Yet Another Simple Solution for the Concurrent Programming Control Problem

Alex A. Aravind, *Member*, *IEEE*

Abstract—As multicore processors are becoming increasingly common everywhere, the future computing systems and devices are becoming inevitably concurrent. Also, on the applications side, automation is steadily infiltrating into everyday life, and hence, most software systems are becoming increasingly complex and concurrent. As a result, recent developments and projections indicate that we are entering into the era of concurrent programming. Synchronizing asynchronous concurrent processes in accessing a shared resource is an important issue. Among the synchronization issues, mutual exclusion is fundamental. Solutions to most higher level synchronization problems rely on the assurance of mutual exclusion. Several algorithms with varying characteristics are proposed in the literature to solve the mutual exclusion problem. This paper presents two new algorithms to solve the mutual exclusion problem. The algorithms are simple and have many nice properties.

Index Terms—Mutual exclusion, process/thread synchronization, concurrent programming, nonatomicity, LRU fairness, fault tolerant, bounded timestamps, shared memory.

1 Introduction

Concurrent programming control problem (also called mutual exclusion problem) is a fundamental problem in concurrent computing systems. Dijkstra proposed the first solution to solve the problem for n-process case in 1965 [8]. In a sense, this work of Dijkstra planted a seed for distributed computing. The solution basically resolves conflicting accesses to a shared resource by concurrent processes. The conflict is resolved by enforcing mutually exclusive access to the shared resource. Mutual exclusion algorithms are generally complex and notorious for subtle errors.

Although Dijkstra envisioned the importance of concurrent programming more than four decades ago, only in recent times we are starting to witness its full impact on software development [4], [11], [25], [28]. This trend is primarily due to the ubiquitous use and the availability of single-chip multiprocessors or multicore processors.

Following Dijkstra's algorithm, several solutions have been published in the literature to solve the mutual exclusion problem [1], [2], [13], [4], [11], [25], [30]. This paper proposes two new algorithms to solve the mutual exclusion problem. The algorithms are simple and have many nice properties.

Among the solutions to the mutual exclusion problem, the bakery algorithm by Lamport in [19] is very popular for its simplicity and many other remarkable properties. Our algorithms, in many ways, are similar to the bakery algorithm. One of the limitations of the bakery algorithm is that it uses unbounded size timestamps and that restricts its use in practical systems. Several modifications are proposed in the past to bound the timestamp values [16], [17], [29], [30], [31], [32], [33]. These modifications are quite complex and also compromise nonatomicity property—the most attractive property of the algorithm. The first algorithm proposed in this paper also uses unbounded size timestamps. However, with a trivial modification, this algorithm is then transformed to use bounded size registers. More importantly, the resulting algorithm retains all the properties of the original algorithm, including nonatomicity property.

Among the various criteria used to resolve the contention for shared resources, fairness property is very crucial one. It decides which process among the competing processes is allowed to succeed next. First-In-First-Out ($FIFO$) and Least Recently Used (LRU) are two important fairness criteria widely used in many applications. While FIFO assures fairness based on the order of requests, LRU favors the infrequent users of the resource compared to the frequent users when contention occurs. LRU-based approach is used for resource or service management in many situations in computing systems, including cache replacement in processor execution context and page replacement in main memory management systems.

Most of the mutual exclusion algorithms available in the literature [1], [2], [25], [30] assure some sort of $FIFO$ fairness. The algorithms proposed in this paper assure LRU fairness. In addition, our algorithms are simple, symmetric, and applicable for weaker memory systems where read/write need not be atomic. Multiport memories are of this weaker type, and in recent times most embedded systems are coming up with multiport memories. The second algorithm works with bounded memory.

Concurrent programming, without doubt, is harder than sequential programming. Despite its complexity, recent trend clearly indicates that concurrent programming is becoming mainstream, and therefore, research on concurrent programming issues is back on the spotlight. Several concurrent programming approaches have been proposed in the recent past.

• *The author is with the Department of Computer Science, University of Northern British Columbia, Prince George, British Columbia, Canada. E-mail: csalex@unbc.ca.*

Manuscript received 18 Aug. 2009; revised 6 Dec. 2009; accepted 11 Dec. 2009; published online 28 Sept. 2010.
Recommended for acceptance by R. Bianchini.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-08-0374.
Digital Object Identifier no. 10.1109/TPDS.2010.173.

Among the concurrent programming paradigms proposed recently, the approach of transactional memory has been considered as the promising solution to concurrent programming problems. Yet, this approach is also not without limitations [3], [23], [6]. Implementing transactional memories seems to be expensive and introduces complexities. This observation suggests that whether transactional memories can be adopted as a programming paradigm for practical applications or not is not clear. We believe that no single concurrent programming approach is expected to emerge as suitable for all applications and systems. Different approaches may be suitable and convenient for different applications and systems. For example, a mutual exclusion algorithm proposed in this paper may be suitable for applications which benefit from LRU fairness. Also, this kind of algorithms, due to their overhead on remote memory accesses, seem to be attractive for synchronizing threads in accessing shared data within single address space.

