A note on group mutual exclusion

[Extended Abstract]

Vassos Hadzilacos*
Department of Computer Science
University of Toronto
vassos@cs.toronto.edu

ABSTRACT

Group mutual exclusion is a natural problem, formulated by Joung in 1998, that generalises the classical mutual exclusion problem. In group mutual exclusion a process requests a "session" before entering its critical section; processes are allowed to be in the critical section simultaneously provided they have requested the same session. To rule out solutions that cause processes to delay each other even when they all request the same session, group mutual exclusion algorithms must satisfy a property called "concurrent entering". Joung stated this property only informally. Keane and Moir later gave a precise statement of this property and devised a simple group mutual exclusion algorithm that satisfies it.

We argue that Keane and Moir's formulation is not quite as strong as the property Joung described informally. We propose a new precise and simple formulation of concurrent entering that is stronger than Keane and Moir's and properly captures Joung's intention. Keane and Moir's algorithm does not satisfy this stronger property, while Joung's original algorithm does. We present another algorithm that satisfies this stronger property and has some advantages over Joung's.

1. INTRODUCTION

Group mutual exclusion is a natural generalisation of the classical mutual exclusion problem [3, 8] that was recently identified and solved by Joung [5]. In mutual exclusion, we have a collection of processes that alternate repeatedly between two sections of code: a possibly nonterminating *noncritical section* (NCS) and a terminating *critical section* (CS). The problem is to provide a synchronisation algorithm in the form of a *trying* and an *exit* protocol to be executed, respectively, immediately before and after the CS so that:

*Supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

- no two processes are in the CS at the same time;
- a process that leaves the NCS eventually enters the CS; and
- a process that leaves the CS eventually returns to the NCS.

In the *group mutual exclusion* problem, a process wishing to enter the CS requests a "session". Processes that request different sessions cannot be in the CS simultaneously, but processes that have requested the same session can. Sessions represent resources each of which can be used simultaneously by an arbitrary number of processes, but no two of which can be used simultaneously.

An example of group mutual exclusion, described by Joung, is a CD "juke box" shared by multiple processes [5]: Any number of processes can simultaneously access the currently loaded CD, but processes wishing to access a CD other than the currently loaded one must wait. In this case, the sessions are the CDs in the juke box. A different example is a server that can cache locally one chunk (e.g., one page) of a large distributed database (e.g. the web). Clients of this server interested in the currently cached chunk can read it simultaneously; clients interested in a different chunk must wait. In this case, the sessions are the database chunks (e.g., the web pages). In the first example, it is perhaps reasonable to assume that the processes know the entire set of sessions and that this set is fixed. In the second example, however, these assumptions are not reasonable: No client can know the entire set of web pages — in fact, this set changes over time and is of unbounded size. We will return to this issue later.

The informal specification of group mutual exclusion given above can be trivially satisfied by using a traditional mutual exclusion algorithm and forcing processes to enter the CS one-at-a-time, even if all processes are requesting the same session. To avoid this (obviously inefficient) solution, Joung required that a group mutual exclusion algorithm additionally satisfy the following property, which he called concurrent entering:

- If some processes request a session and no process requests a different session, then the processes can concurrently enter the CS.

Unfortunately, the phrase "can concurrently enter", although suggestive, is not precise. Joung contrasts concurrent *entering* with a different and weaker requirement that might be called "concurrent *occupancy*", in which processes requesting the same session can occupy the CS concurrently but may delay each other during entry: "The intent of this stronger condition is to prevent unnecessary synchronization" among processes requesting a session when no one else requests a different session. Again, the phrase "unnecessary synchronization" is suggestive but not entirely clear. Keane and Moir [6] gave a precise but, as we will argue, weaker definition of concurrent entering. Their definition captures the concurrent occupancy requirement, but not the stronger one of concurrent entering.

