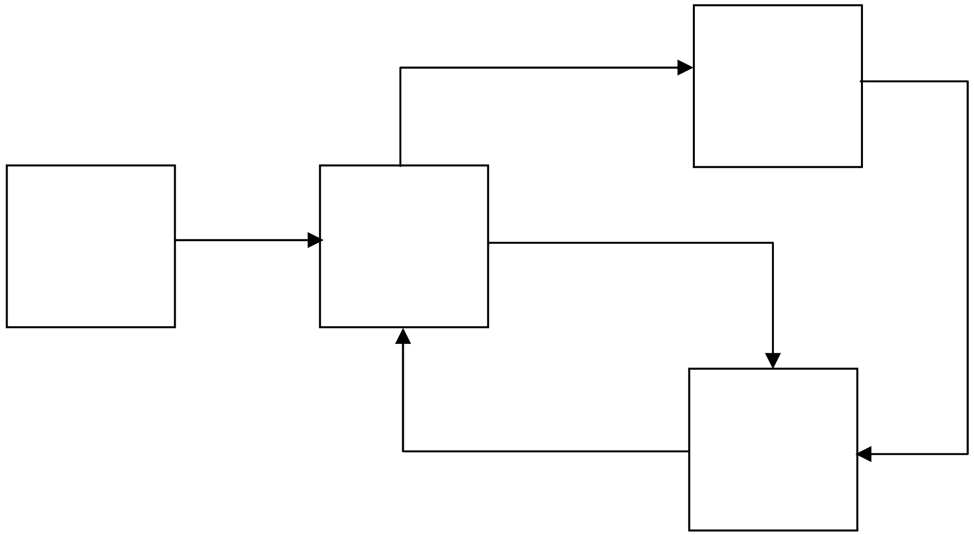
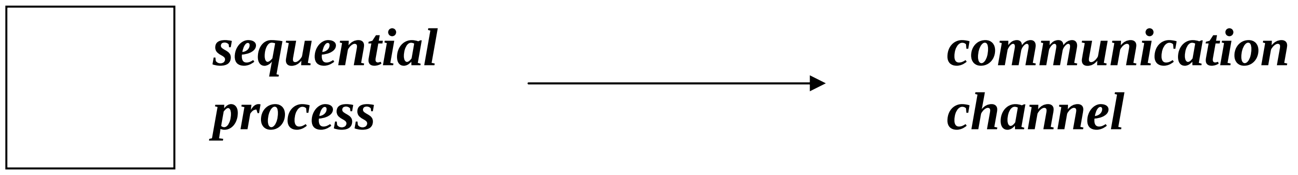
Communicating Sequential Processes (CSP)