2 System Model and Problem Statement

We consider a shared memory system of n processes with ids $1, 2, \ldots, n$. A shared memory is an abstraction of a collection of shared variables that support two memory operations *read* and *write*. Each shared variable has a unique *name* and a *type*. The type determines the domain of values for that variable. Write operation puts the supplied value into the specified shared variable. A read on a shared variable returns a value from its domain.

In this system, several processes can simultaneously access the same memory location, may be through independent ports. We assume R as the shared resource that requires *mutually exclusive access* among the processes. The **mutual exclusion problem** is to design an algorithm that assures the following properties:

P1. *At any time, at most one process is allowed to access R (safety property)* and

P2. *When one or more processes interested in accessing R, one of them eventually succeeds in accessing R (liveness property).*

In addition to these two properties the following is a desirable property:

P3. *Any process interested in accessing R will be able to do so in finite time (freedom from starvation property).*

For practical applications, higher level of fairness such as a bound on the number of overtakes in accessing R is often most desirable.

The *code segment* of a competing process can be divided into two parts: the part which accesses the shared resource R (*Critical Section*) and the remaining part (*Noncritical Section*). Mutually exclusive access to the shared resource R is achieved by exclusive execution of the corresponding critical section. So, in this context, the access to the critical section means the access to the shared resource R.

A solution to the mutual exclusion problem has essentially two components: *Entry Section* and *Exit Section*. Essentially, the entry section facilitates communication among the competing processes to determine which goes next to access R, and the exit section is used to inform the other competing processes that the current access to R is completed, and therefore, it is free for the next process to access. The entry section and the corresponding exit section must be inserted (or called), respectively, before and after critical section, as shown in Fig. 1, to ensure mutually exclusive access to R.

```
while(true) {
    Non-critical Section
    Entry Section
    Critical Section Accessing R
    Exit Section
}
```

Fig. 1. Code structure with mutual exclusion.

The following are the assumptions made on the system:

A1. *Processes communicate only through read and write operations on shared variables.*

A2. *The execution speed of any process is finite but unpredictable.*

A3. *No process will access R continuously forever.*

In a simplest shared memory model, no two memory operations can overlap in time and read always returns the value of the latest write. Modern systems are much more complex than this. In multiport memories, operations on the same location can overlap in time. We briefly look at various shared memory models and state under what type of memory model our algorithms work.

Based on the number of processes that can write on a shared variable, it is classified either as 1-writer shared variable or multiwriter shared variable. That is, if only one process is allowed to write and all processes are allowed to read a shared variable, then it is 1-writer shared variable. If more than one process is allowed to read and write on a shared variable then it is multiwriter shared variable. Based on the accuracy of the value a read can return, Lamport in his seminal paper [21] defines the following three types of (1-writer) shared variables:

Definition 2.1. *A shared variable X is said to be:*

1. ***Safe*** *if a read on X returns the most recently written value on X, when it does not overlap with any write on X; otherwise, a read on X may return any value from the domain of X.*
2. ***Regular*** *if it is safe and a read on X that overlaps with one or more writes on X returns either the value of the most recent write or the values of one of the overlapping writes.*
3. ***Atomic*** *if it is regular and the reads and writes on X behave as if they occur in some definite order. That is, reads and writes on X do not interfere with each other concurrently.*

Majority of the existing mutual exclusion algorithms assume shared memory with atomic registers. A shared memory with safe registers is the weakest model and designing mutual exclusion algorithm for safe variables is usually difficult. The bakery algorithm is the first mutual exclusion algorithm proposed to work in the shared memory model with safe variables, but it requires unbounded size shared space. Our algorithms can work with safe variables.

A4. *The shared variables are assumed to be* **safe.**

That is, no two $writes$ on the same memory location overlap, but any other combinations may overlap. When a read overlaps with writes, it can return any arbitrary value from the domain.

Nowadays, cache memories have become integral part of modern computing systems. Cache memories and distribution of shared memory among multiple processors further complicate the read and write behavior on shared variables. In such system, several processors can independently and simultaneously issue memory operations on the same shared variable. Distributed memory systems typically use messages for their communication through interconnection networks. But, for programming convenience, process communication is abstracted into reads and writes (referred as distributed shared memory). In these systems, reads and writes are translated into (send and receive) messages at the lower level. These messages could be executed out of order, resulting in very complex and unpredictable behavior, which is not suitable for many real world applications. So, in distributed shared memory systems, the usual concepts, such as "the latest write," "next read," and "next write," applicable to the traditional shared memory, become unclear. This raises the question of what kind of memory behavior is acceptable during concurrent accesses to shared memory. The answer to this question depends on the application requirement. Vast majority of the applications require predictable or deterministic memory behavior. So, without proper control over the memory accesses, modern systems are not applicable to concurrent programming for majority of real world applications [15], [5].