In this paper we give a simple and precise formulation of concurrent entering which we believe captures the property that Joung intended to specify. Joung's algorithm [5] satisfies this property, but Keane and Moir's algorithm [6] does not. We also present a group mutual exclusion algorithm that satisfies this property. Our algorithm has two advantages over Joung's. First, it satisfies a strong fairness property: processes requesting different sessions gain access to the CS in the order in which they request entry. Second, it does not require that processes know in advance the set of sessions or that the set of sessions be bounded. In contrast, Joung's algorithm requires that processes know the set of possible sessions, and that this set be bounded. On the negative side, our algorithm requires $\Theta(N^2)$ shared variables, while Joung's requires only $\Theta(N)$, where N is the number of processes.

The remainder of this paper is organised as follows. In Section 2 we describe the model of computation, and we give a precise specification of the group mutual exclusion problem. We also discuss our formulation of the concurrent entering property, and how it is related to other formulations. In Section 3 we describe our group mutual exclusion algorithm and discuss its performance with respect to various complexity measures. The proof of correctness is omitted from this extended abstract, but is available in the full version of the paper. We conclude in Section 4 with a discussion of related and future work.

2. MODEL AND SPECIFICATION

2.1 The model

We consider a system consisting of N processes, named $1, 2, \ldots, N$, and a set of shared variables. Each process also has its own private variables. Processes can communicate only by writing into and reading the shared variables. An execution is modeled as a sequence of process steps. In each step, a process can perform exactly one of three actions: perform some local computation (affecting only its private variables), write into a shared variable, or read a shared variable. Processes take steps asynchronously. More precisely, this means that in an execution an unbounded (but finite) number of other processes' steps can lie between two successive steps of a process. Although processes are asynchronous, they are *live*: if a process has not terminated, it will eventually execute its next step. Note that individual write and read operations are atomic, but processes do not have access to powerful synchronisation operations such as atomic test-and-set or compare-and-swap.

```
repeat
    NCS
    Trying protocol
    CS
    Exit protocol
forever
```

Figure 1: A group mutual exclusion process

2.2 Specification

In group mutual exclusion each process consists of a nonterminating loop, inside which the process executes, in sequence, four sections of code: the noncritical section (NCS), the trying protocol, the critical section (CS), and the exit protocol (see Figure 1). Each iteration of the loop is called a *passage*.

We assume that the shared variables used by the trying and exit protocols are not accessed by the NCS and CS. We further assume that a process can be in the CS for a finite but unbounded amount of time, but a process can (eventually) be in the NCS forever. These assumptions are as in the classical mutual exclusion problem. For *group* mutual exclusion, we also assume that before exiting the NCS to start the trying protocol, a process sets a private variable *mysession* to the session it is requesting. While a process is outside its NCS, we say that it *requests session s* if the value of its variable *mysession* is s.

The group mutual exclusion problem is to devise code for the trying and exit protocols so that the following properties hold:

Mutual exclusion If two processes are in the CS at the same time, then they request the same session.

Lockout freedom If a process enters the trying protocol, then it eventually enters the CS.

Bounded exit If a process enters the exit protocol, then it enters the NCS within a bounded number of its own steps.

Concurrent entering If a process i requests a session and no process requests a different session, then i enters the CS within a bounded number of its own steps.

2.3 Comments on the specification

Some comments about these properties are in order. The bounded exit property is sometimes replaced by the following, weaker, property that more closely resembles lockout freedom:

Terminating exit If a process enters the exit protocol, then it eventually enters the NCS.

Since a process can be in the CS for an unbounded (but finite) amount of time, the trying protocol cannot be bounded. In contrast, the exit protocol can be bounded, and thus the stronger property is preferable.

As mentioned in Section 1, Joung's original paper on the group mutual exclusion problem did not provide a precise statement of concurrent entering. Keane and Moir gave a precise definition for a version of this property. To distinguish it from our version, and for reasons that will be explained shortly, we call their property concurrent occupancy:

Concurrent occupancy If a process i requests a session and no process requests a different session, then i eventually enters the CS (even if other processes do not leave the CS).

Note that this property is a trivial consequence of lockout freedom, unless runs with nonterminating CS executions are admissible.[1]

For now let us focus on executions where all requests are for the same session. Joung's informal statement of concurrent entering was that (in such executions) processes should be able not only to concurrently occupy the CS but to concurrently enter it without "unnecessary synchronisation". This means that (in such executions) processes should not delay one another as they are trying to enter the CS. Concurrent occupancy ensures that a process i trying to enter the CS is not delayed by other processes that have *already* entered the CS. It does not, however, prevent i from being delayed (for arbitrarily long) by other processes that are simultaneously *trying* to enter the CS.

