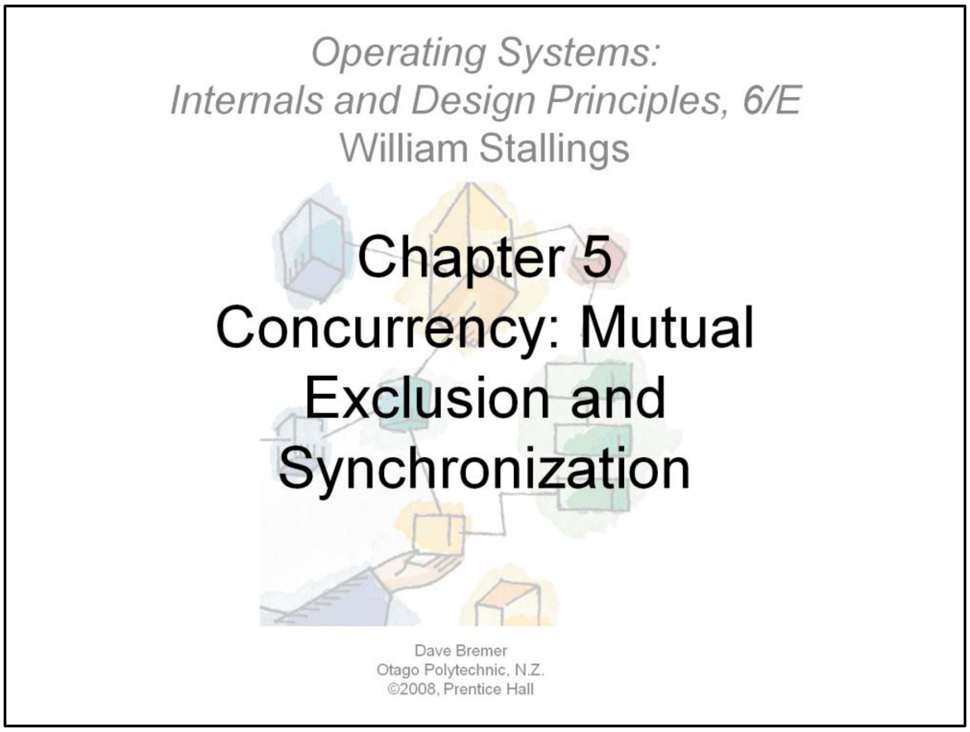
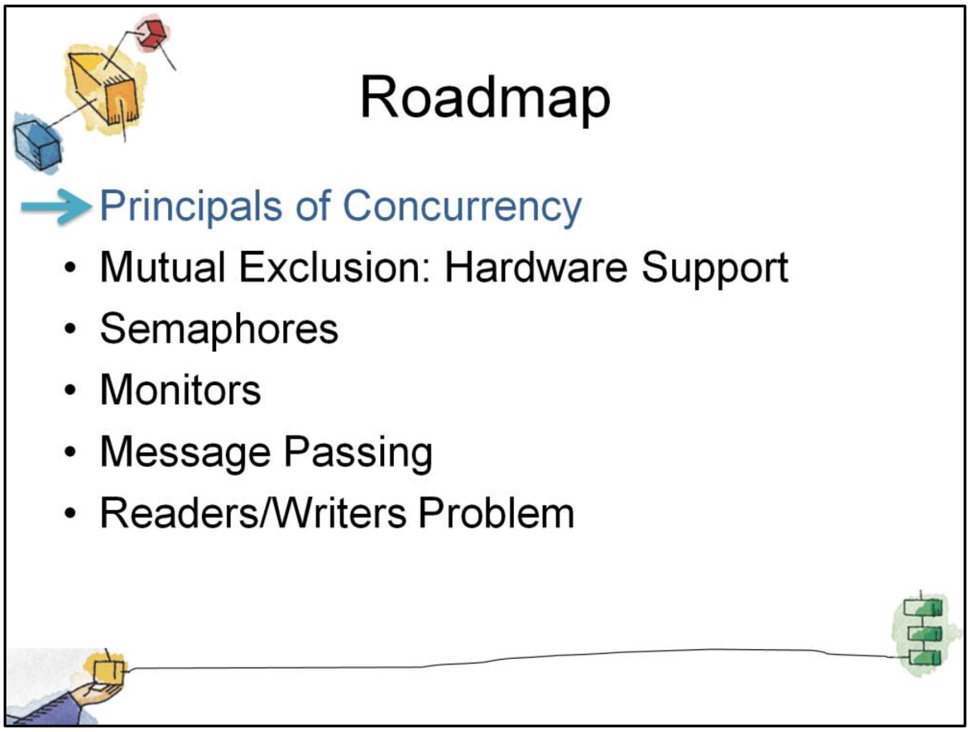


Operating Systems:
Internals and Design Principles, 6/E
William Stallings

# Chapter 5 Concurrency: Mutual Exclusion and Synchronization

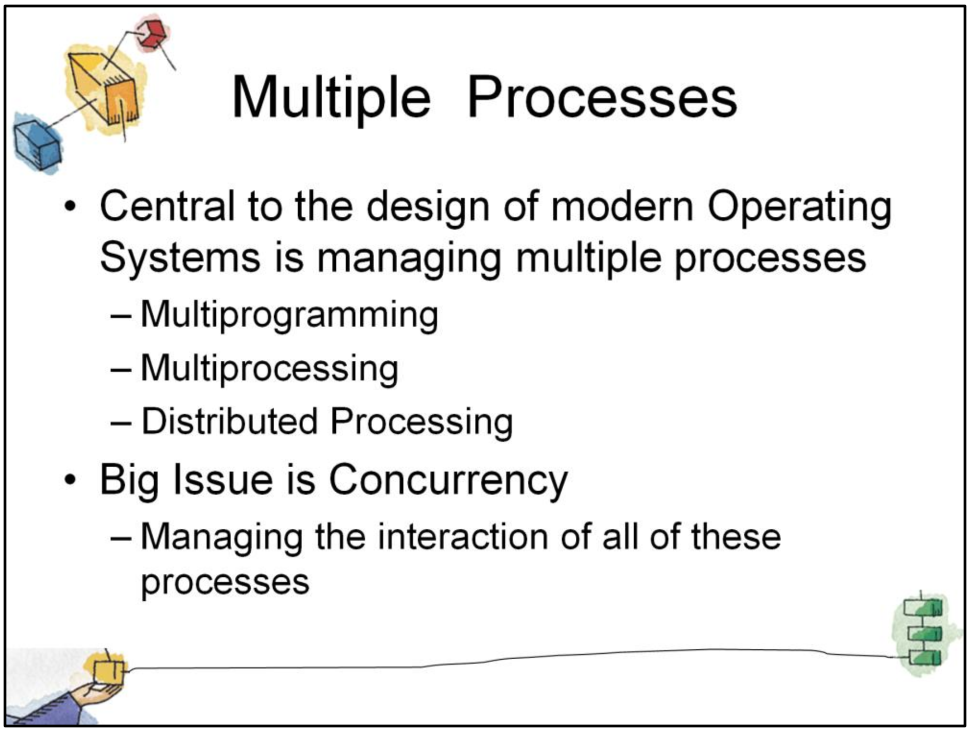
Dave Bremer
Otago Polytechnic, N.Z.

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.



## Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem



## Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

The central themes of operating system design are all concerned with the management of processes and threads:

- Multiprogramming:

  The management of multiple processes within a uniprocessor system.

- Multiprocessing:

  The management of multiple processes within a multiprocessor.

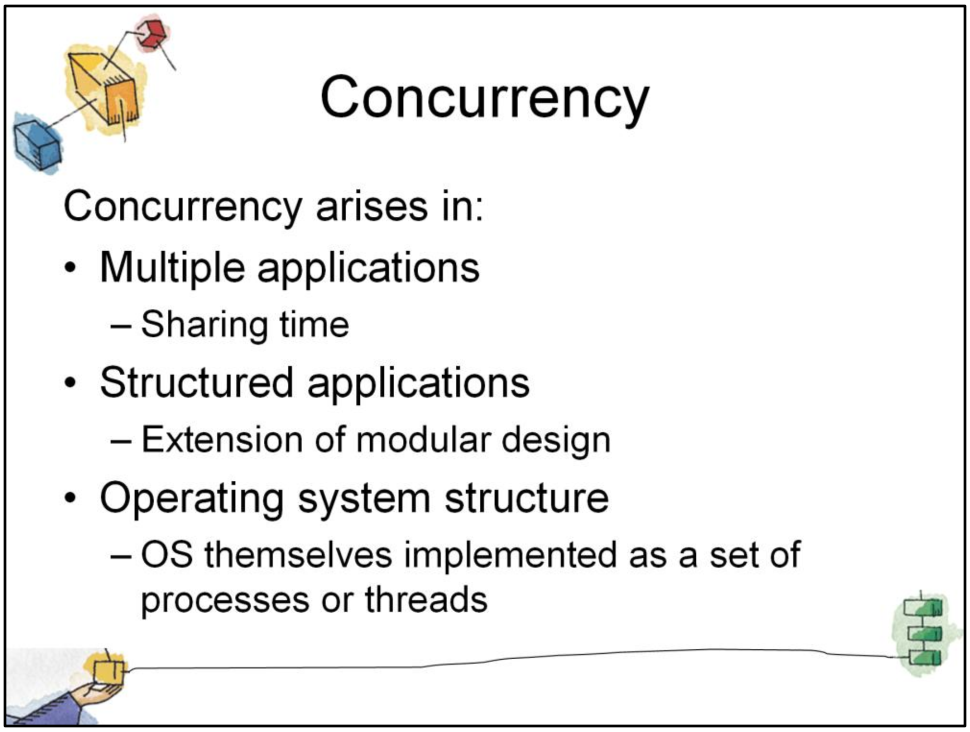
- Distributed processing:

  The management of multiple processes executing on multiple, distributed computer systems.

  E. G clusters

Concurrency encompasses a host of design issues, including

- communication among processes,
- sharing of and competing for resources (such as memory, files, and I/O access),
- synchronization of the activities of multiple processes, and
- allocation of processor time to processes.



## Concurrency

Concurrency arises in:

- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

• Multiple applications:

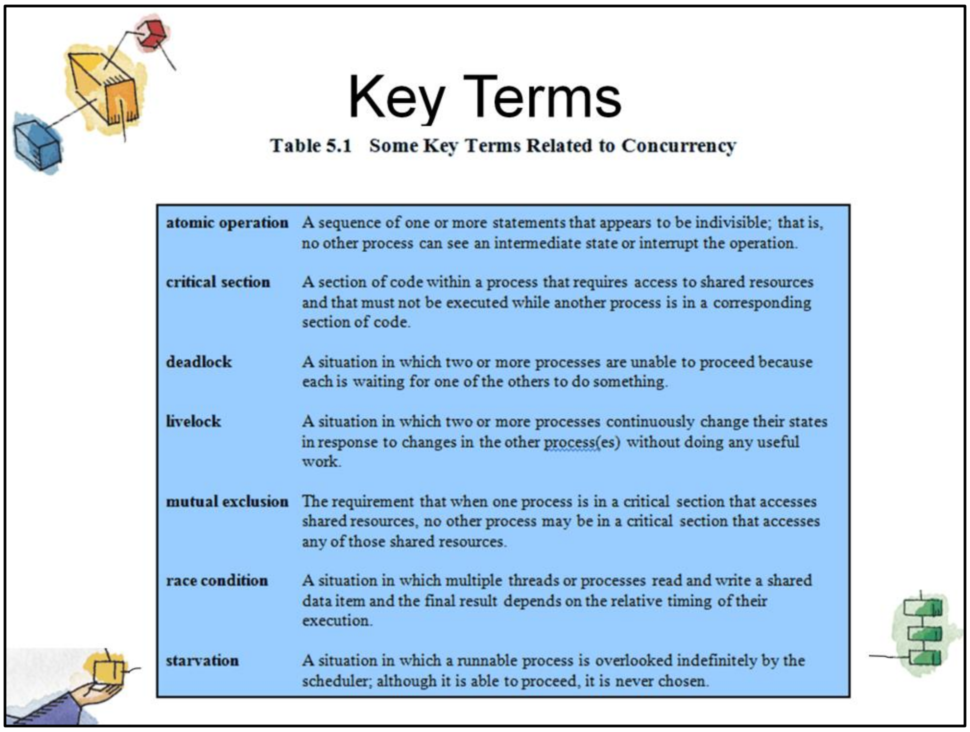
Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.

• Structured applications:

As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.

• Operating system structure:

The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.



## Key Terms

Table 5.1 Some Key Terms Related to Concurrency

| | |
|---|---|
| **atomic operation** | A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. |
| **critical section** | A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code. |
| **deadlock** | A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something. |
| **livelock** | A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work. |
| **mutual exclusion** | The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources. |
| **race condition** | A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution. |
| **starvation** | A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen. |

These terms will be explained throughout this chapter – do not labor this table.

It may be best skipped over and given as a reference point for students.

## Interleaving and Overlapping Processes

- Earlier (Ch2) we saw that processes may be interleaved on uniprocessors

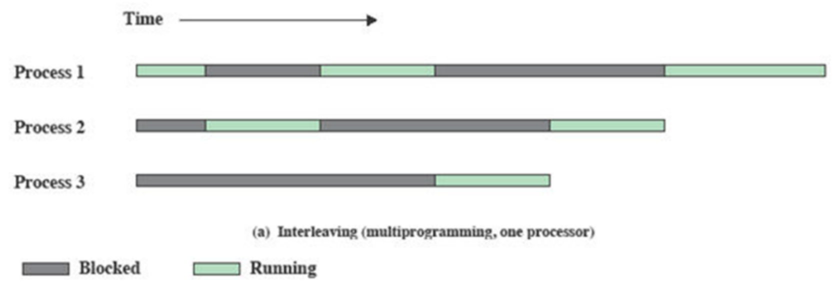
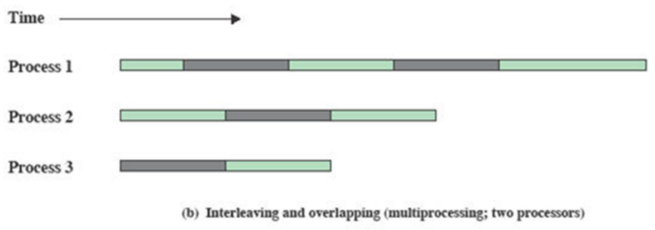


Figure 2.12 Multiprogramming and Multiprocessing

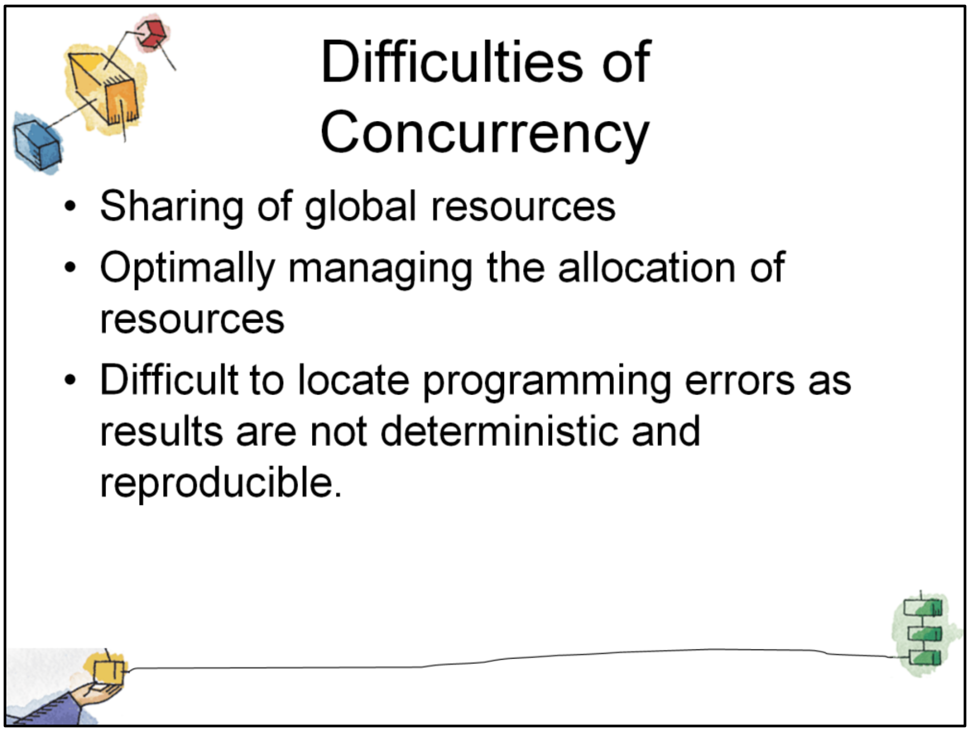
## Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors


Time
Process 1
Process 2
Process 3
(b) Interleaving and overlapping (multiprocessing; two processors)

Blocked
Running

Figure 2.12 Multiprogramming and Multiprocessing



## Difficulties of Concurrency

- Sharing of global resources
- Optimally managing the allocation of resources
- Difficult to locate programming errors as results are not deterministic and reproducible.

**The sharing of global resources**

If two processes both make use of the same global variable

and **both perform reads and writes** on that variable,

then **the order** in which the various reads and writes are executed is critical.

**Managing Resources**

•It is difficult for the OS to manage the allocation of resources optimally.

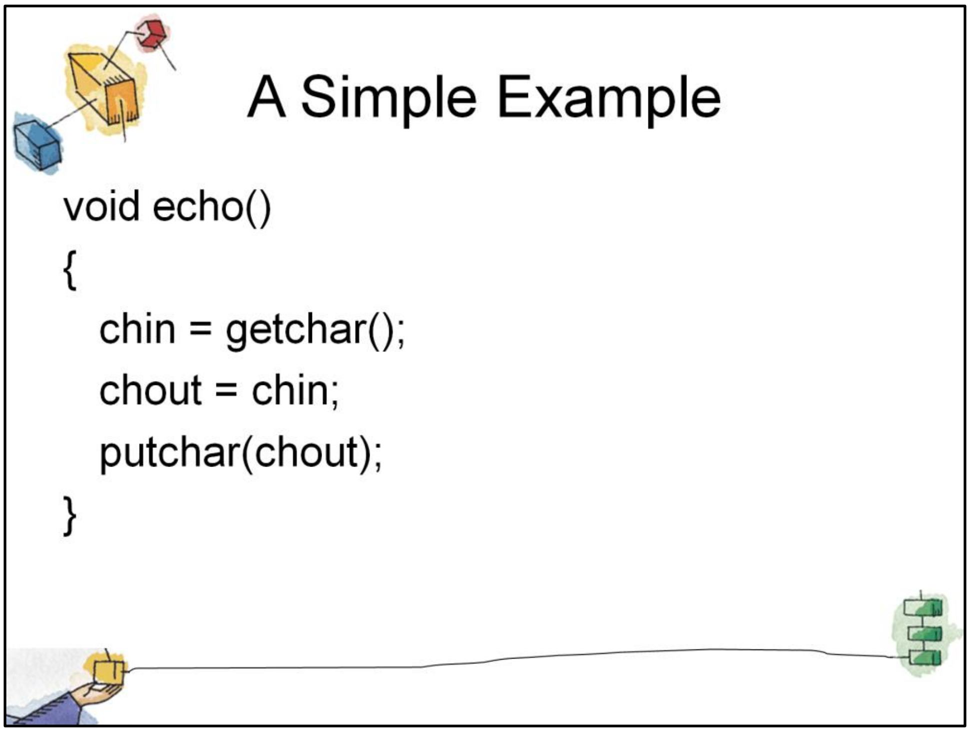
•E.G. A process may request use of, and be granted control of, a particular I/O channel and then be suspended before using that channel.

