64位Windows ABI虚拟化方法研究

黄聪会[1] 陈 靖[1] 龚水清[1] 陈明华[2]
(空军工程大学信息与导航学院 西安 710077)[1] (中国人民解放军94669部队 芜湖 241007)[2]

摘 要 针对64位Windows ABI在Linux上的虚拟化问题,对x86-64 ABI在Windows和Linux中的差异进行了分析,提出并研究了实现64位Windows ABI虚拟化的3个关键问题,即程序加载与链接、程序库接口仿真和系统调用仿真。在此基础上,对在用户空间和内核空间实现64位Windows ABI虚拟化的两种解决方案进行了分析,并基于用户空间方案设计实现了一种兼容Win64应用程序的操作系统KgdLinux。实验测试结果表明,64位Windows ABI虚拟化方法是可行的。

关键词 二进制兼容,虚拟化技术,应用二进制接口,系统调用
中图法分类号 TP311 **文献标识码** A

Research of Method for Virtualizing 64-bit Windows Application Binary Interface

HUANG Cong-hui[1] CHEN Jing[1] GONG Shui-qing[1] CHEN Ming-hua[2]
(Information and Navigation Institute, Air Force Engineering University, Xi'an 710077, China)[1]
(PLA 94669 Army, Wuhu 241007, China)[2]

Abstract Aiming at the problem of virtualizing 64-bit Windows ABI on Linux platform, the differences in Windows and Linux x86-64 ABI were analyzed, and then the three key issues were proposed and studied to achieve 64-bit Windows ABI Virtualization, which are how to load and link the Windows program, emulate the library interface and the system call. On this basis, the two solutions to achieve 64-bit Windows ABI virtualization on the user space and kernel space of Linux platform were analyzed, and then an operating system named KgdLinux was implemented, which is compatible with Win64 applications based on the solution of Linux user space. The result of experimental tests shows the method for virtualizing 64-bit Windows ABI is feasible.

Keywords Binary compatibility, Virtualization technology, Application binary interface, System call

随着x86-64架构微处理器的广泛使用,32位计算向64位计算过渡已成发展趋势。64位计算拥有更快的处理速度,支持更大内存,但其优势的发挥需要64位操作系统和应用程序的配合。目前64位应用程序十分匮乏,尤其在64位Linux桌面操作系统上。为增强64位Linux的吸引力,丰富其应用程序资源,可采用系统虚拟化[1]或进程虚拟化[2]技术使64位Windows应用程序运行于Linux。

系统虚拟化技术是指提供完整的系统环境,支持Linux平台安装运行Windows操作系统及其64位应用程序的技术。系统虚拟化技术能兼容所有64位Windows应用程序,但存在资源消耗大、性能偏低、需安装Windows操作系统等缺点。进程虚拟机技术则直接支持64位Windows应用程序在Linux上运行,具有较高的效率。进程虚拟化技术可采用高级语言虚拟机[3]方法和程序库层虚拟化方法[4]实现。高级语言虚拟机方法仅支持特定语言编写的程序跨平台运行,具有一定的局限性。程序库层虚拟化方法通过虚拟64位Windows ABI,理论上可二进制兼容所有64位Windows应用程序,因此具有广泛的应用前景,但实现难度很大,目前相关研究很少。

针对64位Windows ABI的虚拟化问题,本文在分析64位Windows ABI的基础上,对虚拟64位Windows ABI的关键问题和实现方案进行研究,最后给出原型系统实现及测试结果。测试结果表明,64位Windows ABI虚拟化方法是可行的。

1 64位WindowsABI分析

计算机科学中,应用程序二进制接口(Application Binary Interface,ABI)描述了应用程序(或者其他类型)和操作系统之间或其他应用程序的低级接口[5]。ABI通常由目标文件格式、程序库接口、系统调用编码与调用方法、数据类型、大小和对齐、调用约定等细节构成。一个完整的ABI允许支持其操作系统上的程序不经修改便可在其他支持此ABI的操作系统上运行。

1.1 目标文件格式

目标文件格式是操作系统ABI的核心。围绕目标文件

到稿日期:2013-03-25 返修日期:2013-06-07 本文受国家自然科学基金(61172083)资助。
黄聪会(1985-),男,博士生,主要研究方向为虚拟化技术,E-mail:huangdoubleten@126.com;陈 靖(1964-),女,博士,教授,主要研究方向为虚拟化技术、无线自组织网络;龚水清(1987-),男,博士生,主要研究方向为虚拟化技术;陈明华(1982-),男,助理工程师,主要研究方向为计算机网络。

格式，如何加载二进制映像、动态链接、系统调用、函数调用等规定和要求共同构成了操作系统 ABI。Win64 操作系统上的目标文件格式为 PE32＋，而 Win32 操作系统为 PE32[6]。PE32＋与 PE32 遵循几乎同样的数据结构，其格式如图 1 所示，仅有的不同存在于可选映像头（IMAGE_OPTIONAL_HEADER64）和 TLS 数据目录（IMAGE_TLS_DIRECTORY64）中。

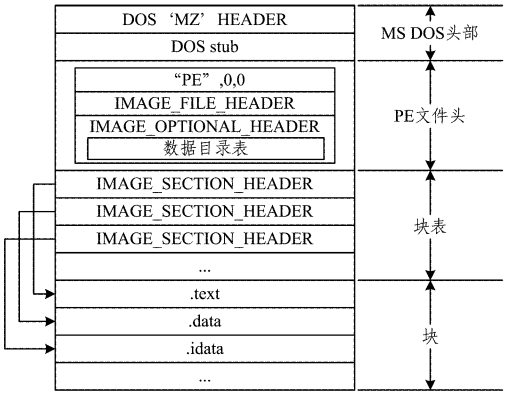

图 1　PE 文件格式

PE32＋中可选映像头的数据结构如图 2 所示。PE32＋中可选映像头与 PE32 的区别在于字段 ImageBase、SizeOfStackReserve、SizeOfStackCommit、SizeOfHeapReserve、SizeOfHeapCommit 的数据类型为 ULONGLONG，占 8 字节空间。而在 PE32 中这些字段数据类型为 DWORD，占 4 字节空间。此外，PE32 中存在数据类型为 DWORD 的字段 BaseOfData，而 PE32＋中不存在。

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
  WORD Magic;/* 0x20b */
  BYTE MajorLinkerVersion;
  …
  DWORD AddressOfEntryPoint;
  DWORD BaseOfCode;
  ULONGLONG ImageBase;
  DWORD SectionAlignment;
  …
  WORD DllCharacteristics;
  ULONGLONG SizeOfStackReserve;
  ULONGLONG SizeOfStackCommit;
  ULONGLONG SizeOfHeapReserve;
  ULONGLONG SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBER-
  OF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, * PIMAGE_OPTIONAL_
HEADER64;
```

图 2　PE32＋可选映像头的数据结构

PE32＋中 TLS 数据目录的数据结构如图 3 所示。PE32＋中 TLS 数据目录与 PE32 的区别在于字段 StartAddressOfRawData、EndAddressOfRawData、AddressOfIndex 和 AddressOfCallBacks 的数据类型为 ULONGLONG，占 8 字节空间。而在 PE32 中这些字段数据类型为 DWORD，占 4 字节空间。

