HW3

罗兴攀 PB19051150

1.

requirements of the entry and exit implementation:

- 互斥。没有两个进程可以同时在他们的临界区内执行
- 每个进程都以非0速度运行，但是不应假设进程的相对速度和CPU数量
- 进步。如果没有进程在其临界区内执行，并且有进程需要进入临界区，那么只有那些不在剩余区内执行的进程可以参加选择，以便确定谁能下次进入临界区，并且这种等待不能无线推迟
- 有限等待。从一个进程做出进入临界区的请求知道这个请求允许为止，其它进程允许进入其临界区的次数具有上限。

严格轮转并不符合所有要求，严格轮转不符合要求：进步

2.

Peterson's solution

```
1  int turn; /* who can enter critical section */
2  int interested[2] = {FALSE,FALSE}; /* wants to enter critical section*/
3
4  void enter_region( int process ) { /* process is 0 or 1 */
5  int other; /* number of the other process */
6  other = 1-process; /* other is 1 or 0 */
7  interested[process] = TRUE; /* want to enter critical section */
8  turn = other;
9  while ( turn == other &&interested[other] == TRUE )
10 ; /* busy waiting */
11 }
12
13 void leave_region( int process ) { /* process: who is leaving */
14 interested[process] = FALSE; /* I just left critical region */
15 }
```

- turn只能为0或1，第九行的while(turn==other&&**)使得turn为0与1中的某一值时被阻止（忙等），因此只有一个进程可以进入临界区。满足互斥要求
- 当进程i尝试进入临界区时，如果进程j不想进入，那么interested[j]==FALSE，第九行的while循环不生效，进程i不会因为j而被阻塞，因此满足进步要求
- 假设进程i和j都想进入，而turn只会是其中一个，不妨假设turn=i,则进程i进入临界区，进程j被阻塞等待；进程i结束后，执行第14行的interested[i]=FALSE,于是下一次进程i会被第9行的while循环阻塞，进程j就可以进入临界区了。因此，j在i进入临界区后最多一次就能进入，符合有限等待的要求。

3.

死锁：在多道程序设计中，多个进程可以竞争有限数量的资源。当一个进程申请资源时，如果这时没有可用资源，那么这个进程进入等待状态。有时，如果申请的资源被其它等待进程占有，那么该等待进程有可能再也无法改变状态。这种情况称为死锁。例如，当一组进程内的每个进程都在等待一个事件，而这一个事件只能由这一组进程的另一个进程引起，那么这组进程就处于死锁状态。

死锁发生的条件：

- 互斥：至少有一个资源必须处于非共享模式，即一次只有一个进程可使用。如果另一个进程申请该资源，那么申请进程应等到该资源释放为止。
- 占有并等待：一个进程应占有至少一个资源，并等待另一个资源，而该资源为其它进程所占有
- 非抢占：资源不能被抢占，即资源只能被进程在完成任务后自愿释放。
- 循环等待：有一组等待进程{P0,P1,...,Pn}，P0等待的资源被P1占有，P1等待的资源被P2占有，...,Pn等待的资源被P0占有

4.

表格中的数字不是一个整体，例如1202其实是(1,2,0,2)四个数，为了方便，写为1202

表1	Allocation	Max	Need	finish
T0	1202	4316	3114	0
T1	0112	2424	2312	0
T2	1240	3651	2411	0
T3	1201	2623	1422	0
T4	1001	3112	2111	0

无法找到need<=Available（2223） 故a.Available(2223)这个状态不安全

同理无法找到need<=Available(4411),故状态b也不安全

T0的need<=Available(3014),可进行分配再回收，表1变为表2,work=(3 0 1 4)+(1 2 0 2)=

(4 2 1 6)

表2	Allocation	Max	Need	finish
T0	1202	4316	3114	1
T1	0112	2424	2312	0
T2	1240	3651	2411	0
T3	1201	2623	1422	0
T4	1001	3112	2111	0

然后T4的need<=Available(4216),可进行分配再回收，于是，表2变为表3,

work=(4 2 1 6)+(1 0 0 1)=(5 2 1 7)

表3	Allocation	Max	Need	finish
T0	1202	4316	3114	1
T1	0112	2424	2312	0
T2	1240	3651	2411	0
T3	1201	2623	1422	0
T4	1001	3112	2111	1

找不到need<=work。因此状态c不安全。

检查状态d。work=Available(1 5 2 2).

