An assertional proof of a lock synchronization algorithm using fetch_and_store atomic instructions

Ting-Lu Huang

Dept. of Computer Science
and Information Engineering
National Chiao-Tung University
1001 Ta-Hsueh Road
Hsin-Chu, Taiwan 30050
Republic of China

Jann-Hann Lin

Dept. of Computer Science
and Information Engineering
National Chiao-Tung University
1001 Ta-Hsueh Road
Hsin-Chu, Taiwan 30050
Republic of China

Abstract: A new lock synchronization algorithm, proposed independently by Craig and the authors, not only eliminates memory contention caused by process spinning but also preserves First_in_first_out property. A previous result, the MCS lock algorithm, requires both compare_and_swap and fetch_and_store instructions, or the FIFO property is lost and hence starvation may occur. The new one requires only fetch_and_store. We provide an assertional proof for the new algorithm. Most of behavioral proofs of concurrent programs are error-prone since it is difficult and tedious to take all possibilities of interleaving among the processes into consideration. An assertional proof replaces a large number of possibilities of interleaving by a small number of invariants. New techniques in this proof are : (1) an assertional characterization of token bit accessibility, (2) the definition of effective assignments that brings about the notion of token creation/destruction, (3) the definition of token count that derives the mutual exclusion theorem, (4) the constructing procedure of a token-list that faithfully records the arrival time sequence of lock requests so that FIFO ordering can be enforced.

1 Introduction

Synchronization among the processes in a shared-memory multiprocessor system is usually implemented by busy-waiting (spinning) on some shared variables. Spinning on a single shared variable causes serious memory contention problems since that variable can easily become a hot spot. Mellor-Crummey and Scott[6] proposed a lock synchronization algorithm(known as **MCS lock**) whereby each process spins on distinct variable. Memory contention is greatly reduced. However, the MCS lock requires that both *fetch_and_store* and *compare_and_swap* atomic instructions be available, or the first_in_first_out (FIFO) ordering property will be lost. Although FIFO ordering is not absolutely necessary for lock synchronization, at least some weaker ordering control is needed in order to prevent starvation. The alternate version of the MCS lock functioning in an environment where only *fetch_and_store* is available fails in this regard. The *fetch_and_store* instruction is much more common than *compare_and_swap.* For example, Sequent S27 multiprocessors have Intel 80386 as the processors. It has "XCHG" instruction which is a *fetch_and_store,* but it does not has *compare_and_swap*[8]. Craig[7] and Lin-Huang[2] independently proposed an algorithm for such environment that not only preserves the FIFO property, and therefore is starvation free, but also preserves the property that each process spins on a distinct token bit. For convenience of discussion, we name it as **Craig-LH algorithm**.

Our proof essentially follows the model used by Lamport[9]. In the model, read/write operation on a variable is assumed to be atomic. Lamport has developed a non-assertional proof technique[12] for the bakery algorithm that does not require atomic read/write operations. However, the bakery algorithm is not suitable for shared-memory multiprocessors since it induces an extremely large amount of network contention. Balbo et al. [4] reports a proof technique that combines both correctness proof and performance evaluation of Lamport's fast mutual exclusion algorithm[11]. Colored generalized stochastic Petri net model is used. However, the algorithm is not

0-8186-6555-6/94 $04.00 © 1994 IEEE

suitable for shared-memory multiprocessor systems for the same reason as in the bakery algorithm. Rigorous proofs of synchronization algorithms that make good use of some atomic instructions so that memory contention is greatly reduced are still lacking. Our proof is the first that takes advantage of an assertional characterization of the *fetch_and_store* instructions. A recent report[5] has a proof of the MCS lock algorithm that uses atomic instructions. However, the proof is based on a high level concept of abstract data type (waiting queue) that is assumed to exist in the system. Our proof is based on the basic notion of state transition model[3] that has been widely accepted. It assumes no such high level concept that needs further justification. In fact, a similar high level concept of waiting queue can be justified in our proof only after many supporting theorems are proved first.

We benefit from the well-known concepts of precondition and post-condition with respect to an action, i.e. the Hoare triple[1]. However, the following techniques in this proof are new:
1. an assertional characterization of token bit accessibility,
2. the definition of effective assignments that brings about the notion of token creation/destruction,
3. the definition of token count that derives the mutual exclusion theorem,
4. the constructing procedure of a token-list that faithfully records the arrival time sequence of lock requests so that FIFO ordering can be enforced.

Organization of the paper follows. Section 1 is the introduction. Section 2 describes the algorithm. Section 3 states and proves several useful properties that support the derivation of the final part of the proofs. Section 4 finally states and proves the the major properties that the algorithm possesses. Section 5 is the conclusion.

