操作系统原理与设计

第 6 章 Processe Synchronization（进程同步）

陈香兰

中国科学技术大学计算机学院

April 9, 2014

Outline

1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
    - TestAndSet Instruction
    - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. Monitors
8. Synchronization Examples
9. 小结和作业

Outline

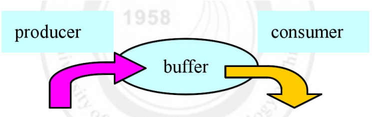
1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
    - TestAndSet Instruction
    - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. Monitors
8. Synchronization Examples
9. 小结和作业

Background

- The processes are **cooperating** with each other **directly or indirectly**.
  - **Independent process cannot** affect or be affected by the execution of another process
  - **Cooperating process can** affect or be affected by the execution of another process
- **Concurrent** access (并发访问) to shared data may result in data inconsistency(不一致)
  - for example: printer, shared variables/tables/lists
- Maintaining data consistency requires **mechanisms** to **ensure the orderly execution** of cooperating processes

Background: Producer-Consumer Problem

- **Producer-Consumer Problem (生产者-消费者问题，PC 问题)**: Paradigm for cooperating processes
  - **producer (生产者)** process produces information that is consumed by a **consumer (消费者)** process.
- **Shared-Memory solution**
  - a buffer of items shared by producer and consumer



  - **Two types of buffers**
    - **unbounded-buffer** places no practical limit on the size of the buffer
    - **bounded-buffer**√ assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

Shared variables reside in a shared region

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0; // index of the next empty buffer
int out = 0; // index of the next full buffer
```

Insert() Method

```
while (true) {
    /* Produce an item */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing −− no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Remove() Method

```
while (true) {
    while (in == out)
        ; // do nothing −− nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements
    - **all empty?** $\Leftarrow (in == out)$
      VS. **all full?** $\Leftarrow (((in + 1)\%BUFFE\_SIZE) == out)$

Another solution using counting value

- Suppose to provide a solution to the PC problem that fills **all** the buffers (not **BUFFER_SIZE-1**).
  - using an integer **count** that keeps track of the number of full buffers.
    - Initially, count = 0.
    - incremented by the producer after it produces a new buffer, and decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Consumer

```
while (true) {
    while (count == 0)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count−−;
    /* consume the item in nextConsumed
}
```

Background: Race Condition(竞争条件)

count++ could be implemented as

$$\begin{aligned} register1 &= count \\ register1 &= register1 + 1 \\ count &= register1 \end{aligned}$$

count−− could be implemented as

$$\begin{aligned} register2 &= count \\ register2 &= register2 - 1 \\ count &= register2 \end{aligned}$$

Code Example

```
0000000000400544 <main>:
#include <stdio.h>
#include <unistd.h>
int count = 1234;

void main(void){
400544: 55 push %rbp
400545: 48 89 e5 mov %rsp,%rbp
400548: 48 83 ec 10 sub $0x10,%rsp

    count ++;
40054c: 8b 05 d6 0a 20 00 mov 0x200ad6(%rip),%eax # 601028 <count>
400552: 83 c0 01 add $0x1,%eax
400555: 89 05 cd 0a 20 00 mov %eax,0x200acd(%rip) # 601028 <count>

......
```

Background: Race Condition(竞争条件)

count++ could be implemented as

$$\begin{aligned} register1 &= count \\ register1 &= register1 + 1 \\ count &= register1 \end{aligned}$$

count−− could be implemented as

$$\begin{aligned} register2 &= count \\ register2 &= register2 \text{ - } 1 \\ count &= register2 \end{aligned}$$

- Consider this execution interleaving with “count = 5” initially:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}
  - S3: consumer execute register2 = register2 - 1 {register2 = 4}
  - S4: producer execute count = register1 {count = 6 }
  - S5: consumer execute count = register2 {count = 4}

Race Condition≡A situation:

where several processes **access and manipulate the same data concurrently** and the **outcome** of the execution **depends on the particular order** in which the access take place

Outline

Critical-Section (临界区)

- **Critical Resources**(临界资源):
  在一段时间内只允许一个进程访问的资源
- **Critical Section** (CS, 临界区):
  a segment of code, access and may change shared data (critical resources)
  - Make sure, that **any two processes will not execute in its own CSes at the same time**
- the CS problem is to design a protocol that the processes can use to cooperate.

