基于 Elastos 线程同步机制的死锁检测技术研究*)

张 捷[1,2] 陈 榕[1]

(同济大学基础软件工程中心 上海 200092)[1] (安徽师范大学 芜湖 241000)[2]

摘 要 Elastos是基于构件的操作系统,构件对象的行为模式决定了内核底层机制的实现。在Elastos中,进程对象、线程对象以及线程的同步对象等都是构件对象。介绍了Elastos的CAR构件技术及与之相应的线程同步机制,在此基础上分析了一类资源死锁产生的可能原因;结合Elastos本身的相关特点,讨论了实现死锁检测技术的各个环节,综合考虑了可靠性、正确性及效率方面的要求,给出了一个可行的死锁检测算法。通过实际验证,此算法可以得到正确的结果。

关键词 Elastos,构件对象,线程同步,死锁检测

Research on Techniques for Deadlock Detection Based on Synchronization Mechanism of Thread in Elastos

ZHANG Jie[1,2] CHEN Rong[1]

(System Software Center, Tongji University, Shanghai 200092, China)[1] (Anhui Normal University, Wuhu 241000, China)[2]

Abstract Elastos is a component-based os. The behavior model of component object determines bottom mechanism in kernel. In Elastos, process object, thread object and synchronized object of thread are all component objects. Introduced the CAR component technique and the corresponding synchronization mechanism of thread in Elastos, analysed the possible reason for causing deadlock; Combining with the characteristic of Elastos, it discussed each tache of implemention for deadlock detection techniques, considered demands of credibility, correctness and efficiency, and put forward a feasible algotithm of deadlock detection. This algotithm can obtain correct results by verification of actual case.

Keywords Elastos, Component object, Thread synchronization, Deadlock detection

1 引言

"和欣"[1](Elastos)操作系统是由科泰世纪公司开发的、拥有自主知识产权的新一代嵌入式操作系统,是国家"863"计划的重要成果。Elastos将微内核模型与构件技术充分结合,形成了称为"灵活内核"[1](Agile Kernel)的体系架构。其内各个功能模块基于构件设计,可以随环境需求灵活变动,适合智能手机等嵌入式应用。

Elastos内核以构件对象为基础提供底层服务。Elastos的线程同步机制则以同步对象为中心,这些同步对象的内部实现多数是锁相关的。在目前的应用开发中,受嵌入式平台本身资源限制,需要特别注意多线程并发执行的场合,处理稍有不当,便可能引发死锁,而死锁的产生总是难以避免的。当应用程序僵死时,要确定是否已经产生死锁并进一步找到死锁的原因,目前"和欣"的开发平台上还没有提供完善的方案。本文分析了Elastos中影响死锁检测技术实现的各个环节,旨在找到一种可行的解决方案。

2 CAR构件平台

构件技术因其本身的灵活性、实时性以及安全性等特点,在嵌入式操作系统开发中受到越来越多的重视。CAR[2]既是一套网络时代的构件化编程规范,也是构件运行支撑平台,其主要特点有:规定构件的框架部分以元数据形式描述,使得构件执行时通过反射(Reflection)机制自动参与组装计算;定义了构件在各种情况下的通信方式,支持构件配置在不同计算容器(进程、Domain、机器)中实现分布式、协同计算等。基于CAR构件平台,Elastos内核的绝大多数功能模块按构件规范建立为构件对象。内核向应用程序提供的大部分系统服务都要通过内核对象的接口来完成,只有少量系统服务为了方面使用,以API函数形式给出。表1列出了一些主要的内核对象及所提供的接口服务。

表1 Elatos中主要内核对象及接口

对象	说明	主要接口	接口中的方法
CProcess	进程	IProcess	Start, Kill, GetThreads,…
CThread	线程	IThread	Start, Suspend, Resume, Join,…
CSharedMemory	共享内存	ISharedMemory	Attach, Detach
CMutex	互斥量	IMutex	Lock, TryLock, Unlock
CModule	代码或数据模块	IModule	GetEntryPoint, GetName,…

