Message passing and channels

IN5170/IN9170 – Models of concurrency

Message passing and channels

Fall 2022

03. Oct. 2022



Outline

Course overview:

- Part I: concurrent programming; programming with shared variables
- Part II: "distributed" programming

Outline: asynchronous and synchronous message passing

- concurrent vs. distributed programming[1]
- asynchronous message passing: channels, messages, primitives
- example: filters and sorting networks
- from monitors to client–server applications
- comparison of message passing and monitors
- about synchronous message passing

[1] The dividing line is not absolute. One can make perfectly good use of channels and message passing also in a non-distributed setting.

Shared memory vs. distributed memory

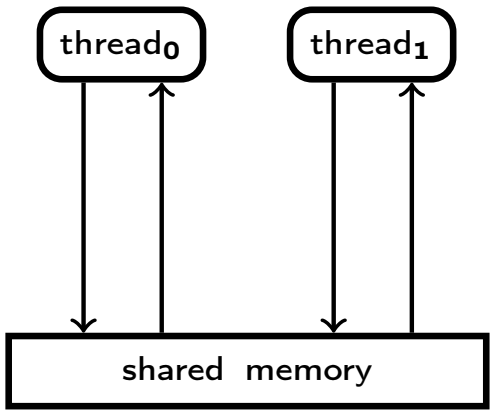
traditional system architectures have one shared memory:

- many processors access the same physical memory
- example: fileserver with many processors on one motherboard

Distributed memory architectures:

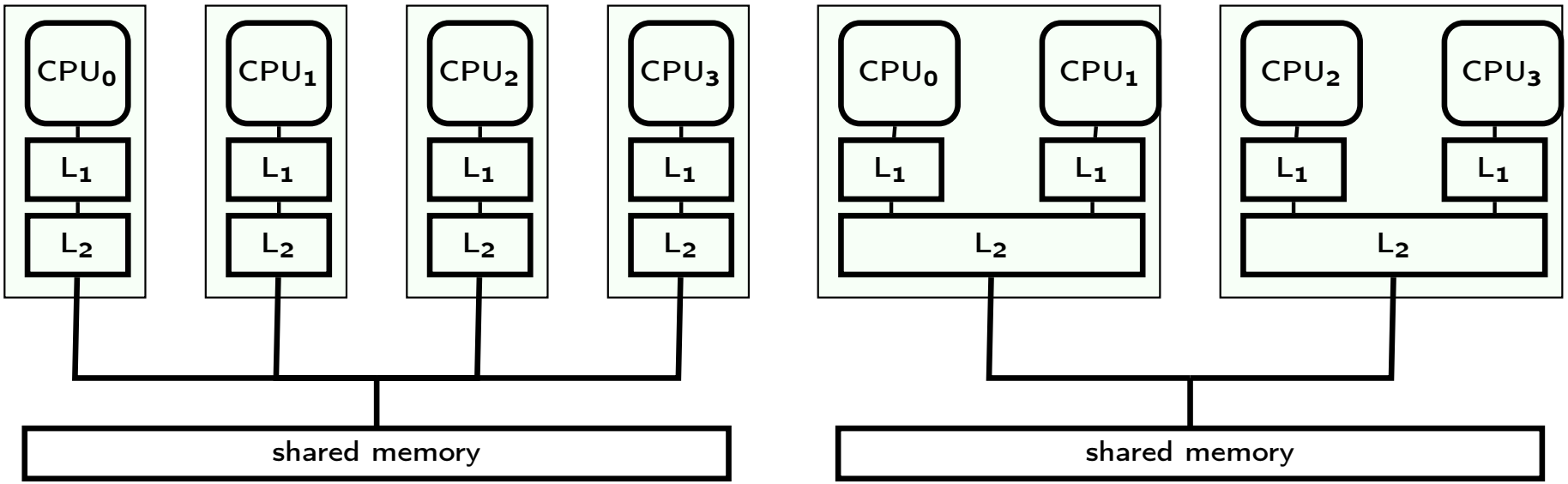
- processor has private memory and communicates over a "network" (inter-connect)
- examples:
  - multicomputer: asynchronous multi-processor with distributed memory (typically contained inside one case)
  - Workstation clusters: PC's in a local network, NFS
  - Grid system: machines on the Internet, resource sharing
  - cloud computing: cloud storage service
  - NUMA-architectures
  - cluster computing ...

