分散式共享記憶體的容錯與執行緒同步機制之研究 (II)

計畫編號 ：NSC89 - 2213 - E009 - 069
執行期間 ： 88 /08 /01 -- 89 /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： 分散式共享記憶體， 多執行緒， 並行性控制， Java， 死結， deadlock， monitor）

我們今年計畫的重點在，改進去年所提出的 Java-based 的執行緒同步機制上。去年所提出的改良版的 Java-based monitor (EMonitor) 機制, 在語法上並不是和 Java 程式語言整合得很好。所以今年，我們改良了我們的 monitor 機制的語法。此外，在 monitor lock 的實作上，我們也做了一些改良，以降低使用時的 overhead。而且，在 condition queue class 的設計上，我們也重新依所需的應用程式的特性，來提供各種專用的 condition queue 機制. 我們已經將改良過的 EMonitor 機制的成果，投稿至 *Software – Practice & Experience* 期刊。目前，我們的 paper 已經完成了 first revision，有希望在短時間內被該期刊接受。

英文摘要 (Keywords: distributed shared memory, multi-thread, concurrency control, Java, deadlock, monitor)

The focus of our project during this year is on refining the Java-based thread synchronization mechanism (EMonitor) that proposed by us in the last year. The syntax of original EMonitor is not well-integrated with the Java programming language.Hence, we try to smoothly integrate the syntax of EMonitor with the syntax of Java. Besides, for reducing the running overhead, we also refine the implementation of the monitor lock of EMonitor. In addition, we also redesign the condition queue classes for better fitting the requirements of concurrent applications. We have sumitted the result of the refined EMonitor mechanism to the international – *Software – Practice & Experience*. Currently, our paper is undergone the first revision and resubmitted for publication.

二、計畫緣由與目的

Java 所提供的執行緒同步機制，基本上有四個大問題:

No-priority monitor: 依照 [BFC95] 裡所提出的分類方式，Java 的 monitor 是一種 no-priority monitor。在一個 monitor 裡，可能存在四種不同的 queue，它們分別是: 一個 entry queue、一個 waiting queue、一個 signaller queue、以及一個或多個 condition queues。在這四種 queue 裡，只有 condition queue 裡的執行緒必須等待某項事件發生後，才能繼續執行。至於其他三種 queue 裡的執行緒，則只等著一個一個魚貫進入 monitor。因此，我們也稱這三種 queue 為 ready queues。

至於 entry queue 的優先權，則應該是最低的，否則，可能會有 starvation 的問題發生。如果 entry queue 的優先權等於 signaller queue 或 waiting queue，則這種 monitor 稱作 *no-priority monitor*；反之，則稱作 *priority monitor*。Java 的 monitor 因為使用了 nested mutually exclusive lock，來實作 monitor 的互斥性質。所以，我們可以將 monitor lock 裡的 lock waiting queue，看作是前述的 monitor 的 model 裡，entry queue 和 waiting queue 的結合。因為在 Java 並沒有指定 lock waiting queue 的性質 [GJS96]，因此，我們可以假設 entry queue 和 waiting queue 的優先權是相等的，這表示 Java 的 monitor 是一種 no-priority 的 monitor。

No-priority monitor 所會帶來的第一個問題是，wait 的 postcondition 和 signal 的 precondition 可能會不相同 [OW97][BFC95][And91]。這個問題發生的原因在，會有其他的執行緒搶先被 signal 的執行緒進入 monitor，而將 signal 的 precondition 破壞。No-Priority Monitor 的第二個問題是，它會使 scheduling 程式的撰寫變得更複雜 [BFC95][Geh93] [And91]。

只有一個 Condition Queue: 每個 Java 物件裡，只有一個 condition queue。如果停滯在 condition queue 裡的 thread, 所等待的都是相同的事件，就不會有進一步的問題發生。然而，當停滯在 condition queue 裡的 thread，所等待的事件超過兩種以上時，就有可能出現，被 notify( ) 叫醒的執行緒，所等待的並不是 notify( ) 所要提示的事件的情況發生。所以，被叫醒的 執行緒，就必須要檢查它所等待的事件, 是不是真的發生了, 如果不是，它就必須呼叫 notify( )，叫醒另一個 condi-

tion queue 裡停滯的執行緒。接著，再呼叫 wait( )，讓自己重新停滯。上述的動作會一直重複，直到一個正確的執行緒被叫醒為止。然而，如果沒有執行緒等待 notify( ) 所代表的事件，上述的演算法會無法停止。此外，這種方法還會叫醒不相關的執行緒。這會引發額外的執行緒 context switch

