

Concurrency: Mutual Exclusion and Synchronization

Chapter 5

Management of Processes and Threads

- ▶ **Multiprogramming (多道程序设计)**

- ▶ The management of multiple processes within a uniprocessor system

- ▶ **Multiprocessing (多处理)**

- ▶ The management of multiple processes within a multiprocessor (for example, SMP)

- ▶ **Distributed processing (分布式处理)**

- ▶ The management of multiple processes executing on multiple, distributed computer system (for example, cluster)

Thus, concurrency is fundamental to OS.



Independent and Cooperating Processes

- ▶ *The concurrent processes may be either **independent processes** or **cooperating processes***
 - ▶ *A process is **independent** if it cannot affect or be affected by the other processes executing in the system*
 - ▶ *Any process that does not share any data with any other process is independent*
 - ▶ *A process is **cooperating** if it can affect or be affected by the other processes executing in the system*
 - ▶ *Any process that shares data with other processes is a cooperating process*



Key Terms Related to Concurrency (1)

- ▶ **Critical section/region (临界区)**
 - ▶ A portion of program code where process will access shared resource. When another process is executing in this portion, this process can not
- ▶ **Deadlock (死锁)**
 - ▶ A group (two or more) of processes can not make any progress in their execution because each one is waiting for another process to complete something.
- ▶ **Livelock (活锁)**
 - ▶ Two or more processes **continually change their state** in response to changes in the other processes, but they do nothing useful.



Key Terms Related to Concurrency (2)

- ▶ **Mutual Exclusion (互斥)**

- ▶ *Making sure that if one process is using a shared variable or file, the other processes are excluded from doing the same thing.*

- ▶ **Race Condition (竞争条件??) 竞态**

- ▶ *The situations where two or more processes are reading or writing some shared data and the final result depends on who runs exactly when, are called the race condition*

- ▶ **Starvation (饥饿)**



Principles of Concurrency

Design Issues for Concurrency

- ▶ **Concurrency includes a host of design issues:**
 - ▶ **Communication among processes**
 - ▶ **Sharing and competing for resources**
 - ▶ **Synchronization of activities of multiple processes**
 - ▶ **Allocation of processor time**



When does concurrency arise?

- ▶ **Concurrency arises in three different contexts**
 - ▶ **Multiple applications → Multiprogramming**
 - ▶ **Structured application → Application can be constructed as a set of concurrent processes**
 - ▶ **Operating-system structure → Operating systems are themselves implemented as a set of processes or threads**



Difficulties with Concurrency

- ▶ **Sharing global resources is fraught with peril.**
 - ▶ For example: the order of reading and writing a global variable is critical
- ▶ **It is difficult to manage the allocation of resources optimally.**
 - ▶ For example: deadlock
- ▶ **It is very difficult to locate a programming error**
 - ▶ Because the results are typically not deterministic and reproducible



A Simple Example

(1)

```
void echo()
```

```
{
```

```
    chin = getchar();
```

```
    chout = chin;
```

```
    putchar(chout);
```

```
}
```

→ *Save one keystroke from a keyboard to chin*

→ *Transfer the input character from chin to chout*

→ *Send the variable chout to the display*

Here, the variable chin is a shared global variable



A Simple Example (2)

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

The input to the process P1 is lost



A Simple Example

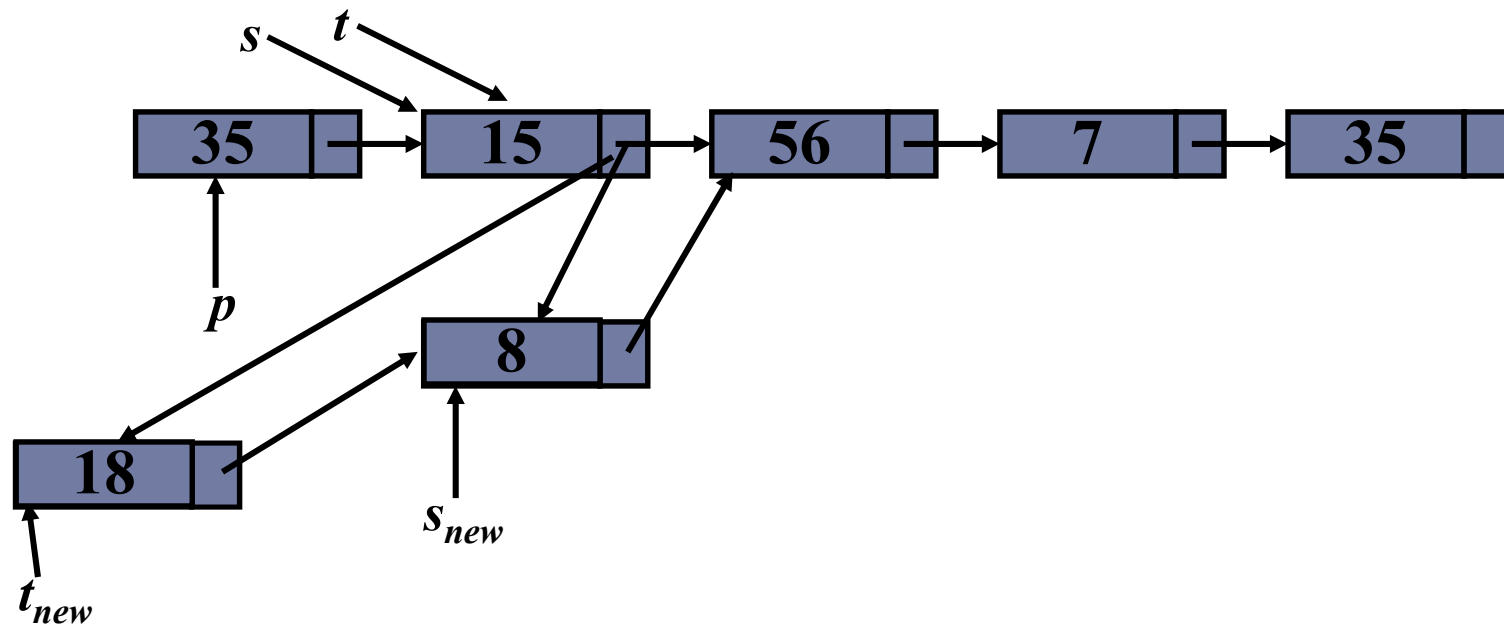
(3)

- ▶ **Why does the problem occur?**
 - ▶ On a uniprocessor system, an interrupt can stop instruction execution anywhere in a process
 - ▶ On a multiprocessor system, two processes may be executing simultaneously and both trying to access the same global variable
- ▶ ***Solution: controlled access to the shared resources***



Example: Insertion into a linked list

顺序执行



$s_{new} := \text{create a data block};$

$s_{new} \rightarrow \text{data} := 8;$

$s := p \rightarrow \text{next};$ // s points to the second element

$s_{new} \rightarrow \text{next} := s \rightarrow \text{next};$

$s \rightarrow \text{next} := s_{new};$

$t_{new} := \text{create a data block};$

$t_{new} \rightarrow \text{data} := 18;$

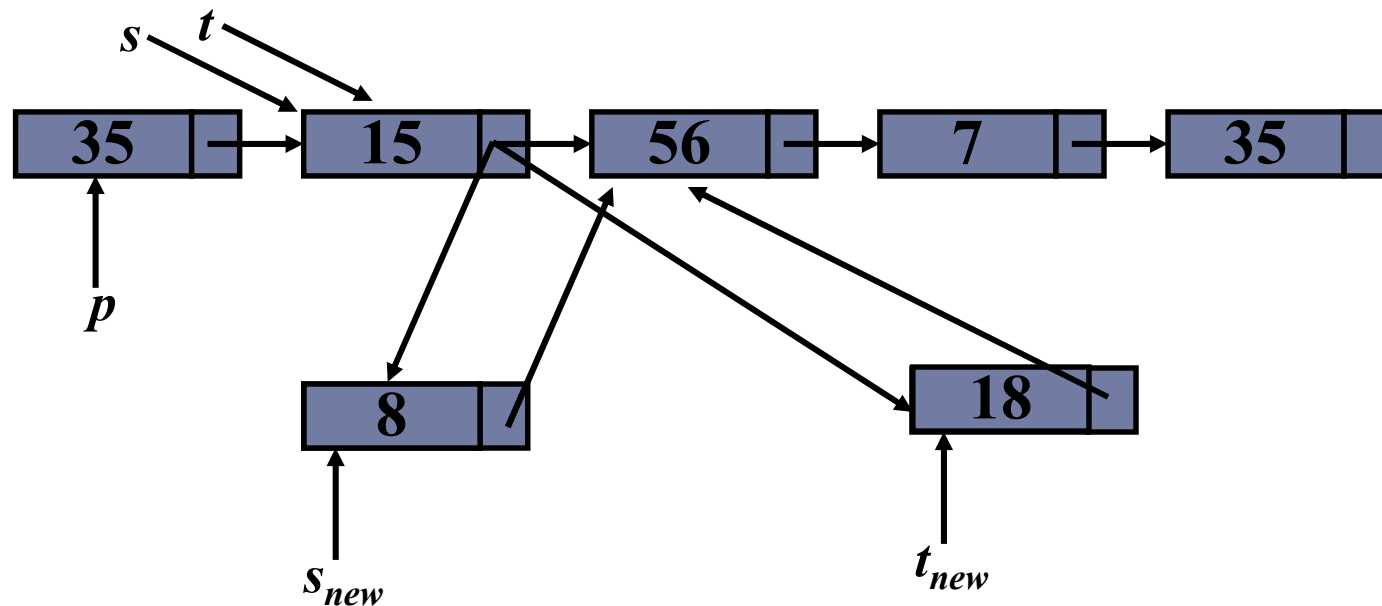
$t := p \rightarrow \text{next};$ // t points to the second element

$t_{new} \rightarrow \text{next} := t \rightarrow \text{next};$

$t \rightarrow \text{next} := t_{new};$

Example: Insertion into a linked list

并发执行



$s_{new} := \text{create a data block};$
 $s_{new} \rightarrow \text{data} := 8;$
 $s := p \rightarrow \text{next};$ // s points to the second element
 $s_{new} \rightarrow \text{next} := s \rightarrow \text{next};$
 $s \rightarrow \text{next} := s_{new};$

$t_{new} := \text{create a data block};$
 $t_{new} \rightarrow \text{data} := 18;$
 $t := p \rightarrow \text{next};$ // t points to the second element
 $t_{new} \rightarrow \text{next} := t \rightarrow \text{next};$
 $t \rightarrow \text{next} := t_{new};$

Operating System Concerns

- ▶ **Keep track of active processes (Chapter 3, 4)**
- ▶ **Allocate and deallocate resources**
 - ▶ Processor time (Part Four)
 - ▶ Memory (Part Three)
 - ▶ Files (Chapter 12)
 - ▶ I/O devices (Chapter 11)
- ▶ **Protect data and resources (Chapter 15)**
- ▶ ***Result of process must be independent of the speed of execution of other concurrent processes (This Chapter-Chapter 5)***



Types of Process Interaction

- ▶ **Based on the degree to which they are aware of each other's existence, the ways in which processes interact**
 - ▶ **Competition:** Processes unaware of each other
 - ▶ **Cooperation by sharing:** Processes indirectly aware of each other
 - ▶ **Cooperation by communication:** Process directly aware of each other



Competition for Resources (1)

Problem Definition

I/O Devices
Memory
Processor time
Clock

Two or more processes need to access a resource during the course of their execution.

