Operating Systems, Concurrency and Time

Synchronization and Communication

Johan Lukkien

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Concurrency & communication facilities

- Processes and threads
  - we use the word “process” for both, unless otherwise indicated

- Shared memory, e.g. within kernel, between processes and threads
  - just shared variables: data structures, event flags, spinlocks
  - semaphores, mutexes
  - condition variables with signalling
  - readers/writers locks

- Message passing
  - structured: message queues (channels)
  - streaming: pipes, fifos, sockets

- Signals

Synchronization primitives in systems

- Win 32
  - Semaphore and Mutex (synchronization object)
    - WaitForSingleObject
    - ReleaseSemaphore / ReleaseMutex
  - Critical Section (= mutex between threads of a process)
    - EnterCriticalSection
    - LeaveCriticalSection
  - Event
    - WaitForSingleObject
    - SetEvent
    - ResetEvent
    - PulseEvent (~buffersize=0)
    - Condition Variables
- Posix:
  - Semaphore (counting)
  - Mutex (between threads only; two states)
  - Condition variables with Wait/Signal
- Java object
  - Monitor (“synchronized”)
  - notify; wait (like Posix conditions)

Semaphores (Dijkstra)

- Semaphore *s* is an integer with initial value $s_0 \geq 0$ and *atomic* operations *P(s)* and *V(s).* The effect of these operations is defined as follows:

  P(s): < await(s>0); s := s-1 >

  V(s): < s := s+1 >

  (here we used "< >" to denote atomicity)

- *await:* a statement to indicate suspension
  - implemented, for example, by busy waiting (e.g. spin-locks based)...
  - ... but an OS implementation will not busy-wait
- A thread that executes *P(s)* is suspended until *s>0.*
- A thread executing *V(s)* possibly releases suspended threads

Semaphore invariants

- From the definition we derive four properties (invariants):

$$S0: s \geq 0$$

$$S1: s = s_0 + \#V(s) - \#P(s)$$

(‘#’ denotes the number of executions of an action)

- Combining *S0, S1*:

$$S2: \#P(s) \leq s_0 + \#V(s)$$

- Semaphores may be
  - fair (called *strong*, e.g. FIFO) or
  - unfair (called *weak*).

Counting semaphores (POSIX 1003.1b)

- Naming and creation
  - "name" within kernel, persistent until re-boot, like a filename
    - Posix names: for portability
      - start names with '/' to make it unique system-wide
      - do not use any subsequent '/'
    - for use between processes or between threads
  - also "unnamed" semaphores, for use in shared memory
    - shared memory between processes
  - hence, two interfaces for creation and destruction
    - initialize existing memory structure & OS-level allocation

```
sem_t *sem;
sem = sem_open (name, flags, mode, init_val); /* name is system-wide        */
status = sem_close (sem);                     /* semaphore still reachable */
status = sem_unlink (name);                   /* now it is removed          */

status = sem_init (sem, pshared, init_val);  /* memory space for sem must be defined, e.g.
                                                through shm or malloc */
status = sem_destroy (sem);                  /* pshared: whether multiple processes
                                              * access sem; should be true */
```

Semaphore operations

- Basic interface, designed for speed

- Obtaining the value is tricky
  - value is unstable
  - negative value: interpret as number of waiters (length of queue)

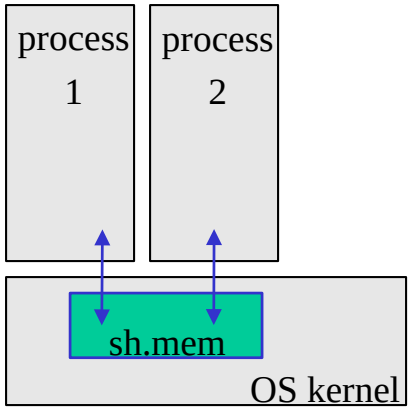
P(sem)

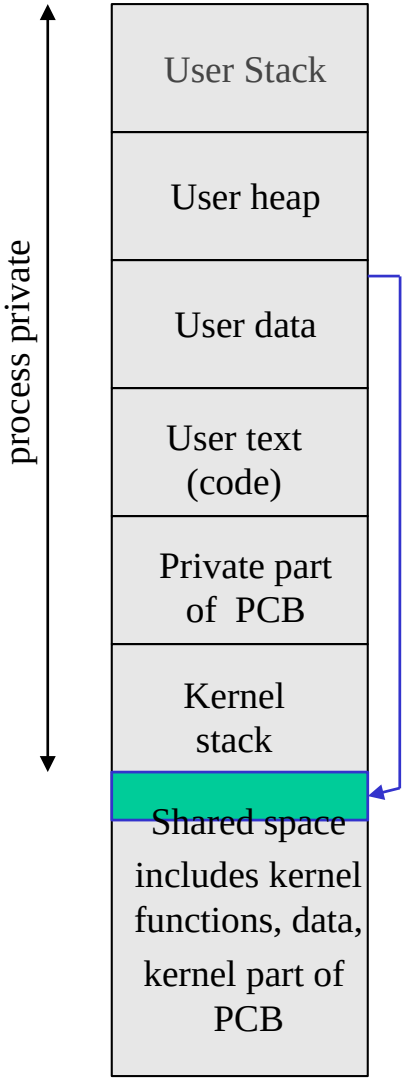
```
status = sem_wait (sem);
status = sem_trywait (sem);          /* returns error (EBUSY?) if sem == 0        */
status = sem_post (sem);
status = sem_getvalue (sem, &val);  /* current value
                                     * when negative: absolute value = # waiters */
```

V(sem)

Shared memory between processes

- A memory segment is reserved in the kernel memory
- It is added to the space addressable in user mode
- Since it is in the shared space, it is available to other processes
- *No kernel involvement is required upon subsequent reading and writing*





POSIX: Shared Memory API

int shmget(key_t *key*,size_t *size*, int *shmflg*);

/* key: an identifier, for reference
 * size: in #bytes
 * shmflg: indicates how to open, e.g. readonly, create if does not exist
 * shmget() returns an id for manipulation
 */

void *shmat(int *shmid*, const void **shmaddr*, int *shmflg*);

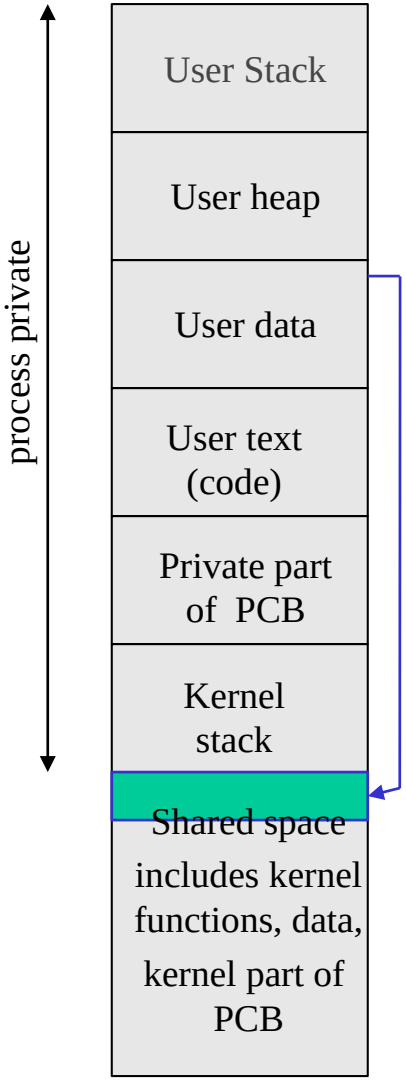
/* shmid: from shmget()
 * shmaddr: place in process space where shared mem should be referenced
 * shmflg: detail and steer the operation
 * shmat() returns a pointer to a shared byte array of size bytes (see shmget())
 */

int shmdt(const void *shmaddr);