```
do {
    entry section(each process must request permission to enter its CS)
        critical section
    exit section
        remainder section
}while (TRUE)
```

Solution to Critical-Section Problem

- A solution to the Critical-Section problem must **satisfy**:

  1. **Mutual Exclusion** (互斥):
     If process P_i is executing in its CS, no other processes can be executing in their CSes.
  2. **Progress** (空闲让进):
     If no process is executing in its CS and there exist some processes that wish to enter their CSes, the selection of the processes that will enter the CS next cannot be postponed indefinitely
  3. **Bounded Waiting** (有限等待):
     A bound must exist on the number of times that other processes are allowed to enter their CSes after a process has made a request to enter its CS and before that request is granted

     - Assume that each process executes at a **nonzero** speed
     - No assumption concerning relative speed of the N processes

Outline

1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
   - TestAndSet Instruction
   - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. Monitors
8. Synchronization Examples
9. 小结和作业

Overview

- Peterson's Solution:
  A classic **software**-based solution, only **two** processes are concerned
- Assume that the **LOAD and STORE** instructions are **atomic**; that is, cannot be interrupted.

- Algorithms 1~3 are not satisfied
- Perterson's Solution is correct

Algorithm 1

- Let the two threads share a common integer value **turn**

  volatile int turn=0; // initially turn = 0

  - turn =i$\Rightarrow$ T_i can enter its CS

T_i

```
Do {
    while (turn!=i)
        ; // do nothing
            CRITICAL SECTION
    turn = j;
            REMAINDER SECTION
} while(1);
```

Analysis:

- ? **Mutual execution**: $\surd$
- ? **Progress**: ×

Algorithm 2

- Replace the shared variable turn with a shared array:

  volatile boolean flag[2];

  - Initially flag[0] = flag[1] = false;
  - flag[i] = true ⇒ T_i want to enter its CS, and enter its CS

T_i

```
do {
    While (flag[j]); // do nothing
    flag[i] = true;
        CRITICAL SECTION
    Flag[i]=flase;
        REMAINDER SECTION
} while(1);
```

Analysis:

- ? **Progress**: $\surd$
- ? **Mutual execution**: $\times$
  When flag[0] and flag[1] changes from false to true almost at the same time, they enter the CS at the same time

Algorithm 3

- flag[i] = true ⇒ T_i is hoping to enter its CS

T_i

```
do {
    flag[i] = true;
    While (flag[j]) ; // do nothing
        CRITICAL SECTION
    Flag[i]=flase;
        REMAINDER SECTION
} while(1);
```

Analysis:

- **Progress** (×) and **Bounded waiting** (×)
  When flag[0] and flag[1] changes from false to true almost at the same time, both processes cannot enter the CS (forever)

Peterson's Solution

- Combining the key ideas of algorithm 1 & 2.

The two processes share two variables:

```
int turn;
Boolean flag[2]
```

Algorithm for Process P_i

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j)
            ; // do nothing
            CRITICAL SECTION
        flag[i] = FALSE;
            REMAINDER SECTION
}
```

This solution is correct.

