Operating Systems

Chapter 5 Mutual Exclusion(互斥) and Synchronization(同步)

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

Concurrency

- Concurrency arises in three different contexts:
  - Multiple applications(多应用程序)
    - Multiprogramming
  - Structured application(结构化应用程序)
    - Some application can be designed as a set of concurrent processes
  - Operating-system structure(操作系统结构)
    - Operating system is a set of processes or threads

Concurrency

Table 5.1 Some Key Terms Related to Concurrency

critical section 临界区	A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.
deadlock 死锁	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 活锁	A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.
mutual exclusion 互斥	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 竞争条件	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 饥饿	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

5.1 Principles of Concurrency

- 5.1.1 A Simple Example
- 5.1.2 Race Condition
- 5.1.3 Operating System Concerns
- 5.1.4 Process Interaction
- 5.1.5 Requirements for Mutual Exclusion

Difficulties of Concurrency

- Sharing of global resources
- Operating system managing the allocation of resources optimally
- Difficult to locate programming errors

A Simple Example

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

A Simple Example(Uniprocessor)

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

A Simple Example(Multiprocessor)

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

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

5.1 Principles of Concurrency

- 5.1.1 A Simple Example
- 5.1.2 Race Condition
- 5.1.3 Operating System Concerns
- 5.1.4 Process Interaction
- 5.1.5 Requirements for Mutual Exclusion

Race Condition(竞争条件)

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes or thread.

(竞争条件发生在当多个进程或者线程在读写数据时，其最终结果依赖于多个进程或者线程的指令执行顺序)

5.1 Principles of Concurrency

- 5.1.1 A Simple Example
- 5.1.2 Race Condition
- 5.1.3 Operating System Concerns
- 5.1.4 Process Interaction
- 5.1.5 Requirements for Mutual Exclusion

Operating System Concerns

- Keep track of various processes(through PCB)
- Allocate and deallocate resources
  - Processor time
  - Memory
  - Files
  - I/O devices
- Protect data and resources
- Output of process must be independent of the speed of execution of other concurrent processes

5.1 Principles of Concurrency

- 5.1.1 A Simple Example
- 5.1.2 Race Condition
- 5.1.3 Operating System Concerns
- 5.1.4 Process Interaction
- 5.1.5 Requirements for Mutual Exclusion

Process Interaction(进程交互)

- Processes unaware of each other
- Processes indirectly aware of each other
- Process directly aware of each other

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that one Process has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Competition Among Processes for Resources(进程间的资源争用)

- Mutual Exclusion(互斥)
  - Critical sections
    - Only one program at a time is allowed in its critical section
    - Example only one process at a time is allowed to send command to the printer
- Deadlock(死锁)
- Starvation(饥饿)

Illustration of Mutual Exclusion by critical section

.entercritical：进入临界区　.exitcritical：退出临界区

```
/* PROCESS 1 */

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

```
/* PROCESS 2 */

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

. . .

```
/* PROCESS n */

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

Figure 5.1 Illustration of Mutual Exclusion

5.1 Principles of Concurrency

- 5.1.1 A Simple Example
- 5.1.2 Race Condition
- 5.1.3 Operating System Concerns
- 5.1.4 Process Interaction
- 5.1.5 Requirements for Mutual Exclusion

Requirements for Mutual Exclusion

1. Only one process at a time is allowed in the critical section for a resource (一次只允许一个进程进入临界区，忙则等待)
2. A process that halts in its noncritical section must do so without interfering with other processes (阻塞于临界区外的进程不能干涉其它进程)
3. No deadlock or starvation (不会发生饥饿和死锁，有限等待)

Requirements for Mutual Exclusion

4. A process must not be delayed access to a critical section when there is no other process using it (闲则让进)
5. No assumptions are made about relative process speeds or number of processes (对相关进程的执行速度和处理器数目没有要求)
6. A process remains inside its critical section for a finite time only (有限占用)

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

5.2 Mutual Exclusion: Hardware Support

- 5.2.1 Interrupt Disabling(中断禁止)
- 5.2.2 Special Machine Instructions(特殊机器指令)

Hardware Support: Interrupt Disabling

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

Hardware Support: Interrupt Disabling

- A process runs until it invokes an operating system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion on uniprocessor system
- Disadvantage:
  - Processor is limited in its ability to interleave programs
  - disabling interrupts on one processor will not guarantee mutual exclusion in multi-processors environment.

5.2 Mutual Exclusion: Hardware Support

- 5.2.1 Interrupt Disabling
- 5.2.2 Special Machine Instructions

Hardware Support: Special Machine Instructions

- Performed *in a single instruction cycle*
  - Access to the memory location is blocked for any other instructions

Hardware Support: Special Machine Instructions

- Compare and swap

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

Hardware Support for Mutual Exclusion

