RESOURCE ALLOCATION WITH IMMUNITY TO LIMITED PROCESS FAILURE [†]

(Preliminary Report)

Michael J. Fischer (1), Nancy A. Lynch (2), James E. Burns (2), Allan Borodin (3)

(1) Dept. of Computer Science
University of Washington
Seattle, Washington 98195

(2) School of Info. & Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

(3) Dept. of Computer Science
University of Toronto
Toronto, Canada M5S 1A7

Abstract: Upper and lower bounds are proved for the shared space requirements for solution of several problems involving resource allocation among asynchronous processes. Controlling the degradation of performance when a limited number of processes fail is of particular interest.

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 processors 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 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 entry 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 entry protocol allocates the resource corresponding to the critical section and the exit protocol returns it to the system.

We find it useful to group a process' states into four regions: the entry or trying region consists of those states which a process can be in while running its entry protocol, and the exit region similarly consists of states corresponding to the exit protocol. The states between the entry and exit protocols comprise the critical region. All other states belong to the remainder region. We make no assumptions about what a process does while in its critical or remainder region except that it does not attempt to communicate with other processes running their protocols.

Various solutions in the literature differ with respect to the underlying model of computation, "fairness properties" or relative order in which processes are admitted to their critical regions, 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. Mcreover, 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 and that it return to its remainder region when it is later restarted. "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 $\ell \geq 1$ of processes (but not more) are permitted to be simultaneously in their critical sections. Regarded as a resource-allocation problem, we consider ℓ 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.

[†] This research was supported in part by the NSF under grants MCS77-02474, MCS77-15628, MCS78-01689 and US Army Research Office Contract Number DAAG29-79-C-0155.

CH1471-2/79/0000-0234$00.75 © 1979 IEEE

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

A trivial generalization of a binary semaphore yields a system exhibiting ℓ-exclusion and no ℓ-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 region 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 ℓ, and stores the result back into COUNT. The process then proceeds to its critical section if it saw the count less than ℓ, 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 = ℓ) whenever it happens to examine COUNT and therefore be "locked out" of its critical section.

Rather than devise new algorithms for the ℓ-resource problem with stronger fairness conditions, an obvious approach is to try to reduce the ℓ-resource problem to a 1-resource 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 arrival 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. Our first observation, used for Theorem 4.1, is that such a reduction is generally possible, and the number of values of shared memory increases only by a factor of $(\ell+1)$ over the requirement of the 1-critical section solution used. We use the previously-described semaphore solution but we protect access to the busy-wait loop with a 1-critical section algorithm of our choosing. Entry to the critical section then becomes a two-stage process. First, a process gains exclusive access to the semaphore, COUNT, by executing the 1-critical section algorithm. Then it waits for COUNT to become less than ℓ, whereupon it increments COUNT, releases its exclusive access to COUNT, and enters its critical section. (Processes are always permitted to decrement COUNT.) The code for each process is:

1. Perform the entry protocol for the 1-critical section algorithm.
2. (Busy Waiting) Wait until COUNT < ℓ, then set COUNT := COUNT + 1.
3. Perform the exit protocol for the 1-critical section algorithm.
4. Execute the critical section.
5. Set COUNT := COUNT - 1.

If the 1-critical section algorithm has the property that processes return to their remainder regions in the same order as they entered their critical regions, then the ℓ-resource algorithm satisfies fairness conditions corresponding to those of the 1-critical section solution.

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 deadlocking definition to include controlling the degradation of performance 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 sleeps if it simply ceases to execute steps of its program when it is not in its remainder region. However, unlike a process which shuts down, a sleeping process does not announce to the world that it is sleeping. If it is not really sleeping, but 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 gone to sleep; indeed, no finite portion of a computation suffices to determine whether a process is sleeping or just running very slowly. The distinction can only be made in terms of the asymptotic or infinite behavior of the system -- an active process eventually takes another step, whereas a sleeping process does not.

Our interest in this kind of failure stems partly from the practical problem 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 sleep cannot exhibit such simple dependencies. For example, if process A waits for process B to take some action and process B sleeps, 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 sleeping has caused the system to fail by locking out A. Insisting that algorithms be robust in the face of a limited amount of sleeping gives us a formal way of studying degrees of concurrency which in turn have some implications on the overall running time of the system.

Intuitively, an algorithm is "k-robust" if it is immune to failure (sleeping) of fewer than k processes in their entry or exit regions. By this, we mean that if a total of fewer than k processes go to sleep in either region, the system continues to operate properly -- other processes wanting to enter their critical sections eventually get to do so, and the algorithm continues to exhibit the same fairness conditions that it did previously.

At first sight, the concepts of robustness and fairness, say FIFO ordering, appear to be contradictory. Robustness says that if one process sleeps in its trying region, the system must continue to function, so other processes which later enter their trying regions will enter their critical regions ahead of the sleeper. That, however, violates usual definitions of FIFO ordering (such as (C5) in Section 3). One might simply exempt sleeping processes from fairness constraints, but the resulting conditions are impossible to implement because of the fact that sleeping 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 regions but rather by the order in which they become enabled to enter their critical regions. By "enabled", 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 enabled, 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 entry protocol. The key distinction between enabling and actual entry to the critical region, which our algorithms exploit, is that a process might become enabled 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.

The main results of the paper are three robust algorithms (Theorems 4.2, 4.3, and 4.4) for solving the ℓ-critical section problem for N processes, and two lower bound results (Theorems 5.2 and 5.3). The first algorithm (Theorem 4.2) has unlimited robustness, enables processes in FIFO order, and uses $O(N^2)$ values of shared memory, assuming ℓ 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, keeping the queue 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 very strong independence conditions, and we give an $\Omega(N^2)$ lower bound (Theorem 5.2) on the size of shared memory for any algorithm satisfying such conditions.

The second algorithm (Theorem 4.3) achieves bounded waiting, is k-robust, and uses only $O(N)$ values of shared memory, assuming k and ℓ are fixed. Important ideas used in its implementation are that of a virtual process which always runs and serves as a scheduler, and a distributed redundantly-stored data base to hold essential scheduling information. We show how to implement these two concepts within our model. We also prove (Theorem 5.3) a corresponding $\Omega(N)$ lower bound.

The third algorithm (Theorem 4.4) uses ideas similar to those of the second to achieve FIFO order of enabling, k-robustness, and $O(N(\log N)^c)$ values of shared memory, where c depends only on k and ℓ, which are again assumed fixed. Theorems 4.3 and 4.4 are first steps toward the development of uniform methods for introducing robustness into non-robust synchronization algorithms, while keeping shared data requirements reasonably low.

Two other results appear in the paper. Theorem 4.1 describes bounds achievable for the ℓ-critical section problem using the "bank teller" idea described above, in conjunction with various known 1-critical section algorithms. Theorem 5.1 describes a corresponding lower bound.

2. A Formal Model for Systems of Processes

A process is a triple $P = (V,X,\delta)$ where V is a set of values, X is a (not necessarily finite) set of states partitioned into disjoint subsets R, T, C and E, where R is nonempty, and the transition function δ is a total function, $\delta : V \times X \to V \times X$ with the following properties:

(a) $x \in R$, $v \in V$ imply $\delta(v,x) \in V \times (T \cup C)$,
(b) $x \in T$, $v \in V$ imply $\delta(v,x) \in V \times (T \cup C)$,
(c) $x \in C$, $v \in V$ imply $\delta(v,x) \in V \times (E \cup R)$,
(d) $x \in E$, $v \in V$ imply $\delta(v,x) \in V \times (E \cup R)$.

The set V is referred to as the *shared variable*, X is the set of *local states* of process P. R, T, C and E are the *remainder region*, *trying region*, *critical region*, and *exit region* of P, respectively. A transition from (v,x) to $\delta(v,x)$ is a *step* of process P.

Thus, processes are deterministic. Conditions (a) and (c) say that a process can leave its remainder region or critical region at any time on its own, but (b) and (d) indicate that the same is not necessarily the case for the trying and exit regions. We "abstract away" all program steps executed by a process while in its remainder and critical regions, treating only the protocols explicitly.

For $N \varepsilon \mathbb{N}$, [N] denotes $\{1,\ldots,N\}$. For $N \varepsilon \mathbb{N}$, a *system of N processes* is a 2N + 1 - tuple $S = (V,X_1,\ldots,X_N,\delta_1,\ldots,\delta_N)$ where for each $i \varepsilon [N]$, $P_i = (V,X_i,\delta_i)$ is a process. The remainder, trying, critical and exit regions of process P_i are denoted by R_i, T_i, C_i, and E_i, respectively. A system of N processes in which $E_i = \phi$ for all $i \varepsilon [N]$ has *null exit regions*.

An *instantaneous description* (i.d.) of S is an N + 1 - tuple $q = (v,x_1,\ldots,x_N)$, where $v \varepsilon V$ and $x_i \varepsilon X_i$ 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 $\delta_i(v,x_1,\ldots,x_N) = (v',x_1,\ldots,x_{i-1},x',x_{i+1},\ldots,x_N)$, 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, $(v,x_1,\ldots,x_N) \varepsilon R_i$ if and only if $x_i \varepsilon R_i$.

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 a "computation" of system S, when applied to any i.d. q of S. Namely, if $h = h_1,\ldots,h_k$ is a finite schedule for S, then

$r(q,h) = \delta_{h_k}(\delta_{h_{k-1}}(\ldots\delta_{h_1}(q)\ldots))$ is the *result* of applying schedule h to i.d. q. I.d. q' is *reachable* from q *via* schedule h provided $r(q,h') = q'$ for some prefix h' of h. I.d. q' is *reachable* from q provided q' is reachable from q via some finite schedule h. Let $q \xrightarrow[S]{h} p$ mean $p = r(q,h)$.

P_i is *critical-able* (resp. *remainder-able*) at q if $r(q,i) \varepsilon C_i$ (resp. R_i); in either case, P_i is *able* at q. P_i is *critical-enabled* (resp. *remainder-enabled*) at q provided for all finite schedules h not containing i, P_i is critical-able (resp. remainder-able) at r(q,h). In either case, P_i is *enabled* at q. Let CEn_i (resp. REn_i) denote $\{q : P_i$ is critical-enabled (resp. remainder-enabled) at $q\}$.

Thus, the enabled i.d.'s are those in which a process is not required to execute any protocol in order to proceed to its critical or remainder region. Unlike R_i, T_i, C_i and E_i, a process i can be caused to enter CEn_i or REn_i by steps of other processes. Thus, CEn_i and REn_i can be thought of as describing "passive" belonging to the critical and remainder regions respectively. Note that a process can reach the critical or remainder region without becoming enabled.

We write *T(q)* for $\{i : q \varepsilon T_i\}$, and use similar notation for other sets of i.d.'s.

