Lecture 5: Message Passing & Other Communication Mechanisms (SR & Java)

- Intro: Synchronous & Asynchronous Message Passing
- Types of Processes in Message Passing
- Examples
  - Asynchronous Sorting Network Filter (SR)
  - Synchronous Network of Filters: Sieve of Eratosthenes (SR)
  - Client-Server and Clients with Multiple Servers with Asynchronous Message Passing (SR)
  - Asynchronous Heartbeat Algorithm for Network Topology (SR)
  - Synchronous Heartbeat Algorithm for Parallel Sorting (SR+Java)
- RPC & Rendezvous
  - Examples

Introduction to Message Passing

- Up to now concurrency constructs (critical sections, semaphores, monitors) have been based on shared memory systems.
- However with network architectures & distributed systems in which processors are only linked by a communications medium, message passing is a more common approach.
- In message passing the processes which comprise a concurrent program are linked by *channels*.
- If the two interacting processes are located on the same processor, then this channel could simply be the processor's local memory.
- If the 2 interacting processes are allocated to different processors, then channel between them is mapped to a physical communications medium between the corresponding 2 processors.

Message Passing Constructs

- There are 2 basic message passing primitives, **`send`** & **`receive`**

`send` primitive:	sends a message (data) on a specified channel from one process to another,
`receive` primitive:	receives a message on a specified channel from other processes.

- The send primitive has different semantics depending on whether the message passing is *synchronous* or *asynchronous*.
- Message passing can be viewed as extending semaphores to convey data as well as synchronisation.

Synchronous Message Passing

- In synchronous message passing each channel forms a direct link between two processes.
- Suppose process A is sending data to process B:

  When process A executes `send` primitive

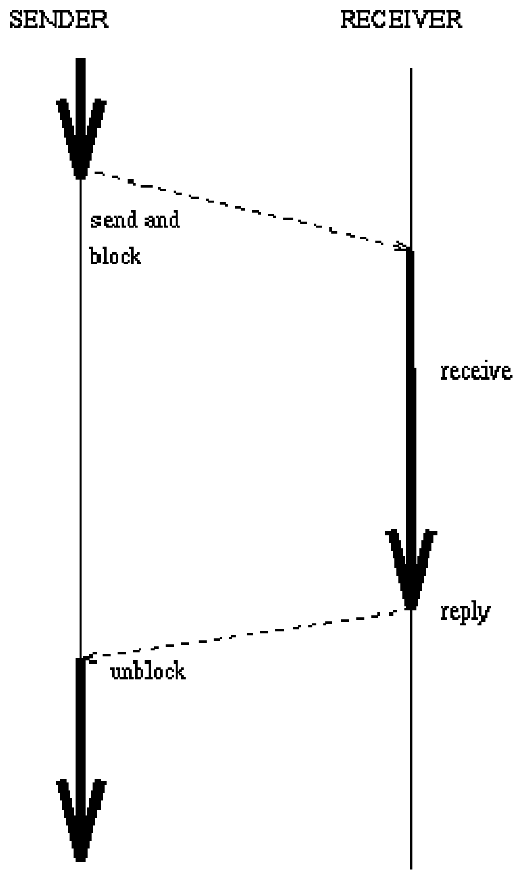
  it waits/blocks until process B executes

  its `receive` primitive.
- Before the data can be transmitted both A & B

  must ready to participate in the exchange.
- Similarly the `receive` primitive in one process

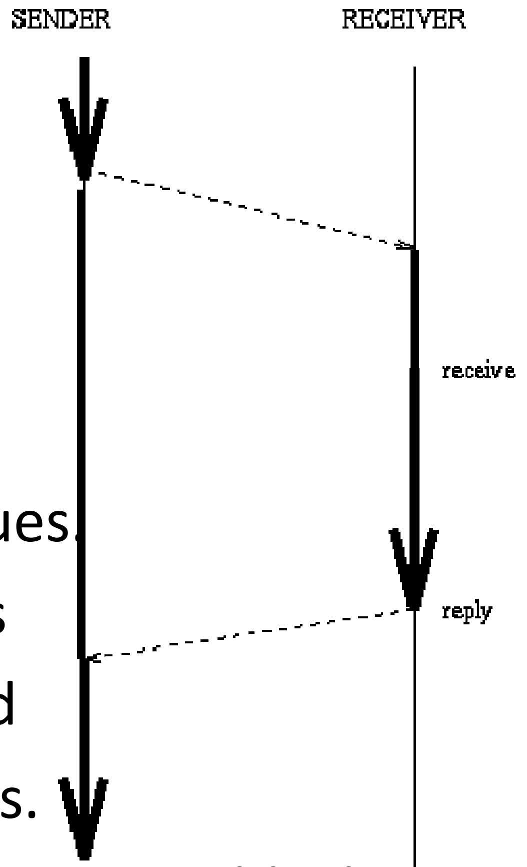
  will block until the send primitive in the

  other process has been executed.



Asynchronous Message Passing

- In asynchronous message passing `receive` has the same meaning/behaviour as in synchronous message passing.
- The `send` primitive has different semantics.
- This time the channel between processes A & B isn't a direct link but a message queue.
- Therefore when A sends a message to B, it is appended to the message queue associated with the asynchronous channel, and A continues.
- To receive a message from the channel, B executes a `receive` removing the message at the head of the channel's message queue and continues.
- If there is no messages in the channel the receive primitive blocks until some process adds a message to the channel.



Additions to Asynchronous Message Passing

- Firstly, some systems implement an `empty` primitive which tests if a channel has any messages and returns true if there are no messages.
- This is used to prevent blocking on a receive primitive when there is other useful work to be done in the absence of messages on a channel.
- Secondly, most asynchronous message passing systems implement buffered message passing where the message queue has a fixed length.
- In these systems the `send` primitive blocks on writing to a full channel.

