CISC 3595 Operating Systems

Fall 2022

Assignment #4

Out 11/04; back 11/18

Q1. In the reading, it is mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur? (5pts)

How can such effects be minimized? (5 pts)

Under which conditions should interrupts are ignored? (5 pts)

Q2. What is the meaning of the term *busy wait*? (5 pts) What other kinds of waiting is there in an operating system? (5 pts)

 Explain how busy waiting can be avoided altogether? (5 pts) Under what conditions is it OK to use a busy wait? (5 pts)

Q3. Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated. (5 pts)

Illustrate how a binary semaphore can be used to implement mutual exclusion among n threads. Assume that the threads are running the same code with the same critical section. (5 pts)

Is your implementation in the previous part of this question susceptible to (i) deadlock (ii) starvation or (iii) priority inversion? Explain. (15 pts)

Q4. Explain in detail how the synchronization works using semaphores in the Readers-Writers Problem 1 where Readers are prioritized. Use the names of the synchronization primitives in your explanation. (10 pts)

 Explain in detail how the synchronization changes in the Readers-Writers Problem 2 where Writers are prioritized. What additional primitives are needed? (10 pts) [Hint: look at the points! Give details for semaphores used.]

Q5. Dining Philosophers problem in class uses a monitor. Implement the same solution (simulating monitors) using semaphores in C++ (20 pts). [*Hint: Use a class, mutex and a queue of waiting philosophers who are Hungry but didn’t have access to 2 chopsticks.]*

monitor DiningPhilosophers

{

 enum { THINKING; HUNGRY, EATING) state [5] ;

 condition self [5];

 void pickup (int i) {

 state[i] = HUNGRY;

 test(i); // Tests if chopsticks are available

 if (state[i] != EATING) self[i].wait;

 }

 void putdown (int i) {

 state[i] = THINKING;

 // test left and right neighbors

 test((i + 4) % 5);

 test((i + 1) % 5);

 }

 void test (int i) {

 // both chopsticks must be available

 if ((state[(i + 4) % 5] != EATING) &&

 (state[i] == HUNGRY) &&

 (state[(i + 1) % 5] != EATING) ) {

 state[i] = EATING ; // Gets chopsticks

 self[i].signal () ;

 }

 }

 initialization\_code() {

 for (int i = 0; i < 5; i++)

 state[i] = THINKING;

 }

}