Message passing: determinism, choice and buffer capacity

Block 1. Determinism, choice and buffer capacity

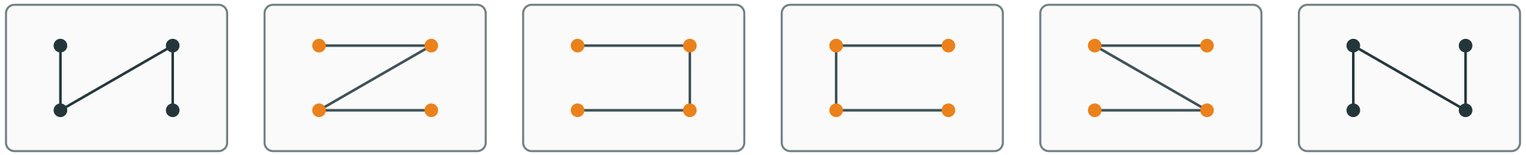
Once processes share nothing, where can the scheduler's choice still reach the result of a program, and what does the capacity of a channel change?

Previously: shared memory and the scheduler (session 7)

```java
// two threads run this at once, from balance = 100
int b = balance;       // both may read 100
balance = b + 50;      // both write 150
```

All six schedules of the four instructions: two end at 200, four at 150. In the interleaving model, four of the six lose a deposit.

1. Threads share memory; the scheduler decides how their steps interleave.
2. Any conflicting, unordered access can change the result.
3. Nothing in the program text marks which accesses those are.



Once processes share nothing, where can the scheduler's choice still reach the result of a program? What does an interleaving then mean, and can a program still deadlock?

The topics of the two blocks and the lab

Block 1	The RC 4000 nucleus (1969); nondeterminism and the absence of sharing; Kahn process networks; contested communications and three resolutions; the RC 4000 through the lens: requests and replies; channels (CSP, occam, Go); buffer capacity.
Block 2	Actors and mailboxes (Hewitt, Agha, Erlang); fairness; the delivery assumption and the boundary with session 9; the cost of message passing, and channels against actors.
The lab	A message-passing library built by the students: a toy RC 4000 nucleus, one queue under two disciplines, the handshake, `select`, and a closing comparison.

PART 1

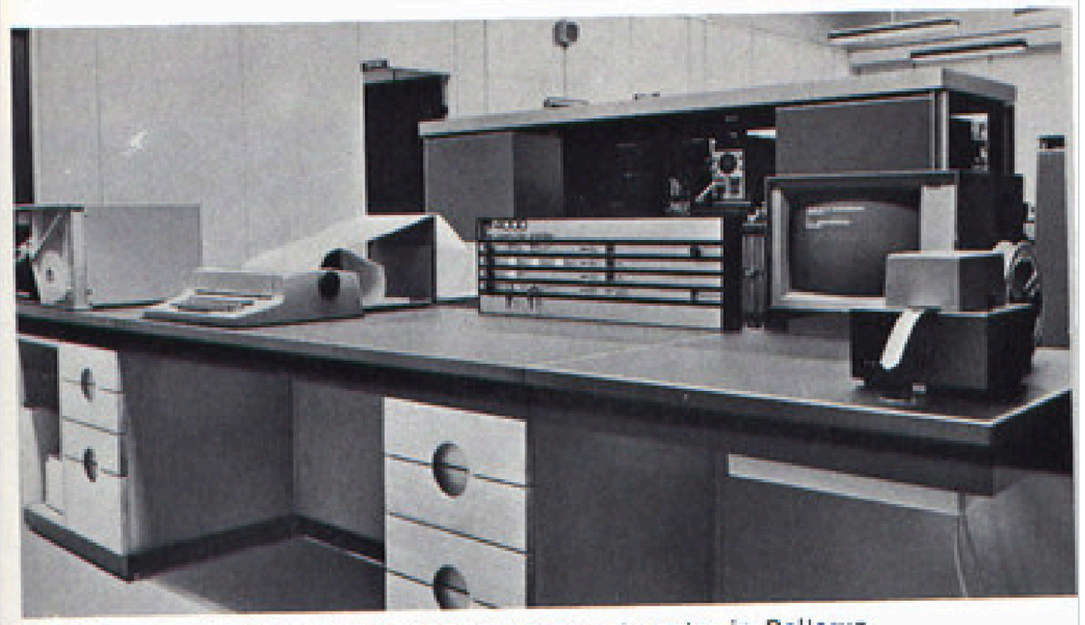
The RC 4000 nucleus, Copenhagen, 1969

the earliest message-based system of the session; messages chosen over semaphores

The RC 4000 and its nucleus

- **1969**: a group led by **Per Brinch Hansen** finished its multiprogramming system (Brinch Hansen, April 1970): not a conventional operating system but a small **nucleus** with **three jobs**, in the paper's words:
  1. the **simulation** of processes;
  2. **communication** among processes;
  3. the **creation, control and removal** of processes.

- **Messages** were its **basic means of communication** between processes; operating systems and user programs ran on top as ordinary processes.



RC 4000 computer system at the RC development center in Ballerup, 10 kilometers west of Copenhagen

RC 4000 computer system at the RC development center in Ballerup, 10 kilometers west of Copenhagen (Regnecentralen, Highlights of RC's History, datamuseum.dk).
Its nucleus: 4,800 words; a send 0.6 ms, a wait 0.4 ms, a full conversation about 2 ms.

- **1967**: Regnecentralen's **RC 4000**, its first prototype monitored an ammonia plant at Puławy in Poland.

Messages instead of semaphores

- Brinch Hansen had learned synchronization from Dijkstra's *Cooperating Sequential Processes*, the manuscript of 1965 that made semaphores widely known; the paper acknowledges it.
- Having learned semaphores, he withheld them from processes and gave them messages.

> **The semaphore concept alone "does not fulfill our requirements of safety and efficiency in a dynamic environment in which some processes may turn out to be black sheep and break the rules of the game" (page 239).**

The RC 4000 returns once more in this block, for *where* its shared memory went (part 2); its full conversation is in the appendix.

PART 2

Nondeterminism and the absence of sharing

interleavings and nondeterminism; processes that share nothing; the shared memory inside the runtime

Determinism, nondeterminism and the data race

Data race (recall)	Two accesses to the same location, at least one of them a write, and nothing ordering them. Locks restore order, but only if every access takes them; two locks taken in opposite orders can deadlock.
Execution and result (recall)	The **execution**, which steps run in which order, can differ from run to run while the **result** stays the same. Unless the execution is named, nondeterminism in this lecture means a difference in the result.
Deterministic program	Every run on the same input data gives the same observable results.
Nondeterministic program	The same input can give different results, depending on choices the program text does not fix, here the scheduler's. A clock or a random draw counts not as input but as a source of nondeterminism.

Nondeterminism in a shared-memory program

The useful way to describe this is not "shared memory is hard".

> In a shared-memory program, **nondeterminism can enter at any conflicting, unordered access to shared data**, and nothing in the program text marks which accesses those are. The **scheduler** makes the choice, everywhere, all the time.

