DISTRIBUTED FIFO ALLOCATION OF IDENTICAL RESOURCES USING SMALL SHARED SPACE

Michael J. Fischer, Nancy A. Lynch,
James E. Burns and Allan Borodin

YALEU/DCS/TR 421
June, 1985

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 421	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DISTRIBUTED FIFO ALLOCATION OF IDENTICAL RESOURCES USING SMALL SHARED SPACE		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael J. Fischer, Nancy A. Lynch, James E. Burns and Allan Borodin		8. CONTRACT OR GRANT NUMBER(s) ONR: N00014-82-K-0154; U.S.ARO: DAAG29-79-C-0155; NSF: MCS77-02474, MCS77-15628, MCS78-01689, MCS-8116678
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science/Yale University Dunham Lab./10 Hillhouse Avenue New Haven, Connecticut 06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research 800 N. Quincy Arlington, VA 22217 ATTN: Dr. R.B. Grafton		12. REPORT DATE June, 1985
		13. NUMBER OF PAGES 39
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distributed unlimited

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

18. SUPPLEMENTARY NOTES

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

Distributed algorithm	Shared space complexity
Resource allocation	Fault tolerance
FIFO order	Stopping failures
Queue	Lower Bound
Critical section	

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

We present a simple and efficient algorithm for the FIFO allocation of k identical resources among asynchronous processes which communicate via shared memory. The algorithm simulates a shared queue but uses exponentially fewer shared memory values, resulting in practical savings of time and space as well as program complexity. The algorithm is robust against processes failure through unannounced stopping, making it attractive also for use in an environment of processes of widely differing speeds. In addition to its practical advantages, we show the algorithm is optimal (to within a constant factor) with respect to shared space complexity.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Distributed FIFO Allocation of Identical Resources Using Small Shared Space[†]

Michael J. Fischer
Yale University
New Haven, Connecticut

Nancy A. Lynch
Massachusetts Institute of Technology
Cambridge, Massachusetts

James E. Burns
Indiana University
Bloomington, Indiana

Allan Borodin
University of Toronto
Toronto, Ontario

June 26, 1985

[†]This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the U. S. Army Research Office under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS77-02474, MCS77-15628, MCS78-01689, and MCS-8116678.

Abstract

We present a simple and efficient algorithm for the FIFO allocation of k identical resources among asynchronous processes which communicate via shared memory. The algorithm simulates a shared queue but uses exponentially fewer shared memory values, resulting in practical savings of time and space as well as program complexity. The algorithm is robust against processes failure through unannounced stopping, making it attractive also for use in an environment of processes of widely differing speeds. In addition to its practical advantages, we show the algorithm is optimal (to within a constant factor) with respect to shared space complexity.

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 [BJLFP82], [Bur81], [CH75], [CH78], [CH79], [deB67], [Dij65], [EM72], [FLBB79], [Knu66], [Lam74], [Lam76], [Lam77], [Lam80], [Mor79], [Pet80], [Pet81], [PF77], [RP76]. 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 the code to be run when the process leaves its critical section and returns to the remainder of its code, called the *remainder section*. Equivalently, the trying protocol allocates the resource corresponding to the critical section and the exit protocol returns it to the system.

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. Again, entry to the critical section corresponds to allocation of a resource copy, but we ignore questions of just how the individual copies of the resource are managed.

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 call this property "avoiding k-deadlock".

Precise definitions of these properties are deferred until Section 3, after the algorithms have been presented.

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 an atomic transaction on COUNT which, in one indivisible step, reads the value of COUNT, increments it if it was less than k, and stores back the result. The process then proceeds to its critical section if it saw 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 will be "locked out" of its critical section forever.

Rather than devise new algorithms for the k-critical section problem with stronger fairness conditions, an obvious approach is to try to reduce the k-critical section problem to the 1-critical section problem and then apply known solutions to the latter problem, e.g. [BJLFP82], [CH75], [CH78], [CH79], [Lam74]. Such a hybrid solution 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 "essentially" FIFO, modulo possible delay between leaving the head of the queue and arriving at a teller[1]. Such a reduction is possible given any 1-critical section solution, and the number of values of shared memory increases by only a factor of $(k + 1)$.

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

[1]By running, a person might actually arrive at a teller before another who was ahead of him in the queue. Nevertheless, we consider this to be a reasonable approximation of what people mean by FIFO since once one arrives at the front of the queue, one no longer has to wait for others.

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 [RP76] and [PF77]. Unlike a process which shuts down, a failed process does not announce to the world that it has failed. Rather, we say a process *fails* if there is a time after which it executes no more steps of its program. To distinguish a failed process from a correct one that is merely running very slowly, one must look infinitely far into the future and determine that it never takes another step. Thus, other processes have no way of distinguishing a failed process from a correct one in a finite amount of time. (In particular, timeouts won't work since we make no assumptions about the relative speeds of processes.)

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 of one process waiting for another indicates a lack of concurrency in the whole solution. Taken together, these dependencies 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, if process A waits for process B to take some action and process B were to fail, then process A would wait forever and make no further progress toward its goal. Hence, B's failure would cause the whole 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, for example, that if one process fails in its trying protocol, the system must continue to function. In particular, other processes which enter their trying protocols after the failed one will necessarily enter their critical sections ahead of it. Since the appar-

2 k-critical Section Algorithms

In this section, we present four algorithms, each a refinement of the preceding, the last of which is the Colored Ticket Algorithm. The algorithms run in an environment consisting of N processes, each with its own private memory, and a common shared memory (database) through which the processes communicate. Access to the shared memory is via atomic transactions that allow a process, in one indivisible step, to read the entire contents of shared memory and modify it in an arbitrary way, depending in general on the data just read.

We specify our algorithms and the transactions they use in a Pascal-like language augmented with two new statements for specifying transactions, start and commit. Statements executed dynamically after start and before the next commit comprise a single atomic transaction. While it is possible in this language to write transactions of unbounded size (for example, by executing a loop between start and commit), the transactions we actually use are all bounded, a fact that is important for the implementation in terms of "test-and-set" instructions which we give in Section 3.

In order for our algorithms to have the desired correctness and robustness properties, we make two assumptions about the implementation of transactions:

1. A process crash in the middle of a transaction does not cause the system to hang and leaves the shared memory as it was before beginning the transaction.

