



华北电力大学

毕 业 设 计（论 文）

题 目 基于 Qume 模拟器移植 rCore 操作系统的
开发与实现

院 系	控制与计算机工程学院
专 业	计算机科学与技术
班 级	计算 1702 班
学生姓名	杨秉学
学 号	120171080212
指导教师	琚 贇

二〇二一年六月

摘 要

RISC-V 是一款近年来最为流行的开源指令集架构，而且被广泛应用于各个场景。

本文是采用 Qemu 虚拟化模拟仿真出 RISC-V 指令环境，用于解决跨平台的模型部署和运行问题。并在模拟出来的 RISC-V 处理器上面进行移植 Linux 内核、文件系统以及网络协议栈，最后尝试在上面运行 RISC-V 程序，主要目的是因为本科所学知识理论与实践衔接并不紧密，感觉“纸上得来终觉浅，绝知此事要躬行”，很多内容似懂非懂，一方面还能凭借书本的记忆说上几句，做对几道题，回答上老师的几个问题；另一方面心里又十分清楚地认识到自己学得不是很到位，很多知识内容如果稍微较真一点深入地探讨一下，就会发现自己有很多内容解释不清楚，更谈不上熟练驾驭了，所以本文将对本科所学有关操作系统，计算机组成原理，计算机体系结构还有计算机网络等计算机科学核心课程内容结合 RISC-V 指令集和 Linux 操作系统的移植包括一步步完善基本功能来达到对知识的综合理解与回顾。

关键字：RISC-V, Qemu, linux

ABSTRACT

RISC - V is one of the most popular open source instruction set architecture in recent years, and is widely used in various application scenarios.

This paper uses Virtualization Technology of QEMU to simulate RISC-V instruction environment, which is used to solve the problem of cross-platform model deployment and operation. In addition, I transplanted Linux kernel, file system and network protocol stack on the simulated RISC-V processor, and finally tried to run RISC- V program on it. The main purpose was that there was not relative combination of theory and practice of the knowledge that I had learned in the undergraduate course. As the old saying goes "Paper will sleep shallow, never know the matter want to practice." On the one hand, can also rely on the memory of the book to say a few words, do a few questions, answer the teacher's several questions; On the other hand, I clearly realize that I have not learned well. If I study up on it a little more deeply, I will find that I can't explain a lot of concept clearly, let alone master it skillfully. Therefore, this paper will refer to the operating system, principle of computer organization, Computer architecture and computer networking that are the core courses of computer science, incorporating RISC-V instruction set and Linux operating system, including step by step improvement of basic functions to achieve a comprehensive understanding of the knowledge.

KEY WORDS:RISC-V, Qemu,linux

目 录

摘 要.....	I
ABSTRACT	II
第 1 章 绪论	1
1.1 课题研究背景和意义.....	1
1.2 国内外研究现状.....	1
1.3 本文研究内容	2
第 2 章 计算机组成与设计——基于 RISC-V	3
2.1 指令表示方法与指令格式	3
2.1.1 R 指令	3
2.1.2 I 指令	3
2.1.3 S 指令与 B 指令	4
2.1.4 U 指令和 J 指令	4
2.2 流水线.....	4
2.2.1 处理器性能度量方法.....	4
2.2.2 结构冒险.....	4
2.2.3 数据冒险.....	5
2.2.4 控制冒险.....	6
第 3 章 Qemu & KVM 基本原理.....	7
3.1 虚拟化.....	7
3.1.1 软件虚拟化	7
3.1.2 硬件虚拟化	7
3.2 硬件虚拟化介绍.....	7
3.2.1 CPU 虚拟化.....	7
3.2.2 内存虚拟化	9
3.2.3 I/O 虚拟化.....	10
3.3 KVM & Qemu 模拟器介绍	10
3.3.1 KVM 内核模块	11
3.3.2 QEMU 用户态设备模拟.....	12
3.4 KVM 上层管理工具.....	13
3.4.1 libvirt	13
3.4.2 virsh.....	13
3.4.3 virt-manager	13
第 4 章 实验流程.....	15
4.1 前期准备	15

4.1.1 交叉工具链的准备	15
4.2 Qemu & KVM 搭建	17
4.3 移植过程介绍	17
4.4 制作 BootLoader——BBL(Berkeley Boot Loader)	17
4.5 创建根文件系统	17
4.6 配置 SSH 服务	17
4.7 最终效果	18
第 5 章 与前人比较	20
5.1 收获	20
5.2 不足	20
第 6 章 总结	21
参考文献	23
附 录	24
附录 A	24
附录 B	24
致 谢	28

第 1 章 绪论

1.1 课题研究背景和意义

在“中兴事件”刚过没多久，某些国家就开始发动“华为事件”，进行各种制裁，这对我国半导体产业的打击极大，是可忍孰不可忍，放眼整个产业链，我们国家连一个像样的成熟指令集架构都没有，连一个可大规模商用的操作系统都没有，当我们想研究开发产品竟然还需要看别人脸色来得到授权，这是莫大的耻辱。为什么外国人能做出来，我们就做不出来，我们也并不比他们差什么。所以，我觉得我有必要研究这样的问题了，这就是为什么我毕业设计要研究操作系统，指令集这样的内容，帝国主义封锁我们，不让我们发展，但我偏偏就要研究这些偏底层的原理。

首先是选择的指令集——Risc-v，它是一个最近流行基于 RISC 的指令集架构，它就类似于软件领域开源的 linux 一样，Risc-V 也有在很多领域发挥作用，如果将 linux 与 RiSC-V 结合起来发展的话，我们就可以利用已有的生态，站在巨人们的肩膀上完成自己的目标，而且开源的话可以集思广益，大家一起参与进来，在这个过程中可以发现自己的不足，然后改正，这本身就是一个学习提升的过程，是一个充满意义的事情。

1.2 国内外研究现状

美国的加州大学伯克利分校是最早开始 RISC-V 的开发，并且经过多次优化完善最终形成的第 5 代精简指令集 (RISC)，并于 2014 年发布，它集百家之长吸收了 ARM、MIPS、x86 和 PowerPC 的丰富经验，是一个经过模块化的可扩展可以面对不同的应用场景的指令集，它可以通过组合他的不同模块满足不同需求，而且像 ARM 的传统指令集是不可以进行指令集扩展的，RISC-V 是支持指令扩展的，最关键的还是 RISC-V 指令架构是没有历史包袱的，具体体现在以下几个方面，1. 手册页数比较少，与 x86 和 ARM 架构的手册达到几千页的数量不同 RISC-V 架构的手册也就区区不到三百页；2. 指令数目上，不同于 x86 和 ARM 的指令数那么多并且繁杂，在加上由于历史原因他们不同架构之间的不同分支也相互彼此不兼容。而 RISC-V 没有历史包袱，在设计指令集的时候重新从头开始设计，最终只包含 40 多个基础指令，并根据这些做为公共部分，再扩展出其他常用的模块指令。实现起来比较简单，在硬件平台上实现也不是很难。

虽然我们的学校的确与美帝之间存在差距，但是这都不算什么，因为就算他们现在比我们发展要超前也这不代表以后永远都会领先我们，如果我们把精力，资源放在这方面，假以时日我们是会迎头赶上的，也可以做出属于自己的产业链。

目前国外 SiFive 公司服务做的不错，主要业务是提供商业化的针对 RISC-V 指令集架构设计的应用处理器的 IP、相关配套的软件开发平台以及与芯片相关的多种应对策略。SiFive 在 2018 年注册独立公司 SaiFan 在中国运行，来服务于中国市场为客服提供服务；另一家国外公司——Green WavesIoT，该公司主营功耗低基于 RISC-V 的用于边缘应用的 IP 和处理器 [1]。

我们之前说了很多关于真正实现国产化指令集的设想，现在 RISC-V 指令集架构就是这样的机遇。以前的国产芯片领域的开发基本上都需要得到国外公司的授权的才能开发，典型的就 ARM 架构，我们的发展花费了大约有十几年的时间来壮大，但重要的是，这些指令集架构是属于外国公司，并不是免费白让我们使用，从根本上来讲是需要他们这些大公司来允许我们使用，如果我国以后使用 RISC-V 的话，一方面是可以省去把用来给外国公司交的授权费，用来放在更好的科研攻关上面；另一方面，外国企业可以随时完全停止合作授权的时代一去不复返了。相比较我们民族大力发展来让自己实现一套自主的指令集架构，成本高，技术难度大，所以这件事本身又没有太大的经济技术应用价值，因为开发出来一个处理器是要在整个世界上流通的，这样才有可能让各地的软件生态加入建设中来。所以 RISC-V 的横空出现，就可以很好的处理相应的关系。

国内的芯来科技模仿 SiFive 做的也不错，提供多个 RISC-V IP，并同时提供蜂鸟 E203 开发板；位于中国台湾的公司——Andes Technology 同样推出基于 RISC-V 的芯片产品；在 2018 年 4 月，阿里的平头哥收购了中天微并展开了自己的研究，并在 2021 年发布了目前世界范围内性能最优的 RISC-V 芯片玄铁 910，并且可以与 Android OS 衔接完美，这表明目前完善的 Android 生态环境可以运行在国产 RISC-V 芯片上了，换句话说软件的生态已经基本解决了。

