行程內與行程間的通訊同步機制 (II)

Intra and Inter Process Communication Mechanism (II)

計畫編號 ：NSC87 - 2213 - E009 - 025
執行期間 ： 85 /08 /01 -- 86 /07 /31
主 持 人：袁賢銘 教授 國立交通大學資訊科學系

一、中文摘要（關鍵字： 分散式系統，多執行緒，並行性控制， Windows NT）

在分散式系統裡，為了避免同時存取共享的資源所可能發生的競賽狀況，在存取共享資源時，都會使用同步控制機制來避免這種問題發生。Lock，或類似的機制，是目前被使用的最廣泛的一種同步控制機制。我們發現，在很多情況下，程式在對 Lock 的使用上會發生 Nesting 的情況；也就是說，某個執行單元在獲得 Lock 之後，會再向系統要求相同的 Lock。這在某些自動提供同步控制的系統，例如交易處理系統；或是某些設計時不是很注意的程式裡，都可以看到這樣的特性。因此，我們設計了一個階層式的 Lock 機制，來利用程式中的這項特性。它使用了代理者的概念，來減少在要求一個nested lock時所產生的額外負擔。另外，它也提供了容錯的功能，可以容許系統裡一個伺服器節點發生錯誤的情況。它還提供了一個圖形使用者介面，方便使用者管理與容錯相關的功能。我們這套同步控制機制目前被實作在 Windows NT 4.0 上，它和 NT 內建的 critical section 機制完全相容；而且它還可以提供程式更高的平行性。因為整個機制被完全地實作在使用者模式下，所以也很容易移植到其他的作業系統平台上。另外，因為 Windows NT 是一種支援 SMP 的作業系統，所以我們也利用了一些 MP 的特性，來減少這個機制所帶來的額外負擔。

英文摘要 (Keywords: distributed systems, multi-thread, concurrency control, Windows NT)

In distributed systems, race condition may occur if two or more execution units try to access a shared resource concurrently. In order to prevent this problem, these execution units will use concurrency control facilities during they access shared resources. Locks, or similar facilities, are the most popular concurrency control facilities for their simplicity. We find out that some programs often use nesting locks; that is, after an execution unit has been granted a lock, it still requests the same lock before it releases the lock. This property can be found in systems that provide automatic concurrency control, such as transaction processing systems. It also can be found in a program with mindless design. Hence, we devise a hierarchical lock facility to exploit this property. It employs the notion of proxy to reduce the overhead of requesting a nested lock. In addition, it is also fault-tolerant, and can tolerate single-site failure of server nodes. Moreover, it provides a GUI interface for managing fault-tolerant-related functions conveniently. We implement this facility upon Windows NT 4.0. It is compatible with the critical section facility that was built in Windows NT natively; furthermore, it makes programs exploit their concurrency more easily than using critical section. It can also be ported to other operating systems with ease because it is fully implemented under user-mode. Besides, since Windows NT is an operating system that supports SMP hardware platform, we take advantage of some characteristics of multi-processor systems, and try to reduce the overhead of using this facility. In this thesis, we will examine each aspect of this facility.

二、計畫緣由與目的

在分散式系統裡，為了避免同時存取共享的資源所可能發生的競賽狀況 (race condition) [2][6]，在存取共享資源時，都會使用同步控制機制來避免這種問題發生。Lock [2][6]，或類似的機制，是目前被使用的最廣泛的一種同步控制機制。在分散式系統裡，實作 lock 機制的方式，比較常見的有使用中央式的 lock manager 的方式 (central lock manager approach) [11]，或是使用分散式的互斥演算法 (distributed waiting queue approach) [11]，以及在 DSM 系統裡，非常常見的 distributed waiting queue [11] 等方式。

另外，我們還發現，在很多情況下，程式在使用 lock 時，會發生 nesting 的情況；意思是，當某個執行單元，在獲得一個 lock 之後，在釋放這個 lock 之前，還會再要求同樣的一個 lock。比方說，在 transaction processing 系統 [11] 裡，因為系統要自動提供 concurrency control 的功能，而且，程式執行時所會存取

