Solution for Mutual Exclusion

Bhawna Ahuja¹, Adit Popli², Krishna Prakash K³ Balachadra Muniyal⁴

¹, ², ³ & ⁴ Department of Information & Communication Technology, Manipal Institute of Technology, Karnataka-576104, INDIA

ABSTRACT

Mutual exclusion makes sure that any given process while in its execution, trying to modify any shared resource prevents the other processes from doing the same. While one process accesses the shared resource, the other processes requiring to access the same will be kept waiting. When the process completes accessing the shared variable, among the other waiting processes, one is selected based on a certain criteria. The proposed algorithm makes sure that there is no possibility of deadlock by using time complexity and a newly defined parameter-timeout. In case of abrupt termination of the process due to unexpected system failure, the recent safe state will be restored. The proposed algorithm ensures that at any given time, shared resource never reaches any inconsistent state.

Keywords: Time Complexity, Aging, Mutual Exclusion, Critical Section, Starvation

1. INTRODUCTION

If we can arrange matters such that no two processes enter into their critical section simultaneously, we can avoid serious problems like race condition, deadlock etc. Certain criteria like age count (a newly defined parameter) and time complexity as explained in [1] are used to choose one among a lot of waiting processes to enter into the critical section [2] when exited by the other process. These two factors help to decide the next process that will enter into critical section. The concept of threads discussed in [3] has been used to depict the same. Aging count is required to ensure that no process is left unattended beyond a predefined age limit.

If the deadlock occurs or the process becomes unresponsive, owing to the slightest of possibility, adequate measures will be taken to roll back the system to a safe state and resume its execution from thereon thus discarding the inconsistent changes.

2. OBJECTIVE

The objective of proposed solution for mutual exclusion is to avoid deadlock and starvation. For this purpose, we define a new term called age count.

Age count: It is a pre-defined constant, which specifies the maximum number of processes that arrive after a given process but can execute before it.

With the use of age count, starvation can be avoided. Also there will never be any possibility of race condition [4] as the entry to critical section is based on time complexity as described in [5] and aging combined together. The algorithm also ensures that there is no busy waiting [6] situation.

As in Dekker’s algorithm for Mutual Exclusion [7], notion of favoured thread is used to resolve the conflict over which thread should execute first. The favoured status alternates between threads. When there is a conflict, one process is favoured and the priority reverses after successful execution. This has been overcome as the favoured process depends on aging and time complexity combined together. Also the busy waiting problem in Petersons Algorithm [8] has been avoided by introducing the key parameters defined below. As the Dekker’s Algorithm and Petersons Algorithm have been quite famous for providing mutual exclusion [9] owing some to its historical importance, they cannot deal with large number of processes. The proposed algorithm can deal with N number of processes and still re-assure no deadlock.

3. ALGORITHM AND ITS IMPLEMENTATION

A mutual exclusion is a program object that prevents simultaneous access to modifiable shared resource. This resource is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one process owns mutex at a time, thus a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access of the resource. Upon releasing the resource, thread unlocks the mutex.
Here, we define five terms that will be used throughout the implementation of our algorithm:

\( a) \) **Threshold age count** (ac):
It is a pre-defined constant, which specifies the maximum number of processes that arrive after a given process but can execute before it. It makes sure that no process is left unattended for a very long time (Prevents Starvation).

\( b) \) **Aging co-efficient** (ace):
It is a pre-defined constant, whose value is to chosen appropriately by the user. It determines the efficient implementation of the proposed algorithm.

\( c) \) **Permissible Waiting Time (P.W.T)**:
It is a relative value which gives the maximum amount of time a process can be asked to wait after its arrival.

\[
P.W.T = \text{ace} \times \text{time complexity} \quad (1)
\]

\( d) \) **Timeout**:
It gives the absolute deadline regarding the time by when the process should start executing latest. This parameter is used for queuing of the processes during job scheduling.

\[
\text{Timeout} = \text{Arrival Time} + \text{P.W.T} \quad (2)
\]

\( e) \) **Label**:
If the age counts value of a process has reached ac. It is marked as true, otherwise false. If process is labelled true, no new process can be placed before it in the queue.

