分 类 号：TP302.7
研究生学号：2009542049
单位代码：10183
密 级：公 开

吉 林 大 学

硕士学位论文

基于内核的虚拟机的研究

Research on Kernel-based Virtual Machine

作者姓名：时卫东
专 业：软件工程
研究方向：并行与分布式计算
指导教师：徐高潮 教授
培养单位：软件学院

2011 年 4 月

基于内核的虚拟机的研究

Research on Kernel-based Virtual Machine

作者姓名：时卫东

专业名称：软件工程

指导教师：徐高潮 教授

学位类别：软件工程硕士

答辩日期：2011年 5月 28日

吉林大学硕士学位论文原创性声明

本人郑重声明：所呈交学位论文，是本人在指导教师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：[illegible]

日期：2011 年 6 月 2 日

摘　　要

基于内核的虚拟机的研究

目前国内外关于云计算、虚拟化的研究如火如荼。虚拟化软件的快速发展，其种类繁多，广泛的使用场合，以及 Linux 操作系统的广泛应用和 KVM(Kernel-based Virtual Machine，即基于内核的虚拟机)技术的日益成熟与快速发展，再加上国内关于虚拟化技术和虚拟机的研究成果现在还比较少，使得关于虚拟化技术与虚拟机的研究显得更加重要和迫切。

虚拟化技术的研究和应用已经有几十年的历史了。虚拟化技术最初主要应用于服务器上面，但随着个人计算机性能的增强，针对个人机的虚拟化技术的应用日益多了起来，各种虚拟化软件也层出不穷，并且得到了广泛的使用。

近几年随着虚拟化技术的不断发展，硬件辅助虚拟化技术的出现及其逐渐成熟，弥补了原先软件虚拟化技术的不足，提高了虚拟机的性能，大大地降低了虚拟化软件的开发难度，促进了虚拟化软件的发展。虽然虚拟化技术的发展与进步使得虚拟机的性能不断地得到提升，但虚拟机的性能较之物理机性能而言还是有一定的差距，因此，虚拟机的性能测试也是非常重要的。可以通过对虚拟机性能的测试来对不同的虚拟化产品的优劣进行比较与分析，从而对在选择虚拟化产品时提供一定的依据和帮助。

目前市场上已经出现了许多优秀的虚拟化产品，如 VMware 公司所提供的 VMware 系列虚拟化产品，RedHat 公司主推的 KVM，剑桥大学开发的 Xen 以及 Oracle 公司的 VirtualBox 等等。在众多的虚拟化软件中，KVM 的出现虽然比较晚，但 KVM 开源、基于硬件辅助虚拟化、结构简单、集成与 Linux 内核、性能优越等优点使其成为众多虚拟化软件中的佼佼者。本文是针对 KVM 的学习与研究，主要介绍了其实现机制，使用方法以及对其性用能进行了测试。因此，本文对 KVM 以及虚拟机的学习与使用能够起到一定的帮助作用。

本文的总体组织结构是首先对虚拟化技术以及几款主流的虚拟化软件进行了简单的介绍，然后对 Linux 内核中关于的 KVM 部分核心代码进行了分析，之后又对 KVM 的安装使用进行了介绍与说明，最后对 VMware Workstation、KVM、QEMU 和 VirtualBox 这几种主流的虚拟化软件上创建的虚拟机的性能进行了测试，比较与分析

了它们在网络吞吐量、磁盘输入/输出、处理器等方面的性能表现，并且对运行在这些虚拟化软件中的虚拟机的综合性能也进行了测试。

关键词：

KVM，虚拟化，Linux 内核，虚拟机

Abstract

Research on Kernel-based Virtual Machine

Nowadays, researches on cloud computing and virtualization are very hot. The rapid development and widespread use of virtualization software, plus the Linux operating system getting more and more popular and KVM (Kernel-based Virtual Machine) technology has become more sophisticate. And until now researches on virtualization technology and virtual machine in our country are still relatively short. All of these making researches about virtualization technology and virtual machine are more important and urgent.

Researches on virtualization have been doing for several decades. At the beginning, virtualization technology is mainly applied to server machines, but with the personal computer performance enhancement, there are more and more virtualization software are used for personal computers.

With the development of virtualization technology and hardware support virtualization technology in last few years, some defects of software virtualization technology have been improved, such as increased the performance of virtual machine, greatly reduced the difficulty of virtualization software development and accelerated the development of virtualization software. Although virtualization technology has been developed for a long time, and it has enhanced the performance of virtual machine, there are still some performance loss between virtual machines and physical machines. So the virtual machine testing is very important. Through the tested results of virtual machines' performance, we can compare the advantages of different virtualization products. And finally provide some useful information when choose virtualization software.

There are many excellent virtualization software, such as VMware Workstation and VMware vShpere which are belonged to VMware Company, KVM from RedHat, Xen developped by Cambridge University and Oracle's support VirtualBox, etc. Among so many virtualization products, although KVM came out late, it has become one of the most

popular virtualization software by its advantages, such as open source, hardware support virtualization, integrated into Linux kernel, simple implementation and high performance, etc. This paper focused on studying KVM. It introduced the implementation mechanism of KVM, gave a simple tutorial about using KVM and run some performance tests on it. So this paper will provide a little help to those people who are learning and using or to those who are going to learn and use virtual machines.

This paper first introduced virtualization and several virtualization software, then analyzed part of the KVM core-code which is integrated in the Linux kernel source-code. After that described how to install and use KVM. Finally tested and compared the performance of several virtual machines which are created by virtualization software. Compared and analyzed their performances about network throughput, I/O speed and CPU processing differences. In addition, this paper also tested the whole virtual machine system performance.

Keywords:

KVM, Virtualization, Linux Kernel, Virtual Machine

目　　录

第 1 章　绪 论

1.1　研究背景

随着虚拟化技术在 x86 架构上的流行，各种虚拟化产品也随之出现，主要的代表产品有 VMware、KVM、Xen、VirtualBox、QEMU 等等。采用虚拟化技术可以带来很多好处，比如可以在同一台机器上同时运行多个操作系统，提高资源的利用率，从而可以降低成本。

虚拟化也是当今热门研究领域——云计算的基础。因此，国内外无论是学术型机构(比如国内外的许多大学与研究所)还是商业公司(如 Intel，AMD，IBM，RedHat，VMware，微软，西门子等等)，对虚拟化的研究都投入了巨大的精力与资源，而且也取得了丰硕的成果，最直接的成果就是多种多样的虚拟化软件以及与其相关软件的应用。

在 KVM，即基于 Linux 内核的虚拟机的发展历程中，RedHat 公司(KVM 最初由 Qemuranet 公司开发，最初版本的开发工作主要由 Kivity 和 Kamay[1]两人完成，后来 Qemuranet 公司被 RedHat 收购。因此，目前 RedHat 公司推出的虚拟化产品主要就是 KVM)、Intel 公司以及 AMD 公司起到了巨大的作用(Intel 和 AMD 都在其处理器中加入了新功能来支持虚拟化功能，即硬件辅助虚拟化技术，分别为 Intel-VT[2]和 AMD-SVM[3]。没有硬件的支持，KVM 将无法使用)。当然 KVM 的发展也离不开开源社区的支持，正是由于这些来自世界各地的编程爱好者的支持，才使得 KVM 得以快速的发展。

1.2　虚拟化技术简介

虚拟化是一种使用如软、硬件分区或聚合，机器模拟，仿真技术，分时共享等方法划分或组合计算资源来呈现一个或多个操作环境的技术[4]。虚拟化技术在服务器整合，内核的调试与开发，同时支持多个操作系统，提高系统安全和系统的动态迁移等方面有广泛的应用。

根据抽象程度的不同，虚拟化技术一般应用于下面五个抽象层之一：硬件抽象层、

指令集架构层、操作系统层、编程语言层(应用层)和库函数层[4]。本文介绍的虚拟化软件除 QEMU 属于指令集架构层外，其余几款如 VMware、KVM、Xen 和 VirtualBox 均属于硬件抽象层，如图 1.1 所示：

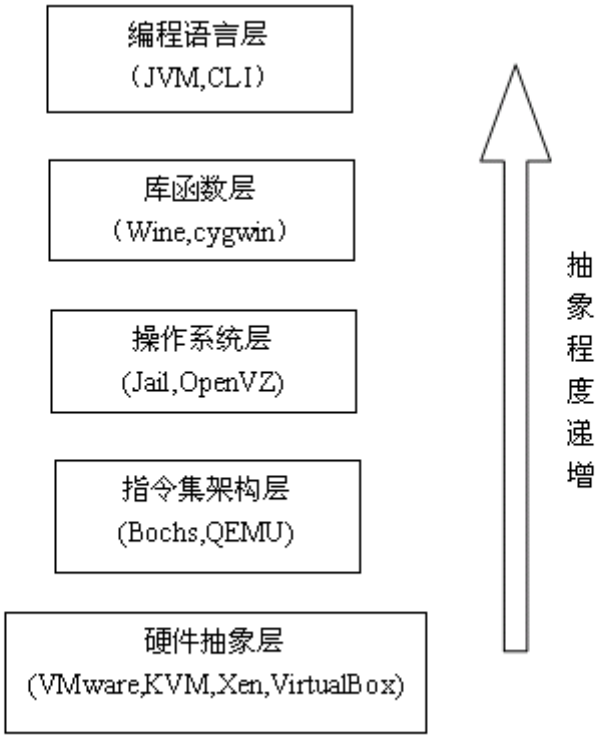

图 1.1　虚拟化技术应用层次分类

指令集架构层的虚拟化技术是通过软件模拟指令集来实现的。指令集架构层的虚拟化软件通常被称为模拟器[5]，模拟器通过软件模拟将 guest 机发出的指令转换为 host 机的指令在可用的硬件上执行。具有代表性的模拟器有 QEMU 和 Bochs。

硬件抽象层的虚拟化技术是当今虚拟化研究的重点与热点，它在多个客户操作系统之间复用硬件资源，位于此层的虚拟化软件被称为 Hypervisor 或 VMM(Virtual Machine Monitor)，由 Hypervisor 来管理与调度客户机的执行。

虚拟化技术主要有两种实现方式：全虚拟化(Full Virtualization)和泛虚拟化(paravitualization)[6]。全虚拟化是通过完全对底层硬件模拟来提供一个特定的虚拟机环境的虚拟化技术，使用全虚拟化创建的虚拟环境与真实的物理环境除性能差异外无任何差别。而泛虚拟化是对虚拟机提供一个与底层硬件相似的软件接口的虚拟化技术。泛虚拟化的目的是通过对 guest OS 本身做一定的修改来减少软件模拟的时间消耗，

从而提高虚拟机的性能，但非开源的操作系统则无法使用泛虚拟化软件。

近年来，由于 Intel 和 AMD 都在其处理器中提供了硬件辅助虚拟化的功能，有了硬件的支持，不仅使全虚拟化的性能开销大大降低，对全虚拟化的发展起到巨大的作用，而且使得开发 Hypervisor 的复杂性也明显降低，KVM 就是最好的例子。

1.3　虚拟化软件介绍

1.3.1　VMware

虽然现在各大 IT 公司都在争先恐后地涉足虚拟化领域，不过 VMware 公司是虚拟化领域的市场领导者。它是全球领先的虚拟化和云基础架构解决方案提供商，所提供的经客户验证的虚拟化解决方案可以大幅降低 IT 复杂性。目前在全世界拥有超过 170000 家用户，其中包括财富 100 强中的全部企业[7]。

VMware 公司提供的虚拟化产品种类繁多，包括有用于数据中心和服务器端 的 VMware vSphere 与用于桌面和终端用户的 VMware Workstation 等。本文在第 4 章虚拟化软件性能比较中用到的是 VMware Workstation 的试用版(可以免费试用 30 天)。VMware Workstation 使用简单，能够安全、可靠地支持多种操作系统，并且可以提供较好的性能。

1.3.2　KVM

KVM 也就是基于内核的虚拟机[8]，最初由 Qemuranet 公司开发，后来 Qemuranet 公司被 RedHat 公司收购，现在 KVM 的主要开发工作也就由 RedHat 公司与 KVM 社区的广大编程爱好者负责。KVM 同时也受到其他很多著名的 IT 公司的青睐，如 Intel、AMD、IBM 等。

KVM 是目前唯一一款集成到 Linux 内核中的虚拟化软件，它最早集成到了 Linux2.6.20 版本的内核中。正是由于 KVM 集成与 Linux 内核之中，它利用了内核提供的代码，如进程调度，内存管理等现有程序，使得 Linux 内核成为了一个 Hypervisor，而 KVM 本身的实现则相对简单。如图 1.2 所示：

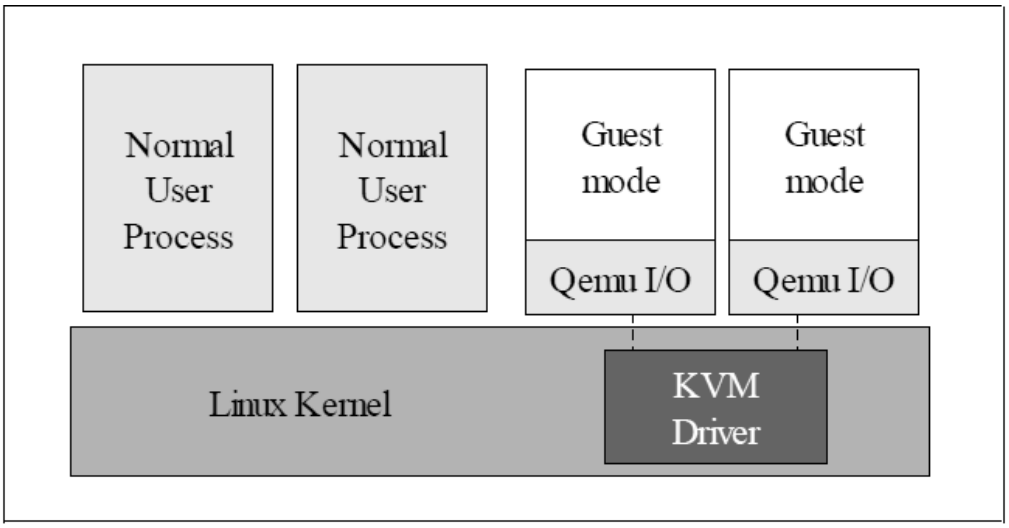

图 1.2　kvm Based Architecture[9]

一般而言，一个普通的 Linux 进程有两种执行模式：内核模式与用户模式。KVM 添加了一个新的模式，即客户模式，如图 1.2 所示。虚拟机运行于这个模式之下，客户模式也有其内核模式和用户模式。对主机操作系统而言，每一个虚拟机都被看作一个标准的 Linux 进程，通过进程调度程序调度执行。既然虚拟机被内核看作为一个 Linux 进程，那么，所有的进程管理程序或工具均可对其进行操作，如可用 top 命令查看虚拟机进程的资源使用情况。

从图 1.2 可以看出，KVM 在 Linux 内核中的角色是一个驱动程序，准确地说是字符驱动程序[10]，一般的 Linux 发行商在配置内核时都将 KVM 以模块的方式编译。在加载 KVM 模块之后，将会生成“/dev/kvm”设备文件，应用程序将通过系统调用 ioctl 对此文件操作来创建和运行虚拟机。

