**CLASSIC PROBLEMS OF SYNCHORNIZATION**

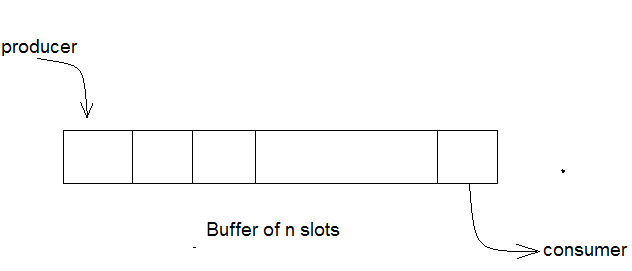
Below are some of the classical problems depicting flaws of process synchronization in systems where cooperating processes are present.

1. Bounded-Buffer (Producer-Consumer) Problem.
2. Readers-Writers Problem.
3. Dining-Philosophers Problem.

Using semaphores for synchronization is the traditional way to present the solutions.

**1)The Bounded-Buffer Problem:**

The bounded buffer problem, which is also called the **producer-consumer problem**, is one of the classic problems of synchronization. There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. The problems arises are

1) The producer must not insert data when the buffer is full.

2) The consumer must not remove data when the buffer is empty.

3)The producer and consumer should not access the buffer simultaneously, which may lead to race conditions and synchronization issues.

To solve the bounded buffer problem, synchronization mechanisms such as semaphores are used.

**Here’s a Solution**

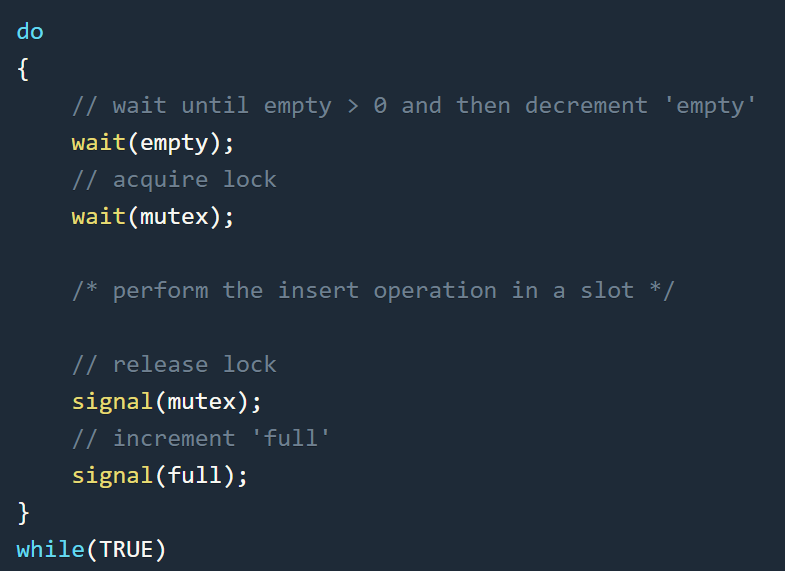
One solution to this problem is to use semaphores. The semaphores which will be used here are:

* m(mutex), a **binary semaphore** that is used to acquire and release the lock.
* empty, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
* full, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and fully represents the number of occupied slots in the buffer.

**The Producer Operation**

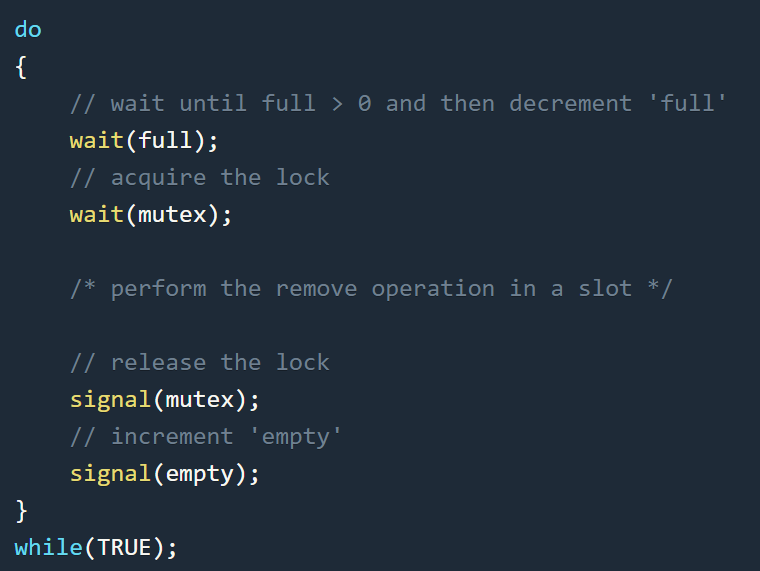
The pseudocode of the producer function looks like this:



* Looking at the above code for a producer, we can see that a producer first waits until there is at least one empty slot.
* Then it decrements the **empty** semaphore because there will now be one less empty slot since the producer is going to insert data in one of those slots.
* Then, it acquires a lock on the buffer, so that the consumer cannot access the buffer until the producer completes its operation.
* After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

