操作系统　Operating System

第四章 进程与并发程序设计(2)
——同步与互斥

沃天宇
woty@buaa.edu.cn
2024年4月10日

REVIEW & EXECISES

- 在单用户系统中，有n个进程，排在就绪队列和等待队列中的进程个数的取值范围是：0~(n-1), 0~n
- 判断：
  - Y – 若系统中没有Running进程，则一定没有Ready进程
  - N – 若系统中既没有Running进程，也没有Ready进程，则系统中没有进程
  - N – 不同的进程必然对应不同的程序
  - N – 用户进程可以自行修改PCB
  - Y – 进程在运行中可以将自身状态变为阻塞状态
  - N – 子进程可以继承父进程拥有的全部资源
  - N – 中断是进程切换的充分必要条件
- 区分：父进程、子进程，主程序、子程序
- 一个进程进入阻塞状态，其执行断点保存在什么地方？如何恢复执行？

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

程序的并发执行

并发是OS的设计基础，也是所有（如，同步互斥）问题产生的原因。

- 进程的三个特征：
  - **并发：**体现在进程的执行是间断性的；进程的相对执行速度是不可测的。（间断性）
  - **共享：**体现在进程/线程之间的制约性（如共享打印机）（非封闭性）。
  - **不确定性：**进程执行的结果与其执行的相对速度有关，是不确定的（不可再现性）。

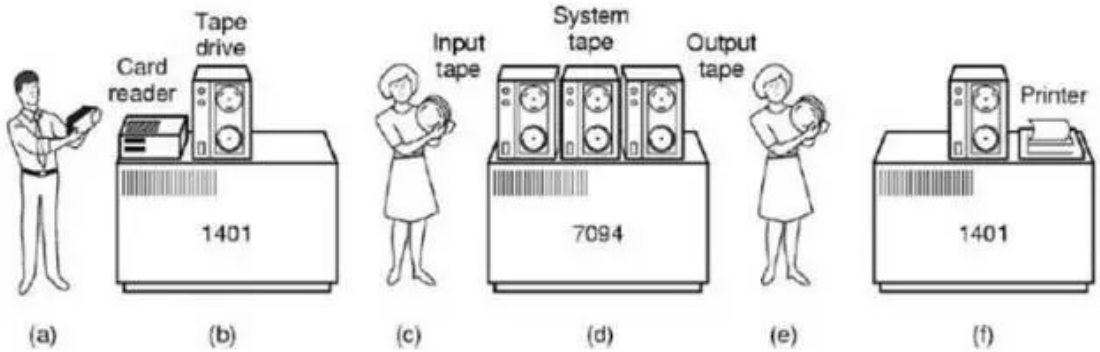
程序的并发执行

并发执行，不可避免地产生了多个进程对同一个共享资源访问，造成了资源的争夺。

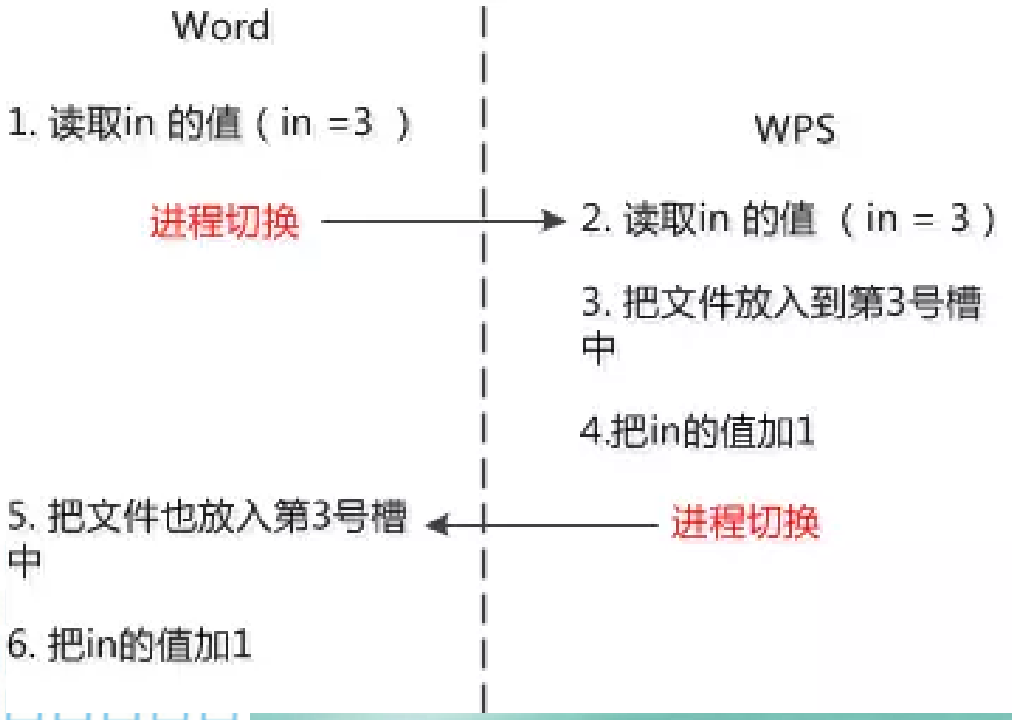
- 竞争：两个或多个进程对同一共享数据同时进行访问，而最后的结果是不可预测的，它取决于各个进程对共享数据访问的相对次序。这种情形叫做竞争。
- 竞争条件：多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关。
- 临界资源：我们将一次仅允许一个进程访问的资源称为临界资源。
- 临界区：每个进程中访问临界资源的那段代码称为临界区。

