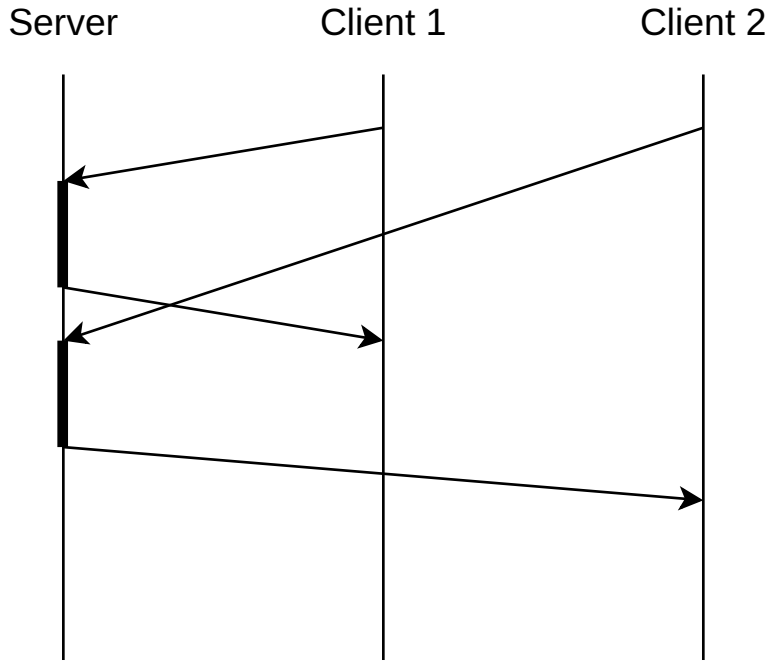
Communicating Processes

Concurrency

Events happen in unpredictable order.

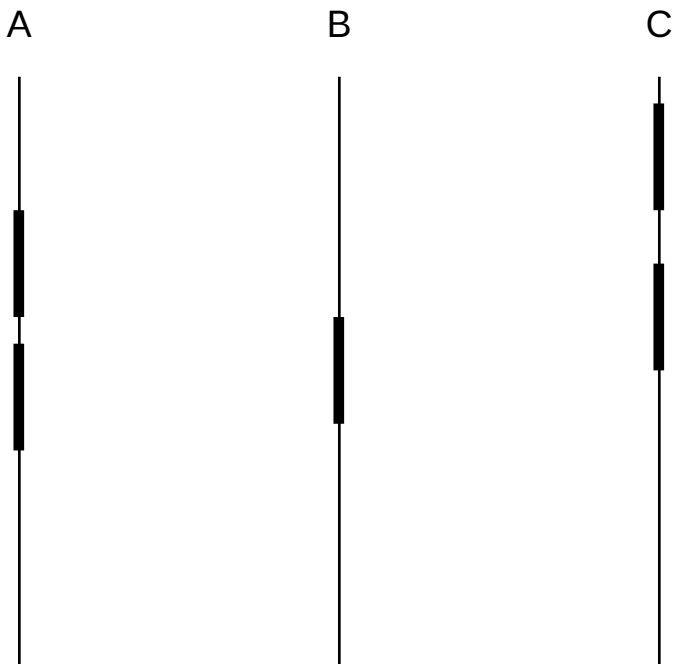


How do we model this fact in programs?

Examples:

- Http Servers
- File Servers
- Chat Serves
- Stream Processors
- Event Queues
- Monitors
- Dashboards
- "Internet of Things"

Processes



> Furthermore, we have stipulated that the processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other. In particular we disallow any assumption about the relative speeds of the different processes.

Dijkstra, E. W. 1965. Cooperating Sequential Processes.

