清 华 大 学

攻读硕士学位研究生选题报告

研 究 生 李 曈 学 号 2016210986

院（系、所） 计算机科学与技术系

学 科 计算机科学与技术

指 导 教 师 陈 渝 职 称 副教授

2019年1月21日

# 1. 选题意义

## 1.1 Preempt-RT Linux实时性补丁

实时操作系统（Real-Time Operating System，RTOS）[8]是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统做出快速响应，调度一切可利用的资源完成实时任务，并控制所有实时任务协调一致运行的操作系统。提供及时响应和高可靠性是其主要特点。

实时系统分为硬实时系统和软实时系统。硬实时系统是指如果没有及时处理任务，可能会造成灾难性后果，比如航天器中的嵌入式系统、核反应堆控制器中的系统等，系统的失误会造成人身和财产损失。软实时系统是指如果没有及时处理任务，后果并不是灾难性的，只是性能、用户体验等有损失，比如视频音频的播放器，若果没有及时解码，会造成视频音频的卡顿，但并不会造成灾难性后果。

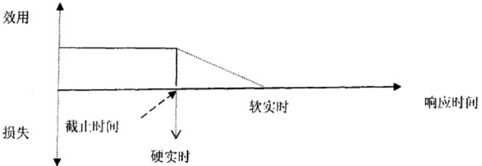


图1.1-1 实时系统效用与响应时间的关系

商业的实时系统很早就出现了，比如著名的VxWorks[9]是1983年设计开发的嵌入式实时系统，广泛用于通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中。

Linux是世界上应用最广泛的开源操作系统，是作为通用操作系统被开发的，其内核在实时处理能力上先天不足。2004年十月，Preempt-RT Linux内核原型在Linux Kernel mailing list中被正式提出。这个原型引入了可抢占锁，在共享资源互斥的前提下，允许低优先级任务在临界区中被高优先级任务抢占，使得Linux的实时性大幅提高。Preempt-RT Linux和商业的实时操作系统相比，虽然实时性能稍差，但是不需要重写各种驱动程序，有更强大的社区支持，所以发展非常迅速。

1.2 并发程序与Linux内核中Bug现状研究

随着Linux内核不断增加新的特性，内核的代码量不断增长，但内核中的Bug数量也随之不断增长，而且Bug存在的平均时长也在不断增长。这说明随着内核的不断发展，Bug的模式更加复杂，查找也更加困难。



图1.2-1 Linux内核中Bug数量和存在时间随版本的变化

多核CPU的发展促进了并发程序设计的发展，但由于程序开发人员的线性编程思维和并发程序的不确定性导致bug难以复现，并发程序缺陷的检测和修复仍需要很多努力。并发程序缺陷中，超过70%的缺陷是违反了操作的原子性[11]。

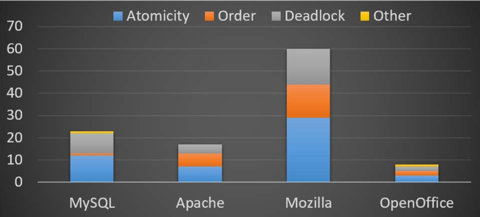


图1.2-2程序中的Concurrency Bug的分类统计

在Linu内核中，内核代码一般来说运行在进程上下文或原子上下文中，进程上下文是内核代表用户进程在进行一些操作，典型的例子是系统调用；原子上下文是指中断上下文和被spinlock保护的临界区[1]。

原子上下文中的代码不能睡眠，原因如下：

（1）一个进程在持有一个spinlock，并在释放它之前进入睡眠，那么第二个进程（或者中断处理程序）如果要获得这个锁的话，就会进入忙等。在单处理器系统中，第二个进程将会锁住CPU，导致第一个进程无法被唤醒，spinlock也就无法释放，形成死锁。

（2）中断处理程序是基于事件执行的，没有自己的task\_struct，被进程抢占后无法恢复运行。

Linux系统本身庞大而复杂，目前对Preempt-RT Linux实时性补丁做的测试比较少，已知的原子上下文中睡眠的bug都是在实际使用过程中出问题才去修复的，没有相关的工具去进行分析和测试。

我们选择了Preempt-RT Linux 4.9.84-rt62版本的补丁文件进行分析，统计结果如图1。统计发现，共351个补丁文件中，有37个补丁是用来修复之前版本的Preempt-RT Linux补丁引入的BUG，这个37个补丁中有21个补丁中的BUG原因是Preempt-RT Linux引入了新型的自旋锁后，原本代码的原子上下文性质被破坏。

图1. Preempt-RT Linux 4.9.84-rt62中BUG的分类统计

基于此，我们希望能设计出基于导向符号执行的Bug检测工具，可以对Preempt-RT Linux的代码进行分析，尽量找出代码中存在的原子上下文被破坏的BUG，提高Preempt-RT Linux的鲁棒性。

