國立交通大學

資訊科學與工程研究所

碩 士 論 文

互斥演算法加上近端自轉：原則與實例演練

Adding local spin to mutual exclusion algorithms: principles and experience

研 究 生：簡正明
指導教授：黃廷祿　教授

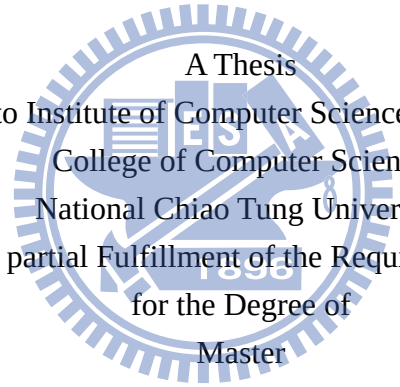
中 華 民 國 九 十 八 年 十 一 月

互斥演算法加上近端自轉：原則與實例演練
Adding local spin to mutual exclusion algorithms: principles and experience

研 究 生：簡正明　　Student：Cheng-Ming Chien
指導教授：黃廷祿　　Advisor：Ting-Lu Huang

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所

碩 士 論 文



A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

November 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年十一月

互斥演算法加上近端自轉：原則與實例演練

學生：簡正明　　指導教授：黃廷祿 博士

國立交通大學資訊科學與工程研究所碩士班

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，提出一種通用的方法(generic approach)，一般在 atomic read/write model 下的 mutual exclusion algorithms 均可依此方法加上 local spin 機制，以 Dijkstra's、Knuth's、Eisenburg-McGuire's、Lamport's bakery 和 Burns' mutual exclusion algorithm 為實例，透過 safety property 和 progress property 的證明，清楚分辨這些 algorithms 在 trying region 的結構之後，即可輕易加上 local spin 得到大幅降低 shared memory contention 的初步結果，再深入瞭解 algorithms 運作細節，若 algorithm 具備 bounded waiting 的性質，則可以再改進 processes 之互動而大幅降低 remote memory access 次數。

Gadi Taubenfeld 提出 local-spinning bakery mutual exclusion algorithm，我們嘗試引用相同的方式（使用 2-dimensional permitted bits）為其它在 atomic read/write model 下的 mutual exclusion algorithms 加上 local spin，利用 tree 的結構分析 the generic approach 與此方法之間的差異，並證明出此方法並不是所有的 mutual exclusion algorithms 都適用。

綜合實例演練的經驗，我們歸納出依據 algorithm 具備的性質較適合使用何種方式加上 local spin 機制，大致上可區分為兩大類：具有 starvation 狀況的 algorithm 適合使用 2-dimensional permitted bits 加上 local spin；具備 bounded waiting 性質的 algorithm 適合 the generic approach 再加 focused release 的方式加上 local spin。

關鍵詞：mutual exclusion、local spin、shared memory、atomic read/write register

Adding local spin to mutual exclusion algorithms: principles and experience

Student：Cheng-Ming Chien　　Advisor：Dr. Ting-Lu Huang

Institute of Computer Science and Engineering
National Chiao Tung University

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 Dijkstra's, Knuth's, Eisenburg-McGuire's, Lamport's bakery and Burns' mutual exclusion algorithm as examples, two levels of local spin version 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, e.g. bounded waiting, significantly reduces the number of remote memory accesses.

There is another approach used in Gadi Taubenfeld's local-spinning bakery mutual exclusion algorithm by adding 2-dimensional permitted bits. We try this approach to add local spin to the other mutual exclusion algorithms of the atomic read/write model. We find out the difference between the generic approach and this approach by the tree structure. And this approach is not suitable for all mutual exclusion algorithms.

According to the experience of adding local spin, we divide mutual exclusion algorithms into two parts: the algorithms with starvation are suitable for using 2-dimensional permitted bits to add local spin; the algorithms with bounded waiting are suitable for the generic approach plus focused release to add local spin.

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

誌　謝

這份論文能順利完成，首先要感謝我的學位指導教授黃廷祿老師。

老師對於納入 OS 教科書中廣為人知的 Eisenburg-McGuire's mutual exclusion algorithm 研究甚深，並由此 algorithm 發想出本論文中最重要的 generic approach 以及 focused release、fast track。在老師引領下所進行的 debug algorithm EM_2、EM_3、EM_4 研究過程，讓我不僅熟悉了 generic approach 的精神，也對互斥演算法的設計有更深刻的了解，進而催生此論文的誕生。

老師不僅指導研究演算法應有檢驗正確性的觀念，亦教授了如何利用時間軸與事件發生的邏輯順序來證明 safety property；在 Knuth's mutual exclusion algorithm 具備 $2^{n-1}-1$ bounded waiting 性質的證明裡，也是由老師提供一個 5 processes 的 execution sequence 例子，讓我獲得找出 recursive function 的靈感，進而得以證明。

由衷感激老師帶領我進入分散式演算法的領域，並使我能對互斥演算法有深入的了解。更感謝口試指導委員曾建超所長及黃冠寰教授百忙中撥冗給予細心指正、教導。

感恩我的父母長久以來的鼓勵、支持與付出心力，提供一切支援讓我得以專心於學業，並於良好基礎教育所打下的扎實根基中進行研究。值此將為人父的時刻，更加體會父母愛子之心，及其所給予之關愛與照護之情。在論文即將完成的最後這段時間，發現母親罹患腦瘤身受病痛之苦，衷心祈禱母親能安然度過開刀過程之苦，早日恢復健康與笑顏。

最後謝謝我的太太求學期間提供許多建議，她取得台大、師大雙碩士的經驗對我能順利完成學業提供許多幫助。願我們的寶貝能健康平安的誕生，一同分享這份喜悅。

目　錄

表 目 錄

圖 目 錄

演 算 法 目 錄

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 t 無任何 process 在 CS，而且已有 process 在 trying region（如：Figure 1），則存在某一個 process 在 t 之後某時刻可進 CS。

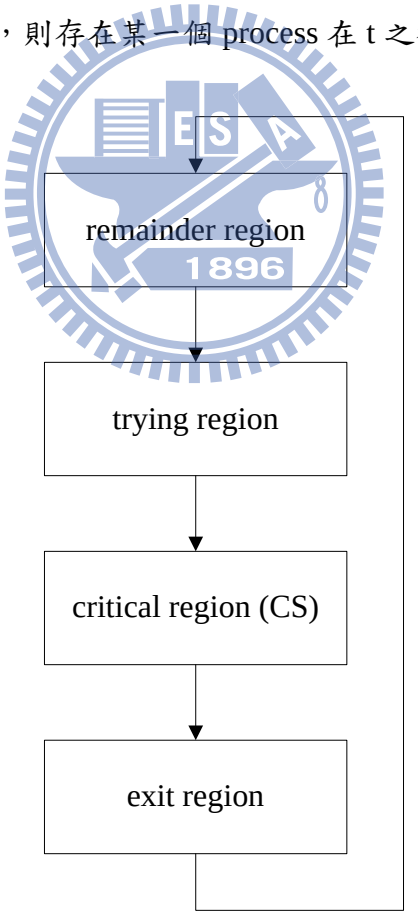


Figure 1: Life cycle of mutual exclusion algorithm

Mutual exclusion algorithm 的 code 可區分為四個 region：trying region、critical region（CS）、exit region、remainder region[9]，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。

當 algorithm 執行時，若已有 process 進入 CS，其它在 trying region 的 processes 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 distributed shared memory（DSM）的架構下，讓需要 busy waiting 的 process 只需讀取 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，其它 processes 也可用 remote access 方式讀寫此塊 memory，但速度慢許多。MCS[10]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[12]與 Chen and Huang[3]。在一般 atomic read/write model 之下 local spin algorithms 難度較高，除了上述 Tsay[12]兼顧 algorithms derivation 正規方法之外，尚有 Yang and Anderson[13]以 tree 的方式將在 contention 之下 trying region 中的 time complexity 從 $O(n)$ 降到 $O(\log n)$，並引用 Lamport's fast mutual exclusion algorithm[7]的精神，在 contention-free 的狀況下只需 $O(1)$ 即可進入 CS，但是如果在 fast path 遇到其它 process 的 contention，則 worst case 仍然是 $O(n)$；之後 Anderson and Kim[1]將 fast path 中遭遇 contention 的 time complexity 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 Dijkstra's[4]、Knuth's[6]、Eisenburg-McGuire's[5]、Lamport's bakery[8]和 Burns'[2] mutual exclusion algorithms 為例，實際演練 generic approach，得到大幅降低 shared memory contention 之初步具 local spin 的 algorithm；之後又分析這些 algorithm 運作細節，大幅更改 program structure，更進一步嘗試減少在 exit region 中使用 remote write 叫醒其它 process 的次數到 1 次，這種只釋放特定的一個 local-spinning process 的動作稱之為"focused release"，有了 focused release 後，我們嘗試找到"fast track"，讓被釋放的 process 可省去檢查其它 processes 狀態所產生的一連串 remote memory access 動作而直接進入 CS，所得 algorithm 仍然維持原本 safety property 和 progress property 的正確性。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 processes 在 trying region 等待，有利於本文 algorithms 之執行效率，其中以 Eisenburg-McGuire's[5] 為基礎得到的最佳版本 EM_4 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

本章針對本文中使用的專由名詞、指令、函數，解釋其定義與作用，如下列描述。

(1) 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 D_0之 $flag(i)$ 專屬於process i。

(2) 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。

(3) $permitted(i)$:

The generic approach額外使用了這n個shared variables(稱為1-dimensional permitted bits)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。

(4) $permitted(i, j)$

類似generic approach，但是額外使用了n^2個shared variables(稱為2-dimensional permitted bits), for every i, j, $0 \le i \le n-1$, $0 \le j \le n-1$, writable by i and j, readable by i，每一組$permitted(i, j)$, $0 \le j \le n-1$就位於process i的local shared memory中。當process i因為process j的狀態而需要busy waiting時，會spinning on $permitted(i, j)$，這種spinning亦不會引發memory contention，而process j會在造成其它processes進入busy waiting的因素消失後，用remote write to $permitted(i, j)$方式叫醒i。邏輯定義為：

process *j* 釋放 process *i* 若且唯若 process *i* 因為 process *j* 的狀態而進入 local spin 且等待 process *j* 的釋放。

(5) await *some condition*:

等待 *some condition* 為 *true* 才算完成的指令。

(6) focused release:

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

(7) 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。

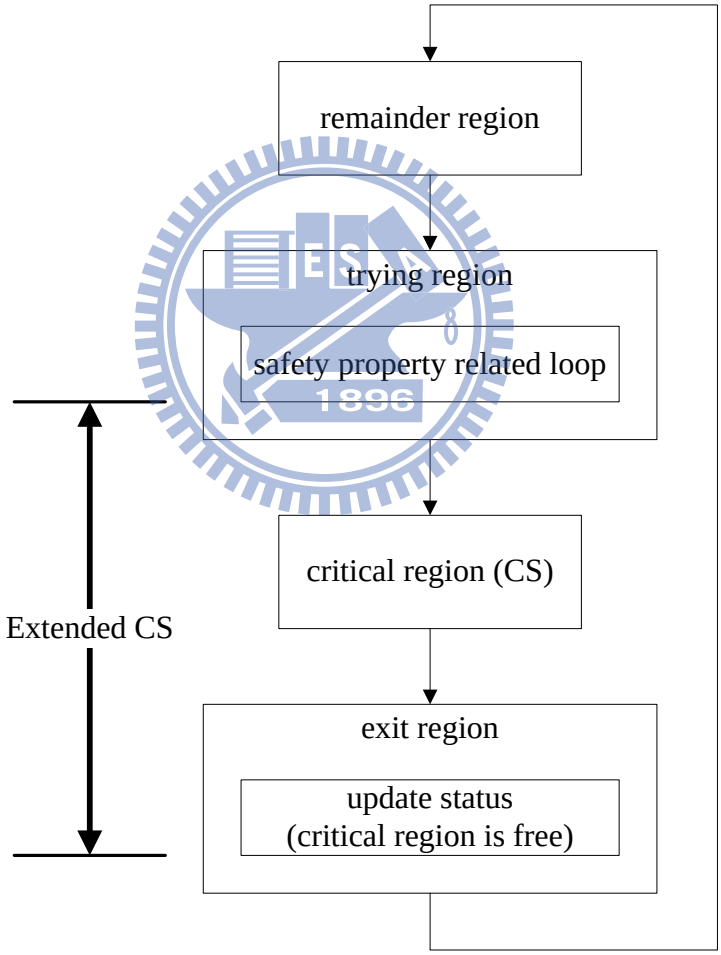


Figure 2: Extended CS

三、 The proposed generic approach

基本理念是要在原有 mutual exclusion 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's[4]、Knuth's[6]、Eisenburg-McGuire's[5]、Lamport's bakery[8] 和 Burns'[2]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。

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 $permitted(i)$，靜待其它某 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。

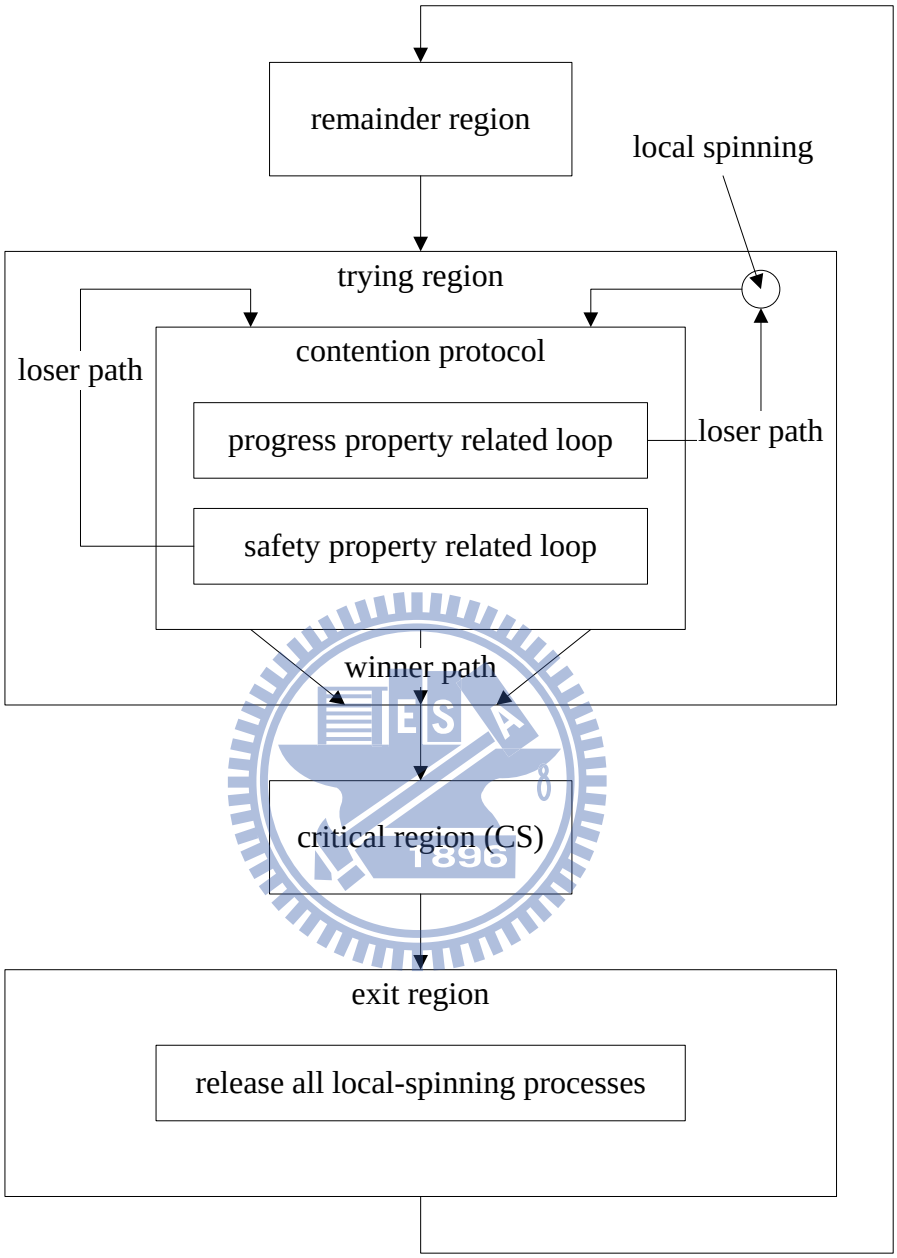


Figure 3: The generic approach

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 都在等待其它 processes 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

Figure 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*(*i*)）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 process *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 process *j* 搶先進 CS，卻因為太早『不作為』而消極性地妨礙到 process *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、 The practice

本章會以 Dijkstra's[4]、Knuth's[6]、Eisenburg-McGuire's[5]、Lamport's bakery[8]和 Burns'[2] mutual exclusion algorithm 為例，分別以額外 2-dimensional permitted bits[1]與 the generic approach 為上述的 algorithms 加上基本的 local spin 機制，並說明兩者之間的優缺點，接著再以使用 the generic approach 加上 local spin 的版本，嘗試找出 focused release 和 fast track。

Dijkstra's[4]、Knuth's[6]和 Eisenburg-McGuire's[5] mutual exclusion algorithm 是非常相似的演算法，Knuth 解決 Dijkstra's[4]中 process 會 starvation 的問題，它具備 $2^{n-1}-1$ bounded waiting 的性質，之後 Eisenburg and McGuire 將 Knuth's[6]更改進到具備 $n-1$ bounded waiting 的性質，這個演進過程中 code 的變化影響到 2-dimensional permitted bits 的適用性，以及在使用 the generic approach 之後，是否容易獲得 focused release 和 fast track。

Lamport's bakery[8]和 Burns'[2] mutual exclusion algorithm 的特點為只使用 one-writer/multiple-reader registers，其中 Lamport's bakery[8]具備 first come fist served 的性質，而 Burns'[2]卻有 starvation 的問題，相同地，有沒有 bounded waiting 的性質會影響 local spin 機制的選擇，以及是否容易獲得 focused release 和 fast track。除此之外，這兩個演算法的 S-loop 也可以加上 local spin 而不會引發 dead lock。綜合所有的演算法，本文展示的版本如 Table 1 所列。

	original version	local spin by using 2-dimensional permitted bits	local spin by the generic approach	adding focused release	adding focused release & fast track
Dijkstra	D_0	D_1	D_2		
Knuth	K_0	K_1 (Dead Lock)	K_2	K_3	
Eisenberg & McGuire	EM_0	(Dead Lock)	EM_2	EM_3	EM_4
Lamport's bakery	L_0	L_1	L_2	L_3	
Burns	B_0	B_1			

Table 1: Algorithm versions

後續的章節將會依照 Table 1 所列的號碼依序說明。

[1]使用 2-dimensional permitted bits 加上 local spin 的方法源於 Gadi Taubenfeld 在 Synchronization Algorithms and Concurrent Programming 一書中提出的 local-spinning bakery mutual exclusion algorithm。

4.1 Dijkstra's mutual exclusion algorithm and its variation

4.1.1 The original version: Algorithm D_0

Original version，如 Algorithm 1，具備 safety property、progress property，但是並沒有具備 starvation freedom，variable *turn* 的目的在於維持 progress property。

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i , $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes

Process i :
(private variable j : integer)

```
R:   ** Remainder Region **
T1:  L:  flag(i) := want − in
T2:      while turn ≠ i do                ▷ P-loop
T3:          if flag(turn) = idle then turn := i fi
T4:      od
T5:      flag(i) := in − cs
T6:      for j ≠ i do                     ▷ S-loop
T7:          if flag(j) = in − cs then goto L fi
T8:      od
C:   ** CS **
E:       flag(i) := idle
R:   ** Remainder Region **
```

Algorithm 1: D_0

Claim：D_0 具備 safety property。

Proof by contradiction：令 E_y^x 表示 process x 完成演算法中 line y 的事件，$t(E_y^x)$ 表示事件發生的時間。假設有 process α 和 β 同時進入 CS，可以得到下列事件時間的先後順序：

(1) $t(E_{T5}^{\alpha}) < t(E_{T6-T8}^{\alpha})$，因為這兩個事件是同一個 process 所執行。

(2) $t(E_{T5}^{\beta}) < t(E_{T6-T8}^{\beta})$，因為這兩個事件是同一個 process 所執行。

(3) $t(E_{T6-T8}^{\beta}) < t(E_{T5}^{\alpha})$，因為 process β 沒有發現 process α 的狀態為 $in-cs$ 而進入 CS。

(4) $t(E_{T6-T8}^{\alpha}) < t(E_{T5}^{\beta})$，因為 process α 沒有發現 process β 的狀態為 $in-cs$ 而進入 CS。

