追求纯粹的 COMPUTER SCIENCE
纯 C 论坛
2004.11（总第 2 期）
http://purec.binghua.com

【 卷 首 语 】

我心目中的程序员

原中国安天实验室 赵志刚[1]

记得前段时间紫丁香程序设计板搞了一次“你是夜猫子程序员么？”的投票调查，在建议中 pineapple 说了一句话让我想了很久：“怎样才算程序员呢？”。没错，这个问题很简单也很模糊，而且从不同角度回答会有不同的答案，每个人心目中都会有个答案。对我来说，成为程序员是我的理想。

程序设计是门科学，更是门艺术。如果局限于完成程序设计任务，只能说你是个编码人员。如果空有宏篇大论，自己却不能用程序设计来实现它，你可能会成为计算机科学家，但绝不是程序员！

我心目中的程序员应该是善于用程序设计的科学方法创造性的解决问题，并能从程序设计的实践中升华出新的程序设计理论和方法的人。成为程序员需要有扎实的基础，广阔的知识面，系统地学习，更需要坚持不懈的程序设计实践。

有人说程序员是碗青春饭。作为一个三十多岁的老编程工作者，我不同意这种观点，正所谓学无止境，活到老学到老。还记得最早开发时用的 APPLE II，然后是 PC/XT，286，386，486……，到现在超线程 P4。从 BASIC，FORTRAN，PASCAL，ADA，PL/I……，

[1] 赵志刚，原中国安天实验室创始人之一，高级工程师。电子邮件：hit-007@hit.edu.cn

到现在 C/C++，Java。经历了很多，也失去了很多。还记得为了解决一个技术问题，连续七天七夜不眠不休。过程中有多少失败的痛苦已经记不清了，但是换来的收获是值得这些付出的。现在新技术层出不穷，再加上我们国家在核心技术上的落后，确实让人觉得有些茫然无所适从。但是剥去这些新东西的华丽外衣，其实无非还是那些再基础不过的东西。没必要整天感叹自己的无知，盲目追逐新技术、新知识，应该从基础开始，循序渐进，罗马不是一天建成的。核心技术落后，是因为钻研核心技术的人太少了。

很多人妄自菲薄，更有人嘲笑那些搞核心技术的人在浪费时间，重复别人的工作。殊不知这些看似简单的重复、努力，正是创造的来源，创造不是无本之木，不会凭空出现，实践出真知。中国需要更多的踏踏实实钻研技术的人，尤其是核心技术。

我会继续付出我的努力，实现成为真正的程序员的理想。

仅以此文与大家共勉。

目　录

【　卷首语　】		
➢ 我心目中的程序员	赵志刚	1-2
【 计算机体系结构 】		
➢ 听大牛们谈未来的体系结构研究方向（一）	王凯峰	4-6
【　操作系统　】		
➢ Linux 核心（The Linux Kernel）（英汉对译）（一）	毕昕（等）	7-17
➢ Linux 核心（The Linux Kernel）（英汉对译）（二）	毕昕（等）	18-26
➢ Linux 核心（The Linux Kernel）（英汉对译）（三）	毕昕（等）	27-38
➢ 操作系统概念（第六版）（译）（一）	吕建鹏	39-56
➢ 保护模式下 8259A 芯片编程及中断处理探究（上）（Version 0.02）	谢煜波	57-64
➢ 保护模式下 8259A 芯片编程及中断处理探究（下）（Version 0.02）	谢煜波	65-74
【　C 与 C++　】		
➢ 浅析 C 语言函数传递机制及对变参函数的处理	谢煜波	75-93
【 算法与数据结构 】		
➢ 第 29 届 ACM 国际大学生程序设计竞赛亚洲区北京赛区预选赛试题题解（一）	哈工大 ACM/ICPC 组织	94
■ A 题 Finding Nemo 解题报告	宋鑫莹	95-96
■ B 题 Searching the Web 解题报告	刘禹	97
■ C 题 Argus 解题报告	肖颖	98
■ H 题 The Separator in Grid 解题报告	刘子阳	99-100
【　编辑部通讯　】		
➢ 关于将本刊改为双月刊的决定	本刊编辑部	101
➢ 投稿指南	本刊编辑部	102-103
➢ 读者俱乐部（含最新勘误表）	本刊编辑部	104
➢ 本期信息汇总	本刊编辑部	105
➢ SP1 版对原版的修订说明	本刊编辑部	106

听大牛们谈未来的体系结构研究方向（一）

哈尔滨工业大学 并行计算实验室 王凯峰[1]

这些大牛们都在各自的研究领域享有盛誉，听听他们谈谈对未来系统结构研究的看法将是十分有益和有趣的。

Saman Amarasinghe[2] —— MIT

当我们进入十亿晶体管时代，计算机设计者将会发现难以充分利用过剩的计算资源。通过不断地开发各式各样巨量的并行，高性能体系结构的峰值性能将会无限提高。然而，我们是否有足够的能力从应用级代码中开发出足够的并行呢？这将是决定我们能否达到最大性能极限的限制因素。在绝大多数情况下，这一问题并不是由应用程序自身决定的，而是由编程模型——高级语言的语义无法充分表述那些对取得高性能至关重要的某些特性。传统的程序设计模型大多数陷入两个极端：第一种情况是语言是为程序员设计的，忽略了先进的体系结构和高度优化的编译器的需要和能力。虽然这些语言得到了广泛地接受和使用，但是却给编译器设计者带来了麻烦（想一下 C/C++中的 Aliasing 问题、Java 中的内存模型）。另一个极端是，程序设计语言是为体系结构设计，忽略了程序员的需要。例如：为了获得高性能而采用的附加标识（HPF）或针对一个特定机器采用的某些体系结构专用的结构（Cray Fortran）。这些语言给程序员带来了额外的工作，与此同时还牺牲了程序的可编程性和可移植性。如果一种语言想要被广泛的使用，它必须保证能够在一系列机器上被高效实现的同时提高程序员的工作效率。我们倡议计算机体系结构设计者应该积极参与到高级语言的设计中来。通过认真的设计，有可能定义出既提高程序员效率又增加了编译器可用信息的语言初集。简单例子的包括：数据类型，结构化的控制流以及过程抽象。例如 Cg,Occam 和 Ada。实际上，我们已经将这种理念应用到 StreamIt（一种数字信号处理高级语言）的设计中去了。StreamIt 提供了几种同时有利于程序员和编译器的抽象：例如包括了一个不访问全局内存的自治过滤结构，一个能够通过过滤来重用输入项而无需维护复杂的环状缓冲的“peek”结构。我们认为一个编译器敏感的语言设计将是下个十年内保证高性能计算的关键。这需要计算机体系结构设计者们负起责任，怀着这种理念参与到程序设计语言的设计过程中来。否则，花费数以亿计的晶体管去设计一个复杂的硬件结构只能够获得很小的性能收益。

Krste Asanovic[3] —— MIT

并行体系结构是计算机体系结构的未来，现在的体系结构研究者转回到并行系统的时候了。串行结构上的效率提升始终有限。当前不少商业通用处理器生产上已经倾向于转向多处理器体系结构，或者 CMP 或者 SMT 或者二者兼而有之。而不是仅仅试图提高单一处理器

---

[1] 王凯峰，哈尔滨工业大学计算机科学与技术学院博士研究生，电子邮件：wkf@pact518.hit.edu.cn

[2] Saman Amarasinghe，麻省理工电子工程与计算机科学学院“计算机与人工智能实验室”副教授，曾是学术界著名的并行编译器项目组（SUIF compiler，斯坦福大学）研究组成员，并且是新型处理器结构 Raw Project 的主要领导者之一。目前领导 MIT 的 Commit Group。主要研究方向为：编译器优化技术，计算机系统结构，软件工程与并行计算。

[3] Krste Asanovic，麻省理工计算机与人工智能实验室副教授，研究方向为：计算机系统结构与 VLSI 设计。目前领导 SCALE 研究组，主要研究低功耗高性能体系结构。曾是著名的 IRAM 研究项目成员。

的 IPC。在嵌入式领域，专用的并行结构在当前的技术条件下能提升性能和功耗效率达到 10 到 1000 倍。然而，使我们苦恼的是，我们对并行编程知之甚少。目前的计算机科学家已经不是那些应用大型计算机的计算科学家了。并行应用程序的开发是一个缓慢而且费心的过程。只有那些有巨大的计算需求或者有严格的预算和功耗限制的人才会去开发。虽然目前很多关于并行编译器和软件开发工具的研究会对设计可用的并行系统有益，但我认为，除非我们为并行软件设计出更好的并行系统，否则，一切努力都将白费。

当前的并行系统是由单一处理器结构发展而来。后来发现，为支持并行代码而受到了内存一直模型和同步模型的困扰（还有更糟的就是然程序员自己是用消息传递来显式地指定数据的移动和同步）。这种底层硬件实在是粗糙，难以用来构建并行编程抽象。硬件应该更多地支持并行编程。

一些早期的研究为新的并行体系结构作了尝试。原始的数据流结构由于只关注并行而忽略局部性和可预测性而失败，但它确实提供了一个真实的并行结构的例子。后来人们又在数据流的思想上进行了局部性方面的研究，但这方面的工作进行的还远远不够。像数据流，缓存及前瞻技术的各种组合都值得深入研究。程序语言的设计者，编译器的设计者都应该加入到开发高效模块化的并行抽象上来。这些新的语言特性可能需要某些新的基础体系结构的支持。例如，在早期的机器上加入 jump and link 指令以支持过程的递归调用。同样，过去在串行计算机结构和变成抽象上取得的进步（大多数是为了六七十年代的超级计算机设计）已经渗入到各个应用领域，包括嵌入式领域和手持设备。我认为在并行计算机结构和编程抽象上取得的进步将在未来十年内渗入到各个领域。当前的技术早已使单芯片并行系统封装成为可能，而我们却缺乏在上面进行高效编程的能力，未来的纳米技术将使这一差距继续拉大。并行体系结构将是我们下一步要前进的方向。

Doug Burger[1] —— University of Texas at Austin

如果计算机性能和容量能够以过去的速度继续增长，下面三个研究方向将十分重要。第一，对于功耗敏感的体系结构研究仍应继续；第二，设计能够高效开发显示并行的结构十分重要；第三，能够开发隐式并行的大指令窗口的体系结构也十分必要。对于功耗敏感的体系结构：静态和动态功耗方面的考虑已成为处理器设计过程中最大的限制。虽然，对于“动态可调整结构”的研究进行了不少的工作，但我们发现他取得的性能收益正逐渐递减。而且太复杂了，难以应用到系统的大部分部件中去。设计先进体系结构和嵌入式应用时要时刻考虑功耗问题—因为它是以指数级形式上升。另外，功耗问题应该和可靠性问题一同考虑，二者密不可分。对于显示并行：由于流水线深度和时钟速率上的局限，新结构性能得提升将主要来源于开发更多的并行。一种方法是显示并行，它能够用于提高那些易提取并行性的代码的性能。目前这方面的挑战有：1）确定片上该集成何种机制能够提高这部分代码的可扩展性；2）确定 CMP 中 PE 的最佳粒度；3）要扩大那些能够有效地运行在并行结构上的代码类型；4）找到有效的方法来消除引脚接口的瓶颈—有研究表明，目前它已经成为影响性能的一大因素。这是因为片上可集成的晶体管数目增长的速度要远远大于引脚数增长的速度。隐式并行：对于开发单线索代码中的隐式并行方面的研究也十分重要。目前的体系结构可开发的隐式并行与代码中存在的隐式并行相差有一到二个数量级。这方面研究之所以重要是因为：第一，绝大多数代码无法显示并行化；第二，体系结构时钟速率方面的限制也强迫我们必须转向开发更多的并行。这些研究领域不应该被分割开来，理想的情况下，应该研究出一

[1] Doug Burger，德克萨斯大学奥斯汀分校计算机科学系副教授，电子与计算机工程系副教授，学术界著名的超标量模拟器 Simplescalar 的作者之一，CMP 结构思想的早期提出者。目前主要研究项目名为：TRIPS，一种创意十分超前的计算机体系结构，该项目的研究内容涉及计算机系统结构，操作系统，编译器技术以及应用程序设计。是目前风头最劲的年轻一代计算机体系结构研究领域执牛耳者。

种从根本上，简洁的底层机制一次性地解决上述问题，而不是将各种研究机制复杂的组合在一起。

Scott Mahlke[1] —— University of Michigan

进入 21 世纪之后，计算机系统结构的研究重心将从桌面系统转向嵌入式系统。我认为下列 4 个挑战将成为计算机系统结构研究的重点。

第一个挑战是移动环境下的高性能计算。普适计算成为现实还要走很长的一段路。目前我们的核心计算技术仍然无法满足下一代设备的需要。像连续语音识别和实时图像处理等新应用都对计算性能，成本和电池寿命有较高要求。研究出新一代具有超级计算机性能的便携式设备势在必行。

第二个挑战是可编程计算平台的定制。在嵌入式领域，通用性变得不再那么重要，不需要某个硬件执行所有的计算任务。虽然通过专用电路设计，ASICs 已经性能，成本以及功耗效率上取得了几个数量级的提高。但我们真正需要的并不是 ASICs，它是一种固化了的方法，同软件系统相比没有一点灵活性和可编程性。研究出新的硬件定制技术使得 ASIC 级设计能够应用于可编程平台将十分重要。

第三个挑战是可靠性。随着规模，反危机复杂度的增加，现在的硬件和软件都变得越来越不可靠。尤其是软件，里面存在着大量的 bug，安全漏洞和错误假设。显而易见，软件的正确性验证研究十分必要。达到完全可靠是不现实。但是最重要的是我们要能够使用不可靠的部件为用户搭建出一定程度上可靠的计算机系统。

最后一个挑战是编译技术。在嵌入式领域，为了保证性能，手工编写关键部分的代码司空见惯。目前的编译技术对这些常见的各种不规则的体系结构的支持远远不够。另外，编译对那些在嵌入式领域中广泛出现的并行计算环境的支持也不够。下一代的编译器必须能够高效的处理这类任务级并行的应用，针对具有不规则内存结构和互连的体系结构产生高质量的代码。当然，我们十分清楚，要让编译器对任何硬件平台都产生高质量的代码是不可能的。但对未来的嵌入式计算平台来讲，编译技术将是首要的研究任务。

【未完待续 • 责任编辑：kylix@hitbbs】

[1] Scott Mahlke，密歇根大学电子工程与计算机科学系高级计算机体系结构，软件系统实验室副教授，主要研究方向为：面向应用的处理器设计，编译器技术，计算机体系结构与高层次综合。

Linux 核心（The Linux Kernel）

原著：David A Rusling[1]

翻译[2]：毕昕[3]、胡宁宁[4]、仲盛[5]、赵振平[6]、周笑波[7]、李群[8]、陈怀临[9]

[编者按]

这几位现在已经从南京大学毕业的学生为我们留下了一份宝贵的讲解 Linux 内核的资料，在此，我们不得不对他们当年的辛苦的工作表示由衷的感谢。由于当时的翻译是按照原作者 0.8-2 版进行的，在本次收录的时候编者按照原作者的 0.8-3 版进行了校对，并对其中的一些地方进行了增改及补译。

哈尔滨工业大学计算机科学与技术学院 IBM 技术中心的吴晋老师对全部译稿进行了审校，在此特表感谢！

序

我们来自南京大学计算机系分布式计算实验室。石头城下，南北园里，我们一起度过了一些青春岁月。

历尽半年多，《Linux 核心》终于和大家见面了。作为译者，心中非常高兴。基于 Linux 核心 2.0.33，本书介绍了 Linux 核心是如何工作的。它不是一本关于核心的手册，而是描述了 Linux 核心中使用的原理，机制和 Linux 为什么使用这些原理和机制。希望本书能给读者带来些益处。

编译本书的过程中，我们没有局限于原作者的内容，加入了一些译者自己的理解。由于我们的专业和英语水平有限，疏漏之处在所难免。敬请读者谅解并望指出。请给我们发 Email 或在我们的 BBS 上留下您的意见。谢谢。

谢谢。

[1] 电子邮件：david.rusling@arm.com
[2] 由于无法取得所有作者现在的真实情况，所有作者介绍均来自作者翻译时的前注。
[3] 毕业于美国 Purdue 大学， 获 MS 学位。现供职于美国 GTE 公司。电子邮件：bixin@yahoo.com
[4] 现为美国 Carnegie Mellon 大学计算机系 Ph.D Student。电子邮件：bixin@yahoo.com
[5] 现为美国 Yale 大学 Ph.D。电子邮件：sheng.zhong@yale.edu
[6] 现任职于 Lode Soft 公 司，南京。电子邮件：ping@lodesoft.com
[7] 现为美国 Wayne State Univ.Assistant Professor。电子邮件：zbo@cs.wayne.edu
[8] 现为美国 DartMouth College Ph.D Candidate。电子邮件：qun.li@dartmouth.edu
[9] 美国硅谷软件工程师。电子邮件：niuniu_888@hotmail.com

Copy Right

This book is for Linux enthusiasts who want to know how the Linux kernel works. It is not an internals manual. Rather it describes the principles and mechanisms that Linux uses; how and why the Linux kernel works the way that it does.

Linux is a moving target; this book is based upon the current, stable, 2.0.33 sources as those are what most individuals and companies are now using.

This book is freely distributable, you may copy and redistribute it under certain conditions. Please refer to the copyright and distribution statement.

Version 0.8-3
David A Rusling
david.rusling@arm.com

版权声明[1]

这本书是为那些想知道 Linux 内核是怎样工作的 Linux 狂热者写的。它不是一份内部手册，而是描述了 Linux 所使用的一些原理及机制：Linux 内核是怎样工作的，以及为什么会这样工作的？

Linux 在不断的发展着，这本书基于当前的，稳定的，2.0.33 版源码，这个版本被现在的许多公司及个人使用。

这本书可以自由发布，你可以在限定条件下复制并重新发布它。请阅读版权声明及发行说明中的相应条款。

版本：0.8-3
David A Rusling
David.rusling@arm.com

[1] 此节原译稿没有，为编者补译

Preface

Linux is a phenomenon of the Internet. Born out of the hobby project of a student it has grown to become more popular than any other freely available operating system. Too many Linux is an enigma. How can something that is free be worthwhile? In a world dominated by a handful of large software corporations, how can something that has been written by a bunch of “hackers” (sic) hope to compete? How can software contributed to by many different people in many different countries around the world have a hope of being stable and effective? Yet stable and effective it is and compete it does. Many Universities and research establishments use it for their everyday computing needs. People are running it on their home PCs and I would wager that most companies are using it somewhere even if they do not always realize that they do. Linux is used to browse the web, host web sites, write theses, send electronic mail and, as always with computers, to play games. Linux is emphatically not a toy, it is a fully developed and professionally written operating system used by enthusiasts all over the world.

前言

Linux 是 Internet 的产物，从属于一个学生(Linus Torvalds)的个人爱好演变成为一个当今最流行的免费操作系统。对许多人而言，Linux 似乎是个迷。一个免费的东西怎么会有价值？在一个被一群软件巨头统治的(系统)软件王国里，一个由一些电脑 hackers 编写的操作系统如何能够参与竞争？一个由不同的国家许多不同的人编写的软件如何能够保持其稳定性和高效性？这里的答案是 Linux 具有非常好的可靠性、高效性和竞争能力。许多大学和研究机构每天都在用 Linux 来做计算。许多人已在其 PC 上安装了 Linux。而且我敢打赌，许多公司在很多地方都用到了 Linux，不过他们并不总是认识到了这一点。Linux 被广泛地用来浏览 web 站点，文件处理，发送 email，玩计算机游戏。Linux 绝不是一个计算机界的玩具，而是一个由全世界的爱好者开发的非常完善的，专业化的，并被全世界的狂热者使用的操作系统。

The roots of Linux can be traced back to the origins of Unix. In 1969, Ken Thompson of the Research Group at Bell Laboratories began experimenting on a multi-user, multi-tasking operating system using an otherwise idle PDP-7. He was soon joined by Dennis Richie and the two of them, along with other members of the Research Group produced the early versions of Unix. Richie was strongly influenced by an earlier project, MULTICS and the name Unix is itself a pun on the name MULTICS. Early versions were written in assembly code, but the third version was rewritten in a new programming language, C. C was designed and written by Richie expressly

Linux 的源头可以追溯到 Unix 家族。1969 年，贝尔实验室的研究人员 Ken Thompson 开始在一台空闲的 PDP-7 机器上实验其多用户，多任务的操作系统。不久 Dennis Richie 和其他两位同事加入了他的行列。他们与实验室中的其他同事一道开发出了最早期的 Unix 版本。Richie 在早期的项目 MULTICS 中发挥了很大的作用。Unix 其实是 MULTICS 的双关语。早期的 Unix 是用汇编写的。第 3 版时采用了 C 语言。C 语言是 Richie 设计并编写的，以用来作为编写操作系统设计的语言。用 C 改写过的 Unix 使得 Unix 可以被移植 PDP-11/45 和 DIGITAL 11/70 计算机上。后来发生的一切，正如他们所说，都成为了历史。Unix 移植到 DIGITAL

as a programming language for writing operating systems. This rewrite allowed Unix to move onto the more powerful PDP-11/45 and 11/70 computers then being produced by DIGITAL. The rest, as they say, is history. Unix moved out of the laboratory and into mainstream computing and soon most major computer manufacturers were producing their own versions.

11/70 是一个历史性的转折，使得 Unix 正式从实验室走向大型机计算环境。很快，绝大多数的计算机制造商都发布了其相应的 Unix 版本。

Linux was the solution to a simple need. The only software that Linus Torvalds, Linux's author and principle maintainer was able to afford was Minix. Minix is a simple, Unix like, operating system widely used as a teaching aid. Linus was less than impressed with its features, his solution was to write his own software. He took Unix as his model as that was an operating system that he was familiar with in his day to day student life. He started with an Intel 386 based PC and started to write. Progress was rapid and, excited by this, Linus offered his efforts to other students via the emerging world wide computer networks, then mainly used by the academic community. Others saw the software and started contributing. Much of this new software was itself the solution to a problem that one of the contributors had. Before long, Linux had become an operating system. It is important to note that Linux contains no Unix code, it is a rewrite based on published POSIX standards. Linux is built with and uses a lot of the GNU (GNU's Not Unix) software produced by the Free Software Foundation in Cambridge, Massachusetts.

Linux 诞生的原因极其简单。Linus Torvalds，Linux 的作者和主要管理者，当时穷的只能够付的起 Minix。Minix 是一个非常简单，Unix 风格的，被广泛用在教学上的操作系统。Linus 对 Minix 的功能不是很满意（译者注：不知 Andrew Tanenbaum 看见这句话会有何感想。“后生可畏？”。有兴趣的读者可以访问 Andrew 的主页。）决定自己动手编写一个软件。因为在学校里每天用的都是 Unix，所以他选择将 Unix 作为他的软件的模型。最开始的工作是在一台 Intel 386 的 PC 机上。他的进展很迅速。Linus 对他所做的事情充满了兴趣，并通过刚刚出现的，还局限在学术领域的计算机网络，将已有的代码共享给其他的学生。其他的人看见了 Linus 的软件并开始加入开发的行列。不同的人由于在使用 Linux 时碰到不同的问题，所以这个软件也就不断地被更新和完善。不久之后，Linux 就成为一个完整的操作系统了。值得注意的是 Linux 中没有任何 Unix 代码，而是根据 POSIX 标准重新编写的。Linux 中使用了许多在 Cambridge, Massachusetts 的 Free Software Foundation 提供的 GNU 软件。

Most people use Linux as a simple tool, often just installing one of the many good CD ROM-based distributions. A lot of Linux users use it to write applications or to run applications written by others. Many Linux users read the HOWTOs avidly and feel both the thrill of success when some part of the system has been correctly configured and the

多数人仅把 Linux 当成一个简单的工具来使用，通常只是安装众多优秀的 Linux 的 CD-ROM 发行版中的一种。也有许多用户在 Linux 上进行应用程序的开发或者运行其它用户写的程序。许多 Linux 用户积极的阅读 HOWTOs，常常会因为成功的配置了系统的某一部份而兴奋，或者承受由失败带来的挫折感。只有很少数的人敢于为 Linux 写设备

frustration of failure when it has not. A minority are bold enough to write device drivers and offer kernel patches to Linus Torvalds, the creator and maintainer of the Linux kernel. Linus accepts additions and modifications to the kernel sources from anyone, anywhere. This might sound like a recipe for anarchy but Linus exercises strict quality control and merges all new code into the kernel himself. At any one time though, there are only a handful of people contributing sources to the Linux kernel.

驱动程序和核心的 patches（译者注：这个词不太好翻译），他们是 Linux 内核的创造者及维护者。 Linus Torvalds 接受来自任何人、任何地方的关于核心的补充和修改。这似乎有点像无政府主义。不过 Linus 严把质量关，并由他自己将新的代码加入核心。当然，在任何时候，从事 Linux 核心开发的毕竟只是一少部份人。

The majority of Linux users do not look at how the operating system works, how it fits together. This is a shame because looking at Linux is a very good way to learn more about how an operating system functions. Not only is it well written, all the sources are freely available for you to look at. This is because although the authors retain the copyrights to their software, they allow the sources to be freely redistributable under the Free Software Foundation's GNU Public License. At first glance though, the sources can be confusing; you will see directories called kernel, mm and net but what do they contain and how does that code work? What is needed is a broader understanding of the overall structure and aims of Linux. This, in short, is the aim of this book: to promote a clear understanding of how Linux, the operating system, works. To provide a mind model that allows you to picture what is happening within the system as you copy a file from one place to another or read electronic mail. I well remember the excitement that I felt when I first realized just how an operating system actually worked. It is that excitement that I want to pass on to the readers of this book.

大多数 Linux 用户似乎并不关心这个操作系统是如何构造和运行的。这不是个明智的决定，因为学习 Linux 是更好理解一个操作系统功能的有力途径。Linux 不仅仅是设计的很好，更重要的是其源代码是公开的。这是因为虽然作者拥有这个软件的版权，但在 Free Software Foundation's GNU Public License 的基础上，源代码是可以免费获取的。对刚接触源码的人，起初，当看见一些叫做 kernel、mm 和 net 的子目录时，会觉得迷惑。它们含有些什么？这些代码是如何工作的？为了解决上述问题，我们需要的是对整个 Linux 的结构与目标有个总体的了解。这也正是本书的目的：提供一个关于 Linux 核心如何工作的清楚的画面。使你的脑海中可以建立起一种模型，当你从一个地方将文件复制到另一个地方的时候，或者当你阅读电子邮件的时候，这个模型能让你勾画出操作系统内部此时正在干什么的画面（译者注：这一点是为什么要鼓励应用程序开发者了解操作系统结构的原因。例如，不理解操作系统的调度，很难想象一个应用程序开发者可以编出一个高效的多线程并发程序）。我还对我第一次认识到操作系统是怎样工作时的激动情景记忆犹新。我想通过这本书把这种激动也带给我的读者。

My involvement with Linux started late in 1994 when I visited Jim Paradis who was working on a port of Linux to the Alpha AXP processor based systems. I had worked for

（编者注：下面这段原译稿中没有，由编者补译）

我最早同 Linux 打交道是在 1994 年下半年。那时我拜访了 Jim Paradis，他正在致力于将 Linux 系统移植到 Alpha AXP 下工作。在这之前，从 1984 年开始我在为 DEC 公司

Digital Equipment Co. Limited since 1984, mostly in networks and communications and in 1992 I started working for the newly formed Digital Semiconductor division. This division's goal was to enter fully into the merchant chip vendor market and sell chips, and in particular the Alpha AXP range of microprocessors but also Alpha AXP system boards outside of Digital. When I first heard about Linux I immediately saw an opportunity to have fun. Jim's enthusiasm was catching and I started to help on the port. As I worked on this, I began more and more to appreciate not only the operating system but also the community of engineers that produces it.

的网络与通信部门工作。1992 年，我开始为新组建的数字半导体部门工作。这个部门的目标是充分进入商用芯片市场卖芯片，特别是 Alpha AXP 系列的微处理器，也包括 Digital 以外的 Alpha AXP 系统版卡。当我第一次听说 Linux 的时候，我就立即发现这是一个很有趣的机会（编者注：作者认为这是一个可以从中得到乐趣的机会）。Jim 的狂热感染了我，我也开始帮助他做移植工作，随着越来越多的同它的接触，我也越来越欣赏这个操作系统，以及创造它的工程师们。

However, Alpha AXP is only one of the many hardware platforms that Linux runs on. Most Linux kernels are running on Intel processor based systems but a growing number of non-Intel Linux systems are becoming more commonly available. Amongst these are Alpha AXP, ARM, MIPS, Sparc and PowerPC. I could have written this book using any one of those platforms but my background and technical experiences with Linux are with Linux on the Alpha AXP and, to a lesser extent on the ARM. This is why this book sometimes uses non-Intel hardware as an example to illustrate some key point. It must be noted that around 95% of the Linux kernel sources are common to all of the hardware platforms that it runs on. Likewise, around 95% of this book is about the machine independent parts of the Linux kernel.

然而，Alpha AXP 只是可以运行 Linux 的众多硬件平台中的一种。大部份的 Linux 内核运行在基于 Intel 处理器的系统上，当然，非 Intel 的 Linux 系统也越来越常见，比如 Alpha AXP、ARM、MIPS、Sparc 以及 PowerPC。我可以基于上面的任何一个平台来写这本书，不过我的背景以及与 Linux 有关的技术经验更主要来源于 Alpha AXP，其次是 ARM。 这就是这本书时常用一些非 Intel 的硬件作为例子来解释一些重要观点的原因。然而 95%的 Linux 核心源代码是与具体的硬件平台无关的。换而言之，本书 95%的内容是讲述与硬件无关的 Linux 核心。

Reader Profile

This book does not make any assumptions about the knowledge or experience of the reader. I believe that interest in the subject matter will encourage a process of self education where necessary. That said, a degree of familiarity with computers, preferably the PC will help the reader derive real benefit from

关于读者

本书不要求读者必须具备任何知识和经验的前提。我相信对这个主题感兴趣的读者在需要的时候，会努力的去自学一些相关知识。具备一定的计算机知识，尤其是 PC 机的知识，和 C 语言功底将帮助读者更好地理解本书。

the material, as will some knowledge of the C programming language.

Organization of this Book

本书的组织

This book is not intended to be used as an internals manual for Linux. Instead it is an introduction to operating systems in general and to Linux in particular. The chapters each follow my rule of “working from the general to the particular”. They first give an overview of the kernel subsystem that they are describing before launching into its gory details.

本书并不是关于 Linux 内部的手册。它是关于通常意义上的操作系统，特别是对于 Linux 的介绍。每一章节按照从“普通到特殊”的规则来布局。首先在具体介绍细节之前，我们给出一个对核心子系统的综述。

I have deliberately not described the kernel's algorithms, its methods of doing things, in terms of routine_X() calls routine_Y() which increments the foo field of the bar data structure. You can read the code to find these things out. Whenever I need to understand a piece of code or describe it to someone else I often start with drawing its data structures on the white-board. So, I have described many of the relevant kernel data structures and their interrelationships in a fair amount of detail.

关于核心的具体算法被特意地略去，这些算法大多是完成这些操作的具体方法。例如：routine_x()调用了 routine_Y()，routine_Y()增加了 bar 这一结构体的 foo 域的值。你可以阅读源代码找到这些具体的操作方法。无论什么时候，只要我需要理解一段代码或者向另外一个人描述它，我会一开始就在白板上画出它的数据结构，因此本书也将重点放在核心数据结构和它们之间的关系上。

Each chapter is fairly independent, like the Linux kernel subsystem that they each describe. Sometimes, though, there are linkages; for example you cannot describe a process without understanding how virtual memory works.

本书每一章都相对独立，就像每一个 Linux 子系统完成相对独立的功能一样。当然章节之间也是相关联的。例如，如果不理解虚拟内存是如何工作的，你就不能很好地描述一个进程。

The Hardware Basics chapter gives a brief introduction to the modern PC. An operating system has to work closely with the hardware system that acts as its foundations. The operating system needs certain services that can only be provided by the hardware. In order to fully understand the Linux operating system, you need to understand the basics of the underlying hardware.

硬件基础章节给出了一个对当代 PC 的简要介绍。一个操作系统必须工作在一个硬件基础上。操作系统中的某些功能是与硬件相关的。它需要的某些服务只能由硬件提供。为了理解 Linux 操作系统，读者需要具备一定的底层硬件知识。

The Software Basics chapter introduces basic software principles and looks at assembly

软件基础章节介绍了基本的软件知识原理并论及了汇编和 C 语言。它们都是编写一

and C programming languages. It looks at the tools that are used to build an operating system like Linux and it gives an overview of the aims and functions of an operating system.

个操作系统不可缺少的工具。另外本章给出了对操作系统的功能和目标的一个综述。

The Memory Management chapter describes the way that Linux handles the physical and virtual memory in the system.

内存管理章节描述了 Linux 如何管理系统中的物理和虚拟内存。

The Processes chapter describes what a process is and how the Linux kernel creates, manages and deletes the processes in the system.

进程章节描述了什么是一个进程，Linux 核心如何创建、管理和删除系统中的进程。

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms. Signals and pipes are two of them but Linux also supports the System V IPC mechanisms named after the Unix release in which they first appeared. These inter process communications mechanisms are described in Chapter IPC chapter.

进程之间、进程与核心之间通过通讯来调整或说安排它们的行为。Linux 支持很多种进程间通讯——IPC（Inter Process Communication）机制。信号与管道是其中的两种。Linux 还支持 System V IPC 机制，这一机制在 Unix 中首次出现，并在 Unix 被发行之后命名。关于 IPC 机制的内容在 IPC 章节描述。

The Peripheral Component Interconnect (PCI) standard is now firmly established as the low cost, high performance data bus for PCs. The PCI chapter describes how the Linux kernel initializes and uses PCI buses and devices in the system.

