Operating Systems 2 (CS3523) Quiz 2

SURAJ TELUGU

CS20BTECH11050

1.Sol

```
Monitor{
int wait_h,wait_o;   // Number of waiting hydrogen threads and waiting oxygen threads
int re_h,re_o;         // Number of reactive hydrogen threads and reactive oxygen threads
semaphore sem_lock = 1; // Lock semaphore used as mutex lock
 condtional_variable hydrogen, oxygen;
// Since there are conditions to form a water molecule conditional variables are used
Hydrogen() {
    wait_ h++;
    wait(sem_lock);
    while(re_h == 0){
      if(wait_h >=2 && wait_o >= 1) {
      wait_h = wait_h -2;  wait_o--; // Converting Waiting atoms
       re_h = re_h + 2;  re_o++;        // into Reacting atoms
      signal(hydrogen);  signal(oxygen); //Sends Signal that water molecule is formed }
     else {
      signal(sem_lock); wait(hydrogen); wait(sem_lock);
      // Waits for the formation of reactive hydrogens}
   }
    signal(sem_lock);
    re_h--;
}
```

```
Oxygen() {
   wait_o++
   wait(sem_lock);
   while(re_o == 0) {
    if(wait_h >= 2 && wait_o >= 1){
      wait_h = wait_h -2;  wait_o--; // Converting Waiting atoms
       re_h = re_h + 2;  re_o++;        // into Reacting atoms
      signal(hydrogen);  signal(oxygen); //Sends Signal that water molecule is formed
     }
    else {
      signal(sem_lock);  wait(oxygen); wait(sem_lock);
       // Waits for the formation of reactive oxygens
    }
    signal(sem_lock);
    re_o--;
}
}
```

2.Sol

The Mutex lock can be developed by using boolean pointer an input using the following functions Mutex_lock and Mutex_Unlock.

```
void Mutex_Lock(boolean *lock){
  while(true){
    while(LoadLinked(&lock) == 1);
    if(StoreConditional(&lock,1) == 1) return;
  }
}
void Mutex_Unlock(boolean *lock){
   lock = 0;
}
```

3.Sol

The following is the implementation of monitors with signal and continue procedure

```
semaphore mutex = 1;
semaphore x_sem;
In each function:{
 wait(mutex);
 .. body of F...
}
x.wait(){
 x count++;
signal(mutex);
wait(x_sem);
}
x.signal():
if (x count > 0) {
 signal(x_sem); continue;
}
```

4.sol

The following code provides mutual execution, progress and bounded waiting using Compare and Swap instruction which functions atomically.

```
#include <iostream>
#define N 100 // Let number of threads be 100 we can change as we want
int x; // shared variable
int i = 0 , j = 0;
  fetch_xor () {
    while (i<=N) {
    waiting[i] = true;
      key = 1;
      while (waiting[i] && key == 1) key = compare_and_swap(&lock,0,1);
      waiting[i] = false;
      x = x^x; // Performing XOR operation in the critical section
       j = (i + 1) % N;
      while ((j != i) && !waiting[j])
      j = (j + 1) % N;
      if (j == i) lock = 0;
     else waiting[j] = false;
     i++;
  }
}
```

5.Sol

“next” is a binary semaphore used for processes to suspend themselves in signal and wait implementation of monitors and next_count tracks down the number of processes that suspended themselves.

In line 6:

In the body of function, it checks if there whether there is any process which was suspended before and allows it to continue its execution by signal(next) operation.

In line 11:

Before any new process starts its execution using x.wait() it checks if there is any process which is suspended in middle of execution and allows it to resume using signal(next).

In line 19:

If any process sends a signal from signal(x_sem) that particular process suspends its self and waits using wait(next) operation.

6.Sol

The dining philosopher’s problem solution using monitors cannot have deadlocks but processes certainly starve. Let the 5 philosophers (processes) be P1,P2,P3,P4,P5;

Assume P1 intially picks up fork i.e pick_up(1) while it is eating assume all the other process became into hungry state now since it tests only its neighbours P2,P5 may start eating while they are eating again P1 might be set into hungry state and the and P2 or P5 finished their eating P1 starts eating again. In such situation of a tricycle is formed where only P1,P2,P5 have possibility to eat whereas P4,P5 starve indefinitely. Thus, starving cannot be avoided in the solution of dining philosopher’s problem using monitors.