A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables

GARY L. PETERSON
The University of Rochester

A new solution to the concurrent programming control (mutual exclusion) problem that is immune to process failures and restarts is presented. The algorithm uses just four values of shared memory per process, which is within one value of the known lower bound. The algorithm is implemented using two binary variables that make it immune to read errors occurring during writes, that is, "flickering bits."

Categories and Subject Descriptors: D.4.1 [Operating Systems]: Process Management—concurrency; multiprocessing/multiprogramming; mutual exclusion; synchronization; D.4.5 [Operating Systems]: Reliability—fault-tolerance; F.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Critical section, shared variables

1. INTRODUCTION

Dijkstra's problem in concurrent programming control [4] has been extensively studied and his original solution improved [3, 5, 7]. Lamport [9] extended the problem to allow processes to fail and to restrict usage of shared variables. Lamport's Bakery Algorithm requires shared variables of unbounded size and is not immune to infinite failures. However, the algorithm is FIFO and allows read errors to occur during writes. Bounded-size solutions that are immune to infinite failures were found by Rivest and Pratt [13], Peterson and Fischer [11, 12], and Katseff [6].

One measure of solutions to the problem is the count of the maximum number of distinct values of shared memory used by any process. Algorithms which satisfy a very strong fairness condition (to be discussed later) require a large number of values; for example, Katseff's algorithm is FIFO but requires $O(2^n)$ shared memory values per process, where n is the number of processes. A Linear Waiting algorithm which uses a constant number of values per process is known

This research was supported in part by National Science Foundation grant MCS 79-02971.

Author's address: Department of Computer Science, The University of Rochester, Rochester, NY 14627.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0164-0925/83/0100-0056 \$00.75

ACM Transactions on Programming Languages and Systems, Vol. 5, No. 1, January 1983, Pages 56-65.

[11], but it is very complex. The algorithm given here is quite simple but does not have these kinds of fairness properties. It uses just four values of shared memory per process. (The lower bound is known to be three [11].) Hence, this solution uses a nearly optimal amount of shared memory without being overly complex.

This algorithm is robust in the sense of its immunity to the two types of errors that Lamport's algorithm also handles. These errors are (1) (unbounded) process failure and restart and (2) the possibility of read errors occurring during the writing of a shared variable (when the algorithm is appropriately modified). In this case, the four values are represented by two binary variables. These bits may "flicker" an arbitrary number of times during a write.

The algorithm is also interesting because it uses parts of two other algorithms. One part, adapted from [1, 8], resolves contention between processes very easily but has lockout. The other part, a kind of clock mechanism from [11], is used to avoid lockout. This illustrates that the techniques may be quite general and useful in a large number of situations.

2. THE CONCURRENT CONTROL PROBLEM

The full definition of the problem can be found in [9, 11, 13] and elsewhere; only a general description is given here. One is presented with n ($n \ge 2$) processes that communicate with each other through shared variables. The code of each process has been divided into two parts: a critical section (hence, the common name for this problem is "the critical section problem") and a noncritical section. All that is known is that, once the critical section is entered, a process will leave it and return to the noncritical section in a finite amount of time. Processes start execution at a specified location in the noncritical part with all of the variables set to initial values. A process may execute its critical section any number of times. Each process is assumed always to be making "finite progress" whenever it is not in its noncritical section. "Finite progress" means that a finite, but possibly unbounded, amount of time occurs between the execution of individual steps of any process's program.

One of the properties of the critical section is that two or more processes simultaneously executing their critical sections must be avoided. For example, suppose each process's critical section is the code that accesses a common file. The integrity of the data may be destroyed if two processes write it at the same time. *Mutual exclusion*, then, is the first property that must be preserved.