# The Consumer Operation

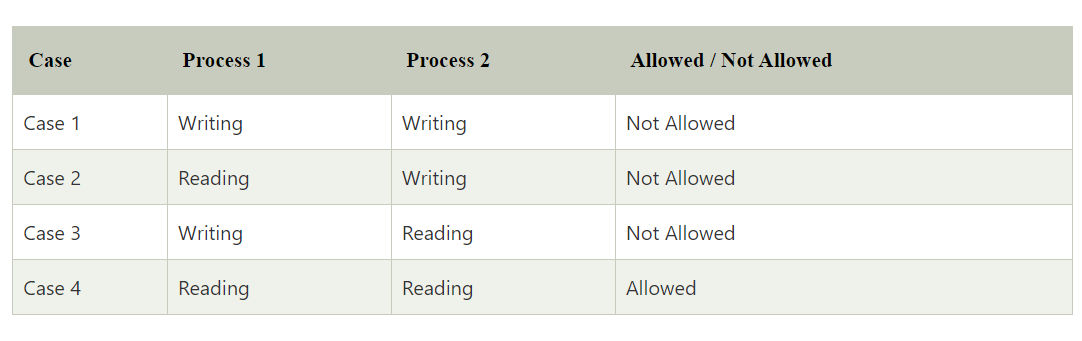
The pseudocode for the consumer function looks like this:



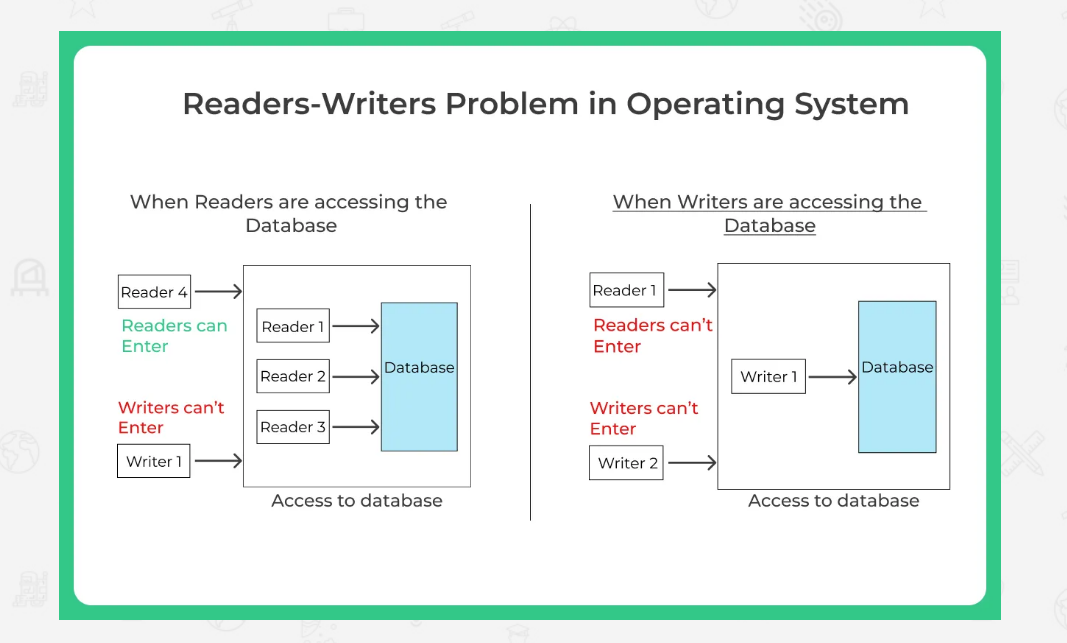
* The consumer waits until there is at least one full slot in the buffer.
* Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one after the consumer completes its operation.
* After that, the consumer acquires a lock on the buffer.
* Following that, the consumer completes the removal operation so that the data from one of the full slots are removed.
* Then, the consumer releases the lock.
* Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

**2)The Readers-Writers problem:**

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.



Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, then it may lead to a race condition or data inconsistency problems. To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem.



The solution of readers and writers can be implemented using two binary semaphores and an integer variable.

1.**Mutex**, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.

2.**wrt,** a semaphore (initialized to 1) common to both reader and writer processes.

3.**readcount,** an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.

**Writer Process:**

The following is the pseudo-code for the writer process

do {

// writer requests for critical section

wait(wrt);

// performs the write

// leaves the critical section

signal(wrt);

} while(true);

* Writer requests the entry to critical section.
* If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
* It exits the critical section.

**Reader Process:**

The following is the pseudo-code for the reader process

do {

wait(mutex); // Reader wants to enter the critical section

readcount++; // The number of readers has now increased by 1

// there is atleast one reader in the critical section

**// this ensure no writer can enter if there is even one reader**

**// thus we give preference to readers here**

if (readcount==1)

wait(wrt);

// other readers can enter while this current reader is inside

// the critical section

signal(mutex);

// current reader performs reading here

wait(mutex);   // a reader wants to leave

readcount--;

if (readcount == 0) // that is, no reader is left in the critical section,

signal(wrt);         // writers can enter

signal(mutex); // reader leaves

} while(true);

i)Reader requests the entry to critical section.

ii)If allowed:

* it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.
* It then, signals mutex as any other reader is allowed to enter while others are already reading.
* After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.

