ISSN1000-9825 Journal of Software 软件学报 2000,11(5):701~706

Java 虚拟机用户级多线程的设计与实现*

丁宇新 程虎

(中国科学院软件研究所 北京 100080)

E-mail: chenghu@126.com

摘要 详细介绍了国产开放系统平台 Java 虚拟机多线程的设计与实现.在线程调度上,采用带有独立队列的静态级别轮巡调度,较好地解决了独立循环线程的调度问题.对于线程的同步,采用了哈希混合锁的设计方案.实验结果证明,该锁具有空间小、执行效率高等特点.

关键词 Java,虚拟机,线程.

中图法分类号 TP312JA

线程是程序中独立运行的控制流,与进程相比,不同线程可共享同一地址空间,系统开销小.多线程作为Java语言的组成部分而成为软件开发人员描述程序并发行为的有利工具.Java 虚拟机(Java virtual machine,简称 JVM)是运行 Java 语言的工具,它负责线程的建立、执行、切换、终止以及同步.JVM 中线程的实现可利用操作系统的线程,也可使用用户级的线程库,或者采用两者相结合的方式.SUN 最初在 Solaris 操作系统上实现的虚拟机采用的是用户级的线程库,以后改用内核线程和用户线程相结合的方式;Linux 系统上的虚拟机 kaffe 采用的是纯用户级的线程;Krall 等人在为 DEC Unix 平台设计 64 位 Java 虚拟机时[1],采用的也是用户级的线程.

采用用户级的线程一般是基于以下几个因素:

(1) 操作系统本身不支持多线程.

(2) 不同操作系统线程的实现机制不同.不仅操作系统线程 API 不同(目前,大部分操作系统采用 IEEE POSIX 10003.1c 标准[2]),而且不同平台其线程运行的效率相差很大.采用用户级的线程可使程序具有良好的移植性.

(3) 可根据 JVM 自身的特点建立最适合的线程库.

下面就国产开放系统平台 COSIX(Chinese opening system-UNIX)Java 虚拟机用户级线程的设计与实现进行讨论.

1 多线程的设计与实现

1.1 线程的数据结构

Java 程序在虚拟机中以线程的方式运行,在应用程序被执行前,虚拟机自动为其创建一个名为 main 的主线程,在以后的运行过程中,虚拟机负责该线程的管理,因此,线程是虚拟机中重要的数据结构.此外,Java 语言提供了线程类供用户创建自己的线程.无论是 JVM 系统线程,还是用户程序创建的线程,两者在虚拟机内部都被看成是统一的线程对象.因此,在线程数据结构定义上,两者应保持一致(具体数据结构见 Java 线程类 API),以便于虚拟机对线程进行统一的管理与维护.

虚拟机接管对所有线程的控制,例如,何时唤醒线程以及线程运行空间的切换等.因此,只有线程数据结构

* 本文研究得到国家"九五"重点科技攻关项目基金(No.96-737)资助.作者丁宇新,1972 年生,博士,主要研究领域为计算机语言及其编译,神经网络.程虎,1938 年生,研究员,博士生导师,主要研究领域为计算机语言及其编译,软件工程,人工智能,神经网络.

本文通讯联系人:程虎,北京 100080,中国科学院软件研究所

本文 1999-03-23 收到原稿,1999-05-26 收到修改稿

是不够的,还应为此建立相应的线程背景信息.线程背景信息应记录线程空间的内存位置、相关的线程队列信息、线程的恢复执行点以及自身的状态等内容.

我们可以利用线程类的实例变量 PrivateInfo 来建立线程与其背景信息的对应关系.在 JVM 中,线程栈空间可由虚拟机运行时的命令行参数给出,也可由虚拟机按缺省值分配固定的空间.

1.2 线程的调度

在 Java 虚拟机规范中并未规定线程的调度策略[3].一般的线程调度可采用如下几种策略:

(1) FIFO(first in first out)调度.该调度策略采用先来先服务的方式,线程被无条件地执行,直到其运行完毕或被堵塞为止.该方法一般适用于运行时间短且对时间敏感的线程的调度.

