进程间通讯
Inter-Process Communication

cat chapter1 chapter2 | grep tree

问题：

1. 一个进程如何把信息传递给另一个进程(线程无此问题)

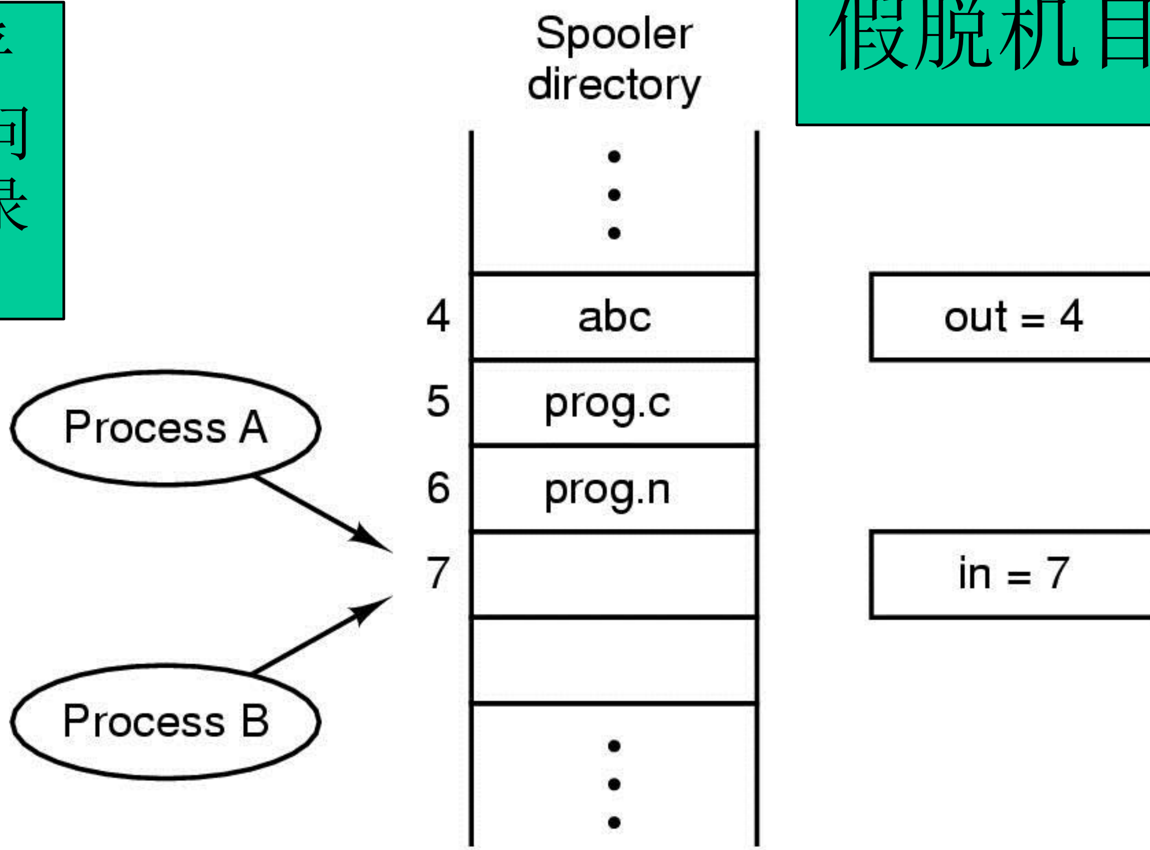
2. 关键活动，多个进程不会把事情搞乱(不交叉，卖同一张票)

3. 正确的顺序(生产者-消费者)

Race Conditions（竞争条件）

A和B几乎同时想访问in=7的目录

假脱机目录



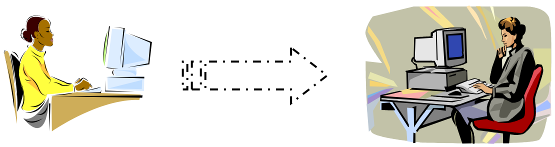
共享变量，所有进程可以访问

Two processes want to access shared memory at the same time.