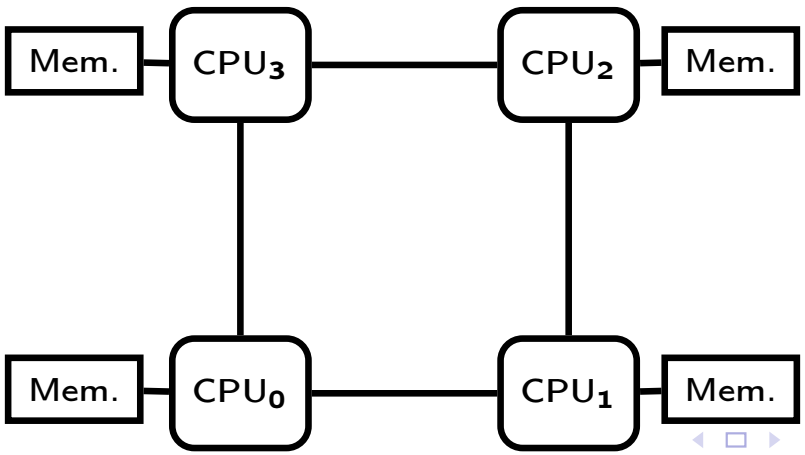
Shared memory concurrency in the real world



- the memory architecture does not reflect reality
- out-of-order executions:
  - modern systems: complex memory hierarchies, caches, buffers...
  - compiler optimizations,

SMP, multi-core architecture, and NUMA





Concurrent vs. distributed programming

Concurrent programming:

- Processors share one memory
- Processors communicate via reading and writing of shared variables

Distributed programming:

- Memory is distributed
  $\Rightarrow$ processes cannot share variables (directly)
- Processes communicate by sending and receiving *messages* via shared *channels*

  or (in future lectures): communication via *RPC* and *rendezvous*

Asynchronous message passing: channel abstraction

Channel: abstraction, e.g., of a physical communication network[2]

- One–way from sender(s) to receiver(s)
- unbounded FIFO (queue) of waiting messages
- preserves message order
- atomic access
- error–free
- typed

Variants: errors possible, untyped, lossy, unnamed, bounded . . .

[2] but remember also: producer-consumer problem

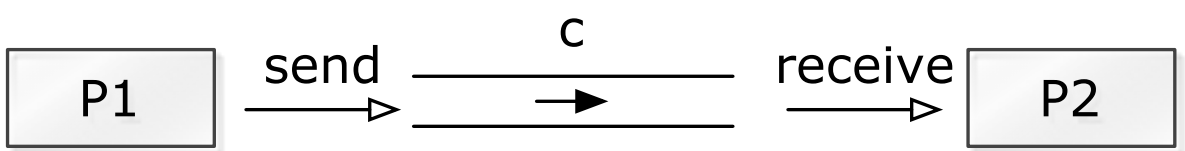
Channel declaration

$$\text{chan c}(\text{type}_1 \text{id}_1, \ldots, \text{type}_n \text{id}_n);$$

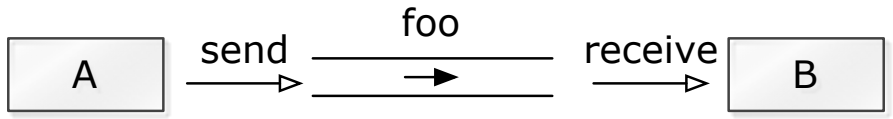
Messages: n-tuples of values of the respective types

communication primitives:

- send $c(\text{expr}_1, \ldots, \text{expr}_n)$;
  Non-blocking, i.e. asynchronous
- receive $c(\text{var}_1, \ldots, \text{var}_n)$;
  Blocking: receiver waits until message is sent on the channel
- empty (c);
  True if channel is empty