-It may be undesirable for the OS simply to lock the channel and prevent its use by other processes;

- indeed this may lead to a deadlock condition,

**Locating Programming Errors**

It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible



## A Simple Example

```
void echo()
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

A program that will provide a character echo procedure;

- input is obtained from a keyboard one keystroke at a time.
- Each input character is stored in variable chin.
- It is then transferred to variable chout
- and finally sent to the display.

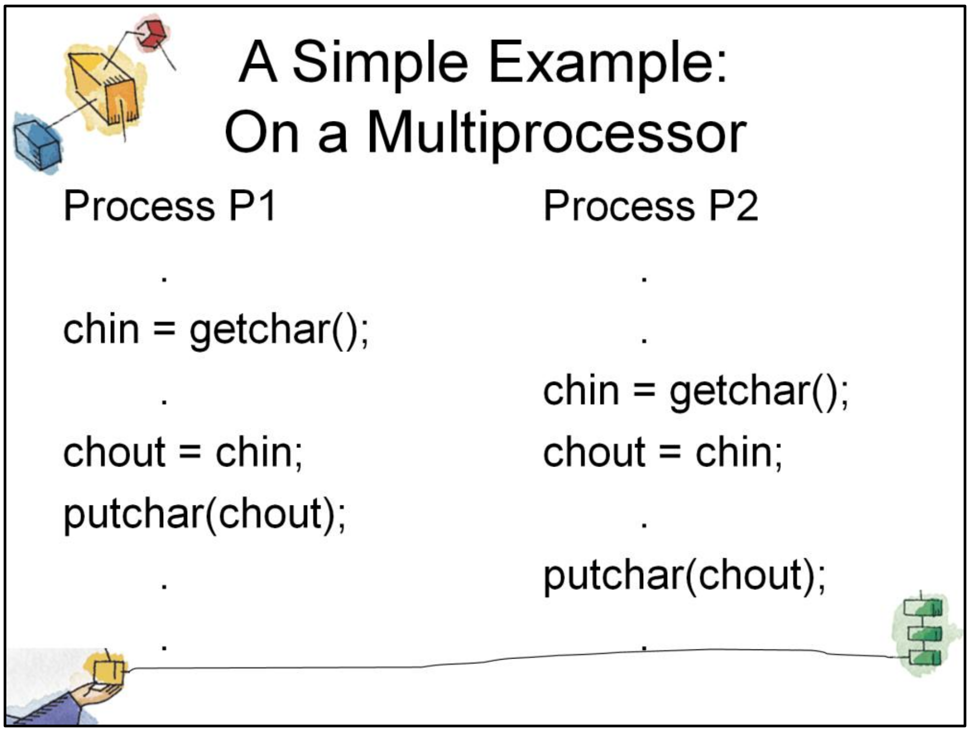
Any program can call this procedure repeatedly to accept user input and display it on the user’s screen.

Now consider that we have a single-processor multiprogramming system supporting a single user.

- The user can jump from one application to another, and each application uses the same keyboard for input and the same screen for output.

Each application needs to use the procedure echo,

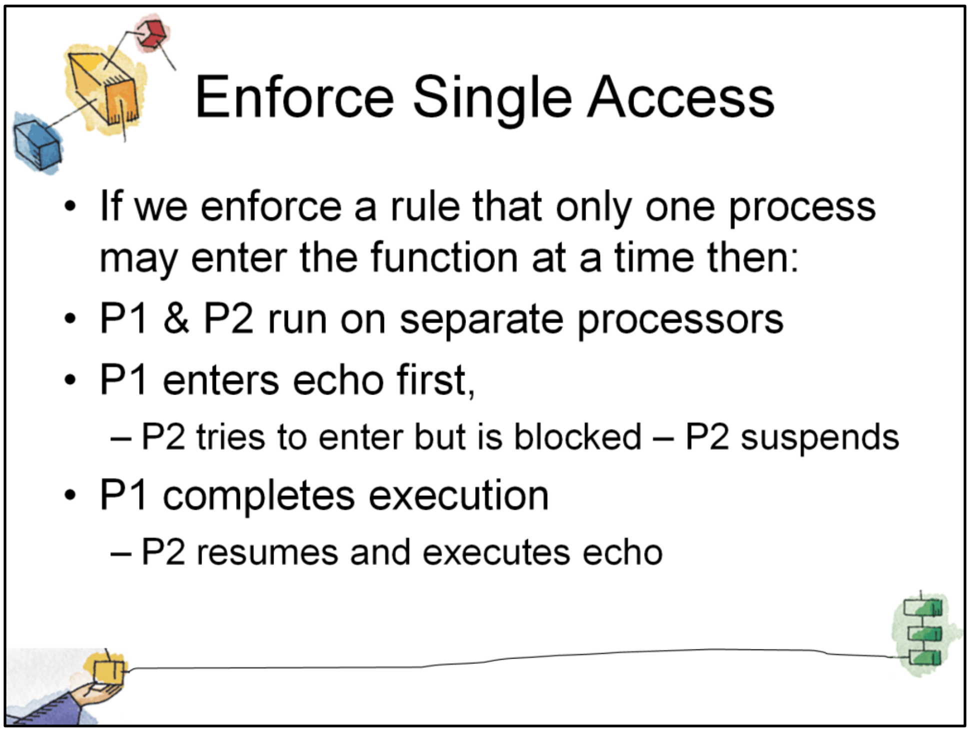
- So it makes sense for it to be a shared procedure that is loaded into a portion of memory global to all applications.
- Thus, only a single copy of the echo procedure is used, saving space.



## A Simple Example: On a Multiprocessor

| Process P1 | Process P2 |
|---|---|
| . | . |
| chin = getchar(); | . |
| . | chin = getchar(); |
| chout = chin; | chout = chin; |
| putchar(chout); | . |
| . | putchar(chout); |
| . | . |

The result is that the character input to P1 is lost before being displayed, and the character input to P2 is displayed by both P1 and P2.



## Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then:
- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

1. Processes P1 and P2 are both executing, each on a separate processor.
   - P1 invokes the echo procedure.

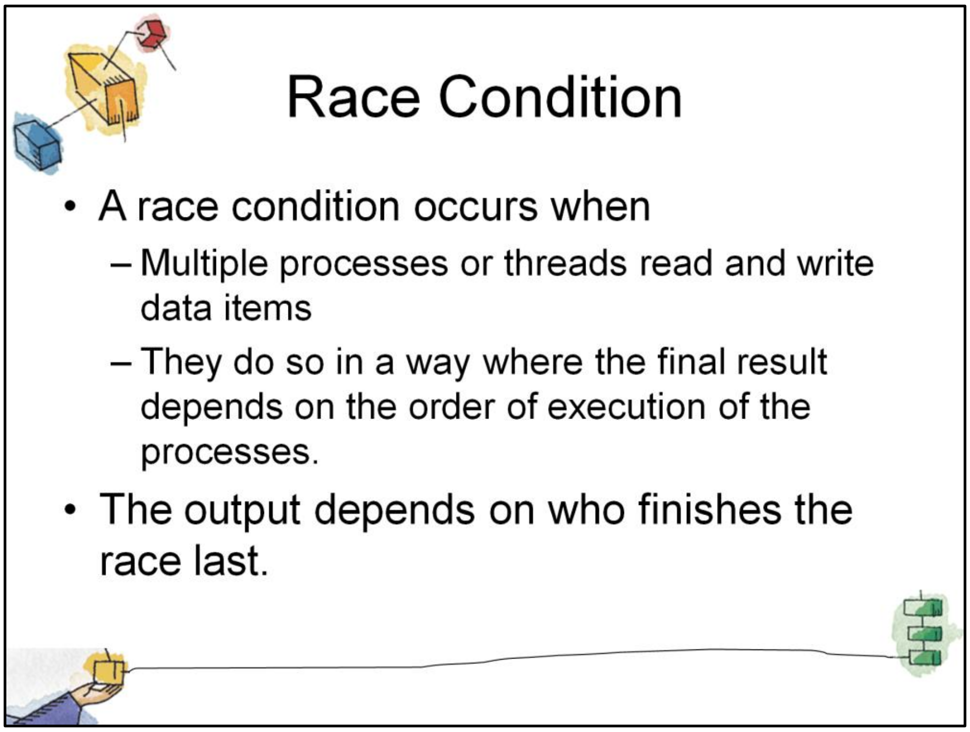
2. While P1 is inside the echo procedure, P2 invokes echo.

   • Because P1 is still inside the echo procedure (whether P1 is suspended or executing), P2 is blocked from entering the procedure.

   • Therefore, P2 is suspended awaiting the availability of the echo procedure.

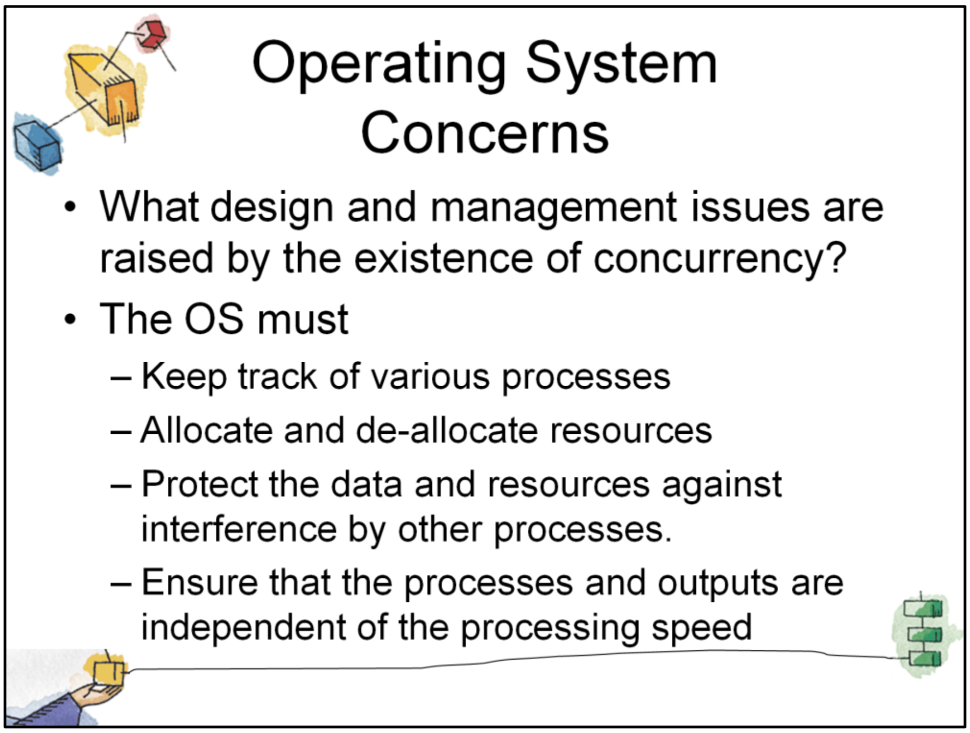
3. At a later time, process P1 completes execution of echo, exits that procedure, and continues executing.

   • Immediately upon the exit of P1 from echo, P2 is resumed and begins executing echo.



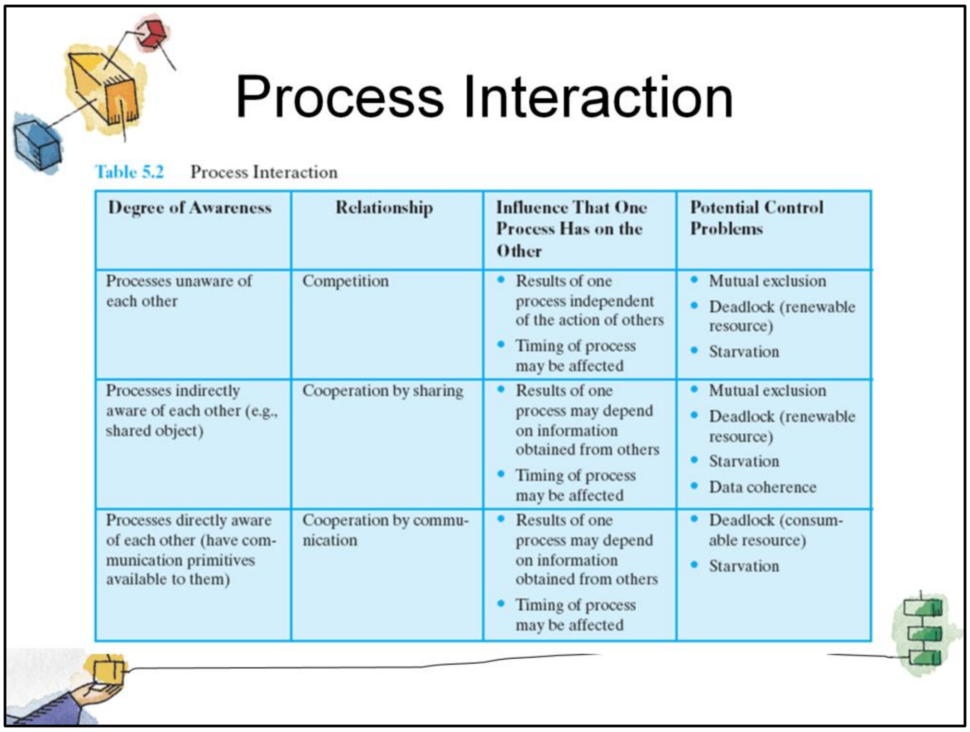
## Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.



## Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed



## Process Interaction

Table 5.2 Process Interaction

| Degree of Awareness | Relationship | Influence That One Process Has on the Other | Potential Control Problems |
|---|---|---|---|
| Processes unaware of each other | Competition | • Results of one process independent of the action of others<br>• Timing of process may be affected | • Mutual exclusion<br>• Deadlock (renewable resource)<br>• Starvation |
| Processes indirectly aware of each other (e.g., shared object) | Cooperation by sharing | • Results of one process may depend on information obtained from others<br>• Timing of process may be affected | • Mutual exclusion<br>• Deadlock (renewable resource)<br>• Starvation<br>• Data coherence |
| Processes directly aware of each other (have communication primitives available to them) | Cooperation by communication | • Results of one process may depend on information obtained from others<br>• Timing of process may be affected | • Deadlock (consumable resource)<br>• Starvation |

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence.

This table (continues on the next slide) lists three possible degrees of awareness plus the consequences of each.

Introduce each degree of awareness briefly – mention that things are often as clear cut as in this table.

**Processes unaware of each other:**

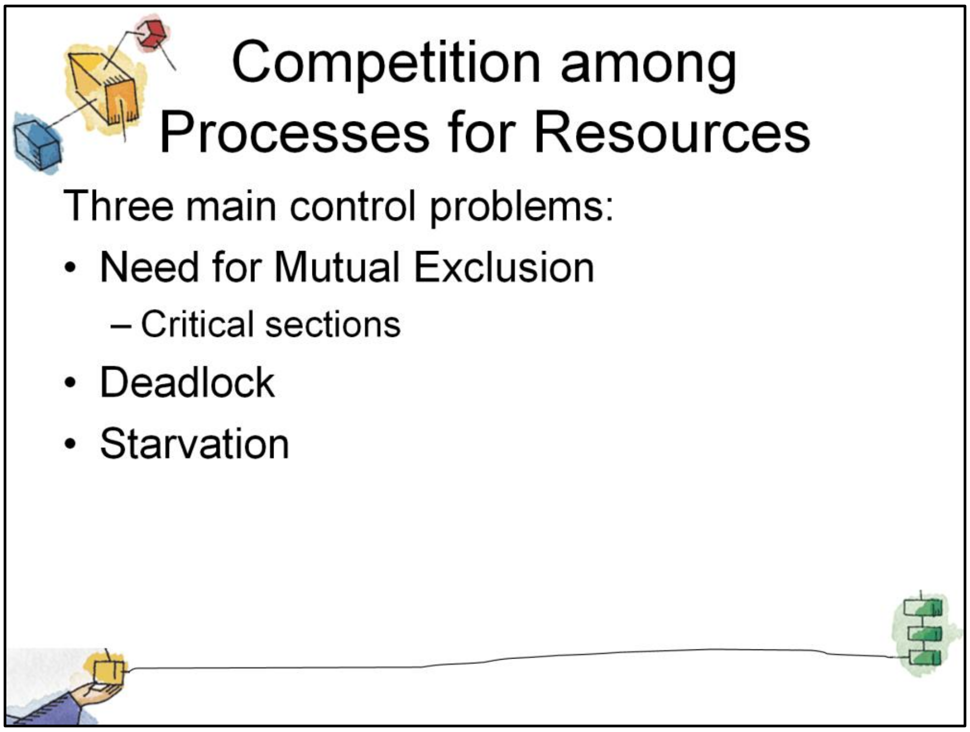
- Independent processes that are not intended to work together.
- E.G. multiprogramming of multiple independent processes.
- Although the processes are not working together, the OS needs to be concerned about competition for resources.
- E.G. two independent applications may both want to access the same disk or file or printer.

**Processes indirectly aware of each other:**

- Processes that are not necessarily aware of each other by their respective process IDs but that share access to some object, such as an I/O buffer.
- Such processes exhibit cooperation in sharing the common object.

**Processes directly aware of each other:**

- Processes that are able to communicate with each other by process ID and that are designed to work jointly on some activity.
- Again, such processes exhibit cooperation.



## Competition among Processes for Resources

Three main control problems:

- Need for Mutual Exclusion
  - – Critical sections
- Deadlock
- Starvation

**Need for mutual exclusion.**

•Suppose two or more processes require access to a single nonsharable resource, such as a printer.

• During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data.

• We will refer to such a resource as a **critical resource**, and the portion of the program that uses it a **critical section** of the program.

• It is important that only one program at a time be allowed in its critical section.

• We cannot simply rely on the OS to understand and enforce this restriction because the detailed requirements may not be obvious.

• In the case  of the printer, for example, we want any individual process to have control of the printer while it prints an entire file.

•Otherwise, lines from competing processes will be interleaved.

The enforcement of mutual exclusion creates two additional control problems:

**deadlock.**

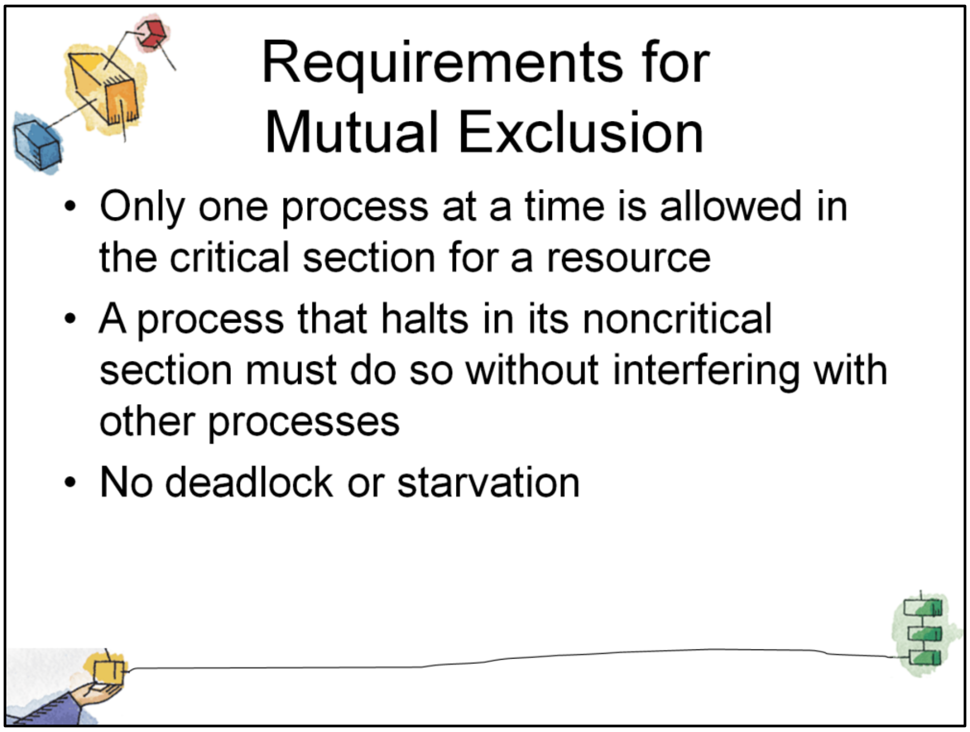
• Two processes is waiting for the same resources (or each waiting for a resource that the other has exclusive use to)

• Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources.

• The two processes are deadlocked.

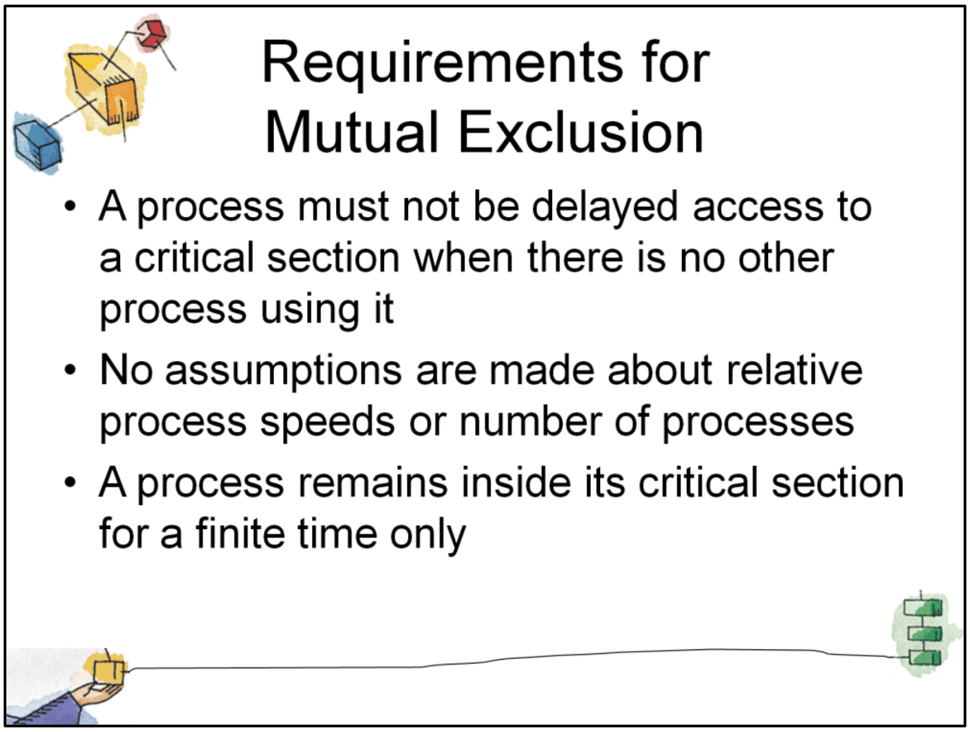
**starvation.**

• The OS may grant access to resources to a number of processes while neglecting another



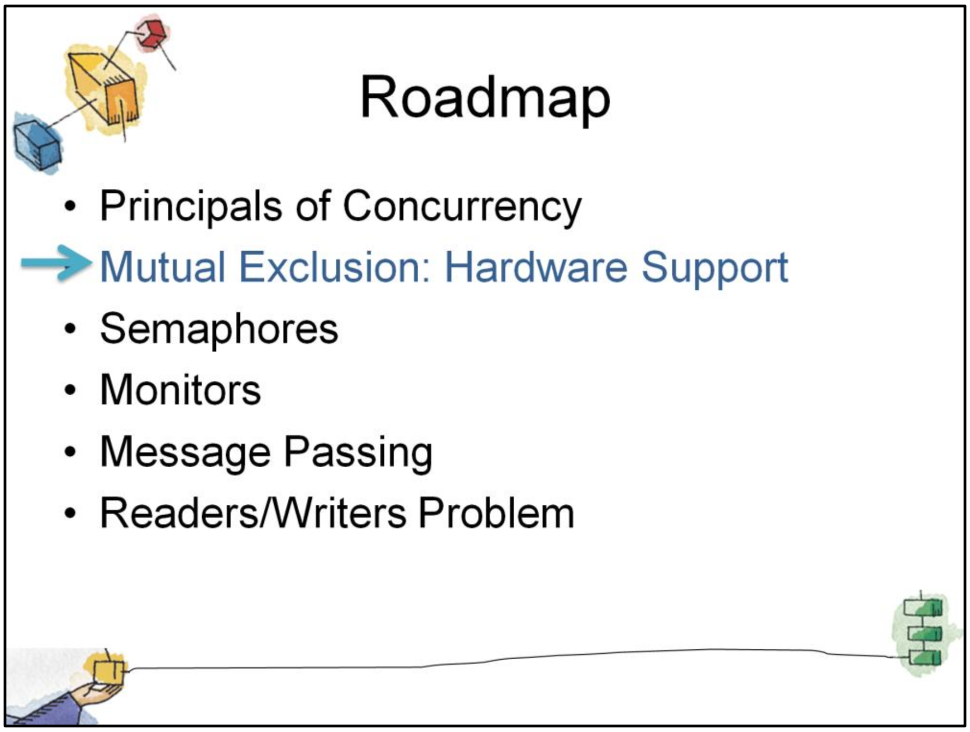
## Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation



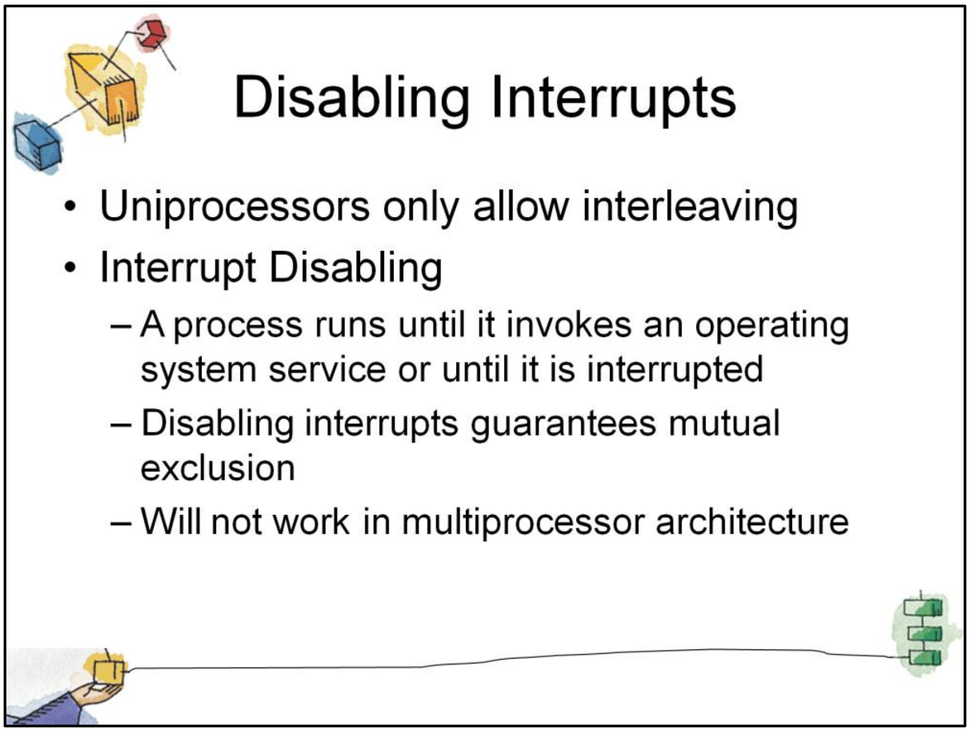
## Requirements for Mutual Exclusion

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only



## Roadmap

- Principals of Concurrency
- → Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

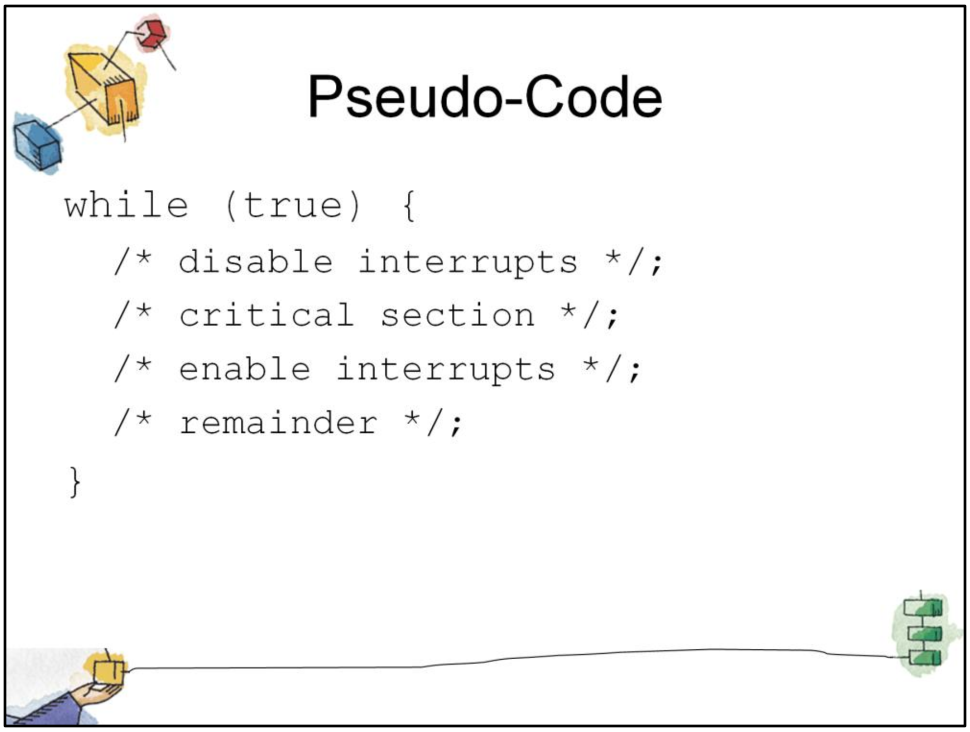


## Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved.

To guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted.



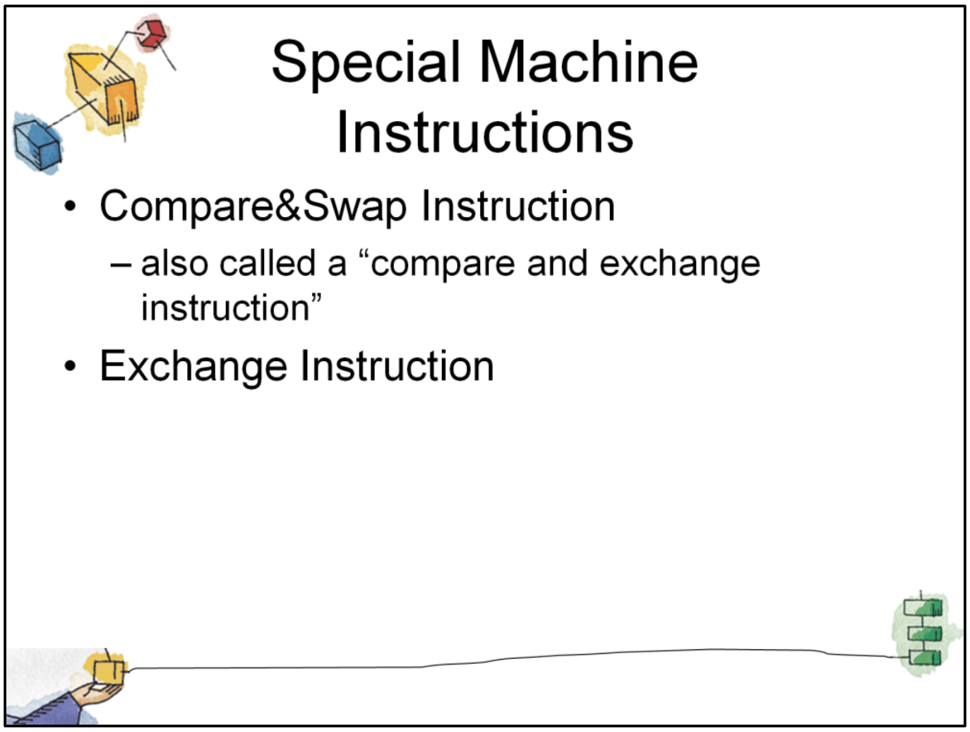
## Pseudo-Code