外围组件互连（PCI）标准已经成为了 PC 机上一个低成本，高性能的 PC 数据总线标准。PCI 章节讲述了 Linux 核心如何初始化和使用 PCI 总线和系统中的 PCI 设备。

The Interrupts and Interrupt Handling chapter looks at how the Linux kernel handles interrupts. Whilst the kernel has generic mechanisms and interfaces for handling interrupts, some of the interrupt handling details are hardware and architecture specific.

中断和中断处理章节探讨了 Linux 核心如何处理中断。尽管操作系统核心里提供一些通用的机制和接口来处理中断，但一些关于中断处理的细节是与硬件体系结构有关的。

One of Linux's strengths is its support for the many available hardware devices for the modern PC. The Device Drivers chapter describes how the Linux kernel controls the physical devices in the system.

Linux 的优点之一是它支持许多现代 PC 机上使用的硬件设备。设备驱动程序章节描述了 Linux 核心如何控制系统中的物理设备。

The File system chapter describes how the Linux kernel maintains the files in the file systems that it supports. It describes the Virtual File System (VFS) and how the Linux kernel's real file systems are supported.

文件系统章节讲述了 Linux 核心如何维护其文件系统中的文件，描述了虚拟文件系统（VFS）和如何支持 Linux 核心中的真正文件系统。

Networking and Linux are terms that are almost synonymous. In a very real sense Linux is a product of the Internet or World Wide Web (WWW). Its developers and users use the web to exchange information ideas, code and Linux itself is often used to support the networking needs of organizations. Chapter networks chapter describes how Linux supports the network protocols known collectively as TCP/IP.

网络与 Linux 几乎是一对同义词。从某种真实的意义上讲，Linux 是 Internet 或 World Wide Web（WWW）的产物。Linux 的开发和使用者通过 web 来交换信息和代码。Linux 也通常被用来支持机构的网络需求。网络章节描述了 Linux 如何支持 TCP/IP 协议族。

The Kernel Mechanisms chapter looks at some of the general tasks and mechanisms that the Linux kernel needs to supply so that other parts of the kernel work effectively together.

核心机制章节探讨了 Linux 核心需要支持或说完成的一些通用任务和机制，以用来使得核心其他部分有效地工作在一起。

The Modules chapter describes how the Linux kernel can dynamically load functions, for example file systems, only when they are needed.

模块章节讲述了 Linux 如何只在需要的时候动态地装载功能模块，例如文件系统。

The Processors chapter gives a brief description of some of the processors that Linux has been ported to.

（下段原译稿没有，由编者补译）

处理器章节简要的描述了一些已经移植了 Linux 的处理器。

The Sources chapter describes where in the Linux kernel sources you should start looking for particular kernel functions.

源代码章节描述了对应于特定的核心功能的 Linux 核心源代码地址。读者可以依据本章的介绍来开始阅读源代码。

Conventions used in this Book

本书的约定

The following is a list of the typographical conventions used in this book.

下面是一些本书在排版上的约定：

serif font identifies commands or other text that is to be typed literally by the user.

serif font：用户必须输入的命令或文本。

type font refers to data structures or fields within data structures.

type font：数据结构和数据结构中的域。

Throughout the text there references to pieces of code within the Linux kernel source tree (for example the boxed margin note adjacent to this text). These are given in case you wish to look at the source code itself and all of the file references are relative to /usr/src/linux. Taking foo/bar.c as an example, the full filename would be /usr/src/linux/foo/bar.c If you are running Linux (and you should), then looking at the code is a worthwhile experience and you can use this book as an aid to understanding the code and as a guide to its many data structures.

在全书的正文中，引用了 Linux 内核源代码树中的部分代码（比如，出现在正文空白处的小框）（编者注：原书正文的空白部份有不少小框标明了此处所描述的内容出现在哪个源文件中）这是给你机会，希望你能自己阅读源码。本书中所引用的源程序都基于相对路径/usr/src/linux。如 foo/bar.c，则其实际的含路径的完整文件名是/usr/src/linux/foo/bar.c。如果你正在运行 Linux（你应当这样做），那么阅读源代码是非常有价值的，你可以把这本书当作你理解源代码及众多数据结构的助手和相导。

（编者注：以下一节原译稿中没有，由编者补译）

The Author

I was born in 1957, a few weeks before Sputnik was launched, in the north of England. I first met Unix at University, where a lecturer used it as an example when teaching the notions of kernels, scheduling and other operating systems goodies. I loved using the newly delivered PDP-11 for my final year project. After graduating (in 1982 with a First Class Honours degree in Computer Science) I worked for Prime Computers (Primos) and then after a couple of years for Digital (VMS, Ultrix). At Digital I worked on many things but for the last 5 years there, I worked for the semiconductor group on Alpha and StrongARM evaluation boards. In 1998 I moved to ARM where I have a small group of engineers writing low level firmware and porting operating systems. My children (Esther and Stephen) describe me as a geek.

作者介绍

1957 年，在苏联人造卫星发射升空的几周前，我出生在英格兰北部。我第一次接触 Unix 是在大学里，当时有位老师在讲解内核概念，调度概念及其它操作系统产品的时候，用它做了一个例子。在最后一年里，我用最新的 PDP-11 完成了项目的开发。毕业后（1982 年，计算机科学一级荣誉学位）（编者注：抱歉，对于作者所获得学位的准确称谓，实在没有能力准确把握），我开始为 Prim Computers（Primos）工作，两年后，加入 Digital（VMS，Ultrix）。在 Digital，我从事过多项工作，不过在 Digital 的最后 5 年，被调入开发 Alpha 及 StrongARM 测试板的半导体组。1998 年，我来到 ARM，在这里，我领导一小组工程师在编写底层的固件及移植操作系统。我的孩子（Esther 与 Stephen）常说我是一个很滑稽的人。

People often ask me about Linux at work and at home and I am only too happy to oblige. The more that I use Linux in both my professional and personal life the more that I become a Linux zealot. You may note that I use

有人经常在工作及生活中，问我一些关于 Linux 的问题，我常常乐极而忘言谢。在我的职业及生活中用 Linux 越多，我也越来越多的成为一个 Linux 的狂热者。你必须注意到，我用的是“狂热者”而不是“顽固者”。

the term `zealot' and not `bigot'; I define a Linux zealot to be an enthusiast that recognizes that there are other operating systems but prefer not to use them. As my wife, Gill, who uses Windows 95 once remarked ``I never realized that we would have his and her operating systems". For me, as an engineer, Linux suits my needs perfectly. It is a superb, flexible and adaptable engineering tool that I use at work and at home. Most freely available software easily builds on Linux and I can often simply download pre-built executable files or install them from a CD ROM. What else could I use to learn to program in C++, Perl or learn about Java for free?

我认为一个 Linux 狂热者一定是这样的一个人，他知道还有很多其它的操作系统，但却不愿意使用它们。作为一个 Windows 95 用户，我的妻子，Gill，曾经提及：“我从来没有想到我会用到他或她的操作系统。”（编者注：包歉，对于 Gill 这句话的意思，实在没有能力准确把握）对于我来说，作为一个工程师，Linux 最能满足我的需要。它是一个我在工作及生活中使用的极妙的、灵活的、可修改的技术工具。非常多的免费的好用的软件可以很容易的在 Linux 上安装。我常常下载一些未安装的可执行文件或者从 CD-ROM 中安装它们。还有什么可以被用来免费学习 C++、Perl 或者 Java 呢？

Acknowledgements

I must thank the many people who have been kind enough to take the time to e-mail me with comments about this book. I have attempted to incorporate those comments in each new version that I have produced and I am more than happy to receive comments, however please note my new e-mail address.

A number of lecturers have written to me asking if they can use some or parts of this book in order to teach computing. My answer is an emphatic yes; this is one use of the book that I particularly wanted. Who knows, there may be another Linus Torvalds sat in the class.

Special thanks must go to John Rigby and Michael Bauer who gave me full, detailed review notes of the whole book. Not an easy task. Alan Cox and Stephen Tweedie have patiently answered my questions - thanks. I used Larry Ewing's penguins to brighten up the chapters a bit. Finally, thank you to Greg Hankins for accepting this book into the Linux Documentation Project and onto their web site.

（编者注：这节原译稿也没有，由编者补译）

致谢

我必须感谢许多人，他们不惜花费时间，把他们对这本书的意见和建议，通过电子邮件的方式告诉我。我正尝试将所有的评论合并到一个新版本中。我也非常高兴能收到更多的评论，不过，请注意我的新邮件地址。

很多教师写信向我询问他们是否可以将这本书中的一些章节用于教学目的。我的回答是非常肯定的。这是我特别期待的一种用途。谁能知道，这些课堂上会不会坐着另一个 Linus Torvalds 呢？

要特别感谢 John Rigby 与 Michael Bauer，他们全面而细致的审阅了此书——这不是一件轻松的工作。非常感谢 Alan Cox 与 Stephen Tweedie 对我的问题的耐心解答！Larry Ewing 的企鹅让这本书增色不少。最后，感谢 Greg 将该书列入 LDP（Linux 文档计划）并将该书发布在他们的网站上。

【未完待续 • 责任编辑：iamxiaohan@hitbbs】

Linux 核心（The Linux Kernel）——第一章

原著: David A Rusling

翻译: 毕昕、胡宁宁、仲盛、赵振平、周笑波、李群、陈怀临

Chapter 1 Hardware Basics

An operating system has to work closely with the hardware system that acts as its foundations. The operating system needs certain services that can only be provided by the hardware. In order to fully understand the Linux operating system, you need to understand the basics of the underlying hardware. This chapter gives a brief introduction to that hardware: the modern PC.

When the “Popular Electronics” magazine for January 1975 was printed with an illustration of the Altair 8080 on its front cover, a revolution started. The Altair 8080, named after the destination of an early Star Trek episode, could be assembled by home electronics enthusiasts for a mere $397. With its Intel 8080 processor and 256 bytes of memory but no screen or keyboard it was puny by today's standards. Its inventor, Ed Roberts, coined the term “personal computer” to describe his new invention, but the term PC is now used to refer to almost any computer that you can pick up without needing help. By this definition, even some of the very powerful Alpha AXP systems are PCs.

Enthusiastic hackers saw the Altair's potential and started to write software and build hardware for it. To these early pioneers it represented freedom; the freedom from huge batch processing mainframe systems run and guarded by an elite priesthood. Overnight fortunes were made by college dropouts fascinated by this new phenomenon, a computer that you could have at home on your

第一章 硬件基础

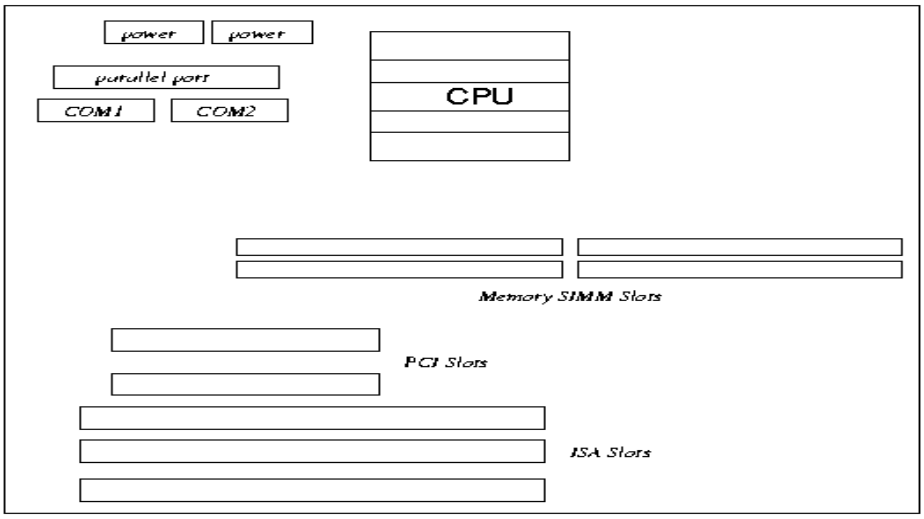
一个操作系统必需紧密地和其支撑——硬件系统结合在一起。操作系统需要一些只能由硬件提供的一些服务。为了很好的理解 Linux 操作系统，读者需要明白一些低层的硬件知识。本章对当代 PC 系统的一些硬件作一个介绍。

1975 年的 1 月，当“Popular Electronics”杂志在其封面上给出 Altair 8080 的照片后，一场革命就开始了。Altair 8080（这个名字源于一部早期电影--星际旅行--的目的地的），这个可以被家用电器爱好者组装出来的机器，当时的价格是 397 美金。对于今天而言，其 Intel 8080 的处理器，256 字节的内存，没有屏幕和键盘的配置是微不足道的。它的发明者 Ed Roberts 缔造了一个新词，“个人计算机”来描述他的新发明，不过现在PC已经被用在所有你可以一个人搬走的计算机上了。按照这个定义来讲，即使一些性能更高的 Alpha AXP 系统也称之为 PC。

充满激情的计算机黑客们看见了 Altair 的潜力，并开始为它写软件和在其上为它配置硬件。对这些先驱者来说，这是一种摆脱呆版的、基于批处理的大型机的自由。一些退学的在校生通过这些个人计算机（一个你可以放在家中厨房桌子上的计算机）一夜之间获得了巨大的财富。市场上出现了大量的硬件设备。软件 hackers 们非常高兴地为这些新机器编写软件。有趣的是，IBM 公司坚定

kitchen table. A lot of hardware appeared, all different to some degree and software hackers were happy to write software for these new machines. Paradoxically it was IBM who firmly cast the mould of the modern PC by announcing the IBM PC in 1981 and shipping it to customers early in 1982. With its Intel 8088 processor, 64K of memory (expandable to 256K), two floppy disks and an 80 character by 25 lines Color Graphics Adapter (CGA) it was not very powerful by today's standards but it sold well. It was followed, in 1983, by the IBM PC-XT which had the luxury of a 10Mbyte hard drive. It was not long before IBM PC clones were being produced by a host of companies such as Compaq and the architecture of the PC became a de-facto standard. This de-facto standard helped a multitude of hardware companies to compete together in a growing market which, happily for consumers, kept prices low. Many of the system architectural features of these early PCs have carried over into the modern PC. For example, even the most powerful Intel Pentium Pro based system starts running in the Intel 8086's addressing mode. When Linus Torvalds started writing what was to become Linux, he picked the most plentiful and reasonably priced hardware, an Intel 80386 PC.

的制定了现代 PC 的模式，并在 1981 年制造了当代 PC——IBM PC，并在 1982 年交给用户使用。当时 IBM PC 的配置是 Intel 8088 处理器，64K 内存(可扩展到 256K)，两个软磁盘和一个 80 字符，25 行的 CGA 显示适配器。按照今天的标准来看，它并不具有很好的性能，但它卖得很好。1983 年，IBM 推出了 IBM PC-XT，含有一个昂贵的 10M 硬盘。不久之后，许多公司，如 COMPAQ，推出了 IBM PC 的兼容机。PC 的体系结构变成了一个事实上的工业标准。这个事实上的工业标准使得大量的硬件公司在一个增长的市场中进行竞争。从而使的 PC 价格变得越来越便宜，用户对此非常高兴。早期 PC 的许多结构特征被现代 PC 所继承。例如，即使先进的 Intel Pentium Pro 系统在开始时也运行在 Intel 8086 的地址模式下。当 Linus Torvalds 开始写 Linux 时，他选择了当时最多的，价格也较合理的 Intel 80386 PC。



（图 1.1 一个典型的 PC 主板）

Looking at a PC from the outside, the most obvious components are a system box, a keyboard, a mouse and a video monitor. On the front of the system box are some buttons, a little display showing some numbers and a floppy drive. Most systems these days have a CD ROM and if you feel that you have to protect your data, then there will also be a tape drive for backups. These devices are collectively known as the peripherals.

从外面观察一个 PC，最明显的部件是一个系统主机，键盘，鼠标器和一个显示器。系统主机前面有几个按钮，一个小的可以显示几个数字的显示屏和一个软驱。今天的大多数系统还有 CD ROM。如果你想要保护数据，还可以有一个磁带驱动器用来做数据备份。这些设备通称为外设。

Although the CPU is in overall control of the system, it is not the only intelligent device. All of the peripheral controllers, for example the IDE controller, have some level of intelligence. Inside the PC (Figure 1.1) you will see a motherboard containing the CPU or microprocessor, the memory and a number of slots for the ISA or PCI peripheral controllers. Some of the controllers, for example the IDE disk controller may be built directly onto the system board.

虽然 CPU 是系统的主要控制部件，但它不是系统中唯一具有智能的设备。所有的外设控制器，例如，IDE 控制器，都有一定的智能成份。在一个 PC 的内部（如图 1.1 所示），你可以看见一个含有 CPU 或微处理器的主板、内存以及一些 ISA 或 PCI 外设控制器插槽。有些控制器，如 IDE 磁盘控制器，有可能被直接作在系统板上。

1.1 The CPU

1.1 CPU

The CPU, or rather microprocessor, is the heart of any computer system. The microprocessor calculates, performs logical operations and manages data flows by reading instructions from memory and then executing them. In the early days of computing the functional components of the microprocessor were separate (and physically large) units. This is when the term Central Processing Unit was coined. The modern microprocessor combines these components onto an integrated circuit etched onto a very small piece of silicon. The terms CPU, microprocessor and processor are all used interchangeably in this book.

CPU，或微处理器，是任何一个计算机系统的心脏。CPU 通过读取并执行内存中的指令来进行计算，执行逻辑运算和管理数据流。在早期，微处理器的功能部件是分离元件（在外形上较大）。这就是中央处理单元这一术语的由来。现代的微处理器把这些部件都集成到一块很小的硅片上。CPU、中央处理器在本书中均可以交替使用。

（编者注：本节后面几段由编者补译）

Microprocessors operate on binary data; that is data composed of ones and zeros.

微处理器处理的是二进制的数据，这些二进制的数据仅由 1 和 0 组合而成。

These ones and zeros correspond to

1 和 0 与电路开关中开与关的状态是相

electrical switches being either on or off. Just as 42 is a decimal number meaning ``4 10s and 2 units", a binary number is a series of binary digits each one representing a power of 2. In this context, a power means the number of times that a number is multiplied by itself. 10 to the power 1 (10^1) is 10, 10 to the power 2 (10^2) is 10*10, 10^3 is 10*10*10 and so on. Binary 0001 is decimal 1, binary 0010 is decimal 2, binary 0011 is 3, binary 0100 is 4 and so on. So, 42 decimal is 101010 binary or (2 + 8 + 32 or $2^1 + 2^3 + 2^5$). Rather than using binary to represent numbers in computer programs, another base, hexadecimal is usually used.

对应的。正如十进制数 42 表示“4 个 10 与 2 个 1 一样”，一个二进制数是一串二进制数字，每一个二进制数字都表示一个 2 的次幂。在这种上下文中，次幂表示一个数自乘的次数。10 的一次幂（10^1）是 10，10 的 2 次幂（10^2）是 10*10，10 的 3 次幂（10^3）是 10*10*10 以此类推。在二进制中，0011 是 3，0100 是 4 以此类推。因此，十进制数 42 用二进制表示是 101010 或者 2 + 8 + 32 或者 $2^1 + 2^3 + 2^5$。除了在计算机程序中用二进制表示数，其它时候通常使用 16 进制。

In this base, each digital represents a power of 16. As decimal numbers only go from 0 to 9 the numbers 10 to 15 are represented as a single digit by the letters A, B, C, D, E and F. For example, hexadecimal E is decimal 14 and hexadecimal 2A is decimal 42 (two 16s) + 10). Using the C programming language notation (as I do throughout this book) hexadecimal numbers are prefaced by “0x”; hexadecimal 2A is written as 0x2A.

在这个条件下，每一个数字表示一个 16 的次幂。由于十进制数字只能从 0 到 9，所以 10 到 15 就用字母 A、B、C、D、E 及 F 来表示。例如，十六进制数 E 就是十进制数 14，十六进制数 2A 就是十进制数 42（2*16+10）。用 C 语言来表示（在本书就我都会这样做）十六进制数的前缀是“0x”，十六进制数 2A 被写作 0x2A。

Microprocessors can perform arithmetic operations such as add, multiply and divide and logical operations such as “is X greater than Y?”.

微处理器可以完成算术运算，比如：加、减、乘、除；以及逻辑运算，比如：“X 是比 Y 大吗？”

The processor's execution is governed by an external clock. This clock, the system clock, generates regular clock pulses to the processor and, at each clock pulse, the processor does some work. For example, a processor could execute an instruction every clock pulse. A processor's speed is described in terms of the rate of the system clock ticks. A 100Mhz processor will receive 100,000,000 clock ticks every second. It is misleading to describe the power of a CPU by its clock rate as different processors perform different amounts of work per clock tick. However, all things being equal,

处理器的执行受外部的一个时钟控制。这个时钟，即系统时钟，会有规律的向处理器发送脉冲，每个时钟脉冲，处理器都会完成一些工作。例如，一个处理器可以每个时钟脉冲执行一条指令。一个处理器的速度常用系统时钟的滴达速率来描述。一个 100MHz 的处理器每秒会收到 100,000,000 个时间滴达。不过，用时钟滴达速率来描述处理器的性能是不对的，因为不同的处理器每个时钟滴达所完成的工作量是不同的。不过，如果所有情况都相同，时钟速率越快，意味着处理器性能越强。处理器执行的指令都非常简单，比如“将内存地址为 X 处的内容读到寄

a faster clock speed means a more powerful processor. The instructions executed by the processor are very simple; for example ``read the contents of memory at location X into register Y". Registers are the microprocessor's internal storage, used for storing data and performing operations on it. The operations performed may cause the processor to stop what it is doing and jump to another instruction somewhere else in memory. These tiny building blocks give the modern microprocessor almost limitless power as it can execute millions or even billions of instructions a second.

存器 Y 里”。寄存器是微处理器内部的存储空间，用来存放数据，在上面执行一个操作。有些操作可能会引起处理器停止当前的工作而跳到内存中的另一条指令去执行。紧凑的 blocks（编者注：抱歉，对于此处的确切意思实在无法把握）的设计使现代的微处理器几乎获得了无限的性能，它们可以每秒执行数百万条甚至数十亿条的指令。

The instructions have to be fetched from memory as they are executed. Instructions may themselves reference data within memory and that data must be fetched from memory and saved there when appropriate.

这些指令必须从内存中取出来然后执行。指令自身可能会使用到内存中的数据，这些数据也必须从内存中取出来，以及在适当的时候保存在那里。

The size, number and type of register within a microprocessor is entirely dependent on its type. An Intel 4086 processor has a different register set to an Alpha AXP processor; for a start, the Intel's are 32 bits wide and the Alpha AXP's are 64 bits wide. In general, though, any given processor will have a number of general purpose registers and a smaller number of dedicated registers. Most processors have the following special purpose, dedicated, registers:

处理器中寄存器的大小，数量以及类型都取决于处理器的类型。一个 Intel 4086 处理器拥有一套同 Alpha AXP 不同的寄存器。首先，Intel 的处理器是 32 位的，而 Alpha AXP 的处理器是 64 位的。不过，通常来讲，任何一个处理器都有一些通用寄存器以及少量的专用寄存器。许多处理器有下面一些专用寄存器：

Program Counter (PC)

This register contains the address of the next instruction to be executed. The contents of the PC are automatically incremented each time an instruction is fetched,

程序计数器（PC）

这个寄存器保存有下一条将被执行的指令的地址。每次取出指令后，PC 中的内容会自动增加。

Stack Pointer (SP)

Processors have to have access to large amounts of external read/write random access memory (RAM) which facilitates temporary storage of data. The stack is a way of easily saving and restoring temporary values in

堆栈指针（SP）

处理器经常需要访问外部的大量的可读写的随机存储器（RAM），它们非常适合临时存储数据。在处部存储器中，堆栈是一个很简单存储及恢复数据的方法。通常，处理器有专门的指令让你把数据推入栈中然后再将

external memory. Usually, processors have special instructions which allow you to push values onto the stack and to pop them off again later. The stack works on a last in first out (LIFO) basis. In other words, if you push two values, x and y, onto a stack and then pop a value off of the stack then you will get back the value of y.

其弹出。堆栈遵循后进先出（LIFO）的原则。换句话说，如果你推入两个值，x 与 y 到堆栈里，然后你从堆栈里弹出一个值，你弹出的将是 y 值。

Some processor's stacks grow upwards towards the top of memory whilst others grow downwards towards the bottom, or base, of memory. Some processor's support both types, for example ARM.

有些处理器的堆栈朝着内存顶部向上增长，有些朝着内存底部向下增长，也有一些处理器两种增长方式都支持，比如说 ARM。

Processor Status (PS)

Instructions may yield results; for example “is the content of register X greater than the content of register Y?” will yield true or false as a result. The PS register holds this and other information about the current state of the processor. For example, most processors have at least two modes of operation, kernel (or supervisor) and user. The PS register would hold information identifying the current mode.

处理器状态（PS）

指令可能会产生一些结果，比如“X 寄存器中的内容比 Y 寄存器中的内容大吗”会产生真或假作为结果。PS 寄存器保存了这样的信息以及其它一些有关处理器当前状态的信息。比如，许多处理器都至少拥有两种工作模式，内核模式（或者管理模式）以及用户模式。PS 寄存器需要保存用来表示当前是在哪个模式下工作的信息。

1.2 Memory

All systems have a memory hierarchy with memory at different speeds and sizes at different points in the hierarchy. The fastest memory is known as cache memory and is what it sounds like - memory that is used to temporarily hold, or cache, contents of the main memory. This sort of memory is very fast but expensive, therefore most processors have a small amount of on-chip cache memory and more system based (on-board) cache memory. Some processors have one cache to contain both instructions and data, but others have two, one for instructions and the other for data. The Alpha AXP processor has two internal memory caches; one for data (the D-Cache) and one for

1.2 存储器

所有的系统都有一个存储器的层次结构。每一层的速度和大小不一样。最快的存储器是缓冲(cache)存储器，用来暂时存放主存储器的内容。这种存储器非常快但价格很贵，因此大多数处理器在芯片中只含有较少的 cache 存储器。更多的则在系统板上。有些处理器在 cache 中混合存放数据和指令，有些则分开存放，一个 cache 为指令，其他一个 cache 为数据。Alpha 处理器中含有两个内部存储 cache，D-cache 存放数据，I-cache 存放指令。外部的 cache(B-cache)混合存放指令与数据。最后，相对于外部 cache 而言非常慢的是主存。如果与 CPU 内部的 cache 比较，主存肯定就像在爬一样。

instructions (the I-Cache). The external cache (or B-Cache) mixes the two together. Finally there is the main memory which relative to the external cache memory is very slow. Relative to the on-CPU cache, main memory is positively crawling.

The cache and main memories must be kept in step (coherent). In other words, if a word of main memory is held in one or more locations in cache, then the system must make sure that the contents of cache and memory are the same. The job of cache coherency is done partially by the hardware and partially by the operating system. This is also true for a number of major system tasks where the hardware and software must cooperate closely to achieve their aims.

缓冲与主存之间必须保持一致性(coherent)。换句话说，如果主存中的一个字在 cache 中的一个或多个地方，系统必须保证 cache 中的内容与主存中的一致。cache 一致性的工作一部份是由硬件完成的，一部份是由操作系统完成的。这一点对于大多数主要的系统任务来说都是一样的。系统中的软硬件必须互相合作完成功能。

1.3 Buses

The individual components of the system board are interconnected by multiple connection systems known as buses. The system bus is divided into three logical functions; the address bus, the data bus and the control bus. The address bus specifies the memory locations (addresses) for the data transfers. The data bus holds the data transferred. The data bus is bidirectional; it allows data to be read into the CPU and written from the CPU. The control bus contains various lines used to route timing and control signals throughout the system. Many flavours of bus exist, for example ISA and PCI buses are popular ways of connecting peripherals to the system.

1.3 总线

系统板上的独立的部件通过总线相连。系统总线按逻辑功能分成三种类型：地址总线，数据总线和控制总线。地址总线指定数据传送的地址。数据总线包含要传送的数据。数据总线是双向的，允许数据读进 CPU 和从 CPU 写出。控制总线包含一些信号线，用来控制时序和系统中的其他控制信号。还有许多其它总线，比如 ISA 及 PCI 总线常常被系统用来连接外部设备。

1.4 Controllers and Peripherals

Peripherals are real devices, such as graphics cards or disks controlled by controller

1.4 控制器和外设

外设是真实的设备，例如显示卡或磁盘，它们被主板上的控制芯片或者那些插在主板

chips on the system board or on cards plugged into it. The IDE disks are controlled by the IDE controller chip and the SCSI disks by the SCSI disk controller chips and so on. These controllers are connected to the CPU and to each other by a variety of buses. Most systems built now use PCI and ISA buses to connect together the main system components. The controllers are processors like the CPU itself, they can be viewed as intelligent helpers to the CPU. The CPU is in overall control of the system.

上的板卡上的控制芯片所控制。IDE 磁盘由 IDE 控制芯片控制,SCSI 磁盘被 SCSI 磁盘控制芯片所控制，以此类推。控制器与 CPU 之间，控制器与控制器之间，通过总线相连。大多数系统通过 PCI 和 ISA 总线将主要的系统部件相连。控制器与 CPU 一样，本身就是一种处理器，它们可以看作是 CPU 的智能助手。CPU 是系统的控制中心。

All controllers are different, but they usually have registers which control them. Software running on the CPU must be able to read and write those controlling registers. One register might contain status describing an error. Another might be used for control purposes; changing the mode of the controller. Each controller on a bus can be individually addressed by the CPU, this is so that the software device driver can write to its registers and thus control it. The IDE ribbon is a good example, as it gives you the ability to access each drive on the bus separately. Another good example is the PCI bus which allows each device (for example a graphics card) to be accessed independently.

所有的控制器都不相同，但通常它们都有一些可以用来控制它们的寄存器。在 CPU 之上运行的软件必须能够读写这些控制寄存器。一个寄存器可能包含一个用来描述错误的状态，另一个寄存器可能被用来作为控制，例如改变控制器的模式。总线上的每个控制器都可以被 CPU 单独寻址，因而，设备驱动程序软件可以写上述寄存器以控制这些控制器。IDE 扁平电缆就是一个很好的例子，它给了你可以单独访问总线上每一个设备的能力。另一个好的例子是 PCI 总线，它使每一个设备（例如一个图形显示卡）可以被独立访问。

1.5 Address Spaces

The system bus connects the CPU with the main memory and is separate from the buses connecting the CPU with the system's hardware peripherals. Collectively the memory space that the hardware peripherals exist in is known as I/O space. I/O space may itself be further subdivided, but we will not worry too much about that for the moment. The CPU can access both the system space memory and the I/O space memory, whereas the controllers themselves can only access system memory indirectly and then only with the help of the

1.5 地址空间

系统总线将 CPU 与主存相连,这与将 CPU 与系统硬件外设相连的总线是分开的。硬件外设所占据的存储空间叫做 I/O 空间。I/O 空间自身还可以被细分，但我们不用在这一问题上深究。CPU 既可以存取系统存储空间，也可以存取 I/O 空间。然而，控制器只能在 CPU 的帮助下间接地访问系统主存。从设备的观点看，例如软盘控制器，它只能看见其控制寄存器所在的(ISA)空间的地址，不能看到系统主存。一般而言，CPU 使用不同的指令系统来存取系统存储器和 I/O 空间。例如，可能存在一个指令“从 I/O 地址 0x3f0 读一个

CPU. From the point of view of the device, say the floppy disk controller, it will see only the address space that its control registers are in (ISA), and not the system memory. Typically a CPU will have separate instructions for accessing the memory and I/O space. For example, there might be an instruction that means “read a byte from I/O address 0x3f0 into register X”. This is exactly how the CPU controls the system's hardware peripherals, by reading and writing to their registers in I/O space. Where in I/O space the common peripherals (IDE controller, serial port, floppy disk controller and so on) have their registers has been set by convention over the years as the PC architecture has developed. The I/O space address 0x3f0 just happens to be the address of one of the serial port's (COM1) control registers.

字节到寄存器 x”。CPU 就是这样通过读和写 I/O 空间中的外设的寄存器，来控制系统外设的。在 I/O 空间中，一些常用的硬件外设（IDE 控制器、串行口、软盘驱动器等）的寄存器地址，在多年的 PC 体系结构发展中已经被固定下来了。例如，I/O 地址空间中的地址 0x3f0 正好是串口一（com1）的一个控制寄存器的地址。

There are times when controllers need to read or write large amounts of data directly to or from system memory. For example when user data is being written to the hard disk. In this case, Direct Memory Access (DMA) controllers are used to allow hardware peripherals to directly access system memory but this access is under strict control and supervision of the CPU.

有时候，控制器需要在系统主存之间传送大量的数据。例如将数据写入硬盘。这种情况下，直接储存器访问（DMA）控制器被用来使得硬件外设直接存取系统内存。但是这个过程是在 CPU 的严格控制和监督之下的。

1.6 Timers

All operating systems need to know the time and so the modern PC includes a special peripheral called the Real Time Clock (RTC). This provides two things: a reliable time of day and an accurate timing interval. The RTC has its own battery so that it continues to run even when the PC is not powered on, this is how your PC always “knows” the correct date and time. The interval timer allows the operating system to accurately schedule essential work.

1.6 定时器

所有的操作系统都需要知道时间。所以现代PC中含有一个特殊的设备叫做实时时钟(Real Time Clock)。RTC 提供两种功能：每天的可靠的时间和精确的时间间隔。RTC 有其自己的电池，因此即使 PC 没开电源，RTC 也在运行。这就是为什么你的 PC 知道正确的日期及时间的原因。时间间隔定时器使得操作系统可以准确的调度必要的工作。

【未完待续 • 责任编辑：iamxiaohan@hitbbs】

Linux 核心（The Linux Kernel）——第二章

原著: David A Rusling

翻译: 毕昕、胡宁宁、仲盛、赵振平、周笑波、李群、陈怀临

Chapter 2 Software Basics

A program is a set of computer instructions that perform a particular task. That program can be written in assembler, a very low level computer language, or in a high level, machine independent language such as the C programming language. An operating system is a special program which allows the user to run applications such as spreadsheets and word processors. This chapter introduces basic programming principles and gives an overview of the aims and functions of an operating system.

