操作系统原理与设计

第6章 Processe Synchronization1（进程同步1）

陈香兰

中国科学技术大学计算机学院

2009年10月28日

提纲

Outline

allowframebreaksBackground

- 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 to shared data may result in data inconsistency
  - for example: printer, shared variables/tables/lists
- Maintaining data consistency requires mechanisms to **ensure the orderly execution** of cooperating processes

Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - unbounded-buffer
    places no practical limit on the size of the buffer
  - bounded-buffer $\surd$
    assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution I

- Solution is correct, but can only use **BUFFER_SIZE-1** elements
- Shared data

```
#define BUFFER_SIZE 10

typedef struct { ... } item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

Bounded-Buffer – Shared-Memory Solution II

- 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;
}
```

Bounded-Buffer – Shared-Memory Solution III

- 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;
}
```

another solution using counting value I

- Suppose that we wanted to provide a solution to the producer-consumer 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 is set to 0.
  - incremented by the producer after it produces a new buffer
  - and is decremented by the consumer after it consumes a buffer.
- Producer

another solution using counting value II

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

another solution using counting value III

- Consumer

```
while (true) {

    while (count == 0)

        ; // do nothing

    nextConsumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

    count--;

    /* consume the item in nextConsumed

}
```

Race Condition I

- count++ could be implemented as

  register1 = count

  register1 = register1 + 1

  count = register1

- count– could be implemented as

  register2 = count

  register2 = register2 - 1

  count = register2

- Consider this execution interleaving with "count = 5" initially:

Race Condition II

- 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）:
  - a segment of code, access and may change shared data (critical resources)
- Make sure, that **any two processes will not execute in its own critical sections at the same time**
- the critical-section problem is to design a protocol that the processes can use to cooperate.
  - **entry section**: each process must request permission to enter its critical-section
  - critical section
  - exit section
  - remainder section

```
do {
    entry section
        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 critical section, then no other processes can be executing in their critical sections
  2. **Progress** (让进)- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section 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 critical sections after a process has made a request to enter its critical section 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

- **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: T_i can enter its CS

- T_i

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

- 分析：Satisfies mutual execution, but **not 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

- Ti

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

- 分析：满足空闲让进，但当flag几乎同时从false转为true时，会出现同时进入临界区的现象，即不满足“互斥”

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);
```

- 分析：当几乎同时将flag设为true后，两者都进不了临界区（永远），同时违背“空闲让进”和“有限等待”

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);
                CRITICAL SECTION
        flag[i] = FALSE;
                REMAINDER SECTION
}
```

- This solution is correct.

Outline

Synchronization Hardware

- Many systems provide **hardware** support for critical section code
- Uniprocessors – could **disable interrupts**
  - Currently running code would execute without preemption
  - Generally **too inefficient** on multiprocessor systems
    - OSes using this not broadly scalable
- Modern machines provide **special atomic hardware instructions**

  Atomic = non-interruptable

  - Either test memory word and set value
  - Or swap contents of two memory words

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

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)

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)

Outline

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

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

using semaphore

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

using as counting semaphore

- **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 as binary semaphore

- **Binary semaphores**
  also known as **mutex lock**s, 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);
```

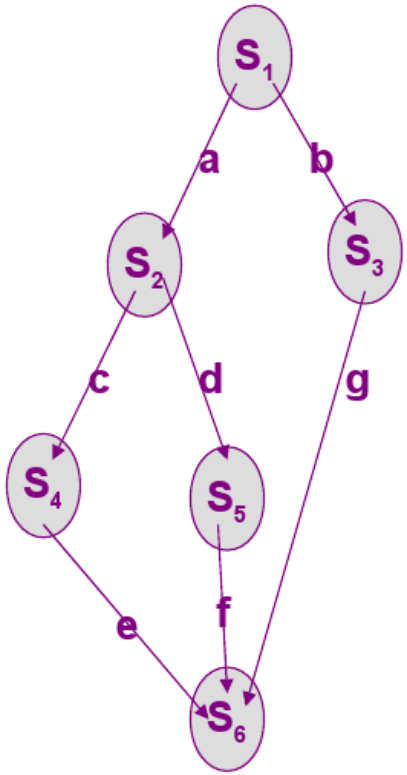
using as synchronization tools I

- 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

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 I

- **Disadvantage**: the previous semaphore may cause **busy waiting**
  - this type of semaphore is also called a **spinlock**, 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**
  - 记录型信号量, depend on **block()** & **wakeup()** operations

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

Semaphore Implementation II

- **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 III

- 分析S→value
  - 对于wait操作，开始时：
    - 当value≥1时，说明有资源剩余；只需要减1
    - 当value<1时，说明没有资源剩余；减去1，并等待
  - 对于signal操作，开始时，
    - value ≥0，说明没有等待者，不必唤醒，只需要加1
    - value<0，说明有等待者；加1，并唤醒1个进程
  - 查看value
    - value ≥0，说明没有等待者，此时，**value的绝对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
  - 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

- AND型信号量的基本思路：
  - 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和Ssignal操作

AND型信号量 II

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 I

- 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

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

Outline

小结

谢谢！