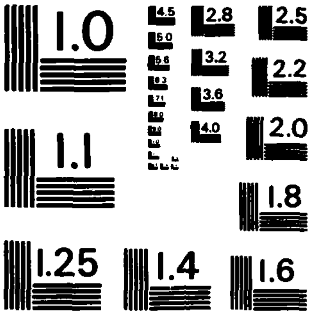
AD-A148 696 THE COLORED TICKET ALGORITHM(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE M FISCHER ET AL. AUG 83 MIT/LCS/TM-269 N00014-83-K-0125 1/1

UNCLASSIFIED F/G 9/2 NL

END



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

12

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-269

THE COLORED TICKET ALGORITHM

Michael Fischer

Nancy A. Lynch

James Burns

Allan Borodin

August 1983

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
MIT/LCS/TM-269

2. GOVT ACCESSION NO.
AD-A148696

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
The Colored Ticket Algorithm

5. TYPE OF REPORT & PERIOD COVERED
Interim Research
August 1983

6. PERFORMING ORG. REPORT NUMBER
MIT/LCS/TM-269

7. AUTHOR(s)
Michael Fischer, Nancy A. Lynch,
James Burns and Allan Borodin

8. CONTRACT OR GRANT NUMBER(s)
DARPA/DOD
N00014-83-K-0125

9. PERFORMING ORGANIZATION NAME AND ADDRESS
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
DARPA/Dept. of Defense
1400 Wilson Boulevard
Arlington, VA 22209

12. REPORT DATE
August 1983

13. NUMBER OF PAGES
21

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
ONR/Dept. of the Navy
Information Systems Program
Arlington, VA 22217

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

resource allocation, fault-tolerance, asynchronous systems, shared variables, and test and set

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Upper and lower bounds are proved for shared space requirements for solution of a problem involving resource allocation among asynchronous processes. The problem is to allocate some number, $k \geq 1$, of resources, in an environment in which processes can fail by stopping without. Allocation is to be as FIFO as possible, subject to variations imposed by the possibility of failures.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The Colored Ticket Algorithm

Michael Fischer
Computer Science Department
Yale University
New Haven, CT 06517

Nancy A. Lynch
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

James Burns
Indiana University
Computer Science
Bloomington, IN 47405

and

Allan Borodin
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A7

August, 1983

Keywords: resource allocation, fault-tolerance, asynchronous systems, shared variables, and test and set

*This work was supported in part by the Office of Naval Research under Contract N00014-83-K-0125, by the Office of Army Research under Contract DAAG29-84-K-0058, and by the National Science Foundation under Grants MCS 8306854 and 8302391-A01-DCR.

>or =

Abstract:

Upper and lower bounds are proved for shared space requirements for solution of a problem involving resource allocation among asynchronous processes. The problem is to allocate some number, k $\geq$ 1, of resources, in an environment in which processes can fail by stopping without warning. Allocation is to be as FIFO as possible, subject to variations imposed by the possibility of failures.

1. Introduction

The critical section problem has been widely studied for its illustrative value in problems of synchronization as well as for its practical application to real concurrent systems [BFJLP, CH1, CH2, Di1, EM, Kn, Lam, PF, RP]. The problem is to devise protocols for each of several communicating asynchronous parallel processes to control access to a designated section of code called the *critical section*. Such code might manipulate a common resource, in which case access to the critical section corresponds to allocation of the resource. In the simple case of a single nonsharable reusable resource such as a line printer or a tape drive, the two basic properties desired of the access policy are mutual exclusion and impossibility of deadlock. Mutual exclusion means that two processes can never simultaneously be executing their critical sections. Deadlock is a situation in which one or more processes are attempting to enter or leave their critical sections, but none of them ever succeeds. Finding appropriate protocols to insure these two properties is the *critical section problem*.

Two protocols comprise a solution. The *trying protocol* is the section of code that a process executes before being admitted to its critical section, and the *exit protocol* is run when the process leaves its critical section. Equivalently, the trying protocol allocates the resource corresponding to the critical section and the exit protocol returns it to the system.

Various solutions in the literature differ with respect to the underlying model of computation, "fairness properties" or permissible relative orders in which processes are admitted to their critical sections, time and space complexity of the algorithms, and immunity to various types of permitted "failure" of components of the system.

Burns, et al. [BFJLP] and Cremers and Hibbard [CH1, CH2] provide upper and lower bounds on the amount of shared memory needed to insure certain fairness properties such as absence of lockout, bounded waiting, and FIFO service order. The model used in those papers assumes a shared memory with a test-and-set primitive as the basic interprocessor communication mechanism. Rivest and Pratt [RP] and Peterson and Fischer[PF] also analyze the memory requirements for such problems, but their model assumes a much more limited form of access to the shared memory. Moreover, their algorithms are designed to continue to function correctly even under repeated "failure" of any number of processes, provided only that a failed process signal that it is no longer active. "Shutdown" is perhaps a more appropriate term for that kind of failure, for it carries no connotation of malfunction.

In this paper, we generalize the critical section problem to the case where some number $k \geq 1$ of processes (but not more) are permitted to be simultaneously in their critical sections. Regarded as a resource-allocation problem, we consider k identical copies of a non-sharable reusable resource, where each process can request at most one copy of that resource. We use the test-and-set model of [BFJLP] and, as in that paper, attempt to minimize the amount of shared memory used.

