UNIVERSITÀ
DEGLI STUDI
DI PADOVA

J. Pazzini
PADOVA UNIVERSITY

8 - INTRO TO CONCURRENCY

Management and Analysis of Physics Datasets - Module B

Physics of Data

A.A. 2023/2024

PROCESSES AND THREADS

PROCESS → **An instance of a program in execution.**
Every process is a independent entity with executable code and memory.

THREAD → **A basic unit of CPU utilization.**
Every thread consists of a program counter, memory, and set of registers.
A thread is part of a process, and each process can be comprised of multiple threads.

PROCESSES AND THREADS

PROCESS → **An instance of a program in execution.**
Every process is a independent entity with executable code and memory.

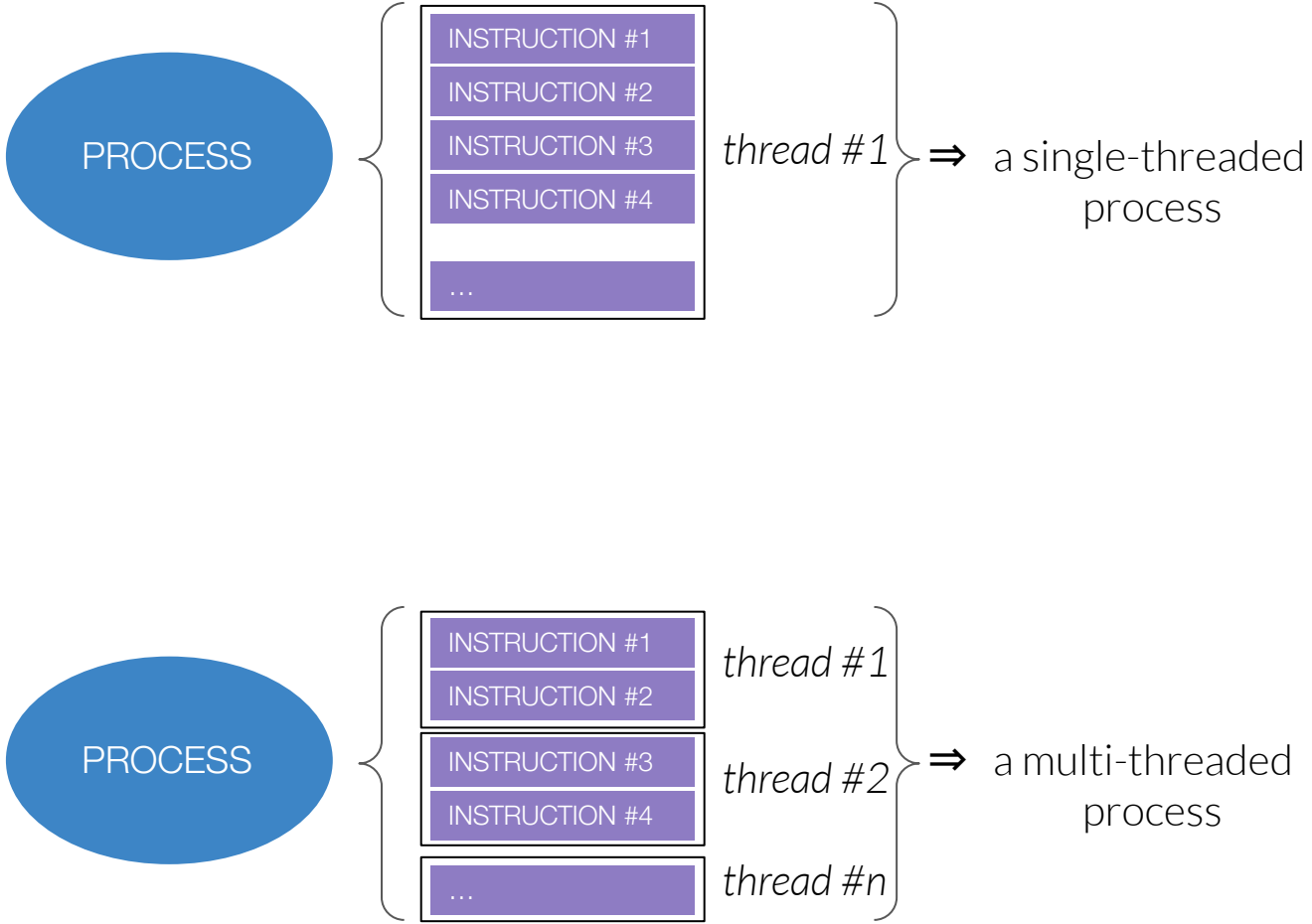
A word processor (e.g. MS Word) is a *program* → executable file containing the list of instructions
A word processor actively being executed is a process.

Multiple instances of the same program can be executed ⇒ same *program*, but multiple *processes*

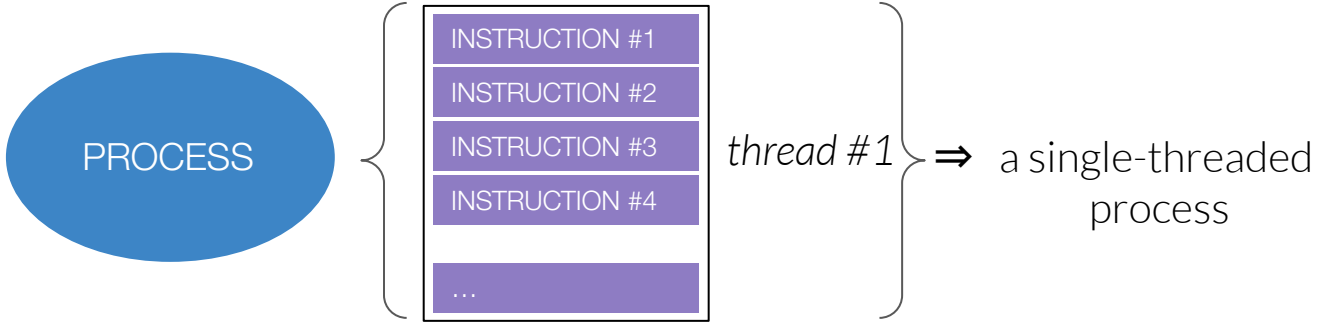
THREAD → **A basic unit of CPU utilization.**
Every thread consists of a program counter, memory, and set of registers.
A thread is part of a process, and each process can be comprised of multiple threads.

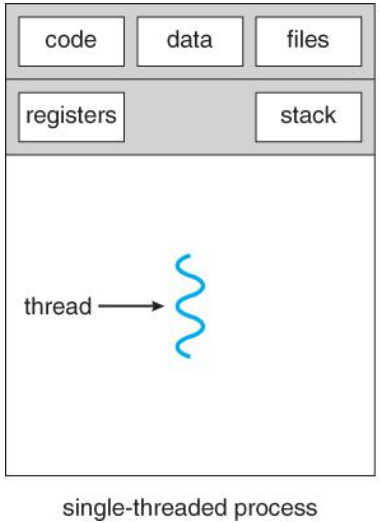
In a word processor, a thread may check spelling and grammar while another thread processes user input (keystrokes), while yet a third thread loads an image from the hard drive, and a fourth does periodic automatic backups of the file being edited.

