UNIT 2

PROCESS SYSNCHRONIZATION

Inter Process Communication (IPC)

A process can be of two types:

1. Independent process.
2. Co-operating process.

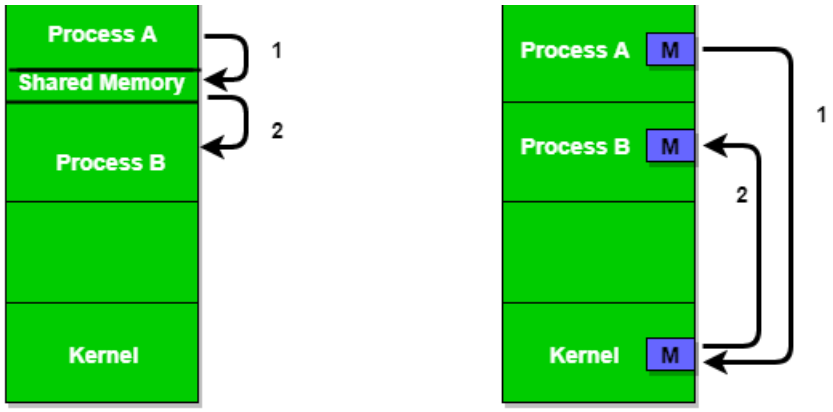
Independent Processes

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.

Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.



Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

An operating system can implement both methods of communication

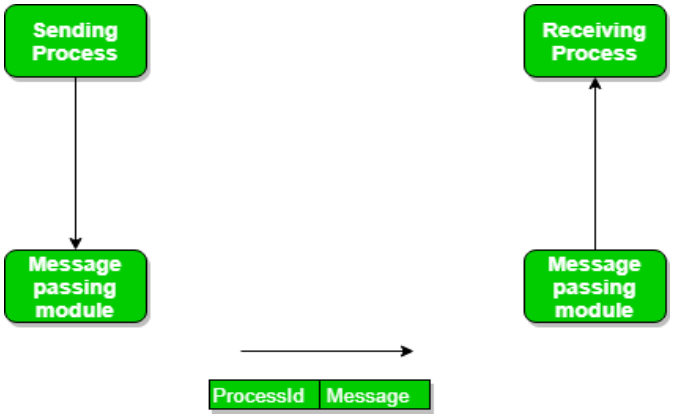
Shared Memory

Shared Memory

1. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it.
2. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly.

3. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes. Let’s discuss an example of communication between processes using the shared memory method.

Messaging Passing Method



In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

1. Establish a communication link (if a link already exists, no need to establish it again.)
2. Start exchanging messages using basic primitives.
   We need at least two primitives:
   – **send**(message, destination) or **send**(message)
   – **receive**(message, host) or **receive**(message)

A standard message can have two parts: header and body.

The header part is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Process Creation vs Process Termination in Operating System

Process Creation

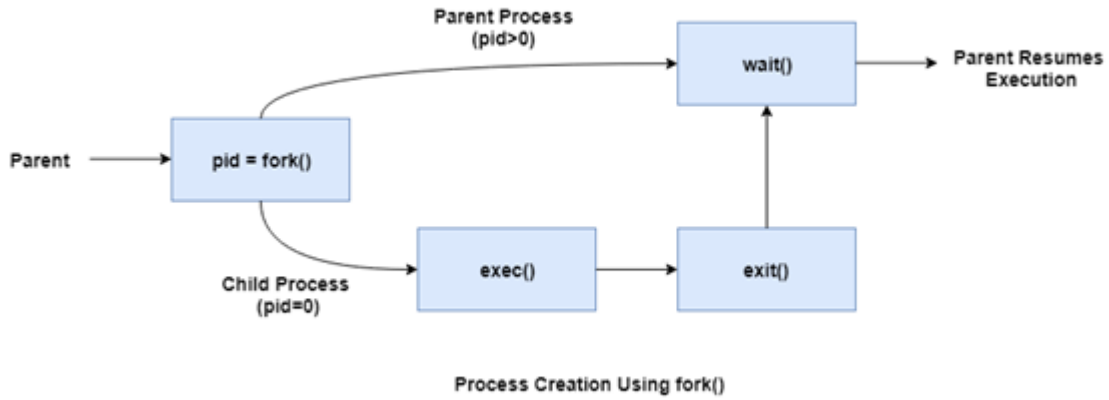
A process may be created in the system for different operations. Some of the events that lead to process creation are as follows −

1. User request for process creation
2. System Initialization
3. Batch job initialization
4. Execution of a process creation system call by a running process

A process may be created by another process using fork().

