10 - Shared memory concurrency II

florian.felten@uni.lu

Discussion

With the dining philosophers’ problem, you should have experienced issues when playing with multiple locks.

In the previous lecture, we saw how to make a program thread-*safe*. That is, by satisfying mutual exclusion.

In this lecture, we will go through the additional properties a program must satisfy to guarantee it terminates. Along with the properties, we will of course introduce a set of techniques.

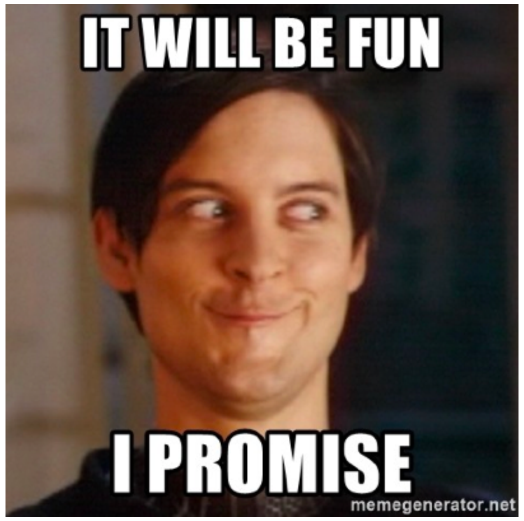
Let’s see some code!

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

philosophers_lock.ml

The promise

- You will be introduced to the problems which arise when blocking mechanisms are introduced;
- You will understand the properties a program should satisfy to be correct and terminate;
- You will be exposed to other forms of locks.



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

Deadlock

a situation which arises when all the threads are waiting for a resource currently held by another.

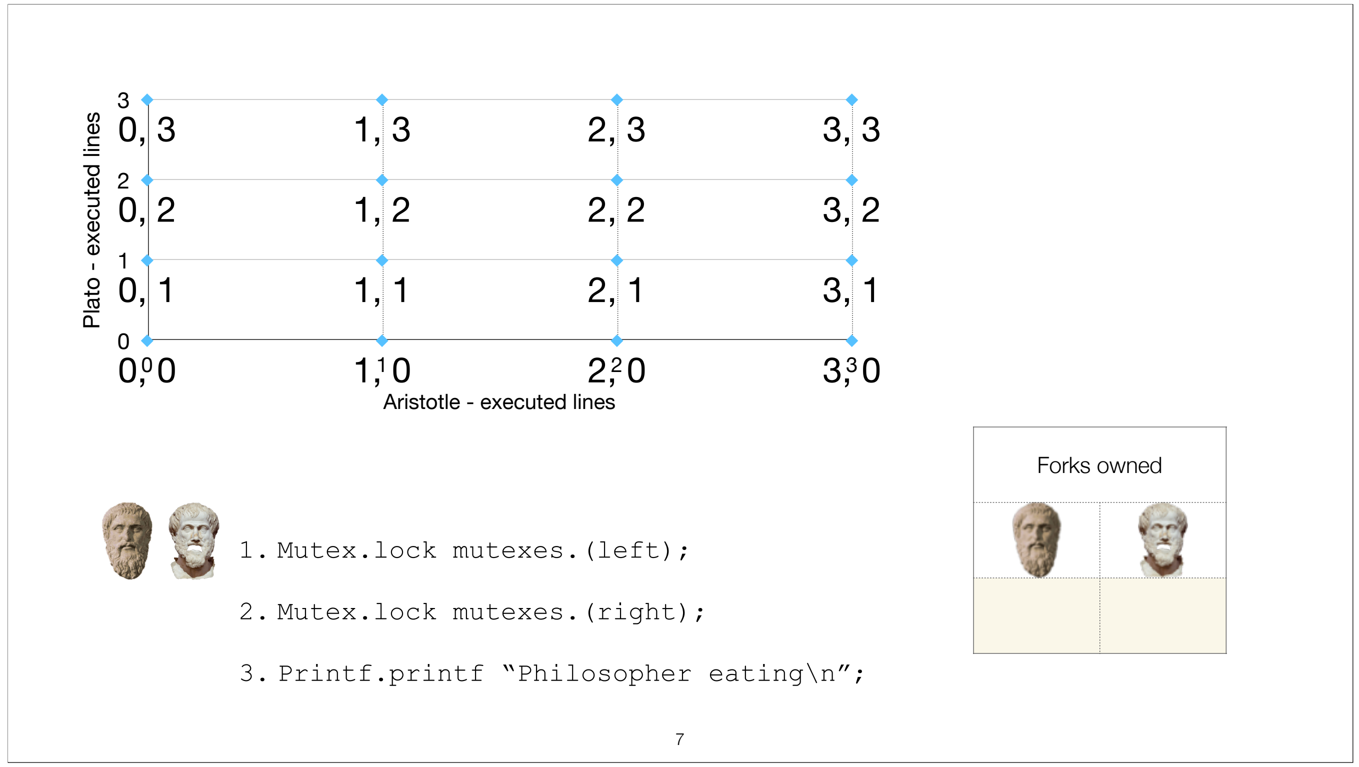
“Move your car, you’re blocking me”
- Unknown artists