To achieve mutual exclusion, the critical section will be surrounded by two sections of code, called the *protocol*, one of which is executed before beginning the critical section (called the *entry protocol*) and the other after leaving it (called the *exit protocol*). Clearly, there are trivial protocols that provide mutual exclusion by simply preventing any one from ever executing its critical section. A more realistic requirement must be added. No *deadlock* or *lockout* is allowed; that is, once a process is in a protocol, it is guaranteed to leave it eventually for any valid execution sequence (i.e., it cannot get "stuck"). (A valid execution sequence is any, possibly infinite, interleaving of instructions of the processes that has the property that a process always performs the next instruction within a finite amount of time, except possibly for halting in the noncritical section.) Such a protocol is called *fair*. Additional fairness requirements might require *Linear*

Waiting (no process can enter its critical section twice while another process is waiting) or FIFO (first-in, first-out; no beginning process can pass an already waiting process).

In addition to the above criteria, others are imposed in order to make the algorithm more *robust*, that is, immune to some significant failures. The first type is that of *process failure*: a process may repeatedly fail and restart. A process fails by leaving the critical section (if it is in it) and resetting all of its variables, shared and local, to their initial values. So global variables cannot be used, since the process associated with a global variable may fail and take the variable with it. This means that no process can, in general, rely on another's variables; therefore, only very simple *message board* variables will be allowed for communication. A message board variable can be written only by the one process which "owns" that variable, but it may be read by all other processes if it is a shared variable. A process restarts in its noncritical section with each variable in its initial state.

The second type of failure is *read errors during writes*. This may occur because mutual exclusion of access to shared variables cannot be guaranteed. It is, therefore, possible for a sequence of reads during a write to return an arbitrary sequence of values. This gives the appearance that the bits of the variables are "flickering."

The algorithm given here is immune to the first type of failure and can be modified to be immune to the second type. Even if these failures do not occur or are not so severe as in the general case, this algorithm is sufficiently simple that it can be used when such robustness is not required.

While the originally intended application of solutions to this version of the problem is in distributed systems, there are other possible uses. One example where such a solution may be useful is on a VLSI (very large-scale integration) chip with thousands of processors. On such a large chip it is no longer possible to have all the processors run on the same clock because of the time delay in a clock pulse across the chip. If the processors wish to synchronize via the concurrent control problem, then interesting problems arise. The first is that the reliability of an individual processor may not be guaranteed, for example, due to nonuniform conditions in the wafer from which it is made. Second, when two processors running on different clocks communicate, the sum of pulses from the two processors might result in a so-called "runt" pulse. The runt pulse can in turn cause flip-flops to take an indefinite amount of time to stabilize; that is, the bit represented by the flip-flop flickers. Another problem is that, if a shared memory value is forced to be broken into its bits, then a process reading those bits while they are being changed may get some combination of the old and new values of the bits. Also, due to long propagation delays on a large chip, it is possible for one process to read another's value as it is being changed and get the new value. But then later a third process may do a read and get the old value, since the new value has not yet reached it. All of these communication problems are merely special cases to an algorithm which is totally immune to read errors during writes. On such a chip, where area is at a premium, as little space as possible should be taken up by the wires which connect the processors, whether the connection is a crossbar or a permutation network. This algorithm uses the minimum amount of shared memory per process when measured in bits (only two), an obvious

advantage. While not all of these problems may be important in the future of VLSI technology, the algorithm given here can be adapted to handle them all.

The space complexity of solutions is measured by counting the maximum number of shared memory values required by any process. Clearly, two values are required in order to communicate at all. Three values were shown to be necessary in general [11]. The main result of this paper shows that four values suffice. Tight space bounds appear frequently in studying the complexity of concurrent control problems; see, for example, [2, 10, 11].

3. THE ALGORITHM

The algorithm presented here uses an algorithm discovered by Burns [1] and Lamport [8] and the "stable clock" algorithm in [11]. The exit protocol is trivial; so the focus here is on explaining the entry protocol. The correctness of the algorithm will be shown as the development of the algorithm proceeds. The processes are numbered from 1 to N, with process P_i using the shared variable C[i] in addition to local variables. The shared variables take on values from 0 to 3 (0 initially). The simplest outline of the algorithm for P_i is given below. (ClockCode comes from [11], while ContendCode is adapted from [1, 8].)

```
ClockCode /* entry */
ContendCode /* protocol */
Critical Section
C[i] := 0 /* exit protocol */
```