最后 RISC-V 对 IoT 的发展也同样巨大，现在碎片化是物联网和边缘计算的发展表象，而且以应用为核心的发展方向也逐步成为发展趋势，不同以往传统的围绕着芯片，因为这样的发展方式就会导致以发展模组和应用的公司为核心的，而替代了从前那种以芯片公司为核心的发展方式，传统方式已经面对如今的发展趋势表现出力不从心了。我们以 ARM 为例，其特点是不仅价格不亲民每次发布还挺长时间的，如果用于物联网相同的使用场景，就会在市场上趋近相同的竞争，加上成本高等因素，导致大家都不愿意去开发，最终除了少数大企业才能最早买到 IP，其他公司只能望尘莫及，这样的后果就是 ARM 的发展很难去高效的来应对碎片化应用需求。

1.3 本文研究内容

本文主要参考孙卫真，刘雪松，朱威浦和向勇老师的《基于 RISC-V 的计算机系统综合实验设计》[2] 上面的内容在 Qemu 模拟器上将 linux 试图移植到 RISC-V 的平台上面，并逐步实现实验环境的搭建，交叉工具链的编译，完善文件系统，安装网络协议栈等现代操作系统应该具备的基本功能，并解决在此过程中遇到的种种挑战，种种报错以及解决方案，最后运行一个 RISC-V 指令集的简单程序验证一下是否移植成功。

虽然，已经国内已经有人做出了基于 RISC-V 处理器内核，linus 也把 riscv-linux 合并到 Linux 的主分支上了，但是我这个毕设的内容就是自己想进行一遍，是验证性的实验，虽然我也知道我现在做的这些事情看来有些微不足道，但是我想既然我已经努力开始做了，就已经具有了加速度，根据 $v = v_0 + at$ 让子弹再飞一会儿，最终就会获得丰收的成绩。

第 2 章 计算机组成与设计——基于 RISC-V

依据 David A. Patterson 的著作 Computer Prganization and Design RISC-V Edition [3] 以及浙江大学刘鹏博导的公开课 [4] 作为参考理解，这里主要讨论 RISC-V 的指令集以及流水线。

2.1 指令表示方法与指令格式

其实抛开课堂上 PPT 枯燥的概念，如果把计算机比作一个正在说话的人，那么指令就是那个人每句话中的每一个单词，而指令集就是全部词汇所构成的词汇表，也正是有指令集的存在，计算机才能实现“开口说话”与沟通，接下来介绍一下 RISC-V 的常用指令。

2.1.1 R 指令

R 指令的作用是用于处理处理器内部的寄存器与寄存器之间算数运算产生的。

一条 R 型指令的格式划分如图2- 1所示，一个 R 型指令可以被划分成为 6 个域，其中 31-25 这 7bit 宽的是功能码 7，24-20 这 5 位是源寄存器 2，19-15 这 5 位是源寄存器 1,14-12 这 3bit 宽的是功能码 3，11-7 这 5bit 宽的目的寄存器 rd，6-0 这 7 位宽的操作码 (部分规定了该指令是什么指令)。

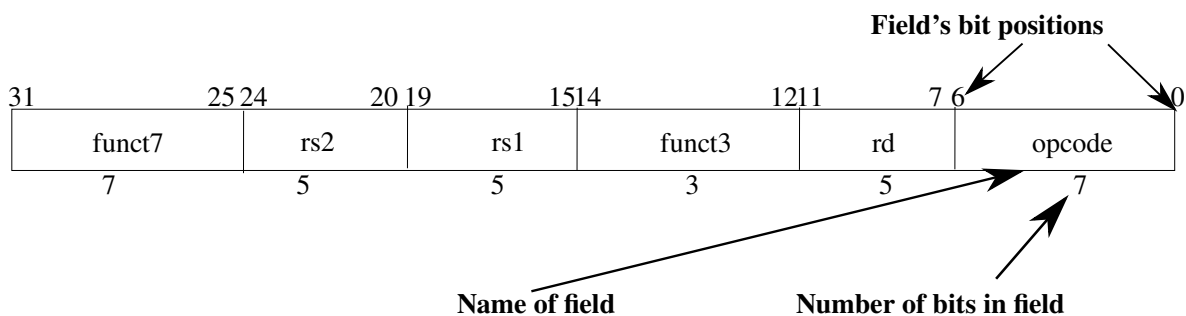


图 2- 1 R 型指令格式的划分，其中每个域中的文字表示该域的名称；域上的数字表示，该域的起始与终止位置；每个域下面的数字，表示每个域所占据的 bit 位数

所有 R 指令的 opcode 部分都是 2 进制数：0110011，funct7 与 funct3 是和 opcode 结合起来一起使用的，三者组合一起规定指令的操作。

rs1, rs2 是源寄存器 (Source Rggister)，是用于暂存源操作数的地址；rd 是目的寄存器 (Destination Rggister)，指定接收结果值的寄存器编号，这三个寄存器都存放 5 位无符号整数 (对应十进制 $2^5 = 32$)，对应 0 31 的通用寄存器中的其中一个。

2.1.2 I 指令

I 指令是用于寄存器与立即数之间算数运算和读取产生的。

如果换成是我自己设计指令架构，我很有可能将操作寄存器，操作数用一个指令全部实现，但是我看了别人的设计之后发现我以前的想法中存在一个问题，就是指令中的位长，比方说 R 指令的源地址和目的地址才 5bit，最多就能表示 10 进制的 32 位，那要是操作数

字就显得有些有限；但是如果你把指令中的位长加长的话，就会造成可以表示的 10 进制数太多，但是寄存器就 32 了，造成不必要的浪费。

所以，I 型指令可以在 R 型指令的基础上做一些稍微的改动即可，如图 2- 2

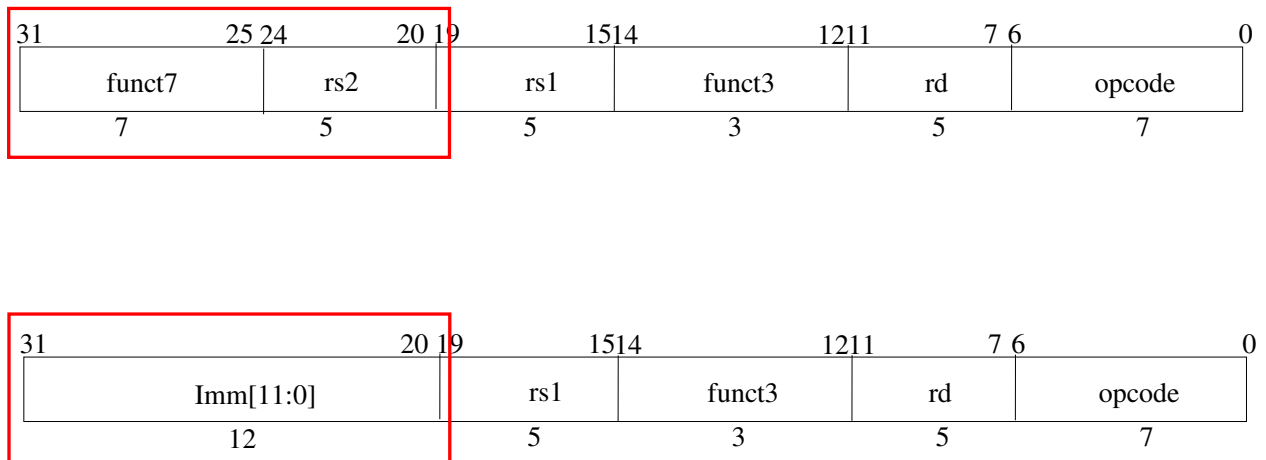


图 2- 2 I 型指令就是把 R 型指令的前两个域合并成一个了 12 位的有符号数

2.1.3 S 指令与 B 指令

S 指令是用于写存储器的。

B 指令是用于分支转移操作的，其实本质上是 S 指令的一个变体，之前也叫 **SB 指令**

2.1.4 U 指令和 J 指令

U 指令用于高 20-bit 位立即数操作。

J 指令是用于跳转操作，其实是 U 指令的一个变体，之前也叫 **UJ 指令**

2.2 流水线

2.2.1 处理器性能度量方法

首先，我认为在我们研究之前应该明确我们常说的提高性能是指什么，是表示更快的响应时间？从而更快的完成需要执行的任务？还是单位时间内能完成更多的任务？还是使用寿命更长一些？

我觉得性能的本质是——处理器干起活来利不利索，如公式 (2- 1) 是与处理器性能有关的定理。

$$\frac{Time}{Program} = \frac{Instructions}{Program} * \frac{Cycles}{Instruction} * \frac{Time}{Cycle} \quad (2- 1)$$

2.2.2 结构冒险

本质上讲造成出现结构冒险的原因就是因为在硬件层面上是不支持同一周期来执行多条指令的。

但出现这个结构冒险，也比较好解决，既然是硬件层面的问题，那就从硬件的方向来入手解决 1. 让其他阻塞一下等待硬件空闲下来在使用，2. 要么就是多添加点硬件。

2.2.3 数据冒险

导致数据冒险发生简言之就是不同指令之间存在**数据的关联**，造成了无法提供指令执行所需的数据，进而指令不能在预期的时钟周期内执行。

解决方案就是 1. 让其他阻塞一会儿等待数据使用完在继续使用；2. 增加一个**旁路 (bypassing)**，这样的好处就是不需要浪费时间等指令执行完成就可以立即解决数据冒险。如图2-3 所示，第一条 add 指令执行 EX 阶段的输出前递到 sub 指令的 EX 阶段的输入，替换 sub 指令在第二阶段督促的寄存器 X1 的值，这样就很好的解决了数据冒险。

