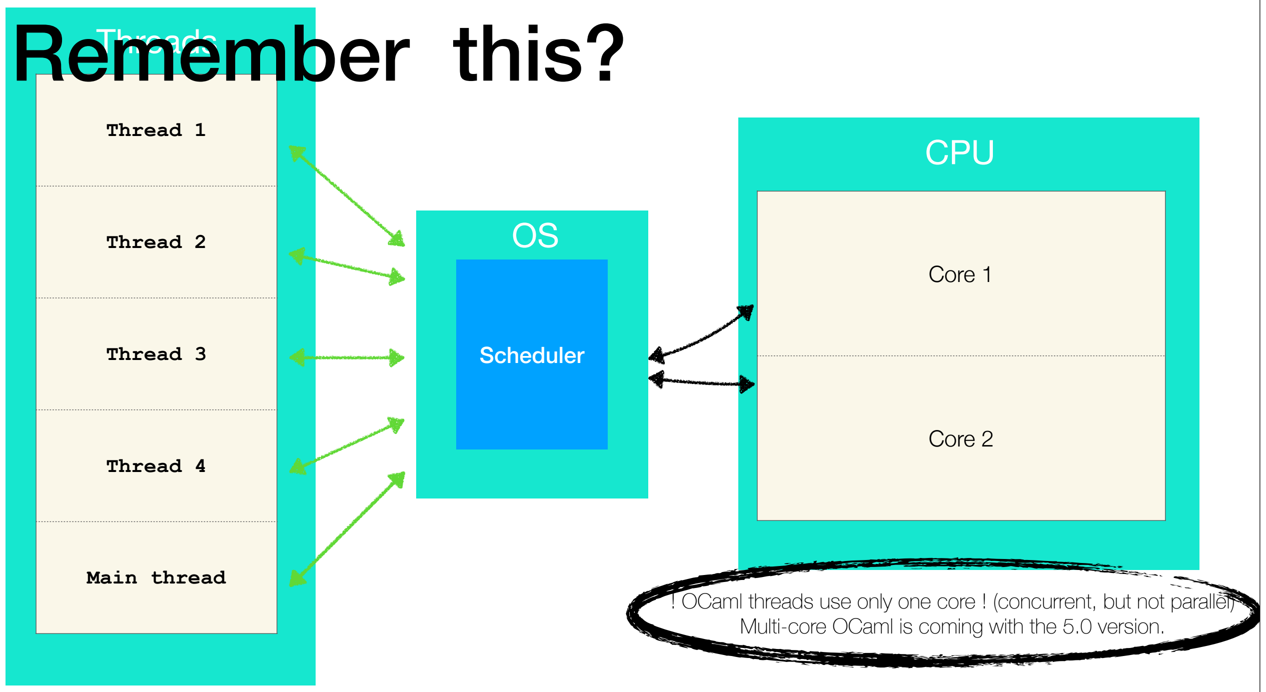
11 - Multicore OCaml

florian.felten@uni.lu

Remember this?

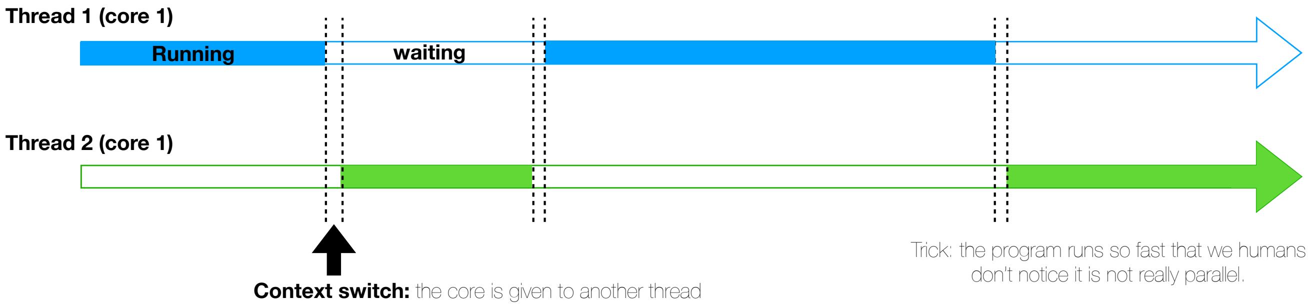


! OCaml threads use only one core ! (concurrent, but not parallel)
Multi-core OCaml is coming with the 5.0 version.