2 Craig-LH Lock Algorithm

Figure 1(a) is the data structure and Figure 1(b) the initial state. Define a PARTICIPATING process to be one that has requested the per process data structure from the system and has executed at least one action of the synchronization algorithm. Let N be the number of participating processes in the system. Each process is allocated a request record which contains a token bit. The system prepares an extra token bit besides the N token bits. Therefore, there would be $N+1$ token bits in total. Private variables are only accessed by its owner. Global variables are accessed

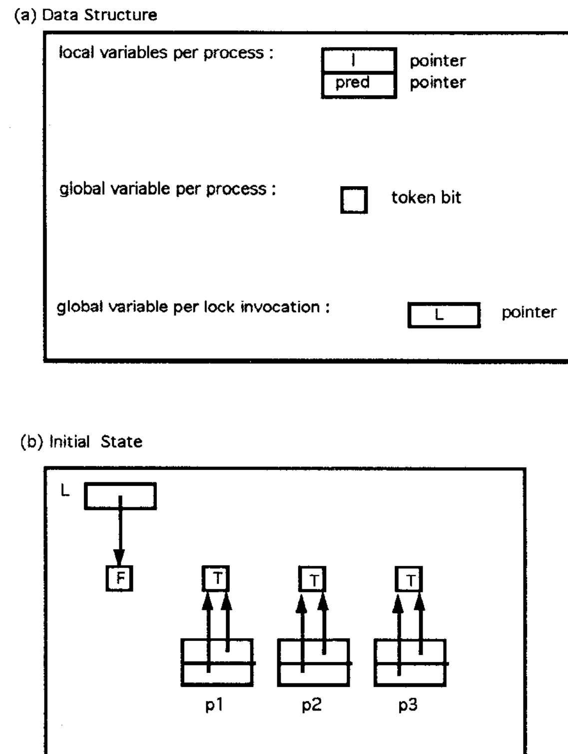

Figure 1: Data structure and the initial state for Craig-LH algorithm

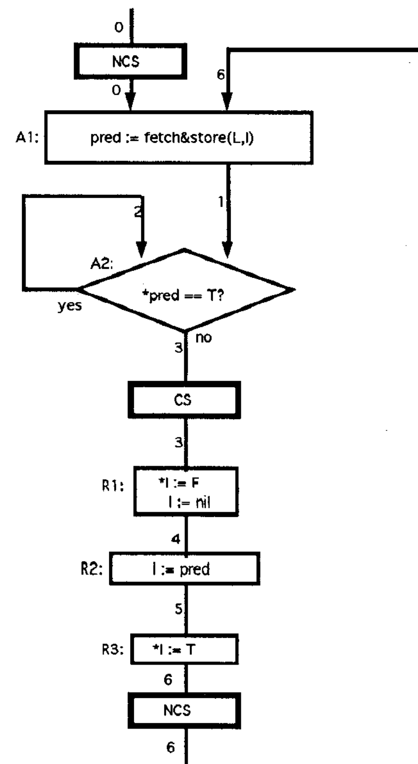

Figure 2: Craig-LH algorithm

by all participating processes. Private variables are: I and *pred*, both of which are pointers that will point to any of the token bits. Global variables are: L and the N token bits. Variable L is accessed only through the `fetch&store(L,I)` instruction each time a process makes a request to enter critical section. The token bits can only be accessed via a pointer variable, i.e., via either I or *pred*.

We will use $*pred$ to denote the token bit pointed to by *pred*. Since there is only one field in a request record, the token bit, there is no ambiguity in what $*pred$ refers to. Likewise, we will use $*I$ to denote the token bit pointed to by I.

Based on the data structure so described and the initial state shown in Figure 1(b), the following set of predicates is sufficient to characterize the initial state:

1. $\{\ L \neq I_i\ \ \forall i\ \}$
/* L points to a token bit that is different from those accessible by all processes via I-type pointers. */
2. $\{\ I_i \neq I_j\ \ \forall i \neq j\ \}$
/* A process's I pointer must access a token bit that is different from what other processes can access via I pointers. */
3. $\{\ I_i = pred_i\ \ \forall i\ \}$
/* Both the pred-type pointer and the I-type pointer belonging to a process must point to the same token bit. */
4. $\{\ *L = F\ \}$
/* L points to a token bit that has false value. */
5. $\{\ *I_i = T\ \ \forall i\ \}$
/* All token bits that are pointed to by the I-type pointers have true values. */

Several more predicates for the initial state can be derived from the set:

6. $\{\ L \neq pred_i\ \ \forall i\ \}$
/* L points to a token bit that is different from those pointed to by all pred-type pointers. */
7. $\{\ pred_i \neq pred_j\ \ \forall i \neq j\ \}$
/* Each pred-type pointer points to a distinct token bit. */
8. $\{\ *pred_i = T\ \ \forall i\ \}$
/* All token bits pointed to by the pred-type pointers have true values. */

Figure 2 is the algorithm represented in a flow-chart like graph. Several cursory observations follows. One or more edges can go to the same statement. One and only one edge comes out from an assignment statement. Two edges come out from a decision statement, one for "Yes", and the other for "No". A1 and A2 statements, together with the edges, constitute the `acquire` procedure. R1, R2 and R3 statements, together with the edges, constitute the `release` procedure. The directed edges represent the control flow of each process.

A critical section (CS) is placed on edge 3. A non-critical section (NCS) is placed on edge 0 and another one on edge 6. The CS, and each NCS as well, represents a set of actions that do not change any variables used in the algorithm. In the initial state, all processes reside at edge 0. A process begins to participate by executing A1.

