Operating Systems: Internals and Design Principles

Chapter 5 Concurrency: Mutual Exclusion and Synchronization

Seventh Edition
By William Stallings

Multiple Processes

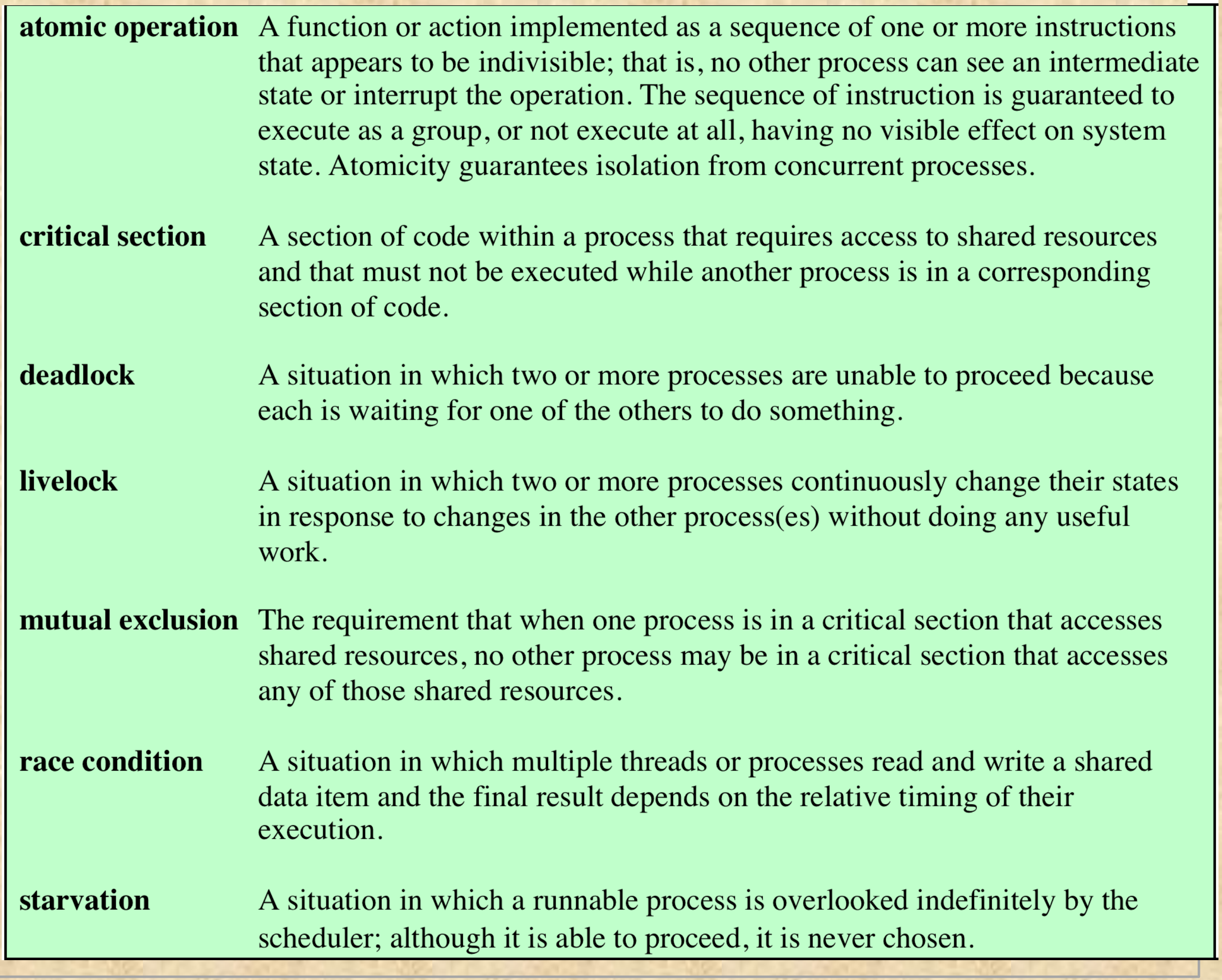
- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing

Concurrency & Shared Data

- Concurrent processes may share data to support communication, info exchange,...
- Threads in the same process can share global address space
- Concurrent sharing may cause problems
- For example: lost updates

Concurrency

Key Terms



atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
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.

Table 5.1 Some Key Terms Related to Concurrency

Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

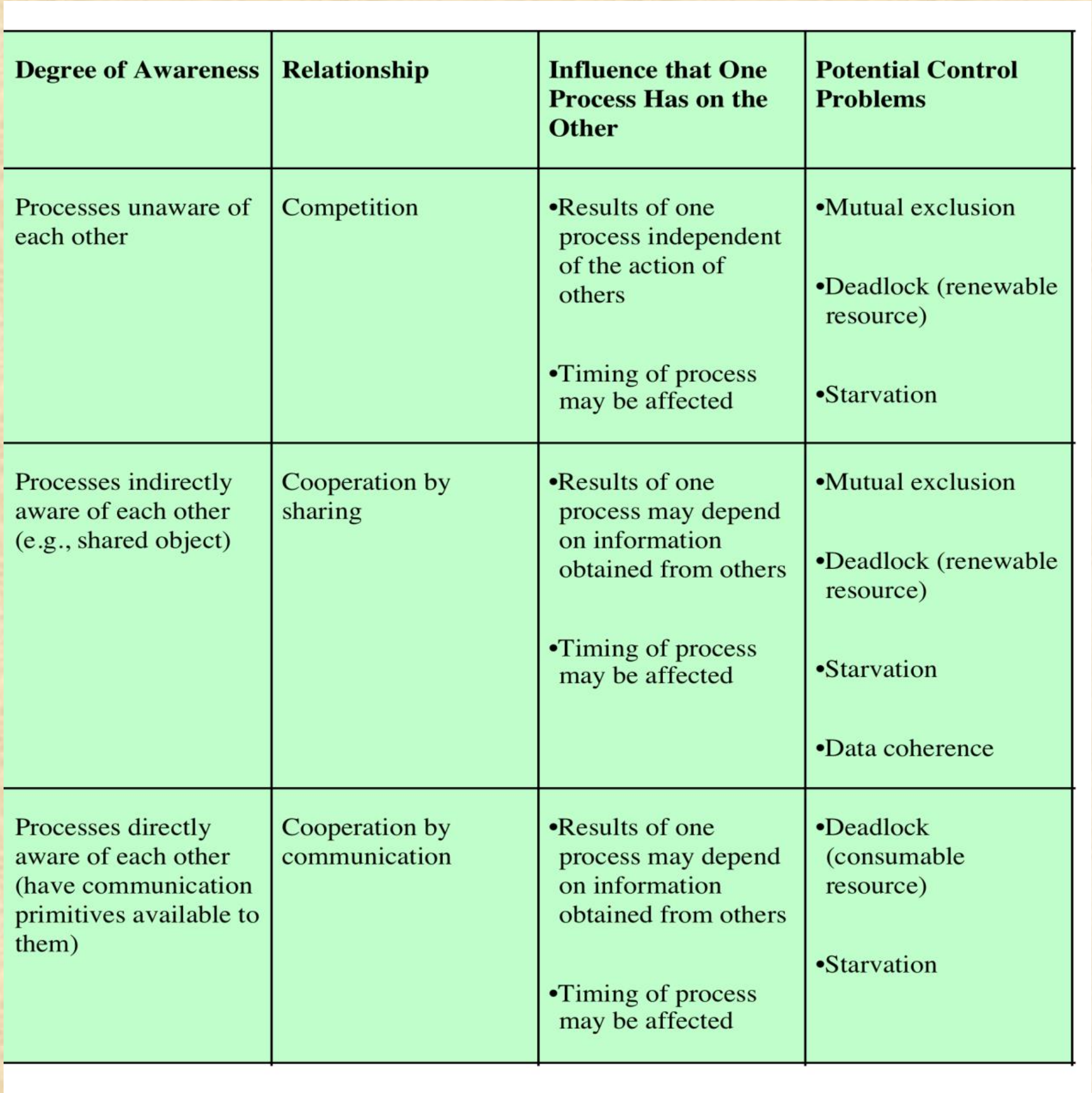
Race Condition

- Occurs when multiple processes or threads read and write shared data items
- The final result depends on the order of execution
  - the "loser" of the race is the process that updates last and will determine the final value of the variable

Operating System Concerns

- Design and management issues raised by the existence of concurrency:
  - The OS must:
    - be able to keep track of various processes
    - allocate and de-allocate resources for each active process
    - protect the data and physical resources of each process against interference by other processes
    - ensure that the processes and outputs are independent of the processing speed

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 •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •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 •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •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 •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Resource Competition

- Concurrent processes come into conflict when they use the same resource (competitively or shared)
  - for example: I/O devices, memory, processor time, clock
- Three control problems must be faced
  - Need for mutual exclusion
  - Deadlock
  - Starvation
- Sharing processes also need to address coherence

Need for Mutual Exclusion

- If there is no controlled access to shared data, processes or threads may get an inconsistent view of this data
- The result of concurrent execution will depend on the order in which instructions are interleaved.
- Errors are timing dependent and usually not reproducible.

A Simple Example

- Assume P1 and P2 are executing this code and share the variable **`a`**
- Processes can be preempted at any time.
- Assume P1 is preempted after the input statement, and P2 then executes entirely
- The character echoed by P1 will be the one read by P2 !!

```
static char a;

void echo()
{
   cin >> a;
   cout << a;
}
```

What’s the Problem?

- This is an example of a *race condition*
- Individual processes (threads) execute sequentially in isolation, but concurrency causes them to interact.
- We need to prevent concurrent execution by processes when they are changing the same data. We need to enforce *mutual exclusion*.

The Critical Section Problem

- When a process executes code that manipulates shared data (or resources), we say that the process is in its critical section (CS) for that shared data
- We must enforce mutual exclusion on the execution of critical sections.
- Only one process at a time can be in its CS (for that shared data or resource).

The Critical Section Problem

- Enforcing mutual exclusion guarantees that related CS's will be executed *serially* instead of *concurrently*.
- The critical section problem is how to provide mechanisms to enforce mutual exclusion so the actions of concurrent processes won't depend on the order in which their instructions are interleaved

The Critical Section Problem

- Processes/threads must request permission to enter a CS, & signal when they leave CS.
- Program structure:
  - entry section: requests entry to CS
  - exit section: notifies that CS is completed
  - remainder section (RS): code that does not involve shared data and resources.
- The CS problem exists on multiprocessors as well as on uniprocessors.