Outline

1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
    - TestAndSet Instruction
    - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. Monitors
8. Synchronization Examples
9. 小结和作业

Synchronization Hardware

- **Generally**, any solution to the CS problem requires a **LOCK**
  - a process
    - acquires a lock before entering a CS
    - releases the lock when it exits the CS

```
do {
    acquire lock
        critical section
    release lock
        remainder section
}while (TRUE);
```

  - CSes are protected by locks
  - Race conditions are prevented

Synchronization Hardware

- Many systems provide **hardware** support for CS code
  - Uniprocessors – could **disable interrupts**
    - Current code would execute without preemption

```
do {
    disable interrupt
        critical section
    enable interrupt
        remainder section
}while (TRUE);
```

    - Generally **too inefficient** on multiprocessor systems, OSes using this not broadly scalable
  - Modern machines therefore provide **special atomic hardware instructions**

Atomic = non-interruptable

- TestAndSet()
- Swap()

TestAndSet Instruction

Definition:

```
boolean TestAndSet (boolean
*target) {
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

Truth table (真值表)

target		return value
before	after	
F	T	F
T	T	T

- **Mutual-execlution** solution using TestAndSet
  - Shared boolean variable **lock**, initialized to **false**.

Solution:

```
while (true) {
        while ( TestAndSet (&lock ))
                ; // do nothing
                // critical section
        lock = FALSE;
                // remainder section
}
```

TestAndSet Instruction

Definition:

```
boolean TestAndSet (boolean
*target) {
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

Truth table (真值表)

target		return value
before	after	
F	T	F
T	T	T

- **Mutual-execlution** solution using TestAndSet
  - Shared boolean variable **lock**, initialized to **false**.

Solution:

```
while (true) {
        while ( TestAndSet (&lock ))
                ; // do nothing
                // critical section
        lock = FALSE;
                // remainder section
}
```

Swap Instruction

Definition:

```
void Swap (boolean *a, boolean *b) {
        boolean temp = *a;
        *a = *b;
        *b = temp;
}
```

Truth Table

(a,b)	
before	after
(T,T)	(T,T)
(T,F)	(F,T)
(F,T)	(T,F)
(F,F)	(F,F)

- **bounded-waiting**?×

Swap Instruction

- **Mutual-exclusion** solution using Swap
  - Shared Boolean variable **lock** initialized to **FALSE**;
  - Each process has a **local** Boolean variable **key**.

Solution:

```
while (true) {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
               // critical section
        lock = FALSE;
               // remainder section
}
```

- Truth Table

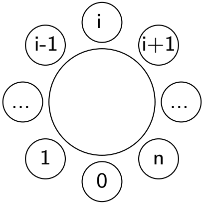
(lock,key)	
before	after
(T,T)	**(T,T)**
(T,F)	(F,T)
(F,T)	**(T,F)**
(F,F)	(F,F)

- **bounded-waiting**?×

Bounded-waiting mutual exclusion with TestAndSet()

Shared data

boolean waiting[n]; // initialized to false
boolean lock; // initialized to false



```
do {
    waiting[i]=TRUE;
    key=TRUE;
    while (waiting[i] && key)
        key=TestAndSet(&lock);
    waiting[i] = FALSE;
        // critical section
    j=(i+1)%n;// consider other processes
    while((j!=i)&&!waiting[j])
        j=(j+1)%n;
    if (j==i) // nobody waiting!
        lock=FALSE;//release lock
    else
        waiting[j]=FALSE;// let it run!
        // remainder section
}while(TRUE);
```

Outline

1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
    - TestAndSet Instruction
    - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. Monitors
8. Synchronization Examples
9. 小结和作业

Semaphore

- The various **hardware-based solutions** to the critical-section problem are **complicated** for application programmers to use
- Semaphore **S** – integer variable (整型信号量)
  - **Initialization** + Two standard operations modify **S**:
    - **wait()** and **signal()**
    - Originally called **P()** and **V()**
  - Can only be accessed via two indivisible (**atomic**) operations

wait() and signal()

```
wait (S) {
    while (S <= 0) ; // no-op
    S --;
}
```

```
signal (S) {
    S++;
}
```

using semaphore

- Using as
  1. **counting** semaphore
     - control access to a given resource consisting of a finite number of instances
  2. **binary** semaphore
     - provide mutual execlusion, can deal with the critical-section problem for multiple processes
  3. **synchronization tools**
     - solve various synchronization problems

