SWEN90004
Modelling Complex Software Systems

Concurrent programming languages

Nic Geard

Lecture Con.12

Semester 1, 2019

Concurrency via shared memory

Processes (threads) interact with one another (communicate) via reading and writing to shared memory.

It is therefore critical to protect the integrity of data stored in shared memory by ensuring that only one process has access to it at a time.

We looked at how this could be achieved using semaphores and monitors.

Issues with shared memory

Commands to control access to shared memory is scattered all over the code.

This can make code hard to read, error prone, and difficult to modify.

Concurrency via shared memory

Java handles this by providing higher order concurrency objects; for example

- `Lock`: an explicit version of the implicit lock used by synchronzed code, which offers advantages such as fairness (keeps a queue of waiting processes), ability to 'give up' if lock is not immediately available, or after a timeout.
- `ThreadPool`: a collection of constantly running threads, that will be used/reused as required as tasks queue and complete, reducing the overhead of allocating and deallocating threads.
- `BlockingQueue`: a thread-safe, first-in-first-out data structure that blocks or times out when the queue is full/empty when attempting to add/remove items.

Concurrency via message passing

In FSP, there was no shared memory. Processes interacted via shared actions, which could be used to send and receive data:

```
VAR = VAR[0],
VAR[u:T] = (read[u]->VAR[u] | write[v:T]->VAR[v]).
```

This approach derives from Hoare's original CSP (Communicating Sequential Processes) model, which has also been the inspiration for a (growing) number of programming languages: Occam, Erlang, Concurrent ML, Go, Rust, etc.

Message passing allows us to avoid many of the issues that arise from shared memory.

Message passing

Communication requires too processes: one to send a message and one to recieve it.

In synchronous communication, the exchange of a message is an *atomic* action requiring the participation of both the *sender* and the *receiver* of the message.

The unavailability of one of these parties blocks the other.

The act of communicating synchronises the execution of two processes.

