Concurrency and Synchronization

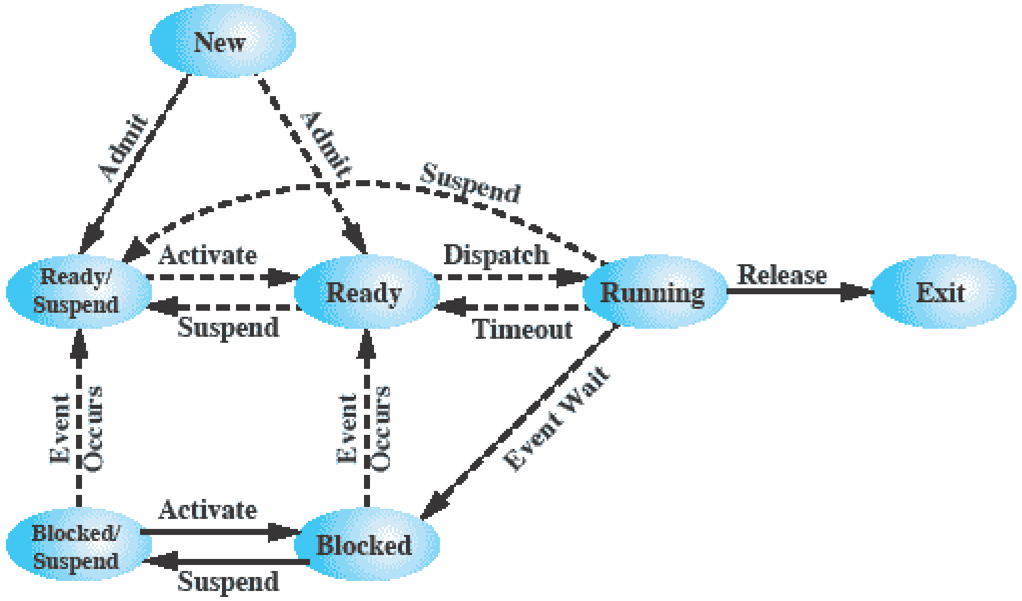
Part I

S.Rajarajan
SASTRA

Basics of Concurrency and its Problems

What are Concurrent processes ?

- At any point of time there are a number of processes existing in the memory for execution.
- Such coexisting processes are termed as concurrent processes.



Concurrent processes present in different states

Why synchronization necessary among concurrent processes?

- Coexisting processes need to become cooperating processes.
- Otherwise they will be competing for resources and will lead to conflicts in resource usages.
- Synchronization is enforcing an ordering among the processes aiming for the same resource.
- The basic technique used to implement synchronization is to block a process until an appropriate action is performed by another process or until a condition is fulfilled.

What leads to the presence of concurrent processes?

- Central to the design of modern OS is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- In all these cases multiple processes are handled simultaneously (concurrent processes)

- Fundamental design issue is Concurrency, it includes
  - Facilitating communication among processes
  - Sharing of resources among competing processes without conflicts

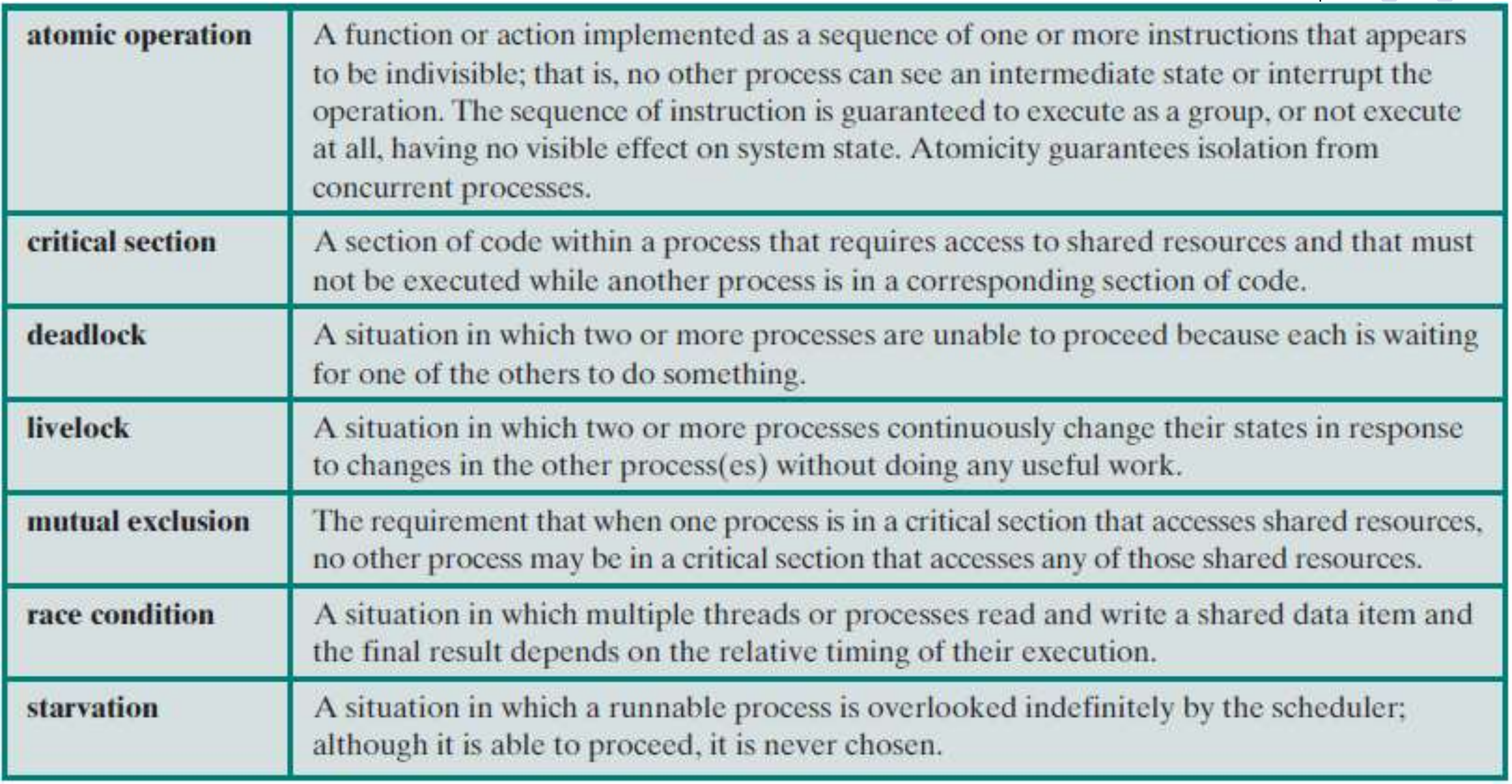
In a uni-processor system Concurrency arises in:

- Multiple independent applications
  - Sharing CPU’s time due to multiprogramming
- Structured applications
  - Extension of modular design with multiple concurrent processes or threads
- Operating system structure
  - OS themselves implemented as a set of processes or threads

What is the solution ?

- Basic requirement for supporting multiple concurrent processes is ***Mutual Exclusion***
- **Mutual exclusion** is the ability to exclude all other processes from a course of action while one process is granted the ability
- E.g. On a railway track when one train is already moving, no other train is permitted at the same time.

Important 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.

Atomic Operation

- For example while we swipe our debit cards at the ATM centers for withdrawing money, it involves two operations:
  - Our account is debited by the money we withdraw.
  - Dispensing the Cash from the ATM machine.
- These two operations are treated as atomic operations such that either both should succeed or none should succeed.

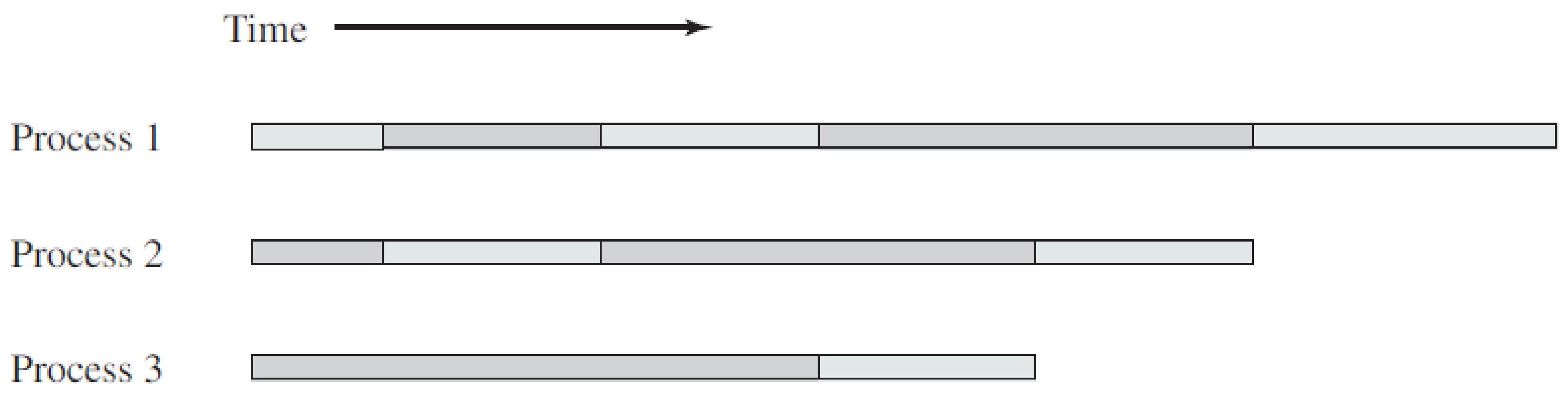
Critical Section

- Central to the concurrency problems is the usage of “shared resources” by the concurrent processes.
- When the same resource is needed by more than one process it becomes a potential conflict.
- Unless the processes are regulated they could result in mutual conflict and wrong usage.
- Critical section is the section of code of all those processes that are going to demand for a particular resource during the course of their execution.

Principles of Concurrency

Interleaving Processes

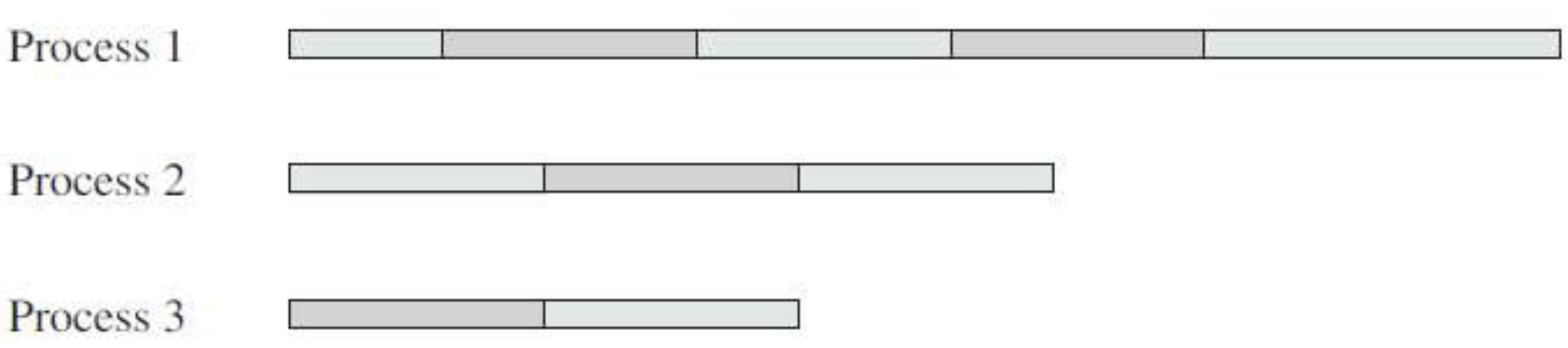
- In a single processor multi-programmed system, processes are interleaved in time to provide the appearance of simultaneous execution.
- Even though actual parallel processing is not achieved, interleaving provides major benefits.



