OS 第三章作业

191220008 陈南疃

问答题

1、Does Peterson's solution to the mutual-exclusion problem shown in Fig. 2-24 work when process scheduling is preemptive? How about when it is nonpreemptive?

```
#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 */ ;
}

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

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Peterson算法适用于抢占式进程调度，因为Peterson算法本就是实现临界区管理的软件方法。（但若出现下一题中的优先级反转现象时，则不适用）

但Peterson算法不适用于非抢占式进程调度，因为当进程间只能串行时，每次进行while循环条件判断时，因interested[other]被一定为FALSE，所以永远不会进入while循环。因此，每个进程会直接进入临界区，故临界区缺少保护，不再适用。

2、In Sec. 2.3.4, a situation with a high-priority process, *H*, and a low-priority process, *L*, was described, which led to *H* looping forever. Does the same problem occur if round-robin scheduling is used instead of priority scheduling? Discuss.

2.3.4 Sleep and Wakeup

Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the defect of requiring busy waiting. In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.

Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, *H*, with high priority, and *L*, with low priority. The scheduling rules are such that *H* is run whenever it is in ready state. At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes). *H* now begins busy waiting, but since *L* is never

scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever. This situation is sometimes referred to as the **priority inversion problem**.

当使用轮转调度算法时，不会出现这种情况。因为每个进程都以时间片为单位进行调度执行，则H进程不会一直占有CPU，所以L进程一定能在若干时间片后退出临界区。

3、Consider the following solution to the mutual-exclusion problem involving two processes *P0* and *P1*. Assume that the variable turn is initialized to 0.

```
/* Other code */

while (turn != 0) { } /* Do nothing and wait. */
Critical Section /* . . . */
turn = 0;

/* Other code */
```

Process *P0*'s code is presented below. For process *P1*, replace 0 by 1 in above code. Determine if the solution meets *all* the required conditions for a correct mutual-exclusion solution.

该方法不适用于临界区问题。

因为turn变量初始化为0，因此P0进程可以顺利进入临界区，退出临界区时turn的值仍为0；而P1进程除了退出临界区时能将turn置为1，其他地方无法将turn置为1，所以始终满足while(turn != 1)的条件，将会永久等待，无法进入临界区。

应用题

3. 有两个优先级相同的进程 P1 和 P2,其各自执行的操作如下,信号量 S1 和 S2 的初值均为 0。试问 P1、P2 并发执行后,x、y、z 的值各为多少?

```
P1() {                          P2() {
    y = 1;                          x = 1;
    y = y + 3;                      x = x + 5;
    V(S1);                          P(S1);
    z = y + 1;                      x = x + y;
    P(S2);                          V(S2);
    y = z + y;                      z = z + x;
}                               }
```

由代码中的PV语句可知，V(S1)执行后P(S1)之后的代码才能被执行，V(S2)执行后P(S2)之后的代码才能被执行。在P(S1)或V(S1)之前的语句顺序对结果没有影响，因此可以得到x=6，y=4。

接下来考虑剩下四个语句的顺序：

```
z = y + 1 ...(1)
y = z + y ...(2)
x = x + y ...(3)
z = z + x ...(4)
```

已知(1)在(2)之前，(3)在(4)之前。

此外，(3)对(1)和(2)的运算结果没有影响，所以只需考虑(1)，(2)，(4)之间的顺序：

① (1)→(2)→(4)

x = 10，y = 9，z = 15

② (1)→(4)→(2)

x = 10，y = 19，z = 15

③ (4)→(1)→(2)

x = 10，y = 9，z = 5

22. 现有 k 个进程,其标号依次为 $1,2,\cdots,k$,如果允许它们同时读文件 file,但必须满足条件:参加同时读文件的进程的标号之和要小于正整数 $M(k<M)$。请使用(1)信号量与 PV 操作、(2)管程,编写出协调多进程读文件的程序。

(1) 信号量与PV操作