(2) 静态级别轮巡调度.在该策略下,优先级最高的线程被抢先执行,只有该线程执行完毕或被堵塞,其他线程才能被执行.优先级相同的线程按分时原则轮流执行.目前,AIX,DCE 操作系统采用该策略进行线程调度.

(3) 动态级别轮巡调度.在该策略下,线程的先先级按其执行的次数和时间进行动态调整.目前,Windows NT 及 OS/2 的线程调度采用该策略.

Java 语言规定了线程的优先级,线程的最高优先级为 10,最低为 1.因此,在虚拟机中,一般采用后两种调度策略.目前,Java 虚拟机主要以 Sun 公司和 Microsoft 公司的产品为主,它们的虚拟机都是利用操作系统的线程来实现 Java 线程.因此,JVM 线程调度策略依赖于具体的操作系统.如 Sun 公司在 Sparc 上的虚拟机线程调度策略采用简单的优先权调度[4],Microsoft 公司为 Windows 操作系统开发的虚拟机则采用动态级别轮巡调度[5].

对于用户级的线程,设计者可根据实际情况设计合适的调度算法.FIFO 调度算法因其简单且无线程的频繁切换而执行效率较高,但它没有考虑到线程的优先级,因而应用受到一定的限制.静态级别轮巡调度虽然考虑到线程的优先级,但由于其严格按优先级的次序调度可能会使低优先级的线程长时间地处于等待状态而不能被执行.动态级别轮巡调度使各线程都有被执行的机会,但线程切换较为频繁,因此执行效率较低.

我们在程序设计时经常会设计这样的线程——它们相对独立且无条件地重复执行,如主页上的动画循环播放程序,对于这样的线程采用静态级别轮巡调度显然是不合适的,因为这有可能使低优先级的线程永远得不到服务;采用动态级别轮巡调度又会降低运行效率.为此,我们在线程运行队列设计时,单独建立了一个独立线程运行队列,该队列的优先级标志为 0.独立线程队列无固定的级别,它与当前最高级别线程队列中的线程被调度程序分时调度.不难看出,当独立线程队列为空时,该调度算法就是静态级别轮巡调度.

该设计的主要目的是解决独立循环线程的调度问题,独立循环线程可放入独立队列.这样,既不会发生较低级别的线程永远得不到执行的情况,也能满足循环线程自始至终都被运行的条件.在运行性能上,我们可证明分时执行线程数目越少,系统的执行效率越高,一般可节省若干个线程切换时间.该调度策略的分时线程数目仅限于独立队列与当前最高级别线程队列,分时线程数目介于静态级别轮巡调度与动态级别轮巡调度之间,因此性能也介于两者之间.

1.3 线程同步

提到多线程,不可避免地涉及到线程之间的同步问题.Java 语言提供了同步语句与同步方法协调线程对共享资源的使用.在虚拟机中,每个 Java 对象及类都有一个与其相关联的锁,当线程访问共享资源时,该线程必须获取共享资源所在对象(或类)的锁,若该锁已被其他线程占有,则该线程必须进入锁的等待队列,等待该对象(或类)锁被释放.当已获取对象锁的线程释放共享资源时,同时释放该锁,若该锁的等待队列非空,则使第 1 个等待线程获取该锁.在 Java 语言中,同一线程可对同一对象(或类)多次加锁.

在实际中,一般用下面的数据结构实现对象的互斥锁.

```
Lock{struct thread * holder;      //当前占有对象的线程
int      count;                   //占有对象的线程对对象的加锁次数
struct thread * mutexWaiters;     //等待对象锁的线程队列
}
```

锁在线程的设计中占有重要的地位,有效的设计方法不仅可以改善程序的运行效率,也可大大节省程序内

存的需求.目前,JVM 中锁的设计主要有以下两种典型代表,现分别叙述如下.

1.3.1 哈希锁

哈希锁是 Krall 等人在为 DEC UNIX 平台设计 64 位 Java 虚拟机时提出的设计方法[1],它具有节省内存、运行效率较高等特点.