using semaphore

1. **Counting semaphore**

   also named as **Resource semaphore**

   - Initialized to N, the number of resources available
   - resource **requesting**: **wait()**
     - if the count of resource goes to 0, waiting until it becomes > 0
   - resource **releasing**: **signal()**
   - usage

```
semaphore resources; /* initially resources = n */
do {
    wait ( resources );
        Critical section;
    signal( resources );
        Remainder section;
} while(1);
```

using semaphore

2. **Binary semaphores**

   also known as **mutex lock**s (互斥锁), provides mutual exclusion

   - integer value: **0 or 1**;
   - can be simpler to implement;

     Can implement a counting semaphore **S** as a binary semaphore
   - usage:

```
Semaphore S; // initialized to 1
do {
    wait (S);
        Critical Section
    signal (S);
        Remainder section
} while (TRUE);
```

using semaphore

3. using semaphore to slove various synchronization problems
   可以描述前趋关系
   - if $p_1 : S_1 \rightarrow p_2 : S_2$, then
     Semaphore **synch**, initialized to **0**, and

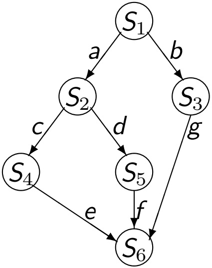
p1	p2
...	...
S1	...
signal(synch)	wait(synch)
...	S2

using semaphore

3. using semaphore to slove various synchronization problems
    - Example

前趋图举例

```
semaphore a,b,c,d,e,f,g = 0,0,0,0,0,0,0
begin
    parbegin
        begin S1;signal(a);signal(b);end;
        begin wait(a);S2;signal(c);signal(d);end;
        begin wait(b);S3;signal(g);end;
        begin wait(c);S4;signal(e);end;
        begin wait(d);S5;signal(f);end;
        begin wait(e);wait(f);wait(g);S6;end;
    parend
end
```



Semaphore Implementation

- **Disadvantage**:
  the previous semaphore may cause **busy waiting(忙等)**
  - this type of semaphore is also called a **spinlock** (自旋锁), suitable situation
    1. busy waiting (for I/O) time $<$ context switching time, or
    2. multiprocessor systems & busy waiting time is very short
- Semaphore implementation with **no busy waiting**
  Record semaphore(记录型信号量)
  - depend on **block()** & **wakeup()** operations

Semaphore Implementation

- **Record semaphore** (记录型信号量)

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- **wait()**

```
wait(Semaphore *S){
    S->value−−;
    if (S->value<0){
        add this process to S->list;
        block();
    }
}
```

- **signal()**

```
signal(semaphore *S){
    S->value++;
    if (S->value <= 0){
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Semaphore Implementation

- 分析 S->value
  - 对于 wait 操作：
    - 当 value≥1 时，说明有资源剩余；申请资源只需要减 1
    - 当 value<1 时，说明没有资源剩余；此时，减去 1，并等待
  - 对于 signal 操作，
    - 若 value ≥0，说明没有等待者，不必唤醒，只需加 1 释放资源
    - 若 value<0，说明有等待者；加 1 缩短等待队列长度，并唤醒 1 个进程（资源分配给这个进程）
  - 查看 value
    - value ≥0，说明没有等待者，此时，**value 值表示剩余资源的个数**
    - value<0，说明有等待者，此时 L 上有等待进程；此时，**value 的绝对值表示等待进程的个数**

the synchronization problem about semaphores

- the synchronization problem about semaphores
  - **No two processes can execute P/V operation on the same semaphore at the same time**
  - HOW to be executed **atomically**?

  - **uniprocessors**: **inhibiting interrupt while wait() and signal()**
  - **multiprocessors**:
    - inhibiting interrupt globally
    - or spin lock

Misuse of semaphore: 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	P_1
wait(S)	wait(Q)
wait(Q)	wait(S)
...	...
signal(S)	signal(Q)
signal(Q)	signal(S)

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

AND 型信号量 I

- Basic idea
  - 将进程在整个运行过程中需要的所有资源，一次性全部的分配该进程，待进程使用完后再一起释放。
  - 即资源分配具有原子性，要么全分配；要么一个都不分配
- Swait() and Ssignal()

Swait(S1,S2,···,Sn)