由(2)、(3)、(4)得到 $t(E_{T6-T8}^{\alpha}) < t(E_{T5}^{\alpha})$，與(1)發生矛盾。

Claim：D_0 具備 progress property。

Proof by contradiction：假設在某時刻 S 無任何 process 在 CS 而且已有 process 在 trying region，在 S 之後無任何 process 可進 CS。若 *turn* 值所表示的 process 的狀態為 *idle*，那麼某個 process 在 line T2-T4 進行 *turn* 值的測試時，會執行 $turn := i$；若 *turn* 值所表示的 process 的狀態為 non-*idle*，表示有另外一個 process 曾經執行過 $turn := i$，這表示在 S 之後存在某個時刻 S1，在 S1 之後 *turn* 值所代表的 process 一定是 contender，也就是說在 S1 之後 $flag(turn) \neq idle$ 恆定。在 S1 之後，當 process 通過 line T2-T4 到 line T6-T8 時，因為沒有任何 process 可進入 CS，則必然因為某個 process 的狀態為 $in-cs$ 而回到 line T1 重新測試 *turn* 值，此時 process 若發現 *turn* 值為自己的 index 就可以直接通過 line T2-T4 的 loop 而到 line T6-T8，反之則會因為 $flag(turn) \neq idle$ 而停留在 line T2-T4，依此循環。因為 processes 數量是有限個，不失一般性假設在 S1 之後存在某個時刻 S2，從 S2 之後所有的 processes 都在 remainder region 或 trying region 中，且沒有任何 process 再進入 trying region 中，則存在某時刻 S3，從 S3 之後 *turn* 值不再有任何變動。所以在 S3 之後除了 *turn* 值所代表的 process 可以前進到 line T6-T8 之外，其餘 processes 都將停留在 line T2-T4，而這些 processes 的狀態都將停留在 $want-in$，所以存在一個 process 可以通過 line T6-T8 進入 CS，發生矛盾。

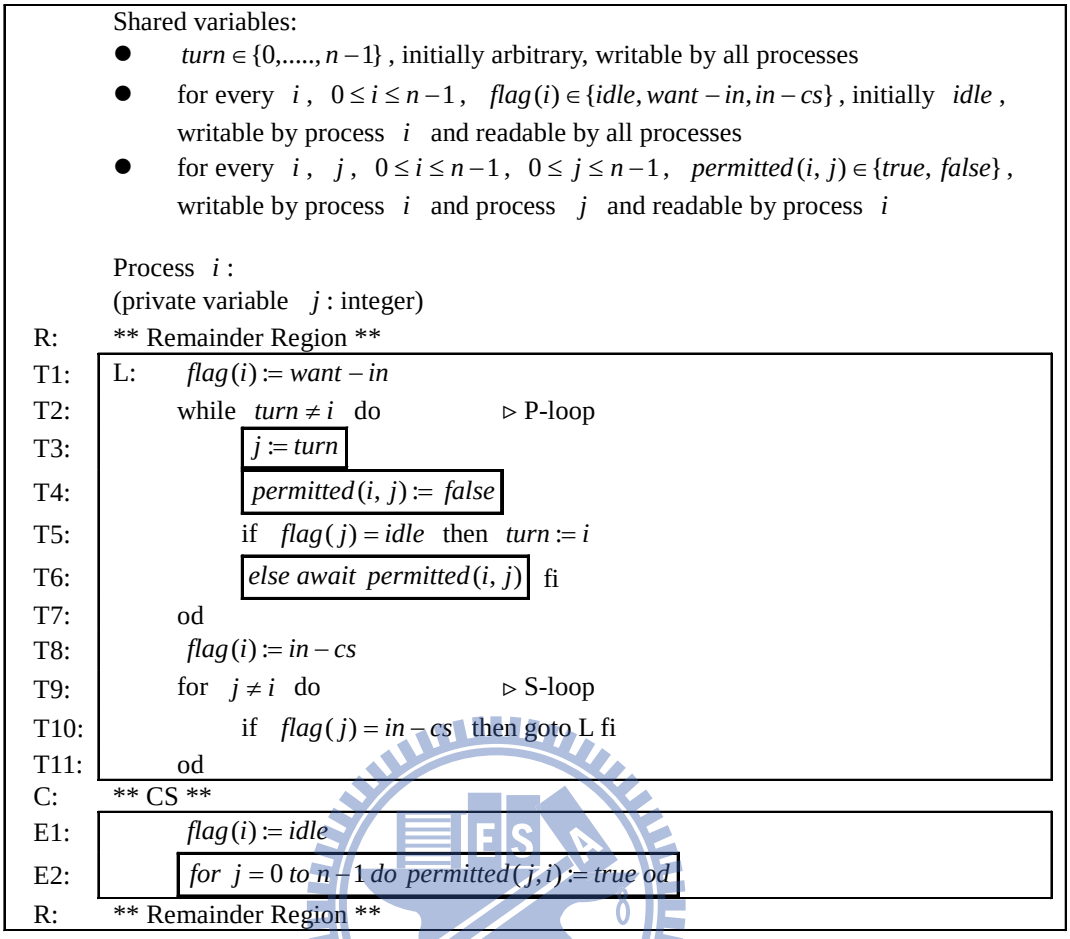
由上述的證明過程可知 line T2-T4 為 P-loop，而 line T6-T8 為 S-loop，清楚分辨出 algorithm 中的 P-loop 與 S-loop 之後，便可以著手加入 local spin 機制。

4.1.2 Adding local spin to D_0 by using 2-dimensional permitted bits: Algorithm D_1

2-dimemsion permitted bits 的邏輯性跟 generic approach 非常相似，依照相同的概念可以得到此版本，如 Algorithm 2。因為 processes 在 spinning on $permitted(i, j)$ 時，j 值不盡然相同，所以反過來在釋放時，當然就不一定會釋放所有的 processes。以 5-processes 為例所形成的 5×5 *permitted bits* 陣列表格來說明這個現象，如 Table 2。

	permitted(1,2)	*permitted*(1,3)	*permitted*(1,4)	*permitted*(1,5)
permitted(2,1)		*permitted*(2,3)	*permitted*(2,4)	*permitted*(2,5)
permitted(3,1)	*permitted*(3,2)		*permitted*(3,4)	*permitted*(3,5)
permitted(4,1)	*permitted*(4,2)	*permitted*(4,3)		*permitted*(4,5)
permitted(5,1)	*permitted*(5,2)	*permitted*(5,3)	*permitted*(5,4)	

Table 2: 5×5 *permitted bits*



Algorithm 2: D_1

Table 2 中方框包圍的 *permitted*(*i*, *j*) 表示 process *i* 在等待 process *j* 的釋放，process 2 在 exit region 會釋放所有 *permitted*(*x*,2)，很顯然因為這個 algorithm 的 *turn* 值在 line T5 會隨時變動的特性，使得 process 2 在釋放時只會釋放 process 1 和 process 4，這會有一個很大的疑慮，這些被釋放的 processes 真的可以進入 CS 而不會造成 dead lock 嗎？證明如下。

Claim：D_1 具備 progress property。

Proof：區分為兩個 case 來討論，

(1) 在 process α 跳出 P-loop 之後到 process α 完成執行 line E2 之間，*turn* 值維持為 α 不再變動：

在這個 case 可以發現當 processes 被釋放時，*turn* 值所代表的 process α's status 已為 *idle* ，依照 D_0 的 progress property，被釋放的 processes 中必然有一個可以進入 CS。

(2) 在 process α 跳出 P-loop 之後到 process α 完成執行 line E2 之間，*turn* 值

不為 α ：

這個 case 可以發現既然 *turn* 值不是 α ，那就表示存在至少一個 process 在 process α 跳出 P-loop 之後執行了 line T5 的 $turn := i$ ，這麼最後一個執行 $turn := i$ 的 process β 既不會進行 local spin 但也不會進入 CS，只要當 process α 在執行 line E1 之後，process β 就可以立即進入 CS，則 process α 有無釋放任何 processes 都無傷 progress property。

目前我們發現 2-dimensional permitted bits 的版本需要注意被釋放的 processes 是否真的能夠進入 CS，因為並不是所有的 processes 都會被釋放，但是使用 generic approach 加上 local spin 機制時並沒有這個疑慮。

4.1.3 Adding local spin to D_0 by the generic approach: Algorithm D_2

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i , $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes
- for every i , $0 \le i \le n-1$, $permitted(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer)

```
R:    ** Remainder Region **
T1:   L:  flag(i) := want − in
T2:       while turn ≠ i do                    ▷ P-loop
T3:             permitted(i) := false
T4:             if flag(turn) = idle then turn := i
T5:             else await permitted(i) fi
T6:       od
T7:       flag(i) := in − cs
T8:       for j ≠ i do                         ▷ S-loop
T9:             if flag(j) = in − cs then goto L fi
T10:      od
C:    ** CS **
E1:       flag(i) := idle
E2:       for j = 0 to n−1 do permitted(j) := true od
R:    ** Remainder Region **
```

Algorithm 3: D_2

依據 D_0 在 safety property 與 progress property 的證明，得知在 trying region 中只有一個 P-loop，在大量 processes 競爭 CS 時，該 P-loop 中 processes 不斷反覆讀取 *turn* 值與 $flag(turn)$ 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

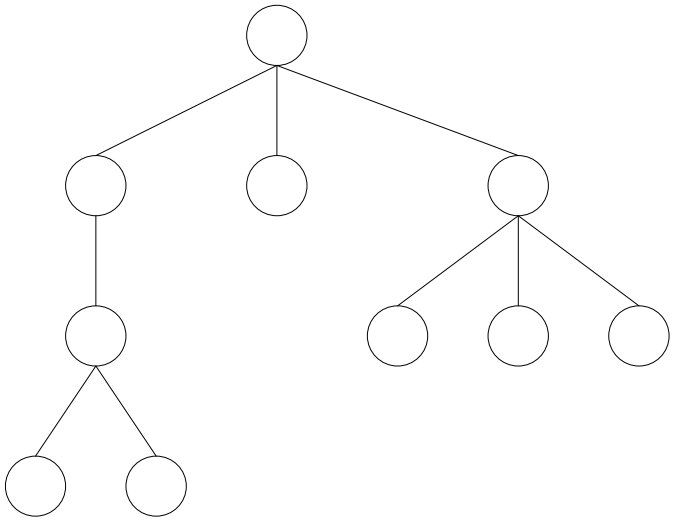
會造成 dead lock。由於 local spin 是額外加上 variables 和 spin 動作，並沒有變動與 safety property 有相關的 code，所以仍維持原有的 safety property，實際上 generic approach 所得之 algorithm D_2，如 Algorithm 3，只在原始的 algorithm D_0 加上額外三行 code（以方框標識）。

在加上 local spin 之後比較 D_0 與 D_2 行為，基本上驗證了前面 the generic approach 章節所述，參考 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*(*turn*) 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 D_0 的 progress property 證明即可以輕易仿製成 D_2 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 line E2 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。

