Operating System

Chapter 5. Concurrency: Mutual Exclusion and Synchronization

Lynn Choi

School of Electrical Engineering

Process Interaction

- **In a single-processor system,**
  - Process executions are interleaved to increase CPU utilization
  - The relative speed of execution of processes cannot be predicted
    - It depends on the activities of other processes, the way OS handles interrupts, and the scheduling policies
  - The following difficulties arise
    - Mutual exclusion
    - Deadlock
    - Starvation
    - Race condition
- **The same problems exist in a multiprocessor system**
  - Process executions are overlapped (run in parallel)
  - Again, the relative speed of execution of processes is unpredictable

Competing Processes

- Sharing of global resources can create
  - Need for *mutual exclusion*
  - *Deadlock*
  - Starvation
- Mutual exclusion
  - Suppose two processes require access to a single printer
  - Printer is a nonsharable resource
    - Without care, lines from competing processes will be interleaved
  - During the course of execution, only one process should be allowed to access the resource at a time.
    - The portion of program that accesses the resource is called a ***critical section*** of the program.
    - Only one process at a time is allowed in its critical section
  - How to enforce mutual exclusion?

Competing Processes

❑ Deadlock

- Consider two processes P1 and P2, and two resources R1 and R2
- Each process needs to access both resources to complete its function
- Suppose the following scenario:
  - OS assigns R1 to P2 and R2 to P1
  - Each process is waiting for the other resource
  - Neither will release the resource until it acquires the other resource
  - Two processes are deadlocked!

❑ Starvation (Indefinite postponement)

- Consider three processes P1, P2, and P3.
- Each process requires access to resource R.
- Suppose the following scenario:
  - P1 has R and both P2 and P3 wait for R
  - OS grants access to P3, then P1, then P3, …
  - P2 may be indefinitely postponed to access the resource

Cooperating Processes

- Sharing of global data may lead to *race condition*
- Race condition
  - Consider the following two processes

```
P1:     a = a + 1;
        b = b + 1;
P2:     b = 2 * b;
        a = 2 * a;
```

  - If we start with a = b, each process taken separately will leave a = b
  - Now consider the following concurrent execution:
    - Two processes respect mutual exclusion on each individual data item

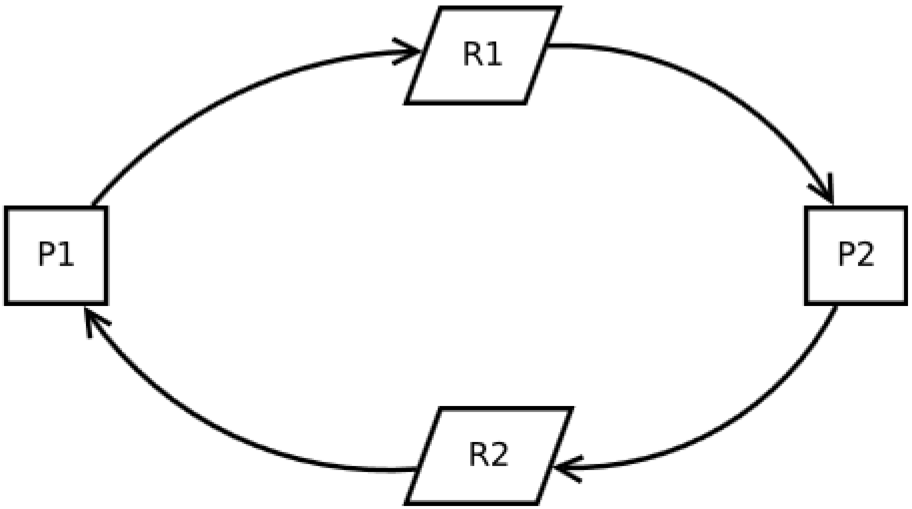
```
        a = a + 1;
        b = 2 * b;
        b = b + 1;
        a = 2 * a;
```

  - If we start with a = b = 1, at the end we have a = 4, b = 3.
  - The problem can be avoided by declaring the entire sequence in each process to be a critical section

Deadlock & Starvation

❑ Deadlock

- A situation where two or more competing processes are waiting for the other to release a resource



❑ Starvation (Infinite Postponement)

- A situation where the progress of a process is indefinitely postponed by the scheduler