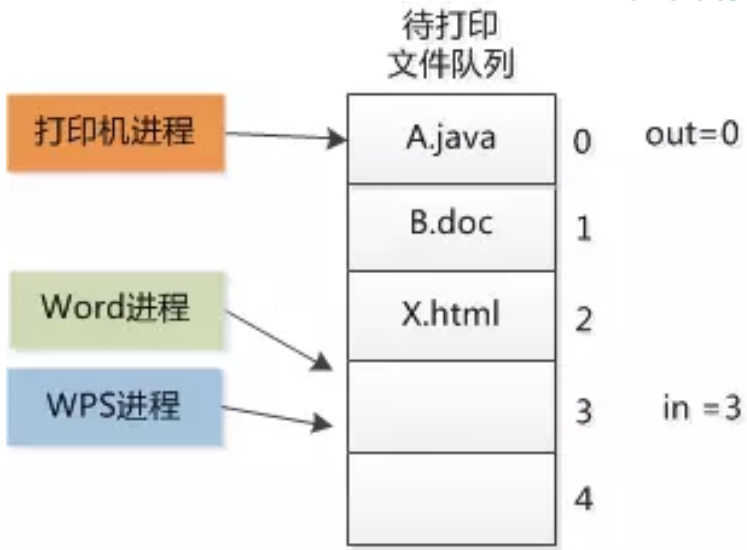
临界资源

- 一次只允许一个进程使用的资源，如打印机。



Spooling系统





临界区

- 每个进程中访问临界资源的那段代码称为临界区。

P：

……

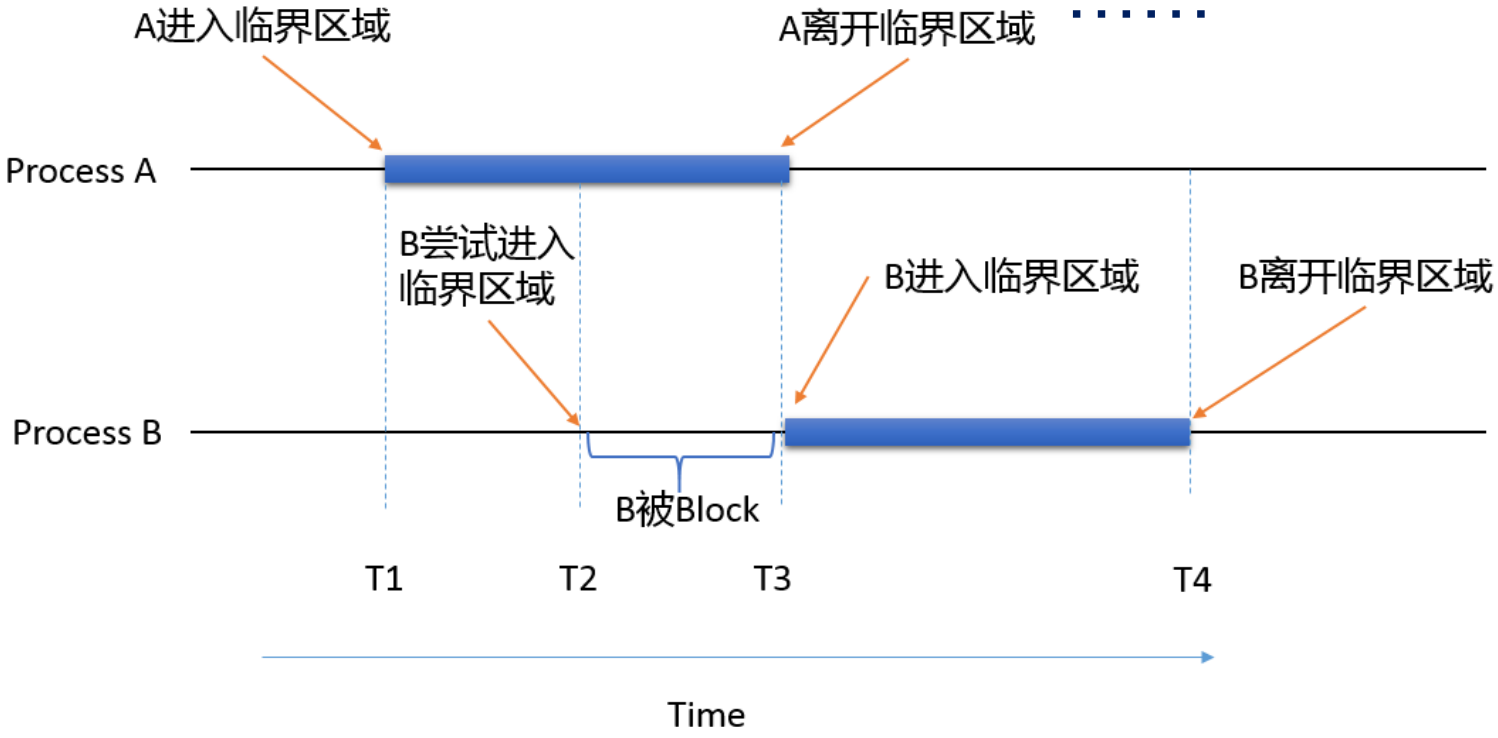
Entry section

critical section

exit section

remainder section

……



进程的同步与互斥

- 进程互斥（间接制约关系）：
  - 两个或两个以上的进程，不能同时进入关于同一组共享变量的临界区域，否则可能发生与时间有关的错误，这种现象被称作进程互斥。
  - 进程互斥是进程间发生的一种间接性作用，一般是程序不希望的。
- 进程同步（直接制约关系）：
  - 系统中各进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性的过程称为进程同步。
  - 进程同步是进程间的一种刻意安排的直接制约关系。即为完成同一个任务的各进程之间，因需要协调它们的工作而相互等待、相互交换信息所产生的制约关系。

同步与互斥的区别与联系

- 互斥：某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。互斥无法限制访问者对资源的访问顺序，即访问是无序访问。

- 同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有对资源的写入的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

互斥区管理应满足的条件

1. 没有进程在临界区时，想进入临界区的进程可进入。
2. 任何两个进程都不能同时进入临界区（Mutual Exclusion）；
3. 当一个进程运行在它的临界区外面时，不能妨碍其他的进程进入临界区（Progress）；
4. 任何一个进程进入临界区的要求应该在有限时间内得到满足（Bounded Waiting）。

正确、公平、效率

机制设计上应遵循的准则

- 空闲让进：临界资源处于空闲状态，允许进程进入临界区。如，临界区内仅有一个进程执行
- 忙则等待：临界区有正在执行的进程，所有其他进程则不可以进入临界区
- 有限等待：对要求访问临界区的进程，应在保证在有限时间内进入自己的临界区，避免死等。
- 让权等待：当进程（长时间）不能进入自己的临界区时，应立即释放处理机，尽量避免忙等。

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
  - 软件方法
  - 硬件方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

软件方法尝试1

P:

……

while(Occupied); ①

Occupied=true;

临界区 ③

Occupied=false;

……

Q:

……

while(Occupied); ②

Occupied=true;