Another important parameter in the proposed solution is Time complexity. It is defined as the amount required by a particular process to complete its execution [5]. Time complexity is independent of the programming language used, does not depend on the speed of computer and the quality of compiler. Hence it acts as an important factor in making decision to grant permission to a process to enter into its critical section. There are many methods to find time complexity of a process. Major operation count, Steps per execution and Global variable count are the methods that can be used to find the time complexity as explained in [5].

\[
\text{ALGORITHM}
\]

- Let **threshold age count** variable value = x
- Let **aging co-efficient** variable value = y
- Let **time_complexity** array store the respective time complexity values of all the processes.
- Let **aging** array store the respective age count values of all the processes.
- Let **label** array store the respective labels of all the processes.

**STEP 1 : Initialization**

At system initialization, all the processes ready for execution are monitored and their timeout values are calculated based on the respective time complexities and the y and are put in a queue, in the ascending order of their timeout values and their aging values are set to 0 and labels to false.

**STEP 2 : Process Queue Management**

The three tasks that are simultaneously being performed are as follows:
a) All the new incoming processes are monitored and are put in the queue, at the appropriate position based on their timeout values and the labels of other processes.

b) Aging values of all the processes which arrived before the new incoming process but are placed after it in the queue are incremented by 1.

c) If the aging value of any process reaches ac, its label is marked as true.

STEP 3 : Backup of the shared data

The values of the shared variables used in the process being dispatched for execution (at the head of the queue) are pushed onto the top of the stack.

STEP 4 : Verification of the process completion

After waiting for an interval equal to the ‘time complexity’ of the selected process, the completion of the process is verified.

One of the following two tasks is performed:

a) If the process is complete, it has released all the resources and its traces are deleted from the system.

b) If the process is still running, it’s forcibly stopped and the system is restored to the previous safe state [10] by popping the values of the shared resources from the stack and the remaining traces of the process are deleted from the system.

STEP 5 : New process selection for execution

Go to STEP 3

Note: Here an assumption has been made that since time complexity of a process and its CPU burst can be related, we express the time complexity as a measure of the CPU burst

Firstly the incoming processes are entered into an array list and upon their arrival; time complexity of each is calculated and entered into an array using any one of the previously listed methods. Another corresponding array is made which contains the age counts corresponding to each process and is incremented whenever a new process arrives, for all the processes that arrived earlier but are present after the new process in the queue and if its age count value reaches ac, its label is marked as true. The initial age count value of a process, on its arrival is initialized to zero and its label is set to false. A threshold aging value is predefined. Whenever, the age count value of a process exceeds the threshold, its given priority over all the processes which arrive later.

When the CPU is idle, the process at the head of the queue may enter into its critical section by pushing onto stack the contents of shared variable i.e. its recent safe state as already stated in [10], a counter is maintained to run the given process till its time complexity. After it has taken the amount of time specified in time complexity array, it checks to determine if the process has finished its task or was aborted due to certain system inconsistencies.

If the process has completed its execution within time, that process is removed from the array list so as to reduce the space occupied. However, if the process was not completed, it is forcibly aborted and it becomes necessary to restore the system back to its previous safe state by popping the contents from the stack. After the rollback has been performed safely, the contents from stack or queue [11] should be deleted and that process is removed from the list of processes. Same process is repeated till all the processes waiting to execute on a shared variable have been executed.

In the given algorithm following classes and array lists are used:

**Figure 1 :** Testing data-load current (amperes)
I) Global class:
This class contains the variable ‘decide’ which is explicitly used to kill the ongoing process by setting it to false.

II) MonitorProcesses class:
It creates a thread that monitors any new incoming process and inserts it into the arrays at the appropriate position.

III) ModifyAge class:
It creates a thread that modifies the age count values of the appropriate processes after a new process arrives.