- ▶ Each one is *unaware of* the existence of the other processes
- ▶ Each process is *unaffected* by the execution of the other processes
- ▶ There is no exchange of information between the competing processes

Each process should leave the state of any resource that it uses unaffected.

Competition for Resources (2)

Three Control Problems

- ▶ Mutual Exclusion

- ▶ Critical Resource (临界资源)
- ▶ Critical sections (临界区)
 - ▶ 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)

- ▶ A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

- ▶ Starvation (饿死)



Critical Resources

临界资源

- ▶ **Sharable and nonsharable resources**

- ▶ **If a resource can be accessed at the same time by several processes, this resources is called sharable (also called non-critical)**
 - ▶ Memory, disks
- ▶ **If a resource can be accessed by only one process at a time, it is called critical (also called nonsharable)**
 - ▶ Printer, some variables, data, tables and queues



Mutual Exclusion Mechanism in Abstract Terms

```
/* program mutualexclusion */  
const int n= /* number of processes */
```

```
void P(int i) {  
    while (true) {  
        entercritical(i);  
        /*critical section*/;  
        exitcritical(i);  
        /* remainder */;  
    }  
}
```

```
void main()  
{  
    parbegin (P (R1) , P (R2) , ... , P (Rn) ) ,  
}
```

Parbegin(P1, P2, ..., Pn) means the following:

- (1) Suspend the execution of the main program;
- (2) Initiate concurrent execution of procedures P1, P2, ..., Pn;
- (3) When all of P1, P2, ..., Pn have terminated, resume the main program

Entercritical(i) and exitcritical(i) are two functions provided to enforce mutual exclusion

Each one takes as an argument the name of resource that is the subject of competition

Any process attempt to enter its critical section while another process is in its critical section, for the same resource, is made to wait.

Cooperation by Sharing (1)

- ▶ **Multiple processes have access to shared data**
 - ▶ Shared variables or shared files or shared databases.
 - ▶ The control mechanism must **ensure the integrity** of the shared data
- ▶ **Writing must be mutually exclusive**
- ▶ **Critical sections are used to provide data integrity**



Cooperation by Sharing (2)

An Example for Data Coherence

- ▶ The relationship to be maintained between two variables is $a=b$
 - ▶ *Initial values:* $a=1$ and $b=1$
- ▶ Two Processes

P1 : $a=a+1;$ $b=b+1;$	P2 : $b=2*b;$ $a=2*a;$
---------------------------	---------------------------

Anything Wrong?

Cooperation by Communication

- ▶ **Communication provides a way to synchronize, or coordinate, the various activities.**
 - ▶ Typically, communication can be characterized as consisting of messages of some sort
 - ▶ Messages are passed (nothing is shared)
 - ▶ Mutual exclusion is not a control requirement
- ▶ **Possible to have deadlock**
 - ▶ Each process waiting for a message from the other process
- ▶ **Possible to have starvation**
 - ▶ Two processes sending message to each other while another process waits for a message



Requirements for Critical Section Problem Solution

- ▶ **Mutual exclusion must be enforced:** Only one process at a time is allowed in the critical section for a resource
- ▶ **Progress:** A process must not be delayed access to a critical section when there is no other process using it
- ▶ **Bounded waiting:** After a process has made a request and before that request is granted, there exists a bound or limit on the number of times that other processes are allowed to enter their critical sections



Ways to Satisfy the Requirements for Mutual Exclusion

- ▶ **Leave the responsibility with the process**
 - ▶ Software approach (section 5.2)→Overhead and Bugs
- ▶ **Involve the use of specific-purpose machine instructions**
 - ▶ Hardware support (section 5.3)→reduced overhead
- ▶ **Provide some level of support within the operating system or a programming language**
 - ▶ Semaphores (section 5.4)
 - ▶ Monitors (section 5.5)
 - ▶ Message passing (section 5.6)



Mutual Exclusion: Software Approaches

Assumption

- ▶ **Elementary mutual exclusion at the memory access level**
 - ▶ **Simultaneous accesses to the same location in main memory are serialized (串行化) by some sort of memory arbiter.**
- ▶ **Typical Software Approaches**
 - ▶ Dekker's Algorithm
 - ▶ Peterson's Algorithm



Dekker's Algorithm

- ▶ ***Used for mutual exclusion for **two** processes P0 and P1.***
- ▶ *Designed by Dutch Mathematician Dekker (reported by Dijkstra in 1965)*
- ▶ ***Here, the algorithm is developed in stages***
 - ▶ *To illustrate the common bugs in developing concurrent programs*



Dekker's Algorithm

First Attempt –Strict Alternation

- ▶ The value of a global variable `turn` indicates which process may proceed to its critical section
 - ▶ `turn=1` for P1 to enter the critical section; `turn=0` for P0
- ▶ **Busy Waiting**
 - ▶ Process is always checking to see if it can enter the critical section
 - ▶ Process can do nothing productive until it gets permission to enter its critical section



<pre>/* Process 0 */ ... while (turn!=0) /* do nothing */; /* critical section */; turn = 1; </pre>	<pre>/* Process 1 */ ... while (turn!=1) /* do nothing */; /* critical section */; turn = 0; </pre>
---	---

(a) First Attempt

Dekker's Algorithm

First Attempt

- ▶ ***This solution guarantees the mutual exclusion property***
- ▶ *Two drawbacks*
 - ▶ Processes must **strictly alternate** in their use of their critical section
 - ▶ More seriously, ***if one process fails (even outside the critical section), the other process is permanently block.***



Dekker's Algorithm

Second Attempt

- ▶ Each process can examine the other's status but cannot alter it
 - ▶ When a process wants to enter the critical section it checks the other processes first
 - ▶ If no other process is in the critical section, it sets its status for the critical section
- ▶ This method **does not guarantee mutual exclusion**
- ▶ Each process can check the flags and then proceed to enter the critical section at the same time



<pre> Enum boolean {false=0; true=1;}; boolean flag[2]={false, false}; </pre>	
<pre> /* Process 0 */ while (flag[1]) /* do nothing */; flag[0]=true; /* critical section */; flag[0]=false; </pre>	<pre> /* Process 1 */ while (flag[0]) /* do nothing */; flag[1]=true; /* critical section */; flag[1]=false; </pre>

(b) Second Attempt



Second Attempt

A Step Further – lock variable

- ▶ **Why bother using a boolean vector?**
 - ▶ **One boolean variable (lock variable) is enough!!**
 - ▶ **The shared lock variable is set to 0, initially**
 - ▶ **When a process wants to enter its critical section, it first tests the lock**
 - **If the lock is 0, the process sets it to 1 and enters the critical section**
 - **If the lock is 1, the process just waits until it becomes 0**
- ▶ **Still cannot guarantee the mutual exclusion**



Dekker's Algorithm

Third Attempt

- ▶ Set flag to enter critical section before check other processes
- ▶ If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section
- ▶ **Deadlock is possible** when two process set their flags to enter the critical section. Now each process must wait for the other process to release the critical section



<pre>/* Process 0 */ flag[0]=true; while (flag[1]) /* do nothing */; /* critical section */; flag[0]=false;</pre>	<pre>/* Process 1 */ flag[1]=true; while (flag[0]) /* do nothing */; /* critical section */; flag[1]=false;</pre>
--	--

(c) Third Attempt

Dekker's Algorithm

Fourth Attempt

- ▶ A process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag
- ▶ Other processes are checked. If they are in the critical region, the flag is reset and later set to indicate desire to enter the critical region. This is repeated until the process can enter the critical region.



```
/* Process 0 */
```

```
...
```

```
...
```

```
flag[0]=true;
```

```
while (flag[1])
```

```
{
```

```
    flag[0]=false;
```

```
    /* delay */;
```

```
    flag[0]=true;
```

```
}
```

```
/* critical section */;
```

```
flag[0]=false;
```

```
...
```

```
...
```

```
/* Process 1 */
```

```
...
```

```
...
```

```
flag[1]=true;
```

```
while (flag[0])
```

```
{
```

```
    flag[1]=false;
```

```
    /* delay */;
```

```
    flag[1]=true;
```

```
}
```

```
/* critical section */;
```

```
flag[1]=false;
```

```
...
```

```
...
```

(d) Fourth Attempt



Fourth Attempt

- ▶ It is possible for each process to set their flag, check other processes, and reset their flags. This scenario will not last very long so it is not deadlock. It is undesirable
- ▶ **Livelock**
 - ▶ *There are possible sequences of execution that succeed, but it is also possible to describe one or more execution sequences in which no process ever enters its critical section*



Dekker's Algorithm

Correct Solution

- ▶ To avoid “mutual courtesy” (相互谦恭) in the fourth attempt.
 - ▶ Use the `turn` variable from the first attempt
- ▶ Each process gets a turn at the critical section
- ▶ If a process wants the critical section, it sets its flag and may have to wait for its turn




```
boolean flag[2]={false, false};  
int turn=1;
```

```
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*/  
    }  
}
```

