Operating Systems: Internals and Design Principles

Chapter 5
Concurrency: Mutual Exclusion and Synchronization

Ninth Edition
By William Stallings

Multiple Processes

- Operating System design is 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
        - The recent proliferation of clusters is a prime example of this type of system

Concurrency
Arises in Three Different Contexts:

Multiple Applications

Invented to allow processing time to be shared among active applications

Structured Applications

Extension of modular design and structured programming

Operating System Structure

OS themselves implemented as a set of processes or threads

		Table 5.1
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.	**Some Key Terms Related to Concurrency**
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.	

Mutual Exclusion: Software Approaches

- Software approaches can be implemented for concurrent processes that execute on a single-processor or a multiprocessor machine with shared main memory

- These approaches usually assume elementary mutual exclusion at the memory access level
  - Simultaneous accesses (reading and/or writing) to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time
  - Beyond this, no support in the hardware, operating system, or programming language is assumed

- Dijkstra reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker
  - Following Dijkstra, we develop the solution in stages
  - This approach has the advantage if illustrating many of the common bugs encountered in developing concurrent programs

```
        /* PROCESS 0 /*                    /* PROCESS 1 */

        •                                  •
        •                                  •
        while (turn != 0)                  while (turn != 1)
            /* do nothing */ ;                 /* do nothing */;
        /* critical section*/;             /* critical section*/;
        turn = 1;                          turn = 0;
        •                                  •
```

(a) First attempt

```
        /* PROCESS 0 */                    /* PROCESS 1 */

        •                                  •
        •                                  •
        while (flag[1])                    while (flag[0])
            /* do nothing */;                  /* do nothing */;
        flag[0] = true;                    flag[1] = true;
        /*critical section*/;              /* critical section*/;
        flag[0] = false;                   flag[1] = false;
        •                                  •
```

(b) Second attempt

Figure 5.1 Mutual Exclusion Attempts (page 1)

```
/* PROCESS 0 */                 /* PROCESS 1 */

    .                               .
    .                               .
flag[0] = true;                 flag[1] = true;
while (flag[1])                 while (flag[0])
    /* do nothing */;               /* do nothing */;
/* critical section*/;          /* critical section*/;
flag[0] = false;                flag[1] = false;
    .                               .
```

(c) Third attempt

```
/* PROCESS 0 */                 /* PROCESS 1 */

    .                               .
    .                               .
flag[0] = true;                 flag[1] = true;
while (flag[1]) {               while (flag[0]) {
    flag[0] = false;                flag[1] = false;
    /*delay */;                     /*delay */;
    flag[0] = true;                 flag[1] = true;
}                               }
/*critical section*/;           /* critical section*/;
flag[0] = false;                flag[1] = false;
    .                               .
```

(d) Fourth attempt

Figure 5.1 Mutual Exclusion Attempts (page 2)

```
boolean flag [2];
int turn;
void P0()
{
    while (true) {
        flag [0] = true;
        while (flag [1]) {
            if (turn == 1) {
                flag [0] = false;
                while (turn == 1) /* do nothing
*/;
                flag [0] = true;
            }
        }
        /* critical section  */;
        turn = 1;
        flag [0] = false;
        /* remainder    */;
    }
}
void P1( )
{
    while (true) {
        flag [1] = true;
        while (flag [0]) {
            if (turn == 0) {
                flag [1] = false;
                while (turn == 0) /* do nothing
*/;
                flag [1] = true;
            }
        }
        /* critical section   */;
        turn = 0;
        flag [1] = false;
        /* remainder    */;
    }
}
void main ()
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

Figure 5.2   Dekker's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
    while (true) {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1) /* do nothing */;
        /* critical section */;
        flag [0] = false;
        /* remainder */;
    }
}
void P1()
{
    while (true) {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0) /* do nothing */;
        /* critical section */;
        flag [1] = false;
        /* remainder */
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
```

Figure 5.3 Peterson's Algorithm for Two Processes

Principles of Concurrency

- Interleaving and overlapping
    - Can be viewed as examples of concurrent processing
    - Both present the same problems

- Uniprocessor – the relative speed of execution of processes cannot be predicted
    - Depends on activities of other processes
    - The way the OS handles interrupts
    - Scheduling policies of the OS

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 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 unintended interference by other processes

And The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes

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

Table 5.2

Process Interaction

Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - For example: I/O devices, memory, processor time, clock

In the case of competing processes three control problems must be faced:

