吴雨朗 22920192204097

厦門大學
XIAMEN UNIVERSITY
ADD:FUJIAN XIAMEN CABLE:0633 P.C:361005

5.3

解: a. P1 执行 x=x-1;
P1 执行 x=x+1;
P2 执行 x=x-1;
P1 执行 if(x!=10)
P2 执行 x=x+1;
P1 执行 printf("x is %d", x);

这样 P1 就打印出了 "x is 10".

b.

指令	P1:R0	P2:R0
P1: LD R0, x	10	—
P1: DECR R0	9	—
P1: STO R0, x	9	—
P2: LD R0, x	9	9
P2: DECR R0	9	8
P2: STO R0, x	9	8
P1: LD R0, x	8	8
P1: INCR R0	9	—
P2: LD R0, x	9	8
P2: INCR R0	9	9
P2: STO R0, x	9	9
P2: if(x!=10) printf("x is %d", x);		
P1: STO R0, x	9	9
P1: if(x!=10) printf("x is %d, x);		
P1: LD R0, x	9	9
P1: DECR R0	8	—
P1: STO R0, x	8	—

P2: LD R0, x	8	8
P2: DECR R0	8	7
P2: STO R0, x	8	7
P1: LD R0, x	7	7
P1: INCR R0	8	7
P1: STO R0, x	8	7

P1: if(x!=10)
printf("x is %d", x);

运行 P1 则打印出为 "x is 8"

5.4

解: a. tally 的下界是2，上界是100.

tally=2 的情况如下：

① 进程P1载入 tally 值，tally 值加到1，在此时失去处理机，且此时 tally 值未存储

② 进程P2载入tally值，此时tally值为0，运行完4个次自加1操作，将4存储到 tally 中后，失去处理机控制权.

③ 进程P1重新获得处理机控制权完成它的第一次存储，将1存入tally，然后失去处理机控制权.

④ 进程P2重新开始运行，当tally当前值1载入自己的存储区，失去处理机.

⑤ 进程P1重新开始运行，运行完4个次自加1操作，将5存储到 tally 中，失去处理机.

⑥ 进程P2将自己存储区中的值加1，得到2，存入共享变量 tally 中，得到 tally 的最终结果2.

tally = 100 的情况如下：

① 进程P1载入tally值，运行完50次自加1操作，将50存储到tally中后，失去处理器控制权.

② 进程P2载入tally值，运行完50次自加1操作，将得到的100存入共享变量tally中，得到tally的最终结果100.

b. 当有N个进程并行执行时，tally值的最终取值范围是[2, 50N]，因为在tally=2情况下的第①步，除了进程PN外的所有进程得到的结果都可能在最后被进程PN破坏掉.

5.5

解：一般情况下等待的效率比阻塞等待的效率低. 但有一种特殊情况，当进程执行到程序的某一点处，在此处需等待直到条件满足，在某个时刻条件已满足，此时忙等待会立即有结果，而阻塞等待却会消耗资源在进程的换出与换入上.

5.7

解：a. 当一个进程想进入临界区时，它被分配到一个票号，这个票号是那些等待进入临界区进程和已经在临界区进程所持票号的最大值加1. 有最小票号的进程有最高优先级进入临界区. 如果多个进程得到相同的票号，进程号最小的进程进入临界区. 当一个进程退出临界区时，重新设置它的票号为0.

b. 因为每个进程的进程号是不同的，所以会有一个唯一的访问临界区的进程顺序，所以这个算法避免了死锁.

c. 要证明它实现了互斥，先证明如果进程Pi在它的临界区，而进程Pk已经计算出了它的number[k]，并试图进入临界区，此时下列成立：

(number[i], i) < (number[k], k)

为证明互斥引理，定义如下时刻：

T_{i1}：在 $j=k$ 时，P_i 最后一次读 choosing[k]，也是它的第一次等待，可以在 T_{i1} 时有 choosing[k] = false.

T_{i2}：在 $j=k$ 时，P_i 开始第二个 while 循环的最终执行，可以有 $T_{i1} < T_{i2}$

T_{k1}：P_k 进程进入该循环的开始.

T_{k2}：P_k 完成 number[k] 的计算

T_{k3}：P_k 把 choosing[k] 置为 false，有 $T_{k1} < T_{k2} < T_{k3}$

在 T_{i1} 时刻，choosing[k] = false，可以有 $T_{i1} < T_{k1}$ 或 $T_{k3} < T_{i1}$. 在第一种情况下，有 number[i] < number[k]. 因为 P_i 在 P_k 之前被分配了序号，可以满足引理的条件. 在第二种情况下，有 $T_{k2} < T_{k3} < T_{i1} < T_{i2}$，因此 $T_{k2} < T_{i2}$，可以在 T_{i2} 处，P_i 读取了 number[k] 的当前值，而且 T_{i2} 是 $j=k$ 的第二个 while 的最终执行时刻，有 (number[i], i) < (number[k], k)，可以完成了引理的证明.

接下来证明算法实现了互斥. 假设 P_i 进入了临界区，P_k 试图要进入临界区，但因为 number[i] ≠ 0 且 (number[i], i) < (number[k], k)，所以 P_k 无法进入临界区. 可以算法实现了互斥. 证毕.

