Analysis of Factors Affecting Process Synchronization

Arti Chhikara
Assistant Professor, Dyal Singh(e)College, University of Delhi, Delhi, India
aartichhikara@gmail.com

ABSTRACT - Process synchronization is one of the important responsibilities of operating system. Process Synchronization involves the methodical sharing of system resources by processes. In this research paper, we will study various methods to ensure the orderly execution of co-operating processes.

Keywords: process, synchronization, mutex, semaphores, critical section, race condition

I. INTRODUCTION

In addition to process scheduling, an another very important responsibility of operating system is process synchronization. Synchronization involves the methodical sharing of system resources by processes.

Concurrency specifies two or more sequential programs (program that specifies sequential execution of statements) that may be executed concurrently as parallel processes.

When two or more processes work on the same data concurrently, strange things can happen. Processes executing concurrently in the operating system may be either independent or co-operating process.

Co-operating Process: A process is said to be co-operating if the execution of a process can affect or be affected by the execution of other processes.

Independent Process: A process is said to be independent if its execution is not affected by other processes in the system.

In this paper, we will study various approaches to guarantee the methodicalimplementation of co-operating processes.

RACE CONDITION

It is a situation when several processes try to access and modify the same data simultaneously and the outcome of the execution depends on the particular order in which the access takes place.

Let us consider two concurrent processes: P1 and P2

P1 is executing the following set of statements:

```
int count;
count=10;
count=count+2;  // the value of count is incremented by 2
```

AndProcess P2 is executing following set of statements:

```
int count;
count=10;
count=count-2;     // the value of count is decremented by 2
```

Here count is the shared variable.

Now when the process P1 and P2 are executing concurrently, the value of variable counter may be 8,10 or 12

However, the only correct value of variable count is 10, which is produced when both the processes P1 and P2 are executed separately.

When the statement count=count+2 is executed, it can be implemented in machine language as follows

```
registerA=count;   // here registerA is a local CPU register
registerA=registerA+2;
count=registerA;
```

When the statement count=count-2 is executed, it can be implemented in machine language as follows

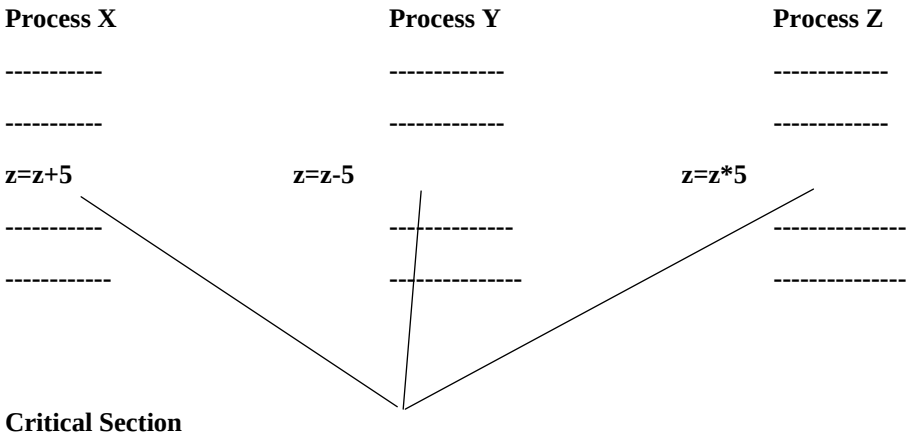
```
registerB=count;   // here registerB is a local CPU register
registerB=register-2;
count=registerB;
```

when these statements are executed concurrently we will get an incorrect state. Such a situation where several processes are accessing and modifying the shared data simultaneously is called a race condition.

II. CRITICAL SECTION

A critical section is that part of the program where shared variables,shared resources will be placed, accessed and updated.

Example:



Critical Section

Here,z is the shared memory resource.

No two processesare allowed to execute in their critical section simultaneously.

