



操作系统原理及应用

李 伟

xchlw@seu.edu.cn

计算机科学与工程学院、软件学院
江苏省网络与信息安全重点实验室



Chapter 6 Process Synchronization



Outline

- **Background**
- **The Critical-Section Problem**
- **Synchronization Hardware**
- **Semaphores**
- **Classical Problems of Synchronization**
- **Monitors**
- **Synchronization Examples**



Background

- **Concurrent access to shared data may result in data inconsistency.**
- **Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.**



Background

- Shared-memory solution to **bounded-buffer problem** (Chapter 3) allows at most $n - 1$ items in buffer at the same time.
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
 - We can do so by having an integer **count** that keeps track of the number of full buffers



Bounded-Buffer

- **Shared data**

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```



Bounded-Buffer

- **Producer process**

```
item nextProduced;
```

```
while (1) {
```

```
    while (counter == BUFFER_SIZE);
```

```
        /* do nothing */
```

```
    buffer[in] = nextProduced;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
    counter++;
```

```
}
```



Bounded-Buffer

- **Consumer process**

```
item nextConsumed;  
while (1) {  
    while (counter == 0);  
    /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter- -;  
}
```




Bounded-Buffer

- The statements
 counter++;
 counter--;
must be performed ***atomically.***
- Atomic operation means an operation that completes in its entirety without interruption.



Bounded-Buffer

- The statement “**count++**” may be implemented in machine language as:
 - **register₁ = counter**
 - **register₁ = register₁ + 1**
 - **counter = register₁**
- The statement “**count--**” may be implemented as:
 - **register₂ = counter**
 - **register₂ = register₂ - 1**
 - **counter = register₂**



Bounded-Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.



Bounded-Buffer

- Assume counter is initially 5, what is it finally?

Producer

register1 = counter

register1 = register1 + 1

counter = register1

Consumer

register2 = counter

register2 = register2 - 1

counter = register2



Race Condition

- **Race condition** occurs, if:
 - **two or more** processes/threads access and manipulate the **same** data **concurrently**
 - the outcome of the execution **depends on the particular order** in which the access takes place.
- To prevent race conditions, concurrent processes must be **synchronized**.



Outline

- Background
- **The Critical-Section Problem**
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples



The Critical-Section Problem

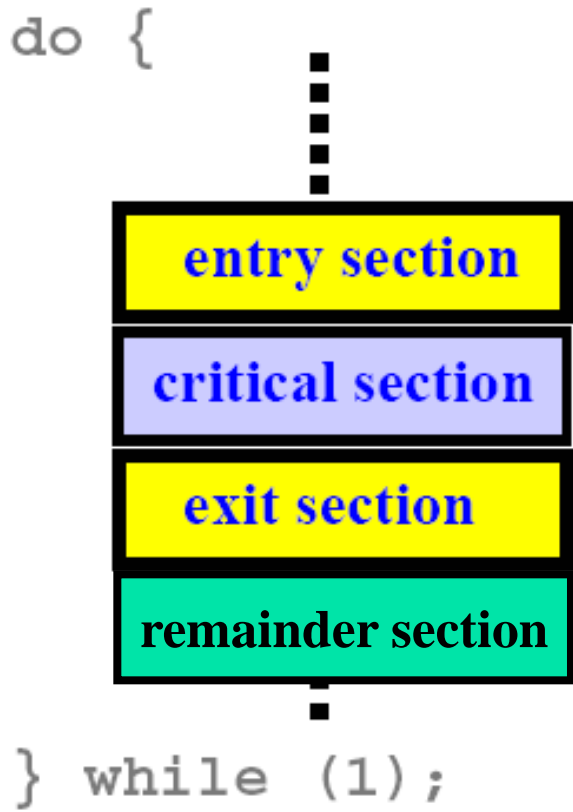
- n processes all competing to use some shared data
- Each process has **a code segment**, called ***critical section***, in which the shared data is changed.
- **Problem** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.