```
if(S1 ⩾ 1 and S2 ⩾ 1 and ···and Sn ⩾ 1 ) then
    for i:=1 to n do
        Si:=Si − 1;
    endfor
else
    将进程加入第一个条件不满足的 Si 的等待队列上，并修改程序指针到 Swait 操作的开始部分
endif
```

Ssignal(S1,S2,···,Sn)

```
for i:=1 to n do
    Si:=Si + 1;
    若 Si 有等待进程，则唤醒
endfor
```

信号量集

- 信号量集的目标：更一般化
  - 例如，一次申请多个单位的资源；
  - 又如，当资源数低于某一下限值时，就不予分配

Swait(S1, t1, d1,S2, t2, d2,···,Sn,tn,dn)

```
if(S1≥t1 and S2≥t2 and ···and Sn≥tn )then
    for i:=1 to n do
        Si:=Si−di;
    endfor
else
     将进程加入第一个条件不满足的 Si 的等待队列上，
并且修改程序指针到 Swait 操作的开始部分
endif
```

Ssignal(S1, d1,S2, d2,··· ,Sn,dn)

```
for i:=1 to n do
    Si:=Si + di;
     若 Si 有等待进程，则唤醒
endfor
```

信号量集

- 信号量集的几种特殊情况：
  - Swait(S,d,d)：多单位分配
  - Swait(S,1,1)：一般的记录型信号量
  - Swait(S,1,0)：$s \geq 1$ 时，允许多个进入临界区；$s = 0$ 后，阻止一切

Outline

1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
    - TestAndSet Instruction
    - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. Monitors
8. Synchronization Examples
9. 小结和作业

Classical Problems of Synchronization

- Use semaphores to solve
  1. Bounded-Buffer Problem，生产者 -消费者问题（PC Problem）
  2. Readers and Writers Problem，读者 -写者问题
  3. Dining-Philosophers Problem ，哲学家就餐问题

Classical Problems of Synchronization

1. Solution to Bounded-Buffer Problem (PC problem, 生产者 -消费者问题)
   - N buffers, each can hold one item
   - Semaphore **mutex** initialized to the value 1
   - Semaphore **full** initialized to the value 0
   - Semaphore **empty** initialized to the value N.

The structure of the producer process

```
while (true) {
        // produce an item
    wait (empty);
    wait (mutex);
        // add the item to the buffer
    signal (mutex);
    signal (full);
}
```

The structure of the consumer process

```
while (true) {
    wait (full);
    wait (mutex);
        // remove an item from buffer
    signal (mutex);
    signal (empty);
        // consume the removed item
}
```

Classical Problems of Synchronization

2. Sulotion to Readers-Writers Problem(读者 – 写者问题)
   - 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.
   - Shared Data
     - Data set
     - Semaphore **mutex** initialized to 1.
     - Semaphore **wrt** initialized to 1.
     - Integer **readcount** initialized to 0.

Classical Problems of Synchronization

2. Sulotion to Readers-Writers Problem(读者 – 写者问题)

The structure of a writer process

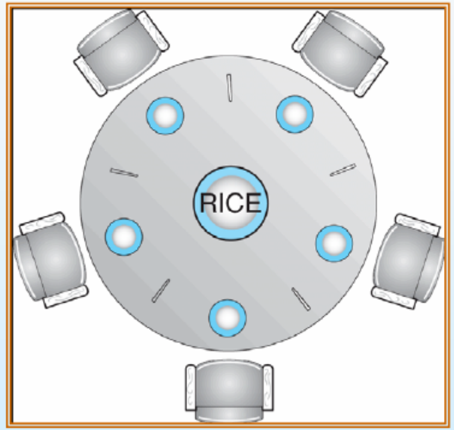
```
while (true) {
    wait(wrt);
        // writing is performed
    signal(wrt);
}
```

The structure of a reader process

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

Classical Problems of Synchronization

3. Dining-Philosophers Problem (哲学家就餐问题)



Classical Problems of Synchronization

3. Dining-Philosophers Problem (哲学家就餐问题)

    - Shared data
        - Bowl of rice (data set)
        - Semaphore **chopstick [5]** initialized to 1
    - This solution may cause a **deadlock**.
        - **WHEN?**

The structure of Philosopher i:

```
While (true) {
    wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );
        // eat
    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
        // think
}
```

Classical Problems of Synchronization

3. Dining-Philosophers Problem (哲学家就餐问题)

- Several **possible remedies**
    - Allow **at most 4 philosophers** to be sitting simultaneously at the table.
    - Allow a philosopher to pick up her chopsticks only if **both chopsticks are available**
    - **Odd** philosophers pick up first her left chopstick and then her right chopstick, while **even** philosophers pick up first her right chopstick and then her left chopstick.
- 注：deadlock-free & starvation-free

Problems with Semaphores

- Incorrect use of semaphore operations:

signal (mutex) ···. wait (mutex)

- the mutual-exclusion requirement is violated, processes may in their CS simultaneously

wait (mutex) ···wait (mutex)

- a deadlock will occur.

Omitting of wait (mutex) or signal (mutex) (or both)

- either mutual-exclusion requirement is violated, or a deadlock will occur

Outline

1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
   - TestAndSet Instruction
   - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. **Monitors**
8. Synchronization Examples
9. 小结和作业

Monitors I

- Monitor type:
  A **high-level abstraction** that provides a convenient and effective mechanism **for process synchronization**
  - encapsulates private data with public methods to operate on that data.
  - **Mutual exclusion: Only one process may be active within the monitor at a time**

Syntax of a monitor

```
monitor monitor-name {
    // shared variable declarations
    procedure P1 (···) {···}
        ···
    procedure Pn (···) {···}
    Initialization code (···.) {···}
}
```

- Within a monitor
  - a procedure can access only local variables and formal parameters
  - the local variables can be accessed by only the local precedures

Monitors II

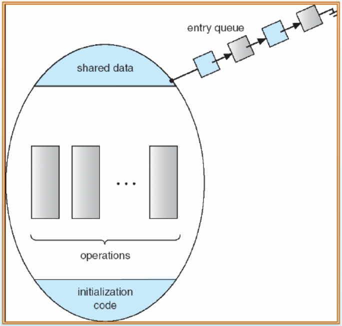


Figure: Schematic view of a Monitor

Condition Variables

- the monitor construct is **not sufficiently powerful** for modeling some synchronization scheme.
- Additional synchronization mechanisms are needed.
- **Condition variables**:

  condition x, y;

  - Two operations on a condition variable:

x.wait()

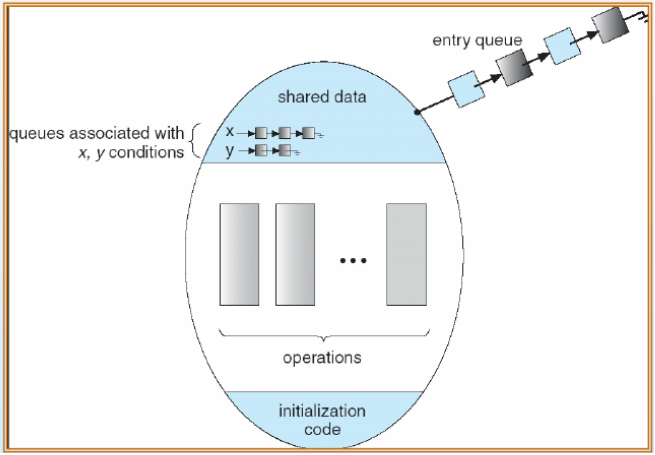
- a process that invokes the operation is suspended.

x.signal()

- resumes one of processes (if any) that invoked x.wait ()

Condition Variables

- Monitor with Condition Variables



Condition Variables

- **Problem** with x.signal()
  - process P invokes x.signal, and a suspended process Q is allowed to resume its execution, then ?
    - **signal and wait**
    - **signal and continue**
  - in the language **Concurrent Pascal**, a compromise was adopted
    - when P executes the signal operation, it immediately leaves the monitor, hence, Q is immediately resumed.

A deadlock-free solution to Dining Philosophers (哲学家就餐问题) I

- the monitor

