eMag
gazines form Csdn
Csdn.net 社区电子杂志

纯C论坛杂志

2005 年 3 月第 2 期 总第 4 期

伤心糊涂

再哀水木

我眼中的计算机科学

哈尔滨工业大学 计算机科学与技术学院
王宏志

20 世纪，计算机科学制造了一场触及每一个人灵魂的大革命，它不仅仅带来了新的工业的革命，而且对科学，文化，经济，政治等人类生活的方方面面都产生了革命性的影响。

计算机科学是一门自然科学学科，它有着探索未知的任务。从休漠到弗洛伊德到克里克，哲学家和科学家们对人的意识或者说灵魂在各个层面上进行了探索。一个颇为遗憾的事情是，在‘灵魂科学’中，得到切实可信的实验结果是非常困难的，毕竟用精神健全的人来进行可能具有破坏性的实验是文明社会不允许的。而计算机科学提供了这样的机会。尽管现在计算机科学及其支撑学科(比如微电子，材料科学) 的发展尚不足以使计算机具有足够的智能，但其毕竟在某种程度上对思维进行了模拟，可以看做是灵魂的镜子。可以预见，如果某一天脑科学和计算机科学实现了接轨，对人类的科学将会产生本质的影响。

计算机科学不仅仅是一门科学，因为其还需要回答如何创造新世界的问题，而这是工程学科的任务。而和其他工程学科不同，计算机科学的研究对象是看不见，摸不到的，它的产品也是如此。正因为此，计算机科学对生产方式也产生了影响。计算机科学是为数不多的自带对生产组织方式研究的工程学科，这个研究分支就是我们熟悉的软件工程。所谓智力产品的概念，是计算机科学带来的。可以说，如果有一天，人类直接生产的产品都是纯粹的智力产品，这才算实现了“人有人的用途”。这并非是痴人说梦，现在，连接 UG 软件的生产中心号称已经能够从产品的设计的图纸自动生产出产品。当然，当前这种生产方式仅仅限制在一个非常小的范围之内，但也足以使我们看到其耀眼的光芒。

当计算机完全改变了每个人的生活的时候，它就为人类社会带来了革命。智慧可以直接的成为产品；信息这种抽象的东西变得可以象任何具体产品一样出售。不论如何鄙视“形而下者”的社会科学学者都无法否认计算机对人类社会产生的影响。一种新的产业叫做 IT，一个新的人群叫网虫。新概念，新思想爆炸式的产生和传播。庄生晓梦迷蝴蝶，在网络和游戏中，人们可以找到另一个世界，另一个自己。而这一切革命的来源，是计算机科学的发展。技术与社会的互动达到了前所未有的程度，看看我们的生活被 BBS, QQ 和 BLOG 改变了多少就知道了。

尽管计算机科学对这个世界产生的如此之大的影响，它仍然是一个在襁褓中的学科，充满了未知和挑战。从 NP 问题到图灵测试，计算机科学的许许多多根本的问题尚未解决，与此同时人类社会对计算机提出了越来越高的要求。60 年前图灵提出的计算模型是否是唯一的？计算机能否具有智能？计算机的计算能力是否具有极限？计算机对人会产生何种程度的异化？坐在计算机面前，一个又一个难题摆在我们面前。让计算机具有更快的计算能力，支持更大规模信息的检索，创造更多的功能，这些是社会对计算机科学要求。

从祖师图灵算起，计算机科学的历史充满了英雄和天才，计算机科学的天空群星璀璨，如我这般萤火之光或许默默的湮没在其中。尽管如此，作为以计算机科学为事业和职业的人，我为能够和如此之多的聪明人并肩作战感到幸运，我为能够在计算机科学为人类铺就的未来之路中做一枚小小的石子感到骄傲。

在计算机科学为我们创造的世界中，每个人都有发表自己观点的权力和场所，具有相同的志趣灵魂可以自由的组成特别的社区。纯 C 论坛就是热爱计算机科学的人　构成的特别的社区，相信打开这本杂志的人对计算机科学都有着同样的热爱，愿我们的智慧在其中碰撞出闪亮的火花。星星之火，可以燎原，今天的一颗火花说不定就是明天计算机科学天空中一颗耀眼的明星。

王宏志：1978 年出生，博士研究生。曾参加过 ACM/ICPC，发表过多篇论文。

目录

以此期纪念“水木清华”BBS

纯 C 论坛杂志
2005 年 3 月第 2 期(总第 4 期)

纯 C 论坛
CSDN 电子杂志社区 合作

本期主编： worldguy
编　　辑：
Sun, kylix, iamxiaohan, pineapple, worldguy, xiong, true, hitool, ogg

特邀编辑： 张华

论坛：
http://purec.binghua.com

投稿信箱：
purec@126.com

注: 封面图片取自网络。感谢作者的辛勤劳动!

听大牛们谈未来的体系结构研究方向
（二）

哈尔滨工业大学并行计算实验室
王凯峰（译）

Guri Sohi, University of Wisconsin, Madison

微处理器研究领域的著名教授，世界级著名学者，目前很多活跃在学术界和工业界的著名 Architecturer 都出自其门下，其主要成就：1）80 年代中期，在人们把主要精力放在研究按序提交模型时，较早的较系统地提出了乱序执行模型（超标量）；2）提出非阻塞 Cache 模型，大大提高了 Cache 的带宽，该技术已经被目前绝大多数商用处理器采用；3）90 年代初期，在人们研究乱序执行结构时，提出了 Multiscalar 项目和线索级前瞻概念，线索级前瞻可以说是目前最热地研究方向之一；4）存储器相关预测的提出和早期研究者，该技术在 Alpha 处理器被采用；5）指令重用的早期提出者，目前也是十分热地研究方向；6）他领导的研究组是学术界使用的十分广泛的超标量模拟器 Simplescalar 的作者。

当我们预测未来 20 年内计算机体系结构的研究方向时，看看过去所走过的路将是十分有必要的。在我们这个领域中的那个成为很多革新技术的基本原理也适用于这里，那就是通过历史的信息预测未来。回首过去，我们发现一个事实，那就是过去的历史中充斥着很多后来被证明是错误的观点。有些观点强烈建议某些研究方向应该紧跟（80 年代的 AI 热潮）或者某些研究方向尽量不要做（80 年代的单芯片处理器结构）。虽然这些观点作为参考意见来讲，对错都无所谓，但是如果因为他们而使得整个研究领域的研究方向单一化或者影响了某些研究活动的话，就将十分危险。对于整个研究领域来讲，只追随某几个研究方向将是十分有害的。我们应该不断的扩大我们的“基因库”，应该有多种多样的研究议程和方向来不断的巩固加强整个研究领域。无意义的研究方向会自然的被淘汰，但是研究方向是否有意义却很难事先确定。因此，我觉得我们不应该向研究领域中的研究团体给出这样的建议，像哪个研究方向尤其重要啦（例如功耗问题）或者哪个方向不要再搞啦（像单线索高性能技术）。而应该不断地创建出新的研究方向并且培养出一种锐意进取，勇于革新的研究氛围。

另一个发现的事实是，过去涌现出了众多的革新技术并且严重的影响着计算机体系结构的发展。比别的研究领域幸运的是，计算机体系结构不仅由于创新技术的提出自身发展起来，而且他们还深刻的影响着工业界如何来制造计算机。为什么会如此呢？我认为我们的成功之处主要源自于我们能够以新颖的方式对技术方面的进步产生回应。我们能够判断出究竟什么时候技术进步到达了一定水平，使得可以采用新的方法。我们利用这些机会开辟出不同的和新的方向。一旦门被打开，某个巨大的革新就会出现，使得我们的研究活动向前走一段时间，而这些研究活动产生的结果远远超出了当初人们的想象。

过去 25 年间在单芯片处理器方面的革新就是一个很好的例子。在 80 年代早期，通过对 RISC 结构的研究，我们发现当时的技术已经允许我们将整个处理器集成到一块芯片上，这将改变处理器微结构设计中的一些规则。这一发现直接产生了很多的革新思想，不仅仅在计算机体系结构领域里而且影响到了计算机系统中相关的其他领域。80 年代晚期曾有一小段时间停滞，很多人认为这方面的研究可能已经到头了，可是后来又出现了前瞻技术和乱序执行技术，并且技术的进步也允许我们将这两种思想付诸实现。当前，这股热潮仍在继续，正如我们现在所看到的，各种各样的前瞻技术方面的革新仍然不断涌现。产生的最直接的结果就是今天的高性能

单片微处理器已经和 25 年前的截然不同了。大量涌现的革新技术使我们可以追逐我们的主要目标，那就是不断增长的对高性能的需求。

那么未来我们要去往何处？怎样才能创造出另一个巨大的革新呢？我认为我们应该提出更多新的研究方向，这将为我们的研究领域注入新的思想。紧记这一点，我认为一个很可能取得成功的研究方向就是我们现在应该重新考虑如何构建单片上多处理器，因为目前的技术已经允许我们在单块芯片上构造多个处理器。我将使用“多微结构”（multiarchitecture）这个词来指单片多处理器的微结构或组成。目前，我们可以在单块芯片上构建一个小规模的多处理器结构，但是目前的多微结构和传统的使用多块芯片构建多处理器的方法基本相同。这种情况需要改变。在单块芯片上，我们可以构建不同的多微结构，例如原来在多块芯片间某些信息难以快速有效的传送，但如果在一块芯片中，就将十分容易。技术的进步将使我们发明出完全不同的实现多微结构的方法，我们也将紧跟着革新技术来实现新型的多微结构。我认为将来，在如何设计多处理器的问题上我们需要更多新的思想。

即使我们采用革新技术构造出了多处理器结构，但只有当我们使用正确的方法来利用这些处理能力的时候，才算成功。那么，怎样才能充分利用已有的处理能力呢？我们使用这些并行处理能力来做什么呢？这里，我将提出两个提议，作为进行深入探讨的起点。第一，通过并行处理取得高性能是大势所趋，但是，可能对并行处理模型的使用方法将与现在的手段大相径庭。我认为，目前程序员们之所以不愿意写并行程序不仅仅是因为并行编程难度很大，还因为很难看到显著的性能提升。当程序的不同部分之间需要进行信息交换，但这些程序却运行在不同的芯片上时，基本上很难看到性能提升；一般来讲，并行化反而会导致性能下降！我们的新多微结构必须要采用新的方法克服这些阻碍性能提升的问题。在单片多处理器之间进行通信将使得通信延迟大大减少从而提高整体性能。我们还需要不同的并行执行模型，尤其是在单芯片或以小组芯片上的并行执行时，这时候并行执行模型可能需要和底层的多微结构紧密地结合。我们的多微结构还需要一些能够辅助程序并行化和简化程序员并行编程时工作量的一些新的特征。

第二，我们必须使用新奇的方法充分利用多余的并行处理能力，为应用程序软件提供更多可用的功能。我们需要鼓励程序员们，不仅仅要写完成某种功能所必需的代码，而且在不影响函数功能的同时还要写一些附加代码。这些附加代码可以用来监视，纠错，提高安全性，提高可靠性等等。如果增加这样的代码对程序来讲基本上不影响性能并且编程也不复杂的话，程序员们很可能会愿意写这些附加的代码。我们需要新的处理模型包括编程结构，编译器以及新的多微结构，如果成功的话，程序员们会逐渐的为软件增加功能，反过来又会推动我们在单块芯片上集成更多的处理器。

Mateo Valero, Universidad Politecnica de Catalunya

西班牙籍国际著名教授，曾在巴塞罗那负责创建并主持欧洲并行计算中心，曾作过西班牙 IBM 研究中心主任；2004 年开始，成为西班牙巴塞罗那超级计算中心主任。

在设计未来微处理器结构时我们面临很多挑战。其中一些是新的挑战，另外一些是比较老的问题。其中一个较老的问题就是内存壁垒问题（memory wall）。随着技术的进步，这个问题愈来愈严重。我们已经花费了很大的精力来尽量减少由于内存延迟的增加而给单处理器或多处理器的 IPC 带来的影响。传统的方法，像更精妙的 Cache 设计，较好的硬件/软件预取算法仍然值得我们进一步关注。较新的技术，如帮助线程（helping threads）将可能是一个比较聪明的解决方案，但是可能需要浪费更多的能量。

一种很有希望的解决超长内存访问延迟（Cache 实效，需要访问内存）的方法是增加在乱序执行处理器中可同时执行的指令数目。我们提出的千条指令处理器（thousands of in-flight instructions）。对于数值计算类型的应用程序来讲，如果 in-flight 指令数目增加的足够多的话，那么内存延迟将对处理器的效率几乎不产生影响。在开发允许大量 in-flight 指令的乱序执行/按序提交的处理器的研究中，像一些关键资源，如：寄存器文件，重排序缓冲和指令队列实际上他们的利用率越来越低。我们提议只对其中一些或某些特殊指令（某些 Loads 和 Branches）进行检查点纪录，以尽量减少在体系结构中需要管理的关键资源的数量。这种检查点机制使得：是乱序提交机制易于实现（实际上，使用了检查点以后，传统的 ROB 将没必要存在），以一种更加激进的方式释放资源（例如，寄存器的延迟分配和尽早回收技术的结合），以更加明智的管理机制来管理指令队列。

对于整型应用程序来讲，有另外两个影响处理器性能的主要因素：跳转误预测和 Load 相关（例如：指针追逐现象）。我们认为设计千条指令处理器将会开启新的研究课题并且某些古老的问题也会被重新提出来加以考虑：

我们需要研究当出现这种令人讨厌的 Load 问题时，如果相同的指令流曾经出现多次的话，应该使用新的方法来重用这些指令。

我们需要继续研究跳转预测技术，尤其对于整型应用程序来讲，可同时执行的指令数越多，跳转预测的精度就显得越发重要。

我们需要重新考虑谓词执行和多路径执行技术。虽然在某些只具有有限的 ROB 资源的处理器中，这些技术并没产生什么令人印象深刻的结果。但是，如果把它放在允许上千条指令同时运行的处理器结构中，将会完全不同，有可能产生良好的效果。

【责任编辑：kylix@hitbbs】

操作系统概念（第六版）

第三章 操作系统结构

Abraham Silberschatz，Peter Baer Gkdalvin，GreGangne 著
吕建鹏(sweeil@yahoo.com.cn) 译

操作系统提供了程序执行的环境。各种操作系统的内部构成相当不同，它们沿着一些不同的思路组织。新的操作系统的设计是重大的工作（major task）。设计之前必须要明确定义系统目标，针对所期望的系统类型选择算法和策略。

可以从几个有利的位置来观察一个操作系统。首先是分析它所提供的服务。其次是分析它向用户和程序员提供的接口。最后是分析系统组件和这些组件之间的联系。本章我们将研究操作系统的这三个特征，并展现用户、程序员以及操作系统设计者的观点。我们要考虑：操作系统提供了什么样的服务、它是怎样提供这些服务的，以及设计这样的系统需要什么样的方法学理论。

3.1 系统组件

构造像操作系统这样大型的复杂的系统就必须要把它划分成许多小块（piece，可以理解为子系统）。每一小块必须是系统某一部分的详细定义，包括输入、输出和功能。显然，不是所有的系统都有着同样的结构。然而许多现代操作系统有着同样的目标，就是支持从 3.1.1 节到 3.1.8 节所列出的系统组件。

3.1.1 进程管理

程序本身并不能做什么，只有在 CPU 执行它的指令时才能有所作为。可以把**进程**看作是正在运行的程序，但是当我们进一步研究时，对进程的定义将更为普遍。一个分时用户程序（如编译器）是一个进程。个人用户在 PC 上运行的字处理程序是一个进程。一个系统任务（如输出到打印机）也是一个进程。现在，我们认为进程是一个作业或分时程序，但是，稍后你将明白这个概念更加普遍。在第四章中将会看到我们可以提供允许进程创建与其并发执行的子进程的系统调用。

进程需要特定的资源（包括 CPU 时间、内存、文件和 I/O 设备）来完成工作。这些资源或者在进程创建时分配给它，或者在其运行时。除了在进程创建时所获得的各种物理资源和逻辑资源以外，各种各样的初始化数据（或输入）也可能一同传送给进程。例如，考虑一个能够在终端的显示屏上显示一个文件状态的进程。这个进程将获得包含输入的文件名，并且将执行相应的指令和系统调用来获取所期望的信息并显示在终端上。

我们着重强调程序本身不是进程；程序是静态实体（passive entity）（像是存储在磁盘中的文件的内容），而进程是动态实体（active entity），它用一个**程序计数器**来指明要执行的下一条指令。进程必须要按顺序执行。CPU 执行完进程的一条指令后再执行下一条，直到进程结束。更进一步讲，一次最多执行一条代表该进程的指令。这样，从来就不会出现两个独立运行的序列。一个程序在运行时创建多个进程是非常普遍的。

在系统中，进程是工作单元。这样的系统由进程集合构成，有些是操作系统进程（那些执行的系统代码），其它的是用户进程（那些执行的用户代码）。通过对 CPU 的多路复用，所有的这些进程可以被并发执行。

操作系统要负责下列与进程管理相关的工作：

- 创建和撤销用户及系统进程
- 挂起和恢复进程
- 提供进程同步机制
- 提供进程通信机制
- 提供死锁处理机制

我们将在第四章到第七章中讨论进程管理。

3.1.2 主存储器管理

就像我们在第一章中所讨论的，主存储器是现代计算机系统运行的核心。主存储器是由字或字节组成的大型队列，容量在数十万到数十亿之间。每个字或字节都有它自己的地址。主存储器是 CPU 和 I/O 设备共享的大容量快速存储器。中央处理器在取指令周期中从主存储器中读取指令，而且在取数据周期中从主存储器中读写数据。通过 DMA，I/O 操作也实现了对主存储器的数据读写。通常主存储器是 CPU 唯一能够直接寻址和访问的大容量存储空间。例如，CPU 要处理磁盘中的数据，那么 CPU 首先发出 I/O 调用将这些数据传送到主存储器中。同样，指令必须在存储器中才能够由 CPU 执行。

必须要把程序映射到绝对地址并载入内存中才可以执行。在程序运行时，它通过产生绝对地址来从内存中访问程序指令和数据。最后，程序结束，它将释放所占内存空间，下一个程序能够被载入并执行。

为了提高 CPU 利用率和计算机响应速度，我们必须在内存中保留多个程序。有许多不同的内存管理策略，而且不同算法的效率取决于具体的环境。为具体的系统选择内存管理策略要考虑到许多因素——尤其是系统的硬件设计。每种算法都需要自己的硬件支持。

操作系统要负责下列与内存管理相关的工作：

- 跟踪内存使用情况，明确哪一部分正在使用和为谁所用
- 在内存空间有效时决定将哪个进程载入内存
- 根据需要分配和释放内存空间

将在第九章和第十章中讨论内存管理技术。

3.1.3 文件管理

文件管理是操作系统中可视性最强的组件之一。计算机能够将数据存储在各种类型的物理介质上。磁带、磁盘和光盘是最常用的介质。每种介质都有自己的特性和物理结构。每个存储媒体由一个驱动器控制（如磁盘驱动器或磁带驱动器），这种驱动器也有自己的独有特性。这些特性包括访问速度、容量、数据传输率和存取方式（顺序的或随机的）。

为了便于使用计算机系统，操作系统提供了一个计算机系统的整体逻辑层面。操作系统把存储设备的物理属性抽象定义为一个逻辑存储单元——文件。文件被映象到物理媒介中，通过存储设备来访问这些文件。

文件是由其创建者定义的相关信息的集合。一般的文件表现为程序（源程序和目标代码）和数据。数据文件可能是数字的、字母的或二者混合的。文件可能是形式自由的（如文本文件），也可能有严格定义的格式（如固定字段）。(Files may be free-form (for example, text files), or may be formatted rigidly (for example, fixed fields).) 由字、字节、行或记录组成的文件结构是其创建者定义的。文件概念具有相当广泛的含义。

操作系统通过管理大量存储体（如由驱动器控制的磁盘和磁带）实现了文件的抽象概念。另外，为了更简易的使用文件，通常将他们组织到目录中。最后，如果有多个用户访问文件，我们可能需要控制谁以什么样的方式访问（例如：读、写、追加）。

操作系统要负责下列与文件管理相关的工作：

- 创建和删除文件
- 创建和删除目录
- 将文件映象到辅助存储器中
- 将文件备份到永久（非易失性）存储体中

我们将在第十一章和第十二章中讲述文件管理技术。

3.1.4 I/O 系统管理

操作系统的目的之一就是要向用户隐藏具体的硬件特性。例如，在 UNIX 中，通过 **I/O 子系统**向操作系统本身隐藏了 I/O 设备的特性。I/O 子系统由以下几个方面组成：

- 一个内存管理模块，这包括：**buffering**、**caching** 和 **spooling**
- 一个通用设备驱动程序接口
- 针对具体硬件设备的驱动程序

只有设备驱动程序了解所指定的具体设备特性。

在第二章中我们讨论了高效的 I/O 子系统结构是如何应用中断处理程序和设备驱动程序的。在十三章，我们将讨论 I/O 子系统怎样与其它的系统模块相连、怎样管理设备、怎样传输数据和怎样探测 I/O 操作结束。

3.1.5 辅助存储器管理

操作系统的主要目的是执行程序。这些程序在运行时（以及它们要访问的数据）都必须在**主存储器**中。因为主存储器的容量太小不能存储所有的程序和数据，而且掉电后会丢失所有的存储信息，所以计算机系统必须要提供**辅助存储器**作为主存储器的后备。大多数现代计算机系统使用磁盘作为存储程序和数据的主要联机存储体。大多数程序（包括编译程序、汇编程序、排列程序、编辑程序和格式化程序）在载入内存之前存储在磁盘上，并且在运行时利用磁盘存储它们所处理的源文件和目标文件。因此，合理的磁盘管理对一个计算机系统来说是至关重要的。

操作系统要负责下列与辅助存储器管理相关的工作：

- 空闲空间管理
- 空间分配
- 磁盘调度

因为频繁的使用辅助存储器，所以必须要能够高效运行。计算机的整体运行速度取决于磁盘子系统的速度和该子系统的实现算法的效率。将在第十四章讨论辅助存储器管理。

3.1.6 网络管理

分布式系统是一个处理机的集合，这些处理机既不共享内存和外围设备，也不共享时钟。而是每个处理机拥有自己的本地内存和时钟，并且这些处理机可以通过各种通信线路（如高速总线或网络）进行通信。一个分布式系统中的处理机在规模和功能上有所不同。其中可能包括小型微处理器、工作站、小型机和大型通用计算机。

在（分布式）系统中，处理机通过**通讯网络**相连接，有多种不同的方法可以配置该网络。这种网络可以完全或部分的连接。通信网络的设计必须要考虑到报文路由选择和连接策略，以及争用和安全的问题。

分布式系统将物理上相互独立的可能不同种类的系统集合成为一个独立相连的系统，向用户提供了访问由系统维护的各种资源的能力。对共享资源的访问加快了计算速度、增强了系统功能、提高了数据的可用性并加

强了可靠性。操作系统把网络细节包含在了网络接口设备驱动程序中，于是将网络访问泛化为一种文件访问的形式。（Operating systems usually generalize network access as a form of file access, with the details of networking being contained in the network interface's device driver.）分布式系统所使用的协议在很大程度上影响到系统的效用和普及。环球网的创新在于为信息共享提供了新的途径。它改进了现有的**文件传输协议（FTP）**和**网络文件系统（NFS）**，去掉了用户必须登陆才能访问远程资源的限制。环球网定义了一个新的协议——**超文本传输协议（http）**，它用于在 web 服务器和 web 浏览器之间进行通信。一个 web 浏览器只需要向远程机器的 web 服务器发送一个信息请求，然后获得返回信息（文本、图片、指向其它信息的链结）。这种在易用性上的提高促进了 http 和 Web 应用的快速成长。

我们将在第十五章到第十七章中讨论网络和分布式系统。

3.1.7 系统保护

如果一个计算机系统有多个用户并允许并行执行多个进程，那么必须要保护各个进程免受其它进程的侵扰。为此，需要提供一种机制来保证只有那些从操作系统获取了合适权限的进程能够操作文件、存储段、CPU 和其它资源。

例如，内存寻址硬件确保了一个进程只能在自己的地址空间内执行。计时器确保进程最终能够放弃对 CPU 的控制。用户不能够访问设备控制寄存器，这样就保护了各种外围设备的完整性。

保护是由操作系统定义的控制程序、进程或用户访问的机制。这个机制必须要提供一种方法来描述要施加的控制，以及强制执行的方法。

通过检测子系统接口中潜伏的错误，保护能够增强系统的可靠性。对接口错误的早期检测常常能够阻止一个子系统故障波及到其它正常的子系统。一个未受保护的资源难以防止未授权或不适当的用户的使用（或误用）。一个面向保护的系统提供了一种区分经授权的和未授权的使用的方法，这将在第十八章中讨论。

3.1.8 命令解释程序

命令解释程序是操作系统中最重要的系统程序之一，它是用户和操作系统间的接口。有些操作系统在内核中包含了命令解释程序。其它的操作系统（像 DOS 和 UNIX）把命令行解释器当作一个特殊的程序，（在分时系统中）当一个用户登陆到系统或初始化一个任务时，它就已经在运行了。

许多命令是通过**控制语句**提供给操作系统的。（Many commands are given to the operating system by control statements.）当批处理系统中的一个新作业开始时，或当一个用户登录到分时系统中时，一个读取并解释控制语句的程序会自动运行。这个程序有时被称为**控制卡片解释程序**或**命令行解释程序**，而且经常被称为 **shell**。它的功能很简单：获取下一条命令语句并执行。

通常操作系统在命令解释程序方面的差别很大，一个用户界面友好的命令解释程序会使操作系统更容易为一些用户所接受。用户界面友好的一个风格是用在 Macintosh 和 Microsoft Windows 中的基于鼠标的窗口菜单系统（mouse-based window-and-menu system）。可以移动鼠标的位置以指向图像、**图标**，或屏幕（屏幕用于显示程序、文件和系统功能）上。根据鼠标指针的定位，单击鼠标上的按钮可以执行一个程序、选择文件或目录（被称为**文件夹**）或弹出一个包含命令的菜单。有些用户喜欢功能更强大更复杂也更难掌握的命令解释程序。在这些命令解释程序中，通过键盘键入命令并在屏幕上显示或在终端上打印出来，然后通过“enter”（或“return”）键向命令解释程序发出信号，告知一条命令输入完成并等待运行。MS-DOS 或 UNIX 命令解释程序就是以这种方式执行。

命令语句本身负责进程的创建和管理、I/O 管理、辅助存储器管理、主存储器管理、文件系统访问、保护和网络。

3.2 操作系统服务

操作系统提供了程序运行的环境。它为程序和程序用户提供了特定的服务。当然，不同的操作系统提供的具体服务不同，但是我们能够找出其共同部分。提供的这些操作系统服务是为了便于程序员设计程序。

• **程序执行**：系统必须要能够将程序载入内存并运行它。程序必须能够正常的或异常的（指示错误）结束运行。

• **I/O 操作**：一个正在运行的程序可能要请求 I/O 操作。这可能会涉及到文件或 I/O 设备。针对具体的设备，需要特定的功能（如倒卷一个磁带驱动器或清空一个 CRT 屏幕显示）。出于系统效率和保护的原因，用户通常不能够直接控制 I/O 设备。因此，操作系统必须要提供一种 I/O 运行机制。

• **文件系统处理**：文件系统相当有意思。显然，程序需要能够读写文件，也要能够创建和删除文件。

• **通信**：在很多情况下，一个进程需要与另外一个进程交换信息。这种通信可以通过两种主要的方式。第一种方式是在运行在同一台计算机上的进程间通信；另外一种是在运行在由一个计算机网络连接的不同的计算机系统上的进程间通信。可以通过**共享存储器**或**报文传送**（这种方式中，操作系统在进程之间将信息打包移动）的方式实现进程间的通信。

• **错误检测**：操作系统需要经常注意可能发生的错误。错误可能发生在 CPU 和内存（如存储错误或电源故障）、I/O 设备（如磁带奇偶检验错误、连结网络失败或打印机缺纸）以及用户程序（如运算溢出、试图访问非法存储器地址或过多占用 CPU 时间）中。对于每种类型的错误，操作系统应该能够采取针对性措施以确保计算的正确性和相容性。

另外，存在着另外一些操作系统功能，它们不是为了帮助用户工作，而是为了确保系统本身的高效运行。通过在多个用户间共享计算机资源，多用户系统能够获得高运行效率。

• **资源分配**：当多个用户登录到系统中或同时有多个作业运行时，必须要在它们之间分配资源。操作系统管理许多不同类型的资源。有些资源（如 CPU 周期、主存储器和外存储器）可能有专用的分配代码，而其它的（如 I/O 设备）可能有更通用的请求和释放代码。例如，在决定如何最好的使用 CPU 的问题上，操作系统的 CPU 调度程序要考虑到 CPU 速度、必须要执行的作业、有效的寄存器数量和其它的一些因素。也可能有一些程序用于向作业分配磁带驱动器。一个这样的程序定位一个未用的磁带驱动器并将该设备的新用户记录在内部表中。其它的程序用于清空这个表。这些程序也能够分配绘图仪、调制解调器和其它外围设备。

• **账户管理**：我们希望跟踪记录每个用户使用哪些类型的计算机资源和用了多少。这个记录保持可能用于记账（以便用户付账）或简单的用于累加使用率统计。对于研究者来说，使用率统计可能是个有效的工具。利用使用率统计，研究者可以重新配制系统以改善计算服务。

• **保护**：存储在多用户计算机系统上的信息的所有者希望能够控制对该信息的使用。当多个不相关的进程并发执行时，一个进程不应该能够干扰其它进程或操作系统本身。保护包括了监控所有对系统资源的访问。对来自外界的系统**安全检测**（security）也是非常重要的。这种保护往往通过密码的方式，用户向系统验证口令才能访问资源。它也包括了保护外部 I/O 设备（包括调制解调器和网络适配器）免于非法的访问企图和记录所有这样的非法闯入。如果一个系统受到保护并且是安全的，那么就必须要建立预防措施。系统的安全强度只与其最薄弱的一环有关。（A chain is only as strong as its weakest link.）

3.3 系统调用

系统调用提供了进程和操作系统间的接口。这些调用通常是以汇编语言的形式，而且它们通常会在汇编语言程序员的使用手册中列出来。

某些系统允许高级语言直接进行系统调用，这通常是将系统调用预定义为函数或子程序调用。它们调用一个指定的运行时程序（run-time routine），这个程序完成系统调用，或者直接在代码里加入系统调用。

有些语言（如 C、C++和 Perl）已经取代了汇编语言进行系统程序设计。这些语言允许直接进行系统调用。例如，在 C 或 C++程序中可以直接调用 UNIX 系统调用。Microsoft Windows 平台中的系统调用是 Win32 **应用程序接口（API）**的一部分，Win32 API 可以用于所有针对 Microsoft Windows 的编译器。

作为一个使用系统调用的例子，考虑一个简单的程序，它从一个文件中读取数据并将数据拷贝到另外一个文件中。这个程序首先需要获得这两个文件的文件名：输入文件和输出文件。指定文件名的方法有很多种，这要看具体的操作系统。一种方法是由程序向用户询问这两个文件的名称。在交互式系统中，这种方法将请求一系列系统调用：首先在屏幕上写出提示信息，然后读取键盘的输入字符（这些字符指定了两个文件）。在基于鼠标的窗口菜单系统中，一个名为“文件”的菜单通常显示在窗口上。用户可以使用鼠标选择源文件名，并且能够打开一个类似的窗口用于指定目标文件。

一旦获得了这两个文件名，这个程序就要打开输入文件并创建输出文件。这些操作的每一步都需要请求另外的系统调用，并可能会出现错误。当这个程序尝试打开输入文件时，它可能会发现没有这个名字的文件存在或文件被访问保护。在这种情况下，程序必须在控制台上打印出一段消息（要用到另外的系统调用），或者我们删除现存的文件（也要用到另外的系统调用）并创建一个新文件（又要用到另外的系统调用）。在交互式系统中，程序会询问用户是替换现有文件还是终止程序。

现在两个文件都建立起来了，接下来，我们从输入文件中循环的读出数据（用到一个系统调用），然后写入输出文件（用到另一个系统调用）。每个读写操作必须要返回状态信息以便于考虑各种可能出现的错误。在输入过程中，程序可能会发现到达文件末端或在读取时发生了硬件故障（比如奇偶校验错误）。在输出过程中，依据不同的输出设备，可能会发生各种错误（比如，磁盘空间不足，磁带末端，打印机缺纸）。

最后，在完成对整个文件的拷贝之后，程序就需要关闭这两个文件，向控制台中写一段消息（需要另外的系统调用）并最终正常结束（最后的系统调用）。正如我们所看到的，即使是一个简单的程序也可能要调用大量的系统调用。

然而大多数用户从未见到过这些细节。运行时（run-time）支持系统（功能构件在库中，并且包含一个编译器）为高级语言提供了更为简洁的接口。例如，C++中的 cout 语句被编译为一条对 run-time 支持程序的调用，该程序进行必要的系统调用，检查错误并最终将结果返回给用户程序。这样，编译器和 run-time 支持包向程序员隐藏了操作系统接口的大部分细节。

根据对计算机的使用情况，有多种方式可以进行系统调用。在进行系统调用时通常需要更多的信息，而不仅仅是系统调用的标识。根据具体的操作系统和系统调用，所需的信息类型和数量不尽相同。例如，为了获取输入，我们可能需要指定文件或设备作为信息来源，并且要把地址和内存缓冲区的长度放到输入读取的位置。（For example, to get input, we may need to specify the file or device to use as the source, and the address and length of the memory buffer into which the input should be read.）当然，设备或文件以及长度可能隐含在调用中。

有三种通用的方法可以向操作系统中传送参数。最简单的方法是将参数传送到寄存器中。然而，在某些情况下寄存器存不下所有的参数。这样，通常将参数存储到内存中的块或表中，然后将内存块的地址作为参数传送到寄存器中（图 3.1）。Linux 采用了这种方法。也可以使程序将参数放置或推入堆栈中，然后由操作系统从堆栈中弹出。有些操作系统更喜欢利用块或堆栈，因为这些方法没有限制参数的数量和长度。

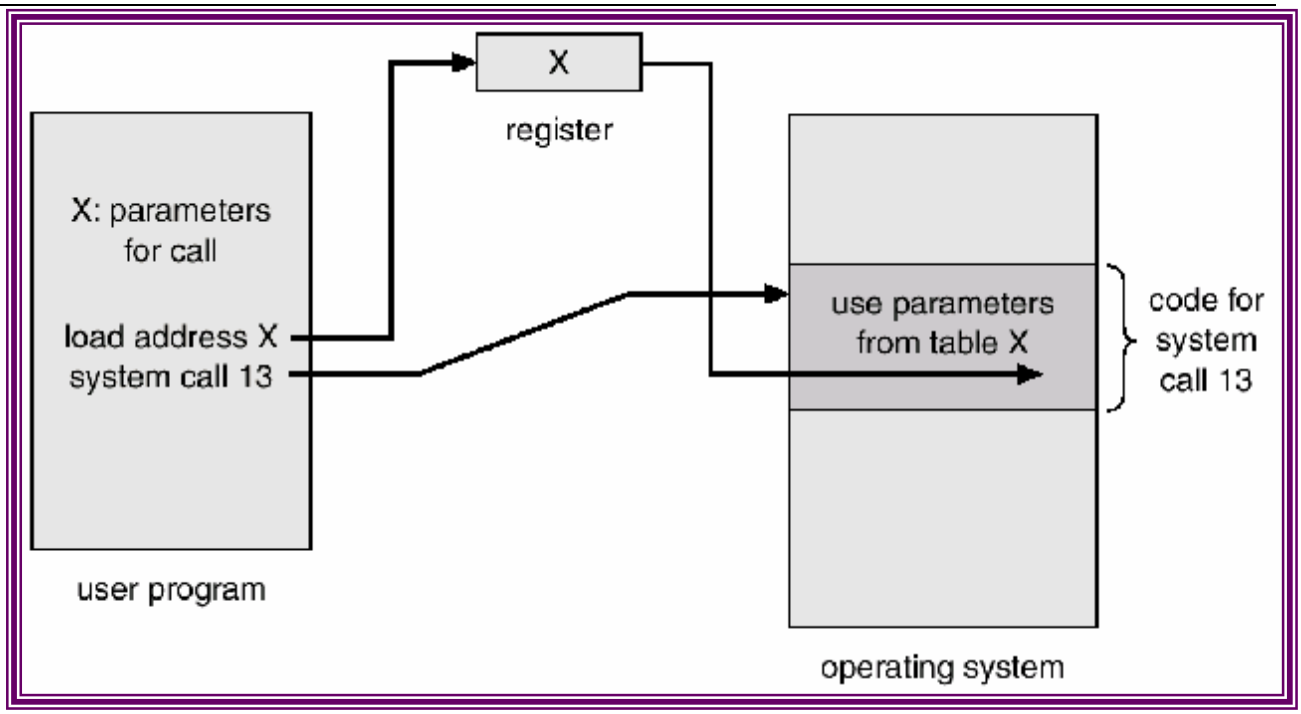


Figure 3.1 Passing of parameters as a table.

可以大概的将系统调用分为五类：**进程控制**、**文件管理**、**设备管理**、**信息维护**和**通信**。从 3.3.1 节到 3.3.5 节，我们简要的讨论一个操作系统可能会提供的系统调用的类型。其中大多数这些系统调用支持（或被支持）后面的章节中所讨论的概念和功能。图 3.2 概述了一个操作系统通常提供的系统调用类型。

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

Figure 3.2 Types of system calls.

Figure 3.2 Types of system calls.

3.3.1 进程控制

正在运行的程序需要能够正常（结束）或非正常（异常终止）的结束执行。如果一个程序是被一个系统调用异常终止的，或者是程序运行时出现问题并产生了错误自陷，有时要使用一段内存信息和产生的错误消息。这段内存被转储到磁盘上，以便**调试器**检查问题的起因。在正常或异常的情况下，操作系统都必须将控制移交给命令解释程序。然后，命令解释程序读取下一条命令。在交互式系统中，命令解释程序简单的继续下一条命令；它假设用户对任何错误都有合适的命令处理。在批处理系统中，命令解释程序通常会终止整个作业并开始下一个作业。有些系统允许控制卡片指明在错误发生时做出指定的校正动作。如果一个程序发现了一个输入错误并希望能够异常终止，那么它可能也需要定义一个误差级（error level）。通过更高级别的错误参数可以指明更多严重的错误。通过把正常终止定义为 0 级错误的方式可以将正常和非正常终止结合起来。命令解释程序或随后的程序可以根据这个误差级自动的决定下一步的操作。

一个进程或运行中的作业程序可能要载入并执行另外一个程序。这个特性允许命令解释程序通过例如用户命令、鼠标单击或批处理命令等方式直接运行程序。一个有趣的问题是：在载入的程序结束后，把控制返回到哪里呢？这个问题关系到现有的程序是被丢失、保存或是被允许与新的程序并行执行。

如果在新程序结束时将控制返回给现有的程序，我们必须保存该程序的内存映象；这样，我们要创建一个有效的机制用于使一个程序调用其它的程序。如果想要两个程序能够并行执行，我们就要进行多道程序设计来创建新作业或进程。（If both programs continue concurrently, we have created a new job or process to be multiprogrammed.）许多系统调用因此而生（create process 或 submit job）。

如果我们创建了一个新作业或进程，或者甚至是一组作业或进程，我们应该能够控制它（它们）的运行。这种控制需要能够测定和重新设置作业或进程的标志，这包括了进程的优先权、最大运行时间等等（get process attributes 和 set process attributes）。如果我们发现创建的作业或进程出错或不再有用时，可能也想终止它们（terminate process）。

创建新作业或进程后，我们可能需要等待它们执行结束。我们可能要等待确定的时间（wait time）；更有可能等待特定事件的发生（wait event）。当事件发生时，作业或进程应该发出信号（signal event）。将在第七章详尽讨论这种类型的系统调用（处理并行进程的系统运行）。

其它的系统调用对排除程序故障有所帮助。有些系统提供了系统调用以内存转储。这种规定有益于故障排除。一个程序**跟踪**它执行时的每条指令列表；很少有系统提供了这种功能。甚至微处理器也提供了一种被称为**单步执行**的 CPU 模式，这种模式下 CPU 在每条指令之后自陷一次。通常一个调试程序捕获这个自陷，这个捕获程序设计用于帮助程序员发现和修正错误的系统程序。

许多操作系统为程序提供了一个时间表。它用于指示该程序在特定位置或区域的执行时间。时间表需要具备跟踪功能或规则的计时中断。在每次计时中断发生时，就记录一次程序计数器值。如果计时中断足够频繁，就能够获取表示程序的各个部分的执行时间的静态图。

进程和作业控制包含了太多的细节和变化，所以我们应该通过一些例子来阐明这些概念。MS-DOS 是一个单任务系统的例子，它包含一个开机就开始运行的命令解释程序（图 3.3a）。因为 MS-DOS 是个单任务系统，所以它以一种简单的方法运行一个程序并不再创建新进程。它将程序载入内存，写到自己内存地址的上方，并为其分配尽可能多的内存（图 3.3b）。然后将指令指针指向程序的第一条指令。然后程序开始运行，或者产生一个错误而自陷或者调用一个系统调用而结束执行。不论发生那种状况，都要将错误代码保存在系统内存中备用。紧随其后，命令行解释器中常驻内存的那一小部分恢复执行。它首先将命令解释器的其它部分重新从磁盘中载入内存。然后，命令解释器将先前程序的错误码提供给用户或下个程序。

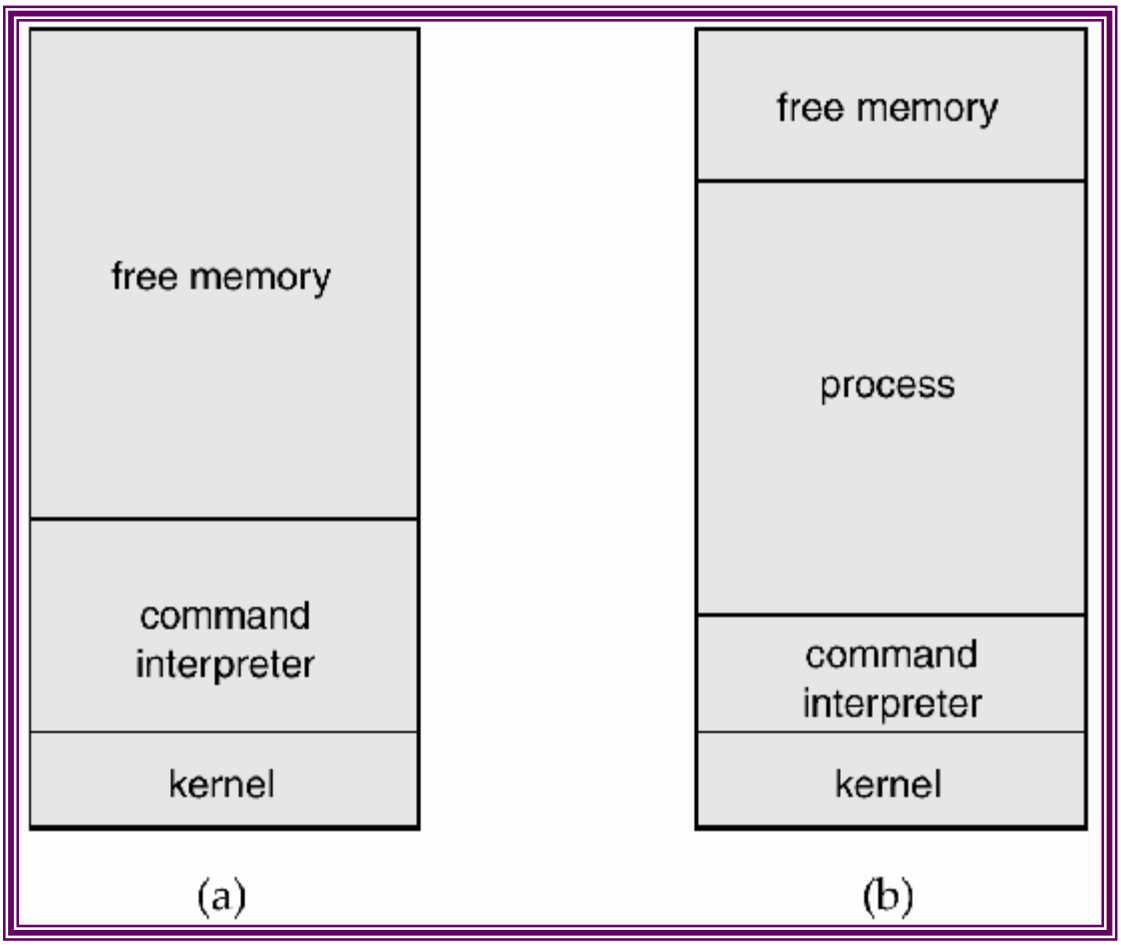


Figure 3.3 MS-DOS execution. (a) At system startup. (b) Running a program.

虽然 MS-DOS 操作系统不具备通常的多任务处理能力，但是它提供了一种方法用于有限的并行执行。一个 TSR 程序是一个程序，它“hook”一个中断并且利用 terminate 和 stay resident 系统调用退出。例如，它把它的某个子程序的地址写到中断程序列表中以便于在系统时钟触发时被调用，就这样钩（hook）住了时钟。这样，TSR 程序每秒钟执行数次，而且是在时钟 tick 时开始。Terminate 和 stay resident 系统使 MS-DOS 保留 TSR 程序所占用的内存空间，所以在命令解释程序重新载入时不会被覆盖。

FreeBSD 是一个多任务系统的例子。当用户登录到系统之后，用户所选择的 shell（命令解释程序）就开始运行。这个命令解释程序与 MS-DOS 命令解释程序类似，它接收并执行用户请求的程序。然而，因为 FreeBSD 是一个多任务系统，命令解释程序在其它程序执行时继续运行（图 3.4）。命令解释程序通过执行一个 fork 系统调用来创建一个新进程。然后，被选择的程序通过 exec 系统调用载入内存并执行。根据输入命令的不同，命令解释程序等待进程结束或在后台运行进程。在后一种情况下，命令解释程序立即请求下一个命令。当一个进程在后台运行时，它不能够直接接收键盘输入，因为命令解释程序正在使用这个资源。因此，I/O 通过文件完成或者通过窗口菜单接口。其间，用户可以自由的要求命令解释程序运行其它的程序、监视正在运行的进程的进度、更改程序优先级等。当一个进程完成时，它执行一个 exit 系统调用结束运行，返回给调用进程一个状态代码或是一个非零的错误代码。然后这个状态（或错误）代码对命令解释程序或其它程序有效。将在第四章中讨论进程。

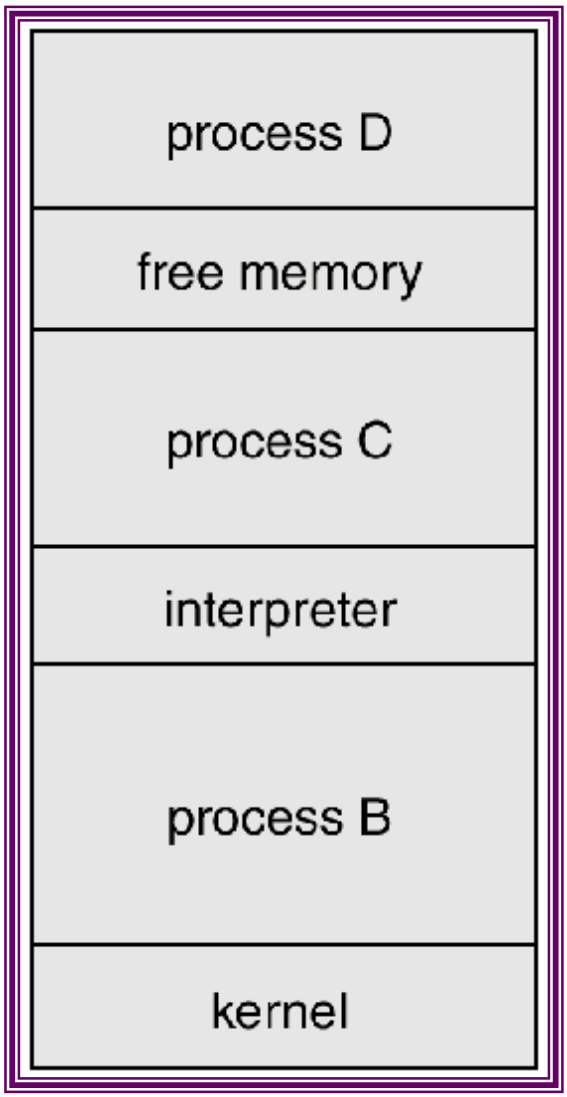


Figure 3.4 UNIX running multiple programs.

3.3.2 文件管理

我们将在第十一章和第十二章中详细讨论文件系统。然而我们可以认识几个通用的处理文件的系统调用。

首先，我们需要能够创建和删除文件。每个系统调用都需要获得文件名并且也许还需要知道一些文件属性。一旦创建了文件，我们需要打开并使用它。我们也可能读、写或重定位（例如：倒带或跳到文件末端）。最后，我们需要关闭文件，表明我们不再使用它。

如果在文件系统中我们使用一个目录结构来组织文件，那么就需要针对目录的这些同样的一系列操作。另外，对于文件或目录，我们需要能够确定其各种属性值，并能够在需要的时候重新设置。文件属性包含了文件名、文件类型、保护代码、计数信息等等。至少需要 get file attribute 和 set file attribute 两个文件调用。有些操作系统提供了更多的调用。

3.3.3 设备管理

当一个程序运行时，可能需要额外的资源。这（额外的资源）可能是更多的内存、磁带驱动器、文件等等。如果资源有效，那么就可以将它们分配给用户程序并将控制返回给用户程序；否则，程序就需要等待，直到所需的资源有效。

可以把文件看成抽象或虚拟设备。这样，设备也需要针对文件的许多系统调用。如果系统有多个用户，就必须要首先请求设备，以保证对设备的独占使用。在使用设备结束后，我们必须释放它。这些功能与打开和关闭文件的系统调用非常类似。

一旦设备被请求（并分配给用户），我们就能够读、写和（可能）重定位该设备，正像我们对文件所做的那样。事实上，I/O 设备和文件相当的类似，包括 UNIX 和 MS-DOS 在内的众多操作系统将二者合并为文件设备结构（file-device structure）。这样，通过特殊的文件明识别设备。

3.3.4 信息维护

有些系统调用仅仅是用于在用户程序和操作系统之间传输信息。例如，大多数系统有一个系统调用是返回当前时间和日期（time 和 date）。其它的系统调用可能会返回关于系统的信息，比如：当前的用户数、操作系统的版本号、空闲内存或磁盘数量等等。

另外，操作系统保持着所有的进程信息，而且有系统调用可以访问这些信息。通常也有系统调用用于重新设置进程信息（get process attributes 和 set process attributes）。在 4.1.3 节，我们将讨论通常保留哪些信息。

3.3.5 通信

有两种通用的通信模型。在**消息传递模型**中，操作系统提供进程间通信机制来实现信息交换。在通信之前必须要建立一个连接。必须要知道其它通信点的名称，这可能是处于同一个 CPU 上的另外一个进程，或者是在通过通信网络连接的另外一台计算机上的一个进程。网络上的每台计算机都有一个**主机名**（如 IP 名称），通过这个主机名来识别主机。类似的，每个进程都有一个**进程名**，可以把它转化为能够由操作系统引用的等效标识符。Get hostid 和 get processid 系统调用做这种转化。然后，根据系统的通信模型，这些标识符通过由文件系统提供的 open 和 close 调用或 open connection 和 close connection 系统调用传递。接收进程必须要利用一个 accept connection 调用提供通信许可。大多数接受连接的进程是系统提供的特定用途的**守护进程**（daemon）。它们执行一个等待连接的调用，当连接建立之后被唤醒。通信的源端叫做客户端，接收守护进程叫做服务器，信息交换通过读写消息系统调用完成。Close connection 调用终止本次通信。

在**共享存储器模型**（shared-memory model）中，进程利用 map memory 系统调用访问其它进程的内存。回想一下，操作系统通常不允许一个进程访问另一个进程的内存。共享存储器请求几个进程同意解除这种限定。（Shared memory requires that several process agree to remove this restiction.）然后，他们可以在这些共享空间上读写数据来交换信息。这些进程决定了数据的形式和位置，这不在操作系统控制之下。进程也要确保不会同时往同样的地点写入信息。在第七章中讨论这种机制。我们也将看到另为一种进程模型——**线程**——它们在默认情况下共享内存。

在操作系统中，这两种模型是通用的，而且有些系统甚至（同时）实现了二者。因为不需要避免冲突，所以在需要交换的数据量较小的情况下，消息传递很有用。对于计算机间的通信，它也更容易实现。共享存储器允许最高的速度和便利的通信，因为在一个计算机内的情况下它能够以内存的速度完成。然而，它的问题在于保护和同步的一些方面。图 3.5 展现了这两种模型的对照。

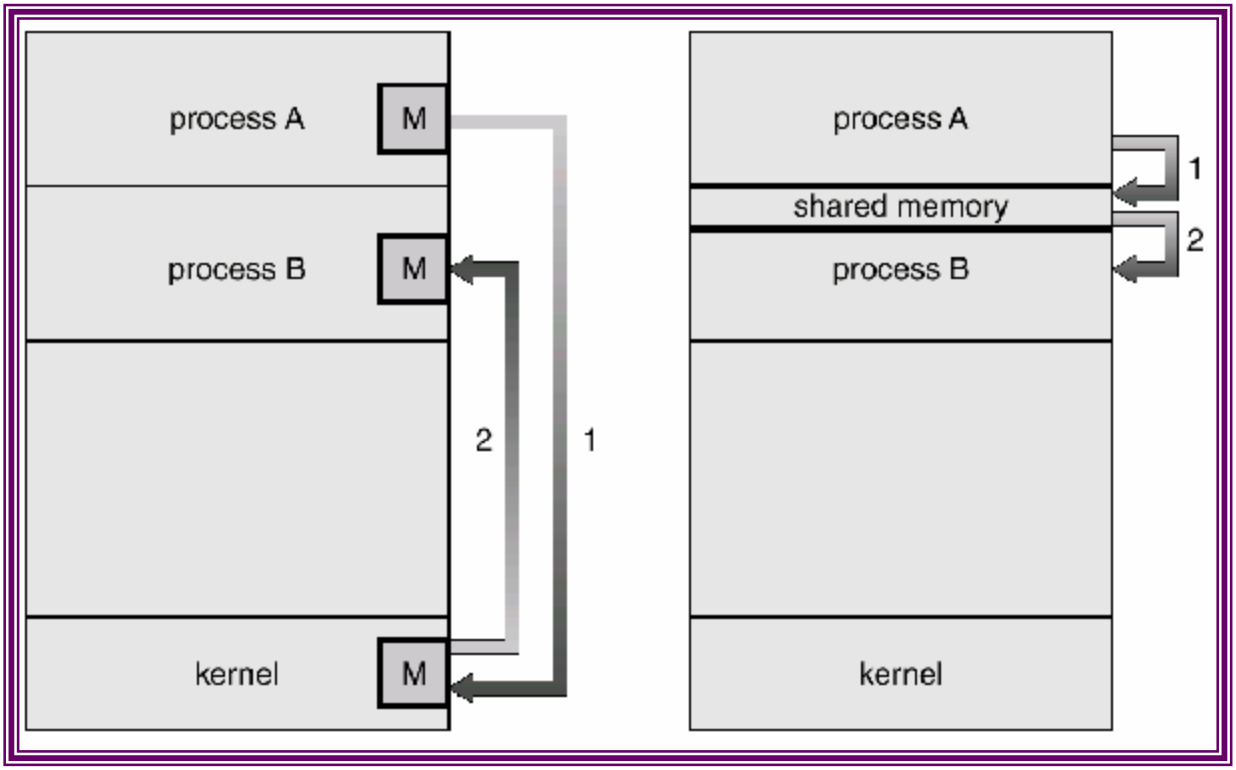


Figure 3.5 Communications models. (a) Msg passing. (b) Shared memory.

