Concurrency

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

Concurrent vs parallel

- **Concurrency**
  - Do many unrelated things “at once”
  - Goals are expressiveness, responsiveness, and multitasking
- **Parallelism**
  - Get a faster answer with multiple CPUs

Concurrent vs parallel

- **Concurrency**
  - Do many unrelated things “at once”
  - Goals are expressiveness, responsiveness, and multitasking
- **Parallelism**
  - Get a faster answer with multiple CPUs

Here we will focus on the concurrency part

Threads

Thread

Threads are sequential computations that share memory.

Threads of control (aka lightweight process) execute concurrently in the same memory space. Threads communicate by in-place modifications of shared data structures, or by sending and receiving data on communication channels.

Threads

Thread

Threads are sequential computations that share memory.

Threads of control (aka lightweight process) execute concurrently in the same memory space. Threads communicate by in-place modifications of shared data structures, or by sending and receiving data on communication channels.

Two kinds of threads

1. *Native threads (a.k.a. OS Threads)*. They are directly handled by the OS.
   - Compatible with multiprocessors and low level processor capabilities
   - Better handling of input/output.
   - Compatible with native code.
2. *Green threads (a.k.a. light threads or user space threads)*. They are handled by the virtual machine.
   - More lightweight: context switch is much faster, much more threads can coexist.
   - They are portable but must be executed in the VM.
   - Input/outputs must be asynchronous since a blocking system call blocks all the threads within the process.

Which threads for whom

Green threads To be used if:

- Don’t want to wait for user input or blocking operations
- Need a lot of threads and need to switch from one to another rapidly
- Don’t care about about using multiple CPUs, since "the machine spends most of its time waiting on the user anyway".
- Typical usage: a web server.

Which threads for whom

Green threads To be used if:

- Don’t want to wait for user input or blocking operations
- Need a lot of threads and need to switch from one to another rapidly
- Don’t care about about using multiple CPUs, since "the machine spends most of its time waiting on the user anyway".
- Typical usage: a web server.

Native threads To be used if:

- Don’t want to wait for long running computations
- Either long running computation must advance “at the same time” or, better, run in parallel on multiple processors and actually finish faster
- Typical usage: heavy computations

Haskell mixed solution

Native threads: *1:1* Threre is a one-to-one correspondence between the application-level threads and the kernel threads

Green threads: *N:1* the program threads are managed in the user space. The kernel is not aware of them, so all application-level threads are mapped to a single kernel thread.

Haskell mixed solution

Native threads: *1:1* Threre is a one-to-one correspondence between the application-level threads and the kernel threads

Green threads: *N:1* the program threads are managed in the user space. The kernel is not aware of them, so all application-level threads are mapped to a single kernel thread.

Haskell and Erlang solution:

Hybrid threads: *N:M* (with $N \geqslant M$) Intermediate solution: spawn a whole bunch of lightweight green threads, but the interpreter schedules these threads onto a smaller number of native threads.

- ⊕ Can exploit multi-core, multi-processor architectures
- ⊕ Avoids to block all the threads on a blocking call
- ⊖ Hard to implement in particular the scheduling.
- ⊖ When using blocking system calls you actually need to notify somehow kernel to block only one green thread and not kernel one.

Multi-threading

Two kinds of multi-threading

1. *Preemptive threading:* A scheduler handles thread executions. Each thread is given a maximum time quantum and it is interrupted either because it finished its time slice or because it requests a “slow” operation (e.g., I/O, page-faulting memory access ...)
2. *Cooperative threading:* Each thread keeps control until either it explicitly handles it to another thread or it execute an asynchronous operation (e.g. I/O).

Multi-threading

Two kinds of multi-threading

1. *Preemptive threading:* A scheduler handles thread executions. Each thread is given a maximum time quantum and it is interrupted either because it finished its time slice or because it requests a “slow” operation (e.g., I/O, page-faulting memory access ...)
2. *Cooperative threading:* Each thread keeps control until either it explicitly handles it to another thread or it execute an asynchronous operation (e.g. I/O).

Possible combinations

1. Green threads are mostly preemptive, but several implementations of cooperative green threads are available (eg, the `Lwt` library in OCaml and the Coro module in Perl).
2. OS threads are nearly always preemptive since on a cooperative OS all applications must be programmed “fairly” and pass the hand to other applications from time to time

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

Shared memory model/ Process synchronization

Threads/processes are defined to achieve together a common goal therefore they do not live in isolation:

Shared memory model/ Process synchronization

Threads/processes are defined to achieve together a common goal therefore they do not live in isolation:

- To ensure that the goal is achieved threads/processes must *synchronize*.
- The purpose of *process synchronization* is to enforce constraints such as:
  - Serialization: this part of thread A must happen before this part of thread B.
  - Mutual exclusion: no two threads can execute this concurrently.

Shared memory model/ Process synchronization

Threads/processes are defined to achieve together a common goal therefore they do not live in isolation:

- To ensure that the goal is achieved threads/processes must *synchronize*.
- The purpose of *process synchronization* is to enforce constraints such as:
  - Serialization: this part of thread A must happen before this part of thread B.
  - Mutual exclusion: no two threads can execute this concurrently.

Several software tools are available to build synchronization policies for shared memory accesses:

- Semaphores
- Locks / Mutexes / Spinlocks
- Condition variables
- Barriers
- Monitors

Concurrent events

Two events are concurrent if we cannot tell by looking at the program which will happen first.

Concurrent events

Two events are concurrent if we cannot tell by looking at the program which will happen first.

Thread A

```
a1 x = 5
a2 print x
```

Thread B

```
b1 x = 7
```

Concurrent events

Two events are concurrent if we cannot tell by looking at the program which will happen first.

Thread A

```
a1 x = 5
a2 print x
```

Thread B

```
b1 x = 7
```

Possible outcomes:

- output 5 and final value for `x` = 7 (eg, a1→a2→b1)
- output 7 and final value for `x` = 7 (eg, a1→b1→a2)
- output 5 and final value for `x` = 5 (eg, b1→a1→a2)

Concurrent events

> Two events are concurrent if we cannot tell by looking at the program which will happen first.

Thread A

```
a1 x = 5
a2 print x
```

Thread B

```
b1 x = 7
```

Possible outcomes:

- output 5 and final value for `x` = 7 (eg, a1→a2→b1)
- output 7 and final value for `x` = 7 (eg, a1→b1→a2)
- output 5 and final value for `x` = 5 (eg, b1→a1→a2)

Thread A

```
x = x + 1
```

Thread A

```
x = x + 1
```

Concurrent events

> Two events are concurrent if we cannot tell by looking at the program which will happen first.

Thread A

```
a1 x = 5
a2 print x
```

Thread B

```
b1 x = 7
```

Possible outcomes:

- output 5 and final value for `x` = 7 (eg, a1→a2→b1)
- output 7 and final value for `x` = 7 (eg, a1→b1→a2)
- output 5 and final value for `x` = 5 (eg, b1→a1→a2)

Thread A

```
x = x + 1
```

Thread A

```
x = x + 1
```

If initially `x = 0` then both `x = 1` and `x = 2` are possible outcomes

Concurrent events

Two events are concurrent if we cannot tell by looking at the program which will happen first.

Thread A

```
a1 x = 5
a2 print x
```

Thread B

```
b1 x = 7
```

Possible outcomes:

- output 5 and final value for x = 7 (eg, a1→a2→b1)
- output 7 and final value for x = 7 (eg, a1→b1→a2)
- output 5 and final value for x = 5 (eg, b1→a1→a2)

Thread A

```
x = x + 1
```

Thread A

```
x = x + 1
```

If initially `x = 0` then both `x = 1` and `x = 2` are possible outcomes

Reason: The increment may be **not atomic**: $(t \leftarrow \texttt{read}\ \ x; x \leftarrow \texttt{read}\ \ t)$

For instance, in some assembler, `LDA $44; ADC #$01; STA $44` instead of `INC $44`

Model of execution

We must define the model of execution

- On some machines `x++` is atomic
- But let us not count on it: we do not want to write specialized code for each different hardware.
- Assume (rather pessimistically) that:
  - Result of concurrent writes is undefined.
  - Result of concurrent read-write is undefined.
  - Concurrent reads are ok.
  - Threads can be interrupted at any time (preemptive multi-threading).

Model of execution

We must define the model of execution

- On some machines `x++` is atomic
- But let us not count on it: we do not want to write specialized code for each different hardware.
- Assume (rather pessimistically) that:
    - Result of concurrent writes is undefined.
    - Result of concurrent read-write is undefined.
    - Concurrent reads are ok.
    - Threads can be interrupted at any time (preemptive multi-threading).

To solve synchronization problems let us first consider a very simple and universal software synchronization tool: *semaphores*

Semaphore

Semaphores are

- ⊕ *Simple*. The concept is just a little bit harder than that of a variable.
- ⊕ *Versatile*. You can pretty much solve all synchronization problems by semaphores.
- ⊖ *Error-prone.* They are so low level that they tend to be error-prone.

Semaphore

Semaphores are

- ⊕ *Simple*. The concept is just a little bit harder than that of a variable.
- ⊕ *Versatile*. You can pretty much solve all synchronization problems by semaphores.
- ⊖ *Error-prone.* They are so low level that they tend to be error-prone.

We start by them because:

- They are good for learning to think about synchronization

However:

- They are *not* the best choice for common use-cases (you'd better use specialized tools for specific problems, such as mutexes, conditionals, monitors, etc).

Semaphore

Semaphores are

- ⊕ *Simple*. The concept is just a little bit harder than that of a variable.
- ⊕ *Versatile*. You can pretty much solve all synchronization problems by semaphores.
- ⊖ *Error-prone.* They are so low level that they tend to be error-prone.

We start by them because:

- They are good for learning to think about synchronization

However:

- They are *not* the best choice for common use-cases (you'd better use specialized tools for specific problems, such as mutexes, conditionals, monitors, etc).

Definition (Dijkstra 1965)

A semaphore is an integer $\mathtt{s} \geqslant 0$ with two operations P and S:

- $P(s)$: if `s>0` then `s--` else the caller is suspended
- $S(s)$: if there is a suspended process, then resume it else `s++`

Semaphore object

In Python:

A semaphore is a class encapsulating an integer with two methods:

- `Semaphore(`*n*`)` initialize the counter to *n* (default is 1).
- `acquire()` if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release(). The order in which blocked threads are awakened is not specified.
- `release()` If another thread is waiting for it to become larger than zero again, wake up that thread otherwise increment the internal counter

Semaphore object

In Python:

A semaphore is a class encapsulating an integer with two methods:

- `Semaphore(`*n*`)` initialize the counter to *n* (default is 1).
- `acquire()` if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release(). The order in which blocked threads are awakened is not specified.
- `release()` If another thread is waiting for it to become larger than zero again, wake up that thread otherwise increment the internal counter

Variations that can be met in other languages:

- `wait()`, `signal()` (I will use this pair, because of the *signaling* pattern).
- negative counter to count the process awaiting at the semaphore.

Semaphore object

In Python:

A semaphore is a class encapsulating an integer with two methods:

- `Semaphore(`*n*`)` initialize the counter to *n* (default is 1).
- `acquire()` if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release(). The order in which blocked threads are awakened is not specified.
- `release()` If another thread is waiting for it to become larger than zero again, wake up that thread otherwise increment the internal counter

Variations that can be met in other languages:

- `wait()`, `signal()` (I will use this pair, because of the *signaling* pattern).
- negative counter to count the process awaiting at the semaphore.

Notice: no `get` method (to return the value of the counter). **Why?**

Semaphores to enforce Serialization

Problem:

Thread A

```
statement a1
```

Thread B

```
statement b1
```

How do we *enforce* the constraint: « a1 before b1 » ?