The Critical-Section Problem

- Thus, the execution of critical sections must be *mutually exclusive* (e.g., at most one process can be in its critical section at any time).
- The *critical-section problem* is to design a protocol that processes can use to cooperate.

The Critical Section Protocol



- A critical section protocol consists of **two** parts: an **entry section** and an **exit section**.
- Between them is the critical section that must run in a **mutually exclusive** way.



Solution to Critical-Section Problem

- Any solution to the critical section problem must satisfy the following three conditions:
 - **Mutual Exclusion** (互斥)
 - **Progress** (空闲让进)
 - **Bounded Waiting** (有限等待)
- Moreover, the solution cannot depend on **relative speed** of processes and **scheduling policy**.



Mutual Exclusion

- If a process **P** is executing in its critical section, then **no** other processes can be executing in their critical sections.
- The **critical section protocol** should be capable of blocking processes that wish to enter but cannot.
- Moreover, when the process that is executing in its critical section exits, the **critical section protocol** must be able to know this fact and allows a waiting process to enter.



Progress

- If **no** process is executing in its critical section and some processes wish to enter their critical sections, then
 - Only those processes that are waiting to enter can participate in the competition (to enter their critical sections).
 - No other process can influence this decision.
 - This decision cannot be postponed indefinitely.



Bounded Waiting

- **After** a process made a request to enter its critical section and **before** it is granted the permission to enter, there exists a **bound** on the **number of times** that other processes are allowed to enter.
- Hence, even though a process may be blocked by other waiting processes, **it will not be waiting forever.**
- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the n processes



Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)
do {
 entry section
 critical section
 exit section
 remainder section
} while (1);
- Processes may share some common variables to synchronize their actions.



Algorithm 1

- Shared variables:
 - **int turn;**
initially **turn = i (or turn=j)**
- Process P_i :
 - do { while (turn != i) ;**
 - critical section**
 - turn = j;**
 - remainder section**
 - } while (1);**
- are forced to run in an alternating way.
- Satisfies **mutual exclusion**, but not **progress**



Algorithm 2

■ Shared variables

- **boolean** `flag[2]`;
initially `flag[0] = flag[1] = false`.
- `flag[i] = true` $\Rightarrow P_i$ ready to enter its critical section

■ Process P_i

```
do {  
    flag[i] = true;  
    while (flag[j]) ;  
    critical section  
    flag[i] = false;  
    remainder section  
} while (1);
```

- Satisfies **mutual exclusion**, but not **progress**.



Is the following algorithm correct?

■ Process P_i :

do {

while (flag[j]) ;

flag[i] = true;

critical section

flag [i] = false;

remainder section

} while (1);



Algorithm 3—Peterson's Solution

- Combined shared variables of algorithms 1, 2.
- Process P_i
 - do {
 - $\text{flag}[i] = \text{true};$
 - $\text{turn} = i;$
 - $\text{while} (\text{flag}[j] \text{ and } \text{turn} == j) ;$
 - critical section
 - $\text{flag}[i] = \text{false};$
 - remainder section
 - } while (1);
- Meets all three requirements; solves the critical-section problem **for two processes**.



Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...



Bakery Algorithm

- **Notation**

- $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
- $\max(a_0, \dots, a_{n-1})$ is a number k , such that $k \geq a_i$ for i from 0 to $n - 1$

- **Shared data**

boolean choosing[n];

int number[n];

Data structures are initialized to false and 0 respectively



Bakery Algorithm

```
do {  
    choosing[ i ] = true;  
    number[ i ] = max(number[0], number[1], ..., number [n – 1])+1;  
    choosing[ i ] = false;  
    for (j = 0; j < n; j++) {  
        while (choosing[ j ]) ;  
        while ((number[ j ] != 0) && ((number[ j ],j) < (number[ i ],i))) ;  
    }  
    critical section  
    number[ i ] = 0;  
    remainder section  
} while (1);
```

Discussion



Outline

- Background
- The Critical-Section Problem
- **Synchronization Hardware**
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples



Hardware Support

- **There are two types of hardware synchronization supports**
 - **Disabling/Enabling interrupts**
 - This is slow and difficult to implement on multiprocessor systems.
 - **Special machine instructions**
 - TestAndSet (TS)
 - Swap

Interrupt Disabling

```
do {
```



entry

disable interrupts

critical section

enable interrupts



exit

```
} while (1);
```

- Because interrupts are disabled, no context switch will occur in a critical section.
- Infeasible in a multiprocessor system because all CPUs must be informed.
- Some features that depend on interrupts (e.g., clock) may not work properly.



