Homework 5

Gavin Monroe - ComS 352

Q 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:

There is an issue in the code as when bid value is greater than highestBid it should set the amount to highest bid. But in the code, it is happening vise versa. Answer is in the code below.

```
void bid(double amount){
  if (amount gt; highestBid)
     highestBid = amount;        #wrong
}
void bid(double amount){
if (amount gt; highestBid)
    amount=highestBid ;        #correct
}
```

Q 6.13) The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables: boolean flag[2]; /* initially false */ int turn; The structure of process Pi (i == 0 or 1) is shown in Figure 6.18. The other process is Pj (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

The given algorithm satisfies the three conditions of mutual exclusion.

The flag and turn are the variables which are responsible for the mutual exclusion to take its place. If process 1 and process 2 set their flag to true, only one processor will get its resource.

- The resource allocated to the particular process is based upon its turn. The process which is waiting to get its resource will enter into the critical section till the other processor updates its turn value.

The algorithm does not give the strict alteration. If process I wishes to enter into the critical section, the flag variable is set to true.

- When it comes out of the critical section the flag is set to false. Likewise the other process can enter into the critical section by setting its flag variable to true.
- This process repeats and the progress is provided through the flag and tum variables.

Turn variable gives the support to the bounded waiting.

Consider the situation that the processes 1 and the processes 2 are willing to enter into the critical section.

- For this purpose, they both set the flag variable time and enter into the critical section.
- Even both the processes has the variable set into true, it is not possible by both the process to enter into the critical section.
- One process has to wait till other returns. When it completes the section one process leaves way for the other processes to enter into the section.
- If The bounded waiting is not preserved, the waiting process would wait for indefinitely while the other process will enter into the section repeatedly.

Hence, the Dekker's algorithm has a process set the value of tum to the other process, thereby ensuring that the other process will enter its critical section next.

Q 6.16) Consider how to implement a mutex lock using the compare and swap() instruction. Assume that the following structure defining the mutex lock is available: typedef struct { int available; } lock; The value (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 compare and swap() instruction: • void acquire(lock *mutex) • void release(lock *mutex) Be sure to include any initialization that may be necessary

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

void init(lock *mutex){
   // available==0 ->
   //lock is available, available==1 -
   //> lock unavailable
   mutex->available=0;
}
int test_And_Set(int *target){
   int rv = *target;
   *target = true;
   return rv
}
void acquire(lock *mutex){
   while(test_and_set(&mutex->available,1)==1);
}
void release(lock *mutex){
   mutex->available=0;
}
int compare_and_Swap(int *ptr,int expected,int new){
   int actual = *ptr;
   if(actual == expected)
      *ptr = new;
   return actual;
}
void acquire(lock *mutex){
   while(compare_and_swap(&mutex->available,0,1)==1);
```

}
The lock is to be held for a short duration - It makes more sense to use a spinlock as it may in fact be faster than using a mutex lock which requires suspending –and awakening - the waiting process. The lock is to be held for a long duration - a mutex lock is preferable as this allows the other processing core to schedule another process while the locked process waits.
A thread may be put to sleep while holding the lock - a mutex lock is definitely preferable as you wouldn't want the waiting process to be spinning while waiting for the other process to wake up.

Q 6.19) Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a EX-22 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.

Difference between Mutex and Spinlock:

The fundamental difference between spinlock and mutex is that spinlock keeps checking the lock (busy waiting), while mutex puts threads waiting for the lock into sleep (blocked). A busy-waiting thread wastes CPU cycles, while a blocked thread does not.

(a) The lock is to be held for a short duration; the Spin lock is better

Reason: the matter with mutexes is that putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take it slow. If now the mutex was only locked for a awfully short amount of your time, the time spent in putting a thread to sleep and waking it up again might exceed the time the thread has actually slept out and away and it'd even exceed the time the thread would have wasted by constantly polling on a spinlock.

(b) The lock is to be held for a long duration: Mutex is better

Reason: If the lock is held for a longer amount of time, the spin lock will waste a lot more CPU time and it would have been much better if the thread was sleeping instead.

(c) A thread may be put to sleep while holding the lock: Mutex is better

Reason: Thread holding a mutex can definitely be put to sleep and it probably is put to sleep.

If you have got a single-core, then only 1 single thread can run at one moment in time. If you have got 10 threads running on one core then 9 are asleep at anyone time.

Imagine what would happen if a thread with a mutex couldn't be put to sleep, then only that one thread would run until the mutex is released. Your process would essentially become single-threaded whenever you are taking out a mutex whether or not the opposite nine threads don't care about the thing the mutex is protecting.

Q 6.22)Consider the code example for allocating and releasing processes shown in Figure 6.20.
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).

a): No race conditions on the variable number_of_processes.
Explanation: The race condition in the program is A race condition means two or more threads access a shared variable at the same time. Here number of processes is a variable and this variable access in the two functions are allocate process and release process

b) Acquire() call must be placed upon entering each function and release() call should be placed immediately before exiting each function.