Unit 3

Process Synchronization

What is Process Synchronization?

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes

Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

Process Synchronization

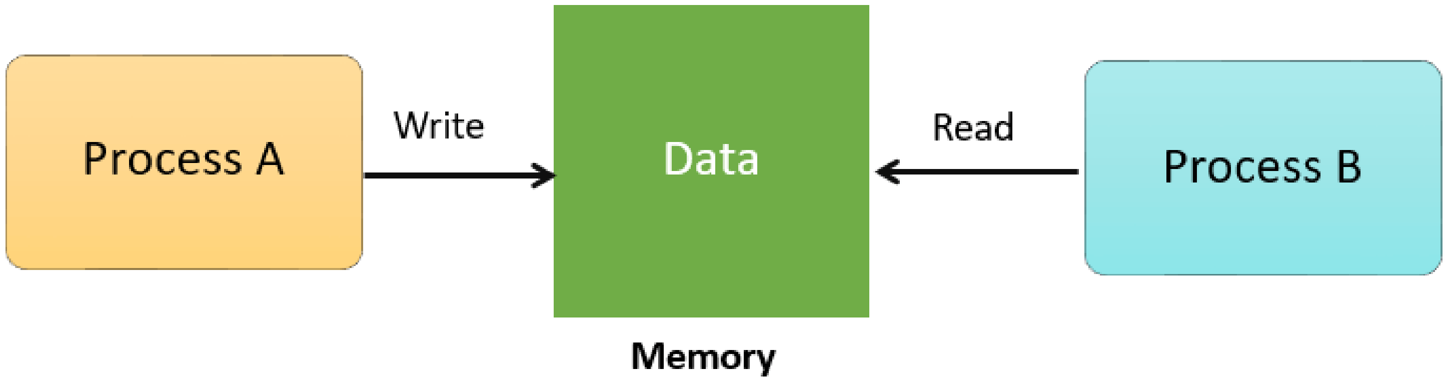
It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

- 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.
- Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.
- It is specially 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 data at the same time.

- This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

How Process Synchronization Works?

For Example, process A changing the data in a memory location while another process B is trying to read the data from the **same** memory location. There is a high probability that data read by the second process will be erroneous.

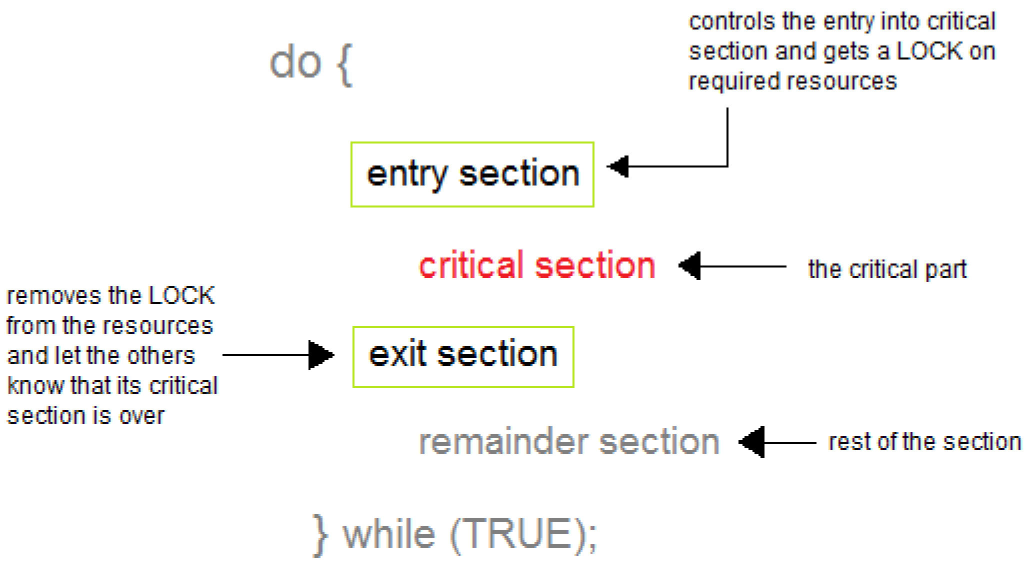


The Critical-Section Problem

A critical section is a segment of code which can be accessed by a signal process at a specific point of time. The section consists of shared data resources that required to be accessed by other processes.