The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows −



Process Creation Using fork()

Process Creation using fork()

Process Termination

Process termination occurs when the process is terminated The exit() system call is used by most operating systems for process termination.

Some of the causes of process termination are as follows −

1. A process may be terminated after its execution is naturally completed. This process leaves the processor and releases all its resources.
2. A child process may be terminated if its parent process requests for its termination.
3. A process can be terminated if it tries to use a resource that it is not allowed to. For example - A process can be terminated for trying to write into a read only file.
4. If an I/O failure occurs for a process, it can be terminated. For example - If a process requires the printer and it is not working, then the process will be terminated.

In most cases, if a parent process is terminated then its child processes are also terminated. This is done because the child process cannot exist without the parent process.

If a process requires more memory than is currently available in the system, then it is terminated because of memory scarcity.

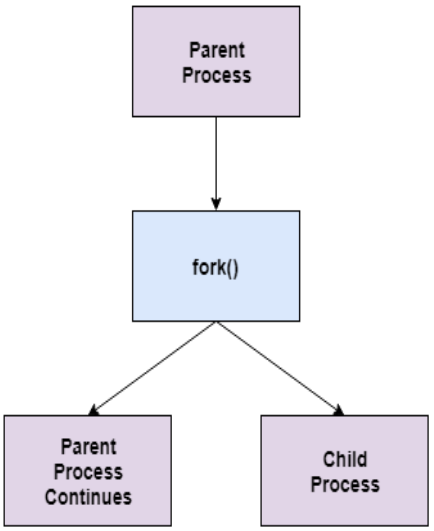
Difference between Process, parent process and child process

Process

1. A process is an active program i.e a program that is under execution. It is more than the program code as it includes the program counter, process stack, registers, program code etc. Compared to this, the program code is only the text section.
2. A process changes its state as it executes. This state partially depends on the current activity of a process. The different states that a process is in during its execution are new, ready, running, blocked, terminated.
3. A process control block is associated with each of the processes. It contains important information about the process it is associated with such as process state, process number, program counter, list of files and registers, CPU information, memory information etc.

Parent Process

1. All the processes in operating system are created when a process executes the fork() system call except the startup process.
2. The process that used the fork() system call is the parent process. In other words, a parent process is one that creates a child process.
3. A parent process may have multiple child processes but a child process only one parent process.



Child Process

1. A child process is a process created by a parent process in operating system using a fork() system call. A child process may also be called a subprocess or a subtask.
2. A child process is created as its parent process's copy and inherits most of its attributes. If a child process has no parent process, it was created directly by the kernel.

Process Synchronization

1. Process Synchronization was introduced to handle problems that arose while multiple process executions.
2. 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.
3. It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
4. In order to synchronize the processes, there are various synchronization mechanisms.
5. 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.

