A Turn Function Scheme Realized in the Asynchronous Single-Writer/Multi-reader Shared Memory Model

Tom Altman[1], Yoshihide Igarashi[2], and Michiko Omori[3]

[1] Department of Computer Science and Engineering, University of Colorado at Denver, Denver, CO 80217, USA
taltman@carbon.cudenver.edu
[2] Department of Computer Science, Gunma University, Kiryu, Japan 376-8515
igarashi@comp.cs.gunma-u.ac.jp
[3] Business Networks Division, NEC Corp., Abiko, Chiba, 270-1198, Japan
m-oomori@dc.jp.nec.com

Abstract. We consider a set of users wishing to receive a service in an asynchronous distributed system. Such users declare their wishes and then wait to gain admittance to be served. Except for the initial transient period, at least one user must be waiting to be served, and the system should be as fair as possible for users. A procedure that ensures such a situation is called a *turn* function. It can be easily implemented in the asynchronous multi-writer/reader shared memory model. The *turn* function is useful to solve some problems concerning temporal orders within an asynchronous distributed system (e.g., the mutual exclusion problem and the k-exclusion problem). In this paper we propose an algorithm for the *turn* function that can be implemented in the single-writer/multi-reader shared memory model. Its implementation is simpler than the general method of simulating operations of multi-writer/reader shared variables by using a bounded concurrent time-stamp scheme in the single-writer/multi-reader shared memory model. We show the correctness of our algorithm and discuss its efficiency.

1 Introduction

The shared memory is an abstraction of asynchronous interprocess communication, where the senders and the receivers correspond to the writers and the readers, respectively. Mutual exclusion is a problem of managing access to a single indivisible resource that can only support one user at a time. It is well known as an important and fundamental problem in the area of asynchronous distributed systems. The n-process algorithm by Peterson [12] and the tournament algorithm by Peterson and Fischer [13] are well known lockout free mutual exclusion algorithms. These algorithms use multi-writer/reader shared variables to decide which of the processes is the last arrival. The accelerated version of n-process algorithm by Igarashi and Nishitani [6] also uses multi-writer/reader shared variables.

The k-exclusion problem is a natural generalization of the mutual exclusion problem. In k-exclusion, some number of processes, specified by the parameter k, are allowed to be concurrently inside the critical region, where corresponding users can use the resource. This generalization was defined and solved by Fischer et al. in the shared memory model [4]. Afek et al.[1] gave a *first-in* and *first-enable* solution to the k-exclusion problem. Their algorithm uses time-stamps. However, bounded concurrent time-stamp schemes known so far contain complicated construction in order to give temporal relations among events in an execution [2,3,5]. We can easily design algorithms for the k-exclusion problem if we are allowed to use multi-writer/reader shared variables as shown in [10,11]. However, the realization of multi-writer/reader shared variables is not easy.

The multi-writer/reader shared variables, named *turn*'s, used in [6,10,11,12, 13] have the same function. If every process can write its identifier in a multi-writer/reader shared variable, by checking the value of the shared variable any process can easily decide which process is the last writer to the shared variable. We call this function a *turn* function. The *turn* function is useful to solve problems in an asynchronous distributed system where a temporal order of events in an execution is required. The mutual exclusion problem and the k-exclusion problem are such problems.

An algorithm for simulating multi-writer/reader shared variables by single-writer/multi-reader variables has been proposed [8]. However, it uses bounded concurrent time-stamps. In this paper we propose an algorithm without using time-stamps for the *turn* function in the single-writer/multi-reader shared memory model. The structure of our algorithm is simple, and it has efficient running cost. By using our algorithm for the *turn* function, the algorithms given in [6, 10,11,12,13] can also be implemented, without using time-stamps, in the single-writer/multi-reader shared memory model. The algorithms in [10,11] for the k-exclusion problem satisfy lockout avoidance (i.e., they are immune to stopping failures of fewer than k processes). Simulation of those algorithms for the k-exclusion problem, by using our *turn* function algorithm, preserves lockout avoidance as well. In Section 3 we describe our algorithm. We prove the correctness of the algorithm and discuss its efficiency in Sections 4 and 5.