- single thread of control
- autonomous
- encapsulated
- named
- static

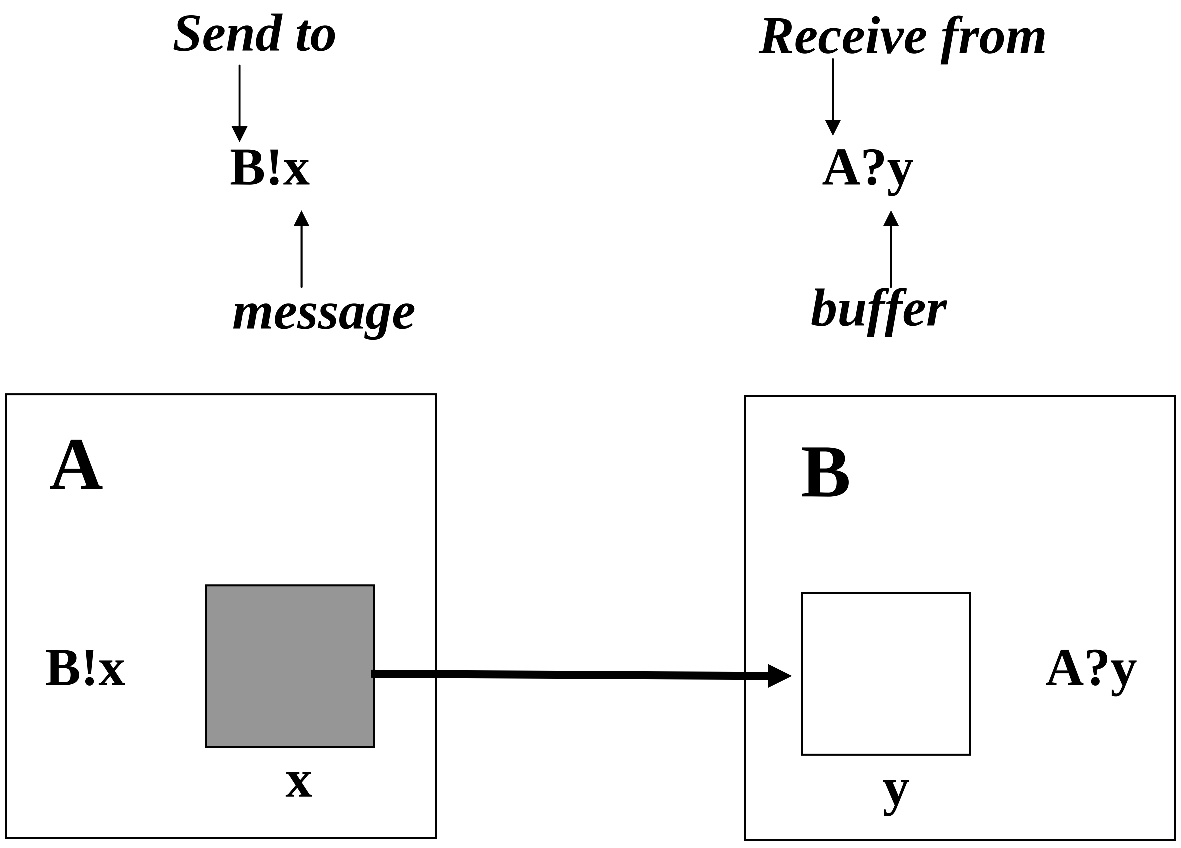
- synchronous
- reliable
- unidirectional
- point-to-point
- fixed topology

Communicating Sequential Processes (CSP)

operators: **! (send)**

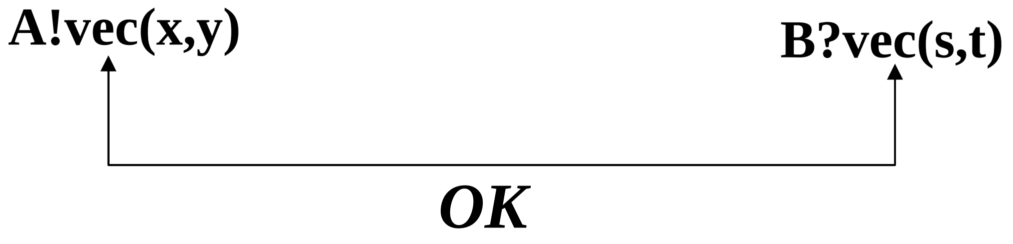
? (receive)

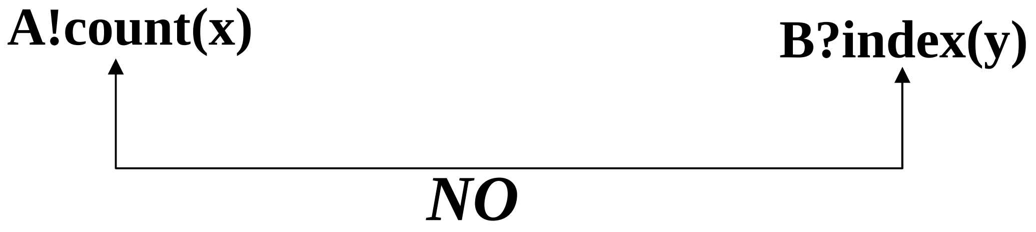
usage:



Communicating Sequential Processes (CSP)

- rendezvous semantics: senders (receivers) remain blocked at send (receive) operation until a matching receive (send) operation is made.
- typed messages: the type of the message sent by the sender and the type of the message expected by the receiver must match (otherwise abort).