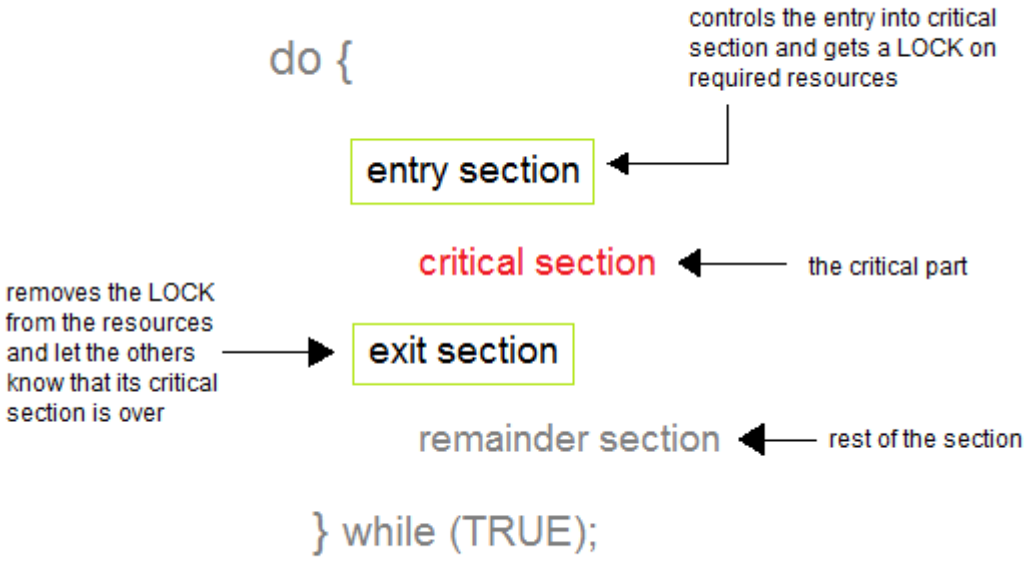
Race Condition

When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition.

Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Critical Section Problem

A Critical Section is a code segment that accesses shared variables and resources. In a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.



Entry Section

In this section mainly the process requests for its entry in the critical section.

Exit Section

This section is followed by the critical section.

The solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

Solutions for the Critical Section

Some widely used method to solve the critical section problem are as follows:

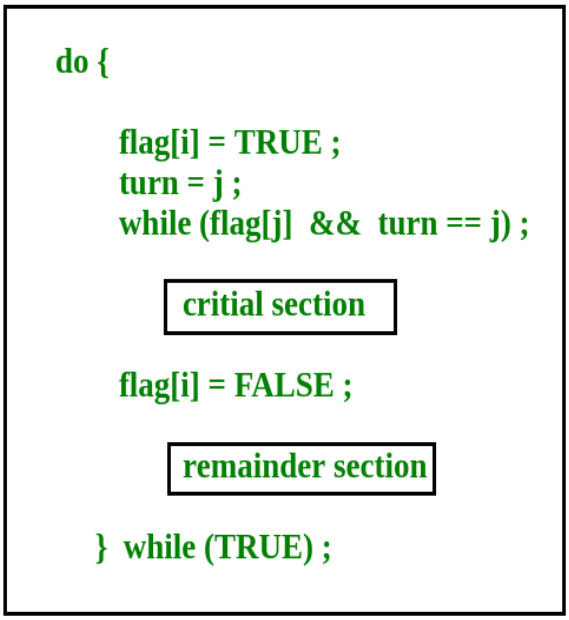
1. Peterson's Solution

This is widely used and software-based solution to critical section problems. Peterson's solution was developed by a computer scientist Peterson that's why it is named so.

With the help of this solution whenever a process is executing in any critical state, then the other process only executes the rest of the code, and vice-versa can happen. This method also helps to make sure of the thing that only a single process can run in the critical section at a specific time.

This solution preserves all three conditions:

a. Mutual Exclusion is comforted as at any time only one process can access the critical section.
b. Progress is also comforted, as a process that is outside the critical section is unable to block other processes from entering into the critical section.
c. Bounded Waiting is assured as every process gets a fair chance to enter the Critical section.



Disadvantages of Peterson's Solution

a. It involves Busy waiting
b. It is limited to 2 processes.

In Peterson's solution, we have two shared variables

a. boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
b. int turn : The process whose turn is to enter the critical section.

Working

a. Let two processes Pi and Pj
b. Pi=0 and Pj=1-Pi (Pi=0 and Pj =1)
c. Initially flag value is initialized to FALSE, because Pi and Pj both are not interested to enter in critical section.

d. Then Pi shows its interest and put flag[i]=TRUE {Pi is interested to enter in critical section} and turn=j {Pi giving first chance of execution to other process}.
e. If condition {while(flag[i] && turn =j } is true, then Pj enters in critical section.
f. After execution of Pj, Pi enters in Critical section.
g. When Pi completed its execution, it put flag[i]= FALSE{ Pi is not interested in execution}

2. Lock Variable

This is the simplest synchronization mechanism. This is a busy waiting solution which can be used for more than two processes.

In this mechanism, a Lock variable **lock** is used. Two values of lock can be possible, either 0 or 1. Lock value 0 means that the critical section is vacant while the lock value 1 means that it is occupied.

A process which wants to get into the critical section first checks the value of the lock variable. If it is 0 then it sets the value of lock as 1 and enters into the critical section, otherwise it waits.

The pseudo code of the mechanism looks like following.

