CS3523: Operating Systems - II

Quiz 2

# Soumi Chakraborty

ES19BTECH11017

# Question 1

# **In a system there exist multiple threads of two types: a) hydrogen b) oxygen**

# **Each hydrogen thread waits until it's made into a water molecule to complete execution. In other words, it waits for another hydrogen and one oxygen molecule.**

# **Each oxygen thread waits for two other hydrogens to make a water molecule.**

# **Design procedures for Hydrogen and Oxygen threads that use monitors for synchronization.**

monitor water

{

//shared data

int stored\_H;

int stored\_O;

Condition hyd, ox;

//procedures

process H()

{

//create molecule

if (stored\_H >= 1 && stored\_O >= 1)

{

stored\_H--;

stored\_O--;

hyd.signal();

}

else

{

stored\_H++;

hyd.wait();

}

}

process O()

{

//create molecule

if (stored\_H >= 2)

{

stored\_H -= 2;

ox.signal();

}

else

{

stored\_O++;

ox.wait();

}

}

//initialisations

stored\_H = 0;

stored\_O = 0;

}

# 

# 

# Question 2

**Develop a mutex lock based on the given instructions.**

Building upon the mutex lock functions as described in the textbook. Assuming that these functions are used to update the available variable.

LoadLinked

StoreConditional

available = 1

acquire() {

while (!LoadLinked(available)); // busy wait

//keep looping until available is set to false or 0

while(StoreConditional(available, 0));

}

release() {

//keep looping until available is set to true or 1

while(StoreConditional(available, 1));

}

# 

# Question 3

**Hold and continue using monitors.**

The code and process names for this process has been built by modifying the code from the hold and wait case as given in the textbook.

In the hold and continue case, the signalling process P is allowed to continue executing while the suspended process Q has to wait until P finishes executing or leaves the monitor for some other reason.

Note: next, x\_count are initialised to 0 whereas mutex is initialised to 1.

**Each external function F():**

wait(mutex);

...

body of F

...

if (next\_count > 0)

signal(next);

else

signal(mutex);

**x.wait():**

This function is almost the same as the one in the book except that it won’t do wait(x\_sem) after it finishes signalling.

x\_count++;

if (next\_count > 0)

signal(next);

else

signal(mutex);

x\_count--;

**x.signal():**

if (x\_count > 0) {

next\_count++;

wait(next);

next\_count--;

}

# Question 4

**C++ atomics provides an operation: fetch\_xor. This instruction atomically applies bitwise XOR to the contained value, say x. Can you please provide an implementation of this operation using CAS operation.**

**Some clarification for this question - multiple threads are updating a shared value x. When this operation fetch\_xor is applied on x, this instruction atomically applies a bitwise XOR on x.**

This solution has been implemented by using the left shift operator.

**Solution 1:**

void fetch\_xor(atomic int \*x, atomic int a)

{

int temp;

do {

temp = \*x;

}

while (temp != compare and swap(x, temp, temp^a));

}

**Solution 2:**

void fetch\_xor(atomic int \*x, atomic int \*a)

{

int temp\_x, temp\_a, result = 0, multiplier = 0;

do {

temp\_x = \*x, temp\_a = \*a;

result = result + (pow(10, multiplier++))\*(1 - (temp\_x | 0) \* (temp\_y | 0));

}

while (temp\_x != compare\_and\_swap(x, temp\_x, temp\_x>>1) && temp\_a != compare\_and\_swap(a, temp\_a, temp\_a>>1));

}

# Question 5

**Explain the role of the ‘next’ semaphore.**

In the signal-and-wait implementation, a signalling process has to wait (i.e., suspend themselves) until the resumed process either finishes or leaves for some reason. To achieve this, the ‘next’ semaphore was introduced. The next\_count variable keeps track of how many processes are currently suspended on next.

In line 6, we first check if there are processes that are currently suspended on next. If yes, we release/allow them to execute them using signal() and make the current process wait. Otherwise, the current process can finish executing and call signal(). This successfully implements mutual exclusion.

In line 11, next serves the same purpose as it does in line 6, but for the condition x.

In x.signal(), the x\_count is increased which means there are multiple processes waiting for this. They must thus be suspended and added to the waiting queue, and this is exactly what line 19 is doing.

# Question 6

**In the class, we discussed a solution to the dining philosopher’s problem using monitors. But this solution is susceptible to process starvation. Please explain how?**

This let us consider a case where Philosopher 2 is hungry, but both Philosophers 1 and 3 are busy eating. Thus, since no chopsticks/forks are available, Philosopher 2 starts waiting for the chopsticks to start eating. Now, let Philosopher 3 finish eating. If Philosopher 4 was also hungry and had access to both their chopsticks, then they’d start eating, finish, and put the chopsticks back. Then Philosopher 3 gets hungry and starts eating again. Now when Philosopher 1 also finishes eating, Philosopher 2 still can’t start eating since they only have one chopstick. Thus, though they were hungry before Philosopher 3, they can’t eat and have to wait some more, thereby facing starvation.

# Question 7

**Placement Interviews Problem**

waiting\_count = 0

mutex(0); //1 if interviewer is interviewing otherwise 0

work(0); //wait for interviewer to finish other work

chairs(0); //students waiting on chairs

representative() {

while(true) {

if(waiting\_count == 0)

{

wait(work);

}

else

{

signal(work);

}

}

}

student() {

while(true) {

if(waiting\_count <= N)

{

wait(chairs);

}

wait(mutex);

waiting\_count–-;

signal(mutex);

signal(chairs);

}

}