行政院國家科學委員會專題研究計畫　成果報告

近端自轉互斥演算法中不可分割指令之階層架構

計畫類別： 個別型計畫
計畫編號： NSC92-2213-E-009-064-
執行期間： 92 年 08 月 01 日至 93 年 07 月 31 日
執行單位： 國立交通大學資訊工程學系

計畫主持人： 黃廷祿

計畫參與人員： 陳勝雄

報告類型： 精簡報告

處理方式： 本計畫可公開查詢

中　華　民　國 93 年 10 月 29 日

行政院國家科學委員會專題研究計畫成果報告

近端自轉互斥演算法中不可分割指令之階層架構

Hierarchical Structure of Atomic Instructions Used in Local-spin Mutual Exclusion Algorithms

計畫編號：NSC 92-2213-E-009-064

執行期限：92 年 8 月 1 日至 93 年 7 月 31 日

主持人：黃廷祿　國立交通大學資訊工程學系

E-mail: tlhuang@csie.nctu.edu.tw

計畫參與人員：陳勝雄　國立交通大學資訊工程學系

E-mail: chenss@csie.nctu.edu.tw

一、中文摘要

針對像嵌入式即時系統這樣具有時間及資源限制的應用，所設計的互斥問題必須公平且降低所需的記憶體空間。因此，我們提出一個公平(bounded-bypass)且僅使用二個變數的互斥演算法。一個變數為 *read-write* register，另一個為 *fetch&store* register。若以達到相同公平性而言，我們也證明了使用 historyless 物件來設計互斥演算法則至少需要二個物件，亦即本計畫所提出的演算法用了最少的記憶體。所謂 historyless 物件(包含 *read-write* register 及 *fetch&store* register) 即其值決定在最後一個寫入的指令，與原本的值無關。此外，即使所用的物件可包含無限多的數值，這個至少需要二個物件的下限依然成立。

關鍵詞：互斥問題、不可分割指令、分享記憶體系統、公平性、空間複雜杜、下限

Abstract

For a shared memory system with time and resource constraints such as an embedded real-time system, a mutual exclusion should be fair and space-efficient. We present a bounded-bypass algorithm using only constant two shared variables: one *read-write* register and one *fetch&store* register. To achieve the same level of fairness, we show that, using historyless objects, two shared object instances are necessary, and therefore our algorithm is space optimal. An object is described as historyless if and only if its value depends only on the last nontrivial operation applied to it. This lower bound holds even if the objects have infinite size.

Keywords: mutual exclusion, atomic instructions, shared-memory systems, fairness, space complexity, lower bound

二、緣由與目的

The mutual exclusion problem [4] is fundamental in asynchronous shared memory systems for managing accesses to a single indivisible resource. In this problem, a process accesses the resource within a distinct part of code called its *critical region*. Before and after executing the critical region, a process executes trying and exit regions, two other parts of code, respectively. The problem is to design the trying and exit regions guaranteeing the following requirements.

- **Mutual Exclusion**: At most one process at a time is permitted to enter its critical region.
- **Progress**: If some process is in the trying region and no one is in the critical region, then at some later point some process enters the critical region. In addition, a process in the exit region will eventually enter the rest of code, called the remainder region.

The burgeoning applications for embedded real-time systems such as automotive control systems, cellular phones and home electronics have created a demand for algorithms in these systems [13]. An algorithm suitable for these environments must meet two constraints: time constraint and resource constraint. Thus, a mutual exclusion algorithm should be fair and space-efficient.

A mutual exclusion algorithm may not guarantee the critical region is granted "fairly" to different processes; that is, starvation may occur. A fair mutual exclusion algorithm means that it has the ability to control the order of granting requests in a fair manner such that no process will starve. In a system with time constraint, a process has a deadline in executing a particular job. The goal of a fair mutual exclusion is to reduce the worst-case time, preventing a process overshoots its deadline.

On the other hand, the major goal of a space-efficient mutual exclusion algorithm is to reduce the memory consumption. It is crucial for systems with resource constraint. For instance, embedded systems often have small memory (about 32--64 kBytes [14]) since making low production costs is one of the primary concerns in their design. Thus, an algorithm for such systems must be space-efficient.