Let’s talk threads

How is it possible that OCaml use only one core but
still looks as if multiple things happen at the same time?

Concurrency = multiple threads (on >= 1 core)



Thread 1 (core 1)

Running

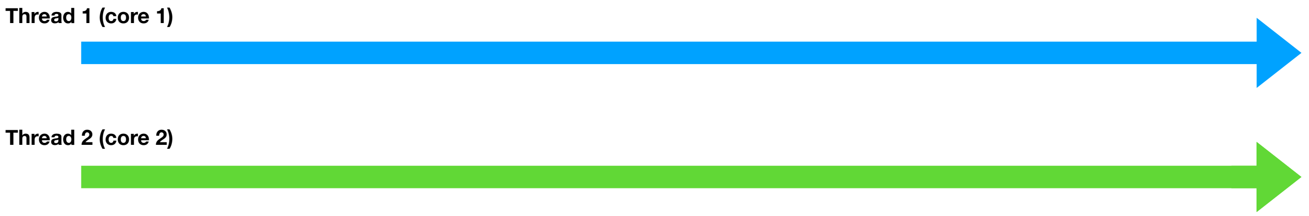
waiting

Thread 2 (core 1)

Context switch: the core is given to another thread

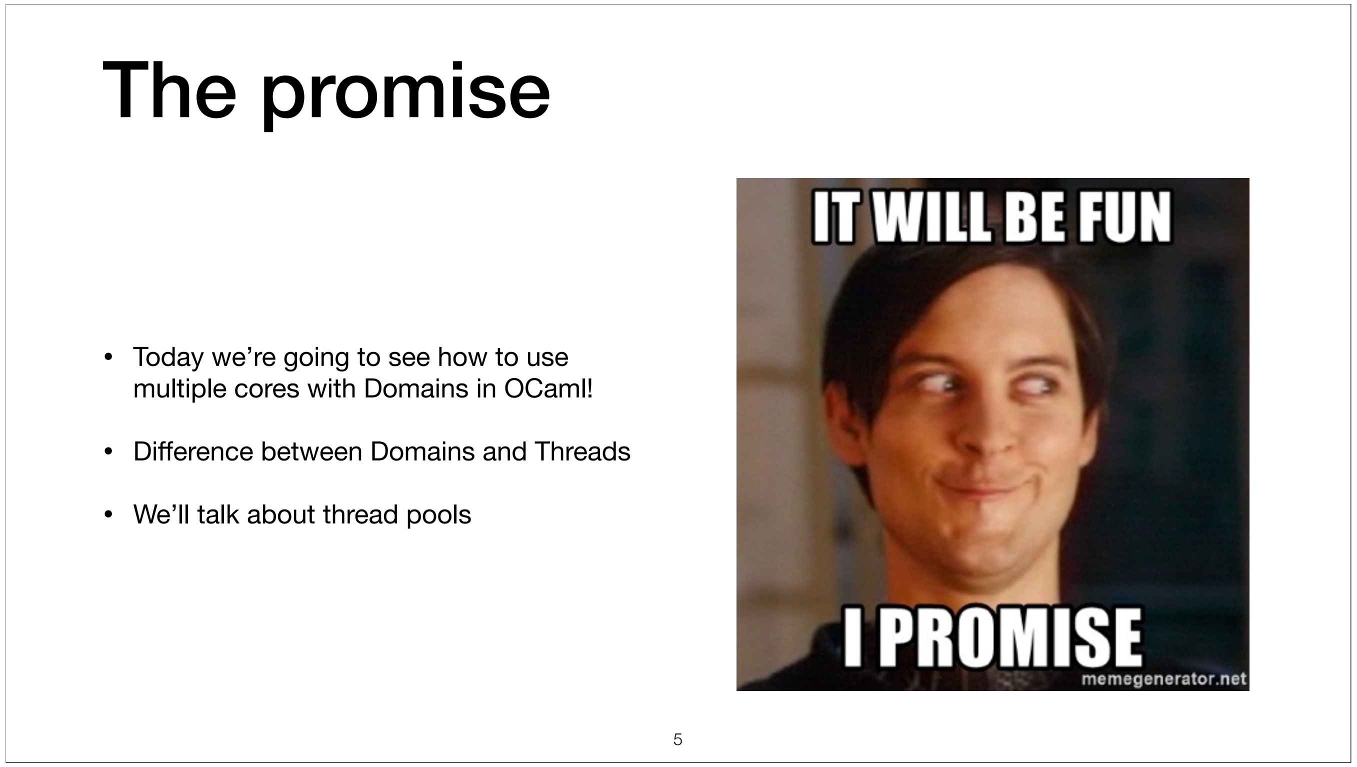
Trick: the program runs so fast that we humans don't notice it is not really parallel.