Example: message passing



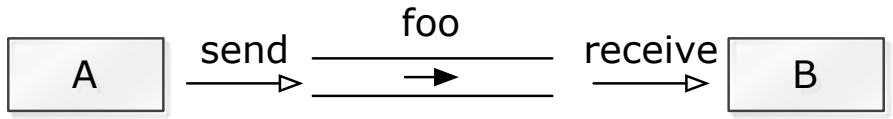
(x,y) =

```
chan foo(int);

process A {
  send foo(1);
  send foo(2);
}

process B {
  receive foo(x);
  receive foo(y);
}
```

Example: message passing



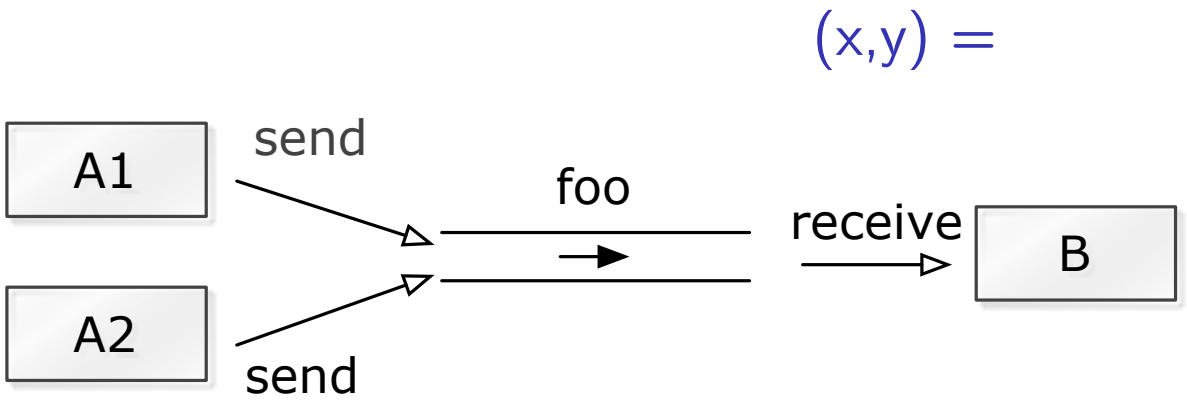
(x,y) = (1,2)

```
chan foo(int);

process A {
  send foo(1);
  send foo(2);
}

process B {
  receive foo(x);
  receive foo(y);
}
```

Example: shared channel



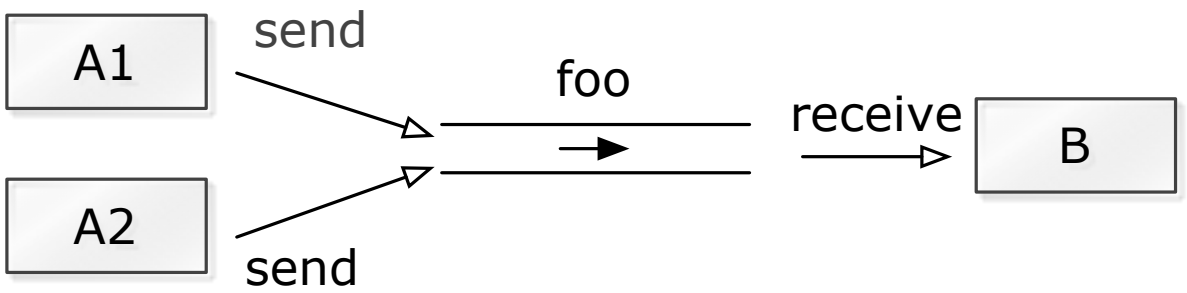

```
process A1 {
  send foo(1);
}

process A2 {
  send foo(2);
}

process B {
  receive foo(x);
  receive foo(y);
}
```

Example: shared channel

(x,y) = (1,2) or (2,1)




```
process A1 {
  send foo(1);
}

process A2 {
  send foo(2);
}

process B {
  receive foo(x);
  receive foo(y);
}
```