- **The need for mutual exclusion**
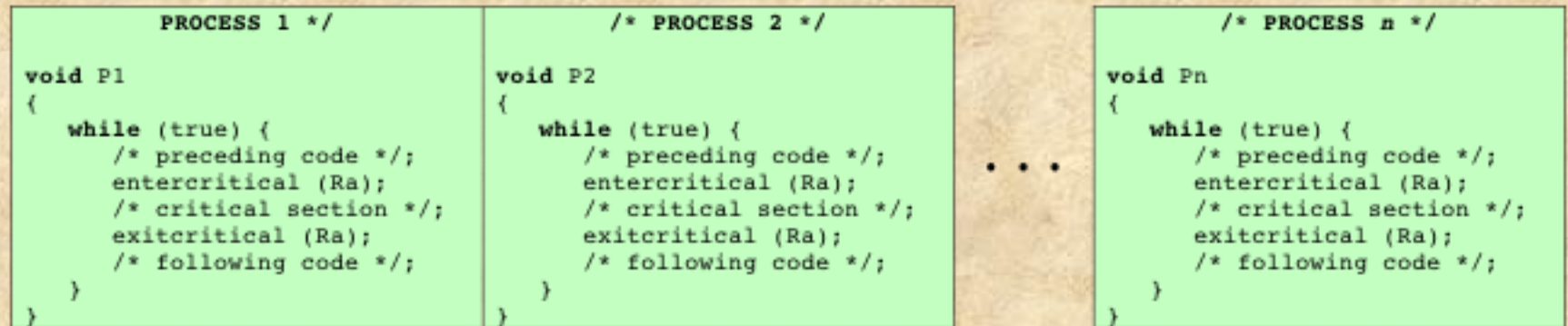- **Deadlock**
- **Starvation**

```
            PROCESS 1 */              /* PROCESS 2 */              /* PROCESS n */

void P1                       void P2                       void Pn
{                             {                             {
   while (true) {                while (true) {                while (true) {
      /* preceding code */;         /* preceding code */;         /* preceding code */;
      entercritical (Ra);           entercritical (Ra);           entercritical (Ra);
      /* critical section */;       /* critical section */;       /* critical section */;
      exitcritical (Ra);            exitcritical (Ra);            exitcritical (Ra);
      /* following code */;         /* following code */;         /* following code */;
   }                             }                             }
}                             }                             }
```

Figure 5.4 Illustration of Mutual Exclusion

Cooperation Among Processes by Sharing

Covers processes that interact with other processes without being explicitly aware of them

Processes may use and update the shared data without reference to other processes, but know that other processes may have access to the same data

Thus the processes must cooperate to ensure that the data they share are properly managed

The control mechanisms must ensure the integrity of the shared data

Because data are held on resources (devices, memory), the control problems of mutual exclusion, deadlock, and starvation are again present

- The only difference is that data items may be accessed in two different modes, reading and writing, and only writing operations must be mutually exclusive

Cooperation Among Processes by Communication

- The various processes participate in a common effort that links all of the processes

- The communication provides a way to synchronize, or coordinate, the various activities

- Typically, communication can be characterized as consisting of messages of some sort

- Primitives for sending and receiving messages may be provided as part of the programming language or provided by the OS kernel

- Mutual exclusion is not a control requirement for this sort of cooperation

- The problems of deadlock and starvation are still present

Requirements for Mutual Exclusion

■ Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

- ■ Mutual exclusion must be enforced: only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object
- ■ A process that halts must do so without interfering with other processes
- ■ It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation
- ■ When no process is in a critical section, any process that request entry to its critical section must be permitted to enter without delay
- ■ 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

- In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved
- A process will continue to run until it invokes an OS service or until it is interrupted
- Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted
- This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts

Disadvantages:

- The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes
- This approach will not work in a multiprocessor architecture

Mutual Exclusion: Hardware Support

- Compare&Swap Instruction
  - Also called a "compare and exchange instruction"
  - A **compare** is made between a memory value and a test value
  - If the values are the same a **swap** occurs
  - Carried out atomically (not subject to interruption)

```
/* 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

```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
  while (true) {
    int keyi = 1;
    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

Figure 5.5   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, the selection of a waiting process is arbitrary; some process could indefinitely be denied access

Deadlock is possible

`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.

Table 5.3

Common

Concurrency

Mechanisms

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) A semaphore may be initialized to a nonnegative integer value