The exclusion property of the k-critical section problem, that at most k processes are ever simultaneously in their critical sections, we call *k-exclusion*. To avoid degenerate solutions, we must also formalize the notion that "it should be possible for as many as k processes to be simultaneously in their critical sections." We interpret this to mean, roughly, that if fewer than k processes are in their critical sections, then it is possible for another process to enter its critical section, even though no process leaves its critical section in the meantime. We this property "avoiding k-deadlock".

A trivial generalization of a binary semaphore yields a system exhibiting k-exclusion and no k-deadlock. Assume a shared variable, COUNT, which at any time contains the correct count of the number of processes currently in their critical sections. A process wanting to enter its critical section performs a test-and-set instruction on COUNT which, in one indivisible step, reads the value of COUNT, increments it if it was less than k, and stores the result back into COUNT. The process then proceeds to its critical section if it saw the count less than k, and it loops back and repeats the test otherwise (busy-waiting). A process leaving its critical section simply decrements COUNT.

This algorithm imposes no fairness criteria on the order in which processes enter their critical sections, and in fact it is possible that an individual process will always find the critical section "full" (i.e. COUNT = k) whenever it happens to examine COUNT and therefore be "locked out" of its critical section.

Rather than devise new algorithms for the k-critical section problem with various fairness conditions, an obvious approach is to try to reduce the k-critical section problem to a 1-critical section problem and then apply known solutions to the latter problem. A solution of this kind is commonly used in banks for scheduling people waiting for a teller. People entering the bank line up in a single queue. When one or more tellers become available, the person at the head of the queue goes to any free teller. To see the reduction that is illustrated by this simple example, think of the position at the head of the queue as a "resource". Only one person has this resource at a time, and the queue itself serves to allocate that resource in first-in-first-out (FIFO) order. Only the person holding the head-of-queue resource is permitted to go to a teller, so the order of service by a teller is likewise FIFO. Such a reduction is generally possible, and the number of values of shared memory increases by only a factor of (k + 1) over the requirement of the 1-critical section algorithm used.

The bank algorithm has a rather subtle defect which becomes apparent when several tellers become simultaneously free. If k $\geq$ 2 tellers are free, one would like the first k people in line to all move "simultaneously" to a teller, yet the algorithm requires them to file past the head of the queue one at a time. If the person at the front of the line is slow, the k - 1 people behind him are forced to wait unnecessarily. In fact, if the person at the front of the line "fails", then the people behind him wait forever and the system stops functioning. In this case, one failure can tie up all of the system's resources!

We are thus led to generalize our requirements to include controlling the degradation of processing in the event that a limited number of processes fail during the execution of their protocols.

Our notion of "failure" is quite different from the "shutdown" considered in [RP] and [PF]. We say a process *fails* if it simply ceases to execute steps of its program when it in its protocol. However, unlike a process which shuts down, a failed process does not announce to the world that it has failed. If it has not really failed, but is merely delaying its next step, it will later resume execution as if nothing had happened. Thus, there is no way for other processes to detect that a given process has failed: indeed, no finite portion of a computation suffices to determine whether a process has failed or is just running very slowly. The distinction can only be made in terms of the infinite behavior of the system - an active process *eventually* takes another step, whereas a failed process does not.

Our interest in this kind of failure stems partly from the practical problems of building fault-tolerant distributed systems and partly from the desire to understand the dependencies among processes competing for entry to their critical sections. Each instance where one process must wait for another indicates a lack of concurrency in the whole solution which, taken together, tend to cause the whole system to run at the speed of the slowest process. Algorithms which continue to operate correctly even when a limited number of processes fail cannot exhibit such simple dependencies. For example, it process A waits for process B to take some action and process B fails, then process A will wait forever and make no further progress toward its goal. Assuming that correct operation implies absence of lockout, then B's failure has caused the system to fail by locking out A. Insisting that algorithms be robust in the face of a certain amount of failure gives us a formal way of studying degrees of concurrency which in turn have implications for the running time of the system.

At first sight, the concepts of robustness and fairness, say FIFO ordering, appear to be contradictory. Robustness says that if one process fails in its trying protocol, the system must continue to function, so other processes which later enter their trying protocols will enter their critical sections ahead of the failed process. That, however, violates usual definitions of FIFO ordering. One might simply exempt failed processes from fairness constraints, but the resulting conditions are impossible to implement because of the fact that failure cannot be detected by the system after any finite length of time. The problem is circumvented by defining the fairness conditions not in terms of the order in which processes *enter* their critical sections but rather by the order in which they become *committed* to enter their critical sections. By "committed", we mean that a process no longer needs to wait for action by any other process before it can go into its critical section, nor can the actions of other processes prevent it from entering its critical region. Intuitively, when a process becomes committed, a copy of the resource is reserved for it, and actions of other processes are no longer needed in order for the given process to complete its trying protocol. The key distinction between committing and actual entry to the critical region is that a process might become committed passively as a result of some other process changing the value of the shared memory, whereas entry to the critical region can take place only by a positive action of the given process.

In this paper, we describe an algorithm, the Colored Ticket Algorithm, for solving the k-critical section problem, and a corresponding lower bound result.

The algorithm is robust, commits processes in FIFO order, and uses $O(N^2)$ values of shared memory, assuming that k is fixed. The algorithm simulates the behavior that would be achieved by allowing the entire queue of waiting processes to reside in the shared variable. However, actually keeping the queue in the shared variable would require a number of values exponential in N; our algorithm achieves the same effect with a "distributed implementation" of the queue, which reduces greatly the shared memory requirements. The algorithm satisfies strong robustness conditions, and we give an $\Omega(N^2)$ lower bound on the size of shared memory for any algorithm these conditions.

