操作系统中的死锁检测①

张海鹏[1], 李 曦[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230027)
[2](中国科学技术大学 苏州研究院, 苏州 215123)

摘 要: 本文提供了一种检测操作系统中死锁的方法. 该方法包含三个步骤: (1)通过检测进程加锁与解锁是否匹配来获得锁的持有者; (2)从异常进程中筛选出锁的等待者; (3)通过检查锁的持有者与等待者是否会形成循环等待图来判定死锁. 通过实验发现, 该方法对系统性能的影响小于 1%, 而且不需要修改内核源码和源程序.

关键词: 操作系统; 死锁检测; 并发程序; 异常进程

Deadlock Detection in Operating System

ZHANG Hai-Peng[1], LI Xi[1,2]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

Abstract: This paper introduces a method to detect operating system deadlock. The method consists of three stages: (1) finding the lock-holders by detecting whether the locks and unlocks are paired. (2) finding the lock-waiters by filtering the abnormal process. (3) the method will occur deadlock warning if there is a cycle in the waits-for graph. The results show that the method has only 1% influence to the performance, and does not need to modify Linux kernel and detected program source code.

Key words: operating system; deadlock detection; concurrent programs; abnormal processes

1 引言

为了充分发挥 cpu 多核的性能, 并发程序设计已经十分广泛, 但是开发并发程序面临很多挑战, 死锁就是其中的一个. 在设备驱动错误中有 19%的错误是由于并发导致的[1], 在这些并发错误中 72%(67/93)跟死锁有关, 文献[2]通过对 4 大开源软件: MySQL、Apache、Mozilla 和 OpenOffice 中的并发错误进行分析, 发现 30%(31/105)的错误是由死锁造成的.

死锁检测的方法分为静态和动态两种. 静态检测通过工具来分析待检测程序的源代码来找出可能要发生死锁的程序位置, 这种方法不适用于检测代码量大的程序, 而且很不准确. 文献[3]提供的静态检测方法在应用到 JDK 时发现了 100,000 多个死锁, 实际只有 70 个. 动态检测是在程序运行时, 检查可能发生的死锁, 但是动态监测性能开销大, 不适用于内核、驱动中死锁的检测[4].

关于 java 多线程程序中的死锁检测, 无论是静态方法[5,6]还是动态方法[7,8], 过去都已经做过大量研究. 而且目前已经有比较成熟的工具可以直接检查 java 程序中的死锁, 如 jstack、lockstat 等. 由于操作系统代码量大、对性能敏感, 过去关于操作系统死锁方面的研究比较少. 死锁检测工具 pluse[9]是在操作系统层实现的, 但是该工具只能检测应用程序中的死锁, 并不能检测操作系统本身的死锁.

本论文将使用一种动态检测死锁的方法, 可以准确检测出操作系统中的死锁. 本论文提供的方法对操纵系统性能的影响小于 1%, 而且不需要修改内核源码和任何的应用程序.

本论文安排如下: 第 2 部分介绍本文采用的死锁检测的算法. 第 3 部分介绍如何获得锁的持有者. 第 4

① 基金项目:国家自然科学基金(61272131, 61202053);江苏省自然科学基金(SBK201240198)
收稿时间:2013-03-31;收到修改稿时间:2013-04-28

部分介绍如何获得锁的等待者. 第 5 部分根据第 3、4 部分的结果判断是否形成循环等待图. 第 6 部分是实验结果. 第 7 部分将对论文进行总结.

2 死锁检测算法

当进程发生死锁时, 进程既是锁的持有者, 也是锁的等待者, 而且进程间会形成一个循环等待图. 本文采用的死锁检测算法就是从锁的持有者中筛选出锁的等待者, 然后进一步判断它们之间是否形成一个循环等待图.

图 1 描述了本文采用的死锁检测算法:

(1) 每隔一定时间(*search_cycle* 指定)检查锁持有者链表 HOLDER_LIST 是否为空.

(2) 若不为空则检查这些锁的持有者是不是锁的等待者.

(3) 若检查结果为 *ture*, 则检查这些锁的持有者与等待者之间是否会形成一个循环等待图.