第二章 软件基础

程序就是一组执行特定任务的计算机指令。程序既可以用非常低级的计算机语言——汇编语言，也可以用高级的、独立于机器的语言如 C 语言来编写。操作系统是一种特殊的程序，它允许用户运行各种应用程序如制表程序和字处理程序。本章介绍基本的程序设计原理，并对操作系统的目标和功能做一综述。

2.1 Computer Languages

2.1.1 Assembly Languages

The instructions that a CPU fetches from memory and executes are not at all understandable to human beings. They are machine codes which tell the computer precisely what to do. The hexadecimal number 0x89E5 is an Intel 80486 instruction which copies the contents of the ESP register to the EBP register. One of the first software tools invented for the earliest computers was an assembler, a program which takes a human readable source file and assembles it into machine code. Assembly languages explicitly handle registers and operations on data and they are specific to a particular microprocessor. The assembly language for an Intel X86 microprocessor is very different to the assembly language for an Alpha AXP microprocessor. The following Alpha AXP assembly code shows the sort of operations that a program can

2.1 计算机语言

2.1.1 汇编语言

CPU 从内存中取出并运行的指令对人来说根本无法理解。它们是精确指示机器如何操作的机器代码。例如，十六进制数 0x89E5 是 Intel 80486 的一条指令，它指示把 ESP 寄存器的内容拷贝到 EBP 寄存器中。汇编器是最早发明的软件工具之一，它输入人类可以理解的源代码，汇编为机器代码。汇编语言显式地处理寄存器和数据操作，与特定的微处理器相关（应为与特定的处理器相关--译者注）。Intel X86 微处理器的汇编语言就与 Alpha AXP 微处理器的汇编语言大相径庭。以下 Alpha AXP 汇编代码表示了程序可以进行的一种操作：

perform:

```
    ldr r16, (r15)     ; Line 1
    ldr r17, 4(r15)    ; Line 2
    beq r16,r17,100    ; Line 3
    str r17, (r15)     ; Line 4
100:                   ; Line 5
```

```
    ldr r16, (r15)     ; Line 1
    ldr r17, 4(r15)    ; Line 2
    beq r16,r17,100    ; Line 3
    str r17, (r15)     ; Line 4
100:                   ; Line 5
```

The first statement (on line 1) loads register 16 from the address held in register 15. The next instruction loads register 17 from the next location in memory. Line 3 compares the contents of register 16 with that of register 17 and, if they are equal, branches to label 100. If the registers do not contain the same value then the program continues to line 4 where the contents of r17 are saved into memory. If the registers do contain the same value then no data needs to be saved. Assembly level programs are tedious and tricky to write and prone to errors. Very little of the Linux kernel is written in assembly language and those parts that are written only for efficiency and they are specific to particular microprocessors.

第一条指令（见第一行）把寄存器 15 中存放的地址中的内容装入寄存器 16。下一条指令把内存中下一个位置的内容装入寄存器 17。第三行把寄存器 16 和寄存器 17 的内容比较，如果相等，则转向标号 100 处。如果两个寄存器包含数值不等，程序继续运行第四行，把寄存器 17 的内容存到内存。如果两个寄存器包含数值相等，那么没有数据需要保存。编写汇编语言程序枯燥乏味、技巧性强而且易于出错。Linux 核心只有很少的一点用汇编语言编写，目的是为了效率，或者用在一些与特定处理器相关的地方。

2.1.2 The C Programming Language and Compiler

2.1.2 C 语言和编译器

Writing large programs in assembly language is a difficult and time consuming task. It is prone to error and the resulting program is not portable, being tied to one particular processor family. It is far better to use a machine independent language like C. C allows you to describe programs in terms of their logical algorithms and the data that they operate on. Special programs called compilers read the C program and translate it into assembly language, generating machine specific code from it. A good compiler can generate assembly instructions that are very nearly as efficient as those written by a good assembly programmer. Most of the Linux kernel is written in the C

用汇编语言编写大型程序十分困难而且消耗大量时间。这样做易于出错，得到的程序也无法移植，而被限制在特定的处理器族上。用独立于机器的语言如 C，会好得多。C 允许你用逻辑算法和其操作的数据结构来描述程序。称之为编译器的特定程序读入 C 程序，并把它翻译成汇编语言，生成相应的机器代码。好的编译器所产生的汇编指令的效率接近于好的汇编语言程序员编写的汇编语言程序。大部份 Linux 核心是用 C 语言编写的。以下的 C 片段：

language. The following C fragment:

```
        if (x != y)
                x = y ;
```

performs exactly the same operations as the previous example assembly code. If the contents of the variable x are not the same as the contents of variable y then the contents of y will be copied to x. C code is organized into routines, each of which perform a task. Routines may return any value or data type supported by C. Large programs like the Linux kernel comprise many separate C source modules each with its own routines and data structures. These C source code modules group together logical functions such as filesystem handling code.

C supports many types of variables, a variable is a location in memory which can be referenced by a symbolic name. In the above C fragment x and y refer to locations in memory. The programmer does not care where in memory the variables are put, it is the linker (see below) that has to worry about that. Some variables contain different sorts of data, integer and floating point and others are pointers.

Pointers are variables that contain the address, the location in memory of other data. Consider a variable called x, it might live in memory at address 0x80010000. You could have a pointer, called px, which points at x. px might live at address 0x80010030. The value of px would be 0x80010000: the address of the variable x.

C allows you to bundle together related variables into data structures. For example,

```
        struct {
                int i ;
                char b ;
```

```
        if (x != y)
                x = y ;
```

与前一个例子中汇编代码的操作完全相同。如果变量 x 和 y 的内容并不完全相同，就把 y 的内容拷贝给 x。C 代码组织为例程，每一个例程执行一个任务。例程可以返回 C 支持的任何数值或者数据类型。像 Linux 核心这样的大型程序包含很多独立的 C 源模块，每个模块都有自己的例程和数据结构。这些 C 源代码模块把像文件系统处理这样的逻辑功能代码组合在一起。（编者注：这里直译不太好理解，其实意思就是把一些相关模块组织起来，完成像文件系统这样的逻辑功能。）

C 支持很多类型的变量。所谓变量，就是内存中的一个位置，可以用符号名字来引用。在以上 C 片段中，x 和 y 指引了内存中的位置。程序员不关心变量究竟存放在内存中的何处，这是连接器（见下面所述）的任务。一些变量含有不同类型的数据，整数和浮点数，另一些则是指针。

指针就是包含地址——其它数据在内存中的位置——的变量。考虑叫做 x 的变量，它可能处于内存地址 0x80010000。你可以有一个指针，叫做 px，指向 x。px 可能处于地址 0x80010030，而 px 的值是 0x80010000，即变量 x 的地址。

C 允许你把相关的变量绑在一起，形成数据结构。例如，

```
        struct {
                int i ;
                char b ;
```

```
        } my_struct ;
```

is a data structure called my_struct which contains two elements, an integer (32 bits of data storage) called i and a character (8 bits of data) called b.

```
        } my_struct ;
```

是一个叫做 my_struct 的数据结构，它包含两个元素：一个叫做 i 的整数（32 位数据）和一个叫做 b 的字符（8 位数据）。

2.1.3 Linkers

Linkers are programs that link together several object modules and libraries to form a single, coherent, program. Object modules are the machine code output from an assembler or compiler and contain executable machine code and data together with information that allows the linker to combine the modules together to form a program. For example one module might contain all of a program's database functions and another module its command line argument handling functions. Linkers fix up references between these object modules, where a routine or data structure referenced in one module actually exists in another module. The Linux kernel is a single, large program linked together from its many constituent object modules.

2.1.3 连接器

连接器是一种程序，它可以把几个目标模块和库连接在一起，产生一个独立的、连贯的程序。目标模块是汇编器或编译器生成的机器代码输出，含有可执行的机器代码和数据，以及允许连接器把模块连接起来的信息。例如一个模块可能含有程序中所有的数据库函数，而另外一个则含有命令行参数处理函数。连接器负责解决目标模块之间的引用，例如一个模块中引用的例程或数据结构事实上在另外一个模块之中。Linux 核心就是一个与很多成员目标模块连接在一起的独立的大程序。

2.2 What is an Operating System?

Without software a computer is just a pile of electronics that gives off heat. If the hardware is the heart of a computer then the software is its soul. An operating system is a collection of system programs which allow the user to run application software. The operating system abstracts the real hardware of the system and presents the system's users and its applications with a virtual machine. In a very real sense the software provides the character of the system. Most PCs can run one or more operating systems and each one can have a very different look and feel. Linux is made up of a

2.2 什么是操作系统?

没有软件的计算机就是一堆发热的电子器件。如果说硬件是计算机的核心，那么软件就是计算机的灵魂。所谓操作系统，就是允许用户在其上运行应用软件的一组系统程序。操作系统对系统的真正硬件进行抽象，向系统的用户和应用程序给出一个虚拟机。在很现实的意义上说，软件提供了系统的特点。绝大部份 PC 能运行一个或多个操作系统，每一个操作系统都有一个完全不同的外观和风格。Linux 是由一批功能上分离的部件组成，其中明显的一个是核心本身。但是即使是核心，如果脱离库和外壳程序（编者注：其实就是 Shell 程序，这里编者认为不译最好，但考

number of functionally separate pieces that, together, comprise the operating system. One obvious part of Linux is the kernel itself; but even that would be useless without libraries or shells.

虑到不少材料都将其译成外壳程序，也为了尊重原作者故而未做改动。另外本文不少地方将 Kernel 译为核心，其实就是常说的内核）也是没有用的。

In order to start understanding what an operating system is, consider what happens when you type an apparently simple command:

为了开始理解什么是操作系统，请考虑当你敲入以下的简单命令时会发生的情况：

```
$ ls
Mail    c    images    perl
docs    tcl
$
```

```
$ ls
Mail    c    images    perl
docs    tcl
$
```

The $ is a prompt put out by a login shell (in this case bash). This means that it is waiting for you, the user, to type some command. Typing ls causes the keyboard driver to recognize that characters have been typed. The keyboard driver passes them to the shell which processes that command by looking for an executable image of the same name. It finds that image, in /bin/ls. Kernel services are called to pull the ls executable image into virtual memory and start executing it. The ls image makes calls to the file subsystem of the kernel to find out what files are available. The filesystem might make use of cached filesystem information or use the disk device driver to read this information from the disk. It might even cause a network driver to exchange information with a remote machine to find out details of remote files that this system has access to (filesystems can be remotely mounted via the Networked File System or NFS). Whichever way the information is located, ls writes that information out and the video driver displays it on the screen.

这里$是由登录外壳程序（在此例为 bash）给出的提示符。这意味着它在等待你——用户——敲入命令。敲入 ls 后，键盘驱动程序识别出已经有字符输入。键盘驱动程序把这些字符传给外壳程序，外壳程序则通过寻找可执行程序的映像来处理这个命令。它在/bin/ls 发现了映像，于是调用核心服务来把 ls 可执行程序的映像拖入虚拟内存，并开始执行。ls 的映像调用核心的文件子系统，以找出有哪些文件可以获得。文件系统有可能要充分使用放在被缓存的文件系统信息或者用磁盘驱动程序从磁盘读出这些信息，甚至可能用网络驱动程序与远程机器交换信息，以找出本系统能够存取的远程文件的细节（文件系统可以通过网络文件系统或 NFS 来远程挂载）。无论是用哪种方式定位信息，ls 都会把信息写出来，由视频驱动程序把它显示在屏幕上。

All of the above seems rather complicated but it shows that even most simple commands reveal that an operating system is in fact a co-operating set of functions that together give

以上看起来很复杂，但是说明了一个道理：即使是最简单的命令，也需要相当的处理，操作系统事实上是一组互相合作的函数，它们在整体上给用户以一个系统的完整印

you, the user, a coherent view of the system.

象。（以上一句是根据译者理解翻译的，未必忠实于原文——译者注）

2.2.1 Memory management

With infinite resources, for example memory, many of the things that an operating system has to do would be redundant. One of the basic tricks of any operating system is the ability to make a small amount of physical memory behave like rather more memory. This apparently large memory is known as virtual memory. The idea is that the software running in the system is fooled into believing that it is running in a lot of memory. The system divides the memory into easily handled pages and swaps these pages onto a hard disk as the system runs. The software does not notice because of another trick, multi-processing.

2.2.1 内存管理

如果有无限的资源，例如内存，很多操作系统需要做的事情都是多余的。操作系统的一个基本技巧是使一小块内存看起来像很多内存。这种表面上看起来大的内存称为虚拟内存。其思想是使系统中运行的软件以为它在很多内存上运行。系统把内存分成很多容易处理的页面，在系统运行的时候，把一些页面交换到硬盘上。由于另外的一个技巧——多道处理，软件注意不到这一点。

2.2.2 Processes

A process could be thought of as a program in action, each process is a separate entity that is running a particular program. If you look at the processes on your Linux system, you will see that there are rather a lot. For example, typing ps shows the following processes on my system:

2.2.2 进程

进程可以想象为一个活动中的程序。每一个进程是一个独立的实体，在运行一个特定程序。如果你看看你的 Linux 系统中的进程，你就会发现一大堆。例如，在我的系统中敲入 ps 可以显示如下进程：

```
$ ps
  PID TTY STAT  TIME COMMAND
  158 pRe 1     0:00 -bash
  174 pRe 1     0:00 sh /usr/X11R6/bin/startx
  175 pRe 1     0:00 xinit /usr/X11R6/lib/X11/xinit/xinitrc --
  178 pRe 1 N   0:00 bowman
  182 pRe 1 N   0:01 rxvt -geometry 120x35 -fg white -bg black
  184 pRe 1 <   0:00 xclock -bg grey -geometry -1500-1500 -padding 0
  185 pRe 1 <   0:00 xload -bg grey -geometry -0-0 -label xload
  187 pp6 1     9:26 /bin/bash
  202 pRe 1 N   0:00 rxvt -geometry 120x35 -fg white -bg black
  203 ppc 2     0:00 /bin/bash
 1796 pRe 1 N   0:00 rxvt -geometry 120x35 -fg white -bg black
 1797 v06 1     0:00 /bin/bash
 3056 pp6 3 <   0:02 emacs intro/introduction.tex
```

```
 3270 pp6 3      0:00 ps
$
```

If my system had many CPUs then each process could (theoretically at least) run on a different CPU. Unfortunately, there is only one so again the operating system resorts to trickery by running each process in turn for a short period. This period of time is known as a time-slice. This trick is known as multi-processing or scheduling and it fools each process into thinking that it is the only process. Processes are protected from one another so that if one process crashes or malfunctions then it will not affect any others. The operating system achieves this by giving each process a separate address space which only they have access to.

如果我的系统有很多 CPU，每个进程（至少在理论上）可以运行在一个不同的 CPU 上。不幸的是，我只有一个 CPU，所以系统只能求助于让每个进程轮流运行一小段时间的办法。这一小段时间称为时间片。这种技巧称为多道处理或者调度，它使得每个进程以为自己是唯一的进程。在进程之间进行保护，以便当一个进程崩溃或者错误时，不会影响其它进程。操作系统给每个进程一个单独的、只有它自己能存取的地址空间，以达到这样的目的。

2.2.3 Device drivers

Device drivers make up the major part of the Linux kernel. Like other parts of the operating system, they operate in a highly privileged environment and can cause disaster if they get things wrong. Device drivers control the interaction between the operating system and the hardware device that they are controlling. For example, the filesystem makes use of a general block device interface when writing blocks to an IDE disk. The driver takes care of the details and makes device specific things happen. Device drivers are specific to the controller chip that they are driving which is why, for example, you need the NCR810 SCSI driver if your system has an NCR810 SCSI controller.

2.2.3 设备驱动程序

设备驱动程序构成了 Linux 核心的主要部份。就像操作系统的其它部份一样，设备驱动程序在高特权级的环境下运行，一旦发生错误就可能造成危险。设备驱动程序控制操作系统和其控制的硬件设备之间的互相作用。例如，在把块写到 IDE 硬盘时，文件系统使用一个通用块设备接口。驱动程序负责细节，控制与设备相关的事情。设备驱动程序是针对其控制的特定芯片的，所以如果你有一个 NCR810 SCSI 控制器，那么你就需要一个 NCR810 SCSI 驱动程序。

2.2.4 The Filesystems

In Linux, as it is for Unix, the separate filesystems that the system may use are not accessed by device identifiers (such as a drive number or a drive name) but instead they are

2.2.4 文件系统

在 Linux 中，就像在 Unix 中一样，系统可以使用的不同的文件系统，并非通过设备标识来存取（例如驱动器号或驱动器名），而是被组织在一个单一的分层树结构里，每个

combined into a single hierarchical tree structure that represents the filesystem as a single entity. Linux adds each new filesystem into this single filesystem tree as they are mounted onto a mount directory, for example /mnt/cdrom. One of the most important features of Linux is its support for many different filesystems. This makes it very flexible and well able to coexist with other operating systems. The most popular filesystem for Linux is the EXT2 filesystem and this is the filesystem supported by most of the Linux distributions.

文件系统用一个实体来表示。文件系统通过把新的文件系统 mount 在某个目录下——例如/mnt/cdrom，从而把新的文件系统加入树中。Linux 的最重要特点之一是支持多个不同的文件系统，这使得其灵活性好，易于与其它操作系统共存。最广为人知的 Linux 文件系统是EXT2,受到所发行的绝大部份的Linux的支持。

A filesystem gives the user a sensible view of files and directories held on the hard disks of the system regardless of the filesystem type or the characteristics of the underlying physical device. Linux transparently supports many different filesystems (for example MS-DOS and EXT2) and presents all of the mounted files and filesystems as one integrated virtual filesystem. So, in general, users and processes do not need to know what sort of filesystem that any file is part of, they just use them.

文件系统屏蔽了下层物理设备或文件系统类型的细节，给予用户察看硬盘上文件和目录的一个清楚视角。Linux 透明地支持多种不同文件系统（例如 MS-DOS 和 EXT2），并把所有 mount 上的文件和文件系统都组织在一个虚拟文件系统之中。所以，一般而言，用户和进程并不需要知道哪个文件在哪个文件系统之中，只需要直接去用它就可以了。

The block device drivers hide the differences between the physical block device types (for example, IDE and SCSI) and, so far as each filesystem is concerned, the physical devices are just linear collections of blocks of data. The block sizes may vary between devices, for example 512 bytes is common for floppy devices whereas 1024 bytes is common for IDE devices and, again, this is hidden from the users of the system. An EXT2 filesystem looks the same no matter what device holds it.

块设备驱动程序隐藏了物理块设备类型之间的差别（例如 IDE 和 SCSI 的差别），从文件系统的角度来看，物理设备只是数据块的线形聚集。不同设备的块大小不同，例如软盘通常是512字节,而IDE设备通常是1024字节，但是系统的用户是看不到这一点的。无论放在什么设备上,EX2 文件系统看起来都一样。

2.3 Kernel Data Structures 2.3 核心数据结构

The operating system must keep a lot of information about the current state of the system. As things happen within the system

操作系统必须保存关于系统当前状态的很多信息。在系统中发生了事情之后，这些数据结构必须修改，以反映当前的真实状况。

these data structures must be changed to reflect the current reality. For example, a new process might be created when a user logs onto the system. The kernel must create a data structure representing the new process and link it with the data structures representing all of the other processes in the system.

例如，在一个用户登录到系统之后，可能会创建一个新的进程。核心必须创建表示新进程的数据结构，并把它和表示系统中所有其它进程的数据结构连接在一起。

Mostly these data structures exist in physical memory and are accessible only by the kernel and its subsystems. Data structures contain data and pointers: addresses of other data structures or the addresses of routines. Taken all together, the data structures used by the Linux kernel can look very confusing. Every data structure has a purpose and although some are used by several kernel subsystems, they are more simple than they appear at first sight.

通常这些数据结构存在于物理内存之中，只能由核心及其子系统存取。数据结构包括数据和指针——其它数据结构的地址或例程的地址。总而言之，Linux 使用的数据结构看起来很复杂难懂。虽然其中某些可能为几个核心子系统所使用，但是每个数据结构都有其目的，所以这些数据结构事实上比刚看起来要简单。（以上一句是根据译者的理解翻译，未必正确和符合原文——译者注）

Understanding the Linux kernel hinges on understanding its data structures and the use that the various functions within the Linux kernel makes of them. This book bases its description of the Linux kernel on its data structures. It talks about each kernel subsystem in terms of its algorithms, its methods of getting things done, and their usage of the kernel's data structures.

理解 Linux 核心依赖于理解其数据结构以及核心中各种函数对数据结构的使用。本书对 Linux 核心的描述就是基于其数据结构的。本书讨论了每个核心子系统的算法、完成工作的方法和对核心数据结构的使用。

2.3.1 Linked Lists

Linux uses a number of software engineering techniques to link together its data structures. On a lot of occasions it uses linked or chained data structures. If each data structure describes a single instance or occurrence of something, for example a process or a network device, the kernel must be able to find all of the instances. In a linked list a root pointer contains the address of the first data structure, or element, in the list and each data structure contains a pointer to the next element in the list. The last element's next pointer would be 0 or

2.3.1 链表

Linux 使用一系列软件工程技术来把数据结构连接起来。在很多情况下要使用链式的数据结构。如果每个数据结构描述某事物的一个实例或一次发生，例如一个进程或一个网络设备，那么核心就必须能够找到所有的实例。在链表中，根指针含有表中第一个数据结构，或者称为“元素”的地址，而每个数据结构都含有一个指向表中下一个元素的 next 指针。最后一个元素的 next 指针为 0 或 NULL，以示它已经是表尾。在双链表中，每个元素含有指向表中下一个元素的next指针和指向表中前一个元素的 previous 指针。

NULL to show that it is the end of the list. In a doubly linked list each element contains both a pointer to the next element in the list but also a pointer to the previous element in the list. Using doubly linked lists makes it easier to add or remove elements from the middle of list although you do need more memory accesses. This is a typical operating system trade off: memory accesses versus CPU cycles.

使用双链表方便了增加或删除表中间的元素，当然，内存的存取也增加了。这是一个典型的操作系统中的折中：消耗内存存取与消耗 CPU 周期的折中。

2.3.2 Hash Tables

Linked lists are handy ways of tying data structures together but navigating linked lists can be inefficient. If you were searching for a particular element, you might easily have to look at the whole list before you find the one that you need. Linux uses another technique, hashing to get around this restriction. A hash table is an array or vector of pointers. An array, or vector, is simply a set of things coming one after another in memory. A bookshelf could be said to be an array of books. Arrays are accessed by an index, the index is an offset into the array. Taking the bookshelf analogy a little further, you could describe each book by its position on the shelf; you might ask for the 5th book.

2.3.2 Hash 表

链表可以方便地把数据结构绑在一起，但是浏览链表效率很低。如果你想查找特定的一个元素，很可能不得不看完全表才找到需要的那个。Linux 使用 Hash 技术来避开这个限制。所谓 Hash 表，是一个指针的数组或者向量。这里说的数组或者向量，是指内存中的一组顺序存放的东西。因此，书架可以说成是书的数组。数组使用索引进行存取，而索引就是数组中位置的偏移量。把书架的比喻继续下去，你可以通过在书架上的位置来描述每本书。例如，你可以要求拿第 5 本书。

A hash table is an array of pointers to data structures and its index is derived from information in those data structures. If you had data structures describing the population of a village then you could use a person's age as an index. To find a particular person's data you could use their age as an index into the population hash table and then follow the pointer to the data structure containing the person's details. Unfortunately many people in the village are likely to have the same age and so the hash table pointer becomes a pointer to a chain or list of data structures each describing people of the same age. However, searching these shorter chains is still faster than searching

所谓 Hash 表，是指向数据结构的指针数组，采用数据结构中的信息作为 Hash 表的索引。如果你有描述村庄人口的数据结构，那么你可以采用人的年龄作为索引。为了寻找某个特定人的数据，你可以采用年龄作为人口 Hash 表的索引，然后按照指针找到含有此人详细情况的数据结构。（这里的指针相当于普通教科书上所说的 Hash 函数：Hash(ad)=“*ad”，——译者注）不幸的是，很可能村中的许多人年龄相同，所以 Hash 表的指针成了指向一个数据结构链的指针，链上的每个数据结构描述相同年龄的一个人。然而，搜索这些短链还是比搜索全部数据结构要快。

all of the data structures.

As a hash table speeds up access to commonly used data structures, Linux often uses hash tables to implement caches. Caches are handy information that needs to be accessed quickly and are usually a subset of the full set of information available. Data structures are put into a cache and kept there because the kernel often accesses them. There is a drawback to caches in that they are more complex to use and maintain than simple linked lists or hash tables. If the data structure can be found in the cache (this is known as a cache hit), then all well and good. If it cannot then all of the relevant data structures must be searched and, if the data structure exists at all, it must be added into the cache. In adding new data structures into the cache an old cache entry may need discarding. Linux must decide which one to discard, the danger being that the discarded data structure may be the next one that Linux needs.

由于 Hash 表加快了普通数据的存取，Linux 经常使用 Hash 表来实现 cache。cache 通常是总体信息的一部份，被抽出来需要加速存取。操作系统常用的数据结构常常放在 cache 中保存。cache 的不利之处在于使用和维护都比简单链表或 Hash 表更加复杂。假如能在 cache 中找到数据结构（称为“命中”），那么好得很。假如找不到，那么所有相关数据结构都要被搜索，如果最终能找到，那么该数据结构将被加入 cache 中。把新的数据结构加入 cache，可能会需要淘汰 cache 中一项旧的数据结构。Linux 必须决定到底淘汰哪一个才好，以尽可能避免恰恰淘汰出下次马上要用的数据结构。

2.3.3 Abstract Interfaces

The Linux kernel often abstracts its interfaces. An interface is a collection of routines and data structures which operate in a particular way. For example all network device drivers have to provide certain routines in which particular data structures are operated on. This way there can be generic layers of code using the services (interfaces) of lower layers of specific code. The network layer is generic and it is supported by device specific code that conforms to a standard interface.

2.3.3 抽象接口

Linux 核心经常对接口进行抽象。所谓接口，是例程和数据结构的集合，它通过某种特定方式进行操作。例如，所有的网络设备驱动程序必须提供操作某些特定数据结构的某些特定例程。在这种方式下，存在一些使用低层特殊代码提供的服务（接口）的通用代码层。例如网络层是通用的，受到遵循标准接口的与设备相关的代码的支持。（编者注：这几句话直译不太好理解，其实就是说下面的底层代码是与硬件相关的，但他们提供的服务却是标准化了的，与硬件无关的，这种服务就是一种被抽象了的接口。）

Often these lower layers register themselves with the upper layer at boot time. This registration usually involves adding a data structure to a linked list. For example each filesystem built into the kernel registers itself with the kernel at boot time or, if you are using

通常这些低层在启动时注册到高层。注册时一般要把一个数据结构加到一个链表中。例如，核心中内置的每个文件系统在启动时或者（如果使用模块的话）首次使用时注册到核心。通过查看文件 /proc/filesystems 能够看到哪些文件系统

modules, when the filesystem is first used. You can see which filesystems have registered themselves by looking at the file /proc/filesystems. The registration data structure often includes pointers to functions. These are the addresses of software functions that perform particular tasks. Again, using filesystem registration as an example, the data structure that each filesystem passes to the Linux kernel as it registers includes the address of a filesystem specific routine which must be called whenever that filesystem is mounted.

【未完待续 • 责任编辑：iamxiaohan@hitbbs】

已经注册了。注册的数据结构通常包含函数指针。这些函数指针都是进行特定任务的软件函数的地址。再次用文件系统注册作为一个例子，每个文件系统在注册时传给 Linux 核心的数据结构包含与文件系统相关的函数地址，这些函数在该文件系统 mount 时必须被调用。

操作系统概念（第六版）

原著：Abraham Silberschatz，Peter Baer Gkdalvin，GreGangne

翻译：吕建鹏[1]

译序

编辑要我说一下自己的想法，我想，好吧。首先还是介绍一下《操作系统概念》。我用的是第六版，仅从版本号上就可以了解这本书是怎样的经典了。就像《最终幻想》，人家能做到十几二十版，不容易。这本书讲的非常详细，提到了很多的东西，有些是在大学里从未见过的。以第七章为例，第七章讲的是进程同步，最后一节提到了从数据库系统中引入操作系统的一些技术，比如数据恢复技术以及原子事务的串行性。在翻译这些的时候需要了解这些英文词汇的标准译文，有意思的是，上述这些都来自数据库课程的幻灯片中。

随着翻译的进行，越来越感觉到底层技术的重要性。像 Symantec，放弃编译系统后做反病毒软件，逐步成为全球前几大软件公司（好像是前三大，至少进了前十）。过去很奇怪这是为什么，有时候感觉我们追赶他们真的太成问题，但是不断的出现很多强大的软件公司的崛起（比如：Helix）又让我觉得莫名其妙，他们怎么做到的？有人认为我们没有经验、还不能融入世界范围内，或者怎样怎样。但是只要你有好的产品，那你要考虑的就是最多能卖多少，而不是有没有人要。想想很久以前的 Turbo Pascal。我觉得是因为他们对底层太熟悉了，所以能做出很强大的软件。试想，如果 Borland 哪一天改行，去做财务管理软件、反病毒软件，或者办公软件，那会是个什么样子？国产软件现在似乎还有点摸不着北，我不想攻击谁，但是要说一点问题。前些日子下载了《超级解霸 V8》，安装，然后打开，然后卸载，最后是进 DOS 删除几个文件。为什么非要在系统内部安装一些我们不想要的东西呢？我可不可以不要？为什么卸载之后还留有残余文件呢？更可恨的是这些文件还在占用你的内存！还有瑞星的专杀工具，一个专杀工具何必要随着系统启动并自动工作呢？不要老提到消费者

[1] 吕建鹏 1981 年 10 月 28 日生（和 Bill Gates 是一天！）。2003 年毕业于西安邮电学院计算机系同年参军。电子邮件：tulipsys@sohu.com，主页：http://www.tulipsys.com

不支持正版，厂家不为消费者着想无异于自掘坟墓。我觉得我们尤其是想做大的软件公司还是应该真正的搞搞技术。

对于操作系统，我们必须要有自己的内核。这个东西关系太大了，给人留下后门，太危险了。不得不提到 Linux。有一个疑问，就是怎样才算是拥有了 Linux 的核心技术呢？现在很多地方老提到拥有 Linux 自主产权了、核心技术了，那么凭什么这么说呢？拥有源代码吗？如果是 Windows 的源代码，那还行，Linux 源代码谁都有！毕业设计即将结束时决定要搞操作系统，很希望能够写一个自己的内核，兼容 POSIX 标准，这样就可以使用 Linux 上的 Shell、X-Windows 等等了。《操作系统概念》才看到第七章，差的太远了。我这个人很懒，忙活一会儿就想打打游戏。不过，只要做下去总会有一点结果。这是一个老师告诉我的，我想，是这样的。

有点扯远了，不说了。

欢迎访问我的个人网站 TULIPSYS.com（http://www.tulipsys.com），不会做 Flash，也不会写 ASP，除了图片就是文字。

操作系统概念（第六版）—— 第一章 引论

原著：Abraham Silberschatz，Peter Baer Galvin，GreGangne

翻译：吕建鹏

操作系统是一种管理计算机硬件的程序，为应用程序提供了基本的运行条件，在计算机用户和计算机硬件之间扮演着中介的角色。操作系统的一个让人感到惊奇的方面就是它们所表现出来的丰富的多样性。大型计算机操作系统的首要设计目标是优化对硬件的使用。个人计算机（PC）操作系统则提供了对复杂的游戏、商业应用，以及对介于二者之间的所有应用软件的支持。手持计算机操作系统则向用户提供了一个运行计算机程序的便利的环境。这样，有些操作系统追求易用性，有些追求效率，还有些则是两者的折衷。

要理解什么是操作系统，必须要首先清楚它们是如何发展的。在本章中，我们将从最初的 hands-on 系统开始，经过多道程序系统和分时系统，到个人计算机系统和手持计算机系统，以此来追寻操作系统的发展。我们也将讨论操作系统的变种，比如，并行系统、实时系统和嵌入式系统。在跨越各种各样的平台时，我们将看到作为一种自然而然的解决方案，操作系统是怎样在早期的计算机系统中发展的。

1.1 什么是操作系统?

操作系统几乎是所有的计算机系统的一个重要组成部份。大体上可以将一个计算机系统分为四部分：硬件、操作系统、应用程序和用户（图 1.1）。

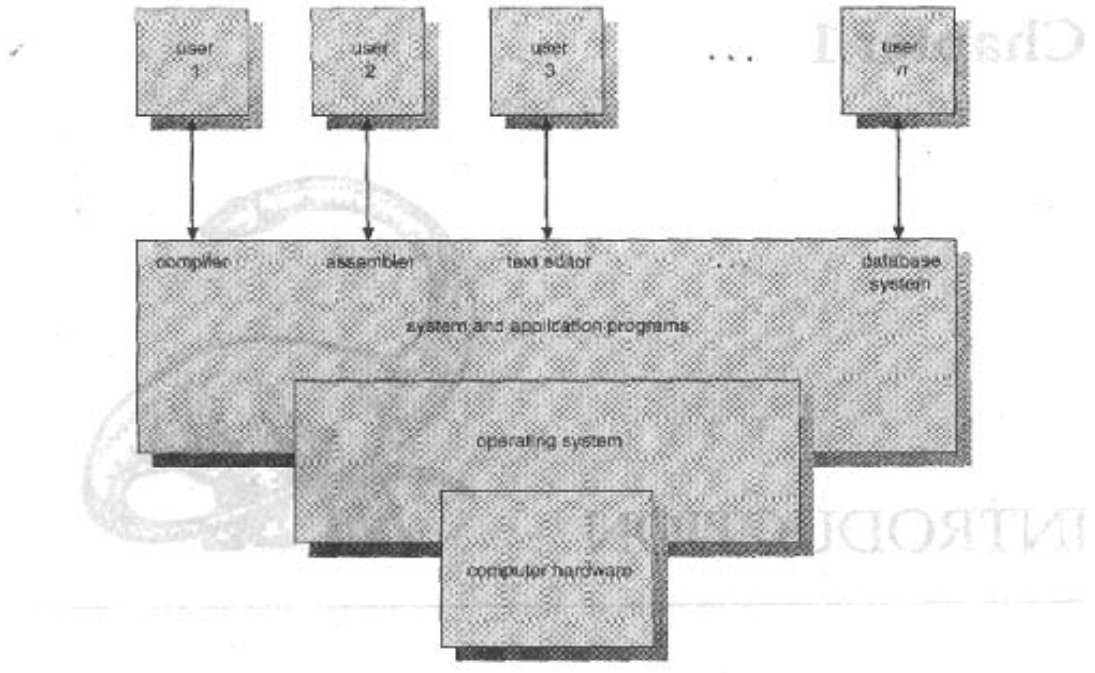


Figure 1.1 Abstract view of the components of a computer system.