/* shmaddr: shared memory to detach, obtained from shmat()
 * shmdt returns just error condition
 */

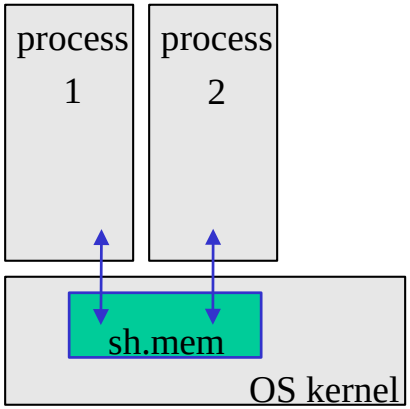
int shmctl(int *shmid*, int *cmd*, struct shmid_ds **buf*);

/* control operations, cmd = IPC_RMID removes the shared memory */



Example

- from Dave Marshall, http://www.cs.cf.ac.uk/Dave/C/node27.html



- Server
  - creates shared memory
  - writes ‘ab...z’ into it
  - (busy) waits until ‘a’ is replaced by ‘*’

- Client
  - attaches to (not: creates) the created shared memory segment
  - reads and prints the content
  - puts a ‘*’ at the expected place

Client

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* chosen in server */

    /* Locate the segment */
    if ((shmid = shmget(key,
            SHMSZ, 0666)) < 0) {
        perror("shmget"); exit(1);
    }

    /* attach the segment to our data space */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1);
    }

    /* Now read what the server put in the memory. */
    for (s = shm; *s != NULL; s++) putchar(*s);
    putchar('\n');

    /* Change the first character of the segment to '*' */
    *shm = '*';

    exit(0);
}
```

Server

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* arbitrary name */
    /* Create the segment */
    if ((shmid = shmget(key, SHMSZ,
                IPC_CREAT | 0666)) < 0) {
        perror("shmget"); exit(1);
    }

    /* Now we attach the segment to our data space */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1);
    }

    /* Now put some things into the memory for the
     * other process to read.
     */
    s = shm;
    for (c = 'a'; c <= 'z'; c++) *s++ = c;
    *s = NULL;

    /* wait until the other process changes the first
     * character of our memory to '*'
     */
    while (*shm != '*') /* sleep(1) */;

    exit(0);
}
```

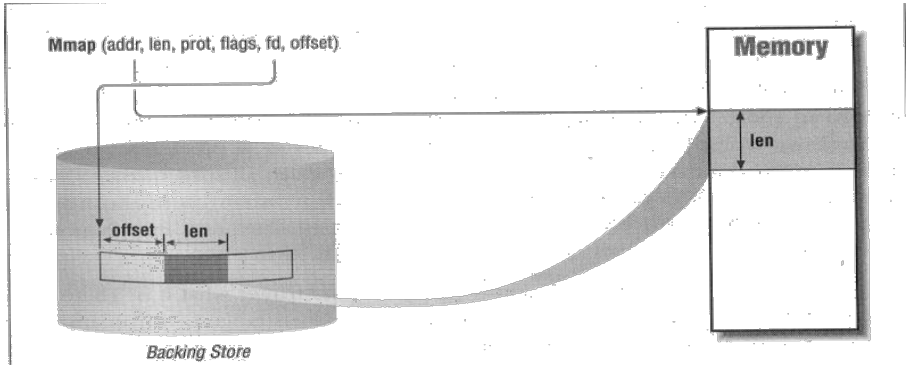
Output

```
user@ubuntu8041:~/2IN05$ ./SHMserver
&
[3] 19817
user@ubuntu8041:~/2IN05$ ./SHMclient
abcdefghijklmnopqrstuvwxyz
user@ubuntu8041:~/2IN05$ ./SHMclient
*bcdefghijklmnopqrstuvwxyz
[3]-  Done                    ./SHMserver
user@ubuntu8041:~/2IN05$ ./SHMclient
*bcdefghijklmnopqrstuvwxyz
user@ubuntu8041:~/2IN05$
```

- Notes:
  - busy waiting is not an acceptable synchronization method at the level of the OS
    - it uses the entire timeslice and destroys performance
    - the *sleep()* avoids continuous checking at the expense of a one second delay
  - instead, synchronization methods provided by the OS should be used
    - message passing, or one of the methods discussed later
  - since the shared memory is not removed it remains after termination of server and client
    - to remove: add *shmctl (shmid, IPC_RMID, NULL)*
  - there is a newer version of these primitives

Shared memory interface (1003.1d)

- Naming is similar to OS-level names for semaphores
  - new namespace within kernel
  - persistent until re-boot
- Two-step approach
  - handle is a file descriptor
  - this file is mapped into memory




```
int fdes;

fdes = shm_open (name, flag, mode); /* file descriptor with limited functionality */
status = ftruncate (fdes, totalsize);     /* set the size or resize it                */
status = shm_close (fdes);                /* memory object remains there              */
status = shm_unlink (name);               /* destroy                                  */

shm_area = mmap (WhereIWantIt /* 0 */, len, protection, flags, fd, offset /* 0 */);
                                  /* finally gives a pointer to the shared memory */
```

The use of semaphores

- Ordering of actions of *different* threads
  - producer-consumer problems
  - exclusion problems (resource reservation)

- Method:
  - *'P(s); A'* is a fragment in one thread (*P* before *A*)
  - *'B; V(s)'* is a fragment in another one (*V* after *B*)
  - then
    - $0 \leq \#P(s) - \#A \leq 1$
    - $0 \leq \#B - \#V(s) \leq 1$
  - hence, using *S4*
    - $\#A \leq \#P(s) \leq s_0 + \#V(s) \leq s_0 + \#B$

A producer/consumer problem

$P_X =$
$\quad x := 0;$
$\quad$ **while** *true* **do**
$\quad\quad$ **A**: $x := x+1$
$\quad$ **od**

$\|$

$P_Y =$
$\quad y := 0;$
$\quad$ **while** *true* **do**
$\quad\quad$ **B**: $y := y+1$
$\quad$ **od**

Synchronize P_X and P_Y such that invariant

$I0: x \leq y \qquad (= \#A \leq \#B)$

is maintained. (x counts consumed items and y produced items)

Solution: let A be *preceded by P(s)* and B be *followed by V(s).*
Choose $s_0 = 0$.