- The entry to the critical section is handled by the wait() function, and it is represented as P().
- The exit from a critical section is controlled by the signal() function, represented as V().

In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution. Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables. In the entry section, the process requests for entry in the **Critical Section.**


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Sections of a Program

Here, are four essential elements of the critical section:

- **Entry Section:** It is part of the process which decides the entry of a particular process.
- **Critical Section:** This part allows one process to enter and modify the shared variable.
- **Exit Section:** Exit section allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It also checks that a process that finished its execution should be removed through this Section.
- **Remainder Section:** All other parts of the Code, which is not in Critical, Entry, and Exit Section, are known as the Remainder Section.

Rules for Critical Section

The critical section need to must enforce all three rules:

- **Mutual Exclusion:** Mutual Exclusion is a special type of binary semaphore which is used for controlling access to the shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems. Not more than one process can execute in its critical section at one time.
- **Progress:** This solution is used when no one is in the critical section, and someone wants in. Then those processes not in their reminder section should decide who should go in, in a finite time.

- **Bound Waiting:** When a process makes a request for getting into critical section, there is a specific limit about number of processes can get into their critical section. So, when the limit is reached, the system must allow request to the process to get into its critical section.

Solutions To The Critical Section

Peterson's Solution

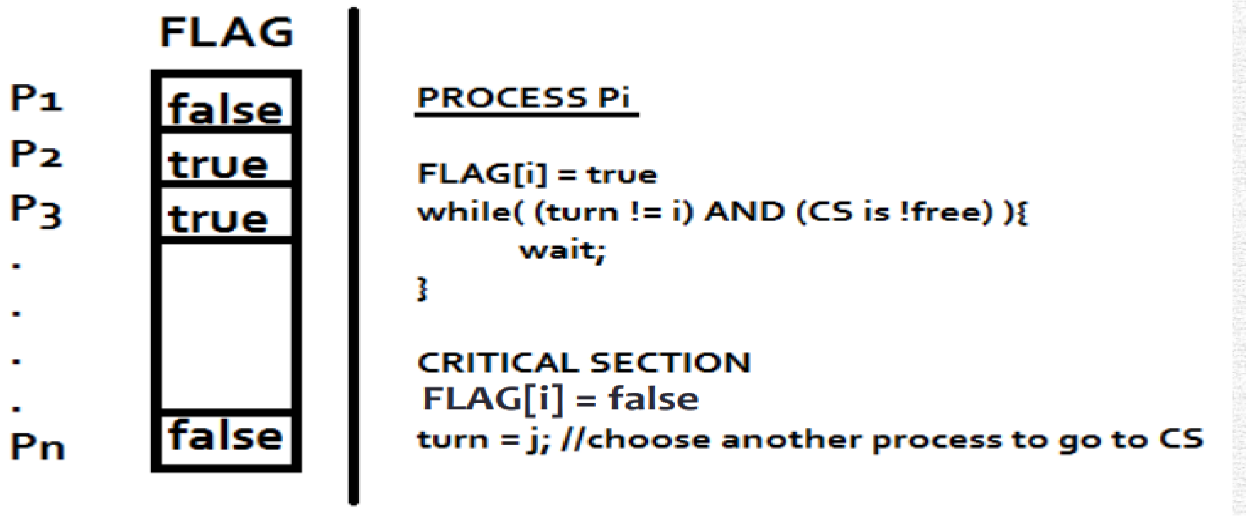
Peterson's Solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts. Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections. For convenience of discussion, "this" process is Pi, and the "other" process is Pj. (I.e. j = 1 - i)

Peterson's solution requires two shared data items:

- **int turn** - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
- **boolean flag[2]** - Indicates when a process **wants to** enter into their critical section. When process i wants to enter their critical section, it sets flag[i] to true.

