An Improvised Method for Process Synchronization using One Process Code and it's Applications in Operating System

Shivankur Thapliyal
M.C.A
Doon Institute of Engineering and Technology,
Rishikesh, Uttarakhand, India

Renu Bahuguna
M. Tech - CSE
Doon Institute of Engineering and Technology,
Rishikesh, Uttarakhand, India

ABSTRACT
In the anatomy of Process Management[1], which is the fundamental functionality of operating system, for the execution of any process some resources are to be required to accomplished the task. Now the resources shared and unshared in nature. For unshared resources there will be no overhead, processes must wait for that resource, if it will allocate to some other process and when some other process released that resource, then it will be allocate to this one. But when if the resources will come under the category of shared resource , now here we faced some overhead , each processes have similar right to accomplish that resource in similar manner , here many processes conflicts , now a one main challenges comes into existence , and we required to synchronize that processes with respect to these resources.So we have already some code for process synchronization [1] which ensures the two main fundamental objectivity of process synchronization [1] are preserves, that is Mutual Exclusion and Progress, but all code, which would developed previously that were write for two processes for process synchronization, which called "The two process code method for Process Synchronization" and it's also applied on any two processes at a time but not homogeneous manner. A two separate code would write for any two processes at a time which ensure Mutual Exclusion and Progress, but here we do this by using only one code, which guarantee to synchronize any processes and also preserves or ensures the two main fundamental objectivity of Process Synchronization[1] which is Mutual Exclusion and Progress.

General Terms
Operating System, Process Management, Process Synchronization, Process Scheduling, Critical Section Problem.

Keywords
Process Synchronization, Process Synchronization code, Process Synchronization using one Process code, Process Synchronization mechanism in Operating System, Improvised code for Process Synchronization.

1. INTRODUCTION
For the synchronization of processes we partitioned these processes into two types

a. **Independent Processes [2],[6].**
b. **Cooperative Processes [2],[6].**

a. **Independent Processes [2],[6]:** Processes their execution doesn't effects the execution of another processes is called independent processes.

b. **Cooperative Processes [2],[6] :**Processes their execution must effects the execution of some another processes is called cooperative processes.

In the Scenario of Independent Processes resources are nor sharable in nature so each processes executes independently , but in the case of Cooperative processes[2] here resources are sharable in nature so processes execution must depend on some other processes execution.

So we require to synchronize processes in the case of Cooperative Processes[2].

Before describing the detailed introduction about Process Synchronization, we must familiar with these terms and properties of Process Synchronization[2].

1.1 Race Condition
This is one of the major situation which will arise when multiple processes have compete for same resource at a time. We describe this condition through an example:
Suppose we have a process P1 (cooperative process) and a resource R1 (sharable in nature) and these R1 resource have to allocate to process P1 then after some times other processes suppose (P2, P3, P4, P5) comes into main memory and wants to acquire these resource R1 and these all processes generate requests for resource R1 but we also known that processes never wait why ? because these resource R1 is sharable in nature and all these remaining five processes (P2, P3, P4, P5) compete for this resource R1 and after some time when P1 execution completes and it released resource R1 , then a very complex situation may arise that which processes will give this resource R1 and here conflicts all processes for resource R1, that's why we require that we have a perfect solutions to synchronize all processes and this processes confliction for resource R1 (which is sharable in nature) is called Race Condition and this sharable resource is called Critical Section and generally this time of resource allocation problem is called Critical Section Problem.

Note: Race Condition always occur in Cooperative processes and with those resources which is sharable in nature.
Now we see some solutions for Critical Section Problem, for Process Synchronization we require to solve critical section problem and we consider these three terminology which is desirable solution for Critical Section.

1.1.1 Mutual Exclusion [2]
1.1.2 Progress [2]
1.1.3 Bounded Waiting [2]

1.1.1 Mutual Exclusion[2]

In this preferences if one process is in the critical section , then no other process enters in critical section (Here sharable resource is called Critical Section) . In other words only one process enters into critical section or at a time critical section share with only one process other processes, which acquire critical section must wait until process released that critical section. So it's one of the most fundamental property of process synchronization.