2) The semWait operation decrements the semaphore value

3) The semSignal operation increments the semaphore 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

```
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 5.6 A Definition of 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 5.7 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
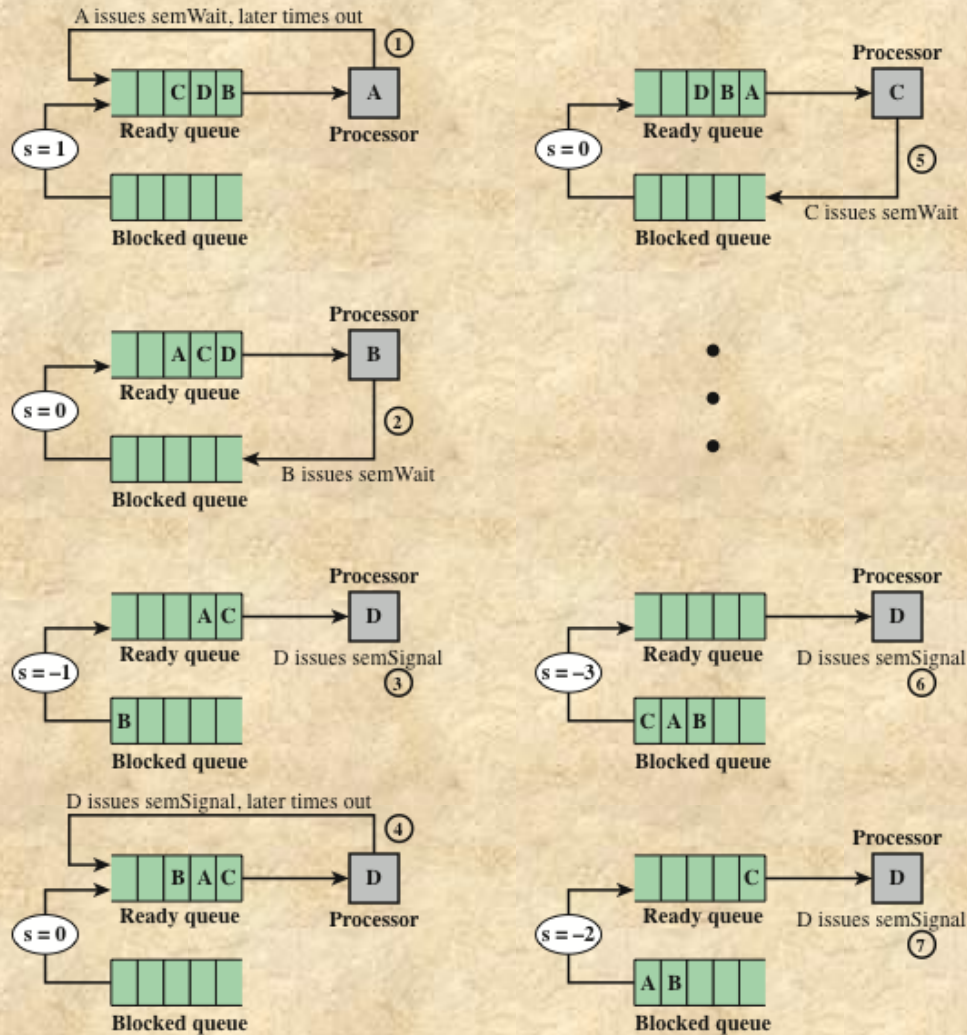
Figure 5.8 Example of Semaphore Mechanism

```
/* 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 5.9 Mutual Exclusion Using Semaphores
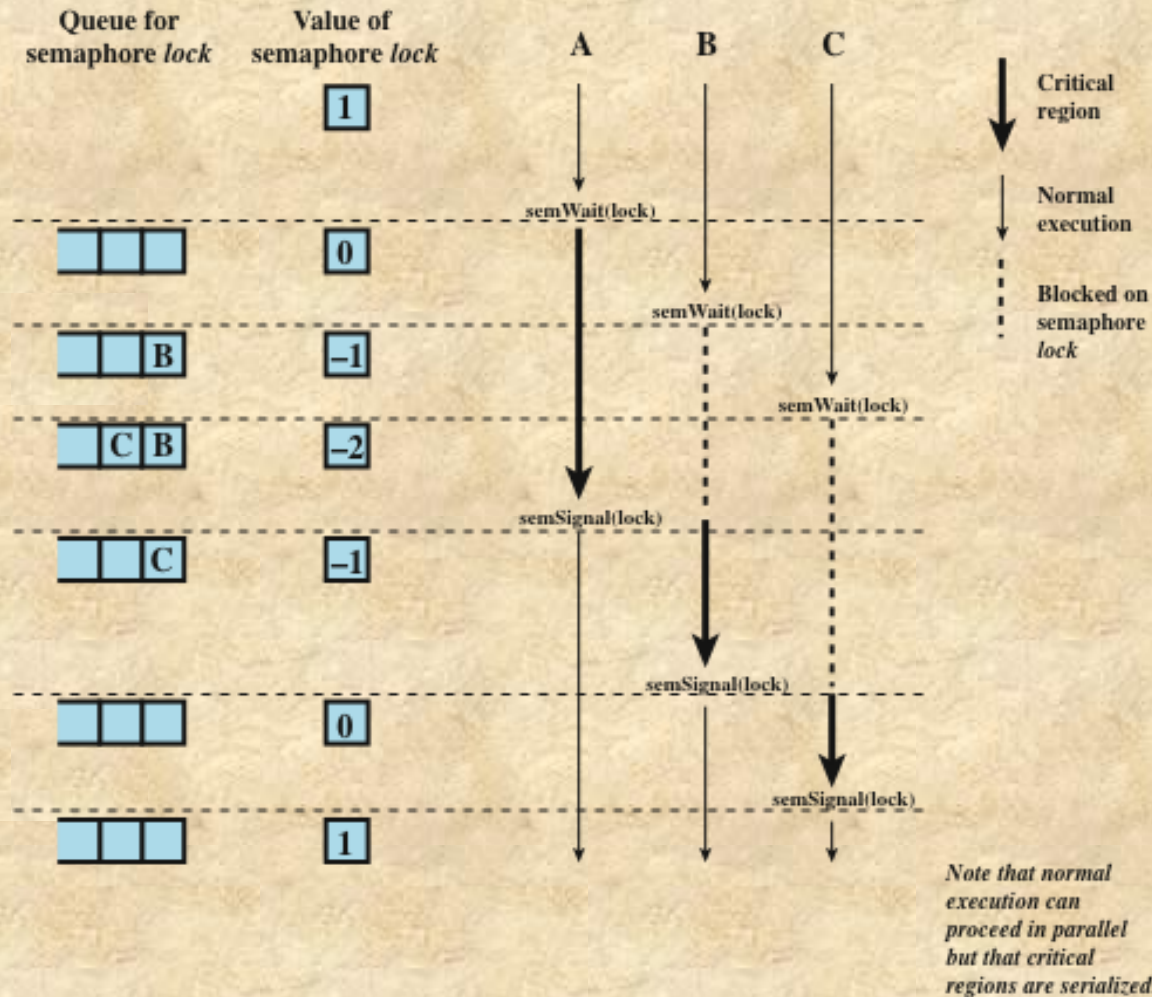
Figure 5.10 Processes Accessing Shared Data Protected by a Semaphore