In the following diagram, the entry and exit sections are enclosed in boxes.

- In the entry section, process i first raises a flag indicating a desire to enter the critical section.
- Then turn is set to **j** to allow the **other** process to enter their critical section **if process j so desires.**
- The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
- Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.



```
FLAG[i] = true
while( (turn != i) AND (CS is !free) ){
        wait;
}

CRITICAL SECTION
 FLAG[i] = false
turn = j; //choose another process to go to CS
```

- Assume there are N processes (P1, P2, ... PN) and every process at some point of time requires to enter the Critical Section
- A FLAG[] array of size N is maintained which is by default false. So, whenever a process requires to enter the critical section, it has to set its flag as true. For example, If Pi wants to enter it will set FLAG[i]=TRUE.
- Another variable called TURN indicates the process number which is currently wating to enter into the CS.
- The process which enters into the critical section while exiting would change the TURN to another number from the list of ready processes.
- Example: turn is 2 then P2 enters the Critical section and while exiting turn=3 and therefore P3 breaks out of wait loop.

To prove that the solution is correct, we must examine the three conditions listed above:

- **Mutual exclusion** - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
- **Progress** - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
- **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

Note that the instruction "turn = j" is ***atomic,*** that is it is a single machine instruction which cannot be interrupted.

Synchronization Hardware

In Synchronization hardware, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software based APIs available to application programmers. These solutions are based on the premise of locking; however, the design of such locks can be quite sophisticated.

These Hardware features can make any programming task easier and improve system efficiency. Here, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem. If we could prevent interrupts from occurring while a shared variable was being modified. The critical-section problem could be solved simply in a uniprocessor environment. In this manner, we would be assuring that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is the approach taken by non-preemptive kernels. But unfortunately, this solution is not as feasible in a multiprocessor environment. Since the message is passed to all the processors, disabling interrupts on a multiprocessor can be time consuming.

System efficiency decreases when this massage passing delays entry into each critical section. Also the effect on a system's clock is considered if the clock is kept updated by interrupts. So many modern computer systems therefore provide special hardware instructions that allow us that we can test and modify the content of a word or to swap the contents of two words atomically—that is, as one uninterruptible unit. We may use these special instructions to solve the critical-section problem in a relatively simple manner. Now we abstract the main concepts behind these types of instructions. The TestAndSet() instruction can be defined as shown in below code.

```
boolean test and set(boolean *target){

  boolean rv = *target;

  *target = true;

  return rv;

}
```

Definition of the test and set() instruction.

The essential characteristic is that this instruction is executed atomically. So, if two TestAndSet C) instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false, if the machine supports the TestAndSet () instruction. The structure of process P, is shown in below.

Example

```
do {

  while (test and set(&lock)) ;

  /* do nothing */

  /* critical section */

  lock = false;

  /* remainder section */

}
```

```
while (true);
```

Mutual-exclusion implementation with test and set().

The SwapO instruction, in contrast to the TestAndSet0 instruction, operates on the contents of two words; it is executed atomically. mutual exclusion can be provided as follows if the machine supports the SwapO instruction. Here, a global Boolean variable lock is declared and is initialized to false. Additionally, each process has a local Boolean variable key. The structure of process P, is shown in figure below.

```
int compare_and_swap(int *val, int expected, int new val){

  int temp = *val;

  if (*val == expected)

  *val = new val;

  return temp;

}
```

Definition of the compare and swap() instruction.

```
do{

  while (compare_and_swap(&lock, 0, 1) != 0) ;

  /* do nothing */

  /* critical section */

  lock = 0;

  /* remainder section */

}

while (true);
```

Mutual-exclusion implementation with the compare and swap() instruction.