It is necessary to design a protocol **called critical section problem** that the processes can use to co-operate. The critical section environment contains

Entry section: Part of the code requesting access into the critical section

Critical Section: Section of the code in which one process can execute at one time

Exit Section: This section refers to the end of the critical section,releasing or allowing others in

Remainder Section:The rest of the remaining code

The general structure of a typical process $P_i$is shown in algorithm below:

```
Repeat

[Entry section;]
Critical section
[Exit section;]
Remainder section

Until false;
```

Fig 1: general structure of a typical process Pi

Solution to the critical section problem

A critical section problem solution must satisfy the following three requirements

- **Mutual Exclusion:** Mechanism that ensure that only one process can access the critical section at a time.
- **Progress:** If a critical section contains no executing process and if some other process wants to enter then thoseprocesses, which are not currently accessing in their remainder section can decide in a finite time that which process should go inside the critical section.
- **Bounded waiting:** Processes that wants to enter into critical section must be bounded by some limit or some finite time so that a process will never be locked out of a critical section indeterminately.

There are basically two general approaches that are used to handle critical sections in operating systems:

(i)**Preemptive kernels**: Here process is allowed to be preempted while it is running in kernel mode. So Preemptive kernels must be carefully designed so as to ensure that shared kernel data are free from race conditions.

(ii)**Non Preemptive kernels**: Here process is not allowed to be preempted while it is running in kernel mode. A process is allowed to run until it exits kernel mode or voluntarily yields control of the CPU. As only one process is active in the kernel at a time, a non-preemptive kernel is essentially free from race conditions.

Peterson's Algorithm

G.L. Peterson discovered a software based solution to the critical section problem in 1981.The solution was discovered to achieve mutual exclusion between two processes.

Here the processes are numbered P0 and P1.

This solution requires the two process to share two data items:

int choose;

boolean interested[2];

Here the variable choose indicates whose turn is to enter its critical section. It can take value 0 and 1. If the value of the variable choose is j i.e choose==j, then process P_j is allowed to execute in its critical section. The arrayinterested[] is used here to specify if a process is ready to enter its critical section.

For instance, if interested[j] is true, this value indicates that P_j is ready to enter itscritical section.

Below figure shows the general structure of a typical process P_j.

```
repeat
interested [j] = true;
choose=k;
While(interested[k] && choose k);
  Critical section;
interested[j]=false;
remainder section
until true;
```

Fig 2: General structure of a process P_jPeterson's solution

Analysis: Here the j^{th} process indicates that it wants to enters the critical section by setting the variable interested[j]=true, then the j^{th} process sets variable choose=k(in order to see if k^{th} process works to enter the critical section or not). Then in while loop it checks , then if interested[k]=true or not. If interested[k]=true, that means k^{th} process wants to enter the critical section. So, if both the conditions are true, then j^{th} process waits. If either of the condition is false, the j^{th} process enters the critical section. If both the process wants to enter the critical section at the same time by setting the value of interested to true and changing the value of choose; but the variable choose will have value that was set lastly. As a result, only one process enters the critical section at a time so **mutual exclusion** is satisfied.

In this case, one process does not block other process from entering the critical section so **progress** condition is also satisfied.

The bounded-waiting requirement is also met because suppose if process $P_k$wants to enter the critical section; can it go second time without letting P_j go?

If $P_k$enters critical section then choose=k; but then interested[j] is set to false on exit thereby allowing $P_j$to enter critical section.

Mutex

Mutex short for Mutual Exclusion is a software tool that is designed to solve the critical section problem.A mutex is a program object that enables sharing of resources by multiple program threads and prevents concurrency. The mutex lock is used to protect the critical regions and thus prevent race conditions. A process can enter critical section only if it acquires lock first and it releases the lock when it exits from the critical section.

This is described in figure below:

```
do
    {
get lock
        Critical section
issue lock
            Remainder section
      }
while(TRUE);
```

Fig3 : critical section problem solution using lock

The get() function acquires the lock and the issue function releases the lock.

```
get()
{
while(!obtainable)
  ;                    /* busy wait
*/
obtainable=false;
}
```

Fig4: get function description

A mutex lock has a boolean variable obtainable whose value specifies if the lock is available or not. If the lock is obtainable, a call to get() is successful, and the obtainable variable is set to false.