Semaphores to enforce Serialization

Problem:

Thread A

```
statement a1
```

Thread B

```
statement b1
```

How do we *enforce* the constraint: « a1 before b1 » ?

The signaling pattern:

```
sem = Semaphore(0)
```

Thread A

```
statement a1
sem.signal()
```

Thread B

```
sem.wait()
statement b1
```

You can think of `Semaphore(0)` as a locked lock.

Semaphores to enforce Mutual Exclusion

Problem:

Thread A

```
x = x + 1
```

Thread B

```
x = x + 1
```

Concurrent execution is non-deterministic

How can we *avoid* concurrent access?

Semaphores to enforce Mutual Exclusion

Problem:

Thread A

```
x = x + 1
```

Thread B

```
x = x + 1
```

Concurrent execution is non-deterministic

How can we *avoid* concurrent access?

Solution:

```
mutex = Semaphore(1)
```

Thread A

```
mutex.wait()
  x = x + 1
mutex.signal()
```

Thread B

```
mutex.wait()
  x = x + 1
mutex.signal()
```

Code between `wait` and `signal` is atomic.

More synch problems: readers and writers

Problem:

Threads are either writers or readers:

- Only *one* writer can *write* concurrently
- A reader cannot *read* concurrently with a writer
- Any number of readers can *read* concurrently

More synch problems: readers and writers

Problem:

Threads are either writers or readers:

- Only *one* writer can *write* concurrently
- A reader cannot *read* concurrently with a writer
- Any number of readers can *read* concurrently

Solution:

```
readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

Writer threads

```
roomEmpty.wait()
   critical section for writers
roomEmpty.signal()
```

Reader threads

```
mutex.wait()
  readers += 1
  if readers == 1:
     roomEmpty.wait()  # first in lock
mutex.signal()
     critical section for readers
mutex.wait()
  readers -= 1
  if readers == 0:
     roomEmpty.signal() # last out unlk
mutex.signal()
```

Readers and writers

Let us look for some common patterns

- The scoreboard pattern (readers)
  - Check in
  - Update state on the scoreboard (number of readers)
  - make some conditional behavior
  - check out
- The turnstile pattern (writer)
  - Threads go through the turnstile serially
  - One blocks, all wait
  - It passes, it unblocks
  - Other threads (ie, the readers) can lock the turnstile

Readers and writers

Readers while checking in/out implement the *lightswitch* pattern:

- The first person that enters the room switch the light on (acquires the lock)
- The last person that exits the room switch the light off (releases the lock)

Readers and writers

Readers while checking in/out implement the *lightswitch* pattern:

- The first person that enters the room switch the light on (acquires the lock)
- The last person that exits the room switch the light off (releases the lock)

Implementation:

```
class Lightswitch:
  def __init__(self):
      self.counter = 0
      self.mutex = Semaphore(1)

  def lock(self, semaphore):
      self.mutex.wait()
          self.counter += 1
          if self.counter == 1:
              semaphore.wait()
      self.mutex.signal()

  def unlock(self, semaphore):
      self.mutex.wait()
          self.counter -= 1
          if self.counter == 0:
              semaphore.signal()
      self.mutex.signal()
```

Before:

```
readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

Writer threads

```
roomEmpty.wait()
   critical section for writers
roomEmpty.signal()
```

Reader threads

```
mutex.wait()
  readers += 1
  if readers == 1:
     roomEmpty.wait()  # first in lock
mutex.signal()
     critical section for readers
mutex.wait()
  readers -= 1
  if readers == 0:
     roomEmpty.signal() # last out unlk
mutex.signal()
```

Before:

```
readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

Writer threads

```
roomEmpty.wait()
   critical section for writers
roomEmpty.signal()
```

Reader threads

```
mutex.wait()
  readers += 1
  if readers == 1:
     roomEmpty.wait()  # first in lock
mutex.signal()
     critical section for readers
mutex.wait()
  readers -= 1
  if readers == 0:
     roomEmpty.signal() # last out unlk
mutex.signal()
```

After:

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Writer threads

```
roomEmpty.wait()
   critical section for writers
roomEmpty.signal()
```

Reader threads

```
readLightswitch.lock(roomEmpty)
     critical section for readers
readLightswitch.unlock(roomEmpty)
```

Programming golden rules

When programming becomes too complex then:

1. Abstract common patterns
2. Split it in more elementary problems

The previous case was an example of abstraction. Next we are going to see an example of modularization, where we combine our elementary patterns to solve more complex problems

The unisex bathroom problem

A women at Xerox was working in a cubicle in the basement, and the nearest women's bathroom was two floors up. She proposed to the Uberboss that they convert the men's bathroom on her floor to a unisex bathroom.
The Uberboss agreed, provided that the following synchronization constraints can be maintained:

1. There cannot be men and women in the bathroom at the same time.
2. There should never be more than three employees squandering company time in the bathroom.

You may assume that the bathroom is equipped with all the semaphores you need.

The unisex bathroom problem

Solution hint:

```
    empty = Semaphore(1)
    maleSwitch = Lightswitch()
    femaleSwitch = Lightswitch()
    maleMultiplex = Semaphore(3)
    femaleMultiplex = Semaphore(3)
```

- `empty` is 1 if the room is empty and 0 otherwise.
- `maleSwitch` allows men to bar women from the room. When the first male enters, the lightswitch locks `empty`, barring women; When the last male exits, it unlocks `empty`, allowing women to enter. Women do likewise using `femaleSwitch`.
- `maleMultiplex` and `femaleMultiplex` ensure that there are no more than three men and three women in the system at a time (they are semaphores used as locks).

The unisex bathroom problem

A solution:

Female Threads

```
femaleSwitch.lock(empty)
    femaleMultiplex.wait()
        bathroom code here
    femaleMultiplex.signal()
femaleSwitch.unlock(empty)
```

Male Threads

```
maleSwitch.lock(empty)
    maleMultiplex.wait()
        bathroom code here
    maleMultiplex.signal()
maleSwitch.unlock(empty)
```

The unisex bathroom problem

A solution:

Female Threads

```
femaleSwitch.lock(empty)
    femaleMultiplex.wait()
        bathroom code here
    femaleMultiplex.signal()
femaleSwitch.unlock(empty)
```

Male Threads

```
maleSwitch.lock(empty)
    maleMultiplex.wait()
        bathroom code here
    maleMultiplex.signal()
maleSwitch.unlock(empty)
```

Any problem with this solution?

The unisex bathroom problem

A solution:

Female Threads

```
femaleSwitch.lock(empty)
    femaleMultiplex.wait()
        bathroom code here
    femaleMultiplex.signal()
femaleSwitch.unlock(empty)
```

Male Threads

```
maleSwitch.lock(empty)
    maleMultiplex.wait()
        bathroom code here
    maleMultiplex.signal()
maleSwitch.unlock(empty)
```

Any problem with this solution?

> *This solution allows starvation. A long line of women can arrive and enter while there is a man waiting, and vice versa.*

Find a solution

Hint: Use a turnstile to access to the lightswitches: when a man arrives and the bathroom is already occupied by women, block turnstile so that more women cannot check the light and enter.

The no-starve unisex bathroom problem

```
turnstile = Semaphore(1)
empty = Semaphore(1)
maleSwitch = Lightswitch()
femaleSwitch = Lightswitch()
maleMultiplex = Semaphore(3)
femaleMultiplex = Semaphore(3)
```

Female Threads

```
turnstile.wait()
    femaleSwitch.lock(empty)
turnstile.signal()

    femaleMultiplex.wait()
        bathroom code here
    femaleMultiplex.signal()

femaleSwitch.unlock (empty)
```

Male Threads

```
turnstile.wait()
    maleSwitch.lock(empty)
turnstile.signal()

    maleMultiplex.wait()
        bathroom code here
    maleMultiplex.signal()

maleSwitch.unlock (empty)
```

The no-starve unisex bathroom problem

```
   turnstile = Semaphore(1)
   empty = Semaphore(1)
   maleSwitch = Lightswitch()
   femaleSwitch = Lightswitch()
   maleMultiplex = Semaphore(3)
   femaleMultiplex = Semaphore(3)
```

Female Threads

```
turnstile.wait()
    femaleSwitch.lock(empty)
turnstile.signal()

    femaleMultiplex.wait()
        bathroom code here
    femaleMultiplex.signal()

femaleSwitch.unlock (empty)
```

Male Threads

```
turnstile.wait()
    maleSwitch.lock(empty)
turnstile.signal()

    maleMultiplex.wait()
        bathroom code here
    maleMultiplex.signal()

maleSwitch.unlock (empty)
```

Actually we could have used the same multiplex for both females and males.

Summary so far

- Solution composed of *patterns*
- Patterns can be encapsulated as objects or modules
- Unisex bathroom problem is a good example of use of both *abstraction* and *modularity* (lightswitches and turnstiles)

Summary so far

- Solution composed of *patterns*
- Patterns can be encapsulated as objects or modules
- Unisex bathroom problem is a good example of use of both *abstraction* and *modularity* (lightswitches and turnstiles)

- Unfortunately, patterns *often* interact and interfere. Hard to be confident of solutions (formal verification and test are not production-ready yet).
- Especially true for semaphores which are very low level:
  - ⊕ They can be used to implement more complex synchronization patterns.
  - ⊖ This makes interference much more likely.

Summary so far

- Solution composed of *patterns*
- Patterns can be encapsulated as objects or modules
- Unisex bathroom problem is a good example of use of both *abstraction* and *modularity* (lightswitches and turnstiles)

- Unfortunately, patterns *often* interact and interfere. Hard to be confident of solutions (formal verification and test are not production-ready yet).
- Especially true for semaphores which are very low level:
  - ⊕ They can be used to implement more complex synchronization patterns.
  - ⊖ This makes interference much more likely.

Before discussing more general problems of shared memory synchronization, let us introduced some higher-level and more specialized tools that, being more specific, make interference less likely.

- Locks
- Conditional Variables
- Monitors

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

Locks

Locks are like those on a room door:

- *Lock acquisition:* A person enters the room and locks the door. Nobody else can enter.
- *Lock release:* The person in the room exits unlocking the door.

Persons are *threads*, rooms are *critical regions*.

Locks

Locks are like those on a room door:

- *Lock acquisition:* A person enters the room and locks the door. Nobody else can enter.
- *Lock release:* The person in the room exits unlocking the door.

Persons are *threads*, rooms are *critical regions*.
A person that finds a door locked can either wait or come later (somebody lets it know that the room is available).

Locks

Locks are like those on a room door:

- *Lock acquisition:* A person enters the room and locks the door. Nobody else can enter.
- *Lock release:* The person in the room exits unlocking the door.

Persons are *threads*, rooms are *critical regions*.
A person that finds a door locked can either wait or come later (somebody lets it know that the room is available).

Similarly there are two possibilities for a thread that failed to acquire a lock:

1. It keeps trying. This kind of lock is a *spinlock*. Meaningful only on multi-processors, they are common in High performance computing (where most of the time each thread is scheduled on its own processor anyway).
2. It is suspended until somebody signals it that the lock is available. The only meaningful lock for uniprocessor. This kind of lock is also called *mutex* (but often *mutex* is used as a synonym for lock).

Difference between a mutex and a binary semaphore

A mutex is *different* from a binary semaphore (ie, a semaphore initialized to 1), since it combines the notion of *exclusivity of manipulation* (as for semaphores) with others extra features such as *exclusivity of possession* (only the process which has taken a mutex can free it) or *priority inversion protection*. The differences between mutexes and semaphores are operating system/language dependent, though mutexes are implemented by specialized, faster routines.

Difference between a mutex and a binary semaphore

A mutex is *different* from a binary semaphore (ie, a semaphore initialized to 1), since it combines the notion of *exclusivity of manipulation* (as for semaphores) with others extra features such as *exclusivity of possession* (only the process which has taken a mutex can free it) or *priority inversion protection*. The differences between mutexes and semaphores are operating system/language dependent, though mutexes are implemented by specialized, faster routines.

Example

What follows can be done with semaphore s but not with a mutex, since B unlocks a lock of A (cf. the signaling pattern):

