9 - Shared memory concurrency I

florian.felten@uni.lu

How can a chat receive your keyboard input and inputs from the network at the same time?

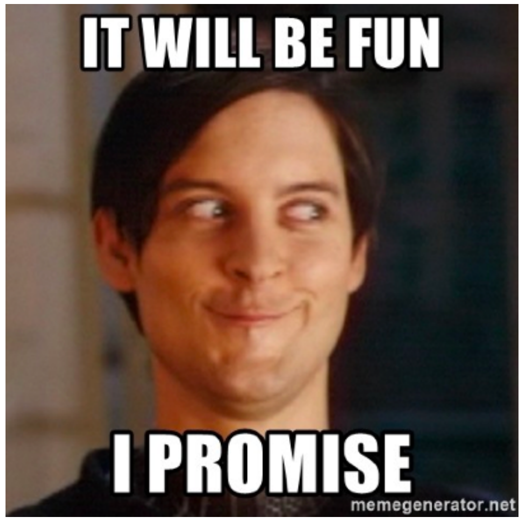
Doing multiple things at the same time

Why is there so much hype with GPUs?

How does excel not freeze when you input a number in cell and it has to compute new results in other cells?

The promise

- You will be introduced to the concept of Thread;
- You will understand some of the issues related to concurrent programming;
- We will present one of the possible solutions to solve these issues.

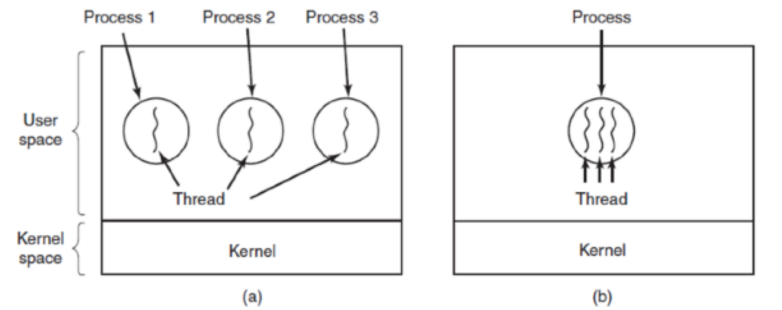


By the end of this course, my promise is that you…

Threads

What is a *thread* ?

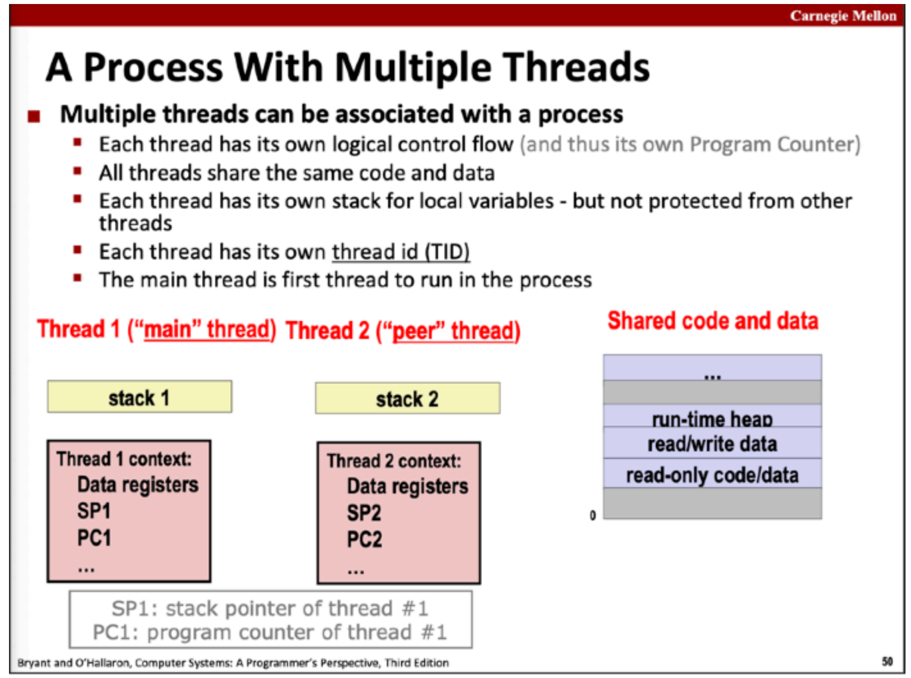
- **A "lightweight" process within a process which does not have its own address space** but serves the same purpose of running activities in parallel
- Enables to define multiple flows of control within a process



(a) Three processes each with one thread. (b) One process with three threads.

Last year, the concept of thread has been introduced in Computing Infrastructure 1.

Threads can be considered as lightweight processes with their own control flow within the one process.



Each thread has its own program counter (own logical flow), its own stack (local variables), and its own registers.

The main difference with processes is that threads share code and data (e.g. from the heap).

Threads in OCaml

`create: ('a -> 'b) -> 'a -> thread`	`Thread.create funct arg` creates a new thread of control, in which the function application **`funct arg` is executed concurrently** with the other threads of the program. The application of `Thread.create` returns the handle of the newly created thread.
`id: thread -> int`	Return the identifier of the given thread. A thread identifier is an integer that identifies uniquely the thread.
`self: unit -> thread`	Return the handle for the thread currently executing.
`join: thread -> unit`	`join th` suspends the execution of the calling thread until the thread `th` has terminated.