```
typedef struct _IMAGE_TLS_DIRECTORY64 {
  ULONGLONG StartAddressOfRawData;
  ULONGLONG EndAddressOfRawData;
  ULONGLONG AddressOfIndex;
  ULONGLONG AddressOfCallBacks;
  DWORD SizeOfZeroFill;
  DWORD Characteristics;
} IMAGE_TLS_DIRECTORY64, * PIMAGE_TLS_DIRECTO-
RY64;
```

图 3　PE32＋中 TLS 数据目录的数据结构

1.2　程序库接口

程序库接口是对操作系统底层系统调用服务的包装和抽象，以简化应用程序的开发过程，提高开发效率，并保障应用程序的向后兼容性。在 Windows 操作系统中，程序库接口更是隐藏操作系统底层实现细节的重要手段。除了指针的长度和其他与 64 位地址空间相关的数据类型外，Win64 程序库接口与 Win32 程序库接口基本相同。它们之间的细微差异与 Win64 采用的 LLP64 数据模型密切相关。

Win64 采用 LLP64 数据模型，而 Win32 采用 ILP32 数据模型，64 位 Linux 则采用 LP64 数据模型[7]。它们之间的比较如表 1 所列。LLP64 与 ILP32 相比，LLP64 除增加了一种 64 位的 long long 数据类型，指针数据类型增加到 64 位外，两者并无区别。这也是 Win64 程序库接口与 Win32 程序接口能保持一致的重要原因。LP64 与 LLP64 相比，它们的差异在于：LP64 的 long 数据类型增加到 64 位，而 LLP64 的 long 数据类型仍为 32 位。此外，LLP64 中存在 64 位 long long 数据类型，而 LP64 中不存在该数据类型。

表 1　数据模型比较

DataType	LLP64	ILP32	LP64
char	8	8	8
short	16	16	16
int	32	32	32
long	32	32	64
long long	64	—	—
point	64	32	64

1.3　函数调用约定

当函数被调用时，调用函数将参数传递给被调用的函数，返回值则返回给调用函数。函数调用约定即是对函数调用过程中参数的传递顺序、参数的传递方式、寄存器分配方式、栈清理责任的规定。目前在 x86-64 平台上，存在微软 x64 和 System V AMD64 ABI 两种调用约定[8]，如表 2 所列。从表 2 可知，两种函数调用约定的主要区别在于参数传递的寄存器分配。微软 x64 调用约定中，寄存器 RCX、RDX、R8 和 R9 用于整形或指针参数的传递，寄存器 XMM0、XMM1、XMM2、XMM3 则用于浮点型参数的传递，额外的参数通过栈传递。整形返回值通过寄存器 RAX 返回，浮点返回值则通过 XMM0 返回。而在 System V AMD64 ABI 调用约定中，前 6 个整形或指针型参数通过寄存器 RDI、RSI、RDX、RCX、R8 和 R9 传递，浮点型参数则由 XMM0-7 传递。额外的参数同样通过栈传递，返回值则存储在寄存器 RAX 中。

表 2　x86-64 调用约定比较

x86-64 平台	支持的操作系统或编译器	参数传递的寄存器分配	栈中参数传递顺序	由谁清理栈
微软 x64 调用约定	Windows（微软 Visual C++，Inter C++编译器）	RCX/XMM0，RDX/XMM1，R8/XMM2，R9/XMM3	RTL（C）	调用函数
System V AMD64 ABI	Linux，BSD，Mac OS X（GCC，Inter C++编译器）	RDI，RSI，RDX，RCX，R8，R9，XMM0-7	RTL（C）	调用函数

2　ABI 虚拟化关键问题研究

x86-64 ABI 在 Windows 和 Linux 操作系统间存在很大差异，因此 64 位 Windows PE32+应用程序无法在 Linux 上运行。为二进制兼容 PE32+应用程序，应解决 Linux 平台上 Win64 ABI 的虚拟化问题。具体而言，应重点解决 PE32+程序的加载和链接、程序库接口仿真和系统调用仿真 3 个关键问题。

2.1　程序加载与链接

实现 PE32+应用程序的加载与动态链接库的链接，是 Linux 二进制兼容 PE32+应用程序的第一步。PE32+程序加载与链接功能主要分成两部分。一部分实现 PE32+可执行程序的识别，代码段、数据段到内存的映射，解释器的载入等。另一部分实现动态链接库（DLL）的装载与卸载、函数地址的链接、地址的重定位等。PE32+程序加载到内存中并动态链接所需 DLL 后，执行流程将转向 PE32+程序的执行入口。

根据 PE32+程序加载和链接的功能需求，其模块可划分为加载模块和解释器模块。根据模块所处位置的不同，实现方法有如下两种。一是将 PE32+加载模块放入 Linux 内核空间中实现，解释器模块则在用户空间实现。通常 Linux 操作系统中 ELF 目标文件的加载和共享文件的链接即采用此方案。二是完全在用户空间实现加载模块和解释器模块。前者可简化 PE32+程序加载流程，使之具有较快的启动速度，但可移植性弱。后者在用户空间模拟 PE32+程序的加载过程，效率较低，但可移植性强。