```
while (true) {
  /* disable interrupts */;
  /* critical section */;
  /* enable interrupts */;
  /* remainder */;
}
```

Because the critical section cannot be interrupted, mutual exclusion is guaranteed.

The price of this approach, however, is high.

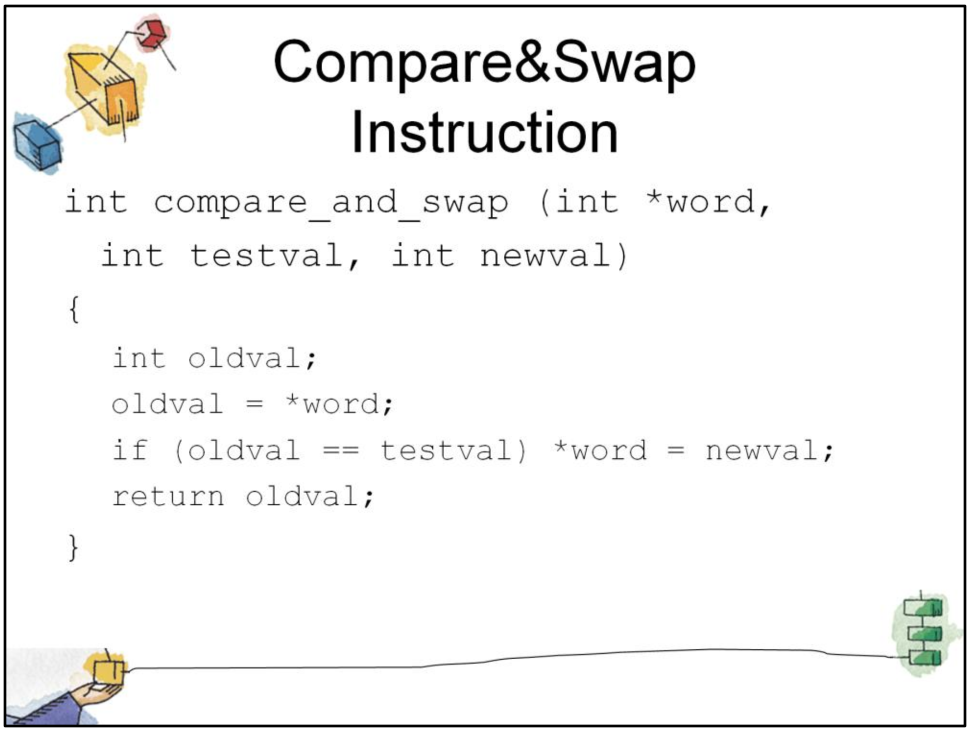
The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes.



## Special Machine Instructions

- Compare&Swap Instruction
  - also called a “compare and exchange instruction”
- Exchange Instruction

More details on these approaches in the following slides



## Compare&Swap Instruction

```
int compare_and_swap (int *word,
  int testval, int newval)
{
   int oldval;
   oldval = *word;
   if (oldval == testval) *word = newval;
   return oldval;
}
```

This version of the instruction checks a memory location (*word) against a test value (testval).

If the memory location's current value is testval, it is replaced with newval;

otherwise it is left unchanged.

The old memory value is always returned;

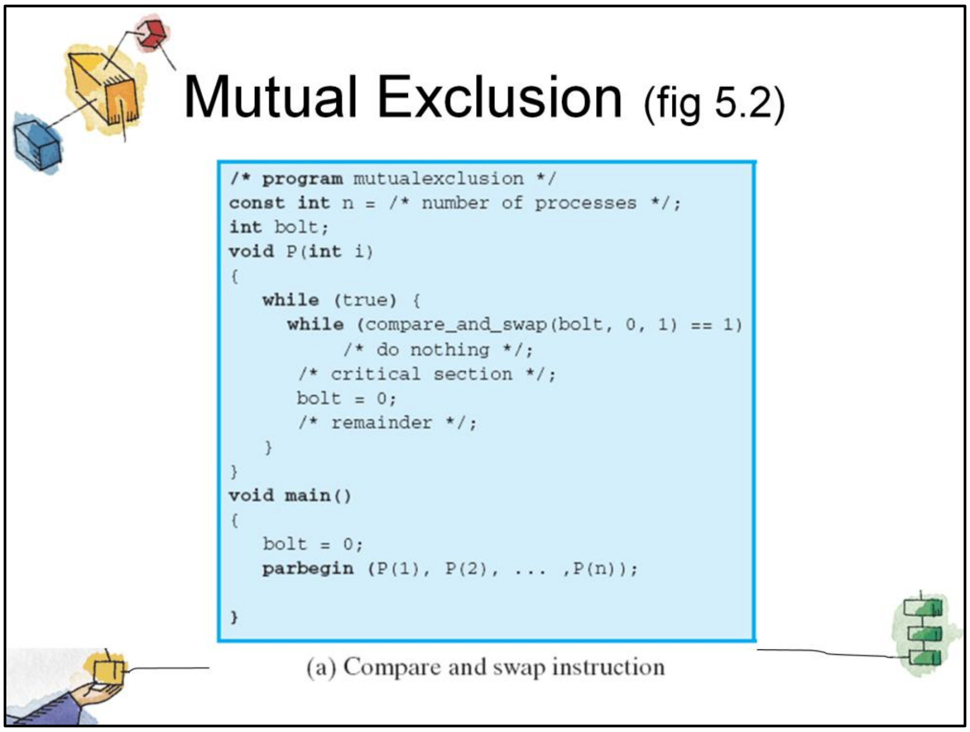
thus, the memory location has been updated if the returned value is the same as the test value.

1) A compare is made between a memory value and a test value;
2) if the values differ a swap occurs.

The entire compare&swap function is carried out atomically;

Ie  it is not subject

to interruption.



## Mutual Exclusion (fig 5.2)

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

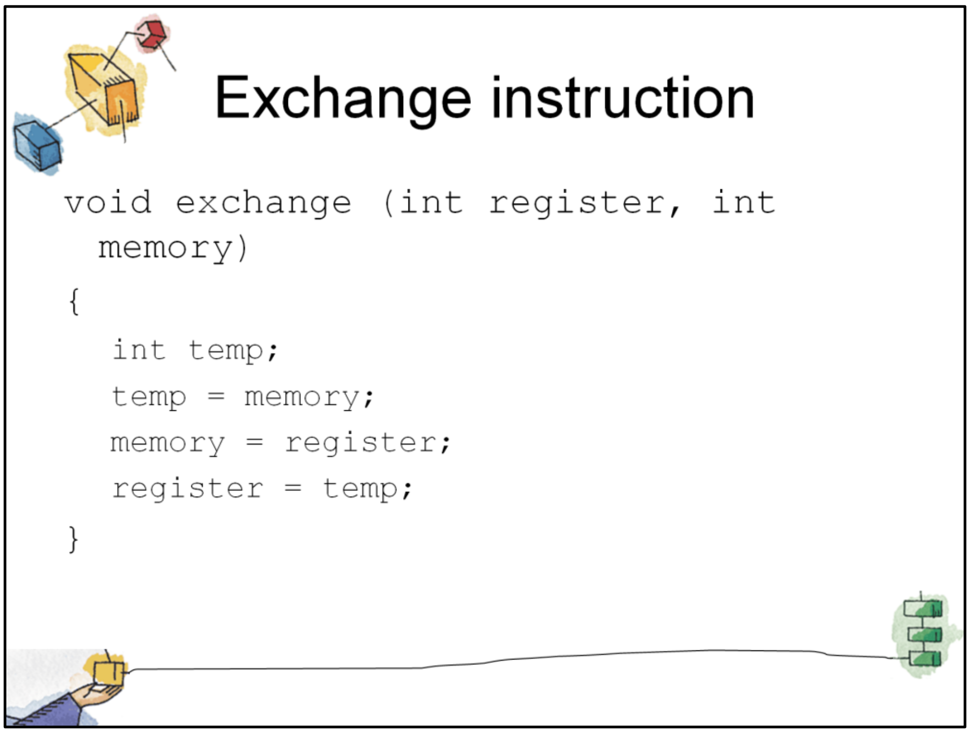
(a) Compare and swap instruction

This shows a mutual exclusion protocol based on the use of this instruction.

A shared variable ***bolt*** is initialized to 0.

The only process that may enter its critical section is one that finds bolt equal to 0.

All other processes that enter their critical section go into a busy waiting mode.

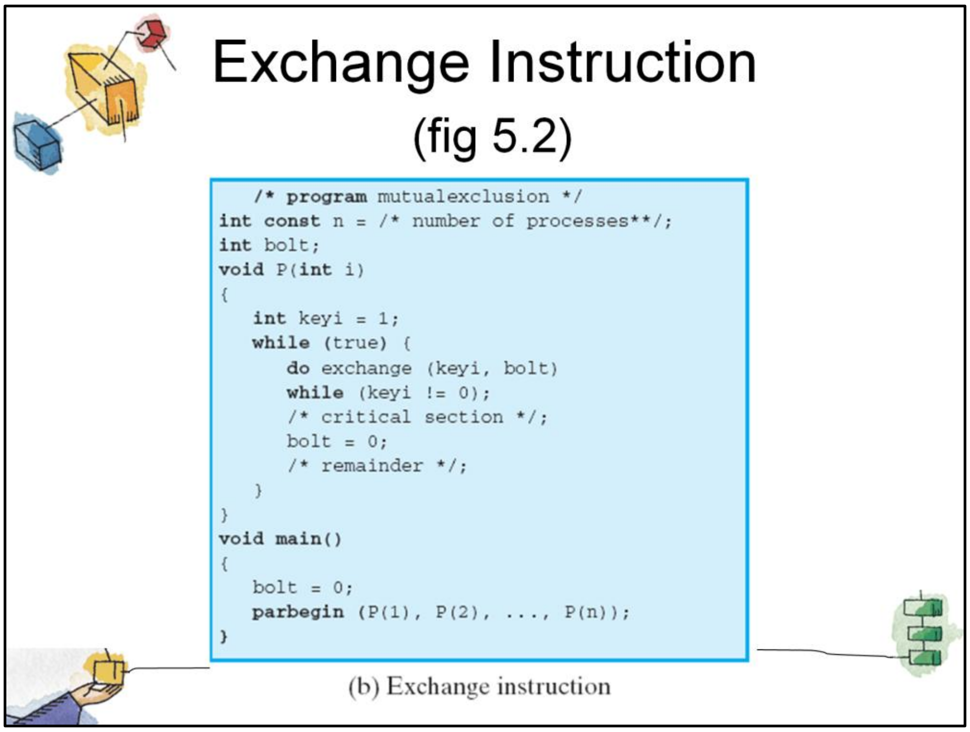


## Exchange instruction

```
void exchange (int register, int
  memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

The instruction exchanges the contents of a register with that of a memory location.

Both the Intel IA-32 architecture (Pentium) and the IA-64 architecture (Itanium) contain an XCHG instruction.



## Exchange Instruction

(fig 5.2)

