RESEARCH PAPER ON SOFTWARE SOLUTION OF CRITICAL SECTION PROBLEM

Er.Ankit Gupta, Er. Arpit Gupta, Er. Amit Mishra
ank_mgcgv@yahoo.co.in, arpit_jp@yahoo.co.in, amitmishra.mtech@gmail.com,
Faculty Of Engineering and Technology(IT department), Mahatma Gandhi Chitrakoot Gramodaya Vishwavidyalaya.
Chitrakoot-485780, Satna, Madhya Pradesh, India

Abstract

This theory practically depends on the critical section problem. After studying the overview of CSP, it's seen that there are lots of drawbacks in CSP but most of the different solutions are given by the different authors. But still no one has got the perfect solution to overcome this problem. But from the my point of view, it's considered as "it can get a solution including all four possible conditions like (Mutual exclusion, No-preemption, Bounded waiting and starvation).the best way to go for this approach, it should be completely checked it out of all the instruction, but it's very true no any process can move Parallely in critical section, practically one has to wait finally. According to these conditions it provides a great convenience to prove this problem. No one has proved the CSP till now, it's proved by myself in each case of set instruction and algorithms. It provides a gateway to solve the problem of CSP in coming future.

CSP *Keywords*- Algorithms of critical section problem, used semaphore properties,synchronization of all the process,Different solutions of CSP.

Introduction

In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore. By carefully controlling which variables are modified inside and outside the critical section (usually, by accessing important state only from within), concurrent access to that state is prevented. A critical section is typically used when a multithreaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure a shared resource, for example a printer, can only be accessed by one process at a time. In computers, a critical section routine is an approach to the problem of two or more programs competing for the same resource at the same time. Imagine that two programs want to increment a counter. If both do it at the same time: fetch the operand , increment it, and store back the incremented value, then one of the increments will be lost. On today's processors, the programs can use an atomic read-modify-write instruction, such as fetch-and-op, compare-and-swap, or exchange. On early processors, these instructions did not exist; the problem was to accomplish the incrementing atomically, using only ordinary assembler instructions. The problem was defined and first solved by Edsgar Dijkstra. "Critical section routine" was his name for the code that solved the problem.

```
Do
{
Entry section
Critical section
Exit section
Remainder section
}
While(1);
```

General structure of a typical process pi

The simplest method is to prevent any change of processor

Methodology

control inside the critical section. On uni-processor systems, this can be done by disabling interrupts on entry into the critical section, avoiding system calls that can cause a context switch while inside the section and restoring interrupts to their previous state on exit. Any thread of execution entering any critical section anywhere in the system will, with this implementation, prevent any other thread, including an interrupt, from getting the CPU and therefore from entering any other critical section or, indeed, any code whatsoever, until the original thread leaves its critical section.

This brute-force approach can be improved upon by using semaphores. To enter a critical section, a thread must obtain a semaphore, which it releases on leaving the section. Other threads are prevented from entering the critical section at the same time as the original thread, but are free to gain control of the CPU and execute other code, including other critical sections that are protected by different semaphores.

Some confusion exists in the literature about the relationship between different critical sections in the same program.[citation needed] In general, a resource that must be protected from concurrent access

may be accessed by several pieces of code. Each piece must be guarded by a common semaphore. Is each piece now a critical section or are all the pieces guarded by the same semaphore in aggregate a single critical section? This confusion is evident in definitions of a critical section such as "... a piece of code that can only be executed by one process or thread at a time". This only works if all access to a protected resource is contained in one "piece of code", which requires either the definition of a piece of code or the code itself to be somewhat contrived.

Currency arises in three different contexts:

- Multiple applications – Multiple programs are allowed to dynamically share processing time.
- Structured applications – Some applications can be effectively programmed as a set of concurrent processes.
- Operating system structure – The OS themselves are implemented as set of processes.

Concurrent processes (or threads) often need access to shared data and shared resources.

- Processes use and update shared data such as shared variables, files, and data bases.

Writing must be mutually exclusive to prevent a condition leading to inconsistent data views.

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

1) *solution to Critical-Section Problem*

1. Mutual Exclusion. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
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.

• Assume that each process executes at a nonzero speed.

• No assumption concerning relative speed of the n processes

Algorithm 1

Shared variables:

– var turn: (0..1);

initially turn = 0

– turn = i ⇒Pi can enter its critical section

• Process Pi

repeat

while turn = i do no-op;

critical section

turn := j;

remainder section

until false;

• Satisfies mutual exclusion, but not progress

Algorithm 2

• Shared variables

– var flag: array [0..1] of boolean;

initially flag[0] = flag[1] = false.