2 Preliminaries

The asynchronous single-writer/multi-reader shared memory model is used in this paper. It is a collection of n processes and single-writer/multi-reader shared variables. Each user of the system corresponds to its own process. Interactions between a process and its corresponding user are by input actions from the user to the process and by output actions from the process to the user. A single-writer/multi-reader shared variable can be written by at most one process, but can be read by multiple processes. All communication among the processes is via the shared memory. A full description of an asynchronous shared memory system can be given as such a model [9].

Access to a shared variable has a time duration starting from an invocation and ending with a response. We do not make assumptions about relative speeds of the processes that are communicating via shared variables. Since we only use single-writer/multi-reader shared variables in our algorithm, write operations do not overlap each other. Overlapping read operations are assumed not to affect each other. However, as pointed out by Lamport [7], when a read operation from a shared variable overlaps one or more write operations to the same shared variable, various situations can be considered. Lamport defined three categories for single-writer/multi-reader shared variables according to possible assumptions about what can happen in concurrent cases of read operations and write operations. These are *safe*, *regular*, and *atomic* shared variables [7]. For simplicity, in this paper we assume that all shared variables are atomic. That is, we assume that for any execution in the system, there is some way of totally ordering the read operations and write operations as if the operations are performed in that order without overlapping.

Suppose that a set of users wishing to receive a service is in the system. Such users declare their wishes and then wait to gain admittance to be served. Except for the initial transient period, at least one user must be waiting for the service. A procedure to keep such a situation is called the *turn* function. The system should be as fair as possible.

A user who is receiving the service is modeled as being in the service region. When a user is not involved in the service, he/she is said to be in the remainder region. In order to gain admittance to be in the service region, a process executes a trying protocol. The duration of time from the start of executing the trying protocol to the entrance of the service region is called the waiting time. When a user wishes to leave the service region, he/she sends his/her wish (i.e., the message that he/she wishes to exit from the service region) to the corresponding process. Once the process receives the wish of exit, it sends its corresponding user a message telling that he/she may return to the remainder region. For each user (as well as his/her corresponding process), visits to these regions can be repeated in cyclic order, from the remainder region to the waiting region, then to the service region, and then back again to the remainder region.

We assume that the n processes are numbered $1, \cdots, n$. Each process i corresponds to user U_i ($1 \leq i \leq n$). The inputs to process i from user U_i are $wish_i$ which means a request by user U_i for access to the service region, and $exits_i$ which means an announcement of the end of receiving the service by U_i. The outputs from process i to user U_i are $serve_i$ which means the grant of the service to U_i, and rem_i which tells U_i that it can continue with the rest of its work with no relation to the service. The system to manage the *turn* function should satisfy the following conditions.

(1) Once any process enters the waiting region, at least one process should stay in the waiting region.
(2) If at least two faultless processes are in the waiting region, then at some later point some process enters the service region.
(3) The interaction between users and processes are well formed.

Conditions (1), (2) and (3) above are called non-emptiness of the waiting region, progress for the waiting region, and well formedness between users and processes. Progress for the waiting region is a weaker property than lockout freedom ((4) below), and lockout freedom is a weaker property than lockout avoidance ((5) below).

(4) Any process wishing to be served will eventually enter the service region if every process is faultless, and if a process will enter the waiting region within a finite length of time from any point in time.
(5) Any faultless process wishing to be served will eventually enter the service region if a faultless process will enter the waiting region within a finite length of time after any point in time.

The fairness of the *turn* function can be easily satisfied if we are allowed to use a multi-writer/reader shared variable. Ideally, every process in the waiting region except for the last arrival at the waiting region should be promptly permitted to enter the service region. This property is called the *promptness*. It is straightforward to give a protocol for the *turn* function satisfying fairness and promptness if we are allowed to use a multi-writer/reader shared variable as shown in the following procedure, *mw/r-turn*(n).

```
procedure $mw/r$-turn($n$)
shared variable:
  $turn$, initially arbitrary, writable and readable by all processes;

process $i$    $(1 \le i \le n)$
  input actions {inputs to process $i$ from user $U_i$}: $wish_i$, $exit_i$;
  output actions {outputs from process $i$ to user $U_i$}: $serve_i$, $rem_i$;

  ** remainder region **

01:   $wish_i$:
02:      $turn := i$;
03:      waitfor $turn \neq i$;
04:      $serve_i$;

  ** service region **

05:   $exit_i$:
06:      $rem_i$;
```

3 An Algorithm without Using Multi-writers

In this section we give an algorithm, called procedure *n-turn*, for the *turn* function in the single-writer/multi-reader shared memory model, where n is the number of users as well as the number of processes. The algorithm proposed here does not use time-stamps.

