Parallel and Distributed Computing

No. 15

Adding local spin to mutual exclusion algorithms: a generic approach and its practice

互斥演算法加上近端自轉：方法與實例演練

Ting-Lu Huang (黃廷祿)

National Chiao Tung University, Taiwan, ROC

Department of Computer Science,

Email: tlhuang@cs.nctu.edu.tw

Cheng-Ming Chien (簡正明)

National Chiao Tung University, Taiwan, ROC

Department of Computer Science,

Email: cmchien@cs.nctu.edu.tw

Abstract — Busy waiting is common in shared memory mutual exclusion algorithms. To reduce memory contention incurred by busy waiting, we follow the concept of local spin made popular by Mellor-Crummey and Scott and propose a generic approach for adding local spin to mutual exclusion algorithms of the atomic read/write model. Taking Eisenburg-McGuire algorithm as an example, two local spin versions were obtained. The first is an easy product of the generic approach. The second, with better inter-process communication made possible by an in-depth understanding of the algorithm, significantly reduces the number of remote memory accesses.

***Keywords*—mutual exclusion, local spin, shared memory, atomic read/write register**

摘 要 **— 在 shared memory mutual exclusion algorithms 中，免不了要 busy waiting，為了減少 shared memory contention，我們援用 Mellor-Crummey and Scott 之 local spin 概念，提出一種方法，一般 mutual exclusion algorithms of the atomic read/write model 可依此加上 local spin 機制，以 Eisenburg-McGuire mutual exclusion algorithm 為實例，可輕易加上 local spin 得到初步結果，經過深入瞭解此 algorithm 運作細節之後，可再改進 processes 之互動而大幅降低 remote memory access 次數。**

***關鍵詞*—mutual exclusion, local spin, shared memory, atomic read/write register**

一、 Introduction

Mutual exclusion problem 在 multiprocessing system 中是一個很基本的問題，對於某些不可分割的資源，必須確保在任何瞬間最多只有一個 process 被允許存取該資源；一台印表機無法同時給多個 processes 同時使用，所以印表機需要在 mutual exclusion 的條件下才能正常使用。概念上，mutual exclusion problem 就是在設計一個 concurrent algorithm，其中有一段稱為 critical region (或 critical section，**CS** for short)的 code 保證任何瞬間最多只有一個 process 能夠進入 CS 執行。Mutual exclusion problems 的定義包含二項:

(1)**safety property** (任何時刻至多一個 process 可以執行 CS)

(2)**progress property** (如果在某時刻 s 無任何 process 在 CS 而且已有 process 在 trying region，則存在某一個 process 在 s 之後某時刻可進 CS)。

Mutual exclusion algorithm 的 code 可區分為四個 region: trying region、critical region (CS)、exit region、remainder region[10]，在進入 CS 之前必須透過 trying region 中所設計的 contention protocol 來決定哪 process 可以進入 CS，在 contention protocol 中所有的 processes 都必須讓其它 processes 發現自己是 contender，CS 之後緊接著 exit region，在 exit region 中必須讓所有其它 processes 發現到自己已經離開 CS，在完成 exit region 之後，process 就回到 remainder region，如 Figure 1 。

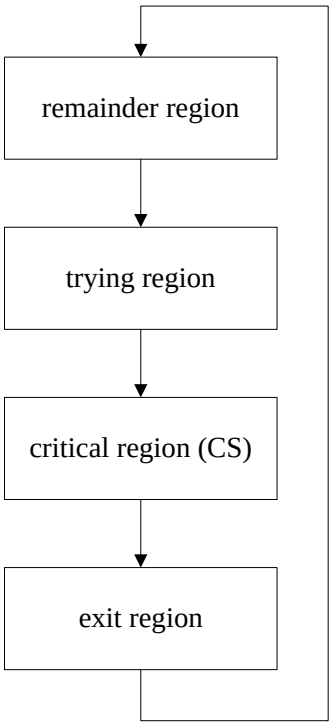

Figure 1: Life cycle of mutual exclusion algorithms

當 algorithm 執行時，若已有 process 進入 CS，其它在 trying region 的 process 就必須 waiting，常見的作法有主動地 spinning on shared variables 或是被動地 sleeping 讓 OS 的 scheduler 來負責喚醒，但是在某些沒有 scheduler 的系統架構下，只能用 spinning 的方式，例如作業系統本身，而 spinning on shared variables 會傷害系統效能，processes 必須一直將 shared memory 中的資料讀取到 register 做條件判斷，在 memory bandwidth 有限的狀況下，shared memory contention 將造成系統的負擔。

