Message Passing

- Two fundamental communication & synchronization paradigms

- Message passing
  - Asynchronous communication in distributed systems
  - Not always available

- Shared memory
  - Efficient, familiar
  - Potentially insecure

Syntax:

```c
send(process : process_id, message : string)
receive(process : process_id, var message : string)
```
Message Passing Example
Ye Olde Producer/Consumer System

process producer
begin loop
  produce a char 'c'
  send(consumer, c)
end loop
end producer

process consumer
begin loop
  receive(producer, msg)
  consume message 'msg'
end loop
end consumer

Issues
Naming communicants

How do processes refer to each other?
  » Does a sender explicitly name a receiver?

Can a message be sent to a group?

Implementation considerations
  » Synchronization among receivers
Issues
Synchronization semantics

◆ When does a send/receive operation terminate?

  » Blocking
     ◆ sender waits until its message is received
     ◆ receiver waits if no message is available

  » Non-blocking
     ◆ send operation “immediately” returns
     ◆ receive operation returns if no message is available

  » Variants
     ◆ send() / receive() with timeout

Semantics of Message Passing
send(recev, msg)

<table>
<thead>
<tr>
<th></th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronization</td>
<td>Send message to recev. Wait until message is accepted.</td>
<td>Send message to recev.</td>
</tr>
<tr>
<td>Explicit Naming</td>
<td>Broadcast message to all receivers. Wait until message is accepted by all.</td>
<td>Broadcast message to all receivers.</td>
</tr>
<tr>
<td>Implicit Naming</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Semantics of Message Passing

receive (sender, msg)

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Explicit</strong></td>
<td>Wait for a message from sender if there is a message from sender then receive it, else continue.</td>
<td>If there is a message from any sender then receive it, else continue.</td>
</tr>
<tr>
<td><strong>Implicit</strong></td>
<td>Wait for a message from any sender if there is a message from any sender then receive it, else continue.</td>
<td>If there is a message from any sender then receive it, else continue.</td>
</tr>
</tbody>
</table>

### Producer/Consumer Example

**Direct naming, blocking synchronization**

```plaintext
process producer
begin
  loop
    produce a char "c"
    send(consumer, c)
  end loop
end producer

process consumer
begin
  loop
    receive(consumer, msg)
    consume message "msg"
  end loop
end consumer
```

[Diagram of producer and consumer processes with OS kernel]
Producer/Consumer Example
Direct naming, non-blocking synchronization

process producer
begin
loop
  produce a char "c"
end loop
end producer

process consumer
begin
  receive(producer, mesg)
loop
  while (mesg \≠ NULL) do
    receive(producer, mesg)
  end while
  consume message "mesg"
end loop
end consumer

OS Kernel

Producer/Consumer Example
With non-blocking send, blocking receive

process producer
begin
loop
  produce a char "c"
end loop
end producer

process consumer
begin
  for i := 1 to n do
    send(producer, NULL)
  end for
loop
  receive(producer, mesg)
  consume message "mesg"
  send(producer, NULL)
end loop
end consumer

OS Kernel

Synchronization
Data-Flow
Producer/Consumer Example
With blocking send/receive

process producer
begin
loop
   <produce character 'c'>
   send(bufferManager, c)
end loop
end producer

process consumer
begin
loop
   send(bufferManager, request)
   receive(bufferManager, message)
   <consume message 'message'>
end loop
end consumer

process bufferManager
var buff : array [0..n-1] of char
nextIn, nextOut : 0..n-1 := 0
fullCount : 0..n := 0
begin
loop
   if (fullCount < n) then
      receive(producer, message)
      buff[nextIn] := message
      nextIn := nextIn + 1 mod n
      fullCount := fullCount + 1
   end if

   if (fullCount > 0) then
      receive(consumer, request)
      send(consumer, buff[nextOut])
      nextOut := nextOut + 1 mod n
      fullCount := fullCount - 1
   end if
end loop
end bufferManager

Realizing Parallel Execution
Buffered, asynchronous communication

process bufferManager
var buff : array [0..n-1] of char
nextIn, nextOut : 0..n-1 := 0
fullCount : 0..n := 0
begin
loop
   case select() of
      producer:
         deposit()
         if (fullCount = n) then
            remove()
         end if
      consumer:
         remove()
         if (fullCount = 0) then
            deposit()
         end if
   end case
end loop
end bufferManager

procedure deposit()
begin
receive(producer, message)
buff[nextIn] := message
nextIn := nextIn + 1 mod n
fullCount := fullCount + 1
end deposit

procedure remove()
begin
receive(consumer, request)
send(consumer, buff[nextOut])
nextOut := nextOut + 1 mod n
fullCount := fullCount - 1
end remove
Remote Procedure Call

Emulating shared memory via message passing

```
procedure remoteFunc(args)
begin
  return(results)
end remoteFunc
```

```
procedure Func(args)
begin
  <marshall parameters>
  send(serverStub, params)
  receive(serverStub, results)
  <unpack results>
  return(results)
end Func
```

```
proc process P1
begin
  loop
    call Func(args)
  end loop
end P1
```

```
proc process FuncServer
begin
  loop
    sender := select()
    receive(sender, params)
    <unpack parameters>
    call remoteFunc(args)
    <marshall results>
    send(sender, results)
  end loop
end FuncServer
```

Network

"Client" "Server"

Remote Procedure Call

Issues

- How does a client locate a server?
- What types of parameters can be passed?
- What parameter passing paradigms are easy/hard?
- How does one deal with errors & server failures?