Let's see some code!

https://github.com/ffelten/ocaml-snippets/tree/main/shared_memory_l
print_threads.ml

Non-determinism

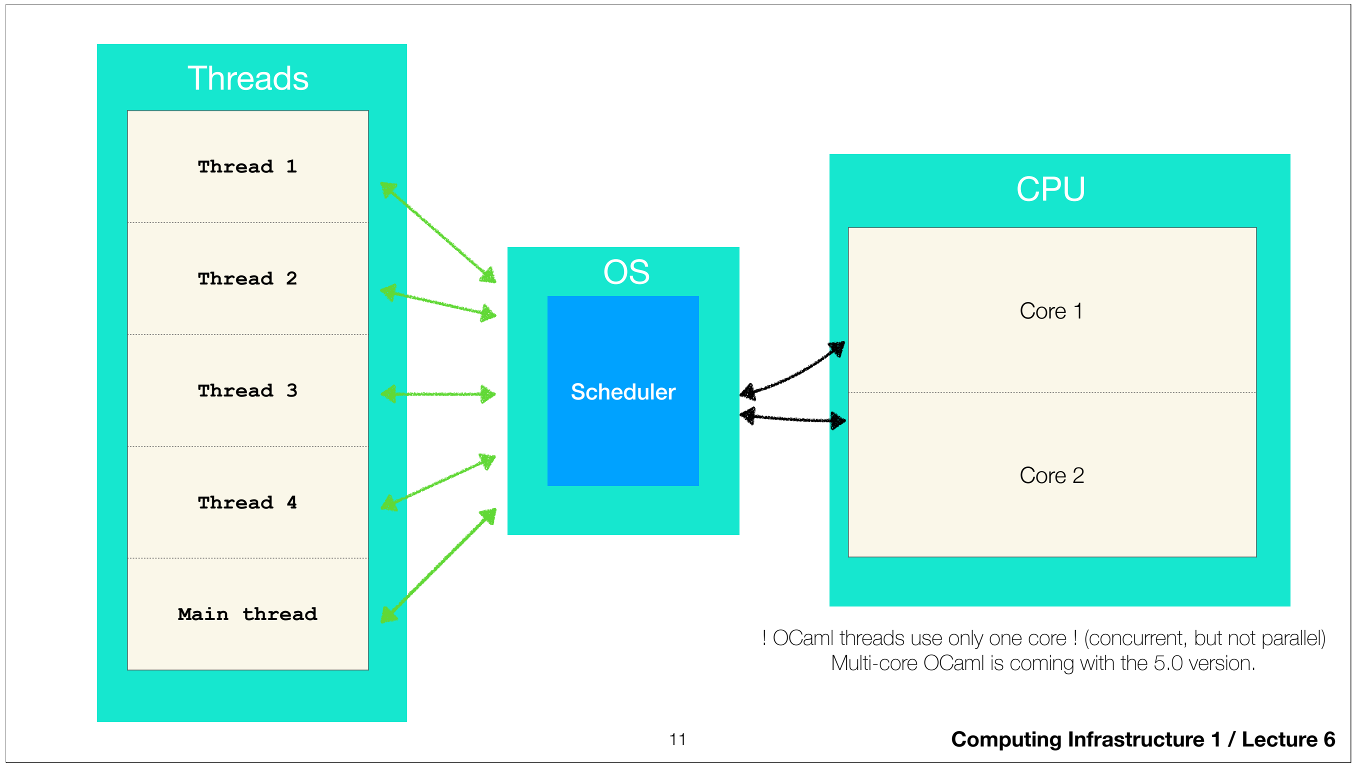
```
thread 1 starts processing.
thread 0 starts processing.
thread 0 done processing.
thread 1 done processing.
```

```
thread 0 starts processing.
thread 1 starts processing.
thread 1 done processing.
thread 0 done processing.
```


Why are the results different? It is the same code and the same inputs…

A program which always returns the same output for the same input is said to be *deterministic*.
A program which might return different outputs for the same input is said to be *non-deterministic*.

