

Message Passing

Up to now the concurrent programming constructs (critical sections, semaphores, monitors) have been based on shared memory systems. However with *network architectures* and 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 local memory of the processor. If the two interacting processes are allocated to different processors, then the channel between the two processes is mapped to a physical communications medium between the corresponding two processors.

There are two basic *message passing primitives*, the **send** primitive which sends a message (data) on a specified channel from one process to another, and the **receive** primitive which 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 the **send** primitive it will wait (block) until process B executes its **receive** primitive. Before the data can be transmitted both processes must be 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* the **receive** primitive has the same semantics (meaning/behaviour) as in synchronous message passing. The **send** primitive has different semantics. This time the channel between the processes is not a direct link between the two processes, but is a *message queue*. Therefore when process A **sends** a message to process B, the message is appended to the message queue associated with the asynchronous channel, and process A continues to execute.

To receive a message from the channel, process B executes the **receive** primitive which removes the message at the head of the message queue associated with the channel, and continues. If there are no messages

in the channel the **receive** primitive blocks until some process adds a message to the channel.

There are two interesting 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.

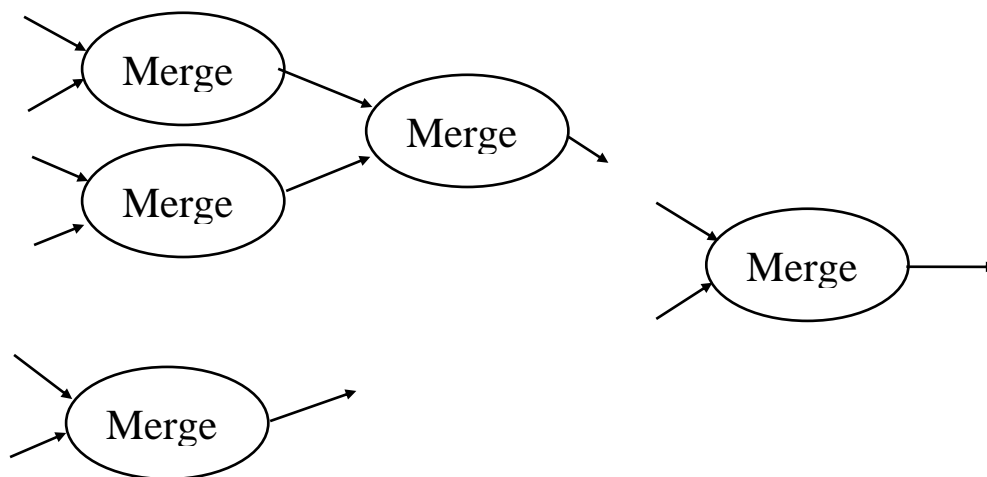
Types of Processes in Message Passing Programs

- 1) *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.
- 2) *Client*: These are triggering processes. Clients 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.

- 3) *Servers*: These are reactive processes. They wait until requests are made, and then reacts 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.
- 4) *Peers*: These are identical processes that interact to provide a service or solve a problem.

An Asynchronous Sorting Network Filter

This consists of a series of merge filters.



```
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
```

Synchronous Network of Filters: Sieve of Eratosthenes

The Sieve of Eratosthenes is a method for finding primes where each prime that is 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.

```
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).

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 it is finished with it. We will assume the following procedures are already written: `get_unit` and `return_unit` find and return units to some data structure; and `list_insert`, `list_remove` and `list_empty` are list management procedures.

```
type op_kind = enum (ACQUIRE, RELEASE)
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, operation:op_kind, unitid:int

  # some initialisation code

do true ->
  receive request (index, operation, unitid)
  if operation = ACQUIRE ->
    if avail > 0 -> # see if 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
```

```

[] operation = RELEASE->
  if list_empty (pending)-> # any 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, ACQUIRE, 0) # "call" request
  receive reply [i] (unit) # rcv reply on my channel
  # with a designated unit
  # use unit

  send request (i, RELEASE, unit)

  ...
end

```

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 close 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. The allocation could be done by a separate process, but since each file server is identical and the initial requests (open file) are the same for each client, a simpler solution is to have *shared communications channel*.