(4) 如果存在循环等待图则说明发生了死锁.

```
1 while (1) {
2       sleep (search_cycle )
3       if HOLDER _LIST not NULL
4             search the lock _holders from HOLDER _LIST
5             if lock _holders are lock _waiters
6                   if there is a cycle in waits _for graph
7                         report deadlock bug
8                   else
9                         deadlock free
10            else
11                  deadlock free
12      else
13            deadlock free
14 }
```

图 1 死锁检测算法

3 获取锁的持有者

在操作系统中线程通过加锁函数来申请锁, 通过解锁函数来释放锁. 成功加锁的线程在解锁之前便是锁的持有者, 也即锁的持有者是那些没有执行解锁函数的线程. 因此, 获取锁的持有者可以跟踪线程的加锁与解锁操作, 找出只成功加锁而没有解锁的线程. 为了能够探测加锁和解锁函数, 文献[10]针对 java 程序中的 monitor entry/exit 操作采用一个面向切面的编译器 AspectJ, 而针对于 pthread 多线程程序则需要修改相关的库函数, 这些方法不利于软件的升级和维护. 本文使用可以动态探测 Linux 内核函数的工具 systemtap, 它既不需要重新编译内核, 也不需要修改任何源程序和库函数.

systemtap 既可以在函数的入口处进行探测, 也可以在函数的出口处进行探测. 若在加锁函数退出的地方进行探测, 那么就可以获得锁的持有者信息, 因为只有成功获得锁, 进程(本论文对线程与进程不区分对待)才能从加锁函数中退出, 否则便处于等待状态. 为了唯一标识进程加锁和解锁操作, 使用由进程号(*pid*)、锁类型(*lock_type*)、资源地址(*resource_addr*)组成的三元组<*pid*, *lock_type*, *resource_addr*>.

图 2 描述了本文采用的获得锁持有者算法:

(1) 当进程加锁的时候, 将<*pid*, *lock_type*, *resource_addr*>插入死锁检测程序维护的链表 HOLDER_LIST 中.

(2) 当进程解锁时, 将<*pid*, *lock_type*, *resource_addr*>从 HOLDER_LIST 链表中删除. 那么 HOLDER_LIST 链表中保存的便是那些只有成功加锁而没有解锁的进程.

(3) 当死锁检查周期 *search_cycle* 时间到的时候, 便可以从 HOLDER_LIST 中获得锁的持有者.

```
1 when thread lock
2       insert <pid , lock _type , resource _addr > to HOLDER _LIST
3 when thread unlock
4       delete <pid, lock _type , resource _addr > from HOLDER _LIST
5 while (1){
6       sleep (search _cycle )
7       if HOLDER_LIST not NULL
8             search the lock holders from HOLDER _LIST
9 }
```

图 2 获得锁持有者算法

4 获取锁的等待者

使用 systemtap 在加锁函数的入口处进行探测, 就可以获得该锁的所有申请者. 申请者中除去锁的持有者, 剩下的便是等待该锁的进程. 操作系统中的互斥锁只允许有一个持有者, 但却允许有 *n* 个等待着, 当 *n* 很大时, 这种获得锁等待者的方法, 将会对系统的性能差生很大影响. 这是因为 systemtap 运行在进程执行的上下文中, 在进程执行的上下文使用什么样的数据结构和算法在来高效的组织这 *n* 个等待者将是一个非常棘手的问题. 因此, 本文使用通过筛选异常进程的方法来获得锁的等待者, 该方法可以在 1s 以内完成系统中 300 个线程的分析.

定义 1. 异常进程: 等待时间超过阈值 *threshold_time* 的进程.

这里的等待既包含进程获不到处理器的等待, 也

包含进程一直占有处理器的忙等. 由于发生死锁的进程要么永远得不到处理器, 要么永远占有处理器, 所以死锁进程必定是异常进程. 异常进程中只有一部分是因为等待锁而异常的, 为了获取这些锁的等待者, 还需要对判定为异常的进程进行筛选.

4.1 计算等待时间

为了获得长时间睡眠的进程, 文献[9]在进程的task_struct 中设置一个布尔型变量 *was_asleep*. 进程创建或该进程被调度的时候设为 *false*, 当周期性检查的时候, 若发现进程已经处于睡眠且该值为 *false* 时, 则将 *was_asleep* 设置为 *ture*; 若该值已经是 *ture* 则认为是长时间睡眠的进程(睡眠时间大于周期性检查的时间). 这种做法有两个明显的缺陷: 第一, 不准确. 根据 4.2.2 的统计进程占有的 cpu 时间不会超过 1s, 将周期性检查周期设置为 1 分钟, 那么进程的执行时间是可以忽略的. 假设周期性检查了 5 次, 进程被调度的时机即为 *was_asleep* 设置为 *false* 的时机, 若按照图 2 进行, 那么即使进程每次睡眠时间 T 均超过周期性检查时间(1 分钟), 那么采用论文中的方法也无法将该进程判定为长时间睡眠的进程. 第二, 无法用于因自旋锁死锁而处于忙等的进程, 这些进程会一直处于运行状态, *was_asleep* 也就没有机会被设置为 *true*.