The waiting region is essentially divided into six parts/sections. In the $n \times n$ shared array *PView*, each row i is writable by process i and readable by all. The same applies to *WRView*$[1..n]$. An entry of "1" in position *PView*$[i, q]$ means that process q was already in the waiting region when process i got there. Other values for the entries will be defined as they are introduced.

Below, we discuss each section in greater detail:

– *Registration section* (lines $02 - 03$): Process i checks in.
– *Give Priority section* (lines $04 - 08$): Process i checks and writes which processes were originally and/or are currently (to the best of its knowledge) in the waiting region, and it also identifies those processes that asked for priority, as well as those that gave priority to process i during its previous stay in the waiting region. If it finds a process q whose *PView*$[q, i]$ entry is equal to "3" (a notification by q to process i that q asks for priority), then process i enters a "2" in its *PView*$[i, q]$ location so that q may eventually proceed to the service region via line 12:b.
– *Prevent Cycle section* (lines $10 - 11$): Process i checks if by releasing itself via line 12:d it could produce a cycle of mutually releasing processes, if that is the case, i does not proceed.
– *Decide section* (line 12): Process i checks the *PView* array, as well as its own local memory $LPView_i$, to determine if it can proceed to the service region. Note that if a process q gave process i prior permission, then, since i has already left the waiting region and came back, process q should now have priority. This is enforced via line 12:c. If two or more processes happened to request priority from each other simultaneously, then all such processes, except the one with the highest identification number (*id* for short) will be allowed to proceed to the service region via line 12:d.
– *Ask for Priority section* (line 13): Getting here indicates that process i could not proceed to the service region and had to stay behind because it was the only one in the waiting region, or it tied with other processes, but it had the highest *id*, it really was the last process to enter the waiting region, or it simply could not make a definite determination. The bookkeeping in line 13 is done so that process i which had to stay behind in the waiting region (for whatever reason) will inform all processes q which did not already give or receive priority to or from i, that next time around they enter the waiting region, they must give priority to i. This section is also used to force a "$3-3$" tie for requests between processes that arrived simultaneously, which will eventually lead to an exit via line 12:d.
– *Check-out section* (lines $15-16$) Process i exits the waiting region and resets the shared variables as needed.

```
procedure n-turn
shared variables
  PView[1..n, 1..n], each row $i$ ($1 \leq i \leq n$) writable by process $i$ and
  readable by all, for each $(i, j)$ ($1 \leq i, j \leq n$) $PView[i, j] \in \{0, 1, 2, 3\}$,
  initially 0;
  WRView[1..n], each entry $i$ writable by process $i$ and readable by all,
  for each $i$ $WRView[i] \in \{0, 1\}$, initially 0;
  $Flag[1..n, 1..n]$, each row $i$ $(1 \le i \le n)$ writable by process $i$ and
  readable by all, initially **false**;

**process** $i$
  **input actions** {inputs to process $i$ from user $U_i$}: $wish_i$, $exit_i$;
  **output actions** {outputs from process $i$ to user $U_i$}: $serve_i$, $rem_i$;
  **local variables**: $LPView_i[1..n]$; $LWRView_i[1..n]$;

  ** remainder region **

01: $wish_i$:
02:   $WRView[i] := 1$;
03:   **for** $q := 1$ **to** $n$ **do begin**
        $LWRView_i[q] := WRView[q]$; $LPView_i[q] := PView[q, i]$ **end**;
04:   **for** $q := 1$ **to** $n$ **do if** $(i \neq q)$ **then**
05:     **if** $(LPView_i[q] = 2)$ **then** $PView[i, q] := 1$
06:       **else if** $(LPView_i[q] = 3)$ **then** $PView[i, q] := 2$
07:         **else if** $(LPView_i[q] = 1)$ **then** $PView[i, q] := 3$
08:           **else** $PView[i, q] := LWRView_i[q]$;
09:   **while true do begin**
10:     **for** $q := i + 1$ **to** $n$ **do** $Flag[i, q] :=$ **false**;
11:     **for** $q := i + 1$ **to** $n$ **do if**
          $(PView[i, q] = 1$ **and** $LPView_i[q] = 0)$ **or**
          $(PView[i, q] = 2$ **and** $LPView_i[q] = 3)$ **or**
          $(PView[i, q] = 1$ **and** $LPView_i[q] = 2)$
          **then** $Flag[i, q] :=$ **true**;
12:     **for** $q := 1$ **to** $n$ **do if**
  a:      $(LPView_i[q] = 1$ **and** $PView[i, q] = 0)$ **or**
  b:      $(LPView_i[q] = 2$ **and** $PView[i, q] = 3)$ **or**
  c:      $(LPView_i[q] = 1$ **and** $PView[i, q] = 2)$ **or**
  d:      $(LPView_i[q] = 3$ **and** $PView[i, q] = 3$ **and** $i < q$ **and**
           $((\neg Flag[i, q] \wedge \neg Flag[i, q + 1] \wedge ... \wedge \neg \mathrm{Flag}[i, n])$ **or**
           $(\exists\ (r, s),\ r, s > i \mid Flag[i, s] \wedge Flag[r, s])))$
          **then goto** 15;
13:     **for** $q := 1$ **to** $n$ **do if**
          $(LPView_i[q] = 0$ **and** $PView[i, q] = 2)$ **or**
          $(LPView_i[q] \neq 2$ **and** $PView[i, q] \neq 2)$ **then** $PView[i, q] := 3$;
14:     **for** $q := 1$ **to** $n$ **do** $LPView_i[q] := PView[q, i]$
      **end** (* **while** *);

15:   $WRView[i] := 0$;
16:   **for** $q := 1$ **to** $n$ **do begin**
        **if** $(PView[i, q] \neq 1)$ **then** $PView[i, q] := 0$;  $Flag[i, q] :=$ **false**
**end**;
17:   $serve_i$;

  ** service region **

18: $exit_i$:
19:   $rem_i$;
```