Since these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. In below code, we present another algorithm using the TestAndSet() instruction that satisfies all the critical-section requirements.

```
do{

  waiting[i] = true;

  key = true;

  while (waiting[i] && key)

    key = test and set(&lock);

  waiting[i] = false;

  /* critical section */

  j = (i + 1) % n;

  while ((j != i) && !waiting[j])

    j = (j + 1) % n;

  if (j == i)

    lock = false;

  else

    waiting[j] = false;

  /* remainder section */

}

while (true);
```

Bounded-waiting mutual exclusion with test and set().

The same data structures are boolean waiting[n]; boolean lock; These data structures are initialized to false. To prove the point that the mutual exclusion requirement is met, we note that process P; can enter its critical section only if either waiting[i] == false or key -- false. The value of key can become false only if the TestAndSet() is executed. The first process to execute the TestAndSet() will find key == false; all others must wait. The variable waiting[i] may become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.

To prove the point that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either set lock to false or sets waiting[j] to false. Both of them allow a process that is waiting to enter its critical section to proceed. To prove the point that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (z' + 1, i + 2,..., n — 1, 0, ..., i — 1). It designates the first process in this ordering that is in the entry section (waiting [j] =- true) as the next one to enter the critical section.

Any process that waiting to enter its critical section will thus do so within n — 1 turns. Unfortunately for hardware designers, implementing atomic TestAndSet() instructions on multiprocessors is not a trivial task.

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.

Properties of Semaphores

1. It's simple and always have a non-negative integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.

Limitations :

1. One of the biggest limitations of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

Characteristic of Semaphore

- It is a mechanism that can be used to provide synchronization of tasks.
- It is a low-level synchronization mechanism.
- Semaphore will always hold a non-negative integer value.
- Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.

Advantages of Semaphores

Benefits of using Semaphores are as given below:

- With the help of semaphores, there is a flexible management of resources.
- Semaphores are machine-independent and they should be run in the machine-independent code of the microkernel.
- Semaphores do not allow multiple processes to enter in the critical section.
- They allow more than one thread to access the critical section.
- As semaphores follow the mutual exclusion principle strictly and these are much more efficient than some other methods of synchronization.
- No wastage of resources in semaphores because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if any condition is fulfilled in order to allow a process to access the critical section.

Disadvantages of Semaphores

- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.
- To avoid deadlocks in the semaphore, the Wait and Signal operations are required to be executed in the correct order.
- Using semaphores at a large scale is impractical; as their use leads to loss of modularity and this happens because the wait() and signal() operations prevent the creation of the structured layout for the system.
- Their use is not enforced but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in detail in coming lessons.

Classical Problems of synchronization

Semaphore can be used in other synchronization problems besides Mutual Exclusion. Below are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.

We will discuss the following three problems:

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

Bounded Buffer (Producer-Consumer) Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

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.