Parallelism = n cores shared by m threads



Thread 1 (core 1)

Thread 2 (core 2)



The promise

- Today we’re going to see how to use multiple cores with Domains in OCaml!
- Difference between Domains and Threads
- We’ll talk about thread pools

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

Domains

```
spawn: (unit -> ‘a) -> ‘a domain
```

`Domain.spawn funct` creates a new domain that runs in parallel with the current domain. Returns a handle of newly created domain.

```
join: ‘a domain -> ‘a
```

`Domain.join d` blocks until domain `d` runs to completion. **If `d` results in a value, then that value is returned by `join d`.** `If d` raises an uncaught exception, then that is re-raised by `join d`.

Let’s see some code!

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

fib.ml + parallel_fib.ml

So this whole Thread thing in OCaml is just a flaw. I better use a Domain every time.

– Naive programmer (2022 A.D.)

Shades of threads

Virtual thread

threads share one core

=> lightweight, low footprint

Thread (Python, **OCaml**, Oz)

Goroutine (Go)

Fiber, lightweightThread (Java)

Typical usage:
User interface (react to clicks, keyboard),
I/Os (don't wait for a file to open, receive file from the web)

Physical thread

thread is a core

=> heavyweight, takes time to instantiate 😢

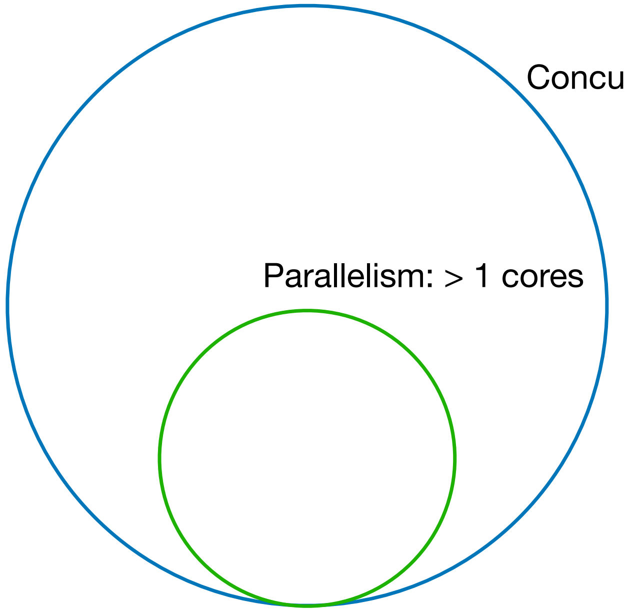
Thread (C, Java, C++)

Domains (**OCaml**)

~multiprocessing (Python)

Typical usage:
Heavy computations (DeepLearning, Parallel search, …),
Query processing in a server (better to use the full power).