When a deadlock happens, the program is stuck and will never finish.

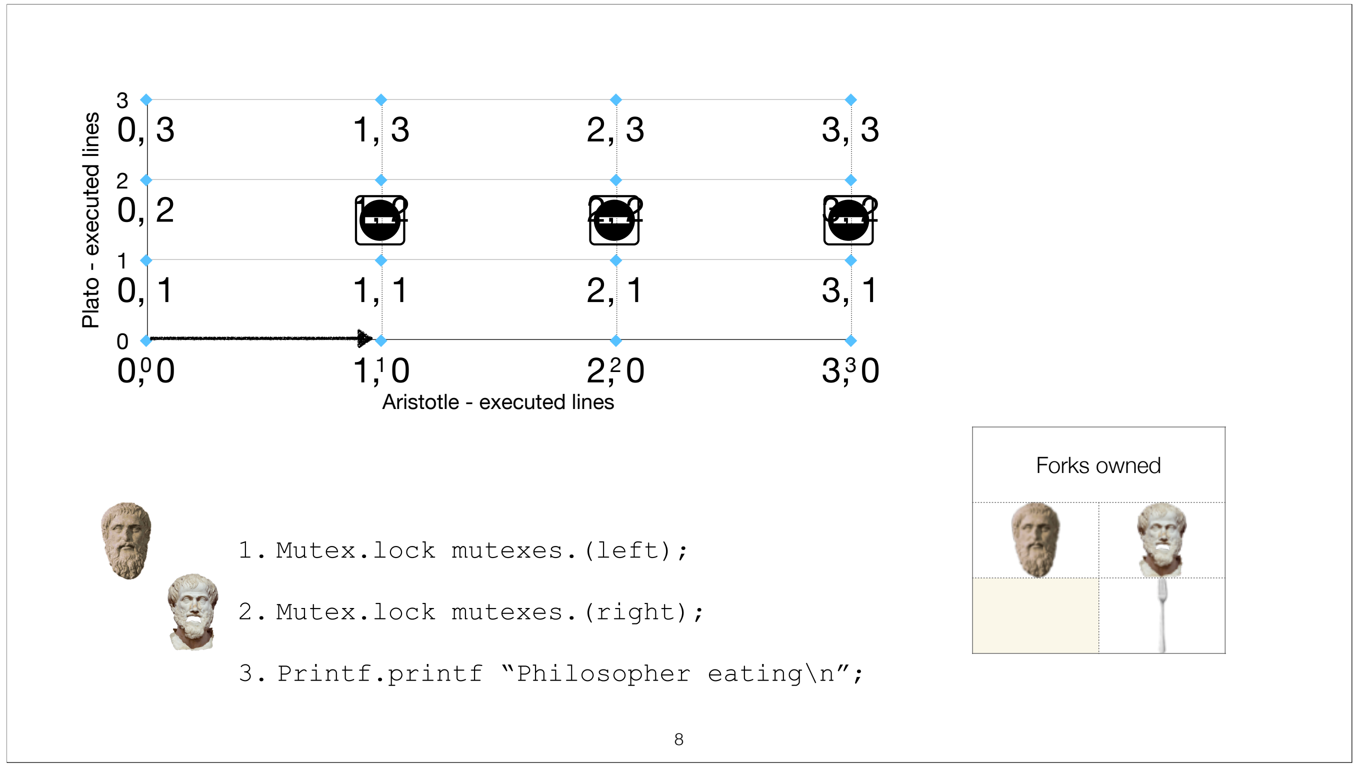
It can happen in computer programs, when all threads are blocked on some kind of locks. But it can also happen in the real world.

