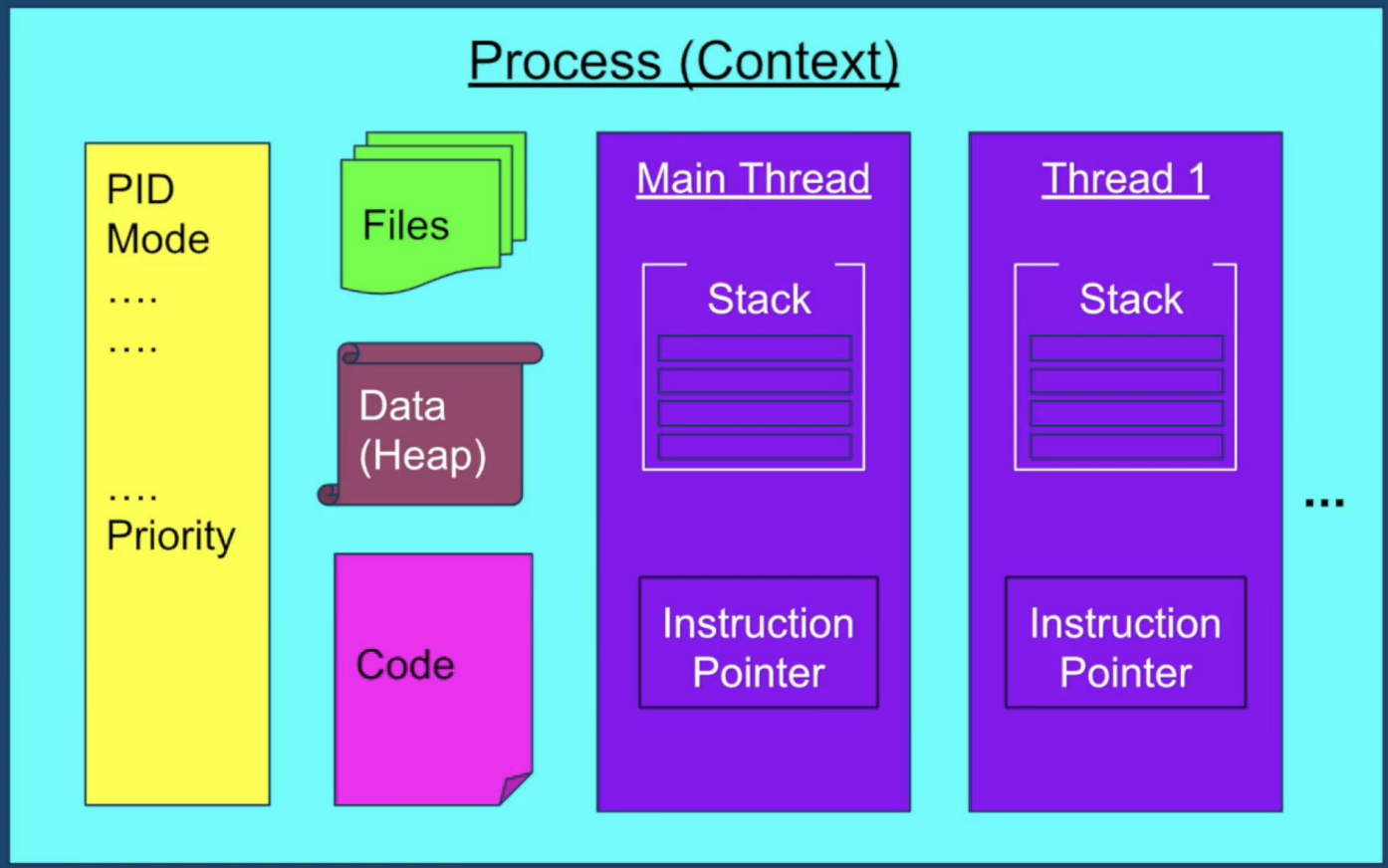
(bonus can be negative)