到的共用資源，並不是在編譯時期就可以確定。所以，lock 機制必須要能提供 nesting 的功能，好讓 transaction process 系統在某個 transaction 執行時，依據它實際存取到的共用資源，來動態地決定該存取哪些資源。如果 lock 機制不支援 nesting 的功能，一個 transaction 在執行時，只要同時以彼此會發生 conflict 的方式，來存取同一個共享資源兩次，這個 transaction 就會卡死在 lock 機制裡，產生死結。另外，在某些程式裡，也有可能發生，某段程式碼所呼叫的另一段程式，和呼叫者程式碼本身，都使用到某個共享的資源的情況。在這種情況下，如果 lock 機制本身不支援 nesting 的功能，程式也會被卡死在 lock 機制裡，產生死結。這種問題，在由多個人所撰寫、體積龐大的程式裡，相當有可能出現。

因此，我們設計了一個階層式的 lock 機制，來利用程式裡這項 nesting 的特性。它在每個行程，及每部機器裡，都使用了代理者，只有在必要時，行程才會和 lock manager 通訊，這可以減少在同一部機器內的 interprocess communication，以及在網路上傳送的訊息。如果使用的是傳統中央式的 lock manager，在每次有行程需要修改它使用的 lock 的狀態時，都會需要在網路上傳遞訊息。通常，我們會以在網路上傳遞的訊息量，來做為衡量一個分散式的 concurrency control 機制，所帶來的 overhead 的一項重要標準。因此，如果程式裡，使用 lock 機制的方式，有相當比例的 nesting，相較於傳統中央式的 lock manager 的實作方法，使用我們的機制所帶來的 overhead 會比較小。另外，因為分散式的互斥演算法，所需要傳遞的訊息，高過傳統中央式的 lock manager，所以，在前述的程式裡，所產生的 overhead，也一定會高過我們提出的機制。Distributed waiting queue 機制，雖然會比中央式的 lock manager 方法，需要在網路上傳遞更多的訊息，但因這些訊息的傳遞，是分散在各個機器間的，並不會像中央式的 lock manager，所有訊息的傳遞都會牽扯到 lock manager，所以，系統裡不會出現像 lock manager 一樣的瓶頸。而且，distributed waiting queue 機制，在遇到使用 nested lock 的程式時，也可以減少在網路上傳送的訊息量。所以，在我們未來的研究計畫裡，會試圖將我們所提供的 lock 機制，和 distributed waiting queue 機制，做一個系統化的比較。

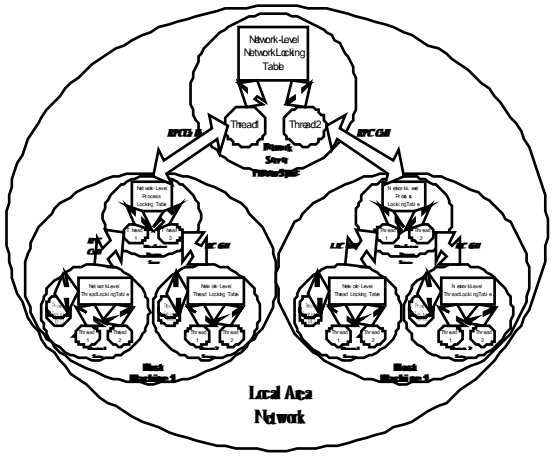
因為分散式系統裡的節點很多，所以，系統裡有節點發生錯誤的機率，也會隨之增加。為了避免這種只要有一部機器出錯，系統就會完全失效的情況出現，我們也為我們的 lock 機制，提供了容錯的功能。我們使用的是 modular redundancy 的 process resilience 方式。這項功能，使系統可以容忍一個伺服器節點發生錯誤的情況。另外，它還提供了 GUI 的界面，讓使用者可以很輕易地，管理和容錯相關的功能。

我們的這套機制，目前被實作在 Windows NT 4.0 上，它和 Windows NT 內建的 critical section 機制完全相容；而且，它還可以提供程式，比原本 Windows NT 內建的 critical section 機制，更高的 concurrency。還有，因為我們的機制，被實作成在使用者模式下執行的程式庫，所以，可以很容易地被移植到其他的作業系統平台上。另外，因為 Windows NT 是一種支援了 SMP 硬體平台的作業系統，因此，在這裡，我們也利用了 multiprocessor 系統的特性，來減少在我們的機制內部使用的同步方式，所產生的 overhead。

