Chapter 2 (Third part)

Monitors, Reentrant Code, Message Passing

Introduction

It is difficult to produce correct programs using locks and semaphores!!!

Correct ordering of Up and Down operations is tricky!

Desirable:

Language / compiler support for IPC

What are suitable high-level abstractions for synchronization?

Monitors

Collect related, shared objects together in a *"monitor"*

Characteristics:

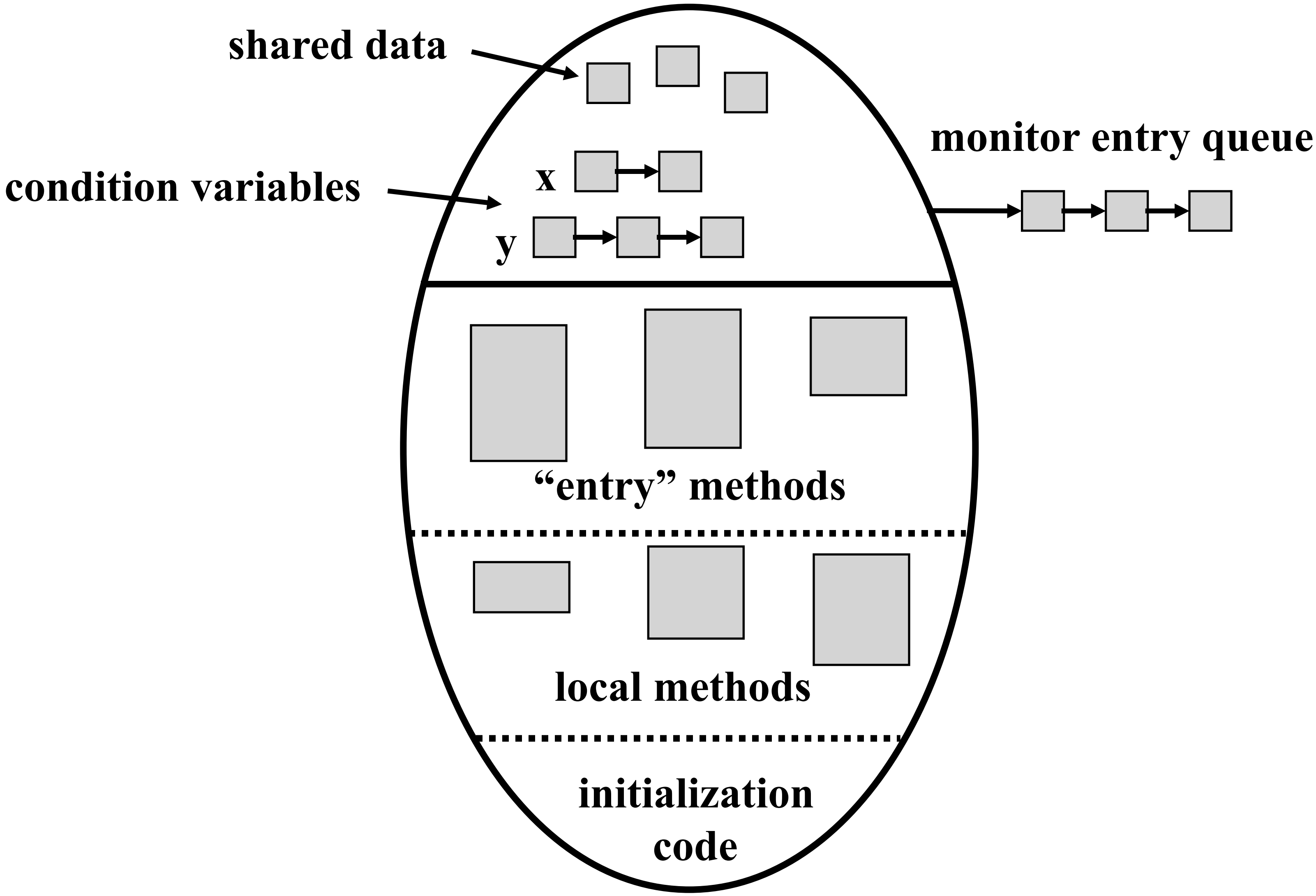
- **Local data variables are accessible only via the monitor's procedures/methods**
- **Threads enter the monitor by invoking one of its procedures/methods**
- **Only one thread may execute within the monitor at a given time**

