LECTURE 39

THREADS AND CONCURRENCY

MCS 260 Fall 2020

Emily Dumas

REMINDERS

- Worksheet 14 available (it's short)
- TA office hours (open to all) replace discussions tomorrow. Zoom links in course announcement.

- **Concurrency** — The ability to have several independent operations underway at the same time
- **Parallelism** — The ability to have several independent operations running at the same time

Thus parallelism is one way to achieve concurrency. Pausing and switching is another way.

E.g. a single person has very limited parallelism, but can often handle extensive concurrency.

PROCESSES

A **process** is the notion of one instance of a running program, consisting of code and data loaded into memory, ready for the CPU to execute.

A process can't (normally) access the memory of other processes.

Process management and scheduling is an important OS function.

THREADS

A **thread** is a division within a process. Each thread executes concurrently with the others.

Each process has at least one thread.

E.g. One thread handles the GUI, another handles communication with APIs. The GUI remains responsive even if an API call is slow.

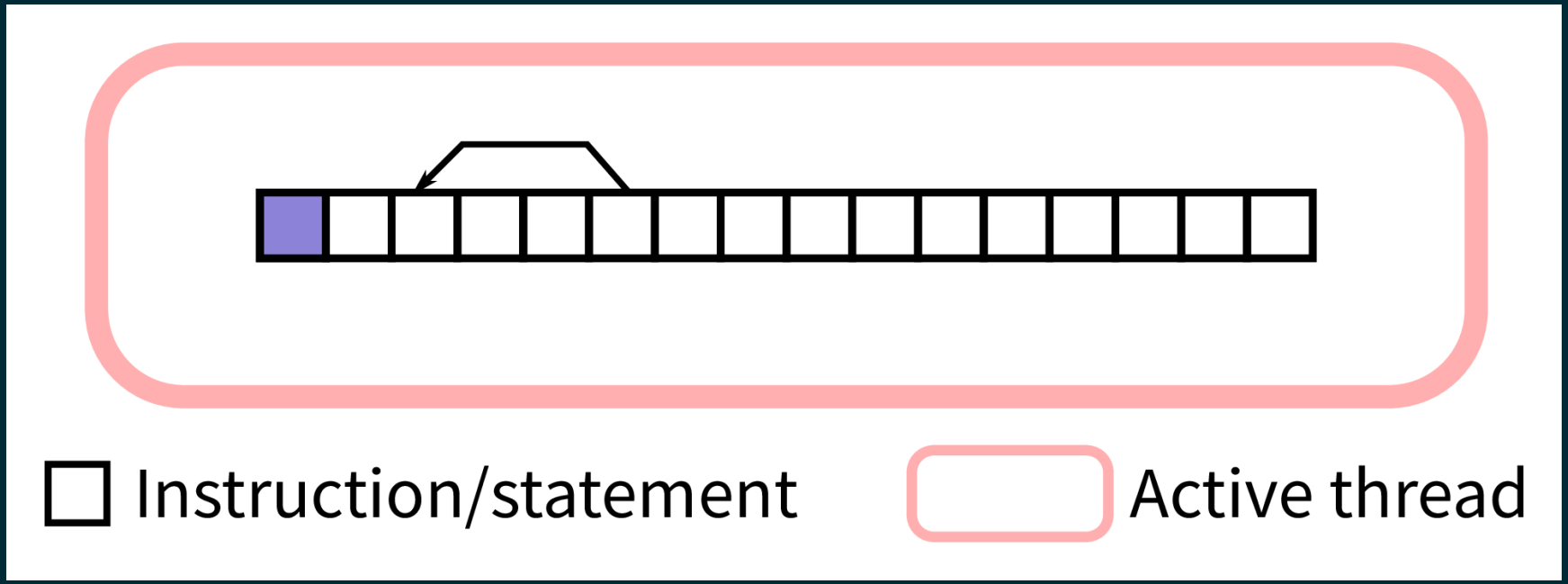
Key: *Threads of a single process share memory space, i.e. code and data.*

CPUS AND CORES

A modern CPU typically contains multiple **cores**. For most purposes these behave as separate CPUs running in parallel.

Thus most CPUs run several processes or threads in parallel.