3.4 系统程序

现代操作系统的另外一个特征是系统程序。回想一下图 1.1，它描述了计算机逻辑层次。最底层是硬件。其次是操作系统，然后是系统程序，最上层是应用程序。系统程序为程序开发和执行提供了便利的环境。有些系统程序只是用户和系统调用的接口；其它的相当复杂。可以把系统程序分为一下几类：

• **文件管理：**这些程序创建、删除、拷贝、重命名、打印、内存转储（dump）、列表文件和目录。

• **状态信息：**有些程序简单的向系统要求数据、时间、有效内存或磁盘空间、用户数目等类似的信息。然后这些信息被格式化并打印到中断、其它输出设备或文件中。

• **文件修改：**一些文本编辑器也许能够创建和修改存储在磁盘或磁带上的文件。

• **程序设计语言支持：**通用程序设计语言（如 C、C++、Java、Visual Basic 和 PERL）的编译器、汇编器和解释器常常与操作系统一起提供给用户。其中有些程序现在单独标价出售。

•**程序载入和运行：**一个程序一旦被汇编或编译完毕，它必须要载入内存并执行。系统可能提供了绝对（地址）载入程序（absolute loader）、可重定位载入程序（relocatable loader）、连接编辑程序（linkage editor）和 overlay loader。也可能需要针对高级语言或机器语言的调试系统。

• **通信：**这些程序提供了进程、用户和在不同的计算机系统间建立虚拟通信的机制。允许用户发送信息到另一个用户的屏幕上、浏览 Web 页面、发送电子邮件、远程登录或通过机器传输文件。

大多数操作系统提供了这些程序来解决共同的问题或实现通用的操作。这些程序包括了网页浏览器、字处理器和文本格式化程序、电子制表软件、数据库系统、编译程序的编译程序、绘图和静态分析包（plotting and statistical-analysis package），以及游戏。这些程序被称为**系统工具**（system utility）或**应用程序**。

对一个系统来说，或许最重要的系统程序是命令解释程序，其主要功能是获取并执行下一条用户指定的命令。

在这一层次的一些命令用于文件操作：创建、删除、列表、打印、拷贝、执行等等。这些命令能够以两种

通用的方式实现。一种是命令解释程序本身包含了执行命令的代码。例如，输入命令删除一个文件，那么命令解释程序跳到（删除命令的）代码，这段代码设置参数并进行适当的系统调用。假若如此，给定的命令的数量决定了命令解释程序的大小，因为每条命令都需要自己的实现代码。

另一种方法是由系统程序来实现大多数命令，这是 UNIX 以及其它的操作系统所采用的。在这种方式下，命令解释程序不再去理解命令，它仅仅利用这个命令来识别一个文件并将这个文件载入内存中执行。这样，删除文件的 UNIX 命令 rm G 将寻找一个名为 rm 的文件，将其载入内存并以参数 G 执行。命令 rm 的功能完全定义在名为“rm”的文件的代码中。以这种方法，通过创建新文件，程序员可以很容易的给系统添加新的命令。命令解释程序（它可以非常小）则不需要为了添加的新命令而做出任何改变。

这种设计命令解释程序的方法也面临许多问题。首先，因为执行命令的代码是一个独立的系统程序，那么操作系统必须要提供一种从命令解释程序向系统程序中传递参数的机制。但这常常很困难，因为命令解释程序和系统程序不一定会同时在内存中，而且参数列表可能会非常巨大。而且，将程序载入内存执行的速度要比在当前程序中简单的跳过其它的代码执行要慢。

另一个问题是把参数的解释留给了系统程序程序员。这样，程序间和用户看到的参数可能有所不一致，因为这是不同的程序员在不同的时间写的。（Another problem is that the interpretation of the parameters is left up to the programmer of the system program. Thus, parameters may be provided inconsistently across programs that appear similar to the user, but were written at different times by different programmers.）

大多数用户眼中的操作系统是由系统程序定义的，而不是由实际的系统调用定义的。考虑一下 PC。当你的计算机运行 Microsoft Windows 时，你可能会看到 MS-DOS 命令行解释程序或窗口和菜单图形界面。虽然二者使用了同样的系统调用，但是这些系统调用看上去很不一样并且有不同的行为。因此，你的视角从真实的系统结构中转移了。因此有用的友好的用户界面的设计不是操作系统的一个直接的功能。在本书中，我们将专心于向用户提供充分的服务的基本问题。从操作系统的视角出发，我们不区分用户程序和系统程序。

3.5 系统结构

如果要使像现代操作系统这样庞大复杂的系统正确运行并易于修改，那么必须要精心设计。一种通用的方法是将任务分为许多小部分，而不是一个整体的系统（monolithic system）。每一个模块必须是一个明确定义的系统的一部分，这包括了对输入、输出和功能的详尽定义。我们已经简要讨论了操作系统的共同的组件（3.1 节）。在这一节，我们将讨论这些组件互联和融合成一个内核的方法。

3.5.1 简单结构

许多商用系统没有明确定义的结构。这些操作系统常常作为小的简单有限的系统开始，然后成长的规模远超过最初的设计：MS-DOS 就是这样的一个系统。MS-DOS 最初由不多的人设计和实现，设计者没有想到 MS-DOS 会如此流行。MS-DOS 的设计是为了在最少的空间（因为它运行在有限的硬件中）内提供大多数功能，所以没有将它仔细的分成多个模块。图 3.6 展示了它的结构。

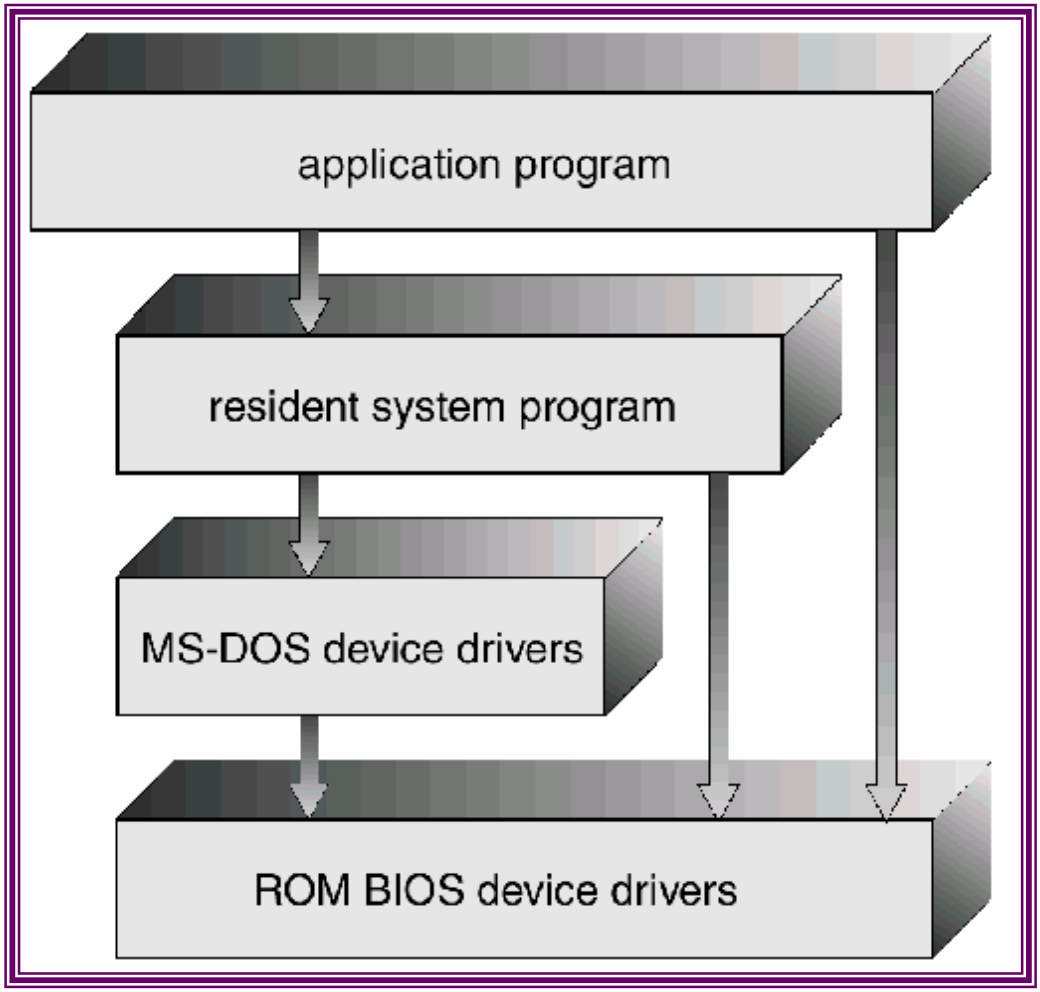


Figure 3.6 MS-DOS layer structure.

UNIX 是另一个系统，它开始时受到硬件功能的限制。UNIX 由两个可分离的部分组成：内核和系统程序。内核可以进一步分离为接口和设备驱动程序，这是随着 UNIX 的发展而逐年添加和扩展的。我们可以以图 3.7 中所展示的层次化结构来观察传统的 UNIX。处于系统调用接口之下和物理硬件之上的是内核。内核通过系统调用提供了文件系统、CPU 调度、内存管理和其它的操作系统功能。总体上讲，为数众多的功能组合在了一个层面中。这使得难以增强 UNIX 的功能，因为每一区域的变动都会对其它的方面产生不利影响。

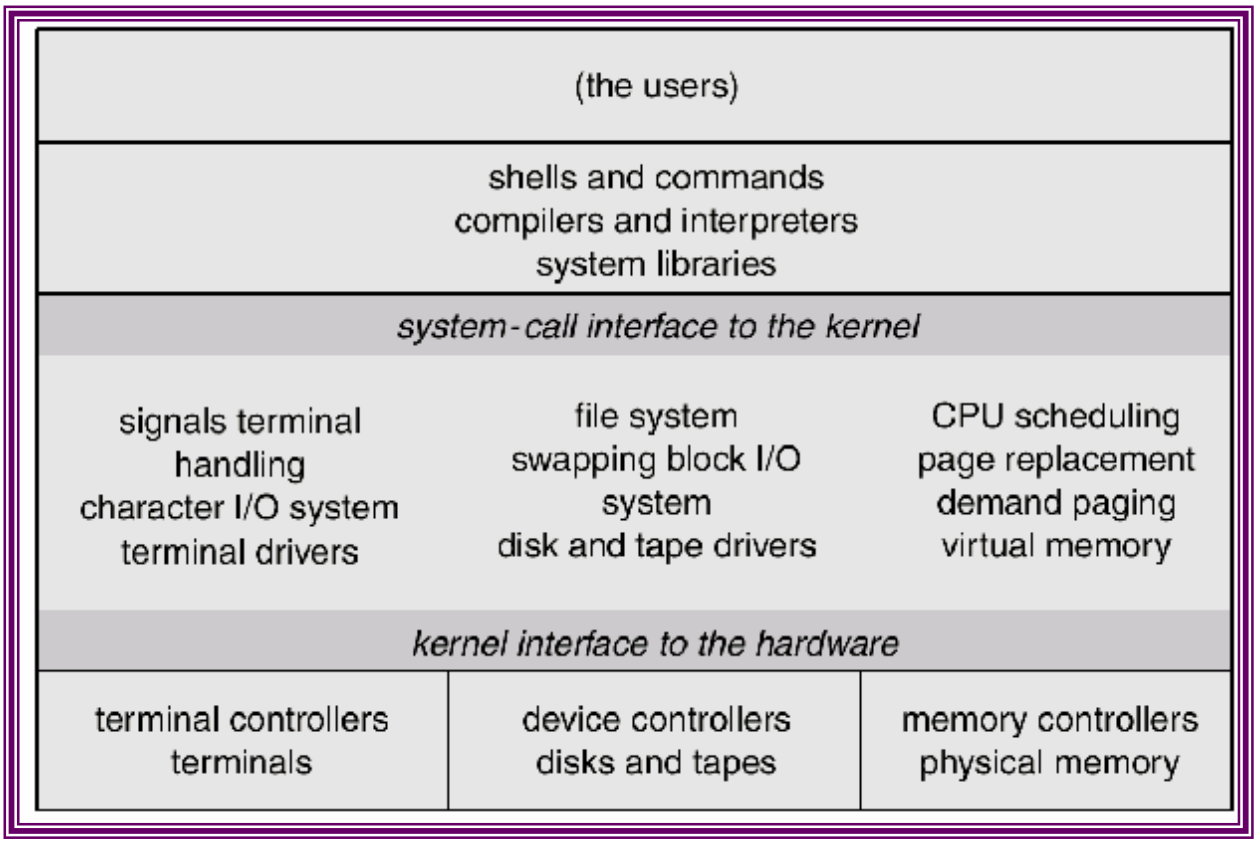


Figure 3.7 UNIX system structure.

系统调用为 UNIX 定义了 API；这些系统程序共同定义了系统接口。程序员和用户接口定义了内核必须支持的上下文。

UNIX 的新版本设计采用了更先进的硬件。由于获得了充分的硬件支持，就可以把操作系统分成更小更合适的部分，而不像 MS-DOS 或 UNIX 系统那样。于是操作系统能够保持对计算机和使用计算机的应用程序的更大的控制。实现者在选择系统内部工作或创建操作系统模块时有了更大的自由。（Implementers have more freedom to make changes to the inner workings of the system and in the creation of modular operating system.）在自顶向下（top-down）的设计方法中，把全部的功能和特性分离成组件。这种分离允许程序员隐藏信息；所以他们可以自由的实现中意的底层程序，在不改变程序的外部接口的情况下使程序完成所需的任务。

3.5.2 层次化设计

有多种方法可以完成系统的模块化设计。一种方法是**层次化设计（layered approach）**，这种技术将操作系统分为若干层（或层次），每一层构建在下面一层之上。最底层（第 0 层）是硬件；最高层（第 N 层）是用户接口。

操作系统的一层是对一个抽象对象的实现，它封装了数据和对这些数据的操作。图 3.8 描述了一个典型的操作系统层（M 层）。它由数据结构和可由更高层调用的一系列程序构成。M 层也能够调用更低层的操作。

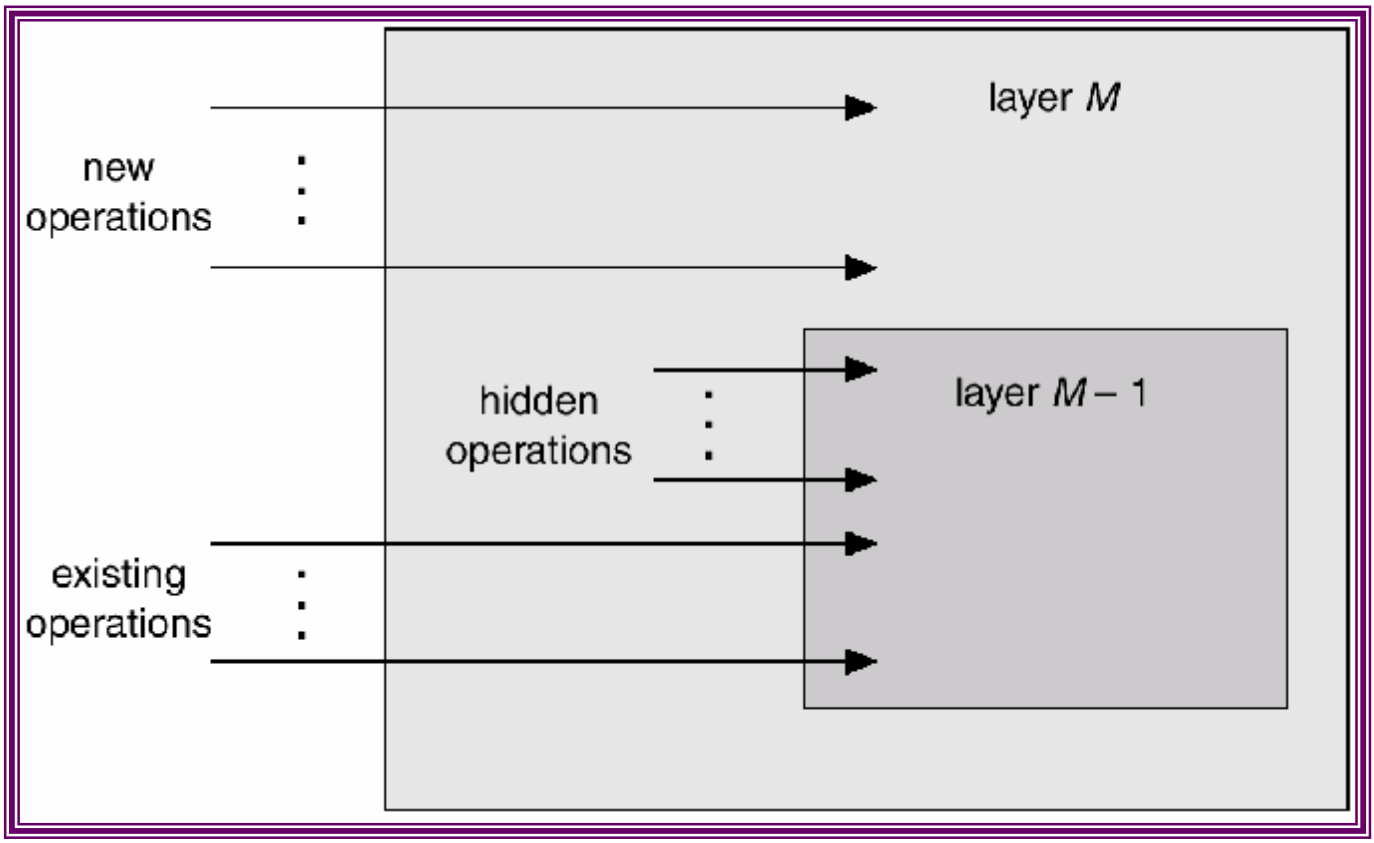


Figure 3.8 An operating-system layer.

层次化设计的主要优点是**模块化**。每一层都只会用到更低层所提供的功能（或操作）和服务。这种技术简化了调试和系统验证。调试第一层时能够不受系统其它部分的干扰，因为根据定义它仅仅使用了基本硬件（假设硬件工作正常）来实现它的功能。一旦第一层调试完毕，调试第二层时可以假设第一层工作正常，依此类推。如果在调试特定的一层时发现了错误，那么这个错误一定在这一层，这是因为下面的各层已经调试通过了。这样，把系统拆为层次，就简化了系统的设计和实现。

每一层只能够过低层所提供的那些操作实现。一个层不需要知道这些操作的具体实现；只需要知道这些操作的作用。因此，每一层向更高层隐藏了具体的数据结构、操作和硬件。

层次化设计的主要困难包含了对层的详细定义，因为一个层只能使用其下的那些层（所提供的操作）。例如，存储器管理需要具备应用磁盘的能力，所以虚拟存储器算法中所使用的磁盘驱动程序就必须要处于存储器管理程序下面的层上。

其它的请求可能就没有这么明显了。备份存储驱动器可能需要等待 I/O，而且可以在这段时间内重新调度 CPU，所以它通常要处于 CPU 调度程序上层。然而，在一个大型系统上，内存容量可能不能满足 CPU 调度程序存储活动进程信息的要求。这样，就需要将这些信息换入和换出内存，这就需要备份存储驱动器程序处于 CPU 调度程序下层。

层次化设计的最后一个问题是它的效率低于其它类型的系统。例如，当一个用户程序执行一个 I/O 操作时，它就要执行一个自陷到 I/O 层的系统调用（it executes a system call that is trapped to the I/O layer），这个调用再调用存储器管理层，再由存储器管理层调用 CPU 调度层，然后 CPU 调度层转到硬件。在每一层中，可能要修改参数，可能要传递数据，等等。每一层都要增加系统调用的开销；最终要比采用非分层的系统花费更多的时间。

近几年，这些限制造成了对分层思想的一些的质疑。在设计中采用更少的层次，使每一层提供更多的功能，这样就能够在避开层次定义和交互的困难问题的同时提供了模块化代码的大多数优点。例如，OS/2 是 MS-DOS 的一个后继产品，它添加了多任务和双模式操作，以及其它的新特性。因为复杂性的提高和硬件功能的增强，

OS/2 的设计实现中就分了更多层。把图 3.9 描述的结构与 MS-DOS 的结构（图 3.6）对比；从系统设计和实现出发，OS/2 有它的优点。例如，不允许用户直接访问底层程序、向操作系统提供了对硬件的更多的控制和更多的用户程序使用资源的信息。

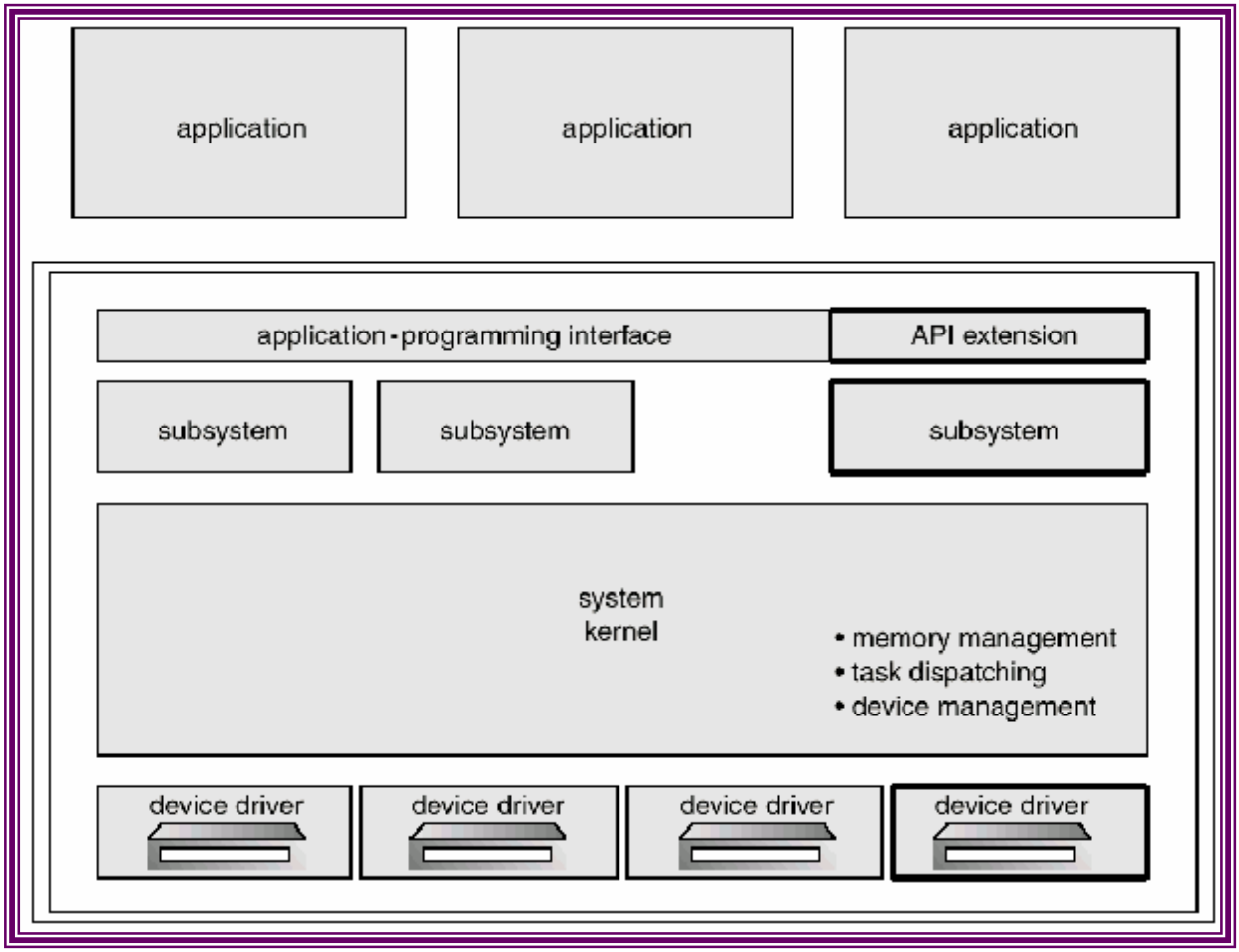


Figure 3.9 OS/2 layer structure.

作为另外一个例子，考虑一下 Windows NT 的历史。Windows NT 的第一版有一个高度面向层次的结构；但是性能却比 Windows 95 差。Windows NT 4.0 将一些层次从用户空间移到内核空间，并更紧密的整合，部分的解决了性能问题。

3.5.3 微内核

随着 UNIX 操作系统的不断扩展，其内核变得巨大而且难以管理。80 年代中期，Carnegie Mellon 大学的研究人员开发了名为 Mach 的操作系统，它采用**微内核**技术来模块化内核。这种技术将所有不必要的组件从内核中去掉，并将它们作为系统和用户层程序来实现。结果造就了更小的内核。对于哪些服务应该保留在内核中和哪些应该在用户空间实现需要达成共识。然而，微内核通常典型的最小限度的提供了进程和存储器管理，以及通信机制。

微内核的主要功能是在客户端程序和同样运行在用户空间的服务程序之间提供通信机制。通信是通过消息传递提供的，在 3.3.5 节描述。例如，如果客户程序想要访问一个文件，它就必须要与文件服务程序交互。客户程序和服务程序之间不会直接交互。相反，它们利用微内核交换信息而间接的通信。

微内核的优点还包括简化了对操作系统功能的扩充。向用户空间中添加的所有的新服务都不需要修改内核。必须修改内核时，改变造成的影响更小，因为微内核是一个更小的内核。这样，操作系统更易于移植。(The

resulting operating system is easier to port from one hardware design to another.)因为大多数服务作为用户进程而不是作为内核进程运行，所以微内核也提供了更强的安全性和可靠性。如果一个服务发生故障，操作系统的其它部分不会受到影响。

当前有几个操作系统采用了微内核技术。Tru64 UNIX（以前的 Digital UNIX）向用户提供了一个 UNIX 接口，但它是以 Mach 内核实现的。Mach 内核将 UNIX 中的那些系统调用映射为发给用户级服务的消息。（The Mach kernel maps UNIX system calls into messages to the appropriate user-level services.）Apple MacOS X Server 基于 Mach 内核。

QNX 是一个实时操作系统，它也是基于微内核设计。QNX 微内核提供了消息传递和进程调度的服务。它也处理底层网络通信和硬件中断。QNX 中所有其它的服务由在内核外以用户模式运行的标准的进程提供。

Windows NT 采用了一种混合结构。我们已经注意到了 Windows NT 体系结构部分采用了层次化设计。Windows NT 设计用于运行各种程序，包括 Win32（原生 Windows 程序）、OS/2 和 POSIX。它为每种类型的程序提供了一个运行在用户空间的服务。针对每种类型程序的客户程序也运行在用户空间。内核协调着客户端应用程序和应用程序服务器间的消息传递。图 3.10 描述了 Windows NT 的客户端 / 服务器结构。

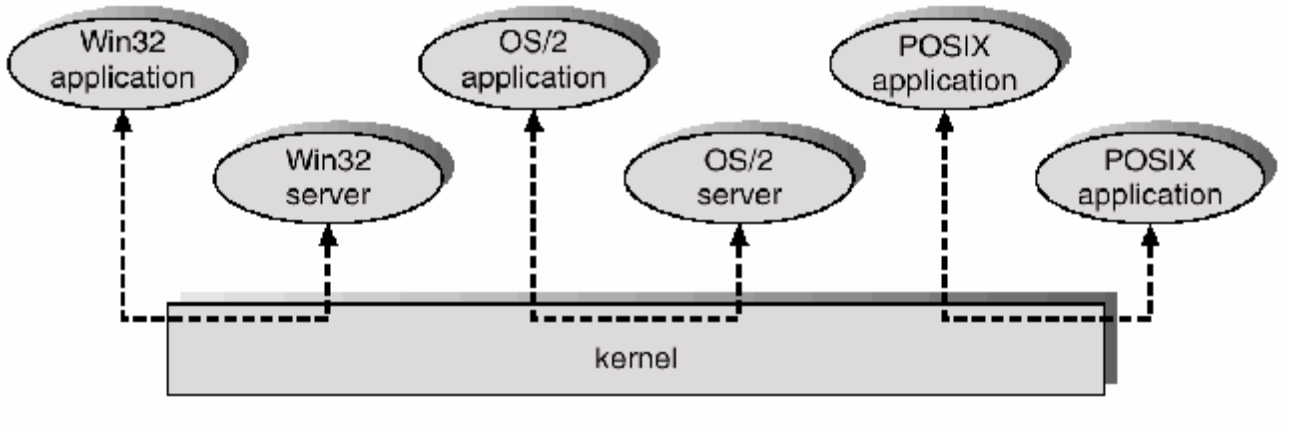


Figure 3.10 Windows NT client-server structure.

3.6 虚拟机

从概念上讲，一个计算机系统由多个层次组成。在所有这样的系统中，硬件处于最底层。内核运行在紧挨着的上一层，它使用硬件指令为外层创建系统调用。内核之上的系统程序可以使用系统调用或硬件指令，二者没有什么区别。这样，虽然对它们的访问有所不同，但是它们都提供了函数，程序能够利用这些函数实现更高级的功能。系统程序则把硬件和系统调用看作在同一层上。

通过允许应用程序容易的调用系统程序，有些系统进一步发展了这种机制。像以前一样，虽然系统程序比其它的程序所处的层次更高（这个地方是说，系统程序比硬件、内核所处的层次高），但是应用程序可能把在层次结构上处于它们之下的一切看作机器本身的一部分。这种层次化设计就形成了**虚拟机**概念。IBM 系统的 VM（虚拟机）操作系统是虚拟机概念的最好的例子，因为 IBM 是这一领域的倡导者。

通过使用 CPU 调度（第六章）和虚拟机技术（第十章），操作系统能够给人一种错觉，以为进程拥有自己的处理器和自己的（虚拟）存储器。当然，进程通常有着不是由纯硬件提供的额外特性（如系统调用和文件系统）。换句话说，虚拟机技术没有提供任何额外的功能，而是提供了一个等同于底层纯硬件的一个接口。为每个进程提供了一个（虚拟的）底层计算机的拷贝（图 3.11）。

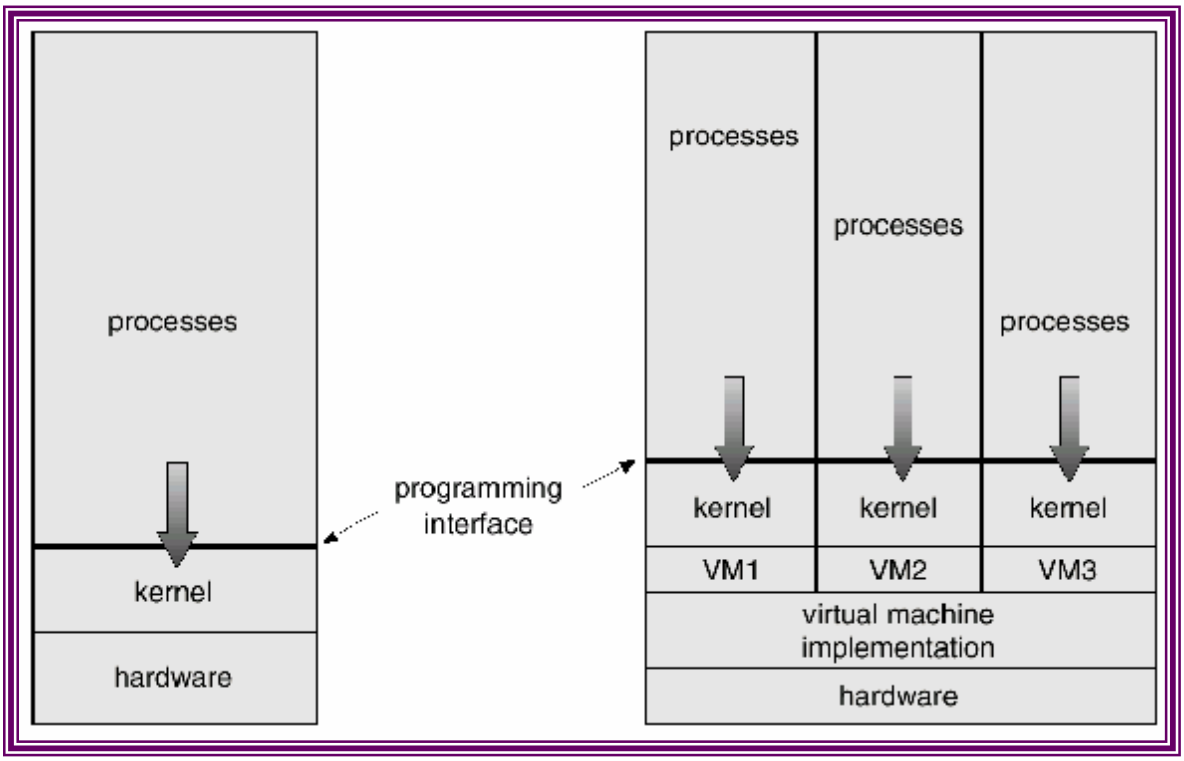


Figure 3.11 System models. (a) Nonvirtual machine. (b) Virtual machine

虚拟机之间共用物理计算机的资源。CPU 调度可以分配 CPU，这样就产生了用户拥有自己的处理器的特性。Spooling 技术和文件系统能够提供虚拟读卡机和虚拟行式打印机。一个普通的分时用户终端提供了虚拟机操作员的控制台的功能。

虚拟机技术的一个重大的困难涉及到了磁盘系统。假如物理机器有三个磁盘驱动器，但是要支持七个虚拟机。很明显，不能为每个虚拟机分配一个磁盘驱动器。要记住虚拟机软件本身需要真实的磁盘空间来提供虚拟内存。解决方法是提供虚拟磁盘，除了容量外所有的特征等同于实际的磁盘，在 IBM 的 VM 操作系统中称为虚盘（minidisk）。系统在物理磁盘上为每个虚盘提供所需的空间，这样来实现虚盘。很明显，所有虚盘的总容量一定要小于物理磁盘的容量。

用户因此获得了自己的虚拟机。于是他们可以运行任何对底层机器有效的操作系统或软件包。对于 IBM VM 系统，用户通常运行 CMS，这是一个单用户交互式操作系统。虚拟机软件专注于在一台物理机器上并行执行多个虚拟机，而不需要考虑用户支持软件。这种设计有助于将多用户交互式系统的设计问题分为更小的两个部分。（The virtual-machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but it does not need to consider any user-support software. This arrangement may provide a useful partitioning into two smaller pieces of the problem of designing a multiuser interactive system.）

3.6.1 实现

虽然虚拟机概念非常有用，但是却难以实现。为了提供一个精确的底层机器的复本，许多工作是必需的。底层机器有两种模式：用户模式和监控模式。因为虚拟机软件是操作系统，所以它能够运行在监控模式下。虚拟机本身只能执行在用户模式。正像是物理机（physical machine）有两种模式，虚拟机也必须要有两种模式。因此，必须要有一个虚拟用户模式和一个虚拟监控模式，二者运行在真实的用户模式下。这些活动造成了在真实机器上从用户模式到监控模式的转换（任何系统调用或试图执行特权指令），也会造成在虚拟机上从虚拟用户模式到虚拟监控模式的转换。

这些转换可以相当容易的完成。例如，当一个运行在虚拟机上以虚拟用户模式运行的程序调用一个系统调用时，将在真实机器上产生一个到虚拟机监控程序的转换。当虚拟机监控程序获得控制后，它就能够改变寄存器值和程序计数器使虚拟机模拟该系统调用的效果。然后，它能够重新启动虚拟机，要注意到它现在处于虚拟监控模式。如果虚拟机试图操作，例如从虚拟读卡器中读取数据，它将执行一个特权 I/O 指令。因为虚拟机运行在物理用户模式，该指令将自陷给虚拟机监控程序。然后虚拟机监控程序必须模拟该 I/O 指令的执行效果。首先，它找到实现了虚拟读卡机的假脱机文件。然后，它将对虚拟读卡器的读取转换为对假脱机磁盘文件的读取，并将下一条虚拟“卡片映象”转移到虚拟机的虚拟存储器。最后，它重新启动虚拟机。虚拟机状态被修改，就像是运行在真正的监控模式的真实机器上的一个真实读卡器执行了这条 I/O 指令。（The state of the virtual machine has been modified exactly as though the I/O instruction had been executed with a real card reader for a real machine executing in a real monitor mode. ）

当然，最大的差别在于时间。然而真实的 I/O 可能需要 100 毫秒，虚拟 I/O 可能需要更少时间（因为它采用 spooling）或更多时间（因为它被解释执行）。另外，CPU 在多个虚拟机间进行并行控制，进一步以不可预知的方式降低了虚拟机速度。在极端的情况下，它可能必须要模拟所有的指令来提供真正的虚拟机。因为通常的虚拟机指令能够直接在硬件上执行，所以 VM 能够为 IBM 机器工作。（VM works for IBM machines because normal instructions for the virtual machines can execute directly on the hardware.）只有特权指令（主要用于 I/O）必须要模拟实现，因此执行也更慢一些。

3.6.2 优点

虚拟机技术主要有两个优点。首先，通过完全的保护系统资源，虚拟机提供了一个健壮的安全保护层。其次，虚拟机允许在不干扰正常的系统操作的情况下进行系统开发。

每个虚拟机与其它所有的虚拟机是完全独立的，各种系统资源完全受到保护，所以我们没有安全方面的问题。例如，从 Internet 上下载的不可靠的应用程序运行在一个独立的虚拟机上。这种环境的一个缺点是没有直接的共享资源：实现了两种共享方法。首先，它可能共享一个虚盘。这种方案模拟物理共享磁盘，但是以软件实现。利用这种技术，可以共享文件。其次，可以定义一个虚拟机网络，网络中每个虚拟机可以在虚拟通信网络上发送信息。这样，这种网络模拟了物理通信网络，但是以软件实现。

对于操作系统的研究和开发来说，这样的虚拟机系统是一个完美的手段。通常修改操作系统是非常困难的。因为操作系统是一个大型的复杂的程序，某处的一个改变可能会造成其它地方的隐含错误。操作系统的功能又非常强大，这更是雪上加霜。因为操作系统执行在监控模式，对一个指针的错误修改可能会造成一个可能毁坏整个文件系统的错误。因此，必须要仔细的测试对操作系统的所有的改变。

然而，操作系统运行在机器上并控制着整个机器。所以，在修改和测试（操作系统）时必须要停下当前系统并且停止使用。通常把这个阶段称为**系统开发时间**（system-development time）。因为在这个过程中用户不能够使用系统，所以系统开发经常在晚上或周末，在系统利用率低的时候。

虚拟机系统可以消除许多这样的问题。系统程序员使用自己的虚拟机，在虚拟机上完成系统开发，而不是在真实的物理机器上。正常的系统操作很少中断系统开发。不管这些优点，只是最近才在技术上有所进步。

作为一种解决系统兼容性的方法，虚拟机的应用日益增多。例如，在基于 Intel CPU 的系统上有数以千计的应用程序可以运行在 Microsoft Windows 上。像 Sun Microsystems 这样的计算机供应商使用其它的更快的处理器，但是希望他们的客户能够运行这些 Windows 应用程序。解决方法是在原本的处理器上创建一个虚拟 Intel 机器。一个 Windows 程序运行在这种环境中，它的 Intel 指令被转换成原生指令集。Microsoft Windows 也可以运行在这个虚拟机上，所以 Windows 应用程序照常可以调用 Windows 系统调用。最终结果是：一个程序看上去运行在基于 Intel 处理器的系统上，但实际上是在一个不同的处理器上执行。如果该处理器的速度足够快，那么 Windows 程序将运行的很快，即使是每条指令都要被转换为几条原生指令（native instruction）。相似的，基于 PowerPC 的 Apple Macintosh 包括了一个 Motorola 68000 虚拟机，可以运行为过去的基于 68000 的 Macintosh

编写的二进制代码。问题是，被仿真的机器越复杂，就越难以（为它）构建一个精确的虚拟机，而且该虚拟机的运行速度越慢。

更近的一个例子是 Linux 操作系统。现在其虚拟机允许在基于 Linux 的计算机上运行 Windows 应用程序。该虚拟机可以运行 Windows 应用程序以及 Windows 操作系统。

Java 的一个主要特性是它运行在一个虚拟机上，因此允许 Java 程序运行在安装了 Java 虚拟机的任何计算机系统上。

3.6.3 Java

Java 是 Sun Microsystems 在 1995 年底开发的一个非常流行的面向对象程序设计语言。除了语言规范和一个大型 API 库之外，Java 也为 **Java 虚拟机（JVM）**提供了一个规范。

Java 对象是由类构造来定义的；一个 Java 程序由一个或多个类构成。Java 编译器为每个类生成一个平台无关（architecture-neutral）的**字节码**输出文件（.class），它可以在任何一个 JVM 实现上运行。

JVM 是一个抽象计算机的描述。JVM 包括一个**类装载器**、一个**类检查器**和一个负责执行字节码的 **Java 解释器**。类装载器负责装载程序和 Java API 中的类文件，以便 Java 解释器执行。类装载之后，检查器检查这个类文件是否合法和会不会导致栈的上溢或下溢。它也要确认字节码不做指针运算，因为那可能会导致非法的内存访问。类通过检查之后，就可以通过解释器运行了。JVM 可以自动管理内存，执行**垃圾回收**——收回对象不再使用的内存并交还给系统。有很多人研究垃圾回收的算法，想要提高虚拟机运行程序的效率（性能）。

Java 解释器可以作为软件模块，它一次解释一个字节码；也可能是一个 **JIT（just-in-time）**编译器，它将字节码转化成本机的机器语言。因为 JIT 编译器性能更好，大多数 JVM 的实现都使用 JIT 编译器。作为模块的解释器，（通常）可以通过硬件实现，从而本身就可以执行 Java 字节码。图 3.12 展示了一个 JVM。

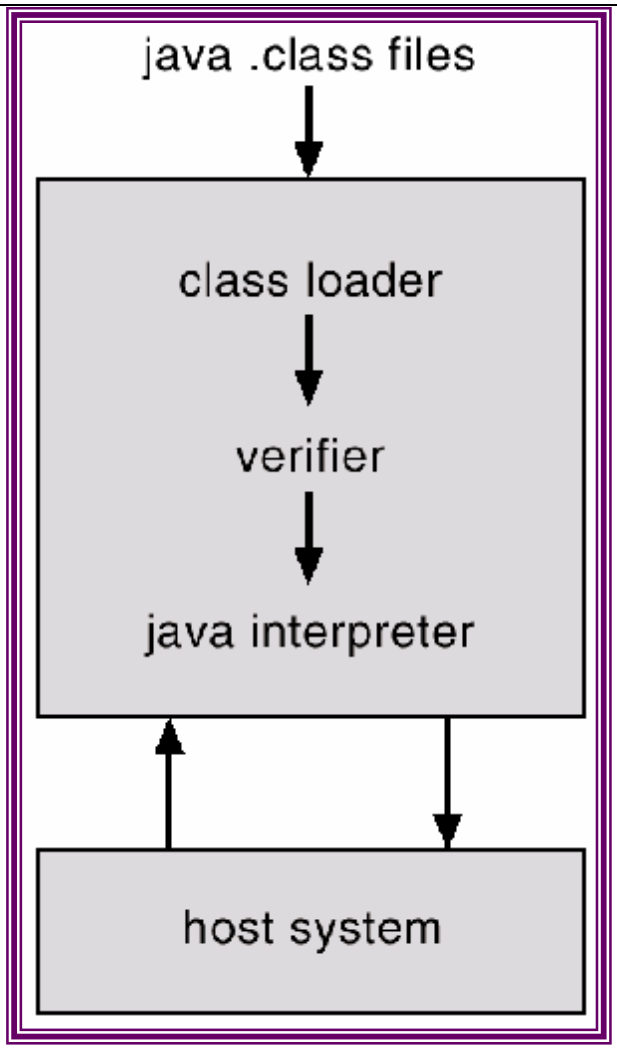


Figure 3.12 The Java virtual machine.