Process P_i *halts* in schedule h if i appears only finitely often in h. If P_i halts in h and q is an i.d., we define final(i,q,h) to be the internal state of process i when it halts. Formally, *final(i,q,h)* = y if there exists an i.d. $q' = (v,y_1,\ldots,y_N)$, schedules h_1, h_2 with h_1 finite and $h = h_1h_2$ such that h_2 contains no occurrence of i, $r(q,h_1) = q'$ and $y_i = y$. Process P_i *sleeps* in h from q provided i halts in h and final(i,q,h) $\notin R_i$. P_i *fails* in h from q provided i halts in h and final(i,q,h) $\varepsilon T_i \cup E_i$. *Fail(q,h)* = $\{i : P_i$ fails in h from $q\}$. If $k \varepsilon \mathbb{N}$, then h is *k-admissible* from q provided $|Fail(q,h)| < k$.

3. Properties of Systems

In this section, S denotes a system of N processes, q an i.d. and k, ℓ natural numbers. We consider the following conditions.

(C1) ℓ-Exclusion

q *violates ℓ-exclusion* if $|C(q)| > \ell$.
S *satisfies ℓ-exclusion from* q if no i.d. reachable from q in S violates ℓ-exclusion.

We next define "(k, ℓ)-deadlock", a central concept for this paper. Intuitively, as long as fewer than k processes fail (in their trying or exit regions), the system should continue to "accomplish something." Tasks to be accomplished are permitting processes to leave their trying regions for their critical regions (as long as there is room available), and permitting processes to leave their exit regions for their remainder regions. (Our definition treats the two protocols symmetrically; this amounts to thinking of the remainder region as a second critical region, but with a trivial bound of N on the number of processes which can coexist there.) The possibility of failure necessitates careful formulation of these tasks. The system might "allocate" a slot in the critical region to a process which subsequently fails before actually advancing to its critical region. Technically, there are still unoccupied slots in the critical region, but the system can do no more to cause them to become occupied. Thus, rather than regard the critical region as full when all slots are actually occupied, we will instead regard it as full when all slots are either occupied or else allocated.

Thus, the definition of deadlock requires a preliminary (abstract) definition of allocation. The simplest such definition would allocate slots to *all* sleepers in their trying regions. (We need only consider allocation of slots to sleepers, since all other processes will actually advance to occupy their slots.) However, our algorithms have a stronger property: slots actually become allocated to groups of processes in the sense that members of those groups can take possession of their slots simply by executing one step:

Let $G \subseteq T(q)$ (resp. $E(q)$). G is *C-group-enabled* (resp. *R-group enabled*) at q provided for all schedules h in which each $i \in G$ appears at least once, at least $|G|$ distinct processes go directly from trying region to critical region (resp. from exit region to remainder region) in h applied from q. (Thus, we permit a process not in G to prevent one in G from entering its critical region by entering in its place, but that is all the damage such a process can do.) *C-allocation(q)* (resp. *R-allocation(q)*) = max $\{|G| : G$ is C-group-enabled (resp. R-group-enabled) at $q\}$.

(C2) (k,ℓ)-Deadlock-Free

Schedule h *exhibits (k,ℓ)-deadlock* from q provided (a)-(d) hold.

(a) h is infinite and k-admissible from q.

(b) No process changes regions in h applied from q.

(c) C-allocation and R-allocation do not change when h is applied from q.

(d) At least one of (d1) and (d2) holds.

(d1) $|T(q)| >$ C-allocation(q) and C-allocation(q) $+ |C(q)| < \ell$.

(d2) $|E(q)| >$ R-allocation(q).

S *is (k,ℓ)-deadlock-free* from q provided there do not exist q' reachable from q and schedule h such that h exhibits (k,ℓ)-deadlock from q'.

Condition (d) above states that a situation in which no progress is occurring is considered to be "deadlock" when there remain unallocated slots in the critical region as well as processes in the trying region to which no slot is allocated. Also, we consider such a situation to be "deadlock" when there are processes in the exit region to which no "slot" in the remainder region is allocated. The case where k=1 corresponds to no failure and thus formalizes the intuitive notion of ℓ-deadlock alluded to in the Introduction.

We next define three increasingly strong fairness properties. For each property, there are two versions, one useful for the more usually considered case where no failure occurs, and the other, weaker version (defined in terms of "enabling") useful for our new setting.

Finite schedule h is a *cycling schedule* for P_i from q if $i \in R(q)$ and $i \in C(q')$ where $q' = r(q,h)$. P_i *cycles n times* during h from q provided h can be written as $k_0h_1k_1h_2 \ldots k_{n-1}h_nk_n$ where each h_j, $1 \leq j \leq n$, is a cycling schedule for P_i from $r(q, k_0h_1 \ldots k_{j-1})$. Similarly, P_i *cycles ∞ times* during h from q provided $h = k_0h_1k_1h_2 \ldots$ and each h_j, $1 \leq j$, is a cycling schedule for P_i from $r(q, k_0h_1 \ldots k_{j-1})$.

Finite schedule h is an *enabling schedule* for P_i from q if $i \notin CEn(q) \cup REn(q)$ and $i \in CEn(q') \cup REn(q')$, where $q' = r(q,h)$. P_i *becomes enabled* in h from q provided $h = h_1h_2h_3$, with h_2 an enabling schedule for P_i from $r(q,h_1)$. P_i *makes progress* in h from q provided P_i either changes regions or becomes enabled in h from q.

P_i *b-waits* (resp. *b-waits for enabling*) for P_j in h from q, where $b \in \mathbb{N} \cup \{\infty\}$ provided $i \in T(q) \cup E(q)$ (resp. $i \in (T(q) - CEn(q)) \cup (E(q) - REn(q))$), P_i does not change regions (resp. make progress) in any proper prefix of h from q, and P_j cycles b times during h from q. (We explain our consideration of *proper* prefixes only, after the definition of FIFO enabling (C5').)

(C3) k-Finite Waiting (resp. *(C3') k-Finite Waiting for Enabling*)

S satisfies *k-finite waiting* (resp. *k-finite waiting for enabling*) from q provided there do not exist q' reachable from q, k-admissible schedule h and processes P_i and P_j such that P_i ∞-waits (resp. ∞-waits for enabling), for P_j in h from q'.

In the literature (including [BFJLP], a property called "no lockout" is usually formalized instead of (C3). "No lockout" is generally expressed in terms of each process making eventual progress. This requirement really includes two different conditions - a condition which states that the system as a whole continues to make progress, and a condition which states that no process is indefinitely discriminated against in favor of other processes. Here, these two conditions are treated separately, as (C2) and either (C3) or (C3').

(C4) Bounded Waiting (resp. (C4') Bounded Waiting for Enabling)

S satisfies b-bounded-waiting (resp. b-bounded-waiting for enabling) from q ($b \in \mathbb{N}$), provided there do not exist q' reachable from q, schedule h and process i and j such that P_j b+1-waits (resp. b+1-waits enabling) for P_j in h from q'.

S satisfies bounded waiting (resp. bounded waiting for enabling) from q if S satisfies b-bounded waiting (resp. b-bounded waiting for enabling) from q for some values of $b \in \mathbb{N}$.

(C5) FIFO (resp. (C5') FIFO Enabling)

S is FIFO (resp. FIFO enabling) from q if S satisfies 0-bounded waiting (resp. 0-bounded-waiting for enabling) from q.

The definition of FIFO enabling as 0-bounded waiting for enabling explains our use of proper prefixes only in defining b-bounded waiting for enabling. Namely, violation of our intuitive notion of FIFO enabling occurs if a process P_i remains in its trying region and does not become enabled, while another process P_j enters its trying region and progresses to a point where it becomes enabled. This computation, followed by a single step of P_j, causes P_j to go to its critical region. The resulting augmented schedule violates our formal definition of FIFO enabling. However, no such violation would occur if we considered all prefixes in our definition, since P_i could become enabled by the last step of P_j.

(Note the slight difference in style between definition (C3) and definitions (C4) and (C5). k-admissibility is not required for (C4) or (C5) because violations of bounded waiting conditions always occur in a finite schedule.)

Finally, one might imagine an algorithm which stores the entire queue of waiting and critical processes in the shared variable. A process in any of the first ℓ positions of the queue is permitted to enter the critical region. Such an 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 its entry to the trying region and remainder region. (No exit region is required.) Such an algorithm satisfies (C1), (C2) and (C5'), but it requires too much space in the shared variable. However, some algorithms of interest have the property that they "simulate" the above algorithm without keeping the entire queue in the variable. This idea is captured in the final property (C6).

S is ℓ-full from q provided for all q' reachable from q, it is the case that $|CEn(q')| = \min(\ell - |C(q')|, |T(q')|)$.

(C6) ℓ-Queuelike

S is ℓ-queuelike from q provided S satisfies ℓ-exclusion from q, has null exit regions, is ℓ-full from q and is FIFO enabling.

Note that (C6) implies (C1), (C2), (C3'), (C4'), and (C5'). In particular, (C6) for a particular ℓ implies (C2) for the same ℓ and arbitrary k.

4. Upper Bound Results

It is straightforward to formalize in our model the construction, described in the Introduction, of an ℓ-exclusion algorithm from a 1-exclusion algorithm. Roughly, for S any system of N processes, $q \in \prod_{i \in [N]} R_i$, $\ell \in \mathbb{N}$, let S' be the new system of N processes constructed by the transformation in Section 1. Clearly, S' has null exit regions. The variable of S' is the same as that of S, augmented with COUNT. q', the starting i.d. of S', is the same as q with COUNT(q') initialized at 0. A total mapping τ is defined from reachable i.d.'s of S' to those of S and another mapping σ is defined from "computations" of S' to those of S. (Given i.d. q_1 of S' and schedule h, schedule $\sigma(q_1,h)$ is obtained by deleting from h (1) all occurrences of process numbers corresponding to S' transitions which are busy-waiting for COUNT to become less than ℓ, and (2) all occurrences of process numbers corresponding to moving from critical to remainder region in S'). These mappings are used to prove the next lemma.

A system S is said to be order preserving from q provided the order of entry to the critical region is the same as the order of return to the remainder region.

Lemma 4.1. Let S be any system of N processes. $q \in \prod_{i \in [N]} R_i$, $\ell \in \mathbb{N}$, $b \in \mathbb{N} \cup \{0\}$. Assume S is (1,1)-deadlock-free and order preserving from q. Then S' satisfies ℓ-exclusion (C1) and is $(1,\ell)$-deadlock-free from q' (C2). Furthermore, (a) and (b) hold.

(a) If S satisfies 1-finite waiting (C3) from q, then S' satisfies 1-finite waiting from q'.

(b) If S satisfies b-bounded waiting ((C4) or (C5)) from q, then S' satisfies b-bounded waiting from q'.

Theorem 4.1. Let ℓ, $N \in \mathbb{N}$. For each of the following, there exist S a system of N processes and q an i.d. such that S has null exit regions, satisfies ℓ-exclusion (C1) and is $(1,\ell)$-deadlock-free (C2) from q, and such that the following hold:

(a) S satisfies 1-finite waiting (C3) from q, and $|V| \leq (\ell+1)(\lfloor \frac{N}{2} \rfloor+9)$.

(b) S satisfies 1-bounded waiting (C4) from q, and $|V| \leq (\ell+1)(N+3)$.