Producer/Consumer Problem

General Statement:	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 a time
	Only one producer or consumer may access the buffer at any one time

The Problem:

Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer
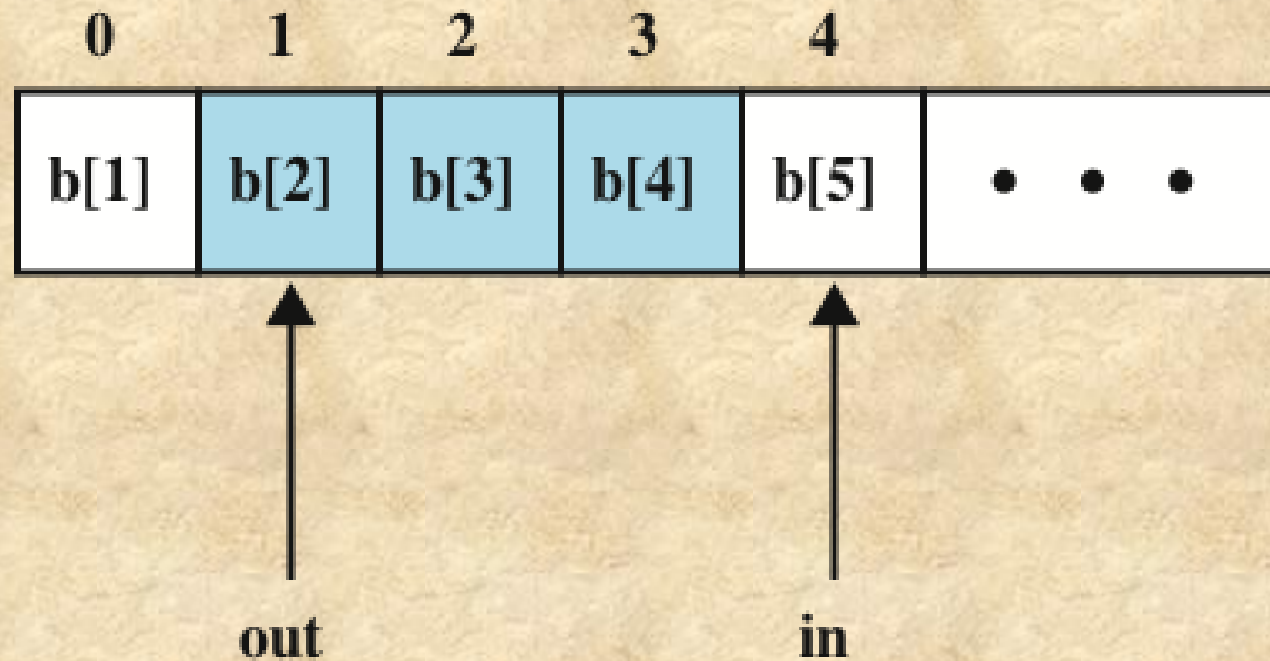
Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

```
/* 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);
}
```

Figure 5.12 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Table 5.4
Possible Scenario for the Program of Figure 5.12

	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		`semSignalB(s)`	1	−1	0

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

```
/* 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);
}
```

