

Hello, would you like to hear a TCP joke?

Yes, I'd like to hear a TCP joke.

OK, I'll tell you a TCP joke.

OK, I'll hear a TCP joke.

Are you ready to hear a TCP joke?

Yes, I am ready to hear a TCP joke.

OK, I'm about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline.

OK, I'm ready to hear the TCP joke that will last 10 seconds, has two characters, does not have a setting and will end with a punchline.

I'm sorry, your connection has timed out... ..Hello, would you like to hear a TCP joke?

The demonstration

```
i = i + 1
```

Three operations

Read

Modify

Write

We can be interrupted at any time

A thread can "overwrite" the work done by another

The problem

Using a resource simultaneously

The result depends on ordering in time

This is known as a **Race Condition**

Two solutions

Prevent things from happening at the same time

- Introduce a "bottleneck"

- This removes concurrency!

Do not share resources

- Send messages

- This is a different "way of thinking". The problem must be "translated"

Bottlenecks

Some sort of "flag" that says if a resource is being used

Modifying this flag must not introduce another race condition

Flag modifications must be indivisible

An **Atomic Operation**

Implemented using hardware instructions

x86 lock prefix

CMPXCHG

How this works isn't part of this course

Bottlenecks

Semaphore

Integer flag with value always greater than or equal to 0

Two operations:

`wait()` – decrements by one *P / Prolaag / Probeer te verlagen*

`signal()` – increments by one *V / Verhogen*

Aka. `notify()`, to indicate that control is not transferred to the waiter

⇒ Cannot decrement if the value is zero

Thread(s) will be awoken when **someone else** `signal()`s

Bottlenecks

Binary semaphore (as opposed to **Counting Semaphore**)

Can only have values 1 or 0

Available/unavailable

Locked/unlocked

Require that only the one using it can `signal()`

A concept of ownership

This is known as a **Mutex** (MUTual EXclusion)

With ownership we can also provide **Priority Inheritance**

Bottlenecks

Granularity

- "Course-grained" locking: one huge global lock

 - Complete denial of concurrent execution

- "Fine-grained" locking: one lock per resource

 - Quickly gets out of hand with many locks & resources

Creating a mutex

C

```
#include <pthread.h>

int main(){
    pthread_mutex_t mtx;

    // 2nd arg is a pthread_mutexattr_t
    pthread_mutex_init(&mtx, NULL);

    pthread_mutex_lock(&mtx);
    // Critical section
    pthread_mutex_unlock(&mtx);

    pthread_mutex_destroy(&mtx);
}
```

<http://pubs.opengroup.org/onlinepubs/7990989775/xsh/pthread.h.html>

Python

```
from threading import Lock
```

```
mtx = Lock()
```

```
mtx.acquire()
# Critical section
mtx.release()
```

OR:

```
with mtx:
    # Critical section
# (Scope end)
```

<http://docs.python.org/2/library/threading.html#lock-objects>

Message passing

	Synchronization	Communication
Message passing	Explicit (if synchronous) Implicit	Explicit
Shared Variables	Explicit	Implicit

The two approaches are fundamentally different

Message passing is “no-sharing” by default

Share memory by communicating, instead of communicating by sharing memory

Message passing

Synchronous vs asynchronous:

Sender must wait until receiver can receive

"Phone" vs "Mail"

Symmetric vs asymmetric:

Receiver specifies where (which sender/channel) it is receiving from

"Instant messaging" vs "email"

Go channels

```
func make(Type, size IntegerType) Type
```

```
someChannel := make(chan T)           // makes a synchronous channel
```

```
someChannel := make(chan T, 5)        // makes a buffered (async) channel
```

```
make(chan T, 1)    - An interesting special case: See the mutex similarity?
```

```
func foo(someChannel <-chan int) {} // this function can only read from  
                                   // the channel
```

```
func bar(someChannel chan<- int) {} // this function can only write to  
                                   // the channel
```

Go channels

```
// Writing/sending  
// Will only happen if  
// 1) someone is waiting to read (unbuffered)  
// or 2) the buffer is not full (buffered)  
someChannel <- val  
  
// Reading/receiving  
// Will only happen when there is a value to read  
val1 int  
val1 <- someChannel  
val2 := <- someChannel  
// Receive & discard. Useful when waiting for events  
<- someChannel
```

Go channels

The “correct way”:

```
for {  
    select {  
        // Receiving  
        case msg1 := <- chan1:  
            // action 1  
        case msg2 := <- chan2:  
            // action 2  
        // Sending (use with caution...)  
        case chan3 <- msg3:  
            // action 3, whenever we sent something on chan3  
    }  
}
```

Go channels

Sending a message invokes a behaviour in the recipient

Everything is triggered on events

Any of these could happen at any time:

Consider them all "simultaneously", take one action, consider them again

Always responsive to any of them

No need for mutual exclusion

Does NOT mean there is no globally shared state

Solving concurrent access with MP

Think of threads/routines as "actors"/"office workers"

Put the resource in their mailbox, and have them put it in another mailbox when they are done with it.

If others need to read the resource, have the workers scribble over each other on the office whiteboard.

Recruit a worker to manage the keys to the room with the resource. This worker either has the keys, or has lent them out.

Avoiding concurrent acces with MP

Give the resource to a worker (a server)

And then tell **them** to change it

For-select-loop with:

Cases for each kind of modification (here: adding & subtracting)

A case for getting a copy of the current value