```
semaphore waits;
semaphore mutex;
waits = 0;
mutex = 1;
int numbersum = 0;
cobegin
    process readeri(int number) {           //i = 1, 2, ...
```

```
        P(mutex);
        while(numbersum + number >= M) {
            V(mutex);
            P(waits);
            P(mutex);
        }
        numbersum = numbersum + number;
        V(mutex);
        /*读文件*/
        P(mutex);
        numbersum = numbersum - number;
        V(waits);
        V(mutex);
    }
coend
```

(2) 管程

```
type sharefile = MONITOR {
    int numbersum; numbersum = 0;
    int SF_count; SF_count = 0;
    cond SF; SF= 0;
    Interface Module IM
    define startread, endread;
    use enter, leave, wait, signal;
}
procedure startread(int number) {
    enter(IM);
    while(number + numbersum >= M) {
        wait(SF, SF_count, IM);
    }
    numbersum = numbersum + number;
    leave(IM);
}
procedure endread(int number) {
    enter(IM);
    numbersum = numbersum - number;
    signal(SF, SF_count, IM);
    leave(IM);
}
cobegin
    process-i ( );
coend

process-i ( ) {
    int number;
    number = 进程读文件编号;
    sharefile.startread(number);
    /*读文件*/
    sharefile.endread(number);
}
```

24. 系统有 A、B、C、D 共 4 种资源，在某时刻进程 P_0、P_1、P_2、P_3 和 P_4 对资源的占有和需求情况如下表所示，试解答下列问题。

进程	*Allocation*				*Claim*				*Available*			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	3	2	0	0	4	4	1	6	2	2
P_1	1	0	0	0	2	7	5	[illegible]2				
P_2	1	3	5	4	3	6	10	10				
P_3	0	3	3	2	0	9	8	4				
P_4	0	0	1	4	0	6	6	10				

（1）系统此时处于安全状态吗？

（2）若此时进程 P_1 发出请求 ***request***1(1,2,2,2)，系统能分配资源给它吗？为什么？

（1）系统处于安全状态

存在安全的调用序列：P0→P3→P4→P1→P2

（2）不能分配

若分配后，仍然没有满足P1的最大需求量，而剩下的Available的资源也无法满足任何一个进程的最大需求量，系统处于不安全状态。

29. 进程 $A_1, A_2, \cdots, A_{n1}$ 通过 m 个缓冲区向进程 B_1、B_2、$\cdots$、B_{n2} 不断地发送消息。发送和接收工作符合以下几条规则：(1)每个发送进程每次发送一条消息，写入一个缓冲区，缓冲区的大小与消息长度相等；(2)对于每条消息，$B_1, B_2, \cdots, B_{n2}$ 都需要接收一次，并读入各自的数据区内；(3)当 m 个缓冲区已满时则发送进程等待；当没有消息可读时接收进程等待。试用信号量和 PV 操作编制正确控制消息的发送和接收的程序。

```
semaphore mutex;
semaphore empty[n2];
semaphore full[n2];
mutex = 1;
for(int i = 0; i < n2; i++) {
    empty[i] = m;
    full[i] = 0;
}
main() {
cobegin
    A1();
    A2();
    ...
    An1();
    B1();
    B2();
    ...
    Bn2();
coend
}

procedure send() {                  /*进程Ai发送消息*/
    for(int i = 0; i < n2; i++)
```

```
        p(empty[i]);
    p(mutex);
    /*将消息放入缓冲区*/
    V（mutex）;
    for(int i = 0; i < n2; i++)
        V(full[i]);
}
procedure receive(i) {          /*进程Bi接收消息*/
    p(full[i]);
    p(mutex);
    /*将消息从缓冲区取出*/
    V(mutex);
    V(emtpy[i]);
}
Ai() {                          /*发送进程Ai的描述*/
    while(true) {
        send();
        ...
    }
}
Bi() {                          /*接收进程Bi的描述*/
    while(true)  {
        receive(i);
        ...
    }
}
```