```
/* program mutualexclusion */
const int n = /* 进程个数 */;
int bolt;
void P(int i)
{
    while (true) {
      while (compare_and_swap(bolt, 0, 1)
      == 1)
          /* 不做任何事 */;
      /* 临界区 */;
      bolt = 0;
      /* 其余部分 */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));
}
```

Hardware Support: Special Machine Instructions

- Exchange Instruction

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

Hardware Support for Mutual Exclusion

```
    /* program mutualexclusion */
int const n = /* 进程个数 **/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true)  {
        do exchange (keyi, bolt)
        while (keyi != 0);
        /* 临界区 */;
        bolt = 0;
        /* 其余部分 */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

Hardware Support: Special Machine Instructions

- Advantages
  - By sharing main memory，it is applicable to any number of processes
    - single processor
    - multiple processors
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Hardware Support: Special Machine Instructions

- Disadvantages
  - Busy-waiting(忙等待) consumes processor time
  - Starvation(饥饿) is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock (死锁) is possible
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

5.3 Semaphores

- 5.3.1 Overview
- 5.3.2 Mutual Exclusion
- 5.3.3 The Producer/Consumer Problem
- 5.3.4 Implement of Semaphores

Semaphores(信号量)

- Fundamental principle(基本原理): Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.(两个或者多个进程可以通过简单的信号进行合作，一个进程可以被迫在一个位置停止，直到它收到一个信号)
- For signaling, special variables called semaphores are used(一种称为*信号量*的特殊变量用来传递信号)
- If a process is waiting for a signal, it is suspended until that signal is sent (如果一个进程在等待一个信号，它会被挂起，直到它等待的信号被发出）

Semaphores

- Semaphore is a variable that has an integer value(整数值)
  - Initialize：May be initialized to a nonnegative number(非负数)
  - semWait (P)：Wait operation decrements the semaphore value，If the value becomes negative, then the process executing the semWait is blocked.
  - semSignal (V)：Signal operation increments semaphore value，If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Semaphore Primitives(原语，原子操作)

```
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(二元信号量)

- Binary Semaphore is a variable that has an integer value
  - May be initialized to 0 or 1.
  - semWaitB：checks the semaphore value. If the value is zero, then the process executing the semWaitB is blocked. If the value is one, then the value is changed to zero and the process continues execution.
  - semSignalB：checks to see if any processes are blocked on this semaphore. If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.

Binary Semaphore(二元信号量) Primitives(原语)

```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};

void semWaitB(binary_semaphore s)
{
      if (s.value ==  1)
            s.value = 0;
      else
            {
                  place this process in s.queue;
                  block this process;
            }
}
void semSignalB(semaphore s)
{
      if (s.queue.is_empty())
            s.value = 1;
      else
      {
            remove a process P from s.queue;
            place process P on ready list;
      }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Semaphore terms (信号量术语)

- Binary Semaphore (二元信号量)
- Mutex (互斥信号量)
- Counting Semaphore (计数信号量)
- General Semaphore(一般信号量)
- Weak Semaphore(弱信号量)
- Strong Semaphore(强信号量)