In [FLBB1], we consider other versions of the problem, using weaker robustness requirements. That paper contains a k-critical section algorithm which achieves bounded waiting, satisfies the weaker robustness requirements, and uses only O(N) values of shared memory. (The time requirements are worse than those of the Colored Ticket Algorithm, however.) It also contains an algorithm which achieves FIFO order of committing, satisfies the weaker robustness requirements, and uses only $O(N(\log N)^c)$ values of shared memory. (Again, time requirements are much worse than those of the algorithm in this paper.)

A preliminary version of the results of this paper, as well as the results of [FLBB1], appears in [FLBB2].

The reader is referred to [LF2] for two additional k-critical section algorithms. (Process failure is not considered in that paper.) [Pet] contains results similar to those of [LF2].

2. A Formal Model for Systems of Processes

The model used in this paper is derived from that of [BFJLP]. It can also be regarded as a special case of the general model of [LF1].

2.1. Processes and Systems

A *process* is a triple P = (V,X,δ), where

- V is a set of values for a shared variable,

- X is a (not necessarily finite) set of states, and

- δ is a total function from V x X to V x X, the *transition function*.

A transition from (v,x) to δ(v,x) is a *step* of process P.

We assume that the state set, X, is partitioned into disjoint subsets, or "regions": R, the *remainder region*, T, the *trying region*, C, the *critical region* and E, the *exit region*, where R is assumed to be nonempty. We assume that for every $x \in X$ and $v \in V$, the following conditions hold.

(a) $x \in R \cup T$ implies $\delta(v,x) \in T \cup C$, and

(b) $x \in C \cup E$ implies $\delta(v,x) \in E \cup R$.

Thus, processes are deterministic. A process in its remainder region (resp. critical region), if it takes a step, will either go directly to its critical region (resp. remainder region) or will enter its trying region (resp. exit region). Once in the trying region (resp. exit region), a process will remain in that region until it progresses to its critical region (resp. remainder region). We "abstract away" all program steps executed by a process while in its remainder and critical regions, treating only the protocols explicitly.

For a natural number N, let [N] denote {1,...,N}. A *system of N processes* is a collection of N processes, $P_i = (V, X_i, \delta_i)$, $1 \leq i \leq N$, all having the same shared variable. The remainder, trying, critical and exit regions of process P_i are denoted by R_i, T_i, C_i and E_i respectively.

If S is a system of N processes, then an *instantaneous description* (i.d.) of S is an N + 1-tuple, $q = (v,x_1,...,x_N)$, where v is a value of V and x_i is a state in X_i for all i. The functions δ_i of the individual processes have natural extensions to the set of i.d.'s of S, defined by $\delta_i(v,x_1,...,x_N) = (v',x_1,...,x_{i-1},x',x_{i+1},...,x_N)$, where $\delta_i(v,x_i) = (v',x')$. We also use (ambiguously) the notation R_i, T_i, C_i and E_i for the natural extensions of the denoted sets of states to corresponding sets of i.d.'s. For example, $(v,x_1,...,x_N) \in R_i$ if and only if $x_i \in R_i$.

If S is a system of N processes, then any finite or infinite sequence of elements of [N] will be called a *schedule* for S. In a natural way, each schedule defines an "execution" of system S, when applied to any i.d., q, of S. If $h = h_1,...,h_k$ is a finite schedule for S, then $r(q,h) = \delta_{h_k}(\delta_{h_{k-1}}(...\delta_{h_1}(q)...))$ is the *result* of applying schedule h to i.d. q. I.d. q' is *reachable* from q via schedule h provided $r(q,h) = q'$. I.d. q' is *reachable* from q provided q' is reachable from q via some finite schedule h.

2.2. Dependencies Among Processes

We have noted that processes are assumed to be always free to leave their remainder or critical regions on their own, but the same is not true for their trying and exit regions. We next give several important definitions which describe the possible dependencies among processes for making progress through their regions.

A process, P_i, in a system of processes is *C-able* (resp. *R-able*) in q provided that r(q,i) $\in C_i$ (resp. R_i). That is, the process can proceed directly to its critical region (resp. remainder region) by taking one step. P_i is *C-always-able* (resp. *R-always-able*) in q provided that for all finite schedules h not containing i, P_i is C-able (resp. R-able) in r(q,h). Thus, the C-always-able (resp. R-always-able) i.d.'s are those in which a process is poised, ready to enter its critical (resp. remainder) region whenever it next takes a step.

The properties described just above are of a somewhat different style from membership in the ordinary regions. Namely, it is possible for process P_i to become C-able, etc. because of steps of other processes. The C-always-able (resp. R-always-able) property can be thought of as describing passive belonging to the critical (resp. remainder) region. It is useful to combine the two concepts of actual and passive belonging to a region. Thus, we say that P_i is *C-committed* (resp. *R-committed*) in q provided that P_i is either in its critical region or else is C-always-able (resp. in its remainder region or else is R-always-able) in q.

Note that it is possible for a process to reach its critical region (resp. remainder region) without first becoming C-always-able (resp. R-always-able).

We say that P_i is *T-waiting* (resp. E-waiting) in q provided that P_i is in its trying region but is not C-committed (resp. in its exit region but not R-committed) in q.

2.3. Equivalence of Systems

Let S and S' be systems of N processes, with q and q' i.d.'s of S and S' respectively. We say that (S,q) and (S',q') are *equivalent* provided that for every finite schedule h, all processes are in the same regions in r(q',h) and r(q,h). That is, for every h and i, we have R(r(q,h)) = R(r(q',h)), and similarly for T, C and E.