2. The system never aborts transactions. (Alternatively, a transaction that is retried repeatedly will eventually succeed.)

While these assumptions are difficult to implement exactly, they can be approximated in real systems, so we believe our algorithms will be useful in practice.

As a convenience, we use the construct "wait until C" as an abbreviation for "while not C do [commit; start]". Thus, every time around the wait loop ends one transaction and begins another.

2.1 The Queue Algorithm

We first 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 first k positions of the queue is permitted to enter its critical section. This algorithm requires no communication among processes other than that provided by the queue itself, and in fact, each process need only change shared memory at the moments of entry to the trying protocol and remainder section.

In the code given in Figure 1, the shared memory contains a single queue which admits two operations. ENQUEUE places an element at the rear of the queue, and REMOVE deletes a particular element from the queue, regardless of where it occurs. Initially, the queue is empty.

```
repeat forever
    start;
    ENQUEUE($i$);
    wait until $i$ is in one of the first $k$ positions of QUEUE;
    commit;

    { Critical Section }

    start;
    REMOVE($i$);
    commit;

    { Remainder Section }

end repeat.
```

Figure 1: Queue algorithm (code for process i).

Note that many transactions might be executed before the process reaches its critical section since each execution of the wait loop ends one transaction and begins another. However, only the first of these actually updates shared memory; the others are all "read-only".

2.2 Ticket Systems

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. In other words, we wish to devise a space-efficient "distributed simulation" of the Queue Algorithm.

All of our remaining algorithms are modeled after the ticket systems often used in bakeries. A process wishing to enter its critical section takes the next available *ticket* from an ordered sequence of tickets and then waits until its ticket becomes *valid*, at which point it is enabled to enter its critical section and proceeds to do so on its next step. When it leaves its critical section, it discards its ticket and *validates* the next ticket in order, thereby allowing the next process in line to proceed. (In case no process is currently waiting, the next ticket is nevertheless validated. and when a process eventually takes that validated ticket, it will proceed directly to its critical section.) Once a ticket becomes valid, it remains valid until discarded. Tickets are validated in the same order as they are issued, and at any time. exactly k (non-discarded) tickets are valid, some of which may not have yet been issued.

Since every process in its critical section holds a valid ticket, k-exclusion is satisfied. Since tickets are validated in the order in which they are issued, processes are enabled in FIFO order. so the algorithm satisfies our fairness condition. Robustness follows since a process does not modify shared memory from the time it enters its trying protocol until the time it returns to its remainder section: hence, whether or not it is alive in the meantime has no effect on the rest of the system.

Any such ticket algorithm simulates the Queue Algorithm in the sense that if the natural correspondence is made between transactions of the two algorithms and those transactions are run in the same order, then processes enter and leave their critical and remainder sections in exactly the same order in both. Indeed, the simulated queue of the Queue Algorithm can be obtained by arranging the processes holding tickets in increasing order of their tickets. Issuing a ticket corresponds to adding a process to the end of the queue, and discarding a ticket together with validating the next corresponds to removing a process from the queue. The k valid tickets always correspond to the first k positions of the queue.

Code to implement this basic paradigm is shown in Figure 2. Function TAKE_NEXT_TICKET *issues* the next available ticket and returns it to the calling program. Function IS_VALID(T) returns a Boolean value telling whether or not the ticket T is valid. Procedure VALIDATE_NEXT_TICKET(T) discards the ticket T and updates shared memory so as to cause the next invalid ticket in sequence to become valid. In order to fully specify a *ticket* algorithm, one must specify these three subroutines.

```
local variable TICKET;

repeat forever

    start;
    TICKET := TAKE_NEXT_TICKET;
    wait until IS_VALID(TICKET);
    commit;

    { Critical Section }

    start;
    VALIDATE_NEXT_TICKET(TICKET);
    commit;

    { Remainder Section };

end repeat.
```

Figure 2: Basic ticket algorithm (code for process i).

2.3 The Numbered Ticket Algorithm

The first ticket system we present, the Numbered Ticket Algorithm, uses an infinite number of values and hence requires an unbounded amount of shared memory for its implementation. The Colored Ticket algorithm, which uses only finite shared memory, is then described as two further modifications of this algorithm.

In the Numbered Ticket Algorithm, tickets are natural numbers in their usual order. The algorithm maintains two variables in shared memory. ISSUE holds the most recently issued ticket, and VALID holds the most recently validated ticket. Initially ISSUE = 0 and VALID = k. An entering process takes a ticket by incrementing ISSUE and using the variable's new value as its ticket number. Ticket number t is valid whenever VALID $\geq t$; hence, any process can determine by looking in shared memory whether or not its ticket is valid. A process returning to its remainder section discards its ticket and increments VALID.

The code for the Numbered Ticket Algorithm is shown in Figures 2 and 3. The initial value of the local variable TICKET does not matter to the operation of the algorithm.

```
global variable ISSUE = 0, VALID = $k$;

function TAKE_NEXT_TICKET: ticket;
begin
    ISSUE := ISSUE + 1;
    return ISSUE;
end;

function IS_VALID($T$): Boolean
begin
    return ($T \leq$ VALID)
end;

procedure VALIDATE_NEXT_TICKET($T$);
begin
    VALID := VALID + 1;
end;
```

Figure 3: Numbered ticket algorithm.

The drawback to the Numbered Ticket Algorithm is, of course, that ISSUE and VALID grow without bound.

2.4 Colored Ticket Algorithms

We now give two variations of the Numbered Ticket Algorithm based on the idea of colored tickets, the second of which is the final Colored Ticket Algorithm.

In the previous algorithm, either ISSUE or VALID could be larger than the other, and we say the larger one *leads* the smaller. (In case of equality, each *leads* the other.) However, they could never be too far apart. If VALID leads ISSUE, then there are VALID − ISSUE valid but not-issued tickets; hence

$$\text{VALID} - \text{ISSUE} \leq k.$$

If ISSUE leads VALID, then all k valid tickets are held by processes, and there are ISSUE − VALID invalid tickets held by processes waiting in their trying protocols. Since there are only N processes in all, k of which hold valid tickets, we conclude that

$$\text{ISSUE} - \text{VALID} \leq N - k.$$

Let $M \geq 1 + \max(k, N-k)$. Then we can determine which variable leads the other given only the information:

- $B = (\lfloor \text{VALID}/M \rfloor = \lfloor \text{ISSUE}/M \rfloor)$
- $V = \text{VALID} \bmod M$
- $I = \text{ISSUE} \bmod M$

Namely, if $B =$ **true**, then VALID leads ISSUE iff $V \geq I$, and if $B =$ **false**, then VALID leads ISSUE iff $V < I$.[2] Thus, we divide the tickets into blocks of size M. B is **true** iff VALID and ISSUE are in the same block; V and I are the relative positions of VALID and ISSUE within their respective blocks. It is easy to see that if VALID and ISSUE are not in the same block, then they must be in consecutive blocks, and the condition on M insures that which block leads which can be determined by comparing V and I.