Statement R1 contains two actions, `*I := F` and `I := nil`, both of which as a whole will be executed atomically. The first action is meant to be implemented, while the second is only for convenience of proof. Deleting the second action from R1 does not affect the algorithm in any way since the actions following R1 will not use the old value of I anymore. Such actions are called *auxiliary actions*.

Figure 3 shows an example of several changes of data structure for several processes. Figure 3(a) is the initial state. Each process is allocated a token bit, an I pointer and a *pred* pointer. In Figure 3(b) p1 has executed A1, where L and $pred_1$ are swapped atomically. In Figure 3(c) p2 and then p3 have executed A1 in sequence. In Figure 3(d) p1 has successfully entered critical section and has executed R1 and R2, leaving an "F" in the $*pred_2$ token bit and advancing the I_1 to where $pred_1$ points to. P1 will have the same data structure as in the initial state after it executes R3.

Definition 1 (Process lifecyle) *A process i is in* INITIAL *stage if $\{pred_i = I_i\}$ and $\{*I_i = T\}$ hold. It is in* ENQUEUING *stage if $\{pred_i \neq I_i\}$ and $\{*pred_i = T\}$ hold. It is in* ELIGIBLE *stage if $\{pred_i \neq I_i\}$ and $\{*pred_i = F\}$ hold but is still at edge 1 or 2. It is in* CS *stage if it is at edge 3. It is in* DEQUEUING *stage if $\{I_i = nil\}$ holds. It is in* RETURNING *stage if $\{pred_i = I_i\}$ and $\{*I_i = F\}$ hold.*

A process starts up in an initial stage. It enters an enqueuing stage via A1 action. It enters an eligible stage when the $*pred_i$ token bit is modified by another process. It enters a CS stage after it actually examines that token bit. It enters a dequeuing stage via R1 action. It enters a returning stage via R2 action. It re-enters an initial stage via R3 action.

3 Some useful properties and their proofs

This section introduces some properties that are instrumental for the proof of the properties of a lock synchronization algorithm.

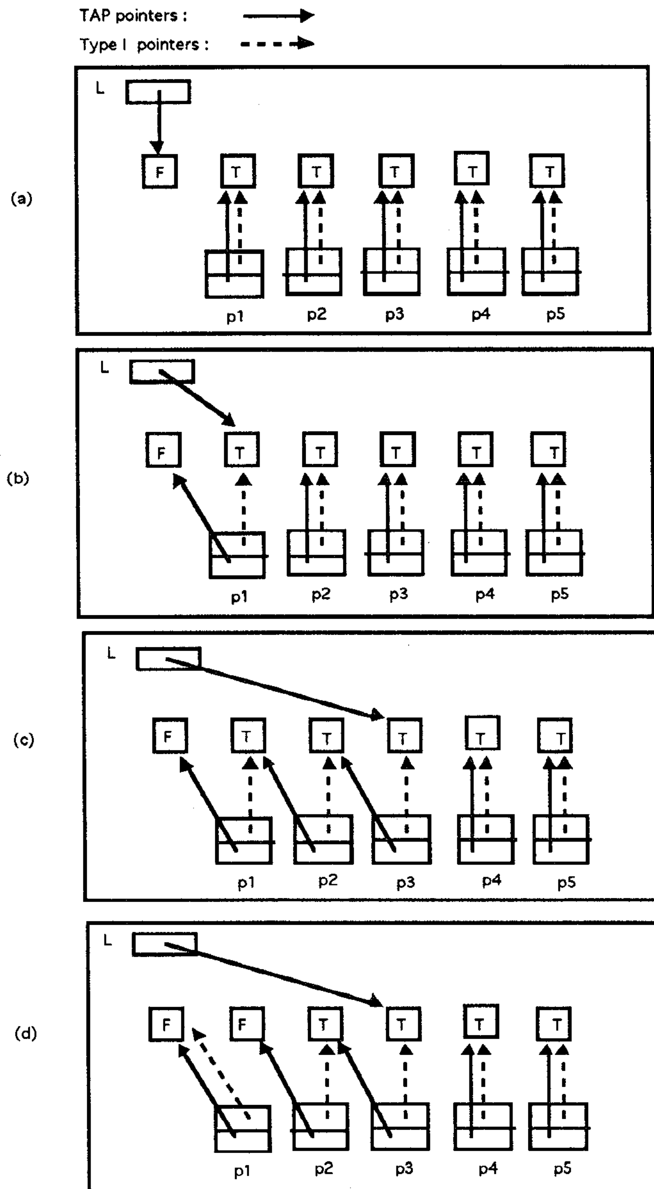

Figure 3: An example of data structure changes for Craig-LH algorithm

3.1 State Transition Model

We use the state transition model[3] in our correctness proof. A state of the synchronization system consists of the values of all variables and the edge numbers all processes are currently at. The initial state characterized by predicates 1–5 in section 2 is actually a set of such states. When no confusion arises, we do not make such distinction in our discussion.

A statement consists of a set of actions that are enclosed by a rectangle node or a diamond node in the algorithm graph. The actions in a statement are executed as a group that is indivisible. For example, the A1 statement, `pred := fetch&store(L,I)`, consists of two actions that are indivisible as a whole:
1. $pred_i := L$
2. $L := I_i$

Variable $pred_i$ can be considered as a register that is local to process i and therefore is not accessible to other processes. Executing A1, as any other statement, causes a state transition. A state transition does not necessarily induce a state change, however. For example, a process repeatedly executing A2 with "Yes" results in a row does not cause state changes.

