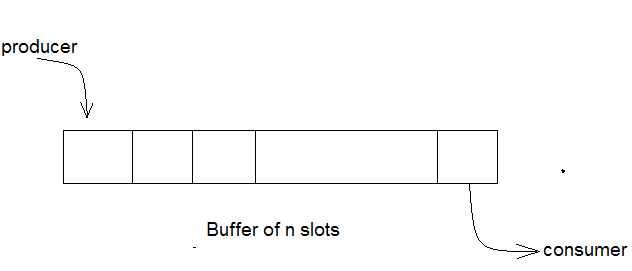
The bounded buffer/ producer consumer problem

<http://www.it.uu.se/education/course/homepage/os/vt18/module-4/bounded-buffer/> ………Last visited: April 10, 2022.

The bounded-buffer problems is a classic example of concurrent access to a shared resource. A bounded buffer lets multiple producers and multiple consumers share a single buffer. Producers write data to the buffer and consumers read data from the buffer.

* Producers must block if the buffer is full.
* Consumers must block if the buffer is empty

<https://www.studytonight.com/operating-system/bounded-buffer> ......Last visited: April 10,2022

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.

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. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

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

* m, a **binary semaphore** which 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 full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

|  |
| --- |
| Do{ |

|  |
| --- |
| // produce an item in nextp Wait (empty); |

|  |
| --- |
| Wait (Mutex); |

|  |
| --- |
| // add the item to the buffer |

|  |
| --- |
| Signal (Mutex); |

|  |
| --- |
| Signal (full); |

|  |
| --- |
| } while (True); |

* Looking at the above code for a producer, we can see that a producer first waits until there is atleast 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 lock on the buffer, so that the consumer cannot access the buffer until 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:

|  |
| --- |
| Do { |

|  |
| --- |
| Wait (full); |

|  |
| --- |
| Wait (Mutex); |

|  |
| --- |
| //remove an item from buffer to next |

|  |
| --- |
| Signal (Mutex); |

|  |
| --- |
| Signal (empty); |

|  |
| --- |
| // consume the item in next |

|  |
| --- |
| } while (true); |

* The consumer waits until there is atleast 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 lock on the buffer.
* Following that, the consumer completes the removal operation so that the data from one of the full slots is 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.

Reader writer problem

[https://www.tutorialspoint.com/readers-writers-problem ......Last](https://www.tutorialspoint.com/readers-writers-problem%20......Last) visited: April 10, 2022

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

<https://www.geeksforgeeks.org/readers-writers-problem-set-1-introduction-and-readers-preference-solution/> …….Last visited: April 10, 2022

Problem parameters: 

* One set of data is shared among a number of processes
* Once a writer is ready, it performs its write. Only one writer may write at a time
* If a process is writing, no other process can read it
* If at least one reader is reading, no other process can write
* Readers may not write and only read

**Solution when Reader has the Priority over Writer**

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution 

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt;  //    **readcnt** tells the number of processes performing read in the critical section, initially 0

**Functions for semaphore :**

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

**Writer process:** 

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

 do {

// writer requests for critical section

wait(wrt);

// performs the write

// leaves the critical section

signal(wrt);

} while(true);

**Reader process:** 

1. Reader requests the entry to critical section.
2. 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.
3. If not allowed, it keeps on waiting.

do {

// Reader wants to enter the critical section

wait(mutex);

// The number of readers has now increased by 1

readcnt++;

// 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 (readcnt==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

readcnt--;

// that is, no reader is left in the critical section,

if (readcnt == 0)

signal(wrt);         // writers can enter

signal(mutex); // reader leaves

} while(true);

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.

<https://www.studytonight.com/operating-system/readers-writer-problem> .... Last visited: April 10,2022

### The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

## The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** m and a **semaphore** w. An integer variable read\_count is used to maintain the number of readers currently accessing the resource. The variable read\_count is initialized to 0. A value of 1 is given initially to m and w.

Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read\_count variable.

while(TRUE)

{

wait(w);

/\* perform the write operation \*/

signal(w);

}

while(TRUE)

{

//acquire lock

wait(m);

read\_count++;

if(read\_count == 1)

wait(w);

//release lock

signal(m);

/\* perform the reading operation \*/

// acquire lock

wait(m);

read\_count--;

if(read\_count == 0)

signal(w);

// release lock

signal(m);

}

* As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
* After performing the write operation, it increments **w** so that the next writer can access the resource.
* On the other hand, in the code for the reader, the lock is acquired whenever the **read\_count** is updated by a process.
* When a reader wants to access the resource, first it increments the **read\_count** value, then accesses the resource and then decrements the **read\_count** value.
* The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
* The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
* Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

<https://afteracademy.com/blog/the-reader-writer-problem-in-operating-system> ..... Last visited: April 10, 2022

In an Operating System, we deal with various processes and these processes may use files that are present in the system. Basically, we perform two operations on a file i.e. read and write. All these processes can perform these two operations. But the problem that arises here is that:

* If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state. Only one process should be allowed to change the value of the data present in the file at a particular instant of time.
* Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read because the process writing on the file will change the value of the file, but the process reading that file will read the old value present in the file. So, this should be avoided.

#### The solution

We can solve the above two problems by using the semaphore variable(learn more about semaphore from [here](https://afteracademy.com/blog/what-is-semaphore-and-what-are-its-types)). The following is the proposed solution:

* If a process is performing some write operation, then no other process should be allowed to perform the read or the write operation i.e. no other process should be allowed to enter into the critical section(learn more about critical section from [here](https://afteracademy.com/blog/what-is-process-synchronization-in-operating-system)).
* If a process is performing some read operation only, then another process that is demanding for reading operation should be allowed to read the file and get into the critical section because the read operation doesn't change anything in the file. So, more than one reads are allowed. But if a process is reading a file and another process is demanding for the write operation, then it should not be allowed.

So, we will use the above two concepts and solve the reader-writer problem with the help of semaphore variables. The following semaphore variables will be used in our solution:

* **Semaphore "writer":** This semaphore is used to achieve the mutual exclusion property. It is used by the process that is writing in the file and it ensures that no other process should enter the critical section at that instant of time. Initially, it will be set to "1".
* **Semaphore "mutex":** This semaphore is used to achieve mutual exclusion during changing the variable that is storing the count of the processes that are reading a particular file. Initially, it will be set to "1".

Apart from these two semaphore variables, we have one variable *readerCount* that will have the count of the processes that are reading a particular file. The *readerCount* variable is initially initialized to 0.

We will also use two function *wait()* and *signal()*. The wait() function is used to reduce the value of a semaphore variable by one and the *signal()* function is used to increase the value of a semaphore variable by one.

The following is the pseudo-code for the process that is writing something in the file:

wait(writer)

...

write operation

...

signal(writer)

The above code can be summarized as:

* The *wait(writer)* function is called so that it achieves the mutual exclusion. The wait() function will reduce the *writer* value to "0" and this will block other processes to enter into the critical section.
* The write operation will be carried and finally, the *signal(writer)* function will be called and the value of the *writer* will be again set to "1" and now other processes will be allowed to enter into the critical section.

This is how we can deal with the process of doing the write operation. Now, let's look at the reader problem.