5.3 Semaphores

- 5.3.1 Overview
- 5.3.2 Mutual Exclusion
- 5.3.3 The Producer/Consumer Problem
- 5.3.4 Implement of Semaphores

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

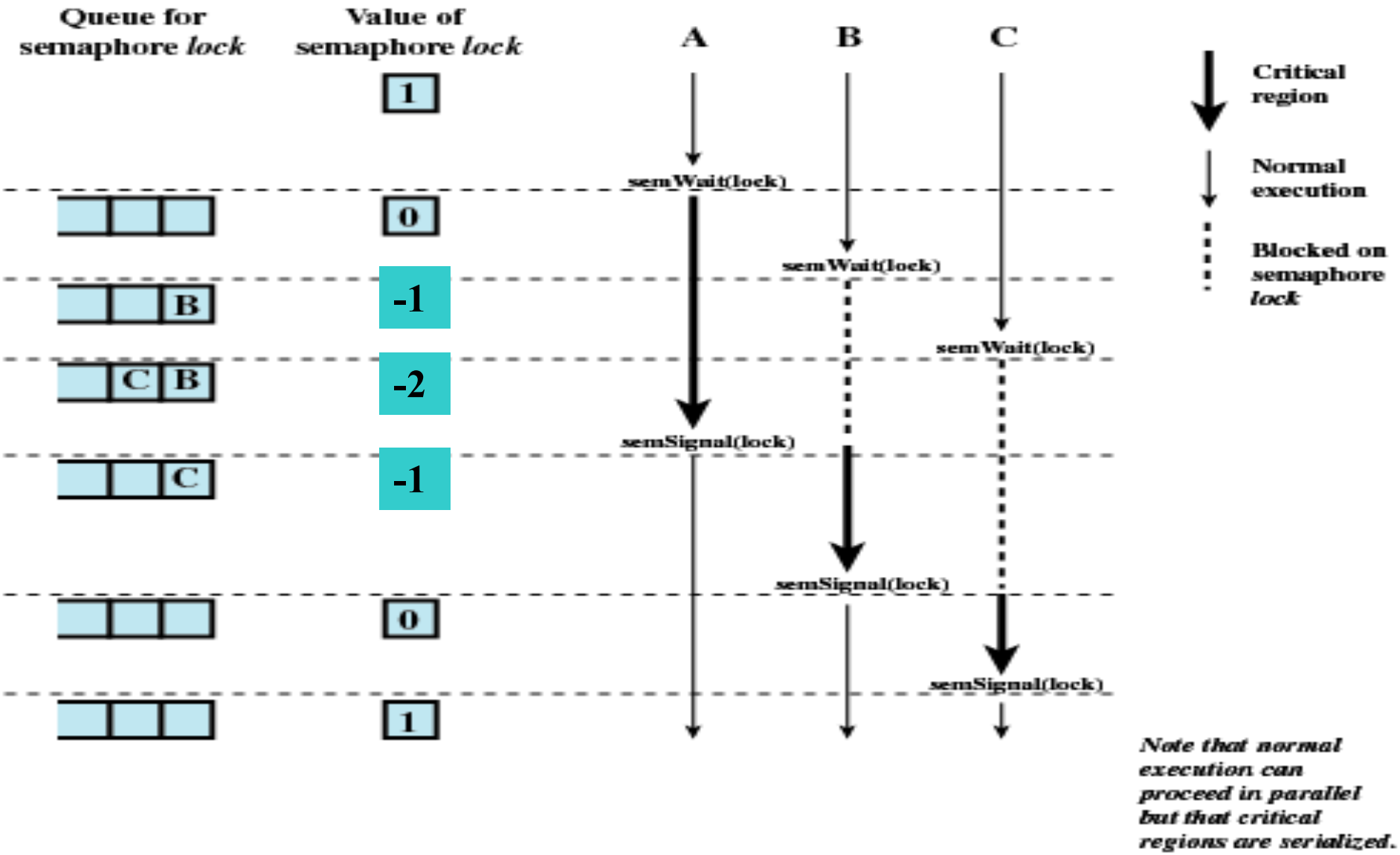


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

More than one process in its critical section at a time(多个进程同时在临界区内)

- Initialize the semaphore to the specified value
- *s.count* >= 0: *s.count* is the number of processes that can execute semWait(s) without suspension
- *s.count* < 0: The magnitude of *s.count* is the number of processes suspended in *s.queue*.