Local spin 是一個解決 shared memory contention 的方法之一，在 cache coherence (CC) 和 distributed shared memory (DSM)的架構下，讓需要 busy waiting 的 process 只需讀取 cache 或 local shared memory 中的 data，以減少 remote memory access 的次數，提升系統的效能。 DSM 系統下的每個 process 擁有屬於自己的 local shared memory, 可供自己作快速的 local access, 別的 process 也可用 remote access 方式讀寫此塊 memory, 但速度慢許多。MCS [11] 在 1991 年提出符合 local spin 概念的 mutual exclusion algorithms，並且在 2006 年獲得了 Edsger W. Dijkstra Prize，但它使用了 compare-and-swap 指令，依賴功能較強的硬體才能實現；之後國內亦有 local spin 相關研究,如 Tsay [3] 與 Chen and Huang[4]。在一般 atomic read/write model 之下 local spin algorithms 難度較高, 除了上述 Tsay [3] 兼顧 algorithms derivation 正規方法之外，尚有 Yang and Anderson[13] 以 tree 的方式將 time complexity under contention 從 O(n)降到 O(log n)，並引用 Lamport's fast mutual exclusion algorithm[8] 的精神，一樣具備 O(1) in contention-free cases，但是如果在 fast path 遇到 contention，則 worst case 仍然是 O(n)；之後 Anderson and Kim[1]將 fast path 中 time complexity under contention 又降到 O(log n)。

本文提出一個 generic approach，一般以 atomic read/write registers 為 model 的 mutual exclusion algorithms, 均可透過此 generic approach 加上 local spin 的機制以便在 DSM 系統上執行，之後以 Eisenburg-McGuire algorithm[6] 為實例演練 generic approach，得到初步 local spin algorithm；之後又分析此 algorithm 運作細節，大幅更改 program structure, 更進一步減少 exit

region 使用 remote write 叫醒別 process 的次數到 1 次，稱之為“focused release”，有了 focused release 後，我們找到“fast track”，讓被釋放的 process 可省去檢查其他 process 狀態的一連串 remote memory access 動作而直接進入 CS，所得 algorithm 仍然維持原本正確性。本文所提 local spin algorithms 之 worst case time complexity 雖不如 Anderson and Kim[1]， 但如果只討論系統在 heavy loading 時的表現,則因經常有許多 process 在 trying region 等待，有利於本文 algorithms 之執行效率，以其中最好的版本 EM_2 而言,進出 CS 一次所需 remote memory access 平均次數降低到很小 constant 值 。

二、 Definitions

(a) update status:

每一 process 會利用專屬自己才可更改的 **local** shared variable 來記錄執行的路徑或狀態，以供其它 processes 區分該 process 目前在 algorithm 中執行的進度，這種行為統稱為 update status。如本文 algorithm EM_0 之 flag(i) 專屬於 process i。

(b) extended CS :

範圍涵蓋部分 trying region、整個 CS、部分 exit region，如 Figure 2。在 trying region 的 contention protocol 中必然有一段為維持 safety-property 的 loop，以確保在通過該段 checking 之後的 process 必然是唯一可以進入 CS 的 process，從該 checking 判別可以進 CS 到真正進 CS 之前的範圍，將它納入到 extended CS 中；而在 exit region 中 update status 執行之前的所有動作也可以納入 extended CS 中，update status 執行之後才真正地把 CS 釋出。經過這兩方向擴張後,整個 extended CS 是一個連續的 region，在同一時間只有一個 process 可以執行這範圍內的指令，這概念對設計 mutual exclusion algorithm 有很大的幫助，在這 extended CS 範圍內的指令受到 mutual exclusion 保護，同時刻至多有一 process 可以執行。因此原本較難分析的 concurrent program 在此範圍內可視之為 sequential program。

(c) *permitted* (i):

此 approach 額外使用了這 n 個 shared variables (稱為 1-dimensional)以便 local spin 使用。For every i， $0 \le i \le n-1$， *permitted* (i) 是 writable by all processes，readable by process i，每個 *permitted* (i) 位於 process i 的 local shared memory 中。當 process i 需要 busy waiting 時會 spinning on *permitted* (i)，這種 spinning 不會引發 memory contention, 而任何一個 process j 在離開 CS 後要把所有正在 spinning 的 process i 用 remote write to *permitted* (i) 方式叫醒 i。