The code for release function is given below:

```
issue()
        {
obtainable=true;
      }
```

Fig 5: code for release function

This solution requires no context switch when a processor must wait on a lock, and context switch may take considerable time.However, it offers one disadvantage also; when a process is in its critical section all other processes that try to enter their critical section must wait continuously in the call to get function. This type of mutex lock is also called spinlock because the process “spins” while waiting for the lock to become available. This problem of continual looping is not good for multiprogramming system where a single CPUisaccessed by many processes. Busy waiting leads to wastage of CPU cycles.

Mutex are generally used in multiprocessor environment where one thread can spin on one processor while another thread executes its critical section on another processor.

III. SEMAPHORES

We have already studied simple tool like Mutex locks to solve the critical section problem. But as discussed above, it has a drawback, busy waiting which leads to CPU cycle wastage. Hence a better tool is needed which behave like mutex locks but can also provide better methods for processes to synchronize their activities.

Definition

Semaphore is an effective synchronization tool that can be used to co-ordinate and synchronize activities in systems in which multiple processes compete for the same operating system resource. A semaphore is an integer variable that stores a value that each process can check and change. Based on this value, the process can use the resource or will find that is already in use and must wait for some time before trying again.

Generally, a process that uses semaphores checks the value of semaphore. Depending on the value of semaphore, the process finds that no other process is using the required resource, it obtains the resource then it changes the value of semaphore to highlight this, so that subsequent semaphore users will know to wait if they tried to access the resource.

Types of Semaphores

Semaphores are of mainly two types:

1. Counting semaphores
2. Binary semaphores

Counting Semaphores

A semaphore S is an integer variable, whichcan be accessed only through two standard atomic operations: **wait and signal**. The definition of wait and signal operation are:

```
wait(S): while S<=0 do skip;
            S:=S-1

Signal(S): S:=S+1
```

Fig 6: definition of wait and signal operation

Counting semaphore value can range over an unrestricted domain. They can be used to control access to a given resource consisting of finite number of instances. Let us suppose, if there are five resources available then the semaphore is initialized with value 5. Any process that wants to use a resource performs a wait() operation on the semaphore(thereby decrementing the count i.e 5-1=4)

When a resource is released by a process, signal() operation is performed

(incrementing the count i.e 4+1=5).

When the value of semaphore become 0, it indicates that all resources are being used. Any other process wants to use a resource will get block until the count becomes greater than 0i.e a process releases a resource.

Binary Semaphores

Binary Semaphores are those semaphores whose values are restricted to 0 and 1(locked/unlocked). Binary semaphores are often termed as mutex locks in some systems as they are locks that provide mutual exclusion. But there is a subtle difference between the two. The purpose of mutex and semaphores are different. May be, due to similarity in their implementation a mutex would be referred to as binary semaphore.

Let us consider two concurrently running processes:P1 with a statement S1 and P2 with statement S2. Suppose we require that S2 be executed only after S1 has completed. This scheme can be implemented readily by permitting P1 and P2 share a common variable **S**, initialized to 0.

In process P1, we insert the following statements

S1;

Signal(S);

In process P2, we insert the following statements

Wait(S);

S2;

Execution flow will be

1. S1 executes
2. Signals the semaphore
3. The waiting process P2 receives the signal
4. S2 starts executing.

Semaphore Implementation

The problem with semaphore and mutual exclusion solution is that they suffer from busy waiting i.e. while a process is in its critical region, any other process trying to enter its critical region must continually loop in the entry code. It's clear that through busy waiting, CPU cycles are wasted by which some other processes might use those productively.

To overcome busy waiting, we modify the definition of wait and signal operations. When a wait operation is executed by a process, and finds that the value of semaphore is not positive, the process blocks itself. The block operation places the process into waiting state. After this, the control is transferred to CPU scheduler, which can start execution of another process in queue.