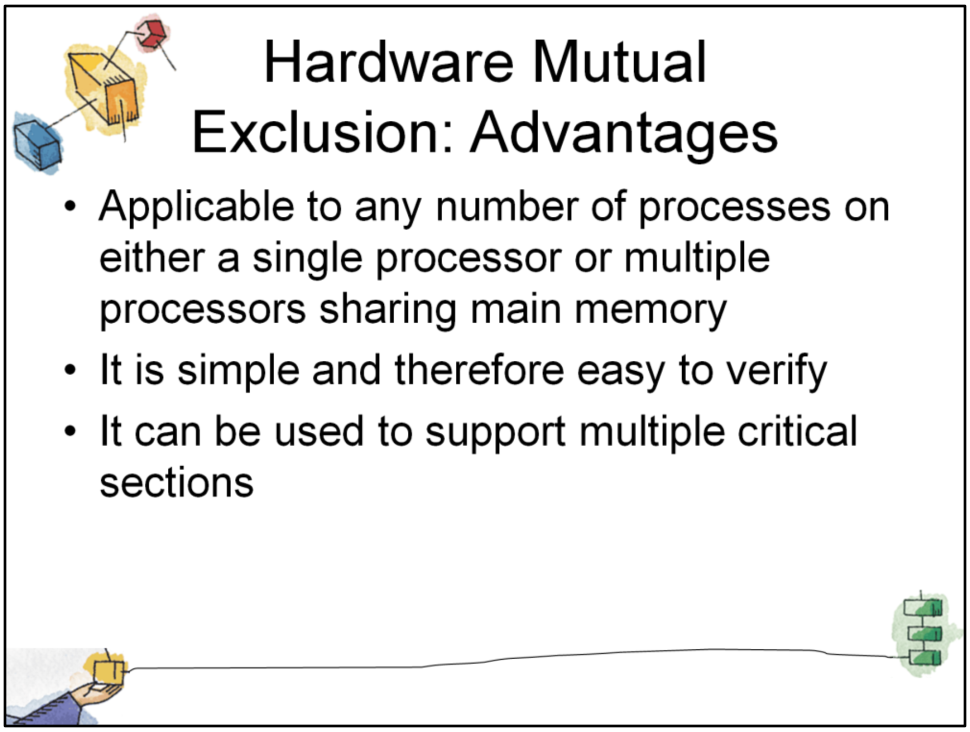
```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

This shows a mutual exclusion protocol based on the use of an exchange instruction.

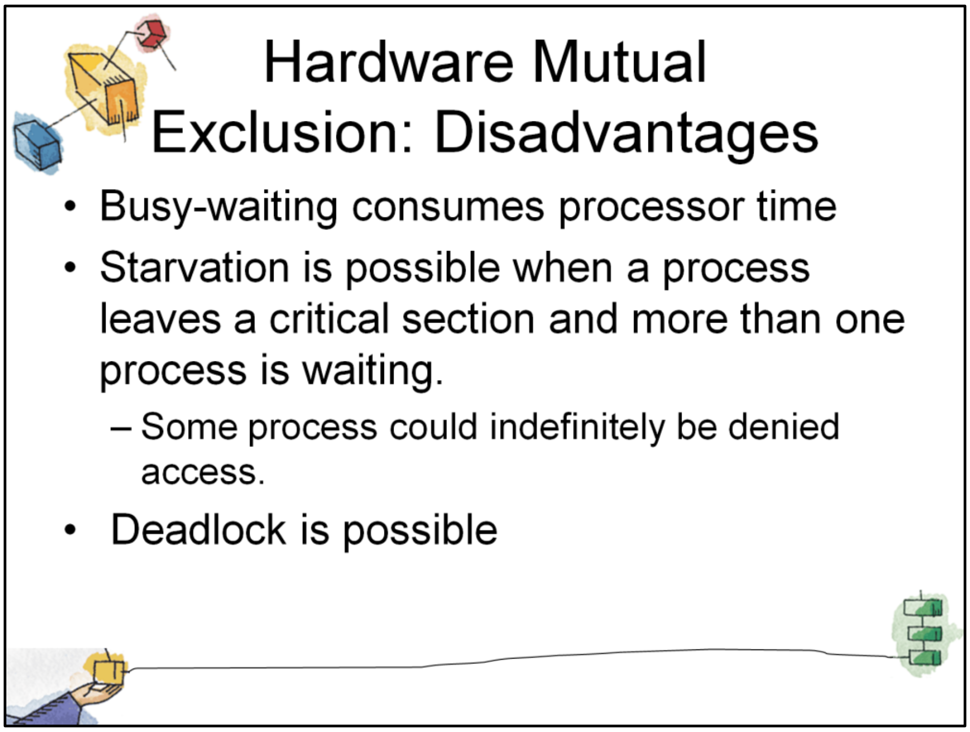
A shared variable bolt is initialized to 0.

- •Each process uses a local variable key that is initialized to 1.
- •The only process that may enter its critical section is one that finds bolt equal to 0.
- •It excludes all other processes from the critical section by setting bolt to 1.
- •When a process leaves its critical section, it resets bolt to 0, allowing another process to gain access to its critical section.



## Hardware Mutual Exclusion: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections



## Hardware Mutual Exclusion: Disadvantages

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Some process could indefinitely be denied access.
- Deadlock is possible

**Busy waiting**

While a process is waiting for access to a critical section, it continues to consume processor time.

**Starvation is possible.**

When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary.

Thus, some process could indefinitely be denied access.

**Deadlock is possible.**

Example (on a uniprocessor).

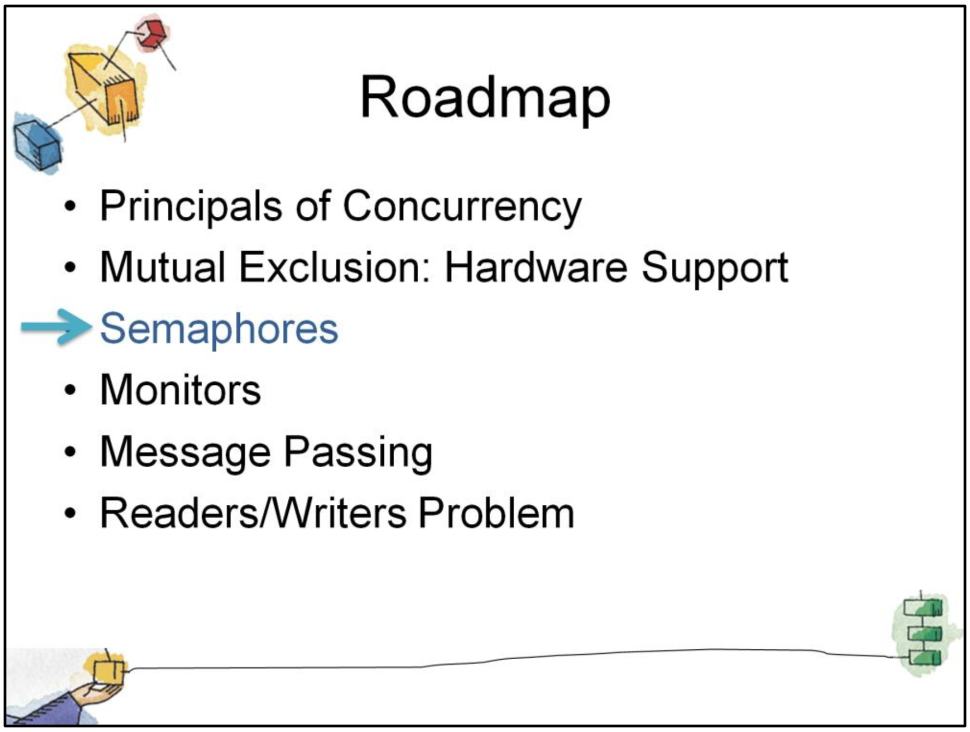
• Process P1 executes the special instruction (e.g., compare&swap, exchange) and enters its critical section.

• P1 is then interrupted to give the processor to P2, which has higher priority.

• If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism.

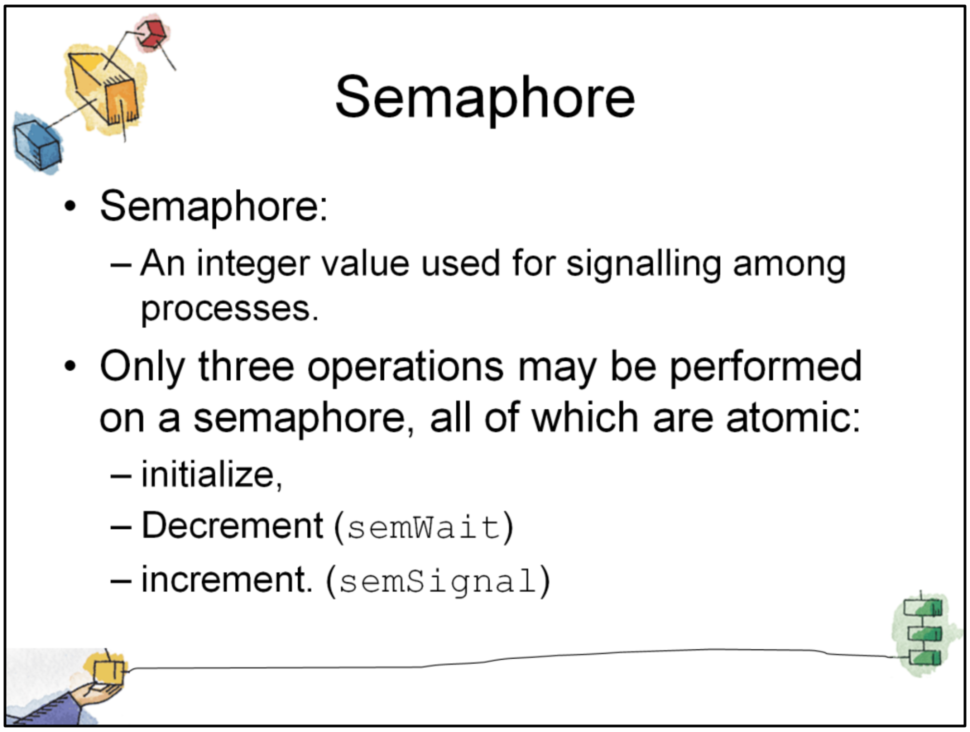
  • Thus it will go into a busy waiting loop.

•However, P1 will never be dispatched because it is of lower priority than another ready process, P2.



## Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- → Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem



## Semaphore

- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - initialize,
  - Decrement (`semWait`)
  - increment. (`semSignal`)

1. A semaphore may be initialized to a nonnegative integer value.

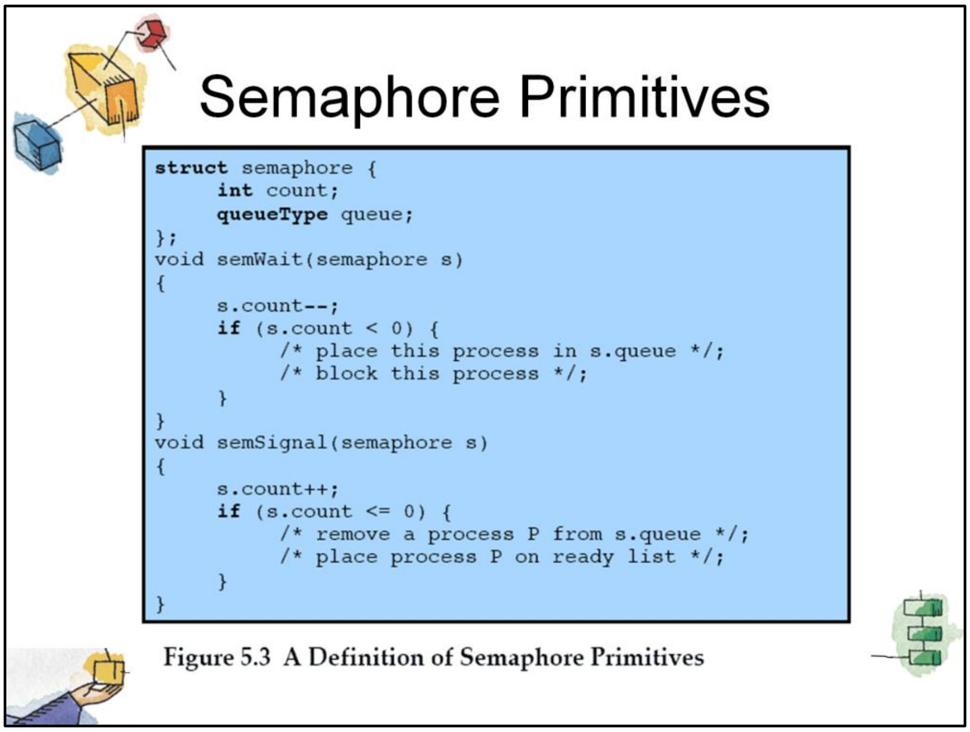
2. The semWait operation decrements the semaphore value.

   •If the value becomes negative, then the process executing the semWait is blocked.

   • Otherwise, the process continues execution.

3. The semSignal operation increments the semaphore value.

   • If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

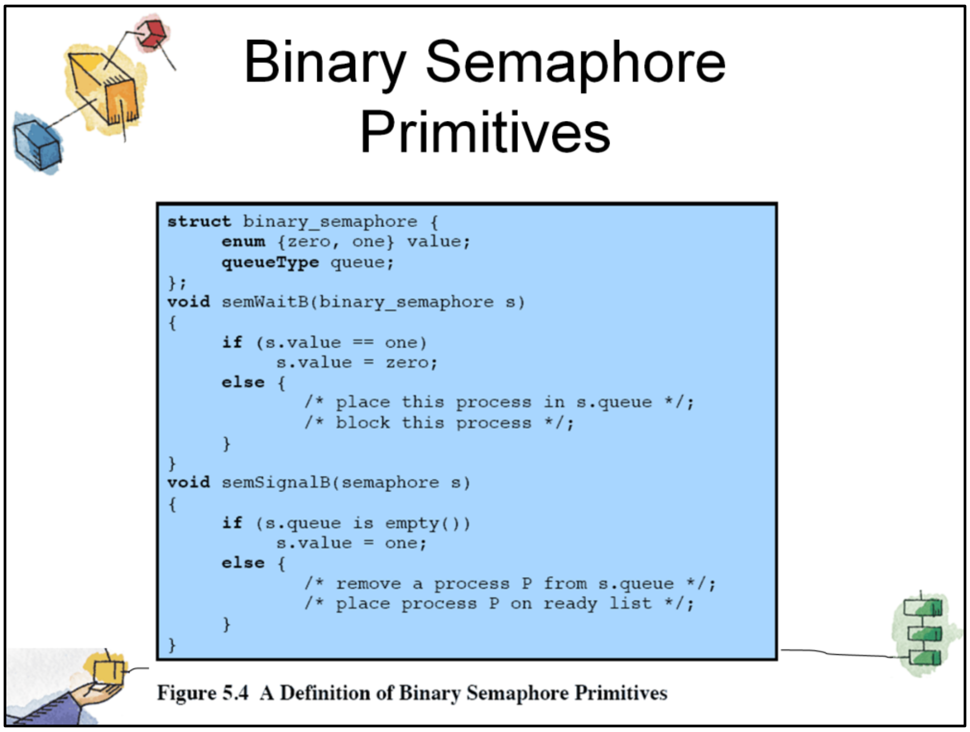


## Semaphore Primitives

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

An example of a semaphore



## Binary Semaphore Primitives

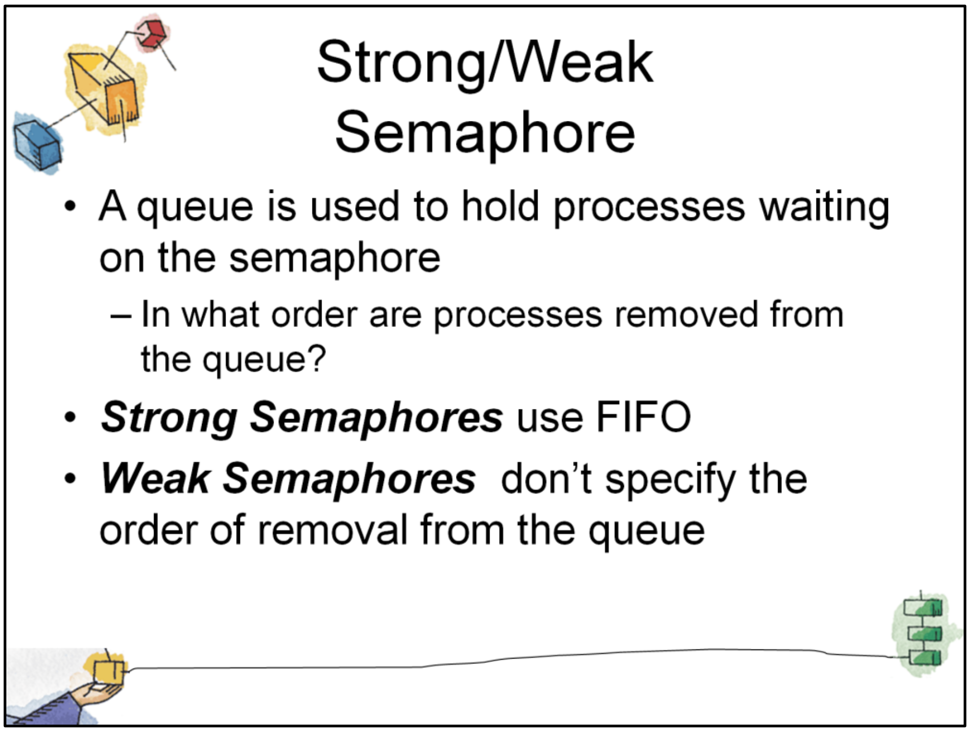
```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
             /* place this process in s.queue */;
             /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
             /* remove a process P from s.queue */;
             /* place process P on ready list */;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

A more restrictive semaphore which may only have the value of 0 or 1

A similar concept related to the binary semaphore is the **mutex**.

- A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
- In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.



## Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- ***Strong Semaphores*** use FIFO
- ***Weak Semaphores*** don't specify the order of removal from the queue

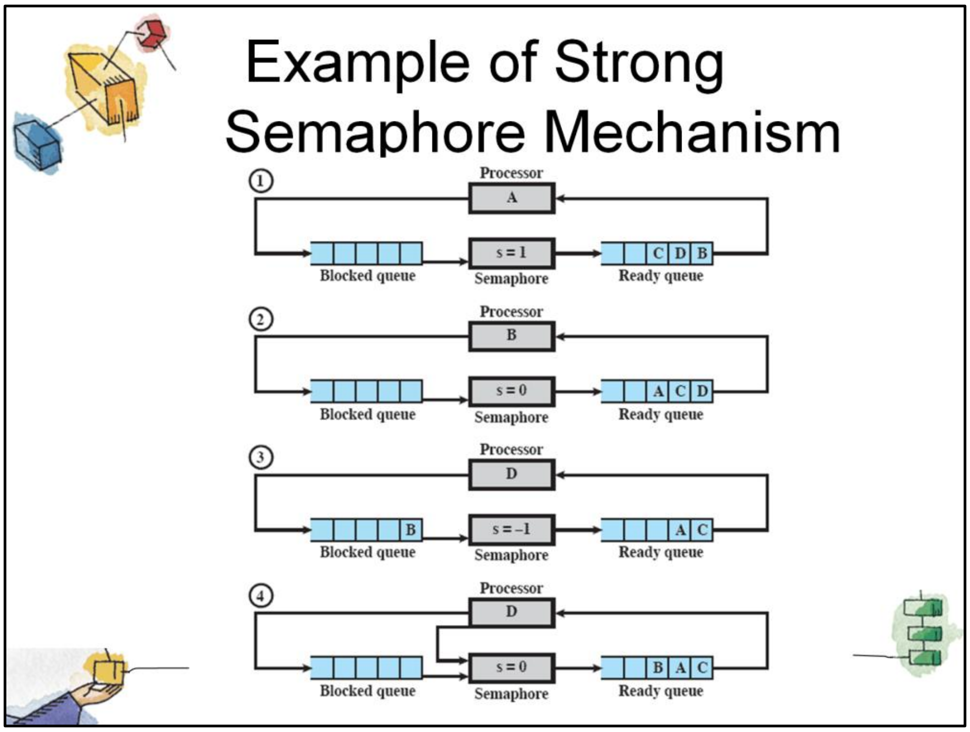
For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore.

The question arises of the order in which processes are removed from such a queue.

The fairest removal policy is first-in-first-out (FIFO):

•The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a **strong semaphore.**

•A semaphore that does not specify the order in which processes are removed from the queue is a weak semaphore.



## Example of Strong Semaphore Mechanism

**Animated Slide –** animation shows sections of the diagram to focus and remove distraction

Processes A, B, and C depend on a result from process D.

Initially (1), A is running;

- B, C, and D are ready;
- the semaphore count is 1, indicating that one of D's results is available.
- When A issues a semWait instruction on semaphore **s**, the semaphore decrements to 0, and A can continue to execute;
- subsequently it rejoins the ready queue.

Then B runs (2), eventually issues a semWait instruction, and is blocked,

allowing D to run (3).

When D completes a new result, it issues a semSignal instruction,

which allows B to move to the ready queue (4).

D rejoins the ready queue and …

## Example of Semaphore Mechanism

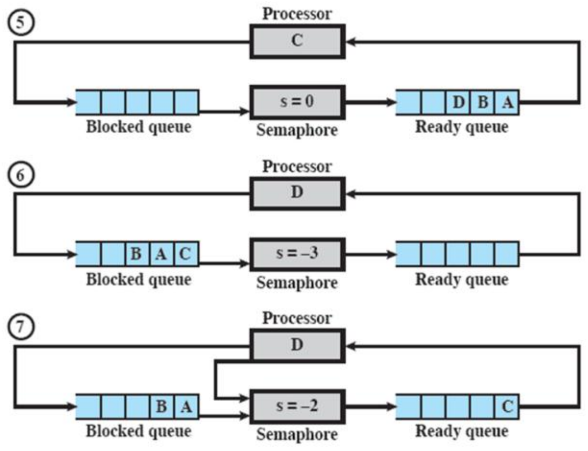
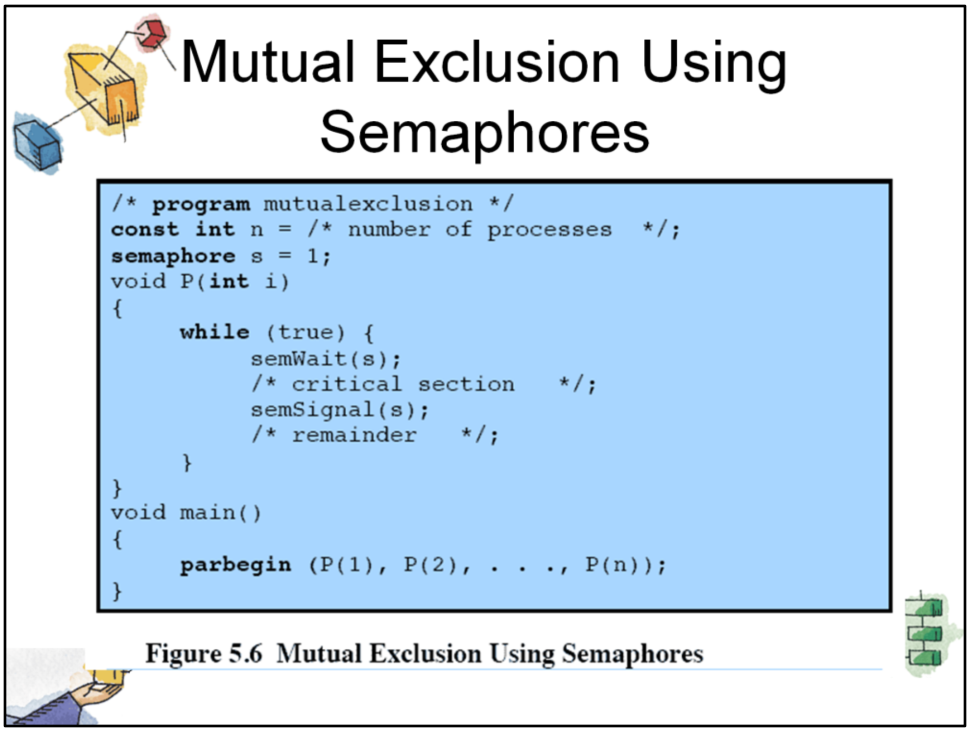


**Figure 5.5 Example of Semaphore Mechanism**

C begins to run (5)

- but is blocked when it issues a semWait instruction.
- Similarly, A and B run and are blocked on the semaphore,

allowing D to resume execution (6).When D has a result, it issues a semSignal, which transfers C to the ready queue. Later cycles of D will release A and B from the Blocked state.