ContendCode is presented first. Assume for now that ClockCode prevents any process numbered higher than P_i from passing entirely through ClockCode to ContendCode while P_i is in ContendCode. Clockcode uses the values 1 and 2, while ContendCode mainly uses 3 for the shared variables C[i]. The outline of ContendCode for P_i is given below.

```
S := C[i]; \qquad /^* C[i] \text{ is either 1 or 2 }^*/
RESET: C[i] := S;
\text{wait until } \forall j > i, C[j] \neq 3;
C[i] := 3;
\text{if } \exists j > i \text{ such that } C[j] = 3 \text{ then goto } RESET;
\text{wait until } \forall j < i, C[j] \neq 3;
```

LEMMA 1. ContendCode preserves mutual exclusion.

PROOF. That ContendCode preserves mutual exclusion is easy to see due to the last four lines alone. For a process to enter its critical section it must first set its C[i] to 3 and then not see any other process's shared variable as 3. Therefore, if P_i and P_j were both to be in their critical sections at the same time, then P_i would have set C[i] to 3 and not have seen C[j] as 3, while P_j must have similarly set C[j] to 3 and not have seen C[i] as 3. There is no legal way to order these four events so that both processes enter their critical sections. \Box

Lemma 2. Assuming that ClockCode prevents P_i from going from the noncritical section through ClockCode to ContendCode while P_i is in ContendCode if i < j, then no process waits forever in ContendCode.

PROOF. Arguing inductively, we show that, if P_n to P_{i+1} cannot get stuck in ContendCode, then P_i cannot get stuck in ContendCode. Consider P_n ; note that

its first wait loop is trivial, and it never returns to RESET; therefore, it can only be stuck if it is cycling at the last wait loop. In order for P_n to fail the last test forever, it must see at least one other process, call it P_j , with C[j] = 3. But if P_j is in the critical section, it will eventually leave and cannot reenter (and reset to 3) due to its first wait loop in ContendCode (j being less than n). Otherwise, P_j will go back to RESET and then wait. Similarly, other processes wind up waiting at the first wait loop. After that point, P_n can go on into the critical section. Proceeding with the induction, if P_i is waiting at the first wait loop, then, using the fact that P_{i+1} to P_n are not stuck, we know that eventually every higher numbered process leaves ContendCode and cannot reenter due to the special property of ClockCode. So, eventually, P_i will set its C[i] to 3 and not fail the next if test. From that point on, it only waits for lower numbered processes whose C[j] values are 3. But, as in the case of P_n , eventually all of these processes will either leave the critical section or return to the first wait loop in ContendCode. Hence P_i will eventually execute its critical section. \Box

ClockCode uses a modified version of the "stable clock" algorithm in [11]. (The stable clock is an improvement over a less stable version [11]. Both algorithms allow one to keep many processes relatively in step, hence the name "clock.") There are two "sides" for each process, C[i] = 1 and C[i] = 2. There is a joining rule (for initially picking a side) and a ticking rule (for changing sides). It is shown in [11] that no process may be kept from ticking forever; it also follows from the definition of ticking that no process may tick twice while a lower process does not tick at all. These facts are crucial to the correctness of the ClockCode, whose very simple outline is given below.

joinclock tick twice

The informal descriptions of joining and ticking are given first; the complete descriptions are given later along with the full algorithm. In the original clock algorithm [11], a process selects a clock value by looking at processes to its "left" and taking their clock value (ignoring processes with no clock value). To the "left" means that P_i starts with P_{i-1} and searches down to P_1 if necessary. In the case of P_1 or if no lower numbered process has a clock value, then P_i searches from P_n down to P_{i+1} and takes the opposite side of the first process it sees. If P_i sees no other process with a clock value, then it selects 1. The ticking rule for P_i is to tick if all lower numbered processes have the opposite value. In the case of P_1 or if no lower processes have a clock value, then P_i again searches P_n through P_i and ticks if the first value it sees is the same as its own. This is repeated indefinitely until the process can tick. Note that, if P_i alone has a clock value, then it is able to tick by seeing that it has its own clock value.