使用 JVM 可以开发出平台无关的可移植的程序。JVM 的实现是特定于某个系统的，因系统不同而需要不同的实现（例如 Windows 或 UNIX），JVM 以一种标准的方式为 Java 程序抽象了系统（JVM abstracts the system in a standard way to the Java program），提供了一个简洁、平台无关的接口。这样，类文件就可以在任何一个实现了合乎规范的 JVM 的系统上运行。

Java 利用了完全由虚拟机实现的环境的优点。它的虚拟机的设计提供了一个安全、高效、面向对象的、可移植的且平台无关的平台，在这个平台上面可以运行 Java 程序。

3.7 系统设计和实现

在这一节，我们讨论在设计和实现一个系统时所面临的问题。对于设计问题没有一个完全的解决方案，但是却有一些成功的方法。

3.7.1 设计目标

设计一个系统所面临的第一个问题是定义该系统的目标和规范。在最高层，对硬件和系统类型的选择将影响到系统的设计：批处理系统、分时系统、单用户系统、多用户系统、分布式系统、实时系统或通用系统。

除了这个最高层以外，需求可能更难以指定。可以把系统需求分为两大类：用户目标和系统目标。

在一个系统中，用户期望获得一些显而易见的特性：系统应该便于使用、易于学习、可靠、安全和快速。当然，对于怎样达成这些目标没有一般的认识，这些细节在系统设计中并不是特别有用。

那些必须要设计、创建、维护和操作该系统的人可以定义类似的系统需求：这个操作系统应当易于设计、实现和维护；它应当灵活、可靠、error free 并且高效。这又是一些含糊不清且没有通用的解决方案的需求。

对于定义一个操作系统的需求这个问题，没有唯一的解决方案。在多个领域中的大量系统的事实表明，不同环境中的不同的需求需要截然不同的解决方案。例如，MS-DOS（针对微型计算机的单用户操作系统）的需求与 MVS（针对 IBM 大型计算机的大型的多用户多路存取操作系统）的需求必须要有本质上的不同。

3.7.2 Mechanism 和 Policy

操作系统的规范和设计是一个高度创造性的工作。虽然没有任何教科书能够告诉你怎么做，但是通用的**软件工程**原理尤其适用于操作系统。

一个重要的原理是将 policy 和 mechanism 分离。**Mechanism** 定义了怎样做一件事；**Policy** 定义了将要做什么。例如，计时器结构（2.5 节）是一个用于 CPU 保护的 mechanism，但是为一个特定的用户设置多长的计时器值是一个 policy 决定。

对 policy 和 mechanism 的分离对于系统灵活性是很重要的。Policies 很可能随地点和时间而发生变化。在最糟的情况下，policy 中的每项改变都需要底层下面的 mechanism 做出相应的调整。我们更期望通用的 mechanism。在 policy 中的每项改变仅仅需要重新定义确定的一些系统参数。例如，在一个计算机系统里有这样一个 policy，它规定 I/O 密集（I/O-intensive）的程序比 CPU 密集（CPU-intensive）的程序有更高的优先级，那么如果能很好的将 policy 和 mechanism 分离的话，就可以很容易的在其他计算机上实现相反的 policy。

通过实现一组基本的原构造块（set of primitive building blocks），基于微内核的操作系统把 mechanism 和 policy 的分离做到了一个极限。这些块几乎与 policy 无关的，它们允许通过用户创建的内核模块或应用程序来添加更高级的 mechanisms 和 policies。另一个极端，例如 Apple Macintosh 操作系统，把 policy 和 mechanism 都编码到系统中，使得整个系统有一样的观感。因为接口本身构建到了内核当中，所以所有的应用程序都有着类似的接口。

必须为所有的资源分配和调度问题做出 policy 决定。当这个问题是怎么样而不是是什么的时候，就必须要决定一个 mechanism。（Whenever the question is *how* rather than *what*, it is a mechanism that must be determined.）

3.7.3 实现

一旦设计完毕，就要实现这个操作系统。传统上使用汇编语言编写操作系统。然而现在经常使用高级语言（如 C 或 C++）。

第一个没有使用汇编语言来实现的系统是为 Burroughs 计算机开发的 MCP（Master Control Program）。MCP 用 ALGOL 的一个变种语言编写。MIT 开发的 MULTICS 主要使用 PL/1 编写。针对 Prime 计算机的 Prime 操作系统使用一个专门的 Fortran 语言（a dialect of Fortran）编写。UNIX 操作系统、OS/2 和 Windows NT 主要使用 C。在原始的 UNIX 中（original UNIX），大约只有 900 行代码使用了汇编语言，其中大多是调度程序和设备驱动程序。

使用高级语言或至少是系统实现语言来实现操作系统的优点就像是用它们写应用程序那样：编写代码更快、更紧凑且更易于理解和排错。另外，改进的编译程序技术可以简单的通过重新编译来为整个操作系统改善生成的代码。最后，使用高级语言编写的程序更易于移植到其它的硬件上。（Finally, an operating system is far easier to **port**-to move to some other hardware-if it is written in a high-level language.）例如，MS-DOS 使用 Intel

8088 汇编语言编写。因此，它只能运行在 Intel 系列 CPU 上。

另一方面，UNIX 操作系统几乎全部使用 C 语言，它可以运行在很多不同的 CPU 上，这包括了 Intel 80X86、Pentium、Motorola 680X0、Ultra SPARC、Compaq Alpha 和 MIPS RX000。

反对者声称使用高级语言实现操作系统的主要缺点在于降低了（系统）速度并增加了存储器需求。虽然一个汇编语言专家程序员能够生成高效的小程序，但是对于大型的程序，现代的编译器能够实现复杂的分析并能对代码进行优化，生成优异的代码。现代的处理器拥有深流水线（deep pipelining）和多个功能单元，它们能处理复杂的相关性，这突破了人类智力的限制。（Modern processors have deep pipelining and multiple functional units, which can handle complex dependencies that can overwhelm the limited ability of the human mind to keep track of details.）

跟在其它系统中的情况一样，操作系统性能的提高主要在于更优的数据结构和算法，这要比优秀的汇编语言代码更重要。另外，虽然操作系统体积庞大，但只有小部分代码的性能是决定性的；内存管理和 CPU 调度几乎占了这些关键性程序的大部分。在系统编写完毕和能够正确工作后，可以确定哪些是瓶颈程序并以等价的汇编语言取代之。

为了确定哪些是瓶颈程序，我们必须能够监控系统性能。必须要添加用于计算的代码并且要显示对系统行为的测量数据。在许多系统中，操作系统通过产生系统行为列表来完成此任务。要把所有关注的事件以及他们的时间和重要参数记录到一个文件中。然后，分析程序处理这个日志文件以确定瓶颈和低效的程序。跟踪也能够帮助人们发现操作系统行为中的错误。

另一种方法是实时计算和显示系统性能。例如，一个计时器可以启动一个程序来存储当前的指令指针值。这样就会产生一个关于程序各部分的执行频率的静态图。这种方法可以使系统操作员熟悉系统行为并进行实时调整。

3.8 系统生成

我们可以为一个节点上的一台机器专门设计、编码、实现一个操作系统。但是，更普遍的，操作系统是为一类机器设计的，它们在不同的节点上，有不同的外设。那么，就要为每个具体的机器配置和产生系统，这个过程叫做**系统生成（SYSGEN）**。

通常通过磁盘或磁带发布操作系统。为了生成一个系统，我们使用一个特殊的程序。SYSGEN 程序从一个给定的文件中读取或向该系统的操作询问硬件系统的具体配置或直接检查硬件来确定有什么组件。必须要确定下列类型的信息：

•将使用什么样的 CPU？安装了什么选项（扩展指令集、浮点运算等）？对于多 CPU 系统，必须要描述每个 CPU。

•有多少有效存储器？有些系统自己检查内存容量。一个内存地址一个内存地址的试探，直到产生一个“非法地址”（访问）错误。这个过程定义了最大的合法地址，也就是可用内存的数量。

•什么设备有效？系统要知道如何为每个设备编址（设备号），要知道设备中断号、设备类型和型号以及任何特定的设备特性。

•需要哪些系统选项或要用到哪些参数值？这些选项或参数值可能包含了需要用几个多大的缓冲区、所需 CPU 调度算法的类型、支持的最大进程数，等等。

一旦确定了这些信息，就能够以各种方式来使用。作为一种极端的方式，系统管理员能够利用这些信息修改操作系统源代码，然后完全编译操作系统。通过数据声明、初始化、常量以及条件编译，产生一个专为这个系统的要求定制的操作系统版本。

当定制水平稍低时，系统描述能够产生一些表并从预编译库中选择模块。（At a slightly less tailored level, the system description can cause the creation of tables and the selection of modules from a precompiled library.）这些模块连接在一起形成生成的操作系统。这种选择允许库中包含所有支持的 I/O 设备的驱动程序，但只有需要的才被

链接到操作系统中。因为不需要重新编译，所以系统能够更快生成，但是可能会导致系统过分的通用（general）。

另一种极端情况，可以构建一个完全使用表格驱动（table driven）的系统。所有的代码总是系统的一部分，选择发生在运行时，而不是在编译或链接时。系统生成包含了简单的创建用于描述系统的表格。大多数现代操作系统以这种方式构建。Solaris 在安装时进行一些系统配置，在启动时执行一些。系统管理员可以使用配置文件来调整系统变量，但是内核自动进行硬件的配置。同样的，Windows 2000 在安装或启动时都不需要手动配置。一旦解决了磁盘布局和网络配置等基本问题，安装程序自动检测系统硬件并安装一个合适的生成系统。

这些方法的主要不同在于生成的系统的体积和通用性以及更改硬件配置的简易性。考虑到为了支持一个新图形终端或另外的磁盘驱动器而更新系统的代价。要把这些变化的频率（是否经常）和相应的代价相权衡。

生成一个操作系统之后，必须要使硬件使用它。但硬件是怎样知道内核的位置，或者是怎样载入内核呢？载入内核启动计算机的过程被称为**引导**系统。大多数计算机系统有一段存储在 ROM 中的代码，这被称为**引导程序**或**引导装入程序**。这段代码能够定位内核，把内核载入主存储器并使它开始运行。有些计算机系统（如 PC）使用两步进程，一个简单的引导装入程序从磁盘中装入一个更复杂的引导程序，然后由后者载入内核。在 14.3.2 节和附录 A 中讨论系统启动。

3.9 摘要

操作系统提供了许多服务。在最底层，系统调用允许一个正在运行的程序直接向操作系统提出请求。在一个更高的层次，命令解释程序或 shell 提供了用户不用编写程序（用户通过输入命令）就可以发出请求的机制。在批处理方式下命令可能来自于文件，在交互式或分时模式下直接来自于键盘输入。为了满足许多通用的用户请求提供了许多系统程序。

依据请求的层次，请求的类型也很不同。系统调用层必须要提供基本的功能，如进程控制以及文件和设备管理。高层的请求（由命令解释程序或系统程序满足的）被转换为一个系统调用对列。可以把系统服务分为如下几类：程序控制、状态请求和 I/O 请求。可以认为程序错误是隐性的服务请求。

在定义过系统服务之后就可以开发操作系统结构了。这需要各种表来记录定义计算机系统和系统作业的信息。

新系统的设计是主要的任务（major task）。必须要在设计开始之前明确定义系统目标。它们是将来选择各种算法和策略的根据。

因为操作系统体积庞大，所以模块化是非常重要的。我们认为使用层次化设计或微内核技术来设计系统是非常不错的。虚拟机概念采用了层次化设计，把操作系统内核和硬件都当作硬件来对待。甚至可以把其它的操作系统装载到这个虚拟机的顶端。

因为 JVM 为 Java 程序抽象了底层的系统（JVM abstracts the underlying system to the Java program），提供了一个系统无关的接口，所以实现了 JVM 的操作系统可以运行所有 Java 程序。

在整个操作系统设计流程中，我们必须要小心分离 policy 决定和实现细节（或 mechanism）。如果要在日后更改 policy 决定，那么这种分离允许最大化的灵活性。

现在几乎总是使用系统实现语言或高级语言来实现操作系统。这改善了它们的实现、维护和可移植性。要为特定的机器配置创建一个操作系统，我们必须要完成系统生成。

词汇

Java 解释器：Java interpreter
Java 虚拟机：Java virtual machine, JVM
JIT：just-in-time
参数值：parameter value
层次化设计：layered approach
磁盘布局：disk layout
磁盘空间：disk space
磁盘系统：disk system

错误状态：error condition
单步执行：single step
底层计算机：underlying computer
调度程序：scheduler
调试系统：debugging system
读卡机：card reader
多路存取：multiaccess
多用户：multiuser
分层：layered
浮点运算：floating-point arithmetic
共享存储器：shared memory
固定字段：fixed field
机器配置：machine configuration
计算机供应商：computer vender
计算机间的通信：intercomputer communication
计算业务：computing service
假脱机：spooling
兼容性：compatibility
进程名：process name
控制语句：control statement
垃圾回收：garbage collection
类检查器：class verifier
类装载器：class loader
流水线：pipeline
模块化：modularity
瓶颈：bottleneck
软件工程：software engineering
上溢：overflow
设计流程：design cycle
数据定义：data declaration
条件编译：conditional compilation
通讯网络：communication network
微内核：microkernel
文本文件：text file
文件夹：folder
物理机器：physical machine
系统工具：system utility
系统兼容性：system compatibility
系统开发时间：system-development time
系统生成：system generation, SYSGEN
系统行为：system behavior
系统验证：system verification
下溢：underflow
消息传递、报文传送：message passing
消息传递模型：message-passing model
行式打印机：line printer
虚拟机：virtual machine
虚拟机监控程序：virtual machine monitor
虚拟内存：virtual memory
引导：booting
引导程序：bootstrap program
引导装入程序：bootstrap loader
应用程序：application program
应用程序接口：Application Programmer Interface, API
用户空间：user space
语言规范：language specification
原生指令：native instruction
源代码：source code
指令指针：instruction pointer
主机名：host name
状态请求：status request
字节码：bytecode

Pyos 中软盘驱动、DMA 及文件系统的实现（上）

哈尔滨工业大学 并行计算实验室
谢煜波（xieyubo@126.com）

前言

Pyos 的软盘驱动及文件驱动实际上已经完成了很长一段时间了，但由于此后一直忙于学务及实习，至到今天才完成这篇实验报告，实在是让大家久等了。

软盘驱动实际上是一个比较复杂的系统，涉及到对软驱控制器及 **DMA** 控制芯片的操作以及错误处理。操作的命令、步骤及执行的结果都比较繁杂。从笔者来讲，希望此篇报告如同前几篇报告一样，在描述上能尽量的简洁易懂一些，因此，几易此稿，最终成此模样，但还是觉得其中有太多的地方直得推敲，希望审阅此篇报告的各位读者能把你们的意见及时的告诉我。

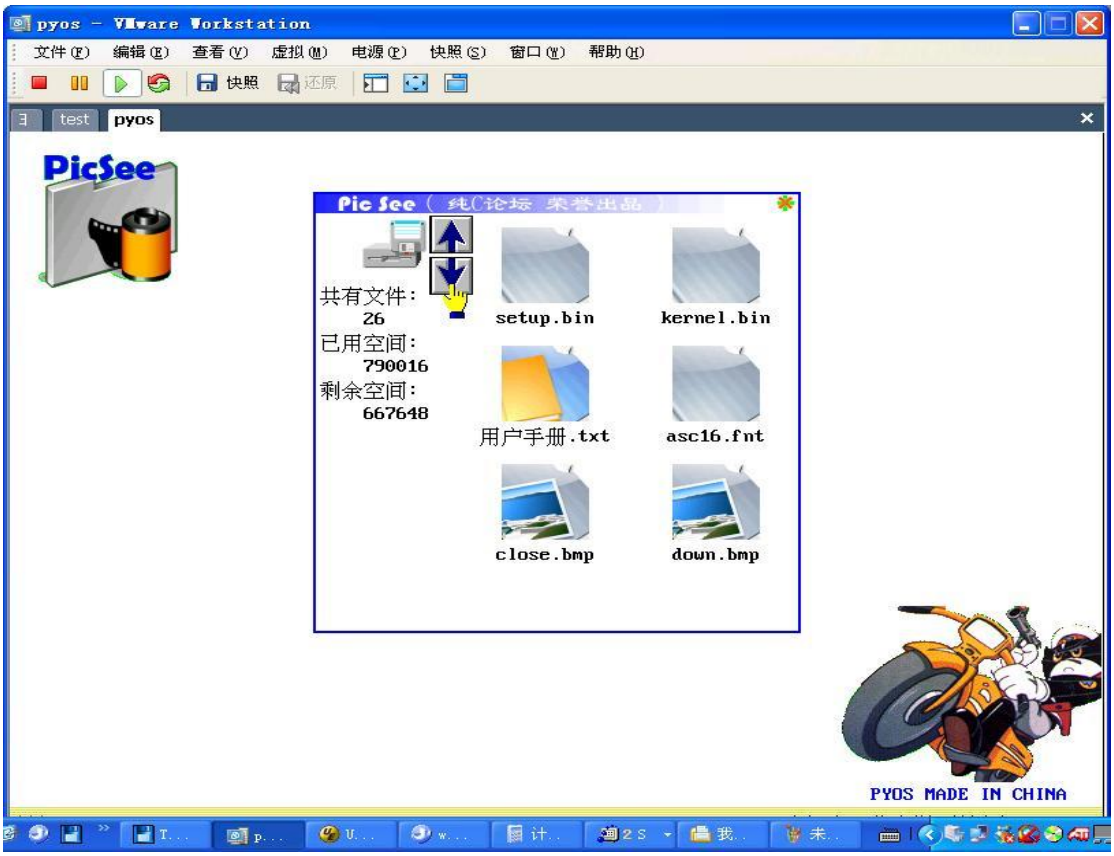
这篇报告谈的是对硬件的操作，对硬件操作一般都比较麻烦，但麻烦却不一定很困难，相对而言，单纯的对硬件的操作，基本上涉及不到什么算法，因此也没有很高的难度，很多情况下都是按照硬件的说明书进行，对与本实验而言，主要就是软驱控制器的说明书及 **DMA** 控制芯片的说明书，也就是本文所列的参考文献。它们阐明了我们应当怎样去操作相应的硬件，第一步做什么，第二步做什么，因此，我们只需按照说明书上所阐明的步骤进行就可以了。在阅读本篇报告之前，我建议你能先大致的，很快速的翻看一下本文的参考文献，这样会非常有助于你很轻松的阅读本文。

这篇报告与前几篇报告一样，都属于 **pyos** 操作系统实验系列中的一篇实验报告。同前几篇一样，本篇将详细结合 **pyos** 的源码，介绍一下怎样从编写操作系统的角度去实现一个简单的软盘驱动及文件系统。此份实验报告分为上下两篇，上篇介绍软盘驱动及 **DMA** 控制的实现，下篇将详细介绍一下 **FAT 12** 文件系统的实现，另外将介绍一下 **pyos** 自行实现的一种自己所专有的文件系统格式。

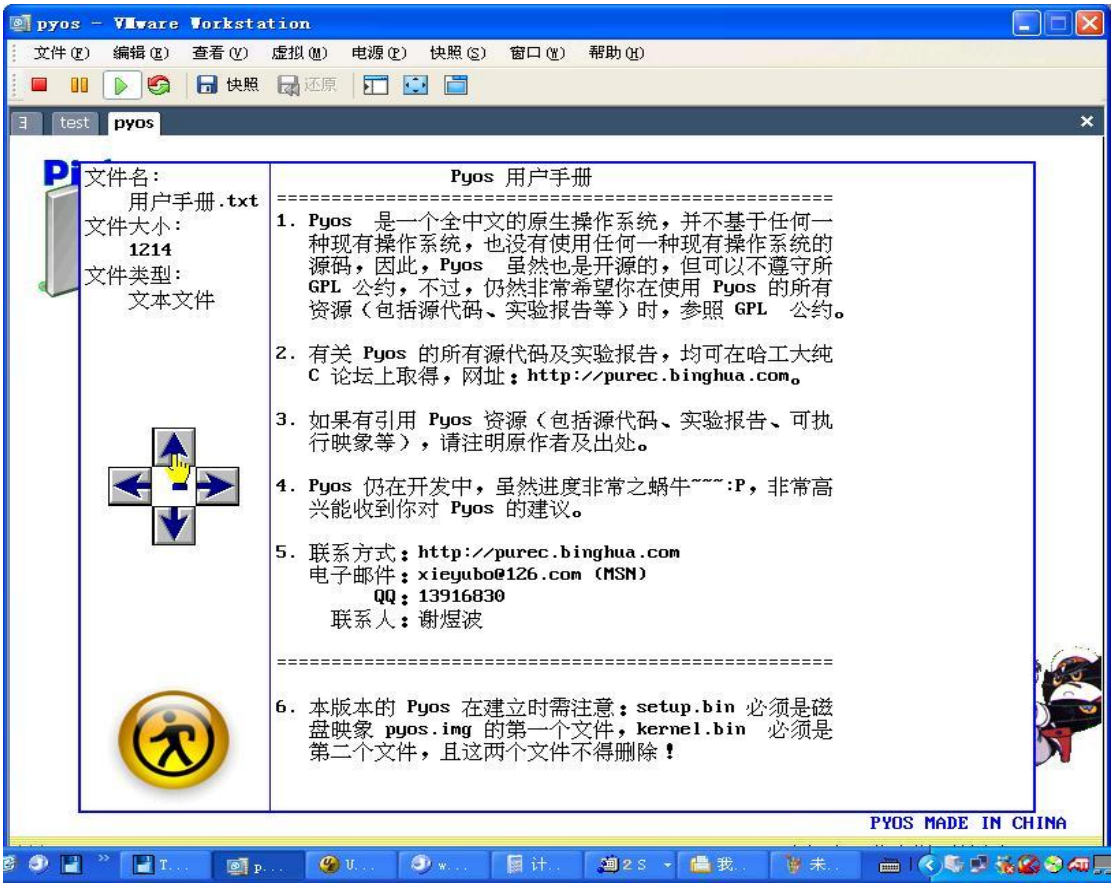
我们先来看看这次实验的效果图，你可以了解一下，通过这次实验，你可以达到的效果。


pyos - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟(M) 电源(P) 快照(S) 窗口(W) 帮助(H)
快照 还原
test
pyos
Pyos 中文操作系统
正在启动
@哈工大@纯C论坛@

（启动界面）


pyos - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟(M) 电源(P) 快照(S) 窗口(W) 帮助(H)
快照 还原
test
pyos
PicSee
Pic See (纯C论坛 荣誉出品)
共有文件:
26
已用空间:
790016
剩余空间:
667648
setup.bin
kernel.bin
用户手册.txt
asc16.fnt
close.bmp
down.bmp
PYOS MADE IN CHINA

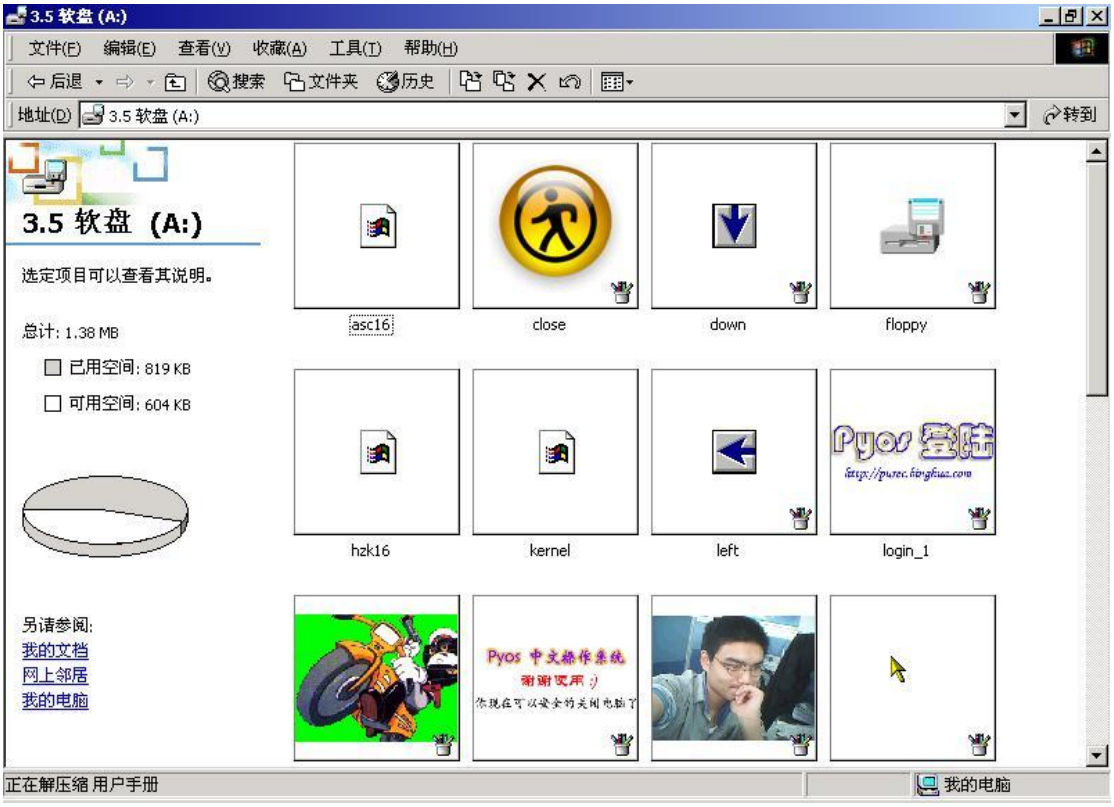
（文件管理器）


pyos - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟(M) 电源(P) 快照(S) 窗口(W) 帮助(H)
快照 还原
test
pyos
文件名:
用户手册.txt
文件大小:
1214
文件类型:
文本文件
Pyos 用户手册
1. Pyos 是一个全中文的原生操作系统，并不基于任何一种现有操作系统，也没有使用任何一种现有操作系统的源码，因此，Pyos 虽然也是开源的，但可以不遵守所 GPL 公约，不过，仍然非常希望你在使用 Pyos 的所有资源（包括源代码、实验报告等）时，参照 GPL 公约。
2. 有关 Pyos 的所有源代码及实验报告，均可在哈工大纯 C 论坛上取得，网址：http://purec.binghua.com。
3. 如果有引用 Pyos 资源（包括源代码、实验报告、可执行映象等），请注明原作者及出处。
4. Pyos 仍在开发中，虽然进度非常之蜗牛~~~:P，非常高兴能收到你对 Pyos 的建议。
5. 联系方式：http://purec.binghua.com
电子邮件：xieyubo@126.com (MSN)
QQ：13916830
联系人：谢煜波
6. 本版本的 Pyos 在建立时需注意：setup.bin 必须是磁盘映象 pyos.img 的第一个文件，kernel.bin 必须是第二个文件，且这两个文件不得删除！
PYOS MADE IN CHINA

（文本浏览）


pyos - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟(M) 电源(P) 快照(S) 窗口(W) 帮助(H)
快照 还原
test
pyos
文件名:
me.bmp
文件大小:
153656
文件类型:
位图文件
长:
320
宽:
240
PYOS MADE IN CHINA
关机
哈尔滨工业大学 谢煜波 2004
你尚未安装 VMware Tools.

（图片浏览）



（同 Windows 系统共享文件）

所有的实验代码及文档均可以在哈工大·纯 C 论坛（http://purec.binghua.com）中下载，也可以在《纯 C 论坛·电子杂志》中找到相应内容。部份参考文献也可以在上面找到。纯 C 论坛·BBS 讨论区上还专门建有一个“操作系统实验讨论专区”，欢迎大家常去逛逛。

还是那句老话，由于受水平所限，本文中很多地方都值得推敲，甚至有可能是错误的描述。因此，绝对不要以本文所述为准，本文只希望能对不了解而又想了解的朋友有些许帮助，如果您发现了本文的不足，请您及时来信告诉我，我在此先行向您表示感谢！

本次实验得到了 AlAdDin（摩洛哥）的大力支持及帮助，通过电子邮件及 MSN ，他很热心的帮我解决了一些技术上的问题，没有他的帮助，这次实验根本不可能如此顺利的成功,在此谨表示我最真诚的谢意！

此外，哈尔滨工业大学计算机学院教师、《纯 C 论坛·电子杂志》编委孙志岗老师，在百忙之中审阅了此稿，在此，谨对孙老师表示由衷的感谢。

我的电子邮件是 xieyubo@126.com，非常高兴能收到您的反馈，我也很乐意与您对某些问题进行交流，但请不要让我帮您寻找程序中的 Bug。

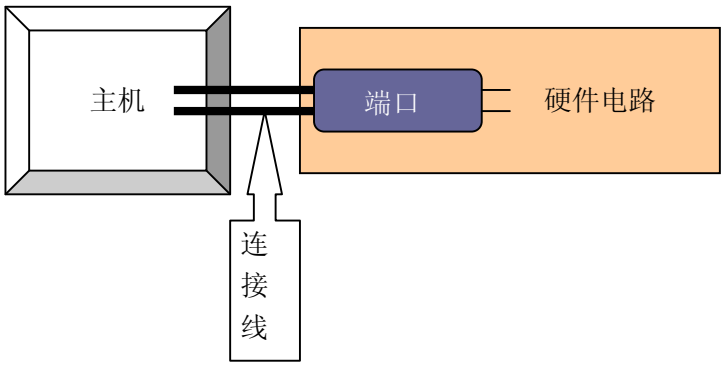
一、操作系统中软盘驱动工作方式的概述

随着 U 盘的流行，软盘在现在的个人电脑中已显得越来越不重要，很多新出的电脑及笔记本上根本就没有软盘驱动器，不过对于学习编写操作系统来说，学习一下怎么编写软盘驱动还是有实际意义的，从其中我们可以学习到操作系统（软件）控制硬件的基本方法及操作系统（软件）与硬件进行交互的基本思路，这对我们更好的理解操作系统的行为，甚至编写操作系统，无疑有巨大作用。

在前几篇报告中，我们已经介绍了操作系统对其它一些常用硬件的控制，比如说鼠标、键盘等。虽然硬件种类不同，但它们有一点是相同的，就是它们都是通过中断和端口与操作系统进行交互。

中断可能大家都非常熟悉了，当操作系统发出一个命令给硬件后，硬件就去执行这个命令，由于这个命令的执行需要时间，为了把这段时间也利用起来，操作系统可以切换 CPU，让其去进行另一项工作，当硬件完成了操作系统先前发出的命令后，硬件会发出一个中断请求信号中断 CPU 对当前任务的执行，然后，CPU 切换到相应的中断处理程序中去执行相应的操作，这个中断处理程序会检测硬件的状态，并根据硬件状态完成一些后续的处理。这方面的内容在任何一本讲计算机原理方面的书中都会有详尽的描述，本系统的实验报告中也有一篇——《保护模式下的 8259A 芯片编程及中断处理探究》对此进行过描述，因此，这里就不再多说了。

下面我们来看一看这个过程中所涉及到的一个重要概念“端口”。上面我们说过，操作系统会发一个命令给硬件，但这个命令是怎么发给硬件的呢？是通过什么发给硬件的呢？在 IBM PC 结构中，这是通过硬件的“端口”进行的。不光是操作系统发送命令及数据给硬件需要端口，硬件把结果数据回送给操作系统也需要通过端口进行。从一个易于理解的角度上去描述，“端口”其实是一个输入输出系统，它包括一些寄存器，用来暂时存放操作系统送来的数据或者需要送给操作系统的数据。除此之外，它还包括一些控制电路，这些控制电路用来完成控制逻辑，“端口”其实是两种元件（这两种元件可以是软件与硬件，也可以是硬件与硬件，但一般软件与软件之间不这么称呼）之间的一种输入与输出的接口界面，如下图所示：



上图就是一个端口的示意图，图中的“连接线”就是我们常说的数据线等。端口可以做在外围的硬件电路一边，也可以做在主机的主板上，也可能两边都存在。

实际上，系统中有很多硬件，有的硬件甚至有两三个端口，那么系统怎么来分辨这些不同的端口呢？同分辨不同的内存单元一样，系统给每一个端口都分配一个唯一的地址，这又存在两种编址方式，一种是把端口与内存做为一个整体统一分配地址，这称之为“统一编址”，另一种是把端口与内存分开，端口与内存各自独立的分配地址，两种地址间没有什么关联，这称之为“独立编址”，在 IBM PC 中，采用的是独立编址，系统也提供了专门的机器指令来访问这些端口地址，比如 IBM PC 汇编中的 in 及 out 指令。

对软驱的操作其实上就是对软驱控制器进行操作，操作的方式就是发送命令以及从软驱控制器接收结果数据，这都是通过端口进行的。在后文中多次提到“向软驱控制器发送 XX 命令”，其实就是指把“XX”命令（也就是一个数值）通过 in 指令写到软驱控制器的相应端口中；“从软驱控制器读取返回结果”其实也就是指通过 out 指令，从软驱控制器的相应端口读取数值。希望大家能在脑海中对此留有印象，以免在阅读后文时，对这些说法感到疑惑。

这里，还要强调一点，很多“命令”或者“结果数据”都有好几个字节，特别是一些带有参数的命令，而 in、out 这些读写端口的指令一次只能发送或读取一个字节，那应当怎么办呢？遇到这种情况的时候，我们只需要把这些多个字节按着顺序一个一个的发送就行了，前面说了，“端口”其实是一个很小的输入输出系统，它自身就带有一定的控制逻辑，这些控制逻辑会自己对这种情况进行处理，我们无须对此进行额外的处理。

OK，了解了怎么发送命令和接收结果之后，我们就可以准备正始开始编写我们的软盘驱动了。

二、软驱操作步骤

2. 1 floppy_read()

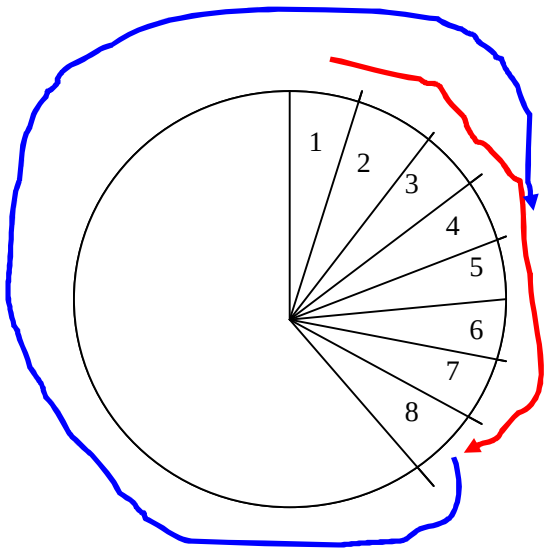
前言中我们已经说过，对软驱的操作最重要的就是根据软驱控制器的硬件说明书上所阐明的步骤一步一步的进行，现在我们就来看看都主要有些什么步骤，请看 pyos 中相应源码：

```
/* 软驱读取函数一次读一扇区 */
void floppy_read( unsigned char sector , unsigned char cylinder , unsigned char head )
{
  //设置中断标志，在等待中断时使用
  floppy_called = 0 ;

  //启动软驱马达
  floppy_start_motor() ;

  //校准磁头，使其定位于正确位置
  floppy_recalibrate() ;

  //打开 dma 通道
  dma_open_dma_channel( 2 , floppy_buffer_address , 512 , DMA_READ_MODEL ) ;

  //发送读取扇区的命令
  floopy_send_read_commmand(head, cylinder, sector);

  //等待中断产生
  floppy_wait_for_interrupt() ;

  //读取命令执行的结果
  floppy_read_result_of_read_command();

  //停止驱动器
  floppy_stop_motor() ;
}
```

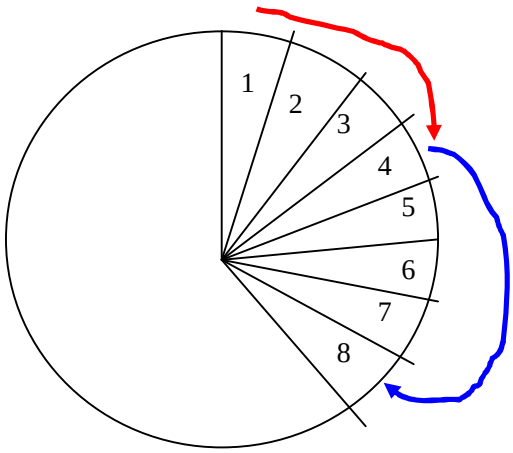
上面就是 pyos 系统中软驱驱动的一个主要函数——读扇区函数，它会将用户指定的扇区的数据读到由 floppy_buffer_address 所指定的内存块中。从这里我们可以发现这样一个事实，对于软驱驱动来讲，它只负责读（写）数据，而对数据的格式或者说内容什么的，根本就毫不知情，这块数据是文件还是图形还是其它什么东东，这是由更高一层的系统管理的，比如说文件系统，它知道哪个扇区存的是什么数据，由它命令软驱驱动把数据原封不动的读上来，然后再进行处理。软驱驱动实际上就是一个搬运工，把软驱上的一块数据搬移到

指定的内存中就行了。

呵呵，这里我们来考虑这样一个比较有趣的问题。假设一个文件被分成了三个部份，第一部份放在第 1 扇区，第二部份放在第 8 扇区，第三部份放在第 4 扇区，那么你就需要连续调用三次上述函数，先从第 1 扇区取出文件的第一部份，然后在从第 8 扇区取出第二部份，最后从第 4 扇区取出文件的第三部份，在这一过程中，软驱控制器会控制磁头先运动到第 1 扇区上，然后再运动到第 8 扇区上，最后再折回来，运动到第 4 扇区上读取，我们来看看这条运动路径（我们假设磁盘只向一个方向转动）：



这个图实在是太难看了一点（Sorry,我的美术细胞估摸等于 0L），不过希望还是能表达清楚意思，即从 1 到 8 再到 4，我们整整绕了一圈！如果我们把次序改一下，先读第 1 扇区的第一部份，再读第 4 扇区的第三部份，最后再去读第 8 扇区的第二部份，这时的磁头的运动路径如下所示：



我想这样的效果是非常显著的，第二次的运动路径比上一次的运动路径少了整整一圈！显然，这极大的提高了软驱的读取性能。

从上面可以看出，都是同样的软驱，都调用的是同一个函数，都调了同样多次，但就是由于调用的顺序不同就会对性能产生非常大的影响，这就是算法或说理论的威力。一个好的操作系统必须有非常良好而高效的算法，而要想写出良好的算法就必须有很扎实的理论基础，这就需要大量的阅读一些操作系统的经典理论教材，需要在上课的时候多多学习、多多思考、多向老师请教，实践与理论是需要并重的，只重理论或只重实践都不

是正确的态度，任一方面的缺失都会限制我们更进一步的发展。

磁盘的调度产生过很多非常有名的算法，其中有一个称为电梯算法的算法非常经典，还记得我们乘坐的那种箱式电梯吗？如果现在电梯正从三楼走向四楼，你在一楼呼叫电梯，而在你之后，又有一个人在七楼呼叫电梯，虽然是你呼叫在先，但电梯到达四楼后还是会先去七楼响应七楼的呼叫，然后才向下运行到一楼响应你的呼叫，也就是说，它总是最先响应与它运行方向一致的请求，而不是按照先来先服务的方式运行的，这一点上，它并没有讲究一个先来后到的关系。

具体采用什么算法，这是一个操作系统设计及编写者必须考虑的问题，把这算法设计到哪一个层面，这也是一个设计者及编写者必须考虑的问题之一。比如，我们可以把它做到文件系统中，因为前面说了，只有文件系统才知道一个文件的具体格式，才知道我这个文件是放到什么地方的，因而，我可以在文件系统内部进行调度，安排以怎样的顺序发出读写请求，而软驱驱动就不用去考虑调度问题了，因为来的请求都是已经被文件系统安排好了的，故而肯定是顺序恰当的，软驱驱动直接执行就行了。但也可以把它做到软驱驱动中，文件系统不需要去调度，只需按文件自身的顺序发出读写请求，然后由软驱驱动去根据这些请求，运行一个调度算法，按最恰当的次序响应这些请求。这样做还有一个另外的好处，即不光是文件系统来的请求，有可能是其它系统来的请求，它都可以对其进行统一调度。

上面两种策略，很难说哪种好哪种不好，也或许你想在两边都做上这种算法，因为你的文件系统或是软驱驱动需要有很大的独立性及灵活性。比如我把这个文件系统安装到另外的操作系统中，我就无需考虑新的操作系统的软驱驱动是否会运行这样的调度算法，反正是由我的文件系统调度的。这对于软驱驱动来说也是一样的，它无需考虑新的操作系统中的文件系统是否会有这样的调度算法，反正自己会调度的。这样看来似乎是一种两全其美的选择，其实不然，因为如果把它们配在一起，就会使读一次文件，但却运行了两次这样的调度算法，这对性能是一个不小的影响。

上面谈的都是设计方面的问题，当系统真的很大的时候如何去设计是一个非常重要又非常困难的工作，它往往需要从多方面把握，综合考虑各种因素、各种优劣，使设计达到一种平衡，这种平衡不一定是最好的，但一定是合适的。

不过，**pyos** 没有考虑这么多，它的软驱驱动非常简单，就是按顺序执行你的读取命令就完了，**pyos** 不考虑性能，因此，它不会是一个良好的可实用的系统。

2. 2 floppy_start_motor()

呵呵，上面扯得有点远了，现在我们就来看看软驱读取操作的真正流程！

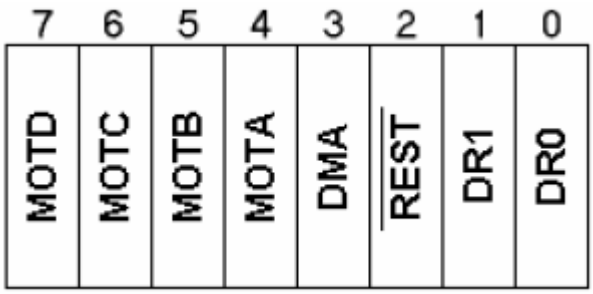
首先，它把一个中断触发标志（`floppy_called`）置为 `0`。前面我们已经说过，硬件在执行完一个命令的时候，会引发一个中断，CPU 会去执行一个相应的中断处理程序，我们在这个中断处理程序中会把此标志置为 `1`，这样，我们的软驱驱动就知道有中断发生了，即软驱已经执行完一个命令了。请见如下的中断处理函数：

```
/* 中断处理函数 */
void floppy_handle_for_floppy_interrupt()
{
  floppy_called = 1 ;
}
```

接着前面的主程序往下走，它调用 `floppy_start_motor()`这个函数，这个函数可以将驱动器的马达置成“使能”状态，下面就是这个函数的相应代码：

```
void floppy_start_motor()
{
  /* 启动驱动器 */
  io_write_to_io_port( 0x3f2 , 0x1c ) ;
}
```

`io_write_to_io_port()`这个函数是在 **io.c** 文件中定义的（这里提一下 **pyos** 中的一个命令规则，每个变量名或函数名的第一小节都指出了此变量或函数所在的文件，希望这个规则会对大家阅读 **pyos** 的源代码有所帮助），它的作用就是把 **0x1c** 这个数写到 **0x3f2** 这个端口里。在前面我们已经对端口有过很详细的描述了，**0x3f2** 是这个端口的端口地址，那么这个端口是干什么用的呢？这就需要我们查阅相应的硬件资料了，下面，就是本文参考资料中所表示出的此端口的结构：



从硬件资料上我们可以知道，此端口称作：数据输出寄存器（**Data Output Register**， **DOR**），我们来看看它的各个二进制位都是表示什么含义的（这部份内容同样是从硬件资料上获得的）：

DR1、DR0 位：用来选择驱动器（一个软驱控制器可以控制四个驱动器），它可以是如下的值：**00：**驱动器 **0**（**A**）；**01：**驱动器 **1**（**B**）；**10**：驱动器 **2**（**C**）；**11**：驱动器 **3**（**D**）

REST 位：如果是 **0** 则表示复位软驱控制器。

DMA 位：如果是 **1** 则表示使用 **DMA** 进行传输（关于 **DMA** 会在后面描述）。

位 4 到位 7：用来表示 **A**、**B**、**C**、**D** 四个马达是否启动。如果有某位为 **1**，则表示相应的马达启动，否则表示停止相应的马达。

由于现在我们假设计算机中只有一个软驱（**A**），故而，我们将 **DR1**、**DR0** 置为 **00**，位 **4** 到位 **7** 置为：**0001**；使用 **DMA** 进行传输，因此，**DMA** 位置为 **1**；无需复位软驱控制器，故而 **REST** 位置为 **1**，因此，我们可以得到这样一串二进制数：**0001 1 1 00**，即 **0x1c**，这就是我们传到这个地址为 **0x3f2** 的端口的数据，现在你知道`floppy_start_motor()`这个函数都做了些什么事情了吧。

2.3 floppy_recalibrate()

继续读前面的程序，它调用了`floppy_recalibrate()`这个函数。这个函数用来重新校准磁头所在的位置。因为在你放入磁盘后，磁头对着磁盘上的什么位置是随机的，软驱控制器并不知道，这样它就无法将磁头准确的定位到指定的扇区上，因此，在使用前，我们需要先进行磁头位置的校准操作，这又是一个比较复杂的操作，我们还是先来看看此函数的实现代码吧：

```
/* 校准磁头 */
void floppy_recalibrate()
{
  floppy_send_byte_return_is_ok( 7 ) ; // 发送校准命令
  floppy_send_byte_return_is_ok( 0 ) ; // 发送参数

  floppy_wait_for_interrupt() ; // 等待中断发生

  floppy_send_byte_return_is_ok( 8 ) ; // 发送获取中断命令
  unsigned char st0 ;
  floppy_read_byte_return_is_ok( &st0 ) ; // 读取 ST0
  unsigned char current_cylinder ;
```

```
  floppy_read_byte_return_is_ok( &current_cylinder ) ; // 读取 PCN
}
```

上面就是校准磁头的实际代码，它首先发送了“校准命令”，这个命令是一个多字节的命令，我们来看看这个命令的格式：

Bit Byte	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1
1	x	x	x	x	x	0	DR1	DR0

（此图来源于本文参考文献）

图中 DR1、DR0 同前面所介绍过的 DR1、DR0 一样，是用来选择驱动器的，00 表示选择 A 驱动器，忘记的朋友可以翻回前面看看。

软驱控制器支持很多命令，不同的命令有不同的格式，而具体有哪些命令，又具体对应着什么样的格式，每个命令又有一些什么样的参数，这都需要我们去阅读软驱控制器的硬件资料，建议各位朋友在阅读本文的时候打开本文所附的参考文献，随时查阅，对比阅读。

发送命令完成之后，软驱控制器就去执行命令去了，这个时候，系统调用了 `floppy_wait_for_interrupt()`这个函数来等待软驱中断的发生，下面就是这个函数的代码：

```
/* 检测中断函数 */
void floppy_wait_for_interrupt()
{
  while( !floppy_called ) ;
  floppy_called = 0 ;
}
```

这个函数非常简单，就是一个死循环，并在循环中检测 `floppy_called` 这个变量是否被置 1（前面已经说过，在磁盘的中断处理函数中我们将此变量置为 1），如果被置 1 则表明磁盘中断已经发生，于是终止循环并返回。这里可见 pyos 性能非常低的又一原因，它在此时，cpu 是处于原地等待中，其实更好一点的策略是在此时切换到另外的进程去执行，当磁盘中断发生后再切换回来运行，这就涉及到进程的调度。

比如说，我们可以这样去设计：A 进程调用软驱驱动读取一个扇区，于是，软驱驱动向软驱控制器发出命令，然后软驱驱动调用一个阻塞函数，把自己给阻塞了，由于是在 A 进程中调用的软驱驱动，所以这一阻塞就把 A 进程也给阻塞了。阻塞其实就是调用的操作系统的进程调度函数，此函数会把申请阻塞的进程放入一个后备的队列中排队（入队），然后，操作系统检查后备队列中是否有其它的进程（出队），如果有，不防假设此进程为 B 进程，操作系统就把这个 B 进程调出来并运行它。在 B 进程的运行过程中，软驱控制器可能完成了先前 A 进程通过软驱驱动发送的命令，于是，它触发一个中断，操作系统中断正在运行的 B 进程，然后转到中断处理程序中去执行，中断处理程序会把软驱中断的触发标志置为 1，然后中断返回，CPU 回到 B 进程中继续后续的执行，突然，操作系统发现 B 进程的时间片用完了，于是操作系统又把 B 进程扔进后备队列中排队，跟着，后备队列中的下一个进程出队了，这个进程可能就是先前进队的 A 进程，操作系统接着运行它，结果这一运行发现中断触发标志已经被置位了，这说明软驱控制器已经完成了操作，故而，A 进程就可以继续运行了。

不过需要注意这样一个有趣的问题，假设，后备队列中有一百个进程在排队，因此，就算是中断触发了，但 A 进程也要等前面的一百个进程都出队完成了之后，才有可能被操作系统调度到，这才又有可能再继续运行。如果这是一个很急的任务，或许也就是我们常说的一个“实时任务”，需要我们“实时”处理，那么这样的算法显然是不行的，不过我可以进行一下修改，比如说，在软驱驱动调用的时候，把当前调用的进程的进程号，这里就是 A 进程的进程号给保存起来，这样，当中断发送时，操作系统就可以跟据这个进程号马上在后备队列中查找到是哪个进程在等待此中断，然后把它立刻提到后备队列的队头，以便下次紧接着就能运行它，当然也

可以直接就把它调出来运行，而不用再到后备队列中去排队了。具体采用什么方式，这同前面谈的对磁盘调度算法的设计一样，需要我们的设计人员综合分析。现代的操作系统中，不光有后备队列，还有阻塞队列等。操作系统发展至今，出现了很多经典的进程调度算法，想在在此方向有所做为的朋友，或说想编写出一个实用的操作系统的朋友，可以多找点这方面的理论性文章读读，这里又是一处理论与实践的结合。

呵呵，又扯远了，我们接着往下看。下面，系统调用了 `floppy_send_byte_return_is_ok( 8 );`这个命令去获取中断结果，这里其实就是读取软驱控制器执行命令后返回的结果，当然我们需要了解这些结果都表示什么意思，然后分析一下看看命令是否已经正确执行了，如果没有正确执行，那么我们是否需要对其进行什么处理？是重发一次，还是直接返回一个错误编号给调用者由调用者进行处理？这都是需要仔细设计的。错误处理是系统设计中很关键的一部份，特别是像操作软驱这样的硬件，永远不要认为你的操作的结果会永远正确。呵呵，不过，pyos 为了简化设计，突出矛盾，它自认为所有的操作均正确 J，因此，pyos 在此处并没有进行任何的错误处理，而只是简单的把系统的返回值读取出来了。或许有朋友会认为，既然不处理，那为什么还要去读呢？其实在对硬件进行操作的时候，我们需要注意的就是在很多时候，如果硬件有返回值，那么我们一定要去读，因为硬件会一直保存这个值，只到你读走为止，你要是不读，硬件就无法完成后面的新的操作了。

2.4 dma_open_dma_channel()

OK，校准磁头的函数我们分析完了，现在应当回到最初的函数中继续分析了。在校准磁头完毕之后，我们调用了 `dma_open_dma_channel()`这个函数来打开 DMA 通道。这里先来详细介绍一下传输的 DMA 机制。

我们先来想象一下，正常情况下或说原始情况下，软盘驱动是怎么工作的：首先操作系统通过软驱控制器的端口发送命令，然后，软驱控制器控制软驱从磁盘上读取数据，数据读到之后，软驱控制器通过中断请求向 CPU 发送中断请求信号，CPU 中断现行处理程序，然后从软驱控制器的相应寄存器端口取得数据。乍一看，这是一个近乎于完美的工作过程，然而，让我们仔细的想想，我们就能发现其中的不妥之处。

我们知道，每一个端口寄存器的容量是有限的，在 IBM PC 机中它就是一字节，即每次你用端口命令（in、out）只能写入或读取一字节的数据。我们知道我们从软盘读取的数据量通常是几十字节，甚至几 K 字节，假设我们现在要从软盘中读出一个文件，而这个文件的大小是 1K 字节，那么如果按照上面的方式，软盘控制器就需要发出 1K 次中断请求，相应的，操作系统也需要响应 1K 次中断请求才能完成这个文件内容的读取，这将是非常花费时间的。那有没有什么办法可以解决这个问题呢？有！这就是 DMA。

DMA 是直接存储器访问（Direct Memory Access）这一术语的缩写。顾名思义，通过 DMA，外设与内存之间交换数据就不再需要通过 CPU，即先前的中断方式进行了。现在，假设我们启用了 DMA，那么软驱控制器就会通过 DMA 将指定大小的数据块传送到指定的内存块中，仅仅在传输完成之后，才向 CPU 发出一个中断请求信号告诉 CPU 数据已经读取完成了。DMA 控制芯片会完成将软盘的数据输出到内存中的一系列控制逻辑，而无需 CPU 费心，CPU 在此时完全可以去完成其它的任务，这显然大大的提高了 CPU 的利用率。

在系统中有专门的 DMA 控制器来控制对 DMA 的使用。因此，在使用 DMA 的时候，也即在我们的软驱驱动中，需要完成对 DMA 控制器的设置操作。DMA 控制器有很多所谓的通道（channel），常用的 8 位 DMA 有 4 个通道，每一个通道有三个端口（寄存器）用来支持对这个通道的使用，这三个端口（寄存器）分别是：

- 页寄存器：用来指明是使用的那一块内存（下面对此将会有详细描述）
- 偏移量寄存器：用来指明页内的偏移量
- 计数器（数据长度寄存器）：用来指明所传送的数据的长度

它们各自的端口地址与各通道的匹配情况如下表所示：

通道号	页寄存器地址	偏移量寄存器地址	计数器地址
0	0x87	0x0	0x1
1	0x83	0x2	0x3

2	0x81	0x4	0x5
3	0x82	0x6	0x7

通过软件选择使用哪一个通道并不是完全随意的，而是由系统的整体结构决定的，比如你的外设本来就是连到第 1 通道的，你为此外设编写的软件（驱动）当然就不能使用别的通道而应当同外设所选择的通道一致。需要知道的是，软驱控制器默认选择的是 2 号通道。

下面我们来看看前面提到的“页寄存器”、“偏移量寄存器”这几个寄存器具体的作用。还记得实模式下，内存是被分成一段一段的，每一个内存地址都通过段寄存器及偏移量进行访问的吗？与之类似，DMA 控制器也将内存分为一段一段的，每一段有 64K。不知大家是否还记得在 8086 中，我们能访问的最大内存地址为 1M，DMA 控制器就把这 1M 的地址分为 1M / 64K = 16 页，页号与地址的对应关系如下表所示：

页号	址址	页号	地址
0	0000:0000~0000:FFFF	1	1000:0000~1000:FFFF
2	2000:0000~2000:FFFF	3	3000:0000~3000:FFFF
4	4000:0000~4000:FFFF	5	5000:0000~5000:FFFF
6	6000:0000~6000:FFFF	7	7000:0000~7000:FFFF
8	8000:0000~8000:FFFF	9	9000:0000~9000:FFFF
A	A000:0000~A000:FFFF	B	B000:0000~B000:FFFF
C	C000:0000~C000:FFFF	D	D000:0000~D000:FFFF
E	E000:0000~E000:FFFF	F	F000:0000~F000:FFFF

可见，这与我们在实模式上非常熟悉的段基址加偏移量的访问模式是非常相近的。不过从上面也可以看出：第一，我们需要在最初的 1M 内存中划出一块内存来给 DMA 使用，而不能使用地址大于 1M 的内存了；第二，DMA 每一次传输不能超过 64K，而且不能跨越页边界，即你不能在一次传输中把一部份内容传到 A 页中，而另一部份内容传到 B 页中，每次传输只能在一页中进行操作。

现在，我们假设我们的软盘上有一个 65K 的文件，需要传到 2M 内存处，那么我们应当怎么传呢？

首先根据规则，DMA 只能传输到最初的 1M 内存中，但我需要传到 2M 内存处，这应当怎么处理呢？对于这个问题，我们可以先在 1M 内存中划出一块，比如就划出 DMA 第 1 页，即 10000~1FFFF 这块内存，用来专给 DMA 做临时存储数据用。当第一次传输完成后，DMA 通过中断通知操作系统，操作系统把 10000~1FFFF 这块内存中的数据搬移到 2M 内存处，然后重新设定 DMA 参数，启动 DMA 做第二次传输，等传完后再次把数据搬移到相应位置就可以了。

现在，来考虑第二个问题。由于每次 DMA 操作最大的传输数据量是 64K，那么现在我们这个 65K 的文件，假设需要从内存写入到软磁盘中，我们一次最多能传输 64K，因此，我们需要分成两次传输，第一次传输 64K，第二次传输余下的 1K。但这个时候可能又出现这样一个问题：现在假设第一次操作系统传了 64K 进到软盘中，但在传余下的 1K 的时候出错了，显然，操作系统必须恢复原来磁盘中被第一次所传的 64K 的数据所覆盖的东东。也许这听起来很复杂，但举个简单例子你就会明白了。比如用户通过你的操作系统用内存中的一个文件去覆盖磁盘中的一个文件，而这个时候覆盖出错了，那么磁盘中原来的那个文件应当还在，而不能是被你覆盖了一半的文件。因此，你在出错后必须进行恢复操作，也即必须进行错误处理。

再从另一个方面考虑一下问题，假设这个文件是需要从软盘读到系统内存中，这里我们假设有两种读法，一种是先读 64K，再读留下的 1K，或者先读 1K，再读留下的 64K。假设第一次读是没问题的，但第二次读出错了，于是操作系统必须放弃第一次读的结果，按第一种读法，那么有 64K 被浪费了，而按第二种读法只浪费了 1K，当然同样的分析也可以应用到前面分析的写操作上。

良好的文件读取系统应当如同一个数据库系统一样，把整个文件的读写操作视为一个事务。事务可以由多个步骤组成，但一个事务是一个整体，其中的任何一个步骤出错或者失败，那么整个事务就被视为出错或者失败。一旦一个事务出错或者失败，那么系统应当能恢复到这个事务开始前的状态。并且当一个事务未完成时，其中的任何一个步骤都不会对其它的应用产生影响。