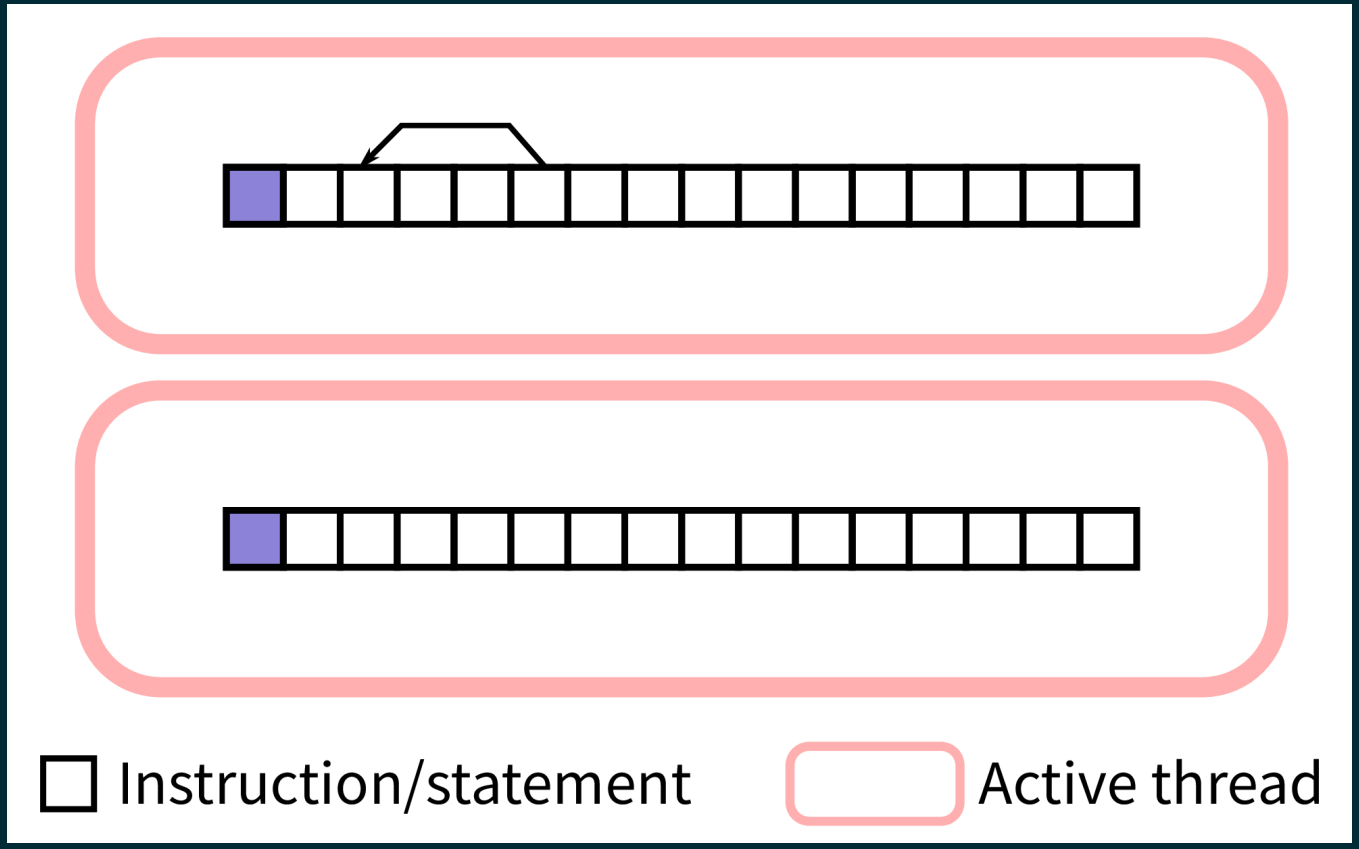
SINGLE-THREADED PROCESS



Sequential execution

Note: The animation above won't play in PDF slides.

