ON THE GENERALIZED CRITICAL REGION CONSTRUCT

Krishna Kant[1]
Abraham Silberschatz[2]

TR-151 July 1980

[1] Department of Mathematical Sciences, University of Texas at Dallas, Richardson, Texas 75080.

[2] Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712.

This research was supported in part by the National Science Foundation under grant no. MCS-7702463 and in part by the Office of Naval Research under contract no. N00014-80-C-0592.

Abstract

In his paper "Implementation of a Generalized Critical Region Construct", Ford has proposed a modification to Brinch Hansen and Hoare's concept of conditional critical regions. In this paper we point out some of the problems with his proposal and provide possible solutions to them.

1. INTRODUCTION

In a recent paper [1], Ford has proposed a modification to the concept of conditional critical regions of Brinch Hansen [2] and Hoare [3]. The most striking points of difference are--

(1) The same shared variable is allowed to be used in more than one critical regions.

(2) No automatic mutual exclusion is enforced for entry to a critical region and the processes can not wait upon arbitrary conditions at arbitrary points within a critical region. Rather, the synchronization is controlled entirely by the "entry conditions" specified in the critical region declarations.

(3) The proposal uses Dijkstra's idea of a secretary process [4] for handling the synchronization. The code for the secretary process is automatically generated by the compiler using the critical region declarations. There is a single secretary for all the processes of a concurrent program. All the processes communicate with the secretary by means of messages.

The critical regions are declared using the following syntax:

```
   <region name>: shared (<shared var list>) cond <entry condition>
```

A process can operate on the shared variables of a critical region using the statement

```
   critical <region name> do <statements>
```

which is translated by the compiler into the following sequence of actions:

(1) Send the request message to the secretary and wait for the permission to continue.

(2) Execute the <statements>.

(3) Send the release message to the secretary.

The secretary maintains two counters A(i) and W(i) for each region i. A(i) specifies the number of processes executing within region i and W(i) specifies the number of processes waiting to enter region i. The entry conditions are constrained to be boolean expressions over linear relations involving A and W counters and constants only.

Whereas the idea of separating out the computation and synchronization aspects is very attractive, the proposal has a number of shortcomings that we would like to point out and propose some solutions.

2. MUTUAL EXCLUSION

Critical regions were originally designed to guard against the time dependent errors by ensuring that a shared variable can not be accessed simultaneously by more than one process [3]. This is not guarenteed by Ford's construct since the same shared variable can be used in more than one critical regions. Thus, in order to ensure the mutual exclusion of a shared variable "v", it is required that the condition A(r)=0 hold upon entry to every region r using the variable "v". In general this is very difficult to ensure. First, such a check requires determining whether an entry condition logically implies the condition A(r)=0, a problem which is likely to be worse than NP-hard, perhaps undecidable. Second, the region declaration does not carry any information on the nesting of the regions, therefore, the whole program must be consulted in order to set up the implications to be checked. It is important to note that this verification is not needed just for avoiding some redundant terms; merely adding the assertion A(r)=0 to the entry conditions of all the relevant critical regions can lead to deadlock whenever nested regions are involved.

Here we present an alternate notation for the region declarations which makes the informal proof of the mutual exclusion easier. The regions using a common set of shared variables* will be put together as a group and called regions whereas their components will be called subregions. We shall require explicit declaration of the allowed nesting of the critical regions. We also include the facility for initializing the shared variables of a region in order to collect all declarative information for a region at one place. The region declaration now assumes the following syntax:

```
 <region name>: region (<shared var list>) =
                            <subregion1>: cond <entry condition1>
                                  .
                                  .
                            <subregionk>: cond <entry conditionk>
                       nesting={ (<subregioni1>,<subregioni2>,... );}
                       [initialization: { <shared var>:= <value>;}  ]
                   end;
```

*The explicit counters introduced by Ford in section VII of his paper are also allowed and will be treated like other shared variables.

The shared variables declared within a region can only be accessed within the subregions associated with that region. The entry conditions can reference the activation and wait counters (henceforth referred to as the implicit counters) and the explicit counters, if any, of the parent region only. A process can execute a critical region using the following syntax:

critical <region-name>.<subregion-name> do <statements>

The nesting clause specifies allowable nesting of the subregions as ordered lists of the subregion names where the first element of each list specifies the outermost subregion. As a special case, this declaration also specifies the subregions which are not nested within any other subregion. These lists must be mutually consistent i.e. no two subregions may be nested within each other. A process can use a subregion only in ways permitted by these declarations.