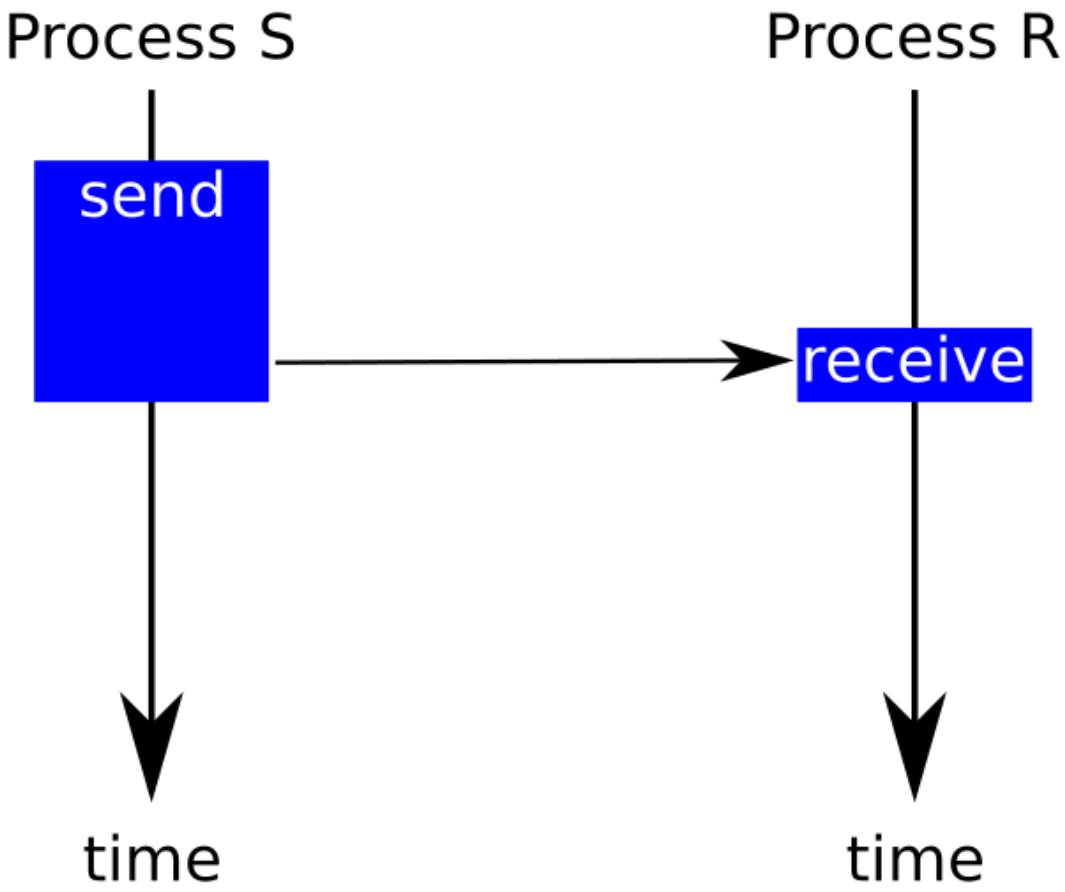
Message passing

In asynchronous communication, there is no temporal dependence between the execution of the two processes.

The receiver could be executing another statement when the message is sent, and only later check the communication channel for messages. This of course requires that the communication channel be capable of buffering messages.