Figure 5.13   A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary 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);
}
```
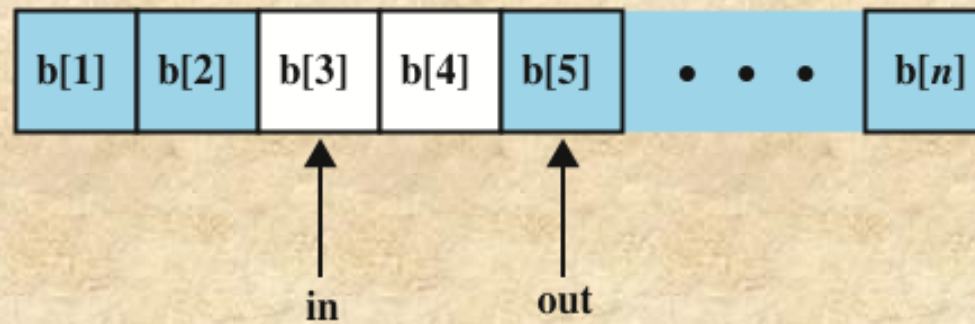
Figure 5.14   A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores
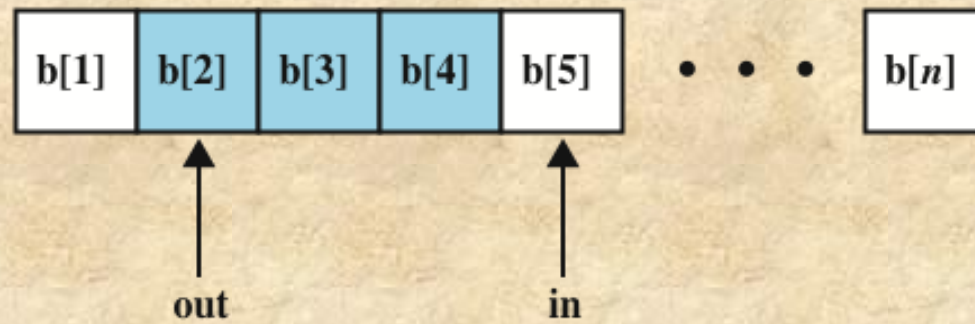
Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

```
/* 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 5.16   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

- Another alternative is to use one of the hardware-supported schemes for mutual exclusion

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0)
*/;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
        allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(a) Compare and Swap Instruction   (b) Interrupts

Figure 5.17   Two Possible Implementations of Semaphores

Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control

- Implemented in a number of programming languages
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, 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