2 Dining philosophers rumbling

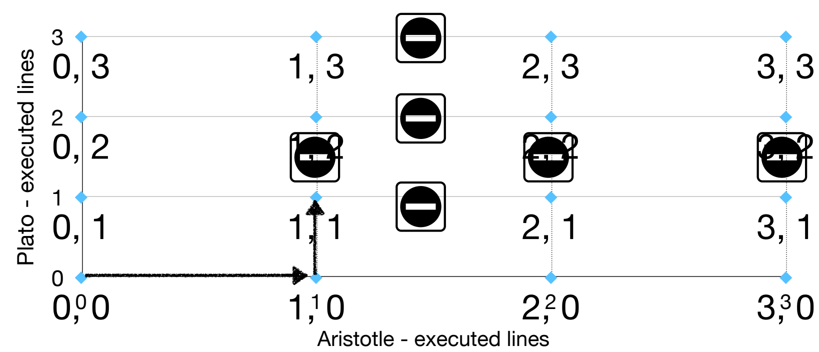
Let us interest ourselves to the dining philosophers problem with only 2 philosophers.



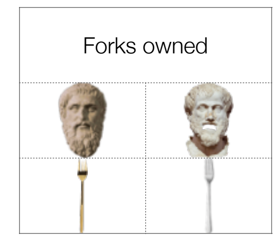
Let's see the execution of the program…



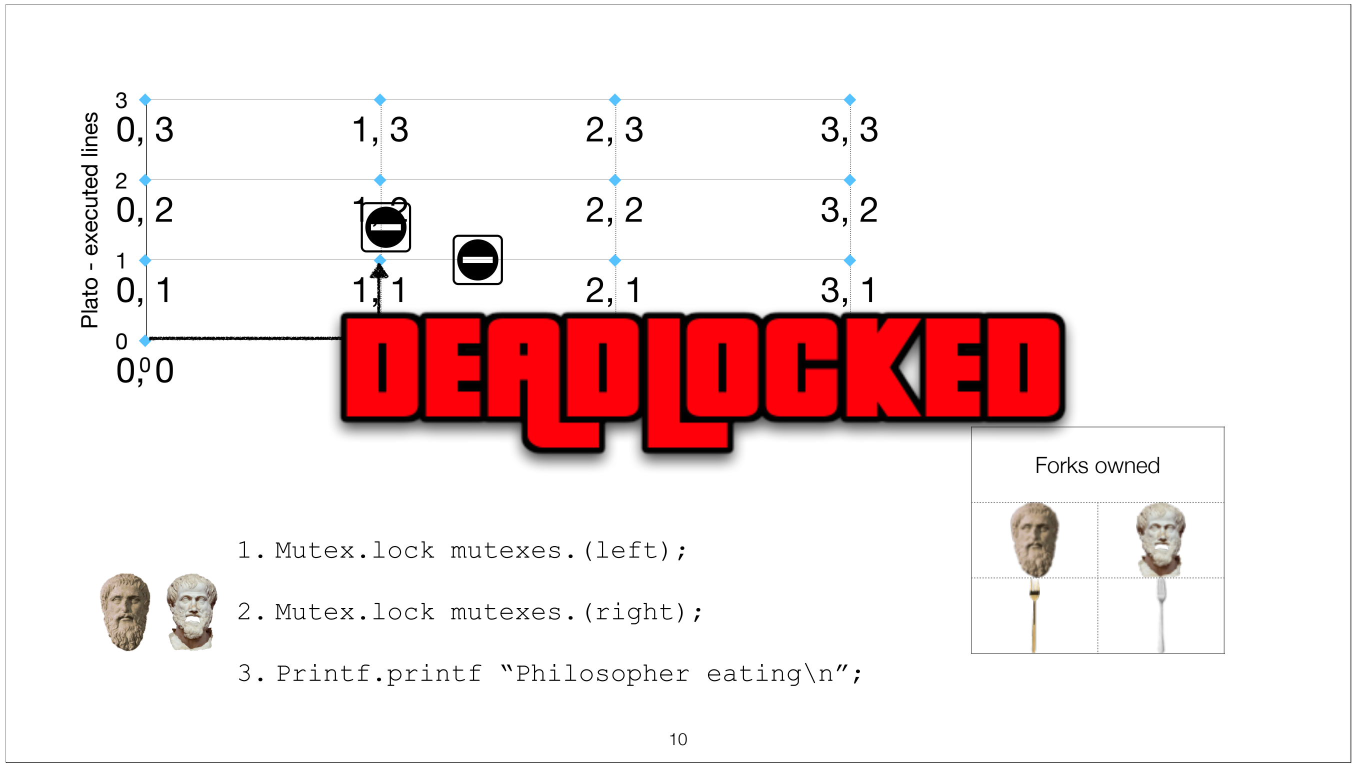
Aristotle is chosen to execute its first line, he takes its left fork (the bottom one).