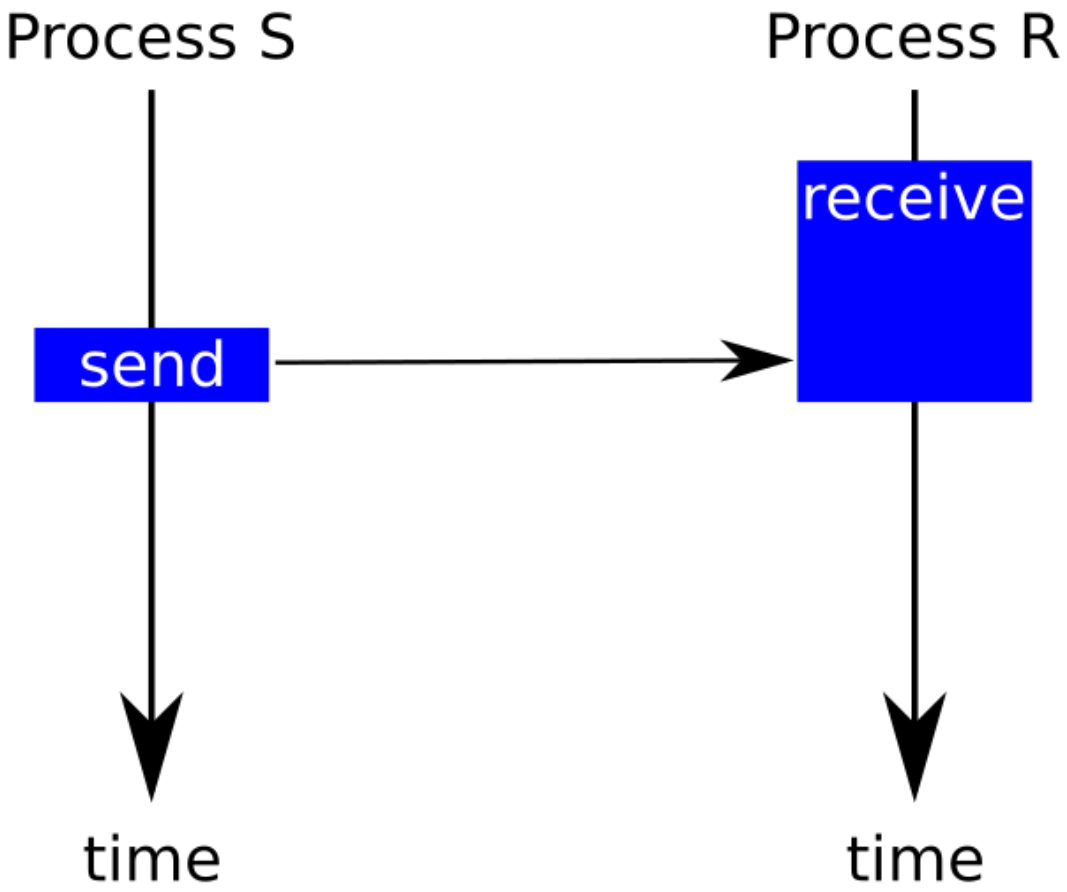
Message protocols

Synchronous message: sender waits



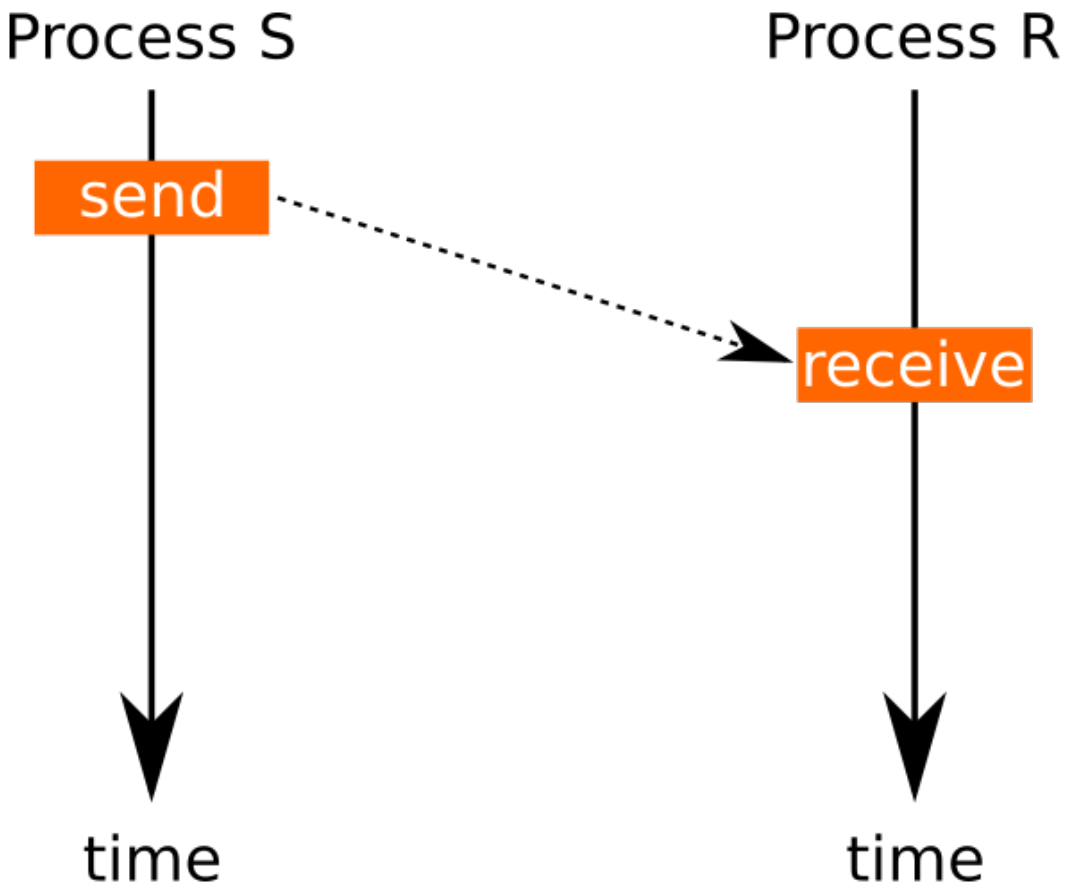
Message protocols

Synchronous message: receiver waits



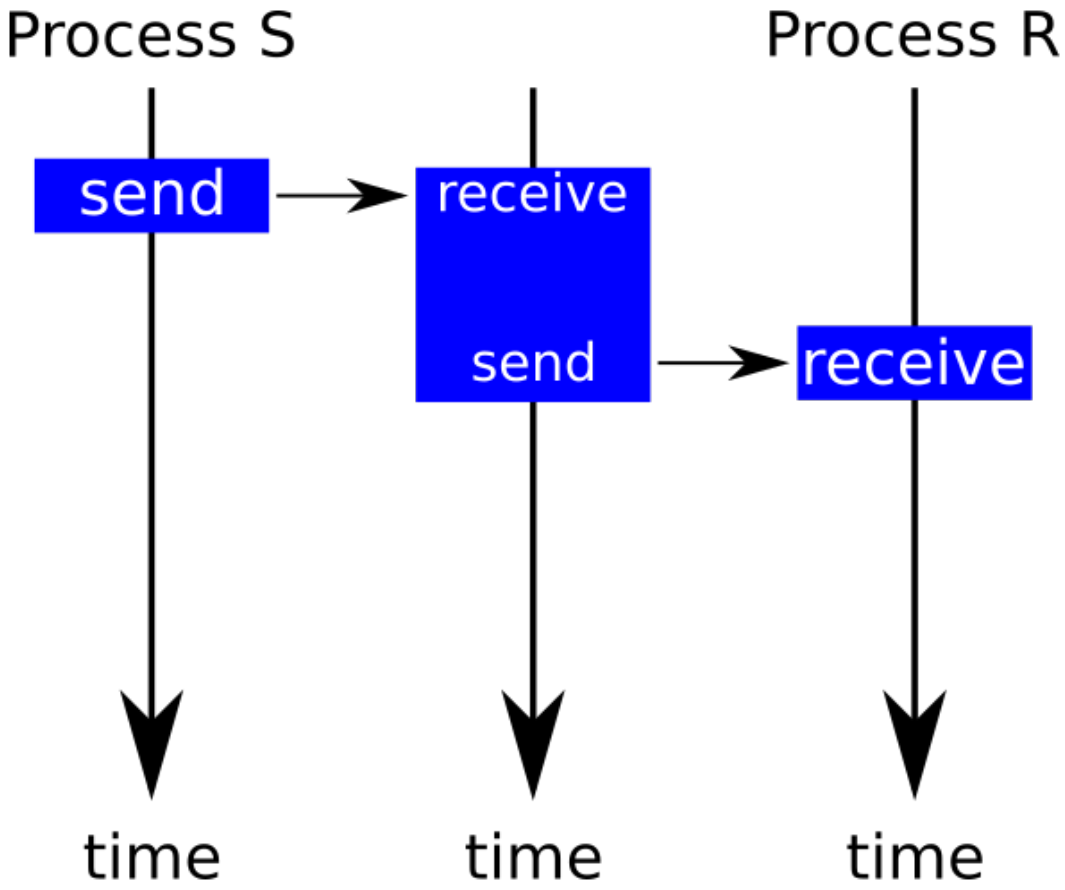
Message protocols

Asynchronous message



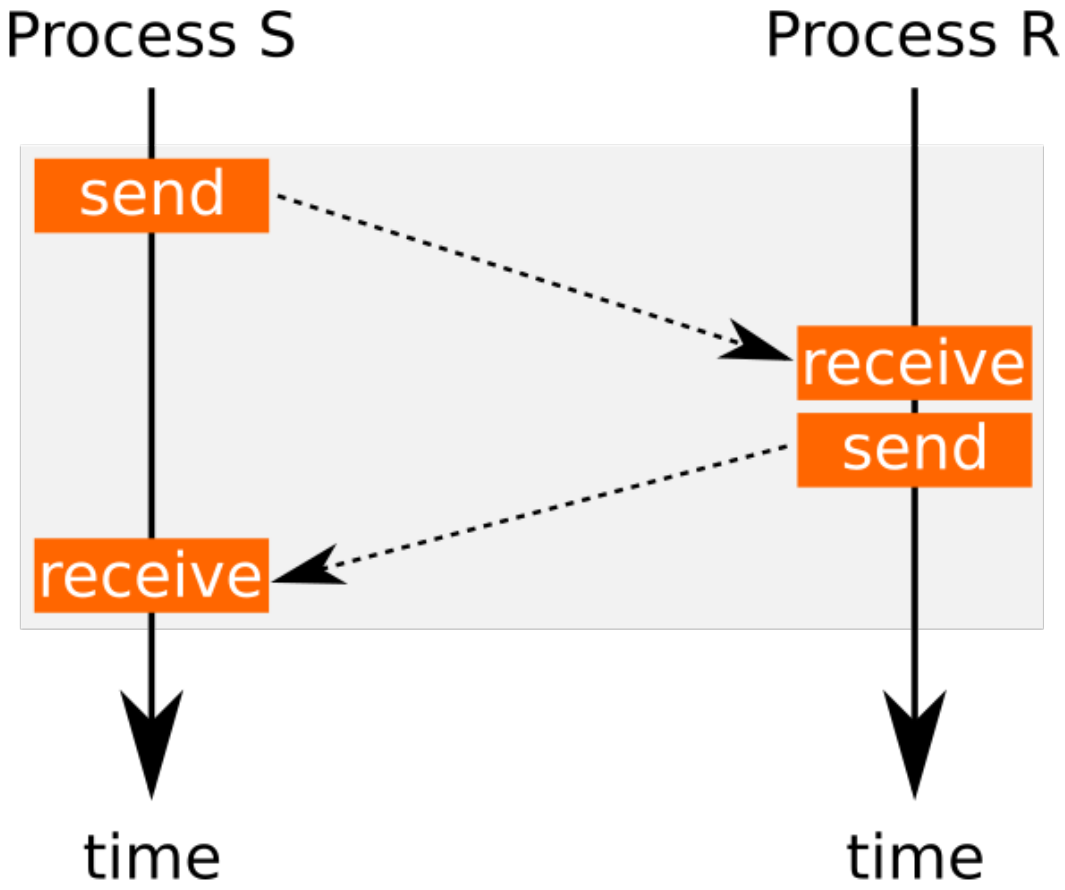
Message protocols

Simulating asynchronous messages



Message protocols

Simulating synchronous messages



Analogies

Synchronous message == a telephone call

- both participants need to be available
- their activity becomes 'synchronised' at the point of the call
- if one is unavailable, the other is blocked

Asynchronous message == email

- a participant may send any number of messages
- the receiver may choose to check incoming email at any time
- capacity is limited

Other characteristics

Addressing