Threads scheduling

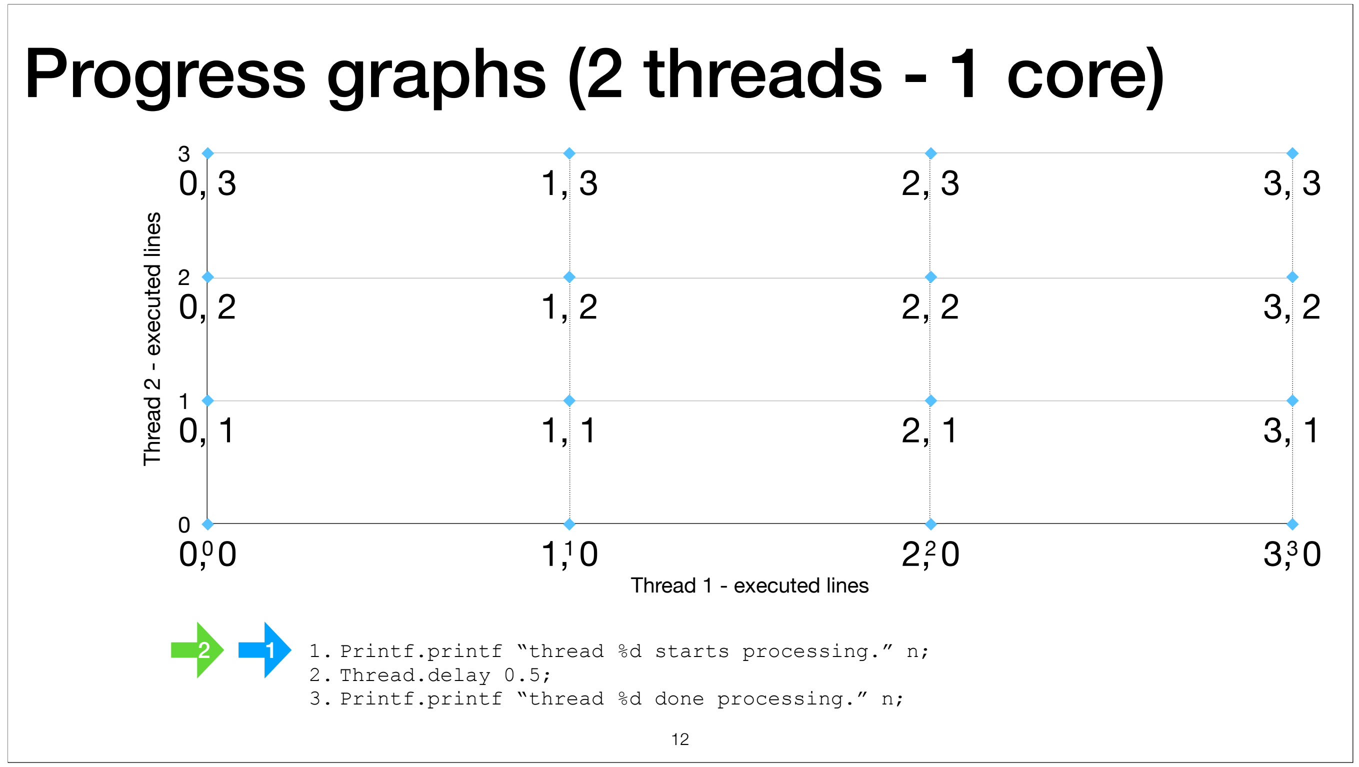


Remember from Computer Infrastructure 1. Processes are shared between processors, it is the Operating System that is responsible for context switches.

The same idea applies to threads: they share the CPU. It is the scheduler which is responsible to run and stop threads on the CPUs. From a programmer point of view, you can think of submitting one task in a queue when you launch a thread. It will eventually be picked and run until it is paused again or it has finished.

Note that it also works for multiple threads sharing one core - this is the difference between *concurrent (>= 1 core)* and *parallel (>1 core)* programs.

There are multiple ways to schedule threads, but it is out of scope for this course. Our main concern here is that the scheduling is non-deterministic. We cannot infer anything from the order in which the threads will be executed. This explains why we saw different outputs for the same program earlier.



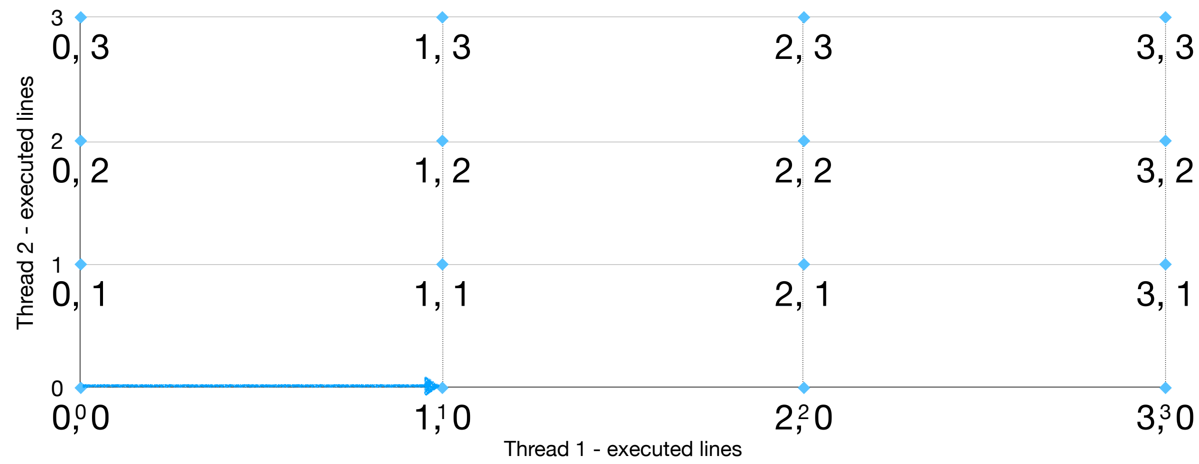
Progress graphs are a nice way to visualise the possible *interleavings* between thread executions.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible execution state (Line#, Line#). For example, (1, 0) means thread 1 *finished* operation at line 1 and thread 2 did not finish any.

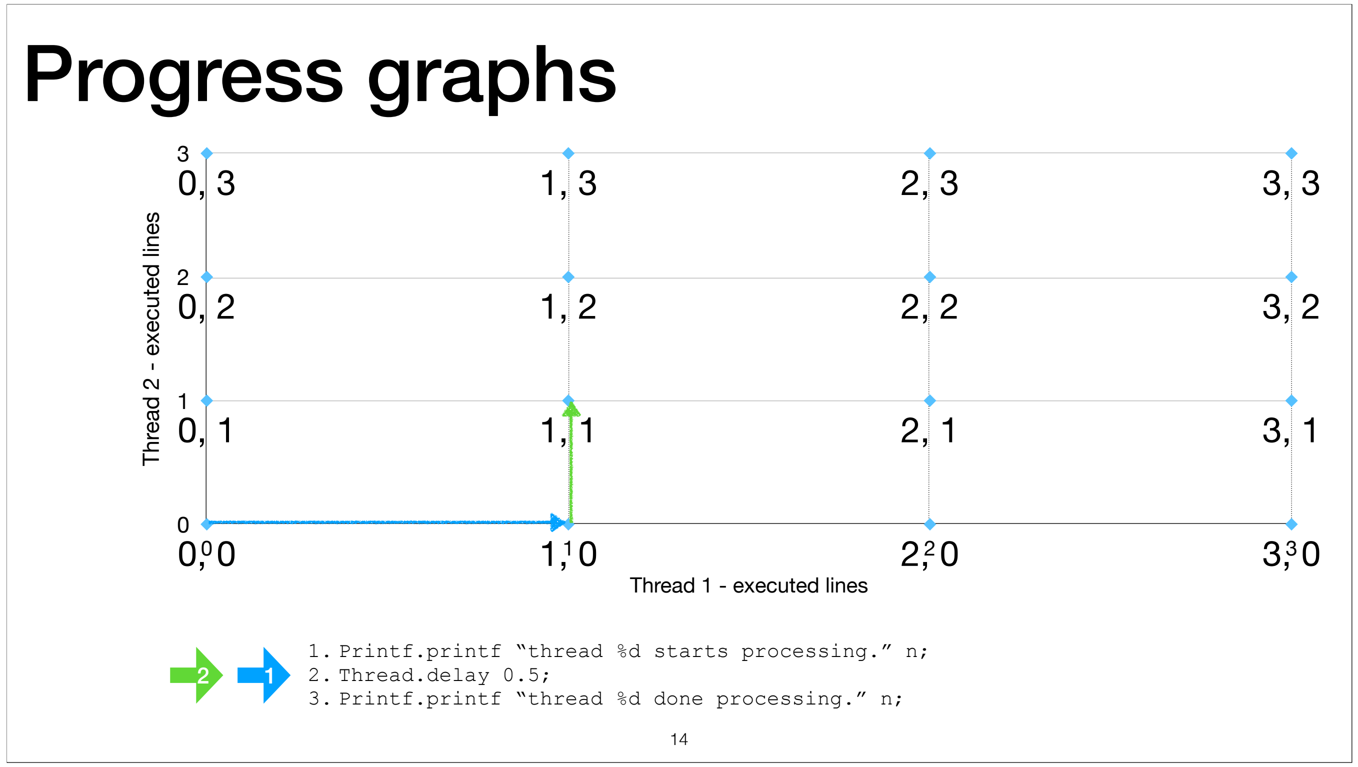
In this case, since we deal with only one CPU, we have a choice to execute instructions from thread 1 or thread 2 at each timestep.

