SOFTWARE SOLUTIONS FOR HANDLING CONCURRENT PROCESSES

Seema Moudgil*

Isha Sharma**

ABSTRACT

Interprocess synchronization is necessary to prevent timing errors due to concurrent accessing of shared resources, such as data structures or I/O devices, by contending processes. Without adequate interprocess synchronization, updating of shared variables can induce concurrency related timing errors that are often difficult to debug. This paper is offered as a guide to the principles and practice of concurrent processes in operating system design. It introduces various software solutions to handle concurrent programming in a mutually exclusive manner which can be accomplished by allowing at most at a time to enter the critical section of code within which a particular shared variable or a data structure is updated. Special attention is paid to the Dutch mathematician Dekker's and Dijkstra's solutions. The paper concludes with an analysis of programmed solutions satisfying mutual exclusion requirements. It is assumed that the reader is familiar with the basic principles of computer architecture and data structures and is conversant with programming in a high level language.

Keywords: *Interprocess Synchronization, Concurrent process, Dekker's solution, Semaphore, Operating system*.

*Assistant Professor, MAIMT, Jagadhri, Haryana, India.

**Assistant Professor, MAIMT, Jagadhri, Haryana, India.

INTRODUCTION

A process is a "little bug" that crawls around on the program executing the instructions it sees there. Normally (in so-called *sequential* programs) there is exactly one process per program, but in *concurrent* programs, there may be several processes executing the same program. Cooperating processes can affect or be affected by other processes executing in the system. These processes can either directly share a logical address space (i.e. both code and data) or be allowed to share data only through files or messages. Concurrent access to shared data may result in data inconsistency. Interprocess synchronization is necessary to prevent timing errors due to concurrent accessing of shared resources, such as data structures or I/O devices, by contending processes. Without adequate interprocess synchronization, updating of shared variables can induce concurrency related timing errors that are often difficult to debug.

There exists hardware as well as software solutions to the problem. This paper offers the software solutions as a guide to the principles and practice of concurrent processes in operating system design. These preliminary software solutions can be used to handle concurrent programming in a mutually exclusive manner which can be accomplished by allowing at most one process at a time to enter the critical section of code.

CRITICAL SECTION

A section of code, common to 'n' cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

Entry Section: Code requesting entry into the critical section.

Critical Section: Code in which only one process can execute at any one time.

Exit Section: The end of the critical section, releasing or allowing others in.

Remainder Section: Rest of the code AFTER the critical section.

RULES TO BE ENFORCED BY CRITICAL SECTION

The critical section must enforce all three of the following rules:

Rule 1: Mutual Exclusion

Definition 1: It refers to the problem of ensuring that no two processes or threads (henceforth referred to only as processes) can be in their critical section at the same time.

Definition 2: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

Definition 3: No more than one process can execute in its critical section at one time.

Rule 2: Progress

If no one is in the critical section and someone wants in, then those processes not in their remainder section must be able to decide in a finite time that should go in.

Rule 3: Bounded Wait

All requesters must eventually be let into the critical section.

SOLUTIONS TO CRITICAL-SECTION PROBLEM

1. Mutual Exclusion requirements must be fulfilled.

2. Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

MUTUAL EXCLUSION REQUIREMENTS

1. Ensure mutual exclusion between processes accessing protected shared resource.
2. Make no assumption about relative speeds and priorities of contending processes.
3. Guarantee that crashing or terminating of any process outside of its critical section does not affect the ability of other contending processes to access shared resource.
4. When more than one process wishes to enter critical section, grant entrance to one of them in finite time.

IMPLEMENTING MUTUAL EXCLUSION PRIMITIVES / SOLUTIONS FOR CONCURRENT PROCESSES

Here the solutions are being described by The First Algorithm, The Second Algorithm, The Third Algorithm and Dekker's Solution.

THE FIRST ALGORITHM

Pseudocode

```
program/ module mutex1;
…
type
        who = (proc1,proc2);
var
        turn:=who;
process p1;
```

```
        begin
                while true do
                begin
                        while turn=proc2 do {keeptesting}
                        critical_section;
                        turn:=proc2;
                        other_p1_processing;
                end
        end;
process p2
        begin
                while true do
                begin
                        while turn=proc1 do {keeptesting}
                        critical_section;
                        turn:=proc1;
                        other_p2_processing;
                end
        end;
{parent process}
begin
        turn:= … ;
        initiate p1,p2;
end
```

Description: This algorithm uses turn variable to give turn to processes one by one. Therefore, in any case, one process crashes, the algorithm is in loop.

Here TURN is a global variable, used to control access to an unspecified shared resource. It can assume only two values: proc1, proc2, to indicate the identity of the process allowed entering its critical section. Each process refrains from entering the critical section when the turn belongs to the other.

