# 目录 Contents





# Concurrency: Mutual Exclusion and Synchronization

## 4.1 Principles of Concurrency

### 1 Concurrency Overview

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Concurrency Arises in Three Different Contexts:
  - Multiple Applications: Multiprogramming was invented to allow processing time to be shared among active applications
  - Structured Applications: extension of modular design and structured programming
  - Operating System Structure: OS themselves implemented as a set of processes or threads



## Concurrency Overview.

- Fundamental to operating system design is **concurrency**.
- Concurrency encompasses a host of design issues, including:
  - communication among processes
  - sharing of and competing for resources (such as memory, files, and I/O access)
  - synchronization of the activities of multiple processes
  - allocation of processor time to processes

## Difficulties of Concurrency

- examples of concurrent processing:
  - Interleaving
  - Overlapping
- Basic characteristic of multiprogramming systems: relative speed of execution of processes cannot be predicted.

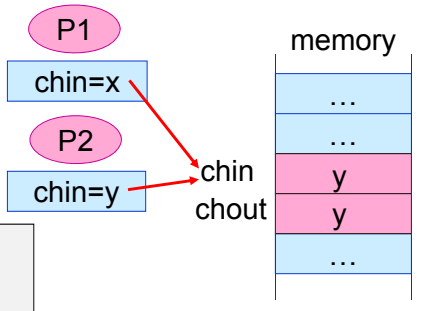
- Sharing of global resources: For example: Global variable
- Operating system managing the allocation of resources optimally: For example: I/O channel
- Difficult to locate programming errors: Program results are not deterministic and reproducible(再现).



## 4.1 Principles of Concurrency

### 2 Race condition(竞争条件)

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




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

- **Murphy Law**: If something can go wrong, it will.

## Race condition.

**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.

- **Critical section(临界区)**: 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.

**Process P1**

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



## Process Interaction

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

## Competition Among Processes For Resources

- **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.
- **Synchronization(同步)**: Refers to the relationship between two or more processes in a system that need to exchange information so that they can work together. A group of concurrent processes in a synchronous relationship is called a cooperative process, and the signals that the cooperative processes send to each other are called messages or events.

> Solution to Race Condition: Control access to the shared resource, that is, **Mutual Exclusion**.



## Competition Among Processes For Resources.

> Enforcement of mutual exclusion incurs two additional problems: Deadlock & Starvation.

- **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 states in response to changes in the other process(es) without doing any useful work.
- **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.

## 4.1 Principles of Concurrency

### 3 Mutual Exclusion

```
/*program mutual exclusion */
const int n=/*number of processes*/;

void P(int i)
{   while(true)
    {   entercritical(i);
        /*critical section */;
        exitcritical(i);
        /*remainder*/
        }
}
void main()
{   parbegin(P(1), P(2)…, P(n));
}
```

```
while(true){
    entry section;
    critical section;
    exit section;
    remainder section;
}
```



## Requirements for Mutual Exclusion

1. Mutual exclusion must be enforced: Only one process at a time is allowed in the critical section for a resource
2. A process that halts must do so without interfering with other processes
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation
4. A process must not be denied access to a critical section when there is no other process using it
5. No assumptions are made about relative process speeds or number of processes
6. A process remains inside its critical section for a finite time only

## 4.2 Hardware Approaches to Mutual Exclusion

### 1 Interrupt Disabling

- In uniprocessor system, A process runs until it invokes an operating system service or until it is interrupted.
- Disabling interrupts guarantees mutual exclusion.
- Disadvantages:
  - Processor is limited in its ability to interleave programs
  - This approach will not work in a multiprocessor architecture

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



## 4.2 Hardware Approaches to Mutual Exclusion

### 2 Special Machine Instruction

- In a multiprocessor configuration, several processors share access to a common main memory.
- Special Machine Instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions
- Test and Set Instruction(TSL)
- Exchange Instruction

## Test and Set Instruction(TSL)

- Variable **Lock**
  - Lock==1, access to the memory location is blocked, Lock==0, allow memory access.

```
enter_region:
   tsl register, lock    ; copy lock to register, then set lock=1
   cmp register, #0   ; compare
   jne enter_region   ; if ≠0, jump to “enter_retion”
   ret
```

```
leave_region:
   move lock, #0 ; open lock
   ret
```

```
call enter_region
critical_region
    ……
call leave_region
noncritical_region
   ……
```

- **Busy-waiting** is employed



## Some Key Terms Related to Concurrency

- **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.