# 2. 国内外研究现状

## 2.1 Preempt-RT Linux

Sven-Thorsten Dietrich和Daniel Walker在2005年总结了Linux实时性的发展，以及Ingo Molnar提出的Preempt-RT Linux实时性补丁[4]。介绍了Preempt-RT Linux实时性补丁中新型spinlock的性质和细节，对死锁的检测做了重点的描述，但是没有涉及原子上下文中不能睡眠的问题。

Preempt-RT Linux实时性补丁提出后，关于它的研究多关注于性能和应用。Fayyad-Kazan, Hasan等人在2013年对Preempt-RT Linux 3.6.6-rt17的研究表明[5]，Preempt-RT Linux的实时性能已经接近商业实时系统，但是稳定性仍需要开发者共同努力提高。

2.2 并发程序缺陷检测

常见的并发程序缺陷的检测方法有以下三类[12]：

1. 静态分析

静态分析不执行代码，而是通过在编译过程中进行分析来找到代码中符合某种特征的并发程序缺陷。优势在于原理是通过分析代码来寻找程序缺陷，可以找到实际运行过程中难以抵达的代码，效率相对较高。劣势在于因为静态分析不执行程序，不能对做精准的检测，误报率相对较高。

1. 动态分析

动态分析通过执行被测试的程序，检测程序执行的轨迹，记录所有访问内存和同步互斥操作的历史，通过一些算法检测是否有程序缺陷。优势在于可以精确地记录并发程序缺陷发生时变量和线程执行的情况，可以检测各种不同类型的程序缺陷；劣势在于受限于测试集。

1. 符号执行

符号执行用符号量代替真是值作为程序的输入，根据符号量取值的不同，遍历程序的所有执行路径。优势在于对每个程序状态的取值有具体的分析，误报率很低；劣势在于会产生路径爆炸的问题，难以应用在大规模的软件分析上。

Palix, Nicolas等人在2014年研究了Linux 2.6中的各类bug[2]，并开源了一套比较基础的分析工具，其中包括对原子上下文中睡眠BUG的检测工具——BlockLock。但是分析并没有对代码进行任何处，所以只使用了简单的上下文信息，检测的BUG比较少。

白家驹等人在2018年基于Nicolas的工作[3]，进一步设计了基于数据流分析的工具——DSAC，专门检测原子上下文中睡眠的BUG。DSAC综合使用了流敏感和流不敏感分析，将内核中原子上下文中的代码路径抽取出来，并使用启发式搜索对内核中可能睡眠的内核接口进行搜索。但是DSAC没有处理函数指针的问题，而且启发式搜索这一步在我们的场景中不需要，可以使用更高效的算法解决。

汪黎等人在2007年使用专家系统Agent的方法[6]，对操作系统内核函数执行上下文进行自动检测。这套工具先建立了知识集，然后在知识集中自动推断。方法比较先进，但缺陷并不适合这类问题。

Heming Cui等人在2013年设计了WoodPecker[13]，基于符号执行方法，将和符号量无关的路径进行剪枝，从而大幅提高了符号执行的效率，对比KLEE  
可以提高3～17倍。

Yufeng Zhang等人在2015年设计了动态符号执行[14]，对符号执行分析路径的顺序做一个导向，优先分析存在Bug可能性大的路径，从而提高了找出Bug的效率。

Stefan Bucur等人设计的cloud9[15]在KLEE的基础上增加了对Pthread API的支持，可以对Pthread应用做符号执行。

# 3. 研究内容

静态分析和符号执行在执行速度和误报率上各有优势，需要互相结合，并根据研究对象的特点做一定的优化，才能设计出合适的方案。总体来说，我的研究内容是基于导向符号执行的并发缺陷分析，研究内容具体来说可以细分为以下几部分：