TestAndSet

- **Test** and **modify** the content of a word **atomically**

```
boolean TestAndSet(boolean &target)
{
    boolean rv = &target;
    &target = true;
    return rv;
}
```



Mutual Exclusion with TestAndSet

- Shared data:
 boolean lock = false;
- Process P_i
 do {
 while (TestAndSet(lock)) ;
 critical section
 lock = false;
 remainder section
 }



Swap

- **Atomically** swap two variables.

```
void Swap(boolean &a, boolean &b)
```

```
{
```

```
    boolean temp = &a;
```

```
    &a = &b;
```

```
    &b = temp;
```

```
}
```



Mutual Exclusion with Swap

- Shared data (initialized to false): **(Global)**
boolean lock;
- **Local** variable for each process
boolean key;
- Process P_i
do { **key = true;**
 while (key == true)
 Swap(lock, key);
 critical section
 lock = false;
 remainder section
 }

Satisfying Three Conditions with TestAndSet

Initially Boolean `waiting[i] = false`; `lock=false`

Enter Critical Section

```
waiting[ i ] = true;  
key = true;  
while (waiting[ i ] && key)  
    key = TestAndSet(lock);
```

```
waiting[i] = false;
```

如果删除该语句，
会产生什么后果？

Leave Critical Section

```
j = (i+1)%n  
while ((j!=i) && !waiting[ j ])  
    j = (j+1)%n;  
if (j == i)  
    lock = false;  
else  
    waiting[ j ] = false;
```



Outline

- Background
- The Critical-Section Problem
- Synchronization Hardware
- **Semaphores**
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples



Semaphores

- Synchronization tool
- Semaphore S – integer variable
- can only be accessed via two standard **atomic** operations

wait (S): $P()$

while $S \leq 0$;
 $S--$;

signal (S): $V()$

$S++$;



Two Types of Semaphores

- ***Counting semaphore*** – integer value can range over an unrestricted domain.
- ***Binary semaphore*** – integer value can range only between 0 and 1; also called **mutex locks**.
- Can implement a counting semaphore S as a binary semaphore.



Critical Section of n Processes

- Shared data:

semaphore mutex; //initially *mutex* = 1

- Process P_i :

do {

 wait(mutex);

 critical section

 signal(mutex);

 remainder section

} while (1);



Wrong or Right?



Semaphore as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
\vdots	\vdots
A	<i>wait (flag)</i>
<i>signal (flag)</i>	B



Semaphore Implementation

- The main disadvantage of the above classical semaphore definition is that it requires **busy waiting**. (while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code)
- This type of semaphore is also called a **spinlock** (自旋锁) because the process spins while waiting for the lock.
- To overcome the need for busy waiting, the process can **block itself** rather than engaging in busy waiting.



Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
```

```
    int value;
```

```
    struct process *L; // waiting queue
```

```
} semaphore;
```

- Assume two simple operations:
 - **block()** suspends the process that invokes it.
 - **wakeup(*P*)** resumes the execution of a blocked process *P*.



Implementation

- Semaphore operations now defined as

wait(S):

S.value--;

if (S.value < 0) {

add this process to S.L;

block();

}

signal(S):

S.value++;

if (S.value <= 0) {

remove a process P from S.L;

wakeup(P);

}



Implementation

- If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- The critical aspect of semaphores is that they be executed **atomically**. (**Critical-section Problem**)
- Busy waiting has **not** been **completely eliminated**.
 - Busy waiting has been removed from the entry section to the critical sections of application programs.
 - Furthermore, we have limited busy waiting to the critical sections of the **wait()** and **signal()** operations.



Exercises

- 5个进程共享某一临界资源，则互斥信号量的取值范围是多少？
- 有4个进程共享一程序段，而每次最多允许2个进程进入该程序段，则信号量的初值是多少？



Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

P_0
wait(S);
wait(Q);
⋮
signal(S);
signal(Q)

P_1
wait(Q);
wait(S);
⋮
signal(Q);
signal(S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.



Outline

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- **Classical Problems of Synchronization**
- Monitors
- Synchronization Examples



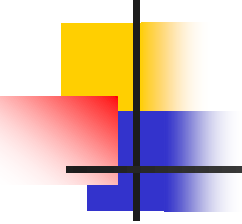
Classical Problems of Synchronization

- **Bounded-Buffer Problem**
- **Readers and Writers Problem**
- **Dining-Philosophers Problem**



Bounded-Buffer Problem

- **Shared data**
 - **Semaphore: full, empty, mutex;**
 - **Initially: full = 0, empty = n, mutex = 1**



Bounded-Buffer Problem

Producer Process

```
do {  ...  
    produce an item in nextp  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    add nextp to buffer  
    ...  
    signal(mutex);  
    signal(full);  
} while (1);
```



Bounded-Buffer Problem Consumer Process

```
do {  
    wait(full)  
    wait(mutex);  
    ...  
    remove an item from buffer to nextc  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    consume the item in nextc  
    ...  
} while (1);
```



作业1

- 在生产者——消费者问题中，信号量 `mutex`, `empty`, `full` 的作用是什么？如果分别对调生产者进程中的两个 `wait` 操作和两个 `signal` 操作，则可能发生什么情况？



Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do **not** perform any updates
 - Writers – can both read and write.
- **Problem** – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
 - First readers-writers problem——**Reader first**
 - Second readers-writers problem——**Writer first**



First Readers-Writers Problem

- Shared data

semaphore mutex, wrt;

int readcount;

- Initially

mutex = 1, wrt = 1, readcount = 0



First Readers-Writers Problem

Writer Process

```
do {  
    wait(wrt);  
    ...  
    writing is performed  
    ...  
    signal(wrt);  
} while (1)
```



First Readers-Writers Problem

Reader Process

```
do {  
    wait(mutex);  
    readcount++;  
    if (readcount == 1)  
        wait(wrt);  
    signal(mutex);  
  
    ...  
    reading is performed  
  
    ...  
    wait(mutex);  
    readcount--;  
    if (readcount == 0)  
        signal(wrt);  
    signal(mutex);  
}
```



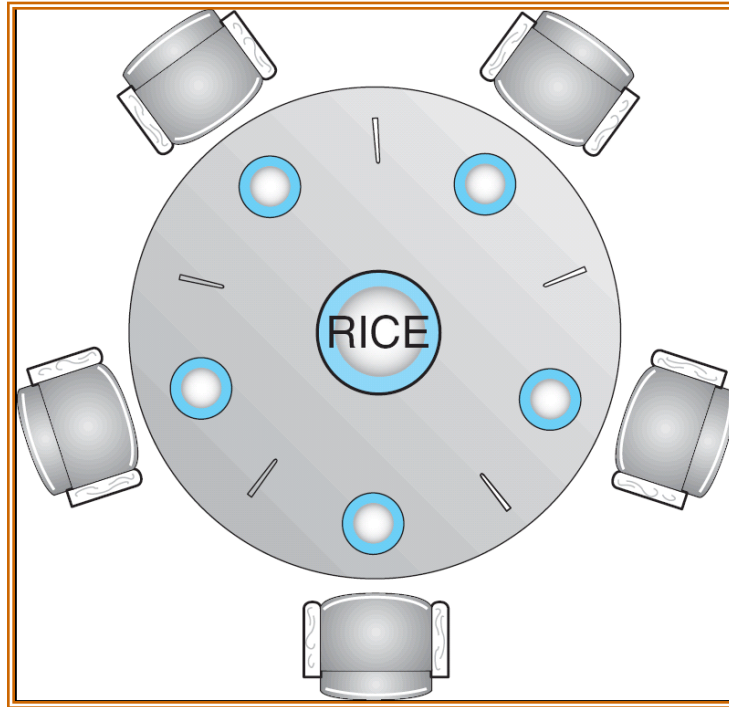
Wrong or Right?



作业2

- 用信号量解决无饥饿的读者——写者问题。

Dining-Philosophers Problem



- Shared data

semaphore chopstick[5];

Initially all values are 1



Dining-Philosophers Problem

- Philosopher i :