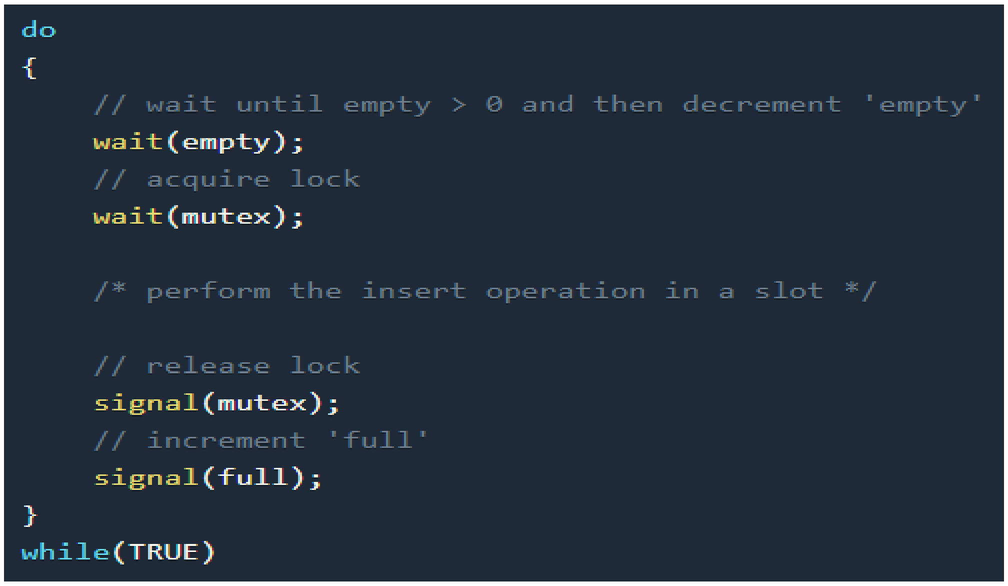
Here's a Solution

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

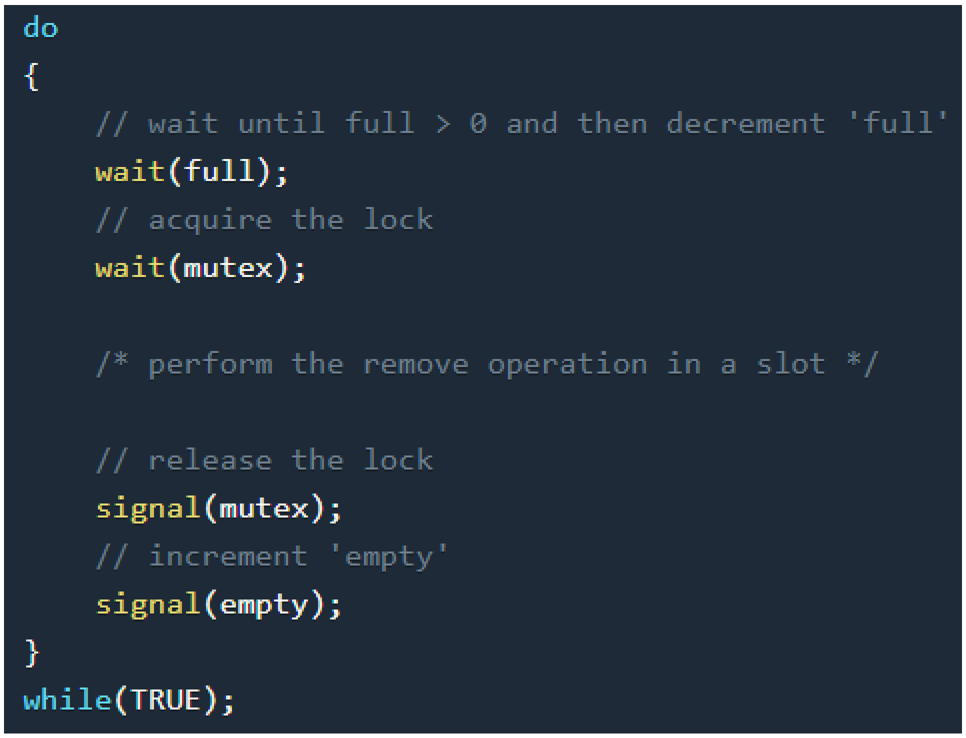


```
do
{
    // wait until empty > 0 and then decrement 'empty'
    wait(empty);
    // acquire lock
    wait(mutex);

    /* perform the insert operation in a slot */

    // release lock
    signal(mutex);
    // increment 'full'
    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 until full > 0 and then decrement 'full'
    wait(full);
    // acquire the lock
    wait(mutex);

    /* perform the remove operation in a slot */

    // release the lock
    signal(mutex);
    // increment 'empty'
    signal(empty);
}