```
type op_kind = enum (READ, WRITE, CLOSE)
type result_type = enum (...)
const N:int := 20, M:int := 8
op open (file_name:string [20], clientid:int) # C->S
op access [M] (service:op_kind, ...) #client -> server
op open_reply [N] (serverid:int) # server -> client
op access_reply[N] (results:result_type) #server->client

process File_Server (i:= 1 to M)
  var service: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] (service, ...)
    if service = READ -> process read request
    [] service = WRITE-> process write request
    [] service = CLOSE ->
        close file
        more := false
    fi
    send access_reply [clientid] (results)
od
od
end

process client (i:= 1 to N)
    var server:int # the id of the server channel

    send open ("myfile", i) # i wants to open 'myfile'
    receive open_reply [i] (server) # reply from server
    send access [server] (...) # reply on server channel
    receive access_reply [i] (results) # reply on my
    ... # channel with results
end

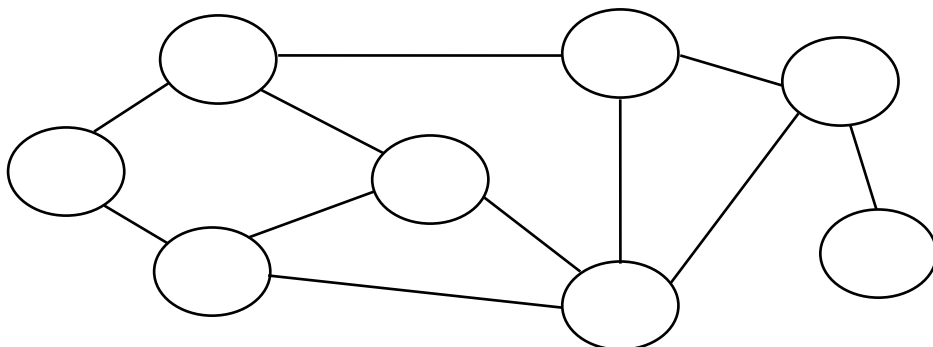
```

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.

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.



Each node has a processor and initially only knows about the other nodes to which it is directly connected. The goal of the algorithm is for each node to determine the overall network topology.

The two phases of the heartbeat algorithm are:

- i) transmit current knowledge of network to all neighbours, and

ii) 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 two links of itself; and it will have received information about all nodes with two links of its neighbours, that is within three links of itself. In general, after i iterations it will know about all nodes within $(i+1)$ links of itself.

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 $n \times n$ 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.

```

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

process node_heartbeat (i := 1 to N)
  var links [N]:bool
  var active[N]:bool#n'bors who are 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 & the row for 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 topology [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 topology[j] (sender, qdone,newtop)
      top := top or newtop
      if qdone -> active [sender] := false fi
    af
    # check if all rows in top have a true entry
    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 complete topology to all neighbours who
# are still active
fa  j:= 1 to N st active [j] ->
    send topology [j] (i, done, top)
af

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

```

If m is the maximum number of neighbours any node has, and D is the diameter of the network, i.e. the max. value of the minimum number of links between any two nodes, 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.

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, then each process must iterate a worst-case number of iterations, or

communicate with a central controller which monitors the global state of the algorithm, and issues a termination message to each process when it is required.

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 its 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.

An important point to note is that, since both *sending* and *receiving* are blocking in synchronous message passing, $P1$ and $P2$ cannot execute the **send** and **receive** primitives in the same order (as they would be able to in asynchronous message passing).

```

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;