(c) S is FIFO (C5) from q, and $|V| \leq (\ell+1)(N+7)$.

Proof. By Lemma 4.1 , together with the systems described in [BFJLP,CH2]. Those systems have starting i.d.'s q in $\bigcap_{i\varepsilon[N]} R_i$, are (1,1)-deadlock-free and order-preserving from q and have appropriate fairness properties and space bounds. ((c) uses an improvement of the result of [CH2].)

□

The remaining algorithms will be presented in an Algol-like, Pascal-like language similar to that in [CH2]. Added to the usual sequential programming constructs are two synchronization statements, LOCK and UNLOCK; LOCK locks the shared variable for the private use of the program executing the statement, while UNLOCK releases it. We consider a general translation of systems of LOCK-UNLOCK programs into our basic model, where each program i gets translated into a process P_i. All computation of a program occurring after a LOCK and before the next UNLOCK is intended to be included within a single test-and-set in the translated system. For translation of a system of programs to be possible, the executions of the system must satisfy certain conditions.

First, LOCK and UNLOCK statements are "dynamically paired" : the first (if any) synchronization statement executed as well as the next (if any) synchronization statement following an UNLOCK, is a LOCK, and for each LOCK executed, say of program i, there is a next synchronization statement executed, and it is an UNLOCK (of program i). Furthermore, all computation involving the shared variable must be included within such LOCK-UNLOCK pairs. A program i's local computation will not be constrained. (Where it occurs outside of LOCK-UNLOCK pairs, the translation will incorporate its result into the next (if any) test-and-set of process P_i. If program i loops forever without executing another LOCK statement, it is treated as if it had stopped performing computation immediately after its previous UNLOCK statement.)

With these restrictions, translation proceeds as follows. Assume a system of N processes is given, together with their initializations, and consider program i. The states of process P_i are identified with finite vectors consisting of an UNLOCK statement (or the START statement) of program i, together with values of all the local variables of program i. The values of V are just those of the shared variable of the given system of programs. The transition function δ_i is defined as follows. Consider the case where x describes a configuration of program i and v is a value of the shared variable such that x and v are simultaneously reachable via executions of the given system, and also program i started in configuration x reaches a LOCK statement. Then define $\delta_i(v,x) = (w,y)$, where w and y are the value of the shared variable and the configuration of program i respectively, which result when program i is run from configuration x, using v for the value of the shared variable, just until it executes an UNLOCK statement. (We know that such an UNLOCK statement is reached because of the given restrictions.) If x and v are not simultaneously reachable then define $\delta_i(v,x)$ arbitrarily. If program i started in configuration x never reaches a LOCK statement, define $\delta_i(v,x) = (v,x)$.

In this translation, quite complex LOCK-UNLOCK computation might be translated into a single test-and-set. Such computation might include apparent changes to the shared variable, followed by tests and computation involving the new value, followed in turn by further changes to the shared variable. Furthermore, the shared variable might be organized as several component variables and computation might be expressed in terms of these components. The apparent earlier changes to the shared variable are overwritten before the variable is released for use by other processes.

One cannot effectively determine, in general, whether a given LOCK-UNLOCK program satisfies the restrictions needed for translation. Moreover, even if it is known that the restrictions are satisfied, finding a translation is not effective, in general. However, for the algorithms in this paper, it is not difficult to see that the restrictions are satisfied and to find a translation.

Next, we present our $O(N^2)$ queuelike algorithm.

Theorem 4.2. Let ℓ, $N \varepsilon \mathbb{N}$. There exist S a system of N processes and q an i.d. such that S is ℓ-queuelike (C6) from q and $|V|$ is $O(N^2)$. (The constant of proportionality is $\binom{2\ell-1}{\ell}(\ell+1)^2$.)

Proof. If $\ell \geq N$, a trivial algorithm suffices. Thus in what follows we can assume $N > \ell$.

We imagine $(\ell+1)N$ reusable tickets to the critical region, with N tickets (numbered 1 to N) of each of $\ell+1$ colors (numbered 0 to ℓ). 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. From time to time, a ticket becomes validated. If a process holds a valid ticket, it can enter its critical region. At any time, exactly ℓ tickets are valid;

whenever there are fewer than ℓ processes in the system, some of the valid tickets will not currently be issued.

We preserve FIFO enabling (C5'); thus we validate tickets 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 ℓ of color 0 are valid.

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 ℓ processes in the system, Valid leads Issue, having already validated the next ticket(s) to be issued. When there are more than ℓ processes, Issue leads Valid, indicating that there are processes in the trying region waiting to be allowed to enter their critical regions. When there are exactly ℓ processes, the two pointers coincide, indicating that the ℓ active processes hold the ℓ valid tickets. Valid can lead Issue by at most ℓ, while Issue can lead Valid by at most N-ℓ.

Although FIFO enabling holds, some processes may get "skipped over" for actual order of entry to their critical regions (if they sleep or go slowly, for example). Thus, valid tickets may become 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. (Symmetrically, if there are valid tickets which are not currently 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 already issued or validated. The fact that it is still always possible to select a new color when needed is a consequence of the first consecutivity condition of the previous paragraph.

The 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.

(a) Issue, (the number and color of) the ticket most recently issued.
(Initially, (N,ℓ) appears.)

(b) Valid, the ticket most recently validated.
(Initially, (ℓ,0) appears.)

(c) Quant(i), $0 \leq i \leq \ell$, the quantity of each color represented by the ℓ valid tickets.
(Initially, Quant(0) = ℓ and all others are 0.)

The bound on variable size holds because the information in (c) represents only $\binom{2\ell-1}{\ell}$ distinct possibilities.

Considerable information can be determined from the value of the variable only. In particular, the variable suffices to determine newcolor, a color different from those of all the valid and issued tickets. It determines whether Issue leads Valid, or vice versa, or whether they coincide. (The hypothesis N > ℓ is used here.) If Issue leads Valid, then the values of these two pointers together suffice to determine all the intervening tickets, since N > ℓ. Thus, because of the consecutivity properties previously discussed, the variable suffices to determine the set of invalid issued tickets. These capabilities are used in the program below. (All processes are identical.)

```
Process i    Local variable: Ticket
Trying Protocol
   START;
   LOCK;

   /*Take the next ticket.*/

   IF Issue.Number ≠ N
   THEN Ticket := (Issue.Number+1,Issue.Color)
   ELSEIF Valid leads Issue
   THEN Ticket := (1,Valid.Color)
   ELSE Ticket := (1,Newcolor);

   /*Update the shared variable.*/

   Issue := Ticket;
   UNLOCK;

   /*Wait until your ticket is valid.*/

W: LOCK;
   IF your ticket is invalid
   THEN [UNLOCK; GOTO W]
   ELSE UNLOCK;
Exit Protocol
   LOCK;

   /*Validate a new ticket.*/

   IF Valid.Number ≠ N
   THEN Valid := (Valid.Number+1,Valid.Color)
   ELSEIF Issue leads Valid
   THEN Valid := (1,Issue.Color)
   ELSE Valid := (1,Newcolor);

   /*Update quantity information.*/

   Quant(Valid.Color) := Quant(Valid.Color)+1;
   Quant(Ticket.Color) := Quant(Ticket.Color)-1;
   UNLOCK;
```

□

Systems with a Supervisor

Up until now, we have been considering only systems of processes in which all processes are treated uniformly -- every process has a critical region and a protocol, every process is permitted to halt in its remainder region, etc. Adding another process to such a system can make implementing a synchronization algorithm far easier, for the new "supervisor" process can keep track of all the necessary scheduling information, assuming of course that the supervisor never fails.

Our interest in systems with a supervisor stems from the fact that under certain conditions the supervisor can be eliminated to yield an ordinary N-process system of the kind studied in this paper while preserving desired properties. We use the strategy of first constructing a system with a supervisor and then eliminating the supervisor to obtain two ℓ-exclusion algorithms: an $O(N)$ space method with bounded waiting for enabling, and an $O(N(\log N)^c)$ space method with FIFO enabling.

To make these notions more precise, we define a system of N worker processes and a supervisor to be a (2N+3)-tuple

$$S = (V, X_1, \ldots, X_{N+1}, \delta_1, \ldots, \delta_{N+1})$$

where $P_i = (V, X_i, \delta_i)$ is a process, $1 \leq i \leq N+1$. $P_1, \ldots, P_N$ are the worker processes and P_{N+1} is the supervisor. S can be regarded as an ordinary system of N+1 processes except that only the worker processes have regions. All of the notions of schedule, halting, instantaneous description, etc. which do not concern the regions apply unchanged to S, and the definitions of critical-enabled and remainder-enabled apply to $P_1, \ldots, P_N$. The significant change comes in the definition of k-admissible schedule, for we now restrict it to be one in which fewer than k worker processes fail and the supervisor does not halt unless all the workers halt. Properties (C1)-(C6) and (C3')-(C5') extend directly to $P_1, \ldots, P_N$ using this new notion of k-admissibility.

Supervisor Elimination

In this section, we show how, under certain conditions, to replace a system of N worker processes and a supervisor by an ordinary N-process system. The new system preserves all of our properties, (in particular, (C1), (C2), (C4'), and (C5')), and the space requirement increases by at most a constant factor (for fixed k and ℓ).

We eliminate the supervisor by having the worker processes simulate it. No individual worker can have sole responsibility for carrying out the simulation, since any one process might fail or sleep. Rather, any worker with access to the current supervisor state might simulate the next step of the supervisor by updating the supervisor state and shared variable accordingly. Thus, an execution of the new system simulates a computation by the original system in which the supervisor takes at most one step between every pair of worker steps. Assuming infinitely many supervisor steps get simulated, this will be a valid computation of the original system.

The most obvious place to store the current supervisor state is in shared memory. Then every worker process can simulate one step of the supervisor each time it performs a test-and-set. The only difficulty is that the size of shared memory increases too much if the supervisor has many states.

An alternate strategy is to have a worker process P_i keep the supervisor state in its internal memory and to simulate a supervisor step each time it runs. This has two difficulties. First, P_i might fail, in which case the supervisor would stop. Second, P_i might halt in its remainder or critical region, again stopping the supervisor.

Both of these problems are circumvented by storing the supervisor state redundantly in several workers executing their protocols. Whenever any of them runs, it updates the supervisor state and sends the new state to the other processes. When it wants to enter its critical or remainder region, it sends its state to some other active process. This strategy works as long as there are always enough active processes, for at most k-1 can fail simultaneously.

Our method is to combine these two strategies. The first is used whenever the number of active processes falls below a certain threshold c_1. Otherwise, the current supervisor state is stored redundantly in the internal memories of k distinct workers in their trying regions. By assumption, not all of them can fail. The supervisors we simulate have the property that the number of reachable supervisor states is small when there are few active processes, so storing the state then in shared memory does not increase execssively the number of shared values.

We now describe, for the case where the supervisor state is stored redundantly, how the redundant copies of the current supervisor state are managed and updated. Assume initially that k distinct processes each have a copy of the current supervisor state stored in its internal memory. Any of them has the information necessary to simulate a step of the supervisor and does so on its next step, modifying the simulated shared memory accordingly (which is stored as a component of the real shared variable). It must now inform k-1 other processes of the new state, or k other processes if it itself is about to enter its critical region.