(a) Interleaving (multiprogramming; one processor)

Overlapping Processes

In multiple processor systems it is possible not only to intervene processes, but also to overlap them.



(b) Interleaving and overlapping (multiprocessing; two processors)

- At first glance it may seem that interleaving and overlapping represent fundamentally different problems.
- But both are examples for concurrent processing and both present the same problems.
- In case of uni-processor systems , the problem arises due to multiprogramming
  - The relative speed of execution of processes is unpredictable.
  - It depends on other processes, interrupts and dispatcher’s scheduling policy.

- The following difficulties arise:
  - Sharing of global resources is fraught with peril.
  - Difficult for OS to optimally manage the allocation of resources
  - Difficult to locate programming errors as results are not deterministic and reproducible.

- All of the foregoing problems present themselves in multiprocessor system as well.
- Here too the relative speed of execution of processes is unpredictable.
- Multiprocessor system must also deal with problems arising from the simultaneous execution of multiple processes.

An example to understand the problem under Uni & Multi Processor systems

- Consider a procedure called ***echo*** that needs to be used by several applications.
- So it is better to make echo as a ***shared procedure***.
- Thus only a ***single copy of echo*** is kept in memory, saving space.
- But sharing can lead to problems

echo() procedure

```
void echo()
{
chin = getchar(); // User input taken through keyboard
chout = chin;  // chin value is assigned to chout variable
putchar(chout); // chout’s value is displayed
}
```

Problem with a Uniprocessor System

- Consider the following possible sequence of execution of P1 and P2 processes:

1. Process P1 invokes echo and is interrupted immediately after gechar returns its value and stores it in chin. At this point the most recently entered value ‘X’ is in chin.
2. Process P2 is activated and it invokes echo which runs without interruption, completing from inputting to displaying the character.
3. The process P1 is resumed. By this time the value ‘X’ has been overwritten by P2. So the present value of chin ‘Y’ is assigned to chout and displayed

- **<u>P1 Executes</u>** **chin** $\boxed{\text{Y}}$

chin = getchar(); <- input taken .

<Interrupted>

chout = chin;

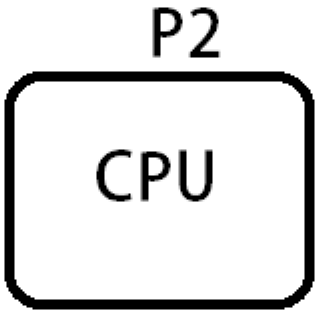
putchar(chout);

<u>P2 Executes</u>

chin = getchar(); <- input taken

chout = chin;

putchar(chout);



Problem with a Multiprocessor System

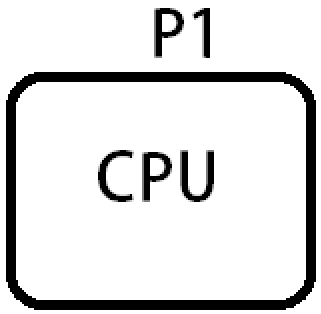
- Consider the following sequence

1. Process P1 and P2 are executing on separate processors. First P1 invokes echo procedure.
2. P1 calls getchar and receives input into chin. Now P1 is interrupted.
3. P2 starts execution on another processor. It gets input, assigns to chin, transfers to chout, displays it and ends.
4. When P1 resume on first processor it will display the value stored by P2.

chin $\boxed{Y}$

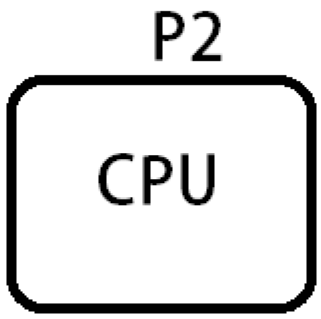
<u>P1 on CPU1</u>

```
chin = getchar();
<Interrupted>
chout = chin;
putchar(chout);
```



<u>P2 on CPU2</u>

```
.
chin = getchar();
chout = chin;
putchar(chout);
```



Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then:
- **On Uniprocessor:** - P1 invokes echo and interrupted. Then P2 starts but since P1 is already inside echo P2 is blocked.
- **On Multiprocessor:** - P1 runs on CPU1, invokes echo and is interrupted, when P2 attempts to enter echo it is blocked since P1 is inside.
- **This solution is called Mutual exclusion**

Race Condition

- A race condition occurs when multiple processes or threads read and write data items in such a way that the final result depends on the order of execution of instructions in the multiple processes.
- Let us consider two simple examples.
  - As a first example, suppose that two processes, P1 and P2, share the global variable a.
  - At some point in its execution, P1 updates a to the value 1, and at some point in its execution, P2 updates a to the value 2.
  - Thus, the two tasks are in a race to write variable a. In this example the “loser” of the race (the process that updates last) determines the final value of a.

Example 1

int a

<u>Process 1</u>
a=1

<u>Process 2</u>
a=2

- If the sequence of execution of the two processes is P1 followed by P2 then ‘a’ will be modified as
- P1: a=1 & P2: a=2
- So the final value of ‘a’ will be 2
- If the sequence of execution of the two processes is P1 followed by P2 then ‘a’ will be modified as
- P2: a=2 & P1: a=1
- Then the final value of ‘a’ will be 1

Example 2

b=1 c=2

<u>Process P3</u>

b = b + c

<u>Process P4</u>

c = b + c

- Though the two processes update different variables still the final values of the b & c depends on the order in which two processes are executed.
- If the order is P3 followed by P4 then b= 3, c= 5
- If the order is P4 followed by P3 then b= 4, c=3

Operating System Concerns on Concurrency

- What design and management issues are raised by the existence of concurrency?
- The OS must
  1. Keep track of various processes. Done with the help of process control blocks (PCBs)
  2. Allocate and de-allocate resources for active processes. These resources include
     - Processor time
     - Memory
     - Files
     - I/O devices

3. Protect the data and resources against interference by other processes.
4. Ensure that the processes and outputs are independent of the processing speed

Concurrency and Synchronization

Part 2

S.Rajarajan
SASTRA

Recap

- Concurrent processes & problems
- Shared resource
- Problems – race condition, data coherence, deadlock, starvation, livelock, Non-deterministic execution
- Mutual exclusion
- Common on uni & multiprocessor systems – interleaving & overlapping

Processes Awareness of Other Processes

Process Interaction

- 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
  - Processes ***unaware of each other***

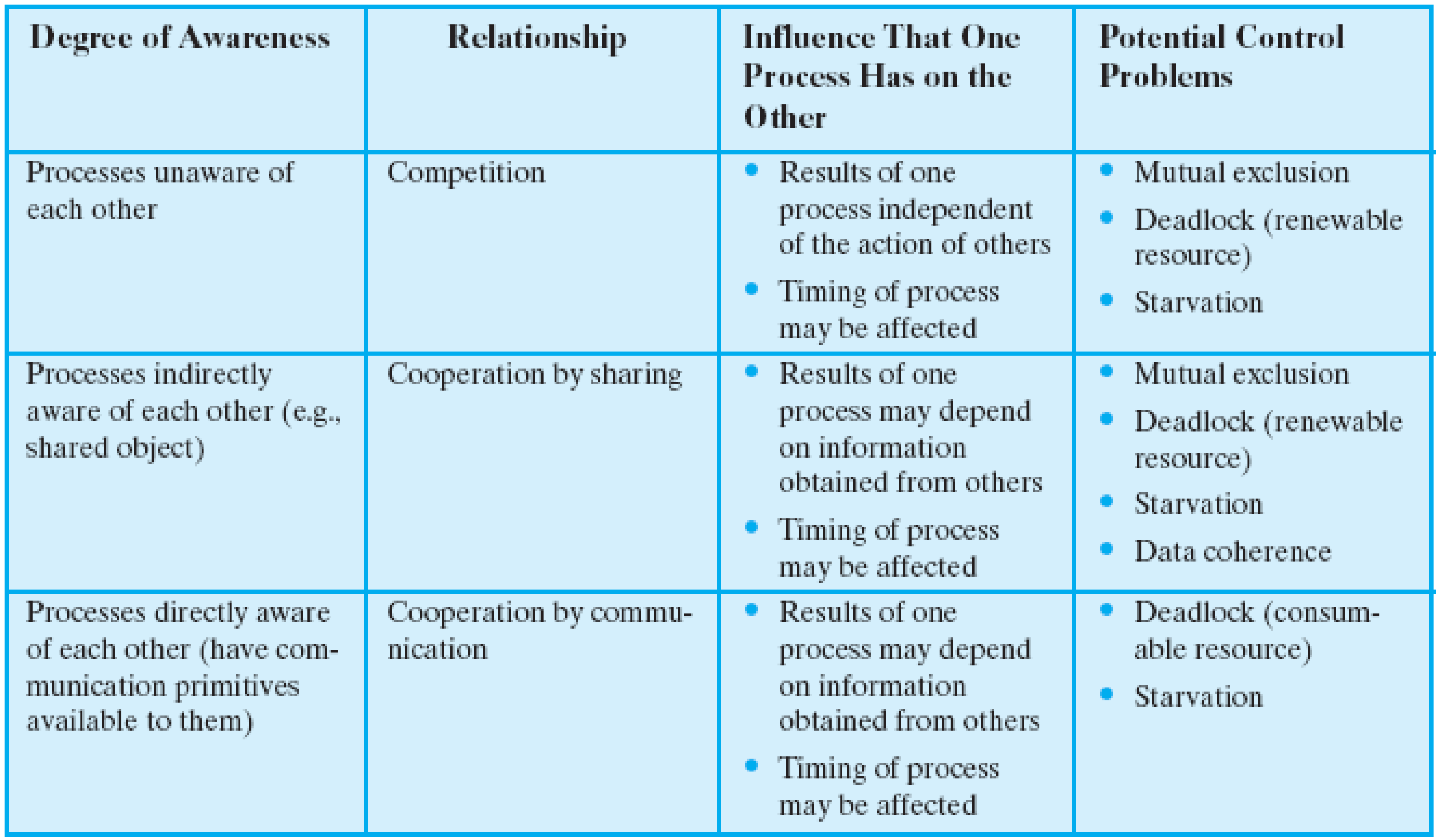
    e.g independent processes
  - Processes ***indirectly aware*** of each other

    e.g processes that share a common resource such as an IO buffer
  - Processes ***directly aware*** of each other

    e.g processes that interact with each other through IPC using their process Ids.