在 Java 虚拟机规范中,每一对象都配有一对象锁,由此可见,对象锁在内存中会占据相当一部分空间.若将线程的数据结构用整数代替,则锁的空间缩至 6 个字节,即便如此,Javac 编译程序中锁的空间也要占据近 1 兆字节.实际应用中,在某一时刻,全部被加锁的锁数目一般很少多于 20,这也就是说,在 JVM 中,我们只需保留 20 把锁就可以满足实际的需求,为此可设计一个锁哈希表,对象锁存放在哈希表中,哈希表中的锁随着线程对对象的加锁而建立,当线程释放锁后,该锁在哈希表中被取消,锁的定位通过哈希表来进行查找.为了提高加锁、解锁的效率,加锁、解锁操作的代码应尽可能短,必要时应使用汇编语言.

1.3.2 薄锁(thin lock)

薄锁是 IBM T.J.Watson 研究中心在为 AIX 操作系统开发的基于 IBM JDK1.1.2 Java 虚拟机时提出的设计方案[2],它具有加锁、释锁快速,占用内存小(只需 24 位),易维护等特点.

线程的锁操作一般可分为 5 种情况:

(1) 对未被占用的对象锁加锁;

(2) 线程对一个对象加锁若干次(注:次数较少);

(3) 线程对一个对象加锁多次;

(4) 线程因访问冲突而进入对象锁的等待队列,此时,等待队列为空;

(5) 多个线程同时等待某一对象锁.

IBM T.J.Watson 研究中心在对大量的多线程程序测试后发现,80%的对象锁操作属于第 1 种情况,且 80%的对象锁的加锁次数较少,一般不会超过几十次.因此,他们提出了薄锁的设计方案[2].他们采取一定的编码方式在对象数据前空出 3 个字节作为对象锁,其中第 1 位为锁标志,若该标志为 0,指示该锁为薄锁,则该位后 15 位为线程标志,最后 8 位为加锁次数.若锁标志位为 1,表明在运行过程中第 3、4、5 种锁操作情况出现,此时应进行锁转换,将薄锁换为厚锁(fat lock)(厚锁的数据结构见第 1.3 节).薄锁一经变为厚锁就不再作相反的转换,在这种情况下,锁标志位的后 23 位为厚锁在厚锁表中的索引.采用薄锁使得 80%的锁操作(第 1 种情况)只用少数硬件指令就可完成加锁或解锁操作,这些硬件指令是不可分的,如 IBM 370 系列中的"比较与对换"(compare and swap)指令,Intel i386 的"交换"(exchg)指令,正是由于采用了硬件指令,才使得锁操作的效率大大提高.薄锁的执行效率是传统互斥锁执行效率的 3.7 倍,详细的测试结果可见文献[5].

1.3.3 哈希混合锁

哈希锁与薄锁各有其特点,哈希锁的主要优点是锁占用的空间少,可大大节省内存,薄锁的特点是大部分的加锁、释锁操作可用不可分的机器指令完成,不仅执行效率高,而且安全.那么,如何将两者结合呢?我们在设计国产操作系统 COSIX 虚拟机时提出了哈希混合锁的设计方案.

我们的设计方案将锁与对象数据分离,采用上节哈希锁的设计方法,将锁存放在锁哈希表中.采用两级哈希表结构,第 1 级是对象到薄锁哈希表的映射,第 2 级是薄锁到厚锁哈希表的映射.两级哈希表的数据结构如下所示,fatLock 的数据结构见第 1.3 节.

第 1 级哈希表:

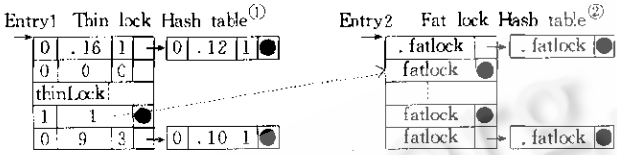
```
struct entry1{
    object *obj;   //对象指针
    int32 thinLock;   //为对象分配的薄锁
    struct entry* next;
    }
```

第 2 级哈希表:

```
struct entry2{
    thinLock *tLock;   //薄锁指针
    struct fatLock fLock;   //为薄锁分配的厚锁
    struct entry2* next;
}
```