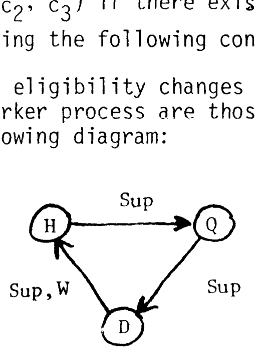
There are two ways another process can obtain the new state. First, a process having the previous supervisor state can update its copy knowing only enough about the new state to distinguish it from the other possible new states which could be reached in one step of the supervisor on other values of the shared variable. This distinguishing information is placed in shared memory by the process which simulated the supervisor step. The old value of the shared variable suffices to determine the new supervisor state from the old, but if $\{\delta_{N+1}(v,x) : v \in V\}$ is much smaller than V, significantly less information suffices, thereby saving on memory.

A process not holding either the current or previous supervisor state gets the current state by receiving a "message" from some stateholder containing the complete state. The state is encoded in binary and sent one bit at a time. The sender and receiver use a simple protocol for the orderly transmission of this information. Since either sender or receiver might fail, a given transmission is not guaranteed to terminate successfully. By having k senders and 2k-1 potential receivers, with each sender transmitting to each receiver over k·(2k-1) independent "channels", we are guaranteed that at lease k receivers eventually get the updated state, for at least one sender and k receivers do not fail.

With this motivation, we list the conditions on a system with a supervisor that permit the simulation to be carried out.
Let $S = (V,X_1,\ldots,X_{N+1},\delta_1,\ldots,\delta_{N+1})$ be a system of N workers and a supervisor. As usual, $P_i = (V,X_i,\delta_i)$ is a process, $1 \leq i \leq N+1$. $P_1,\ldots,P_N$ are the workers and P_{N+1} is the supervisor. Let q_0 be the initial i.d., and say q is reachable if it is reachable from q_0.

For each reachable i.d. q, assume a partition of $\{1,\ldots,N\}$ into three sets H(q), Q(q), and D(q). H(q), the helpers, are those processes which are eligible to help with the task of storing and updating the supervisor state, Q(q), the quitters, are processes becoming ineligible to help, and D(q), the drones are the other processes. A system S with initial i.d. q_0 is k-simulatable (with constants c_1, c_2, c_3) if there exists such a partition satisfying the following conditions.

(R1) The only eligibility changes that can occur for a worker process are those described by the following diagram:



Here, an arrow labelled by Sup denotes a change in eligibility permitted by a single step of the supervisor, and an arrow labelled by W denotes a change permitted on a move of a worker. For example, this condition would be violated by the existence of reachable i.d.'s q, q' and workers P_i, P_j for which $q' = \delta_j(q)$, $i \in Q(q)$, and $i \in D(q')$.

(R2) A worker can tell, knowing only its internal state and the shared variable, which eligibility class it is in, i.e. there exists a function $elig_i : V \times X_i \to \{'H', 'Q', 'D'\}$ such that for any reachable i.d. $q = (v,x_1,\ldots,x_N)$, $i \in H(q)$ (resp. $Q(q)$, $D(q)$) iff $elig_i(v,x_i) = 'H'$ (resp. 'Q', 'D').

(R3) $H(q) \cup Q(q) \subseteq T(q)$.

(R4) If i is critical-able at q, then $i \in D(q)$.

(R5) The supervisor can tell, knowing only its internal state and the value of the shared variable, the cardinality of T(q), i.e. there exists a function active: $V \times X_{N+1} \to [N]$ such that for any reachable i.d. $q = (v,x_1,\ldots,x_{N+1})$, $|T(q)| = active(v,x_{N+1})$.

(R6) If $|T(q)| > c_1$ then $|H(q)| \geq 2k-1$. This insures that sufficiently many helpers are available when many processes are in their trying protocols.

(R7) For each $v \in V$, $|\{x \in X_{N+1} : active(v,x) \leq c_1\}| \leq c_2$. c_2 bounds the amount of memory needed to store the supervisor state when it is kept in shared memory.

(R8) For each $x \in X_{N+1}$, $|\{\delta_{N+1}(v,x) : v \in V\}| \leq c_3$. c_3 bounds the amount of memory needed to encode the next step of the supervisor so that processes holding the previous state can update it.

Let S be a system of N workers and a supervisor as above, and let $S' = (V',X_1',\ldots,X_N',\delta_1',\ldots,\delta_N')$ be an ordinary system of N processes $P_1',\ldots,P_N'$, where each $P_i' = (V',X_i',\delta_i')$ is a process with remainder, trying, critical, and exit regions R_i', T_i', C_i', and E_i', respectively. Let q_0' be the initial i.d. of S'. Let τ be a mapping from reachable i.d.'s of S' to i.d.'s of S, and let σ: (reachable i.d.'s of S') × $[N] \to [N+1]^+$. Each step of S' simulates one or more steps of S; σ tells which ones. We extend σ to arbitrary schedules in its second argument. $\sigma(q',h')$ is then the schedule of S simulated by S' when S' is started in i.d. q' and run according to schedule h'.

(i) $\sigma(q',\lambda) = \lambda$, the empty schedule;

(ii) $\sigma(q',hj) = \sigma(q',h).\sigma(r(q',h),j)$ for h finite;

(iii) $\sigma(q',h) = \lim_{h_0<h} \sigma(q',h_0)$ for h infinite.
("<" denotes finite prefix.)

S' k-*simulates* S if (S1) - (S4) hold:

(S1) τ preserves regions of $P_1',\ldots,P_N'$, e.g. $i \in C'(q')$ iff $i \in C(\tau(q'))$. Also, $\tau(q_0') = q_0$.

(S2) The following diagram commutes:

$$\begin{tikzcd}
q' \arrow[r, "j"', "S'"] \arrow[d, "\tau"'] & p' \arrow[d, "\tau"] \\
q \arrow[r, "\sigma(q',j)", "S"'] & p
\end{tikzcd}$$

(S3) $\sigma(q',j) \in \{j,(j, N+1)\}$ for all reachable i.d.'s q' in S' and $j \in [N]$.
Thus, each step of P_j' simulates a step of P_j possibly followed by one step of the supervisor, P_{N+1}.

(S4) $\sigma(q',h')$ is k-admissible from $\tau(q')$ in S for every reachable i.d. q' and k-admissible schedule h' from q' in S'.

Intuitively, the system S' maintains a representation of each i.d. of S, given by the mapping τ. (S1) and (S2) ensure that the simulation is faithful and that τ preserves reachability. (S3) requires the simulation to be step by step with respect to the worker processes and prevents the supervisor from running too fast. (S4) ensures that infinitely many supervisor steps are simulated as long as the system S' remains active.

Lemma 4.2. Let S be a system of N workers and a supervisor and S' an ordinary N-process system which k-simulates S. Let q_0 and q_0' be the initial i.d.'s of S and S', respectively. Each of properties (C1), (C2), (C4'), and (C5') holds for S' if it holds for S.

(Note: Actually, all of our properties are preserved, but it is only these four which we require for the rest of our algorithms.)

Proof. For each property, we show that if it fails for S', then it fails for S.

Suppose (C1) does not hold for S'. Then there is a reachable i.d. q' with $|C'(q')| > \ell$. Let h' be a finite schedule such that $q' = r(q_0',h')$ in S'. Then $h = \sigma(q_0',h')$ is a finite schedule such that $\tau(q') = r(q_0,h)$ in S. Since τ preserves regions, $|C(\tau(q'))| > \ell$, so (C1) fails for S.

Suppose (C2) does not hold for S'. Then there exists a reachable state q' and an infinite k-admissible schedule h' for which conditions (a)-(d) in the definition of (C2) hold. Let $q = \tau(q')$ and $h = \sigma(q',h')$. h is k-admissible from q, so (a) holds from q. (b) holds from q since τ preserves regions. Since no process changes regions when h is applied from q, the C-allocation and R-allocation can only increase, and that can happen only a finite number of times. Choose h_1', h_2' so that $h' = h_1'.h_2'$. Let $p' = r(q',h_1')$, $p = \tau(p')$, $h_1 = \sigma(q',h_1')$, and $h_2 = \sigma(p',h_2')$. h_1' can be chosen to insure that (c) holds for p and h_2. (a) and (b) hold for p and h_2 since they held for q and h. We note that if G is C- (R-) group-enabled at $\tau(p')$ in S, then G is also C- (R-) group-enabled at p' in S', but the converse does not necessarily hold. Thus, C-allocation(p') $\geq$ C-allocation(τ(p')) and R-allocation(p') $\geq$ R-allocation(τ(p')). Hence, C-allocation(q') = C-allocation(p') $\geq$ C-allocation(p) and R-allocation(q') = R-allocation(p') $\geq$ R-allocation(p), so (d) holds for p and h_2. Hence, schedule h_2 exhibits (k,ℓ)-deadlock from p. Since p is reachable, (C2) fails for S.

Suppose S' fails to have b-bounded waiting ((C4') or (C5')). Then for some i, j, there is a reachable i.d. q' and a finite schedule h' such that P_i' (b+1)-waits for enabling for P_j' in h' from q'. Thus, P_j' cycles b+1 times during h' from q', and P_i' does not make progress in any proper prefix of h' from q'. Choosing such an h' of minimal length, we must have $h' = g'j$ and $j \in C'(r') - C'(p')$, where $r' = r(q',h')$ and $p' = r(q',g')$. Let $g = \sigma(q',g')$ and define q, p, r according to the diagram:

$$\begin{tikzcd}
q' \arrow[r, "g'"] \arrow[d, "\tau"'] & p' \arrow[r, "j"] \arrow[d, "\tau"] & r' \arrow[d, "\tau"] \\
q \arrow[r, "g"] & p \arrow[r, "\sigma(p',j)"] & r
\end{tikzcd}$$

Since P_i' (b+1)-waits, $i \in T'(q')-CEn'(q')$ or $i \in E'(q')-REn'(q')$, and P_i' does not make progress in any proper prefix of h' from q'. Hence, $i \in T(q)-CEn(q)$ or $i \in E(q)-REn(q)$, and P_i also does not make progress during g from q, nor is P_i enabled at p.

By (S1), $j \in C(r)-C(p)$. If $\sigma(p',j) = j$, then P_i (b+1)-waits for P_j in schedule gj from q. Otherwise, $\sigma(p',j) = (j,N+1)$, so for some s,

$$p \xrightarrow{j} s \xrightarrow{N+1} r$$

Then $j \in C(s)$, since P_j cannot change regions on a supervisor step. Again, gj is a schedule in which P_i $(b+1)$-waits for P_j from q.

In either case, S fails to have b-bounded waiting since q is reachable. Thus, each of (C4') and (C5') fails for S' only if it fails for S.