Figure 1.1 Abstract view of the components of a computer system.

中央处理单元（CPU）、**存储器**和**输入输出（I/O）设备**等硬件提供了基本的计算资源。字处理软件、电子制表软件、编译器和网页浏览器等应用程序定义了利用这些资源来解决用

户计算问题的方法。操作系统为用户在各种应用程序之间控制和协调着对硬件的使用。

计算机系统由硬件、软件和数据组成。在计算机系统的运行中，操作系统提供了利用这些资源的合理途径。操作系统与政府十分相似。像一个政府，其本身并不能做什么。操作系统仅仅提供了一个环境，其它程序可以在此做有用的工作。我们可以从两个视角来研究操作系统：用户视角和系统视角。

1.1.1 用户视角

计算机的用户视角随应用接口的不同而变化不一。大多数计算机用户坐在一台由显示器、键盘、鼠标和系统单元组成的个人计算机前面。这样的系统针对独占资源的单用户，以求最优化用户的工作（或娱乐）。这样，操作系统的主要设计目标是易用性，同时也兼顾了性能，而没有太多考虑资源利用率。对用户来说，性能是至关重要的，但是他们并不介意系统是不是频繁处于空闲状态等待缓慢的 I/O 处理。

有些用户使用连接到**大型计算机**或**小型计算机**的终端。有些用户通过终端访问同一台计算机，他们共享资源并且可以进行信息交流。这种操作系统的设计目标是最优化**资源利用**，以保证对现有的所有 CPU 时间、存储器和 I/O 设备的高效利用，使每个用户都公平地分享资源。

有些用户使用**工作站**，并连接到其它工作站或服务器的网络。这些用户在他们的工作中互相提供资源，也共享资源，例如：网络和服务器（文件、计算和打印服务器）。所以这种操作系统的设计是为了协调个人的可用性和资源的利用率。

最近，各种各样的手持计算机流行起来。这些设备大多独立运行，一般由个人用户独自使用。有些通过有线（这种更为普遍）或无线的方式与网络连接。受到电源容量和操作界面的限制，他们仅能执行相关的远程操作。这种操作系统主要面向个人应用，节能也很重要。

有些计算机没有或者仅有很小的用户视角。例如：家用设备和汽车上的嵌入式计算机可能仅仅有一个数字键盘和能够用开和关来表示状态的指示灯。但是，它们和它们的操作系统的运行大多不需要用户介入。

1.1.2 系统视角

从计算机视角看来，操作系统是与硬件最为密切的程序。我们可以把操作系统视为**资源分配程序**。一个计算机系统有许多资源（硬件和软件，系统工作时可能会需要它们）：CPU 时间、内存空间、文件存储空间、I/O 设备等等。操作系统扮演着这些资源的管理者的角色。面对为数众多而且可能相互冲突的资源请求，操作系统必须决定如何为特定的程序和用户分配资源，力求高效、公平。

操作系统视角与用户视角的一个轻微的不同点在于它强调了对各种 I/O 设备和用户程序的控制需求。操作系统是一种控制程序。**控制程序**管理用户程序的运行，以防止发生错误和对计算机的不合理利用。它尤其关注对 I/O 设备的操作和控制。

然而，通常我们没有对操作系统做完全充分的定义。操作系统之所以存在是因为它们是构造便于使用的计算环境的合理途径。计算机系统的基本目标是执行用户程序和简化用户问题。基于这个目标构建了计算机硬件。因为纯粹的硬件并不易于使用，于是开发了应用程序。这些应用程序需要一些公共的操作，比如控制 I/O 设备。于是控制和分配资源的公共功能就被整合到了一个软件中，这就是操作系统。

另外，对于操作系统应该包含哪些部分，我们还没有达成共识。一个简单的观点认为当你向商家订购“操作系统”时就包含了一切。存储器(内存、磁盘和磁带)的需求和特性包含其中，然而随着系统的不同而差异巨大。(系统的存储容量以吉为单位度量。(1 千字节或 1KB 相当于 1,024 字节，1 兆字节或 1MB 相当于 $1,024^2$ 字节，1 吉字节或 1GB 相当于 $1,024^3$ 字节，

但是计算机制造商往往简单的称 1 兆为 1 百万字节，吉为 10 亿字节。)）有些系统拥有不足一兆的存储空间，甚至缺少一个全屏幕的编辑器，相反，有些则需要数百兆的存储空间，并且基于图形视窗系统。一个更为普遍的定义是：操作系统是一个自始至终在计算机中运行的程序（通常称之为**内核**），其它程序则属于应用程序。操作系统应该由哪些部件组成，变得越来越重要了。1998 年，美国司法部起诉了微软，状告微软在她的操作系统中集成了太多应用程序，阻碍了应用程序开发商与她的竞争。

1.1.3 系统目标

通过说明它做什么比它是什么更容易定义操作系统，但是这样也十分很困难。有些操作系统的首要目标是用户的易用性。操作系统存在是因为它们被假定为较容易的计算手段。当你看待个人计算机时，这种观点就会异常清晰。

其它操作系统的首要目标是计算机系统的高效性。这是对大型、共享、多用户的系统而言的。这些系统非常昂贵，所以希望它们尽可能的高效。易用性和高效性这两个目标有时相互冲突。过去，效率要比易用性更为重要（1.2.1 节）。因此许多操作系统理论更为注重对计算资源的优化使用。随着时间的推移，操作系统也在发展，例如 UNIX 开始时以键盘和打印机作为它的用户接口，限制了用户的易用性。随着时间的过去，硬件变了，UNIX 也移植到了新硬件上并提供了更加友好的界面。许多**图形用户界面（GUI）**诞生了，这允许 UNIX 在保持其高效性的前提下为用户提供更佳的易用性。

操作系统的设计是相当复杂的工作。设计者要在设计和实现的过程中面对众多的利弊权衡，许多人不仅仅满足于实现，他们还不断的修订和更新操作系统。对于某个操作系统是否达到了其设计目标的讨论是公开的，不同用户各持己见。

要领会操作系统是什么和做什么，首先让我们考虑一下它在过去的 45 年中是如何发展的。通过回顾这个发展，我们可以明确操作系统的共同点，并理解这些系统如何和为什么会有这样的经历。

操作系统和计算机体系结构在很大程度上相互影响。为了推动对计算机硬件的利用，研究人员开发了操作系统。而后操作系统用户提议改变硬件的设计以求简化他们的工作。在这个简短的历史回顾中，要注意到为了解决操作系统的问题而导致了一些新硬件的引入。

1.2 大型计算机系统

大型计算机系统是最早在商业和科学领域应用的计算机系统。在这一节里，我们将从简单的**批处理系统**到**分时系统**来追寻大型计算机系统的成长过程(批处理系统运行一个且仅仅一个应用程序；分时系统允许用户与计算机系统交互)。

1.2.1 批处理系统

早期的计算机是通过控制台运行的体积巨大的机器。通用的输入设备是读卡器和磁带驱动器。输出设备是行式打印机、磁带驱动器和卡片穿孔机。用户不直接与计算机系统交互，而是准备好一个作业（由程序、数据和一些控制信息（控制卡片）组成）并提交给计算机操作员。作业一般以穿孔卡片的形式提交。一段时间之后（数分钟、数小时或数天之后）结果就会出来。输出由程序结果和用于调试的最后的内存和寄存器内容组成。

这些早期计算机中的操作系统相当的简单。它的主要任务是自动的一个一个的传送作业。操作系统总是驻留在内存中（图 1.2）。

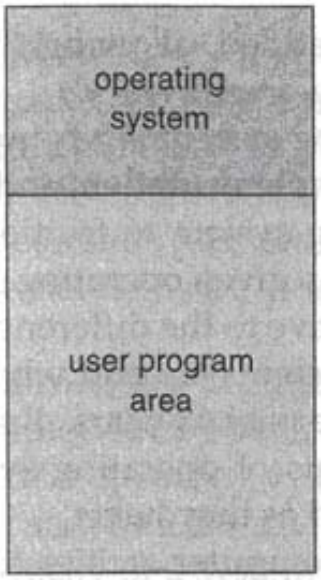


Figure 1.2 Memory layout for a simple batch system.

Figure 1.2 Memory layout for a simple batch system.

为了提高处理速度，操作员根据作业需求的相似性将它们**分批**，然后以组为序来运行。这样，程序员只需要把程序留给操作员。操作员根据程序的需求将它们分批，当计算机可用时，运行每一批程序。作业的执行结果将返回给相应的程序员。

因为机械 I/O 设备的速度本质上就要比电子设备慢，所以在这种运行环境中 CPU 经常处于等待状态。甚至一个慢速的 CPU 也要以微秒级的速度运行，每秒可以执行数千个指令。另一方面，即使快速读卡机也只能每分钟读 1200 张卡（每秒钟 20 张）。这样，CPU 和 I/O 设备的速度大约就相差三个数量级或者更多。随着时间的推移，技术的进步和磁盘的引入造就了更快速的 I/O 设备。然而 CPU 速度的提高远为迅速，所以问题不但没有解决，反而加剧了。

磁盘的引入允许操作系统将所有的作业保留在一张磁盘上，这优于串行读卡器。由于直接读取多个作业，操作系统可以实现**作业调度**，能够高效的利用资源和执行任务。我们讨论了作业和 CPU 调度的几个重要方面；在第六章中将有详细的讨论。

1.2.2 多道程序系统

作业调度最重要的方面就是多道程序设计。一般而言，单个用户不能总是保持 CPU 或 I/O 设备忙碌。通过组织多个作业，**多道程序设计**提高了 CPU 的利用率。如此，CPU 总是有一个作业可以执行。

其思想是：操作系统在内存中同时保留多个作业（图 1.3）。因为可同时保存在内存中的作业数要远小于作业池中的作业数，所以内存中的作业集是作业池中的作业集的一个子集。操作系统从内存中挑选一个作业并执行。最终，作业可能必须等待一些工作（比如说 I/O 操作）事先完成。而在非多道程序系统中，CPU 将处于等待状态。在多道程序系统中，CPU 简单的转向执行另外一个作业。当一个作业需要等待时，CPU 就执行另外一个，以此类推。最终，第一个作业结束等待并重新获得 CPU。既然至少有一个作业需要执行，那么 CPU 就不会空闲下来。

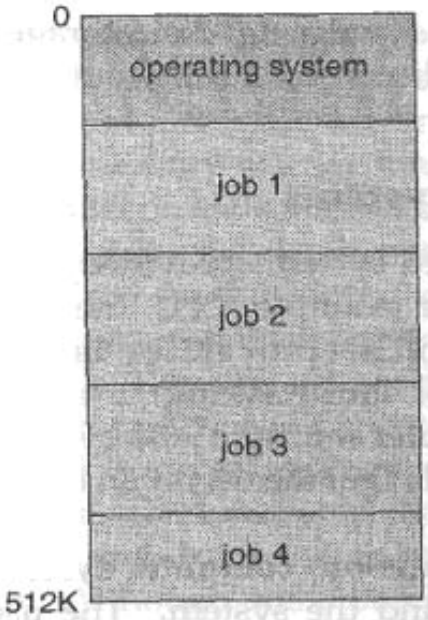


Figure 1.3 Memory layout for a multiprogramming system

Figure 1.3 Memory layout for a multiprogramming system.

这种思想在日常生活中是很普遍的。一个律师每次不会只为一个委托人工作。当一个案子等待审判或打印文件时，律师可以为另一个案子工作。如果有足够的委托人，就不会因为缺乏工作而无事可做。（空闲的律师往往会成为政客，因为那儿有确定的社会价值保持律师整天忙碌。）

多道程序设计是操作系统必须为用户做出选择的第一个实例。多道程序操作系统也因此有相当大的改进。所有作业在进入系统之后保存在作业池中。作业池中包含了所有存储在磁盘中等待分配内存的进程。如果几个作业准备完毕要被放到内存中，但是又没有足够的空间全部容纳，这时系统必须要从中选择。作业调度做出这个选择，这将在第六章中进行讨论。操作系统从作业池中选择一个作业后就将这个作业调入内存执行。因为同时有多个程序在内存中，所以就需要某种形式的内存管理，第九章和第十章中涉及到这些内容。另外，如果多个作业同时准备好，系统就必须从中做出选择。这是 CPU 调度的任务，将在第六章中讨论。最后，多个作业的并发运行只能够影响到操作系统的某些部分，包括进程调度、磁盘存储器和内存管理等。对这些事项的讨论贯穿全书。

1.2.3 分时系统

多道程序系统和批处理系统提供了一个可以高效利用系统资源（如 CPU、存储器和外围设备）的环境，但是没有为用户提供与计算机系统交互的能力。**分时**技术（或者说是**多任务处理**）是对多道程序设计的逻辑扩展。通过在多个作业间转换 CPU，可以同时执行多个作业，但是这种转换发生的如此频繁以至于用户在程序运行的同时可以与计算机交互。

交互式（或 hands-on）计算机系统为用户提供了与计算机直接通信的能力。通过键盘或鼠标，用户直接向操作系统或程序发出指令，然后等待会立即出现的结果。因此，**响应时间**应该较短——典型的大约在一秒钟之内。

分时操作系统允许多个用户同时共享计算机。因为分时系统中的每个操作或命令往往都

非常短，所以每个用户仅仅需要一点 CPU 时间。系统快速的从一个用户转向另外一个，每个用户却感觉到自己独占了整个计算机系统，即便此时是由多个用户共享。

分时操作系统利用 CPU 调度和多道程序设计为每个用户提供了分时系统资源的一小部分。每个用户至少有一个独立的程序在内存中。程序在装入内存并执行后通常被视为**进程**。当一个程序正在运行时，在它结束或需要进行 I/O 操作之前往往只会执行小段时间。输入输出可能是交互式的；换句话说，输出要显示给用户，输入则来自用户的键盘、鼠标或其它设备。因为交互式的输入输出典型的运行在一个“人的速度”上，它可能需要很长的时间才能完成。例如，输入可能会受限于用户的打字速度；对人们来说，8 个字符每秒是很快的，但对计算机来说却相当缓慢。当交互式输入发生时，与其让 CPU 闲置还不如让操作系统快速的将 CPU 转向其他用户的程序。

分时操作系统甚至比多道程序操作系统更加复杂。二者都必须在内存中同时保留多个作业，所以系统必须要有内存管理和保护（第九章）。为了获得一个合理的响应时间，作业可能需要在主存和磁盘（作为主存的后备存储器）之间换进换出。为达到这个目标，一个通用的方法是虚拟内存，这是一种允许将作业部分内容装入内存就可以运行的技术（第十章）。虚拟内存策略的主要优点是程序可以比**物理内存**更大。更进一步讲，它将主存抽象到一个大的统一的存储队列中，从而分开来看待逻辑内存和物理内存。这种方案使程序员免受内存容量的限制。

分时系统也必须提供一个文件系统（第十一章和第十二章）。文件系统驻留在磁盘中；因此，必须要提供磁盘管理（第十四章）。而且分时系统提供了一个并行运行机制，这需要完善的 CPU 调度策略（第六章）。为了保证作业有序运行，系统必须提供作业的同步和通信机制（第七章），并且要确保作业不会在死锁发生时困住而永远处于相互等待状态（第八章）。

早在 1960 年就论证了分时思想，但那时构建分时系统不但困难而且昂贵，这种状况一直持续到 70 年代早期。虽然批处理系统依然存在,但今天的大多数系统是分时系统。因此多道程序设计和分时技术是现代操作系统的主题，也是本书的主题。

1.3 桌面系统

个人计算机在 70 年代出现。在最初的十年中，个人计算机中的 CPU 缺少保护操作系统不受用户程序损伤的特性。个人计算机操作系统因此而不支持多用户和多任务处理。然而随着时间的推移，这种操作系统的目标有所变化；取代对 CPU 和外围设备的最优化利用，用户的易用性和响应速度成为系统的主要设计目标。这些系统包括运行 Microsoft Windows 和 Apple Macintosh 的个人计算机。微软的 MS-DOS 操作系统被同样来自微软的多个产品取代，而 IBM 则使用 OS/2 多任务处理系统替代了 MS-DOS。Apple Macintosh 操作系统引入了更多的先进硬件，并且包含了新的特征，例如：虚拟内存和多任务处理。随着 MacOS X 的发布，现代操作系统的内核为了可测量性、性能和特性而基于 Mach 和 FreeBSD UNIX，但也保留了丰富多彩的图形用户界面。Linux 是一个用于个人计算机的类 UNIX 操作系统，最近也十分流行。

通过借鉴大型计算机操作系统的开发成果，个人计算机操作系统受益匪浅。微型计算机可以很快采用为更大型的操作系统研发的一些技术。另一方面，微型计算机的硬件价格已经足够低了，个人可以单独的使用计算机，CPU 的利用率不再是主要问题。因此，大型计算机操作系统中的一些设计可能不适用于更小型的系统。

其它的一些设计依然应用，例如：文件保护。最初，文件保护在个人计算机上不是必需的。然而，现在这些计算机经常要与其它的计算机通过局域网或因特网连接在一起。当其他的计算机和其他的用户能够访问一台个人计算机中的文件时，文件保护开始成为操作系统的

一个重要特征。缺少了这种保护，恶意程序就可以轻易的破坏系统中的数据（如 MS-DOS 和 Macintosh）。恶意程序可以自我复制，能够通过蠕虫或病毒机制快速传播，并且可能会破坏公司或全世界的网络。仅仅依靠像内存保护和文件许可这样的高级分时特征并不能令人满意。近来频繁出现的安全漏洞再次证明了这一点。这些内容将在第十八章和第十九章中展开讨论。

1.4 多处理机系统

目前为止，大多数系统是单处理机系统；也就是说，他们仅仅拥有一个主 CPU。然而**多处理机系统**（通常称之为**并行系统**或**紧密耦合系统**）的成长是非常重要的。这样的系统拥有多个密切通讯的处理器，这些处理器共享计算机总线、时钟和外围处理机，有时也共享内存。

多处理机系统有以下三个主要优点：

1. **提高了系统的吞吐量。**通过增加处理器的数目，我们希望能够在更短的时间内完成更多的工作。N 个处理器的加速比不是 N；而是要小于 N。当多个处理器协同处理一个任务时，保证所有部分正确工作会导致一个固定的开销。这个开销，再加上对共享资源的竞争，降低了增加处理器所期望获得的性能提高。相似的，N 个程序员协同工作不会产生 N 倍的工作效率。
2. **节省投资。**多处理机系统要比多个单处理机系统节省费用，因为它们可以共享外围处理机、存储器和电源。如果几个程序对相同的数据进行操作，将数据存储在磁盘并由所有的处理器共享要比使用多台计算机利用本地磁盘储存多个拷贝便宜。
3. **提高了系统的可靠性。**如果能够将工作适当地跟配给多个处理器，那么当一个处理器出现故障时就不会导致整个系统崩溃。如果我们有十个处理器，当一个出现故障时剩下的九个处理器必须要得到有故障处理器所负责的工作的一部分。这样，整个系统的运行速度仅仅会降低 10%，而不会全盘瘫痪。这种持续提供服务的能力被称为**故障弱化**，它与所剩硬件占整个系统的比例有关。这种设计也被称为**容错**。

故障发生时的延续运行需要一个允许故障探测、分析和纠错（在可能的情况下）的机制。Tandem 系统通过使用硬件和软件副本以确保在发生故障时可以继续运行。系统拥有两个同样的处理器，每个都有自己的本机存储器。它们通过总线连接，一个处理器为主处理器，另一个作为后备。每个进程留有两个拷贝：一个在主处理器中，另一个在后备处理器中。在系统执行的确定的检查点上，主处理器中每个作业的状态信息（包括内存映象的拷贝）被拷贝到备份处理器中。如果探测到一个故障，就激活备份并从最近的检查点重新开始。因为这种方案包含了相当可观的硬件副本，所以代价高昂。

今天的大多数通用多处理机系统采用了**对称多处理（SMP）**技术，每个处理器运行一个同样的操作系统拷贝，而且这些拷贝在需要时相互通讯。有些系统采用了**非对称多处理**，每个处理器有着明确的任务。一个主处理器控制着系统；其它的处理器照应主处理器或者有预定义的任务。这种方案定义了一种主从关系。主处理器调度从处理器并为其分配工作。

SMP 意味着同等对待所有的处理器；处理器之间没有主从关系。每个处理器并行的运行一份操作系统拷贝。图 1.4 例举了一种典型的 SMP 体系结构。SMP 系统的一个例子是为 Multimax 计算机设计的 UNIX 的 Encore 版。这种计算机可以使用数十个处理器，它们都运行 UNIX 的拷贝。这种模型的优点是可以同时运行多个进程，有 N 个处理器就可以运行 N 个进程，而不会引起严重的性能恶化。然而，我们必须小心控制输入输出，以保证处理器对数据的正确接收。而且，因为 CPU 是独立的，就可能当一个疲于应付时另一个却无事可做，导致系统的低效。如果处理器共享确定的数据结构，那么就可以避免这种低效。这种形式的多处理机系统将允许多个处理器共享进程和资源（如内存），并且能够降低处理器之间的差异。

开发这种系统必须要十分小心，这就像我们将在第七章中看到的那样。实际上所有的现代操作系统（包括 Windows NT、 Solaris、 Digital UNIX、OS/2 和 Linux）都提供了对 SMP 的支持。

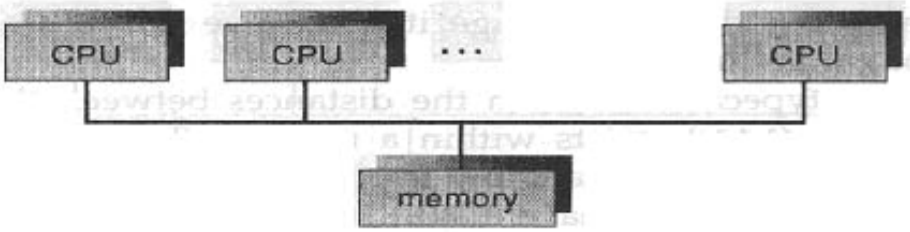


Figure 1.4 Symmetric multiprocessing architecture.

Figure 1.4 Symmetric multiprocessing architecture.

对称多处理和非对称多处理的不同可能与硬件或软件有关。专用硬件能够区分多个处理器，或者可以开发软件以允许一个主处理器和多个从处理器。例如，Sun 的操作系统 SunOS 第 4 版提供了对非对称多处理的支持，而基于同样硬件的第 5 版（Solaris 2）是对称多处理系统。

因为微处理器变得更便宜更强大，从处理器（**后端**处理器）中去掉了附加的操作系统功能。例如，添加一个处理器并为其配备自己的存储器以控制磁盘系统是相当容易的。这个微处理器可以从主 CPU 接收请求队列并实现自己的磁盘队列和调度算法。这种方案减轻了主 CPU 的磁盘调度开销。个人计算机在键盘中包含了一个微处理器来将击键转换成编码并发送给 CPU。事实上，微处理器的这种应用变得如此广泛，以至于我们不再考虑多处理技术了。

1.5 分布式系统

以一种最简单的术语来讲，**网络**就是两个或多个系统通讯的路径。分布式系统依靠网络来实现其功能。通过通信，分布式系统能够共享计算任务，并且为用户提供了丰富的特性。

不同的网络采用的协议、节点间距离和传输介质各不相同。虽然 ATM 和其它的一些协议应用广泛，但是 TCP/IP 是最通用的网络协议。同样，操作系统对协议的支持也不尽相同。包括 Windows 和 UNIX 在内的大多数操作系统支持 TCP/IP 协议。有些系统为了适应自己的需求而支持专有的协议。对一个操作系统来说，网络协议仅仅需要一个简单的接口设备——网络适配器，如利用一个设备驱动程序对它进行管理，在通信协议中利用软件将数据打包发送和解包接收。对这些概念的讨论贯穿全书。

网络的分类根据它们的节点间的距离。**局域网（LAN）**在一个房间、一个楼层或者是一个建筑物中。**广域网（WAN）**通常在建筑物、城市或国家间。一个跨国公司可以有一个 WAN 来连接世界范围内的办公室。这些网络可以运行一个或多个协议。新技术的不断涌现带来了新的网络形式。例如，**城域网（MAN）**可以连接一个城市的建筑。蓝牙设备在几步远的短距离内进行通信，这在本质上就构建了一个**小范围网络**。

网络介质同样各种各样。这包括铜线、光纤和在卫星、微波天线和无线电接收装备之间的无线传输。甚至射程较短的红外线通讯也能够用来构建网络。未来，计算机可以随时随地的使用网络或者建立网络进行通信。根据它们的性能和可靠性，这些网络也不尽相同。

1.5.1 客户端／服务器系统

因为个人计算机变得更快、更强，并且更便宜，设计者已不再将精力集中在系统结构上。现在，个人计算机取代了连接到中心计算机的终端。相应的，原先由中心计算机直接控制的用户交互功能现在也越来越多的交给 PC 来处理。结果，中央计算机现在作为一个**服务系统**来满足**客户系统**发起的请求。图 1.5 描述了一个客户端 / 服务器系统的通用结构。

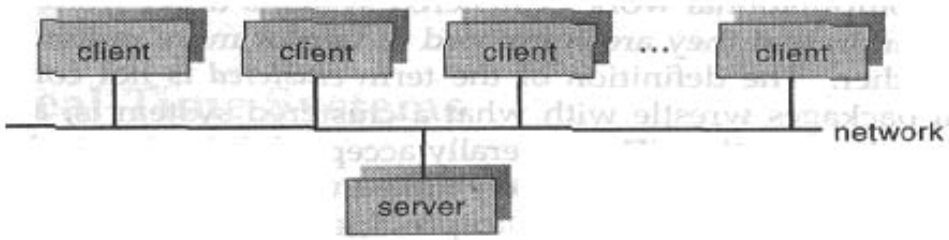


Figure 1.5 General structure of a client-server system.

Figure 1.5 General structure of a client-server system.

服务器系统可以概括性的分为计算服务器和文件服务器两类。

• **计算服务系统**提供了一个客户端接口，客户端通过此接口发送要执行的操作，计算服务器执行这个操作并将结果返回给客户端。

• **文件服务系统**提供了一个文件系统接口，客户端可以在此创建、更新、读取和删除文件。

1.5.2 对等网络系统

计算机网络的成长，尤其是因特网和环球网（WWW），对最近的操作系统的发展产生了深远的影响。当个人计算机在 70 年代引入时，它们的设计是为了“个人”应用并且通常作为独立的计算机。随着在 80 年代开始的电子邮件、ftp 和 gopher 等因特网公共应用的普及，许多个人计算机连接到了计算机网络上。随着 90 年代中期互联网的出现，网络连接成为计算机系统的必要组成部分。

实际上所有的现代个人计算机和工作站都具有运行一个网络浏览器以访问 Web 上的**超文本**文档的能力。目前的操作系统（如 Windows、OS/2、MacOS 和 UNIX）也包含了系统软件（如 TCP/IP 和 PPP），这些软件可以使计算机通过局域网或电话线访问因特网。这包括了网页浏览器、远程登录和文件传输客户端和服务器。

与在 1.4 节讨论的紧密耦合系统对比，在这些应用中使用的计算机网络由处理器集合组成，它们不共享内存或时钟，而是每个处理器有它自己的内存。处理机之间通过各种各样的通信线路进行通信，比如：高速总线或电话线。这些系统通常被称为**松散耦合系统**（或**分布式系统**）。

一些操作系统具备了网络和分布式系统的概念，而远不止提供网络连接。**网络操作系统**是一种操作系统，它提供了像跨越网络的文件共享这样的特征，并且包含了一个允许不同计算机间的不同进程相互交换信息的通讯方案。虽然一台运行网络操作系统的计算机注意到网络的存在并且能够与其它联网的计算机通信，但它却独立于网络上其它的所有的计算机。分布式操作系统具有更低的独立性：不同的操作系统密切联系，以至于让人感觉只有一个独立的操作系统在控制网络。我们将在第十五章和第十七章中讨论计算机网络和分布式操作系统。

1.6 集群系统

与并行系统相似，集群系统也集中了多个 CPU 来完成计算工作。与并行系统不同，集群系统由两个或更多独立的计算机连接在一起组成。对术语“集群”的定义并不具体；对于什么是集群系统和为什么一种优于另一种，多个商业包互相竞争。通常公认的定义是集群计算机共享存取器，并且通过局域网紧密连接。

通常集群系统实现了对**高可用性**的支持。集群软件层运行在众多节点之上。每个节点能够（通过局域网）监控一个或多个其它节点。如果被监控的机器出现故障，那么监控的机器能够获取被监控机器的存储器，并重新开始运行被监控机器中所运行的应用程序。有故障的机器可以维持在这种故障状态，但是用户和客户端应用程序仅仅会感受到一个简短的服务中断。

在**非对称集群**中，当其它的计算机运行应用程序时有一台计算机处于**开机备用模式**。备用主机除了监控其它活动服务器外什么也不做。如果这个服务器出现故障，备用机就会成为活动服务器。在**对称模式中**，两个或更多的主机同时运行应用程序并且相互监控。因为利用了所有可利用的硬件，这种模式显然更加高效。它就需要多个可以运行的应用程序。

其它形式的集群包括并行集群和构建在广域网上的集群。并行集群允许多个主机通过共享存储器访问数据。因为大多数操作系统缺乏对多个主机同时访问数据的支持，所以并行集群通常由软件和应用程序的特别版本实现。Oracle 并行服务器是 Oracle 数据库的一个用于运行在并行集群系统上的版本。每个机器有完全访问数据库中所有数据的权力。

尽管分布式系统已经取得了长足的进步，大多数系统并没有提供多方面的分布式文件系统。因此，大多数集群系统不允许共享访问磁盘上的数据。为此，分布式文件系统必须提供对文件的访问控制和锁定，以避免互相矛盾的操作。这种类型的服务通常被认为是**分布式锁定管理（DLM）**。针对分布式通用文件系统的研发工作正在进行，像 SUN 这样的开发商已经宣布了在其操作系统中包含 DLM。

集群技术发展迅速，其中包括了全球集群，在这种系统中，机器可以遍布全世界（或者是 WAN 所能到达的任何地方）。这样的工程依然是研究和发展的主题。

随着**存储区域网络**（SAN，在 14.6.3 节中详细描述）变得更为流行，集群系统的用途和特性将有更大程度的扩展。SAN 可以很容易的在多个主机和多个存储器单元之间建立连接。由于连接的复杂性，当前的集群系统通常限制 2 到 4 台主机共享数据。

1.7 实时系统

另外一种具有特殊用途的操作系统是**实时系统**。如果对运算或数据流有严格的时间要求的话，就需要使用实时系统；因此它经常作为控制设备出现在专门的应用中。传感器获取数据并发送给计算机。计算机必须分析数据并在需要的情况下调整控制。实时系统在科学实验控制、医学成像、工业控制和某些显示系统中应用广泛。有些汽车引擎燃油喷射控制系统、家用控制器和武器系统也是实时系统。

实时系统具有明确定义的、不变的时间约束。处理过程必须在规定的时间内完成，否则系统就失效了。例如，实时系统必须在一个用于汽车制造的机器人手臂撞到(正在制造的)汽车之前停止它。一个实时系统只有在规定的时间内返回正确的结果才算是能够正确工作。实时系统的这个要求跟分时系统和批处理系统不同，在这点上注意跟分时系统和批处理系统对比，我们期望（不是强制的）分时系统能够快速响应，而对批处理系统则根本就没有时间限制。

实时系统分为两种：硬实时系统和软实时系统。**硬实时系统**要保证按时完成关键性的任务。为此，需要限制系统中所有的延迟，从数据检索到操作系统结束请求模式的时间需求。这样的时间约束限定了硬实时系统的功能。辅助存储器通常十分有限或者没有使用，数据存

储在短期存储器或只读存储器（ROM）中。ROM 是一种非易失性的存储设备，即使在掉电情况下它也可以保存数据；大多数其它类型的存储器是易失性的。大多数的高级操作系统特性往往将用户和硬件分离，这样就会造成许多不确定的时间需求，因此它们也不会出现在硬实时系统中。例如，在实时系统中几乎见不到虚拟内存技术（第十章）。因此硬实时系统与分时系统是相互冲突的，并且二者不能混为一体。由于现有的操作系统都没有提供对硬实时的支持，所以在本书中我们就没有涉及此方面内容。