Types of Processes in Message Passing Programs

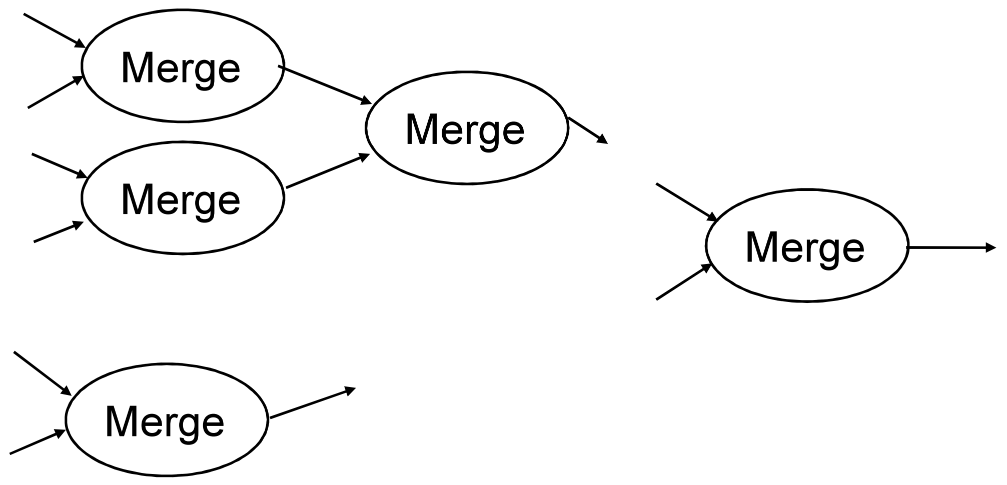
- Filters:
  - These are data transforming processes.
  - They receive streams of data from their input channels, perform some calculation on the data streams, and send the results to their output channels.
- Clients:
  - These are triggering processes.
  - They make requests from server processes and trigger reactions from servers.
  - The clients initiate activity, at the time of their choosing, and often delay until the request has been serviced.

Types of Processes in Message Passing Programs (cont'd)

- Servers:
  - These are reactive processes.
  - They wait until requests are made, and then react to the request.
  - The specific action taken depends on the request, the parameters of the request and the state of the server.
  - The server may respond immediately or it may have to save the request and respond later.
  - A server is a non-terminating process that often services more than one client.
- Peers:
  - These are identical processes that interact to provide a service or solve a problem.

Message Passing Example 1:An Asynchronous Sorting Network Filter

This consists of a series of merge filters.

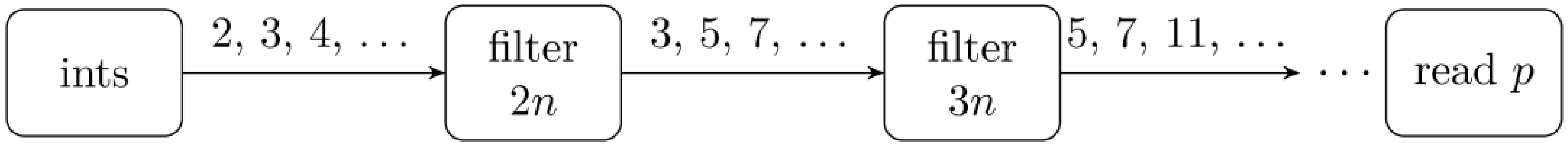


Message Passing Example 1:An Asynchronous Sorting Network Filter

```
const EOS := high (int)          # end of stream marker
op stream1 (x:int), stream2 (x:int), stream3 (x:int)

process merge
        var v1, v2:int
        receive stream1 (v1); receive stream2 (v2)
        do v1 < EOS and v2 < EOS ->
                if v1 <= v2 ->
                        send stream3 (v1)
                        receive stream1 (v1)
                [] v2 < v1 ->
                        send stream3 (v2)
                        receive stream2 (v2)
                fi
        od

        if v1 = EOS ->
                send stream3 (v2)
        [] else ->
                send stream3 (v1)
        fi
        send stream3 (EOS)
end
```

Message Passing Example 2: Synchronous Network of Filters: Sieve of Eratosthenes

- This is a method for finding primes where each prime found acts as a sieve for multiples of it to be removed from the stream of numbers following it.
- The trick is to set up a pipeline of filter processes, of which each one will catch a different prime number.



Message Passing Example 2: Synchronous Network of Filters: Sieve of Eratosthenes

```
op Sieve [L] (x:int)

process p1
        var p:int := 2, i:int
        # send out all odd numbers
        fa i:=3 to N by 2 -> call sieve [1] (i) af
end

process p(i:= 2 to L)
        var p:int, next:int

        receive sieve [i-1] (p)
        do true ->
                receive sieve [i-1] (next)
                # pass on next if it is not a multiple of p
                if (next mod p) != 0 -> call sieve [i] (next) fi
        od              # kick off another process
end
```

- This program will terminate in deadlock.
- How can you stop this? (hint: use a sentinel, see previous filter example).

Example 3(a): Client-Server with Asynchronous Message Passing

- The following is an outline of a resource allocation server and its clients.
- Each client request a resource from a central pool of resources, uses it and releases it when finished with it.
- We assume the following procedures are already written: **`get_unit`** and **`return_unit`** find and return units to some data structure
- And that we have the list management procedures: **`list_insert`**, **`list_remove`** & **`list_empty`**

Example 3(a): Client-Server with Asynchronous Message Passing (cont'd)

```
type op_kind = enum (ACQ, REL)
const N:int := 20
const MAXUNITS:int := 5
op request (index, op_kind, unitid:int)
op reply [N] (unitid:int)

process Allocator
    var avail:int := MAXUNITS
    var index:int, oper:op_kind, unitid:int
    # some initialisation code
    do true ->
      receive request (index, oper, unitid)
        if oper = ACQ ->
            if avail > 0 -> # any available?
                    avail := avail - 1
                    unitid = get_unit ( )
                    send reply [index] (unitid)
            [] avail = 0 -> # none available
                    list_insert (pending, index)
                              # put off for now
            fi
        [] oper = REL->
            if list_empty(pending)-> # postponed?
                    avail := avail+1
                              # nothing postponed
                    return_unit (unitid)
            [] not list_empty (pending) ->
                              # sth postponed
                    index := list_remove(pending)
                              # retrieve it
                    send reply [index] (unitid)
                              # reply to client
            fi                # index with unitid
        fi
    od
end
```

```
process client (i:= 1 to N)
var unit:int

      send request (i, ACQ, 0)# call request
      receive reply[i](unit)
                # rcv reply on my channel
                # with a designated unit

                # use unit and release it
      send request (i, REL, unit)

      . . .
end
```

Example 3(b): Multiple Servers

- This example is a file server with multiple servers.
- When a client wants to access a file, it needs to open the file, access the file (read or write) and then closes the file.
- With multiple servers it is relatively easy to implement a system in which several files can be open concurrently.
- This is done by allocating one file server to each open file.
- A separate process could do the allocation, but as each file server is identical and the initial requests (‘open’) are the same for each client, it’s simpler to have *shared communications channel*.
- This is an example of conversational continuity.
- A client starts a “conversation” with a file server when that file server responds to a general open request.
- The client continues the “conversation” with the same server until it is finished with the file, and hence the file server.