The colored ticket algorithms replace numbered tickets by ***colored tickets*** that consist of ordered pairs $T = (t, c)$, where t, the ***value*** of T, is a number between 0 and $M - 1$ indicating the position of the ticket within the block, and c, the ***color*** of T, is a non-negative integer indicating the block that contains the ticket. We write T.VALUE and T.COLOR to denote the two components of T. There is a natural one-to-one correspondence ψ between numbered ticket i and colored ticket $(i \bmod M, \lfloor i/M \rfloor)$. Using this correspondence, a process can determine for colored tickets whether VALID leads ISSUE without using the ordering on colors by computing:

- $B :=$ (VALID.COLOR = ISSUE.COLOR)
- $V :=$ VALID.VALUE
- $I :=$ ISSUE.VALUE

[2] In case $k \neq N - k$, we can actually take $M = \max(k, N-k)$ and adjust the conditions appropriately.

and then applying the above remarks. It also follows from the above remarks that $|\text{VALID.COLOR} - \text{ISSUE.COLOR}| \leq 1$.

Now, a process can easily determine whether or not a ticket T that it holds is valid. T is always valid if its color differs from both VALID.COLOR and ISSUE.COLOR, for then its color must be less than both. If T's color is the same as VALID.COLOR, then T is valid iff T.VALUE $\leq$ VALID.VALUE. Finally, if T's color is the same as ISSUE.COLOR but different from VALID.COLOR, then T is valid iff VALID leads ISSUE. Using these ideas, the function IS_VALID can be defined as in Figure 4.

```
function LEADS($A, B$): Boolean; { Tests if $A$ leads $B$ }
begin
    if $A$.COLOR = $B$.COLOR then
        return ($A$.VALUE $\geq$ $B$.VALUE)
    else
        return ($A$.VALUE $<$ $B$.VALUE);
end;

function IS_VALID($T$): Boolean;
begin
    if $T$.COLOR = VALID.COLOR then
        return ($T$.VALUE $\leq$ VALID.VALUE)
    else if $T$.COLOR = ISSUE.COLOR then
        return LEADS(VALID, ISSUE)
    else
        return true;
end;
```

Figure 4: Validity testing functions for colored tickets.

Unbounded Colored Ticket Algorithm We complete the Unbounded Colored Ticket Algorithm by exhibiting in Figure 5 the definitions for the ticket issuing and validating functions. Initially, ISSUE = $(0, 0)$ and VALID = $(k, 0)$.

The Unbounded Colored Ticket Algorithm simulates the Numbered Ticket Algorithm using the correspondence ψ between numbered tickets and

```
constant $M = 1 + \max(k, N - k)$;

global variable ISSUE = $(0,0)$, VALID = $(k,0)$

function TAKE_NEXT_TICKET: ticket;
begin
     if ISSUE.VALUE $< M - 1$ then
          ISSUE.VALUE := ISSUE.VALUE + 1
     else begin
          ISSUE.VALUE := 0;
          ISSUE.COLOR := ISSUE.COLOR + 1;
          end;
     return ISSUE;
end;

procedure VALIDATE_NEXT_TICKET($T$);
begin
     if VALID.VALUE $< M - 1$ then
          VALID.VALUE := VALID.VALUE + 1
     else begin
          VALID.VALUE := 0;
          VALID.COLOR := VALID.COLOR + 1;
          end;
end;
```

Figure 5: Unbounded colored ticket algorithm.

colored tickets given above. Thus, we have bounded the set of ticket "values" at the cost of introducing an unbounded set of "colors". It may appear that no progress has been made, but the algorithm paves the way for the final modification which yields the space-efficient Colored Ticket Algorithm.

Colored Ticket Algorithm We now present the main contribution of the paper, the Colored Ticket Algorithm. Like the previous algorithms, it simulates the Queue Algorithm, but it is very space efficient, requiring only $\mathcal{O}(N^2)$ values of shared memory. It is obtained by modifying the Unbounded Colored Ticket Algorithm so that only $k+1$ different colors are used. This requires that tickets (and colors) be reused.

The change from the unbounded version of the algorithm comes when ISSUE.COLOR or VALID.COLOR is to be incremented. The new algorithm instead considers two cases. If the leading pointer (ISSUE or VALID) is being incremented, then a new color is chosen that is different from the color of any currently issued or validated ticket and different from the color of the other pointer. This insures that no two processes ever simultaneously hold the same ticket. If the trailing pointer is being incremented, then it is set equal to the color of the leading pointer. That this is correct follows from the fact that the pointers (in the Numbered Ticket Algorithm) never differ by more than M.

To see that it is always possible to select a new color when needed, we show (for the Unbounded algorithm) that every color in use *at the time a new color is needed* is the same as the color of some valid ticket; hence, at most k colors are then in use. A color is *in use* if it is the color of a valid or issued ticket that has not been discarded, or if it is equal to VALID.COLOR or ISSUE.COLOR. Note that in the exit protocol, a new ticket is validated immediately before the old one is discarded, so except for the brief moment between validating the new ticket and discarding the old one, exactly k tickets are valid, the most recently validated ticket T is still valid, and VALID.COLOR = T.COLOR. Hence, VALID.COLOR is always the color of one of the k valid tickets, so it suffices to show that when a new color is needed, both ISSUE.COLOR and the colors of all issued but not yet validated tickets are the same as VALID.COLOR.

There are two cases. If a new color is needed because VALID is about to be incremented, then VALID.VALUE = $M-1$, VALID leads ISSUE, and a process is in its exit protocol attempting to validate a new ticket. Then ISSUE.COLOR = VALID.COLOR since ψ^{-1}(VALID) $-$ ψ^{-1}(ISSUE) $\leq k \leq M-1$. Since there are no issued but not validated tickets, the only colors in use are those belonging to the k valid tickets.