```
monitor DP {
    enum { THINKING; HUNGRY, EATING} state[5] ;
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
```

A deadlock-free solution to Dining Philosophers (哲学家就餐问题) II

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

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

A deadlock-free solution to Dining Philosophers (哲学家就餐问题) III

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

```
dp.pickup(i)
EAT
dp.putdown(i)
```

- not **starvation-free**

Monitor Implementation Using Semaphores I

- Monitor implementation
  - Variables

    semaphore mutex; // (initially = 1) , for enter and exit monitor
    semaphore next; // (initially = **0**)
    int next-count = 0;
  - Each **external** procedure F will be replaced by

```
wait(mutex);
    ...
    body of F;
    ...
if (next-count > 0)
    signal(next)
else
    signal(mutex);
```

  - Mutual exclusion within a monitor is ensured.

Monitor Implementation Using Semaphores II

- Condition variable implementation:
  - For each condition variable x, we have:

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

x.wait can be implemented as:

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

x.signal can be implemented as:

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

Outline

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
  1. semaphores
  2. condition variables
  3. **adaptive mutexe**s (for short CS less than a few hundred instructions)
  4. **readers-writers lock**s
  5. **turnstile**s (十字转门) to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
     - a type of blocked threads queue
     - organized according to a priority-inheritance protocol to prevent priority inversion (only for kernel locking)

Windows XP Synchronization

- **Windows XP** is a **multithreaded** kernel, supporting **real-time** applications and **multiple processors**.
- To protect access to global resources in kernel:
  - Uses **interrupt masks** on uniprocessor systems
  - Uses **spinlocks** on multiprocessor systems
  - A thread holding a spinlock will never be preempted.
- For threads outside the kernel, provides **dispatcher objects** which may act as
  1. mutexes
  2. semaphores
  3. events (much like a condition variable)
  4. timers

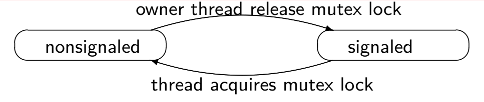


Figure: Mutex dispatcher lock

Linux Synchronization

- The **Linux kernel**
  - before 2.6, **nonpreemptive** kernel
    But now, **fully preemptive** kernel
  - **MEANING**: a process running in kernel mode **could not** be preempted, or **could**.
- For kernel, Linux provides:
  - **semaphores**, **spinlocks**, and **reader-writer versions** of these two locks
- The fundamental locking mechanism for short CS durations in kernel.

single processor	multiple processors
Disable kernel preemption: preempt_disable()	acquire spinlock
Enable kernel preemption: preempt_enable()	Release spinlock

  - **NOTE**: spinlocks are along with enabling and disabling kernel preemption.

Pthreads Synchronization

- **Pthreads API** is **OS-independent**
- For thread synchronization, it provides:
  - **mutex locks**
  - **condition variables**
  - **read-write locks**
- **Non-portable** extensions include:
  - **semaphores** (belong to the POSIX SEM extension)
  - **spin locks**

Outline

小结

1. Background
2. The Critical-Section Problem (临界区问题)
3. Peterson's Solution
4. Synchronization Hardware
    - TestAndSet Instruction
    - Swap Instruction
5. Semaphores
6. Classical Problems of Synchronization
7. Monitors
8. Synchronization Examples
9. 小结和作业

作业

- 参见课程主页

谢谢！