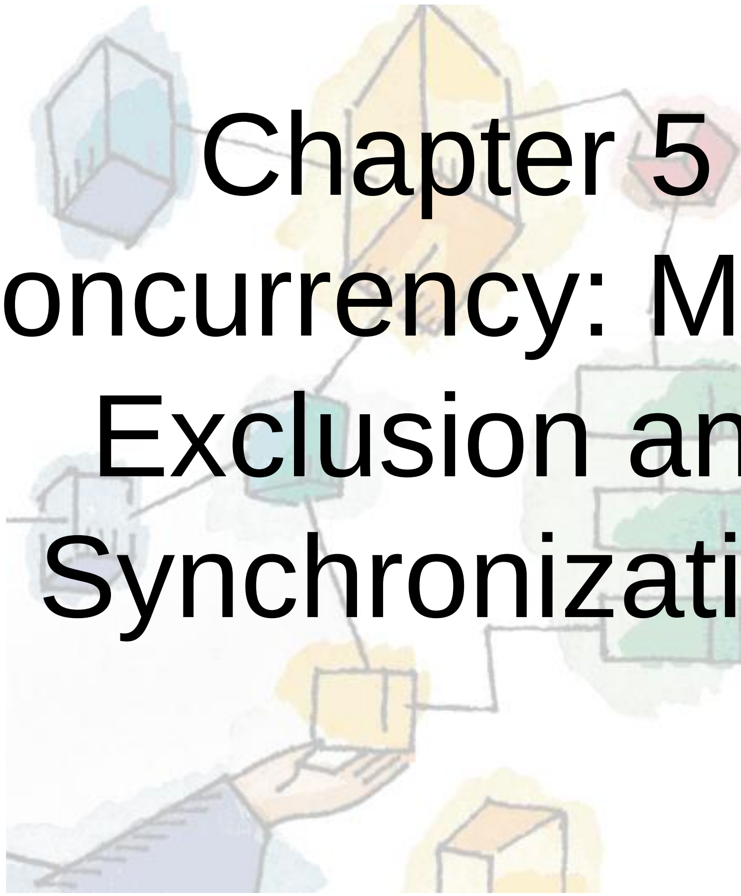
Operating Systems:
Internals and Design Principles, 6/E
William Stallings



Chapter 5 Concurrency: Mutual Exclusion and Synchronization

Roadmap

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

Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes / threads
  - Multiprogramming (management of multiple processes within a uniprocessor system)
  - Multiprocessing (management of multiple processes within a multiprocessor)
  - Distributed Processing (The management of multiple processes executing on multiple, distributed computer systems.)
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

Concurrency **design issues**, including

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

Concurrency

Concurrency arises in:

- Multiple applications
  - Sharing time (Multiprogramming)
- Structured applications
  - Extension of modular design, some applications can be effectively programmed as a set of concurrent processes.
- Operating system structure
  - OS themselves implemented as a set of processes or threads

Key Terms

Table 5.1 Some Key Terms Related to Concurrency

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

Interleaving and Overlapping Processes

- Earlier we saw that processes may be interleaved on uniprocessors

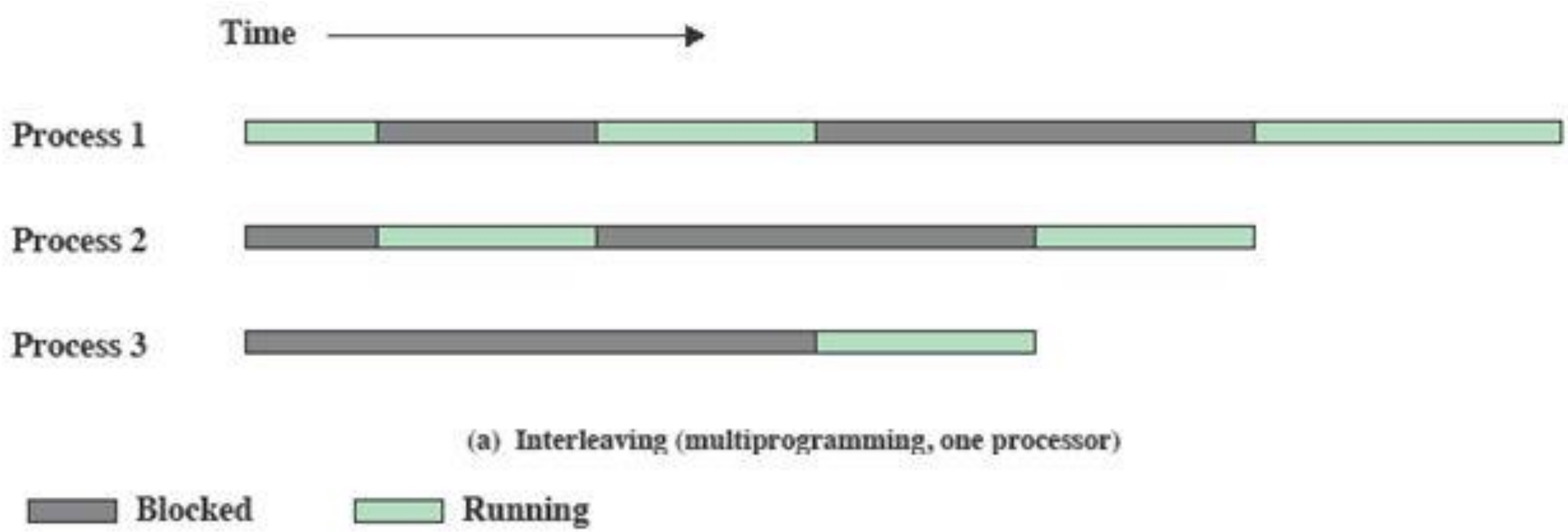


Figure 2.12 Multiprogramming and Multiprocessing

Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors

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

Figure 2.12 Multiprogramming and Multiprocessing

Difficulties of Concurrency

- Sharing of global resources
  - Writing a shared variable: the order of writes is important
  - Incomplete writes a major problem
- Optimally managing the allocation of resources
- Difficult to locate programming errors as results are not deterministic and reproducible.