***"Condition Variables"* (cv)**

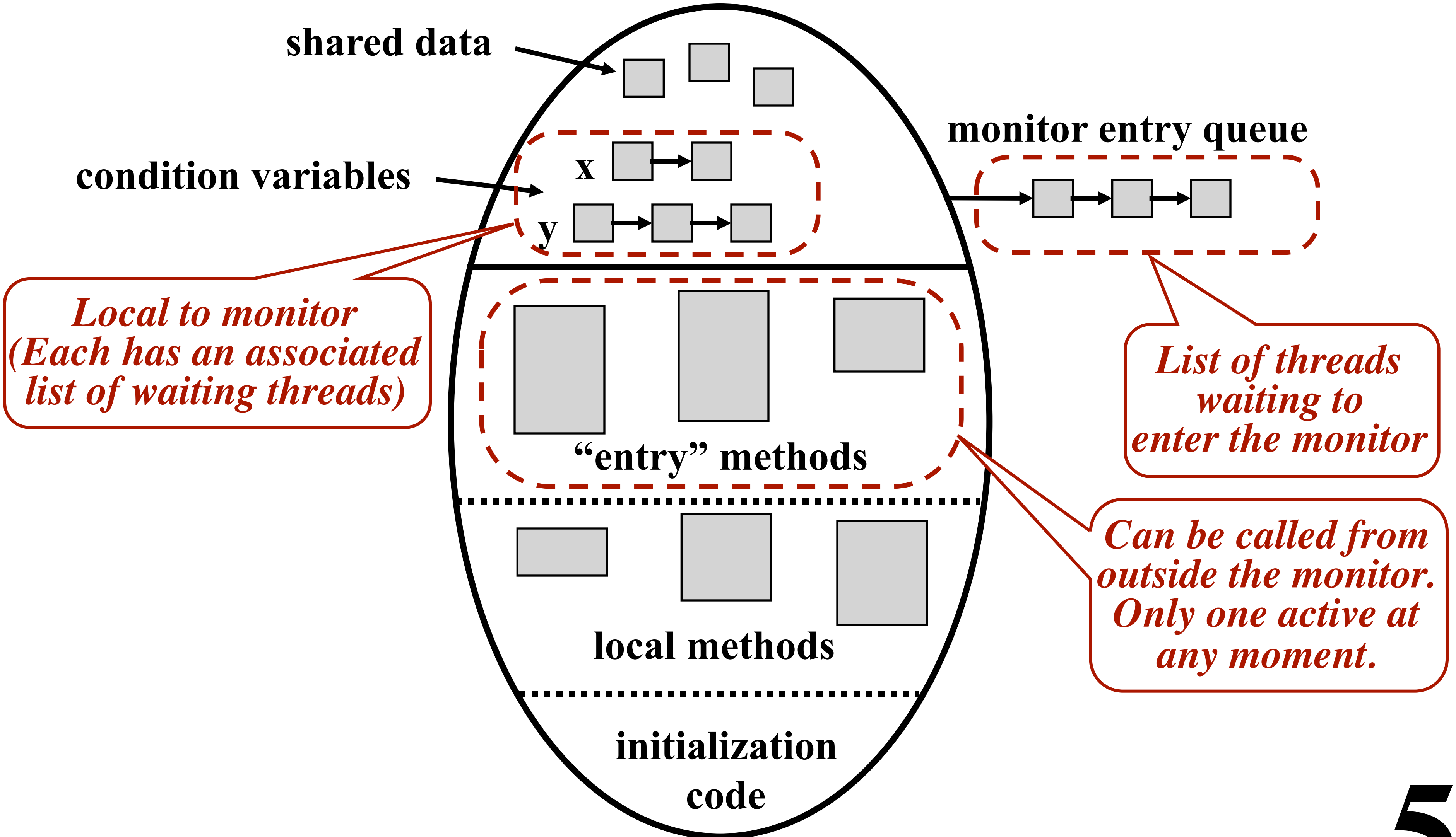
Wait(cv) – block on condition

Signal(cv) – wake up one thread waiting on cv

Monitor structures



Monitor structures



Example: The "Bounded-Buffer" Monitor

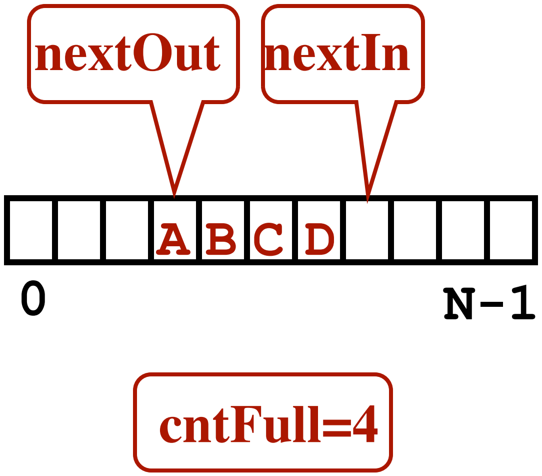
```
Producer Thread:
  while true
    -- Produce char "c"
    BoundedBuffer.deposit(c)
  endWhile
```

```
Consumer Thread:
  while true
    c = BoundedBuffer.remove()
    -- Consume char "c"
  endWhile
```

```
monitor BoundedBuffer
  var buffer: ...
     nextIn, nextOut :...

  entry deposit(c: char)
     begin
       ...
     end

  entry remove()
     begin
       ...
       return c
     end

endMonitor
```

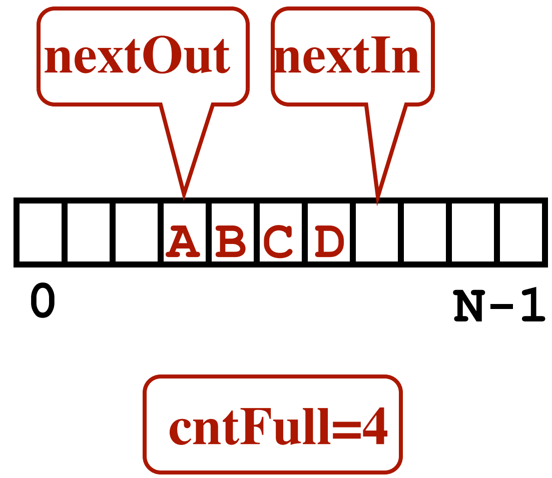
The “BoundedBuffer” Monitor

```
monitor BoundedBuffer
  var buffer: array[n] of char
      nextIn, nextOut: int = 0
      cntFull: int = 0
