COMP 3500

Homework 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. 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.