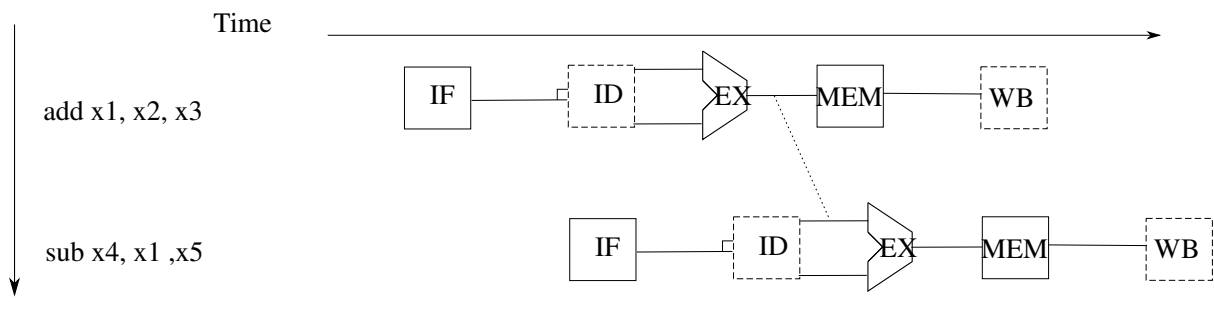


图 2-3 这张图片就显示了旁路的效果

旁路看起来挺完美的，好像只需要加上一条新的电路，这样就可以节省出那么多的时间，然后利用这省出的时间可以完成更多的指令，这听起来的确太完美了，但是现实往往让人比较无奈，有时尽管已经使用了旁路，还是会出现不可避免的需要**流水线停顿 (pipeline Stall)**，俗称**气泡 (bubble)**，图2-4就是在 load 指令执行之后，紧跟着一条需要使用他结果的 R 型指令，可惜 R 指令需要到执行运算的时候才需要上一条 load 指令的值，所以这里就只能添加 bubble 来等待。

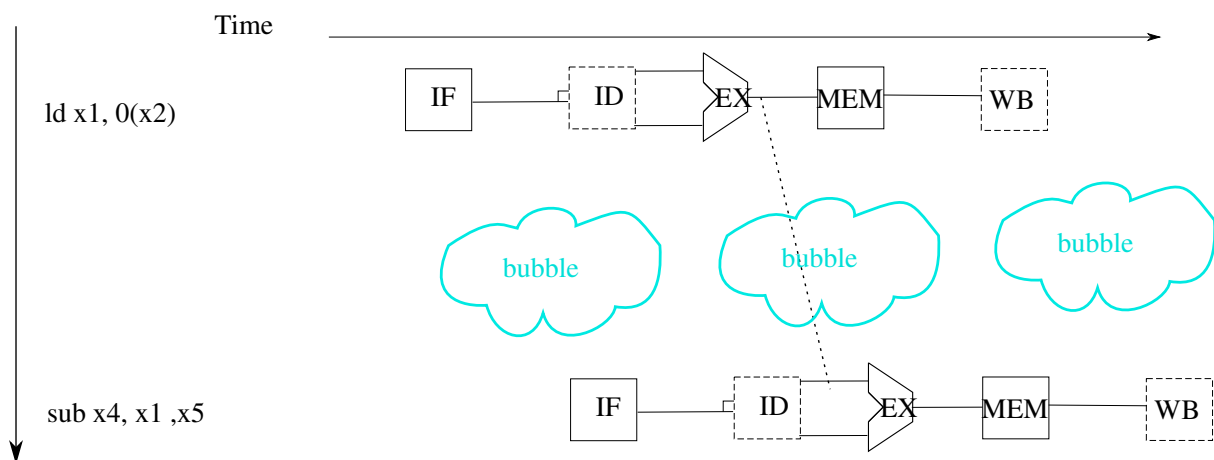


图 2-4 这张就表示有时即使使用旁路，也许要阻塞等待

2.2.4 控制冒险

这种冒险发生在根据一条指令的结果判断接下来要执行的分支程序。如图2- 5所示

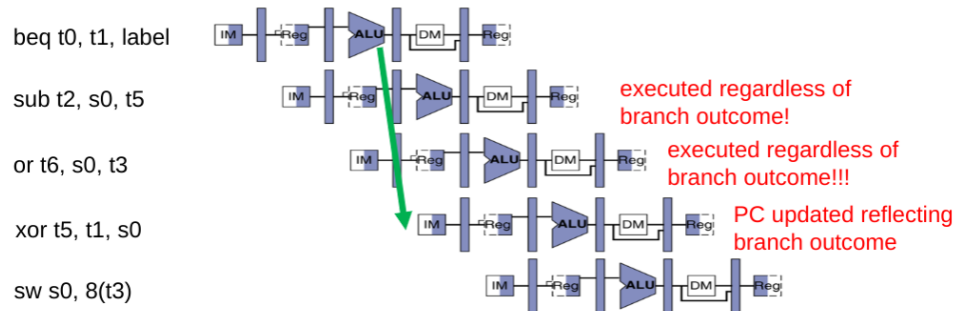


图 2- 5 控制冒险

解决方法 1. 阻塞等待, 2. 采用**分支预测**, 要是预测对了, 就继续执行, 如图2- 6; 如果预测错了, 流水线清空刚才加载错误的指令, 重新在装载正确的指令, 如图2- 7。

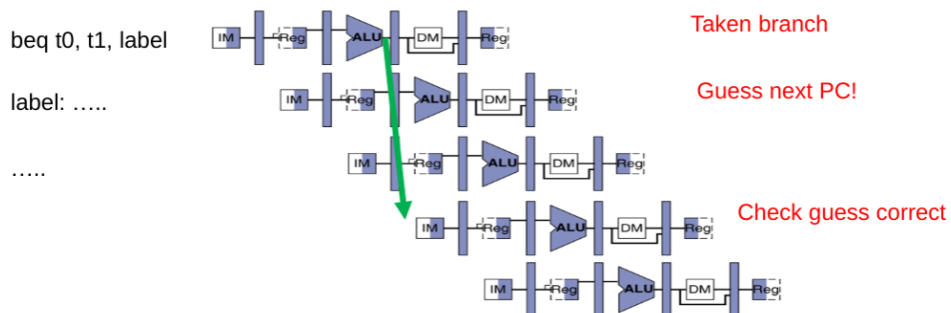


图 2- 6 分支预测：预测成功

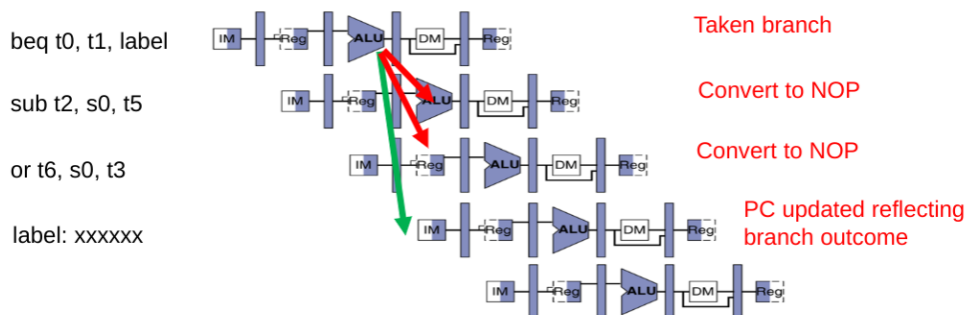


图 2- 7 分支预测：预测失败

第 3 章 Qemu & KVM 基本原理

3.1 虚拟化

在维基百科中虚拟化的解释为：“In computing, virtualization refers to the act of creating a virtual(rather than actual)version of something, including virtual computer hardware platforms, storage devices, and computer network resources.” [5], 简单来说, 计算机模拟出一个虚拟并不存在的（而非实际）计算机实体就是虚拟化, 其中包括虚拟出来的计算机硬件平台、存储设备, 以及网络资源。由此可见, 虚拟化技术是一种将硬件资源（如处理器、Memory、storage、NetWork resource etc）提取出来, 最终成为一个虚拟的资源池, 并对其提供分割、重新组合的操作, 以达到资源利用最大化。

3.1.1 软件虚拟化

本文所介绍的是使用 Qemu 来实现 VMM 层, 即仅使用软件实现的方式来运行客户机要执行的指令。Qemu 工作方式是在软件层面实现对指令的二进制翻译, 简单来说就是 Qemu 是一个翻译官, 它将使用某套指令集的二进制编码翻译成基于另一套指令集的编码并最终输出到最终架构下的代码交给客户机, QEMU 会拦截每一条客户机执行的指令, 并转换成宿主机处理器架构的指令, 最后由宿主机的物理硬件来执行相应的指令, 由此往复完成虚拟化的实现。

3.1.2 硬件虚拟化

这里以通用的 x86 体系结构为例, Intel 在 2005 年就加入硬件虚拟化的支持——Intel VT。这也就以为着硬件平台本身支持客户机与宿主机指令间相互转换并执行, 而不再像以前那样需要 VMM 截获重定向这种软件实现方式。

其实我们通过纯粹用软件中的模块管理层面上直接对管理整个客户机服务的进程就是通过要在整个 linux 上的一个内部物理执行进程, 客户机正常执行访存操作时, 实际上我们就是通过 Linux 上对内核管理的虚拟内存, 以前是使用 linux 中的内核软件来实现 GVA(客户机虚拟地址) 到 GPA(客户机物理地址) 再到 HVA(宿主机虚拟地址), 最后到 HPA(宿主机物理地址) 的这四步的转换, 这个机制也称为“影子页表”。但是原本就是访问一个地址很简单的操作却大费周折要经过四步执行, 对于计算机来说, 这样的代价特别大, 所以后来这种靠软件实现的方式逐步被硬件逻辑取代, 这种硬件逻辑就是 Intel 的 EPT 技术(或者 AMD 的 NPT 技术), 靠硬件自动算出 GPA 到 HPA 的过程, 转换的过程一步到位。