There are several proposals in the literature to characterize behavior of memory accesses, suitable for different kind of application requirements. These characterizations are collectively referred to as memory consistency models, basically defining what orderings of read and write events on the shared variables are legal. The most popular memory consistency model, again due to Lamport, is sequential consistency, defined as follows:

Definition 2.2 (Sequential Consistency). *A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [20].*

Comprehensive reviews of shared memory consistency models can be found in [26], [10], [7]. Among the consistency models, sequential consistency is most simple and intuitive model that naturally extends the memory behavior of uniprocessor system to multiprocessors systems. Although implementing sequential consistency is considered to be costly, vast majority of applications require this level of consistency [15], [5]. Several approaches have been proposed in the literature to implement sequential consistency [22], [9], [14]. For our algorithms to work with distributed shared memory, we assume sequential consistency.

A5. *If the shared memory is distributed, then we assume that it assures* **sequential consistency**.

To define fault tolerance, Taubenfeld introduces two types of process failures in this context [30].

Definition 2.3. *The two types of process failures defined in [30] are as follows:*

1. Immediate failure: *When a process p fails, it returns to its noncritical section and the values of shared variables p owns are immediately reset to their initial default values.*
2. Continuous failure: *When a process p fails, it returns to its noncritical section and the values of shared variables p owns may assume arbitrary values for a period of time, but eventually must be reset to their initial default values (until p is restarted).*

When a process fails, first, it simply stops without any disruption to the system, and second, it recovers to a safe state by returning to noncritical section and resetting the values of its shared variables to initial default values.

Definition 2.4 (Fault Tolerance [30]). *An algorithm satisfies a given property in the presence of m failures, if the algorithm satisfies the properties for all correct processes, as long as no more than m failures occur.*

A6. *If a process fails, it fails immediately or continuously.*

3 The Basic LRU Algorithm

The basic idea behind the algorithm is very simple that, among the competing processes, the process which accessed R "least recently" succeeds to access R next. We implement this idea by using logical timestamp to record the latest access on R by each process. For the basic algorithm, we assume that the timestamps are unbounded in size. This assumption will be eliminated later for the improved algorithm presented in the next section.

The algorithm uses two binary arrays: c (to indicate the competition to access R) and $stage$ (to indicate the status in the competition), and one integer array ts (for timestamps). All the arrays are of size n. Initially, for every j, $c[j] := 0$, $stage[j] := 0$, and $ts[j] := j$.

Each process is expected to cross two logical stages, 0 and 1, to access R. The binary array $stage$ is used to record this progress. A process p starts its competition by setting its $c[p]$ to 1. Then, it checks to see if there are processes with lower timestamps competing for R. If no such process is found, then the process proceeds further to the stage 1. Otherwise, the process blocks itself until all the competing processes with lower timestamps complete their accesses on R and leave the competition.

Although it might happen rarely, it is quite possible that two or more processes find themselves, one after another, eligible to access R, and therefore, proceed further to the stage 1.

Consider that a process p with timestamp k starts its competition, finds no process with timestamp lower than k is competing, and therefore, decides to proceed further to the stage 1. Now, another process q with timestamp $j(< k)$ starts its competition, finds no process with timestamp lower than j is competing, and therefore, decides to proceed further to the stage 1. Safety may be violated if both p and q are allowed to proceed further and access R.

Shared Variables:
$c[n], stage[n]$: binary arrays of size n;
$ts[n]$: integer array of size n;

Initialization:
$[\forall j, c[j] := 0; stage[j] := 0; ts[j] := j];$

Entry Section:
1. $c[p] := 1;$
2. **do** {
3. $stage[p] := 0;$
4. **await**$(\forall q \neq p, (c[q] = 0) \vee (ts[q] > ts[p]));$
5. $stage[p] := 1;$
6. }**while** $(\exists q \neq p,\ stage[q] = 1)$

Exit Section:
7. $ts[p] := max\{ts[j]/1 \leq j \leq n\} + 1;$
8. $stage[p] := 0; c[p] := 0;$

Fig. 2. The LRU algorithm.

To assure safety, a condition must be introduced to facilitate further progress. That is, a process in the stage 1 can proceed further to access R only if no other process is in the stage 1. If more than one process is in the stage 1, then all of them go back to the stage 0 and recheck their priorities (timestamps) to determine which can go to the stage 1 under the current circumstance. That is, only the process with highest priority can go to the stage 1. Since the priorities of these processes are totally ordered, only one among them will have the highest priority and that process can reach the stage 1. All other processes will be blocked. Now the process which reaches the stage 1 alone can proceed further to access R.

If, again, more than one process reaches the stage 1 due to new arrivals, then they all go back to the stage 0, and the pattern repeats until one process reaches the stage 1 alone. Since the system has only a finite number of processes, eventually one process will reach the stage 1 alone. This gives the basic LRU algorithm. Entry section, exit section, and the shared variables declaration of LRU algorithm for process p are presented in Fig. 2.

After completing an access on R, in the exit section, the process chooses a timestamp one greater than the current maximum. This timestamp value will be used for the determination of its priority during the next competition for R.