Recent work on the mutual exclusion problem has focused on designing local-spin algorithms which minimize the required number of remote memory references [11, 3, 7, 9, 8]. It is because remote memory references cause processor-to-memory traffic which may result in memory bottleneck in general distributed shared memory systems. Since each process must have at least one shared variable that is locally-accessible, at least *n* shared variables are needed, where *n* is the number of processes. However, the number *n* may be very large and therefore these algorithms are not suitable for space-limited systems, although some of these algorithms are fair.

The primary contribution of this project is a fair and space-efficient mutual exclusion algorithm. Our algorithm has the following advantages.

- **Fair**: The algorithm is 2-bounded-bypass.
- **Space-efficient**: Only constant two shared variables are needed.

We say that a mutual exclusion algorithm satisfies *b*-bounded bypass if a requesting process cannot be bypassed by any certain process in accessing the resource for more than *b* times. An algorithm is bounded-bypass if it is *b*-bounded bypass for some *b*. 2-bounded bypass is very close to the first-in-first-out (FIFO) order, the most stringent fairness requirement, which is a kind of 1-bounded bypass. (More precisely, a FIFO algorithm is also 1-bounded bypass, but the reverse is not true.) For most applications, we believe that our 2-bounded-bypass algorithm is good enough.

To implement our algorithm, we use primitive *fetch&store* in addition to *read* and *write* primitives. Burn and Lynch [1] has shown *n* shared variables are necessary to solve the *n*-process mutual exclusion problem if only *read* and *write* primitives are available. Thus, we need certain more powerful primitive to reduce the space complexity. Fortunately, modern microprocessors often provide some read-modify-write (RMW) primitives such as *test&set*, *fetch&store*, *compare&swap*, etc. In one instantaneous step, a RMW primitive can read a shared variable and write back a new value according to the current value and the submitted function. We use *fetch&store* to implement our algorithm since it is the most commonly supported instruction in modern microprocessors such as a series of processors of Intel and AMD, Motorola 88000, and SPARC [12], making the algorithm more portable.

Notice that, in the literature, there are several algorithms using only one shared variable and guaranteeing certain level of fairness. For instance, Fischer et al. [6] devised a FIFO algorithm. Burns et al. [2]

devised a bounded-bypass algorithm and a starvation-free algorithm[1]. Unfortunately, all of these algorithms used hypothetical RMW primitives which have never been implemented in any system shipping today. In contrast, our algorithm uses no hypothetical RMW primitive and requires only one more shared variable than these algorithms.

Our algorithm is inspired by the circular list-based mutual exclusion algorithm proposed by Fu and Tzeng [7, 9]. Similar to their method, our algorithm also let waiting processes form a list. But the way to convey permission in a list and between two lists is very different from theirs. In fact, the problem they tackle is to reduce the number of remote shared memory accesses, but we desire to reduce the space complexity and meanwhile, guarantee certain level of fairness.

In addition, it is impossible to obtain bounded-bypass algorithm with less than two shared variables, using *fetch&store* as well as *read* and *write*; that is, our algorithm is space optimal. We prove it by showing a more general result: using only *historyless* objects, two shared object instances are required to implement a bounded-bypass algorithm. The definition of a historyless object is given by Fich et al [5]. Informally, an object is historyless if and only if its value depends only on the last nontrivial operation applied to it. A nontrivial operation is one that will write a value into the object. For example, *read-write* registers, *fetch&store* registers and *test&set* registers are historyless. This lower bound holds even if the objects have infinite size.

Our lower bound proof technique is related to the method introduced by Burn and Lynch to prove the lower bound of *n* on the number of *read-write* registers required to solve the *n*-process mutual exclusion problem [1]. The difference is that our lower bound applies to all historyless objects rather than only *read-write* registers. Moreover, our lower bound is for bounded-bypass mutual exclusion algorithms, whereas Burn and Lynch consider the general mutual exclusion problem.

[1] Indeed, their work aimed at theoretical discussion between data requirements and different fairness conditions.

三、結果與討論

The Algorithm

```
Shared variables:
   L ∈ {nil,1,…,n}, initially nil
   P ∈ {nil,1,…,n}, initially nil

Process i : (1 ≤ i ≤ n)

Private variables:
next ∈ {nil,1,…,n}
tail ∈ {nil,1,…,n}

      while true do
R:       Remainder region
T1:      next := fetch&store(L, i);
T2:      if next = nil then
T3:         await P = nil;
T4:         P := i;
T5:      else
T6:         await P = i;
T7:      fi
C:       Critical region
E1:      if next = nil then
E2:         tail := fetch&store(L, nil);
E3:         if tail ≠ i then
E4:            P := tail
E5:            await P = i;
E6:         fi
E7:         P := nil;
E8:      else
E9:         P := next;
E10:     fi
      od
```

Figure 1. The algorithm.