## Mutual Exclusion Using Semaphores

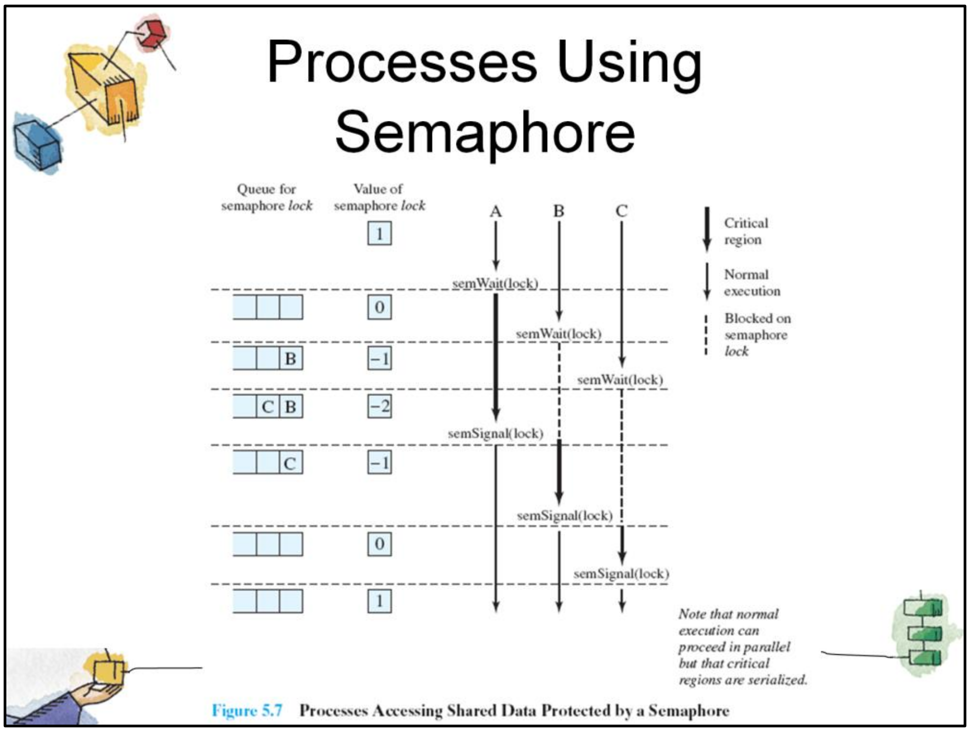
```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

This shows a straightforward solution to the mutual exclusion problem using a semaphore

Consider n processes, identified in the array P(i), all of which need access to the same resource s.

- Each process has a critical section used to access the resource.
- In each process, a semWait(s) is executed just before its critical section.
    - If the value of s becomes negative, the process is blocked.
    - If the value is 1, then it is decremented to 0 and the process immediately enters its critical section;
- because s is no longer positive, no other process will be able to enter its critical section.



## Processes Using Semaphore

Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
1
semWait(lock)
0
semWait(lock)
B
−1
semWait(lock)
C B
−2
semSignal(lock)
C
−1
semSignal(lock)
0
semSignal(lock)
1
Critical region
Normal execution
Blocked on semaphore *lock*
*Note that normal execution can proceed in parallel but that critical regions are serialized.*

**Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore**

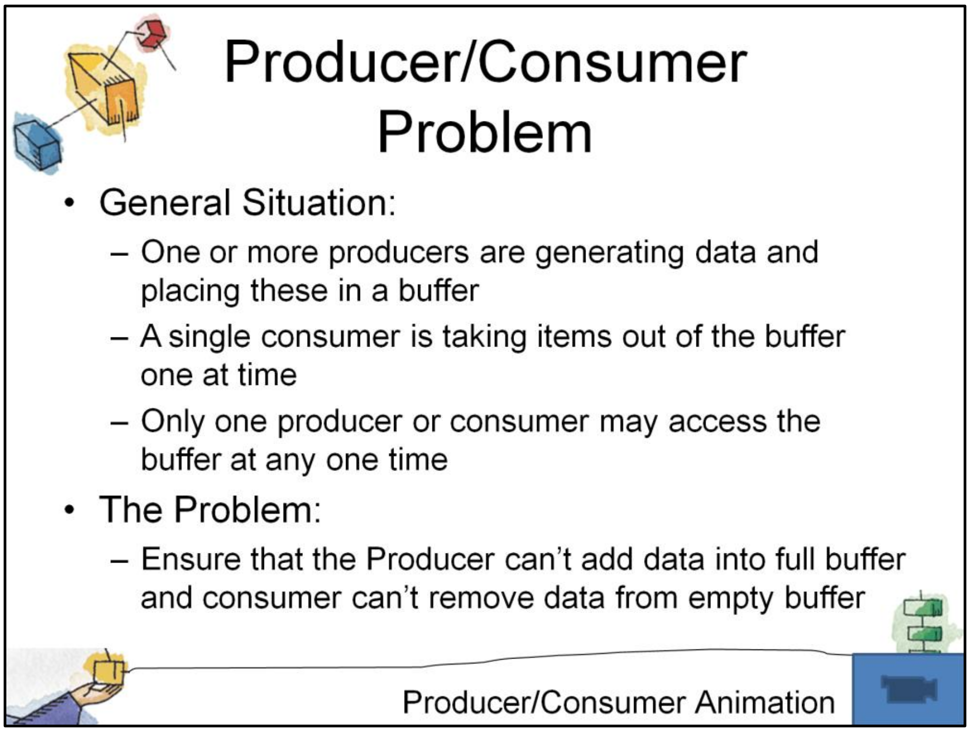
Figure 5.7, shows a possible sequence for three processes using the mutual exclusion discipline of Figure 5.6.

Three processes (A,B, C) access a shared resource protected by the semaphore *lock.*

- *Process*-A executes semWait(lock);
  - because the semaphore has a value of 1 at the time of the semWait operation, A can immediately enter its critical section and the semaphore takes on the value 0.

- While A is in its critical section, both B and C perform a semWait operation and are blocked pending the availability of the semaphore.

- When A exits its critical section and performs semSignal(lock), B, which was the first process in the queue, can now enter its critical section.



## Producer/Consumer Problem

- General Situation:
  - One or more producers are generating data and placing these in a buffer
  - A single consumer is taking items out of the buffer one at time
  - Only one producer or consumer may access the buffer at any one time
- The Problem:
  - Ensure that the Producer can't add data into full buffer and consumer can't remove data from empty buffer

Producer/Consumer Animation

The general statement is this:

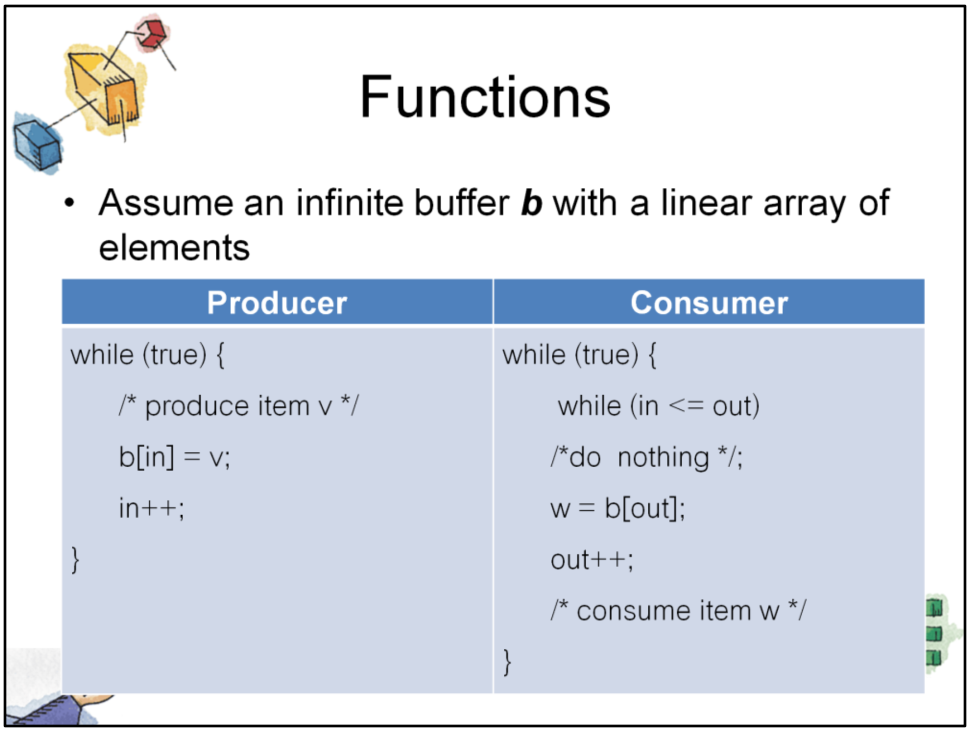
• There are one or more producers generating some type of data (records, characters) and placing these in a buffer.

• There is a single consumer that is taking items out of the buffer one at a time.

•The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

We will look at a number of solutions to this problem to illustrate both the power and the pitfalls of semaphores.

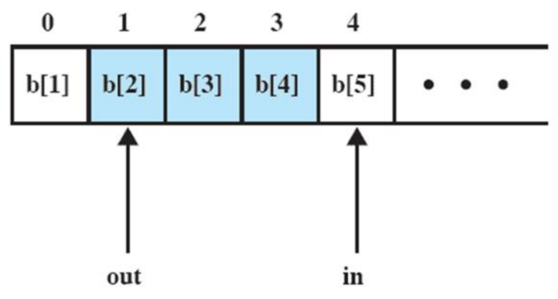


## Functions

- Assume an infinite buffer ***b*** with a linear array of elements

| Producer | Consumer |
|---|---|
| while (true) { | while (true) { |
|     /* produce item v */ |     while (in <= out) |
|     b[in] = v; |     /*do  nothing */; |
|     in++; |     w = b[out]; |
| } |     out++; |
| |     /* consume item w */ |
| | } |

## Buffer



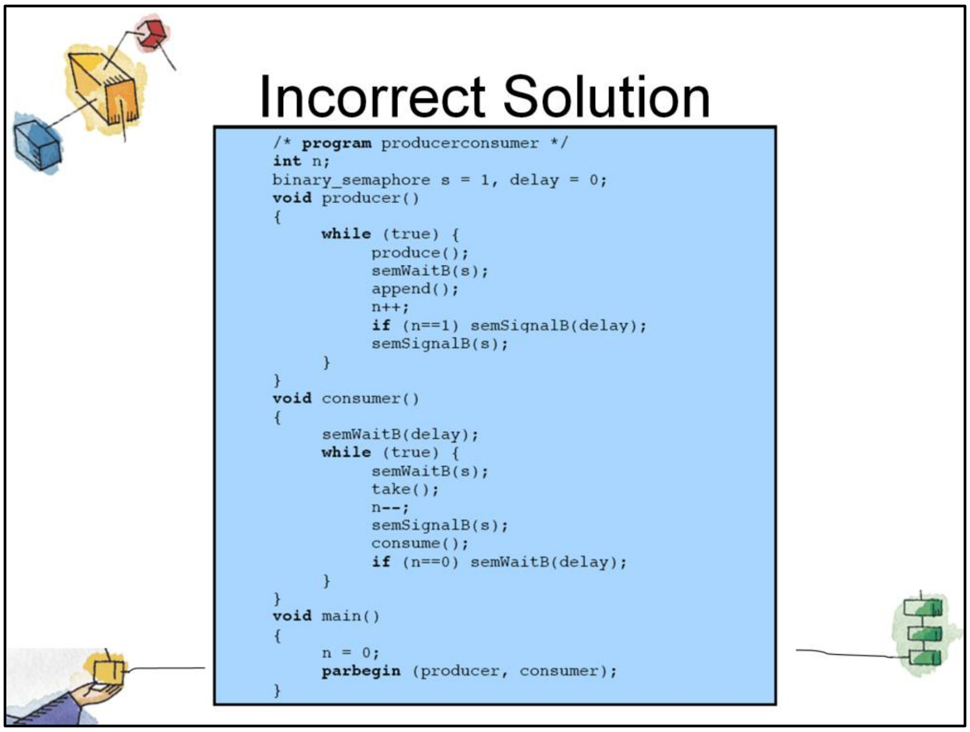
Note: shaded area indicates portion of buffer that is occupied

**Figure 5.8 Infinite Buffer for the Producer/Consumer Problem**

The producer can generate items and store them in the buffer at its own pace.

Each time, an index (in) into the buffer is incremented.

The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer.



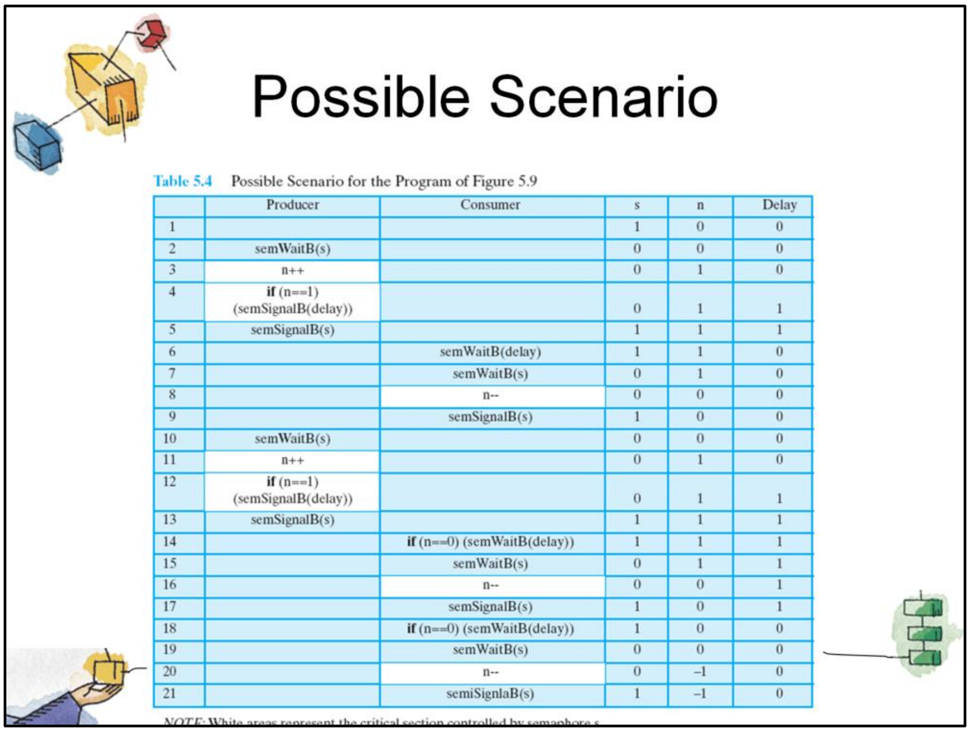
## Incorrect Solution

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Rather than deal with the indices in and out, we can simply keep track of the number of items in the buffer, using the integer variable n (= in - out).

The semaphore *s* is used to enforce mutual exclusion;

the semaphore *delay* is used to force the consumer to semWait if the buffer is empty.



## Possible Scenario

Table 5.4 Possible Scenario for the Program of Figure 5.9

| | Producer | Consumer | s | n | Delay |
|---|---|---|---|---|---|
| 1 | | | 1 | 0 | 0 |
| 2 | semWaitB(s) | | 0 | 0 | 0 |
| 3 | n++ | | 0 | 1 | 0 |
| 4 | **if** (n==1) (semSignalB(delay)) | | 0 | 1 | 1 |
| 5 | semSignalB(s) | | 1 | 1 | 1 |
| 6 | | semWaitB(delay) | 1 | 1 | 0 |
| 7 | | semWaitB(s) | 0 | 1 | 0 |
| 8 | | n-- | 0 | 0 | 0 |
| 9 | | semSignalB(s) | 1 | 0 | 0 |
| 10 | semWaitB(s) | | 0 | 0 | 0 |
| 11 | n++ | | 0 | 1 | 0 |
| 12 | **if** (n==1) (semSignalB(delay)) | | 0 | 1 | 1 |
| 13 | semSignalB(s) | | 1 | 1 | 1 |
| 14 | | **if** (n==0) (semWaitB(delay)) | 1 | 1 | 1 |
| 15 | | semWaitB(s) | 0 | 1 | 1 |
| 16 | | n-- | 0 | 0 | 1 |
| 17 | | semSignalB(s) | 1 | 0 | 1 |
| 18 | | **if** (n==0) (semWaitB(delay)) | 1 | 0 | 0 |
| 19 | | semWaitB(s) | 0 | 0 | 0 |
| 20 | | n-- | 0 | –1 | 0 |
| 21 | | semiSignlaB(s) | 1 | –1 | 0 |

*NOTE:* White areas represent the critical section controlled by semaphore s.

There is, however, a flaw in this program.

When the consumer has exhausted the buffer, it needs to reset the delay semaphore so that it will be forced to wait until the producer has placed more items in the buffer.

This is the purpose of the statement: if n == 0 semWaitB (delay).

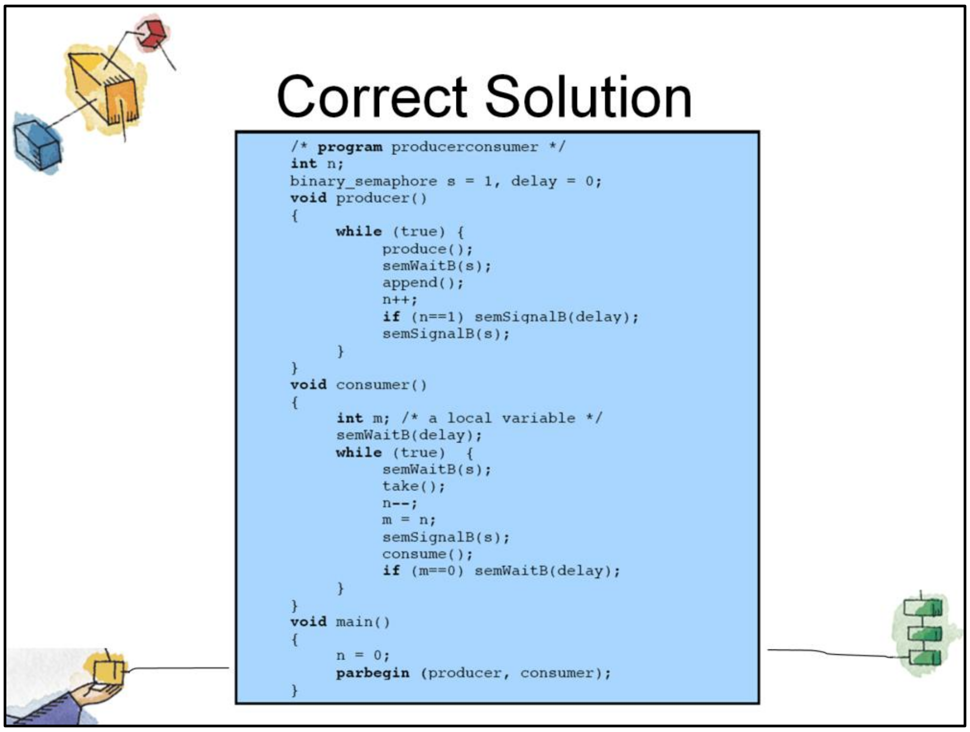
In line 14, the consumer fails to execute the semWaitB operation.

The consumer did indeed exhaust the buffer and set *n* to 0 (line 8),

but the producer has incremented *n* before the consumer can test it in line 14.

The result is a semSignalB not matched by a prior semWaitB.

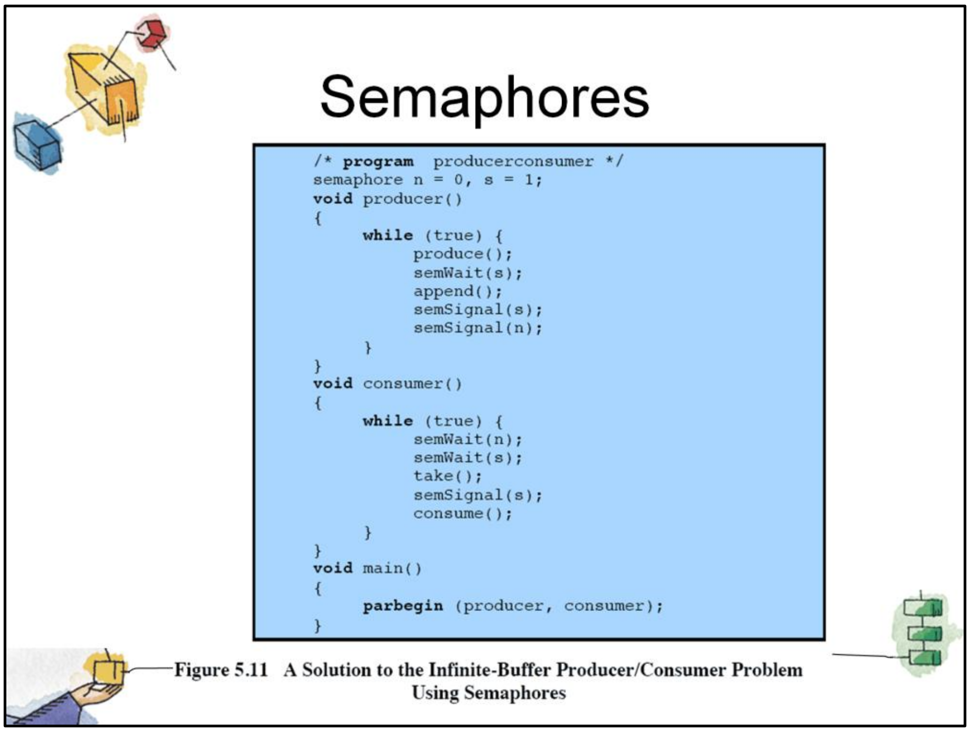
The value of -1 for *n* in line 20 means that the consumer has consumed an item from the buffer that does not exist. It would not do simply to move the conditional statement inside the critical section of the consumer because this could lead to deadlock (e.g., after line 8 of the table).



## Correct Solution

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

A fix for the problem is to introduce an auxiliary variable that can be set in the consumer's critical section for use later on.



## Semaphores

```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

**Figure 5.11** A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

A somewhat cleaner solution can be obtained if general semaphores (also called counting semaphores) are used

The variable *n* is now a semaphore.

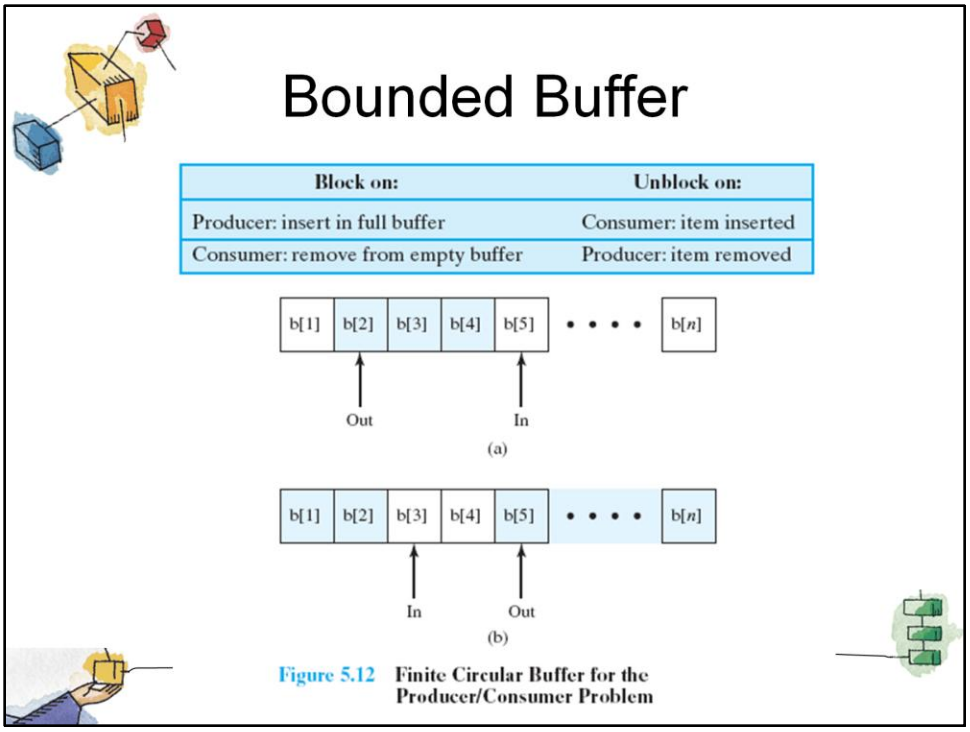
Its value still is equal to the number of items in the buffer.

Suppose now that in transcribing this program, a mistake is made and the operations *semSignal(s)* and *semSignal(n)* are interchanged.

This would require that the *semSignal(n)* operation be performed in the producer’s critical section without interruption by the consumer or another producer.

Would this affect the program?

No, because the consumer must wait on both semaphores before proceeding in any case.



## Bounded Buffer

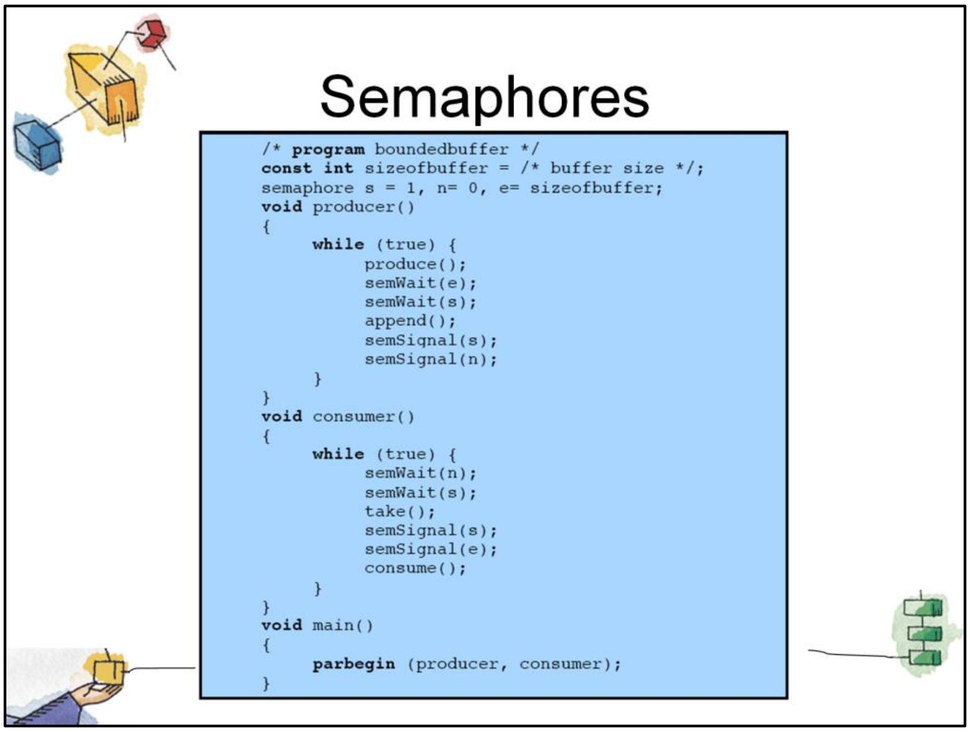
| Block on: | Unblock on: |
|---|---|
| Producer: insert in full buffer | Consumer: item inserted |
| Consumer: remove from empty buffer | Producer: item removed |

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

Finally, let us add a new and realistic restriction to the producer/consumer problem:

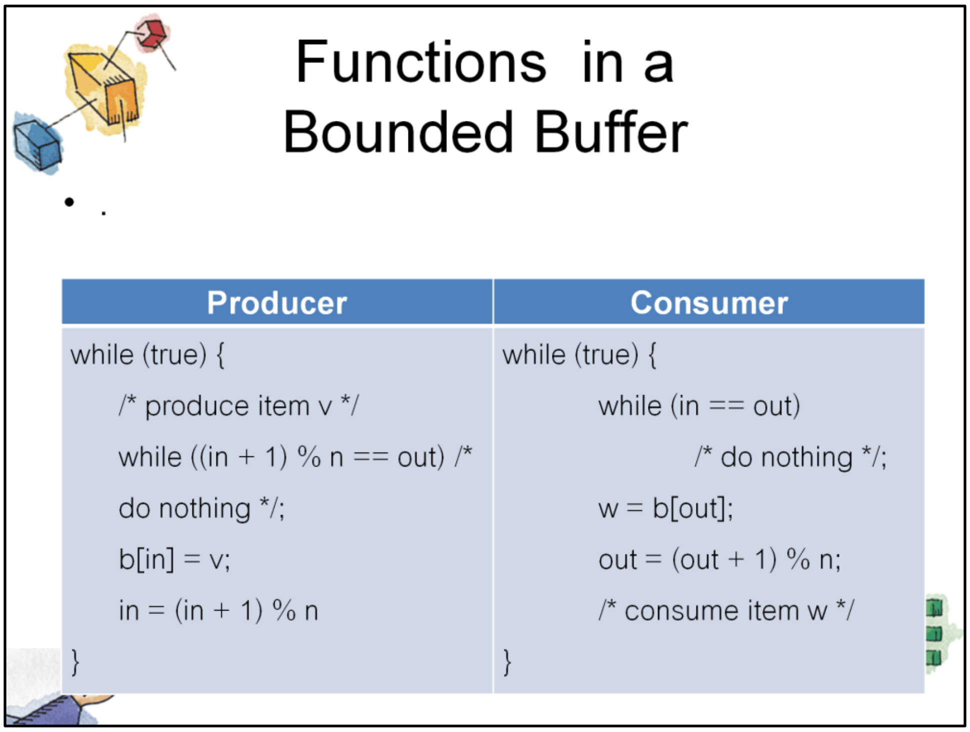
namely, **that the buffer is finite.**

The buffer is treated as a circular storage, and pointer values must be expressed modulo the size of the buffer.