Go

https://go.dev/

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, 世界")
}
```

Designed by Robert Griesemer, Rob Pike, Ken Thompson. First appeared 2009.

Processes in Go

```
go func() { fmt.Println("A1"); fmt.Println("A2") }()

go func() { fmt.Println("B1"); fmt.Println("B2") }()

go func() { fmt.Println("C1"); fmt.Println("C2") }()
```

demo/processes/main.go

Communication is synchronization

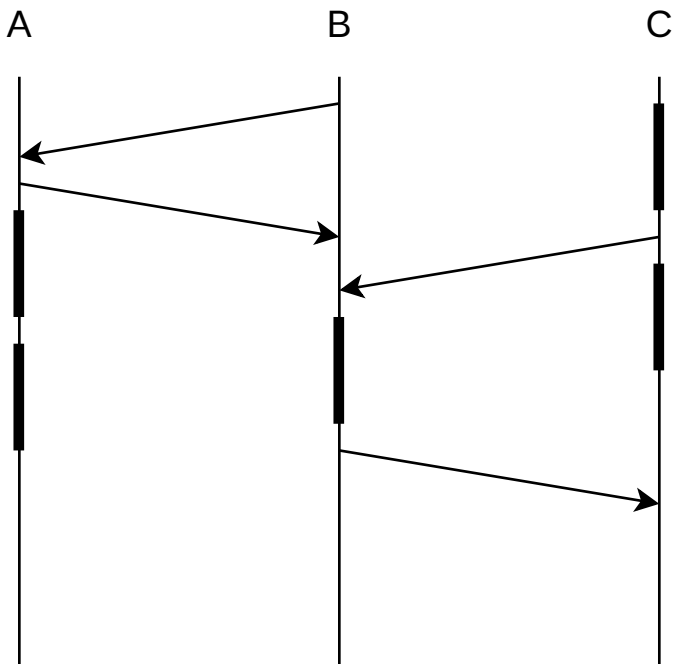
Happens Before:

(1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.

(2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.

(3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two distinct events a and b are said to be concurrent if $a \nrightarrow b$ and $b \nrightarrow a$.



Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system.

Rendezvous Channels

> Such communication occurs when one process names another as destination for output *and* the second process names the first as source for input. In this case, the value to be output is copied from the first process to the second. There is *no* automatic buffering: In general, an input or output command is delayed until the other process is ready with the corresponding output or input.

C. A. R. Hoare. 1978. Communicating sequential processes.

Rendezvous Channels in Go

https://go.dev/ref/spec#Channel_types
https://go.dev/ref/spec#Send_statements
https://go.dev/ref/spec#Receive_operator

```
c := make(chan int)
c <- 1
x := <-c
```

```
c := make(chan int)
go func() { c <- 1 }()
x := <-c
```

demo/channels/main.go

demo/client/main.go

Closing Channels in Go

https://pkg.go.dev/builtin#close

```
func close(c chan<- Type)
```

> The close built-in function closes a channel, which must be either bidirectional or send-only. It should be executed only by the sender, never the receiver, and has the effect of shutting down the channel after the last sent value is received. After the last value has been received from a closed channel c, any receive from c will succeed without blocking, returning the zero value for the channel element. The form
>
> ```
> x, ok := <-c
> ```
>
> will also set ok to false for a closed and empty channel.

demo/pipeline_consumers/main.go

WaitGroups in Go

https://pkg.go.dev/sync#WaitGroup

> A WaitGroup is a counting semaphore typically used to wait for a group of goroutines or tasks to finish.
>
> For example:
>
> ```
> var wg sync.WaitGroup
> wg.Go(task1)
> wg.Go(task2)
> wg.Wait()
> ```
>
> A WaitGroup may also be used for tracking tasks without using Go to start new goroutines by using WaitGroup.Add and WaitGroup.Done.

demo/pipeline_producers/main.go

Guarded Commands

> A guarded command with an input guard is selected for execution only if and when the source named in the input command is ready to execute the corresponding output command. If several input guards of a set of alternatives have ready destinations, only one is selected and the others have n o effect; but the choice between them is arbitrary.

C. A. R. Hoare. 1978. Communicating sequential processes.

Guarded Commands in Go

https://go.dev/ref/spec#Select_statements

> A "select" statement chooses which of a set of possible send or receive operations will proceed. It looks similar to a "switch" statement but with the cases all referring to communication operations.

```
select {
case text := <-input:
    fmt.Println("You entered:", text)
case <-timeout:
    fmt.Println("Timeout!")
}
```

demo/select/main.go

demo/chat_client/main.go

demo/chat_server1/main.go

demo/chat_server2/main.go

Race Condition

Originally, a term coming from hardware.

In Software: when the outcome of a program depends on the timing of processes.

Can be intentional or unintentional.

Examples seen so far: channels and select.

Data Race

In modern programming languages, when two processes modify a shared mutable
reference, the outcome is undefined behavior.