```
Thread A                     Thread B
   ⋮                            ⋮
some stuff                   some stuff
   ⋮                            ⋮
wait(s)                      signal(s) (* A can continue *)
   ⋮                            ⋮
some other stuff
```

Mutex

Since semaphores are *for what concerns mutual exclusion* a simplified version of mutexes, it is clear that mutexes have operations very similar to the former:

- A `init` or `create` operation.
- A `wait` or `lock` operation that tries to acquire the lock and suspends the thread if it is not available
- A `signal` or `unlock` operation that releases the lock and possibly awakes a thread waiting for the lock
- Sometimes a `trylock`, that is, a non blocking locking operation that returns an error or false if the lock is not available.

Mutex

Since semaphores are *for what concerns mutual exclusion* a simplified version of mutexes, it is clear that mutexes have operations very similar to the former:

- A `init` or `create` operation.
- A `wait` or `lock` operation that tries to acquire the lock and suspends the thread if it is not available
- A `signal` or `unlock` operation that releases the lock and possibly awakes a thread waiting for the lock
- Sometimes a `trylock`, that is, a non blocking locking operation that returns an error or false if the lock is not available.

A mutex is *reentrant* if the same thread can acquire the lock multiple times. However, the lock must be released the same number of times or else other threads will be unable to acquire the lock.

Nota Bene: A reentrant mutex has some similarities to a counting semaphore: the number of lock acquisitions is the counter, but only one thread can successfully perform multiple locks (exclusivity of possession).

Implementation of locks

Some examples of lock implementations:

- Using hardware special instructions like `test-and-set` or `compare-and-swap`
- Peterson algorithm (spinlock, deadlock free) in Python:

```
flag=[0,0]
turn = 0          # initially the priority is for thread 0
```

Thread 0

```
 flag[0] = 1
 turn = 1
 while flag[1] and turn : pass
     critical section
 flag[0] = 0
```

Thread 1

```
 flag[1] = 1
 turn = 0
 while flag[0] and not turn : pass
     critical section
 flag[1] = 0
```

`flag[i] == 1`: Thread `i` wants to enter;

`turn == i` it is the turn of Thread `i` to enter, if it wishes.

- Lamport's bakery algorithm (deadlock and starvation free)
  Every threads modifies only its own variables and accesses to other variables only by reading.

Condition Variables

Locks provides a passive form of synchronization: they allow waiting for shared data to be free, but do not allow waiting for the data to have a particular state. *Condition variables* are the solution to this problem.

Definition

A *condition variable* is an atomic waiting and signaling mechanism which allows a process or thread to atomically stop execution and release a lock until a signal is received.

Rationale

It allows a thread to sleep inside a critical region without risk of deadlock.

Condition Variables

Locks provides a passive form of synchronization: they allow waiting for shared data to be free, but do not allow waiting for the data to have a particular state. *Condition variables* are the solution to this problem.

Definition

A *condition variable* is an atomic waiting and signaling mechanism which allows a process or thread to atomically stop execution and release a lock until a signal is received.

Rationale

It allows a thread to sleep inside a critical region without risk of deadlock.

Three main operations:

- `wait(`*`lock`*`)` releases the lock, gives up the CPU until signaled and then re-acquire the lock.
- `signal()` wakes up a thread waiting on the condition variable, if any.
- `broadcast()` wakes up all threads waiting on the condition.

Condition Variables

Note: The term "condition variable" is misleading: it does not rely on a variable but rather on signaling at the system level. The term comes from the fact that condition variables are most often used to notify changes in the state of shared variables, such as in

- Notify a reader thread that a writer thread has filled its data set.
- Notify consumer processes that a producer thread has updated a shared data set.

Condition Variables

Note: The term "condition variable" is misleading: it does not rely on a variable but rather on signaling at the system level. The term comes from the fact that condition variables are most often used to notify changes in the state of shared variables, such as in

- Notify a reader thread that a writer thread has filled its data set.
- Notify consumer processes that a producer thread has updated a shared data set.

Semaphores and Condition variables

- semaphores and condition variables both use `wait` and `signal` as valid operations,
- the purpose of both is somewhat similar, but
- *they are different:*
  - With a semaphore the signal operation increments the value of the semaphore even if there is no blocked process. The signal is remembered.
  - If there are no processes blocked on the condition variable then the signal function does nothing. The signal is not remembered.
  - With a semaphore you must be careful about deadlocks.

Monitors

Motivation:

- Semaphores are incredibly versatile.
- The problem with them is that they are *dual purpose*: they can be used for both *mutual exclusion* and *scheduling* constraints. This makes the code hard to read, and hard to get right.
- In the previous slides we have introduced two separate constructs for each purpose: mutexes and conditional variables.
- Monitors groups them together (keeping each disctinct from the other) to protect some shared data:

Monitors

Motivation:

- Semaphores are incredibly versatile.
- The problem with them is that they are *dual purpose*: they can be used for both *mutual exclusion* and *scheduling* constraints. This makes the code hard to read, and hard to get right.
- In the previous slides we have introduced two separate constructs for each purpose: mutexes and conditional variables.
- Monitors groups them together (keeping each disctinct from the other) to protect some shared data:

Definition (Monitor)

a lock and zero or more condition variables for managing concurrent access to shared data by defining some given operations.

Monitors

Example: a synchronized queue

In pseudo-code:

```
monitor SynchQueue {
  lock = Lock.create
  condition = Condition.create

  addToQueue(item) {
    lock.acquire();
    put item on queue;
    condition.signal();
    lock.release();
  }

  removeFromQueue() {
    lock.acquire();
    while nothing on queue do
       condition.wait(lock)    // release lock; go to
    done                       // sleep; re-acquire lock
    remove item from queue;
    lock.release();
    return item
  }
}
```

Different kinds of Monitors

Need to be careful about the precise definition of signal and wait:

Mesa-style: (Nachos, most real operating systems)

- Signaler keeps lock, processor
- Waiter simply put on ready queue, with no special priority. (in other words, waiter may have to wait for lock)

Hoare-style: (most textbooks)

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Waiter gives lock, processor back to signaler when it exits critical section or if it waits again.

Above code for synchronized queuing happens to work with either style, but for many programs it matters which one you are using. With Hoare-style, can change "while" in removeFromQueue to an "if", because the waiter only gets woken up if item is on the list. With Mesa-style monitors, waiter may need to wait again after being woken up, because some other thread may have acquired the lock, and removed the item, before the original waiting thread gets to the front of the ready queue.

Different kinds of Monitors

Need to be careful about the precise definition of signal and wait:

Mesa-style: (Nachos, most real operating systems)

- Signaler keeps lock, processor
- Waiter simply put on ready queue, with no special priority. (in other words, waiter may have to wait for lock)

Hoare-style: (most textbooks)

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Waiter gives lock, processor back to signaler when it exits critical section or if it waits again.

Above code for synchronized queuing happens to work with either style, but for many programs it matters which one you are using. With Hoare-style, can change "while" in removeFromQueue to an "if", because the waiter only gets woken up if item is on the list. With Mesa-style monitors, waiter may need to wait again after being woken up, because some other thread may have acquired the lock, and removed the item, before the original waiting thread gets to the front of the ready queue.

Preemptive threads in OCaml

Four main modules:

- Module `Thread`: lightweight threads (abstract type `Thread.t`)
- Module `Mutex`: locks for mutual exclusion (abstract type `Mutex.t`)
- Module `Condition`: condition variables to synchronize between threads (abstract type `Condition.t`)
- Module `Event`: first-class synchronous channels (abstract types `'a Event.channel` and `'a Event.event`)

Preemptive threads in OCaml

Four main modules:

- Module `Thread`: lightweight threads (abstract type `Thread.t`)
- Module `Mutex`: locks for mutual exclusion (abstract type `Mutex.t`)
- Module `Condition`: condition variables to synchronize between threads (abstract type `Condition.t`)
- Module `Event`: first-class synchronous channels (abstract types `'a Event.channel` and `'a Event.event`)

Two implementations:

- System threads. Uses OS-provided threads: POSIX threads for Unix, and Win32 threads for Windows. Supports both bytecode and native-code.
- Green threads. Time-sharing and context switching at the level of the bytecode interpreter. Works on OS without multi-threading but cannot be used with native-code programs.

Preemptive threads in OCaml

Four main modules:

- Module `Thread`: lightweight threads (abstract type `Thread.t`)
- Module `Mutex`: locks for mutual exclusion (abstract type `Mutex.t`)
- Module `Condition`: condition variables to synchronize between threads (abstract type `Condition.t`)
- Module `Event`: first-class synchronous channels (abstract types `'a Event.channel` and `'a Event.event`)

Two implementations:

- System threads. Uses OS-provided threads: POSIX threads for Unix, and Win32 threads for Windows. Supports both bytecode and native-code.
- Green threads. Time-sharing and context switching at the level of the bytecode interpreter. Works on OS without multi-threading but cannot be used with native-code programs.

Nota Bene: Always work on a single processor (because of OCaml's GC). No advantage from multi-processors (apart from explicit execution of C code or system calls): threads are just for structuring purposes.

Module Thread

- `create : ('a -> 'b) -> 'a -> Thread.t`

  `Thread.create f e` creates a new thread of control, in which the function application `f(e)` is executed concurrently with the other threads of the program.

- `kill : Thread.t -> unit`

  `kill p` terminates prematurely the thread `p`

- `join : Thread.t -> unit`

  `join p` suspends the execution of the calling thread until the termination of `p`

- `delay : float -> unit`

  `delay d` suspends the execution of the calling thread for `d` seconds.

```
# let f () = for i=0 to 10 do Printf.printf "(%d)" i done;;
val f : unit -> unit = <fun>
#  Printf.printf "begin ";
   Thread.join (Thread.create f ());
   Printf.printf " end";;
begin (0)(1)(2)(3)(4)(5)(6)(7)(8)(9)(10) end- : unit = ()
```

Module `Mutex`

- `create : unit -> Mutex.t` Return a new mutex.
- `lock : Mutex.t -> unit` Lock the given mutex.
- `try_lock : Mutex.t -> bool` Non blocking lock.
- `unlock : Mutex.t -> unit` Unlock the given mutex.

Module `Mutex`

- `create : unit -> Mutex.t` Return a new mutex.
- `lock : Mutex.t -> unit` Lock the given mutex.
- `try_lock : Mutex.t -> bool` Non blocking lock.
- `unlock : Mutex.t -> unit` Unlock the given mutex.

Dining philosophers

Five philosophers sitting at a table doing one of two things: eating or meditate. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of rice in the center. A chopstick is placed in between each pair of adjacent philosophers and to eat he needs two chopsticks. Each philosopher can only use the chopstick on his immediate left and immediate right.