也许上面的描述比较生硬，现在我们再用一个例子来解释一下它。假设你的操作系统提供了一个文件系统，这个文件系统提供了一个支持 FTP 的访问的功能（你或许会认为 FTP 是一个网络应用程序，不应当是文件系统的一部份，但你可以这样考虑一下，假设你的操作系统是一个分布式的操作系统，你的操作系统可以，或者说需要管理很多台计算机，那么你的文件系统是否可以用类似于 FTP 的方式在各个结点间进行文件的交流？），现在假设有一个应用在写这个文件，但发生了写错误，那么系统应当恢复到写之前的状态，即这个文件还是已前那个老文件，而不能是有一部份是新文件的内容，但新文件又因为出错而没写完，这样新老文件都不能使用，这是非常糟糕的一种情形！另外，当你在写文件时，有另外一个应用需要读取这个文件，那么由于你的写操作还没有完成，应此，它现在应当读的是原有的旧文件，而不是这个未写完的新文件，对于这种情况，你的操作系统又怎么处理呢？你或者说，我可以使用加锁操作，即当正在写的时候，我不允许读这个文件，必须当我写操作完成之后，我才允许你去读它。但请你想一想，如果你要写的文件很大，即你写它要花费很长的时间，那么你就会把这个文件锁定很长的时间，在这一个相当长的时间内，就有很多应用无法读取它，但如果我换一种方式，我另外在磁盘上找一块地方，先把这个文件写进去，然后再把目录表中指向旧文件的目录项更改为指向新的文件，这样我只需要在更改目录项的时候进行锁定就行了，而因为更改目录项的时间一段都很短，故而我们只需要锁定很短的时间就可以了，这样系统的性能显然就提高了不少。或行你会说，既然是旧文件，那么不使用它，让它不能访问也没什么，呵呵，话不能这样说，当你访问网页时，看见一个未更新的网页，总比看见“该页无法访问”要好。

当然，即使读写操作本身不会出错，但也不能保证读写出来的数据也不出错。比如说因为周围的电磁场的原故，有可能让数据的二进制位在读写的时候发生反转，比如二进制 101010 的最后一位因电磁场的影响翻转成了 1，即变成了 101011，对于操作系统也应当有一种策略去检测是否出现了这样的错误。当然，在通常的环境中，这样的错误不太容易碰见，但你的操作系统可能被用在工厂厂房的计算机中，甚至被用在航天飞机、卫星上，这时候电磁场就可能很强了，特别是在宇宙中，各种带电粒子不计其数，当一个带电粒子穿过磁盘的时候，说不定就会影起磁盘中的数据的一些二进制位出现变化，出现翻转，这个时候，你的操作系统就必须具有检错，而且不光要检错还必须有能力恢复，即常说的纠错，因为在卫星上你仅仅报错不恢复是没有意义的，你报错了也没人理会，你要是不自行恢复，卫星肯定就没啦。更要命的是你的操作系统本身的某些部份也被这样的磁场给影响了，那么你的操作系统是否还有自恢复的能力？

出错检测与恢复，这是目前研究很热的一个领域，国际上目前把这些相关东东称之为“容错”，容错不光有软件上的，还有硬件上的，我所就读的学校，哈尔滨工业大学，就专门有研究室在研究“容错计算机”，呵呵，这里就不打广告了:)，对此感兴去的朋友可以在网上找些相关资料看看。

还是那句老话，一个完善而真正实用的操作系统，需要我们考虑很多细节，需要我们开动脑筋去寻求解决办法，这是一个很浩大的工程，其间需要实际的动手能力去应用，也需要很扎实的理论基础作为根基、作为支撑。也许很多人认为学校学的理论都是没用的，其实这是一种非常错误的看法，理论会使我们的眼界更宽，考虑的问题更细致、更周密，理论的作用在于大刀阔斧的为我们劈开一片开阔的领域，当然要想在这片领域中有所收获，我们还必须动手实践、精耕细作。因此自己动手写操作系统的同时，更多的学习各种专业理论知识是非常必要的。

言归正传，前面我们介绍了 DMA 传输的一些基本要求，下面我们就来看看实际的代码是怎样去操作 DMA 控制器的，先来看看我们的 dma_open_dma_channel()函数：

```
void dma_open_dma_channel( unsigned char dma_channel , void* start_address_in_memory , unsigned short count , enum dma_read_or_write_model_enum read_or_write_model )
{
  /* 获取通道对应的页寄存器端口号 */
  int page_port ;

  switch( dma_channel ){
```

```
    case 2 :
      page_port = 0x81 ;
      break ;
  } ;

  // 关中断
  interrupt_close_interrupt() ;

  // 先关闭 dma 通道
  dma_close_dma_channel( dma_channel ) ;

  if( dma_channel <= 3 ){ // 使用的是第一个 DMA 控制器
    // 清零字节计数器
    io_write_to_io_port( 0xc , 0 ) ;

    // 设置模式位
    union {
      unsigned char ch ;
      struct{
        unsigned char channel : 2 ;
        unsigned char transfer_type : 2 ;
        unsigned char auto_intialization_enable : 1 ;
        unsigned char address_increment_or_decrement : 1 ;
        unsigned char model : 2 ;
      } ;
    } s ;
    s.model = 1 ;
    s.address_increment_or_decrement = 0 ;
    s.auto_intialization_enable = 0 ;
    s.transfer_type = read_or_write_model ;
    s.channel = dma_channel ;
    io_write_to_io_port( 0xb , s.ch ) ;

    // 发送物理地址的偏移量
    unsigned int offset = ( unsigned int )start_address_in_memory ;
    io_write_to_io_port( dma_channel * 2 , offset ) ;      // 发送偏移量的低 8 位
    io_write_to_io_port( dma_channel * 2 , offset >> 8 ) ; // 发送偏移量的高 8 位

    // 发送页号(即物理地址的段地址)
    io_write_to_io_port( page_port , offset >> 16 ) ;

    // 发送传送数量
    io_write_to_io_port( dma_channel * 2 + 1 , ( count - 1 ) ) ;      // 发送传送数量的低 8 位
```

```
    io_write_to_io_port( dma_channel * 2 + 1 , ( count - 1 ) >> 8 ) ; // 发送传送数量的高 8 位

    // 打开 dma 通道
    switch( dma_channel ){
      case 2 :
        io_write_to_io_port( 0xa , 0x2 ) ;
        break ;
    } ;
  }

  // 开中断
  interrupt_open_interrupt() ;
}
```

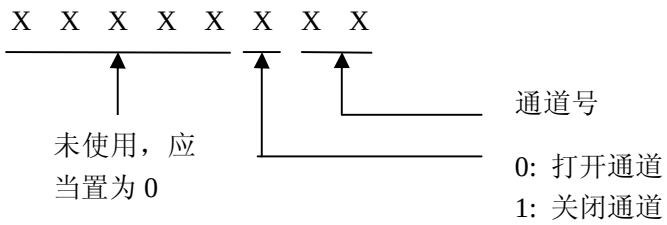
又是一个比较庞大的函数，让我们一行一行的来分析它。

首先，它根据用户指定的通道号，取得其相应的“页寄存器”的端口号，并存在 page_port 这个变量中。从程序里我们可以很显然的发现，它只处理了第 2 通道，并且这个第 2 通道的页寄存器的端口号是 0x81，这又是一个来源于硬件资料的数据。

随后，它关闭了中断，接着关闭了 dma 通道。因为很可能前次打开了此通道，因此在我们重新设置它的参数的时候，我们需要先关闭它。下面我们就来看看这个关闭 dma 通道的函数：

```
/* 关闭 dma 通道 */
void dma_close_dma_channel( unsigned char dma_channel )
{
  switch( dma_channel ){
    case 2 :
      io_write_to_io_port( 0xa , 0x6 ) ;
      break ;
  } ;
}
```

很明显，此函数是向 0xa 端口写入了一个值为 0x6 的数据。从硬件资料上我们可以知道，地址为 0xa 端口的寄存器是一个属于第 2 通道的用于使能目的的寄存器，它的结构如下图所示：

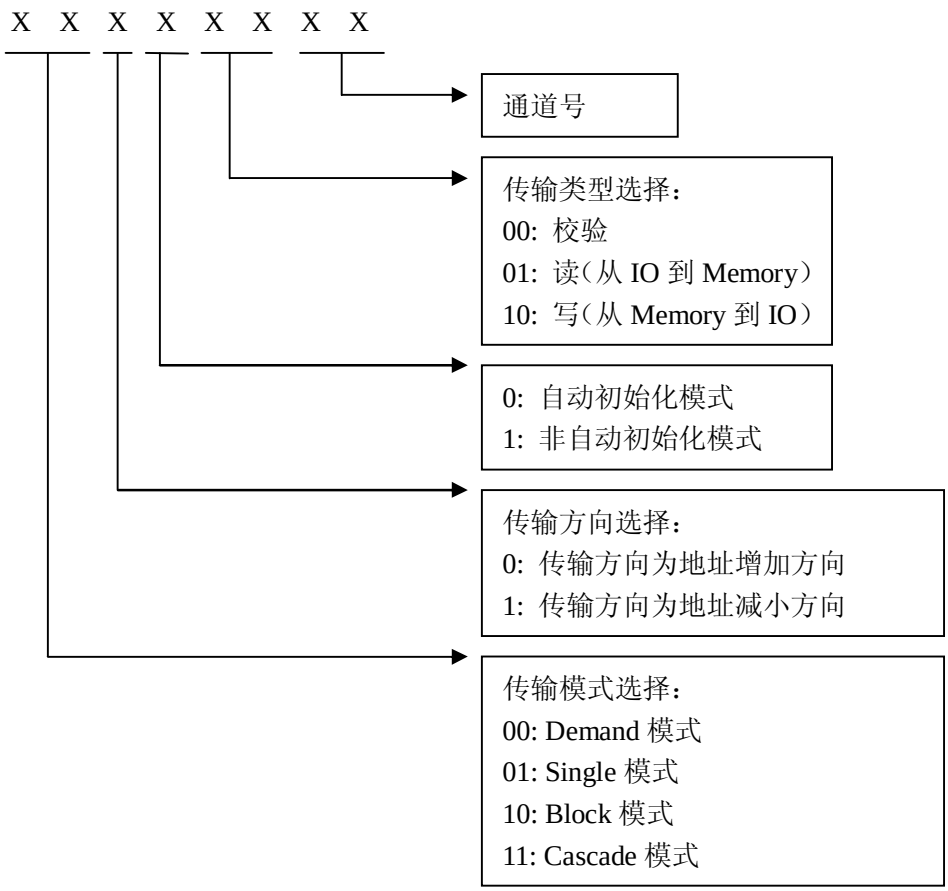


前面我们写入的数据值为 0x6，也即 00000 1 10，很明显，我们是关闭了第 2 通道。

继续向下看，它首先测试了一下是不是对主 DMA 控制器进行操作，这里需要说明一下的是，系统中可以有两个 DMA 控制器，一个是 8 位的，又称之为主 DMA 控制器，管理第 0 至第 3 通道；还有一个是 16 位的，常作为从控制器，管理第 4 至第 7 通道。虽然操作它们的方法是基本一样的，但是各自的控制端口所分配到的端口地址是不一样的，因此，需在程序中根据指定的通道号判断一下应当是对哪一个 DMA 控制器进行操作，并获得相应的控制端口的端口地址（即端口号）。

随后，它通过 `io_write_to_io_port( 0xc , 0 );`这一语句把数据 0 发送到了 0xc 端口。从硬件资料上我们可以知道，0xc 端口就是主 DMA 控制器的计数器寄存器。这个计数器用来表示有多少数据已经被传送了，因此在使用前我们需要清零这个计数器。

接着，它通过 `io_write_to_io_port( 0xb , s.ch );`这一句语设置了 DMA 的工作模式，这是通过将一个数值发送到 0xb 端口实现的，现在我们就来看看这个数值的结构：



这又是一个非常庞大的结构，非常不幸及报歉的是，目前就我手中的相关资料中，对上面各种选项都没有很细致的解释，而且某些资料上的一些解释还是相反的（感受兴趣的读者可以仔细对比一下本文所列参考文献的描述），这也使的我都上面各选项的含义没有确切的把握，不太清楚在什么情况下我们需要进行怎样的设置，或者选择不同的模式会对传输产生一些什么样的影响。如果您对此方面比较有研究，非常希望您能来信指教，我也会及时更新这份文档。

不过现在，对于我们这次的软驱驱动实验而言，我们只需如此设置就行了：

通道号：设为 10，因为前面已经提过很多次了，软驱传输使用的是 8 位 DMA 的第二通道。

传输类型：这个应当根据需要设定，由于此次实验我们只演示从软驱读出数据，故而此处设为 01。

自动初始化模式：我们设为 0，即让其自动初始化；

传输方向：我们选择地址增加的方向，这是因为，我们使用的目标地址其实是基地址，即数据的第一个字节所在的地址，选择地址增加的方向，DMA 控制器在传输的时候就会自动增加这个地址，将后面的字节传输到后续地址中。

传输模式：我们选择 Single 模式，但是我现在还不确切的知道为什么需要选此模式，只不过从一些资料及实际代码实验中发现应当是选择此模式，如果您对此有所了解，请您一定来信指教，谢谢！

OK，现在我们可以知道，我们只需要发送数据：01 0 0 01 10，即 0x46 就行了。

完成了模式设置之后，我们还需要将传输目的地的基地址告诉DMA，前面已经说过，这个地址是由一个页号及偏移量组成的。于是，代码中先发送了偏移量，然后紧接着发送了页号。由于偏移量是一个16位的数，因此，代码中将其分成了两次发送，第一次发送偏移量的低字节，第二次发送了偏移量的高字节，这些都能在代码中很明显的看出来。需要做说明的是，对于DMA第2通道而言，页号寄存器的端口地址为0x81，偏移量寄存器，也即DMA的地址寄存器的端口，这个端口地址对于8位DMA控制器而言，数值上恰好就是DMA通道号乘以2。

发送完目标地址之后，我们最后还需要把需要传输的字节数告诉DMA，这需要向DMA的字节计数器发送数据，而这个字节计数器的端口地址为前面地址寄存器的地址加+1，在上面的代码中应当很明显的看出这一点。

现在，我们的DMA初始化终于完成了，让我们回到最初的`void floppy_read()`函数中继续分析。

2.5 floopy_send_read_commmand()

DMA初始化完成后，我们就可以向软驱控制器发送读写命令了，这里发送读命令的操作是通过`floppy_send_read_command()`这个函数完成的，还是先展示一下这个函数的代码：

```
void floppy_send_read_command(unsigned char head, unsigned char cylinder, unsigned char sector)
{
  floppy_send_byte_return_is_ok( 0x66 ) ;
  floppy_send_byte_return_is_ok( head << 2 ) ;
  floppy_send_byte_return_is_ok( cylinder ) ;
  floppy_send_byte_return_is_ok( head ) ;
  floppy_send_byte_return_is_ok( sector ) ;
  floppy_send_byte_return_is_ok( 2 ) ;
  floppy_send_byte_return_is_ok( 18 ) ;
  floppy_send_byte_return_is_ok( 27 ) ;
  floppy_send_byte_return_is_ok( 255 ) ;
}
```

函数的本身逻辑很简单，其实只干了一件事，就是发送了一个命令，只不过这个命令是一个含有多个参数的多字节的命令，因此，它重复调用了`floppy_send_byte_return_is_ok()`这个函数，把这个命令的参数按顺序发送给了软驱控制器。现在我们就来看看这个命令到底是个什么格式：

1	M F S 0 0 1 1 0
2	x x x x x HD DR1 DR0
3	磁道号
4	磁头号
5	扇区号
6	扇区尺寸因子（x）
7	磁道的扇区数
8	空隙长度
9	读取的数据长度

其实软驱控制器有多个读命令，这里介绍的是“读扇区”命令，它一次可以读出一个整个扇区，当然还有“读磁道”命令，一次可以读出整个磁道。

这个命令共有九个字节，其中读一个字节是功能号，后面的都是此命令的参数。

M：为 **1** 的时候表示多个磁头（即多个磁盘面，如软驱的正反两面）同时读取，为 **0** 的时候，表示只读取指定的磁头

F：为 **1** 的时候表示读取的是高密度磁盘，为 **0** 的时候表示的是读取低密度磁盘

S：为 **1** 的时候表示跳过删除标志（**sorry**，我查阅的资料上均未提到此删除标志有何作用，如果您对此有所了解，非常希望您能来信指教）

HD：与你的磁头号一致

DR1、DR0：指示驱动器号，00（A）、01（B）、10（C）、11（D）

磁道号、磁头号、扇区号：这些需要根据你实际想读哪一个扇区而具体指定。

扇区尺寸因子（x）：此因子被用来计算扇区的尺寸，采用如下的公式：扇区大小 **= 128 * 2^x**，因为，对于 **3.5** 寸磁盘，一扇区应为 **512** 字节，即可得：**512 = 128 * 2^x**，解此方程，可得 **x = 2**，故此扇区尺寸因子的值应为 **2**。

磁道的扇区数：对于 **3.5** 寸磁盘，此值为 **18**。

空隙长度：在扇区与扇区之间，有一部份空隙，对于 **3.5** 寸磁盘，此值应为 **27**

读取的数据长度：由于此命令是一次读一个扇区，而一扇区为 **512** 字节，故而此值应为 **512**。

我想现在你可以对照着前面的代码看看，每个参数值是否应当如此设置 **J**。

2.6 floppy_read_result_of_read_command()

发送了读扇区命令之后，软驱控制器便去执行这个命令了，它会将数据直接通过 **DMA** 读到内存中，然后触发一个中断，即在中断发生后将可以知道软驱控制器已经完成了读取操作，并且数据已经存在于内存中了，故而我们在进行后续处理之前，必须先等待这个中断的到来，这是通过 `floppy_wait_for_interrupt()`这个函数完成的。此函数已在前面详细描述过了，这里就不再多谈了。

`floppy_wait_for_interrupt()`返回后，表明软驱控制器已完成了读取命令，故而我们又调用了`floppy_read_result_of_read_command()`这个函数读取软驱控制器给我们返回的结果，这个函数的实现如下：

```
void floppy_read_result_of_read_command()
{
```

```
  for( int i = 0 ; i < 7 ; ++i ){
    unsigned char byte ;
    floppy_read_byte_return_is_ok( &byte ) ;
  }
}
```

返回的结果共 7 个字节，它包含了很丰富的信息，我们可以通过这些信息得知读取操作是否成功，或者出了什么错误。具体的返回结果的含意，我们可以查阅相应的硬件资料。前面已经说过，pyos 默认一切操作都会得到正确的结果，因此，这里只是将结果简单的读取了出来，并没有进行进一步的处理。还是那句话，如果你是在编写一个实际的操作系统，你必须很小心的处理结果，因为，你需要处理各种各样的错误。

2.7 floppy_stop_motor()

读取了返回结果后，我们来到了整个函数的最后一步，这一步调用了 floppy_stop_motor()这个函数停止驱动器的马达，这个函数与先前的 floppy_start_motor() 其实是同一性质的，都是向同一个寄存器中写入一个数据，这个寄存器的结构在介绍 floppy_start_motor()函数的时候已经介绍过了，这里就不再多说了，我们来看看它的实现代码，你可以同前面 floppy_start_motor()的代码比较一下：

```
void floppy_stop_motor()
{
  /* 停止驱动器 */
  io_write_to_io_port( 0x3f2 , 0 ) ;
}
```

2.8 floppy_send_byte_return_is_ok()、floppy_read_byte_return_is_ok()

这两个函数虽然没有出现在 floppy_read()里面，但实际上在几乎每个被 floppy_read()所调用的函数中都出现了，它们是用来发送命令及读取返回结果的，我们先来看看这两个函数的函数体：

```
/* 读取数据函数,返回值是数据是否读取成功 */
int floppy_read_byte_return_is_ok( unsigned char *save_read_byte )
{
  // 这里本应当设置一个计数器,当尝试到一定时候即认为出错,返回 false,但由于是在虚拟机上使用,
  // 因此默认不会出错,故采取无限长等待策约,直到满足条件为止,因而返回结果总是真
  while( 1 ){
    unsigned char msr = io_read_from_io_port( 0x3f4 ) ;
    if( ( msr & 0xc0 ) == 0xc0 ){ // 检查 msr 的头两位是不是 11
      *save_read_byte = io_read_from_io_port( 0x3f5 ) ;
      return 1 ;
    }
  }
}

/* 发送数据函数,返回值是数据是否发送成功 */
int floppy_send_byte_return_is_ok( unsigned char byte )
```

```
{
  // 这里本应当设置一个计数器,当尝试到一定时候即认为出错,返回 false,但由于是在虚拟机上使用,
  // 因此默认不会出错,故采取无限长等待策约,直到满足条件为止,因而返回结果总是真
  while( 1 ){
    unsigned char msr = io_read_from_io_port( 0x3f4 ) ;
    if( ( msr & 0xc0 ) == 0x80 ){ // 检查 msr 的头两位是不是 10
      io_write_to_io_port( 0x3f5 , byte ) ;
      return 1 ; // 成功返回
    }
  }
}
```

程序是很简单的，从注释上我想大家应当能够完全理解程序的逻辑了，这里只对其中的一些关键地方简单介绍一下。

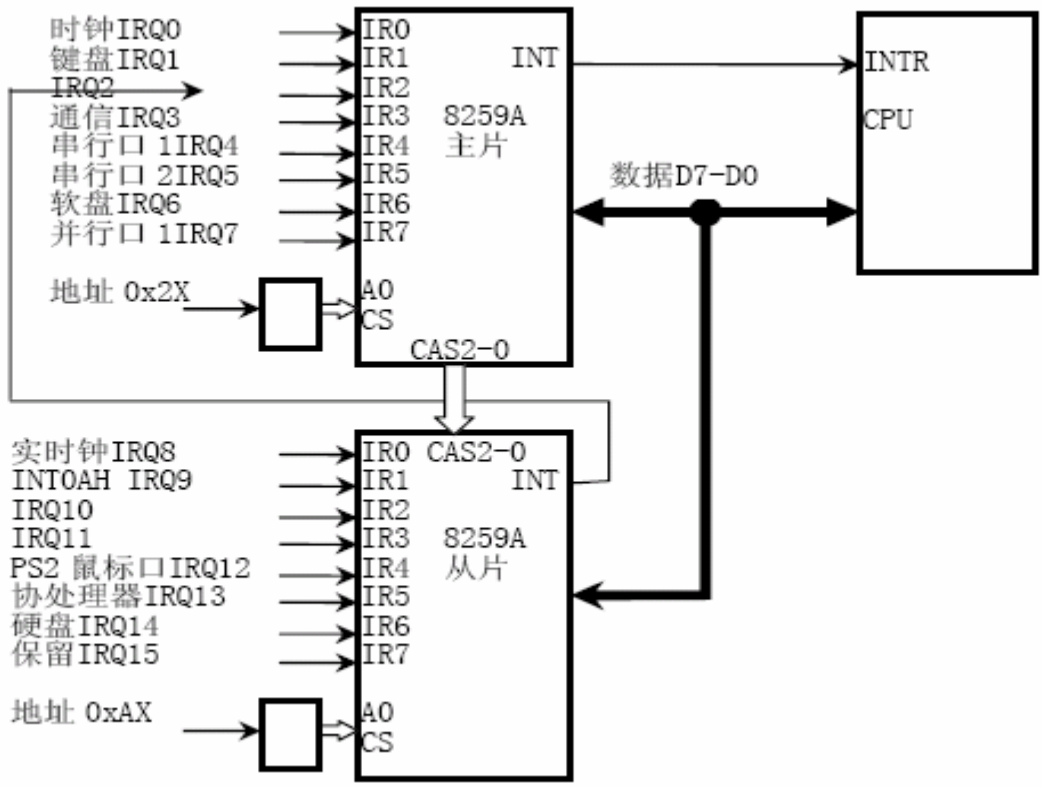
向软驱控制器发送命令及读取返回的结果，主要是同两个端口打交道，一是 0x3f4 端口（这里是对于 8 位的主 DMA 而言，对于 16 位的从 DMA，这个端口的端口地址是 0x374），此端口常称为“主状态寄存器”（Main Status Register，MSR），如果它的开头两位是 11，表示软驱控制器的返回值已经准备好了，于是，你可以通过 0x3f5 端口（对于 16 位的从 DMA 而言，此端口的地址是 0x375）读取返回值，故而，你可以看见 `floppy_read_byte_return_is_ok()`在读数据之前，测试了 MSR 的开头两位。

如果你要发送命令，你也必须先测试一下 MSR 寄存器，如果它的开头两位是 10，表示软驱控制器已经准备好接收命令了，于是，你可以通过 0x3f5 端口，把你需要发送的数据（命令）发送给软驱控制器。故而，`floppy_send_byte_return_is_ok()`在发送数据之前，同样也测试了 MSR 的开头两位。

从代码中的注释中你可以看出来，上面两段代码不是一个真正可实用的代码，因为，它没有进行任何的错误处理，这是你编写实际的系统的时候必须注意的问题。

2. 9 floppy_init()

前面不止一次提到，软驱控制器是通过中断与系统进行交互的，这在前面的代码分析中相信大家对此问题也应当是深有体会了。因此，要使我们的软驱中断能够工作，我们必须首先为软驱驱动安装好中断处理程序。要完成这个操作，那么我们还必须清楚它使用的是哪一个中断请求信号（IRQ），还记得下面一幅在《保护模式下的 8259A 芯片编程及中断处理探究》中所出现的图吗？



（此图最初来源于《Linux 0.11 内核完全注释》）

在上图中我们可以清楚的看到，软盘驱动使用的中断请求信号是 **IRQ6**。因此，我们需要在系统启动后，把这个 **IRQ6** 中断的中断处理程序设置为我们软驱驱动所提供的中断处理程序。这一点是在 **floppy_init()**中完成的，**floppy_init()**还完成了一些其它必要的设置，我们来看看它的实现：

```
/* 初始化函数 */
void floppy_init( void *buffer_address )
{
  /* 关中断 */
  interrupt_close_interrupt() ;

  floppy_called = 0 ;

  /* 安装中断 */
  interrupt_install_handle_for_interrupt( 0x26 , floppy_asm_handle_for_floppy_interrupt ) ;

  /* 设置中断屏蔽位 */
  interrupt_set_interrupt_mask_word( 6 , 1 ) ;

  /* 开中断 */
  interrupt_open_interrupt() ;

  /* 设定缓冲区地址 */
  floppy_buffer_address = buffer_address ;
```

```
  /* 初始化 */
  floppy_reset() ;
}
```

有关中断部份的操作，在以前的实验报告中有很详细的介绍了，这里就不再多说了，如果对此不清楚的朋友，可以参看一下《保护模式下的 8259A 芯片编程及中断处理探究》。这里我们来看看 floppy_reset()这个函数，它完成了对软驱控制器的复位或说初始化操作。

```
void floppy_reset()
{
  /* 复位软驱控制器 */
  io_write_to_io_port( 0x3f2 , 0 ) ; // 向 DOR 寄存器发送 0
  io_write_to_io_port( 0x3f7 , 0 ) ;
  io_write_to_io_port( 0x3f2 , 0x0c ) ;

  /* 等待中断 */
  floppy_wait_for_interrupt() ;

  /* 读取返回结果*/
  for( int i = 0 ; i < 4 ; ++i ){
    floppy_send_byte_return_is_ok( 8 ) ;
    unsigned char st0 ;
    floppy_read_byte_return_is_ok( &st0 ) ;
    unsigned char cylinder ;
    floppy_read_byte_return_is_ok( &cylinder ) ;
  }

  /* 设定规格 */
  floppy_set_specification() ;
}
```

这个函数的流程从代码及注释中应当可以非常清楚的看出来了，首先，它发送了一个复位命令，让软驱控制器复位（初始化）。这个命令是最简单的命令之一，只需要向数据输出寄存器（DOR）送出 0 即可，对于 8 位的 DMA 来说 DOR 寄存器的端口地址是 0x3f2，它的结构在前面介绍 floppy_start_motor()函数的时候已经详细介绍过了，记不太清的朋友不妨翻回到前面看看。

复位命令发送后，同前面一样，它等待一下中断的到来，以确认软驱控制器已经完成了复位操作。由于一个软驱控制器可以控制 4 个驱动器，故而，复位命令共有 4 组返回值，因此，后面用了一个循环将这 4 组返回值都读取了出来。当然，反复强调，如果是一个实际的操作系统，你需要对反回值进行检验，以确定是否命令被成功完成，如果出错，你还需要进行相应的出错处理。

读取完返回结果之后，我们需要设定规格，因为软驱是多种多样的，系统中可能安装的是常见的 3.5 英寸（1.44M）驱动器，也可能是已经濒临灭绝的 5.25 英寸驱动器。还是先来看看这个函数的实现吧：

```
/* 设定规格 */
void floppy_set_specification()
{
  floppy_send_byte_return_is_ok( 3 ) ;
  floppy_send_byte_return_is_ok( 0xdf ) ;
  floppy_send_byte_return_is_ok( 2 ) ;
```

```
}
```

函数本身逻辑非常简单，不是发送了一个设定规格的命令，下面让我们看看这个命令的格式：

Bit Byte	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1
1	Step Rate				Head Unload Time			
2	Head Load Time							NDM

（此图来源于本文的参考文献）

这个命令的格式非常复杂，其中 NDM 表示的是否使用 DMA 传输，如果使用 DMA，此位应当清 0，否则应当置为 1。对于其余及个参数的数值，大家如果想要详细了解，可以查看本文的参考文献，或者其它一些介绍磁盘物理参数的硬件资料。其实对于现在最常用的 1.44M 软磁盘并使用 DMA 传输而言，这两个值是固定的，应当分别为 0xdf 及 0x2，你只需要按此数值发送命令就可以了。当然，如果是一个兼容性很强的实际系统，你必须检验软驱类型，并根据实际情况进行设置，这方面的内如在本文的参考文献里有专门的论述，如果感兴趣的话，可以翻看一下。

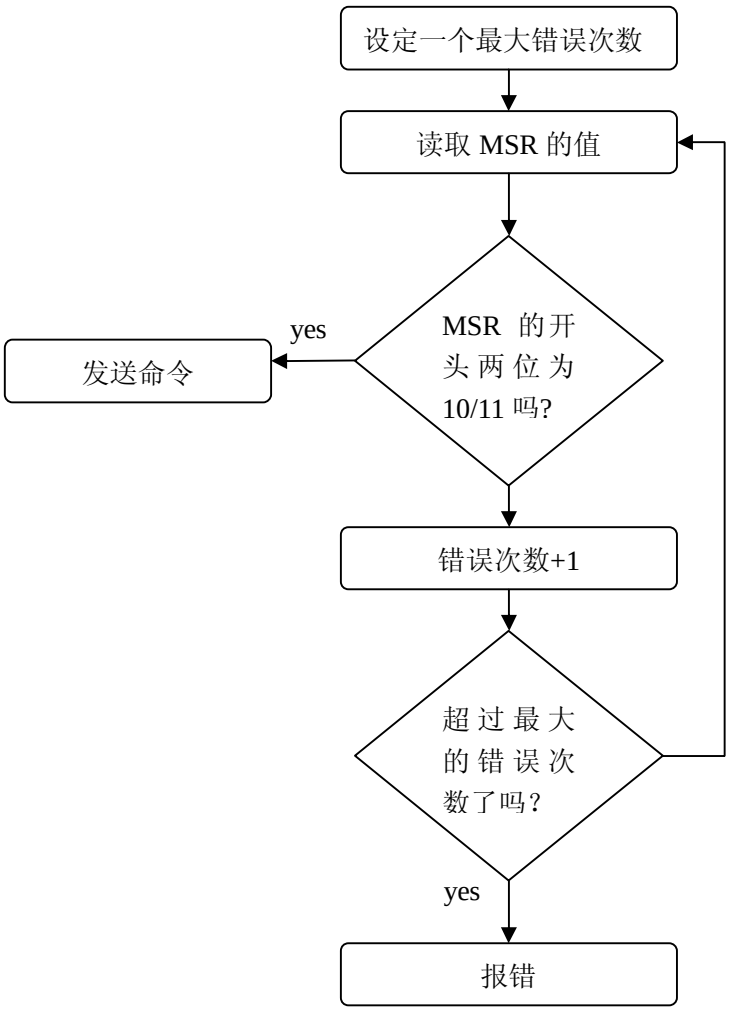
三、总纲

行文到此，本实验的主要内容已经基本叙述完成了，不过前面的描述实在太过于纷繁，因此这里对前文所述进行一下总结。

其实对与 pyos 的软驱驱动来讲，就两个主要的大函数，一个就是 `floppy_init()`函数，它是在系统初始化的时候被调用，完成对软驱控制器的复位，也即初始化工作。另一个就是 `floppy_read()`函数，它是在系统需要读取软驱数据的时候被系统调用，每次读取一个扇区的数据到指定的内存中。其中涉及到对很多端口的操作，下面一张表是来源于本文参考文献的一张有关软驱控制器的端口的分配表，希望能对大家阅读本文及源代码有些许帮助。

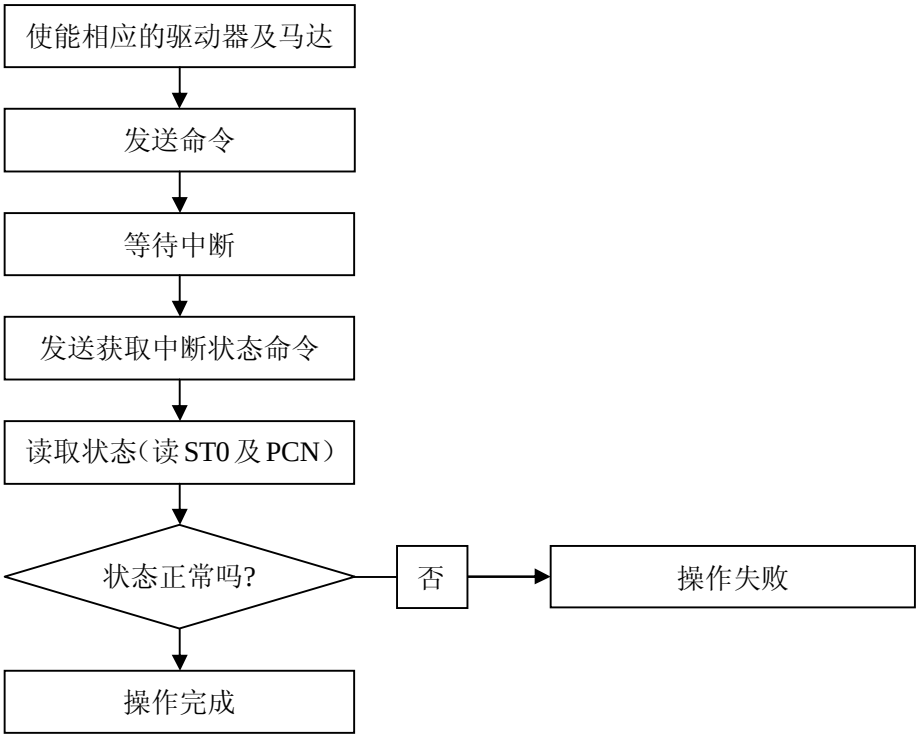
端口用途/名称	作为主控制器时的地址	作为从控制器时的地址	读写权限
状态寄存器 A（PS/2）	0x3f0	0x370	只读
状态寄存器 B（PS/2）	0x3f1	0x371	只读
数据输出寄存器（Data Output Register，DOR）	0x3f2	0x372	只写
主状态寄存器（MSR）	0x3f4	0x374	只读
数据数率选择寄存器（Data rate Select Register，DSR）（PS/2）	0x3f4	0x374	只写
数据寄存器	0x3f5	0x375	读写
数据输入寄存器（Data Input Register，DIR）（AT）	0x3f7	0x377	只读
配制控制寄存器（AT）	0x3f7	0x377	只写

其次，还有两个重要的功能函数，一个就是 `floppy_read_byte_return_is_ok()`函数，它是用来读取从软驱控制器来的各种命令的返回值的，另一个就是 `floppy_send_byte_return_is_ok()`函数，它是用来向软驱控制器发送命令的。这两个函数其实有一个相对固定的流程，如下图所示：

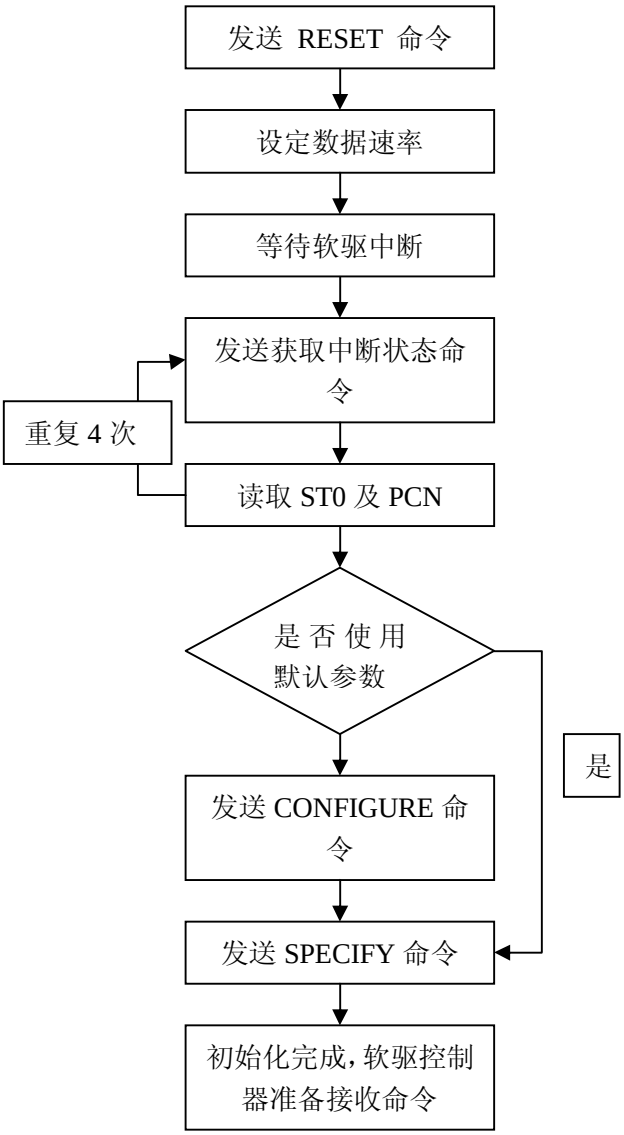


上面的流程考虑了错误处理，与 **pyos** 未考虑错误处理的流程相对，有稍微的差别，如果你在编写一个实际的系统，那么你显然应当按照此流程去处理。

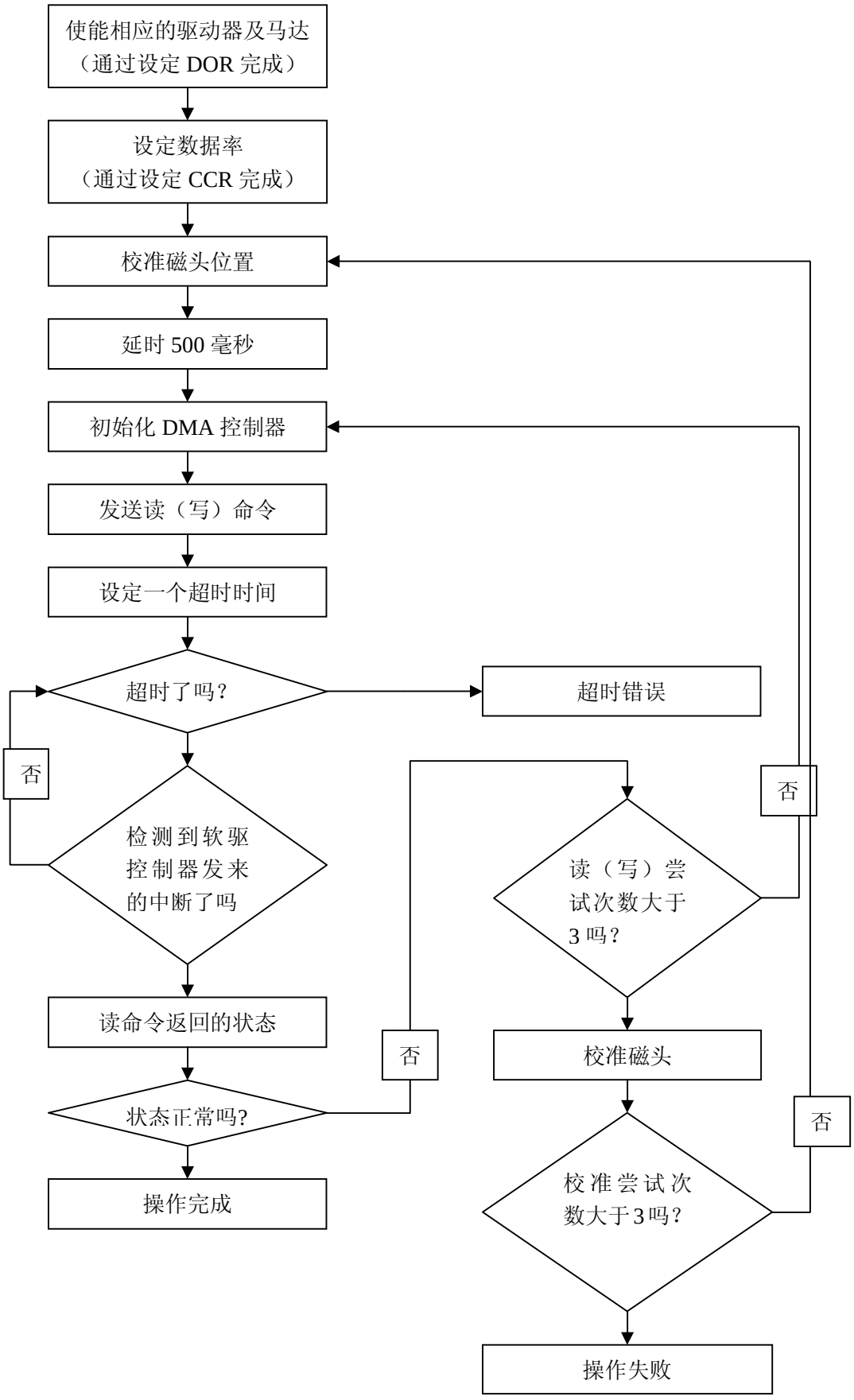
对于一般的发给软驱控制器的命令，我们应当按如下流程去处理：



对于初始化操作，应按如下的流程去处理：


发送 RESET 命令
设定数据速率
等待软驱中断
发送获取中断状态命令
重复 4 次
读取 ST0 及 PCN
是否使用默认参数
是
发送 CONFIGURE 命令
发送 SPECIFY 命令
初始化完成，软驱控制器准备接收命令

对于读写操作，流程如下：


使能相应的驱动器及马达（通过设定 DOR 完成）
设定数据率（通过设定 CCR 完成）
校准磁头位置
延时 500 毫秒
初始化 DMA 控制器
发送读（写）命令
设定一个超时时间
超时了吗？
超时错误
否
检测到软驱控制器发来的中断了吗
读命令返回的状态
状态正常吗?
否
操作完成
读（写）尝试次数大于3吗？
否
校准磁头
校准尝试次数大于3吗？
否
操作失败

上面基本上就是软驱驱动中最常见的几个操作的基本流程，在本文的参考文献中有详细的描述，不过这里为了大家使用方便，我还是将它们画了出来。将这些流程及前面介绍的代码进行对照，我想你至少可以看见这么两点：

第一，对硬件操作的确是难度不高但又极其麻烦的一件事。说它难度不高，是指在你掌握了相应的硬件资料的情况下，你可以很容易的按照硬件资料上所描述的流程编写出代码，说它极其麻烦是指操作步骤烦多，而且需要非常小心的进行错误处理，这些大概也算是底层编程的共性吧。

第二，**pyos** 的确是一个非常简单初级不可实用的系统，为了保持系统像拼音一样简单（呵呵，**pyos** 名字的来原就是因为笔者想让它成为一个像拼音一样简单的，连小学生都能看懂的系统，当然，其实最根本的原因是笔者目前还没有能力完成一个复杂的系统：)），**pyos** 做了太多的假定，最主要的是没有处理任何错误，但我想，在明白了基本原理和基本方法以后，绝大多数人应当都有能力去完善它，去开发出自主的可实用的系统。

如果 pyos 能成为一个引入者的角色，pyos 也算是完成了它的使命，笔者也会因此而感道非常欣慰。

参考资料

1.《82077AA Floppy Controller Datasheet》.Intel

这是 Intel 公司的关于 82077AA 软驱控制芯片的官方资料，非常详尽，介绍了所有的命令及操作的流程。详尽权威是这篇文档的优点，但有时候会让你觉得详尽得过余繁复了，而且没有一个实际的可操作的例子，难免会让读者觉得感性认识不足。

2.《Programming Floppy Disk Controllers》

这篇文档是从网上得到的，很遗憾文档中没有标出作者是谁（但我想这个应当能通过 Google 查到），这篇文档很简明，可以当命令的速查手册，遗憾的是虽然最后给了一个小的汇编例子，但描述很少，难免会使读者对于整个操作的流程难有一个很强的整体认识，光读此文档，你可以很快的知道软驱控制芯片都能干什么，但却很难知道应当怎么干。

3.《DMA Programming》.Justin Deltener

这篇文档是描述怎么操作 DMA 的非常好的一篇文档，即简单又不失详尽，稍微有点遗憾的是最后给出的一个例子，对于初用 DMA 的读者而言，可能稍显复杂，也没有说清楚怎样去实际调用。

4.《How To Program The DMA》.Breakpoint

又是一篇很好的 DMA 文档，我觉得比上一篇对于入门而言更好一点，例子得当，注释丰富，简单明了，还在附录中给出了怎样在 32 位保护模式下操作 DMA，遗憾的是，好像权威性及详尽性方面比上一篇文档感觉上要稍弱一点，有些地方的描述与上篇文档有出入。

（全文完）

【责任编辑：sun@hitbbs】

车辆牌照识别系统的预处理算法

哈尔滨工业大学 生物信息技术实验室
刘鹏翔(hoticehotice@sohu.com)

对于这个给定题目，我们将使用人类的方式进行问题分析。也就是说，按照一定的逻辑推理顺序进行。当我们拿到一个陌生的问题的时候，由于所给的已知条件很少，我们需要一点一点的发掘;在发掘的过程中，逐步解决每一个小问题，最后接近解决方案。

让我们来进行分析这个题目：车牌识别。

车牌是啥东西大家应该清楚吧。第一个问题搞定。下一个，什么是识别？这个很熟悉的词汇还不太容易解释。在我的理解中，要识别什么，首先你要以前见过这个东西，也就是说，**你拥有关于这个东西的先验知识，然后根据这些知识，来判断你所遇到的某种事物是不是原来你遇到的那些，如果是，给出它们的内容，并且说出它们和你以前遇到的东西的区别**。对于人来说，处理起这些识别问题轻而易举：恶劣的环境下，最笨的人会在 100～200ms 内识别出某种物体，而对于计算机，这类识别问题的处理规模甚至接近无穷。

现在问题分析完了：你以前见过车牌，并且能够认出上面的子母和数字。什么？！你说你没见过？哦，那好，首先找一辆车……#$%^$% 现在，这个问题交给计算机去处理。也就是说，**让计算机从给定的材料中，辨认其中是否有车牌，如果有，则读出其中的内容**。

从题目上能看出的东西就这么多，第一层次的分析结束。第一层的分析完了。那么另外一个问题来了：计算机要做这件事，需要获得什么样的材料？

我们再进行第二层次的发掘：计算机处理的事物对象。在这里，识别车牌是从视觉上来实现的。**对于计算机，处理视觉信息主要依靠静止的图像**。对于动态的图像，实际上也是分解成若干的“帧”来进行的。这些“帧”，也是静止的图像。也就是说，计算机通过直接或间接获取的图像进行处理，进行转换或者从中获取有用的信息。第二层次分析结束。

第三层的发掘：对获取的图像进行怎样的处理，才能达到识别并且分离信息的目的？这是一个大的问题了，也是最终的解决方法所在的层面。在这个层面，首先还是概念上的分析，然后是处理方法上进行逻辑分析。

在计算机识别中处理的图像，大多为位图，也就是说，图像是由一个个单个像素构成的点阵图，通过像素的颜色和明暗来记录信息。每个点从宏观内容上看或许有关连，但从逻辑上是各自独立的。我们可以单独对任意一个像素进行处理而不影响其他的颜色和明暗度。图像识别中，对像素的处理就是：对于这些宏观上看起来意义很明显的东西，怎么让计算机去找出它们的内在关系，然后把它们标记出来（计算机还真是笨@_@）。计算机文件的存储还是基于物理的，它根本不去考虑存储的东西是什么，而人类则是以事件和联想为基础的记忆。所以，让计算机去识别一个物体难度还是相当大的，甚至实现的过程很机械。对此方面有兴趣的朋友可以去了解人工智能、人工神经网络、计算机视觉等方面的材料。

对于问题背景的分析完成。好了，下面我们开始工作了。为了让没有编程基础的朋友也能很透彻的了解文章的内容，我们这里只论算法，不涉及具体的哪一种语言。在分析的过程中主要用到一些数字图像处理的知识，

相关材料可以到网上搜索，会有很多。

需要说明的是，下面是个很初级和浅显的例子，对这个问题的分析和解决也仅仅停留在实验的阶段上，程序所能解决的问题相当初级。那为什么还要写篇东西呢？原因就是，我们从一个简单的问题分析入手，逐渐加深对于理解，对于复杂问题的分析很有帮助。写这个东西的另外一个目的是，对于理工科目出身的朋友来说，如果想清晰地描述一个问题，语言表达能力是一定不能忽视的。很多人能够写出很漂亮的程序，但是对于描写开发过程，也就是说产品的文档部分，有很大的欠缺。

图一：车辆图片

这是一幅跑车的图片，车牌识别预处理所要完成的任务是：首先，从图片中提取出车牌所在的矩形区域；然后，在提取出的车牌区域中把车牌中的数字和字母单象素化，以便于文字识别系统的直接识别。这里我们只讨论车牌区域的提取和单象素化。

牌照识别目前已经有可用的商业化软件。识别的总体策略有高通滤波和阈值分割的方法。这里采用的策略是比较简单的阈值分割方法。

想要把车牌区域提取出来，必须清楚车牌有什么样的特征。首先，车牌的区域反差比较大；其次，车牌文字和底色是黑白的，如果不考虑由于拍摄白平衡因素造成的轻微偏色，车牌颜色的 RGB 分量应当很接近。

根据信息学的方法判断，图像中方差比较大的区域比较可能为存在大量信息的区域。这样，大体上可以分成两步走：第一步，将画面分割成若干相邻的小区域，计算每一个区域内的标准方差，设定一个根据经验值估计出来的阈值，小于阈值的区域，直接填充成全黑块（填充成全黑是因为正常拍摄的照片几乎不会出现 RGB＝0，0，0 的像素点），大于阈值的区域保留下来进行进一步递归处理，直到满足条件跳出；第二步，对于第一步处理后的图像进行加工：第一步的处理将产生很多零碎的、未被填充的小区域，对这些区域进行连接或进一步填充成全黑块。

那这个过程怎么让计算机来实现呢?必须给出一个算法。**所谓一个算法，必须能按照一定顺序执行，最后能达到停机的状态，并且给出一个正确或者错误的结果**。

根据以上分析，我们自然的想到递归的办法。递归的算法有一大类名称叫做**分治**（就是分而治之的简称）：**把一个大问题分解成小的子问题的组合，也就是把任务分解下去，把问题的解归解为一系列小问题的解的组合，然后把这些解组合，得到原来问题的答案。**

好了，闲话少叙，铺垫了这么多，该进入正题了。对于方差，我们首先考察整幅图片；第一次考察的结果肯定大于给定阈值，因为这个区域里有足够的信息；于是把问题分解，变成几个子问题的求解。在这里，我们把原图片分解成四个区域：左上、右上、左下、右下。那么分解就存在一个问题:计算机一般按照 2 的整数倍来处理问题，这里如果边长不能一直被 2 整除怎么办呢?这里用了一个近似的办法:考虑到车牌识别的应用有一些特点，比如拍摄条件固定，车牌的大小，颜色固定，车辆的位置也固定等;根据这些特点可知，车牌一般位于图像的中间部分。于是我们人为的把图片裁成 2 的幂指数倍大小，然后把多余的填充。我们从图像的左上角开始画一个 256*256 像素的图像块，而把其他的都涂成全黑。而实际应该在图像的正中间取这个图像块。这样选只是为了编程方便，因为在编程语言里，一般从左上角开始算作（0，0）点。涂黑后的图像如下图所示。