More restrictions

Suppose that we also want:

$I3: y \leq x+10$, i.e., $\#B \leq \#A+10$

Introduce a new semaphore t. Let A be followed by $V(t)$ and B be preceded by $P(t)$. Then,

$\#B \leq \#P(t) \leq t_0 + \#V(t) \leq t_0 + \#A$

Choose $t_0 = 10$.

```
P_X =
    x := 0;
    while true do
        P(s); A: x := x+1; V(t)
    od
```

||

```
P_Y =
    y := 0;
    while true do
        P(t); B: y := y+1; V(s)
    od
```

And more...

Suppose that instead of *I0* we want

$I4: 2x \leq y$, i.e., $2\#A \leq \#B$

Let *A* be preceded by *two* times *P(s)* (denoted as $P(s)^2$)

Then ,

$2\#A \leq \#P(s)$

hence,

$2\#A \leq \#P(s) \leq s_0 + \#V(s) \leq s_0 + \#B$

etc....

Exercises

A.1 Given are *N* threads of the form

$$\boxed{Pr_{(n,\ 0 \le n < N)} = \textbf{while } \textit{true} \textbf{ do } X(n) \textbf{ od}}$$

Here, *X(n)* is a non-atomic program section that must be executed under exclusion. In addition, synchronize this system such that:

a. the sections are executed one after the other, in order:

X(0); X(1); X(2); ....; X(N-1); X(0)...

b. *X(i)* is executed at least as often as *X(i+1)*, for $0 \le i < N-1$.

Exercises

A.2 Given is a collection of threads using system procedures *A0* and *A1*. Synchronize the execution of these procedures such that exclusion is provided and that one execution of *A0* and two executions of *A1* alternate:

A0; A1; A1; A0; A1; A1 …

- Is there any danger of deadlock?
- What about the fairness?

Exercises

A.3 Consider the parallel execution of the three program fragments below.

while *true* **do A**: $x := x+2$ **od**
while *true* **do B**: $y := y-1$ **od**
while *true* **do C**: $x := x-1$; **D**: $y := y+2$ **od**

Initially, $x = y = 0$

Synchronize the system in order to maintain

$I0: 0 \leq y$

$I1: x \leq 10$

Can you give an argument for absence of deadlock? Which additional restrictions might cause deadlock?

Exercises

A.4 A collection of threads uses a collection of *K* resources. For each resource there is an associated data structure, recorded in an array.

The threads repeatedly reserve and release resources using procedures *Reserve()* and *Release(i).* Through a call of *i := Reserve(),* variable *i* is assigned the index of a free resource which is then claimed. This resource is subsequently released through *Release(i).*

Write these two functions. Take care of exclusion on the array.

```
var Res: array [0..K-1] of
        record avail: bool;
                { other variables }
        end
```

```
Func Reserve (): int
Proc Release (i: int)
```

Avoiding deadlock

- Let critical sections terminate
  - in principle, no *P* operations between *P(m)...V(m)*
- Use a fixed order in *P*-operations on semaphores
  - *P(m);P(n); ....* in one thread may deadlock with *P(n);P(m);...* in another thread
  - in fact: satisfy the synchronization conditions in a fixed order
- Beware of greedy consumers
  - Let $P(a)^k$ be an indivisable operation when there is a danger of deadlock
- Use concurrency control protocol with

In general: avoid cyclic waiting!

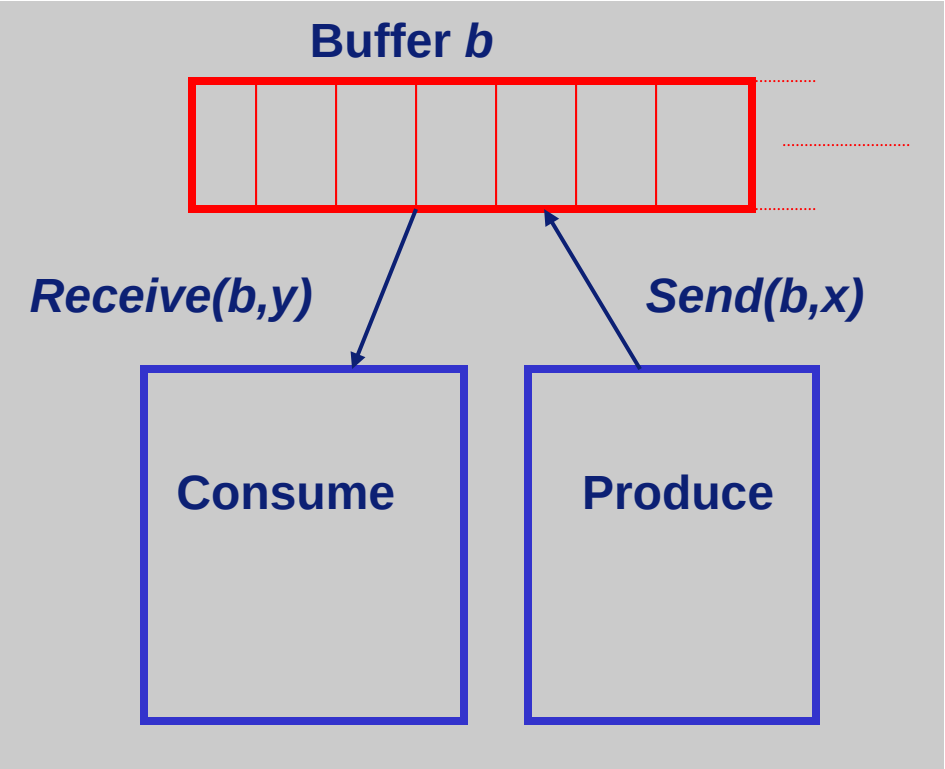
(Un)bounded buffer

Specification:

1. Sequence of values received equals sequence of values sent.

2. No receive before send.

3. For the bounded buffer: number of sends cannot exceed number of receives by more than a given positive constant *N*.



Design

- Data structure supporting FIFO: queue *q*, with operations *PUT(q,x)* and *GET(q,y)*
  - Introduce variable *q* of type queue.

- Exclusive access is required since *PUT* and *GET* are not atomic.
  - Introduce semaphore m, $m_0 = 1$.

- The second requirement translates into

  $$\#GET(q,...) \leq \#PUT(q,...)$$

  - Introduce semaphores t, $t_0 = 0$.

- The third requirement translates into

  $$\#PUT(q,...) \leq \#GET(q,...) + N$$

  - Introduce semaphore s, $s_0 = N$.