- A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are a special data type in monitors which 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
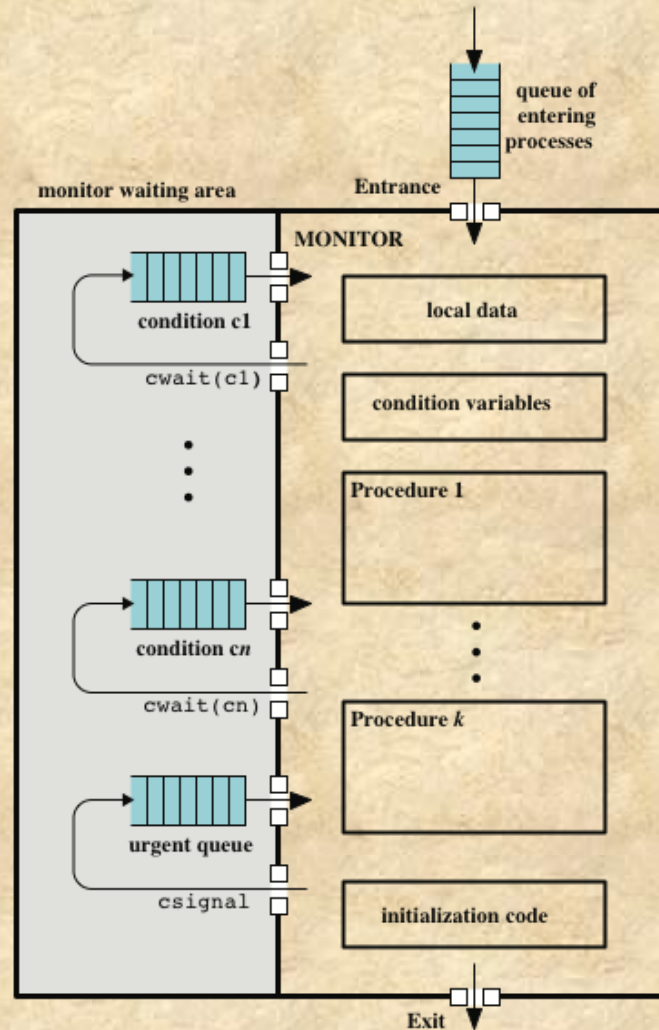
Figure 5.18 Structure of a Monitor

```
/* 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 5.19 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a 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 5.20 Bounded Buffer Monitor Code for Mesa Monitor

Message Passing

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

Synchronization	Communication
• To enforce mutual exclusion	• 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*

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

Queueing 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

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

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

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
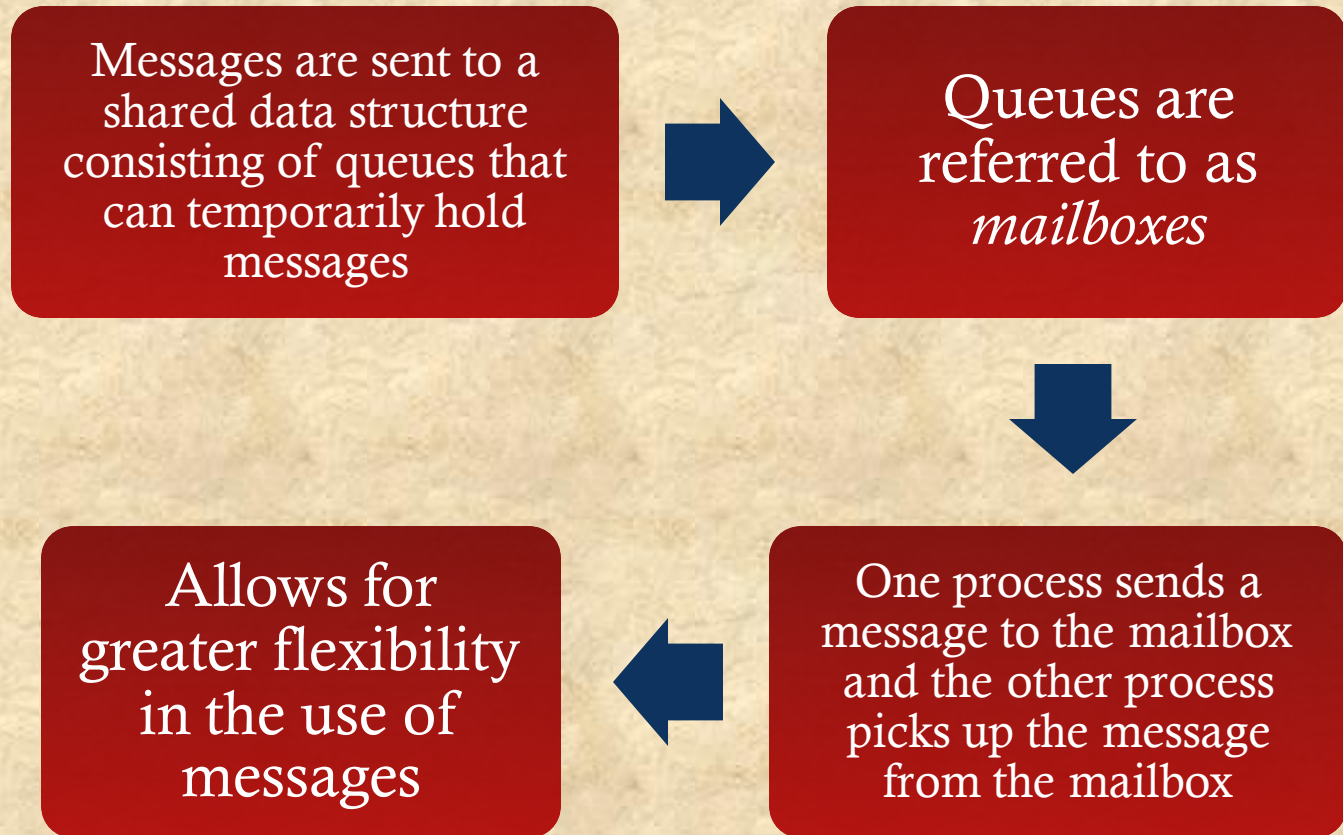