Peterson's Algorithm

- ▶ **Peterson's algorithm provides a simpler and more elegant solution than Dekker's**
- ▶ One process in critical section will prevent another from entering
 - ▶ P0 is in its critical section $\rightarrow \text{flag}[0] == \text{true}$
P1 is in its critical section $\rightarrow \text{flag}[1] == \text{true}$
- ▶ When two processes both want to enter critical section
 - ▶ The turn variable will serve as the arbiter



```
boolean flag[2]={false, false};  
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*/  
    }  
}
```

Mutual Exclusion: Hardware Support

Hardware Support

- ▶ **Disabling/Enabling Interrupts**
- ▶ **Machine Instructions**
 - ▶ Test and set instruction
 - ▶ Exchange instruction



Interrupt Disabling (1)

▶ **for Uniprocessor Machine**

- ▶ Concurrent processes can only be interleaved, and cannot be overlapped
- ▶ A process runs until it invokes an operating-system service or until it is interrupted
- ▶ Disabling interrupts guarantees mutual exclusion

▶ **Unsuitable for Multiprocessor**

- ▶ Disabling interrupts on one processor will not guarantee mutual exclusion



Interrupt Disabling (2)

- ▶ It is unwise to give user processes the power to turn off interrupts
 - ▶ Imagine that one process did it and never turned it on again!! → The end of the system
- ▶ Disabling interrupts is often a useful technique with the operating system itself
 - ▶ It is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists



Interrupt Disabling (3)

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



Special Machine Instructions

- ▶ **Access to a memory location excludes any other access to that same location**
- ▶ **Several machine instructions have been implemented to carry out two actions atomically**
 - ▶ Performed in a *single instruction cycle*
 - ▶ Not subject to interference from other instructions
- ▶ **Two commonly implemented instructions**
 - ▶ Test and Set Instruction
 - ▶ Exchange Instruction



“Test and Set” Instruction or TSL

- ▶ **Some computers designed with multiple processors in mind have an instruction like TSL RX, lock**
- ▶ **It reads the contents of the memory word lock into register RX, and then store a nonzero value at the memory address lock.**
- ▶ **The operations of reading and storing are guaranteed to be indivisible**
 - ▶ **The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done**

What is the difference between disabling interrupts and locking the memory bus?

“Test and Set” Instruction or TSL

```
boolean testset (int i) {  
    if (i == 0) {  
        i = 1;  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

*i=0 means the critical
section is open*

*i=1 means the critical
section is closed*

*What if this function is implemented in programming
language instead of a machine instruction?*



Mutual Exclusion with testset()

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

111

Busy Waiting!!!

“exchange” Instruction or XCHG

The **exchange** (or **XCHG**) instruction exchanges the contents of two locations atomically, for example: a register and a memory word

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



Mutual Exclusion with exchange ()

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

Busy Wating!!

The process that finds bolt equal to 0 is the only process that may enter its critical section

XCHG on Intel x86 CPUs

All Intel x86 CPUs use XCHG instruction for low-level synchronization

enter_region:

```
    MOVE REGISTER, #1    //put a 1 in the register
    XCHG REGISTER, LOCK  //swap the contents of the register and lock variable
    CMP REGISTER, #0     //was lock zero?
    JNE enter_region     //if it was non zero, lock was set, so loop
    RET                  //return to caller; critical region entered
```

Leave_region:

```
    MOV LOCK, #0         //store a 0 in lock
    RET                  //return to the caller
```



Advantages of Machine-Instruction Approach

- ▶ **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**



Disadvantages of Machine-Instruction Approach

- ▶ **Busy-waiting** consumes 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. Thus, some process could indefinitely be denied access
- ▶ **Deadlock 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 (**Priority Inversion** 优先级倒置)



Semaphores

信号量

Contents

- ▶ **What are semaphores?**
- ▶ **Several uses of semaphores**
 - ▶ Mutual exclusion
 - ▶ Producer/consumer problem
 - ▶ A barbershop problem
- ▶ **How to implement semaphores?**



Semaphores

- ▶ Special variable called a semaphore is used for signaling
- ▶ If a process is waiting for a signal, it is suspended until that signal is sent
- ▶ Wait and signal operations cannot be interrupted
- ▶ Queue is used to hold processes waiting on the semaphore



What is a semaphore?

- ▶ Semaphore is a **variable that has an integer value**, with **three operations** defined upon it.
 - ▶ A semaphore may be **initialized** to a nonnegative value
 - ▶ The **wait** operation decrements the semaphore value. If the value become negative, then the process executing the **wait** is blocked
 - ▶ The **signal** operation increments semaphore value. If the value is not positive, then a process blocked by a **wait** operation is unblocked



A Definition of Semaphore Primitives

```
struct semaphore {  
    int count;  
    queueType queue;  
}
```

```
void wait(semaphore s)  
{  
    s.count--;  
    if (s.count<0)  
    {  
        place this process in s.queue;  
        block this process;  
    }  
}
```

```
void signal(semaphore s)  
{  
    s.count++;  
    if (s.count<=0)  
    {  
        remove a process P from s.queue;  
        place process P on ready list;  
    }  
}
```



A Definition of Binary Semaphore (二元信号量) Primitives

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

void waitB(binary_semaphore s)
{
    if (s.value==1)
        s.value=0;
    else
    {
        place this process in s.queue;
        block this process;
    }
}

void signalB(binary_semaphore s)
{
    if (s.queue.is_empty())
        s.value=1;
    else
    {
        remove a process P from s.queue;
        place process P on ready list;
    }
}
```

If there are no processes waiting for the semaphore

Strong Semaphore vs. Weak Semaphore

强信号量vs弱信号量

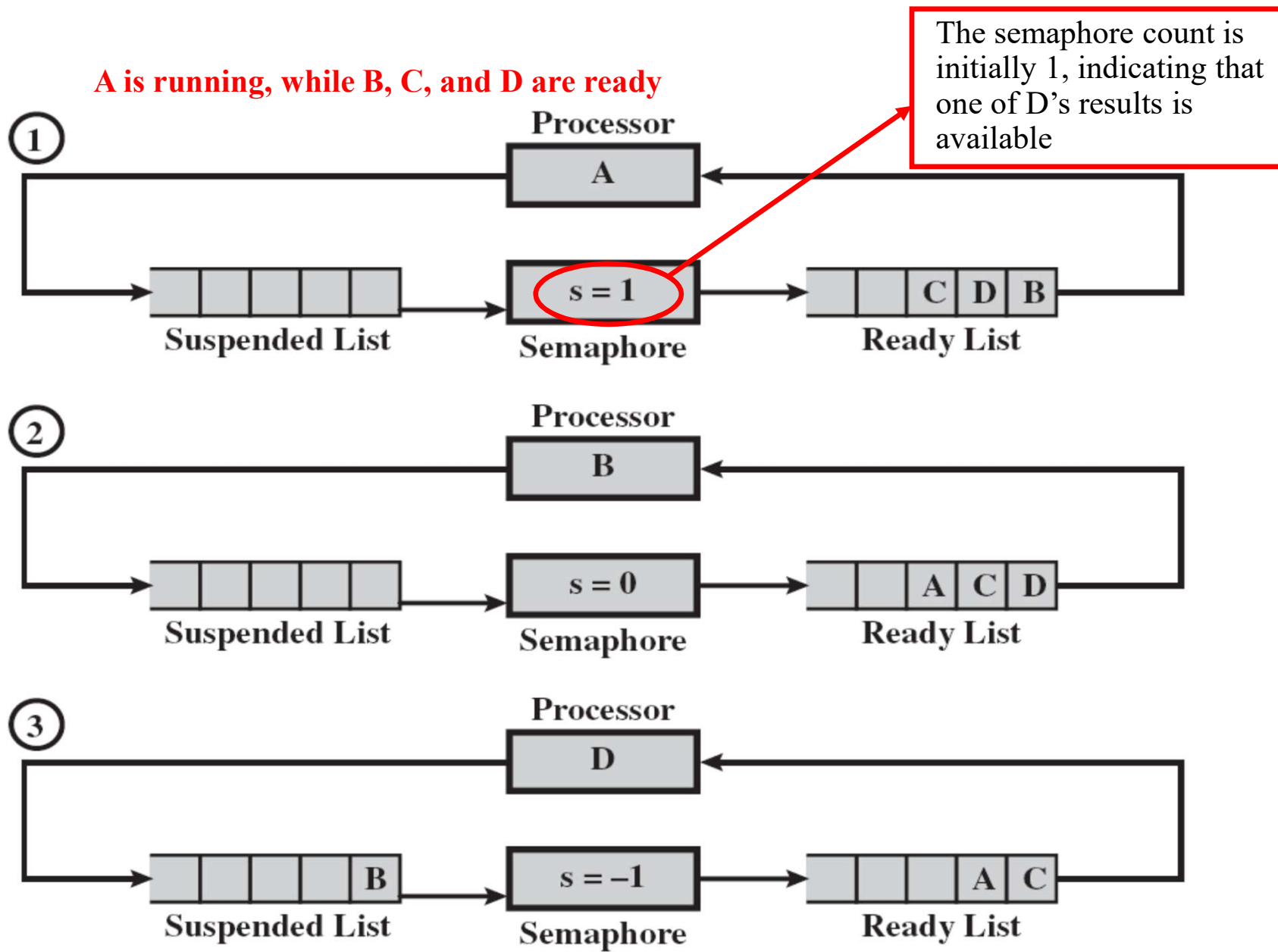
- ▶ **For both semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore**
 - ▶ How about the order in which processes are removed from such a queue?
- ▶ **Strong Semaphore: the process that has been blocked the longest is released from the queue first**
 - ▶ Typically provided by operating system, freedom from starvation
- ▶ **Weak Semaphore: does not specify the order in which processes are removed from the queue**
 - ▶ Weak semaphore may suffer from starvation.