      notEmpty: Condition
      notFull: Condition

  entry deposit(c: char)
      ...




  entry remove()
      ...





endMonitor
```



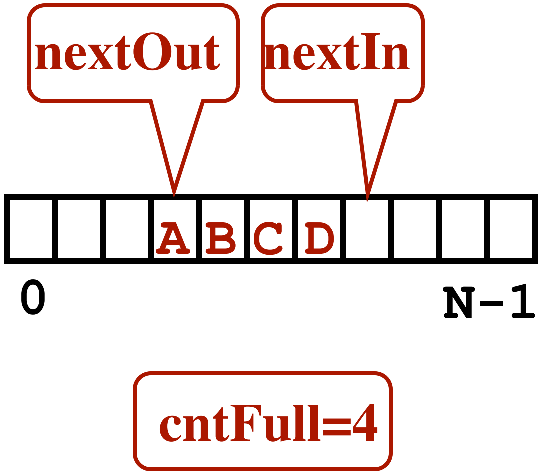
Code for the “deposit” entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     if cntFull == N
       notFull.Wait()
     endIf
     buffer[nextIn] = c
     nextIn = (nextIn+1) mod N
     cntFull = cntFull + 1
     notEmpty.Signal()
   endEntry

  entry remove()
     ...

endMonitor
```



Code for the "remove" entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     ...

  entry remove()
     if cntFull == 0
       notEmpty.Wait()
     endIf
     c = buffer[nextOut]
     nextOut = (nextOut+1) mod N
     cntFull = cntFull - 1
     notFull.Signal()
   endEntry

endMonitor
```



Condition Variables

"Condition variables allow processes to synchronize based on some state of the monitor variables."

