Linux设备驱动重用研究

王 欢[1] 茅俊杰[2] 王 丹[1] 陈 渝[2]
(北京工业大学计算机学院 北京 100124)[1] (清华大学计算机科学与技术系 北京 100084)[2]

摘 要 设备驱动是影响操作系统适用性的重要因素。考虑到完全重新开发设备驱动代价过大,重用已有操作系统中的设备驱动便成为了提高操作系统适用性的首选方法。设备驱动的重用过程本质上是在目标环境中建立设备驱动的运行环境的过程,重用一个设备驱动并不需要实现所有内核服务。代码依赖分析可以分析驱动代码对内核服务的依赖关系,因此可以使用代码依赖分析技术自动构建设备驱动运行环境。通过在嵌入式操作系统ucore OS中重用e1000网卡驱动来证明方法的可行性。

关键词 Linux,设备驱动重用,代码依赖分析,设备驱动运行环境
中图法分类号 TP311,TP316 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.04.009

Research on Linux Device Driver Reuse

WANG Huan[1] MAO Jun-jie[2] WANG Dan[1] CHEN Yu[2]
(College of Computer Science, Beijing University of Technology, Beijing 100124, China)[1]
(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)[2]

Abstract Device drivers is an important factor affecting the applicability of the operating system. Considering the large cost of completely developing a device driver, reusing existing device drivers in the operating system became the preferred method for improving the applicability of operating system. Reuse of device driver process is essentially a target environment set up process of device drivers, reusing a device driver does not need to implement all of the kernel services. Code dependency analysis could recognize the dependency between device driver and kernel service. Device driver environment can be built by code dependency analysis technology automatically. Through reusing the e1000 network adapter driver, the feasibility of this method was proved.

Keywords Linux, Device driver reuse, Code dependency analysis, Device driver environment

1 引言

设备驱动是影响操作系统适用性的重要因素。许多在特定平台上具有一定研究意义或实用价值的操作系统因缺少足够的设备驱动支持,无法广泛传播和应用。考虑到完全重新开发设备驱动代价过大,重用已有操作系统中的设备驱动便成为了提高操作系统适用性的首选方法。

重用设备驱动可以使开发者不用了解驱动代码中的内部逻辑,只需要使目标操作系统能够支持重用的设备驱动正常运行即可。除此之外,已有操作系统(如Linux等)中的代码可靠性更高,出现问题的概率较低。因此,设备驱动重用能够有效减少操作系统开发过程中的工作量和时间投入,且相比于设备驱动的重新开发,对开发人员的要求更低,代码质量也更高。

Linux是一套免费使用和自由传播的类Unix操作系统,自1991年第一次正式向外公布以来已有20多年的发展历史,经历了数百次的版本更新变化,目前其整体代码的数量已达千万级别,并且无论是在商业机构还是在研究教育机构中都得到了广泛的应用,因此其具备相当多的设备驱动程序,是设备驱动重用工作的首要驱动来源。但由于Linux本身代码数量庞大且结构复杂,重用Linux设备驱动并不简单。

本文以重用Linux设备驱动为研究方向,提出了一种结合代码依赖分析构造设备驱动运行环境(Device Driver Environment,DDE)的Linux设备驱动重用方法,并通过实际重用Linux的网卡驱动来证明该方法的可行性。

2 相关工作

2.1 Linux代码重用

1997年,美国犹他大学Flux研究小组开发了一套设计操作系统的工具包:OSKit[4]。OSKit最初的设计意图是为操作系统的开发者们提供一套可以重用的操作系统模块,而并不是设计一个操作系统。OSKit通过OS Environment重用

到稿日期:2015-11-30 返修日期:2016-03-01

王 欢(1991—),男,硕士,主要研究方向为软件分析与理论、操作系统,E-mail:mikewang.exe@gmail.com;茅俊杰(1989—),男,博士,主要研究方向为操作系统、软件分析;王 丹(1969—),女,教授,主要研究方向为操作系统、可信计算;陈 渝(1972—),男,副教授,主要研究方向为操作系统、计算机信息安全。

了 Linux 和 FreeBSD 中的部分设备驱动。OS Environment 是一个驱动程序的设计规范，通过这个规范，OSKit 就可以将其他操作系统中的驱动程序以源码的方式应用到新设计的操作系统或其他需要驱动程序的应用中。