Note: The max function used in the exit section may be replaced with a simple increment using an extra integer variable tsc (timestamp counter), initialized to n. That is, $ts[p] := max\{ts[j]/1 \leq j \leq n\} + 1$ now will become $ts[p] := ++tsc$. This change will not affect the safety, liveness, LRU fairness, and nonatomicity properties. But it introduces a centralized control through tsc.

3.1 Correctness

The entry section of the algorithm can be viewed as having two logical components: a *safety net* and a *filter*. The **do-while** loop (lines 2 and 6 combined) acts as the safety net and **await** statement (line 4) acts as the filter. Also, the processes in the entry section have to cross two logical stages: 0 and 1, and the filter acts as the boundary line between these two stages.

The safety net allows at most one process to pass through and access R at any time. That is, a process in the stage 1 goes back to the stage 0 if it finds another process in the stage 1. The filter blocks all but the highest priority process so that, at any moment, the process with the highest priority can cross the safety net and access R.

In essence, the safety net and the filter are designed to jointly facilitate the accesses on R in LRU fashion. The priority (i.e., the ts value) for the next access on R is computed at the end of the current access on R. The formal correctness proofs are presented next.

Theorem 3.1. *The LRU algorithm assures safety.*

Proof. We need to prove that, at any moment, at most one process can access R. Assume that two processes p and q, respectively, set their $stage$ values in line 5 during the time intervals $[t^p_{(w,s)}, t^p_{(w,e)}]$ and $[t^q_{(w,s)}, t^q_{(w,e)}]$, read other's $stage$ value in line 6 during the time intervals $[t^p_{(r,s)}, t^p_{(r,e)}]$ and $[t^q_{(r,s)}, t^q_{(r,e)}]$. Here the subscripts r and w, respectively, refer to $read$ and $write$ operations, and s and e, respectively, refer to the start and end of the operation. Assume that these are their latest operations on $stage[p]$ and $stage[q]$.

Since the operations within each process are totally ordered (by sequential consistency in case of distributed shared memory),

$$t^p_{(w,e)} < t^p_{(r,s)} \quad \text{and} \quad t^q_{(w,e)} < t^q_{(r,s)} \text{ ------ (a).}$$

For the process p to cross the line 6, it must have read all other processes' $stage$ values as 0. Particularly, p must have obtained 0 from its latest read on $stage[q]$. This read on $stage[q]$ in the line 6 must have started before q has completed its latest write on $stage[q]$ in line 5. That is,

$$t^p_{(r,s)} \leq t^q_{(w,e)} \text{ ------ (b).}$$

From (a) and (b), $t^p_{(w,e)} < t^p_{(r,s)} \leq t^q_{(w,e)} < t^q_{(r,s)}$. This implies that $t^p_{(w,e)} < t^q_{(r,s)}$. So the latest read of q on $stage[p]$ must have started well after p has completed its writing the value 1 on $stage[p]$. In that case, q must have read the value of $stage[p]$ as 1 and therefore cannot cross the line 6 to access R.

Similar analysis holds true if q has crossed the line 6 to access R. Therefore, at any time, not more than one process can cross the line 6 and access R. Hence, safety is assured. □

Theorem 3.2. *The LRU algorithm assures that when one or more processes interested in accessing R, one of them will eventually access R.*

Proof. The system starts with totally ordered timestamps. When a process leaves the competition, it updates its timestamp (at line 7) to one greater than the current maximum of timestamps. This update maintains total order. Since updates on timestamp are sequential, the system always maintains total order on timestamps.

We need to prove that when processes are competing for an access on R, eventually one process will access R. If there is only one process trying to access R, then it can complete the lines 1-6 without blocking, and hence, it can access R in a finite time.

Assume that more than one process are interested in accessing R. Since timestamps (which indicate priorities) are totally ordered, except the process with the highest priority (i.e., lowest ts value), all other processes will be blocked at the line 4. Note that the condition in the line 4 uses only c and ts values and they are updated at most once (no update on ts and one update on c) in the entry section. Therefore, no perpetual overlapping reads and checks, in the line 4 on c or ts values, are possible. Since at most n processes are in the system, in the worse case, eventually all but one process will be blocked at the line 4. So the process with the highest priority can eventually cross the line 6 and access R. □

Theorem 3.3. *The LRU algorithm assures freedom from starvation.*

Proof. We need to prove that no individual process will be blocked in the entry code forever. By Theorem 3.2, when some processes are interested in accessing R, eventually one process will do. This process, after accessing R, acquires the lowest priority for the next access so that it cannot bypass the currently competing processes when it comes back for another access. That is, when a process comes back with a priority lower than the priority of currently competing processes, it will be blocked at line 4 until all other higher priority processes complete their accesses on R. As the processes complete their accesses on R and leave one by one with priorities lower than the currently competing processes, eventually every competing process will have the highest priority and access R. □

To prove the next theorem on fairness, we assume that the size of ts is unbounded.

Theorem 3.4. *In LRU algorithm, if a process p starts with priority k (among the n processes in the system) then: (a) no process with priority lower than k can overtake p before q has its turn to access R, and (b) no process with priority higher than k can overtake p more than once before q has its turn to access R.*

Proof. Assume that among the processes p and q, p starts first by setting its c value before q does. There are two cases possible.