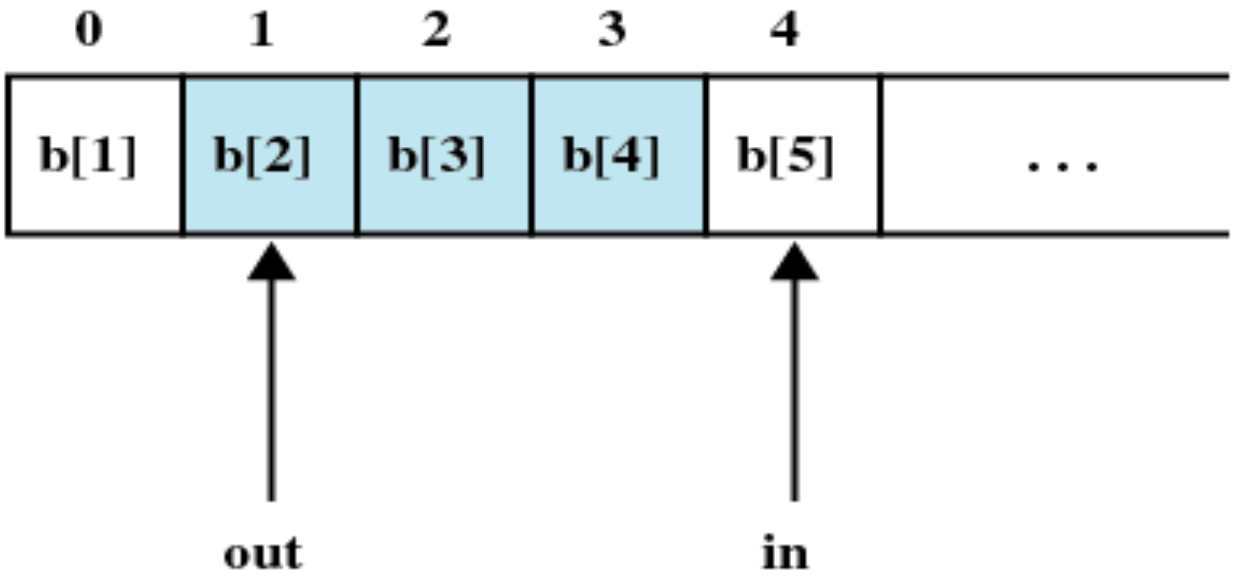
5.3 Semaphores

- 5.3.1 Overview
- 5.3.2 Mutual Exclusion
- 5.3.3 The Producer/Consumer Problem
- 5.3.4 Implement of Semaphores

Producer/Consumer Problem(生产者/消费者问题)

- 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

Producer/Consumer Problem with Infinite Buffer(无限缓冲区)



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Producer with Infinite Buffer

```
producer:
while (true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```

Consumer with Infinite Buffer

```
consumer:
while (true) {
  while (in <= out)
     /*do  nothing */;
  w = b[out];
  out++;
  /* consume item w */
}
```

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore 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);
}
```

缓冲区中的项数

控制进入临界区

避免“超前”消费

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

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.