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 • 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

1. Competition among Processes for Resources

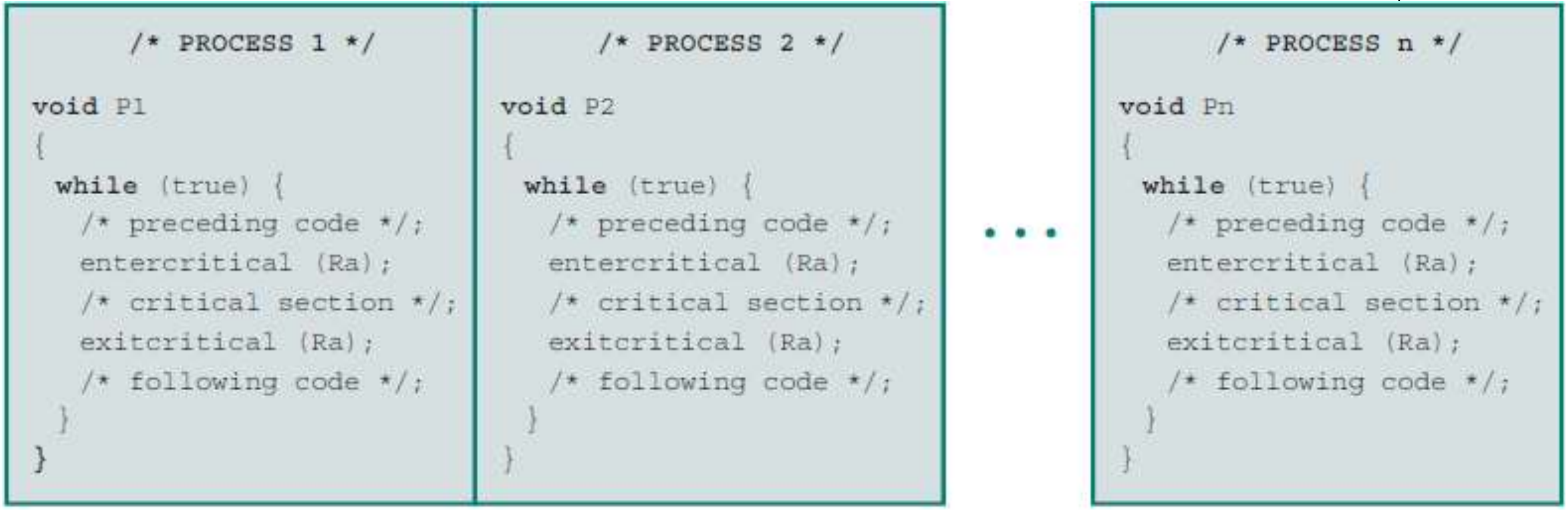
Three main control problems:

- Two or more processes need to access a resource
- Each process unaware of other process
- A process must leave the state of the resource it used unaffected by its usage
- Need for ***Critical sections*** and ***Mutual exclusion***
- But mutual exclusion may lead to ***Deadlock*** and ***Starvation***

What is needed

- ***Critical Resource****:-* A resource that leads to conflict between processes.
- **Critical Section**:-The portion of the program that uses it.
- **Mutual exclusion:-** It is important that only one program at a time be allowed in its critical section
- Only one program at a time to be allowed in its critical section.

Illustration of 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 */;
 }
}
```

The enforcement of mutual exclusion creates two additional control problems. One is that of **deadlock** and the other is **Starvation**

2. Cooperation among processes by sharing

- Processes interact with other processes without explicitly aware of them
- Multiple processes may have access to shared variables, shared files or databases
- Two modes - Reading and writing
- Control problems of deadlock and starvation are present, but only writing need to be exclusive.
- ***Data coherence*** also need to be enforced.
- E.g Producer-consumer, Reader-writer

- Consider that ‘a’ and ‘b’ are global variables accessed by P1 & P2
- Assume that initially a & b are initially equal (a = b = 2) and it has to be preserved.
- P1:
  - a=a+1; b=b+1; (a=2, b=2,a=3, b=3)
- P2
  - b=2 *b; a=2*a; (b=6,a=6)
- Data coherence may be affected due to interleaving based execution

P1	P2	P1	P2
a=a+1; (3)	b=2*b; (4)	b=b+1; (5)	a=2*a; (6)

(a=6, b=5)

3. Cooperation among processes by communication

- Processes communicate with each other by directly sending messages
- Because nothing is shared between processes mutual exclusion is not necessary
- But problems of deadlock and starvation might occur.
- As an example of deadlock, P1 is blocked waiting for message from P2 and at the same time P2 is also blocked waiting for a message from P1.

Mutual Exclusion

Requirements for ME

1. ME must be enforced: only one process at a time is allowed into its CS
2. A process that halts in its non CS must do so without interfering with other processes
3. A process must not be delayed access to a critical section when there is no other process using it
4. No assumptions are made about relative process speeds or number of processes
5. A process remains inside its critical section for a finite time only

Different ways for ME

- Leave the responsibility with the processes that wish to execute concurrently, processes would cooperate with one another- S/W approaches
- Special purpose machine instructions- H/W approaches
- Support from OS and programming language

Disabling Interrupts

- In uni-processor system, no overlapped execution and only 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
  - Efficiency of execution noticeably degraded

Pseudo-Code

```
while (true) {
/* disable interrupts */;

/* critical section */;

/* enable interrupts */;

/* remainder */;

}
```

Special Machine Instructions

- In a multiprocessor environment, processors access memory on a independent peer relationship
- But at the h/w level, access to memory location excludes any other access to the same location
- Using this feature, designers have proposed several machine instructions that carry out two actions atomically

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

Mutual Exclusion based on Compare & Swap

- A shared variable bolt is initialized to 0
- The only process that may enter its CS is the one that finds bolt equal to 0.
- All other processes go into a busy waiting mode
- Busy waiting means a process keeps on checking until it gets to see bolt to have 0 as its value.

Compare & Swap Instruction

Checks a local value ***word** against a **testval**, it they are same, replaces with a newval

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

```
int Compare_and_Swap(int *word, int testval, int newval)
{                                       0              0                1
        int oldval;
        oldval = *word; 0
        if ( oldval 0 == testval 0) *word = newval; 1
        return Oldval; 0;
}
const int n= /* number of processes */
int bolt;
void P( Int i) 1
{
    while( Compare_and_Swap( bolt 0 0 , 0 1)==1)
        /* do nothing */
    /* Critical Section */
    bolt = 0;
    /* remaining code */
}
void main()
{
    bolt = 0;
    parbegin(P(1), P(2), ...P(n));
}
```

bolt $\boxed{0\;1}$

```
int Compare_and_Swap(int *word, int testval, int newval)
{                                      1              0             1
        int oldval;
        oldval = *word; 1
        if ( oldval 1 == testval 0 ) *word = newval;
        return Oldval; 1;
}                                                               bolt  1
const int n= /* number of processes */
int bolt;
void P( Int i 2 )
{
    while( Compare_and_Swap( bolt 1 , 0 1)==1)
        /* do nothing */
    /* Critical Section */
    bolt = 0 ;
    /* remaining code */
}
void main()
{
    bolt = 0 ;
    parbegin(P(1), P(2), ...P(n));
}
```

Exchange instruction

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

Exchange Instruction

```
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;
        /remaining code */
    }
}
void main()
{
    bolt =0;
    parbegin (p(1), P(2)....P(n));
}
```

P1: Keyi $\boxed{1}$ bolt $\boxed{0}$

P2: Keyi $\boxed{0}$ bolt $\boxed{1}$

P(1) : exchange (1, 0)

P(2): exchange (1, 1)

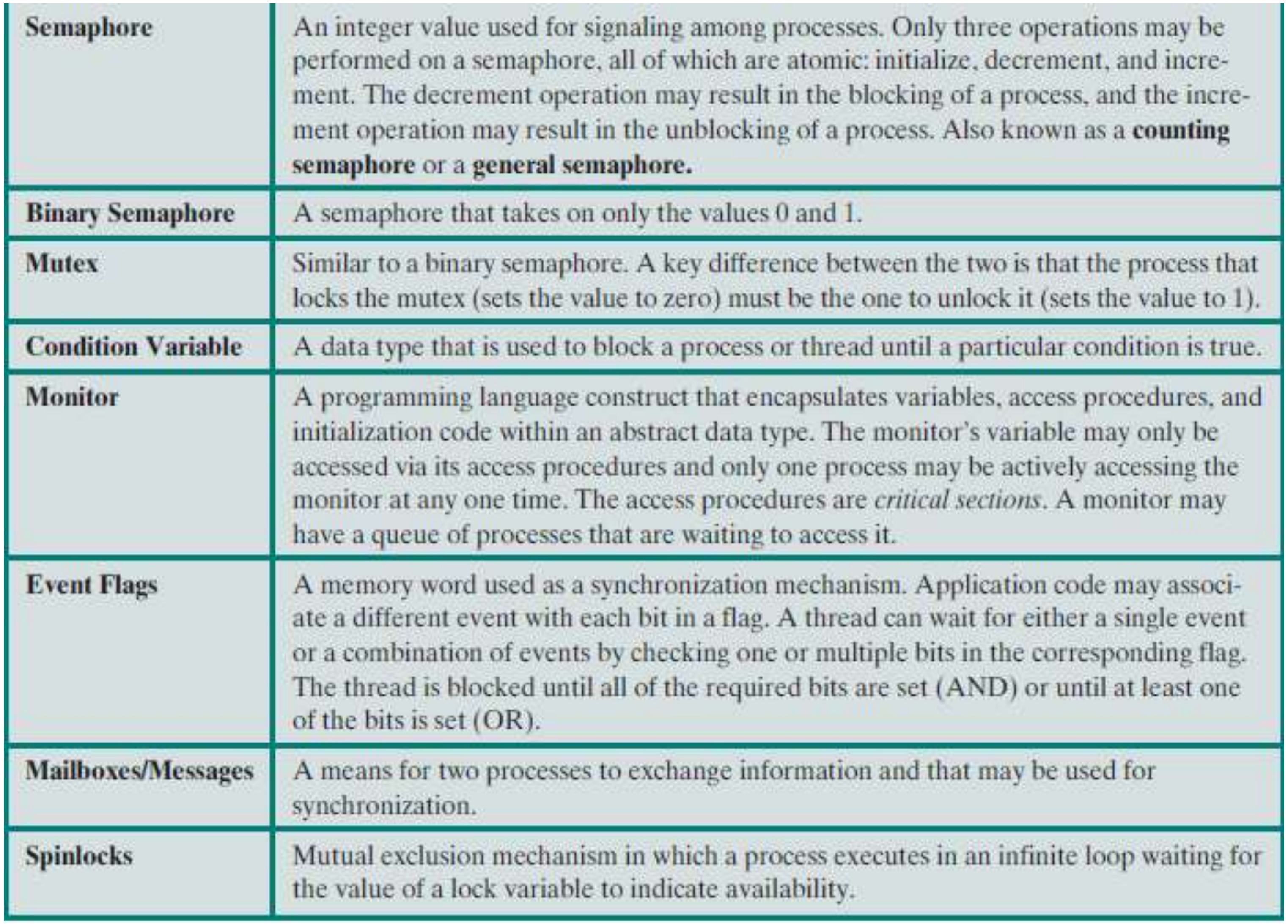
Hardware Mutual Exclusion: Advantages

- Applicable to any number of processes on single processor or multiple processors.
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections; each CS can be defined by its own variable

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

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

- OS mechanism
- Semaphore: An integer value used for signaling among processes.
- Only three operations may be performed on a semaphore, all of which are ***atomic***:
  - ***Initialized*** to non-negative value,
  - ***Decrements*** semaphore and if the value becomes –ve, process executing sem wait blocked (`semWait`)
  - ***Increment*** semaphore and if the resluting value is less than or equal to 0, then a blocked process is unblocked. (`semSignal`)

- The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.
- To transmit a signal via semaphore, a process executes ***semSignal***
- To receivea signal via semaphore, a process executes ***semWait***
- If the corresponding signal is nor received, the process is suspended until the transmission takes place.

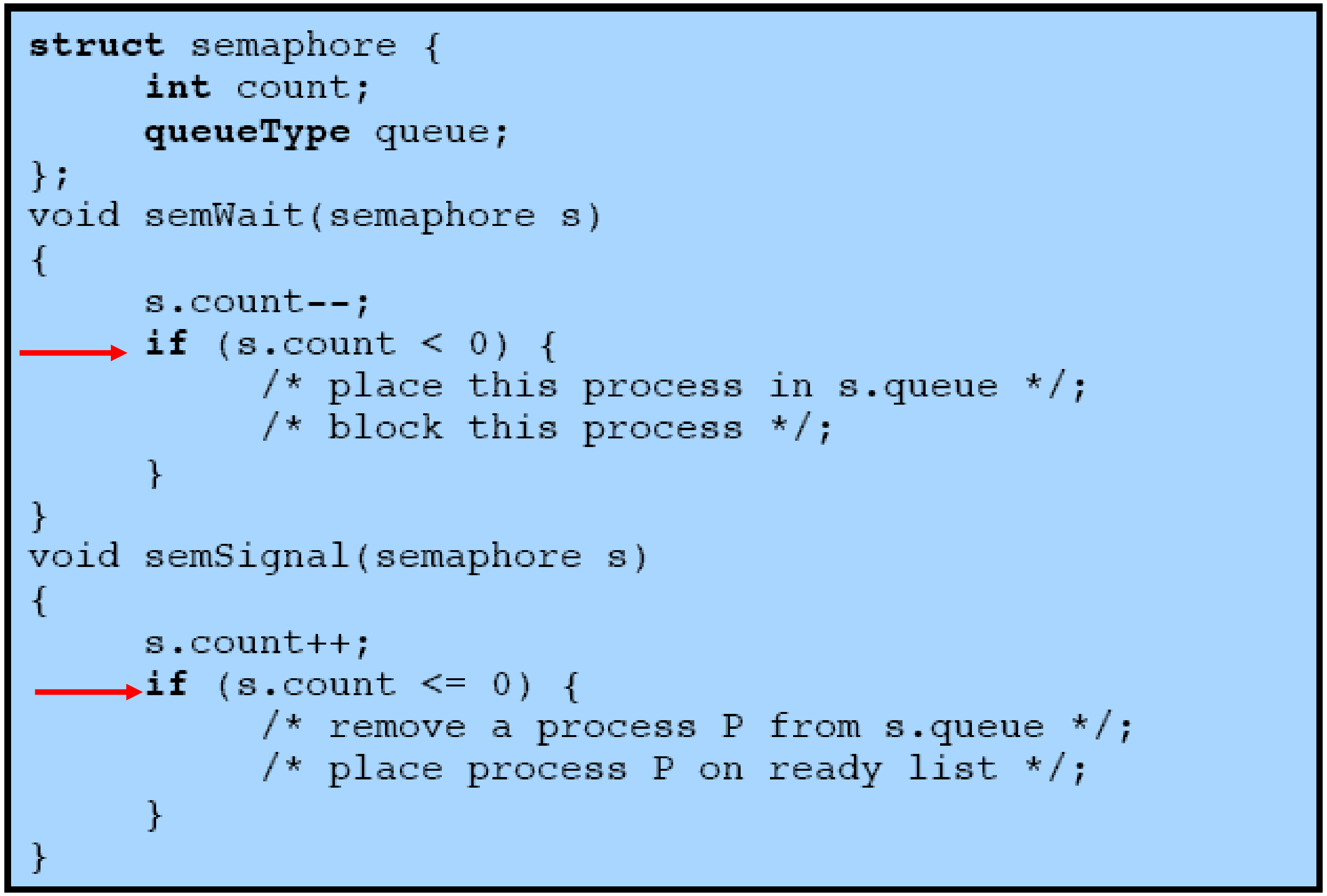
- To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which three operations are defined:
- 1. A semaphore may be initialized to a nonnegative value.
- 2. SemWait operation decrements the semaphone value. If the semaphore value becomes -ve, the process executing semwait is blocked.

- 3. SemSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by semWait is unblocked.
- Other than these three operations, there is no way to inspect or manipulate semaphones.

- The success of the protocol requires applications follow it correctly. Fairness and safety are likely to be compromised (which practically means a program may behave slowly, act erratically, hang or crash) if even a single process acts incorrectly. This includes:
- requesting a resource and forgetting to release it;
- releasing a resource that was never requested;
- holding a resource for a long time without needing it;
- using a resource without requesting it first (or after releasing it).

- Even if all processes follow these rules, *multi-resource* deadlock may still occur when there are different resources managed by different semaphores and when processes need to use more than one resource at a time

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

s.count -1 0

P1

SemWait()
Critical_Section
SemSignal()

P2

SemWait()
Critical_Section
SemSignal

Binary Semaphore Primitives

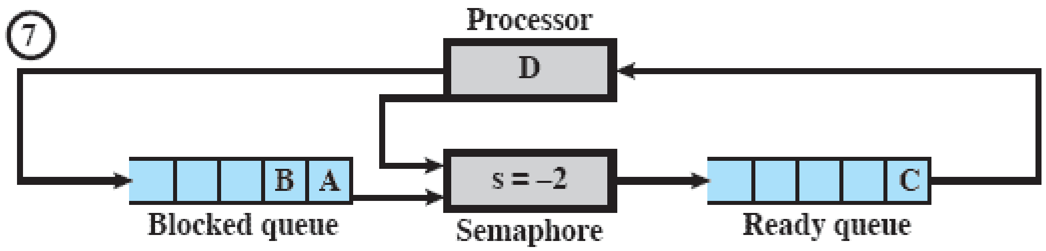
```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
               /* place this process in s.queue */;
               /* block this process */;
     }
}
void semSignalB(semaphore 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