Progress graphs



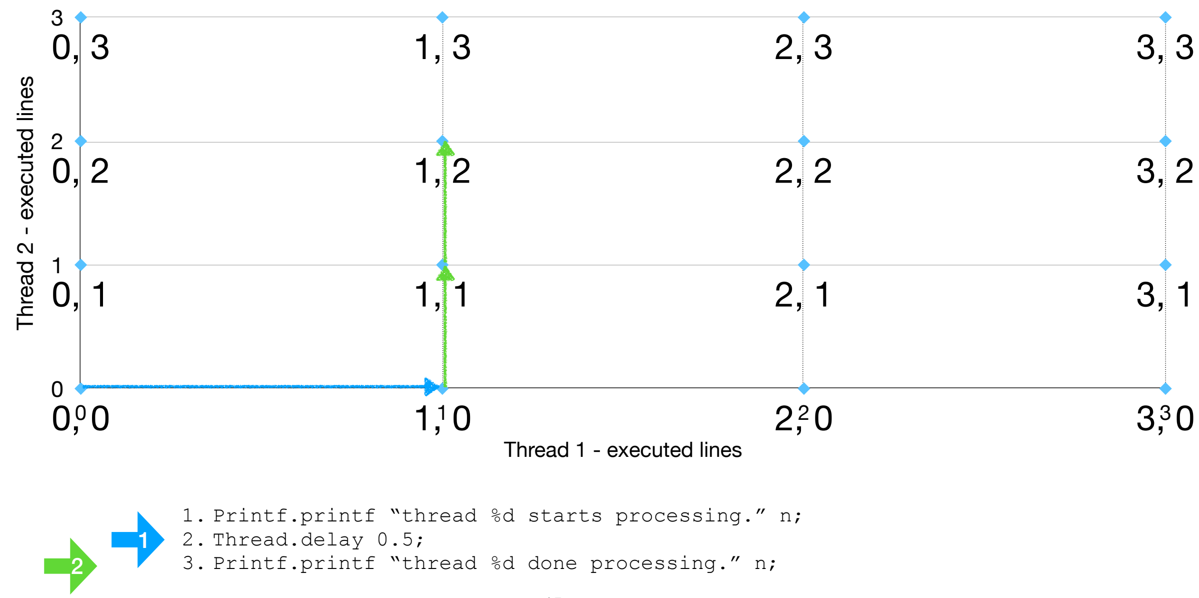

```
1. Printf.printf "thread %d starts processing." n;
2. Thread.delay 0.5;
3. Printf.printf "thread %d done processing." n;
```


Let’s say the scheduler first chooses thread 1 to be executed. We arrive to state (1,0), meaning thread one finished the first print while thread 2 did nothing.



Then, the scheduler chooses thread 2, which executes its print instruction.

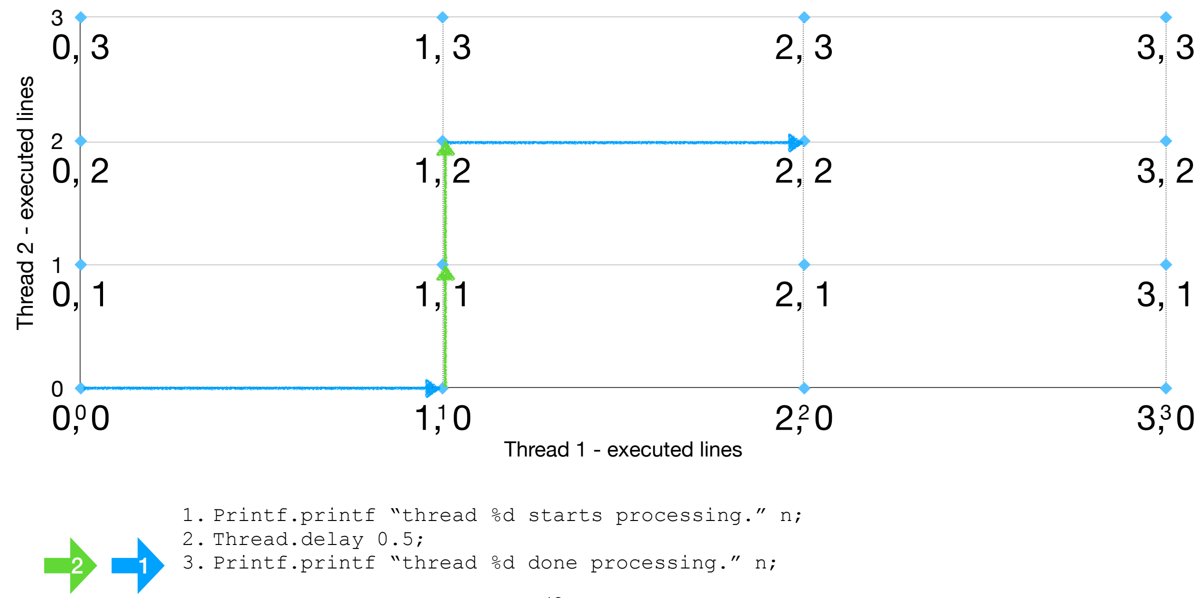
Progress graphs




```
1. Printf.printf "thread %d starts processing." n;
2. Thread.delay 0.5;
3. Printf.printf "thread %d done processing." n;
```


Thread 2 is chosen again.