缺乏對 Scheduling 的支援: 所謂的 scheduling，指的是從一堆 request 裡，選出一個讓它繼續執行。這同時也暗示了，scheduler 必須擁有關於這些 request 的 global information。有一些現存的 monitor 機制，就提供了 prioritized condition queue，來簡化程式設計者撰寫 scheduling 的程式[YH97][OH96][OH95][SCH95][And91]。Prioritized condition queue，會自動將 pending thread 依指定的優先權大小排序。因此，它適合用來處理 static scheduling 的問題。在這種排程問題裡，只要被排程的 thread 的優先權一被指定，它的優先權就會固定下來。相反地，在 dynamic scheduling 的問題裡，被排程的 thread 的優先權則可能會變動。這使得 prioritized condition queue 變得一點用都沒有了。反觀 Java，只提供了一個簡陋的 random condition queue [GJS96]，也沒有提供其他任何的機制，來幫助程式維護 scheduling 所需的 global information。因此，如果想在 Java 上撰寫 scheduling 的程式，程式設計者必須自行以更複雜、成本更高的方式，來維護所需的 global information。

Inter-Monitor Nested Call 的 Deadlock 問題: Java 使用 nested mutually exclusive lock 來實作 synchronized method 的互斥性質[GJS96]。但是，由於 Java 沒有區分 intra-monitor nested call 及 inter-monitor nested call 的呼叫機制 (method invocation mechanism)，而使得 deadlock 有可能發生。這種 deadlock 還可以再被細分為兩類。第一類是所謂的 mutual-dependent deadlock。假設現在有 A 和 B 兩個執行緒。執行緒 A 正在執行物件 M_1 裡的 synchronized method，而執行緒 B 也已經在物件 M_2 的某個 synchronized method 裡。換句話說，執行緒 A 已經持有物件 M_1 的 monitor lock，相同地，執行緒 B 也已經持有物件 M_2 的 monitor lock。如果現在執行緒 A 呼叫物件 M_2 裡的 synchronized method，而執行緒 B 同時也呼叫了物件 M_1 裡的 synchronized method，就會發生 mutual-dependent deadlock。這個問題，也已經在 [Bro98][VA98][OW97] 裡被指出。

第二種可能發生在 inter-monitor nested call 的 deadlock 是所謂的 condition-wait deadlock。假設某個執行緒 C，由物件 M_1 起，經由物件 M_2、M_3 … 直到物件 M_N，連續呼叫了 N 次的 synchronized method。接著，執行緒 C 會在物件 M_N 裡，呼叫 wait( ) 將自己停滯，並釋放物件 M_N 的 monitor lock。現在再假設有另一個執行緒 D，會設法進入物件 M_N，並呼叫 notify( ) 把執行緒 C 叫醒。如果它的呼叫路徑，會經過 M_1 到 M_{N-1} 間的任何一個物件，就會發生 condition-wait deadlock。

本計畫的 EMonitor 機制，就是為了改進 Java monitor 前述的幾個問題而設計的。

三、結果與討論

新的執行緒同步機制的特色

基於效率及相容性的考量，我們所提出的新執行緒同步機制，仍然是衍生自 monitor，(我們稱它作 EMonitor)，它有以下的幾點特色:

- 它是一種 priority monitor。
- 有多種不同的 monitor ，各自支援不同的 signal semantic:
  - Blocking signal。
  - Non-blocking signal。
- 同一個 EMonitor 物件裡，可以有一個以上的 condition queue。此外，我們共提供了三種不同的 condition queue:
  - FIFO conditon queue: 類似 Java 目前所提供的 condition queue。
  - Prioritized condition queue: 提供類似其他 monitor 機制上已經存在的 prioritized condition queue 機制，供 static priority scheduling 程式使用。
  - Customizable condition queue: 專供 dynamic priority scheduling 程式使用。
- 提供了 open call [And91][Kot87] 及 closed call，兩種不同的 inter-monitor nested call 的 semantic。