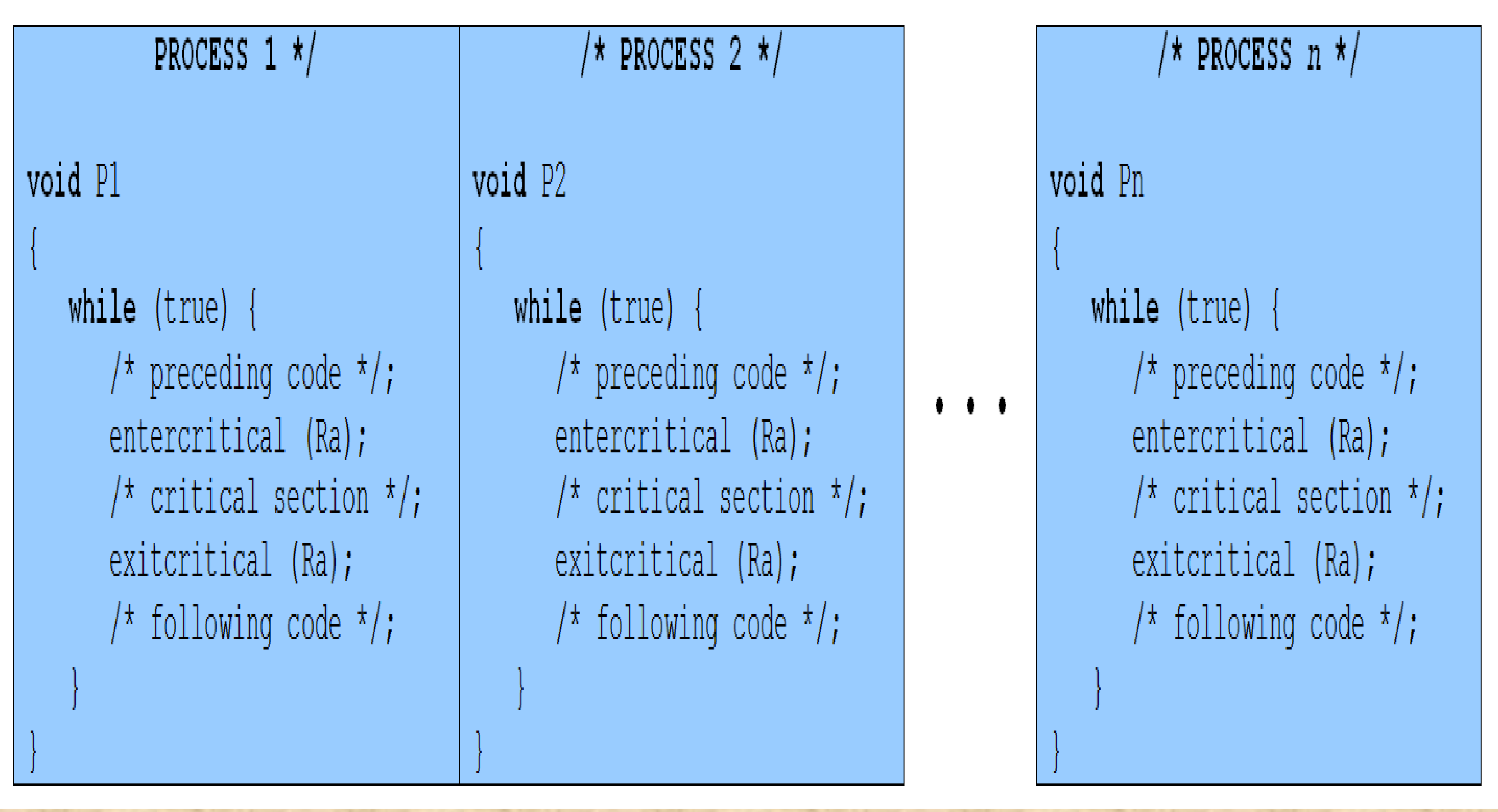
Mutual Exclusion and Data Coherence

- Mutual Exclusion ensures data coherence if properly used.
- Critical Resource (CR) - a shared resource such as a variable, file, or device
- Data Coherence:
  - The final value or state of a CR shared by concurrently executing processes is the same as the final value or state would be if each process executed serially, in some order.

Deadlock and Starvation

- Deadlock: two or more processes are blocked permanently because each is waiting for a resource held in a mutually exclusive manner by one of the others.
- Starvation: a process is repeatedly denied access to some resource which is protected by mutual exclusion, even though the resource periodically becomes available.

Mutual Exclusion



```
PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
/* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

• • •

```
/* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

Figure 5.1 Illustration of Mutual Exclusion

Requirements for Mutual Exclusion

- Mutual Exclusion: must be enforced
- Non interference: A process that halts must not interfere with other processes

- No deadlock or starvation
- Progress:A process must not be denied 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

Mutual Exclusion: Hardware Support

- **Interrupt Disabling**
  - uniprocessor system
  - disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will not work in a multiprocessor architecture

Mutual Exclusion: Hardware Support

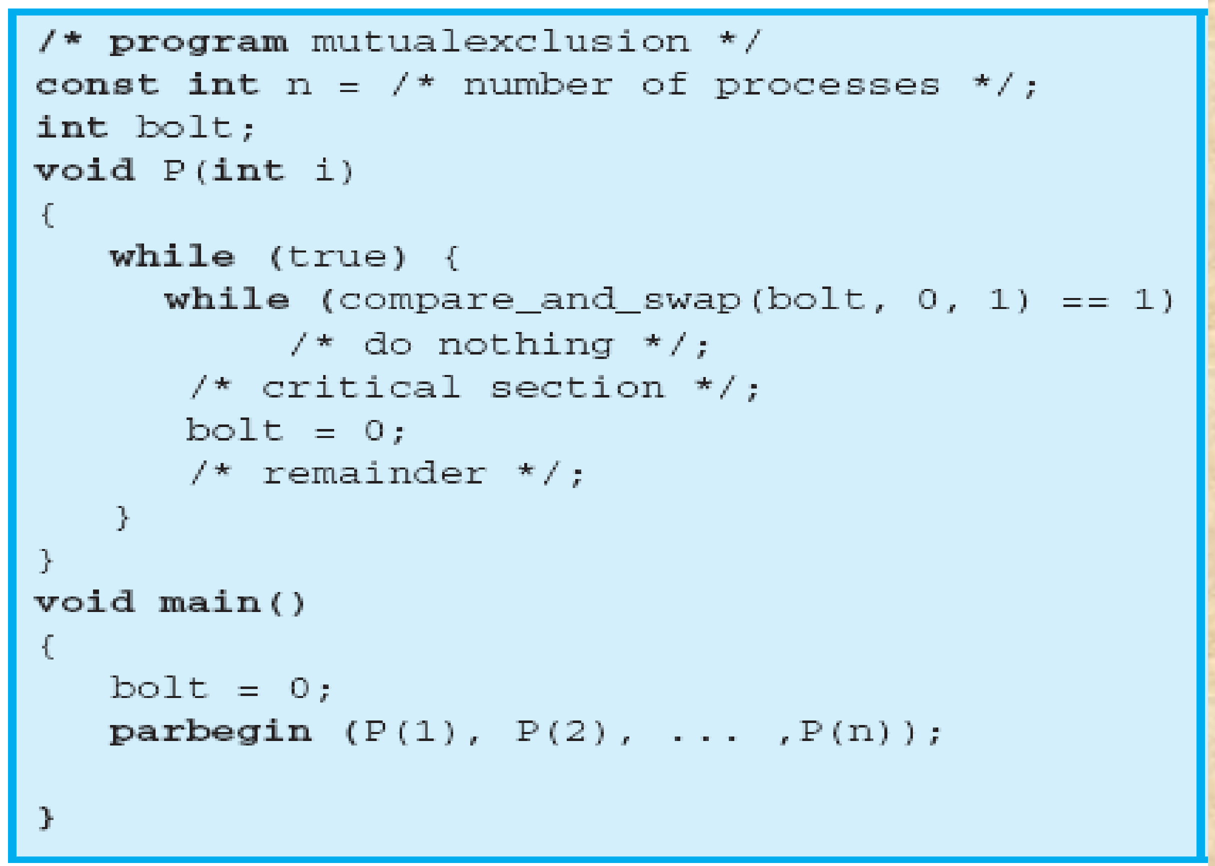
- Special Machine Instructions
  - Compare&Swap Instruction
    - also called a "compare and exchange instruction"
    - a **compare** is made between a memory value and a test value
    - if the old memory value = test value, swap in a new value to the memory location
    - always return the old memory value
    - carried out atomically in the hardware.

Mutual Exclusion: Hardware Support

- Compare&Swap Instruction
  - Pseudo-code definition of the hardware instruction:

```
compare_and_swap (word, test_val, new_val)
if (word ==test_val)
    word = new_val;
return new_val
```