Asynchronous message passing and semaphores

Comparison with general semaphores:

channel	$\simeq$	*semaphore*
send	$\simeq$	***V***
receive	$\simeq$	***P***

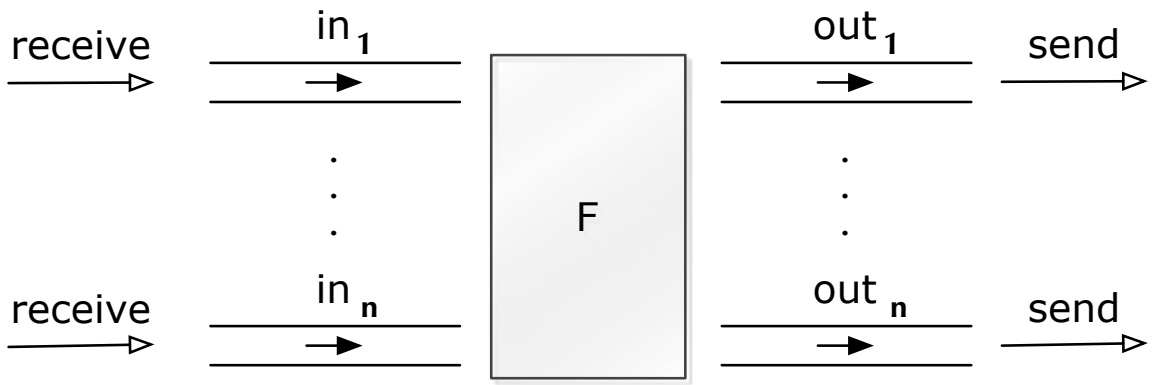
Number of messages in queue = value of semaphore

(Ignores content of messages)

Filter **F**

= process which:

- receives messages on input channels,
- sends messages on output channels, and
- output is a function of the input (and the initial state).



- A filter is specified as a predicate.
- Some computations: naturally seen as a composition of filters.
- cf. *stream* processing/programming (feedback loops) and *dataflow programming*

Example: A single filter process

Problem: Sort a list of n numbers into ascending order.

process **Sort** with input channels **input** and output channel **output**.

Define:

n : number of values sent to **output**.
$sent[i]$: i'th value sent to **output**.

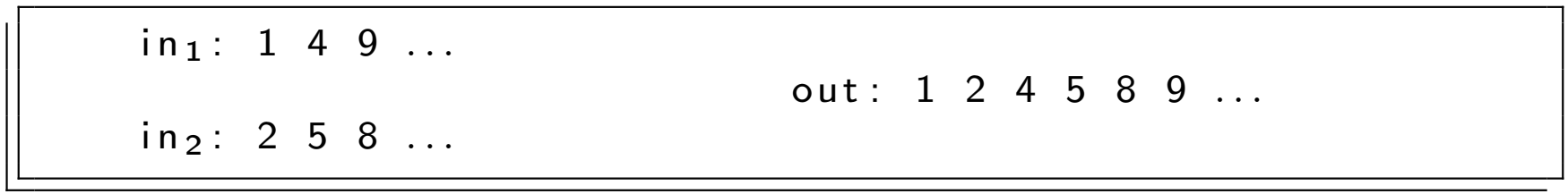
Sort predicate

$\forall i : 1 \leq i < n.\ \big(sent[i] \leq sent[i+1]\big)$
$\wedge$ *values sent to* ***output*** *are a permutation of values from* ***input***.

Filter for merging of streams

Problem: Merge two sorted input streams into one sorted stream.

Process `Merge` with input channels $\mathbf{in}_1$ and $\mathbf{in}_2$ and output channel **out**:



Special value **EOS** marks the end of a stream.

Define:

n : number of values sent to **out**.
$sent[i]$: i'th value sent to **out**.