## Semaphores

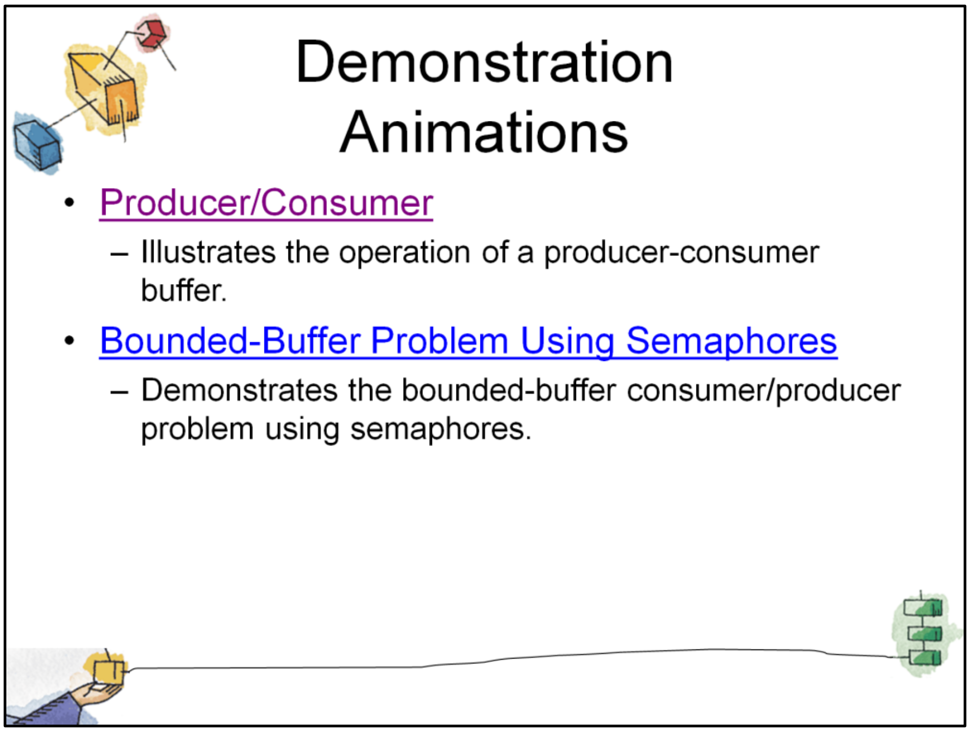
```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```



## Functions in a Bounded Buffer

- .

| Producer | Consumer |
|---|---|
| while (true) { | while (true) { |
| /* produce item v */ | while (in == out) |
| while ((in + 1) % n == out) /* | /* do nothing */; |
| do nothing */; | w = b[out]; |
| b[in] = v; | out = (out + 1) % n; |
| in = (in + 1) % n | /* consume item w */ |
| } | } |



## Demonstration Animations

- Producer/Consumer
  - Illustrates the operation of a producer-consumer buffer.
- Bounded-Buffer Problem Using Semaphores
  - Demonstrates the bounded-buffer consumer/producer problem using semaphores.

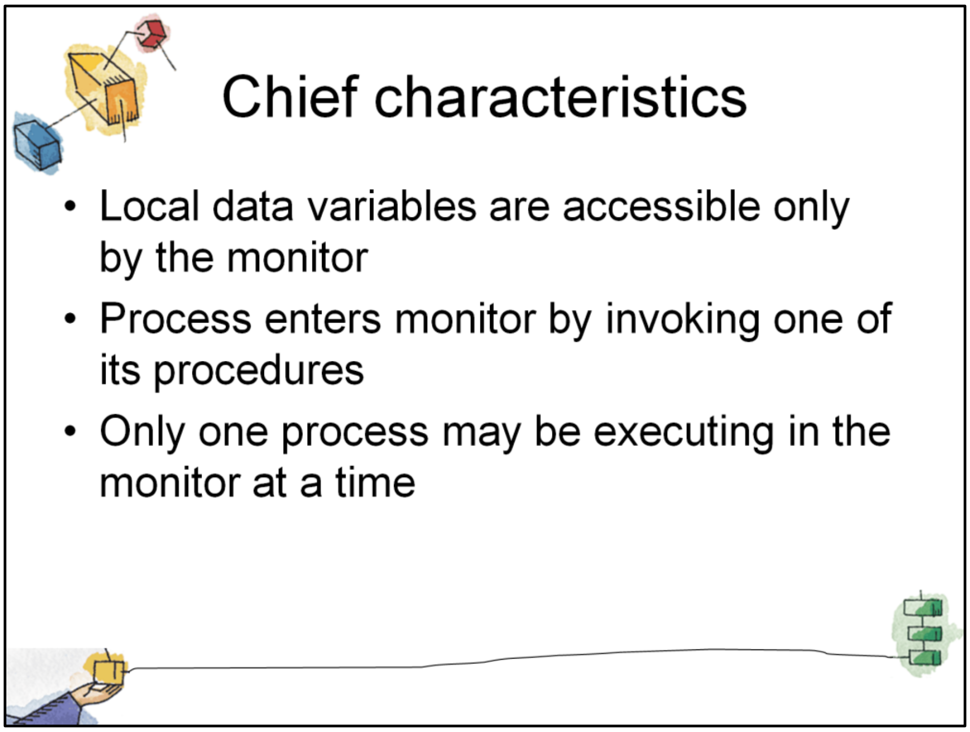
Introduction to the Animations http://williamstallings.com/OS/Animations-intro.html

## Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- → Monitors
- Message Passing
- Readers/Writers Problem

## Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, and Java.



## Chief characteristics

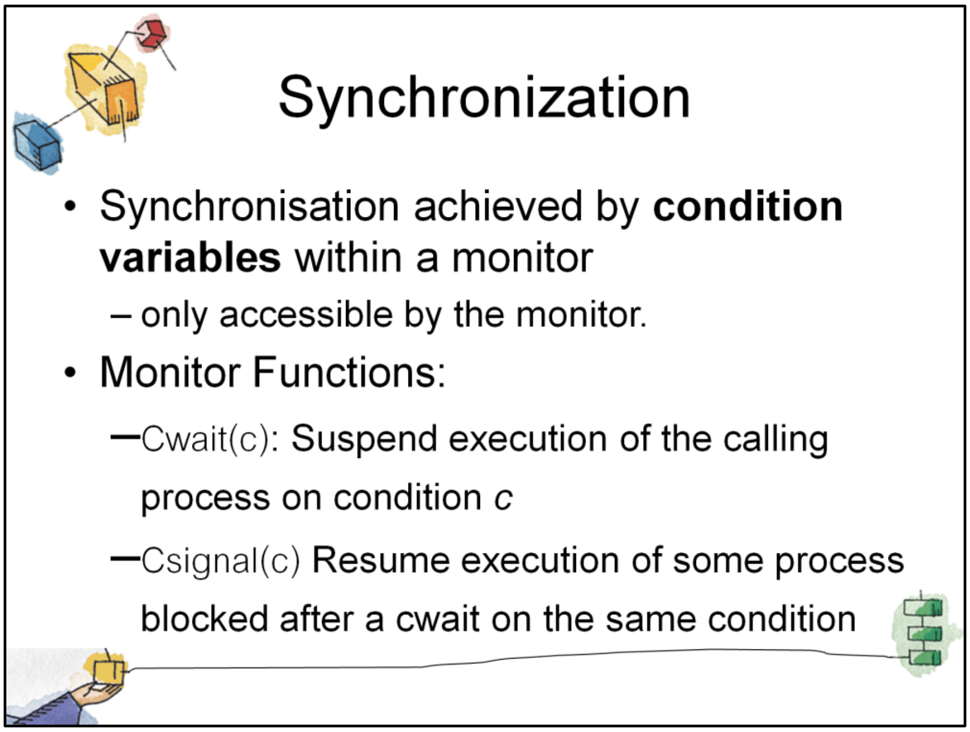
- Local data variables are accessible only by the monitor
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time

The chief characteristics of a monitor are the following:

1. The local data variables are accessible only by the monitor's procedures and not by any external procedure.

2. A process enters the monitor by invoking one of its procedures.

3. Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.



## Synchronization

- Synchronisation achieved by **condition variables** within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - Cwait(c): Suspend execution of the calling process on condition *c*
  - Csignal(c) Resume execution of some process blocked after a cwait on the same condition

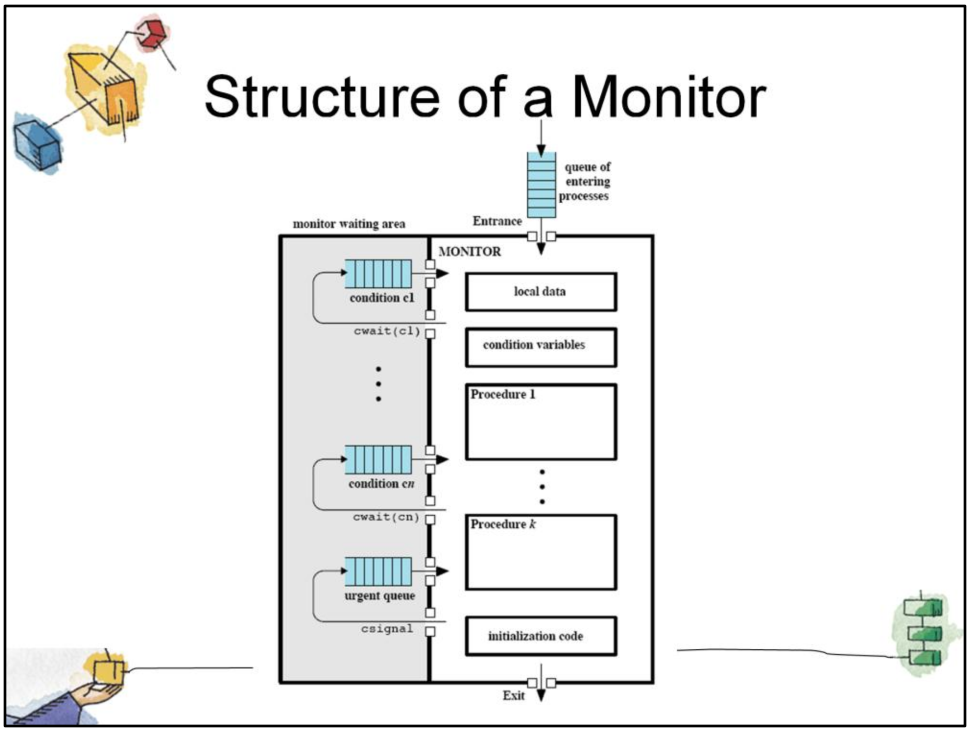
A monitor supports synchronization by the use of **condition variables that are** contained within the monitor and accessible only within the monitor.

cwait(c): Suspend execution of the calling process on condition *c.*

The monitor is now available for use by another process.

csignal(c): Resume execution of some process blocked after a cwait on the same condition.

If there are several such processes, choose one of them; if there is no such process, do nothing.



## Structure of a Monitor

Although a process can enter the monitor by invoking any of its procedures, we can think of the monitor as having a single entry point that is guarded so that only one process may be in the monitor at a time.

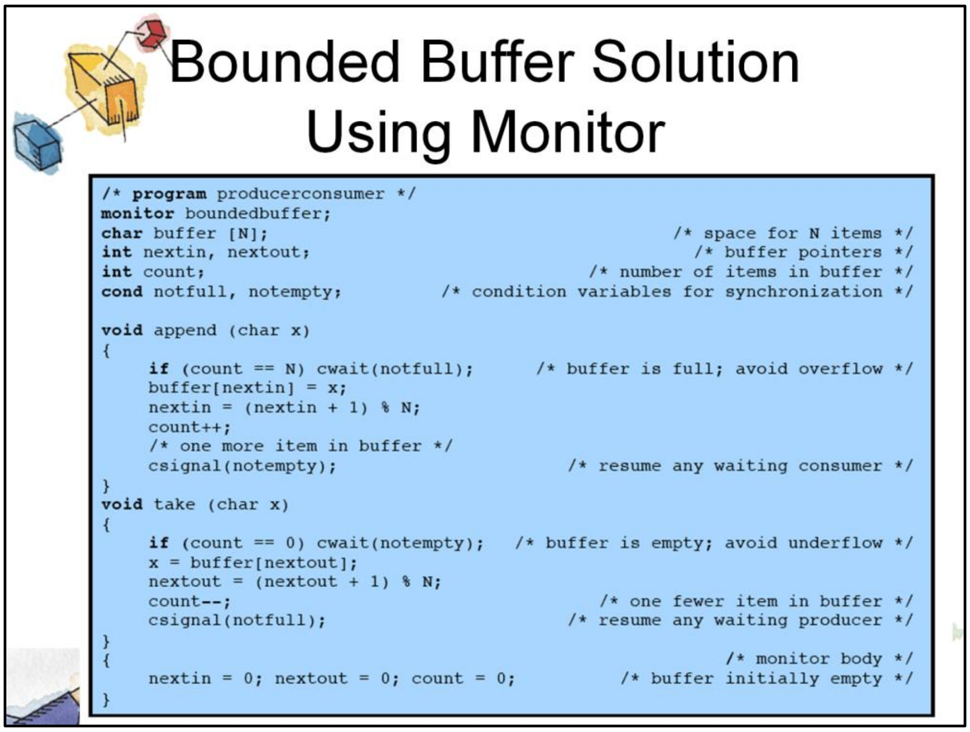
•Other processes that attempt to enter the monitor join a queue of processes blocked waiting for monitor availability.

Once a process is in the monitor, it may temporarily block itself on condition x by issuing cwait(x);

• it is then placed in a queue of processes waiting to re-enter the monitor when the condition changes, and resume execution at the point in its program following the cwait(x) call.

If a process that is executing in the monitor detects a change in the condition variable x, it issues csignal(x),

which alerts the corresponding condition queue that the condition has changed.



## Bounded Buffer Solution Using Monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                          /* space for N items */
int nextin, nextout;                                        /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;            /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                          /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                       /* one fewer item in buffer */
     csignal(notfull);                           /* resume any waiting producer */
}
{                                                               /* monitor body */
     nextin = 0; nextout = 0; count = 0;              /* buffer initially empty */
}
```

Returning to the bounded-buffer producer/consumer problem – this is a solution using a monitor.

The module, *boundedbuffer*, controls the buffer used to store and retrieve characters.

The monitor includes two condition variables (declared with the construct cond):

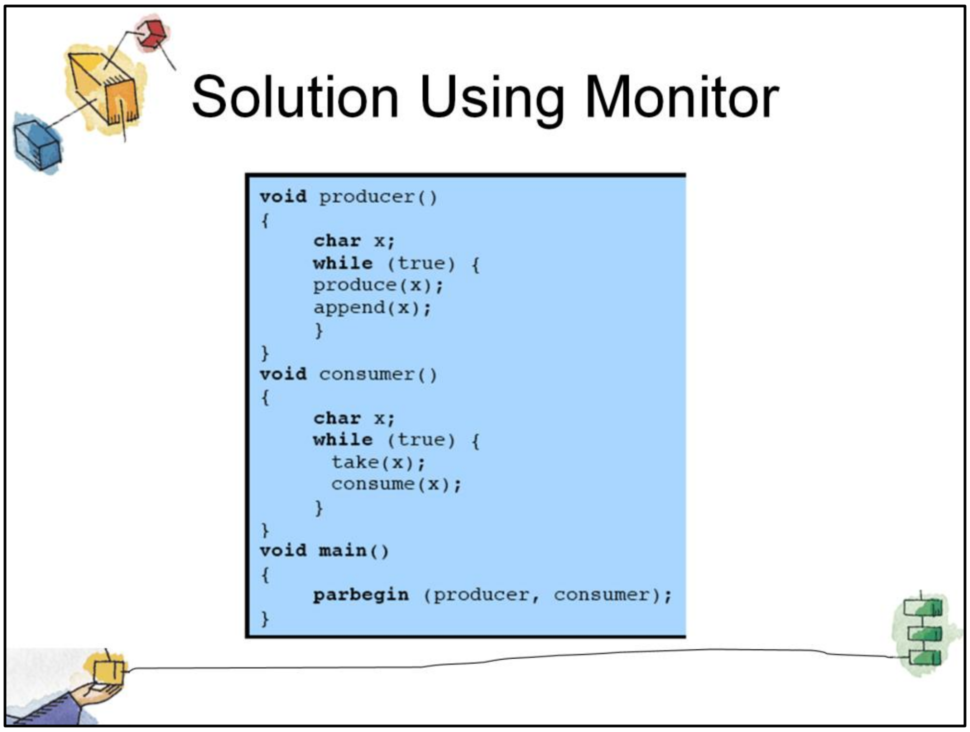
- *notfull* is true when there is room to add at least one character to the buffer,
- and *notempty* is true when there is at least one character in the buffer.

This example points out the division of responsibility with monitors compared to semaphores.

In the case of monitors, the monitor construct itself enforces mutual exclusion:

- It is not possible for both a producer and a consumer simultaneously to access the buffer.
- However, the programmer must place the appropriate *cwait* and *csignal* primitives inside the monitor to prevent processes from depositing items in a full buffer or removing them from an empty one.

In the case of semaphores, both mutual exclusion and synchronization are the responsibility of the programmer.

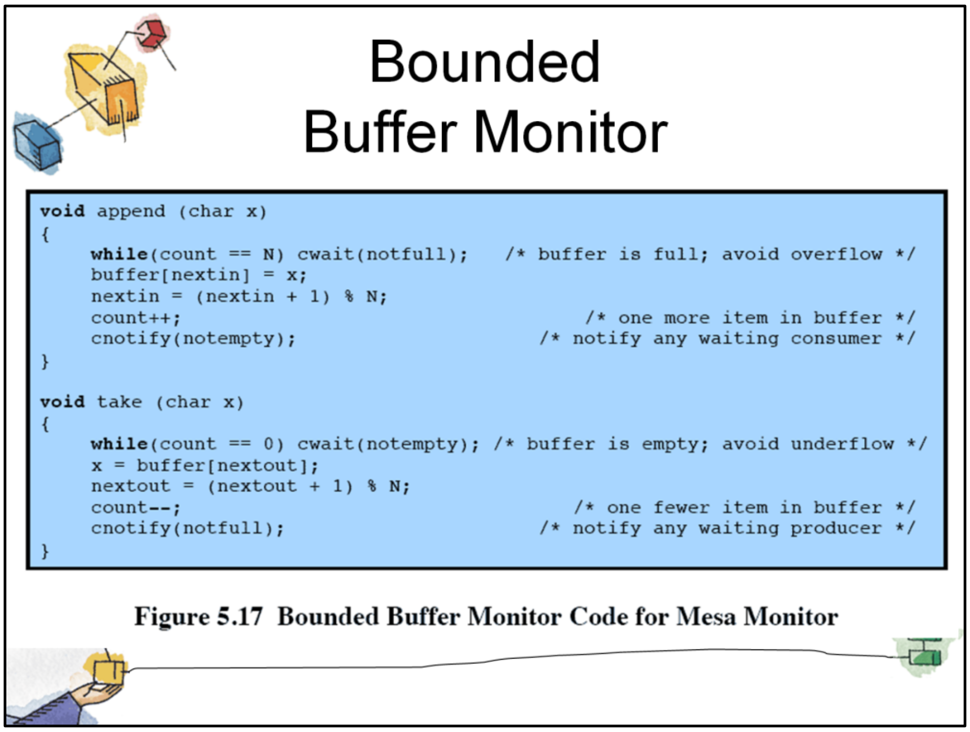


## Solution Using Monitor

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
        take(x);
        consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

Note here that a process exits the monitor immediately after executing the csignal function.