A Simple Example

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

A Simple Example: On a Multiprocessor

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

Enforce Single Access

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

Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution-
    - the “loser” of the race is the process that updates last and will determine the final value of the variable

Example

consider two process, P1 and P2, sharing global variables b and c , having values

b = 1 and c = 2 .

- P1 executes b = b + c ,
- P2 executes c = b + c .
- If P1 executes its assignment statement first, then the final values are b = 3 and c = 5 . If P2 executes its assignment statement first, then the final values are b = 4 and c = 3 .

Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes (PCB)
  - Allocate and de-allocate resources (Processor time, Memory, files and I/O Devices)
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

Process Interaction

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected
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
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

Competition among Processes for Resources

Three main control problems:

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

Requirements for Mutual Exclusion

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

Requirements for Mutual Exclusion

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

Roadmap

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

Disabling Interrupts

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

Pseudo-Code

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

Special Machine Instructions

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

Compare&Swap Instruction

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

The entire compare&swap function is carried out atomically;

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

Mutual Exclusion (fig 5.2)

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

}
```

(a) Compare and swap instruction

Exchange instruction

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

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

Exchange Instruction (fig 5.2)

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

(b) Exchange instruction

PROPERTIES OF THE MACHINE-INSTRUCTION APPROACH

Hardware Mutual Exclusion:

Advantages

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

Hardware Mutual Exclusion: Disadvantages

- Busy-waiting (While a process is waiting for access to a critical section, it continues to consume processor time)consumes
- 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

Deadlock is possible (Example (on a uniprocessor).

- Process P1 executes the special instruction (e.g., compare&swap, exchange) and enters its critical section.
- P1 is then interrupted to give the processor to P2, which has higher priority.
- If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism.
  - Thus it will go into a busy waiting loop.
- However, P1 will never be dispatched because it is of lower priority than another ready process, P2.

Mutual Exclusion: software Support

boolean flag[2];

int turn

Both are global variables

Decker's Algorithm (boolean flag[2]; int turn;)`

```
void P0(){
while(TRUE) //main while loop
{
 flag[0] = TRUE; //turn on your own flag
 while(flag[1]) //test for P1's flag
{
 If (turn == 1) //if P1 is active then
        {
        flag[0] = FALSE; //turn off your own flag
        while (turn == 1) //wait for P1 to release
        /*do nothing, stay here*/;
        flag[0] = TRUE; //you got the turn now
        }
        /* CS */;
        turn = 1; //give turn to other guy
        flag[0] = FALSE; //turn off your flag
        /* remainder */
} //end of test for P1
} //end of main while
} //end of P0
```

```
void P1()
{
while(TRUE) //main while loop
{
 flag[1] = TRUE; //turn on your own flag
 while(flag[0]) //test for P0's flag
{
        If (turn == 0) //if P0 is active then
        {
        flag[1] = FALSE; //turn off your own flag
        while (turn == 0) //wait for P0 to release
        /*do nothing, stay here*/;
        flag[1] = TRUE; //you got the turn now
        }
        /* CS */;
        turn = 0; //give turn to other guy
        flag[1] = FALSE; //turn off your flag
        /* remainder */
} //end of test for P0
} //end of main while
} //end of P1
void main()
{
 flag[0] = FALSE; flag[1] = FALSE;
 turn = 0;  parabegin (P0, P1);
}
```

eterson's Algorithm(boolean flag[2]; int turn;)

```
void P0()
{
    while(TRUE) //main while loop
    {
     flag[0] = TRUE; //turn on your own flag
     // so that P1 can't get in
     turn = 1; //Give a chance to P1
     while(flag[1] && turn == 1)
     //test for P1's flag and his turn
    {
 /*do nothing, stay here*/;
 }
 /* CS */;
 flag[0] = FALSE; //turn off your flag
 /* remainder */
} //end of main while
} //end of P0
```

