HW5

Q1

6.2

- Busy waiting means a practice in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.
- we could choose to make a process wait by waiting the processor, and block on a condition and wait to be awakened at some appropriate time in the future, that means it will not consuming the processor when the waiting period.
- The CPU Utilization rate will be higher, but it make the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

  reference: https://www.baeldung.com/cs/os-busy-waiting

6.3

Spin locks will make all processes outside the critical section always call acquire(), waiting to obtain processor resources, but in a single-processor system, this method will cause a lot of waste to the CPU, but a multi-processor system can make a Processes operate on one processor, and others enter critical sections on other processors.

6.8

- the code

  ```
  highestBid = amount
  ```

  has a race condition,because some processes try to change the variables at the same time.
- we could add a mutex lock to prevent the race condition.

  ```
  Semaphore mutex_lock=1;
  void bid(double mount)
  {
  wait(mutex_lock);
  if(amount>highestBid)
  highestBid=amount;
  signal(mutex_lock);
  }
  ```

6.11

This idiom will work. the process will meet the spinlock when lock is 1, and it will break when the lock is 0. The interrupt will happens after “if(*lock == 0)”,and lock is to 1, “compare_and_swap” will hand the process, the lock will return to 1, so the statement”break” will not be executed.

6.13

- Mutual exclusion

Only with flag[j]=false and turn=i, Pican enters the critical zone. If flag[i] = flag[j] = true, then just one can enter the critical section, because turn cannot be 1 and 0 at the same time. When entering the critical region, flag[i] == true and turns == i, which means that Pj cannot enter the critical region

- Progress

It prevents Pi from entering the critical zone if flag[j] = true and turn = j. If Pj is are not ready to enter the critical zone, flag[j] = false and Pi can enters the critical zone. If Pi and Pj has set the flag to true, it depends on the turn. If turn = i, Pi will enters the critical zone. If turn = j, then Pj will enter the critical zone. However, once Pj exit has its critical zone, it resets flag[j]=false, turn =i, allowing Pi to enter the critical zone. If the Pj resets flag[j] = true, because turn = i, pi enters the critical zone

- Bound waiting

Pi will enter the critical section successfully.

Q2

7.1

Different locking mechanisms are available depending on the needs of the application developer. Spinlock is useful for multiprocessor systems where threads can run in busy loops (for short periods of time) rather than incurring the overhead of being put into the sleep queue. Mutexes are useful for locking resources. More semaphores and condition variables Proper synchronization tools when resources must be held for a long time, because spinlock is inefficient for a long time.

7.3

Need to replace calls to wait (mutual exclusion) and signal (mutex). Therefore, they are calls to the mutex's API.

Q3

8.12

a.

- Mutual exclusion: Only one car at a time on the crossing
- Hold and wait: The car at the intersection blocked the intersection, waiting for other cars to give way
- No preemption: No car can grab the position of other cars
- Circular wait: Every car can’t grab the position of other cars

b. Let cars in one direction go first, if they can’t cross the intersection, don’t enter the intersection.

8.18

a.

Not deadlocked

Order :T2-T3-T1

b.

deadlocked

cycle: T1-R3-T3-R1-T1

c.

Not deadlocked

Order:T2-T1-T3

d.

Deadlocked

Cycle: T1-R2-T3-R1-T1 / R1-T2-R2-T4-R1

e.

Deadlocked

Cycle: R1-T1-R2-T4-R1

f.

Not deadlocked

Order:T2-T4-T1-T3

8.22

Now we can suppose the system is deadlocking, Since there are three processes and four resources. as the question, we know that one process need two resources at least. This process requires no more resource, so it will release the resource after done.

8.23

$$\sum_{i=1}^{n} MAX_i < m + n$$

$$Max_i \geq i \qquad\qquad Max \leq m$$

$$\sum Need_i + \sum Allocation < m + n$$

$$And\ we\ get: \sum Need_i + m < m + n$$

$$\sum_{i=1}^{n} Need_i < n$$

This implies that there exists a process Pi such that Need_i = 0. Since Max_i ≥ 1 it follows that P_i has at least one resource that it can release. Hence the system cannot be in a deadlock state

8.28

a.

T	Need(ABCD)	Available(ABCD)
T0	3 3 3 2	2 2 2 4
T1	2 1 3 0	
T2	0 1 2 0	
T3	2 2 2 2	
T4	3 4 5 4	

Finish	Work
Init	2 2 2 4
T2	4 6 3 7
T3	8 7 4 7
T1	10 8 4 9
T0	13 9 8 10
T4	15 11 10 11

<Init,T2 ,T3,T1,T0,T4> all processes could be finished.

b.

If a request from thread T4 arrives for (2, 2, 2, 4), we get:

T	Need(ABCD)	Available(ABCD)
T0	3 3 3 2	0 0 0 0
T1	2 1 3 0	
T2	0 1 2 0	
T3	2 2 2 2	
T4	1 2 3 0	

Performing a security check finds that all processes cannot be completed, which will cause the system to be in an insecure state, so the request cannot be guaranteed.

c.

If a request from thread T2 arrives for (0, 1, 1, 0),we get:

T	Need(ABCD)	Available(ABCD)
T0	3 3 3 2	2 1 1 4
T1	2 1 3 0	
T2	0 0 1 0	
T3	2 2 2 2	
T4	3 4 5 4	

Finish	Work
Init	2 1 1 4
T2	4 6 3 7
T3	8 7 4 7
T1	10 8 4 9
T0	13 9 8 10
T4	15 11 10 11

<Init,T2 ,T3,T1,T0,T4> all processes could be finished.

Performing a security check finds that all processes can be completed, so the request can be guaranteed.

d.

If a request from thread T3 arrives for (2, 2, 1, 2), we get:

T	Need(ABCD)	Available(ABCD)
T0	3 3 3 2	0 0 1 2
T3	0 0 1 0	6 3 3 4
T1	2 1 3 0	8 4 3 6
T2	0 1 2 0	13 7 9 8
T4	3 4 5 4	15 11 10 11

<Init,T0 ,T3,T2,T1,T4> all processes could be finished.

Performing a security check finds that all processes can be completed, so the request can be guaranteed.