3. Properties of Systems

Because of the fact that k-exclusion is only interesting if there are more processes than resources, we assume henceforth in this paper that N is strictly greater than k.

In this section, we state the properties which we would like a solution to the k-critical section problem to satisfy. Throughout this section, let S denote a system of N processes, q an i.d. of S, and k a natural number.

3.1. k-Exclusion

Our first condition is the basic k-exclusion condition. We say that q *violates k-exclusion* if the number of processes which are in their critical regions in q is strictly greater than k. S *satisfies k-exclusion* from q if no i.d. reachable from q in S violates k-exclusion.

3.2. Avoiding k-Deadlock

Our second condition describes our robustness requirements.

We say that the critical region is *full* in q provided that the number of processes which are C-committed in q is at least k.

We say that an infinite schedule h *exhibits k-deadlock* from q provided that no process changes regions or becomes committed in h applied from q, and at least one of the following two conditions holds.

(a) Some process is T-waiting, and the critical region is not full, in q.

(b) Some process is E-waiting in q.

S *avoids k-deadlock* from q provided there do not exist q' reachable from q and schedule h such that h exhibits k-deadlock from q'.

3.3. FIFO Committing

Our third and final condition describes the fairness property we require, FIFO committing. Intuitively, violation of FIFO committing occurs if a process remains T-waiting while another process leaves its remainder region and becomes C-committed. Similarly, a violation occurs if a process remains E-waiting while another process leaves its critical region and becomes R-committed.

More formally, we say that S *violates FIFO C-committing* (resp. FIFO R-committing) from q provided there exist q' reachable from q, schedule h and processes i and j such that i is T-waiting (resp. E-waiting) in q, i does not become C-committed (resp. R-committed) during h applied from q', P_j is in its remainder region in q' and P_j is C-committed in r(q',h). We say that S *violates FIFO committing* from q provided there S either violates FIFO C-committing or FIFO R-committing from q. S *satisfies FIFO committing* from q provided that S does not violate FIFO comitting from q.

3.4. The Problem

We say that a system S *satisfies the k-critical section problem* from i.d. q provided that S satisfies k-exclusion, avoids k-deadlock and satisfies FIFO committing, from q.

The following lemma will be used in demonstrating correctness of the Colored Ticket Algorithm.

Lemma 1: Assume (S,q) is equivalent to (S',q'). If S satisfies the k-critical section problem from q, then S' satisfies the k-critical section problem from q'.

Proof: Straightforward. ▮

4. The Queue Algorithm

In this section, we describe a simple but inefficient solution to the k-critical section problem. This basic algorithm, the Queue Algorithm, stores the entire queue of waiting and critical processes in the shared variable. A process in any of the firs(k positions of the queue is permitted to enter the critical region. This algorithm requires no nontrivial communication among processes, and in fact, each process need only make changes in the system i.d. at the moments of entry to the trying region and remainder region.

4.1. A Language for Describing Systems of Processes

In order to describe our algorithms, we require a suitable language. The algorithms are described in an Algol-like, Pascal-like language similar to the one used in [CH2] but designed to make the translation into the basic model transparent. Added to the usual sequential programming constructs are two synchronization statements, *lock* and *unlock*. In addition, the construct *waitfor C* is used as an abbreviation for "while not C do [unlock; lock]".

Lock and unlock statements always occur in pairs, an "unlock" followed immediately (syntactically) by a "lock". *Location counter values* correspond to the points in the code immediately preceding each lock statement. States of the process, P = (V,X,δ), defined by a program, correspond to a particular location counter value together with values for all the program's local variables. Transitions are defined as follows. If the program is started with its location counter and local variable values described by state x, and v as the value of the shared variable, and if the program is run according to usual sequential programming rules, it might or might not reach an unlock statement. If it does, if x' is the state describing the resulting location counter and local variable values, and if v' is the new value of the shared variable, then let $\delta(v,x) = (v',x')$. If it does not, then let $\delta(v,x) = (v,x)$. (In general, of course, this decision is not effective, but still gives a well-defined answer. In actual execution, the values leading to the second alternative should never occur.)

4.2. The Queue Algorithm

We are now ready to present the Queue Algorithm. The shared variable contains only a single (generalized) queue, called QUEUE. This queue admits two operations, ADD, which adds an element at the rear, and REMOVE, which removes an element with a particular value from anywhere in the queue. Initially, QUEUE is empty.

```
Queue Algorithm (Code for Process i):

while true do
  [ADD(i, QUEUE);
   waitfor i to be in one of the first k positions of QUEUE;
   unlock; /* Critical Region */ lock;
   REMOVE(i, QUEUE);
   unlock; /* Remainder Region */ lock]
```

The start state of a process has the location counter at the *last* lock statement of the program. The "initial" *i.d. consists of the given initialization of the shared variable and the start states of all processes. The regions are defined by the location counter values: any state for which the location counter is at the first (resp. last) explicit lock statement of the program is in the critical (resp. remainder) region Any state for which the location counter is at the lock statement implicit in the waitfor statement, is in the trying region. The exit region is empty.

Lemma 2: The Queue Algorithm solves the k-critical section problem.

Proof: Straightforward. ▮

While the Queue Algorithm satisfies all the correctness properties we want, keeping the queue in shared memory requires too much space to make the algorithm very interesting. Our goal is to find an algorithm equivalent to the Queue Algorithm which keeps a lot less information in the shared variable.

Thus, our problem is to devise a space-efficient "distributed simulation" of the Queue Algorithm.