On the other hand, if a new color is needed because ISSUE is about to be incremented, then ISSUE.VALUE $= M - 1$, ISSUE leads VALID, and a ticket is about to be issued to an entering process. Again, ISSUE.COLOR = VALID.COLOR, for $\psi^{-1}(\text{ISSUE}) - \psi^{-1}(\text{VALID}) \leq N - k \leq M - 1$.[3] Moreover, any outstanding invalid tickets lie between VALID and ISSUE, so they also have color VALID.COLOR. Again, the only colors in use are those belonging to the k valid tickets.

We conclude that with $k+1$ colors altogether, there is always a free color whenever a new one is needed.

To permit a process to determine which color is free, we introduce an array QUANT of length $k + 1$ into the shared variable, where QUANT$(c) \in \{0, 1, \ldots, k\}$ gives the number of valid tickets of color c. There are exactly k valid tickets, so the total number of different values for the QUANT array is the number of partitions of k into $k + 1$ sets, or $\binom{2k}{k}$. While this number is exponential in k, it is independent of N. QUANT is updated whenever a ticket is discarded and a new one is validated.

The code for finding a new color is shown in Figure 6. It simply scans for a color with QUANT = 0.

```
function NEW_COLOR: integer; { *Returns unused color* }
local variable $C$;
begin
    $C := 0$;
    while QUANT($C$) > 0 do $C := C + 1$;
    return $C$
end;
```

Figure 6: Find unused color function (used by colored ticket algorithm).

The final algorithm is contained in Figures 2, 4, 6 and 7. Initially, ISSUE = (0,0) and VALID = $(k, 0)$.

This algorithm simulates the Unbounded Colored Ticket Algorithm. To prove this, one shows that any two issued or validated (and not discarded) tickets T and T' have the same color in this algorithm iff they have the same

[3]Actually, $\psi^{-1}(\text{ISSUE}) - \psi^{-1}(\text{VALID}) \leq N - k - 1$ since the entering process does not yet hold a ticket, but we do not make use of this fact.

```
constant $M = 1 + \max(k, N - k)$;

global variable ISSUE = (0, 0), VALID = (k, 0), QUANT[0], ..., QUANT[k] = 0;

function TAKE_NEXT_TICKET: ticket;
begin
    if ISSUE.VALUE < $M - 1$ then
        ISSUE.VALUE := ISSUE.VALUE + 1
    else begin
        if LEADS(ISSUE, VALID) then
            ISSUE.COLOR := NEW_COLOR
        else
            ISSUE.COLOR := VALID.COLOR;
        ISSUE.VALUE := 0
        end;
    return ISSUE;
end;

procedure VALIDATE_NEXT_TICKET($T$);
begin
    if VALID.VALUE < $M - 1$ then
        VALID.VALUE := VALID.VALUE + 1
    else begin
        if LEADS(VALID, ISSUE) then
            VALID.COLOR := NEW_COLOR
        else
            VALID.COLOR := ISSUE.COLOR;
        VALID.VALUE := 0
        end;

    { Update quantity information. }

    QUANT(VALID.COLOR) := QUANT(VALID.COLOR) + 1;
    QUANT($T$.COLOR) := QUANT($T$.COLOR) - 1;
end;
```

Figure 7: Colored ticket algorithm.

color in the Unbounded algorithm; hence, the two algorithms always make the same decisions. We leave the details to the reader.

The total number of shared memory values needed by the Colored Ticket Algorithm is the product of the number of values assumed by QUANT, ISSUE, and VALID. This works out to $\binom{2k}{k}((k+1)M)^2 = \mathcal{O}(N^2)$ as desired, since $M = \mathcal{O}(N)$.

3 A Formal Model for Systems of Processes

We now present a formal model of computation and state the conditions that define the k-critical section problem. The model is derived from that of [BJLFP82]. It can also be regarded as a special case of the general model of [LF81].

3.1 Processes and Systems

A *process* is a quadruple $P = (V, X, \delta, \mathcal{R})$, where

- V is a set of values for a shared variable,
- X is a (not necessarily finite) set of process states,
- δ is a total function from $V \times X$ to $V \times X$, the *transition function*, and
- $\mathcal{R}$ is a total function from X to $\{R, T, C, E\}$, the *region function*.

Assume process P is in state x and the shared memory has value v. A *step* of P changes the state to x' and the shared memory to v', where $(v', x') = \delta(v, x)$.

For a state $x \in X$, $\mathcal{R}(x)$ gives the *region* of x, where R denotes the *remainder region*, T the *trying region*, C the *critical region* and E the *exit region*. We assume that δ respects $\mathcal{R}$ as follows. For every $(v, x) \in V \times X$:

1. $\mathcal{R}(x) \in \{R, T\}$ implies $\mathcal{R}(\delta(v, x)) \in \{T, C\}$, and
2. $\mathcal{R}(x) \in \{C, E\}$ implies $\mathcal{R}(\delta(v, x)) \in \{E, R\}$.

The allowed transitions are indicated in Figure 8. The transitions out of R and T comprise the trying protocol, and the transitions out of C and E comprise the exit protocol.[4] We "abstract away" the steps comprising the critical and remainder sections treating only the protocols explicitly; hence the absence of self-loops on R and C. Thus, the next step of a process in R takes it out of R, and similarly for C.

[4]Our formal model imposes a slight restriction on the form of protocols in that all transitions leaving a state of the trying region must belong to the trying protocol (and similarly for the exit region and protocol). Thus, a process, once permitted to begin its critical section, must first take a step to leave the trying region before it begins executing steps of its critical section, and the step which takes it out of the trying region is considered to be a part of the trying protocol. This restriction is for technical convenience only and does not weaken the results, for any protocol can be easily put into this form by adding dummy steps to the ends of the trying and exit protocols.

