分散式共享記憶體的容錯與執行緒同步機制之研究 (I)

計畫編號 ：NSC88 - 2213 - E009 - 087
執行期間 ： 87 /08 /01 -- 88 /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，提供分散式共享記憶體一個新的程式設計界面。因為我們所採用的程式語言是 Java，所以，我們今年的主要研究重心放在，改進 Java 的執行緒同步機制上。Java 本身就有提供一個同步機制，但是它並不能真正的做到完全同步，因為其所提供的機制功能太少，常常要花很多的功夫才能達到所要的目的，而且對於一個程式設計者，他必須考慮很多細節才有辦法開發出不會有 deadlock 不會有 race condition 的同步應用程式。雖然目前在 Java 上已有一些解決的方法可以讓程式設計者很方便的開發同步應用程式，但是其最大的缺點則是效率太差。Monitor 是已經在同步機制上應用很久的技術，Java 本身的同步機制也是利用一些這種概念，但是由於是個簡化的 monitor，所以用起來很明顯的不足，無論在找錯誤或者維護程式方面，使用起來不方便。這篇論文將會先敘述 Java 在同步應用程式上缺點以及其他的目前的解決方案，並且將介紹如何用我們所開發的 monitor 在程式語言層次上改進 Java 的同步機制以及說明如何用所提供的函式庫與前置處理器來幫助程式設計者開發同步應用程式，最後將會與其他的解決方案做各方面的比較。

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

There are many synchronized mechanisms announced now, and different platforms or programming languages have their own suitable solutions. But there are no proper solutions for Java. Java has its own synchronized mechanism, but it is not truly concurrent, it supports a limited form of currency. A programmer must take care of many details when developing concurrent applications. Even there are some solutions are designed for Java, and also let a programmer can develop concurrent applications more conveniently, but the poor performance is the common problem. Monitor is a technique used in concurrency control for a long time, Java synchronization mechanism also take advantages of this concept a little, but is not enough. This paper will declare the disadvantage of other solutions and will also introduce an approach for enhancing this mechanism in programming level bye use of our monitors. We will provide libraries and a preprocessor that are helping for concurrency control. Finally, we will compare our solution with others in performance and the complexity of coding style.

二、計畫緣由與目的

在 Java 上其本身的同步機制在運作上只有兩個動作，一是"wait"另一個是"notify",有如一個簡單的 monitor，而這兩個動作只能在宣告成 synchronized 的 method 或 block 裏才可以有效的運作。Wait 與 notify 這兩個 method 是繼承於 Object 這個物件，也就是說任何一個物件裏的 method 或 block 都可以被宣告成 synchronized，當有執行緒要使用此物件時，必須先拿到此物件的"lock"才可以使用，否則就會被停滯直到此物件的"lock"不屬於任何 thread 時才會繼續動作。因為此機制是個簡單的同步機制，所以使用起來不太方便，不論在寫程式上或者 debug 上都很麻煩，而且也容易有死結的發生，會造成這種現象的主要原因有四個；一、每個物件只有一種 condition variable，二、作 inter-nested 的 synchronized method 呼叫時，"lock"不會被釋放，三、只能作非停滯性的 signal，四、正在 wait 的執行緒無法給與優先度以及某種標記。

在目前 Java 上的解決方案中，Guard-based system 是比較廣泛用到的技術，如 CJava，CORRELATE，以及 active object [8,12,13,14,15]，其優點是撰寫程式方便，維護容易，死結不易發生，可是效率太差是此系統最大的缺點。此系統之所以效率會差是因為它必需有個執行緒自動的不斷的去檢查與運算目前的狀態，此執行緒有可能是正在作某個工作的執行緒或者是以一個單獨執行緒當作 daemon，專門用來計算檢查系統狀態，此計算的結果將會被用來做為下一個動作的依據，此系統另一個主要的討論重點在於何時去做檢查與計算的動作，不過無論是用 busy waiting 或者是用 wait-notify 的方式來做檢查與計算，其所需要花費的 context switch 一定會不少，所以無論是那一種機制，只要是利用 guard-based system 來設計的，其效率都不會很好。

現在已經有許多的同步機制發表於世，monitor 是其中一個運作很好的機制，其主要的特點為在同一個時間內，只會有一個執行緒可以在 monitor 內執行，若有執行緒要進入 monitor 的時候，已有其他的執行緒正在執行它的工作，此時這個尚未進入monitor 的執行緒就要被停滯，直到在monitor 內的執行緒釋放monitor lock 後才可以進入。Monitor 的另外一個特色則是它有狀態變數(condition variable)以及與其相關的兩個動作"wait"與"signal"，當在monitor 內的執行緒發現內部的狀態無法再繼續執行下去時，就會停滯在某個狀態變數並且釋放出 lock 的擁有權直到被其他的執行緒叫醒(signal)。

