CmpE 322

January 29, 2022

Contents

1 Discussion V

1) Critical Sections

(a) If we have multiple critical sections in a program, do you recommend (i) having an entry section and an exit section around each critical section or (ii) having one entry section before the first critical section and one exit section after the last critical section? Why?

Answer: It doesn't have a definite answer. Having separate critical sections allow for more parallelism. If different critical sections do not affect common data, then different processes can operate in different critical sections as long as the scheduler allows. Using individual critical sections allows concurrent use of the separate data. While a thread is executing on a critical section, other threads can execute other critical sections protected by different semaphores. On the other hand, having one entry section before the first critical section and one exit section after the last critical section reduces the overhead introduced by the scheduler. However, I'd recommend implementing the former because it makes our program more efficient compared to the latter.

(b) Is there a special case for which your answer will change due to advantages and disadvantages of each version?

Answer: If individual critical sections are not that independent and they modify the same data, then I'd prefer the second implementation. So the answer may change based on the independence of the critical sections from each other. In a different scenario, cost of entering and exiting from a critical section may be so high that implementing one critical section rather than multiple turns out to be more effective.

2) Why do we need different types of semaphores? Please give examples requiring different types of semaphores.

Answer: There are two types of semaphores: Binary semaphores and counting semaphores. Binary semaphores can be likened to mutex locks, it has only two

values -0 and 1. On systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion. On the other hand, value of a counting semaphore can range over an unrestricted domain. Initial value of the semaphore determines the number of processes that can enter their critical sections concurrently. Counting semaphores are used to control access to a given resource consisting of a finite number of instances. If a semaphore value is negative, its magnitude represents the number of processes waiting on that semaphore. One example for usage of counting semaphores is the readers-writers problem in which there are more than one readers. Counting semaphores don't guarantee mutual exclusion, since more than one process or thread can enter the critical section at a time. In binary semaphore, we have mutual exclusion. In counting semaphore, mutual exclusion is not guaranteed.

3) What is the relationship between dining philosophers and process synchronization? Is it an important issue?

Answer: The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems.It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are where all the elements of chopstick are initialized to 1. Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. But this situation can be handled by some rules.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

4) Compared to classical process synchronization problems, what is different in the readers-writers problem? Does it require a special solution or not? Why?

Answer: In bounded buffer (consumer-producer) problem, there is a symmetric synchronization between the producer and the consumer, that is, code written to solve the problem is very similar for both cases. However, in the readers-writers problem, solution requires an asymmetric synchronization, code for a reader process and a writer process differs from each other. In readers-writers problem, more than one reader can read the data simultaneously. To avoid race conditions, readers should also be taken into a queue, denoted by another semaphore.

5) Deadlocks and Starvation

Deadlock happens when every process holds a resource and waits for another process to hold another resource.

Starvation happens when a low priority program requests a system resource but cannot run because a higher priority program has been employing that resource for a long time. In other words, starvation occurs when a process **can acquire the lock**, but it is being overlooked.

(a) Can we have a deadlock without starvation? If yes, please give an example?

Answer: Deadlock means that some thread attempts to acquire the lock, but no threads succeed. That is a special kind of starvation. Therefore it's not possible that there is a deadlock without starvation.

(b) Can we have starvation without a deadlock? If yes, please give two examples.

Answer: Yes, starvation can happen without a deadlock. Say, there are three processes A, B and C. They compete for the same resource. A and B have priority over C in the queue. Immediately after B finishes its task with the resource, it is pushed back to the queue while A is still there. Then A is popped and pushed after it completes its task. This loop continues forever and since C has low-priority, it starves indefinitely. A and B didn't have a critical section that prevented C from executing or vice versa. In this example, there is starvation without a deadlock. Such situations can be solved with aging method or priority-inheritance protocol.