⑧289-292

第46卷 第3期
2000年6月

武汉大学学报(自然科学版)
J. Wuhan Univ. (Nat. Sci. Ed.)

Vol. 46 No. 3
June, 2000, 289～292

文章编号：0253-9888(2000)03-0289-04

一种基于锁集的多线程数据竞争的动态探测算法

李克清，陈莘萌，郑无疾

（武汉大学 数学与计算机科学学院；武汉大学 软件工程国家重点实验室，武汉 430072）

TP311.1

摘 要：通过对多线程环境下共享变量运行状态变化的分析，提出了一种基于锁集的多线程数据竞争的动态探测算法. 该算法在探测共享变量间的读写冲突方面具有一定的实用价值. 它能充分地发掘出程序中的并行性，使处理机得到最大限度的利用.

关 键 词：锁；数据竞争；多线程程序设计

中图分类号：TP 311.1；TP 311.5　　**文献标识码**：A

动态探测算法

0 引 言

传统的多用户、多任务操作系统（如 UNIX，WINDOWS 95/NT 等）是多进程系统，这种多进程系统普遍存在着如下问题：1）进程间信息交换复杂；2）进程间切换涉及到多种资源，管理开销大，耗时长，每个进程在执行自己或其他进程的时候，都要占用资源，造成包括内存在内的资源浪费；3）由于进程独享资源空间，不利于数据与代码的共享，使编程复杂化.

多线程程序设计是近几年提出的一种新概念. 通过多线程实现程序的并发运行，既可以提高程序的性能，又能增强程序的功能[1]. 多线程在程序中有多条执行路径，这些不同的执行路径可以并发地工作. 线程的提出，屏蔽了计算机并发执行的底层原理，使程序设计人员只须从软件的角度进行系统分析，而不必考虑其他的细节. 因此多线程相对于进程来说简单得多，程序中线程的行为就像是函数，可以通过全局变量交换信息，也可以共享内在文件等资源.

多线程已经成为一种普遍的程序设计技术，这种技术不仅被广泛地运用于操作系统的设计，而且在很多应用程序的设计中也被大量地使用. 如 Netscape Navigator 和 Microsoft Word 就采用了多线程的程序设计技术.

1 线程间的数据竞争

线程（thread）是一个顺序执行的指令序列，仅有一个入口、一个出口. 它是一个程序内部的顺序控制流. 从逻辑上看，多线程意味着一个程序的多个语句同时执行；多个线程共享数据空间，以避免进行无谓的数据复制，同时每个线程也都有自己的执行堆栈和程序计数器作为其执行的上下文. 多线程下的进程结构如图 1 所示.

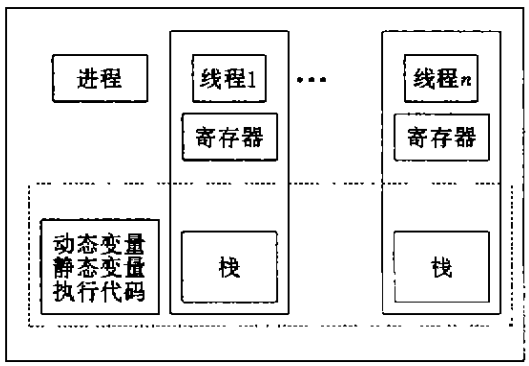

图1 多线程进程结构

在一个进程内有多个线程，它们并发执行，在任何时刻都有多个执行点，线程在共享的地址空间内执行，读写相同的内存变量，因此会不可避免地引起

收稿日期：1999-12-27
作者简介：李克清（1966-），男，硕士生，现从事分布并行处理，计算机网络研究.

线程间的数据竞争.两个并发执行的线程产生数据竞争的时机为:

①至少有一个线程是写操作;

且 ②另外一些线程没有采用明确的机制来阻止其他线程的同时访问.

如果一个程序内部本身存在着数据竞争,那么对共享变量的访问是否发生冲突就依赖于线程执行过程中的重叠交叉部分.尽管程序员有时也允许这种看似无害的非确定性的数据竞争,但如果不能正确地进行线程间的同步的话,可能会导致严重的执行错误.

在早期的计算环境下,很多系统是采用管程(monitor)的方法来解决数据竞争的[2],它要求程序员在编程时将一些要共享访问的数据放在临界区(critical)内,任何进程要访问临界区,必须先获得该临界区的同步锁,而在离开临界区时将该同步锁释放.任何未获得同步锁的进程必须暂停其运行以等待获取该同步锁.也就是说,只有拥有该临界区锁的进程才能访问这些共享变量.管程提供了静态的,编译期间的避免数据竞争的解决方案,但它却无法解决程序运行期间动态分配的共享变量的数据竞争.