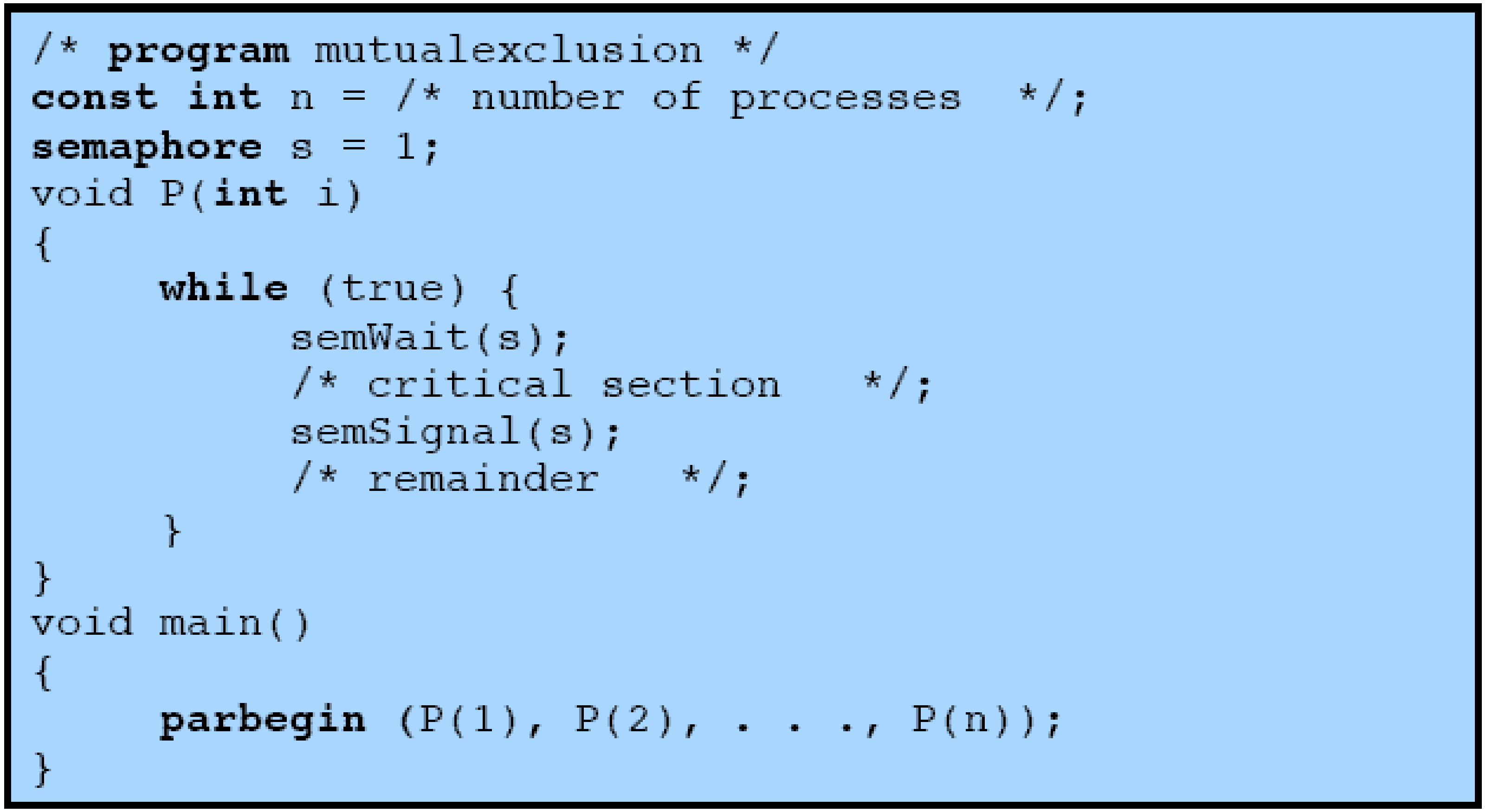
Strong/Weak Semaphore

- 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

Example of Strong Semaphore Mechanism



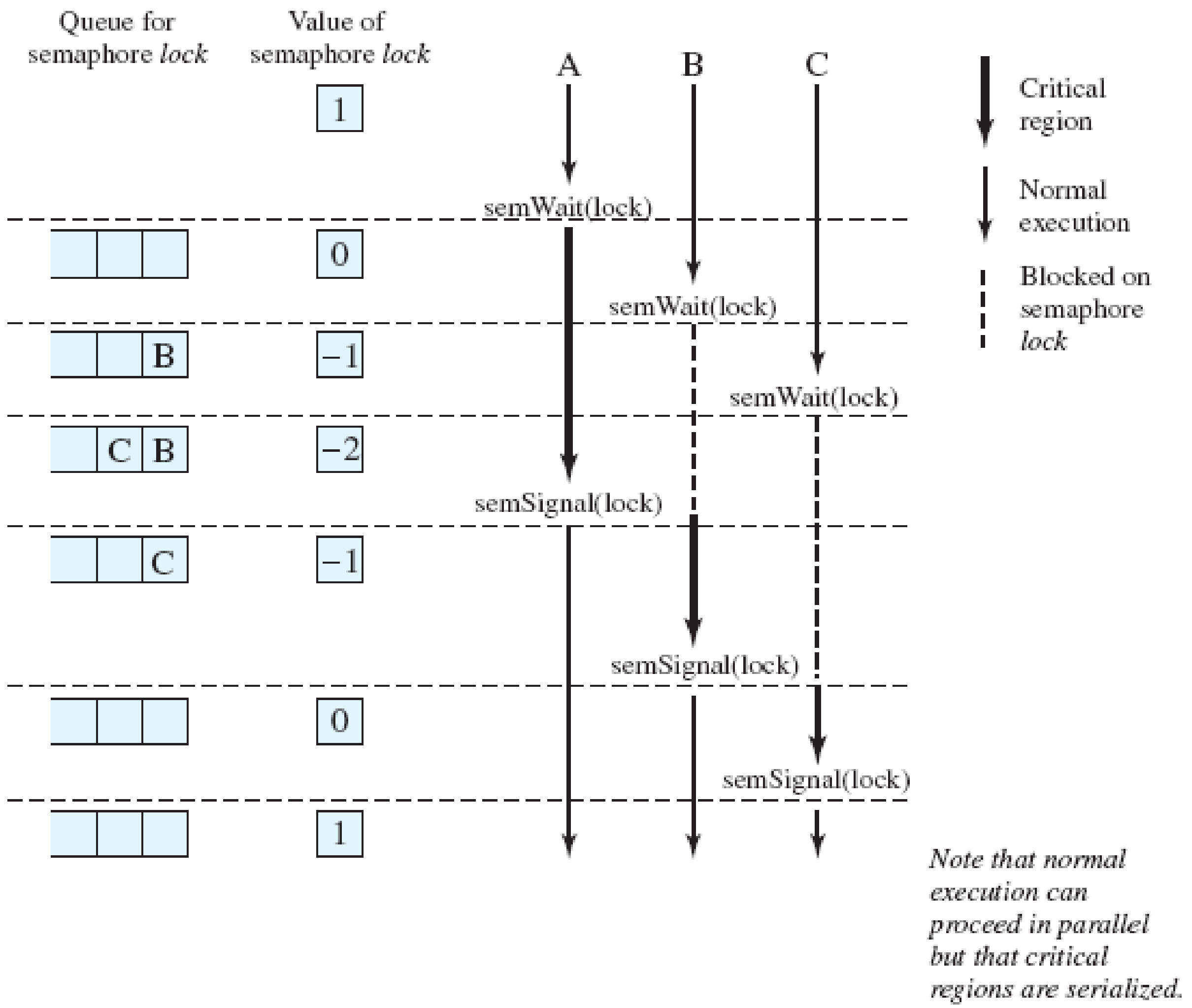
Mutual Exclusion Using Semaphores



```
/* 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

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**

Producer/Consumer Problem

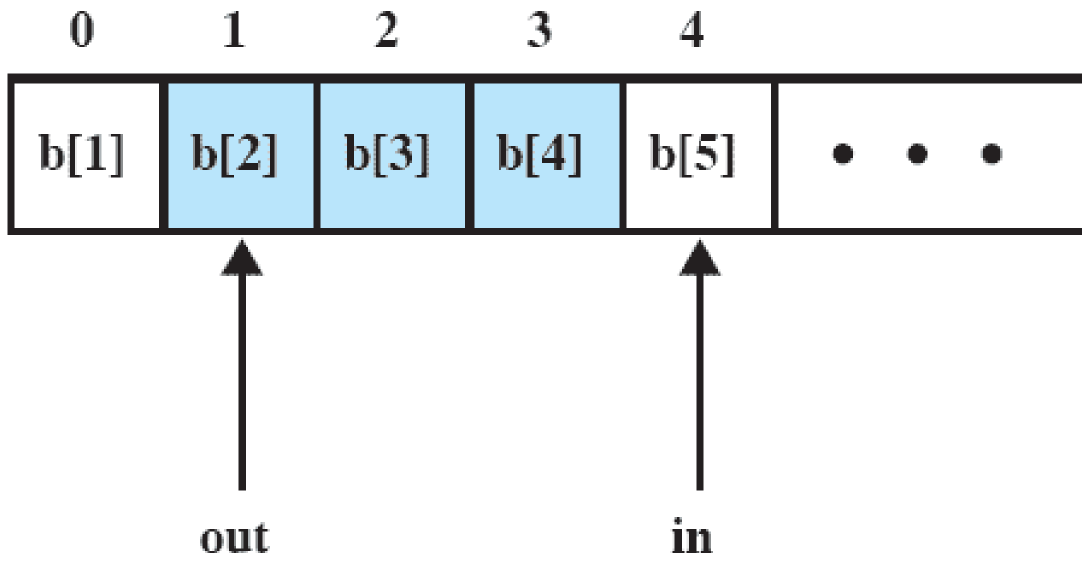
- **General Situation:**
  - One or more producers are generating data and placing these in a buffer
  - A single consumer is taking items out of the buffer one at time
  - Only one producer or consumer may access the buffer at any one time