3 Elastos中的线程同步机制

3.1 进程对象与线程对象

如上所述,Elastos中进程对象的实现符合CAR构件规范,进程对象必须通过接口访问,进而实施控制。对于每个新实例化的进程,Elastos内核都会产生一个唯一的进程对象与

*)基金项目:国家"863"计划资助项目(2001AA113400),安徽高校省级自然科学研究重点项目(编号:KJ2008A104)。张 捷 硕士研究生,主要研究领域为嵌入式操作系统、系统软件支撑技术;陈 榕 教授,博导,主要研究领域为网络操作系统、嵌入式操作系统、构件技术。

之对应，用来管理该进程。以创建进程为例，调用系统 API 中的 EzCreateProcessObject 函数会创建一个新的进程对象，并将此进程对象与一个程序模块对应。此函数如果调用成功，会返回该进程对象的 IProcess 接口指针。注意，此时只是进程对象被创建，相应的进程并没有运行。通过只有调用 IProcess 接口的 Start 方法才能真正开始运行。这种以构件接口为中心的行为模式在 Elastos 中是很常见的。

线程是程序执行的单位，同样也是构件化对象，对其访问需通过主要接口 IThread。线程对象生命期大于等于它所对应的线程本身。

3.2 线程同步机制

线程相对于进程总是活跃的。近来的多数操作系统模型中线程已经成为最基本的调度单位(Elastos 采取内核级线程调度模型)。Elastos 的线程同步机制是以同步对象为基础实现的，这些同步对象都是 CAR 构件。Elastos 内提供了 4 种内核同步对象：Mutex，Condition，ReaderWriterLock 和 Event，它们工作于内核模式下。另外还有两种工作在用户模式下的 Critical Section 与互锁函数(Interlocked Functions)，前者没有相对应的内核对象，后者干脆不是对象。关于这些同步机制已经有大量的书与文档介绍它们[3-6]，这里不再重复叙述。我们只以 Critical Section 为例来了解 Elastos 中同步机制的使用方法。

Critical Section 即临界区，是一种轻量级同步机制，工作在用户模式下，它最大的特点是并不总是执行向内核模式的控制转换，而这一转换代价昂贵。要获得一个未占用临界区，事实上只需要对相应内存位置做很少的修改，其速度非常快(使用互锁函数，而它是硬件级的)。只有在向系统申请获取一个已占用临界区时，它才会跳转至内核模式(事实上是通过一个 Event 内核对象来阻塞和唤醒请求线程)。这一轻量级特性的缺点在于临界区只能用于同一进程内的线程同步。

API 函数 EzInitializeCriticalSection 调用后，临界区对象即开始存在。第一个线程自 EzEnterCriticalSection 中正确返回后，所有其他调用 EzEnterCriticalSection 的线程都将被阻塞，直到第一个线程调用 EzLeaveCriticalSection 函数为止。最后，当不再需要该临界区时，需要用 EzDeleteCriticalSection 释放。下面是个例子：

```
CRITICAL_SECTION  cs;// 申明一个临界区对象
int i = 0;// 关键代码段
void main() {
    EzInitializeCriticalSection(&cs);// 初始化
    IThread * p1, * p2;// 线程对象接口
    EzCreateThread(ThreadFunc,(void * )NULL,0,&p1);// 创建并执行第一个线程
    EzCreateThread(ThreadFunc,(void * )NULL,0,&p2);// 创建并执行第二个线程
    EzDeleteCriticalSection(&cs);// 释放 cs
    p1->Release();// 释放第一个线程
    p2->Release();// 释放第二个线程
}
void ThreadFunc(void * pArg) {
    EzEnterCriticalSection(&cs);  // 进入临界区
    i++;
    EzLeaveCriticalSection(&cs);  // 离开临界区
}
```