PROCESSES AND THREADS

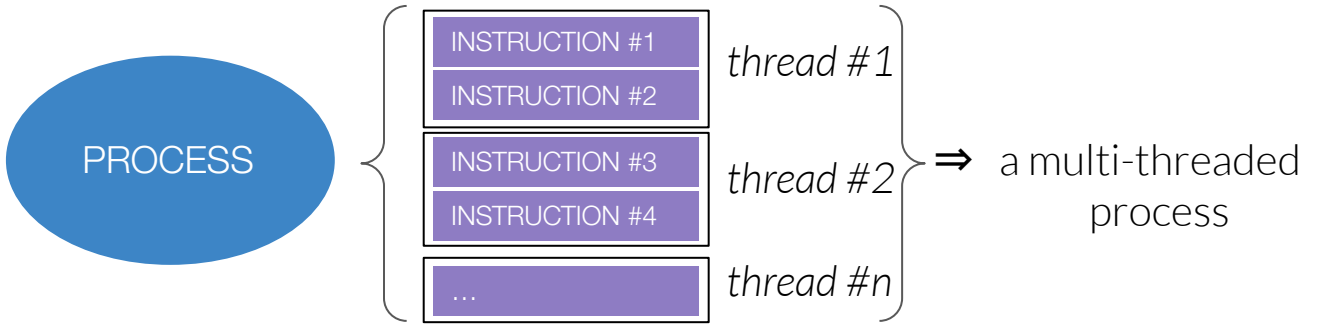


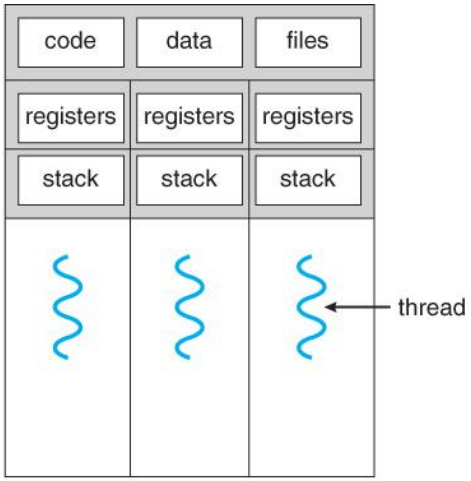
PROCESSES AND THREADS





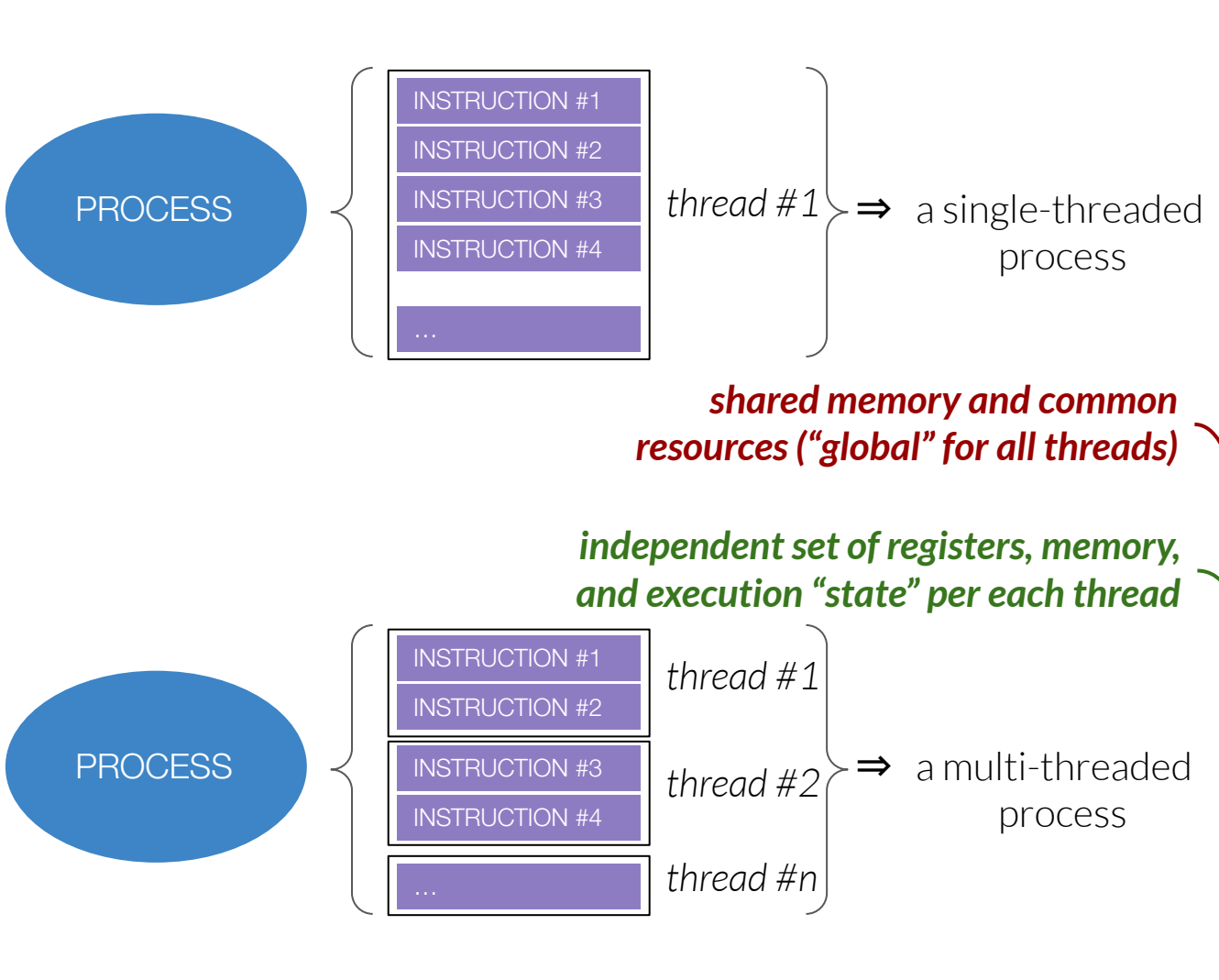
single-threaded process





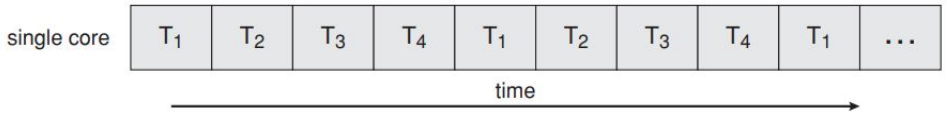
multithreaded process

PROCESSES AND THREADS

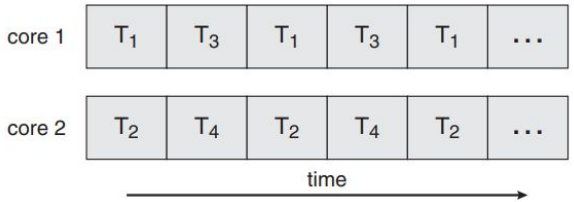


CONCURRENCY AND PARALLELISM

CONCURRENCY → **Multiple tasks being *in progress* at the same time.**
Tasks can start/run/stop independently and can be executed out of order.



PARALLELISM → **Multiple tasks being *in execution* at the same time.**
True capability to execute code simultaneously.



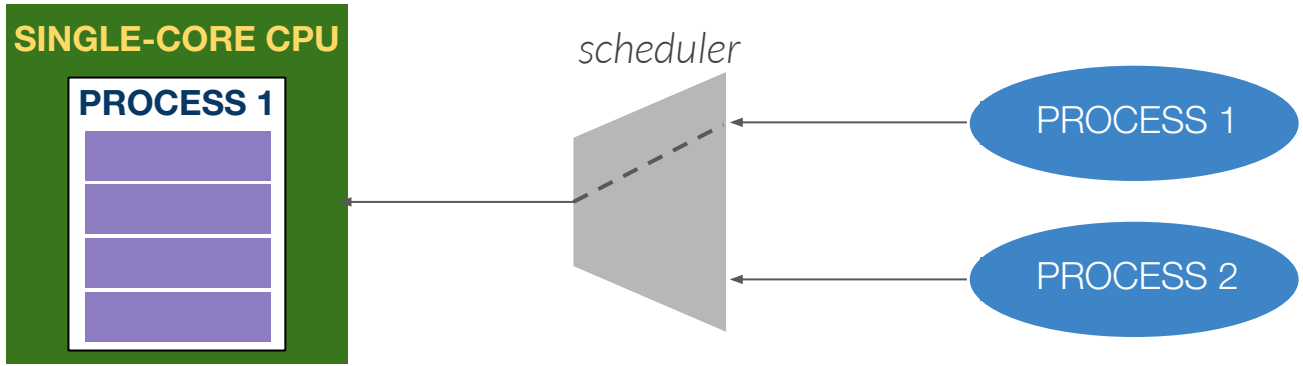
MULTI-TASKING

Execution of multiple *concurrent* tasks enabled by the Operating System

The OS instructs the CPU to share its time between the various tasks

- **Pre-emptive →** a scheduler alternates between the N tasks sharing ~equally (+ priority) the CPU time
- **Collaborative →** the system switch processes only upon the active one release control to the scheduler
- …