Not only are such delays allowed by the specification of the property, but the algorithm given in [6] exhibits this behaviour. Described at a high level, this algorithm works as follows. In the trying protocol, a process i first obtains an exclusive lock. Under certain conditions, i can proceed directly to the CS after releasing the lock. If these conditions do not hold, i places itself in a queue, releases the lock and waits. In the exit protocol, i obtains again the exclusive lock. If there are processes waiting in the queue, i chooses one of the requested sessions and enables all processes that have requested that session to enter the CS. It then releases the lock.

Thus, even if all requests are for the same session, a process i may be delayed in its trying protocol by another process j that is also in its trying (or exit) protocol and has obtained the exclusive lock: If j "goes to sleep" while holding the lock, i will have to wait. Therefore, this algorithm does not satisfy our formulation of concurrent entering, which requires that (in the absence of requests for another session) i should be able to enter the CS in a bounded number of its own steps, regardless of the speed of other processes. In contrast, Joung's algorithm satisfies our formulation of concurrent entering.

3. AN ALGORITHM

Lockout freedom states that all requests for entry to the CS are *eventually* satisfied. It may be desirable to ensure a stronger property: Requests for entry to the CS (under different sessions) are satisfied in the order in which they are made. There are various meaningful ways of formalising what it means for a process to request entry to the CS before another. One of them is to require that the trying protocol of each process start with a wait-free section of code (i.e., a section of code that can be completed in a bounded number of the process' own steps), called the *doorway*. Then a process i requests entry before process j, if i completes its doorway before j enters its doorway.[2]

In this section we present a group mutual exclusion algorithm that also satisfies the *first-come-first-served* (FCFS) property:

FCFS If process i completes the doorway before process j enters the doorway and the two processes request different sessions, then j does not enter the CS before i.

Our algorithm is based on a somewhat simplified version of a mutual exclusion algorithm by Lycklama and Hadzilacos [10]. That algorithm has the property that when a process i is waiting in the trying protocol, it is waiting for another process j, *whose identity i knows*, to change the value of some shared variable. Described in general terms, the idea is to turn this into a *group* mutual exclusion algorithm by changing the original algorithm so that i waits for j only if j has requested a different session than i. To enable i to determine the session that j has requested, j writes the name of the session it requests into a shared variable $Session[j]$. It turns out that this simple idea requires considerable embellishment before it yields a correct group mutual exclusion algorithm. An obvious problem is that, due to asynchrony, the reading of another process' "current" session can reflect outdated information and can mislead a process into not waiting when, in fact, it should. A subtler problem arises in connection with the FCFS mechanism of the Lycklama-Hadzilacos algorithm. We will explain this problem and its solution shortly.

The algorithm for process i is shown in Figure 2. First we introduce some notational conventions and then we explain how the algorithm works. The first letter of each shared variable's name is capitalised; private variable names are in lower case. Variables whose initial value is not explicitly stated in Figure 2 can be initialised arbitrarily. The statement form "**wait until** B", where B is a Boolean expression, is a more informative syntax for "**while not** B **do skip end while**", where **skip** is the null statement. We assume that session names are integers other than 0. A session number of 0 indicates that a process is not presently requesting a session.

The trying protocol of the algorithm consists of lines 2–20. It is composed of two parts that can be explained independently of each other. The first part, consisting of lines 2–9, ensures the FCFS property. The second part, consisting of lines 10–20, ensures mutual exclusion.

[1]In fact, admissible runs with nonterminating CS executions should be restricted to those in which (eventually) all requests are for the same session. This restriction is not explicitly stated by Keane and Moir but without it mutual exclusion and lockout freedom are incompatible.