□

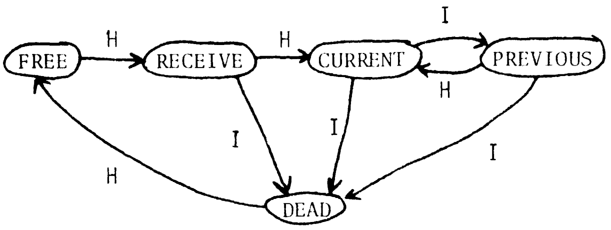
Lemma 4.3. Let S be a k-simulatable system with constants c_1, c_2, c_3 of N workers and a supervisor with value set V. There exists an ordinary N-process system S' with value set V' which k-simulates S. Moreover, $|V'| \leq d \cdot |V|$, where d is a constant depending only on k, c_1, c_2, c_3.

Proof. As usual, let $S = (V, X_1, \ldots, X_{N+1}, \delta_1, \ldots, \delta_{N+1})$, and let $P_i = (V, X_i, \delta_i)$, $1 \leq i \leq N+1$. We construct $S' = (V', X_1', \ldots, X_n', \delta_1', \ldots, \delta_N')$, $P_i' = (V, X_i', \delta_i')$, $1 \leq i \leq N$, by exhibiting the algorithm for an arbitrary process P_i'. As before, the transition function is described in a Pascal-like notation.

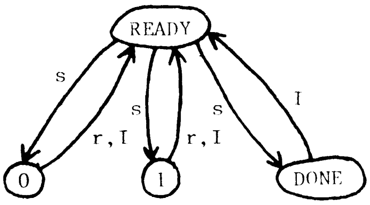
In the above intuitive outline of the algorithm, we mentioned a mechanism for one process to transmit a copy of the supervisor state to another. In greater detail, we have 3k-1 slots. Associated with each slot s is a word of shared memory, status(s), which can take on one of five values: FREE, RECEIVE, CURRENT, PREVIOUS, and DEAD. A slot may or may not have a worker associated with it. status(s) ≠ 'FREE' iff it has a worker, in which case status(s) indicates the role the worker is currently playing in the simulation. 'CURRENT' means the worker has the current supervisor state stored in its internal memory. 'PREVIOUS' means the supervisor state in the worker's internal memory is one step out-of-date. 'RECEIVE' means the worker does not have either a current or previous supervisor state but is trying to obtain a current copy. 'DEAD' means the worker no longer contains valid information and should release the slot on its next step.

The status word can be manipulated by either the slotholder or by the process which initiates the simulation of a supervisor step. (We think of a stage of simulation as consisting of a simulation of a supervisor step followed by the transmission of the new state to the helpers.)

Letting I denote the step initiator and H the slotholder, the possible value transitions on the status word are given by the graph:



If the holder of slot s has the current state, it will attempt to send it to slot r using "channel" ch(s,r), for each possible r. ch(s,r) is stored in shared memory and can take on one of four values: 'READY' denotes that the channel is clear and ready for the sender to transmit the next bit of its message; '0' and '1' are the two message bits; and 'DONE' denotes a successful end of transmission. The protocol used is described by a simple graph, where s represents the sender, r the receiver, and I the step initiator:



Other data in shared memory are the following:

v - the simulated value of the shared variable in S.

stateholders - the number of helpers currently holding the current supervisor state ($0 \leq$ stateholders $\leq k$).

step - an encoding of the most recent supervisor step.

xs - the current supervisor state in case active(q) $\leq c_1$; or the previous state on a step in which active(q) becomes $> c_1$.

loccur - location of current supervisor state: 'SHARE' means in xs, 'DIST' means in certain helpers' internal memories.

locprev - the previous value of loccur.

The total size of shared memory is some constant times $|V|$, for all of the variables except v are bounded by functions of k. In particular, xs takes on at most c_2 values and step requires at most c_3.

Each process has a number of local variables. Besides the obvious temporaries, the following are of interest:

s - the number of the slot held by the process, or 0 if no slot is held.

x - the current or previous supervisor state, or undefined, depending on s and status(s).

w - the internal state of the corresponding worker process.

rbuf(t) - a buffer in which the partial state received so far from slot t is accumulated.

n(t) - the number of the next bit of the supervisor state to be sent to slot t.

prevhelper - true iff the process was a helper the last time it ran.

The code makes use of some functions which are described below: $elig_i$ and active are the functions mentioned in (R2) and (R5).

δ_{N+1} is represented by three functions: newval, transno, and newstate. newval(v,x) gives the new value of the shared variable produced by the transition $\delta_{N+1}(v,x)$, and transno(v,x) gives an encoding of the new supervisor state. newstate is the corresponding decoding function. Thus, $\delta_{N+1}(v,x)$ = (newval(v,x), newstate(x,transno(v,x))). We assume that transno is such that for each $x \in X_{N+1}$, there is an integer m(x) so that $\{transno(v,x) : v \in V\} = [m(x)]$, and that transno is chosen to minimize m(x). Thus, the encoding is compact. By (R8), $m(x) \leq c_3$.

Finally, we have three operations on bit-strings. Length(z) gives the number of bits in z, bit(i,z) is the i^{th} bit of z, and $\circ$ denotes concatenation of bit-strings. We use these operations on states assuming a binary encoding of the state.

One further convention: "slots" is a constant denoting [3k-1].

The code for process P_i' appears on the following page.

Let $q_0 = (v_0, x_{0,1}, \ldots, x_{0,N+1})$ be the initial i.d. of S. The shared memory of S' is initialized by setting status(t) = 'FREE' and ch(t,t') = 'READY' for all t, t' $\in$ slots; v = v_0; stateholders = 0; step = 1; xs = $x_{0,N+1}$; and loccur = locprev = 'SHARE'. The local variables of process P_i' are initialized by setting s = 0; x = $x_{0,N+1}$; w = $x_{0,i}$; rbuf(t) = λ and n(t) = 1 for each t $\in$ steps; and prevhelper = false. All other temporaries are set arbitrarily.

The regions of S' are defined by the regions of S, that is, a state of P_i' is in R_i' (respectively T_i', C_i', E_i') iff $w \in R_i$ (respectively T_i, C_i, E_i), where w denotes (ambiguously) the contents of local variable w in that state.

Whenever P_i' takes a step, it simulates a step of P_i. It then simulates a supervisor step if either the current supervisor state is in shared memory (indicated by loccur = 'SHARE') or P_i' holds some slot s with status(s) = 'CURRENT' and in addition stateholders = k. To show (S4) holds for S', we must argue that these conditions are satisfied infinitely often as long as not every worker process halts, and hence infinitely many supervisor steps get simulated.

If loccur = 'SHARE', then another step of the supervisor is simulated as soon as any P_j' takes a step.

Assume now that loccur = 'DIST' and that a supervisor step has just been simulated. We argue that a time is reached before the next supervisor step is taken when stateholders = k. At that time, exactly k helpers (and possibly other processes as well) have the current state.

There are two cases to consider. Suppose locprev = 'SHARE'. Then all slots are either 'DEAD' or 'FREE', and all but k-1 of these eventually becomes 'FREE'. Hence, the remaining 2k slots are or will become available to the helpers. By conditions (R5) and (R6), there are at least 2k-1 helpers at this time. (R1) insures that no helper quits before the next supervisor step is simulated, and (R3) and (R4) insure no helper will enter its critical or remainder region (where it might stop). When a helper takes a slot in this situation, it picks up the old supervisor state and updates it immediately to become a current stateholder. At most k-1 helpers fail, so at least k helpers do eventually become current stateholders.

Now suppose locprev = 'DIST'. Inductively, we may assume that immediately before the current supervisor step was taken there were k helpers with the current state (as well as possibly other non-helping processes). After the step, these k processes hold the now-previous supervisor state, and some of them additionally may have entered class 'Q', thereby ceasing to be helpers. However, all of them are allowed to keep their slots (and indeed must do so), whereas each other slotholder will release its slot on its next step. Ignoring failing processes for the moment, at least 2k-1 helpers will eventually get slots. Even considering failures, at most k-1 slots can be lost to failing processes and at most k-1 helpers can fail, so at least k active helpers end up with slots. Since at least one of the previous stateholders also stays active, k helpers eventually obtain the current state. Another supervisor step is then simulated on the next step of any of these

Code for Process P_i' (Lemma 4.3)

```
REPEAT FOREVER
  LOCK;

  /*Simulate one step of $P_i$.*/

  (v,w) := $\delta_i$(v,w);

  /*Simulate supervisor step, if possible.*/

  IF loccur = 'SHARE' or
     (s≠0 & status(s) = 'CURRENT' & stateholders=k)
  THEN
    BEGIN
    xx := IF loccur = 'SHARE' THEN xs ELSE x;
    v := newval(v,xx);
    step := transno(v,xx);
    xx' := newstate(xx,step);
    locprev := loccur;
    stateholders := 0;
    IF active(v,xx') > $c_1$
    THEN

      /*New state is distributed.*/

      BEGIN
      loccur := 'DIST';
      FOR ALL t ε slots DO
        IF status(t) = 'CURRENT'
        THEN status(t) := 'PREVIOUS'
        ELSEIF status(t) ε {'RECEIVE', 'PREVIOUS'}
        THEN status(t) := 'DEAD';
      FOR ALL t, t' ε slots DO ch(t,t') := 'READY'
      END
    ELSE

      /*New state is kept in shared memory.*/

      BEGIN
      loccur := 'SHARE';
      xs := xx';
      FOR ALL t ε slots DO
        IF status(t) ≠ 'FREE'
        THEN status(t) := 'DEAD'
      END
    END;

  /*Release current slot if necessary.*/

  IF s≠0 & (status(s) = 'DEAD' or elig$_i$(v,w) = 'D'
     or (status(s) = 'PREVIOUS' & not prevhelper))
  THEN [status(s) := 'FREE'; s := 0];

  /*Take a new slot if needed and possible.*/

  IF loccur = 'DIST' & s = 0 & elig$_i$(v,w) = 'H' &
     status(t) = 'FREE' for some t ε slots
  THEN
    BEGIN
    s := t;
    status(s) := 'RECEIVE';
    for all t' ε slots do rbuf(t') := λ
    END;

  /*Service slot held, if any.*/

  IF s≠0 & stateholders < k
  THEN
    BEGIN
    IF status(s) = 'CURRENT'
    THEN
      FOR ALL t ε slots DO
        IF ch(s,t) = 'READY'
        THEN
          IF n(t) > length(x)
          THEN ch(s,t) := 'DONE'
          ELSE [ch(s,t) := bit(n(t),x);
             n(t) := n(t) + 1]
    ELSEIF status(s) = 'PREVIOUS'
    THEN
      BEGIN
      x := newstate(x,step);
      status(s) := 'CURRENT';
      IF elig$_i$(v,w) = 'H'
      THEN stateholders := stateholders + 1;
      FOR ALL t' ε slots DO n(t') := 1
      END
    ELSEIF status(s) = 'RECEIVE'
    THEN
      IF locprev = 'SHARE'
      THEN

        /*Get previous state from shared memory.*/

        [x := xs; status(s) := 'PREVIOUS']
      ELSE

        /*Receive state through some channel.*/

        FOR ALL t ε slots DO
          IF ch(t,s) = 'DONE'
          THEN

            /*Complete state has been received from
               slot t.*/

            BEGIN
            x := rbuf(t);
            status(s) := 'CURRENT';
            stateholders := stateholders + 1;
            FOR ALL t' ε slots DO n(t') := 1;
            EXIT LOOP
            END
          ELSEIF ch(t,s) ε {0,1}
          THEN

            /*Next bit has arrived.*/

            BEGIN
            rbuf(t) := rbuf(t)∘ch(t,s);
            ch(t,s) := 'READY'
            END
    END;
  prevhelper := (elig$_i$(v,w) = 'H');
  UNLOCK
END REPEAT
```

processes (if not before).

