Mustafa Al Awadi

ID: 40217764

Comp346

2024-10-28

Theory Assignment 2

1. The biggest difference between the user-kernel thread models is how the user threads are mapped to kernel threads. In many-to-one, many user threads are mapped to one kernel level thread. In one-to-one, each user level thread is mapped to a single kernel thread, one blocking causes all to block. In many-to-many, many user level threads are mapped to many (could be lesser) kernel level threads.

   The many-to-one model is likely to trash the system if used without constraints as it only takes one user level thread to get blocked in order to affect the single kernel level thread as they are mapped and because there's only one kernel level thread, there is a higher possibility for the entire system to crash.

2. They are referred to as light weight since they share the memory with the process that creates and because they share the memory used by the parent process and other threads. Because they share the same memory, communication via inter-process is not required therefore we have freed some resources. When thread is created, only registers, PC, and stack are created where the code is shared. A process, however, can create new PC, stack and registers with a new code and data.

3. A. Shared memory provides faster communication because the kernel is only utilized once when initializing the shared memory, but in message passing the kernel is used whenever a message needs to be passed. Therefore, shared memory does not depend too much on kernel which is what makes it quicker. The shared memory is not suitable at all for inter-process communications when two processes don't share the same physical memory.

   B. Message passing is usually the model that is likely to be a lot slower because it involves copying messages between user thread and kernel thread which introduces overhead, and also because in message passing a process may have to wait for a message to arrive, which then could cause blocking and increased response times.

4. a)

   I. If we start with process B, it will take the mutex and will make the value equal to 0, after that the process will be stuck at wait (goB), which could be signaled by process A, however, it is also stuck at wait (mutex). Likewise, process C is also stuck at wait (mutex). Therefore, all three processes are blocked permanently.

   II. If we start with process A, it will take the mutex then signal (goB) to signal (mutex). Process C will start and have the mutex to turn the value to 0 and get stuck at wait (goC), and this can be signaled by process B but B is also stuck at wait (mutex) because it is taken by process C. Therefore, processes B and C are blocked permanently.

   III. If we start with process A, it will take the mutex, then signal (goB) which then signal (mutex). When process B begins, it takes the mutex, reach wait (goB), then

signal (goC) and also signal (mutex). When process C starts, it takes the mutex, crosses wait (goC) and signal (mutex). Thus, no processes are blocked permanently.

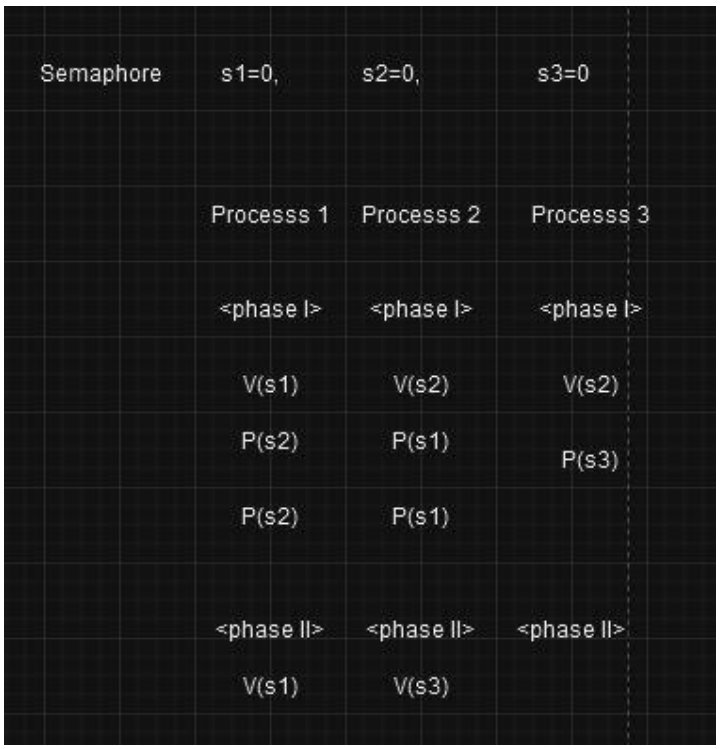
b)

I. If process B starts first, it grabs the mutex but gets stuck at wait (goB), needing a signal from process A. Meanwhile, process A also gets stuck at wait (mutex), causing both processes to be permanently blocked. If process A starts and finishes the loop first, goB will have a value of m. Process B can then start and finish without getting stuck, as m > n and no processes get permanently blocked.