Entry Section →

While (lock! = 0);

Lock = 1;

//Critical Section

Exit Section →

Lock =0;

Working:

a. Initially the value of lock variable is 0. The process which needs to get into the critical section, enters into the entry section and checks the condition provided in the while loop.
b. The process will wait infinitely until the value of lock is 1 (that is implied by while loop). Since, at the very first time critical section is vacant hence the process will enter the critical section by setting the lock variable as 1.
c. When the process exits from the critical section, then in the exit section, it reassigns the value of lock as 0.

Drawback of Lock variable:

The problem with the lock variable mechanism is that, at the same time, more than one process can enter in the critical section. Hence, the lock variable doesn't provide the mutual exclusion that's why it cannot be used in general.

3. Test Set Lock Mechanism

In lock variable mechanism, Sometimes Process reads the old value of lock variable and enters the critical section. Due to this reason, more than one process might get into critical section.

But with test and set instruction, the testing of lock and setting of its value is perform in one go, which provides mutual exclusion.

4. Semaphore

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

a. **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.

b. **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

a. 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.

b. 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.

c. Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

a. Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.

b. 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.

c. Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Counting Semaphore

There are the scenarios in which more than one processes need to execute in critical section simultaneously. However, counting semaphore can be used when we need to have more than one process in the critical section at the same time.

```
int S; // processes that can enter in the critical section simultaneously.
queue type L; // L contains set of processes which get blocked
Wait (Semaphore S)
{
S.value = S.value - 1; //semaphore's value will get decreased when a new process enter in the critical section
if (S.value< 0)
{
put_process(PCB) in L; //if the value is negative then the process will get into the blocked state.
Sleep();
}
else
return;
}

Signal(Semaphore s)
{
```

S.value = S.value+1; //semaphore value will get increased when it makes an exit from the critical section.
if(S.value<=0)
{
select a process from L; //if the value of semaphore is positive then wake one of the processes in the blocked queue.
wake-up();
}
}

Working:

a. Let the semaphore variable S (the value of Semaphore variable indicate the numbers of processes which may enter in critical section at a time.)
b. L denote the queue of blocked processes.
c. Wait() operation
    - S.value = S.value – 1 (semaphore's value will get decreased when a new process enter in the critical section)
    - If condition S<0 , is false (means S is positive, then process may enter in critical section)
    - If condition S<0, is true(means S is negative, then process place in blocked process queue L{Sleep() operation}.

d. Signal() operation
    - S.value = S.value+1 (semaphore value will get increased when any process exit from the critical section)
    - If condition (S.value<=0) ,is true. Means blocked processes are present in blocked queue L. At first send those processes in critical section which are present in blocked queue L{ wake() operation}
    - If condition (S.value<=0) ,is false. Means no blocked processes are present. Increment the value of S, so that next processes may enter in critical section.
e. Counting semaphore do not provide proper mutual exclusion. It only synchronize the resources.

Binary Semaphore or Mutex Lock

Binary Semaphore strictly provides mutual exclusion. Here, instead of having more than 1 slots available in the critical section, we can only have at most 1 process in the critical section. The semaphore can have only two values, 0 or 1.

Semaphore S Value 0 means, Critical section is vacant

Semaphore S value 1 means, critical section is occupied

Queue type L;

```
/* L contains all PCBs corresponding to process
Blocked while processing down operation unsuccessfully.
*/
wait (semaphore S)
{
   if (s.value == 1) // if a slot is available in the
   //critical section then let the process enter in the queue.
   {
      S.value = 0; // initialize the value to 0 so that no other process can read it as 1.
   }
   else
   {
      put the process (PCB) in L; //if no slot is available
      //then let the process wait in the blocked queue.
      sleep();
   }
}
Signal (semaphore S)
{
   if (L is empty) //an empty blocked processes list implies that no process
   //has ever tried to get enter in the critical section.
   {
      S.Value =1;
   }
   else
   {
      Select a process from S.L;
      Wakeup(); // if it is not empty then wake the first process of the blocked queue.
   }
}
```

Working:

a. Let the semaphore variable S (the value of Semaphore variable indicate the numbers of processes which may enter in critical section at a time.). Here S only contain either 0 or 1.
b. L denote the queue of blocked processes.
c. Wait() operation
    - At start S.value is 1 (indicate no process is enter in critical section)
    - Then process turn S.value from 1 to 0 (process Pi enter in critical section)
    - Now S.value is equals to 0
    - If another process Pj tries to enter in critical section, it will get S.value=0 (Process Pj enters in Blocked queue L, stays until S.value==1 again,) {sleep() operation}

d. Signal() operation
    - Check Blocked processes are present in Blocked list L or not. If L is not empty put Blocked processes in critical section.{ wake() operation}
    - If Blocked queue L is empty, turn S.value == 1 from S.value==0.

