Theory Assignment 2

By

Keyur Patel

40154883

A report submitted in partial fulfilment of the requirements of COMP346

Concordia University

March, 2022

**Answer 1**:

User Threads:

Require no context switching, which makes them the faster model in terms of switching. Since they are only made by system calls, the OS cannot see them , which would lead to poor scheduling done by the CPU

Kernel Threads:

Kernel threads require context switching, which would mean that it would take more time in terms of switching. Since the kernel oversees the scheduling, the CPU can allocate enough memory for all the processes to run smoothly.

The only model that can trash the system without any constraints would be the one-to-one relationship model. It will create many kernel threads which would make the CPU unable to allocate memory properly and thus crashing the system.

**Answer 2:**

Threads are "lightweight" because they can communicate with one another without requiring inter-process communication. It's cheaper to switch between threads than it is to switch between processes (just moving some pointers around). Inter-process communication is also more costly than thread communication.

Because a thread is a component of the process, it does not require any additional resources when it is formed; instead, it shares the memory space of the process from which it was generated. Whereas, creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open ﬁles, and environment variables. Therefore, thread can use all the resources of a process it is part of.

**Answer 3:**

Shared memory provides a faster interaction between user processes because a shared memory space allows for faster communication from the OS. Shared memory should not be used for inter-processed communications since it might cause synchronization problems, hence making it unsuitable.

**Answer 4:**

a)

i) If we start with process B, it will signal the mutex, goes to process A or C which then will block it completely because it will either keep on doing go B or go C. The same result can be achieved by starting with process C.

ii) By starting at process A, we will go to C, which will lead for A to finish, However, since C will be blocked by waiting for go C, this will also lead for B to wait for the mutex to be released. Hence this will block B and C.

iii) To ensure this, we have to start with A, which then goes to B, which then goes to C.

b)

i) m>n just means that process A will run more than process B. This allows the process to create more permits for the semaphore go B, and since it will always create permits for go B, these processes will never be blocked permanently.

ii) Since process A is not waiting on anything, there is no way it will be blocked permanently, However, once it reaches process B, it will block completely because it will keep on waiting on go B constantly.

**Answer 6:**

1. S1 = 0, S2 = 0, S3 = 0

Process P1{ Process P2{ Process P3{

<phase I> <phase I> <phase I>

V(S1) V(S2) V(S3)

P(S2) P(S3) P(S1)

<phase II> <phase II> <phase II>

} } }

1. Process P1{ Process P2{ Process P3{

<phase I> <phase I> <phase I>

V(S1) V(S2) V(S3)

P(S2) P(S3) P(S1)

<phase II> <phase II> <phase II>

V(S1) P(S2) P(S3)

V(S2) V(S3) P(S1)

} } }

**Answer 7:**

We will irregularities with the threads if we do not maintain the operation as atomic. Ex:

Let s(1) = 3  
Let p1 executes signal(), but suddenly it gets switched to wait() by p2. By the time we reach to completing signal() for p1, s(1) =3 will be rendered false.

**Answer 8:**

In multiprogramming, there is a lot of overflows of the connected hardware. If two people try to connect to a different device with their Bluetooth devices but both hardware are capable to be paired with each separate devices. With Bluetooth speakers to a computer, multiprogramming may cause unnecessary data exchange from connecting to the wrong computer and hence creating unnecessary problems.