在與 2-dimensional permitted bits 相較之下，很顯然 generic approach 造成的 shared memory contention 是比較嚴重的，因為在 exit region 釋放的 processes 數量在 2-dimensional permitted bits 模式會≤在 generic approach 模式，我們用 tree 來呈現這兩者的不同，若 process α 在等待 process β 的釋放，那麼 process β 為 process α 的 parent node，依照這樣的規則可以畫出示意圖 Figure 4，root 就代表可以進入 CS 的 process，當 root 釋放 children 時，這些 processes 會競爭成為 root，沒有成為 root 的 processes 將會成為新 root 的 children，或是 level 更深的位置，利用 Figure 4 我們也可以發現 2-dimensional permitted bits 中 level 越深的 processes 比在 D_0 中更容易發生 starvation，雖然降低了 shared memory contention，但是某些 processes 得承擔更嚴重的 starvation，而 generic approach 形成的 tree 只有 2 levels，釋放所有 local-spinning processes 會讓整體系統的行為回復到跟 D_0 一樣。

另外值得注意的是在 exit region 中釋放 processes 的動作是在 extended CS 之外，也就是說有可能會有兩個以上的 processes 在做釋放的動作，但是這個並沒有影響 algorithm 的正確性，因為任何一個被釋放的 process 都必須再依照原本的 algorithm 執行，為 algorithm 加上 local spin 機制時只擔心 processes 沒有被釋放，但是不擔心被重複釋放或同時釋放，而 focused release 可以減少這種重複性釋放。

2-dimensional permitted bits



The generic approach

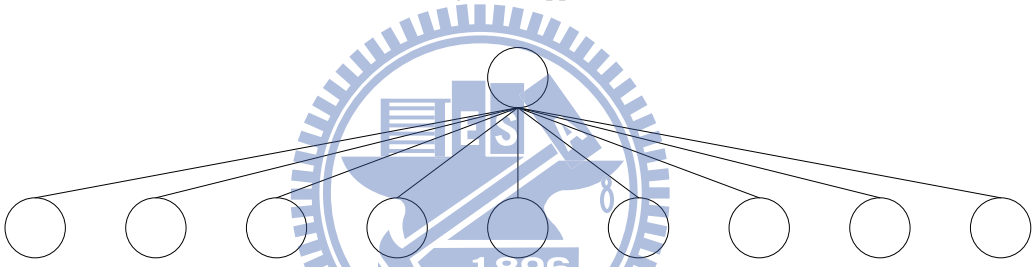


Figure 4: The waiting-release relation

4.2 Knuth's mutual exclusion algorithm and its variation

4.2.1 The original version: Algorithm K_0

Knuth's[6]和 Dijkstra's[4]最大的不同點在於它將 $turn := i$ 這個動作移到了 extended CS 中，如 Algorithm 4，這不但減少了 processes 對 *turn* 複寫的競爭，也確保 *turn* 值在任何的瞬間只有一個 process 可以變更其值。在 safety property 的設計上維持與 Dijkstra's[4]一致的做法，而在維持 progress property 上為兼顧 bounded waiting 的特性就有不同的設計。

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i , $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes

Process i :
(private variable j : integer)

```
R:   ** Remainder Region **
T1:  repeat
T2:        flag(i) := want – in
T3:        j := turn
T4:        while  j ≠ i  do                ▷ P-loop
T5:              if  flag( j) ≠ idle  then  j := turn
T6:              else  j := j +1 mod n  fi
T7:        od
T8:        flag(i) := in – cs
T9:        j := 0
T10:       while ( j < n ) and ( j = i  or  flag( j) ≠ in – cs ) do  j := j +1  od
T11: until ( j ≥ n )                       ▷ S-loop
T12: turn := i                             ▷ extended CS begin
C:   ** CS **
E1:  turn := i +1 mod n
E2:  flag(i) := idle                       ▷ extended CS end
R:   ** Remainder Region **
```

Algorithm 4: K_0

Claim：K_0 具備 progress property。

Proof by contradiction：假設在某時刻 S 無任何 process 在 CS 而且已有 process 在 trying region，在 S 之後無任何 process 可進 CS。因為改變 *turn* 值的動作 line T12 及 E1 都在 extended CS 之中，所以在 S 之後 *turn* 值不會有任何變動。因為 processes 數量是有限個，不失一般性假設在 S 之後存在某個時刻 S1，在 S1 之後所有的 processes 都在 remainder region 或 trying region 中，且沒有任何 process 再進入 trying region 中，則存在一個 index 最靠近 *turn* 值（包含等於）的 process α 可以順利通過 line T4-T7 的測試，其

餘的 processes 都會因為 process α 而在 line T4、T5 之間不斷重複執行（因為 *turn* 值沒有變化），而這些 processes 的狀態都是 *want* – *in*，所以 process α 可以通過 line T10-T11 的測試而進入 CS，發生矛盾。

欲證明 K_0 具備 $2^{n-1}-1$ bounded waiting 的性質，先觀察出幾個 algorithm 的特性：

(1) 當 *turn* 值所代表的 process 狀態是 *idle* 時，存在一個 execution sequence 讓所有的 processes 都可以通過 P-loop，並停留在將執行 line T8 而尚未執行的階段，此時第一個執行 line T8 並通過 S-loop 的 process 將可以進入 CS，之後的 processes 只要待先前進入 CS 的 process 執行 line E2 之後，再繼續執行 S-loop 的驗證即可進入 CS，而不需要成為 S-loop 中的 loser 回到 P-loop 重新執行。

(2) 假設有三個 non-*idle* processes α、β、γ，其中 $\alpha<\beta<\gamma$，且 $\gamma=\beta+1$，process β 進入 CS 且 *turn* 值為 β，則 process α 會比 process γ 先進入 CS 唯一的 execute sequence 就是依照(1)，process α 已經通過 P-loop 而停留在將執行 line T8 而未執行的階段，並且在 process β 執行 line E2 後，立即執行 line T8 且順利通過 S-loop，即使 process β 在 line E1 將 *turn* 值改為 γ 都無法阻擋 process α 搶先 process γ 之前進入 CS。

(3) 在多個 processes 的 status 為 non-*idle* 的狀況，於 line E1 和 P-loop 的合作下，同一個 process 不可能連續進入 CS 兩次。

(4) 若兩個 non-*idle* processes β、γ，其中 $\beta<\gamma$，當 process β 在 process γ 之前曾經進入 CS 兩次以上，則依照(2)和(3)，在 process β 兩次進入 CS 之間必然存在一個 process α 進入 CS，其中 $\alpha<\beta$ 或 $\alpha>\gamma$。

Claim：K_0 具備 $2^{n-1}-1$ bounded waiting 屬性。

Proof：假設 process α 被其它 processes 盡可能的 bypass，不失一般性假設 $\alpha=n-1$，再假設 $\beta=0$，則在 bypass process α 的 execution sequence 中依照觀察(4)，process β 只可能進入 CS 一次，將 process β 進入 CS 的唯一一次視為分水嶺將 execution sequence 拆成兩段，在 process β 進入 CS 之前可以視做 $n-1$ 個 processes 在執行，而 process β 執行 line E1 將 *turn* 值 assign 成 1 的時候，若 process 1 此時的狀態若恰為 *idle*，則又可再次視為 $n-1$ 個 processes 在執行，若 $f(n)$ 表示 n processes 中某個 process 被 bypass 的最大次數，可以列出 recursive function 為 $f(n)=2f(n-1)+1$ and $f(1)=0$，得 $f(n)=2^{n-1}-1$。

同樣地，依據證明的過程可以清楚分辨 P-loop 與 S-loop，這有助於加上 local spin

機制。

4.2.2 Adding local spin to K_0 by using 2-dimensional permitted bits: Algorithm K_1

```
        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 , j , 0 ≤ i ≤ n−1 , 0 ≤ j ≤ n−1 , permitted(i, j) ∈ {true, false} ,
          writable by process i and process j and readable by process i

        Process i :
        (private variable j : integer)
R:      ** Remainder Region **
T1:     repeat
T2:         flag(i) := want − in
T3:         j := turn
T4:         while j ≠ i do                    ▷ P-loop
T5:             permitted(i, j) := false
T6:             if flag( j) ≠ idle then
T7:                 await permitted(i, j)
T8:                 j := turn
T9:             else j := j +1 mod n fi
T10:        od
T11:        flag(i) := in − cs
T12:        j := 0
T13:        while ( j < n ) and ( j = i or flag( j) ≠ in − cs ) do j := j +1 od
T14:    until ( j ≥ n )                      ▷ S-loop
T15:    turn := i                             ▷ extended CS begin
C:      ** CS **
E1:     turn := i +1 mod n
E2:     flag(i) := idle                       ▷ extended CS end
E3:     for j = 0 to n −1 do permitted( j,i) := true od
R:      ** Remainder Region **
```

Algorithm 5: K_1

Knuth's[6]的特點在於它解決了 Dijkstra's[4]中 starvation 的問題，和 Dijkstra's[4]相較之下，*turn* 值只在 extended CS 中被改變的特性，反而造成不適用 2-dimensional permitted bits 的方式來加上 local spin 機制。

如 Algorithm 5，此版本會引發 dead lock，簡略的執行順序如下：

(1) *turn* 值為 0，process 0 為 *idle* 。

(2) process 2 發動通過 S-loop 停在要執行 line T15 但尚未執行。

(3) process 1 發動，但是只執行完 line T2 就暫時停止。

(4) process 3 發動並 local spin on $permitted(3,1)$ 。

(5) process 2 執行 line T15 之後進入 CS，在 exit region 執行 line E1，此時 *turn* 值為 3。

(6) process 1 繼續執行並 local spin on $permitted(1,3)$ 。

(7) process 2 執行 line E3 釋放 $permitted(j,2)$， $j=0$ to $n-1$，但是 process 1、3 在互相等待對方的釋放，造成 dead lock。

由此反例可得知 2-dimensional permitted bits 的方式並不是通用的法則。

4.2.3 Adding local spin to K_0 by the generic approach: Algorithm K_2

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i , $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes
- for every i , $0 \le i \le n-1$, $permitted(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer)

```
R:   ** Remainder Region **
T1:  repeat
T2:      flag(i) := want − in
T3:      j := turn
T4:      while j ≠ i do                ▷ P-loop
T5:          permitted(i) := false
T6:          if flag(j) ≠ idle then
T7:              await permitted(i)
T8:              j := turn
T9:          else j := j + 1 mod n fi
T10:     od
T11:     flag(i) := in − cs
T12:     j := 0
T13:     while ( j < n ) and ( j = i or flag(j) ≠ in − cs ) do j := j + 1 od
T14: until ( j ≥ n )                   ▷ S-loop
T15: turn := i                         ▷ extended CS begin
C:   ** CS **
E1:  turn := i + 1 mod n
E2:  flag(i) := idle                   ▷ extended CS end
E3:  for j = 0 to n − 1 do permitted(j) := true od
R:   ** Remainder Region **
```

Algorithm 6: K_2

K_2 依照 generic approach 的精神維持了 progress property，如 Algorithm 6，但是在 exit region 中 line E3 有 linear remote write 的行為，利用 exit region 中 line E1 更改 *turn* 值的行為，來找出只要釋放一個 process 的可能性，如下節所述。