1.3.3　Xen

Xen 是由剑桥大学开发的一款功能强大的、开放源代码的、用于虚拟化的虚拟化软件。Xen 支持多种体系架结构的机器，如 x86、IA64 和 ARM 等，支持多种操作系统，如 Linux、Windows、Solaris 等等[11]。Xen 的功能强大，性能优越并且安全，而且最早提出并实现了泛虚拟化技术，只不过安装起来相对比较复杂，并且泛虚拟化需要对 Guest OS 的代码进行修改，从而提升虚拟机的性能，因此对于内核代码不可修改的操作系统而言则不可用，如微软的 Windows 系列操作系统就不适用。Xen 主要用于服务器。

虽然 Xen 有很多的优点，但是最终没有(至少现在没有)被 Linux 内核所采用，而采用的是 KVM，原因之一可能是相比 KVM 而言 Xen 过于庞大与复杂了吧。或许也有其他的原因，而其答案可能只有 Linus Torvalds 和那些 Linux 内核版本管理人员知道了吧。

1.3.4 VirtualBox

VirtualBox 是 Sun 公司(Sun 公司在 2010 年被 Oracle 公司收购，Sun 公司虽然推出很多先进的技术以及优秀的软件产品，最终却落到被收购的地步，实在是太可惜了)的一款用于 x86 体系结构的虚拟化软件，可用于企业服务器、个人应用以及嵌入式产品中，可在多种操作系统如 Solaris、Linux、Windows 和 Macintosh 中使用，并且它是免费的。VirtualBox 支持大量的客户操作系统，其中包括 Windows 系列(NT 4.0，XP，Server 2003，Vista，Windows 7)、Linux、Solaris、OpenSolaris 和 OpenBSD 等[12]。

1.3.5 QEMU

QEMU 既是一款开源的模拟器，又是一款虚拟化软件，用来创建与运行虚拟机。QEMU 支持两种操作模式：用户模式和系统模拟模式。在用户模式， QEMU 可以用于交叉编译和交叉调试。在系统模拟模式中，QEMU 可以模拟出包括一个处理器和多个外部设备的完整的系统。

QEMU 是通过动态翻译的手段将待执行的代码翻译成目标代码，并且能够取得相对较好的性能，但是相比其他几款虚拟化软件而言性能就会差一点。QEMU 能够模拟包括 x86、ARM、PowerPC 和 Sparc 在内的一些列处理器架构[13]，它的模拟功能在软件开发、测试以及调试等阶段可以起到非常大的作用。

1.4 本文的结构与主要内容

本文一共分为 5 章，在第 1 章主要是对本文的研究背景以及几款主流的虚拟化软件进行了简单的介绍。第 2 章是对 Linux2.6.31.4 内核版本中关于 KVM 中的部分核心源代码进行分析与说明，这也是本文的重点之一。第 3 章介绍在 Ubuntu 操作系统下 KVM 的使用以及使用 KVM 创建虚拟机的方法。第 4 章是对前面介绍的几种虚拟化

软件的性能做了多个方面的分析与比较。第 5 章则是对本文的总结以及对将来工作的展望。

1.5　本章小结

本章首先介绍了本文的研究背景，然后对虚拟化技术以及几款主要的虚拟化软件做了简单的介绍，最后对本文研究的主要内容与本文的组织结构做了简要说明。

第 2 章　KVM 部分核心代码分析

2.1　代码分析方式介绍

在对 linux2.6.31 内核[14]中关于 kvm 的部分源代码进行分析之前，需要先对本文所分析代码的范围以及分析方式进行简要的说明。在此内核代码结构中，关于 kvm 的代码主要分与为体系结构无关的代码和与体系结构有关的代码。与体系结构无关的代码位于 virt/kvm 目录下，与体系结构相关的代码分布于各个体系结构的 kvm 目录下，此外还有一些相关的头文件分布在 include/asm 或 include/linux 目录下，以及一些相关的 c 文件分布在各个体系结构的 kernel 目录下等等。

由于篇幅、时间以及个人能力的关系，本文会对 kvm 中体系结构无关的部分代码和体系结构相关的部分代码进行分析，其中体系结构相关的代码主要分析 x86 体系结构相关的部分。对于 ia64、s390 和 powerpc 等体系结构(在 linux2.6.31 内核中这些体系结构的目录下也包含了 kvm 目录，也就是说这些体系结构也实现了 kvm)中相关的代码将不做分析。

对于代码的分析方式而言，本文不会对每个源文件中的代码逐行分析及解释。由于是通过动态加载模块的方式来使用 KVM 提供的功能的，所以分析的方式也就按照加载模块时所调用的函数顺序来进行分析，本文也会对在代码分析过程中所遇到的内核其他文件(非 kvm 目录中的文件)中实现的一些函数和宏定义进行分析与解释。

这里有一点需要说明的是在分析源代码时会将所涉及到的全部或部分源代码会以图的形式在本文中显示，并且会在源代码的第一行添加注释来标明代码在内核源代码中的位置。最后一点需要说明的是在本文中会看到有很多小括号(或者叫圆括号)，其中的内容主要是对其前面内容的说明、注释或补充。

2.2　核心代码分析

对 linux 2.6.x 内核而言，使用 Kbuild 来构建内核。Kbuild 是一个用于简化编写 Makefile 文件来完成复杂任务的 Makefile 框架，它的设计目标就是可以在所有支持的平台上有相似的行为、高度的灵活性以及使得 Makefile 的编写和维护更加的简单。在

后面的 Makefile 文件分析中将会看到内核中的 Makefile 文件与教科书中书写 Makefile 文件的不同。

在配置内核过程中，使用 make xconfig(或 make menuconfig、make gconfig 等命令，可以根据个人的爱好和机器中配置的不同使用不同的命令)命令时，将读取 Kconfig 文件，之后向用户显示配置选项，在用户配置完成之后，内核配置工具将自动的生成名为.config(注意，在 config 前面有个点，这样的点文件在 linux 操作系统下属于隐藏文件，使用 ls -a 命令可以查看)的内核配置文件。在编译内核时，根目录的 Makefile 文件会读取此.config 文件，根据此文件中各选项的值最终生成定制的内核镜像文件。

由于本文主要用来分析关于 x86 体系结构的 kvm 中的代码和与体系结构无关的代码，所以在分析具体的源代码之前，先对 arch/x86/kvm 目录下的 Kconfig 和 Makefile 文件进行分析，由此来对内核关于 kvm 的配置和编译结果有一定的了解，之后根据配置和编译的结果对部分代码进行详细的分析。

2.2.1　Kconfig 文件的分析

首先，对 Kconfig 文件进行分析，此 Kconfig 文件提供了关于虚拟化以及 KVM 的相关配置内容。它的内容如图 2.1 所示：

```
#
# KVM configuration
#
config HAVE_KVM
       bool
config HAVE_KVM_IRQCHIP
       bool
       default y
menuconfig VIRTUALIZATION
    bool "Virtualization"
    depends on HAVE_KVM || X86
      [......]
if VIRTUALIZATION
config KVM
    tristate "Kernel-based Virtual Machine (KVM) support"
    depends on HAVE_KVM
     [......]
config KVM_INTEL
    tristate "KVM for Intel processors support"
    depends on KVM
      [......]
config KVM_AMD
    tristate "KVM for AMD processors support"
    depends on KVM
    ---help---
      Provides support for KVM on AMD processors equipped with the AMD-V
      (SVM) extensions.
      [......]
config KVM_TRACE
    bool "KVM trace support"
      [......]
source drivers/lguest/Kconfig
source drivers/virtio/Kconfig
endif # VIRTUALIZATION
```

图 2.1　Kconfig 代码片段

上述代码中位于“#”后面同一行的文字属于注释性文字，表明此文件是 KVM 配置文件。后面的 config HAVE_KVM 定义一个 HAVE_KVM 配置选项，位于其下面的 bool 表示其取值只能是“y”或“n”。对于 HAVE_KVM_IRQCHIP 而言，其提供的默认值是“y”。menuconfig VIRTUALIZATION 与 configVIRTUALIZATION 类似，只不过是显示的方式有所不同，它要求子选项以独立的行显示。depends on 定义依赖关系，在此表示 VIRTUALIZATION 依赖于 HAVE_KVM 或 X86，如果依赖选项的取值为“n”的话此选项不可将，VIRTUALIZATION 默认值也是“y”，--help--下面是一些帮助信息用来帮助用户作出决定，在此省略(在此需要特殊说明的是在本文中对代码中省略的部分一律使用中括号里面包含省略号即[.....]的方式[15]表示)。if VIRTUALIZATION 到代码 endif 之间是一个条件块，如果 VIRTUALIZATION 选项的值被选为“n”，则说明不需要提供虚拟化功能，那么在 if 到 endif 之间的配置选项将不会显示，否则它

们将会作为 VIRTUALIZATION 的子项出现，tristate 表示此配置选项可以有三个值来选择，分别是“y”、“n”、“m”，其中“m”表示按模块的方式编译。select 是反向依赖，在 config KVM 选项中，如果 KVM 被置为“y”或“m”则 select 后面的配置选项将自动被设置。接下来的 config KVM_INTEL 和 config KVM_AMD 是对 Intel 与 AMD 处理器相关的配置项，其均依赖于 KVM。source 用于读取其后指定的 Kconfig 文件，在此处对指定的 Kconfig 文件进行解析，将 drivers/lguest/Kconfig 和 drivers/virtio/Kconfig 两个配置文件解析只不过是增加了一些在这两个配置文件中指定的配置选项，在此就不对这两个文件的内容进行详细的分析了。关于 Kconfig 的更多信息可以参考 linux 内核代码中 Documentation/kbuild 目录下关于 Kconfig 的文件 kconfig.txt。

在配置选项 KVM-AMD 中的 help 信息没有省略的原因是为了说明帮助信息的作用。在终端上运行命令，将当前目录转到 linux 内核源代码的根目录，运行 make xconfig 命令，之后将出现如图 2.2 所示的配置页面，鼠标点中“KVM for AMD processors support”选项，在界面的最下方将显示帮助信息，其内容和 Kconfig 文件中 KVM_AMD 选项的帮助信息相同。

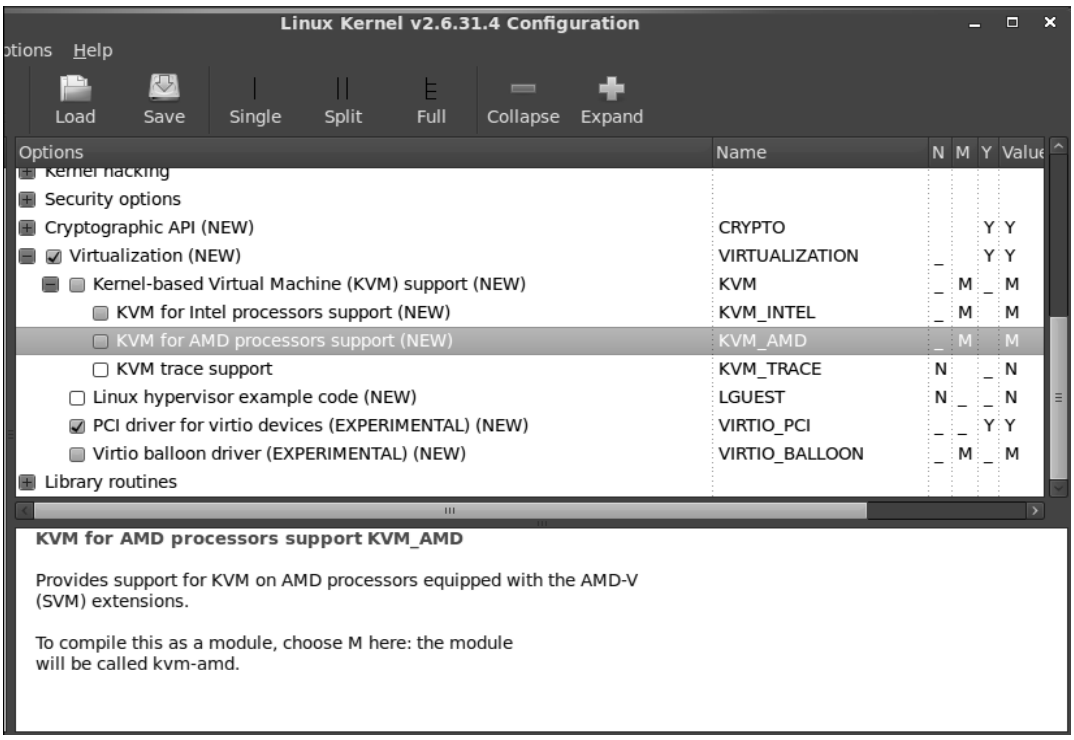

图 2.2　Linux 内核配置

2.2.2　Makefile 文件的分析

分析完 Kconfig 之后，接下来对 Makefile 文件进行分析，此文件的内容如图 2.3 所示：

```
---------------------------------------------------------------------------
# /arch/x86/kvm
    #
    # Makefile for Kernel-based Virtual Machine module
    #

    common-objs = $(addprefix ../../../virt/kvm/, kvm_main.o ioapic.o \
                coalesced_mmio.o irq_comm.o)
    ifeq ($(CONFIG_KVM_TRACE),y)
    common-objs += $(addprefix ../../../virt/kvm/, kvm_trace.o)
    endif
    ifeq ($(CONFIG_IOMMU_API),y)
    common-objs += $(addprefix ../../../virt/kvm/, iommu.o)
    endif

    EXTRA_CFLAGS += -Ivirt/kvm -Iarch/x86/kvm

    kvm-objs := $(common-objs) x86.o mmu.o x86_emulate.o i8259.o irq.o lapic.o \
        i8254.o timer.o
    obj-$(CONFIG_KVM) += kvm.o
    kvm-intel-objs = vmx.o
    obj-$(CONFIG_KVM_INTEL) += kvm-intel.o
    kvm-amd-objs = svm.o
    obj-$(CONFIG_KVM_AMD) += kvm-amd.o
---------------------------------------------------------------------------
```

图 2.3　Makefile 代码片段

在这个 Makefile 文件中没有出现一般的教科书中所介绍的 make 规则中使用的一些如进行编译、链接等工作的命令，实际这些工作是由 Kbuild 来做了，从而减轻了内核程序员的工作量以及简化了 Makeifile 的书写。

接下来介绍在这个 Makefile 文件内容的各个部分，上述代码片段中位于“#”后面的文字是注释，来说明此文件的作用。common-objs 则是指位于根目录下/virt/kvm 目录下面对应的 c 文件编译后的一些目标文件。后面的两个判断语句分别用来判断 KVM_TRACE 和 IOMMU_API 在 .config 文件中的值是否为 Y，如果是则在 common-objs 后面追加上面列出的目标文件。EXTRA_CFLAGS 新增加两个寻找路径，即 virt/kvm 和 arch/x86/kvm，其作用是告诉 Kbuild 关于构建后面的模块所需要的头文件与 c 源文件的位置(相对于内核代码树形结构根目录)。

在构建内核过程中 KVM,KVM_INTEL 以及 KVM_AMD 一般会设置为 m(当然这不是必须的，也可以选择 y 或 n)，如图 2.2 所示，如果选择了 m，也就是以模块的形

式编译，最后将生成 kvm.ko，kvm-intel.ko 和 kvm-amd.ko 模块文件。其中 kvm.ko 文件是由跟 kvm-objs 指定的这些目标文件的同名的 c 文件经过编译和链接后生成的。同样地，kvm-intel.ko 文件是由 vmx.c 编译后的 vmx.o 文件生成。kvm-amd.ko 文件是由 svm.c 文件编译后的 svm.o 文件生成。

2.2.3　KVM 部分代码分析

分析完 Kconfig 和 Makefile 文件之后，接下来要将按照前面介绍的分析方式对 kvm 的部分实现代码进行分析。由前面的介绍可知，在配置内核时如果对 KVM 相关的配置项均选择 m 以模块的形式编译，那么 Kbuild 将根据 arch/x86/kvm/Makefile 生成三个模块，分别为 kvm.ko，kvm-amd.ko 和 kvm-intel.ko，接下来将分别介绍与分析这三个模块。

2.2.3.1　kvm.ko 模块分析

kvm.ko 模块是体系结构无关的模块。对于此模块而言，它并没有注册入口函数，在加载这个模块[16]时，其实它只是导出通过宏定义 EXPORT_SYMBOL_GPL()(在 kvm_main.c 文件中可以看到很多关于这个宏定义的使用)导出了一些符号(也就是函数名)供其他的模块，即 kvm-intel.ko 模块或 kvm-amd.ko 模块使用。当模块加载完成后，在命令行中输入 modinfo kvm-intel 或 modinfo kvm-amd 后都会在输出信息中有一行 depends:　kvm 信息，表明 kvm-intel.ko 模块和 kvm-amd.ko 模块都依赖于 kvm.ko 模块。

宏定义 EXPORT_SYMBOL_GPL()的作用是将符号导出，即放入内核符号表中以供其他的模块使用，但是由它导出的符号只能由遵循 GPL 协议(或称为 GPL 许可，为了保持一致，在本文后面都使用 GPL 协议来表示)的模块使用。如果要想使到出的符号供所有的模块使用，也就是说不遵循 GPL 协议的模块也可以用，则可以使用 EXPORT_SYMBOL()宏定义。

那么如何判定一个模块是否遵循了 GPL 协议呢？估计看过模块编程的书籍或者阅读过 linux 内核代码的读者都会经常见到 MODULE_AUTHOR() 和 MODULE_LICENSE()这两个宏定义。如果有 MODULE_LICENSE("GPL")，则说明此

模块是遵循 GPL 协议的。比如在命令行中输入 modinfo kvm，则在输出中有 license: GPL 和 author: Qumranet 这样的信息来表明此模块的 license 以及 author。

由 Makefile 可以看出 kvm.ko 模块是由 12 到 14 个源文件编译链接后生成的，之所以源文件数不定是因为用户配置内核时有两个(即 kvm_trace.c 和 iommu.c)是根据用户的选择来决定是否加入此模块中。在 kvm_main.c 源文件中通过宏定义 EXPORT_SYMBOL_GPL()导出的有 30 个符号供外部代码使用。导出的这些符号如图 2.4 所示：

```
/* virt/kvm/kvm_main.c */
    kvm_vcpu_cache,
    kvm_vcpu_init,
    kvm_vcpu_uninit,
    kvm_get_kvm,
    kvm_put_kvm,
    __kvm_set_memory_region,
    kvm_set_memory_region,
    is_error_page,
    is_error_pfn,
    kvm_is_error_hva,
    gfn_to_memslot_unaliased,
    kvm_is_visible_gfn,gfn_to_pfn,
    gfn_to_page,
    kvm_release_page_clean,
    kvm_release_pgn_clean,
    kvm_release_page_dirty,
    kvm_release_pfn_dirty,
    kvm_set_page_dirty,
    kvm_set_pfn_dirty,
    kvm_set_pfn_accessed,
    kvm_get_pfn,
    kvm_read_guest_page,
    kvm_read_guest,
    kvm_write_guest_page,
    kvm_clear_guest_page,
    kvm_clear_guest,
    kvm_resched,
    kvm_handle_fault_on_reboot,
    kvm_init,
    kvm_exit.
```

图 2.4　kvm_main.c 导出符号图示例

上面列出 kvm.ko 模块导出的这些符号的目的是先通过这些符号名称对此模块提供的功能有一个初步和整体的认识。在这里并不对此模块导出的符号也就是在 kvm_main.c 文件中的相关的函数的实现做介绍，在后面其他的模块中用到时将会做具体的分析。

2.2.3.2　kvm-amd.ko 模块分析

接下来要介绍的是 kvm-amd.ko 模块，当使用命令 modprobe kvm-amd(使用 insmod 命令也可以加载模块，modprobe 和 insmod 命令提供的功能是有区别的，最主要的区别是 insmod 命令只能加载指定的模块，而 modprobe 可以在加载指定模块的同时也将加载模块依赖的模块加载。但是它们的之间的具体区别在此不做介绍，它们的使用以及区别可以参考介绍 linux 模块编程的书。本文中介绍到加载模块时，使用的命令统一是 modprobe，在此声明，此后遇到相同情况不再说明)加载此模块时，将调用在宏定义 module_init()(宏定义 module_init()在模块的目标代码中增加特殊的段来表明在哪里可以找到模块的入口函数，入口函数有时也被称为初始化函数)中注册的函数 svm_init()作为这个模块入口的函数，此函数位于 svm.c 文件中，在函数 svm_init()中调用函数 kvm_init()，svm_init()的源代码如图 2.5 所示：

```
/* arch/x86/kvm/svm.c */
    static int __init svm_init(void)
    {
        return kvm_init(&svm_x86_ops, sizeof(struct vcpu_svm),
                        THIS_MODULE);
    }
```

图 2.5　svm_init()函数代码

在 svm_init()函数的定义中，用到了 static 关键字[17]，static 在 c++或 java 等面向对象编程语言中用来修饰类成员，被修饰的成员被称为静态变量或静态方法，而在处，它的主要作用是用来修饰函数或变量，用于限制使所修饰的成员的访问范围，使其只能在函数或变量所定义的文件内部访问，而且被 static 修饰的成员也不能使用宏定义 EXPORT_SYMBOL_GPL()或 EXPORT_SYMBOL()来导出供其他模块使用。函数定义中的__init 标志(实际上__init 是一个宏定义，它的定义在 init.h 头文件中)是用来告诉编译器此函数只是在初始化时使用，模块加载完成后将不会使用这个函数，编译器会将标识了__init 标志的代码放置在一个特殊的内存块中，初始化结束后将会把这个代码所占用的资源释放掉。

在调用 kvm_init()函数过程中，传入三个实参，第一个是类型为 kvm_x86_ops 的结构体变量 svm_x86_ops 的地址，结构体 kvm_x86_ops 定义在 kvm_host.h 头文件中，

它被包含在了 svm.c 文件中。在 kvm_host.h 这个头文件中定义了关于 x86 体系结构的 kvm 的相关接口，结构体 kvm_x86_ops 的成员是一些函数指针变量，这些指针变量在 svm.c 中被赋值，也就是在定义并初始化结构体变量 svm_x86_ops 时完成的，svm_x86_ops 的定义及初始化代码片段(由于篇幅的关系，省略号部分省略了部分代码)如图 2.6 所示：

```
/* arch/x86/kvm/svm.c */
    static struct kvm_x86_ops svm_x86_ops = {
        .cpu_has_kvm_support = has_svm,
        .disabled_by_bios = is_disabled,
        .hardware_setup = svm_hardware_setup,
        .hardware_unsetup = svm_hardware_unsetup,
        .check_processor_compatibility = svm_check_processor_compat,
        .hardware_enable = svm_hardware_enable,
    [......]
        .vcpu_create = svm_create_vcpu,
        .vcpu_free = svm_free_vcpu,
        .vcpu_reset = svm_vcpu_reset,
    [......]
    };
```

图 2.6　svm_x86_ops 定义代码片段

第二个参数是通过 sizeof 操作符计算的结构体 vcpu_svm 的大小，结构体 vcpu_svm 定义在 kvm_svm.h 头文件中。第三个参数 THIS_MOUDLE 是一个宏定义，它定义在在 module.h 头文件中，其定义的代码片段如图 2.7 所示：

```
/* include/linux/module.h */
    #ifdef MODULE
    [......]
    #define     THIS_MODULE (&__this_module)
    #else
    [......]
    #define THIS_MODULE ((struct module *)0)
    #endif
```

图 2.7　宏定义 THIS_MODULE 代码片段

__this_module 是一个 struct module 变量，代表当前模块，可以通过 THIS_MODULE 宏来初始化模块的 struct module 结构体变量。在 kvm_init()函数将会看到对此结构体变量的使用。

接下来就开始分析 kvm_init()函数，此函数是本文重点分析的函数之一，位于 kvm_main.c 文件中，由于 kvm_init()函数代码较长，因此将其分为两部分，它的代码

如图 2.8 和 2.9 所示：

```
/* virt/kvm/kvm_main.c */
/*kvm_init() part one*/
    int kvm_init(void *opaque, unsigned int vcpu_size,
              struct module *module)
    {
        int r;
        int cpu;
        kvm_init_debug();
        r = kvm_arch_init(opaque);
        if (r)
            goto out_fail;
        bad_page = alloc_page(GFP_KERNEL | __GFP_ZERO);
        if (bad_page == NULL) {
            r = -ENOMEM;
            goto out;
        }
        bad_pfn = page_to_pfn(bad_page);
        if (!zalloc_cpumask_var(&cpus_hardware_enabled, GFP_KERNEL)) {
            r = -ENOMEM;
            goto out_free_0;
        }
        r = kvm_arch_hardware_setup();
        if (r < 0)
            goto out_free_0a;
        for_each_online_cpu(cpu) {
            smp_call_function_single(cpu,
                    kvm_arch_check_processor_compat,
                    &r, 1);
            if (r < 0)
                goto out_free_1;
        }
        on_each_cpu(hardware_enable, NULL, 1);
        r = register_cpu_notifier(&kvm_cpu_notifier);
        if (r)
            goto out_free_2;
        register_reboot_notifier(&kvm_reboot_notifier);
        r = sysdev_class_register(&kvm_sysdev_class);
        if (r)
            goto out_free_3;
        r = sysdev_register(&kvm_sysdev);
        if (r)
            goto out_free_4;
```

图 2.8　kvm_init()函数代码片段 1

```
    /* A kmem cache lets us meet the alignment requirements of fx_save. */
            kvm_vcpu_cache = kmem_cache_create("kvm_vcpu", vcpu_size,
                                       __alignof__(struct kvm_vcpu),
                                       0, NULL);
            if (!kvm_vcpu_cache) {
                r = -ENOMEM;
                goto out_free_5;
            }
            kvm_chardev_ops.owner = module;
            kvm_vm_fops.owner = module;
            kvm_vcpu_fops.owner = module;
            r = misc_register(&kvm_dev);
            if (r) {
                printk(KERN_ERR "kvm: misc device register failed\n");
                goto out_free;
            }
            kvm_preempt_ops.sched_in = kvm_sched_in;
            kvm_preempt_ops.sched_out = kvm_sched_out;
            return 0;

    out_free:
        kmem_cache_destroy(kvm_vcpu_cache);
    out_free_5:
        sysdev_unregister(&kvm_sysdev);
    out_free_4:
        sysdev_class_unregister(&kvm_sysdev_class);
    out_free_3:
        unregister_reboot_notifier(&kvm_reboot_notifier);
        unregister_cpu_notifier(&kvm_cpu_notifier);
    out_free_2:
        on_each_cpu(hardware_disable, NULL, 1);
    out_free_1:
        kvm_arch_hardware_unsetup();
    out_free_0a:
        free_cpumask_var(cpus_hardware_enabled);
    out_free_0:
        __free_page(bad_page);
    out:
        kvm_arch_exit();
        kvm_exit_debug();
    out_fail:
        return r;
    }
```

图 2.9　kvm_init()函数代码片段 2

在 kvm_init()函数中，首先调用的是 kvm_init_debug()函数，其代码如图 2.10 所示：

```
/* virt/kvm/kvm_main.c */
    static void kvm_init_debug(void)
    {
        struct kvm_stats_debugfs_item *p;

        kvm_debugfs_dir = debugfs_create_dir("kvm", NULL);
        for (p = debugfs_entries; p->name; ++p)
            p->dentry = debugfs_create_file(p->name, 0444, kvm_debugfs_dir,
                            (void *)(long)p->offset,
                            stat_fops[p->kind]);
    }
```

图 2.10　kvm_init_debug()函数代码

在 kvm_init_debug()中调用 debugfs_create_dir()在 debugfs 文件系统(用来提供调试信息的虚拟文件系统[18])中，即在/sys/kernel/debug 目录下创建名为“kvm”的子目录，之后在 for 循环中调用 debugfs_create_file()函数，其作用是在 debugfs 文件系统中创建一个文件，所创建的文件的名字、访问权限、父路径等特性由调用方在调用函数时传递的实参来指定。由于篇幅的关系，在此不对 debugfs_create_dir()和 debugfs_create_file()的具体实现代码进行分析。

由此可见 kvm_init_debug()函数在 debugfs 虚拟文件系统根目录下创建了名为“kvm”目录，之后在这个目录下创建了一些只读(由 debugfs_create_file()中的参数 0444 可知)文件用于提供调试信息。

回到 kvm_init()函数，接下来调用的函数是 kvm_arch_init()，对 KVM 体系结构相关的部分进行初始化操作，函数 kvm_arch_init()的代码如图 4.11 所示：

```
/* arch/x86/kvm/x86.c */
    int kvm_arch_init(void *opaque)
    {
        int r, cpu;
        struct kvm_x86_ops *ops = (struct kvm_x86_ops *)opaque;
        if (kvm_x86_ops) {
            printk(KERN_ERR "kvm: already loaded the other module\n");
            r = -EEXIST;
            goto out;
        }

        if (!ops->cpu_has_kvm_support()) {
            printk(KERN_ERR "kvm: no hardware support\n");
            r = -EOPNOTSUPP;
            goto out;
        }
        if (ops->disabled_by_bios()) {
            printk(KERN_ERR "kvm: disabled by bios\n");
            r = -EOPNOTSUPP;
            goto out;
        }

        r = kvm_mmu_module_init();
        if (r)
            goto out;

        kvm_init_msr_list();

        kvm_x86_ops = ops;
        kvm_mmu_set_nonpresent_ptes(0ull, 0ull);
        kvm_mmu_set_base_ptes(PT_PRESENT_MASK);
        kvm_mmu_set_mask_ptes(PT_USER_MASK, PT_ACCESSED_MASK,
                PT_DIRTY_MASK, PT64_NX_MASK, 0);
        for_each_possible_cpu(cpu)
            per_cpu(cpu_tsc_khz, cpu) = tsc_khz;
        if (!boot_cpu_has(X86_FEATURE_CONSTANT_TSC)) {
            tsc_khz_ref = tsc_khz;
            cpufreq_register_notifier(&kvmclock_cpufreq_notifier_block,
                        CPUFREQ_TRANSITION_NOTIFIER);
        }
        return 0;
    out:
        return r;
    }
```

图 2.11　kvm_arch_init()函数代码

在 kvm_arch_init()中，首先将传递过来的参数类型转换为 struct kvm_x86_ops 的指针类型，之后判断全局变量 kvm_x86_ops 是否已经非空，如果是则通过 printk()函数(printk()函数是内核代码中的标准输出函数，类似于 c 语言类库中提供的 printf()函数，在这里不对 printk()函数做详细的介绍)打印错误信息，然后跳转到 out 标签所在的位置，执行其后的代码，也就是直接返回。