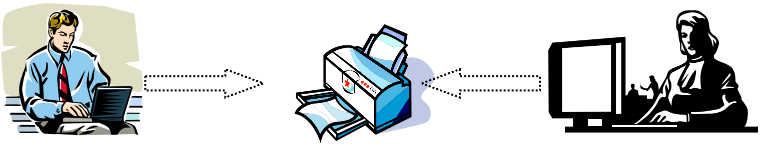
竞争条件

竞争条件：两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确顺序。

同步 synchronization
进程之间的一种通信方式，有时序上的制约关系，是进程之间为了协同工作而存在的一种等待关系



互斥 mutual exclusion
进程之间对临界资源的一种竞争关系，排他性的对资源的访问方式



临界资源是一次仅允许一个进程使用的共享资源

竞争条件

怎样避免竞争条件？

关键：找出某种途径来阻止多个进程同时读写共享的数据。

互斥！即以某种手段确保当一个进程在使用一个共享变量或文件时，其他进程不能做同样的操作。

原语？ primitive or atomic action 是由若干多机器指令构成的完成某种特定功能的一段程序，具有不可分割性。即原语的执行必须是连续的，在执行过程中不允许被中断

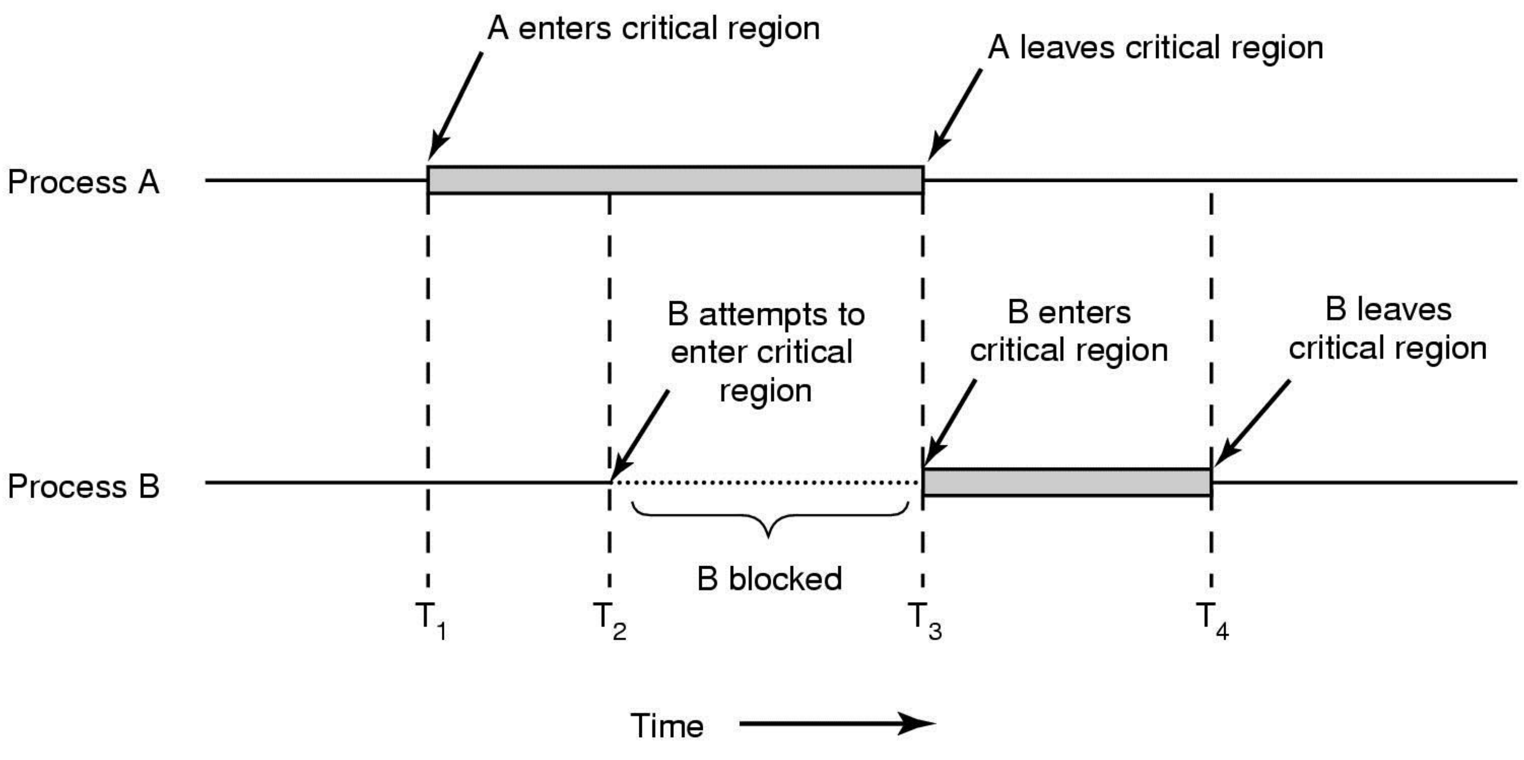
Critical Regions (临界区)

每个进程中访问临界资源的那段代码称为**临界区**。

Conditions required to avoid race condition:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

Critical Regions



Mutual exclusion（互斥） using critical regions.

