OS HW03

1.

- **互斥**：若pi在其临界区内执行，则其他进程都不能在其临界区内执行；
- **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，且这种选择不能无限推迟；
- **有限等待**：从一个进程做出进入临界区的请求到其被允许为止，其他进程允许进入临界区次数有限；
- 假定每个进程执行速度不为0，对n个进程相对速度不做假设。
- **严格轮转**不能满足进步要求，根据turn的值，临界区外的进程可能可以阻塞其他进程。

2.

- **互斥**：对于pi，pj，只有当flag[j] == false或turn == i时，进程Pi才能进入临界区，而且由于turn只有一个值，即使flag[i] == flag[j] == true，也只有一个进程能够成功执行完while语句，而且只要该进程在临界区内，flag[j] == true和turn == j就同时成立，故满足互斥；
- **进步及有限等待**：首先，只有当flag[j] == true且turn == j成立，Pi才会陷入while循环并被阻止进入临界区；若此时Pj并不准备进入，flag[j] == false，Pi则可以进入。如果Pj也在while中，则谁进入取决于turn的值。然而当Pj退出临界区时，会将flag[j]设置为false以允许Pi进入，而若设置flag[j]为true，turn也会被设置为i。故进程执行while语句时并不改变turn，使其能够进入临界区（**进步**），且PiPj在进入临界区之后最多等待一次就能进入（**有限等待**）。

3.

- 两个或多个进程无限等待一个事件，但这个时间只能由这些等待进程产生。这些进程就成为**死锁**；
- **互斥**：至少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。如果另一进程申请该资源，则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，而被等待的该资源为其他进程所占有；
- **非抢占**：资源不能被抢占，即只能在被进程完成任务后资源释放；
- **循环等待**：有一组等待进程 {P0,P1,……,Pn}，前者等待的资源均被后者占有，Pn等待的资源为P0占有。
- 四个条件同时成立才会出现死锁。

4.

进程	Allocation	Max	Need
	A B C D	A B C D	A B C D
T0	1 2 0 2	4 3 1 6	3 1 1 4
T1	0 1 1 2	2 4 2 4	2 3 1 2
T2	1 2 4 0	3 6 5 1	2 4 1 1
T3	1 2 0 1	2 6 2 3	1 4 2 2
T4	1 0 0 1	3 1 1 2	2 1 1 1

- Available = (2, 2, 2, 3)

    - i = 4, Work = (3, 2, 2, 4)
    - i = 0, Work = (4, 4, 2, 6)
    - i = 1, Work = (4, 5, 3, 8)
    - 此后全部都能满足，即系统出于安全状态。其中一个序列为<4, 0, 1, 2, 3>。
- Available = (4, 4, 1, 1)
    - i = 2, work = (5, 6, 5, 1)
    - 此后全部都能满足，即系统处于安全状态。其中一个序列为<2, 0, 1, 3, 4>。
- Available = (3, 0, 1, 4)
    - 无法找到i使得Need <= Work，故此时系统处于非安全状态。
- Available = (1, 5, 2, 2)
    - i = 3, work = (2, 7, 2, 3)
    - 此后全部都能满足，即系统处于安全状态。其中一个序列为<3, 0, 1, 2, 4>。

5.

- **信号量**是指一个除了初始化只能通过两个标准原子操作wait()和signal()进行访问的整型变量S；
- **（计数）信号量**代表了可用资源数，并能够决定某资源是否可以为一个进程所用。其初值为可用资源数，当进程需要使用资源时，对信号量执行wait()操作以减少计数；当释放资源时，对信号量执行signal()操作以增加计数。当计数为0时所有资源均在使用中，阻塞所有需要使用资源的进程直到计数大于0。

6.