```
void P1() {
while(TRUE) //main while loop{
 flag[1] = TRUE; //turn on your own flag
 // so that P0 can't get in
 turn = 0; //Give a chance to P0
 while(flag[0] && turn == 0)
 //test for P0's flag and his turn
        {
         /*do nothing, stay here*/;
         }
 /* CS */;
 flag[1] = FALSE; //turn off your flag
 /* remainder */
} //end of main while
} //end of P1
//MAIN PROGRAM
void main()
{
 flag[0] = FALSE;  flag[1] = FALSE;
 turn = 0;  parabegin (P0, P1);
}
```

Roadmap

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

Semaphores introduced by Dijkstra (1930 – 2002)

- Born in Rotterdam, The Netherlands
- 1972 recipient of the ACM Turing Award (Nobel Prize for computing)
- Responsible for
  - The idea of building operating systems as explicitly synchronized sequential processes
  - The formal development of computer programs
- Best known for
  - His efficient shortest-path algorithm
  - Having designed and coded the first Algol 60 compiler.
  - Famous campaign for the abolition of the `GOTO` statement
- http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1205.PDF

Semaphore

- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - initialize
  - Decrement (`semWait`) If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.
  - increment. (`semSignal`) If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

- P = Probeer ('Try')
- V = Verhoog ('Increment', 'Increase by one').

Semaphore Primitives

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

Figure 5.3 A Definition of Semaphore Primitives

Binary Semaphore Primitives

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

Figure 5.4 A Definition of Binary Semaphore Primitives

Strong/Weak 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 Semaphore Mechanism