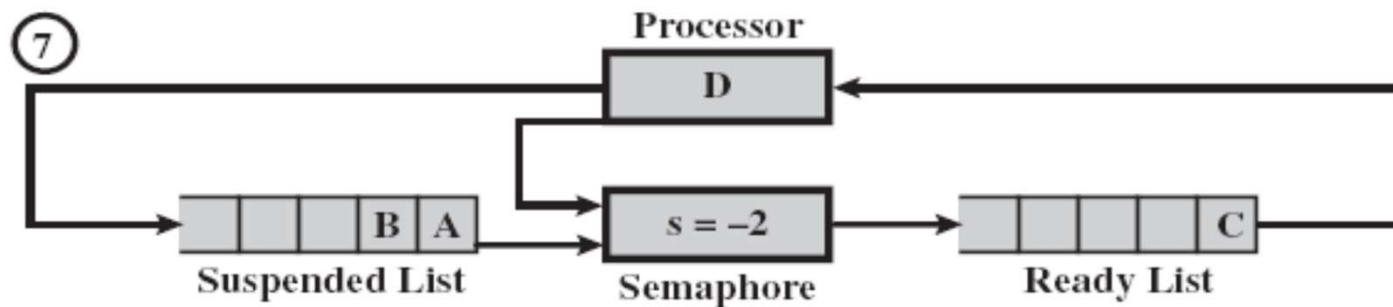
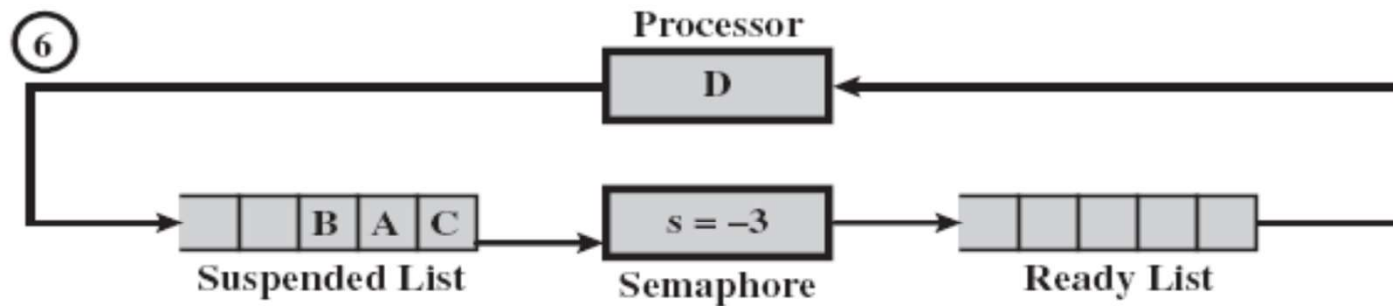
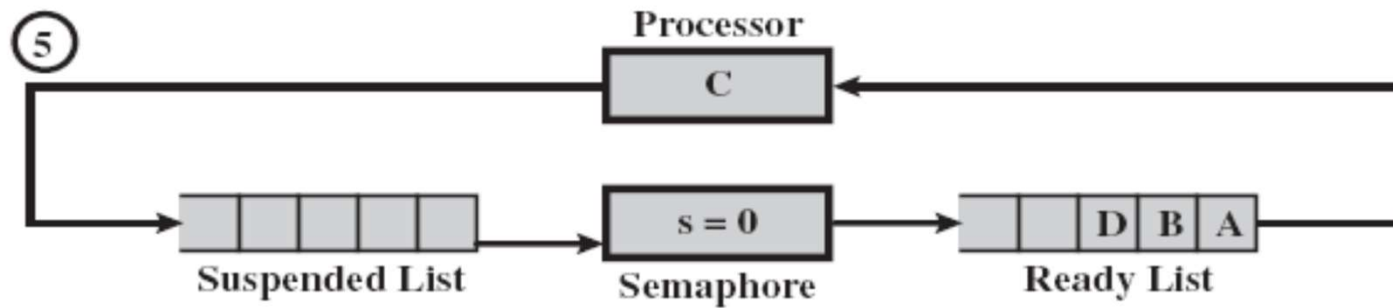
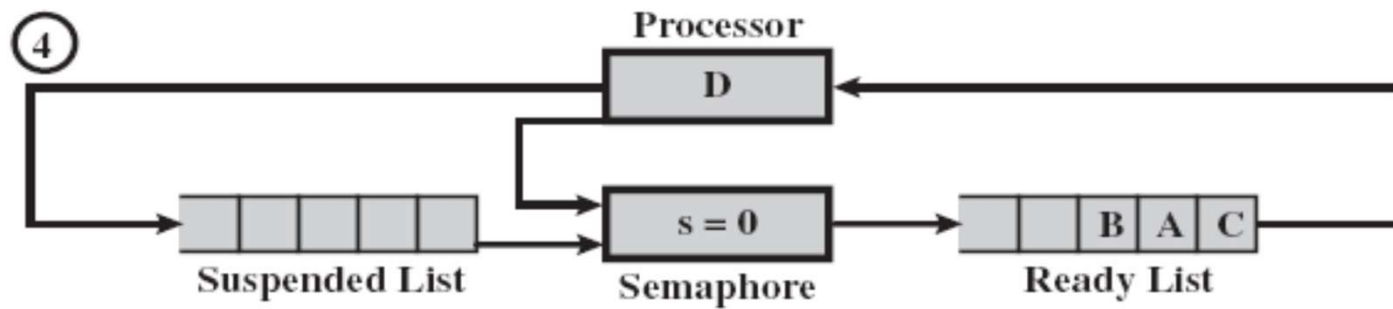


An Example of Semaphore Mechanism

- ▶ **There are four processes: A, B, C, and D**
 - ▶ Processes A, B, and C depend on a result from process D.








Mutual Exclusion (1)

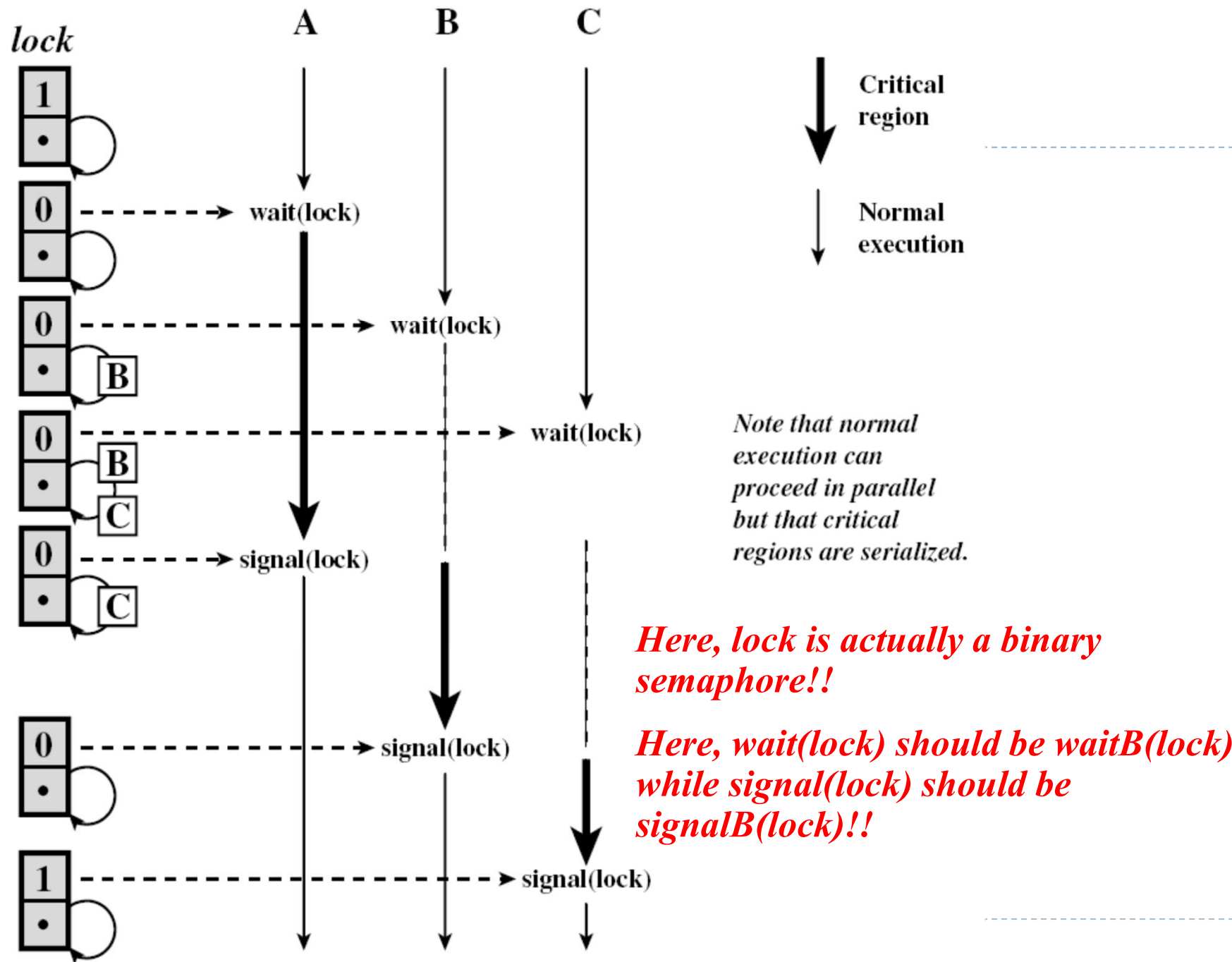
A Straightforward Solution

```
const int n=/*number of processes*/  
semaphore s=1;
```

```
void P(i){  
    while(true){  
        wait(s);  
        /*critical section*/  
        signal(s);  
        /* remainder */  
    }  
}
```

```
void main(){  
    parbegin(P(1), P(2), ..., P(n));  
}
```





Mutual Exclusion (2)

Interpretation of `s.count`

- ▶ `s.count ≥ 0`: `s.count` is the number of processes that can execute `wait(s)` without blocking
- ▶ `s.count < 0`: the magnitude of `s.count` is the number of processes blocked in `s.queue`