之所以會選則monitor 是因為 Java 本身的同步機制就是利用monitor 的概念而成的，在觀念上也很容易懂，並且在效率方面其 context switch 也遠比 guard-based system 還要小的多，所以此計畫主要就是延伸 Java 本身類似 monitor 的同步機制，使其可以有多個狀態變數，並且可以做 inter-nested 的呼叫而不會有死結的發生，如此若以 Java 來開發並行性的應用軟體，則無論在寫程式的難易度上或者在除錯方面都會比用 Java 原來所提供的機制還來的好，甚至其效率方面也不會比較差。

三、結果與討論

根據[5]裏我們可以知道一個基本的 monitor 會有四種不同優先度的 queue 用來依情況存放不同的執行緒，這四種 queue 分別為 entry-queue，condition-queue，signaler-queue 以及 waiting-queue，其中可執行的 queue 有三種，分別為 entry-queue，signaler-queue，以及 waiting-queue，我們可以跟據這三種 queue 做出許多種不同的 monitor，可是為了避免饑餓(starvation)形式的死結，在被 signal 之後其執行緒可以在 monitor 狀態不被更動到的情況下很快的起來執行，以及必需能讓在 monitor 內部的執行緒可以盡快的結束執行而釋放 monitor lock，有兩條規定必需遵守，一是每個 monitor 內其 queue 之間的優先度必需要固定，另一個則是 entry queue 的優先度必需是所有 queue 之間最低的，所以真正使用的 monitor 只有兩種，一種是 blocking monitor 另一種則是 non-blocking monitor。

Blocking monitor 指的是在 monitor 內的執行緒在 signal 某個狀態變數後，要馬上讓被 signal 的執行緒起來執行，自己則要停滯在 signaler queue 裏，所以這三種 queue 的優先度為 Wp>Sp>Ep，(表示 waiting queue 優先度高於 signaler queue 高於 entry queue)。相反的 non-blocking monitor 則是在 signal 某個狀態變數後要把停滯在 condition queue 裏執行緒移到 waiting queue 裏等待重新執行，而自己則可以繼續在 monitor 執行其所要的工作，所以其優先度為 Sp>Wp>Ep。此兩種 monitor 各有各的特點，在 blocking monitor 裏，在 monitor 內的某個狀態已經發生的時候，在某個 condition queue 裏等待此狀態的執行緒可以不經過檢查而馬上起來執行，而在 non-blocking monitor 裏，正在執行的執行緒可以一次 signal 許多的狀態變數而不需要停滯，不過在等待此狀態的執行緒在準備重新進入 monitor 時通常必需要做檢查的動作以確定此 monitor 狀態是否有被改變，若沒有則可以放心的進入 monitor，否則就必需重新在停滯在某個狀態變數。我們可以視需要選則一個適當的 Monitor，如 context switch 比較少的而且確定狀態不會被改到的就可以選則 non-blocking monitor。

在 nested monitor call 方面，有兩種形式[2]:

1. Intra-nested monitor call
   同一個物件的 monitor lock 被同一個執行緒要求許多次
2. Inter-nested monitor call
   要求不同的 monitor lock
   a. Open call
      當要離開此 monitor 而進入令一個 monitor 之前，要釋放之前所拿到的 monitor lock。
   b. Close call
      在拿到不同的 monitor lock 後，只有必需停滯在某個狀態變數的時候，才會將所拿到的 lock 全部釋出。

第一種的 nested call 只要將 lock counter 加一就可以了，而第二種 nested call 在離開某個 monitor 之前就要計錄其 counter，由其是 close call 更要記錄其 monitor 物件，等到結束 nested call 返回原來的 monitor，再把離開此 monitor 之前的狀態回復。不過在返回原先的 monitor 時，有可能已有其他的執行敘正在此 monitor 內，故不能馬上反回進入此 monitor，必需在一個 queue 裏暫存著，因此在與之前的四種 queue 來一起看，我們的 monitor 總共有五種 queue，除了 condition queue 外其他四種 queue 都是可執行 queue，也是有兩種 monitor 是有用的，不過在加入暫存的 queue 之後，其優先度則變成如下表所示:

Signal Characteristics	Priority
Blocking	Signal and Urgent Wait Ep < Sp < Rp < Wp Priority Blocking (PB)
Non-Blocking	Signal and Continue Ep < Rp < Wp < Sp Priority Non-locking (PNB)

其中 Rp 表示新增加的 queue 的優先度，之所以會這樣安排其理由為在結束 inter nested monitor call 後，此執行緒能越快回來原先的 monitor 越好，但是又不能馬上無條件的回去此 monitor，以免發生不一致的狀況，所以此 queue 的優先度 Rp 一定要低於 Wp，若能高於 Sp 的話就盡量高於 Sp，所以就會形成此表的狀況。

四、研究成果自評

本計畫所做出來的結果，有許多與其他各項的解決方法做個比較，其所得到的數據顯示可看出我們的方法優於目前的其他方法，無論是在效率方面或者程式撰寫的難易度上，預計會將結果在一年內發表於國外的相關期刊上。

五、結論

任何須要分散式計算環境的系統其原因不外乎是單機的系統其所表現的效率無法滿足使用者，所以我們沒有理由還要去使用效率比較不好的機制，但是目前在 Java 上的解決方案，若是要求要有好的效率，則其寫程式的難度會增高，除錯不易，反過來若是要求能夠方便的撰寫程式，能夠寫出易懂的程式，則其效率通常不會很好，因此我們改良目前 Java 所提供的同步機制，在程式語言的層次上提供函式庫來讓程式設計者使用，以達到方便撰寫以及不錯的效率，除此之外使用者也可以在不改變使用習慣下，如同使用原來 Java 所提供的同步機制一樣，繼續使用我們所設計的 monitor。

六、參考文獻：

[1].Scott Oaks and Henry Wong, "Java Threads", O'REILLY & Associates, Inc. 1997

[2]. Andrews, Gregory R. "Concurrent programming, principles and practice", New York Benjamin Cummings, 1991

[3].Andrews S. Tanenbaum. "Distributed Operating Systems", Prentice-Hall, Inc. A Simon & Schuster Company, 1995.

[4]Andrews S. Tanenbaum. "Modern Operating Systems", Prentice-Hall, Inc. A Simon & Schuster Company, 1992.

[5].Peter A. Buhr and Michel Fortier. "Monitor Classification," ACM Computing Surveys, Vol. 27, No. 1, March 1995. pp. 69-73, pp. 89.

[6].Mukesh Singhal. "Deadlock Detection in Distributed Systems," IEEE Computer, November 1989. pp. 37-40

[7].Young Man Kim; Ten Hwang Lai; Soundarajan, N. "Efficient distributed deadlock detection and resolution using probes, tokens, and barriers", IEEE Parallel and Distributed Systems, 1997. Proceedings. pp. 584-590.

[8].Martin Carroll, "Active Objects Made Easy," PRACTICE AND EXPERIENCE, Vol. 28(1), 1-21, January 1998. pp. 1-5, pp. 13-20.

[9].Jose R. Gonzalez de Mendivil, Federico Farina, Carlos F. Alastruey. " A safe distributed deadlock resolution algorithm", Journal of Systems Architecture 1998. pp. 887-891

[10].Miyoshi, A.; Kitayama, T.; Tokuda, H. "Implementation and Evaluation of Real-Time Java Threads", Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE, pp. 166-173

[11].Demartini, C.; Sisto, R. "Static analysis of Java multithreaded and distributed applications", IEEE Software Engineering for Parallel and Distributed Systems, 1998. Proceedings. pp. 215-221.

[12]. Carlos A. Varela, Gul A. Agha. "What after Java? From objects to actors", Computer Networks and ISDN Systems 1998. pp. 573-576.

[13]. Gianpaolo Cugola and Carlo Ghezzi. "Cjava: Introduction concurrent Objects in Java" Dipartimento di Elettronica e Informazione, Politecnico di Milano.

[14]. Chao-Hsin Lin; Elrad, T. "An enhanced reflective architecture for adaptation of the object-oriented language/software", IEEE Software Engineering Conference, 1998. Proceedings. pp. 20-24.

[15]. Ciaran McHale "Synchronization in Concurrent Object-oriented Languages: Expressive Power, Genericity and Inheritance", A thesis submitted to University of Dublin, Trinity College.

[16] http://java.sun.com/docs/books/jls/html/index.html

[17]. C. H. Wu "Enhancing Thread Synchronization Mechanism of Java" Master Thesis, Institute of Computer and Information Science, College of Electrical Engineering and Computer Science, National Chiao Tung University, June 1998.