为微内核 L4 设计的 DDEKit 和操作系统框架 Genode[6] 为了重用 Linux 设备驱动，在各自的系统框架中以与 Linux 内核一致的形式实现了 Linux 设备驱动所需要的内核服务，形成了一个可以运行 Linux 设备驱动的运行环境。

2002 年，K42 团队成功将 Linux 内核中的多个驱动程序、完整的 TCP/IP 协议栈和文件系统重用到了 K42 中[1]。他们的重用方法主要有 4 种：1)将 K42 作为 Linux 内核支持的一种体系结构；2)允许 K42 调用 Linux 代码；3)使用 K42 的代码实现 Linux 内部接口；4)选择性地编译 Linux 内核，使其只包含必要的组件。

2004 年，Joshua LeVasseur 等人通过在自己的系统中建立 Linux 虚拟机来重用 Linux 中的设备驱动[3]。这个方法可以有效地解决驱动程序的运行环境问题，但是需要通过虚拟机优化和对 Linux 内核进行裁剪才能够保证驱动程序的高效运行。

2007 年，Bernhard Poess 提出了一种能够重用二进制 Linux 驱动程序的方法[2]，他通过虚拟机使二进制驱动程序运行在目标操作系统的用户态，然后在目标操作系统中构建二进制驱动程序所需要的符号信息，满足了驱动程序运行的所有需求。

OSKit，DDEKit 和 Genode 在自己的系统中构建了 Linux 代码运行环境，而 K42 则是在 Linux 中构建自己的系统，最后两种方法均是通过虚拟机间接地保留了代码的运行环境，这些系统的共同之处在于为 Linux 代码提供了运行环境。因此，无论是设备驱动、协议栈，还是其他 Linux 内核代码，重用的必要过程之一便是为其提供完整的运行环境。如何快速地帮助开发者搭建 Linux 设备驱动的运行环境便是本文的研究目标。

2.2 代码依赖分析

代码依赖分析属于程序依赖分析的一种。程序依赖分析在很多领域都具有十分重要的研究意义，其中最普遍的程序依赖分析的应用便是代码浏览器中提供的函数、类型跳转等功能。除此之外，国内外的学者均利用程序依赖分析实现了各个方面的研究内容。

Franck Xia 等人[7] 在 2004 年提出了“变化影响依赖(Change Impact Dependency，CID)”的概念，用来分析软件的可维护性。CID 通过结合变化影响分析和依赖分析，能够分析软件的可维护性。2001 年，Zhifeng Yu 等人通过研究程序中隐藏的依赖性，提出了使用抽象系统依赖图(Abstract System Dependence Graph)分析程序中隐藏的依赖性的方法[9]。周晓宇等人于 2004 年提出了 C 程序单元级依赖性分析[12]，将依赖性分析的粒度从语句级别细化到了单元级别，包括 C 程序中函数间的调用依赖、参数传递依赖、全局数据依赖以及文件间的包含依赖和外部变量定义依赖，并提出了单元依赖图来表达这些依赖关系。2007 年，Lulu Huang 和 Yeong-Tae Song 将动态影响性分析与程序依赖分析相结合，实现了一种针对面向对象程序的精确的动态影响性分析[8]。

程序依赖分析的应用范围非常广泛，其优点在于能够从语法层次上理解程序代码，帮助程序员更好地理解或者维护大型软件程序，或者为研究者提供将分析范围受限的方法扩展到整个程序中。

2.3 设备驱动的内核服务需求分析

设备驱动的重用过程的本质是在目标环境中建立设备驱动运行环境的过程。Linux 内核本身提供的所有内核服务是一个能够运行 Linux 中所有设备驱动的环境，但重用一个设备驱动并不需要实现所有 Linux 内核服务。不同的设备驱动所需要的内核服务并不相同，但可能会存在相同的部分。2014 年，茅俊杰博士[13] 对 Linux 设备驱动的内核服务需求特征进行了研究与分析，从重用设备驱动的角度，通过分析设备驱动源代码和提交的历史信息，对不同类型的设备驱动所需要的内核服务进行了分类统计和变化分析，得到了如下结论[13]：

1)在基于设备子系统的分类下，同类设备驱动对内核服务的需求具有相似性，不同种类的设备驱动对内核服务的需求具有较大的差异；

2)绝大多数内核服务接口的改动不涉及接口语义的变化，而是函数原型的变化。

结论 1)表明不同 Linux 设备驱动的重用工作并不是完全没有交集的，并且如果两个设备驱动属于同一设备子系统，那么其重用工作中可能有很大一部分是相同的。因此，同一类型的设备驱动的重用工作往往集中在第一个上。结论 2)表明在不考虑设备驱动本身的更新改动的情况下，绝大多数由于内核服务接口的改动变化而导致的设备驱动的改动变化是可以按照固定模式通过工具来自动分析完成的。