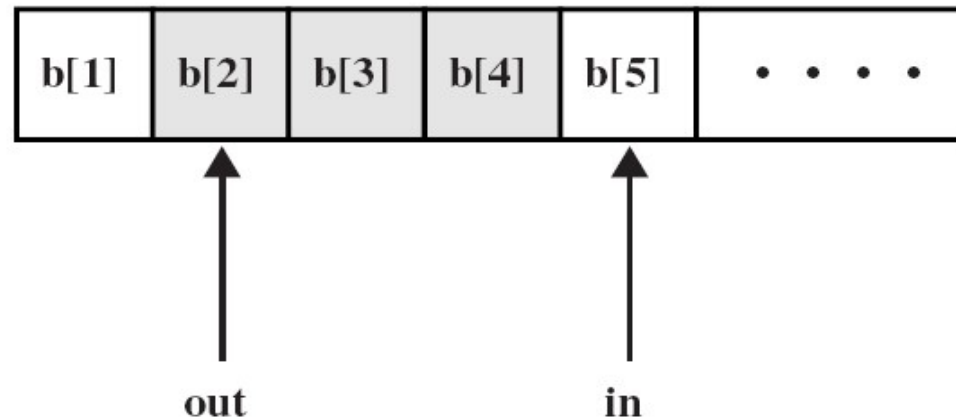


Producer/Consumer Problem

- ▶ **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



Infinite Buffer



Note: shaded area indicates portion of buffer that is occupied

Assume that the buffer is infinite and consists of a linear array of elements



Producer With Infinite Buffer

```
producer:
```

```
while (true) {
```

```
    /* produce item v */
```

```
    b[in] = v;
```

```
    in++;
```

```
}
```

***b** is the buffer
in is the next empty element in the buffer*



Consumer with Infinite Buffer

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

**Make sure that producers
has already advanced in
beyond out.**



A Incorrect Solution with Binary Semaphores (Infinite Buffer)

<code>int n; binary_Semaphore s=1; binary_semaphore delay=0;</code>	
<code>void producer() { while(true) { produce(); waitB(s); append(); n++; if (n==1) signalB(delay); signalB(s); } }</code>	<code>void consumer() { waitB(delay); while(true) { waitB(s); take(); n--; signalB(s); consume(); if (n==0) waitB(delay); } }</code>
<code>void main() { n=0; parbegin(producer, consumer); }</code>	



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

A Correct Solution with Binary Semaphores (Infinite Buffer)



A Solution with General Semaphores (Infinite Buffer)

Semaphore n=0; Semaphore s=1;	
<pre>void producer() { while(true) { produce(); wait(s); append(); signal(s); signal(n); } }</pre>	<pre>void consumer() { while(true) { wait(n); wait(s); take(); signal(s); consume(); } }</pre>
<pre>void main() { parbegin(producer, consumer); }</pre>	



Finite Circular Buffer

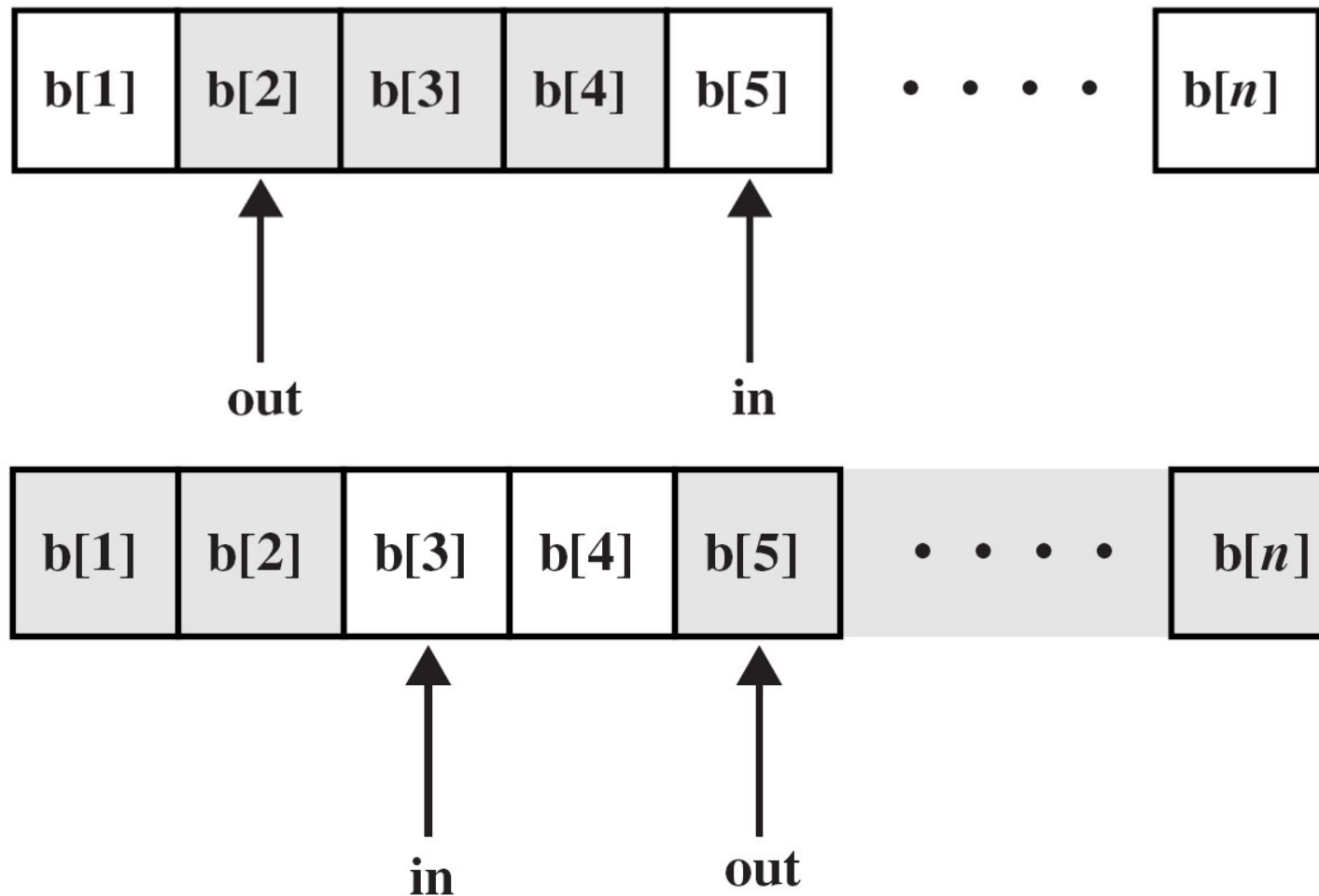


Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

Producer with Finite Circular Buffer

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



Consumer with Finite Circular Buffer

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



A Solution with Semaphores (Finite Circular Buffer)

<pre>const int sizeofbuffer=/*buffer size*/; semaphore s=1; semaphore n=0; semaphore e=sizeofbuffer;</pre>	
<pre>void producer(){ while(true){ produce(); wait(e) wait(s); append(); signal(s); signal(n); } }</pre>	<pre>void consumer(){ while(true){ wait(n); wait(s) take(); signal(s); signal(e); consume(); } }</pre>

sizeofbuffer or (sizeofbuffer-1)?



Implementation of Semaphores

- ▶ *It is imperative to implement the wait and signal operations as **atomic primitives** (原语).*
 - ▶ *How?*
- ▶ *The essence of semaphore implementations*
 - ▶ *Only one process at a time may manipulate a semaphore with either a wait or signal operation.*
- ▶ *Thus, Possible implementations include:*
 - ▶ *In a uniprocessor system, disabling interrupts*
 - ▶ *In a multiprocessor, hardware approaches (*testset* or *exchange*) or software approaches (Dekker's or Peterson's algorithm)*



Semaphore Implementation by Disabling Interrupts

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

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

*For a **single-processor** system, the relatively **short duration of these operations** means that this approach is reasonable.*



Semaphore Implementation by the testset instruction

The semaphore s now includes a new integer component s.flag

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

```
signal(s){
  while(testset(s.flag))
    /* do nothing */
  s.count++;
  if (s.count<=0){
    remove a process from s.queue;
    place this process on ready list;
  }
  s.flag=0;
}
```



Semaphore Implementation Summary

- ▶ **The implementations of semaphores have not completely eliminate the busy-waiting**
 - ▶ They are implemented by hardware approaches or software approaches.
- ▶ ***What is the advantages of semaphore, compared to the software or hardware approaches?***



Barbershop Problem: Description (1)