Parallelism vs concurrency



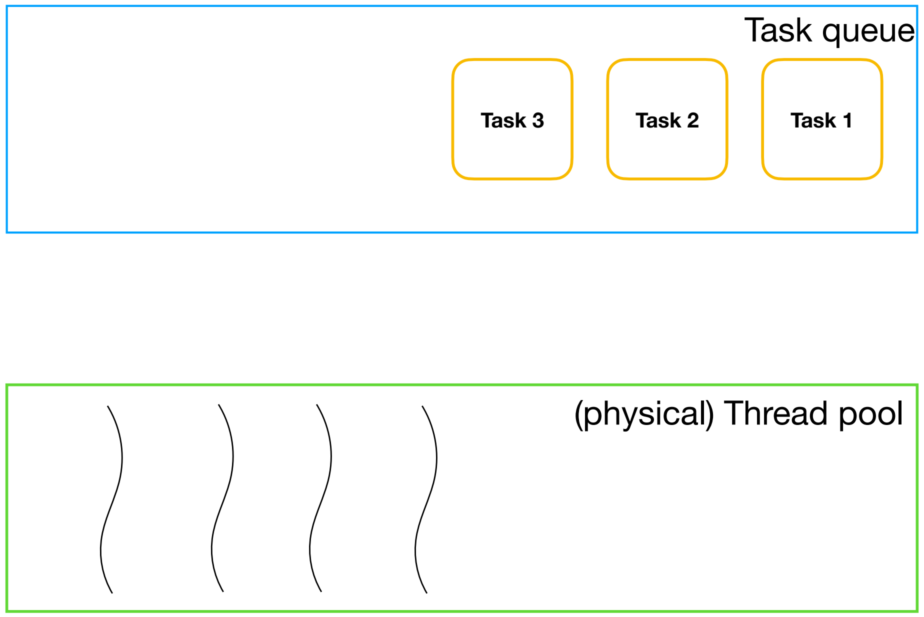
Parallelism is a subset of concurrency.

We have the same non-determinism problems when using multiple cores as when relying on context switching.

If we go to low-level, we have more problems with parallel e.g. cache misses are slow, sharing memory can be hard, … (we won’t see that here).

Thread pool

Using multiple cores with less overhead?
Reuse spawned domains!



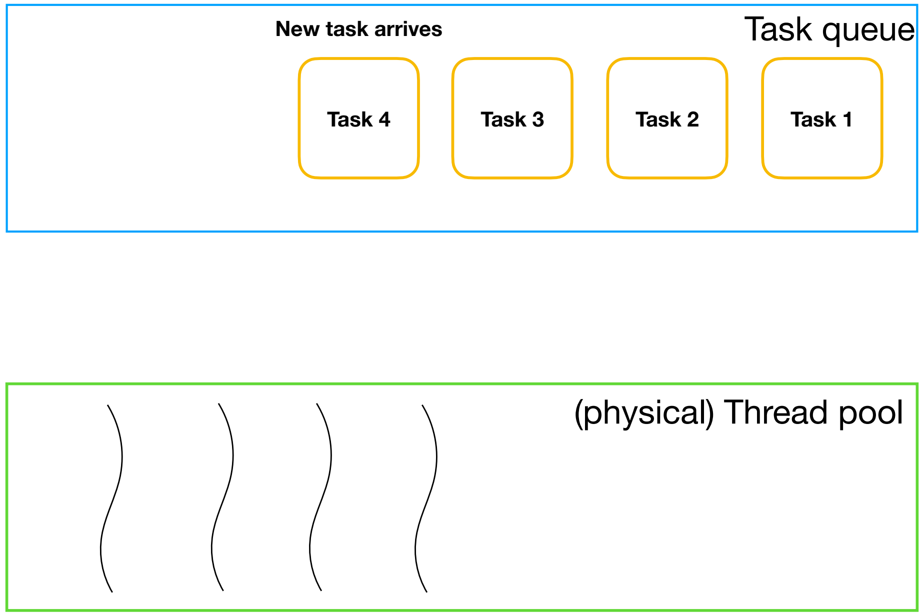
Task queue

Task 3

Task 2

Task 1

(physical) Thread pool



New task arrives

Task queue

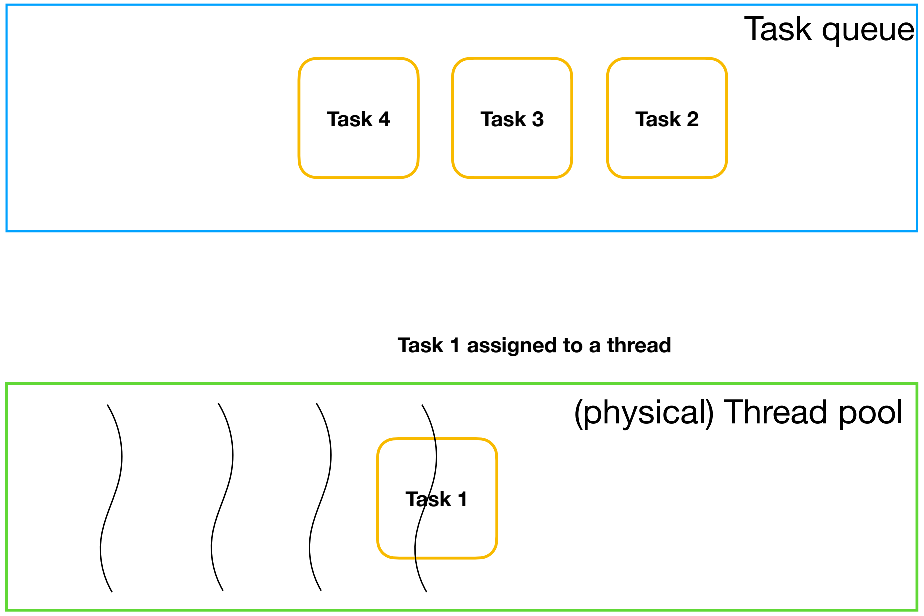
Task 4

Task 3

Task 2

Task 1

(physical) Thread pool



Task queue

Task 4

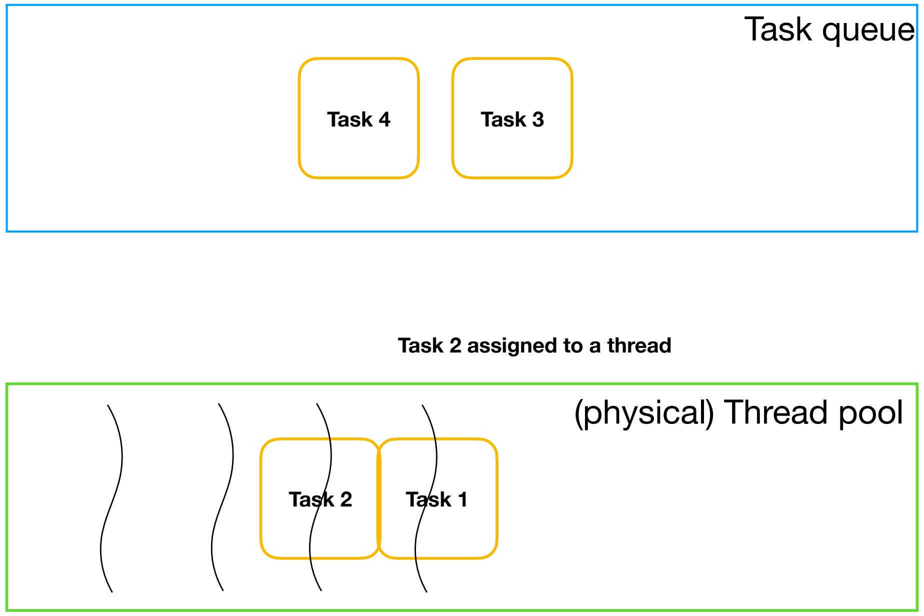
Task 3

Task 2

Task 1 assigned to a thread

(physical) Thread pool

Task 1



Task queue

Task 4

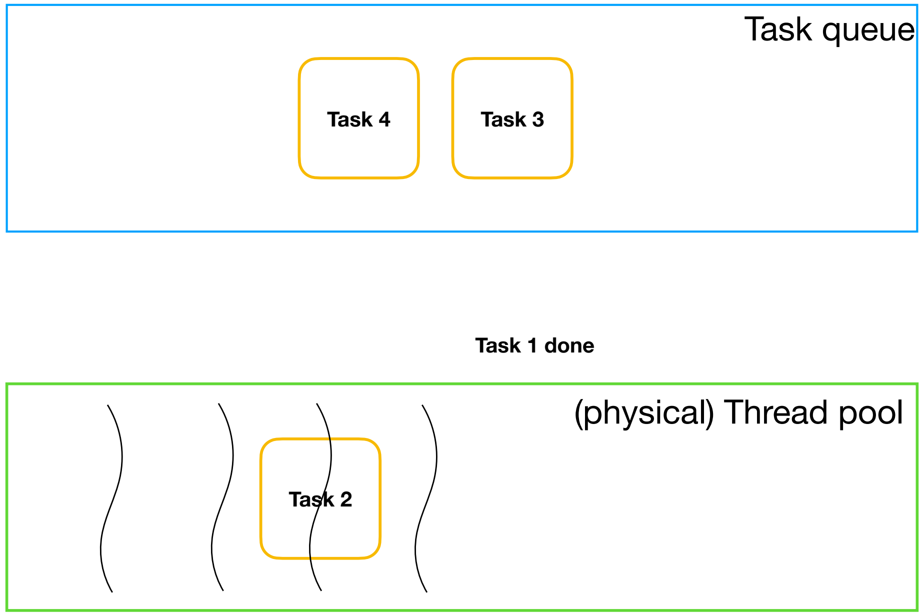
Task 3

Task 2 assigned to a thread

(physical) Thread pool

Task 2

Task 1



Task queue

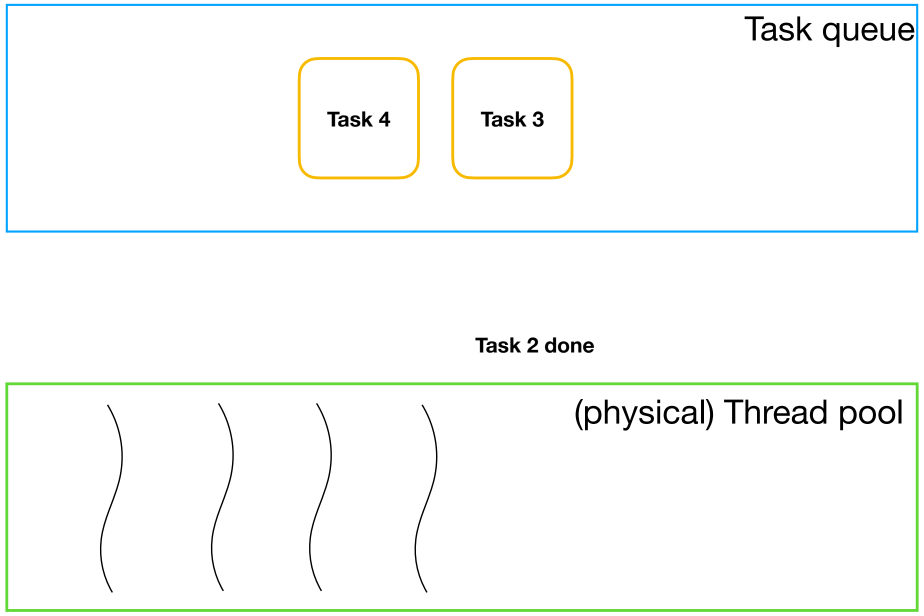
Task 4

Task 3

Task 1 done

(physical) Thread pool

Task 2



Task queue

Task 4

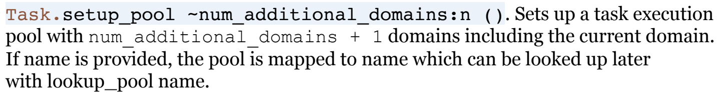
Task 3

Task 2 done

(physical) Thread pool

Task: OCaml thread pool usage

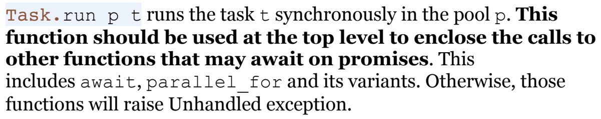
```
type ‘a task = unit -> ‘a

setup_pool:
  ?name:string ->
  num_additional_domains:int ->
  unit ->
pool
```



`Task.setup_pool ~num_additional_domains:n ()`. Sets up a task execution pool with `num_additional_domains + 1` domains including the current domain. If name is provided, the pool is mapped to name which can be looked up later with lookup_pool name.

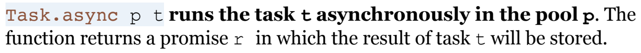
?name is an optional argument, num_additional_domains is a named argument

```
run: pool -> unit -> a -> a
```



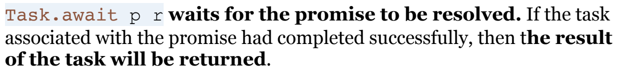
`Task.run p t` runs the task `t` synchronously in the pool `p`. **This function should be used at the top level to enclose the calls to other functions that may await on promises**. This includes `await`, `parallel_for` and its variants. Otherwise, those functions will raise Unhandled exception.

```
async: pool -> ‘a task -> ‘a promise
```



`Task.async p t` **runs the task `t` asynchronously in the pool `p`**. The function returns a promise `r` in which the result of task `t` will be stored.

```
await: pool -> ‘a promise -> ‘a
```



`Task.await p r` **waits for the promise to be resolved.** If the task associated with the promise had completed successfully, then t**he result of the task will be returned**.

https://github.com/ocaml-multicore/domainslib/blob/0.4.2/lib/task.mli

Let’s see some code!

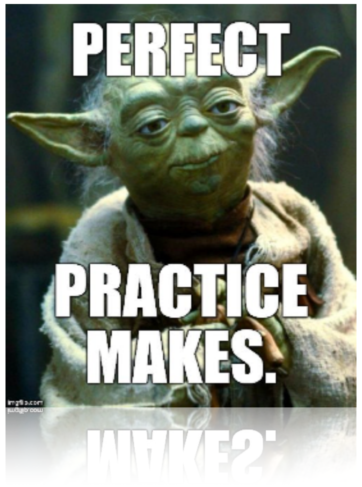
https://github.com/ffelten/ocaml-snippets/tree/main/

many_tasks_parallel.ml, pool.ml

Take aways - did I hold my promise?

- ✓ Concurrency => context switching on one core
  Parallelism => multiple cores
- ✓ Domain = core, Thread = Virtual thread
- ✓ Domains are expensive to instantiate, but they may be useful for long computations
- ✓ Thread pools allow to share instantiated domains to run multiple tasks in parallel.

Exercises



Resources

- https://github.com/ocaml-multicore/parallel-programming-in-multicore-ocaml
- https://github.com/ocaml-multicore/domainslib/blob/0.4.2/lib/task.mli
- https://kcsrk.info/ocaml5-tutorial/