临界区

Occupied=false;

……

Occupied：临界区空满标志

true：临界区内有进程

false：临界区内无进程

Occupied的初值为false

问题：先检查有无标志，后留标志，造成一个空挡，不能实现互斥。

软件方法尝试2

```
P:
……
while(turn==Q);
turn=P;
      临界区
turn=Q;
……
```

```
Q:
……
while(turn==P);
turn=Q;
      临界区
turn =P;
……
```

turn：进程进入临界区的优先标志。
turn的初值为P或Q（对两个进程而言就是0或1）

问题：固然实现了互斥，但要求两进程严格交替进入临界区。否则，一个临界区外的进程会阻止另一个进程进入临界区，违反了progress原则。

软件方法尝试3

P:

……

pturn=true;

while(qturn);

临界区

pturn=false;

……

Q:

……

qturn=true;

while(pturn);

临界区

qturn=false;

……

pturn, qturn：进程需要进入临界区的标志，初值为false

P进入临界区的条件：pturn∧not qturn。

Q进入临界区的条件：qturn∧not pturn

问题：竞争时都可能无法进入临界区，违反了progress原则。

软件方法尝试4

P:

```
……
pturn=true;
while(qturn)
    pturn=false;
pturn=true;
```

临界区

```
pturn=false;
……
```

Q:

```
……
qturn=true;
while(pturn)
    qturn=false;
qturn=true;
```

临界区

```
qturn=false;
……
```

问题：增加了让权等待，导致互斥规则被破坏。

Dekker算法

引入变量turn，以便在竞争时选出进入临界区的进程

P:

```
……
pturn=true;
while(qturn)  {
   if(turn ==1) {
         pturn=false;
         while(turn==1);
         pturn=true;
   } }
   临界区

turn = 1;
pturn=false;
……
```

让权

缺点：忙等浪费CPU时间

Q:

```
……
qturn=true;
while(pturn)  {
  if(turn ==0) {
         qturn=false;
         while(turn==0);
         qturn=true;
   } }
   临界区

turn = 0;
qturn=false;
……
```

1965年第一个用软件方法解决了临界区问题

Peterson算法

```
#define FALSE 0
#define TRUE  1
#define N     2                  // 进程的个数
int turn;                        // 轮到谁?
    int interested[N];
   // 兴趣数组，初始值均为FALSE

void enter_region ( int process)
                    // process = 0 或 1
    {
        int other;
              // 另外一个进程的进程号
        other = 1 - process;
        interested[process] = TRUE;
             // 表明本进程感兴趣
        turn = process;
             // 设置标志位
        while( turn == process &&
    interested[other] == TRUE);
    }
```

循环

```
void leave_region ( int process)
    {
        interested[process] = FALSE;
              // 本进程已离开临界区
    }
```

```
进程i:
    … …
    enter_region ( i );
       临界区
    leave_region ( i );
    … …
```

Peterson算法解决了互斥访问的问题，而且克服了强制轮流法的缺点，可以完全正常地工作（1981）

N个进程互斥的软件算法

- 实现进程互斥的软件的结构框架是：

  Repeat

  entry section

  critical section

  exit section

  remainder section

  Until false

- 进程互斥的软件实现算法有：Lamport面包店算法和Eisenberg算法。这两种算法均假定系统中进程的个数有限，如n个。

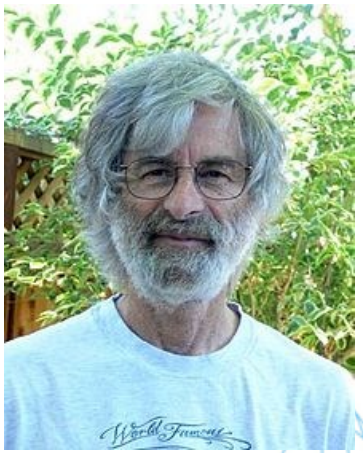
Lamport Bakery Algorithm

面包店算法（ Bakery Algorithm ）的基本思想来源于顾客在面包店购买面包时的排队原理。顾客进入面包店前，首先抓取一个号码，然后按号码从小到大的次序依次进入面包店购买面包，这里假定：