First solution

```
type buffer =
record   q: queue of elem;
         s, t, m: Semaphore
end;
```

```
proc Send (var b: buffer; x: elem) =
|[ with b do
       P(s); P(m); PUT(q,x); V(m); V(t)
   od
]|
```

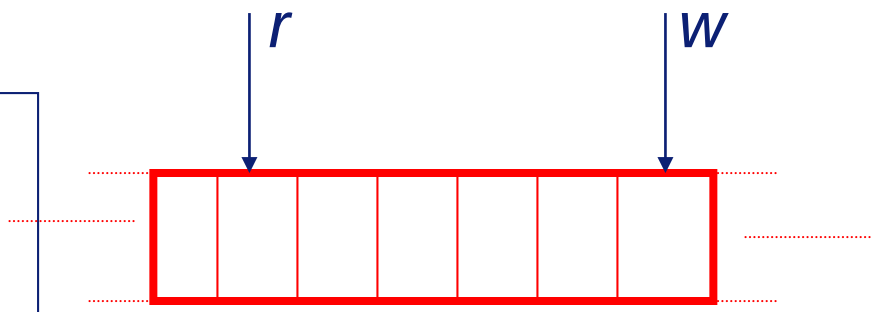
```
proc Receive (var b: buffer; var y: elem) =
|[ with b do
       P(t); P(m); GET(q,x); V(m); V(s)
   od
]|
```

Notice the order of the *P*-operations: critical sections should always terminate

Using arrays

Consider an infinite array as an implementation of a queue. Variables *r* and *w* denote read- and write positions respectively (initially 0).

```
type queue =
record   b: array of elem;
         r, w: int
end;
```




```
proc PUT (var q: queue; x: elem) =
|[  with q do
        { w = #PUT(q,…) }
        b[w] := x; w := w+1
  od ]|
```

```
proc GET (var q: queue; var y: elem) =
|[  with q do
        { r = #GET(q,…) }
        y := b[r]; r := r+1
  od ]|
```

Property of this implementation

- When this array is used in the bounded buffer
  - the array is never accessed by both producer and consumer at the same place,
  - and never more than *N-1* apart:
    - when writer is at "*b[w] := x*" and reader is at "*y := b[r]*"
      then $0 < w-r < N$

- Hence,
  - Semaphore *m* for exclusion is not needed!
  - An array of size *N*, used in a circular manner suffices.

Putting it together

```
type buffer =
record   q: queue of elem;
         s, t: Semaphore
end;
```

```
type queue (elem) =
record   b: array [0..N) of elem;
         r, w: int
end;
```

```
proc Send (var b: buffer; x: elem) =
|[ with b, q do P(s);  b[w] := x; w := (w+1) mod N;  V(t) od ]|
```

```
proc Receive (var b: buffer; var  y: elem) =
|[ with b, q do P(t);  y := b[r]; r := (r+1) mod N;  V(s) od ]|
```

Exercises

B.1 Suppose that a bounded buffer is to be shared by two producers, i.e., two different processes write in it. What must be changed?

B.2 Two consumers share a bounded buffer. The first consumer always takes 3 portions before it can do its work and the second always takes 4. Solve this problem (assuming first- come-first- serve) and answer the following questions:

- Is waiting minimal? If not, can you imagine a situation that leads to a deadlock?
- Does your solution work for a circular buffer of size 2?

Now make a general routine to retrieve *n* messages.

Specialize this solution for the case of a 1-place buffer.

Exercises

B.3 *N* producers produce messages for one consumer. The messages must be handled exclusively, one by one. Producer *i* waits until the consumer has handled its message.

a. Write programs for producers and consumer.

b. Specialize your solution for the case of a buffer with just one single place.

B.4 Consider two threads. One thread produces a whole video frame per cycle, the other consumes the frame sample by sample. There are *m* samples per frame. We have a two place buffer for the frames. The producer can only produce a frame when there is a place is available. Give a properly synchronized implementation of the two threads.

Condition synchronization

- Semaphores:
  - good for problems that require just counting
- Not expressive enough
  - some problems lead to many semaphores
  - synchronize on some boolean expression in the program variables
- Condition synchronization principle:
  - At places where the truth of a condition is required: check and block
  - At places where a condition *may have become true*: signal waiters

Condition variables

- **var** *c: Condition;*
- 4 basic operations:
  - *Wait(..., c)* — suspend execution of caller
  - *Signal(c)* — free *one* thread suspended on *Wait(c)* (void if there is none)
  - *Sigall(c)* — free *all* threads suspended on *Wait(c)*
  - *Empty(c)* — "there is no thread suspended on *Wait(c)*" (boolean function)
- Many extra operations in specific implementations.

"*A condition variable is a memory-less semaphore*"

Example

- Maintain $x \geq 0$ in a program with arbitrary assignments to x

```
var cv: condition;
      m: Semaphore (initially 1)
|[ ......                                  |[ .....
  P(m);                                      P(m);
  while x<10 do
    V(m); { Beware.... }
    Wait (cv);
    P(m)
  od;
  { x≥10 }
  x := x-10;                  ||            x := x+100;
                                            Sigall (cv);
  V(m);                                     V(m);
]|                                          ]|
```

Combine condition variable & semaphore

- Need to combine $\boxed{V(m);\ Wait(cv);\ P(m)}$ atomically: *Wait(m,cv)*

```
var cv: condition; m: Semaphore (initially 1)

|[ ......                                  |[ .....
   P(m);                                     P(m);
   while x<10 do                             x := x+100;
     Wait (m, cv);            ||            Sigall (cv);
   od;                                       V(m);
   { x≥10 }                                  .....
   x := x-10;                             ]|
   V(m);
]|
```

Program structure

- Structure program in condition critical sections
  - establish truth of condition necessary for executing the section
    - use a repetition, because there may be competing threads
  - decide carefully which variables must be protected together
    - usually, these occur together in at least one condition
  - access to these variables only within critical sections
    - reads in other parts are harmless but unstable
- Section structure:
  - *lock;*
  - **while** *not condition* **do** *wait* **od**;
  - *critical section;*
  - *possible signals;*
  - *unlock;*

```
var cv: condition; m: Semaphore (initially 1)

|| ......                          || ....
   P(m);                              P(m);
   while x<10 do                      x := x+100;
     Wait (m, cv);          ||       Sigall (cv);
   od;                                V(m);
   { x≥10 }                           .....
   x := x-10;                      ||
   V(m);
||
```

Condition variables

- Associated with a condition
- With associated semaphore or other exclusion mechanism
  - can have several condition variables per semaphore for more accurate signalling
  - example: Java, POSIX
- Sometimes extended with timeout mechanism
- Must only be signalled *within* critical sections

- Choice of *Signal* or *Sigall*
  - efficiency, careful analysis
  - strategies:
    - *Signal* at least one condition variable satisfying:
      - associated condition is true and there are waiters on it
    - *Sigall* all condition variables of which the associated condition may have changed from *false* into *true.*

Exercises

V.1 Given is a collection of threads that modify shared variable *s* through statements of the form

$$s := s+E$$

where *E* is the outcome of an expression. It is given that *s* is initially 0.

The program must maintain: $s \geq 0$

a. Make a data structure containing both *s* and the required variables for exclusion.

   type *s_rec* = **record** *sval: int; ....* **end;**

   Replace *s := s+E* by a conditional critical section.

b. Make a function to update *s* so that the above statements can be replaced by a call to this function.

   func *Update_s (***var** *sr: s_rec; E: int)*

V.2 Can you give an implementation of general semaphores using condition variables? Try to avoid "*Sigall*". Can you make the implementation fair?

Exercises

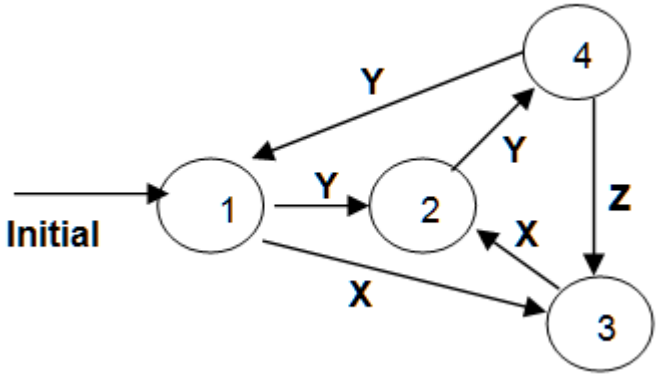
V.3 Redo exercise B.2 but now only with condition variables. (Restrict yourself to the relevant parts.)