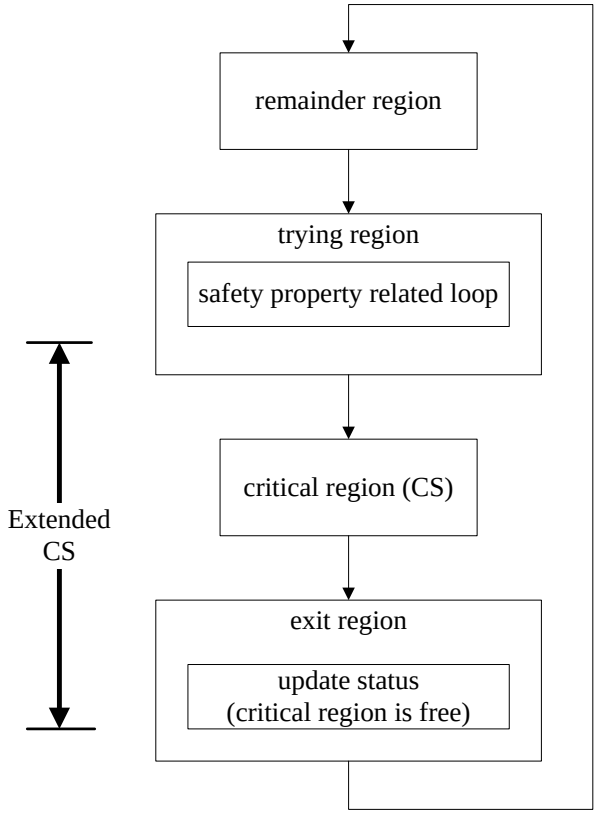

Figure 2: Extended CS

(d) await some condition:

等待 some condition 為 true 才算完成的指令。

(e) focused release:

process 在 exit region 中只釋放特定的一個 local-spinning process。

(f) fast track:

local-spinning process 在 trying region 中被釋

放時，在屬於 focused release 的 前提下，可直接進入 CS 的捷徑。

三、 The proposed generic approach

基本理念是要在原有 algorithm 外加一些 variables 與指令,讓多數在 trying region 開始競爭的 processes 受到阻擋之後,就進入 local spin 狀態暫停活動,競爭者需要在 trying region 的迴圈(program loops)來回好幾次,陸陸續續可能又有競爭者暫停活動,可是絕對要讓一個 process 順利通過 trying region 所有 loops 而進 CS。當它到 exit region 時就以 n-1 次 remote writes 叫醒所有暫停活動者,系統又恢復到 algorithm 本來應有情況。然後又再次不停地重覆上述競爭過程。整個執行細節,如果不看外加的這些 variables 與指令,則仍然符合原有 algorithm 之運作法則,上述暫停活動可視為暫時因 context switching 而沒動作,如此而已。其功效卻可避免許多 processes 以 remote memory access 方式在 trying region 做無謂的嘗試。這理念可再詳述如下。

在 trying region 中的 contention protocol 讓 processes 決定何者可以成為 winner 而進入 CS，該 protocol 的設計會使 mutual exclusion algorithm 兼具 progress property 與 safety property，觀察 Dijkstra[5] 、 Knuth[7] 、 Eisenburg-McGuire[6] 、 Burns[2] 、 Lamport[9] 等 mutual exclusion algorithms，可在這些 contention protocols 的 program code 找到 progress property related loop (**P-loop** for short) 和 safety property related loop (**S-loop** for short)，這兩種 loop 缺一不可，互相呼應，其結構型態頗多，有巢狀的型態、有完全分離的型態、也有合而為一的型態，通常某個 process 若能通過 S-loop 的阻擋就代表已經成為最後結果之 winner 而可以進入 CS， 而 P-loop 功能為促使系統免於陷入死結 (deadlock)，其主要目的是要阻擋一些 process 且要讓少數比較優勢之 process 通過， process 在 trying region 執行路徑中，可能要經過幾次 P-loop、 S-loop、 P-loop 、S-loop 、P-loop … 交替阻擋，若系統處於 heavy loading 狀態下,大量 processes 開始執行 trying region，此 P-loop 往往是絕大多數 process 被阻擋之處，同時也是引發 memory contention 最嚴重的；有趣的是，此 P-loop 即是加上 local spin 絕佳之處，主因在於 P-loop 阻擋 process 之功能包含積極面與消極面，分述如下。