- (1)—面包店按由小到大的次序发放号码，且两个或两个以上的顾客有可能得到相同号码（要使顾客的号码不同，需互斥机制）；
- (2)—若多个顾客抓到相同号码，则按顾客名字的字典次序排序 i.e., 1,2,3,3,3,3,4,5... （假定顾客没有重名）。

Lamport Bakery Algorithm

- 计算机系统中，顾客相当于进程，每个进程有一个唯一的标识，用Pi表示，对于Pi和Pj，若有i<j，即Pi先进入临界区，则先为Pi服务。
- 基本思想：设置一个发号器，按由小到大的次序发放号码。进程进入临界区前先抓取一个号码，然后按号码从小到大的次序依次进入临界区。若多个进程抓到相同的号码则按进程编号依次进入。



Leslie Lamport
2013 Turing Award

- **面包店算法是1974年由莱斯利·兰波特给出的。著名的排版软件LaTeX也是他的贡献。**

数据结构

int choosing[n]; //表示进程是否正在抓号，初值为0。
若进程i正在抓号，则choosing[i]=1.

int number[n]; //记录进程抓到的号码，初值为0。
若number[i]=0，则进程i没有抓号

排序方法：字典序

(a, b)＜(c, d) → (a＜c) or ((a == c) and (b＜d))

初始值：

Choosing，Number：array [1..N] of integer = {0};

Bakery Algorithm (for Pi)

```
Entry Section (i) {    // i → process i
    while (true) {
        Choosing[i] = 1;
        Number[i] = 1 + max(Number[1],...,Number[N]);
        Choosing [i] = 0;
        for (j=1; j<=N; ++j) {
              while (Choosing[j] != 0)  { }
              // wait until process j receives its number
              while ((Number[j]!=0) && ((Number[j],j) <(Number[i],i))) { }
              // wait until processes with smaller numbers, or with the
              // same number, but with higher priority, finish their work
        }
            // critical section...
        Number[i] = 0;
          // non-critical section...
    }
}
```

Bakery算法的说明

- 当进程Pi计算完max(…)+1但尚未将值赋给number[i]时，进程Pj中途插入，计算max(…)+1，得到相同的值。在这种情况下，Choosing[j]的使用可保证编号较小的进程先进入临界区，并不会出现两个进程同时进入临界区的情况。
- 忙式等待：上述Lamport面包店算法中，若while循环的循环条件成立，则进程将重复测试，直到条件为假。实际上，当while循环条件成立时，进程Pi不能向前推进，而在原地踏步，这种原地踏步被称为忙式等待。忙式等待空耗CPU资源，其它进程无法使用，因而是低效的。

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
  - 软件方法
  - 硬件方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

硬件方案1：中断屏蔽

中断屏蔽方法：使用“开关中断”指令。

- 执行“关中断”指令，进入临界区操作；
- 退出临界区之前，执行“开中断”指令。

优缺点：

- 简单。
- 不适用于多CPU系统：往往会带来很大的性能损失；
- 单处理器使用：很多日常任务，都是靠中断的机制来触发的，比如时钟，如果使用屏蔽中断，会影响时钟和系统效率，而且用户进程的使用可能很危险！
- 使用范围：内核进程（少量使用）。

硬件方案2：使用test and set指令

- TS（test-and-set ）是一种不可中断的基本原语（指令）。它会写值到某个内存位置并传回其旧值。在多进程可同时存取内存的情况下，如果一个进程正在执行检查并设置，在它执行完成前，其它的进程不可以执行检查并设置。
- Test and Set指令
  - IBM370系列机器中称为TS；
  - 在INTEL8086中称为TSL。
- 语义：

```
TestAndSet(boolean_ref lock) {
    boolean  initial = lock;
    lock = true;
    return initial; }
```

自旋锁Spinlocks

- 利用test_and_set硬件原语提供互斥支持
- 通过对总线的锁定实现对某个内存位置的原子读与更新

```
acquire(lock) {
   while(test_and_set(lock) == 1);
}

        critical section

release(lock) {
    lock = 0;
}
```

Spinlocks

0

```
acquire(int *lock) {
    while(test_and_set(lock) == 1)
        /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```

Let me in!!!

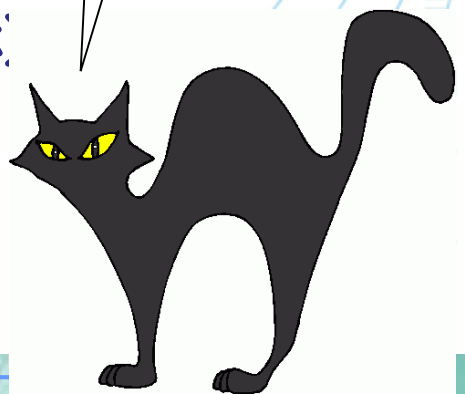
No, Let *me* in!!!