Compare and Swap Instruction



```
/* 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

word = bolt
test_val = 0
new_val = 1

If bolt is 0 when the C&S is executed, the condition is false and P enters its critical section. (leaves bolt = 1)
If bolt = 1 when C&S executes, P continues to execute the while loop. It's busy waiting (or spinning)

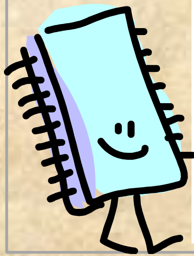


Figure 5.2 Hardware Support for Mutual Exclusion

Special Machine Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

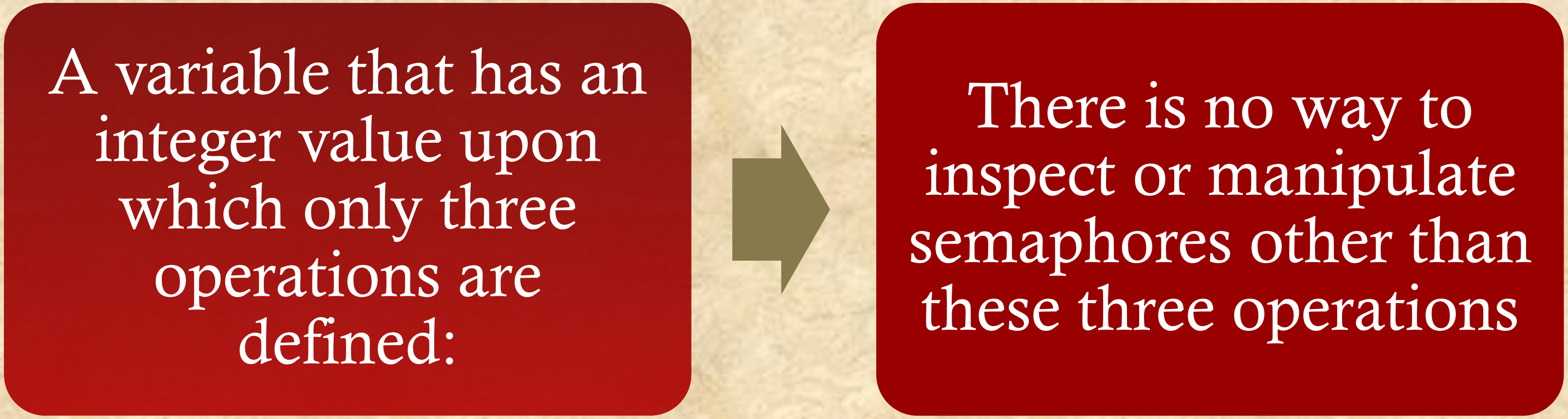
Special Machine Instruction: Disadvantages

- Busy-waiting is employed, thus 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
- Deadlock is possible if priority-based scheduling is used

Common Concurrency Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. 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).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Semaphore



A variable that has an integer value upon which only three operations are defined:

There is no way to inspect or manipulate semaphores other than these three operations

1) May be initialized to a nonnegative integer value
2) The semWait operation decrements the value
3) The semSignal operation increments the value

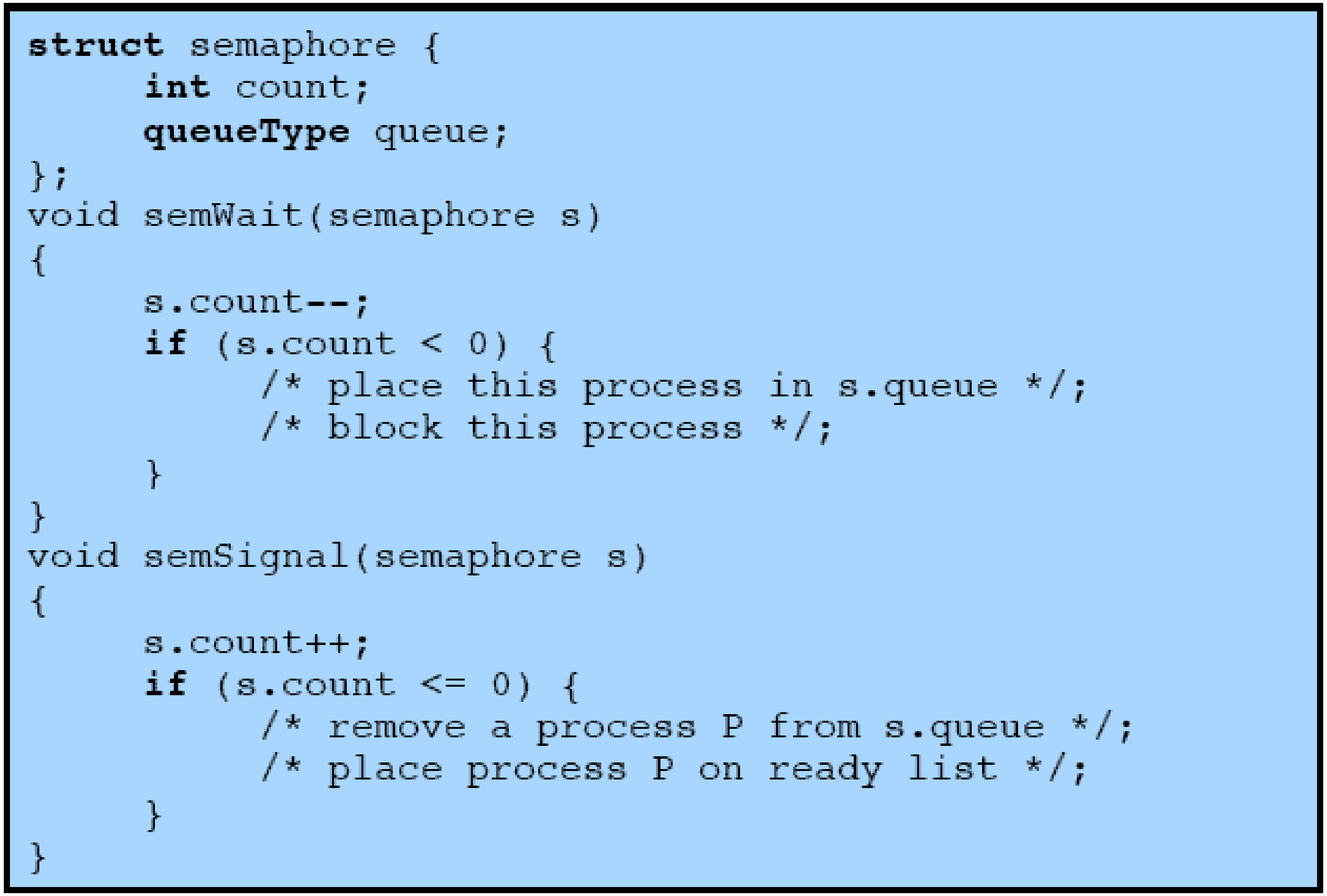
Consequences

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

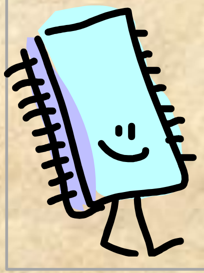
You don't know whether another process is waiting so the number of unblocked processes may be zero or one

Semaphore Primitives

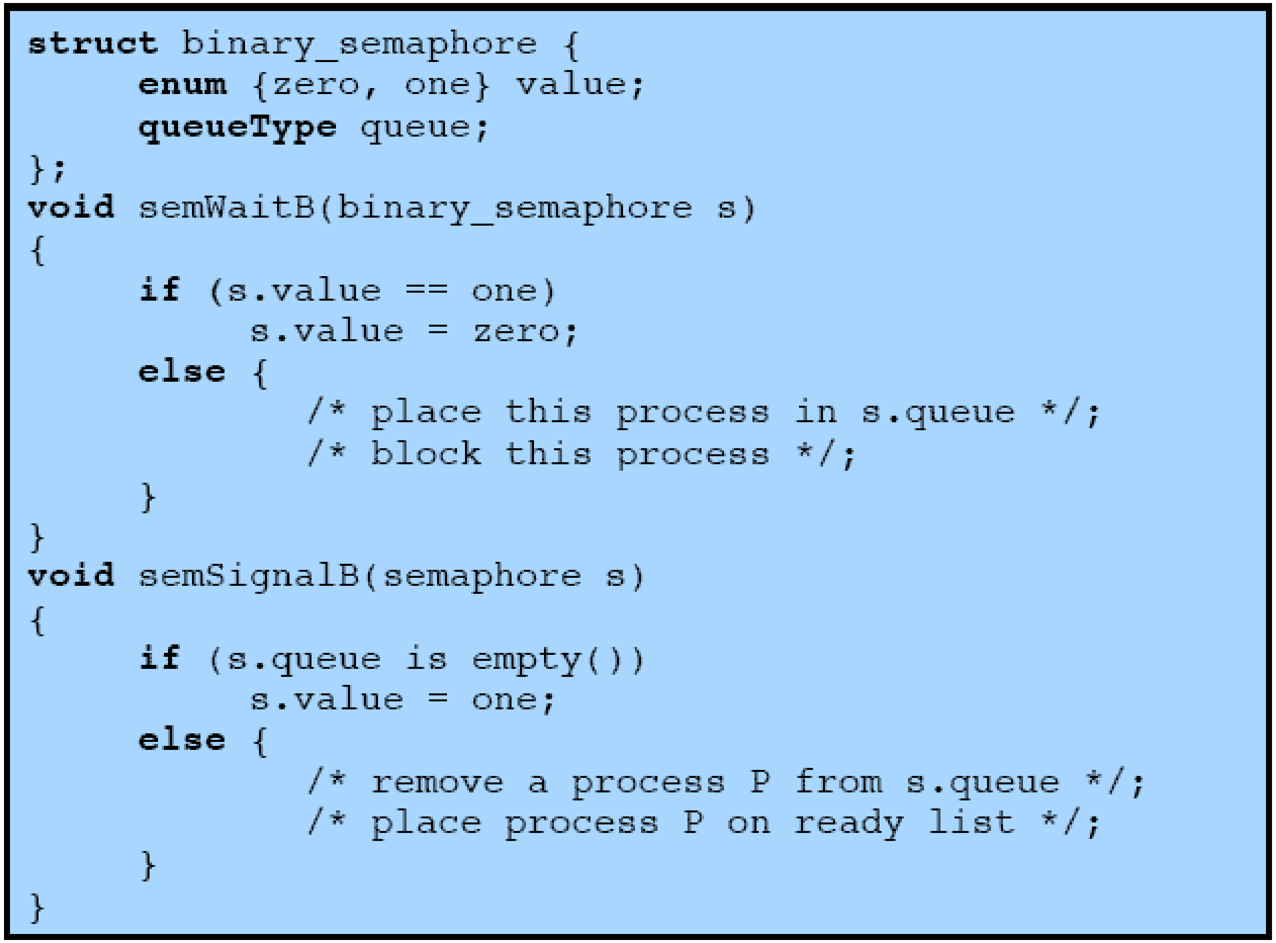


```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
     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