3.2 硬件虚拟化介绍

3.2.1 CPU 虚拟化

根据上文的介绍, 软件实现虚拟化的成本代价比较大, 所以像 Intel 这样的硬件厂商慢慢地在硬件层面上实现虚拟化。

关键技术——VMX（virtual-machine extensions），这项技术就是 Intel 在处理器层面上提供了对实现虚拟化在技术上的支持。根据查询 Intel 的官方文档 [6] 可知 VMX 有两种操作模式：VMX 根操作（root operation）与 VMX 非根操作（non-root operation）。

简单的区分，在 VMX 根操作模式运行 KVM 下，而 VMX 非根模式下运行的是整个运行在虚拟机中的客户机完整的软件栈（包括操作系统和应用程序）。VMX 根操作模式与非 VMX 模式之间是可以相互转化的，当从 VMX 跟模式进入 VMX 非根操作模式被称为“VM Entry”；从 VMX 非根操作模式退出到 VMX 根操作模式，被称为“VM Exit”。

可能都是 Intel 发明的缘故，VMX 的根操作模式与非 VMX 模式像极了 x86 处理器的各种执行模式，区别只是 VMX 的根操作现在已经支持了新的 VMX 相关的指令集以及一些对相关控制寄存器的操作。而由于 VMX 的非根操作模式是一个相对受限的执行环境，为了更好地适应在虚拟化中的变化而专门进行了一定的修改；VMX 的根操作模式与非 VMX 模式之间的转换如同处理器的各种门的特权集转换一样，当客户机中执行的一些特殊的敏感指令或者一些异常会触发“VM Exit”退到虚拟机监控器中，从而运行在 VMX 根模式。这样的行为看似很麻烦，但是也正是因为这样的限制，才让虚拟机监控器保持了对处理器资源的更好的控制。

为了更好的理解，VMM 与 Guest 之间以及 VMX 的根操作模式与非根操作模式是如何转换的，交互一个虚拟机监控器软件的最基础的运行生命周期及其与客户机的交互如图 3-1 所示。软件想进入 VMM 操作模式，需要通过执行 VMXON 指令才能进入；当已经进入 VMM 模式以后，想在进入客户机执行模式，即 VMX 非根模式，需要通过执行 VMLAUNCH 和 VMRESUME 指令进入；当在 VMX 非根模式下触发 VM Exit 时，处理器执行控制权会再次回到宿主机的虚拟机监控器上；最后虚拟机监控可以通过执行 VMXOFF 指令退出 VMX 执行模式。

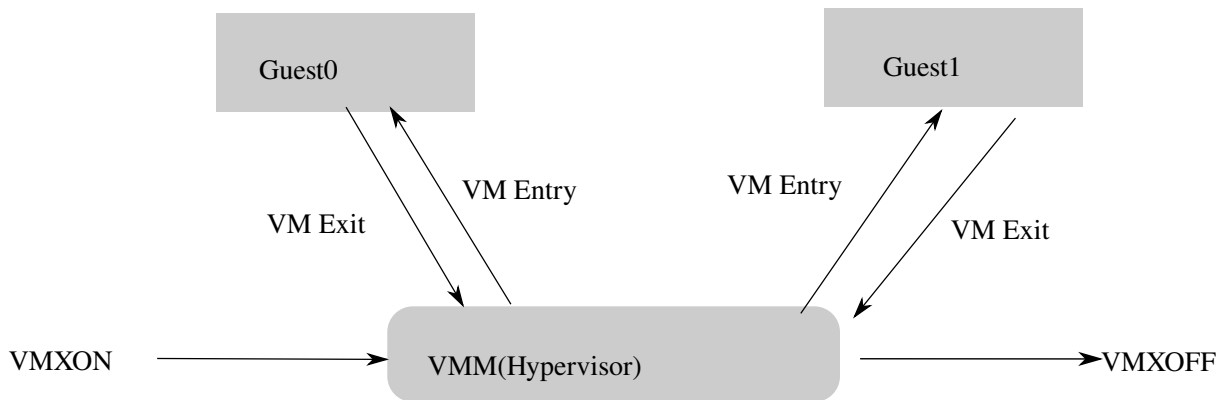


图 3-1 VMM 与 Guest 之间的交互

如图3-1中需要完成的转换过程,在该过程中是通过使用 VMCS（virtual-machine control data structure）的数据结构来实现逻辑处理器完成根模式和非根模式之间的切换的；而处理器是通过使用 VMCS 指针来进行访问 VMCS 结构的，并对其进行操作。其实 VMCS 指针其实也没什么神奇的，本质上就是一个指向 VMCS 结构的 64 位的地址，可以通过使用 VMPTRST 和 VMPTRLD 指令对 VMCS 指针进行读写，使用 MREAD、VMWRITE 和 VMCLEAR 等指令对 VMCS 实现配置。

对于一个逻辑处理器，表面上看它是可以一起维护多个 VMCS 数据结构，但是具体到某一时刻的时候，就只有一个 VMCS 在当前真正生效。多个 VMCS 之间也是可以相互切换的，VMPTRLD 指令就让某个 VMCS 在当前生效，而其他 VMCS 就自然成为不是当前生效的。一个虚拟机监控器会为一个虚拟客户机上的每一个逻辑处理器维护一个 VMCS 数据结构。

3.2.2 内存虚拟化

内存虚拟化的目的是给虚拟客户机操作系统提供类似实际的物理存储空间——即一个从 0 地址开始的连续物理内存空间，但是他不仅仅就是提供地址空间，还要同时在多个客户机之间实现隔离和调度。

在虚拟环境下内存地址如图 3-2 所示。

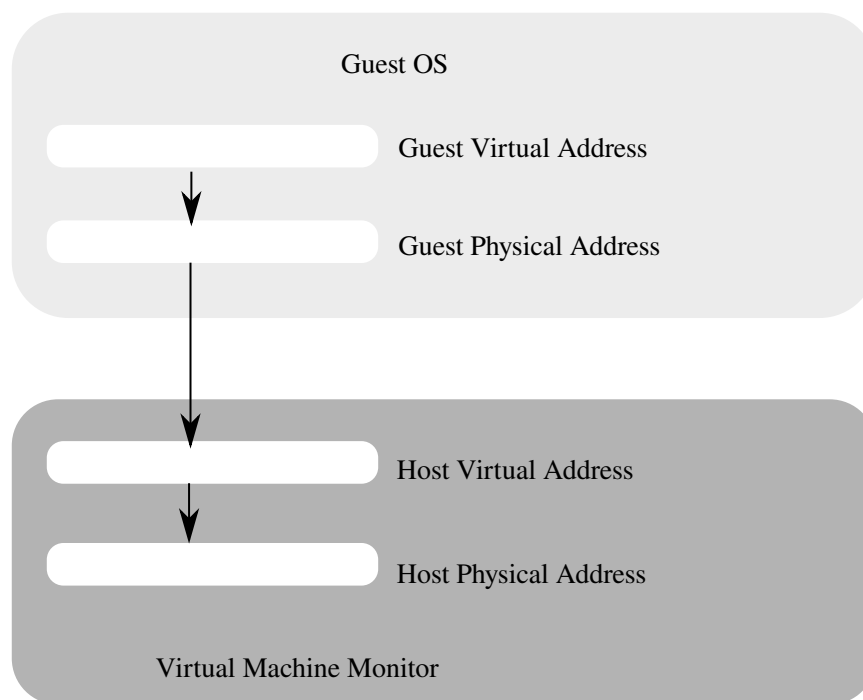


图 3-2 虚拟化环境下的内存地址

内存虚拟化就是要将客户机虚拟地址（GVA）转化为最终能够访问的宿主机上的物理地址（HPA）。对于客户机操作系统而言，它不可能感知到有内存虚拟化的存在，在应用程序访问虚拟地址时，通过 CR3 寄存器可以将其转化为物理地址，但是在虚拟化环境中这个物理地址只是客户机的物理地址，还不是真实内存硬件上的物理地址。所以，虚拟机监控器就非常需要维护从物理客户机虚拟地址映射到客户宿主机上的物理虚拟地址之间的一个动态映射空间关系，在没有硬件提供的内存虚拟化之前，这个维护映射关系的页表叫作影子页表（Shadow Page Table）。内存的访问和数据的更新通常是非常频繁的，加之需要一直维护影子页表中的对应关系将会变得非常复杂，开销也较大。同时它还需要为每一个运行的客户机都维护一份影子页表，当客户机数量较多时，其影子页表也会越来越大，占用的内存较大也会是一个问题。

Intel CPU 在整个硬件上的设计就引入了 EPT（Extended Page Tables，扩展页表），从

而将一个客户机虚拟地址到宿主机物理地址的转换通过硬件来实现。

3.2.3 I/O 虚拟化

在虚拟化的架构下，虚拟机监控器必须支持来自客户机的 I/O 请求。通常情况下有以下 4 种 I/O 虚拟化方式。

1) 设备模拟：在虚拟机监控器中模拟一个传统的 I/O 设备的特性，比如在 QEMU 中模拟一个 Intel 的千兆网卡或者一个 IDE 硬盘驱动器，在客户机中就暴露为对应的硬件设备。客户机中的 I/O 请求都由虚拟机监控器捕获并模拟执行后返回给客户机。