软实时系统是一种限制较少的实时系统，其关键任务的优先权要高于其它任务，并保持拥有这个优先权直到结束。在硬实时系统中，实时任务不能无休止的等待系统内核来执行它，因此需要限制操作系统的内核延迟。软实时系统可以与其它类型的系统混为一体，这是可以实现的。然而软实时系统比硬实时系统有着更多受限的应用。介于它们缺乏对操作时限的支持，在工业控制和机器人中应用软实时系统是比较危险的。然而软实时系统通常在其它几个领域中应用广泛，这包括了多媒体、虚拟现实和高级科学项目（如海底探险和行星探测）。这些系统需要一些硬实时系统无法支持的高级操作系统特性。由于软实时系统应用的不断扩展，当前的大多数操作系统都包含了该技术，其中包括 UNIX 的主要版本。

我们将在第六章中涉及到在操作系统中实现软实时功能所需的调度机制；在第十章中描述软实时计算中的内存管理。最后，在第二十一章中描述 Windows 2000 操作系统的实时部分。

1.8 手持系统

手持系统，包括像 PalmPilot 这样的**个人数字助理（PDA）**或连接到某种网络（如因特网）的蜂窝电话。手持系统和应用程序的开发者要面对许多挑战，其中大多来自于这种设备的体积限制。比如，典型的 PDA 大约长 5 英寸，宽 3 英寸，重量不足 0.5 磅。由于受到尺寸的限制，大多数手持系统包含了一个小容量存储器、一个低速处理器和一个小尺寸显示屏。我们将逐一讨论这些限制。

许多手持设备的内存容量在 512KB 到 8MB 之间。（与此相比，典型的个人计算机或工作站可能拥有几百兆内存！）这样，操作系统和应用程序必须能够高效的管理内存，这包括在不再使用内存时将所持有的内存归还给内存管理程序。我们将在第十章中研究虚拟内存，这是一种允许开发者的程序好像运行在比物理内存更多的系统之上的技术。目前，多数手持设备没有使用虚拟内存技术，这就迫使开发者受到物理内存容量的限制。

手持系统开发者所关注另外一个的问题是处理器速度。大多数手持设备的处理器运行速度通常为个人计算机处理器运行速度的几分之一。更快的处理器需要更强劲的电源。在手持设备中使用更快的处理器就需要配备更大容量的电池，而且需要更频繁的更换电池（或者是充电）。为了尽可能减小手持设备的尺寸，经常使用体积更小、速度更慢、能耗更低的处理器。为此，操作系统和应用程序就必须减轻处理器的负担。

手持设备程序的开发者所面临的最后一个问题是要面对所使用的小尺寸显示屏。家用计算机显示器的尺寸可能会达到 21 英寸，而手持系统的显示屏通常不足 3 英寸。像阅读电子邮件和浏览网页这样的常见的工作必须要精简到更小的显示屏中。显示网页内容的一种方法是**网页裁剪**技术，就是只将网页中的一小部分内容传送并显示在手持设备上。

一些手持设备可以使用像 BlueTooth（1.5 节）这样的无线技术，允许远程访问电子邮件和浏览网页。与 Internet 连接的蜂窝电话也是这种类型。当前很多 PDA 没有提供无线访问的支持。为了向这种设备中下载数据，往往首先将数据下载到个人计算机或工作站，然后再传到 PDA 中。有些 PDA 允许通过红外线相互直接传输数据。通常,PDA 的这些功能上的限制根据手持设备的易用性和便携性来折衷。随着网络连接越来越广泛，手持系统的应用也日

渐广泛，照相机和 MP3 等一些其它的设备不断发展也扩大了它们的应用。

1.9 特征迁移

总的来说，对大型计算机和微型计算机操作系统的比较显示：微型计算机已采用了过去大型计算机所独有的特性。同样的概念适用于不同类型的计算机：大型计算机、小型计算机，微型计算机和手持计算机。图 1.6 所描述的许多概念将在稍后介绍。然而，要开始理解现代操作系统，你需要了解特征迁移，并且认识到许多操作系统特征的发展史。

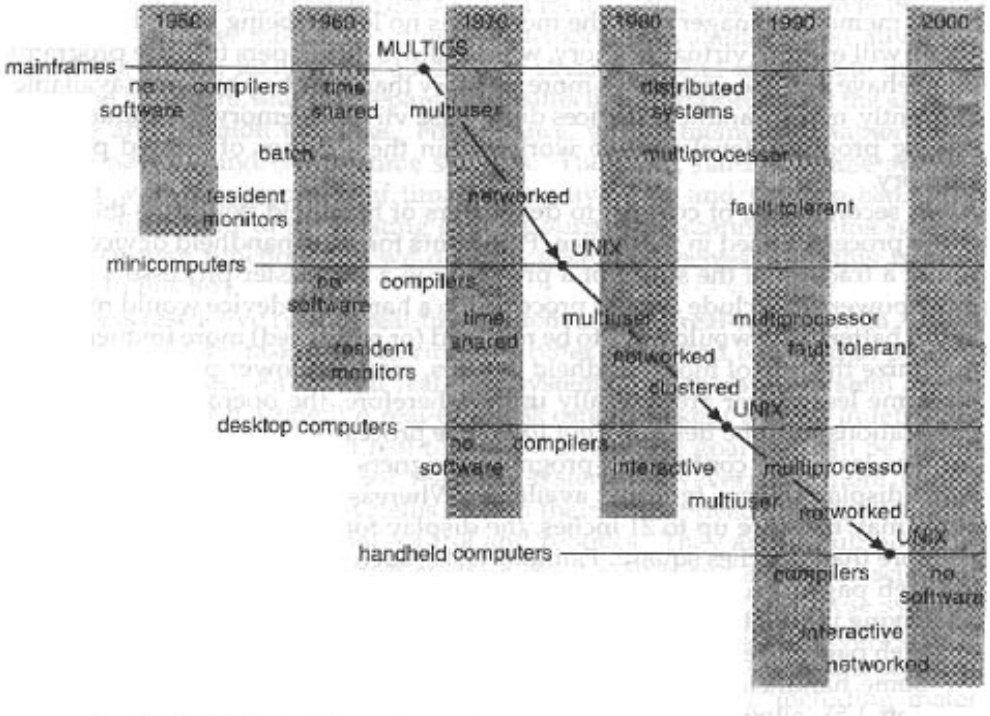


Figure 1.6 Migration of operating-system concepts and features.

Figure 1.6 Migration of operating-system concepts and features.

一个典型的特征迁移的例子是多路复用信息与计算服务（MULTICS）操作系统。MULTICS 是 MIT（Massachusetts Institute of Technology）在 1965 年到 1970 年间开发的一个计算工具。它运行在一个大型的复杂的大型计算机（GE 645）上。许多为 MULTICS 开发的技术后来被贝尔实验室（一个开发 MUTICS 的合作伙伴）用在了 UNIX 的设计上。UNIX 操作系统大约在 1970 年为 PDP-11 小型计算机研制。大约在 1980 年，UNIX 的许多特征成了用在微型计算机中的类 UNIX 操作系统的基础，这包括近来的许多操作系统，比如：Microsoft Windows NT、IBM OS/2 和 Macintosh。如此，随着时间的推移，为大型计算机系统设计的许多技术移植到了微型计算机上。

在大型操作系统的特性被按比例缩小以适应个人计算机的同时，更强、更快、更完善的硬件系统也在发展。**个人工作站**是一种大型的个人计算机，比如：Sun SPARCstation、HP/Apollo、IBM RS/600 和运行 Windows NT 或 UNIX 派生系统的 Intel Pentium 系列。许多大学和企业拥有相当数量的工作站，这些工作站又通过局域网紧密连接。当个人计算机拥有更完善的硬件和软件时，大型计算机和微型计算机的分类变得模糊不清了。

1.10 计算环境

从最初的 hands-on 系统到多道程序系统和分时系统，再到个人计算机和手持计算机，我们回顾了操作系统的发展，那么现在就应该能够给出一个综述：这样的系统是怎样在多样化的计算环境中应用的。

1.10.1 传统计算

随着信息处理技术的成熟,传统计算环境的概念变得模糊了。考虑一下“典型的办公环境”。仅仅在数年前，这个环境由连接到提供打印服务网络的个人计算机构成。远程访问并不好用，而膝上型计算机提供了便携的工作环境。在许多公司中，连接到大型机的终端的应用也非常的普遍，它们的远程访问和便携能力还要差一些。

当前的趋势向着提供更多的访问这些环境的能力发展。互联网技术扩展了传统计算。公司实现了通过互联网访问他们内部服务器的**入口**。**网络计算机**是能够解释 Web 计算的必要的终端。便携式计算机能够与 PC 同步，这样就可以轻便的使用公司的数据。它们也能够连接到无线网络以利用公司的互联网入口（也有其它无限的互联网资源）。

大多数用户在家里使用一台独立的计算机，利用慢速调制解调器连接到办公室、因特网或二者都有。网络连接速度曾经非常昂贵，现在就便宜多了，这就允许在公司内部或通过互联网访问更多的数据。这些快速的数据连接允许家用计算机充当 Web 网页服务，并且包含了打印机、客户端 PC 和服务器。有些家庭甚至拥有防火墙来保护这些家庭环境免收侵害。这种防火墙在数年前价值数千美元，而十年前甚至还没有。

1.10.2 基于 Web 的计算

Web 已经变得无处不在，于是人们通过各种各样的设备访问网络，这与我们在过去的几年中所梦想的相比有过之而无不及。个人计算机依旧是最普遍的访问设备，工作站（高档的面向图形的 PC）、手持个人数字助理，甚至蜂窝电话也提供了访问支持。

网络计算提高了网络连接的重要性。先前没有网络连接的设备现在拥有有线或无线访问能力。网络设备现在有了更快的网络连接，这来自于提高网络技术、优化网络实现代码，或二者都有。

基于 Web 运算的实现促使了各种新型设备的出现，如负载均衡器，它在众多类似的服务器之间分配网络连接。像 Windows 95 这样的操作系统扮演着 Web 客户端的脚色，它已经发展到了 Windows ME 和 Windows 2000，Windows 2000 不但可以作为客户端，还可以作为服务器。通常，Web 提高了设备的复杂性，因为用户需要它们能够连接到 Web。

1.10.3 嵌入式计算

嵌入式计算机是现在最为流行的计算机形式，它们运行嵌入式实时操作系统。从汽车引擎和工业机器人到录像机和微波炉,这些设备随处可见。嵌入式计算机往往有着特殊的任务。它们运行的系统通常很简单并缺少一些高级特征，比如虚拟内存，甚至是磁盘。这样，操作系统仅仅提供了有限的特性。它们的用户接口通常很小或者没有，而更多的是在监控和管理硬件设备，比如汽车引擎和机器人手臂。

作为一个例子，考虑前述的防火墙和负载均衡器。有些就是通用计算机，它们运行标准的操作系统（如 UNIX）和特殊用途的应用程序来实现它们的功能。其它的是一些使用特殊用途的嵌入式操作系统的硬件设备，这些设备仅仅提供了所需的功能。

嵌入式系统的应用仍在不断扩展。这些设备，不管是作为独立的单元还是网络或 Web 成员，它们的能力必定会不断提高。整个住宅能够实现计算机化，这样，不管是通用计算机还是嵌入式系统都可以控制供热、照明、报警系统，甚至能够控制咖啡壶。家庭成员可以在回到家之前通过 Web 使室内升温。也许会有一天，电冰箱在牛奶用光时会自动通知食品店。

1.11 摘要

因为两个主要的原因，操作系统在过去的 45 年中持续发展。首先，操作系统尝试调度计算活动，以确保计算系统的优良性能。其次，它为程序的开发和运行提供了一个便利的环境。最初，通过前端控制台使用计算机系统。汇编程序、装载程序、连接程序和编译程序等软件使程序设计更加便利，但是也需要很多配制时间。为了减少配置时间，就要雇佣操作员，对相似的作业打包。

批处理系统允许常驻内存的操作系统对工作自动排序，大大提高了计算机的整体利用率。计算机就不再需要等待人工操作了。然而，对于 CPU 来说，I/O 设备的速度太慢了，所以 CPU 的利用率仍旧很低。脱机操作使得单个 CPU 可以同时控制读卡机、磁带驱动器、打印机等多个低速设备。（Off-line operation of slow devices provides a means to use multiple reader-to-tape and tape-to-printer systems for one CPU.）

为了提高整个计算机的性能，引入了多道程序设计，可以在内存中同时保留多个作业。CPU 在这些作业之间来回转换，减少了执行作业的总时间。

多道程序技术也允许分时。而分时操作系统允许多个用户（从一个到几百个）同时交互式的使用计算机。

个人计算机是微型计算机；它明显比大型机系统更小，更便宜。这种计算机的操作系统在多个方面受益于大型机操作系统的发展。然而，因为个人单独的使用计算机，CPU 利用率不再是首要问题。因此，大型计算机操作系统中的一些设计就不适于这些更小的系统。由于个人计算机现在能够通过网络和 Web 连接到其它的计算机和用户，这样，其它的一些设计则对小型系统和大型系统同样适用，如安全策略。

并行系统拥有多个密切通讯的 CPU；CPU 共享计算机总线和外围处理机，有时也共享内存。这样的系统提高了吞吐量并增强了可靠性。分布式系统允许在地理上分散的主机共享资源。集群系统允许多台机器同时对共享数据进行运算，并且能够在某些子集群成员出现故障的情况下维持计算持续运行。

硬实时系统通常作为控制设备出现。硬实时操作系统具有明确定义的固定的时间限定。进程必须要在限定的时间内完成，否则系统就失效了。软实时系统则具有较少的时间限制，而且不支持限期调度。

最近，出于 Internet 和 WWW 的影响，整合了网页浏览、网络连接和通讯软件的现代操作系统的开发倍受鼓舞。

CPU 的发展需要更高级的功能，通过这条主线我们讨论了操作系统的发展过程。随着硬件价格的进一步降低，计算机的特性也有所改变，这个趋势可以通过现在 PC 的发展感受到。

词汇

并行系统：parallel system
操作系统：operating system
超文本：hypertext
城域网：metropolitan-area network, MAN
存储器：memory
存储区域网络：storage-area network, SAN
大型计算机：mainframe
大型计算机系统：mainframe computer system

对称多处理：symmetric multiprocessing, SMP
对等网络：Peer-to-Peer
对等网络系统：Peer-to-Peer system
多处理机系统：multiprocessor system
多道程序设计：multiprogramming
多路复用信息与计算服务：Multiplexed Information and Computing Service, MULTICS
多任务处理：multitasking
非对称多处理：asymmetric multiprocessing
分布式锁定管理：distributed lock manager, DLM
分布式系统：distributed system
分时：time sharing
分时系统：time-shared system
辅助存储器（二级存储器）：secondary storage
高可用性（高效率）：high availability
个人工作站：personal workstation
个人计算机：personal computer, PC
个人数字助理：personal digital assistant, PDA
工作站：workstation
功能退化，故障弱化：graceful degradation
广域网：wide-area network，WAN
环球网：World Wide Web, WWW
集群系统：clustered system
计算服务系统：compute-server system
交互式计算机系统：interactive computer system
紧密耦合系统：tightly coupled system
进程：process
局域网：local-area networks，LAN
客户端 / 服务器系统：client-server system
控制程序：control program
蓝牙：BlueTooth
内核：kernel
批处理系统：batch system
容错：fault tolerant
入口：portal
软件：software
软实时系统：soft real-time system
实时系统：real-time system
手持系统：handheld system
输入输出设备：input/output (I/O) device
松散耦合系统：loosely coupled system
图形用户界面：graphic user interface, GUI
网络：network
网络操作系统：network operating system
网络计算机：network computer

文件服务系统：file-server system
物理内存（物理存储器）：physical memory
响应时间：response time
小范围网络：small-area network
小型计算机：minicomputer
异步传输模式：Asynchronous Transmission Mode, ATM
因特网：Internet
硬件：hardware
硬实时系统：hard real-time system
中央处理单元:central processing unit, CPU
资源分配程序：resource allocator
资源利用：resource utilization
作业调度：job scheduling
作业调度程序：job scheduler

【未完待续 • 责任编辑：iamxiaohan@hitbbs】

保护模式下 8259A 芯片编程及中断处理探究（上）

Version 0.02

哈尔滨工业大学 并行计算实验室 谢煜波[18]

简介

中断处理是操作系统必须完成的任务，在 IBM PC 中，常用一块中断控制芯片（PIC）——8259A 来辅助 CPU 完成中断管理。在实模式下，中断控制芯片（PIC）8259A 的初始化是由 BIOS 自动完成的，然而在保护模式下却需要我们自行编程初始化。本篇拟从操作系统的编写角度详细描述下笔者在此方向上所做的摸索，并在最后通过 pyos 进行实验验证。此是这部份内容的上篇，将详细描述 8259A 芯片的编程部份，对于操作系统中的中断处理以及程序验证将在下篇里面详细描述。

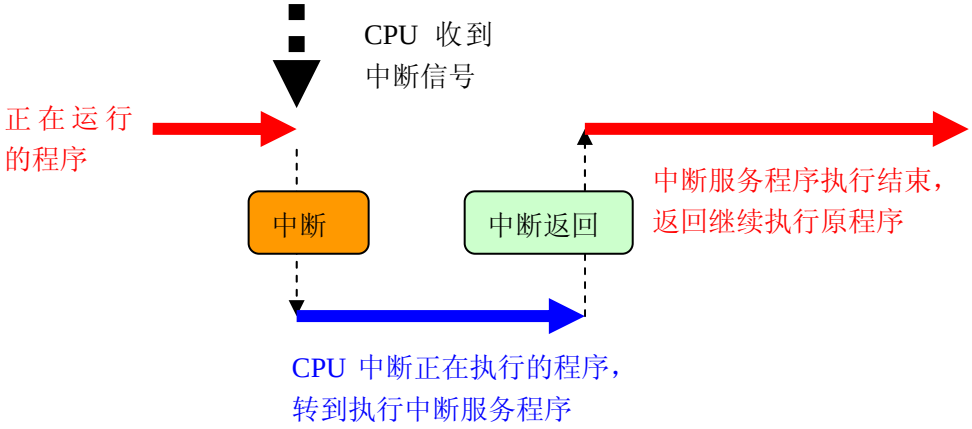
此文只是我在进行操作系统实验过程中的一点心得体会，记下来，避免自己忘记。对于其中可能出现的错误，欢迎你来信指证。

一、中断概述

相信大家对于中断一点都不陌生，这里也不准备详细介绍中断的所有内容，只简单做下概要介绍，这样使对中断没有概念的朋友能建立起一点概念。

计算机除了 CPU 外，还有很多外围设备，然而我们都知道 CPU 的运行速度是很快的，而外围设备的运行速度却不是很快了。假设我们现在需要从磁盘上读入十个字节，而这需要 10 秒钟（很夸张，但这只是一个例子），那么在这 10 秒钟之内，CPU 就无所事事，不得不等待磁盘如蜗牛般的读入这十个字节，如果在这 10 秒钟之内，CPU 转去运行其它的程序，不就可以防止浪费 CPU 的时间了吗？但是这就出现了一个问题，CPU 怎么知道磁盘已经读完数据了呢？实际上，这时磁盘的控制器会向 CPU 发送一个信号，CPU 收到信号之后，就知道磁盘已经读完数据了，于是它就中断正在运行的程序，重新回到原先等待磁盘输入的程序上来继续执行。这只是一个很简单的例子，也只是中断应用的一个很简单的方面，但基本上可以说明问题。可以这么认为：中断就是外部设备或程序同 CPU 通信的一种方式。CPU 在接收到中断信号时，会中断正在运行的程序，转到对中断的处理上，而这个对中断的处理程序常常称为中断服务程序，当中断服务程序处理完中断后，CPU 再返回到原先被中断的程序上继续执行。整个过程如下图所示：

[18] 谢煜波，哈尔滨工业大学计算机科学与技术学院硕士研究生，电子邮件：xieyubo@126.com



（图 1）

中断有很多类型，比如可屏蔽中断（顾名思义，对此种中断，CPU 可以不响应）、不可屏蔽中断；软中断（一般由运行中的程序触发）、硬中断……等很多分类方法。中断可以完成的任务也很多，比如设备准备完毕、设备运行故障、程序运行故障……，这许多突发事件都可以以中断的方式通知 CPU 进行处理。

二、认识中断号及 8259A 芯片

我们都知道计算机可以挂接上许多外部设备，比如键盘、磁盘驱动器、鼠标、声卡……等等一系列设备，而这些设备都可能在同一时刻向 CPU 发出中断信号，那么 CPU 到底应当响应哪一个设备的中断信号呢？这都通过另外一个芯片来控制，在 IBM PC 机中，这个芯片常常被称作：可编程中断控制器（PIC）8259A，说它可编程，是因为我们可以通过编程来改变它的功能。比如我可以通过编程设定 CPU 应当优先响应哪一个中断，屏蔽哪些中断等等一系列事件。

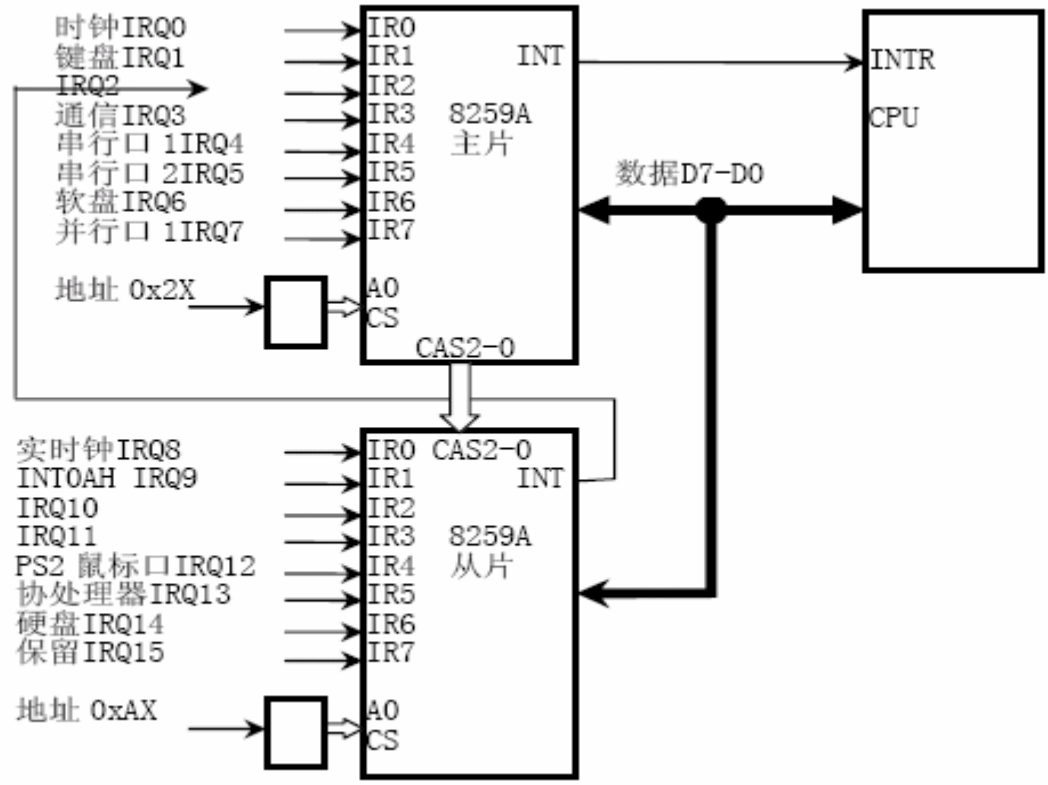
一个 8259A 芯片共有中断请求（IRQ）信号线：IRQ0~IRQ7，共 8 根。在 PC 机中，共有两片 8259A 芯片，通过把它们联结起来使用，就有 IRQ0~IRQ15，共 16 根中断信号线，每个外部设备使用一根或多个外部设备共用一根中断信号线，它们通过 IRQ 发送中断请求，8259A 芯片接到中断请求后就对中断进行优先级选定，然后对多个中断中具有最高优先级的中断进行处理，将其所对应的中断向量送上通往 CPU 的数据线，并通知 CPU 有中断到来。

这里出现了一个中断向量的概念，其实它就是一个被送往 CPU 数据线的一个整数。CPU 给每个 IRQ 分配了一个类型号，通过这个整数，CPU 来识别不同类型的中断。这里可能很多朋友会寻问为什么还要弄个中断向量这么麻烦的东东？为什么不直接用 IRQ0~IRQ15 就完了？比如就让 IRQ0 为 0，IRQ1 为 1……，这不是要简单的多么？其实这里体现了模块化设计规则以及节约规则。

首先我们先谈谈节约规则，所谓节约规则就是所使用的信号线数越少越好，这样如果每个 IRQ 都独立使用一根数据线，如 IRQ0 用 0 号线，IRQ1 用 1 号线……这样，16 个 IRQ 就会用 16 根线，这显然是一种浪费。那么也许马上就有朋友会说：那么只用 4 根线不就行了吗（2^4=16）？

对于这个问题，则体现了模块设计规则。我们在前面就说过中断有很多类，可能是外部硬件触发，也可能是由软件触发，然而对于 CPU 来说中断就是中断，只有一种，CPU 不用管它到底是由外部硬件触发的还是由运行的软件本身触发的，因为对于 CPU 来说，中断处理的过程都是一样的：中断现行程序，转到中断服务程序处执行，回到被中断的程序继续执

行。CPU 总共可以处理 256 种中断，而并不知道，也不应当让 CPU 知道这是硬件来的中断还是软件来的中断，这样，就可以使 CPU 的设计独立于中断控制器的设计，因为 CPU 所需完成的工作就很单纯了。CPU 对于其它的模块只提供了一种接口，这就是 256 个中断处理向量，也称为中断号。由这些中断控制器自行去使用这 256 个中断号中的一个与 CPU 进行交互。比如，硬件中断可以使用前 128 个号，软件中断使用后 128 个号，也可以软件中断使用前 128 个号，硬件中断使用后 128 个号，这于 CPU 完全无关了，当你需要处理的时候，只需告诉 CPU 你用的是哪个中断号就行，而不需告诉 CPU 你是来自哪儿的中断。这样也方便了以后的扩充，比如现在机器里又加了一片 8259 芯片，那么这个芯片就可以使用空闲的中断号，看哪一个空闲就使用哪一个，而不是必须要使用第 0 号，或第 1 号中断。其实这相当于一种映射机制，把 IRQ 信号映射到不同的中断号上，IRQ 的排列或说编号是固定的，但通过改变映射机制，就可以让 IRQ 映射到不同的中断号，也可以说调用不同的中断服务程序，因为中断号是与中断服务程序一一对应的，这一点我们将在随后的内容中详细描述。8259A 将中断号通知 CPU 后，它的任务就完成了，至于 CPU 使用此中断号去调用什么程序它就不管了。下图就是 8259A 芯片的结构：



（图 2 来源《Linux 0.11 内核完全注释》）

上图就是 PC 机中两片 8259A 的联结及 IRQ 分配示意图。从图中我们可以看到，两片 8259A 芯片是级联工作的，一个为主片，一个为从片，从片的 INT 端口与主片的 IRQ2 相连。主片通过 0x20 及 0x21 两个端口访问，而从片通过 0xA0 及 0xA1 这两个端口访问。

至于为什么从片的 INT 需要与主片的 IRQ2 相连而不是与其它的 IRQ 相联，很遗憾，我目前无法回答这个问题：(，如果你知道答案，非常希望你能来信指教！不过幸运的是，我们只要知道计算机是的确是这样联的，并且这样连它就可以正常工作就行了！

三、8259A 的编程

8259A 常常称之为 PIC（可编程中断控制器），因此，在使用的时候我们必须通过编程对它进行初始化，需要完成的工作是指定主片与从片怎样相连，怎样工作，怎样分配中断号。在实模式下，也就是计算机加电或重启时，这是由 BIOS 自动完成的，然而当转到保护模式下后，我们却不得不对它进行编程重新设定，这都是由该死的 IBM 与 Intel 为维护兼容性而搞出来的麻烦：(。

在 BIOS 初始化 PIC 的时候，IRQ0~IRQ7 被分配了 0x8~0xF 的中断号，然而当 CPU 转到保护模式下工作的时候，0x8~0xF 的中断号却被 CPU 用来处理错误！一点不奇怪，CPU 是 Intel 生产的，而计算机却是由 IBM 生产的，两家公司没有协调好：（。

因此，我们不得不在保护模式下，重新对 PIC 进行编程，主要的目的就是重新分配中断号。幸好这不是一件太难的工作。

对 8259A 的编程是通过向其相应的端口发送一系列的 ICW（初始化命令字）完成的。总共需要发送四个 ICW，它们都分别有自己独特的格式，而且必须按次序发送，并且必须发送到相应的端口，下面我们先来看看第一个 ICW1 的结构：

ICW1：发送到 0x20（主片）及 0xa0（从片）端口

7	6	5	4	3	2	1	0
0	0	0	1	M	0	C	I

I 位：若置位，表示 ICW4 会被发送。（ICW4 等下解释）

C 位：若清零，表示工作在级联环境下。

M 位：指出中断请求的电平触发模式，在 PC 机中，它应当被置零，表示采用“边沿触发模式”。

ICW2：发送到 0x21（主片）及 0xa1（从片）端口

7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	0	0	0

ICW2 用来指示出 IRQ0 使用的中断号是什么，因为最后三位均是零，因此要求 IRQ0 的中断号必须是 8 的倍数，这又是一个很巧妙的设计。因为 IRQ1 的中断号就是 IRQ0 的中断号+1，IRQ2 的中断号就是 IRQ0 的中断号+2，……，IRQ7 的中断号就是 IRQ0 的中断号+7，刚好填满一个 8 个的中断向量号空间。

ICW3：发送到 0x21（主片）及 0xa1（从片）端口

ICW3 只有在级联工作的时候才会被发送，它主要用来建立两个 PIC 之间的连接，对于主片与从片，它结构是不一样的。

（主片结构：）

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

上面，如果相应的位被置 1，则相应的 IRQ 线就被用于与从片连接，若清零则表示被连接到外围设备。

（从片结构：）

7	6	5	4	3	2	1	0
0	0	0	0	0	IRQ		

上面的 IRQ 位指出了是主片的哪一个 IRQ 连到了从片，这需要同主片上发送的上面的主片结构字一致。

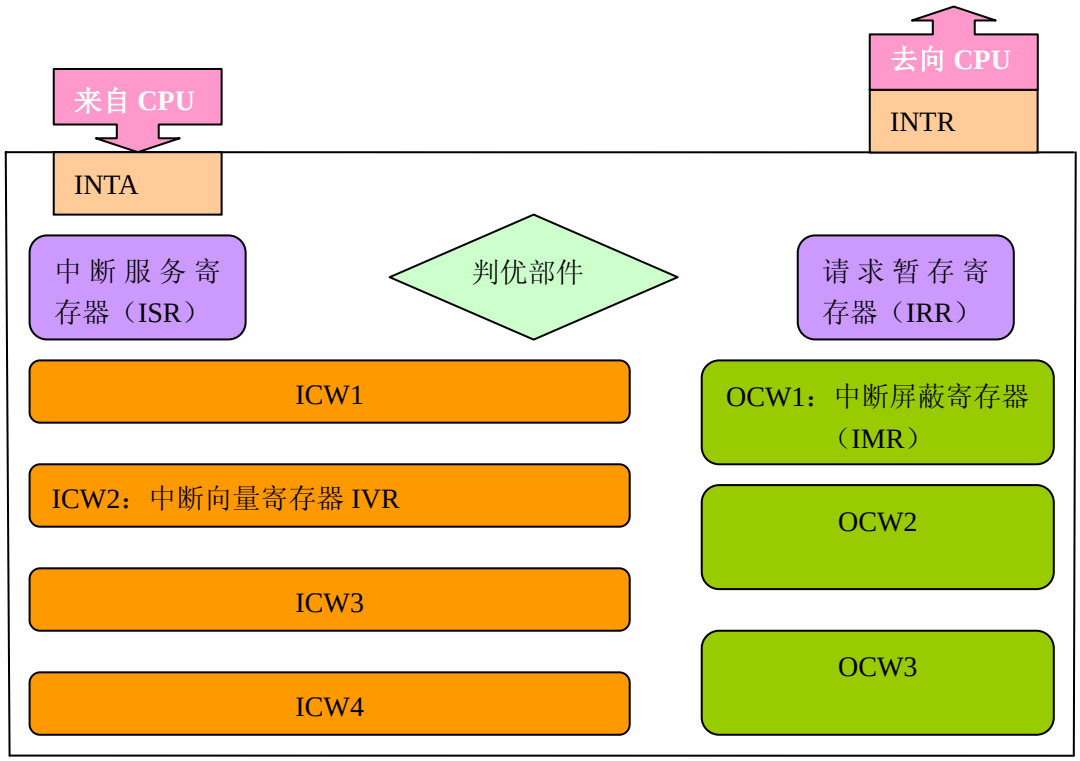
ICW4：发送到 0x21（主片）及 0xa1（从片）端口

7	6	5	4	3	2	1	0
0	0	0	0	0	0	EOI	80x86

80x86 位：若置位则表示工作在 80x86 架构下。
EOI 位：若置位则表示自动清除中断请求信号。在 PC 上这位需要被清零。要理解这一位，我们需要详细了解一下 8259A 的内部中断处理流程。

三、8259A 的内部中断处理流程

下面我们就来从一个系统程序员（System Programmer）的角度看看 8259A 的内部结构。



（图 3）

首先，一个外部中断请求信号通过中断请求线 IRQ，传输到 IMR（中断屏蔽寄存器），IMR 根据所设定的中断屏蔽字（OCW1），决定是将其丢弃还是接受。如果可以接受，则 8259A 将 IRR（中断请求暂存寄存器）中代表此 IRQ 的位置位，以表示此 IRQ 有中断请求信号，并同时向 CPU 的 INTR（中断请求）管脚发送一个信号，但 CPU 这时可能正在执行一条指令，因此 CPU 不会立即响应，而当这 CPU 正忙着执行某条指令时，还有可能有其余的 IRQ 线送来中断请求，这些请求都会接受 IMR 的挑选，如果没有被屏蔽，那么这些请求也会被放到 IRR 中，也即 IRR 中代表它们的 IRQ 的相应位会被置 1。