Communicating Sequential Processes (CSP)

Guarded Commands

<guard> $\rightarrow$ <command list>

boolean expression

at most one ? , must be at end of guard, considered true iff message pending

Examples

n < 10 $\rightarrow$ A!index(n); n := n + 1;
n < 10; A?index(n) $\rightarrow$ next = MyArray(n);

Communicating Sequential Processes (CSP)

Alternative Command

$$[\ G_1 \rightarrow S_1\ []\ G_2 \rightarrow S_2\ []\ \ldots\ []\ G_n \rightarrow S_n\ ]$$

1. evaluate <u>all</u> guards
2. if more than on guard is true, <u>nondeterministically</u> select one.
3. if no guard is true, <u>terminate</u>.

Note: if all true guards end with an input command for which there is no pending message, then delay the evaluation until a message arrives. If all senders have terminated, then the alternative command terminates.

Repetitive Command

$$*\ [\ G_1 \rightarrow S_1\ []\ G_2 \rightarrow S_2\ []\ \ldots\ []\ G_n \rightarrow S_n\ ]$$

repeatedly execute the alternative command until it terminates

Communicating Sequential Processes (CSP)

Examples:

```
[x >= y --> m := x [] y >= x --> m := y ]
```

assign x to m if x is greater than or equal to y
assign y to m if y is greater than or equal to x
assign either x or y to m if x equals y

```
* [ c: character; west?c --> east!c ]
```

Transmit to the process named <u>east</u> a character received from the process named <u>west</u> until the process named <u>west</u> terminates.

Communicating Sequential Processes (CSP)

```
SEARCH
        i := 0; * [ i < size; content(i) != n --> i := i + 1 ]
```

Scan the array context until the value n is found or until the end of the array of length size is reached

```
LISTMAN:: *[ n : integer; X?insert(n) --> INSERT
             []
             n : integer; X?has(n) --> SEARCH; X!(i < size)
            ]
```

LISTMAN has a simple protocol defined by two messages - an insert message and a has message. The types insert and has are used to disambiguate the integer value passed on each communication with X. INSERT is code (not shown) that adds the value of n to the array content. SEARCH is the code shown above. LISTMAN replies with a boolean value to each has message.

Signals between Processes

A message bearing a type but no data may be used to convey a “signal” between processes. For example:

```
Semaphore::
  val:integer; val = 0;
  *[   X?V()--> val = val + 1
    []
       val > 0; Y?P()--> val = val - 1
   ]
```

Communicating Sequential Processes (CSP)

```
BoundedBuffer::
     buffer: (0..9) portion;
     in, out : integer; in := 0; out := 0;
     * [ in < out + 10; producer?buffer(in mod 10)
              --> in := in + 1;
        []
          out < in; consumer?more()
              --> consumer!buffer(out mod 10);
                   out := out + 1;
       ]
```

Implements a bounded buffer process using the array buffer to hold up to a maximum of 10 values of type portion. Note how the guarded commands do not accept <u>producer</u> messages when the buffer is full and do not accept <u>consumer</u> messages when the buffer is empty.

Communicating Sequential Processes

```
lineimage:(1..125) character;
i: integer; i:=1;
* [ c:character; X?c -->
       lineimage(i);+ c;
       [ i <= 124 --> i := i+1;
        []
         i = 125 --> lineprinter!lineimage; i:=1;
       ]
   ]
   [ I = 1 --> skip
     []
     i>1 --> *[i <= 125 --> lineimage(i):= space; i:= i+1;]
             lineprinter!lineimage
   ]
```

Read a stream of characters from X and print them in lines of 125 characters on a lineprinter completing the last line with spaces if necessary.