2.2　程序库接口仿真

由于 Windows 操作系统的闭源性，微软并没有提供 Windows 程序库接口的实现细节。为仿真 Windows 程序库接口，需使用软件逆向工程[9]技术对其实现细节进行分析，然后在 Linux 上实现，最后应对仿真的程序库接口进行一致性测试[10]，以确保仿真的程序库接口与 Windows 提供的程序库接口功能和行为保持一致。

根据 Win64 程序库接口与 Linux 程序库接口的相似度，可将其分为 3 类：当接口极为相似时，可直接引用 Linux 程序库接口实现；当接口较为相似时，可借鉴使用 Linux 程序库接口的部分源代码；当接口完全不同时，可借助 Linux 底层系统调用实现。

Linux 中编译 Win64 程序库接口仿真代码的编译器必须支持微软的 x64 函数调用约定，否则 Win64 应用程序无法链接动态链接库中的接口函数。

2.3　系统调用仿真

Windows 系统调用的仿真是 ABI 虚拟化的关键部分。几乎所有程序都将通过系统调用获得系统资源的使用权。通常系统调用仿真可采取两种不同的方法：一是在 Linux 内核空间实现 Windows 系统调用；二是在 Linux 用户空间将 Windows 系统调用嫁接到 Linux 系统调用上。两种不同的方法将影响核心动态链接库的设计方案。

在 Windows 操作系统中，只有核心动态链接库 ntdll. dll、kernel32. dll、user32. dll 和 gdi32. dll 涉及系统调用[11]。当在 Linux 内核空间实现 Windows 系统调用时，核心动态链接库可直接进行系统调用。而当在 Linux 用户空间实现 Windows 系统调用时，则必须将核心动态链接库中的系统调用重定向到 Linux 系统调用。

3　原型系统实现与测试

根据对 Win64 ABI 虚拟化关键问题的研究，可产生两种解决方案。一是在内核空间利用 Linux 内核接口和相关源代码实现一个完整的 Win64 ABI。二是在用户空间利用 Linux 程序库接口虚拟一个有限的 Win64 ABI。前者能完全兼容 Win64 应用程序、驱动程序，但实现难度大，工程量大，可移植性差，代码稳定性差。后者能兼容绝大部分 Win64 应用程序，可移植性强，代码稳定性高，实现相对容易，但不兼容 Win64 驱动程序。相较而言，第二种方案更具有优势。因此，基于第二种方案，本文提出了一种采用层次体系结构的兼容 Win64 应用程序的操作系统 KgdLinux，如图 4 所示。

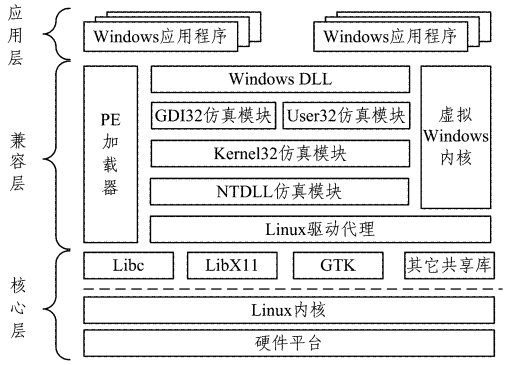

图 4　KgdLinux 操作系统体系结构

KgdLinux 分别由应用层、兼容层和核心层组成。其中应用层主要由 Windows 应用程序以及 Linux 应用程序构成。核心层则由 Linux 内核、Libc、LibX11、GTK 等各种共享库模块组成。兼容层主要由 PE 加载器、虚拟 Windows 内核、核心 DLL 仿真模块和 Linux 驱动代理组成。兼容层是 KgdLinux 实现兼容 Win64 应用程序的关键部分，其中 PE 加载器负责 PE32+可执行程序的加载和链接，虚拟 Windows 内核负责 Windows 系统调用的仿真，核心动态链接库负责程序库接口仿真以及重定向 Windows 系统调用到虚拟 Windows 内核。

在开源软件 Wine 和 64 位 ubuntu 10. 04 基础上，实现了支持 Win64 应用程序运行的 64 位 KgdLinux 操作系统。经测试，少量 Win64 原生程序已能在 64 位 KgdLinux 上运行，如表 3 所列。

表 3　Win64 应用程序在 KgdLinux 中运行状态

Win64 应用程序	运行状态
notepad. exe	良好
MineSweeper. exe	良好
wordpad. exe	良好
calc. exe	良好

结束语　本文首先分析了 x86-64 ABI 在 Windows 和

Linux 中的差异，然后提出并研究了实现 64 位 Windows ABI 虚拟化的关键问题，分析了在 Linux 用户空间和内核空间虚拟 64 位 Windows ABI 的优劣，最后基于 Linux 用户空间虚拟 64 位 Windows ABI 的解决方案，在开源软件 Wine 和 64 位 Ubuntu 10.04 基础上，实现了一种兼容 64 位 Windows 应用程序的操作系统 KgdLinux。实验测试表明，64 位 KgdLinux 已能支持部分 Win64 应用程序的运行，由此证明 64 位 Windows ABI 虚拟化方法是可行的。

参 考 文 献

[1] Vallee G, Naughton T, Engelmann C. System-Level Virtualization for High Performance Computing[C]//Proceedings of 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing. Washington: IEEE Computer Society, 2008: 636-643