Classical Problems of Concurrency

1. Producer Consumer Problem/ Bounded buffer Problem

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one processes.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

Let's understand what is the problem?

a. The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
b. Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
c. Accessing memory buffer should not be allowed to producer and consumer at the same time.

Producer Code

```
int count = 0;
void producer( void )
{
 int itemP;
 while(1)
 {
  Produce_item(item P)
  while( count == n);
  buffer[ in ] = item P;
  in = (in + 1)mod n
  count = count + 1;
 }
}
```

Consumer Code

```
int count;
void consumer(void)
{
 int itemC;
 while( 1 )
 {
  while( count == 0 );
  itemC = buffer[ out ];
  out = ( out + 1 ) mod n;
  count = count -1;
 }
}
```

2. Readers-Writers Problem

Consider a situation where we have a file shared between many people.

If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
However if some person is reading the file, then others may read it at the same time.
Precisely in OS we call this situation as the readers-writers problem

Problem parameters:

a. One set of data is shared among a number of processes
b. Once a writer is ready, it performs its write. Only one writer may write at a time
c. If a process is writing, no other process can read it
d. If at least one reader is reading, no other process can write
e. Readers may not write and only read
f. Solution when Reader has the Priority over Writer (Here priority means, no reader should wait if the share is currently opened for reading).

Solution

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

a. **semaphore mutex-** semaphore mutex is used to ensure mutual exclusion for reader enters or exit from the critical section
b. **semaphore wrt** - used by both readers and writers
c. **int readcnt**- readcnt tells the number of processes performing read in the critical section, initially 0

Functions for semaphore:

a. wait() : decrements the semaphore value.
b. signal() : increments the semaphore value.

Writer process:

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

a. Reader requests the entry to critical section.
b. 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.
c. 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.

3. The Dining Philosophers Problem

a. 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.

b. A bowl of noodles is placed at the centre of the table along with five chopsticks for each of the philosophers.

c. 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.
d. 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.

Five Philosophers sitting around the table

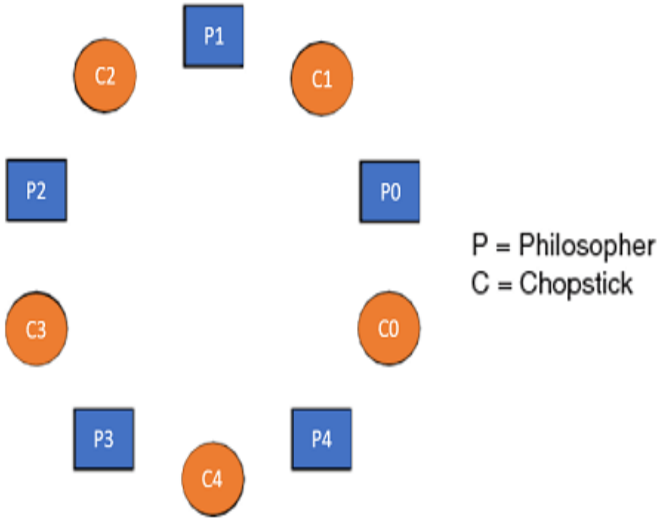
The five Philosophers are represented as P0, P1, P2, P3, and P4 and five chopsticks by C0, C1, C2, C3, and C4.

```
Void Philosopher
 {
 while(1)
  {
   take_chopstick[i];
   take_chopstick[ (i+1) % 5] ;

    EATING THE NOODLE

   put_chopstick[i] );
   put_chopstick[ (i+1) % 5] ;

    THINKING
  }
}
```