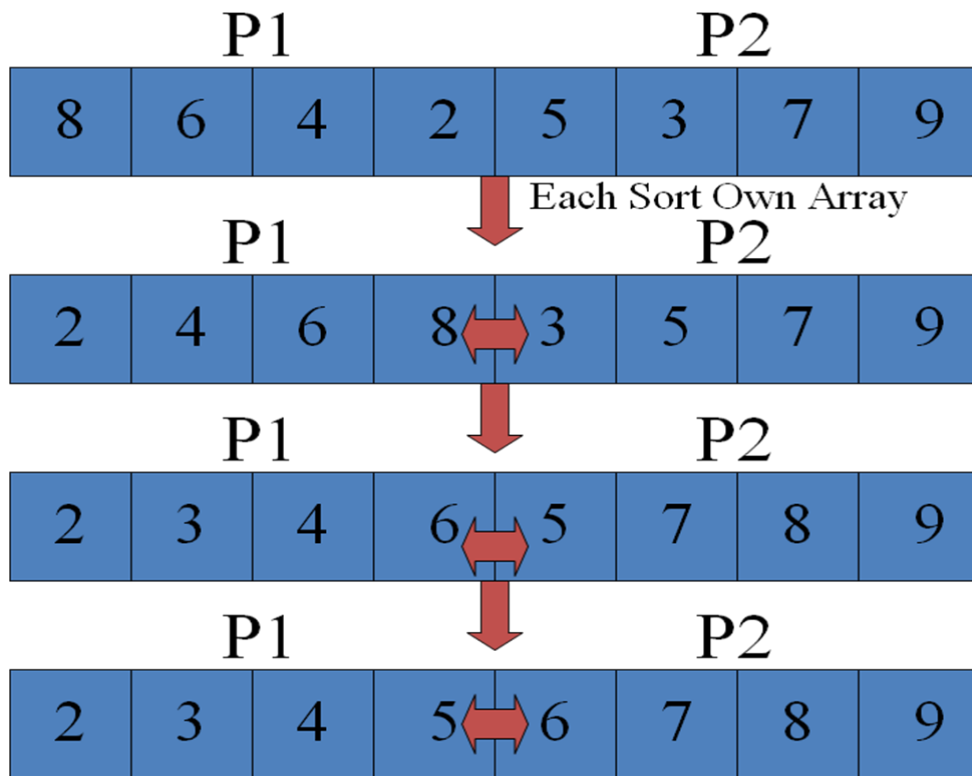
  # sort a into non-descending
  order

  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

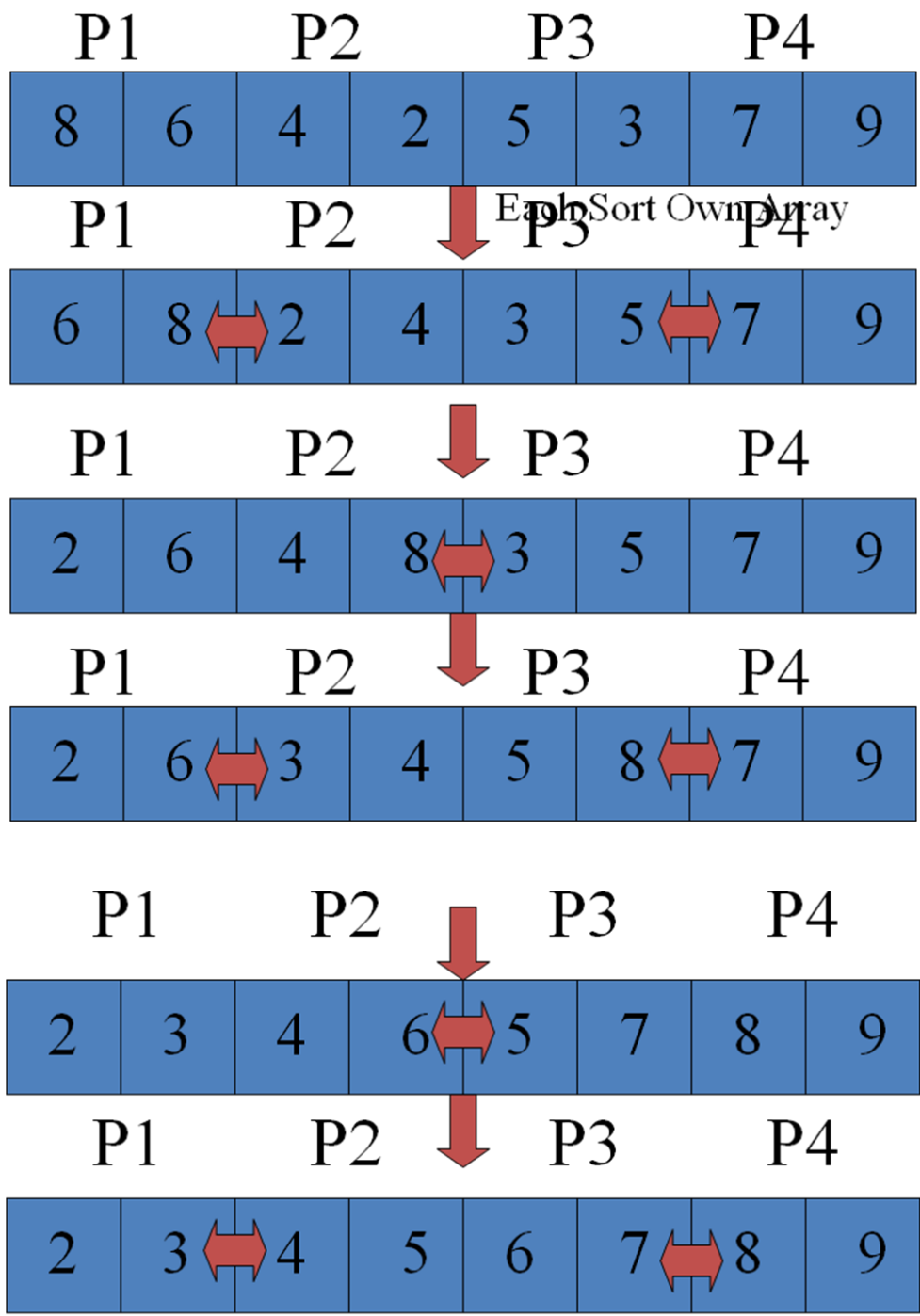
```

Demonstration of Odd/Even Sort for 2 Processes



We 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 $P1$, and odd-numbered processes act as $P2$. $P[1]$ does nothing, and $P[k]$ does nothing, even if k is even.



Done!

This is called the *odd/even exchange sort* algorithm and can be extended up to $k = n$. A piece of sample Java code for this is shown below:

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;
                    }
                }
        }
    }
}
```