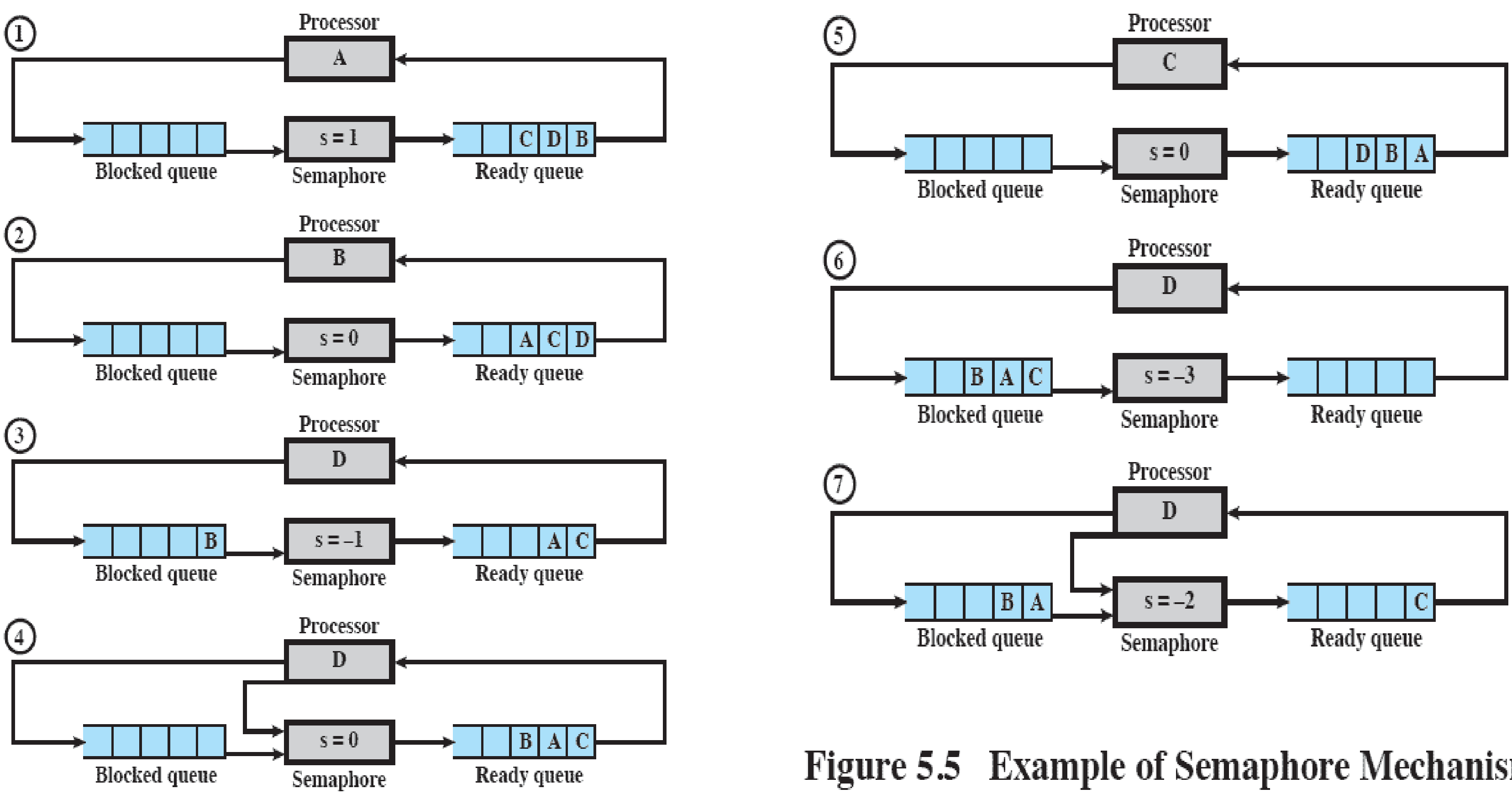
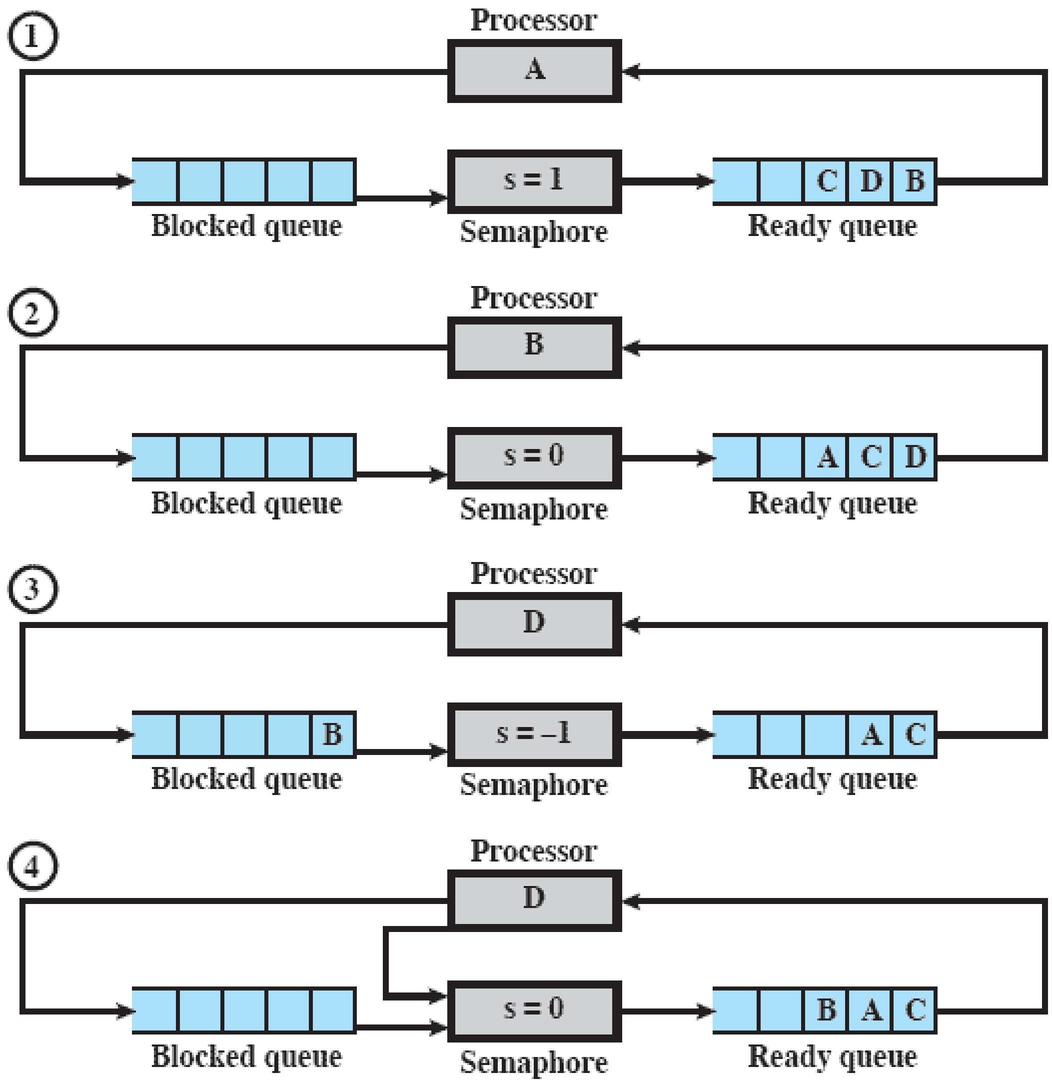


Figure 5.5 Example of Semaphore Mechanism

Example of Strong Semaphore Mechanism



Example of Semaphore Mechanism

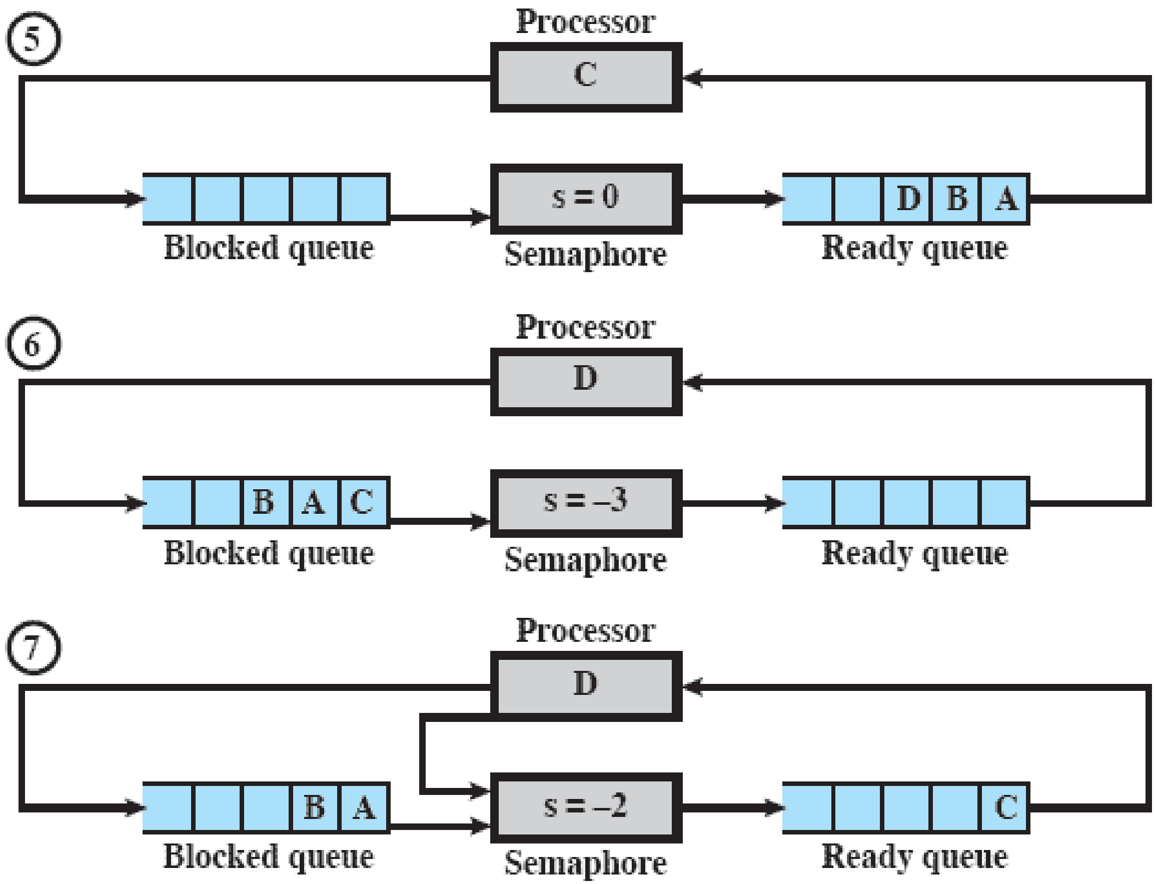


Figure 5.5 Example of Semaphore Mechanism

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

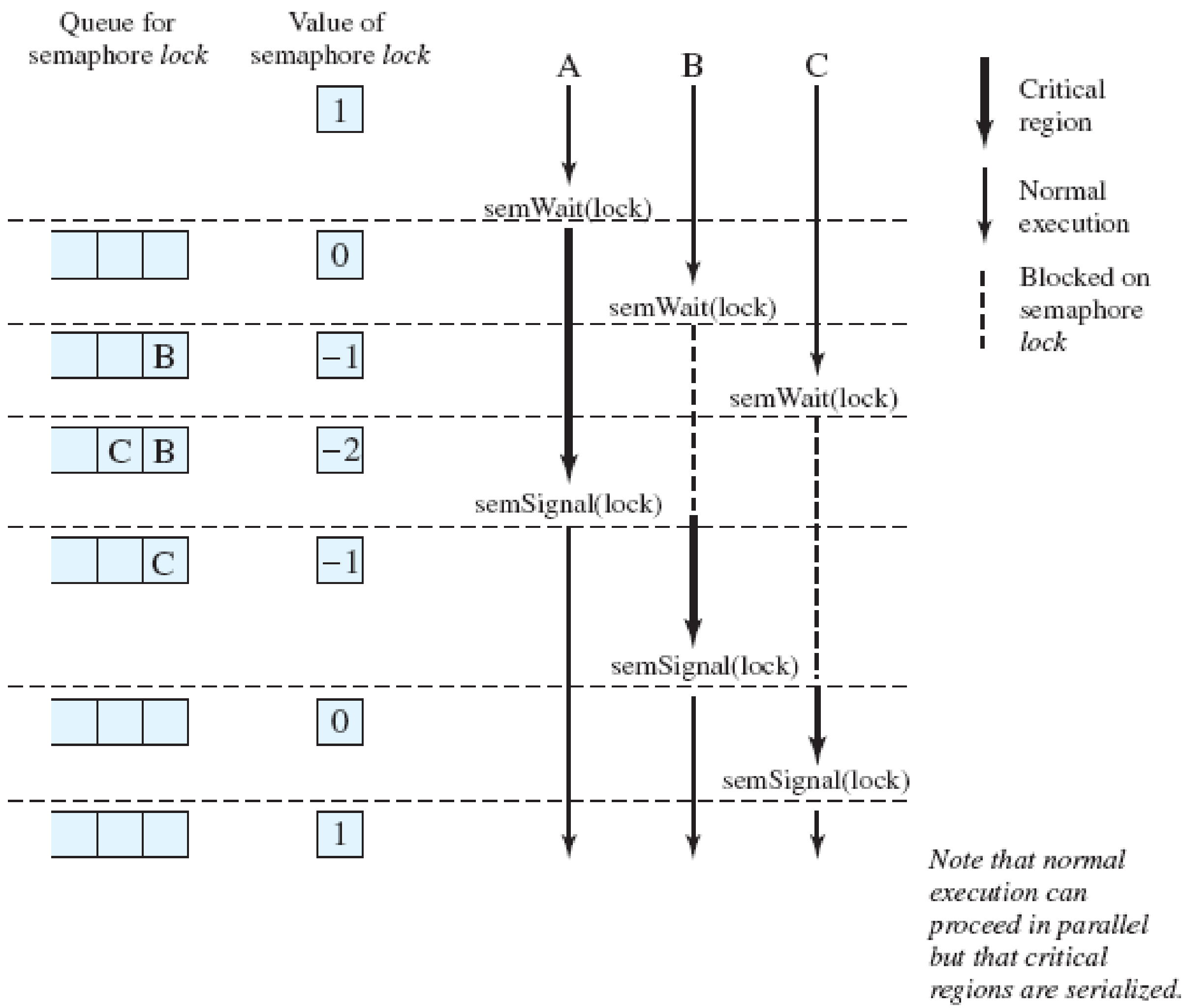


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

Producer/Consumer Problem

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

Producer/Consumer Animation

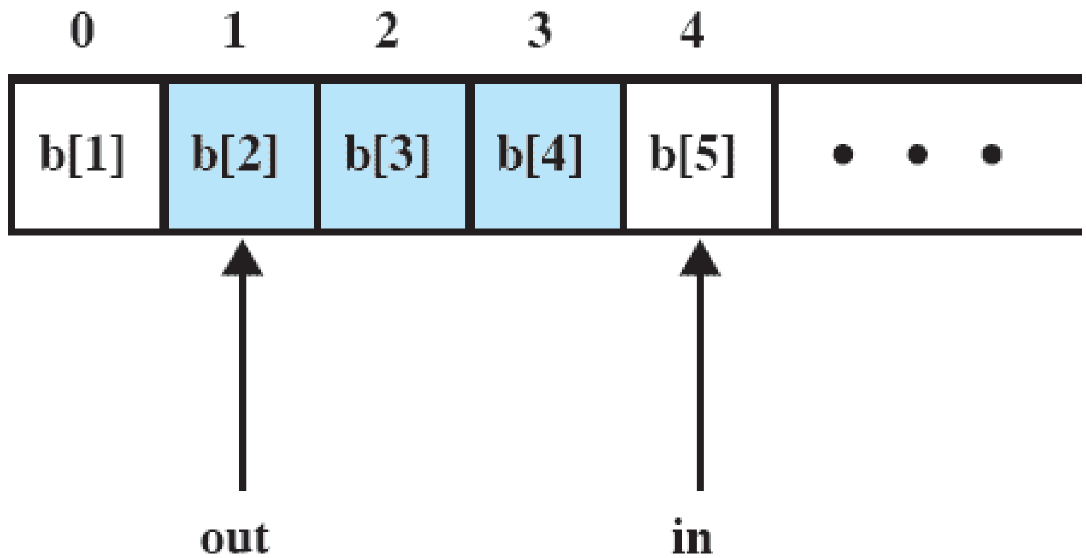
Functions

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

Producer	Consumer

while (true) {
    /* produce item v */
    b[in] = v;
    in++;
}
	while (true) {
	    while (in <= out)
	    /*do nothing */;
	    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);
}
```

