Homework 6

余北辰 519030910245

6.8 Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount)
{
    if (amount > highestBid) highestBid = amount;
}
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

如果多个进程并行时，有两个进程的amount都大于highestBid，从而都成功进入if语句，amount值较大的进程A先修改了highestBid的值，此时highestBid的值等于进程A的amount的值；而立刻进程B又将highestBid的值修改为其amount值。于是最后highestBid的值被修改为了进程B的amount的值，而这个值并非所有进程的最大的amount值，导致错误。

应该将这个if语句及其内部设置为临界区，在进入临界区前加锁，离开临界区后解锁，保证临界区的互斥性，即保证同一时间内只有一个进程能够进入if语句。

6.13 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 , share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 6.18. The other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
while (true) {
    flag[i] = true;

    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
```

```
                ; /* do nothing */
            flag[i] = true;
        }
    }

    /* critical section */

    turn = j;
    flag[i] = false;

    /* remainder section */

}
```

互斥：

如果两个进程同时想要进入临界区，则两个flag[i]都会被对方置为true，而进入while循环；之后就要由turn的值决定该进程进入临界区与否。如果turn为j，那么P_i就不能进入临界区，他将自己的flag[i]置为false，让对方进入临界区，之后自己陷入while循环进行忙等待；而等P_j通过临界区后，会将turn修改并且将自己的flag[j]置为false使得P_i退出忙等待。而P_i在进入临界区前也会将自己的flag[i]置为true来阻止对方进入临界区，从而保证临界区内只能有一个进程。

进步：

如果只有一个进程要进入临界区，他能修改自己的flag来进入临界区；

如果两个进程都欲进入临界区，那么就由turn的值来决定谁先进入临界区；而当进入临界区的进程退出临界区时会修改turn的值，使得对方成功进入临界区，因此进入临界区的选择不会无限推迟。

有限等待：

当一个进程被拒绝进入临界区时，等到另一个进程退出临界区，将turn和flag的值复位后，其一定能进入临界区，因此其等待是有限的。

6.21 A multithreaded web server wishes to keep track of the number of requests it services (known as hits). Consider the two following strategies to prevent a race condition on the variable *hits*. The first strategy is to use a basic mutex lock when updating *hits*:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two is more efficient.

第二种方法更有效。

在使用互斥锁作为同步机制时存在许多的潜在问题，例如优先级反转、死锁等。互斥锁机制也会导致各个进程之间对锁的优先级的竞争非常激烈。而采用第二种方法，即事务性内存系统，也可以保证原子性，而因为不涉及锁，也不可能发生死锁。此外，事务内存系统可以识别哪些原子块内的语句可以并发执行。

（参考《操作系统概念中文翻译版（原书第9版）》 第198页）