5.11

解：

```
var   j: 0…n-1;
      key: boolean;
while (true)
{
   wait[i] = true;
   key: =true;
   while (wait[i] && key)  key = (compareandswap(lock, 0, 1) == 0);
   wait[i] = false;
   /*临界区*/
```

```
    j:= i+1 mod n;
    while (j!=i && !wait[j]) j:= j+1 mod n;
    if (j==i) lock:= false;
    else wait:= false;
    /* 其余部份 */
}
```

5.12

解：这两种定义的效果是相同的。可以在不改变程序意义的前提下，用一个定义代替另一个。在图5.3的定义中，当信号量的值为负数时，它的绝对值表示现在有多少个进程在等待。这两种定义的功能是相同的。

5.18

解：不正确。有一个问题，V(passenger_released)可以解除一名旅客的阻塞，该旅客被阻塞在P(passenger_released)里，他不是生在执行V()的车里的旅客。

5.21

解：a,b,c,d这4种互换都会导致程序错误。信号量s控制着对临界区的访问。append()和take()是临界区，各应被semWait(s)和semSignal(s)包围起来。

5.23

解：
```
int troubles=0, deers=0; //troubles为遇到问题并且可以得到帮助的求援人数，
                         //deers为回到家里的驯鹿的数量.
bool flag[2]={false, false}; // flag[]表示驯鹿和驯鹿进程是否发出了唤醒圣诞老人
                             //的信号.
bool delay=false; //表示是否需要延期
```

semaphore x=0, y=0, z=0, solved=0, work=0, to=0, t1=0, S=1;

//solved 控制其他精灵等待圣诞老人解决完三个小精灵的问题.

//x 控制 solved 队列上有一个小精灵在等待，其他小精灵在x队列上等待.

//y保证 troubles 的互斥.

//z保证 deers 的互斥.

// work 控制圣诞老人的醒来.

//t0,t1分别保证 flag[0], flag[1]的互斥.

//S保证圣诞老人一次只做一件事，帮小精灵解决问题或帮驯鹿等雪橇.

```
wid  child()
{
    while(1){
            semWait(x);
            if (trouble==3) semWait(solved);
            semSignal(x);

            semSignal(y);
            trouble ++;
            if (trouble==3){

                semSignal(work);  //满足了3个小精灵有问题的条件，叫醒圣诞老人.
                semWait(to);
                flag[0]=true;  //小精灵叫醒圣诞老人
                semSignal(to);
            }
            semSignal(y);
    }
}
```

厦門大學
XIAMEN UNIVERSITY
ADD:FUJIAN XIAMEN CABLE:0633 P.C:361005

```
void reindeer()
{
   while(1){
        semWait(z);
        deers++;
        if(deers==9){
            delay=true; //9只驯鹿都回来，准备雪橇更重要
            semSignal(work) //9只驯鹿都到家，叫醒圣诞老人
            semWait(t1);
            flag[1]=true; //是驯鹿叫醒圣诞老人
            semSignal(t1);
        }
        semSignal(z);
   }
}

wid senta()
{
   while(1){
       semWait(work); //圣诞老人等待被叫醒.
       if(flag[0]){  //小精灵发出唤醒信号
           if(!delay){ // 最后一只驯鹿没回来
               semWait(t0);
               flag[0]=false;
               semSignal(t0);

               semWait(s);
               solve();
               semWait(y);
               tramble=0;
               semSignal(y));
               semSignal(solved); //问题解决，其他小精灵解除等待.
               semSignal y));
           }
       }
```

```
if (flag[1]) { // 到底唤醒圣诞老人
    semWait(t1);
    flag[1]=false;
    semSignal(t1);
    semWait(s);
    install-sled();
    semSignal(s);
   }
  }
}
```

5.24

解: a. 用信号量实现消息传递.

定义 mbuf 为可用的消息槽总数, own 和 dest 是每个进程的消息队列, 最初为零.

```
send (message, dest)
semWait (mbuf)
semWait (mutex)
/* 获得自由缓冲区 */
/* 将邮件复制到插槽 */
/* 将插槽连接到其他消息 */
semSignal (dest.sem) //唤醒目的地进程
semSignal (mutex)

receive (message)
semWait (own.sem)
semWait (mutex)
/* 解除插槽与 own.queue 的连接 */
/* 将缓冲槽复制到消息 */
/* 将缓冲槽添加到自由列表 */
semSignal (mbuf)
semSignal (mutex)
```

6. 用消息传递实现信号量

同步进程为每个信号量维护一个计数器和一个等待进程的链表。为了执行WAIT或SIGNAL，进程调用相应的库程序WAIT或SIGNAL，向同步进程发送一条消息，指定所需的操作和要使用的信号量。然后，库程序执行接收操作，从同步进程中获取回复。消息到达时，同步进程会检查计数器，以确定所需的操作是否可以完成。SIGNALs始终可以完成，但如果信号量的值为0，WAITs将被阻止。如果操作是WAIT且信号量为0，则同步进程进入阻塞队列，且不发送回复。结果是执行WAIT的进程被阻塞。当SIGNAL被执行，同步进程选择一个进程在信号量上阻塞，要么以先进先出的顺序，要么以其他顺序，并且发送一个回复。这里避免了竞争条件，因为同步进程每次只处理一个请求。

5.25

解：错在读者进程可能会使写者进程处于永久饥饿状态，因为读者进程可能永远都不会离开临界区，即临界区中始终都有一个读者进程，因此wrt信号量可能永远不会发送给写者进程，写者进程就无法得到wrt信号量并进入临界区。