此外，线程还可以使用 EzTryEnterCriticalSection 来尝试进入临界区。若该临界区已被占用，则立即返回，线程不会在该临界区上阻塞(非阻塞同步)。注意，被临界区阻塞的线程只能由该临界区唤醒，一次唤醒一个。这里为了叙述简洁，对线程对象及临界区对象的使用是通过 API 函数的，实际这些 API 函数内部实现仍是对构件对象接口的调用。

3.3 死锁的产生

Coffman 等 1971 年给出的死锁的 4 个必要条件[7]仍然适用于现今的大多数单机上死锁产生的原因。一般这些死锁多与临界资源相关，称之为资源死锁[8,9]。当然也存在其它非资源死锁的情况，在这里我们不准备讨论这个问题。在 Elastos 目前的应用中，死锁的出现多数与资源有关。目前我们在遇到死锁时采用的策略是：能够检测出死锁并找到引发死锁的原因，这对智能手机应用的开发而言是合适的。目前阶段我们不准备在代码中加入预防死锁(即破坏上述四个条件之一)的功能，一来这将导致系统性能的下降，二来可能会引起一些意想不到的问题。

我们改写上面那个例子，使之可以在 Elastos 中引发死锁：

```
CRITICAL_SECTION  cs;
IMutex * pMutex;// 全局 Mutex 对象接口
void main() {
    EzInitializeCriticalSection(&cs);
    EzCreateMutex(&pMutex);// 创建一个 Mutex 内核对象
    IThread * p1, * p2;// 线程对象接口
    EzCreateThread(ThreadFunc1,(void * )NULL,0,&p1);// 创建并执行第一个线程
    EzCreateThread(ThreadFunc2,(void * )NULL,0,&p2);// 创建并执行第二个线程
    EzDeleteCriticalSection(&cs);// 释放 cs
    p1->Release();// 释放第一个线程
    p2->Release();// 释放第二个线程
    pMutex->Release();// 释放 Mutex
}
void ThreadFunc1(void * pArg) {
    EzEnterCriticalSection(&cs);
    EzSleep(1000);// 等待 1000 个时间片
    pMutex->Lock();
    pMutex->Unlock();
    EzLeaveCriticalSection(&cs);
}
void ThreadFunc2(void * pArg) {
    pMutex->Lock();
    EzSleep(1000);// 等待 1000 个时间片
    EzEnterCriticalSection(&cs);
    EzLeaveCriticalSection(&cs);
    pMutex->Unlock();
}
```

使用 Lock 方法，线程可以获取 Mutex 对象，一旦取得其他请求，此对象的线程将被阻塞，直到占用者使用 Unlock 方法交出此 Mutex 对象。

可以看出，这是我们有意营造的两个进程间的资源死锁。

4 一个可行的死锁检测算法

上面的例子是因为在两个线程之间形成环路而造成的死锁，形成环路的原因是代码编写不当。在资源本身的互斥性和不可抢占性不可更改的情况下，环路等待条件是我们最想

要检测出来并修正的。此算法检测是否在请求同步对象的线程与占有同步对象的线程中存在环路。如果有，输出整个占有-请求链并标明在何处形成环。

考虑到对 Critical Section(以下简称 CS)的使用有可能不进入内核模式，我们要求线程在占有(进入)一个 CS 后主动提供必要的信息给内核，这些信息至少应包括：该占有线程的 ID，以及该 CS 内部使用的一个 Event 对象的信息(我们说过 CS 通过 Event 阻塞与唤醒请求线程，在内核中此 Event 的信息就代表了 CS)。信息的采集位置见 3.2 节。

对 Mutex 与 ReaderWriterLock 的使用本身就含有占有、释放及请求的概念，Condition 则相对复杂一些，虽然本身已是内核对象，在它内部仍需要一个 Mutex 提供原子操作。由此我们规定谁创建此 Mutex，便是“占有”了 Condition，谁在等待这个 Condition 成立便是“请求”。这里做概念上的转换是为了处理的一致性。