- The statement is made in the interleaving model: every language's memory model allows at least its interleavings, and some allow more.
- It holds in a language whose types do not require shared data to be marked: Java, C and Go; Rust's types do (Java marks it with keywords like synchronized and volatile, not types)
- Hoare, 1978: processors that cooperate by "inspection and updating of a common store" can create severe problems for writing correct programs.

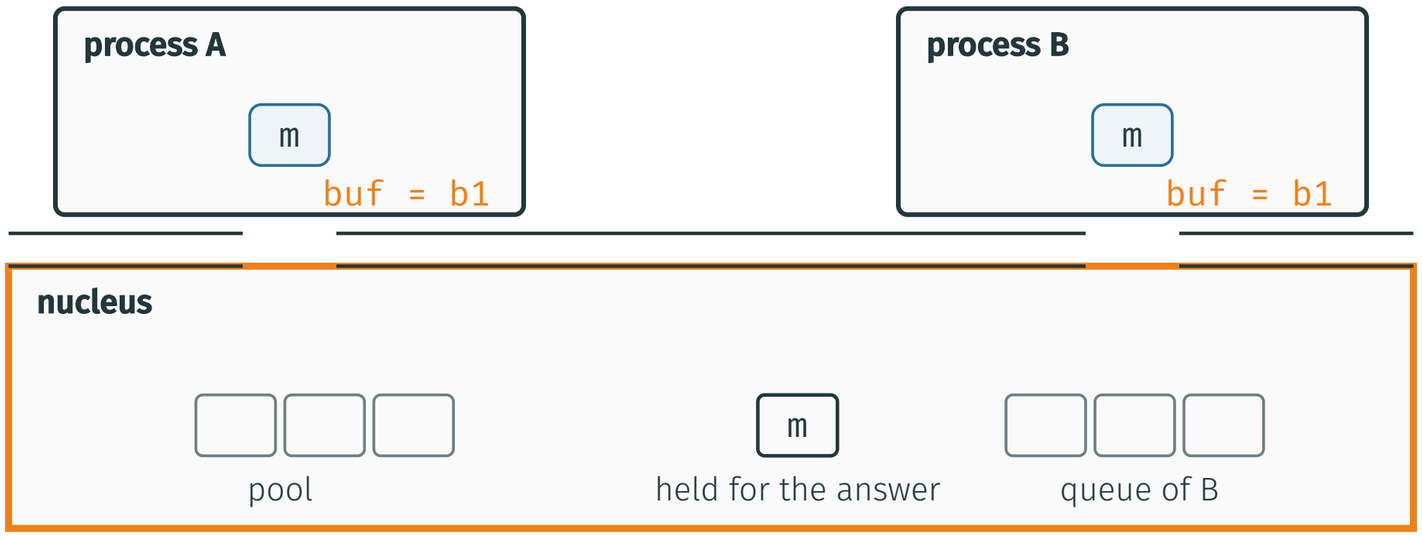
Message passing instead of shared variables

Each process owns its data, and the only way to affect another process is to **send it a value**: processes send and receive messages instead of reading and writing shared variables (Andrews and Schneider, 1983).

> The move **fences nondeterminism into a few visible places**. Inside a process that reads no clock, draws no random numbers, takes no other outside input and uses no construct whose order the language leaves open, code is **sequential and deterministic**. Timing can matter only **where processes meet**, at communications.

- What is sent is a value, not a pointer to memory both sides still use, which Go permits.
- Go's iteration over a map is one construct whose order is left open. (non-determinism on a single process, no sceduler or sharing involved)

The send path of the RC 4000



In the model, m goes from A to B, and A keeps its own copy.

Send message: the nucleus takes a free buffer from the pool.

It links the buffer, b1, at the tail of B's queue, and send message returns its identity to A.

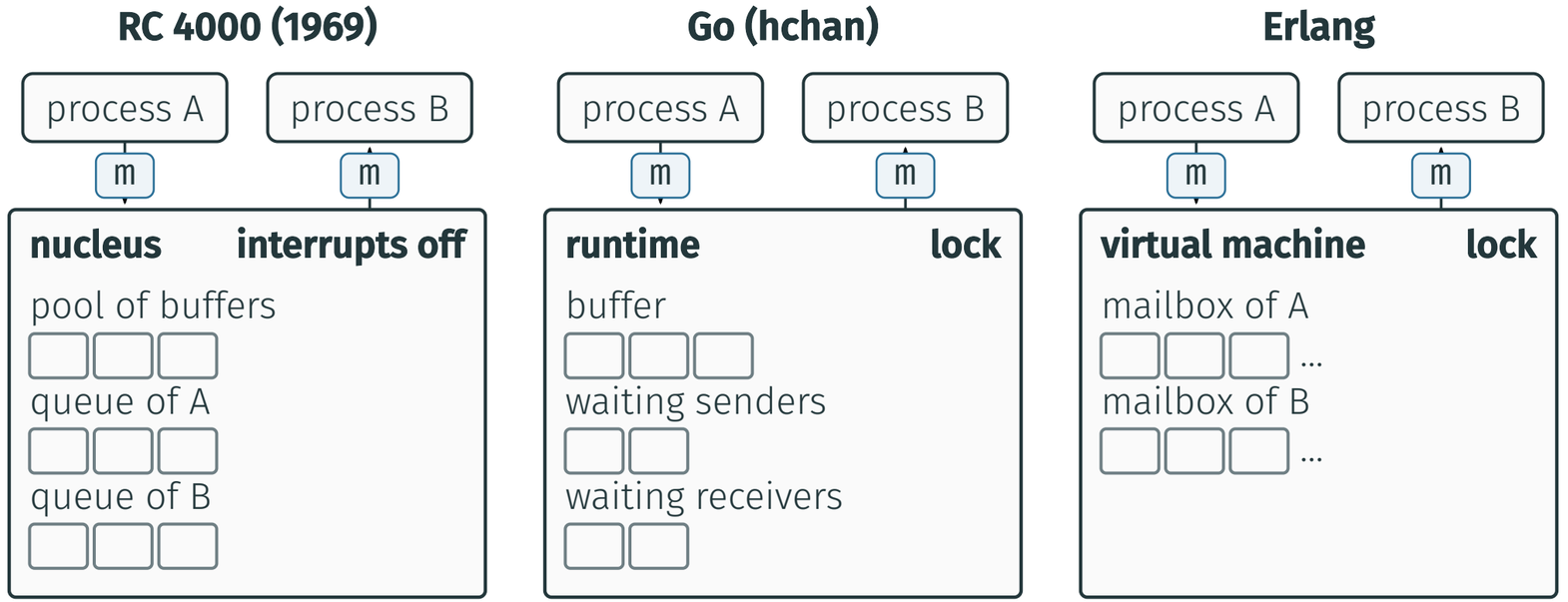
Both primitives went through the nucleus to the pool and the queue: **the shared memory is still there**, in one place. On the single processor the paper implies, a primitive runs with interrupts off, so in effect **one lock** guards that place.