Mutual Exclusion with Busy Waiting 忙等待

Proposals for achieving mutual exclusion:

- Disabling interrupts ：禁止中断
- Lock variables ：锁变量
- Strict alternation ：严格轮转
- Peterson‘s solution ：Peterson解法
- The TSL instruction ：TSL指令

Disabling interrupts: 禁止中断

方法：进程进入临界区后禁止所有中断，离开前打开中断。

缺点：

1.用户权利过大？

2.多CPU如何保证其余CPU进入临界区？

禁止中断对于OS是一项很有用的技术；

但是对于用户进程则不是合适的互斥机制。

Lock variables: 锁变量

方法：软件解决方案，设置共享（锁）变量。

0：没有进程在临界区；

1：有进程在临界区。

缺点：与假脱机类似的问题

P1读锁变量为0，在设置其值为1之前，P1时间到，P1阻塞；

P2得到CPU并进入临界区；未执行完，时间到，P2阻塞；

P1得到CPU，进入临界区。竞争条件产生

见教材关于进入前二次检查的讨论。

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1.

Strict alternation: 严格轮转

忙等待：turn被用于进程连续测试，直到出现进程自身希望的值
turn被称为自旋锁（spin lock）

缺点：浪费CPU！
在有效的CPU内，未得到turn合理值，只能忙等

严格轮转：P1未进入临界区，并退出前，P2是无法进入的！

在一个进程比另一个慢很多的情况下，严格轮转不是好办法！

Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}
```

保证不会同时进入临界区

```
void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Peterson’s solution for achieving mutual exclusion.

The TSL Instruction
需要硬件支持

```
enter_region:
    TSL REGISTER,LOCK      | copy lock to register and set lock to 1
    CMP REGISTER,#0        | was lock zero?
    JNE enter_region       | if it was nonzero, lock was set, so loop
    RET                    | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0           | store a 0 in lock
    RET                    | return to caller
```

Entering and leaving a critical region using the TSL instruction.

Test and Set Lock

（锁住内存总线，禁止其他CPU在本指令结束前访问内存）

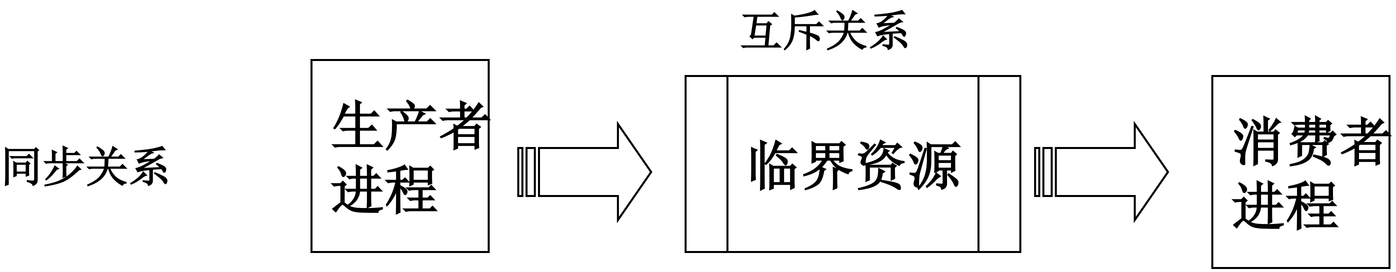
两个经典的同步/互斥问题

- 生产者与消费者

- 写者与读者

生产者与消费者问题

- 模型的抽象化与进程分析



信号量的设置

Mutex= 1　　　临界资源（互斥用信号量）

Empty= n 空缓冲区的个数（同步用信号量）

Full= 0　　满缓冲区的个数（同步用信号量）

Sleep and Wakeup

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                          /* repeat forever */
          item = produce_item( );            /* generate next item */
          if (count == N) sleep( );          /* if buffer is full, go to sleep */
          insert_item(item);                 /* put item in buffer */
          count = count + 1;                 /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);  /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                          /* repeat forever */
          if (count == 0) sleep( );          /* if buffer is empty, got to sleep */
          item = remove_item( );             /* take item out of buffer */
          count = count − 1;                 /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                /* print item */
     }
}
```

The producer-consumer problem with a fatal race condition.