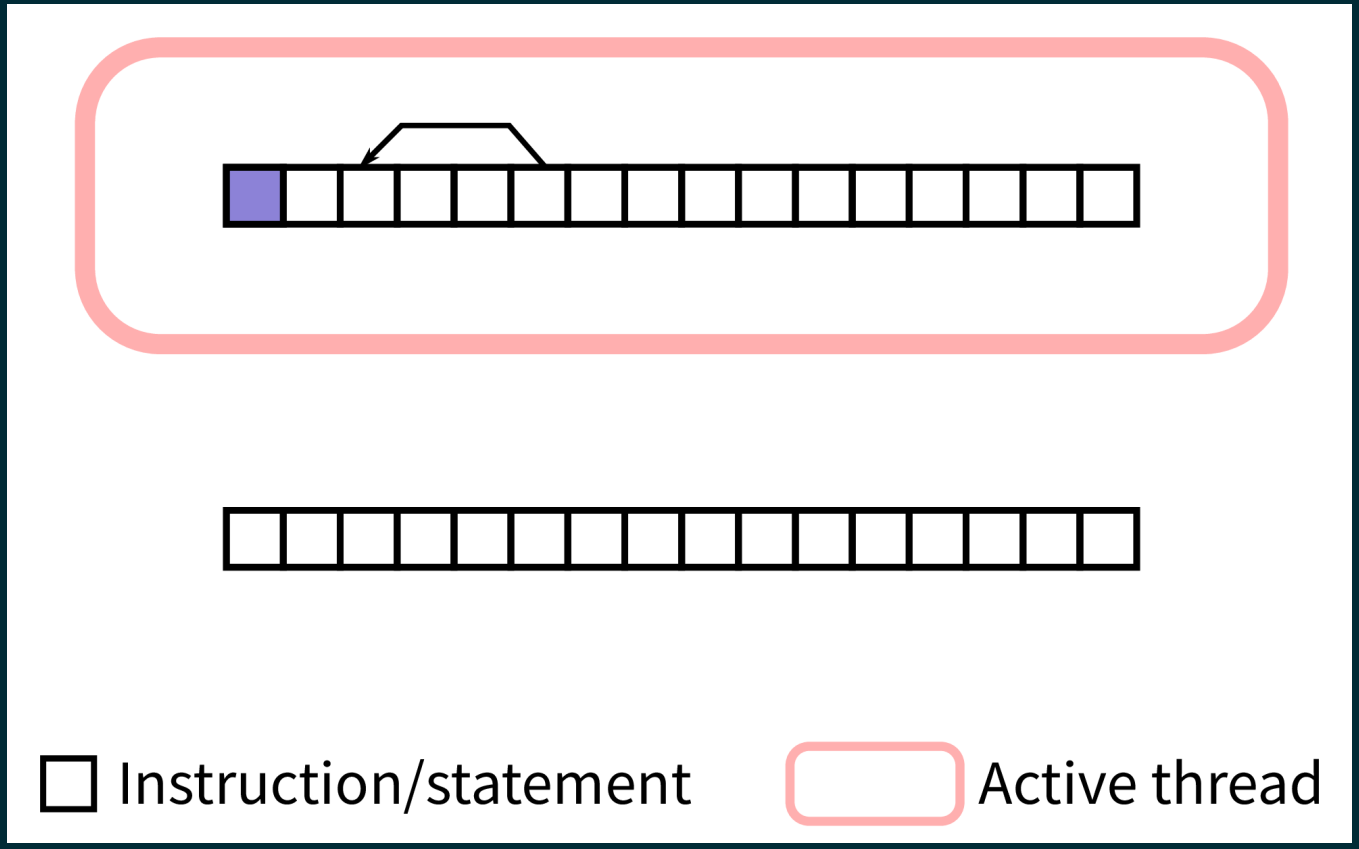
MULTI-THREADED PROCESS (THEORY)



Parallel execution

Note: The animation above won't play in PDF slides.

MULTI-THREADED PROCESS (PYTHON)



Concurrent execution

Note: The animation above won't play in PDF slides.

Big limitation: Only one thread of a Python program can be executing Python code at any given time.

Still, using threads for *concurrency* can be very useful, especially in GUI programs.

Threads of a single program often need to communicate; we'll talk about this a bit later.

BASIC THREAD EXAMPLE

The `threading` module allows a Python program to create new threads.

Let's make a program with two threads that each print some data to the terminal.

`threading.Thread(target=func,args=alist)` prepares a new thread that calls function `func` with the elements of `alist` as arguments. The `.start()` method of this object actually starts the thread.

OBSERVATIONS

The program exits when its last thread is done.

We have no direct control over order of execution in different threads.

WAITING AND DAEMONS

A `threading.Thread` object has a method `.join()` which tells the calling thread to wait until the other thread is finished.

Alternatively, a **daemon thread** is one that is automatically killed when the main thread exits. Create one by passing `daemon=True` to the `threading.Thread` constructor.

WORKER PATTERN

Often, the main thread spawns some **worker threads** to handle slow tasks in the background.

The main thread sends jobs to the workers, and may wait for results to be returned.

This is also called the **producer-consumer pattern**.

THREAD COMMUNICATION

If multiple threads access the same object, with at least one of them writing, you're asking for trouble.

Instead of directly accessing a shared object, you should use a data structure designed for communication between threads.

`queue.Queue` from the module `queue` does this. It has a `.put(val)` method to submit an object to the queue, and a `.get()` method to retrieve an object.

REFERENCES

- Python module documentation:
  - threading
  - queue
- Section 6.6 of the optional course text by Brookshear & Brylow discusses threading and concurrency
- A tutorial on concurrency by Jim Anderson (from RealPython)

REVISION HISTORY

- 2020-11-25 Corrected threading.Thread constructor arguments
- 2020-11-21 Initial publication