5. The Colored Ticket Algorithm

In this section, we present the main contribution of the paper, the Colored Ticket Algorithm. This algorithm is very space-efficient, and is equivalent to the Queue Algorithm.

We begin with a presentation of an algorithm, the Uncolored Ticket Algorithm, which is intermediate between the Queue Algorithm and the Colored Ticket Algorithm. The Colored Ticket Algorithm is then described as a modification of the Uncolored Ticket Algorithm.

5.1. The Uncolored Ticket Algorithm

In this subsection, we present the Uncolored Ticket Algorithm. This algorithm is easily seen to be equivalent to the Queue Algorithm. Its efficiency is poor, however: it requires an infinite number of values of the shared variable.

We begin with an informal exposition which describes the operation of the algorithm.

We imagine an infinite sequence of numbered tickets to the critical region, starting with ticket number 1. A ticket is *issued* to each process as it enters the system. No ticket is ever reused. From time to time, a ticket becomes *valid*. If a process holds a valid ticket, it can enter its critical region, and all tickets held by processes in their critical regions are valid. At any time, exactly k tickets are valid; whenever there are fewer than k processes in their combined trying and critical regions, some of the valid tickets will not currently be issued.

In order to preserve FIFO enabling, tickets are validated in the same order as they are issued. Tickets are issued, starting with ticket 1, in numerical order. Initially, tickets numbered 1 to k are valid.

We imagine two pointers, ISSUE for the most recently issued and VALID for the most recently validated ticket respectively, both traversing tickets in increasing order. (We introduce a "dummy ticket" numbered 0, to which ISSUE points at initialization.) Either pointer may *lead* the other. When there are fewer than k processes in the system, VALID leads ISSUE, having already validated all issued tickets, as well as the next ticket(s) to be issued. When there are more than k processes, ISSUE leads VALID, indicating that there are processes in their trying regions waiting to be allowed to enter their critical regions. In this case, all valid tickets are issued, and there are also invalid issued tickets. When there are exactly k processes, the two pointers coincide, indicating that the k active processes hold the k valid tickets. VALID can lead ISSUE by at most k, while ISSUE can lead VALID by at most N - k.

Although FIFO enabling holds, some processes might get "skipped over" for actual order of entry to their critical regions (if they fail or go slowly). Thus, valid tickets might get widely separated. However, at any time when there are processes with invalid issued tickets, it is the case that the most recently validated ticket and all invalid issued tickets are consecutive: starting with the valid ticket, they have consecutive numbers until the ticket indicated by the ISSUE pointer is reached.

(Although we will not need to use it, there is a symmetric property to the above consecutivity property. Namely, if there are valid tickets which are not issued, then the last issued ticket and all the non-issued valid tickets are consecutive.)

The shared variable must contain enough information to indicate to each entering process what ticket it has been issued, and to indicate to each process whether its ticket is valid. Our variable contains the following information.

(a) ISSUE, (the number of) the ticket most recently issued.

(Initially, 0 appears.)

(b) VALID, the ticket most recently validated.

(Initially, k appears.)

Note that because of the consecutivity property, the shared variable alone suffices to determine the set of invalid issued tickets. Therefore, if a process in its trying region has a ticket, then the value of the shared variable allows that process to determine, at any time, whether its own ticket is valid.

The code follows. All processes have the same program.

```
Uncolored Ticket Algorithm (Code for Process i)

Local variable: TICKET

while true do
  [/* Take the next ticket. */

   TICKET := ISSUE + 1;
   ISSUE := TICKET;

   waitfor TICKET to be valid;
   unlock; /* Critical Region */ lock;

   /* Validate the next ticket. */

   VALID := VALID + 1;
   unlock; /* Remainder Region */ lock]
```

The start state of a process is defined by value 0 for TICKET and the location counter at the *last* lock statement of the program. The "initial" i.d. q consists of the given initialization of the shared variable and the start states of *all processes. The regions are defined* by the location counter values: any state for which the location counter is at the first (resp. last) explicit lock statement of the program is in the critical (resp. remainder) region. Any state for which the location counter is at the lock statement implicit in the waitfor statement, is in the trying region. The exit region is empty.

Lemma 3: The Uncolored Ticket Algorithm is equivalent to the Queue Algorithm.

Proof: Straightforward. ▮

In spite of Lemma 3, the Uncolored Ticket Algorithm is not particularly interesting in its own right. This is because the number of tickets is unbounded, so that the number of values taken on by the pointers in the shared variable is infinite.

5.2. The Colored Ticket Algorithm

In this subsection, we show how to modify the Uncolored Ticket Algorithm so that only a bounded amount of shared space is required. We do this by coloring and reusing the tickets.

In the Colored Ticket Algorithm, we use only a bounded number, (k + 1)N, of reusable tickets to the critical region, with N tickets (numbered 1 to N) of each of k + 1 colors (numbered 0 to k). A ticket is *issued* to each process as it enters the system, which it relinquishes upon leaving the system. No two processes are ever simultaneously in possession of the same ticket. As before, from time to time, a ticket becomes *valid*. If a process holds a valid ticket, it can enter its critical region, and all tickets held by processes in their critical regions are valid. At any time, exactly k tickets are valid; whenever there are fewer than k processes in the system, some of the valid tickets will not currently be issued.

As before, tickets are validated in the same order as they are issued. Tickets are issued, starting with ticket 1 of color 0, in numerical order within a color. After ticket N of a color has been issued, distribution resumes with a different-colored ticket numbered 1. Initially, tickets numbered 1 to k of color 0 are valid.