In each case, we have shown that another supervisor step is eventually simulated, establishing (S4). The remaining properties, (S1) - (S3), are straightforward but tedious to verify.

Lemma 4.4. For each k, ℓ, there exist constants c_1, c_2, c_3 such that the following holds. For each N, there is a k-simulatable system S with constants c_1, c_2, c_3 of N workers with a supervisor satisfying (C1), (C2), and (C4') which uses only O(N) values of shared memory.

Proof. We use an extension of the ideas of [CH1] and [BFJLP] as well as those of the colored ticket algorithm which proves Theorem 4.2. The system maintains a queue of "buckets" which are identified by color. The first bucket is called buf, the second, main, the third, gate, and the remaining are a sequence queue of length at most ℓ. buf and main can each hold up to N processes; the remaining buckets never contain more than k-1 processes.

A process first entering its trying region joins bucket buf. From then on, there are two ways it can advance toward its critical section. First, from time to time, the supervisor rearranges the buckets, so at some later time the process may find its bucket is well into the queue. The other method requires an explicit action on the part of the process. When the supervisor sets MOVE to true, any process in main can move to gate. When the supervisor sets enb(c) non-zero, any process in bucket c can enter its critical section.

The following data is kept in shared memory:

buf - color of current buffer bucket;

bufcount - number of processes currently in buf;

main - color of current main bucket;

gate - color of current gate bucket;

enb(c) - the number of processes holding color c that are group-enabled (undefined if $c \notin \text{queue} \cup \{\text{buf}, \text{main}, \text{gate}\}$);

cscount - number of processes in their critical regions;

move - true to request a process to move from main to gate.

elig(c) - enables a worker to determine its eligibility class. Value is 'H' when $c \in \{\text{buf}, \text{main}, \text{gate}\}$, 'Q' from the time c enters the queue until enb(c) is first set non-zero, and 'D' thereafter.

We need a total of ℓ+3 colors. Thus, all of the shared variables are bounded by a function of k and ℓ except for bufcount, which can take on N+1 values. Hence, the storage requirement is O(N) for k and ℓ fixed.

The supervisor keeps the following additional data in local memory:

queue - a sequence $q_1,\ldots,q_m$ of colors, where $m \leq \ell$;

count(c) - the number of processes holding color c less enb(c), where $c \in$ queue;

maincount - the number of processes currently in main;

gatecount - the number of processes currently in gate.

All of these variables are bounded by a function of k and ℓ except for maincount, which can take on N+1 values. However, maincount is bounded by the total number of processes in their trying regions, so property (R7) will hold for a suitable choice of c_2.

The supervisor now uses the following simple strategy. Whenever it finds an empty bucket on its queue, it deletes it from the queue. Whenever it finds main is empty, it moves the entire bucket buf to main and chooses a new color for buf. Whenever it finds gate is empty, it uses one of two strategies to try to fill gate. If main has at most k-1 processes, then the entire bucket is moved to gate. Otherwise, MOVE is set true to request one process in main to move to gate. The supervisor then waits for this to occur, which it must since not all of the processes in main can fail. Whenever the supervisor finds that the queue has room for another bucket, it puts gate on the queue. Finally, whenever it finds a place in the critical section not already filled or allocated (through group-enabling), it group-enables another process from the first bucket on the queue not already fully enabled.

In the code on the following page, "colors" is a constant denoting [ℓ+3]. Whenever a new color is chosen, there are at most ℓ+2 colors in the variables buf, main, gate and queue, so that procedure newcolor described below will always succeed in obtaining an unused color. The function delete(c, queue) returns queue with all occurrences of c deleted. The syntax "WHILE (LOCK; test) DO body;" is used as an abbreviation for "LOCK; WHILE (test) DO [body; LOCK];".

Initially, all the counts are 0, buf, main, and gate are assigned distinct colors, elig(c) = 'H' for all $c \in$ colors, move = false, and the queue is empty.

A process P_i in its trying region holding color c is in eligibility class elig(c) and is in Class 'D' otherwise; hence (R2) and (R3) hold. By inspection of the code and the fact that elig(c) is set to 'D' the first time enb(c) is set non-zero, we establish (R1) and (R4). Also, (R5) is immediate.

Let $c_1 = \ell \cdot (k - 1) + 2k - 2$. At most

Code for Supervisor (Lemma 4.4)

```
REPEAT FOREVER

  /*Eliminate empty buckets from queue.*/

  LOCK;
  FOR ALL c ε colors DO
    IF c ε queue & count(c) + enb(c) = 0
    THEN queue := delete(c, queue);
  UNLOCK;

  /*Fill empty main.*/

  IF maincount = 0
  THEN
    BEGIN
    WHILE (LOCK; bufcount ≠ maincount) DO
      [maincount := maincount + 1; UNLOCK];
    main := buf;
    buf := newcolor();
    bufcount := 0;
    UNLOCK
    END;

  /*Fill empty gate.*/

  IF gatecount = 0
  THEN
    BEGIN
    LOCK;
    IF maincount ≤ k - 1
    THEN
      BEGIN
      gate := main;
      gatecount := maincount;
      main := newcolor();
      maincount := 0
      END
    ELSE
      BEGIN
      move := true;
      WHILE move DO [UNLOCK; LOCK];
      maincount := maincount - 1;
      gatecount := 1
      END
    UNLOCK;
    END

  /*Fill vacant slot on queue.*/

  IF length(queue) < ℓ
  THEN
    BEGIN
    LOCK;
    queue := queue∘gate;
    count(gate) := gatecount;
    enb(gate) := 0;
    elig(gate) := 'Q';
    gate := newcolor();
    gatecount := 0;
    UNLOCK
    END;

  /*Enable new process.*/

  LOCK;
  totcount :=    Σ    count(c);
              cεqueue
  totenb :=    Σ    enb(c);
            cεqueue
  IF cscount + totenb < ℓ & totcount > 0
  THEN
    BEGIN
    i := 1;
    WHILE count(queue_i) = 0 DO i := i+1;
    c := queue_i;
    elig(c) := 'D';
    enb(c) := enb(c) + 1;
    count(c) := count(c) - 1
    END;
  UNLOCK
END REPEAT
```

```
PROCEDURE newcolor()
FOR c ε colors DO
  IF c ∉ queue ∪ {buf, main, gate}
  THEN EXIT LOOP;
enb(c) := 0;
elig(c) := 'H';
RETURN c
END
```

Code for Process P_i (Lemma 4.4)

```
Trying Protocol
LOCK;
c := buf;
bufcount := bufcount + 1;
UNLOCK;
WHILE (LOCK; enb(c) = 0) DO
  BEGIN
  IF move & c = main
  THEN [move := false; c := gate];
  UNLOCK
  END
enb(c) := enb(c) - 1;
cscount := cscount + 1;
UNLOCK

Exit Protocol
LOCK;
cscount := cscount - 1;
UNLOCK
```

$\ell\cdot(k - 1)$ processes can be in all of the buckets in the queue, establishing (R6). As previously mentioned, (R7) holds by choosing c_2 sufficiently large. Finally, (R8) holds since the only supervisor variable whose size is not bounded by a function of k and ℓ, maincount, is never changed on a single step except by incrementing or decrementing it or setting it to 0.

Theorem 4.3. Let k, ℓ, $N \in \mathbb{N}$. There exists a system S' of N processes with initial i.d. q_0' which satisfies ℓ-exclusion (C1), is (k,ℓ)-deadlock-free (C2) and satisfies bounded waiting for enabling (C4'), and S' uses O(N) values of shared memory. (The constant coefficient is exponential in each of k and ℓ.)

Proof. Apply Lemma 4.4 to obtain a k-simulatable system of S of N workers with a supervisor having the desired properties. Lemma 4.3 eliminates the supervisor from S to yield S'. By Lemma 4.2, (C1), (C2), and (C4') are preserved. The number of shared memory values of S' is only a constant times larger than the number for S, which is O(N).
□

Lemma 4.5. For each k, ℓ, there exist constants c_1, c_2, c_3 such that the following holds. For each N, there is a k-simulatable system S with constants c_1, c_2, c_3 of N workers with a supervisor satsifying (C1), (C2), and (C5') which uses $O(N (\log N)^c)$ values of shared memory where c is a constant depending only on k and ℓ.

Sketch of Proof. We describe the main ideas used in constructing S and leave the complete construction to the full paper.

We use many of the ideas in the algorithms which prove Theorem 4.2 and Lemma 4.4. The system maintains a queue of processes. Processes within the first ℓ positions of the queue are enabled or already in their critical regions. A process marks its queue entry as empty when it leaves its critical region, and the supervisor subsequently deletes it entirely, thereby enabling a waiting process. The queue is kept in two parts: the first $2\ell + 2k - 1$ entries are the exposed part and are kept in shared memory; the rest of the queue is kept in the private memory of the supervisor and is called the hidden part.

Processes on the queues are identified by temporary and changeable names. Every process has a long name which it knows, and it may also have a short name. Long names are used on the hidden part of the queue and short names on the exposed part. The supervisor knows the long name of every process on the hidden part. For the long name, we use the notion of ticket introduced in the proof of Theorem 4.2. A ticket has two parts: a color and a number. There are $2\ell + 3k + 2$ distinct colors, and the number part is in [N]. An entering process is issued the next available ticket of the current issue color. (As described below, the supervisor from time to time changes the current issue color, so that tickets never run out.)

A short name consists of a color and an integer in [k-1]. Long names of color c are converted to short names by a map M(c). The supervisor constructs M(c) and leaves it in the shared variable. There is a fixed universal function "short" whereby any process can find the short name of its ticket t, short(M, t), given a map M for t's color. Short preserves the color of t.

Short names are not in general unique. However, if T is a monochromatic set of long names and $|T| \leq k-1$, then there is a map M_T which uniquely encodes T in the sense that if $t_1, t_2 \in T$, then $\text{short}(M_T, t_1) = \text{short}(M_T, t_2)$ iff $t_1 = t_2$.

The problem of finding short names has a history in terms of monotone formula lengths for threshold k functions. Kleiman and Pippenger summarize the history in [KP]. Briefly, Khasin [Kh] gives a non-explicit construction of formulas of length O(N log N) (which, translated into our formulation, shows that only O(log N) different maps are needed to encode every monochromatic subset T of tickets with $|T| \leq k - 1$.) Korobkov [Ko] provides an explicit construction of formulas of length $O((N \log N)\binom{k}{2}^{\log^* N})$. We describe a particularly simple explicit encoding (probably the same as Korobkov's construction) that requires $O((\log N)^{k-2})$ values per map, and thus $O((\log N)^{(k-2)(2\ell+2k-1)})$ values for all the maps.