- **The Problem:**
  - Either the producer or the consumer is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

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

Incorrect Solution

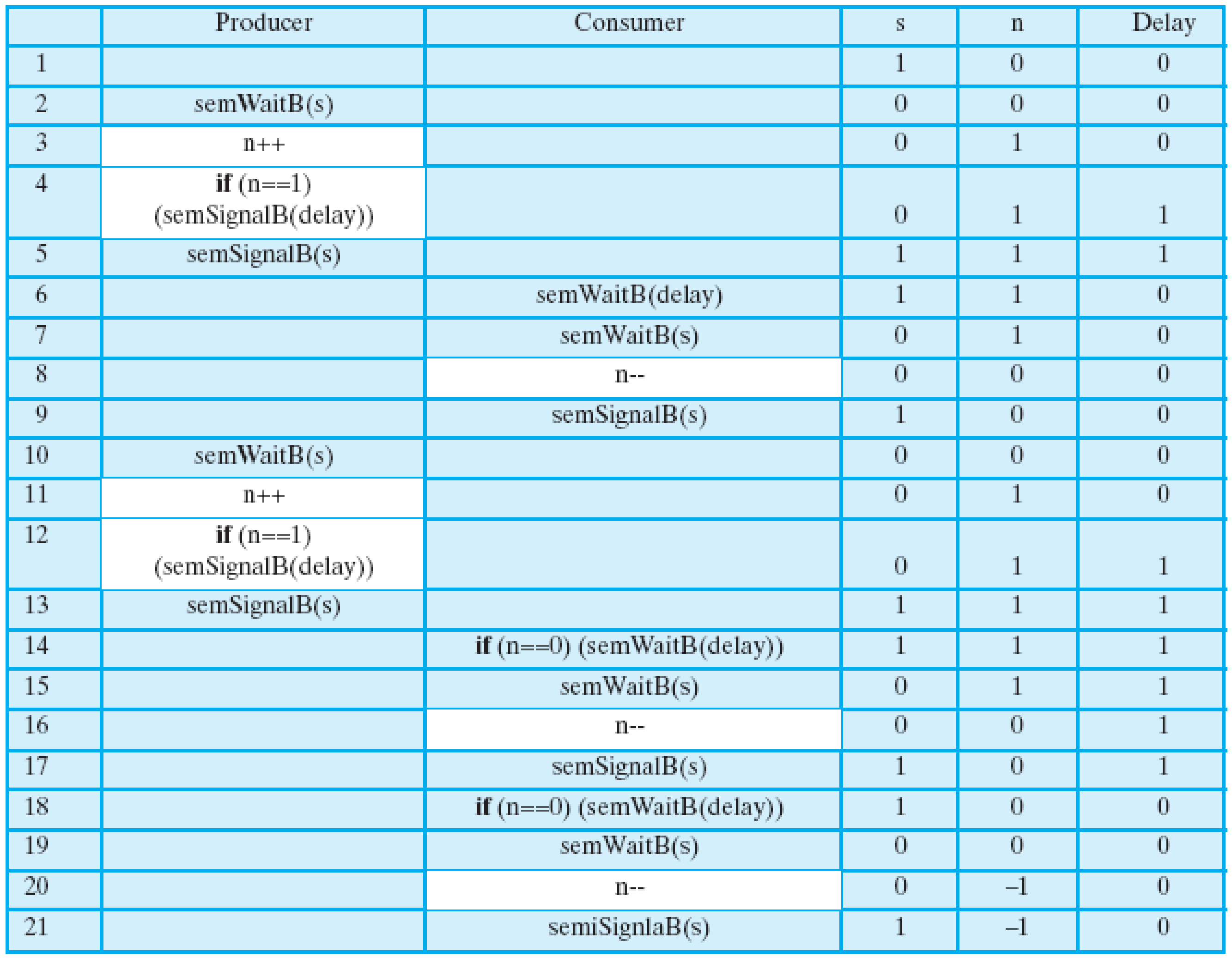
```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);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

Semaphores

S – for ME
delay – for consumer when buffer empty

Possible Scenario

Table 5.4 Possible Scenario for the Program of Figure 5.9

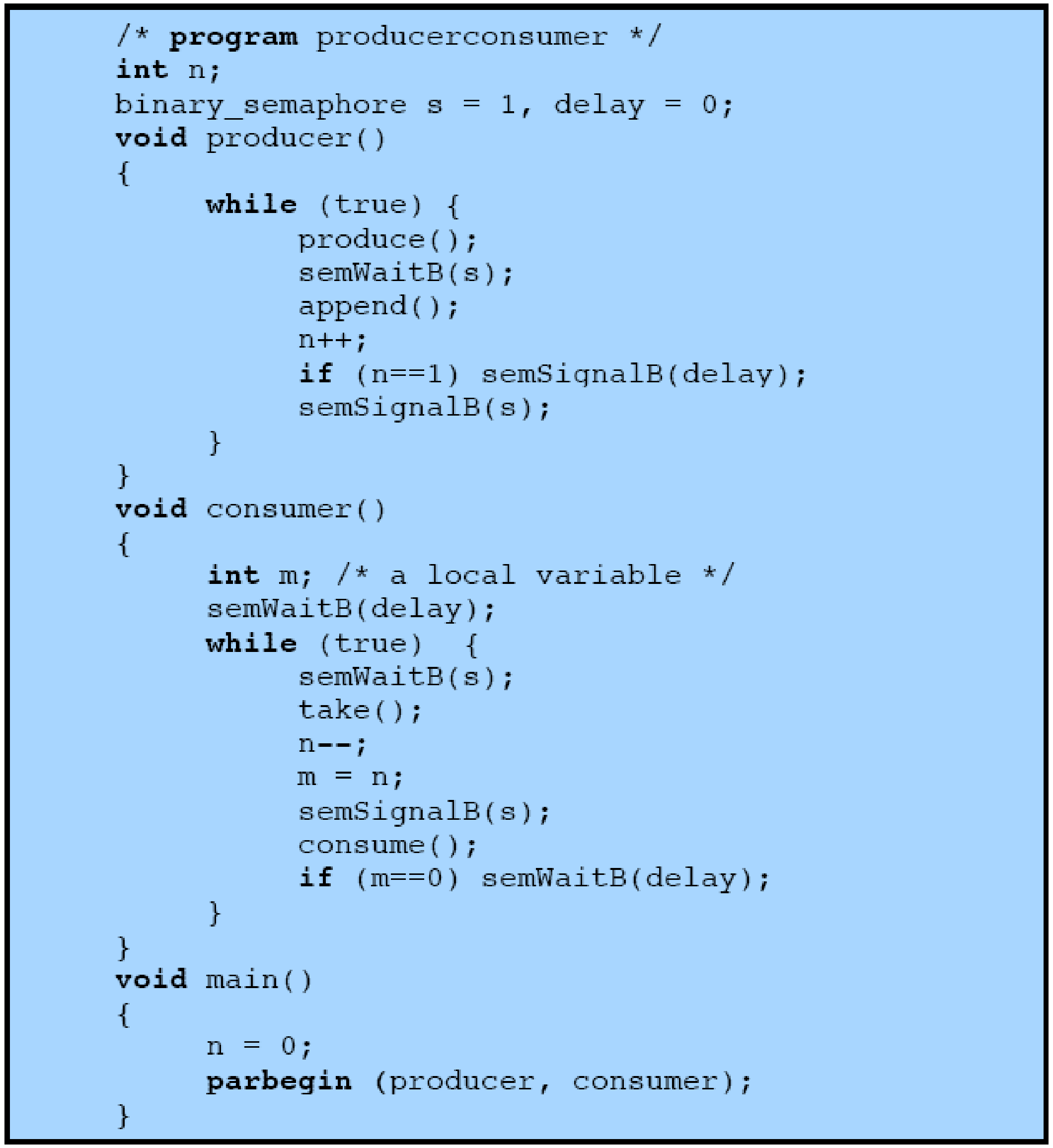


	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	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.