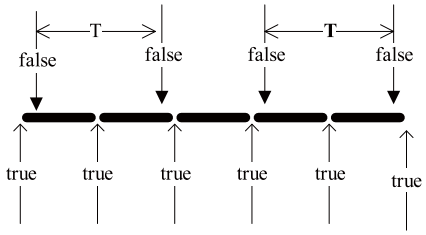

图 3　特殊的长时间睡眠进程

本文使用进程描述符 task_struct 结构体中的 *last_arrival* 为参考, 来判断进程的等待(包括忙等)时间是否达到 *threshold_time*. 在进程的状态切换中, 只有当进程由就绪状态切换到运行态的时候, 才改变 *last_arrival* 的值. 因此, 无论进程是在阻塞状态还是在运行态(忙等), 该值均不会改变. 使用当前的系统时间减去 *last_arrival* 便是进程睡眠或忙等的时间, 若该差值超过 *threshold_time*, 那么就认为该进程是异常进程.

4.2 计算阈值 *threshold_time*

判断进程是否是异常进程, 需要确认时间阈值 *threshold_time*, 该值不能太大也不能太小, 太大将会延长死锁线程对系统的影响, 太小可能会误报死锁. *threshold_time* 必须大于进程持有锁的时间 *lock_time*, 否则会将正常持有锁的进程判为异常进程. 另外阈值 *threshold_time* 也必须大于进程的时间片 *time_slice*, 这是为了筛选因为忙等而异常的进程. 因此, 阈值 *threshold_time* 应该大于两者中的最大值, 即 *thread_time* > max (*lock_time*, *time_slice*).

4.2.1 统计进程持有锁时间 *lock_time*

操作系统中的锁可以分为两类: 睡眠锁和非睡眠锁. 睡眠锁允许锁的获得者在锁保护的临界区中睡眠, 而非睡眠锁则不允许. 由于睡眠锁粒度大, 进程持有睡眠锁的时间要比持有非睡眠锁的时间长, 因此统计进程持有睡眠锁的时间更有意义.

操作系统中的睡眠锁有信号量(semaphores)、读/写信号量(read/write semaphore)和互斥体(mutex), 针对不同的使用场景, 这些睡眠锁提供了不同的加锁函数(表 1), 如 down_killable 允许通过信号(signal)来杀死等待者, 其他函数含义及使用方法参见 Linux 内核源码.

表 1　操作系统中睡眠锁的加锁

	semaphores	read/write semaphore	mutex
加锁函数	down down_killable down_trylock down_interruptible	down_read down_write down_read_trylock down_write_trylock	mutex_lock

加锁函数与解锁函数之间的时间便是线程持有该锁的时间. 在表 2 所示的环境上测 10 分钟, 筛选出各个睡眠锁允许线程的最大持有时间如图 4 所示. 从图 4 上可以看出, 互斥体(mutex)允许线程持有更长的时间, 但是该时间也没有超过 14 毫秒.

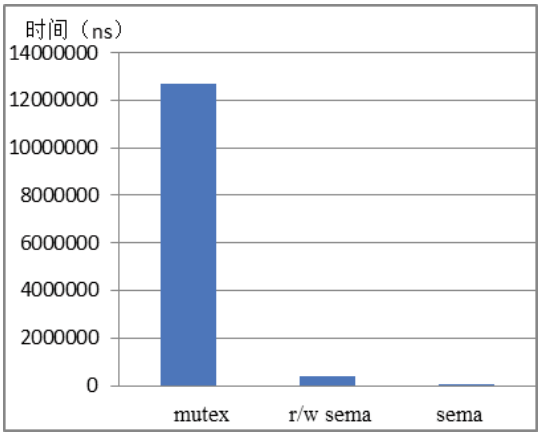

图 4　线程持有睡眠锁的时间

4.2.2 统计时间片 *time_slice*

在进程的描述符中会记录进程总的运行时间(*sum_exec_runtime*)以及运行次数(*pcount*), 这两者之间的比值便是进程的平均时间片 *time_slice*. 目前, Linux 的进程调度器 CFS 会对 cpu 密集型的任务进行奖励, 从而让其占有更长时间的处理器. 为了获得 cpu 密集型任务的时间片, 使用 stress[11]工具来模拟 cpu 密集型的任务, 然后在表 2 所示的环境上使用 systemtap 统计出 10 分钟内, 时间片最长的 3 个任务(如图 5), 从图 5 可知进程 stress 的时间片最长, 但是也没有超过 900 毫秒.