3.2 Definition of Mutual Exclusion

We assumes that a process endlessly executes the cycle : NCS, acquire, CS and release. One of the main goals in designing a lock synchronization algorithm is to ensure *mutual exclusion*. The goal is, in the representation of the algorithm in terms of a graph, to ensure that there exists a set of edges that satisfies the following two criteria:

Indispensability A process cannot completes its acquire-release cycle without visiting some edge in the set.

Exclusiveness All edges in the set as a group cannot be visited simultaneously by two or more processes.

Since executing a CS does not cause any state transition, it causes only an arbitrarily long but finite delay. When constructing a proof, we can ignore the effect of executing a CS and concentrate on proving that the CS set is indeed indispensable and exclusive.

Likewise, we must identify a set of edges that satisfies the indispensability criterion (but not necessarily the exclusiveness criterion) and place a non-critical section (NCS) at each of the edges. The NCS set is those edges that coming to the first statement of the acquire procedure. The placement of CS and NCS in

this way conforms our assumption that a process endlessly executes the cycle : NCS, acquire, CS and release. Executing a NCS, like executing a CS, does not modify the variables used by the lock algorithm and therefore causes no state transitions. It only causes an arbitrarily long but finite delay. Therefore, we can concentrate on the state transitions caused by executing acquire or release procedures, and can ignore the actions in both CS and NCS.

From Figure 2, it is clear that placing CS at edge 3 and placing NCS at edges 0 and 6 satisfies the indispensability criterion. What remains to be proved, for mutual exclusion, is to prove that edge 3 cannot be visited by two or more processes simultaneously.

3.3 Token Bit Accessibility via pred-type pointers

The *fetch_and_store* instruction, with the global variable L as an operand, plays an important role in the algorithm. A fundamental property of the algorithm, Theorem 1, will be needed in many places in the derivation of other properties. It is derived from an assertional reasoning on the atomic action `pred := fetch&store(L,I)`. The following definition helps to simplify the description of the theorem.

Definition 2 *The token-access-pointer (TAP for short) set consists of* L *and all the pred variables.*

When N is the number of participating processes, there will be $N+1$ such pointers in the set and the same number of token bits in the system. The following theorem states a fundamental property that maps the $N+1$ pointers to the $N+1$ token bits.

Theorem 1 *For every reachable state, predicate* $\{\ pred_i \neq pred_j \quad \forall\, i \neq j\ \}$ *and predicate* $\{\ L \neq pred_i \quad \forall\, i\ \}$ *always hold. Stated equivalently, for every reachable state, each TAP variable points to a distinct token bit.*

$<$ *proof* $>$ The predicates hold in the initial state. Consider only the atomic action `pred := fetch&store(L,I)` since it is the only one that can possibly cause changes to the TAP pointers. From the flow of the algorithm, we see that *pred* is assigned with the value of L after the atomic action, and I is assigned with the value of *pred* at R2. Therefore, before each execution of the `pred := fetch&store(L,I)` action by an arbitrary process i, predicate $\{pred_i = I_i\}$ always holds. An execution of the atomic action always results in the values of L and $pred_i$ being swapped, since L is assigned with the old value of I_i, which is equal to $pred_i$, and $pred_i$ is assigned with the old value of L. Swapping a pair of the TAP pointers does not falsify either of the predicates. The predicates remain true after every execution of the atomic action. Q.E.D.

From a cursory observation of the algorithm, we know that a process never access a token bit directly via the L pointer. A process can access a token bit either via pred-type or via I-type pointers. Suppose we are dealing with a particular token bit and are asking the question: *How many processes can possibly access this token bit via pred-type pointers?* The theorem says that, at any reachable state, there is at most one process that can possibly access the token bit via a pred-type pointer.

Suppose we are asking another question: *How many processes can possibly access this token bit via I-type pointers?* The following sub-section answers such questions.

3.4 Token Bit Accessibility via I-type pointers

Theorem 2 *For all reachable states, the following predicate holds:*
$\{\ I_i \neq I_j \quad \forall\, i \neq j\ ,\ I_i \neq nil,\ I_j \neq nil\ \}$
Stated equivalently, it is impossible to reach a state in which a token bit is pointed to by two or more I-type variables.

$<$ *proof* $>$ Examines how an action can increase number of pointers to a token bit. The A1 action can be ruled out since it only swaps two pointers atomically. Swapping two pointers atomically does not change the number of pointers to the token bits. The `I := nil` action in R1 decreases number of pointers to a token bit. The only action that can increase number of pointers to a token bit is R2: `I := pred`. The action increases the number by one on the token bit $*pred$ and, at the same time, decreases the number by one on the token bit $*I$. We can conclude that for all actions, at most one pointer is added to a token bit at a time. There is no action which can add two or more pointers to a token bit.