2) 前后端驱动接口：在虚拟机监控器与客户机之间定义一种全新的适合于虚拟化环境的交互接口，比如常见的 virtio 协议就是在客户机中暴露为 virtio-net、virtio-blk 等网络和磁盘设备，在 QEMU 中实现相应的 virtio 后端驱动。

3) 设备直接分配：将一个物理设备，如一个网卡或硬盘驱动器直接分配给客户机使用，这种情况下 I/O 请求的链路中很少需要或基本不需要虚拟机监控器的参与，所以性能很好。

4) 设备共享分配：其实是设备直接分配方式的一个扩展。在这种模式下，一个（具有特定特性的）物理设备可以支持多个虚拟机功能接口，可以将虚拟功能接口独立地分配给不同的客户机使用。如 SR-IOV 就是这种方式的一个标准协议。

表3- 1展示了这 4 种 I/O 虚拟化方式的优缺点，前两种都是纯软件的实现，后两种都需要特定硬件特性的支持。

	优点	缺点
设备模拟	兼容性好，不需要额外驱动	1. 性能较差 2. 模拟设备的功能特性支持不够多
前后端接口	性能有所提升	1. 兼容性差一些：依赖客户机总安装特定驱动 2. I/O 压力大时，后端驱动的 CPU 资源占用较高
设备直接分配	性能非常好	1. 需要硬件设备的特性支持 2. 单个设备只能分配一个客户机 3. 很难支持动态迁移
设备共享分配	1. 性能非常好 2. 单个设备可共享	1. 所需设备硬件的特性支持 2. 很难支持动态迁移

表 3- 1 常见 I/O 虚拟化方式的优缺点

3.3 KVM & Qemu 模拟器介绍

首先 Qemu(Quick Emulator) 本身并不完全是 KVM 的一部分，它是一套由软件模拟实现的。

而 KVM(Kernel Virtual Machine) 是有两部分组成，一部分是 Linux 内核的 KVM 模块，另一块是经过简化后的 Qemu。有了 KVM 的 linux 主机将变成一个 Hypervisor（虚拟机监控器）。在 x86 处理器支持 VMX（Virtual Machine Extension）功能以后，Linux 在原有的用户模式和内核模式中又新增加了客户模式，并且客户模式也拥有自己的内核模式和用户模式，其中虚拟机部分就是运行在客户模式中。三层结构如图3- 3所示。

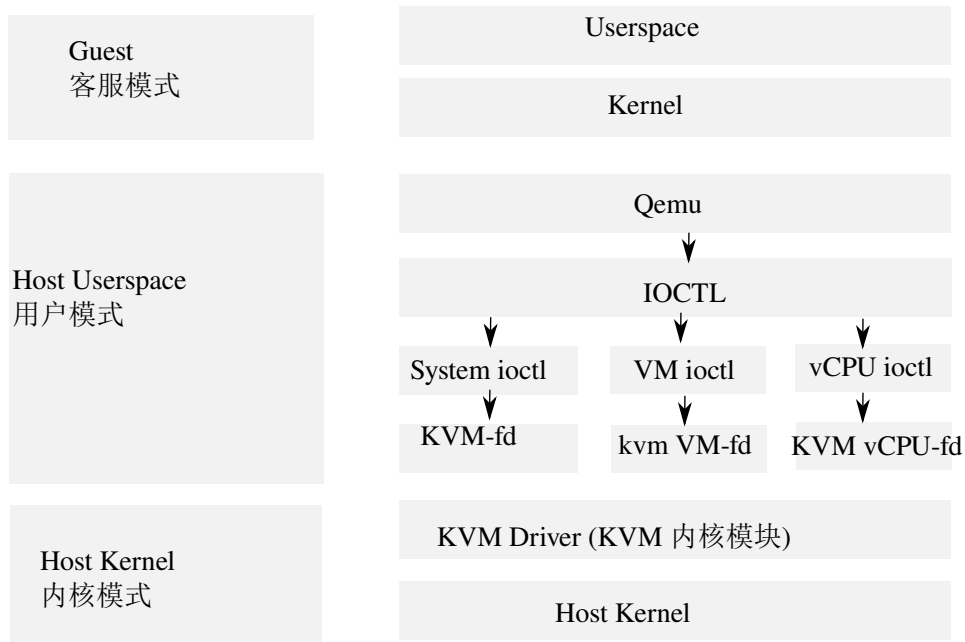


图 3-3 KVM 三种模式的层次关系

KVM 就是在硬件辅助虚拟化技术之上构建起来的虚拟机监控器。

当然，并非要所有这些硬件虚拟化都支持才能运行 KVM 虚拟化，KVM 对硬件最低的依赖是 CPU 的硬件虚拟化支持。

3.3.1 KVM 内核模块

KVM 模块是 KVM 虚拟化的核心部分，它在内核是由两部分组成：其中一个部分是与处理器架构无关的部分，用 `lsmod` 命令中可以看到，如图 3-4 所示，叫作 `kvm` 模块；另一个部分是处理器架构相关的部分，本文是在 Intel 平台上实验，所以看到的的就是 `kvm_intel` 这个内核模块。KVM 的主要工作就是初始化硬件资源，开启虚拟化模式，将客户机运行在虚拟机模式中，虚拟客户机运行时提供一定的支持。

```

[tony@tony-tm1701 ~]$ lsmod | grep kvm
kvm_intel          327680  0
kvm                933888  1 kvm_intel
irqbypass         16384   1 kvm
    
```

 图 3-4 `lsmod`

以 KVM 在 Intel 的 x86 系列的 CPU 上运行为例，在内核加载以后，KVM 模块首先会初始化内部的数据结构；完成一切准备工作以后，KVM 模块会继续检测系统当前的 CPU，然后打开 CPU 控制寄存器 CR4 中的虚拟化模式开关，并通过执行 `VMXON` 指令将宿主操作系统（包括 KVM 模块本身）置于 CPU 执行模式的虚拟化模式中的根模式；最后，KVM 模块创建特殊设备文件 `/dev/kvm` 来等待从用户空间发过来的命令。接下来，虚拟机的创建和运行将是一个用户空间的应用程序（QEMU）和 KVM 模块相互配合的过程。

/dev/kvm 这个设备文件是一个标准的字符设备，KVM 模块与用户空间 QEMU 的通信接口主要是一系列针对这个特殊设备文件的 `ioctl` 调用。当然，每个虚拟客户机针对 /dev/kvm 文件的最重要的 `ioctl` 调用就是“创建虚拟机”。在这里，“创建虚拟机”可以理解成 KVM 为了某个特定的虚拟客户机（用户空间程序创建并初始化）创建对应的内核数据结构。同时，KVM 还会返回一个文件句柄来代表所创建的虚拟机。针对该文件句柄的 `ioctl` 调用可以对虚拟机做相应的管理，比如创建用户空间虚拟地址和客户机物理地址及真实内存物理地址的映射关系，再比如创建多个可供运行的虚拟处理器（vCPU）。同样，KVM 模块会为每一个创建出来的虚拟处理器生成对应的文件句柄，对虚拟处理器相应的文件句柄进行相应的 `ioctl` 调用，就可以对虚拟处理器进行管理。

“执行虚拟处理器”是虚拟处理器最重要的 `ioctl` 调用，通过该调用，可在 KVM 模块的支持下将用户空间准备好的虚拟客户机放置在虚拟化模式中的非根模式，并执行相应的二进制指令。在非根模式下执行二进制指令时，如果出现敏感指令会被处理器理科捕捉到，然后保存现场并切换到根模式，接下来的操作就有 KVM 来决定（一种方法是在 KVM 模块直接处理，另一种方法是返回到用户空间交让用户空间的程序来进行处理）。

除了处理器的虚拟化以外，内存虚拟化也是由 KVM 模块实现的，包括上文中提到的使用硬件提供的 EPT 特性，通过两次的地址转换机制最终实现从客户机虚拟地址到宿主机物理地址之间的转换。

处理器主要是通过使用 I/O 指令和 MMIO 来实现的对设备的访问，其中 I/O 指令是处理器直接截获进行处理，而 MMIO 会通过配置内存虚拟化来捕捉。但是，模拟外设的模块一般不是采用 KVM 模块来负责，因为只有像虚拟中断控制器和虚拟时钟这种对性能要求比较高的虚拟设备才会由 KVM 内核模块来直接负责，因为这样做可以大量减少因处理器模式来回切换的开销。而大部分的输入输出设备交给下一节将要介绍的用户态程序 QEMU 来负责。

3.3.2 QEMU 用户态设备模拟

QEMU 原本就是一个著名的开源免费的用于虚拟化的软件项目，并不是 KVM 软件的一部分。不同于 KVM，QEMU 最初实现的是一个纯软件的虚拟机，通过使用二进制翻译来实现对虚拟的客户机中 CPU 指令模拟执行，所以实际性能比较低。但是，它能发展到今天主要是因为他有一个显著特点——跨平台，QEMU 支持在 Linux、Windows、FreeBSD、Solaris、MacOS 等多种不同的操作系统上运行，能支持在 QEMU 本身编译运行的平台上就实现虚拟机的功能，甚至可以支持客户机与宿主机并不是同一个架构（比如在 x86 平台上运行 ARM 客户机）。作为一个存在已久的虚拟机监控器软件，QEMU 的开发代码中有完整的虚拟机实现，包括处理器虚拟化、内存虚拟化，以及 KVM 也会用到的虚拟设备模拟（比如网卡、显卡、存储控制器和硬盘等）。