## Test and Set Instruction(TSL).

```
/*program mutual exclusion*/
const int n = /* number of processes */;
int bolt;
void P(int i)
{   while (true)
    {   while( !testset(bolt) );  /* do nothing */
        /* critical section */;
        bolt = 0;
        /* remainder */
    }
}
```

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

```
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));
}
```



## Exchange Instruction

```
/*program mutual exclusion*/
const int 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 exchange(int *register, int *memory)
{   int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

```
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));
}
```

## Advantages And Disadvantages

- 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** is employed, thus while a process is waiting for access to a critical section it continues to consume processor time
  - **Starvation** is possible when a process leaves a critical section and more than one process is waiting
  - **Deadlock** is possible



## Deadlock: Priority Inversion Problem

- **two processes**: A process(High priority) B process(Low priority)
- **Scheduling**: Process A can run as long as it is in a ready state.
- Assume that low-priority process B has entered the critical section first, then process A is ready.
- **Results**: Process B could not leave the critical region; Process A is busy-waiting, and always hungry, that is, deadlock.

## Solution Of Busy Waiting

- The basic idea of the busy waiting method is that when a process wants to enter a critical section, it checks whether it is allowed to enter. If not, the process is busy waiting until it can enter.
- Solution: **Sleep** and **Wakeup**
- Solution idea: use primitives to make the process unable to enter the critical region, go to sleep, also means **blocked**, and wake up the sleep process when the process out of the critical region.



## 4.3 Semaphores

### 1 Semaphores

- Semaphore: A variable that has an integer value upon which only two **primitives**(原语) operations: wait,signal (down,up) (P,V)

Wait: decrements the semaphore value
if semaphore ≥ 0, process continues;
if semaphore < 0, block this process

Signal: increments semaphore value
if semaphore> 0, process continues;
if semaphore≤ 0, a process blocked by a Wait operation is unblocked

## 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;
    }
}
```

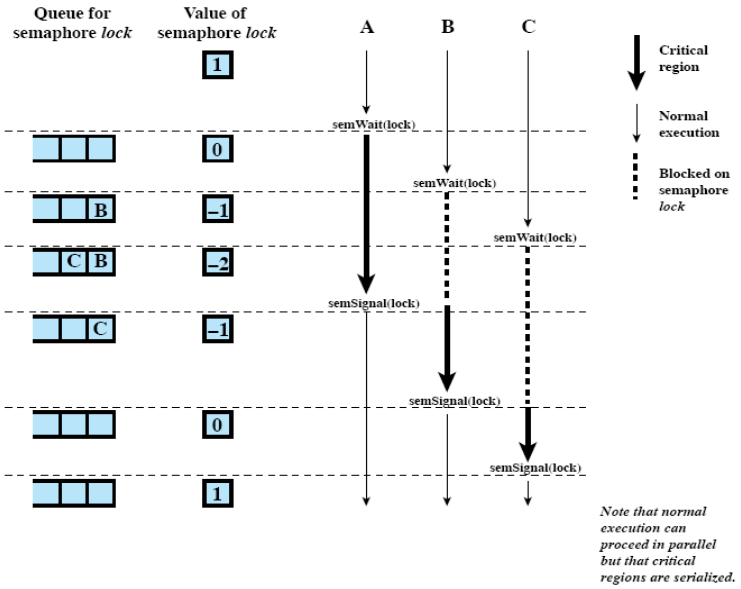
```
void semSignal (semaphore &s)
{   s.count++;
    if (s.count <= 0)
    {   remove a process P from s.queue;
        place process P on ready list;
    }
}
```

## 4.3 Semaphores

### 2 Mutual Exclusion

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

## Mutual Exclusion.






## Semaphore Value Analysis

- N concurrent processes access a same resource
  - 1: No process enters a critical section
  - 0: One process entered the critical section
  - -1~ -(N-1): 1 to N-1 processes are blocked and waiting to enter the critical section

- N concurrent processes access to the critical region, if the same resource can hold a maximum of M processes, what should the semaphore value be?