The following shall hold when **Merge** *terminates*:

$\boldsymbol{in}_1$ *and* $\boldsymbol{in}_2$ *are empty* $\wedge$ $sent[n+1] = \boldsymbol{EOS}$
$\wedge \quad \forall i : 1 \leq i < n\big(sent[i] \leq sent[i+1]\big)$
$\wedge$ *values sent to* **out** *are a permutation of values from* $\boldsymbol{in}_1$ *and* $\boldsymbol{in}_2$

Example: **Merge** process

```
chan in1(int), in2(int), out(int);

process Merge {
  int v1, v2;
  receive in1(v1);            # read the first two
  receive in2(v2);            # input values

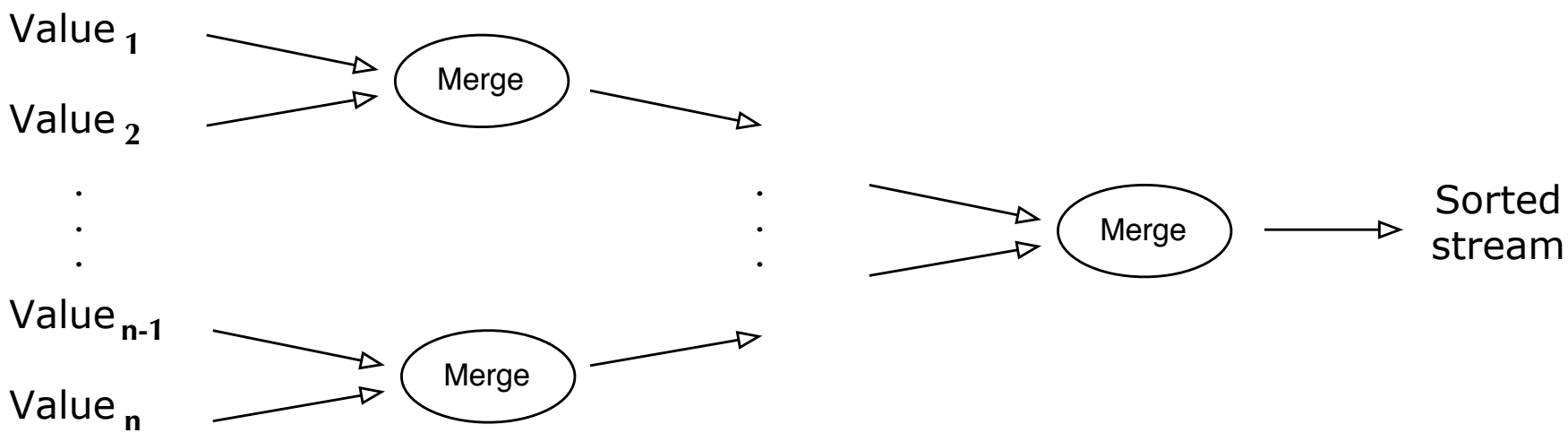
  while (v1 ≠ EOS and v2 ≠ EOS) {
    if (v1 ≤ v2)
      { send out(v1); receive in1(v1); }
    else                                # (v1 > v2)
      { send out(v2); receive in2(v2); }
  }

                              # consume the rest
                              # of the non-empty input channel
  while (v2 ≠ EOS)
    { send out(v2); receive in2(v2); }
  while (v1 ≠ EOS)
    { send out(v1); receive in1(v1); }
  send out(EOS);    #  add special value to out
}
```

Sorting network

We now build a network that sorts n numbers.

We use a collection of `Merge` processes with tables of shared input and output channels.



(Assume: number of input values n is a power of 2)

Client-server applications using messages

Server: process, repeatedly handling requests from client processes.

Goal: Programming client and server systems with asynchronous message passing.

```
chan request(int clientID, ...),
     reply[n](...);

client nr. i                              server
                                          int id;  # client id.


                                          while(true) {  # server loop
send request(i,args);          ⟶           receive request(id,vars);
:                                 :          :
receive reply[i](vars);        ⟵           send reply[id](results);
                                            }
```

Monitor implemented using message passing

Classical monitor:

- controlled *access* to shared resource
- Permanent variables (monitor variables): safeguard the resource state
- access to a resource via *procedures*
- procedures: executed under *mutual exclusion*
- *condition* variables for synchronization

also implementable by server process + message passing
Called "active monitor" in the book: active process (loop), instead of passive procedures.[3]

[3] In practice: server may spawn local threads, one per request.