如果不为空，则通过 ops->has_kvm_support()(由结构体 kvm_x86_ops 的定义和在 kvm_init()中传递的参数&svm_x86_ops 的可知，实际是调用位于 svm.c 中的 has_svm()

函数)来判断硬件是否支持 kvm，即查看 CPU 是否支持 SVM，如果不支持则跳转到 out 标签执行。之后由 ops->disabled_by_bios()(调用 svm.c 中的 is_disabled())来判断 kvm 是否被 bios 禁止，如果是就跳转到 out 处执行。

接下来调用函数 kvm_mmu_module_init()来进行内存模块的初始化，其代码如图 2.12 所示：

```
/* arch/x86/kvm/mmu.c */
    int kvm_mmu_module_init(void)
    {
        pte_chain_cache = kmem_cache_create("kvm_pte_chain",
                          sizeof(struct kvm_pte_chain),
                          0, 0, NULL);
        if (!pte_chain_cache)
            goto nomem;
        rmap_desc_cache = kmem_cache_create("kvm_rmap_desc",
                          sizeof(struct kvm_rmap_desc),
                          0, 0, NULL);
        if (!rmap_desc_cache)
            goto nomem;

        mmu_page_header_cache = kmem_cache_create("kvm_mmu_page_header",
                            sizeof(struct kvm_mmu_page),
                            0, 0, NULL);
        if (!mmu_page_header_cache)
            goto nomem;

        register_shrinker(&mmu_shrinker);

        return 0;

    nomem:
        mmu_destroy_caches();
        return -ENOMEM;
    }
```

图 2.12　kvm_mmu_module_init()函数代码

由代码可以看出，此函数就是调用 kmem_cache_create()创建一个新的缓存，所创建的高速缓存的名字为“kvm_pte_chain”，高速缓存中每个元素的大小为 sizeof(struct kvm_pte_chain)，第三个参数是指定对象的对齐方式，此处为 0，表示标准对齐。第四个参数是用来控制高速缓存的行为，此处也是 0 表示没有特殊的行为，最后一个参数是高速缓存的构造函数，此处为 NULL。kmem_chche_create()函数调用成功的话会返回一个指向新创建的高速缓存的指针，否则返回 NULL。此函数的具体实现细节在此就不做具体分析了。

变量 pte_chain_cache、rmap_desc_cache 和 mmu_page_header_cache 是定义在 mmc.c 文件中的全局变量，来保存 kmem_cache_create()的返回值，它们的类型都是 struct kmem_cache *，结构体 kmem_cache 的作用就是管理一个 cache。通过向函数

register_shrinker()传递一个 struct shrinker 变量的地址来注册一个页面回收函数 mmu_shink()，将来被 vm 调用。如果在创建缓存过程中有失败，则程序会转到 nomem 标签，并通过 mmu_destory_caches()将已经创建的 cache 释放掉。

在 kvm_mmu_module_init()返回后，接着调用 kvm_init_msr_list()，从函数名可知，此函数是用来初始化 msr_list 的，其代码如图 2.13 所示：

```
/* arch/x86/kvm/x86.c */
    static void kvm_init_msr_list(void)
    {
        u32 dummy[2];
        unsigned i, j;

        for (i = j = 0; i < ARRAY_SIZE(msrs_to_save); i++) {
            if (rdmsr_safe(msrs_to_save[i], &dummy[0], &dummy[1]) < 0)
                continue;
            if (j < i)
                msrs_to_save[j] = msrs_to_save[i];
            j++;
        }
        num_msrs_to_save = j;
    }
```

图 2.13　kvm_init_msr_list()函数代码

变量 msrs_to_save 是一个 u32 类型的数组，定义于 x86.c 中，其中的值是 msr 号，这些值通过用户空间的程序利用 ioctl 系统调用来获取或设置。函数 rdmsr_save()是将 msrs_to_save[i]中的值的低 32 位赋予 dummy[0]，高 32 位赋予 dummy[1]。从上面的代码中可以看出，如果某此 rdmsr_save()返回值小于 0，则会使 j 的值小于 i，那么将会使那次执行 rdmsr_save()所对应的 msrs_to_save[i]的值被后面此函数执行成功的 msrs_to_save[i]的值覆盖。

返回到 kvm_arch_init() 函数，接下来调用的三个函数 kvm_mmu_set_nonpresent_ptes()，kvm_mmu_set_base_ptes() 和 kvm_mmu_set_mask_ptes()是对页表项的初始化设置，在此不做详细说明。

在 kvm_arch_init()函数返回后，首先会判断其返回值是否为 0，如果不是，则跳转到 goto 语句后面标签 out_fail 所在的位置执行，即直接返回。在 kvm_arch_init()函数成功返回后，接着将调用 alloc_page 函数分配一页内存并将其地址赋予 bad_page，它是一个全局的类型为 struct page 指针。然后判断 bad_page 是否为空，如果是说明调用 alloc_page 分配页时失败，则会跳转到 out 标签所在位置继续执行。

接着通过宏 page_to_pfn()得到这个刚分配的页在页表中的位置，然后调用 zalloc_cpumask_var()(zalloc_cpumask_var 定义在 cpumask.h 中，它是调用 cpumask_clear()来实现其功能的)函数来将一个全局 cpumask 结构体变量 cpu_hardware_enabled 进行初始化，实际就是将其中的成员置 0。如果调用失败，则会设置返回值的错误类型，之后跳转到指定位置执行。

接下来在 kvm_init()函数中调用的是 kvm_arch_hardware_setup()，这个函数的代码如图 2.14 所示：

```
/* arch/x86/kvm/x86.c */
    int kvm_arch_hardware_setup(void)
    {
        return kvm_x86_ops->hardware_setup();
    }
```

图 2.14　kvm_arch_hardware_setup()代码

由前面 kvm_x86_ops 的定义可知，kvm_x86_ops->hardware_setup()实际上调用的是定义在 svm.c 中的 svm_hardware_setup()函数对硬件进行设置，其代码如 2.15 所示：

```
/* arch/x86/kvm/svm.c */
    static __init int svm_hardware_setup(void)
    {
        int cpu;
        struct page *iopm_pages;
        void *iopm_va;
        int r;
        iopm_pages = alloc_pages(GFP_KERNEL, IOPM_ALLOC_ORDER);
        if (!iopm_pages)
            return -ENOMEM;
        iopm_va = page_address(iopm_pages);
        memset(iopm_va, 0xff, PAGE_SIZE * (1 << IOPM_ALLOC_ORDER));
        iopm_base = page_to_pfn(iopm_pages) << PAGE_SHIFT;
        if (boot_cpu_has(X86_FEATURE_NX))
            kvm_enable_efer_bits(EFER_NX);
        if (boot_cpu_has(X86_FEATURE_FXSR_OPT))
            kvm_enable_efer_bits(EFER_FFXSR);
        if (nested) {
            printk(KERN_INFO "kvm: Nested Virtualization enabled\n");
            kvm_enable_efer_bits(EFER_SVME);
        }
        for_each_online_cpu(cpu) {
            r = svm_cpu_init(cpu);
            if (r)
                goto err;
        }
        svm_features = cpuid_edx(SVM_CPUID_FUNC);
        if (!svm_has(SVM_FEATURE_NPT))
            npt_enabled = false;
        if (npt_enabled && !npt) {
            printk(KERN_INFO "kvm: Nested Paging disabled\n");
            npt_enabled = false;
        }
        if (npt_enabled) {
            printk(KERN_INFO "kvm: Nested Paging enabled\n");
            kvm_enable_tdp();
        } else
            kvm_disable_tdp();
        return 0;
    err:
        __free_pages(iopm_pages, IOPM_ALLOC_ORDER);
        iopm_base = 0;
        return r;
    }
```

图 2.15　svm_hardware_setup()函数代码

函数 alloc_page()分配一页内存并将返回值赋予 iopm_pages，函数 page_address()的作用简单的说就是返回该 iopm_pages 对应的线性地址，memset()将此块地址填充指定的值，page_to_pfn()在前面也介绍了，此处将不再重复介绍。

宏定义 boot_cpu_has()是用来测试 CPU 是否具备某种特性，而在此与之配合使用的宏定义 kvm_enable_efer_bits()则是在 boot_cpu_has()返回为 1 的情况下，对传递给自己的参数的值的特定位进行清零操作。

for_each_online_cpu()表示循环每一个 online 的 cpu，它其实是一个 for 循环，在

循环中调用函数 svm_cpu_init()对 cpu 进行初始化，其代码如图 2.16 所示：

```
/* arch/x86/kvm/svm.c */
    static int svm_cpu_init(int cpu)
    {
        struct svm_cpu_data *svm_data;
        int r;

        svm_data = kzalloc(sizeof(struct svm_cpu_data), GFP_KERNEL);
        if (!svm_data)
            return -ENOMEM;
        svm_data->cpu = cpu;
        svm_data->save_area = alloc_page(GFP_KERNEL);
        r = -ENOMEM;
        if (!svm_data->save_area)
            goto err_1;

        per_cpu(svm_data, cpu) = svm_data;

        return 0;

    err_1:
        kfree(svm_data);
        return r;

    }
```

图 2.16　svm_cpu_init()函数代码

在函数的内部首先调用 kzalloc()函数为结构体 svm_cpu_data 分配内存空间，之后将其成员变量 cpu 和 save_area(alloc_page()函数的返回值)进行初始化，如果调用 alloc_page()失败，即 svm_data->save_area 为空，则跳转到 err_1 标签所在位置，执行后面的 kfree()释放分配给 svm_data 所指的空间之后返回错误值(-ENOMEM)，表示没有总够的可用空间。宏定义 per_cpu()用来访问其他 cpu 的变量的副本，并将在循环中创建的 svm_data 的地址赋给各 cpu 中相应的变量。因此在循环结束后各个 cpu 在内存中都有自己的空间用于存放结构体 svm_cpu_data。

分析完 svm_cpu_init()后，回到 svm_hardware_setup()，在调用 svm_cpu__init()过程中如果返回非 0 值，则跳转到 err 标签所在位置，执行后面的语句，即释放分配的内存页。否则，接下来 svm_hardware_setup()会调用 cpuid_edx()函数，用于返回 edx 寄存器的值。函数 has_svm()用来判断 CPU 是否支持 SVM，如果支持则返回 1，否则返回 0。接下来将会根据 npt_enabled 的值来调用 kvm_enable_tdp()函数或 kvm_disable_tdp()函数，这两个函数都很简单，前者是将变量 tdp_enabled 设为 true，后者将 tdp_enabled 设为 false。最后，函数 kvm_arch_hardware_setup()执行完毕，返

回 0。

在 kvm_arch_hardware_setup() 返回后，接下来 kvm_init() 通过宏 for_each_online_cpu(其实宏展开后就是一个 for 循环)来对每一个 online cpu 调用 smp_call_function_single()，这个函数的作用就是对于给定的 cpu，运行传给它的函数，在这个特定的上下文环境中，则是在这个函数体中将调用传给它的函数指针所对应的函数，也就是 kvm_arch_check_processor_compat()函数。

在 kvm_arch_check_processor_compat() 函数中调用 svm_check_processor_compat()，而这个函数的代码非常简单，就是将传递给他的参数进行类型转换后将其所指地址的值赋值为 0。代码如图 2.17 所示：

```
/* arch/x86/kvm/svm.c */
    static void svm_check_processor_compat(void *rtn)
    {
        *(int *)rtn = 0;
    }
```

图 2.17　svm_check_processor_compat()函数代码

在 svm_check_processor_compat()中 rtn 对应于调用 smp_call_function_single()时的 r，因此在 smp_call_function_single()返回后 r 的值被赋为 0。

对于 on_each_cpu()而言，它有两个定义，一个是位于 smp.h 文件中，是一个宏定义；而另一个定义位于 softirq.c 中，是一个函数，在此处会根据是否支持 SMP 来选择使用其中的一个。on_each_cpu()的作用就是对所用的处理器调用指定的函数，在此处调用的是位于 kvm_main.c 中的 hardware_enable()函数，即开启虚拟化功能。其代码如图 2.18 所示：

```
/* virt/kvm/kvm_main.c */
    static void hardware_enable(void *junk)
    {
        int cpu = raw_smp_processor_id();

        if (cpumask_test_cpu(cpu, cpus_hardware_enabled))
            return;
        cpumask_set_cpu(cpu, cpus_hardware_enabled);
        kvm_arch_hardware_enable(NULL);
    }
```

图 2.18　hardware_enable()函数代码

在函数 hardware_enable()中首先调用的是 raw_smp_processor_id()，其作用是返回当前 cpu 号。接下来由 cpumask_test_cpu() 来判断此 cpu 值对应的 cpus_hardware_enabled 位是否设置，如果设置，则返回，否则由 cpumask_set_cpu()函数设置此位。最后调用 kvm_arch_hardware_enable()函数，实际会根据处理器的不同调用不同的函数，对 amd.ko 模块而言，调用的是 svm.c 中的 svm_hardware_enable()函数，其代码如图 2.19 所示：

```
/* arch/x86/kvm/svm.c */
    static void svm_hardware_enable(void *garbage)
    {

        struct svm_cpu_data *svm_data;
        uint64_t efer;
        struct desc_ptr gdt_descr;
        struct desc_struct *gdt;
        int me = raw_smp_processor_id();

        if (!has_svm()) {
            printk(KERN_ERR "svm_cpu_init: err EOPNOTSUPP on %d\n", me);
            return;
        }
        svm_data = per_cpu(svm_data, me);

        if (!svm_data) {
            printk(KERN_ERR "svm_cpu_init: svm_data is NULL on %d\n",
                   me);
            return;
        }

        svm_data->asid_generation = 1;
        svm_data->max_asid = cpuid_ebx(SVM_CPUID_FUNC) - 1;
        svm_data->next_asid = svm_data->max_asid + 1;

        asm volatile ("sgdt %0" : "=m"(gdt_descr));
        gdt = (struct desc_struct *)gdt_descr.address;
        svm_data->tss_desc = (struct kvm_ldttss_desc *)(gdt + GDT_ENTRY_TSS);

        rdmsrl(MSR_EFER, efer);
        wrmsrl(MSR_EFER, efer | EFER_SVME);

        wrmsrl(MSR_VM_HSAVE_PA,
               page_to_pfn(svm_data->save_area) << PAGE_SHIFT);
    }
```

图 2.19　svm_hardware_enable()函数代码

在这个函数中，使用到的函数和宏定义，其中有些(raw_smp_processor_id()，has_svm()，per_cpu()，cpuid_ebx()，page_to_pfn()等)已经在前面介绍过了，因此在这里只介绍前面没有遇到过的函数或宏定义。

asm volatile()这并不是一个函数，它是 c 语言中嵌入汇编的书写格式，在括号中可以有汇编指令，输入部分，输出部分等。volatile 是一个关键字，并不是必须的组

成部分，它的作用是为了防止 gcc[19]在代码优化时将这条 asm 语句作为 unused 删除掉。在 svm_hardware_enable()函数中，asm volatile ("sgdt %0" : "=m"(gdt_descr))的作用就是读取 gdt 的地址，并将其赋给变量 gdt_descr。