- 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*

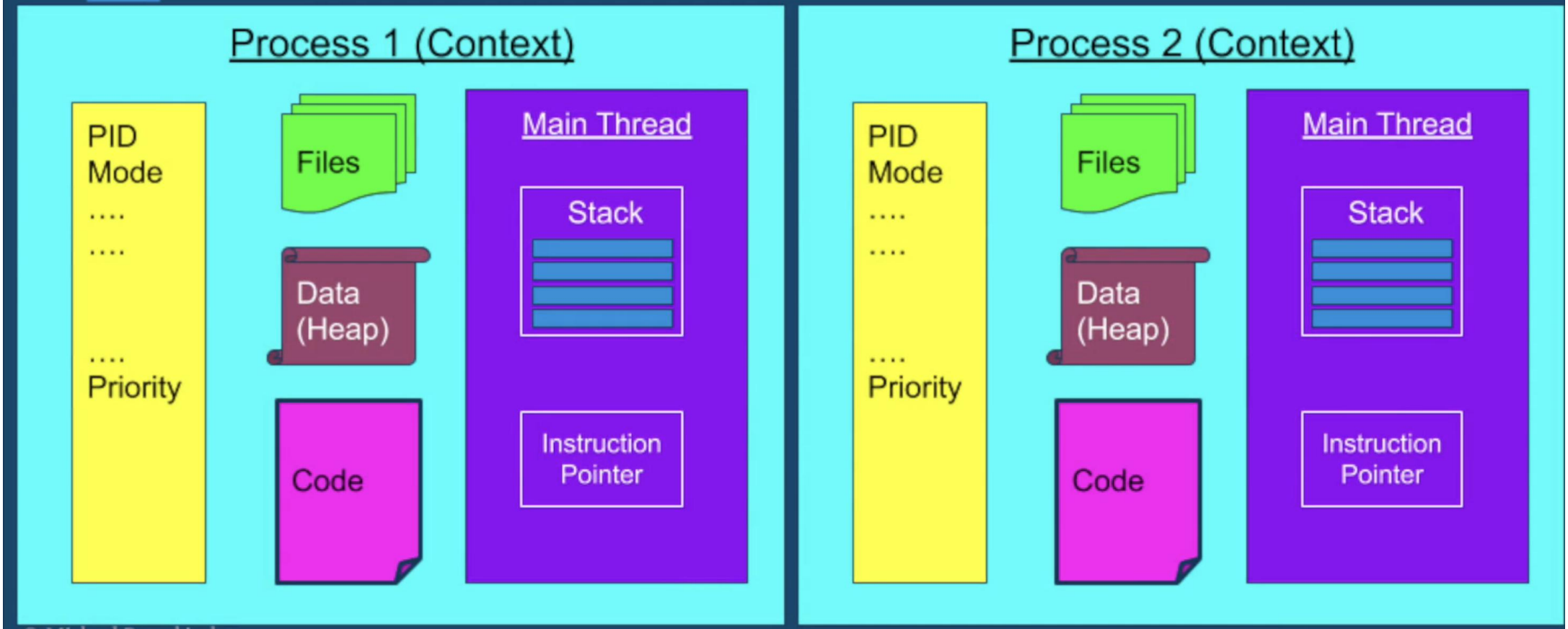
What we learn in this lecture

- Context switch
- Thread scheduling
- Threads vs Processes

Multiple threads



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)

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

What we learn in this lecture

- Thread creation with java.lang.Runnable
- Thread class capabilities
- Thread Debugging

What we learn in this lecture

- Thread creation with java.lang.Thread
- Case study - interactive multithreaded application

Summary

- Thread class - Encapsulates all thread related functionality
- Two ways to run code on a new thread
  a. Implement *Runnable* interface, and pass to a new Thread object
  b. Extend *Thread* class, and create an object of that class
- Both ways are equally correct