The following is the pseudo-code for the process that is reading something from the file:

wait(mutex) -----

readerCount++ |

**if**(readerCount == 1) |--- changing the readerCount

**wait**(writer) |

**signal**(mutex) -----

...

read operation

**wait**(mutex) -----

readerCount-- |

**if**(readerCount == 0) |--- changing the readerCount

**signal**(writer) |

**signal**(mutex) -----

The above code can be summarized as:

* We are using the *mutex* variable to change something in the *readerCount* variable. This is done because if some process is changing something in the *readerCount* variable, then no other process should be allowed to use that variable. So, to achieve mutual exclusion, we are using the mutex variable.
* Initially, we are calling the *wait(mutex)* function and this will reduce the value of the mutex by one. After that, the *readerCount* value will be increased by one.
* If the *readerCount* variable is equal to "1" i.e. the reader process is the first process, in this case, no other process demanding for write operation will be allowed to enter into the critical section. So, the *wait(writer)* will be called and the value of the writer variable will be decreased to "0" and no other process demanding for write operation will be allowed.
* After changing the *readerCount* variable, the value of the *mutex* variable will be increased by one, so that other processes should be allowed to change the value of the *readerCount* value.
* The read operation by various processes will be continued and after that when the read operation is done, then again we have to change the count the value of the *readerCount* and decrease the value by one.
* If the *readerCount* becomes "0", then we have to increase the value of the *writer* variable by one by calling the *signal(writer)* function. This is done because if the *readerCount* is "0" then other writer processes should be allowed to enter into the critical section to write the data in the file.

Dining philosophers problem

<https://www.tutorialspoint.com/dining-philosophers-problem-dpp> ..... Last visited: April, 2022

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available.Otherwise a philosopher puts down their chopstick and begin thinking again.

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

**Solution of Dining Philosophers Problem**

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below −

semaphore chopstick [5];

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows −

do {

   wait( chopstick[i] );

   wait( chopstick[ (i+1) % 5] );

   . .

   . EATING THE RICE

   .

   signal( chopstick[i] );

   signal( chopstick[ (i+1) % 5] );

   .

   . THINKING

   .

} while(1);

In the above structure, first wait operation is performed on chopstick[i] and chopstick[ (i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[ (i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

**Difficulty with the solution**

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows −

* There should be at most four philosophers on the table.
* An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
* A philosopher should only be allowed to pick their chopstick if both are available at the same time.

<https://www.javatpoint.com/os-dining-philosophers-problem> ...... Last visited: April 10, 2022