T3的need<=work. 所以表1变为表4：

表4	Allocation	Max	Need	finish
T0	1202	4316	3114	0
T1	0112	2424	2312	0
T2	1240	3651	2411	0
T3	1201	2623	1422	1
T4	1001	3112	2111	0

work=(1 5 2 2)+(1 2 0 1)=(2 7 2 3)

T1的need<=work,表4变为表5：

表5	Allocation	Max	Need	finish
T0	1202	4316	3114	0
T1	0112	2424	2312	1
T2	1240	3651	2411	0
T3	1201	2623	1422	1
T4	1001	3112	2111	0

work=(2 7 2 3)+(0 1 1 2)=(2 8 3 5)

继续下去：

need[4]<=work --> finish[4]=1-->

work=(2 8 3 5)+(1 0 0 1)=(3 8 3 6)-->

need[0]<=work-->finish[0]=1-->

work=(3 8 3 6)+(1 2 0 2)=(4 10 3 8)-->

need[2]<=work-->finish[2]=1

综上，序列<T3,T1,T4,T0,T2>可是所有finish=1,因此状态d是安全的。

综上，状态a,b,c都不安全，状态d安全，使d安全的序列为<T3,T1,T4,T0,T2>

5.

信号量是一种数据类型，表示资源的状态或数量。分为二进制信号量和计数信号量。信号量可能是一个结构体，这个结构体中一般含有一个整型变量，最基础的信号量可以认为就是一个整型变量，但真正的信号量中具有更复杂的实现，例如可能还包含有一个指针，比如一个指向下一个信号量的指针。

信号量在进程同步中的功能：信号量在进程同步中可以作为一个消息，例如信号量S，初始化为0，表示消息没有产生，而当执行V操作时，S++,S=1表示消息发生，另外一个进程就可以根据这个消息进行同步，使用P操作，如果消息未发生就等待，如果消息发生了就继续执行下面的语句。

```
//进程P1:
/* P1执行的代码 */
signal(S)

//进程P2:
wait(S)
/*P2执行的代码 */
```

如上代码，P2只有在P1完成后才能执行，否则就会被wait(S)阻塞。

6.

哲学家就餐问题的无死锁信号量实现

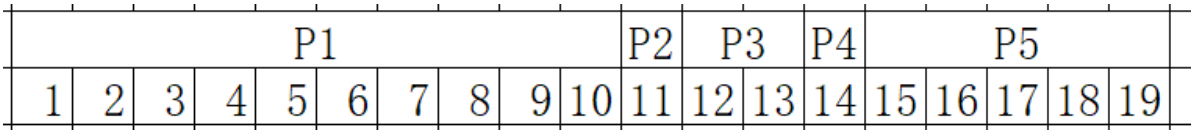
```
//共享变量
#define N 5
#define LEFT ((i+N-1)%N)
#define RIGHT ((i+1)%N)

int state[N]
semaphore mutex=1;
semaphore s[N];

//main function
void philosopher(int i){
    think();
    take(i);    //entry
    eat();  //临界区
    put(i); //exit
}
//Section entry
void take(int i){
    down(&mutex);   //保证后续操作原子性
    state[i]=HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);//如果s[i]被up过了，说明我可以eat(),即可以进入临界区，否则等待
}
//Section exit
void up(int i){
    down(&mutex);
    state[i]=THINKING;
    test(LEFT);     //尝试唤醒左边的人eat
    test(RIGHT);    //尝试唤醒右边的人eat
```

```
31      up(&mutex);
32  }
33  //
34  void test(int i){
35      if(state[i]==HUNGRY && state[LEFT]!=EATING && state[i]!=EATING){
36          state[i]=EATING;
37          up(&s[i]);
38      }
39  }
```

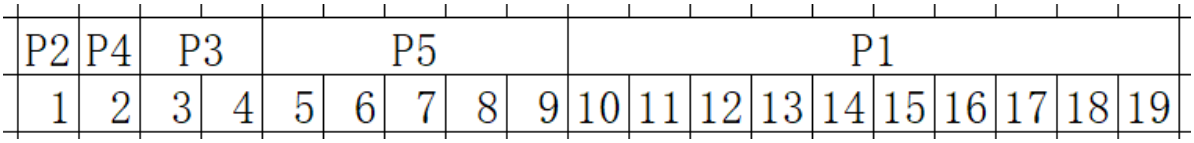
实现代码如上。信号量在其中起到的主要作用是将无法eat()的哲学家通过down(&s[i])进行阻塞，而可以eat()的则可以继续执行。

7.

