《操作系统》

进程同步

Objectives

Describe the critical-section problem and illustrate a race condition

Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables

Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem

Evaluate tools that solve the critical-section problem in low-. Moderate-, and high-contention scenarios

本节内容

Process Synchronization

- 进程的同步与互斥：概念
  - Race condition
  - Critical section
- 解决同步问题的机制：信号量
- 信号量的应用：
  - 互斥
  - 经典同步问题的解决方案

参考阅读：教材：Abraham Silberschatz等人，《操作系统概念》第六章

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  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 **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

并发进程之间的制约关系

互斥

资源共享关系：进程之间互相间接制约

- 多个进程彼此无关，完全不知道或只能间接感知其它进程的存在
- 系统须保证各进程能互斥地访问临界资源
- 系统资源应统一分配，而不允许用户进程直接使用

同步

相互合作关系：进程之间互相直接制约

- 系统应保证相互合作的诸进程在执行次序上的协调和防止与时间有关的差错

同步关系:进程合作

```
司机
while (true){
等待    启动车辆;
        正常运行;
        到站停车;
```

```
售票员
while (true){
        关门;
        售票;
等待    开门; }
```

- 应保证相互合作的进程在执行次序上的协调
- 某些操作之间要保证先后次序
- 某个操作能否进行需要满足某个条件，否则就只能等待
- 互斥关系是一种特殊的同步关系

课堂讨论

有两个并发执行的进程P1和P2，共享初值为1的变量x。P1对x加1，P2对x减1。
加1 和 减1 操作的指令序列分别如下所示：

```
load   R1 , x     // 取 x 到寄存器 R1 中
inc    R1
store x , R1      // 将 R1 的内容存入 x
```

```
load   R2 , x
dec    R2
store x , R2
```

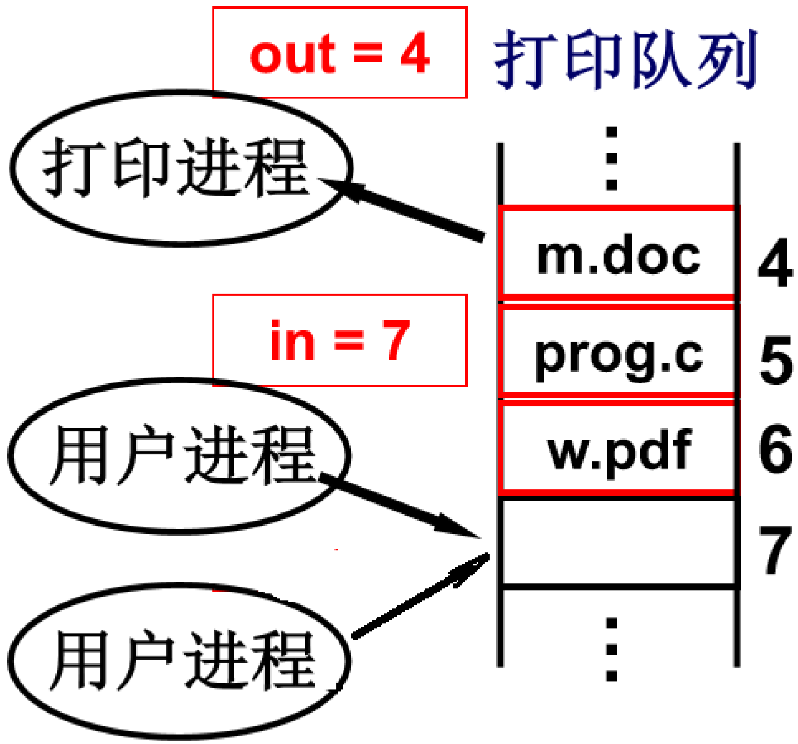
两个操作完成后，x 的值：

A.只能为 1
B. 可能为 -1 或 3
C.可能为 0 、1 或 2
D. 可能为 -1 、0 、1 或 2

并发访问共享资源实例

1. two process share “printer”

2. One process deposit, the other withdraw from an account



Race Condition

`counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

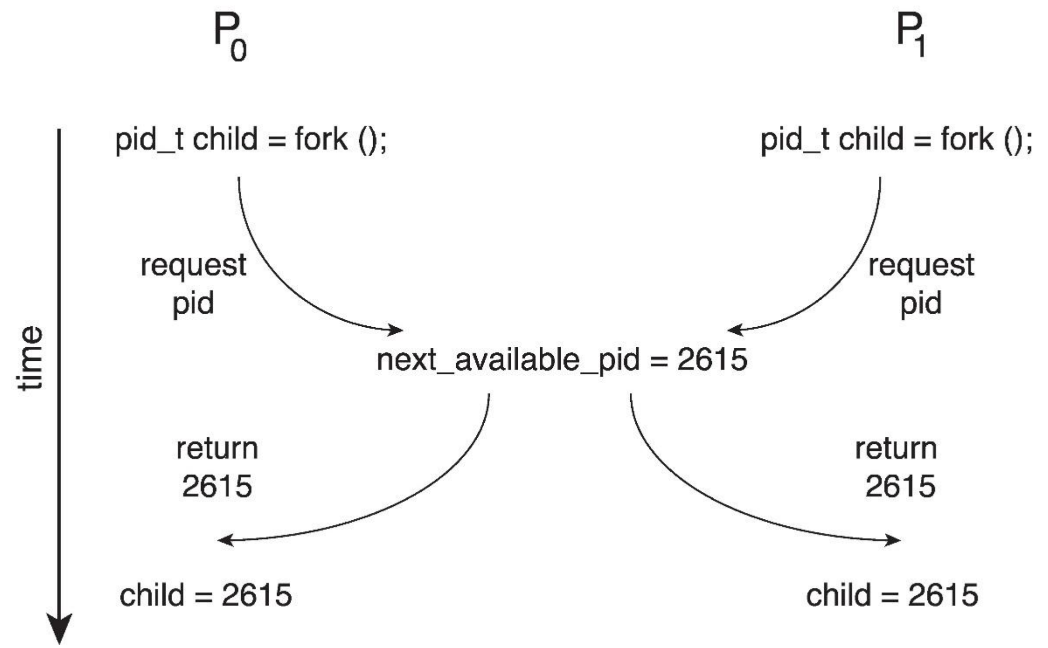
Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition

Processes P_0 and P_1 are creating child processs using the `fork()` system call

Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



Unless there is mutual exclusion, the same pid could be assigned to two different processes!

Critical Section Problem

Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$

Each process has **critical section** segment of code

- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other may be in its critical section

Critical section problem is to design protocol to solve this

Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

解决互斥的办法:临界区 Critical Section

General structure of process P_i

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

临界区访问机制的四个原则

1. **Mutual Exclusion 互斥 – 忙则等待**

   当已有进程进入自己的临界区时，所有企图进入临界区的进程必须等待

2. **Progress** - 有空让进

   当无进程处于临界区时，选择一个请求进入临界区的进程立即进入自己的临界区（选择不能无限推迟）

3. **Bounded Waiting 有限等待**

   对要求访问临界资源的进程，应保证该进程能在有限时间内进入自己的临界区

4. **让权等待**

   当进程不能进入自己的临界区时，应释放处理机

进程互斥访问临界资源的解决方案

Mechanisms for Process Synchronization：

- 软件方法
  - Peterson’s Solution（软件算法）
- 硬件方法
  - Synchronization Hardware（硬件同步）
- Semaphores（信号量）
  - Dijkstra提出的信号量机制
  - 广泛应用于单处理机、多处理机系统以及计算机网络中
- Monitors（管程）
  - 和信号量（Sophomore）等价的同步机制
  - Java语言的同步机制

先看软件方法…

Peterson’s Solution

一个经典的，基于软件的临界区问题解决方案

假设：有两个进程，共享两个数据项:

int **turn**;   // 编号（**轮到谁进入临界区**）

Boolean **flag[2]**   **//标志（准备好进入）**

flag[i] = true implies that process $\mathbf{P_i}$ is ready!

Peterson's Solution：Algorithm for Process P_i

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[ j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

1.有空让进

两者都有意愿进去时，flagi、flagj均为真，但turn取值确定，肯定会有一个进程进入临界区，所以空闲让进。

2、互斥：忙则等待

一个进程进入临界区，说明另一进程或者无意进入即flag为假、或者来晚一步。这两种情况前者flag均为真且turn 指向前者保持不变直到前者退出临界区，故而互斥。

3、有限等待?

4、让权等待?

现代处理器:Peterson算法失灵

Why Peterson's solution is not guaranteed to work correctly on modern computer architectures?

- **Most modern CPUs reorder memory accesses** to improve execution efficiency
- Memory reordering can be used to fully utilize different cache and memory banks.
- On most modern uniprocessors memory operations are not executed in the order specified by the program code.

再看硬件方法…

Synchronization Hardware

Many systems provide hardware support for critical section code

Some machines provide special atomic hardware instructions

- Atomic = non-interruptable

TestAndSet ()

Or

swap contents of two memory words: Swap()

TestAndSet Instruction

Definition:

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

Solution using TestAndSet

Shared boolean variable lock, initialized to FALSE

Solution:

```
do {
        while ( TestAndSet ( &lock ))
              ;   // do nothing

           //    critical section

        lock = FALSE;

           //    remainder section

} while (TRUE);
```

用TestandSet() 实现有限等待的互斥

```
Boolean waiting[n], lock ; global variables,  to be initialize to false
Do {
    waiting[i]=true;
    key= true;
    while ( waiting[i] && key )  key=TestAndSet(lock);
    waiting[i]=false;

    critical section;

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

}while(1)
```

waiting[i] =false 意味着另一个进程从CS退出，并将Pi设置为可进入

Key=false 意味着已经没有一个进程处于CS

寻找下一个正在等待进入CS的进程

已经没有正在等待进入CS的进程

将Pj选为下一个可进入CS的进程

用TestandSet() 实现有限等待的互斥

- **This algorithm satisfies all the critical section requirement .**
- **To prove that the mutual-exclusion（互斥条件） requirement**
  - **Pi** 可以进入**CS**只有在 **waiting[i]== false** 或 **key == false**时
  - 只有在执行了 **TestAndSet**后，**key** 才可能变为 **false**；而且只有第一个执行了 **TestAndSet** 的进程，才能得到 **key== false**，其它进程必须等待
  - 只有在一个进程离开**CS**时才会有一个（且最多仅有一个）进程的 **waiting[i]** 由**true**变为**false**
  - 从而确保满足互斥条件

用TestandSet() 实现有限等待的互斥

To prove that the progress （有空让进条件） requirement

- 任何一个已经进入CS的进程在"exit section"时，设置：lock =false 或 waiting[j]= false，确保了至少可以让一个进程进入CS

To prove that the bounded-waiting （有限等待条件） requirement

- 任何一个已经进入CS的进程Pi在"exit section"时， 将会依次扫描waiting 数组（i+1,i+2,…n-1,0,…i-1），并仅将Pi后面最先找到的进程j的waiting[j]设置为false
- 这就使进程能依此循环进入CS

Synchronization Hardware

- 硬件方法的优点
  - 适用于任意数目的进程，在单处理器或多处理器上
  - 简单，容易验证其正确性
  - 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量
- 硬件方法的缺点
  - 等待要耗费**CPU**时间，不能实现"让权等待"
  - 可能"饥饿"：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
  - 可能死锁

再看“信号量”机制…

Semaphore(信号量）

相比TestandSet()，更合适程序员使用

- 1965年，荷兰学者Dijkstra提出的信号量机制是一种卓有成效的进程同步工具。在长期广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量机制发展到记录型信号量机制，进而发展为“信号集”机制。现在信号量机制已广泛应用于OS中。
- **Synchronization tool that does not require busy waiting.** 一种不需要忙等待的同步工具.
- **Semaphore *S* – integer variable**信号量***S*** – 整型变量
- 解决**N**个进程的同步互斥问题

Semaphore

Semaphore *S* – 一个整数变量

只有两个操作可以修改S: wait() and signal()

或者 P() (from荷兰语 *proberen*, “to test”)

and V() (from荷兰语 *verhogen*, “to increment”)

two indivisible (atomic) operations ：原子操作

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

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

Semaphore as General Synchronization Tool

Binary semaphore – 二进制信号量：0 and 1

Also known as **mutex locks**

Counting semaphore – 计数信号量，integer value can range over an unrestricted domain

Can implement a counting semaphore S as a binary semaphore

二进制信号量实现互斥：

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

无“忙等待”的实现

- To overcome the problem of “busy waiting”
- Define a semaphore as a record

```
typedef struct {
    int value;
    struct process *L;
} 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);
}
```

- value 是负数，表示处于阻塞状态的进程数

信号量的应用

1. 利用信号量实现进程互斥

```
Var mutex: semaphore  = 1;
    begin
    parbegin
     process 1:     begin
                    repeat
                    wait(mutex);
                    critical section
                    signal(mutex);
                    remainder section
                    until false;
                end
```

```
process 2:
      begin
            repeat
               wait(mutex);
               critical section
               signal(mutex);
               remainder section
               until false;
      end

parend
```

课堂练习

设与某资源关联的信号量初值为 3， 当前值为 1 。若 M 表示该资源的可用个数， N 表示等待该资源的进程数，则 M 、 N 分别是:

A、0 、1

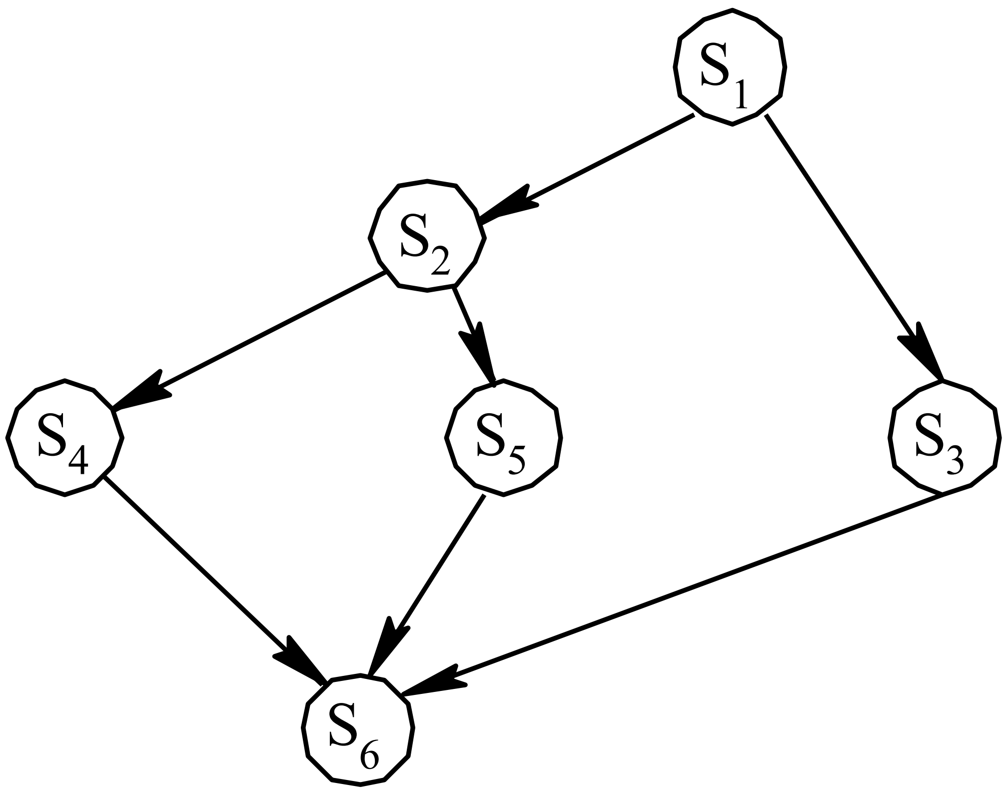
B、I 、2

C、I 、0

D、2、0

信号量的应用（续）

2. 利用信号量实现前趋关系



前趋图举例

2. 利用信号量实现前趋关系

```
var a,b,c,d,e,f,g; semaphore =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(e); end;
      begin wait(c); S4; signal(f); end;
      begin wait(d); S5; signal(g); end;
      begin wait(e); wait(f); wait(g); S6; end;
    parend
end
```

#include <semaphore.h>

```
if (sem_init(&sem,0,1) == -1)
        printf("%s\n",strerror(errno));

if (sem_wait(&sem) != 0)
        printf("%s\n",strerror(errno));

if (sem_post(&sem) != 0)
        printf("%s\n",strerror(errno));

if (sem_destroy(&sem) != 0)
        printf("%s\n",strerror(errno));
```

Badcnt.c 代码的问题？

```
volatile long cnt=0;

void *runner(void *param) {

        long i, ntiers=*((long *) param);

        for (i=0; i<ntiers; i++)
                cnt++;

        return NULL;
  }
```

goodcnt.c

```
volatile long cnt=0;
sem_t mutex;

void *runner(void *param) {
            long i, ntiers=*((long *) param);
           for (i=0; i<ntiers; i++) {
           sem_wait(&mutex);
           cnt++;
           sem_post(&mutex);
           }
           return NULL;
     }

int main(int argc, char *argv[])
{ …
            sem_init(&mutex,0,1);
…}
```

下一节

经典同步问题的解决方案

管程:Monitor

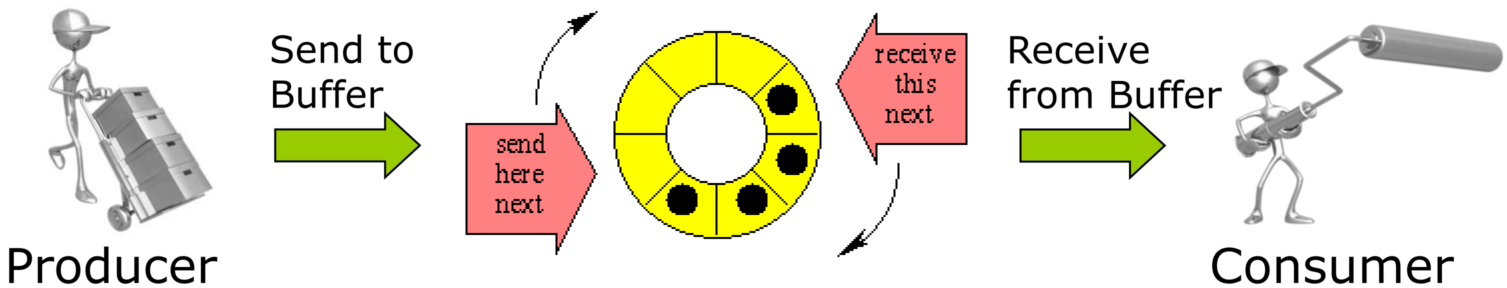
经典同步问题的解决方案

Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

Bounded-Buffer Problem



Bounded-Buffer Problem

任意两个进程（无论是生产者、还是消费者）之间，修改缓冲区时必须互斥

缓冲区满时，不能生产

缓冲区空时，不能消费