Examples from producer/consumer:

"Buffer-Not-Full" condition

"Buffer-Not-Empty" condition

Operations Wait(cv) and Signal(cv) allow synchronization within the monitor

When a producer thread adds an element...

A consumer may be sleeping

Need to wake the consumer... Signal

Condition synchronization semantics

"Only one thread can be executing in the monitor at any one time."

Scenario:

Thread A is executing in the monitor.

Thread A does a Signal, waking up thread B.

What happens now?

Signaling and signaled threads can not both run!

Condition synchronization semantics

Option 1: Hoare Semantics

What happens when a Signal is performed?

The signaling thread (A) is suspended.

The signaled thread (B) wakes up and runs immediately.

B can assume the condition is now true/satisfied

- Stronger guarantees
- Easier to prove correctness

When B leaves monitor, then A can run.

After B leaves monitor...

A might resume execution immediately

... or maybe another thread (C) will slip in!

Condition synchronization semantics

Option 2: MESA Semantics (Xerox PARC)

What happens when a Signal is performed?

- **The signaling thread (A) continues.**
- **The signaled thread (B) waits.**
  When A leaves monitor, then B runs.

Issue: What happens when B waits?
When A leaves the monitor,
can some other thread (C) slip in first?
(Can some other thread (C) run
after A signals, but before B runs?)

- **A signal is more like a hint.**
- **Requires B to recheck the state of the monitor variables to see if it can proceed or must wait some more.**

Code for the “deposit” entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     if cntFull == N
       notFull.Wait()
     endIf
     buffer[nextIn] = c
     nextIn = (nextIn+1) mod N
     cntFull = cntFull + 1
     notEmpty.Signal()
   endEntry

  entry remove()
     ...

endMonitor
```

Hoare Semantics

Code for the "deposit" entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     while cntFull == N
       notFull.Wait()
     endWhile
     buffer[nextIn] = c
     nextIn = (nextIn+1) mod N
     cntFull = cntFull + 1
     notEmpty.Signal()
   endEntry

  entry remove()
     ...

endMonitor
```

MESA Semantics

Code for the “remove” entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     ...

  entry remove()
     if cntFull == 0
       notEmpty.Wait()
     endIf
     c = buffer[nextOut]
     nextOut = (nextOut+1) mod N
     cntFull = cntFull - 1
     notFull.Signal()
   endEntry

endMonitor
```

Hoare Semantics

Code for the "remove" entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     ...

  entry remove()
     while cntFull == 0
       notEmpty.Wait()
     endWhile
     c = buffer[nextOut]
     nextOut = (nextOut+1) mod N
     cntFull = cntFull - 1
     notFull.Signal()
   endEntry

endMonitor
```

MESA Semantics (while … endWhile)

“Hoare Semantics”

What happens when a Signal is performed?

The signaling thread (A) is suspended.

The signaled thread (B) wakes up and runs immediately.

B can assume the condition is now true/satisfied

From the original Hoare Paper:

“No other thread can intervene [and enter the monitor] between the signal and the continuation of exactly one waiting thread.”

“If more than one thread is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting thread. This gives a simple neutral queuing discipline which ensures that every waiting thread will eventually get its turn.”

Implementing Hoare Semantics

Implementation?

Thread A holds the monitor lock.

Thread A issues a Signal.

Thread B will be moved back to the ready queue.

Thread A must be suspended...

Possession of the monitor lock must be passed from A to B.

When B finishes and gets ready to return...

The lock can be released.

Thread A must re-aquire the lock.

Perhaps A is blocked, waiting to re-aquire the lock.

Implementing Hoare Semantics

Problem:

"Possession of the monitor lock must be passed from A to B."

Each mutex remembers which thread holds it.

My version of Mutex:

Any attempt by thread B to release the monitor lock will cause an error message.

***Your Solution*:**

Modify Mutex to eliminate the check?