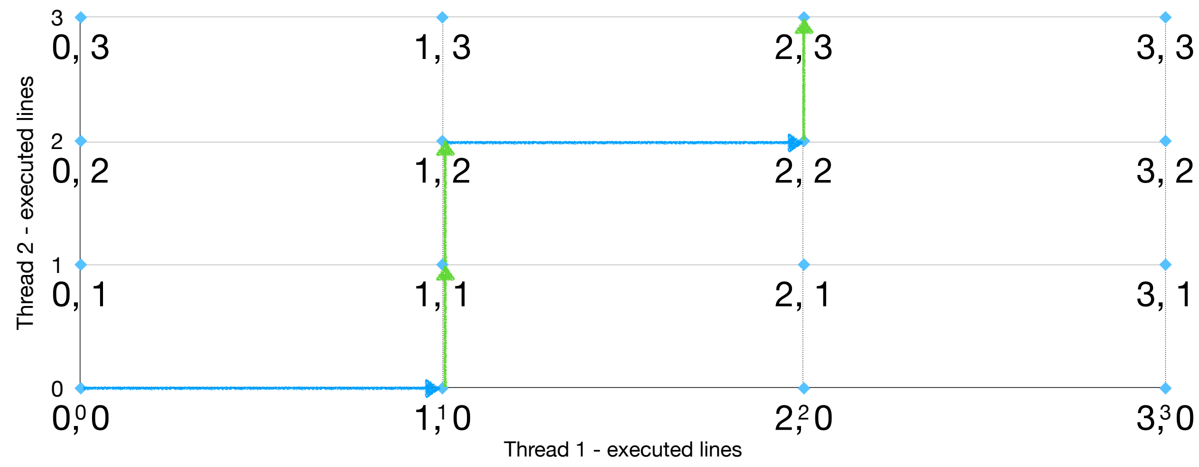
Progress graphs



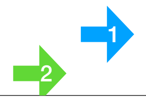

```
1. Printf.printf "thread %d starts processing." n;
2. Thread.delay 0.5;
3. Printf.printf "thread %d done processing." n;
```


Then thread 1…

Progress graphs

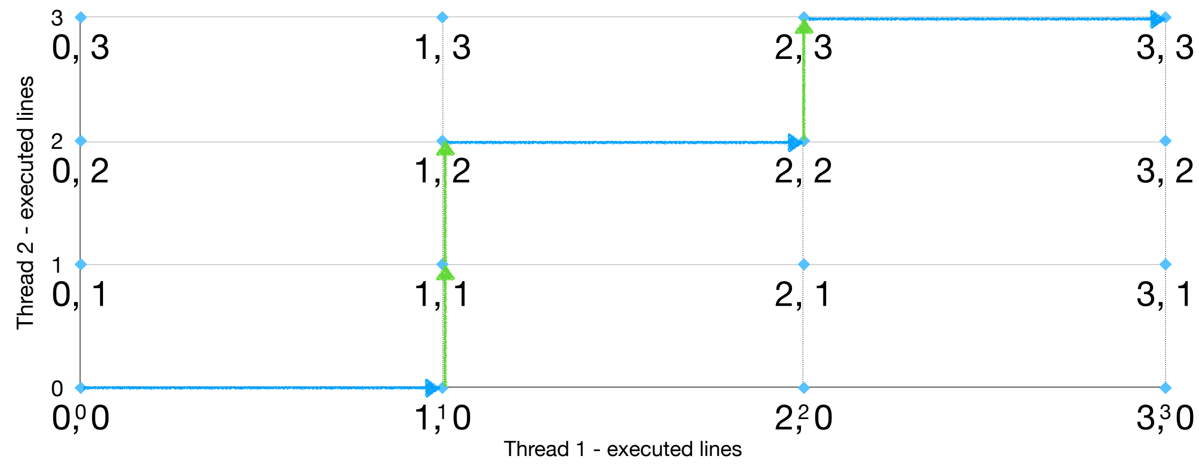



```
1. Printf.printf "thread %d starts processing." n;
2. Thread.delay 0.5;
3. Printf.printf "thread %d done processing." n;
```



Thread 2 is chosen and finishes its execution.

Progress graphs

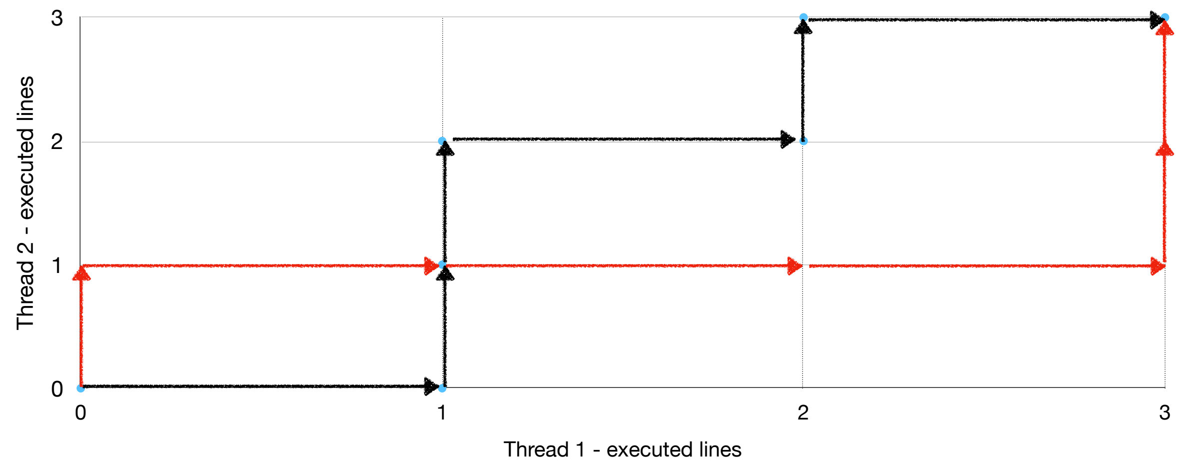



```
1. Printf.printf "thread %d starts processing." n;
2. Thread.delay 0.5;
3. Printf.printf "thread %d done processing." n;
```


Thread 1 finally finishes as well.