As an example of the usefulness of these constructs consider the final version of the reader-writer problem given in section IV of Ford's paper. This can be written as follows in our notation:

```
Readwrite: region(database)=
                readpriority:  cond  W(write)=0 ∧ A(write)>0
                read:  cond  A(write)=0
                write:  cond  A(write)=0 ∧ A(readpriority)=0
             nesting: (readpriority,read); (write);
   end;
```

A reader process accesses the database by executing--

```
Critical Readwrite.readpriority do;
         Critical Readwrite.read do; "read from database" end;
                                 end;
```

A writer process accesses the database by executing--

```
Critical Readwrite.write do; "write into database" end;
```

We note the following advantages of this notation over the one presented by Ford-

(1) All the interdependent critical regions appear at one place thus making the declarations more modular and less error-prone.

(2) By examining the nesting clause, one can immediately conclude that the assertion A(write)=0 is not required in the entry condition of the subregion "readpriority" for ensuring mutual exclusion for the writer. One can also conclude that the assertion A(readpriority)=0 must not be implied by the entry condition of the "read" subregion otherwise deadlock will occur.

(3) The compiler can easily verify if the program actually conforms to the nesting declarations and thus aviod erroneus synchronization resulting from improper nesting. For example, in the program above, the compiler can make sure that the "read" subregion is not used outside the scope of the subregion "readpriority".

3. SCHEDULING

In section IV of the paper, Ford states that "a simple way to add high level scheduling capability to the construct is to permit entry conditions to reference these wait counters.... This obviates the need for artificial numerical priorities". These claims are misleading since the only mechanism available for scheduling is the use of wait counts in the entry conditions; this is inadequate for implementing even a simple FIFO scheduler. In the absence of any other constructs, the only possible form of scheduling is where all the processes have statically assigned priorities and there is a separate critical region for each priority class. All the examples cited in Ford's paper posses these characteristics.

Introducing generalized scheduling capability to the construct requires the use of local process data in the entry conditions. Unfortunately, such a generalization will make the implementation very expensive. Here we propose a more restricted construct for specifying scheduling and show that it is adequate for solving many common scheduling problems.

The critical region invocation will be allowed to specify a priority expression whose value is sent as a part of the "request" message to the secretary. The secretary maintains the wait queue for a region in the nondecreasing order of these priority values. When the wait condition becomes true the waiting processes are repeatedly woken up starting at the head of the queue until either the wait condition becomes false or there are no more waiting processes. This idea is similar to the "scheduled waits" of Hoare [5]. We adopt the following syntax for specifying the critical region invocations:

critical <region-name> (priority=<expression>) do <statements>

Where the <expression> does not use any shared variables and is free of any side effects. The first restriction is necessary so as to avoid any concurrent access to the shared variables as a result of the priority evaluation.

Now we present two examples of scheduling- A shortest-job-next (SJN) scheduler and a simple disk head scheduler [6]. We shall use our notation in the latter example.

3.1 Shortest Job Next Scheduler:

```
SJN: region (resource) cond A(SJN)=0
```

A user process uses this scheduler by specifying the required service time T as the priority--

```
Critical SJN (Priority= T) do; ....  end;
```

3.2 Disk head Scheduler:

In this example we shall use an explicit counter "count". Note that the nesting clause does not permit any nested use of the subregions.

```
Disk: region (headpos: cylinder; count,pass: integer) =
                Request: cond  A(Request)=0 ∧ A(Acquire)=0
                Acquire: cond  Count=0 ∧ A(Request)=0 ∧ A(Acquire)=0
                Release: cond  Count>0
            nesting= (Request);(Acquire);(Release);
            initialization:  pass:=0; count:=0;
      end;
```

The code for a typical user process looks as follows--

```
dest,cyl: cylinder; mypass: integer;
"Get disk for use"
Critical Disk.Request  do;
   If dest>= headpos then do; mypass:= pass; cyl:= dest; end;
                      else do; mypass:= pass+1; cyl:= cylmax-dest; end;
                       end;
Critical Disk.Acquire (Priority= (mypass,cyl))  do;
                     headpos:= dest; pass:= mypass;
                     Increment(count);         end;

"Use and release disk"

Critical Release do; "Use disk....";  Decrement(count); end;
```

The variable "count" gives the total number of users simultaneously using the disk. The program above does not take care of the eventual overflow of the variable "pass". This deficiency can be easily fixed.