– flag[i] = true ⇒Pi ready to enter its critical section

• Process Pi

repeat

flag[i] := true;

while flag[j] do no-op;

critical section

flag[i] := false;

remainder section

until false;

• Does not satisfy the mutual exclusion requirement.

Algorithm 3

• Combined shared variables of algorithms 1 and 2.

• Process Pi

repeat

flag[i] := true;

turn := j;

while (flag[j] and turn=j) do no-op;

critical section

flag[i] := false;

remainder section

until false;

• Meets all three requirements; solves the critical-section

• Before entering its critical section, process receives a number.

Holder of the smallest number enters the critical section.

• If processes Pi and Pj receive the same number, if i < j, then

Pi is served first; else Pj is served first.

• The numbering scheme always generates numbers in

increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm (Cont.)

• Notation <≡ lexicographical order (ticket #, process id #) – (a, b) < (c, d) if a < c or if a = c and b < d – max (a0, an−1) is a number, k, such that k ≥ ai for i = 0,..., n – 1

• Shared after checking all the conditions of critical section it's estimated that, all the four possible conditions is not becoming satisfied, but if we use semaphore requirements, Bakery's algo,Dekker's algo,Peterson algo tell the solution of only in separate conditions using semaphore set instructions. The overall meaning of this objective different types of solution are given but everywhere some part of error is counted so to meet all the four possible conditions one algo is implemented according to above conditions which are given below:

If we assume turn= 1 value is kept false and then again false

main program

{

Var: turn: Boolean

Pi

{

Repeat

Flag[i] =T;

Turn = j;

While flag[j]AND

Turn=j do skip

Critical section

Flag[i]=F;

Until

False

}

Pj

{

Repeat

Flag [j]=T;

Turn = i;

While (flag[i] AND turn = i

Do skip

Critical section

Flag[j] = f;

Until

False

}

}

In this case all four conditions satisisfies.

data

Var choosing: array [0...n−1] of Boolean;

Number: array [0...n−1] of integer;

Data structures are initialized to false and 0, respectively

Bakery Algorithm (Cont.)

Repeat

choosing[i] := true;

Number[i] := max (number [0], number [1], number [n − 1]) +1;

choosing[i] := false;

for j := 0 to n − 1

do begin

*A proper solution to the Critical-Section problem must meet three requirements***:**

Only one thread is allowed in its critical section at a time. That is, execution of critical sections is mutually exclusive.

If there's no thread in its critical section, but some threads are waiting to enter their critical sections, only the waiting threads may participate in deciding who gets to enter first. Ultimately, there must be progress in the resolution and one thread must be allowed to enter.

Threads waiting to enter their critical sections must be allowed to do so in a bounded timeframe. That is, threads have bounded waiting.

This is the basic scaffold for proposed solutions to the Critical-Section problem:

Repeat

Entry section

Critical section

Exit section

Remainder section

Until false;

There are multiple threads that will run the above code. Each will have unique critical sections and remainder sections, but their entry and exit sections will be identical. In Java terms, the entry section is what happens behind the scenes at the top of a synchronized block. The exit section is what happens when a synchronized block ends. The critical section is the code inside the synchronized block, and the remainder section is some arbitrary code that's not synchronized. As you can see, this is just a simplification of cooperating threads, and any solutions to this problem will apply to all other Critical-Section scenarios.

After using the semaphore section finally these factors involves in each steps in spite of solutions too

```
while choosing[j] do no-op;
while number[j] = 0
and (number[j],j) < (number[i], i) do no-op;
end;
Critical section
Number[i] := 0;
Remainder section
Until false;
```

Two Types of Semaphores

• *Counting semaphore* – integer value can range over an Unrestricted domain.

• *Binary semaphore* – integer value can range only between 0 And 1; can be simpler to implement.

Difficulties with Semaphores

Semaphores provide a powerful synchronization tool.

wait(s) and signal(s) are scattered among several processes. Therefore, it is difficult to understand their effects.

Usage must be correct in all the processes.

One bad process (or one programming error) can kill the whole system.

The use of a special machine instruction to enforce has a no. of advantages:

1. it's applicable to any no. of processes on either a single possessors sharing main memory .
2. it's simple and therefore easy to verify.
3. it can be used to support multiple critical sections; each critical section can be defined by it's own variable.

• Can implement a counting semaphore S as a binary Semaphore

Synchronization tool that does not require busy waiting.