在程序执行过程中,JVM 为首次申请加锁的对象在薄锁哈希表中分配薄锁,薄锁 thinLock(32 位)的首位为

锁标志,该位若为0,则其后面的21位为线程标识,线程标识后10位为加锁次数;若锁标志位为1,则其后面的31位为厚锁在厚锁哈希表中的哈希值(注:在实际中可能只用其中几位,可视厚锁哈希表的大小而定),当对未被线程占用的对象加锁时,可利用简单的机器指令完成薄锁加锁操作,若第1.3.2节中的情况3、4、5发生或wait,notify,notifyall等操作作用于该对象时,则必须进行锁的转换.在厚锁哈希表中建立该对象的厚锁.两级哈希表的数据结构示意图如图1所示.为提高效率,各哈希表的入口锁被事先分配,并被初始化,运行过程中不被释放.锁的哈希值取对象或薄锁地址的最低n位,哈希表的大小为2^n.



①薄锁哈希表,②厚锁哈希表.

Fig. 1
图1

哈希混合锁的加锁算法

线程对哈希混合锁的加锁算法可描述如下:

```
lockMutex thin(object* obj)
{  entry1* mutex1;  //薄锁
   entry2* mutex2;  //厚锁
   mutex1=findOrCreate thinlock_for_obj(obj);  //查找obj的薄锁,若没有,为其新建
   success1=lock thin1(&mutex1->thinLock,currentThreadID);  //情况1加锁
   if (success1) return;
   else {  success2=lock_thin2(&mutex1->thinLock,currentThreadID);  //情况2加锁
         if (success2) return;
         else {if (is thinLock(mutex1))  //是否需进行锁转换
                  mutex2=thinLock2fatLock(&mutex1->thinLock);  //薄锁换厚锁
               else mutex2=find_fatlock_for_obj(&mutex1->thinLock);  //查找厚锁
               lockMutex(&mutex2->fLock);  //厚锁加锁
               }
      }
}
```

在上述算法中,加锁算法首先在薄锁哈希表中找到对象所对应的薄锁,若没找到,则在哈希表中为对象分配薄锁,之后执行对未被线程占有对象的加锁操作(情况1).若加锁成功则返回,否则尝试对情况2的加锁.若成功则返回;若不成功,说明加锁情况3、4、5出现.此时,若薄锁未被转换成厚锁,则应进行锁的转换;否则,在厚锁哈希表中找到该锁.最后对该锁加锁.lock_thin1(int*,int)可用简短的汇编语言实现,由于80%的加锁操作都是属于情况1,因此,加锁效率与传统锁相比会有很大提高.

哈希混合锁的释锁算法

线程对哈希混合锁的释锁算法可描述为:

```
unlockMutex_thin(object* obj)
{  entry1* mutex1;  //薄锁
   entry2* mutex2;  //厚锁
   mutex1=findOrCreate_thinlock_for_obj(obj);  //查找obj的薄锁
   success1=unlock_thin1(&mutex1->thinLock,currentThreadID);  //情况1释锁
   if (success1) return;
```

```
else {  success2=unlockThin2(&mutex1->thinLock,currentThreadID);  //情况 2 释锁
        if (success2)
        {  if (mutex1->thinLock.count==0)  //是否取消该锁
             release thinLock(mutex1);  //在哈希表中取消该锁
           return;
        }
      else { mutex2=find fatlock_for_obj(&mutex1->thinLock);  //查找厚锁
             unlockMutex(&mutex2->fLock );  //厚锁释锁
             if (mutex2->fLock.count--0)
              release fatLock(mutex2);  //在哈希表中取消该厚锁
           }
      }
;
```

在上述算法中,释锁算法首先在薄锁哈希表中找到对象所对应的薄锁,之后执行对未被线程占有的对象的释锁操作(情况 1).若释锁成功则返回,否则尝试对情况 2 的释锁.若成功则返回,返回前判断该锁是否有必要被取消;若不成功,说明该锁为厚锁,在厚锁哈希表中找到该锁,并释放该锁.释锁后,判断是否有必要在哈希表中取消该锁.哈希表的入口锁不被释放.unlock thin1(int*,int)也可用简短的汇编语言实现.因此,释锁的效率也比传统锁要高.上述算法中的 lock thin2,unlock_thin2 是对本线程已占有对象的加锁和释锁操作(情况 2),它们的使用频率也较高,也可用汇编语言来实现.