$ts[p] < ts[q]$: When q reaches the line 4, it will notice that $c[p] = 1$ and therefore will be blocked there until p completes its access on R and acquires a priority lower than that of q. So q cannot overtake p. This proves part (a) of the theorem.

$ts[p] > ts[q]$: In this case, even though p has started earlier, q may overtake p due to its LRU priority over p. But, when leaving the competition after its access on R, q will acquire a lower priority than that of p, and therefore, cannot overtake p anymore until p completes an access on R. This proves part (b) of the theorem. □

Corollary 3.5. *In LRU algorithm, if a process p has priority k when it starts its competition for accessing R then the maximum number of overtakes possible over p is $k-1$.*

Theorem 3.6. *The LRU algorithm is fault tolerant (as per Definition 2.4).*

Proof. We need to prove that the algorithm allows processes to fail at any time, yet it works for the remaining processes satisfying safety, liveness, and freedom from starvation, and fairness. Suppose a process p fails, by Definition 2.3, it eventually returns to its noncritical section and the values of $c[p], stage[p]$, and $ts[p]$, respectively, will be reset to their default values $0, 0$, and p. This is equivalent to the state that the process p is not competing. That is, failure of a process 1) does not affect the values of any other shared variables, and 2) is indistinguishable from the state of not competing for the critical section. Since the algorithm does not require all the processes to compete all the time to satisfy safety, liveness, and freedom from starvation, and fairness properties, it can work for the remaining processes satisfying these properties. □

Shared Variables:
$c[n]$: binary array of size n;
$tk[n]$: integer array of size n;

Initialization:
$[\forall j, c[j] := 0; tk[j] := 0];$

Entry Code:
1. $c[i] := 1;$
2. $tk[i] := 1 + max(\{tk[j]/1 \leq j \leq n\});$
3. $c[i] := 0;$
4. **for** $j := 1$ to n {
5. **await** $(c[j] = 0);$
6. **await** $(tk[j] = 0) \vee ((tk[i], i) \preceq (tk[j], j));$
7. }

Exit Code:
8. $tk[i] := 0;$

Fig. 3. Bakery algorithm [19].

3.2 Analysis

There are many classes of mutual exclusion algorithms, such as classical, fast, adaptive, local spin, etc., presented in the literature [25], [1], [4], [30], [11], and our algorithm belongs to the class of classical algorithms by Dijkstra, Knuth, Lamport, and Peterson. Particularly, *LRU* algorithm has many similarities to the bakery algorithm: it is simple, symmetric, fault tolerant under similar condition, fair, and nonatomic. So we restrict our analysis mostly related to the bakery algorithm which is given in Fig. 3, for process i. (In the algorithm, $((tk[i], i) \preceq (tk[j], j))$ means $(tk[i] < tk[j]) \vee ((tk[i] = tk[j]) \wedge (i < j))) \vee (i = j)$.

In the bakery algorithm, token values are computed in the entry code and used for the current competition. In our algorithm, they are computed in the exit code and used for the subsequent competition. In the bakery algorithm, due to concurrent competition, more than one process can compute same token value. As a result, the tie is broken statically using process ids. That is, in such situations, process with lowest id always wins. This may be considered as a fairness issue. Therefore, such static priority assignment has been apparently discouraged in the literature ("The solution must satisfy the following requirement: the solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority" [8]). Also, there seems to be no simple way to rectify this limitation. Our algorithm does not have this issue because no two processes can

compute same timestamp and total order on timestamps are always maintained.

The bakery algorithm is immune to a certain type of process failure where a process may fail and recover. Under immediate and continuous failure models, both the bakery algorithm and LRU algorithm are fault tolerant.

Next we analyze about nonatomicity—one of the most important and interesting properties in this context. Among the algorithms exist in the literature, only a few algorithms satisfy nonatomicity property [30] and most of them are quite complex. In a nonatomic model, reads can overlap with writes and an overlapping read may return any arbitrary value. The bakery algorithm and the LRU algorithm proposed in this paper are nonatomic algorithms.

The proof of the safety property of LRU algorithm is based only on $stage$ values and it does not require atomic access on $stage$. Inconsistent reads on ts are possible only when an outgoing process updates its ts value and that does not affect the safety and fairness of the algorithm. This is because the process which updates its ts anyway exits the competition right away.

FIFO and LRU are well-known fairness criteria and the bakery algorithm satisfies "doorway-based" FIFO (an approximate FIFO) and LRU algorithm satisfies LRU fairness in accessing the shared resource. In some practical systems, LRU may be considered more acceptable fairness criterion. Consider a system where the resource size is fixed (for example, number of tickets for a concert) and each access on the shared resource is expected to consume some amount of this finite resource. Here, for a balanced consumption of resources, LRU may be considered more fair than FIFO.