Allocator for multiple–unit resources

Multiple–unit resource: a resource consisting of multiple units

Examples: memory blocks, file blocks.
Users (clients) need resources, use them, and return them to the allocator ("free" the resources).

- here simplification: users get and free *one* resource at a time.

two versions

1. monitor
2. server and client processes, message passing

Allocator as monitor

Uses "passing the condition" pattern ⇒ simplifies later translation to a server process

Unallocated (free) units are represented as a set, type **set**, with operations **insert** and **remove**.

Recap: "semaphore monitor" with "passing the condition"

```
monitor Semaphore          { # monitor invariant: s ≥ 0
  int s := 0;                # value of the semaphore
  cond pos;                  # wait condition

  procedure Psem() {
    if   (s=0)
         wait (pos);
    else
         s := s - 1
  }

  procedure Vsem() {
    if   empty(pos)
         s := s + 1
    else
         signal(pos);
  }
}
```

(Fig. 5.3 in Andrews [Andrews, 2000])

Allocator as a monitor

```
monitor Resource_Allocator {
  int avail := MAXUNITS;
  set units := ... # initial values;
  cond free;          # signalled when process wants a unit

  procedure acquire(int &id) {  # var.parameter
    if (avail = 0)
      wait(free);
    else
      avail := avail-1;
    remove(units, id);
  }

  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail := avail+1;
    else
      signal(free);                # passing the condition
  }
}
```

([Andrews, 2000, Fig. 7.6])

Allocator as a server process: code design

1. interface and "data structure"
2. control structure: nested **if**-statement (2 levels):
3. synchronization, scheduling, and mutex

Allocator as a server process: code design

1. interface and "data structure"
   1.1 allocator with two types of operations: get unit, free unit
   1.2 1 request channel[4] $\Rightarrow$ must be *encoded* in the arguments to a request.
2. control structure: nested **if**-statement (2 levels):
3. synchronization, scheduling, and mutex

[4] Alternatives exist

Allocator as a server process: code design

1. interface and "data structure"
   - 1.1 allocator with two types of operations: get unit, free unit
   - 1.2 1 request channel[4] $\Rightarrow$ must be *encoded* in the arguments to a request.
2. control structure: nested **if**-statement (2 levels):
   - 2.1 first checks the kind of the operation,
   - 2.2 proceeds correspondingly to monitor-**if**.
3. synchronization, scheduling, and mutex

[4] Alternatives exist

Allocator as a server process: code design

1. interface and "data structure"
   - 1.1 allocator with two types of operations: get unit, free unit
   - 1.2 1 request channel[4] $\Rightarrow$ must be *encoded* in the arguments to a request.
2. control structure: nested **if**-statement (2 levels):
   - 2.1 first checks the kind of the operation,
   - 2.2 proceeds correspondingly to monitor-**if**.
3. synchronization, scheduling, and mutex
   - 3.1 cannot wait (**wait(free)**) when no unit is free.
   - 3.2 must save the request and return to it later
     $\Rightarrow$ queue of pending requests (**queue**; **insert**, **remove**).
   - 3.3 request: "synchronous/blocking" call $\Rightarrow$ "ack"-message back
   - 3.4 no internal parallelism $\Rightarrow$ mutex

[4] Alternatives exist

Channel declarations:

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitID);
chan reply[n](int unitID);
```

Allocator: client processes

```
process Client[i = 0 to n-1] {
  int unitID;
  send request(i, ACQUIRE, unitID)        # make request
  receive reply[i](unitID);               # works as ``if synchronous''
  ...                                     # use resource unitID
  send request(i, RELEASE, unitID); # free resource
  ...
}
```

(Fig. 7.7(b) in Andrews)

Allocator: server process