```
do {  
    wait(chopstick[i])  
    wait(chopstick[(i+1) % 5])  
  
    ...  
    eat  
  
    ...  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
  
    ...  
    think  
  
    ...  
} while (1);
```



Wrong or Right?



Exercise

- 用信号量解决“独木桥”问题：同一方向的行人可连续过桥，当某一方向有人过桥时，另一方向的人必须等待；当某一方向无人过桥时，另一方向的行人可以过桥。



Exercise

■ Shared data

int countA = 0 //表示A方向上已在独木桥上的行人数目

int countB = 0 //表示B方向上已在独木桥上的行人数目

semaphore MA =1 //实现对countA的互斥修改

MB =1 //实现对countB的互斥修改

mutex =1 //实现两个方向的行人对独木桥
的互斥使用



Exercise

A方向过桥进程:

```
do{  
    wait(MA);  
    countA ++;  
    if (countA == 1) then wait(mutex);  
    signal(MA);  
    过桥;  
    wait(MA);  
    countA --;  
    if (countA == 0) then signal(mutex);  
    signal(MA)  
} while (1);
```

B方向过桥进程?



Outline

- **Background**
- **The Critical-Section Problem**
- **Synchronization Hardware**
- **Semaphores**
- **Classical Problems of Synchronization**
- **Monitors**
- **Synchronization Examples**



Problems with Semaphores

- **Incorrect use of semaphore operations**
 - **signal (mutex) wait (mutex)**
 - **wait (mutex) ... wait (mutex)**
 - **Omitting of wait (mutex) or signal (mutex) (or both)**

把分散在各进程中的临界区集中起来管理



Monitors

- **High-level synchronization construct** that allows the safe sharing of an abstract data type among concurrent processes.

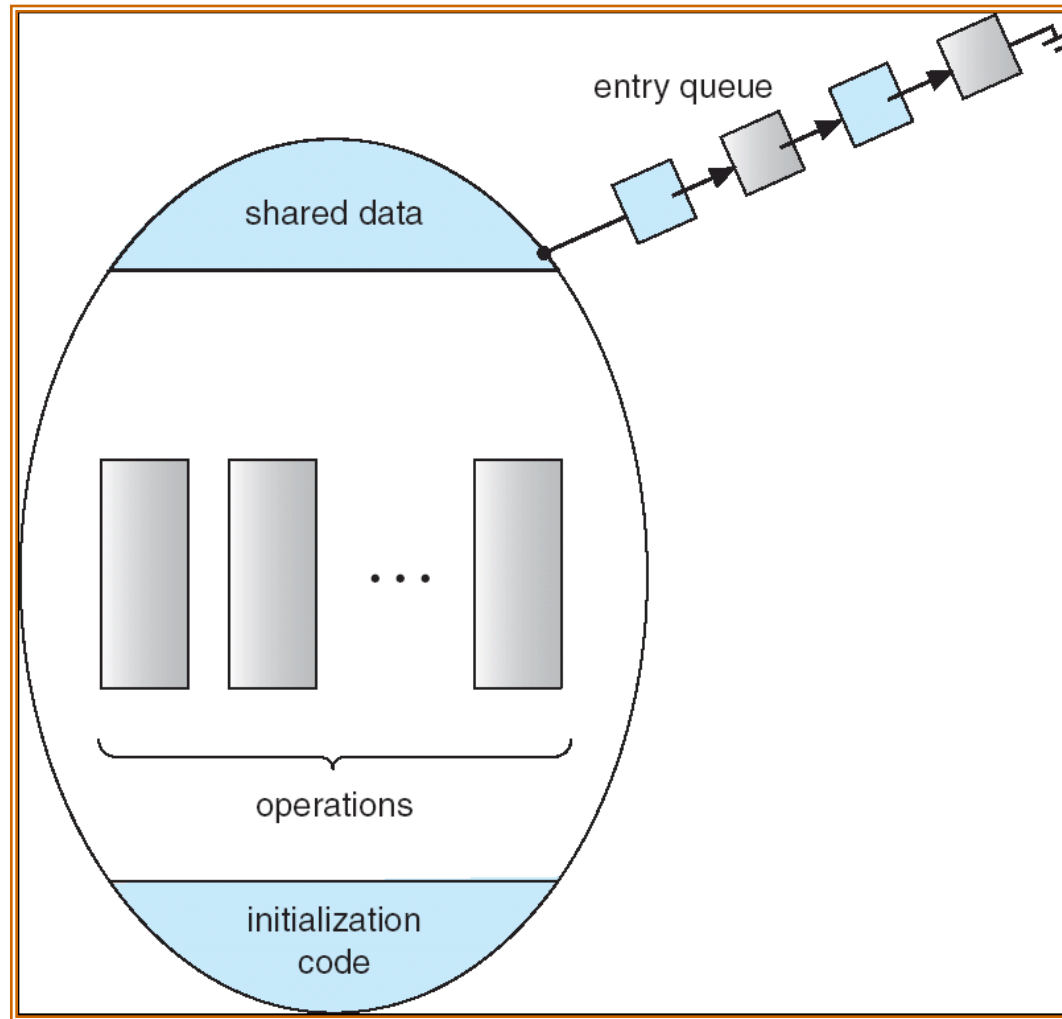
```
monitor monitor-name