图二：涂黑后的图像

对于分治算法，分解后还有一个结束循环的问题，也就是跳出条件。究竟运行到什么程度才跳出呢？对于图像，显然是一个有穷个数像素的集合，最终肯定可以穷尽的处理每一个像素。但是，这并不是我们想要的结果，虽然倒不是因为处理颗粒度太细需要比较长的运行时间。我们关心的是，在这个过程中，小方块的大小小到一定程度时候，块内的像素相关程度变大，也就是说，颜色趋于一致，方差自然越来越接近给定阈值，最终小于它，进而结束循环。车牌部位的方差固然大，但如果小方块太小，也可能使块内的方差小于给定阈值，造成车牌部位被填充。

也就是说，我们要根据两个条件来确定是否跳出：第一是块的大小，第二是块内的标准方差大小。

对于第一点，我们可以根据试探性的测试来确定这个跳出条件，我们把 8*8 像素大小的块作为循环中止时的块大小。

而对于第二点，方差阈值的确定，就比较麻烦。究竟确定多大的阈值才算科学，或者说更加有效的进行判断？在这里，我们采用了标准方差是有道理的。原因如下：第一，对于同一批车牌，不同天气条件下拍摄的图像亮度不一样；第二，即使同一种天气下，不同光照角度下拍摄的图像亮度也有很大差异。

所以，采用标准方差衡量图像，其实就是比较图像中物体的不同反光率，来推测可能存在车牌的区域，因为标准方差每一个像素和比较周围的衡量标准。亮度高的时候，所有区域的亮度都高，低的时候，都低；唯一不同的就是物体不同部位的不同反光率。假如这个算法变成一个应用程序，来进行车牌识别，那么这里假设应用条件相对单一，也就是说车牌种类和车牌在所获取图像中的大小比例一定。那么，我们就可以根据一批图像进行测试，来确定一个相对有效的阈值。在这里，根据这种原则，我们就可以试探性的给出一个阈值 T，来判断是否涂黑这个小块。

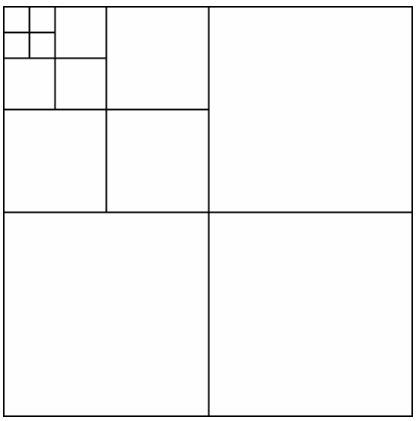
下面给出这个分割算法的伪代码：

```
void sgmt(int nNx，int nNy，int nNw，int nNh)  //块的起始 x，起始 y，块的宽，块的高
 {
 if(块大小<8 像素)  return ;  //小块的尺寸小到 8 个像素宽，退出循环
   if(fc(nNx，nNy，nNw，nNh)<阈值 T) //如果块内像素灰度标准差小于阈值 T，把整个块涂黑
     {
      for(i=nNx;i<nNx+nNw;i++)
       for(j=nNy;j<nNy+nNh;j++)
          Pixels[i][j]=0 ;
      return ;                    //搞定，跳出递归循环   :)
     }

  sgmt(nNx，nNy，nNw/2，nNh/2) ;               //左上块递归循环
  sgmt(nNx+nNw/2，nNy，nNw/2，nNh/2) ;          //右上块递归循环
  sgmt(nNx，nNy+nNh/2，nNw/2，nNh/2) ;          //左下块递归循环
  sgmt(nNx+nNw/2，nNy+nNh/2，nNw/2，nNh/2) ;    //右下块递归循环
}
```

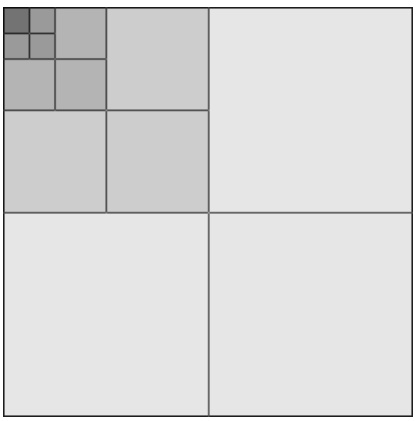
其中，void fc(块起始的坐标点 x，y；块的大小)是计算标准方差的函数，任意给定一个起始点和块的大小，计算出它的像素之间的标准方差，并返回。

好了，确定了方差给定阈值和跳出循环时候的块大小以后，这个算法就可以开始工作了。

下面开始运行算法。下图给出的是对于这个规则的块进行不断的二分的过程，当运行到块大小为 8*8 像素的时候，跳出循环。



图三：算法划分图



图四：算法运行顺序图

下面是执行完算法过程的结果。咦？怎么会出现这种状况？和想象中的似乎不一样。产生原因是，除了车牌部位，还有很多部分的方差和块的大小也满足跳出条件。那怎么办呢？从图上可以看出，车牌部位作为一个连续的区域被保留下来。也就是说，车牌的高和宽都是 8 的整数倍个像素大小，并且，这些小块彼此相邻。

图五：算法执行以后的结果

怎么处理呢？还需要耍点小聪明，呵呵。我们发现，这些乱七八糟的小块都是散兵游勇，没有形成统一的区域和势力。这就好办了。我们可以从水平和竖直方向按照单行或者单列来扫描，记录并考察它们连续的程度，如果连续的程度小于一个给定的阈值，就把这一行或者列都涂黑。这样，当水平和竖直方向都扫描完以后，就把这些小块都涂黑了。下面是处理的结果。怎么样？结果还算满意。

图六：消除杂乱块后的结果

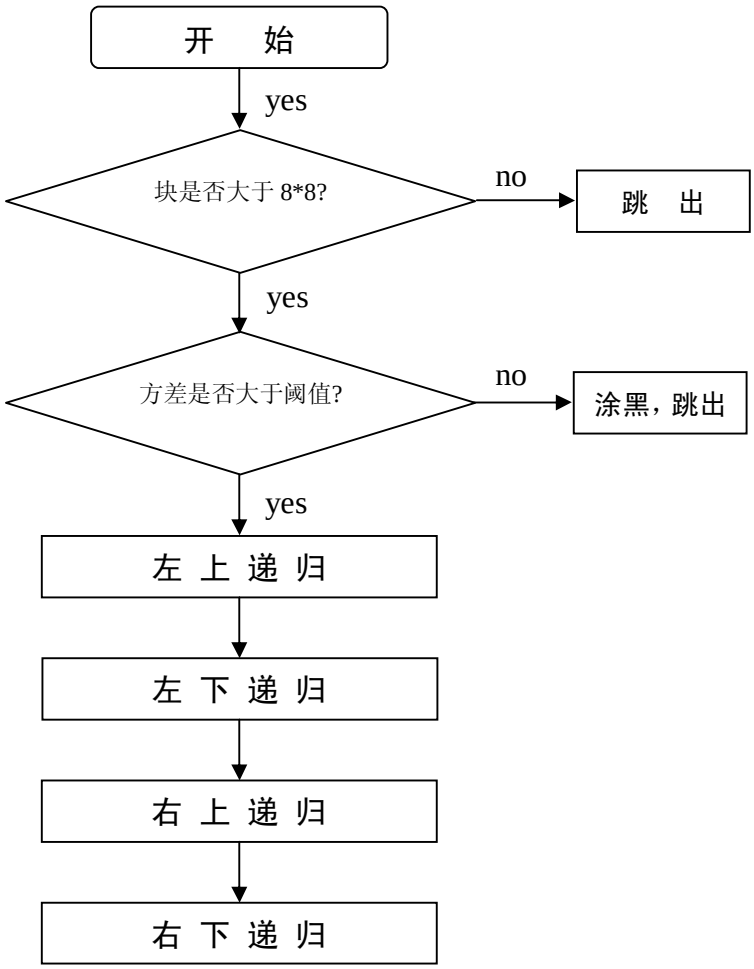
接下来，我们进行提取字母和数字的部分工作。完成了上面的部分，就可以很顺利的进行这一部分了。我们只需按照竖直方向扫描，如果连续几个像素遇到同一种颜色，就把它涂黑，如果不是，就涂成绿色。下面就是结果。

图七：提取出数字和字母后的结果

至此，我们把车牌识别中的预处理部分完成了。接下来的工作应该是使用腐蚀算法把图中的绿色字母和数字变成单像素。由于涉及到很多图像处理相关的知识，在这里就不一一赘述了，有兴趣的朋友可以接着进行深入的研究。

最后，对于车牌识别的阈值确定原则，还应该有更好的研究方法，这里仅仅为了演示这个过程，故还希望各位编程玩家能够给出更好的方法。还有一个需要说明的问题是，虽然这里使用了二分法来划分块，并不是说分治算法就一定使用二分法，你也可以 3 分法，5 分法等等。当然，时间和空间上的复杂度也不同，有兴趣的朋友可以动手试试看。

附录一：第一处理过程的流程图



附录二：程序对于实际车牌的处理尝试

在这里，由于图像尺寸和车牌所占比例比较大，我们把跳出循环的块大小修改成 16*16，结果显示在消除杂块的部分还有待于进一步加强。程序的实际应用能力还很弱，不能称之为一个能够工作的软件。上面的演示的结果还算满意的结果是有依赖性的，也就是说，我们相当于针对这幅图像进行了算法优化。

在文章的结尾，希望有志于此的朋友接着进行研究，作者在此表示感谢。

【审核：张华[1]】

[1] 张华：北京大学 计算机学院 硕士研究生，本刊特邀审核。

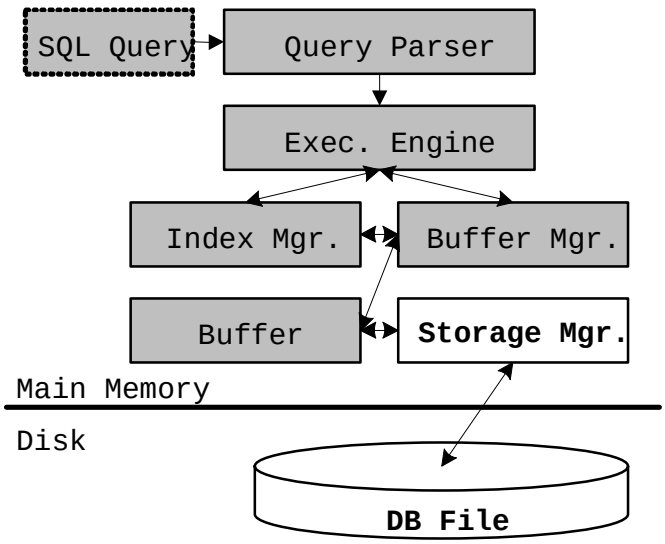
MyBase®物理存储结构的设计

赵锴[1](zhaokai@hit.edu.cn)

编者按：本文详细介绍了一个正在设计的实验性数据库管理系统项目 MyBase 的物理存储结构设计方案。包括数据字典、数据表的存储结构和访问这些信息的内存数据结构。相信这些工作对于希望了解数据库管理系统内部原理和实现机制的读者们能够有所助益。

1 简介

MyBase 是一个正在设计中的实验性数据库管理系统（DBMS）。它以现有的通用 RDBMS 为蓝本，旨在设计和实现一个轻量级的 DBMS，以帮助加深对数据库管理系统实现的理解。当前的目标包括支持 SQL 语句的子集和支持内存算法[2]。MyBase 系统的结构框图参见“图表 1”[3]，需要说明：查询优化器是 DBMS 的重要组成部分，我们将其划分到执行引擎(Exec Engine)中。



图表 1 MyBase 系统结构

本文主要介绍 MyBase 的物理存储结构以及存储管理器（“图表 1”中浅色部分）的设计。MyBase 系统运行于 Microsoft® Windows® 操作系统，并依赖于操作系统的文件系统管理存储结构。

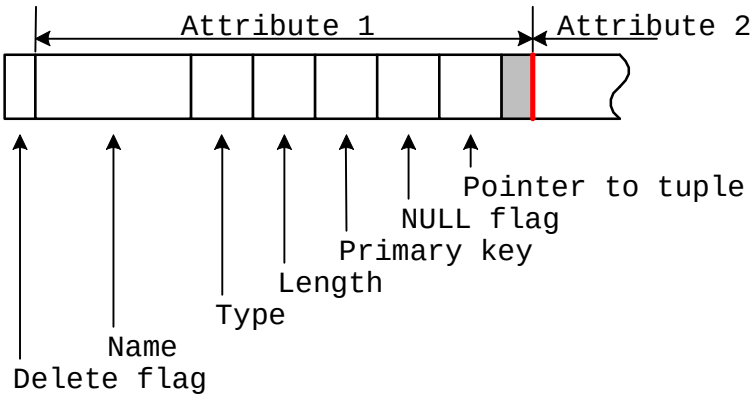
[1] 哈尔滨工业大学数据库与并行计算研究中心硕士研究生，研究方向为：压缩数据库技术

[2] 编者注：数据库系统设计的核心问题是优化对磁盘存储器的 I/O 操作，和建立有效的缓冲机制。绝大多数数据库操作实际上都是针对内存缓冲区中 copy 执行的主存算法。

[3] 编者注：完整的 RDBMS 管理系统框图和各部分功能说明参见 H. Garcia-Molina *et al*. *Database System Implementation*. Prentice Hall, 2000: pp 7. 目前 MyBase 系统结构中没有包含的模块主要有：事务处理、日志、并发控制以及用户接口。

2 数据字典(元数据)的存储

数据字典存储在独立的.dic 文件中，其包括每一个表建立的时间、表中元组的个数、每一个元组的属性、属性类型、长度、是否主键、是否为空等信息。“图表 2”给出了元组属性在内存中的顺序存储格式：



图表 2 元数据在内存中的顺序存储格式

数据字典中描述数据表和每一元组各属性的元数据数据结构如下所示[4]：

```
struct tagAttribute
{
  CHAR     name[MAX_NAME_LENGTH];     //属性名称 10
  INT      type;                      //该属性的的类型 4
  UINT     length;                    //该属性的长度 4
  BOOL     is_primary_key;            //是否为主键 4
  BOOL     is_null;                   //是否可以为空 4
  CHAR*    ptr;                       //该属性在内存中的地址 4
  CHAR     reserved[2];               //保留，填满 32 字节
};
typedef struct tagAttribute   Attribute;
typedef struct tagAttribute*  pAttribute;
struct tagTableSpace
{
    SYSTEMTIME    create_table_date; //该关系的创建时间
    INT           tuple_number; //该关系的元组的总个数
    INT           tuple_length; //每个元组的长度
    INT           attribute_number; //该关系的属性个数
    Attribute     attribute_list[MAX_ATTRIBUTE_NUMBER];
```

[4] 这里使用了 Windows 平台下的一些定义类型，INT, CHAR, UINT, BOOL 分别可以对应到标准 C++的 int, char, unsigned int 和 int 类型。

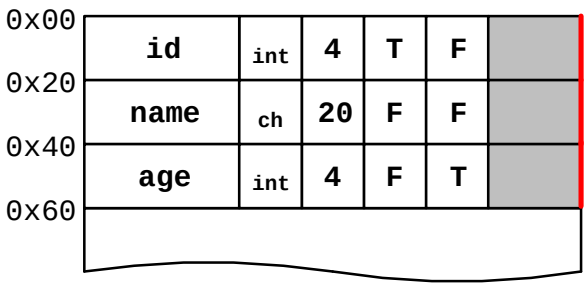
```
    INT             current_index; //当前在缓冲中所指的记录的索引
    CHAR*           ptr_current_index; //当前在缓冲中所指的记录的指针
    CHAR            cache[TABLE_CACHE_SIZE]; //关系的缓冲
    INT             tuple_number_in_cache; //当前调入缓存的元组个数
    INT             begin_index; //当前在缓冲中所指的记录的索引的最小值
    INT             end_index; //当前在缓冲中所指的记录的索引的最大值
    HANDLE          table_file_handle;//表文件句柄
    CHAR            table_file_name[TABLE_FILE_NAME_LENGTH];
    BOOL            is_table_file_open; //该文件是否打开
    BOOL            is_table_space_open;//判断该关系是否打开
};
```

图表 3 描述元数据的数据结构

元数据在磁盘上的存储结构，不同的 DBMS 有所不同。Microsoft® SQL Server® 2000 使用单独的数据表将元数据等同为元组存储。例如一个表的建立会在 `master` 数据库的 `syscolumns` 中插入每一个属性列的详细信息。这里为了直观起见，采用顺序存储的描述方式，举例如下：

```
CREATE TABLE student(id int primary key,
                     name char(20) not null,
                     age int);
```

这条 SQL 语句将创建名为 `student` 的表和三个属性列。MyBase 会在元数据中增加对应的数据项，如下图：

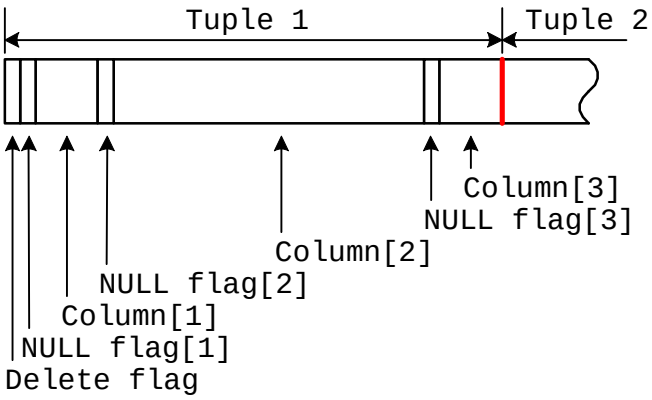


图表 4 元数据存储示例

与“图表 1 图表 2”的内存格式对应，“图表 4”的元数据中存储了每一个属性的名称、类型、长度、是否为主键和是否为空等信息。

3 关系表(元组)的存储

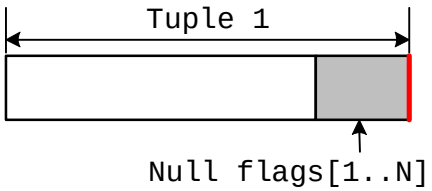
每一个关系表存储在以.mdb 结尾的独立文件中。元组的存储采用定长记录。并根据元数据记录确定每一字段的长度等信息。



图表 5 元组在文件中的顺序存储格式

Delete flag 用来标示该元组是否被删除。MyBase 采用假删除方式，只有删除元组的数目达到某一上限是才将被删除的元组从文件中真正删除，然后紧缩文件。

为节省空间，可将所有的 NULL flags 以位图形式存储（这也是 Microsoft® SQL Server® 2000 的存储方法），对该存储结构的改进如下：

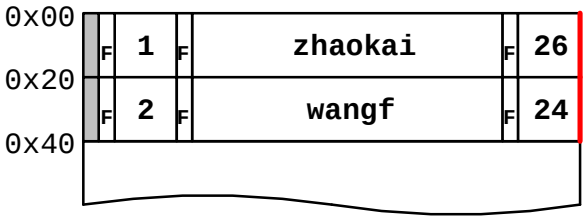


图表 6 改进的元组顺序存储格式

下面用一个例子说明数据插入：

```
INSERT INTO student VALUES(1,'zhaokai',26);
INSERT INTO student VALUES(2,'wangf',24);
```

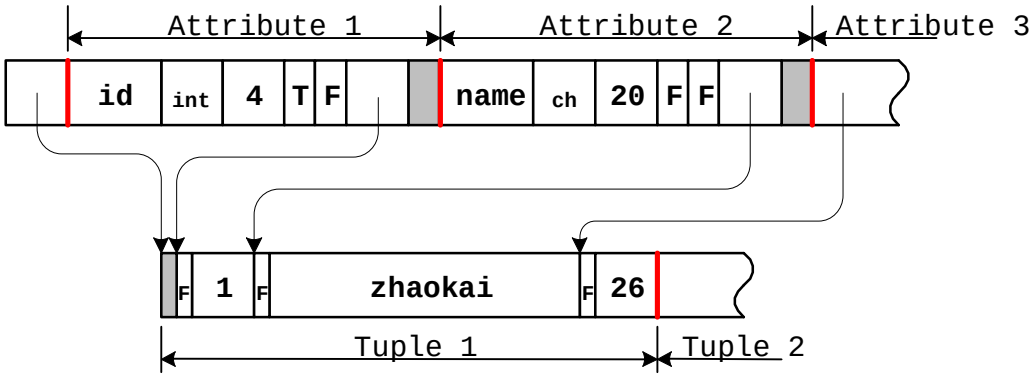
这两条 SQL 语句向 student 数据表插入两个元组。得到的顺序存储结构如下：



图表 7 元组顺序存储示例

4 元组的存取控制

数据库管理系统需要访问某一元组时，需要按照一定的调度策略[5]将元组加载到缓冲区，下面说明如何存取缓冲区中的元组。前面已经介绍过，元组中的每一个属性都有一个内存结构来控制该属性的存取，包括该属性值是否为空、属性的名称、属性的长度、属性的类型、是否为主键，以及指向该属性值在缓冲区内存块中的地址的指针。



图表 8 利用元数据结构进行元组存取的示例

“图表 8”展示了如何利用前面介绍的元数据的数据结构来存取顺序加载到内存缓冲区中得元组。需要说明：在内存中只要保留一套 Attribute 的实例，顺次修改各指针域的地址即可完成对连续元组的遍历操作。

5 结论

本文较为详细的讨论了一个正在设计的数据库管理系统 MyBase 的物理存储结构。MyBase 是一个混合体，它吸取了当前若干 DBMS 的设计理念。由于处于设计阶段，结构还不是非常清楚，摆在前面的问题还很多。希望感兴趣的朋友们多多指教，互相交流，共同进步。

6 参考文献

1. Hector Garcia-Molina, Jeffrey D. Ullman Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000
2. Kalen Delaney. *Inside SQL Server 2000*. Microsoft Press, 2000
3. Ron Soukup. *Microsoft Inside SQL Server 6.5*. Microsoft Press, 1997
4. mSQL 1.0.16 源代码. http://www.hughes.com.au/products/msql/
5. Hbase 中若干 DML 模块的实现. http://www.5iebook.com/soft/312.htm

【责任编辑：pineapple@hitbbs】

[5] 编者注：关于数据库管理系统适用的缓存调度策略，有兴趣的读者可以参阅 Irving L. Traiger. *Virtual Memory Management forData Base Systems*. Operating Systems Review, 1982.

设计一个十分简单的 16 位 CPU

黄海(easyright@163.net)

1 简介

1.1 目的

本项目的目的是设计一个十分简单的基于冯·诺依曼架构的 16 位 CPU。我们将这颗 CPU 命名为 ERVS16-CPU（ EasyRight Very Simple 16-bit CPU ）。 ERVS16 有它自己的指令集。并且，为了测试 ERVS16，我们将在项目的最后用它的指令集编写一个十分简单的操作系统。简单的说，我们在这个项目中只考虑 CPU，寄存器，内存和指令集之间的关系。这就是说我们只需要实现：

- 读/写寄存器
- 读/写内存
- 执行指令集中的所有指令

图 1.1 是 ERVS16 的抽象图

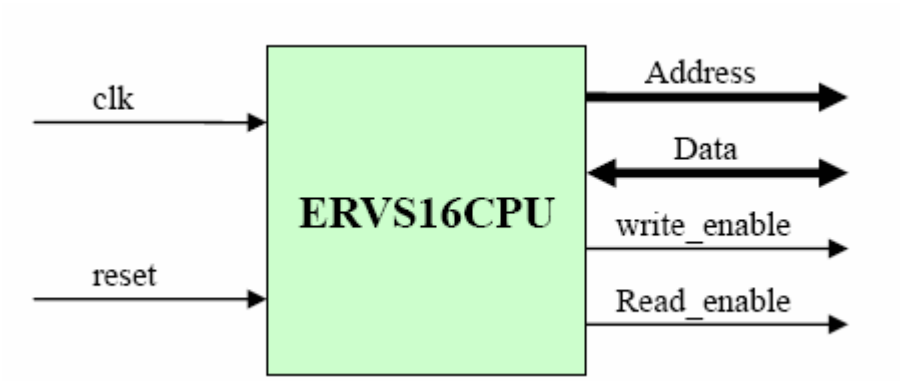


图 1.1 ERVS16 抽象图

假设系统时钟频率是 1MHz，我们将在设计中使用正缘触发时钟频率（Positive Edge Triggered Clocking）技术。复位信号首先输入一个高电平初始化 CPU ，接着当复位信号变为低电平时开始运行位于 0 地址的指令。

内存读/写循环时，要确保可读/可写信号是在低电平。如图 1.2 和 1.3，所有内存读写操作都需要一个信号周期来完成。

时钟技术（Clocking Methodology）定义了信号可以被读写的时间。读写操作不可能同时进行。边缘触发时钟频率技术（Edge-triggered Clocking Methodology）正是被用来防止这样的情况发生。边缘触发时钟频率技术是指机器那存储的所有值都只能在时钟边缘被更新。

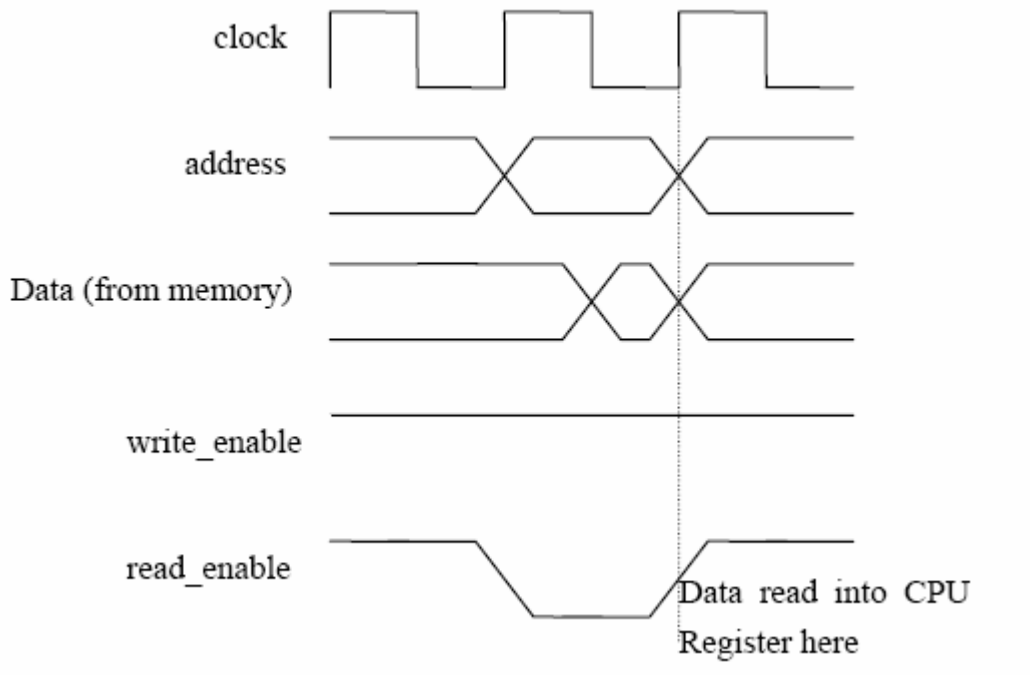


图 1.2　内存读取循环时间图

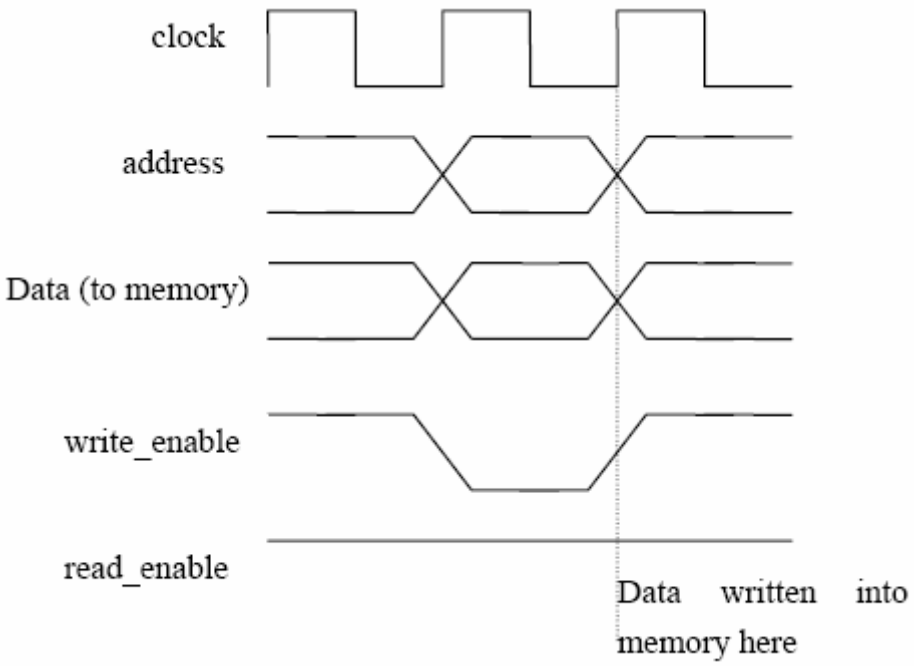


图 1.3　内存写入循环时间图

1.2 测试环境

我们使用 XILINX development board 来测试对 ERVS16 的设计。XILINX development board 包括如下部件：

- 一系列 I/O 设备　（包括一系列开关（Switch）和 7 Segment Displays
- RAM: 1k x 16
- ROM: 1k x 16：它被用于存储测试程序。在模拟开始的时候，　ROM 镜像从一个测试文件（Test File）中

加载。这个文件的每一行都包括由空格隔开的两个值：一个十进制的地址和一个二进制的值。比如：

1 1010100010101010
2 0101011001100110
4 0101011110101000
7 0010101011010101
5 0111000101010000

- 系统：包括内存和 I/O 设备的界面

图 1.4 显示了 development board 的主要组成部分。我们只需要实现红色的部分

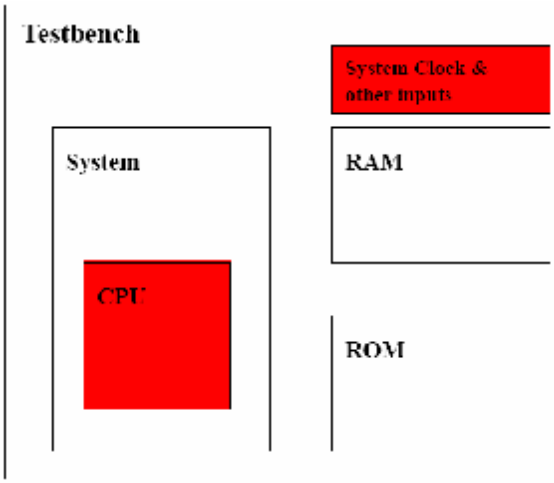


图 1.4 设计结构

下表显示了 development board 上所有设备的技术细节。

设备	地址范围	访问模式
ROM (1KW)	0x0000 – 0x03FF	read only
RAM (1KW)	0x0400 – 0x07FF	read and write
开关	0x8000	read only
7 Segment Displays	0x8001 – 0x8002	read and write
第一个串口	0xC000 – 0xC001	read and write
第二个串口	0xC002 – 0xC003	read and write

2 指令集

2.1 限制和假设

限制

(1) 字长 16 字节
(2) 固定指令长度
(3) 支持子程序
(4) 位移语句的位移量是 2 的倍数

假设

(1) 基于字(word)的寻址方式

(2) 寄存器到寄存器
(3) 8 个 16 位寄存器
(4) 8 个 1 位寄存器
(5) 3 个操作数格式
(6) 当获取指令时 PC 自动增加

2.2 指令及其格式

ERVS16 拥有它自己的指令集，每条指令都与一个硬件操作精确对应。这些指令可以分为四类：

算术指令

机器码	ASM 格式	作用
0001 ddds ssii i000	srl Rd, Rs , Iamt	Rd <= Rs >> Iamt (unsigned)
0001 ddds ssii i001	sra Rd, Rs, Iamt	Rd <= Rs >> Iamt (signed)
0001 ddds sstt t010	sla Rd, Rs, Iamt	Rd <= Rs << Iamt
0001 ddds sstt t011	add Rd, Rs, Rt	Rd <= Rs + Rt
0001 ddds sstt t100	sub Rd, Rs, Rt	Rd <= Rs – Rt
0110 ddds sstt teee	mult Re, Rd, Rs, Rt	Rd <= (Rs * Rt)15:0, Re <= (Rs * Rt)31:16
0111 ddds sstt teee	div Re, Rd, Rs, Rt	Rd <= Rs / Rt, Re <= Rs % Rt
0001 ddds ss00 0101	not Rd, Rs	Rd <= not Rs
0001 ddds sstt t110	and Rd, Rs, Rt	Rd <= Rs AND Rt
0001 ddds sstt t111	or Rd, Rs, Rt	Rd <= Rs OR Rt

测试指令

机器码	ASM 格式	作用
0000 ddds ssii i100	tstlt Bd, Rs, Rt	Bd <= '1' if Rs < Rt else '0'
0000 ddds ssii i001	tstgt Bd, Rs, Rt	Bd <= '1' if Rs > Rt else '0'
0000 ddds ssii i110	tstle Bd, Rs, Rt	Bd <= '1' if Rs <= Rt else '0'
0000 ddds ssii i011	tstge Bd, Rs, Rt	Bd <= '1' if Rs >= Rt else '0'
0000 ddds ssii i010	tsteq Bd, Rs, Rt	Bd <= '1' if Rs = Rt else '0'
0000 ddds ssii i101	tstne Bd, Rs, Rt	Bd <= '1' if Rs != Rt else '0'

内存指令

机器码	ASM 格式	作用
0100 ddd0 iiii iiii	li Rd, immed	(Rd)15:8 <= 0; (Rd)7:0 <= immed
0101 ddd0 iiii iiii	lui Rd, immed	(Rd)15:8 <= immed; (Rd)7:0 <= (Rd)7:0
1100 ddds ssii iiii	lw Rd, immed(Rs)	Rd <= mem[immed + Rs] (signed +)
1101 ddds ssii iiii	sw Rd, immed(Rs)	mem[immed + Rs] <= Rd (signed +)

跳转和分支指令

机器码	ASM 格式	作用
1011 ddd0 0000 0000	jr Rd	PC <= Rd
1010 ddd0 0000 0000	jalr Rd	R7 <= PC; PC <= Rd
1000 dddi iiii iiii	beqz Bd, offset	PC <= PC + offset if Bd = '0' else PC + 1 (signed)

关于上面图标的命名规则：

Rd：存储目标操作数的寄存器，它保存操作结果

Rs：第一个源操作数寄存器
Rt：第二个源操作数寄存器
Bd：存储布尔值的寄存器;
Re：为乘法和除法指令准备的额外寄存器
I：整数

3 数据路径(Datapath)和控制器(Control)

3.1 介绍

ERVS16 基于 MIPS，因此，它有下列特征[1]：

ERVS16 有两个主要部分：数据路径和控制。数据路径负责处理算术操作，而控制器按照程序指令的指示来控制数据路径，内存和 I/O 设备的行为。

在 ERVS16 钟，对于任何一个指令，头两步都是相同的：

(1) 将程序计数器（program counter，PC）送到包含有代码的内存中，并获取后面的指令。

(2) 读取一个或者两个寄存器，用指令空间来选择读取的寄存器。

ERVS16 的功能单元执行包括两种逻辑元素：操作数值的元素和操作数值的元素。

(1) 操作数值的元素（Combinational elements）：它们的输出只取决于当前的输入

(2) 包含状态的元素（Sequential/State elements）：有一些临时存储。一个包含状态的元素至少有两个输入（被写入到元素的数值和时钟）以及一个输出（在前面某个时钟周期写入的值）。包含状态的元素只有在可写（write_enable）信号被声明的一个时钟边缘才能被改写。

3.2 构建数据路径

对于不同的指令，我们将构建不同的数据路径。

3.2.1 获取指令（Fetching instructions）和增加程序计数器值（incrementing PC）

数据路径包括：

a. 存放指令的内存（Instruction memory）：它用来存储一个程序中的指令，是一个包含状态的元素（state element）；

b. 程序计数器：用于保存下一条指令的地址。是一个包含状态的元素；

c. 加法器（Adder）：用于使程序计数器的值指向下一条指令。是一个操作数值的元素。

如上所述，为了运行任何一条指令，我们都必须由从内存中获取这条指令开始。为了做好运行下一条指令的准备，我们还必须增加程序计数器的值使其指向下一条指令，也就是后面的 2 字节（16 位）。图 3.1 显示了这一步的数据路径。

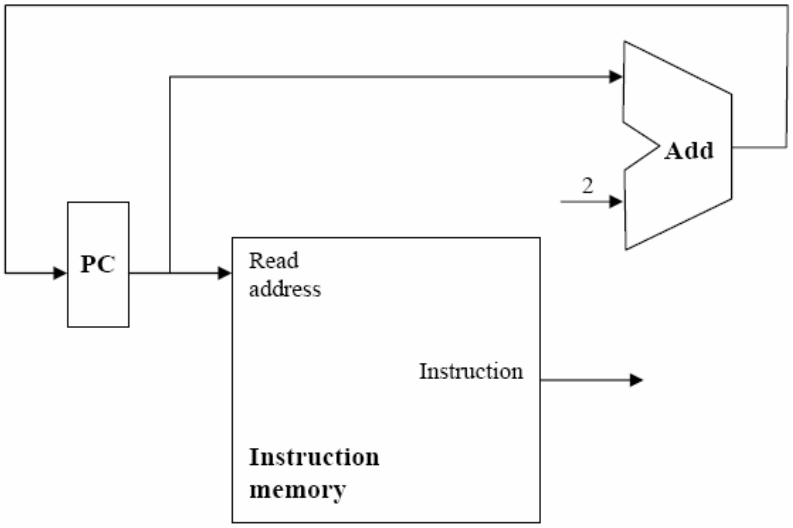


图 3.1 用于获取指令和增加程序计数器值的数据路径

3.2.2 算术和测试指令（不包括 mult, div, not, srl, sra, sla）

ERVS16 的 8 个寄存器被放在一个名为 Register File 的结构中。Register File 是一个可以通过指定某个寄存器在这个区域的编号来对其进行读写的区域。此外，我们需要一个算术逻辑单元（ALU）来对从寄存器读取的值进行操作。

因为这些指令有三个操作数，我们在执行每条指令的时候，需要从 Register File 读取两个字的数据，并且写入一个字的数据到 Register File。为了读取数据，我们需要一个指定寄存器编号的输入来读取该寄存器。为了写入数据，我们需要两个输入：一个用来指定被写入的寄存器编号，另外一个用来提供将要写入到寄存器中的数据。只要指定需要读取的寄存器的编号，Register File 就会输出该寄存器的内容。然而，写操作是被写控制信号（Write Control Signal）控制的，在执行写操作之前的时钟边缘（Clock Edge）必须先声明它。因此，我们总共需要四个输入（三个用于 3 字节的寄存器编号，一个用于 16 字节的数据）以及两个输出（都是用于 16 字节的数据）以及一个信号（1 字节的 reg_write）。

ALU 被四字节的信号所控制。它接收两个 16 字节的输入，并且输出一个 16 字节的结果。

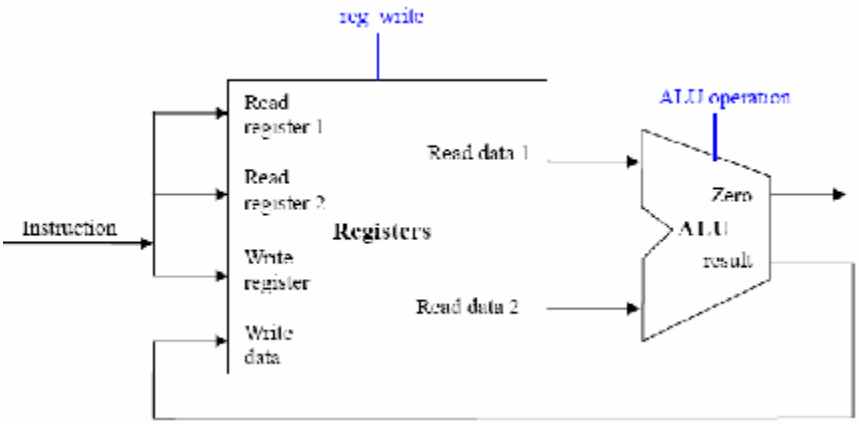


图 3.2 算术和测试指令的数据路径(不包括 mult, div, not, srl, sra, sla)

3.2.3 读取字（Load word ，lw） 和存储字（store word ，sw） 指令

这两条指令通过用基址寄存器（Base Register，Rd）的值加上指令中包含的 6 字节偏移量来计算出内存地址。因此，我们需要用到 Register File 和 ALU。

另外，我们需要一个单元（Unit）来将 6 位的偏移量扩展为带符号的 16 位值，并使用一个存储数据的内存单元（Data Memory Unit）来进行读取或者写入的操作。正如第一章提到的那样，存储数据的内存单元的读写操作受可读（read_enabel）和可写（write_enable）信号控制，它们必须在读写操作的时钟边缘之前被声明。它会获取一个地址输入和一个被写入内存的数据的输入。

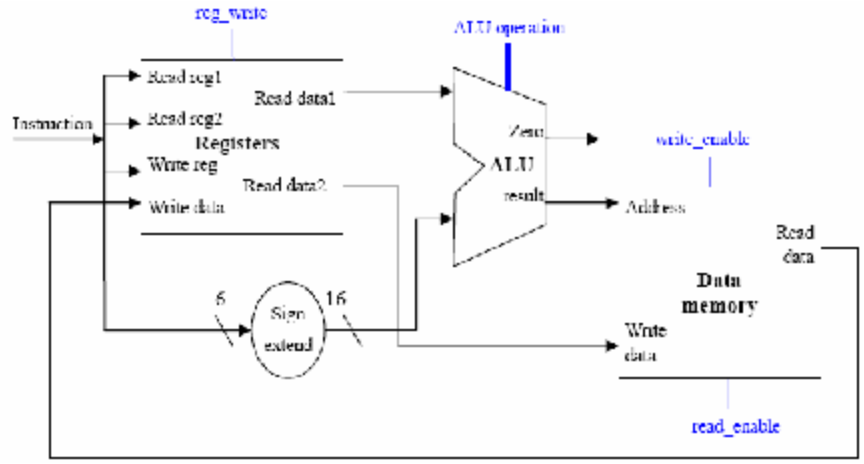


图 3.3 寄存器的读取或者存储访问，然后是内存地址计算，从内存中读写，在读取一条指令后向 RegisterFile 中写入的数据路径

3.2.4 beqz 指令

为了实现这条指令，我们必须计算分支跳转语句的目标地址（Branch Target Address），该地址等于扩展后的指令中的偏移量加上 PC 的值。我们应当注意：

- 分支跳转语句的目标地址的基址是下一条指令的地址；
- 因为寻址方式是字(word)寻址，所以偏移量要左移 1 位；左移功能使寻址范围增加了 2 倍。

分支数据路径必须完成两个操作：计算分支跳转语句的目标地址并决定寄存器的值是否为零。如果我们用 0 减去寄存器的值并且我们得到的结果为零，这就是说寄存器的值为零。

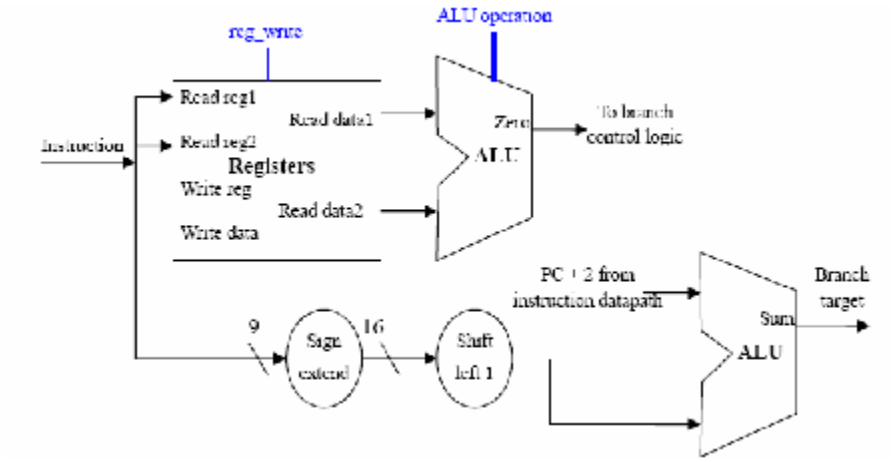


图 3.4 分支的数据路径使用 ALU 来判断分支条件，一个独立的加法器(Adder)计算分支目标作为 PC 的增量，并且扩展指令的低 9 位，偏移量左移 1 位

3.2.5 综合以上数据路径

在本节中，我们要将前面章节所提到的所有数据路径。然后我们会强化我们的设计，使其支持 ERVS16 的所有指令集。

我们从简单的设计开始：综合的数据路径（Combined Datapath）在一个时钟周期内完成所有指令。这就是说所有数据路径资源（Datapath Resource）都不可能在同一条指令中使用超过一次，所以所有使用超过一次的元素（Element）都必须被复制（Duplicated）。这就是为什么我们需要数据存储器（Data Memory）和指令存储器（Instructions Memory）。

然而，许多元素可以被许多不同的指令流（Instructions Flows）共享。要在两个不同的指令类（Instruction Classes）之间共享数据路径元素（Datapath Element），我们必须允许建立到输入元素的多重连接（Multiple Connections），并且要有一个控制信号在输入之间进行选择。这样的选择通常是由一个名为多路复用器（Multiplexor）的设备完成。多路复用器通过设置控制行（Control Lines）从而在不同的输入之间进行选择。

(1) 为内存指令、算术指令和测试指令整理数据路径

图 3.2 和图 3.3 十分接近，但是它们有两处不同：

- ALU 的第二个输入是一个寄存器（算术指　令和测试指　令）或者是符号扩展过的指　令低位字节（内存指　令）；
- 值被储存在一个来自 ALU 的目的寄存器（Destination Register）中（算术指　令和测试指　令）或者内存（load）中。

我们可以使用两个多路复用器来解决这两个不同点。参见图 3.5，绿色的部分是在图 3.3 的基础上更改的。

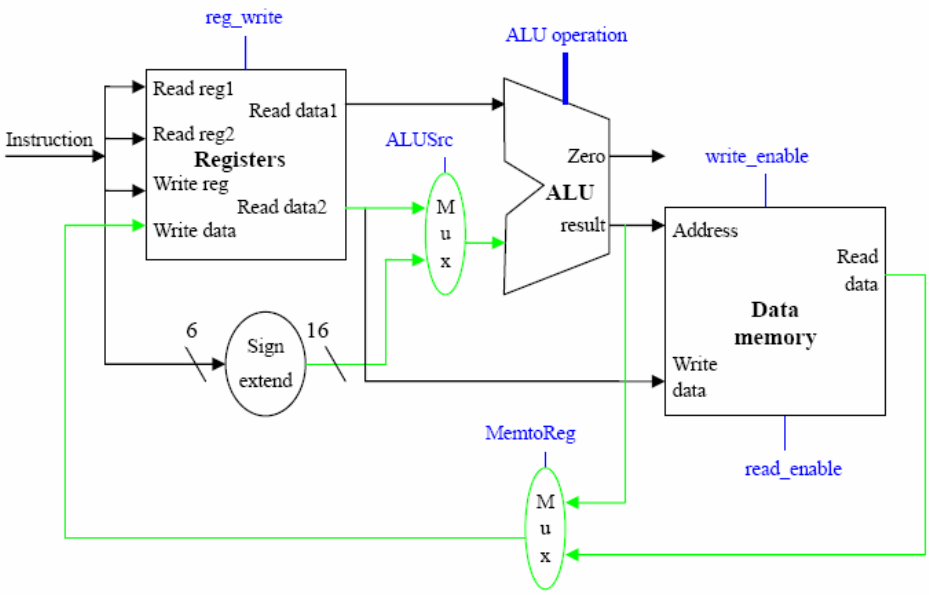


图 3.5 内存指令、算术指令和测试指令的复合数据路径

(2) 为获取指令和增加程序计数器（PC）增加数据路径

获取指令和增加程序计数器（图 3.1）操作可以很简单的加入到图 3.5 的数据路径中。复合的数据路径包括一个指令存储器（Memory for Instructions）和一个单独的数据存储器（Memory for Data）。复合的数据路径

需要一个加法器和一个 ALU，加法器被用来增加程序计数器，ALU 被用来在同一个周期中执行指令。

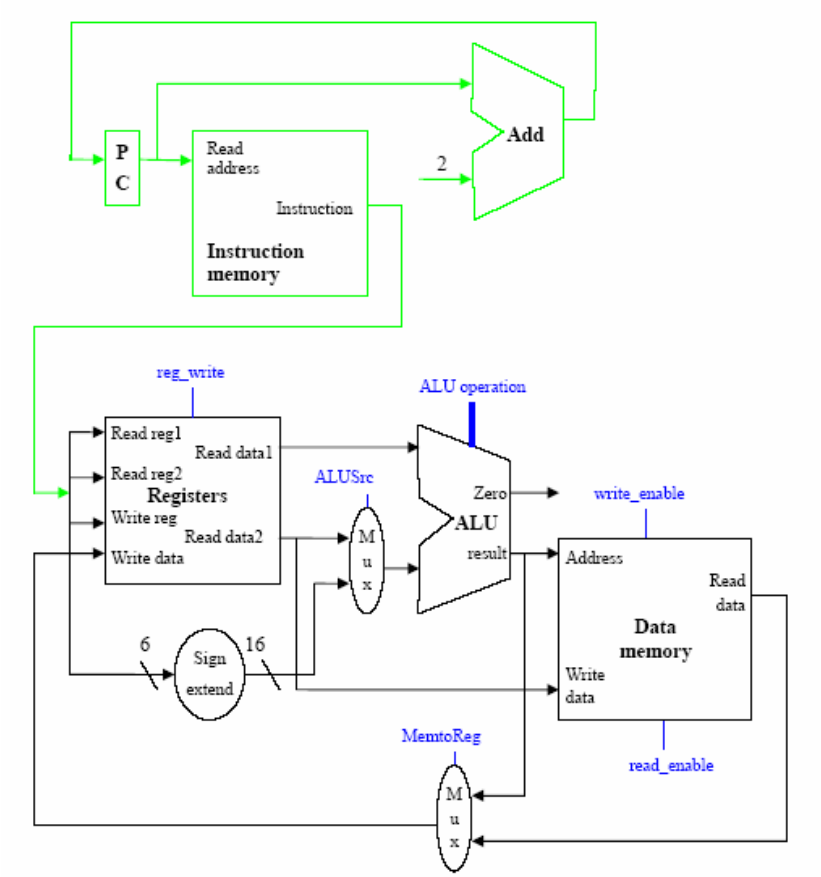


图 3.6 图 3.1, 3.2, 3.3 的复合数据路径

(3) 为 beqz 指令增加数据路径

beqz 指令使用主要的 ALU 来判断寄存器操作数是否为 0。另外需要两个多路复用器。一个（PCSrc）被用来选择是根据指令地址（PC+2）还是将分支目标地址（Branch Target Address）写入 PC。另外一个（BeqzSrc）被用来选择是将 ALUSrc 的值还是将 0 输入到主 ALU 中。