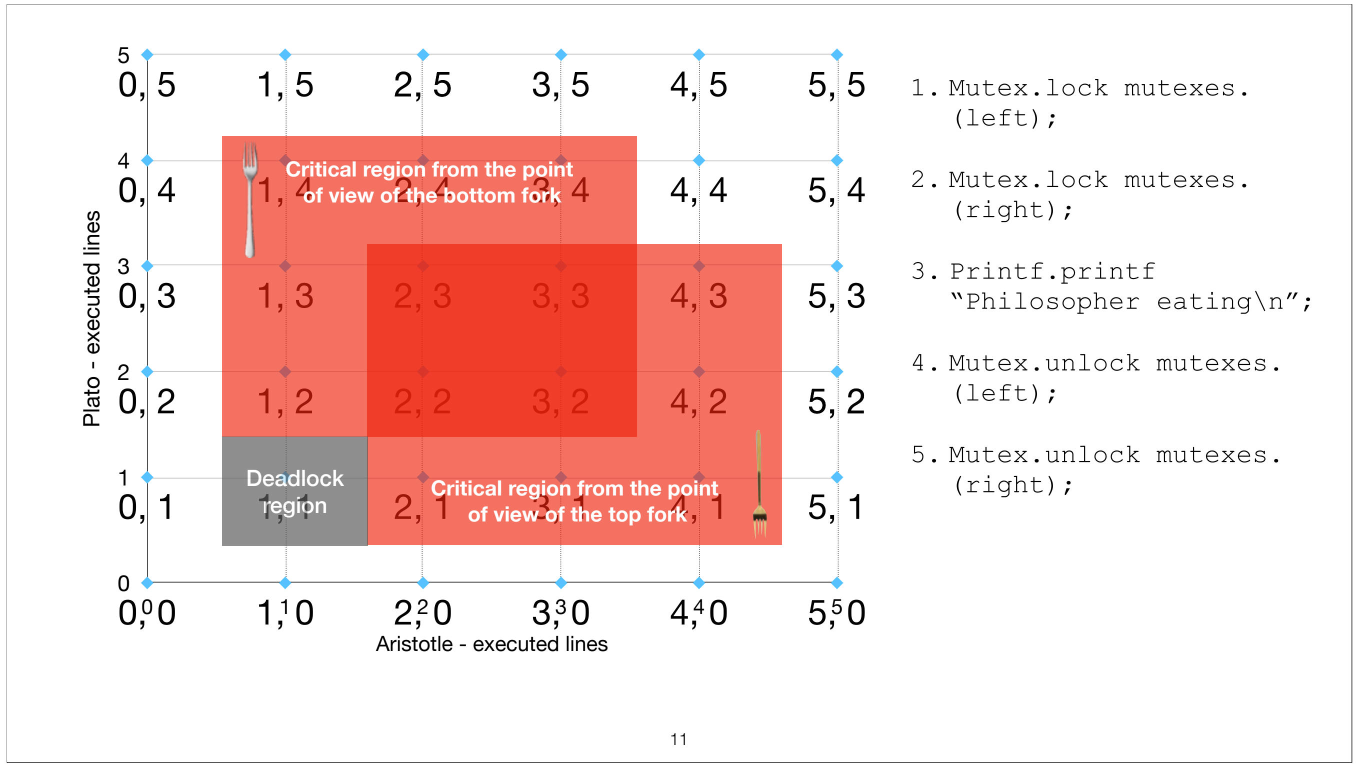
```
1. Mutex.lock mutexes.(left);

2. Mutex.lock mutexes.(right);

3. Printf.printf "Philosopher eating\n";
```



Plato is chosen next, he takes his left fork (the top one).