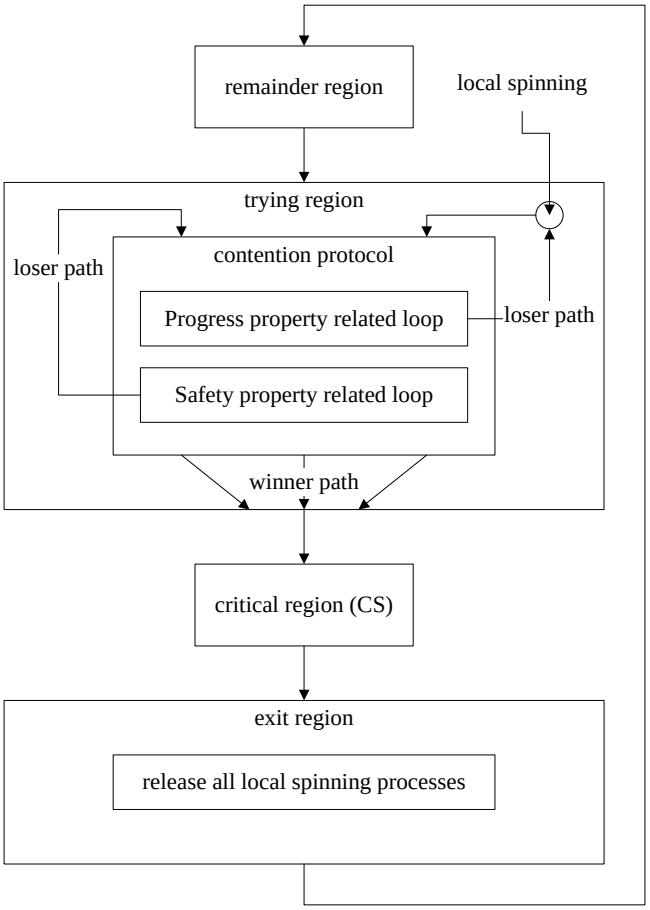

Figure 3: The generic approach

在大量processes執行trying region時，P-loop **阻擋多數process,**僅讓少數process繼續往前**(**此為阻擋功能的**積極面)**。透過shared variables每個process依著algorithm法則測知自己是可以脫離P-loop，或者必須停留在P-loop；一般狀況之下，僅有少數競爭者可以脫離loop (測知為暫時winner)，其他大部分競爭者必須停留於P-loop中(測知為暫時loser)。本文所提之approach要讓process i 在P-loop測知自己目前已經不可能順利通過P-loop,則馬上spin on 自己的local shared variable *permitted(i)*，靜待其他某process進入CS在exit region用remote memory write to 這個

variable叫醒自己。如此，僅有上述少數processes會繼續往前，自然就降低系統remote memory access次數。設計得好的P-loop會讓繼續往前的processes個數，經過幾次P-loop、 S-loop、 P-loop … 交替篩選後很快就降低到剩下一個。這單一的process再往前作S-loop的測試時因為沒有其他process干擾,可以順利的脫離而進CS。因此, local spin設置在P-loop,可以大幅降低系統 remote memory access次數。

另一方面，P-loop 雖會阻擋多數 process,**但絕不阻斷所有 process 往前(**此為阻擋功能的**消極面)**。這消極面有助於此 generic approach 所得之 algorithms 可免陷入 local spin 機制所引發的 deadlock 。設計 local spin algorithms 過程中，如果 local spin 的位置與方式不當，很可能讓系統陷入 deadlock,例如某段時間內 k 個競爭者($k \le n$)在 trying region 執行當中都測知自己暫為 loser，因而全部進入 local spin 等待狀態，則此時這 k 個 processes 都在等待其他 process 來叫醒自己，此即為 deadlock，必須預先設防。這問題對於初學者並不容易,若依本文之approach將 local spin 設在 P-loop 之 loser path (如 Fig.3 右上角小圓形所示)上的**適當位置**，即能避免這種 deadlock。如何選定 local spin 適當位置,仍需要對個別 algorithm 的 progress property 內容有某種程度的瞭解，詳述如下。