In the initial state, the predicate of the theorem holds. Each token bit is pointed to by at most one I-type variable. We only have to prove that it is impossible to reach a state in which there exists a token bit that is pointed to by exactly two I-type variables. We don't have to consider those states in which some token bits are pointed to by more than two I-type variables since an action can add at most one pointer to

a token bit.

Denote by S_{II} the set of states in which exactly two I-type variables point to a toke bit. By way of contradiction, assume that it is possible to reach a state in S_{II} from the initial state s_{init}. Let $s_{ii} \in S_{II}$ be the first state that is reached from s_{init}. Let B be the token bit in s_{ii} that we identify to be pointed to by exactly two I-type variables. All other token bits are pointed to by at most one I-type variable. Without loss of generality, we assume that B was pointed to by exactly one I-type variable in the initial state s_{init}. There exists a behavior(history) whereby state s_{ii} is reached from s_{init}. In state s_{ii}, there exist process i and process j such that $\{I_i = I_j, I_i \neq nill, I_j \neq nil\}$ and $\{*I_i = *I_j = B\}$ hold. Consider the case where process i had its I variable pointing to B earlier than process j did. (By symmetry, we can argue for the opposite case similarly.) Then the behavior must be of the form: $s_{init}, \cdots, s_{ii}$. In order to reach s_{ii} from s_{init}, there must be at least the following events:

1. Process i does A1, resulting in $\{I_i \neq pred_i\}$. Variable I_i remains to point to B, but $pred_i$ points away. Only after $pred_i$ points away can process j have $pred_j$ to point to B .
2. Process j does A1, resulting in $\{pred_j = I_i\}$. Variable $pred_j$ points to B.
3. Process j does R2, resulting in $\{pred_j = I_j\}$. Variable I_j points to B too.

We observe that before event 1, B has a "T" in it and after event 1, B still has a "T". Before process i itself set B to "F" at R1, no other process can do that to B. Before process i itself did that, event 3 cannot occur because process j cannot pass its spinning at A2, waiting on B for an "F". By the time process i did R1, B is set to "F", but at the same time, I_i is assigned *nil* in the same atomic action. Although event 3 can occur now, B is pointed to by I_j, but not by I_i anymore. We conclude that s_{ii} cannot be reached.

For the case that B was not pointed to by any I-type variable in the initial state s_{init}, we consider a behavior(history) of the form: $s_{init}, \cdots, s_i, \cdots, s_{ii}$, where s_i is the first state in which B was pointed to by exactly one I-type variable. Similarly, we argue that it is impossible to reach s_{ii}. Q.E.D.

3.5 Assertional Characterization of Token Bit Accessibility

We have discussed in section 3.3 and section 3.4 token bit accessibility via the pred-type pointers, and token bit accessibility via the I-type pointers, respectively. The results are the two theorems that are summarised in several predicates. Such predicates unambigously characterize the behavior of both type of pointers.

We now consider how a token bit can be accessed by processes via both types of pointers, be it of pred-type or of I-type. We will express the result in an assertional manner in the following theorem.

Theorem 3 *Let N be the number of participating processes. For any reachable state, each of the $N+1$ token bits can be pointed to by at most two pointers. At least one of the token bits is pointed to by exactly one pointer.*

< proof > For N participating processes, there are $2N+1$ pointers among which N pointers are of pred-type, another N pointers are of I-type, and the last one is the L pointer. Recall the both the *pred* pointers and the L pointer are defined as of TAP-type. From Theorem 1, we know each token bit is pointed to by each of the TAP pointers, i.e., the following two predicates are true: $\{\ pred_i \neq pred_j \ \ \forall\ i \neq j\ \}$ and $\{\ L \neq pred_i \ \ \forall\ i\ \}$. We must also consider the I-type pointers that can also point to the token bits. From Theorem 2, we know a token bit cannot be pointed to by two or more I-type pointers:
$\{\ I_i \neq I_j \ \ \forall\ i \neq j\ ,\ I_i \neq nil,\ I_j \neq nil\ \}$
Since no other pointers can possibly point to the token bits, we conclude that a token bit can be pointed to by at most two pointers:
(1) one of TAP-type, which is necessary, and
(2) one of I-type, which is not necessary.
The reason that a token bit is not necessarily pointed to by an I-type pointer is that there are only N such pointers in total while there are $N+1$ token bits in total. There is always one token bit that is left out. Since an I-type pointer can sometimes become nil by the R1 statement, there might be two or more token bits that are pointed to by exactly one pointer, which must be of TAP type. When no process is at edge 4, there is exactly one token bit that is pointed to by exactly one pointer. Q.E.D.

3.6 Constructing Invariants Table

Table 1 is a collection of predicates that are to be proved correct at certain edges for a process, regardless of the actions of other processes. We will examine each predicate for each edge and then mark with a "*"symbol if the predicate is proved correct for that edge. When a predicate has been marked for some edges, we call it a **point invariant** at those edges.