We begin by presenting the main idea of the algorithm in an informal pseudocode style as shown in Figure 1. Exactly two shared variables are used in the algorithm: variable *L* is used to arrange processes' requests to critical regions; while variable *P* to indicate which process has permission to enter its critical region. Initially, variables *L* and *P* are set to *nil*, respectively.

Through variable *L* and *fetch&store*

primitive, the order to enter the critical region is organized as a circular waiting list in which the first element has the identity of the last one, and each other element has the identity of its predecessor. A circular list is formed as follows. Each process *i* makes a request by a *fetch&store* onto *L* (T1), announcing its process identity and obtaining the predecessor's identity if has one. Any process which acquires a *nil* from *L* (i.e., *next* = *nil*) becomes the header; otherwise, it becomes a list member. (A header is also dubbed a *controller* and has extra duty at its exit region.) A waiting list is closed after the controller leaves its critical region and resets *L* as *nil* (E2). The controller stores the identity of the last element in the list into its private variable *tail*. This closed waiting list contains all processes making a request between the controller obtaining *nil* from the *L* (T1) and resetting *L* as *nil* (E2). Note that, only after the current controller closes its waiting list such that *L* will become *nil* again, a new list might start to form.

The value of shared variable *P* indicates which process has permission to enter its critical region now. After making a request, a controller repeatedly tests the value of *P* until *P* is equal to *nil* (T3), a specific permission for a controller. The controller takes the permission by assigning *P* as its identity (T4). (This action prevents another new controller to enter its critical region.) In contrast, a list member *i*---that is, if $next_i \neq nil$---checks the value of *P* until *P* = *i* (T6) indicating *i* gains the permission to enter its critical region. Since *P* is *nil* initially, the first controller at all will gain the permission to enter its critical region.

After a process leaves its critical region, it should convey the permission to certain waiting process if has one. As a list member, the process simply transfers the permission to its predecessor by setting *P* as *next* (E9) and then enters its remainder region. As a controller, after closing the waiting list, if the list contains any process other than the controller, it passes the permission to the last element in the list by setting *P* as *tail* (E4). The permission will be passed from the last element back to the controller, i.e., in the reverse order of processes making a request. The controller is blocked until the permission passes back to itself (E5).

Although resetting *L* as *nil* might introduce a new controller, this new controller and subsequent requesting processes will not obtain the permission and this new waiting list will not be closed unless all processes in the previous circular waiting list have finished their critical regions. (Hence, there are at most 2 waiting lists simultaneously, and at most one of these two lists contains the permission.) This contributes to the bounded bypass property of our algorithm. The new controller will get the permission after the permission passes back to the previous controller causing the previous controller to reset *P* as *nil* (E7).

An execution of the algorithm.

An example is given in Figure 2, showing that how to arrange the order to enter the critical region for requesting processes. Initially, both of shared variables *L* and *P* are equal to *nil* (see Figure 2(a)). Process 1 first makes a request by executing T1. Since $next_1$ = *nil* and *P* = *nil*, process 1 enters its critical region after assigning *P* as 1 (see Figure 2(b)). As process 1 is in *C*, processes 2 and 3 execute T1 in turn. Because neither process 2 nor process 3 gets *nil* from *L*, processes 2 and 3 are waiting at T6. The waiting list is shown at Figure 2(c). Then, process 1 leaves its critical region. Since process 1 is a controller ($next_1$ = *nil*), process 1 closes the waiting list by executing E2 which returns the tail of the list and resets *L* as *nil* (see Figure 2(d)). Process 1 passes the permission to the tail of the list (see Figure 2(e)). The permission will be passed one after one in the waiting list. Process 1 is blocked until the permission backs to itself. During the period after process 1 closes the waiting list, subsequent requesting processes will be blocked. For example, after process 4 gets *nil* from *L* by executing T2, it is waiting at T3. Figure 2(f) shows that the permission backs to process 1, that is, all requesting processes in the circular waiting list have finished their critical regions. Process 1 resets *P* as *nil* to let process 4 enter *C* (see Figure 2(g)). After observing *P* is equal to *nil*, process 4 will enter its critical region.