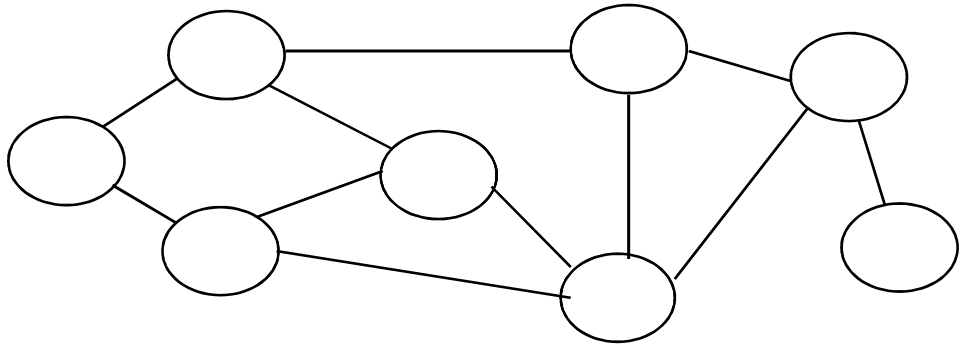
Example 3(b): Multiple Servers (cont’d)

```
type op_kind = enum (READ, WRITE, CLOSE)
type result_type = enum (...)
const N:int := 20, M:int := 8
op open (fname:string[20], c_id:int)     # Cl->Se
op access [M](svce:op_kind, ...)         # Cl->Se
op open reply [N] (s_id:int)             # Se->Cl
op access_reply[N](res:result_type)      # Se->Cl

process File_Server (i:= 1 to M)
    var svce:op_kind, clientid:int
    var fname:string [20]
    var more:bool := false

    do true ->
      receive open (fname, clientid)
      send open_reply [clientid] (i)
      more := true

      do more = true ->
          receive access [i] (svce, ...)
          if svce = READ -> # process read req
          [] svce = WRITE-> # process write req
          [] svce = CLOSE-> # close file
          more := false
          fi

          send access_reply [clientid] (results)
      od
    od
end
```

```
process client (i:= 1 to N)
var server:int # server channel’s id
  send open(“myfile”,i)
      # i wants to open‘myfile’
  receive open_reply [i] (server)
      # reply from server
  send access [server] (...)
      # reply comes on server channel
  receive access_reply [i] (results)
      # reply on my channel with results
end
```

Asynchronous Heartbeat Algorithms

- Heartbeat algorithms are a typical type of process interaction between peer processes connected together by channels.
- They are called heartbeat algorithms because the actions of each process is similar to that of a heart; first expanding, sending information out; and then contracting, gathering new information in.
- This behaviour is repeated for several iterations.
- An example of an asynchronous heartbeat algorithm is the algorithm for computing the topology of a network.

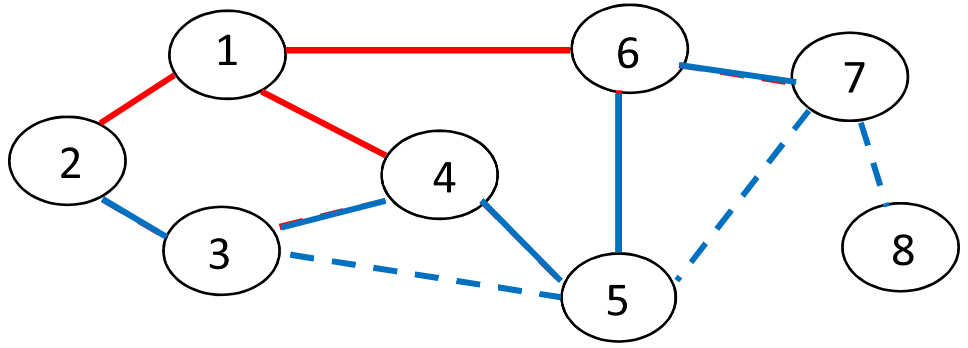


Message Passing Algorithms Example 4: Asynchronous Heartbeat Algorithm for Computing Network Topology

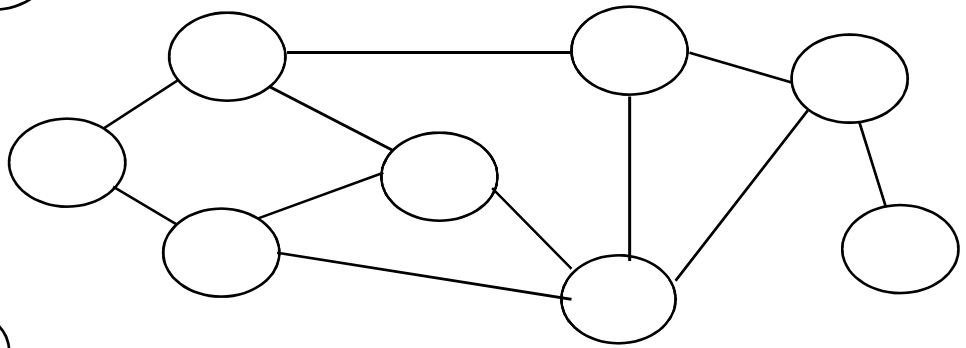
- Each node has a processor and initially only knows about the other nodes to which it is directly connected.
- Algorithm goal is for each node to determine the overall n/w topology.
- The two phases of the heartbeat algorithm are:
  1. transmit current knowledge of network to all neighbours, and
  2. receive the neighbours' knowledge of the network.
- After the first iteration the node will know about all the nodes connected to its neighbours, that is within two links of itself.
- After the next iteration it will have transmitted, to its neighbours, all the nodes with 2 links of itself; and it will have received information about all nodes with 2 links of its neighbours, that is within 3 links of itself.
- In general, after *i* iterations it will know about all nodes within *(i+1)* links of itself.