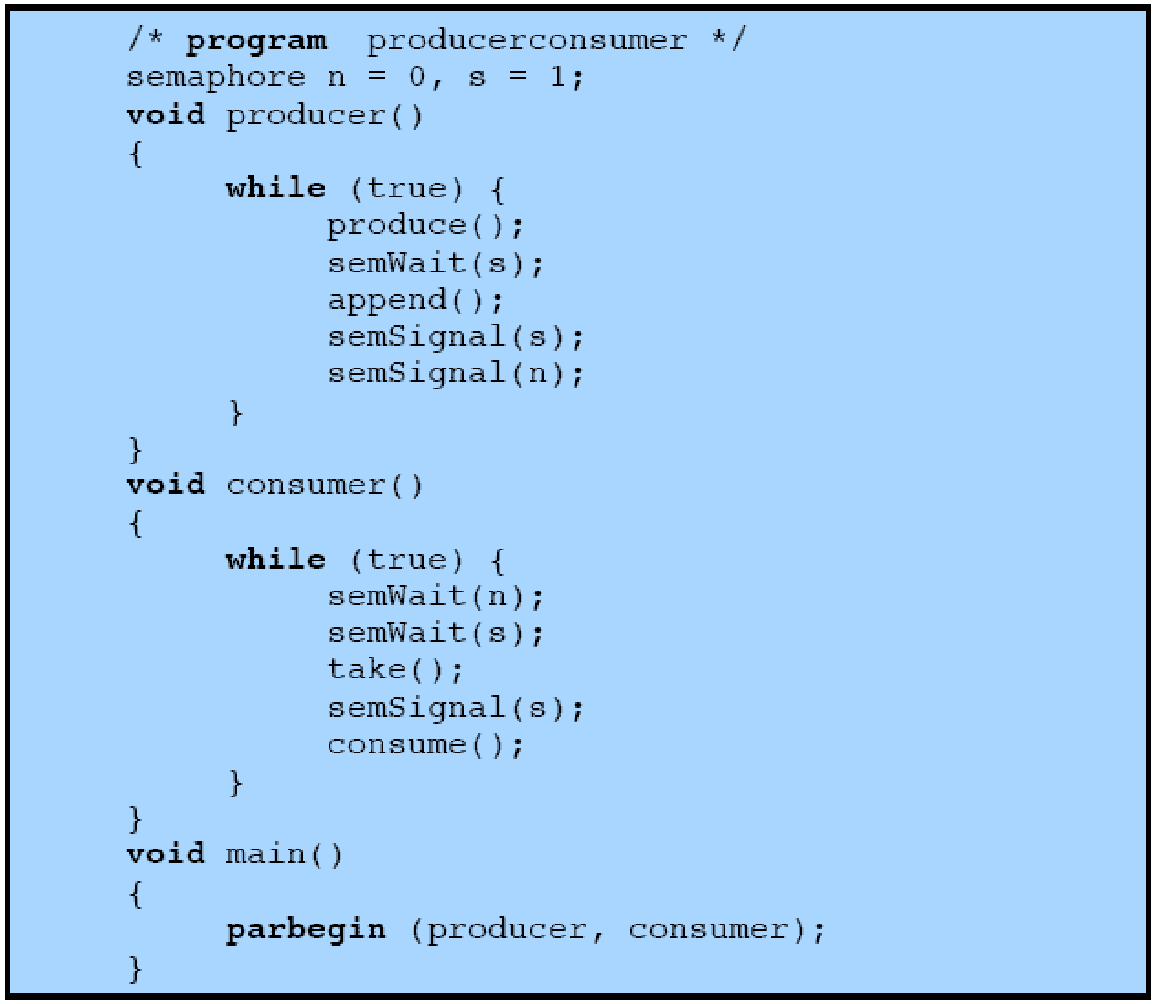
	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10		**if** (n==0) (semWaitB(delay))	1	0	0
11	semWaitB(s)		0	0	0
12	n++		0	1	0
13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

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);
}
```

General Semaphore



```
/* 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

<u>Semaphores</u>

s - ME

n – data count in buffer
to prevent consumer when buffer empty

Bounded Buffer

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

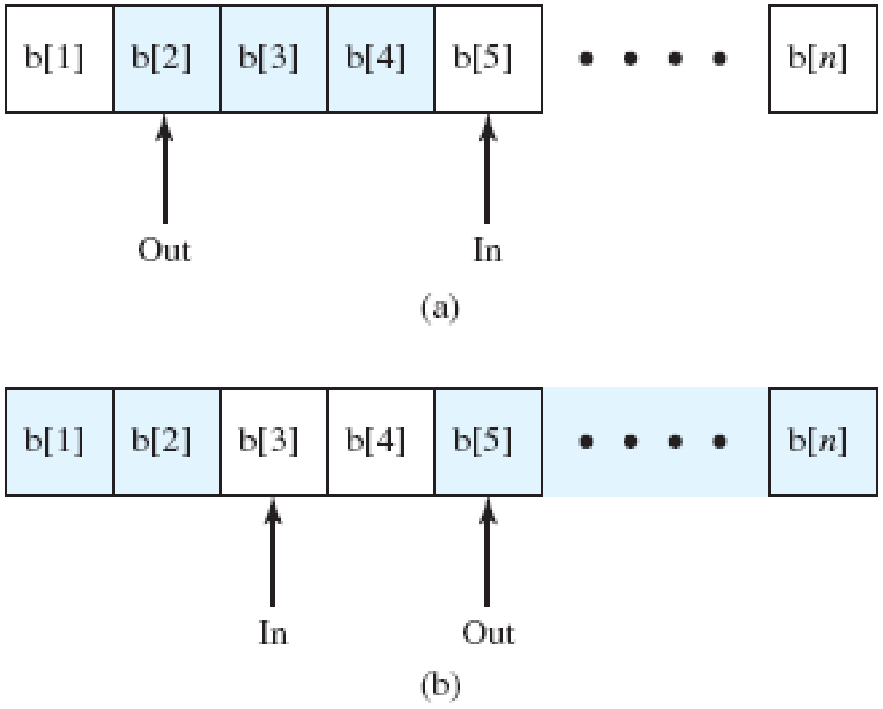
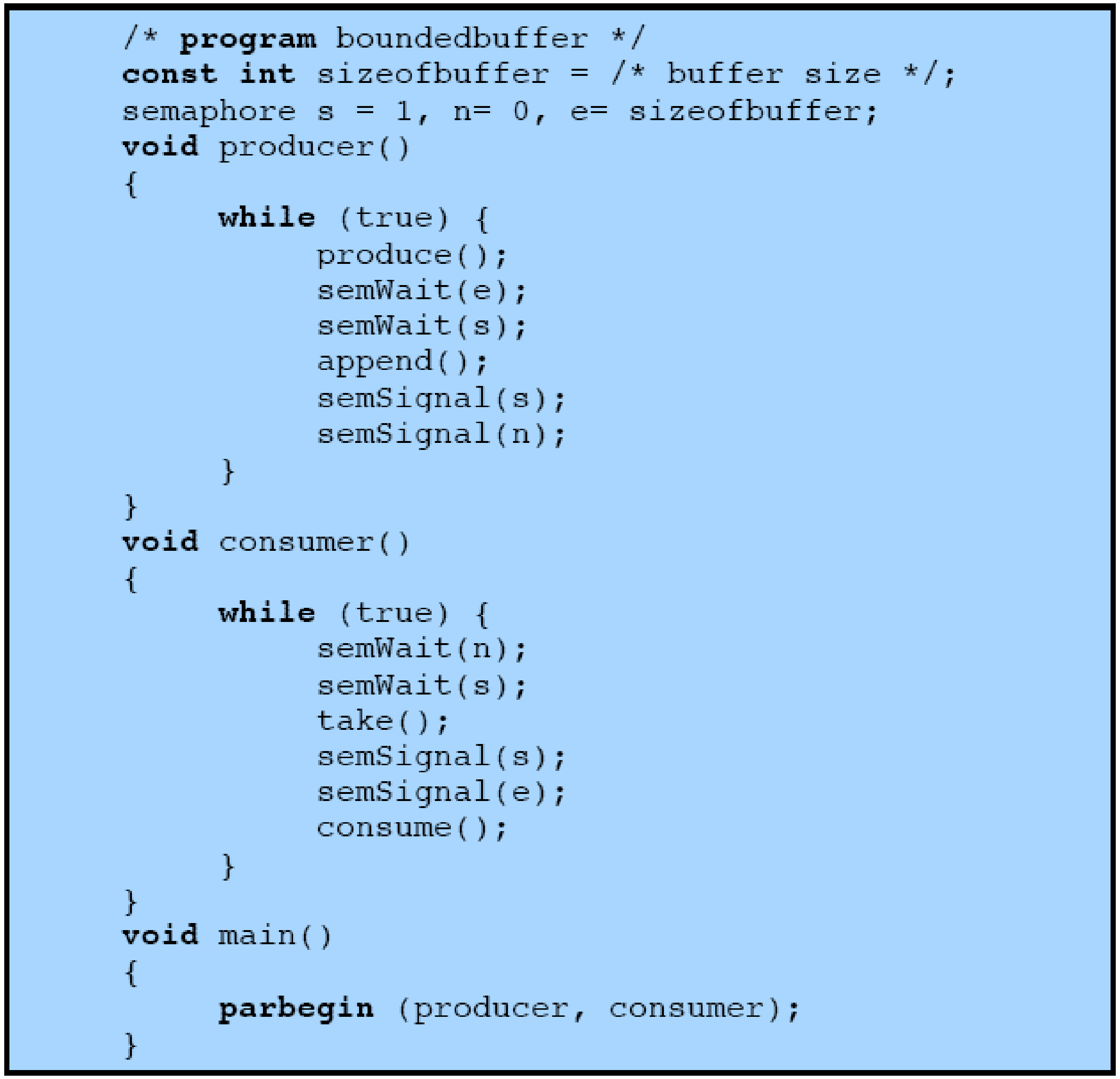