Figure 5.3 A Definition of Semaphore Primitives

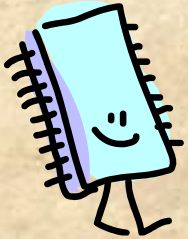


Binary Semaphore Primitives



```
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



Strong/Weak Semaphores

- A queue is used to hold processes waiting on the semaphore

Strong Semaphores

- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- the order in which processes are removed from the queue is not specified

Example of Semaphore Mechanism

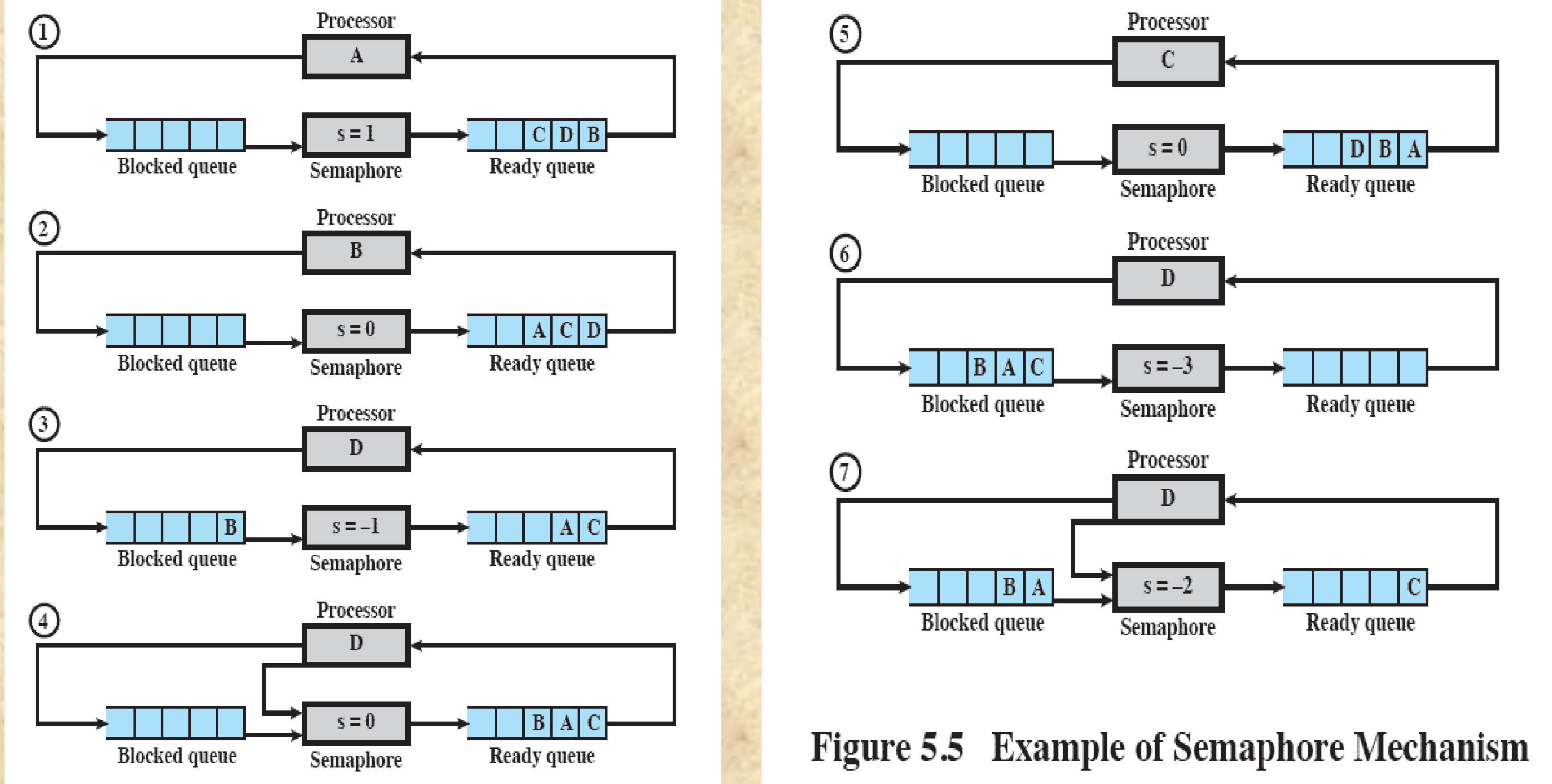
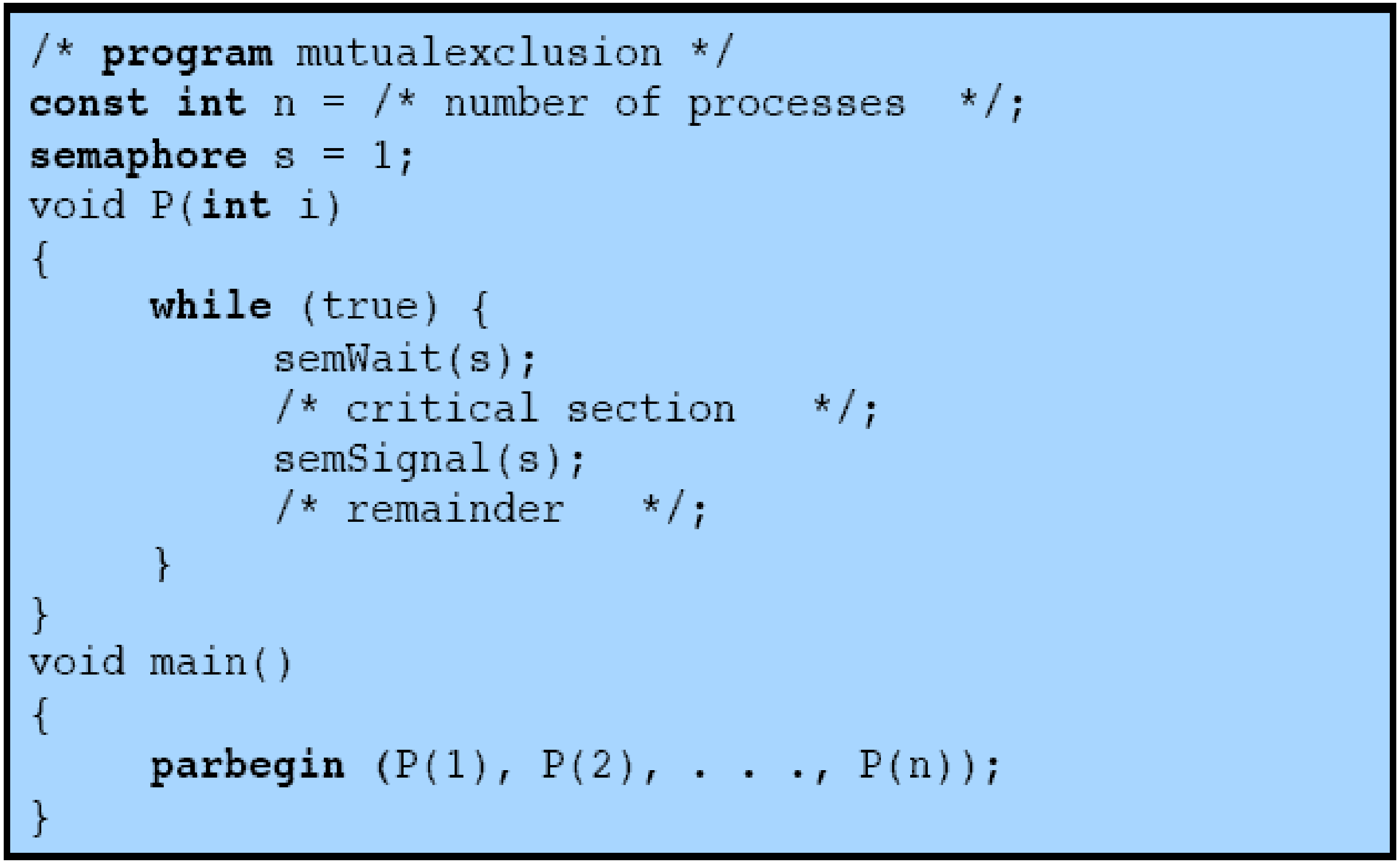


Figure 5.5 Example of Semaphore Mechanism

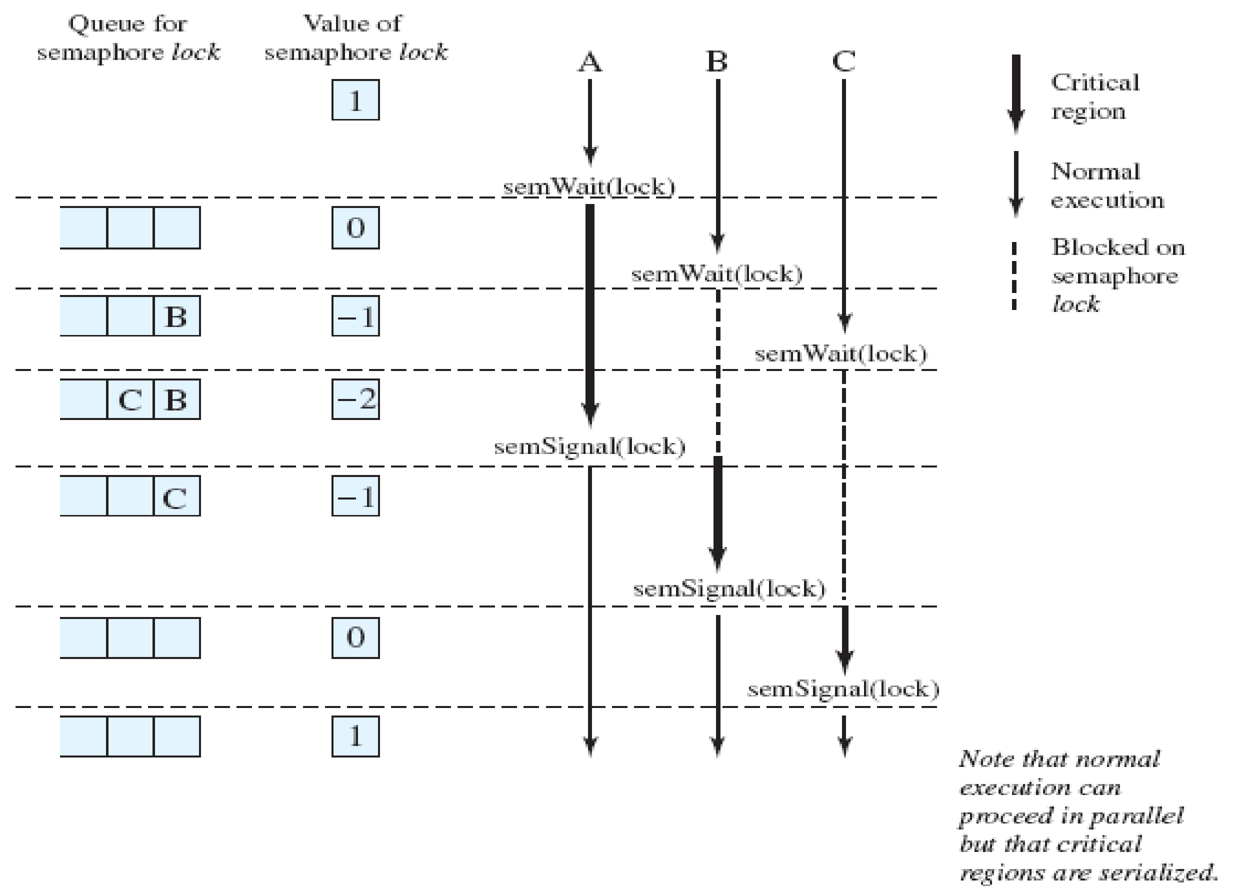
Mutual Exclusion