Ex 4: Asynchronous Heartbeat Algorithm for Computing Network Topology: Algorithm Operation

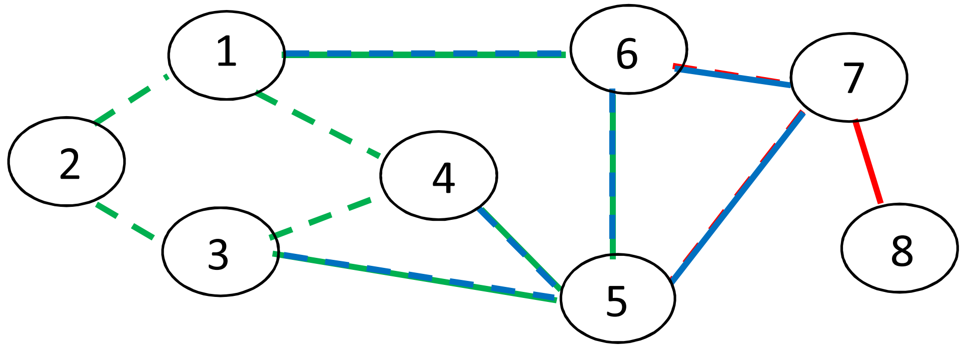
Firstly from Node 1's Point of View



....and Node 1 is done!



Next from Node 8's Point of View



....and Node 8 is done!

Example 4: Asynchronous Heartbeat Algorithm for Network Topology(Cont’d)

- How many iterations are necessary?
- Since the network is connected, every node has at least one neighbour.
- If we store the known network topology at any given stage in an *nxn* matrix **`top`** where

  `top`[*i,j*] = true if a link exists between node *i* and *j*,

  then a node knows about the complete topology of the network when every row in top has at least one true value.
- At this point the node needs to perform one more iteration of the heartbeat algorithm to transmit any new information received from one neighbour to its other neighbours.

Ex 4: Asynchronous Heartbeat Algorithm for Network Topology(Cont’d)

```
op topol[N](sender:int,done:bool,top[N,N]:bool)

process node_heartbeat (i := 1 to N)
    var links [N]:bool
    var active[N]:bool #neighbors still active
    var top [N,N]:bool := ([N * N] false)
    var row_ok:bool
    var done:bool := false
    var sender:int, qdone:bool, newtop [N,N]:bool
    # initialise links to neighbours
    ...
    # initialise active my row to my neighbors
    active := links
    top [i,1:N] := links
    do not done ->
    # send local knowledge to all neighbors
      fa j:= 1 to N st links[j] ->
        send topol [j] (i, done, top)
      af
    # receive local knowledge of the neighbors
    # and or it with our own knowledge
      fa j:= 1 to N st links[j]->
        receive topol[j](sender, qdone, newtop)
        top := top or newtop
        if qdone -> active [sender] := false fi
      af
      # check if all rows in top have a ‘true’
      done := true
      fa j:= 1 to N st done ->
        row_ok := false
        fa k:= 1 to N st not row_ok ->
          if top [j,k]= true -> row_ok=true fi
        af
        if row_ok = false -> done = false fi
      af
    od
```

Main Loop

```
# send full topology to all active n’bors

fa j:= 1 to N st active [j] ->
    send topol [j] (i, done, top)
af

# receive message from each to clear channels
fa j:= 1 to N st active [j] ->
    receive topol[j](sender, qdone, newtop)
af
end
```

Ex 4: Asynchronous Heartbeat Algorithm for Network Topology(Cont'd)

- If *m* is the maximum number of neighbours any node has, and *D* is the n/w diameter[1], then the number of messages exchanged must be less than $2n * m * (D+1)$.
- A centralised algorithm, in which top was held in memory shared by each process, requires only 2n messages. If m and D are small relative to n then there is relatively few extra messages.
- In addition, these messages must be served sequentially by the centralised server. The heartbeat algorithm requires more messages, but these can be exchanged in parallel.

[1] i.e. the max. value of the minimum number of links between any two nodes

Ex 4: Asynchronous Heartbeat Algorithm for Network Topology(Cont'd)

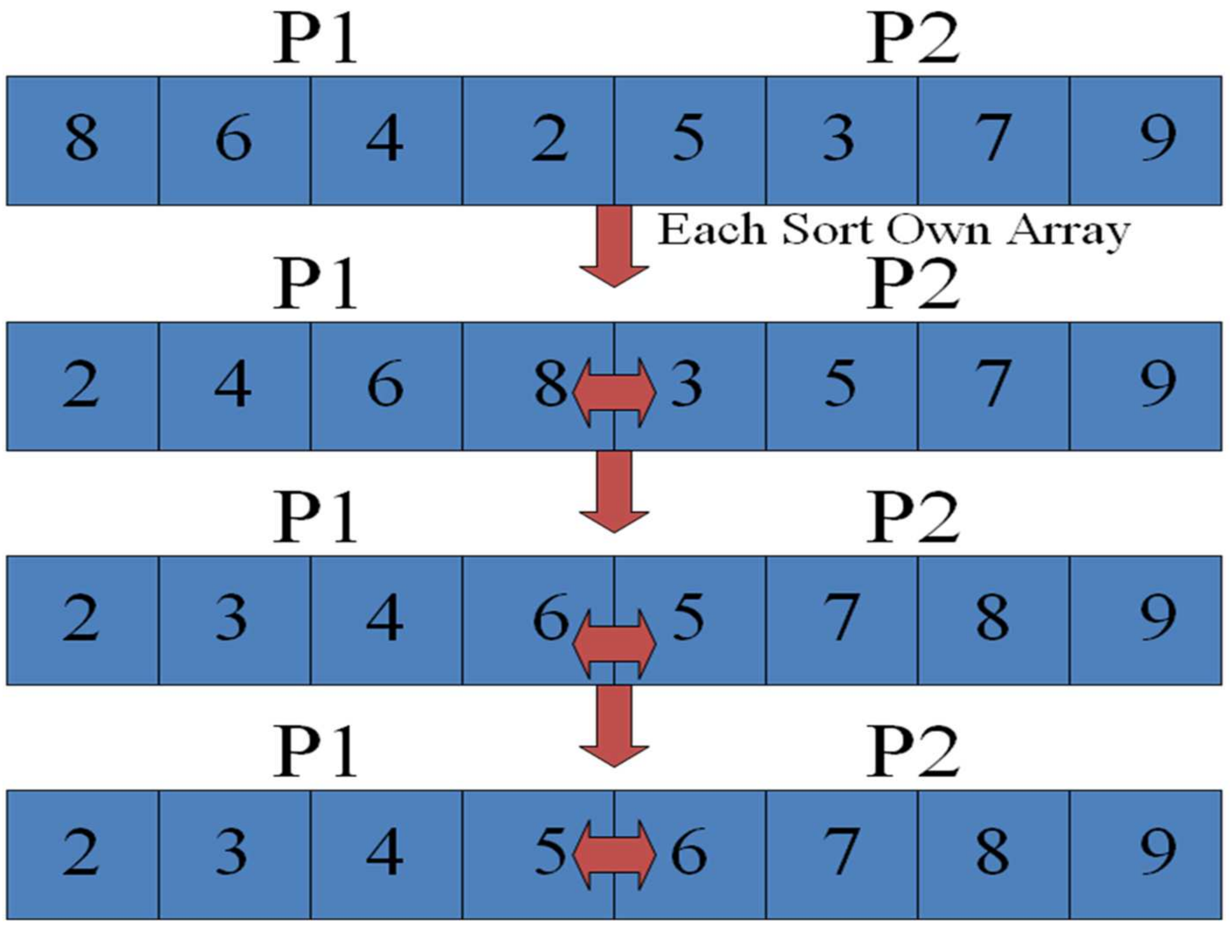
- All heartbeat algorithms have the same basic structure; send messages to neighbours, and then receive messages from neighbours.
- A major difference between the different algorithms is termination. If the termination condition can be determined locally, as above, then each process can terminate itself.
- If however, the termination condition depends on some global condition, each process must iterate a worst-case number of iterations, or communicate with a central controller monitoring the global state of the algorithm, and issues a termination message to each process when required.