{
  shared variable declarations
  procedure body P1 (...) {
    ...}
  procedure body P2 (...) {
    ...}
  procedure body Pn (...) {
    ...}
  { initialization code}
}
```



Monitors: Mutual Exclusion

- ***No more than one process*** can be executing ***within*** a monitor. Thus, ***mutual exclusion*** is guaranteed within a monitor.
- When a process calls a monitor procedure and enters the monitor successfully, it is the ***only*** process executing in the monitor.
- When a process calls a monitor procedure and the monitor has a process running, the caller will be ***blocked outside of the monitor***.

Schematic View of a Monitor

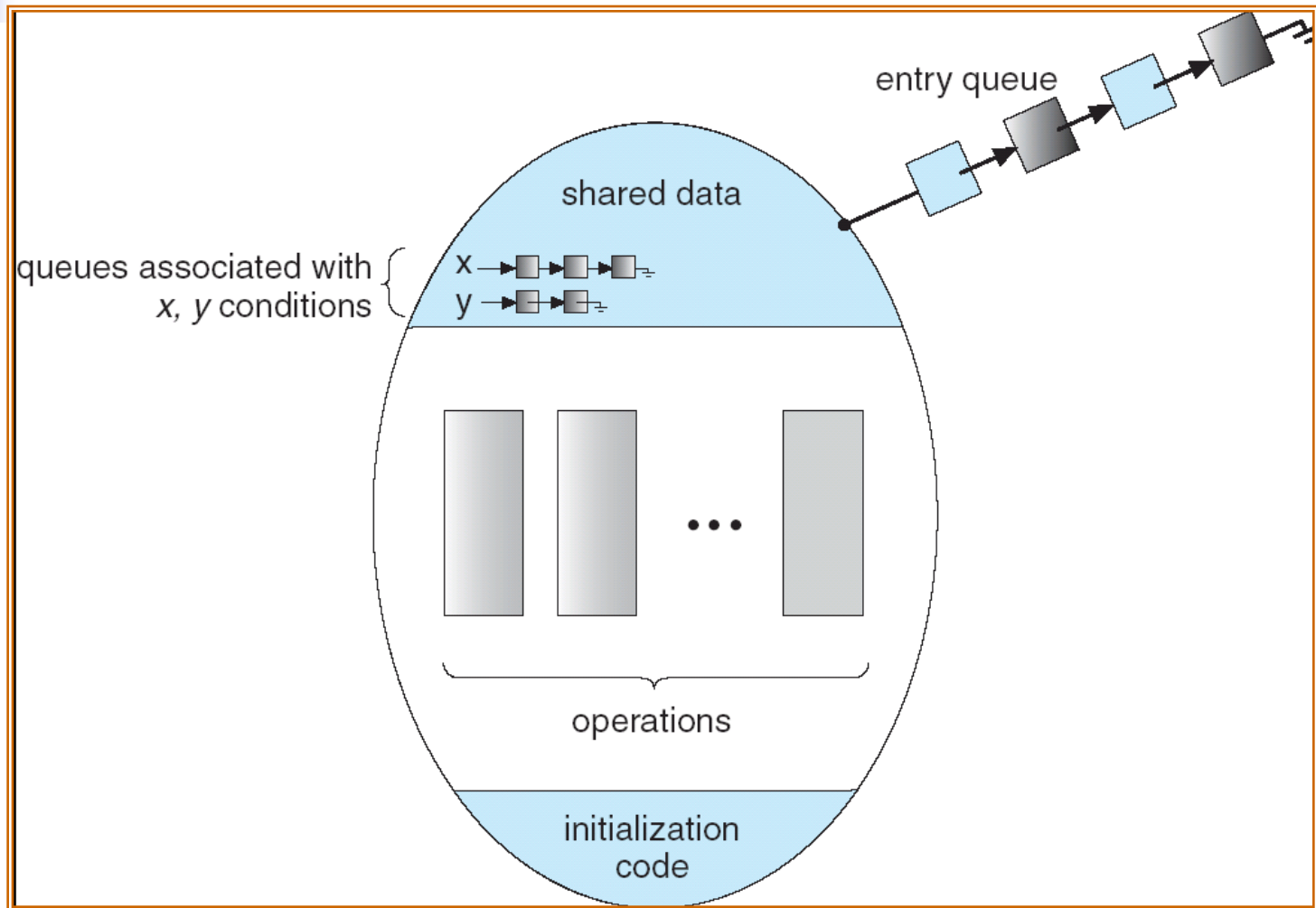




Condition Variables

- To allow a process to wait within the monitor, a **condition** variable must be declared, as **condition x, y;**
- Condition variable can only be used with the operations **wait** and **signal**.
 - **x.wait()** means that the process invoking this operation is suspended until another process invokes **x.signal()**;
 - The **x.signal()** operation resumes exactly one suspended process. If no process is suspended, then the **signal()** operation has no effect.

Monitor With Condition Variables





Condition Signal

- Consider the released process and the signaling process
 - There are **two** processes executing in the monitor, and mutual exclusion is violated!
