软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(6):1682−1700 [doi: 10.13328/j.cnki.jos.006244]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于锁增广分段图的多线程程序死锁检测*

鲁法明[1], 郑佳静[1], 包云霞[2], 曾庆田[1], 段 华[2], 王晓宇[1]

[1](山东科技大学 计算机科学与工程学院,山东 青岛 266590)

[2](山东科技大学 数学与系统科学学院,山东 青岛 266590)

通讯作者: 鲁法明, E-mail: fm_lu@163.com

摘 要: 死锁是并行程序常见的缺陷之一,动态死锁分析方法根据程序运行轨迹构建锁图、分段图等模型来检测死锁.然而,锁图及其现有的各种变型无法区分同一循环中锁授权语句的多次执行,扩展锁图中记录的锁集无法捕捉线程曾经持有而又随后释放的锁信息,分段图无法刻画锁的获取和释放操作与线程启动操作耦合而导致的段间依赖关系.上述问题导致了多种死锁的误报.为解决上述问题,对已有的锁图和分段图模型进行改进,在锁图基础上扩充语句的执行时序信息,在分段图的基础上扩充锁的获取和释放信息,对段进行更细粒度的划分以建模锁对象导致的段间依赖关系;最终,在上述锁增广分段图与时序增广锁图的基础上,提出一种新的死锁检测方法.所提方法能够有效消除前述各种误报,从而提高死锁检测的准确率.文中开发相应的原型系统,并结合多个程序实例对所提方法的有效性进行评估验证.

关键词: 程序验证;死锁检测;锁图;分段图;动态死锁分析

中图法分类号: TP311

中文引用格式: 鲁法明,郑佳静,包云霞,曾庆田,段华,王晓宇.基于锁增广分段图的多线程程序死锁检测.软件学报,2021,32(6):1682−1700. http://www.jos.org. cn/1000-9825/6244.htm

英文引用格式: Lu FM, Zheng JJ, Bao YX, Zeng QT, Duan H, Wang XY. Deadlock detection of multithreaded programs based on lock-augmented segmentation graph. Ruan Jian Xue Bao/Journal of Software, 2021,32(6):1682−1700 (in Chinese). http://www.jos.org.cn/1000-9825/6244.htm

Deadlock Detection of Multithreaded Programs Based on Lock-augmented Segmentation Graph

LU Fa-Ming[1], ZHENG Jia-Jing[1], BAO Yun-Xia[2], ZENG Qing-Tian[1], DUAN Hua[2], WANG Xiao-Yu[1]

[1](College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China)

[2](College of Mathematics and Systems Science, Shandong University of Science and Technology, Qingdao 266590, China)

Abstract: Deadlocks are a common defect of parallel programs. To detect deadlocks, dynamic deadlock analysis methods build models such as lock graphs and segment graphs according to program running traces. However, a lock graph and its existing variants cannot

* 基金项目: 国家自然科学基金(61602279, 61472229); 国家重点研发计划(2016YFC0801406); 山东省泰山学者工程专项基金(ts20190936); 山东省高等学校青创科技支持计划(2019KJN024); 山东省自然科学基金智慧计算联合基金(ZL2019LZh001); 山东省博士后创新专项基金(201603056); 国家海洋局海洋遥测工程技术研究中心开放基金(2018002); 山东科技大学领军人才与优秀科研创新团队项目(2015TDJH102)

Foundation item: National Natural Science Foundation of China (61602279, 61472229); National Key Research and Development Plan (2016YFC0801406); Taishan Scholars Program of Shandong Province (ts20190936); Excellent Youth Innovation Team Foundation of Shandong Higher School (2019KJN024); Smart Computing Joint Fund of Shandong Provincial Natural Science Foundation (ZL2019LZh001); Postdoctoral Innovation Foundation of Shandong Province (201603056); Open Foundation of First Institute of Oceanography, MNR (2018002); Shandong University of Science and Technology Research Fund (2015TDJH102)

本文由“形式化方法与应用”专题特约编辑邓玉欣教授推荐.

收稿时间: 2020-08-28; 修改时间: 2020-10-26, 2020-12-19; 采用时间: 2021-01-18; jos 在线出版时间: 2021-02-07

distinguish different executions of one lock acquisition statement in a loop structure. The lock set used in extended lock graphs cannot capture those locks which were once held and then released. A segmentation graph cannot model the inter-segment dependencies caused by the coupling of lock release/acquisition operation and thread start operation. The above problems have led to a variety of false positives. To address this issue, existing lock graph and segment graph models are improved. Specifically, a lock graph is extended with statement execution order information. A segmentation graph is expanded with lock acquisition and release information. Furthermore, segments in a segmentation graph are more finely divided in the improved model to capture the inter-segment dependencies caused by lock objects. Eventually, based on the improved models, a new deadlock detection method is proposed. It can eliminate the aforementioned false alarms effectively and improve deadlock detection accuracy. A corresponding prototype system is developed for experimental evaluation. The validity of the proposed method is verified through experiments.

Key words: program verification; deadlock detection; lock graph; segmentation graph; dynamic deadlock analysis

为提高软件系统的运行效率,并行编程技术得到广泛应用.与此同时,随着软件规模和复杂度的不断扩大,加之并行程序在程序调度等方面的不确定性,死锁、数据竞争和原子性违背等并发缺陷日益显现[1,2].死锁是最具代表性的并发缺陷之一,死锁的发生会导致程序无法正常运行甚至是系统崩溃,由此带来不必要的损失.而且统计显示,约 30%的并发缺陷与死锁有关[3].因此,死锁检测成为提高软件可靠性和安全性亟需解决的问题.

文献[1]将近年来的死锁检测方法分为 3 类:定理证明与模型检验[4,5]、数据流分析[6−10]和动态分析[11−20].

- 数据流分析技术直接分析程序源码,组合使用调用图分析、指向分析和逃逸分析等静态分析技术,计算静态锁占用约束或者锁占用顺序图,使用约束求解和环检测算法在其上检测环,将环作为可能的死锁报告出来.该类方法缺乏精确的运行时信息,一般对变量值作保守估计,因此能较全面地发现潜在死锁,但会产生较多的误报;
- 定理证明与模型检验通常对程序行为进行形式化建模,之后通过模型的分析探索程序所有可能的执行路径,进而在理论上暴露潜在的死锁.该类方法的缺点是模型建模需大量的人工参与,而且如何保证抽象模型与程序语义的等价性至今悬而未解;
- 动态分析通过运行代码,获取程序执行轨迹,进而抽取执行轨迹中蕴含的程序行为模式,在此基础上进行死锁检测.动态分析方法充分利用了程序的运行时信息,故而检测的准确度较高,误报较少;而且由于运行轨迹只蕴含了程序的部分行为,这使得动态分析的效率也较高.不过,这同时带来了动态分析漏报率高的缺点.然而,鉴于死锁误报排除上的困难性,以及动态分析可自动化、效率高等优点,动态分析仍然是程序死锁检测的主流方法.

一般而言,动态死锁分析方法从程序运行轨迹中提取锁授权顺序中存在的特定模式,并据此检测潜在死锁.例如,文献[11]首次基于锁图(将每个锁对象作为图中的一个节点;当某线程在持有锁对象 A 的情况下申请锁对象 B 时,在节点 A 到 B 之间添加一条有向弧,由此形成的图称为锁图)提出了一个死锁的动态分析工具 Visual Threads,它将锁图中的每个环路视为一个潜在死锁.这种方法简单有效,但是存在多种类型的误报.比如,单一线程访问锁对象导致的环路、由门锁保护的环路、具有因果关系的多个线程之间的锁授权操作导致的环路等,这些环路都会导致死锁误报.文献[12]提出了基于锁树的 GoodLock 死锁检测算法,该算法能排除单线程环和门锁环导致的误报,但只能检测两个线程之间由于资源的持有和等待导致的死锁.文献[13]对 GoodLock 算法进行了改进,使其能够检测任意线程之间导致的死锁.文献[14]提出了环锁依赖链(cyclic lock dependency chain)的概念,它在锁图有向弧的基础上扩充了线程 ID、当前持有的锁集等信息,能同时排除单线程环和门锁保护环导致的误报,而且对构成死锁的线程数量没有限制.文献[15]设计方法对基于环锁依赖链的死锁检测方法进行性能改进,基本思想是:先设计算法识别和消除可移除的锁依赖关系,之后再进行死锁的定位.由此提出了扩展性和效率更高的 Magiclock 死锁检测方法.前述死锁检测方法能消除单线程环和门锁环导致的误报,实际上,线程的 *start* 和 *join* 语句也会引起多个线程间锁授权操作上的因果关系,这同样会导致误报.针对这一问题,文献[16,17]基于线程的 *start* 和 *join* 语句对线程的操作进行分段,根据段之间的依赖关系提出了分段图的概念;与此同时,在传统锁图的基础上扩充线程 ID、当前持有的锁集、段号等信息提出了扩展锁图的概念;最终,综合分段图和扩展锁图提出一种新型的死锁检测方法,可以排除单线程环、门锁环和多线程间具有因果关系的锁授权操作环

导致的误报.类似地,文献[18]定义了一类时间戳和向量时钟来刻画线程之间由于 *start* 和 *join* 语句所引起的操作上的因果关系,并进而与环锁依赖链相结合,提出了一种基于向量时钟和环锁依赖链的死锁检测新方法,也可以排除单线程环、门锁环和多线程间具有因果关系的段锁授权操作环导致的误报.

然而,上述死锁检测方法均存在如下问题.

(1) 通过锁图、锁树、环锁依赖链和扩展锁图等模型刻画锁授权顺序时,对同一个锁授权语句的多次执行不加区分.而实际当中,位于一个循环中的同一个锁获取语句,可能在某些轮次的循环中引起死锁,而另一些轮次中不会引起死锁,后面这类情况会引起死锁的误报;

(2) 分段图被用来确定构成环路的锁获取操作之间是否具有因果依赖,但分段图仅刻画了同一线程内操作间的因果依赖以及由于线程的 *start*,*join* 操作导致的线程间因果依赖关系,而实际上,锁对象的释放和获取在与线程的 *start* 操作耦合时也可能导致不同线程操作上的因果依赖;

(3) 传统锁图等工具仅记录了锁申请操作执行时正持有的锁集,并据此排除门锁环导致的误报.然而,除了持有锁集中的锁对象外,线程在获取上述锁的过程中可能存在某些其他曾经获取而又随后释放锁的操作,它们也可能导致死锁的误报.

文献[21]曾指出过上述第3类误报,但他们是通过死锁重演来排除这一误报,其给出的死锁检测方法并不能排除该误报.

为解决上述问题,本文一方面在扩展锁图的基础上添加语句的执行时序信息以区分循环中不同轮次的锁获取操作,据此提出时序增广锁图的概念;另一方面,在分段图的基础上扩充锁的获取和释放信息,对段进行更细粒度地划分,以更为准确地刻画操作上的因果依赖关系,据此提出了锁增广分段图的概念;最终,综合这两个工具提出一种新的死锁检测方法,它不仅可以消除单线程环、门锁环、多线程间由于线程的 *start* 和 *join* 操作而引起的因果关系导致的死锁误报,还能消除由于锁对象的释放和获取与线程 *start* 操作耦合引起的因果关系导致的死锁误报,而且能排除历史持有锁导致的误报,由此减少死锁误报,提高死锁检测的准确率.

1 实例与动机分析

1.1 程序实例

表1给出了一个多线程程序的实例 Program 1.它包含4个线程(主线程、*threadA*、*threadB*、*threadC*)和7个锁对象(*G*,*o*1,*o*2,*m*,*n*,*p*,*q*).*threadA* 中包含一个循环语句块,循环体中的锁授权语句会被多次执行.某些轮次的锁授权操作会引起死锁,而另一些轮次则不会导致死锁.此外,*threadB* 和 *threadC* 中关于锁对象 *m*,*n* 的嵌套获取会导致一个死锁,而关于 *p*,*q* 的嵌套获取由于曾经持有的锁构成门锁而不会导致死锁.

具体而言,主线程依次启动 *threadA* 和 *threadC*,之后阻塞直到 *threadA* 执行结束.*threadA* 启动后执行两轮循环,每一轮循环都是先获取锁 *G*;之后,在持有锁 *G* 的前提下,依次执行获取锁 *o*1、释放锁 *o*1、获取锁 *o*2、释放锁 *o*2 的操作;最后释放锁 *G*.两轮循环的不同之处在于:第1轮循环时(*flag*=0 时),*threadA* 会创建并启动 *threadB*;而第2轮循环时(*flag*=1 时)无此操作.*threadB* 被启动后,先执行获取锁 *G* 和释放锁 *G* 的操作,之后顺序执行获取 *o*2、获取 *o*1、释放 *o*1 和释放 *o*2 的操作.不难发现,“*threadA* 第1轮循环中关于锁 *o*1 和 *o*2 的嵌套获取”与“*threadB* 中关于锁 *o*2 和 *o*1 的嵌套获取”不会导致死锁.因为 *threadA* 第1轮循环中始终持有锁对象 *G*,而 *threadB* 中各个操作会因无法获取锁 *G* 而阻塞,仅当 *threadA* 释放锁 *G* 后,*threadB* 才能执行后续操作.换言之,锁 *G* 的释放和获取在上述两个操作之间构成一种因果依赖关系,使得两者无法并发,从而不导致死锁.相反地,*threadA* 第2轮循环的执行过程中,若是 *threadB* 先获取锁 *G* 并释放,之后在持有锁 *o*2 的同时申请锁 *o*1,而此时,*threadA* 在持有锁 *G* 和 *o*1 的同时申请 *o*2,则此时会触发死锁.

此外,*threadB* 除上述操作外,还将依次执行获取锁 *m*、获取锁 *n*、释放锁 *n* 的操作,并在持有锁 *m* 的情况下执行获取 *q*、获取 *p*、释放 *p*、释放 *q* 的操作,最后再释放锁 *m*.*threadC* 被主线程启动后,先依次获取 *n*、获取 *m*、释放 *m*,之后在持有锁 *n* 的前提下依次获取 *p*、获取 *q*、释放 *q*、释放 *p*,最后再释放锁 *n*.不难发现:*threadB* 和 *threadC* 中关于锁对象 *m*,*n* 的嵌套获取构成一个锁对象的持有等待回路,对应一个真实死锁.然而,“*threadB* 在第27行和

第 28 行关于锁 *q* 和 *p* 的嵌套获取”与“*threadC* 在第 35 行和 36 行关于锁 *p* 和 *q* 的嵌套获取”不会导致死锁,因为 *threadB* 获取 *q* 和 *p* 的前提是持有锁 *m*,而一旦 *threadB* 先持有了 *m*,则 *threadC* 便会被阻塞在第 34 行获取 *m* 的位置,从而无法嵌套获取 *p* 和 *q*;反之,*threadC* 获取 *p* 和 *q* 的前提是持有锁 *n*,而一旦 *threadC* 先持有了 *n*,则 *threadB* 将被阻塞在第 26 行获取 *n* 的位置,从而无法嵌套获取 *q* 和 *p*.由此可见:虽然 *threadB* 执行第 28 行获取 *p* 的操作时持有的锁集为{*m*,*q*},*threadC* 执行第 36 行获取 *q* 的操作时持有的锁集为{*n*,*p*},两个持有锁集互不相交,但是它们之前曾经持有和随后释放的锁对象 *n* 和 *m* 却也形成一组门锁,从而消除了这两个操作并发的可能性,使得它们不会导致死锁.

Table 1 Pseudo-code of a multithreaded program, denoted by Program 1

表 1 多线程程序 Program 1 的伪代码

Main Thread:
01:public static void *main*(*String*[·] args) throws *InterruptedExce*{
02: *flag*=0;
03: new *threadA.start*(·);
04: new *theradC.start*(·);
05: *threadA.join*(·);
06:}
	ThreadB:
	20: poblic void run{
	21: *synchronized*(*G*){·}
	22: *synchronized*(*o*2){
	23: *synchronized*(*o*1){·}
	24: }
	25: *synchronized*(*m*){
	26: *synchronized*(*n*){·}
	27: *synchronized*(*q*){
	28: *synchronized*(*p*){·}
	29: }
	30: }
	31: }

ThreadA:
07: poblic void run{
08: **for** (*int i*=0; *i*<2; *i*++){
09: *synchronized*(*G*){
10: **if** (*flag*==0){
11: new *threadB.start*(·);
12: *flag*=1;
13: }
14: *synchronized*(*o*1){
15: *synchronized*(*o*2){·}
16: }
17: }
18: }
19: }
	ThreadC:
	32: poblic void run{
	33: *synchronized*(*n*){
	34: *synchronized*(*m*){·}
	35: *synchronized*(*p*){
	36: *synchronized*(*q*){·}
	37: }
	38: }
	39: }

1.2 动机分析

如上节所述,Program 1 存在 4 处锁对象嵌套获取构成的循环依赖环路,但只有两处对应真实死锁,另两处不是死锁.然而,传统的动态死锁分析方法无法区分上述情况,由此导致死锁误报.

具体来说,假设 Program 1 某次运行轨迹见表 2.其中,*start*(*u*,*v*)代表线程 *u* 启动线程 *v* 的操作,*stop*(*u*)代表线程 *u* 的终止操作,*join*(*u*,*v*)代表线程 *u* 同步等待直至线程 *v* 终止,*acq*(*t*,*l*)表示线程 *t* 获取锁 *l* 的操作,*rel*(*t*,*l*)代表线程 *t* 释放锁 *l* 的操作.下面以基于分段图和扩展锁图的死锁检测方法为例,说明传统动态分析方法的检测结果.

文献[17]基于 *start* 和 *join* 将源程序的操作分为很多段,并根据各个段在执行顺序上的先后依赖关系构造分段图.具体来说,最初仅为主线程添加一个初始段;之后,每当执行操作 *start*(*u*,*v*)时,在线程 *u* 此操作之处添加一个新段,为线程 *v* 添加一个初始段,并在线程 *u* 的上一个段与这两个新段之间各添加一条有向弧;每当执行操作 *join*(*u*,*v*)时,为线程 *u* 添加一个新段,在 *u* 的上一个段与此段间添加一条有向弧,并在线程 *v* 的线程终止段与此段间也添加一条有向弧(该有向弧的添加需要待线程 *v* 执行终止操作时方能添加).对于表 1 的运行轨迹,构造所得的分段图如图 1(a)所示.

除构造分段图外,文献[17]为锁图每条有向弧⟨$lock_1$,$lock_2$⟩扩充标记信息 *arcID*:⟨seg_1ID,(*threadID*,*lockSet*),seg_2ID⟩,提出了扩展锁图的概念.其中,*arcID* 表示弧的唯一 ID,*threadID* 表示当前锁申请操作所属线程的 ID,*seg1ID* 代表 *threadID* 获取锁 $lock_1$ 所在的段号,seg_2ID 代表 *threadID* 获取锁 $lock_2$ 时所在的段号,*lockSet* 代表

threadID 获取锁 $lock_2$ 前持有的锁集.图 1(b)为表 1 运行轨迹对应的扩展锁图,它由 2 个子图构成.

得到分段图和扩展锁图后,文献[17]按照如下规则识别死锁:(1) 扩展锁图中两个锁的获取操作构成一条有向环;(2) 两个操作分属于的不同的线程;(3) 分段图中,这两个操作所在的段不存在有向路径相连(若存在有向路径则说明这两个操作存在执行次序上的先后依赖关系,无法并发,也就不构成死锁);(4) 这两个操作执行时所持有的锁集互不相交.

Table 2 A running trace of Program 1

表 2 Program 1 的一条运行轨迹

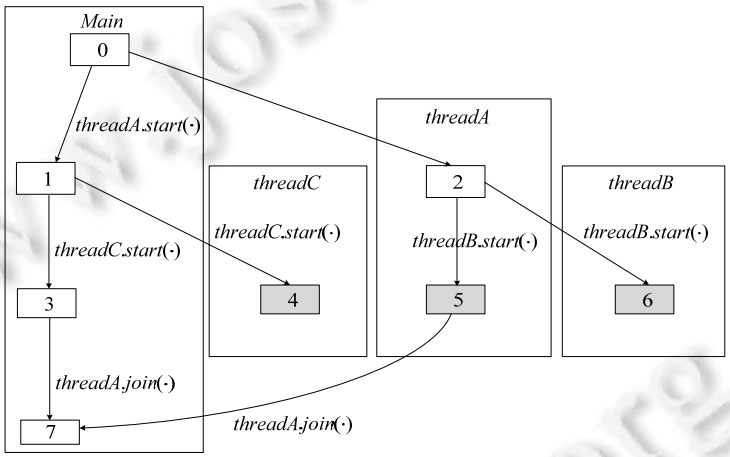
操作序号 ID	主线程	*threadA*	*threadB*	*threadC*
1	03:*start*(*Main*,*threadA*)			
2		09:*acq*(*threadA*,*G*)		
3		11:*start*(*threadA*,*threadB*)		
4		14:*acq*(*threadA*,*o*1)		
5		15:*acq*(*threadA*,*o*2)		
6		15:*rel*(*threadA*,*o*2)		
7		16:*rel*(*threadA*,*o*1)		
8		17:*rel*(*threadA*,*G*)		
9		09:*acq*(*threadA*,*G*)		
10		14:*acq*(*threadA*,*o*1)		
11		15:*acq*(*threadA*,*o*2)		
12		15:*rel*(*threadA*,*o*2)		
13		16:*rel*(*threadA*,*o*1)		
14		17:*rel*(*threadA*,*G*)		
15			21:*acq*(*threadB*,*G*)	
16			21:*rel*(*threadB*,*G*)	
17			22:*acq*(*threadB*,*o*2)	
18			23:*acq*(*threadB*,*o*1)	
19			23:*rel*(*threadB*,*o*1)	
20			24:*rel*(*threadB*,*o*2)	
21			25:*acq*(*threadB*,*m*)	
22			26:*acq*(*threadB*,*n*)	
23			26:*rel*(*threadB*,*n*)	
24			27:*acq*(*threadB*,*q*)	
25			28:*acq*(*threadB*,*p*)	
26			28:*rel*(*threadB*,*p*)	
27			29:*rel*(*threadB*,*q*)	
28			30:*rel*(*threadB*,*m*)	
29		19:*stop*(*threadA*)		
30			31:*stop*(*threadB*)	
31	04:*start*(*Main*,*threadC*)			
32				33:*acq*(*threadC*,*n*)
33				34:*acq*(*threadC*,*m*)
34				34:*rel*(*threadC*,*m*)
35				35:*acq*(*threadC*,*p*)
36				36:*acq*(*threadC*,*q*)
37				36:*rel*(*threadC*,*q*)
38				37:*rel*(*threadC*,*p*)
39				38:*rel*(*threadC*,*n*)
40				39:*stop*(*threadB*)
41	05:*join*(*Main*,*threadA*)			
42	06:*stop*(*Main*)			

按上述规则,图 1(b)中 ID 为 2,7 的两条有向弧对应的操作会被认为导致死锁.因为两者在扩展锁图中形成环路,分属不同的线程,所在的 5 号段和 6 号段不存在有向路径相连,两者持有的锁集{*G*,*o*1}与{*o*2}不相交.此外,ID 为 5 的有向弧与 ID 为 2 的两条有向弧是同一个锁获取语句在两个不同轮次循环中的具体执行,扩展锁图无法对其区分,两者的标记信息完全一样,从而也会认为导致一个死锁.而实际上,如第 1.1 节所述,其中一个是真实死锁,另一个是死锁的误报.最后,图 1(b)中 ID 为 11,15 的两条有向弧对应的操作也符合前述 4 条规则,从而也会被判定导致死锁.但如上节所述,这个死锁也是误报.

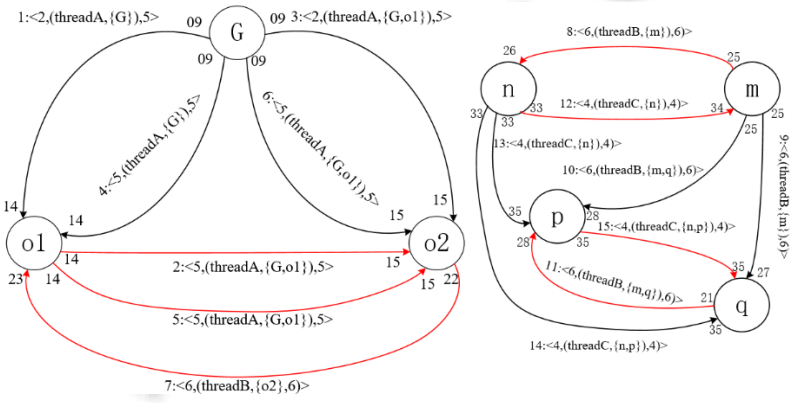
上述第 1 个误报的产生根源是:*threadA* 在持有锁 *G* 之后方才启动线程 *threadB*,从而导致图 1 中 2,7 两条弧对应的操作间具有因果关系(仅当弧 2 对应的操作执行完并释放锁 *G* 后,弧 7 对应的操作方能执行),而这种因果关系在分段图中未能刻画(5 号段和 6 号段间不存在有向路径).为处理这一问题,本文稍后将定义一种“锁-*start*”耦合因果依赖关系.

第 2 个误报产生的根源是:*threadB* 获取锁 *p* 必须先获取 *n*,而锁 *n* 的释放导致了 *threadB* 获取 *q* 时的持有锁

集不会包含 *n*;同理,*threadC* 欲获取锁 *q* 必须先获取 *m*,而锁 *m* 的释放导致了 *threadB* 获取 *p* 时的持有锁集不会包含 *m*,这就导致了他们的持有锁集交集为空,而已有方法仅根据持有锁集是否相交排除门锁环误报,所以会认定其构成一个潜在死锁.而实际上,若要真正触发上述死锁,则锁 *n* 需要被 *threadB* 获取并释放后才能授权给 *threadC*.类似地,锁 *m* 需要被 *threadC* 获取并释放后才能授权给 *threadB*,也就是说,对于同时存在于线程 *u* 之历史持有锁集与另一线程 *v* 之持有锁集中的锁对象 *o* 而言,历史持有锁的获取操作 *acg*(*u*,*o*)应先于持有锁的获取操作 *acg*(*v*,*o*),我们称这种关系为"历史持有锁-持有锁"耦合因果依赖关系,其严格定义稍后给出.Program 1 中,这种依赖关系要求操作 26:*acq*(*threadB*,*n*)先于 33:*acq*(*threadC*,*n*)执行;同时,操作 34:*acq*(*threadC*,*m*)先于 25:*acq*(*threadB*,*m*)执行.又因为 33:*acq*(*threadC*,*n*)与 34:*acq*(*threadC*,*m*)同属一个线程,语句顺序在前的操作 33:*acq*(*threadC*,*n*)必然先于 34:*acq*(*threadC*,*m*)执行,由此可得 26:*acq*(*threadB*,*n*)需要先于 25:*acq*(*threadB*,*m*)执行.然而这是不可能的,因为作为同一线程 *threadB* 的两步操作,第 26 行语句对应的 26:*acq*(*threadB*,*n*)不可能先于第 25 行语句对应的操作 25:*acq*(*threadB*,*m*)执行.这说明本例中的"历史持有锁-持有锁"耦合因果依赖关系无法得到满足,此时,即使多个锁获取操作构成环路也无法真正触发死锁.已有的分段图和扩展锁图等模型均未对这种"历史持有锁-持有锁"耦合因果依赖关系进行刻画和可满足性判定,故导致了本例的第 2 种死锁误报.



(a) 分段图



(b) 扩展锁图

Fig.1 Segment graph and extended lock graph corresponding to the running trace in Table 2

图 1 表 2 运行轨迹对应的分段图和扩展锁图

一般而言,死锁误报产生的原因就是程序模型遗漏了上述某些因果依赖或互斥关系,本文将上述实例中展

现的各种因果和互斥关系进行处理.为此,首先定义多线程程序操作间的依赖和互斥关系如下.

定义 1(操作的依赖与互斥关系). 给定一个多线程程序的运行轨迹$\sigma=e_1e_2e_3...e_n$.

(1) 若事件 e_i 与 e_j 同属一个线程(即 $e_i.thread=e_j.thread$)且 $i<j$,则称 e_i 和 e_j 之间存在线程内因果依赖关系,记为 $e_i<_{thread}e_j$;

(2) 若 $e_i=start(u,v)$,存在 $k\leqslant i$ 使得 $e_k.thread=u$,存在 $m>i$ 使得 $e_m.thread=v$,则称 e_k 与 e_m 之间存在线程间因果依赖关系,记为 $e_k<_{start}e_m$;若 $e_i=join(u,v)$,存在 $k<i$ 使得 $e_k.thread=v$,存在 $m\geqslant i$ 使得 $e_m.thread=u$,则也称 e_k 与 e_m 之间存在线程间因果依赖关系,记为 $e_k<_{join}e_m$;

(3) 设 $e_i=acq(u,o)$,$e_j=rel(u,o)$,$i<j$,且 e_i 与 e_j 之间不存在事件 $rel(u,o)$,但存在事件 $e_m=start(u,v)$,则对于区间 $[m,j]$中的任意整数 k 与线程 v 的任意操作 e,只要 $e_k.thread=u$,就称 e_k 与 e 之间存在"锁-*start*"耦合因果依赖关系,记为 $e_k<_{lock\_start}e$;

统称前述 3 种关系为因果依赖,记为<;

(4) 设 $e_i=acq(u,o1)$,$e_j=acq(v,o2)$,且两者相互间不具备前述 3 种因果依赖关系,记线程 u 执行 e_i 时持有的锁集为 $lockSet(e_i)$,线程 v 执行 e_j 时持有的锁集为 $lockSet(e_j)$.若 $lockSet(e_i)\cap lockSet(e_j)\neq\varnothing$,则称 e_i 和 e_j 之间存在锁集互斥关系,记为 $e_i\#_{lockSet}e_j$;

(5) 设 $e_i=acq(u,o1)$,$e_j=acq(v,o2)$,且两者相互间不具备前述 3 种因果依赖关系和锁集互斥关系,记线程 u 获取 $lockSet(e_i)$中各把锁的过程中曾经获取过的所有锁的集合为 $lockSet\_OnceHeld(e_i)$,称之为 e_i 执行时的历史持有锁集;类似可得 e_j 对应的历史持有锁集 $lockSet\_OnceHeld(e_j)$.记 $Lock\_IntSecion=lockSet\_OnceHeld(e_i)\cap lockSet(e_j)$,若要使得 e_i 与 e_j 能并发以导致死锁,则对于$\forall o\in Lock_IntSecion$,线程 u 获取历史持有锁 o 的操作 $e_i'=acq(u,o)$ 应先于线程 v 最后一次获取持有锁 o 的操作 $e_j'=acq(v,o)$ 而执行,称 e_i' 与 e_j' 之间存在"历史持有锁-持有锁"耦合因果依赖关系,记为 $e_i'<_{lock\_held}e_j'$.

不难发现:对于上述 5 类关系,若遗漏线程内依赖关系,则可能导致单线程环死锁误报;若遗漏线程间因果依赖关系,则可能出现 *start*/*join* 导致的因果环死锁误报;若遗漏锁集互斥关系,则可能导致门锁环误报;若遗漏"锁-*start*"耦合因果依赖关系和"历史持有锁-持有锁"耦合因果依赖关系,则可能导致本文所给程序实例中的误报.已有的各类基于锁图及其各种变形模型的死锁动态分析方法主要识别和处理前 3 种关系导致的误报,难以解决本文给出的两种新型关系导致的误报.

针对上述问题,本文一方面在扩展锁图的基础上添加语句的执行时序信息,以区分循环中不同轮次的锁获取操作,据此提出时序增广锁图的概念;另一方面,在分段图的基础上扩充锁的申请和释放信息,以此识别锁获取和释放操作与线程 *start* 操作耦合引起的因果关系,在此基础上对分段图作进一步细化,由此提出了锁增广分段图的概念;最终,综合两个工具提出一种更为准确的死锁检测方法.下面给出具体介绍.

2 锁增广分段图与时序增广锁图的构造

本节对多线程程序的运行轨迹给出形式化定义,并对锁增广分段图和时序增广锁图的构造规则予以说明.

2.1 多线程程序的运行轨迹

定义 2(多线程程序运行轨迹). 对于一个多线程程序,其运行轨迹定义为死锁相关的并发原语在执行过程中形成的轨迹σ,严格定义如下:

$$\sigma\in ConPrimitive^*,$$

$$ConPrimitive::=start(u,v)|stop(v)|join(u,v)|acq(u,l)|rel(u,l).$$

其中,

- $ConPrimitive$ 表示各类并发原语的集合,$ConPrimitive^*$为并发操作构成的序列全体;
- $u,v\in Tid$ 表示线程,$l\in Lock$ 表示锁;
- $start(u,v)$:线程启动线程;

- *stop*(*v*):线程执行终止;
- *join*(*u*,*v*):线程等待线程执行终止;
- *acq*(*u*,*l*):线程锁对象;
- *rel*(*u*,*l*):线程释放获取锁对象.

表 2 给出的就是 Program 1 的一个程序运行轨迹.

2.2 锁增广分段图

本节在传统分段图的基础上扩充锁的申请和释放信息,并根据“锁-*start*”耦合因果依赖关系对段的划分进行细化,由此提出了锁增广分段图的概念.以表 2 中的程序运行轨迹为例,拟构造的锁增广分段图如图 2 所示.

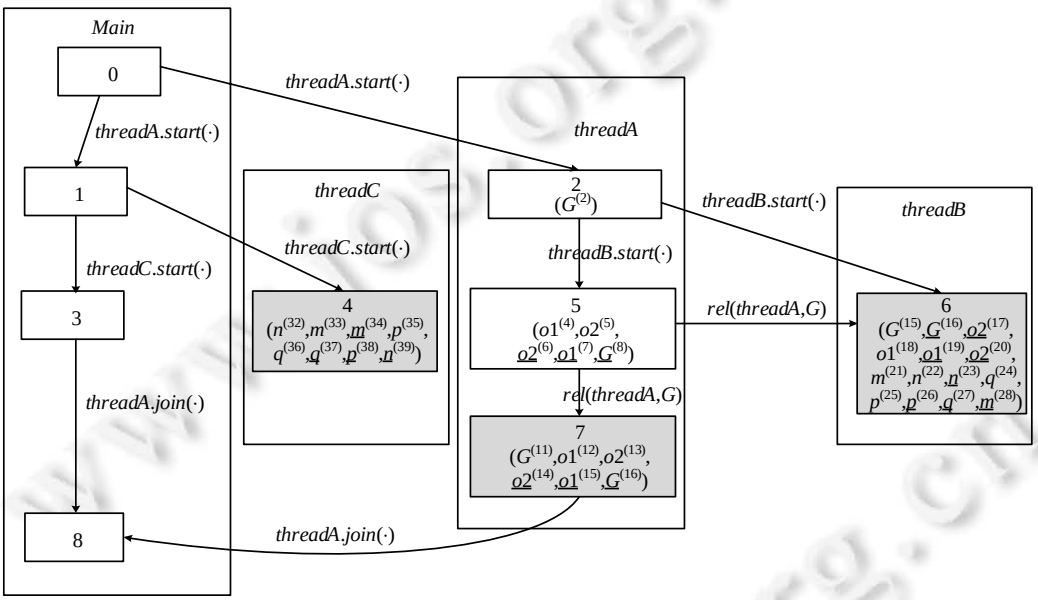


Fig.2 Lock-augmented segmentation graph corresponding to the running trace in Table 2

图 2 表 2 运行轨迹对应的锁增广分段图

在锁增广分段图中,每个段除含有段号信息外,增加了此段中所执行的锁授权和释放操作的信息.以 4 号段为例,它依次执行了获取锁 *n*、获取 *m*、释放 *m*、获取 *p*、获取 *q*、释放 *q*、释放 *p*、释放 *n* 的操作,这一过程通过标记($n^{(32)}$,$m^{(33)}$,$\underline{m}^{(34)}$,$p^{(35)}$,$q^{(36)}$,$\underline{q}^{(37)}$,$\underline{p}^{(38)}$,$\underline{n}^{(39)}$)刻画(标记中,各个锁对象右上角括号中的序号是本次操作在运行轨迹中的操作序号 ID,锁对象底部加下划线表示锁的释放操作,不加下划线时表示锁的获取操作).

此外,由于线程 *threadA* 在 2 号段获取了锁 *G* 而未释放,“*threadB* 在 6 号段开始处获取锁 *G* 的操作”只有待“*threadA* 于 5 号段释放锁 *G* 的操作”结束后方能执行,两者间构成一种因果关系.为了刻画这种关系,锁增广分段图中,将传统分段图中的 5 号段以锁 *G* 的释放操为分割点一分为二,并在新的 5 号段与 6 号段之间添加一条有向弧,据此刻画“锁-*start*”耦合导致的依赖关系.

对这种“锁-*start*”耦合因果依赖关系,锁增广分段图构造过程中的识别规则及对应的段/弧添加规则如下.

(1) 每当有一个线程 *t* 在段 *s* 中执行一次锁 *l* 的释放操作,检查 *t* 此前获取 *l* 的操作是否在同一个段 *s* 中:若不是,则在此处为线程 *t* 添加一个新段(例如,图 2 中 5 号段末尾锁 *G* 的释放导致了 7 号段以及 5 号段到 7 号段之间有向弧的添加);

(2) 每当有一个线程 *t*′在段 *s*′中执行一次锁 *l* 的获取操作,从该段节点逆向寻找第一个获取锁 *l* 的段节点 *s*″,若 *s*″不属于线程 *t*′且 *s*″中未释放 *l*(例如,图 2 中线程 *threadB* 获取锁 *G* 时,*s*″为 6,*s*′为 2),则:

(2.1) 将 *s*′从当前 *l* 的获取操作处开始一个新的分段,记段号为 *s*′_*new*(图 2 中 *G* 的获取操作位于 6 号

段的开头,此时无需添加新的分段,s'_new 为 6);

(2.2) 与 s''同属一个线程的、s''的子孙节点中必然存在一个以 l 的释放为尾操作和分段点的段 s''_desc(例如图 2 种的 5 号段),显然,s''_desc 与 s'_new 存在一种依赖关系,仅当 s''_desc 执行结束、释放 l 后,s'_new 中的操作方能执行,故在 s''_desc 与 s'_new 之间添加一条有向弧(例如,图 2 种由 5 号段到 6 号段之间的有向弧).

相比传统的分段图,锁增广分段图中添加的锁授权和释放信息有助于死锁检测过程中计算各个操作执行前历史曾经持有的锁信息,而且按前述规则对段进行细分后,能够刻画锁的释放和授权操作引起的段之间的因果关系,这为后续准确地进行死锁检测奠定了基础.锁增广分段图的形式化定义及构造规则如下(与传统分段图构造规则不同的部分加粗显示).

定义 3(锁增广分段图). 给定一个多线程程序的运行轨迹$\sigma=e_1e_2e_3...e_n$,其对应的锁增广分段图 $SegG\_Lock_\sigma$ 是满足如下条件的四元组$(SegSet_\sigma,SegR_\sigma,\varphi_s,\varphi_r)$,其中,$SegSet_\sigma$是顶点集,$SegR_\sigma\subseteq SegSet_\sigma\times SegSet_\sigma$是关系集,$\varphi_s$ 与 φ_r 分别是定义在顶点和边上的标签函数:

(1) $SegSet_\sigma$是顶点集,每个顶点对应程序的一个分段,其生成规则如下.

(a) 添加一个 0 号段作为主线程的初始分段;

(b) 每当有线程 u 执行操作 $start(u,v)$时,则在线程 u 此操作之处添加一个新段,并为线程 v 添加一个初始分段;

(c) 每当有线程 u 执行操作 $join(u,v)$时,则为线程 u 添加一个新分段;

(d) 每当有一个线程 u 在段 s 中执行操作 $rel(u,l)$时,若 u 此前获取 l 的操作不在段 s 中,则在此处为线程 u 添加一个新段;

(e) 每当有一个线程 v 在段 s'中执行操作 $acq(v,l)$时,从该段节点逆向寻找第一个获取锁 l 的段节点 s'',若 s''所属线程 $u\neq v$ 且 s''中未释放 l,则将 s'从当前 l 的获取操作处开始一个新的分段,记其段号为 s'_new(若 $acq(v,l)$是 s'的第 1 个操作,则不必添加新分段,令 $s'_new:=s'$即可);记与 s''同属线程 u 的、s''的子孙节点中第 1 个以 $rel(u,l)$为尾操作的段为 s''_desc,将(s''_desc,s'_new)加入关系集 $SegR_\sigma$并令这个关系对应的φ_r标签函数值为 $rel(u,l)$;

(f) 每个段设置一个全局唯一的段号作为其 ID(从 0 开始递增,每次增加 1);

(2) $SegR_\sigma\subseteq SegSet_\sigma\times SegSet_\sigma$记录段之间执行次序上的先后关系,$\varphi_r$ 是定义在 $SegR_\sigma$上的标签函数,他们的产生和定义规则如下.

(a) 每当有线程 u 执行操作 $start(u,v)$时,记线程 u 之前的最后一个段为 $latest\_Seg(u)$,记因为这个操作而为 u 和 v 添加的新段为 $new_Seg(u)$和 $new\_Seg(v)$,向 $SegR_\sigma$中添加两个新关系($latest\_Seg(u)$, $new\_Seg(u)$)和($latest\_Seg(u)$,$new\_Seg(v)$);并令这两个关系对应的φ_r标签函数值为 $start(u,v)$;

(b) 每当有线程 u 执行操作 $join(u,v)$时,记线程 u 之前的最后一个段为 $latest\_Seg(u)$,记因为这个操作而为 u 添加的新段为 $new\_Seg(u)$,向 $SegR_\sigma$中添加新关系($latest\_Seg(u)$,$new\_Seg(u)$);并令这该关系对应的φ_r标签函数值为 $join(u,v)$;

(c) 每当有线程 v 执行操作 $stop(v)$时,若之前曾有线程 u 执行操作 $join(u,v)$,并假设线程 u 由于操作 $join(u,v)$而新添加的分段为 $new\_Seg(u)$,同时记线程 v 的最后一个段为 $latest\_Seg(v)$,则向 $SegR_\sigma$中添加新关系($latest\_Seg(v)$,$new\_Seg(u)$);并令这两个关系对应的φ_r标签函数值为 $join(u,v)$;

(d) 每当有线程 u 因执行锁的释放 $rel(u,l)$操作而添加一个新段 s 时,记线程 u 之前的最后一个分段为 s',将(s',s)加入关系集合 $SegR_\sigma$中,并令这个关系对应的φ_r标签函数值为 $rel(u,l)$;

(e) 每当有线程 u 因执行锁的获取 $acq(u,l)$操作而添加一个新段 s 时,记线程 u 之前的最后一个分段为 s',将(s',s)加入关系集合 $SegR_\sigma$中,并令这个关系对应的φ_r标签函数值为 $acq(u,l)$;

(3) φ_s是定义在 $SegSet_\sigma$上的标签函数,其生成规则如下.

(a) 每个段最初生成时,其φ_s标签函数值为空;

(b) 每当有线程 u 执行操作 $acq(u,l)$时,假设该操作当前所在的段为 $currentSeg$,该操作的序号 ID 为 k,则令$\varphi_s(currentSeg):=\varphi_s(currentSeg)\circ l^{(k)}$(其中,$\circ$表示字符串的拼接函数);

(c) 每当有线程 v 执行操作 $rel(u,l)$时,假设该操作当前所在的段为 $currentSeg$,该操作的序号 ID 为 k,则令$\varphi_s(currentSeg):=\varphi_s(currentSeg)\circ \underline{l}^{(k)}$.

以表 1 中的程序运行轨迹为例,按照上述定义得到的锁增广分段图如图 2 所示.

显然,锁增广分段图中的每个段对应一个线程的若干操作.若图中的两个段 s_1 和 s_2 满足 $(s_1,s_2)\in SegR_\sigma^+$(其中,$^+$表示关系的闭包运算),则仅当 s_1 中的操作全部执行完毕后方能执行 s_2 中的操作,记此种关系为 $s_1\triangleright s_2$.

图 2 中,$2\triangleright 6$ 与 $5\triangleright 6$ 成立,意味着仅当 2 号段与 5 号段中操作执行完毕后方能执行 6 号段中的操作;$6\triangleright 7$ 与 $7\triangleright 6$ 均不成立,这意味着,6 号段与 7 号段之间的操作不存在执行次序上的先后依赖关系,此时认为这两组操作可以并发.

2.3 时序增广锁图

本节在扩展锁图的基础上添加语句的执行时序信息,以区分循环中不同轮次的锁获取操作;同时,在图中标注每个操作在运行轨迹中的操作序号 ID(据此定位操作在分段图中的位置),由此提出时序增广锁图的概念.

以表 2 中的程序运行轨迹为例,拟构造的时序增广锁图如图 3 所示.其中,2 号弧的标记信息由扩展锁图图 1(b)中的$\langle 5,(threadA,\{G,o1\}),5\rangle$扩充为了$\langle 5,(threadA,\{G,o1\}),1,5\rangle^{(5)}$,扩充图中的数字 1 表示本次操作是线程 $threadA$ 第 1 次申请锁 $o2$,标记右上角数字 5 为该弧所对应操作在执行轨迹上的操作序号 ID.5 号弧的标记信息变为了$\langle 7,(threadA,\{G,o1\}),2,7\rangle^{(11)}$.在扩展锁图中标记相同、无法区分的 2 号弧和 5 号弧,在图 3 的时序增广锁图中有了不同的标记.

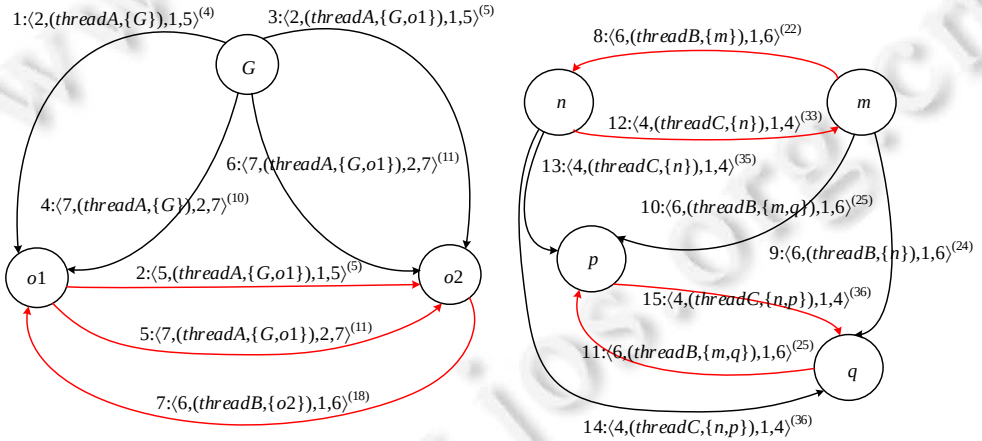


Fig.3 Time-augmented lock graph corresponding to the running trace in Table 2

图 3 表 2 运行轨迹对应的时序增广锁图

经上述处理后,一方面可以保证同一个锁授权语句在循环中的多次执行能区分开来;另一方面,可根据各条弧所对应操作在运行轨迹中的序号 ID 确定其在锁增广分段图中的位置,从而为后续死锁检测提供方便.下面给出时序增广锁图的形式化定义与构造规则.

定义 4(时序增广锁图). 给定一个多线程程序的运行轨迹$\sigma=e_1e_2e_3...e_n$,其对应的时序增广锁图 $LockG_Order_\sigma$是满足如下条件的二元组(V_σ,R_σ).

(1) V_σ为顶点集,每个顶点对应σ中出现的一个锁对象;

(2) $R_\sigma\subseteq V_\sigma\times\Gamma\times V_\sigma$为关系集,$\Gamma$是图中各关系所对应之有向弧的标签集合;

(3) 每当σ中有一个线程 t 在持有锁 $lock_1$ 的前提下获取锁 $lock_2$ 时,假设该操作的序号 ID 为 k,则为此操作

添加一个关系$(lock_1,\gamma,lock_2)\in R_\sigma$,其中的$\gamma$形如:

$$\langle seg_1ID,(t,lockSet),m,seg_2ID\rangle^{(k)},$$

其中,seg_1ID 是线程 t 获取锁 $lock_1$ 时所在的段号,seg_2ID 是线程 t 获取锁对象 $lock_2$ 时所在的段号,$lockSet$ 是线程 t 获取锁 $lock_2$ 前持有的锁集,m 记录线程 t 的当前操作是其第几次获取锁 $lock_2$;

(4) 是满足上述条件的最小有向图.

以表 2 中的程序运行轨迹为例,按照上述定义得到的时序增广锁图如图 3 所示.

显然,时序增广锁图中任意两点间若存在有向环,则它对应着一组锁对象的循环持有和等待.例如,图 3 中共有 4 组有向弧对{2,7},{5,7},{8,12},{11,15}构成有向环.传统的基于锁图以及基于分段图和扩展锁图的死锁检测方法会判定这 4 组有向环均导致死锁.实际上,如第 1 节所述,它们当中只有两个是真实死锁,而另两个是误报.基于本文提出的时序增广锁图和锁增广分段图可以排除上述误报.下节给出具体方法.

3 基于锁增广分段图与时序增广锁图的死锁检测

传统的基于分段图和扩展锁图的死锁检测方法通过如下规则识别死锁:(1) 锁图中多个锁的申请操作构成一条有向环;(2) 锁申请操作分属于不同的线程;(3) 记任意两个锁操作所在的段为 s_1 和 s_2,则分段图中 s_1 和 s_2 之间不存在有向路相连,意即这些操作在执行顺序的先后上无因果依赖关系;(4) 锁操作执行时的持有锁集不相交.

第 1 个规则要求多个锁申请操作满足锁对象的环路等待条件,后 3 个规则本意是要保证多个锁申请操作能够并发.然而,上述识别规则仅在部分情况下是正确的.例如,图 3 右侧子图中{5,7},{8,12}两组有向弧对满足上述规则,它们对应的操作既构成锁对象的环路等待,又可以并发,故对应两个真实死锁.但是存在某些情况,虽然前述规则均满足,但它们却不能并发,从而会导致死锁误报,具体如下.

(1) 以图 3 左侧子图中 ID 为 2 和 7 的两条有向弧为例,它们对应的 5 号段与 6 号段虽然在图 1(a)的传统分段图中不存在有向路径相连,但是如第 1.2 节所述,两者却因为锁对象 G 而存在“锁-*start*”耦合因果依赖关系(仅当弧 2 对应的操作执行完并释放锁 G 后,弧 7 对应的操作方能执行).这种因果关系在传统分段图中得不到体现,但是,如第 2.2 节所述,锁增广分段图通过将 5 号段细分为两个段,并在细分后的段间添加依赖关系解决了这一问题(图 1 中的 5 号段在图 2 中细分为了段 5 和段 7,而且段 5 和段 6 间添加了一条依赖关系).因此,对这一误报,只要基于锁增广分段图对段间的依赖关系进行更为精准的判定,便可消除之.例如,在图 2 中,ID 为 2 的有向弧对应的操作属于 5 号段,ID 为 7 的有向弧对应的操作属于 6 号段,$5\triangleright 6$ 成立,由此可断定{2,7}不会导致死锁;

(2) 锁申请操作执行时的持有锁集互不相交,但持有锁集与历史持有锁集却可能相交.此时可能存在“历史持有锁-持有锁”耦合因果依赖关系,这种关系有时会导致操作无法并发.以图 3 中 ID 为 11 和 15 的弧为例,它们所对应操作的持有锁集分别为$\{m,q\}$和$\{n,p\}$,持有锁集互不相交,看似不存在门锁而可并发.然而如第 1.1 节所述,这两个操作却因为其持有锁集与历史持有锁集中的公共元素 m 和 n 无法满足“历史持有锁-持有锁”耦合因果依赖关系而无法并发.

为排除上述死锁误报,除记录操作执行时持有的锁集 *lockSet* 外,还要计算它们的历史持有锁集 *lockSet_OnceHeld*.不难发现,*lockSet_OnceHeld* 可通过锁增广分段图计算得到:首先,根据操作 ID 定位其在锁增广分段图的位置,假设其对应的段号为 x、在段标签中的序号为 *index*,并假设该操作属于线程 t;接下来,从中 *index*–1 位置开始,按照段内标签从右向左、一个段访问结束后访问线程 t 的上一段这种规则,遍历各段的标签,直至 *lockSet* 中所有锁都在遍历过程中被获取为止.上述遍历过程中,曾出现过的锁对象的集合即为 *lockSet_OnceHeld*.以图 3 中 ID 为 11 的弧为例,它属于线程 *threadB*,对应锁增广分段图中的 6 号段,该操作执行时的持有锁集 $lockSet_{11}=\{m^{(21)},q^{(24)}\}$.根据前述规则,需访问的标签依次为 $q^{(24)},\underline{n}^{(23)},n^{(22)},m^{(21)}$,由此可得 $lockSet\_OnceHeld_{11}=\{q^{(24)},n^{(22)},m^{(21)}\}$.类似地,对图 3 中 ID 为 15 的弧,它对应线程 *threadC* 段 4 中的一个操作,操作执行时的持有锁集 $lockSet_{15}=\{n^{(32)},p^{(35)}\}$,需访问的标签依次为 $p^{(35)},\underline{m}^{(34)},m^{(33)},n^{(32)}$,由此可得 $lockSet\_OnceHeld_{15}=\{p^{(35)},m^{(33)},n^{(32)}\}$.

显然,$lockSet\_OnceHeld_{11}\cap lockSet_{15}=\{n\}$,$lockSet\_OnceHeld_{15}\cap lockSet_{11}=\{m\}$.欲使弧 11 和弧 15 对应的操作能够并发以触发死锁,则这两个交集中的锁对象获取操作应满足“历史持有锁-持有锁”耦合因果依赖.为此,定义一种“历史持有锁-持有锁”耦合因果关系图,据此判定这种关系的可满足性:

定义 5(“历史持有锁-持有锁”耦合因果关系图). 设 $c=e_{k\_1}e_{k\_2}...e_{k\_l}$ 构成时序增广锁图中的一条有向环路,其中每个 e_{k_i} 是一条弧对应的锁申请操作.称满足如下条件的二元组 $DependG\_Lock_c=(V_c,R_c)$ 为 c 对应的“历史持有锁-持有锁”耦合因果关系图,其中,V_C 为顶点集,每个顶点对应一个锁获取操作;点 $R_c\subseteq V_c\times V_c$ 为有向边的集合,每条有向边用以刻画两个弧顶点所对应操作之间的因果依赖关系.V_C 与 R_C 的产生规则如下.

(1) 只要存在锁对象 o 与操作 $e_{k\_i}=acq(u,o)$,$e_{k_j}=acq(v,o)$满足 $o\in lockSet_OnceHeld(e_{k_i})\cap lockSet(e_{k_j})$且 $u\neq v$,记线程 u 获取历史持有锁 o 的各个操作构成的集合为 $E\_OnceHeld(e_{k\_i},o)$,线程 v 最后一次获取持有锁 o 的操作为 e'_{k_j},则:

(1.1) 对应线程 v 最后一次获取持有锁 o 的操作 e'_{k_j},向 V_C 中添加一个 ID 为$(v,o^{(y)})$的操作,其中,y 为操作 e'_{k_j} 在程序运行轨迹中的序号 ID;

(1.2) 对应线程 u 每一个获取历史持有锁 o 的操作 $e'_{k\_i}\in E\_OnceHeld(e_{k\_i},o)$,向 V_C 中添加一个 ID 为$(u,o^{(x)})$的操作,其中,x 为操作 e'_{k_i} 在程序运行轨迹中的序号 ID;与此同时,向 Rc 中添加一条由顶点$(u,o^{(x)})$指向顶点$(v,o^{(y)})$的有向边;

(2) 对 Vc 中任意两顶点$(u,p^{(i)})$与$(v,q^{(j)})$,若 $u=v\wedge i<j$,则向 Rc 中添加一条由顶点$(u,p^{(i)})$指向顶点$(v,q^{(j)})$的有向边.

不难发现:定义 5 中的规则(1.2)添加的有向边用以刻画第 1 节给出的“历史持有锁-持有锁”耦合因果依赖关系,规则(2)添加的有向边用以刻画第 1 节给出的线程内因果依赖关系.若要使得环路 c 对应一个真实死锁,则这些关系应该同时得到满足.而这些因果同时得到满足的一个必要条件就是“历史持有锁-持有锁”耦合因果关系图中不存在有向环,因此,对于时序增广锁图中的任意一条有向环路 c 而言,若其对应的“历史持有锁-持有锁”耦合因果关系图中存在有向环,则 c 对应的潜在死锁是一个误报.

以图 3 时序增广锁图中有向弧 11 与 15 构成的有向环路为例,它们对应的操作分别为 e_{26}=28:$acq(threadB,p)$和 e_{36}=36:$acq(threadC,q)$.不难发现,$lockSet\_OnceHeld(e_{26})=\{m,n,q\}$,$lockSet(e_{36})=\{n,p\}$.显然,$threadB$ 与 $theadC$ 是两个不同的线程,且存在锁对象 $n\in lockSet\_OnceHeld(e_{26})\cap lockSet(e_{36})$,规则(1)的条件满足;接下来,根据规则(1.1),应向 $DependG\_POrder_c$ 中添加 ID 为$(threadC,n^{(32)})$的顶点,它对应 $threadC$ 最后一次获取持有锁 n 的操作;根据规则(1.2),向 $DependG\_POrder_c$ 中添加 ID 为$(threadB,n^{(22)})$的顶点,它对应 threadB 获取历史持有锁 n 的操作;再根据规则(1.2)添加一条由$(threadB,n^{(22)})$指向$(threadC,n^{(32)})$的有向边.类似可见,$m\in lockSet\_OnceHeld(e_{26})\cap lockSet(e_{36})$,根据规则(1),应向 $DependG\_Lock_c$ 添加顶点$(threadC,m^{(33)})$,$(threadB,m^{(21)})$以及两者之间的一条有向边.再根据规则(2),应根据线程内因果关系添加由$(threadB,m^{(21)})$指向$(threadB,n^{(22)})$以及由$(threadC,n^{(32)})$指向$(threadC,m^{(33)})$的两条有向边,最终可得图 4 中的“历史持有锁-持有锁”耦合因果关系图.

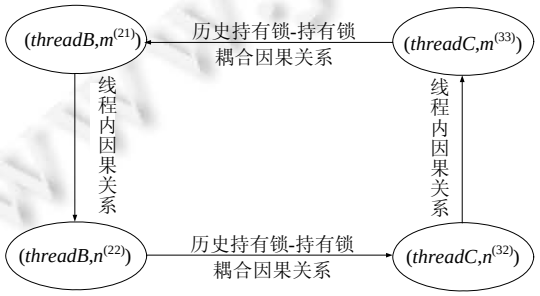


Fig.4 Partial order dependency graph of deadlock detection corresponding to loops {11,15} in Fig.3

图 4 弧{11,15}构成之有向环路对应的“历史持有锁-持有锁”耦合因果关系图

显然,图 4 所示“历史持有锁-持有锁”耦合因果关系图存在有向环,这意味着图中各操作之间的线程内因果

依赖关系与“历史持有锁-持有锁”耦合因果依赖关系无法同时满足,从而意味着“ID 为 11 的弧对应的锁获取操作”与“ID 为 15 的弧对应的锁获取操作”无法并发,故弧{11,15}构成之有向环路对应的潜在死锁非真实死锁.

基于上述分析,可得基于锁增广分段图和时序增广锁图的死锁检测算法如下,检测本质是如下 5 条规则.

(1) 时序增广锁图中多个锁申请操作构成一条有向环路(对应算法第 1 步);

(2) 环路中的锁申请操作分属于的不同的线程(对应算法第 3 步);

(3) 环路中任意两个锁申请操作无线程间因果依赖和“锁-*start*”耦合因果依赖,即,在锁增广分段图中无路径相连(对应算法第 17 步);

(4) 环路中的任意两个锁申请操作对应的持有锁集互不相交(对应算法第 28 步);

(5) 环路对应的“历史持有锁-持有锁”耦合因果关系图不存在有向环(对应算法第 30 步).

算法 1. 基于锁增广分段图和时序增广锁图的死锁检测.

输入:锁增广分段图 $SegG\_Order_\sigma$,时序增广锁图 $LockG_Order_\sigma$及其中的环路集合 $Cycles(LockG\_Order_\sigma)$;

输出:LG_Order_σ中与潜在死锁对应的环路的集合 $DeadlockCycles(LockG\_Order_\sigma)$.

步骤:

1. **FOR** EACH ($c\in Cycles(LockG\_Order_\sigma)$){
2. 记 c 中的有向弧集合为$\varepsilon=\{\varepsilon_1,\varepsilon_2,\varepsilon_3,...,\varepsilon_n\}$;
3. **IF** (ε中各条弧对应的操作都属于不同的线程){
4. *SegDepFLAG*:=0; //*SegDepFLAG* 用以标记ε中各操作在锁增广分段图中是否具有因果依赖关系
5. **FOR** EACH ($\varepsilon_i\in\varepsilon$){
6. **FOR** EACH ($\varepsilon_j\in\varepsilon$){
7. **IF** ($i\neq j$){
8. 记ε_i 和ε_j 对应的操作在段图中所属段的段号为 s_i 和 s_j;
9. **IF** (($s_i \triangleright s_j$)$\vee$($s_j \triangleright s_j$)){ //在锁增广段图中两个操作存在执行顺序上的先后依赖关系
10. *SegDepFLAG*:=1; //*SegDepFLAG* 为 1,意味着两个操作在分段图中存在因果依赖
11. **BREAK**;
12. } //**IF**
13. } //**IF**
14. } //**FORE** ACH
15. **IF** (*SegDepFLAG* ==1) **BREAK**;
16. } //**FOR** EACH
17. **IF** (*SegDepFLAG*==0){ //环路中任意两个操作在锁增广段图中均不存在执行因果依赖
18. *LockExclusiveFlag*=0; //用以标记环路中的锁操作是否存在锁集互斥关系
19. **FOR** EACH ($\varepsilon_i\in\varepsilon$){
20. **FOR** EACH ($\varepsilon_j\in\varepsilon$){
21. **IF** ($i\neq j,\&\&(lockSet_{\varepsilon_i}\cap lockSet_{\varepsilon_j}\neq\varnothing)$) {
22. *LockExclusiveFlag*=1; //表示任意两个操作对应的持有锁集的交集不为空
23. **BREAK**
24. } //**IF**
25. } //**FOR** EACH
26. **IF** (*LockExclusiveFlag*==1) **break**;
27. } //**FOR** EACH
28. **IF** (*LockExclusiveFlag*==0){ //环路中的操作相互之间均不存在锁集互斥关系
29. 根据定义 4 构造环路 c 对应的“历史持有锁-持有锁”耦合因果关系图 $DependG\_Lock_c$;

30. **IF** (*DependG_Lock*$_c$ 不存在有向环)
31. 将 *c* 加入集合 *DeadlockCycles*(*LockG_Order*$_\sigma$); //得到一个潜在死锁
32. } //**IF**
33. } //**IF**
34. } //**IF**
35. } //**IF**
36. } //**FOR** EACH
37. **RETURN** *DeadlockCycles*(*LockG_Order*$_\sigma$).

相比传统方法基于分段图和扩展锁图进行的死锁检测,算法 1 的规则(3)基于锁增广分段图进行的段间因果依赖的判定更加准确,第 5 条规则能排除“历史持有锁-持有锁”耦合因果依赖关系导致的误报.如此一来,对于图 3 中 4 组有向弧对{2,7},{5,7},{8,12},{11,15}构成的有向环路,算法 1 能准确地识别出其中{5,7}和{8,12}对应的环路会导致死锁,而{2,7}和{11,15}对应的环路不会导致死锁,从而避免了它们给传统方法带来的死锁误报.

4 相关工作对比与实验评估

4.1 相关工作对比

基于锁图、环锁依赖链、分段图与分段锁图以及本文方法对表 2 中的程序运行轨迹进行分析,所得死锁检测结果见表 3.除本文方法之外,其余 3 类方法会将锁图中的每条环路都认定为一个死锁.第 1 行对应的死锁误报,是因为它们无法识别“锁-*start*”耦合因果依赖关系;第 4 行对应的死锁误报,是因为它们未处理“历史持有锁-持有锁”耦合因果依赖关系.而实际上,这两类关系都可能导致环路中的某些操作无法并发.本文方法通过对传统锁图和分段图的改进弥补了上述不足,并借助锁增广分段图和“历史持有锁-持有锁”耦合因果关系图对上述情形进行了识别,从而能避免了传统方法中存在的上述死锁误报现象.

Table 3 Deadlock detection results of different methods towards dynamic analysis of the trace in Table 1

表 3 不同方法对表 1 运行轨迹进行动态分析所得之死锁检测结果

锁对象的循环等待 (对应潜在死锁状态)	时序增广锁图中对应的操作与弧	是否真实死锁	不同模型的死锁检测结果			
			锁图	环锁依赖链	分段图与分段锁图	本文方法
threadA 第 1 次执行第 14 行代码 *threadB* 执行第 22 行代码	$acq(o2)@2{:}\langle 5,(threadA,\{G,o1\}),1,5\rangle^{(5)}$ $acq(o1)@7{:}\langle 6,(threadB,\{o2\}),1,6\rangle^{(18)}$	**否**	是	是	是	**否**
threadA 第 2 次执行第 14 行代码 *threadB* 执行第 22 行代码	$acq(o2)@5{:}\langle 7,(threadA,\{G,o1\}),2,7\rangle^{(11)}$ $acq(o1)@7{:}\langle 6,(threadB,\{o2\}),1,6\rangle^{(18)}$	**是**	是	是	是	**是**
threadB 执行第 25 行代码 *threadC* 执行第 33 行代码	$acq(n)@8{:}\langle 6,(threadB,\{m\}),1,6\rangle^{(22)}$ $acq(m)@12{:}\langle 4,(threadC,\{n\}),1,4\rangle^{(33)}$	**是**	是	是	是	**是**
threadB 执行第 27 行代码 *threadC* 执行第 35 行代码	$acq(p)@11{:}\langle 6,(threadB,\{m,q\}),1,6\rangle^{(25)}$ $acq(q)@15{:}\langle 4,(threadC,\{n,p\}),1,4\rangle^{(36)}$	**否**	是	是	是	**否**

需要进一步指出的是:动态分析方法从程序运行轨迹中捕获的程序行为信息始终有限,丢失的信息既可能导致检测出的潜在死锁不是真实死锁,还可能导致原本不存在死锁的程序被误定为存在死锁.针对这种情况,在死锁分析领域,除上述提到的死锁检测方法外,有学者开始研究潜在死锁的重演.所谓死锁重演,就是针对检测到的潜在死锁,通过对程序执行的主动干预和调度,使潜在死锁在为真实死锁的情况下会被尽可能地触发,重演成功的潜在死锁必是真实死锁.例如,对本文提出的“历史持有锁-持有锁”耦合因果依赖导致的死锁误报,文献[21]中曾有一个程序实例.但文献[21]中采用的死锁检测算法 ConLock+并不能排除该误报,文中是通过死锁重演来排除这一误报的.相比而言,本文提出的死锁检测算法无需重演,在检测阶段即可将该误报排除.不过,即便如此,由于程序轨迹中不可避免地会丢失一些程序行为信息,即使本文方法也存在某些类似的误报现象,所以死锁的重演也是本文后续研究的重点方向之一.

在死锁重演方面,文献[20,22]首次提出一种面向缺陷重演的随机调度方法,通过不同调度方案下程序的大

量运行,来尽量触发死锁等并发缺陷.该方法面向缺陷重演的调度是盲目的、完全随机的,考虑到死锁通常是低概率事件,这导致该方法在效率和可靠性方面的不足.相比而言,文献[14,18,21,23–25]先进行潜在死锁的检测,之后有针对性地设计程序调度方案,以此提高真实死锁重演成功的概率.具体地说,文献[14,18,23]面向潜在死锁提出一种启发式调度策略,在线程到达潜在的死锁点时挂起线程,以此增加死锁触发的概率.不过,这种方法本质上仍是随机的,真实死锁通常需多次运行才能重演成功,而且它们不能提供系统的、完整的覆盖.考虑到死锁的触发应同时考虑死锁的发生位置和该位置之前一些相关的锁授权操作,文献[24]提出一种基于屏障的死锁重演调度算法 ASN,将程序调度的干预时机进行了优化.类似地,文献[21,25]针对潜在死锁生成一组程序调度的约束集,当程序的运行不符合生成的约束时,挂起线程的执行.上述方法提高了真实死锁重演成功的概率,不过,其面向死锁重演生成的程序调度方案不够直观,而且同前述潜在死锁的检测方法一样,均对同一个锁授权语句的多次执行不加区分,这也为死锁重演带来了困难.

本文所提出的死锁检测工具,一方面能对循环中同一个锁授权语句的多次执行加以区分;另一方面,锁增广分段图中既对不同线程中的各个操作根据执行顺序上的依赖关系进行了刻画,还完整记录了锁的获取和释放操作.基于它们,有望为死锁的重演生成一种更为直观可行的重演方案,篇幅所限,具体方法将在以后展开.

4.2 原型系统与实验评估

4.2.1 原型系统

作者基于开源的 Java 多线程程序主动调试平台 CalFuzzer[26]开发了相应的死锁检测工具.基于该工具,用户只需要输入一个多线程程序,通过分析 CalFuzzer 产生的程序运行事件流,工具便能自动生成相应的时序增广锁图和锁增广分段图,并给出死锁检测的结果.所开发工具的架构和运行界面如图 5、图 6 所示.

原型系统架构分为 3 层:最底层为用户输入层,需要用户输入一个多线程程序,然后运用 Calfuzzer 挖掘出程序的运行轨迹;中间层为处理层,处理层接收到运行轨迹,进行模型挖掘,构建时序增广锁图和锁增广分段图,并基于算法 1 的检测规则识别潜在死锁;界面层向用户展示挖掘得到的锁增广分段图、时序增广锁图,在显示模型的同时给出死锁检测的结果(如图 5 所示).

工具运行界面截图中,位于左侧的是锁增广分段图,在锁增广分段图中,同一颜色的段号代表属于同一个线程,其中段中所记录的锁获取释放信息可设置为对隐藏或显示;位于右上的是时序增广锁图;位于右下的是用本文方法检测到的潜在死锁信息.图中给出的是对论文中程序实例 Program 1 的死锁检测结果(如图 6 所示).

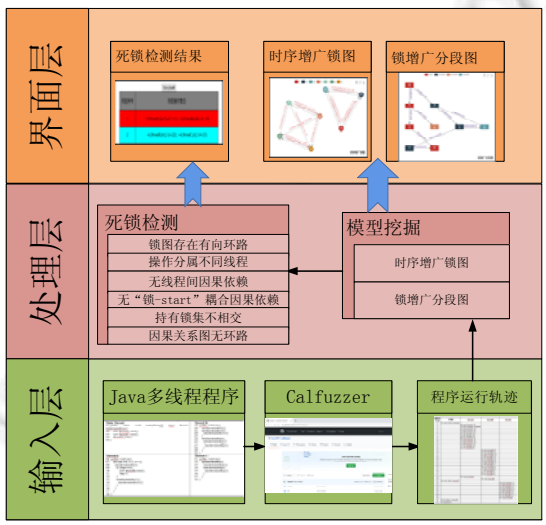


Fig.5 Architecture of the deadlock detection prototype system

图 5 死锁检测原型系统架构

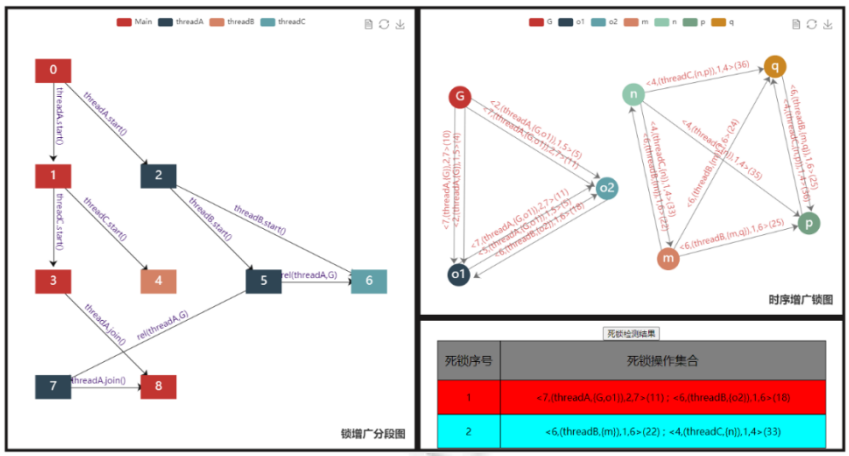


Fig.6 Running interface of the prototype system

图 6 原型系统运行界面

4.2.2 实验评估

第 1 节结合多线程程序 Program 1,从原理上说明了本文方法相比 iGoodlock 等经典死锁分析方法的优势.本节结合表 4 中更多的 Java 多线程程序实例进行实验评估,表中前 8 个样例来自于 CalFuzzer 开源项目中提供的程序样例;pingpong,cyclicDemo 来自于文献[27];ConLockDemo 为文献[21]设计的程序实例,其中包含本文要解决的“历史持有锁-持有锁”耦合因果依赖导致的死锁误报,文中通过重演的方法排除误报,检测阶段无法排除;PetriDeadLockDemo 包含“锁-*start*”耦合因果依赖关系导致的传统方法的死锁误报,文中通过 Petri 网行为分析排除误报,效率较低;OnceHeldLockDemo 是在本文 Program 1 基础上添加延时等操作后得到的完整程序实例,它既包含“锁-*start*”耦合因果依赖导致的死锁误报,也包含“历史持有锁-持有锁”耦合因果依赖导致的误报.以上程序实例仅涉及两个线程导致的死锁.为进行更一般化的验证,MulThreadDeadlockDemo 是专门设计的一个多线程死锁的程序实例,其中包含一个 3 线程导致的真死锁和 4 线程导致的死锁误报.

Table 4 Experimental results of program examples

表 4 程序实例实验结果

Java 多线程程序样例	Java 多线程程序包含的线程总数	潜在死锁数		死锁误报数		时间开销(s)		内存开销(MB)	
		iGood-lock	本文方法	iGood-lock	本文方法	iGood-lock	本文方法	iGood-lock	本文方法
Test1a	5	1	1	0	0	0.198	**0.157**	**3.695**	5.219
Test1b	3	1	1	0	0	0.129	0.115	**3.656**	4.842
Test2a	5	0	0	0	0	**0.113**	0.171	**4.020**	6.551
Test3	3	1	1	0	0	0.190	**0.144**	**3.621**	4.960
Test4	3	1	1	0	0	0.196	**0.122**	**3.641**	4.594
Test6	5	0	0	0	0	**0.113**	0.146	**3.785**	5.158
Test7	3	1	1	0	0	0.190	**0.153**	**3.523**	5.233
Test8	5	4	4	0	0	0.202	0.200	**3.566**	4.892
pingpong	5	0	0	0	0	0.162	0.156	**3.910**	4.486
cyclicDemo	5	0	0	0	0	0.130	0.111	**4.286**	5.009
ConLockDemo	3	4	**3**	1	**0**	0.192	0.176	**3.884**	4.730
PetriDeadLockDemo	3	2	**1**	1	**0**	0.135	0.112	**3.898**	4.812
OnceHeldLockDemo	4	4	**2**	2	**0**	2.158	**2.118**	**3.904**	4.956
MulThreadDeadlockDemo	4	2	**1**	1	**0**	0.726	**0.235**	**3.816**	5.026

注:1. PetriDeadLockDemo 来自链接 http://kns.cnki.net/kcms/detail/11.5946.TP.20210421.1820.049.html 的 Program 1;
2. OnceHeldLockDemo 的链接 https://pan.baidu.com/s/14jUjNflAQ1uyk1N-X_NFcw 提取码:og9n;
3. MulThreadDeadlockDemo 的链接 https://pan.baidu.com/s/1-ng2uVOuOlgYwizM_zLy8g 提取码:zquj

具体实验结果见表 4.下面从死锁检测准确性、时间和内存开销这 3 个方面进行实验对比,对比结果优势明显的项目用加粗显示.对照的死锁检测基准程序为 CalFuzzer 自身集成的死锁动态分析工具 iGoodlock.实验环

境中,CPU 为 Intel(R) Core(TM)i5-6300HQ CPU@2.30GHz,内存为 8.00GB,操作系统为 Ubuntu 18.10.

就死锁检测的准确性而言,iGoodlock 仅消除了单线程环和门锁环导致的误报,而本文方法进一步消除了线程间因果依赖关系、“锁-*start*”耦合因果依赖关系以及“历史持有锁-持有锁”耦合因果关系导致的误报.就死锁检测的时间性能而言,当程序中存在潜在死锁时,本文方法的时间性能要优于 iGoodlock.一个主要原因是:本文基于文献[28]所提出的图矩阵化方法计算锁增广分段图中的环路,其时间复杂度是多项式级的,而 iGoodlock 基于图的遍历完成环路检测,其时间复杂度是指数级的.当程序中不存在潜在死锁时,矩阵化方法检测环路与遍历法检测环路的时间开销接近,此时,因本文需要构造的锁增广分段图和时序增广锁图相比 iGoodlock 要构造的环锁依赖链含有更多的信息,同时,本文方法还需构建“历史持有锁-持有锁”耦合因果图并要判断其中是否包含环路,故本文方法的时间性能此时会稍差.就空间性能而言,本文方法比 iGoodlock 消耗更多的内存:一方面,本文需要构建的锁增广分段图和时序增广锁图相比 iGoodlock 构建的环锁依赖链包含了锁的获取和释放以及操作时序等额外信息,这会占用更多的内存;另一方面,本文检测死锁过程中构造的“历史持有锁-持有锁”耦合因果关系图也是 iGoodlock 等传统方法中不曾有的,它也会引起更大的内存开销.