[2]This way of defining the order of requests is due to Lamport, and the name "doorway" derives from the metaphor underlying his well-known "bakery algorithm" for mutual exclusion, where processes obtain a ticket at the "doorway" of a bakery and get served (enter the CS) in the order indicated by their tickets [7].

```
    shared variables
        Session: array [1..N] of integer, initially all 0
        Turn: array [1..N] of 0..3
        Bypass: array [1..N, 1..N] of Boolean
        Competing: array [1..N] of Boolean, initially all false

    private variables
        mysession: integer
        j: 1..N
        turn-snap: array [1..N] of 0..3
        bypassers: set of 1..N

    repeat
1       NCS

2       Session[i] := mysession
3       for j := 1 to N do Bypass[i, j] := false; turn-snap[j] := Turn[j] end for
4       Turn[i] := (Turn[i] + 1) mod 4
5       bypassers := ∅
6       for j := 1 to N do
7           wait until Session[j] ∈ {0, mysession} or Turn[j] ≠ turn-snap[j] or Bypass[i, j]
8           if Session[j] = mysession then bypassers := bypassers ∪ {j} end if
9       end for

10 L:   Competing[i] := true
11      for j := 1 to i − 1 do
12          if Competing[j] and Session[j] ≠ mysession then
13              Competing[i] := false
14              wait until not Competing[j] or Session[j] = mysession
15              goto L
16          end if
17      end for
18      for j := i + 1 to N do
19          wait until not Competing[j] or Session[j] = mysession
20      end for

21      CS

22      for each j ∈ bypassers do Bypass[j, i] := true end for
23      Competing[i] := false
24      Session[i] := 0
    forever
```

Figure 2: Group mutual exclusion algorithm — pseudocode for process i

In the FCFS part of the trying protocol, each process first executes the doorway, consisting of lines 2–4. The doorway works as follows. Each process i owns (i.e., is the only process that can write) a shared variable $Turn[i]$ whose value is drawn from the set $\{0, 1, 2, 3\}$. $Turn[i]$ is initialised arbitrarily and is incremented (modulo 4) by process i each time i passes through the doorway. In the doorway, process i first writes the session it has requested into the shared variable $Session[i]$ (line 2). It then sets to false the i-th row of the shared matrix *Bypass* (whose purpose will be explained shortly) and records in private variable *turn-snap* a "snapshot" of every process' *Turn* value (line 3). Finally, it increments (modulo 4) its own *Turn* value (line 4).

After the doorway, process i executes a waiting statement (line 10) whose purpose is to enforce the FCFS property. Specifically, i waits for every process j until it finds that j is in the NCS ($Session[j] = 0$) or that j requests the same session as i ($Session[j] = mysession$) or that FCFS does not require i to wait for j ($Turn[j] \neq turn\text{-}snap[j]$ or $Bypass[i, j] = \text{true}$).

We now further explain the last point. FCFS requires i to wait for j if j completed the doorway before i entered. In that event, j writes $Turn[j]$ in line 4 before i records the value of $Turn[j]$ in $turn\text{-}snap[j]$ in line 3. Consequently, $Turn[j] = turn\text{-}snap[j]$ remains true until j increments $Turn[j]$ in line 4 of its *next* passage through the doorway — and therefore until after j has completed the CS of its current passage, at which point i no longer has to wait for j.[3] This

[3] For this mechanism to work it suffices that $Turn[i]$ takes on values in $\mathbb{Z}_m$ and that it is incremented modulo m, for *any* $m \geq 4$. Using $m = 2$ is not enough. To see why, suppose only two processes, i and j leave the NCS, and

FCFS mechanism was used in the Lycklama-Hadzilacos mutual exclusion algorithm. In the algorithm presented here, however, a complication arises that can lead to deadlock: A fast process j can enter the CS repeatedly, each time overpassing in the doorway a slow-moving i, *if j requests the same session as i*. If this happens four times and j then requests a different session, it is possible that i will find $Turn[j] = turn\text{-}snap[j]$ in line 7, *even though it should not wait for j*.

The shared variable $Bypass[i, j]$ provides the mechanism that enables i to detect this circumstance and avoid waiting (possibly forever) for j: When j notices that it is requesting the same session as i it adds i to the set *bypassers* (line 8). After j completes the CS, it sets $Bypass[i, j]$ to true (line 22). Thus, when i executes the waiting statement in line 7 it knows not to wait for j, because it finds $Bypass[i, j] =$ true.