❑ Livelock

- A situation where two or more processes continuously change their states without making progress

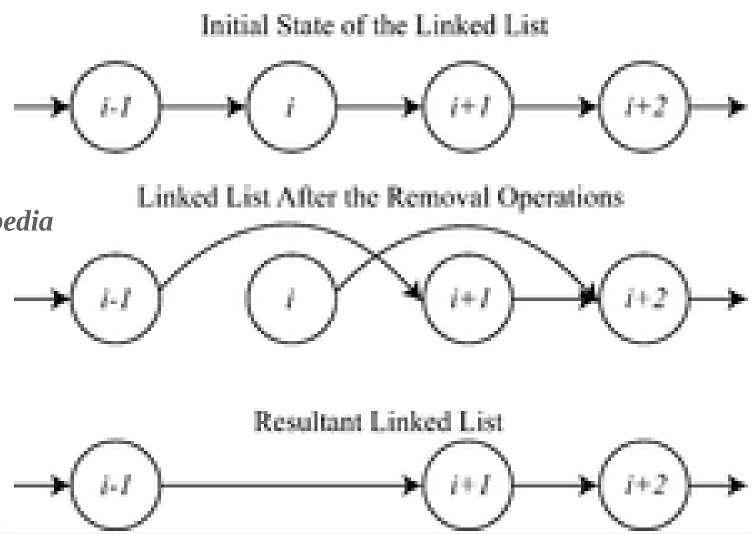
Race Condition

❑ Race condition occurs

- When two or more processes/threads access shared data and they try to change it at the same time. Because thread/process scheduling algorithm can switch between threads, you don't know which thread will access the shared data first. In this situation, both threads are '*racing*' to access/change the data.
- Operations upon shared data are *critical sections that must be mutually exclusive* in order to avoid harmful collision between processes or threads.
  - Regarded as a programming error
  - Difficult to locate this kind of programming errors as results are *nondeterministic* and *not reproducible*

❑ Example

- Two processes attempt to remove two nodes simultaneously from a singly-linked list
  - Only one node is removed instead of two.



Source: Wikipedia

Concurrency: Key Terminologies

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.

Source: Pearson

Table 5.1 Some Key Terms Related to Concurrency

Atomic Operation

- "*Atomic*" means
  - Indivisible, uninterruptable
  - Must be performed atomically, which means either "success" or "failure"
    - *Success*: successfully change the system state
    - *Failure*: no effect on the system state
- Atomic operation
  - A function or action implemented as a single instruction or as a sequence of instructions that appears to be indivisible
    - No other processes can see an intermediate state
  - Can be implemented by hardware or by software
  - HW-level atomic operations
    - Test-and-set, fetch-and-add, compare-and-swap, load-link/store-conditional
  - SW-level solutions
    - Running a group of instructions in a *critical section*

HW Support for Mutual Exclusion

- **In a uniprocessor,**
  - A process continues 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.
- **Disable interrupt on an entry to a critical section**

```
While (true) {
        /* disable interrupt */
        /* critical section */
        /* enable interrupt */
        /* remainder */
}
```

  - The simplest approach - No context switching guarantees mutual exclusion
- **Problem**
  - Performance degradation
  - Only for a uniprocessor
    - Disabling interrupt does not affect other cores/processors
    - Other cores are free to run any code
      - Can enter a critical section for the same shared resource

HW Support for Mutual Exclusion

Special machine instructions

- Carry out two operations atomically, such as reading and writing of a single memory location
- Test-and-set, fetch-and-add, compare-and-swap etc.
  - Access to a shared memory location is exclusive and atomic
  - Test-and-set is supported by most processor families
    - x86, IA64, SPARC, IBM z series, etc.
- These are atomic operations supported by the machine instructions
- Can be used to implement semaphores and other SW solutions
- Can also be used for multiprocessors
- Problem
  - Busy waiting
    - Other process or thread accessing the same memory location must wait and retry until the previous access is complete
  - Deadlock and starvation can also happen

Compare and Swap Instruction

- "*Compare and Swap*" instruction
  - A **compare** is made between a memory value and a test value

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

  - Some version of this instruction is available on nearly all processor families (x86, IA64, SPARC, IBM z series, etc.)
    - Atomicity is guaranteed by HW

Critical Section using Compare and Swap

```
/* 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

Source: Pearson

Exchange Instruction

❑ "Exchange" instruction

- Exchange the content of a register with that of a memory location.

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

- x86 and IA-64 support XCHG instruction

Critical Section using Exchange

```
/* 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

Source: Pearson

Special Instructions: +/–

❑ 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

❑ Disadvantages

- Busy-waiting
- 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
- Deadlock is possible
  - Process P1 executes compare and swap and enter its critical section
  - P1 is then interrupted and give control to P2 who has higher priority.
  - P2 will be denied access due to mutual exclusion and go to busy waiting loop.
  - P1 will never be dispatched since it has lower priority than P2.