As before, we imagine two pointers, ISSUE for the most recently issued and VALID for the most recently validated ticket respectively, traversing tickets in the same order. Either pointer may *lead* the other. When there are fewer than k processes in the system, VALID leads ISSUE, having already validated all issued tickets, as well as the next ticket(s) to be issued. When there are more than k processes, ISSUE leads VALID, indicating that there are processes in their trying regions waiting to be allowed to enter their critical regions. In this case, all valid tickets are issued, and there are also invalid issued tickets. When there are exactly k processes, the two pointers coincide, indicating that the k active processes hold the k valid tickets. VALID can lead ISSUE by at most k, while ISSUE can lead VALID by at most N - k.

Although FIFO enabling holds, some processes might get "skipped over" for actual order of entry to their critical regions. Thus, valid tickets might get widely separated. However, at any time when there are processes with invalid tickets, it is the case that the most recently validated ticket and all invalid issued tickets are "*consecutive*" in the following sense. Starting with the valid ticket, they have consecutive numbers and are all of the same color until ticket number N is reached; if this occurs, then the sequence resumes with number 1 of the color of ISSUE, and continues with consecutive numbers of that color, until the ticket indicated by the ISSUE pointer is reached.

(As before, a symmetric property holds. Namely, if there are valid tickets which are not issued, then the last issued ticket and all the non-issued valid tickets are consecutive.)

In order to insure that two processes never simultaneously hold the same ticket, the algorithm follows the policy that no ticket with number 1 ever gets issued or validated by a leading pointer if there is a ticket of the same color currently issued or validated. The fact that it is always possible to select a new color when needed follows from the consecutivity condition of the previous paragraph. Namely, assume first that ISSUE is the leading pointer, is currently at ticket N of color c, and is ready to move to ticket 1 of some new color. By the consecutivity property, all issued tickets which are not valid are of color c, and at least one valid ticket is of color c. There are at most k - 1 other valid tickets, in the worst case each of a different color from each other and from c. At most k colors are used in total, so there is at least one color with no ticket currently issued or validated. Next, assume that VALID is the leading pointer, is currently at ticket N of color c, and is ready to move to ticket 1 of some new color. Then there are no invalid issued tickets, and at most k valid tickets. It follows that there is some color having no valid tickets.

As before, the shared variable must contain enough information to indicate to each entering process what ticket it has beer issued, and to indicate to each process whether its ticket is valid. Our variable contains the following information.

(a) ISSUE, (the number and color of) the ticket most recently issued.

(Initially, (N,k) appears.)

(b) VALID, the ticket most recently validated.

(Initially, (k,0) appears.)

(c) QUANT(i), $0 \leq i \leq k$, the quantity of each color represented by the k valid tickets.

(Initially, QUANT(0) = k and all others are 0.)

Considerable information can be determined from the value of the variable only. In particular, the variable suffices to determine whether ISSUE leads VALID, or vice versa, or whether they coincide. If ISSUE leads VALID, then the values of these two pointers together suffice to determine all the intervening tickets. Thus, because of the consecutivity property, the shared variable alone suffices to determine the set of invalid issued tickets.

If either VALID or ISSUE is a leading pointer and indicates a ticket with number N, then the shared variable alone suffices to determine NEWCOLOR, a color different from those of all the valid and issued tickets. Namely, NEWCOLOR can always be chosen to be some i for which QUANT(i) = 0. The reason this works was sketched above.

If a process has a ticket, then the value of the variable allows that process to determine, at any time, whether its own ticket is valid. This is because the shared variable suffices to determine the set of invalid issued tickets.

We use the capabilities just described, where needed, in the program below. All processes have the same program.

```
Colored Ticket Algorithm (Code for Process i)

Local variable: TICKET

while true do
  [/* Take the next ticket. */

   TICKET := if ISSUE.NUMBER < N then (ISSUE.NUMBER + 1, ISSUE.COLOR)
             else if VALID leads ISSUE then (1, VALID.COLOR)
             else (1, NEWCOLOR);
   ISSUE := TICKET;

   waitfor TICKET to be valid;
   unlock; /* Critical Region */ lock;

   /* Validate the next ticket. */

   VALID := if VALID.NUMBER < N then (VALID.NUMBER + 1, VALID.COLOR)
            else if ISSUE leads VALID then (1, ISSUE.COLOR)
            else (1, NEWCOLOR);

   /* Update quantity information. */

   QUANT(VALID.COLOR) := QUANT(VALID.COLOR) + 1;
   QUANT(TICKET.COLOR) := QUANT(TICKET.COLOR) - 1;
   unlock; /* Remainder Region */ lock]
```

The start state of a process is defined by an initial value of (1,0) for TICKET and the location counter at the *last* lock statement of the program. The "initial" i.d. q consists of the given initialization of the shared variable and the start states of all processes. The regions are defined by the location counter values: any state for which the location counter is at the first (resp. last) explicit lock statement of the program is in the critical (resp. remainder) region. Any state for which the location counter is at the lock statement implicit in the waitfor statement, is in the trying region. The exit region is empty.

Lemma 4: The Colored Ticket Algorithm is equivalent to the Uncolored Ticket Algorithm.

Proof: The equivalence is straightforward from the preceding discussion. ∎

Theorem 5: The Colored Ticket Algorithm solves the k-critical section problem. Also, if V is the shared variable in the Colored Ticket Algorithm, then $|V|$ is $O(N^2)$. (The constant is (2k choose k)$(k + 1)^2$.)

