**Comp 4735 Winter 2015**

## Lab Instructor: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ SET : 4D

**Name: Jens Christiansen Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Lab 4

Solve the following exercises. Work in pairs. Discuss each exercise with your lab instructor.

1. Describe, compare and contrast the concepts presented in Table 5.1.

Some key terms related to Concurrency are Atomic operation, Critical Section, Deadlock, Livelock, Mutual Exclusion, Race condition, and Starvation. The fundamental premise that these concepts underpin is the ability to enforce mutual exclusion, which is the ability to exclude all other process from a course of action while one process is granted that ability.

Mutual exclusion is the requirement that one process and one process only has access to a particular access. If not enforced, a race condition might occur, which is a situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

Atomic operations are actions that are indivisible; that is, no other process can interrupt that action while it is being executed. Atomic operations can be enforced by enclosing them around a “critical section”, a section of code within a process that grants mutual exclusion to that process.

If two or more processes are waiting for each other to do something, they will be unable to proceed.

Starvation is a situation where a process is overlooked indefinitely by the scheduler, although it is able to proceed; however, it is never chosen.

1. Discuss the concept of race condition (5.1) and give a code example.

A race condition is a situation where the outcome is determined by the speed in which various threads or processes manipulate shared data. A race condition is a stochastic system in which the designer must implement proper concurrency mechanisms through techniques such as the use of semaphores, mutexes, and monitors. An example of a race condition is where two processes are incrementing a shared variable x = 0. If Process 1 (P1) loads the variable at the same time as Process 2 (P2) loads it, then both processes will have x = 0. When they each increment and then store x back into memory, x will hold the value of 1, as opposed to 2, as we might expect when the two Processes increment x.

|  |  |
| --- | --- |
| Process 1 {  Shared int x;  x++;  printf(“x is %d”, x);  } | Process 2 {  Shared int x;  x++;  printf(“x is %d”, x);  } |

The following time slices may occur, which would result in an unexpected output:

P1: Load x into register; // x = 0

P2: Load x into register; // x = 0

P1: Increment x; // x = 1

P2: Increment x; // x = 1

P1: Store x back into memory; // x = 1

P2: Store x back into memory; // x = 1

P1: printf(“x is %d”, x); // “x is 1”

1. Discuss the concept of mutual exclusion.

Mutual exclusion is a concept where a process or thread has exclusive access to a shared resource. When the process or thread accesses this resource, it “locks” its availability in a “critical section”, and no other process or thread can manipulate that resource. Once the process has finished using the resource, it will “unlock” it for other processes or threads to access.

Requirements for Mutual Exclusion

* Must be enforced
* A process that halts must do so without interfering with other processes
* No deadlocks or starvation
* A process must not be denied access to a critical section when there is no other process using it
* No assumptions are made about relative process speeds or number of processes
* A process remains inside its critical section for a finite time only
* There are ways in which the requirements for mutual exclusion can be satisfied:
  + Hardware approach using special-purpose machine instructions
  + Software approach

1. Solve problem 5.3 in textbook.
   1. Sequence:
      1. P1: x = 10; // 10
      2. P2: x = 10; // 10
      3. P1: while (1)
      4. P2: while (1)
      5. P1: x = x – 1; // 9
      6. P1: x = x + 1; // 10
      7. P2: x = x – 1; // 9
      8. P1: if (x != 10) // true
      9. P2: x = x + 1; // 10
      10. P1: printf(“x is %d”, x); // “x is 10”
   2. Sequence:
      1. P1: x = 10; // 10
      2. P2: x = 10; // 10
      3. P1: while (1)
      4. P2: while (1)
      5. P1: x = x – 1; // 9
      6. P2: LD R0,X // 9
      7. P2: DEC R0 // 8
      8. P1: x = x + 1; // 10
      9. P2: STO R0; // 8
      10. P1: if (x != 10) // true
      11. P1: printf(“x is %d”, x) // “x is 8”
2. What is mutual exclusion hardware support? What are the advantages and disadvantages of this method?

**Interrupt Disabling**

If a resources needs to be locked, the system will create a critical section by disabling all interrupts, effectively preventing other processes from interrupting the current processes. Although this technique guarantees mutual exclusion, the process will not be able to interleave processes. Furthermore, it will only work with a uniprocessor system, and not work in a multiprocessor architecture: More than one process can be running on each processor at the same time that can interrupt each other, breaking the mutual exclusion guarantee.

**Special Machine Instructions**

Special Machine Instructions are mechanisms to exclude access to a memory location when a process is using it. They also allow two atomic actions to be carried out of a single memory location with one instruction fetch cycle. Two of the most commonly implemented instructions are Compare&Swap Instruction and Exchange Instruction.

The advantages of using special machine instructions are:

* It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
* It is simple and therefore easy to verify.
* It can be used to support multiple critical sections; each critical secion can be defined by its own variable.

The disadvantages are:

* Busy waiting is employed: Thus, while a process is waiting for access to a critical section, it continues to consume processor time.
* When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.
* Deadlock is possible.

1. What is a semaphore?

A semaphore is NOT a counter!

* 1. Is the incrementation or decrementation of the counter variable safe?

Yes, the incrementation or decrementation of the counter variable is safe, because the operations are atomic, and cannot be interrupted until they are completed.

* 1. Is the OS aware of the semaphore?

The OS is aware of the semaphore because the semaphore exists in the kernel; they are OS primitive variables. Monitors, on the other hand, exist in the application and are controlled by the application, and cannot be seen by the OS.

* 1. Write the pseudo-code of a semaphore’s methods. Any race condition?

struct semaphore {

int count;

queueType queue;

}

void semWait(semaphore s)

{

s.count--;

if (s.count < 0) {

/\* place this process in s.queue \*/;

/\* block this process \*/;

}

}

void semSignal(semaphore s)

{

s.count++;

if (s.count<= 0) {

/\* remove a process P from s.queue \*/;

/\* place process P on ready list \*/;

}

}

1. Consider a doctor’s office with two doctors each in his room. Consider a waiting room with four chairs. If no chair is available, patients should wait outside. Solve the problem in pseudo-code with semaphores.

I made the following assumptions:

The doctors will be in their rooms in the moment when the office is opened

Doctors do not leave their rooms

Doctors are resources which the clinic utilizes to process patients

The clinic is a process which uses doctors as resources

Patients are resources to be processed

sem\_occupied\_chairs = 2 // the number of patients that the clinic can process with its doctors

sem\_empty\_chairs = 5

mutex[5] mutex\_chairs

----------------------------------------

CLINIC (CONSUMER)

while(true)

wait(sem\_occupied\_chairs)

wait(mutex\_chairs)

see\_patient()

signal(mutex\_chairs)

signal(sem\_empty\_chairs)

dismiss\_patient()

}

----------------------------------------

PATIENT (PRODUCER)

walk\_in() //produce

wait(sem\_empty\_chairs)

wait(mutex\_chairs)

sit()

signal(mutex\_chairs)

signal(sem\_occupied\_chairs)

signal(sem\_occupied\_chair)