```
x1 = r.read
y1 = x1 + 1
r.write(y1)
```

```
x2 = r.read
y2 = x2 + 1
r.write(y2)
```

Many programming languages provide no guarantee at all about such programs.

demo/races/main.go

```
go run -race
```

Mutexes

> To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store.

E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control.

Mutexes in Go

https://pkg.go.dev/sync#Mutex

> ```
> func (m *Mutex) Lock()
> ```
>
> Lock locks m. If the lock is already in use, the calling goroutine blocks until the mutex is available.
>
> ```
> func (m *Mutex) Unlock()
> ```
>
> Unlock unlocks m. It is a run-time error if m is not locked on entry to Unlock.

demo/races/main.go

demo/chat_sever3/main.go

Deadlocks

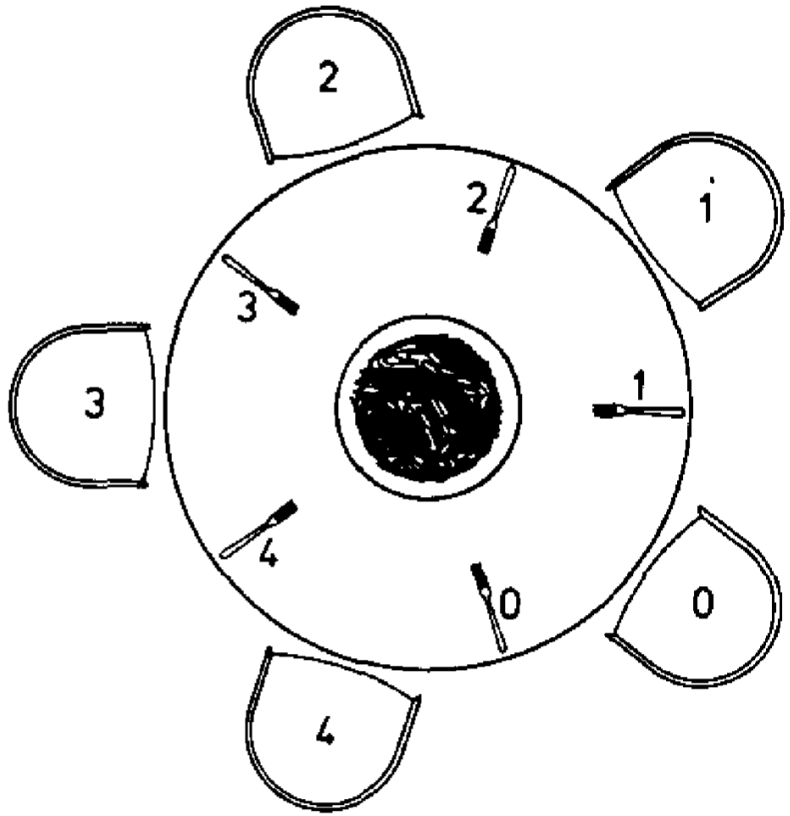
> One of the objectives of recent developments in operating systems − incorporating multi-programming, multiproeessing, etc. − has been to improve the utilization of system resources (and hence reduce the cost to users) by distributing them among many concurrently executing tasks. In any operating system of this type the problem of deadlock must be considered. Requests by separate tasks for resources may possibly be granted in such a sequence that a group of two or more tasks is unable to proceed - each task monopolizing resources and waiting for the release of resources currently held by others in that group.

E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System Deadlocks.

Dining Philosophers

> Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a large bowl of spaghetti, and the table is laid with five forks (see Figure 1). On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he has finished, he puts down both forks, and leaves the room.

Fig. 1.



C. A. R. Hoare. 1978. Communicating sequential processes.

Dining Philosophers in Go

Philosophers are goroutines.

```
go func() {
    fork0.Lock();
    fork1.Lock()
    fmt.Println("eating");
    fork0.Unlock();
    fork1.Unlock()
    wg.Done()
}()
```

Forks are mutexes: picking them up is locking, putting them down is unlocking.

demo/dining/main.go

Summary and Outlook

Communicating processes model concurrency.

Next week: transactional memory.