May be asymmetric if the receiver does not know the identity of the caller (a telephone call), or symmetric (an email).

Data flow

May be one way (an email) or two-way (a telephone call).

Choice of which protocol is appropriate will depend on hardware available and requirements of a specific scenario.

Go as a concurrent language

Go is a programming language created by Google in 2009. It implements concurrent programming features based upon CSP.

In Go, processes are called go-routines. They have been designed to be ultra-lightweight; you may well have tens or hundreds of thousands running concurrently.

While Go does allow for go-routines to communicate via shared variables, this is not recommended. It raises all the usual synchronisation problems and calls for locks or similar devices.

Instead, go-routines usually communicate by sending and receiving data on named channels. And this is how synchronisation happens.

Go channels

A channel is a communication pipe; it can be one-way or two-way.

Any number of channels can be created and used.

```
var a chan int      // declare a: channel of ints
var b chan bool
a = make(chan int)  // create channel a
b = make(chan bool)
```

Or, more briefly,

```
a := make(chan int) // declare and create channel a
b := make(chan bool)
```

A channel is typed and can carry data of all sorts of type.

Go-routines

A go-routine is a function that is executed independently.

A named function can be started as a go-routine by putting the keyword "go" in front of the call.

```
go quicksort(a,b,c)
```

This call is now executed concurrently with its caller.

A go-routine has its own call stack—no a priori limit to size.

No low-level synchronisation/locking operations are required to constrain the go-routine interactions.