Arrays of Processes

`X(i: 1..100):: […process definition…]`

declares an array of processes all with the same code but with different names (e.g., X(1), X(2),…, X(100))

Communication among processes in the array is facilitated by the use of input/output commands as illustrated in this code fragment:

`*[ (i:1..100)X(i)?(params) --> …; X(i)!(result) ]`

where the bound variable i is used to identify the communicating partner process

CSP - Comparison with Monitors

Guarded Commands

- Monitor: begin executing every call as soon as possible, waiting if the object is not in a proper state and signaling when the state is proper
- CSP: the called object establishes conditions under which the call is accepted; calls not satisfying these conditions are held pending (no need for programmed wait/signal operations).

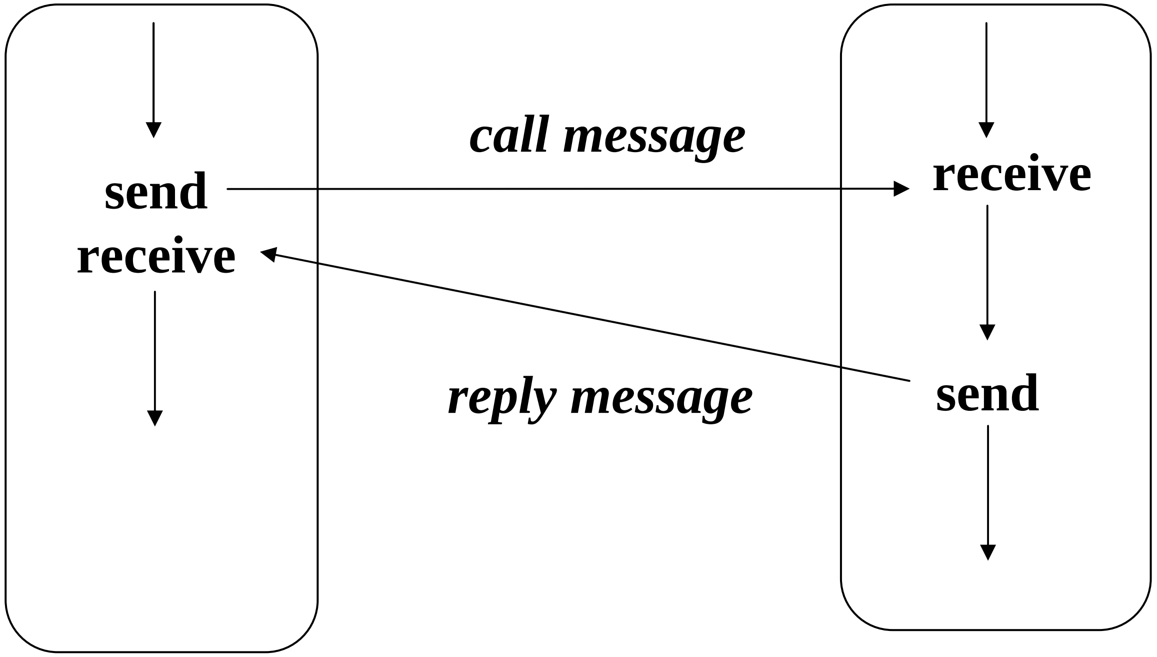
Rendezvous

- Monitor: the monitor is passive (has no independent task/thread/activity)
- CSP: synchronization between peer, autonomous activities.

CSP

Distribution:

- Monitor: inherently non-distributed in outlook and implementation
- CSP: possibility for distributed programming using synchronous message passing



Rendezvous in ADA

```
task bounded-buffer is
    entry store(x : buffer);
    entry remove(y: buffer);
end;
task body bounded-buffer is
...declarations...
begin
  loop
      select
            when head < tail + 10 =>
            accept store(x : buffer) ... end store;
      or
            when tail < head =>
            accept remove(y: buffer) ... end remove;
      end select;
  end loop
end
```