The cut through the floor shows what the model hides: **the nucleus**, with a pool of buffers and one queue per process.

It copies m from A's memory into the buffer.

Wait message: the nucleus takes b1 from B's queue and gives B a second copy of m, with the sender and b1's identity; b1 stays out of the pool, held for the answer.

The shared memory inside the runtime



message passing on one node does not remove shared memory;
it hides it in one trusted place

The RC 4000: the nucleus holds a pool of buffers and one queue per process, the sorting room no process enters; a primitive runs with interrupts off, in effect one lock.

An Erlang mailbox is **a queue per process** in the virtual machine, with no capacity; a lock per mailbox guards its incoming queue.

If sharing is hidden in the runtime, can the result of the program still depend on the scheduler?

A Go channel is a runtime structure, `hchan`, with a buffer, the queue of waiting senders, the queue of waiting receivers and **a lock**: the runtime of go1.27, not the specification, which says nothing of locks.

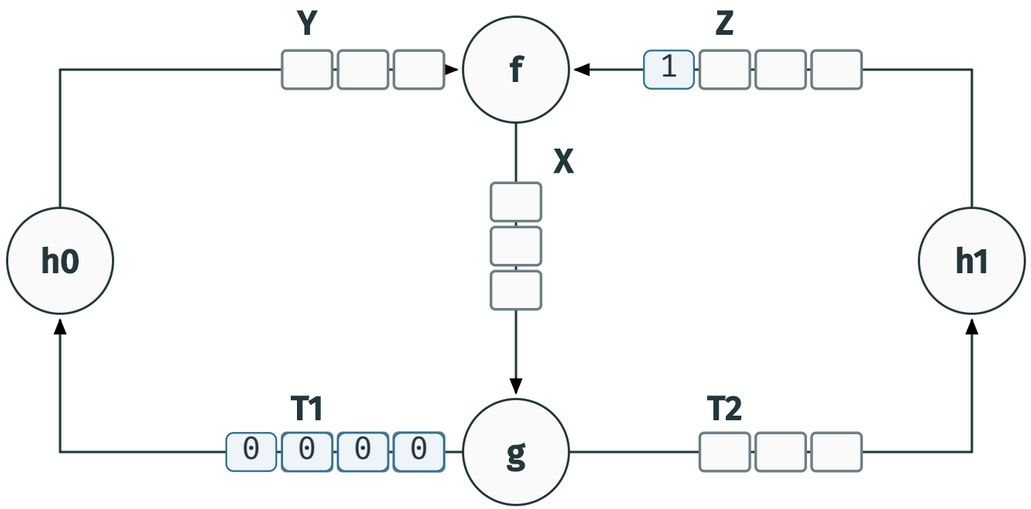
That place **only has to be right once**, as long as messages carry values and not shared pointers.

PART 3

Kahn process networks: determinism without choice

the model of 1974 and its result; two hello worlds; Unix pipelines; the three limits

Kahn's networks (1974)



f reads Y or Z as its own B says, never whichever channel holds a value

Each channel has **exactly one writer** and one reader: X is written by f and read by g, and by nobody else; no channel on the figure has two arrows in.

A read names one channel and **waits until a value is there**: f waits on Y while Z already holds a value.

Kahn's result, determinism. Under any fair schedule, one in which every process that can run eventually does, the network computes the same sequences on its channels: here X carries 0, 1, 0, 1, … in every schedule, and f prints them. The figure shows only the wiring; the determinism is in each process's text, which names the channel it reads next. An unfair schedule computes a prefix of the same sequences.

A write never waits: g's sends on T1 land in the queue however slowly h0 reads, and the queue grows; the channel is **an unbounded queue**.

No choice among inputs anywhere: a process cannot ask whether something is on a channel, and cannot wait on whichever channel is ready first; which channel f reads next is fixed by its own program text.

Where does a Kahn network run on every laptop, and which of its rules does the laptop keep?

Unix pipelines as Kahn networks

A plain shell pipeline is **a Kahn network with bounded pipes**: each pipe has one writer and one reader, and each program is deterministic, reading one input and writing one output. Five command lines, run on the lecture machine:

```
seq 1 100000 | grep 7 | sort -r | head -5
seq 5 | while read x; do sleep 0.$RANDOM; echo $x; done | awk '{print $1*$1}'
printf 'b\nZ\na\nB\nA\n' | LC_ALL=C sort
(dd if=/dev/zero bs=1k count=200 2>/dev/null; echo 'writer done' >&2) | (sleep 3; wc -c)
{ (sleep 0.0$RANDOM; echo hello) & (sleep 0.0$RANDOM; echo world) & wait; } | cat
```

The first printed the same five lines in 53 runs.

The second took between 1.35 and 2.38 seconds and printed `1 4 9 16 25` every time: a slow stage **changes when, not what**.

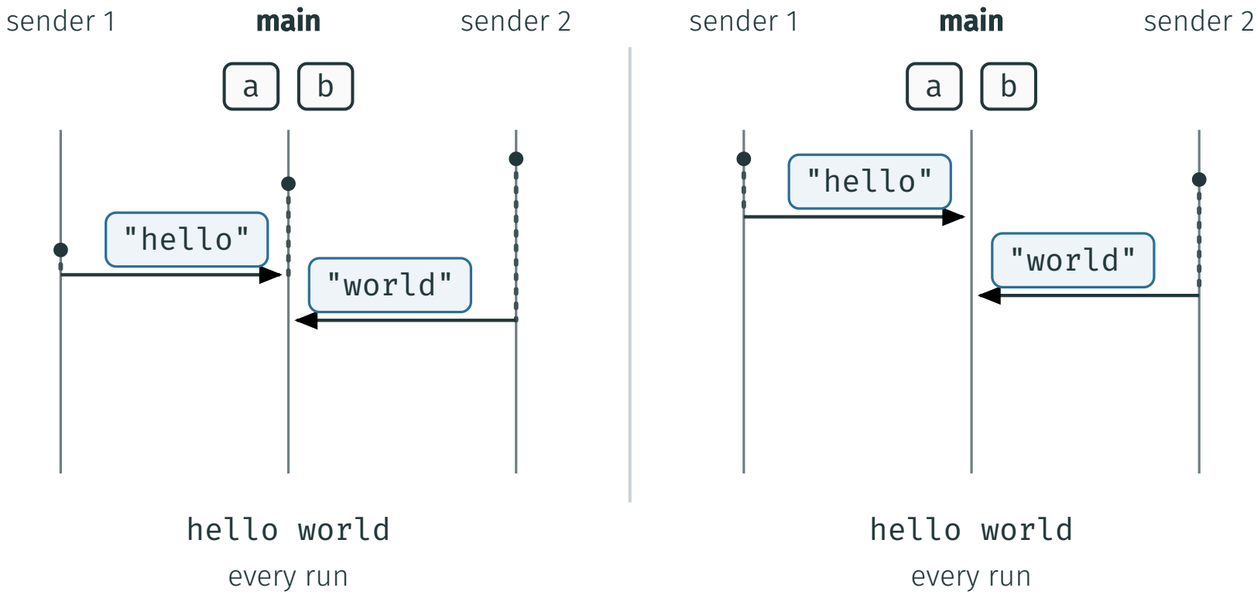
The third, under `LC_ALL=C`, printed `A B Z a b`; under `en_US.UTF-8` it printed `a A b B Z`: **the environment is an input**.