原始algorithm 擁有mutual exclusion problem定義下的**safety property**與**progress property**。我們的generic approach以不更改原始algorithm的基本program structure為首要原則，利用額外的local shared variables (如上述*permitted*) 供processes做local spin用，完全保留了原始 algorithm 原有的 safety property，變更的只是processes在做local spin 時暫時停止執行原始algorithm指令，等待其他某process進CS後在exit region以remote write叫醒自己恢復執行原始algorithm指令。困難的是,要如何確保一定存在某process進CS? 基於原 algorithm 具有的progress property，如果沒有設置local spin的話,**必然存在一個process j可進入CS。**本文所提generic approach的一項重要工作在於尋找local spin適當位置以滿足兩條件: (1)上述process j 絕對不會做local spin,且(2) 所有做local spin 的process絕對不會妨礙process j進CS。則process j在exit region時會釋放(叫醒)所有正在local spinning的processes，讓整個系統又可以繼續執行原始algorithm的正常指令，不會因為local spin而陷入deadlock。

如何選定local spin 適當位置,可依上面兩條件分別說明。【條件1】: 執行P-loop受到阻擋者不會是將來要搶先進CS者,將local spin設在loser path上,就可排除process j做local spin之可能性,因為 j 每次經過P-loop時都通過(但在S-loop時可能受到幾次阻擋)。【條件2】: 因為不更改原來program structure,基本上做local spin者只是暫停,不可能做出妨礙 process j進CS的動作,但是它有可能因為在P-loop中太早就判定自己是loser而進入local spin等待,本應依原有algorithm與process j有些互動而促成 j 搶先進CS,卻因為太早『不作為』而消極性地妨礙到 j 進CS。事實上,loser之判定即使過度拖延也不會引起錯誤,僅會影響到local spin之成效。因此,將loser之判定適度延後是比較安全。實務上在處理個別algorithm時,若能瞭解作者對於**progress property之證明**,就很容易找到設置local spin不早也不晚的適當位置。有了適當位置,得到local spin algorithm的明確版本之後,可仿照原作者progress property之證明得到新algorithm的progress property之證明。萬一證明失敗,可再找更晚一點的local spin位置，再嘗試證明。如此才真正回答了前段的困難問題。

總而言之，the generic approach必需維持與善用原始algorithm之正確性: (1) local spin機制應使用額外設置的variables,以維持原始algorithms既有的*safety property*,(2) local spin機制設置點之選擇應善用原始algorithms既有的*progress property*來預防local spin可能引起的deadlock。

四、Eisenburg-McGuire Mutual Exclusion Algorithm

(A) The original version : Algorithm EM_0

```
Shared variables:
  ● turn ∈ {0,.....,n-1}, initially arbitrary, writable by all processes
  ● for every i, 0 ≤ i ≤ n-1, flag(i) ∈ {idle, want-in, in-cs}, initially idle, writable by process i and readable by all processes

Process i:   ( private variable   j : integer )

** Remainder Region **

repeat
        flag(i) := want-in
        j := turn
        while j ≠ i do
                if flag( j) ≠ idle then j := turn
                else j := j+1 mod n fi
        od
        flag(i) := in-cs
        j := 0
        while ( j < n) and ( j = i or flag( j) ≠ in-cs) do j:= j+1 od
until ( j ≥ n) and (turn = i or flag(turn) = idle)
turn := i                          ▷ extended CS begin

** CS **

j := i+1 mod n
while flag( j) = idle do j := j+1 mod n od
turn := j
flag(i) := idle                    ▷ extended CS end

** Remainder Region **
```

Algorithm 1: EM_0

Eisenburg-McGuire 在 1972 年所提 mutual exclusion algorithm[6] (亦見 Operating System Concepts, 5th ed., Silberschatz & Galvin，p.201) 除了符合基本的正確性外又具備 n-1 bounded waiting 的特性，主要是依靠 algorithm 中 shared variable *turn*，它的值只在 extended CS 中被修改，也就是說任何的瞬間最多就只有一個 process 能夠更改 *turn* 值；process 離開 CS 在 exit region 中要作 linear search 找出下一個 non-idle process 並將 *turn* 值改為此 process 之 id (此動作受到 extended CS 保護)，而 *turn* 值所指的 process 只要是 non-idle 就是下一個能優先進入 CS 的 process。