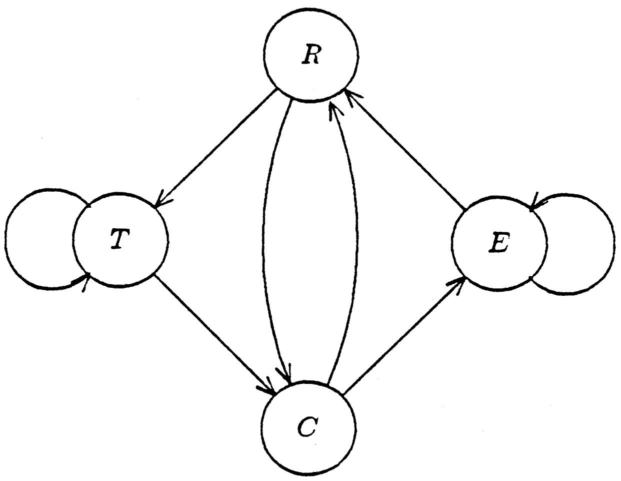

Figure 8: Possible region changes.

For a natural number N, let $[N]$ denote $\{1, \ldots, N\}$. A *system S of N processes* is a collection of processes $P_i = (V, X_i, \delta_i, \mathcal{R}_i)$, $1 \leq i \leq N$, all having the same shared variable V.

An *instantaneous description* (i.d.) q of S is a snapshot of the configuration of S and completely determines S's possible future behaviors. Formally, q is an $(N+1)$-tuple $(v, x_1, \ldots, x_N)$, where $v \in V$ is the contents of the shared variable and $x_i \in X_i$, $1 \leq i \leq N$, are the states of the N processes. We denote v by $V(q)$ and x_i by $X_i(q)$, $1 \leq i \leq N$.

The functions δ_i and $\mathcal{R}_i$ of the individual processes are naturally extended to functions on the set of i.d.'s of S by defining

$$\delta_i(v, x_1, \ldots, x_N) = (v', x_1, \ldots, x_{i-1}, x', x_{i+1}, \ldots, x_N)$$

where $(v', x') = \delta_i(v, x_i)$, and

$$\mathcal{R}_i(v, x_1, \ldots, x_N) = \mathcal{R}_i(x_i).$$

A *schedule* h for S is any finite or infinite sequence of elements of $[N]$.[5] A schedule describes the interleaving of process steps in a particular "run" of the system. Since the processes are deterministic, the entire run is determined by the starting i.d. q of the system and a schedule h. Formally, the

[5]Note that h is *not* required to be "fair". Processes that take only finitely many steps in h are considered to have failed.

run determined by q and $h = h_1, h_2, \ldots$ is the finite or infinite sequence of i.d.'s $Q(q,h) = q_0, q_1, q_2, \ldots$ such that:

1. If h is infinite then $Q(q,h)$ is infinite, and if h is finite then $|Q(q,h)| = |h| + 1$.
2. $q_0 = q$.
3. If q_{i-1}, q_i are successive elements of $Q(q,h)$, then $q_i = \delta_{h_i}(q_{i-1})$.

If $Q(q,h)$ is finite, then the last i.d., q_s, is the ***result*** of $Q(q,h)$, and we denote q_s by $\delta(q,h)$, extending δ once again. I.d. q' is ***reachable from q via h*** provided $\delta(q,h) = q'$, and q' is ***reachable from q*** if q' is accessible from q via some finite schedule h.

3.2 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*** if for every finite schedule h, all processes are in the same regions in $\delta(q,h)$ and $\delta'(q',h)$; that is, for every $i \in [N]$, $\mathcal{R}_i(\delta(q,h)) = \mathcal{R}'_i(\delta'(q',h))$.

3.3 Dependencies Among Processes

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

Let Z denote any region. A process P_i in a system of processes is ***Z-enabled*** in i.d. q if for every schedule $\bar{h}$ in which i occurs infinitely often, there is a finite prefix h of $\bar{h}$ such that $\mathcal{R}_i(\delta(q,h)) = Z$. Thus, the Z-enabled i.d.'s are those in which a process is either already in Z or will eventually enter Z if it takes infinitely many steps, no matter what the other processes do. Note that a process P_i can become Z-enabled because of its own actions or because of actions of other processes. A Z-enabled process can be thought of as passively belonging to region Z.

We say that P_i is ***T-waiting*** in q if it is in T but is not C-enabled in q. Similarly, we say that P_i is ***E-waiting*** in q if it is in E but is not R-enabled in q.

3.4 Properties of Systems

We now state the properties that define the k-critical section problem. Throughout this section, S denotes a system of N processes, q an i.d. of S, $k < N$ a natural number, and $\#Y$ the cardinality of the set Y.

Our first condition is the basic k-exclusion condition.

- **k-Exclusion.** I.d. q satisfies *k-exclusion* if $\#\{i \in [N] \mid \mathcal{R}_i(q) = C\} \leq k$. S satisfies *k-exclusion from* q if every i.d. reachable from q in S satisfies k-exclusion.

Note that any set of processes that are C-enabled but not in C can, by taking steps on their own, reach an i.d. in which all are simultaneously in C. Thus, if S satisfies k-exclusion from q, the number of C-enabled processes in any i.d. reachable from q is at most k.

Our second condition describes our robustness requirements. We say that i.d. q is *k-full* if $\#\{i \in [N] \mid P_i \text{ is } C\text{-enabled in } q\} \geq k$. We say that a process P_i *makes progress* in a run if, for some pair of i.d.'s q' and q'' in the run, either

1. $\mathcal{R}_i(q') \neq \mathcal{R}_i(q'')$, or
2. P_i is T-waiting in q' but not in q'', or
3. P_i is E-waiting in q' but not in q''.

- **Avoidance of k-Deadlock.** An infinite schedule h *exhibits k-deadlock from* q if no process makes progress in the run $Q(q, h)$, and either

  1. some process is T-waiting in q and q is not k-full, or
  2. some process is E-waiting in q.[6]

  S *avoids k-deadlock* from q if no infinite schedule exhibits k-deadlock from any i.d. reachable from q.

[6]Intuitively, a schedule exhibits k-deadlock if some process "wants" to make progress and progress is possible, but no process actually does make progress. At first sight, it might seem necessary to exclude failed processes from consideration in the formal definition, for we do not consider that progress is possible for failed processes. However, it is unnecessary to distinguish between failed and non-failed processes because our convention of no self-loops on R and C implies that *every* non-failed process "wants" to make progress (since it cannot continue taking steps and remain in R or C), and at least one process is non-failed in every infinite schedule.