Semaphore

❑ Semaphore

- A *variable* that provides a simple abstraction for controlling access to a common resource in a programming environment
- The value of the semaphore variable can be changed by only 2 operations
  - V operation (also known as "signal")
    - Increment the semaphore
  - P operation (also known as "wait")
    - Decrement the semaphore
  - The value of the semaphore **S** is usually the number of units of the resource that are currently available.

❑ Type of semaphores

- *Binary semaphore*
  - Have a value of 0 or 1
    - *0 (locked, unavailable)*
    - *1 (unlocked, available)*
- *Counting semaphore*
  - Can have an arbitrary resource count

Semaphore

- A variable that has an integer value upon which only three operations are defined
  1) May be initialized to a nonnegative integer value
  2) The semWait (P) operation decrements the value
  3) The semSignal (V) operation increments the value
- There is no way to inspect or manipulate semaphores other than these three operation

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

Source: Pearson

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

Source: Pearson

Strong/Weak Semaphores

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

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

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

Example of Semaphore Mechanism

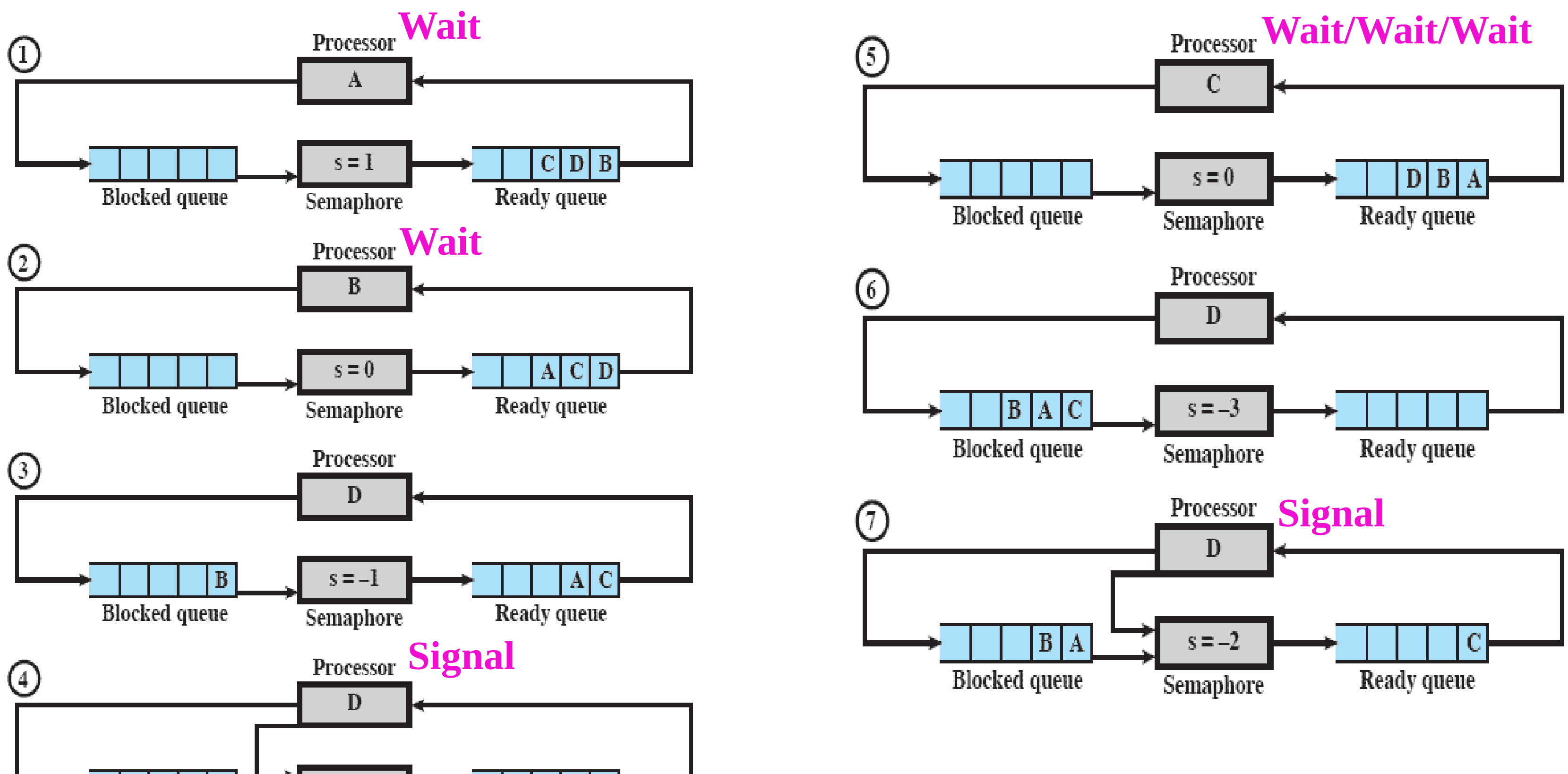


Figure 5.5 Example of Semaphore Mechanism

Source: Pearson

Mutual Exclusion

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

Figure 5.6 Mutual Exclusion Using Semaphores

Source: Pearson

Shared Data Protected by a Semaphore

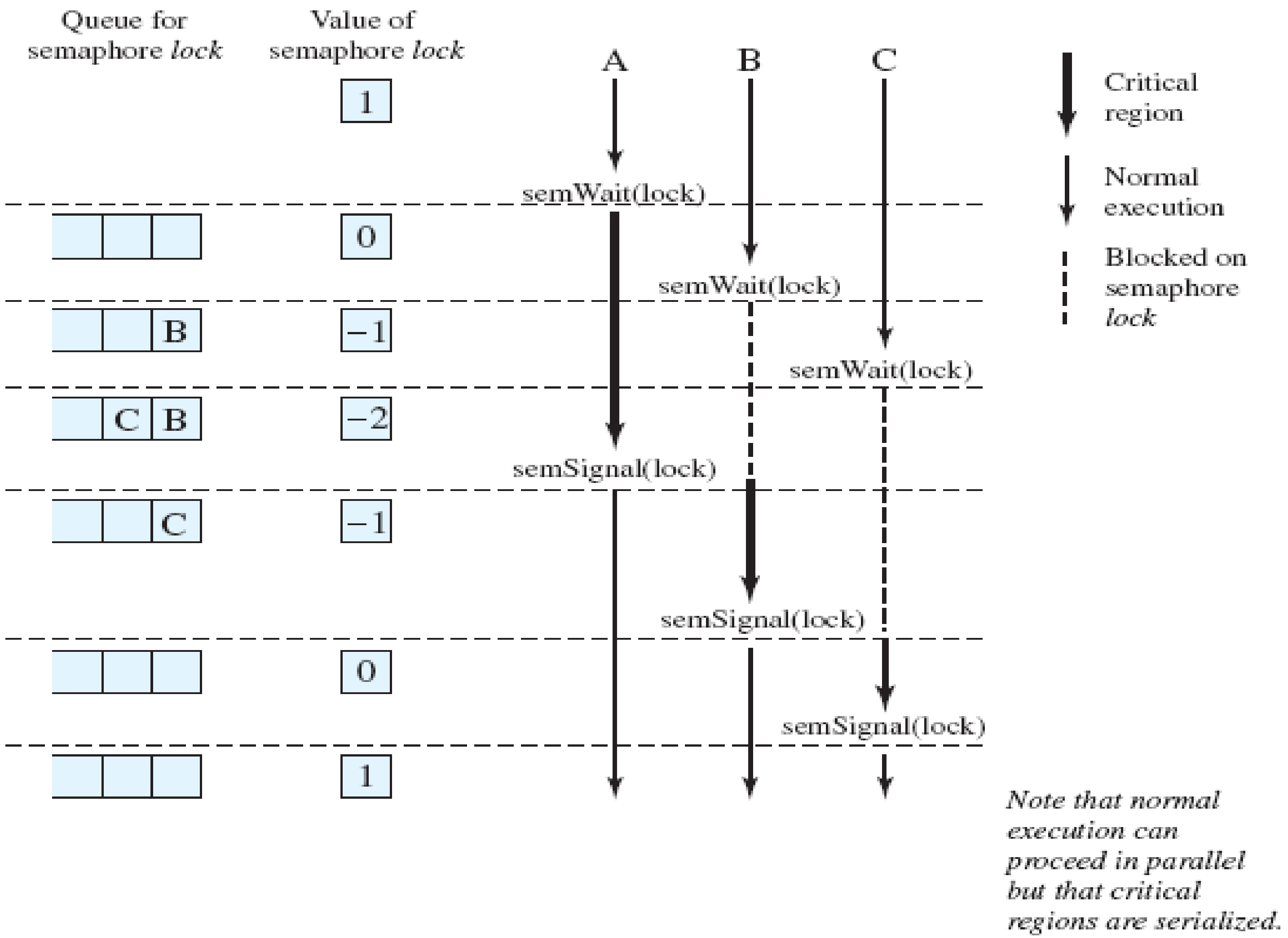