Lemma 4.1 ：在 Eisenburg-McGuire mutual exclusion algorithm 中， process α 在 exit region 中所找到的下一個 non-idle process β 將必然繼 process α 之後進入 CS。

```
Shared variables:
  ● turn ∈ {0,.....,n-1}, initially arbitrary, writable by all processes
  ● for every i, 0 ≤ i ≤ n-1, flag(i) ∈ {idle, want-in, in-cs}, initially idle, writable by process i and readable by all processes
  ● for every i, 0 ≤ i ≤ n-1, permitted(i) ∈ {true, false}, writable by all processes and readable by process i

Process i:   ( private variable   j : integer )

** Remainder Region **

repeat
        flag(i) := want-in
        j := turn
        while j ≠ i do          ▷ P-loop
                permitted(i) := false
                if flag( j) ≠ idle then
                        await permitted(i)
                        j := turn
                else j := j+1 mod n fi
        od
        flag(i) := in-cs
        j := 0
        while ( j < n) and ( j = i or flag( j) ≠ in-cs) do j:= j+1 od
until ( j ≥ n) and (turn = i or flag(turn) = idle)   ▷ S-loop
turn := i                          ▷ extended CS begin

** CS **

j := i+1 mod n
while flag( j) = idle do j := j+1 mod n od
turn := j
flag(i) := idle                    ▷ extended CS end
for j = 0 to n-1 do permitted( j) := true od

** Remainder Region **
```

Algorithm 2: EM_1

Proof：假設有 process γ 搶先在 process β 之前繼 process α 之後進入 CS，之前當 process γ 測試 $(j \geq n)$ *and* $(turn = i$ *or* $flag(turn) = idle)$ 邏輯條件時所得結果必然為 true，由於 $(j \geq n)$ 成立，當時 process α 必已作完 extended CS 最後一行:*flag(i):=idle*，當然早已作完前一行: *turn := j* 。

process γ 所讀 *turn* 值必然為 β 且 *flag(turn)* 值必然為 non-idle, 那麼 process γ 根本不可能通過上述 "*and*" 的邏輯條件而進 CS, 這是矛盾。所以 process β 必然繼 process α 之後進入 CS。

Shared variables:
- turn ∈ {0,.....,n-1}, initially arbitrary, writable by all processes
- for every i, 0 ≤ i ≤ n-1, flag(i) ∈ {idle, want-in, in-cs}, initially idle, writable by process i and readable by all processes
- for every i, 0 ≤ i ≤ n-1, permitted(i) ∈ {true, false}, writable by all processes and readable by process i

Process i: (private variables j: integer; spin-wake-up: bit)

** Remainder Region **

```
spin-wake-up := false
repeat
        flag(i) := want-in
        j := turn
        while j ≠ i do          ▷ P-loop
                permitted(i) := false
                spin-wake-up := false
                if flag( j) ≠ idle then
                        await   permitted(i)
                        spin-wake-up := true
                        j := turn
                else j := j+1 mod n fi
        od
        flag(i) := in-cs
        if spin-wake-up then goto CS fi   ▷ fast track
        j := 0
        while ( j < n) and ( j = i or flag( j) ≠ in-cs) do j:= j+1 od
until ( j ≥ n) and (turn = i or flag(turn) = idle)   ▷ S-loop
turn := i                       ▷ extended CS begin
```

** CS **

```
j := i+1 mod n
while flag( j) = idle do j := j+1 mod n od
turn := j
flag(i) := idle                 ▷ extended CS end
if j = i then for k = 0 to n-1 do permitted(k) := true od
else permitted( j): = true fi           ▷ focused release
```

** Remainder Region **

Algorithm 3: EM_2

這個 *Lemma* 掌握了 algorithm EM_0 的 processes 之間溝通交接進 CS 的規律性，也是吾人賴以改進 generic approach 初步結果，增加 focused release 和 fast track 兩項功效的重要基礎。

(B) Adding local spin to EM_0 by the generic approach : Algorithm EM_1

觀察 the original algorithm 之後，發現在 trying region 中只有一個 P-loop，在大量 processes 競爭 CS 時，該 P-loop 中 processes 不斷重新讀取 *turn* 值與 flag 值造成嚴重的 shared memory contention，依照 generic approach 的精神，可以確定這個 P-loop 非常適合加上 local spin，其它的 loop 不但屬於 S-loop 而且加上 local spin 會造成 dead lock。由於 local spin 是額外加上 variables 和 spin 動作，並沒有變動與 safety property 有相關的 code，所以仍維持原有的 safety properties，實際上 generic approach 所得之 algorithm EM_1 只在原始的 algorithm EM_0 加上三行 code (以斜體加粗体表示) 。