Exe

- **V.4** There are three groups of threads, corresponding to the following three procedures.

var *st: int /* initially 1 */*

proc *A* = **|[while** *true* **do** *X* **od]|;**
proc *B* = **|[while** *true* **do** *Y* **od]|;**
proc *C* = **|[while** *true* **do** *Z* **od]|;**



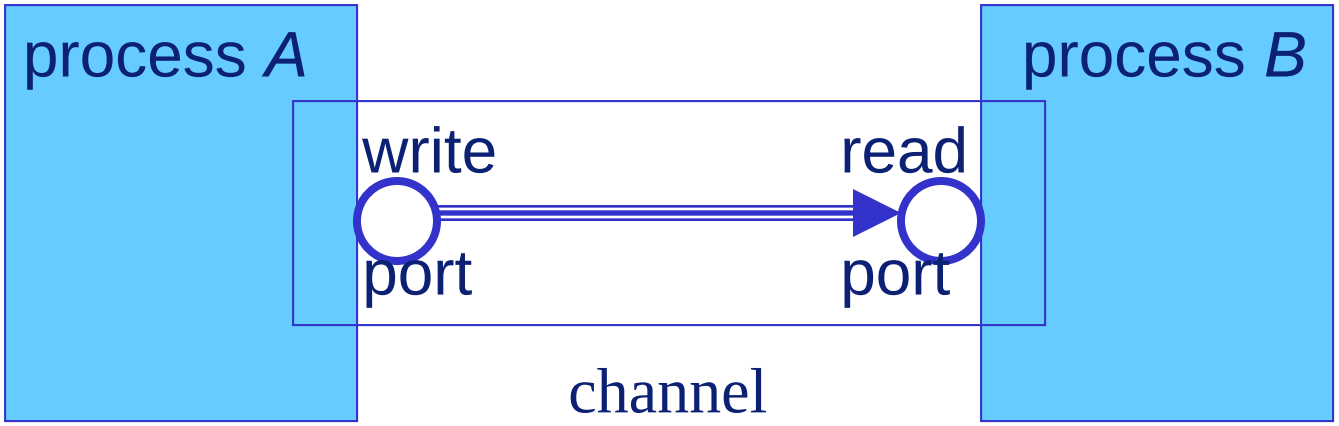
- Execution of these threads must be constrained according to the state transition diagram. Variable *st* records the state and must be modified together with the execution of the actions *X, Y* and *Z.*

  Synchronize this system using condition variables according to this diagram. Add the modifications of variable *st* to the three threadsand make sure that proper exclusion is guaranteed.

Communication

Synchronization: ordering of actions

Communication: data transport



Channel: abstraction from medium (e.g., bounded buffer, ethernet connection, wires, ...).

"Pair of connected ports"

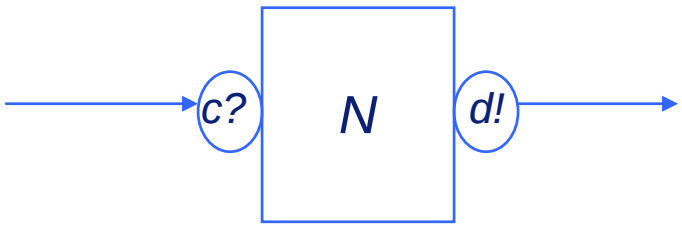
Channel definition

Channel *c*: object with three operations:

- *read a message from* c, assigning it to a variable
  Notations: *read(c,v), get(c,v), receive(c,v), c?v*
- *write the value of an expression to c*
  Notations: *write(c,e), put(c,e), send(c,e), c!e*
- *check if a read resp. write would succeed*
  Notation: *probe(c?) resp. probe(c!), (or just probe(c))*
  not empty(c) resp. not full(c) ...

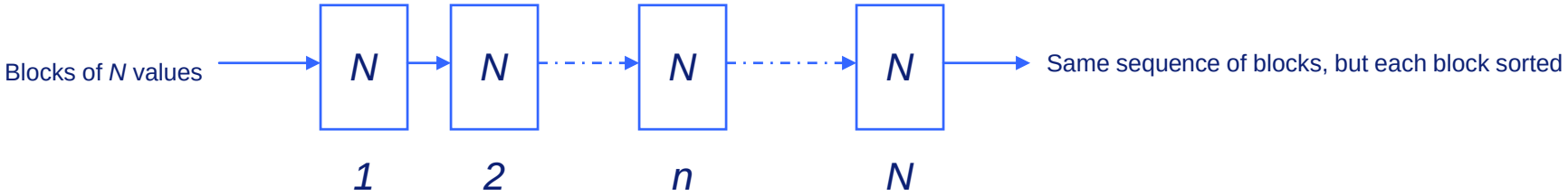
- testing often restricted to input only
- channel is point-to-point connection!
  - *synchronous*: *read* and *write* alway go together
    - empty/full means: partner is not ready
  - *buffered*: a limited excess in *writes* is allowed
  - *asynchronous*: *writes* are never blocked

Example: Block sorter



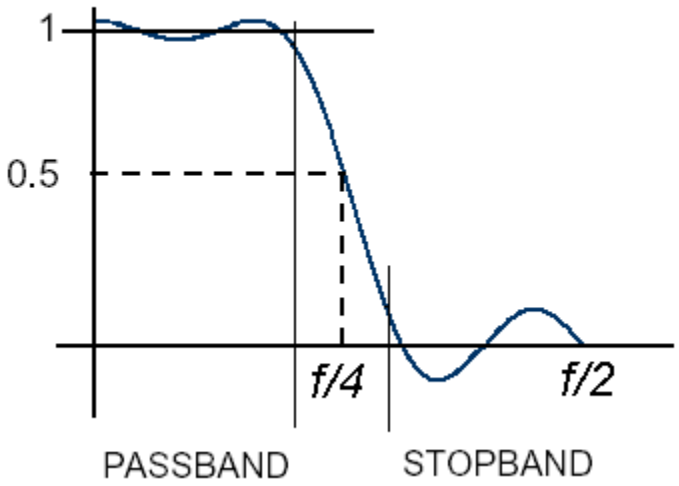

```
P_Bubble (var c?, d!: port) =
|[ var m, x: int;
   while true do
     receive(c,m);
     for i := 2 to N do
          receive(c,x);
          if x>m then send(c,m); m := x
          else send(c,x)
          fi
     od;
     send(c,m)
   od
]|
```