```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

```
acquire(houselock);
Nap_on_couch();
Release(houselock);
```

1

1



Spinlocks

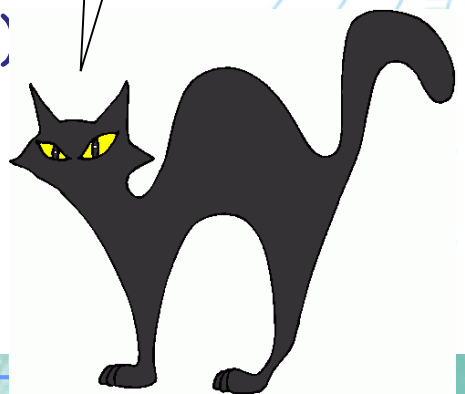

```
acquire(int *lock) {
   while(test_and_set(lock) == 1)
            /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```

Yay, couch!!!

I still want in!

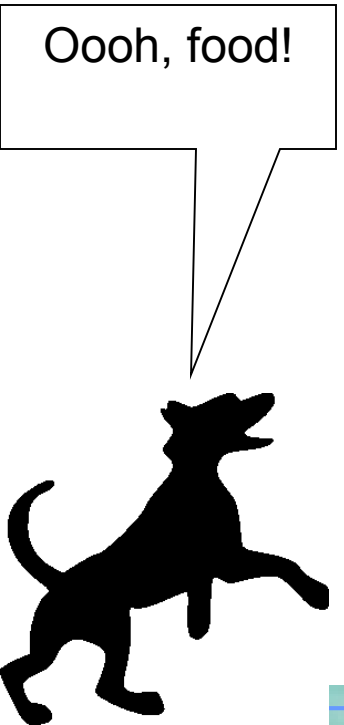
```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

```
acquire(houselock);
Nap_on_couch();
Release(houselock);
```



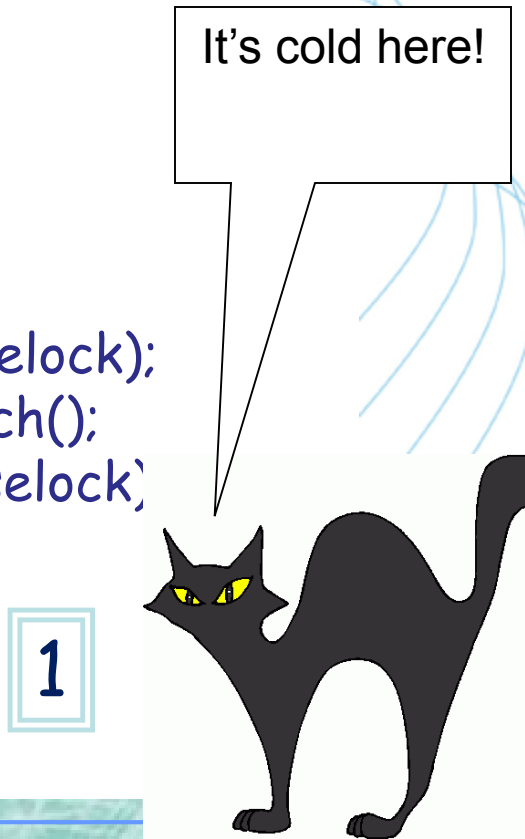
Spinlocks


```
acquire(int *lock) {
   while(test_and_set(lock) == 1)
            /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```