```
process Resource_Allocator {
  int avail := MAXUNITS;
  set units := ...                    # initial value
  queue pending;                      # initially empty
  int clientID, unitID; op_kind kind; ...
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind = ACQUIRE) {
      if (avail = 0)                  # save request
        insert(pending, clientID);
      else { #  perform request now
          avail:= avail-1;
          remove(units, unitID);
          send reply[clientID](unitID);
        }
    }
    else {                            # kind = RELEASE
      if empty(pending) {             # return units
        avail := avail+1; insert(units, unitID);
      } else {                        # allocates to waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
} } } }                              # Fig. 7.7 in Andrews (rewritten)
```

Allocator as a monitor

```
monitor Resource_Allocator {
  int avail := MAXUNITS;
  set units := ... # initial values;
  cond free;          # signalled when process wants a unit

  procedure acquire(int &id) {  # var.parameter
    if (avail = 0)
      wait(free);
    else
      avail := avail-1;
    remove(units, id);
  }

  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail := avail+1;
    else
      signal(free);                 # passing the condition
  }
}
```

([Andrews, 2000, Fig. 7.6])

Duality: monitors, message passing

monitor-based programs	*message-based programs*
monitor variables	local server variables
process-IDs	**request** channel, operation types
procedure call	**send request()**, **receive reply[i]()**
go into a monitor	**receive request()**
procedure return	**send reply[i]()**
wait statement	save pending requests in a queue
signal statement	get and process pending request (`reply`)
procedure body	branches in **if** statement wrt. op. type

Synchronous message passing

Primitives:

- New primitive for sending:
  synch_send c($\text{expr}_1, \ldots, \text{expr}_n$);

  Blocking send:
  - sender waits until message is received by channel,
  - i.e. sender and receiver "synchronize" sending and receiving of message
- Otherwise: like asynchronous message passing:
  receive c($\text{var}_1, \ldots, \text{var}_n$);
  empty(c);

Synchronous message passing: discussion

Advantages:

- Gives maximum size of channel.
  Sender synchronises with receiver
  $\Rightarrow$ receiver has at most 1 pending message per channel per sender
  $\Rightarrow$ sender has at most 1 unsent message

Disadvantages:

- reduced parallellism: when 2 processes communicate, 1 is always blocked.
- higher risk of deadlock.

Example: blocking with synchronous message passing

```
chan values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    ... # computation ...;
    synch_send values(data[i]);
} }

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    ... # computation ...;
} }
```

Example: blocking with synchronous message passing

```
chan values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    ... # computation ...;
    synch_send values(data[i]);
} }

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    ... # computation ...;
} }
```

Assume both producer and consumer vary in time complexity.
Communication using `synch_send/receive` will block.

With *asynchronous* message passing, the waiting is reduced.

Example:

```
chan in1(int), in2(int);

process P1 {
  int v1 = 1, v2;
  synch_send in2(v1);
  receive in1(v2);
}

process P2 {
  int v1, v2 = 2;
  synch_send in1(v2);
  receive in2(v1);
}
```

Example: deadlock using synchronous message passing

```
chan in1(int), in2(int);

process P1 {
  int v1 = 1, v2;
  synch_send in2(v1);
  receive in1(v2);
}

process P2 {
  int v1, v2 = 2;
  synch_send in1(v2);
  receive in2(v1);
}
```

P1 and **P2** block on **synch_send** – deadlock.
One process must be modified to do **receive** first
$\Rightarrow$ asymmetric solution.

Example: deadlock using synchronous message passing

```
chan in1(int), in2(int);

process P1 {
  int v1 = 1, v2;
  synch_send in2(v1);
  receive in1(v2);
}

process P2 {
  int v1, v2 = 2;
  synch_send in1(v2);
  receive in2(v1);
}
```

P1 and **P2** block on **synch_send** – deadlock.
One process must be modified to do **receive** first
$\Rightarrow$ asymmetric solution.

With asynchronous message passing (**send**) all goes well.

References I

[Andrews, 2000] Andrews, G. R. (2000).
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley.