第 33 卷 第 5 期 计 算 机 工 程 2007 年 3 月
Vol.33 No.5 Computer Engineering March 2007
·软件技术与数据库· 文章编号：1000—3428(2007)05—0065—03 文献标识码：A 中图分类号：TP311

基于 LKM 的嵌入式 Linux 内核调试模型及实现

王培东，黄凌碧
（哈尔滨理工大学计算机科学与技术学院，哈尔滨 150080）

摘 要：为了克服嵌入式 Linux 调试领域中的插桩模型的缺点，该文在 LKM 技术的基础之上，引入了“寄生技术”，提出了一种嵌入式 Linux 内核调试模型，实现了调试代理的功能。实验表明，该模型降低了寄生代码与嵌入式 Linux 内核的耦合度，减少了对嵌入式 Linux 内核的修改量，提供了更为灵活的扩展机制和更容易移植的条件。

关键词：插桩；LKM；寄生技术；嵌入式 Linux；调试代理

LKM Based Embedded Linux Kernel Debug Model and Its Implementation

WANG Peidong, HUANG Lingbi
(Dept. of Computer Science & Technology, Harbin University of Science & Technology, Harbin 150080)

【Abstract】 For overcoming defects of the stub model in embedded Linux debug domain, this paper refers to parasite technology and proposes a new embedded Linux debug model based on LKM. The debug agent based on the new model is developed. The experimental results indicate that comparing with KGDB based on the stub model, the new model reduces the coupling degree between the parasitic code and the embedded Linux kernel, and provides the more flexible expansion mechanism. The new model rarely modifies the embedded Linux kernel, so it is easier to port.

【Key words】 Stub; Loadable kernel module(LKM); Parasite technology; Embedded Linux; Debug agent

本文提出了一种新的嵌入式 Linux 调试模型。该模型引入了寄生技术，定义了一个硬件抽象层(DHAL)，将调试代理与嵌入式 Linux 内核从逻辑上分开，利用了 Linux 可加载内核模块技术 LKM(Loadable Kernel Module)，将调试代理扩展成嵌入式 Linux 内核的一部分，通过调试代理与宿主机调试器 GDB 进行通信，来完成对嵌入式 Linux 内核及其驱动程序的调试。

1 相关技术

寄生技术的提出主要是为了提供一种容错系统的故障检测手段。其主要思想是通过让一段特定的代码附着在原始程序当中来达到故障检测的目的。为保证程序的可重视性，把这段代码称之为“寄生代码”，把寄生代码依附的代码称之为“宿主代码”[1]。寄生代码运行在内核级，以提高其运行效率。寄生代码插入方法有 2 种：(1)静态插入寄生代码，寄生代码与源程序混为一体；(2)动态插入寄生代码，通过陷阱技术和中断技术，在运行时根据实际情况激活[2]。2 种代码插入方法的原理如图 1 所示。

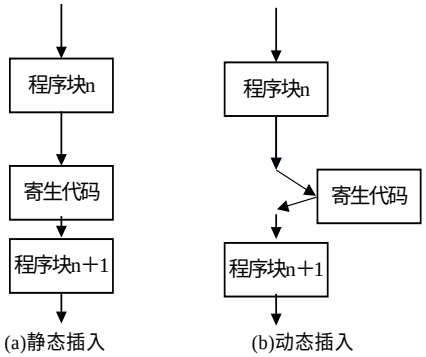

图 1 寄生技术的 2 种代码插入方式

LKM技术[3]是被Linux系统所采用的内核扩展技术，一个模块(Module)提供了一个功能，这些模块是可以按需要随时装入和卸载。这样做可以使得内核的大小和通信量都达到最小。模块不必预先绑定在内核当中，但是当模块安装到内核后就作为核心的一部分，不进行实时安全检测，全速运行。这样做有 2 个优点：(1)将来修改内核时，不必全部重新编译，可节省不少时间；(2)减少内核对系源的占用，内核可以集中精力做最基本的事情，把一些扩展功能都交由模块实现。每个LKM模块必须(最少)提供 2 个基本的函数，即

```
int init_module(void) /*模块初始化时调用*/  {...}
void cleanup_module(void) /*模块卸载时进行清理*/  {...}
```

嵌入式 Linux 系统中与模块相关的命令有：

(1)lsmode：列出已经被内核调入的模块；

(2)insmod：将某个 module 插入到内核中；

(3)rmmod：将某个 module 从内核中卸载；

(4)depmod：生成模块间依赖文件；

(5)modinfo：显示内核模块信息；

(6)modprobe：自动加载或卸载内核模块。

LKM 技术是本文所使用的关键技术，它是本文所提出的模型能够实现的前提。

2 插桩模型研究

嵌入式Linux系统中对内核进行远程调试，需要内核中插入一段实现调试功能的代码模块(stub)[4]，该功能模块负责接管目标机CPU的所有异常处理，能够读写被调试内核的寄存器现场和内核地址空间，并负责驱动选定的通信接口，该功

作者简介：王培东(1953－)，男，教授，主研方向：并行处理与控制技术，嵌入式应用；黄凌碧，硕士生

收稿日期：2006-03-08 **E-mail：**wpdlg@hrbust.edu.cn

能模块与Linux内核同时初始化，在内核的引导过程中就可以对内核进程进行调试。功能模块通过指定的远程调试协议与宿主机上的调试器进行交互，将目标机发生的异常信息传送给宿主机，调试器将调试命令发送给目标机，由该功能模块实现这些调试命令。该模块还要实现单步、断点等功能供调试器调用[5,6]。插桩模型的基本结构如图 2 所示。

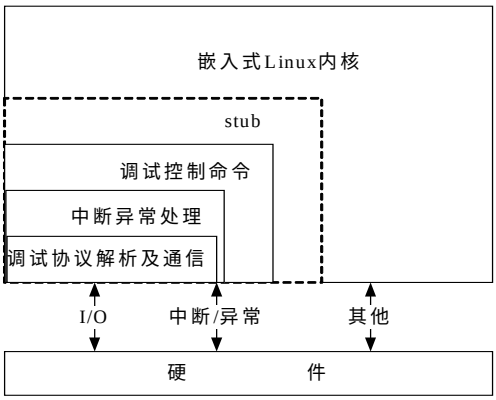

图 2 插桩模型结构

研究发现，基于插桩模型的实现代码对 Linux 内核的修改主要体现在以下几个方面：

(1)系统的初始化过程。插桩模块要在第一时间控制系统的中断和陷阱，通过对中断和陷阱处理函数的修改，插桩模块才能随时接管目标机上的异常事件。

(2)调试功能的实现部分。由于调试功能的实现必须和异常处理系统紧密结合起来，因此对内核中断和异常处理部分的修改幅度很大。

(3)与宿主机通信部分。需要修改串口中断处理函数，并且添加通信协议解析部分的寄生代码。

插桩模型是一种纯软件的解决方案,利用 GDB 或 Insight 的强大客户端功能，可以为中小型嵌入式 Linux 应用构建一个价格低廉、功能完善、界面友好的调试平台。但是插桩模型的缺点也是显而易见的：

(1)对嵌入式 Linux 内核源代码修改的幅度较大，有可能引入新的问题。

(2)没有统一的修改标准，寄生代码与内核代码紧密耦合，不利于系统的扩展和移植，无法跟上嵌入式 Linux 的发展速度。

(3)插入代码必须和嵌入式 Linux 内核一起静态编译，致使其本身的开发调试效率较低。

(4)插入代码的卸载过程比较复杂，卸载后很容易在嵌入式 Linux 内核内引入新的 BUG，导致最终产品不稳定。

在插桩模型的基础上改进,是无法克服以上这些问题的，这就需要提出一种新的模型，既能保留插桩模型的优点，又能弥补插桩模型的不足。

3 新模型的基本思想和结构

为了弥补插桩模型的缺陷，本文引入了“寄生技术”中第 2 种寄生代码的插入思想，提出了一种新的调试模型。该模型基本的设计思想是：尽量不去修改嵌入式 Linux 的内核，而是将内核需要修改的地方抽象成数据或函数指针，将这些指针集中到一起。当调试代理模块没有加载时，这些指针都指向嵌入式 Linux 内核本身的实现，这时的嵌入式 Linux 内核相当于没有修改过的“裸核”。当调试代理模块被加载到嵌入式 Linux 内核当中的时候，这些指针指向了调试代理模块中相应的实现，调试代理模块将嵌入式 Linux 内核的中断描述符表(IDT)等重要的数据结构备份起来，并用调试代理模块的数据结构和函数替换，这时，调试代理模块将完全接管中断、陷阱和异常处理，而嵌入式 Linux 内核只能通过指定接口与调试代理模块进行通信，嵌入式 Linux 内核已经完全被调试代理模块所监视和控制。调试代理模块在卸载前，会将备份的数据结构和函数指针回复，将控制权返还给嵌入式 Linux 内核，然后从嵌入式 Linux 内核中卸载下来，这时对于内核来说，调试代理模块就好像从来不存在一样。

结合上述的基本设计思想，本文提出了一种新的模型，其结构如图 3 所示。

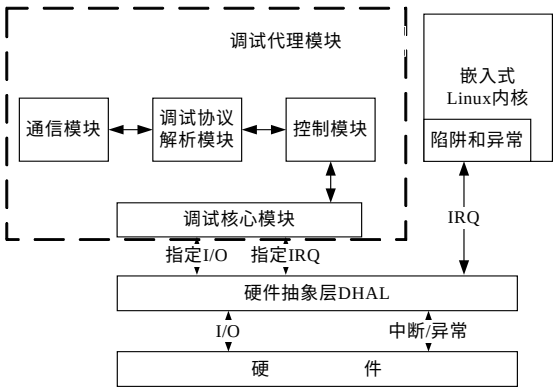

图 3 新模型的结构

硬件抽象层(Debug Hardware Abstract Layer, DHAL)是对目标机硬件体系以及嵌入式 Linux 内核修改部分的一个抽象。为了容易捕获那些对调试模块非常重要的内核函数，将所有需要的内部数据和函数的指针聚集到 DHAL 当中，当调试器需要时，调试代理核心模块可以将它们动态地转换，将 DHAL 中的函数指针指向那些用来替换 Linux 内核函数的函数。DHAL 的本质是一组和内核相关的数据结构和函数接口。

调试核心模块是整个调试机制的实现核心，是在 DHAL 中定义的接口函数的实现部分，它实现了调试时所必须的中断异常处理函数和一些替换函数,并且提供统一的调试机制，当调试代理核心模块加载到内核并被激活后，嵌入式 Linux 内核的任何举动都在它的监视和控制之中。

通信模块的工作是初始化通信端口(如串口、USB 接口、网络接口等)，提供接收数据，发送数据等基本的通信函数，负责与宿主机调试器的通信。

调试协议解析与转换模块主要负责两方面工作：(1)将通信模块接收的数据按照远程通信协议的格式进行解析，并转换成指定的调试命令传给控制模块；(2)将从控制模块接收到的信息转换成指定的格式发送到通信模块。

控制模块实现具体的调试命令，如查看寄存器命令、设置断点、单步执行等。

4 硬件抽象层和调试核心模块的实现

为了更好地对本文提出的模型进行验证，针对 x86 平台实现了 DHAL 层的抽象，主要代码如下：

```
struct debug_hal {
    struct desc_struct *idt_table;
//指向 IDT(中断描述表)的指针；
    unsigned int (*getflags)(void);
//获得标志寄存器的状态
        void (*setflags)(unsigned int flags);
```

```
//设置标志寄存器的状态
        void (*mask_and_ack_irq)(unsigned int irq);
//控制中断控制器的函数，屏蔽某些中断
        void (*unmask_irq)(unsigned int irq);
//控制中断控制器的函数，打开某些中断
        void * irq_controller_lock;
        void *irq_desc;
//描述当前中断的数据结构(状态,中断处理函数,中断嵌套层次等)
        int *irq_vector; //调试代理模块所使用的中断向量表
        unsigned int *regs;    //指向寄存器的指针
        ...};
```

为了保证和 Linux 内核的兼容，还需要定义以下的数据结构，这些数据结构保存了 Linux 内核的原始状态：

```
static struct debug_hal linux_debug_hal;
//保存 Linux 内核的原始状态，如 IDT 的指针，寄存器的指针
extern struct irqdesc irq_desc[];//中断描述符表的实际保存位置
extern unsigned int linux_regs[]; //寄存器信息的实际保存位置
```

DHAL 将调试模块对嵌入式 Linux 中的修改部分定义成一组程序界面，调试模块只使用这组界面和嵌入式 Linux 内核进行沟通。这样一来就可以把对嵌入式 Linux 内核源代码的改动降至最低，使调试代理模块和嵌入式 Linux 内核之间的耦合度更小。

调试核心模块是本模型的核心，它的存在使调试代理的各个模块成为一个有机的整体。调试核心模块被加载到嵌入式 Linux 内核中时，还处于休眠状态，好像没有事情发生一样。当系统需要时，其他模块调用相应函数将其激活。调试核心模块一旦被激活，嵌入式 Linux 内核与硬件之间的联系就会被调试核心模块捕捉、过滤。从这一刻起，中断和异常处理就被调试核心模块接管。

模块被加载时，系统首先运行的是这个模块的初始化函数 init_module，其主要完成以下功能：

(1)初始化 DHAL 层所有的控制变量和数据结构。

(2)备份嵌入式 Linux 内核的原始中断描述符表等重要数据。

(3)初始化与中断相关的芯片。

当调试核心模块被加载并激活后，最重要的变化是嵌入式 Linux 内核不再具有中断和异常的管理能力。事实上，DHAL 中的中断异常处理函数指针不再指向由嵌入式 Linux 内核实现的处理函数，而是改为指向由调试核心模块实现的新的处理函数，这些新函数将负责为嵌入式 Linux 内核提供一套对中断和异常的软件模拟。这样，调试核心模块做到了既保持嵌入式 Linux 内部中断和异常操作的前后一致，又确保了 Linux 可以随时被调试核心模块所控制。

调试核心模块最重要的操作“中断和异常拦截”的基本工作流程如下：

(1)截获中断和异常信号；

(2)如果不是调试核心指定的中断信号，将这个中断传递给嵌入式 Linux 内核；

(3)如果是异常信号，则马上挂起嵌入式 Linux 内核，则按照预设的方式将异常信息发送给宿主机调试器，并等待宿主机调试器的命令；

(4)如果指定的中断信号，如串口中断，则按照预设的方式进行中断处理，并将从宿主机接收到调试命令解析出来，交与指定的调试函数处理。

中断和异常的拦截如图 4 所示。

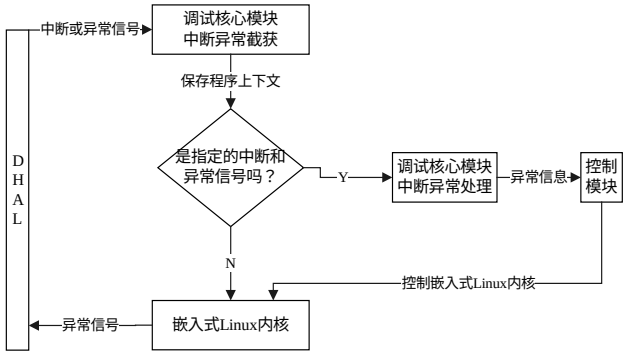

图 4 中断和异常的拦截

5 验证结果

针对 x86 平台，本文与基于插桩模型的调试代理 KGDB 进行了比较，如表 1 所示。可以看出，新模型比插桩模型更具优势。

表 1 新模型与插桩模型比较结果

特性 \ 实现方法	基于插桩模型 (KGDB)	基于新模型
对嵌入式 Linux 内核源代码的修改	分布于大约 35 个文件中，修改了共约 5 000 行内核代码	集中在 entry.s，system.h，irq.c 文件中，修改约 200 行内核代码
新增 LKM 模块	无	4 个
内核耦合度	紧耦合	松耦合
实现难度	高	低
移植性	差	好
扩展性	差	好

6 结论

本文在深入研究插桩模型的基础上，引入了寄生技术，提出了一种新的嵌入式 Linux 调试模型。在该模型的基础上，利用 LKM 技术，实现了调试代理的功能。实验结果表明，该模型具有实现简单、被动修改内核、机制与策略分离、易于扩展等优点，完全可以替代插桩模型，成为中小型嵌入式 Linux 应用调试方案的首选。

参考文献

1 Martin T. A General Theory of Software Reliability[J]. IEEE Trans. on Reliability, 1990, 39(1): 92-96.

2 Jang Yi, Wang Peidong. The Fault-tolerant System Based on Parasite Technology and uC/OS[C]//Proc. of International Conference on Embedded Systems. 2001: 297-300.

3 Degoyeneche J M, Desousa E A F. Loadable Kernel Modules[J]. IEEE Software, 1998, 15(1): 65-71.

4 张　磊, 王学慧. Linux 内核调试技术[J]. 计算机工程, 2003, 29(10): 81-83.

5 郭胜超, 吕　强, 杨季文, 等. GDB 远程调试及其在嵌入式 Linux 系统中的应用[J]. 计算机工程与科学, 2004, 26(10): 100-103.

6 张栋岭, 刘献科, 邓晓艳, 等. 嵌入式应用的远程调试[J]. 计算机工程, 2003, 29(11): 76-78.