三、結果與討論

我們曾在去年的國科會計畫 - 行程內與行程間的通訊同步機制 (II) 裡，設計並實作出一個兩層式 (thread level/process level) 的行程內同步機制 [1]，我們今年的研究工作，就是試圖在原有的研究成果上，繼續擴充，在原本的 thread level/process level 同步機制上，再加上一層 network level 的同步機制。我們提出了幾種可能的架構，經過分析後，我們決定採用圖 1 的系統架構。

圖 1：Network-Level Lock Facility 的系統架構



這種架構雖然在設計及實作上的複雜度較高，但是，因為它在每部機器上，多了一個 host agent，因此，在網路上傳遞的訊息數量，會因此而減少，進而增加了系統的 throughput。而且，使用這個方法，可以在完全不更動我們之前的 thread level/process level 的同步機制的情況下，將 network level 的同步機制，建置在原有的系統上。

至於在使用者界面上，還是沿用類似 thread level/process level 同步機制的使用者界面。下面是關於這個界面的簡單說明：

1. int *open_nlock*(char **nname*, int *lock_type*);

   開啟一個 network level lock。

2. **BOOL** *set_nlock*(int *descriptor*, int *op*);

   要求某個已被開啟的 network level lock。

3. **BOOL** *reset_nlock*(int *descriptor*);

   釋放某個已被要求的 network level lock。

4. **BOOL** *close_nlock*(int *descriptor*);

   關閉某個已被開啟的 network level lock。

5. **BOOL get_nlock_status**(int *descriptor*);

   取得某個已被開啟的 network level lock 的狀態。

至於這些 API 詳細的使用方法，請參考 [16] 裡的完整說明。

在支援容錯的功能上，我們採用了 modular redundancy [7] 的容錯方式。這種方式雖然有 failure recovery 比較快，以及可以避免 Byzantine failure 等好處，但是，當被複製的 server 是 concurrent server 時，會因為每個被複製的 server 上，concurrency control 的執行結果不確定性，而導致每個 server 的副本，可能會產生不同的執行結果的情況出現。在這裡，我們為了提高系統的 throughput，而決定採用 concurrent server，但我們也想出一個方法，來解決這樣的問題。至於進一步的細節，請參考 [16] 裡的完整說明。

我們在 Windows NT 4.0、Pentium-133/32 MB RAM 的機器上，比較我們的 network-level lock facility 和 file server 上，鎖定一個使用 exclusive access mode 的檔案，之間的效率差異。

我們利用鎖定檔案的方法 (使用 CreateFile( ) API)，在 Windows NT 4.0 上，當作一種模擬 distributed concurrency control 機制的方法。但需要注意的是，這種方法只能模擬沒有 nesting 功能的 mutual exclusive locks。因此，我們也只能拿 network-level lcok facility 所提供的 mutual exclusive lock 功能，來和它做比較。

較。

我們比較的第一種情況是，lock server 和使用 lock 的 client，分處於不同機器上的情況。我們會分別測量，在這兩種不同的機制上，要求一個 lock，以及釋放一個 lock 時，所產生的 round-trip delay。我們的 network-level lock facility，要求一個 lock 然後釋放，需要花 8 milliseconds 的時間，但使用 CreateFile( ) 只需花費 2 milliseconds. 這項結果應該是合理的，因為 Windows NT 的檔案系統是實作在作業系統核心裡的；反觀我們的機制，在運作時，會分別經過一個在 client 端、以及一個在 server 端的 server，這些 server 都是在使用者模式下執行，因此，會帶來比較大的 overhead。

接著，我們測量這兩項機制，在 throughput 上的差異。我們在這裡使用了兩個 client，這兩個 client 和 server 是分開的。每個 client 裡，都包含了 6 個 thread，每個 thread 會分別執行 8 次 lock 的要求，以及釋放動作。在獲得 lock 後，進行釋放前，會執行一個迴圈，裡面執行的是指數函數的運算。在這裡，我們的 network-level lock facility 表現，要比 CreateFile( ) 來得好得多。請參考表 1。

	Simple Version		Fault Tolerant Version	
	No Nesting	Nesting	No Nesting	Nesting
CreateFile( )	0.83676	N/A	N/A	N/A
Distributed Mutex	2.12886	1.81536	1.96374	1.6614