除了二进制翻译的方式，QEMU 也能与基于硬件虚拟化的 Xen、KVM 结合，为它们提供客户机的设备模拟。通过与 KVM 的密切结合，让虚拟化的性能提升得非常高，在真实的企业级虚拟化场景中发挥重要作用，所以我们通常提及 KVM 虚拟化时就会说“QEMU/KVM”这样的软件栈。

为了简化代码的架构以及让代码复用，KVM 的开发者根据 KVM 的特性在原有的 QEMU 基础上进行了修改（后来这些代码被合并到 QEMU 开发主线代码中了，所以现在 QEMU 已经可以原生支持 KVM 虚拟了）。虚拟机在运行时，QEMU 会通过 KVM 模块提供的系统调用进入内核，然后 KVM 模块将其置于处理器的特殊模式下运行。

但是站在 QEMU 的立场而言，QEMU 使用了 KVM 模块的虚拟化功能，来为自身提供硬件虚拟化的加速，从而很大幅度的改进了虚拟机的性能。除此之外，配置和创建虚拟机，虚拟机运行时依赖的虚拟设备、用户交互界面、运行环境，还有一些针对虚拟机的高级技术像动态迁移技术，热备份等等也都是 QEMU 自身完成实现的。

QEMU 除了提供完全模拟的设备（如：e1000 网卡、IDE 磁盘等）以外，还支持 virtio 协议的设备模拟。virtio 是一个沟通客户机前端设备与宿主机上设备后端模拟的比较高性能的协议，在前端客户机中需要安装相应的 virtio-blk、virtio-scsi、virtio-net 等驱动，而 QEMU 就实现了 virtio 的虚拟化后端。QEMU 还提供了叫作 virtio-blk-data-plane 的一种高性能的块设备 I/O 方式，它最初在 QEMU 1.4 版本中被引入。virtio-blk-data-plane 与传统 virtio-blk 相比，它为每个块设备单独分配一个线程用于 I/O 处理，data-plane 线程不需要与原 QEMU 执行线程同步和竞争锁，而且它使用 ioeventfd/irqfd 机制，同时利用宿主机 Linux 上的 AIO（异步 I/O）来处理客户机的 I/O 请求，使得块设备 I/O 效率进一步提高。

总之，QEMU 既是一个功能完整的虚拟机监控器，也在 QEMU/KVM 的软件栈中承担设备模拟的工作。

3.4 KVM 上层管理工具

虽然本文涉及到的 KVM 上层管理软件比较少，但是我觉得还是将 KVM 作为一个完整的整体来处理，这里介绍一下我在阅读 KVM 官网信息的时候遇到常用的管理工具。

3.4.1 libvirt

libvirt 是使用最广泛的对 KVM 虚拟化进行管理的工具和应用程序接口，已经是事实上的虚拟化接口标准，本节后部分介绍的其他工具都是基于 libvirt 的 API 来实现的。作为通用的虚拟化 API，libvirt 不但能管理 KVM，还能管理 VMware、Hyper-V、Xen、VirtualBox 等其他虚拟化方案。

3.4.2 virsh

virsh 是一个常用的管理 KVM 虚拟化的命令行工具，对于系统管理员在单个宿主机上进行运维操作，virsh 命令行可能是最佳选择。virsh 是用 C 语言编写的一个使用 libvirt API 的虚拟化管理工具，其源代码也是在 libvirt 这个开源项目中的。

3.4.3 virt-manager

virt-manager 是专门针对虚拟机的图形化管理软件，底层与虚拟化交互的部分仍然是调用 libvirt API 来操作的。virt-manager 除了提供虚拟机生命周期（包括：创建、启动、停止、

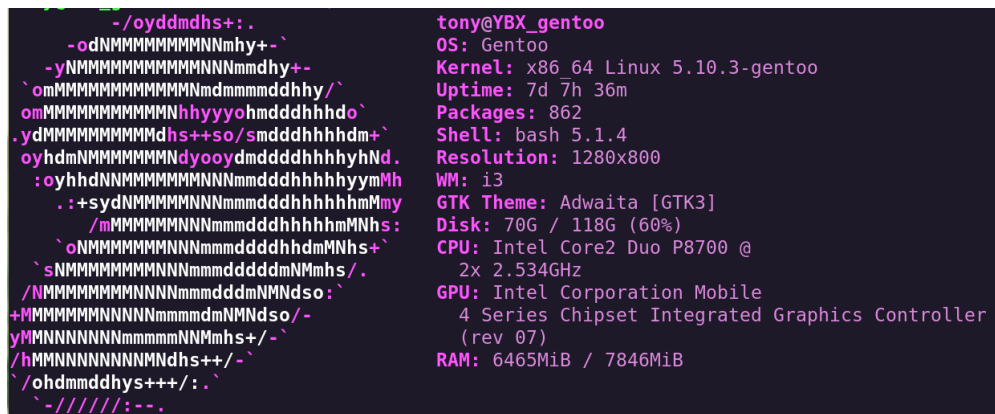
打快照、动态迁移等）管理的基本功能，还提供性能和资源使用率的监控，同时内置了 VNC 和 SPICE 客户端，方便图形化连接到虚拟客户机中。`virt-manager` 在 RHEL、CentOS、Fedora 等操作系统上是非常流行的虚拟化管理软件，在管理的机器数量规模较小时，`virt-manager` 是很好的选择。因其图形化操作的易用性，成为新手入门学习虚拟化操的首选管理软件。

第 4 章 实验流程

本文是依据 Github 上的很多年以前的项目——BusyBear [7] 来实现的，但是该项目已经有四年没有更新了，在实验过程中，本人发现说明文档中很多步骤以及脚本内容已经不适用了，由于软件的更新迭代还出现很多新问题，比方说当时采用 Linux Kernel v4.4，当时 Linux 主分支不支持 RISC-V，但是本文采用现在主流的 Linux Kernel 版本 Linux Kernel v5.*，该版本的 Linux 主分支已经支持了 RISC-V 架构了；还有就是 Qemu 在 4.* 的时候需要自己编译 OpenSBI，但是在 Qemu 5.* 以后就内嵌 OpenSBI，一些流程也有所改变，但是清楚 Linux 启动流程就可以了，详细参考《从按下电源开始的一场接力赛》[8]。

4.1 前期准备

本实验环境是在 Intel 的 x86 平台，实验运行系统是 Gentoo linux [9]，系统具体信息如图4-1 所示。



```

-/oyddmdhs+:.
-odNNNNNNNNNNmhy+-`
-yNNNNNNNNNNNNmmdhy+-
`omNNNNNNNNNNNNmmdmmdhhy/`
omNNNNNNNNNNNhyyyohmdddhhd`
.ydNNNNNNNNMdh++so/smdddhhdmd+`
oyhdmNNNNNNNNdyooydmddddhhyhNd.
:oyhhdNNNNNNNNmmdddhhyymMh
.:+sydNNNNNNNNmmdddhhyhmMmy
/mNNNNNNNNmmdddhhyhmMNs:
`oNNNNNNNNmmdddhdmMNs+`
`sNNNNNNNNmmddddmMmhs/.
/NNNNNNNNNmmdddmNMNdso:~
+NNNNNNNNNNmmdmNMNdso/-
yNNNNNNNNmnmNMmhs+/-`
/hNNNNNNNNNMdh++/-`
`ohdmdhys+++:.
`-////////:--.

tony@YBX_gentoo
OS: Gentoo
Kernel: x86_64 Linux 5.10.3-gentoo
Uptime: 7d 7h 36m
Packages: 862
Shell: bash 5.1.4
Resolution: 1280x800
WM: i3
GTK Theme: Adwaita [GTK3]
Disk: 70G / 118G (60%)
CPU: Intel Core2 Duo P8700 @
2x 2.534GHz
GPU: Intel Corporation Mobile
4 Series Chipset Integrated Graphics Controller
(rev 07)
RAM: 6465MiB / 7846MiB
  
```

图 4-1 实验操作系统

4.1.1 交叉工具链的准备

本文我使用的是 GNU 提供的交叉工具链，实验前需要安装并配置好实验所需的工具链，并根据仓库的 README.md 文档，在 x86 平台交叉编译出 RISC-V 所需的工具链，并进行安装与配置 [10] [11] [12] [13] [14] [15]，本实验所需的各种 GNU 工具链软件信息如图4-2。

其中，这些仓库比较多一个一个下载比较浪费时间，而且加起来也很大，所以后来发现在执行的时候可以递归下载 riscv-gnu-toolchain [16]，可以一次将全部所需的工具链下载完成，最后在使用 Autotool 进行 configure，并根据自身机器进行配置，详细内容可见附录 B。

```

tony@YBX_gentoo ~/Documents/Risc-v $ ./info.sh
=====
      _ _ _ _ _
     |   |   |   |
     |   |   |   |
     |   |   |   |
gcc (Gentoo 9.3.0-r2 p4) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

=====

      _ _ _ _ _
     |   |   |   |
     |   |   |   |
     |   |   |   |
GNU Make 4.3
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

=====

      _ _ _ _ _
     |   |   |   |
     |   |   |   |
     |   |   |   |
git version 2.29.2

=====

      _ _ _ _ _
     |   |   |   |
     |   |   |   |
     |   |   |   |
qemu-riscv64 version 5.2.0
Copyright (c) 2003-2020 Fabrice Bellard and the QEMU Project developers

=====

      _ _ _ _ _
     |   |   |   |
     |   |   |   |
     |   |   |   |
VIM - Vi IMproved 8.2 (2019 Dec 12, compiled Dec 29 2020 22:49:45)
Included patches: 1-814
Modified by Gentoo-8.2.0814-r100
Compiled by tony@localhost
    
```