• Semaphore S – integer variable

• can only be accessed via two indivisible (atomic) operations

```
Wait(S): while S ≤ 0 do no-op;
S: = S − 1;
Signal(S): S: = S + 1;
```

Fig 1.Implementing Threads in the Kernel

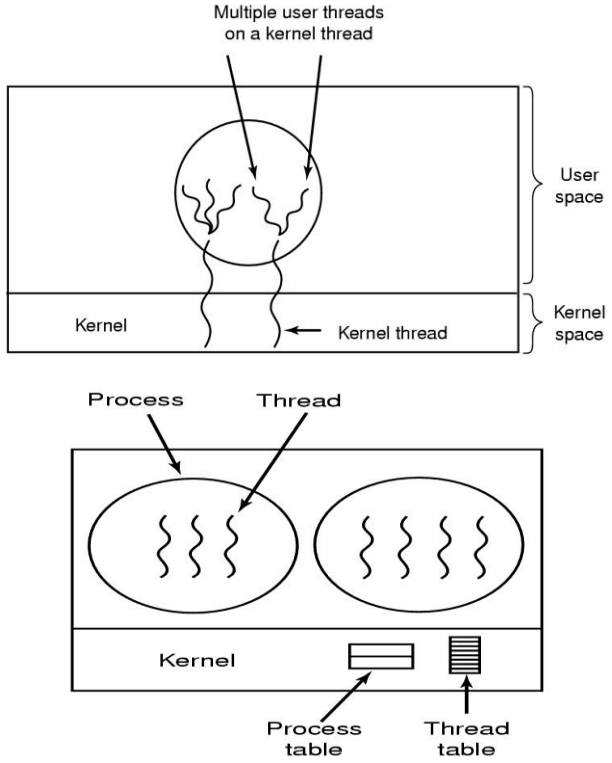

Fig2. Hybrid Implementations

wait(s) and signal(s) are scattered among several processes. Therefore, it is difficult to understand their effects.

Usage must be correct in all the processes.

One bad process (or one programming error) can kill the whole system.

2) ***According to shared variable-***

enum Boolean {false=0; true=1 ;}

Boolean flag [2] = {false, true}

In this way it can be seen that if one process fails outside the CS including the flag setting code then the other process is not blocked. But if process fails inside it's CSP or after setting it's flag[true], just before entering it's CSP then the other problem is permanently blocked. It doesn't guarantee of the mutual exclusion.

The main idea is behind of that the CSP problem can be solved by simply in a uniprocessor environment if we can forbid interrupts to occur while a shared variable is being modified. In this manner,we can be sure that current sequence of instructions will be allowed to execute in oder without preemption. No other instruction will be run, so no unexpected modifications could be made to the shared variable.

Many machines therefore provide special hardware instruction that allow either to test and modify content of a word,or to swap the contents of two words, atomically that is to say as one uninterruptible unit.thus, these special instructions can be used to solve the critical problem in simple manner.

The use of a special machine instruction to enforce has a no. of advantages:

1. it's applicable to any no. of processes on either a single processssors sharing main memory .
2. it's simple and therefore easy to verify.

   it can be used to support multiple critical sections;each critical section can be defined by it's own variable.

Summary and Conclusion

When the multiple processes access, so called the critical section of the memory, they use a spin lock to allow only one of the processes to access the critical section with Test & Set instruction or similar types at a time. But my understanding is that those processes running on the different processors are independent(the processes are made to be independent to run in parallel), which makes me think why they have to access the same memory locations. I would guess it should be to synchronize the processes or something, but cannot think of an actual example.

To be honest, part of the reason that writers have ignored the CRITICAL_SECTION structure is that it's implemented very differently under the two main Win32 code bases: Microsoft® Windows® 95 and Windows NT®. Everyone knows that both code bases have spawned numerous descendents (most recently, Windows Me and Windows XP, respectively), but it's not necessary to list them all here. The point is that now that Windows XP is well established, developers can soon drop support for the Windows 95 line of operating systems. We have done so in this article. The life of a critical section begins when it's passed to InitializeCriticalSection (or more accurately, when its address is passed). Once initialized, your code passes the critical section to the EnterCriticalSection and LeaveCriticalSection APIs. Once a thread returns from EnterCriticalSection, all other threads that call EnterCriticalSection block until the first thread calls LeaveCriticalSection. Finally, when you no longer need the critical section, good coding practice dictates that you pass it to DeleteCriticalSection.