Using Go channels

A "left-arrow" indicates how a channel is being used:

```
c <- 42          // Send value on channel c (blocking)
result = <-c     // Receive value from channel c
<-c              // Receive and discard
```

A function can specify that it only uses a channel in one direction:

```
func f(done chan<- string, job <-chan Job) {
    processNext(<-job)
    done <- "Okay"
}
```

Channels are first-class citizens. They can be created dynamically, be elements of structs, get passed to functions, returned by functions, and so on. Channels themselves can be communicated via channels.

100,001 channels and 100,002 go-routines

```
func main() {
    const n = 100000
    leftmost := make(chan int)
    right := leftmost
    left := leftmost
    for i := 0; i<n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }
    go func() {right <- 42} ()
    fmt.Println(<-leftmost)
}

func f(left, right chan int) {
    left <- 1 + <-right
}
```

This example creates 100,001 go-routines in a communication chain.

The body of `f` says "take the value coming in on channel "`right`", add 1, and place the result on channel "`left`".

"`func() {right <- 42} ()`" is a nameless (and parameter-less) function which places an initial value on channel "`right`".

Buffered channels

A channel communication happens in synchrony between two go-routines. A channel created with "`make(chan myType)`" thus provides a way of enforcing synchronous communication.

However, Go also allows for asynchronous communication via buffered channels:

```
a := make(chan int)       // synchronous
b := make(chan int, 100) // asynchronous, will queue
                          // up to 100 messages
```

Buffered channels

```
const jobs = 1000

func main() {
    ch := make(chan int, 10)
    done := make(chan bool)
    go produce(ch, done)
    go consume(ch, done)
    <-done
    <-done
    fmt.Printf("%d\n", <-ch)
}
```

This example creates a buffered channel for two go-routines, `produce` and `consume`, to use.

The channel `done` is for those two go-routines to signal when they have completed their tasks.

When both have completed, the buffered channel is reused (by the consumer to send some value back to the main routine).

Buffered channels

```
func produce(c chan<- int, done chan<- bool) {
  for i := 0; i < jobs; i++ {
    time.Sleep(time.Duration(rand.Intn(10)) * time.Millisecond)
    c <- i+1
  }
  done <- true
}

func consume(c chan int, done chan<- bool) {
  j := 0
  for i := 0; i < jobs; i++ {
    time.Sleep(time.Duration(rand.Intn(10)) * time.Millisecond)
    j = <-c
  }
  done <- true
  c <- j
}
```

Note how channel directions are indicated.

Go's "select"

The `select` statement is similar to FSP's choice operator.

```
select {
case <communication1>:
    statements1
  :
case <communicationN>:
    statementsN
}
```

Each "communication" involves some channel. These are evaluated, and the "select" blocks until some communication can proceed; then the corresponding "statements" get executed. If several can proceed, select chooses non-deterministically.

Go's "select"

The select statement can be made non-blocking by adding a "default" option:

```
select {
case <communication1>:
    block1
  :
case <communicationN>:
    blockN
default:
    block0
}
```

If none of the channel communications can proceed immediately, rather than block, select will then execute the default block.

Go's "select"

```
func main() {
  ch := make([]chan bool, 3)
  for i := range ch {
    ch[i] = make(chan bool)
  }

  go func() {
    for {
      ch[rand.Intn(3)] <- true
    }
  } ()

  for i := 0; i < 24; i++ {
    var n int
    select {
    case <-ch[0]:
      n = 1
    case <-ch[1]:
      n = 2
    case <-ch[2]:
      n = 3
    }
    fmt.Printf("%d ", n)
  }
  fmt.Println()
}
```

Output may be 3 1 3 3 2 1 1 3 2 2 1 3 1 2 1 3 2 2 3 1 1 1 2 1, say.

Reading

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

M. Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Addison Wesley, 2012.

For a nice introduction, by Rob Pike, to concurrency using Go, see `https://www.youtube.com/watch?v=f6kdp27TYZs`. The daisy chain communication example was taken from Pike's video.