COMP 3500 – Introduction to Operating Systems
Homework 1

Group 13:

John Carroll

Bethany Edgars

RayShawn Ware

Individual Submissions:

Bethany Edgar

1.1 P1: blue (each line of code is complete assembly lang execution unless otherwise indicated)
P2: green

```
shared int x;
x = 10;
shared int x;
x = 10;
while (1) {
while (1) {
        x = x – 1;          /*x = 9*/
        x = x – 1;          /*x = 8*/
        x = x + 1;          /*x = 9*/
        if (x != 10)
        x = x + 1;          /*x =10*/
                printf("x is %d", x)   /* prints: x is 10 */
```

1.2 P1: blue (each line of code is complete assembly lang execution unless otherwise indicated)
P2: green

```
shared int x;
x = 10;
shared int x;
x = 10;
while (1) {
while (1) {
        x = x – 1;          /*x = 9*/
        x = x – 1;          /*x = 8*/
        x = x + 1;  interrupted by  x = x + 1;
```

LD RO, x ;x = 8	1st	LD RO, x ;x=8	2nd
INC RO	3rd	INC RO	5th
STO RO, x ;x=9	4th	STO RO, x ;x=9	6th

```
        if (x != 10)
                printf("x is %d", x)  /* prints: x is 9 */
        if (x != 10)
                printf("x is %d", x)  /* prints: x is 9 */
        x = x – 1;          /* x = 8 */
        x = x – 1;          /* x = 7 */
        x = x + 1;          /* x = 8 */
        if (x != 10)
                printf("x is %d", x)  /* prints: x is 8 */
```

2. A binary semaphore has the same implementation and operation as a mutex lock. Thus its possible values are locked or unlocked, this doesn't solve the problem of starvation that is sometimes experienced with a mutex. However, a general semaphore has both a value (to indicate locked or

unlocked) as well as a pointer to the next waiting process so it can form a sort of queue, such that one process will not repetitively use the CPU while all other processes continue to wait.

3. A monitor is another way of dealing with concurrent threads. It is a data type in which all the data from the processes is gathered within a pool. Only one process can be inside the monitor at once to use the data. That way all the data is up to date, if another process needs to access the data in the monitor, it must wait until the first process is done. This is easier to implement than a semaphore but is more restrictive and the signals for another process to enter the monitor can sometimes be lost. (For instance if a process ends and signals for the next process to enter the monitor but there are no processes currently waiting, then this signal can be lost if not handled properly.)

4. wait() and signal() are the two semaphore operations. These operations can be performed in different orders and in different places in a process. However, they cannot be duplicated back to back. If you perform two wait operations on a semaphore within a process, that process will never signal any other processes to start, then those processes waiting will undergo starvation. If two signal operations are performed then multiple processes will try to run at once, resulting in a deadlock.

RayShawn Ware

1.

1.1. P1: x = x – 1 => x = 9
P2: x = x – 1 => x = 8
P1: x = x + 1 => x = 9
P1 into if statement
P2: x = x + 1 => x = 10
P1: “x is 10”

1.2. P1: LD R0, X => x = 10
P1: DEC R0 => x = 9
P2: LD R0, X => x = 9
P2: DEC R0 => x = 8
P1 into if statement
P1: “x is 8”

2. Binary semaphores are easier to implement and are restricted to only having 0 and 1. General semaphores, also called counting semaphores, have a variable k whose value is equal to the number of items in the buffer.

3. A monitor consists of a lock and condition variables. It allows a semaphore to have mutual exclusion and the ability the block due to a condition becoming true. Monitors can also signal other threads that their condition has been met.

4. Two operations can be performed on semaphores: wait and signal. Wait is a locking mechanism and signal/acquire is an unlocking mechanism. When the semaphore is locked, it is not accessible, and when it is unlocked, it is accessible to users.

John Carroll

1.

1.1

P1: x = x – 1 => x = 9

P2: x = x – 1 => x = 8

P1: x = x + 1 => x = 9

P1: if statement

P2: x = x + 1 => x = 10

P1: “x is 10”

1.2

P1: LD R0, X => x = 10

P1: DEC R0 => x = 9

P2: LD R0, X => x = 9

P2: DEC R0 => x = 8

P1: into if statement

P1: “x is 8”

2. Binary semaphore is a semaphore with the integer value ranges over 0 and 1 whereas the counting semaphore's integer value ranges over unrestricted domain. Binary semaphores are easier to implement comparing with the counting semaphore.

Binary semaphore allows only one thread to access the resource at a time. But counting semaphore allows N accesses at a time.

The 2 operations that are defined for binary semaphores are take and release.

The 2 operations that are defined for counting semaphores are wait and signal.