1.1.2 Progress [2]

Progress play's a major role in the anatomy of Process Synchronization , progress says that if one process executes their critical section and other processes must wait for critical section then if the process which is presently reside or execute critical section wants to again execute it's own critical section then it's must execute with out any trouble means other processes doesn't interfere with this process until it's released their critical section as their wish. We understand this with the help of an appropriate example, suppose we have a process P1 and a sharable resource R1 (which is critical section for process P1) and at present instance P1 is in their critical section or R1 allocates to P1 but some time later or some other instances processes (P2, P3, P4, P5) resides in main memory and wants to demand critical section but here all remaining processes (P2, P3, P4, P5) must wait and if P1 completely executes their critical section R1 and P1 again executes their critical section R1 then it's must execute other remaining processes (P2, P3, P4, P5) doesn't interfere in the execution of Critical Section (R1) again by process P1.

1.1.3 Bounded Waiting [2]

Bounded Waiting says that if one process executes their Critical Section then other processes which also wants to acquire critical section must wait infinitely until this process which is currently reside in critical section will released or regret critical section as wish.

Here represents the skeleton of Critical Section Problem.

```
do
{
        Entry section
        Critical section
        Exit section
        Remainder section

}while(1);
```

Fig a : Block diagram of critical section problem

Here Shows that how a process enters in Critical Section , now first is called Entry Section , where a process enters in Critical Section , now the second one is called critical section which is a sharable resource generally we called it's Critical Section and now the third one is called exit section where a process released the Critical Section and the last one is called remainder section where the process executes remaining code.

2. SOME PREVIOUSLY KNOWN METHODS

2.1 Turn Variable [3]

Turn Variable is one of the most fundamental method for process synchronization , but we know that we ensures process synchronization if we achieve Mutual Exclusion , Progress and Bounded Waiting , but using turn variable we only achieve Mutual Exclusion means only one process at a time is in Critical Section , but we don't achieve progress and Bounded Waiting , so this method is not very appropriate to do process synchronization.

Let's see some chunks of code , which ensures process synchronization (only mutual exclusion) using Turn Variable.

Initially we assume turn = 0,

```
P0                                  P1

while(1)                            while(1)
{                                   {
        while(turn!=0);                     while(turn!=1);
        //critical section                  //critical section
         turn = 1;                           turn = 0;
        //remainder section                 //remainder  section
        //exit section                      //exit section
}                                   }
```

Fig. b turn variable block diagram

2.2 Flag Variable [3]

Flag variable is also one of the most fundamental method for Process Synchronization , but one main disadvantage using flag variable is , some time system is in deadlock , so the throughput of this method is not very well so that's why this method doesn't contains a well defined practical applications approach , but one main advantage with this method is that it's ensure mutual exclusion , but system is in deadlock so no progress occurred or no bounded waiting.

Let's see some chunks of code of process synchronization using flag variable.

Initially,

flag[0]	flag[1]
F	F

Flag variable also works with two processes , so here we write two process codes for process synchronization using flag variable, initially we use two flag variable name says flag[0] and flag [1] and we assume both variable value False (F) initially , so let's see how Flag variable works.

```
P0                                  P1

while(1)                            while(1)
{                                   {
        flag[0]='T';                        flag[1]='T';
        while(flag[1]);                     while(flag[0]);
        //critical section                  //critical section
        flag[0]='F';                        flag[1]='F';
        //remainder section                 //remainder section
      //exit section                      //exit section
                                          }
}
```

Fig. c A block diagram of flag variable.

So the flag variable also ensures Mutual Exclusion , but not ensures progress and bounded waiting.

2.3 Peterson's Algorithm[3]

Peterson's Algorithm is one of the most appropriate and adequate solution of process synchronization. Before going to the depth of this algorithm firstly we must know that Peterson's also gives a two process solutions for process synchronization , means we have a two process code for process synchronization.

Peterson's Algorithm also ensures all major objectives of Process Synchronization :

2.3.1 Mutual Exclusion
2.3.2 Progress
2.3.3 Bounded Waiting

So , that's why it's becomes one of the most strong algorithm for process synchronization.Generally Peterson's Algorithm is a combination of turn variable method and flag variable.

Symbolically
Peterson's Algorithm = Turn Variable + Flag Variable.
We consider some parameters before going to the chunks of code of Peterson's Algorithm.
Initially we take turn = 0 ,
and flag variables values are as follows :

flag[0]	flag[1]
F	F

Let's see some chunks of code of Peterson's Algorithm.

```
P0                                  P1
while(1)                            while(1)
{                                   {
        flag[0]='t';                        flag[1]='t';
        turn=1;                             turn=0;
        while(turn==1&&f                    while(turn==0&&f
lag[1]=='t');                       lag[0]=='t');
        //critical section                  //critical section
        flag[0]='f'                         flag[1]='f'
        //remainder section                 //remainder section
      //exit section                      //exit section
}                                   }
```

Fig. d : A block diagram of Peterson's Algorithm

So Peterson's Algorithm satisfied Mutual Exclusion, Progress and Bounded Waiting.

