**Module 3: Discussion**

Process synchronization is a way to organize processes that use shared data. While executing many simultaneous processes, process synchronization aims to sustain shared data consistency. Considering problems such as Bounded-Buffer, Readers-Writers, and Dining-Philosophers Problems, discuss your thoughts on process synchronization. In response to your peers, provide constructive feedback on the process synchronizations that were posted. Include additional code examples if applicable.  
  
Alejandro Ricciardi

Jun 21, 2023

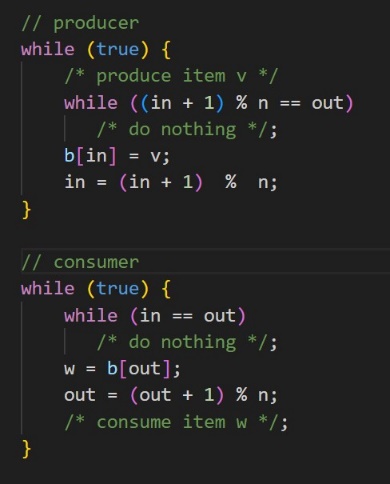
Hello class,

in the context of an operating system (OS), process synchronization can be defined as a set of methods and techniques used to coordinate the concurrent execution of multiple processes or threads. In a multi-process environment, which can also be referred to as "multiprocessing" or "parallel computing", process synchronization plays a crucial role in ensuring that shared data is accessed and modified correctly by addressing several problems that may arise in multi-process environments. Classic problems of process synchronization are the Bounded-Buffer, Readers–Writers, and Dining-Philosophers problems. Solutions to these problems can be developed using tools such as semaphores, binary semaphores,  Message passing, and monitors (Silberschatz et al., 2018, p. 314).

1. **the Bounded-Buffer Problem:**

 The Bounded-Buffer Problem is a classic synchronization problem that involves coordinating the operations of producers and consumers accessing a fixed-size buffer. A Bounded-Buffer can lead to Race Condition.  A Race Condition is “a situation in which multiple threads or processes read and write a shared data item, and the final result depends on the relative timing of their execution” (Stallings, 2018, Chapter 5). Mutual exclusion must be enforced to ensure that only one process can access the shared buffer at a given time.

The following is a Bounded-Buffer producer and consumer pseudocode.



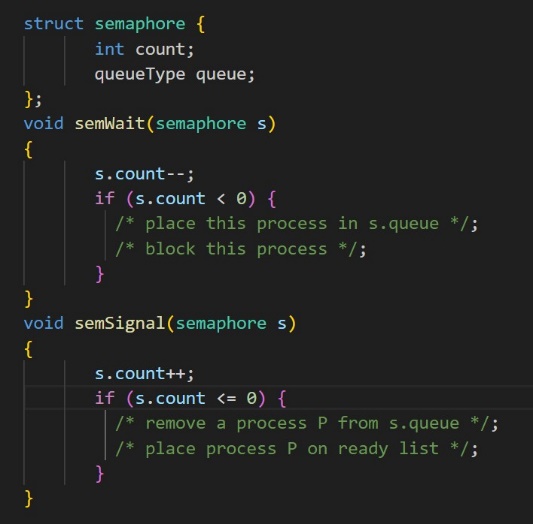
(Stallings, 2018, Chapter 5.4)

The problem Description:

* The fixed-size buffer can hold a limited number of items.
* Multiple producers may add items to the buffer concurrently.
* Multiple consumers may remove items from the buffer concurrently.
* Producers must wait if the buffer is full.
* Consumers must wait if the buffer is empty.

A solution to the Bounded-Buffer Problem is the implementation of semaphore, which is a mutual exclusion method.

A semaphore (also called a counting semaphore or a general semaphore) is an integer value used for signaling among processes, and can only be accessed through two standard atomic operations: semWait() and semSignal(). The letter 's' is generally used to denote a semaphore, below is an example of semaphore pseudocode.

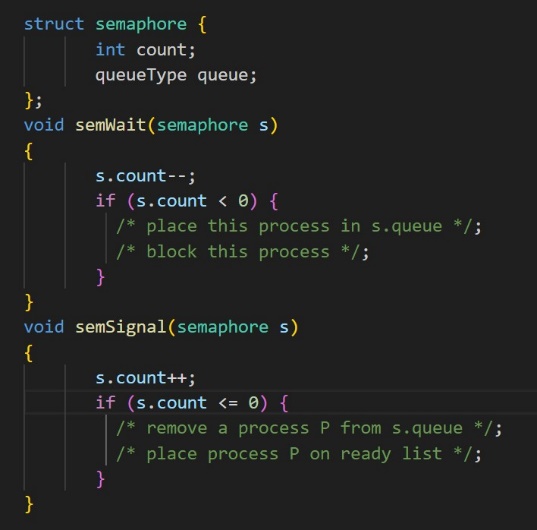


(Stallings, 2018, Figure 5.6)

Not that only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process.

