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**

```plaintext
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**

```plaintext
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**

```plaintext
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**

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

**process Consumer**

```plaintext
begin
   :
   <count := count - 1>
   MOV R2, @count
   SUB R2, 1
   MOV @count, R2
   :
end Consumer
```
Algorithms for Mutual Exclusion

◆ General algorithm structure

\[\text{process } P_i \begin{align*} &\text{begin} \\
&\text{loop} \\
&\quad \text{Entry	extunderscore Protocol} \\
&\quad <\text{critical section}> \\
&\quad \text{Exit	extunderscore Protocol} \\
&\quad : \\
&\quad \text{end loop} \\
&\text{end } P_i \end{align*}\]

◆ Correctness conditions

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

Mutual Exclusion

◆ Disable Interrupts

\[\text{process } P_1 \begin{align*} &\text{begin} \\
&\text{loop} \\
&\quad \text{Disable Interrupts} \\
&\quad <\text{critical section}> \\
&\quad \text{Enable Interrupts} \\
&\quad \text{end loop} \\
&\text{end } P_1 \end{align*}\]

\[\text{process } P_2 \begin{align*} &\text{begin} \\
&\text{loop} \\
&\quad \text{Disable Interrupts} \\
&\quad <\text{critical section}> \\
&\quad \text{Enable Interrupts} \\
&\quad \text{end loop} \\
&\text{end } P_2 \end{align*}\]
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(consumer, msg)
    consume message “msg”
  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(recurv, mesg)`**

### Synchronization

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

**receive(sender, mesg)**

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