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

- a) 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.

- b) Execute enough iterations to guarantee that the list will be sorted. For this algorithm it requires k iterations.

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 synchronise 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*.

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.

```

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

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

    [] fork [(i-1) 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] ( ) -> # receive a 'full' call from my philo
      eating := false # set my 'hungry' variable as false

```

```

[] else -> # can do some of the following housework at random
if hungry and haveL and haveR -> # have all my philo needs
    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-1)mod 5] ( ) # call lh waiter for my fork
                                # block until call comes

    haveL := true

[] hungry and not haveR -> # have all except rh form
    call fork [(i+1) mode 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

    # eat

    call phil_full [i] ( ) # tell my waiter 'I'm full!'

    # think
od
end

```

The duality between Monitors and Message Passing

We have already seen the relationship between semaphores and monitors. Since message passing is just another approach to the problem of concurrent processing there should be a relationship between message passing and monitors.

	<i>Monitor-Based Programs</i>	<i>Message-Based Programs</i>
Client Side	permanent variables	local server variables
	procedure identifiers	<i>request</i> channels and operation kinds
	procedure call	send <i>request</i> ; receive <i>reply</i>
Server Side	monitor entry	receive <i>request</i>
	procedure return	send <i>reply</i>
	_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

Notes Page 49

Notes Page 79/80

Message Passing in Java

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

```
// File Reader Server
```

```
import java.io.*;
import java.net.*;

public class FileReaderServer
{
    public static void main (String[ ] args)
    {
        try
        {
            // create server socket and listen on port 9999
            ServerSocket listen = new ServerSocket (9999);

            while (true)
            {
                System.out.println ("waiting for connection");
                // this blocks until client reqs connection on port
                Socket socket = listen.accept ( );

                // applies buffering to some character 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

```
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 server closes the 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);
}
}
}

```

The server is executed by:

```
java FileReaderServer
```

```
/* server outputs */
"waiting for connection"
```

and the client is started by:

```
java Client hostname filename
```

```
e.g. "java Client 127.0.0.1 Hello.java"
```

```
/* and server outputs */
"Reading Hello.java"
```

```
/* and client outputs */
```

```
class Hello
```

```
{
    public static void main(string[] args)
        system.out.println('Hi There');
}
```

```
/* and server outputs */
"waiting for connection"
```