- Approaches to address this problem
 - The released process takes the monitor and the signaling process waits somewhere. (唤醒并等待)
 - The released process waits somewhere and the signaling process continues to use the monitor. (唤醒并继续)



Semaphore vs. Condition

Semaphores	Condition Variables
Can be used anywhere, but not in a monitor	Can only be used in monitors
<code>wait()</code> does not always block its caller	<code>wait()</code> always blocks its caller
<code>signal()</code> either releases a process, or increases the semaphore counter	<code>signal()</code> either releases a process, or the signal is lost as if it never occurs
If <code>signal()</code> releases a process, the caller and the released <i>both continue</i>	If <code>signal()</code> releases a process, either the caller or the released continues, but <i>not both</i>



Monitor vs. Process

- 管程定义的是公用数据结构，而进程定义的是私有数据结构
- 管程把共享变量上的同步操作集中起来，而临界区却分散在每个进程中
- 管程是为管理共享资源而建立的，进程主要是为占有系统资源和实现系统并发性而引入的
- 管程是被欲使用共享资源的进程所调用，管程和调用它的进程不能并发工作，而进程之间能并发工作
- 管程是语言或操作系统的成分，不必创建或撤销，而进程有生命周期，由创建而产生至撤销便消亡



Dining Philosophers

monitor dp

{

enum {thinking, hungry, eating} state[5];

condition self[5];

void pickup(int i)

// following slides

void putdown(int i)

// following slides

void test(int i)

// following slides

void init()

{ for (int i = 0; i < 5; i++)

state[i] = thinking; }

}



Dining Philosophers