- Neither party is required to wait

Addressing

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

  Direct addressing

  Indirect addressing
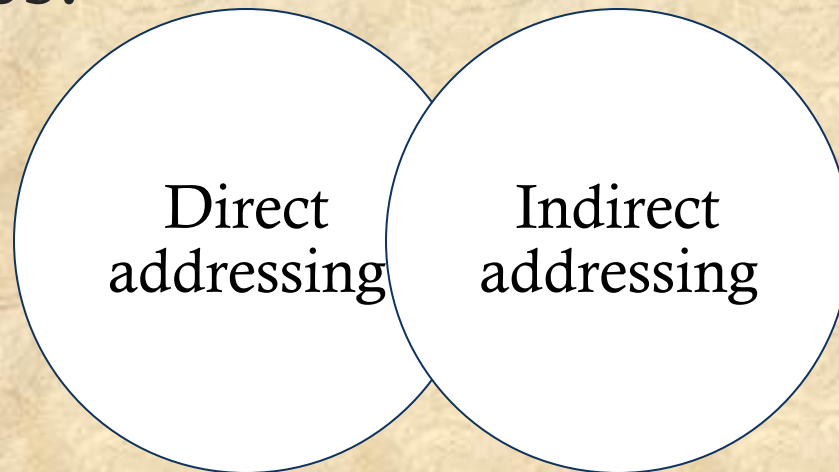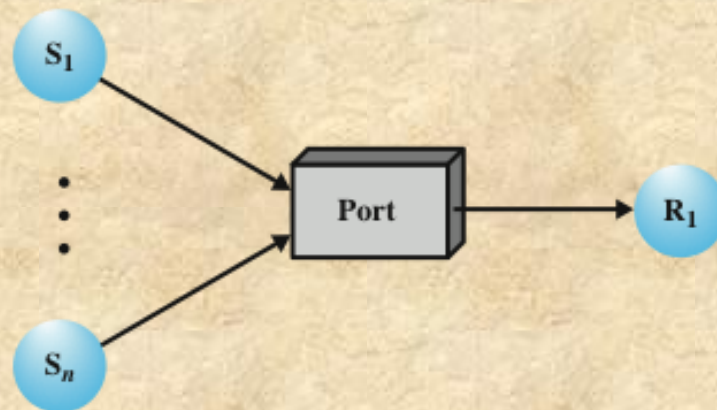
Direct Addressing

- 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
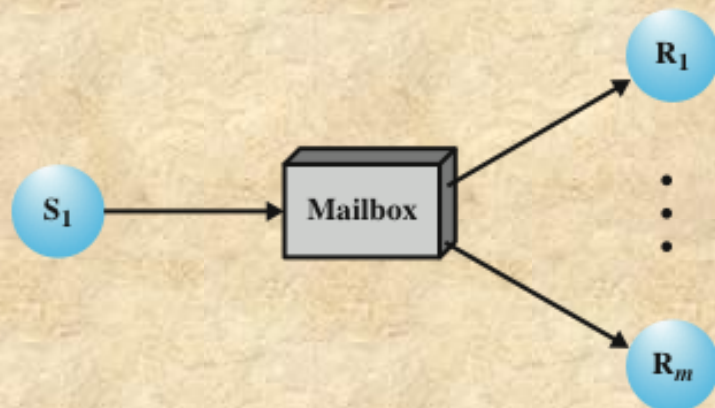
Indirect Addressing

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
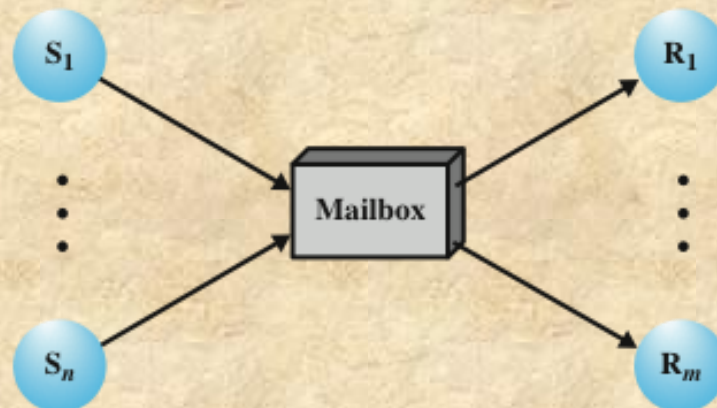
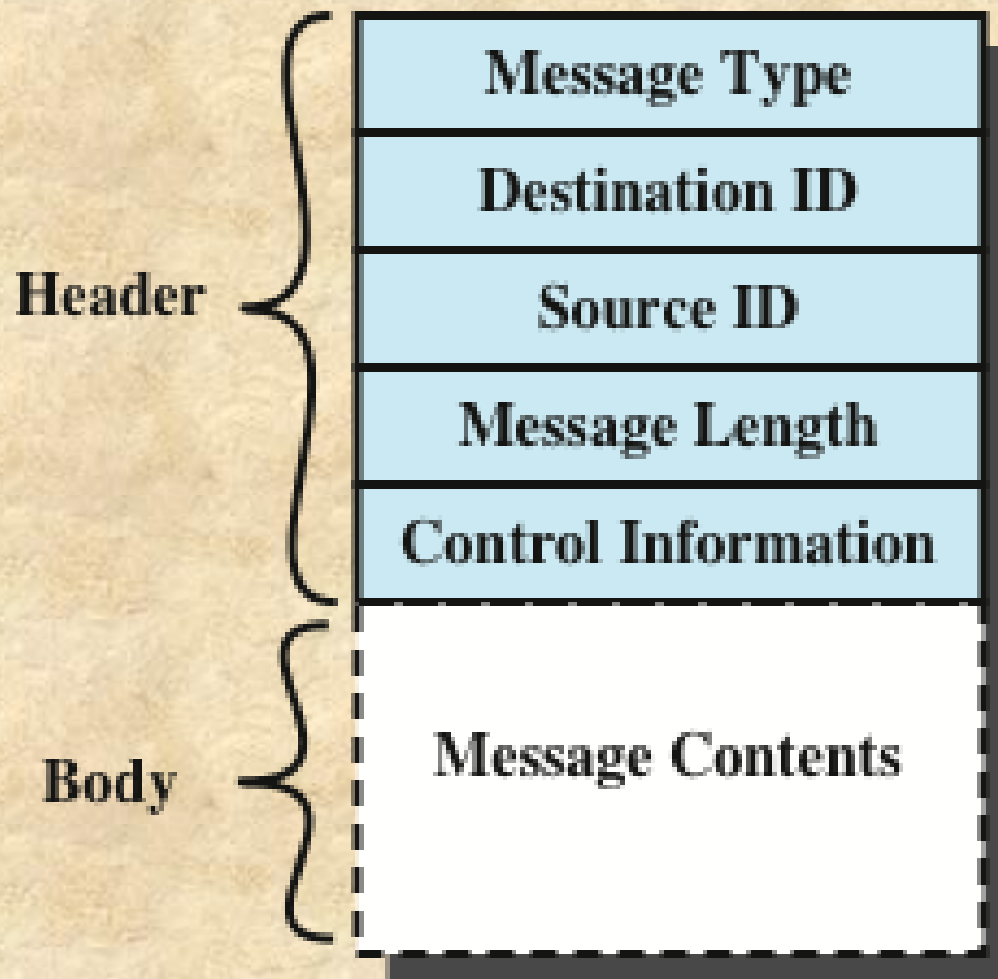Figure 5.21   Indirect Process Communication

Figure 5.22   General Message Format

Queueing Discipline

- The simplest queueing discipline is first-in-first-out
  - This may not be sufficient if some message are more urgent than others

- Other alternatives are:
  - To allow the specifying of message priority, on the basis of message type or by designation by the sender
  - To allow the receiver to inspect the message queue and select which message to receive next