4 Correctness of the Algorithm

Let us first informally explain how and why the algorithm *n-turn* works. As stated before, most of the communication between the processes is carried out via the *PView* array. In fact, since its main diagonal is never used, the *WRView* array could have been represented there, but for clarity of presentation, we chose to use two separate arrays.

There are four possible values that may be entered into $PView[i, q]$. An entry of "0" means upon entering the waiting region, process i did not encounter process q, whereas an entry of "1" means that it did. Process i will request priority from process q by setting the $PView[i, q]$ entry to "3". This request will be granted by q upon its next visit to the waiting region, which will be indicated with an entry of "2" in *PView*, see line 06.

Each process i uses a local array $LPView_i[1..n]$ to store the contents of the ith column of the (shared) *PView* array. This approach is used in order to provide (for each process) as consistent a snapshot of *PView* array's contents as possible.

When deciding if process i may actually leave the waiting region, it must find proof that either one of four conditions (labeled 12: a through d) was met:

(a) A new process q arrived into the waiting region and it had originally not been seen by process i. (case when $PView_i[q] = 1$ and $PView[i, q] = 0$),
(b) i requested priority from q and it was granted (case when $PView_i[q] = 2$ and $PView[i, q] = 3$),
(c) i gave priority to q, but q no longer needs it as it reentered the waiting region (case when $PView_i[q] = 1$ and $PView[i, q] = 2$),
(d) If two or more processes happened to request priority from each other simultaneously, then all such processes, except the one with the highest *id* number will be allowed to proceed to the service region.

The above exit conditions are mutually exclusive and we will show that, together with the cycle prevention mechanism, implemented via the shared array *Flag*, they guarantee that when process i leaves, at least one process will remain in the waiting region.