宏定义 rdmsrl()和 wrmsrl()的作用就是读写机器特定的寄存器的值，rdmsrl()在此的作用是将 MSR_EFER 的值赋给变量 efer，它实际是通过调用函数 native_read_msr()实现的，这个函数的具体实现在此不做说明。wrmsrl()在此的作用就是修改 MSR_EFER 和 MSR_VM_HSAVE_PA 的值，它是通过调用 native_write_msr 函数实现的，在此也不对此函数做具体介绍。

介绍完 svm_hardware_enable()函数，回到 kvm_init()函数，接下来看 register_cpu_notifier()函数，它也和 SMP 有关，在内核代码 cpu.h 中，如果没有定义 CONFIG_SMP 或者定义了 CONFIG_SMP 并且定义了 MODULE，则 register_cpu_notifier()函数直接返回 0，否则它的实现代码如图 2.20 所示：

```
/* kernel/cpu.c */
    int __ref  register_cpu_notifier(struct notifier_block *nb)
    {
        int ret;
        cpu_maps_update_begin();
        ret = raw_notifier_chain_register(&cpu_chain, nb);
        cpu_maps_update_done();
        return ret;
    }
```

图 2.20　register_cpu_notifier()函数代码

其中，函数 cpu_maps_update_begin()和函数 cpu_maps_update_done()的作用分别是获得和释放互斥锁 cpu_add_remove_lock。函数 raw_notifier_chain_register()最终则是通过调用 notifier_chain_register()函数(在 raw_notifier_regirster()函数中调用，此函数用来在一个特定的通知链中加入一个 notifier_block 对象)将 nb 插入到 cpu_chain 链中合适的位置(根据 nb 中的成员 priority 的值决定)。

在 cpu 的热插拔中会使用通知链，函数 register_cpu_notifier()的作用就是在通知链 cpu_chain 中添加新的节点。在注册新节点时，会在这个新节点中的 notifier_call 域指定一个回调函数，通过函数 raw_notifier_call_chain()调用。 在函数 kvm_init()中调用 register_cpu_notifier()时传递的参数是 kvm_cpu_notifier 的地址，kvm_cpu_notifier 结构体变量的 notifier_call 域的值是其指定的回调函数为 kvm_cpu_hotplug，即回调函

数的名字。函数 kvm_cpu_hotplug 的代码如图 2.21 所示：

```
/* vrit/kvm/kvm_main.c */
    static int kvm_cpu_hotplug(struct notifier_block *notifier, unsigned long val,
                   void *v)
    {
        int cpu = (long)v;

        val &= ~CPU_TASKS_FROZEN;
        switch (val) {
        case CPU_DYING:
            printk(KERN_INFO "kvm: disabling virtualization on CPU%d\n",
                   cpu);
            hardware_disable(NULL);
            break;
        case CPU_UP_CANCELED:
            printk(KERN_INFO "kvm: disabling virtualization on CPU%d\n",
                   cpu);
            smp_call_function_single(cpu, hardware_disable, NULL, 1);
            break;
        case CPU_ONLINE:
            printk(KERN_INFO "kvm: enabling virtualization on CPU%d\n",
                   cpu);
            smp_call_function_single(cpu, hardware_enable, NULL, 1);
            break;
        }
        return NOTIFY_OK;
    }
```

图 2.21　kvm_cpu_hotplug()函数代码

由上述代码可以看出，根据变量 val 的值来执行相应的操作，即如果 val 的值为 CPU_DYING 或者 CPU_UP_CANCELED 则会调用 hardware_disable()函数，实际调用的函数是 cpu_svm_disable()，其作用是在特定 cpu 上禁止虚拟化(SVM)。如果 val 的值是 CPU_ONLINE，则会调用 hardware_enable()函数，使特定 cpu 支持虚拟化，其具体实现已经在前面介绍过，因此在此不再说明。如果 register_cpu_notifier()函数调用失败，即返回值不为 0,则会跳转到 out_free_2 标签所指的位置，执行其后的代码，在此对这些错误处理的代码就不做详细的介绍了。

再回到 kvm_init()函数中，接下来调用的是 register_reboot_notifier()函数，该函数的作用是通过传递给它的参数 kvm_reboot_notifier 注册一个函数 kvm_reboot()，在机器(虚拟机)重启的时候调用。

函数 register_reboot_notifier()和前面介绍的 register_cpu_notifier()类似，最终也是通过 notifier_chain_register()函数在 reboot_notifier_list 中加入一个新的节点，他们之间主要的不同时注册的链表不同以及回调函数的调用时机不一样。函数 kvm_reboot()实现代码很简单，它的代码如图 2.22 所示：

```
/* virt/kvm/kvm_main.c */
    static int kvm_reboot(struct notifier_block *notifier, unsigned long val,
                  void *v)
    {
        /*[......]*/
        printk(KERN_INFO "kvm: exiting hardware virtualization\n");
        kvm_rebooting = true;
        on_each_cpu(hardware_disable, NULL, 1);
        return NOTIFY_OK;
    }
```

图 2.22　kvm_reboot()函数代码片段

为了节省篇幅，在此省略注释部分。函数 kvm_reboot()先是打印一条输出信息，然后布尔型变量 kvm_rebooting 设为 true，接下来对每一个 cpu 调用 hardware_disable()函数，最后返回 NOTIFY_OK。

函数 kvm_init()接下来调用的函数是 sysdev_class_register()，此函数是用来注册一个名为“kvm”的设备类，一个设备类用来描述一种类型的设备。“kvm”是由在调用函数 sysdev_class_register()时传递给它的参数 kvm_sysdev_class 的 name 成员指定的。变量 kvm_sysdev_class 的定义如图 2.23 所示：

```
/* virt/kvm/kvm_main.c */
    static struct sysdev_class kvm_sysdev_class = {
        .name = "kvm",
        .suspend = kvm_suspend,
        .resume = kvm_resume,
    };
```

图 2.23　变量 kvm_sysdev_class 的定义代码

由上面的代码可以看出，在此提供了两个操作，即 kvm_syspend 和 kvm_resume。对这两个函数而言，实际就是分别调用 hardware_disable()函数和 hardware_enable()函数。函数 hardware_enable()已经在前面介绍过，而函数 hardware_disable()相当于 hardware_enable()函数的逆操作，也在前面有简单的介绍。因此，在此就不再赘述。

调用 sysdev_class_register()函数返回后，如果没有错误(即返回值为 0 表示调用成功)则会调用函数 sysdev_register()，此函数是用于在内核树中添加一个系统设备，此函数的具体实现在此将不做分析。

函数 kmem_cache_create()用来创建缓存对象，在前面分析函数 kvm_mmu_module_init()时遇到过，在此不再赘述。接下来的 if 语句用来判断缓存创

建是否成功，如果创建失败，即返回值为空，则设置错误号，并跳转到错误处理代码处，即 out_free_5 标签所在位置，执行后面的错误处理代码。如果函数调用成功，则跳过 if 后面语句块中的代码，继续向后执行。接下来便是如图 2.24 所示的三行代码：

```
/* virt/kvm/kvm_main.c */
    kvm_chardev_ops.owner = module;
    kvm_vm_fops.owner = module;
    kvm_vcpu_fops.owner = module;
```

图 2.24　kvm_ini()函数代码片段

上面的三行代码只是简单的赋值语句。kvm_chardev_ops、kvm_vm_fops 和 kvm_vcpu_fops 三个变量都是 struct file_operations 类型的结构体变量。在 file_operations 结构体的定义中，其第一个成员便是一个模块指针类型的变量，而并不是一个操作，这个成员是用来防止模块在使用的情况下被卸载。由对函数 kvm_init() 的调用传递过来的实参可知，上面代码中三个变量的 owner 成员的值都被赋值为 THIS_MODULE。

接下来调用的函数是 misc_register()，单从名字上来看此函数与前面的两个函数 sysdev_class_register()和 sysdev_register()类似，都是完成注册功能。此函数实际是通过调用 device_create()函数来创建一个设备并向 sysfs 注册。调用 misc_register()函数结束后，将会在“/dev/”目录下创建“kvm”文件，并且提供了对此文件的操作。在后面 2.3 节中将看到用户空间的测试程序对“/dev/kvm”文件的操作代码。

再回到 kvm_init()函数，接下来执行的两行代码如图 2.25 所示：

```
/* virt/kvm/kvm_main.c */
    kvm_preempt_ops.sched_in = kvm_sched_in;
    kvm_preempt_ops.sched_out = kvm_sched_out;
```

图 2.25　kvm_ini()函数代码片段

这两行赋值语句的代码是用于任务被抢占或重新被调用时指定特定的函数来执行，在任务(虚拟机进程)被重新调度时，将调用 kvm_sched_in()函数，加载 guest 状态，进入 guest 模式，任务被抢占时调用 kvm_sched_out()函数，保存 guest 状态，加载 host 状态，退出 guest 模式。

最后，在前面函数调用没有出错的情况下，kvm_init()函数将执行最后一条语句 return 0，返回到 svm_init()函数中，此时 svm_init()也成功返回。到此为止，就将 kvm-amd.ko 模块加载过程中的函数调用过程及其实现以及遇到的一些其他的相关函

数分析完了。

由于在这个模块加载的过程中，调用的函数比较多，并且调用层次比较深，为了更清晰的展现出函数之间的调用关系，在此将函数的调用关系做如下的整理。

```
svm_init()
    -->kvm_init()
        -->kvm_init_debug()
            -->debugfs_create_dir()
            -->debugfs_create_file()
        -->kvm_arch_init()
            -->ops->cpu_has_kvm_support()
                -->has_svm()
            -->ops->disabled_by_bios()
                -->is_disabled()
            -->kvm_mmu_module_init()
                -->kmem_cache_create()
                -->register_shrinker()
            -->kvm_init_msr_list()
                -->rdmsr_safe()
            -->kvm_mmu_set_nonpresent_ptes()
            -->kvm_mmu_set_base_ptes()
            -->kvm_mmu_set_mask_ptes()
        -->alloc_page()
        -->page_to_pfn()
        -->zalloc_cpumask_var()
        -->kvm_arch_hardware_setup()
            -->kvm_x86_ops->hardware_setup()
                -->svm_hardware_setup()
                    -->alloc_pages()
                    -->page_address()
                    -->boot_cpu_has()
                    -->kvm_enable_efer_bits()
                    -->svm_cpu_init()
                    -->cpuid_edx()
        -->smp_call_function_single()
            -->kvm_arch_check_processor_compat()
                -->svm_check_processor_compat()
        -->hardware_enable()
        -->register_cpu_notifier()
        -->register_reboot_notifier()
        -->sysdev_class_register()
        -->sysdev_register()
```

```
        -->kmem_cache_create()
        -->misc_register()
    out_free:
        -->kmem_cache_destroy()
    out_free_5:
        -->sysdev_unregister()
    out_free_4:
        -->sysdev_class_unregister()
    out_free_3:
        -->unregister_reboot_notifier()
        -->unregister_cpu_notifier()
    out_free_2:
        -->on_each_cpu()
            -->hardware_disable()
    out_free_1:
        -->kvm_arch_hardware_unsetup()
            -->kvm_x86_ops->hardware_unsetup()
                -->svm_hardware_unsetup()
    out_free_0a:
        -->free_cpumask_var()
    out_free_0:
        -->__free_page()
    out:
        -->kvm_arch_exit()
            -->kvm_mmu_module_exit()
        -->kvm_exit_debug()
            -->debugfs_remove()
```

在前面表示的函数调用关系中，箭头表示函数的调用关系，缩进表示函数调用的层次关系，为了简明起见，此处并没有将所调用的函数全部表示出来。为了不引起混淆，调用关系表示的更加清晰，对于 kvm_init()函数的最后部分中出现的标签部分，函数和标签一同给出。

标签后面的函数或宏定义的作用一般情况下都与跳转语句前面的函数或宏定义相对应，当然作用是相逆的。如果在某个函数执行失败，则需要注销前面所获取的资源，否则内核很可能就处于一个不稳定的状态，为了防止这样的情况发生，因此就有了这些标签及其后面的函数。比如说前面的函数是分配资源，则对应的就是释放掉分配的资源，前面的是创建文件，对应的则是删除文件，前面的是进行注册，则与之对应的是取消注册等等。由于篇幅的关系，在此不对这些错误处理函数的实现做详细的

介绍了。

此时，已经成功的加载了 kvm-amd.ko 模块，之后就可以使用命令行或者图形化方式来创建虚拟机了，当然还需要一些其他的准备工作，具体创建和运行虚拟机的方法将会在第 3 章介绍。

2.2.3.3 kvm-intel.ko 模块分析

对于最后一个模块 kvm-intel.ko 来说，加载这个模块使用命令 modprobe kvm-intel。在加载时会调用定义在 vmx.c 文件中的 vmx_init()函数，在此函数中同样像 svm_init()函数一样调用 kvm_init()函数，只不过在调用它之前和之后有一些函数调用来设置一些条件，这些函数调用在此就不做详细的介绍了。

当进入 kvm_init()函数后，其执行流程与 kvm-amd.ko 模块是相同的，与 kvm-amd.ko 的不同之处在于当调用体系结构相关的函数(如一些初始化或一些其他的函数)时，此处调用的是位于 vmx.c 文件中的函数而不是位于前面介绍的 svm.c 中的函数，具体函数由 kvm_x86_ops 定义时指定，但是它们提供的功能是相同的或有些细微的差别，因此就不对使用到的 vmx.c 文件中的这些函数的具体实现做介绍了。

2.3 KVM 其他部分的代码简要分析

分析完 KVM 位于内核部分的代码后，为了能够更加完整的介绍 KVM，在此简要介绍 KVM 在用户空间的部分程序，从而了解使用 KVM 创建和运行虚拟机的代码流程。

本文接下来要分析的程序是 kvm-12 的用户空间的程序(也就是位于 kvm-12/user 目录下的部分程序)以及介绍 KVM 用户空间程序与内核交互的函数调用过程。kvm-12 程序的版本号为 12，本版本是 Linux2.6.20 内核中提供虚拟化功能所需的版本(KVM 的各个版本源代码可在 http://sourceforge.net/projects/kvm/files/网站下载获得)。

位于 user 目录下的 main.c 文件可以用来测试 KVM 启动和运行虚拟机，接下来将通过简要分析位于此文件中的 main()函数来理解使用 KVM 创建和运行虚拟机的内部过程，代码如图 2.26 所示：

```
/* kvm-l2/user/main.c */
    int main(int ac, char **av)
    {
        void *vm_mem;
        kvm = kvm_init(&test_callbacks, 0);
        if (!kvm) {
            fprintf(stderr, "kvm_init failed\n");
            return 1;
        }
        if (kvm_create(kvm, 128 * 1024 * 1024, &vm_mem) < 0) {
            kvm_finalize(kvm);
            fprintf(stderr, "kvm_create failed\n");
            return 1;
        }
        [......]
        kvm_show_regs(kvm, 0);

        kvm_run(kvm, 0);
        return 0;
    }
```

图 2.26　kvm-12/user/main.c 文件中 main()函数代码片段

在 main()函数中首先会调用 kvm_init()函数来创建一个新的 kvm 上下文环境，并将一个类型为 strcut kvm_context* 的变量 kvm 指向这个新创建的上下文环境。传递给 kvm_init()函数的第一个参数是类型为 static struct kvm_callbacks 变量 test_callbacks 的地址，第二个参数没有用到，因此赋值为 0。kvm_init()函数的源代码如图 2.27 所示：

```
/* kvm-l2/user/kvmctl.c */
    kvm_context_t kvm_init(struct kvm_callbacks *callbacks,
                   void *opaque)
    {
        [......]
        fd = open("/dev/kvm", O_RDWR);
        [......]
        kvm = malloc(sizeof(*kvm));
        kvm->fd = fd;
        kvm->callbacks = callbacks;
        kvm->opaque = opaque;
        return kvm;
     out_close:
        close(fd);
        return NULL;
    }
```