表 1： CreateFile( ) 與 network-level lock facility 之間的 Throughput 比較

因為 CreateFile( ) 不支援 lock nesting 及容錯，因此，我們也沒辦法進行這方面的測量。另外，因為支援容錯功能的 network-level lock facility，比起不支援容錯的版本，多增加了一道執行 backup server 的手續，因此，效率上也會變得比較差一點。

四、 研究成果自評

本計畫原本預計在 Windows NT 及 OSF/1 系統上，實作出功能相同的機制。但因為 OSF/1 作業系統本身不穩定，而且支援的程式開發工具不足，因此，本計畫只將最後的成果，實作在 Windows NT 作業系統下。本計畫的性質，比較偏向系統實作，預計會將結果在一年內發表於國外的相關期刊上。

目前，和本年度計畫相關，已發表的著作有 [16]。

五、 結論

Concurrency control 機制，可以避免在分散式

系統裡，可能會發生 race condition 的問題。因為使用及實作上的簡便，lock 成為目前被使用得最廣的一種 concurrency control 機制。在這篇結果報告裡，我們介紹了一個階層式的 lock 機制。它利用了在程式裡，對 lock 的使用上，會有 nesting 現象的特性，並使用了代理者的方式，來減少在同一部機器內的 interprocess communication，以及在網路上傳送的訊息，這會讓使用它所帶來的 overhead 變小。在這篇成果報告裡，我們介紹了這個機制的整體架構 (system architecture)、採用這項架構的原因、程式設計界面、以及實際的效率測量。至於有關容錯功能方面的細節、系統所使用到的資料結構與演算法、以及如何使用這個機制的詳細說明，請參考 [16]。

六、參考文獻：

[1] W. P. Tsay, "The Design and Implementation of an IIPC Lock facility", Master Thesis, Institute of Computer Science and Information Engineering, College of Electrical Engineering and Computer Science, National Chiao Tung University, June 1997.

[2] Abraham Silberschatz, Perter B. Galvin, "Operating System Concepts", Fourth Edition, Addison-Wesley Publishing Company, Inc., 1994.

[3] Jeffrey Richter, "Advanced Windows", Third Edition, Microsoft Press, 1997.

[4] John Shirley and Ward Rosenberry, "Microsoft RPC Programming Guide", O'Reilly & Associates, Inc., 1995.

[5] "Multithreaded Programming Guide", Sun Microsystems, Inc., 1994.

[6] Jim Beveridge, Robert Wiener, "Multithreading applications in Win32: The Complete Guide to Threads", Addison-Wesley Publishing Company, Inc.,1997.

[7] Pankaj Jalote, "Fault Tolerance in Distributed Systems", Prentice Hall International, Inc., 1994.

[8] K.S. Yap, P. Jalote, S. Tripathi, "Fault Tolerant Remote Procedure Call", 11th IEEE conf. on Distributed Computing Systems, pp. 48-54, 1988.

[9] Chaio-Chung Wu, "The Design and Implementation of Distributed Semaphore", Master Thesis, Institute of Computer Science and Information Engineering, College of Electrical Engineering and Computer Science, National Chiao Tung University, June 1992.

[10] Donald Mclaughlin, Shantanu Sardesai and Partha Dasgupta, "Calypso NT: Reliable, Efficient Parallel Processing on Windows NT Networks", http://www.eas.asu.edu.tw/~calypso/research_papers/calnt.

[11] Andrew S. Tanenbaum, "Distributed Operating Systems", Prentice-Hall International, Inc., 1995.

[12] Joseph Boykin, David Krischen, Alan Langerman, and Susan LoVerso, "Programming under Mach", Addison-Wesley Publishing Company, Inc., 1993.

[13] Bennett, J.K., Carter, J.K., and Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", Proc. Second ACM Symp. on Principles and Practice of Parallel Programming, ACM, pp. 168-176, 1990.

[14] Bershad, B.N., and Zekauskas, M.J., "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors", CMU Report CMU-CS-91-170, September 1991.

[15] Evan Speight and John K. Bennett, "Brazos: A Third Generation DSM System", Proceedings of the First USENIX Windows NT Workshop, August 1997.

[16] W. U. Lin, "A Low-Overhead, Fault-Tolerant Distributed Concurrency Control facility," Master Thesis, Institute of Computer and Information Science, College of Electrical Engineering and Computer Science, National Chiao Tung University, June 1998.