- 核心思想为，只有当一个哲学家两根筷子都可用时，才能拿起筷子。为此，需要区分三种状态：

```
enum {THINKING, HUNGRY, EATING} state[5];
```

- 哲学家i只有在其两个邻居均不在就餐时才能设置变量

```
state[i] = EATING;
(state[(i + 4) % 5] != EATING);
(state[(i + 1) % 5] != EATING);
```

- 除此之外再设置

```
condition self[5];
```

  让哲学家饥饿且不能拿到筷子时可以延迟自己。

- 总体代码

```
monitor DiningPhilosophers
define NUM 5
{
enum {THINKING, HUNGRY, EATING} state[NUM]; // semaphore
condition self[NUM];

void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        self[i].wait();
}
```

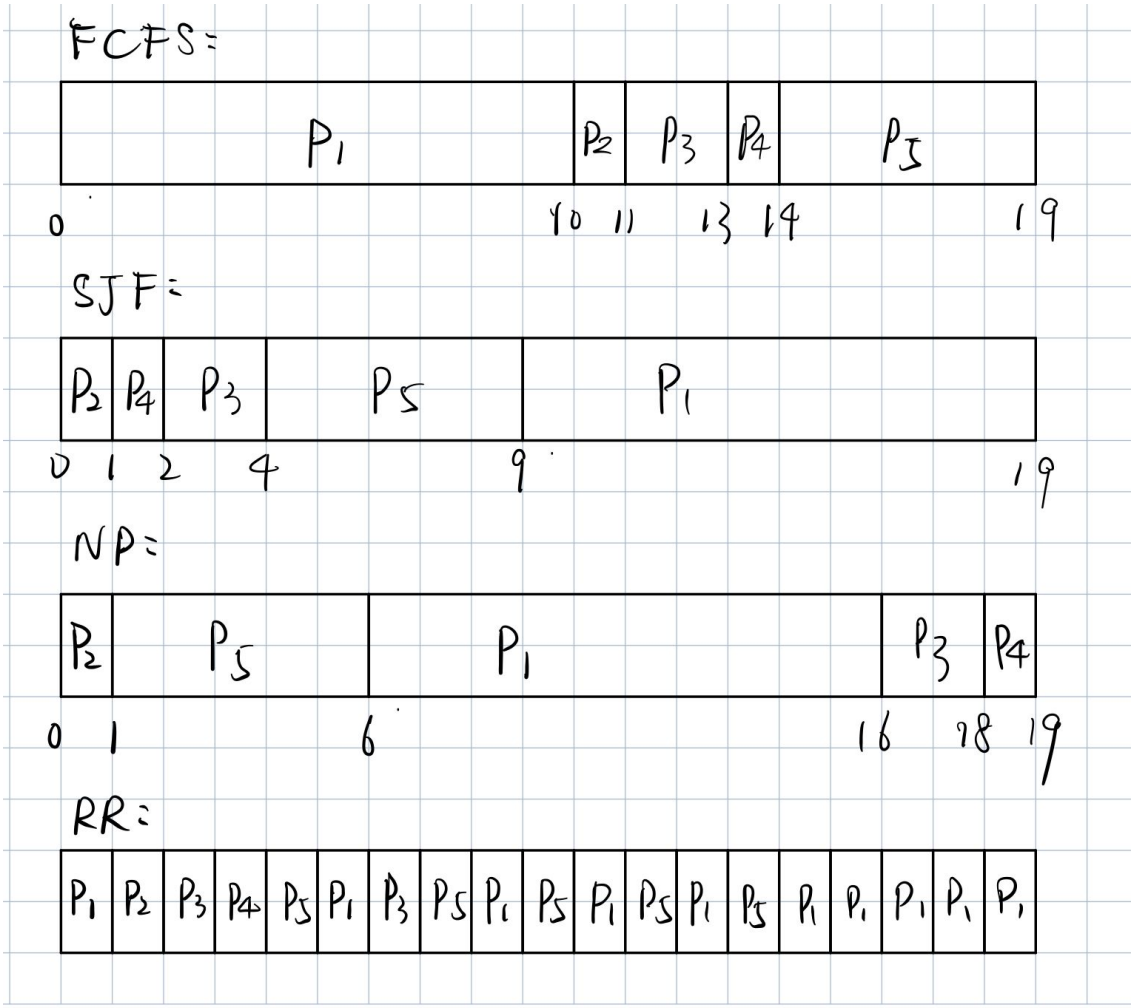
```
void putdown(int i) {
    state[i] = THINKING;
    test((i - 1) % NUM);
    test((i + 1) % NUM);
}

void test(int i) {
    if ((state[(i - 1) % NUM] != EATING) &&
        (state[i] == HUNGRY) &&(state[(i + 1) % NUM] != EATING)) {
        state[i] = EATING;
        self[i].signal();
    }
}

initialization code() {
    for (int i = 0; i < NUM; i++)
        state[i] = THINKING;
}
}
```