图 2.27　kvm-12/user/kvmctl.c 文件中 kvm_init()函数代码片段

函数 kvm_init()首先调用 open()函数打开“/dev/kvm”设备文件(在 kvm_init()函数中省略的部分是一些变量定义、错误判断以及错误处理的代码，在此不做介绍)，打开文件之后调用 malloc()函数分配内存空间，然后就是一些赋值操作来初始化创建的 kvm 环境，最后如果没有错误就返回 kvm，即指向 kvm 上下文环境的指针。

调用 kvm_init()返回之后判断返回值 kvm 是否为空，如果是则说明调用失败，此时则会输出错误信息来表明 kvm_init 调用失败，并且返回错误值 1。否则接着调用 kvm_create()函数通过 ioctl 系统调用来创建一个虚拟机，传递给 kvm_create()函数的第一个参数是由 kvm_init()函数创建的 kvm 上下文环境指针，即 kvm_init()的返回值;第二个参数是用来指定分配给此虚拟机的物理内存的大小;第三个参数是一个 void**类型的变量，用来指向有 kvm_create()函数分配的空间。同样，如果 kvm_create()函数调用失败，则会调用 kvm_finalize()函数来清除 kvm_init()函数创建的上下文环境，输出错误提示信息，之后返回错误值 1。

函数 kvm_create()的代码片段如图 2.28 所示：

```
-----------------------------------------------------------------------------
int kvm_create(kvm_context_t kvm, unsigned long memory, void **vm_mem)
{
    [......]
    r = ioctl(fd, KVM_SET_MEMORY_REGION, &low_memory);
    [......]
    r = ioctl(fd, KVM_CREATE_VCPU, 0);
    [......]
}
-----------------------------------------------------------------------------
```

图 2.28　kvm-12/user/kvmctl.c 文件中 kvm_create()函数代码片段

由图 2.28 可以看出，在 kvm_create()函数中，通过 ioctl 系统调用来设置虚拟机的内存区域，之后同样通过 ioctl 创建虚拟 cpu。通过对 ioctl 传递的命令参数 KVM_CREATE_VCPU 调用 kvm_dev_ioctl_create_vcpu()函数，而在其中通过 kvm_arch_ops->vcpu_create()根据加载模块的不同调用 svm_create_cpu()函数或 vmx_create_cpu()函数完成虚拟 cpu 的创建，最终完成虚拟机的创建过程。

接下来调用 kvm_show_regs()函数(如果加载的是 kvm-amd.ko 模块，在其中通过系统调用 ioctl,最终调用函数 svm_get_msr())来输出一些关于此虚拟机的寄存器信息。

最后会调用 kvm_run()函数来运行虚拟机，kvm_run()函数的执行过程与 kvm_create()相似，在 kvm_run()函数中，通过 ioctl 向内核传递 KVM_RUN 命令参数

调用 kvm_dev_ioctl_run() 函数，同样在其中通过 kvm_arch_ops->run() 调用 svm_vcpu_run()或 vmx_cpu_run()函数来运行虚拟机。并且在虚拟机运行过程中调用由变量 test_callbacks 指定的函数来进行一些测试，在此对 test_callbacks 中的域值所对应的函数不做分析。

2.4　本章小结

本章的内容是本文的主要内容，主要介绍了在 Linux 内核中关于 KVM 实现的部分实现代码。除了对具体的 c 语言代码进行分析外，还包括对相关配置文件 Kconfig 与 Makefile 文件的分析，并且在 2.3 节对 KVM 用户空间的一小部分程序，即如何创建与运行虚拟机的代码进行了简单的介绍。

第 3 章　KVM 的安装使用

3.1　Ubuntu 操作系统

Ubuntu 是一个开关机速度快、界面精美、操作简单的 Linux 操作系统，并且是免费使用的[20]。对 Linux 各个操作系统产品而言，Ubuntu 是本人使用最多也是最喜欢的一款。

Ubuntu 吸引人的另一个地方就是它具有非常完备的软件集，安装软件非常简单方便，在其软件中心(Ubuntu Software Center)里面有大量的软件可供选择，不仅仅包含文字处理和电子邮件等日常软件，还包括大量的编程工具，软件中心提供的这些软件足以日常的应用与学习，并且这些软件的安装和删除都非常的简单，只需要点击几下按钮就能完成。

3.2　Libvirt

Libvirt 是一套开源的，用于和虚拟化 Hypervisor 交互的 API。它对本文涉及到的 Xen、KVM、VMware 和 Virtualbox 等 Hypervisor 以及 QEMU 模拟器都提供支持。在 3.4 节介绍的 Ubuntu 操作系统下利用图形化软件使用 KVM 时，需要先安装 Libvirt0 软件包。

3.3　VMM

此处的 VMM 是指“Virtual Machine Manager”,它是一款用来管理虚拟机的应用程序软件(在本章中如无特殊说明，VMM 就是指“Virtual Machine Manager”)，使用它可以通过可视化的方式创建和运行虚拟机，并且可以在其运行界面上方便的查看虚拟机内存的使用情况和 CPU 的利用率等统计信息。在下一节中将看到使用 VMM 创建虚拟机的各个步骤的使用界面。

3.4　KVM 在 Ubuntu 下的使用方法

在使用 VMM 创建虚拟机之前，简要的介绍了 VMM 和 Libvirt，它们与 KVM 之间的关系如下图 3.1 所示：

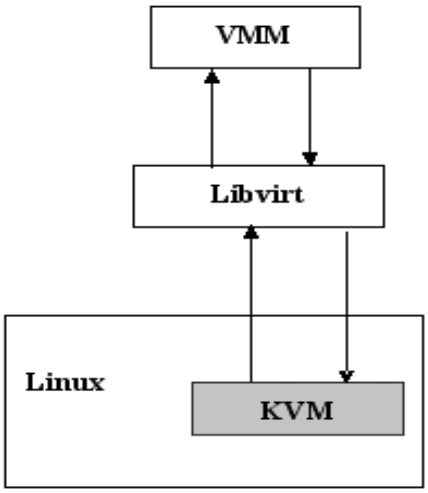

图 3.1　VMM、Libvirt 和 KVM 的层次关系

如果想通过下载源代码(源代码的下载地址在 2.3 节中已经给出)，手动编译运行 KVM，可参考 http://www.linux-kvm.org/page/HOWTO1 中的内容，文中以命令行的方式比较完整的提供了对 KVM 的编译、运行以及对虚拟机的配置等方法。

本文利用 KVM 来在 Ubuntu10.04 版本的操作系统下创建虚拟机，但是采用另一种简单、方便、图形化的方法。具体步骤如下 (前 4 步是创建虚拟机的前期工作，从第 5 步开始创建虚拟机)：

1、加载 KVM 模块

在命令行窗口中输入命令：

modprobe kvm

和

modprobe kvm-amd

(对于 Intel 的处理器来说使用 modprobe kvm-intel)来加载 kvm 模块。加载 kvm 需要硬件的支持，如果处理器不支持，则会在执行加载命令时出错，并会显示“Operation not supported”错误信息。

2、安装 Libvirt

在 Ubuntu Software Center 中搜索 libvirt,便可找到 libvirt0,点击之后的安装按钮对其进行安装。

3、安装 VMM

在安装完 libvirt0 后搜索 virtual machine manager,找到之后点击安装按钮即可安装。

4、运行 VMM

在安装好 VMM 后，便可在 Application->System Tools 下看到 Virtual Machine Manager，点击即可运行，如图 3.2 所示：

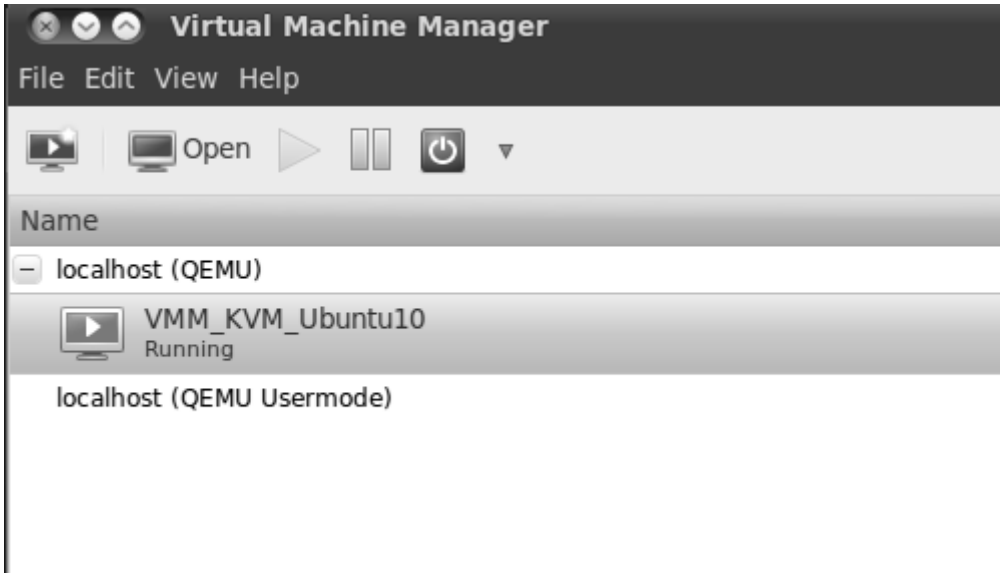

图 3.2　Virtual Machine Manager 使用界面

5、创建虚拟机

这时就可以使用 VMM 创建虚拟机了，在上面的图中有一个创建新虚拟机的按钮，即 open 按钮左边的按钮，点击之后就会出现创建虚拟机的向导，具体步骤如下所示：

第一步：需要输入虚拟机的名字，如“test”，下面是选择连接方式，我们选择 localhost(QEMU/KVM)，后面的单选按钮是选择安装操作系统的方式，此时选择本地安装，之后点击下一步，也就是的 forward 按钮，如下图 3.3 所示：

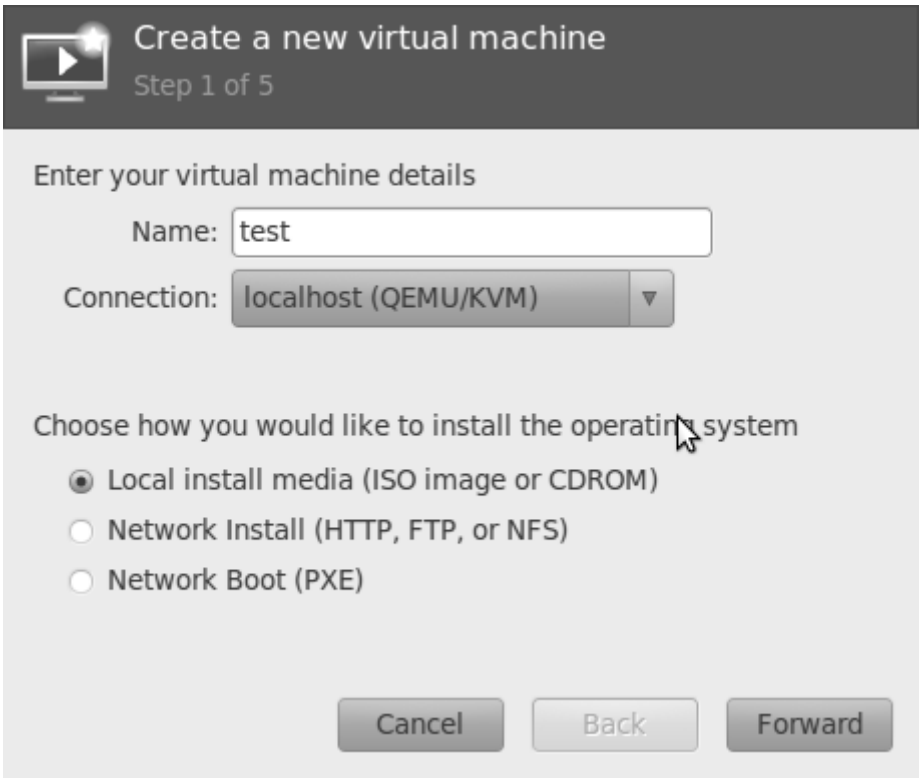

图 3.3　使用 KVM 创建虚拟机步骤一

第二步：选择安装媒介，可以根据具体情况选择是光驱安装还是 iso 镜像安装，我选择了后者，后面要选择操作系统的类型以及版本，此时根据前面的光驱或者是 iso 镜像文件的信息来选择，这里我选择的操作系统类型为 Linux，版本为 Ubuntu 10.04，之后点击 forward 按钮，如下图 3.4 所示：

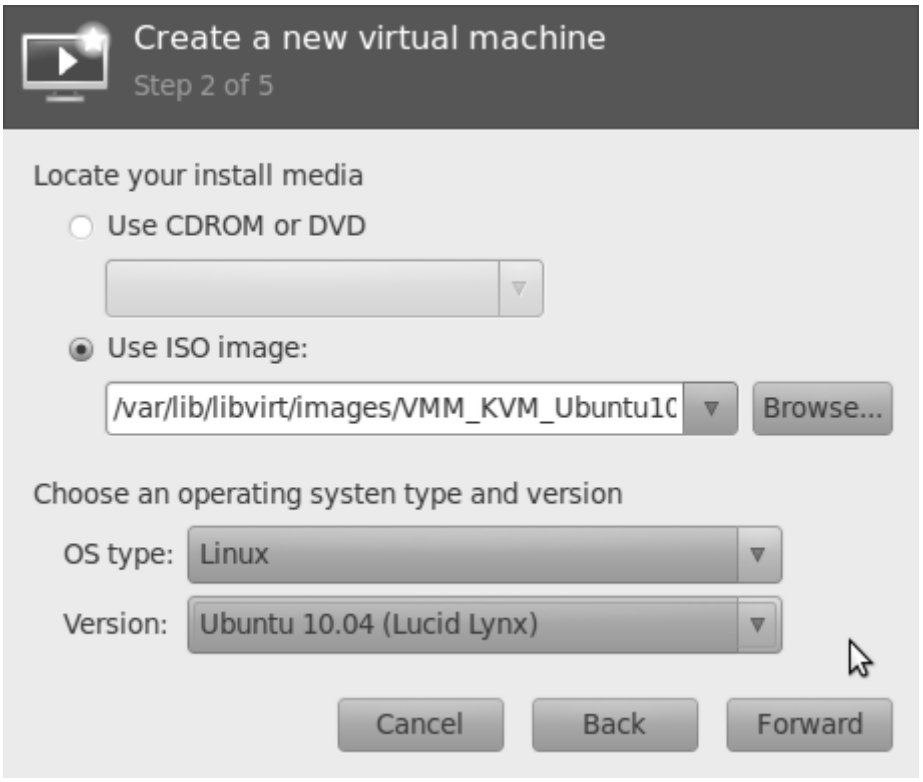

图 3.4　使用 KVM 创建虚拟机步骤二

第三步：这一步是设置分配给虚拟机的内存的大小和 cpu 的个数，在设置的下方有可用内存和 cpu 的上限，可以在可选范围内设置，我在此设置的内存为 512MB，cpu 个数为一，之后点击 forward 按钮进入下一步，第三步安装视图如下图 3.5 所示：