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.

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);
}
```

Semaphores

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

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

Bounded Buffer

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

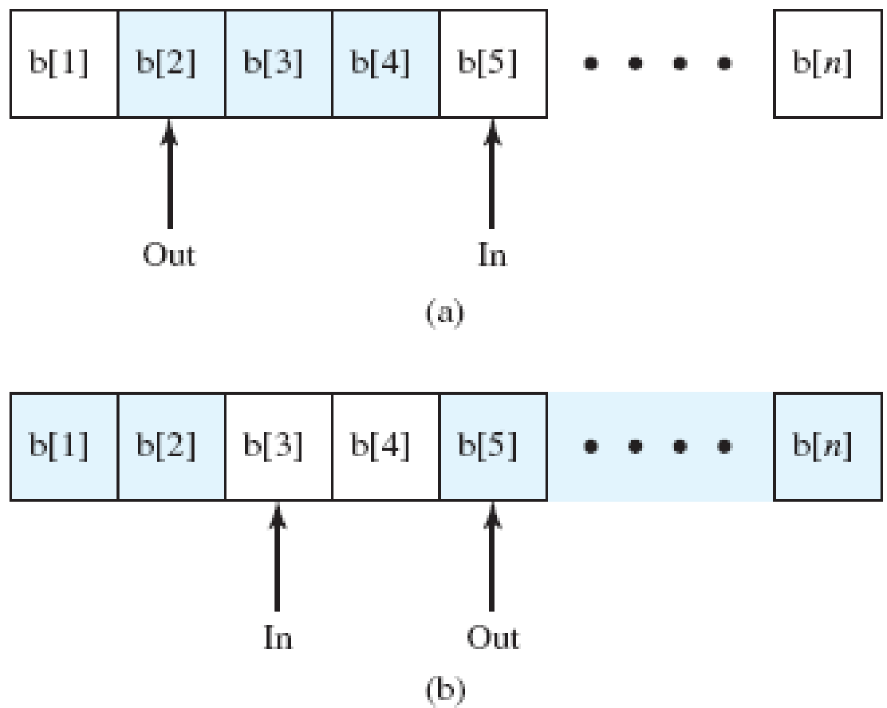


Figure 5.12 **Finite Circular Buffer for the Producer/Consumer Problem**

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);
}
```