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.

The Readers Writers Problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

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.

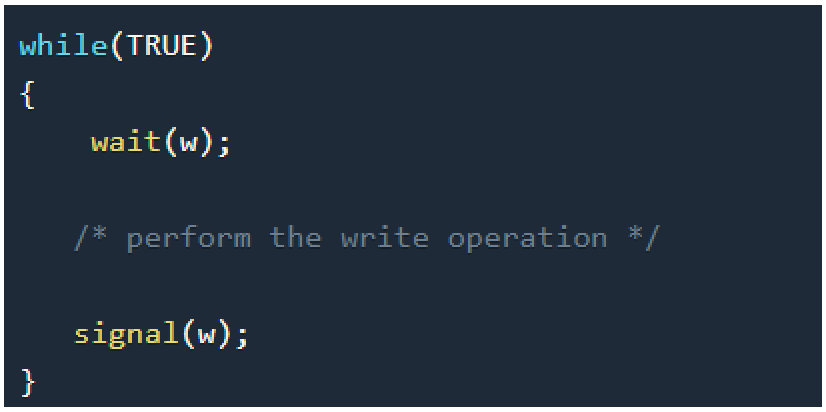
The Solution

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.

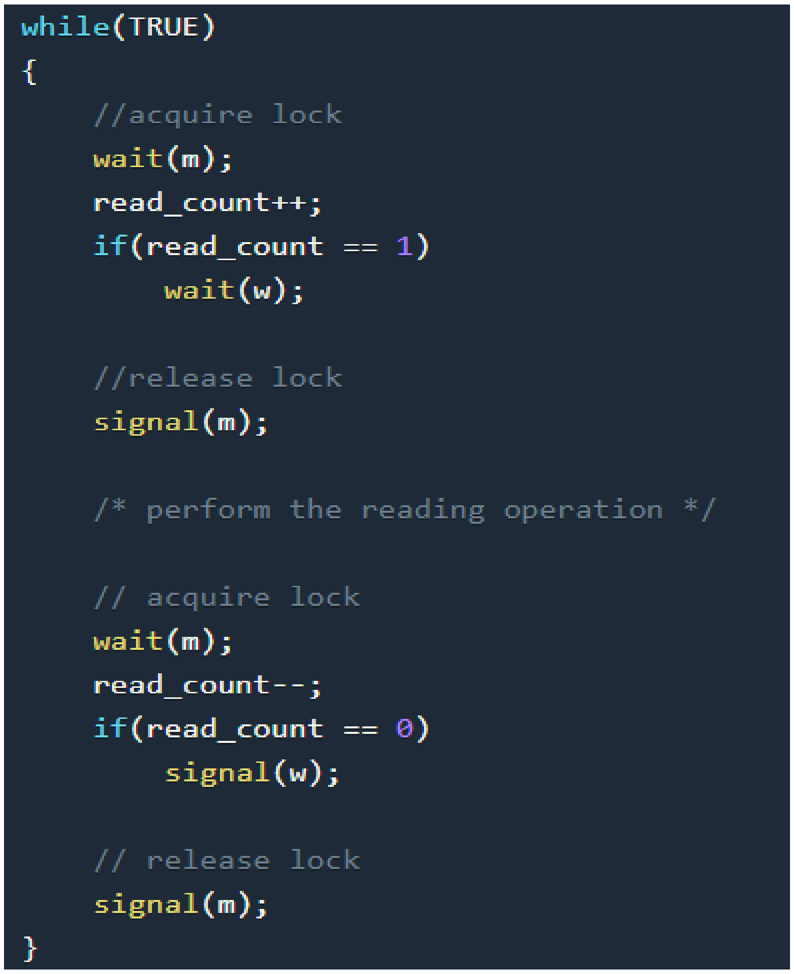
Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

The code for the **writer** process looks like this:



```
while(TRUE)
{
    wait(w);

   /* perform the write operation */

   signal(w);
}
```

And, the code for the **reader** process looks like this:



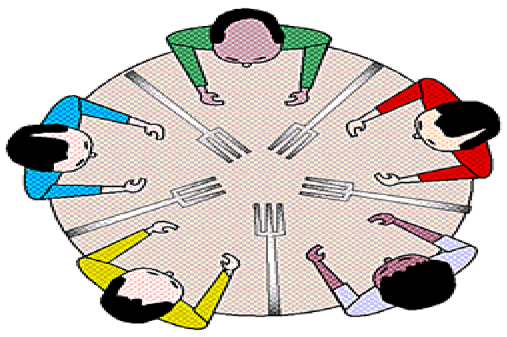
```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);