4.2.4 Adding focused release to K_2: Algorithm K_3

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i , $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes
- for every i , $0 \le i \le n-1$, $permitted(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer, k : integer)

```
R:    ** Remainder Region **
T1:   repeat
T2:       flag(i) := want - in
T3:       j := turn
T4:       while  j ≠ i  do                    ▷ P-loop
T5:             permitted(i) := false
T6:             if  flag(j) ≠ idle  then
T7:                   await permitted(i)
T8:                   j := turn
T9:             else  j := j + 1 mod n  fi
T10:      od
T11:      flag(i) := in - cs
T12:      j := 0
T13:      while ( j < n ) and ( j = i  or  flag(j) ≠ in - cs ) do  j := j + 1  od
T14:  until ( j ≥ n )                         ▷ S-loop
T15:  turn := i                               ▷ extended CS begin
C:    ** CS **
E1:   turn := i + 1 mod n
E2:   if  flag(turn) ≠ idle then  j := turn
E3:   else  j := -1  fi
E4:   flag(i) := idle                         ▷ extended CS end
E5:   if  j = -1 then  for k = 0 to n - 1 do  permitted(k) := true  od
E6:   else  permitted(j) := true  fi          ▷ focused release
R:    ** Remainder Region **
```

Algorithm 7: K_3

從 original version 可以知道 *turn* 值所代表的 process 有較大的優勢可以通過 P-loop 而到達 S-loop，雖然不一定能通過 S-loop，但可以確保至少有一個 process 可以到達 S-loop。從這一點引發 focused release 的想法是，既然在 exit region 將 *turn* 值 assign 成 $\alpha = i + 1 \bmod n$ 的話，那麼如果 process α 's status 是 non-*idle* ，是不是只要釋放 process

α 就可以維持 progress property 呢？我們在 extended CS 中加上判斷 process α 是否為 non-*idle*，如 Algorithm 7，如果條件成立，那麼只要釋放 process α 即可，又因為 *turn* 值只會在 extended CS 中被改變，所以在其中去 read 一次 $flag(turn)$ 值即是 $flag(\alpha)$ 的值，此處必須驗證 progress property 是否因此被破壞。

Claim：K_3 的 focused release 維持 progress property。

Proof：假設 process i 將 *turn* 值 assign 成 $\alpha = i+1 \bmod n$，而且在 extended CS 發現 process α's status 是 non-*idle*，那麼在 process α 被釋放後到下一個 process 進入 CS 之前的這段時間內，依 *turn* 值有無被改變分成兩個 case 來看，（1）如果 *turn* 值不變，那麼 process α 必然為唯一進入 CS 的 process，（2）如果 *turn* 值改變，那就表示存在另外一個 process β 已經進入 extended CS 執行了 line T15，那麼 process β 也會進入 CS，所以 progress property 沒有被破壞。

後續嘗試在 K_3 加上 fast track，但是事實上是很困難的，從 K_0 知道雖然它具備 starvation freedom，但是 bounded waiting 的 upper bound 是 $2^{n-1}-1$，K_0 存在一個現象：*turn* 值代表的 process 即使是 non-*idle*，它也未必能夠進入 CS，也就是說缺少足夠的資訊來分辨哪一個 process 是真正能夠進入 CS 的 process，那麼 fast track 的條件就很難發現，但是具備 $n-1$ bounded waiting 的 Eisenburg-McGuire's[5] mutual exclusion algorithm 就很容易找到 fast track 的條件。

4.3 Eisenburg-McGuire's mutual exclusion algorithm and its variation

4.3.1 The original version: Algorithm EM_0

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i, $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes

Process i :
(private variable j : integer)

```
R:   ** Remainder Region **
T1:  repeat
T2:      flag(i) := want − in
T3:      j := turn
T4:      while j ≠ i do                    ▷ P-loop
T5:          if flag(j) ≠ idle then j := turn
T6:          else j := j + 1 mod n fi
T7:      od
T8:      flag(i) := in − cs
T9:      j := 0
T10:     while ( j < n ) and ( j = i or flag(j) ≠ in − cs ) do j := j + 1 od
T11: until ( j ≥ n ) and ( turn = i or flag(turn) = idle )   ▷ S-loop
T12: turn := i                             ▷ extended CS begin
C:   ** CS **
E1:  j := i + 1 mod n
E2:  while flag(j) = idle do j := j + 1 mod n od
E3:  turn := j
E4:  flag(i) := idle                       ▷ extended CS end
R:   ** Remainder Region **
```

Algorithm 8: EM_0

Eisenburg-McGuire 在 1972 年所提 mutual exclusion algorithm[5] (亦見 Operating System Concepts，5th ed.，Silberschatz & Galvin，p.201)，如 Algorithm 8，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 index(此動作受到 extended CS 保護)，而 *turn* 值所代表的 process 只要是 non-*idle* 就是下一個能優先進入 CS 的 process。在 safety property 的設計仍然保留與 Dijkstra's[4]一樣的機制，而 progress property 的證明也與 Knuth's[6]雷同。

Lemma 4.3.1：EM_0 中，process α 在 exit region 中找到的下一個 non-*idle* process β 將必然繼 process α 之後進入 CS。

Proof by contradiction：假設有 process γ 搶先在 process β 之前繼 process α 之後進入 CS，可以得到下列事件發生順序的關係，

(1) $t(E_{T2}^{\beta}) < t(E_{E2}^{\alpha})$，process α 在 exit region 中找到下一個 non-*idle* process 為 process β。

(2) $t(E_{E2}^{\alpha}) < t(E_{E3}^{\alpha}) < t(E_{E4}^{\alpha})$，這是同一個 process 所執行。

(3) $t(E_{E4}^{\alpha}) < t(E_{T11}^{\gamma})$，process γ 繼 process α 之後進入 CS，則 process γ 在 line T11 的 $j \geq n$ 測試條件必為 true。

(4) 由(2)、(3)得 $t(E_{E3}^{\alpha}) < t(E_{T11}^{\gamma})$，process γ 在 line T11 執行時 $turn = \beta$，故 $turn = i$ 的測試必為 false。

(5) $t(E_{T11}^{\gamma}) < t(E_{T2}^{\beta})$，因為 process γ 在 line T11 對 $turn = i$ 的測試為 false，則 $flag(turn) = idle$ 的測試必為 true，方可進入 CS。

由(1)、(2)、(3)得 $t(E_{T2}^{\beta}) < t(E_{T11}^{\gamma})$，與(5)發生矛盾。

Lemma 4.3.1 掌握了 algorithm EM_0 的 processes 之間溝通交接進 CS 的規律性，不但可以證明 $n-1$ bounded waiting，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。

Claim：EM_0 具備 $n-1$ bounded waiting 的性質。

Proof by contradiction：欲證 $n-1$ bounded waiting 等價於證明 1 bounded bypass。假設兩個 non-*idle* processes α、β 欲進入 CS，在 process β 進入 CS 之前，process α 進入 CS 兩次。在 process α 第一次進入 CS 之後，於 exit region 尋找下一個 non-*idle* process 時必然找到第三個 non-*idle* process γ，並將 *turn* 值改為 γ，不失一般性假設 $\alpha < \gamma < \beta$，則在 process γ 進入 CS 之後到 process α 第二次進入 CS 之前，*turn* 值的變化過程中必然曾經等於 β，但是卻跳過 process β 而讓 process α 第二次進入 CS，這與 Lemma 4.3.1 矛盾。

在 EM_0 上使用 2-dimensional permitted bits 的方式加上 local spin 會與 K_1 一樣發生 dead lock。在 EM_0 上使用 generic approach 可輕易地得到初步的 local spin algorithm EM_2，如 Algorithm 9，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_2 為實例詳述如何加入 focused release 與 fast track 兩項功效如 4.3.2、4.3.3 節所述。

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i , $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes
- for every i , $0 \le i \le n-1$, $permitted(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer)

```
R:   ** Remainder Region **
T1:  repeat
T2:      flag(i) := want - in
T3:      j := turn
T4:      while j ≠ i do                ▷ P-loop
T5:          permitted(i) := false
T6:          if flag(j) ≠ idle then
T7:              await permitted(i)
T8:              j := turn
T9:          else j := j + 1 mod n fi
T10:     od
T11:     flag(i) := in - cs
T12:     j := 0
T13:     while ( j < n ) and ( j = i or flag(j) ≠ in - cs ) do j := j + 1 od
T14: until ( j ≥ n ) and ( turn = i or flag(turn) = idle )   ▷ S-loop
T15: turn := i                         ▷ extended CS begin
C:   ** CS **
E1:  j := i + 1 mod n
E2:  while flag(j) = idle do j := j + 1 mod n od
E3:  turn := j
E4:  flag(i) := idle                   ▷ extended CS end
E5:  for j = 0 to n - 1 do permitted(j) := true od
R:   ** Remainder Region **
```

Algorithm 9: EM_2

4.3.2 Adding focused release to EM_2: Algorithm EM_3

以 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，我們做 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 所有其它 local-spinning processes），依據 Lemma 4.3.1，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；EM_3，如 Algorithm 10，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 做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 [10]有這種 algorithms。

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i , $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes
- for every i , $0 \le i \le n-1$, $permitted(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer, k : integer)

```
R:    ** Remainder Region **
T1:   repeat
T2:       flag(i) := want - in
T3:       j := turn
T4:       while j ≠ i do                          ▷ P-loop
T5:             permitted(i) := false
T6:             if flag(j) ≠ idle then
T7:                   await permitted(i)
T8:                   j := turn
T9:             else j := j + 1 mod n fi
T10:      od
T11:      flag(i) := in - cs
T12:      j := 0
T13:      while ( j < n ) and ( j = i or flag(j) ≠ in - cs ) do j := j + 1 od
T14:  until ( j ≥ n ) and ( turn = i or flag(turn) = idle )      ▷ S-loop
T15:  turn := i                                   ▷ extended CS begin
C:    ** CS **
E1:   j := i + 1 mod n
E2:   while flag(j) = idle do j := j + 1 mod n od
E3:   turn := j
E4:   flag(i) := idle                             ▷ extended CS end
E5:   if j = i then for k = 0 to n - 1 do permitted(k) := true od
E6:   else permitted(j) := true fi                ▷ focused release
R:    ** Remainder Region **
```

Algorithm 10: EM_3

4.3.3 Adding fast track to EM_3: Algorithm EM_4

Shared variables:

- $turn \in \{0,.....,n-1\}$, initially arbitrary, writable by all processes
- for every i, $0 \le i \le n-1$, $flag(i) \in \{idle, want-in, in-cs\}$, initially $idle$, writable by process i and readable by all processes
- for every i, $0 \le i \le n-1$, $permitted(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer, k : integer, $spin-wake-up$: boolean)

```
R:   ** Remainder Region **
T1:  spin − wake − up := false
T2:  repeat
T3:      flag(i) := want − in
T4:      j := turn
T5:      while j ≠ i do                          ▷ P-loop
T6:          permitted(i) := false
T7:          spin − wake − up := false
T8:          if flag(j) ≠ idle then
T9:              await permitted(i)
T10:             spin − wake − up := true
T11:             j := turn
T12:         else j := j + 1 mod n fi
T13:     od
T14:     flag(i) := in − cs
T15:     if spin − wake − up then goto CS fi     ▷ fast track
T16:     j := 0
T17:     while ( j < n ) and ( j = i or flag(j) ≠ in − cs ) do j := j + 1 od
T18: until ( j ≥ n ) and ( turn = i or flag(turn) = idle )   ▷ S-loop
T19: turn := i                                   ▷ extended CS begin
C:   ** CS **
E1:  j := i + 1 mod n
E2:  while flag(j) = idle do j := j + 1 mod n od
E3:  turn := j
E4:  flag(i) := idle                             ▷ extended CS end
E5:  if j = i then for k = 0 to n − 1 do permitted(k) := true od
E6:  else permitted(j) := true fi                ▷ focused release
R:   ** Remainder Region **
```

Algorithm 11: EM_4

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's mutual exclusion algorithm[5]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（S-loop，原本目的在於測知其他 processes 之狀態）。以這樣的構想，在 fast track 的設計上利用額外的 private variable 由 process 自身記錄執行軌跡，如 Algorithm 11，EM_4 中的 $spin-wake-up$ 。

Claim：EM_4 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（line T17、T18），直接進 CS。假設 β 在 fast track 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 4.3.1 所述之前提，因此除了 β 外無其它 process 可以進 CS。

版本 EM_4 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。

4.4、4.5 節將繼續介紹兩個 one-writer/multiple-reader algorithm 如何加上 local spin 的機制，除了適用 2-dimensional permitted bits 之外，也發現這兩個 algorithm 中的 S-loop 亦可加上 local spin。

4.4 Lamport's bakery mutual exclusion algorithm and its variation

4.4.1 The original version: Algorithm L_0

Shared variables:

- for every i, $0 \le i \le n-1$, $choosing(i) \in \{true, false\}$, initially $false$, writable by process i and readable by all processes
- for every i, $0 \le i \le n-1$, $number(i) \in N$, initially 0, writable by process i and readable by all processes

Process i :
(private variable j : integer)

```
R:  ** Remainder Region **
T1: choosing(i) := true
T2: number(i) := 1 + max{number(j) | 0 ≤ j ≤ n-1, j ≠ i}
T3: choosing(i) := false
T4: for j = 0 to n-1, j ≠ i do
T5:     while ( choosing(j) ) do nothing od
T6:     while ( number(j) ≠ 0 and (number(i), i) > (number(j), j) ) do nothing od
T7: od
C:  ** CS **
E:  number(i) := 0
R:  ** Remainder Region **
```

Algorithm 12: L_0

Lamport's bakery[8] mutual exclusion algorithm，如 Algorithm 12，在 line T1-T3 有一個取號碼牌的動作，這個部分稱之為 doorway，在 process 通過 doorway 之後便具備了 first come first served 的特性，我們可以觀察到 original version 中原本就有兩個 waiting 的地方，一個是 line T5 處在 waiting 其它 processes 將號碼取完，這個部份可以說比較沒有規則可言，任何 process 都會造成其它 processes 必須 waiting，另一個則是 line T6 處在 waiting 號碼比自己小的 processes 離開 CS。假設一個情境：若 process 0 正在 CS 中，那麼就會造成所有的 processes 都集中在 waiting process 0，此時造成的 shared memory contention 將非常嚴重。

定義 $(number(x), x)$ 函數：

$(number(i), i) > (number(j), j)$

$\Leftrightarrow number(i) > number(j)$ *or* $(number(i) = number(j)$ *and* $i > j)$

Claim：L_0 具備 safety property。

Proof by contradiction：假設兩個 process α 與 β 同時進入 CS。則因為 line T6 可同時獲得 $(number(\alpha), \alpha) < (number(\beta), \beta)$ 以及 $(number(\alpha), \alpha) > (number(\beta), \beta)$，發生矛盾。

Claim：L_0 具備 progress property。

Proof：在通過 doorway 之後，$(number(x), x)$ 最小的 process 即可進入 CS。

由上述的證明過程可以發現 L_0 只有一個 loop（line T4-T7），此 loop 既是 P-loop 也是 S-loop，這是少見的兩者合而唯一的例子。在前面的章節曾經描述過 S-loop 加上 local spin 機制可能會有 dead lock 的風險；以 Knuth's[6]和 Eisenburg-McGuire's[5]為例，使用 2-dimensional permitted bits 亦造成 dead lock，但是 Lamport's bakery[8]不但可以在 S-loop 中加上 local spin，亦可以使用 2-dimensional permitted bits 的方式，詳述如後。

4.4.2 Adding local spin to L_0 by using 2-dimensional permitted bits: Algorithm L_1

Shared variables:

- for every i, $0 \le i \le n-1$, $choosing(i) \in \{true, false\}$, initially $false$, writable by process i and readable by all processes
- for every i, $0 \le i \le n-1$, $number(i) \in N$, initially 0, writable by process i and readable by all processes
- for every i, j, $0 \le i \le n-1, 0 \le j \le n-1$, $permitted.ch(i, j) \in \{true, false\}$, writable by process i and process j and readable by process i
- for every i, j, $0 \le i \le n-1, 0 \le j \le n-1$, $permitted.nu(i, j) \in \{true, false\}$, writable by process i and process j and readable by process i

Process i :
(private variable j : integer)

```
R:   ** Remainder Region **
T1:  choosing(i) := true
T2:  number(i) := 1 + max{number(j) | 0 ≤ j ≤ n-1, j ≠ i}
T3:  choosing(i) := false
T4:  for j = 0 to n-1 do permitted.ch(j,i) := true od
T5:  for j = 0 to n-1, j ≠ i do
T6:      permitted.ch(i, j) := false
T7:      if ( choosing(j) ) then await permitted.ch(i, j) fi
T8:      permitted.nu(i, j) := false
T9:      if ( number(j) ≠ 0 and (number(i),i) > (number(j), j) )
T10:     then await permitted.nu(i, j) fi
T11: od
C:   ** CS **
E1:  number(i) := 0
E2:  for j = 0 to n-1 do permitted.nu(j,i) := true od
R:   ** Remainder Region **
```

Algorithm 13: L_1

此版本原由 Gadi Taubenfeld[11]所提出，如 Algorithm 13，依照之前的經驗，得驗

證使用 2-dimension permitted bits 加上 local spin 是否有可能造成 dead lock；因為 original version 中有兩個 waiting 的地方，且 waiting 的條件不同，所以得準備兩組 permitted bits 以區隔成兩個 local spin。第一個 local spin（line T7）是在等待其它 processes 取完號碼牌，很顯然不可能造成 dead lock；第二個 local spin（line T10）是否會造成 dead lock 的驗證如下。

Claim：L_1 中第二個 local spin（line T10）維持 progress property。

Proof：若 process α 進入第二個 local spin，表示 process α 在 line T9 的測試發現存在另外一個 process β，其中 $(number(\alpha),\alpha) > (number(\beta),\beta)$，當 process β 進入 CS 後到 exit region 時，便會釋放 process α，若 process β 也進入 local spin，則表示存在 process γ，其中 $(number(\beta),\beta) > (number(\gamma),\gamma)$，依此類推，依照 original version 的 progress property 可以得知 $(number(x),x)$ 最小的 process 不會進入 local spin，那麼它就會在 exit region 中釋放 $(number(x),x)$ 次小的 process（次小的 process 只會因最小的 process 而進入 local spin），當 $(number(x),x)$ 最小的 process 回到 remainder region 時，$(number(x),x)$ 次小的 process 即成為 $(number(x),x)$ 最小的 process，再依此類推，process α 必然會被釋放而進入 CS。

加上 local spin 之後，上一節提到的情境 process 0 在 CS 中造成非常嚴重的 memory contention 將獲得大幅度的改善。

4.4.3 Adding local spin to L_0 by the generic approach: Algorithm L_2

依照 generic approach 的精神加上 local spin，如 Algorithm 14，觀察 L_2 的行為在第一個 local spin（line T9）的部份，無論哪一個 process 取完號碼之後，都會將所有的 processes 釋放（line T6），這個部份建議採用 2-dimensional permitted bits 的方式，避免增加很多不必要的釋放，造成多餘的動作，若嘗試用 graph 來呈現 processes 之間等待與釋放的關係，這個 graph 的 edges 會沒有規則地將所有的 nodes 連接。

第二個 local spin（line T13）也有類似的情況，但是因為能夠同時到 exit region 做釋放動作（line E2）的 processes 數量遠少於第一個 local spin 的釋放（line T6），所以重複性的釋放並沒有那麼嚴重；一樣可以用 tree 來呈現 processes 之間等待與釋放的關係，參照 Figure 4，不過 L_1 所形成的 tree 有一個額外的特性，就是（1）parent node 的 $(number(x),x)$ 一定比 children 小，（2）$(number(x),x)$ 次小的 process 一定是 root 的 child。在 line E2 釋放的行為需要 linear time，但若能找到 focused release，那麼就可以減少多餘的釋放，事實上只要能找到 $(number(x),x)$ 次小的 process，那麼它就是下一個進入 CS 的 process。

Shared variables:

- for every i, $0 \le i \le n-1$, $choosing(i) \in \{true, false\}$, initially $false$, writable by process i and readable by all processes
- for every i, $0 \le i \le n-1$, $number(i) \in N$, initially 0, writable by process i and readable by all processes
- for every i, $0 \le i \le n-1$, $permitted.ch(i) \in \{true, false\}$, writable by all processes and readable by process i
- for every i, $0 \le i \le n-1$, $permitted.nu(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer)

```
R:   ** Remainder Region **
T1:  permitted.ch(i) := false
T2:  permitted.nu(i) := false
T3:  choosing(i) := true
T4:  number(i) := 1 + max{number(j) | 0 ≤ j ≤ n-1, j ≠ i}
T5:  choosing(i) := false
T6:  for j = 0 to n-1, j ≠ i do permitted.ch(j) := true od
T7:  for j = 0 to n-1, j ≠ i do
T8:      while ( choosing(j) ) do
T9:          await permitted.ch(i)
T10:         permitted.ch(i) := false
T11:     od
T12:     while ( number(j) ≠ 0 and (number(i),i) > (number(j), j) ) do
T13:         await permitted.nu(i)
T14:         permitted.nu(i) := false
T15:     od
T16: od
C:   ** CS **
E1:  number(i) := 0
E2:  for j = 0 to n-1 do permitted.nu(j) := true od
R:   ** Remainder Region **
```