这两个结论是从设备驱动对内核服务的需求角度得到的，与传统的对设备驱动的理解和分类相比，这种方式对于设备驱动的重用工作帮助更大，也更加合理。因此，本文从设备驱动对内核服务的需求角度出发，对 Linux 设备驱动重用进行研究，提出了结合代码依赖分析技术自动构建 DDE 的方法。

3 利用代码依赖分析构造 DDE

3.1 整体思路

完整的 Linux 内核无疑是能够支持所有设备驱动的运行环境，但对于单个或者部分设备驱动来说，完整的 Linux 内核过于庞大，在重用过程中并不需要完整实现所有内核服务。因此提出了使用代码依赖分析技术对设备驱动需要的内核服务进行分析的方法，以帮助重用工作构造仅包含必要内核服务的 DDE。

利用代码依赖分析构造 DDE 的整体设计如图 1 所示。分析工具通过代码依赖分析分析设备驱动所依赖的内核服务，并将其提取出来构造成 DDE，然后将重用的设备驱动和构造的 DDE 放在一起进行验证，最后开发者可以在 DDE 中封装目标操作系统中的内核服务，使设备驱动可以间接依赖

(图 1 中由虚线表示)目标操作系统的内核服务运行。本设计思路主要分为两个部分:代码依赖分析和构造 DDE。

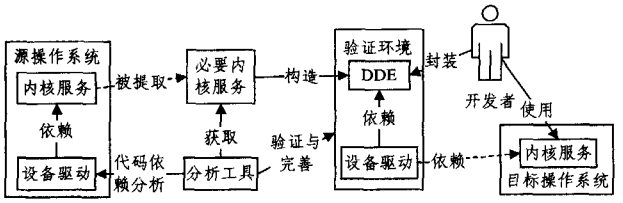

图 1　利用代码依赖分析构造 DDE 的整体设计

3.2　代码依赖分析

代码依赖分析属于程序依赖分析的一种。程序依赖性分析是一种重要的程序分析与理解的方法,根据分析粒度的不同,程序依赖性分析可以分为文件间依赖分析[11]、函数依赖分析[12]和代码依赖分析等。其中文件间依赖分析的粒度最大,代码依赖分析的粒度最小。代码依赖分析能够分析出每条语句所包含的语法级别的依赖关系,这能够保证分析结果与设备驱动的依赖性,过滤掉其他无依赖性的代码。

构造设备驱动 DDE 需要分析出驱动代码中包含的所有依赖关系,这种依赖关系可以通过语法来体现,如函数的调用、声明和定义等。由于 Linux 内核代码数量庞大,并且包含了许多底层代码的设计技巧,导致很多分析工具并不能很好地支持对 Linux 内核的分析。编译器能够将源代码转换成抽象语法树(Abstract Syntax Tree,AST)。AST 是源代码的抽象语法结构的树状表现形式,树上的每个节点都表示源代码中的一种结构,它可以完整地表现出源代码的语法信息。因此,采用对 AST 的分析来实现代码依赖分析。

通过对 Linux 内核源码中语法信息的总结和分类,将代码依赖分析的结果分为以下 5 个子集。

1)DECLS。与分析代码具有依赖关系的函数,数据类型。

2)ALL_DECLS。在生成 AST 过程中遇到的所有函数,数据类型,无论是否存在依赖关系。

3)MACROS。在分析过程中遇到的所有宏定义。

4)DEPS。头文件之间的包含关系。

5)PROTOTYPES。函数的声明原型。

算法 1 是描述代码依赖分析过程的伪代码。通过算法 1 可以得到上述 5 个子集的结果。

算法 1　代码依赖分析

```
设 n 是 AST 中的一个语法节点。
CodeDependencyAnalysis(n)
1. IF(n. declare_file∉DEPS)
2.    DEPS:=DEPS∪n. declare_file;
3. IF(n. type==DECLARATION)
4.    ALL_DECLS:=ALL_DECLS∪n;
5.    IF(n. file ==analyzed_file)
6.       DECLS:=DECLS∪n;
7.    IF(n. body≠∅)
8.       PROTOTYPES:=PROTOTYPES∪n. body;
9. IF(n. type==MACRO)
10.   MACROS:=MACROS∪n;
```