Let $T = \{t_1, \ldots, t_m\}$ be a monochromatic set of tickets, $m \leq k-1$, and let $t_i = (c, x_i)$, $1 \leq i \leq m$. Express $x_1, \ldots, x_m$ in binary notation. There exist m-1 bit positions $i_1, \ldots, i_{m-1}$ which distinguish all of the x's, that is, for each r, s, if x_r agrees with x_s in position i_j for each $1 \leq j \leq m-1$, then r = s. This is easily established by induction on m. Now, let $b_{r,j}$ be bit i_j of x_r, and let $a_r = (b_{r,1}, \ldots, b_{r,m-1})$. Define the map $M_T = (i_1, \ldots, i_{m-1}, a_i, \ldots, a_m)$. Now, if t = (c,x), then $\text{short}(M_T, t) = (c, r)$ where r is the least r' for which bit i_j of x equals $b_{r',j}$ for all j, $1 \leq j \leq m-1$, and r = 1 if no such r' exists. Clearly, short has the desired property that $\text{short}(M_T, t_i) = (c, i)$, $1 \leq i \leq m$.

We now sketch the action of the supervisor. Its only functions are to delete processes from the queue which have left their critical regions and to maintain the property that the first $2\ell + 2k - 1$ entries are exposed. Thus, a deletion must always be accompanied by the move of a process from the hidden part to the exposed part whenever the latter part is non-empty.

Let P_i be a process to be moved from the hidden queue to the exposed queue, and color_i the color of its ticket. In order to move P_i, the supervisor must define $M(\text{color}_i)$ if it has not already been defined. Let B(c) be the set of processes currently holding tickets of color c. Assume inductively that if B(c) contains any ticket already on the exposed part, for any color c, then $|B(c)| \leq k-1$ and $M(c)$ has been defined. Thus, if $B(\overline{color}_i)$ contains any ticket already on the exposed part, then nothing further needs to be done. If $|B(color_i)| \leq k-1$ but $M(color_i)$ has not yet been defined, it can be defined at this time to encode $B(color_i)$, since every process in $B(color_i)$ is on the hidden queue and therefore the supervisor knows its long name. Otherwise, $|B(color_i)| > k-1$, and every process in $B(color_i)$ is on the hidden part of the queue. In this case, the supervisor attempts to converse with all processes in $Z = \bigcup\{B(c') : M(c') \text{ is not yet defined}\}$.

A conversation follows a particular protocol. The supervisor selects two unused colors, $color_1$and $color_2$. Then it requests to talk to any process in Z. The process responds by sending its ticket to the supervisor, thereby identifying itself. The supervisor examines the ticket and depending on what it sees issues a new ticket to the process. If the process is P_i, the new ticket is $(color_1,1)$. Otherwise, the new ticket is the next available ticket of $color_2$. When the process receives the new ticket, it discards its old one and acknowledges the receipt to the supervisor, which then replaces the old ticket on the hidden queue with the new one. This completes the conversation. The actual transmission of a ticket takes place by sending a serial bit-string.

Because of the possibility of failure, the supervisor carries on k conversations concurrently with k different members of Z. Since at most k-1 processes fail, at least one conversation runs to completion. The supervisor then begins a new conversation with yet another process in Z. This continues until all but k-1 members of Z have completed conversations with the supervisor.

At that time, P_i is either the only process in $B(color_1)$ or P_i is one of the remaining k-1 processes in Z, P_i is still in $B(color_i)$, and $|B(color_i)| \leq k-1$ (since $B(color_i) \subseteq Z$). In either case, the map for P_i's color can be constructed. Let c' be its current color. In case c' is also the current color of newly issued tickets, a new unused $color_3$ is chosen for the latter purpose so that additional processes cannot enter B(c') and invalidate M(c').

M(c') is now defined, so the supervisor moves P_i by placing short(M(c'), t) at the end of the exposed part of the queue and deleting t from the head of the hidden part.

We count the number of colors in use at this point. Each process on the exposed part of the queue might have a different color, so that accounts for at most $2\ell + 2k - 1$ colors. All of the processes on the hidden part of the queue either have colors the same as processes on the exposed part, or they are in Z or have colors $color_2$ or $color_3$. Thus, the total numbers of colors in use is $2\ell + 3k$. Two new colors are needed during the conversations with members of Z, so $2\ell + 3k + 2$ colors are sufficient to make the algorithm work.

We now say a few words on how this algorithm can be made k-simulatable. Let the processes in the first ℓ positions of the exposed part be drones, let the next process be a quitter, and let the remaining processes on the queue (including all of the hidden part) be helpers. Any process not on the queue is also a drone. Choose $c_1 = \ell + 2k - 1$. (R1) is satisfied since only the supervisor can insert or delete entries from either part of the queue. (R2) - (R4) and (R6) are immediate from our definition of the eligibility classes and c_1. (R7) is easily satisfied for appropriate choice of c_2 since the hidden part is empty whenever $|T(q)| \leq c_1$. This is because at most ℓ processes on the exposed part are not in their trying regions. (R8) can be made to hold by using the same trick for copying the current issue ticket number into supervisor memory as was used in the algorithm of Lemma 4.4 to copy bufcount to maincount.

Theorem 4.4. Let $k, \ell, N \in \mathbb{N}$. There exists a system S' of N processes with initial i.d. q_0' which satisfies ℓ-exclusion (C1), is (k,ℓ)-deadlock-free (C2) and FIFO enabling (C5'), and S' uses $O(N\cdot(\log N)^c)$ values of shared memory, where c is a constant. (The constant coefficient in the big "oh" is exponential in each of k and ℓ, while c grows only linearly in k and ℓ.)

Proof. Apply Lemma 4.5 to obtain a k-simulatable system S of N workers with a supervisor satisfying the desired properties. Lemma 4.3 eliminates the supervisor from S to yield S'. By Lemma 4.2, (C1), (C2), and (C5') are preserved. The number of shared memory values of S' is only a constant times larger than the number for S, which is $O(N\cdot(\log N)^c)$.

□

5. Lower Bound Results

If q is an i.d., then V(q) denotes the value of the shared variable and $X_i(q)$ denotes the state of process i in q. Let $I \subseteq N$. We say i.d. q looks like i.d. q' to processes P_i, $i \in I$, provided $V(q) = V(q')$ and provided $X_i(q) = X_i(q')$ for each $i \in I$.

The proofs in this section proceed by contradiction; assuming too few values of V, certain i.d.'s look like other i.d.'s to certain sets of processes. This leads those processes to exhibit behavior which violates one of the needed conditions.

Corresponding to Theorem 4.1(b), we have:

Theorem 5.1. Let $\ell, N \in \mathbb{N}$, $1 \le \ell \le N$. Let S be a system of N processes with null exit regions, and q an i.d. such that S satisfies ℓ-exclusion (C1), is $(1,\ell)$-deadlock-free (C2) and satisfies bounded waiting (C4) from q. Then $|V| \ge \ell(N-\ell)$.

Proof. The theorem is trivial for $\ell = N$, so assume $\ell < N$. Let q' be reachable from q with all processes in their remainder regions (by (C2)). Fix i, j, $1 \le i \le \ell$, $0 \le j \le N - \ell - 1$. Construct a schedule as follows. From q', each of $P_1,\ldots,P_\ell$ in turn goes to its critical region (by (C2)) and stops. Next, each of $P_{\ell+1},\ldots,P_{\ell+j}$ takes one step, moving to its trying region (by (C1), since the critical region is full). Then each of $P_1,\ldots,P_i$ takes one step, going to its remainder region (since there are no exit regions). The resulting i.d. is denoted q(i,j); it has $P_1,\ldots,P_i$ and $P_{\ell+j+1},\ldots,P_N$ in their remainder regions, $P_{i+1},\ldots,P_\ell$ in their critical regions and $P_{\ell+1},\ldots,$ $P_{\ell+j}$ in their trying regions. In particular, the critical region is not full at q(i,j), and P_N has not appeared in the described schedule from q' to q(i,j). We show that the shared variable has a distinct value for each q(i,j).

Assume the contrary and consider two cases.

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

Construct schedule h as follows. Starting from q(i,j), $P_{\ell+1},\ldots,P_{\ell+j}$ go through critical regions to their remainder regions (by (C2)). Then P_N cycles b+1 times (from remainder to critical region), where b is such that S satisfies b-bounded waiting from q (by (C2)). But q(i',j') looks like q(i,j) to $P_{\ell+1},\ldots,P_{\ell+j}$ and P_N, so h causes the same behavior from q(i',j'). Thus, P_{j+1} (b+1)-waits for P_N, so that b-bounded waiting (C4) is violated.

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

Construct schedule h as follows. Starting from q(i',j), i+1 processes from among those in $\{P_1,\ldots,P_i\} \cup \{P_{\ell+1},\ldots,P_N\}$ move into their critical regions and stop. (Since $\ell < N$, there are sufficiently many processes.) This is possible, by (C2), because only $\ell-i'$ ($< \ell-i$) processes are critical and no process other than those in the given sets is in its protocols. But q(i,j) looks like q(i',j) to $P_1,\ldots,P_i$ and $P_{\ell+1},\ldots,P_N$, so h causes the same behavior from q(i,j). But $P_{i+1},\ldots,P_\ell$ are critical at q(i,j); thus h applied from q(i,j) causes a violation of ℓ-exclusion.

□

In order to prove a lower bound corresponding to Theorem 4.2, we first require a construction and a lemma. Let $\ell, N \in \mathbb{N}$, $N \ge \ell+2$. Let S be a system of N processes, q an i.d. such that S is ℓ-queuelike (C6) from q. Let $q' \in \bigcap_{i\in[N]} R_i$ be reachable from q. Fix i,j, $\ell \le j < i \le N-1$. Construct a schedule as follows. Starting at q' each of $P_1,\ldots,P_\ell$ goes in turn to its critical region. Then each of $P_{\ell+1},\ldots,P_N$ takes one step, going to its trying region. (Let P_N's state after its entry be denoted by x, for later reference.) Then one of the critical processes returns to its remainder region (leaving one empty critical slot). (Call the resulting i.d. q", for later reference.) Next, each of $P_{\ell+1},\ldots,P_i$ in turn takes two steps (thereby returning to its remainder region). Finally, each of $P_{\ell+1},\ldots,P_j$ takes one step (thereby entering its trying region once again). The resulting i.d. is denoted q(i,j). Note that P_{i+1} is critical-enabled at q(i,j).