Implementing Hoare Semantics

Recommendation:

Do not modify the methods that I am supplying.

(Future code I release will use them)

Create new classes:

MonitorLock -- similar to Mutex

HoareCondition -- similar to Condition

Implementing Hoare Semantics

Scenario:

Thread B does a Wait.

Thread A executes a Signal.

Thread B wakes up, executes, and returns.

Last thing B does: Unlock the monitor lock.

Problem: What happens next?

Thread A is waiting for B to finish.

It is trying to reaquire the monitor lock.

What about thread C?

Also trying to acquire lock, and waiting longer?

Hoare: "A must get the lock after B."

C must continue to wait.

Implementing Hoare Semantics

Things are getting complex.
Simply ending monitor entry methods with

```
monLock.Unlock()
```

will no longer work.

<u>Implementation Ideas:</u>
Need a special thing called a "*MonitorLock*".
Consider a thread like A to be "urgent".
Thread C is not "urgent".
Consider 2 wait lists associated with each *MonitorLock*

- **UrgentlyWaitingThreads**
- **NonurgentlyWaitingThreads**

Want to wake up urgent threads first, if any.

Brinch-Hansen Semantics

Hoare Semantics

On signal, allow signaled process to run.

Upon its exit from the monitor, signaler process continues.

Brinch-Hansen Semantics

Signaler must immediately exit following any invocation of signal.

(Implementation is easier.)

Reentrant Code

A function/method is said to be "reentrant" if...

"A function that has been invoked may be invoked again before the first invocation has returned, and will still work correctly."

Recursive routines are reentrant.

In the context of multi-programming...

A reentrant function can be executed simultaneously by more than one thread, with no ill effects.

Reentrant Code

Consider this function...

```
var count: int = 0

function GetUnique () returns int
  count = count + 1
  return count
endFunction
```

What if it is executed by different threads?

Reentrant Code

Consider this function...

```
var count: int = 0

function GetUnique () returns int
  count = count + 1
  return count
endFunction
```

What if it is executed by different threads?

The results may be incorrect!

This routine is not reentrant!

Reentrant Code

When is code "reentrant"?

Assumptions:

A multi-threaded program

Some variables are

"local" -- to the function/method/routine

"global" -- sometimes called "static"

Access to local variables?

A new stack frame is created for each invocation.

Access to global variables?

Must use synchronization!

Making this Function Reentrant

```
var count: int = 0
    myLock: Mutex

function GetUnique () returns int
  var i: int
  myLock.Lock()
  count = count + 1
  i = count
  myLock.Unlock()
  return i
endFunction
```

Message Passing

Interprocess Communication

- **via shared memory**
- **across machine boundaries**

Message passing can be used locally or remotely.

Can be used for...

synchronization, or

general communication

Processes use Send and Receive primitives

- **Receive can block (like Waiting on a Semaphore)**
- **Send unblocks a process blocked on Receive**

  (Just as a Signal unblocks a Waiting process)

Design Choices for Message Passing

Option 1: ***"Mailboxes"***

System maintains a buffer of sent, but not yet received, messages.

Must specify the size of the mailbox ahead of time.

Sender will be blocked if buffer is full.

Receiver will be blocked if the buffer is empty.

Design Choices for Message Passing

Option 1: ***"Mailboxes"***

System maintains a buffer of sent, but not yet received, messages.

Must specify the size of the mailbox ahead of time.

Sender will be blocked if buffer is full.

Receiver will be blocked if the buffer is empty.

Option 2: ***The kernel does no buffering***

If Send happens first, the sending thread blocks.

If Receiver happens first, the receiving thread blocks.

"Rendezvous"

Both threads are ready for the transfer.

The data is copied / transmitted

Both threads are then allowed to proceed.

Producer-Consumer with Message Passing

Idea:

After producing, the producer sends the
data to consumer in a message.
The system buffers messages.
The producer can out-run the consumer.
The messages will be kept in order.
After consuming the data,
the consumer sends back an "empty" message.
A fixed number of messages (N=100)
The messages circulate back and forth.