算法 1 分析了 AST 中的每个节点 n,根据 n 的声明文件确定依赖集合 DEPS,根据 n 的类型将其分类到 ALL_DECLS 或 MACROS 子集中。如果 n 所处文件与被分析的源码文件相同,则表示 n 是被源码直接调用。函数类型的节点会将其函数声明保存在 PROTOTYPES 中。

3.3　构造 DDE

构造 DDE 需要满足两方面的内容:头文件的依赖关系和内核服务的完整。

Linux 中设备驱动源代码会引用内核中的头文件,而被引用的头文件也会引用其他头文件,这就是头文件之间的依赖关系。如果没有处理好这种依赖关系会导致在编译过程中出现函数重定义、提前声明等问题。构造的 DDE 要能够在不修改设备驱动源代码的情况下将其重用,这就需要 DDE 的结构与 Linux 内核保持一致。

内核服务的完整是保证设备驱动可以正常运行的前提,这里的完整并不是指所有的 Linux 内核服务,而是指设备驱动依赖的相关内核服务。代码依赖分析的目的便是找出设备驱动依赖的所有内核服务,以代码依赖分析为基础,本文设计了 DDE 的构造算法,以保证 DDE 中仅包含设备驱动在语法层次上依赖的最少内核服务。

算法 2　构造 DDE

```
设 dde 为最终构造的 DDE 的对象。
ConstructDDE(dde):
1. FOREACH(i∈DEPS)
2.    dde. Headers:=dde. Headers∪i;
3. FOREACH(d∈DECLS)
4.    dde. Declarations:=dde. Declarations∪d;
5. FOREACH(m∈MACROS)
6.    dde. Macros:=dde. Macros∪m;
7. DO
8.    dde. lacks:=VerifyDDE(dde);
9.    FOREACH(l∈dde. lacks)
10.      f:=ALL_DECLS. find(l);
11.      dde. Declarations:=dde. Declarations∪f;
12.      PROTOTYPES:=PROTOTYPES∪f. body;
13. WHILE(dde. lacks≠∅)
14. FOREACH(p∈PROTOTYPES)
15.   dde. Implementations:=dde. Implementations∪p;
```

算法 2 以一个空的设备驱动运行环境对象 dde 为输入,通过读取 5 个子集中的内容来构造目标设备驱动的 DDE,并且可以通过验证过程发现 DDE 中缺少的内核服务并进行修复和完善。如果将 ALL_DECLS 中所有的内核服务都添加到 DDE 中,可以使算法 2 省去验证和完善的过程,但会导致最终的 DDE 过于庞大并存在很多冗余代码。通过基于直接依赖的内核服务构造 DDE 并逐步补充完善,可以得到不存在冗余代码的 DDE。

4　驱动重用实例

为了证明本文方法的可行性,设计并实现了工具 header-gen,并且成功将 Linux-3. 10 的 e1000 网卡驱动重用到了

ucore[10]中。ucore 是一个微型操作系统，包含了大部分主要的操作系统内核服务，但并不支持网络传输和网卡设备。对于重用过程，首先使用 header-gen 生成 e1000 的 DDE，然后将 DDE 放入 ucore 中，并通过 ucore 的内核服务实现 DDE 中必要的内核服务，最后运行并测试 e1000 的功能。

表 1 是使用 header-gen 对包括 e1000 在内的 4 个驱动程序进行代码依赖分析后得到的 5 个子集的数量情况。使用代码统计工具进行数量统计，header-gen 构造的 e1000 的 DDE 代码的总行数为 21941，相对于 Linux-3.10 内核共 190 多万的头文件代码数量已经非常精简。

表 1 驱动代码依赖分析结果

子集	DECLS	ALL_DECLS	MACROS	DEPS	PROTOTYPES
e1000	2643	13682	2841	1649	1443
ixgbevf	2471	12838	2067	1608	1328
e1000e	2758	13718	3251	1685	1577
virtio	1676	8881	1590	1250	947

为了使 e1000 能够在 ucore 中正常运行，需要实现几种内核服务：1)PCI；2)net_device 的创建、配置和操作；3)DMA 地址的分配和映射；4)ioremap 的实现；5)NAPI 收包的相关函数；6)sk_buff 的创建和操作；7)中断注册。由于 ucore 系统本身并不支持 TCP/IP 协议，因此为了测试重用的 e1000 能否正常工作，在 ucore 中移植了 lwip，使得 ucore 可以支持简单的 TCP/IP 协议。在 ucore 中搭建 http 服务，使用主机系统对 ucore 进行 http 访问，通过这种方式来对 e1000 进行测试。