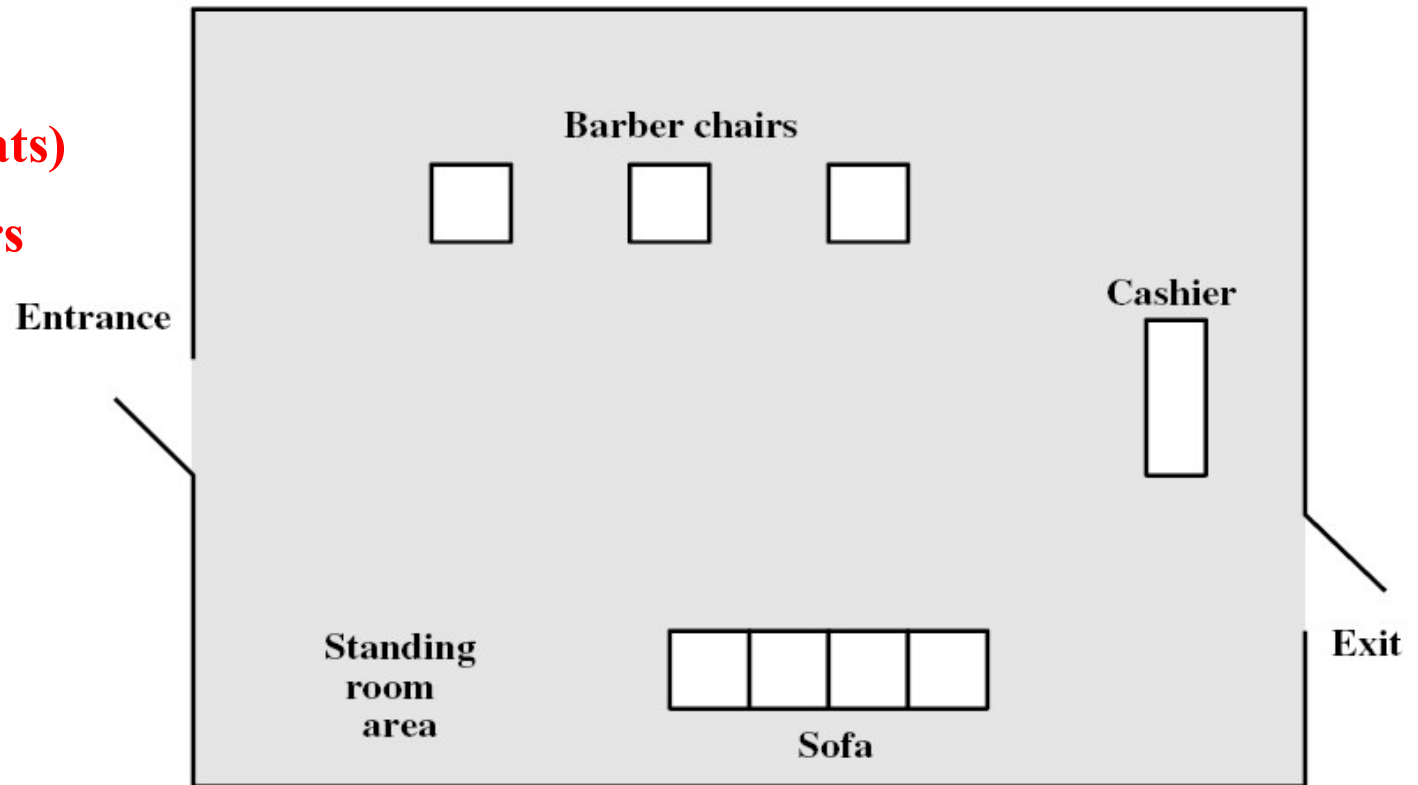
Three Barber Chairs

Three Barbers

A Sofa (with four seats)

**At most 20 customers
in the barbershop**

One cash register



Barbershop Problem: Description (2)

- ▶ A customer will not **enter the shop** if it is filled to the capacity with other customers
- ▶ Once inside, the customer **takes a seat on the sofa** or stands if the sofa is filled
- ▶ When a barber is free, the customer that has been on the sofa the longest is served
 - ▶ If there are any standing customers, the one that has been in the shop the longest takes a seat on sofa
- ▶ When a customer's haircut is finished, payment is accepted for one customer at a time
 - ▶ The barbers divide their time among cutting hair, accepting payment, and sleeping in their chair waiting for a customer.




```

void customer() {
    wait(max_capacity);
    enter_shop();
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair;
    signal(cust_ready);
    wait(finished);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity);
}

```

```

void barber() {
    while(true) {
        wait(cust_ready);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}

```

```

void cashier() {
    while(true) {
        wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}

```



Semaphore	Wait Operation	Signal Operation
<i>max_capacity</i>	Customer waits for space to enter shop.	Exiting customer signals customer waiting to enter.
<i>sofa</i>	Customer waits for seat on sofa.	Customer leaving sofa signals customer waiting for sofa.
<i>barber_chair</i>	Customer waits for empty barber chair.	Barber signals when that barber's chair is empty.
<i>cust_ready</i>	Barber waits until a customer is in the chair.	Customer signals barber that customer is in the chair.
<i>finished</i>	Customer waits until his haircut is complete.	Barber signals when done cutting hair of this customer.
<i>leave_b_chair</i>	Barber waits until customer gets up from the chair.	Customer signals barber when customer gets up from chair.
<i>payment</i>	Cashier waits for a customer to pay.	Customer signals cashier that he has paid.
<i>receipt</i>	Customer waits for a receipt for payment.	Cashier signals that payment has been accepted.
<i>coord</i>	Wait for a barber resource to be free to perform either the hair cutting or cashiering function.	Signal that a barber resource is free.

```
void customer() {  
    wait(max_capacity);  
    enter_shop();  
    wait(sofa);  
    sit_on_sofa();  
    wait(barber_chair);  
    get_up_from_sofa();  
    signal(sofa);  
    sit_in_barber_chair;  
    signal(cust_ready);  
    wait(finished);  
    leave_barber_chair();  
    signal(leave_b_chair);  
    pay();  
    signal(payment);  
    wait(receipt);  
    exit_shop();  
    signal(max_capacity);  
}
```

```
void barber() {  
    while(true) {  
        wait(cust_ready);  
        wait(coord);  
        cut_hair();  
        signal(coord);  
        signal(finished);  
        wait(leave_b_chair);  
        signal(barber_chair);  
    }  
}
```

```
void cashier() {  
    while(true) {  
        wait(payment);  
        wait(coord);  
        accept_pay();  
        signal(coord);  
        signal(receipt);  
    }  
}
```

An Unfair Barbershop

Why Unfair?

- ▶ It is possible that there are three customers blocked on **wait (finished)**
- ▶ They will be released in the order they sat in the barber chair
- ▶ *What if one of the barbers is very fast, or one of the customers is quite bald?*
- ▶ *The same problem exists for the cashier!!*



```

void customer(){
    int custnr;
    wait(max_capacity);
    enter_shop();
    wait(mutex1);
    count++;
    custnr=count;
    signal(mutex1);
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair();
    wait(mutex2);
    enqueue1(custnr);
    signal(cust_ready);
    signal(mutex2);
    wait(finished[custnr]);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity);
}

```

```

void barber(){
    int b_cust;
    while(true){
        wait(cust_ready);
        wait(mutex2);
        dequeue1(b_cust);
        signal(mutex2);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished[b_cust]);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}

```

```

void cashier(){
    while(true){
        wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}

```

*A Fair
Barbershop*

Monitors

管程

Monitors

- ▶ **This concept was proposed by Brinch Hansen (1973) and Hoare, C (1974).**
 - ▶ To make it easier to write correct programs
- ▶ **The monitor construct has been implemented in a number of programming languages**
 - ▶ Concurrent Pascal
 - ▶ Pascal-Plus
 - ▶ Modula-2
 - ▶ Modula-3
 - ▶ Java



What are Monitors?

- ▶ **Monitor is a software module, consisting of**
 - ▶ *One or more procedures*
 - ▶ *An initialization sequence*
 - ▶ *Local data*
- ▶ **Chief characteristics**
 - ▶ *Local data variables are accessible only by the monitor*
 - ▶ *Process enters monitor by invoking one of its procedures*
 - ▶ ***Only one process may be executing in the monitor at any instance***



Mutual Exclusion Enforced by the Compiler

- ▶ It is up to the **compiler** to implement mutual exclusion on monitor entries
 - ▶ The compiler can commonly use a semaphore
- ▶ It is much less likely that something will go wrong, because
 - ▶ The compiler (not the programmer) is arranging for the mutual exclusion.



Mutual Exclusion Facility Provided by Monitor

- ▶ **The monitor enforces an important property:**
 - ▶ Only one process can be active in a monitor at any instance.
- ▶ **The monitor provides a mutual exclusion facility for accessing its data variables.**



Mutual Exclusion Facility Provided by Monitor

```
monitor anADT {
private:
    semaphore mutex = 1;
    <ADT data structures>
    ...
public:
    proc_i(...) {
        wait(mutex);
        <processing for proc_i>
        if urgentcount>0 then
            signal(urgent);
        else
            signal(mutex);
    };
    ...
};
```

*For each monitor, a semaphore **mutex** is used to ensure that the bodies of the local procedures exclude each other.*

Condition Variables

- ▶ Although monitors provide an easy way to achieve mutual exclusion, that is not enough
 - ▶ We need a way for processes to block when they discover that they can not proceed
- ▶ The solution lies in the introduction of **condition variables**, along with **two operations** on it.



Condition Variables – Synchronization Supported by Monitor (1)

- ▶ **Condition variables are contained within the monitor and accessible only within the monitor**
- ▶ **Two functions operate on condition variables**
 - ▶ **`cwait(c)`: block 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 processes, do nothing**



Condition Variables – Synchronization Supported by Monitor (2)

- ▶ **A condition variable is neither true nor false**
 - ▶ It does not have any stored value accessible to the program
 - ▶ In practice, a conditional variable will be represented by a queue of processes



Condition Variables – Synchronization Supported by Monitor (3)

- ▶ When a process signals a condition on which another process is waiting, the **signaling process must wait** until the resumed process permits it to proceed
- ▶ Thus, a second semaphore (`urgent`) is introduced (It is initialized to be 0)
 - ▶ Signaling processes suspend themselves by the operation `wait(urgent)`.
- ▶ Before releasing exclusion, each process must test whether any other process is waiting on `urgent`
 - ▶ If so, it will release it by `signal(urgent)` instead of `signal(mutex)`
 - ▶ Thus, we therefore need to count the number of processes waiting on `urgent` with an integer “`urgentcount`”



Condition Variables – Synchronization Supported by Monitor (4)

- ▶ For each condition variable, we introduce **a semaphore “condsem” (initialized to 0)**, on which a process desiring to wait suspends itself by a `wait(condsem)` operation
- ▶ Since a process signaling this condition needs to know whether anybody is waiting, we also need a count of the number of waiting process (in an integer `condcount`)



"cwait" operation:

```
condcount:=condcount+1;  
if urgentcount>0 then  
    signal(urgent)  
else  
    signal(mutex) ;  
wait(condsem) ;  
condcount:=condcount-1;
```