The fourth shows that a pipe is bounded: it holds **65,536 bytes** on this machine, so the writer of 200 KiB finishes only after the reader wakes, at 3.02 seconds.

The last puts two writers on one pipe: in 20 runs, 14 printed `hello world` and 6 `world hello`. It is **a merge**, the first way out of Kahn's conditions.

What becomes of Kahn's determinism when one of its rules breaks: two writers on one pipe here, and two senders on one channel in four lines of Go?

Two senders and one receiver, with and without a contest



with one channel c the same senders print world hello or hello world

Go — two_senders.go

```
c := make(chan string)
go func() { c <- "hello" }()
go func() { c <- "world" }()
fmt.Println(<-c, <-c)
```

Go — two_channels.go

```
a := make(chan string)
b := make(chan string)
go func() { a <- "hello" }()
go func() { b <- "world" }()
fmt.Println(<-a, <-b)
```

One run: world is received first and printed first; sender 1 waits until main receives again. This is the order nearly every run shows.

One channel per sender, main reading `a` then `b`: in one run main waits on `a` while world already waits on `b`; in the other, world waits. The waiting differs, **the output never does**.

Another run: hello first, and **the output changes**. With go1.27.1 at GOMAXPROCS 2 it happened **10 times in 3,000**; the specification leaves the order among senders open.

If the output never differs, what does the fixed order of reading cost, and is that cost necessary?

The trade-off between determinism and responsiveness

> A contested communication **buys responsiveness** and **costs determinism**; no contest keeps determinism and costs waiting.

- Here the contest is a **merge**, two writers on one channel; a **guarded choice** in the program is the other way to let a contest in. Part 4 names both.
- The cost is **a possibility, not a necessity**: a choice whose branches lead to the same result keeps the result deterministic although the execution is not.
- A network without choice waits needlessly only where the channel it reads is empty while another is not.

The limits of a deterministic network

A deterministic network **cannot**:

1. serve whichever client asks first;
2. give up after a timeout;
3. react to whichever of two events happens first.

> Each of the three is **a decision that depends on timing**, and a Kahn process cannot take one: its outputs are a function of **the histories of its inputs**, and a history records the values and their order, not the times at which they arrived. The three limits are **what determinism costs**.

A merge that takes from whichever input has a value is accordingly not a Kahn process (Lee and Parks 1995). Real systems need all three, so **competition** has to be let back in.

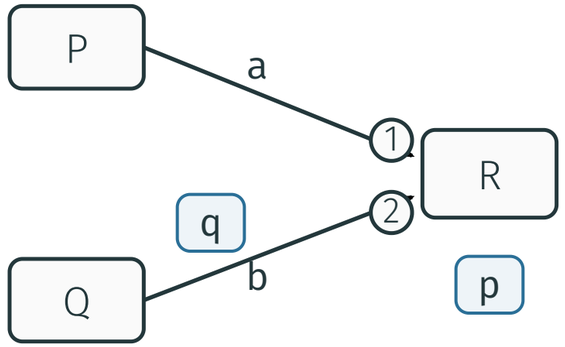
PART 4

Contested communications: three resolutions

no contest, guarded choice and arrival order; where nondeterminism enters once nothing is shared

Resolutions of a contested communication

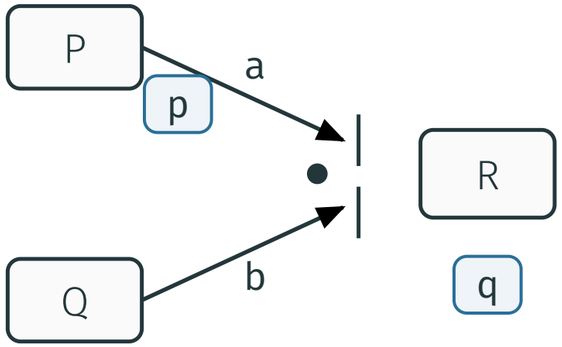
no contest
Kahn networks



one outcome: p, then q

`<-a, then <-b`

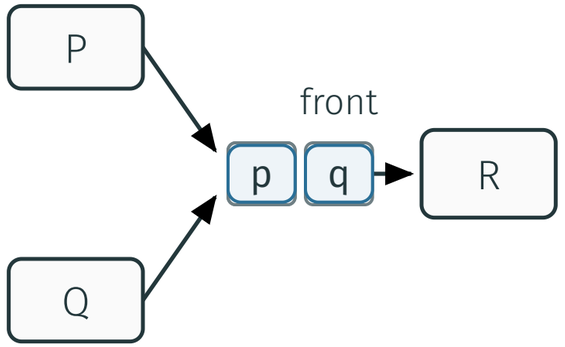
guarded choice
CSP, Go select



q first here; p first in another run

`select on a and b`

arrival order
RC 4000, actors, Erlang



q first here: it arrived first

`receive Msg ->`

No contest. One channel per sender, and R reads a, then b: Kahn's network. q, sent first, waits on b; there is **one outcome**, p then q, at the price of that wait.

Arrival order in a queue. Both messages go into R's queue, and R takes them from the front: the RC 4000, actors, Erlang. q arrived first and is served first; another run can bring p first: two outcomes.

A guarded choice in the program. R takes a partner that is ready: Hoare's guarded commands, Go's `select`. Here q is ready first and is taken; where both are ready, Go takes one by a uniform pseudo-random selection, so p can come first: two outcomes.

Where exactly can nondeterminism enter?

Sources of nondeterminism in message passing

Once nothing is shared, nondeterminism can enter at **four communication points**, or inside a process (Lee and Parks's list of 1995 mapped onto them: appendix):

1. many senders into one receiver, a **merge**;
2. many receivers on one channel, a **split**;
3. a **guarded choice** written by the programmer, asking which input is ready;
4. a **test for emptiness**, asking whether a message is there without waiting.

> In a program whose processes share nothing by any other route, whose messages carry values and whose runtime delivers them, nondeterminism can enter only where several parties compete for one communication point, where a process tests whether a communication is possible, or inside a process. A contest allows a different outcome and does not force one.

The RC 4000: the queue for requests, the buffer's identity for replies (appendix). **What does a language built from such meetings look like?**

PART 5

Channels: synchronous communication and guarded choice

CSP (1978), its channel form (1985), occam and Go

Communicating sequential processes (Hoare, 1978)

Hoare proposes **input and output as basic primitives** of programming, and parallel composition of communicating sequential processes as a fundamental way to structure programs. Four of its choices define the language as this story reads it.