With respect to FIFO fairness, our algorithm assures that no process can overtake another process more than once. In the context of shared memory system, absolute FIFO is impossible anyway. Assume that two processes p and q start their competition for R at times t_p and t_q by writing some values to shared variables (for example, in our algorithm, $c[p] := 1$ and $c[q] := 1$ at line 1). Absolute FIFO requires either the values of t_p and t_q or their order. These values must come from a global clock, which itself is a shared variable. The global clock and c are two distinct shared variables and access to them, by Assumption A3, is independent and nondeterministic. So there is no way that the times or the order of writes to c can be derived by reading the global clock. Thus, the best FIFO fairness we can aim for in this context is that no process overtakes another process more than once. The LRU algorithm satisfies this level of FIFO fairness.

Both the bakery algorithm and the LRU algorithm are symmetric. That is, both algorithms do not use any special global variable (for example, the global variable like $turn$ used in the algorithms by Dijkstra and Knuth) for some kind of centralized control. The entry and exit sections for each process are symmetric.

The bakery algorithm assumes unbounded size shared space to hold timestamp in order to assure safety. The LRU algorithm assumes ts values to be unbounded to avoid starvation.

Assume that a ts can hold the maximum value N and the process p computes its ts value as N. Any later process will compute the timestamp value as $N+1$ and that might cause an overflow resulting a value less than N as its ts value. That means a later process can overtake p. This way of overtaking on p can repeat any number of times as long as processes keep competing for R continuously. So there is a possibility in LRU algorithm that a process can starve if ts value is bounded.

Several modifications are proposed in the literature to bound the timestamp values in the bakery algorithm [16], [17], [29], [30], [31], [32], [33]. These modified algorithms are: 1) quite complex compared to the original algorithm; 2) use extra variables, extra processes, or complex operations; and 3) compromise nonatomicity property—the most attractive property of the original algorithm. In contrary, our algorithm requires a minor modification to bound the timestamp values. More importantly, the resulting algorithm retains all the properties of the original algorithm, including nonatomicity property, and does not require any extra variables.

When a ts value reaches a specified maximum value, the ts values of all the processes are simply reset to a set of smaller values. If the maximum timestamp value $N \gg n$ (the number of processes), which is usually the case, then this reset occurs occasionally.

4 Bounded-Timestamps-Based LRU Algorithm

Every access to R increments the timestamp value by 1. For example, if ts is a 32 bit shared variable then the maximum value ts can hold is $N = 2^{32} - 1 = 4{,}294{,}967{,}295$. In this case, the timestamp can overflow after $4{,}294{,}967{,}296 - n$ accesses to R. To avoid starvation, ts values must be reset to lower values as soon as it reaches the maximum.

When a timestamp value reaches the maximum value N (we assume that $N \geq 2n$), the timestamp values of all the process are reset to their initial default values (i.e., $\forall j, ts[j] := j$). This is done in the exit section by adding a condition statement to check whether the timestamp has reached the maximum limit N and reset the values if the condition is true.

No extra shared variable is needed for this modification and no change is needed in the entry section of the algorithm. The only change is in the exit section and the modified algorithm, referred to as $BLRU$, is given in Fig. 4. Underlining is used to expose the added code explicitly.

4.1 Correctness

Theorems 3.1 and 3.2 remain valid for bounded-timestamp-based LRU algorithm, and the arguments of proofs still hold good. Theorem 3.4 holds good when no timestamp reset is encountered.

Lemma 4.1. *In bounded-timestamp-based LRU algorithm, no process will encounter more than one reset on ts values during its wait for a turn to access R.*

Proof. A reset on ts values can occur when a ts value reaches N. The reset brings all the ts values to their initial default values. That is, the priorities are reset between 1 and n. Then by Corollary 3.5, at most $n-1$ overtakes are possible over a process. These overtakes can increment ts

Shared Variables:
$c[n], stage[n]$: binary arrays of size n;
$ts[n]$: integer array of size n;

Initialization:
$[\forall j, c[j] := 0; stage[j] := 0; ts[j] := j];$

Entry Section:
1. $c[p] := 1;$
2. **do** {
3. $stage[p] := 0;$
4. **await**$(\forall q \neq p, (c[q] = 0) \vee (ts[q] > ts[p]));$
5. $stage[p] := 1;$
6. }**while** $(\exists q \neq p,\ stage[q] = 1)$

Exit Section:
7. $ts[p] := max\{ts[j]/1 \leq j \leq n\} + 1;$
X. **if** $(ts[p] \geq N)$ **then for** $j := 1$ **to** n $\{ts[j] := j\ \}$
8. $stage[p] := 0; c[p] := 0;$

Fig. 4. $BLRU$ algorithm.

at most $n-1$ times resulting a maximum ts value less than $2n$. So once a reset occurs, no more reset on ts is possible before all the current processes complete their accesses on R. □

Theorem 4.2. *Assume that in BLRU algorithm a reset occurs during the period a process p with priority k (before reset) waits for its access on R. Then: (a) no process with priority lower than k can overtake p more than once before q has its turn to access R, and (b) no process with priority higher than k can overtake p more than twice before q has its turn to access R.*

Proof. By Lemma 4.1, at most one reset on ts values is possible during a wait for an access to R. Assume that a process q with priority lower than that of p starts its competition after p does. By Theorem 3.4(a), q cannot overtake p before a reset on ts values. By Theorem 3.4(b), q can overtake p at most once after the reset on ts values. This completes the proof of part (a) of the theorem.