Example 5: Synchronous Heartbeat Algorithm: Parallel Sorting

- To sort an array of n values in parallel using a synchronous heartbeat algorithm, we need to partition the n value equally among the processes.
- Assume that we have 2 processes, P1 and P2, and that n is even.
- Each process initially has n/2 values and sorts these values into non descending order, using a sequential sort algorithm.
- Then at each iteration P1 exchanges it largest value with P2’s smallest value, and both processes place the new values into the correct place in their own sorted list of numbers.
- Note: since both sending & receiving block in synchronous message passing, P1 and P2 cannot execute the send, receive primitives in the same order (as could in asynchronous message passing).

Example 5(a): Synchronous Heartbeat Algorithm: Parallel Sorting: Algorithm Operation (Cont'd)

- Demonstration of Odd/Even Sort for 2 Processes:



Example 5(a): Synchronous Heartbeat Algorithm: Parallel Sorting: Code

```
op channel_1 (x:int)
op channel_2 (x:int)

process P1
    var a[N/2]:int, new:int
    var largest:int := N/2

    # sort a into non-descending order

    call channel_2 (a[largest])
    receive channel_1 (new)

    do a[largest] > new ->
      a[largest] := new
      fa i:=largest downto 2 st a[i] > a[i-1] ->
          a[i] :=: a[i-1] #swap
      af
      call channel_2(a[largest])
      # send my largest along ch_2
      receive channel_1 (new)
      # rcv its smallest along ch_1
    od
end
```

```
process P2
    var a[N/2]:int, new:int
    var largest:int := N/2;

    # sort a into non-descending order

    receive channel_2 (new)
    call channel_1 (a[1])

    do a[1] < new ->
      a[1] := new
      fa i:= 2 to largest st a[i] < a[i-1] ->
          a[i] :=: a[i-1]#swap
      af
      receive channel_2 (new)
      # rcv its largest along ch_2
      call channel_1 (a[1])
      # send my smallest along ch_1
    od
end
```

Example 5(a): Synchronous Heartbeat Algorithm: Parallel Sorting: (Cont'd)

- Can extend this to k processes by initially dividing the array so that each process has n/k values which it sorts using a sequential algorithm.
- Then we can sort the n elements by repeated applications of the two process compare and exchange algorithm.
- On odd-numbered applications:
  - Every odd-numbered process acts as P1, and every even numbered process acts as P2.
  - Each odd numbered process P[i] exchanges data with process P[i+1].
  - If k is odd, then P[k] does nothing on odd numbered applications.
- On even-numbered applications:
  - Even-numbered processes act as P1, odd numbered processes act as P2.
  - P[1] does nothing, and P[k] does nothing, even if k is even.

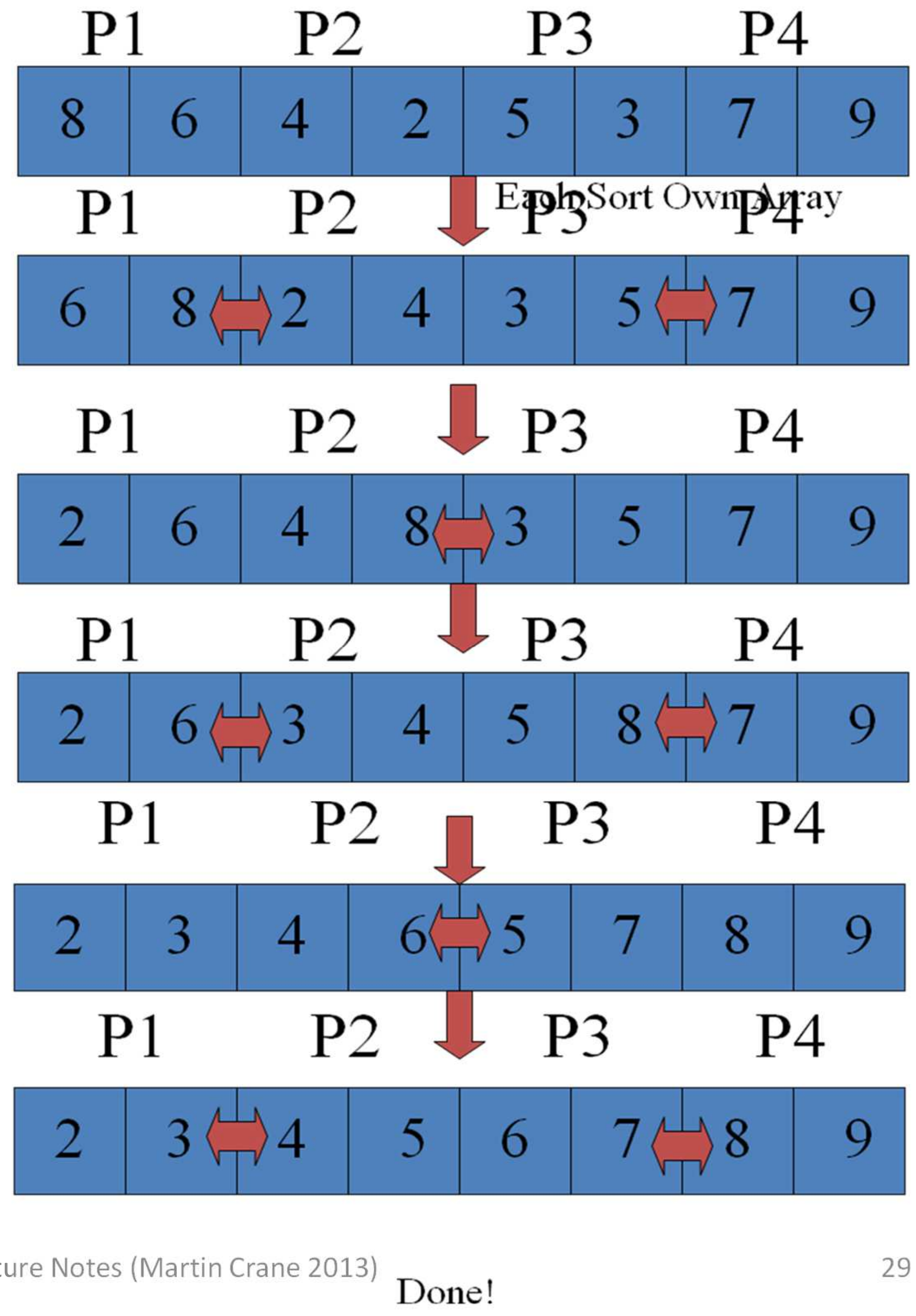
Ex 5 (a): Synchronous Heartbeat Algorithm: Parallel Sorting: (Cont'd)