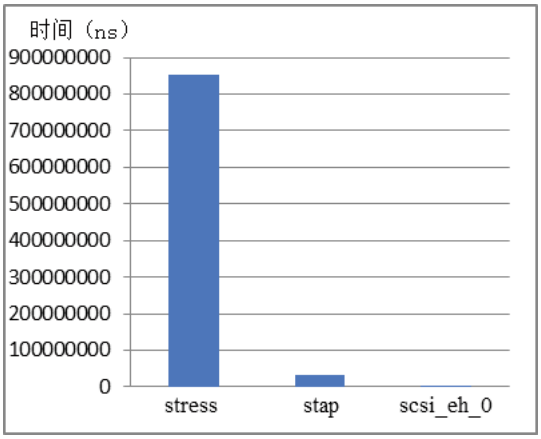

图 5 进程时间片

通过对进程持有锁时间 *lock_time* 以及进程时间片 *time_slice* 的统计发现, 阈值 *threshold_time* 设定为 10s 是合适的, 该值允许用户自己设定.

4.3 筛选异常进程

异常进程中只有一部分是因为等待锁而异常的, 因此需要对异常进程进行筛选, 找出因等待锁而异常的进程. 当进程执行加锁函数的时候, 内核会将加锁函数的返回地址压入进程的内核栈. 当进程从加锁函数成功退出的时候, 再将该地址从栈中弹出. 因加锁函数而异常的进程, 其内核栈上必定存在加锁函数的返回地址, 内核提供的函数 kallsyms_lookup 可以根据该地址找到对应的加锁函数. 因此, 可以通过判断异常进程的内核栈上有无加锁函数, 来从异常进程中筛选出锁的等待者.

5 循环等待图

根据图 1 的死锁检测算法, 为了检查线程间是否发生了死锁, 需要判断线程的资源依赖关系是否会形成一个循环等待图(图 1 第 6 行代码). 若线程 *T*1、*T*2、*T*3 发生死锁, 那么它们的资源依赖关系将如图 6 所示.

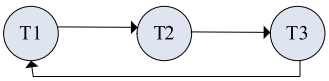

图 6 死锁中的循环等待

定义 2. 资源依赖关系: 假设线程 *T1* 等待资源 *R2*, 线程 *T2* 是资源 *R2* 的持有者, 那么线程 *T1* 与 *T2* 之间的关系称作资源依赖关系, 表示为 *T1->T2*.

这种资源依赖关系是可传递的, 若 *T1->T2*, *T2->T3*, 则有 *T1->T3*. 死锁发生的充要条件是形成图 6 的循环等待图. 从该图中的任意线程 *Ti*, 使用资源依赖关系的传递性质, 均可以获得线程指向自己的依赖: *Ti->Ti*. 因此, 发生死锁的充要条件可以描述为, 是否存在线程指向自己的依赖关系.

从 HOLDER_LIST 中选择任意进程 *P1*, 从进程的内核栈上找出进程等待的资源 *R2*, 然后从 HOLDER_LIST 找出资源 *R2* 的持有者 *P2*, 这样便确定了进程 *P1* 和 *P2* 的资源依赖关系 *P1->P2*. 接下来从进程 *P2* 开始重复上述过程, 直到遍历完 HOLDER_LIST 链表或找到依赖关系 *Pj->Pi*, *Pi* 是已经遍历过的进程, 这样便形成了进程间的循环等待, 可以断定这些进程发生了死锁.

6 实验结果

为了检测死锁检测方法的有效性, 使用操作系统中的信号量模拟三个线程访问三种临界资源. 每个线程获得一种资源后, 延迟 1 秒钟, 然后申请另外一种资源. 线程第一次申请的是三种不同的资源, 可以成功获得, 第二次申请是其它线程已经拥有的资源, 由于加锁顺序不当导致发生死锁.