在分布式计算环境下,对动态数据竞争的探测方法大都是基于 Lamport 的 HB 关系[3,4](Happens Before relation,记作→).HB 关系是一个偏序关系,它具有如下特征:

① 如果事件 x,y 在同一线程中发生,且 x 在 y 之前发生,则有 $x\rightarrow y$.

② 如果存在消息 m,则有 send(m)→receive(m).

③ 对于任意的三个线程 x,y,z,如果 $x\rightarrow y,y\rightarrow z$,那么 $x\rightarrow z$.

线程间的 HB 关系可用共享锁来实现.如图 2 表示了两个线程执行相同代码段的一种可能次序,线程 1 要先于线程 2 执行,即线程 1→线程 2.

```
线程 1          线程 2
lock(mu);
v:=v+1;
unlock(mu);
                ↘
                lock(mu);
                v:=v+1;
                unlock(mu);
```

图 2 Lamport's HB 关系

假定两个线程都要访问同一个共享变量,而这个访问却没有被 HB 关系所定序,那么在这个程序的另一次执行时,如果较快的进程运行慢了或较慢的进程运行快了,都有可能引起两线程同时访问该共享变量,亦即数据竞争发生了.如图 3 所示,线程 1 和线程 2 都访问共享变量 y,按照图上所示的执行次序,对 y 的访问不会构成数据竞争,但如果线程 2 先于线程 1 投入运行,则可能导致对 y 的改写冲突.针对这种情况,本文提出了一种基于锁集的解决数据竞争的算法.

```
线程 1          线程 2
y:=y+1;
lock(mu);
v:=v+1;
unlock(mu);
                ↘
                lock(mu);
                v:=v+1;
                unlock(mu);
                y:=y+1;
```

图 3 程序允许在 y 上进行数据竞争

2 锁集算法

2.1 变量的状态变迁

程序中的数据可以划分为变量和常量两大类,而数据竞争只可能发生在对某些共享变量的访问上,并不是对所有变量的读写都会发生数据竞争.在程序中,某些变量除了在定义时执行写操作外,其值一直保持不变,直到程序运行结束,因此线程对它们的访问只是读操作,是不会发生数据竞争的;而另一些变量的值会随着程序的向前推进而不断变化,如果它又为几个线程所共享的话,是可能发生数据竞争的.为了讨论方便,对于每一个变量 v,可以根据它在程序运行过程中的表现将其所处的状态划分成以下几种(如图 4):

◆初始态:当变量 v 刚刚被定义时所处的状态,它是一个瞬间状态.一俟创建完毕,即转入私有态.

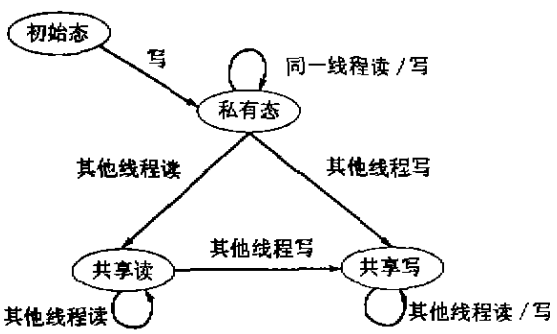

图 4 变量的状态变迁图

◆私有态:当变量 v 仅被某一线程读访问时所处的状态,此时变量 v 为该线程所私有.

◆ 共享读态:当变量 v 被两个以上的线程读访问后所处的状态.此时变量 v 为多个线程所共享.

◆ 共享写态:当变量 v 被读访问以外的线程写访问后所处的状态.此时变量 v 为多个线程所共享

根据上述对变量在程序运行期间状态变迁的分析,可以将程序运行期间的变量分成3类:

◆ 初始类:处于独享态的变量;

◆ 共享读类:处于共享读态的变量;

◆ 读写锁类:处于共享写态的变量.

显然,对于前面两类变量是不存在读写冲突的,因此对这两类变量在运行期间可以不用作任何特殊地处理,仅把它们当作一般的变量进行处理即可.但对于最后一类的共享变量却要好好地处理才行.

2.2 锁集算法

对于任何一个共享变量 v,定义 $C(v)$为线程运行过程中所拥有的加载到 v 上的所有锁的集合.初始时为 v 的所有候选锁集,即所有包含 v 的线程中用于保护共享读写 v 的锁的集合.

对于任何线程 t,定义 locks_held(t)为线程 t 在运行过程中所拥有的锁集,初值为空.

对于任何线程 t,定义 write_locks_held(t)为线程 t 在运行过程中所拥有的写方式下的锁集,初值为空.

对于下列序列,它们值的变化如下:

执行序列	$C(v)$	locks_held	write_locks_held
	{mu1,mu2}	{}	{}
lock(mu1);		{mu1}	
$v_i=v+1$;	{mu1}		{mu1}
unlock(mu1);		{}	{}
lock(mu2);	{mu2}		
$v_i=v+1$;	{mu2}		{mu2}
unlock(mu2);		{}	{}

从上面的3个锁集 $C(v)$,locks_held 和 write_locks_held 的变化中可以看出,$C(v)$的值是依赖于 locks_held 和 write_locks_held 的.在下述算法中,私有变量 cv 代表 $C(v)$的瞬间值.对于系统中任何正在运行的线程 t 执行如下算法.