That P_j cannot tick twice if P_i does not tick for j > i follows directly from the ticking rule. P_j may start with a value different from P_i and tick once but not again until P_i leaves the clock or changes. That no process is forever prevented from ticking is proved in [11]. The proof notes that the clock value of the highest numbered process that might be unable to tick will eventually be picked up by any higher numbered process, and then the lowest process will take the opposite side. This value will be propagated up by any intermediate processes to the original process, which then ticks.

The clock algorithm is slightly adapted for use in ClockCode. A value of 3, when seen by a higher numbered process trying to tick, is considered a blocking value; that is, ticking is prevented. It is considered the same as 1 at all other times. This gives us the following trivial lemma and one not-so-trivial lemma.

LEMMA 3. No process P_j may pass entirely from the noncritical section through ClockCode and into ContendCode while some P_i is in ContendCode if i < j.

PROOF. A process passing through ClockCode must tick twice. All lower numbered processes in ContendCode will have either a fixed clock value or 3. If 3 is seen, then the process waits. In order for the process to tick twice, all lower numbered processes must change clock values. But a process in ContendCode does not change clock values.

LEMMA 4. No process waits forever in ClockCode.

PROOF. Since joining the clock is wait-free, we only have to show that ticking always occurs eventually. Assume that some P_j can be stuck waiting forever to tick, and, furthermore, it is the highest numbered such process. The processes higher numbered than P_i (if there are any) will, after a finite amount of time, each be doing one of the following general things: (1) waiting in ContendCode, (2) cycling through some subset of the protocol which includes ClockCode, (3) never leaving the noncritical section, (4) waiting in ClockCode to tick, or (5) cycling through part of ClockCode and failing. The combination of Lemmas 2 and 3 rule out the first case. The second part is impossible since it requires a process to tick twice while a lower numbered process does not tick at all. Processes in the third case can be ignored. The fourth case violates our assumption that P_i is the highest numbered such process. It is the fifth case that is nontrivial. Consider the next higher process which is cycling through part of ClockCode and then failing. Its clock value while it is active is going to be the same as P_i 's. Any still higher numbered process which is also cycling will similarly take on P_i 's clock value, either by directly seeing P_i 's value or indirectly through the intervening process. So eventually all processes above P_j will have P_j 's clock value when they are in the clock and cannot tick if P_i does not tick. Look now at the lowest numbered process in the clock. It will see P_i 's clock value (again directly or indirectly) and will change to the opposite value, if it does not already have that value. Similarly, this opposite value gets propagated up until all processes lower than P_j have the opposite value of P_j , and then P_j ticks. \square

THEOREM 1. The complete algorithm preserves mutual exclusion and does not have deadlock or lockout.

PROOF. Mutual exclusion is preserved by ContendCode (Lemma 1). No dead-lock or lockout is guaranteed by Lemmas 2, 3, and 4. Lemmas 2 and 3 show that waiting forever in ContendCode is impossible; waiting forever in ClockCode is ruled out by Lemma 4. \Box

The full algorithm for P_i is given in Figure 1. All variables except the shared array C are local to P_i . The initial and failure values of all variables are 0, except for i, which always contains P_i 's process number, and n, which is the number of processes.

```
function left(i);
                     /* looking for process to left */
  for h := i - 1 step -1 until 1 do
  begin
    t := C[h];
    if t = 1 or t = 3 then return 1
                                        /* normal */
    elseif t = 2 then return 2
  end:
  for h := n step -1 until i do
  begin
    t := C[h];
    if t = 1 or t = 3 then return 2
                                        /* opposite if higher */
    elseif t = 2 then return 1
  return 1
               /* no side values seen */
end left;
procedure tick(i);
                       /* changing sides */
begin
  WAITLOOP: wait until C[i] \neq left(i);
  for h := 1 step 1 until i - 1 do
                                                                     Figure 1
  begin
    t := C[h];
    if t = C[i] or t = 3 then goto WAITLOOP
                       /* tick */
  C[i] := 3 - C[i]
end tick;
/* main program */
                  /* ClockCode */
C[i] := left(i);
tick(i);
tick(i);
S := C[i];
RESET: C[i] := S;
                        /* ContendCode */
for j := i + 1 step 1 until n do
  wait until C[j] \neq 3;
C[i] := 3
for j := i + 1 step 1 until n do
  if C[j] = 3 then goto RESET;
for j := 1 step 1 until i - 1 do
  wait until C[j] \neq 3;
Critical Section
C[i] := 0;
```