  - M: No process enters a critical section
  - 0: M processes entered the critical section
  - -1~ -(N-M): 1 to N-M processes are blocked and waiting to enter the critical section

## The Interpretion Of Mutex Value

- Thus, at any time, the value of s.count can be interpreted as follows:
  - **s.count>0**: s.count is the number of processes that can execute semWait(s) without suspension(if no semSignal(s) is executed in the meantime). Since semWait is executed before accessing the shared resource, the initial value of a semaphore can be used to represent the number of available resources of the same kind in the system. So when the initial value of a semaphore can be greater than 0, it is also called **a resource semaphore**.
  - **s.count=0**: No shared resources are available
  - **s.count<0**: The magnitude of s.count is the number of processes suspended in s.queue.



## Mutual Exclusion

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

**Mutual Exclusion**

one semaphore

The initial value is the number of available resources

## Binary Semaphore

- **A binary semaphore** may only take on the values 0 and 1 and can be defined by the following three operations:
  - A binary semaphore may be initialized to 0 or 1.
  - The semWaitB operation checks the semaphore value. If the value is zero, then the process executing the semWaitB is blocked. If the value is one, the value is changed to zero and the process continues execution.
  - The semSignalB operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). 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.
- To contrast the two types of semaphores, the nonbinary semaphore is often referred to as either **a counting semaphore** or **a general semaphore**.



## Binary Semaphore Primitives

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

```
void semWait (binary_semaphore s)
{   if (s.value == one)
        s.value = zero;
    else {   place this process in s.queue;
             block this process    }
}
```

```
void semSignal (semaphore s)
{   if (s.queue is empty())
        s.value  = one;
    else{   remove a process P from s.queue;
            place process P on ready list;    }
}
```

## Strong/Weak Semaphores

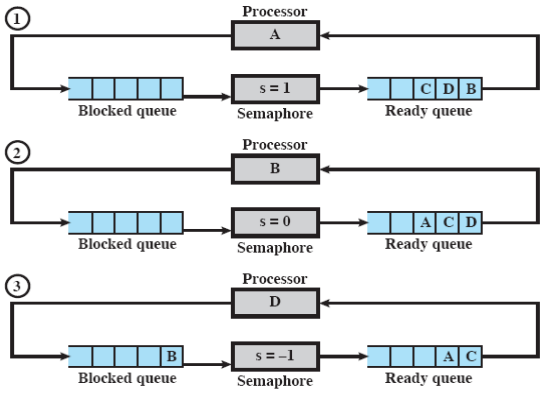
- A queue is used to hold processes waiting on the semaphore.
- A question arises of the order in which processes are removed from such a queue.

- **Strong Semaphores**: the process that has been blocked the longest is released from the queue first (FIFO)
- **Weak Semaphores**: the order in which processes are removed from the queue is not specified.

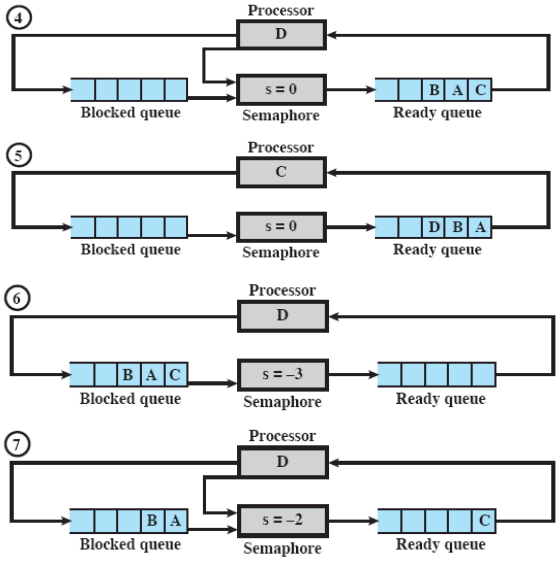


## Example of Strong Semaphore Mechanism

- Processes A, B and C depend on the result of process D
- Initially semaphore count is 1, indicating that only one of D's results is available.



## Example of Strong Semaphore Mechanism.

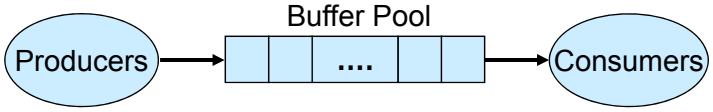





## 4.3 Semaphores

### 3 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




- The Problem: ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer.