Figure 5.12 **Finite Circular Buffer for the Producer/Consumer Problem**

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);
}
```

Semaphores

s - ME
n – data count in buffer, to prevent consumer when buffer empty
e- space count in buffer , to prevent producer from inserting when buffer full

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 */
}	}

Implementation of semaphore

- It is imperative that the semWait and semSignal operations be implemented as atomic primitives.
- One obvious way is to implement them in hardware or firmware.
- Failing this, a variety of schemes have been suggested.
- The essence of the problem is one of mutual exclusion.
- Only one process may manipulats a semaphore with either semWait or semSignal.
- An alternative is to use one of the hardware-supported schemes. It involves busy waiting, but since he semWait/semSignal is short, the busy waiting is relatively minor issue.
- In a uniprocessor system it is possible to inhibit interrupts for the duration of a semWait or semSignal

Monitors

- Semaphores provide a primitive yet powerful and flexible tool for enforcing mutual exclusion and for coordinating processes.
- However, it may be difficult to produce a correct program using semaphore.
- The difficulty is that the semWait and semSignal are scattered throughout a program and it is not easy to see the overall effect of these operations on the semaphores they affect.

- 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.
- It allows a programmer to put a monitor lock on any pbject.

- A monitor is a software module consisting one or more procedures, an initialization sequence, and local data.
- **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

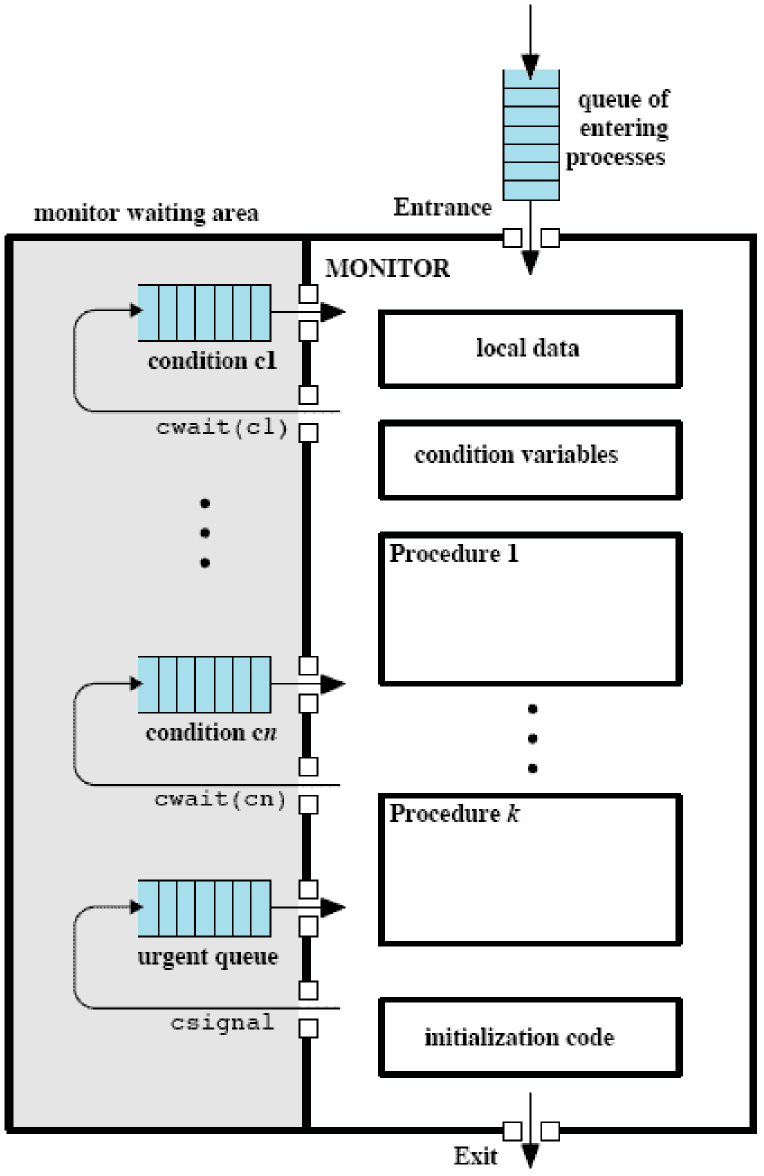
- By enforcing the discipline of one process at a time, the monitor is able to provide mutual exclusion.
- The data variables in the monitor can be accessed by only one process at a time.
- Thus, a shared data structure may be protected by placing it inside a monitor.
- To useful for concurrent processing, the monitor must include synchronization tools.

- For example, suppose a process invokes the monitor, and while in the monitor, must be blocked until some condition is satisfied.
- A facility is needed by which the process is not only blocked but releases the monitor so that some other process may enter it.
- Latter when the condition is satisfied, and the monitor is again available the process need to be resumed and allowed to reenter the monitor at the point of its suspension.

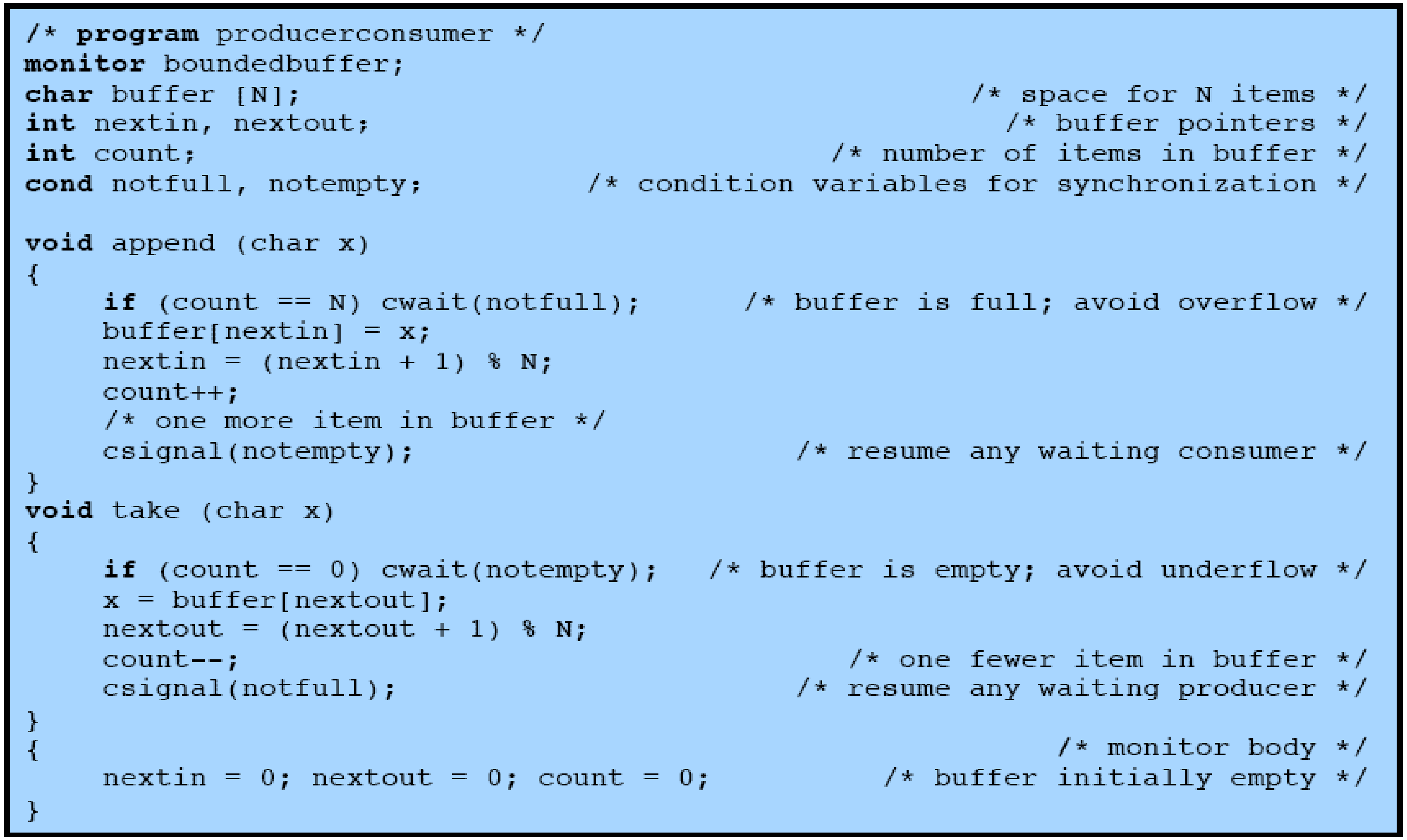
Synchronization

- Synchronization 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

Structure of a Monitor



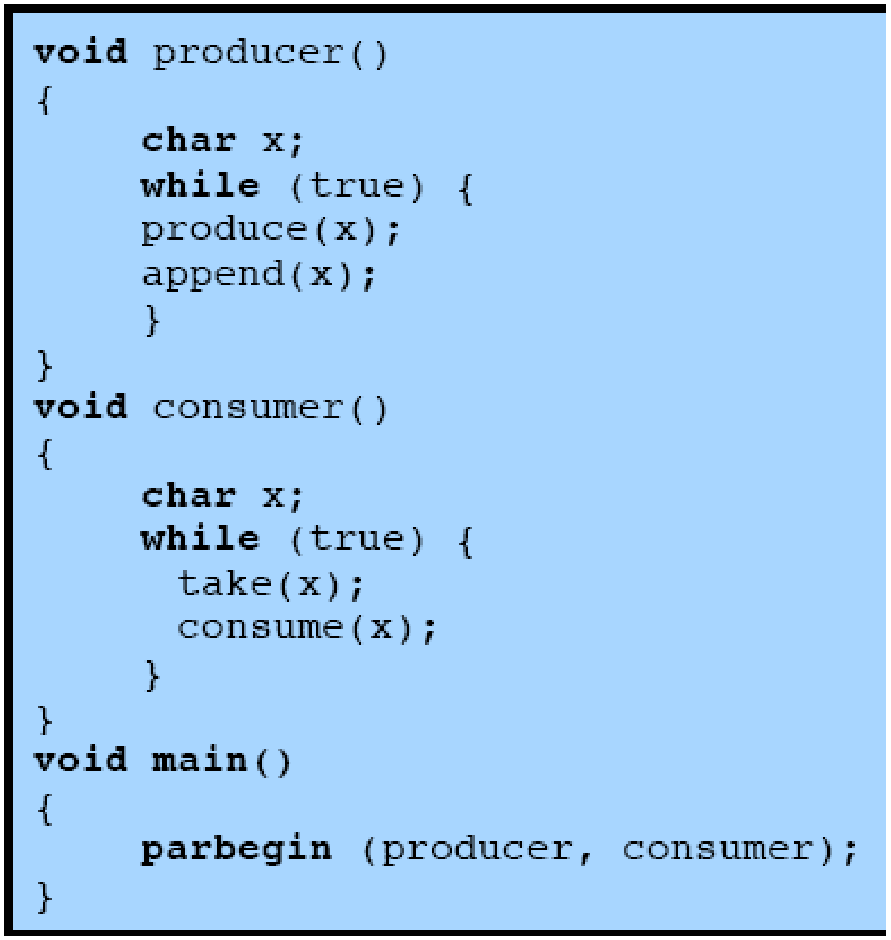
Bounded Buffer Solution Using Monitor



```
/* 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 */
}
```

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);
}
```

- In the case of monitors, the monitor construct itself enforces mutual exclusion.
- It is not possible for both producer and consumer simultaneously to enter the buffer.
- Cwaits are needed only to prevent processes to deposit into full or to consume from empty buffers.
- A process exits monitor immediately after executing the csignal.
- If a process fails to issue a csignal at the end of the procedure, then it is blocked to allow monitor to other processes

- If there are no processes waiting on a signal x, then csignal(x) has no effect.
- Like semaphore, if either of the csignal functions in the monitor are omitted, the processes are hung up permanently.
- The advantage of monitors over semaphore is that all of the synchronization functions are confined to monitor.
- So it is easy to verify that the synchronization has been correctly or not .
- With semaphone, resource access is correct only if all of the processes that access the resource are programmed correctly

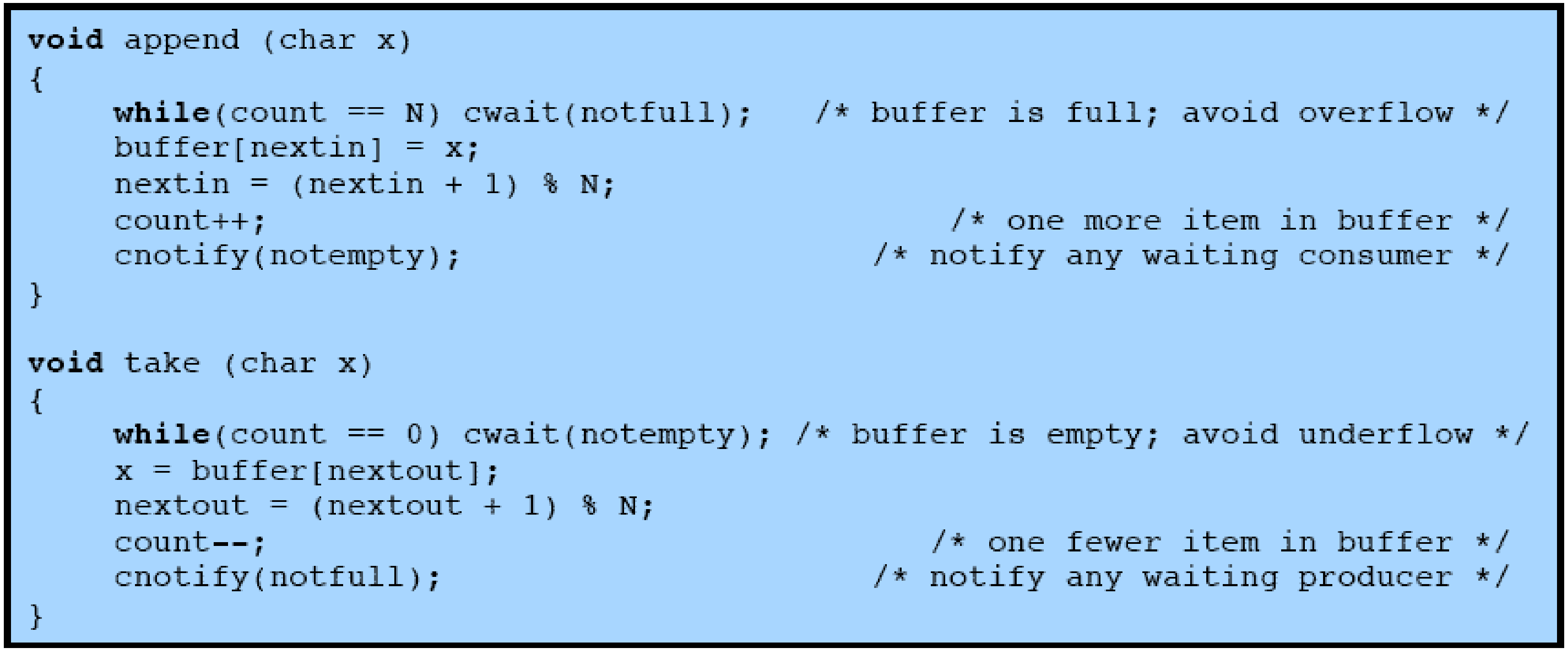
Alternate Model of Monitors with Notify and Broadcast

- Actual definition of monitor requires that if there is at least one process in a condition Q, a process from that Q runs immediately when another process issues a ***csignal.***
- Thus, the process issuing csignal must either immediately exit the monitor or be blocked.
- There are two drawbacks to this approach.
  - 1. If the process issuing csignal has not finished with the monitor, then two additional process switches are required.