Out of N values hold up the highest value thus far encountered



Example: FIR filter

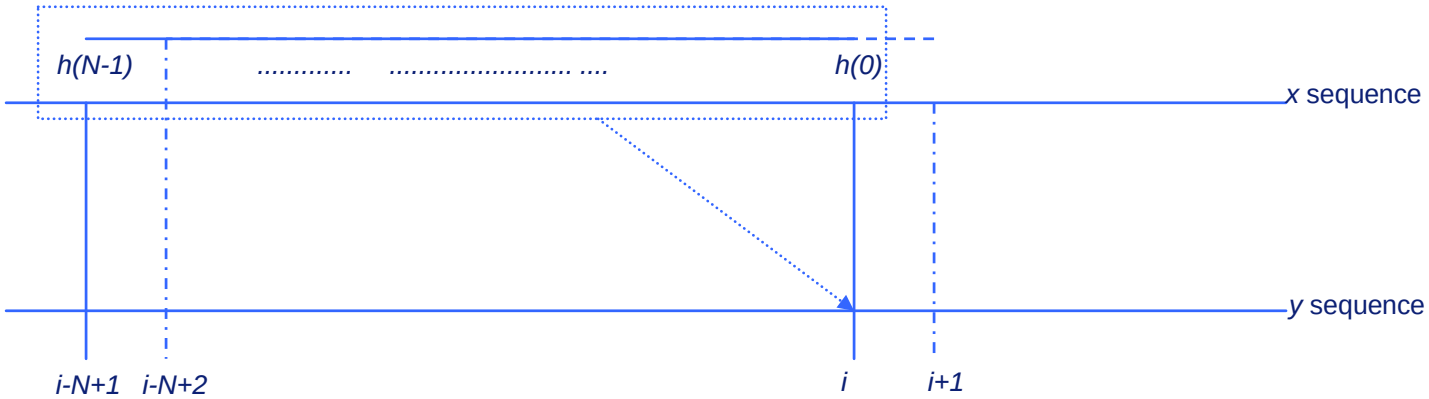
- Finite Impulse Response filters
  - Examples: low pass, band pass, high pass filters
  - A form of digital signal processing
- Example:
  - Sound sampled *f* times per second
    - can capture frequencies in band 0 to *f/2*
  - Half pass FIR filter can filter e.g. the upper part of that band



FIR filter

- Window computation, input sequence *x*, output sequence *y*
  - each output value is the result of a window of length *N*
  - for the next output, the window jumps one to the right
- Finite Impuls Response: multiply and accumulate the window with a fixed sequence of size *N: h(j: 0≤j<N)*
- Specification:

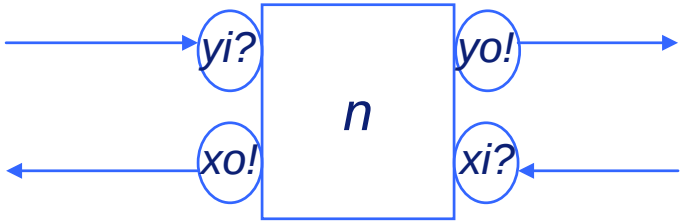
$$y(i) = \sum_{j=0}^{N-1} h(j) \cdot x(i-j)$$

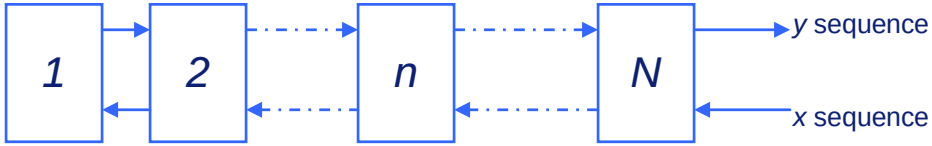


FIR filter, solution

- Just add a local contribution to a partial result

- $y_n(i) = \sum_{j=0}^{n-1} h(j+N-n)\cdot x(i-j)$
  $= y_{n-1}(i-1)+h(N-n)\cdot x(i)$





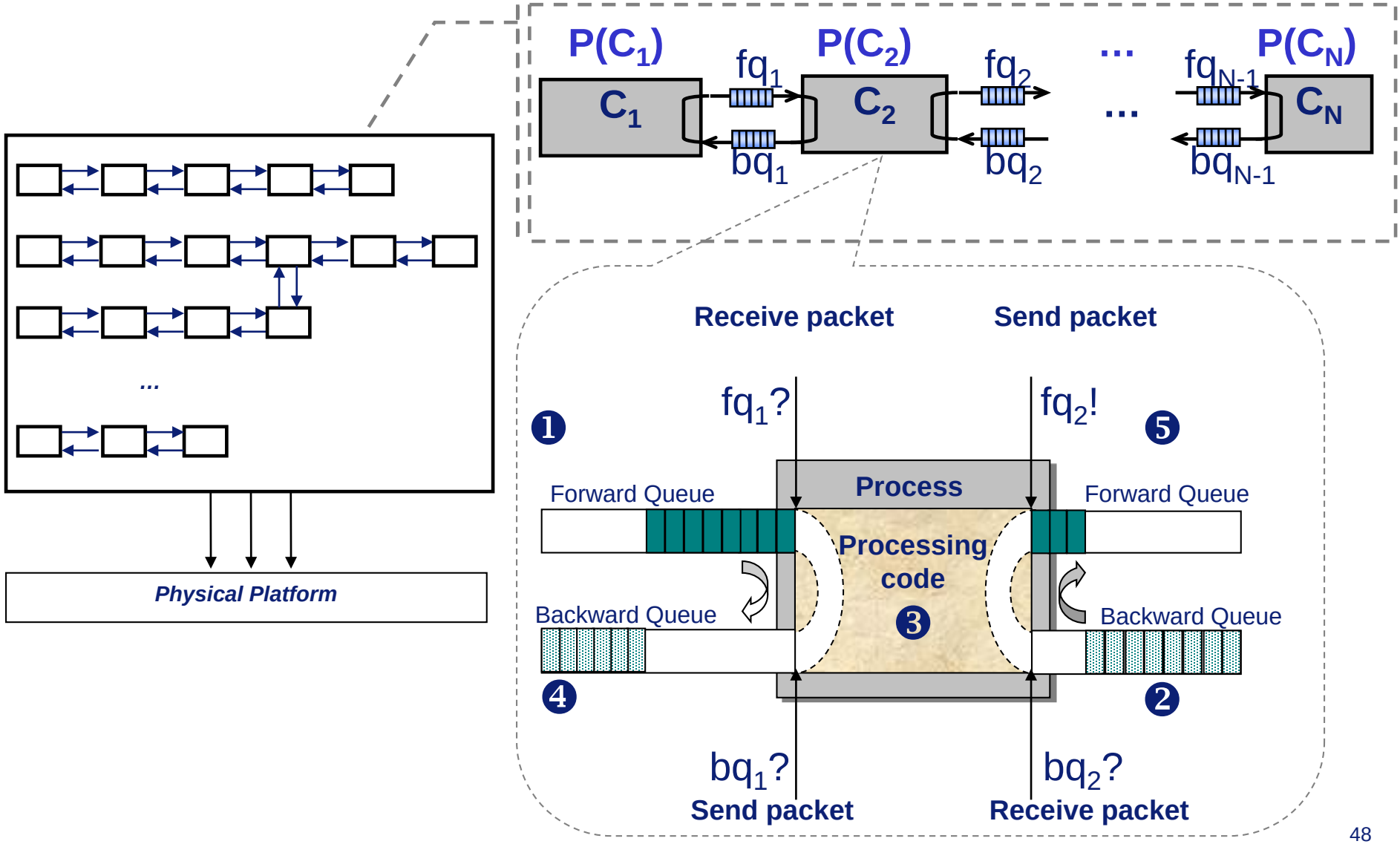
Case n>1

```
P_FIR (var xi?, xo!, yi?, yo!: port;
        H: int { H = h(N-n) } ) =
|[  var vy, vx: int;
    vy:=0; receive(xi, vx);
    while true do
      send(yo,vx*H+vy) || send(xo,vx);
      receive(yi,vy) || receive(xi,vx)
    od
]|
```