[2] Hazelwood K. Process-Level Virtualization for Runtime Adaptation of Embedded Software[C]//Proceedings of the 48th Design Automation Conference. New York: ACM, 2011: 895-900

[3] Smith E, Nair R. 虚拟机：系统与进程的通用平台[M]. 安虹，张昱，吴俊敏，译. 北京：机械工业出版社，2009：8-14

[4] Susanta N, Tzi-Cker C. A Survey on Virtualization Technologies [R]. USA: ECSL, 2005

[5] 黄聪会，陈靖，罗樵，等. 面向二进制移植的虚拟化技术[J]. 计算机应用研究，2012，29(11)：4185-4188

[6] Microsoft. Microsoft PE and COFF Specification[EB/OL]. http:// msdn. microsoft. com/library/windows/hardware/gg463119. aspx, 2013-02-06

[7] 宿继奎，吴亚栋，吕必俊. 32 位到 64 位的移植[J]. 计算机应用与软件，2007，24(3)：174-176

[8] Wiki pedia. x86 calling conventions[EB/OL]. http://en. wikipedia. org/wiki/X86_calling_ conventions, 2011-03-20

[9] Bryant A R, Mills R F, Peterson G L. Software Reverse Engineering as a Sensemaking Task[J]. Journal of Information Assurance and Security, 2012, 6(6): 483-494

[10] Jing Yi-ming, Ahn G-J, Hu Hong-xin. Model-based Conformance Testing for Android[C]//Proceedings of the 7th International Workshop on Security. Japan: Springer, 2012: 1-18

[11] Russinovich M E, Solomon D A. Microsoft Windows Internals (the fifth edition)[M]. USA, Microsoft Press, 2009

(上接第 30 页)

[30] Bartizal D. Thomas Northfield. Solid State Drive Performance White Paper[EB/OL]. http://www. csee. umbc. edu/～squire/images/ssd2. pdf, 2008-3-24/2012-6-7

[31] Benefits of SSD vs. HDD [EB/OL]. http://www. amplicon. com/docs/white-papers/SSD-vs-HDD-white-paper. pdf, 2012-3-21/2012-7-8

[32] Solid State Drive vs. Hard Disk Drive Price and Performance Study [EB/OL]. http://www. dell. com/downloads/global/products/pvaul/en/ssd_vs_hdd_price_and_performance_study. pdf, 2011-5-1/2012-8-19

[33] Flash Memory Technology in Enterprise Storage Flexible Choices to Optimize Performance [EB/ OL]. http://www. itdialogue. com/wp-content/ uploads/2010/04/Flash-in-Enterprise-Storage. pdf, 2008-11-1/2012-3-4

[34] Debnath B, Sengupta S, et al. Chunkstash: speeding up inlinestorage deduplication using flash memory[A]//Proceedings of the 2010 USENIX Annual Technical Conference[C]. Boston: USENIX, 2010: 16-16

[35] The Art of Data Deduplication[OL]. http:// www. ecsl. cs. sunysb. edu/tr/rpe21. pdf