- 2. Process scheduling associated with a signal must be perfectly reliable. When a csignal is issued, a process from the corresponding condition Q must be activated immediately and the scheduler must ensure that no other process enters the monitor before the activation.

- Lampson and Redell developed a different definition of monitors.
- The ***csignal*** premitive is replaced by ***cnotify***
- When a process executing in a monitor executes cnotify(x), it causes the x condition queue to be notified, but the signaling process continues to run.
- The result of the notification is that, the process at the head of the condition queue will be resumed at some convenient time in future when the monitor is available.
- But because there is no guarantee that some other process will not enter the monitor before this process, the waiting process must recheck the condition

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

- A refinement to be made to cnotify- a timer may be associated with each condition primitive/variable.
- A process that has been waiting for the maximum timeout interval will be placed into the ready queue regardless of whether the condition has been notified.
- The timeout prevents the indefinite starvation of a process.

- With the rule that a process is notified rather than forcibly reactivated, it is possible to add a ***cbroadcast*** primitive.
- The broadcast causes all processes waiting on a condition to be placed in a Ready state.
- This is convenient when a process does not know how many processes need to reactivated.

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

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)

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)

Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- Allows for tight synchronization between processes.

Non-blocking Send

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

Addressing

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

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

Indirect Process Communication

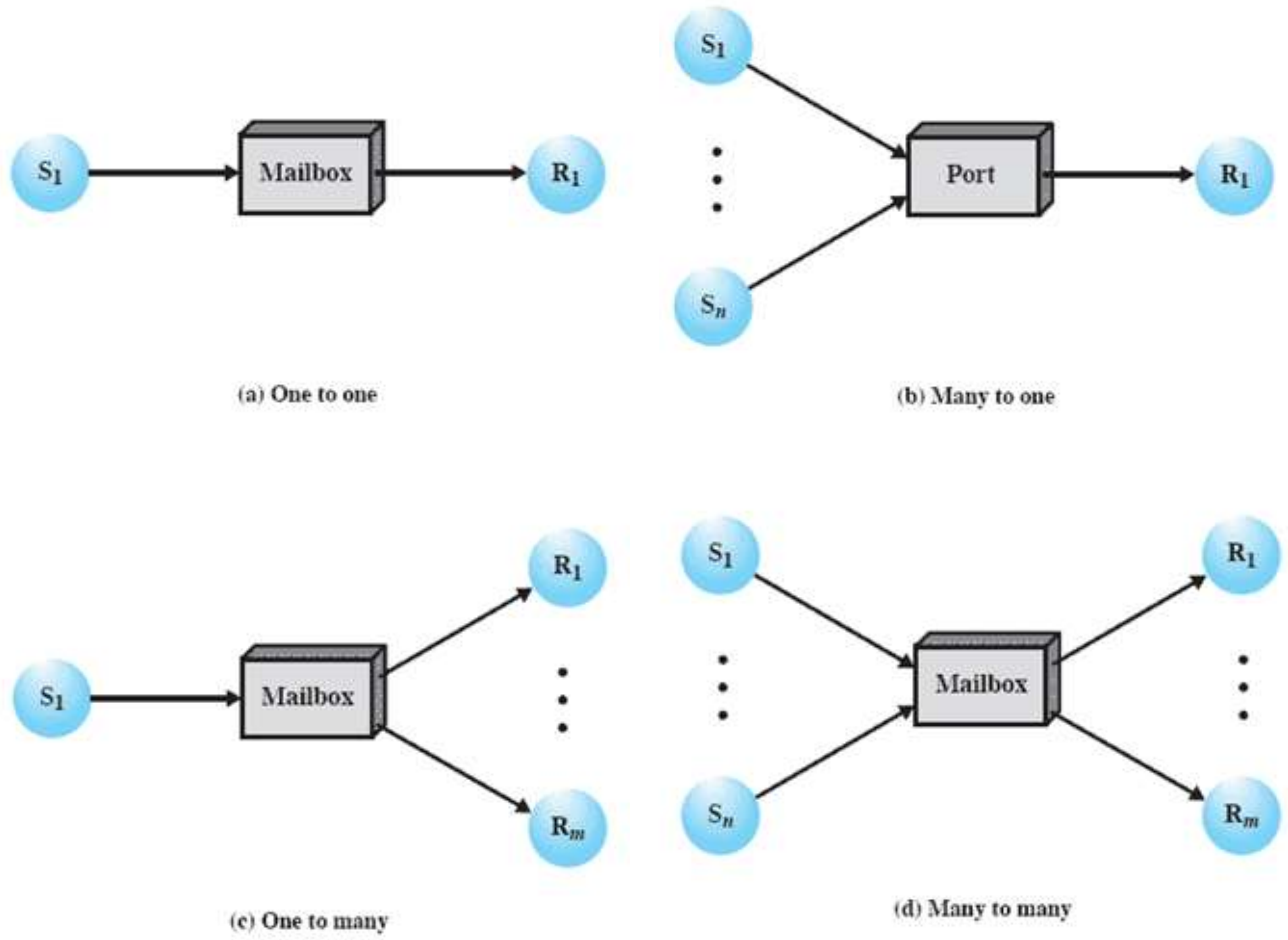


Figure 5.18 Indirect Process Communication

General Message Format

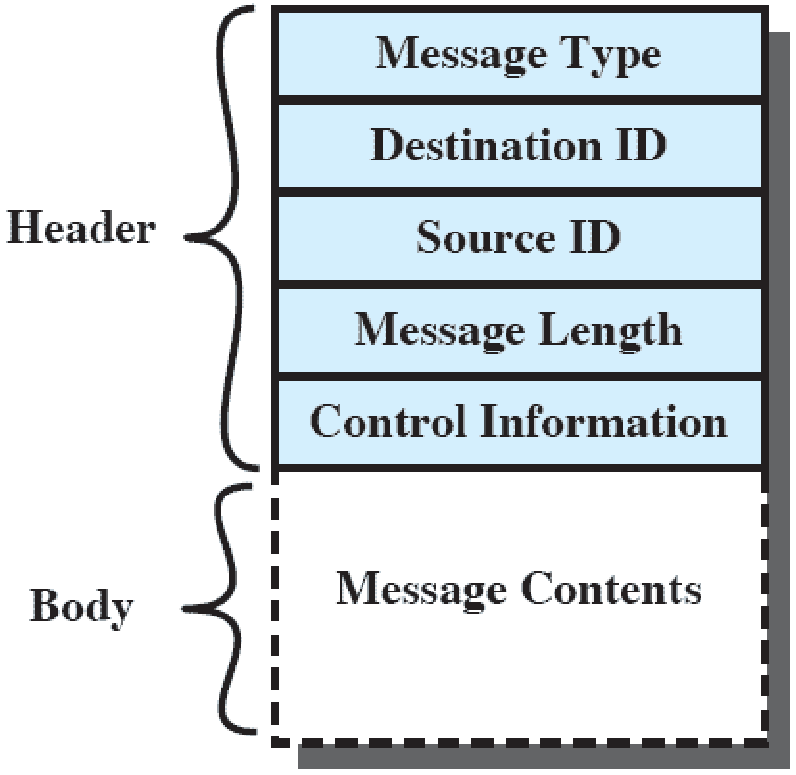


Figure 5.19 General Message Format

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

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);
}
```

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.

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);
}
```

Wsem – to prevent Writers when readers are Reading

x – To enforce ME among readers in accessing readcount

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);
    }
}
```

Writers have Priority Without Z Semaphore

```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {

          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);

     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
```

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);
}
```

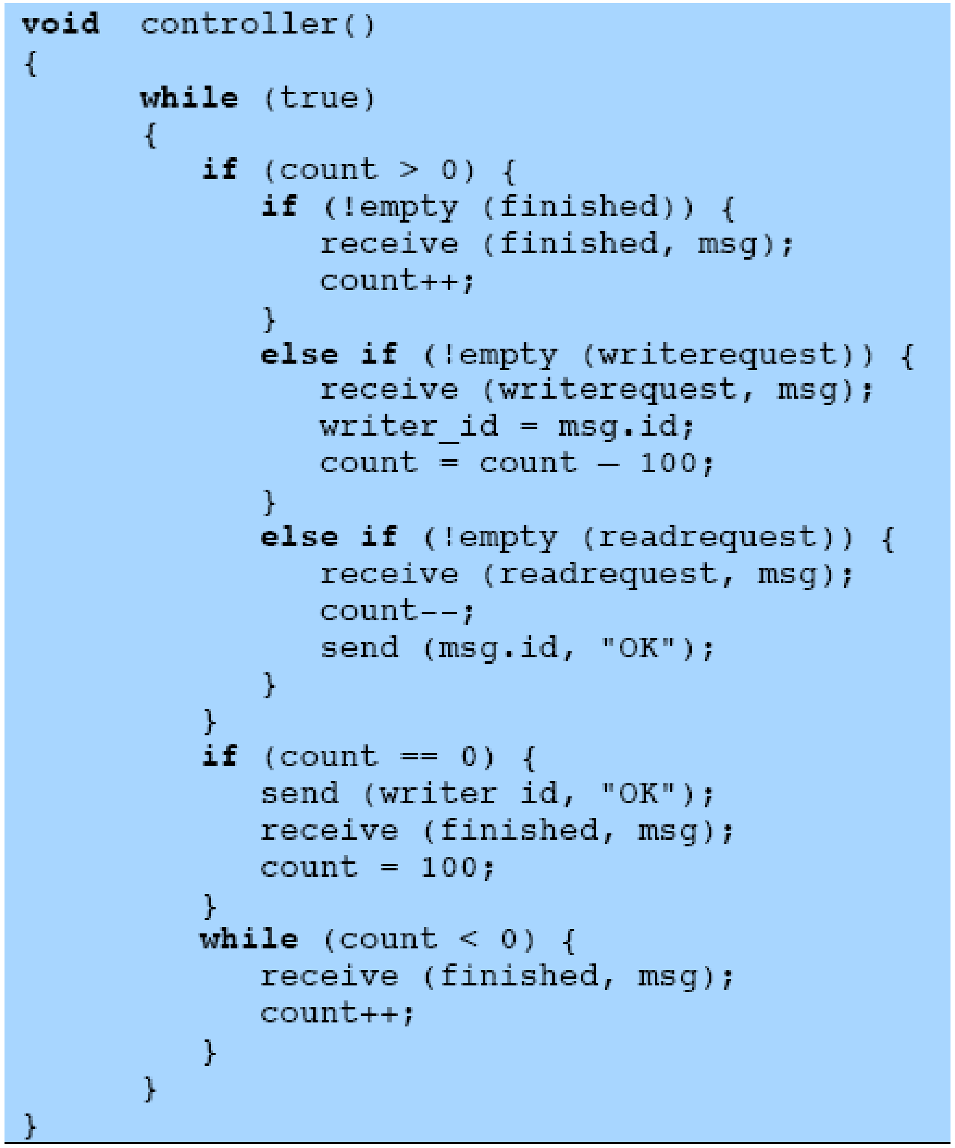
Readers only in the system	• *wsem* set • no queues
Writers only in the system	• *wsem* and *rsem* set • writers queue on *wsem*
Both readers and writers with read first	• *wsem* set by reader • *rsem* set by writer • all writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*
Both readers and writers with write first	• *wsem* set by writer • *rsem* set by writer • writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*

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++;
         }
      }
}
```

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++;
         }
      }
}
```