遗憾的是，我们还没有找到对 Event 适用的概念转换，这是由 Event 本身的使用特性决定的。虽然等待一个事件我们可以算作“请求”，但严格意义上 Event 对象不能被任何一个线程“占有”。Elastos 对 Event 同步对象做了很多有用的扩展，使之能适应更复杂的同步应用需要。后面的改进版本中我们会考虑死锁检测这一块的需求。

4.1 数据结构

现在除了 Event 与互锁函数(本身不是对象)外，内核能够获知其它 4 种同步对象的占有者。系统在执行期间会产生多个同步对象，当然也会产生多个线程。考虑到维护的方便，我们按线程组织所有信息。按同步对象组织所有信息，实现较为困难，因为同步对象本身并不像线程那样拥有标识 ID，即使是命名同步对象在组织成较大结构时也不易维护。

在 Elastos 中，线程的最大数目是有限的，而每个线程的 ID 又是唯一的，所以我们构建了一张标准的 Hash 表。表的容量与线程最大数一致，表的索引值通过对线程 ID 的 Hash 获得。很幸运，“和欣”中线程 ID 的构成法决定这张 Hash 表不存在冲突。每个表项唯一对应一个线程，里面存放占有链表的头部。因为一个线程在某个执行时刻可能会占有不只一个同步对象，每当新占有一个同步对象，都会把此对象的信息结点加入到链表；每释放一个同步对象，则会在链表中删除相关结点。图 1 给出了整个结构的概况。

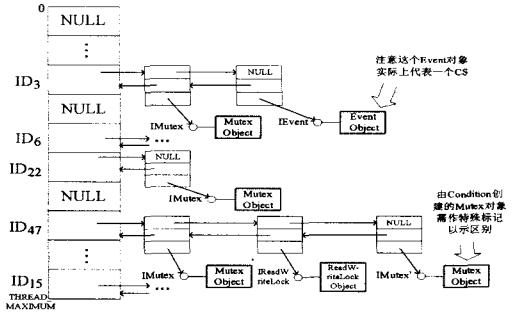

图 1 线程 Hash 表及占有链表

一个重要问题是，这个结构放在哪里合适？首先放在进程空间内是不合适的。我们说过，Elastos 是内核级线程调度，这个结构应当面向系统中当前的所有线程，而非只是当前进程的线程。其次是放在共享内存中。共享内存是进程级的通信机制，理论上只要映射了共享内存到自己的地址空间，所有进程都能访问此结构。问题是我们只是为了检测环路死锁而要更改系统中所有进程的源码，这样做似乎代价太大。

解决此问题的一个思路是，既然所有的线程都要用到，那就把这个结构放到内核空间内，由内核负责分配、释放空间。内核对其内部保留空间的管理是严格而高效的，而且能很好地协调多线程同时访问此结构。一个需要注意的问题是结构本身不宜过大。

4.2 算法

(1)信息采集。每当线程完成一次对同步对象的占有，都会将本线程的 ID 以及该同步对象的接口指针作为结点传给内核。内核 Hash 这个 ID 找到结构中对应表项，将结点插入到该表项的已占有链表中。同样，每当释放一个同步对象，也要在已占有链表中删除相应结点。

(2)查找环路。我们将查找函数放入内核调试程序(Kernel Debugger)内，只有在程序运行出现僵死后进 Debug 才能启动查找，这样做不会影响运行正常的程序。查找算法的形式化描述如下：