Working:

a. 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)
b. 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)

c. 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

a. We use a semaphore to represent a chopstick and this truly acts as a solution of the Dining Philosophers Problem.
b. 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.
c. The structure of the chopstick is an array of a semaphore which is represented as shown below -
   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

```
void Philosopher
 {
 while(1)
  {
   Wait( take_chopstickC[i] );
   Wait( take_chopstickC[(i+1) % 5] ) ;
   . .
   . EATING THE NOODLE
   .
   Signal( put_chopstickC[i] );
   Signal( put_chopstickC[ (i+1) % 5] ) ;
   .
   . THINKING
  }
}
```

Working of Solution Code:

a. 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.
b. 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?

a. 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
b. 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
c. 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 this 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)

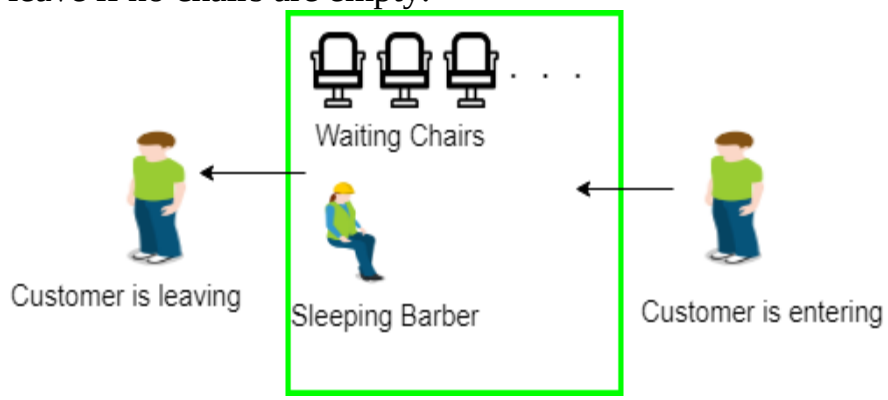
4. Sleeping Barber problem in Process Synchronization

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.

a. If there is no customer, then the barber sleeps in his own chair.
b. When a customer arrives, he has to wake up the barber.

c. 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

a. The solution to this problem includes three semaphores.
   - 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.

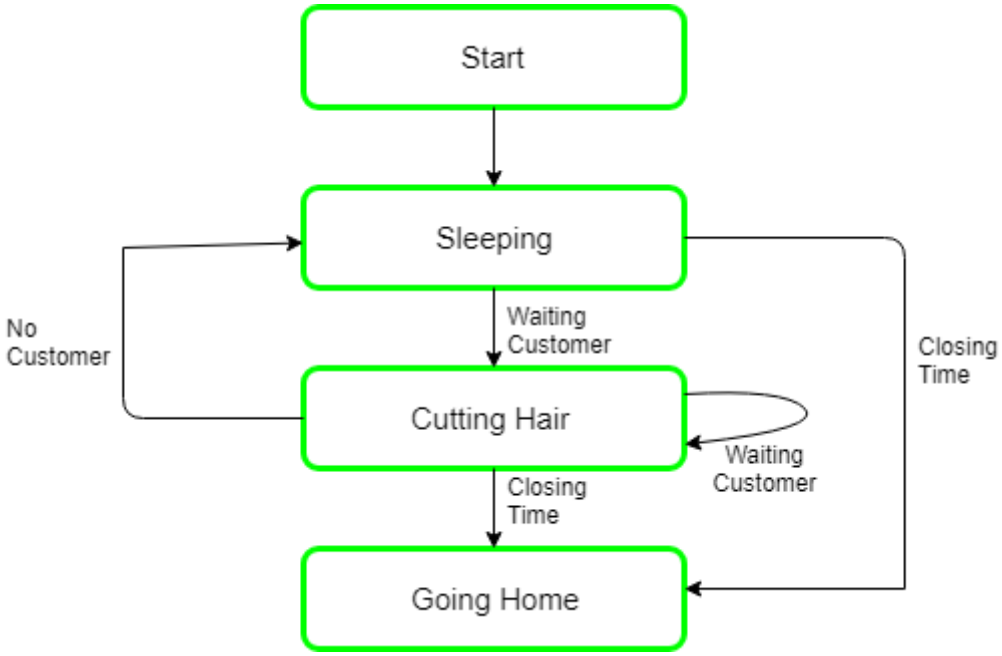
b. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.

d. 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.

e. 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.

f. 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.

g. 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.



Algorithm for Sleeping Barber problem:

```
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
        }
    }
}
```