图 4-2 实验所需软件信息

4.2 Qemu & KVM 搭建

因为使用的系统环境是 Gentoo Linux，所以根据 Gentoo 官方的 Wiki 手册中 Qemu 部分来按照步骤完成即可 [17]，主要修改系统中的 USE 来支持 RISC-V [18]，详细内容可见已经录制好的视频《第一次安装配置 Gentoo》[19] 中 Qemu & KVM 部分。

4.3 移植过程介绍

Linux 在 v5.0 以后，既然内核已经可以支持 RISC-V，这样只需要在使用 make menuconfig 配置内核的时候加上支持 RISC-V 的参数，并在 make 的时候加上选项 ARCH=riscv，并指定需要的交叉编译器参数即可，当然编译不是一次就能成功的，可能已经编译很长时间最后还是失败了，但是可以根据日志来进行修改配置文件，最后其他设置就具体情况具体分析即可。

4.4 制作 BootLoader——BBL(Berkeley Boot Loader)

这里是根据 riscv-pk [15] 中 README.md 的步骤完成实现的。

需要注意的是在编译的时候需要指定 risc-v linux Kernel 的最原始未压缩的内核文件——vmlinux，完成编译 BootLoader，具体实现脚本可见附录 B。

4.5 创建根文件系统

busybox 使用了 riscv-qemu 的 virtio-net 与 virtio-block。首先需要使用 qemu-img 生成一块磁盘，用于分区，并存放文件系统。

在 busybear 的说明文档中有提到可以使用 busybox 来制作文件系统，参考 busybox 的手册 [20] 可以完成其安装配置，并让其完美的运行在 Qemu 上。

对于文件系统格式化，采用通用的 ext4 格式，挂载到临时文件夹，并创建根目录所需的一切结构文件夹，结束后要将临时文件夹删除。

4.6 配置 SSH 服务

考虑目前的系统比较简单不完善，所以考虑使用 SSH 来进行访问与操作。

在给系统实现 SSH 服务时，这里使用 dropbear 来实现提供对简单的 risc-v 操作系统的 SSH 服务，其实我想实现 SSH 操作并且可以外部访问，根据 Dropbear 的官方文档 [21] 实现安装与运行操作。

4.7 最终效果

最终启动结果如图4- 3所示，可以支持 SSH 访问，如图4- 4所示，并且可以执行程序如图4- 5所示。

```
(none) login: root
Password: ntpd: setting time to 2021-03-27 08:20:25.303826 (offset +1616833210.014591s)

          _ _ _ _ _
         / / / / /
        / / / / /
       / / / / /
      / / / / /
     / / / / /
    / / / / /
   / / / / /
  / / / / /
 / / / / /
/ / / / /

root@(none):~# ls
root@(none):~# pwd
/root
root@(none):~# cd /
root@(none):/# ls[ 36.699674] random: fast init done

bin      lib      lost+found  sbin      usr
dev      lib64    proc        sys        var
etc      linuxrc  root        tmp

root@(none):/# cd bin/
root@(none):/bin# ls
arch      echo      ldd        netstat    setarch
ash        ed        link       nice       setpriv
base64     egrep     linux32    nuke       setserial
busybox    false     linux64    pidof      sh
cat        fatattr   ln         ping       sleep
chattr     fdflush   login      ping6      stat
chgrp      fgrep     ls         pipe_progress stty
chmod      fsync     lsattr     printenv   su
chown      getopt    lzop       ps         sync
conspy     grep      makemime   pwd        tar
cp         gunzip    mkdir      reformime  touch
cpio       gzip      mknod      resume     true
cttyhack   hostname  mktemp     rev        umount
date       hush      more       rm         uname
dd         ionice    mount      rmdir     usleep
df         iostat   mountpoint rpm        vi
dmesg      ipcalc   mpstat     run-parts watch
dnsdomainname kbd_mode mt         scriptreplay zcat
dumpkmap   kill     mv         sed

root@(none):/bin#
```

图 4- 3 运行成功

为了表明自己实验成功是 RISC-V 的，实现编译好一个 RISC-V 的程序，来验证该操作系统能否正常的使用，./truth.riscv 是实现编译好的一个演示程序，运行结果如图4- 5所示，可以正常运行成功。

```

-enab
root@none:~/riscv-gcc-gcc-5_2_0-release# ./contrib/download_prerequisite
s
Connecting to gcc.gnu.org (8.43.85.97:21)
mpfr-2.4.2.tar.bz2 3% |*| 41640 0:00:24
mpfr-2.4.2.tar.bz2 62% |*****| 653k 0:00:01
mpfr-2.4.2.tar.bz2 100% |*****| 1052k 0:00:00
ETA
Connecting to gcc.gnu.org (8.43.85.97:21)
gmp-4.3.2.tar.bz2 4% |*| 85416 0:00:21
gmp-4.3.2.tar.bz2 57% |*****| 1072k 0:00:01
gmp-4.3.2.tar.bz2 100% |*****| 1853k 0:00:00
ETA
Connecting to gcc.gnu.org (8.43.85.97:21)
mpc-0.8.1.tar.gz 43% |*****| 232k 0:00:01
mpc-0.8.1.tar.gz 100% |*****| 532k 0:00:00
ETA
Connecting to gcc.gnu.org (8.43.85.97:21)
isl-0.14.tar.bz2 2% |*| 41640 0:00:32
isl-0.14.tar.bz2 55% |*****| 759k 0:00:01
isl-0.14.tar.bz2 100% |*****| 1367k 0:00:00
ETA
root@none:~/riscv-gcc-gcc-5_2_0-release# ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:192.168.122.2  Bcast:192.168.122.255  Mask:255.255.255.
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:322878  errors:0  dropped:2  overruns:0  frame:0
          TX packets:110618  errors:0  dropped:0  overruns:0  carrier:0
          collisions:0  txqueuelen:1000
          RX bytes:486480054 (463.9 MiB)  TX bytes:8045748 (7.6 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0  errors:0  dropped:0  overruns:0  frame:0
          TX packets:0  errors:0  dropped:0  overruns:0  carrier:0
          collisions:0  txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@none:~/riscv-gcc-gcc-5_2_0-release#

tony@YBX_gentoo ~ $ ssh root@192.168.122.2
root@192.168.122.2's password:
root@none:~# ifconfig
-sh: ifconfig: not found
root@none:~# ls
gcc-4.7.4      gcc-5_2_0-release
gcc-4.7.4.tar.gz  riscv-gcc-gcc-5_2_0-release
root@none:~# /bin/
arch      fdflush      mkdir      rpm
ash       fgrep        mknod      run-parts
base64    fsync        mktemp     sed
busybox   getopt       more       scriptreplay
cat        grep         mount      setarch
chattr    gunzip       mountpoint setpriv
chgrp     gzip         mpstat     setserial
chmod     hostname     mt         sh
chown     hush         mv         sleep
conspicy  ionice       netstat    stat
cp         iostat      nice       stty
cpio       ipcalc      nuke       su
cttyhack  kbd_mode    pidof      sync
date       kill         ping       tar
dd         ldd         ping6      touch
df         link        pipe_progress true
dmesg     linux32     printenv   umount
dnsdomainname linux64     ps         uname
dumpkmap  ln          pwd         usleep
echo      login       reformime  vi
ed         ls          resume     watch
egrep     lsattr      rev        zcat
false     lzop        rm
fatattr   makemime   rmdir
root@none:~# /bin/pi
pldof     ping        ping6      pipe_progress
root@none:~# ping
ping ping6
root@none:~# ping www.baidu.com
PING www.baidu.com (104.193.88.123): 56 data bytes
64 bytes from 104.193.88.123: seq=0 ttl=50 time=284.034 ms
64 bytes from 104.193.88.123: seq=1 ttl=50 time=276.416 ms
64 bytes from 104.193.88.123: seq=2 ttl=50 time=265.479 ms
64 bytes from 104.193.88.123: seq=3 ttl=50 time=280.354 ms
^C

```

图 4-4 支持 SSH 网络连接

```

root@none:~# uname -a
Linux (none) 5.0.0 #2 SMP Sat Mar 27 16:07:19 CST 2021 riscv64 GNU/Linux
root@none:~# chmod +x truth.riscv
root@none:~# ./truth.riscv
YBX is the most handsome man in NCEPU!!!
root@none:~#

```

图 4-5 运行程序

第 5 章 与前人比较

虽然本实验以前有人在 Github 上已经实现了，并且发布成项目 [7]，但是该实验工程已经四年多没有更新，在根据 README.md 文档重新复现该实验的时候，发现其中很多步骤已经失效了，而且运行很多步骤也会报错，但这些问题并不算什么的，我依旧可以凭借着这几年下来所学的知识进行解决，并成功运行一个演示程序。

比前人改进的地方就是全部软件更新到现在主流版本，修改了一些步骤上遇到的错误，以及解决遇到的一系列问题。

5.1 收获

通过该课程设计，让我对 Qemu 有了更深的理解，再加上对操作系统底层有更深入的理解，更深体会到编译工具的熟练使用。

5.2 不足

虽然，本文已经可以运行一个程序，但是目前还有如下问题：(1) 图形界面，就是用户需要使用图形界面的，这样可以更加方便用户使用，但是目前还没有想清楚如何添加图形界面方法；(2) 需要做横向对比，和 MIPS，ARM 比较，看一下 RISC-V 的平台会提高多少性能；(3) 其实这个也不算标准的移植，成功运行了，但是不代表日后会不会出现函数库，以及依赖的问题；