在实验中，分两组对 e1000 进行重用，其中一组使用 header-gen 构造的 DDE，另一组采用传统的方式直接将 e1000 的代码移植到 ucore 中。最终的结果是两组均成功将 e1000 重用到 ucore 中，并且可以通过文件传输的测试内容，但使用 header-gen 的小组所用时间明显少于另一组。由于重用过程中没有对 e1000 源代码进行改动，因此使用 header-gen 的小组不需要去理解 e1000 的内部设计逻辑；而另一组需要花费大量时间研究 e1000 的源代码，以便能够分析出 ucore 需要提供哪些内核服务给 e1000。实验结果表明，本文利用代码依赖分析构造 DDE 的方法能够为开发者创建精简的 DDE，方便其开展设备驱动重用的相关工作，从而可以有效地减少 Linux 设备驱动重用的开发时间，降低重用工作的难度。

结束语 本文针对 Linux 设备驱动重用提出了使用代码依赖分析构造 DDE 的方法来进行重用工作，并对这一方法进行了具体的构思和设计，最后实现了相应的工具 header-gen。通过使用 header-gen 成功将 Linux 中的 e1000 驱动程序重用在微型操作系统 ucore 中，验证了该方法的可行性。在真实的重用工作中，通过使用该方法构造的 DDE 来开展重用工作，切实体会到了该 DDE 为重用工作提供了很好的开端，并且节省了大量时间，从一定程度上简化了重用过程。但我们也在实践中发现了这个方法的不足之处：其构造的 DDE 受限于代码依赖分析的能力范围，只能在语法层次上分析设备驱动对内核服务的需求。如果能够在语义层次进行分析，则可提供更完备的 DDE 构造能力，这也是我们下一步的研究重点。

参 考 文 献

[1] APPAVOO J, AUSLANDER M, DASILVA D, et al. Utilizing Linux kernel components in K42: Technical report[R]. IBM Watson Research, 2002.

[2] POESS B. Binary device driver reuse[D]. Universität Karlsruhe (TH), 2007.

[3] STOESS J L V V U J, GÖTZ S. Unmodified device driver reuse and improved system dependability via virtual machines[C]// Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation. 2004.

[4] FORD B, BACK G, BENSON G, et al. The Flux OSKit: A substrate for kernel and language research[C]// Proceedings of the 16th ACM Symposium on Operating Systems Principles. ACM, 1997.

[5] ALONI D. Cooperative linux[C]// Proceedings of the Linux Symposium. 2004: 23-31.

[6] GENODE: Operating System Framework[OL]. http://genode.org.

[7] BOHNER S A, ARNOLD R S. Software change impact analysis [M]. Wiley-IEEE computer society press, 1996.

[8] HUANG L, SONG Y T. Precise dynamic impact analysis with dependency analysis for object-oriented programs[C]// 5th ACIS International Conference on Software Engineering Research, Management & Applications, 2007 (SERA 2007). IEEE, 2007: 374-384.

[9] YU Z, RAJLICH V. Hidden dependencies in program comprehension and change propation[C]// 9th International Workshop on Program Comprehension, 2001 (IWPC 2001). IEEE, 2001: 293-299.

[10] GitHub[OL]. https://github.com/chyyuu/ucore_plus.

[11] HUANG W W. C program file dependency analysis[D]. Nanjing: Southeast University, 2004. (in Chinese)
黄文伟. C 程序文件间依赖性分析[D]. 南京：东南大学，2004.

[12] ZHOU X Y, HUANG W W, SHI L, et al. Analysis of Dependences among C Program Units[J]. Computer & Digital Engineer, 2005, 32(6): 1-4. (in Chinese)
周晓宇，黄文伟，史亮，等. C 程序单元级依赖性分析[J]. 计算机与数字工程，2005，32(6)：1-4.

[13] MAO J J, CHEN Y. Understanding requirement characteristics of Linux device drivers for kernel services[J]. Journal of Tsinghua Universiry (Science and Technology), 2015(8): 911-915. (in Chinese)
茅俊杰，陈渝. Linux 设备驱动的内核服务需求特征[J]. 清华大学学报(自然科学版)，2015(8)：911-915.