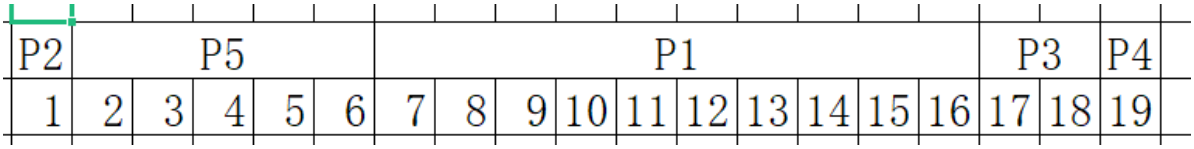
FCFS:先到先服务



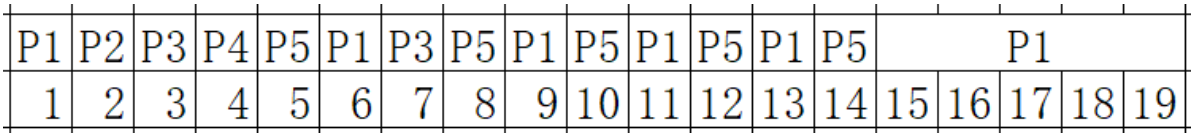
SJF(nonpreemptive)



nonpreemptive priority



RR



四种算法等待时间和周转时间

	burst time	waiting time				turnaround time			
		FCFS	SJF	priority	RR	FCFS	SJF	priority	RR
P1	10	0	9	6	9	10	19	16	19
P2	1	10	0	0	1	11	1	1	2
P3	2	11	2	16	5	13	4	18	7
P4	1	13	1	18	3	14	2	19	4
P5	5	14	4	1	9	19	9	6	14
average		9.6	3.2	8.2	5.4	13.4	7	12	9.2

最小等待时间

SJF算法的平均等待时间最小，为3.2

算法比较

FCFS:优点是公平性较好、实现简单。缺点是对短作业不友好，尤其是短作业前面有较长作业时，会使等待时间很长。

SJF:优点是对于长作业与短作业混合出现的情况，可以极大减小等待时间，周转时间也相比FCFS减少不少。

但该算法对长作业不利，如果有一长作业进入系统的就绪队列，由于调度程序总是优先调度那些(即使是后进来的)短作业，将导致长作业长期不被调度，例如上述例子中的P1要等到其它作业完成才轮到它。该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业会被及时处理。

优先级调度：优点是可以自定义作业优先级，对紧迫性作业友好。缺点是等待时间、周转时间与用户规定的优先级有关，且有可能发生无穷阻塞的问题，即一个较低优先级的作业可能会等待无穷时间。

RR:优点是兼顾长短作业，公平性好，不管什么样的作业都能在可预期的时间内得到响应，用户体验好；缺点是平均等待时间较长，上下文切换需要消耗资源。

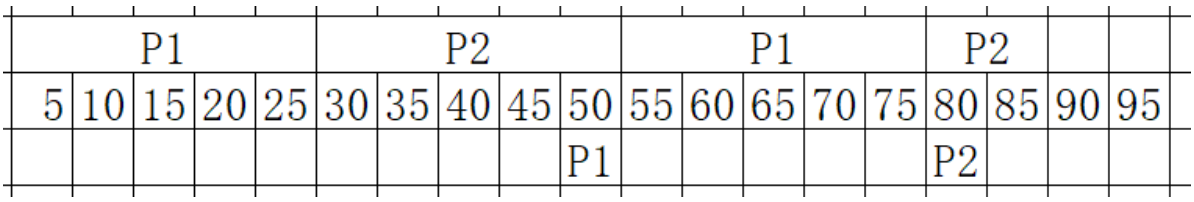
8.

- 单调速率调度
  - 抢占的、静态优先级的调度算法
  - 核心思想是，以进程周期的倒数作为优先级，周期越小，优先级越高。高优先级的进程可以抢占低优先级的进程
- 最早截至期限优先调度
  - 根据截至期限动态分配优先级
  - 截至期限越早，优先级越高；截至期限越晚，优先级越低
  - 当一个进程的周期到来时，它需要向系统公布截至期限，系统根据截至期限来判断是否应该将正在进行的任务进行抢占。

在满足截至期限的方面，单调速率劣于最早截至期限有限调度的例子：

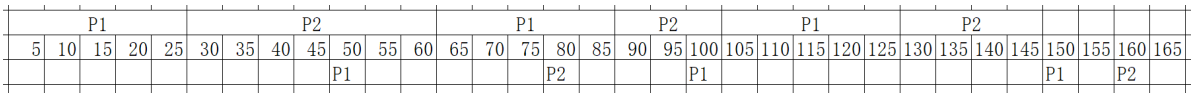
假设有进程P1,周期p1=50,CPU执行时间为T1=25；进程P2,周期p2=80,t2=30。

如果采用单调速率调度，则P1的优先级大于P2，即使P2正在运行，P1也可以抢占P2。因此调度图如下：



如上图所示，P2在80-85仍在运行，超过了截至期限80，因此，对于上面这个例子，单调速率调度无法满足截至期限要求。

但如果采用最早截至期限有限调度，则调度图如下：



如上图，P1和P2始终能在截至期限前完成任务。

综上，上面这个例子中，单调速率调度在满足截至期限要求方面劣于最早截至期限优先调度。