Figure 5.7 **Processes Accessing Shared Data Protected by a Semaphore**

Source: Pearson

Producer/Consumer Problem

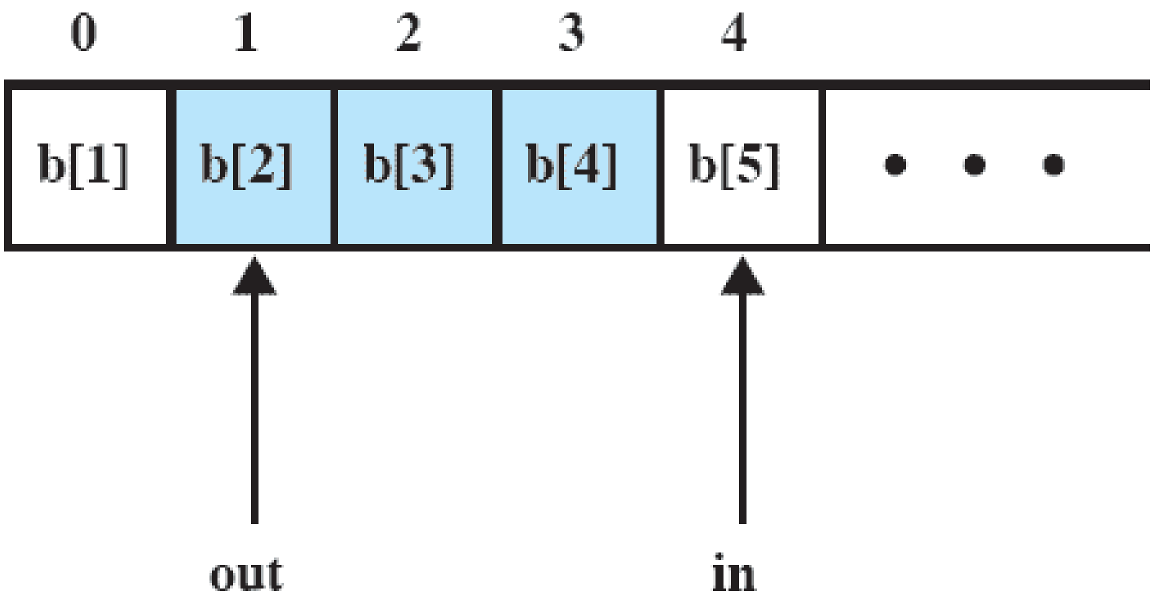
General Situation

- One or more producers
  - Produce data item and insert it in a buffer
- One consumer
  - Delete it from the buffer and consume the data item
- Only one producer or consumer may access the buffer at any time

The problem

- Ensure that the producer can't add data into a full buffer
- Consumer can't remove data from an empty buffer

Buffer Structure



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Source: Pearson

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);
}
```

Source: Pearson

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

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.

Source: Pearson

Correct Solution with Binary Semaphore