1. **No shared variables.** Processes in a parallel command are disjoint.
2. **Named partners.** An input names its source process and an output its destination: `X?(x, y)` in one process and `DIV!(3*a+b, 13)` in the other together act as the assignment `(x, y) := (3*a+b, 13)`.
3. **No buffering.** "There is no automatic buffering." Whichever side is ready first waits, and "such delay is invisible to the delayed process".
4. **An input fails if its source has terminated, and an output fails if its destination has.** The paper's own statement, page 669.

> **Which construct makes this more than single meetings?**

Guarded choice

```
*[ n < 10; producer?x →
       buffer(n) := x; n := n + 1
 □ n > 0;  consumer?more() →
       consumer!buffer(0); shift
 ]
```

Hoare's bounded buffer, sketched: the process offers two communications and takes whichever partner turns up.

A guard may end in an input command, and the guarded branch runs **only if the named source is ready** to output: the producer while the buffer has room, the consumer while it holds stock.

The branch after the arrow runs once the communication has happened: the item stored and counted, or the first item sent and the buffer shifted.

When several guards are ready, **exactly one is chosen, arbitrarily**: the language selects, at this one visible point.

This is the guarded choice in the program: the programmer marks **one visible point**, the language selects among the partners ready there, and everything else stays deterministic.

How do Kahn's networks and CSP compare?

Kahn's networks and CSP compared

Hoare's four differences (1978, §7.7)	On Kahn's side	On CSP's side	Effect on the values a program computes
choice between sources of input	none: which channel a process reads next is fixed by its text and the values it has read	a guarded choice among the partners that are ready	a program with a guarded choice can compute different values on different runs, which a program without choice cannot: which partner is taken changes what is received
buffering of a send	the queue is unbounded; a send never waits	no buffer: sender and receiver meet in a handshake	in a program without choice no value differs; the handshake waits more and can deadlock where the unbounded network runs on. With a choice, capacity also changes which outcomes are possible (part 6)

Kahn's networks and CSP compared

Hoare's four differences (1978, §7.7)	On Kahn's side	On CSP's side	Effect on the values a program computes
fan-out of a value	a value is copied to every reader (a duplicator node in Kahn's paper; "fanned out" in Hoare's account)	one partner per exchange	no value differs; each reader receives the same copy. Several readers on one channel is another thing, the split of part 4
termination of the processes	processes run forever	the parallel command is normally intended to terminate	no value differs; CSP's history is a prefix of Kahn's, unless a process acts on a partner's termination, which can add values the network never produces

Of the four differences, **only the choice between sources of input changes the values** a program computes; the other three change its liveness, not its values, with the one exception the termination row names: how long it waits, whether it deadlocks, where its output stops.

> **What follows for a CSP program without choice?**

Kahn's networks and CSP compared

> A CSP program with deterministic processes, no alternative command whose guards can succeed together, one writer and one reader per channel, and no process acting on a partner's termination computes, wherever it completes, the same values as the Kahn network of the same processes with unbounded channels; it can deadlock where that network runs on. **Choice, not synchrony**, separates the two models in their results.

Deterministic processes: the definition of part 2, the same results on every run of the process alone; no clock and no random draw.

One writer and one reader per channel: the fan-out row, which names the split; one writer per channel excludes the merge as well, so that no two processes compete for one communication.

How did named partners become channels?

No alternative command whose guards can succeed together: the choice row of the table, the one difference that changes the values; a guard made of a boolean alone counts too (1978, page 669).

No process acting on a partner's termination: the termination row; in CSP 1978 an input fails when its source has terminated, and a process that acts on that failure takes a path the network never takes.

From named partners to channels

```
B :: *[ x: integer; P?x → C!x ]
```

1978: B names its partners, P and C.

In 1978 style, B names its partners, P and C. To reuse B between two other processes, the programmer must **rewrite B**.

The handshake is unchanged; **only the naming changed**. occam (1983) and Hoare's 1985 book made **named synchronous channels** the norm, and that channel version is what Go took up.

```
B(in, out) = in?x → out!x → B(in, out)
```

1985: B names **its own two ends**.

With channels, the parameters `in` and `out` are the channels and the right-hand side is B's behavior; whoever assembles the system plugs real channels into them: `B(left, mid) ‖ B(mid, right)` joins two buffers through `mid`.

What does Go's specification say?

Go's rules as the specification states them

- With capacity zero, a channel is unbuffered and "communication succeeds only when **both a sender and receiver are ready**". With capacity *n*, a send proceeds if the buffer is not full and a receive if it is not empty.
- A channel is a first-in-first-out queue for one sender and one receiver; the specification states **no order among concurrent senders**.
- A send on a closed channel **panics**; a send or receive on a `nil` channel blocks forever.
- A `select` chooses among cases that can proceed by "a uniform **pseudo-random** selection". With none ready, it takes `default` if there is one; otherwise it blocks.

Go's `select` allows **send cases** as well as receive cases; Hoare's 1978 language allowed only input guards. That difference matters in session 12.

> **How does the RC 4000's identified reply look in Go?**

The reply channel carried in the request

Go

reply_server.go ↗

```
type req struct {
    n     int
    reply chan<- int // send-only
}

func counter(reqs <-chan req) {
    total := 0
    for r := range reqs { // one request at a time
        total += r.n
        r.reply <- total  // the answer goes back
    }
}

// a client
reply := make(chan int)
reqs <- req{5, reply}
fmt.Println(<-reply)
```

The request carries its reply channel, typed send-only, `chan<- int`.

The answer goes back on the channel the request carried: **the identified reply**.

It sends the request with the channel inside.

The server takes one request at a time from `reqs`: the queue of requests is the nucleus's queue of the server.

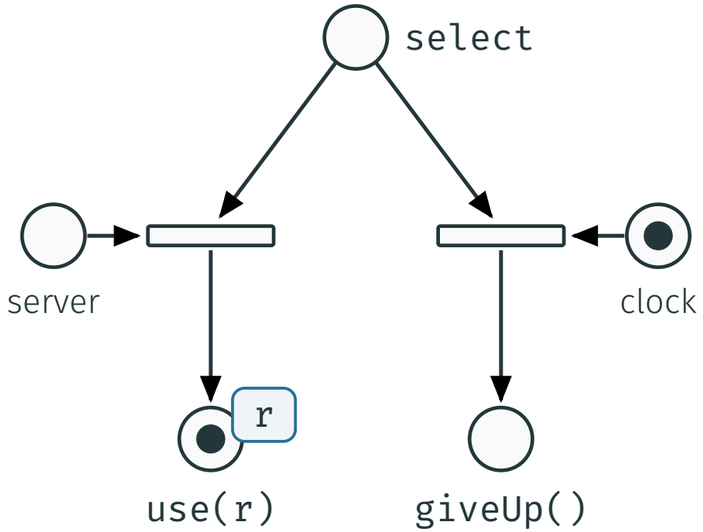
A client makes **a fresh reply channel**, the identity of this conversation.