我们在薄锁的加、释锁算法中,没有考虑线程数目越界及线程加锁次数越界的问题,这是因为,在薄锁中为线程标识保留 21 位,加锁次数保留 10 位,足以满足实际的需要.

1.3.4 实 验

由于 IBM 薄锁的设计方案较难实现,主要是因为文献[2]未给出薄锁是采用何种编码方式存放在对象中,因此,我们只将哈希混合锁的性能与哈希锁进行比较.测试方式是对同一方法在同步方式和非同步方式下分别调用 30 万次,在 Java 语言中,在同步方法执行前,线程必须先获取该方法所在对象的对象锁,方法执行完毕后再释放对象锁,我们在这里测试的方法为空方法,执行时使用及时编译器,测试环境为 COSIX v1.3,PC586/133.实验数据见表 1.

Table 1
表 1

Object lock①	Run time in secs②		Call cost in μs③
	No sync.④	Sync.⑤	
Hash lock⑥	0.22	0.4	0.6
Mixed Hash lock⑦	0.22	0.3	0.27

①对象锁,②运行时间,③平均调用时间,④非同步,⑤同步,⑥哈希锁,⑦混合哈希锁.

2 总 结

本文详细介绍了国产开放系统平台 Java 虚拟机多线程的设计与实现,在线程调度上,我们采用了带有独立队列的静态级别轮巡调度,较好地解决了独立循环线程的调度问题.对于线程的同步,采用了哈希混合锁的设计方案,我们认为,这是基于锁的执行效率与空间分配考虑的一个可行的折衷方案.实验证明,该锁具有空间小、执行效率高等特点.

多线程设计只是虚拟机设计的一个方面,虚拟机整体性能的提高依赖于它的各个组成部分,其中包括无用单元回收、例外处理、尤其是解释器与及时编译器的设计.目前,我们已基本完成了国产开放系统平台 Java 虚拟机与编译器的开发工作,我们也希望能就其中的关键技术与读者共同探讨.

参考文献

1 Krall A. AND probst: monitors and exceptions: how to implement Java efficiently. In: Hennessy J L ed. Proceedings of the ACM Workshop on Java for High-Performance Network Computing. Atlanta, GA: ACM Press, 1998. 100~106

2 David F B. Thin locks: featherweight synchronization for Java. ACM SIGPLAN on Programming Language, 1998,33(5): 259~268

3 Lindhdm T. The Java Virtual Machine Specification. Berkeley, CA: Addison Wesley Longman, 1996

4 Stevens W R. UNIX Network-Programming Network APIs. 2nd Edition. Hpper Saddle: Simon & Schuster Company. 1998. 601~633

5 Horstmann C S. Core JAVA (V. 1. 1) —— Advanced Features. Palo Alto: Sun Microsystems, 1998

Design and Implementation of User Level Multi-Threads in Java Virtual Machine

DING Yu-xin CHENG Hu

(*Institute of Software The Chinese Academy of Sciences Beijing* 100080)

Abstract In this paper, the authors discuss the design and implementation of Java multi-threads. A new thread scheduling algorithm, named Preemptive Round-Robin Scheduling with a Free Queue, is presented. Under this policy threads in the free queue are not assigned with a constant priority. Their priority is the same as the highest-priority threads which are in running state. The highest-priority threads and the threads in the free queue are time-sliced scheduled. This algorithm solves the scheduling problem for independent looping thread. To improve the efficiency of thread synchronization, a new design for object lock is presented. It is called the Mixed Hash Object Lock. This design is a tradeoff between the lock efficiency and its space. The experimental results have proved that the design is feasible. Compared with the traditional design, the efficiency for locking and unlocking is high and the space allocated to lock is small.

Key words Java, virtual machine, thread.