```
/* 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);
}
```

Correct Solution with Counting 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

Source: Pearson

Scenario

	Producer	Consumer	s	n
1			1	0
2	Wait(s)		0	0
3	Signal(s)		1	0
4	Signal(n)		1	1
5		Wait(n)	1	0
6		Wait(s)	0	0
7		Signal(s)	1	0
8	Wait(s)		0	0
9	Signal(s)		1	0
10	Signal(n)		1	1
11		Wait(n)	1	0
12		Wait(s)	0	0
13		Signal(s)	1	0
14	Wait(s)		0	0
15	Signal(s)		1	0
16	Signal(n)		1	1
17		Wait(n)	1	0
18		Wait(s)	0	0
19		Signal(s)	1	0

Source: Pearson

Implementation of Semaphores

```
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;
}
```

(a) Compare and Swap Instruction

```
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;
}
```

(b) Interrupts

Figure 5.14 Two Possible Implementations of Semaphores

Source: Pearson

Monitor

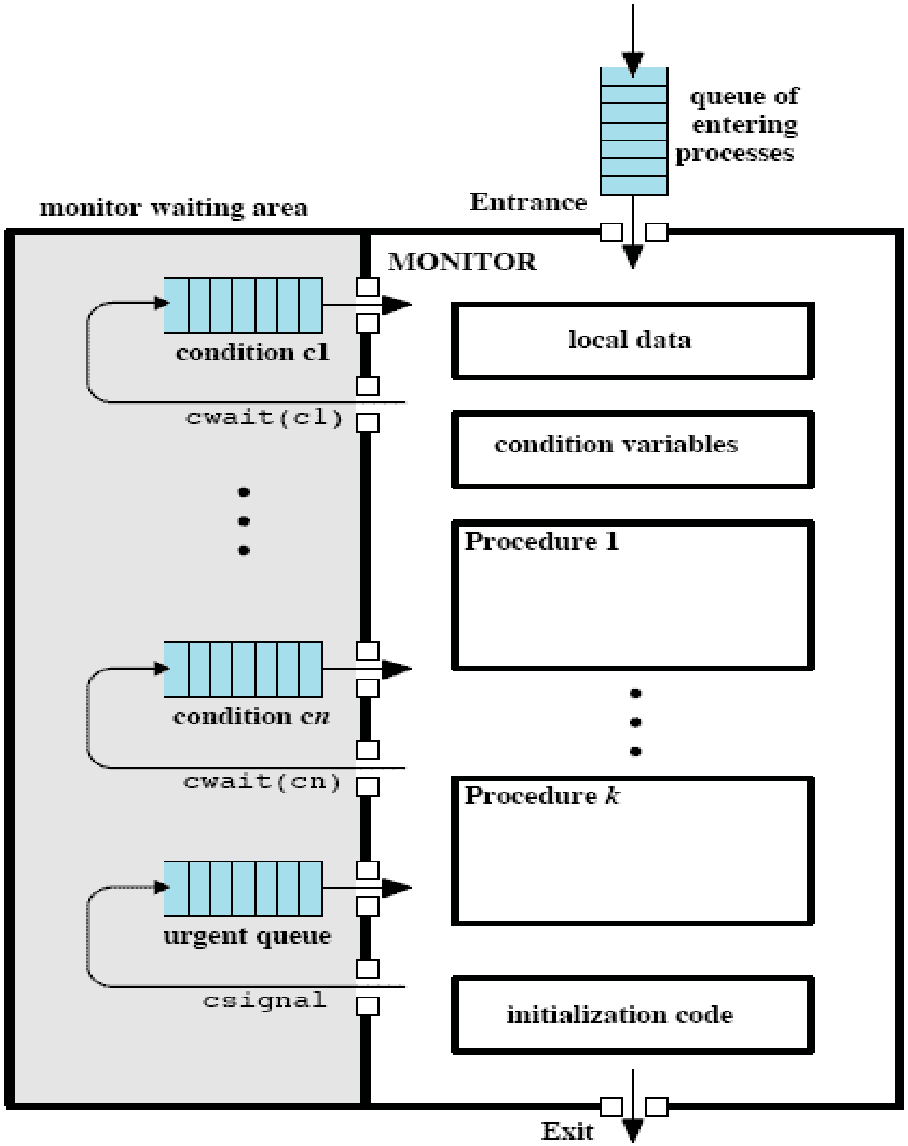
- **Motivation**
  - Semaphore
    - It is not easy to produce a correct program using semaphores
    - semWait and semSignal operations may be scattered throughout a program and it is not easy to see the overall effect of these operations
- **Monitor**
  - Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
  - Implemented in a number of programming languages
    - Including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
  - Monitor consists of one or more procedures, an initialization code, and local data
    - Local data variables are accessible only by the monitor's procedures and not by any external procedure
    - Process enters the monitor by invoking one of its procedures
    - Only one process may be executing in the monitor at a time

Synchronization with Monitor

Condition variable

- Monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor
- Condition variables are operated by two functions