The FCFS section of the trying protocol does not ensure mutual exclusion: It is possible for two processes to execute line 3 before either executes line 4. In this case they will not wait for each other in line 7 and they will move to the next statement together. Mutual exclusion is enforced by the section of code consisting of lines 10–20. This is an elegant mutual exclusion algorithm discovered independently by Burns [2] and Lamport [8]. Each process i owns (i.e., is the sole writer of) a single bit, $Competing[i]$. Before entering the CS, i sets its bit to true (line 10) and checks all lower-numbered processes (lines 11–17). If any of them, say j, is found to have set its bit to true and to be requesting a different session, process i resets its bit to false allowing the smaller-numbered processes to make progress (line 13), waits for j's bit to become false or for j to request the same session as i (line 14), and then restarts the competition to enter the CS (line 15).[4] Having checked the lower-numbered processes, i then checks the higher-numbered ones and waits for each of them to set its bit to false or to request the same session as i (lines 18–20). Now, however, while i is waiting it does not set its bit to false. After that i can enter the CS.

It turns out that this simple algorithm guarantees mutual exclusion. By itself, it does not guarantee lockout freedom; it guarantees a weaker property called deadlock freedom:

Deadlock freedom If a process enters the trying protocol and never enters the CS, then some process enters the CS infinitely often.

However, in conjunction with the FCFS mechanism within which it is embedded, it guarantees lockout freedom.

We now briefly discuss the performance of the algorithm with respect to various complexity measures. Joung defined and evaluated his algorithm with respect to three complexity measures. Roughly speaking,

- *Delay* is the maximum number of CS entries that can occur while a process is waiting to enter the CS.[5]
- *Session switches* is the maximum number of times that sessions can change while a process is waiting to enter the CS.
- *Concurrency* is the maximum number of entries into the CS by processes requesting the same session while some process requesting a different session is waiting to enter the CS.

Under all three measures, our algorithm has complexity $\Theta(N)$. The fundamental reason for this is the FCFS property that it enforces.

Keane and Moir evaluated their algorithm with respect to two additional complexity measures, which had been defined in the context of the classical mutual exclusion problem: contention-free time complexity and number of remote references. *Contention-free time complexity* is the maximum number of steps executed by a process in its trying and exit protocols during one passage, assuming all other processes are in the NCS. The contention-free time complexity of our algorithm is $\Theta(N)$.

The number of remote references is a complexity measure defined for two types of multiprocessors, called *non-uniform access memory access* (NUMA) and *cache coherent* (CC), where accesses to shared variables can be either local or remote. A remote access traverses the processor-to-memory interconnect, which takes much longer than an access to a locally stored variable. In a NUMA multiprocessor, each shared variable is local to exactly one process and remote to the others. Accordingly, in such a multiprocessor a process' access to a shared variable is local or remote.

In a CC multiprocessor, we can think of all shared variables as being stored remotely from all processes. When a process first reads a shared variable the value of the variable is copied from its remote location to a local cache and the locally cached copy is marked "valid". When a process writes a shared variable, the new value is written in the variable's

each reads the other's *Turn* value before either increments it. Then both processes increment their *Turn* values and process i temporarily goes to sleep. Process j enters and leaves the CS, and then quickly reenters the trying protocol and increments $Turn[j]$ again, setting it back to its original value (since $m = 2$). At this point i wakes up and finds the value of $Turn[j]$ unchanged. It will therefore wait for j. Process j will also wait for i because the value of $Turn[i]$ hasn't changed, leading to deadlock. A more complex scenario shows that deadlock is also possible if $m = 3$. (This scenario can be extracted from the proof that deadlock freedom is satisfied when $m \geq 4$.) There is a version of this algorithm in which $Turn[i]$ can be incremented modulo 3, rather than 4, but the version described here is simpler.

[4]In view of the conditional in line 12, it may appear redundant for i to test in line 14 whether j has requested the same session. It is possible, however, that i determined in line 13 that j requested a different session just as j was leaving the CS during a passage in which it had indeed requested a different session, and then quickly initiated a new passage now requesting the same session as i. If i didn't explicitly check whether j requested the same session and, if so, terminate the waiting statement in line 14, the algorithm would violate concurrent entering.

[5]Joung calls this "time complexity". However, this term has a well-established meaning as the maximum number of steps involved in a computation, which is quite different from the quantity defined here. For this reason, we prefer to use a different term.

remote location and all cached copies of that variable are marked "invalid". When a process reads a previously read (and hence cached) variable, the locally cached copy is used if it is valid; otherwise, the locally cached copy is refreshed and marked "valid" again. Thus, in a CC multiprocessor a process' access to a shared variable is remote unless it is a read of a variable that happens to be in a valid local cache.

The *number of remote references* of a group mutual exclusion algorithm on a NUMA or CC multiprocessor is defined as the maximum number of remote accesses by a process in its trying and exit protocols during one passage. The number of remote references of our algorithm is unbounded in a NUMA multiprocessor and $\Theta(N)$ in a CC multiprocessor.[6]

4. RELATED AND FUTURE WORK

We now compare our algorithm to other algorithms for group mutual exclusion in the model considered in this paper (i.e., asynchronous processes that communicate by writing and reading shared variables). We also discuss directions for future research.

Joung's original algorithm for group mutual exclusion [5] satisfies the strong property of concurrent entering which he stated informally and was formalised here. His algorithm assumes that the set of sessions is bounded and known in advance to the processes. This assumption is not always true and therefore limits the applicability of that algorithm.

As mentioned in Section 2, the algorithm of Keane and Moir [6] uses an exclusive lock that must be obtained by processes before entering the CS and after leaving it. Consequently, this algorithm satisfies concurrent occupancy, but not the stronger requirement of concurrent entering. It also satisfies terminating exit, but not the stronger requirement of bounded exit (see Section 2.3). Like the algorithm described in the present paper (and unlike Joung's), the Keane-Moir algorithm does not require the set of sessions to be bounded or that processes know this set in advance.

An advantage of Keane and Moir's algorithm is that the exclusive lock can be implemented by any one of the myriads of known mutual exclusion algorithms, each of which can be especially good in some setting. For example, one can use Lamport's "fast" mutual exclusion algorithm that is especially good in the absence of contention [9]. Alternatively, one can use the mutual exclusion algorithm by Yang and Anderson that scales well for large number of processes and has attractive properties in certain common parallel architectures [11]. We are exploring whether some of these mutual exclusion algorithms can be transformed to *group* mutual exclusion algorithms, much in the way in which we were able to transform the Lycklama-Hadzilacos mutual exclusion algorithm to solve group mutual exclusion.

In their paper, Keane and Moir acknowledge the fact that the locking mechanism used by their algorithm entails some "unnecessary waiting" when all processes request the same session, and they sketch a way of eliminating such waiting. Their proposal, however, is extremely heavyweight as it relies on universal constructions of wait-free types [4]. Because of their generality, these constructions are quite complex and inefficient. In addition, they must make use of powerful synchronisation primitives such as atomic compare-and-swap, or load-linked and store-conditional. In contrast, our algorithm avoids "unnecessary waiting" while relying only on reading and writing of shared variables. An interesting question is whether one can exploit powerful synchronisation primitives directly to derive more efficient group mutual exclusion algorithms than is possible with only atomic writes and reads.

Alagarsamy and Vidyasankar [1] present two algorithms neither of which satisfies concurrent occupancy — much less concurrent entering. In some cases their algorithms allow processes requesting the same session to enter the CS concurrently. In other cases they do not, and the authors do not state what property their algorithms satisfy in this respect.

Table 1 summarises properties of seven group mutual exclusion algorithms that use shared variables accessible only through read and write operations. Rows of the table correspond to algorithms and columns to properties. The first two columns, labeled "CE" and "CO", state whether the algorithm satisfies the concurrent entry and concurrent occupancy properties, respectively. The remaining columns correspond to the complexity measures discussed at the end of Section 3. An entry of ∞ indicates that the complexity measure is unbounded for the corresponding algorithm. The first row of the table summarises properties of Joung's algorithm. The quantity M that appears in this row is the number of sessions. Recall that Joung's algorithm requires this to be bounded and known to the processes. The next three rows correspond to instantiations of Keane and Moir's algorithm with different ways of implementing the lock it uses. Row two uses Lamport's "fast" mutual exclusion algorithm, and rows three and four use the two versions of Yang and Anderson's "scalable" mutual exclusion algorithm. Rows five and six correspond to the two algorithms by Alagarsamy and Vidyasankar. The last row corresponds to the algorithm described in this paper.

Our algorithm is the only group mutual exclusion algorithm we know that satisfies the FCFS property. As we mentioned, one disadvantage of our algorithm is that it uses $\Theta(N^2)$ shared variables, where N is the number of processes. There is a somewhat simpler version of our algorithm that uses only $O(N)$ shared variables which, however, must hold integers that grow without bound as the execution progresses. We are investigating the feasibility of a FCFS group mutual exclusion algorithm that uses only $O(N)$ *bounded* shared variables.

The algorithms of Joung, Keane and Moir, and Alagarsamy and Vidyasankar use so-called "capturing" mechanisms. The

[6] In general it is harder for an algorithm to achieve a bounded number of remote references in a NUMA multiprocessor than in a CC one. To see why, consider a remote Boolean variable whose value changes every time it is written (i.e., it alternates between true and false). In a NUMA multpiprocessor a statement waiting until the variable becomes true can generate an unbounded number of remote references. In contrast, in a CC multiprocessor such a statement generates at most two remote references: one when the variable is cached and, if the cached value is false, a second one when the cached copy is invalidated.

Algorithm	CE	CO	Remote References		Cont.-free Compl.	Delay	Session Switches	Concurrency
			NUMA	CC				
Joung [5]	yes	yes	∞	∞	$\Theta(N)$	$\Theta(NM)$	$\Theta(M)$	$\Theta(N^3)$
K&M [6] w/ Lamport [9]	no	yes	∞	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$
K&M [6] w/ Y&A (1) [11]	no	yes	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(\log N)$	∞	∞	∞
K&M [6] w/ Y&A (2) [11]	no	yes	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$	∞	∞	∞
A&V (1) [1]	no	no	∞	$\Theta(N^2)$	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N)$	$\Theta(N)$
A&V (2) [1]	no	no	∞	∞	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N)$	∞
this paper	yes	yes	∞	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$

Table 1: Properties of group mutual exclusion algorithms

idea is that a process entering the CS can "capture" other processes that request the same session and help them enter the CS concurrently with it, possibly overtaking other processes that have requested a different session. Of course the capturing mechanism must be used carefully to avoid starvation. An advantage of such a mechanism is that in some applications (such as the CD juke box) changing sessions is time consuming, and it is desirable to reduce the number of session switches during the course of an execution. Such capturing mechanisms can be incorporated to our algorithm, although doing so would violate the FCFS property. Whether it is better to ensure fair entry to the CS or to reduce the session switching cost depends on the application.

Acknowledgements

I am grateful to Mark Moir and K. Vidyasankar for their comments on an earlier draft, and to the anonymous referees for their suggestions on the draft submitted to the program committee.

5. REFERENCES

[1] K. Alagarsamy and K. Vidyasankar. Elegant solutions for group mutual exclusion problem. Unpublished manuscript, 1999.

[2] James E. Burns. *Complexity of Communication Among Asynchronous Parallel Processes*. PhD thesis, Georgia Institute of Technology, January 1981.

[3] Edger Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8:569, 1965.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[5] Yuh-Jzer Joung. Asynchronous group mutual exclusion. In Yehuda Afek, editor, *Proceedings of the Seventeenth ACM Symposium on Principles of Distributed Computing*, pages 51–60, June 1998.

[6] Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In Jennifer Welch, editor, *Proceedings of the Eighteenth ACM Symposium on Principles of Distributed Computing*, May 1999.

[7] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[8] Leslie Lamport. The mutual exclusion problem: Parts I & II. *Journal of the ACM*, 33(2):313–348, 1986.

[9] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[10] Edward Lycklama and Vassos Hadzilacos. A first-come-first-served mutual exclusion algorithm with small communication variables. *ACM Transactions on Programming Languages and Systems*, 13(4):558–576, October 1991.

[11] Jae-Heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:51–60, 1995.