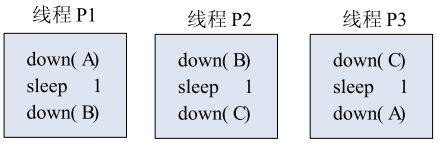

图 7 三个线程产生的信号量死锁

使用本文提供的死锁检测程序, 检查结果如图 8 所示. 临界资源使用资源地址来标识, 进程 2471 拥有资源 *a*0(资源地址前 16 位均一样, 只使用后两位标识),

申请资源 *c*0. 进程 2472 拥有资源 *c*0, 申请资源 *e*0. 进程 2473 拥有资源 *e*0, 申请资源 *a*0. 通过资源依赖的传递关系, 可以得出进程 2471 指向自己的依赖关系 2471->2471. 由第 5 节可以, 这样进程间便存在死锁.

```
1  pid 2471  acquire resource ffffffffa00903 a0
2  pid 2472  acquire resource ffffffffa00903 c0
3  pid 2473  acquire resource ffffffffa00903 e0
4  pid 2471 wait resource ffffffffa00903 c0
5  pid 2472 wait resource ffffffffa00903 e0
5  pid 2473 wait resource ffffffffa00903 a0
7  2471 ->2472 2472 ->2473 2473 ->2471
8  above process has dead lock
```

图 8 三线程死锁检测结果

在表 2 所示的测试环境上使用 Linux 自带的性能检测工具 top, 来测试该死锁检测方法对操作系统性能的影响. top 工具每隔一秒种收集系统的性能参数, 发现在死锁检测程序运行之前 cpu 使用率为 0%～1%, 在死锁检测程序运行时, cpu 使用率依然是 0%～1%, 这说明该方法对 cpu 性能的影响小于 1%.

表 2 测试环境

cpu	内存	操作系统
Q8400	4G	Debian 6.0（Linux-kernel 2.6.38）

本文的死锁检测算法跟[4]提供的检测操作系统死锁工具 RacerX 相比, 具有以下优点:

- 不需要修该内核. RacerX 需要修改内核才能获得操作系统的加锁与解锁操作函数, 本文使用动态探测工具 systemtap, 不需要修改内核.
- 性能开销小. RacerX 需要分析每一行代码 , 因此性能开销比较大. 通过实验发现, 本文的死锁检测算法开销小于 1%.

本文死锁检测算法的缺点是无法获得操作系统中潜在的死锁, 但是 RacerX 采用静态分析每一行代码的方法可以获得潜在的死锁.

7 结束语

并发程序设计存在的一个普遍问题便是死锁, 随着并发特性在操作系统中越来越多的应用, 使得操作系统发生死锁的概率也越来越大. 但是操作系统代码量大, 对性能敏感, 因此检测操作系统中的死锁问题具有一定的难度. 本论文结合 Linux 系统中的探针技术提供一种动态检测操作系统中死锁的方法. 该方法不仅能准确检测出操作系统中的死锁, 而且不需要对操作系统源码以及要检测的源程序做任何修改, 对操作系统性能影响也在 1%以内.

参考文献

1 Ryzhyk L, Chubb P, Kuz I, Heiser G. Dingo: Taming device drivers. Proc. of the 4th ACM European Conference on Computer Systems. 2009. 275–288.

2 Lu S. Learning from mistakes:a comprehensive study on real world concurrency bug characteristics. ACM SIGARCH Computer Architecture News 36.1, 2008: 329–339.

3 Williams A, William T, Ernst MD. Static deadlock detection for Java libraries. ECOOP 2005-Object-Oriented Programming. Springer Berlin Heidelberg, 2005: 602 –629.

4 Dawson E, Ashcraft K. RacerX:effective, static detection of race conditions and deadlocks. ACM SIGOPS Operating Systems Review 37.5, 2003: 237–252.

5 Havelund K. Using runtime analysis to guide model checking of java programs. 7th International SPIN Workshop on Model Checking and Software Verification. 2000. 245–264.

6 Joshi P, Naik M, Sen K. An effective dynamic analysis for detecting generalized deadlocks. Proc. of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE'10). 2010. 327–336.

7 Bensalem S, Havelund K. Scalable dynamic deadlock analysis of multi-threaded programs. PADTAD, 2005.

8 Joshi P, Park CS, Sen K, Naik M. A randomized dynamic program analysis technique for detecting real deadlocks. ACM Sigplan Notices, 2009: 110–120.

9 Tong L, Ellis CS, Lebeck AR, Sorin DJ. Pulse: A dynamic deadlock detection mechanism using speculative execution. Proc. of the 2005 USENIX Technical Conference. 2005. 31–44.

10 Jual H, TralamazzaD, ZamfirC. Deadlock immunity: Enabling systems to defend against deadlocks. OSDI, 2008: 295–308.

11 Linux stress: http://www.gnutoolbox.com/ linux-benchmark-tools/?page=detail&get_id=24&category=12.