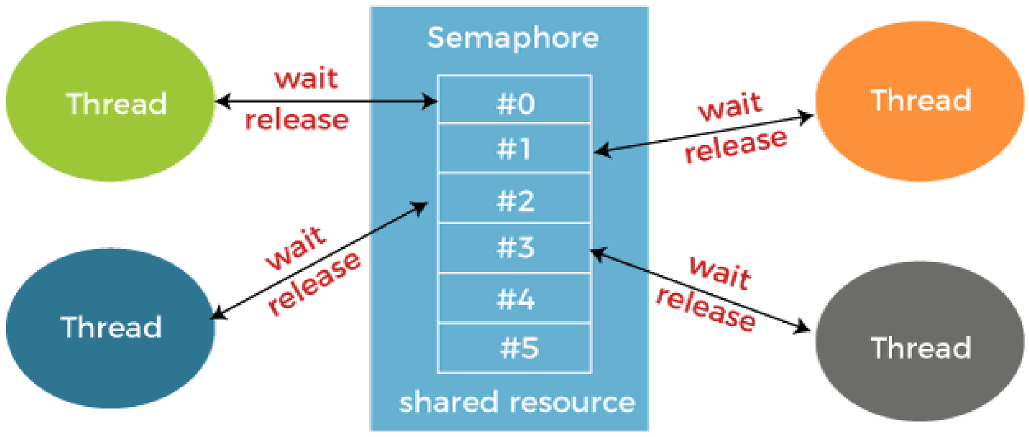
Operating Systems (CS F372)
Tutorial Sheet 8
Semaphore

In this tutorial sheet, we will explore the process synchronization using Semaphores.



Semaphores are synchronization primitives used in concurrent programming. They function as signaling mechanisms, allowing threads or processes to coordinate their activities and avoid race conditions. They are manipulated using atomic wait and signal operations. When a thread attempts access, it performs a wait operation. If the semaphore value exceeds 0, access is granted; otherwise, the thread blocks until a signal operation increases the value, allowing progression. There are two types of semaphores: **binary semaphores** (limited to 0 or 1) act akin to mutex locks, ensuring exclusive resource access, crucial for preventing conflicts. **Counting semaphores** have an unrestricted range and manage multiple resource instances. Processes decrement the count for access and increment it upon release, facilitating efficient synchronization and cooperation among processes.

Unlike mutexes, which permit only one thread at a time, semaphores can handle multiple threads concurrently based on their values. Semaphores are more versatile and can represent a wider range of synchronization scenarios, making them essential tools in concurrent programming for resource management and preventing conflicts in shared environments.

Questions

1. Write a C program to demonstrate how semaphores can be used to control access to a given resource consisting of a finite number of instances. Consider a scenario where 5 threads need to access a shared resource, represented by an array sharedResource[2]. The i-th thread should modify the element at index i % 2. Each thread should increment its respective element in the sharedResource array, print its value, and then sleep for 2 seconds to simulate work. Utilize semaphores to ensure synchronized access to the shared array.

2. Write a C program for a scenario where multiple producers generate items and place them into a shared buffer, while multiple consumers retrieve and process these items. The buffer, with a capacity of 5 items, must be synchronized to prevent race conditions. Producers need to wait if the buffer is full and can only add items when there is available space. Similarly, consumers must wait if the buffer is empty and can only consume items when there are items available. Implement the solution using semaphores to ensure proper coordination, avoiding overproduction or underproduction issues. Ensure mutual exclusion when accessing the buffer and allow both producers and consumers to operate simultaneously.

3. Consider the following solution to the dining philosopher's problem:

   ```
   semaphore chopstick[5];
   /*initialize all elements of chopstick to 1*/
   do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
        // ... eat for awhile ...
        // ... think for awhile ...
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
   } while (true);
   ```

   However, this solution is susceptible to deadlock. Modify the given solution to eliminate the possibility of deadlock. Write a C program implementing the modified solution. Hint: Implement an asymmetric solution.