当 CPU 执行完一条指令时后，会检查一下 INTR 管脚是否有信号，如果发现有信号，就会转到中断服务，此时，CPU 会立即向 8259A 芯片的 INTA（中断应答）管脚发送一个信号。当芯片收到此信号后，判优部件开始工作，它在 IRR 中，挑选优先级最高的中断，将

中断请求送到 ISR（中断服务寄存器），也即将 ISR 中代表此 IRQ 的位置位，并将 IRR 中相应位置零，表明此中断正在接受 CPU 的处理。同时，将它的编号写入中断向量寄存器 IVR 的低三位（IVR 正是由 ICW2 所指定的，不知你是否还记得 ICW2 的最低三位在指定时都是 0，而在这里，它们被利用了！）这时，CPU 还会送来第二个 INTA 信号，当收到此信号后，芯片将 IVR 中的内容，也就是此中断的中断号送上通向 CPU 的数据线。

这个内容看起来仿佛十分复杂，但如果我们用一个很简单的比喻来解释就好理解了。CPU 就相当于一个公司的老总，而 8259A 芯片就相当于这个老总的秘书，现在有很多人想见老总，但老总正在打电话，于是交由秘书先行接待。每个想见老总的人都需要把自己的名片交给秘书，秘书首先看看名片，有没有老总明确表示不愿见到的人，如果没有就把它放到一个盒子里面，这时老总的电话还没打完，但不停的有人递上名片求见老总，秘书就把符合要求的名片全放在盒子里了。这时，老总打完电话了，探出头来问秘书：有人想见我吗？这时，秘书就从盒子里挑选一个级别最高的，并把他的名片交给老总。

这里需要理解的是中断屏蔽与优先级判定并不是一回事，如果被屏蔽了，那么参加判定的机会也都没了。在默认情况下，IRQ0 的优先级最高，IRQ7 最低。当然我们可以更改这个设定，这样在下面有详细描述。

言归正传，当芯片把中断号送上通往 CPU 的数据线后，就会检测 ICW4 中的 EOI 是否被置位。如果 EOI 被置位表示需要自动清除中断请求信号，则芯片会自动将 ISR 中的相应位清零。如果 EOI 没有被置位，则需要中断处理程序向芯片发送 EOI 消息，芯片收到 EOI 消息后才会将 ISR 中的相应位清零。

这里的机关存在于这样一个地方。优先权判定是存在于 8259A 芯片中的，假如 CPU 正在处理 IRQ1 线来的中断，这时 ISR 中 IRQ1 所对应的位是置 1 的。这时来了一个 IRQ2 的中断请求，8259A 会将其同 ISR 中的位进行比较，发现比它高的 IRQ1 所对应的位被置位，于是 8259A 会很遗憾的告诉 IRQ2：你先在 IRR 中等等。而如果这时来的是 IRQ0，芯片会马上让其进入 ISR，即将 ISR 中的 IRQ0 所对应的位置位，并向 CPU 发送中断请求。这时由于 IRQ1 还在被 CPU 处理，所以 ISR 中 IRQ1 的位也还是被置位的，但由于 IRQ0 的优先级高，所以 IRQ0 的位也会被置位，并向 CPU 发送新的中断请求。此时 ISR 中 IRQ0 与 IRQ1 的位都是被置位的，这种情况在多重中断时常常发生，非常正常。

如果 EOI 被设为自动的，那么 ISR 中的位总是被清零的（在 EOI 被置位的情况下，8259A 只要向 CPU 发送了中断号就会将 ISR 中的相应位清零），也就是如果有中断来，芯片就会马上再向 CPU 发出中断请求，即使 CPU 正在处理 IRQ0 的中断，CPU 并不知道谁的优先级高，它只会简单的响应 8259A 送来的中断，因此，这种情况下低优先级的中断就可能会中断高优先级的中断服务程序。所以在 PC 中，我们总是将 EOI 位清零，而在中断服务程序结束的时候才发送 EOI 消息。

四、EOI 消息及 OCW 操作命令字

上面所描述的内容几乎全与 EOI 消息有关，我们现在也已经知道了，应当在中断服务程序结束的时候发送 EOI 消息给芯片，让芯片在这个时候将其相应的位清零。那让我们现在来揭开 EOI 消息的神秘面纱。

要认识 EOI 消息，我们需要先行了解 OCW（操作命令字），它们用来操作 8259A 的优先级、中断屏蔽及中断结束等控制。总共有三个 OCW，它们也都有自己很独特的格式，不过它们的发送却不须按固定的顺序进行。

OCW1：中断屏蔽，发送到 0x21(主片)或 0xa1（从片）端口

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

如果相应的位置 1，则表示屏蔽相应的 IRQ 请求。

OCW2：优先权控制及中断结束命令，发送到 0x20（主片）及 0xa0（从片）端口

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

先来看看中断结束消息（EOI）

EOI 也是 OCW2 型命令中的一种，当 EOI 位被置 1，这就是一个 EOI 消息。SL 是指定级别位，如果 SL 被置位，则表明这是一个指定级别的 EOI 消息，这个消息可以指定将 ISR 中的哪一位清零，即告诉 8259A 应当清除哪一个 IRQ 信号。L2、L1、L0 就用来指定 IRQ 的编号。而在实际运用中我们却将 SL 及 L2、L1、L0 全置零。

SL 置零表示这是一个不指定级别的 EOI 消息，则 8259A 芯片自动将 ISR 中所有被置位的 IRQ 里优先级最高的清零，因为它是正在被处理及等待处理的中断中优先级最高的，也就一定是 CPU 正在处理的中断。

现在再来看看优先级控制命令。

当 R 为 0 时，表明这是一个固定优先权方式，IRQ0 最高，IRQ7 最低。

当 R 为 1 时，表明这是一个循环优先权，比如，如指定 IRQ2 最低，则优先级顺序就为：

IRQ2 < IRQ1 < IRQ0 < IRQ7 < IRQ6 < IRQ5 < IRQ4 < IRQ3

（编者注：可以将其记忆为“IRQ7 < IRQ6 < IRQ5 < IRQ4 < IRQ3 < IRQ2 < IRQ1 < IRQ0”进行循环左移后的结果）

也即，如果 IRQ(i)最低，那么 IRQ(i + 1)就最高。

所以，在这种方式下需要先行指定一个最低优先级。如果 SL 被清零，则表示使用自动选择方式，那么正在被处理的中断服务在下一次，就被自动指定为最低优先级。如果 SL 被置位，那么 L2、L1、L0 所指定的 IRQ 就被指定为最低优先级。

在指定优先级的时候，也可通过置位 EOI，即可以将指定优先级命令与中断结束消息同时发送。

上面的描述看起来很复杂，其实对于一般的操作系统编写来说大多就使用一种形式：0010 0000，我也很乐意将其称为 EOI 消息。

还有一个 OCW3 命令字，可以指定特殊的屏蔽方式及读出 IRR 与 ISR 寄存器，不过一般在操作系统中并不需要这样的操作。在操作系统编写中一般只用到前面两个命令字的格式（至少 pyos 是这样：））由于本文并非硬件手册，只想为操作系统的编写提供一点帮助，因此，如果你想了解完整的 OCW3 命令字格式，请查阅相应的硬件手册，或本文的参考资料 1。

五、上篇结束语

在这一篇中，较为详细的描述了对 8259A 中断控制器芯片编程所需具备的基础知识，在编写操作系统的过程中，我们就需要向相应端口发送相应的 ICW 或 OCW，完成对 8259A 的操作，具体的代码将在下篇中描述。

在下篇中将更主要的描述操作系统对中断服务程序的处理及安排，并以开发中的 pyos 做为例子进行实验。

参考文献

1. 张昆藏. 《IBM PC/XT 微型计算机接口技术》. 清华大学出版社. 1991.3.

【未完待续 • 责任编辑：iamxiaohan@hitbbs】

保护模式下 8259A 芯片编程及中断处理探究（下）

Version 0.02

哈尔滨工业大学 并行计算实验室 谢煜波

简介

在上篇中，我们详细讲述了保护模式下中断处理的基本原理以及对可编程中断控制器 8259A 的编程方法。如果说上一篇更偏重于原理及特定的硬件编程方法，那么本篇就会偏软一点，将详细描述怎样编写操作系统中的中断处理程序，并将通过 pyos 进行验证。在此篇中，你将会详细了解到操作系统是怎样处理中断的，中断处理程序是怎样编写的，操作系统又是怎样调用中断处理程序的。希望本篇可以使你对上述问题有个比较清晰的认识。

本篇是独立的，当然，如果你阅读了上篇，那么对于理解本篇中所描述的内容无疑是有巨大帮助的。

pyos 是一个实验性的操作系统，阅读本篇之后，你可以尝试着改动 pyos 中的中断处理部份，这样你将更可以详细而深入的理解多重中断、现场保护等内容，本篇在最后也将对于怎样进行这样的自我实验做些许描述。如果你在学习“操作系统”或“组成原理”的过程中，对于书中描述的内容感到不太直观，你可以试试用 pyos 去验证你所学习的知识。

再次声明：此文只是我在进行操作系统实验过程中的一点心得体会，记下来，避免自己忘记。对于其中可能出现的错误，欢迎你来信指证。

一、操作系统中断服务概述

现代计算机如果从纯硬件角度，我个人更倾向于将它理解为是利用的一种所谓的“中断驱动”机制，就相当于我们常常津津乐道的 Windows 的“消息驱动”机制一样。CPU 在正常情况下按顺序执行程序，一旦有外部中断到来，CPU 将会中断现行程序的运行，转到中断服务程序进行中断处理，当中断处理完成之后，CPU 再回到原来执行程序被中断的地方继续执行，并等待下一个中断的来临。

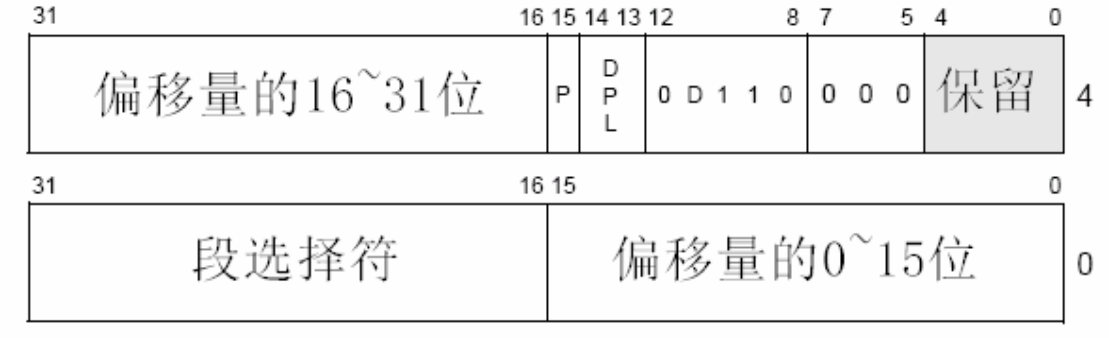
CPU 需要响应的中断有很多种，比如键盘中断、磁盘中断、CPU 时钟中断等等，每种中断的功能都是不同的，而所需要的中断服务程序也是不同的，CPU 又是怎么识别这种种不同的中断的呢？

二、中断描述符表及中断描述符

在上一篇中，我们知道了 CPU 是通过给这些不同的中断分配不同的中断号来识别的。一种中断就对应一个中断号，而一个中断号就对应一个中断服务程序，这样，当有中断到来的时候，CPU 就会识别出这个中断的中断号，并将这个中断号作为一个索引，在一张表中查找此索引号对应的一个入口地址，而这个入口地址就是中断服务程序的入口地址，CPU 取得入口地址后，就跳转到这个地址所指示的程序处运行中断服务程序。

这张存放不同中断服务程序的表在系统中常常称为“中断向量表”。在保护模式下也常常称为“中断描述符表”（IDT），这个表中的每一项就是一个中断描述符，每一个中断描述

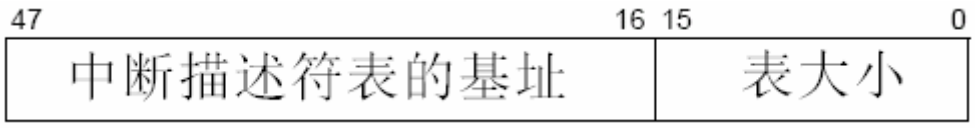
符都包含一个中断服务程序的地址，CPU 通过将中断号作为索引值取得的就是这样一个“中断描述符”，通过“中断描述符”，CPU 就可以得到中断服务程序的地址了，下面，我们就来看看中断描述符的结构：



(图一)

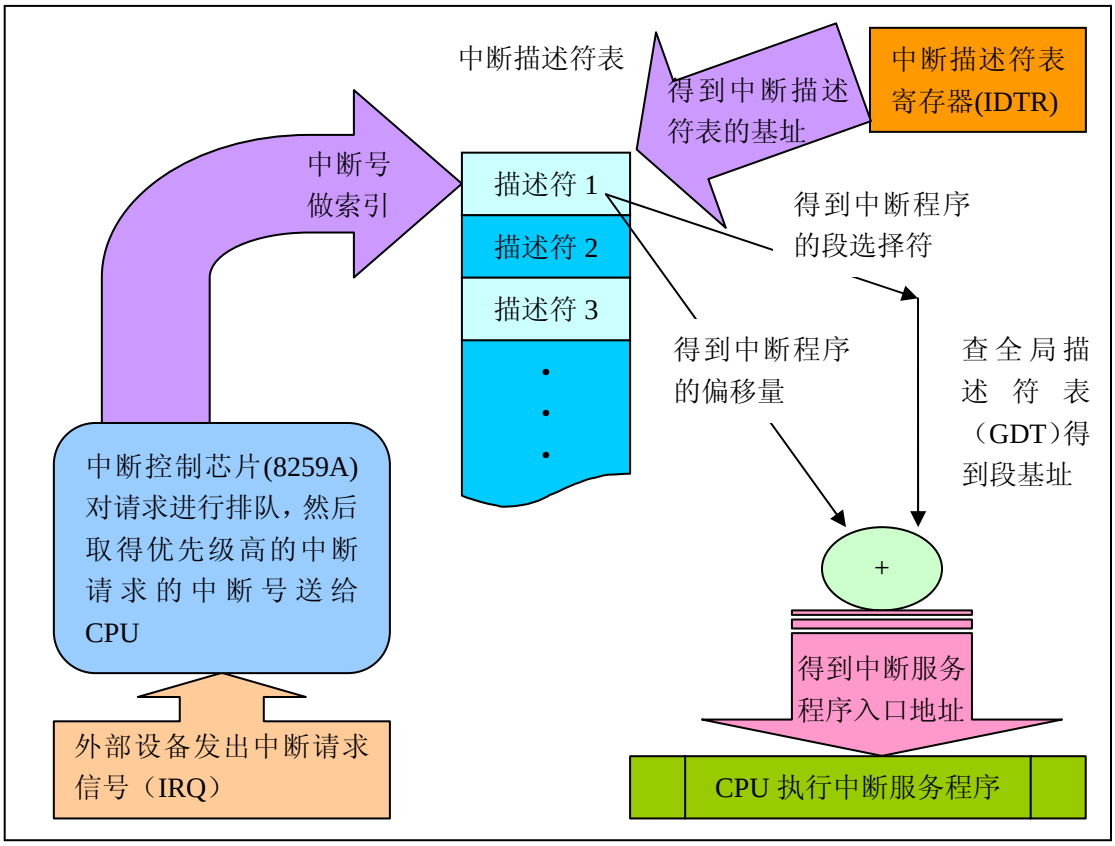
上图就是一个“中断描述符”的结构，其中 P 位是存在位，置 1 的时候就是指这个描符述可以被使用；DPL 是特权级，可以指定为 0~3 中的一级；保留位是留给 Inter 将来用的，在现阶段，我们只需要简单的将其置零就可以了；偏移量总共是 32 位，它表示一个中断服务程序在内存中的位置。由于保护模式下，内存的寻址是由段选择符与偏移量指定的，所以在中断描述符中也分别设定了段选择符与偏移量位，他们共同决定了一个中断服务程序在内存中的位置。有关保护模式下内存的寻址方面的描述，可以参看本系列的另一篇文章：《操作系统引导探究》（编者注：此文在本刊第一期上刊载）。

在前面我们描述了，一个中断描述符是放在一张中断描述符表中的，而中断号就是中断描述符在中断描述符表中的索引或说下标。那么系统又怎么知道中断描述符表是放在什么地方的呢？这在系统中是通过一个称之为“中断描述符表寄存器”（IDTR）实现的，这个寄存器中就存放了“中断描述符表”在内存中的地址。下面，我们也来看看这个寄存器的结构：



(图二)

下面，我们可以比较完整的来欣赏一下 CPU 处理中断的流程：



(图三)

三、pyos 中的中断系统实验

下面，我们将以 pyos 作为例子，用它来实验操作系统中中断系统的实现。当然，在实际的操作系统中这也许是非常复杂的，但我们现在通过 pyos 也完全可以进行这样的实验。在实验正式开始之前，你或许需要下载本实验所用到的源代码，这可以在我们的网站“纯 C 论坛”（http://purec.binghua.com）“操作系统实验专区”中下载（编者注：此源代码已经收录到本刊本期的资源包中），也可以来信向作者索要。对于实验环境的搭建你也许需要看看“操作系统实验专区”中《When Do We Write Our Chinese Os（1）》对此的描述。

在实验正式开始之前，我们将详细描述一下 pyos 的文件组织形式。

3.1 pyos 的文件组织形式

本实验用到的 pyos 目前的文件组织如下：“boot.asm”、“setup.asm”这两个文件是操作系统引导文件，他们负责把 pyos 的内核读入内存，然后转到内核执行（这方面的内容可以参见《操作系统引导探究》一文）。“kernel.cpp”就是 pyos 的内核，pyos 目前是用 c++开发的，因此它的内核看起来非常简单，也比较有结构，下面就是“kernel.cpp”中的内容：

```
#include "system.h"
#include "video.h"

extern "C" void Pyos_Main()
{
```

```
  /* 系统初始化 */
  class_pyos_System::Init() ;
  class_pyos_Video::ClearScreen() ;
  class_pyos_Video::PrintMessage( "Welcome to Pyos :)" ) ;
  for(;;);
}
```

我想，这样的程序也许不用我做什么注释了。“system.h”中定义了一个名为“class_pyos_System”的系统类，用来做系统初始化的工作，“video.h”中定义了一个名为“class_pyos_Video”的显卡类，它封装了对 VGA 显卡的操作，从上面的代码中我们可以看见，系统先通过 class_pyos_System 类的 Init()完成系统初始化，然后调用 class_pyos_Video 类中的 ClearScreen()进行清屏，最后用 PrintMessage()输出了一条欢迎信息。下面，我就一步一步的来剥开上面看上去很神秘的 Init()——系统初始化函数。对于显卡类，你可以参看源代码，本篇中将不进行描述，也许以后我会用专门的一篇来详细描述它。当然，你如果看过《When Do We Write Our Chinese OS（3）》的话，对于显卡类的理解就易容反掌了。

3.2 pyos 的系统初始化

下面，我们来看看 pyos 的系统初始化函数：

```
#include "interrupt.h"
/* 系统初始化 */
void class_pyos_System::Init()
{
  /* 初始化Gdt表 */
  InitGdt() ;
  /* 初始化段寄存器 */
  InitSegRegister() ;
  /* 初始化中断 */
  class_pyos_Interrupt::Init() ;
}
```

是的，他就是这么简单，由于 InitGdt 与 InitSegRegister 的内容在《操作系统引导探究》中已经描述过了，这里我们就专注于我们本篇的核心内容——对于中断系统的初始化。

从上面的代码中我们可以看出，系统首先在“interrupt.h”中定义了一个名为“class_pyos_Interrup”的中断类，专门来处理系统的中断部份。然后，系统调用中断类的 Init()函数，来进行初始化。呵呵，我们马上就去看看这个中断类的初始化函数到底做了些什么：

```
/* 初始化中断服务 */
void class_pyos_Interrupt::Init()
{
  /* 初始化中断可编程控件器 8259A */
  Init8259A() ;
  /* 初始化中断向量表 */
  InitInterruptTable() ;
  /* 许可键盘中断 */
```

```
  class_pyos_System::ToPort( 0x21 , 0xfd ) ;
  /* 汇编指令，开中断 */
  __asm__( "sti" ) ;
}
```

我想，对于这样自说明式的代码，解释是多余的，我们还是抓紧时间来看看它首先是怎样初始化 8259A 的吧：

```
/* 初始化中断控制器 8259A */
void class_pyos_Interrupt::Init8259A()
{
  // 给中断寄存器编程
  // 发送 ICW1 : 使用 ICW4，级联工作
  class_pyos_System::ToPort( 0x20 , 0x11 ) ;
  class_pyos_System::ToPort( 0xa0 , 0x11 ) ;

  // 发送 ICW2，中断起始号从 0x20 开始（第一片）及 0x28开始（第二片）
  class_pyos_System::ToPort( 0x21 , 0x20 ) ;
  class_pyos_System::ToPort( 0xa1 , 0x28 ) ;

  // 发送 ICW3
  class_pyos_System::ToPort( 0x21 , 0x4 ) ;
  class_pyos_System::ToPort( 0xa1 , 0x2 ) ;

  // 发送 ICW4
  class_pyos_System::ToPort( 0x21 , 0x1 ) ;
  class_pyos_System::ToPort( 0xa1 , 0x1 ) ;

  // 设置中断屏蔽位 OCW1 ，屏蔽所有中断请求
  class_pyos_System::ToPort( 0x21 , 0xff ) ;
  class_pyos_System::ToPort( 0xa1 , 0xff ) ;
}
```

从上面的代码可以看出程序是通过向 8259A 发送 4 个 ICW 对 8259A 进行初始化的。其中 ToPort 是 class_pyos_System 类中定义的一个成员函数，它的声明如下：

```
/* 写端口 */
void class_pyos_System::ToPort( unsigned short port , unsigned char data )
```

OK，好像又不需要我多解释了，当然，你也许会问，为什么会发送这几个值的数据，而不是其它值的数据。对于这个问题，因为在“上篇”中已经详细描述了，这里就不再浪费大家的时间了。:）

3.3 初始化 pyos 的中断向量表

从中断初始化的代码中我们可以清楚的看见，pyos 在进行完 8259A 的初始化后，调用 InitInterruptTable()对中断向量表进行了初始化，这可是本篇的核心内容，我们这就来看看这个核心函数：

```
/* 中断描述符结构 */
```

```
struct struct_pyos_InterruptItem{
  unsigned short Offset_0_15 ; // 偏移量的0~15位
  unsigned short SegSelector ; // 段选择符
  unsigned char UnUsed ; // 未使用，须设为全零
  unsigned char Saved_1_1_0 : 3 ; // 保留，需设为 110
  unsigned char D : 1 ; // D 位
  unsigned char Saved_0 : 1 ; // 保留，需设为0
  unsigned char DPL : 2 ; // 特权位
  unsigned char P : 1 ; // P 位
  unsigned short Offset_16_31 ; // 偏移量的16~31位
} ;

/* IDTR所用结构 */
struct struct_pyos_Idtr{
  unsigned short IdtLengthLimit ;
  struct_pyos_InterruptItem* IdtAddr ;
} ;

static struct_pyos_InterruptItem m_Idt[ 256 ] ; // 中断描述符表项
static struct_pyos_Idtr m_Idtr ; // 中断描述符寄存器所用对象

extern "C" void pyos_asm_interrupt_handle_for_default() ; // 默认中断处理函数
/* 初始化中断向量表 */
void class_pyos_Interrupt::InitInterruptTable()
{
  /* 设置中断描述符，指向一个哑中断，在需要的时候再填写 */
  struct_pyos_InterruptItem tmp ;
  tmp.Offset_0_15 = ( unsigned int )pyos_asm_interrupt_handle_for_default ;
  tmp.Offset_16_31 = ( unsigned int )pyos_asm_interrupt_handle_for_default >> 16 ;
  tmp.SegSelector = 0x8 ; // 代码段
  tmp.UnUsed = 0 ;
  tmp.P = 1 ;
  tmp.DPL = 0 ;
  tmp.Saved_1_1_0 = 6 ;
  tmp.Saved_0 = 0 ;
  tmp.D = 1 ;

  for( int i = 0 ; i < 256 ; ++i ){
    m_Idt[ i ] = tmp ;
  }

  m_Idtr.IdtAddr = m_Idt ;
  m_Idtr.IdtLengthLimit = 256 * 8 - 1 ; // 共 256项，每项占8个字节
```

```
  // 内嵌汇编，载入 ldt
  __asm__( "lidt %0" : "=m"( m_Idtr ) ) ; //载入GDT表
}
```

程序首先说明了两个结构，一个用来描述中断描述符，一个用来描述中断描述符寄存器。大家可以对照前面的描述看看这两个结构中的成员分别对应硬件系统中的哪一位。之后，程序建立了一个中断描述符数组 m_Idt，它共有 256 项，这是因为 CPU 可以处理 256 个中断。程序还建立了一个中断描述符寄存器所用的对象。随后，程序开始为这些变量赋值。

从程序中我们可以看出，pyos 现在是将每个中断描述符都设成一样的，均指向一个相同的中断处理程序：pyos_asm_interrupt_handle_for_default()，在一个实际的操作系统中，在最初初始化的时候，也常常是这样做的，这个被称之为“默认中断处理程序”的中断服务程序通常是一个什么也不干的“哑中断处理程序”或者是一个只是简单报错的处理程序。而要等到实际需要时，才使用相应的处理程序替换它。

程序在建立“中断描述符表”后，用 lidt 指令将中断描符述表寄存器所用的内容载入了中断描述符寄存器（IDTR）中，对于“中断描述符表”的初始化就完成了，下面我们可以来看看，pyos_asm_interrupt_handle_for_default()这个程序到底做了些什么事：

3.4 中断处理程序的编写

```
pyos_asm_interrupt_handle_for_default:
  ;保护现场
  pushad
  ;调用相应的C++处理函数
  call pyos_interrupt_handle_for_default
  ;告诉硬件中断处理完毕，即发送 EOI 消息
  mov al , 0x20
  out 0x20 , al
  out 0xa0 , al
  ;恢复现场
  popad
  ;返回
  iret
```

这个程序是在一个名为“interrupt.asm”的汇编文件中，显然，它是一个汇编语言写的源程序。为什么这里又要用汇编语言编写而不直接用 C++内嵌汇编编写呢，比如写成下面这样：

```
void pyos_asm_interrupt_handle_for_default()
{
  __asm__( "pushad" ) ;
  /* do something */
  __asm__( "popad" ) ;
  __asm__( "iret" ) ;
}
```

这里我们需要了解这样一个问题。中断服务程序是由 CPU 直接调用的，随后，它使用 iret 指令返回，而不像一般的 c/c++函数由 ret 返回。c/c++的编译器在处理 c/c++语言的函数的时候，会在这个函数的开头与结尾加上很多栈操作，以支持程序调用，比如上边的代

码就有可能被 c/c++编译器处理成如下形式：(其中绿色为编译器自行加上的代码)

```
pusha
pushad
/* do something */
popad
iret
popa
ret
```

请注意这样一个实事，当程序运行到 iret 时就返回了，而随后的 popa 就不会被执行，因此这样就破坏了堆栈，于是，我们就只能通过汇编语言编写中断处理程序。

现在再来看看 pyos 中用汇编语言写的中断处理程序，首先它用 pushad 指令把寄存器中的内容压入堆栈，这常常称之为保护现场，因为之后程序需要返回被中断的程序中继续运行，因此这些寄存器中的内容也必须在中断处理程序结束时恢复。保护现场完了之后，它调用了一个 c++语言程序 pyos_interrupt_handle_for_default()，随后，它发送了 EOI 消息通知 8259A 中断处理完成（关于 EOI 消息，在“上篇”中有详细描述），然后，它弹出原先保存在堆栈中的寄存器的内容，这常常称为恢复现场，最后通过 iret 返回被中断的程序处继续执行。

晕！原来 pyos_asm_interrupt_handle_for_default()只是一个汇编的壳，而真正的处理函数是 pyos_interrupt_handle_for_default()!怪不得它会在名字中多个“asm”呢？：)。

下面，我们就来看看真正的中断处理程序做了什么：

```
extern "C" void pyos_interrupt_handle_for_default()
{
  /* 处理中断 */
  /* 读 0x60 端口，获得键盘扫描码 */
  char ch = class_pyos_System::FromPort( 0x60 ) ;
  /* 显示键盘扫描码 */
  class_pyos_Video::PrintMessage( ch ) ;
}
```

这才是真正的中断处理程序，不过它的内容很简单，就是读键盘的 0x60 端口，获得键盘的扫描码，然后显示这个扫描码。是不是很简单？:)

3.5 class_pyos_Interrupt::Init()的最后工作

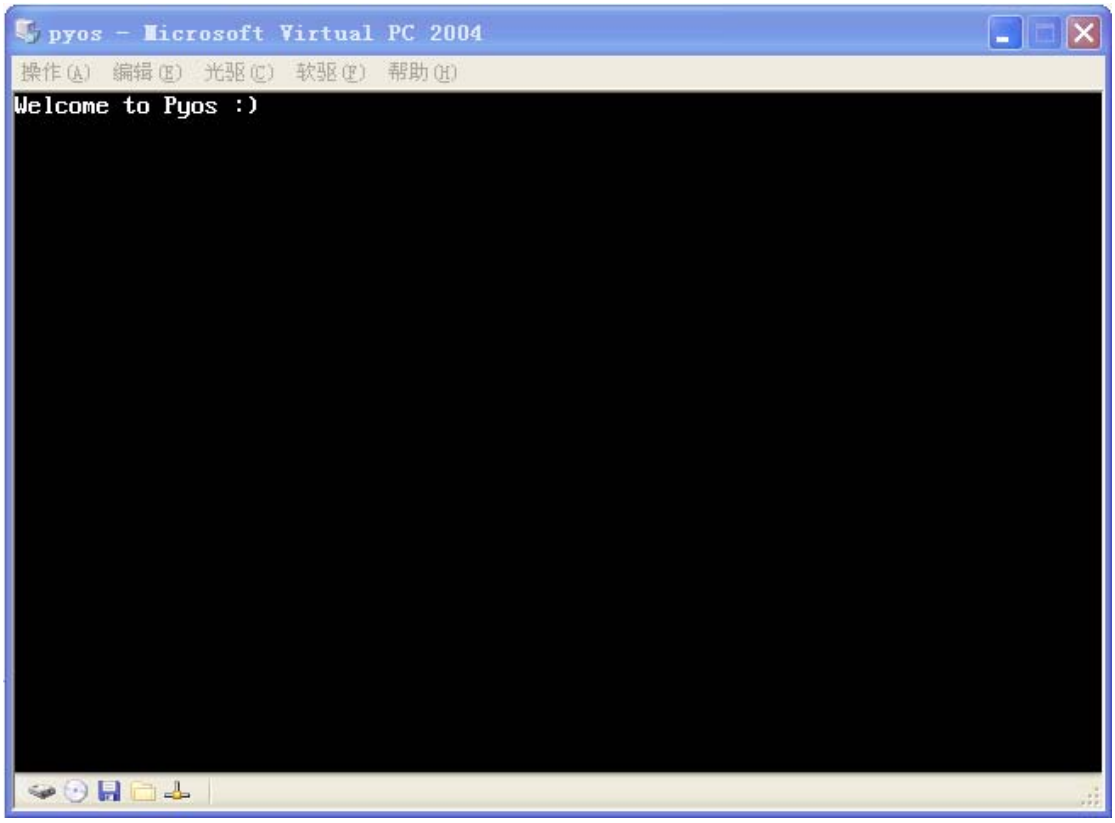
现在让我们重新回到中断类的初始化程序 Init()中吧。Init()在完成了 8259A 及中断描述符表的初始化工作之后，它的工作就也近尾声了。随后，它调用了下面一个程序：

```
  /* 许可键盘中断 */
  class_pyos_System::ToPort( 0x21 , 0xfd ) ;
```

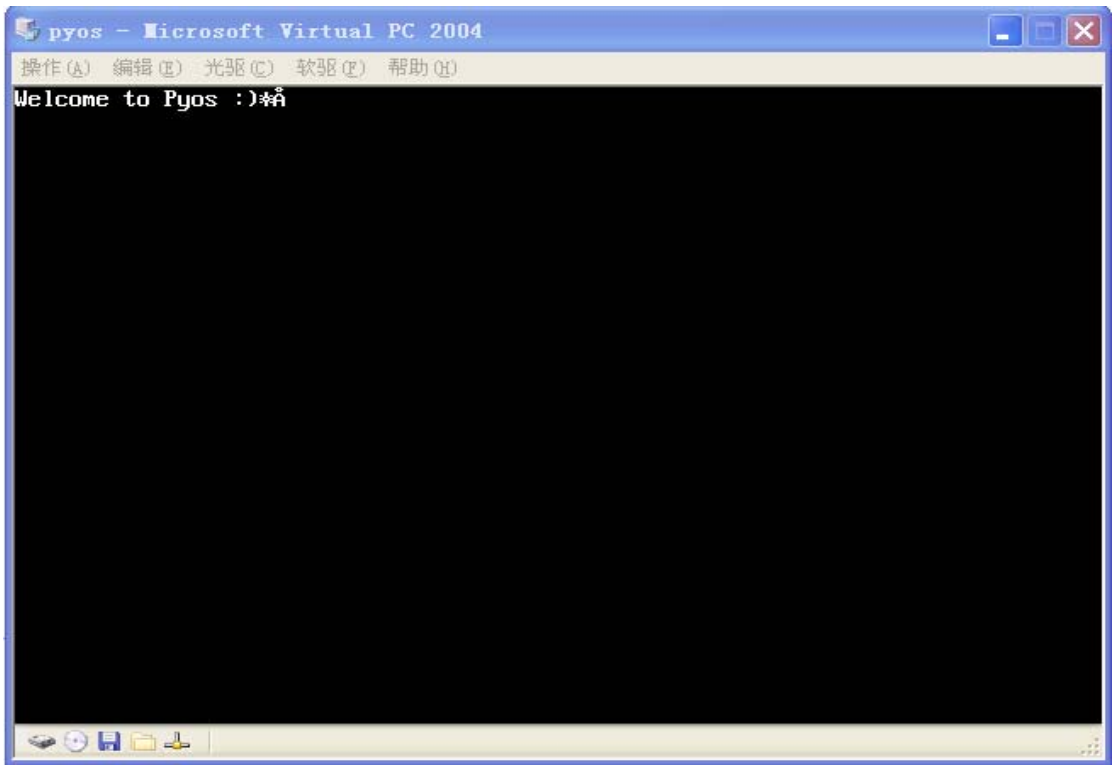
这是向 8259A 发送中断屏蔽字，十六进制 fd 所对应的二进制为 1111 1101，在“上篇”中我们知道了 1 代表屏蔽，而 0 代表不屏蔽，因此 fd 就表示屏蔽了 IRQ0、IRQ2~IRQ7，而唯 IRQ1 没有屏蔽。通过“上篇”我们知道 IRQ1 是代表的键盘中断，因此这一语句的意思就是屏蔽掉除键盘中断之外的所有中断，也即：只响应键盘中断。最后，程序用 sti 汇编指令打开了 CPU 的中断请求功能。