    //release lock
    signal(m);

    /* perform the reading operation */

    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

    // release lock
    signal(m);
}
```

Here is the Code uncoded(explained)

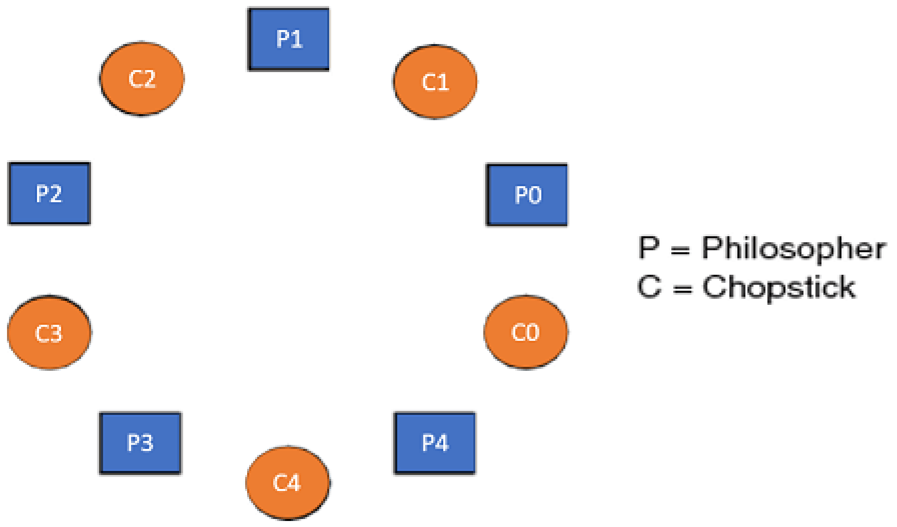
- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock 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.

Dining Philosophers Problem

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.



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.



```
process P[i]
 while true do
   {  THINK;
      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
      EAT;
      PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute PICKUP **[i];** by doing this it holds **C0 chopstick** after that it execute PICKUP **[(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 PICKUP **[i];** by doing this it holds **C1 chopstick** after that it execute PICKUP **[(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:

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5], for each of the five chopsticks.

The code for each philosopher looks like:

```
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.

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.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Monitors Synchronizations in Solaris

- Solaris implements variety of locks to support multitasking, multithreading and multiprocessing. It uses adaptive mutexes, conditional variables, semaphores, read-write locks, turnstiles to control access to critical sections.
- An **adaptive mutex** uses for protecting every critical data item which are only accessed by short code segments.
- On A multiprocessor system it starts as a standard semaphore spin-lock. If the lock is held by a thread which is running on another CPU then the thread spins. If the lock is held by a thread which is currently in run state,the thread blocks, going to sleep until it is awakened by the signal of releasing the lock.
- The spin-waiting method is exceedingly inefficient if code segment is longer. So **conditional variables, semaphores** are used for them.
- Solaris provides **Read-Write** lock to protect the data are frequently accessed by long section of code usually in read-only manner.
- It uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or read-writer lock. Turnstile is a queue structure containing threads blocked on a lock. They are per lock holding thread, not per object. Turnstiles are organized according to priority-inheritance which gives the running thread the highest of the priorities of the threads in its turnstiles to prevent priority inversion.
- Locking mechanisms are used by kernel is also used by user-level threads, so that the locks are available both inside and outside of the kernel. The difference is only that priority-inheritance in only used in kernel, user-level thread does not provide this functionality.
- To optimize Solaris performance, developers refine the locking methods as locks are used frequently and typically for crucial kernel functions, tuning their implementations and use to gain great performance.