4. THE BOOLEAN VARIABLE IMPLEMENTATION

If shared variables are restricted to be Boolean (binary), then the algorithm can be implemented using just two shared variables per process. Since the algorithm uses four values, two variables are necessary. Each C[i] is replaced by C1[i] and C2[i]. The value of C[i] is encoded into C1[i] and C2[i] in the obvious way, namely, C[i] = 2*C2[i] + C1[i], where **true** is equated with 1. Each change of C[i] is replaced by the appropriate settings of the Boolean variables. The algorithm is sufficiently robust that the specific order of settings is immaterial other than requiring that assignment of variables to true be performed before assignments of variables to false. (This prevents a ticking process from being overlooked while its variables are temporarily both false.) It makes no difference,

for instance, whether C1[i] or C2[i] is set to false first when failure occurs in the critical section. It is assumed, of course, that no variable is written that is not to be changed. The only unusual aspect is that it is possible for one process to see both C1[i] and C2[i] false while P_i was ticking from 2 to 1. (For instance, it reads C1[i] first, then P_i ticks, and then it reads C2[i].) However, by reading a second time when both are found false, this worry can be eliminated, since no process can tick from 2 to 1 twice without restarting (in which case its value would have really been 0 at some point during the read). More specifically, the function read and the procedure write given below are used whenever a shared variable is to be read or written, respectively. read(j) returns the value of C[j], while write(i, old, new) sets C[i] to new if its previous value is old.

```
function read(i);
begin
  for k := 1 step 1 until 2 do
     if C1[j] then if C2[j] then return 3
                             else return 1
     elseif C2[j] then return 2;
  return 0
end:
procedure write[i, old, new];
begin
  case (old, new)
         (0, 1): C1[i] := true;
         (0, 2): C2[i] := true;
         (1, 2): C2[i] := true; C1[i] := false;
         (2, 1): C1[i] := true; C2[i] := false;
         (1, 3): C2[i] := true;
         (2, 3): C1[i] := true;
         (3, 1): C2[i] := false;
         (3, 2): C1[i] := false;
         (3, 0): C1[i] := C2[i] := false;
  endcase
end;
```

Theorem 2. The Boolean variable implementation of the algorithm is correct, even if read errors occur during writes.

Proof. Consider the proofs of Lemmas 1 through 4 individually, Lemma 1 (mutual exclusion) remains the same, since the order for the two processes' operations are write, then read; so any read errors by one process imply that the other is still changing to 3 and will have to defer to the first process since it is yet to read. The proof of Lemma 2 (no lockout in ContendCode) must rely on an adaptation of Lemma 4's proof (ability to tick). The only possible effect read errors can have on a process in ContendCode is if a process's value is seen as 3 forever when in fact it is not. This can only happen if some process can repeatedly change clock values and its C1 and C2 are both seen as true when it is in the midst of ticking. What must be ruled out is some process being able to tick indefinitely while a different process is in ContendCode. Lemma 3 (which remains the same) rules out higher numbered processes ticking while a lower numbered process is in ContendCode; so the opposite has to be ruled out. As in the proof of Lemma 4, a process not ticking (in this case the process in ContendCode) prevents

higher numbered processes from ticking; this means that after a finite amount of time the lowest numbered process will also be unable to tick since it is not seeing the highest process tick. Soon the next lowest cannot tick, and so on until all processes are unable to tick. Lemma 4's proof is similarly unaffected since a process which is not ticking will not have its clock value misread; so the propagation of its clock value will proceed normally. \square