```
/* 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

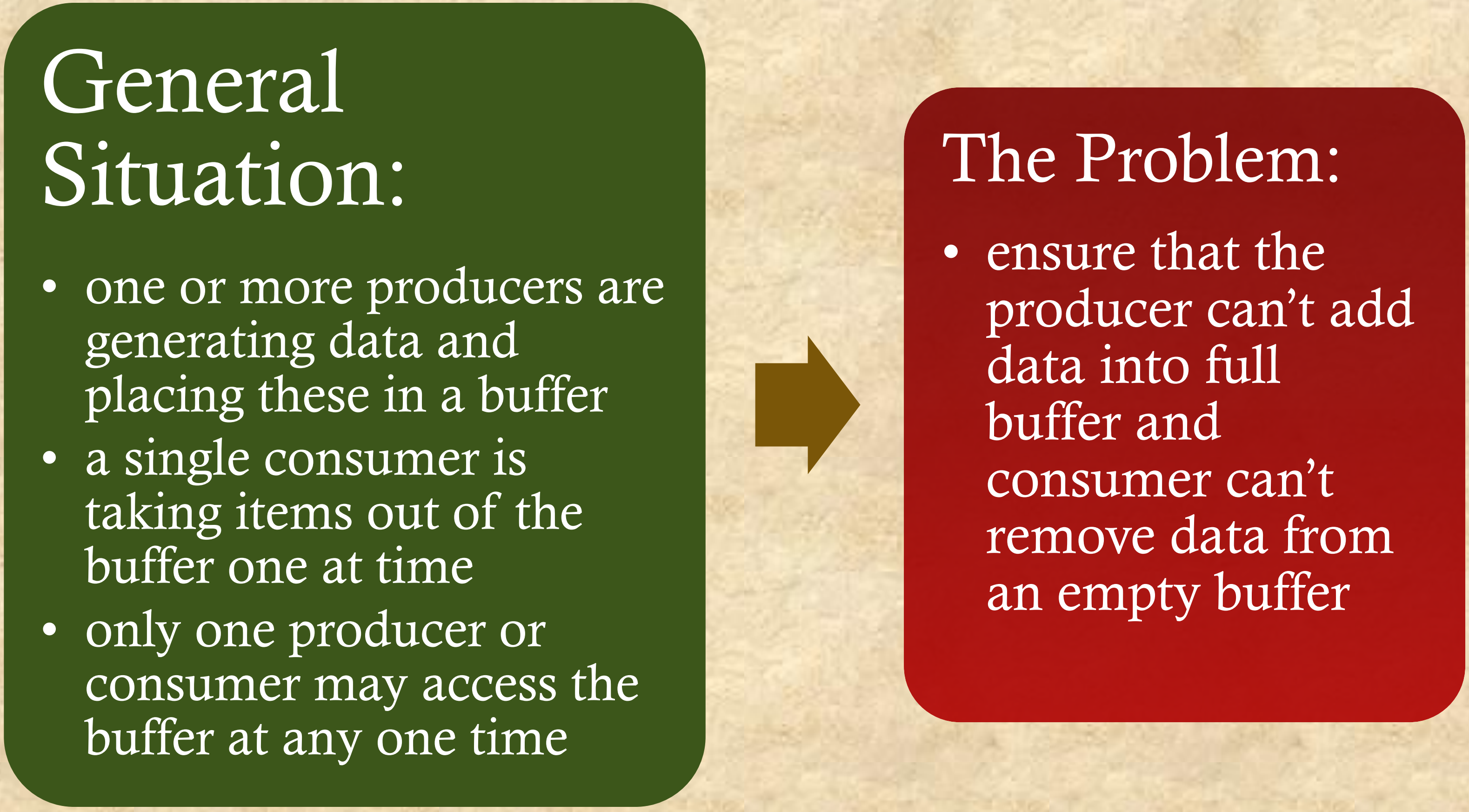
Shared Data Protected by a Semaphore



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

Producer/Consumer Problem



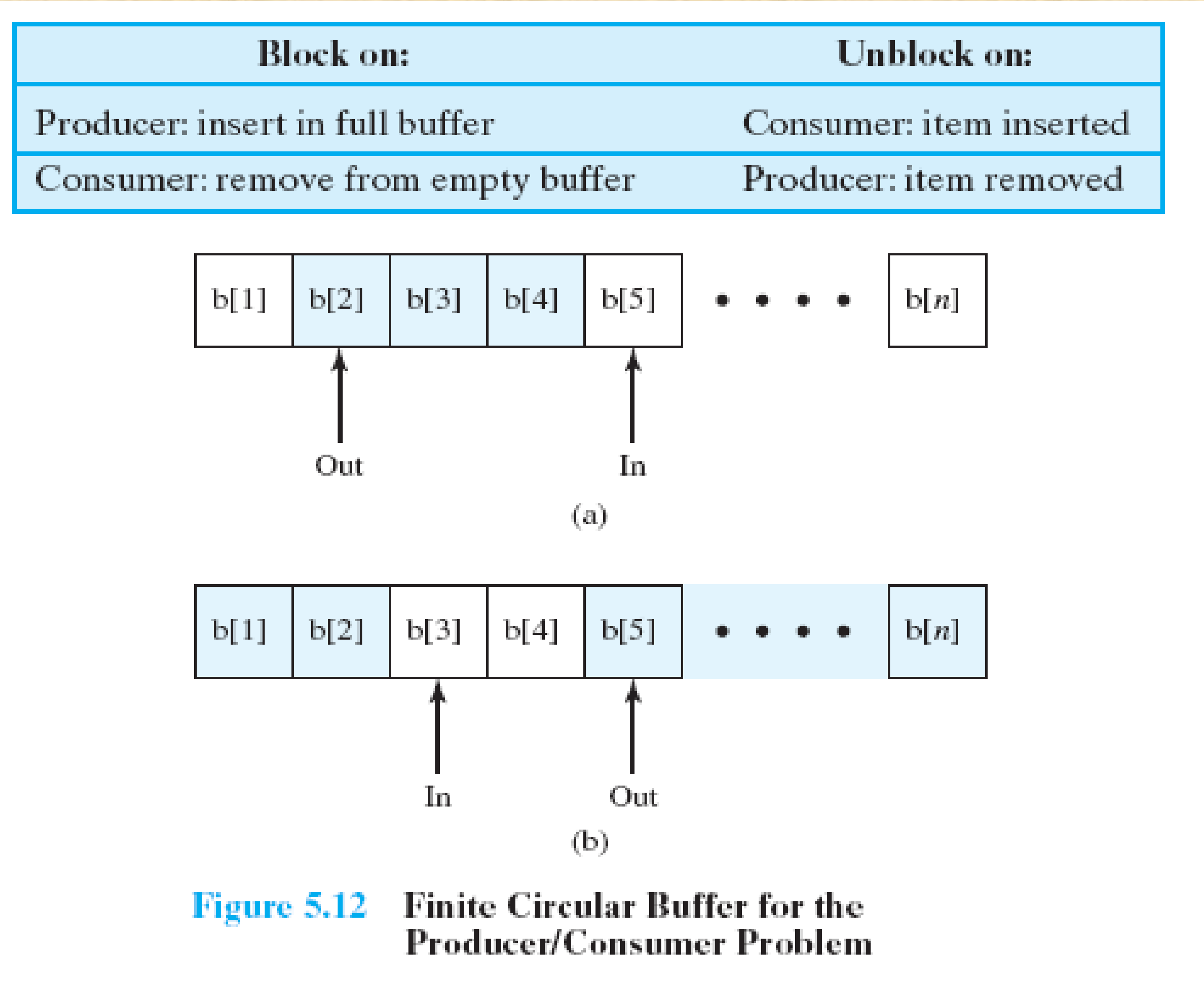
General Situation:

- 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 an empty buffer

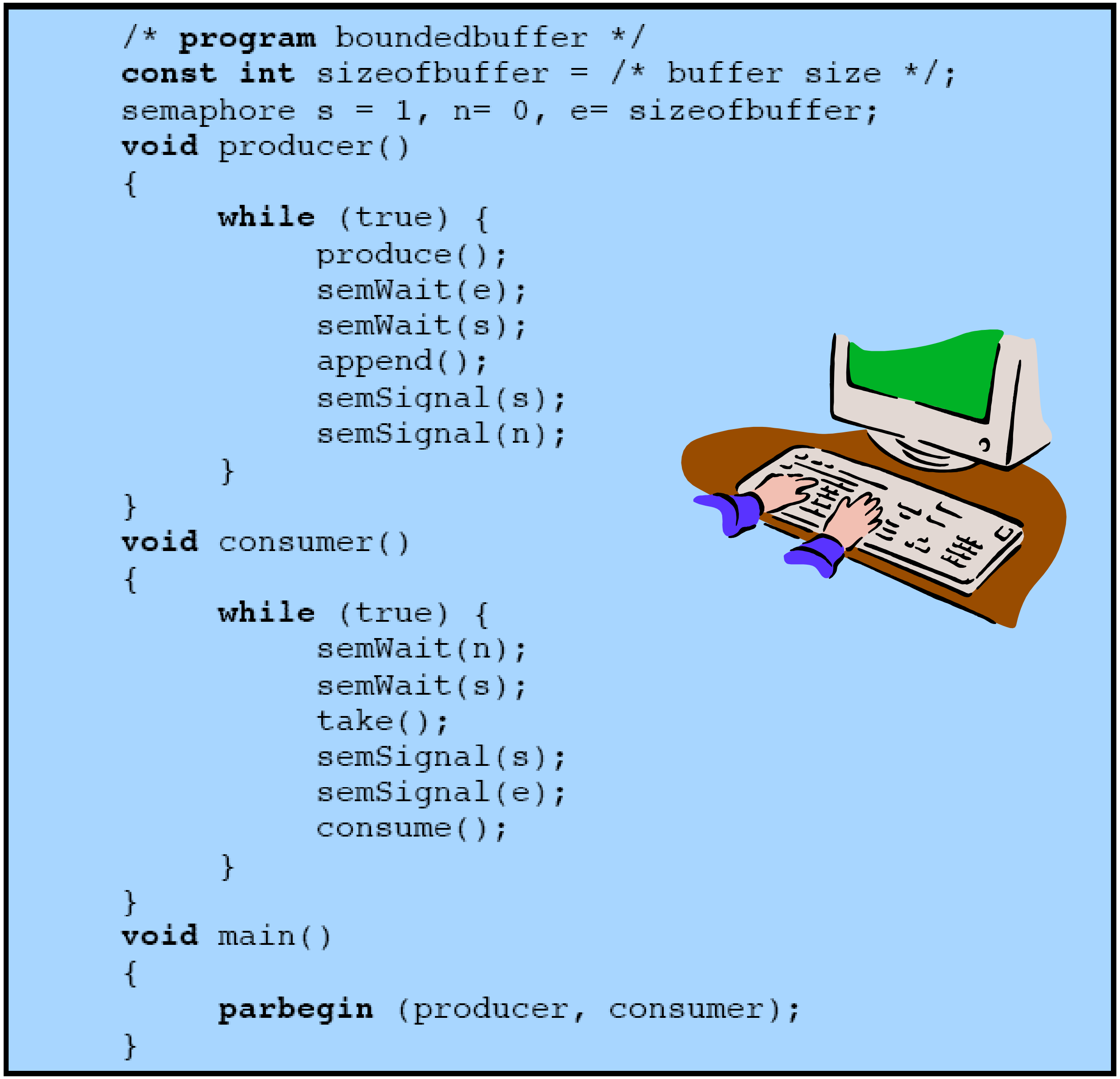
Finite Circular Buffer



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

Solution Using Semapho

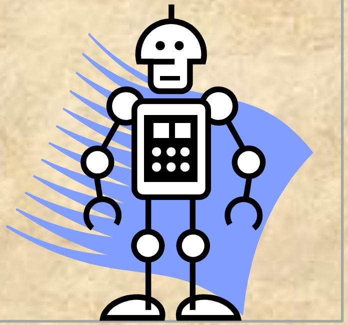


```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void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

Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

Implementation of Semaphores

- Imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives
- Can be implemented in hardware or firmware
- Software schemes such as Dekker's or Peterson's algorithms can be used
- Use one of the hardware-supported schemes for mutual exclusion



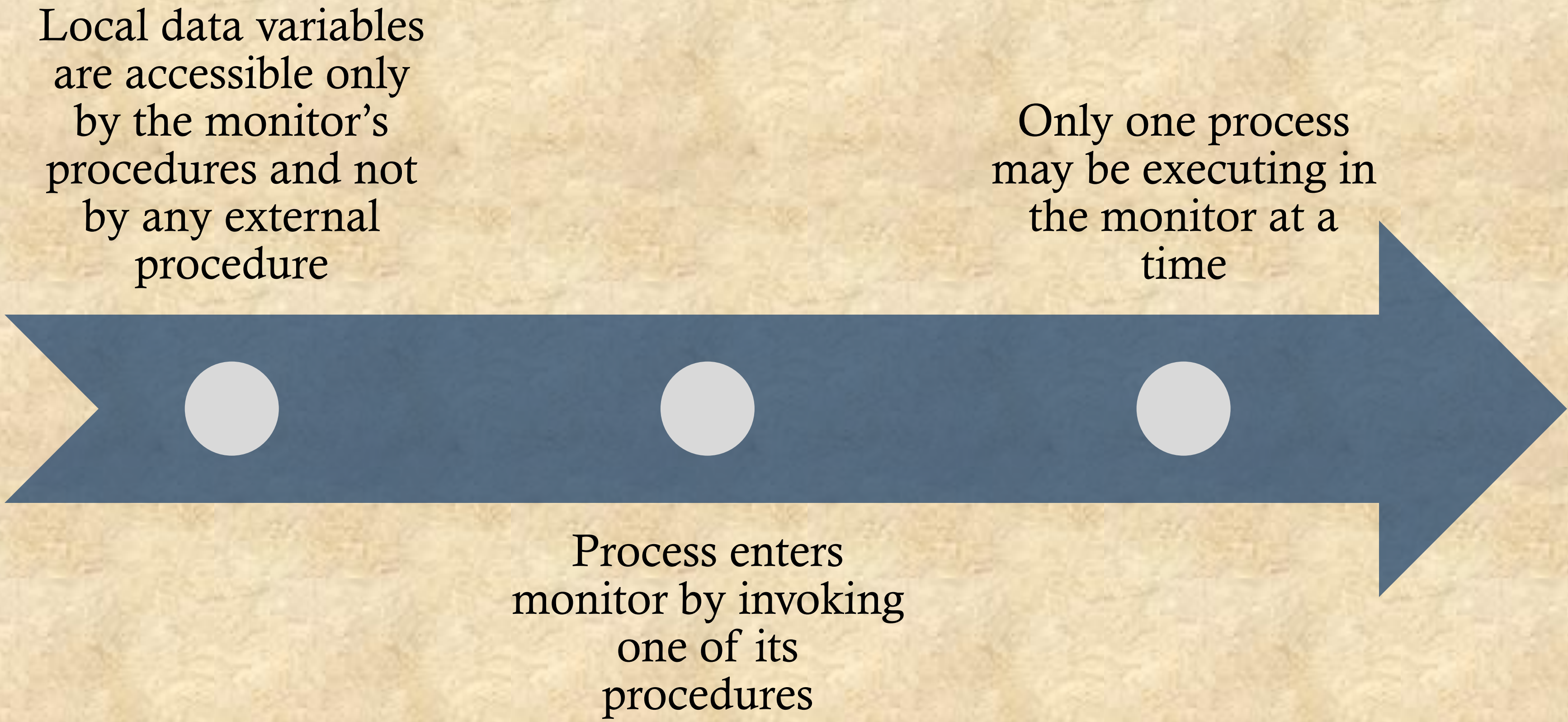
Review

- Concurrent processes, threads
- Access to shared data/resources
- Need to enforce mutual exclusion
- Hardware mechanisms have limited usefulness
- Semaphores: OS mechanism for mutual exclusion & other synchronization issues
- Standard semaphore/counting
- Binary semaphore
- Producer/consumer problem

Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

Monitor Characteristics



Local data variables are accessible only by the monitor's procedures and not by any external procedure

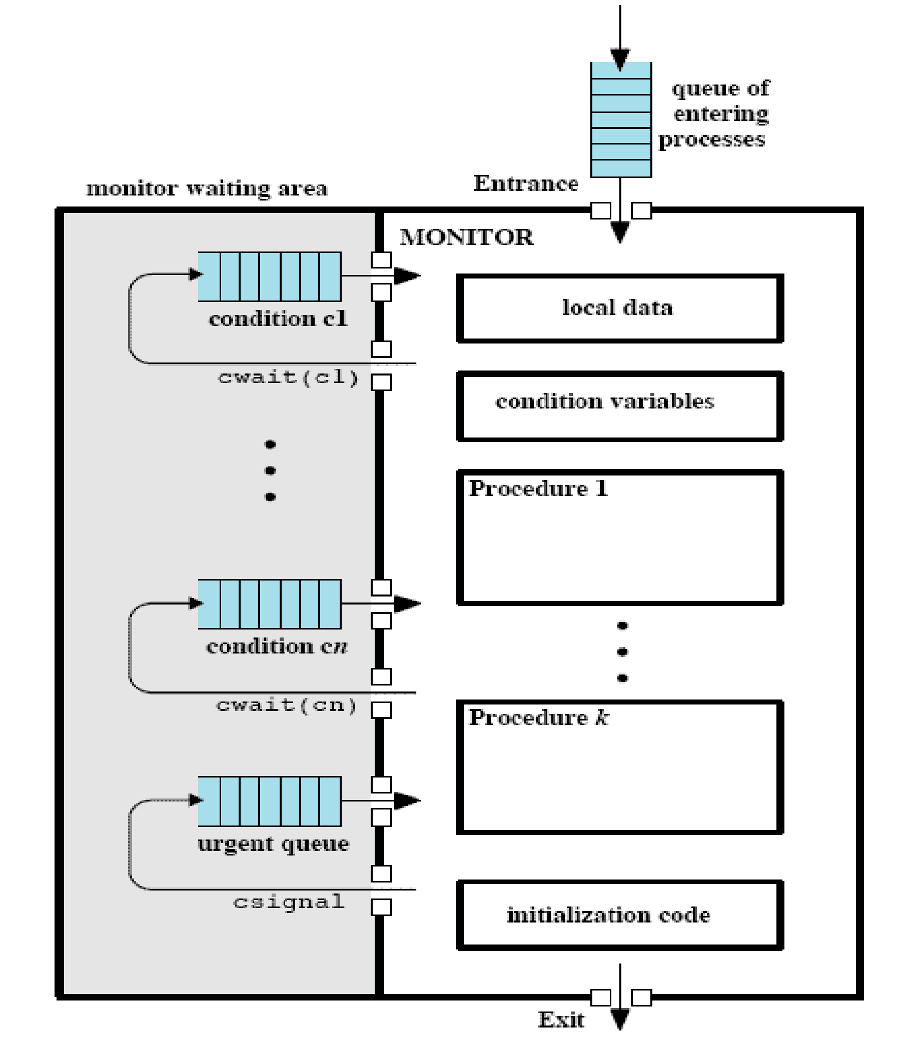
Process enters monitor by invoking one of its procedures

Only one process may be executing in the monitor at a time

Synchronization

- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are operated on by two 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

Structure of a Monitor



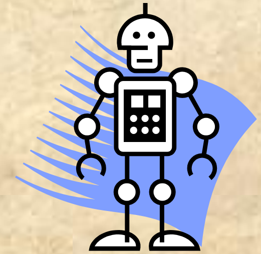
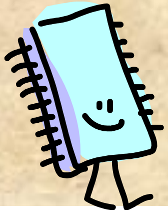
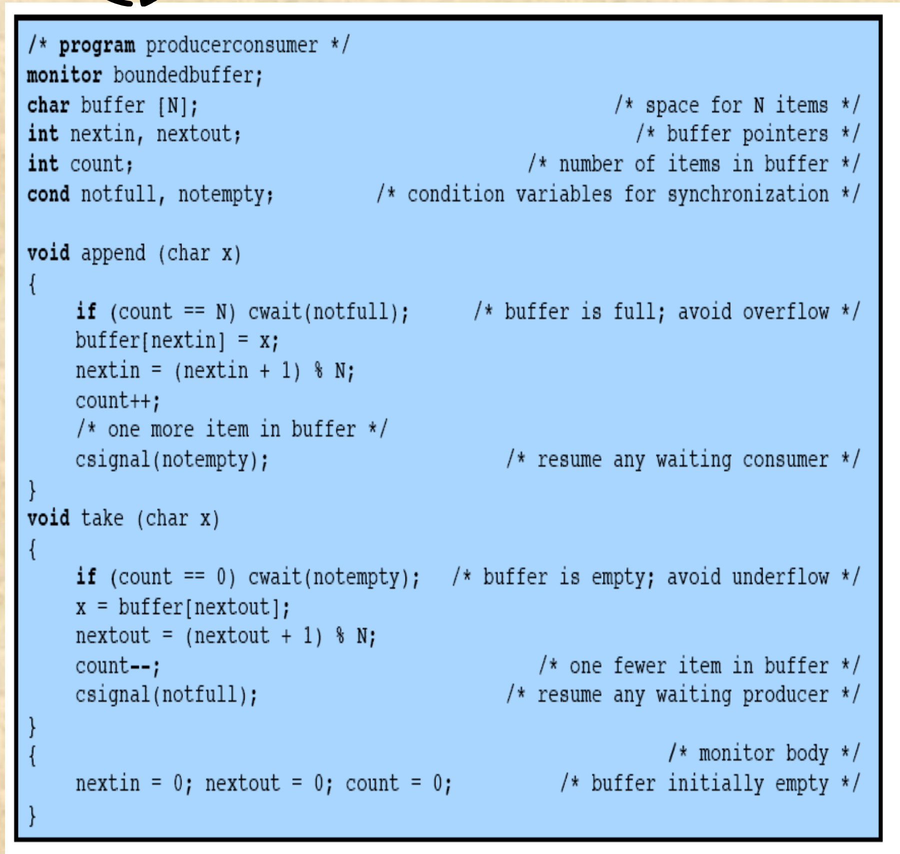


Figure 5.15 Structure of a Monitor

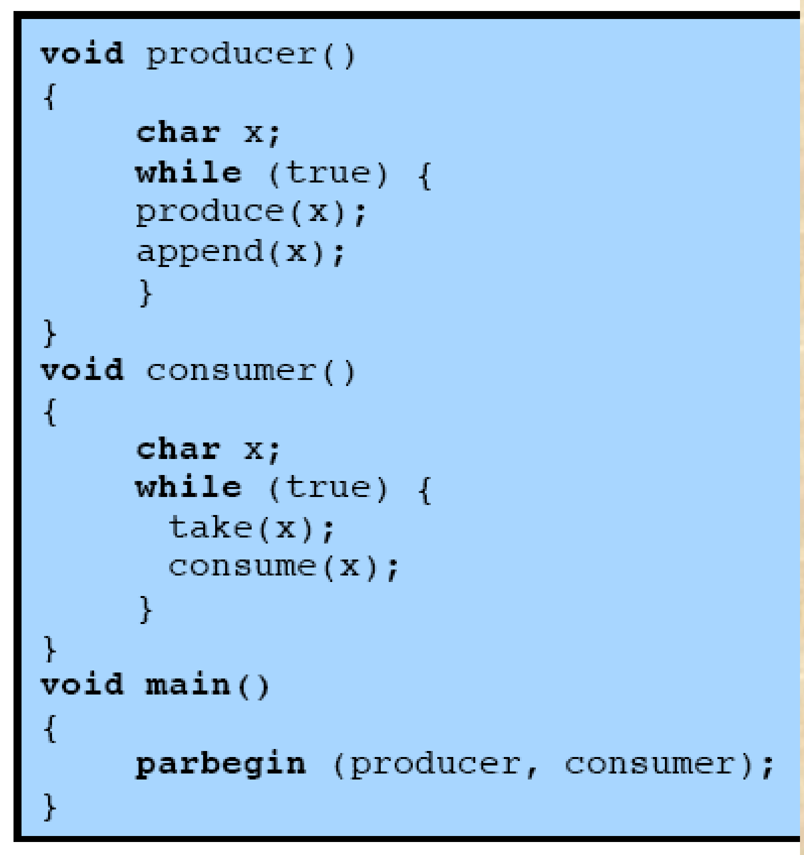
Problem Solution Using a Monitor