2.4 Semaphores[3],[6]

Semaphores is also one of the most convenient and appropriate solution for Process Synchronization , but a one main significant thing , which becomes semaphore most appropriate solution for process synchronization rather than others is that it's also solved the Critical Section problem and also decides the order of execution and Resource management. Peterson's Algorithm only ensures and solved Critical Section problem means it's ensure Mutual Exclusion , Progress and Bounded Waiting but it's doesn't ensure or decide the order of execution of processes and none do Resource management , but Semaphores can do following things:

- **Solving the Critical Section Problem.**
- **Decide the order of execution of processes.**
- **Resource Management.**

Semaphore contains two operations:

- **Wait**
- **Signal**

Initially we take s=1

In Semaphore we write two separate code for these two operations , which create some overhead during process synchronization , but overall it's efficiency and throughput is also very well and one main features which becomes it's most convenient is that decidability of the order of execution of processes.

Let's see some chunks of codes these two operations:

```
Wait operation                      Signal operation
wait(s)                             signal(s)
{                                   {
        while(s<=0);                        s=s+1;
        s=s-1;                      }
}
```

Fig. e : A block diagram of wait , signal operations

Let's see some skeleton of code where these two operations wait and signal have to be used.

Initially we consider s=1;

```
do
{
        wait(s);
        //critical section
        signal(s);
        //remainder section
        //exit section
}while(T);
```

Fig.f : A block diagram of Semaphores

Note: Process also context switch in Critical Section.

3. A NEW PROPOSED METHOD FOR PROCESS SYNCHRONIZATION USING ONE PROCESS CODE

In the all previous methods like (Peterson's Algorithm and Semaphores) all process synchronization codes works with any two processes at a time , means we write two separate code for Process Synchronization , which increases some overhead when we run that sharable code , which works on any two processes but we also achieved the main objectives of Process Synchronization , which is Mutual Exclusion , Progress and Bounded Waiting , but here we do or achieve these three main objectives of process synchronization using only one process code , so no overhead have to occurred or no need to write two separate code for any two processes for process synchronization.

The major objectives which have to achieve using this method are as follows:

- Any number of processes have to shared this code.
- At any time only one process is in the Critical Section , so Mutual Exclusion ensures.
- If a process executes it's critical section and if it wants to execute again it's critical section then it's executes again and again , so progress ensures.
- If a one process is in critical section and another process wants to enter in critical section , then it's must wait so Bounded Waiting also ensures.
- The Time complexity and Space complexity of this code is very easy and simple.
- This code follows versatility no need to write two separate code for synchronization , so no overhead occur during Processes Synchronization.

So these are some major advantages of this code , which discussed above.
The skeleton of this code are as follows:

Here we use two variables to synchronize processes , which is s1 and m.

Initially we take m=0,

```
while(1)
{
        s1=m--;
        while(s1);
        /*critical section*/
        s1++;
        m++;
        /*remainder section*/
        /*exit section*/
}
```

Fig.g : A block diagram of proposed method

Flow chart of this code:

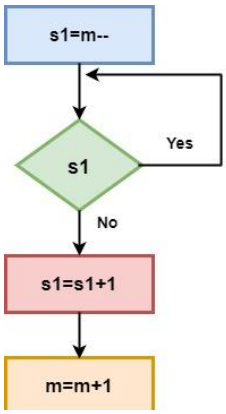

Fig. h : A flow chart of this code

So this is the code for process synchronization , Now we describe that how it's satisfied the three major objectives of process synchronization , which is Mutual Exclusion , Progress and Bounded Waiting one by one.

Mutual Exclusion[4]

Non Pre-emptive processes[4]

We take some literature explaining to verify this:
Suppose we have process p1 like this
And initially the value of m=0 , firstly we check for Non pre-emptive process,

```
P1 , initially m=0

while(1)
{
        s1=m--; //s1=0 and m=-1
        while(s1); //here condition false
        critical section //P1 in CS
        s1++;
        m++;
        remainder section
        exit section
}
```

```
P2 comes after p1 , m=-1

while(1)
{
        s1=m--; //here s1=-1 and m=-2
        while(s1); //condition true , P2 here
        critical section //P1 in CS
        s1++;
        m++;
        remainder section
        exit section
}
```