Each task proceed to run its instructions sequentially when active



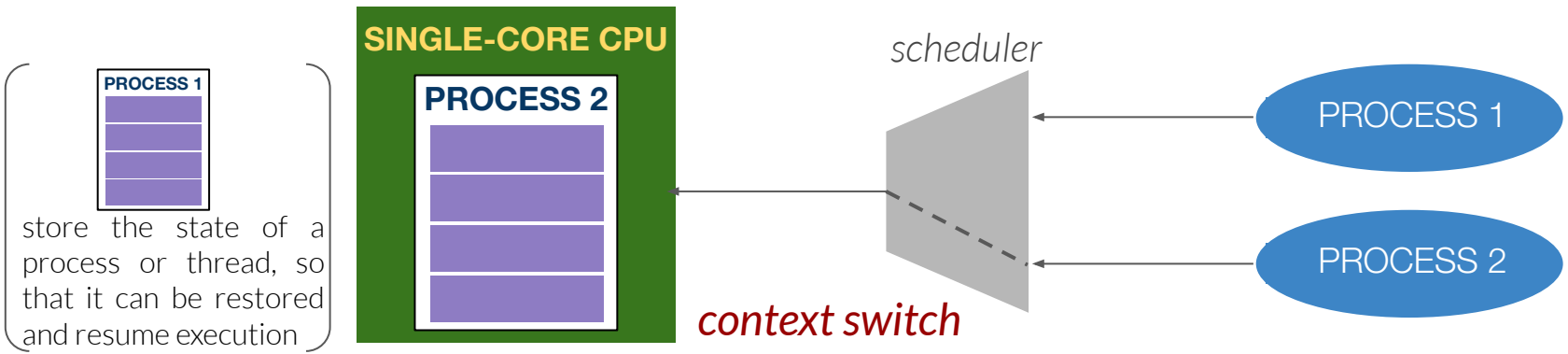
MULTI-TASKING

Execution of multiple *concurrent* tasks enabled by the Operating System

The OS instructs the CPU to share its time between the various tasks

- **Pre-emptive →** a scheduler alternates between the N tasks sharing ~equally (+ priority) the CPU time
- **Collaborative →** the system switch processes only upon the active one release control to the scheduler
- …

Each task proceed to run its instructions sequentially when active



MULTI-THREADING

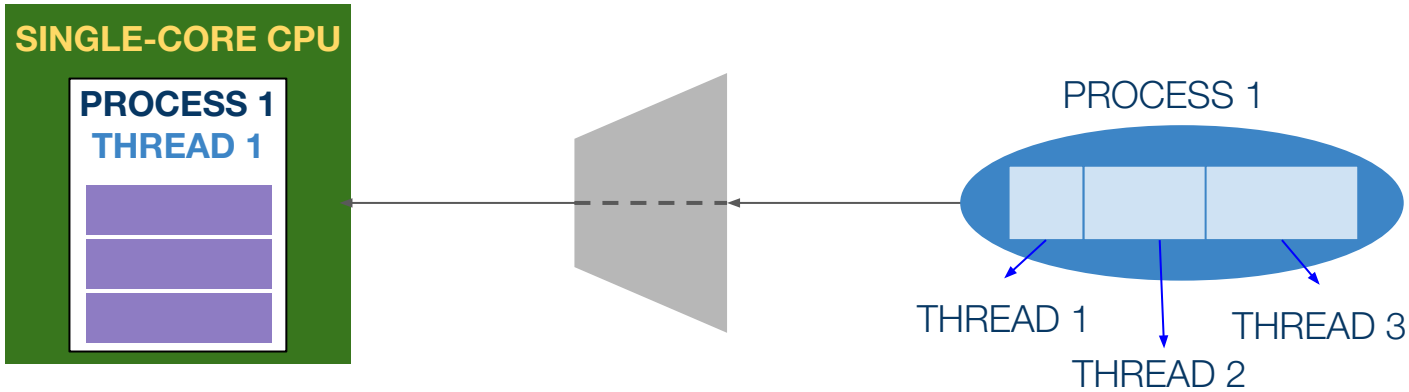
MultiThreading → execution of multiple threads

Depending on the architecture and OS, the execution is either:

- Concurrent, on single-core CPUs with multi-tasking OS
- Or parallel, on 2+ cores CPUs

Context switching of small threads is faster than entire processes

Multiple sub-components of a single process can proceed independently while sharing some common (global) resources



MULTI-THREADING

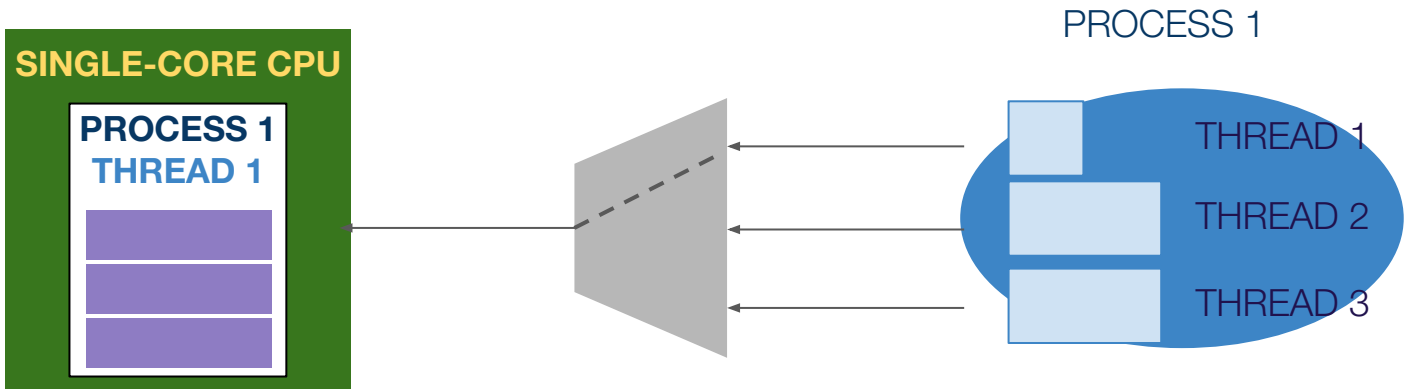
MultiThreading → execution of multiple threads

Depending on the architecture and OS, the execution is either:

- Concurrent, on single-core CPUs with multi-tasking OS
- Or parallel, on 2+ cores CPUs

Context switching of small threads is faster than entire processes

Multiple sub-components of a single process can proceed independently while sharing some common (global) resources



MULTI-THREADING

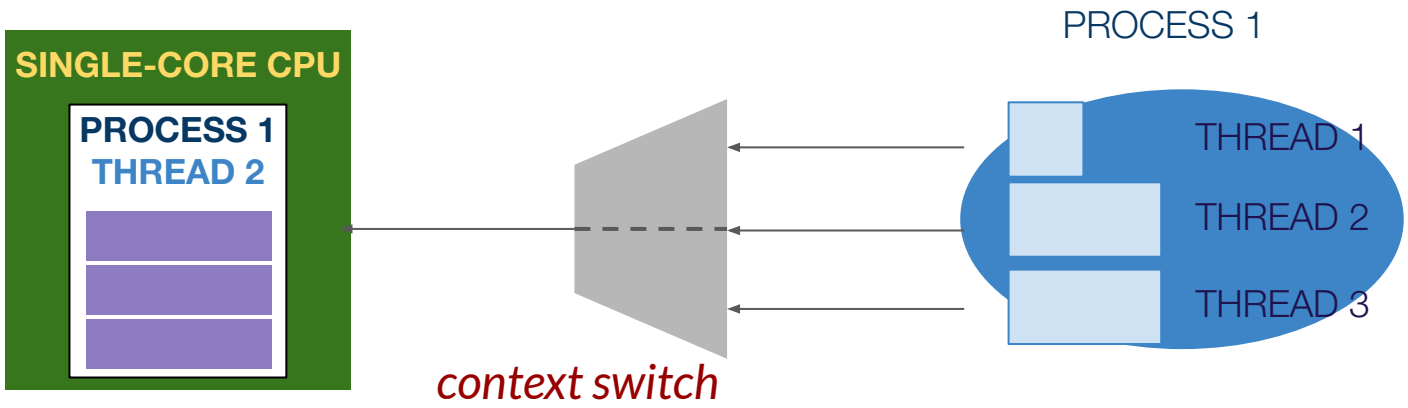
MultiThreading → execution of multiple threads