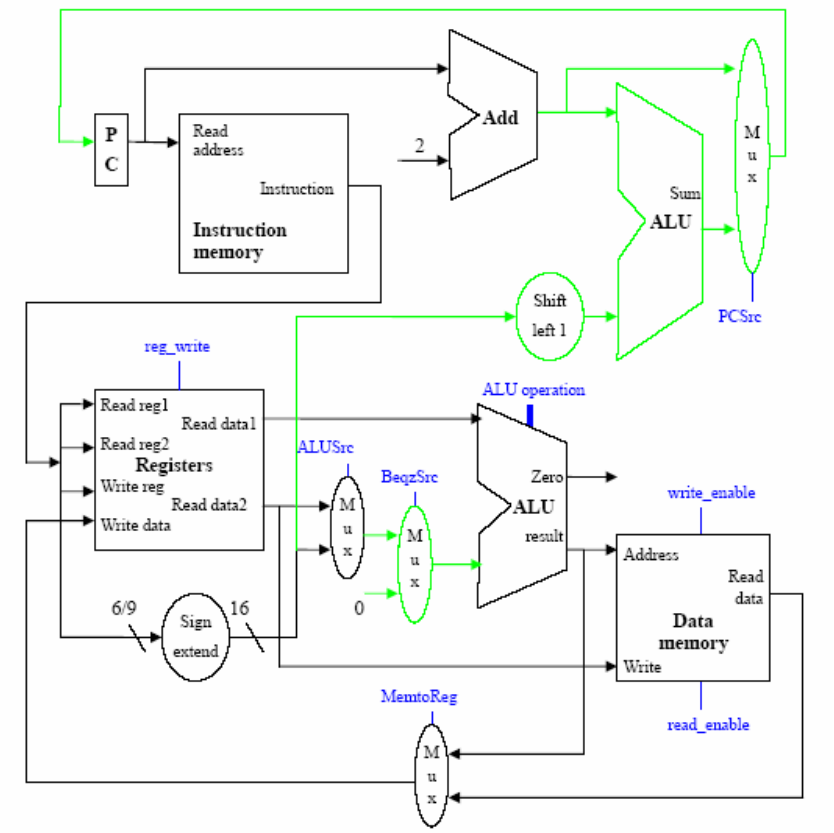


图 3.7　图 3.1, 3.2, 3.3, 3.4 的复合数据路径

(4) 为跳转指令：jr and jalr 增加数据路径

跳转指令 jr 和 jalr 有相同的功能：PC <= Rd。jalr 多一步：R7 <= PC。

我们需要扩展多路复用器（PCSrc）来增加一个选项：存储在 Rd 中的跳转目标地址。PCSrc 就有三个选项，并且，需要两个控制行。

为了将 PC 中的值存储到 R7，我们需要两个多路复用器（PCorData 和 RegorPC）进行实现。参见图 3.8。

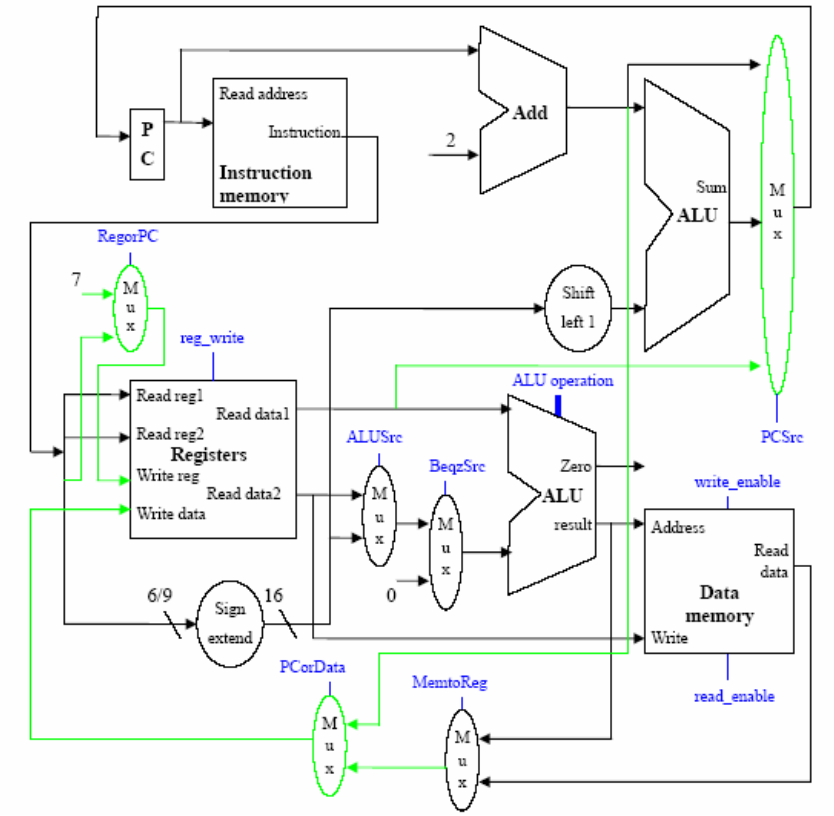


图 3.8　图 3.7 和跳转指令的复合路径

(5) 增加 li 和 lui 指令

为了实现 li 和 lui 指令，我们会用到主 ALU（Main ALU）来左移指令的低八位字节。指令的低八位字节在 li 中会向左偏移 0 字节，在 lui 中会向左便宜 8 字节。因此我们需要扩展多路复用器（BeqSrc）使其能够输入 8。并且我们还需要增加一个多路复用器（LISrc），它包含两个选项：从 Register File 中获取数据或者从符号扩展的指令域（Field of Instructions）获取数据。

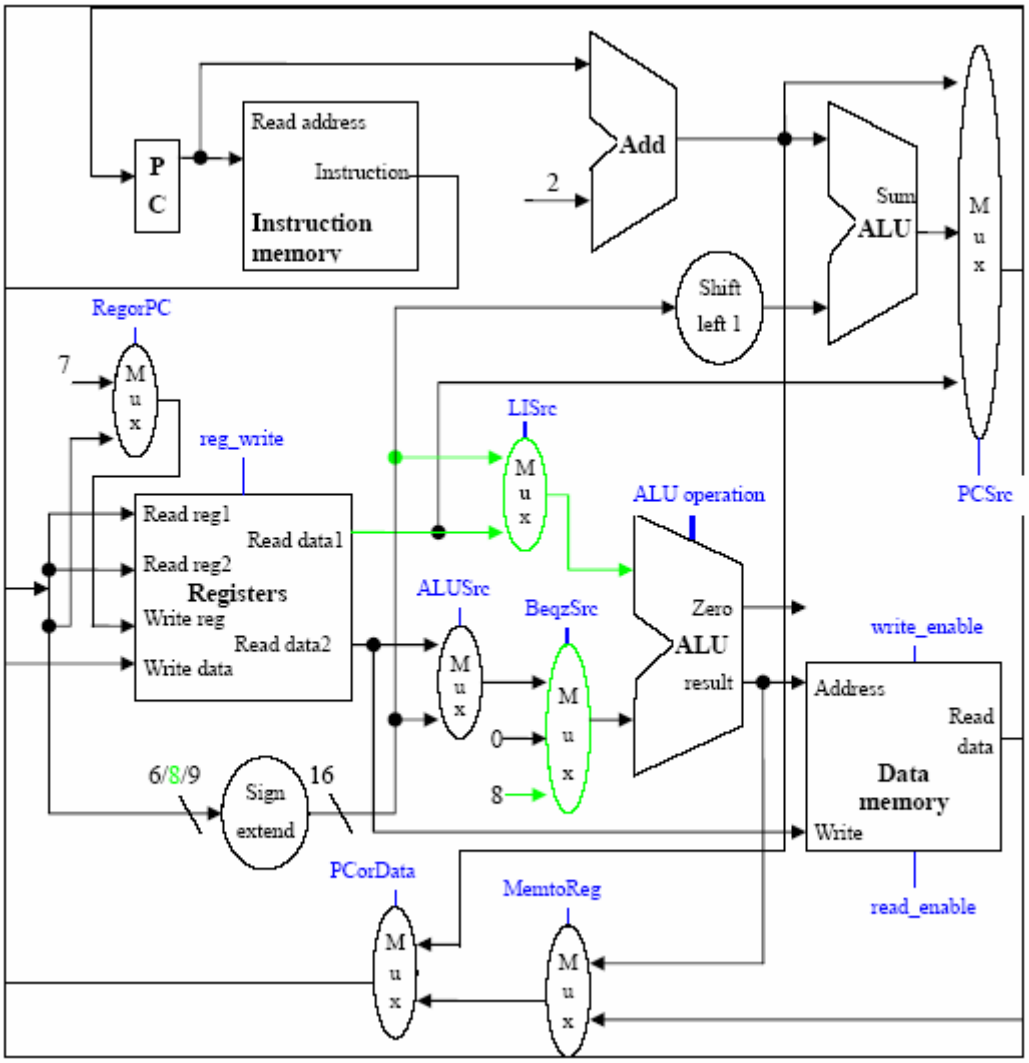


图 3.9　图 3.8 和 li，lui 指令的复合数据路径

(6) not，shift，mult，div 指令

我们假设 not 和 shift 指令在 ALU 内部实现。因此除了操作数的数量和位置外，我们并不需要过多的更改图 3.9。我们将在下一节讨论细节问题。

我们将在下一章实现 mult 和 div 指令：多周期数据路径。

3.3 控制单元（Control Unit）

控制单元必须能够获取输入，并且为每个状态元件（State Element）生成一个写信号（Write Singal），选择器（Selector）控制每个多路复用器以及 ALU 控制器。

3.3.1 ALU 控制器

我们假设 ALU 有下列功能和各自的控制输入：

ALU control input	Function
0000	And

0001	Or
0010	Add
0110	Subtract
0111	set on less than
0101	Not
1100	shift right logical
1000	shift right arithmetic
1001	shift left arithmetic

取决于指令类（Instruction Class），ALU 将会实现上面的其中一个功能。对于算术和测试指令，ALU 实现上面的功能需要取决于指令的低阶（low-order）字段的一个 3 字节的 Funct Field。对于 load word 和 store word 指令，我们使用 ALU 来计算内存地址。对于 li 和 lui，我们使用 ALU 的左移算术操作。对于 beqz，ALU 必须完成一个减法操作。

Instruction	**ALUop**	**Instruction type**	**Funct field**	**Desired ALU action**	**ALU control**	**Write reg**	**Read reg 1**	**Read reg 2**	**Write reg 2**
srl	0001	0001	000	shift right logical	1100	9-11	6-8	3-5	
sra	0001	0001	001	shift right arithmetic	1000	9-11	6-8	3-5	
sla	0001	0001	010	shift left arithmetic	1001	9-11	6-8	3-5	
add	0001	0001	011	add	0010	9-11	6-8	3-5	
sub	0001	0001	100	subtract	0110	9-11	6-8	3-5	
mult		0110				9-11	6-8	3-5	0-2
div		0111				9-11	6-8	3-5	0-2
not	0001	0001	101	not	0101	9-11	6-8	xxx	
and	0001	0001	110	and	0000	9-11	6-8	3-5	
or	0001	0001	111	or	0001	9-11	6-8	3-5	
tstlt	0000	0000	1xx	set on less than	0111	9-11	6-8	3-5	
tstgt	0000	0000	1xx	set on less than	0111	9-11	6-8	3-5	
tstle	0000	0000	1xx	set on less than	0111	9-11	6-8	3-5	
tstge	0000	0000	1xx	set on less than	0111	9-11	6-8	3-5	
tsteq	0000	0000	0xx	subtract	0110	9-11	6-8	3-5	
tstne	0000	0000	0xx	subtract	0110	9-11	6-8	3-5	
li	01xx	0100		shift left arithmetic	1001	9-11		0-7	
lui	01xx	0101		shift left arithmetic	1001	9-11		0-7	
lw	11xx	1100		add	0010	9-11	6-8	0-5	
sw	11xx	1101		add	0010	9-11	6-8	0-5	
jr	xxxx	1011			xxxx		6-8		
jalr	xxxx	1010			xxxx	7	6-8		
beqz	10xx	1000		subtract	0110	9-11		0-8	

从上表中我们可以发现指令 li，lui，sw 和 beqz 有不同的长度：6，8，9。然而，在 li 和 lui 中，第八个字节是 0。因此，只有两个选项：6 或者 9，并且它需要一个多路复用器 SworLW。 并且，为了实现所有测试指令，我们需要将 0 和 ALU 的输出结果通过 AND，OR 和 NOT 门进行综合，并且通过多路复用器 TestType 选择结果。我们必须将测试代码与其他代码通过多路复用器 TestIns 区分开。

介绍 sw，我们需要读取 Rd 的内容，Rd 是指令的第 9，10，11 字节。因此我们需要一个额外的多路复用器：SWSrc。

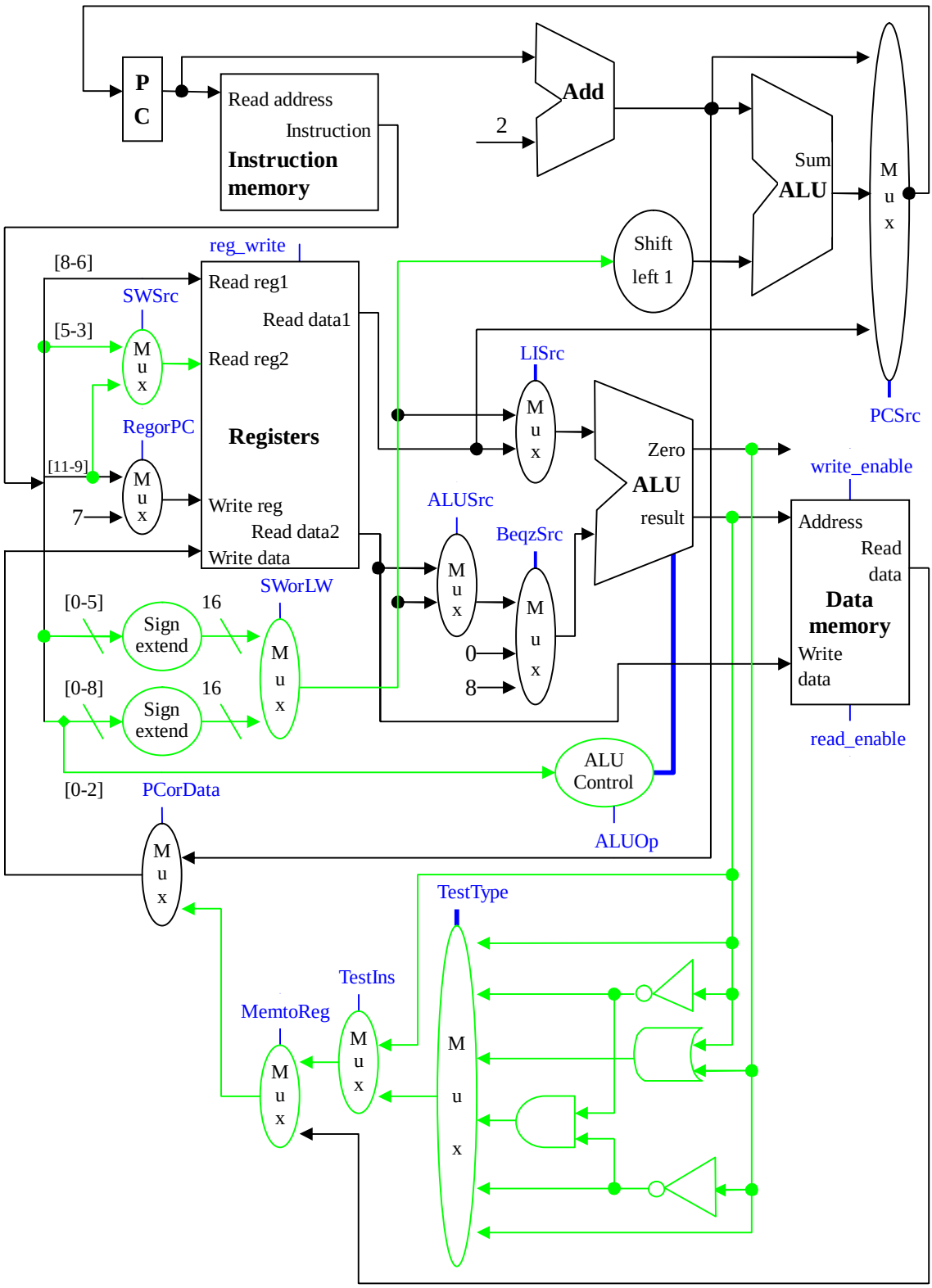


图 3.10　图 3.9 加上所有测试指令和所有必须的多路复用器和控制行

3.3.2 主控单元

主控单元取决于指令的高 4 位。并且主控单元确立了每个多路复用器的控制信号。

Instruction	SWorLW	ALUSrc	BeqzSrc	LI Src	Test Ins	MemtoReg	PCorData	reg_write	RegorPC	PC Src	write_enable	read_enable	Test Type	SW Src
srl sra sla	0	1	0	1	0	0	1	1	0	0	0	0	x	x
add sub and or	x	0	0	1	0	0	1	1	0	0	0	0	x	0
not	x	x	1	1	0	0	1	1	0	0	0	0	x	x
tstlt tstgt tstle tstge tsteq tstne	x	0	0	1	1	0	1	1	0	0	0	0	0 3 2 1 5 4	0
li lui	1	x	1 2	0	0	0	1	1	0	0	0	0	x	x
lw	0	1	0	1	x	1	1	1	0	0	0	1	x	x
sw	0	1	0	1	x	x	x	0	x	0	1	0	x	1
jr	x	x	x	x	x	x	x	0	x	2	0	0	x	x
jalr	x	x	x	x	x	x	0	1	1	2	0	0	x	x
beqz	1	x	1	1	x	x	x	0	x	1	0	0	x	x

现在我们可以讨论单周期数据路径的最终实现。

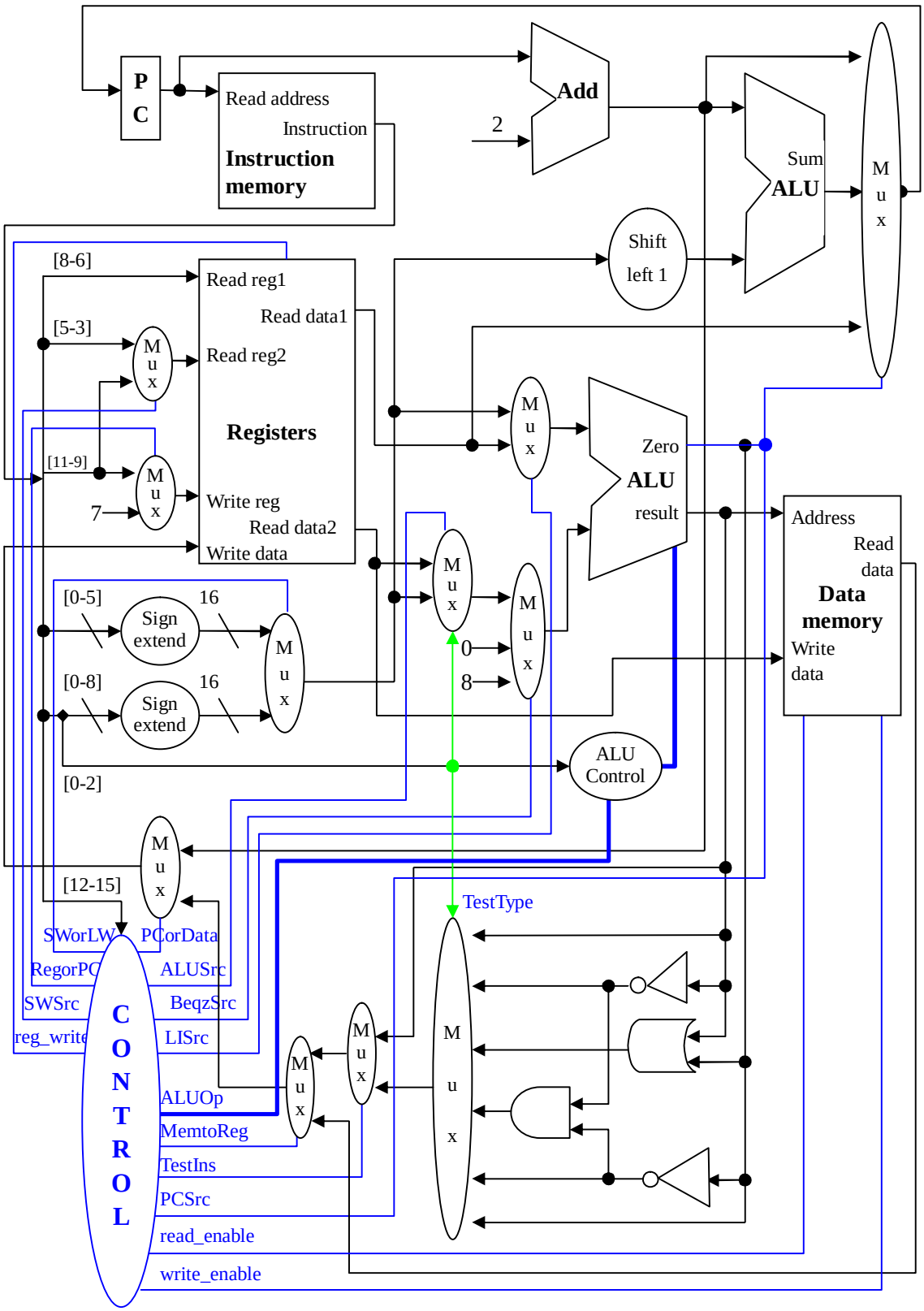


图 3.11　完整的单周期数据路径

4 多周期数据路径（Multicycle Datapath）

在上一章，我说明了怎么建立一个简单的单周期数据路径（single-cycle datapath）和实现大部分的指令集。在这一章，我将改变单周期数据路进为多周期数据路径，实现所有 ERVS16 的指令集。我需要修改 ALU 和它的控制信号。

4.1 单周期 VS 多周期

在多循环周期设计中，不同的指令使用不同数量的 CPU 循环，在执行一条指令时可以重用某些功能部件。尽管单周期路径很简单，但是它很少在现代的设计中使用,因为以下几点原因：

- 单周期路径的效率低，单周期数据路径中的指令的 CPU 循环长度是相同的，该长度是由需要最长 CPU 循环长度的指令(LW)决定的， 因此在单周期数据路径中 CPI 总是为 1；
- 单周期路径的硬件也是昂贵的。单周期路径需要两个内存单元和两个加法器，但是多周期路径只需要一个内存和不需要加法器。尽管多周期路径需要一些额外的寄存器和多路复用器（multiplexers），它还是比单周期路径便宜。

单周期和多周期不同是：

- 指令和数据保存在一个单独的内存单元中；
- 只用一个 ALU 而不是一个 ALU 和两个加法器；
- 在每个主功能单元（Major Functional Unit）后加入一个或多个寄存器，用以保存其输出直到这个值在下一个时钟周期中被使用。

4.2 多周期实现

在多周期中，每一步最多包含一次 ALU 运算，或者一次 Register File 访问，或者一次内存访问。这时，时钟周期等于以上各个操作的执行时间的最大值。

为了实现多周期数据路径，我们需要临时的寄存器来存储这个指令的数据以便下一个周期使用。

- 指令寄存器（Instruction Register，IR）和存储器数据寄存器（Memory Data Register，MDR）分别用于保存从内存输出的指令和数据；
- A 和 B 寄存器用于保存从 Register File 读出寄存器操作数的值；
- ALUOut Register 保存 ALU 的输出

我们将增加 ALU 的功能使它能够计算这些运算：与，或，加，减，逻辑右移，算术右移，算术左移，**==**，**!=**，**<**，**>**，**<=**，**>=**。

而且，我们需要另外增加多路复用器并扩展已有的多路复用器，参见图 4.1。

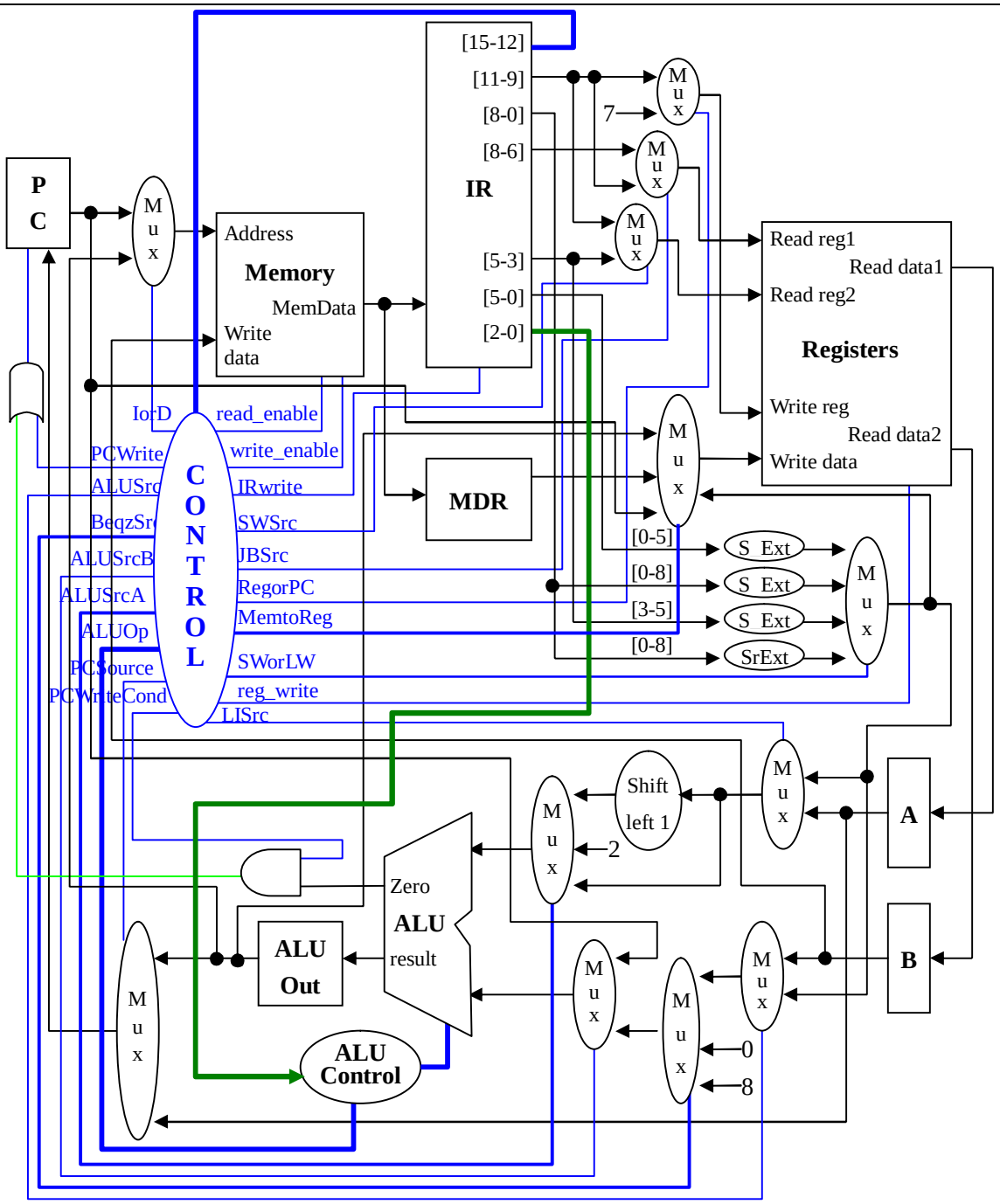


图 4.1　多周期数据路径

4.3 指定控制信号

在这节中，我们将使用有限状态机来指定控制信号。所有指令从状态 0 开始。

在下面个表格列出不同类型的指令和它们的状态：

Shift	Arithmetic & Tests	lui	li	lw	sw	jr	jalr	beqz
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
9	7	11	10	2	2	12	13	6

8	8	8	0	3	5	0	12	0
0	0	0		4	0		0	
				0				

图 4.2 列出了完整的多周期路径的有限状态机控制信号。

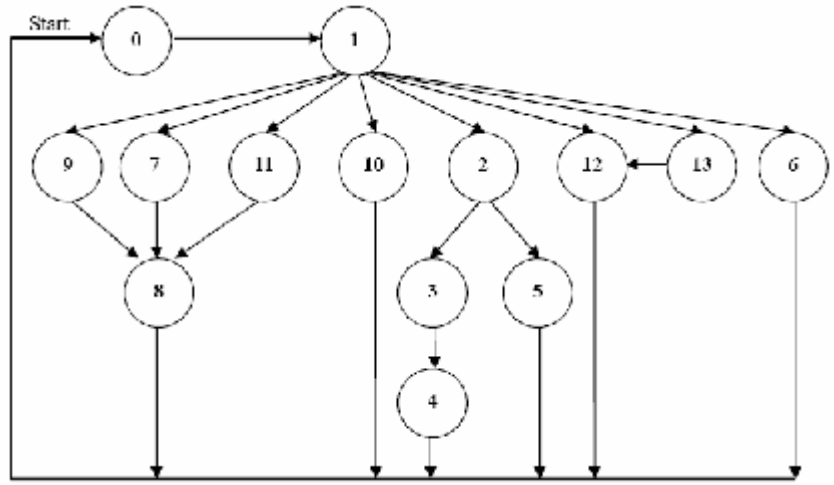


图 4.2 完整的多周期路径的有限状态机控制信号

每个状态的操作和控制信号请看下表：

(0) Instruction fetch read_enable ALUSrcB = 0 IorD = 0 IRWrite ALUSrcA = 1 ALUop = 11xx PCWrite PCSource = 0 Jump to state 1.	(1) Instruction decode / register fetch / PC + Branch Offset [9 bits] SWorLW = 1 LISrc = 0 ALUSrcA = 0 ALUSrcB = 0 ALUop = 11xx For “sw, lw” jump to state 2; For “beqz” jump to state 6; For “Arithemtic, Test” jump to state 7; For “Shift” jump to state 9; For “li” jump to state 10. For “lui” jump to state 11. For “jr” jump to state 12; For “jalr” jump to state 13;	(2) Memory address computation JBSrc = 0 LISrc = 1 ALUSrcA = 2 SWSrc = 1 SWorLW = 0 ALUSrc = 1 BeqzSrc = 0 ALUSrcB = 1 ALUop = 11xx For “lw”, jump to state 3; For “sw”, jump to state 5.
(3) Memory access read_enable IorD = 1 jump to state 4.	(4) Memory read completion step RegorPC = 0 reg_write MemtoReg = 1	(5) Memory access write_enable IorD = 1 Jump to state 0.

	Jump to state 0.	
(6) Branch completion JBSrc = 1 LISrc = 1 ALUSrcA = 2 BeqzSrc = 1 ALUSrcB = 1 ALUop = 10xx PCWriteCond PCSource = 0 SWSrc = 1 Jump to state 0.	(7) Execution JBSrc = 0 LISrc = 1 ALUSrcA = 2 SWSrc = 0 ALUSrc = 0 BeqzSrc = 0 ALUSrcB = 1 ALUop = 000x Jump to state 8.	(8) Arithmetic and Test completion RegorPC = 0 MemtoReg = 0 reg_write Jump to state 0.
(9) Shift JBSrc = 0 LISrc = 1 ALUSrcA = 2 SWorLW = 2 ALUSrc = 1 BeqzSrc = 0 ALUSrcB = 1 ALUop = 000x Jump to state 8.	(10) Li SWorLW = 1 MemtoReg = 2 reg_write Jump to state 0.	(11) Lui SWorLW = 1 LISrc = 0 ALUSrcA = 2 BeqzSrc = 8 ALUSrcB = 1 ALUop = 01xx Jump to state 8.
(12) Jr JBSrc = 1 PCSource = 1 PCWrite Jump to state 0.	(13) Jalr RegorPC = 1 MemtoReg = 3 reg_write Jump to state 12.	

致谢

感谢所有对本项目提供无私帮助的朋友。

如果您对本项目有兴趣，我们欢迎您的加入，请发送 email 到： support@easyright.net.

本文保持更新。若想了解更多信息，请访问 http://www.easyright.net.

参考书目

[1] David A. Patterson and John L. Hennessy “Computer Organization & Design – THE HARDWARE/SOFTWARE INTERFACE”

【责任编辑：kylix@hitbbs】

Hello China 的体系结构

Garry Xin(garryxin@yahoo.com.cn)

1 Hello China 总体概述

在本章里，我们对 Hello China 的开发目标、开发过程，以及开发过程中的控制手段、开发环境等内容，进行概述。如果对本文所涉及的任何内容，有疑问或更正，请发 Email 到：garryxin@yahoo.com.cn。

1.1 Hello China 的总体目标

开始开发的时候，为 Hello China 制订的应用目标是，为一些嵌入式设备（比如，PDA，机顶盒，路由器/交换机/DSLAM 等网络设备，图形/多媒体终端，测量仪器等）提供基础的软件平台，这些嵌入式设备有的是实时性的，比如网络通信设备，而有的则不需要很强的实时性，比如 PDA，多媒体终端等，在开发的时候，Hello China 同时考虑了这两种不同的应用，可以通过修改不同的参数（然后重新编译和连接），来分别适应这些不同的需求。

Hello China 还可以应用于个人计算机（PC），比如一些基于 PC 计算机结构的控制终端（工业控制终端、营业终端等）。

因此，总结起来，Hello China 适应于实时性的应用，也适应于非实时性的应用，具有良好的目标适应性，比较典型的有：

- **基于 PC 机的控制终端**
- **基于实时性要求的网络通信设备**
- **基于非实时性要求的图形终端**
- **基于非实时性要求的 PDA**
- **基于半实时性要求的机顶盒等。**

1.2 Hello China 的开发历程概述

Hello China 从 2004 年 3 月份开始开发，到 2005 年 3 月份为止（Version 1 完成），历时一年的时间。

在这期间，Hello China 的体系结构、组成模块以及功能目标等，都有过很多的调整，但总体目标没有改变，而且开发完成之后，其具备的功能和性能，以及体系结构，跟预期的开发目标（应用目标）一致。

1.3 Hello China 的开发过程控制

在 Hello China 的开发过程中，采用了一些基础的开发控制手段，开发过程中，形成了完善的文档体系，在开发过程中，遵循模块化开发方法，把系统的每个功能划分成模块，然后以面向对象的方式，来封装整个模块。

在开发单独模块的时候，首先完成概要设计（英文），把该模块的主要功能、对外接口、核心功能实现方法等进行描述，形成整体的框架。

概要设计形成之后，再根据概要设计形成详细设计文档（英文），在详细设计文档中，对功能模块的每个小功能（函数）进行代码级的描述，并对每个函数的算法、接口（输入/输出）进行详细描述。

在详细设计形成之后，再根据详细设计进行编码，调试，最终该模块跟整体连接，测试通过后，再进行下一个模块的编写。

在编写一个模块的时候，有的时候需要对其它模块进行修改，这时候同步更新相应的文档体系。

1.4 Hello China 的开发环境概述

操作系统的开发，涉及到各种各样的功能模块，比如引导功能模块、初始化功能模块、硬件驱动功能模块以及系统核心等，这些功能模块很难使用同一个开发环境进行代码的编译和编写，因此，在 Hello China 的开发过程中，针对不同的功能模块，使用了不同的开发环境，主要有：

- Ø 针对引导功能和硬件驱动程序性（比如，键盘驱动程序和字符显示驱动程序），采用汇编语言编写，使用 **NASM 编译器**编译；
- Ø 针对操作系统核心，为了提高移植性和开发效率，采用 C 语言编写，采用 **Microsoft Visual C++**作为代码的编译和编写环境；
- Ø 由于上述开发环境最终形成的目标格式，有时候跟预期的格式不一致，于是采用 Microsoft Visual C++ 编写了**目标处理工具**，这套工具对上述编译器形成的目标文件进行进一步处理，形成计算机可以直接加载并运行的二进制模块。

因此，Hello China 在开发过程中，涉及到的开发环境和开发工具比较多，但该操作系统的核心代码，却完全是使用 C 语言编写的，具备良好的可移植性，其它非 C 语言编写的代码，数量非常少，相对来说是微不足道的，根据粗略统计，非 C 语言编写的代码数量，是总体代码数量的 5%左右。

1.5 Hello China 的规格

Hello China 的相关规格和特性参见下表：

规格名称	规格	备注
任务模型	多任务操作系统	N/A
最大任务数量	128	修改代码可以调整
任务切换机制	抢占式	
任务调度算法	基于优先级的调度	修改代码可以更改
IPC 机制	消息、共享内存等	
内存模型	共享内存	
支持最大物理内存数量	4G	
核心代码所占物理内存	2M	含初始化数据
核心占用物理内存池	16M	含驱动程序加载占用，可调节
页面大小	4K	可以修改
页面维护算法	伙伴算法	
核心内存分配算法	空闲链表与页面位图	
任务内存分配算法	Chunk 与空闲链表	
中断机制	基于向量表的中断机制	

最大中断数量	256	根目标 CPU 一致
最大异常数量	32	跟目标 CPU 一致
支持的 CPU 类型	I32	核心代码可移植
系统时钟频率	10Hz	可调整
驱动程序模型	可加载模块	
系统调用数量	16	可扩展
支持文件系统	FAT32/NTFS	
支持总线类型	PCI/USB	
用户界面	字符/图形	暂时只支持字符界面
输入设备	Keyboard/Mouse	
消息队列长度	32/核心线程	可以修改

2 开发中的面向对象机制

在 Hello China 核心模块的开发中，虽然使用的是 C 语言，但在开发过程中，引入了面向对象的编程与开发思想，把整个核心模块分成一系列对象（比如，内存管理器对象，核心线程管理器对象，页框管理器对象，对象管理器等）来实现，这样实现起来，独立性更强，而且具备更好的可移植性和可裁减性。

但 C 语言本身是面向过程的语言，用 C 语言来进行面向对象的编程，需要对 C 语言做一些简单的预处理，在 Hello China 的开发中，我们先预定义了一系列宏，用来实现面向对象的编程机制，另外，针对 Hello China 的特点，定义了一个对象管理框架，所有开发过程中的对象，都统一纳入对象管理框架中。

在本章中，我们简单描述一下开发过程中的面向对象机制。

2.1 C 语言的面向对象预处理

C 语言是面向过程的语言，缺省情况下不支持面向对象机制，但并不是说，C 语言无法实现面向对象的开发模式，实际上，合理的使用 C 语言，也可以实现面向对象的开发。在 Hello China 的开发中，我们对 C 语言进行了一番预处理，并充分利用了 C 语言的函数指针和结构体机制，来实现了简单的面向对象开发功能。

在 Hello China 的开发中，我们充分利用 C 语言的宏定义机制，以及 C 语言的函数指针机制，实现了下列简单的面向对象机制：

2.1.1 使用结构体定义实现对象

面向对象开发的一个核心思想就是对象，即把任何可以类型化的东西看作对象，而把程序之间的交互以及调用，以对象之间传递消息（实际上就是对象成员函数的调用）的形式来实现。在面向对象的语言中，比如 C++，专门引入了对象类型定义机制（比如，class 关键字），但 C 语言中却没有专门针对面向对象思想，引入对象类型定义机制，但 C 语言中的结构体定义，却十分适合定义对象类型（实际上，在 C++语言中，struct 关键字也用来定义对象类型）。比如，在 C++语言中，定义一个对象类型如下：

```
class __COMMON_OBJECT{
private:
   DWORD    dwObjectType;
   DWORD    dwObjectSize;
Public:
```

```
    DWORD    GetObjectType();
    DWORD    GetObjectSize();
};
```

如果利用 C 语言的 struct 关键字，也可以实现类似的对象定义：

```
struct __COMMON_OBJECT{
    DWORD    dwObjectType;
    DWORD    dwObjectSize;

    DWORD    (*GetObjectType)(__COMMON_OBJECT* lpThis);
    DWORD    (*GetObjectSize)(__COMMON_OBJECT* lpThis);
};
```

与 C++不同的是，C 语言定义的成员函数，增加了一个额外参数：lpThis，这是最关键的一点，实际上，C++语言在调用成员函数的时候，也隐含了一个指向自身的参数（this 指针），因为 C 语言不支持这种隐含机制，因此需要明确的指定指向自身的参数。

这样就可以定义一个对象：

```
__COMMON_OBJECT CommonObject;
```

如果调用对象的成员函数，在 C++里面，如下：

```
CommonObject.GetObjectType();
```

而在 C 语言中（参考上述定义），则可以这样：

```
CommonObject.GetObjectType(&CommonObject);
```

使用这种思路，我们简单实现了 C 语言定义对象的基础支撑机制。

2.1.2 使用宏定义实现继承

面向对象的另外一个重要思想，就是实现继承，而 C 语言不具备这一点。为了实现这个功能，我们在定义一个对象（结构体）的时候，同时也定义一个宏，比如，定义如下对象：

```
struct __COMMON_OBJECT{
    DWORD    dwType;
    DWORD    dwSize;
    DWORD    GetType(__COMMON_OBJECT*);
    DWORD    GetSize(__COMMON_OBJECT*);
};
```

同时，定义如下宏：

```
#define INHERIT_FROM_COMMON_OBJECT \
    DWORD  dwType; \
    DWORD  dwSize; \
    DWORD  GetType(__COMMON_OBJECT*); \
    DWORD  GetSize(__COMMON_OBJECT*);
```

这样，假设另外一个对象从该对象继承，则可以这样定义：

```
struct __CHILD_OBJECT{
    INHERIT_FROM_COMMON_OBJECT
    .. …
};
```

这样，就实现了对象__CHILD_OBJECT 从对象__COMMON_OBJECT 继承的目标。

显然，这样做的一个不利之处是，对象尺寸会增大（每个对象的定义都包含了指向成员函数的指针），但相对给开发造成的便利，以及代码的可移植性而言，是非常值得的。

2.1.3 使用强制类型转换实现动态类型

面向对象语言的一个重要特性就是，子类类型的对象，可以适应父类类型的所有情况。为实现这个特点，我们充分利用了 C 语言的强制类型转换机制。比如，__CHILD_OBJECT 对象从__COMMON_OBJECT 对象继

承，那么理论上说，__CHILD_OBJECT 可以作为任何参数类型是__COMMON_OBJECT 的函数的参数。比如，下列函数：

```
DWORD GetObjectName(__COMMON_OBJECT* lpThis);
```

那么，以__CHILD_OBJECT 对象作为参数是可以的：

```
__CHILD_OBJECT  Child;
GetObjectName((__COMMON_OBJECT*)&Child);
```

可以看出，上述代码使用了强制的类型转换。

在 Hello China 的开发中，我们使用强制类型转换，实现了对象的多态机制。

2.2 对象模型

在面向对象的语言，比如 C++中，实现了一系列对象机制，比如，对象的构造函数和析构函数等，另外，一些基于面向对象的编程框架（比如，OWL 和 MFC 等），对应用程序创建的每个对象都做了记录和跟踪，这样可以实现对象的合理化管理。

但在 C 语言中，缺省情况下却没有这种机制，为了实现这种面向对象的机制，在 Hello China 的开发过程中，根据实际需要，建立了一个对象框架，来统一管理系统中创建的对象，并提供一种机制，对对象创建时的初始化以及销毁时的资源释放，做出支持。

在 Hello China 开发过程中实现的对象机制，主要思路如下：

1、 每个复杂的对象（简单的对象，比如临时使用的简单类型等，不包含在内），在声明的时候，都声明两个函数：Initialize 和 Uninitialize，其中第一个函数对对象进行初始化，第二个函数对对象的资源进行释放，然后定义一个全局数组，数组内包含了所有对象的初始化函数和反初始化函数；

2、 定义一个全局对象，对系统中所有对象进行管理，这个对象的名字是 ObjectManager（对象管理器），该对象提供 CreateObject、DestroyObject 等接口，代码通过调用 CreateObject 函数创建对象，当对象需要销毁时，调用 DestroyObject 函数。

第一点很容易实现，只要在声明的时候，额外声明两个函数即可（这两个函数的参数是__COMMON_OBJECT*），声明完成之后，把这两个函数添加到全局数组中（该数组包含了系统定义的所有对象相关信息，比如对象的大小、对象的类型、对象的 Initialize 和 Uninitialize 函数等）。

对象管理器 ObjectManager 则维护了一个全局列表，每创建一个对象，ObjectManager 都把新创建的对象插入列表中（实际上是一个以对象类型作为 Key 的 Hash 表）。每创建一个对象的时候，ObjectManager 都申请一块内存（调用 KMemAlloc 函数），并根据对象类型，找到该对象对应的 Initialize 函数（通过搜索对象信息数组），然后调用这个函数，于是对象就被初始化。

对于对象的销毁，ObjectManager 则调用对象的 Uninitialize 函数，这样就实现了对象的自动初始化和对象资源的自动释放。

在 Hello China 的开发过程中，一直遵循这种对象模型，实际上，对象模型不局限于对象的自动初始化和自动销毁，而且还适合于对象枚举、对象统计等具体功能，比如，为了列举出系统中所有的核心线程，可以调用 ObjectManager 的特定函数，该函数就会列举出系统中的所有核心线程对象，因为 ObjectManager 维护了自己创建的所有对象的列表，而核心线程对象就是使用 ObjectManager 创建的。

2.3 系统中的对象

在当前版本的 Hello China 实现中，系统中的对象分为全局对象和局部对象两类，所谓全局对象，是全局只有一个实例的对象，比如内存管理器对象（MemoryManager），核心线程管理器对象（KernelThreadManager），对象管理器（ObjectManager）等。

局部对象是指系统运行中可以创建多个的对象，比如核心线程对象，互斥体对象，事件对象，驱动程序对象，文件对象等。

在 Hello China 的实现中，所有局部对象都从__COMMON_OBJECT 继承，而全局对象则不从任何对象继承。需要说明的是，只有局部对象纳入了 Hello China 的对象管理框架（可以使用 ObjectManager 创建和销毁），而全局对象没有纳入该框架，因为全局对象只有一个，在系统实现的时候，直接声明了该对象，没有必要使用 ObjectManager 进行动态创建。

下面我们对当前版本的 Hello China 实现中，采用的全局对象和局部对象做一个简单描述。

2.3.1 全局对象

在 Hello China 当前版本的实现中，存在下列全局对象：

1、 **ObjectManager**，对象管理器，是对象框架的基础，所有局部对象都使用该对象创建和销毁，该对象还实现了对象枚举、对象搜索和查询等功能；
2、 **KernelThreadManager**，核心线程管理器，该对象实现了所有核心线程相关的管理功能，比如核心线程的创建，调度，挂起/恢复（Suspend/Resume）操作等；
3、 **MemoryManager**，内存管理器，该对象实现了所有用户内存的管理功能，并提供了一致的接口，供用户核心线程调用（来分配内存）；
4、 **PageFrameManager**，页框管理对象，负责内存页框的管理，并提供了接口，供其它对象调用。在当前的实现中，该对象仅仅为 MemoryManager 提供服务；
5、 **DeviceInputManager**，管理所有设备输入的对象，该对象把硬件的输入，通知到合适的核心线程；
6、 **System**，系统对象，管理所有系统功能，比如中断管理、定时器管理等，该对象还负责异常管理；
7、 **Loader**，该对象负责所有可执行模块的加载与释放工作；
8、 **DeviceManager**，设备管理器，管理所有的设备驱动程序，以及驱动程序之间的通信，驱动程序和系统之间的通信等。

充分理解上述全局对象的功能，是深入理解 Hello China 体系结构的基础。

2.3.2 局部对象

在 Hello China 当前的实现中，实现了以下局部对象：

1、 **CommonObject**，抽象的公共对象，所有局部对象都从该对象继承，而且该对象是实现对象框架的基础；
2、 **KernelThreadObject**，核心线程对象，所有核心对象相关的数据结构以及操作函数，都集中在该对象中，KenelThreadManager 直接管理该对象；
3、 **CommonSynchronizationObject**，公共的可同步对象，是所有互斥对象的基类，所有互斥对象都从该对象继承；
4、 **Mutex**，互斥体对象；
5、 **Event**，事件对象；
6、 **InterruptObject**，中断对象；
7、 **PriorityQueue**，优先队列对象，系统中所有优先队列的实现，都是基于该对象；
8、 **TimerObject**，事件对象，实现了定时功能；
9、 **ModuleObject**，可加载模块对象；
10、 **DriverObject**，驱动程序对象；

11、 **FileObject**，文件对象，每个打开的文件，对应一个该对象。

3 Hello China 的任务模型

在这一章中，我们对 Hello China 的任务模型进行详细的叙述。

3.1 任务模型概述

一般情况下，在描述操作系统的任务管理机制时，存在三个说法：

- **进程**，所谓进程，是一个动态的概念，一个可执行模块（可执行文件），被操作系统加载到内存，分配资源，并加入就绪队列后，就形成了一个进程，一般情况下，进程有独立的内存空间（比如，在典型的 PC 机操作系统中，一个进程有独立的 4G 虚拟内存空间，如果目标 CPU 是 32 位），如果不通过 IPC 机制，进程之间是无法交互任何信息的，因为进程之间的地址空间是独立的，不存在重叠的部分；
- **线程**，一般情况下，线程是最小的执行单元（CPU 可感知的），一个进程往往包含多个线程，这些线程共享进程的内存空间，线程之间可以直接通过内存访问的方式进行通信，线程之间共享同进程的全局变量，但每个线程都有自己的堆栈和硬件寄存器；
- **任务**，概念同线程类似，但与线程不同的是，任务往往是针对没有进程概念的操作系统来说的，比如，一些实时的操作系统，这些操作系统没有进程的概念，或者说整个操作系统就是一个进程，这种情况下，任务便成了操作系统中最直接的执行单位。

在 Hello China 的实现中，目前情况下是没有进程概念的，但却存在多个执行线索，因此，用任务来描述这些执行线索是最合适的。但考虑到将来的版本，需要实现进程，因此，为了便于跟将来兼容，我们也以“线程”来称呼系统中的执行线索，为了区别将来的用户线程（用户进程中的线程），我们把当前版本下，Hello China 的任务叫做“核心线程（Kernel Thread）”。

在本文的后续部分，我们称系统中的执行线索为“线程”或“核心线程”。

3.2 核心线程对象

在操作系统中，为了对进程或线程进行管理，往往在操作系统核心数据结构中，为每个线程或进程保留一个控制结构（一般成为 PCB/TCB，进程/线程控制块），在这个结构中保存线程或进程的管理数据以及执行环境等，比如，线程或进程的硬件寄存器、页目录和页表、打开的文件句柄等，都保存在控制块中。

在 Hello China 的实现中，也遵循这种思路，为每个核心线程维护一个控制块，叫做核心线程对象。之所以叫做核心线程对象，是因为在 Hello China 的实现中，对于所有的实体，包括各类控制结构、管理结构等，都是按照对象来实现的。

每个核心线程对象包含下列线程相关的数据：

- 线程执行上下文，比如，CPU 的硬件寄存器等；
- 线程状态，记录了当前线程的状态；
- 线程 ID，唯一的区分了系统中的一个线程；
- 等待队列，等待该线程结束的线程都排到这个队列中；
- 线程优先级；
- 线程的当前目录、根目录和模块目录；
- 线程的入口地址和参数；

- 线程消息队列；
- 线程打开的核心对象（比如，互斥体等）句柄表格等。

下面是内核线程对象的定义：

```
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_OBJECT)
    INHERIT_FROM_COMMON_OBJECT
    INHERIT_FROM_COMMON_SYNCHRONIZATION_OBJECT
    __KERNEL_THREAD_CONTEXT         KernelThreadContext;
    DWORD                           dwThreadID;
    DWORD                           dwThreadStatus;
    __PRIORITY_QUEUE*                 lpWaitingQueue;
    DWORD                           dwThreadPriority;
    DWORD                           dwScheduleCounter;
    DWORD                           dwReturnValue;

    DWORD                           dwTotalRunTime;
    DWORD                           dwTotalMemSize;

    __KERNEL_FILE*                    lpCurrentDirectory;
    __KERNEL_FILE*                    lpRootDirectory;
    __KERNEL_FILE*                    lpModuleDirectory;

    BOOL                             bUsedMath;

    DWORD                           dwStackSize;
    LPVOID                            lpInitStackPointer;

    DWORD                            (*KernelThreadRoutine)(LPVOID);
    LPVOID                           lpRoutineParam;

    __KERNEL_THREAD_MESSAGE            KernelThreadMsg[MAX_KTHREAD_MSG_NUM];
    UCHAR                           ucMsgQueueHeader;
    UCHAR                           ucMsgQueueTrial;
    UCHAR                           ucCurrentMsgNum;
    UCHAR                           ucAligment;

    DWORD                           dwLastError;

    __KERNEL_THREAD_OBJECT_TABLE        KernelObjectTable;
    __KERNEL_THREAD_OBJECT*           lpParentKernelThread;

END_DEFINE_OBJECT()
```

可以看出，该对象的定义比较复杂，而且在将来的版本中，为了扩展更多的功能，可能还需要对这个结构进行扩充。

从代码中还可以看出，该对象是从__COMMON_OBJECT 和__COMMON_SYNCHRONIZATION_OBJECT 继承的，因此，该对象可以使用对象管理器（ObjectManager）进行创建，而且该对象还是一个同步对象，其它线程可以等待该对象，直到该对象对应的线程运行结束。