```
# let b =
  let b0 = Array.create 5 (Mutex.create()) in
   for i=1 to 4 do b0.(i) <- Mutex.create() done;
   b0 ;;
val b : Mutex.t array = [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]
```

Dining philosophers

```
# let meditation = Thread.delay
  and eating = Thread.delay ;;
```

```
let philosopher i =
 let ii = (i+1) mod 5
 in while true do
     meditation 3. ;
     Mutex.lock b.(i);
     Printf.printf "Philo (%d) takes his left-hand chopstick" i ;
     Printf.printf " and meditates a little while more\n";
     meditation 0.2;
     Mutex.lock b.(ii);
     Printf.printf "Philo (%d) takes his right-hand chopstick\n" i;
     eating 0.5;
     Mutex.unlock b.(i);
     Printf.printf "Philo (%d) puts down his left-hand chopstick" i;
     Printf.printf " and goes back to meditating\n";
     meditation 0.15;
     Mutex.unlock b.(ii);
     Printf.printf "Philo (%d) puts down his right-hand chopstick\n" i
    done ;;
```

Dining philosophers

We can test this little program by executing:

```
for i=0 to 4 do ignore (Thread.create philosopher i) done ;
while true do Thread.delay 5. done ;;
```

Problems:

- Deadlock: all philosophers can take their left-hand chopstick, so the program is stuck.
- Starvation: To avoid deadlock, the philosophers can put down a chopstick if they do not manage to take the second one. This is highly courteous, but still allows two philosophers to gang up against a third to stop him from eating.

Exercise

Think about solutions to avoid deadlock and starvation

Module `Condition`

- `create : unit -> Condition.t` returns a new condition variable.
- `wait : Condition.t -> Mutex.t -> unit`

  `wait c m` atomically unlocks the mutex `m` and suspends the calling process on the condition variable `c`. The process will restart after the condition variable `c` has been signaled. The mutex `m` is locked again before `wait` returns.
- `signal : Condition.t -> unit`

  `signal c` restarts one of the processes waiting on the condition variable `c`.
- `broadcast : Condition.t -> unit`

  `broadcast c` restarts all processes waiting on the condition variable `c`.

Module `Condition`

- `create : unit -> Condition.t` returns a new condition variable.
- `wait : Condition.t -> Mutex.t -> unit`

  `wait c m` atomically unlocks the mutex `m` and suspends the calling process on the condition variable `c`. The process will restart after the condition variable `c` has been signaled. The mutex `m` is locked again before `wait` returns.
- `signal : Condition.t -> unit`

  `signal c` restarts one of the processes waiting on the condition variable `c`.
- `broadcast : Condition.t -> unit`

  `broadcast c` restarts all processes waiting on the condition variable `c`.

Typical usage pattern:

```
Mutex.lock m;
   while (* some predicate P over D is not satisfied *) do
     Condition.wait c m
   done;
   (* Modify D *)
   if (*  the predicate P over D is now satisfied *) then Condition.signa
Mutex.unlock m
```

Module Condition

- `create : unit -> Condition.t` returns a new condition variable.
- `wait : Condition.t -> Mutex.t -> unit`

  `wait c m` atomically unlocks the mutex `m` and suspends the calling process on the condition variable `c`. The process will restart after the condition variable `c` has been signaled. The mutex `m` is locked again before `wait` returns.
- `signal : Condition.t -> unit`

  `signal c` restarts one of the processes waiting on the condition variable `c`.
- `broadcast : Condition.t -> unit`

  `broadcast c` restarts all processes waiting on the condition variable `c`.

Typical usage pattern:

```
Mutex.lock m;
   while (* some predicate P over D is not satisfied *) do
     Condition.wait c m                    // We put the wait in a while loop: why?
   done;
   (* Modify D *)
   if (*  the predicate P over D is now satisfied *) then Condition.signa
Mutex.unlock m
```

Example: a Monitor

```
module SynchQueue = struct
  type ’a t =
    { queue : ’a Queue.t; lock : Mutex.t; non_empty : Condition.t }

  let create () = {
    queue = Queue.create ();
    lock = Mutex.create ();
    non_empty = Condition.create ()
  }

  let add e q =
    Mutex.lock q.lock;
    if Queue.length q.queue = 0 then Condition.broadcast q.non_empty;
    Queue.add e q.queue;
    Mutex.unlock q.lock

  let remove q =
    Mutex.lock q.lock;
    while Queue.length q.queue = 0 do
      Condition.wait q.non_empty q.lock done;
    let x = Queue.take q.queue in
    Mutex.unlock q.lock; x
end
```

Monitors

OCaml does not provide explicit constructions for monitors. They must be implemented by using mutexes and condition variables. Other languages provides monitors instead, for instance Java.

Monitors in Java:

- In Java a monitor is any object in which at least one method is declared `synchronized`
- When a thread is executing a synchronized method of some object, then the other threads are blocked if they call any synchronized method of that object.

Monitors

OCaml does not provide explicit constructions for monitors. They must be implemented by using mutexes and condition variables. Other languages provides monitors instead, for instance Java.

Monitors in Java:

- In Java a monitor is any object in which at least one method is declared `synchronized`
- When a thread is executing a synchronized method of some object, then the other threads are blocked if they call any synchronized method of that object.

```
class Account{
  float balance;
  synchronized void deposit(float amt) {
     balance += amt;
  }
  synchronized void withdraw(float amt) {
    if (balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
}
```

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

What’s wrong with locking

- **Locking has many pitfalls for the inexperienced programmer**

  Priority inversion: a lower priority thread is preempted while holding a lock needed by higher-priority threads.

  Convoying: A thread holding a lock is descheduled due to a time-slice interrupt or page fault causing other threads requiring that lock to queue up. When rescheduled it may take some time to drain the queue. The overhead of repeated context switches and underutilization of scheduling quanta degrade overall performance.

  Deadlock: threads that lock the same objects in different order. Deadlock avoidance is difficult if many objects are accessed at the same time and they are not statically known.

  Debugging: Lock related problems are difficult to debug (since, being time-related, they are difficult to reproduce).

  Fault-tolerance If a thread (or process) is killed or fails while holding a lock, what does happen? (cf. `Thread.delete`)

- **Programming is not easy with locks and requires difficult decisions:**
    - Taking too few locks — leads to race conditions.
    - Taking too many locks — inhibits concurrency
    - Locking at too coarse a level — inhibits concurrency
    - Taking locks in the wrong order — leads to deadlock
    - Error recovery is hard (eg, how to handle failure of threads holding locks?)
- **A major problem: Composition**
    - Lock-based programs do not compose: For example, consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item A from table t1, and insert it into table t2; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementer of the hash table anticipates this need, there is simply no way to satisfy this requirement.

Locks are non-compositional

Consider the previous (correct) Java bank Account class:

```
class Account{
  float balance;

  synchronized void deposit(float amt) {
    balance += amt;
  }

  synchronized void withdraw(float amt) {
    if (balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
}
```

Now suppose we want to add the ability to transfer funds from one account to another.

Locks are non-compositional

Simply calling withdraw and deposit to implement transfer causes a race condition:

```
class Account{
  float balance;
  ...
  void badTransfer(Acct other, float amt) {
  other.withdraw(amt);
  // here checkBalances sees bad total balance
  this.deposit(amt);
  }
}

class Bank {
  Account[] accounts;
  float global_balance;

  checkBalances () {
    return (sum(Accounts) == global_balance);
  }
}
```

Locks are non-compositional

Synchronizing transfer can cause deadlock:

```
class Account{
  float balance;

  synchronized void deposit(float amt) {
    balance += amt;
  }

  synchronized void withdraw(float amt) {
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }

  synchronized void badTrans(Acct other, float amt) {
    // can deadlock with parallel reverse-transfer
    this.deposit(amt);
    other.withdraw(amt);
  }
}
```

Concurrency without locks

We need to synchronize threads without resorting to locks

Concurrency without locks

We need to synchronize threads without resorting to locks

1. *Cooperative threading*

   The threads themselves relinquish control once they are at a stopping point.

Concurrency without locks

We need to synchronize threads without resorting to locks

1. *Cooperative threading*

   The threads themselves relinquish control once they are at a stopping point.

2. *Channeled communication*

   The threads do not share memory. All data is exchanged by explicit communications that take place on channels.

Concurrency without locks

We need to synchronize threads without resorting to locks

1. *Cooperative threading*

   The threads themselves relinquish control once they are at a stopping point.

2. *Channeled communication*

   The threads do not share memory. All data is exchanged by explicit communications that take place on channels.

3. *Software transactional memory*

   Each thread declares the blocks that must be performed atomically. If the execution of an atomic block causes any conflict, the modifications are rolled back and the block is re-executed.

Concurrency without locks

We need to synchronize threads without resorting to locks

1. *Cooperative threading* → disallow concurrent access

   The threads themselves relinquish control once they are at a stopping point.

2. *Channeled communication* → disallow shared memory

   The threads do not share memory. All data is exchanged by explicit communications that take place on channels.

3. *Software transactional memory* → handle conflicts when happen

   Each thread declares the blocks that must be performed atomically. If the execution of an atomic block causes any conflict, the modifications are rolled back and the block is re-executed.

Concurrency without locks

1. *Cooperative threading:* The threads themselves relinquish control once they are at a stopping point.

   Pros: Programmer manage interleaving, no concurrent access happens

   Cons: The burden is on the programmer: the system may not be responsive (eg, Classic Mac OS 5.x to 9.x). Does not scale on multi-processors. Not always compositional.

2. *Channeled communication:* The threads do not share memory. All data is exchanged by explicit communications that take place on channels.

   Pros: Compositional. Easily scales to multi-processor and distributed programming (if asynchronous)

   Cons: Awkward when threads concurrently work on complex and large data-structures.

3. *Software transactional memory:* If the execution of an atomic block cause any conflict, modification are rolled back and the block re-executed.

   Pros: Very compositional. A no brainer for the programmer.

   Cons: Very new, poorly mastered. Feasibility depends on conflict likelihood.

Concurrency without locks

Besides the previous solution there is also a more drastic solution (not so general as the previous ones but composes with them):

- *Lock-free programming*

  > *Threads access to shared data without the use of synchronization primitives such as mutexes. The operations to access the data ensure the absence of conflicts.*

Idea: instead of giving operations for mutual exclusion of accesses, define the access operations so that they take into account concurrent accesses.

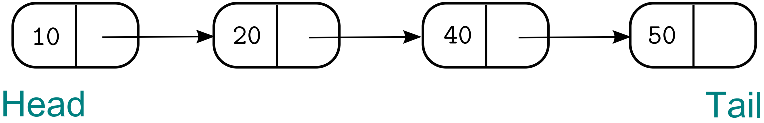
Pros: A no-brainer if the data structures available in your lock-free library fit your problem. It has the granularity precisely needed (if you work on a queue with locks, should you use a lock for the whole queue?)

Cons: requires to have specialized operations for each data structure. Not modular since composition may require using a different more complex data structure. Works with simple data structure but is hard to generalize to complex operations. Hard to implement in the absence of hardware support (e.g., `compare_and_swap`).

Lock-free programming: an example

Non blocking linked list:

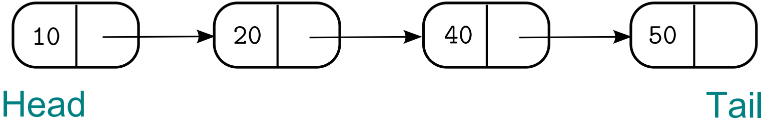
Insertion:



Lock-free programming: an example

Non blocking linked list:

Insertion:

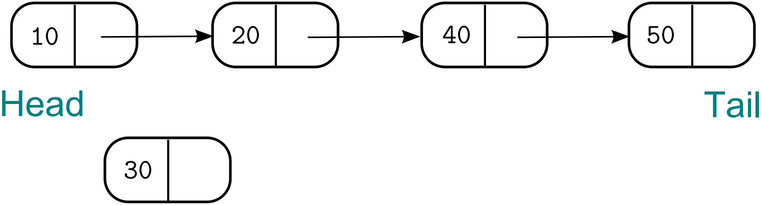


The linked list above is ordered

Lock-free programming: an example

Non blocking linked list:

Insertion:

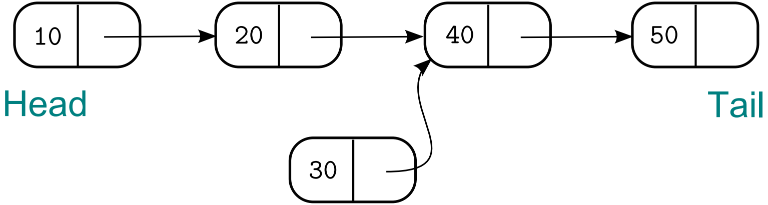


We want to insert a new element *new* at the right position

Lock-free programming: an example

Non blocking linked list:

Insertion:

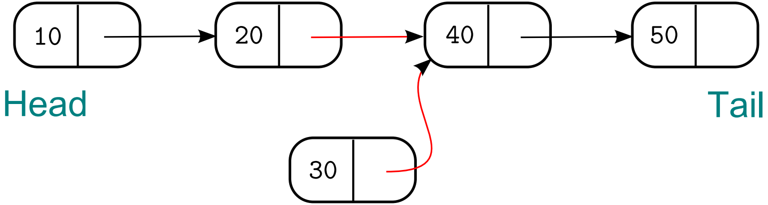


Find the right position *pos* and set to it the field *next* of the *new* element

Lock-free programming: an example

Non blocking linked list:

Insertion:

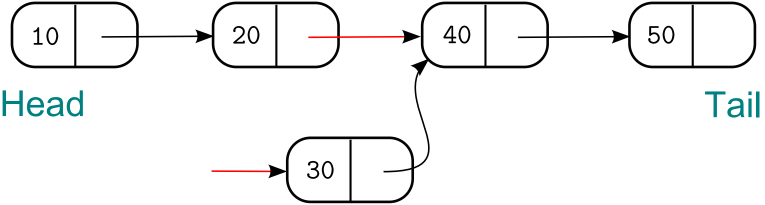


Compare and swap: compare the two links *pos* and *next*

Lock-free programming: an example

Non blocking linked list:

Insertion:



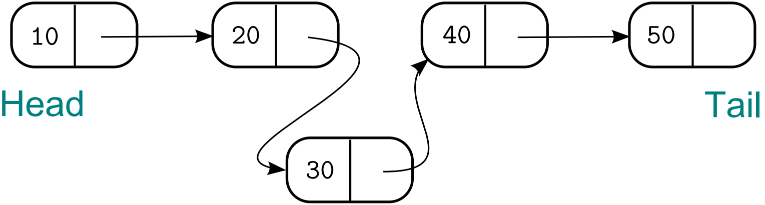
Compare and swap: compare the two links *pos* and *next*

- If they are the same then no conflicting operation has been performed, so atomically swap the successor of the second element with the pointer to the new element.

Lock-free programming: an example

Non blocking linked list:

Insertion:



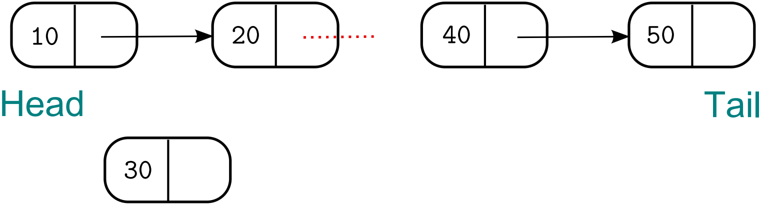
Compare and swap: compare the two links *pos* and *next*

- If they are the same then no conflicting operation has been performed, so atomically swap the successor of the second element with the pointer to the new element. `comp_and_swap(`*pos*`,` *next*`,` *new*`)`

Lock-free programming: an example

Non blocking linked list:

Insertion:

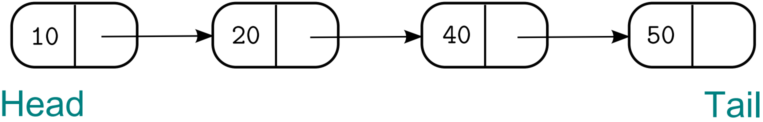


Compare and swap: compare the two links *pos* and *next*

- If they are the same then no conflicting operation has been performed, so atomically swap the successor of the second element with the pointer to the new element.
- Otherwise, a conflicting modification was performed: **retry the insert**.

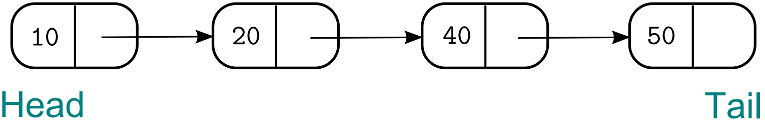
Lock-free programming: Non blocking linked list:

Deletion is more difficult:



Lock-free programming: Non blocking linked list:

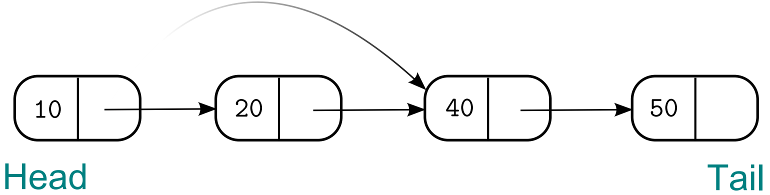
Deletion is more difficult:



Imagine we want to delete the first element. First we try a naive solution

Lock-free programming: Non blocking linked list:

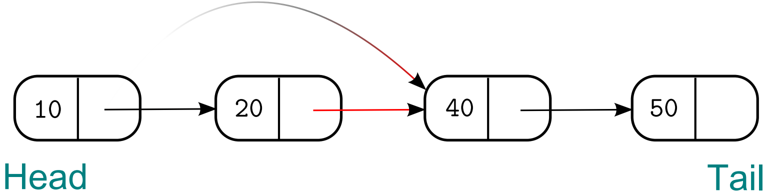
Deletion is more difficult:



Put a pointer to where the *next* field of the 10 element will point

Lock-free programming: Non blocking linked list:

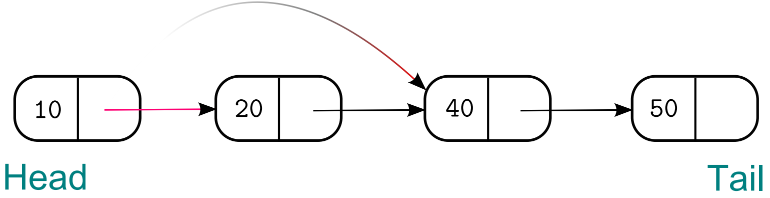
Deletion is more difficult:



Use compare and swap to check whether in the meanwhile any update was done between the 10 and 20 element:

Lock-free programming: Non blocking linked list:

Deletion is more difficult:

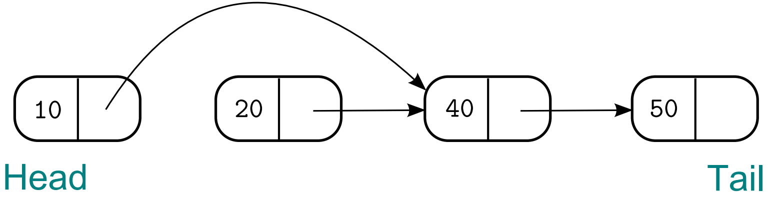


Use compare and swap to check whether in the meanwhile any update was done between the 10 and 20 element:

- **If the test suceeds then update the *next* field of the 10 element**

Lock-free programming: Non blocking linked list:

Deletion is more difficult:

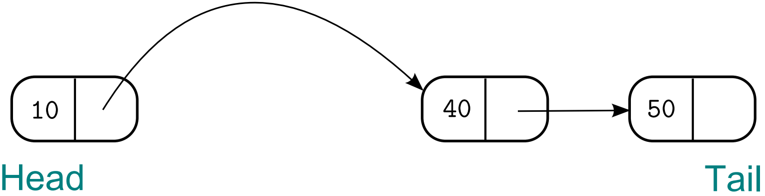


Use compare and swap to check whether in the meanwhile any update was done between the 10 and 20 element:

- **If the test suceeds then update the *next* field of the 10 element (otherwise, restart).**

Lock-free programming: Non blocking linked list:

Deletion is more difficult:

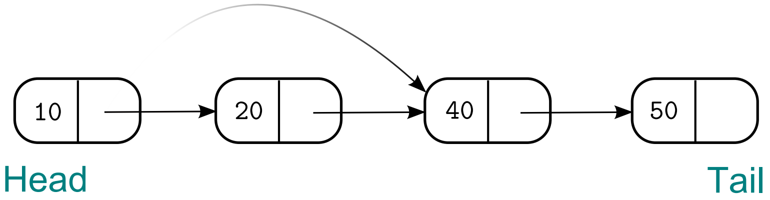


Use compare and swap to check whether in the meanwhile any update was done between the 10 and 20 element:

- **If the test suceeds then update the *next* field of the 10 element (otherwise, restart).**
- **Eventually, garbage collect the 20 element.**

Lock-free programming: Non blocking linked list:

Deletion is more difficult:

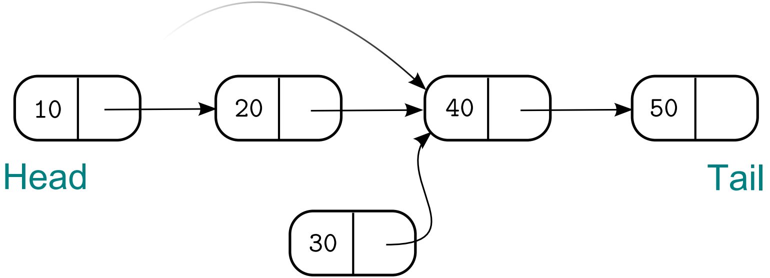


HOWEVER

If before the compare and swap another thread makes an insertion

Lock-free programming: Non blocking linked list:

Deletion is more difficult:

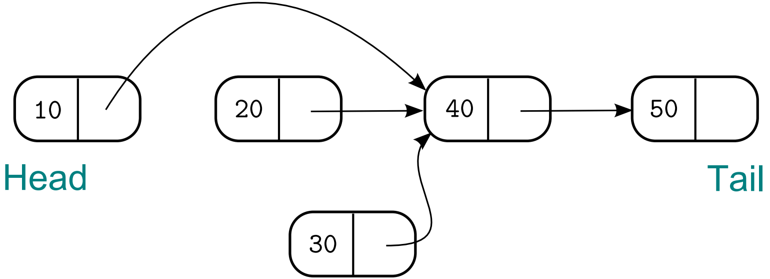


HOWEVER

If before the compare and swap another thread makes an insertion such as the above, then:

Lock-free programming: Non blocking linked list:

Deletion is more difficult:



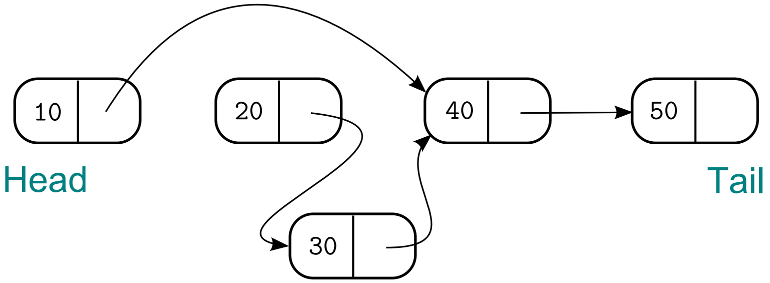
HOWEVER

If before the compare and swap another thread makes an insertion such as the above, then:

- **The compare and swap succeeds**

Lock-free programming: Non blocking linked list:

Deletion is more difficult:



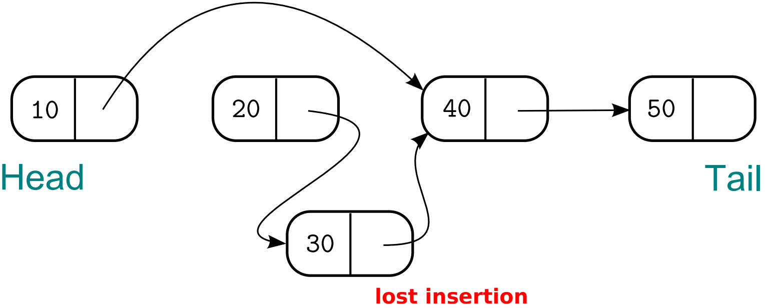
HOWEVER

If before the compare and swap another thread makes an insertion such as the above, then:

- **The compare and swap succeeds**
- **The insertion succeeds**

Lock-free programming: Non blocking linked list:

Deletion is more difficult:



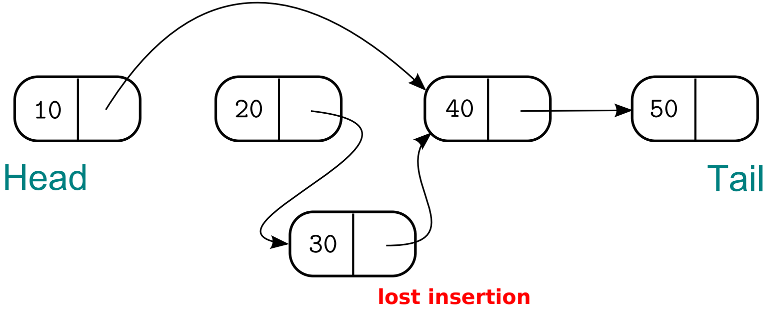
HOWEVER

If before the compare and swap another thread makes an insertion such as the above, then:

- **The compare and swap succeeds**
- **The insertion succeeds**
- **The insertion is lost**

Lock-free programming: Non blocking linked list:

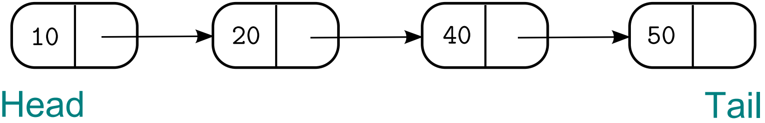
Deletion is more difficult:



we need to modify this solution

Lock-free programming: Non blocking linked list:

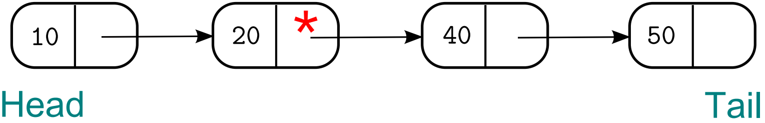
Deletion:



If we want to delete the first element then we must proceed as follows:

Lock-free programming: Non blocking linked list:

Deletion:

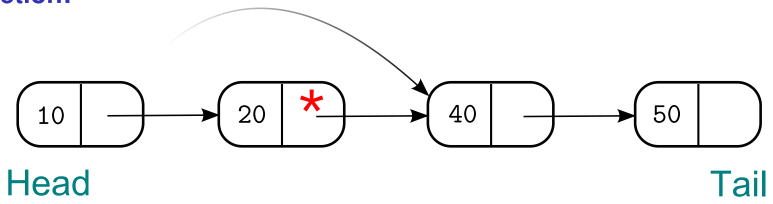


If we want to delete the first element then we must proceed as follows:

1. **Mark the element to delete: a marked point can still be traversed but will be ignored by concurrent insertion/deletions**

Lock-free programming: Non blocking linked list:

Deletion:

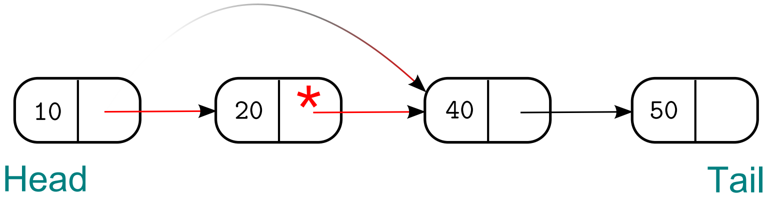


If we want to delete the first element then we must proceed as follows:

1. **Mark the element to delete: a marked point can still be traversed but will be ignored by concurrent insertion/deletions**
2. **Record the pointer to the element next to the one to be deleted**

Lock-free programming: Non blocking linked list:

Deletion:

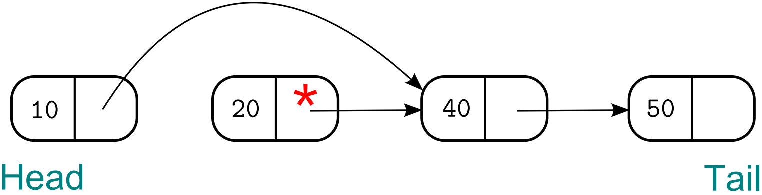


If we want to delete the first element then we must proceed as follows:

1. **Mark the element to delete: a marked point can still be traversed but will be ignored by concurrent insertion/deletions**
2. **Record the pointer to the element next to the one to be deleted**
3. **Compare the old and new pointer to the successor of the 10 element**

Lock-free programming: Non blocking linked list:

Deletion:

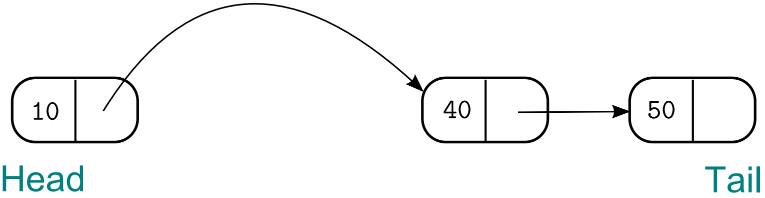


If we want to delete the first element then we must proceed as follows:

1. **Mark the element to delete: a marked point can still be traversed but will be ignored by concurrent insertion/deletions**
2. **Record the pointer to the element next to the one to be deleted**
3. **Compare the old and new pointer to the successor of the 10 element and swap them (atomic compare and swap)**

Lock-free programming: Non blocking linked list:

Deletion:



If we want to delete the first element then we must proceed as follows:

1. **Mark the element to delete: a marked point can still be traversed but will be ignored by concurrent insertion/deletions**
2. **Record the pointer to the element next to the one to be deleted**
3. **Compare the old and new pointer to the successor of the 10 element and swap them (atomic compare and swap)**
4. **Eventually the deleted node is to be garbage collected**

Lock-free programming

Summary:

- Lock-free programming requires specifically programmed *data structures* while the next solutions require specific *control structures*.
- As such, it is of a less general application than the techniques we describe next.
- Also it may not fit modular development, since a structure composed of lock-free programmed data structures may fail to avoid global conflicts.
- However when it works, then it comes from free and can be combined with any of the techniques that follow, thus reducing the logical complexity of process synchronization.

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

Cooperative multi-threading in OCaml: Lwt

Cooperative multi-threading in OCaml is available by the `Lwt` module

Rationale

Instead of using few large monolithic threads, define (1) many small interdependent threads, (2) the interdependence relation between them.

Cooperative multi-threading in OCaml: `Lwt`

Cooperative multi-threading in OCaml is available by the `Lwt` module

Rationale

Instead of using few large monolithic threads, define (1) many small interdependent threads, (2) the interdependence relation between them.

- A thread executes without interrupting till a *cooperation point* where it passes the control to another thread

Cooperative multi-threading in OCaml: Lwt

Cooperative multi-threading in OCaml is available by the `Lwt` module

Rationale

Instead of using few large monolithic threads, define (1) many small interdependent threads, (2) the interdependence relation between them.

- A thread executes without interrupting till a *cooperation point* where it passes the control to another thread
- A *cooperation point* is reached either by explicitly passing the control (function `yield`), or by calling a "cooperative" function (eg, `read`, `sleep`).

Cooperative multi-threading in OCaml: `Lwt`

Cooperative multi-threading in OCaml is available by the `Lwt` module

Rationale

Instead of using few large monolithic threads, define (1) many small interdependent threads, (2) the interdependence relation between them.

- A thread executes without interrupting till a *cooperation point* where it passes the control to another thread
- A *cooperation point* is reached either by explicitly passing the control (function `yield`), or by calling a “cooperative” function (eg, `read`, `sleep`).
- `Lwt` uses a non-preemptive scheduler which is an event loop.

Cooperative multi-threading in OCaml: `Lwt`

Cooperative multi-threading in OCaml is available by the `Lwt` module

Rationale

Instead of using few large monolithic threads, define (1) many small interdependent threads, (2) the interdependence relation between them.

- A thread executes without interrupting till a *cooperation point* where it passes the control to another thread
- A *cooperation point* is reached either by explicitly passing the control (function `yield`), or by calling a "cooperative" function (eg, `read`, `sleep`).
- `Lwt` uses a non-preemptive scheduler which is an event loop.
- At each cooperation point the thread passes the control to the scheduler, which handles it to another thread.

Cooperative multi-threading in OCaml: `Lwt`

Cooperative multi-threading in OCaml is available by the `Lwt` module

Rationale

Instead of using few large monolithic threads, define (1) many small interdependent threads, (2) the interdependence relation between them.

- A thread executes without interrupting till a *cooperation point* where it passes the control to another thread
- A *cooperation point* is reached either by explicitly passing the control (function `yield`), or by calling a "cooperative" function (eg, `read`, `sleep`).
- `Lwt` uses a non-preemptive scheduler which is an event loop.
- At each cooperation point the thread passes the control to the scheduler, which handles it to another thread.

Nota bene

Do not call blocking functions, otherwise all threads will be blocked. In particular *do not* use `Unix.sleep` and `Unix.read`, but the corresponding cooperative versions `Lwt_unix.sleep` and `Lwt_unix.read`, instead.

Lwt threads

A thread computing a value of type `'a` is a value of abstract type `'a Lwt.t`

`Lwt` threads

A thread computing a value of type `'a` is a value of abstract type `'a Lwt.t`

Each thread is in one of the following states:

1. *Terminated:* it successfully computed the value
2. *Suspended:* the computation is not over and will resume later
3. *Failed:* the computation failed (with an exception)

`Lwt` threads

A thread computing a value of type `'a` is a value of abstract type `'a Lwt.t`

Each thread is in one of the following states:

1. *Terminated:* it successfully computed the value
2. *Suspended:* the computation is not over and will resume later
3. *Failed:* the computation failed (with an exception)

Examples:

- `Lwt.return :  'a -> 'a Lwt.t`

  It *immediately* returns a *terminated* thread whose computed value is the one passed as argument.

- `Lwt_unix.sleep :  float -> unit Lwt.t`

  It *immediately* returns a *suspended* thread that will return `()` after some time (if the scheduler reschedules it).

- `Lwt.fail :  exn -> 'a Lwt.t`

  It *immediately* returns a *failed* thread whose exception is the one passed as argument.

`Lwt` threads

`Lwt` **threads are a monad:**

- `Lwt.bind :  'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t`

  The expression `Lwt.bind p f` (that can also be written as `p >>= f`) *immediately* returns a thread of type `'b Lwt.t`, defined as follows:

  - If the thread `p` is *terminated*, then it passes its results to `f`.
  - If the thread `p` is *suspended*, then `f` is saved in the list of the functions waiting for the result of `p`. When `p` terminates, then the scheduler actives these functions one after the other.
  - If the thread `p` is *failed*, then so is the whole expression.

`Lwt` threads

`Lwt` threads are a monad:

- `Lwt.bind :  'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t`

  The expression `Lwt.bind p f` (that can also be written as `p >>= f`) *immediately* returns a thread of type `'b Lwt.t`, defined as follows:

  - If the thread `p` is *terminated*, then it passes its results to `f`.
  - If the thread `p` is *suspended*, then `f` is saved in the list of the functions waiting for the result of `p`. When `p` terminates, then the scheduler actives these functions one after the other.
  - If the thread `p` is *failed*, then so is the whole expression.

Nota bene

Bind is not a cooperation point: it does not imply any suspension

Example