Depending on the architecture and OS, the execution is either:

- Concurrent, on single-core CPUs with multi-tasking OS
- Or parallel, on 2+ cores CPUs

Context switching of small threads is faster than entire processes

Multiple sub-components of a single process can proceed independently while sharing some common (global) resources



MULTI-THREADING

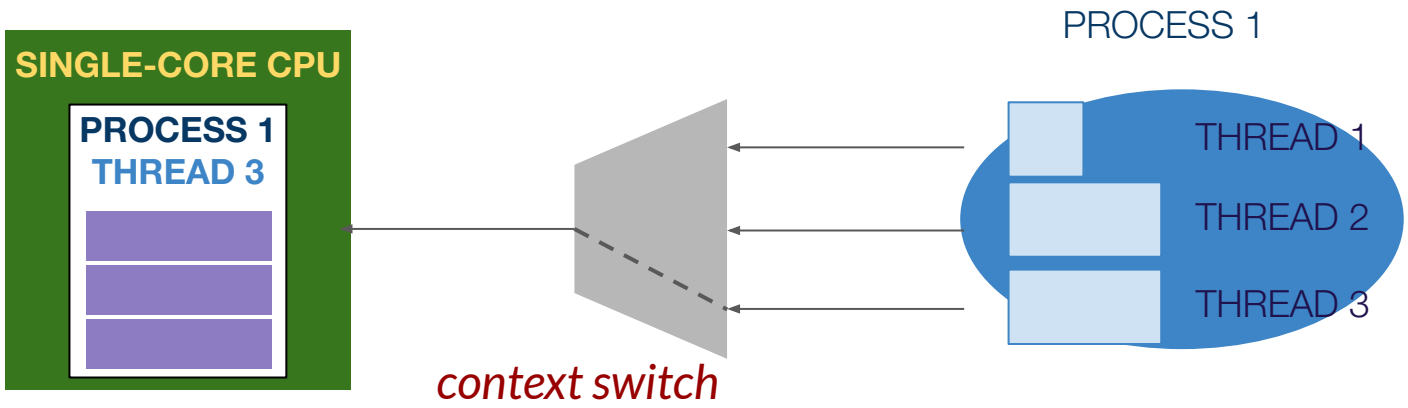
MultiThreading → execution of multiple threads

Depending on the architecture and OS, the execution is either:

- Concurrent, on single-core CPUs with multi-tasking OS
- Or parallel, on 2+ cores CPUs

Context switching of small threads is faster than entire processes

Multiple sub-components of a single process can proceed independently while sharing some common (global) resources



MULTI-THREADING

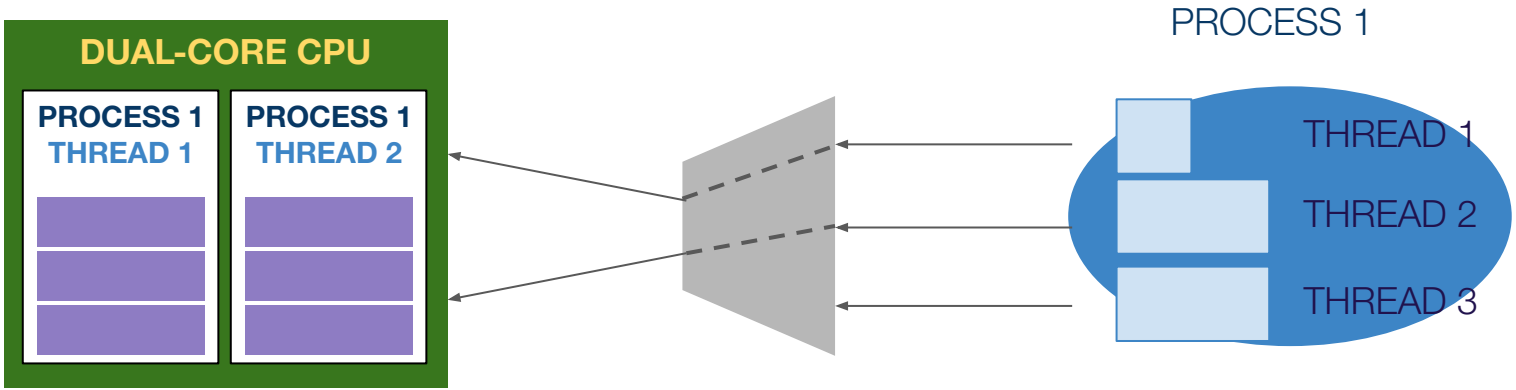
MultiThreading → execution of multiple threads

Depending on the architecture and OS, the execution is either:

- Concurrent, on single-core CPUs with multi-tasking OS
- Or parallel, on 2+ cores CPUs

Context switching of small threads is faster than entire processes

Multiple sub-components of a single process can proceed independently while sharing some common (global) resources



MULTI-THREADING

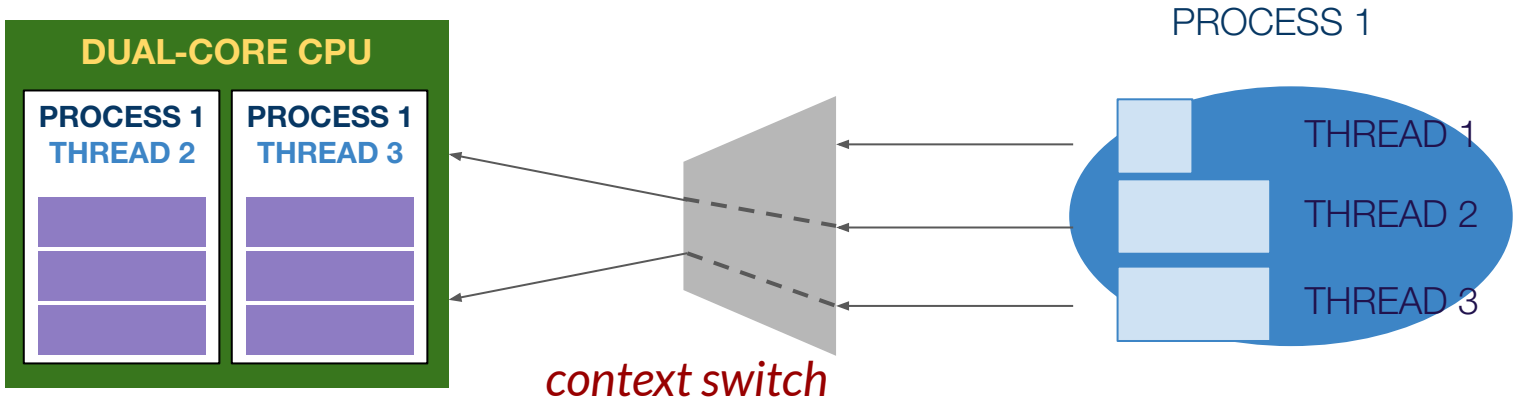
MultiThreading → execution of multiple threads

Depending on the architecture and OS, the execution is either:

- Concurrent, on single-core CPUs with multi-tasking OS
- Or parallel, on 2+ cores CPUs