The priority in this example is a 2-tuple with ordering assumed to be the usual lexicographic ordering on its components.

4. IMPLEMENTATION

The implementation of [request,i] as proposed in section V of Ford's paper is incorrect because W(i) is incremented before checking if Γ_i is true. As a result, while Γ_i is being evaluated, W(i) can never be 0 and none of the reader-writer solutions will work. The problem can be fixed by using the solution given under "simplification 1" in section VI.

Ford attempted to cut down unnecessary boolean expression evaluations by statically determining the entry conditions which can not become true when a counter changes. Here we show that a more detailed analysis allows further improvements.

As shown in Ford's paper, the entry condition Γ_j for region j can be expressed as- Γ_j = Bj(Rj1,Rj2,... ,Rjm) where Bj is a boolean expression over the relations Rj1,Rj2,... ,Rjm and uses the operators $\wedge$ and $\vee$ only. Let all the implicit and explicit counters relevant for a critical region (using our notation for region declarations) be denoted as X(1),X(2),... X(n). Then the relations Rjk [k=1..m] can be written as follows:

$$R_{jk} = \left[\sum_{i=1}^{n} a_{ijk} X(i)\right] > C_{jk}$$

Now we analyze the effect of incrementing X(i)'s on the truth value of the entry conditions. The arguments for the case of decrementing X(i)'s are very similar and are ignored.

We assume that Bj has been put in the disjunctive normal form i.e., $Bj = \bigvee_{K=1}^{m} (Rjk1 \wedge Rjk2 \wedge \ldots \wedge Rjkl_K)$. Clearly, Bj is false if and only if all the disjuncts are false. Consider some disjunct $Rjk1 \wedge Rjk2 \wedge \ldots \wedge Rjkl_K$. Each of Rjkn,$[n=1..l_K]$ is a linear equation with X(i)'s as "unknowns". We also know that all X(i)'s must be nonnegative and all the Rjkn's must hold if this disjunct could possibly make Bj true. Thus we have a linear programming like problem at hand. (There is no optimization to be performed however, we are only interested in the feasible region defined by the constraints.) Now consider a specific X(i). We shall examine Bj one disjunct at a time and keep only those which could possibly become true as a result of incrementing X(i).

(1) If X(i) does not occur anywhere in a disjunct, it can be ignored.

(2) If a disjunct does not define any feasible region, it can be ignored.

(3) If by increasing X(i) we can never crossover from infeasible to the feasible region, then the corresponding disjunct can be ignored. This happens if and only if the coefficient of X(i) term in every constraint Rjkn, is zero or negative.

(4) If the hyperplane corresponding to a constraint Rjkn does not form the boundary of the feasible region, the constraint is redundant and can be deleted from the disjunct. See fig 1 for a 2-D case-

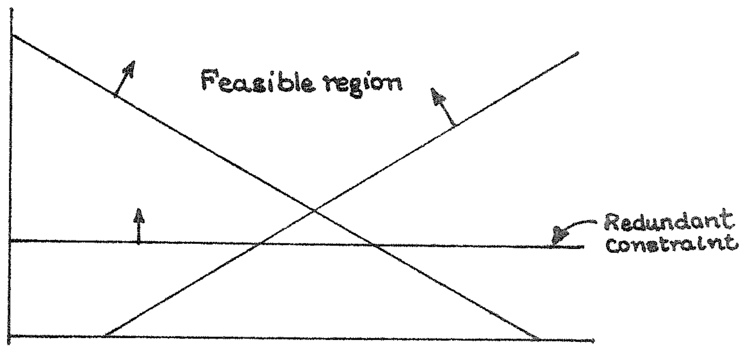

Fig 1

If after applying these tests, Bj still consists of more than one disjunct it might be useful to (a) Consider Bj as a boolean expression and apply the standard minimization techniques and to (b) Arrange the disjuncts in the decreasing "likelihood" of becoming true based upon some heuristics.