```
rlwrap ocaml
Objective Caml version 3.11.1
# #use "topfind";;

# #require "lwt.unix";;

# Lwt.bind (Lwt.return 3) (fun x ->
            print_int x; Lwt.return());;
3- : unit Lwt.t = <abstr>

# Lwt.bind (Lwt_unix.sleep 3.0) (fun () ->
            print_endline "hello"; Lwt.return ());;
- : unit Lwt.t = <abstr>
```

- `(Lwt.return 3) >>= (fun x -> print_int x; Lwt.return())` immediately returns a thread of type `unit Lwt.t` after having printed `3`. Notice the use of `Lwt.return ()` for well typing
- `(Lwt_unix.sleep 3.0) >>= (fun () -> print_endline "hello"; Lwt.return ())` immediately returns a thread of type `unit Lwt.t` and nothing else.

In order to see the last thread to behave as expected (and print after three seconds “hello”) we have to run the scheduler, that is the function `Lwt_unix.run :  'a Lwt.t -> 'a`

- `Lwt_unix.run t` manage a queue of waiting threads without preemption. It terminates when the thread `t` does.
- When we run the scheduler we see the computation above to end (since the schedule reactivates `Lwt_unix.sleep 3.0` which can pass its hand to the next thread (and then it ends after 30 seconds).

