1

系统处于安全状态，意味着存在一个进程序列，使得每个进程在其需要的最大资源数目都能被满足时顺利运行、退出，之后资源被释放给下一个进程。在安全状态下，系统不会发生死锁。

2

在单处理器系统中，关中断可以防止当前代码段被打断，从而实现原子操作。但在多处理器系统中，一个处理器关中断，并不会影响其他处理器：其他 CPU 仍然可以读写共享数据。这样，关中断无法保证多核间的互斥，无法实现真正的同步。

3

3.a

Race condition:

- If two or more processes call `allocate_process()` or `release_process()` concurrently, the increment or decrement operations on `number_of_processes` are not atomic. This can lead to inconsistent or incorrect values.

3.b

Suppose we have a mutex lock called `mutex` with `acquire()` and `release()` methods. Modify the code as following:

```
#define MAX_PROCESSES 255
int number_of_processes = 0;
/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;
    mutex.acquire();
    if (number_of_processes == MAX_PROCESSES) {
        mutex.release();
        return -1;
    }

    /* allocate necessary process resources */
    ++number_of_processes;

    mutex.release();
    return new_pid;
}
/* the implementation of exit() calls this function */
void release_process() {
    mutex.acquire();

    /* release process resources */
    --number_of_processes;

    mutex.release();
}
```

3.c

No. Even the increase and decrease operations are atomic, the `if` statement is not atomic. If two processes check the condition at the same time, they may both find that `number_of_processes` is less than `MAX_PROCESSES`, and both will proceed to allocate resources, leading to an overflow.

4

仍然有可能发生死锁。如果线程 A 持有锁 L1 的读锁，等待锁 L2 的写锁，线程 B 持有锁 L2 的读锁，等待锁 L1 的写锁，这就形成了循环等待，从而产生死锁。

5

5.a

P0 -> P3 -> P1 -> P2 -> P4. In each step find the first process whose need is no more than currently available resources.

5.b

Substract (1, 1, 0, 0) from Available, add it to P1′s Allocation, and recalculate P1′s Need. Then, do the same as in 5.a, and the same sequence still works. So P1′s request can be granted immediately.

5.c

Do the same as in 5.b. P4′s request cannot be granted immediately.

6

```
mutex = new Mutex()            // 保护计数器
north_count = 0                // 桥上北行农民数
south_count = 0                // 桥上南行农民数
north_lock = new Semaphore(1) // 控制北行农民能否进入
south_lock = new Semaphore(1) // 控制南行农民能否进入

// 北行农民过桥
North_Farmer_Wants_To_Cross():
    north_lock.wait()          // 请求北行通行权（防止南北同时进入）
    mutex.lock()
    north_count += 1
    if north_count == 1:       // 第一个北行农民上桥时，锁住南行方向
        south_lock.wait()
    mutex.unlock()
    north_lock.signal()        // 允许其他北行农民请求

    // 过桥
    Cross_Bridge()

    mutex.lock()
    north_count -= 1
    if north_count == 0:       // 最后一个北行农民下桥，释放南行方向
        south_lock.signal()
    mutex.unlock()

// 南行农民过桥
South_Farmer_Wants_To_Cross():
    south_lock.wait()          // 请求南行通行权
    mutex.lock()
    south_count += 1
    if south_count == 1:       // 第一个南行农民上桥时，锁住北行方向
        north_lock.wait()
    mutex.unlock()
    south_lock.signal()        // 允许其他南行农民请求

    // 过桥
```

```
Cross_Bridge()

mutex.lock()
south_count -= 1
if south_count == 0:       // 最后一个南行农民下桥，释放北行方向
    north_lock.signal()
mutex.unlock()
```