初始值:

locks_held(t) ={};

write_locks_held(t)={};

对∀ v,$C(v)$={共享变量 v 的所有候选锁};

创建一个私有的锁堆栈 locks_stack(t),初值为空.

算法体:

```
while(线程 t 未执行完) do {
 switch(遇到锁操作或共享变量 ) {
 case 执行 lock(mutex)语句之后:
      向 locks_held(t) 中加入 mutex;
      将 mutex 压入 locks_stack(t);
      break;
 case 执行完 unlock(mutex)语句之后:
      从 locks_held(t)中去除 mutex;
      从 locks_stack(t)中弹出一个元素;
      if mutex ∈ write_locks_held(t)
        then 从 write_locks_held(t)中去除
        mutex;
      break;
 case 线程 t 中的 v 是读操作:
      cv := C(v)∩locks_held(t);
      if cv={} then issue"无锁保护读";
      else if cv ∩ ⋃(t'≠t) write_lock_held(t') ≠
      {}
        then issue "读写冲突";
      break;
 case 线程 t 中的 v 是写操作:
      // 更新 write_locks_held(t)的信息
      将 locks_stack(t)中的所有元素加入到
      write_locks_held(t)中;
      cv:=C(v)∩ write_locks_held(t);
      if cv={} then issue"无锁保护写";
      else if cv∩ ⋃(t'≠t) write_lock_held(t') ≠{}
      then issue "写写冲突";
      else if cv∩ ⋃(t'≠t) locks_held(t') ≠{}
      then issue "读写冲突";
      break;
 otherwise: 空操作;
 } //end of switch
} // end of while
```

从上述算法可以看出,它既能够探测出不受锁保护的共享变量的读写冲突,又能准确地预报出冲突的种类和位置,因此在线程间的数据竞争的探测方面具有一定的实用性.该算法要对系统中的每一个正在运行的线程进行实时监测,因此该算法的时

间复杂度与系统中正在运行的线程的数目成正比的.采用该算法可能会对系统的效率运行一定的影响.

3 研究展望

虽说上述算法能够有效地探测出不受锁保护的共享变量之间的读写冲突问题,但对于以间接方式访问的共享变量(如C语言中的指针)的冲突还不是能有效地探测出来.相对于探测直接访问的共享变量读写冲突来说,探测间接共享变量的读写冲突要复杂得多,它主要表现在以下几个方面:1) 间接共享变量的大小无法确定,它可能是一个简单变量指针,可能是一个指向数组的指针,也可能是一个指向指针的指针;2) 间接共享变量的类型不明确,如C语言可以采用强制类型转换在相容类型之间进行赋值;3) 对间接共享变量的锁集维护也有一定的困难.假定共享的是间接共享变量的一部分而不是全部,此时如果将整个变量所指向的空间都保护起来的话,虽说不会发生读写冲突,却可能降低程序的运行效率.因此,可对每个读写方式下的间接共享变量设置一个五元组(变量名,变量类型,变量的起始位置,变量的终止位置,该变量的共享锁集),表示相应的运行状态.通过查询共享变量的起止地址来判断变量是否共享、有无锁保护、设置和取消共享锁等操作.对间接共享变量的读写冲突问题还有待于作进一步地研究.

参考文献:

[1] LI Chun-hua, XU Ming, ZHOU Xing-ming. Software Implementation of Multithreading. *Computer Engineering & Science*, 1999, 21(4): 17-21(Ch).

[2] HE Yan-xiang. *The Design of Distributed Operating System*. Beijing: Ocean Press, 1993(Ch).

[3] HE Yan-xiang. *Programming Skill on Concurrent Language*. Wuhan: Wuhan University Press, 1990 (Ch).

[4] Andrew S. *Tanenbaum Distributed Operating Systems*. Beijing: Tsinghua University Press, 1996.

A Dynamic Detective Algorithm Based on Lockset to Solve Multithreaded Data Race

LI Ke-qing, CHEN Xin-meng, ZHENG Wu-ji

(College of Mathematics and Computer Science; The State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China)

Abstract: Multithreaded programming has been applied widely in system software and applications software. Not only can it explore the concurrent part hidden in the executing process, but also can utilize sufficiently the CPU resources. Multi-threaded programming is not only difficult but also prone to error. In this paper, we first analyzed states of shared-variable, then put forth a dynamic detective algorithm based on lockset to find shared-variable data race in multithreaded programming environment. It has certain value in detecting read-write conflict among shared variables.

Key words: lock; data race; multithreaded programming