## One Producer And One Consumer

- An realistic restriction: the buffer is finite, that is, bounded buffer problem

```
define N=100;
semaphore empty = N;
semaphore full = 0;
void producer(void)
{  while(TRUE){
      int item;
      produce-item(&item);
      semWait(empty);
      enter-item(item);
      semSignal(full);  }
}
```

```
void consumer(void)
{  int item;
   while(TRUE){
      semWait(full);
      remove-item(&item);
      semSignal(empty);
      consume-item(item);
   }
}
```

**Synchronization**

two semaphores

complementary initial values

the total amount is the maximum number of resources



## Multiple Producer And Multiple Consumer

- When consumers want to remove data, there is at least one full cell in a bounded buffer.
- When producers want to place data, there is at least one empty cell in a bounded buffer.
- Only one producer or consumer may access the buffer at any one time

- The solution uses three semaphores:
  - one called **full** for counting the number of slots that are full,
  - one calles **empty** for counting the number of slots that are empty,
  - one calles **mutex** to make sure the producer and consumer do not access the buffer at the same time.

## Multiple Producer And Multiple Consumer.

```
#define  N 100
```

```
semaphore  mutex = 1;
semaphore  empty = N;
semaphore  full = 0;
```

```
void producer(void)
{  int item;
   while(TRUE){
      produce_item(item);
      semWait(empty);
      semWait(mutex);
      enter_item(item);
      semSignal(mutex);
      semSignal(full);  }
}
```

```
void consumer(void)
{  int item;
   while(TRUE){
      semWait(full);
      semWait(mutex);
      remove_item(item);
      semSignal(mutex);
      semSignal(empty);
      consume_item(item);  }
}
```



## Multiple Producer And Multiple Consumer.

```
#define  N 100
```

```
semaphore  mutex = 1;
semaphore  empty = N;
semaphore  full = 0;
```

mutex=1
empty=0
full=N

```
void producer(void)
{  int item;
   while(TRUE){
      produce_item(item);
      semWait(mutex);
      semWait(empty);
      enter_item(item);
      semSignal(mutex);
      semSignal(full);  }
}
```

①
mutex=0
empty=-1
blocked

```
void consumer(void)
{  int item;
   while(TRUE){
      semWait(full);
      semWait(mutex);
      remove_item(item);
      semSignal(mutex);
      semSignal(empty);
      consume_item(item);  }
}
```

②
full=N-1
mutex=-1
blocked

## 4.3 Semaphores

### 4 Implementation of Semaphores

> The essence of the problem is one of mutual exclusion: Only one process at a time may manipulate a semaphore with either a semWait or semSignal operation.

- Imperative that the semWait and semSignal operations be implemented as atomic primitives, one obvious way is to implement them in hardware or firmware.
- Any of the software schemes, such as Dekker's algorithm or Peterson's algorithm( Appendix A ), could be used.
- Another alternative is to use one of the hardware-supported schemes for mutual exclusion.



## 4.4 Monitors(管程)

### Monitors

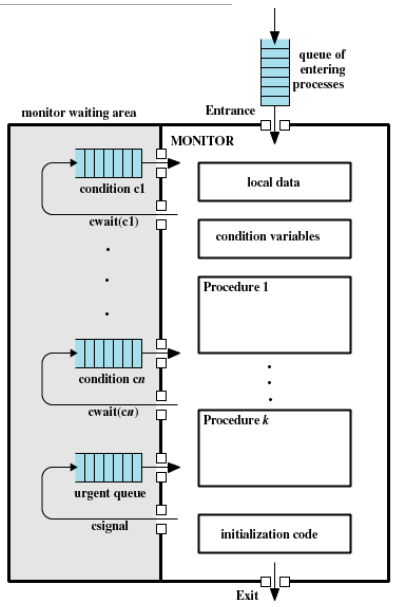
- Monitor is a **software module** consisting of one or more procedures, an initialization sequence, and local data
- Chief characteristics
  - Local data variables are accessible only by the monitor's procedures and not by any external procedure
  - Process enters monitor by invoking one of its procedures
  - **Only one process** may be executing in the monitor at a time

## Condition Variables

- Condition variable are a special data type in monitors, which are operated on by two operations:
  - 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.