7.

1.



2.

进程	FCFS-TA	SJF-TA	NP-TA	RR-TA	FCFS-WT	SJF-WT	NP-WT	RR-WT
P1	10	19	16	19	0	9	6	9
P2	11	1	1	2	10	0	0	1
P3	13	4	18	7	11	2	16	5
P4	14	2	19	4	13	1	18	3
P5	19	9	6	14	14	4	1	9

3. SJF
4.
   - FCFS

     > 优点：最简单且容易实现；
     >
     > 缺点：平均等待时间往往很长；
     >
     > 动态情况下会产生护航效果，导致CPU和设备使用率降低；
     >
     > 非抢占故而对于分时系统非常麻烦。

   - SJF

     > 优点：平均等待时间最短；
     >
     > 缺点：难以实现，需要知道下次CPU执行的长度。

   - NP

     > 优点：能更好的满足更需要刷率先完成某些进程的需求，可以是抢占抑或非抢占；多用于批处理或实时系统；
     >
     > 缺点：容易造成无穷阻塞或饥饿。

   - RR

     > 优点：能够兼顾长短作业，适用于分时系统；
     >
     > 缺点：平均等待时间较长，比较依赖时间片的选取，上下文切换较费时。

8.

- **单调速率**：采用抢占、静态优先的策略，调度周期性任务。当较低优先级进程正在运行且较高优先级可以运行时，较高优先级将抢占低优先级。在进入系统时，每个周期性任务会分配一个优先级，与周期成反比。
- **最早截止期限优先**：根据截止期限动态分配优先级，截止期限越早优先级越高。
- **栗子**：

  假设有P1,P2两个进程：

  ```
  p1 = 50, t1 = 25
  p2 = 80, t2 = 35
  ```

  对于单调速率调度：

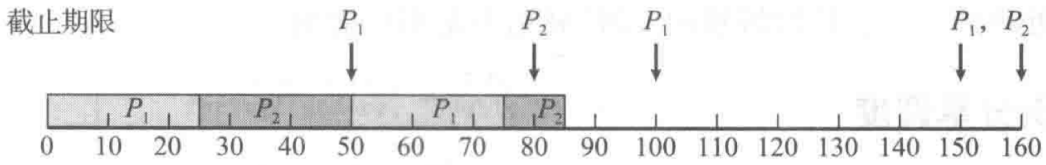


图 5-18　错过截止期限的单调速率调度

对于EDF：

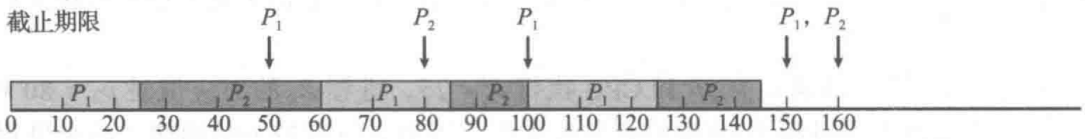


图 5-19　最早截止期限优先调度

显然，单调速率此时会导致错过P2的截止期限。