```
# Lwt_unix.run (Lwt_unix.sleep 30.);;
hello
- : unit = ()
```

Lwt_unix

The main function provided by Lwt_unix is run:

- It manages a queue of threads ready to be executed. As long as this queue is not empty it runs them in the order.
- It maintains a table of open file descriptors together with the threads that wait on them and insert them in the queue as soon as they have received the data they were waiting for.
- It inserts in the queue the threads that exceeded their sleep time.
- It iterates and stops when its argument thread does

Lwt_unix

The main function provided by `Lwt_unix` is `run`:

- It manages a queue of threads ready to be executed. As long as this queue is not empty it runs them in the order.
- It maintains a table of open file descriptors together with the threads that wait on them and insert them in the queue as soon as they have received the data they were waiting for.
- It inserts in the queue the threads that exceeded their sleep time.
- It iterates and stops when its argument thread does

Besides the scheduler `Lwt_unix` provides the cooperative version of most of the functions in the `Unix` module:

- `Lwt_unix.yield : unit -> unit Lwt.t` forces a cooperation point adding the thread in the scheduler queue.
- `Lwt_unix.read`. Works as `Unix.read` but while the latter immediately blocks, the former immediately returns a new thread which:
  - It tries to read the file
  - If data is available, then it returns a result
  - Otherwise, it sleeps in the queue associated to the file descriptor

“do” notation for Lwt_unix

It is possible to use the “lwt_in” notation to mimic Haskell’s “do” notation. So

```
Lwt_chan.input_line ch >>= fun s ->
Lwt_unix.sleep 3. >>= fun () ->
print_endline s;
Lwt.return ()
```

can be written as

```
lwt s = input_line ch in
lwt () = Lwt_unix.sleep 3. in
print_endline s;
Lwt.return ()
```

Example

A thread that writes “hello” every ten seconds

```
let rec f () =
print_endline "hello";
Lwt_unix.sleep 10. >>= f
in f ();
```

Example

A thread that writes “hello” every ten seconds

```
let rec f () =
print_endline "hello";
Lwt_unix.sleep 10. >>= f
in f ();
```

Join of threads

Let f and g be functions that return threads (e.g., `unit -> ‘a Lwt.t`)

```
let first_thread = f () in           // launch the first thread
let second_thread = g () in          // launch the second thread
lwt fst_result = first-thread in     // wait for the first thread result
lwt snd_result = second_thread in    // wait for the second thread result
  ⋮
```

Two versions of cooperative `List.map`

Two versions of `map` running a thread for each value in the list.

```
 # let rec map f l =
   match l with
   | [] -> return []
   | v :: r ->
      let t = f v in
      let rt = map f r in
      t >>= fun v’ ->
      rt >>= fun l’ ->
      return (v’ :: l’);;
 val map : (’a -> ’b Lwt.t) -> ’a list -> ’b list Lwt.t = <fun>
```

```
let rec map2 f l =
  match l with
  | [] -> return []
  | v :: r ->
      f v >>= fun v’ ->
      map2 f r >>= fun l’ ->
      return (v’ :: l’)
 val map : (’a -> ’b Lwt.t) -> ’a list -> ’b list Lwt.t = <fun>
```

Two versions of cooperative `List.map`

Two versions of `map` running a thread for each value in the list. Differences?

```
# let rec map f l =
  match l with
  | [] -> return []
  | v :: r ->
     let t = f v in
     let rt = map f r in
     t >>= fun v' ->
     rt >>= fun l' ->
     return (v' :: l');;
val map : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

```
let rec map2 f l =
  match l with
  | [] -> return []
  | v :: r ->
      f v >>= fun v' ->
      map2 f r >>= fun l' ->
      return (v' :: l')
 val map : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

Two versions of cooperative `List.map`

Two versions of `map` running a thread for each value in the list. Differences?

```
# let rec map f l =
  match l with
  | [] -> return []
  | v :: r ->
     let t = f v in
     let rt = map f r in
     t >>= fun v' ->
     rt >>= fun l' ->
     return (v' :: l');;
val map : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

All the threads are run “in parallel”.

```
let rec map2 f l =
  match l with
  | [] -> return []
  | v :: r ->
     f v >>= fun v' ->
     map2 f r >>= fun l' ->
     return (v' :: l')
 val map : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

The threads are run sequentially: it waits for the first thread to end before running the second

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

Communication on channels

Two kinds of communication:

1. *Synchronous communication:*
   sending a message on an action is blocking.

   Module `Event` of the OCaml's thread library.

2. *Asynchronous communications*:
   sending a message is a non-blocking action: messages are buffered and the order is preserved.

   Erlang-style concurrency.

Synchronous communications: OCaml's `Event`

- `type 'a channel`
  The type of communication channels carrying values of type `'a`.
- `new_channel :  unit -> 'a channel`
  Return a new channel.
- `type +'a event`
  The type of communication events returning a result of type `'a`.
- `send :  'a channel -> 'a -> unit event`
  `send ch v` returns the event consisting in sending the value `v` over the channel `ch`. The result value of this event is `()`.
- `receive :  'a channel -> 'a event`
  `receive ch` returns the event consisting in receiving a value from the channel `ch`. The result value of this event is the value received.
- `choose :  'a event list -> 'a event`
  `choose evl` returns the event that is the parallel composition of all the events in the list `evl`.

Synchronous communications: OCaml's `Event`

The functions `send` and `receive` are not blocking functions

The primitives `send` and `receive` build the elementary events “sending a message” or “receiving a message” but do not have an immediate effect.

- They just create a data structure describing the action to be done
- To make an event happen, a thread must synchronize with another thread wishing to make the complementary event happen
- The `sync` primitive allows a thread to wait for the occurrence of the event passed as argument.
  - `sync : 'a event -> 'a`
    “Synchronize” on an event: offer all the communication possibilities specified in the event to the outside world, and block until one of the communications succeeds. The result value of that communication is returned.
  - `poll : 'a event -> 'a option`
    Non-blocking version of `Event.sync`: offer all the communication possibilities specified in the event to the outside world, and if one can take place immediately, perform it and return `Some r` where `r` is the result value of that communication. Otherwise, return `None` without blocking.

Example: access a shared variable via communications

```
# let ch = Event.new_channel () ;;
# let v = ref 0;;

# let reader () = Event.sync (Event.receive ch);;
# let writer () = Event.sync (Event.send ch ("S" ^ (string_of_int !v)));;

# let loop_reader s d () = (
    for i=1 to 10 do
     let r = reader() in print_string (s ^ ":" ^ r ^ "; "); flush stdout;
     Thread.delay d
    done;
    print_newline());;
# let loop_writer d () =
    for i=1 to 10 do incr v; writer(); Thread.delay d done;;

# Thread.create (loop_reader "A" 1.1) ();;
# Thread.create (loop_reader "B" 1.5) ();;
# Thread.create (loop_reader "C" 1.9) ();;

# loop_writer 1. ();;
A:S1; C:S2; B:S3; A:S4; C:S5; B:S6; A:S7; C:S8; B:S9; A:S10;
- : unit = ()
# loop_writer 1. ();;
C:S11; B:S12; A:S13; C:S14; B:S15; A:S16; C:S17; B:S18; A:S19; C:S20;
- : unit = ()
```

Example: Single position queue

We modify the Queue example so as mutual exclusion is obtained by channels rather than mutexes. We keep the same interface

```
module Cell = struct
  type 'a t =
    { add_ch: 'a Event.channel; rmv_ch: 'a Event.channel }

  let create () =
    let aCh = Event.new_channel ()
    and rCh = Event.new_channel () in
    let rec empty () =
      let e = Event.sync (Event.receive aCh) in
      full e
    and full e =
      empty (Event.sync (Event.send rCh e))
    in
      ignore (Thread.create empty ());
      {add_ch = aCh ; rmv_ch = rCh}

   let add e q =
     Event.sync(Event.send q.add_ch e)

   let remove q =
     Event.sync (Event.receive q.rmv_ch)
end
```

Example: Single position queue

We modify the Queue example so as mutual exclusion is obtained by channels rather than mutexes. We keep the same interface

```
module Cell = struct
  type 'a t =
    { add_ch: 'a Event.channel; rmv_ch: 'a Event.channel }

  let create () =
    let aCh = Event.new_channel ()
    and rCh = Event.new_channel () in
    let rec empty () =
      let e = Event.sync (Event.receive aCh) in
      full e
    and full e =
      empty (Event.sync (Event.send rCh e))
    in
      ignore (Thread.create empty ());
      {add_ch = aCh ; rmv_ch = rCh}

   let add e q = ignore(
     Event.poll(Event.send q.add_ch e) )     //multi-position buffer

   let remove q =
     Event.sync (Event.receive q.rmv_ch)
end
```

Asynchronous communications: Erlang's concurrency

Erlang is a (open-source) general-purpose concurrent programming language and runtime system designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications.

Asynchronous communications: Erlang's concurrency

Erlang is a (open-source) general-purpose concurrent programming language and runtime system designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications.

- The sequential subset of Erlang is a strict and dynamically typed functional language. For concurrency it follows the Actor model. Three primitives
  - `spawn` spawns a new process
  - `send` asynchronously send a message to a given process
  - `receive` reads one the of the received messages
- Concurrency is structured around *processes*.
  Erlang processes are are not OS processes: they are much lighter (scale up to hundreds of million of processes). Like OS processes and unlike OS threads or green threads they have no shared state between them.
- Process communication is done via *asynchronous message passing*: every process has a "mailbox", a queue of messages that have been sent by other processes and not yet consumed.

- Process creation

  ```
  Pid = spawn(Module, FunctionName, ArgumentList)
  ```

  Spawns a new process that executes the function `FunctionName` in the module `Module` with arguments `ArgumentList` and returns *immediately* its identifier.

- Asynchronous send

  ```
          Pid ! Message
  ```

  Put the message `Message` in the buffer of the process whose identifier is `Pid`. So `foo(12) !  bar(baz)` will first evaluate `foo(12)` to get the process identifier and `bar(baz)` for the message to send, and returns *immediately* (the value of the message) without waiting for the message either to arrive at the destination or to be received.

- Selective receive