3.3 线程状态及转换机制

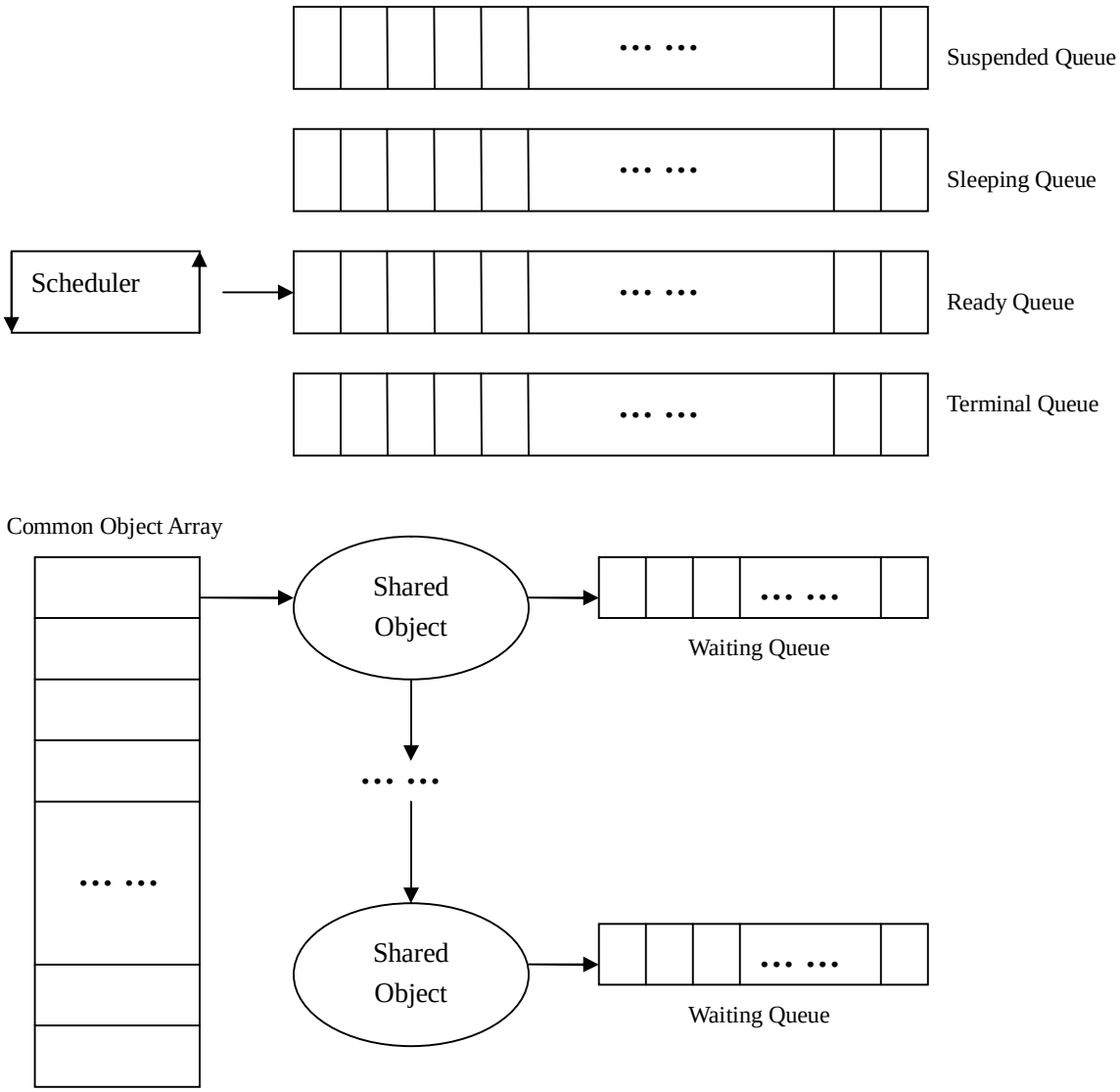
Hello China 的线程可以处于以下几种状态：

- **Ready**：所有线程运行的条件就绪，线程进入 Ready 队列，如果 Ready 队列中没有比该线程优先级更高的线程，那么下一次调度程序运行时（时钟中断或系统调用），该线程将会被选择投入运行；
- **Suspended**：线程被挂起，这是线程执行 SuspendKernelThread 的结果，或者该线程创建时，就指定初始状态为 Suspended，处于这种状态的线程，只有另外的线程调用 ResumeThread，才能把该线程的状

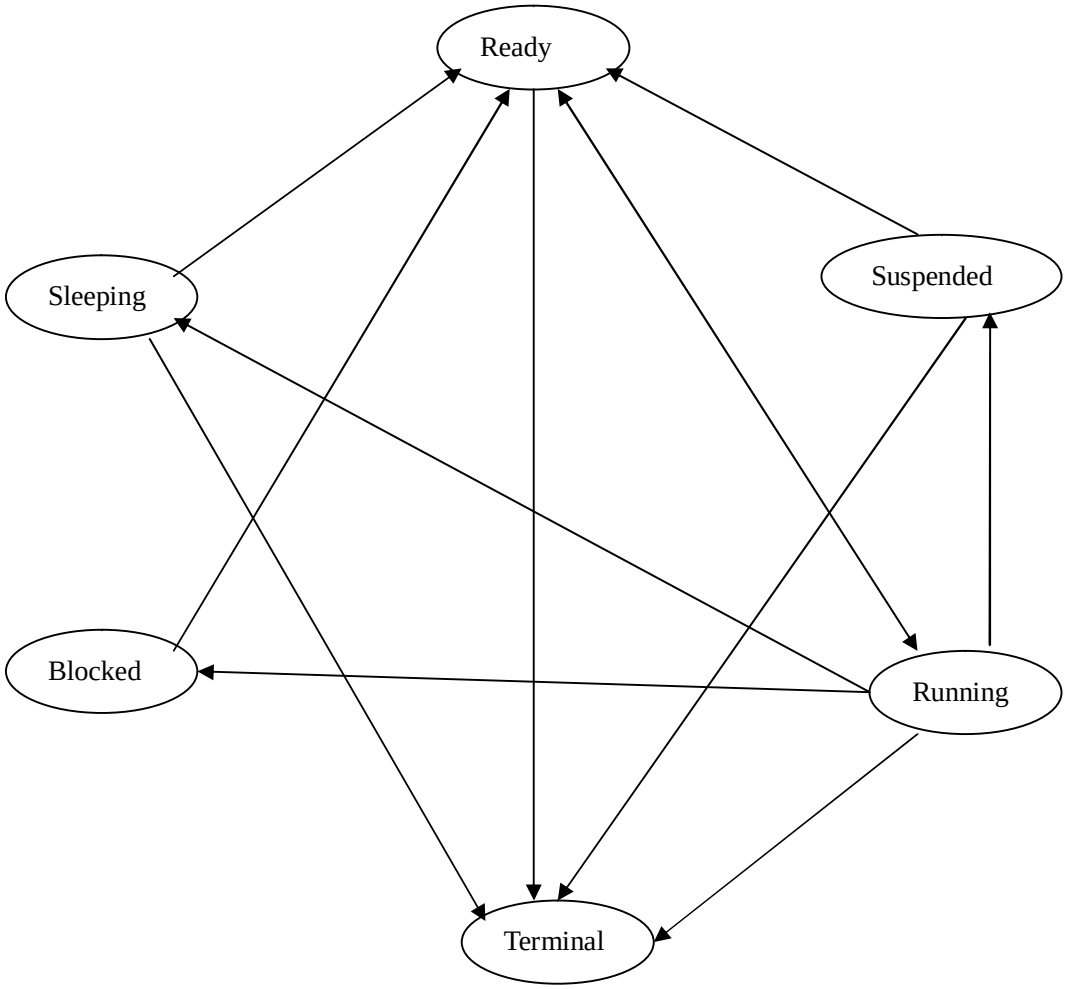
态改变为 Ready；

- **Running**：线程获取了 CPU 资源，正在运行。在单 CPU 环境下，任何时刻只有一个线程的状态是 Running，但在多 CPU 环境下，假设有 N 个 CPU，那么任何时刻，最多的时候，可能有 N 个线程的状态是 Running；
- **Sleeping**：线程处于睡眠状态，一般情况下，处于运行状态（Running）的线程调用 Sleep 函数，则该线程进入睡眠队列，当定时器（由 Sleep 函数指定）到时后，处于该状态的线程被系统从 Sleeping 队列中删除，并插入 Ready 队列，相应地，其状态修改为 Ready；
- **Blocked**：线程不具备运行的条件，比如，线程正在等待某个共享资源，那么该线程就会进入该共享资源的队列中，其状态也会被修改为 Blocked。当共享资源被其它资源释放的时候，会重新修改当前等待该共享资源的线程（Blocked 线程），把其状态修改为 Ready，并放入 Ready 队列；
- **Terminal**：线程运行结束，但用于对该线程进行控制的线程对象（Kernel Thread Object）还没有被删除，处于这种状态的线程对象，被放入 Terminal 队列，直到另外的线程明确的调用 DestroyKernelThread，删除该线程对象为止。

其中，每种状态的线程对象被组织在一个队列中（除了状态是 Running 的线程），每个队列都是一个优先队列对象，因此，位于其中的线程对象可以按照优先级进行排序，对于状态是 Blocked 的线程，被组织在共享资源（或同步对象）的本地等待队列中，调度程序只选择 Ready 队列中的线程投入运行，整个系统的线程对象队列模型，参考下图：



下图给出了线程各个状态之间的转换图示，其中，箭头表示转换方向，单向的箭头，代表状态转换是单向的，比如 Running 到 Suspended 的单向箭头，代表一个处于 Running 状态的线程，可以切换到 Suspended 状态，但反之则不行。



从图中可以看出，Running 状态只能从 Ready 状态转换过来，Blocked 状态和 Suspended 状态，也只能从 Running 状态转换过来。

下面的表格列出了各状态线程之间的切换条件：

初始状态 / 目标状态	Ready	Suspended	Running	Blocked	Sleeping	Terminal
Ready	——	ResumeKernelThread	时间片用完	获得共享资源	定时器到时	——
Suspended	——	——	SuspendKernelThread 自己调用	——	——	——
Running	获得 CPU 资源	——	——	——	——	——
Blocked	——	——	请求共享资源	——	——	——
Sleeping	——	——	调用 Sleep 函数		——	——
Terminal	TerminalKernelThread 其它线程调用	TerminalKernelThread 其它线程调用	运行完	——	TerminalKernelThread 其它线程调用	——

表格中，横的一栏为初始状态，对应的表格为转换原因，竖的一栏为目标状态。

3.4 线程切换机制

线程切换发生在下列两种情况下：

3.4.1 时钟中断

时钟中断发生后，中断处理程序会调用线程调度程序，线程调度程序重新计算当前线程（中断发生时正在运行的线程）的优先级，并跟 Ready 队列中的第一个线程对象（因为 Ready 队列是优先队列，按照线程优先级进行排序，优先级高的线程对象，排在队头）进行比较，如果当前线程的优先级比第一个线程对象的优先级高，则不发生线程切换，继续运行当前线程。
如果当前线程的优先级比 Ready 队列中的第一个线程的优先级低，则调度程序会保存当前线程的执行环境（寄存器、堆栈指针等），然后把当前线程对象插入到 Ready 队列中（以优先级为关键字进行插入），然后选择 Ready 队列中第一个线程对象，恢复第一个线程对象的执行环境，并切换到该线程。

3.4.2 请求共享资源

另外一个线程切换的机会是，当前线程请求共享资源或等待互斥对象。比如，当前线程访问一个多线程共享的资源，那么为了避免发生冲突，一般使用一个互斥体对象（Mutex）或事件对象（Event）来对共享资源进行保护，当前线程在请求共享资源的时候，比如首先等待互斥对象的状态为空闲（调用 WaitForThisObject），如果当前互斥对象不空闲（即被其它线程所拥有），那么该线程在调用 WaitForThisObject 的时候，该函数就会保存当前线程的执行环境，然后设置当前线程状态为 Blocked，并把当前线程对象插入互斥体对象的等待队列中，最后，WaitForThisObject 从 Ready 队列中选择一个就绪线程（队列头线程），恢复就绪线程的执行环境，切换到新的线程进行执行。

3.4.3 线程切换方式

一般情况下，CPU 都提供了相应的任务切换机制，比如，在 Intel 的 CPU 中，就可以通过全局描述表（GDT）的形式进行切换，比如，一个任务（线程、进程等）占用一个 GDT 表象，在切换任务的时候，直接使用 CALL 或 JMP 等跳转指令就可以完成切换，这种方式即所谓的硬件切换方式。
另外一种方式是软件切换方式，在软件切换方式中，对硬件的任务切换机制依赖非常小，只要目标 CPU 支持保存 PC 寄存器（一般情况下，保存到到堆栈中），就可以使用软件的方式完成任务切换。一般情况下，软件切换的过程如下：

1、 中断发生（或系统调用发生），正在执行的线程的下一条指令地址被压入堆栈，然后跳转到中断处理程序（或目标函数）继续运行；
2、 中断处理程序（或目标函数）保存当前线程 PC 寄存器的值（下一条指令地址），一般情况下是保存到线程控制块（在 Hello China 中，是内核线程对象）中，然后再保存当前线程的所有硬件寄存器（执行环境）；
3、 当前线程的执行环境保存完后，调度程序选择一个就绪状态的线程，恢复其执行环境（比如，切换堆栈，把程序计数器 PC 的值恢复到堆栈中等），然后执行相应的返回指令（比如，IRET 或 RET），这样当前执行线程就发生了切换。

可以看出，如果采用硬件任务切换机制，那么对目标 CPU 的依赖将非常大，代码的移植性不强，而使用

软件切换就不存在这个缺陷。

在 Hello China 的实现中，使用软件方式进行切换，这样就使得 Hello China 的线程切换机制可以很容易的移植到不同的目标 CPU 上，充分保证了 Hello China 的可移植性。

3.5 线程调度算法

Hello China 当前版本的实现中，线程的调度算法，是基于抢占式的优先级调度算法。在核心线程对象中，有两个变量：

```
DWORD                           dwThreadPriority;
DWORD                           dwScheduleCounter;
```

这两个变量是实现线程调度的基础。

在当前版本的实现中，只实现了一种算法，即优先级相关的时间片调度算法，该算法执行流程如下：

1、 线程创建时，dwScheduleCounter 初始化为 dwThreadPriority，即线程优先级的值；

2、 线程每运行一个时间片（系统时钟中断间隔），调度程序把 dwScheduleCounter 的值减一，并作为新的优先级；

3、 如果 dwScheduleCounter 减一后，结果是 0，那么重新初始化为 dwThreadPriority 的值。

根据这个算法可以看出，优先级越高的线程，获得的 CPU 资源越多，而且优先级高的线程，可以完全抢占优先级低的线程而投入运行。但这个算法也避免了优先级低的线程出现所谓“饿死”现象，即 CPU 时钟被优先级高的线程占有，优先级低的线程无法占有 CPU 而得不到运行。因为随着运行时间的增加，优先级高的线程，其 dwScheduleCounter 逐渐降低，在降低到一定程度时，优先级低的线程就有机会运行了。

比如，按照这个算法，系统中存在三个线程：

线程 A，优先级为 6；

线程 B，优先级为 4；

线程 C，优先级为 2。

假设 A 先运行，按照这个算法，系统将发生以下进程切换动作：

A) 第一个时钟中断：调度程序减少 A 的 dwScheduleCounter（初始为 6，结果为 5），然后跟 Ready 队列的第一个线程（B）比较，发现 A 比 B 的优先级（dwScheduleCounter）大，于是继续运行 A；

B) 第二个时钟中断：调度程序减少 A 的 dwScheduleCounter，然后跟 Ready 队列的第一个线程（B）比较，发现仍然不小于（只有小于才发生切换），于是继续运行 A；

C) 第三个时钟中断：调度程序减少 A 的 dwScheduleCounter，跟 B 比较，发现小于 B，于是 A 入 Ready 队列（以 dwScheduleCounter 为关键字，此时为 3），然后恢复 B 的上下文，B 开始运行；

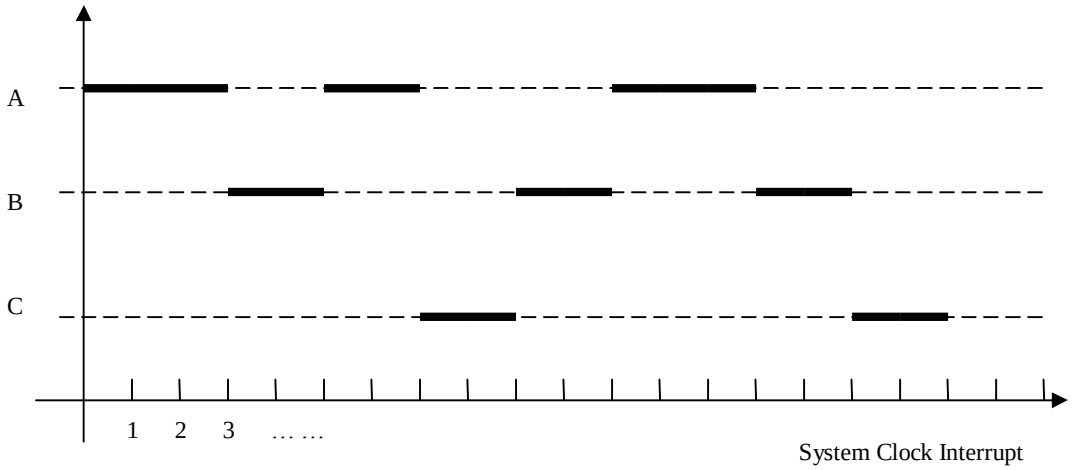
D) 第四个时钟中断：调度程序减少 B 的 dwScheduleCounter（结果为 3），然后跟 Ready 队列中的第一个线程（A）比较，发现不小于，于是继续运行；

E) 第五个时钟中断：调度程序减少 B 的 dwScheduleCounter（结果为 2），然后跟 A 比较，发现小于，于是 B 入 Ready 队列，恢复 A 上下文，A 继续运行；

F) 第六个时钟中断：调度程序减少 A 的 dwScheduleCounter（结果为 2），然后跟 Ready 队列的第一个线程（C）比较，发现不小于，于是继续运行；

G) 第七个时钟中断：调度程序减少 A 的 dwScheduleCounter（结果为 1），然后跟 C 比较，发现小于 C，于是 C 运行。

这三个线程运行的序列如下图所示：



通过上面的描述可以看出，该算法是严格抢先的，比如，又有一个核心线程 D 被创建，D 的优先级为 8，那么，在下一个调度时刻（时钟中断），D 将会被优先选择执行，因为 D 的优先级在当前就绪线程队列中，是最高的。这种严格抢先的调度方式，对实时性系统是非常重要的。

3.6 线程同步机制

有的情况下，线程之间的运行是相互受影响的，比如，对共享资源的访问，就需要访问共享资源的线程相互同步，以勉破坏共享资源的连续性。在当前版本的 Hello China 的实现中，实现了下列线程同步机制：

3.6.1 事件（Event）

事件对象是一个最基础的同步对象，事件对象一般处于两种状态：空闲状态和占用状态。一个内核线程可以等待一个事件对象（通过调用 WaitForThisObject），如果一个事件对象处于空闲状态，那么任何等待该事件对象的线程都不会阻塞，相反，如果一个事件对象处于占用状态，那么任何等待该对象的线程，都进入阻塞状态（Blocked）。

一旦事件对象的状态由占用变为空闲，那么所有等待该事件对象的线程都被激活（状态由 Blocked 改变为 Ready，并被插入 Ready 队列），这一点与下面讲述的互斥体不同。

3.6.2 信号量（Semaphore）

信号两也是最基础的同步对象之一，一般情况下，信号两维护一个计数器，假设为 N，那么每当一个内核线程调用 WaitForThisObject 等待该信号量对象时，那么 N 就减一，如果 N 小于零，那么等待的线程将被阻塞，否则继续执行。

3.6.3 互斥体（Mutex）

互斥体是一个二元信号量，即 N 的值为 1，这样最多只有一个内核线程占有该互斥体对象，当这个占有该互斥体对象的线程释放该对象时，只能唤醒另外一个内核线程，而其它的内核线程将继续等待。

注意互斥体对象与 Event 对象的不同，Event 对象中，当一个占有 Event 对象的线程释放该对象时，所有等待该 Event 对象的线程都将被激活，而 Mutex 对象，则只有一个内核线程被激活。

3.6.4 内核线程对象（KernelThreadObject）

内核线程对象本身也是一个互斥对象，即其它内核线程可以等待该对象，从而实现线程执行的同步。但与普通的互斥对象不同的是，内核线程对象只有当状态是 TERMINAL 时，才是空闲状态，即任何一个线程，如果等待一个状态是非 Terminal 的内核线程对象，那么将会一直阻塞，直到等待的线程运行结束（状态修改为 Terminal）。

3.6.5 睡眠

一个运行的线程，可以调用 Sleep 函数而进入睡眠状态（Sleeping），进入睡眠状态的线程，将被加入 Sleeping 队列。当睡眠时间（由 Sleep 函数指定）到达时，系统将唤醒该睡眠线程（修改状态为 Ready，并插入 Ready 队列）。

3.6.6 定时器

另外一个内核线程同步对象是定时器。定时器是操作系统提供的最基础服务之一，在 Hello China 中，线程可以调用 SetTimer 设置一个定时器，当设置的定时器到时，系统会给设置定时器的线程发送一个消息。
于 Sleep 不同的是，内核线程调用 Sleep 后，将进入阻塞状态，而调用 SetTimer 之后，线程将继续运行。
在当前版本的实现中，一个核心线程可以设置多个定时器，在定时器没有到时时，还可以通过系统调用 CancelTimer 取消已经设置的定时器。

3.7 IPC 机制

在当前的实现中，Hello China 提供了两种 IPC 机制：

3.7.1 消息队列机制

一个内核线程拥有一个消息队列，线程之间可以相互发送消息，相互发送的消息将会被系统保存在线程的消息队列中，当线程运行的时候，会检查消息队列中的消息，并对消息进行处理。

实际上，消息队列是 Hello China 核心线程实现的基础。一个线程可以调用 SendMessage 系统调用，给目标线程发送一个消息，而不管目标线程的状态如何（即使是 Terminal，也可以接收消息）。

一个运行的线程，可以调用 GetMessage 从自己的消息队列中获取消息，在当前版本的实现中，一个核心线程的消息队列长度为 32。

使用消息进行 IPC 通信时，消息类型和消息数据必须约定好，否则目标线程无法识别消息。

3.7.2 共享变量机制

由于目前版本的实现中，核心线程是基于共享内存实现的，因此可以通过全局变量来进行核心线程之间的通信。

4 Hello China 的内存管理模型

本章中，我们详细讲述当前版本 Hello China 的内存管理模型，包括核心内存管理模型、用户内存管理模型等。

在介绍前，首先介绍几个用到的概念：

1、 页框（Page Frame），一般情况下，为了便于管理物理内存，把物理内存划分成尺寸相同的块，每个块就称为一个页框。通常页框的尺寸是 2 的整数次方，一般为 4K，或 8K 字节大小，在 Hello China 当前的实现中，把页框的大小设置为 4K（PAGE_FRAME_SIZE），后续版本中，如果需要更改页框大小，只需要更改宏 PAGE_FRAME_SIZE 的定义即可；

2、 页块，一个或数个连续的页框，组成一个页块，一般情况下，2 的整数次方个页框组成一个页块，在进行内存分配的时候，一般是以页块为单位进行的，在 Hello China 当前实现中，实现了从 4K 到 8M 大小的页块；

3、 Chunk，一个特定用途的页块，提出 Chunk 的概念，是为了管理上的方便。通常情况下，核心线程请求了一块内存，其用途是不同的，比如，有的页块是用来装载可执行代码的，有的页块是堆栈，而有的则是动态内存分配堆，在这种情况下，仅仅凭页块的概念，无法区分这些不同的应用，因此提出了 Chunk 的概念，在当前的实现中，存在代码 Chunk（CODE Chunk）、堆栈 Chunk（Stack Chunk）、堆 Chunk（Heap Chunk）等。

在 Hello China 的实现中，按照内存使用对象的不同划分为两类内存：

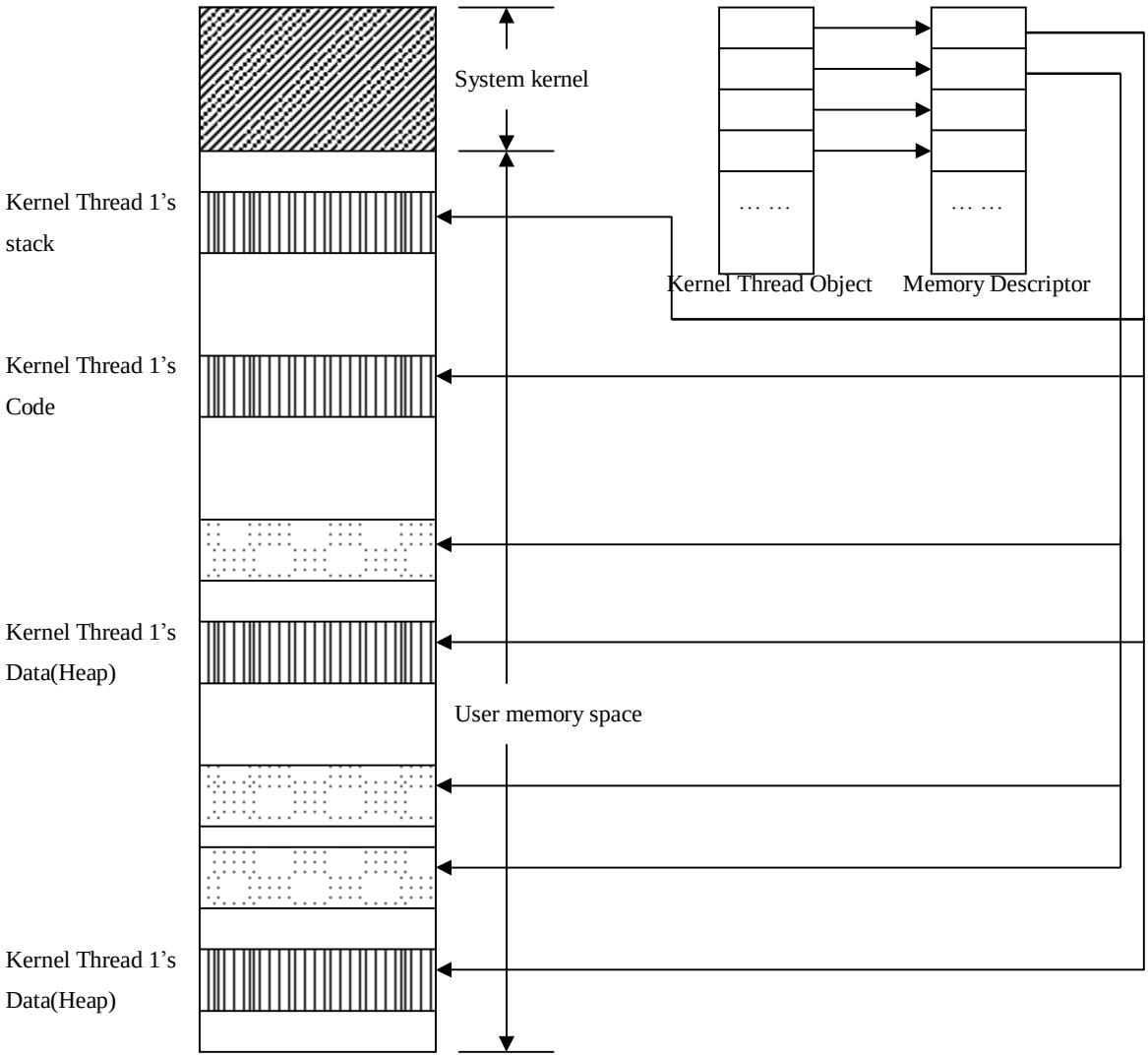
1、 核心内存，由内核使用的内存，比如核心线程对象（KernelThreadObject）占用的内存、各种管理对象（MemoryManager、System、PageFrameManager 等）占用的内存等，驱动程序占用的内存也属于核心内存，对于核心内存，使用空闲链表算法进行管理；

2、 用户内存，供核心线程使用的内存，一般情况下，核心线程通过 MemoryAlloc（内存管理对象提供的接口）请求的内存，都属于用户内存。

下面我们来详细讨论一下 Hello China 的内存管理模型。

4.1 内存模型

下图清晰的展示了 Hello China 的内存布局：



从内存地址 0x00000000 到内存物理地址 0x00FFFFFF 范围，是用于操作系统核心的，属于核心内存（共16M 大小）；从内存地址 0x01000000 到物理内存的末端，是用户内存。

操作系统正常运行需要的数据结构（表格、数组、对象结构等），以及操作系统运行过程中动态申请的内存，都属于核心内存，在当前版本的实现中，核心内存使用空闲链表进行管理，核心例程可以使用 KMemAlloc 函数调用来获得核心内存，使用 KMemFree 例程来释放核心内存。

对用户内存，是按核心对象进行组织的，内存管理对象（MemoryManager）提供了用户内存管理的统一接口。

对用户内存管理的时候，分两个等级，第一个等级是页框层面的管理，由页框管理器（PageFrameManager）实现，页框管理器使用伙伴算法（Buddy Algorithm）来实现页框的分配和释放，给上层内存管理对象提供一个统一的物理内存分配接口。

第二个等级是内存管理器，内存管理器以 Chunk 的方式组织核心线程内存。在内存管理对象（MemoryManager）中，维护了一个内存描述符数组（该数组共有 MAX_KERNEL_THREAD_NUM 个元素），每个数组元素是一个内存描述符结构，每个核心线程对应一个内存描述符，每个线程保留的 Chunk 都记录在内存描述符中，在核心线程对象中，维持了一个内存描述符的索引，通过该索引，可以直接找到该核心线程在内存描述符数组中的内存描述符的索引。

一个核心线程可能分配许多 Chunk，每创建一个 Chunk，内存描述符就把该 Chunk 记录在该线程的 Chunk

链表（或 AVL 树）中，这样当核心线程结束的时候，操作系统核心可以把该线程对应的内存资源删除，而不会造成内存浪费。

4.2 操作系统核心模块内存分配与回收

从物理内存 0x00000000 到物理内存 0x00FFFFFF 之间的内存，是核心内存，核心内存又进一步分为三部分：

1、 从物理内存 0x00000000 到物理内存 0x000FFFFF 之间的内存（共 1M 大小）为保留内存，供硬件作为缓冲区（比如，显示缓冲区等）使用；
2、 从物理地址 0x00100000 到 0x001FFFFF 之间（共 1M）的内存，是操作系统核心代码和初始化数据占用的内存，操作系统核心模块被加载到这段地址；
3、 从物理地址 0x00200000 到 0x00FFFFFF 之间的内存（共 14M）是操作系统核心内存，供操作系统核心运行时动态分配之用，比如，每当创建一个线程，操作系统核心都要分配一块内存，作为核心线程对象（KernelThreadObject）使用，另外，对于设备驱动程序，也加载到这段内存之间。

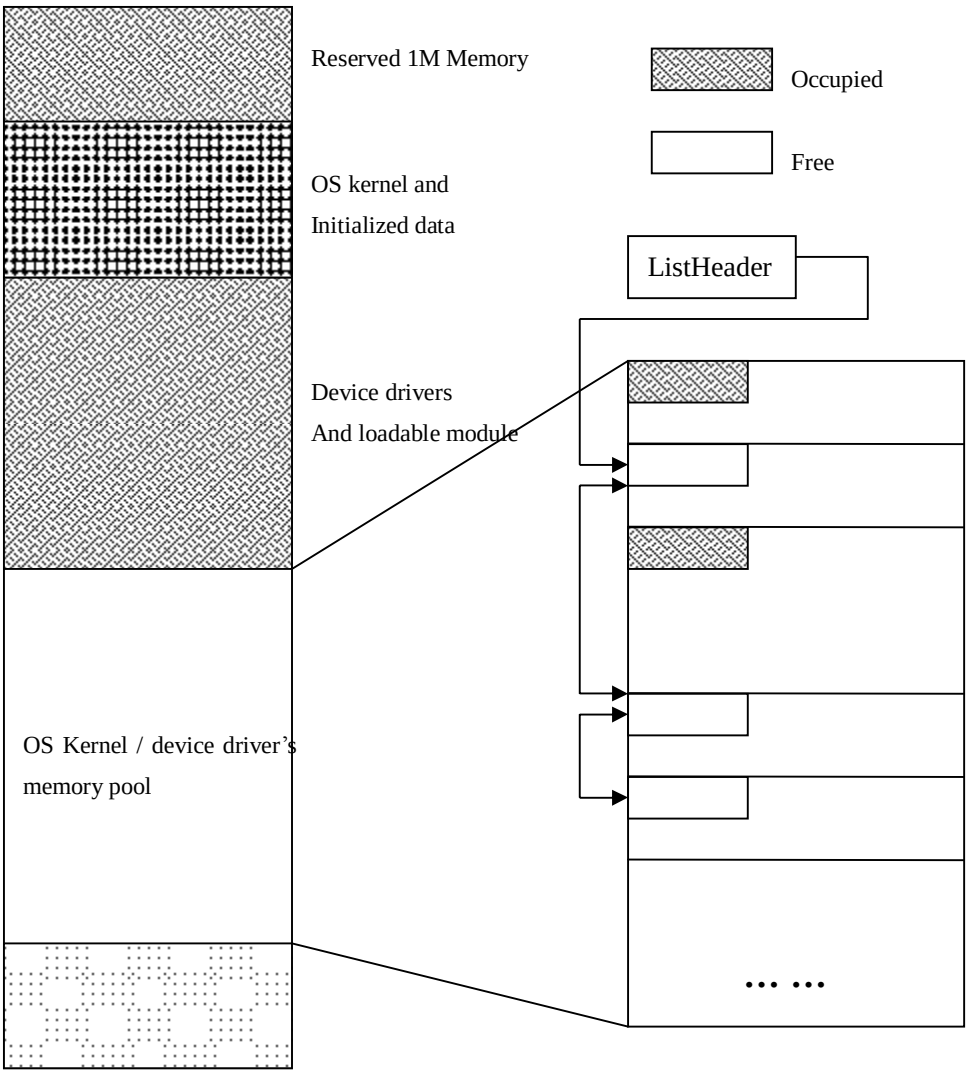
有些情况下，保留 16M 的核心内存可能无法满足实际需要（比如，在加载大量的驱动程序的情况下），这时候可以通过修改代码，来扩展核心内存的大小。

在 Hello China 当前的实现中，采用空闲链表的方式来管理操作系统核心内存。空闲链表算法简单描述如下：

1、 系统维护一个空闲链表，连接所有的空闲内存块。开始的时候，整个核心内存区域作为一个空闲块连接到空闲链表中；
2、 每当有一个内存分配申请到达，内存管理函数遍历空闲链表，寻找一块空闲内存，该内存的大小大于请求的内存（或等于）；
3、 如果不能找到，则返回空指针（NULL）；
4、 如果找到，判断寻找到的内存的大小，如果跟请求的内存大小一致，或比请求的内存大少许（比如，16 字节），那么内存管理函数就把整个内存块返回给用户，然后把该空闲块从内存中删除；
5、 如果找到的内存比用户请求的内存大许多（比如，大于 16 字节），那么内存管理函数把该空闲块分成两块，一块仍然作为空闲块插入空闲链表中，另外一块返回用户。

对于内存回收算法，如下：

1、 回收函数（KMemAlloc）把释放的内存插入空闲链表；
2、 在插入的同时，回收函数判断跟该空闲块相邻的下一块是否可以跟当前块合并（合并成更大的块）；
3、 如果可以合并（地址连续），那么回收函数将合并两块空闲内存块，然后作为一块更大的内存块重新插入空闲链表；
4、 如果不能合并，则简单返回。



为了维护这些空闲块，必须为每块空闲块分配一个控制结构，然后这个控制结构指定了特定的空闲内存块。在分配和回收的时候，需要对空闲块的控制结构进行修改，因此，必须有一种方法，能够快速的定位控制结构。为了解决这个问题，我们把空闲块的控制结构放在空闲块的前端，这样给定一个内存地址，就可以很容易的索引到其控制块，比如，假设给定的内存地址为 lpStartAddr，空闲内存控制结构为__FREE_BLOCK_CONTROL_BLOCK，那么对应该空闲块的控制结构可以这样获取：

```
__FREE_BLOCK_CONTROL_BLOCK*  lpControlBlock =
   (__FREE_BLOCK_CONTROL_BLOCK*)((DWORD)lpStartAddr –
   sizeof(__FREE_BLOCK_CONTROL_BLOCK));
```

这在内存释放（KMemFree）的时候特别有用。

在 Hello China 当前版本的实现中，空闲链表算法使用的是初次适应算法，即把第一次发现的空闲块分配给用户，而不管这个内存块是否太大，这样往往会造成内存碎片，即随着分配次数的增加，内存中零碎的内存片数量逐渐增多，到了一定的程度，整个内存中全部是零碎的内存片，如果用户请求一块大的内存，往往以失败而告终。

但这些缺点仅仅是理论上的，试验表面，首次适应算法能很好的满足实际需求，实际上，很多操作系统的内存分配算法就是使用这种方式实现的，运行效果十分理想。

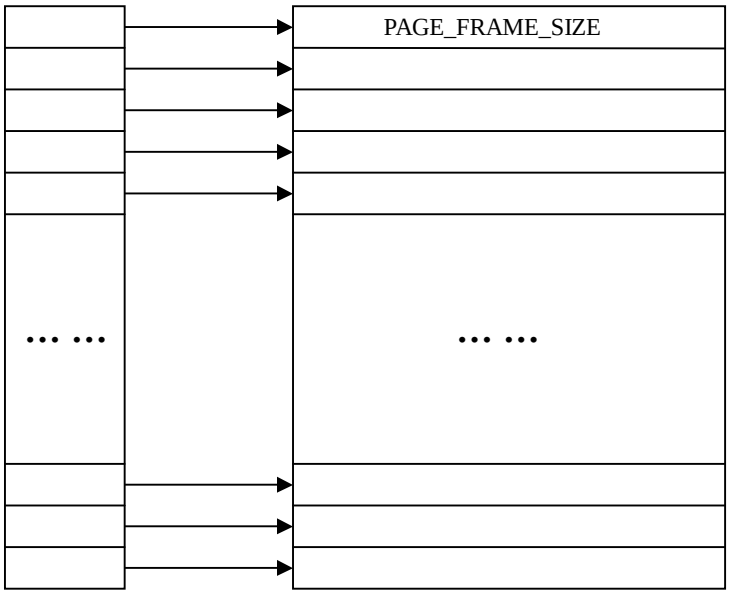
4.3 用户线程内存分配与回收

本节介绍用户内存（从物理地址 0x01000000 到物理内存末端）的管理方法。

4.3.1 页框管理算法

在 Hello China 的当前实现中，把物理内存分成一个一个的页框，每个页框的大小为 PAGE_FRAME_SIZE（当前版本中，该数字定义为 4K），对每一个页框，分配一个页框控制结构，系统中所有页框的控制数据结构组合在一起，形成一个数组。

整体结构请参考下图：



在系统初始化的时候，页框管理数组就被填满了（根据测试到的物理内存数量），并且记录下物理内存的起始地址，这样就建立了页框管理数组和物理内存的一一对应关系，因此，给定一个页框管理结构的索引，就可以唯一的确定一块物理内存（尺寸为 PAGE_FRAME_SIZE），相反，给定任何一个物理地址，就可以确定该物理地址对应的页框管理结构。比如，给定一个页框索引为 N，那么相应的物理内存块初始地址可以这样计算：

lpPageFrameAddr = lpStartAddr + PAGE_FRAME_SIZE*N;

相反，给定一个物理地址，假设为 lpPageFrameAddr，那么对应的页框索引可以这样确定：

dwIndex = (lpPageFrameAddr – lpStartAddr) / PAGE_FRAME_SIZE;

其中，lpStartAddr 为物理内存的起始地址（物理地址）。

然而实际上，对内存的请求往往不是一个页框，而是许多页框组成的块，因此，为了更有效的利用内存，我们采用伙伴算法（buddy system algorithm）来对物理页框进行进一步的管理，伙伴算法的一个核心思想就是，通过尽量的合并小的块，来形成大的块，以避免内存浪费。有关伙伴算法的具体流程，请参考数据结构相关的书籍。

在伙伴算法中，把数量不同的连续的物理页框组合成块，在进行分配的时候，根据请求的大小，选择合适的块分配给请求线程。在当前的实现中，一个线程可以请求下列尺寸大小的内存块：

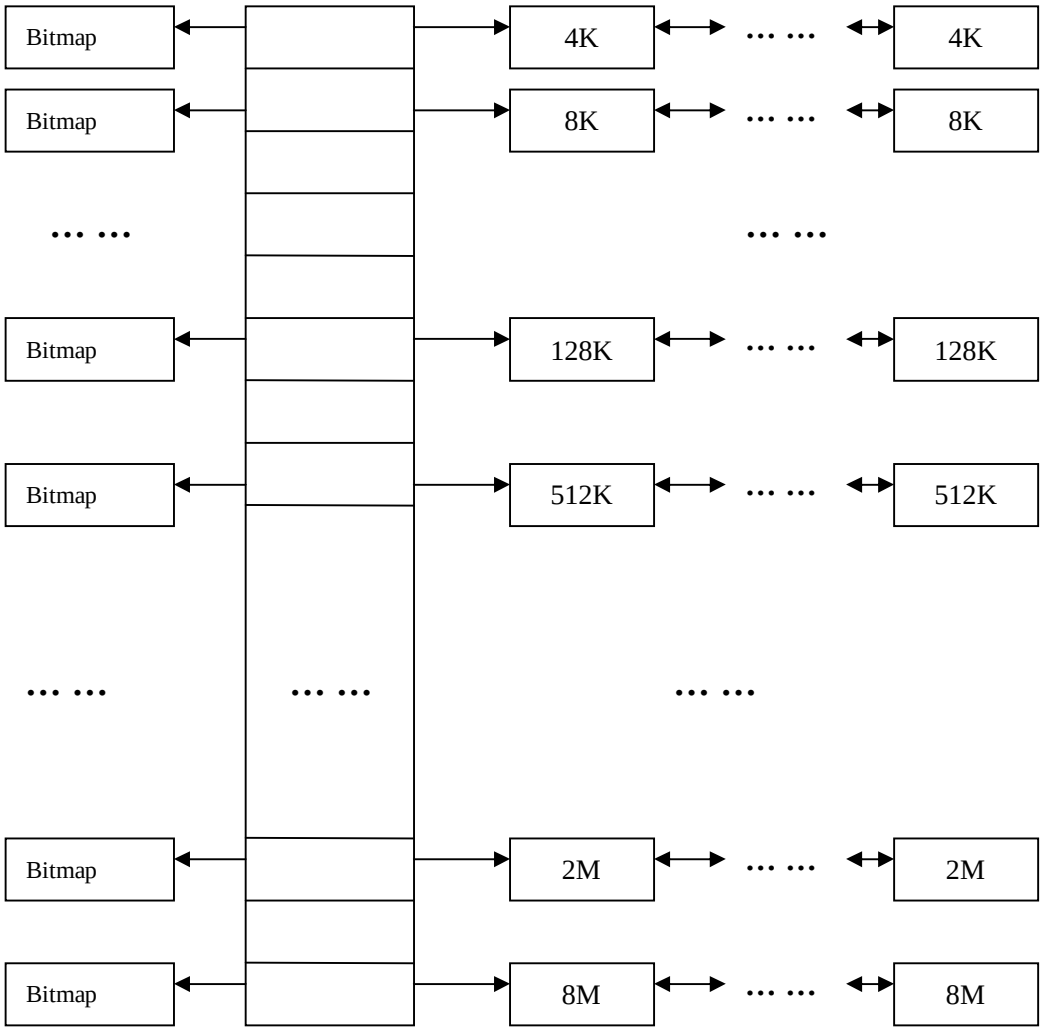
1、 4K；

2、 8K；

3、 16K；
4、 32K；
5、 64K；
6、 128K；
7、 256K；
8、 512K；
9、 1024K；
10、 2048K；
11、 4096K；
12、 8192K。

可以看出，申请的内存块的尺寸，是 4K 的二次方倍数。

在系统核心数据区，维护了下面一个数据结构（参考图）：



Block Control Array

有一个单独的管理对象－页框管理器（PageFrameManager）管理所有的页框以及页块，当一个线程请求一块页块时，页框管理器进行下列动作：

1、 判断申请的块是否超出了最大范围（MAX_CHUNK_SIZE，当前版本定义为 8M），如果是，则返回一个空指针；

2、 如果没有超出，则从第一个页块控制数组开始搜索，判断哪个尺寸的页块合适请求者的要求，比如，请求者请求了 500K 的页块，那么 PageFrameManager 会认为 512K 的页块符合要求；
3、 判断 512K 页块的空闲列表，是否为空，如果不为空，则从中删除一块，返回请求者对应页块的指针；
4、 如果为空，则依次往下（页块尺寸大的方向）搜索，直到找到一块更大的页块，或者失败（搜索到最后一级，仍然没有空闲块）为止；
5、 如果失败，返回用户一个空指针；
6、 否则，依次对分找到的页块，并插入下一级页块空闲列表中，直到 512K 页块为止，然后返回用户经过对分，最后剩下的页块。

每个页块空闲列表控制结构，对应一个位图，系统通过这个位图，来判断对应块的空闲或使用，这样在释放页块的时候，可以根据位图，把连续的页块（伙伴块）组装成更大的页块。

4.3.2 Chunk 与 Chunk 管理

页框管理器（PageFrameManager）提供了一个统一的管理接口，可以直接通过页框管理器申请页框或页块（特等数量的页框组成的集合）。但一个实际的核心线程，可能需要数量不同大的页块，这些页块大小和用途各不相同，比如，一个核心线程，至少需要三个页块：

1、 代码页块，线程执行映象所占用的页块；
2、 堆栈页块，线程堆栈占用的页块；
3、 堆页块，线程堆占用的页块。

有的情况下，还不止三个页块，比如，有的线程可能需要动态分配许多内存，这些内存都从线程堆中分配，因此，一个线程可能有多个堆页块。

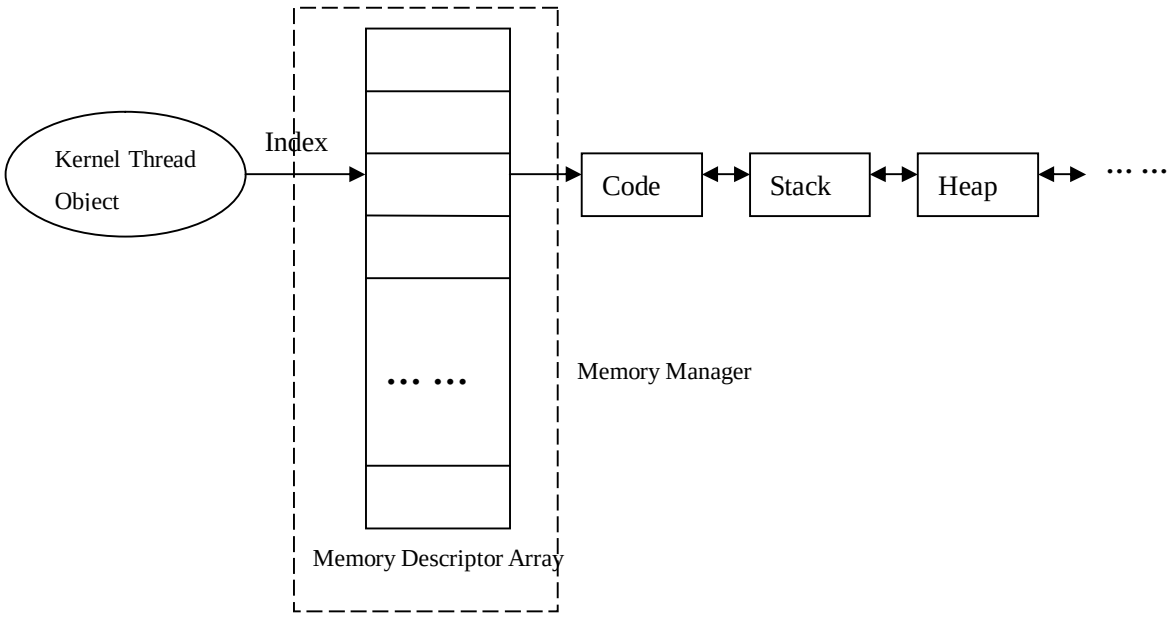
如果仍然按照页块对线程的内存进行管理，将有一些不方便，因此，对于线程的内存，在 Hello China 的实现中，引入 Chunk 的概念。

所谓 Chunk，指的是一个具有特定用途的页块，比如，代码页块，成为代码 Chunk，堆栈页块，成为堆栈 Chunk。

在当前的实现中，专门有一个对象－内存管理器（Memory Manager）实现线程内存的管理，在内存管理器中，维护了一个线程内存描述符数组（当前情况下，该数组的尺寸为 MAX_KERNEL_THREAD_NUM，即操作系统支持的最大的线程个数），每个系统中存在的线程，都有一个数组元素（线程内存描述符）与之对应，而线程内存描述符则具体的描述了该线程的内存占用情况。

比如，针对核心线程的 Chunk，在内存描述符中，有一个双向链表，把所有该线程的 Chunk 连接起来，在 Chunk 的数量非常大（超过了 AVL_SWITCH_NUM，当前定义为 16）的时候，线程内存描述符改用 AVL 树来管理 Chunk，这样可以大大提高查询速度。

在核心线程对象中，保存了一个内存描述符索引，这个索引指定的内存描述符，就是当前线程的内存描述符，关系如下图：



一般情况下，加载程序在加载一个可执行模块的时候，按照这个模型，可以这样进行：

1、 保留一个内存描述符索引（通过调用 ReservDescIndex 函数），假设为 dwDescIndex；
2、 保留一个代码 Chunk（通过调用 ReserveChunk 函数，保留的索引作为参数），然后把可执行映象加载到代码 Chunk 所在的内存；
3、 重定向可执行映象，并获得入口地址；
4、 保留一个堆栈 Chunk；
5、 然后创建核心线程，并把 dwDescIndex 赋值给新创建线程的核心线程对象。

这样由于核心线程对象保存了内存描述符的索引，因此该线程对应的 Chunk，以及该线程分配的所有页框，都是已知的（都可以通过核心线程对象获取），这样在线程结束的时候，系统就可以很方便的删除该线程占用的所有内存资源，而不导致内存浪费。

4.3.3 内存管理器对象

在上面的描述中，对 Chunk 的管理（分配和释放等）都是通过内存管理器来完成的，内存管理器是一个统一的内存管理对象（接口），所有核心线程，都可以通过向内存管理器发送消息（实际上是调用内存管理器提供的函数）来请求服务。当前版本的实现中，内存管理器提供了下列接口：

1、 ReserveDescIndex：保留一个内存描述符索引，一般在新创建一个核心线程的时候，都为新创建的线程保留一个索引，以存储线程所申请的内存资源；
2、 ReleaseDescIndex：释放保留的内存描述符索引；
3、 ReserveChunk：保留一个 Chunk，其大小、类型等信息，通过函数的参数指定；
4、 ReleaseChunk：释放一个 Chunk；
5、 Initialize：初始化函数，初始化内存管理器对象；
6、 MemoryAlloc：从当前线程堆中分配一块内存；
7、 MemoryFree：释放分配的内存；

在 ReserveChunk 函数（接口）的实现中，内存管理器直接调用了页框管理器（PageFrameManager）来分配页框，对于 MemoryAlloc 和 MemoryFree，内存管理器单独进行了实现。

4.3.4 用户堆管理

从上面的描述中可以看出，一个核心线程可以通过 ReserveChunk 的方式，申请保留一块 Chunk，但由于 MemoryManager 直接调用了 PageFrameManager 来分配内存，因此，申请的最小内存数量也是 PAGE_FRAME_SIZE 大小，这样显然太浪费内存了。

因此，为了解决这个问题，我们在 MemoryManager 中直接实现了小块内存申请算法，对外接口就是 MemoryAlloc 和 MemoryFree 函数。

在实现的时候，内存管理器为每个核心线程创建一个 Heap Chunk，并使用与核心内存分配一样的算法（空闲块链表）来管理 Heap Chunk。缺省情况下，每创建一个线程，就为该线程保留一个 HeapChunk，该 Chunk 的大小是 64K。

但如果核心线程申请的内存数量超过了 HeapChunk 的大小，那么内存管理器会继续为该线程创建 Heap Chunk，这时候创建的 Heap Chunk 的大小，是上次创建的 2 倍，这样依次下去，但也不是永无休止的，在当前的实现中，定义了一个常数 MAX_HEAP_CHUNK_NUM（当前版本定义为 8），一旦一个线程因为创建的 Heap Chunk 超过了这个数字，那么内存管理对象就不会再继续创建 Heap Chunk，而是为用户返回一个内存失败标志（NULL）。