3. A monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met. A monitor consists of a mutex (lock) object and condition variables. A condition variable is basically a container of threads that are waiting on a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

4. A semaphore can only be accessed using the following operations: wait() and signal(). wait() is called when a process wants access to a resource. This would be equivalent to the arriving customer trying to get an open table. If there is an open table, or the semaphore is greater than zero, then he can take that resource and sit at the table. If there is no open table and the semaphore is zero, that process must wait until it becomes available. signal() is called when a process is done using a resource, or when the patron is finished with his meal. The following is an implementation of this counting semaphore (where the value can be greater than 1).

Group Answers:

1. [50 points] Consider the following program:

```
P1: {
        shared int x;
        x = 10;
        while (1) {
                x = x - 1;
                x = x + 1;
                if (x != 10)
                        printf("x is %d",x)
        }
}
```

```
P2: {
        shared int x;
        x = 10;
        while ( 1 ) {
                x = x - 1;
                x = x + 1;
                if (x!=10)
                        printf("x is %d",x)
        }
}
```

Note that the scheduler in a uniprocessor system would implement pseudo parallel execution of these two concurrent processes by interleaving their instructions, without restriction on the order of the interleaving.

1.1. [25 points] Show a sequence (i.e., trace the sequence of interleaving of statements) such that the statement "x is 10" is printed.

Bethany Edgar's Answer

P1: blue (each line of code is complete assembly lang execution unless otherwise indicated)
P2: green

```
    shared int x;
    x = 10;
    shared int x;
    x = 10;
    while (1) {
    while (1) {
            x = x – 1;          /*x = 9*/
            x = x – 1;          /*x = 8*/
            x = x + 1;          /*x = 9*/
            if (x != 10)
            x = x + 1;          /*x =10*/
                    printf("x is %d", x)    /* prints: x is 10 */
```

1.2. [35 points] Show a sequence such that the statement "x is 8" is printed. You should remember that the increment/decrements at the source language level are not done atomically, that is, the assembly language code: LD R0,X /* load R0 from memory location x */ INCR R0 /* increment R0 */ STO R0,X /* store the incremented value back in X */

Bethany Edgar's Answer

P1: blue (each line of code is complete assembly lang execution unless otherwise indicated)
P2: green

```
shared int x;
x = 10;
shared int x;
x = 10;
while (1) {
while (1) {
        x = x – 1;          /*x = 9*/
        x = x – 1;          /*x = 8*/
        x = x + 1;  interrupted by  x = x + 1;
```

LD RO, x ;x = 8	1st	LD RO, x ;x=8	2nd
INC RO	3rd	INC RO	5th
STO RO, x ;x=9	4th	STO RO, x ;x=9	6th

```
        if (x != 10)
                printf("x is %d", x)  /* prints: x is 9 */
        if (x != 10)
                printf("x is %d", x)  /* prints: x is 9 */
        x = x – 1;          /* x = 8 */
        x = x – 1;          /* x = 7 */
        x = x + 1;          /* x = 8 */
        if (x != 10)
                printf("x is %d", x)  /* prints: x is 8 */
```

2. [10 points] What is the difference between binary and general semaphores?
   John Carroll's Answer

   Binary semaphore is a semaphore with the integer value ranges over 0 and 1 whereas the counting semaphore's integer value ranges over unrestricted domain. Binary semaphores are easier to implement comparing with the counting semaphore.

   Binary semaphore allows only one thread to access the resource at a time. But counting semaphore allows N accesses at a time.

   The 2 operations that are defined for binary semaphores are take and release.

   The 2 operations that are defined for counting semaphores are wait and signal.

3. [10 points] What is a monitor?
   Bethany Edgar's Answer
   A monitor is another way of dealing with concurrent threads. It is a data type in which all the data from the processes is gathered within a pool. Only one process can be inside the monitor at once to use the data. That way all the data is up to date, if another process needs to access the data in the monitor, it must wait until the first process is done. This is easier to implement than a semaphore

but is more restrictive and the signals for another process to enter the monitor can sometimes be lost. (For instance if a process ends and signals for the next process to enter the monitor but there are no processes currently waiting, then this signal can be lost if not handled properly.)

4. [20 points] What operations can be performed on a semaphore?
   Bethany Edgar's Answer
   A semaphore can only be accessed using the following operations: wait() and signal() are the two semaphore operations. These operations can be performed in different orders and in different places in a process. However, they cannot be duplicated back to back. If you perform two wait operations on a semaphore within a process, that process will never signal any other processes to start, then those processes waiting will undergo starvation. If two signal operations are performed then multiple processes will try to run at once, resulting in a deadlock.