[36] Dubnicki C, Gryz L, et al. HYDRAstor: a scalable secondary storage[A]//Proceedings of the 7th USENIX Conference on File and Storage Echnologies [C]. Berkeley: USENIX, 2009: 197-210

[37] IBM System Storage N series Software Guide [OL]. http://www. redbooks. ibm. com/abstracts/sg247129. html, December 2010

[38] Alvarez C. NetApp deduplication for FAS and V-Series deployment and implementation guide[R]. Technical Report TR-3505. NetApp, 2011

[39] EMC. Achieving storage efficiency through EMC Celerra data deduplication[M]. White paper, Mar. 2010

[40] IBM Corporation. IBM white paper: IBM Storage Tank-A distributed storage system[M]. Jan. 2002

[41] Kulkarni P, Douglis F, et al. Redundancy elimination within large collections of files[A]//Proceedings of the 2004 USENIX Annual Technical Conference[C]. Boston: USENIX, 2004: 59-72

[42] You L L, Pollack K T, et al. Deep Store: An archival storage system architecture[A]//Proceedings of the 21st International Conference on Data Engineering[C]. Los Alamitos: IEEE, 2005: 804-815

[43] Jain N, Dahlin M, et al. TAPER: Tiered approach for eliminating redundancy in replica synchronization[A]//Proceedings of the 5th USENIX Conference on File and Storage Technologies [C]. Berkeley: USENIX, 2005: 281-294

[44] Rhea S, Cox R, et al. Fast, inexpensive content-addressed storage in foundation[A]//Proceedings of the 2008 USENIX AnnualTechnical Conference[C]. Berkeley: USENIX, 2008: 143-156

[45] Meister D, Brinkmann A. dedupv1: Improving deduplication throughput using solid state drives (SSD)[A]//Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies[C]. Piscataway: IEEE, 2010: 1-6

[46] Dong W, Douglis F, et al. Tradeoffs in scalable data routing for deduplication clusters[A]//Proceedings of the Ninth USENIX Conference on File and Storage Technologies [C]. Berkeley: USENIX, 2011: 15-29

[47] Xia W, Jiang H, et al. Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput[A]//Proceedings of the 2011 USENIX Annual Technical Con- ference[C]. Berkeley: USENIX, 2011: 26-28

[48] Zfs deduplication [EB/OL]. https://blogs. oracle. com/bonwick/entry/zfs_dedup, 2009-11-01 /2011. 11. 05

[49] Data striping[EB/OL]. http://en. wikipedia. org /wiki/Data_striping, 2012-08-15/2012-08-23

[50] Reed-Solomon Codes [EB/OL]. http://hscc. cs. nthu. edu. tw/～sheujp/lecture_note/rs. pdf