A process that is blocked, i.e. waiting on semaphore S should be restarted by the execution of signal operation by some other processes. The state of the process is

changed from waiting state to ready state. Now the process is ready to run and is placed in the ready queue**.**

To implement semaphores under this condition, semaphore is defined as shown in following figure:

```
struct semaphore
{
intval;
    List *L;        // process
list
}
```

Fig 7: semaphore definition

Here each semaphore has an integer value and a list of processes L. The processes which are waiting on semaphore is added to the list of processes. Signal operation removes one process from process list and wakes that process. The semaphore operations can be now defined as follows:

Description of wait Operation:

```
wait(S)                    // wait
operation
{
S.val--;
If(S.val<0)
     {
add this process to S.L;
block;
     }
}
```

Fig 8: Semaphore operation: wait

Description of Signal Operation

```
signal(S)                  //signal
operation
{
S.val++;
If(S.val<=0)
     {
remove a process P from S.L;
wakeup (P);
      }
}
```

Fig 9: Semaphore operation: signal

The block operation here suspends the process that invokes it and execution of blocked process is resumed by the wakeup () operation. The operating system provides these two basic operations as basic system calls.

In this implementation of semaphores, the value of semaphore may be negative. Under the classical definition of semaphores with busy cycles the value of semaphore can never be negative; so if the value of semaphore is negative its magnitude is the number of processes waiting on that semaphore.

Critical Section Problem

We must make sure that no two process can execute wait () and signal() operations on the semaphore at the same time. This is referred to as critical-section problem**.**We can solve this problem in a single processor environment by simply preventing interrupts during the time the wait () and signal () operations are executing. This strategy works for a single processor environment because, once interrupts are inhibited, statements coming from different processes cannot be mixed**.** Only the process which is running currently is executed until interrupts are re-activatedand scheduler can regain control. While in a multi- processor environment, interrupts must be deactivated on every processor; otherwise instructions coming from different processes maybe interleaved in some random way. Deactivating interrupts on every processor is a tedious job and it can further leadto performance degradation.Therefore, Multiprocessor systems must provide other locking techniques. The problem of busy waiting is not completely eliminated with the above definition of wait () and signal () operations. Relatively the problem of busy waiting is stimulated from the entry section to the critical sections of application programs. This further results in limiting the busy waiting to the critical sections of the wait () and signal () methods and these sections are short. For application programs with high critical times, busy waiting is extremelyunproductive.

IV. DEADLOCKS AND STARVATION

One of the problems concerning implementation of semaphore with a waiting queue is a situation when several processes are blocked indefinitely for an event that can only be caused by one of the waiting process. The processes in such a situation are said to be deadlocked.

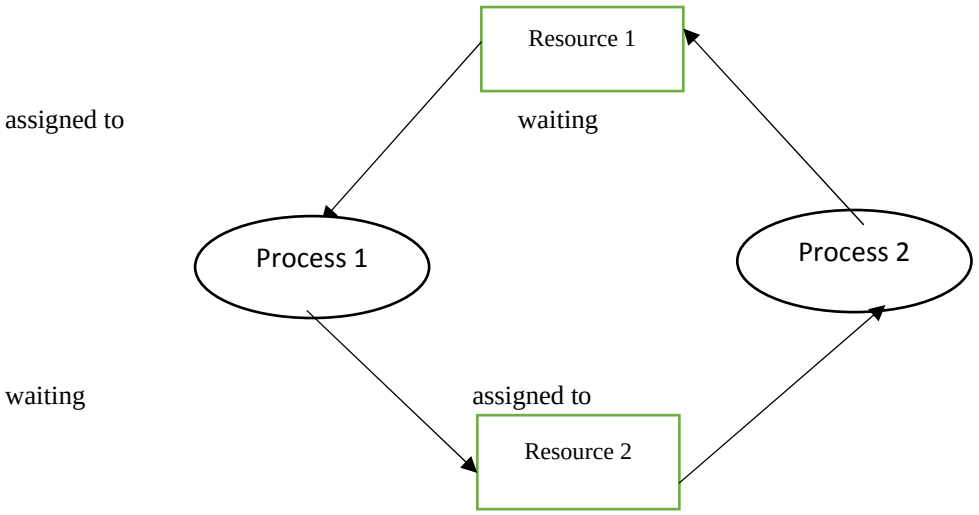


Fig 10 : Deadlock situation

In order to understand this situation, let us take an example: consider a system that contains two processes: P1 and P2.