3.6 实验结果

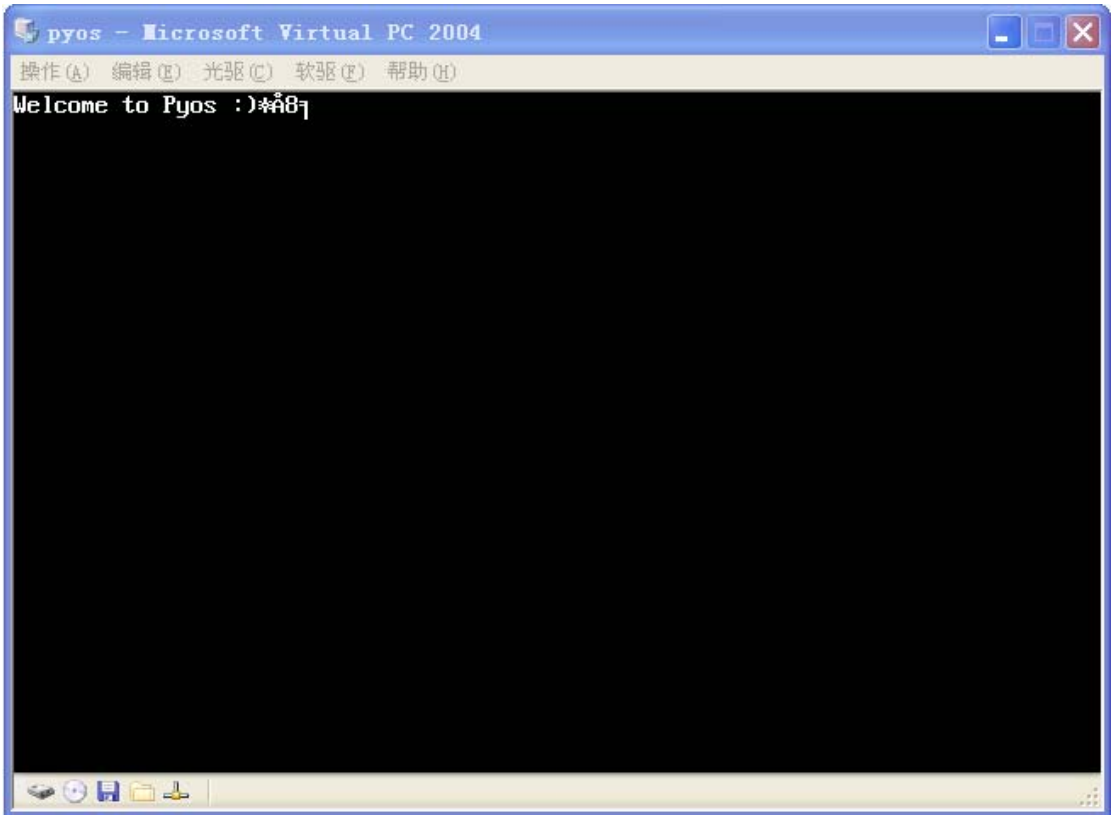
想想我们前面所描述的代码，你现在应当知道了本实验最后所应达到的一个实验结果：pyos 起动后，会响应键盘中断，而中断服务程序的功能是输出键盘的扫描码，也就是说如果你敲击键盘的话，你可以看见计算机的屏蔽上输出键盘的扫描码。下面我们就来看看我们的实验是否达到了程序所预期的效果。下面，我们在 Virtual PC 中启动我们自己的操作系统——pyos：）



我们可以看见，现在 pyos 已经启动了，现在让我们随便敲击一下键盘：



呵呵，屏幕上出现了方才敲键的扫描码。看来我们程序的目的的确是达到了，不过怎么有两个字符呢？再敲一次试试：）



呵呵，又出来两个字符，看来的确是按一个键发生了两次键盘中断，为什么会这样呢？这个问题留到下一篇再解决吧。：）

3.7 实验的改进

pyos 是一个实验系统，开发他的目的就是为了做实验，检验所学，因此，你完全可以用它来进行实验。比如，你可以少屏蔽两个中断，为每个中断指定不同的中断服务程序，以观察多重中断是怎样工作的，中断屏蔽又是怎样起作用的。你还可以用“上篇”所介绍的方法，改变不同中断的优先级，看看中断优先级又是怎样工作的。本实验中的“中断”二字还体现得不明显，但你马上就可以改改，比如在主程序中，不要让它空循环，把原来：

```
for( ;; ) ;
```

改成：

```
for( ;; ){
    class_pyos_Video( '0' ) ;
}
```

然后，你再敲击键盘，你就会体会到“中断”二字的含义。可以完成的实验很多，只要你愿意去做，也非常欢迎你能来信与笔者交流。：）

【全文完 • 责任编辑：iamxiaohan@hitbbs】

浅析 C 语言函数传递机制及对变参函数的处理

哈尔滨工业大学 并行计算实验室 谢煜波

摘 要

本篇文章分析了 C 语言函数参数的传递机制，函数返回值的传递机制，调用约定及变参函数的实现机制。并对 printf()实现原理进行了分析。另外还描述了怎样编写变参函数，及对 va_start()，va_list，va_arg()，va_end()这几个宏的原理进行了剖析。

前 言

对于 C 语言之间的函数调用，我想我们是在熟悉不过了。我们知道，在调用函数时需要把参数传递给所要调用的函数，那么这个参数传递过程内在的机制是什么呢？或说它在编译器内部究竟是怎样处理的呢？

我们同样知道，在使用 printf()函数时，我们可以使用的参数数量是可变的，那么 C 语言又是怎么处理这种可变参数的函数的呢？我们又应当怎么编写这种可变参数的函数的呢？

这篇文章就试图从编译器的角度剖析一下 C 语言中所涉及的函数传递机制，并回答上面两个问题。

如果你想无障碍的阅读本文，你应当对 C 语言比较熟悉，其次你还需要对汇编语言，特别是弹压栈部份有一定了解。

一、C 语言函数参数传递方式

我们首先来看一个实际的例子：

```
/******** a.c ***************/
int f( int a )
{
    return a * a ;
}

void main()
{
    int t = f( 4 ) ;
    ++t ;
}
```

下面我们把这个程序编译，然后看看它反汇编的结果：

```
080482f4 <f>:
 80482f4:       55                      push   %ebp
 80482f5:       89 e5                   mov    %esp,%ebp
 80482f7:       8b 45 08                mov    0x8(%ebp),%eax
```

```
80482fa:       0f af 45 08             imul   0x8(%ebp),%eax
80482fe:       c9                      leave
80482ff:       c3                      ret
```

上面就是 f()的反汇编结果，我们先来分析一下它。

```
80482f4:       55                      push   %ebp
```

这一句是把%ebp 中的内容压入堆栈,因为等下会用到%ebp 的值,所以这里先把原来%ebp 的值保存一下，以便日后恢复。

```
80482f5:       89 e5                   mov    %esp,%ebp
```

这一句是把%esp 的值存入%ebp 中，因此，这句执行之后，%ebp 成了这个函数中所使用的堆栈的栈顶指针。

```
80482f7:       8b 45 08                mov    0x8(%ebp),%eax
```

这一句是把(0x8)%ebp 中的值取到%eax 中，如果用 C 语言伪码表示，这句相当于：

```
%eax = *( %ebp + 0x8 )
```

```
80482fa:       0f af 45 08             imul   0x8(%ebp),%eax
```

这一句是把用(0x8)%ebp 中的值与%eax 中的值相乘，再把结果存为%eax 中，用 C 语言伪码表示，这句相当于：

```
%eax = ( *( %ebp + 0x8 ) ) * %eax
```

在前面一句中，%eax 中已经存了一个*(%ebp + 0x8)的值，故这句相当于：

```
%eax = ( *( %ebp + 0x8 ) ) * ( *( %ebp + 0x8 ) )
```

怪了！怎么没有 i * i 呢？难道(%ebp + 0x8)中存的就是 i 的值？

怎么也没有 return i 呢？难道最后的%eax 的值就是此函数的返回值？

要解决上面两个问题，我们先来看看下面的 main 函数：

```
08048300 <main>:
8048300:       55                      push   %ebp
8048301:       89 e5                   mov    %esp,%ebp
8048303:       83 ec 08                sub    $0x8,%esp
8048306:       83 e4 f0                and    $0xfffffff0,%esp
8048309:       b8 00 00 00 00          mov    $0x0,%eax
804830e:       29 c4                   sub    %eax,%esp
8048310:       83 ec 0c                sub    $0xc,%esp
8048313:       6a 04                   push   $0x4
8048315:       e8 da ff ff ff          call   80482f4 <f>
804831a:       83 c4 04                add    $0x10,%esp
804831d:       89 45 fc                mov    %eax,0xfffffffc(%ebp)
8048320:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
8048323:       ff 00                   incl   (%eax)
8048325:       c9                      leave
8048326:       c3                      ret
8048327:       90                      nop
```

分析一下：

```
8048315:       e8 da ff ff ff          call   80482f4 <f>
```

我们知道 call 语句是用来调用子程序的，故而当执行 call 语句的时候，程序会跳转到 call 语句中所给出的地址，这里是 0x80482f4，也即前面 f()函数的入口地址处开始执行，显然，在这之前必须完成参数传递，否则 f()函数就会收不到参数便被调用执行了。因此，

我们需要看看 call 前面的语句：

```
8048313:        6a 04                           push   $0x4
```

这一句把数 0x4 压入了堆栈。还记得我们在主程序中是怎么调用 f()函数的吗？

```
int t = f( 4 ) ;
```

用来调用 f 的参数刚好也是 4！难道前面的 push $0x4 就是用来传递参数的？

前面的一切都只是我们的假设，现在，我们来分析一下堆栈的变化情况，看看我们的分析正不正确，我们从 8048310: sub $0xc, %esp 语句处开始分析。

下图是执行该语句前的堆栈的状态：

	。。。。。。	地址:996
%esp -----→	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
		地址:976
		地址:972
		地址:968

下面执行 sub $0xc, %esp 语句（注：原文中的堆栈示意图中，esp 的假设值为 992 并不是完全随意的。默认编译选项情况下，在执行 sub $0xc, %esp 之前，esp 值应该是 16 的倍数，而 992 恰好是 16 的倍数。因为 gcc 的缺省编译选项是堆栈 16 字节对齐。所以进入 main 函数前，esp 值为 16 的整数倍。进入 main()后到 sub $0xc, %esp 前，由于返回地址、ebp 入栈（这让堆栈指针 esp 减 8），再执行 sub $8,%esp，使堆栈指针再减 8，这样 esp 值还是 16 的倍数。），此语句的作用是保证在调用函数 f()之前将堆栈对齐到 16 字节，所谓“对齐到 16 字节”就是指让堆栈指针是 16 的整数倍。由于后面会执行一个压栈指令压入参数，而一次压栈会使堆栈指针减 4，故而这里先减掉 12（0xC），这样在压栈后，还可以使堆栈指针，即 esp 的值对齐到 16 字节。不过为什么要对齐到 16 字节？这将在后面解释）[19]。此句执行后，堆栈入下图所示：

	。。。。。。	地址:996
	。。。。。。	地址:992
		地址:988
		地址:984
%esp -----→		地址:980
		地址:976
		地址:972
		地址:968

[19] 此观点感谢本文编辑赵志刚先生指正！

下面执行 push $0x04 语句：

	。。。。。。	地址:996
	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
%esp -----→	4	地址:976
		地址:972
		地址:968

下面执行 call 80482f4 <f> 语句，call 语句会吧 call 语句之后的一条语句的地址，即此时的 IP 中的值压入堆栈：

	。。。。。。	地址:996
	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
	4	地址:976
%esp -----→	IP 值	地址:972
		地址:968

现在，程序跳转 f()函数中了，执行：push %ebp

	。。。。。。	地址:996
	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
	4	地址:976
	IP 值	地址:972
%esp -----→	%ebp	地址:968

执行：mov %esp %ebp

	。。。。。。	地址:996
	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
	4	地址:976
	IP 值	地址:972
%esp, %ebp	%ebp	地址:968

执行：mov　　0x8(%ebp),%eax

注意，现在的%ebp 的值是 968，故而 0x8(%ebp)的值是 968+8=976，而 976 中的值恰好是 main()中通过 push 指令传进来的参数 4！所有的这一切都表明，我们前面的分析与猜测是完全正确的，即：

C 语言中，函数的参数是通过堆栈进行传递的。它由调用它的函数，即调用者把参数压入堆栈里，而被调用的函数在堆栈中取出调用者压入的参数，完成了参数在调用者与被调用者之间的传递。

上面的分析我们解决了前面提出的第一个问题，然而对于第二个问题：“f()是怎么把计算的结果返回给 main()的呢？即 return i 是怎么完成的呢？”我们还不清楚，下面我们继续分析 f()函数。

执行：imul　　0x8(%ebp),%eax

这句前面已经分析过了，这里需要记住的是，现在的计算结果是存放在%eax 中的。

执行：leave

leave 语句其实是两条汇编语句：mov %ebp,%esp 与 pop %ebp 的合成，这两条语句的作用就是恢复原来保存在%ebp 中的堆栈指针%esp 的值，并把保存在堆栈中的%ebp 的值也恢复到%ebp 中。这两句执行之后，堆栈情况如下图所示：

	。。。。。。	地址:996
	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
	4	地址:976
%esp ----→	IP 值	地址:972
	%ebp	地址:968

执行：ret

这条指令从堆栈中弹出 IP 的值，而这个 IP 值就是方才被 call 语句执行时压入的 IP

值，它指出了 call 语句后面一条语句的地址，以便于计算机从 f()返回后，继续执行 call 后面的指令，此语句执行后，堆栈如下图所示：

	。。。。。。	地址:996
	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
%esp ----→	4	地址:976
	IP 值	地址:972
	%ebp	地址:968

这时，程序返回到 main()中，开始执行方才 call 语句后面的那一条语句：

执行：add　　$0x10,%esp

这句堆栈指针越过所压入的参数，即术语中常说的“清栈”，这在后面描述 C 语言函数调用约定的时候还会详细介绍。此句执行后，堆栈如下图所示：

	。。。。。。	地址:996
%esp ----→	。。。。。。	地址:992
		地址:988
		地址:984
		地址:980
	4	地址:976
	IP 值	地址:972
	%ebp	地址:968

此时，整个堆栈与未调用 f()前是一样的。

执行：mov　　%eax,0xfffffffc(%ebp)

0xfffffffc 是-4 的补码，因此，这句用 C 语言伪码表示应当是：

*(%ebp - 4) = %eax

即，把%eax 的值存入(%ebp - 4)[20]的位置。前面我们知道，在调用 f()后，%eax 中其实存放的就是 f()的返回值，这里相当于把 f()的返回值从%eax 中取出，故而我们得到这样一个结论：

C 语言中，返回值可以通过%eax 进行传递，被调用的函数把返回值放在%eax 中，而调

[20] 这里的(ebp - 4)，这其实是在 main 函数中的局部变量 t 的地址，在 C 语言中，局部变量都是建立在堆栈上的，不论这个函数内部使用了多少个局部变量，在进入函数的时候，都会有一个 sub $xxx, %esp 的语句，这个语句就是用来在栈中，留出所有局部变量所占用的空间，有时候，我们在函数内部声明大数组时，比如 int a[10000][10000]，会引起栈溢出，这是由于这个数组需要栈保留出 4 * 10000 * 10000 字节的空间，而栈没有这么大的空间了，故而出错。

用者通过%eax 取得这个返回值。

当然，你也许会产生疑问：如果我返回的是一个结构体，那么%eax 怎么能存得下一个结构体的值呢？下面我们就来看看当返回值是一个结构体的时候，C 语言又怎么处理这个返回值的。

我们把上面编写的程序改动一下，让它返回一个结构体：

```
/******** b.c **************/
struct SS{
        int m_s ;
} ;

struct SS f( int a )
{
        struct SS s ;
        s.m_s = a ;
        return s ;
}

int main()
{
        struct SS s = f( 4 ) ;
        ++s.m_s ;
}
```

同样，将它用 gcc 编译：

```
gcc b.c -o b.o
```

然后，将其编译结果反汇编：

```
objdump -d b.o
```

现在我们来仔细分析一下，还是从主函数 main()开始：

```
0804830c <main>:
804830c:        55                      push   %ebp
804830d:        89 e5                   mov    %esp,%ebp
804830f:        83 ec 08                sub    $0x8,%esp
8048312:        83 e4 f0                and    $0xfffffff0,%esp
8048315:        b8 00 00 00 00          mov    $0x0,%eax
804831a:        29 c4                   sub    %eax,%esp
804831c:        8d 45 fc                lea    0xfffffffc(%ebp),%eax
804831f:        83 ec 08                sub    $0x8,%esp
8048322:        6a 04                   push   $0x4
8048324:        50                      push   %eax
8048325:        e8 ca ff ff ff          call   80482f4 <f>
804832a:        83 c4 0c                add    $0xc,%esp
804832d:        8d 45 fc                lea    0xfffffffc(%ebp),%eax
8048330:        ff 00                   incl   (%eax)
8048332:        c9                      leave
8048333:        c3                      ret
```

假设现在的堆栈如下图所示：

	。。。。。。	地址:992
%esp -----→	。。。。。。	地址:988
		地址:984
		地址:980
		地址:976
		地址:972
		地址:968
		地址:964

执行：push ebp

	。。。。。。	地址:992
	。。。。。。	地址:988
%esp -----→	%ebp	地址:984
		地址:980
		地址:976
		地址:972
		地址:968
		地址:964

执行 mov %esp,%ebp

	。。。。。。	地址:992
	。。。。。。	地址:988
%esp, %ebp	%ebp	地址:984
		地址:980
		地址:976
		地址:972
		地址:968
		地址:964

执行 sub $0x8, %esp

	。。。。。。	地址:992
	。。。。。。	地址:988
%ebp -----→	%ebp	地址:984
		地址:980
%esp -----→		地址:976
		地址:972
		地址:968
		地址:964

执行 and $0xfffffff0,%esp，这句的作用在于强制保证堆栈指针对齐到 16 字节，即保证堆栈指针是 16 的倍数。关于堆栈指针对齐到 16 字节，前面已经多次提到了，这样做的好处是可以获得更好的性能。一般而言，数据对齐到 4 字节效率是最高的。奔腾、奔腾 pro 的 double 和 long double 需要对齐到 8 字节。而 p3 和 p4 的 sse、sse2 数据需要对齐到 16 字节才会获得最高的效率。堆栈对齐不是为了加快入栈、出栈操作的（这样的操作只要对齐到 4 字节就够了），主要是为了加快局部变量的访问速度。[21]

此句执行后，堆栈如图：

	。。。。。。	地址:992
	。。。。。。	地址:988
%ebp -----→	%ebp	地址:984
		地址:980
%esp -----→		地址:976
		地址:972
		地址:968
		地址:964

下面执行：mov $0x0,%eax，与 sub %eax,%esp，这两句并不会影响堆栈的改变，因此我们不用理睬，继续向下看。

执行：lea 0xfffffffc(%ebp),%eax

这一句是关键了！同样，0xfffffffc 是-4 的补码，此句用 C 伪码表示如下：

```
%eax = %ebp + (- 4) = 984 - 4 = 980
```

这句到底是有什么用呢？我们现在先跳过后面的几条语句，直接看最后的两条指令：

```
804832d:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
8048330:       ff 00                   incl   (%eax)
```

为了好看，我们用 C 语言伪码将它改写一下：

```
%eax = %ebp - 4 = 984 - 4 = 980 ;
```

[21] 此观点感谢本文编辑，原中国安天实验室，高级工程师，赵志刚先生的指教。

++(* %eax)

很明显，%eax 中的值 980 是一个地址，而后面一个语句是将这个地址中的数的值增 1，联系我们 main()中的源代码，增 1 操作发生在++s.m_s 处，故 980 这个地址其实就是 main()中局部变量——结构体 s 的成员变量 m_s 的地址。联系到前面的描述：**局部变量都是建立在栈空间中**，这里就应当很好理解了。

下面，继续我们的执行，执行：sub $0x8,%esp，

	。。。。。。	地址:992
	。。。。。。	地址:988
%ebp -----→	%ebp	地址:984
		地址:980
		地址:976
		地址:972
%esp -----→		地址:968
		地址:964
		地址:960
		地址:956

继续执行：push $0x4；push %eax，注意，此处的%eax 已经是 s.m_s 的地址 980 了。

	。。。。。。	地址:992
	。。。。。。	地址:988
%ebp -----→	%ebp	地址:984
		地址:980
		地址:976
		地址:972
		地址:968
	4	地址:964
%esp -----→	980	地址:960
		地址:956
		地址:952

下面继续执行：call 80482f4 <f>

	。。。。。。	地址:992
	。。。。。。	地址:988
%ebp -----→	%ebp	地址:984
		地址:980
		地址:976
		地址:972
		地址:968
	4	地址:964
	980	地址:960
%esp -----→	IP 值	地址:956
		地址:952
		地址:948

这时，程序跳入到 f()中执行:

```
80482f4 <f>:
80482f4:        55                      push   %ebp
80482f5:        89 e5                   mov    %esp,%ebp
80482f7:        83 ec 04                sub    $0x4,%esp
80482fa:        8b 45 08                mov    0x8(%ebp),%eax
80482fd:        8b 55 0c                mov    0xc(%ebp),%edx
8048300:        89 55 fc                mov    %edx,0xfffffffc(%ebp)
8048303:        8b 55 fc                mov    0xfffffffc(%ebp),%edx
8048306:        89 10                   mov    %edx,(%eax)
8048308:        c9                      leave
8048309:        c2 04 00                ret    $0x4
```

由于前面已经分析过，这里我们就不再一句一句的分析 f()函数了，只看看它最关键的地方：

```
80482fa:        8b 45 08                mov    0x8(%ebp),%eax
```

这一句是让 %eax = *（%ebp + 8），按照上面的堆栈分析，我们很容易得出这是将（%ebp + 8）这个地址中存放的 980，也即 main()中 s.m_s 的地址存放在了%eax 中（注意：这里的%ebp 值已经不是上面示意图中的 ebp 值了，因为在 f()的开头两句已经更新了这个 esp 值，大家可以用前面分析 f()的方法分析一下这个 esp 值与我们的描述是不是相合）。

再看这句：

```
8048306:        89 10                   mov    %edx,(%eax)
```

注意，这里是(%eax)而不是%eax，即它是将%edx 的值写进%eax 存的地址中，而不是%eax 中，即将计算的结果直接写到了%eax 所存的地址，即 980 这个地址中。从前面的分析我们知道，这个地址就是 main()中 s.m_s 的地址，故而它是将结果直接存到了 s.m_s 中，而不

是先存到%eax，再由 main()从%eax 中读取，这时计算结果的返回方式与本文最开头的例子是完全不同的。

现在我们可以总结一下：**如果返回值是一个结构体，那么，C 语言函数都将通过直接赋值的方式返回结构体的值。**[22]

在上面的描述中，我们知道了 C 语言函数的基本的参数传递机制及函数返回值的返回机制，不过，我们在上面都是单参数函数，如果是多参数函数又当怎么传递呢？

看看下面的一段代码：

```
/******** c.c ***************/
int f( int a , int b )
{
        return a + b ;
}

int main()
{
        return f( 4 , 5 ) ;
}
```

再看看它反汇编的结果：

```
080482f4 <f>:
80482f4:        55                      push   %ebp
80482f5:        89 e5                   mov    %esp,%ebp
80482f7:        8b 45 0c                mov    0xc(%ebp),%eax
80482fa:        03 45 08                add    0x8(%ebp),%eax
80482fd:        c9                      leave
80482fe:        c3                      ret

080482ff <main>:
80482ff:        55                      push   %ebp
8048300:        89 e5                   mov    %esp,%ebp
8048302:        83 ec 08                sub    $0x8,%esp
8048305:        83 e4 f0                and    $0xfffffff0,%esp
8048308:        b8 00 00 00 00          mov    $0x0,%eax
804830d:        29 c4                   sub    %eax,%esp
804830f:        83 ec 08                sub    $0x8,%esp
8048312:        6a 05                   push   $0x5
8048314:        6a 04                   push   $0x4
8048316:        e8 d9 ff ff ff          call   80482f4 <f>
804831b:        83 c4 10                add    $0x10,%esp
804831e:        c9                      leave
804831f:        c3                      ret
```

注意 main()函数中的这两句，特别注意它们出现的顺序:

[22] 这个世界上没有什么说法是绝对的，这个地方也是，这种说法只是描述的一般情况，实际上当这个返回的结构体只有 4 字节大小，即%eax 可以完全容纳它的时候，某些编译器在优化的时候完全可以通过%eax 而不是直接赋值来返回结果。当然，如果为了同另外的语言或是另外的编译器写的程序正确链接，我们还是应当遵守上述的法则。

```
8048312:       6a 05                    push   $0x5
8048314:       6a 04                    push   $0x4
```

很显然，它是先压入了 5，再压入了 4，在看 main()函数中我们是怎么调用的：

```
f( 4 , 5 ) ;
```

我们很显然能得到这样一个结论：**C 编译器是从右到左依次把函数的参数压入堆栈中的！**

由这个结论能得到一个显然的“推论”，**在函数使用时，需要从堆栈中依次取出参数！**

OK，问题来了，我怎么知道应当取多少参数，每个参数多大呢？呵呵，其实就是函数头提供的信息！比如一个函数头这样写：

```
void f( int a , double b )
```

那么该函数就让编译器知道，它有两个参数存在于堆栈中，靠近栈顶的是参数 a，4 个字节，接着的是参数 b，8 个字节。注意，这里的压栈顺序是从右到左压栈，故而，double b 被先压栈，而 int a 后压栈，按栈的性质，后进先出，故而反倒是后压栈的 int a 更靠近栈顶，而先压栈的 double b 在它后面。按这样的方式，如果%esp 处存放的是 int a 的值，那么在%esp + 4 处存放的就应当是 double b 的值，如果 double b 后还有参数，那么它应当存放到%esp + 4 + 8 的位置，这里 + 8 是因为一个 double 型参数会占据 8 个字节的栈空间。

看看 main()函数最后：

```
804831b:       83 c4 10                 add    $0x10,%esp
```

这里为什么要将%esp 加上 0x10 呢？我们看一下前面：

```
804830f:       83 ec 08                 sub    $0x8,%esp
8048312:       6a 05                    push   $0x5
8048314:       6a 04                    push   $0x4
```

这里首先将 %esp 的值减掉了 8，然后，又压了两个参数，每个 4 字节，共 8 字节，故而堆栈总共减少了 16（0x10）字节，因此最后要将%esp 加上 16 字节，以恢复堆栈空间。

从这里我们可以知道：**C 语言中，函数调用前，调用者会将参数按从右至左的顺序压入堆栈，调用完后，会恢复堆栈，清除掉参数所占用的堆栈空间。这就是所谓的“从右到左压栈，谁调用谁清栈”机制！这其实是一个调用约定，被称为“__cdecl”调用约定，它是 C 语言的默认调用约定。**[23]

二、C 语言函数的变参机制

上一节我们了解了 C 语言函数的调用方式，这个调用方式的一个极大的好处就是可以对变参函数进行良好的支持。我们先来看一个变参函数的典型的例子：

```
int a = 100 ;
int b = 10 ;
printf( "%d %d" , a , b ) ;
```

相信上面一段程序对很多人而言应当是再熟悉不过了，printf()就是一个很典型的变参函数，它可以接受的参数数量是可变的，那么这到底是怎么实现的呢？

我们来用上一节的知识分析一下这个函数：

---

[23] C 语言可以使用很多种调用约定，比如__cdecl，__stdcall，__pascall 等，不同的调用约定主要就是说明压栈顺序是从左到右还是从右到左，是调用者清栈还是被调用函数清栈，而并没有什么深刻的道理，而且不同的编译器对调用约定的支持也是不同的，并不是所有的调用约定都会被所有的编译器所支持，但__cdecl 应当会被所有符合标准的 C 编译器支持。在声明函数的时候，我们可以声明它的调用方式，比如：

```
int __cdecl f( int a , int b )
```

当然，如果不声明，则按编译器默认的调用约定处理，这通常是__cdecl 调用约定。

首先，编译器按从右至左的顺序压栈，则堆栈中第一个被压入的就是 b，其次是 a，最后是控制字符串“%d %d”。这时,栈中东东的顺序是：字符串“%d %d”，然后是参数 a，最后是参数 b。

现在调用 printf()函数，printf 函数从堆栈中依次取出内容，首先得到的就是字符串“%d %d”。

然后，printf()函数扫描这个字符串，首先碰到的就是%d，于是，printf()知道栈中下一个参数是一个 int 型的参数，故而从堆栈中取出这个参数，得到 a 的值，然后 printf()输出 a，随后，printf()继续扫描字符串，得到空格，printf()按原样输出空格，继续扫描，得到又一个%d，于是 printf()再到栈中取出参数 b 的值，然后 printf()输出 b，再扫描字符串，碰见字符串的结束标志\0，于是乎 printf()认为输出结束，于是 printf()返回。

基于上面的分析，我们也可以这样写 printf()函数：

printf("" , a , b) ;

printf("%d %d") ;

这同上面的 printf("%d %d" , a , b)的作用是一样的。

（编者注：文中这段斜体字的描述仅当没有优化编译，并且是默认编译选项下才是正确的。由于连续两次 printf 调用，当堆栈对齐到 16 个字节时，造成了两次调用中的控制字符串地址被压入了同一个堆栈地址，而其上方的第一次压入的两个参数未被破坏。）

这里可以发现，上一节所描述的，C 语言所独特的从右向左压栈顺序是实现这种变参数函数的关键！这真是一个非常巧妙的设计，这样，无论有多少个参数，最后在栈顶的一定是控制字符串，而 printf()并不需要知道有多少个参数被传递进来了，而反正每次从栈顶取出控制字符串，然后分析这个控制字符串，再依次从栈中取出其余参数就行了。当然，由于 printf()本身并不知道有多少个参数传进来了，因此，它不能清栈，而需要调用它的函数清栈，因为，只有调用它的函数才知道它压了多少个参数在栈中，从这一点，我们再一次体会到了这种参数传递方式的巧妙！

C 语言为了简单，支持两种类型的参数压栈：一种就是对每一个参数压栈一次，因此，每次压栈都是 push 进一个 4 字节大小的数据；还有一种就是对每个参数压栈两次，每次压入一个 4 字节大小的数据(注：其实这个不是通过 push 指令实现的,后面再描述),对于 char、short、int、long……及各种指针,这些参数都是一次压栈(这也是我们有时候常常说，存在从 char、short 到 int 的隐式转换的原因之一)，对于 double，由于 double 是 8 字节，于是对于它是两次压栈，每次压入 4 字节（注：这个描述不准确，只辅助你理解 double 被压入堆栈时所占空间的大小，下面会对此有准确的描述），现在还留了个 float，一个 float 是 4 字节，本来一次压栈中可以了,但 c 语言为了统一及一些可移植方面的考虑,就把整个浮点数都作为一种类型处理，于是 float 也像 double 一样进行两次压栈，但由于 float 只有 4 字节，怎么两次压栈呢？编译器就把这个工作借用 cpu 的浮点寄存器完成了。它先用 flds 指令把 float 型的数据载入浮点寄存器,然后用 fstpl 把这个数再存回到堆栈中,这里才真正完成了数据类型的转换！即从 float 到 cpu 中的 extend double，再由 cpu 中的 extend double 到 double。同此类似，double 型的参数也是由 cpu 的浮点寄存器完成压栈的，即先让 cpu 将其载入到浮点寄存器中（这是从 double 到 cpu 中的 extend double 的转换），然后再从浮点寄存器中直接压入堆栈（这是从 extend double 再到堆栈中的 double 的转换）。这样，float 与 double 的处理就统一起来了。所以，在 printf()的控制字符串中只有%f 描述符，而这个描述符描述的只是 double 而没有 float 型，这是因为对于 printf()来说，输入的浮点数只有一种类型就是 double，长度是固定的，就是 8 字节。当然，你或许会说，要是我的一个结构体有 100 个字节，那它怎么传？其实编译器在本质上，把你这 100 个字节的结构体当作 25 个 int，每次压入一个，共压入 25 次，也就是说，结构体的成员，是由编译器一个一个的压入的，也是

由 printf()一个一个的取出的，故而，当结构体太大时，这样传送整个结构体是非常低效的。

所以，对于 printf()来说，参数的转换只存在于参数是 float 型或是 double 型时，对于其余类型的参数，在传递给 printf()的时候都不会进行转换，只是当 printf()从堆栈中取出数据的时候，会根据对应的是%f 还是%d 之类的控制符来解释堆栈中是什么类型的数据。这也就是很多书上强求 printf()中控制字符串与传入参数的类型及个数要严格一致的原因。