The comment, KEEPTESTING, in the null statement wait loop suggests that the test is repeatedly executed, and no other action is taken until variable TURN assumes the value proc1. This tight loop represents the negotiation before use of the shared resource. When it completes its critical section, process P1 sets TURN to proc2 and thus allows process P2 to use the resource next. The code for process P2 is symmetrical, so P2 follows the same protocol in acquiring and releasing the resource.

Analysis: The important question is whether algorithm satisfies mutual exclusion requirements. Answer is affirmative. First is fulfilled. Second, this solution strict turn-taking between two processes, starting with turn = proc1, two processes can only execute in p1, p2, p1, p2, p1, p2, … So, it is a problem. Third, when either of the processes crashes or terminates outside of its critical section, solution blocks further execution of other process which busily loops in vain. So, it is a problem. But, if a process crashes within the critical section itself, there is no easy way of knowing whether the guarded shared resource is left in a consistent state. Thus, algorithm is not satisfying all the requirements of mutual exclusion.

THE SECOND ALGORITHM

Pseudocode

```
program/module mutex2;
…
var
        p1using, p2using : boolean;
process p1;
        begin
                while true do
                begin
                        while p2using do {keeptesting}
                        p1using := true;
                        critical_section;
                        p1using := false;
                        other_p1_processing;
                end
        end;
process p2;
        begin
                while true do
                begin
                        while p1using do {keeptesting}
                        p2using := true;
                        critical_section;
                        p2using := false;
                        other_p2_processing;
                end
```

```
        end;
{parent process}
begin
        p1using := false;
        p2using := false;
        initiate p1,p2
end
```

Analysis: This algorithm fulfills the second and third requirement of mutual exclusion. Here both processes get permission to enter in their critical sections when they wish to enter critical section at the same time. However, this fails to satisfy first requirement arbitrary interleaving of competing processes of mutual exclusion. So, the solution is not satisfactory.

THE THIRD ALGORITHM

Pseudocode

```
program/module mutex3;
…
var
        p1using, p2using : boolean;
process p1;
        begin
                while true do
                begin
                        p1using := true;
                        while p2using do {keeptesting}
                        critical_section;
                        p1using := false;
                        other_p1_processing;
                end
        end;
process p2;
        begin
                while true do
                begin
                        p2using := true;
                        while p1using do {keeptesting}
                        critical_section;
                        p2using := false;
```

```
                    other_p2_processing;
                end
        end;
{parent process}
begin
        p1using := false;
        p2using := false;
        initiate p1,p2
end
```

Analysis: This algorithm fulfills first, second and third requirement of mutual exclusion. But it does not fulfill fourth one which is to grant entrance to one process in finite time, when more than one processes wishes to enter in its critical section at the same time. Here both processes loop forever, each waiting for the other to clear the way.

DEKKER'S SOLUTION

Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming. The solution is attributed to Dutch mathematician Th. J. Dekker by Edsger W. Dijkstra in his manuscript on cooperating sequential processes.

Pseudocode

```
program/module dekker;
type
        who = (proc1, proc2)
var
        turn := who;
        p1using, p2using : boolean;
process p1;
        begin
                while true do
                begin
                        p1using := true;
                        while p2using do
                                if turn = proc2
                                then
                                begin
                                        p1using :=false;
```

```
                                        while turn = proc2 do {keeptesting}
                                        p1using := true
                                end;
                        critical_section;
                        turn := proc2;
                        p1using := false;
                        other_p1_processing;
                end
        end;
process p2;
        begin
                while true do
                begin
                        p2using := true;
                        while p1using do
                                if turn = proc1
                                then
                                begin
                                        p2using :=false;
                                        while turn = proc1 do {keeptesting}
                                        p2using := true
                                end;
                        critical_section;
                        turn := proc1;
                        p2using := false;
                        other_p2_processing;
                end
        end;
{parent process}
begin
        p1using := false;
        p2using := false;
        turn := proc1;
        initiate p1, p2;
end
```

Analysis: It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naive turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented. His solution satisfies all the four requirements of mutual exclusion. His solution incorporates elements of the solutions attempted in our previous three solutions.

Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation. Let us see why the last property holds.

One advantage of this algorithm is that it doesn't require special Test-and-set (atomic read/modify/write) instructions and is therefore highly portable between languages and machine architectures. One disadvantage is that it is limited to two processes and makes use of busy waiting instead of process suspension. (The use of busy waiting suggests that processes should spend a minimum of time inside the critical section.)

Modern operating systems provide mutual exclusion primitives that are more general and flexible than Dekker's algorithm. However, in the absence of actual contention between the two processes, the entry and exit from critical section is extremely efficient when Dekker's algorithm is used.