①任取一个当前线程，获取其 ID_0 以及它正在等待的同步对象接口指针。若无正在等待同步对象，转⑥；否则转②。

②将进程 ID_0、等待同步对象类型、等待同步对象接口指针三项放入局部数组前三项。

③依据刚得到的等待对象的同步接口指针，检索上述数据结构，得到此对象的占有者线程 ID_1，放入局部数组。

④向前回溯局部数组。若有 ID_2 值等于刚得到的 ID_1，标记 ID_2 位置并调用输出环路函数。正确，返回后转⑥；否则转⑤。

⑤获取 ID_1 线程正在等待的同步对象接口指针，若无正在等待同步对象，转⑥；否则将等待同步对象类型与此接口指针放入局部数组，转③。

⑥重复步骤①，直到已取完当前所有线程。

(3)输出环路。在局部数组中，从前述步骤④标记的位置开始直到数组最后，这就是我们找到的环路。这里找到的环路总是最小的，即在此环路内不可能存在更小的环路。注意，在这里若涉及 Event 对象接口指针，那它一定代表某个 CS。同样，我们需要对由 Condition 创建的 Mutex 做特殊标记，以区别一般的 Mutex 对象。

4.3 测试输出

在安装有 Elastos2.0 版本的 X86 单片机环境下，我们拿 3.3 小节的例子测试，这是最简单的两线程间形成环路的情况。输出结果如下：

threadID: 800107 output: $DeadLock $ threadID: 800107 -> MUTEX ==> threadID:901352 -> CRITICAL_SECTION ==> threadID:800107 $DeadLock$

threadID: 901352 output: $DeadLock $ threadID: 901352 -> CRITICAL_SECTION ==> threadID: 800107 -> MUTEX ==> threadID:901352 $DeadLock $

这里->表示请求，==>表示占有。

我们测试了各种情况，一个典型的输出如下：

threadID:801189 output: threadID: 801189 -> MUTEX ==> threadID:005607 -> CONDITION ==> $DeadLock$ threadID:900010 -> MUTEX ==> threadID:100862 -> READ_WRITE_LOCK ==> threadID: 190047 -> CONDITION ==> threadID:900010 $DeadLock$

可以看到线程 threadID:801189 并不是一开始就形成环路,而是受到后面那些线程的影响。我们输出整个占有-请求链而非只输出死锁环路部分,是为了更好地跟踪线程的行为。

结束语 在信息采集部分,只涉及 Hash 以及链表的插入、删除操作,这个过程的时间复杂度是 $O(l)$。而在查找环路部分,因为要对结构进行检索(4.2 节中步骤③),这个过程的时间复杂度是 $O(n^2)$。所幸系统中某时刻的用户线程数总是受限的,而结构中这些线程的平均链表长度也不可能过长,所以算法执行时间不会太长,得到的结果也比较令人满意。我们说过,此算法只在调试状态下执行。如果想要依托 4.1 节的结构对正常运行中的程序进行实时监控,结构的检索算法部分还要进行改进,以期在更少的时间片内完成一次检索,才有可能得到真实可信的结果。还有一个遗留问题是对 Event 对象的处理,可以尝试的一个思路是依据 Event 构件对象本身的自描述性质,在程序的语义分析阶段解决这一问题,那将是一个有趣的课题。

参 考 文 献

[1] Koretide. Elastos 2.0 Operating System Manual[EB/OL]. 2007. http://www.koretide.com.cn

[2] Koretide. CAR's Manual[EB/OL]. 2007. http://www.koretide.com.cn

[3] Tanenbaum A S, Woodhull A S. Operating Systems Design and Implementation Third Edition[M]. Beijing: Publishing House of Electronics Industry, 2006: 48-60

[4] Python Documentation, 15.3.3 Condition Objects[EB/OL]. 2006. http://docs.python.org/lib/condition-objects.html

[5] Richter J. Programming Applications for Microsoft Windows Fourth Edition[M]. Beijing: China Machine Press, 2005

[6] Microsoft MSDN Library, .NET Development 类库, System.Threading[EB/OL]. 2007. http://msdn2.microsoft.com/zh-cn/library/system.threading.aspx

[7] Coffman E G, Elphick M J, Shoshani A. System Deadlocks. Computing Surveys, 1971, 3: 67-78

[8] Levine G N. Defining Deadlocks. Operating Systems Review, 2003, 37: 54-64

[9] Bacon J, Harris T. Operating Systems Concurrent and Distributed Software Design[M]. Beijing: Publishing House of Electronics Industry, 2003: 297-316