```
void pickup(int i)  
{  
    state[i] = hungry;  
    test[i];  
    if (state[i] != eating)  
        self[i].wait();  
}
```

```
void putdown(int i)  
{  
    state[i] = thinking;  
    // test left and right neighbors  
    test((i+4) % 5);  
    test((i+1) % 5);  
}
```



Dining Philosophers

```
void test(int i)
{
    if ( (state[(i + 4) % 5] != eating) &&
        (state[i] == hungry) &&
        (state[(i + 1) % 5] != eating))
    {
        state[i] = eating;
        self[i].signal();
    }
}
```



Dining Philosophers

- Each philosopher / invokes the operations **pickup()** and **putdown()** in the following sequence:

dp.pickup (i)

EAT

dp.putdown (i)

Monitor Implementation Using Semaphores

- Variables

确保管程的互斥调用

```
semaphore mutex; // (initially = 1)
semaphore next;  // (initially = 0)
int next-count = 0;
```

- Each external procedure F will be replaced by

管程内挂起进程的总数

```
wait (mutex);
```

```
...
```

```
body of  $F$ ;
```

```
...
```

```
if (next-count > 0)
```

```
    signal (next)
```

```
else signal (mutex);
```

防止执行
Signal操作后
两个进程同时
在管程中

- Mutual exclusion within a monitor is ensured.



Monitor Implementation

- For each condition variable x , we have

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

挂起等待资源
的进程

- The operation $x.\text{wait}$ can be implemented as

```
x-count++;
if (next-count > 0)
    signal (next);
else
    signal (mutex);
wait (x-sem);
x-count--;
```




Monitor Implementation

- The operation `x.signal` can be implemented as

```
if (x-count > 0)
{
    next-count++;
    signal (x-sem);
    wait (next);
    next-count- -;
}
```



Monitor Implementation

- ***Conditional-wait*** construct: `x.wait(c)`
 - `c` – integer expression evaluated when the wait operation is executed.
 - value of `c` (*a priority number*) stored with the name of the process that is suspended.
 - when `x.signal()` is executed, process with smallest associated priority number is resumed next.



Monitor Implementation

- **Check two conditions to establish correctness of system**
 - **User processes must always make their calls on the monitor in a correct sequence.**
 - **Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.**



Outline

- **Background**
- **The Critical-Section Problem**
- **Synchronization Hardware**
- **Semaphores**
- **Classical Problems of Synchronization**
- **Monitors**
- **Synchronization Examples**



Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses ***adaptive mutex*** for efficiency when protecting data from short code segments.
- Uses ***condition variables*** , ***semaphore***, and ***readers-writers locks*** when longer sections of code need access to data.
- Uses ***turnstile*** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.



Windows XP Synchronization

- Uses **interrupt masks** (中断屏蔽) to protect access to global resources on uniprocessor systems.
- Uses **spinlocks** on multiprocessor systems.
- Also provides **dispatcher objects** which may act as mutex and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.



Linux Synchronization

- **Linux**
 - **disables interrupts to implement short critical sections**
- **Linux provides**
 - **semaphores**
 - **spinlocks**



Pthreads Synchronization

- **Pthreads API is OS-independent**
- **It provides**
 - **mutex locks**
 - **condition variables**
- **Non-portable extensions include**
 - **read-write locks**
 - **spinlocks**