Algorithm 14: L_2

4.4.4 Adding focused release to L_2: Algorithm L_3

找 $(number(x), x)$ 次小的 process 來做 focused release 在概念上是非常清楚的一種做法，如 Algorithm 15，本節提出一個 min function 來找到這個 process，如 Figure 5，首先利用 original version 中 max function 做一次 linear search，看看是不是存在其它 $number(x) \ne 0$ 的 process，如果沒有，則直接釋放所有的 processes，若有，則必然可以找到 $(number(x), x)$ 次小的 process，重點在於進行比較的時候，遇到其它 process 正在取號碼牌時仍然要等待，避免造成誤判。

Shared variables:

- for every i, $0 \le i \le n-1$, $choosing(i) \in \{true, false\}$, initially $false$, writable by process i and readable by all processes
- for every i, $0 \le i \le n-1$, $number(i) \in N$, initially 0, writable by process i and readable by all processes
- for every i, $0 \le i \le n-1$, $permitted.ch(i) \in \{true, false\}$, writable by all processes and readable by process i
- for every i, $0 \le i \le n-1$, $permitted.nu(i) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer, $next$: integer)

```
R:   ** Remainder Region **
T1:  permitted.ch(i) := false
T2:  permitted.nu(i) := false
T3:  choosing(i) := true
T4:  number(i) := 1 + max{number(j) | 0 ≤ j ≤ n-1, j ≠ i}
T5:  choosing(i) := false
T6:  for j = 0 to n-1, j ≠ i do permitted.ch(j) := true od
T7:  for j = 0 to n-1, j ≠ i do
T8:      while ( choosing(j) ) do
T9:          await permitted.ch(i)
T10:         permitted.ch(i) := false
T11:     od
T12:     while ( number(j) ≠ 0 and (number(i),i) > (number(j), j) ) do
T13:         await permitted.nu(i)
T14:         permitted.nu(i) := false
T15:     od
T16: od
C:   ** CS **
E1:  next := min{(number(j), j) | 0 ≤ j ≤ n-1, j ≠ i}
E2:  number(i) := 0
E3:  if next = -1 then for j = 0 to n-1 do permitted.nu(j) := true od
E4:  else permitted.nu(next) := true fi   ▷ focused release
R:   ** Remainder Region **
```

Algorithm 15: L_3

Claim：function min 可以找到次小的 process。

Proof：若 $\max\{number(j) \mid 0 \le j \le n-1, j \ne i\} \ne 0$，則至少存在一個 process 已經完成 line T4 的動作，我們分成兩個 case 來看：

(1) $\max\{number(j) \mid 0 \le j \le n-1, j \ne i\} = number(i)$，則表示有一個以上的 processes 在 doorway 取號碼牌時取到與 process i 相同的號碼，那麼這些相同號碼的 processes 中必然有一個是 $(number(x), x)$ 次小的 process，而且

早在 process i 還在 trying region 時就已經可以確定哪個 process 是下一個進入 CS 的 process。

(2) $\max\{number(j) \mid 0 \le j \le n-1, j \ne i\} > number(i)$，則表示存在至少一個 process 取到的號碼為 $number(i)+1$，在此狀況下也有可能存在 $number(x)$ 與 $number(i)$ 相同的 process，那麼只要在比較的時候等待所有的 processes 確定取完號碼之後，就可以從這些號碼為 $number(i)$ 或 $number(i)+1$ 的 processes 中找 $(number(x), x)$ 最小的 process 即為次小的 process。

所以 function min 可以找到次小的 process。

```
F1:    next := −1
F2:    if ( max{number(j) | 0 ≤ j ≤ n−1, j ≠ i} ≠ 0 ) then
F3:        for  j = 0  to  n−1 ,  j ≠ i  do
F4:            while ( choosing(j) ) do
F5:                await  permitted.ch(i)
F6:                permitted.ch(i) := false
F7:            od
F8:            if ( number(j) ≠ 0 and ( next = −1 or (number(j), j) < (number(next), next) ))
F9:                then  next := j
F10:           fi
F11:       od
F12:   fi
```

$function: \min\{(number(j), j) \mid 0 \le j \le n-1, j \ne i\}$

Figure 5: Function min

當 process 在 exit region 中若可以找到下一個進入 CS 的 process，則需要 $O(3n)$ 次 remote read + $O(1)$ 次 remote write，若是找不到，則需要 $O(n)$ 次 remote read + $O(n)$ 次 remote write，為了維持 first come first served 的特性，在 focused release 上會花相當大的成本來做 search 的動作。

4.5 Burns' mutual exclusion algorithm and its variation

4.5.1 The original version: Algorithm B_0

Burns'[2]最大的特點就是它只用了 n 個 shared variables，正是 atomic read / write register mutual exclusion algorithm 的 lower bound，如 Algorithm 16，但是卻有不可避免的 starvation，可以觀察到 B_0 有三個 loop，分別是 line T2-T4、line T6-T8、line T9-T11，透過 safety property 和 progress property 的證明先分辨出 P-loop 與 S-loop。

Shared variables:

- for every i, $0 \le i \le n-1$, $flag(i) \in \{0,1\}$, initially 0, writable by i and readable by all processes

Process i :
(private variable j : integer)

```
R:    ** Remainder Region **
T1:   L:  flag(i) := 0
T2:       for j = 0 to i − 1 do          ▷ P-loop
T3:           if flag(j) = 1 then goto L fi
T4:       od
T5:       flag(i) := 1
T6:       for j = 0 to i − 1 do          ▷ S-loop
T7:           if flag(j) = 1 then goto L fi
T8:       od
T9:   M:  for j = i + 1 to n − 1 do      ▷ S-loop
T10:          if flag(j) = 1 then goto M fi
T11:      od
C:    ** CS **
E:        flag(i) := 0
R:    ** Remainder Region **
```

Algorithm 16: B_0

Claim：B_0 具備 safety property。

Proof by contradiction：假設有兩個 processes α、β 同時進入 CS，則可得到下列事件發生的先後關係，

(1) $t(E^{\alpha}_{T5}) < t(E^{\alpha}_{T6-T11})$，因為這是同一個 process 所執行。

(2) $t(E^{\beta}_{T5}) < t(E^{\beta}_{T6-T11})$，因為這是同一個 process 所執行。

(3) $t(E^{\alpha}_{T6-T11}) < t(E^{\beta}_{T5})$，因為 process α 順利進入 CS。

(4) $t(E^{\beta}_{T6-T11}) < t(E^{\alpha}_{T5})$，因為 process β 順利進入 CS。

由(2)、(3)、(4)得 $t(E^{\alpha}_{T6-T11}) < t(E^{\alpha}_{T5})$ ，與(1)發生矛盾。

Claim：B_0 具備 progress property。

Proof by contradiction：假設在某時刻 S 無任何 process 在 CS 而且已有 process 在 trying region，在 S 之後無任何 process 可進 CS。因為 processes 數量是有限個，不失一般性假設在 S 之後的某個時刻 S1，在 S1 之後所有的 processes 都在 remainder region 或 trying region 中，且沒有任何 process 再進入 trying region，則在 S1 之後存在某個時刻 S2，在 S2 之後於 trying region 中 id 最小的 process α 必可通過 line T2-T4、line T6-T8 的兩個 loop 而到達 line T9-T11 的 loop，且因為沒有任何 process 可進入 CS，所以 process α 會一直反覆執行 line T9、T10，那麼則表示存在一個 process β 的 *flag* 值為 1，且 $\alpha < \beta$。分成兩個 case 來討論 process β 的行為：（1）process β 正在執行 line T6-T8，則 process β 會因為 process α 而回到 line T1，並且停留在 line T2-T4，此時 process β 的 *flag* 值為 0，（2）process β 正在執行 line T9-T11，那麼因為沒有任何 process 可進入 CS，行為會和 process α 一樣反覆執行 line T9、T10。因此，在 S2 之後存在某個時刻 S3，在 S3 之後所有在 trying region 中 *flag* 值為 1 的 processes 都停留在 line T9-T11 的 loop 中，*flag* 值為 0 的 processes 都停留在 line T2-T4 的 loop 中。在 S3 之後，這些 *flag* 值為 1 的 processes 中，id 最大的 process 必然可以通過 line T9-T11 的 loop 而進入 CS，發生矛盾。

由上述的證明可以得知，line T2-T4 的 loop 為 P-loop，而 line T6-T8 與 line T9-T11 的兩個 loop 為 S-loop。P-loop 由 generic approach 的精神可以知道適合加上 local spin，但是經由上述的證明可以發現 line T9-T11 的 S-loop 亦可加上 local spin，詳述如後。

4.5.2 Adding local spin to B_0 by using 2-dimensional permitted bits: Algorithm B_1

B_1，如 Algorithm 17，雖然是分別在兩個 loop 加上 local spin，不過兩者的阻擋條件都是 *if* $flag(j) = 1$ *then* …，表示被阻擋的因素是一樣的，等待阻擋因素解除後的釋放也是一致的，所以只需要一組 permitted bits，不需要像 L_1 一般分成兩組。依照經驗仍需要驗證這個版本不會發生 dead lock，可以援用 B_0 的 progress property 證明來輔助。

Claim：B_1 具備 progress property。

Proof：假設兩個 process α 、β ，其中 $\alpha < \beta$ ，分成兩個 case 來看：

(1) process β 因 process α 而進入 local spin：

process β 會在 line T6 處 spinning on $permitted(\beta, \alpha)$，此時 process β 的

flag 值為 0，依照 B_0 的 progress property 證明過程可知 process α 必然先 process β 進入 CS，則 process α 在 line E2 即會釋放 process β。

(2) process α 因 process β 而進入 local spin：

process α 會在 line T17 處 spinning on $permitted(\alpha,\beta)$，此時 process α 的值為 1，依照 B_0 的 progress property 證明過程可知 process β 有兩個狀況：①因為 process α 或某個 process γ（$\gamma<\beta$，$flag(\gamma)=1$）而回到 line T1 重新開始，則 process β 在 line T2 處即會釋放 process α，process β 會反過來因為 process α 或某個 process γ（$\gamma<\beta$，$flag(\gamma)=1$）而進入 local spin，無法再次阻擋 process α 進入 CS，②比 process α 先進入 CS，則 process β 在 line E2 即會釋放 process α。

由上述兩點可知，進入 local spin 的 process 必然會被釋放，在 line T2-T9 的 loop 存在至少一個 process 不會進入 local spin，在 line T14-T20 的 loop 亦存在至少一個 process 不會永遠 local spin，所以 B_1 具備 progress property。