Table 1: Invariants for the Craig-LH algorithm

		Edge Number						
	Predicates	0	1	2	3	4	5	6
1	$I = pred$	*					*	*
2	$I \neq pred$		*	*	*	*		
3	$L = I$							
4	$L \neq I$	*					*	*
5	$L = pred$							
6	$L \neq pred$	*	*	*	*	*	*	*
7	$pred_i \neq pred_j \;\; \forall\, i \neq j$	*	*	*	*	*	*	*
8	$*I = T$	*	*	*	*			*
9	$*I = F$						*	
10	$*pred = T$	*						*
11	$*pred = F$				*	*	*	
12	$L \neq pred_i \;\; \forall\, i$	*	*	*	*	*	*	*

When a predicate has been marked for all edges, we call it a **global invariant**. A global invariant is a predicate that is true for all reachable states. If a predicate in the table is not marked for a certain edge, it does not mean that the predicate is always false when a process resides at that edge. Rather, it means that the value of the predicate is not determined. For example, predicates 3 and 5 are not marked for each of the edges. The values of both predicates cannot be determined for each of the edges.

The rest of the subsection shows the examination process of the predicates. The order of the predicates to be examined does not follow the predicate number in the talbe. Rather, it follows the sequence of deriving new invariants from previously established theorems or invariants. Note that the derivation process does not contain **circularity**. An example of circularity in a derivation is when an unproved claim C_1 is used to prove claim C_2, and later, claim C_2 is used to prove claim C_1.

Predicate 1: $I = pred$
Consider a process at edge 0. Initially, $\{I_i = pred_i \; \forall\, i\}$ holds and other processes' actions cannot change the value of either I or *pred* since both variables are local to the process at edge 0. Therefore, predicate 1 is a point invariant at edge 0. We mark **(1,0)**.

Consider a process at edge 5. The local variable I has just been assigned the value of the local variable *pred*. Clearly, predicate 1 must be a point invariant at edge 5. Similarly, it must be a point invariant at edge 6. We mark **(1,5) and (1,6)**.

Predicate 6: $L \neq pred$
By Theorem 1, we can mark **all edges** for this predicate.

Predicate 12: $L \neq pred_i \;\; \forall\, i$
By Theorem 1, we can mark **all edges** for this predicate.

Predicate 7: $pred_i \neq pred_j \;\; \forall\, i \neq j$
By Theorem 1, we can mark **all edges** for this predicate.

Predicate 4: $L \neq I$
The predicate holds for every process in the initial state. Consider a process i remaining at edge 0 while other processes may proceed to execute A1 and other actions. Variable L can, however, be changed to be equal to some I_j each time A1 is executed by some process j. But predicate $\{I_i \neq I_j \;\; \forall\, j \neq i\}$ holds for the initial state and the subsequent states in which process i remains at edge 0. No action of A1 by any other process, nor any other action besides A1, can lead to $\{L = I_i\}$. Therefore, $\{L \neq I\}$ is a point invariant at edge 0. We mark **(4,0)**.

Consider a process i at edge 4, having $\{L \neq pred\}$ by entry (6,4). The action R2 (`I := pred`) clearly results in a point invariant: $\{I = pred\}$ at edge 5. Since $\{L \neq pred\}$ also holds at edge 5 and 6, **we mark (4,5) and (4,6)**.

Predicate 2: $I \neq pred$
Consider a process at edge 1. Before the process does A1, it could have stayed at either 0 or 6. From (4,0) and (4, 6), we know $\{L \neq I\}$ holds prior to action A1. The predicate $\{I \neq pred\}$ must hold at edge 1 since *pred* has been assigned the old value of L while I remains unchanged. Both I and *pred* remains unchanged for the process when it flows from edge 1 to 4, therefore the predicate $\{I \neq pred\}$ must hold at edge 2, 3 and 4. We mark **(2,1), (2,2), (2,3) and (2,4)**.

Predicate 8: $*I = T$
By Theorem 2, all non-nil I-type pointers have distinct values. Consider a process i at edge 0. The predicate holds at edge 0 since the token bit $*I$ is not accessible to any other process as long as process i stays at edge 0. We mark **(8,0)**. Consider a process i at edge 6. Token bit $*I_i$ is pointed to by I_i and $pred_i$, hence no other pointers can also point to it. Process i had just finished R3 (`*I := T`) and the token bit $*I$ is not accessible to any other process as long as process i stays at edge 6. Therefore, we mark **(8,6)**. Consider a process i at edge 1, having executed action A1. Prior to A1, the predicate must hold for process i(since (8,0) and (8,6) are marked). After A1, the predicate still holds since no other process can change the token bit from "T" to "F". (A token can only be assigned "F" via `*I := F` action in R1. But Theorem 2 tells us that other I-type variables must point to some other

token bits.) Therefore, we can mark **(8,1)**. Similarly, we can mark **(8,2) and (8,3)**.

Predicate 10: $*pred = T$

By (8,0) and (1,0), we can mark **(10,0)**. Similarly, by (8,6) and (1,6), we can mark **(10,6)**.

Predicate 11: $*pred = F$

Consider a process i at edge 3. It had a "No" result of the test at A2. At least at that particular moment when the testing is carried out, the predicate is true. The question is whether the predicate remains true as long as process i stays at edge 3. No other process can access the token bit $*pred_i$ via $pred$ variable since $\{pred_i \neq pred_j \quad \forall\ i \neq j\}$ always holds. The only process that can possibly access the token bit is the one, process j, that had executed A1 earlier than process i did and therefore would have $\{I_j = pred_i\}$ until it re-define I_j at R2. Its action at R1 had resulted in process i being able to come to edge 3. Its action at R3 would have done no harm since I_j would have been re-defined to point to elsewhere. No other process can access $*pred_i$ as long as process i stays at edge 3. Therefore, we can mark **(11,3)**. Similarly, we can mark **(11,4) and (11,5)**.