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 */
}	}

Readers/Writers Problem

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

interaction of readers and writers.

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);
}
```

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

```
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);
}
```

Demonstration Animations

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

Roadmap

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

Monitors

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

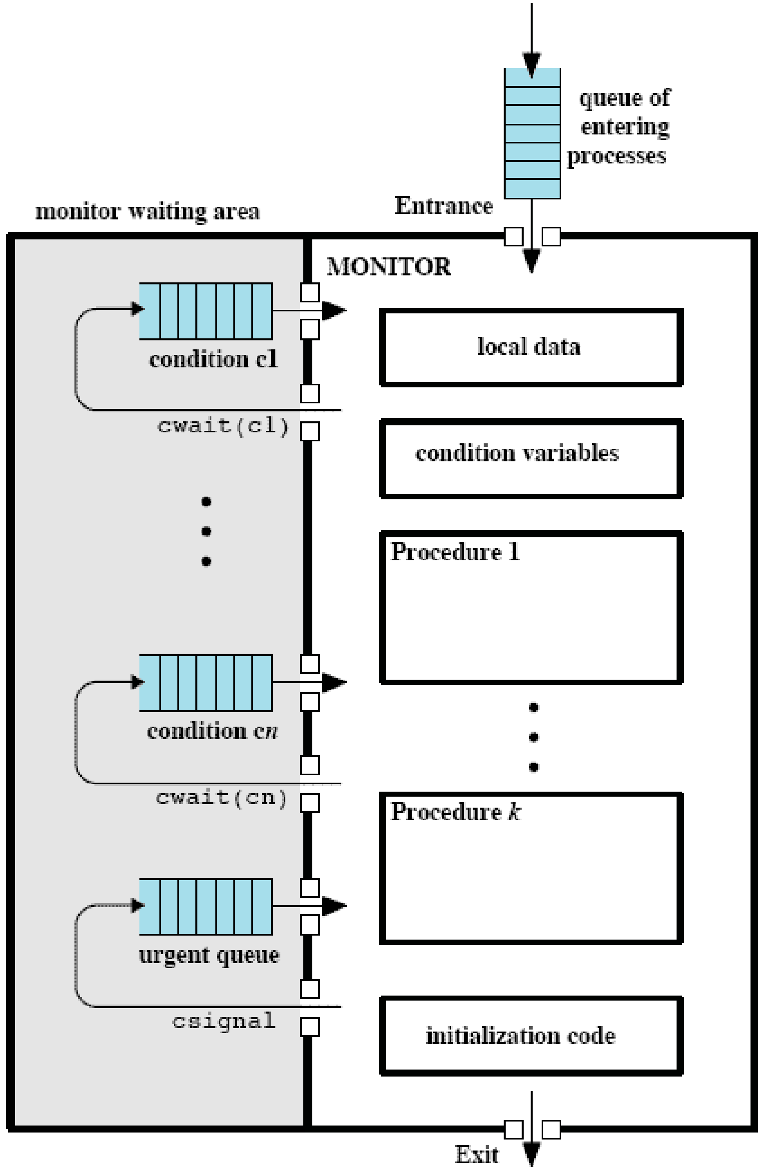
Chief characteristics

- Local data variables are accessible only by the monitor
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time, any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.

Synchronization

- Synchronisation achieved by **condition variables** within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - Cwait(c): Suspend execution of the calling process on condition *c*
  - Csignal(c) Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

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 */
}
```

- The monitor includes two condition variables (declared with the construct cond):
- notfull is true when there is room to add at least one character to the buffer,
- and notempty is true when there is at least one character in the buffer.