It receives on its own channel. Only the client reads it, **by convention**: the type stops the server from reading it, and nothing stops another goroutine that holds it. The RC 4000's nucleus checked the identity of a conversation (appendix); here the program keeps it.

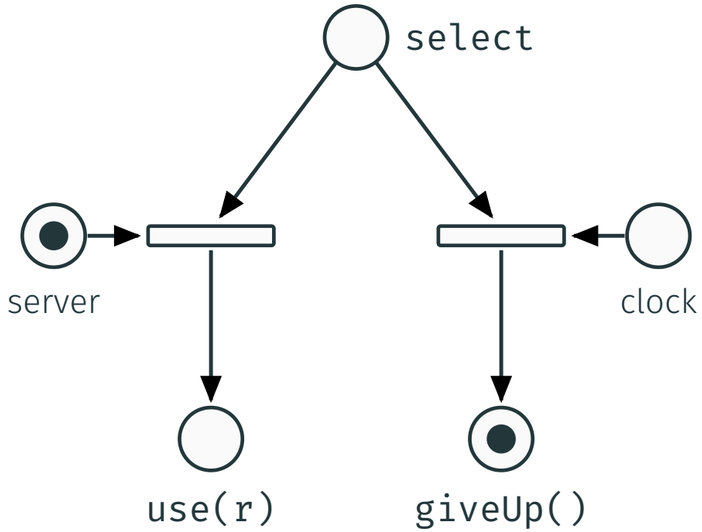
What does a timeout add to a choice?

A timeout as one more branch of the choice

both ready: reply case taken



both ready: timeout case taken



In one run the server is ready first: its bar fires, r crosses it, and the goroutine goes on into `use(r)`.

When both are ready as the `select` runs, it takes one case by **a uniform pseudo-random selection**; the two runs take different branches. In the run that takes the timeout, the server's send on the unbuffered reply channel **stays blocked**, since nothing receives it.

In another run the clock fires first: the timeout branch runs, `giveUp()`.

How does a synchronous world fail?

Deadlock by waiting

Go

send_completion.go ↗

```go
a, b := make(chan int), make(chan int)
go func() { a <- <-b }()
fmt.Println(<-a)
```

```
fatal error: all goroutines are asleep - deadlock!
```

A missing partner: `main` receives on `a`, the goroutine waits on `b` before it would send on `a`, and nobody sends on `b`.

Go

cycle_min.go ↗

```go
toMain, toOther := make(chan int), make(chan int)
go func() { toMain <- 101; fmt.Println(<-toOther) }()
toOther <- 1
fmt.Println(<-toMain)
```

```
fatal error: all goroutines are asleep - deadlock!
```

A cycle: each goroutine sends first and receives afterwards; on unbuffered channels each send waits for a receive the other reaches only after its own send. This cycle deadlocks in every run.

> A deadlock **needs no lock, only waiting**: a lock is one thing a thread can wait for, and any operation that waits can play the part a lock played in session 7.

> **What does the number on a channel change?**

PART 6

Buffer capacity: waiting, deadlock and the sender's knowledge

buffer capacity, independent of how a contest is resolved; the cycle at capacities 0, 1 and 2; capacity and the semaphore count; the two summaries

The capacity of a channel

Capacity	Systems	What a send means
0	CSP (1978, 1985), occam, Go's unbuffered channel	**the receiver has the value**; the smallest cycle of sends already deadlocks
k	Unix pipes (65,536 bytes here); `make(chan T, k)`; the RC 4000's pool, a limit per sending process	the value is queued and the sender goes on; a full pipe or Go channel makes the sender wait, which is **backpressure**
unbounded	Kahn (1974); actors and Erlang's mailboxes (block 2)	a sender never waits and learns nothing of the receiver; no backpressure, and **memory grows**

In a network without choice or test for emptiness and with one writer and one reader per channel, **buffer size does not decide the results**, only waiting, deadlock and what the sender knows; in a program with choice or with a merge, it can also change which outcomes are possible.

Are asynchrony and nondeterminism the same thing?

Asynchrony and nondeterminism

Asynchrony	a send that does not wait for its receiver: a matter of buffering
Nondeterminism	once nothing is shared, a matter of contests, tests for emptiness and processes nondeterminate inside

- The two are **set independently**: an unbounded Kahn network is asynchronous and deterministic; a CSP program with a guarded choice is synchronous and can be nondeterministic.
- Where a contest exists, capacity can still change which outcomes are possible.

> **What does the cycle do at capacity 1?**

The cycle at capacities 0, 1 and 2

Go

cycle.go ↗

```go
// go run cycle.go <capacity> <sends each>
// every receive prints its value; main then
// waits for the goroutine and prints finished
toMain := make(chan int, k)
toOther := make(chan int, k)
go func() {
    for i := 1; i <= sends; i++ { toMain <- 100 + i }
    for range sends { <-toOther }
}()
for i := 1; i <= sends; i++ { toOther <- i }
for range sends { <-toMain }
```

1. capacity 0, one send each
2. capacity 1, one send each
3. capacity 1, two sends each
4. capacity 2, two sends each

```
$ go run cycle.go 0 1
fatal error: all goroutines are asleep
```

```
$ go run cycle.go 1 1
other received 1
main received 101
finished
```

```
$ go run cycle.go 1 2
fatal error: all goroutines are asleep
```

```
$ go run cycle.go 2 2
other received 1
other received 2
main received 101
main received 102
finished
```

What else does capacity change?

Capacity decided **who waits and whether the program deadlocks**; it computed no other value. The missing partner stays at every capacity, since a receive on a channel nobody writes waits forever. A buffer **can break a cycle**; it **cannot supply a missing partner**.

The effects of buffer capacity

1. determinism and the buffer
2. size zero and the meaning of a choice
3. the senders' contest, moved
4. eventual delivery and fairness
5. backpressure against load shedding

Determinism comes from the absence of competition, not from the buffer. A Kahn network gives the same values with any buffer size; with buffers that are too small it can deadlock where unbounded buffers would not. The pipelines of part 3 with one writer per pipe are an instance.

Size zero changes what a choice means. With a handshake, a `select` picks among partners ready at that instant, and the sender knows the value arrived. With buffers, sends finish early, more interleavings become possible, and the sender knows less.

A bounded buffer moves the senders' contest. When it is full, senders compete for the next free slot instead of for the receiver. In Go, a send case in a `select` is ready on a buffered channel when there is room, on an unbuffered one when a receiver is waiting.

Eventual delivery with no bound on delay is where actors get unbounded nondeterminism. The source is the fairness assumption and not the buffer: senders never wait and delivery is assumed eventual, so outcomes can become infinite (block 2).

Backpressure against load shedding. Unbounded buffers give no backpressure: a slow receiver goes unnoticed while its mailbox grows. A bounded buffer that blocks the sender gives backpressure, paid for in blocking bugs; one that drops the message or returns an error sheds load without blocking.

What is capacity's counterpart in session 7?