Predicate 9: $*I = F$

Consider a process i at edge 4. We cannot mark (9,4) even though the process had executed (`*I := F`) at R1 because that action may have let go some process j spinning at A2. Process j may then execute R1, R2 and R3, thereby change the value of I. Consider a process i at edge 5. By entries (11,5) and (1,5), we can mark **(9,5)**. Edge 5 is the only one at which the predicate holds.

4 Correctness Proof of the Algorithm

We now come to the final part of the proof that is eminently interesting to us.

Definition 3 (Effective Assignments) *An assignment action is called* EFFECTIVE *if the new value after the action is different from the old value before the action.*

Theorem 4 *All assignment actions in the algorithm are effective.*

< *proof* > The assignment actions are:

1. the assignment `pred := L` in A1,
2. the assignment `L := I` in A1,
3. `*I := F`,
4. `I := pred`
5. `*I := T`

The assignment `I := nil` in R1 is not included here since it is an auxiliary action which is added only for the convenience of proof.

For each action listed above, we show that the assignment is indeed effective as follows.

1. By entries (1,0) and (1,6), $\{pred = I\}$ holds before the action. By entry (2,1), $\{pred \neq I\}$ holds after the action. Since I has not changed, it must be $pred$ that has changed.
2. Consider a process i taking the assignment action. By entries (4,0) and (4,6), $\{L \neq I_i\}$ holds before the action. By the semantics of `fetch&store(L,I)`, L has been assigned with the value of I_i. Therefore, L has been changed by the assignment.
3. By entry (8,3), $\{*I = T\}$ holds before the action.
4. By entry (2,4), $\{I \neq pred\}$ holds before the action.
5. By entry (9,5), $\{*I = F\}$ holds before the action.

Q.E.D.

Definition 4 *Let* FTOKENCOUNT *of a state be the number of distinct token bits in the system having false values in that state.*

Theorem 5 *For any reachable state, Ftokencount equals the number of processes at edges 4 or 5 plus one.*

< *proof* > The statement of the theorem holds for the initial state. From Theorem 4, all assignment actions are effective. For all actions of R1, *Ftokencount* is increased by one and the number of processes at edge 4 or 5 is also increased by one. The statement of the theorem still holds. For all actions of R3, *Ftokencount* is decreased by one and the number of processes at edge 4 or 5 is also decreased by one. The statement of the theorem still holds. Q.E.D.

Theorem 6 (Mutual Exclusion) *It is impossible to reach a state in which two or more processes are at edge 3.*

< *proof* > By way of contradiction, assume that we reach a state S in which two or more processes are at edge 3. From entries (8,3) and (11,3), we know $\{*I = T\}$ and $\{*pred = F\}$ hold for each process at edge 3. From Theorem 1, we know $\{pred_i \neq pred_j \quad \forall\ i \neq j\}$ holds. We can derive $\{Ftokencount \geq 2\}$ since there are two or more processes at edge 3. By Theorem 5, there must exist at least one process at edges 4 or 5. But $\{*pred = F\}$ holds at edges 4 or 5, observing entries (11,4) and (11,5). The process at those edges must have $\{*pred = F\}$, and therefore contribute one more count into *Ftokencount*. That gives

$\{Ftokencount \geq 3\}$. By Theorem 5, there must exist at least two processes at edges 4 or 5. The argument can proceed in the same way and gives $\{Ftokencount \geq 4\}$, $\{Ftokencount \geq 5\}$, and so on. Therefore, we can write the following predicate:

$$Ftokencount \geq N$$

However, $\{*I = T\}$ holds for each process at edge 3. At least two token bits have true values since at least two processes are at edge 3. Therefore we can write the following predicate:

$$Ftokencount \leq N - 1$$

A contradiction is derived. Q.E.D.

Consider the deadlock possibilities. Following the strategy of Owicki and Gries[13], deadlock freedom can be proved by first enumerating all potential deadlock states. Then we must show that it is impossible to reach any of the deadlock states. From the algorithm graph, it is clear that the only possible group of deadlock states is for all participating processes to be kept spinning indefinitely at A2. If such possibility is ruled out, we are assured that no deadlock can occur.

Theorem 7 (Deadlock Freedom) *It is impossible to reach a state in which all participating processes are kept spinning indefinitely at A2.*

< *proof* > Suppose it is possible to reach such a state S. All participating processes are kept spinning indefinitely at A2, hence are at edges 1 or 2 indefinitely. By Theorem 5, $\{Ftokencount = 1\}$ holds since no process is at edges 4 or 5. By entries (8,1) and (8,2), $\{*I_i = T \; \forall \; i\}$ holds in S. Since all processes are at edges 1 or 2 indefinitely, $\{*pred_i = T \quad \forall \; i\}$ holds in S. For all pointer variables in the system, only L is left to point to the false token bit. Since each token bits must be pointed to by some TAP pointer by Theorem 1, we are assured that state S must satisfy the following predicate:

$$*L = F$$

Since all participating processes are kept spinning at A2, there exists a process i that is the last one in executing `fetch&store(L,I)` and no other process can execute that action afterwards. Let that event be E. The predicate $\{L = I_i\}$ holds after E and will continue to hold. The predicate $\{*I_i = T\}$ holds prior to E and will continue to hold since all processes are to be kept spinning at A2 indefinitely. Combining the two predicates, we conclude that state S must satisfy the following predicate:

$$*L = T$$

A contradiction is derived. Q.E.D.