Producer-Consumer with Message Passing

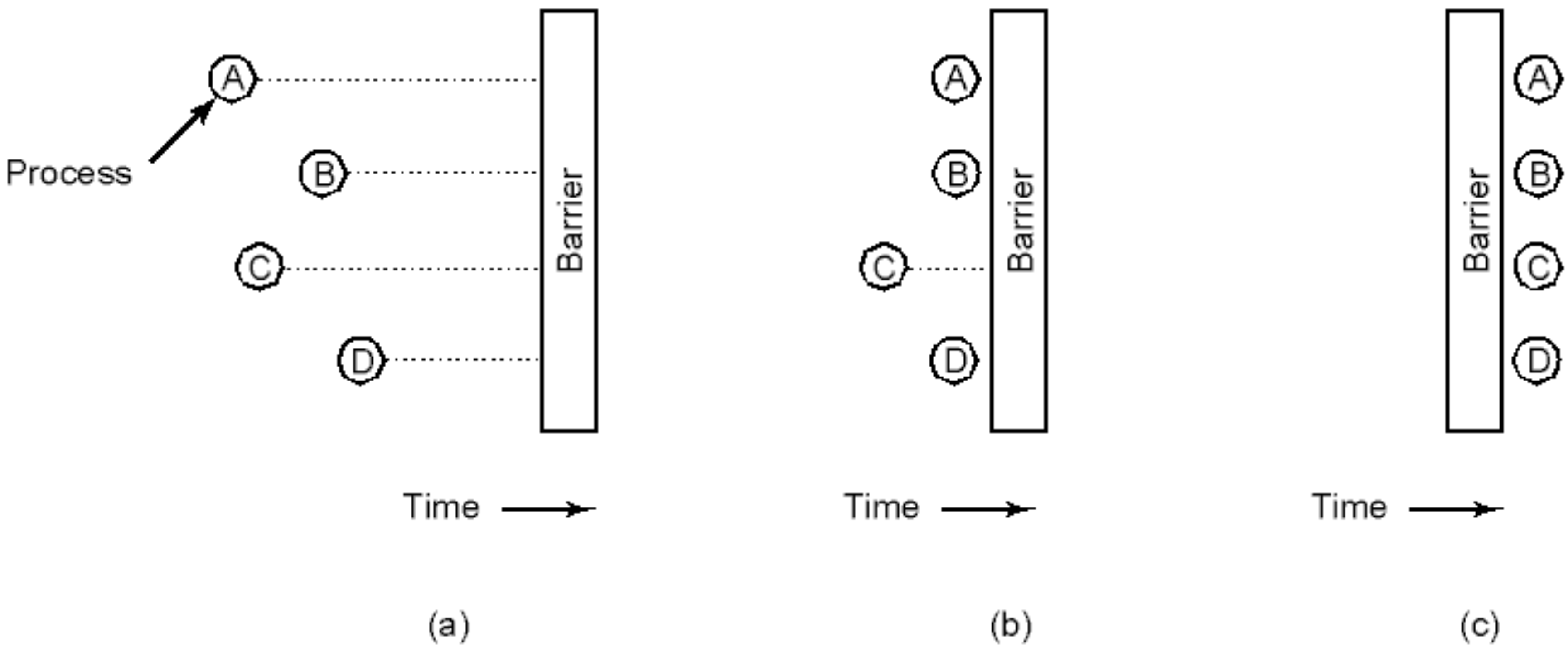
```
const N = 100              -- Size of message buffer
var em: char
for i = 1 to N             -- Get things started by
  Send (producer, &em)     --    sending N empty messages
endFor
```

```
thread consumer
  var c, em: char
  while true
    Receive(producer, &c)    -- Wait for a char
    Send(producer, &em)      -- Send empty message back
    // Consume char...
  endWhile
end
```

Producer-Consumer with Message Passing

```
thread producer
  var c, em: char
  while true
    // Produce char c...
    Receive(consumer, &em)    -- Wait for an empty msg
    Send(consumer, &c)        -- Send c to consumer
  endWhile
end
```

Barriers



- Processes approaching a barrier
- All processes but one blocked at barrier
- Last process arrives; all are let through