第 6 章 总结

本文仅仅是利用本科所学知识来进行的一次应用，就是将 Linux 移植到 risc-v 平台上，涉及到内容比较基础，主要目的是通过本毕设将大学的计算机组成原理，操作系统，计算机体系结构来进行全方面的强化，并将理论与实践相结合，最后希望我的一点探索努力可以更好的完善我国在计算机底层领域的长足发展。

参考文献

- [1] ELEXCON 深圳国际电子展. 关于 RISC-V 发展现状与应用分析和介绍 [EB/OL].
<http://www.elecfans.com/d/1011473.html>, 2019-10-24.
- [2] 孙卫真, 刘雪松, 朱威浦, 向勇. 基于 RISC-V 的计算机系统综合实验设计 [J]. 计算机工程与设计, 2021, 42(04): 1159–1165.
- [3] PATTERSON D A. Computer Organization and Design RISC-V Edition: The Hardware Software Interface (1st Edition) [M]. [S.l.]: Morgan Kaufmann, 2017. 267–321.
- [4] 刘鹏. 计算机组成与设计: RISC-V [EB/OL].
<https://www.icourse163.org/course/HIT-1452997167>, 2020-09-16.
- [5] WIKIPEDIA. Virtualization [EB/OL].
<https://en.jinzhaowiki/wiki/Virtualization>, 2004-01-15.
- [6] INTEL. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 [EB/OL].
<https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>, 2021-04-02.
- [7] MICHAELJCLARK. busybear-linux [EB/OL].
<https://github.com/michaeljclark/busybear-linux>, 2021.
- [8] 杨秉学. 从按下电源开始的一场接力赛 [EB/OL].
https://mp.weixin.qq.com/s/GJczyEAya7v5Yg_rLW6vHg, 2020-05-20.
- [9] GENTOO.ORG. Gentoo AMD64 Handbook [EB/OL].
<https://wiki.gentoo.org/wiki/Handbook:AMD64>, 2021-3-13.
- [10] GNU. riscv-binutils-gdb [EB/OL].
<https://github.com/riscv/riscv-binutils-gdb>, 2021-02-06.
- [11] GNU. riscv-dejagnu [EB/OL].
<https://github.com/riscv/riscv-dejagnu>, 2018-01-12.
- [12] GNU. riscv-gcc [EB/OL].
<https://github.com/riscv/riscv-gcc>, 2021-04-06.
- [13] GNU. riscv-glibc [EB/OL].
<https://github.com/riscv/riscv-glibc>, 2021-05-20.
- [14] GNU. riscv-newlib [EB/OL].
<https://github.com/riscv/riscv-newlib>, 2020-04-05.
- [15] GNU. riscv-pk [EB/OL].
<https://github.com/riscv/riscv-pk>, 2020-12-16.
- [16] GNU. riscv-gnu-toolchain [EB/OL].
<https://github.com/riscv/riscv-gnu-toolchain>, 2021-03-26.

- [17] GENTOO.ORG. Gentoo Qemu[EB/OL].
<https://wiki.gentoo.org/wiki/QEMU>, 2021-01-26.
- [18] 杨秉学. KVM 安装与配置 [EB/OL].
<https://mp.weixin.qq.com/s/5cMsDUIn0yZJhSV-gh3aBw>, 2019-07-12.
- [19] 杨秉学. 第一次安装配置 Gentoo[EB/OL].
<https://www.bilibili.com/video/BV1ny4y1i7G6>, 2020-12-28.
- [20] BUSYBOX. busybox[EB/OL].
<https://busybox.net/>, 2021-03-03.
- [21] DROPBEAR. Dropbear SSH[EB/OL].
<https://matt.ucc.asn.au/dropbear/dropbear.html>, 2020-10-29.

附录

附录 A

关于 Gentoo Linux 的部分配置请见:<https://www.bilibili.com/video/BV1ny4y1i7G6> 的视频演示。

关于本毕设的实验请见: <https://www.bilibili.com/video/BV1QU4y1H7AE>。

附录 B

修改 USE Flags 并安装

```
# vim /etc/portage/make.conf
QEMU_SOFTMMU_TARGETS="riscv32 risc64"
QEMU_USER_TARGETS="x86_64"

# vim /etc/portage/package.use
app-emulation/qemu qemu_softmmu_targets_arm qemu_softmmu_targets_x86_64
                qemu_softmmu_targets_sparc
app-emulation/qemu qemu_user_targets_x86_64

% 进行安装
# emerge --ask app-emulation/qemu -y
```

安装 GNU 工具链

```
mkdir YBX-bishe
cd YBX-bishe/
# 拉取 gnu-toolchain
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

# 编译生成 RISC-V newlib & Linux toolchains
cd riscv-gnu-toolchain
./configure --prefix=/opt/riscv --enable-multilib
make newlib -j5
make linux -j5
export PATH=$PATH:/opt/riscv/bin
export RISCVC=/opt/riscv
$
```

创建根文件系统

```
cd ..
git clone https://github.com/michaeljclark/busybear-linux.git
cd busybear-linux
make -j5
```

但是这没完，因为 busybear 会自动帮你下载好 busybox 但是需要自己进行解压和编译

```
CROSS_COMPILE=riscv{{bits}}-unknown-linux-gnu make menuconfig
CROSS_COMPILE=riscv{{bits}}-unknown-linux-gnu make
```

下面咱们就要制做最小文件系统

```
qemu-img create rootfs.img 1g
mkfs.ext4 rootfs.img
mkdir rootfs
sudo mount -o loop rootfs.img rootfs
cd rootfs
sudo cp -r ../busyboxsource/_install/* .
sudo mkdir proc sys dev etc etc/init.d
cd etc/init.d/
sudo touch rcS
sudo vi rcS
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
/sbin/mdev -s

sudo mod +x rcS
sudo umount rootfs
```

构建 Linux 内核

```
git clone https://github.com/torvalds/linux
cd linux
git checkout v5.4
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu defconfig
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu
```

制作 BootLoader——BBL(Berkeley Boot Loader)

```
cd..
git clone https://github.com/riscv/riscv-pk.git
cd riscv-pk
mkdir build
cd build
../configure \
> --enable-logo \
```

```
> --host=riscv64-unknown-elf \
> --with-payload=../../riscv-linux/vmlinux

make -j8
```

编写各种所需脚本

```
cd ..
mkdir Running
cd Running
```

创建 KVM 启动脚本

```
vim open_kvm
#!/bin/sh
sudo /etc/init.d/libvirtd start
sudo virsh net-start default #开启网络服务
```

创建 KVM 关闭脚本

```
sudo virsh net-destroy default
sudo /etc/init.d/libvirtd stop
```

创建网络启动脚本

```
#!/bin/sh
brctl addif virbr0 $1
ifconfig $1 up
```

创建网络关闭脚本

```
#!/bin/sh
ifconfig $1 down
brctl delif virbr0 $1
```

创建程序运行脚本

```
#!/bin/sh
# QEMU 5.2 以后 . 模拟器内部集成了 OpenSBI
sudo qemu-system-riscv64 \
    -nographic -machine virt \
    -m 1024M \
    -kernel bbl \
    -kernel ~/Documents/Risc-v/busybear-linux/build/linux-5.0/arch
        /riscv/boot/Image \
    -drive file=busybear.bin,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0 \
    -device virtio-net-device,netdev=net0 \
    -netdev type=tap,script=./ifup.sh,downscript=./ifdown.sh,id=net0 \
    -append "root=/dev/vda ro console=ttyS0"
```

运行

```
./Running.sh
```


致 谢

首先，我非常感谢我的家长，从小到大对我的殷勤付出，最终我才能有今天的收获，是他们的坚持不懈才让本因高考没有考上重点大学的我尽可能没有堕落下去；其次要感谢大学这群功利现实冷酷有没有人情的学生们，他们就是我的好教员，他们用他们冷漠自私让我明白一个道理“落后就要挨打，贫穷就要挨饿，失语就要挨骂”，只有站起来而不是跪下去才会有美好的未来；然后，要感谢毛泽东，邓小平等那些思想领袖，从他们哪里我学到了很多，大学之前没接触过计算机让我刚来的时候挂了不少课也被与预警过，但是面对这样的内外交迫的处境，“速战论”激进思想与“亡国论”的消极思想是不可取的，应该要打持久战，这是一场没有硝烟的战争，大致可划分三个阶段——“战略防御、战略相持、战略反攻”，要团结一切力量，积小胜为大胜，面对帝国主义和资产阶级那种腐朽、堕落和淫乱不堪的作风态度要坚定，最后我在大二结束的暑假就自学完本科全部的内容，并大三顺利地进入战略反攻阶段时，摧枯拉朽般把大一大二欠下的全部课程以及大三大四的课程一并补齐。最后还是要走一条具有自身特色的发展路线，要将理论与实践相结合，鞋子合不合脚自己穿了才知道；最后我要感谢琚贇老师，首先他并不像其他老师那样因为我大一大二基础差而区别对待，其次也没有“好心”地劝过“你学习不要的原因就是因为不努力学习，要抓紧学习”仅仅喊口号并没有提出解决方案，更没有说类似“我当了这么多年老师，你不是我见过聪明的学生”的话，而是采用基层民主制度，让我充分发挥自主的能动性，而不像奴才一样必须被用鞭子催着赶着让去做一件事。