Proof: By Lemmas 3 and 4, the Colored Ticket Algorithm is equivalent to the Queue Algorithm. The first conclusion follows by Lemmas 2 and 1.

The analysis is as follows. The variable contains two pointers, each of which can take on $(k + 1)N$ values. The number of distinct values of the QUANT array is (2k choose k).

∎

The foregoing bound is really quite surprisingly low. In a robust algorithm, it is necessary to allow for the possibility that processes stop, and get "skipped over" in entry or leaving their critical regions. It seems, at first, inevitable that the variable must record ticket values for skipped committed processes, so that those processes can later know that they are committed. This seems to necessitate $O(N^k)$ values for the shared variable. The surprising observation is that committed processes can identify themselves without explicit identification of their tickets in the shared variable. Namely, the two pointers in the variable contain sufficient information to identify all the issued, invalid tickets. A process knows that its ticket is valid exactly if its ticket is not in this set.

This algorithm can be regarded as carrying out a distributed simulation of a queue, which produces drastic improvement in the shared memory required. Note, however, that this improvement does not incur a large penalty in local space usage. The only space required by each process is enough for a location counter and a single ticket value!

6. Lower Bound

In this section, we prove a lower bound on the number of values required by any algorithm which solves the k-critical section problem.

First, some additional definitions are required. We say that P_i is *C-independent* (resp. *R-independent*) in q provided that $r(q,i^m) \in C_i$ (resp. R_i) for some $m \geq 1$. That is, the process can proceed to its critical region (resp. remainder region) on its own, by taking some number of steps. We say that P_i is *C-always-independent* (resp. *R-always-independent*) in q provided that for all finite schedules h not containing i, P_i is C-independent (resp. R-independent) in r(q,h). Thus, the C-always-independent (resp. R-always-independent) i.d.'s are those in which a process is always able to enter its critical (resp. remainder) region on its own, provided it gets to take sufficiently many consecutive steps without interruption.

We require a construction and a lemma. Let k, N be natural numbers, with $N \geq k + 2$. Let S be a system of N processes, and q an i.d. such that S solves the k-critical section problem from q. Choose any q' which is reachable from q, with all processes in their remainder regions in q'. (The fact that S avoids k-deadlock can be used to construct such a q'.) Fix i and j, with $k \leq j < i \leq N - 1$. Construct a schedule, h(i,j), as follows.

Starting at q', each of $P_1, \ldots, P_k$ takes steps on its own, just until it enters its critical region. (The fact that S avoids k-deadlock shows that this is possible.) Then each of $P_{k+1}, \ldots, P_N$ takes one step, going to its trying region. Let P_N's state after its entry be denoted by x, for future reference. Then let P_1 take steps on its own, just until it returns to its remainder region, leaving one empty critical slot. Call the resulting i.d. q'' for later reference. Next, each of $P_{k+1}, \ldots, P_i$ in turn takes steps on its own, just until it returns to its remainder region. (The k-deadlock avoidance property and the FIFO committing property are both used here, to show that this is possible.) Finally, each of $P_{k+1}, \ldots, P_j$ takes one step, thereby entering its trying region once again. The resulting i.d. is denoted q(i,j).

It is not hard to see that P_{i+1} is C-always-independent in q(i,j).

Lemma 6: Assume $N \geq k + 2$. Let S be a system of N processes, q an i.d., such that S satisfies the k-critical section problem from q. For each i, j, let q(i,j) be defined as in the preceding construction. Then the shared variable has a distinct value in each q(i,j).

Proof: Assume the contrary, and consider two cases.

Case 1: V(q(i,j)) = V(q(i',j')) and i < i'

$P_{i'+1}$ is C-always-independent in q(i',j'), hence is C-independent in q(i,j). That is, $P_{i'+1}$ can take some number, m, of steps from q(i,j) and enter its critical region. We claim that the schedule h(i,j), followed by m steps of $P_{i'+1}$, violates FIFO committing from q. This is because $P_{i'+1}$ goes from its remainder to its critical region during this execution, while P_{i+1} remains T-waiting. (If P_{i+1} were to become C-committed during this execution, then a violation of k-exclusion would occur if this execution were extended with a step of P_{i+1}.)

Case 2: V(q(i,j)) = V(q(i,j')) and j < j'

Consider schedule h constructed as follows. Starting from q(i,j), $P_{j'+1}$ takes one step, thereby entering the trying region. Then each of $P_{i+1},\ldots,P_N,P_{k+1},\ldots,P_j$, in turn, takes sufficiently many steps to return to its remainder region. Then $P_{j'+1}$ is C-always-independent after application of h to q(i,j).

Now consider the application of h to q(i,j'). It must be that $P_{j'+1}$ is C-independent in q_1 = r(q(i,j'),h). Thus, there is some number m such that $P_{j'+1}$ enters its critical region in the schedule consisting of h(i,j') followed by h followed by m steps of $P_{j'+1}$. We claim that this schedule violates FIFO committing from q. Namely, $P_{j'+1}$ goes from its remainder to its critical region during this execution, while P_{j+1} remains T-waiting. ▮

Now we prove the main lower bound result.

Theorem 7: Assume $N \geq k + 2$. Let S be a system of N processes, q an i.d. of S such that S satisfies the k-critical section problem from q. Then |V| is $\Omega(N^2)$. More precisely, $|V| \geq$ k(N-k-1 choose 2) = (1/2)k(N-k-1)(N-k-2).

Proof: The proof proceeds by induction on k.

Base: k = 1

By Lemma 6, there are at least (N-1 choose 2) $\geq$ 1 x (N-2 choose 2) distinct values.