- The SR algorithm for odd/even exchange sort on n processes can be terminated in many ways; two of which are:

1. Have a separate controller process who is informed by each process, each round, if they have modified their n/k values.
   - If no process has modified its list then the central controller replies with a message to terminate.
   - This adds an extra 2k messages overhead per round.
2. Execute enough iterations to guarantee that the list will be sorted. For this algorithm it requires k iterations.

Example 5(a): Odd/Even Exchange Sort: Algorithm Operation (Cont'd)

- Demonstration of *Odd/Even Exchange Sort* for k Processes:



Ex. 5(b) Odd/Even Exchange Sort for n Processes in Java

```
public class OddEvenSort {
      public static void main(String a[]){
         int i;
         int array[] = {12,9,4,99,120,1,3,10,13};
         odd_even(array,array.length);
         }
public static void odd_even(int array[], int n){
     for (int i = 0; i < N/2; i++){

/* 1st evens: all these can happen in parallel */
     for (int j = 0; j+1 < n; j += 2)
       if (array[j] > array[j+1]) {
         int T = array[j];
         array[j] = array[j+1];
         array[j+1] = T;
       }

/* Now odds: all these can happen in parallel */
      for (int j = 1; j+1 < array.length; j += 2)
       if (array[j] > array[j+1]) {
         int T = array[j];
         array[j] = array[j+1];
         array[j+1] = T;
       }
   }
  }
}
```

Guarded Synchronous Message Passing

- Since both the `send` and `receive` primitives in synchronous message passing block, it is generally desirable not to call them if you have other useful things to be done.
- An example of this is the Decentralised Dining Philosophers Problem where each philosopher has a waiter.
  - It is the waiter processes that synchronises access to the shared resources (forks).
  - When a resource (fork) has been used it is marked as dirty.
  - When a waiter is requested for a fork, it checks if it is not being used and it is dirty.
  - It then cleans the fork and gives it to the requesting waiter.
  - This protocol prevent a philosopher from being starved by the waiter removing one fork before the other fork arrives.
  - This algorithm is also called the *hygienic philosophers algorithm*.

Guarded Synchronous Message Passing (Cont'd)

- The guarded form of the receive command in SR is

  ```
  in op_name st expression -> ... ni
  ```

- and the nondeterministic version is

  ```
      in op_name1 st expression1 -> ...
      [] op_name2 st expression2 -> ...
      [] op_name3 st expression3 ->
      [] ...
      [] else -> ...
      ni
  ```

- The `else` block is executed when there is no non blocking `in` statement.

Ex 5: Hygenic Philosphers

```
op fork[5]( )
op phil_hungry[5](),phil_eat[5]( ),phil_full[5]()
process waiter (i := 1 to 5)
    var eating:bool:= false, hungry:bool := false
    var haveL, haveR:bool
    var dirtyL:bool:= false, dirtyR:bool := false

    if i = 1 -> haveL := true; haveR := true;
                    dirtyL:=true; dirtyR:= true
    [] i >1 and i < 5->haveL:=false;haveR:= true;
                    dirtyR:= true
    [] i = 5 -> haveL := false; haveR := false
    fi

    do true ->
        in phil_hungry [i] ( ) ->
                # receive a call from my philo
          hungry:=true  # set 'hungry' as true

        [] fork [i mod 5] ( ) st
                # rcv a call from lh side waiter
          haveL and not eating and dirtyL ->
                # not eating/using it
          haveL := false; dirtyL := false
                # clean & return my lh fork

        [] fork [(i+1) mod 5] ( ) st
                # rcv a call from rh side waiter
          haveR and not eating and dirtyR ->
                # not eating/using it
          haveR := false; dirtyR := false
                # clean & return my rh fork

        [] phil_full [i] ( ) ->
                # rcv a 'full' call from my philo
          eating := false # not hungry
        [] else -> # can do some things at random
          if hungry and haveL and haveR ->
          # have all my philo needs to eat
                hungry := false; eating := true
                dirtyL := true; dirtyR := true
                call phil_eat [i] ( )
          # tell my philo to eat

          [] hungry and not haveL ->
          # have all except lh form
                call fork [i mod 5] ( )
          # call lh waiter for my fork
          # block until call comes
                haveL := true

          [] hungry and not haveR ->
          # have all except rh fork
                call fork [(i+1) mod 5]
          # call rh waiter for my fork
                haveR := true
          fi
        ni
    od
end

process philosopher (i:= 1 to 5)
    do true ->
        call phil_hungry [i] ( )
        # tell my waiter 'I'm hungry!'
        receive phil_eat [i] ( )
        # block until this reply comes, then eat
        call phil_full [i] ( )
        # tell my waiter 'I'm full!' then think…
    od
end
```

The duality between Monitors and Message Passing

- Have already seen relationship between semaphores and monitors.
- As message passing is just another solution concurrent processing problem, should be a relationship between message passing & monitors.