Looks like both philosophers are stuck waiting for a fork to be available. This is what we call a deadlock.



Since there are two different locks, there are two critical regions. We cannot lock both forks atomically so we have to do it in two different instructions.

Notice that forks are inverted for each philosopher. Aristotle sees the bottom fork at its left, while Plato sees it at its right.

This explains why a deadlock region appears in our graph.

How to fix deadlocks?

the 4 conditions to deadlock:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

For a deadlock to happen, there are 4 conditions to satisfy:

- Mutual exclusion; a fork can be hold by maximum one philosopher!
- Hold and wait; a philosopher who only has one fork keeps it until it gets the second one.
- No preemption; once a fork has been given to a philosopher, it is impossible to forcibly release it. Only the philosopher can make the choice to release it.
- Circular wait; there is a circular chain of philosophers waiting for fork held by another.

Let us see if we can lift one of these.

1. No mutual exclusion?

plates can be eaten without requiring both forks
losing safety 😭

No mutual exclusion means multiple threads can enter critical sections as they wish and the program would stay correct… Given the current setup, it means removing locks. It seems we will lose our thread safety property if we do that.

Note that there exists data structures which allow concurrent modification without the need for any lock to keep the program safe. You can lookup the concept of “Conflict-free replicated data type” or “Atomic variables” if you are interested to read further.

2. No hold and wait?

a philosopher who has one fork can decide to release it if he cannot acquire the second one

This works and actually exists. The idea is that if a thread is blocked, it can decide to release the locks it has already acquired to resolve the locking situation.

Let's see some code!

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

Output


```
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
Philosopher 0 is thinking.
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
Philosopher 0 is thinking.
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
Philosopher 0 is thinking.
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
Philosopher 0 is thinking.
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
Philosopher 0 is thinking.
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
Philosopher 0 is thinking.
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
Philosopher 0 is thinking.
Philosopher 1 could not lock right. Releasing fork.
Philosopher 1 is thinking.
Philosopher 0 could not lock right. Releasing fork.
```

Livelock

a blocking situation where each thread blocked by another is actively trying to resolve the issue on its own.

A *livelock* is a situation where each thread holding a resource releases it if it cannot obtain the next lock. The problem occurs when there is an execution sequence where the threads end up in only exchanging resources, without doing any progress.

You can think of a situation where two people cross each other in a narrow corridor, one of them has to take left and the other right. A livelock situation is when both people choose the same side at the same time, bumping into each other forever.

As we saw in the no hold & wait philosophers' example, it would be a situation where when the deadlock appears, both philosophers decide to release their fork, saying "you eat first" to the other. There is a trajectory where the two philosophers take the first fork and release it forever.
Live: they are doing something
lock: but they do not make any progress overall.

To conclude on the no hold and wait section, it is possible to do it but it requires a bit more refinements :-). You can have a look at the concept of "Monitors" if you want a working example.

3. Preemption?

There is a waiter who can forcibly take the fork from the philosopher

The idea here would be to have the system recognising the program is in a deadlock and fixing it by transferring locks. The problem is that it might lead to losses.