EMonitor 和 Java monitor 在效率上的差異

在這裡，我們只考慮有 contention 發生的情況下的效率差異。我們使用了兩種 benchmark 程式，分別各代表一種類型的問題。

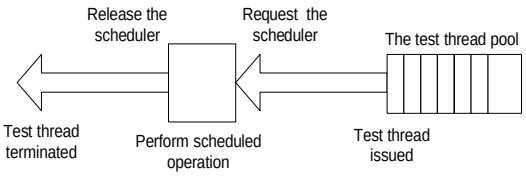
第一類的 benchmark 是 bounded buffer。Bounded buffer 可以用來凸顯 Java monitor 機制，只提供了一個 condition queue 所可能造成的問題。(Java monitor 版的 bounded buffer，只能使用一個 condition queue，而

EMonitor 版的 bounded buffer，則採用兩個

	Bounded Buffer		FIFO Semaphore		Elevator Disk Scheduler		SSF Disk Scheduler	
	5 ms Inv.	6 ms Inv.	2 ms Op.	4 ms Op.	2 ms Op.	4 ms Op.	2 ms Op.	4 ms Op.
$\frac{\text{JM – EMNB}}{\text{EMNB}}$	198.8 %	100.5 %	- 4.7 %	- 0.6 %	16.7 %	11.1 %	13.4 %	11 %
$\frac{\text{EMB – EMNB}}{\text{EMNB}}$	0.2 %	0.3 %	5.5 %	1.5 %	4 %	1.2 %	1.6 %	0.5 %

表一: 所有實驗數據的摘要

condition queue 來實作。)至於第二種的 bencgmark，則是三個不同類型的 scheduling program。FIFO semaphore 代表只需使用一個簡單的 FIFO condition queue 就可以實作的 concurrent object。Elevator disk scheduler 代表可以使用 prioritized condition queue 就能實作的 static scheduling problem。SSF disk scheduler 代表複雜到必須採用 customized condition queue 才能實作的 dynamic schediling problem。我們進行的實驗方法，請參考圖一。在 request pool 裡，會有一群事先已經產生的執行緒。這些執行緒會依指定的時間間隔，一個接著一個離開 request pool，然後，向 scheduler 發出 request。在收到 scheduler 的許可之後，執行緒會去執行一個耗時 2 ms 或 4 ms 的 operation。在這個 operation 結束之後，執行緒會向 scheduler 送出釋放的要求。我們在這裡所測量的，是每一個執行緒向要求 lock 開始，到最後一個執行緒釋放 scheduler 為止，中間所經過的時間。接著，我們再計算平均值。所有前述的四個實驗程式的數據請參考表一。



圖二: 用來測量 scheduling program 的實驗

首先，我們先比較 Java monitor (no-priority、non-blocking signal) 以及 EMonitor (priority、blocking signal) 在效率上的差異。由 bounded buffer 的結果我們可以發現，Java monitor 只能提供一個 condition queue 的限制，對程式的效能產生了相當程度的影響。而且，據我們的經驗顯示，採用 Java monitor 的 bounded buffer 程式，在 contention 到達中等程度時，就會產生很高的 overhead。

理論上來說，no-priority monitor 的 performance 會比 priority monitor 的 performance 來得差。然而，在 FIFO semaphore 這個測試程式的數據裡，我們發現 Java monitor 的 performance 竟然比 EMonitor 的 performance 還約略好一點。我們認為這個現象是因為 EMonitor 的實作方式所致。由於 EMonitor 本身是由 Java monitor 所實作的，在實作上所多出的 overhead 會高過 EMonitor 是 priority monitor 所帶來的 performance gain。但相反地，在其他的另外兩種 scheduling 程式裡，因為 Java monitor 版程式所採用的 monitor lock，本身也是以 Java monitor 來實作的，所以，EMonitor 版的程式的 performance，會優於 Java monitor 版的程式。然而，從這些 scheduling 程式的角度來看，我們認為 Java monitor 和 EMonitor 的效率事實上還是蠻相近的，除非 thread 的 contention 變得非常高 (接近系統所能提供的極限)。

總之，對需要多個 condition queue 或更複雜的 condition queue 的 concurrent object 來說，EMonitor 提供了更好的 programming style，以及更好的 performance。就算是很簡單的 concurrent object (簡單到用 Java monitor 就可以直覺地處理的)，EMonitor 仍然能在 performance 相去無幾的前提之下，避免 no-priority monitor 所帶來的麻煩。由以上的數點結論，我們可以確定 EMonitor 是可以用來取代 Java monitor 來設計 concurrent object 的一種實用的解決方案。