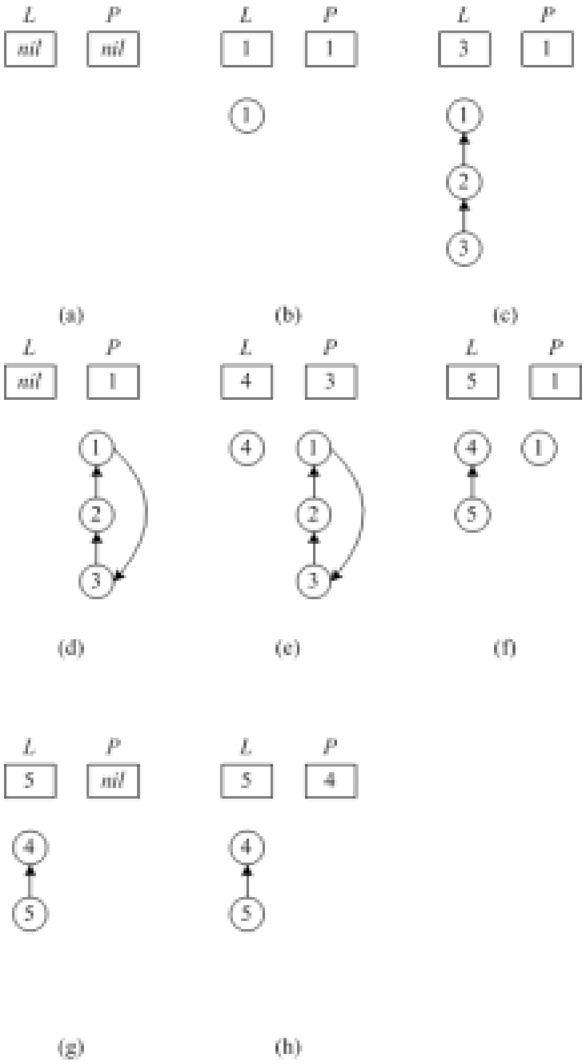

Figure 2. An execution of the algorithm.

Impossibility Result

In this section, we show that there is no mutual exclusion algorithm guaranteeing bounded bypass with fewer than 2 historyless object instances. We follow the proving strategies proposed by Burns and Lynch [1]. Their model contains only *read-write* register. We extend the model to include historyless objects and prove our result. The following definitions will be used in the proof. The first two are directly borrowed from [10].

Definition 1. *System states s and s' are indistinguishable to process i, written as s[i]s', if the state of process i and the values of all object instances are the same in s and s'.*

Definition 2. *A system state s is idle if all processes are in their remainder regions in s.*
Following from the progress condition, a process starting from an idle state and involving its steps only will reach the critical region. Furthermore, a process starting from a system state that is indistinguishable to an idle state for this process and involving its steps only will also reach its critical region, since the state of this process and the values of all object instances are the same in these two system states.

The last definition generalized the one defined by Burns and Lynch [1]. According to their original definition, a process *covers read-write* register *x* if a *write* operation of the process is enabled to write *x*. An enabled *write* will overwrite the variable it involves. Inspecting a historyless object instance *x*, once a non-trivial operation is enabled, it will write a value into the variable and overwrite other processes might have written to *x*. Thus, we generalize the concept of "covering" to all historyless objects.

Definition 3. *Process i covers a historyless object instance x in system state s provided that in state s, a non-trivial operation of x is enabled by process i.*
Once process *i* covers *x*, *i* will write a value into *x* in its next step.

The main idea of the lower bound is that when a process covers a historyless object instance *x*, it will overwrite other processes might have wrote to *x*. If a request of some process is overwritten, we may let another process enter its critical region so many times that violate the bounded bypass condition.

Before proving the lower bound, a basic lemma is needed, showing that a process in its exit region must write something into an object instance.

Lemma 1. *Suppose that A is a mutual exclusion algorithm for* $n \geq 2$ *processes. Suppose that s is a reachable system state in which process i is in the critical region. If process i reaches R in an execution fragment starting from s that involves steps of i only, then it must write some object instance along the way.*

Proof. Let α_1 be any finite execution fragment that starts from *s* (*i* in *C*), involves steps of *i* only, and ends with process *i* in *R*. By way of contradiction, suppose that α_1 does not include any write to an object instance. Let *s′* be the state at the end of α_1 . $s\ [\ j\ ]\ s'$, for all $j \neq i$, since the values of all object instances remain unchanged.