|  |
| --- |
| The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.  The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.  THE DINING PHILOSOPHERS PROBLEM  **Five Philosophers sitting around the table**  **Dining Philosophers Problem**- Let's understand the Dining Philosophers Problem with the below code, we have used fig 1 as a reference to make you understand the problem exactly. The five Philosophers are represented as P0, P1, P2, P3, and P4 and five chopsticks by C0, C1, C2, C3, and C4.  THE DINING PHILOSOPHERS PROBLEM   1. Void Philosopher 2. { 3. **while**(1) 4. { 5. take\_chopstick[i]; 6. take\_chopstick[ (i+1) % 5] ; 7. . . 8. . EATING THE NOODLE 9. . 10. put\_chopstick[i] ); 11. put\_chopstick[ (i+1) % 5] ; 12. . 13. . THINKING 14. } 15. }   Let's discuss the above code:  Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **take\_chopstick[i];** by doing this it holds **C0 chopstick** after that it execute **take\_chopstick[ (i+1) % 5];** by doing this it holds **C1 chopstick**( since i =0, therefore (0 + 1) % 5 = 1)  Similarly suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **take\_chopstick[i];** by doing this it holds **C1 chopstick** after that it execute **take\_chopstick[ (i+1) % 5];** by doing this it holds **C2 chopstick**( since i =1, therefore (1 + 1) % 5 = 2)  But Practically Chopstick C1 is not available as it has already been taken by philosopher P0, hence the above code generates problems and produces race condition. The solution of the Dining Philosophers Problem We use a semaphore to represent a chopstick and this truly acts as a solution of the Dining Philosophers Problem. Wait and Signal operations will be used for the solution of the Dining Philosophers Problem, for picking a chopstick wait operation can be executed while for releasing a chopstick signal semaphore can be executed.  Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:   1. 1. wait( S ) 2. { 3. **while**( S <= 0) ; 4. S--; 5. } 7. 2. signal( S ) 8. { 9. S++; 10. }   From the above definitions of wait, it is clear that if the value of S <= 0 then it will enter into an infinite loop(because of the semicolon; after while loop). Whereas the job of the signal is to increment the value of S.  The structure of the chopstick is an array of a semaphore which is represented as shown below -   1. semaphore C[5];   Initially, each element of the semaphore C0, C1, C2, C3, and C4 are initialized to 1 as the chopsticks are on the table and not picked up by any of the philosophers.  Let's modify the above code of the Dining Philosopher Problem by using semaphore operations wait and signal, the desired code looks like   1. **void** Philosopher 2. { 3. **while**(1) 4. { 5. Wait( take\_chopstickC[i] ); 6. Wait( take\_chopstickC[(i+1) % 5] ) ; 7. . . 8. . EATING THE NOODLE 9. . 10. Signal( put\_chopstickC[i] ); 11. Signal( put\_chopstickC[ (i+1) % 5] ) ; 12. . 13. . THINKING 14. } 15. }   In the above code, first wait operation is performed on take\_chopstickC[i] and take\_chopstickC [ (i+1) % 5]. This shows philosopher i have picked up the chopsticks from its left and right. The eating function is performed after that.  On completion of eating by philosopher i the, signal operation is performed on take\_chopstickC[i] and take\_chopstickC [ (i+1) % 5]. This shows that the philosopher i have eaten and put down both the left and right chopsticks. Finally, the philosopher starts thinking again. Let's understand how the above code is giving a solution to the dining philosopher problem? Let value of i = 0( initial value ), Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **Wait( take\_chopstickC[i] );** by doing this it holds **C0 chopstick** and reduces semaphore C0 to 0**,** after that it execute **Wait( take\_chopstickC[(i+1) % 5] );** by doing this it holds **C1 chopstick**( since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0  Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait( take\_chopstickC[i] );** by doing this it will try to hold **C1 chopstick** but will not be able to do that**,** since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute **Wait( take\_chopstickC[i] );** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait( take\_chopstickC[(i+1) % 5] );** by doing this it holds **C3 chopstick**( since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.  Hence the above code is providing a solution to the dining philosopher problem, A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available else philosopher needs to wait. Also at one go two independent philosophers can eat simultaneously (i.e., philosopher **P0 and P2, P1 and P3 & P2 and P4** can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem) The drawback of the above solution of the dining philosopher problem From the above solution of the dining philosopher problem, we have proved that no two neighboring philosophers can eat at the same point in time. The drawback of the above solution is that this solution can lead to a deadlock condition. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat.  To avoid deadlock, some of the solutions are as follows -   * Maximum number of philosophers on the table should not be more than four, in this case, chopstick C4 will be available for philosopher P3, so P3 will start eating and after the finish of his eating procedure, he will put down his both the chopstick C3 and C4, i.e. semaphore C3 and C4 will now be incremented to 1. Now philosopher P2 which was holding chopstick C2 will also have chopstick C3 available, hence similarly, he will put down his chopstick after eating and enable other philosophers to eat. * A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick. * Only in case if both the chopsticks ( left and right ) are available at the same time, only then a philosopher should be allowed to pick their chopsticks * All the four starting philosophers ( P0, P1, P2, and P3) should pick the left chopstick and then the right chopstick, whereas the last philosopher P4 should pick the right chopstick and then the left chopstick. This will force P4 to hold his right chopstick first since the right chopstick of P4 is C0, which is already held by philosopher P0 and its value is set to 0, i.e C0 is already 0, because of which P4 will get trapped into an infinite loop and chopstick C4 remains vacant. Hence philosopher P3 has both left C3 and right C4 chopstick available, therefore it will start eating and will put down its both chopsticks once finishes and let others eat which removes the problem of deadlock.   