Trajectory

describes *one possible* execution of the threads




```
1. Printf.printf "thread %d starts processing." n;
2. Thread.delay 0.5;
3. Printf.printf "thread %d done processing." n;
```


A *trajectory* is a sequence of legal state transitions that describes one *possible* concurrent execution of the threads.

Should we care about this?

Let's see some code!

https://github.com/ffelten/ocaml-snippets/tree/main/shared_memory_l
increment_threads.ml

Non-determinism

The **many possible trajectories** in multi-threaded programs are at the source of **non-determinism**. When the output of a program changes depending on the trajectory it used, we say that the program is subject to race conditions.

For simplicity: non-determinism ~ same input, different output.

Shared memory synchronisation

Let us now interest ourselves into sharing memory in non deterministic world…

In computer science, we talk about *shared memory concurrency* when multiple threads or processes share access to data.

Note: **Compilers** might **rewrite code**

```
count += 1
```

```
LOAD count in R1
INC R1
WRITE R1 in count
```

Other languages are not safer than OCaml… 🥺

It is not because it looks like a single instruction that it is one!

This guy is also subject to non-determinism if executed on multiple threads…

Those two programs are exactly the same. In fact, the left part is rewritten as the right part during compilation.

Let us clearly identify the source of non-determinism.

Remember that threads *share* some memory (e.g. the heap), but have their own registers and program counter!

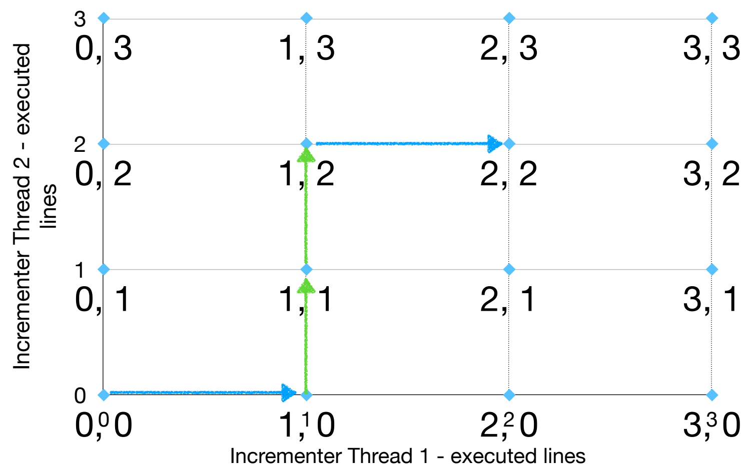
For this example, we examine one loop execution of 2 threads incrementing the counter.

Let’s say we start with a counter value of 0.

Thread 1 is the first to be executed, loading the content of count into R1

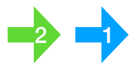
Thread 2 is then chosen to be executed, loading the value of count into its R1.

Thread 2 is chosen again, incrementing R1.



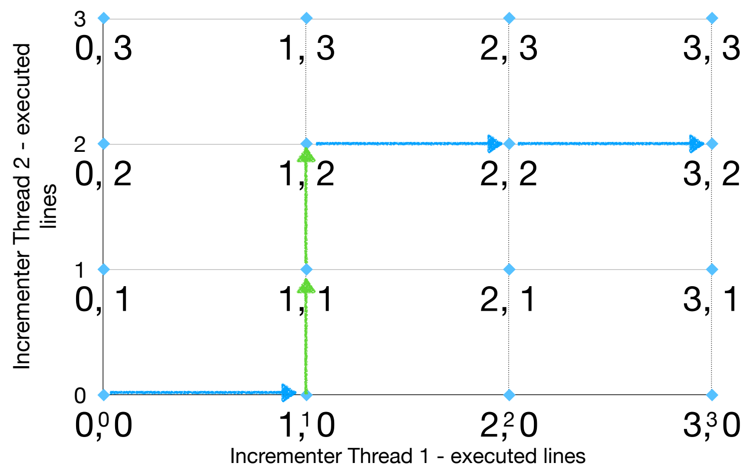
Memory Registers	
Thread 1	Thread 2
R1=1	R1=1

Shared Memory
count=0



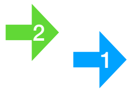

```
1. LOAD count in R1
2. INC R1
3. WRITE R1 in count
```


Thread 1 is chosen, incrementing R1.