IV) Process and Solution classes:
It is used to spawn the parent and child threads respectively, for executing a process’ critical section and stopping the execution. The Process Class starts a thread which in turn creates an object of the class solution, which results in a child thread, which is further responsible for the critical section execution of a process. After a particular amount of time, the parent thread kills itself, which in turn leads to termination of the child thread.

V) Compare_age class:
It compares the age count value of all the processes with the constant threshold value and if it is equal, it marks the label of the process as true.

VI) Threaddemo class:
It is the class containing the ‘main’ function, which is responsible for regulating the flow of control of the entire program. It contains the main thread of execution.

VII) Array Lists:

a) Name:
   It contains the name of the process at the i<sup>th</sup> position.

b) Complexity:
   It contains the time complexity of the process at the i<sup>th</sup> position.

c) Aging:
   It contains the ‘aging factor’ value of the process at the i<sup>th</sup> position.

d) Label:
   It contains the label values of the process at i<sup>th</sup> position.

VIII) Array Lists:

e) Name:
   It contains the name of the process at the i<sup>th</sup> position.

f) Complexity:
   It contains the time complexity of the process at the i<sup>th</sup> position.

g) Aging:
   It contains the ‘aging factor’ value of the process at the i<sup>th</sup> position.

h) Label:
   It contains the label values of the process at i<sup>th</sup> position.

4. DEMONSTRATION OF THE PROPOSED ALGORITHM USING SAMPLE DATA
To develop a better understanding of the algorithm discussed above, let us consider an example.
Let us consider a system, which has two processes on start-up - A and B. As the problem develops, the following turn of events take place:

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W. T</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Since the Timeout (B) < Timeout (A), B is placed before A in the queue.

CPU: B
Table 2: 2 units: Process C arrives

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W.</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>2</td>
<td>0.6</td>
<td>2.6</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Process C is placed above A because:
Timeout (C) < Timeout (A),

CPU: B

---

Table 3: 10 units: Process B completes

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W.</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>2</td>
<td>0.6</td>
<td>2.6</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Process B completes within its time complexity period and CPU is given to the next process in the queue.

CPU: C

---

Table 4: 12 units: Process C completes

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W.</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>2</td>
<td>2</td>
<td>0.6</td>
<td>2.6</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Process C completes within its time complexity period and CPU is given to the next process in the queue.

CPU: A

---

Table 5: 15 units: Process D and E arrive

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W.</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>8</td>
<td>2.4</td>
<td>17.4</td>
</tr>
<tr>
<td>D</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>18.0</td>
</tr>
</tbody>
</table>

Process D and E arrive and E is placed before D in the queue because:
Timeout (E) < Timeout (D)

CPU: A
Table 6: 32 units: Process A completes

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W. T</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>8</td>
<td>2.4</td>
<td>17.4</td>
</tr>
<tr>
<td>D</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>18.0</td>
</tr>
</tbody>
</table>

Process A completes within its time complexity period and CPU is given to the next process in the queue.

CPU: E

Table 7: 40 units: Process E completes

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W. T</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>8</td>
<td>15</td>
<td>2.4</td>
<td>17.4</td>
</tr>
<tr>
<td>D</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>18.0</td>
</tr>
</tbody>
</table>

Process E completes within its time complexity period and CPU is given to the next process in the queue.

CPU: D

Table 8: 50 units: Process D completes

<table>
<thead>
<tr>
<th>Name</th>
<th>A. Time</th>
<th>Complexity</th>
<th>P.W. T</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>18.0</td>
</tr>
</tbody>
</table>

Process D completes within its time complexity period and the CPU becomes idle.

CPU: -

5. Result

The efficiency of the suggested algorithm is shown in terms of calculations based on scheduling algorithms. The algorithm above implemented on a number of sample data and average waiting time for the sample was recorded using First Come First Serve (FCFS) Method, Shortest Job First Method (SJF) and the proposed algorithm, taking into consideration CPU Burst time, which is expressed in terms of time complexity. Also time complexity and CPU burst time can be related as it gives approximation and exact time required to execute a process respectively.

The recorded data highlights the fact that value of the average waiting time for a process calculated using the proposed algorithm lies between the one calculated using FCFS and SJF.