The design of the problem was to illustrate the challenges of avoiding deadlock, a deadlock state of a system is a state in which no progress of system is possible. Consider a proposal where each philosopher is instructed to behave as follows:   * The philosopher is instructed to think till the left fork is available, when it is available, hold it. * The philosopher is instructed to think till the right fork is available, when it is available, hold it. * The philosopher is instructed to eat when both forks are available. * then, put the right fork down first * then, put the left fork down next * repeat from the beginning   <https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/> ….. Last visited: April 10, 2022  There exist some algorithm to solve Dining – Philosopher Problem, but they may have deadlock situation. Also, a deadlock-free solution is not necessarily starvation-free. Semaphores can result in deadlock due to programming errors. Monitors alone are not sufficiency to solve this, we need monitors with *condition variables*  **Monitor-based Solution to Dining Philosophers**  We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. Monitor is used to control access to state variables and condition variables. It only tells when to enter and exit the segment. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.  To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:  **THINKING –** When philosopher doesn’t want to gain access to either fork.  **HUNGRY –** When philosopher wants to enter the critical section.  **EATING –** When philosopher has got both the forks, i.e., he has entered the section.  Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).  // Dining-Philosophers Solution Using Monitors  monitor DP  {  status state[5];  condition self[5];  // Pickup chopsticks  Pickup(int i)  {  // indicate that I’m hungry  state[i] = hungry;  // set state to eating in test()  // only if my left and right neighbors  // are not eating  test(i);  // if unable to eat, wait to be signaled  if (state[i] != eating)  self[i].wait;  }  // Put down chopsticks  Putdown(int i)  {  // indicate that I’m thinking  state[i] = thinking;  // if right neighbor R=(i+1)%5 is hungry and  // both of R’s neighbors are not eating,  // set R’s state to eating and wake it up by  // signaling R’s CV  test((i + 1) % 5);  test((i + 4) % 5);  }  test(int i)  {  if (state[(i + 1) % 5] != eating  && state[(i + 4) % 5] != eating  && state[i] == hungry) {  // indicate that I’m eating  state[i] = eating;  // signal() has no effect during Pickup(),  // but is important to wake up waiting  // hungry philosophers during Putdown()  self[i].signal();  }  }  init()  {  // Execution of Pickup(), Putdown() and test()  // are all mutually exclusive,  // i.e. only one at a time can be executing  for  i = 0 to 4  // Verify that this monitor-based solution is  // deadlock free and mutually exclusive in that  // no 2 neighbors can eat simultaneously  state[i] = thinking;  }  } // end of monitor  Above Program is a monitor solution to the dining-philosopher problem.  We also need to declare  condition self[5];  This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs. We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown()  operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:  DiningPhilosophers.pickup(i);  ...  eat  ...  DiningPhilosophers.putdown(i);  It is easy to show that this solution ensures that **no two neighbors** are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death.  <https://www.codingninjas.com/codestudio/library/dining-philosopher-solution-using-monitors> ..... Last visited: April 10, 2022 ****Possibility of Deadlock**** If philosophers take one fork at a time, taking a fork from the left and then one from the right, there is a danger of deadlock.  This possibility of deadlock means that any solution to the problem must include some provision for preventing or otherwise dealing with deadlocks. ****Possibility of Starvation**** If philosophers take two forks at a time, there is a possibility of starvation. Philosophers P2 & P5 and P1 & P3 can alternate in a way that starves out philosopher P4.  This possibility of starvation means that any solution to the problem must include some provision for preventing starvation.  We require an algorithm to distribute these restricted resources (forks) across numerous processes (philosophers) in such a way that the solution is free from deadlock and starvation.    Some algorithms exist to solve the Dining – Philosopher Problem, although they may have a deadlock situation.  Furthermore, a deadlock-free solution does not always imply a starvation-free solution.  Due to programming flaws, semaphores can cause deadlock.  Monitors alone will not be sufficient to address this problem; we will need monitors with condition variables.  Sleeping barber problem  <https://www.geeksforgeeks.org/sleeping-barber-problem-in-process-synchronization/> …. Last visited: April 10, 2022.  **Problem :** The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.   * If there is no customer, then the barber sleeps in his own chair. * When a customer arrives, he has to wake up the barber. * If there are many customers and the barber is cutting a customer’s hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.   **Solution :** The solution to this problem includes three [semaphores](https://www.geeksforgeeks.org/semaphores-operating-system/).First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.  When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up.  When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex.  If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer’s semaphore this wakes up the barber if he is sleeping.  At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.  Lightbox |