Consider the ordering discipline the algorithm enforces. We start from a definition of arrival time and then show that the algorithm enforces a FIFO discipline following the arrival ordering of the acquiring processes. Note that a process must arrive each time it makes a new request to enter critical section.

Definition 5 *For each lifecycle of a process, the* ARRIVAL TIME *of the process is when the first action of Acquire procedure is executed.*

From the algorithm, it is clear that the arrival time of a process is the when `fetch&store(L,I)` is executed.

For every reachable state, we can always define a data structure called **T-list** that faithfully records the sequence of requests made by the acquiring processes. The synchronization algorithm does not maintain the T-list explicitly. It is defined by a constructing procedure solely for the purpose of proving the FIFO ordering property.

For each reachable state, we can construct the T-list by the procedure.

Constructing procedure of the T-list:

1. Initially T-list consists of the token bit $*L$ alone. Visit that token bit.
2. If the currently visited token bit is pointed to by exactly one pointer, then stop.
3. The other pointer to the token bit must be of I-type. Let the owner process of the pointer be P_i. Append the token bit $*pred_i$ to T-list and visit that token bit.
4. Goto 2.

The following lemma, which can be proved, guarantees that the definition of T-list is valid.

Lemma 1 *The constructing procedure of the T-list always terminates.*

Theorem 8 (FIFO ordering) *The acquiring processes will enter critical section in the order of their arrival time.*

< *proof* > The basis of the proof is the T-list which faithfully records the order of the arrival sequence of the acquiring processes. Full proof is omitted. Q.E.D.

It has been shown by Lamport[10] that a mutual exclusion algorithm is starvation free if it is deadlock free and satisfies FIFO ordering. *Starvation freedom* is assured directly from the Deadlock Freedom Theorem and the FIFO ordering Theorem.

5 Conclusion

Correctness proof of multiprocessing algorithms for shared-memory systems encounters a basic difficulty that any memory access of a process can interfere with that of other processes. One major effort in the proof is to reduce the scope of the interference. The assertional characterization of the token bit accessibility reduces the scope of interference from N-processes to 2-processes. More specifically, the Theorem 1 allows us to consider only the owner of the token bit (via the *I* pointer) and the unique successor process (via the *pred* pointer) as the possible processes having access to a particular token bit. Without the theorem, we would have to consider all participating processes as the possible processes.

Token is often used in synchronization algorithms for message-passing systems. In shared-memory systems, however, the notion of token needs a formal definition. A token is nothing but a false value in a token bit, and therefore can be inadvertently destroyed or created by a memory access action of any process. Our definition of effective assignments formally defines the notion of token creation/destruction by certain actions: a R1 action creates a token, and a R3 action destroys a token. Once we have the notion of token creation and destruction, we are able to derive Theorem 5 that supports the Mutual Exclusion Theorem.

The proofs for Deadlock Freedom and FIFO Ordering Theorems also use the supporting properties that we established in section 3. The reasonings are precise and concise in those proofs because we were able to state the supporting properties in predicates. The rigorous approach that we take in our proofs also helps to prevent circularity in the proof process. Without such rigor, one can be inclined to use high level concepts about the program behavior in the proof process. Worse yet, one can even be tempted to justify the high level concepts using the final theorems. That would be a case of circularity that can easily be left undetected.

References

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10): 576–580 and 583, October 1969.

[2] T. L. Huang and J. H. Lin. An assertional proof of a scalable lock synchronization algorithm for multiprocessors with fetch_and_store atomic instructions. Technical Report CSIE-94-1007, National Chiao-Tung University, Dept. of Computer Science and Information Engineering, March 1994.

[3] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3): 225–262, September 1993.

[4] G. Balbo, G. Chiola, S. C. Bruell, and P-Z Chen. An example of modeling and evaluation of a concurrent program using colored stochastic Petri Nets: Lamport's fast mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 3(2): 221–240, March 1992.

[5] T. Johnson and K. Harathi. A simple correctness proof of the MCS contention-free lock. TR92-040, Dept. of Computer and Information Sciences, Univ. of Florida, 1992.

[6] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-Memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1): 21–65, Feb. 1991.

[7] T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195

[8] Intel Corporation. i486 Microprocessor Programmer's Reference Manual Osborne McGraw-Hill, 1990, Chap. 26.

[9] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2): 125–143, Sep. 1977.

[10] Leslie Lamport. The mutual exclusion problem – Part I and II. *Journal of the ACM*, 33(2): 313–348, April 1986.

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

[12] Leslie Lamport. win and sin : Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3): 396–428, July 1990.

[13] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5): 279–285, May 1976.