Our third and final condition describes the fairness property, FIFO enabling. Intuitively, violation of FIFO enabling occurs if a process remains T-waiting while another process, beginning in its remainder region, becomes C-enabled. Similarly, a violation occurs if a process remains E-waiting while another process, beginning in its critical region, becomes R-enabled. Formally, let q be an i.d. and h a finite schedule. We say P_j *overtakes* P_i in $Q(q,h)$ if P_i is T-waiting in all i.d.'s of $Q(q,h)$, $\mathcal{R}_j(q) = R$, and P_j is C-enabled in $\delta(q,h)$, or if P_i is E-waiting in all i.d.'s of $Q(q,h)$, $\mathcal{R}_j(q) = C$, and P_j is R-enabled in $\delta(q,h)$.

- **FIFO Enabling.** *S achieves FIFO enabling* from q if for all q' reachable from q, all finite schedules h, and all $i,j \in [N]$, P_j does not overtake P_i in $Q(q',h)$.

The Problem Let q be an i.d. with every process in its remainder region. A system S solves the *k-critical section problem* starting from q if it satisfies k-exclusion, avoids k-deadlock, and achieves FIFO enabling from q.

4 Upper Bound

The Colored Ticket Algorithm, when translated into the formalism of our model, shows that the k-critical section problem can be solved by a system S that uses only $O(N^2)$ values of shared memory.

The translation requires a few comments. The transactions used in the algorithm make several accesses to the shared global variables, change internal variables, and branch to one of several possible exits depending on the values in shared and private memory at the start of the transaction. In our formal model, each transaction becomes a single process step. The program counter and all internal storage of a process is represented by the state x, and the entire contents of the global variables is represented by the value v of the shared variable. To construct the value (v', x') of the transition function $\delta(v, x)$, if the program counter in x points to a start instruction, then run the algorithm until it encounters a commit statement, and move the program counter past the commit. x' is the state and v' the shared memory contents that results. If a commit is never reached, or if the program counter in x does not point to a start instruction, then $\delta(v, x)$ is defined arbitrarily. This translation is not fully general, but it is adequate for algorithms such as ours in which every transaction terminates, and the next instruction to be executed after a commit is always a start.

Theorem 4.1 *The Colored Ticket Algorithm, when translated into the formal model as described above, solves the k-critical section problem and uses $(k+1)\binom{2k}{k}(1+\max(k, N-k))^2 = O(N^2)$ values of shared memory.*

A formal proof can be constructed following the development given in Section 2. Namely, one first proves that the Queue Algorithm solves the k-critical section problem. Next one shows that each of the three successively-presented algorithms is equivalent to the preceding in the sense formally defined in Section 3.2. Finally, one applies the following lemma, whose proof is straightforward.

Lemma 4.2 *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'.*

5 Lower Bound

In this section, we establish a lower bound on the size of the shared variable of any system of processes that solves the k-critical section problem. We assume throughout that k and N are natural numbers with $N \geq k+2$, S is a system of N processes, and q_0 is an i.d. with every process in its remainder region such that S solves the k-critical section problem from q_0.

Our method of proof is to construct a collection of runs and show that each leaves the shared variable in a distinct state. In order to carry out the construction, we need several "liveness" lemmas that show certain kinds of progress are always possible.

5.1 Progress Lemmas

We begin with some basic properties which follow from the fact that S solves the k-critical section problem. FIFO enabling places rather severe constraints on the order in which processes can become C-enabled, which are expressed by the relation $\prec_q$ that we next define.

Consider any i.d. q and processes P_i and P_j. We define $i \prec_q j$ to hold precisely if one of the following conditions holds at q:

1. P_i is C-enabled and P_j is in $E \cup R$;
2. P_i is C-enabled and P_j is T-waiting;
3. P_i is T-waiting and P_j is in $E \cup R$;
4. P_i and P_j are both T-waiting, and in some run leading from q_0 to q, P_i last entered T before P_j did.

We also define $\mathrm{ahead}_j(q) = \{i \in [N] \mid i \prec_q j\}$. The ordering $\prec_q$ is illustrated in Figure 9.[7]

The first lemma says that the order in which processes become C-enabled from q respects $\prec_q$.

Lemma 5.1 *Let q be reachable from q_0, and let $i \prec_q j$. Let h be a finite schedule such that P_j is C-enabled in $\delta(q,h)$. Then P_i is C-enabled in some i.d. in $Q(q,h)$.*

[7]One can show that if q is reachable from q_0, then $\prec_q$ is a strict partial order which totally orders the T-waiting processes in q, as illustrated, but we do not need this fact.

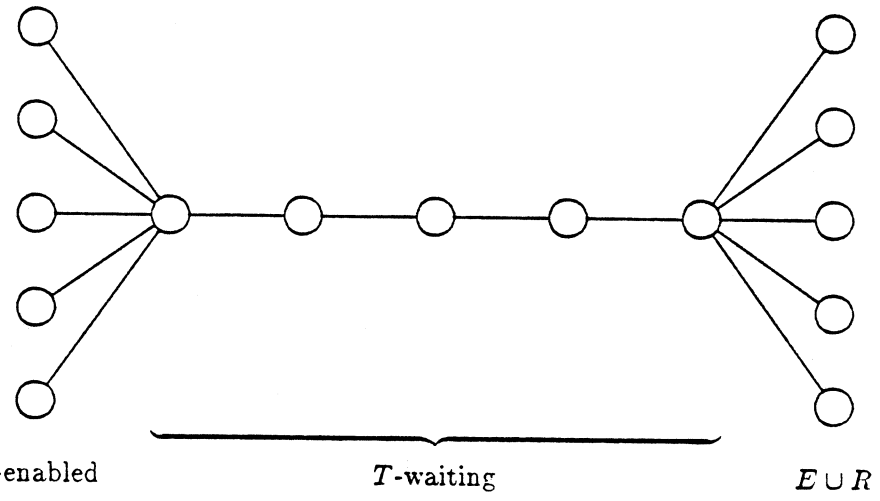

Figure 9: The relation $\prec_q$.

Proof: Assume the conditions of the lemma. Since $i \prec_q j$, P_i is either C-enabled or T-waiting in q. If P_i is C-enabled, then we simply choose $h' = \lambda$, the null schedule, and we are done. Hence, assume P_i is T-waiting in q.