因為 Burns' mutual exclusion algorithm[2]會有 starvation 的問題，就邏輯上很難發現 focused release 以及 fast track 的存在，而前幾個章節也演練過幾次使用 generic approach 加上 local spin，此處就節略不再贅述。

Shared variables:

- for every i, $0 \le i \le n-1$, $flag(i) \in \{0,1\}$, initially 0, writable by i and readable by all processes
- for every i, j, $0 \le i \le n-1$, $0 \le j \le n-1$, $permitted(i, j) \in \{true, false\}$, writable by process i and process j and readable by process i

Process i :
(private variable j : integer)

```
R:    ** Remainder Region **
T1:   L:  flag(i) := 0
T2:       for j = 0 to n-1 do permitted(j,i) := true od
T3:       for j = 0 to i-1 do                ▷ P-loop
T4:           permitted(i, j) := false
T5:           if flag(j) = 1 then
T6:               await permitted(i, j)
T7:               goto L
T8:           fi
T9:       od
T10:      flag(i) := 1
T11:      for j = 0 to i-1 do                ▷ S-loop
T12:          if flag(j) = 1 then goto L fi
T13:      od
T14:  M:  for j = i+1 to n-1 do              ▷ S-loop
T15:          permitted(i, j) := false
T16:          if flag(j) = 1 then
T17:              await permitted(i, j)
T18:              goto M
T19:          fi
T20:      od
C:    ** CS **
E1:       flag(i) := 0
E2:       for j = 0 to n-1 do permitted(j,i) := true od
R:    ** Remainder Region **
```

Algorithm 17: B_1

五、 Conclusion

從本文的實際演練中觀察到的幾個現象，做出以下的推論：

(1) generic approach 適用於任何 mutual exclusion algorithm of atomic read/write model。

(2) 2-dimensional permitted bits 的方式適用於具 starvation 的 algorithm，如：Dijkstra's[4]、Burns'[2]。

(3) 2-dimensional permitted bits 的方式適用於 one-writer/multiple-reader mutual exclusion algorithm，如：Lamport's bakery[8]、Burns'[2]。

(4) fast track 依賴 focused release 而存在。

(5) 具 starvation 的 algorithm 很難找到 focused release，如：Dijkstra's[4]、Burns'[2]。

(6) 具 bounded waiting 的 algorithm 可以找到 focused release，如：Knuth's[6]、Eisenburg-McGuire's[5]、Lamport's bakery[8]。

(7) one-writer/multiple-reader mutual exclusion algorithm 的 S-loop 有機會加上 local spin，如：Lamport's bakery[8]、Burns'[2]。

雖然上述的現象只是從本文中演練的 algorithm 中歸納出來，目前尚不能推廣到所有的 mutual exclusion algorithm，但就邏輯上的推演並不無道理，詳述如下。

(1) 如第三章描述。

(2) 會發生 starvation 的 algorithm 通常都會讓 processes 進行激烈的競爭，何者可以進入 CS 可以說完全是隨機因素決定，也因為如此，使用 2-dimensional permitted bits 加上 local spin 機制時，雖然在釋放過程只有部分 local-spinning processes 被釋放，但是這些被釋放的 local-spinning processes 在進行激烈的競爭時，仍然有能力進入 CS。

(3) one-writer/multiple-reader mutual exclusion algorithm 有一個通則，就是 process 要獲得彼此之間的狀態只能透過 linear reading 其它 process 專屬的 shared variable 方可得知，也因為如此，這類的 algorithm 無論在 P-loop 或 S-loop 測試失敗後，通常還是持續進行 linear reading 的動作，一直到阻礙的因素解除為止，那麼使用 2-dimensional permitted bits 來加上 local spin 機制不會造成 dead lock 的原因就在於行為上正是符合這樣的特性，只是反過來由造成其它 process 進入 local spin 的 process 主動通知（釋放）

local-spinning processes 可以繼續原本該有的行為，如此而已。

(4) 沒有 focused release 就沒有辦法分辨哪個 process 有機會進入 CS，更不能分辨哪個 process 確定可以進入 CS。

(5) 會發生 starvation 的 algorithm 無法確定 processes 進入 CS 的順序，所以無法使用 focused release 還能確保不發生 dead lock。

(6) 具備 bounded waiting 性質的 algorithm 會利用某種機制使得 process 不能連續進入 CS 兩次以上，也就是說，這樣的機制會使得部份 processes 失去競爭的優勢，那麼只要在 exit region 利用反向機制找到仍然具備競爭優勢的 process，它就是 focused release 的標的。

(7) 理由與(3)是一樣的，algorithm 中反覆進行 linear reading 的 loop 若為 S-loop，那麼加上 local spin 也不會造成 dead lock。

綜合上述的演練經驗，不具備 bounded waiting（starvation freedom）的 mutual exclusion algorithm 建議採用 2-dimensional permitted bits 的方式加上 local spin 比較有助於 memory contention 的降低；具備 bounded waiting 的 mutual exclusion algorithm 則建議採用 the generic approach 以及添加 focused release，在 heavy loading 的狀況下會有較佳的表現。

本文的特點包含成功找出 recursive function 來證明 Knuth's mutual exclusion algorithm 具備 $2^{n-1}-1$ bounded waiting 性質；利用一個 Lemma 即證明 Eisenburg-McGuire's mutual exclusion algorithm 加上 focused release、fast track 的正確性，以及具備 $n-1$ bounded waiting 性質；發現使用 2-dimensional permitted bits 加上 local spin 的方法並不適用於任何 mutual exclusion algorithm；利用 tree 的結構來分析 the generic approach 和 2-dimensional permitted bits 的差異。

參 考 文 獻

[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] 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.

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

[5] 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.

[6] 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.

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

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

[9] Nancy A. Lynch, Distributed Algorithms, Morgan Kaufmann, 1996.

[10] 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.

[11] Gadi Taubenfeld, Synchronization Algorithms and Concurrent Programming, Prentice Hall, 2006.

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

[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.

附　錄

Gary L. Peterson's n-processes mutual exclusion algorithm and its variation

(1) The original version: Algorithm P_0

Shared variables:

- for every k, $1 \le k \le n-1$, $last(k) \in \{0,\ldots,n-1\}$, initially arbitrary, writable and readable by all processes
- for every i, $0 \le i \le n-1$, $flag(i) \in \{0,\ldots,n-1\}$, initially 0, writable by process i and readable by all processes

Process i :
(private variable j : integer, k : integer)

```
R:   ** Remainder Region **
T1:  for k = 1 to n-1 do
T2:      flag(i) := k
T3:      last(k) := i
T4:      for j = 0 to n-1, j ≠ i do
T5:          while( flag(j) ≥ flag(i) and last(k) = i ) do nothing od
T6:      od
T7:  od
C:   ** CS **
E:   flag(i) := 0
R:   ** Remainder Region **
```

(2) Adding local spin to P_0 by using 2-dimensional permitted bits: Algorithm P_1

Shared variables:

- for every k, $1 \le k \le n-1$, $last(k) \in \{0,\ldots,n-1\}$, initially arbitrary, writable and readable by all processes
- for every i, $0 \le i \le n-1$, $flag(i) \in \{0,\ldots,n-1\}$, initially 0, writable by process i and readable by all processes
- for every i, j, k, $0 \le i \le n-1$, $0 \le j \le n-1$, $1 \le k \le n-1$, $permitted(i,j,k) \in \{true, false\}$, writable by process i and process j and readable by process i

Process i :
(private variable j : integer, k : integer)

```
R:   ** Remainder Region **
T1:  for k = 1 to n-1 do
T2:      flag(i) := k
T3:      last(k) := i
T4:      for j = 0 to n-1 do permitted(j,i,k) := true od
T5:      for j = 0 to n-1, j ≠ i do
T6:          permitted(i,j,k) := false
T7:          while( flag(j) ≥ flag(i) and last(k) = i ) do
T8:              await permitted(i,j,k)
T9:              permitted(i,j,k) := false
T10:         od
T11:     od
T12: od
C:   ** CS **
E1:  flag(i) := 0
E2:  for k = 1 to n-1 do
E3:      for j = 0 to n-1 do permitted(j,i,k) := true od
E4:  od
R:   ** Remainder Region **
```

(3) Adding local spin to P_0 by the generic approach: Algorithm P_2

Shared variables:

- for every k, $1 \le k \le n-1$, $last(k) \in \{0,\ldots,n-1\}$, initially arbitrary, writable and readable by all processes
- for every i, $0 \le i \le n-1$, $flag(i) \in \{0,\ldots,n-1\}$, initially 0, writable by process i and readable by all processes
- for every i, k, $0 \le i \le n-1$, $1 \le k \le n-1$, $permitted(i,k) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer, k : integer)

```
R:   ** Remainder Region **
T1:  for k = 1 to n-1 do
T2:      permitted(i,k) := false
T3:      flag(i) := k
T4:      last(k) := i
T5:      for j = 0 to n-1, j ≠ i do permitted(j,k) := true od
T6:      for j = 0 to n-1, j ≠ i do
T7:          while( flag(j) ≥ flag(i) and last(k) = i ) do
T8:              await permitted(i,k)
T9:              permitted(i,k) := false
T10:         od
T11:     od
T12: od
C:   ** CS **
E1:  flag(i) := 0
E2:  for k = 1 to n-1 do
E3:      for j = 0 to n-1 do permitted(j,k) := true od
E4:  od
R:   ** Remainder Region **
```

(4) Adding focused release to P_2: Algorithm P_3

Shared variables:

- for every k, $1 \le k \le n-1$, $last(k) \in \{0,\ldots,n-1\}$, initially arbitrary, writable and readable by all processes
- for every i, $0 \le i \le n-1$, $flag(i) \in \{0,\ldots,n-1\}$, initially 0, writable by process i and readable by all processes
- for every i, k, $0 \le i \le n-1$, $1 \le k \le n-1$, $permitted(i,k) \in \{true, false\}$, writable by all processes and readable by process i

Process i :
(private variable j : integer, k : integer, $next$: integer)

```
R:   ** Remainder Region **
T1:  for k = 1 to n − 1 do
T2:      permitted(i, k) := false
T3:      flag(i) := k
T4:      last(k) := i
T5:      for j = 0 to n − 1, j ≠ i do permitted(j, k) := true od
T6:      for j = 0 to n − 1, j ≠ i do
T7:          while( flag(j) ≥ flag(i) and last(k) = i ) do
T8:              await permitted(i, k)
T9:              permitted(i, k) := false
T10:         od
T11:     od
T12: od
C:   ** CS **
E1:  next := last(n − 1)
E2:  flag(i) := 0
E3:  if (next = i) then
E4:      for k = 1 to n − 1 do
E5:          for j = 0 to n − 1 do permitted(j, k) := true od
E6:      od
E7:  else permitted(next, n − 1) := true fi          ▷ focused release
R:   ** Remainder Region **
```