5. SUMMARY

A robust, succinct algorithm that is immune to two types of failures has been presented for the mutual exclusion problem. Processes may repeatedly fail and restart, and read errors during writes may occur. The algorithm uses just four values of shared memory per process, which is within one of the lower bound. This is currently the best result for $n \ge 3$. Huddleston (private communication) has a two-process algorithm that uses two values for one process and three values for the other. (The two-process lower bound given in [11] says that two values per process is impossible; hence, in an n-process solution, at most one process may have two values, and all the rest must have at least three values.)

It might appear at first glance that the algorithm does not exhibit any strong fairness property. It does, however, come close to being Linear Waiting. Note that, once a process joins the clock, no higher process can pass from the noncritical section into ContendCode before the other process finishes ticking. This means that no process can be passed more than two times total: once just as it joins ClockCode (if the other process is already in ContendCode) and again after it reaches ContendCode. Similarly, if failures are rare, then the clock has a strong tendency to stabilize; that is, the processes tick nearly in unison. The above observation would then apply to all lower processes. It might not be worth the effort to devise an efficient FIFO or Linear Waiting algorithm if we already have an algorithm that comes so close. For example, a FIFO algorithm might incur a heavy penalty in time and space.

It is difficult to weaken the model still further and ask for more robustness. If the flickering of bits were to continue beyond the end of a write, then it would become impossible to determine truly whether a process was in its critical section or not. Any more severe form of failure must have the property that a process which has failed in its critical section must be detectable. Similarly, errors in behavior (randomly changing variables and such) must be transient. Lamport [8] considers some of these restricted forms of failures. As a consequence, however, the number of values of shared variables becomes worse than exponential.

REFERENCES

- Burns, J. Complexity of Communication among Asynchronous Parallel Processes. Ph.D. dissertation, Georgia Inst. of Technology, Atlanta, Ga., 1981.
- Burns, J.E., Jackson, P., Lynch, N.A., Fischer, M.J., and Peterson, G.L. Data requirements
 for implementation of N-process mutual exclusion using a single shared variable. J. ACM 29, 1
 (Jan. 1982), 183-205.
- DEBRUIJN, N.G. Additional comments on a problem in concurrent programming control. Commun. ACM 10, 3 (Mar. 1967), 137-138.
- DIJKSTRA, E.W. Solution of a problem in concurrent programming control. Commun. ACM 8, 9 (Sept. 1965), 569.

- EISENBERG, M.A., AND McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem. Commun. ACM 15, 11 (Nov. 1972), 999.
- KATSEFF, H.P. A new solution to the critical section problem. In Conference Record of the Tenth Annual ACM Symposium on Theory of Computing, San Diego, Calif., May 1-3, 1978, pp. 86-88.
- KNUTH, D.E. Additional comments on a problem in concurrent programming control. Commun. ACM 9, 5 (May 1966), 321-322.
- 8. LAMPORT, L. The mutual exclusion problem. SRI International, Menlo Park, Calif., Oct. 1980.
- LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. Commun. ACM 17, 8 (Aug. 1974), 453-455.
- PETERSON, G.L. New bounds on mutual exclusion problems. Tech. Rep. TR68, Computer Science Dep., Univ. of Rochester, Rochester, N.Y., Feb. 1980.
- 11. PETERSON, G.L. Concurrency and complexity. Tech. Rep. TR59, Computer Science Dep., Univ. of Rochester, Rochester, N.Y., Aug. 1979.
- 12. Peterson, G.L., and Fischer, M.J. Economical solutions for the critical section problem in a distributed system. In Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, Boulder, Colo., May 2-4, 1977, pp. 91-97.
- 13. RIVEST, R.L., AND PRATT, V.R. The mutual exclusion problem for unreliable processes: Preliminary report. In Proceedings, 17th Annual Symposium on Foundations of Computer Science, Houston, Tex., 1976, pp. 1-8.

Received June 1980; revised September 1981, May 1982; accepted May 1982