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore 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 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program  producerconsumer */
semaphore n = 0;
semaphore 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);
}
```

控制“超前消费”

控制进入临界区

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

Producer/Consumer Problem with Finite Buffer(有限缓冲区)

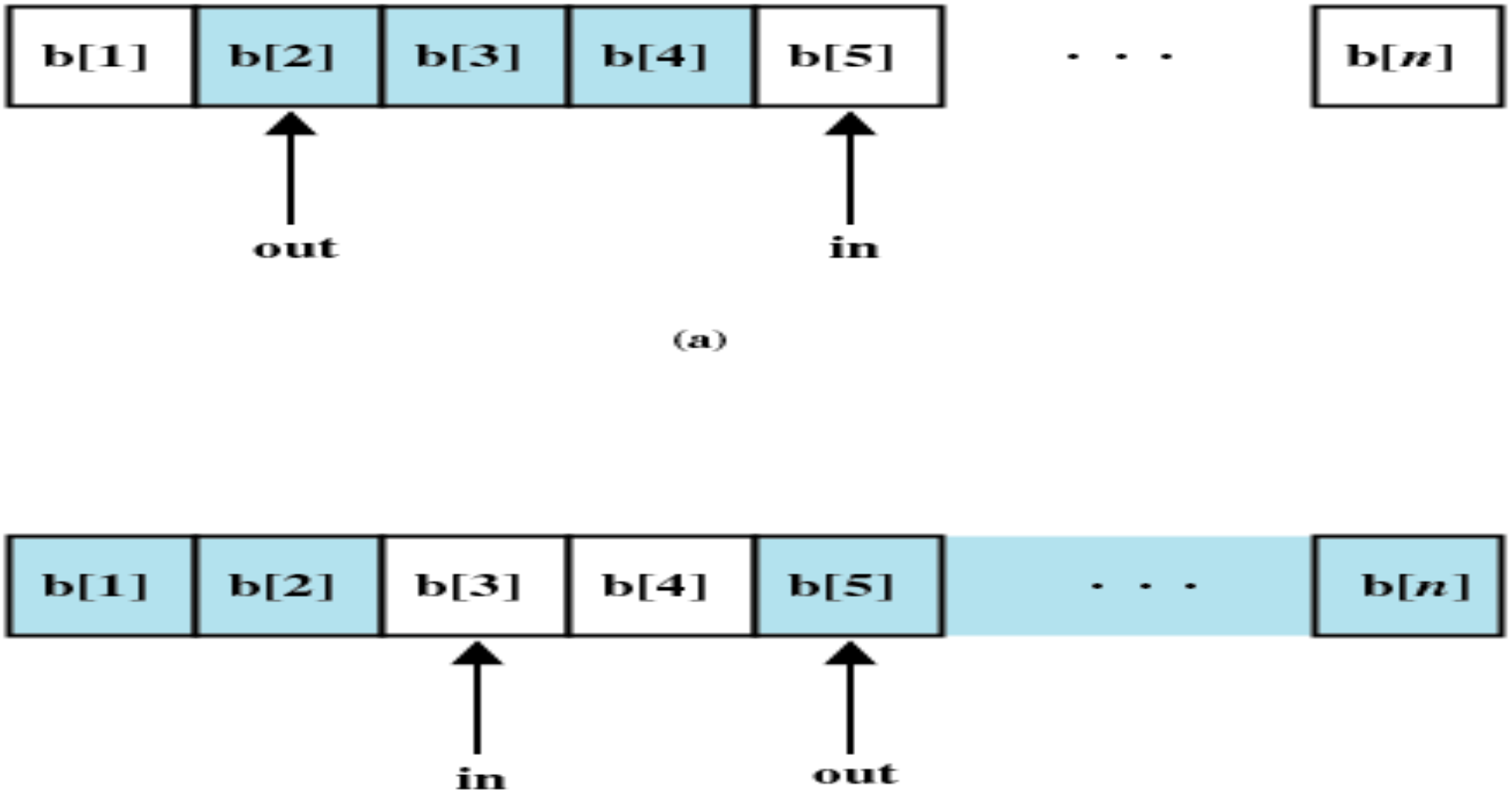


Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

Producer/Consumer Problem with Finite Buffer(有限缓冲区)

- The buffer is treated as a circular storage, and pointer values must be expressed modulo the size of the buffer.
- The following relationships hold:

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

Producer with Circular Buffer