Assume that a process r with priority higher than that of p starts its competition after p does. Now r may overtake p in accessing R at most twice—one before the reset and one after the reset. This proves part (b) of the theorem. □

Corollary 4.3. *Assume that a process p starts with priority k. Then: (a) the maximum number of overtakes possible over p before it gets its turn to access R is $n+k-2$, and (b) the number of overtakes possible on average is $n-1$.*

Of course, in the best case, no process may overtake another process. The worst case overtakes can occur if $k = n$ and again receive the priority n due to a timestamp reset. Now the maximum number of overtakes possible over p is $2n-2$ ($n-1$ before the reset and $n-1$ after the reset).

4.2 Analysis

The proposed modification to bound ts values is primarily aimed to retain the simplicity of the original algorithm. The ts values may be bounded in many other ways to restrict the worst case overtakes to its optimal bound $n-1$. For example, the processes may be sorted based on the current priority order and mapped their ts values accordingly to the range 1 to n, in that order. This would not alter the current priority order after the reset, and therefore, no additional overtakes will be introduced due to timestamp reset. However, we feel that such improvements would compromise the simplicity of the algorithm.

5 Conclusion

As Ben-Ari rightly points out in [4] that the concurrent and distributed program are no longer the esoteric subjects for graduate students and it is gaining importance in many practical applications, such as event-based implementations of GUI, real-time OS, multiuser games, sensor networks and embedded systems, etc. As multicore processors are becoming common everywhere, the free lunch is over [27] and the recent hardware trend indicates that the future programmers inevitably have to develop concurrent programs.

Synchronization of concurrent processes in accessing a shared resource is a fundamental issue in concurrent programming. A solution to this problem basically resolves the conflicting accesses to shared resource by suitably coordinating the concurrent processes. In essence, the conflict is resolved by enforcing mutually exclusive access to shared resource.

There are many mutual exclusion algorithms available in the literature [1], [2], [4], [11], [25], [30] assuring some sort of $FIFO$ fairness. This paper presents a simple solution which assures LRU fairness.

The solutions to the mutual exclusion problem vary in great extend assuming different levels of hardware supports and varying application requirements. Concurrency is supported in many layers from operating system kernel to application level programs. Also, the type of concurrent programs involved varies from application to application. These possibilities clearly indicate that no one class of solutions can be suitable for solving the mutual exclusion problem effectively in all situations. In a complex application requiring mutual exclusion in various components, different algorithms may be more suitable in solving the problem in different conditions.

Among the algorithms proposed in the literature, the bakery algorithm presented in [19] is the simplest and most popular mutual exclusion algorithm. It is based on the idea of using token numbers to resolve the conflict among the competing processes. The LRU algorithm proposed in this paper has many similarity to the bakery algorithm that it is simple, symmetric, fault tolerant under similar condition, fair, and nonatomic. In addition, the second algorithm BLRU uses bounded size shared space.

Nonatomic memories are weaker types of memories. Most multiport memories are of this weaker type, and in recent times majority of embedded systems come with such multiport memories. The algorithms proposed in this paper can be used to solve the mutual exclusion problem in the systems with nonatomic multiport memories.

Acknowledgments

The authors would like to thank the editor and the anonymous reviewers for their valuable comments and inputs which helped to improve the presentation.

REFERENCES

[1] J.H. Anderson, Y.-J. Kim, and T. Herman, "Shared-Memory Mutual Exclusion: Major Research Trends Since 1986," *Distributed Computing,* vol. 16, pp. 75-110, 2003.

[2] A. Aravind and W. Hesselink, "A Queue Based Mutual Exclusion Algorithm," *Acta Informatica,* vol. 46, pp. 73-86, 2009.

[3] H. Attiya, R. Guerraoui, D. Hendler, and P. Kouznetsov, "Synchronizing without Locks Is Inherently Expensive," *Proc. 25th ACM Symp. Principles of Distributed Computing (PODC '06),* pp. 300-307, 2006.

[4] M. Ben-Ari, *Principles of Concurrent and Distributed Programming.* Addison-Wesley, 2006.

[5] R. Bocchino, V. Adve, S. Adve, and M. Snir, "Parallel Programming Must Be Deterministic by Default," *Proc. First USENIX Workshop Hot Topics in Parallelism (HotPar),* 2009.

[6] C. Cascaval, C. Blundell, M. Michael, H.W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why Is It Only a Research Toy?" *Comm. ACM,* vol. 51, no. 11, pp. 40-46, 2008.

[7] V. Cholvi, E. Jimenez, and A.F. Anta, "Interconnection of Distributed Memory Models," *J. Parallel and Distributed Computing,* vol. 69, pp. 295-306, 2009.