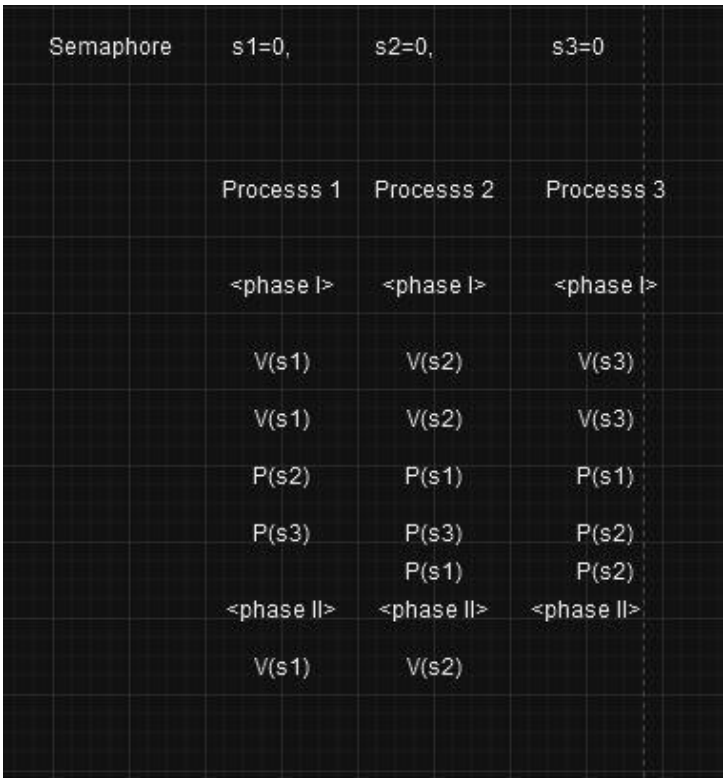
II. If process B starts first, it will grab the mutex and get stuck at wait (goB), which must be signaled by process A. However, process A gets stuck at wait (mutex), which must be signaled by process B, and since B is stuck, it won't be able to do that. Therefore, both processes get permanently blocked. There's no scenario where neither of the processes will be permanently blocked. Since process B calls wait (goB) n times, goB must be signaled at least n times. However, it will only be signaled m times by process A, and here m < n. Hence, in all possible scenarios, process B will be blocked permanently.

5. It is impossible to assign a process the entire main memory because some memory is needed for storage that is reserved for the operating system in the main memory. A kernel always exists in the memory meaning that a single memory can never have the size of the whole address.

6. a)


Semaphore s1=0, s2=0, s3=0
Processs 1 Processs 2 Processs 3
<phase I> <phase I> <phase I>
V(s1) V(s2) V(s2)
P(s2) P(s1) P(s3)
P(s2) P(s1)
<phase II> <phase II> <phase II>
V(s1) V(s3)

b)


Semaphore s1=0, s2=0, s3=0
Processs 1 Processs 2 Processs 3
<phase I> <phase I> <phase I>
V(s1) V(s2) V(s3)
V(s1) V(s2) V(s3)
P(s2) P(s1) P(s1)
P(s3) P(s3) P(s2)
P(s1) P(s2)
<phase II> <phase II> <phase II>
V(s1) V(s2)

7. No, they have to always be implemented as critical sections to ensure proper synchronization and avoid race conditions.

8. There are mainly two problems with multiprogramming, process synchronization and debugging. If we have two or more processes needing to access a shared resource, they need to be synchronized to prevent unexpected behavior. Because processes are executed in a difficult pattern, it is hard to trace fix the errors to solve the bug.