  - cwait(c): suspend the execution of the calling process on condition c
  - csignal(c): resume the execution of a process blocked on the same condition
    - If there are so such processes, the signal is lost (do nothing)

Structure of a Monitor



Source: Pearson

Problem Solution Using a Monitor

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

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

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

Source: Pearson

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

Message Passing

- When processes interact with one another, the following actions must be satisfied by the system
  - Mutual exclusion
  - Synchronization
  - Communication
- Message passing is one approach to provide these functions and
  - Works with shared memory and distributed memory multiprocessors, uniprocessors, and distributed systems
- The actual function is normally provided in the form of a pair of primitives
  - send (destination, message)
    - A process sends information in the form of a *message* to another process designated by a *destination*
  - receive (source, message)
    - A process receives information by executing the `receive` primitive, indicating the *source* and the *message*

Synchronization

- **Communication of a message between two processes implies synchronization between the two**
  - The receiver cannot receive a message until it has been sent by another process
- **Both sender and receiver can be blocking or nonblocking**
  - When a send primitive is executed, there are two possibilities
    - Either the sending process is blocked until the message is received, or it is not
  - When a receive primitive is executed there are also two possibilities
    - If a message has previously been sent, the message is received and the execution continues
    - 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

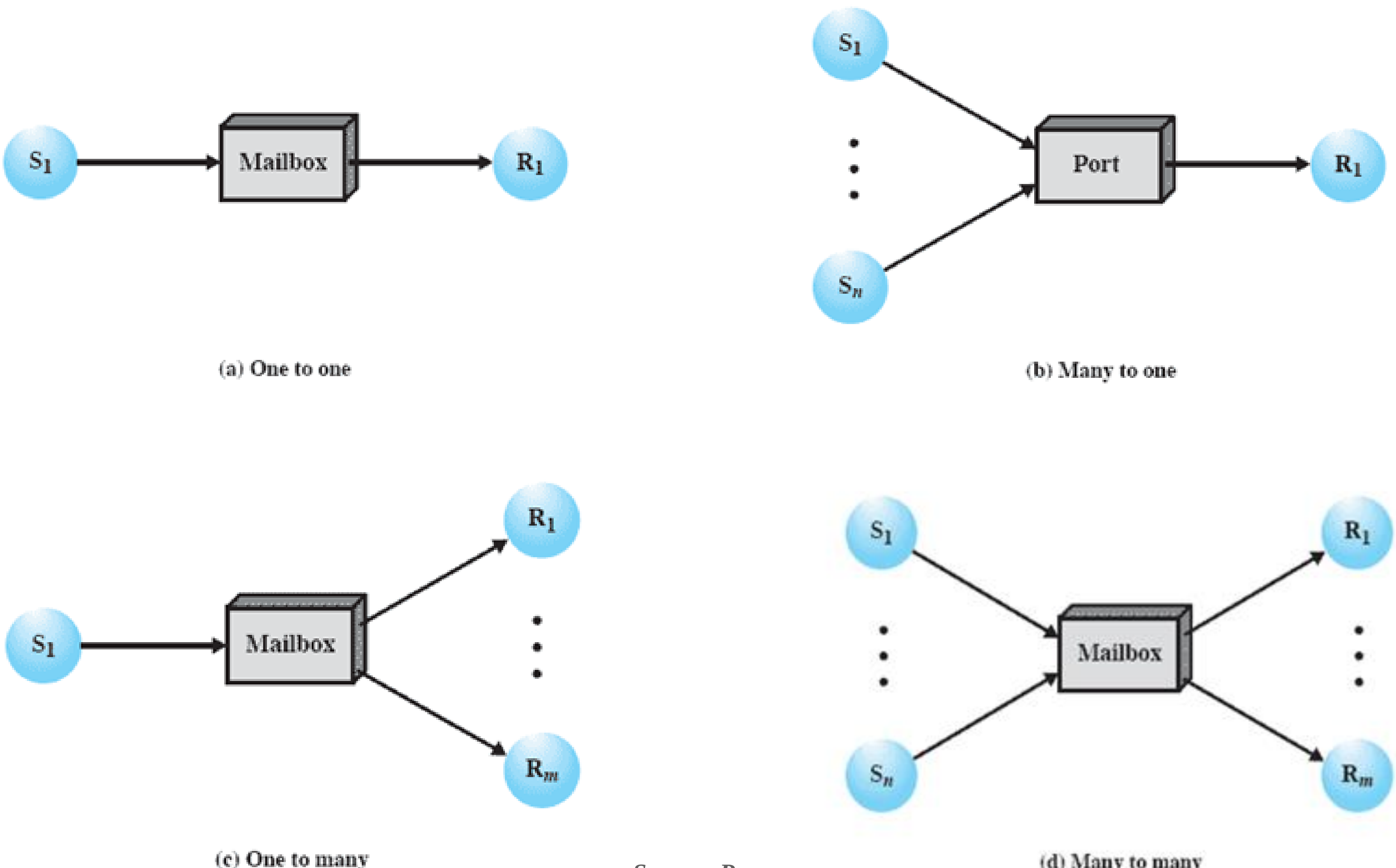
Blocking/Nonblocking Send/Receive

- **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, blocking receive**
  - Sender continues on but receiver is blocked until the requested message arrives
  - The most useful combination
  - It allows a process to send one or more messages to a variety of destinations as quickly as possible
- **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
  - Send primitive includes a specific identifier of the destination process
  - Receive primitive can be handled in one of two ways
    - Explicit addressing
      - Require that the process explicitly designate a sending process
      - Effective for cooperating concurrent processes
    - Implicit addressing
      - Source parameter of the receive primitive possesses a value returned when the receive operation has been performed
- Indirect addressing
  - Messages are sent to a shared data structure consisting of queues that can temporarily hold messages
  - Queues are referred to as *mailboxes*
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox
  - Allows for greater flexibility in the use of messages

Indirect Process Communication



Source: Pearson

General Message Format

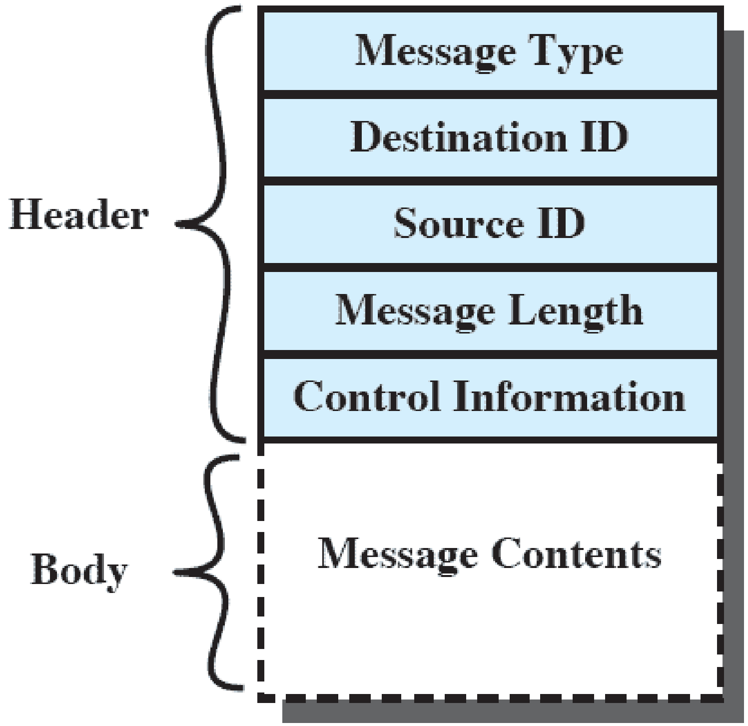


Figure 5.19 General Message Format

Source: Pearson

Mutual Exclusion

```
/* 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

Source: Pearson

Producer Consumer with Message

```
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);
}
```

Source: Pearson

Homework 4

- Exercise 5.1
- Exercise 5.2
- Exercise 5.4
- Exercise 5.7
- Exercise 5.9
- Exercise 5.10