## Structure of a Monitor



## Producer/Consumer Problem

```
/* 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 */
}
```

## Producer/Consumer Problem.

```
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**

## 4.5 Message Passing

### 1 Interprocess Communication

- When processes interact with one another two fundamental requirements must be satisfied:
  - **Synchronization**: to enforce mutual exclusion
  - **Communication**: to exchange information
- Interprocess communication mode
  - **Shared memory**: Processes communicate with each other through operations on the same shared main memory
  - **Message mechanism**: Processes exchange information in the unit of messages.
  - **Shared file**: An open shared file is be used to processes communication.



## 4.5 Message Passing

### 2 Message Passing(消息传递)

- Message Passing can works with distributed systems and shared memory multiprocessor and uniprocessor systems
- A pair of primitives is provided for message passing:
  - **send**(destination, message)
  - **receive**(source, message)
- A process sends information in the form of a **message** to another process designated by a **destination**.
- A process receives information by executing the **receive** primitive, indicating the **source** and the message.

## Message Passing.

- Design issues relating to message-passing
  - Synchronization
  - Addressing
  - Message Format
  - Queuing Discipline



## Synchronization

- Communication of a message between two processes implies synchronization between the two.
- Sender and receiver may or may not be blocking (waiting for message)
  - Blocking send, blocking receive: Both sender and receiver are blocked until message is delivered, called a **rendezvous(会合)**.
  - 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

- Schemes for specifying processes in send and receive primitives fall into two categories:
  - Direct addressing(直接寻址)
  - Indirect addressing(间接寻址)



## Direct Addressing

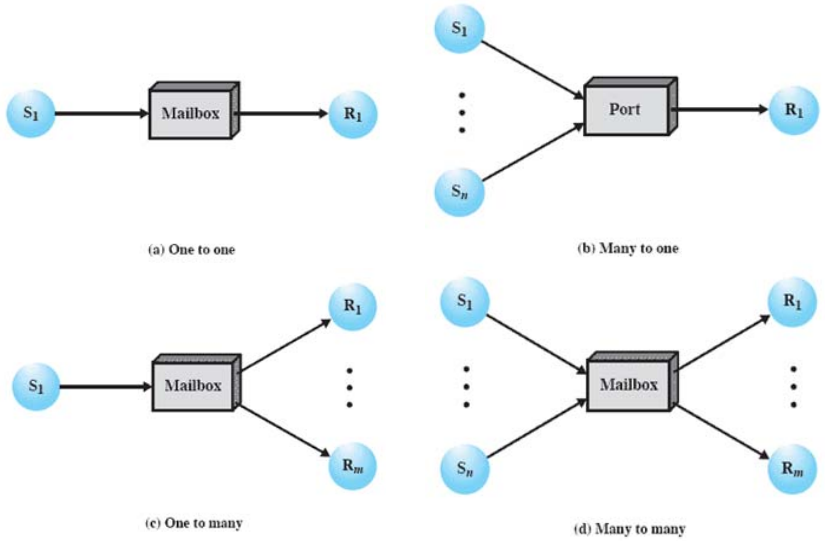
- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process——Receive primitive could know ahead of time which process a message is expected
  - implicit addressing——Receive primitive could use source parameter to return a value when the receive operation has been performed

## Indirect Addressing

- A strength of the use of indirect addressing is that, by decoupling the sender and receiver, it allows for greater flexibility in the use of messages.
  - Messages are sent to a shared data structure consisting of queues
  - Queues are called **mailboxes**
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox



## Indirect Addressing.



## General Message Format

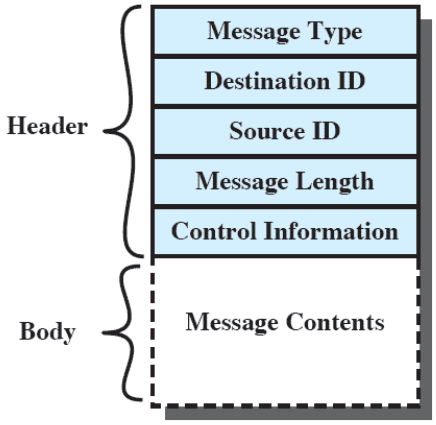



Figure 5.19 General Message Format

- Queuing Discipline
  - FIFO
  - Priority-based strategies



## Mutual Exclusion Using Messages

- Assume the use of the blocking receive primitive and the nonblocking send primitive. A set of concurrent processes share a mailbox, **box**.
  - If there is a message, it is delivered to only one process and the others are blocked
  - If the message queue is empty, all processes are blocked; when a message is available, only one blocked process is activated and given the message.

## Mutual Exclusion Using Messages.