Context switching of small threads is faster than entire processes

Multiple sub-components of a single process can proceed independently while sharing some common (global) resources



RACE CONDITIONS

However, multi-threaded execution can lead to issues if not handled properly…

For instance:

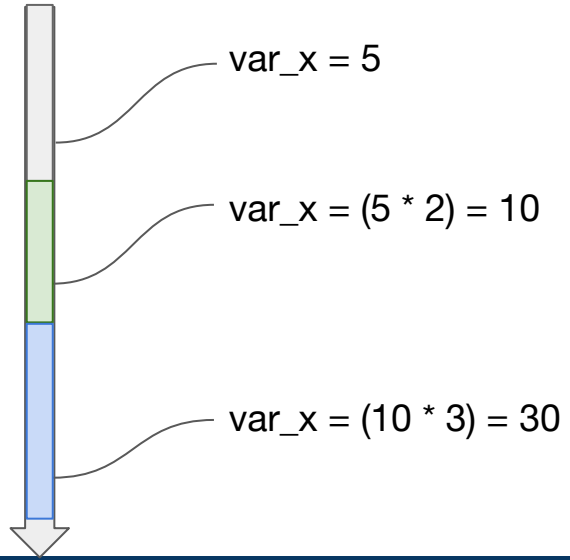
- Multiple threads can access the same memory address (e.g. the memory location of a global variable) → memory is *shared* across threads
- The concurrent/simultaneous execution of 2 instructions on shared data may create inconsistencies across the threads

```
# some code
var_x = 5


...
var_x *= 2



...
var_x *= 3
```



RACE CONDITIONS

However, multi-threaded execution can lead to issues if not handled properly…

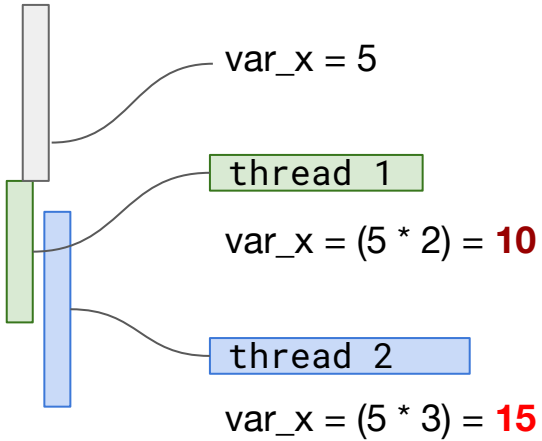
For instance:

- Multiple threads can access the same memory address (e.g. the memory location of a global variable) → memory is *shared* across threads
- The concurrent/simultaneous execution of 2 instructions on shared data may create inconsistencies across the threads

```
# some code
var_x = 5

# thread 1
...
var_x *= 2
# thread 1 - end

# thread 2
...
var_x *= 3
# thread 2 - end
```



RACE CONDITIONS

However, multi-threaded execution can lead to issues if not handled properly…

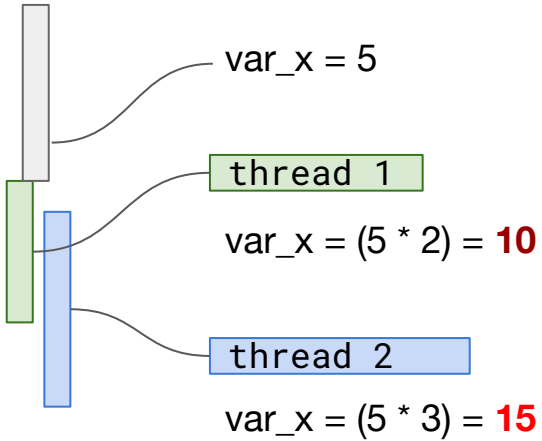
For instance:

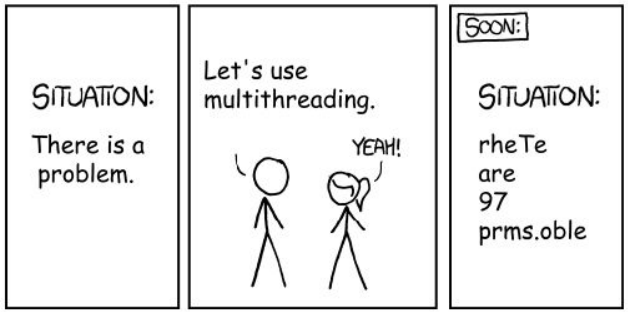
- Multiple threads can access the same memory address (e.g. the memory location of a global variable) → memory is *shared* across threads
- The concurrent/simultaneous execution of 2 instructions on shared data may create inconsistencies across the threads

```
# some code
var_x = 5

# thread 1
...
var_x *= 2
# thread 1 - end

# thread 2
...
var_x *= 3
# thread 2 - end
```





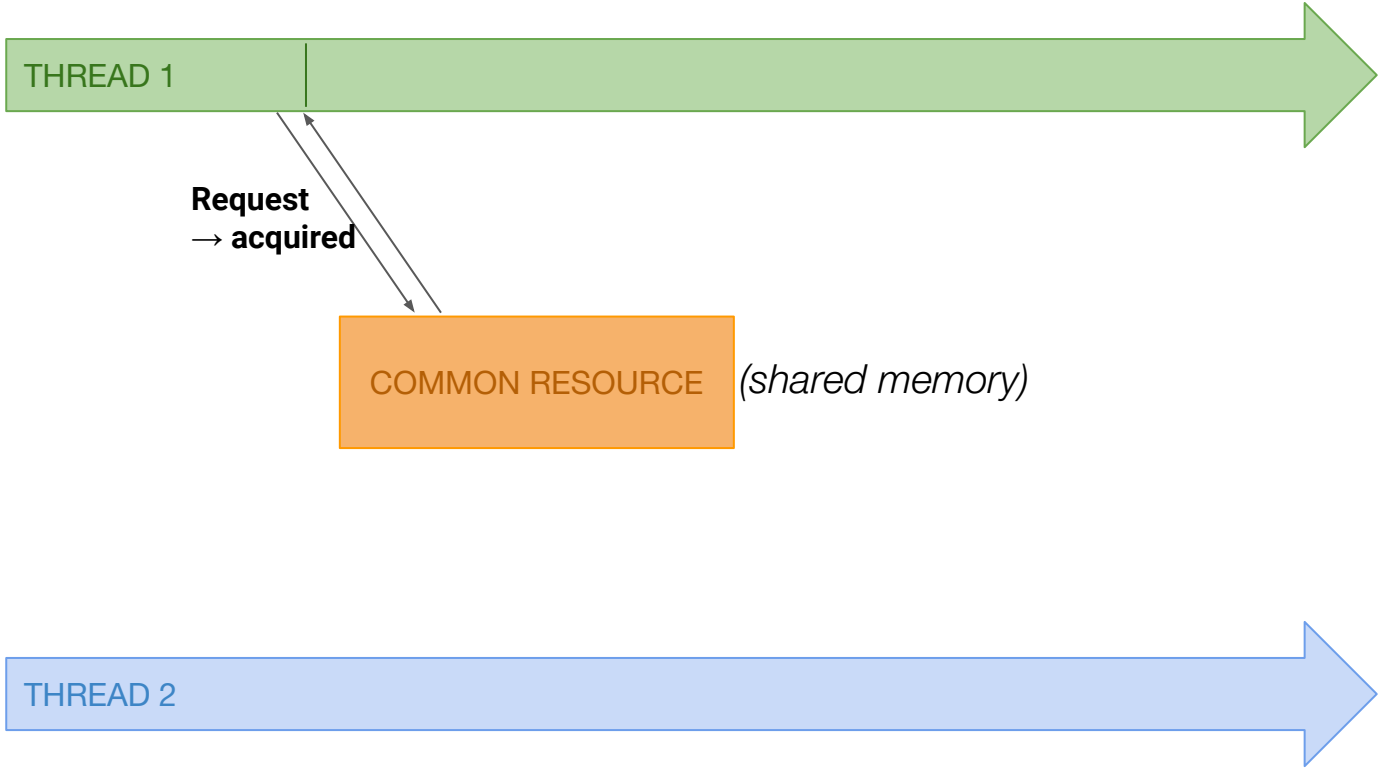
RACE CONDITION

When a given resource (e.g. the same data in memory) is accessed concurrently by multiple threads AND then outcome depends on the particular order in which the access takes place