The same result is verified using data sample shown in Table 9 below.

Table 9: Sample Data

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival time</th>
<th>CPU Burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>
Table 10: Using Aging coefficient = 0.6

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival time</th>
<th>CPU Burst time</th>
<th>Max. Waiting time</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>20</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>10</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>5</td>
<td>3.0</td>
<td>8.0</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td>2</td>
<td>1.2</td>
<td>11.2</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td>7</td>
<td>4.2</td>
<td>16.2</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>6</td>
<td>3.6</td>
<td>18.6</td>
</tr>
</tbody>
</table>

Permissible Waiting Time = Ace * Time Complexity

Timeout = Permissible Waiting Time + Arrival Time

Here, six processes are taken into consideration. Only processes A and B arrive at time interval 0. Since the CPU burst of process B is lesser than A, B gets the CPU. By the time process B completes its execution at time interval 10, processes C and D have also arrived. Out of the three pending processes (A, C and D), C gets the CPU as its timeout period has expired. C completes its execution at 15, and by then all other processes have arrived. But, since the deadline (timeout) for processes D and A have expired at 11.2 and 12.0 respectively, they will be served respectively. Processes E and F continue their execution thereafter.

Table 11: Resultant Values using the three algorithms.

<table>
<thead>
<tr>
<th></th>
<th>FCS</th>
<th>SJF</th>
<th>Proposed Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Waiting Time</td>
<td>20.66</td>
<td>8.33</td>
<td>13.50</td>
</tr>
</tbody>
</table>

Calculations for Mean Waiting Time for the three algorithms are shown below:

**FCFS:**

0-20 A
20-30 B
30-35 C
35-37 D
37-44 E
44-50 F

Mean Waiting Time= \((0 + 20 + 25 + 25 + 29)/6 = 20.66\)

**SJF:**

0-10 B
10-12 D
12-17 C
17-23 E
23-30 F
30-50 A

Mean Waiting Time= \((30 + 0 + 7 + 0 + 11 + 2)/6 = 8.33\)

**Proposed Algorithm:**

Aging Coefficient=0.6

0-10 B
10-15 C
15-17 D
17-37 A
37-44 E
44-50 F

Mean Waiting Time= \((17 + 0 + 5 + 5 + 25 + 29)/6 = 13.5\)
Relation between Aging Coefficient and Average Waiting Time:

Similar calculations were done to find the average waiting time for the given set of processes using different aging coefficient values and the results were recorded as in fig 11.

Table 12: Resultant values (using different values of Aging Coefficient)

<table>
<thead>
<tr>
<th>Aging Coefficient</th>
<th>FCFS</th>
<th>SJF</th>
<th>Proposed Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>20.66</td>
<td>8.33</td>
<td>19.00</td>
</tr>
<tr>
<td>0.6</td>
<td>20.66</td>
<td>8.33</td>
<td>13.50</td>
</tr>
<tr>
<td>1.2</td>
<td>20.66</td>
<td>8.33</td>
<td>8.33</td>
</tr>
</tbody>
</table>

Here, it is evident that whatever be the value of the aging coefficient, the performance (expressed in terms of average waiting time) of proposed algorithm lies between First Come First Serve and Shortest Job First. However, if we increase the value of aging coefficient to a certain maximum value, it performs similar as Shortest Job First.

6. CONCLUSION

With the basic idea of avoiding starvation and busy waiting phenomenon and to guarantee no deadlock occurrence, it supports the proposed solution for mutual exclusion. The threshold aging factor keeps a check that no starvation occurs i.e. no process is left unattended for a long time. Time complexity acts as a measuring quantity to enter into the critical section. The parameter time complexity is useful as it helps to rollback and goes back to previous safe state if any problem occurs. It also overcomes quite a few challenges faced by Dekker’s Algorithm and Peterson’s Algorithm and aging coefficient has been defined for the same. The brief comparison has been done above to bring out the advantages of proposed algorithm.

References
[7.] Teodor Reos, Daniela Rus, Execution Support Environment, Volume 2