Fig. I : A block diagram of non preemptive process

So you see that when we use Non Pre-emptive approach only one process is in the critical section , another process have to wait , In the above example when process P1 is in the critical section , P2 must wait and trap in while loop , Now when process P1 comes out of it's Critical Section , then P2 entering in Critical Section , Let's see:

```
P1 , currently s1=-1 and m=-2  after comes P2

while(1)
{
        s1=m--; //s1=0 and m=-1
        while(s1); //here condition false
        critical section
        s1++; // here s1=0
        m++; // m=-1
        remainder section
        exit section
}
```

```
P2 after executing p1 s1 = 0 and m=-1

while(1)
{
        s1=m--; //here s1=-1 and m=-2
        while(s1); //condition false
        critical section //P2 in CS
        s1++;
        m++;
        remainder section
        exit section
}
```

Fig. j : A block diagram of non preemptive process

So you see that when P1 comes out from it's Critical Section , then P2 entering it's critical section , so when we take those processes which doesn't pre-empt during execution or Non Pre-emptive in nature then we see Mutual Exclusion satisfied.
Note: this code shares any number of or n number of processes.

Pre-emptive Processes[4]

Now we shows that how this code works as same as Pre-emptive processes.Initially take m=0,

```
P1 , initially m=0

while(1)
{
        s1=m--; // here s1=0 and m=-1
      ……..//this place P1 pre-empt.
        while(s1);
        critical section
        s1++;
        m++;
        remainder section
        exit section
}
```

```
P2 comes after P1

while(1)
{
        s1=m--; //here s1=-1 and m=-2
        while(s1); //P2 trap in this loop
        critical section
        s1++;
        m++;
        remainder section
        exit section
}
```

Fig. K: A block diagram of preemptive process

Now you see in this above code firstly P1 comes into the memory and wants to execute Critical Section and run this code , initially we take m=0 , so according to the code when P1 runs the code s1 =0 and m=-1 and then after executing this parameter P1 pre-empt and suddenly P2 comes into the memory and it's also wants to execute Critical Section but P2 also run this code and when P2 run this code the current value of m=-1 so here s1=-1 and m=-2 and the condition of while loop is true P2 trap in this while loop until P1 enters in Critical Section and after execution Critical Section by P1 when it goes out Critical Section , then P2 have a chance to enter in critical Section , so Mutual Exclusion also satisfied for Pre-emptive Processes using this code.

```
P1 , initially m=0

while(1)
{
        s1=m--; // here s1=0 and m=-1
……..//this place P1 pre-empt.
// when P1 achieve resources it's      enter critical section
here s1=0 is store in their PCB AC Register

        while(s1); //condition false
        critical section //P1 is in CS
        s1++; //after executing CS s1=0
        m++; // here m=-1
        remainder section
        exit section
}
```

```
P2 comes after P1

while(1)
{
        s1=m--; //here s1=-1 and m=-2
        while(s1); //P2 trap in this loop

//when P1 comes out critical Section ,then s1=0 and P2 is
in critical section

        critical section //here p2
        s1++; //here s1=1
        m++; here m=0
        remainder section
        exit section
}
```

Fig. l: A block diagram of preemptive process

Now We see that after achieving resource by P1 , execute it's critical section and P1 also read S1 = 0 value and store the previous state in their PCB and after executing it's critical section when P1 exits Critical Section s1++ gives result 0 because P2 can do currently S1 value is -1 so after s1++ the results will become 0 and similarly m1++ gives result -1 because P2 can do m1=-2 , so after executing m1++ the result will become -1 so , when s1=0 P2 is in Critical Section and when P2 exits Critical Section S1 will become 1 and m will become 0 and next process will chance to enter it's Critical Section.

Progress[4]

Progress means if a Process is in their Critical Section and it's wants to execute again their critical section then , it's must executes with our any discrepancy.
Progress may apply on one process or more than one process.
For one Process:
Initially we take m=0, the chunks of code are as follows:

```
P1 , initially m=0

while(1)
{
        s1=m--;     //here s1=0 and m=-1
        while(s1); //condition false
        critical section //p1 is in CS
        s1++;       //here s1=1
        m++;       //here m=0
        remainder section
        exit section
}
```

```
Again P1 execute , Now  m=0

while(1)
{
        s1=m--;     //here s1=0 and m=-1
        while(s1); //condition false
        critical section //p1 is in CS
        s1++;       //here s1=1
        m++;       //here m=0
        remainder section
        exit section
}
```

Fig. m: A block diagram of progress

So, we easily see that we achieve progress , a process is in their critical section or have to executes their critical section as many as their wish.

