6.1

- **Mutual exclusion**：通过 `flag` 和 `turn` 变量实现互斥性。如果两个进程都将 `flag` 设为 true，那么只有一个进程能够成功进入临界区，即当前的轮到的进程。等待的进程只能在另一个进程更新 `turn` 值后进入其临界区。
- **Progress**：通过 `flag` 和 `turn` 变量实现进展性。这个算法并不提供严格的轮流进入临界区，而是让进程在需要进入临界区时将自己的 `flag` 设为 true 并进入临界区。进程只在退出临界区时将 `turn` 设为另一个进程的编号。如果该进程希望再次进入临界区——在另一个进程进入之前——则重复这个过程，进入临界区并在退出时将 `turn` 设置为另一个进程的编号。
- **Bounded Waiting**：通过 `turn` 变量保证有限等待。假设两个进程都希望进入各自的临界区，它们都会将自己的 `flag` 设为 true，但只有当前轮到的进程可以进入，另一个进程需要等待。Dekker 的算法通过在进程退出临界区时将 `turn` 设为另一个进程的编号，确保下一个进入临界区的会是另一个进程。

6.11

```c
// shared data
semaphore customers = 0;   // 坐在椅子上的顾客数
semaphore barber = 1;      // 理发师状态
semaphore mutex = 1;       // 控制对waiting的访问
int waiting = 0;           // 店里总顾客数

void barber() {
    while (true) {
        wait(customers);

        givecut();

        signal(barber);
    }
}

void customer() {
    while (true) {
        wait(mutex);
        if(waiting == n + 1) {
            signal(mutex);
            exit();     // leave
        }
        waiting++;
        signal(mutex);
        signal(customers);
        wait(barber);

        receivecut();

        wait(mutex);
        waiting--;
        signal(mutex);
    }
}
```

6.16

monitors的 `signal()` 操作在以下意义上是非持久性的：如果执行 `signal` 操作时没有等待的线程，那么该 `signal` 操作会被忽略，系统不会记录该信号的发生。如果随后执行了 `wait` 操作，则相应的线程会直接阻塞。
而在信号量中，即使没有等待的线程，每次 `signal` 操作都会导致信号量值相应增加。

补充题 1

(1)

```C
Semaphore Cput = Mput = 0, Mget = Pget = 1;

C {
    while (true) {
        wait(Mget);
        放buf1;
        signal(Cput);
    }
}

M {
    while (true) {
        wait(Cput);
        取buf1;
        signal(Mget);

        wait(Pget);
        放buf2;
        signal(Mput);
    }
}

P {
    while (true) {
        wait(Mput);
        取buf2;
        signal(Pget);
    }
}
```

(2)

```C
Semaphore empty1 = m, full1 = 0, empty2 = n, full2 = 0;
Semaphore mutex1 = mutex2 = 0;

C {
    while (true) {
        wait(empty1);
        wait(mutex1);
        放buf1;
        signal(mutex1);
        signal(full1);
    }
}

M {
    while (true) {
        wait(full1);
        wait(mutex1);
        取buf1;
        signal(mutex1);
        signal(empty1);

        wait(empty2);
        wait(mutex2);
        放buf2;
        signal(mutex2);
        signal(full2);
    }
}

P {
    while (true) {
        wait(Mput);
        wait(mutex2);
        取buf2;
        signal(mutex2);
        signal(Pget);
    }
}
```

补充题 2

(1)

`mutex = 1;`：用于互斥生产线的使用

`partA = 0`：生产线上 A 的数量

`partB = 0`：生产线上 B 的数量

`room = 10`：总的可用个数

`roomA = 8`：A 的最大可放置数量

`roomB = 9`：B 的最大可放置数量

(2)

```c
小张:
while(true) {
    生产A;
    wait(roomA);
    wait(room);
    wait(mutex);
    放置A;
    signal(mutex);
    signal(partA);
}

小王:
while(true) {
    生产B;
    wait(roomB);
    wait(room);
    wait(mutex);
    放置B;
    signal(mutex);
    signal(partB);
}

小李:
while(true) {
    wait(partA);
    wait(partB);
    wait(partB);
    wait(mutex);
    取 1A + 2B;
    signal(mutex);
    signal(roomA);
    signal(roomB);
    signal(roomB);
    signal(room);
    signal(room);
    signal(room);
    组成 C;
}
```

补充题 3

(1)

`int count = 0`：当前读者的数量

`Semaphore wrt = 1`：互斥对文件 F 的读写操作

`r_mutex = 1`：互斥对 count 的使用

`if_read = 1`：用于读的时候，处理写的请求

(2)

```c
writer() {
    wait(if_read);
    wait(wrt);
    写文件F;
    signal(wrt);
    signal(if_read);
}

reader() {
    wait(if_read);
    wait(r_mutex);
    count++;
    if (count == 1) wait(wrt);
    signal(r_mutex);
    signal(if_read);
    读文件F;
    wait(r_mutex);
    count--;
    if (count == 0) signal(wrt);
    signal(r_mutex);
}
```