Before proving that the waiting region cannot become empty (once it has been occupied by any process), we shall define several terms. For a pair of processes, e.g., i and q, process q is said to *release* process i, written $i <= q$, if the pair satisfies any of the exit conditions a–d. Note that q releasing i does not imply that q has to wait for i to exit the waiting region before q can proceed, i.e., q is not being *blocked* by i. For a pair of processes, i and q, process q is said to *implicitly release* process i, written $i <= *q$, if there exists a sequence of m ($m < n - 1$) unique processes such that $i <= q_1 <= q_2 <= ... <= q_m <= q$. Observe that the concept of implicit release preserves the temporal consistency enforced by the exit conditions. Furthermore, process q need not be present in the waiting region in order for it to release process i. As a process (re)enters the waiting region, and starts to assign its new priorities to other processes (lines 05 - 08), the previously defined release relationships naturally disappear. It follows that no process may appear more than once in any implicit release path.

For any fixed point in time, the release relationship defines a partial temporal order between the processes. In order to show that this order is consistent, we must prove that at no time directed cycles may occur in a precedence graph, called the $<=$-graph, whose nodes correspond to the processes and the directed edges correspond to the release relationships. The absence of cycles in such a $<=$-graph also implies that within any group of processes they cannot all grant releases to each other simultaneously and, thereby, make the waiting region empty.

In order to show that no directed cycles exist in $<=$-graphs, we will prove the existence of transitivity in a weak sense. It is possible for three processes to have the following relationship: $q_i <= q_j <= q_k$, but not the case of $q_i <= q_k$ because process q_i could have already left the waiting region and not even requested priority from q_k. For this reason, we define *pseudo-transitivity* in $<=$-graphs as an existence of a virtual release of process q_i by q_k if q_i would have been released by q_k had process q_j not come along.

Lemma 1. *The $<=$-graphs corresponding to two processes are acyclic.*

Proof. Conditions a–d directly disallow mutual release which makes the graph acyclic. Note that for any (non-failed) process pair, eventually one process would release the other in some finite number of steps. □

Lemma 2. *Pseudo-transitivity is preserved by the $<=$-graphs.*

Proof. Let us take three processes, q_i, q_j, and q_k and let us assume that $q_i <= q_j$ and $q_j <= q_k$. It must be shown that $q_i <= q_k$. We now have $q_i <=_1 q_j <=_2 q_k$, where a suffix of $<=$ shows a release type. There are sixteen possible pairings of release types for $<=_1$ and $<=_2$. They are {(a,a), (a,b), ..., (d,d)}. In the first fifteen instances, we have direct evidence that process q_k last came in either after q_j or after q_i, or both. This is because for condition (a), the released process was seen in the waiting region when the granting process arrived. For (b), the priority of "2" is given in response to a request of "3" as soon as the granting process *enters* the waiting region (see line 06), so the requesting process was already in the waiting region (and, hypothetically, may have even left it) when the granting process arrived. In case of (c), the process that granted priority "2" was in the waiting region (and, hypothetically, may have even left it) while the recipient left, got service, reentered the waiting region, and reset its entry to "1" in line 05. Note that any potential cycles involving release pairs that contain exactly one $<=_d$ release are prevented by checking the $flag_i$ condition in line 12:d. For the $q_i <=_d q_j <=_d k$ case, the only way it could take place is if $i < j < k$, making $q_k <= q_i$ impossible. Therefore, $q_i <= q_k$. □

Lemma 3. *The $<=$-graphs corresponding to n processes are acyclic.*

Proof. The proof is by induction on n, with the base case already covered by Lemma 1. Let us assume that all $<=$-graphs for up to k processes are always acyclic. We must show that an introduction of process q_{k+1} can not produce a cycle of length $k+1$. Cycles of shorter length would have been addressed by the inductive hypothesis. Let k processes be present when process q_{k+1} arrives. In order for a $k+1$ cycle to have a chance to exist, the k processes must have already formed a (directed) path via the release relations, which would have the form $q_1 <= q_2 <= ... <= q_k$. Process q_{k+1} would then have to release q_k and be released by q_1. Since $q_1 <= *q_k$, it follows that process q_k came in after q_1 (or, at least, not before), and by pseudo-transitivity $q_1 <= q_k$, i.e., q_k could have released q_1 directly. Therefore, for the cycle to exist, process q_{k+1} would have to be released by q_1, which cannot take place since by pseudo-transitivity $q_1 <= q_{k+1}$. □