According to the progress condition, there is an execution fragment starting from *s′* and not including any steps of process *i*, in which some other process reaches *C*. Because $s\ [\ j\ ]\ s'$, for all $j \neq i$, there is also an such execution fragment starting from *s*.

An execution α violating the mutual exclusion is easily constructed as follows. Execution α begins with a finite execution fragment leading to reachable state *s*, then let another process go to *C* without any steps of *i*. Since there are two processes in *C* at the end of α, this violates the mutual exclusion condition. □

Theorem 1. *If algorithm A solves the mutual exclusion problem for* $n > 2$ *processes and guarantees bounded bypass, using only historyless objects, then A must use at least 2 object instances.*

Proof. Suppose for the sake of contradiction that there is such an algorithm, *A*, using only one historyless object instance, say *x*, and guaranteeing *b*-bounded bypass. We construct an execution of *A* that violates bounded bypass.

There is an execution involving process 1 only, starting from an initial state *s* which is idle, that causes process 1 to enter *C* once and back to an idle state *s′*. Lemma 1 implies that when process 1 is in the exit region, it must write *x*.

First, we construct α_1 by running process 1 alone from *s* until it **last** covers *x*. Then we extend α_1 to α_2 by causing process 2 to perform a locally controlled step in the try region and continuing to run process 1 one step, which writes a value into *x*. Let the final states of α_1 and α_2 be s_1 and s_2, respectively. In states *s′* and s_2, *x* has the same value and therefore $s'\ [\ i\ ]\ s_2$, for all $i \neq 1$ and 2. Only process 1 might know that process 2 has preformed a locally controlled step by the return value when process 1 overwrote *x*.

Since $s'\ [\ i\ ]\ s_2$, for all $i \neq 1$ and 2, and *s′* is an idle state, we run process 3 alone, starting from s_2, and let process 3 to enter the critical region *b*+1 times, which causes process 3 to bypass process 2 more than *b* times. This is the needed contradiction. □

四、計畫成果自評

執行計畫過程中，我們針對計畫的目標稍做修改，改以空間複雜度來作為研究不同 atomic 指令用來設計互斥問題的極限。也就是說，研究運用不同指令設計互斥問題所需的記憶體空間。我們希望可以設計出既公平又節省記憶體空間的演算法。此類演算法非常適合那些需要即時又本身記憶體空間不多的系統，嵌入式即時系統即是一例。

針對這樣的需求，我們利用 fetch&store 設計出既公平（bounded-bypass）又只需二個變數的演算法。並且證明用相同的指令已經達到空間複雜度的下限，亦即不可能有更好的演算法。

相關的成果已經先以會議論文發表，詳見參考文獻[15]。目前正準備投稿至期

刊。

五、參考文獻

[1] J. A. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171--184, December 1993.

[2] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183--205, January 1982.

[3] T. S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 148--156, December 1993.

[4] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[5] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843--862, September 1998.

[6] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90--114, January 1989.

[7] S. S. Fu and N.-F. Tzeng. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):628--639, June 1997.

[8] T.-L. Huang. Fast and fair mutual exclusion for shared memory systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 224--231, June 1999.

[9] T.-L. Huang and C.-H. Shann. A comment on A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):414--415, April 1998.

[10] N. A. Lynch. *Distributed Algorithm*. Morgan Kaufmann, 1996.

[11] 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, February 1991.

[12] I. Rhee. Optimizing a FIFO, scalable spin lock using consistent memory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 106--114, December 1996.

[13] K. Sakamura and N. Koshizuka. T-engine: the open, real-time embedded-systems platform. *IEEE Micro*, 22(6):48--57, 2002.

[14] K. M. Zuberi and K. G. Shin. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 25--34. IEEE, June 1997.

執行本計畫之著作

[15] Sheng-Hsiung Chen, Ting-Lu Huang. A fair and space-efficient mutual exclusion using *read/write* and *fetch&store* primitives. In *Proceedings of the International Conference on Informatics, Cybernetics, and Systems (ICICS'03)*, pp. 1059-1064, Kaohsiung, Taiwan, Dec. 2003. Available at: http://www.csie.nctu.edu.tw/~chenss/papers/ICICS2003.pdf