Bounded Waiting[4]

Bounded Waiting means a process must wait when some another process is in their critical section and , using this code we easily seen that their must be have a bounded waiting. In other words when multiple processes comes into the system , then each process must wait for critical section , which is currently assigned at that moment to that remaining process , and each process bounded waited to each other.

At the end we conclude that this one process code for process synchronization is reliable to work with any number of process and preservers the three main objectives of Process Synchronization through which we ensure that a process must be synchronize and these three objectives are (Mutual Exclusion , Progress and Bounded Waiting).

4 APPLICATIONS OF THIS METHOD[5]

There are a numerous applications of this one process code , some applications of this method are as follows:

- Dining Philosopher's Problem[5].
- Producer Consumer Problem[5].
- Barber Shop Problem[5].
- River Crossing Problem[5].
- Baboon Crossing Problem[5].

These are some numerous problem , where this one process code fit to solving this. This one process code for process synchronization gives a better results to solving this problem with better time complexities.

5 FUTURISTIC SCOPE

As we all known that a Research is never ending process , we can't bound Research work, In the method of this (Process Synchronization using one process code) , various exploration area are still existing here and , In this literature , we discussed some futuristic exploration area or futuristic scope of this method.

- **Using one Variable:** In all the Previous Process Synchronization code like (Peterson's Algorithm and Semaphores) all algorithms works with two variables to synchronize the code , even our novel method also works with two variables , so we achieve reliability but not to flexibility. To achieve flexibility we require that we developed or do some changes in that code that we are able to perform similar actions or similar tasks with one variable and also take concern that when we do this one variable , we preservers the objectivity of Process Synchronization , that is (Mutual Exclusion , Progress and Bounded Waiting).
- **Decide Order of Execution :** Like Semaphores is also capable to decide the order of execution of processes means using Semaphores , we have a capability that we decide the order of execution of that processes , that which process executes first , and which is second , and so on. Suppose Let we have 6 processes name says (P1, P2, P3, P4, P5, P6) and if we using Semaphore we also Synchronize that processes and also we have a ability that we decide the order of execution of that processes , that which comes first and which comes second and so on suppose we want to execute process P2 first then we execute process P2 first and then we want to execute Process P6 then we execute process P6 and so on , and also synchronize all processes. So we do this using Semaphores , but using this one process code , we can't do this so this one is a major futuristic scope in this method.

6 CONCLUSION

Operating System play's a major role in each computing contexts , and it is the most significant tool for every computing devices , but in today's era of computing technologies , the computation becomes complex and we face a big hurdle of concurrency. Now to overcome this hurdle the need of hour is we resolve this and made to operating system versatile in today's era. In Operating System Concurrency directly relates to processes , when multiple processes wants to execute same time , then the problem of con currency occur , but many time concurrency is helpful and many time it shows the sign of inconsistencies , but this is the major question regarding operating system , that how we resolve this concurrency , so this one process code for process synchronization is a desirable method , which resolves this and there are many futuristic scope regarding this method , like in one process code we also to do process synchronization , but we can't decide the order of execution using this method, so this is a big futuristic scope in this method and we also optimize the time complexities of many other process synchronization variable , In the previous methods all adopt two separate code for any two processes , but here we write only one code to do synchronize the processes. Hope so there are still many advancements in this code which will done in the near future.

7 ACKNOWLEDGMENTS

I am very much grateful to all respected professors of DIET Rishikeshfor his kind help, lasting encouragement, valuable suggestion throughout the entire period of my project work. I am highly indebted to his astute guidance, sincere support and boosting confidence to make this Dissertation successful.

The acknowledgment will be incomplete if I fail to express my obligation and reverence to my family members and friends whose moral support is great factor in doing this research.

8 REFERENCES

[1] Andrew S. Tanenbaum, "Modern Operating Systems", 2nd Edition, Pearson Education, 2004.

[2] AbrahamSilberschatz , Peter B Galvin , Gerg Gagne, "Operating System Concepts" ,9th Edition , Wiley , 2015.

[3] Gary Nutt, "Operating Systems", 3rd Edition, Pearson Education, 2004.

[4] Harvey M. Deitel, "Operating Systems", 3rd Edition, Pearson Education, 2004.

[5] DhananjayM.DhamDhere, "Operating Systems A Concept – Based Approach", 3rdEdition, McGraw Hill Education (India) Private Limited, New Delhi, 2003.

[6] Silberschatz , "Operating System Principals", 7th Edition, ISBN 13: 9788126509621 , Wiley.