Again since $i \prec_q j$, P_j is either in $E \cup R$ or is T-waiting in q. In either case, there exists an i.d. q_1 (possibly equal to q) and schedules h_0, h_1 such that q_1 is reachable from q_0 via h_0, q is reachable from q_1 via h_1, P_j is in $E \cup R$ in q_1, and P_i is T-waiting in every $q' \in Q(q_1, h_1)$. P_i is not T-waiting in every $q' \in Q(q, h)$, for if it were, then P_j overtakes P_i in $Q(q_1, h_1 \cdot h)$, violating FIFO enabling. Hence, P_i is C-enabled in some i.d. in $Q(q, h)$. ▌

The next lemma implies that among the T-waiting processes there is one that is "ahead" of all the others.

Lemma 5.2 *Let q be reachable from q_0, and assume that at least one process is T-waiting in q. Then there is a T-waiting processes P_i in q such that $i \prec_q j$ for all $j \neq i$ such that P_j is T-waiting in q.*

Proof: Let q be reachable from q_0 via h, and consider the run $Q(q_0, h)$. Order the T-waiting processes in q according to the times of their most recent entry to T in $Q(q_0, h)$, and let P_i be the first such process. By definition, $i \prec_q j$ holds for all $j \neq i$ such that P_j is T-waiting in q. ▌

Lemma 5.3 *Let q be reachable from q_0.*

1. *If q is not k-full, then no process is T-waiting in q.*
2. *No process is E-waiting in q.*

Proof: 1. Assume that q is not k-full but some process is T-waiting in q. We proceed to derive a contradiction.

By Lemma 5.2, there is a T-waiting process P_i in q such that $i \prec_q j$ for all $j \neq i$ such that P_j is T-waiting in q. Since P_i is T-waiting in q, there is a schedule h in which P_i takes infinitely many steps but it remains in T in every i.d. of $Q(q,h)$; hence P_i is T-waiting in every i.d. of $Q(q,h)$.

Suppose a process P_j becomes C-enabled during $Q(q,h)$. That is, suppose one can write $h = h_1 \cdot h_2 \cdot h_3$ such that P_j is not C-enabled in $q_1 = \delta(q,h_1)$, but P_j is C-enabled in $\delta(q_1,h_2)$. Then $i \prec_{q_1} j$ holds by definition, so by by Lemma 5.1, P_i is C-enabled at some i.d. in $Q(q_1,h_2)$, a contradiction. Hence, no process becomes C-enabled during $Q(q,h)$, so none of the i.d.'s in $Q(q,h)$ are k-full.

Now, for some suffix $Q(q',h')$ of the run $Q(q,h)$, no process makes progress since each process can change region or become R-enabled only a finite number of times without becoming C-enabled. Thus, h' exhibits k-deadlock from q', contradicting the avoidance of k-deadlock condition.

2. The proof is similar (and simpler). Assume that P_i is E-waiting in q. Then there is a schedule h in which P_i takes infinitely many steps but it remains in E in every i.d. of $Q(q,h)$. It follows that P_i is E-waiting in every i.d. of $Q(q,h)$.

Only processes P_j already in E in q can become R-enabled during $Q(q,h)$, and that can happen at most once per process, for otherwise P_j would overtake P_i, violating the FIFO enabling condition. Hence, in some suffix $Q(q',h')$ of the run $Q(q,h)$, no process makes progress since each process can change region or become C-enabled only a finite number of times without becoming R-enabled. Then h' exhibits k-deadlock from q', contradicting the avoidance of k-deadlock condition. ▮

The following lemma says that a process can only be C-enabled while it is in its trying or critical region.

Lemma 5.4 *Let q be reachable from q_0. If process P_i is C-enabled in q, then P_i is in $T \cup C$ in q.*

Proof: Assume the contrary, that P_i is C-enabled in q, and P_i is in $E \cup R$ in q. For each $j \in [N]$, $j \neq i$, run P_j for zero or more steps until an i.d. is reached in which it is in $T \cup C$. This procedure must terminate after a finite number of steps, for otherwise P_j remains forever in $E \cup R$. But that is impossible by Lemma 5.3 and the absence of self-loops on region R. Call the resulting i.d. q'.

In q', every process other than P_i is either T-waiting or is C-enabled. At most k processes can be C-enabled (by the remark following the definition of k-exclusion). Thus, since we assume $N \geq k+2$, some process P_ℓ is T-waiting in q'. P_i is still C-enabled in q' (by definition of enabling), so it enters C in the run $Q(q', i^m)$ for some m. By Lemma 5.3, q' is k-full, so P_ℓ remains T-waiting throughout $Q(q', i^m)$. But then P_i overtakes P_ℓ in $Q(q', i^m)$, violating FIFO enabling. ▮

The next lemma says that, no matter what the other processes do, any process in its trying region that takes infinitely many steps eventually reaches its critical region, provided that there are not too many processes ahead of it.

Lemma 5.5 *Let q be reachable from q_0, and let P_i be in T in q. Then $\#\text{ahead}_i(q) < k$ iff P_i is C-enabled in q.*

Proof: Assume the conditions of the lemma.
($\Rightarrow$) Suppose $\#\text{ahead}_i(q) < k$ but P_i is not C-enabled in q. Then P_i must be T-waiting in q, so by Lemma 5.3, q is k-full. But then all the processes which are C-enabled in q are in $\text{ahead}_i(q)$, so that $\#\text{ahead}_i(q) \geq k$. This is a contradiction.
($\Leftarrow$) If P_i is C-enabled in q, then P_i is in $T \cup C$ by Lemma 5.4. But then $\text{ahead}_i(q) = \emptyset$. ▮

The next lemma says that it is always possible for all the processes to run so as to end up simultaneously in their remainder regions.

Lemma 5.6 *Let q be reachable from q_0. Then there exists q' reachable from q such that every process is in its remainder region in q'. Moreover, q' can be reached from q via a schedule in which no process already in its remainder region in q takes any steps.*

Proof: It suffices to show that if not all processes are in their remainder regions, then there is some P_i not in R which is C- or R-enabled. Assuming we have shown that such a P_i exists, we run P_i until it changes regions. We then repeat this construction on each resulting i.d. until an i.d. is reached in which all processes are in R. This procedure must eventually terminate since each process can change regions only finitely many times before entering its remainder region.