Inductive step: k > 1

By Lemma 6, there are (N - k - 1 choose 2) distinct values of the variable for the i.d.'s q(i,j), where $k \leq j < i \leq N - 2$. Moreover, P_N is not C-independent at any of these i.d.'s. That is, for every i and j, with $k \leq j < i \leq N - 2$, no finite number of applications of δ_N to the pair (V(q(i,j)),x) can put P_N in its critical region.

Now reconsider the construction preceding Lemma 6. Starting at q'', each of $P_{k+1},\ldots,P_{N-1}$ takes steps on its own until it returns to its remainder region. Call the resulting i.d. q'''. Then P_N is C-always-independent in q'''. From q''', consider $P_1,\ldots,P_{N-1}$ as comprising a system, T, of N - 1 processes. Since S solves the k-critical section problem from q, it can be shown that T solves the k-1-critical section problem from (the appropriate restriction of) q'''. Thus, by induction, the number of values which can be taken on by T's variable is at least (k - 1)((N - 1) - (k - 1) - 1 choose 2) = (k - 1)(N - k - 1 choose 2). But P_N is C-always-independent in q''', so that for each v which can be taken on by the shared variable in i.d.'s reachable from q''' using only $P_1,\ldots,P_{N-1}$, some finite number of applications of δ_N to the pair (v,x) will put P_N in its critical region.

Thus, the total number of values of V is at least (N - k - 1 choose 2) + (k - 1)(N - k - 1 choose 2) = k(N - k - 1 choose 2), as needed.

▮

7. Summary and Open Questions

In this paper, we have described the k-critical section problem in general terms, and defined an extremely robust version of the problem: equivalence with a particular (simple but space-inefficient) Queue Algorithm.

As our main result, we have presented an interesting new algorithm, the Colored Ticket Algorithm, which solves the given version of the problem and uses only $O(N^2)$ values of the shared variable. In contrast, we have presented a lower bound proof which shows that any solution requires $\Omega(N^2)$ values.

There is still a large gap between the constants in the upper and lower bounds. Both depend on k, but the constant in the upper bound is exponential in k, while the constant in the lower bound is linear in k. It remains to close this gap.

8. References

[BFJLP] Burns, J., Jackson, P., Lynch, N., Fischer. M. and Peterson, G.
"Data Requirements for Implementation of N-Process Mutual Exclusion
Using a Single Shared Variable",
Journal of the Association for Computing Machinery, Vol. 29, No. 1,
(January 1982), pp. 183-205.

[CH1] Cremers, A. and Hibbard, T.
"Mutual Exclusion of N Processors Using an O(N)-Valued Message Variable",
5th ICALP, Udine, Italy,
Springer Lecture Notes in Computer Science 62, (July 1978, pp. 165-176.

[CH2] Cremers, A. and Hibbard, T.
"Arbitration and Queueing Under Limited Shared Storage Requirements",
TR No. 83, Abtelung Informatik, Universitat Dortmund, (March 1979).

[Di1] Dijkstra, E. W.
"Solution of a Problem in Concurrent Programming Control",
Communications of the ACM, Vol. 8, No. 9 (1965), p. 569.

[EM] Eisenberg, M.. and McGuire, M.
"Further Comments on Dijkstra's Concurrent Control Problem",
Communications of the ACM, Vol. 15, No. 11 (1972), p. 999.

[FLBB1] Fischer, M., Lynch, N., Burns, J. and Borodin, A.
"Resource Allocation with Immunity to Limited Process Failure",

Work in progress.

[FLBB2] Fischer, M., Lynch, N., Burns, J. and Borodin, A.
"Resource Allocation with Immunity to Limited Process Failure",
20th Annual Symposium on Foundations of Computer Science,
(October 29-31, 1979), pp. 234-254.

[Kn] Knuth, D. E.
"Additional Comments on a Problem in Concurrent Programming Control",
Communications of the ACM, Vol. 10 (1967), p. 137.

[Lam] Lamport, L.
"A New Solution of Dijkstra's Concurrent Programming Problem",
Communications of the ACM, No. 17 (1974), pp. 453-455.

[LF1] Lynch, N. A. and Fischer, M. J.
"On Describing the Behavior and Implementation of Distributed Systems", *Theoretical Computer Science*, No. 13 (1981), pp. 17-43.

[LF2] Lynch, N. A. and Fischer, M. J.
"A Technique for Decomposing Algorithms Which Use a Single Shared Variable",
Journal of Computer and System Sciences", Vol. 27, No. 3, (December 1983,
pp. 350-377.

[Pet] Peterson, G.
"A New Solution to Lamport's Concurrent Programming Problem using Small Shared Variables",
ACM Transactions on Programming Languages and Systems,
Vol. 5, No. 1, (January 1983), pp. 56-65.

[PF] Peterson, G. and Fischer, M.
"Economical Solutions for the Critical Section Problem in a Distributed System",
9th Symposium on Theory of Computing, (May 1977), pp. 91-97.

[RP] Rivest, R. and Pratt, V.
"The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report",
17th Symposium on Foundations of Computer Science,
(October 1976), pp. 1-8.

OFFICIAL DISTRIBUTION LIST

1984

Director 2 Copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research 2 Copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 6 Copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center 12 Copies
Cameron Station
Alexandria, VA 22314

National Science Foundation 2 Copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 1 Copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

Dr. G. Hopper, USNR 1 Copy
NAVDAC-OOH
Department of the Navy
Washington, DC 20374