在加上 local spin 之後比較 EM_0 與 EM_1 行為，基本上驗證了前面 the generic approach 章節所述, 參考原著 progress property 證明，稍加瞭解 P-loop 中 processes 互動方式,即可找到 loser path 上的適當位置加 local spin, 原本許多 processes 在 P-loop 不斷地重新讀取 *turn* 值與 *flag* 值的動作因為 local spin 機制的加入而被暫停。因為 local spin 位置選擇成功,讓我們援用 EM_0 的 progress property 證明即可以輕易仿製成 EM_1 的 progress property 證明,知道必然存在一個 process j 可以進 CS。那麼當 process j 出了 CS 在 exit region 執行這一行 code: *for j = 0 to n-1 do permitted(j) := true od* 釋放所有 local-spinning processes 後，就可以讓系統其他 processes 繼續進行，而原本造成非常嚴重的 shared memory contention 也因為 local spin 有了大幅度的改善。

(C) Adding focused release and fast track to Algorithm EM_1 : Algorithm EM_2

使用 generic approach 可輕易地得到初步的 local spin algorithm (EM_1)，這過程吾人不必深入瞭解 algorithm EM_0 之運作細節，頂多只要能認出 trying region 中 P-loop 的 loser path，瞭解 processes 在 P-loop 中互動之方式,找出適當位置加上 local spin code 即可，其結果已經讓 memory contention 降低很多，且因不更改原有 program structure，正確性可確保。這種保守的方式適合於初學者。熟悉 concurrent programming 技巧者，若能深入瞭解個別 algorithm 之運作細節，可以

進一步更改原有的 program structure 將 local spin 之功效發揮得更好，以 EM_1 為實例詳述如何加入 focused release 與 fast track 兩項功效如下。

(1)Focused release: 以 Eisenburg-McGuire 原始 algorithm (EM_0)來說，在其 exit region 有一段 linear search，其主要的目的是為了維持 n-1 bounded waiting 的性質，我們作 local spin 時可利用這個 linear search 的動作找到單一 process 來做 release 標的(故稱為 focused release， 若依 generic approach 是要 release 所有其他 spinning processes)，依據 *Lemma 4.1*，所釋放的單一 process 必然可進入 CS，那麼只需要一次 remote write 叫醒這 process 就夠了，其餘的 n-2 個 remote writes 只是讓被叫醒的 processes 白忙而已，終究得再次測知挫敗而進入 local spin。因此，若能深入瞭解原始 algorithm 運作細節則可用 focused release 大幅度地降低 remote write 次數，Fig.3 Algorithm EM_2 最後兩行 code 即充分掌握 focused release 之機會，當 linear search 測到 non-idle process 標的時僅使用一次 remote write 叫醒它；**但是**,當 linear search 一遍尚未測知 non-idle processes，則系統後續是否已有 non-idle processes **不得而知**，吾人只能保守地使用 n-1 次 remote writes 將 *permitted* bits 準備好，以維持後續運作之正確性。基於 asynchronous atomic read/write shared memory model 之本質，此刻縱使 linear search 作二遍或更多遍尋找下一個 non-idle process 仍然無法克服這“不得而知”的基本困難。若能動用功能較強的 read-modify-write model 協助， 則很容易測知後續是否已有 non-idle processes， MCS [11] 有這種 algorithms。

(2)Fast track: 設計 fast track 時則必須謹慎驗證是否會破壞 safety property：有了 fast track 之後，進入 CS 的 path 會多一個，safety property 正確性之分析變成比較複雜。目前在嘗試加上 fast track 時仍需依 algorithm 之差別而個別推論，仍無一般常規可循，也不是任何 algorithm 都有 fast track 的存在。以 Eisenburg-McGuire mutual exclusion algorithm 來說，只要能夠設法留下路徑記錄讓被釋放的 process 能夠辨識出自己是目前唯一可以進入 CS 的 process，那就有足夠理由可沿 fast track 直接跳到 CS，省下不必要的 n-1 次 remote memory accesses (原本目的在於測知其他 processes 之狀態)。以這樣的構想，在 fast track 的設計上是利用額外的 private variable (如 EM_2 中 *spin-wake-up*) 記錄執行路徑。