Now suppose that every processes not in R is neither C- nor R-enabled in q. Then q is not k-full, since no process is C-enabled, by assumption and Lemma 5.4. By Lemma 5.3, no process is T- or E-waiting in q; therefore, no process is in $T \cup E$ in q. But also no process is in C in q since no process is C-enabled. Hence, every process is in R. It follows that if not all processes are in R, then some such process is C- or R-enabled, as desired. ▮

5.2 The Schedule $h(i,j)$

Now choose any q reachable from q_0 in which all processes are in their remainder regions. q exists by Lemma 5.6. Fix i and j, with $k \leq j < i \leq N-1$. Construct a schedule, $h(i,j)$, as follows.

1. Starting at q, each of $P_1, \ldots, P_k$ takes steps on its own, just until it enters its critical region. This is possible by Lemma 5.5. 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. (Note that x does not depend on i or j.)

2. P_1 takes steps on its own, just until it returns to its remainder region, leaving one empty critical slot. This is possible by Lemma 5.3. Call the resulting i.d. q' for later reference. (Note that q' does not depend on i or j.)

3. Each of $P_{k+1}, \ldots, P_i$ in turn takes steps on its own, just until it returns to its remainder region. This is possible by Lemmas 5.5 and 5.3.

4. Each of $P_{k+1}, \ldots, P_j$ takes one step, thereby entering its trying region once again. The resulting i.d. is denoted $q(i,j)$.

This construction is diagrammed in Figure 10. Arrows are labeled by '0', '1', or '*' to indicate that the corresponding process takes 0, 1, or an unspecified number of steps.

$$\begin{tikzcd}
P_1 & \underbrace{P_2 \dots P_k} & \underbrace{P_{k+1} \dots P_j} & \underbrace{P_{j+1} \dots P_i} & \underbrace{P_{i+1} \dots P_N} & I.D. \\
R \arrow[d, "*"] & R \arrow[d, "*"] & R \arrow[d, "1"] & R \arrow[d, "1"] & R \arrow[d, "1"] & q \\
C \arrow[d, "*"] & C \arrow[d, "0"] & T \arrow[d, "0"] & T \arrow[d, "0"] & T \arrow[d, "0"] & \\
R \arrow[d, "0"] & C \arrow[d, "0"] & T \arrow[d, "*"] & T \arrow[d, "*"] & T \arrow[d, "0"] & q' \\
R \arrow[d, "0"] & C \arrow[d, "0"] & R \arrow[d, "1"] & R \arrow[d, "0"] & T \arrow[d, "0"] & \\
R & C & T & R & T & q(i,j)
\end{tikzcd}$$

Figure 10: The Lower Bound Construction.

5.3 Distinctness of shared values

We now relate the construction to the size of shared memory.

Lemma 5.7 *The shared variable has a distinct value in each $q(i,j)$.*

Proof: Assume to the contrary that $V(q(i,j)) = V(q(i',j'))$ for $(i,j) \neq (i',j')$. Without loss of generality, it suffices to consider two cases.

Case 1: $i < i'$.

Among all T-waiting processes in $q(i',j')$, $P_{i'+1}$ was the first to enter its trying region, so $\#\text{ahead}_{i'+1}(q(i',j')) = k-1$. Then $P_{i'+1}$ is C-enabled in $q(i',j')$ by Lemma 5.5. Also $P_{i'+1}$ is in the same state in both $q(i,j)$ and $q(i',j')$. Since also the shared variable has the same value in both i.d.'s, it follows that $P_{i'+1}$, starting from $q(i,j)$, can take some number m of steps and enter its critical region. We claim that the schedule $\overline{h} = h(i,j) \cdot (i'+1)^m$ violates FIFO enabling from q. This is because $P_{i'+1}$ goes from its remainder to its critical region during $\overline{h}$ while P_{i+1}, which entered its trying region first,

6 Summary and Open Questions

In this paper, we have described the k-critical section problem in general terms and have defined an extremely robust version of the problem: equivalence with a particular simple but space-inefficient algorithm, the 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 $\mathcal{O}(N^2)$ values of the shared variable. Our lower bound proof shows that, for fixed k, this algorithm is optimal to within a constant factor in terms of number of values of shared memory.

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.

Acknowledgement

We are grateful to Brian Coan for helpful comments on an early draft of this paper.

References

[BJLFP82] J. E. BURNS, P. JACKSON, N. A. LYNCH, M. J. FISCHER, AND G. L. PETERSON. "Data Requirements for Implementation of N-Process Mutual Exclusion Using a Single Shared Variable". *J. ACM 29*, 1 (1982), 183-205.

[Bur81] JAMES E. BURNS. "Complexity of Communication among Asynchronous Parallel Processes". Ph. D. Thesis, Georgia Institute of Technology, 1981.

[CH75] ARMIN B. CREMERS AND THOMAS N. HIBBARD. "An Algebraic Approach to Concurrent Programming Control and Related Complexity Problems". Technical Report, University of Southern California, (Nov. 1975). (Presented at Symp. on Algorithms and Complexity, Pittsburgh, April 1976.)

[CH78] ARMIN B. CREMERS AND THOMAS N. HIBBARD. "Mutual Exclusion of N Processors Using an $O(N)$-Valued Message Variable". In *Proc. 5th ICALP, Udine, Italy*, Lecture Notes in Computer Science, vol. 62, Springer Verlag, 1978, 165-176.

[CH79] ARMIN B. CREMERS AND THOMAS N. HIBBARD. "Arbitration and Queueing under Limited Shared Storage Requirements". Technical Report No. 83, Dept. of Informatics, University of Dortmund (Mar. 1979).

[deB67] N. G. DEBRUIJN. "Additional comments on a Problem in Concurrent Control". *Comm. ACM 10*, 3 (Mar. 1967), 137-138.

[Dij65] EDSGAR W. DIJKSTRA. "Solution of a Problem in Concurrent Programming Control". *Comm. ACM 8*, 9 (1965), 569.

[EM72] MURRAY A. EISENBERG AND MICHAEL R. MCGUIRE. "Further Comments on Dijkstra's Concurrent Programming Control Problem". *Comm. ACM 15*, 11 (Nov. 1972), 999.

[FLBB79] MICHAEL J. FISCHER, NANCY A. LYNCH, JAMES E. BURNS, AND ALLAN BORODIN. "Resource Allocation with Immunity to Limited Process Failure". *Proc. 20th Annual IEEE Symp. on Foundations of Computer Science*, (Oct. 1979), 234-254.

[Knu66] DONALD E. KNUTH. "Additional Comments on a Problem in Concurrent Programming Control". *Comm. ACM 9*, 5 (1966), 321-322.