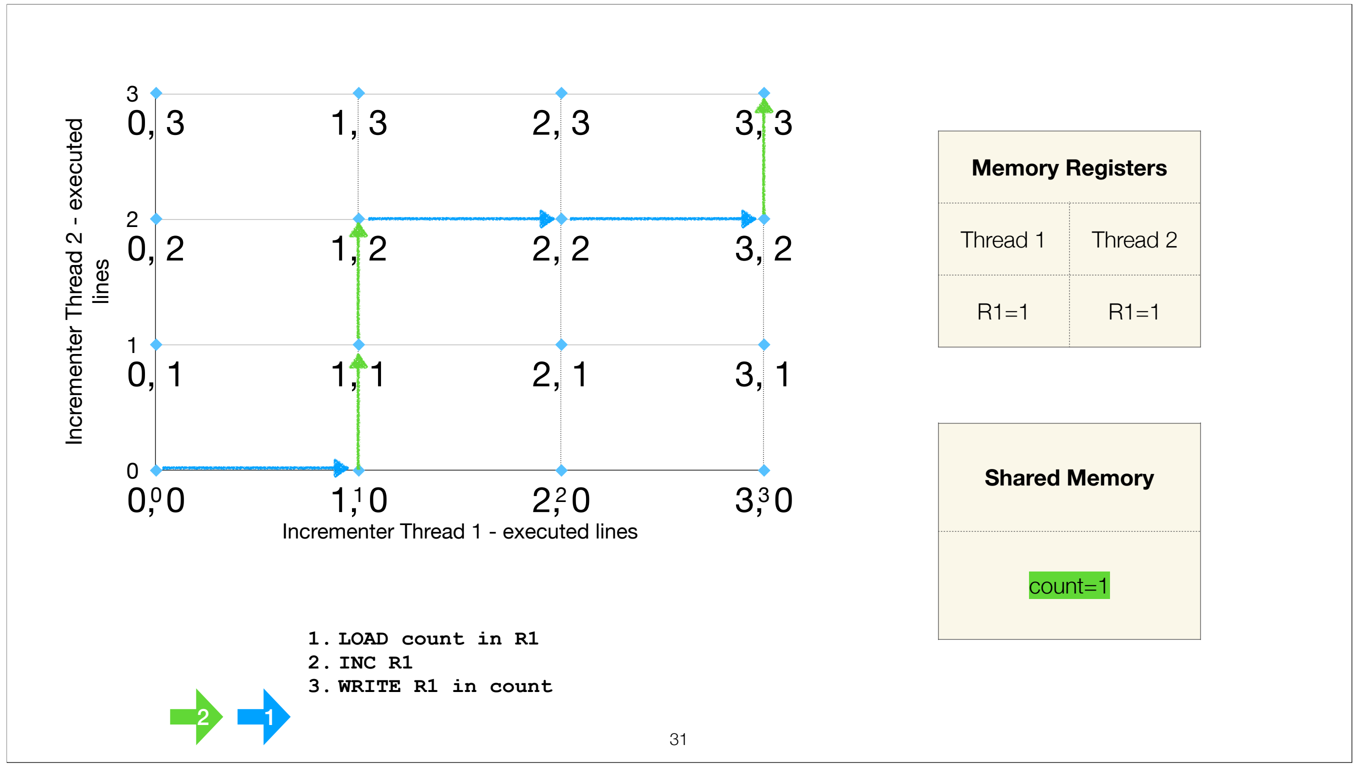
Memory Registers	
Thread 1	Thread 2
R1=1	R1=1

Shared Memory
count=1



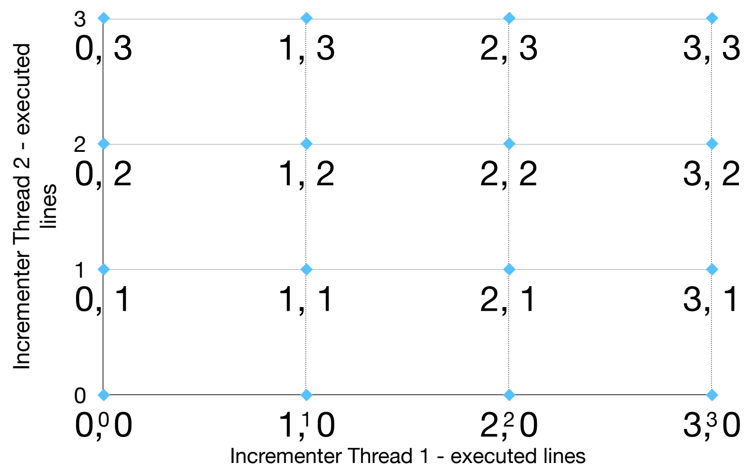

```
1. LOAD count in R1
2. INC R1
3. WRITE R1 in count
```


Thread 1 is chosen again, writing R1 into count.



Thread 2 is chosen, writing R1 into count, leading to no update :(.

Question




```
1. LOAD count in R1
2. INC R1
3. WRITE R1 in count
```

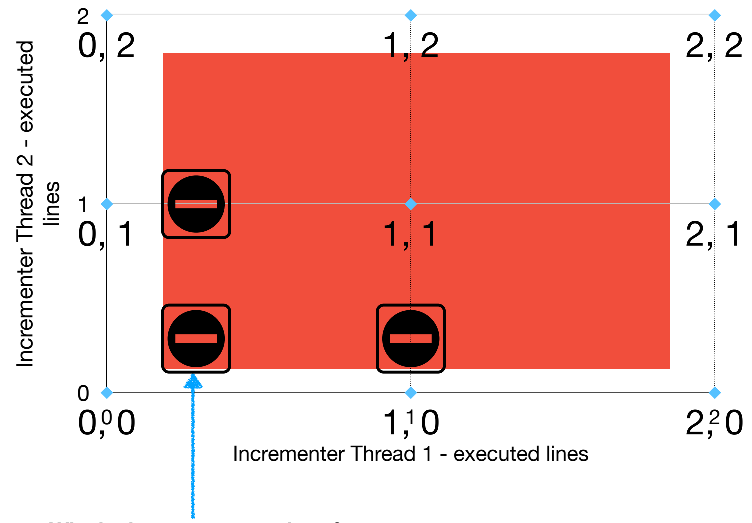
Memory Registers	
Thread 1	Thread 2

Shared Memory
count=0

Are there trajectories leading to the correct result?

Critical section

a block of code that can be accessed by only one thread a time.




```
for _= to n-1 do
  (* Danger zone - critical section *)
  let c = !count in
  …
  count := c + 1;
  (* End of danger zone *)
done
```

Why is there a no-entry here?