Lemma 5.1. Let $\ell, N \in \mathbb{N}$, $N \ge \ell+2$. Let S be a system of N processes, q an i.d. with S ℓ-queuelike (C6) from q. For each i,j, $\ell < j < i \le N-1$, construct q(i,j) as above. Then the variable has a distinct value for 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 critical-enabled at q(i',j'), hence is critical-able at q(i,j). But also P_{i+1} is critical-enabled at q(i,j). Thus, the schedule (i'+1)(i+1) when applied at q(i,j) causes both $P_{i'+1}$ and P_{i+1} to enter their critical regions, violating ℓ-exclusion.

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 $P_{i+1},\ldots,P_N$, $P_{\ell+1},\ldots,P_j$ takes two steps (thereby returning to its remainder region). Then $P_{j'+1}$ is critical-enabled 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 critical-able at $q_1 = r(q(i,j'),h)$. But also P_{j+1} is critical-enabled at q_1. As in Case 1, ℓ-exclusion can be violated. □

Now corresponding to Theorem 4.2, we have:

Theorem 5.2. Let ℓ, N ε ℕ, N > ℓ+2. Let S be a system of N processes, q an i.d. with S ℓ-queuelike (C6) from q. Then $|V| \geq \ell\binom{N-\ell-1}{2} = \frac{1}{2}\ell(N-\ell-1)(N-\ell-2)$.

Proof. By induction on ℓ.

ℓ = 1: By Lemma 5.1, there are at least $\binom{N-1}{2} \geq 1 \times \binom{N-2}{2}$ distinct values.

ℓ > 1: By Lemma 5.1, there are $\binom{N-\ell-1}{2}$ distinct values of the variable for the i.d.'s q(i,j), where $\ell \leq j < i \leq N-2$. Moreover, P_N is not critical-able at any of these i.d.'s. That is, $\delta_N(V(q(i,j)),x) \notin C_N$ for all i,j, $\ell \leq j < i \leq N-2$.

Now reconsider the construction preceding Lemma 5.1.Starting at q", each of $P_{\ell+1},\ldots,P_{N-1}$ takes two steps, thereby returning to its remainder region. Call the resulting i.d. q'''. From q''', consider $P_1,\ldots,P_{N-1}$ as comprising a system T of N-1 processes. Since S is ℓ-queuelike from q, it can be shown that T is (ℓ-1)-queuelike from q'''. Thus, by induction, the number of values which can be taken on by T's variable is at least $(\ell-1)\binom{(N-1)-(\ell-1)-1}{2} = (\ell-1)\binom{N-\ell-1}{2}$. But P_N is critical-enabled at q''', so that $\delta_N(v,x) \in C_N$ for all v which can be taken on by the variable in i.d.'s reachable from q''' using only $P_1,\ldots,P_{N-1}$.

Thus, the total number of values of V is at least $\binom{N-\ell-1}{2} + (\ell-1)\binom{N-\ell-1}{2} = \ell\binom{N-\ell-1}{2}$, as needed.

□

Theorem 5.3. Let k, ℓ, N ε ℕ, $1 \leq \ell < k \leq N$. Let S be a system of N processes, q an i.d. such that S satisfies ℓ-exclusion (C1), is (k,ℓ)-deadlock-free (C2) and satisfies bounded waiting for enabling (C4') from q. Then $|V| \geq \ell(N-\ell)$.

Proof. The theorem is trivial for ℓ = N, so assume ℓ < N.

Let $q' \in \bigcap_{i\in[N]} R_i$ be reachable from q. Define a "primary" schedule h and a sequence of i.d.'s q(i,1), $1 \leq i \leq \ell$, which appear in order when h is applied from q'. Each q(i,1) has the i-1 processes $P_1,\ldots,P_{i-1}$ critical-enabled in their trying regions, P_i in its trying region, $P_{i+1},\ldots,P_{\ell+1}$ in their critical regions, and all other processes in their remainder regions. Namely, starting at q', each of $P_2,\ldots,P_{\ell+1}$ in turn enters its critical region and stops. Then P_1 takes one step, entering its trying region. The resulting i.d. is q(1,1). Assume inductively that q(i,1) has been defined, i < ℓ. Starting at q(i,1), both P_{i+1} and P_{i+2} leave their critical regions and go to their remainder regions without any other processes taking steps (by (C2)). Then P_{i+2} cycles b+1 times (from remainder to critical), where b is such that S satisfies b-bounded waiting for enabling from q. This forces P_i to become critical-enabled (or (C4') would be violated). Then P_{i+1} takes one step, entering its trying region (since a total of ℓ processes are already either critical or critical-enabled). The resulting i.d. is q(i+1,1).

Now fix i, j, $1 \leq i \leq \ell$, $1 \leq j \leq N-\ell$. Construct a "secondary" finite schedule as follows. Starting at q(i,1), each of $P_{\ell+2},\ldots,P_{\ell+j}$ in turn takes one step, entering its trying region (by Lemma 5.2).Call the resulting i.d. q(i,j). Each q(i,j) has $P_1,\ldots,P_{i-1}$ critical-enabled in their trying regions, $P_{i+1},\ldots,P_{\ell+1}$ in their critical regions, P_i and $P_{\ell+2},\ldots,P_{\ell+j}$ in their trying regions, and all other processes in their remainder regions. We show that the variable has a distinct value for each q(i,j).

Assume the contrary, and consider two cases.

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

$P_1,\ldots,P_i$ are all critical-enabled in their trying regions at q(i',j'), so that the schedule h_1 = 12...i applied to q(i',j') moves $P_1,\ldots,P_i$ to their critical regions. None of $P_1,\ldots,P_i$ takes a step either in the defined secondary schedule from q(i,1) to q(i,j), or in h from q(i,1) to q(i',1), or in the secondary schedule from q(i',1) to q(i',j'). Thus, q(i,j) looks like q(i',j') to $P_1,\ldots,P_i$ and so h_1 has the same effect when applied from q(i,j). But $P_{i+1},\ldots,P_{\ell+1}$ are critical at q(i,1) and therefore also at q(i,j). Thus, h_1 applied from q(i,j) causes a violation of ℓ-exclusion.

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

Define a schedule h_1 as follows. Starting at $q(i,j)$, all processes not in their remainder regions, $P_1,\ldots,P_{\ell+j}$, go to their remainder regions. Then $P_1,\ldots,P_{\ell-1}$ go to their critical regions and stop. Then P_ℓ cycles from remainder to critical b+1 times (b as above). Since $q(i,j')$ looks like $q(i,j)$ to $P_1,\ldots,P_{\ell+j}$, the behavior of these processes is the same when h_1 is applied from $q(i,j')$. But $P_{\ell+j'}$ is in its trying region at $q(i,j')$ and remains there throughout the application of h_1. Moreover, $P_{\ell+j'}$ is not critical-enabled at $r(q(i,j'),h_1)$ because the ℓ processes $P_1,\ldots,P_\ell$ are critical at this i.d. Thus b-bounded waiting for enabling (C4') is violated.

□

Note that neither of Theorems 5.1 and 5.3 directly implies the other, although their statements are very similar. This is because the deadlock-free requirement in Theorem 5.3 is more stringent than that in Theorem 5.1, whereas the fairness condition in Theorem 5.1 is more stringent than that in Theorem 5.3.

6. Further Research

Many of the bounds in this paper might be tightened substantially. In general, our upper bounds depend exponentially on k and ℓ, while our lower bounds are linear in ℓ.

We have no lower bound results involving finite waiting ((C3) or (C3')).Such bounds for the case $\ell = 1$ occupy a prominent position in [BFJLP]; we hope that some of the techniques of that paper will extend to yield bounds for larger ℓ.

An important direction for further research is to extend our failure-proofing ideas to other synchronization problems and system designs. For example, the notion of passive communication with a possibly-failed process (through the enabling mechanism) and the use of distributed, redundant copies of crucial system status information seem to be rather general mechanisms for dealing with failure and are not specific to the problems of uniform resource allocation studied here.

A perhaps unfortunate coincidence concerns the formal similarity between the correctness conditions for the uniform resource allocation problems and the definitions for immunity to limited process failure, for both are described in terms of how the system behaves when certain processes stop execution. (The former involves halting in the critical region and the latter concerns stopping in the protocols.) This similarity sometimes makes it difficult to disentangle the ideas which deal with the particular resource allocation problem from those which deal with the failure-proofing. The study of other problems such as Dining Philosophers [Di2] and Readers-Writers [CHP] in the context of limited process failure should help clarify these distinctions.

REFERENCES

[BFJLP] Burns, J.E., Fischer, M.J., Jackson, P., Lynch, N.A., and Peterson, G.L. "Shared Data Requirements for Implementation of Mutual Exclusion Using a Test-And-Set Primitive." International Conference on Parallel Processing, Bellaire, Mich., Aug. 1978.

[CHP] Courtois, P.J., Heyman, F., and Parnas, D.L. "Concurrent Control with 'Readers' and 'Writers'." Comm. ACM 14, 10 (Oct. 1971), 667-668.

[CH1] Cremers, A. and Hibbard, T. "Mutual Exclusion of N Processors Using O(N)-valued Message Variable." (extended abstract). USC, 1977.

[CH2] Cremers, A. and Hibbard, T. "Arbitration and Queueing Under Limited Shared Storage Requirements." Forschungsbericht Nr. 83, 1979, University of Dortmund.

[Di1] Dijkstra, E. "Solution of a Problem in Concurrent Programming Control." CACM 9, 9 (1965), p. 569.

[Di2] Dijkstra, E. "Hierarchical Ordering of Sequential Processes." Acta Informatica 1 (1971), 115-138.

[EM] Eisenberg, M. and McGuire, M. "Further Comments on Dijkstra's Concurrent Programming Control Problem." CACM 15, 11 (1972), p. 999.

[Kh] Khasin, L.S. "Otsenka Slozhosti Realizatsii Monotonnykh Simmetrich Eskish Funktsii Formulami v Bazise V, &, ¬ ." Dok. Akad. Nauk SSSR 189 (1969) 752-755, translated as "Complexity Bounds for the Realization of Monotonic Symmetrical Functions by Means of Formulas in the Basis V, &, ¬ ." Soviet Physics Dokl. 14 (1970) 1149-1151.

[Kn] Knuth, D. "Additional Comments on a Problem in Concurrent Control." CACM 9 (1966), p. 321.

[Ko] Korobkov, V.K. "Realizatsiya Simmetricheskikh Funktsii V Klasse PI-Skhem." Dokl. Akad. Nauk SSSR 109 (1956) 260-263.

[KP] Kleiman, M. and Pippenger, N. "An explicit Construction of Short Monotone Formulae for the Monotone Symmetric Functions." Theoretical Computer Science 7 (1978) p. 325-332.

[Lam] Lamport, L. "A New Solution of Dijkstra's Concurrent Programming Problem." CACM 17, 8 (1974), p. 453.

[PF] Peterson, G., and Fischer, M.J. "Economical Solutions for the Critical Section Problem in a Distributed System." Proc. Ninth ACM Symposium on Theory of Computing, (1977), p. 91-97.

[RP] Rivest, R. and Pratt, V. "The Mutual Exclusion Problem for Unreliable Processes: Prelim. Rpt." Proc. 17th Annual Symposium on Foundation of Computer Science, (1976), p. 1-8.