Chapter 6 Homework

陈文迪 519021910071

作业中的引用内容均已标出

6.8 考虑以下的同步问题。

竞争条件在许多计算机系统中都有可能出现。考虑一种在线拍卖系统，其中维护了目前所有商品的最高竞标价格。一位用户可以通过调用 `bid(amount)` 函数来进行出价。该函数会比较用户所出价格和目前的最高值，如果 `amount` 的值超过了最高竞标价格，则最高竞标价格会被设置为新的 `amount` 。`bid(amount)` 函数的代码如下：

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

请描述在这种情况下一个竞争条件是如何发生的，可以做什么来避免竞争条件的发生。

问题解答：

由于 `highestBid` 是一个公共变量，当两个进程同时访问改变量时，会出现竞争条件。例如，当当前的 `highestBid` 为100，A进程调用了 `bid(150)` ，B进程调用 `bid(200)` ，则有可能两进程都执行完比较语句，而B进程先赋值，A进程再接着赋值，这就导致 `highestBid` 保存的并不是目前的最高出价。我们可以通过互斥锁来解决这个，修改后的代码如下：

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

6.13 考虑以下的临界区问题。

第一种已知的关于二进程临界区问题的正确解答是由Dekker提出的。两个线程P_0 和P_1 分享以下变量

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

进程P_i的结构如下

```
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 */
}
```

证明该算法满足临界区问题的全部三个条件。

问题解答：

我们先证明互斥条件成立。事实上，当两个进程都想进入临界区时，他们会将自己的flag设置为true，但由于turn在外层的while语句中并不会被修改，并且turn同一时刻只能取一个值，因此一个进程会被内部的while循环所阻塞并将自己的flag设为false，而turn所指向的进程则可以进入临界区，直到其临界区的进程从临界区退出时，将turn的值和flag的值进行修改，另一进程才可进入临界区。

接着我们证明进步要求。由于一个进程在剩余区内自身的flag会被设置为false并且turn指向另一个进程，因此此时另一个进程不会被无限阻塞，被允许进入临界区。同时，由于每个进程在进入临界区前总会把自身的flag设置为true，而turn总有一个取值，依据互斥条件的分析，总有一个进程是不会被阻塞的。进步条件成立。

最后我们来证明有限等待。有限等待的实现主要在于turn变量。由于一个进程在执行完临界区后，会将turn的值指向另一个进程，这就导致当这两个进程再一次想同时进入临界区时，已经进入过的进程会被阻塞。因此一个进程在另一个进程进入临界区一次后即可进入临界区。

6.21 考虑以下的同步问题。

多线程Web服务器希望跟踪其服务的请求数（称为 `hits`）。考虑以下两种策略，以防止变量 `hits` 出现竞争条件。第一种策略是在更新 `hits` 时使用互斥锁：

```
int hits;
mutex_lock hit_lock;

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

第二种策略是使用一个原子整型变量：

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

请解释哪一种策略更加高效。

问题解答：

第二种策略更高效。第一种策略采用的是互斥锁，这导致了当其他进程进入临界区时，当前进程会循环地调用`acquire()`，造成忙等待，导致CPU周期的浪费。而第二种通过硬件上的原子变量实现则不会有这样的性能损失。