Noemi Lemonnier
#40001085

Assignment 1

Question 1

In general, a process and a thread are independent sequences of execution. What differentiates them is how they are being processed. A process can have multiple threads, while a program has at least one main thread. There main differences are:

Process	Thread
Process is heavy weight/resource intensive	thread is light weight, taking lesser resources than a process
Process switching needs interaction with OS	Thread switching doesn't need to interact OS (it's because you don't have to switch the memory
In multiple process environments, each process executes same code but has its own memory and file recourses	all threads can share same set of open files, child processes
if 1 process if blocked then no other process can execute until 1st process is unblocked	If 1 thread is blocked and waiting, another one will keep running.
multiple processes without using threads use more resources	multiple threads processes use fewer resources
in multiple processes, each operates independently of the others.	one thread can read, write, or change another thread data.

Question 2

To create a thread you can either in Java extend Thread class or implement Runnable interface, which is recommended to use. A thread is light-weighted as said earlier which means it requires less resource than a process creation.
For the process creation, it requires a Process Control Block (PCB) and each process has its own memory space. This makes the process quite heavy-weighted, resource intensive.
For the creation of a thread, either kernel or user thread, it will require a data structure to regroup: a register, a stack and a priority system, because threads need to know which one goes first or it can lead to errors.

Question 3

First of all, the thread management kernel is not aware of the existence of user level threads but it does manage the kernel level threads. Kernel level threads are generally slower to create and manage than user threads.
Also, user level threads are always associated to a process while kernel threads are not.
Kernel level threads will be more expensive because they come with a kernel data structure, which is expensive to maintain compared to the user level threads and their processes.

Noemi Lemonnier
#40001085

Question 4

Ai) All three process block permanently: If the process starts with Process C or with Process B.
Process C will wait mutex, since mutex is =1, it goes to wait goC =0, so goC waits. Context switch to Process B that has goC. it wait(mutex) but mutex is now at =0 so it will wait and create a deadlock.

Process B will wait mutex, since mutex = 1, it goes to wait goB. Context switch to Process A but it will waits for mutex which is now = 0. No signal for mutex. Deadlock.

Aii) Precisely two processes block permanently
If you start with process A, it will wait mutex, since mutex is =1, it goes to signal goB =1 and signal mutex =1. It goes to Process C, it will wait for mutex, it goes through and then mutex = 0. Then it will wait for goC=0. It context switch to Process B and it waits for mutex = 0. Both process C and B are blocked.

Aiii) No process blocks permanently
If you start with process A, it will wait mutex, since mutex is =1, it goes to signal goB =1 and signal mutex =1. It goes to Process B, it waits for mutex goes through and then mutex = 0 then wait goB goes through and then goB = 0, then it signal goC =1, and signal mutex = 1. It goes to Process C, it will wait for mutex, it goes through and then mutex = 0, and it goes to wait goC but go=1, so it goes through and then goC=0, and it signal mutex. So all Processes go through.
(Although it does not respect first requirement of barrier synchronization)

Bi) There is a scenario if m>n where blocks are permanently blocked:
If m =2, n=1. mutex =1, goB = 0, goC =0
If you start with process B, you wait for mutex, but it goes through and mutex=0. It waits for goB but goB =0.

There is a scenario that blocks are not block:
If m =2, n=1. mutex =1, goB = 0, goC =0
If you start with process A: you wait for mutex, but it goes through and mutex=0. It signal goB, now goB=1, and signal mutex, now mutex=1. It does this again: it goes through and mutex=0. It signal goB, now goB=2, and signal mutex, now mutex=1.
Then it goes to process B: you wait for mutex, but it goes through and mutex=0. It waits for goB, but goB=2, so it goes through, now goB=1, and it signals mutex =1.

Bii) There is a scenario if m>n where blocks are permanently blocked:
If m =1, n=2. mutex =1, goB = 0, goC =0
If you start with process A: you wait for mutex, but it goes through and mutex=0. It signal goB, now goB=1, and signal mutex, now mutex=1.
Then it goes to process B: you wait for mutex, but it goes through and mutex=0. It waits for goB, goB=1, so it goes through, now goB=0, and it signals mutex =1. It does

this again: it wait for mutex, it goes through and mutex=0. It waits goB, but goB=0 will wait for goB(). Even if it Context switch to Process A, process A cannot continue as it wait for mutex but mutex=0.

There is no scenario where both blocks are executed because even if we start with process B, in Process B wait(goB) is =0. It will not be signalled and process A will stay stuck because of wait(mutex). Deadlock.

Question 5

One potential problem with multiprogramming is that one can steal or copy any user's file, one can overwrite someone's program. This is a big security issue. That is why only the kernel should be able to do this and not from a user level.

Question 6

A)

```
semaphore 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 (s1)                 P (s1)
P (s3)                 P (s3)                 P (s2)
<phase II>             <phase II>             <phase II>
}                      }                      }
```

B)

```
semaphore s1 =0, s2=0, s3=0
process P1 {           process P2 {           process P3 {
<phase I>              <phase I>              <phase I>
P (s1)                 V (s1)                 V (s1)
P (s1)                 P (s2)                 P (s3)
<phase II>             <phase II>             <phase II>
V (s2)                 V (s3)                 }
}                      }
```

Noemi Lemonnier
#40001085

Question 7

Critical sections appear when two threads try to access a common shared variable. If they are not controlled, they could lead to erroneous results. By having P() and V(), we can monitor and make sure two threads are not entering the critical section at the same time.

For example : A cup of coffee; A "pourer"(producer); A"drinker"(consumer)

pourer:
while(true){
pour();
}
	Drinkers:
	while(true){
	drink()
	}

Solution: using P and V	

Main{
Semaphore n;
n = new Semaphore(1);
}
	Drinker i:
	
	while(true){
	P(cup)
	drink()
	V(cup)
	}

Question 8

No it is not conceivable that some operating-system processes might have the entire main memory as their address space.

First of all, because the OS kernel needs to stay in the memory at all times. Also, no process can fulfil all memory, so occupy all possible addresses and unoccupied physical memory is allocated to virtual memory manager.
By allocating it to the virtual memory manager, the processes will have addresses inside the virtual space but some of that space is used by the system.

That explained, it is impossible that the OS kernel and its processes can use the entire virtual memory and the entire physical memory.