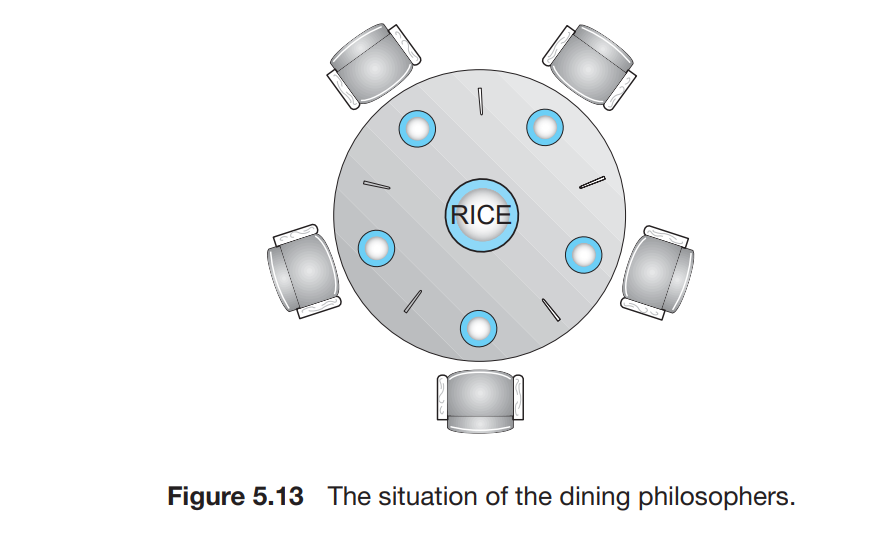
iii)If not allowed, it keeps on waiting.

Thus, the semaphore ‘**wrt**‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

**3)The Dining-Philosophers Problem:**

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating limited resources to multiple processes. The dining philosopher’s problem is a very well-known resource sharing problem in the world of computers.

Consider Five philosophers are sitting around a circular table and their job is to think and eat alternatively. The dining table has five plates, five chopsticks and a bowl of rice in the middle as shown in the below figure.

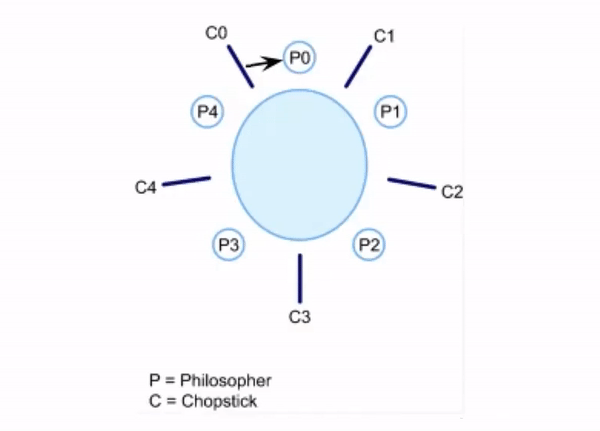


When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her left and her right neighbours). So, while one person is eating, the neighbours to their immediate left and right will have to wait, think, and starve as well. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The **Dining Philosophers Problem** is used to design a scheduling technique such that no philosopher will starve. That means no philosopher can think for an indefinite amount of time.

### **Solution to the problem**

One way to solve the problem is by using Semaphores.

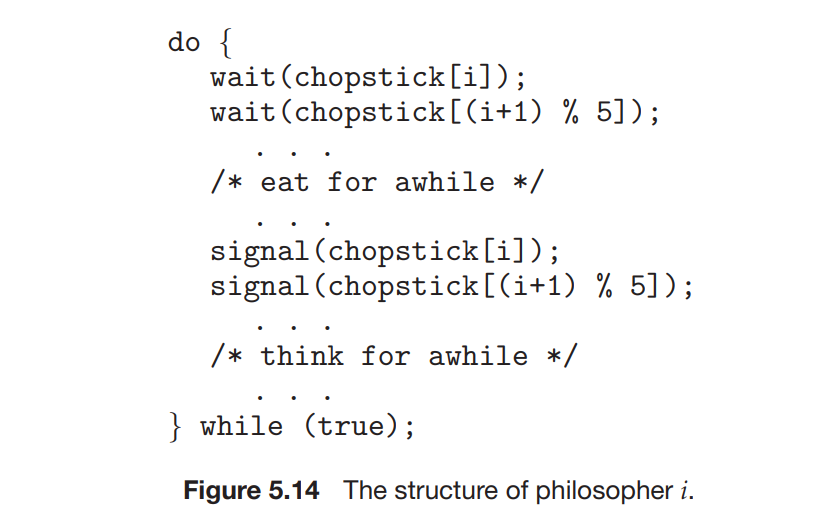


One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores.

Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown below



When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

This is the solution to the problem, but it has some drawbacks as it will 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.

The possible solutions to the deadlock problem are replaced by:

• 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.