```
/* 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 */
}
```



```
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);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

Message Passing

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

synchronization

- to enforce mutual exclusion

communication

- to exchange information

- Message Passing is one approach to providing both of these functions
  - works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

Message Passing

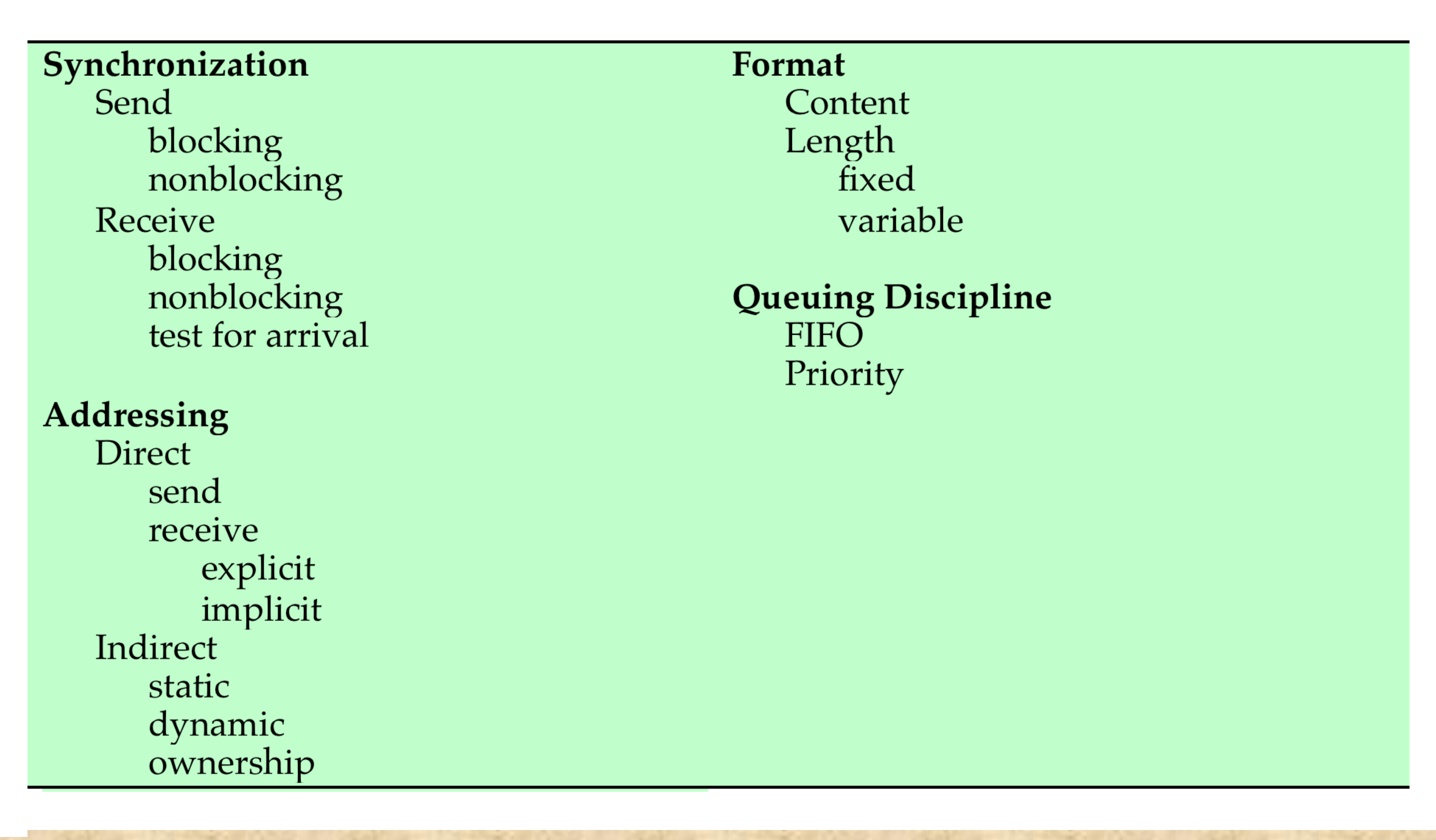
- The actual function is normally provided in the form of a pair of primitives:

  send (destination, message)

  receive (source, message)

- 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*

Message Passing



Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

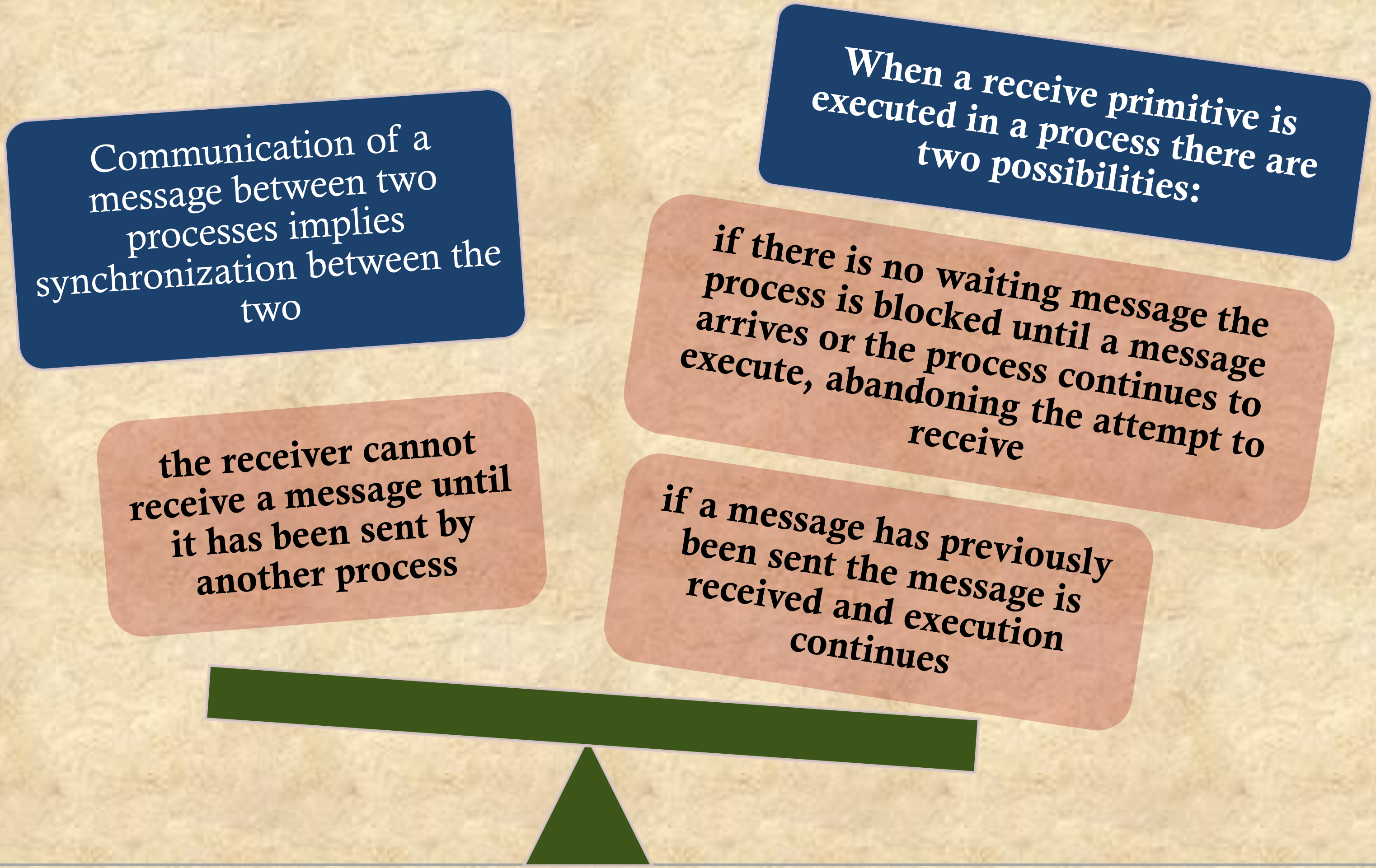
Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queuing Discipline
- FIFO
- Priority

Table 5.5 Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Synchronization



Communication of a message between two processes implies synchronization between the two

the receiver cannot receive a message until it has been sent by another process

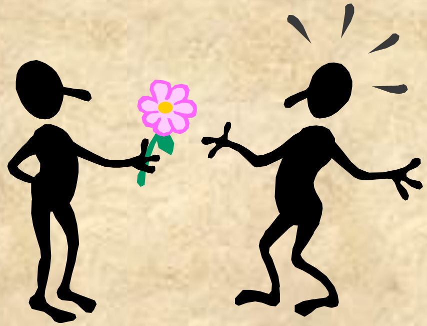
When a receive primitive is executed in a process there are two possibilities:

if there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive

if a message has previously been sent the message is received and execution continues

Blocking Send, Blocking Receive

- Both sender and receiver are blocked until the message is delivered
- Sometimes referred to as a *rendezvous*
- Allows for tight synchronization between processes