Let R and T be two semaphores initialized to 1

P1	P2
wait(R);wait(T);	
wait(T);	wait(R);
.	.
.	.
.	.
signal(R);	signal(T);
signal(T);	signal(R);

Suppose P1 executes wait(R) and then P2 executes wait(T). When P1 executes wait(T), it must wait until P2 executes signal(T). When P2 executes wait(R), it must wait until P1 executes signal(R). As a result, these signal operations are not carried out and processes P1 and P2 are deadlocked. Therefore, the processes are said to be in deadlocked state when every process in the set is waiting for the event that can only be caused by another process in the set.

Starvation

Deadlock problem is also closely related to problem of indefinite blocking called starvation, in which a process waits for an event that may never occur or may occur randomly far in the future.

Priority Inversion

Suppose there are two processes P1 and P2, P1 has high priority than P2. Assume that P1 is performing some input output operation. Process P2 enter its critical section for some data item ds. When P1 completes its I/O operation, process P2 is pre-empted and P1 is scheduled. If P1 now tries to enter a critical section for d_s, it would face a busy wait and CPU is denied to the process P2. As a result, the process P2 will not be able to complete its execution of the CS. This also stops process P1 from entering its CS. In this situation process P1 and P2 waits for each other indefinitely. Since here a process with high priority is waiting for a process with low priority, the situation is referred to as priority inversion. This problem occurs only in systems with more than two priorities. So one solution might be to have systems with only two priorities. This solution is not very efficient for most general purpose operating system, so the problem is addressed using the priority inversion protocol.

Priority Inversion Protocol

This protocol states that all low priority processes that are holding the resources temporarily will acquire the priorities of higher priority processes until they are finished with resources in demand. Their priorities ate revert to their original values once they are finished with their execution. In our example the low priority process P2 would temporarily acquire the priority of process P1, which would enable it to get scheduled and exit from its CS.

Classical Problems in Synchronization

In this segment, we will discuss some of the very important synchronization problems. These problems are generally used to test nearly every newly proposed synchronization scheme. Semaphores are used here for providing solution to these problems. They are defined below:

- Producer/Consumer Problem
- Readers and Writers Problem
- Dining Philosopher Problem

Producer/Consumer Problem

In Producer/Consumer problem producer portion of the application produces data and stores it in shared object. The consumer portion reads information from the shared object. For example, a compiler may produce assembly code which is consumed by an assembler. A item can be produced by a producer while consumer is consuming the another item. Both the processes i.e. producer and consumer must be synchronized, so that the consumer does not try to consume data that has not yet been produced.

The producer consumer problem can be solvedeither through unbounded buffer or through bounded buffer

With a bounded buffer

The bounded buffer assumes a fixed buffer size. Here the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.let us assume a pool with n buffers, each having a capacity of holding one item. The semaphore mutex provides mutual exclusion for access to the buffer pool and is set to the initial value 1.

emp semaphore count the number of empty buffers and ful semaphore count the number of full buffers.

Emp initial value is set to n and ful initial value is set to 0.

The producer and consumer process share the following data structures:

int n; // n buffers, each capable of holding one item

semaphore mutex=1; // provides mutual exclusion

semaphoreemp=n; // counts the no of empty buffers

semaphoreful=0; // counts the no of full buffers

The structure of the **producer process**is shown below:

```
do
{
  ……..
  /* produce an item induced next produced
*/
 ……
wait(emp);
wait(mutex);
  ……
/* add next produced to the buffer */
……
signal(mutex);
signal(ful)
}while(true);
```

Fig11: producer process structure

The structure of the **consumer process**is shown below:

```
do
{
wait(ful);
wait(mutex);
……..
  /* remove an item from buffer to next consumed
*/
 ……
signal(mutex);
signal(emp);
……
/* consume the item in next consumed */
……
}
while(true);
```

Fig 12: the structure of consumer process

With an unbounded buffer

In unbounded buffer ,producer consumer problem, there is no practical limit in the size of the buffer. The producer can always produce new items but the consumer may have to wait for the new items; there are always empty positions in the buffer.