Semaphore Customers = 0;

Semaphore Barber = 0;

Mutex Seats = 1;

int FreeSeats = N;

Barber {

while(true) {

/\* waits for a customer (sleeps). \*/

down(Customers);

/\* mutex to protect the number of available seats.\*/

down(Seats);

/\* a chair gets free.\*/

FreeSeats++;

/\* bring customer for haircut.\*/

up(Barber);

/\* release the mutex on the chair.\*/

up(Seats);

/\* barber is cutting hair.\*/

}

}

Customer {

while(true) {

/\* protects seats so only 1 customer tries to sit

in a chair if that's the case.\*/

down(Seats); //This line should not be here.

if(FreeSeats > 0) {

/\* sitting down.\*/

FreeSeats--;

/\* notify the barber. \*/

up(Customers);

/\* release the lock \*/

up(Seats);

/\* wait in the waiting room if barber is busy. \*/

down(Barber);

// customer is having hair cut

} else {

/\* release the lock \*/

up(Seats);

// customer leaves

}

}

}

## ****Process Synchronization****

<https://www.studytonight.com/operating-system/process-synchronization> .... Last visited: April 10, 2022

It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

* It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
* In order to synchronize the processes, there are various synchronization mechanisms.

Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.

<https://afteracademy.com/blog/what-is-process-synchronization-in-operating-system> ...... Last visited: April 10, 2022

In the Operating System, there are a number of processes present in a particular state. At the same time, we have a limited amount of resources present, so those resources need to be shared among various processes. But you should make sure that no two processes are using the same resource at the same time because this may lead to data inconsistency. So, synchronization of process should be there in the Operating System. These processes that are sharing resources between each other are called **Cooperative Processes** and the processes whose execution does not affect the execution of other processes are called **Independent Processes**.

<https://www.tutorialspoint.com/semaphores-in-operating-system> ..... Last visited: April 10, 2022

Semaphores

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

* **Wait**

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

wait(S)

{

   while (S<=0);

   S--;

}

* **Signal**

The signal operation increments the value of its argument S.

signal(S)

{

   S++;

}

**Types of Semaphores**

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

* **Counting Semaphores**

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

* **Binary Semaphores**

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

**Advantages of Semaphores**

Some of the advantages of semaphores are as follows −

* Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
* There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
* Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

**Disadvantages of Semaphores**

Some of the disadvantages of semaphores are as follows −

* Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
* Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
* Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Monitors

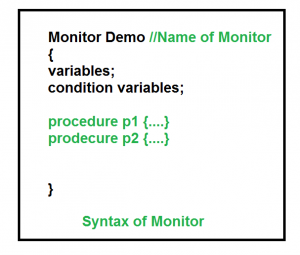
<https://www.geeksforgeeks.org/monitors-in-process-synchronization/> …. Last visited: April 10, 2022

# Monitors in Process Synchronization

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can’t access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

**Syntax:**

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/06/monitors.png)

**Condition Variables:**  
Two different operations are performed on the condition variables of the monitor.

Wait.

signal.

let say we have 2 condition variables  
**condition x, y; // Declaring variable**

**Wait operation**  
x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

**Note:** Each condition variable has its unique block queue.

**Signal operation**  
x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

If (x block queue empty)

// Ignore signal

else

// Resume a process from block queue.

**Advantages of Monitor:**  
Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

**Disadvantages of Monitor:**  
Monitors have to be implemented as part of the programming language . The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.