What we learn in this lecture

- Thread termination
- Thread.interrupt()
- Daemon threads

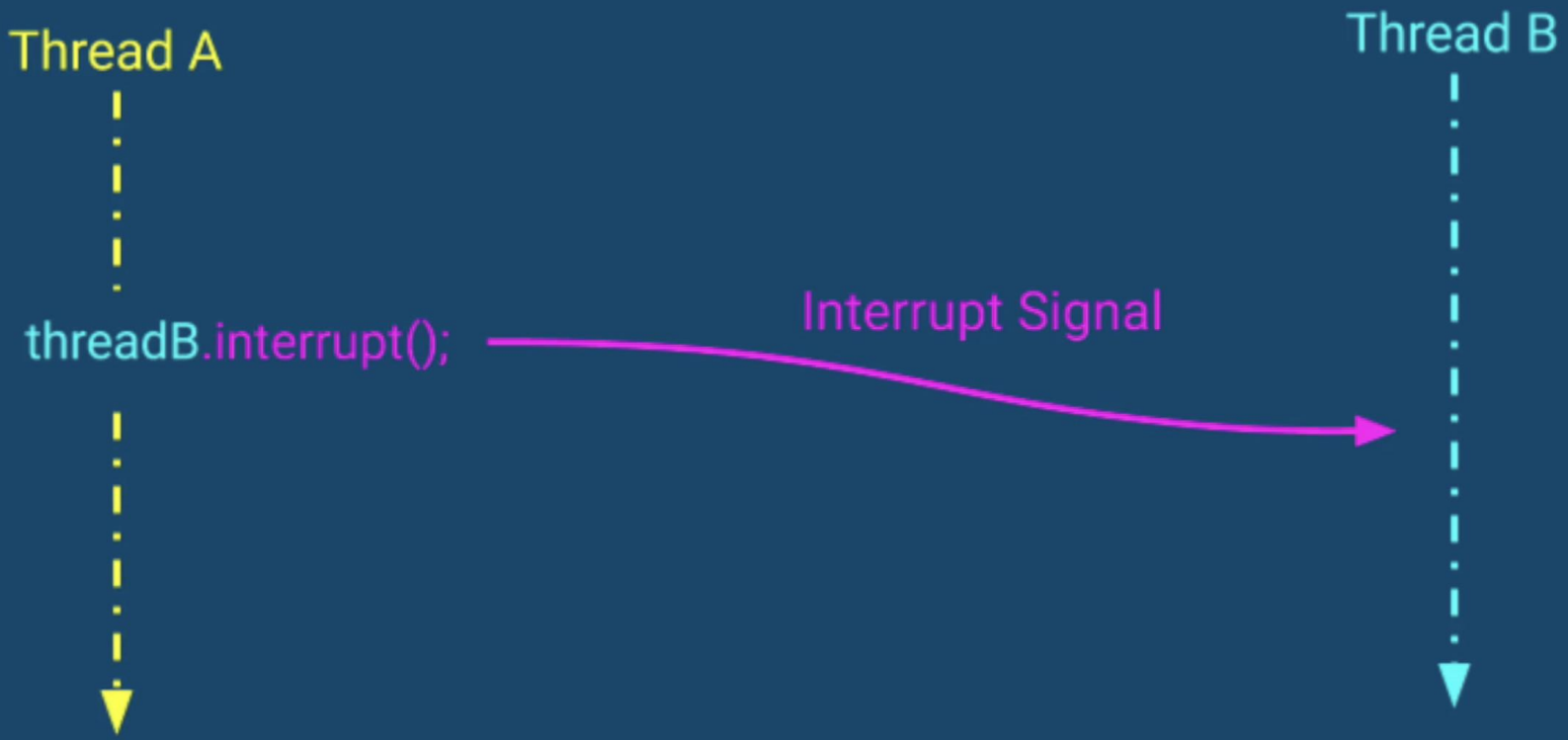
Thread Termination - Why and When?

- Threads consume resources
  - Memory and kernel resources
  - CPU cycles and cache memory
- If a thread finished its work, but the application is still running, we want to clean up the thread's resources
- If a thread is misbehaving, we want to stop it
- By default, the application will not stop as long as at least one thread is still running

What we learn in this lecture

- Thread termination
- Thread.interrupt()
- Daemon threads

Thread.interrupt()



When Can We Interrupt a Thread?

1. If the thread is executing a method that throws an *InterruptedException*
2. If the thread's code is handling the interrupt signal explicitly

What we learn in this lecture

- Thread termination
- Thread.interrupt()
- Daemon threads

Daemon Threads

Background threads that do not prevent the application from exiting if the main thread terminates

Daemon Threads - Scenario 1

- Background tasks, that should not block our application from terminating.

  Example: File saving thread in a Text Editor

Daemon Threads - Scenario 2

- Code in a worker thread is not under our control, and we do not want it to block our application from terminating.

  Example: Worker thread that uses an external library

Summary

- Learned how to stop threads by calling the *thread.interrupt()*
- If the method does not respond to the interrupt signal by throwing the *InterruptedException*, we need to check for that signal and handle it ourselves
- To prevent a thread from blocking our app from exiting, we set the thread to be a *Daemon* thread

What we learn in this lecture

- Threads coordination with *Thread.join()*
- Case study

Thread Coordination - Why do we need it?

