Chapter 5

5.4 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

在单处理器系统中，一次只能有一个线程running，自旋锁会导致线程一直空转（忙等待），直到锁被释放，但是持有锁的线程无法运行，锁就无法被释放，直到时间切片结束。

在多处理器系统中，一次可以有多个线程同时运行，持有锁的线程和等待锁的线程可以同时运行，因此在合理的时间内持有锁的进程会释放锁，等待锁的进程可以获取到锁

5.5 Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

如果wait()和signal()没有原子执行，则有可能在同一时间wait()和signal()被同时调用，最终信号量的值无法预测，造成无法预测的结果

5.10 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

系统时钟是通过中断进行正常运行的，如果用户级程序能够禁止中断，则系统时钟将无法正常运行，从而导致上下文切换可能无法正常运行，该程序可以一直占用CPU

5.11 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

因为消息要传递到所有处理器，多处理器的中断禁止十分耗时，效率较低。

5.15 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
    int available;
} lock;
```

(available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the test and set() and compare and swap() instructions:

• void acquire(lock *mutex)

• void release(lock *mutex)

Be sure to include any initialization that may be necessary.

```
// initialization
mutex->available = 0;

// acquire using compare and swap()
void acquire(lock *mutex) {
    while (compare_and_swap(&mutex->available, 0, 1) != 0);
    return;
}

// acquire using test and set()
void acquire(lock *mutex) {
    while (test_and_set(&mutex->available) != 0);
    return;
}
void release(lock *mutex) {
    mutex->available = 0;
    return;
}
```

5.17 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

• The lock is to be held for a short duration.

• The lock is to be held for a long duration.

• A thread may be put to sleep while holding the lock.

1. spinlock
2. Mutex lock
3. Mutex lock

5.18 Assume that a context switch takes *T* time. Suggest an upper bound (in terms of *T*) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

持有自旋锁的上限应为2T，因为上下文切换需要2T，如果大于2T，使用互斥锁使等待线程睡眠更好

5.20 Consider the code example for allocating and releasing processes shown in Figure 5.23.

a. Identify the race condition(s).

b. Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable with an atomic integer?

a. 对于变量number_of_process存在竞争条件

b.在每个函数的开始acquire()锁，在每一个函数的结束 release()锁

c. 不能，因为是竞争发生在allocate_process()函数中，其中number_of_process首先在if语句中测试，然后根据测试值进行更新。测试时，number_of_process=254，但由于竞争条件，在再次增加之前，另一个线程可能将其设置为255。

5.23 Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the test and set() instruction. The solution should exhibit minimal busy waiting.

```
int guard = 0;
int semaphore_value = 0;
void wait(){
    while(test_and_set(&guard) == 1);
    if (semaphore_value = 0){
      atomically add process to a queue of processes waiting for the semaphore
and set guard to 0;
    }
    else
    {
      semaphore_value--;
      guard = 0;
    }
}

void signal(){
    while(test_and_set(&guard) == 1);
    if (semaphore_value = 0 && there is a process on the wait queue){
      wake up the first process in the queue of waiting processes
    }
    else
    {
      semaphore_value++;
    }
    guard = 0;
}
```

5.28 Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

通过偏袒多个读者而不是允许单个作者独家访问共享值，从而提高了读者-作者问题的吞吐量。另一方面，偏袒读者可能会导致作家挨饿。

当读者到达并注意到另一个读者正在访问数据库，那么只有当没有等待的作者时，它才会进入关键部分。

通过保留与等待过程相关的时间戳，可以避免读者/作家问题的饥饿。当作家完成任务时，它会唤醒等待时间最长的进程。