Monitor-Based Programs	*Message-Based Programs*
permanent variables	local server variables
procedure identifiers	request channels and operation kinds
procedure call	send request; receive reply
monitor entry	receive request
procedure return	send reply
_wait statement	save ‘pending’ request
_signal statement	retrieve and process ‘pending’ request
procedure bodies	arms of “case” statement on operation kinds

cf Reader-Writer Problem c.f. ASMP-Client Server

Message Passing in Java

- Java has no built-in support for message passing
- But it does contain as standard the **`java.net package`**
- This supports low-level datagram communications & high level stream-based communications with sockets.
- Java is particularly suited to the client/server paradigm. Here is a remote file reader implemented in Java.

File Reader Server Code

```
import java.io.*;
import java.net.*;
public class FileReaderServer {
    public static void main( String[] args )      {
      try {
          ServerSocket listen = new ServerSocket (9999); // create server socket, listen on 9999
          while (true)  {
            System.out.println ("waiting for connection"); // blocks till client reqs connection
            Socket socket = listen.accept ( ); // applies buffering to some char inputstream
            BufferedReader from_client =
               new BufferedReader(new InputStreamReader (socket.getInputStream ( ));
            PrintWriter to_client = new PrintWriter(socket.getOutputStream ( ));
            String filename = from_client.readLine ( );
            File inputFile = new File (filename);
                    // first check that file exists, if not close up
            if (!inputFile.exists ( )) {
                to_client.println ("cannot open " + filename);
                to_client.close ( );
                from_client.close ( );
                socket.close ( );
                continue; }
            // read lines from file & send to the client
            System.out.println ("reading " + filename);
            BufferedReader input =new BufferedReader (new FileReader (inputFile));
            String line;
            while ((line = input.readLine ( )) != null)
                to_client.println (line);
            to_client.close ( );
            from_client.close ( );
            socket.close ( );
                    }
          }
      catch (Exception ex){
          System.err.println (ex);
          }
    }
}
```

File Reader Client Code

```
import java.io.*;
import java.net.*;
public class Client {
    public static void main( String[] args )      {
      try {
          // read in command line arguments
          if (args.length != 2) {
            System.out.println ("need host and filename");
            System.exit (1);
          }
          String host = args [0];
          String filename = args [1];

          // open socket to host on port 9999
          Socket socket = new Socket (host, 9999);

          // applies buffering to some character inputstream
          BufferedReader from_server =new
                    BufferedReader (new InputStreamReader ( socket.getInputStream ( )));
          PrintWriter to_server = new PrintWriter (Socket.getOutputStream ( ));

          // send filename to server, read & print lines from server until its closes connection
          to_server.println (filename);
          to_server.flush ( );

          String line;
          while ((line = from_server.readLine ( )) != null)
                    System.out.println (line);
          }
          catch (Exception ex);
          {
                    System.err.println (ex);
          }
    }
}
```

Alternative Communication Methods: Remote Procedure Call (RPC)

- Message passing is powerful enough to handle all four kinds of concurrent processes (filters, clients, servers and peers).
- However, it can be cumbersome when coding client/server programs because information in channels flows in one direction and clients and servers require a two-way information flow between them.
- Therefore, there have to be two explicit message exchanges on two different channels.
- In addition each client needs a different reply channel leading (potentially) to a lot of channels and send/receive statements.
- Remote Procedure Calls (RPC) provide an ideal notation for programming client/server systems.
- RPCs are a combination of some of the ideas of the monitor and synchronous message passing approaches.
- RPCs are a two-way communication mechanism where the client invokes an operation in the server.

Remote Procedure Call (RPC) (Cont’d)

- The caller of an RPC blocks until the server operation has been executed to completion and has returned its results.
- As far as the client is concerned, RPCs resemble sequential procedure calls both syntactically and semantically.
- The client does not care if the RPC is serviced by an operation on the same processor or another processor.
- Each operation is serviced by a procedure in the server.
- Each invocation to an operation is handled by creating a new process to handle each call.

Remote Procedure Call (RPC) (Cont'd)

- The RPC programming component is the *module*. A module contains both processes and procedures.
- Processes in a module can call procedures within the module, or call procedures in other modules using the RPC mechanism.
- The important point about modules is that each module is allowed to exist in a different address space. (Processes in the same address space are called lightweight threads.)
- A module has two sections:

1. a specification part that contains the definitions of the publicly accessible procedures, and
2. a body part that contains the definition of these procedures, as well as local data, initialisation code, local procedures and local processes

Example 6: Implementing Stacks with RPCs.

```
module Stack
    type result = enum (OK, OVERFLOW, UNDERFLOW)
    op push (item:int) returns r:result
    op pop (res item:int) returns r:result

body Stack (size:int)
    var store [1:size]:int, top:int := 0
    proc push (item) returns r
          if top < size ->
              store[++top] := item
              r := OK
          [] top = size ->
              r := OVERFLOW
          fi
    end
    proc pop (item) returns r
          if top > 0 ->
              item := store[top--]
              r := OK
          [] top = 0 ->
              r := UNDERFLOW
          fi
    end
end Stack
```

```
resource Stack_User

    import Stack
    var x: Stack.result
    var s1, s2: cap Stack
    var y:int

    s1 := create Stack(10)
    s2 := create Stack(20)
    ...
    s1.push (4); s1.push (37); s2.push (98)
    if (x := s1.pop(y)) != OK -> ... fi
    if (x := s2.pop(y)) != OK -> ... fi
    ...

end
```

Synchronisation in Modules

- The RPC is purely a communication mechanism to allow for the simpler expression of client/server programs.
- There is some degree of implicit synchronisation in that the caller process is blocked until the remote process is completed.
- We also need some means to synchronise the processes within the module.
- This allows us to assume that processes within a module execute concurrently.
- The simplest way is to use semaphores.
- As an example consider following timer server module (in pseudo-SR) providing timing services to clients via RPCs.

