1.

互斥：如果进程 P1 在其临界区内执行，那么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，而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开始知道这个请求允许为止，其他进程允许进入其临界区的次数具有上限

strict alternation 不满足“进步”，严格交替下，进程 P0 不能连续两次进入临界区，若进程 P0 不想进入临界区，则 P1 被阻塞直到转向。

2.

Peterson's solution

第一条：只有当 flag[j] == false 或 turn == i 时，Pi 才能进入临界区，而且如果两个进程同时在临界区内执行，那么 flag[i] == flag[j] == true，则 Pi、Pj 不可能同时执行它们的 while 语句，因为 turn 的值只可能是 0 或 1，因此只有一个进程 Pi 能成功执行 while 语句，而进程 Pj 应至少再执行一次“turn == j”。而且只要 Pi 在临界区内，“turn == j”和”flag[j] == true”就同时成立，则互斥成立。

第二条：只有条件“turn == i”和”flag[i] == true”同时成立，进程 Pj 才会陷入 while 循环，如果 Pi 不准备进入临界区，则 flag[i] == false，那么 Pj 就可以进入临界区，进步成立。

第三条：如果 Pi 退出临界区，则会执行”flag[i] == false”，以允许 Pj 进入临界区，所以 Pi 在 Pj 进入临界区后最多一次就能进入临界区，满足有限等待。

3.

一个进程集合中的每个进程都在等待只能由该进程集合中的其他进程才能引发的事件，这个进程集合就为死锁。

条件：互斥，持有并等待，非抢占式，循环等待。

4.

| | Allocation | | | | Max | | | | Needed | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |

a.safe T4,T0,T1,T2,T3

b.safe T4,T1,T2,T3,T0

c.unsafe Bk.Needed > 0 (k = 0,1,2,3,4)

d.safe T3,T4,T0,T1,T2

5.

信号量：一个信号量是个整型变量，它除了初始化外只能通过两个标准原子操作：wait()和 signal()来访问。

操作系统常区分计数信号量和二进制信号量，计数信号量的值不受限制，二进制信号量的值只能为 0 或 1。因此二进制信号量类似于互斥锁。计数信号量可以用于控制访问具有多个实例的某种资源。信号量的初始值为可用资源的数量。当进程需要使用资源时，需要对该信号量执行 wait()操作。当进程释放资源时，需要对信号量执行 signal()操作。当信号量计数为 0 时，所有的资源都在使用中，之后需要使用资源的进程将会被阻塞，直到计数器大于 0。

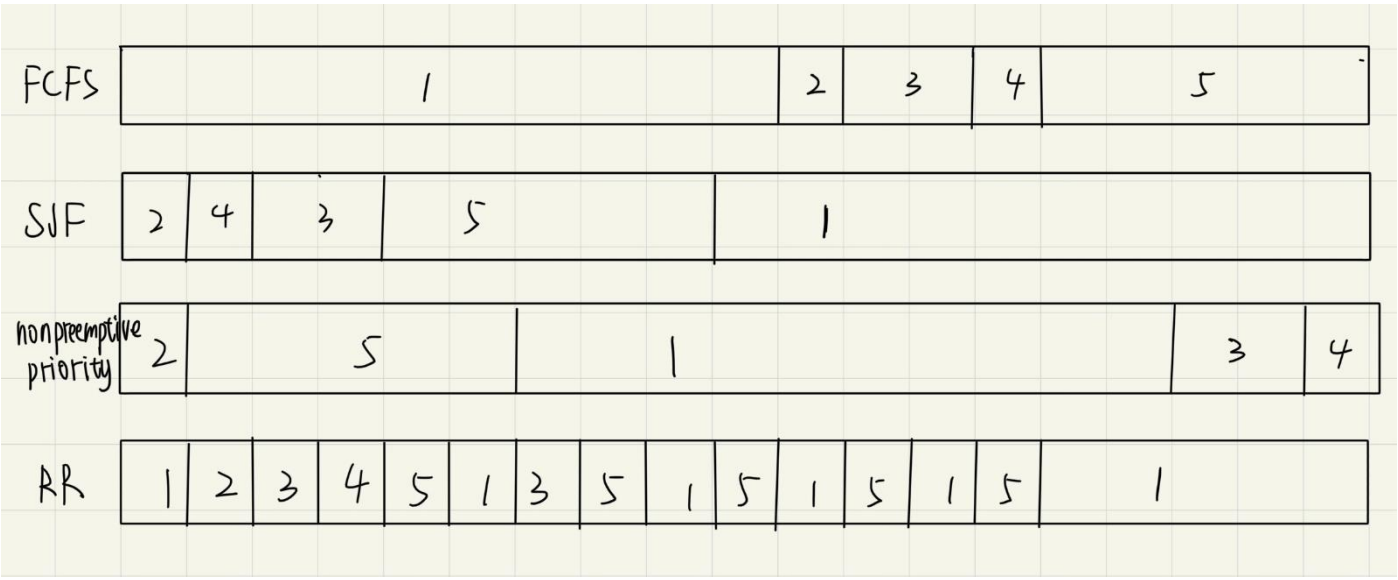
6.

```c
#define N 5
#define LEFT ((i + N - 1) % N)
#define RIGHT ((i + 1) % N)
int state[N];
semaphore mutex = 1;
semaphore s[N];

void take(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
void put(int i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
void test(int i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

7.

a.



b.

| | FCFS | SJF | Nonpreemptive priorit | RR |
|---|---|---|---|---|
| P1 | 10 | 19 | 16 | 19 |
| P2 | 11 | 1 | 1 | 2 |
| P3 | 13 | 4 | 18 | 7 |
| P4 | 14 | 2 | 19 | 4 |
| P5 | 19 | 14 | 6 | 14 |

c.

| | FCFS | SJF | Nonpreemptive priorit | RR |
|---|---|---|---|---|
| P1 | 0 | 9 | 6 | 9 |
| P2 | 10 | 0 | 0 | 1 |
| P3 | 11 | 2 | 16 | 5 |
| P4 | 13 | 1 | 18 | 3 |
| P5 | 14 | 4 | 1 | 9 |

d.SJF

e.

| | FCFS | SJF | Nonpreemptive priorit | RR |
|---|---|---|---|---|
| Pros | 编程简单并且容易理解。 | 平均等待时长最短。 | 可以使重要的进程优先执行。 | 平均响应时间短。 |
| cons | 平均等待时间往往很长。 | 很难获得下次 CPU 执行的长度，不能在短期 CPU 调度级别上加以实现。另外还有可能导致饥饿。 | 容易造成无穷阻塞或饥饿。 | 处理机在进程间频繁的切换，系统开销大。 |

8.

rate_monotonic scheduling:单调速率调度算法采用抢占的，静态优先级的策略。

earliest-deadline-first scheduling:最早截止期限优先调度根据截止期限动态分配优先级。