In the ideal case where a critical section is unowned, a call to EnterCriticalSection is very fast because it simply reads and modifies memory locations within user-mode memory. Otherwise (with one exception that we'll get to later), threads that block on a critical section do so efficiently, without burning up extra CPU cycles. Blocked threads wait in kernel mode and aren't schedulable until the critical section owner releases it. If multiple threads are blocked on a critical section, only one thread acquires the critical section when another thread releases it.

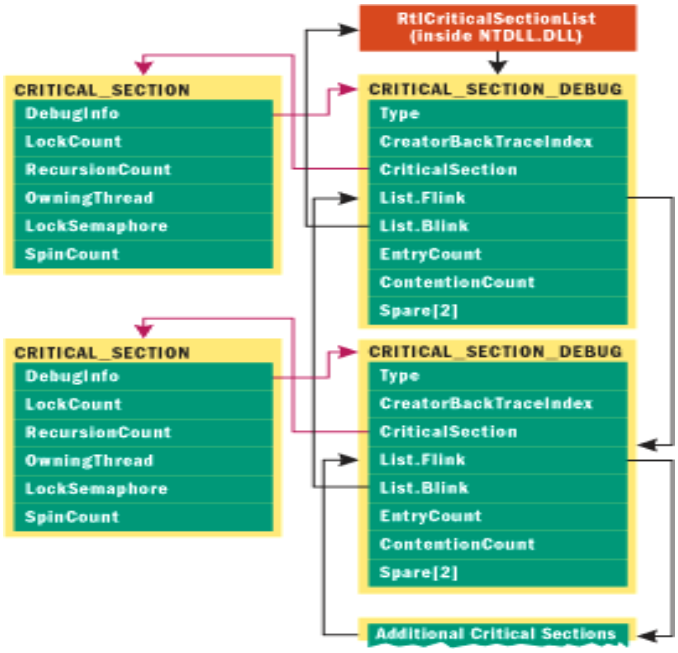

Fig. 3example of a critical section routine

The final case of the critical section problem is:

1) Busy waiting is employed.

2) starvation

3) Deadlock

So it's noticed that different solution are given by for the CSP. But it's concluded that,using semaphore section,bounded buffer solution,producer-consumer problem it's not solved by accurately till now that all the process may run simultaneously. So it's important to know that those properties which are given for the solution of CSP. They are satisfying all the four requirements but in some cases not for all. Therefore CSP is enhanced for one property more,finally one process has to wait,until other process couldn't be released.

References

[1] Howard P. Katseff, ACM New York, NY, USA ©1978

[2] Winter,J. J. ; Breslin,J. T. ; Leupold,H. A., : JUL 1970

[3] PaulMcKenney 15:00, 26 May 2006 (UTC)

[4] Gurudutt 10:16, 10 September 2007 (UTC)

[5] Silberschatz and Galvin c 1998

[6] Dijkstra – The first (1965), Knuth – Presented (1966), Lamport – Presented the Bakery (Baker's) algorithm (1974).

[7] info@semaphoresolutions.ca, Kevin Kotorynski

[8] Jon Emerson)

[9] Michel Raynal: Algorithms for Mutual Exclusion, MIT Press, ISBN 0-262-18119-3

[10] Sunil R. Das, Pradip K. Srimani: Distributed Mutual Exclusion Algorithms, IEEE Computer Society, ISBN 0-8186-3380-8

[11] Thomas W. Christopher, George K. Thiruvathukal: High-Performance Java Platform Computing, Prentice Hall, ISBN 0-13-016164-0

[12] Gadi Taubenfeld, Synchronization Algorithms and Concurrent Programming, Pearson/Prentice Hall, ISBN 0-13-197259-6

[13] Proceedings of the World Congress on Engineering 2010 Vol III

[14] WCE 2010, June 30 - July 2, 2010, London, U.K.

[15] CS425/ECE 428 Distributed Systems, Fall 2009, Instructor: Klara Nahrstedt

[16] Out: Tuesday, September 22, Due Date: Tuesday, October LESLIE LAMPORT

[17] Digitul Equipment Corporution,Pulo Alto, California6,By Paul Krzyzanowski

[18] September 21, 2010 [updated February 17, 2011]

[19] Matt Pietrek and Russ Osterlund,RussOsterlund@adelphia.net,

[20] Abraham Silberschatz and Peter Baer Galvin, Operating System Concepts. John Wiley & Sons, 5th Edition, 1999.

[21] Some parts are taken from GATE-2011 computer science and engineering.

[22] Shivani engineering question bank of O.S. 4th and 6th revised