看看下面的例子

```
unsigned int hi = 0xbfe40000 ;
unsigned int lo = 0x0 ;
printf( "%f\n" , lo , hi ) ;
```

因为，-0.625 的 double 编码是 0x bf e4 00 00 00 00 00 00 所以，上面这段代码相当于：

```
printf( "%f\n" , -0.625 ) ;
```

三、编写变参函数

C 语言为我们提供了很方便的处理变参函数的机制，我们也在实际中常常使用变参函数，但我们自己却很少编写编参函数，下面我们就来看看怎么自己编写变参函数。

编写变参函数最简单最明了的方法，我个人认为是用汇编语言编写，按照前面所描述的基理，我们完全可以编写出自己的变参函数，然而不少朋友可能对怎么编写汇编函数不太熟悉，因此，这里我们就来看看 C 语言为我们提供的它自己的解决方案，使用：va_start()，va_list()，va_arg()与 va_end()来编写变参函数。

首先，我们来看一个例子：

```
// crt_va.c
#include <stdio.h>
#include <stdarg.h>
int average( int first, ... );
int main( void )
{
   /* Call with 3 integers (0 is used as terminator). */
   printf( "Average is: %d\n", average( 2, 3.1, 4.1, 0 ) );

   /* Call with 4 integers. */
   printf( "Average is: %d\n", average( 5, 7.2, 9.3, 11.4, 0 ) );

   /* Call with just 0 terminator. */
   printf( "Average is: %d\n", average( 0 ) );
}

/* Returns the average of a variable list of integers. */
int average( int first, ... )
{
```

```
    int count = 0, sum = 0, i = first;
    va_list marker;

    va_start( marker, first );     /* Initialize variable arguments. */
    while( i != 0 )
    {
        sum += i;
        count++;
        i = va_arg( marker, double);
    }
    va_end( marker );              /* Reset variable arguments.      */
    return( sum ? (sum / count) : 0 );
}
```

使用是很简单的，首先，用 va_list 声明一个变参链表，然后用 va_start 初始化它个变参链表，然后每次调用 va_arg()来取得链表中的参数，最后用 va_end()释放这个变参链表。如果你只想知道怎么编写变参函数，那么，看到这里就可以了，但如果你想知道为什么使用这几个“函数”它就能实现上述功能呢？那么你可以继续往下看~~

其实这几个用 va_开头的“函数”都是在 stdarg.h 中定义的宏，那么这些宏到底都干了些什么好事呢？我们就把它们展开一下。

我们将它用 vc 的命令行工具将它展开一下：

cl /P crt_va.c

下面就是处理的结果（我已经将无关的东东删了）

```
typedef char *  va_list;
int average( int first, ... );
int main( void )
{

    printf( "Average is: %d\n", average( 2, 3.1, 4.1, 0 ) );

    printf( "Average is: %d\n", average( 5, 7.2, 9.3, 11.4, 0 ) );

    printf( "Average is: %d\n", average( 0 ) );
}


int average( int first, ... )
{
    int count = 0, sum = 0, i = first;
    va_list marker;

    /* 下面是 va_start( marker, first ) 的展开 */
    ( marker = (va_list)( &(first) ) + ( (sizeof(first) + sizeof(int) - 1) &
```

```
~(sizeof(int) - 1) ) );
    while( i != 0 )
    {
        sum += i;
        count++;
        /* 下面是 i = va_arg( marker, double); 的展开 */
        i = ( *(double *)((marker += ( (sizeof(double) + sizeof(int) - 1) &
~(sizeof(int) - 1) )) - ( (sizeof(double) + sizeof(int) - 1) & ~(sizeof(int) - 1) )) );
    }
    /* 下面是 va_end( marker ); 的展开 */
    ( marker = (va_list)0 );
    return( sum ? (sum / count) : 0 );
}
```

注意蓝色的部份，首先，val_list 被定义成了 char * 的别名，故而 va_list marker；其实就是 char * marker ;即我们先前所谓声明的变参链表其实本质上就是一个指针。不过这个指针为什么会定义为 char * 型呢？哪儿来的字符串呢？

我们知道每个 char 只占 1 个字节，因此，如果对 char * 型的指针进行加 1 或减 1 操作，刚好可以让地址加 1 或减 1，如果是 int * 型，那么每次加 1 或减 1 操作就会让地址加 4 或减 4，显然，在需要精确计算地址时，用 char * 无疑是最方便的，因此，这里我们只能将 marker 看着是一个内存地址，而不是一个存放字符串的数组的指针，这再一次说明了，没有什么是绝对的，看问题必须看本质。

先看最后一个 va_end(marker) ;的展开：

```
/* 下面是 va_end( marker ); 的展开 */
    ( marker = (va_list)0 );
```

非常简单，其实就是 marker = 0 ; 所以所谓的释放变参链表，就是将变参链表的指针置为 0。

下面来看看这个：

```
    /* 下面是 va_start( marker, first ) 的展开 */
    ( marker = (va_list)( &(first) ) + ( (sizeof(first) + sizeof(int) - 1) &
~(sizeof(int) - 1) ) );
```

这句是初始化变参地址，我们来看看它都做了些什么，上面这个东东太复杂了，我们来把它简化一下，由于 sizeof(int) - 1 = 3 ;故而，上式可化为：

```
marker = &first + ( sizeof( first ) + 3 ) & ~3
```

还是很郁闷，不太好看，我们再简化一下，把~3 写做 0xffff fffc

```
marker = &first + ( sizeof( first ) + 3 ) & 0xffff fffc
```

注意，巧妙之处来了！0xc 的二进制码是 0x1100，所以 & 0xffff fffc 这个操作的效果就是把(sizeof(first) + 3)的最后两位置零！这其实是这样一个数学事实：

1. 求一个不小于 sizeof(first) 的最大整数 m

2. 保证这个整数 m 是 4 的倍数。

这下，这段代码的意图就相当明显了，因为参数是通过压栈传递的，而每次压栈总是压入 4 个字节，因此，first 所占的栈空间一定是 4 字节的倍数，就算 first 本身并不是 4 字节的倍数，但编译器也会将其补足成 4 字节的倍数，因此，上面这个初始化参数表的 va_start()宏的意义在于，把 marker 定位于 first 之后，即第二个参数的位置！

下面我们来看看，更为精妙的 va_arg()宏的展开：

```
/* 下面是 i = va_arg( marker, double); 的展开 */
      i = ( *(double *)((marker += ( (sizeof(double) + sizeof(int) - 1) & ~(sizeof(int) - 1) )) - ( (sizeof(double) + sizeof(int) - 1) & ~(sizeof(int) - 1) )) );
```

老规矩，我们先来把它简化一下，做一个变量代换，令：

```
A = (sizeof(double) + 3) & ~3 ;
```

则原式化为：

```
i = *(double *)( (marker += A ) - A) ;
```

下面就好理解得多了，首先让 marker = marker + A，这是让 marker 指向当前参数的下一个参数，注意前面已分析过的 A 的作用，这里前面的 sizeof(first)被换成了 sizeof(double)，因为，这个当前参数是 double 型的，这是在 `va_arg( marker, double)` 中指定的。于是，上式其实等于：

```
i = *( double * )( marker - A ) ;
```

这下谁都能看懂了吧，方才 marker 已被定位到下一个参数，故而这里再减 A，得到当前参数的位置，然后让 i 等于当前参数。由于这个当前参数是个 double 类型，故而在使用上进行了一下转化。

现在我们对整个 C 语言的参数机制分析也就告一个段落了，希望能对您有用。如果您发现其中有什么不当之处，欢迎您来信与我联系。

附：

这篇文章原本还有如下一段：

另外顺带提一下，比如：

```
void k()
{
        printf( "", 1 ) ;
}
```

这个函数中，printf() 里的控制字符串为空，而参数表却不为空，那么按理，1 被压入堆栈了，而没有被 printf()弹出来，这一来堆栈岂不是乱七八糟了？是否我一调用这个函数，程序马上就崩溃了？其实不会，因为 C 语言为了避免出现类似这种很可能把堆栈弄得乱七八糟的情况，引入了堆栈保护机制，把这个函数反汇编一下：

```
k:
        pushl   %ebp            //先保存 ebp
        movl    %esp, %ebp      //esp -> ebp , 这个时候的 esp 被保存在 ebp 中了
        subl    $8, %esp        //下面两行执行后 esp = esp - 8 产生一点空闲空间
        subl    $8, %esp        //并没有什么特别的原因
        pushl   $1              //压入参数 1 , 注意它们的压栈顺序,先压入的是最
                                //左边的参数
        pushl   $.LC0           //压入控制字符串 "" 的地址,即指向它的指针!
        call    printf          //调用 printf
                                //注意,现在 printf 返回了,堆栈已经被搞坏了!
        addl    $16, %esp       //这里是恢复前用两条 subl $8 , %esp 产生的空闲
                                //空间,但已经余是无补了,堆栈还是坏的!
        leave                   //注意,这是关键了!
                                //leave 语句其实就是
```

```
                                //1.   movl %ebp , %esp ; ebp -> esp
                                //2.   pop %ebp
                                //注意在上面第二行中,esp -> ebp, 这是把最初进
                                //入函数时正确的 esp 保存在了 ebp 中,现在这里的语
句
                                //1.,把这个保护在ebp中的正确的esp又恢复了!所以
                                //虽然前面的printf()把堆栈搞坏了,但由于leave指
                                //令的存在又把堆栈修好了!:P
        ret
```

其实，这就是 C 中所谓的堆栈保护，如果你的 printf()完全正常使用，是用不着这些保护操作的，很多时候，一种技术并不是很高深莫测，也不是非要怎么做，只不过是由于实际要求而做的一些优化或者妥协，说透了，也就自然而然了^^^:P

投稿后，在本文编辑的认真审校中，发现其中笔者犯的一个错误，即笔者认为在 printf()函数在取出参数的过程中进行了弹栈操作，即 printf()取参数是通过弹栈的形式完成的。其实这个理解是错误的，被调用的函数 printf()虽然是取出了参数，但取出参数的方式不是通过弹栈的形式进行的，而是通过如同 0xfffffffc(%ebp)这样的形式进行的，故而即使是 printf(“” , 1)也不会破坏堆栈，也正对应了前文中所描述的：参数的清栈工作是由调用者而不是被调用函数完成的。下面就是本文的编辑对上面那段内容的评注：

（编者注：文中斜体部分所描述的是错误的。printf 并不会把任何参数弹出栈。正如文中前面强调的，是由调用者恢复堆栈的，printf("" , 1)调用不会造成堆栈混乱，addl $16, %esp 语句正是用来恢复堆栈的，恢复的是第二个 subl $8, %esp 、 pushl $1 和 pushl $.LC0，这是堆栈对齐造成的。第一个 subl $8, %esp 也是用来堆栈对齐的，进入函数 k 前，堆栈是 16 字节对齐的，进入后由于返回地址入栈、%ebp 入栈，再加上减 8 正好又成为 16 字节对齐了。）

除此之外，本文编辑还对文中多个不当此处进行了指正，在此，对本文编辑，原中国安天实验室，高级工程师，赵志刚先生表示由衷的感谢与敬意！

【全文完 • 责任编辑：hitool@hitbbs】

第 29 届 ACM 国际大学生程序设计竞赛
亚洲区北京赛区预选赛试题题解（一）

哈尔滨工业大学 ACM/ICPC 组织

【编者按】:

第 29 届 ACM 国际大学生程序设计竞赛（ICPC）亚洲区北京赛区预选赛于 2004 年 11 月 14 日在北京大学举行，来自全国 52 所高校的 72 个代表队参加了比赛，他们是在此前的网上预选赛从 360 支报名队伍中脱颖而出的。比赛时间从上午 9 点到下午 2 点，最后上海交通大学的 Cappuccino 代表队以做出 5 道题目总费时 643 分钟的成绩获得冠军，同时获得了参加世界总决赛的资格。本刊将对此次比赛的试题作详细分析和解答，欢迎大家关注。

题目请参考 http://acm.pku.edu.cn 或者本期附件。

本期刊出 A，B，C，H 题的解题报告，作者分别是宋鑫莹，刘禹，肖颖和刘子阳，对应的源代码请看附件。

ACM ICPC Beijing Regional Contest 2004

A 题 Finding Nemo 解题报告

Written by Song[24]

问题描述：

Nemo 被困在一个由墙和门构成的迷宫里，所有的墙都与 x 或 y 轴平行，厚度不计，门开在墙中。Marlin 从(0,0)点出发，要穿过最少的门找到 Nemo。给出所有墙和门的坐标，给出 Nemo 的位置，求 Marlin 要穿过最少的门的数量。

数据限制：所有的墙两端坐标都是整数，范围[1,199]。门的长度总是 1。

参考 HOJ 1732。[25]

下图给出一个例子：

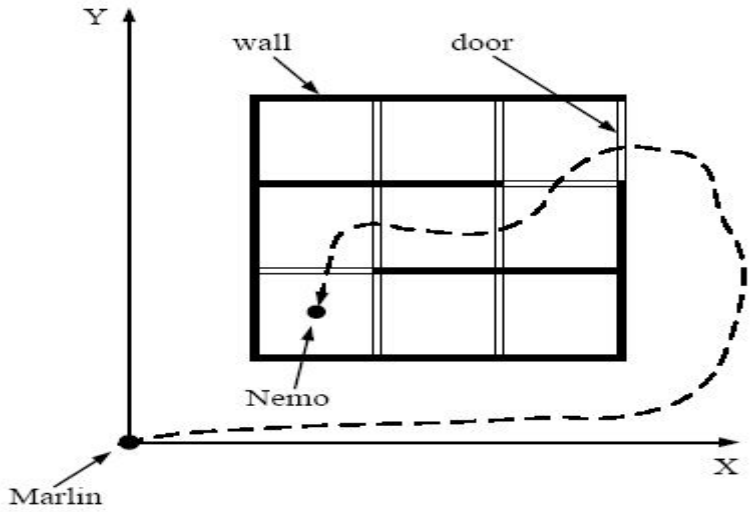


Figure-1. Labyrinth and Path

解题思路：

题目所求是一种最短路，可以用 bfs[26]解决。将平面分成 200*200 共 400 个单位格子，每个格子看成一个点，格子与其相邻的四个格子间或有墙，或有门，或什么也没有。由于要经过最少数量的门，所以应该用优先队列来组织 bfs 搜索顺序：每次出队的应当是从源点出发经过最少的门到达的点。这样就能解决这道题目了。

具体实现：

1. 所有格子可用 200*200 的 int 数组表示(每个格子用该格子的左下角整数点坐标表示)。

2. 所有墙和门可用 200*200*2 的 bool 数组表示(如 wall[1][1][0]=1 表示从(1,2)点出发有一水平长度为 1 的墙，wall[1][1][1]=1 表示从(1,2)点出发有一长度为 1 的墙)。

3. 设一 int 数组 bfn[200][200], bfn[表示从(0,0)格子到(x,y)格子最少需要经过的门的数量，并以此组织优先队列。

细节参考源代码。

---

[24] 宋鑫莹，哈尔滨工业大学实验学院学生，哈尔滨工业大学 ACM/ICPC 代表队成员。电子邮件：songxy@hit.edu.cn

[25] 此处 HOJ 1732 是指哈工大 ACM/ICPC 网站（http://acm.hit.edu.cn）在线测试题库的第 1732 号题，下同。

[26] bfs，breadth first search，广度优先搜索算法

一些细节：

1. bfs 可以从源点到目标点，也可以反向进行。本题在实现中我是反着进行的，即从 Nemo 走到 Marlin。同时把所有从(0,0)格出发不经过门可达的点都标记，这样搜到其中之一便可结束，而不必搜到(0,0)点。
2. 由于题目坐标范围要比实际所给大，可以定出迷宫的外框，这样可以减少搜索的点的数量。
3. 经过 1,2 的优化，程序速度可大大提高。在 HOJ 上可由 0.63s 提高到 0.07s。
4. 最后注意 test data 中 Nemo 的位置可能在[0, 200] * [0, 200]之外，若不考虑，会 sigsegv, 挺阴险。 ：)
5. 类似的题目还有 HOJ 1488 TMD，同样是优先队列的 bfs。

【全文完 • 责任编辑：xiong@hitbbs】

ACM ICPC Beijing Regional Contest 2004

B 题 Searching the Web 解题报告

Written by Liu Yu[27]

在 ACM/ICPC 竞赛中有一类问题，题目中会给出比较明确的算法流程，一般按照题目描述的路线就能够解决，通常称之为模拟题。这一类题目有着比较清晰的解体路线，所以更多的时候旨在考察选手的阅读和现场学习的能力以及细致程度等素质。在近年的比赛中模拟题一般作为简单题或中等难度题目出现。2004 年度北京赛区的 B 题 Searching the Web 就属于这种类型。

熟悉信息检索方法的读者只需要看过第一段就可以估计到题目描述的是倒排表（Inverse Table）的工作原理，题干部分非常详细地介绍倒排表的工作原理，是为了确保那些没有信息检索相关背景的选手通过阅读题目，也可以设计相应算法和数据结构满足题目的要求。限于篇幅我们在这里不译出题目原文，直接进入实现环节。

这里不得不提到：今天的 ACM/ICPC 竞赛所考察的不再是单纯的算法，而且还包括对手中工具的熟悉程度和灵活运用的能力。比如这道题目对于熟悉 Java 的选手来说，所需要的编码量会略小于使用 C++的选手，而如果用纯 C(@_@)来实现对于比赛时间来说几乎是 mission impossible。下面我们针对 C++和 Java 语言给出题目的求解框架。

我们的第一感觉是需要用三元组<keyword,article,page>来描述倒排表中的项，并且需要针对 keyword 构造基于 Hash Table 或者 Search Tree 的索引。前者的定位时间复杂度是 O(1)而后者是 O(log n)，但是根据题目规模，我们估计 O(log n)应该不会超时。对于熟悉 STL 或者 Java 类库的选手，他们可以选择一种合适的内置数据结构完成索引(i.e. std::map 或者 HashTable)。对于平常不那么用心的选手，之好亲力亲为了，当然两者消耗的比赛时间和出错的风险不可同日而语。

完成了索引之后我们事实上已经解决了单个 keyword 的检索以及 NOT keyword 的情形。针对 AND 和 OR 两种逻辑操作，我们还需要对两个 keyword 的查询结果作 join 操作，简单地说就是要排序去重。再一次地，两种语言中都有一些内置的 shortcut 可以偷一下懒 :)

【全文完·责任编辑：xiong@hitbbs】

[27] 刘禹，哈尔滨工业大学计算机科学与技术学院硕士研究生。电子邮件：pamws@hit.edu.cn

ACM ICPC Beijing Regional Contest 2004

C 题 Argus 解题报告

Written by Xiao Ying[28]

C 题应该是最简单的题目。

题大概意思是对一些定时查询返回结果。

做题的第一件事情一般都是估计复杂度，题目中说明，Instruction 最多 1000 个，查询最多 10000 次，而每个 period 的最大值是 3000。

设一共有 N 个 instruction，其中 period 最大为 P，查询 K 次。

那么最坏的情况，可以在 K * P / N 个时间间隔内完成，如果每个时间间隔对 N 个 instruction 进行检验，看是否到周期了，那么最多需要 （K* P / N）* N = K* P 次检验。

题中,K * P 的最大值就是 3 * 10 ^ 7，现在一般的处理器能算不低于 10^8 次 for 循环每秒，所以，估计用简单的穷举，在 1 秒内可以得到答案。

【全文完 • 责任编辑：xiong@hitbbs】

[28] 肖颖，哈尔滨工业大学计算机科学与技术学院硕士研究生。曾获 ACM 竞赛北京赛区第八名。电子邮件：xiaoying@hit.edu.cn

ACM ICPC Beijing Regional Contest 2004

H 题 The Separator in Grid 解题报告

Written by L[29]

问题描述：

一个棋盘由白色棋子（W），黑色棋子（B）和墙（S）组成。白色棋子和黑色棋子都是连通的，墙从中间隔开了白色棋子和黑色棋子，使任何一个白色棋子和任何一个黑色棋子都不相连（这里的相连是指：一个棋子和它周围的八个棋子都算相连）。墙也是连通的，白色棋子在墙的左边，黑色棋子在墙的右面，如图 1 所示。

现在给定一个这样的棋盘，要求只使用以下两种操作，使墙占有的方格数变为最少：

1. 将一部分黑色棋子变为墙。
2. 将一部分原来的墙（不包括由黑色棋子变来的墙）变成白色棋子。

要求变换后，W，B 和 S 仍满足上述条件，即 W，B，S 都是连通的，但任何一个 W 和任何一个 B 不相连。

在图 1 中，将黑色棋子 20，14 变成墙，再将 25，19，13，7 变成白色棋子，即可使墙的格数最少，如图 2 所示。

题目还有一个条件：题目中给出的 S 都是极小的，即它的任何一个子集都不能起到 S 的作用（即不能隔开 W 和 B）。

要求输出最优的 S 的格数。

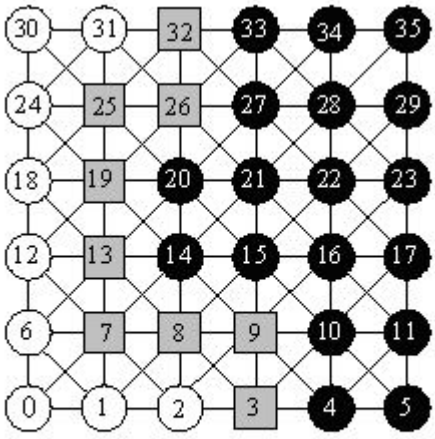


Figure-1. Partition (S, W, B)

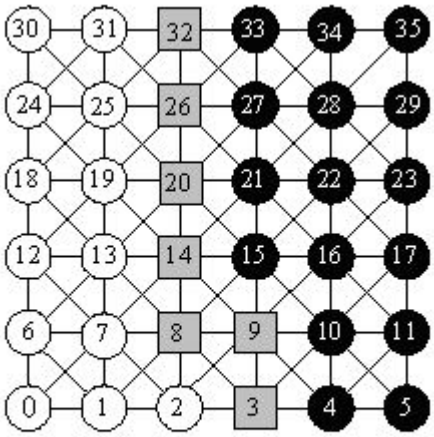


Figure-2. New Partition

解题思路：

“题目中给出的 S 都是极小的”是个最重要，最有用的条件。由于有了这个条件，因此 input 中给出的 S 一定是横竖交替地从上走到下，不存在别的可能，因为经过简单的分析可以知道，任何其他形状的 S 要么无法隔开 W 和 B，要么不是极小的。

由于这道题的约束条件，这道题可以从上到下用贪心的方法直接构造最优的 S。具体的方法是，从上到下跟踪 S 的轨迹，若发现 S 向左转（即向 W 的方向转），则让 S 少转一格。若发现 S 一直向下走或向右转，则保持 S 不变。这样做的原因是，如果 S 向左转，那么让 S

[29] 刘子阳，哈尔滨工业大学实验学院学生，哈尔滨工业大学 ACM/ICPC 代表队成员。电子邮件：ziyang@hit.edu.cn

少转一格，让紧贴 S 右侧的 B 变成 S，并将原来的 S 变成 W，这样可以使 S 的总数减少，并且符合题目的要求。但 S 若向右转，则无法减少 S 的数量，因为 W 不能变成 S，S 也不能变成 B。

这个算法实现起来相当容易，只需在每一行记录优化后的 S 的位置即可。如果一行有一个以上的 S，则记录最后一个，即转向下一行的那个。然后以这个位置为基准，判断下一行是向左转，一直走还是向右转。就这样不断地走下去，直到最后一行，就可以构造出最优的 S 来。在构造的同时统计 S 的格数（当然也可以最后统计）即可。时间复杂度为 O（r*c）。(r 是棋盘的行数，c 是列数)

这道题有一个特别需要注意的地方：如果 S 一开始先直行几格然后向右转，那么要将最开始直行的那几格向右移一格，这样可以使 S 的总数减少 1。这是个比较隐蔽的陷阱，我当时就是因为没考虑到这个因素而 WA（编者注：即 Wrong Answer）了好几次。

【全文完 • 责任编辑：xiong@hitbbs】

关于将本刊改为双月刊的决定

本刊近来稿件容量较大，校对工作量也越来越大，为了保证本刊质量，本刊编辑部经讨论决定从下一期（总第 3 期）开始，将本刊由原来的月刊改为双月刊，每逢单月的 28 号出版，以期有更多的时间准备稿件。稿件质量一直被本刊视为生命线。

下一期（总第三期）将于 2005 年 1 月 28 日发行，欢迎各位作者踊跃投稿！

《纯 C 论坛 • 电子杂志》编辑部

2004.11.28

投稿指南

非常高兴而隆重的宣布，在大家顶力支持及热情支援下，《纯 C 论坛·电子杂志》编辑部于公元 2004 年 9 月 28 日正式成立了！本刊定位为相对底层而纯粹的计算机科学及技术研究，着眼于各专业方向基本理论，基本原理的研究，重视基础，兼顾应用技术，以此形成本刊独特的技术风格。

本刊将于单月 28 号（双月刊）通过网络发行，任何人均可在这一天从纯 C 论坛网站上（http://purec.binghua.com）下载本刊，并将通过电子邮件，向纯 C 论坛的注册用户寄送。

目前本刊有十大骨干技术版块：

栏目	责任编辑	投稿信箱
计算机组成原理及体系结构	kylix@hitbbs	purec_coa@126.com
编译原理	worldguy@hitbbs	purec_compile@126.com
算法理论与数据结构	xiong@hitbbs	purec_algorithm@126.com
C 与 C++语言	sun@hitbbs，hitool@hitbbs	purec_cpp@126.com
汇编语言	ogg@hitbbs	purec_asm@126.com
数据库原理	pineapple@hitbbs	purec_db@126.com
网络与信息安全	true@hitbbs	purec_network@126.com
计算机病毒	swordlea@hitbbs	purec_virus@126.com
人工智能及信息处理	car@hitbbs	purec_ai@126.com
操作系统	iamxiaohan@hitbbs	purec_os@126.com

从现在开始，本刊面向全国、全网络公开征集各类稿件，你的投稿将由本刊各栏目的责任编辑进行审校，对于每一稿件我们都会认真处理，并及时通知您是否选用，或者由各位责任编辑对稿件进行点评。

所有被本刊选用的稿件，或者暂不适合通过电子杂志发表的稿件，将会在纯 C 论坛网站上同期发表。所有稿件版权完全属于各作者自己所有，非常欢迎您积极向本刊投搞，让你的工作被更多的人知道，让自己同更多的人交流、探讨、学习、进步！

为了确保本刊质量，保证本刊的技术含量，本刊对稿件有如下一些基本要求：

1. 主要以原创（包括翻译外文文献）为主，可以不需要有很强的创新性，但要求有一定的技术含量，注重从原理入手，依据原理解决问题。描述的问题可以很小但细致，可以很泛但全面，最好图文并貌，投稿以 word 格式发往各栏目的投稿信箱，或直接与各栏目责任编辑联系。本刊各栏目均为活动性栏目，会随时依据稿件情况新开或暂定各栏目，因此，只要符合本刊采稿宗旨的稿件，本刊都非常欢迎，如果您一时拿不准您的稿件应投往哪一栏目可直接将稿件投到本刊通用联系信箱（purec@126.com）。

2. 本刊对稿件的风格或格式没有特殊要求，注重质量而非形式，版权归各作者所有，不限一稿多投但限制重复性投稿（如果稿件没有在本刊发表，则不算重复性投稿）。稿件的文字、风格除了在排版时会根据需要有必要的改动及改正错别字外，不会对稿件的描述风格、观点、内容、格式进行大的改动，以期最大限度的保留各作者原汁原味的行文风格，因此，各作者在投稿时最好按自己意愿自行排版。也可以到本刊网站（http://purec.binghua.com）下载稿件模板。

3. 来稿中如有代码实现，在投稿时最好附带源代码及可执行文件，本刊每期发行时，除了一个 pdf 格式的杂志外，还会附带一个压缩文件，其中将包含本期所有的源代码及相应资源。当然，提不提供源代码，随各位作者自便。

4. 如果稿件是翻译稿件，请最好附带英文原文，以便校对。另外本刊在刊发翻译稿的同时，如有可能，将随稿刊发英文原文，因此请在投稿前确认好版权问题。

5. 来稿请明确注明姓名、电子邮箱、作者单位等信息，以便于编辑及时与各位作者交流，发送改稿意见，选用通知及寄送样刊等，如果你愿意在发表你稿件的同时，提供一小段的作者简介，我们非常欢迎。

6. 所有来稿的版权归各作者所有，也由各作者负责，切勿抄袭，如果在文中有直接引用他人观点结论及成果的地方，请一定在参考文献中说明。每篇稿件最好提供一个简介及几个关键词，以方便读者阅读及查询。由于本刊有可能被一些海外朋友阅读，所以非常推荐您提供英文摘要。

7. 所有来稿编辑部在处理后，会每稿必复，如果您长时间没有收到编辑部的消息，请您同本编辑部联系。

8. 由于本刊是纯公益性质，没有任何外来经济支持，所有编辑均是无偿劳动，因此，本刊暂无法向您支付稿筹，但在适当的时候，本刊会向各位优秀的作者赠送《纯 C 论坛资料（光盘版）》，以感谢各位作者对本刊支持！

9. 如果您想转载(仅限于网络)本刊作品请注明原作者及出处，如果您想出版本刊作品，请您与本刊编辑部联系。

本刊联系地址：

网　　址：http://purec.binghua.com

联系信箱：purec@126.com

通信地址：哈尔滨工业大学 计算机科学与技术学院 综合楼 520 室

邮　　编：150001

再一次诚恳邀请您加盟本刊，为本刊投稿！

《纯 C 论坛·电子杂志》编辑部

2004.11.28.修订

读者俱乐部

1. 本刊在纯 C 论坛网站上建立了读者俱乐部（由纯 C 论坛→BBS 讨论区→读者俱乐部进入），各位读者可以在上面发表对每期刊物的看法及建议，包括对刊物稿件质量、内容，刊物风格、格式等各方面的建议，对于热心读者我们会邀请他成为本刊特邀请评刊员，并在适当的时候赠送《纯 C 论坛资料（光盘版）》以示感谢！
2. 读者俱乐部中还会对每期刊物所刊发稿件的内容进行跟踪，因此，您可以在上面取得每期刊物最新的勘误表。

欢迎您访问“纯 C 论坛·电子杂志”读者俱乐部！

<<电子杂志·第一期>>勘误！！！

页：30

第二段第一行：E=1-(2^e-1) 应改为 E=1-(2^(e-1)-1)

最后一段第二行：E=E'-(2^e-1) 应改为 E=E'-(2^(e-1)-1)

[感谢 薛风(xf_cau@163.com) 指正]

页:31

第五行：“同样，我们通过一个程序验证一下，不过这次我们把这个程序变一下，直接输入 0.625”其中的“0.625”应改为“-0.625”

[感谢 timw 指正]

页：32

第三段第二行最后：＂最小正数数＂应改为＂最小正数为＂

页:43

图 5 下面第 6 行“于是 CPU 会产生一个所谓的缺页中断来通知操作系统进行处理，操作系统相应这个中断”中“相应”改为“响应”

[感谢 CrazyWind 指正]

页：46

第三行开头两个字：“所为”应改为“所谓”

[感谢 CrazyWind 指正]

图九上面倒数 4 行：“我们的“段选择子”为：0000 0000 0000 10000 ”其中段选择子的最后一个 0 应去掉

[感谢 CrazyWind 指正]

页:47 页

图十下面，“图中的数据存放区用来存发在……” 其中的“存发”应改“存放”

[感谢 CrazyWind 指正]

【本期信息汇总】

期数：2004 年第 11 期（总第 2 期）
发行时间：2004 年 11 月 28 日
修订时间：2005 年 1 月 13 日(SP1)
本期责任编辑：iamxiaohan@hitbbs
本期特邀评刊员：cliff@hitbbs，ssos@hitbbs
主题文章数：10 篇
总页数：106
总字数：72,000

SP1 版对原版的修订说明

1. 修订了如下错误（注：页号均为原版页号）：

20 页

“1.1 CPU”一节，中文翻译第三行最后的“数量流”，改为“数据流”

[感谢 yuanlaishini 指正]

21 页

第三段第一行“微处理器可以完成算术运术”改成“微处理器可以完成算术运算”

[感谢 yuanlaishini 指正]

25 页

“1.5 地址空间”一节，中文翻译第五行，“CPU 可以既可以存取系统存储空间”中“CPU”后面的“可以”两字去掉。

[感谢 yuanlaishini 指正]

26 页

“1.6 定时器”一节，中文翻译倒数第 2 行，“日期及日间”改成“日期及时间”

[感谢 yuanlaishini 指正]

37 页

第一段最后一行“尽可能避免淘汰出下闪马上要用的数据结构”中的“闪”改成“次”

[感谢 yuanlaishini 指正]

63 页中部：

现在再来看看优先级控制命令。
当 R 为 0 时，表明这是一个固定优先权方式，IRQ0 最高，IRQ7 最低。
当 R 为 1 时，表明这是一个循环优先权，比如，如指定 IRQ2 最低，则优先级顺序就为：
IRQ2 < IRQ3 < IRQ4 < …… < IRQ 7 < IRQ0 < IRQ1
也即，如果 IRQ(i)最低，那么 IRQ(i + 1)就最高。

其中红色部份应改为：

IRQ2<IRQ1<IRQ0<IRQ7<IRQ6<IRQ5<IRQ4<IRQ3
（编者注：可以将其记忆为“IRQ7<IRQ6<IRQ5<IRQ4<IRQ3<IRQ2<IRQ1<IRQ0”进行循环左移后的结果）

[感谢 uty 指正]