Buffer capacity and the semaphore count

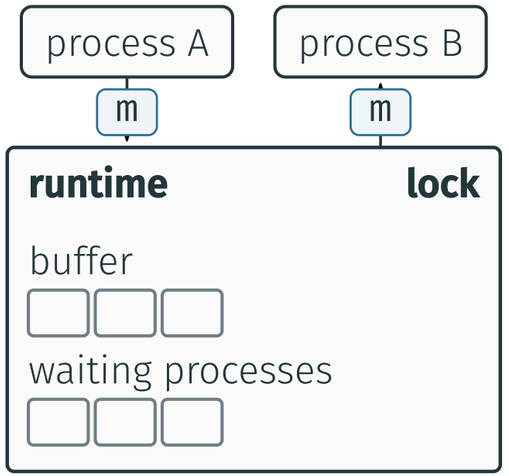
Capacity decides which interleavings of sends and receives can occur; its exact counterpart in session 7 is the **initial count of a counting semaphore**, not the lock. A lock forbids two threads to be in one region at the same time; a capacity forbids one side to run **more than *k* steps ahead** of the other.

Capacity of a channel (session 8)	**Semaphores (session 7)**
0, the handshake	**a rendezvous**: one semaphore signals that a value is there, a second that it was taken, and a third keeps a second sender out; Java's `SynchronousQueue`, which has no internal capacity, "not even a capacity of one"
k	**Dijkstra's bounded buffer** (1965): a semaphore of queuing portions that starts at 0, a semaphore of empty positions that starts at the buffer's size, and a binary semaphore that guards the array; the producer ran at most *k* items ahead
unbounded	Dijkstra's first buffer: the semaphore of queuing portions and the binary semaphore, with no count of empty positions, so the producer never waits

> **What did block 1 settle?**

Summary: the trusted place and the marked points

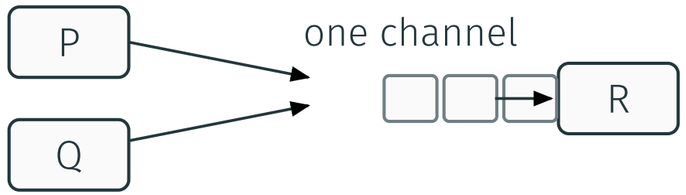
one trusted place



interleavings of accesses, under one lock

a few marked points

a merge



a guarded choice



contests at points the program marks

message passing does not remove interleavings; it moves them,
into one trusted place and onto a few marked points

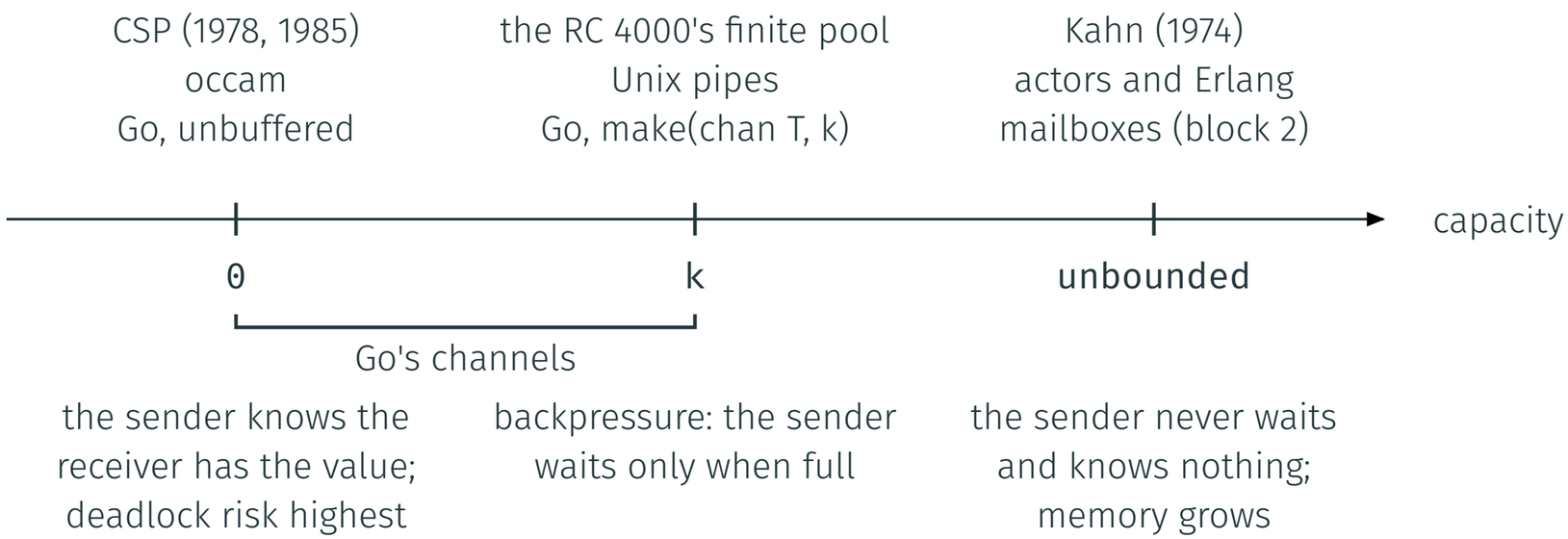
The interleavings of accesses to shared memory go into the **one trusted place underneath**, the buffer and the lock that orders them.

Determinism comes from the **absence of contest**, not from the buffer: no contest leaves one outcome at the price of waiting; a guarded choice or arrival order leaves two, at one visible place.

The scheduler's choices that can reach the results go to the **few marked points** where the program admits a contest: a merge, a guarded choice (CSP, Go), arrival order in a queue (the RC 4000, the actors).

What does capacity decide?

Summary: the axis of buffer capacity



in a Kahn network the values computed are the same everywhere on this axis

At 0, **the handshake**: CSP (1978, 1985), occam, Go's unbuffered channel.

Unbounded: Kahn's networks, and the actors and Erlang mailboxes of block 2.

In a Kahn network the values computed are **the same everywhere on this axis**; capacity decides who waits, when, and how many interleavings remain.

At a finite k: the RC 4000's pool, Unix pipes, `make(chan T, k)`; Go's channels span 0 to k.

At 0 the sender knows the receiver has the value, and the smallest cycle deadlocks; at k backpressure, the sender waiting only when full; unbounded, no waiting, no knowledge, and memory that grows.

Block 2 takes the request half of the RC 4000's conversation to its full form, **the mailbox**: a send that never waits, an arrival order left free, a delivery assumed.

Appendix: the nucleus for the lab, and the slides outside the run

the four primitives; requests and replies; one conversation on the RC 4000; Lee and Parks's list; the reasons for synchronous communication; the count read back

back to the sources of nondeterminism

The communication primitives of the RC 4000 nucleus

```
client:
  buf := send message(server, m)
      -- returns at once
  ...other work...
  answer := wait answer(buf)
```

```
server:
  loop
    (sender, m, buf) := wait message()
    ...serve m...
    send answer(buf, result)
```

The client calls *send message*, naming the server; within its limit of outstanding messages, the call **returns at once** with the identity of the buffer used, and the client goes on with other work.