In the above, the atomic terms of the boolean expressions were assumed to be arbitrary linear relations. However, all the examples we have seen in the literature can be programmed with atomic terms containing just one implicit or explicit counter. In this case each Rjkn will be of the form x>a or -x>a where x is a counter and we can apply the above mentioned tests very easily. Let us again consider the case of incrementing certain χ(i). We can claim the following-

(1) If a disjunct contains multiple occurrences of a counter x then all these occurrences can be reduced to atmost two occurrences in O(n) time by using the rules of integer arithmetic and boolean algebra. For example, the terms x>0 and x>2 will reduce to a single term x>2. Similarly,the terms like -x>-1 and x>2 represent an infeasible case and the disjunct containing them can be ignored.

(2) After applying the simlification (1) above and the aforementioned tests with respect to counter x we are left with only one interesting

case. That is, the disjuct contains two x terms x>a, -x>-b and a<b. This implies that the disjunct is relevant with respect to incrmenting x only when x<=a (and with respect to decrementing x when x>=b). Therefore, if the term x>a (-x>-b) is evaluated first, some unnecessary evaluation can be avoided.

5. OTHER ISSUES

Ford suggests a single secretary for all the critical regions of a concurrent program. This not only makes the secretary the synchronization bottleneck, it also means that the whole program will crash if the secretary fails. It would be more reasonable to have one secretary for each set of critical regions operating on common variables.The syntax introduced in section 2 of this paper is particularly suited for doing this.

Limiting the atomic terms in the entry conditions to linear relations involving implicit counters and constants only, is indeed highly restrictive. To overcome this difficulty, Ford has introduced explicit counters in section VII of his paper. Explicit counters can be used to keep track of arbitrary information about the total state of a concurrent program and thus can be used for solving a variety of synchronization problems (See Gerber [7]). However, their implementation as proposed by Ford has a a number of weaknesses:
(1) All the processes of a concurrent program must use the same step-size for incrementing/decrementing the explicit counters. Clearly, the solution is to pass the step size as a part of the "request" message to the secretary.

(2) The proposal treats the explicit and implicit counters in the same mannar. This is both inefficient and dangerous. The inefficiency arises because many entry conditions may need to be evaluated every time an explicit counter is changed. It is dangerous because the user has to ensure the consistency of the shared data whenever any explicit counter is changed. These problems can be easily avoided if the explicit counters are treated in a special mannar i.e. the secretary just keeps track of the values of the explicit counters; any entry condition evaluation and signalling is done only when the implicit counters change.

6. CONCLUSIONS

In this paper, we have pointed out some problems and their possible solutions with Ford's proposal of a generalized critical region concept. Though our idea of localizing the region definitions and explicitly declaring region nesting help in informal verification of mutual exclusion and deadlock freedom, time dependent errors can still occur. What is really required is the ability to explicitly declare that a certain region needs mutual exclusion and let the compiler verify if that is the case. Unfortunately, this is at best a NP-hard problem.

The introduction of the idea of "scheduled waits" provides limited scheduling capability to the proposed construct. It remains to be seen if more powerful scheduling capability can incorporated in the construct without making the entry conditions dependent on local data of the user processes.

7. REFERENCES

[1] FORD, W.S.- "Implementation of a generalized critical region construct", IEEE Trans. on Software Engg., Nov 78, pp449-455.

[2] HOARE, C.A.R.- "Towards a theory of parallel programming" Operating Systems Techniques, Academic Press, 1972.

[3] BRINCH HANSEN, P.- "Operating Systems Principles", Prentice Hall, 1973, Chapter 3.

[4] DIJKSTRA, E.W.- "Hiearchical ordering of sequential processes", Acta Informatica, Vol.1, Fac.2, 1971, pp115-138.

[5] HOARE, C.A.R.- "Monitors: An operating system structuring concept", Communications of ACM, Oct 74, pp549-557.

[6] HOWARD, J.H.- "Proving Monitors", Communications of ACM, May 76, pp273-279.

[7] GERBER, A.J.- "Process synchronization using counter variables", O.S. Reviews, ACM-SIGOPS, Oct 77, pp6-17.