（1）扩展符号执行环境

\* 分析Pthread应用中并发程序缺陷特征并相应的扩展符号执行环境

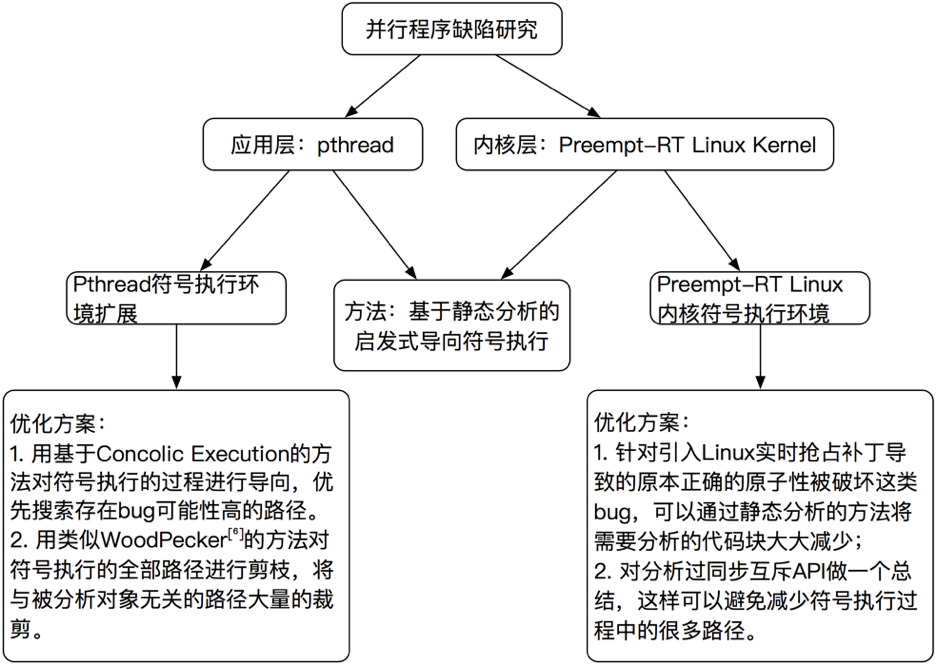
\* 分析RT-Linux中同步互斥原语特征并相应的扩展内核代码符号执行环境

（2）研究结合静态分析的启发式符号执行算法

\* 分析Pthread应用并发程序缺陷特征并研究相应的启发式符号执行算法

\* 分析Preempt-RT Linux内核中同步互斥Bug特征并研究结合静态分析的启发式符号执行算法。

根据目前的调研，符号执行的优化大致有以下几个方向：



1. 动态符号执行(DSE)是指在符号执行的过程中，记录状态机的状态变化的历史信息，并结合静态分析来得到后续状态机可能迁移到终态的路径。最后结合上述信息来指导符号执行的路径选择，让符号执行优先分析存在Bug可能性大的路径。
2. 对符号执行的路径进行剪枝[6]，在符号执行的过程中动态的将与要分析的模式无关的路径剪枝，减少符号执行要分析的路径总数。
3. 我们在Preempt-RT Linux中寻找的Bug是因为引入Preempt-RT Linux后导致原本正确的原子性被破坏的Bug，所以可以用静态分析将与原子操作无关的代码块去掉。
4. 同步互斥的函数大多都比较简短，可以对与简单的函数做一个summary，这样可以进一步减少符号执行过程中需要分析的路径数。

# 附录

[1] Jonathan Corbet. Atomic context and kernel API design. In Linux Weekly News (LWN.net), 2008. https://lwn.net/Articles/274695/.

[2] Palix N, Thomas G, Saha S, et al. Faults in linux 2.6[J]. ACM Transactions on Computer Systems (TOCS), 2014, 32(2): 4.

[3] Jia-Ju Bai, Yu-Ping Wang, et al. DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules[C]. Annual Technical Conference. USENIC, 2018.

[4] Dietrich S T, Walker D. The evolution of real-time linux[C]//7th RTL Workshop. 2005.

[5] Fayyad-Kazan H, Perneel L, Timmerman M. Linux PREEMPT-RT vs. Commercial RTOSs: How Big is the Performance Gap? GSTF Journal on Computing. 2013;3(1):135-142.

[6] 汪黎，杨学军，王戟，罗宇．操作系统内核程序函数执行上下文的自动检验．软件学报，2007, 18(4): 1056 1067

[7] Hind M. Pointer analysis: Haven't we solved this problem yet?[C]//Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, 2001: 54-61.

[8] Wikipedia. Real-time operating system. <https://en.wikipedia.org/wiki/Real-time_operating_system>

[9] <http://blogs.windriver.com/deliman/2008/05/did-you-watch-i.html>

[10] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004: 75.

[11] Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics[J]. ACM SIGOPS Operating Systems Review, 2008, 42(2): 329-339.

[12] Fu H, Wang Z, Chen X, et al. A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques[J]. Software Quality Journal, 2018, 26(3): 855-889.

[13] Cui H, Hu G, Wu J, et al. Verifying systems rules using rule-directed symbolic execution[C]//ACM SIGPLAN Notices. ACM, 2013, 48(4): 329-342.

[14] Zhang Y, Clien Z, Wang J, et al. Regular property guided dynamic symbolic execution[C]//Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, 2015: 643-653.

[15] Bucur S, Ureche V, Zamfir C, et al. Parallel symbolic execution for automated real-world software testing[C]//Proceedings of the sixth conference on Computer systems. ACM, 2011: 183-198.

[16] Guo S, Kusano M, Wang C. Conc-iSE: Incremental symbolic execution of concurrent software[C]//Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on. IEEE, 2016: 531-542.

[17] Wang Z, Wu C, Yuan X, et al. Using Local Clocks to Reproduce Concurrency Bugs[J]. IEEE Transactions on Software Engineering, 2018, 44(11): 1112-1128.

[18] Deligiannis P, Donaldson A F, Rakamaric Z. Fast and precise symbolic analysis of concurrency bugs in device drivers (t)[C]//Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, 2015: 166-177.

[19] Yu T, Zaman T S, Wang C. DESCRY: reproducing system-level concurrency failures[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017: 694-704.