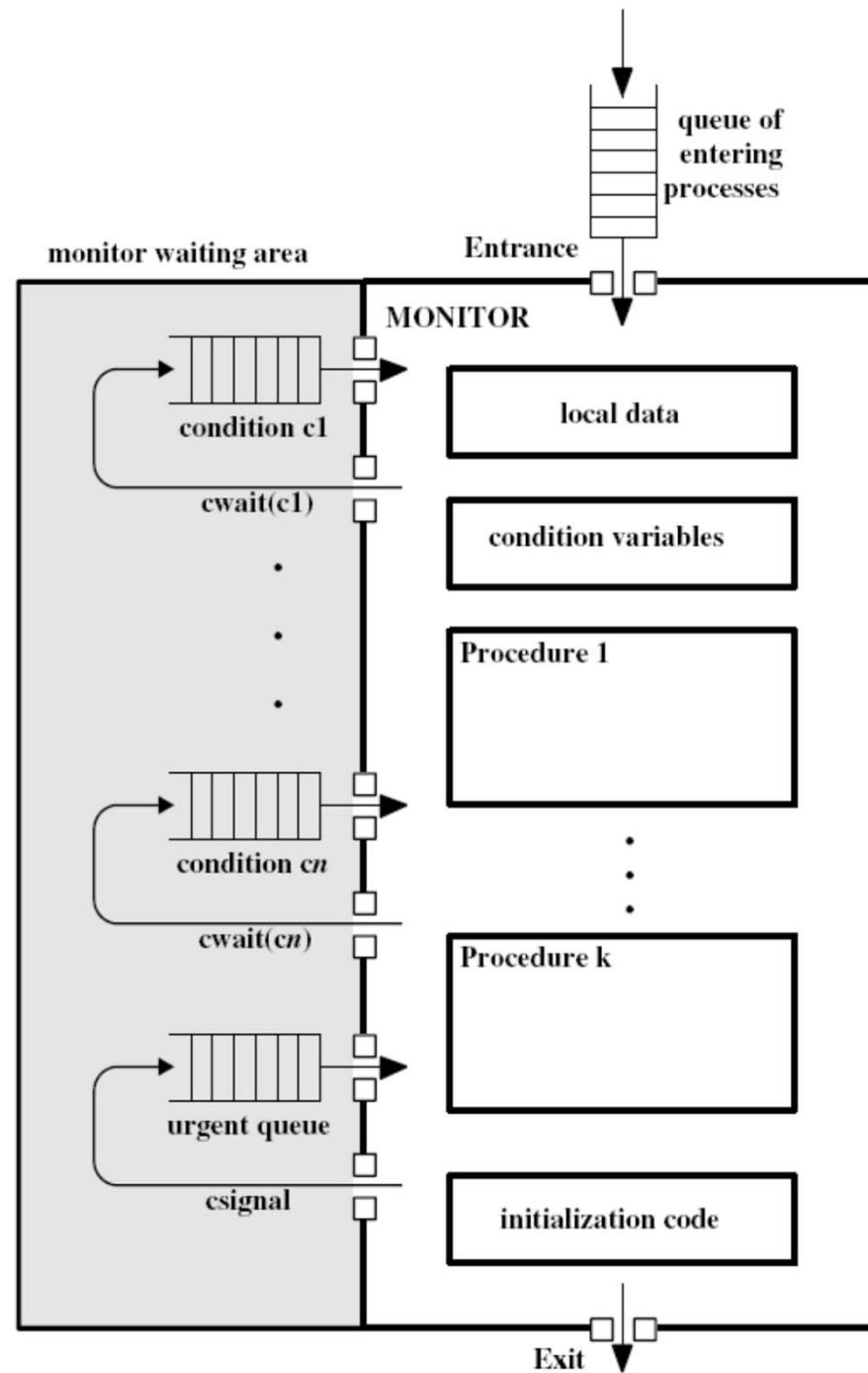
"csignal" operation:

```
urgentcount:=urgentcount+1;  
if condcount>0 then {  
    signal(condsem) ;  
    wait(urgent) ;  
}  
urgentcount:=urgentcount-1;
```

wait/signal Operations Compared with Semaphores

- ▶ **In a monitor, if a process signals and no task is waiting on the condition variable, the signal is lost**





```

monitor boundedbuffer;
char buffer[N];
int nextin,nextout;
int count;
cond notfull,notempty;

void append(char x){
    if (count==N)
        cwait(notfull);
    buffer[nextin]=x;
    nextin=(nextin+1)%N;
    count++;
    csignal(notempty);
}

void take(char x){
    if (count==0)
        cwait(notempty);
    x=buffer[nextout];
    nextout=(nextout+1)%N;
    count--;
    csignal(notfull);
}

{ nextin=0; nextout=0;
  count=0;
}

```

```

void producer(){
    char x;
    while (true){
        produce (x);
        append(x);
    }
}

```

```

void consumer(){
    char x;
    while (true){
        take(x);
        consume(x);
    }
}

```

***A Solution to the Bounded-Buffer
Producer/Consumer problem using a
Monitor***

Comments on Hoare's Definition of Monitors

- ▶ **Hoare's definition of monitors has the following requirement:**
 - ▶ If there is at least one process in a condition queue, a process from that queue runs immediately when another process issues a `csignal` for that condition
- ▶ **Two drawbacks**
 - ▶ Two additional process switches are required
 - ▶ The signaling mechanism should be perfectly reliable



Avoidance of Having Two Active Processes in the Monitor

- ▶ **The operation `csignal()` can unblock a process waiting on a condition variable**
- ▶ **There are three choices to avoid two active process in the monitor at the same time**
 - ▶ **Hoare:** Letting the newly unblocked process run, and block the other one (As described above)
 - ▶ **Brinch Hansen:** the process doing a `csignal()` must exit the monitor immediately
 - ▶ **Lampson and Redell:** let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor



Brinch Hanson' Monitor

- ▶ **Brinch Hanson proposed that a process doing a `csignal()` must exit the monitor immediately**
 - ▶ In other words, a `csignal()` statement may appear only as the final statement in a monitor procedure.



Monitor with Notify and Broadcast (1)

- ▶ **Lampson and Redell developed a different definition of monitors for the language Mesa**
 - ▶ also used in Modula-3
- ▶ **In Mesa, the `csignal` primitive is replaced by `cnotify`, with the following interpretation:**
 - ▶ When a process executing in a monitor executes `cnotify(x)`, it causes the `x` condition queue to be notified, but the signaling process continues to execute.



Monitor with Notify and Broadcast (2)

- ▶ **cbroadcast primitive**
 - ▶ The broadcast causes all processes waiting on a condition to be place in a Ready state
- ▶ **When it is useful?**
 - ▶ When a process does not know how many other processes should be reactivated
 - ▶ For example, a memory manager has j bytes free; a process frees up an additional k bytes. However, the process does not know which waiting process can proceed with a total of $j+k$ bytes



```
void append(char x){
    while (count==N)
        cwait(notfull);
    buffer[nextin]=x;
    nextin=(nextin+1)%N;
    count++;
    cnotify(notempty);
}

void take(char x){
    while (count==0)
        cwait(notempty);
    x=buffer[nextout];
    nextout=(nextout+1)%N;
    count--;
    cnotify(notfull);
}
```

A Solution to the Bounder-Buffer Producer/Consumer problem using the Monitor with cnotify

Message Passing

消息传递

Message Passing

- ▶ **Message passing can be used to**
 - ▶ Enforce mutual exclusion/synchronization
 - ▶ Exchange information
- ▶ **Message passing is applicable in a multicomputer environment.**



Two Primitives

- ▶ **Message passing mechanism is usually provided by OS as a pair of primitives:**
 - ▶ `send(destination, message)`
 - ▶ It sends a message to a given destination
 - ▶ `receive(source, message)`
 - ▶ It receives a message from a given source (or from **ANY**, if the receiver does not care).
 - If no message is available, the receiver can block until one arrives (**blocking**), or
 - Alternatively, it can return immediately with an error code (**non-blocking**)



Design Issues Relating to Message-passing systems

- ▶ **Addressing (寻址)**
 - ▶ Direct (receive: explicit/implicit)
 - ▶ Indirect (static/dynamic/ownership)
- ▶ **Synchronization (同步)**
 - ▶ Send (blocking/nonblocking)
 - ▶ Receive (blocking/nonblocking)
- ▶ **Format (消息格式)**
- ▶ **Queuing Discipline (FIFO/Priority) 排队原则**



Addressing

寻址

- ▶ **It is necessary to specify which process is to receive the message in the send primitive;**
 - ▶ Similarly, most implementations allow a receiving process to indicate the source of a message to be received
- ▶ **Two categories of the schemes for specifying processes in send and receive primitives**
 - ▶ **Direct Addressing (直接寻址)**
 - ▶ **Indirect Addressing (间接寻址)**



Direct Addressing - I

Symmetric (对称) Scheme

- ▶ **With symmetric direct addressing, both the sender and the receiver processes must name the other to communicate**
 - ▶ `send(P, message)` – Send a message to process P
 - ▶ `receive(Q, message)` – Receive a message from process Q
- ◆ **A link is associated with exactly two processes**
 - ◆ It could know ahead of time from which process a message is expected (often be *effective for cooperating concurrent processes*)



Direct Addressing – II

Asymmetric Scheme

- ▶ **Only the sender names the recipient, the recipient is not required to name the sender**
 - ▶ `send(P, message)` – Send a message to process P
 - ▶ `receive(id, message)` – Receive a message from any process; the id is set to the name of the sender
- ▶ **For example, a printer-server process**



Indirect Addressing - I

- ▶ ***Messages are sent to a shared data structure consisting of queues***
 - ▶ Such queues are generally called **mailboxes**
 - ▶ One process sends a message to the mailbox and the other process picks up the message from the mailbox
 - ▶ Each mailbox has a unique identification
 - ▶ A process can communicate with some other processes via a number of different mailboxes
- ▶ ***Benefit of indirect addressing: more flexibility***
 - ▶ *The relationship between senders and receivers can be one-to-one, many-to-one, one-to-many, or many-to-many.*



Indirect Addressing - II

- ▶ send(A,message)
- ▶ Receive(A, message)

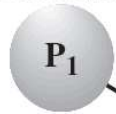


Indirect Addressing - II

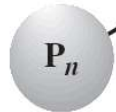
- ▶ **The ownership of a mailbox (信箱的所有权)**
 - ▶ **Process mailbox ownership**
 - ▶ Only the owner process may receive messages from the mailbox, and other processes may send to this mailbox
 - ▶ Mailbox can be created with the process and destroyed when the process dies (or through separate `create_mailbox` and `destroy_mailbox` calls)
 - ▶ **System mailbox ownership**
 - ▶ mailboxes have their own independent existence, not attached to any process
 - ▶ dynamic connection to a mailbox by processes (for send and/or receive)



Sending
Processes



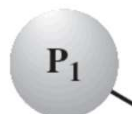
⋮



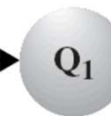
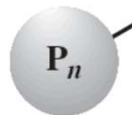
Receiving
Processes