Claim：EM_2 的 fast track 維持 safety property。

Proof：當 process β 測知自己符合 fast track 的 *spin-wake-up* 條件時，若能證明 process β 是目前唯一能進 CS 之 process,則 β 可以免掉檢查 n-1 個 flag bits 的動作，直接進 CS。假設 β 在測知可進 CS 但未進 CS 時停止不動，則由其過去所走路徑可以推知，必然存在另一 process α 曾在 exit region 以 remote write 叫醒 β，兩者之互動符合 Lemma 4.1 所述之前提，因此除了 β 外無其它 process 可以進 CS。

五、Conclusion

我們提出的 generic approach 適用於一般 mutual exclusion algorithms of atomic read/write model， Taubenfeld[6] 的 local-spinning bakery algorithm 採用了額外 n^2 (2-dimensional) permitted bits，在 shared memory contention 的降低程度或許在某些情況下比此文 generic approach 優良，但是 2-dimensional permitted bits 的使用方式並未提供一套可依循的方法，對於初學者而言即使 algorithm 簡短如 EM_0 仍然很容易將 local spin 位置放錯,產生 dead lock。

除了 Eisenburg-McGuire algorithm 外，吾人目前已對十個屬於 atomic read/write model 類別的 mutual exclusion algorithms 作 generic approach 演練，尚未遇到行不通的例子。

本文以循序漸進方式，提出一個適用性很廣的 generic approach ， 並以流傳甚廣的 Eisenburg-McGuire algorithm 為實例，提供一連串加上 local spin 功能的演練過程，最後所得版本 EM_2 程式結構精簡，保有原版 EM_0 具有 n-1 bounded waiting 的優點，當系統處於 heavy loading 時 local spin 的功效特別好。假設 process

在 trying region 的 P-loop 中使用了 remote access **平均次數為 m 次**就被阻擋而進入 local spin 狀態，在 exit region 找下一個 non-idle process 所需 remote access **平均次數為 k 次**, 則經由 fast track 與 focused release 執行路徑,可計算出 process 進出一次 CS 所需 remote access 平均次數為 **m+k+4 次**,其中 m+2 次在 trying region, k+2 次在 exit region。當系統 heavy loading 程度越高, m 值與 k 值越小。

六、References

[1] James H. Anderson and Yong-Jik Kim, “A new fast-path mechanism for mutual exclusion”, Springer-Verlag Distributed Computing, Volume 14, Issue 1, January, 2001, Pages 17-29.

[2] James E. Burns, “Mutual exclusion with linear waiting using binary shared variables”, ACM SIGACT News, Volume 10, Number 2, Summer, 1978, Pages 42-47.

[3] Yih-Kuen Tsay, “Deriving a Scalable Algorithm for Mutual Exclusion,” Lecture notes in computer science, Springer Berlin, Vol. 1499, 1998, Pages 393-407.

[4] Sheng-Hsiung Chen and Ting-Lu Huang, “A Tight Bound on Remote Reference Time Complexity of Mutual Exclusion in the Read-Modify-Write Model”, Journal of Parallel and Distributed Computing, Volume 66, Issue 11, November, 2006, Pages 1455-1471.

[5] Edsger Wybe Dijkstra, “Solution of a problem in concurrent programming control”, Communications of the ACM, Volume 8, Issue 9, September, 1965, Pages 569.

[6] Murray A. Eisenberg and Michael R. McGuire, “Further comments on Dijkstra's concurrent programming control problem”, Communications of the ACM, Volume 15, Issue 11, November, 1972, Pages 999.

[7] Donald E. Knuth, “Additional comments on a problem in concurrent programming control”, Communications of the ACM, Volume 9, Issue 5, May, 1966, Pages 321-322.

[8] Leslie Lamport, “A Fast Mutual Exclusion Algorithm”, ACM Transactions on Computer Systems, Volume 5, Number 1, February 1987, Pages 1–11.

[9] Leslie Lamport, “A new solution of Dijkstra's concurrent programming problem”, Communications of the ACM, Volume 17, Issue 8, August, 1974, Pages 453-455.

[10] Nancy A. Lynch, “Distributed Algorithms”, Morgan Kaufmann, 1996.

[11] John M. Mellor-Crummey and Michael L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”, ACM Transactions on Computer Systems, Volume 9, Number 1, February, 1991, Pages 21-65.

[12] Gadi Taubenfeld, “Synchronization Algorithms and Concurrent Programming”, Prentice Hall, 2006.

[13] Jae-Heon Yang and Jams H. Anderson, “A fast, scalable mutual exclusion algorithm”, Springer Berlin / Heidelberg Distributed Computing, Volume 9, Number 1, March, 1995, Pages 51-60.