```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)  /*
  do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

Consumer with Circular Buffer

```
consumer:
while (true) {
  while (in == out)
     /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore 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);
}
```

控制进入临界区

控制“超前”消费

控制生产“过剩”

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

5.3 Semaphores

- 5.3.1 Overview
- 5.3.2 Mutual Exclusion
- 5.3.3 The Producer/Consumer Problem
- 5.3.4 Implement of Semaphores

5.3.4 Implement of Semaphores

- Implement in hardware or firmware(固件)
- Implement in software, e.g. Dekker or Peterson
- Implement by inhibit interrupts(中断禁止) for a single-processor system

5.3.4 Implement 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 inter-
rupts */;
   }
   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

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

Monitors(管程)

- Monitor is a software module consisting of one ore more procedures, an initialization sequence, and local data(管程由一个或者多个例程、一段初始化代码和局部数据组成). And the chief characteristics are the following:
  - 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

Monitors Operations(管程操作)

- A monitor supports synchronization by the use of **condition variables** (管程通过条件变量实现同步)
- cwait(c): Suspend execution of the calling process on condition *c*. The monitor is now available for use by another process.
- 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.
- Note that monitor *wait* and *signal* operations are different from those for the semaphore. **If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.**

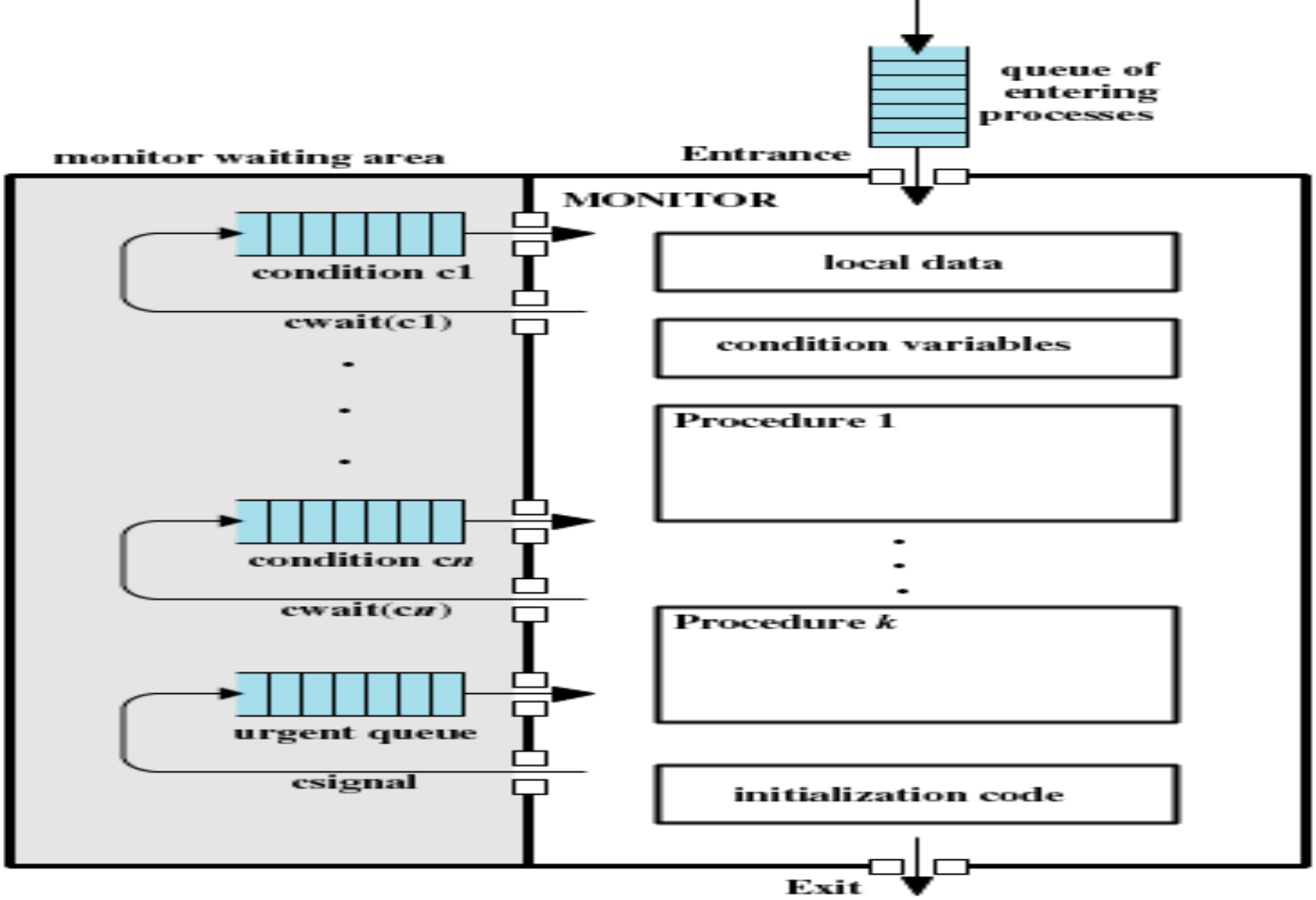


Figure 5.15 Structure of a Monitor

```
/* 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 */
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a 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);
}
```

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

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

Message Passing

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```

Synchronization

- Sender and receiver may or may not be blocking
- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered

Synchronization

- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

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 or receive primitive could use source parameter to return a value when the receive operation has been performed