```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

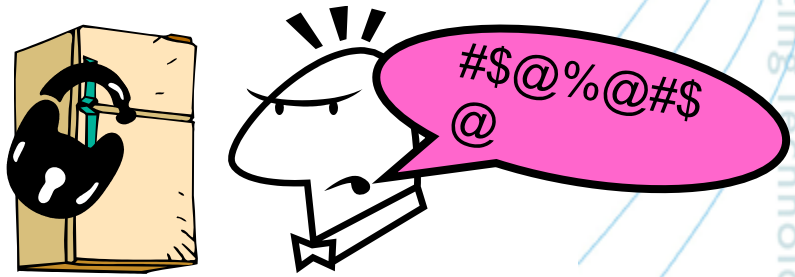
```
acquire(houselock);
Nap_on_couch();
Release(houselock);
```



X86基于TSL的自旋锁汇编代码

```
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 non zero, 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
```



硬件方案3：使用swap指令

- Swap（对换）指令与TSL指令一样，是不会被中断的原子指令，其功能是交换两个字的内容，其语义如下：

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

- 在INTEL8086中为XCHG。

硬件方案3：使用swap指令

使用Swap指令实现进程互斥的描述如下：

```
Boolean k = true;                    // 初始化为1
Boolean use = false;                 // 初始资源空闲
while（k != 0）
        Swap（&use, &k）; //进入区
Critial_region（）;                  //临界区
Use = 0;                                    //退出区
Other_region（）;                    //剩余区
```

- 采用Swap指令与采用TSL指令类似，也会由于循环对换两个变量，造成忙等的情况。

MIPS中的spinlock

指令说明	指令定义	Alpha	MIPS64
寄存器和内存之间原子交换（用在锁和信号量上）	Temp<---Rd; Rd<---Mem[x]; Mem[x]<---Temp	LDL/Q_L; STL/Q_C	LL; SC

- MIPS 提供了 LL（Load Linked Word） 和 SC（Store Conditional Word） 这两个汇编指令来实现Spinlock(对共享资源的保护)。
- Read-modify-write 操作:当我们对某个Mem进行操作时, 我们首先从该 Mem中读回(read) data, 然后对该 data进行修改(modify)后, 再写回(write)该Mem。

MIPS中的spinlock

- LL 指令的功能是从内存中读取一个字,以实现接下来的RMW（Read-Modify-Write） 操作;
- SC 指令的功能是向内存中写入一个字,以完成前面的RMW 操作。
- LL/SC 指令的独特之处:
  - 当使用 LL 指令从内存中读取一个字后，如 *LL d, off(b)*，处理器会记住 LL 指令的这次操作（会在 CPU 的寄存器中设置一个不可见的 bit 位），同时 LL 指令读取的地址 *off(b)* 也会保存在处理器的寄存器中。
  - SC 指令，如 *SC t, off(b)*，会检查上次 LL 指令执行后的 RMW 操作是否是原子操作（即不存在其它对这个地址的操作），如果是原子操作，则 *t* 的值将会被更新至内存中，同时 *t* 的值也会变为1，表示操作成功；反之，如果 RMW 的操作不是原子操作（即存在其它对这个地址的访问冲突），则 *t* 的值不会被更新至内存中，且 *t* 的值也会变为0，表示操作失败。

MIPS中spinlock

```
Spin_Unlock(lockkey)
    sync
    sw zero, lockkey   //给 lockKey 赋值为 0，
                       表示这个锁被释放
```

```
Spin_Lock(lockkey)    // lockKey 是共享资源锁
1:
    ll t0, lockkey      //将 lockKey 读入t0寄存器中
    bnez t0, 1b         //比较 lockKey 是否空闲，如果该锁不可
                          用的话则跳转到1:
    li t0, 1            //给 t0 寄存器赋值为1
    sc t0, lockkey      //将 t0 寄存器的值保存入 lockKey 中，
                          并返回操作结果于 t0 寄存器中。
    beqz t0, 1b         //判断 t0 寄存器的值，为0表示 li的操作失
                          败，则返回 1:重新开始；为1表示 li的操
                          作成功。
    sync                // Sync 是内存操作同步指令，用来保证 sync
                          之前对于内存的操作能够在 sync 之后的指
                          令开始之前完成。
```

几个算法的共性问题

- 无论是软件解法（如Peterson）还是硬件（如TSL或XCHG）解法都是正确的，它们都有一个共同特点：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则该进程将原地等待，直到允许为止。

1. 忙等待：浪费CPU时间

2. 优先级反转：低优先级进程先进入临界区，高优先级进程一直忙等