One language, ConcurrentPascal, insists that nothing follows a csignal call



## Bounded Buffer Monitor

```
void append (char x)
{
     while(count == N) cwait(notfull);   /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;                                     /* one more item in buffer */
     cnotify(notempty);                       /* notify any waiting consumer */
}

void take (char x)
{
     while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                    /* one fewer item in buffer */
     cnotify(notfull);                        /* notify any waiting producer */
}
```

**Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor**

Lampson and Redell developed an approach known as MESA

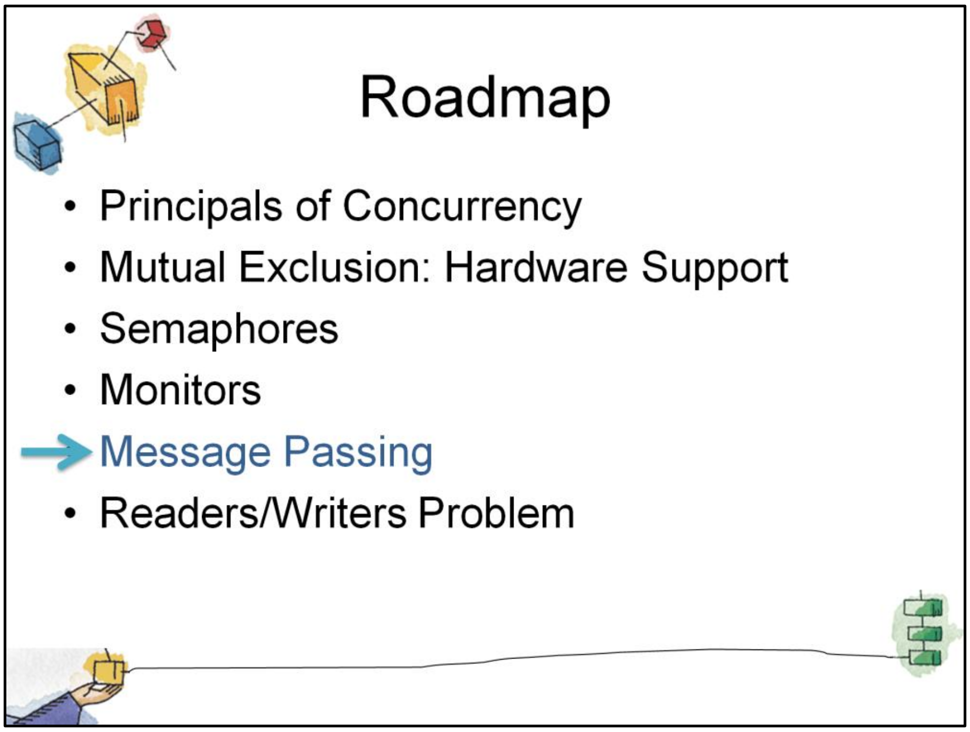
When a process executing in a monitor executes cnotify(x),

> it causes the x condition queue to be notified, but the signalling process continues to execute.

The result of the notification is that the process at the head of the condition queue will be resumed at some convenient future time when the monitor is available.

> However, because there is no guarantee that some other process will not enter the monitor before the waiting process, the waiting process must recheck the condition.

E.G. the procedures in the boundedbuffer monitor would now have the code in this slide

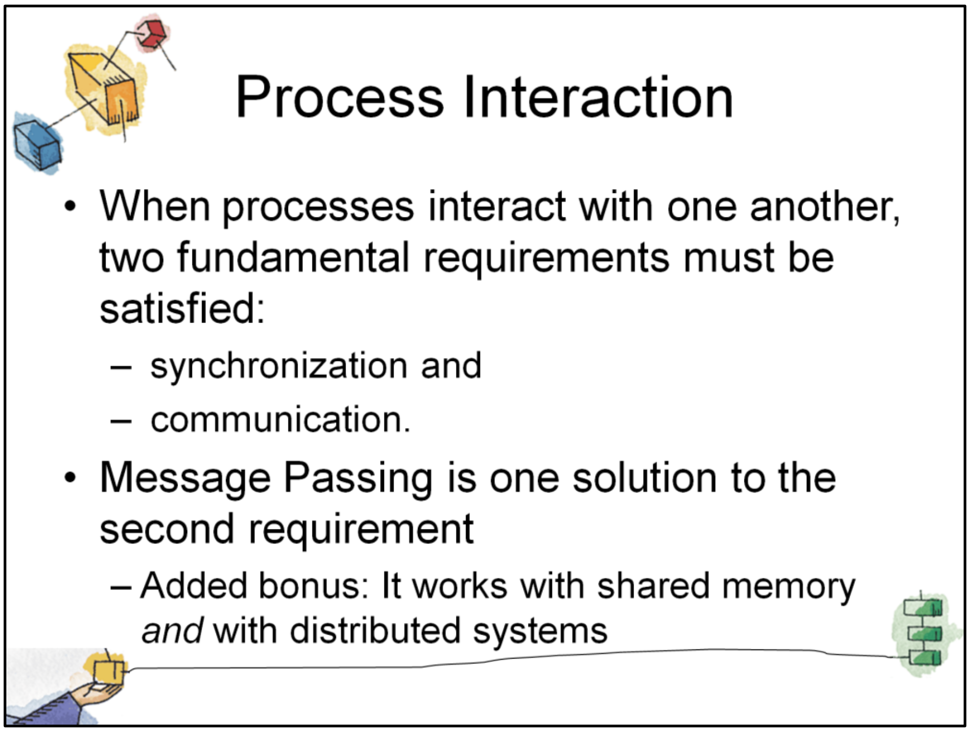


## Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors

→ Message Passing

- Readers/Writers Problem



## Process Interaction

- When processes interact with one another, two fundamental requirements must be satisfied:
  - synchronization and
  - communication.
- Message Passing is one solution to the second requirement
  - Added bonus: It works with shared memory *and* with distributed systems

When processes interact with one another, two fundamental requirements must be satisfied:

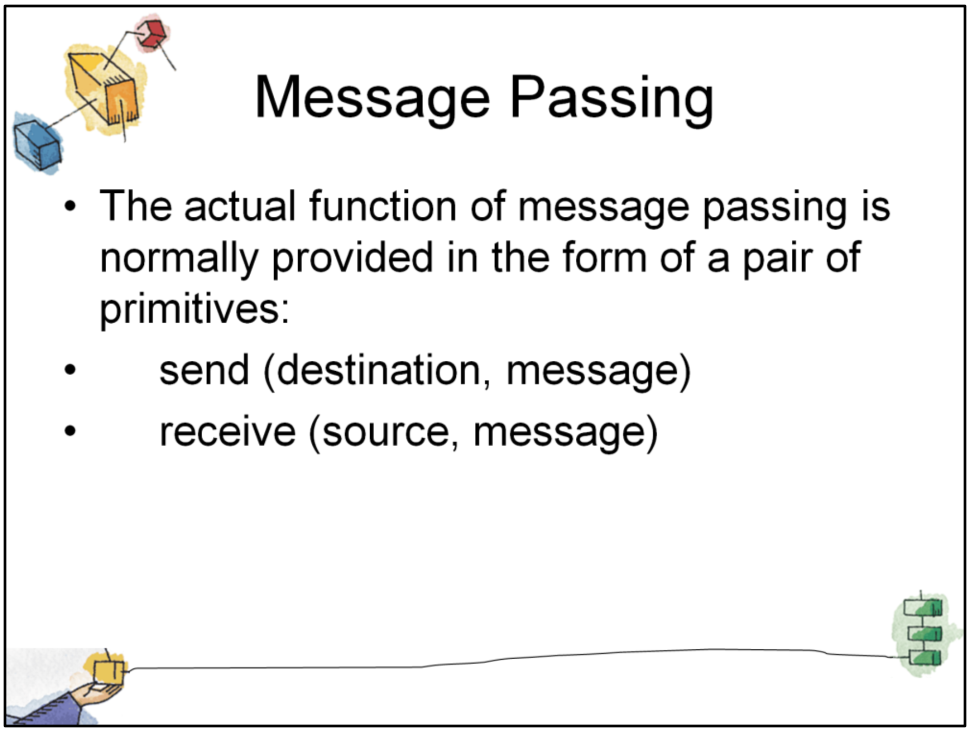
- synchronization and
- communication.

Processes need to be synchronized to enforce mutual exclusion;

cooperating processes may need to exchange information.

One approach to providing both of these functions is message passing.

Message passing has the further advantage that it lends itself to implementation in distributed systems as well as in shared-memory multiprocessor and uniprocessor systems.



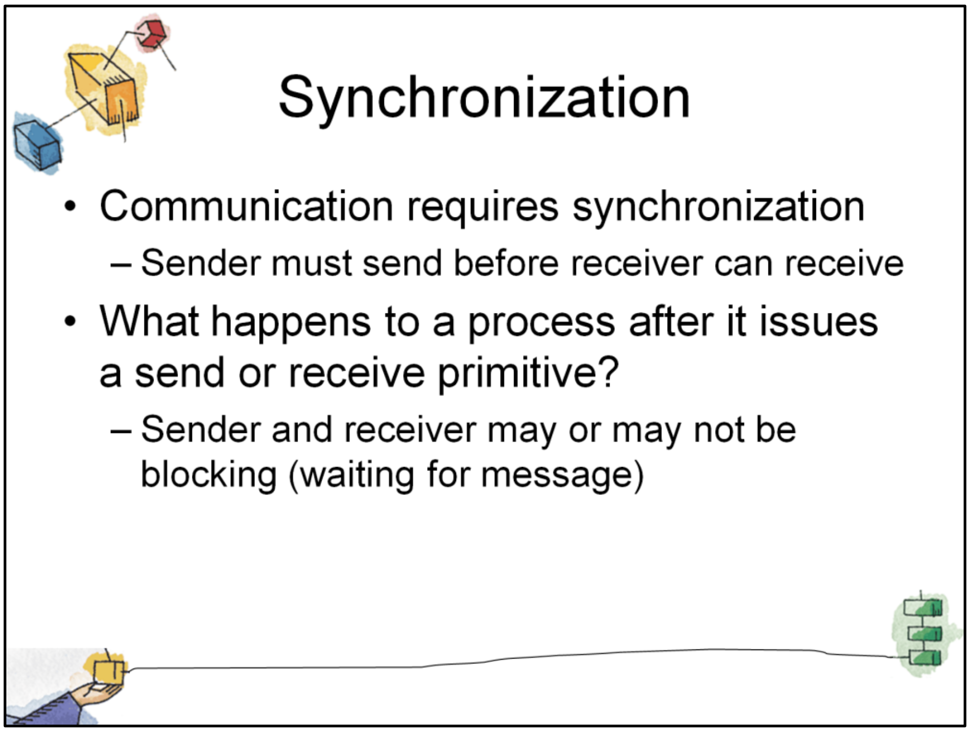
## Message Passing

- The actual function of message passing is normally provided in the form of a pair of primitives:
- send (destination, message)
- receive (source, message)

Emphasize that message-passing systems come in many forms. We provide a ***general*** introduction that discusses features typically found in such systems.

These primitives are a minimum set of operations needed for processes to engage in message passing.

- A process sends information in the form of a message to another process designated by a destination.
- A process receives information by executing the receive primitive, indicating the source and the message.



## Synchronization

- Communication requires synchronization
  - Sender must send before receiver can receive
- What happens to a process after it issues a send or receive primitive?
  - Sender and receiver may or may not be blocking (waiting for message)

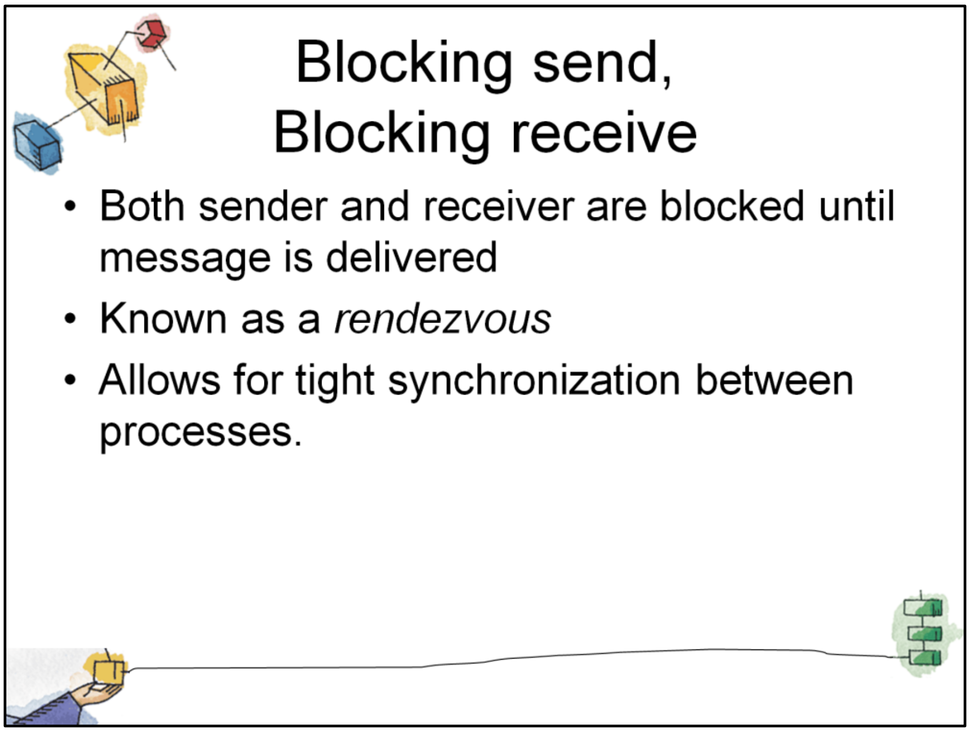
The communication of a message between two processes implies some level of synchronization between the two: the receiver cannot receive a message until it has been sent by another process.

When a send primitive is executed in a process, there are two possibilities:

- Either the sending process is blocked until the message is received,
- or it is not.

Similarly, when a process issues a receive primitive, there are two possibilities:

- If a message has previously been sent, the message is received and execution continues.
- If there is no waiting message, then either
  - (a) the process is blocked until a message arrives, or
  - (b) the process continues to execute, abandoning the attempt to receive.



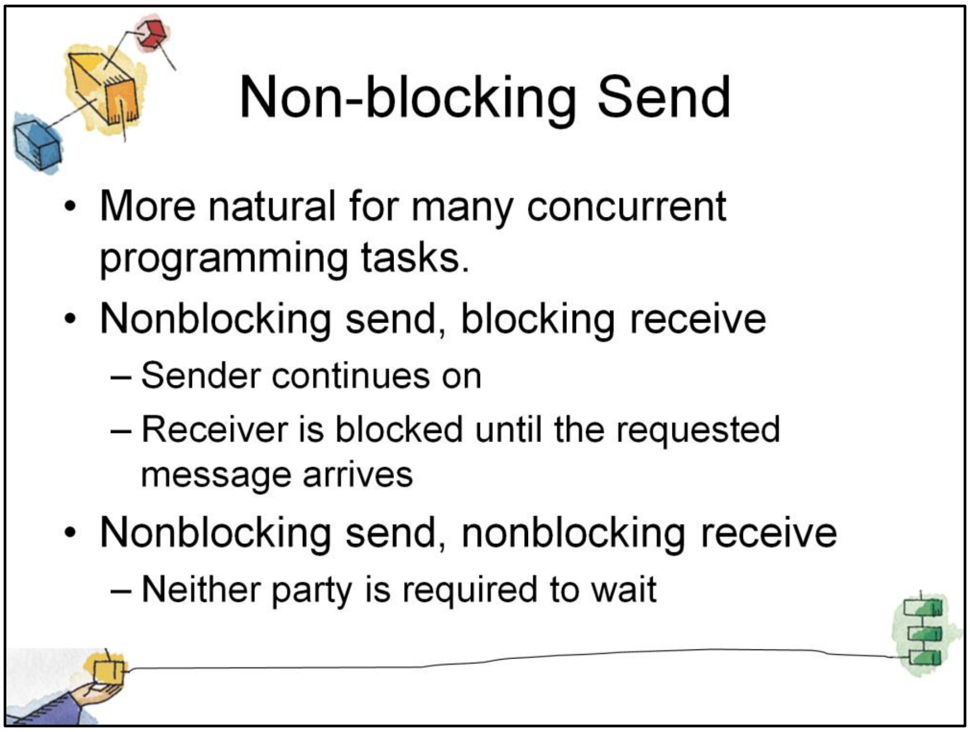
## Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- Known as a *rendezvous*
- Allows for tight synchronization between processes.

Both the sender and receiver are blocked until the message is delivered;

this is sometimes referred to as a *rendezvous.*

This combination allows for tight synchronization between processes.



## Non-blocking Send

- More natural for many concurrent programming tasks.
- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

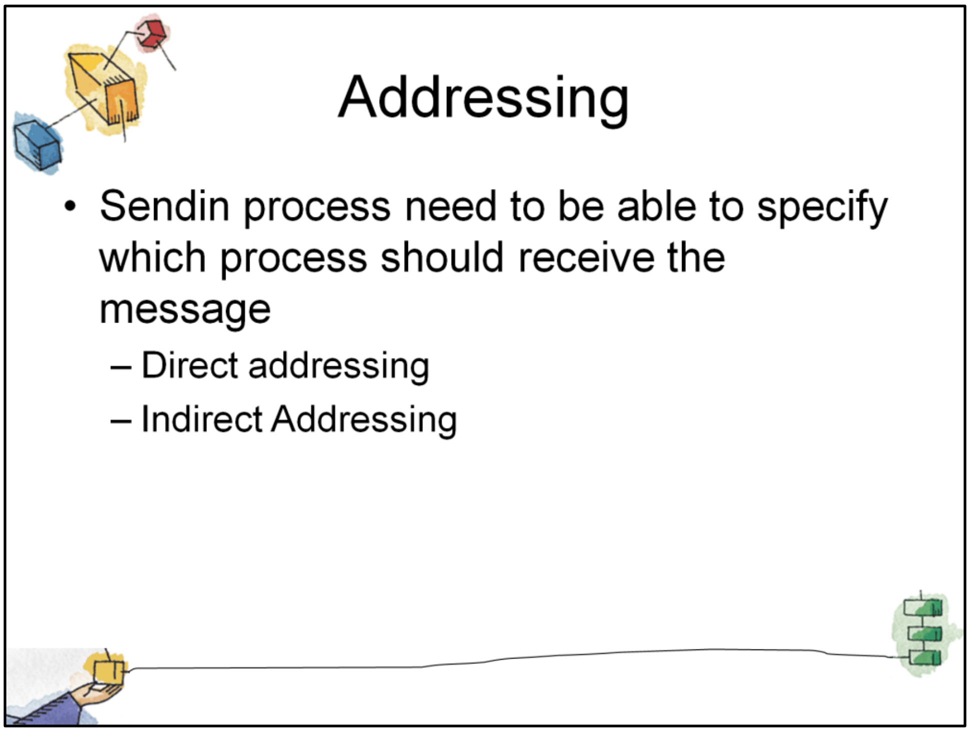
**Nonblocking send, blocking receive:**

Although the sender may continue on, the receiver is blocked until the requested message arrives.

This is probably the most useful combination.

- It allows a process to send one or more messages to a variety of destinations as quickly as possible.
- A process that must receive a message before it can do useful work needs to be blocked until such a message arrives.

- **Nonblocking send, nonblocking receive:**
  - Neither party is required to wait.

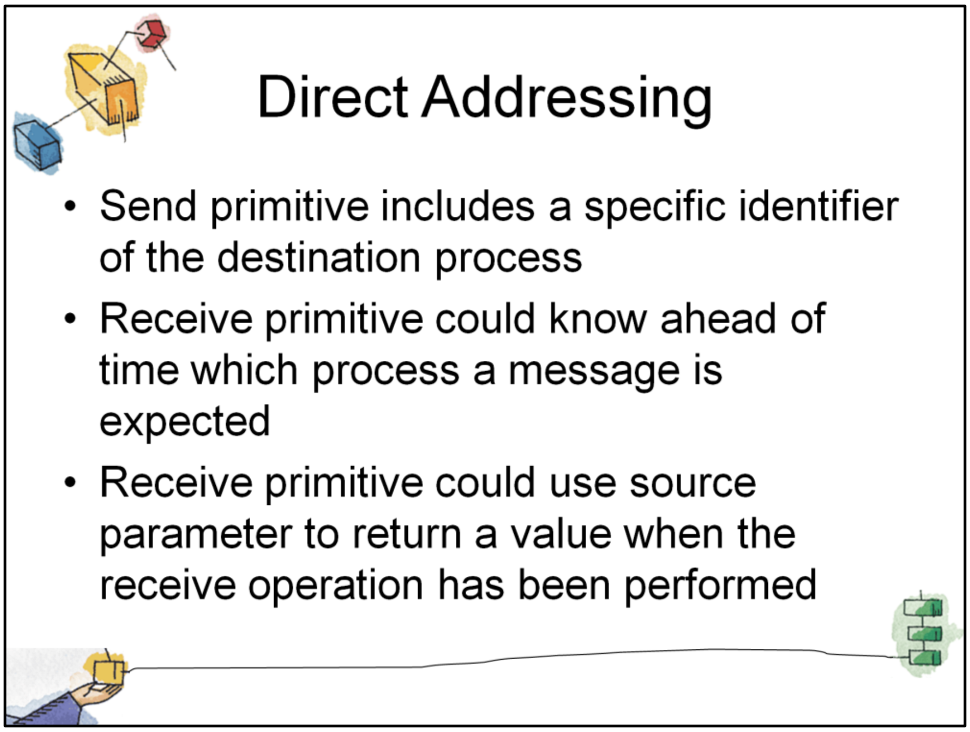


## Addressing

- Sendin process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Addressing

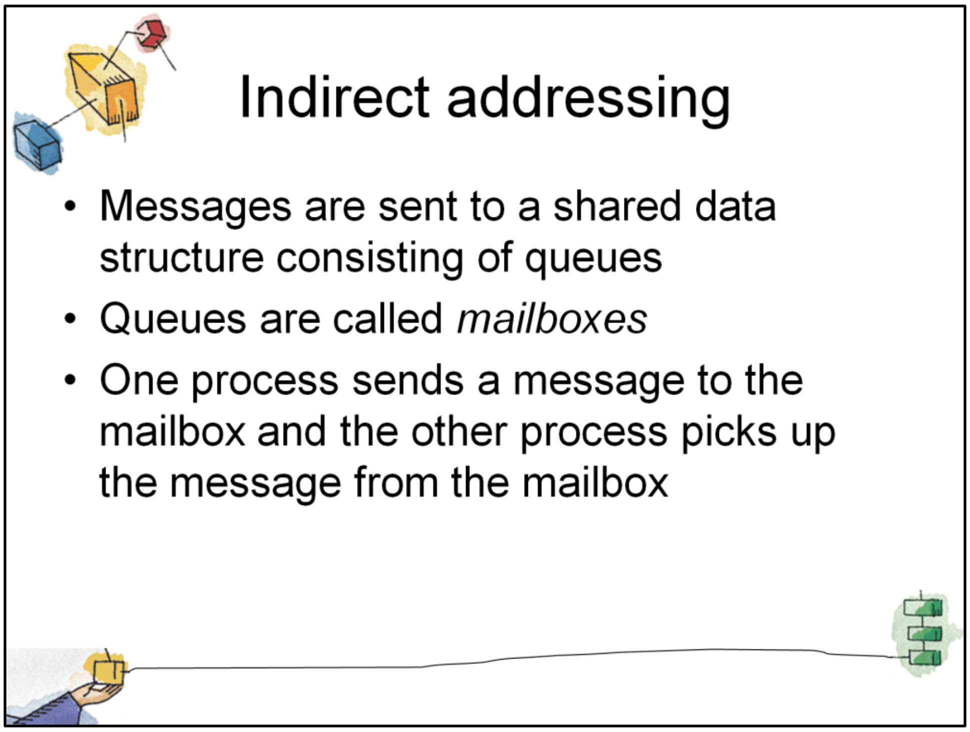
it is necessary to have a way of specifying in the send primitive which process is to receive the message.

Similarly, most implementations allow a receiving process to indicate the source of a message to be received.



