作业六

姓名：刘涵之 学号：519021910102

Practice Exercice: 6.8, 6.13, 6.21

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.

- 两个拍卖者都调用这个函数。两者当前的amount都大于highestBid，都进入了if的真分支。然后两者执行顺序不同将导致不同的结果，这就是竞争的情况
- 可以加信号量来解决（S初值=1）
  - ```
    void bid(double amount) {
        wait(S)
        if (amount > highestBid)
            highestBid = amount;
        signal(S)
    }
    ```

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 */
}
```

Figure 6.19 The structure of process P_i in Dekker's algorithm.

- 互斥成立
  - 对于想进入临界区的进程i，如果j不想进入，那么i可以直接进入。如果两个进程在竞争(flag都等于true)，那么根据turn的条件，只有一个能进入临界区，另一个会把自己的flag设为false并等待。
- 进步成立
  - 在i进程在等待j进程执行时候，如果j进程执行完毕，会把turn设为i，让i进程执行，让i进入临界区。
- 有限等待成立
  - i进程在j进程进入临界区后，只需要j一执行完临界区，i就会执行。

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 strategies is more efficient.

原子化（atomic_int）的策略更高效。如果用互斥锁，会导致busy wait，浪费cpu时间。用原子化操作则没有busy wait。