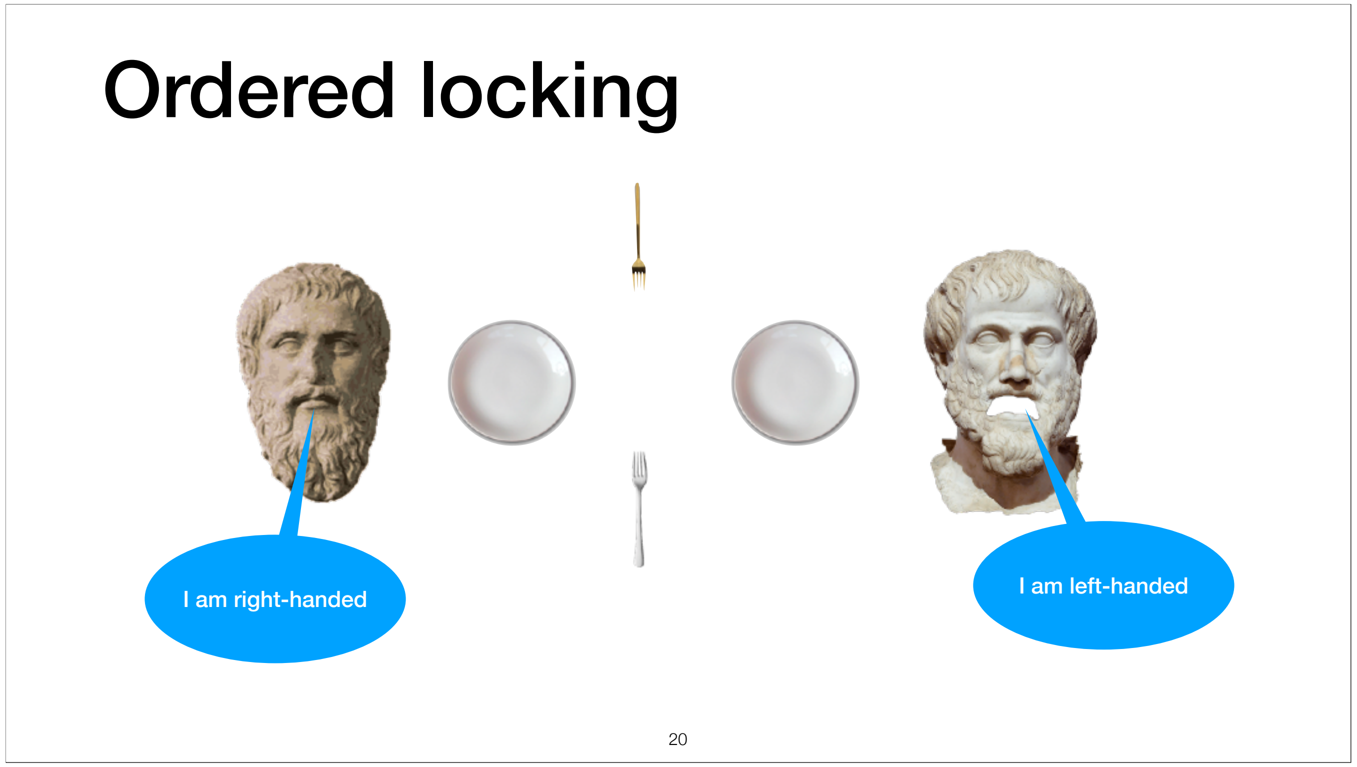
In the philosophers' problem, this would translate by an external person forcing one philosopher to put the fork down when both philosophers are stuck.

You can read further in http://boron.physics.metu.edu.tr/ozdogan/OperatingSystems/week8/node20.html.

4. Breaking circular dependencies?

This solution aims at breaking the potential circular dependencies between thread asking for resources.

There are multiple ways to do so. We will discuss these in the following slides.



The simplest way is to define an ordering of resources and to always lock in the same order. In practice, it is not so easy as conditional statements can have influence on which locks are needed - so you don't know which lock you will take a priori.

Going back to our philosophers' deadlock; the fix is to let one of them start by taking the right fork.

The idea of coarse-grained locking is to group various locks under a “bigger one”, breaking circular dependencies 🤓. It means multiple resources can be protected by one lock. This technique is used quite a lot in practice due to its simplicity.

While this simplifies our problem, it can have significant performance issues. The extreme example is having **one** lock to protect **all** mutable resources… Well it makes our program sequential (remember our initial goal was to get out of sequentiality 🧐).

Starvation

Another issue might arise when playing with multiple threads…

Another way to resolve the philosophers' problem is simply to make one starve. The idea is that this philosopher is never taking any fork. For example, if one is faster than the other to take forks when they are available. Is it really *fair* though?

Formally, in computer science, a program satisfies the property of *fairness* when no thread ever suffer from *starvation*. *Starvation* is a situation where a thread is unable to make any progress.

Properties

Safety ↔ the program gives the correct result

mutual exclusion

➡ Use locks.

Liveness ↔ the program terminates

deadlock-free & livelock-free

➡ Careful with locks:
No hold & wait, break circular dependencies, preemption, no mutual exclusion.

Fairness ↔ every thread gets his chance

starvation-free

➡ Release the locks!