(上接第 233 页)

编译、错误注入和动态监测的自动化工具的总体架构。所实现的测试工具原型 CSTS 主要具备如下优点:

(1)强大的环境错误注入功能。从内存、文件、注册表、进程和网络 5 个方面对组件依赖的环境进行模拟错误注入,最大可能地触发组件安全异常。

(2)全面的动态监测功能。对组件运行中的组件加载、方法调用、参数、返回值、返回地址、异常信息及异常地址等信息进行全面的监测和记录,特别是对异常信息,CSTS 将对其进行单独显示和分析。

(3)工具的自动化程度高,仅需要少量的人工参与。

(4) 测试工程以项目形式管理,有利于组件测试的组织和管理。通过测试项目的有效管理能达到部分测试资源(如用例)的重用,同时可再现历史测试活动,有利于测试结果的对比和分析。

(5) 测试环境可视化程度高。提供了直观的接口信息树状图显示、测试用例编写、方便的错误注入面板和监测信息列表等界面。测试人员在集成测试环境下根据提示便可轻易完成整个测试流程。

当前,我们也只是在组件安全测试自动化方面作了一些有益的探索,所实现的原型工具尚有一些不足之处:

(1) 对一些没有提供或集成 TLB 类型库的 COM 组件无法分析得到接口信息。

(2) 对一些如指针的指针、用户自定义类型等复杂方法参数类型,工具无法自动生成初始的测试用例代码,需要测试人员手工编写测试用例程序。

(3) 工具没有提供完整的组件安全等级评估机制。

参 考 文 献

[1] 毛澄映,卢炎生. 构件软件测试技术研究进展. 计算机研究与发展,2006, 43:1375-1382

[2] 毛澄映,卢炎生,谢晓东,等. 一个 C/C++程序集成测试平台的设计与实现. 小型微型计算机系统,2007,28:1037-1043

[3] Chen Jin-fu, Lu Yan-sheng, Xie Xiao-dong. Testing Approach of Component Security Based on Fault Injection // 2007 International Conference on Computational Intelligence and Security (CIS'2007). IEEE Computer Society, Harbin, China, 2007: 763-767

[4] Chen Jin-fu, Lu Yan-sheng, Xie Xiao-dong, et al. Testing Approach of Component Security Based on Dynamic Monitoring // Second International Multi-Symposiums on Computer and Computational Sciences(IMSCCS 2007). IEEE Computer Society, Iowa City, IA, USA, 2007: 381-386

[5] Hsueh M-C, Tsai T K, Lyer R K. Fault Injection Techniques and Tools. IEEE Compute, 1997, 30: 75-82

[6] Jeffrey V. Fault Injection for the Masses. IEEE Computer, 1997, 30: 129-130

[7] Du Wenliang, Mathur A P. Vulnerability Testing of Software System Using Fault Injection. Coast TR 98-02. 1998: 1-20

[8] Voas J, Mcgraw G. Software Fault Injection: Inoculating Programs Against Errors. John Wiley and Sons, 1997

[9] Looker N, Munro M, Xu J. A Comparison of Network Level Fault Injection with Code Insertion // the 29th IEEE International Computer Software and Applications Conference. Scotland, 2005: 479-484

[10] Whittaker J A, Mottay F E, Ei-Far L K. Testing Exception and Error Cases Using Runtime Fault Injection. Florida Tech Computer Science Technical Report. 2001

[11] Thompson H H, Whittaker J A, Mottay F E. Software security vulnerability testing in hostile environments // the 2002 ACM Symposium on Applied Computing. Madrid, Spain, 2002: 260-264

[12] Whittaker James A. Software's Invisible Users. IEEE software, 2001, 18(6): 84-88

[13] Cowan C, Pu C, Maier D, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks // the Proceedings of the 7th USENIX Security Symposium. San Antonio, Texas, 1998: 63-78