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);
}
```

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

Roadmap

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

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
- When a send primitive is executed in a process, there are two possibilities:
  - Either the sending process is blocked until the message is received, or it is not.
- Similarly, when a process issues a receive primitive, there are two possibilities:
  - If a message has previously been sent, the message is received and execution continues.
  - If there is no waiting message, then either

    (a)the process is blocked until a message arrives, or

    (b)the process continues to execute, abandoning the attempt to receive.

Blocking send,
Blocking receive

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

Non-blocking Send

- More natural for many concurrent programming tasks.
- **Nonblocking send, blocking receive**
  - Sender continues on
  - Receiver is blocked until the requested message arrives (e.g: sever)
- **Nonblocking 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

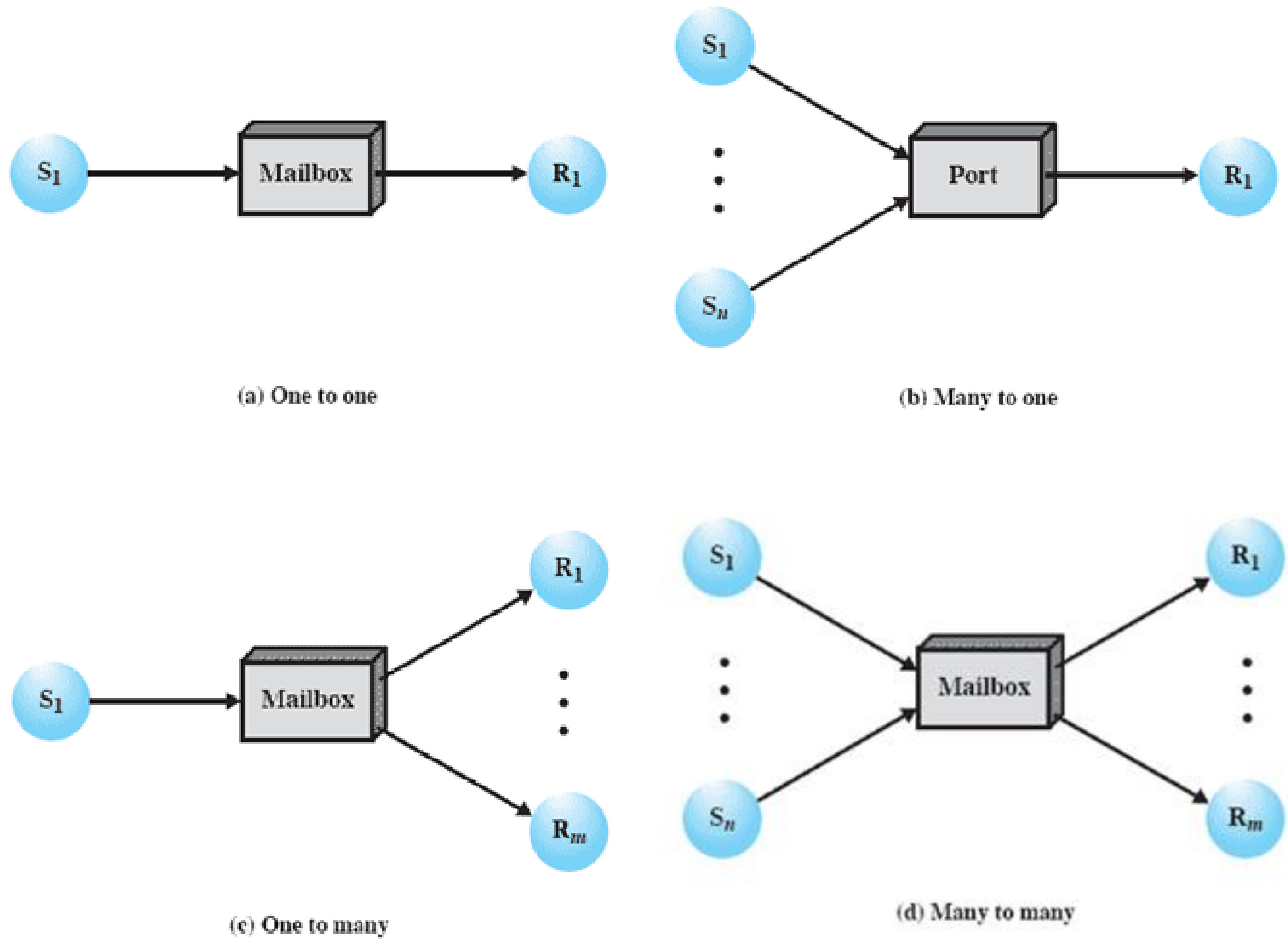


Figure 5.18 Indirect Process Communication

General Message Format

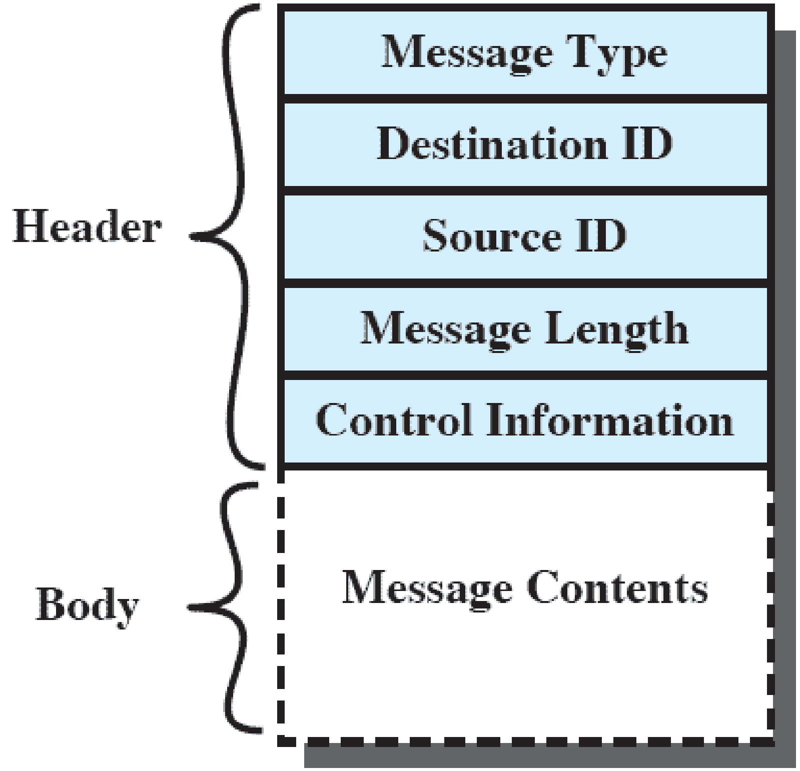


Figure 5.19 General Message Format

Mutual Exclusion Using Messages

- This is one way in which message passing can be used to enforce mutual exclusion.
- We assume the use of the blocking receive primitive and the non-blocking send primitive.
- This assumes that if more than one process performs the receive operation concurrently, then
  - • If there is a message, it is delivered to only one process and the others are blocked, or
  - • If the message queue is empty, all processes are blocked; when a message is available, only one blocked process is activated and given the message.

Mutual Exclusion Using Messages

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

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

Figure 5.20 Mutual Exclusion Using Messages

- This is an example of the use of message passing to the bounded-buffer producer/consumer problem.
- This program takes advantage of the ability of message passing to be used to pass data in addition to signals.
- Two mailboxes are used.
  - As the producer generates data, it is sent as messages to the **mailbox mayconsume.**
  - As long as there is at least one message in that mailbox, the consumer can consume.
- Hence **mayconsume** serves as the buffer; the data in the buffer are organized as a queue of messages.
- The "size" of the buffer is determined by the global variable capacity. Initially, the **mailbox mayproduce** is filled with a number of null messages equal to the capacity of the buffer. The number of messages in **mayproduce** shrinks with each production and grows with each consumption

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);
}
```

Roadmap

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

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++;
         }
      }
}
```