Many modern CPUs execute their instructions in an out-of-order fashion; even memory accesses can be reordered. This algorithm won't work on SMP machines equipped with these CPUs without the use of memory barriers.

MUTUAL EXCLUSION USING SEMAPHORES

DIJKSTRA'S proposal of a mechanism for mutual exclusion among an arbitrary number of processes, called a semaphore, has gained wide acceptance and found its way into a number of experimental and commercial operating systems.

Semaphores are a simple but powerful interprocess synchronization mechanism. They satisfy most of the extensive requirements that we have specified for a "good" concurrency-control mechanism, including the mutual exclusion requirements. The semaphore variable (whether it is binary or general) may be accessed and manipulated only by means of the SIGNAL and WAIT operations. The busy-wait implementation of WAIT and SIGNAL operations is as follows:-

wait(s): while not (s>0) do {keeptesting} ;
s := s – 1 ;

signal(s): s := s + 1 ;

Pseudocode

```
program / module smutex;
…
var mutex : semaphore; {binary}
process p1;
begin
        while true do
        begin
                wait (mutex);
                critical_section ;
                signal (mutex);
                other_p1_processing;
        end {while}
end;
process p2;
begin
        while true do
        begin
                wait (mutex);
                critical_section ;
                signal (mutex);
                other_p2_processing;
        end {while}
end;
process p3;
begin
        while true do
        begin
                wait (mutex);
                critical_section ;
                signal (mutex);
                other_p3_processing;
        end {while}
end;
{parent process}
begin
```

mutex := 1 {free}

initiate p1, p2, p3

end

Analysis: A possible scenario of the run-time behavior of the three processes introduced in the above program is shown in the following table1. Here different columns show the activities of each process, the value of the semaphore variable after the actions indicated on the same line are completed, the identity of the process within the critical section, and list of processes attempting to enter the critical section. It is assumed here that before activating the three processes, the parent process of the program initializes the semaphore variable 'mutex' to 1 to indicate the availability of the shared resource. This state of the system is depicted as time M1 in the table.

TIME	PROCESS STATUS / ACTIVITY			MUTEX: 1 = FREE 0 = BUSY	PROCESSES: IN CRITICAL SECTION ; ATTEMPTING TO ENTER
	P1	P2	P3		
M1	-	-	-	1	- ; -
M2	wait (mutex)	wait (mutex)	wait (mutex)	0	- ; P1, P2, P3
M3	critical_section	waiting	waiting	0	P1 ; P2, P3
M4	signal (mutex)	waiting	waiting	1	- ; P2, P3
M5	other_p1_proc	critical_section	waiting	0	P2 ; P3
M6	wait(mutex)	critical_section	waiting	0	P2 ; P3, P1
M7	waiting	signal(mutex)	waiting	1	- ; P3, P1
M8	critical_section	other_p2_proc	waiting	0	P1 ; P3

Table 1: A scenario of execution of program solving mutual exclusion with semaphores

OTHER SOLUTIONS: SYNCHRONIZATION HARDWARE

Hardware support for concurrency control in one form or another is an integral part of virtually all contemporary computer architectures. Some of the available mechanisms, such as the interrupt disable/enable and test-and-set instructions, are suitable for implementation of pessimistic approaches to concurrency control. Others, such as the compare-and-swap instruction, are well suited for optimistic concurrency control. The TS and CS instructions can also function in multiprocessor systems, provided that the indivisible read-modify-write cycle is supported for shared memory.

Many systems provide hardware support for critical section code

(a) Uniprocessors –could disable interrupts

a. Currently running code would execute without preemption
b. Generally too inefficient on multiprocessor systems
c. Operating systems using this not broadly scalable

(b) Modern machines provide special atomic hardware instructions

a. Atomic = non-interruptable
b. Either test memory word and set value
c. Or swap contents of two memory words

CONCLUSION

Some well-known synchronization problems, such as the producers/consumers and readers/writers, con be solved using semaphores. Because of the large number of their incarnations in practice, both problems are routinely used to test and to compare various synchronization mechanisms. There are criticisms for semaphores that they are unstructured and do not support data abstraction. Therefore, the next are some alternative mechanisms that support or enforce more structured forms of interprocess communication and synchronization.

REFERENCES

1. Silberschatz A., Galvin P.B.,and Gagne G., Operating System Concepts, John Wiley & Sons, Inc.,New York.
2. Godbole, A.S. Operating Systems, Tata McGraw-Hill Publishing Company, New Delhi.
3. Deitel, H.M., Operating Systems, Addison- Wesley Publishing Company, New York.
4. Tanenbaum, A.S., Operating System- Design and Implementation, Prentice Hall of India, New Delhi.
5. Milenkovic, Milan, Operating Systems – Concepts and Design, Tata McGraw-Hill Publishing Company Limited, New Delhi.