[8] E.W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Comm. ACM,* vol. 8, no. 9, p. 569, 1965.

[9] X. Fang, J. Lee, and S.P. Midkiff, "Automatic Fence Insertions for Shared Memory Multiprocessing," *Proc. 17th Ann. Int'l Conf. Supercomputing,* pp. 285-294, 2003.

[10] S. Haldar and K. Vidyasankar, "On Specification of Read/Write Shared Variables," *J. ACM,* vol. 54, no. 6, 2007.

[11] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, 2008.

[12] M. Herlihy and V. Luchangco, "Distributed Computing and the Multicore Revolution," *ACM SIGACT News,* vol. 39, no. 1, pp. 62-72, 2008.

[13] W. Hesselink and A. Aravind, "Queue Based Mutual Exclusion with Linearly Bounded Overtaking," *Science of Computer Programming,* in press.

[14] L. Higham and J. Kawash, "Implementing Sequentially Consistent Programs on Processor Consistent Plateforms," *J. Parallel and Distributed Computing,* vol. 68, pp. 488-500, 2008.

[15] M.D. Hill, "Multiprocessors Should Support Simple Memory-Consistency Models," *Computer,* vol. 31, no. 8, pp. 28-34, Aug. 1998.

[16] A. Israeli and M. Li, "Bounded Time-Stamps," *Distributed Computing,* vol. 6, no. 4, pp. 205-209, 1993.

[17] P. Jeyanti, K. Tan, G. Friedland, and A. Katz, "Bounding Lamport's Bakery Algorithm," *Proc. Conf. Current Trends in Theory and Practice of Informatics Piestany: Theory and Practice of Informatics (SOFSEM),* pp. 261-270, 2001.

[18] D.E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control," *Comm. ACM,* vol. 9, no. 5, pp. 321-322, 1966.

[19] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Comm. ACM,* vol. 17, no. 8, pp. 453-455, 1974.

[20] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers,* vol. C-28, no. 9, pp. 241-248, Sept. 1979.

[21] L. Lamport, "The Mutual Exclusion Problem: Part II—Statement and Solutions," *J. ACM,* vol. 33, no. 2, pp. 327-348, 1986.

[22] J. Lee and D.A. Padua, "Hiding Relaxed Memory Consistency with a Compiler," *IEEE Trans. Computers,* vol. 50, no. 8, pp. 824-833, Aug. 2001.

[23] M. Martin, C. Blundell, and E. Lewis, "Subtleties of Transactional Memory Atomicity Semantics," *IEEE Computer Architecture Letters,* vol. 5, no. 2, pp. 17-20, 2006.

[24] G.L. Peterson, "Myths about the Mutual Exclusion Problem," *Information Processing Letters,* vol. 12, no. 3, pp. 115-116, 1981.

[25] M. Raynal, *Algorithms for Mutual Exclusion Problem.* The MIT Press, 1986.

[26] R.T. Steinke and G.J. Nutt, "A Unified Theory of Shared Memory Consistency," *J. ACM,* vol. 51, no. 5, pp. 800-849, 2004.

[27] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's J.,* vol. 30, no. 3, pp. 202-210, Mar. 2005.

[28] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *ACM Queue,* vol. 3, no. 7, pp. 54-62, Sept. 2005.

[29] G. Taubenfeld, "The Black-White Bakery Algorithm and Related Bounded-Space, Adaptive, Local-Spinning and FIFO Algorithms," *Proc. Int'l Symp. Distributed Computing (DISC),* pp. 56-70, 2004.

[30] G. Taubenfeld, *Synchronization Algorithms and Concurrent Programming.* Pearson Education/Prentice Hall, 2006.

[31] M. Takamura and Y. Igarashi, "Simple Mutual Exclusion Algorithms Based on Bounded Tickets on the Asynchronous Shared Memory Model," *Proc. Ann. Int'l Conf. Computing and Combinatorics (COCOON),* pp. 259-268, 2002.

[32] S. Vijayaraghavan, "A Variant of the Bakery Algorithm with Bounded Values as a Solution to Abraham's Concurrent Programming Problem," *Proc. Conf. Design, Analysis and Simulation of Distributed Systems,* 2003.

[33] T-K. Woo, "A Note on Lamport's Mutual Exclusion Algorithm," *SIGOPS Operating Systems Rev.,* vol. 24, no. 4, pp. 78-81, 1990.

Alex A. Aravind received the MTech degree in computer science from the Indian Institute of Technology (IIT), Kharagpur, India, and the PhD degree in computer science from the Indian Institute of Science (IISc), Bangalore, India. He was a postdoctoral fellow at the Memorial University of Newfoundland, St. John's, Canada, prior to joining the University of Northern British Columbia, Prince George, Canada. He is currently an associate professor in computer science at the University of Northern British Columbia. He is a coauthor of the book *Operating Systems*, published by Pearson Education. His research interests include operating systems, distributed computing, and wireless sensor networks. He is a member of the ACM, IEEE, and SCS.