Readers and Writers Problem

The readers and writers problem is one of the classical synchronization problem. It is often used to compare and contrast synchronization mechanisms. It is also extremely used practical problem. In concurrent processing data objects like files, records etc. is shared among several processes. Some of the processes just wants to read the data and some of the processes wants to just modify the data. The first type is referred to as readers and second type is referred to as writers. If more than one reader accesses the data, no adverse effect will take place. But if a writer or some other process (may be a reader or writer) access the data concurrently problem may occur. So to ensure that such kind of problems do not arise, we must make sure that writers have exclusive access to the shared database while writing to the database.

Let us understand this problem with the help of practical example:

A railway reservation system consists of huge database system with several processes trying to read and write data simultaneously. Reading data from the database will not cause any problem since no data is changed. The problem lies in writing information to the database. If we do not apply any constraints on access to the database, data may change at any moment. By the time a reading process put the outcome of a request for information to the user, the actual information may have changed. So if a process reads the number of available seats, finds a value one and reports it to customer, before the customer has a chance to make their reservation another process makes a reservation for another customer, changing the number of available seats to zero. Such type of synchronization problem is referred to as readers writers problem.

The following set of conditions should be met to solve the readers writers problem

- A single writer is allowed to perform writing at a time
- More than one reader can read the data at the same time
- When a writer is performing writing, reading is forbidden
- A reader gets access to shareable resource ahead of a writer who is waiting, but it cannot preempt an active writer

The readers writers problem can have several variations:

- First Readers Writers Problem
- Second Readers Writers Problem

First readers writers problem:

This problem requires that no reader be kept waiting unless a writer has already obtained permission to use the shared resource or in other words, a reader should not wait for other readers to finish simply because a writer is waiting.

Implication**:** writers may starve

Second readers writers problem:

This problem requires that if writer is ready, then that writer will have access to the shared resource. In other words if a writer is waiting access to the shared resource, no new readers may start reading.

Implication: readers may starve

Solution(First Readers-Writers Problem):

The reader processes share the following data structures:

semaphorerw_mutex=1; // common to both read and write process

semaphore mutex=1 // ensures mutual exclusion when read_count is updated

intread_count=0; // no of processes currently reading the object

The semaphore rw_mutex

- Functions as a mutual exclusion semaphores for the writers
- Used by the first or last readers that enters or exits the critical section.
- Not used by readers who enter or exit while other readers are in their critical sections

The structure of writer process is given below:

```
do
{
wait(rw_mutex)
     ……..
    ……
     /* writing is performed
*/
     ……..
    …….
signal(rw_mutex);
}
While (true);
```

Fig13: writer process structure

The structure of readers process is given below:

```
do
{
wait(mutex);
read_count=read_count+1;
if(read_count ==1)
wait(rw_mutex);
signal(mutex);
          …….
      /* reading is performed */
          …….
wait(mutex);
read_count=read_count-1;
if(read_count==0)
signal(rw_mutex);
signal(mutex);
}while(true);
```

Fig 14: structure of reader process

If a writer is in critical section and ‘n’ readers are waiting then one reader is queued on rw_mutex and n-1 readers are lined up on mutex. Also when the writer executes signal(rw_mutex), we may restart the execution of either the waiting readers or a single waiting writer.

Reader- Writer Locks

Some systems are provided with reader-writer lock in order to generalize readers writers problem and its solutions.

Two modes are defined for reader writer lock

- Read mode
- Write mode

When a process wants to gain access to resource shared only for reading, it requests the reader-writer lock in read mode and if a process wants to just modify the shared data object it request the lock in write mode.

Reader writer lock in read mode can be acquired by the multiple processors but only one process is permitted to acquire the lock for writing.

Reader-writer locks are mostly used in the following situations:

- In applications where number of readers are more than number of writers.
- In applications where we can easily identify those processes which will perform reading operations and which processes will perform writing operations.