Example 7: Time Server with RPCs.

```
module Time_Server
    op get_time ( ) returns time:int
    op delay (interval:int, id:int)

body Time_Server
    var time_of_day:int := 0
    sem m := 1 # mutual exclusion semaphore
    sem d[N]:=([N] 0) # private delay semaphores

    proc get_time ( ) returns time
      time := time_of_day
    end

    proc delay (interval, id)
      var wake_time:int:=time_of_day + interval

      P(m)
      priority_insert_list(wake_list,wake_time,id)
      V(m)
      P(d[id])
    end

 process clock
  var wake_id:int

  do true -> # start hardware timer
   P(m) # wait for interrupt
   time_of_day++
   do time_of_day>=first_entry(wake_list)
     wake_id := remove_list (wake_list)
     V(d[wake_id])
   od
   V(m)
  od
 end
end Time_Server
```

Rendezvous

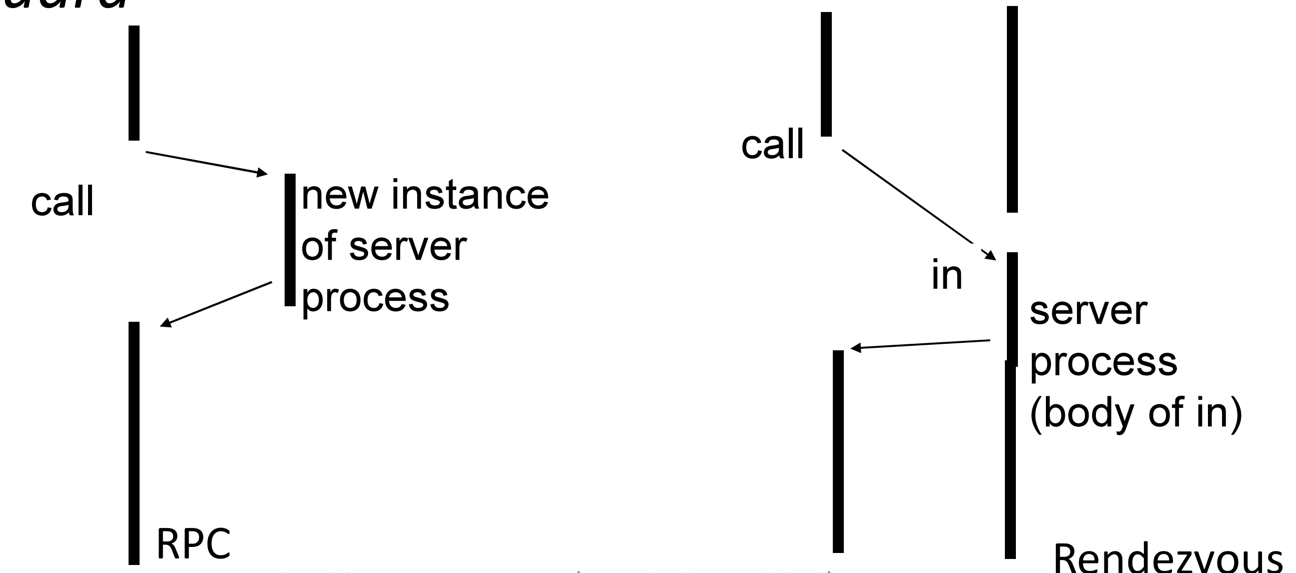
- The RPC mechanism is simply an intermodule communication mechanism. In the module we still need to provide synchronisation.
- The rendezvous mechanism combines the actions of servicing a call with the processing of the information conveyed by the call.
- With rendezvous, a process exports operations that can be called by other processes.
- As in RPC, a process can invoke an operation by calling the operation.
- The key difference between RPC and rendezvous is that the client call to the operation is serviced by an existing server process.
- The server process rendezvous with the client calling the operation by means of executing an **`in`** statement.
- So a server uses the **`in`** statement to wait for, then act on a <u>single call</u>, servicing calls one-at-a-time rather than concurrently.

Rendezvous (Cont'd)

- In SR rendezvous is accomplished by means of the **`in`** statement.
- The general form of the in statement is:

```
in operation (formals_1)
     st sync_expr by sched_expr -> block
[] operation (formals_2)
     st sync_expr by sched_expr -> block
...
[] else -> block
ni
```

- Each arm of **`in`** is a *guarded operation*; the part before the -> is called the *guard*



Rendezvous (Cont'd)

- We have seen the synchronisation expression already with guarded synchronised message passing.
- Since the scope of the operation's formals is the entire guarded operation[1], the synchronisation/scheduling expression can depend on the formals' value & hence on the values of the arguments of the call.
  1. If there is no scheduling expression and several guards are satisfied (the synchronisation expression is true and there is a call to the operation) one of them is chosen nondeterministically.
  2. Of the non-deterministically chosen guard, if there are several invocations, and no scheduling expression, the **`in`** statement services the oldest invocation that makes the guard succeed.
  3. If there is a scheduling expression, then the **`in`** statement services the invocation of the guard which minimises the scheduling expression.

[1] Each arm of **`in`** is a *guarded operation*; the part before the -> is called the *guard*

Example 8: Time Server using Rendezvous

```
module Time_Server
   op get_time ( ) returns time:int
   op delay (wake_time:int)
   op tick ( )         # called by clock interrupt handler

body Time_Server
   process time
       var time_of_day:int := 0

       do true ->
        in get_time ( ) returns time ->
              time := time_of_day

        [] delay (wake_time)
              st wake_time <= time_of_day -> skip

        [] tick ( ) -> time_of_day++
        ni
       od
   end
end
```

Example 9: Shortest Job Next Allocator using Rendezvous.

```
module SJN_Allocator
   op request (time:int), # request for a certain length of time
   op release ( )

body SJN_Allocator
   process sjn
        var free:bool := true

        do true ->
         in request (time) st free by time ->
              free := false # case 3 above: minimise scheduling expr

         [] release ( ) ->
              free := true
         ni
        od
   end
end
```

Example 10: Bounded Buffer using Rendezvous

```
module Bounded_Buffer
   op deposit (item:int)
   op fetch ( ) returns item:int

body Bounded_Buffer (size:int)
   var buffer [1:size]:int
   var count:int := 0, front:int := 0, rear:int := 0

   process worker
        do true ->
         in deposit (item) st count < size ->
               buffer [rear] := item
               rear := (rear +1) mod size
               count++
         [] fetch ( ) returns item st count > 0 ->
               item := buffer [front]
               front := (front + 1) mod size
               count--
         ni
        od
   end
end
```

- In this example the synchronisation expressions are used to prevent overflow/underflow occurring in the buffer.

Lecture 6: Message Passing Interface

- Introduction
- The basics of MPI
- Some simple problems
- More advanced functions of MPI
- A few more examples