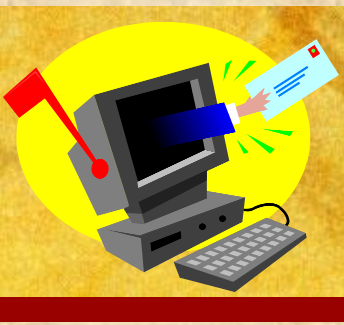
Nonblocking Send

Nonblocking send, blocking receive

- sender continues on but receiver is blocked until the requested message arrives
- most useful combination
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

Nonblocking send, nonblocking receive

- neither party is required to wait



Addressing

- Schemes for specifying processes in `send` and `receive` primitives fall into two categories:

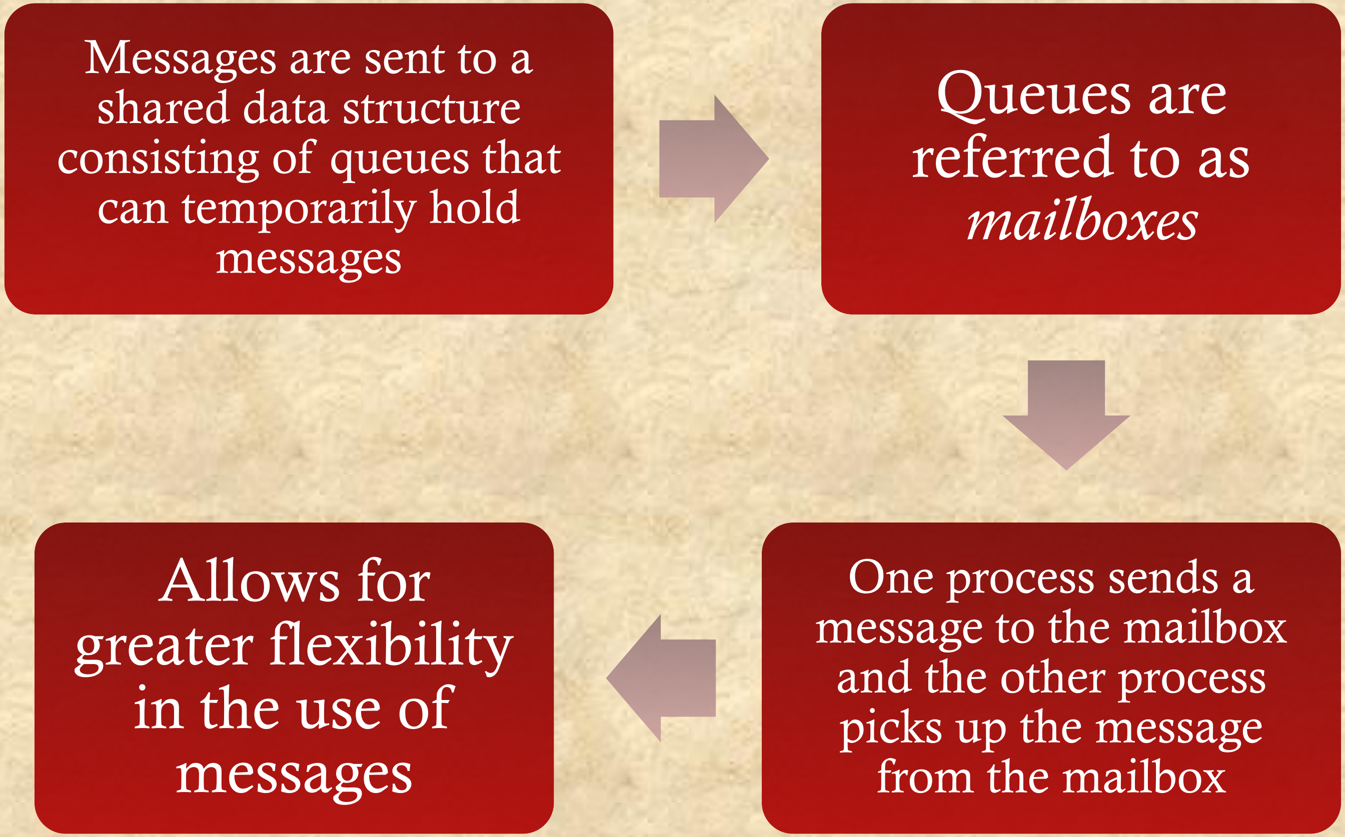
Direct addressing

Indirect addressing

Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process
    - effective for cooperating concurrent processes
  - implicit addressing
    - source parameter of the receive primitive possesses a value returned when the receive operation has been performed

Indirect Addressing



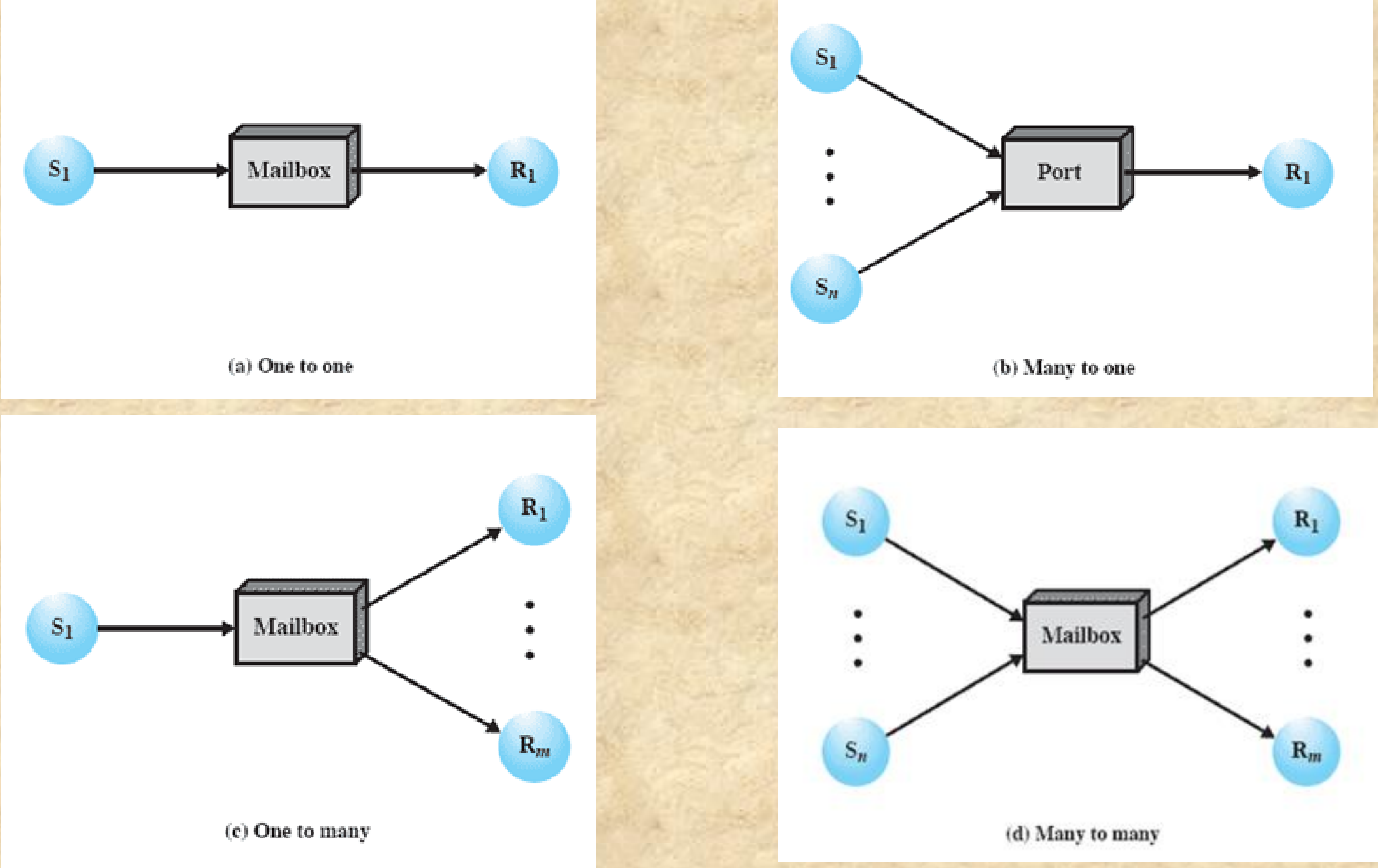
Messages are sent to a shared data structure consisting of queues that can temporarily hold messages

Queues are referred to as *mailboxes*

One process sends a message to the mailbox and the other process picks up the message from the mailbox

Allows for greater flexibility in the use of messages

Indirect Process Communication



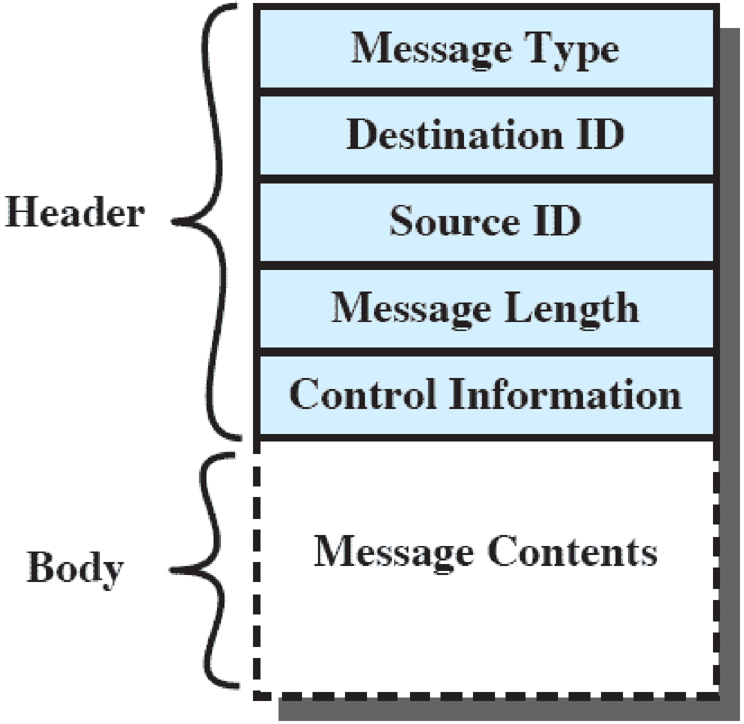
(a) One to one

(b) Many to one

(c) One to many

(d) Many to many

General Message Format



Header

Message Type

Destination ID

Source ID

Message Length

Control Information

Body

Message Contents

Figure 5.19 General Message Format

Summary

Messages

- Useful for the enforcement of mutual exclusion discipline

Operating system themes are:

- Multiprogramming, multiprocessing, distributed processing
- Fundamental to these themes is concurrency
  - issues of conflict resolution and cooperation arise

Mutual Exclusion

- Condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time
- One approach involves the use of special purpose machine instructions

Semaphores

- Used for signaling among processes and can be readily used to enforce a mutual exclusion discipline