Case n=1

```
P_FIR (var xi?, yo!: port;
        H: int { H = h(N-1) } ) =
|[  var vx: int;
    while true do
      receive(xi,vx); send(yo,vx*H)
    od
]|
```

Example: TSSA



Example: TSSA

- threads at the borders are time-driven; other ones are data-driven
  - hence, there must be data available...
  - ... and space to write it

```
while (B) do
  receive(fq_{i−1}, fPacket);
  receive(bq_i, ePacket);
  process_fct_i(fPacket, ePacket);
  // fPacket is empty, ePacket is full
  send(bq_{i−1}, fPacket);
  send(fq_i, ePacket);
od
```

- Priorities determine
  - initialization effects
  - context switches
- Queus are used
  - to synchronize, and to recycle memory
  - to handle variations in computation times

Example: multiplexing

- first come first serve
- potentially unfair
- this special *await* has an efficient implementation

```
P_Combine (var c?, d?, e!: port) =
|[ var p: int;
   while true do
      await (¬ (empty(c) ∧ empty(d)));
      { ¬ (empty(c) ∧ empty(d)) }
      if empty(c) then receive(d,p) else receive (c,p) fi;
      send (e,p)
   od
]|
```

Fair combine and split

```
P_Combine (var c?, d?, e!: port) =
|[ var p: int;
  while true do
    await (¬(empty(c) ∧ empty(d)));
    if ¬empty (c) then receive(c,p); send (e,p) fi;
    if ¬empty (d) then receive(d,p); send(e,p) fi
  od ]|
```

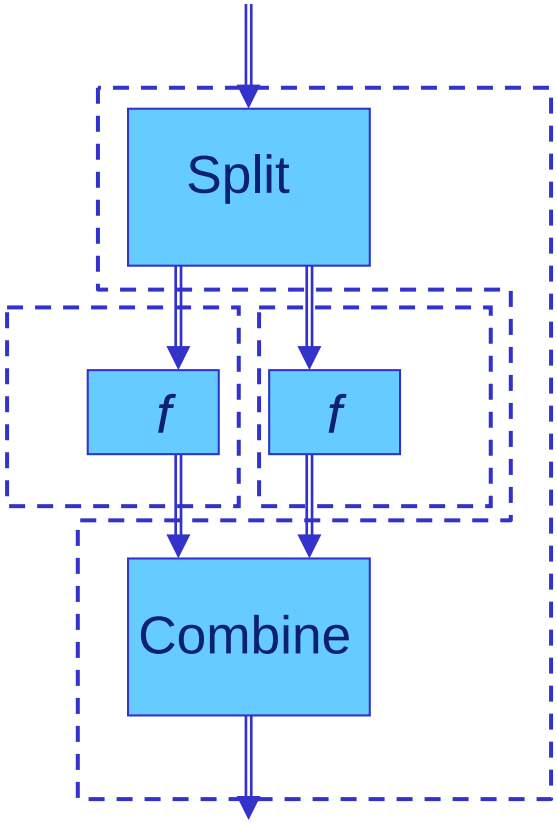
Similarly,

```
P_Split (var c?, d!, e!: port) =
|[ var p: int;
  while true do
    await (¬(full(d) ∧ full(e)));
    if ¬full (d) then receive(c,p); send (d,p) fi;
    if ¬full (e) then receive(c,p); send(e,p) fi
  od ]|
```

Concurrent computation

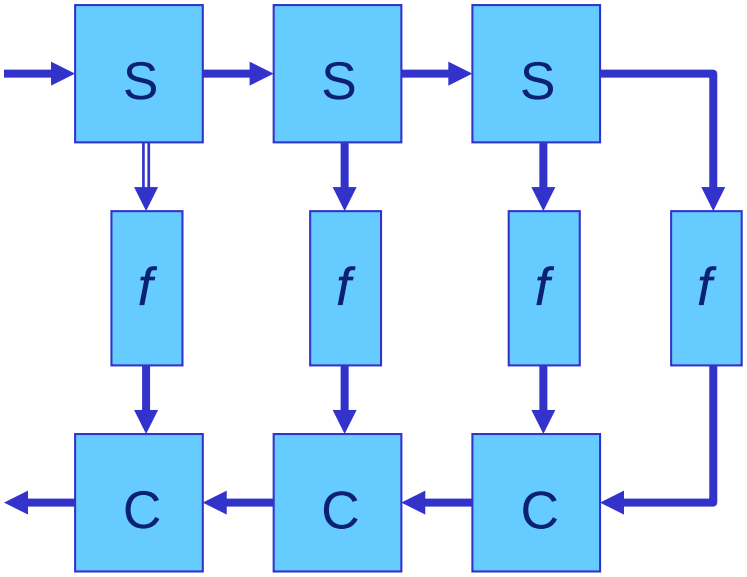
Dashed lines: processor boundaries

- Information stream split across two machines
- Computation on two sub-streams concurrently
- Results combined into one stream again.

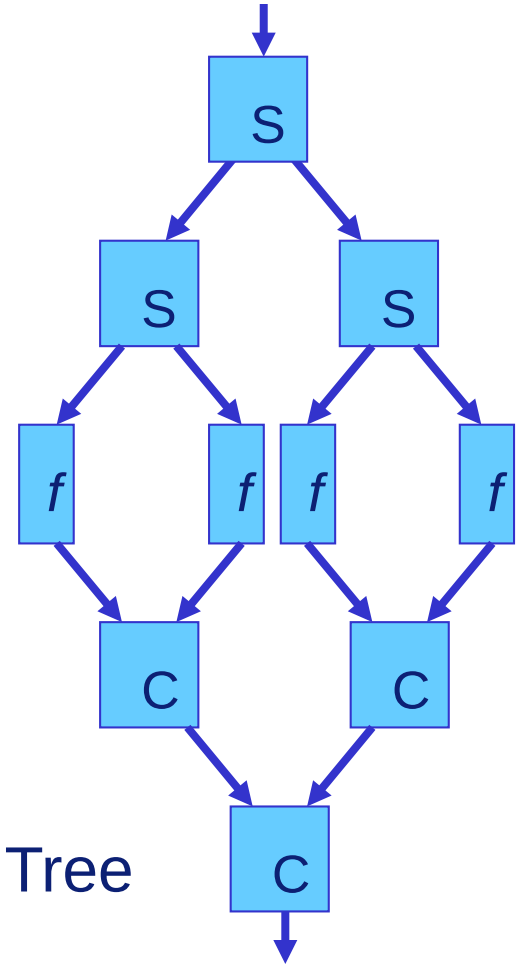


"Processor Farming"

Use a good idea twice...