THREAD SYNCHRONIZATION

In multi-threaded programming the resources are secured by means of:

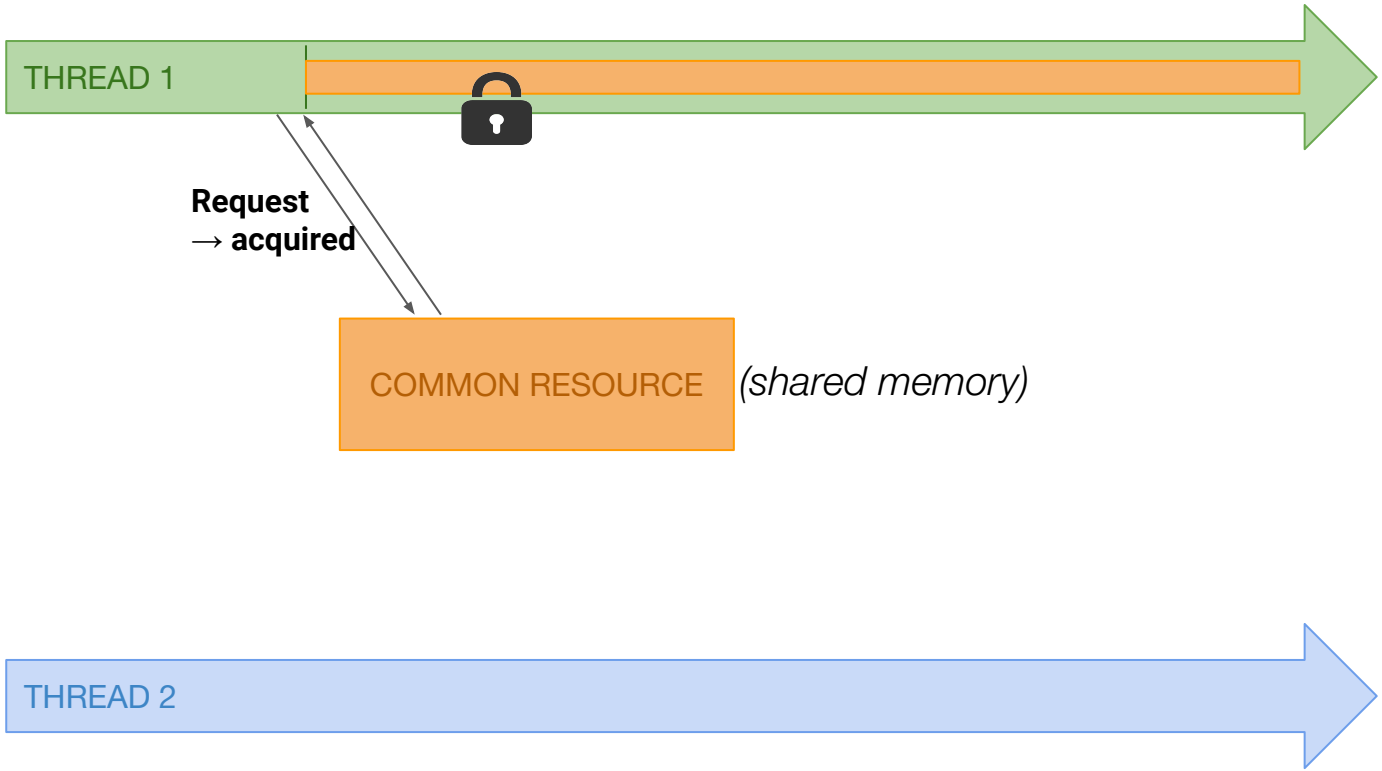
- **Lock/Mutex (mutual exclusion)**
- Binary semaphores
- Counting semaphores
- …



THREAD SYNCHRONIZATION

In multi-threaded programming the resources are secured by means of:

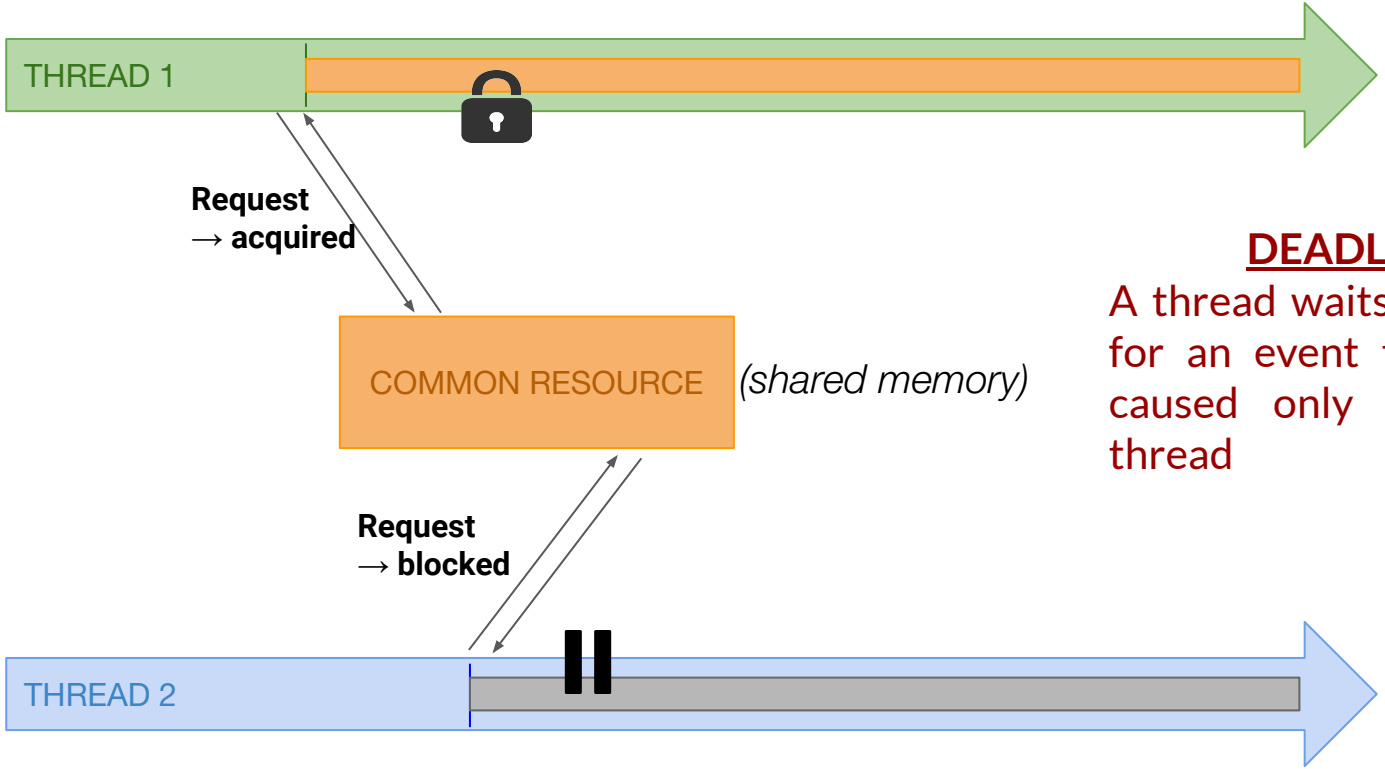
- **Lock/Mutex**
- Binary semaphores
- Counting semaphores
- …



THREAD SYNCHRONIZATION

In multi-threaded programming the resources are secured by means of:

- **Lock/Mutex**
- Binary semaphores
- Counting semaphores
- …



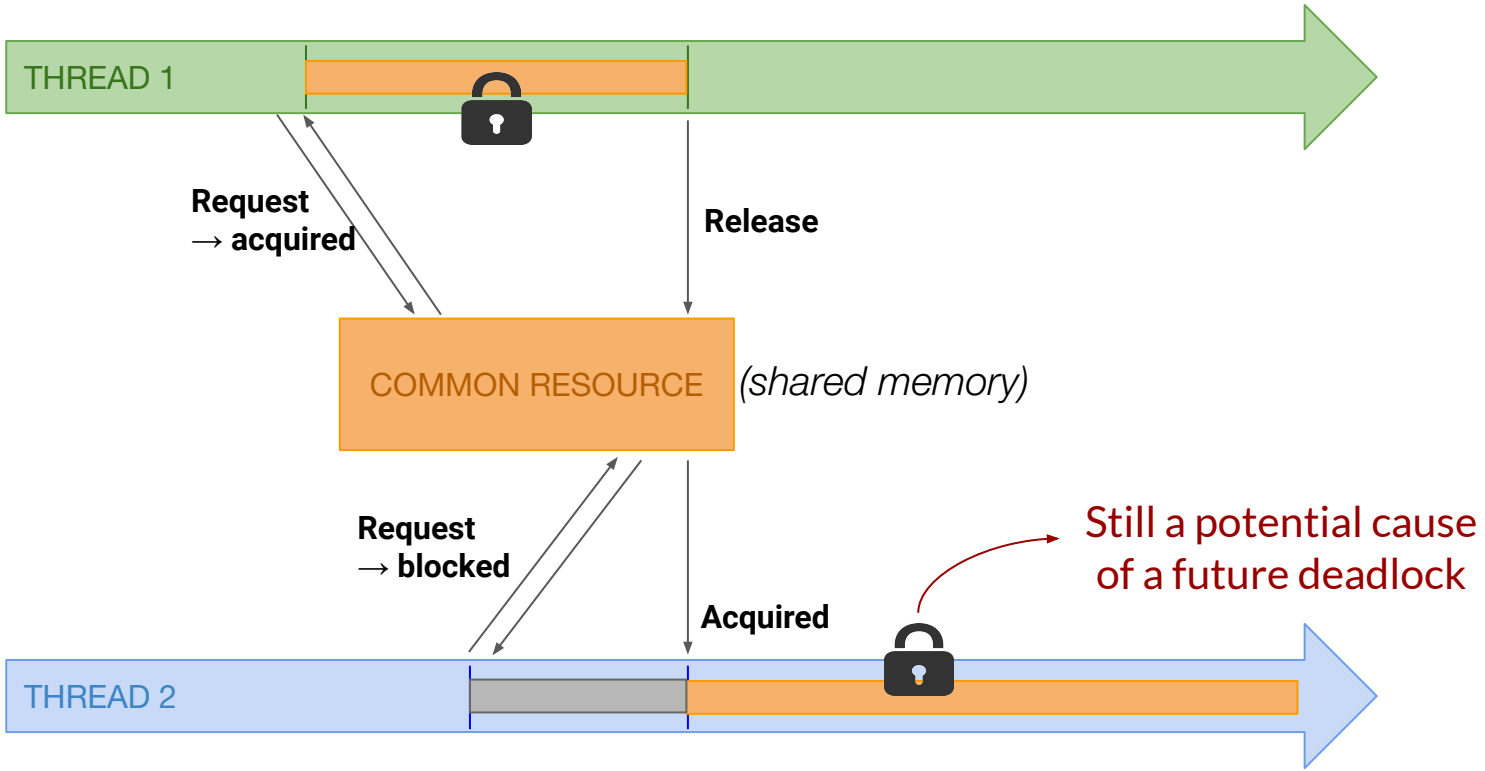
DEADLOCK

A thread waits indefinitely for an event that can be caused only by another thread

THREAD SYNCHRONIZATION

In multi-threaded programming the resources are secured by means of:

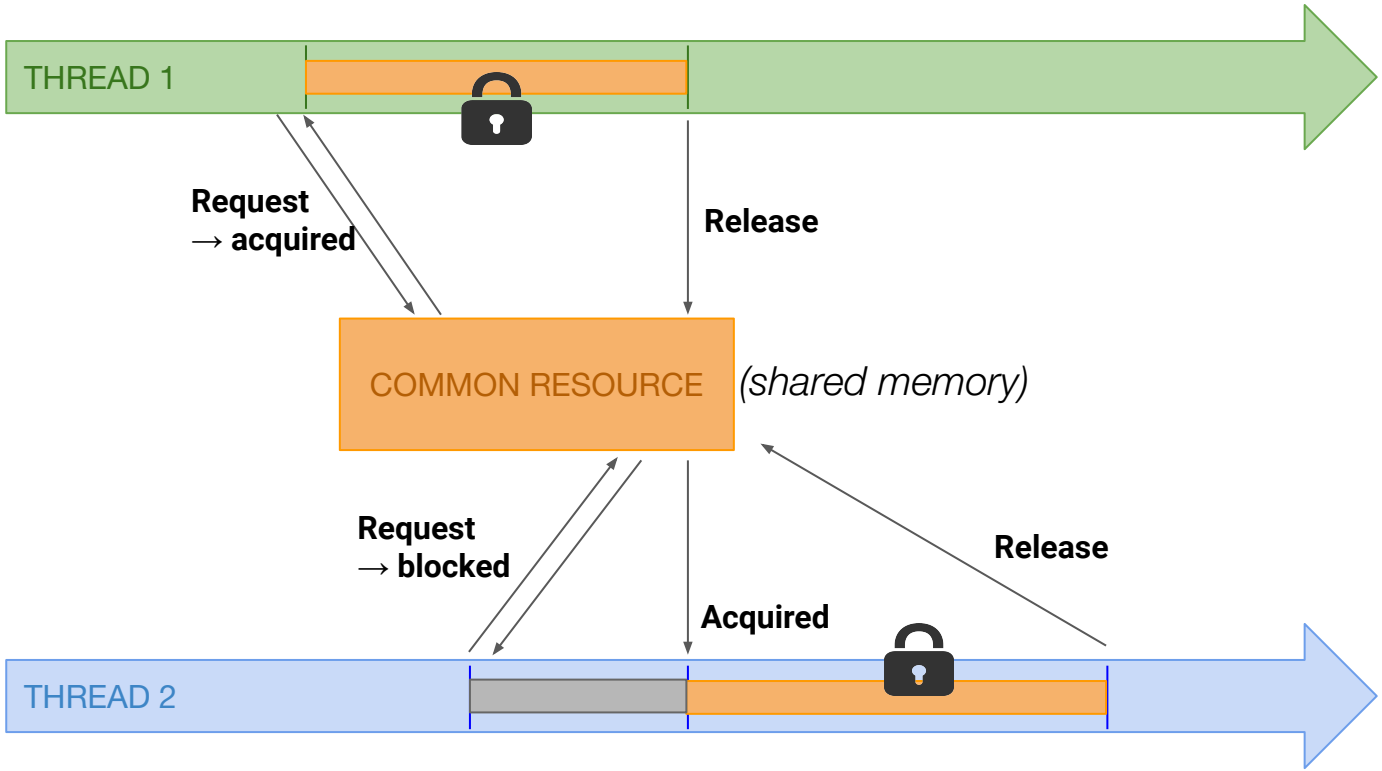
- **Lock/Mutex**
- Binary semaphores
- Counting semaphores
- …



THREAD SYNCHRONIZATION

In multi-threaded programming the resources are secured by means of:

- **Lock/Mutex**
- Binary semaphores
- Counting semaphores
- …



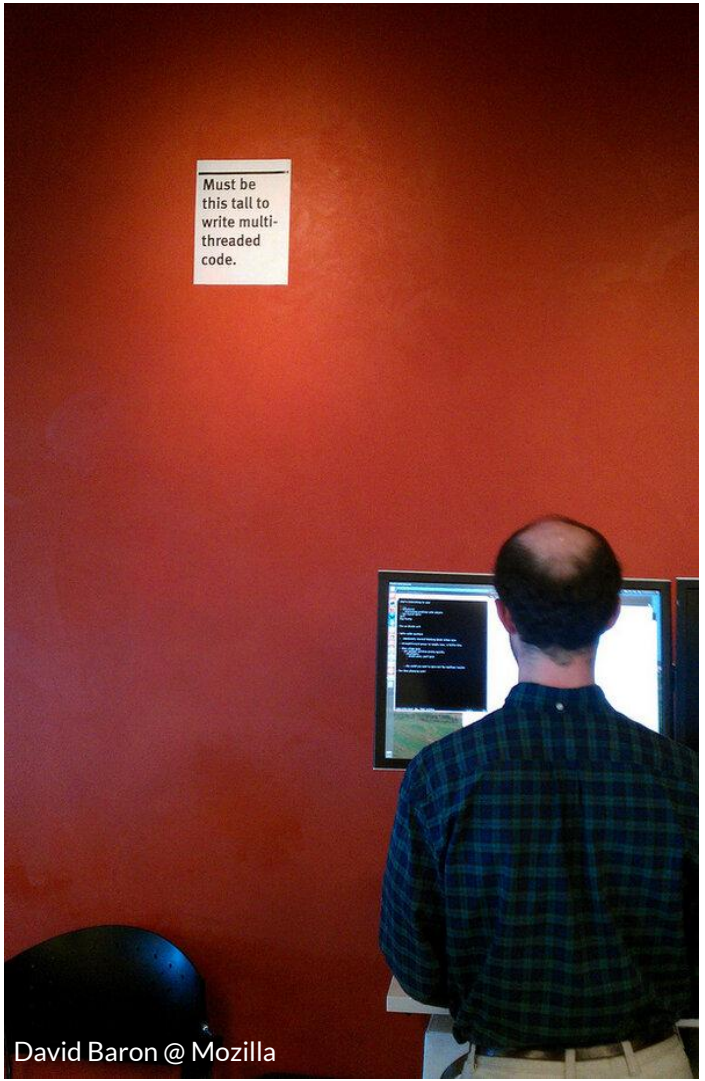
MULTI-THREADED PROGRAMMING IN PYTHON

We will run a simple set of examples of multi-threaded programming using the `threading` Python module:

- Single thread
- Two concurrent threads*
- Threads exchanging/sharing data
- A simple example of thread-safe objects (`queues`, …)

We'll manipulate a set of (still very simple) threads using the `concurrent.futures` Python module:

- Pools of threads working together



David Baron @ Mozilla

*THE GIL

The **GIL** (***Global Interpreter Lock***) is a **mutex on the** very **Python interpreter** itself

⇒ Only 1 thread in execution can acquire the GIL, thus no more than 1 thread of a process can effectively run in parallel (hence disabling thread-based parallelization)

This is a “nasty trick” implemented by **CPython** (the most common python interpreter) to prevent deadlocks and race conditions, as there is only one lock for the whole interpreter

The GIL finally ensures thread safety for the Python interpreter, but it does it by not making it possible to have parallel execution in different threads

*THE GIL

The **GIL** (***Global Interpreter Lock***) is a **mutex on the** very **Python interpreter** itself

⇒ Only 1 thread in execution can acquire the GIL, thus no more than 1 thread of a process can effectively run in parallel (hence disabling thread-based parallelization)

This is a "nasty trick" implemented by **CPython** (the most common python interpreter) to prevent deadlocks and race conditions, as there is only one lock for the whole interpreter

The GIL finally ensures thread safety for the Python interpreter, but it does it by not making it possible to have parallel execution in different threads

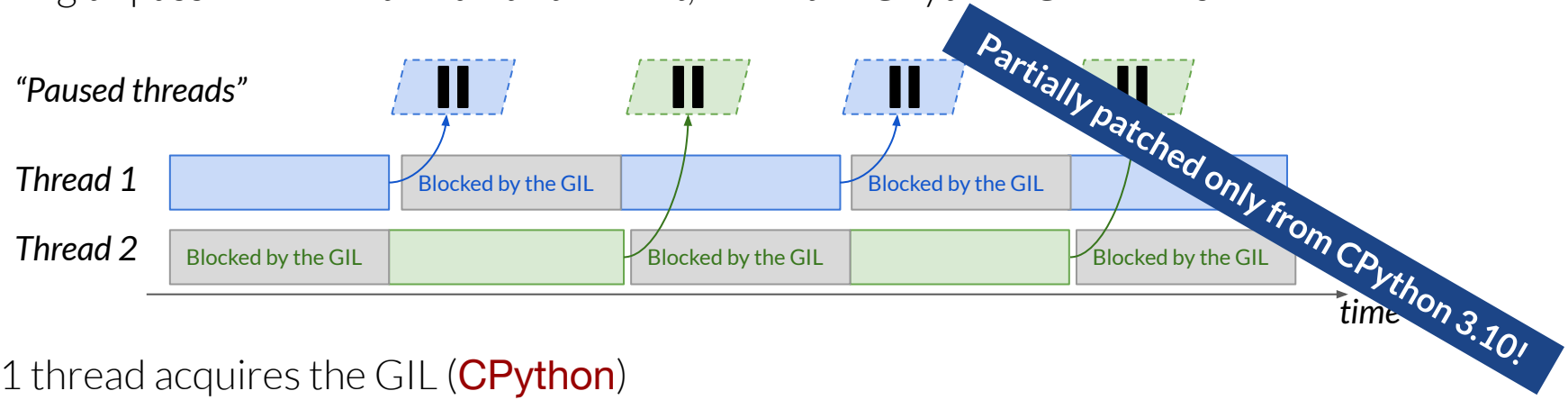
It should be noted that the existence of the GIL is not a requirement of the Python language per se, but rather is the specific implementation of the CPython interpreter to ensure thread-safety

Other Python interpreters do exist not relying on the GIL, e.g. Jython and IronPython

HOWEVER…

The GIL does not prevent all together race conditions to happen in the case of multiple threads using the same resources

The OS scheduler sees **n** CPUs and tries to schedule threads to all of them, operating context switching to pass from one thread to the next, while the CPython GIL blocks.



- 1 thread acquires the GIL (CPython)
- All others threads are scheduled (OS) and make requests, denied by the GIL (CPython)
- The GIL (CPython) is released and context switch happens (OS)

Every so often, thread execution do overlap during the context switch operated by the OS still causing race conditions… even despite the GIL!

It's a very complex matter, for the audacious: https://www.dabeaz.com/python/GIL.pdf

MULTIPROCESSING IN PYTHON

Due to the GIL limitation, multi-threaded programming is not extremely convenient.

We will instead use the `multiprocessing` module to run simple examples of parallel processes in Python.

- Two parallel processes

- Submit a pool of processes using maps, and pools of workers
  - `starmap` (/`map`/`apply`/...) processes
  - Synchronous vs Asynchronous execution

IN THE END… THREADS OR PROCESSES IN PYTHON?

Threading

- Threads share the same memory and can write to and read from shared variables
- Due to CPython GIL, two threads won't be executed at the same time, but only concurrently
- Effective for I/O-bound tasks (e.g. data access from/to disk or remote locations)
- Can be easily implemented in Python with the `threading` module

Multiprocessing

- Every process has is own memory space
- Every process can contain one or more subprocesses
- Can be used to achieve parallelism by taking advantage of multi-core machines
- Effective for CPU-bound tasks (e.g. data "crunching")
- Can be easily implemented in Python with the `multiprocessing` module

ANYTHING ELSE…?

We focused on a few basic examples to provide a simple overview of concurrency and parallel processing.

Many more libraries do exist to enable multi-threading and parallel processing in Python, including `joblib` and `mpi4py`

One notable mention goes to `joblib`: a Python library that implements a variety of functionalities including parallel processing, allowing the user to choose which backend to use (e.g. `threading` or `multiprocessing`).

```
import joblib

def f(x):
   return x**2

joblib.Parallel(n_jobs=4, backend='multiprocessing')\
   (joblib.delayed(f)(_) for _ in range(10))
```