- Different threads run independently
- Order of execution is out of our control

Thread Coordination - Scenario 1

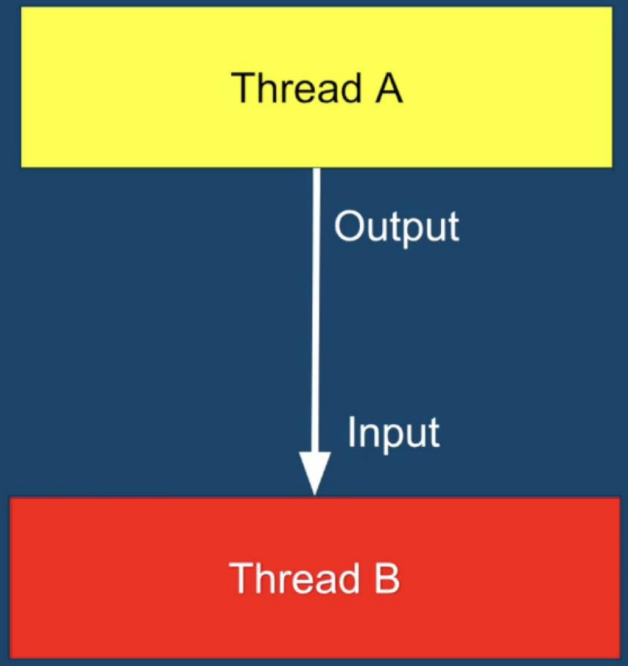
Thread Coordination - Scenario 2

Thread Coordination - Scenario 3

Thread Coordination - Scenario 4

Thread Coordination - Dependency

What if one thread depends on another thread?



Thread Coordination - Naive Solution

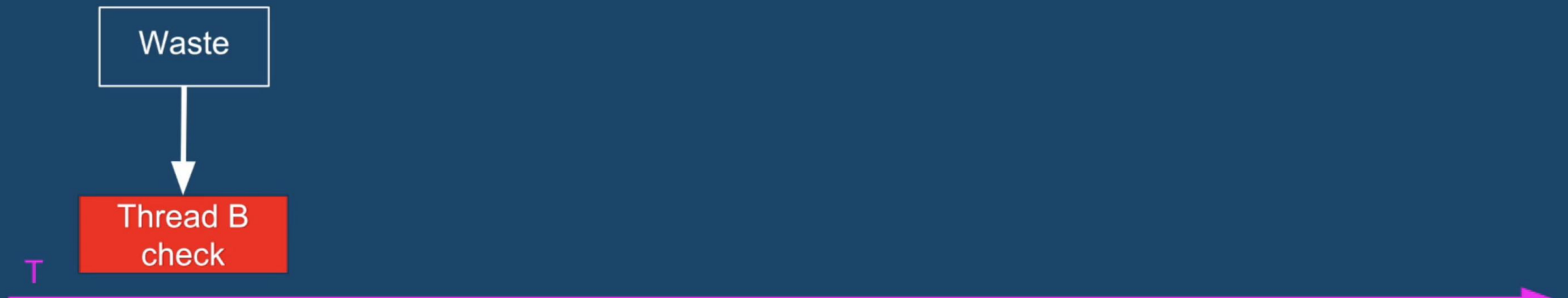
Thread B runs in a loop and keeps checking if Thread A's result is ready.

```
void waitForThreadA() {

        while( !threadA.isFinished()) {
                //burn CPU cycles
        }
}
```

Thread Coordination - Naive Solution

Thread B runs in a loop and keeps checking if Thread A's result is ready.



