CSCE 351
Operating System Kernels

Interprocess Synchronization and Communication

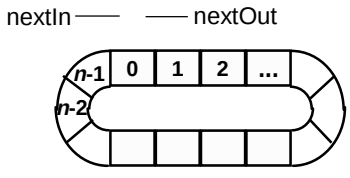
Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE351

Producer/Consumer Implementation

```
process Producer
  var c : char
begin
  loop
    <produce a character "c">
    while nextIn+1 mod n = nextOut do
      NOOP
    end while
    buf[nextIn] := c
    nextIn := nextIn+1 mod n
  end loop
end Producer
```

```
process Consumer
  var c : char
begin
  loop
    while nextIn = nextOut do
      NOOP
    end while
    c := buf[nextOut]
    nextOut := nextOut+1 mod n
   <consume a character "c">
  end loop
end Consumer
```

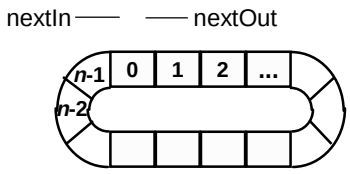



```
globals
   buf : array [0..n-1] of char;
   nextIn, nextOut : 0..n-1 := 0
```

Producer/Consumer Implementation with a shared counter

```
process Producer
  var c : char
begin
  loop
    <produce a character "c">
    while count = n do
      NOOP
    end while
    buf[nextIn] := c
    nextIn := nextIn+1 mod n
    count  := count + 1
  end loop
end Producer
```

```
process Consumer
  var c : char
begin
  loop
    while count = 0 do
      NOOP
    end while
    c := buf[nextOut]
    nextOut := nextOut+1 mod n
    count   := count - 1
   <consume a character "c">
  end loop
end Consumer
```




```
globals
   buf   : array [0..n-1] of char;
   nextIn, nextOut : 0..n-1 := 0
   count : integer := 0
```


The Critical Section Problem

- One implementation of the shared counter

```
process Producer
begin
   :
 <count := count + 1>
  MOV R1, @count
  ADD R1, 1
  MOV @count, R1
   :
end Producer
```

```
process Consumer
begin
   :
 <count := count - 1>
  MOV R2, @count
  SUB R2, 1
  MOV @count, R2
   :
end Consumer
```

Algorithms for Mutual Exclusion

- General algorithm structure

```
process Pi
begin
  loop
     :
    Entry_Protocol
    <critical section>
    Exit_Protocol
     :
  end loop
end Pi
```

- Correctness conditions
  - » Does it guarantee mutual exclusion?
  - » Is it expedient?
  - » Does it provide bounded waiting?

Mutual Exclusion

- Disable Interrupts

```
process P1
begin
  loop
    Disable Interrupts
    <critical section>
    Enable Interrupts
  end loop
end P1
```

```
process P2
begin
  loop
    Disable Interrupts
    <critical section>
    Enable Interrupts
  end loop
end P2
```

Message Passing

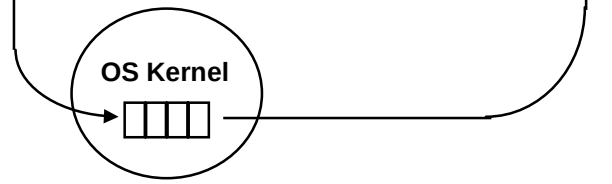
- Two fundamental communication & synchronization paradigms
  - » Shared memory
    - ❖ Efficient, familiar
    - ❖ Not always available
    - ❖ Potentially insecure
  - » Message passing
    - ❖ Awkward, less standardized
    - ❖ Extensible to communication in distributed systems
    - ❖ Syntax:

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


Message Passing Example
Ye Olde Producer/Consumer System

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

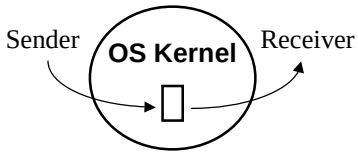
```
process consumer
begin
  loop
    receive(producer, mesg)
    <consume message "mesg">
  end loop
end consumer
```



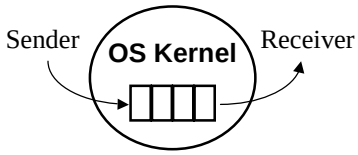
Issues

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(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.

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