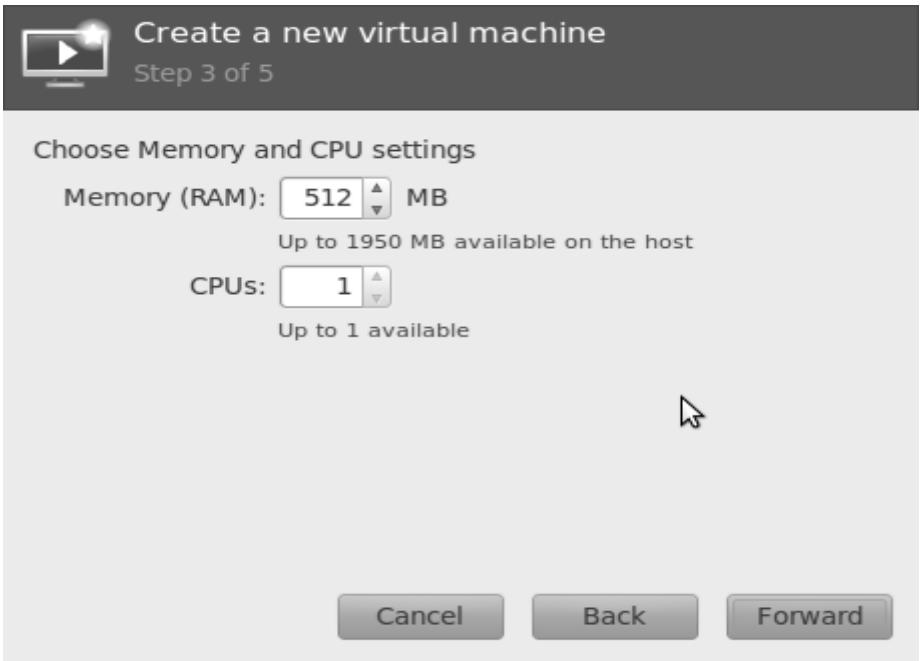

图 3.5　使用 KVM 创建虚拟机步骤三

第四步：为虚拟机分配磁盘空间，可根据下面提示的磁盘剩余空间合理分配，下面还可以选择已有的存储空间，这与我们平时创建或打开文件的操作类似。此时如果是第一次创建虚拟机的话选择前者，此处分配了 8G 磁盘空间，如下图 3.6 所示：

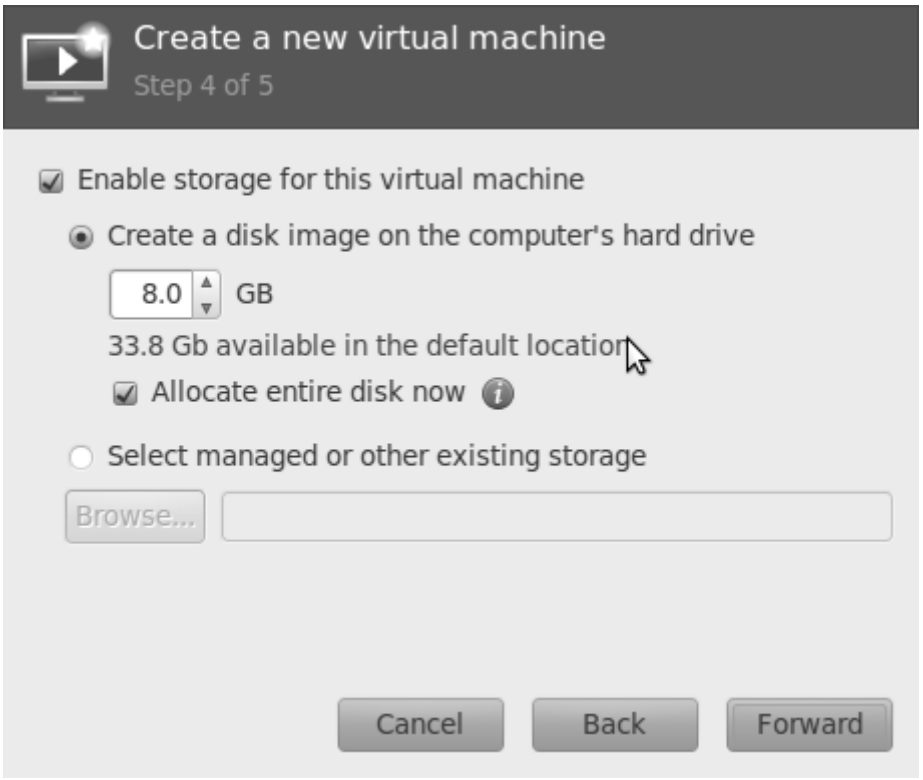

图 3.6　使用 KVM 创建虚拟机步骤四

第五步：在前面的步骤完成后，这一步会显示前面设置信息总结，唯一需要说明的是下面的高级选项，也就是 Advanced options，点开之后，下面可以配置虚拟机网络的连接方式，设置固定的 MAC 地址，选择 hypervisor 类型，即 KVM，还有机器的体系结构类型，设置完成后点击 finish 按钮，就完成了创建虚拟机的步骤，如下图 3.7 所示：

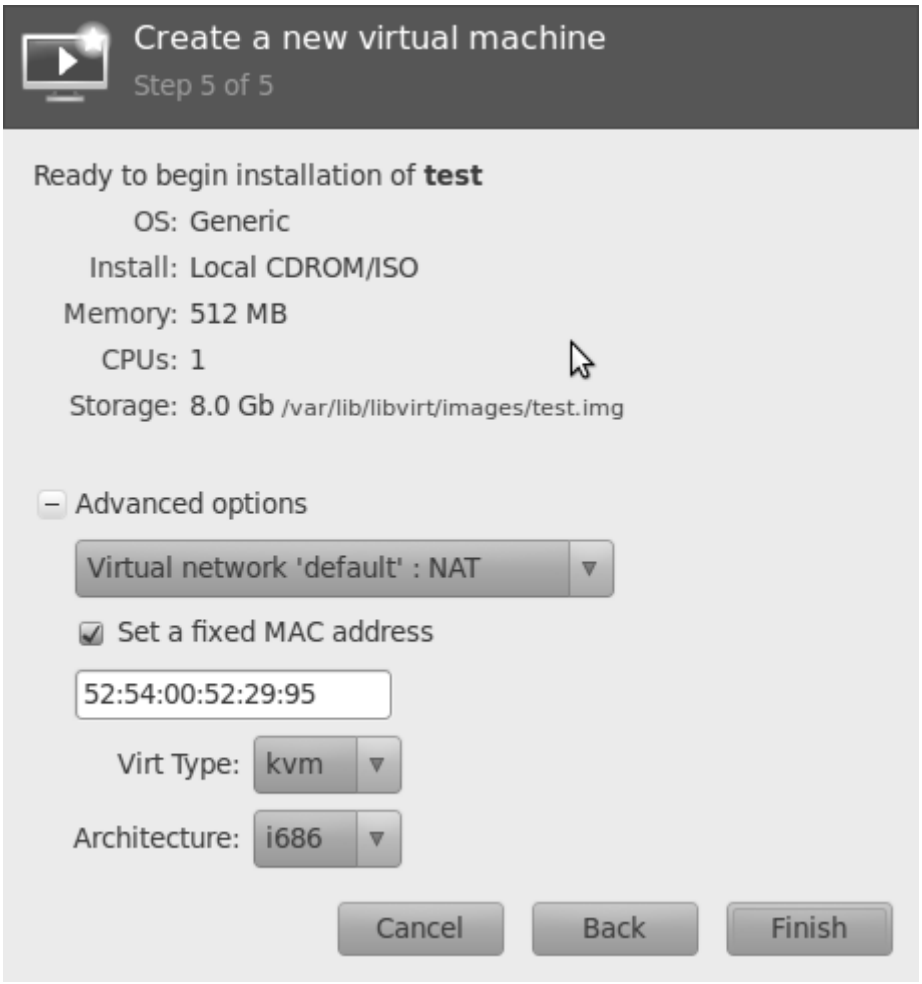

图 3.7　使用 KVM 创建虚拟机步骤五

在创建完虚拟机后就可以启动虚拟机了，点中新创建的虚拟机，在 VMM 上方的一排按钮中有一个三角形的按钮，点击即可启动虚拟机，或者双击新创建的虚拟机也可启动，之后点击 open 按钮就可以看到启动画面，此时便可使用新创建的虚拟机来安装客户操作系统，安装完成后便可使用虚拟机提供的服务了。

3.5　本章小结

本章首先简单的介绍了 Ubuntu 操作系统，然后介绍了 Libvirt 和 VMM，最后介绍了在 Ubuntu10.04 操作系统下利用图形化工具 VMM 使用 KVM 创建虚拟机的方法与步骤。

第 4 章　虚拟化软件性能比较

4.1　虚拟机的配置

本章将对前面介绍的几款虚拟化软件进行性能测试，通过测试在虚拟化软件中创建的虚拟机来衡量各虚拟化软件的性能。为了保证公平性及结果的可靠性，各虚拟化软件中虚拟机的测试环境配置相同，如下表 4.1 所示：

表 4.1　虚拟机环境配置

CPU	Athlon 3000+
内存大小	512MB
磁盘大小	8GB
Guest OS 类型	Ubuntu 10.04

在测试过程中本机(Native，在后面的测试图中统一使用 Native 表示本机。本文在对各虚拟机进行测试的同时也对本机进行了测试，这样做的目的主要是为了在比较各虚拟机性能差异的同时也可以通过图示看到各虚拟机性能与本机之间的差异)的内存大小为 2GB，磁盘大小为 160G，其他配置环境与表 4.1 中的配置相同。各虚拟化软件的版本分别为 VMware Workstation 6.5(在测试结果中用 VMware 表示)，而 KVM、VirtualBox 和 QEMU 的版本是 Ubuntu 10.04 操作系统 Ubuntu Software Center 中自带的版本。

4.2　测试内容与测试目的

本文将会通过一系列的测试程序对本机以及 VMware、KVM、QEMU 和 VirtualBox 的性能进行测试，主要包括对虚拟机的网络性能、磁盘 IO、CPU 处理能力和综合性能等方面的测试。其中，在对网络性能进行测试时，本文选用的网络性能测试软件是 Netperf，对磁盘 IO 进行测试的软件是 IOzone，通过 Phoronix-Test-Suite 测试工具集中提供的 build-linux-kernel 和 compress-gzip 两个测试套件来测试虚拟机对计算密集型程序的处理能力，另外通过 UnixBench 测试软件对虚拟机的系统性能进行测试。

为了减少误差，从而保证测试结果的准确性，本文在使用测试工具对虚拟机性能

进行性能测试时，所采用的测试结果，即下文中测试结果图中的测试数据是三次测试结果的平均值。

本文测试的目的是通过一系列的测试软件对虚拟机多方面的性能进行测试，从而根据各测试数据得出虚拟机的性能差异，由此可以对虚拟机用户在选择虚拟化软件时提供一定的依据和帮助。

4.3　性能测试

4.3.1　Netperf 测试

Netperf 是一款网络性能测试工具，可以用于多方面的网络性能测试[21]。接下来将使用 Netperf 测试本机以及几款虚拟化软件中的虚拟机的网络吞吐量。在 Ubuntu Software Center 中查找并安装 netperf 后，在命令行中输入 netperf -4 -H 127.0.0.1(参数 -4 是指使用 IPv4 协议，参数-H 用于指定连接目的机主机名或 IP 地址，本文指定为本机)命令进行测试，测试结果如下图 4.1 所示：

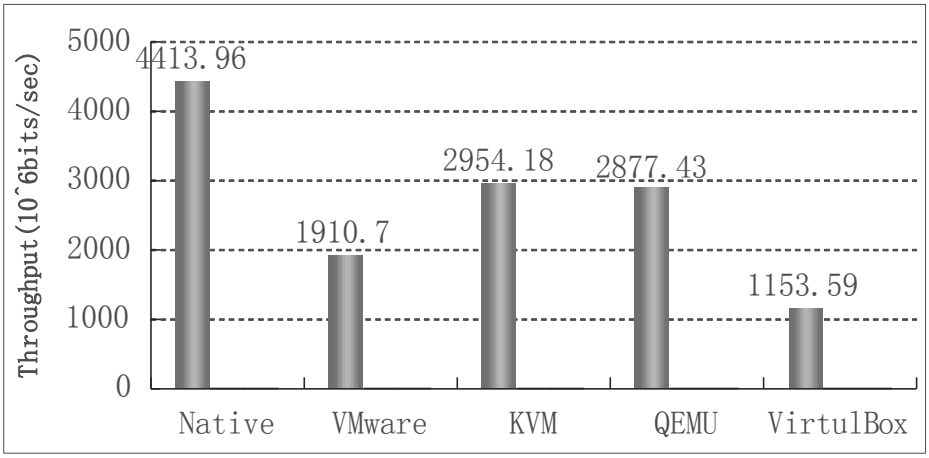

图 4.1　Netperf 测试结果

由图 4.1 Netperf 的测试结果，可以看出这四种虚拟化软件在网络吞吐量方面性能的差异，当然吞吐量越大越好，吞吐量大则更适宜用来构建 Web 服务器、mail 服务器等。因此，这些虚拟化软件在网络性能(吞吐量)测试方面的性能由高到低的排列顺序为 KVM、QEMU、VMware、VirtualBox。

4.3.2　IOzone 测试

IOzone 是一款免费的磁盘文件系统测试工具，可以使用它通过多种文件操作测试来对系统的 I/O 性能进行进行测试。IOzone 测试工具支持的测试包括对文件的读、写、重读、重写、逆向读、随机读等等[22]。本文使用 IOzone 提供的读、写测试来对各个虚拟机性能进行测试。

为了得到准确的测试结果，使用 IOzone 进行测试时，应该将测试文件的大小设置为大于机器内存的值，因此本文将测试文件的大小设置为对应测试机器内存大小的 2 倍。在 Ubuntu Software Center 中查找并安装 IOzone 完成后，对虚拟机进行测试时，在命令行中输入 iozone -i 0 -i 1 -s 1g 命令来对大小为 1g 的文件进行读、写测试，而在对本机(Native)测试时，在命令行中输入 iozone -i 0 -i 1 -s 4g 对大小为 4g 的文件进行测试。测试比较结果如下图 4.2 所示：