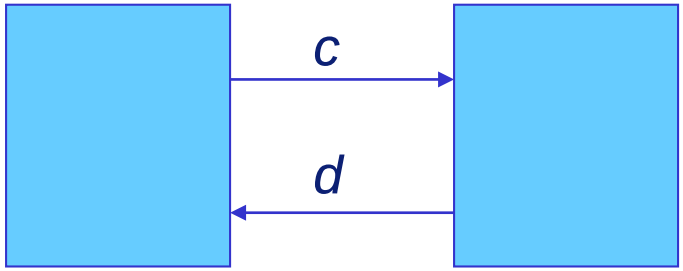
Pipeline



Tree

Sources of deadlock

- Any cycle in the interconnection pattern is potentially hazardous. Simple cases:

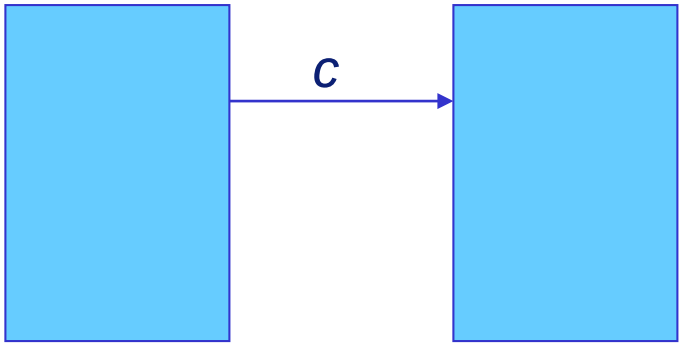


receive(d,v); send (c,3)	*receive(c,u); send (d,4)*	deadlock
receive(d,v); send (c,3)	*send (d,4); receive(c,u)*	ok
receive(d,v); send (c,3)	*receive(c,u) || send (d,4)*	ok
send (c,3); receive(d,v)	*send (d,4); receive(c,u)*	depends on buffering

Active and passive

- A communication action is called
  - **active** if it is performed unconditionally
  - **passive** if it is performed only when the partner starts first

  "Active/passive" is also used for the communicating partners.



For a synchronous channel: **deadlock**

await $(\neg$ *full(c)); send(c,e)*

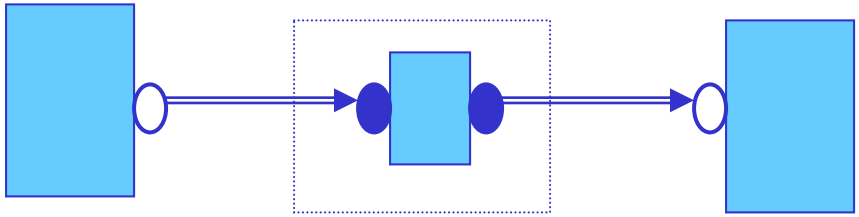
await$(\neg$ *empty(c)); receive(c,v)*

- *At least one partner in synchronous communication must be active.*

Passive threads

- multiplexers (e.g., Combine, Split)
- any “server-like” processes: waiting for inputs from multiple sources
- buffers – possibly

Solving passive-passive problems: add an active process in the middle



One-place-buffer
2 active ports (drawn filled)

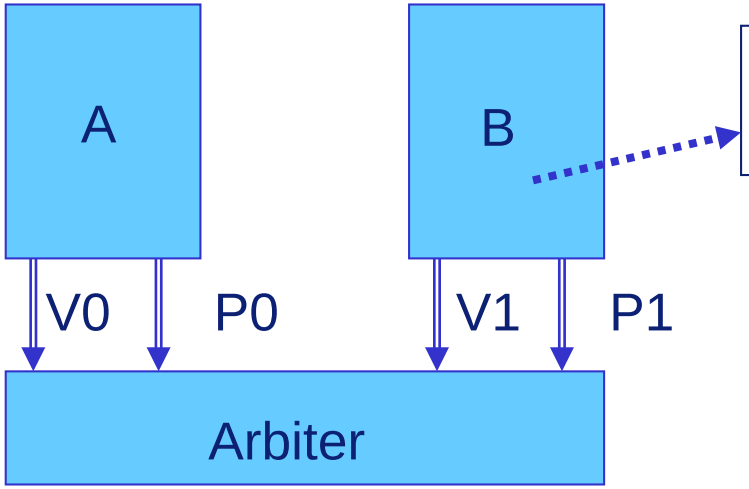
Distributed synchronization

- No shared memory: use *communication* to synchronize (no semaphores available)
- Examples: mutual exclusion, resource allocation,...

- Three steps in the design:
  - decide upon an *architecture*, processes and channels in between them
  - define a communication *protocol* for the required synchronization
  - develop the corresponding program texts
    - consider systematically all possible inputs
    - analyze the state changes and subsequent outputs

- 'Standard': introduce an *arbiter*, a new process especially for the synchronization

Mutual exclusion revisited



Protocol:
...send(P1, -); CriticalSection; send (V1,-)..

- Only ***synchronous*** communication
- Use "-" to denote an arbitrary value

$P_{Arbiter}$ (**var** *P0?, P1?, V0?, V1?: port*) =
|[**while** *true* **do**
  await ($\neg$(*empty(P0)* $\wedge$ *empty(P1)*));
  if *empty(P0)* **then** *receive(P1,-); receive(V1,-)*
  else *receive(P0,-); receive(V0,-)*
  fi
 od]|

Notation

Instead of

```
await (¬(empty(c) ∧ empty(d)));
if ¬ empty(c) then receive(c,...); ....
else receive(d,...); ....
fi
```

use

```
select receive(c,...) do ...
   or   receive(d,...) do ....
endselect
```

Fairness: depends on definition of **select**

Exercises

D.1 In the TSSA system, describe the difference between an increasing priority assignment and a decreasing one. Consider only the data-driven behavior.

D.2 Consider the following alternative to process *Combine*: first wait until an output can be generated and then accept an input. Give an implementation and discuss the difference. A similar alternative for *Split* starts with an input and then selects its output. Can you give an environment that makes the difference visible (e.g., by deadlocking with one implementation)?

Exercises

D.3 Discuss the effect of buffering on the channels in the processor farming example. If buffered channels are given, the buffering can be avoided by adding “request” channels; an input is then preceded by a request for it. Adapt the processes in this way.

D.4 Design and implement an *N*-place buffer. Such a buffer copies its input to its output and is capable of storing at most *N* items (it could use a shared-memory bounded buffer implementation for that purpose). First make a fully passive buffer, then make it active at both input and output. Integrate your solution into a single process that may have internal shared-memory parallelism.

Exercises

D.5 Redesign the arbiter for the case of controlling a collection of *K* resources. The processes may acquire different numbers of resources. The identity of the resources is of no concern, only the number matters. Restrict yourself to two processes.

D.6 The bounded buffer of exercise D.4 now has two producers and two consumers that produce and consume elements in portions of arbitrary size. Design a protocol for the producers and consumers and develop a new solution for the buffer.

Exercises

D.7 Redo exercise A.5 using channels and communication between the processes. Explicitly describe the architecture and the protocol you are using.