```
/* program mutual exclusion */
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));
}
```



## Solution to Producer/Consumer

- mailbox: mayconsume—full;
- mailbox: mayproduce —empty;

```
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);
}
```

# 4.6 Readers/Writers Problem

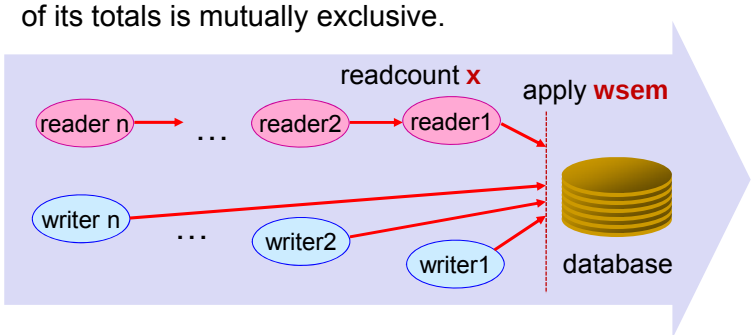
## Readers/Writers Problem

- The readers/writers problem is defined as follows: There is a data area shared among a number of processes. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:
  - 1. Any number of readers may simultaneously read the file.
  - 2. Only one writer at a time may write to the file.
  - 3. If a writer is writing to the file, no reader may read it.



## Readers Have Priority

- A solution using semaphore: Readers have priority
  - **wsem** semaphore: for mutually exclusive access. For writers, wsem must be Wait(down) and Signal(up) when entering a critical section. For readers, the semaphore is operated on only when the first reader arrives.
  - **x** semaphore: is used to ensure that the reader's modification of its totals is mutually exclusive.



## Readers Have Priority

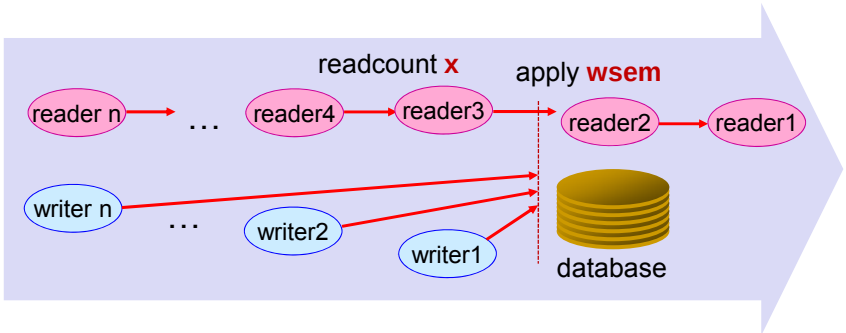
```
/*program readers and writers */
int readcount;
semaphore x=1,wsem=1;
void reader()
{   while(true)
    {   semWait(x);
        readcount++;
        if(readcount==1)
            semWait(wsem);
        semSignal(x);
        READUNIT();
        semWait(x);
        readcount--;
        if(readcount==0)
            semSignal(wsem);
        semSignal(x);    }
```

```
void writer()
{   while(true)
    {   semWait(wsem);
        WRITEUNIT();
        semSignal(wsem);  }
}
void main()
{   readcount=0;
    parbegin(reader, writer);
}
```

- Consider that readers keep coming when there is a writer waiting



## Readers Have Priority



## Writers Have Priority

- A solution using semaphore: Writers have priority
  - **wsem** semaphore: inhibits all writers while there is at least one reader desiring access to the data area.
  - **rsem** semaphore: inhibits all readers while there is at least one writer desiring access to the data area.
  - **rqueue** semaphore: limits the number of readers blocked in the queue of rsem to only one, that is, any additional readers queueing on rqueue. So, writers blocked can jump the queue.
  - **x** semaphore: is used to ensure that the reader's modification of its totals is mutually exclusive.
  - **y** semaphore: is used to ensure that the writer's modification of its totals is mutually exclusive.



```
/*program readers and writers*/
int readcount, writecount;
semaphore x=1, y=1, rqueue=1;
semaphore wsem=1, rsem=1;
void reader()
{   while(true)
    {  semWait(rqueue);
       semWait(rsem);
       semWait(x);
       readcount++;
       if(readcount==1) semWait(wsem);
       semSignal(x);
       semSignal(rsem);
       semSignal(rqueue);
       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);
}
```

## Terminology

- race condition
- critical resource; critical section
- concurrency; concurrency processes
- mutual exclusion; synchronization
- coroutine
- atomic operation; primitives
- deadlock; livelock; starvation
- busy waiting
- semaphore
  - general semaphore
  - binary semaphore; counting semaphore
  - strong semaphore; weak semaphore



## Terminology

- spin waiting
- message passing
- monitor