Dining-Philosophers Problem

Problem Description: There are five philosophers sitting around a circular table. Each philosopher spends his life alternatively thinking and eating spaghetti. In front of each philosopher, a plate of spaghetti is kept and a fork is placed between each pair of philosopher. A philosopher needs two chopsticks to eat and only five chopsticks are available. They agree that each one will only use the fork to his immediate left and right. A philosopher gets hungry from time to time and tries to grab the two forks that are immediate left and right to him. When a philosopher is hungry, he eats spaghetti without releasing the forks. When he finishes eating he puts down both his forks and starts thinking again.

The problem is to design processesfree from deadlocks and starvation that can models the activities of philosophers such that each philosopher can eat when hungry and thinks when required.

A simple solution to this problem is to represent each fork with a semaphore and a philosopher can eat with fork by executing wait operation on fork and can release fork after eating by executing signal operation.

The shared data structure is given below:

semaphore fork[5]; // all elements are initialized to 1

The structure of philosopher process is given in figure below:

```
do
{
wait (fork[i]);
wait (fork[(i+1) mod 5]);
    ……………
  // eating
   …….
signal (fork[i]);
signal (fork[(i+1) mod 5]);
     ……..
  // thinking
    ……
}
while(TRUE);
```

Fig15: structure of philosopher process

The above solution although promises that no two neighbors will eat simultaneously. But there is a possibility of deadlock because if we suppose that all philosophers want to eat simultaneously and each grabs his left fork. As a result, all the elements of fork will be equal to 0. And if each philosopher tries to take his right fork, he will be delayed forever.

The problem of dining philosophers can be free from deadlocks with the following remedies:

- At most four philosophers will sit at the table simultaneously
- A philosopher will pick up the fork only if both forks are available

- Allow a philosopher sitting at odd position to pick up left fork first and then right fork and a philosopher sitting at even position to pick up right fork
  first and then his left fork.

V. CONCLUSION

1) Inter process communication allows several processes to share data and resources. Synchronization must be provided to ensure the orderly execution of co-operating processes.
2) We begin with a discussion of critical sections, which are basically used to access the shared resources in a mutually exclusive environment.
3) Several hardware based solutions are also discussed that ensures mutual exclusion. But these solutions are very complicated and tedious for developers to use.
4) We have also discussed a simple tool like Mutex locks to solve the critical section problem. But as discussed above, it has a drawback, busy waiting which leads to CPU cycle wastage. Hence a better tool is needed which behave like mutex locks but can also provide better methods for processes to synchronize their activities.
5) Semaphores are greatly used for providing solution to synchronization problems. Semaphores can be easily accessed by two atomic operations: wait and signal and can be implemented efficiently.
6) Number of classical synchronization problems such as producer-consumer problem, readers-writers problem, dining philosopher problem are discussed in this paper

REFERENCES

[1] Abraham Silberschatz, Peter B. Galvin, Greg Gagne, “Operating System Concepts”, ninth edition,John Willey Publications.
[2] William Stallings ,“Operating Systems: Internals and Design Principles”, seventh Edition, Pearson Publications
[3] D. M. Dhamdhere ,“Operating Systems: A Concept-Based Approach”,second edition, Tata Macgraw Hill
[4] Andres S. Tanenbaum, “Modern Operating System”, third edition, Pearson Publications
[5] Colin Ritchie,“Operating System: incorporating unix& windows”, third edition, bpb publications
[6] K.A.Sumitradevi, N.P.Banashree, “Operating Systems”,second edition, SPD publications.
[7] Harvey M. Deitel,Paul J. Deitel, David R. Choffnes, “Opearting System”, third edition, Pearson Publications

WEB LINKS

[1] https://en.wikipedia.org/wiki/Operating_system
[2] http://www.tutorialspoint.com/operating_system/os_overview.htm
[3] http://www.studytonight.com/operating-system/process-synchronization
[4] https://en.wikipedia.org/wiki/Synchronization_(computer_science)
[5] http://web.cs.wpi.edu/~cs3013/c07/lectures/Section06-Sync.pdf
[6] http://www.slideshare.net/WayneJonesJnr/chapter-6-process-synchronization-1314574
[7] http://www.slideshare.net/sgpraju/os-process-synchronization-semaphore-and-monitors
[8] https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_Synchronization.html