A real world anecdote

“Told ya”
- Ptolemy

Prof. Edward A. Lee

From https://ptolemy.berkeley.edu/
The Ptolemy project studies **modeling, simulation, and design of concurrent**, real-time, embedded **systems**. The focus is on **assembly of concurrent components**. The **key** underlying **principle** in the project is the use of **well-defined models** of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation.

Ptolemy

Prof. Edward A. Lee

In the early part of the year 2000, my group began developing the kernel of Ptolemy II, a modelling environment supporting concurrent models of computation.

The challenge was to ensure that no thread could ever see an inconsistent view of the program structure. The strategy was to use Java threads with monitors.

code reviewed by experts

automated tests

design reviews

code coverage metrics

code maturity rating system

nightly builds

Deadlock after 4 years

"The fact that a problem as serious as a deadlock that locked up the system could go undetected for four years despite the good practice is alarming."

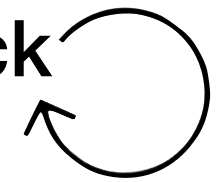
- E. Lee, "The Problem with Threads"

For further information, you can read the excellent article:
E. Lee, “The Problem with Threads,” Computer, vol. 39, pp. 33–42, Jun. 2006, doi: 10.1109/MC.2006.180.

Shades of lock

Mutex locks are not the only form of locks. The following slides present a few more forms of locks.

ReentrantLock



`lock: mutex -> unit`

Before 4.12 `Sys_error` was not raised for recursive locking (platform-dependent behaviour)
Raises `Sys_error` if the mutex is already locked by the thread calling `Mutex.lock`.

(!) The behaviour in those cases depend on the implementation. In other languages, it will be different e.g. no exception is raised.

Semaphore ==Counter + Lock

1962 by
Edsger W. Dijkstra

```
make: int -> semaphore
```

`Semaphore.Counting.make n` returns a new counting semaphore, with initial value `n`. The initial value `n` must be nonnegative.

```
release: semaphore -> unit
```

`release s` increments the value of semaphore `s`. If other threads are waiting on `s`, one of them is restarted.

```
acquire: semaphore -> unit
```

`acquire s` blocks the calling thread until the value of semaphore `s` is not zero, then atomically decrements the value of `s` and returns.

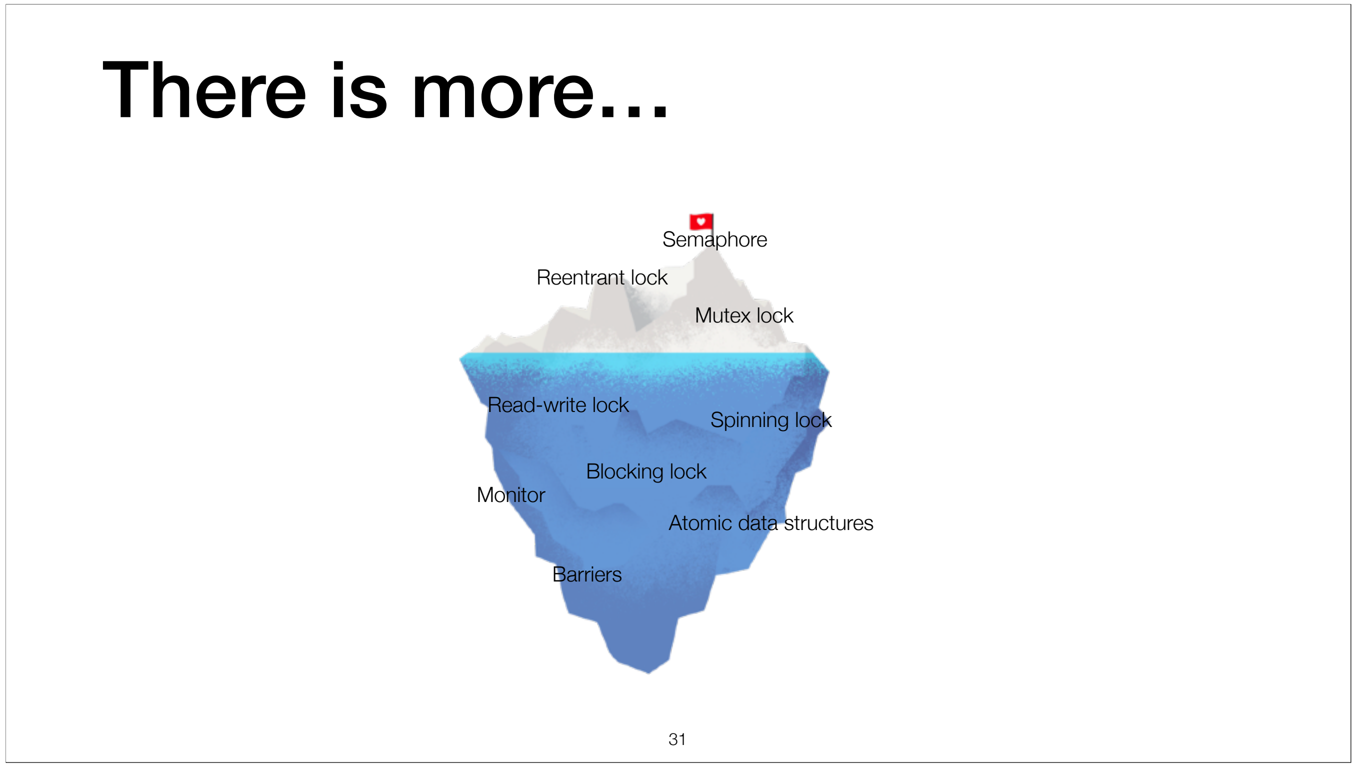
Note `Semaphore.Binary` is actually a mutex.