## Direct Addressing

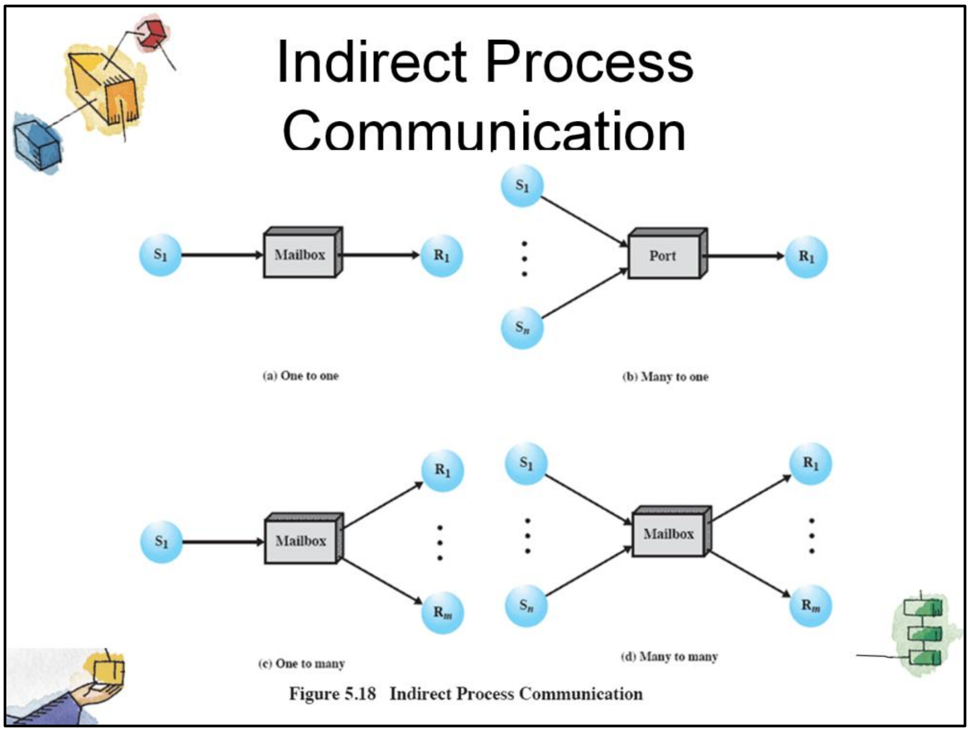
- Send primitive includes a specific identifier of the destination process
- Receive primitive could know ahead of time which process a message is expected
- Receive primitive could use source parameter to return a value when the receive operation has been performed



## Indirect addressing

- Messages are sent to a shared data structure consisting of queues
- Queues are called *mailboxes*
- One process sends a message to the mailbox and the other process picks up the message from the mailbox

A strength of the use of indirect addressing is that, by decoupling the sender and receiver, it allows for greater flexibility in the use of messages.



## Indirect Process Communication

Figure 5.18 Indirect Process Communication

**Animated Slide** – each item below is magnified for instructor to address separately

**1) A one-to-one relationship**

- allows a private communications link to be set up between two processes.
- This insulates their interaction from erroneous interference from other processes.

**2) A many-to-one relationship is useful for client/server interaction;**

- one process provides service to a number of other processes.
- In this case, the mailbox is often referred to as a *port.*

**3) A one-to-many relationship allows for one sender and multiple receivers;**

- it is useful for applications where a message or some information is to be broadcast to a set of processes.

**4) A many-to-many relationship**

- allows multiple server processes to provide concurrent service to multiple clients.

The association of processes to mailboxes can be either static or dynamic.

Ports are often statically associated with a particular process; that is, the port is created and assigned to the process permanently.

Similarly, a one-to-one relationship is typically defined statically and permanently. When there are many senders, the association of a sender to a mailbox may occur dynamically. Primitives such as connect and disconnect may be used for this purpose

## General Message Format

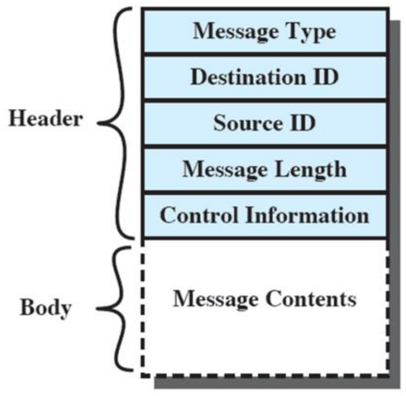


Figure 5.19 General Message Format

The format of the message depends on the objectives of the messaging facility and whether the facility runs on a single computer or on a distributed system.

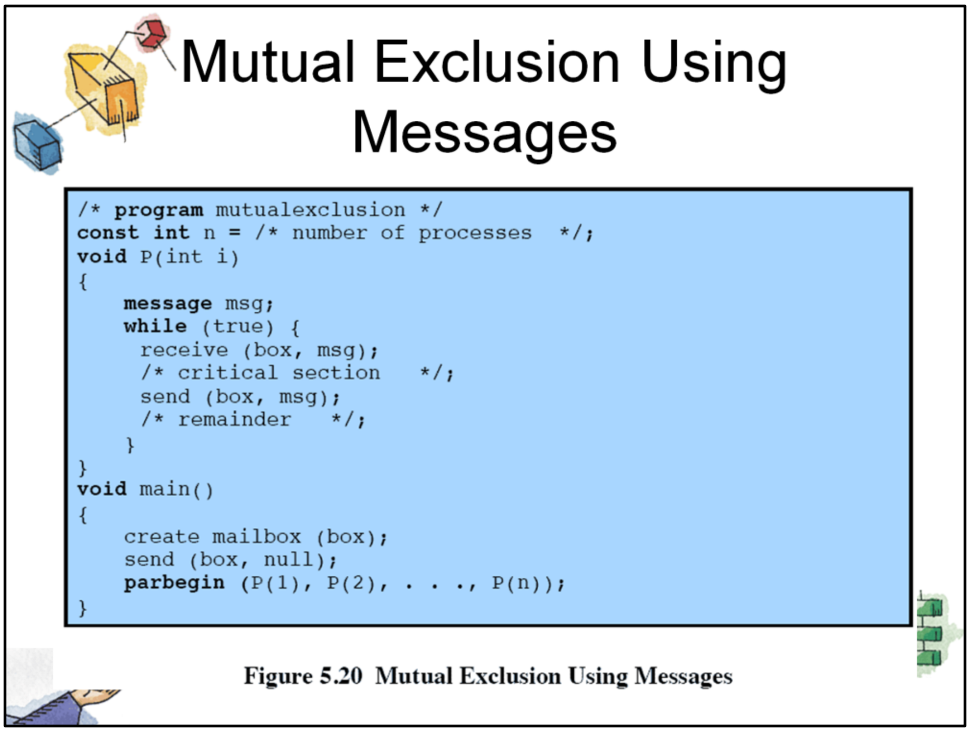
This is a typical message format for operating systems that support variable-length messages.

The message is divided into two parts:

**a header**, which contains information about the message.

- The header may contain an identification of the source and intended destination of the message, a length field, and a type field to discriminate among various types of messages.
- additional control information, e.g. pointer field so a linked list of messages can be created; a sequence number, to keep track of the number and order of messages passed between source and destination; and a priority field.

**a body**, which contains the actual contents of the message.



## Mutual Exclusion Using Messages

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

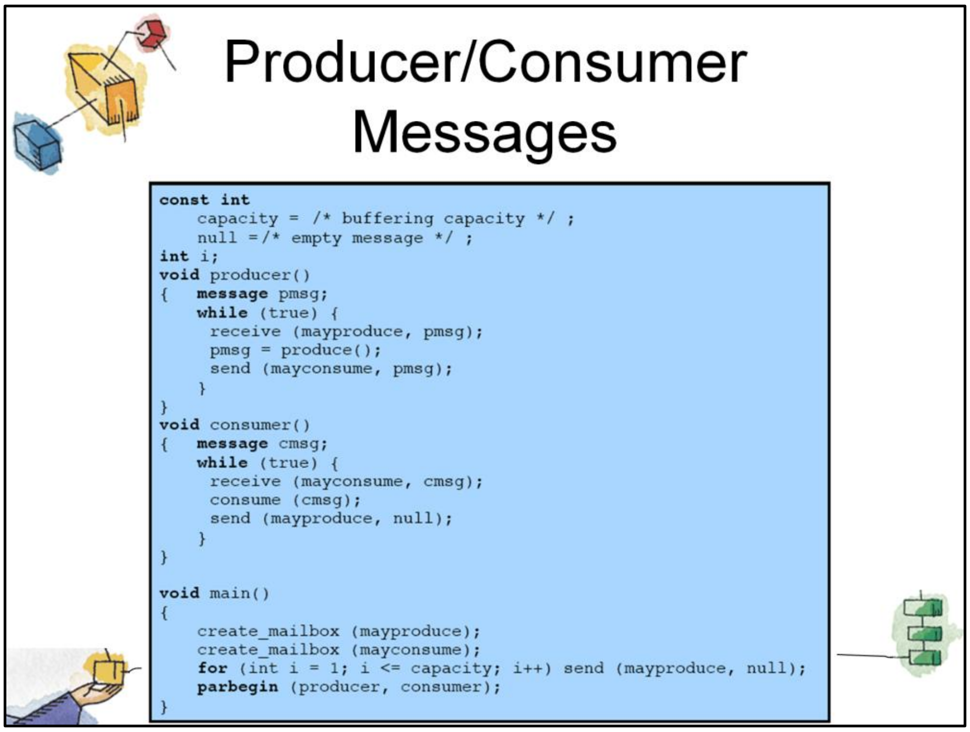
Figure 5.20 Mutual Exclusion Using Messages

This is one way in which message passing can be used to enforce mutual exclusion.

We assume the use of the blocking receive primitive and the non-blocking send primitive.

This assumes that if more than one process performs the receive operation concurrently, then

- If there is a message, it is delivered to only one process and the others are blocked, or
- If the message queue is empty, all processes are blocked; when a message is available, only one blocked process is activated and given the message.



## Producer/Consumer Messages

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
      receive (mayproduce, pmsg);
      pmsg = produce();
      send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
      receive (mayconsume, cmsg);
      consume (cmsg);
      send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce, null);
    parbegin (producer, consumer);
}
```

This is an example of the use of message passing to the bounded-buffer producer/consumer problem.

This program takes advantage of the ability of message passing to be used to pass data in addition to signals.

Two mailboxes are used.

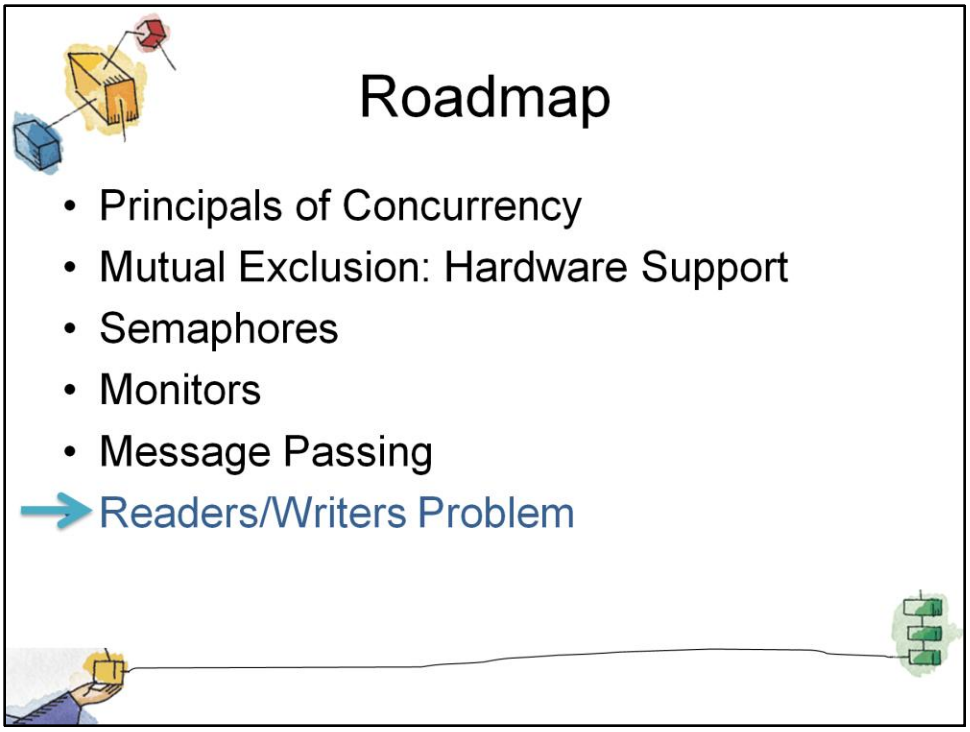
As the producer generates data, it is sent as messages to the mailbox mayconsume.

As long as there is at least one message in that mailbox, the consumer can consume.

Hence mayconsume serves as the buffer; the data in the buffer are organized as a queue of messages.

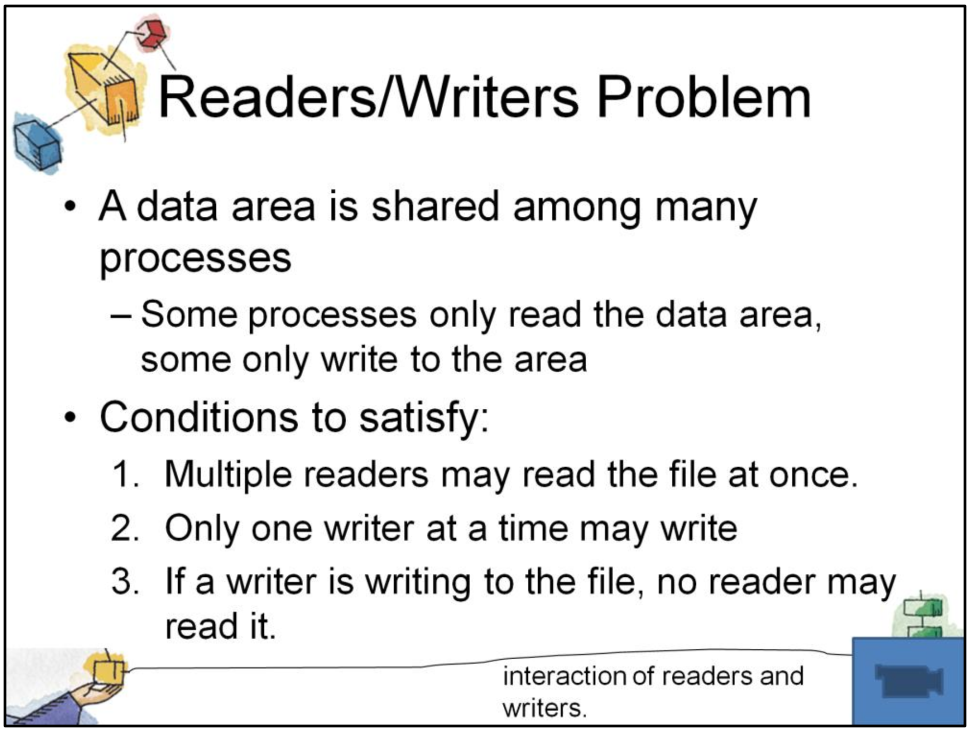
The “size” of the buffer is determined by the global variable capacity. Initially, the mailbox mayproduce is filled with a

number of null messages equal to the capacity of the buffer. The number of messages in mayproduce shrinks with each production and grows with each consumption.



## Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- → Readers/Writers Problem



## Readers/Writers Problem

- A data area is shared among many processes
  - Some processes only read the data area, some only write to the area
- Conditions to satisfy:
  1. Multiple readers may read the file at once.
  2. Only one writer at a time may write
  3. If a writer is writing to the file, no reader may read it.

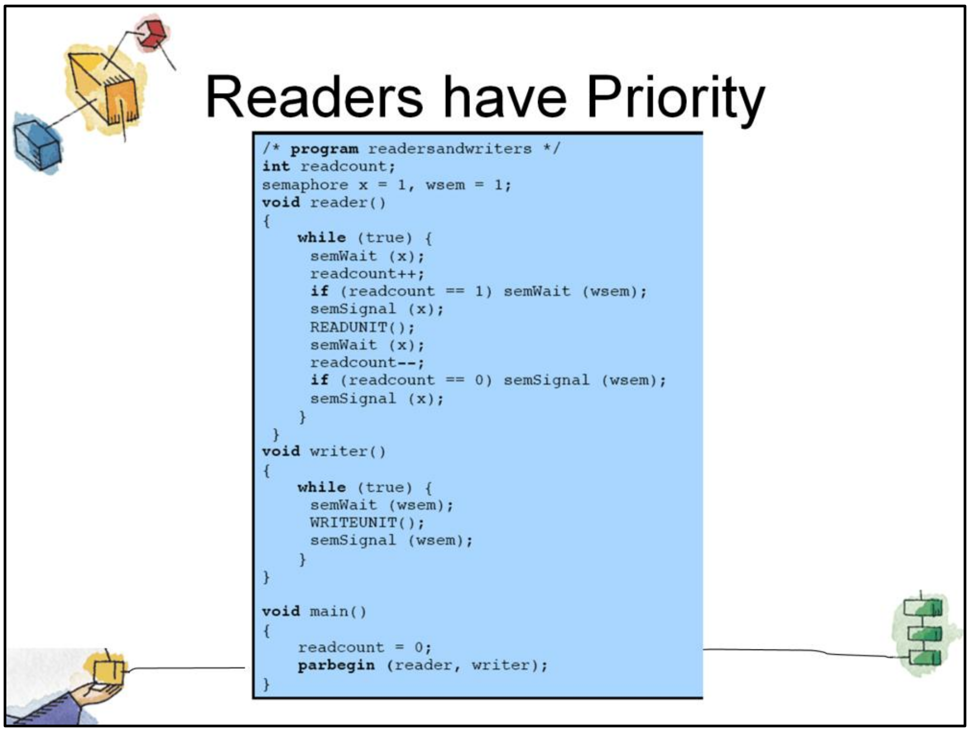
interaction of readers and writers.

The readers/writers problem is:

- There is a data area shared among a number of processes.
  - The data area could be a file, a block of main memory,or even a bank of processor registers.
- There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.



## Readers have Priority

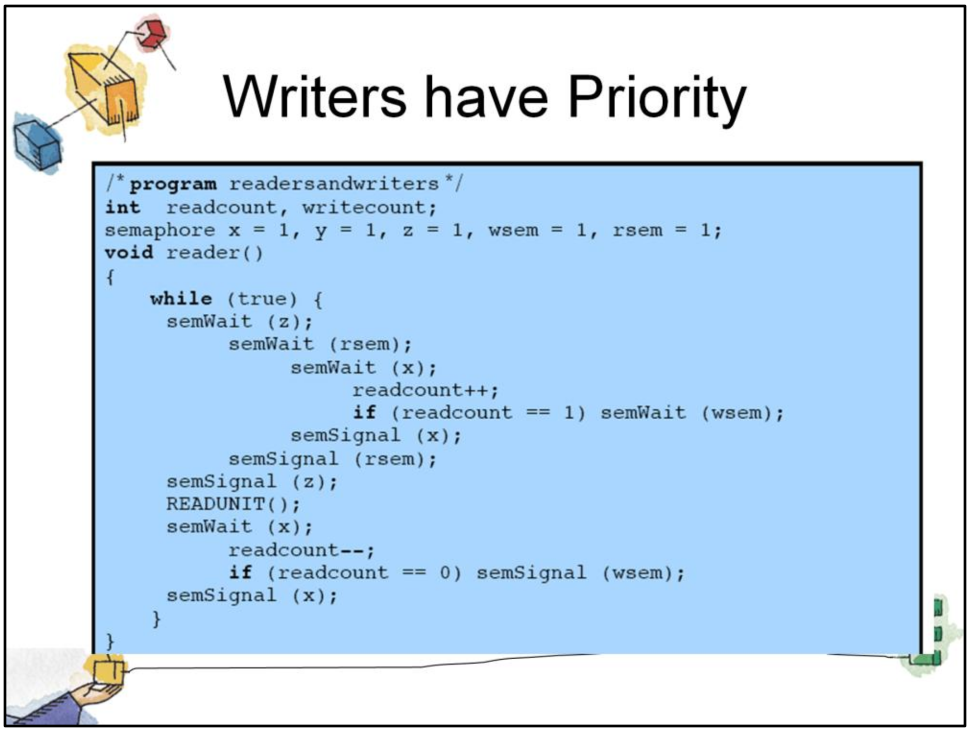
```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

This solution uses semaphores, showing one instance each of a reader and a writer; the solution does not change for multiple readers and writers.

Once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading.

Therefore, writers are subject to starvation.



## Writers have Priority

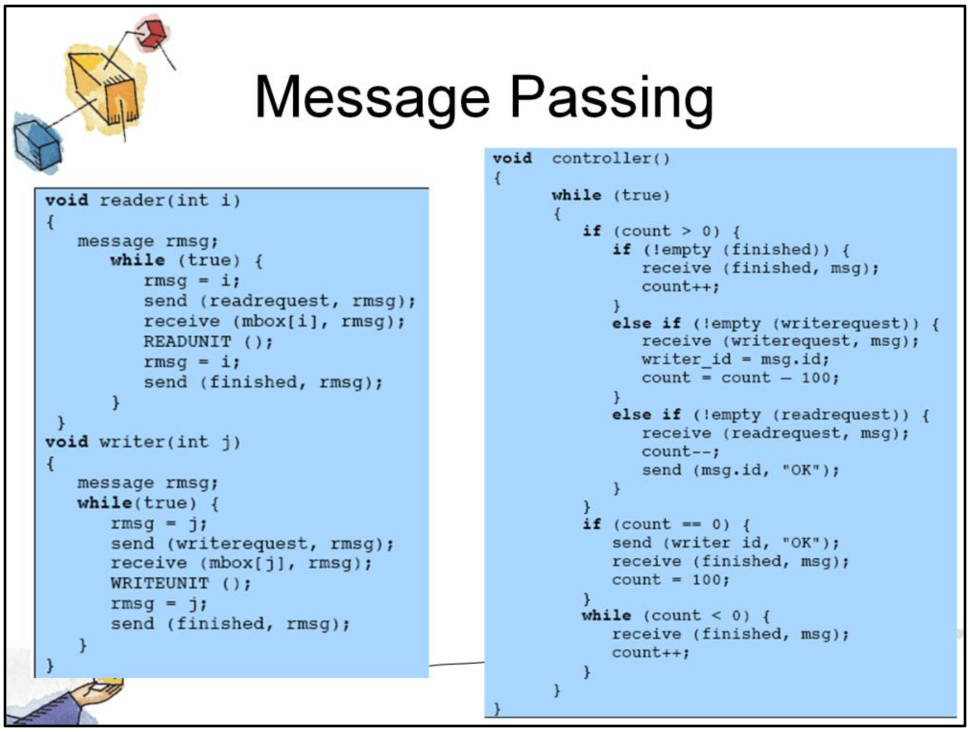
```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
```

This solution guarantees that no new readers are allowed access to the data area once at least one writer has declared a desire to write.

Continued on next slide

## Writers have Priority

```
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```



## Message Passing

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count – 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

This shows an alternative solution, which gives writers priority and which is implemented using message passing.

## Message Passing

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count – 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```