5 总结与展望

围绕多线程程序的死锁检测问题,通过对传统动态死锁分析方法所存误报现象的分析,本文对锁图、分段图等传统的死锁分析模型进行改进:一方面,在扩展锁图的基础上添加语句的执行时序信息来区分同一循环中不同轮次的锁获取操作,据此提出了时序增广锁图的概念;另一方面,在分段图的基础上扩充锁的获取和释放信息,并对段进行更细粒度的划分以刻画锁对象的释放和获取操作导致的因果依赖关系,据此提出了锁增广分段图的概念.最终,在上述两个模型的基础上提出了一种新的死锁检测方法,它不仅可以消除单线程环、门锁环、多线程间由于线程的 *start* 和 *join* 操作而引起的因果关系导致的死锁误报,还能消除由于“锁-*start*”耦合因果依赖导致的死锁误报,而且能排除“历史持有锁-持有锁”耦合因果依赖导致的误报,由此减少了死锁的误报,提高了死锁检测的准确率.然而,本文方法也存在一定不足:一方面,本文仅考虑了线程的 *start*/*stop*/*join* 以及锁对象的释放和获取等并发原语,而实际上,线程的 *wait*/*notify*/*notifyAll* 等并发原语和其他一些程序的条件控制结构等操作也会对死锁产生影响,要进行更为准确的死锁检测,需要考虑更多的可能影响死锁的程序原语;另一方面,程序自动修复也是当前的研究热点之一[29],而如何基于本文的模型开展上述研究值得探究;最后,死锁同样存在于 MPI 并行程序[30]、柔性制造系统[31,32]、软件定义网络[33]等多种领域,如何将本文方法像更多的领域推广也是后续重点需要研究的工作.

References:

[1] Su XH, Yu Z, Wang TT, Ma PJ. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015,395(11):93−111 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2015.02215]

[2] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019,30(1):80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[3] Lu S, Park S, Seo E, Zhou YY. Learning from mistakes—A comprehensive study on real world concurrency bug characteristics. Computer Architecture News, 2008,36(1):329−339.

[4] Bensalem S, Griesmayer A, Legay A, Nguyen TH, Peled D. Efficient deadlock detection for concurrent systems. In: Proc. of the 9th ACM/IEEE Int'l Conf. on Formal Methods and Models for Codesign. IEEE Computer Society, 2011. 119−129. https://doi.org/10.1109/MEMCOD.2011.5970518

[5] Sun J, Liu Y, Dong JS, Liu Y, Shi L, Andre E. Modeling and verifying hierarchical real-time systems using stateful timed CSP. ACM Trans. on Software Engineering and Methodology, 2013,22(1):1−29. [doi: 10.1145/2430536.2430537]

[6] Artho C, Biere A. Applying static analysis to large-scale, multi-threaded Java programs. In: Proc. of the 13th Australian Conf. on Software Engineering. IEEE Computer Society, 2001. 68−75.

[7] Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R. Extended static checking for Java. ACM SIGPLAN Notices, 2002,37(5):234−245. https://doi.org/10.1145/543552.512558

[8] Engler D, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. ACM SIGOPS Operating Systems Review, 2003,37(5):237−252. https://doi.org/10.1145/1165389.945468

[9] Williams A, Thies W, Ernst MD. Static deadlock detection for Java libraries. In: Proc. of the 19th European Conf. on Object-Oriented Programming. Berlin, Heidelberg: Springer-Verlag, 2005. 602−629. https://doi.org/10.1007/11531142_26

[10] Naik M, Park CS, Sen K, Gay D. Effective static deadlock detection. In: Proc. of the 31st Int'l Conf. on Software Engineering. IEEE, 2009. 386−396. https://doi.org/10.1109/ICSE.2009.5070538

[11] Harrow J. Runtime checking of multithreaded applications with visual threads. In: Proc. of the 7th Int'l SPIN Workshop on SPIN Model Checking and Software Verification. Berlin, Heidelberg: Springer-Verlag, 2000. 331−342. [doi: 10.1007/10722468_20]

[12] Havelund K. Using runtime analysis to guide model checking of Java programs. In: Proc. of the 7th Int'l SPIN Workshop on SPIN Model Checking and Software Verification. Berlin, Heidelberg: Springer-Verlag, 2001. 245−264. [doi: 10.1007/10722468_15]

[13] Agarwal R, Wang LQ, Stoller SD. Detecting potential deadlocks with static analysis and run-time monitoring. In: Proc. of the 1st Haifa Int'l Conf. on Hardware and Software Verification and Testing. Berlin, Heidelberg: Springer-Verlag, 2005. 191−207.[doi: 10.1007/11678779_14]

[14] Joshi P, Park CS, Sen K, Naik M. A randomized dynamic program analysis technique for detecting real deadlocks. ACM SIGPLAN Notices, 2009,44(6):110−120. [doi: 10.1145/1543135.1542489]

[15] Cai Y, Chan WK. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. IEEE Trans. on Software Engineering, 2014,44(3):266−281. https://doi.org/10.1145/512529.512558

[16] Bensalem S, Havelund K. Scalable dynamic deadlock analysis of multi-threaded programs. In: Proc. of the Parallel and Distributed Systems: Testing and Debugging (PADTAD-3), IBM Verification Conf. Springer, Berlin, Heidelberg. 2005. [doi: 10.1007/11678779_15]

[17] Agarwal R, Bensalem S, Farchi E, Havelund K, Nir-Buchbinder Y, Stoller SD, Ur S, Wang LQ. Detection of deadlock potentials in multithreaded programs. IBM Journal of Research and Development, 2010,54(5):3:1−3:15. [doi: 10.1147/JRD.2010.2060276]

[18] Samak M, Ramanathan MK. Trace driven dynamic deadlock detection and reproduction. ACM SIGPLAN Notices, 2014, 49(8): 29−42. https://doi.org/10.1145/2555243.2555262

[19] Cai Y, Zhai K, Wu SR, Chan WK. Teamwork: Synchronizing threads globally to detect real deadlocks for multithreaded programs. ACM SIGPLAN Notices, 2013,48(8):311−312. https://doi.org/10.1145/2442516.2442560

[20] Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs. ACM SIGARCH Computer Architecture News, 2010,38(1):167−178. [doi: 10.1145/1735970.1736040]

[21] Cai Y, Lu Q. Dynamic testing for deadlocks via constraints. IEEE Trans. on Software Engineering, 2016,42(9):825−842. [doi: 10.1109/TSE.2016.2537335]

[22] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2008. 209−224.

[23] Cai Y, Chan WK. MagicFuzzer: Scalable deadlock detection for large-scale applications. In: Proc. of the 34th Int'l Conf. on Software Engineering. IEEE, 2012. 606−616. [doi: 10.1109 / ICSE.2012.6227156]

[24] Cai Y, Jia CJ, Wu SR, Zhai K, Chan WK. ASN: A dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. IEEE Trans. on Parallel and Distributed Systems, 2015,26(1):13−23. [doi: 10.1109/TPDS.2014.2307864]

[25] Cai Y, Wu SR, Chan WK. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In: Proc. of the 36th Int'l Conf. on Software Engineering. ACM, 2014. 491−502. https://doi.org/10.1145/2568225.2568312

[26] Joshi P, Naik M, Park CS, Sen K. CalFuzzer : An extensible active testing framework for concurrent programs. In: Proc. of the 21st Int'l Conf. on Computer Aided Verification. Berlin Heidelberg: Springer-Verlag, 2009. 675−681. https://doi.org/10.1007/978-3-642-02658-4_54

[27] Gao J, Yang X, Jiang Y, Liu H, Ying WL, Sun WT, Gu M. Managing concurrent testing of data race with ComRaDe. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2018. 364−367. https://doi.org/10.1145/3213846.3229502

[28] Yuan YH, Li Y, Wang ZG. A matrix algorithm for enumerating all circuits of a graph. Journal of Northwestern Polytechnical University, 1992,10(2):204−210 (in Chinese with English abstract).

[29] Li B, He YP, Ma HT. Automatic program repair: Key problems and technologies. Ruan Jian Xue Bao/Journal of Software, 2019, 30(2):244−265 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5657.htm [doi: 10.13328/j.cnki.jos.005657]

[30] Wei HM, Gao J, Qing P, Yu K, Fang YF, Li ML. MPI-RCDD: A framework for MPI runtime communication deadlock detection. Journal of Computer Science and Technology, 2020,35(2):395−411. [doi: 10.1007/s11390-020-9701-4]

[31] Lu FM, Tao RR, Du YY, Zeng QT, Bao YX. Deadlock detection-oriented unfolding of unbounded Petri nets. Information Sciences, 2019,497:1−22. https://doi.org/10.1016/j.ins.2019.05.021

[32] Lu FM, Zeng QT, Zhou MC, Bao YX, Duan H. Complex reachability trees and their application to deadlock detection for unbounded Petri nets. IEEE Trans. on Systems, Man and Cybernetics: Systems, 2019,49(6):1164−1174. [doi: 10.1109/TSMC.2017.2692262]

[33] Zhu JQ, Sun HZ, Huang YX, Liu M. Delay satisfied route selection and real-time update scheduling in software defined networking. Ruan Jian Xue Bao/Journal of Software, 2019,30(11):3440−3456 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5655.htm [doi: 10.13328/j.cnki.jos.005655]

附中文参考文献:

[1] 苏小红,禹振,王甜甜,马培军.并发缺陷暴露、检测与规避研究综述.计算机学报,2015,395(11):93−111. [doi: 10.11897/SP.J.1016.2015.02215]

[2] 张健,张超,玄跻峰,熊英飞,王千祥,梁彬,李炼,窦文生,陈振邦,陈立前,蔡彦.程序分析研究进展.软件学报,2019,30(1):80−109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[28] 袁亚华,李泳,王自果.一种求网络所有回路的矩阵方法.西北工业大学学报,1992,10(2):204−210.

[29] 李斌,贺也平,马恒太.程序自动修复:关键问题及技术.软件学报,2019,30(2):244−265. http://www.jos.org.cn/1000-9825/5657.htm [doi: 10.13328/j.cnki.jos.005657]

[33] 朱金奇,孙华志,黄永鑫,刘明.软件定义网络中延迟满足的路由选择与实时调度更新.软件学报,2019,30(11):3440−3456. http://www.jos.org.cn/1000-9825/5655.htm [doi: 10.13328/j.cnki.jos.005655]

鲁法明(1981－),男,博士,副教授,博士生导师,CCF 专业会员,主要研究领域为 Petri 网理论与应用,并发系统建模与分析,业务过程管理与决策支持.

郑佳静(1996－),女,硕士生,CCF 学生会员,主要研究领域为软件形式化方法,并行程序验证.

包云霞(1979－),女,讲师,主要研究领域为形式化方法,Petri 网.

曾庆田(1976－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为 Petri 网理论与应用,并发系统建模与分析,业务过程管理与决策支持.

段华(1976－),女,博士,副教授,主要研究领域为形式化方法,Petri 网.

王晓宇(1999－),男,本科生,主要研究领域为软件形式化方法,并行程序验证.