最後，我們想比較的是 EMonitor 的 non-blocking signal，以及它的 blocking signal 的效率。事實上，除了 FIFO semaphore (2ms scheduled operation) 這個測試程式外，這兩種 signal semantic 所造成的 performance 差異並不大，都只侷限在 4% 以內。而且，在前述的例外裡，blocking signal 也只有在 contention level 接近系統的極限值時，效率上才會出現很明顯的差異。由於 blocking signal

能提供更好的 programming style，而且，效率上也還能接受，因此，我們建議在需要時就可以採用它，不必顧忌太多。

四、研究成果自評

本計畫所做出來的成果 - EMonitor，很明顯地優於 Java monitor。目前，我們已經將成果整理完畢，投稿至 Software – Practice & Experience 這本期刊，目前已經根據 reviewer 的意見，完成 revise 的工作，如果順利的話，成果預計會將結果在一年內發表於國外的相關期刊上。

五、結論

我們在今年的計畫裡，改進了去年我們所設計的 EMonitor 機制，並對 Java monitor 和 EMonitor 的效率，做了非常詳細的分析和比較。我們所得到結論是，對需要多個 condition queue 或更複雜的 condition queue 的 concurrent object 來說，EMonitor 提供了更好的 programming style，以及更好的 performance。此外，就算是簡單到用 Java monitor 就可以直覺地處理的 concurrent object，EMonitor 仍然能在 performance 相去無幾的前提之下，避免 no-priority monitor 所帶來的麻煩。由以上的數點結論，我們可以確定 EMonitor 是可以用來取代 Java monitor 來設計 concurrent object 的一種實用的解決方案。

六、參考文獻：

[And91] G. Andrews, *Concurrent Programming - Principles and Practice*, The Benjamin/Cummings Publishing Company, Inc., 1991.

[BFC95] P. Buhr, M. Fortier, and M. Coffin, "Monitor Classification," *ACM Computing Surveys*, vol. 27, no. 1, pp. 63-107, March 1995.

[Bro98] B. Brosgol, "A Comparison of the Concurrency Features of Ada 95 and Java," *Proc. of ACM SIGAda Annual Int'l Conf. on Ada Technology*, pp. 175-192, 1998.

[GJS96] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[Geh93] N. Gehani, "Capsules: A Shared Memory Access Mechanism for Concurrent C/C++," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 795-811, July 1993.

[Hoa74] C. Hoare, "Monitor: An Operating System Constructing Concept," *CACM*, vol. 17, no. 10, pp. 549-557, 1974.

[Kot87] L. Kotulski, "About the Semantic Nested Monitor Calls," *SIGPLAN Notices*, vol. 22, no. 4, pp. 80-82, 1987.

[LR80] B. Lampson and D. Redell, "Experience with Processes and Monitors in Mesa," *CACM*, vol. 23, no. 2, pp. 105-117, Feb. 1980.

[OH95] R. Olsson and C. McNamee, "Tools for Teaching CCRs, Monitors, and CSP Concurrent Programming Concepts," *SIGCSE Bulletin*, vol. 27, no. 2, pp. 31-40, June 1995.

[OH96] R. Olsson and C. McNamee, "Experience Using the C Preprocessor to Implement CCR, Monitor, and CSP Preprocessors for SR," *Software - Practice and Experience*, vol. 26, no. 2, pp.125-134. Feb. 1996.

[OW97] S. Oaks and H. Wong, *Java Threads*, O'Reilly & Associates, Inc., 1997.

[SCH95] S. Stubbs, D. Carver, and A. Hoppe, "IPCC++: A Concurrent C++ Based on a Shared-Memory Model," *Journal of Object-Oriented Programming*, vol. 8, no.2, pp. 45-50, 66, May 1995.

[SLS82] K. Shin, Y. Lee, and J. Sasidhar, "Design of HM^2p - A Hierarchical Multiprocessor for General-Purpose Application," *IEEE Trans. on Computers*, vol. C-31, no. 11, pp. 1045-1053, Nov. 1982.

[YH97] S. Yuan and Y. Hsu, "Design and Implementation of a Distributed Monitor Facility," *Computer Systems Science and Engineering*, vol. 12, no. 1, pp. 43-51, Jan. 1997.

[VA98] C. Varela and G. Agha, "What after Java? From Objects to Actors," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp.573-577, 1998.