```
/* 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 5.23 Mutual Exclusion Using 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);
}
```

Figure 5.24   A Solution to the Bounded-Buffer Producer/Consumer Problem
Using Messages

Readers/Writers Problem

- A data area is shared among many processes
  - Some processes only read the data area, (readers) and some only write to the data area (writers)

- Conditions that must be satisfied:
  - Any number of readers may simultaneously read the file
  - Only one writer at a time may write to the file
  - If a writer is writing to the file, no reader may read it

```
/* 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);
}
```

Figure 5.25   A Solution to the Readers/Writers Problem Using Semaphores: Readers 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);
    }
}
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);
}
```

Figure 5.26   A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority

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*

Table 5.6
State of the Process Queues for Program of Figure 5.26

```
void reader(int i)                          void  controller()
{                                           {
    message rmsg;                               while (true)
        while (true) {                          {
            rmsg = i;                               if (count > 0) {
            send (readrequest, rmsg);                   if (!empty (finished)) {
            receive (mbox[i], rmsg);                        receive (finished, msg);
            READUNIT ();                                    count++;
            rmsg = i;                                   }
            send (finished, rmsg);                      else if (!empty (writerequest)) {
        }                                                   receive (writerequest, msg);
 }                                                          writer_id = msg.id;
void writer(int j)                                          count = count - 100;
{                                                       }
    message rmsg;                                       else if (!empty (readrequest)) {
    while(true) {                                           receive (readrequest, msg);
        rmsg = j;                                           count--;
        send (writerequest, rmsg);                          send (msg.id, "OK");
        receive (mbox[j], rmsg);                        }
        WRITEUNIT ();                               }
        rmsg = j;                                   if (count == 0) {
        send (finished, rmsg);                          send (writer_id, "OK");
    }                                                   receive (finished, msg);
}                                                       count = 100;
                                                    }
                                                    while (count < 0) {
                                                        receive (finished, msg);
                                                        count++;
                                                    }
                                                }
                                            }
```

Figure 5.27   A Solution to the Readers/Writers Problem Using Message Passing

Summary

- Mutual exclusion: software approaches
  - Dekker's algorithm
  - Peterson's algorithm
- Principles of concurrency
  - Race condition
  - OS concerns
  - Process interaction
  - Requirements for mutual exclusion
- Mutual exclusion: hardware support
  - Interrupt disabling
  - Special machine instructions
- Semaphores
  - Mutual exclusion
  - Producer/consumer problem
  - Implementation of semaphores

- Monitors
  - Monitor with signal
  - Alternate model of monitors with notify and broadcast
- Message passing
  - Synchronization
  - Addressing
  - Message format
  - Queueing discipline
  - Mutual exclusion
- Readers/writers problem
  - Readers have priority
  - Writers have priority