https://v2.ocaml.org/api/Semaphore.Counting.html

The original concept comes from the maritime world, where they use flags to communicate. The idea is to tell the boats to enter the harbour or not depending on the space left.

The semaphore idea in computer science comes from there; you can have at most N threads (boats) entering the critical sections (harbour) at any time, when the critical section (harbour) is full, the semaphore blocks the others until a thread (boat) leaves the critical section (harbour).

Even if it is an old concept, semaphore are still widely used. Interestingly, mutex locks can be represented by semaphores, with resources=1.



We already saw different tools we can use to solve the synchronisation problem but there are more!

You can have a look at “Monitors”, which is the concept behind the synchronized keyword in Java, if you want to understand how it works under the hood. The reference book or a Google search will lead you to a lot of resources here 😇.

There are also other ways to solve the mutual exclusion problem, OCaml provides Atomic data structure, on which you can perform *atomic* operations. RTM: https://v2.ocaml.org/api/Atomic.html

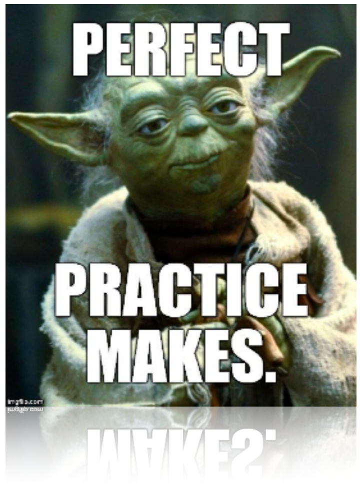
… or you have “read-write” locks. The idea is that multiple readers can access the data at the same time (so there is no critical section when read-only), but only one writer can access the data when it is modifying it. Example in Java: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html

Thread barriers: the idea is to put checkpoints which all threads have to reach together before going further. It is notably useful to save intermediary results. Again, example in Java: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html .

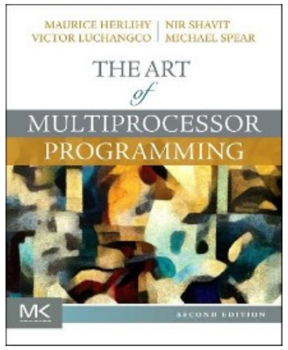
Take aways - did I hold my promise?

- ✔ There are multiple ways to avoid deadlocks
- ✔ There are multiple properties to satisfy in multi-threaded programs
- ✔ There exists multiple forms of locks
- ✔ Locks are not trivial to use… Even experts failed 😱

Exercises



Resources



Chapter 2

The Ptolemy project

E. Lee, "The Problem with Threads," Computer, vol. 39, pp. 33–42, Jun. 2006, doi: 10.1109/MC.2006.180.

P. V. Roy, "Programming Paradigms for Dummies: What Every Programmer Should Know," p. 39.