Addressing

- 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

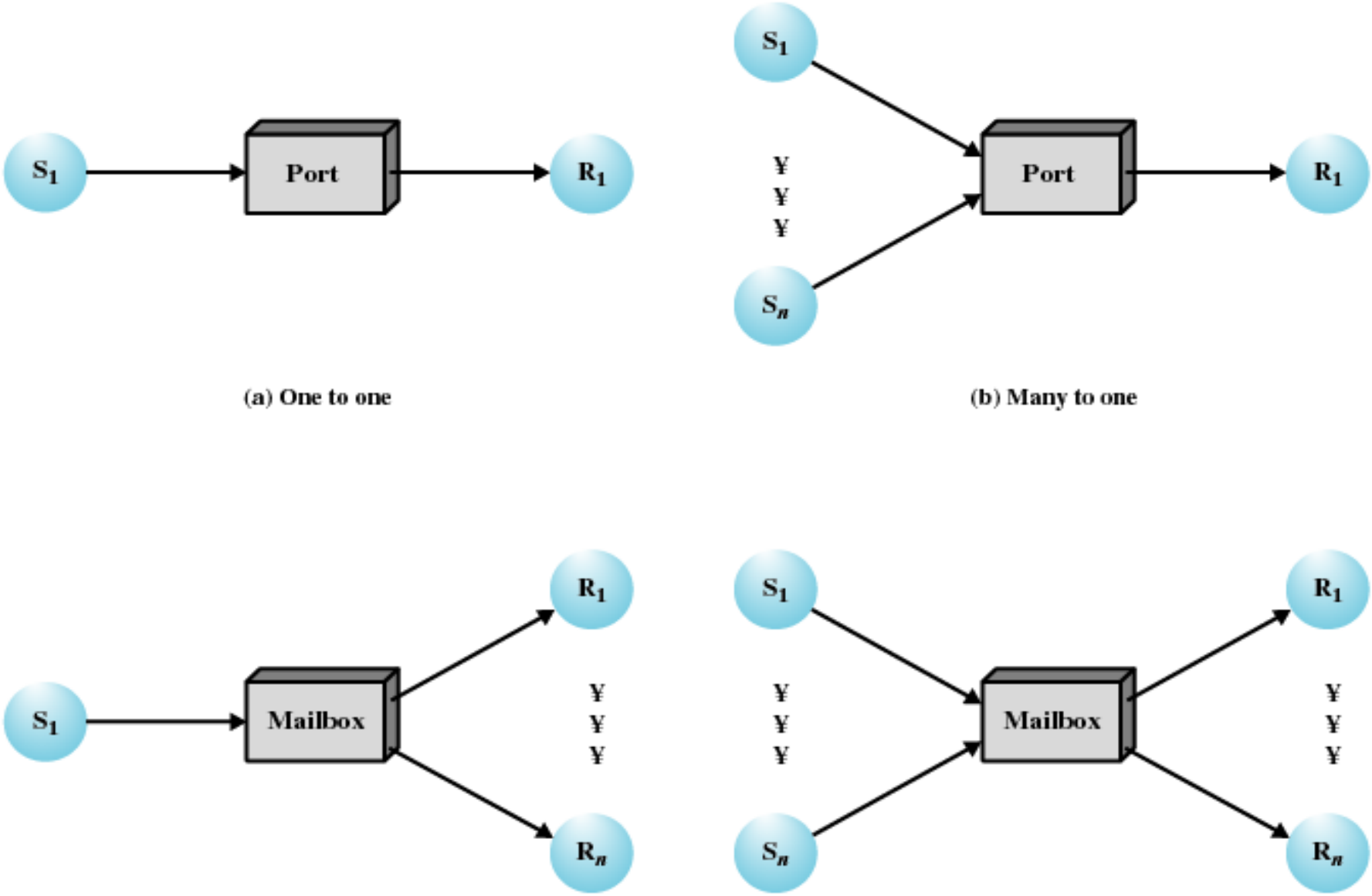


Figure 5.18 Indirect Process Communication

Message Format

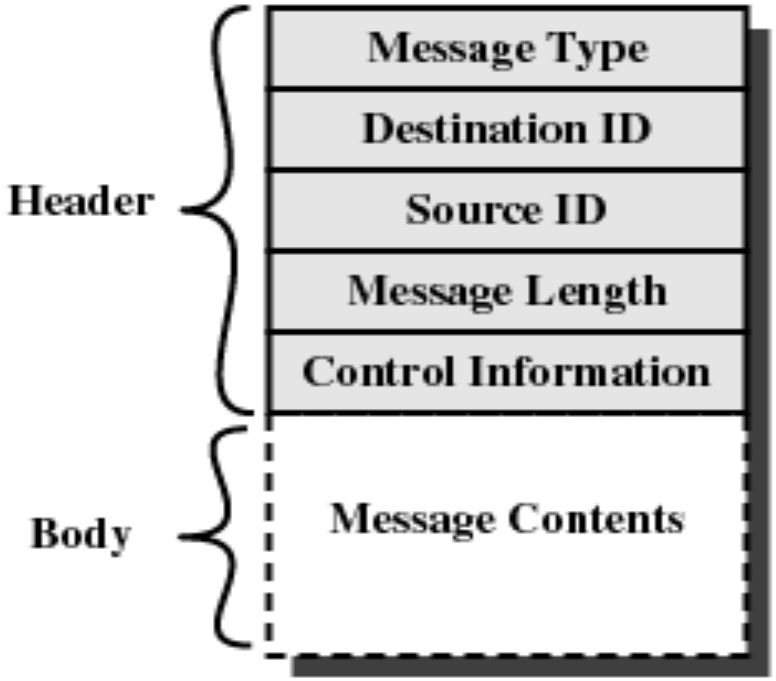


Figure 5.19 General Message Format

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

Figure 5.20 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 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

Readers/Writers Problem

- 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);
     }
```

正在读的进程数

用于读写/写写互斥保护

用于readcount修改保护

Figure 5.22 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);
}
```

readcount:正在读的进程数

rsem：当至少有一个写请求时，对新的读请求进行阻塞

wsem：用于读写互斥保护

writecount:正在和将要进行写的进程数

y：控制writecount的修改

x：控制readcount的修改

z：只允许一个读进程在rsem上排队，其他读进程在z上排队

Figure 5. 23 A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary