**Theory Assignment 2- Édouard Gagné**

**Question # 1**

What are the main differences between the user-kernel threads models? Which one of these models is likely to trash the system I fused without any constraints.

The many to one model assign many user threads to one kernel thread, and thread management is done in user space. The one-to-one model assign each user thread to a kernel thread, creating a new kernel thread when a user thread is created. The many-to-many multiplexes many user threads to a smaller or equal number of kernel threads. One-to-one threads can trash the system when the user creates a large number of thread, since the system is forced to create the same number of kernel threads.

**Question # 2**

Why threads are referred to as “light-weight” processes? What resources are used when a thread is created? How do they differ from those used when a process is created?

A thread is lightweight compared to processes because their memory is shared between their sibling threads. The same resources are used in the process and thread creations, but the memory is shared between multiple thread instead of each process having their own address space.

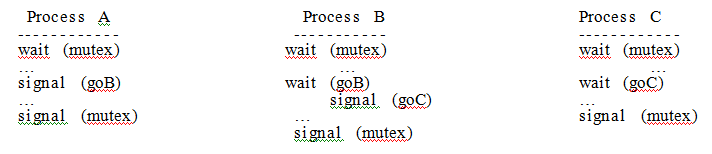
**Question # 3**

Does shared memory provide faster or slower interactions between user processes? Under what conditions is shared memory not suitable at all for inter-process communications?

Shared memory provides faster context switching between user processes that share the same memory (since they share the same address spaces and the same cache), so interactions between threads are much faster than between user processes. If two processes from different machines need to communicate, then threads are not appropriate, since they need toshare the same memory on a single machine.

**Question # 4**

a) Consider three concurrent processes A, B, and C, synchronized by three semaphores mutex, goB, and goC, which are initialized to 1, 0 and 0 respectively:



Does there exist an execution scenario in which:

(i) All three processes block permanently?

If process C goes first until it waits for goC then the other processes are stuck waiting for mutex as well. (Also works if process B goes

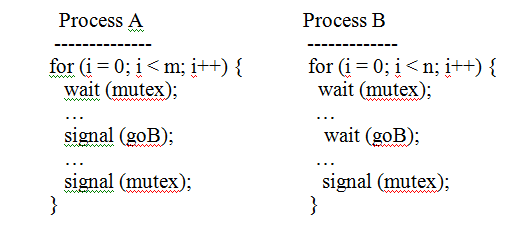
(ii) Precisely two processes block permanently?

If process A goes first and completes then process C executes and waits for goC, then process B is stuck waiting for mutex as well.

(iii) No process blocks permanently? Justify your answers.

If process A goes and complete and then process B goes completely and then process C goes completely, there are no process block.

b) Now consider a slightly modified example involving two processes:



(i) Let m > n. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

No matter the values of m or n, if process B goes first and wait for goB then process A is also stuck waiting for mutex. If m =2 and n =1, process A can go, then B, then A without problems.

(ii) Now, let m < n. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

Both processes can block if B goes before A. There is no scenario where no process block, since there will always be at least an extra process B that need a process A to complete.

**Question # 5**

In a swapping/relocation system, the values assigned to the <base, limit> register pair prevent one user process from writing into the address space of another user process. However, these assignment operations are themselves privileged instructions that can only be executed in kernel mode.

Is it conceivable that some operating-system processes might have the entire main memory as their address space? If this is possible, is it necessarily a bad thing? Explain.

In the described system it is entirely possible that the entire main memory is used for only one process. However, the CPU will only be able to execute that process only since it only has access to the main memory, while the other processes data wait in the secondary memory. In this case, this is fairly inefficient since only the process in the main memory will execute.

**Question # 6**

Sometimes it is necessary to synchronize two or more processes so that all process must finish their first phase before any of them is allowed to start its second phase. For two processes, we might write:

semaphore s1 = 0, s2 = 0;

process P1 { <phase I>

V (s1)

P (s2) <phase II>

process P2 { <phase I> V (s2)

P (s1)

<phase II> }}

a) Give a solution to the problem for three processes P1, P2, and P3.

semaphore s1 = 0, s2 = 0, s3=0;

process P1 {

<phase I>

V (s1)

P (s3)

P (s2)

<phase II>

V (s3)

V(s2)

}

process P2 {

<phase I>

V (s2)

P (s1)

P (s3)

<phase II>

V (s3)

V(s1)

}

process P2 {

<phase I>

V (s3)

P (s2)

P (s1)

<phase II>

V(s2)

V(s1)

}

b) Give the solution if the following rule is added: after all processes finish their first phase,

phase I, they must execute phase II in order of their number; that is P1, then P2 and finally P3.

semaphore s1 = 0, s2 = 0, s3=0, s21=0, s22=0

process P1 {

<phase I>

V (s1)

P (s3)

P (s2)

<phase II>

V(21)

V (s3)

V(s2)

}

process P2 {

<phase I>

V (s2)

P (s1)

P (s3)

P(s21)

<phase II>

V(s22)

V (s3)

V(s1)

}

process P2 {

<phase I>

V (s3)

P (s2)

P (s1)

P(s22)

<phase II>

V(s2)

V(s1)

}

**Question # 7**

Generally, both P and V operation must be implemented as a critical section. Are there any cases when any of these two operations can safely be implemented as a non-critical section? If yes, demonstrate through an example when/how this can be done without creating any violations. If no, explain why these operations must always be implemented as critical sections.

The P and V operations must be atomic so that they can fulfill their role and provide mutual exclusion. A simple example would be that two thread could access the P operation when the semaphore is equal to 1, so they would both be able to go through and not wait, which does not provide mutual exclusion. The same thing could happen with V with two threads signaling V twice, letting two other threads in a critical section. Note that V and P should not be implemented using interrupts, but by using the test and set operation.

**Question # 8**

What is the potential problem of multiprogramming?

The main potential problem with multiprogramming is security. Since in multiprogramming resources are shared between different process, a malicious process could be able to abuse the resources and hinder the other processes.