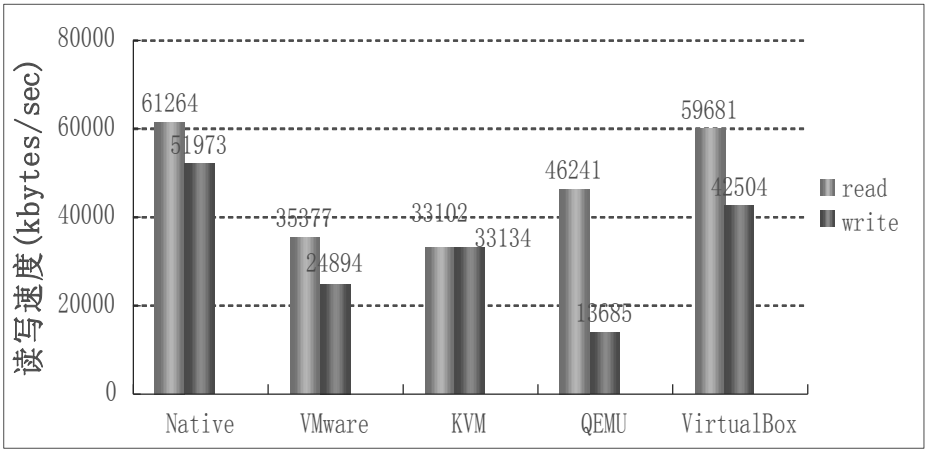

图 4.2　IOzone 测试结果

由图 4.2 IOzone 的测试结果，可以得出上面列举的四款虚拟化软件在 I/O 读写速度方面性能差异的结果。但就文件的读操作而言，它们的性能由高到低顺序为 VirtualBox，QEMU，VMware，KVM。写操作的性能由高到低的顺序为 VirtualBox，KVM，VMware，QEMU。如果将读与写操作的重要性看做相同的话，那么对读写操作的综合性能(将各虚拟机读操作速度与写操作的速度的测试结果相加之后除以 2，作为综合性能的结果)比较而言，他们的性能由高到低的顺序为 VirtualBox、KVM、VMware、QEMU。

4.3.3 UnixBench 测试

UnixBench 是一款类 Unix(Unix-like)操作系统的系统性能测试软件，它提供多种测试来对系统的性能进行多方面(测试不仅包括硬件，还包括操作系统、类库、编译器等[23])的测试。在测试结束后，最终会得出一个系统性能的分数(即图 4.3 中的 Final Score)，分数越高说明系统的综合性能越好。

本文选择 UnixBench 的目的是用来测试系统的综合性能。下载 unixbench-4.1.0 源代码后进行编译，之后运行源代码目录下的 Run 脚本文件，从而得到测试结果。测试结果如下图 4.3 所示：

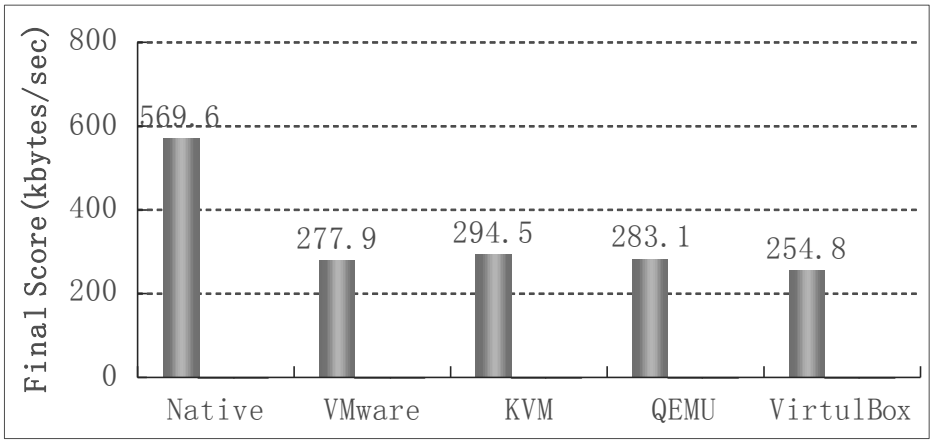

图 4.3　UnixBench 测试结果

在测试结束后，如图 4.3 所示最终会得出一个系统性能的分数(即图 4.3 中纵坐标表示的 Final Score)，分数越高说明系统的综合性能越好。由此可以得出，上述虚拟化软件就 UnixBench 测试结果而言的性能由高到低的顺序为 KVM、QEMU、VMware、VirtualBox。

4.3.4 Phoronix-Test-Suite 部分测试

Phoronix-Test-Suite 是一款测试工具集[24]，它集成多种测试工具与一身，并且可以在多种操作系统上运行。Phoronix-Test-Suite 拥有使用简单、易于扩展、测试结果准确、支持多种平台、测试集完备等特性。

本文采用 Ubuntu 10.04 操作系统 Ubuntu Software Center 中提供的版本，使用 Phoronix-Test-Suite 中的 build-linux-kernel(测试编译 Linux2.6.25 内核所需要的时间)

和 compress-gzip(测试使用 gzip 压缩一个文件所需要的时间)两个测试套件对各虚拟化软件的性能进行测试，测试结果分别如图 4.4 和图 4.5 所示：

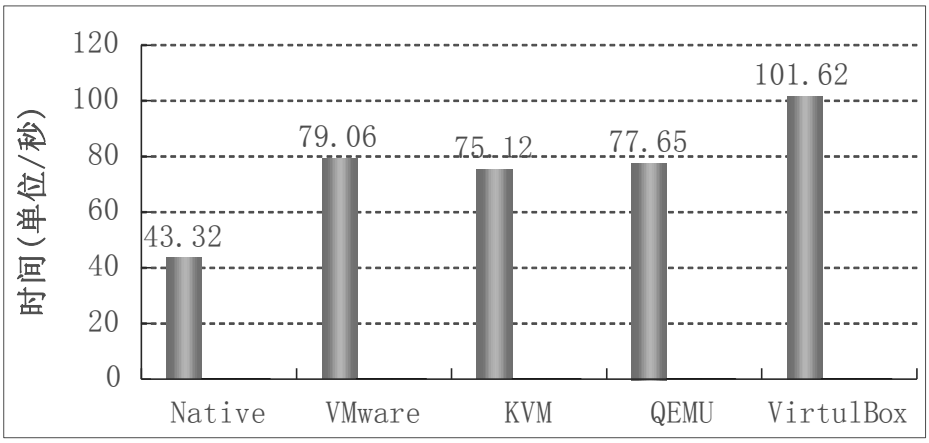

图 4.4　compress-gzip 测试结果

在此之前的测试中，结果值越大说明性能越高，而 compress-gzip 测试的是压缩文件使用的时间，因此值越小性能越高。从图 4.4 测试结果可以看出各虚拟化软件在压缩方面的性能由高到低的顺序为 KVM、QEMU、VMware、VirtualBox。

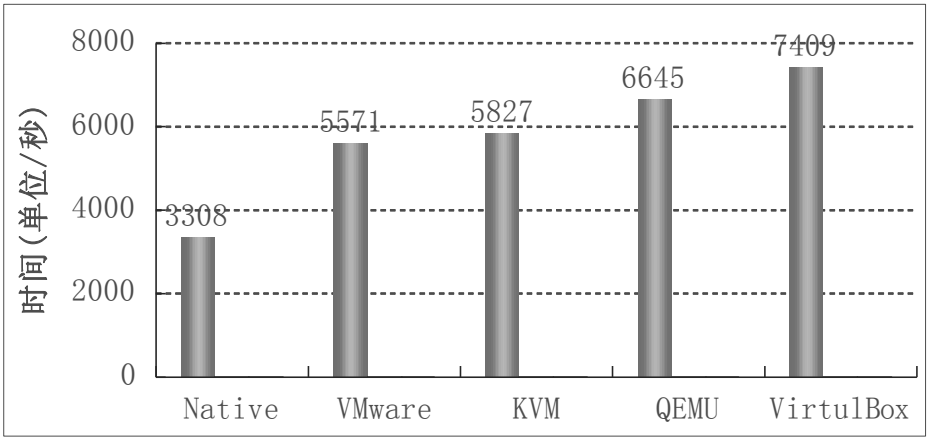

图 4.5　build-linux-kernel 测试结果

图 4.5 中显示的是各虚拟化软件编译 Linux2.6.25 内核所消耗的时间的柱形图，编译内核时消耗的时间越少说明性能越高。因此，它们在编译 Linux 内核时的性能由高到低的顺序为 VMware、KVM、QEMU、VirtualBox。

4.4　测试结果分析总结

从前面的 5 个性能测试结果来看，KVM 都有较好的表现。在网络性能测试中，KVM 在几款虚拟化软件中表现最佳，性能是主机性能的 67%。在磁盘 I/O 测试中，

VirtualBox 的性能在读、写测试两方面均为最好，只落后于主机 10%左右，KVM 的性能次之，其性能为主机性能的 58%，相差较多，其主要原因是 KVM 出现较晚，在 I/O 方面还有很大的改进空间，加之本机器硬件不支持 I/O 虚拟化，若有硬件辅助 I/O 虚拟化的支持，KVM 在 I/O 方面的表现必会有所上升。在 UnixBench 测试中，KVM 相对其他几款虚拟化软件性能最好，为主机性能的 52%。在 compress-gzip 的测试中，KVM 的性能为主机的 59%，性能也是最好的。在 build-linux-kernel 测试中，VMware 性能最好，KVM 次之，但是相差不多，VMware 的性能是主机的 59%，而 KVM 是主机的 57%。

从前面的 5 个测试结果可以看出，KVM 的性能表现很稳定，性能在主机性能的 50%到 60%之间。但是与主机性能相比，性能差异还是比较大的。因此，各虚拟化软件的性能改善空间还很大。在不考虑机器本身的配置(对于特定配置的机器，虚拟化软件的性能可能会不同)以及测试工具参数差异(参数变化有可能会得到不同的测试结果)的情况下，根据本文前面的测试结果，对 VMware、QEMU、KVM 和 VirtualBox 这四款虚拟化软件而言，KVM 的性能无疑是最好的。

4.5　本章小结

本章首先对虚拟机的配置做了简单的介绍，然后对 VMware、KVM、QEMU 和 VirtualBox 这几款虚拟化软件的性能进行了测试，并分别对测试结果进行了比较和分析，最后通过综合考虑得出他们之间性能优劣的比较结果。

第 5 章　总结与展望

5.1 总 结

近几年虚拟化技术快速发展，出现了许多优秀的虚拟化软件，如 VMware 系列、KVM、Xen、VirtualBox 等。随着硬件辅助虚拟化技术的出现，更是使虚拟化技术的研究迈向了一个新的高度，硬件虚拟化极大地降低了开发虚拟化软件的复杂度，并且使得虚拟机的性能得到了很大的提高。在开源领域，也有很多成功的虚拟化软件，最具代表性的要数 Xen 和 KVM 了。相比 Xen 而言，KVM 的结构简单，代码复杂度小，集成与 Linux 内核中，并且 KVM 得到了 RatHat 公司的支持。因此，KVM 必是虚拟化软件中的一颗新星。

本文主要是对基于内核的虚拟机，即 KVM 的研究。首先通过对虚拟化以及多款虚拟化软件进行了简单的介绍，接着使用了较多的篇幅来对 Linux2.6.31 内核中关于 KVM 的部分代码进行了分析，从而可以以通过分析源代码的形式来了解 KVM 的部分实现机制，以此来达到更好的学习 KVM 的效果。

本文不仅对 KVM 的部分实现代码进行了分析，而且对针对 Ubuntu 操作系统而言如何使用 KVM 也进行了详细的介绍，并且在后面对前文中提到的几款虚拟化软件，即 VMware Workstation、KVM、QEMU 和 VirtualBox 进行了多方面的性能分析，对使用的每一种测试软件得出的结果都进行了比较分析，在最后将多个测试结果综合起来进行比较，得出这些虚拟化软件在性能方面的优劣表现。

5.2 展 望

目前，虚拟化技术已经成为热门的研究领域，各种虚拟化软件也层出不穷，但是国内关于 KVM 或其他的虚拟化软件的研究相对于国外还比较落后，还没有出现一款自主研发的能够进入主流行列的虚拟化软件，今后将面临巨大的挑战与机遇。

本文的研究内容虽然到此结束，今后关于 KVM，虚拟机以及虚拟化的研究并不会结束。在将来的学习中，不仅会继续关注 KVM 的发展动态，学习与研究关于 KVM 的新技术，同时也会关注其他的虚拟化软件以及整个虚拟化技术的发展趋势。

附　录

附录 A 英文缩写

API	application program interface
CLI	common language infrastructure
CPU	central processing unit
GCC	gun compiler collection
GNU	gnu's not unix
GPL	general public license
IT	information technology
JVM	java virtual machine
KVM	kernel-based virtual machine
MSR	machine specific registers
SMP	symmetrical multi-processing
SVM	secure virtual machine
VM	virtual machine
VMX	virtual machine extensions
VMM	virtual machine monitor / virtual machine manager
VT	virtualization technology

参考文献

[1] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM:the Linux Virtual Machine Monitor [C].Proceedings of the Linux Symposium, Ottawa, Ontario,Canada,2007.

[2] Uhlig, R. Neiger, G. Rodgers, D. Santoni, A.L. Martins, F.C.M. Anderson, A.V. Bennett, S.M. Kagi, A. Leung, F.H. Smith, L. Intel virtualization technology [J].IEEE/IET,2005.

[3] AMD Inc. AMD64 Architecture Programmer’s Manual Volume 2: System Programming [M/OL].AMD64 Technology,2005,437-504.

[4] Susanta Nanda,Tzi-cker Chiueh. A Survey on Virtualization technologies. http://www.ecsl.cs.sunysb.edu/tr/TR179.pdf

[5] James E.Smith,Ravi Nair.The architecture of virtual machines [J].IEEE/IET,2005.

[6] VMware.A Performance Comparison of Hypervisors. http://www.vmware.com/pdf/hypervisor_performance.pdf

[7] VMware 中文官网.https://www.vmware.com/cn/

[8] KVM Main-Page.http://www.linux-kvm.org/page/Main_Page/

[9] Qumranet Inc.KVM Whitepaper. http://www.linuxinsight.com/files/kvm_whitepaper.pdf

[10] Jonathan Corbet,Alessandro Rubuni,Greg Kroah-Hartman.Linux Device Drivers [M].O’Reilly Media,2005.

[11] Xen Home-Page.http://www.xen.org/

[12] VirtualBox.http://www.virtualbox.org/

[13] QEMU.http://wiki.qemu.org/Main_Page

[14] The Linux Kernel Archives.http://www.kernel.org/

[15] Diomidis Spinellis.Code Reading [M].Addison Wesley,2003.

[16] Robert Love.Linux Kernel Development Second Edition [M].Sams Publishing,2005.

[17] Brian W. Kernighan and Dennis M. Ritchie.The C programming Language [M].Prentice Hall PTR,1988.

[18] Claudia Salzberg Rodriguez, Gordon Fischer, Steven Smolski.The Linux

Kernel Primer [M].Prentice Hall PTR,2005.

[19] GCC online documentation.http://gcc.gnu.org/onlinedocs/

[20] Ubuntu 中文官网.http://www.ubuntu.com.cn/

[21] Netperf.http://www.netperf.org/netperf/

[22] IOzone Filesystem Benchmark.http://www.iozone.org/

[23] UnixBench.http://code.google.com/p/byte-unixbench/

[24] Phoronix-Test-Suite.http://www.phoronix-test-suite.com/

致　　谢

经过几个月的努力，终于完成了硕士毕业论文。在此，我首先要感谢我的指导老师徐高潮教授，徐老师治学严谨、认真负责，并为我提供了宽松、自由的学习环境，使我能专心于本文的研究。在论文完成之际，谨向徐老师致以最诚挚的感谢。

在本文的撰写期间，也得到了其他老师、同学的帮助，在此向他们表示感谢，同时也感谢对 Linux 内核开发以及 KVM 开发做出贡献的所有人员，没有他们的无私的付出，KVM 也不会如此快速的发展，也不会有本文的出现。

最后，需要特别感谢的是我的家人，感谢他们一如既往的鼓励与支持。