Furthermore, a binary semaphore is a semaphore that takes on only the values 0 and 1, and a mutex is “similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to 0) must be the one to unlock it (sets the value to 1)" (Stallings, 2018, Chapter 5.4).

below is an example of binary semaphore pseudocode:

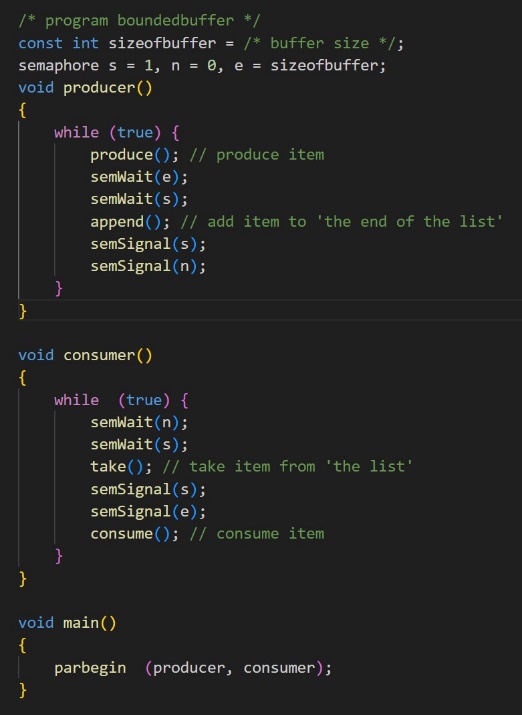


(Stallings, 2018, Figure 5.7)

Note that:

* A binary semaphore may be initialized to zero or one.
* The semWaitB(semaphore s) function checks if the value is zero, then the process executing the semWaitB() is blocked. If the value is one, then the value is changed to zero and the process continues execution.
* The semSignalB(semaphore s)  function checks if the queue is empty (s is equal to zero) on the semaphore, if so, the semaphore is set to one, else a process blocked by a semWaitB() is unblocked (removed from the queue) and placed on the ready list.

The following pseudocode shows a possible solution to the Bounded-Buffer Problem using semaphores



(Stallings, 2018, Figure 5.13)

**2- Readers–Writers Problem**

The Readers–Writers Problem can be described as multiple readers and writers concurrently assessing a resource (e.g., a file or database) and potentially affecting the data integrity. The condition that needs to be met to ensure data integrity are:

* Any number of readers may simultaneously read the file.
* Only one writer at a time may write to the file.
* If a writer is writing to the file, no reader may read it.

Note that readers are processes that are not required to exclude one another, and writers are processes that are required to exclude all other processes, readers and writers alike. (Stallings, 2018, Chapter 5.7)

A solution to the Readers–Writers Problem is the implementation of semaphore or message passing.

Message passing is a mechanism to allow processes or threads to communicate and synchronize their actions. In message passing, communication occurs by explicitly sending a message from a sender to a receiver. The sender encapsulates the data or information to be transmitted into a message and sends it to the receiver. The receiver then receives the message and extracts the information or data from it.

**3- Dining-Philosophers problem**

The dining-Philosophers problem describes a scenario where, for example, five philosophers share a meal at a round table. Every philosopher has a plate and one fork to their right and one fork to their left. They need to use two forks to eat, and it is a total of five forks on the table.  The problem arises when all the philosophers decide to eat at the same time.

The Dining-Philosophers Problem is a classic synchronization problem, it is a metaphor for the problems of deadlock and starvation, which can occur when multiple processes attempt to access multiple resources simultaneously.

Starvation is the situation in which a process or thread waits indefinitely within a semaphore. (Silberschatz et al., 2018, p. G-32)

Deadlock is the state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads. (Silberschatz et al., 2018, p. G-9).

Two main methods exist to prevent or avoid deadlocks. The first one is the deadlock avoidance approach, which grants access to a resource if it cannot result in a deadlock. The second one is the deadlock prevention approach which involves changing the rules so that processes will not make requests that could result in deadlock.

Not that for a deadlock to occur, each of the four conditions listed below must hold:

* Mutual Exclusion: Only one process at a time can use a resource.
* Hold and Wait: A process that holds at least one resource and is waiting to acquire additional resources held by other processes.
* No Preemption: A resource can be released only voluntarily by the process holding it after that process has completed its task.
* Circular Wait: A set of waiting processes.

In other words, a deadlock will not occur by ensuring that at least one of the conditions does not exist. Deadlock prevention prevents at least one of the four conditions of deadlock. On the other hand, deadlock avoidance allows the four conditions, but it allows only three of the four conditions to exist concurrently.

Finally, one of the solutions for Dining-Philosophers Problem is using a monitor-based solution.

A monitor is a programming language construct that allows to put a lock on objects,

The main characteristics of a monitor are the following:

* The local data variables are accessible only by the monitor’s procedures and not by any external procedure.
* A process enters the monitor by invoking one of its procedures.
* Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available. (Stallings, 2018, Chapter 5.5).

In conclusion, process synchronization is a complex subject and it is essential in operating systems to coordinate the concurrent execution of multiple processes or threads. It addresses various problems such as the Bounded-Buffer Problem, Readers-Writers Problem, and Dining-Philosophers Problem. These problems highlight the challenges of managing shared resources, ensuring mutual exclusion, avoiding race condition and deadlock, and preventing starvation.

- Alex

References:

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts*[PDF]. Wiley. Retrieved from: [https://os.ecci.ucr.ac.cr/slides/Abraham-Silberschatz-Operating-System-Concepts-10th-2018.pdfLinks to an external site.](https://os.ecci.ucr.ac.cr/slides/Abraham-Silberschatz-Operating-System-Concepts-10th-2018.pdf)

Stallings, W. (2018). *Operating Systems: Internals and design principles*. Pearson