  ```
  receive
     Pattern1  -> Actions1 ;
     Pattern2  -> Actions2 ;
        ⋮
  end
  ```

  Select in the mailbox the first message that matches a pattern, remove it from the mailbox, and execute its actions. Otherwise suspend.

An example

```
% Create a process and invoke the function
% web:start_server(Port, MaxConnect)
ServerProcess = spawn(web, start_server, [Port, MaxConnect]),

% [Distribution support]
% Create a remote process and invoke the function
% web:start_server(Port, MaxConnect) on machine RemoteNode
RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnect]),

% Send a message to ServerProcess (asynchronously). The message consists
% of a tuple with the atom "pause" and the number "10".
ServerProcess ! pause, 10,

% Receive messages sent to this process
receive
       data, DataContent -> handle(DataContent);
       hello, Text -> io:format("Got hello message: ~s", [Text]);
       goodbye, Text -> io:format("Got goodbye message: ~s", [Text])
end.
```

Erlang-style concurrency

This style of concurrency has been adopted in several other languages

- F#: *cf.* the `MailboxProcessor` class
- Scala: uses the same syntax (and semantics) as Erlang but instead of processes we have “actor objects” that run in separate threads.
- Retlang a Erlang inspired library for .NET and `Jetlang` its Java counterpart.
- Others: Termite Scheme, Coro Module for Perl, ...

Outline

52 Concurrency

53 Preemptive multi-threading

54 Locks, Conditional Variables, Monitors

55 Doing without mutual exclusion

56 Cooperative multi-threading

57 Channeled communication

58 Software Transactional Memory

Compositionality

The main problem with locks is that they are not compositional

```
action1 = withdraw a 100
```

```
action2 = deposit b 100
```

```
action3 =
   do withdraw a 100
       Inconsistent state
      deposit b 100
```

Compositionality

The main problem with locks is that they are not compositional

```
action1 = withdraw a 100
```

```
action2 = deposit b 100
```

```
action3 =
   do withdraw a 100
       Inconsistent state
      deposit b 100
```

Solution: Expose all locks

```
action3 =
   do lock a
      lock b
      withdraw a 100
      deposit b 100
      release a
      release b
```

Compositionality

The main problem with locks is that they are not compositional

```
action1 = withdraw a 100
```

```
action2 = deposit b 100
```

```
action3 =
   do withdraw a 100
       Inconsistent state
      deposit b 100
```

Solution: Expose all locks

```
action3 =
   do lock a
      lock b
      withdraw a 100
      deposit b 100
      release a
      release b
```

Problems

- Risk of deadlocks
- Unfeasible for more complex cases

Software transactional memory

Idea

- Borrow ideas from database people
    - ACID transactions:
        - **atomic** - all changes commit or all changes roll back; changes appear to happen at a single moment in time
        - **consistent** - operate on a snapshot of memory using newest values at beginning of the txn
        - **isolated** - changes are visible to other threads, only after commit and, viceversa, threads in a transaction cannot see other thread changes.
        - **durable** - does not apply to STM (changes do not persist on disk) but we can adapt it when changes are lost if software crashes or hardware fails (cf. locks).
- Add ideas from (pure) functional programming
    - Computations are first-class values
    - What side-effects can happen and where they can happen is controlled

Software transactional memory

Idea

- Borrow ideas from database people
  - ACID transactions:
    - **atomic** - all changes commit or all changes roll back; changes appear to happen at a single moment in time
    - **consistent** - operate on a snapshot of memory using newest values at beginning of the txn
    - **isolated** - changes are visible to other threads, only after commit and, viceversa, threads in a transaction cannot see other thread changes.
    - **durable** - does not apply to STM (changes do not persist on disk) but we can adapt it when changes are lost if software crashes or hardware fails (cf. locks).
- Add ideas from (pure) functional programming
  - Computations are first-class values
  - What side-effects can happen and where they can happen is controlled

Software Transactional Memory

- First ideas in 1993
- New developments in 2005
  - (Simon Peyton Jones, Simon Marlow, Tim Harris, Maurice Herlihy)

Atomic transactions

Write sequential code, and wrap `atomically` around it

```
action3 =
   atomically{
      withdraw a 100
      deposit b 100
   }
```

Atomic transactions

Write sequential code, and wrap atomically around it

```
action3 =
   atomically{
      withdraw a 100
      deposit b 100
   }
```

How does it works?

- Execute body without locks
- Each memory access is logged to a thread-local transaction log.
- No actual update is performed in memory
- At the end, we try to *commit* the log to memory
- Commit may fail, then we retry the whole atomic block

Atomic transactions

Write sequential code, and wrap `atomically` around it

```
action3 =
   atomically{
      withdraw a 100
      deposit b 100
   }
```

How does it works?

- Execute body without locks
- Each memory access is logged to a thread-local transaction log.
- No actual update is performed in memory
- At the end, we try to *commit* the log to memory
- Commit may fail, then we retry the whole atomic block

Optimistic concurrency

Caveats

Simon Peyton-Jones's missile program:

```
action3 =
atomically{
   withdraw a 100
   launchNuclearMissiles
   deposit b 100
}
```

No side effects allowed!

Caveats

Simon Peyton-Jones's missile program:

```
action3 =
atomically{
   withdraw a 100
   launchNuclearMissiles
   deposit b 100
}
```

No side effects allowed!

More in details:

The logging capability is implemented by specific “transactional variables”.

Absolutely forbidden:

- To read a transaction variable outside an atomic block
- To write to a transaction variable outside an atomic block
- To make actual changes (eg, file or network access, use of non-trasactional variables) inside an atomic block...

Caveats

Simon Peyton-Jones's missile program:

```
   action3 =
   atomically{
      withdraw a 100
      launchNuclearMissiles
      deposit b 100
   }
```

No side effects allowed!

More in details:

The logging capability is implemented by specific "transactional variables".

Absolutely forbidden:

- To read a transaction variable outside an atomic block
- To write to a transaction variable outside an atomic block
- To make actual changes (eg, file or network access, use of non-trasactional variables) inside an atomic block...

These constraints are enforced by the type system

STM in Haskell

- Fully-fledged implementation of STM: `Control.Concurrent.STM`
- Implemented in the language also in Clojure and Perl6.
- Implementations for C++, Java, C#, F# being developed as libraries ... difficult to solve all problems
- In Haskell, it is easy: controlled side-effects

```
type STM a
instance Monad STM
    atomically  :: STM a -> IO a
    retry       :: STM a
    orElse      :: STM a -> STM a -> STM a

type TVar a
    newTVar     :: a -> STM (TVar a)
    readTVar    :: TVar a -> STM a
    writeTVar   :: TVar a -> a -> STM ()
```

Sides effects must be performed on specific “transactional variables” `TVar`

STM: `TVar`

Threads in STM Haskell communicate by reading and writing transactional variables

```
type Resource = TVar Int

putR :: Resource -> Int -> STM ()
putR r i = do v <- readTVar r
              writeTVar r (v+i)

main = do ... atomically (putR r 13) ...
```

Operationally, `atomically` takes the tentative updates and actually applies them to the `TVars` involved. The system maintains a per-thread transaction log that records the tentative accesses made to `TVars`.

STM: retry

retry: Retries execution of the current memory transaction because it has seen values in TVars which mean that it should not continue (e.g., the TVars represent a shared buffer that is now empty). The implementation may block the thread until one of the TVars that it has read from has been updated.

```
retry :: STM a

atomically {if n_items == 0 then retry
                 else ...remove from queue...}
```

STM: `retry`

`retry`: Retries execution of the current memory transaction because it has seen values in TVars which mean that it should not continue (e.g., the TVars represent a shared buffer that is now empty). The implementation may block the thread until one of the TVars that it has read from has been updated.

```
retry :: STM a

atomically {if n_items == 0 then retry
                else ...remove from queue...}
```

In summary:

- `retry` says “abandon the current transaction and re-execute it from scratch”
- The implementation *waits* until `n_items` changes
- No *condition variables*, no lost wake-ups!

Blocking composes

```
atomically { x = queue1.getItem()
           ; queue2.putItem( x ) }
```

- If either `getItem` or `putItem` retries, the whole transaction retries
- So the transaction waits until queue1 is not empty AND queue2 is not full
- No need to re-code getItem or putItem
- (Lock-based code does not compose)

STM `orElse`

`orElse`: it tries two alternative paths:

- If the first retries, it runs the second
- If both retry, the whole `orElse` retries.

```
orElse :: STM a -> STM a -> STM a

atomically { x = queue1.getItem();
                queue2.putItem(x)
           ‘orElse‘
                queue3.putItem(x) }
```

- So the transaction waits until
  - queue1 is non-empty, AND
  - EITHER queue2 is not full OR queue3 is not full

  without touching `getItem` or `putItem`

Note:

```
m1 ‘orElse‘ (m2 ‘orElse‘ m3) = (m1 ‘orElse‘ m2) ‘orElse‘ m3
            retry ‘orElse‘ m = m
            m ‘orElse‘ retry = m
```

Compositionality

- All transactions are flat
- Calling transactional code from the current transaction is normal
- This simply extends the current transaction

STM in Haskell

- Safe transactions through type safety
  - A very specific monad STM (distinct from I/O)
  - We can only access TVars
  - TVars can only be accessed in STM monad
  - Referential transparency inside blocks
- Explicit retry – expressiveness
- Compositional choice – expressiveness

STM in Haskell

- Safe transactions through type safety
  - A very specific monad STM (distinct from I/O)
  - We can only access TVars
  - TVars can only be accessed in STM monad
  - Referential transparency inside blocks
- Explicit retry – expressiveness
- Compositional choice – expressiveness

Problems

- Overhead: managing transactions bookkeeping requires some overhead
- Starvation: could the system "thrash" by continually colliding and re-executing?
  - No: one transaction can be forced to re-execute only if another succeeds in committing. That gives a strong progress guarantee.
  - But a particular thread could perhaps starve.
- Performance: potential for many retries resulting in wasted work
- Tools: support is currently lacking
  - for learning which memory locations experienced write conflicts
  - for learning how often each transaction is retried and why

Problems in C++/Java/C#

- Retry semantics
- IO in atomic blocks
- Access of transaction variables outside of atomic blocks
- Access to regular variables inside of atomic blocks

A useful analogy

Switch from manual to automatic gear:

A useful analogy

Switch from manual to automatic gear:

- **Memory management**
  - malloc, free, manual refcounting
  - Garbage collection

A useful analogy

Switch from manual to automatic gear:

- **Memory management**
  - malloc, free, manual refcounting
  - Garbage collection
- **Concurrency**
  - Mutexes, semaphores, condition variables
  - Software transactional memory

Manual/auto tradeoffs

Same kind of tradeoffs:

Manual/auto tradeoffs

Same kind of tradeoffs:

- **Memory management**
  - Manual: Performance, footprint
  - Auto: Safety against memory leaks, corruption

Manual/auto tradeoffs

Same kind of tradeoffs:

- **Memory management**
  - Manual: Performance, footprint
  - Auto: Safety against memory leaks, corruption
- **Concurrency**
  - Manual: Fine tuning for high contention
  - Auto: Safety against deadlock, corruption

Manual/auto tradeoffs

Same kind of tradeoffs:

- **Memory management**
  - Manual: Performance, footprint
  - Auto: Safety against memory leaks, corruption
- **Concurrency**
  - Manual: Fine tuning for high contention
  - Auto: Safety against deadlock, corruption

Rationale

In both cases you pay in terms of performance and of “I do not quite know what is going on”, but they allow you to build larger, more complex systems that won’t break because of wrong “by hand” management.