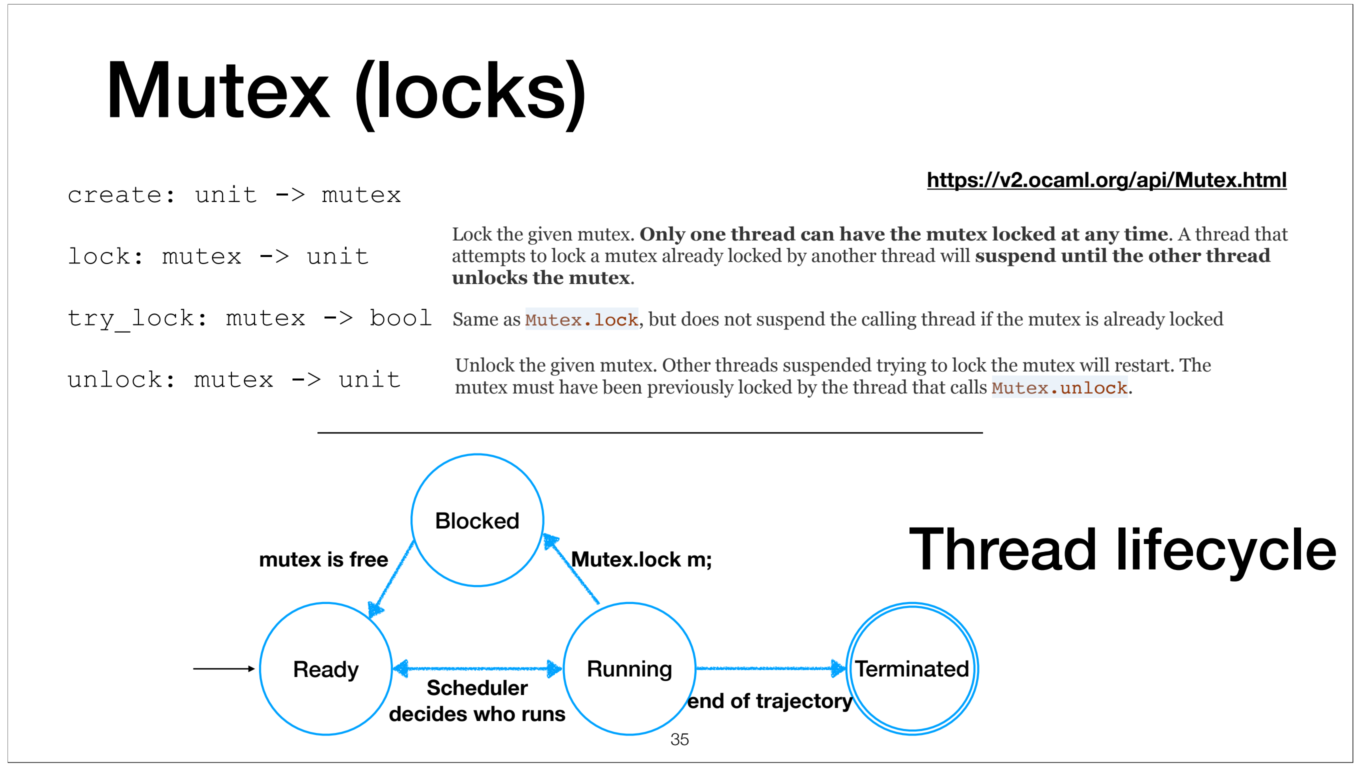
Safety

A program where all threads respect all the critical sections is said to respect the **mutual exclusion property.**

From those concepts, it seems that in order **to make our program correct, we need to define critical sections where we have shared mutable variable**. We also call this **protecting mutability**.

A code block is said to be **thread-safe** if it satisfies the mutual exclusion property. Our program can be considered thread-safe if all its code blocks are thread-safe.

Note that **we do not care about the order in which the threads are actually executed, our goal is to produce the correct result**!



Locks (also called mutex) are the most common answer to satisfy the mutual exclusion property in programming languages supporting multi-threading. You can think of a lock as an entity providing operations which are guaranteed to be executed by only one thread e.g. it is impossible for two different threads to return from the *lock* instruction at the exact same time. Once a thread possesses the lock, it can execute the critical section freely then release the lock. The lock ensures **all the other threads wait** before the critical section until it is freed again.

If you want to understand how locks are implemented, I suggest you to go through Chapter 2 of “The Art of Multiprocessor Programming”.

Now that we have locks, the *lifecycle diagram of threads* can be defined.

* Ready (initial state): thread has instructions to execute, waiting for a core;
* Running: the thread is currently running on a core;
* Blocked: the thread is blocked and cannot make any progress at the moment;
* Terminated: the thread finished to run all its instructions;

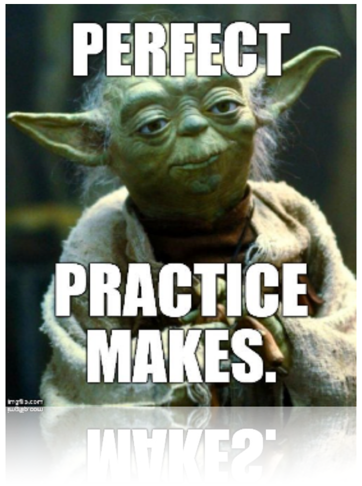
Let’s see some code!

https://github.com/ffelten/ocaml-snippets/tree/main/shared_memory_lock_increment_threads.ml

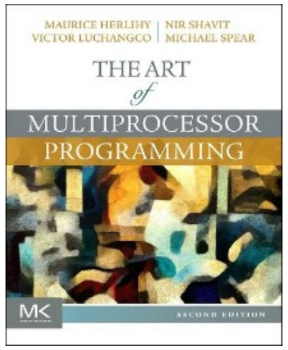
Take aways - did I hold my promise?

- ✔ We saw how to create multiple threads
- ✔ We saw the problems introduced by non-determinism
- ✔ We saw locks: a mechanism allowing to make multi-threaded programs determinist
- ✔ Looks like we're safe, aren't we? 😈

Exercises



Resources



Chapter 1 & 2