Thread Coordination - Naive Solution

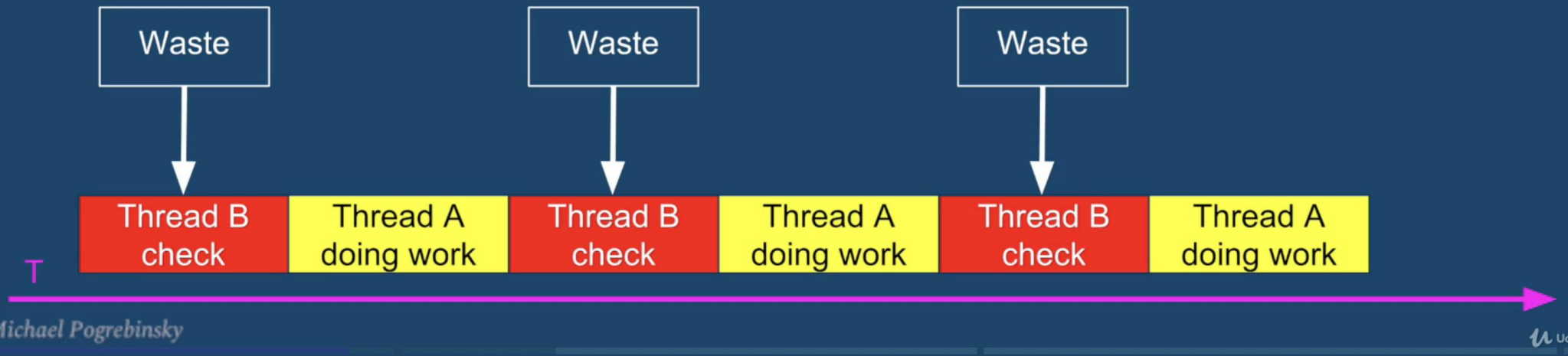
Thread B runs in a loop and keeps checking if Thread A's result is ready.

Thread Coordination - Naive Solution

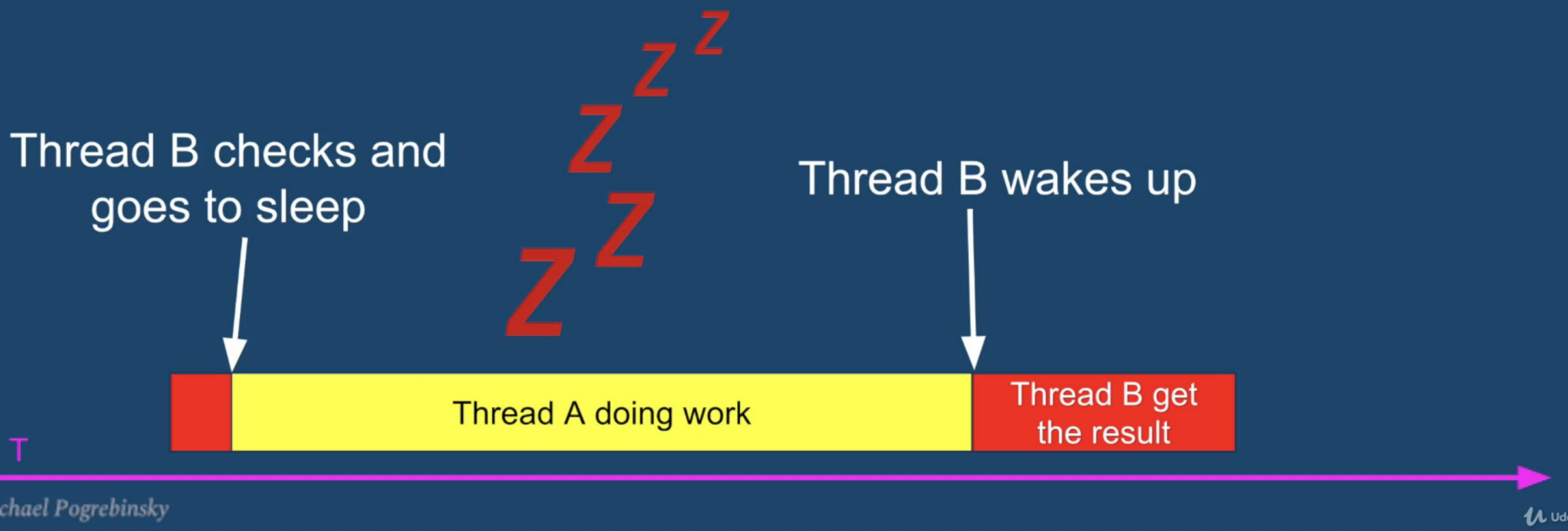
Thread B runs in a loop and keeps checking if Thread A's result is ready.

Thread Coordination - Naive Solution

Thread B runs in a loop and keeps checking if Thread A's result is ready.



Thread Coordination - Desired Solution



Thread Coordination - Thread.join()

- public final void join()
- public final void join(long millis, int nanos)
- public final void join(long millis)

Thread Coordination - Thread.join(..)

- More control over independent threads
- Safely collect and aggregate results
- Gracefully handle runaway threads using Thread.join(*timeout*)

Summary

- Do not rely on the order of execution
- Always use thread coordination
- Design code for worst case scenario
- Threads may take unreasonably long time
- Always use the Thread.join(..) with a time limit
- Stop the thread if it's not done in time