Homework 06

CS307-Operating System (D), Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang)   Student ID: 518030910150   Email: galaxies@sjtu.edu.cn

- (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.

Solution. Here I'll describe one possible race condition as follows.

- Suppose the current highest bid is 100, that is, `highestBid = 100`.
- There are two persons that wishes to bid on the item, and the prices of them are 101 and 102 respectively. So they call the function `bid(101)` and `bid(102)` at the same time.
- Both of them satisfy the condition `amount > highestBid` in `if` instruction, so both of them will execute the instruction `highestBid = amount`.
- If `bid(101)` execute this instruction after `bid(102)`, then the final result of `highestBid` will be modified to 101, which is incorrect since highest bid should be 102.
- Therefore, a race condition happens and causes the wrong result.

Here are some methods to prevent the race condition from occurring.

- We can make the `bid(amount)` function **atomic** in hardware level to forbid executing two `bid(amount)` at the same time, which will prevent the race conditions from occurring.
- We can use **a mutex lock or a semaphore** to prevent the race condition from occurring. Using this method, we can modify the code as follows (here we use a mutex lock).

```
mutex_lock lock;
void bid(double amount) {
    lock.acquire();     /* acquire mutex lock */
    if (amount > highestBid)
        highestBid = amount;
    lock.release();     /* release mutex lock */
}
```

□

- (6.13) The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two process, 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 as follows. 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); /* wait and do nothing */
            flag[i] = true;
        }
    }
        /* critical section */
    turn = j;
    flag[i] = false;
        /* remainder section */
}
```

Proof. Here I'll prove that the algorithm meets all three requirements.

- **(Mutual Exclusion)** Mutual exclusion requirement is met through the variables `turn` and `flag`. Before one process enters the critical section, the `flag` of this process must be `true`, and the `flag` of the other process must be `false`. If two processes come in the same time, then both of them will enter the `while` loop. Since there is only one value of `turn`, only one process can satisfy the condition of `if` instruction and execute the following instructions, which will change the `flag` of current process to `false` and wait. Therefore, the other process can enter the critical section while the current process is waiting. After the other process finishes executing critical section, it will change `turn` and its `flag` so that the current process can stop waiting in the inner `while` loop and start executing critical section. Therefore, there can be one process at most executing the critical section in the same time, which satisfies the requirement of mutual exclusion.
- **(Progress)** Progress requirement is met also through the variables `turn` and `flag`. If a process wants to enter the critical section, then it will set its `flag` to `true`, and if `turn` is also the current process, then it will enter in critical section. After executing critical section, the process will change its `flag` to `false` indicating the latest progress that it has finished executing the critical section, and the process will also change `turn` to let the waiting process (if there is a waiting process) come in and execute the critical section.
- **(Bounded Waiting)** If two processes come in the same time, then both of them will enter the `while` loop. Since there is only one value of `turn`, only one process can satisfy the condition of `if` instruction and execute the following instructions, which will change the `flag` of current process to `false` and wait. Therefore, the other process can enter the critical section while the current process is waiting. After the other process finishes executing critical section, it will change `turn` and its `flag` so that the current process can stop waiting in the inner `while` loop and start executing critical section. Hence, the waiting time of the current process is bounded since the execution speed is non-zero.

□

- (6.21) A multi-threaded 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 rist 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.

  Solution. Atomic integer strategy is more efficient than the basic mutex lock strategy. The reasons are as follows.

  - The atomic integer is implemented in hardware level and it will ensure that only at most one instruction can operate the variable at the same time. Therefore, if many processes want to operate the variable at the same time, there will be a relative order to execute the `atomic_inc(&hits)` to make sure the data won't be operated twice at the same time (i.e., the data is correct). This strategy will **not suspend** other processes while executing instructions of atomic variables, since it is implemented in hardware bus level.
  - The basic mutex lock is implemented in software level and it will also ensure that at most one instruction can enter the critical section, which is between the `acquire()` and `release()` instructions, at the same time. But it will suspend other process while one process is executing the critical section. This will take more time because of the cost of suspending and awakening the process (including context switch time and so on).
  - The mutex lock requires system calls, which will take more time.

  In summary, the atomic integer strategy is more efficient, but the implementation of it will be more complicated since it is in hardware level. □