It serves the request and calls *send answer*, which copies the answer into **the same buffer**; the server continues at once.

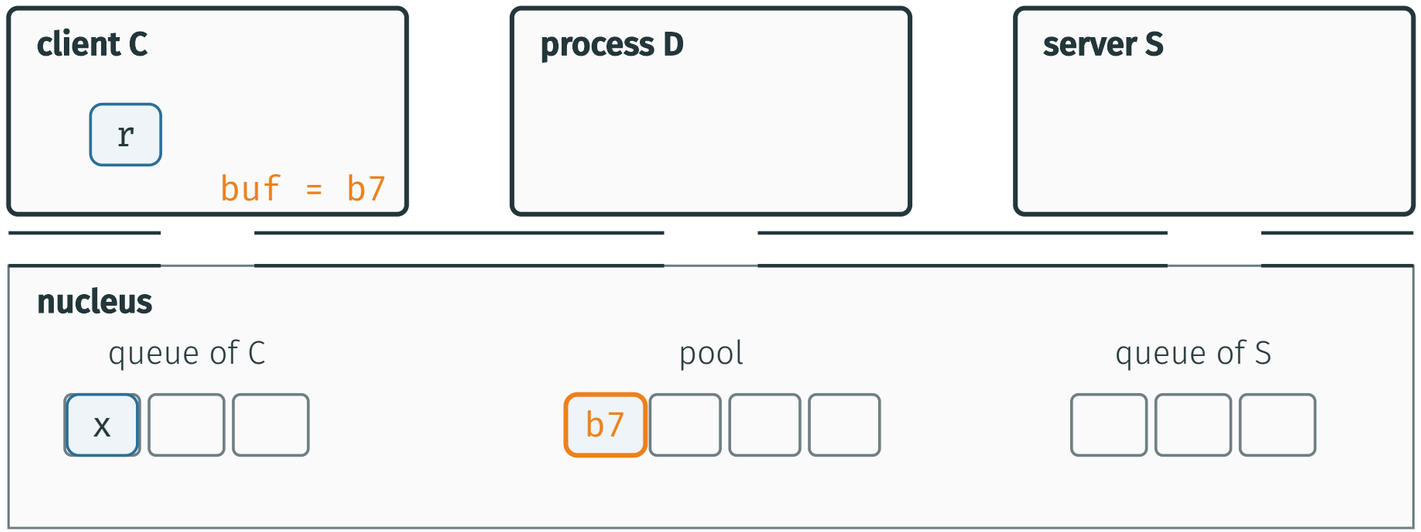
The server waits in *wait message* until a message is in its queue, and receives the sender's name, the message and the buffer.

The client calls *wait answer* on its buffer and waits until the answer is there; the buffer then **goes back to the pool**.

Requests and replies

Requests	A send names a process, which has one queue; messages from every sender land in it in arrival order, and *wait message* serves them first come, first served. When several clients compete for one server, **arrival order in the queue** resolves the contest: the mailbox family's resolution, four years before Hewitt's actors.
Replies	The answer travels back in the same buffer, and the client waits for that buffer by its identity. Nobody competes for a reply, because the client waits for one identified item and not for the next thing in its queue: the no-contest resolution, obtained by **identifying the conversation**.
Today	Go's reply channel carried in the request (part 5); Erlang's reference matched by a selective receive (block 2). The nucleus checks the identity; Go's and Erlang's programs keep it by convention.

One conversation on the RC 4000



C sends m to S: the nucleus takes buffer b7 from the pool and links it behind d; C continues at once, holding b7's identity.

C calls wait answer(b7) before the answer exists: C waits.

Then b7: a copy of m goes into S's memory, and b7 stays in the nucleus, held for the answer. **Arrival order has settled the contest** between D and C.

C's wait answer(b7) takes b7 past x, **by its identity**, which the nucleus checks: a copy of the answer goes into C's memory. **Nobody competed for the reply**.

While C does other work, D sends x to C: another buffer lands in C's queue.

S calls wait message, **first come, first served**: d first, and a copy of d goes into S's memory.

S sends its answer: the nucleus puts it in the same buffer, now b7 · r, which joins C's queue behind x, an unrelated message.

The conversation is over: b7 goes back to the pool, which is **finite**, with a limit per process on the messages it may have outstanding. The primitives in full: appendix, for the lab.

The lecture's sources of nondeterminism and Lee and Parks's list

Lee and Parks (1995), §2.2: the ways to add nondeterminism to a Kahn network	**The lecture's four communication points (part 4)**
a test for emptiness asks whether a value is there without waiting	the test for emptiness is the fourth point
several writers put values on one channel	a merge is the first point: many senders into one receiver
several readers take values from one channel	a split is the second point: many receivers on one channel
a process is nondeterminate inside itself, by a clock or a random draw	nondeterminism inside a process is the case named beside the four points
shared variables let one process change what another reads	the model removes them: nothing is shared, and the memory that remains sits in one trusted place inside the runtime (part 2)
the list has no such item	the **guarded choice** is the third point, the channel family's own addition: a test for emptiness that also performs the communication it waited for, at a point the program marks

Reasons for synchronous communication

Four reasons for the unbuffered send come from different sources; **only the second is Hoare's own**, from the paper of 1978.

1. **Provability.** A communication is one event shared by two processes; with a buffer there is a send event, a receive event, and hidden queue state between them. The algebra of CSP that followed, Hoare's 1985 book, is built on the shared event: a later reading, not a reason given in 1978.

2. **No arbitrary buffer size.** Hoare chose **size zero** and let programmers write buffers as processes, for two reasons: buffering is less realistic on separate processors, and it can be built from the primitives where needed. He concedes the symmetric point, that synchronization could be built from buffered commands (1978, page 675).

Reasons for synchronous communication

3. **Knowledge.** After a synchronous send, the sender knows the receiver has the value, and can make assertions about the receiver's state (Andrews and Schneider, 1983, page 27).

4. **Backpressure.** A fast producer cannot run ahead and fill memory. This is true of any bounded buffer, zero included, so it is **a reason for bounding** rather than for size zero, and it is not in the 1978 paper.

Reason	Provability	No arbitrary buffer size	Knowledge	Backpressure
Source	a later reading	Hoare 1978	Andrews and Schneider 1983	a later reading

Buffer capacity and the semaphore count

> More synchronization always leaves **fewer interleavings**. It leaves fewer observable outcomes **only where a contest exists**, a merge, a split or a choice, or for threads a race the constraint covers; in a Kahn network every value produced is the same at every capacity, a small one only cutting the output short.

The number	**fixed at creation** in both cases; the code that uses it only sends and receives, or acquires and releases
A reading of it	`len(ch)` for a Go channel, `availablePermits()` for a Java semaphore: **out of date by the next statement**, since another thread can change it at once
A decision on it	**a test for emptiness** in the sense of part 4, whose outcome depends on timing; Dijkstra's semaphore has no such operation, P and V being the only way in