⋮



⋮



*Ports are often statically associated
with a particular process.*

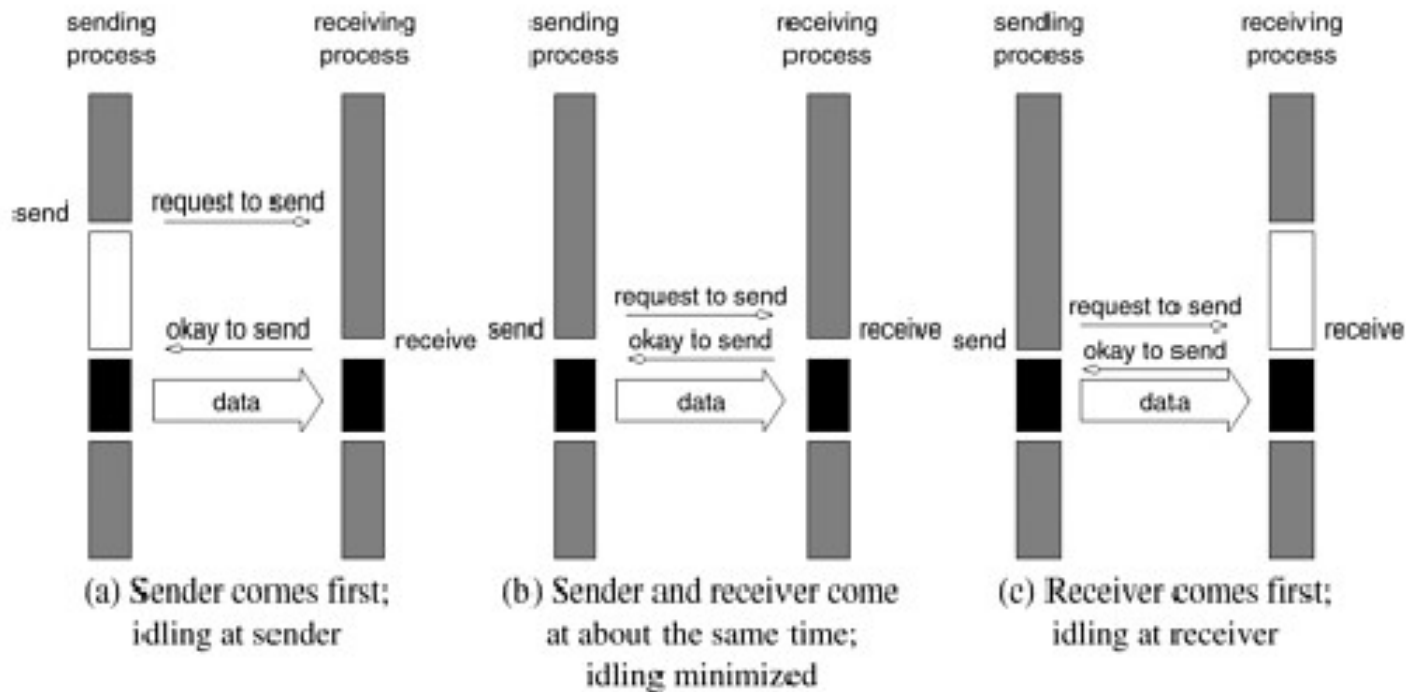
*That is, the port is created and
assigned to the process permanently*

Synchronization (1)

- ▶ **The receiver cannot receive a message until it has been sent by another process.**
- ▶ **How to specify what happens to a process after it issues a send or receive primitive**
 - ▶ For send primitive, either the sending process is blocked until the message is received **by the receiving process or by the mailbox**, or it is not
 - ▶ For receive primitive
 - ▶ If a message has previously been sent, the message is received and execution
 - ▶ If there is no waiting message, the process is either blocked, or abandons the attempt and retrieves a null.
- ▶ **Sender and receiver may or may not be blocking**



Blocking Send() without Buffer



Synchronization (2)

- ▶ **Blocking send, blocking receive**

- ▶ Both sender and receiver are blocked until message is delivered
- ▶ Called a rendezvous (回合制)

- ▶ **Nonblocking send, blocking receive**

- ▶ Sender continues processing such as sending messages as quickly as possible
- ▶ Receiver is blocked until the requested message arrives

- ▶ **Nonblocking send, nonblocking receive**

- ▶ Neither party is required to wait



Message Buffering

消息缓冲

- ▶ ***Whether direct or indirect addressing, there is a queue for exchanged messages. Such a queue can be implemented in three ways:***
 - ▶ *Zero capacity: the sender must block until the recipient receive the message*
 - ▶ *Bounded capacity: the queue has finite length n ; full or not-full?*
 - ▶ *Unbounded capacity: the queue length is potentially infinite. The sender never blocks*
- ▶ ***Mailboxes make the buffering mechanism clear:***
 - ▶ *The destination mailbox holds messages that have been sent but have not yet been accepted*



Mutual Exclusion Using Messages

```
/* 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));
}
```

**Blocking receive primitive, and
non-blocking send primitive**

**This mailbox can be used by all
processes to send and receive**

**Initialized to contain a single
message with null content**



Solution to the Bounded-Buffer Producer-Consumer Problem

```
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);
    }
}
```

```
const int
    capacity=buffering capacity;
    null=empty message;

void main() {
    create_mailbox(mayproduce);
    create_mailbox(mayconsume);
    for (i=1; i<=capacity; i++)
        send(mayproduce, null);
    parbegin(producer, consumer);
}
```



Readers/Writers Problem

Readers/Writers Problem

- ▶ A number of processes that only read the data area (**readers**), and a number of processes that only write the data area (**writers**)
- ▶ **Conditions**
 - ▶ *Any number of readers may simultaneously read the data area*
 - ▶ *Only one writer at a time may write to the data area*
 - ▶ *If a writer is writing to the data area, no reader may read it*



Comparison with Other Problems

- ▶ **With mutual exclusion**
 - ▶ General mutual exclusion is too restrictive!
 - ▶ Efficiency is not enough!!!!
- ▶ **With producer/consumer**
 - ▶ The producer is not just a writer. It must read the queue pointers to determine where to write the next item and whether the buffer is full
 - ▶ Similarly, the consumer is not just a reader!



Readers Have Priority

- ▶ **One integer variable readcount**
 - ▶ How many readers are currently reading
- ▶ **Two semaphores**
 - ▶ **x – mutual exclusion for the variable readcount**
 - ▶ **wsem –**
 - ▶ **a mutual exclusion for the writers**
 - ▶ **For readers, only be used by the first reader to enter and the last read to leave.**



```
int readcount;
semaphore x=1, wsem=1;
```

```
void reader(){
    while (true){
        wait(x);
        readcount++;
        if (readcount==1)
            wait(wsem);
        signal(x);
        READUNIT();
        wait(x);
        readcount--;
        if (readcount==0)
            signal(wsem);
        signal(x);
    }
}
```

```
void writer(){
    while(true){
        wait(wsem);
        WRITEUNIT();
        signal(wsem);
    }
}
```

```
void main(){
    readcount=0;
    parbegin(reader, writer);
}
```

Readers Have Priority

Problem with the first solution

- ▶ Once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading
- ▶ Thus, writers are subject to **starvation**



```
int readcount, writecount;
semaphore x=1, y=1, z=1,
        wsem=1, rsem=1;
```

```
void reader(){
    while (true){
        wait(z);
        wait(rsem);
        wait(x);
        readcount++;
        if (readcount==1)
            wait(wsem);
        signal(x);
        signal(rsem);
        signal(z);
        READUNIT();
        wait(x);
        readcount--;
        if (readcount==0)
            signal(wsem);
        signal(x);
    }
}
```

```
void writer(){
    while(true){
        wait(y);
        writecount++;
        if (writecount==1)
            wait(rsem);
        signal(y);
        wait(wsem);
        WRITEUNIT();
        signal(wsem);
        wait(y);
        writecount--;
        if (writecount==0)
            signal(rsem);
        signal(y);
    }
}
```

```
void main(){
    readcount=writecount=0;
    parbegin(reader, writer);
}
```

Writers Have Priority

A Solution Using Message Passing (1)

- ▶ **Controller process has access to the shared data area**
 - ▶ It has three mailboxes, one for each type of message that it may receive.
- ▶ **Other processes wishing to access the data area send a request message to the controller, are granted access with an “OK” reply message, and indicate completion of access with a “finished” message.**



A Solution Using Message Passing

(2)

▶ The variable count

- ▶ **Initialized** to some number greater than the maximum possible number of readers (Here, we use a value of 100)
- ▶ When a reader is allowed to read, the count is decremented by 1 (when a “finished” from a reader is received, the count is increased by 1)

▶ Several situations

- ▶ If $\text{count} > 0$, then no writer is waiting, and there may or may not be readers active.
- ▶ If $\text{count} = 0$, then the only request is a writer request. Allow the writer to proceed and wait for a “finished” message
- ▶ If $\text{count} < 0$, then a writer has made a request and is being made to wait to clear all active readers



```

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++;
        }
    }
}

```

Summary (1)

- ▶ **Central themes of modern operating systems**
 - ▶ Multiprogramming, multiprocessing, distributed processing
- ▶ **Concurrency is fundamental**
 - ▶ Concurrent processes may interact in a number of ways
 - ▶ The key issues in these interactions are mutual exclusion and deadlock



Summary (2)

- ▶ **What is Mutual exclusion?**
 - ▶ There is a set of concurrent processes
 - ▶ Only one of them is able to access a given resource or perform a given function at any time.
- ▶ **Mutual exclusion techniques**
 - ▶ Used to resolve conflicts (such as competing for resources)
 - ▶ Used to synchronize processes so that they can cooperate (for example: the producer/consumer model)



Summary (3)

- ▶ **Software algorithms for providing mutual exclusion**
 - ▶ Dekker's algorithm
 - ▶ Peterson's algorithm
- ▶ **Hardware support**
 - ▶ Test-And-Set
 - ▶ Exchange
- ▶ **Approaches supported by OS or programming languages**
 - ▶ Semaphores
 - ▶ monitors
 - ▶ Message passing