Lemma 4. *Procedure n-turn will not allow the waiting region to become empty.*

Proof. The exits of processes from the waiting region may only take place when the conditions in line 12 are satisfied. For only two processes, conditions in line 12 prevent both from leaving at the same time. The remaining possibility, where three or more processes give mutual release to each other is also impossible because the corresponding <=-graph is acyclic. □

Lemma 5. *Procedure n-turn provides lockout avoidance.*

Proof. Let us assume that process i has entered the waiting region and failed to meet any of the conditions given in line 12. In getting to line 13, process i requests from the appropriate processes that it be given priority. Note that the conditions in line 13 will eventually be satisfied once some process q has exited the waiting region and reentered setting its $PView$ priorities to "1" or "2" in lines 06 or 07, respectively. Therefore, a process will always eventually be able to write down its priority request and proceed to the service region once one of the conditions in line 12 has been met. □

Theorem 1. *Procedure n-turn correctly implements the turn function.*

Proof. Follows directly from Lemmas 1–5. □

5 Conclusion

The implementation of the algorithm proposed here is much simpler than the general method of simulation required for the multi-writer/reader operations to be carried out using single-writer/multi-reader shared variables. As presented, the algorithm requires $O(n^2)$ bits of memory.

The running time (technically, the waiting time) for each processor in the waiting region is bounded by $O(n)l + c$, where n is the number of processes, l is an upper bound on the time between any two successive atomic steps, and c is an upper bound on time durations between which at least one process enters the waiting region. For all practical purposes, the algorithm is optimal, since each process should perform $O(n)$ operations just to establish the whereabouts of the remaining $n-1$ processes. The bound of c cannot be overcome as well (e.g., consider a situation such that there is only one process in the waiting region).

The algorithm is robust in that it will operate properly even in the presence of up to $n-2$ failures of the stopping type, although the algorithm of course cannot guarantee that the process in the waiting region is a non-failed one.

References

1. Y. Afek, D. Dolev. E. Gafni, M. Merritt, and N. Shavit, "A bounded first-in, first-enable solution to the l-exclusion problem", *ACM Transactions on Programming Languages and Systems*, vol.16, pp.939–953, 1994.
2. D. Dolev and N. Shavit, "Bounded concurrent time-stamp systems are constructible", *21st Annual ACM Symposium on the Theory of Computing*, New York, pp. 454–465, 1989.
3. C. Dwork and O. Waarts, "Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible", *24th Annual ACM Symposium on the Theory of Computing*, Victoria, B.C., pp. 655–666, 1992.
4. M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Resource allocation with immunity to limited process failure", *20th Annual Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, pp. 234–254, 1979.
5. S. Haldar and P. Vitanyi, "Bounded concurrent timestamps using vector clocks", *J. of the ACM*, vol.49, pp. 101–126, 2002.
6. Y. Igarashi and Y. Nishitani, "Speedup of the n-process mutual exclusion algorithm", *Parallel Processing Letters*, vol.9, pp. 475–485, 1999.
7. L. Lamport, "On interprocess communication, Part II: Algorithms", *Distributed Computing*, vol.1, pp. 86–101, 1986.
8. K. Li, I. Tromp, and P. M. B. Vitanyi, "How to share concurrent wait-free variables", *J. of the ACM*, vol.43, pp. 723–746, 1996.
9. N. A. Lynch, "Distributed Algorithms", Morgan Kaufmann, San Francisco, California, 1996.
10. K. Obokata, M. Omori, K. Motegi, and Y. Igarashi, "A lockout avoidance algorithm without using time-stamps", *7th Annual International Computing and Combinatorics Conference (COCOON 2001)*, Guilin, China, Lecture Notes in Computer Science, vol.2108, pp. 571–575, 2001.
11. M. Omori, K. Obokata, K. Motegi and Y. Igarashi, "Analysis of some lockout avoidance algorithms for the k-exclusion problem", *Interdisciplinary Information Sciences*, vol.8, pp. 187–198, 2003.
12. G. L. Peterson, "Myths about the mutual exclusion problem", *Information Processing Letters*, vol.12, pp. 115–116, 1981.
13. G. L. Peterson and M. J. Fischer, "Economical solutions for the critical section problem in a distributed system", *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, Boulder, Colorado, pp. 91–97, 1977.