CSCE 451/851
Introduction to Operating Systems

Interprocess Communications (Message Passing)

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE451

1

Message Passing

- Two fundamental communication & synchronization paradigms
  - » Shared memory
    - ❖ Efficient, familiar
    - ❖ Not always available
    - ❖ Potentially insecure
  - » Message passing
    - ❖ Awkward, less standardized
    - ❖ Extensible to communication in distributed systems
    - ❖ Syntax:

```
send(process : process_id, message : string)
receive(process : process_id, var message : string)
```

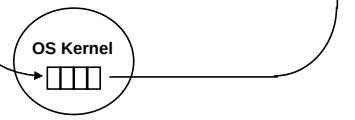
2

Message Passing Example

Ye Olde Producer/Consumer System

```
process producer                 process consumer
begin                            begin
  loop                             loop
    <produce a char "c">             receive(producer, mesg)

    send(consumer, c)                <consume message "mesg">
  end loop                        end loop
end producer                     end consumer
```

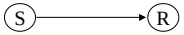


3

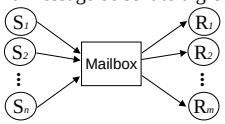
Issues

Naming communicants

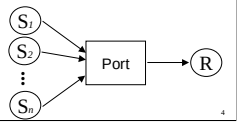
- How do processes refer to each other?
  - » Does a sender explicitly name a receiver?



  - » Can a message be sent to a group?



  - » Implementation considerations
    - ❖ Synchronization among receivers

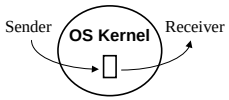


4

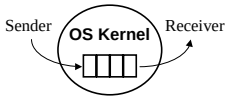
Issues

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*

5

Semantics of Message Passing

`send(recvr,mesg)`

Naming	Synchronization: Blocking	Synchronization: Nonblocking
Explicit	Send message to *recvr*. Wait until message is accepted.	Send message to *recvr*.
Implicit	Broadcast message to all receivers. Wait until message is accepted by all.	Broadcast message to all receivers.

6

Semantics of Message Passing

`receive(sender,mesg)`

Synchronization

Naming	Blocking	Nonblocking
Explicit	Wait for a message from *sender*	If there is a message from *sender* then receive it, else continue
Implicit	Wait for a message from any sender	If there is a message from any sender then receive it, else continue

7

Producer/Consumer Example

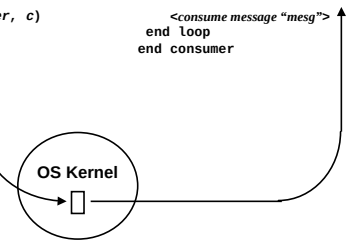
Direct naming, blocking synchronization

```
process producer
begin
  loop
    <produce a char "c">

    send(consumer, c)
  end loop
end producer
```

```
process consumer
begin
  loop
    receive(producer, mesg)

    <consume message "mesg">
 end loop
end consumer
```



8

Producer/Consumer Example

Direct naming, non-blocking synchronization

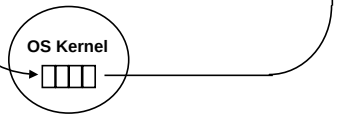
```
process producer
begin
  loop
    <produce a char "c">

    send(consumer, 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
```



9

Producer/Consumer Example

With non-blocking send, blocking receive

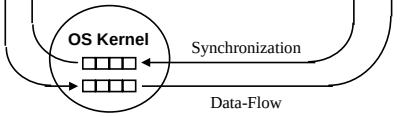
```
process producer
begin
 loop
   <produce a char "c">

   receive(consumer,mesg)
   send(consumer, 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
```



10

Producer/Consumer Example

With blocking send/receive

```
process producer
begin
 loop
   <produce a char "c">

   send(bufferManager, c)
 end loop
end producer
```

```
process consumer
begin
 loop
   send(bufferManager,request)
   receive(bufferManager,mesg)

   <consume message "mesg">
 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, mesg)
      buff[nextIn] := mesg
      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
```

11

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, mesg)
  buff[nextIn] := mesg
  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
```

12

Remote Procedure Call

Emulating shared memory via message passing

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

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


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

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

"Client" "Server"

13

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?

14