采用这种方式，有以下特点：

1、 为用户核心线程设定可申请的内存上限（MAX_HEAP_CHUNK_NUM），这样即使用户核心线程发生了内存泄漏，也不会导致整个系统的内存都消耗掉；
2、 按需分配，开始时分配很小的一部分内存，作为用户的堆，然后随着用户请求的增加，以指数形式递增分配内存量，这样可以很好的满足用户需要，但又不会过渡的浪费内存资源；
3、 对用户堆采用的是空闲链表管理算法，而且每个用户核心线程的空闲链表数据结构相互独立，这样即使一个核心线程由于内存越界等原因，破坏了空闲链表，也不会导致其它线程空闲链表破坏，更不会导致操作系统核心故障，因此，这种方式有很好的可靠性，相反，有的实时操作系统采用了全局统一的空闲链表管理算法，这样一旦有一个任务因为内存越界等原因破坏了空闲链表，那么整个系统都将崩溃。

5 Hello China 的硬件输入模型

5.1 硬件输入模型概述

所谓硬件输入，指的是诸如键盘、鼠标、定时器等硬件产生的输入信号，这些输入信号有的是由用户触发，比如键盘和鼠标的输入，有的是系统硬件自动触发，比如定时器等。

操作系统中一个重要的问题就是如何处理这些硬件输入，一般情况下，硬件输入涉及到两个不同的处理过程：

1、 硬件输入如何通知操作系统？即操作系统如何知道硬件产生了输入；
2、 特定的用户输入事件，比如鼠标点击、键盘输入等，如何最终传输到合适线程？因为系统中可能存在多个线程，而硬件输入可能是针对某一个特定的线程的，比如，一个键盘输入的最终目标是一个编辑器线程，那么这个硬件输入事件如何正确的递交到目标线程，就是操作系统要解决的另一个最基本问题。

CPU 的中断机制是解决第一个问题的办法，正常情况下，系统初始化的时候，给每个硬件设备分配了一个特定的中断号，一旦产生硬件输入，那么相应的硬件就会通过中断的方式，通知 CPU 产生了该中断，这个时

候，CPU 就会停止正在处理的线程，然后把控制转移到一个所谓的“中断处理程序”，该中断处理程序来处理具体的硬件中断，比如，调用对应硬件的驱动程序等。

第二个问题，没有硬件机制可以支撑，因此必须根据操作系统的实际情况，设计合理的硬件输入模型。在 Hello China 的实现中，我们采用硬件输入管理对象来管理硬件输入到具体线程之间的连接。

5.2 中断与异常机制

不同的硬件平台，对于中断和异常的处理机制也不一样，比如，基于中断向量表的硬件平台（Intel 系列），在内存中维护了一个中断向量表，中断向量表中填写了用于处理中断的函数（中断处理函数）。
当中断发生时，硬件系统（CPU）根据中断号，索引中断向量表，找到对应的中断处理程序，然后保存当前的线程执行环境，跳转到中断处理程序继续执行。

而有的 CPU 则没有采用中断向量表的机制，比如，有的 CPU 的所有中断处理程序就是一个，在中断发生的时候，这样的 CPU 把一个中断号压入堆栈，然后直接调用中断处理程序。

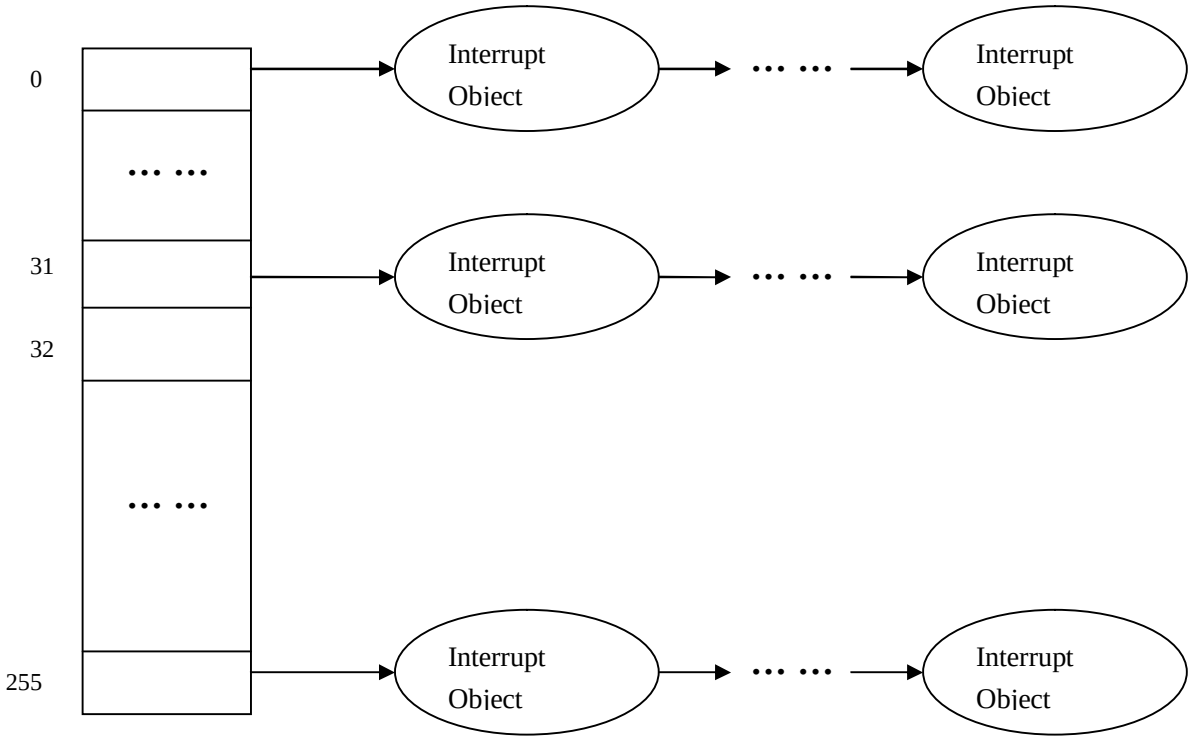
为兼容这两种 CPU 平台，Hello China 在实现的时候，采用了下面一种中断处理机制：

1、 维护一个统一的中断处理程序入口－GeneralInterruptHandler；

2、 针对中断向量表的硬件，对每个中断向量表项填写一个特定的处理例程，该例程仅仅是压入当前中断号，然后调用统一的中断入口；

3、 针对单中断处理程序的 CPU，直接调用该统一入口。

在系统内部，维护了一个中断对象链表数组，数组的每个项指向一个双向链表，中断对象构成了双向链表的元素（请参考下图）。



当中断发生后，统一入口例程（GeneralInterruptHandler）根据中断向量号，索引到合适的中断对象链表数组元素，获得双向链表的头指针，然后遍历这个链表，并调用中断对象的中断处理函数。

把同一向量的中断对象连接到一个链表中，实现了这样一种功能，即多个硬件设备共享一个中断硬件连接。统一入口例程在调用中断对象的处理函数时，如果该中断对象对该中断不感兴趣，那么只要返回 FALSE，则统一入口例程会继续调用下一个中断对象的处理函数，如果中断对象对该中断感兴趣，那么它处理该中断，并返回 TRUE。如果返回了 TRUE，就间接的告诉统一入口例程，中断已经得到了处理，于是统一中断例程就停止进一步的搜索，并从中断中返回。

系统对象（System Object）提供了中断功能处理接口，设备驱动程序可以调用 ConnectionInterrupt 例程（System 对象的一个方法）来注册一个中断处理例程，在注册的时候，可以指定中断号、中断处理函数等参数。

在驱动程序卸载时，可以调用 DisconnectInterrupt 例程来取消已经注册的中断处理函数（例程）。

5.3 硬件输入管理对象

中断机制解决了硬件输入如何通知操作系统的问题，但另外一个问题，即硬件输入如何通知核心线程的问题，仍然没有解决。为了解决这个问题，我们引入硬件输入管理对象（DeviceInputManager）的概念。

硬件输入管理对象，是一个系统对象，它内部维持了两个指针：一个 Shell 指针（Shell 是一个用户接口核心线程，参考用户接口部分），另外一个是当前焦点（Focus）指针，这两个指针的类型都是核心线程对象（KernelThreadObject）。该对象还提供了下列几个方法：

1、 SendDeviceMessage，驱动程序调用，通知硬件输入管理对象，一个硬件输入发生了；

2、 SetFocusThread，用户线程调用，用来更改当前焦点线程；

3、 SetShellThread，用户线程调用，用来更改当前 shell 对象指针；

4、 Initialize，初始化对象，初始化该对象。

这个时候，如果硬件设备发生了一个输入（中断），那么首先通过中断机制，把这个输入传递到操作系统，然后操作系统再找到合适的驱动程序，并把这个硬件输入时间传递到合适的驱动程序。

如果驱动程序发现这个事件需要进一步的传递给其它核心线程，那么，驱动程序可以调用 SendDeviceMessage 例程，来通知硬件输入管理器。

硬件输入管理器接收到该输入事件以后，会向当前焦点线程（内部维护了两个指针）发送一个消息。如果当前焦点线程不存在（指针为空），或者即使存在，但是其状态为 TERMINAL（通过核心线程对象获得），则向 Shell 线程发送一个消息，如果 Shell 线程也为空，那么硬件输入管理器什么都不作。

可以看出，硬件输入管理器起到了驱动程序和核心线程之间的连接纽带的作用。

一般情况下，只有一个当前焦点线程，一旦发生了焦点切换，Shell 线程会更改焦点线程（通过调用 SetFocusThread 方法），这样任何时候，硬件输入事件都会被正确的转发到当前焦点线程。

基于硬件输入管理器的硬件输入模型很好的解决了硬件输入和核心线程的连接问题。比如，在字符 Shell 模式下，如果用户通过命令运行了一个程序，如果用户明确指定该程序应该在后台执行，那么 Shell 在加载该程序的可执行文件，并创建核心线程之后，不更改当前焦点线程，那么所有硬件输入依然会被传递到字符 Shell。

如果用户不指定运行的前后台，那么，Shell 在启动一个程序的同时，会缺省情况下修改当前焦点线程（通过 SetFocusThread 函数），这样新启动的程序就可以直接获得硬件输入信号，在当前执行的线程结束后，Shell 会更改当前输入焦点线程。

需要说明的是，如果当前焦点线程运行结束（即其状态为 TERMINAL），即使没有更改当前焦点线程，那么硬件输入管理器也不会再继续向当前线程发送硬件输入消息，因为硬件管理器会判断当前焦点线程的状态，只有状态不是 TERMINAL，才向当前焦点线程发送硬件输入消息。

【责任编辑：iamxiaohan@hitbbs】

跟我一起写 Makefile

陈皓(haoel@hotmail.com)

概述

什么是 makefile？或许很多 Windows 的程序员都不知道这个东西，因为那些 Windows 的 IDE 都为你做了这个工作，但我觉得要作一个好的和 professional 的程序员，makefile 还是要懂。这就好像现在有这么多的 HTML 的编辑器，但如果你想成为一个专业人士，你还是要了解 HTML 的标识的含义。特别在 Unix 下的软件编译，你就不能不自己写 makefile 了，会不会写 makefile，从一个侧面说明了一个人是否具备完成大型工程的能力。

因为，makefile 关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile 定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 makefile 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令。

makefile 带来的好处就是——“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是一个命令工具，是一个解释 makefile 中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如：Delphi 的 make，Visual C++的 nmake，Linux 下 GNU 的 make。可见，makefile 都成为了一种在工程方面的编译方法。

现在讲述如何写 makefile 的文章比较少，这是我想写这篇文章的原因。当然，不同产商的 make 各不相同，也有不同的语法，但其本质都是在“文件依赖性”上做文章，这里，我仅对 GNU 的 make 进行讲述，我的环境是 Red Hat Linux 8.0，make 的版本是 3.80。必竟，这个 make 是应用最为广泛的，也是用得最多的。而且其还是最遵循于 IEEE 1003.2-1992 标准的（POSIX.2）。

在这篇文档中，将以 C/C++的源码作为我们基础，所以必然涉及一些关于 C/C++的编译的知识，相关于这方面的内容，还请各位查看相关的编译器的文档。这里所默认的编译器是 UNIX 下的 GCC 和 CC。

关于程序的编译和链接

在此，我想多说关于程序编译的一些规范和方法，一般来说，无论是 C、C++、还是 pas，首先要把源文件编译成中间代码文件，在 Windows 下也就是 .obj 文件，UNIX 下是 .o 文件，即 Object File，这个动作叫做编译（compile）。然后再把大量的 Object File 合成执行文件，这个动作叫作链接（link）。

编译时，编译器需要的是语法的正确，函数与变量的声明的正确。对于后者，通常是你需要告诉编译器头文件的所在位置（头文件中应该只是声明，而定义应该放在 C/C++文件中），只要所有的语法正确，编译器就可以编译出中间目标文件。一般来说，每个源文件都应该对应于一个中间目标文件（O 文件或是 OBJ 文件）。

链接时，主要是链接函数和全局变量，所以，我们可以使用这些中间目标文件（O 文件或是 OBJ 文件）来链接我们的应用程序。链接器并不管函数所在的源文件，只管函数的中间目标文件（Object File），在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，我们要给中间目标文件打个包，在 Windows 下这种包叫“库文件”（Library File），也就是 .lib 文件，在 UNIX 下，是 Archive File，也就是 .a 文件。

总结一下，源文件首先会生成中间目标文件，再由中间目标文件生成执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成 Object File。而在链接程序时，链接器会在所有的 Object File 中找寻函数的实现，如果找不到，那到就会报链接错误码（Linker Error），在 VC 下，这种错误一般是：Link 2001 错误，意思说是说，链接器未能找到函数的实现。你需要指定函数的 Object File.

好，言归正传，GNU 的 make 有许多的内容，闲言少叙，还是让我们开始吧。

Makefile 介绍

make 命令执行时，需要一个 Makefile 文件，以告诉 make 命令需要怎么样的去编译和链接程序。

首先，我们用一个示例来说明 Makefile 的书写规则。以便给大家一个感兴认识。这个示例来源于 GNU 的 make 使用手册，在这个示例中，我们的工程有 8 个 C 文件，和 3 个头文件，我们要写一个 Makefile 来告诉 make 命令如何编译和链接这几个文件。我们的规则是：

1）如果这个工程没有编译过，那么我们的所有 C 文件都要编译并被链接。

2）如果这个工程的某几个 C 文件被修改，那么我们只编译被修改的 C 文件，并链接目标程序。

3）如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的 C 文件，并链接目标程序。

只要我们的 Makefile 写得够好，所有的这一切，我们只用一个 make 命令就可以完成，make 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

一、Makefile 的规则

在讲述这个 Makefile 之前，还是让我们先来粗略地看一看 Makefile 的规则。

```
target ... : prerequisites ...
        command
        ...
        ...
```

target 也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，在后续的“伪目标”章节中会有叙述。

`prerequisites` 就是，要生成那个 target 所需要的文件或是目标。

`command` 也就是 make 需要执行的命令。（任意的 Shell 命令）

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。说白一点就是说，prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。

说到底，Makefile 的东西就是这样一点，好像我的这篇文档也该结束了。呵呵。还不尽然，这是 Makefile 的主线和核心，但要写好一个 Makefile 还不够，我会以后面一点一点地结合我的工作经验给你慢慢到来。内容还多着呢。:）

二、一个示例

正如前面所说的，如果一个工程有 3 个头文件，和 8 个 C 文件，我们为了完成前面所述的那三个规则，我们的 Makefile 应该是下面的这个样子的。

```
edit : main.o kbd.o command.o display.o \
```

```
        insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

反斜杠（\）是换行符的意思。这样比较便于 Makefile 的易读。我们可以把这个内容保存在文件为“Makefile”或“makefile”的文件中，然后在该目录下直接输入命令“make”就可以生成执行文件 edit。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下“make clean”就可以了。

在这个 makefile 中，目标文件（target）包含：执行文件 edit 和中间目标文件（*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h 文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 edit 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab 键作为开头。记住，make 并不管命令是怎么工作的，他只管执行所定义的命令。make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令。

这里要说明一点的是，clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 lable 一样，其冒号后什么也没有，那么，make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个 lable 的名字。这样的方法非常有用，我们可以在一个 makefile 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

三、make 是如何工作的

在默认的方式下，也就是我们只输入 make 命令。那么，

1、make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。

2、如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件作为最终的目标文件。

3、如果 edit 文件不存在，或是 edit 所依赖的后面的 .o 文件的文件修改时间要比 edit 这个文件新，那么，他就会执行后面所定义的命令来生成 edit 这个文件。

4、如果 edit 所依赖的.o 文件也存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件。（这有点像一个堆栈的过程）

5、当然，你的 C 文件和 H 文件是存在的啦，于是 make 会生成 .o 文件，然后再用 .o 文件生命 make 的终极任务，也就是执行文件 edit 了。

这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么 make 就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make 根本不理。make 只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

通过上述分析，我们知道，像 clean 这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要 make 执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。

于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如 file.c，那么根据我们的依赖性，我们的目标 file.o 会被重编译（也就是在这个依性关系后面所定义的命令），于是 file.o 的文件也是最新的啦，于是 file.o 的文件修改时间要比 edit 要新，所以 edit 也会被重新链接了（详见 edit 目标文件后定义的命令）。

而如果我们改变了“command.h”，那么，kdb.o、command.o 和 files.o 都会被重编译，并且，edit 会被重链接。

四、makefile 中使用变量

在上面的例子中，先让我们看看 edit 的规则：

```
edit : main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
       cc -o edit main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o
```

我们可以看到[.o]文件的字符串被重复了两次，如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在 clean 中）。当然，我们的 makefile 并不复杂，所以在两个地方加也不累，但如果 makefile 变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了 makefile 的易维护，在 makefile 中我们可以使用变量。makefile 的变量也就是一个字符串，理解成 C 语言中的宏可能会更好。

比如，我们声明一个变量，叫 objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什么啦，只要能够表示 obj 文件就行了。我们在 makefile 一开始就这样定义：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

于是，我们就可以很方便地在我们的 makefile 中以“$(objects)”的方式来使用这个变量了，于是我们的改良版 makefile 就变成下面这个样子：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
       cc -o edit $(objects)
main.o : main.c defs.h
       cc -c main.c
kbd.o : kbd.c defs.h command.h
       cc -c kbd.c
command.o : command.c defs.h command.h
       cc -c command.c
display.o : display.c defs.h buffer.h
       cc -c display.c
insert.o : insert.c defs.h buffer.h
       cc -c insert.c
```

```
search.o : search.c defs.h buffer.h
	cc -c search.c
files.o : files.c defs.h buffer.h command.h
	cc -c files.c
utils.o : utils.c defs.h
	cc -c utils.c
clean :
	rm edit $(objects)
```

于是如果有新的 .o 文件加入，我们只需简单地修改一下 objects 变量就可以了。关于变量更多的话题，我会在后续给你一一道来。

五、让 make 自动推导

GNU 的 make 很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的 make 会自动识别，并自己推导命令。

只要 make 看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果 make 找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件。并且 cc -c whatever.c 也会被推导出来，于是，我们的 makefile 再也不用写得这么复杂。我们的是新的 makefile 又出炉了。

```
objects = main.o kbd.o command.o display.o \
	insert.o search.o files.o utils.o

edit : $(objects)
	cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
	rm edit $(objects)
```

这种方法，也就是 make 的“隐晦规则”。上面文件内容中，“.PHONY”表示，clean 是个伪目标文件。关于更为详细的“隐晦规则”和“伪目标文件”，我会在后续给你一一道来。

六、另类风格的 makefile

即然我们的 make 可以自动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的重复的[.h]，能不能把其收拢起来，好吧，没有问题，这个对于 make 来说很容易，谁叫它提供了自动推导命令和文件的功能呢？来看看最新风格的 makefile 吧。

```
objects = main.o kbd.o command.o display.o \
	insert.o search.o files.o utils.o

edit : $(objects)
	cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

```
.PHONY : clean
clean :
	rm edit $(objects)
```

这种风格，让我们的 makefile 变得很简单，但我们的文件依赖关系就显得有点凌乱了。鱼和熊掌不可兼得。还看你的喜好了。我是不喜欢这种风格的，一是文件的依赖关系看不清楚，二是如果文件一多，要加入几个新的.o 文件，那就理不清楚了。

七、清空目标文件的规则

每个 Makefile 中都应该写一个清空目标文件（.o 和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个“修养”（呵呵，还记得我的《编程修养》吗）。一般的风格都是：

```
clean:
	rm edit $(objects)
```

更为稳健的做法是：

```
.PHONY : clean
clean :
	-rm edit $(objects)
```

前面说过，.PHONY 意思表示 clean 是一个“伪目标”，。而在 rm 命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean 的规则不要放在文件的开头，不然，这就会变成 make 的默认目标，相信谁也不愿意这样。不成文的规矩是——“clean 从来都是放在文件的最后”。

上面就是一个 makefile 的概貌，也是 makefile 的基础，下面还有很多 makefile 的相关细节，准备好了吗？准备好了就来。

Makefile 总述

一、Makefile 里有什么？

Makefile 里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

1、显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由 Makefile 的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

2、隐晦规则。由于我们的 make 有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写 Makefile，这是由 make 所支持的。

3、变量的定义。在 Makefile 中我们要定义一系列的变量，变量一般都是字符串，这个有点你 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。

4、文件指示。其包括了三个部分，一个是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样；另一个是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译#if 一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。

5、注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用“#”字符，这个就像 C/C++中的“//”一样。如果你要在你的 Makefile 中使用“#”字符，可以用反斜框进行转义，如：“\#”。

最后，还值得一提的是，在 Makefile 中的命令，必须要以[Tab]键开始。

二、Makefile 的文件名

默认的情况下，make 命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解释这个文件。在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用“GNUmakefile”，这个文件是 GNU 的 make 识别的。有另外一些 make 只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的 make 都支持“makefile”和“Makefile”这两种默认文件名。

当然，你可以使用别的文件名来书写 Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要指定特定的 Makefile，你可以使用 make 的“-f”和“--file”参数，如：make -f Make.Linux 或 make --file Make.AIX。

三、引用其它的 Makefile

在 Makefile 使用 include 关键字可以把别的 Makefile 包含进来，这很像 C 语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include 的语法是：

```
include <filename>
```

filename 可以是当前操作系统 Shell 的文件模式（可以保含路径和通配符）

在 include 前面可以有一些空字符，但是绝不能是[Tab]键开始。include 和<filename>可以用一个或多个空格隔开。举个例子，你有这样几个 Makefile：a.mk、b.mk、c.mk，还有一个文件叫 foo.make，以及一个变量$(bar)，其包含了 e.mk 和 f.mk，那么，下面的语句：

```
include foo.make *.mk $(bar)
```

等价于：

```
include foo.make a.mk b.mk c.mk e.mk f.mk
```

make 命令开始时，会把找寻 include 所指出的其它 Makefile，并把其内容安置在当前的位置。就好像 C/C++的#include 指令一样。如果文件都没有指定绝对路径或是相对路径的话，make 会在当前目录下首先寻找，如果当前目录下没有找到，那么，make 还会在下面的几个目录下找：

1、如果 make 执行时，有“-I”或“--include-dir”参数，那么 make 就会在这个参数所指定的目录下去寻找。

2、如果目录<prefix>/include（一般是：/usr/local/bin 或/usr/include）存在的话，make 也会去找。

如果有文件没有找到的话，make 会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成 makefile 的读取，make 会再重试这些没有找到，或是不能读取的文件，如果还是不行，make 才会出现一条致命信息。如果你想让 make 不理那些无法读取的文件，而继续执行，你可以在 include 前加一个减号“-”。如：

```
-include <filename>
```

其表示，无论 include 过程中出现什么错误，都不要报错继续执行。和其它版本 make 兼容的相关命令是 sinclude，其作用和这一个是一样的。

四、环境变量 MAKEFILES

如果你的当前环境中定义了环境变量 MAKEFILES，那么，make 会把这个变量中的值做一个类似于 include 的动作。这个变量中的值是其它的 Makefile，用空格分隔。只是，它和 include 不同的是，从这个环境变中引入的 Makefile 的“目标”不会起作用，如果环境变量中定义的文件发现错误，make 也会不理。

但是在这里我还是建议不要使用这个环境变量，因为只要这个变量一被定义，那么当你使用 make 时，所

有的 Makefile 都会受到它的影响，这绝不是你想看到的。在这里提这个事，只是为了告诉大家，也许有时候你的 Makefile 出现了怪事，那么你可以看看当前环境中有没有定义这个变量。

五、make 的工作方式

GNU 的 make 工作时的执行步骤入下：（想来其它的 make 也是类似）

1、读入所有的 Makefile。
2、读入被 include 的其它 Makefile。
3、初始化文件中的变量。
4、推导隐晦规则，并分析所有规则。
5、为所有的目标文件创建依赖关系链。
6、根据依赖关系，决定哪些目标要重新生成。
7、执行生成命令。

1-5 步为第一个阶段，6-7 为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make 会把其展开在使用的位置。但 make 并不会完全马上展开，make 使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

当然，这个工作方式你不一定要清楚，但是知道这个方式你也会对 make 更为熟悉。有了这个基础，后续部分也就容易看懂了。

书写规则

规则包含两个部分，一个是依赖关系，一个是生成目标的方法。

在 Makefile 中，规则的顺序是很重要的，因为，Makefile 中只应该有一个最终目标，其它的目标都是被这个目标所连带出来的，所以一定要让 make 知道你的最终目标是什么。一般来说，定义在 Makefile 中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make 所完成的也就是这个目标。

好了，还是让我们来看一看如何书写规则。

一、规则举例

```
foo.o : foo.c defs.h       # foo 模块
        cc -c -g foo.c
```

看到这个例子，各位应该不是很陌生了，前面也已说过，foo.o 是我们的目标，foo.c 和 defs.h 是目标所依赖的源文件，而只有一个命令“cc -c -g foo.c”（以 Tab 键开头）。这个规则告诉我们两件事：

1、文件的依赖关系，foo.o 依赖于 foo.c 和 defs.h 的文件，如果 foo.c 和 defs.h 的文件日期要比 foo.o 文件日期要新，或是 foo.o 不存在，那么依赖关系发生。

2、如果生成（或更新）foo.o 文件。也就是那个 cc 命令，其说明了，如何生成 foo.o 这个文件。（当然 foo.c 文件 include 了 defs.h 文件）

二、规则的语法

targets : prerequisites
command
...
或是这样：
targets : prerequisites ; command
command
...

targets 是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command 是命令行，如果其不与“target:prerequisites”在一行，那么，必须以[Tab 键]开头，如果和 prerequisites 在一行，那么可以用分号做为分隔。（见上）

prerequisites 也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。这个在前面已经讲过了。

如果命令太长，你可以使用反斜框（‘\’）作为换行符。make 对一行上有多少个字符没有限制。规则告诉 make 两件事，文件的依赖关系和如何成成目标文件。

一般来说，make 会以 UNIX 的标准 Shell，也就是/bin/sh 来执行命令。

三、在规则中使用通配符

如果我们想定义一系列比较类似的文件，我们很自然地就想起使用通配符。make 支持三各通配符：“*”，“?”和“[...]”。这是和 Unix 的 B-Shell 是相同的。

波浪号（“~”）字符在文件名中也有比较特殊的用途。如果是“~/test”，这就表示当前用户的$HOME 目录下的 test 目录。而“~hchen/test”则表示用户 hchen 的宿主目录下的 test 目录。（这些都是 Unix 下的小知识了，make 也支持）而在 Windows 或是 MS-DOS 下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量“HOME”而定。

通配符代替了你一系列的文件，如“*.c”表示所以后缀为 c 的文件。一个需要我们注意的是，如果我们的文件名中有通配符，如：“*”，那么可以用转义字符“\”，如“*”来表示真实的“*”字符，而不是任意长度的字符串。

好吧，还是先来看几个例子吧：

```
clean:
    rm -f *.o
```

上面这个例子我不不多说了，这是操作系统 Shell 所支持的通配符。这是在命令中的通配符。

```
print: *.c
    lpr -p $?
    touch print
```

上面这个例子说明了通配符也可以在我们的规则中，目标 print 依赖于所有的[.c]文件。其中的“$?”是一个自动化变量，我会在后面给你讲述。

```
objects = *.o
```

上面这个例子，表示了，通符同样可以用在变量中。并不是说[*.o]会展开，不！objects 的值就是“*.o”。Makefile 中的变量其实就是 C/C++中的宏。如果你要让通配符在变量中展开，也就是让 objects 的值是所有[.o]的文件名的集合，那么，你可以这样：

```
objects := $(wildcard *.o)
```

这种用法由关键字“wildcard”指出，关于 Makefile 的关键字，我们将在后面讨论。

四、文件搜寻

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当 make 需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉 make，让 make 在自动去找。

Makefile 文件中的特殊变量“VPATH”就是完成这个功能的，如果没有指明这个变量，make 只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make 就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。

```
VPATH = src:../headers
```

上面的的定义指定两个目录，“src”和“../headers”，make 会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

另一个设置文件搜索路径的方法是使用 make 的“vpath”关键字（注意，它是全小写的），这不是变量，这是一个 make 的关键字，这和上面提到的那个 VPATH 变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1、vpath <pattern> <directories>

为符合模式<pattern>的文件指定搜索目录<directories>。

2、vpath <pattern>

清除符合模式<pattern>的文件的搜索目录。

3、vpath

清除所有已被设置好了的文件搜索目录。

vapth 使用方法中的<pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符，例如，“%.h”表示所有以“.h”结尾的文件。<pattern>指定了要搜索的文件集，而<directories>则指定了<pattern>的文件集的搜索的目录。例如：

vpath %.h ../headers

该语句表示，要求 make 在“../headers”目录下搜索所有以“.h”结尾的文件。（如果某文件在当前目录没有找到的话）

我们可以连续地使用 vpath 语句，以指定不同搜索策略。如果连续的 vpath 语句中出现了相同的<pattern>，或是被重复了的<pattern>，那么，make 会按照 vpath 语句的先后顺序来执行搜索。如：

```
vpath %.c foo
vpath %  blish
vpath %.c bar
```

其表示“.c”结尾的文件，先在“foo”目录，然后是“blish”，最后是“bar”目录。

```
vpath %.c foo:bar
vpath %  blish
```

而上面的语句则表示“.c”结尾的文件，先在“foo”目录，然后是“bar”目录，最后才是“blish”目录。

五、伪目标

最早先的一个例子中，我们提到过一个“clean”的目标，这是一个“伪目标”，

clean:

rm *.o temp

正像我们前面例子中的“clean”一样，即然我们生成了许多文件编译文件，我们也应该提供一个清除它们

的“目标”以备完整地重编译而用。 （以“make clean”来使用该目标）

因为，我们并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以 make 无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”，向 make 说明，不管是否有这个文件，这个目标就是“伪目标”。

```
.PHONY : clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这样。于是整个过程可以这样写：

```
.PHONY: clean
clean:
	rm *.o temp
```

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。一个示例就是，如果你的 Makefile 需要一口气生成若干个可执行文件，但你只想简单地敲一个 make 完事，并且，所有的目标文件都写在一个 Makefile 中，那么你可以使用“伪目标”这个特性：

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
	cc -o prog1 prog1.o utils.o

prog2 : prog2.o
	cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
	cc -o prog3 prog3.o sort.o utils.o
```

我们知道，Makefile 中的第一个目标会被作为其默认目标。我们声明了一个“all”的伪目标，其依赖于其它三个目标。由于伪目标的特性是，总是被执行的，所以其依赖的那三个目标就总是不如“all”这个目标新。所以，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。“.PHONY : all”声明了“all”这个目标为“伪目标”。

随便提一句，从上面的例子我们可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖。看下面的例子：

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
	rm program

cleanobj :
	rm *.o

cleandiff :
	rm *.diff
```

“make clean”将清除所有要被清除的文件。“cleanobj”和“cleandiff”这两个伪目标有点像“子程序”的意思。我们可以输入“make cleanall”和“make cleanobj”和“make cleandiff”命令来达到清除不同种类文件的目的。

六、多目标

Makefile 的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并

且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令是同一个，这可能会可我们带来麻烦，不过好在我们的可以使用一个自动化变量“$@”（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合，这样说可能很抽象，还是看一个例子吧。

```
bigoutput littleoutput : text.g
	generate text.g -$(subst output,,$@) > $@
```

上述规则等价于：

```
bigoutput : text.g
	generate text.g -big > bigoutput
littleoutput : text.g
	generate text.g -little > littleoutput
```

其中，-$(subst output,,$@)中的“$”表示执行一个 Makefile 的函数，函数名为 subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，“$@”表示目标的集合，就像一个数组，“$@”依次取出目标，并执于命令。

七、静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。我们还是先来看一下语法：

<targets ...>: <target-pattern>: <prereq-patterns ...>
<commands>
...

targets 定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-parrtern 是指明了 targets 的模式，也就是的目标集模式。

prereq-parrterns 是目标的依赖模式，它对 target-parrtern 形成的模式再进行一次依赖目标的定义。

这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的<target-parrtern>定义成“%.o”，意思是我们的<target>集合中都是以“.o”结尾的，而如果我们的<prereq-parrterns>定义成“%.c”，意思是对<target-parrtern>所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>模式中的“%”（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合。

所以，我们的“目标模式”或是“依赖模式”中都应该有“%”这个字符，如果你的文件名中有“%”那么你可以使用反斜杠“\”进行转义，来标明真实的“%”字符。

看一个例子：

```
objects = foo.o bar.o
all: $(objects)
$(objects): %.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@
```

上面的例子中，指明了我们的目标从$object 中获取，“%.o”表明要所有以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量$object 集合的模式，而依赖模式“%.c”则取模式“%.o”的“%”，也就是“foo bar”，并为其加下“.c”的后缀，于是，我们的依赖目标就是“foo.c bar.c”。而命令中的“$<”和“$@”则是自动化变量，“$<”表示所有的依赖目标集（也就是“foo.c bar.c”），“$@”表示目标集（也就是“foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

```
foo.o : foo.c
	$(CC) -c $(CFLAGS) foo.c -o foo.o
bar.o : bar.c
	$(CC) -c $(CFLAGS) bar.c -o bar.o
```

试想，如果我们的“%.o”有几百个，那种我们只要用这种很简单的“静态模式规则”就可以写完一堆规则，实在是太有效率了。“静态模式规则”的用法很灵活，如果用得好，那会一个很强大的功能。再看一个例子：

```
files = foo.elc bar.o lose.o
$(filter %.o,$(files)): %.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
	emacs -f batch-byte-compile $<
```

$(filter %.o,$(files))表示调用 Makefile 的 filter 函数，过滤“$filter”集，只要其中模式为“%.o”的内容。其的它内容，我就不用多说了吧。这个例字展示了 Makefile 中更大的弹性。

八、自动生成依赖性

在 Makefile 中，我们的依赖关系可能会需要包含一系列的头文件，比如，如果我们的 main.c 中有一句“#include "defs.h"”，那么我们的依赖关系应该是：

```
main.o : main.c defs.h
```

但是，如果是一个比较大型的工程，你必需清楚哪些 C 文件包含了哪些头文件，并且，你在加入或删除头文件时，也需要小心地修改 Makefile，这是一个很没有维护性的工作。为了避免这种繁重而又容易出错的事情，我们可以使用 C/C++编译的一个功能。大多数的 C/C++编译器都支持一个“-M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果我们执行下面的命令：

```
cc -M main.c
```

其输出是：

```
main.o : main.c defs.h
```

于是由编译器自动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自动生成了。需要提醒一句的是，如果你使用 GNU 的 C/C++编译器，你得用“-MM”参数，不然，“-M”参数会把一些标准库的头文件也包含进来。

gcc -M main.c 的输出是：

```
main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \
    /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \
    /usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
    /usr/include/bits/sched.h /usr/include/libio.h \
    /usr/include/_G_config.h /usr/include/wchar.h \
    /usr/include/bits/wchar.h /usr/include/gconv.h \
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \
    /usr/include/bits/stdio_lim.h
```

gcc -MM main.c 的输出则是：

```
main.o: main.c defs.h
```

那么，编译器的这个功能如何与我们的 Makefile 联系在一起呢。因为这样一来，我们的 Makefile 也要根据这些源文件重新生成，让 Makefile 自己依赖于源文件？这个功能并不现实，不过我们可以有其它手段来迂回地实现这一功能。GNU 组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，为每一个“name.c”的文件都生成一个“name.d”的 Makefile 文件，[.d]文件中就存放对应[.c]文件的依赖关系。

于是，我们可以写出[.c]文件和[.d]文件的依赖关系，并让 make 自动更新或自成[.d]文件，并把其包含在我们的主 Makefile 中，这样，我们就可以自动化地生成每个文件的依赖关系了。

这里，我们给出了一个模式规则来产生[.d]文件：

```
%.d: %.c
	@set -e; rm -f $@; \
	 $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
	 sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
	 rm -f $@.$$$$
```

这个规则的意思是，所有的[.d]文件依赖于[.c]文件，“rm -f $@”的意思是删除所有的目标，也就是[.d]文件，第二行的意思是，为每个依赖文件“$<”，也就是[.c]文件生成依赖文件，“$@”表示模式“%.d”文件，如果有一个 C 文件是 name.c，那么“%”就是“name”，“$$$$”意为一个随机编号，第二行生成的文件有可

能是“name.d.12345”，第三行使用 sed 命令做了一个替换，关于 sed 命令的用法请参看相关的使用文档。第四行就是删除临时文件。

总而言之，这个模式要做的事就是在编译器生成的依赖关系中加入[.d]文件的依赖，即把依赖关系：

```
main.o : main.c defs.h
```

转成：

```
main.o main.d : main.c defs.h
```

于是，我们的[.d]文件也会自动更新了，并会自动生成了，当然，你还可以在这个[.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个[.d]文件都包含一个完赖的规则。一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主 Makefile 中。我们可以使用 Makefile 的“include”命令，来引入别的 Makefile 文件（前面讲过），例如：

```
sources = foo.c bar.c
include $(sources:.c=.d)
```

上述语句中的“$(sources:.c=.d)”中的“.c=.d”的意思是做一个替换，把变量$(sources)所有[.c]的字串都替换成[.d]，关于这个“替换”的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为 include 是按次来载入文件，最先载入的[.d]文件中的目标会成为默认目标。

书写命令

每条规则中的命令和操作系统 Shell 的命令行是一致的。make 会一按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以 Tab 键开头的，那么 make 会认为其是一个空命令。

我们在 UNIX 下可能会使用不同的 Shell，但是 make 的命令默认是被“/bin/sh”——UNIX 的标准 Shell 解释执行的。除非你特别指定一个其它的 Shell。Makefile 中，“#”是注释符，很像 C/C++中的“//”，其后的本行字符都被注释。

一、显示命令

通常，make 会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被 make 显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译 XXX 模块.....

当 make 执行时，会输出“正在编译 XXX 模块.....”字串，但不会输出命令，如果没有“@”，那么，make 将输出：

echo 正在编译 XXX 模块.....

正在编译 XXX 模块.....

如果 make 执行时，带入 make 参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的 Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而 make 参数“-s”或“--slient”则是全面禁止命令的显示。

二、命令执行

当依赖目标新于目标时，也就是当规则的目标需要被更新时，make 会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第

一条命令是 cd 命令，你希望第二条命令得在 cd 之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

示例一：

```
exec:
        cd /home/hchen
        pwd
```

示例二：

```
exec:
        cd /home/hchen; pwd
```

当我们执行“make exec”时，第一个例子中的 cd 没有作用，pwd 会打印出当前的 Makefile 目录，而第二个例子中，cd 就起作用了，pwd 会打印出“/home/hchen”。

make 一般是使用环境变量 SHELL 中所定义的系统 Shell 来执行命令，默认情况下使用 UNIX 的标准 Shell——/bin/sh 来执行命令。但在 MS-DOS 下有点特殊，因为 MS-DOS 下没有 SHELL 环境变量，当然你也可以指定。如果你指定了 UNIX 风格的目录形式，首先，make 会在 SHELL 所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在 PATH 环境变量中所定义的所有路径中寻找。MS-DOS 中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如“.exe”、“.com”、“.bat”、“.sh”等后缀。

三、命令出错

每当命令运行完后，make 会检测每个命令的返回码，如果命令返回成功，那么 make 会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么 make 就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如 mkdir 命令，我们一定需要建立一个目录，如果目录不存在，那么 mkdir 就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用 mkdir 的意思就是一定要有这样的一个目录，于是我们就不希望 mkdir 出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们可以在 Makefile 的命令行前加一个减号“-”（在 Tab 键之后），标记为不管命令出不出错都认为是成功的。如：

```
clean:
        -rm -f *.o
```

还有一个全局的办法是，给 make 加上“-i”或是“--ignore-errors”参数，那么，Makefile 中所有命令都会忽略错误。而如果一个规则是以“.IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

还有一个要提一下的 make 的参数的是“-k”或是“--keep-going”，这个参数的意思是，如果某规则中的命令出错了，那么就终目该规则的执行，但继续执行其它规则。

四、嵌套执行 make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的 Makefile，这有利于让我们的 Makefile 变得更加地简洁，而不至于把所有的东西全部写在一个 Makefile 中，这样会很难维护我们的 Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫 subdir，这个目录下有个 Makefile 文件，来指明了这个目录下文件的编译规则。那么我们总控的 Makefile 可以这样书写：

subsystem:

cd subdir && $(MAKE)

其等价于：

subsystem:

$(MAKE) -C subdir

定义$(MAKE)宏变量的意思是，也许我们的 make 需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“subdir”目录，然后执行 make 命令。

我们把这个 Makefile 叫做“总控 Makefile”，总控 Makefile 的变量可以传递到下级的 Makefile 中（如果你显示的声明），但是不会覆盖下层的 Makefile 中所定义的变量，除非指定了“-e”参数。

如果你要传递变量到下级 Makefile 中，那么你可以使用这样的声明：

export <variable ...>

如果你不想让某些变量传递到下级 Makefile 中，那么你可以这样声明：

unexport <variable ...>

如：

示例一：

export variable = value

其等价于：

variable = value

export variable

其等价于：

export variable := value

其等价于：

variable := value

export variable

示例二：

export variable += value

其等价于：

variable += value

export variable

如果你要传递所有的变量，那么，只要一个 export 就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是 SHELL，一个是 MAKEFLAGS，这两个变量不管你是否 export，其总是要传递到下层 Makefile 中，特别是 MAKEFILES 变量，其中包含了 make 的参数信息，如果我们执行“总控 Makefile”时有 make 参数或是在上层 Makefile 中定义了这个变量，那么 MAKEFILES 变量将会是这些参数，并会传递到下层 Makefile 中，这是一个系统级的环境变量。

但是 make 命令中的有几个参数并不往下传递，它们是“-C”,“-f”,“-h”“-o”和“-W”（有关 Makefile 参数的细节将在后面说明），如果你不想往下层传递参数，那么，你可以这样来：

subsystem:

cd subdir && $(MAKE) MAKEFLAGS=

如果你定义了环境变量 MAKEFLAGS，那么你得确信其中的选项是大家都会用到的，如果其中有“-t”,“-n”,和“-q”参数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。

还有一个在“嵌套执行”中比较有用的参数，“-w”或是“--print-directory”会在 make 的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级 make 目录是“/home/hchen/gnu/make”，如果我们使用“make -w”来执行，那么当进入该目录时，我们会看到：

make: Entering directory `/home/hchen/gnu/make'.

而在完成下层 make 后离开目录时，我们会看到：

make: Leaving directory `/home/hchen/gnu/make'

当你使用“-C”参数来指定 make 下层 Makefile 时，“-w”会被自动打开的。如果参数中有“-s”（“--slient”）或是“--no-print-directory”，那么，“-w”总是失效的。

五、定义命令包

如果 Makefile 中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以“define”开始，以“endef”结束，如：

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

这里，“run-yacc”是这个命令包的名字，其不要和 Makefile 中的变量重名。在“define”和“endef”中的两行就是命令序列。这个命令包中的第一个命令是运行 Yacc 程序，因为 Yacc 程序总是生成“y.tab.c”的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

```
foo.c : foo.y
        $(run-yacc)
```

我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包“run-yacc”中的“$^”就是“foo.y”，“$@”就是“foo.c”（有关这种以“$”开头的特殊变量，我们会在后面介绍），make 在执行命令包时，命令包中的每个命令会被依次独立执行。

使用变量

在 Makefile 中的定义的变量，就像是 C/C++语言中的宏一样，他代表了一个文本字串，在 Makefile 中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++所不同的是，你可以在 Makefile 中改变其值。在 Makefile 中，变量可以使用在“目标”，“依赖目标”，“命令”或是 Makefile 的其它部分中。

变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有“:”、“#”、“=”或是空字符（空格、回车等）。变量是大小写敏感的，“foo”、“Foo”和“FOO”是三个不同的变量名。传统的 Makefile 的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

有一些变量是很奇怪字串，如“$<”、“$@”等，这些是自动化变量，我会在后面介绍。

一、变量的基础

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“()”或是大括号“{}”把变量给包括起来。如果你要使用真实的“$”字符，那么你需要用“$$”来表示。

变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。先看一个例子：

```
objects = program.o foo.o utils.o
program : $(objects)
	cc -o program $(objects)
```

```
$(objects) : defs.h
```

变量会在使用它的地方精确地展开，就像 C/C++中的宏一样，例如：

```
foo = c
prog.o : prog.$(foo)
	$(foo)$(foo) -$(foo) prog.$(foo)
```

展开后得到：

```
prog.o : prog.c
	cc -c prog.c
```

当然，千万不要在你的 Makefile 中这样干，这里只是举个例子来表明 Makefile 中的变量在使用处展开的真实样子。可见其就是一个“替代”的原理。

另外，给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以，但我还是强烈建议你给变量加上括号。

二、变量中的变量

在定义变量的值时，我们可以使用其它变量来构造变量的值，在 Makefile 中有两种方式来在用变量定义变量的值。

先看第一种方式，也就是简单的使用“=”号，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后面定义的值。如：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:
	echo $(foo)
```

我们执行“make all”将会打出变量$(foo)的值是“Huh?”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”）可见，变量是可以使用后面的变量来定义的。

这个功能有好的地方，也有不好的地方，好的地方是，我们可以把变量的真实值推到后面来定义，如：

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

当“CFLAGS”在命令中被展开时，会是“-Ifoo -Ibar -O”。但这种形式也有不好的地方，那就是递归定义，如：

```
CFLAGS = $(CFLAGS) -O
```

或：

```
A = $(B)
B = $(A)
```

这会让 make 陷入无限的变量展开过程中去，当然，我们的 make 是有能力检测这样的定义，并会报错。还有就是如果在变量中使用函数，那么，这种方式会让我们的 make 运行时非常慢，更糟糕的是，他会使用得两个 make 的函数“wildcard”和“shell”发生不可预知的错误。因为你不会知道这两个函数会被调用多少次。

为了避免上面的这种方法，我们可以使用 make 中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符，如：

```
x := foo
y := $(x) bar
x := later
```

其等价于：

```
y := foo bar
x := later
```

值得一提的是，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

```
    y := $(x) bar
    x := foo
```

那么，y 的值是“bar”，而不是“foo bar”。

上面都是一些比较简单的变量使用了，让我们来看一个复杂的例子，其中包括了 make 的函数、条件表达式和一个系统变量“MAKELEVEL”的使用：

```
    ifeq (0,${MAKELEVEL})
    cur-dir   := $(shell pwd)
    whoami    := $(shell whoami)
    host-type := $(shell arch)
    MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
    endif
```

关于条件表达式和函数，我们在后面再说，对于系统变量“MAKELEVEL”，其意思是，如果我们的 make 有一个嵌套执行的动作（参见前面的“嵌套使用 make”），那么，这个变量会记录了我们的当前 Makefile 的调用层数。

下面再介绍两个定义变量时我们需要知道的，请先看一个例子，如果我们要定义一个变量，其值是一个空格，那么我们可以这样来：

```
    nullstring :=
    space := $(nullstring) # end of the line
```

nullstring 是一个 Empty 变量，其中什么也没有，而我们的 space 的值是一个空格。因为在操作符的右边是很难描述一个空格的，这里采用的技术很管用，先用一个 Empty 变量来标明变量的值开始了，而后面采用“#”注释符来表示变量定义的终止，这样，我们可以定义出其值是一个空格的变量。请注意这里关于“#”的使用，注释符“#”的这种特性值得我们注意，如果我们这样定义一个变量：

```
    dir := /foo/bar    # directory to put the frobs in
```

dir 这个变量的值是“/foo/bar”，后面还跟了 4 个空格，如果我们这样使用这样变量来指定别的目录——“$(dir)/file”那么就完蛋了。

还有一个比较有用的操作符是“?=”，先看示例：

```
    FOO ?= bar
```

其含义是，如果 FOO 没有被定义过，那么变量 FOO 的值就是“bar”，如果 FOO 先前被定义过，那么这条语将什么也不做，其等价于：

```
    ifeq ($(origin FOO), undefined)
      FOO = bar
    endif
```

三、变量高级用法

这里介绍两种变量的高级使用方法，第一种是变量值的替换。

我们可以替换变量中的共有的部分，其格式是“$(var:a=b)”或是“${var:a=b}”，其意思是，把变量“var”中所有以“a”字串“结尾”的“a”替换成“b”字串。这里的“结尾”意思是“空格”或是“结束符”。

还是看一个示例吧：

```
    foo := a.o b.o c.o
    bar := $(foo:.o=.c)
```

这个示例中，我们先定义了一个“$(foo)”变量，而第二行的意思是把“$(foo)”中所有以“.o”字串“结尾”全部替换成“.c”，所以我们的“$(bar)”的值就是“a.c b.c c.c”。

另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：

```
    foo := a.o b.o c.o
    bar := $(foo:%.o=%.c)
```

这依赖于被替换字串中的有相同的模式，模式中必须包含一个“%”字符，这个例子同样让$(bar)变量的值为“a.c b.c c.c”。

第二种高级用法是——“把变量的值再当成变量”。先看一个例子：

```
x = y
y = z
a := $($(x))
```

在这个例子中，$(x)的值是“y”，所以$($(x))就是$(y)，于是$(a)的值就是“z”。（注意，是“x=y”，而不是“x=$(y)”）

我们还可以使用更多的层次：

```
x = y
y = z
z = u
a := $($($(x)))
```

这里的$(a)的值是“u”，相关的推导留给读者自己去做吧。

让我们再复杂一点，使用上“在变量定义中使用变量”的第一个方式，来看一个例子：

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

这里的$($(x))被替换成了$($(y))，因为$(y)值是“z”，所以，最终结果是：a:=$(z)，也就是“Hello”。

再复杂一点，我们再加上函数：

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

这个例子中，“$($($(z)))”扩展为“$($(y))”，而其再次被扩展为“$($(subst 1,2,$(x)))”。$(x)的值是“variable1”，subst 函数把“variable1”中的所有“1”字串替换成“2”字串，于是，“variable1”变成“variable2”，再取其值，所以，最终，$(a)的值就是$(variable2)的值——“Hello”。（喔，好不容易）

在这种方式中，或要可以使用多个变量来组成一个变量的名字，然后再取其值：

```
first_second = Hello
a = first
b = second
all = $($a_$b)
```

这里的“$a_$b”组成了“first_second”，于是，$(all)的值就是“Hello”。

再来看看结合第一种技术的例子：

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

sources := $($(a1)_objects:.o=.c)
```

这个例子中，如果$(a1)的值是“a”的话，那么，$(sources)的值就是“a.c b.c c.c”；如果$(a1)的值是“1”，那么$(sources)的值是“1.c 2.c 3.c”。

再来看一个这种技术和“函数”与“条件语句”一同使用的例子：

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```

这个示例中，如果定义了“do_sort”，那么：foo := $(sort a d b g q c)，于是$(foo)的值就是“a b c d g q”，而如果没有定义“do_sort”，那么：foo := $(sort a d b g q c)，调用的就是 strip 函数。

当然，“把变量的值再当成变量”这种技术，同样可以用在操作符的左边：

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
```

```
define $(dir)_print
lpr $($(dir)_sources)
endef
```

这个例子中定义了三个变量：“dir”，“foo_sources”和“foo_print”。

四、追加变量值

我们可以使用“+=”操作符给变量追加值，如：

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

于是，我们的$(objects)值变成：“main.o foo.o bar.o utils.o another.o”（another.o 被追加进去了）

使用“+=”操作符，可以模拟为下面的这种例子：

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

所不同的是，用“+=”更为简洁。

如果变量之前没有定义过，那么，“+=”会自动变成“=”，如果前面有变量定义，那么“+=”会继承于前次操作的赋值符。如果前一次的是“:=”，那么“+=”会以“:=”作为其赋值符，如：

```
variable := value
variable += more
```

等价于：

```
variable := value
variable := $(variable) more
```

但如果是这种情况：

```
variable = value
variable += more
```

由于前次的赋值符是“=”，所以“+=”也会以“=”来做为赋值，那么岂不会发生变量的递补归定义，这是很不好的，所以 make 会自动为我们解决这个问题，我们不必担心这个问题。

五、override 指示符

如果有变量是通常 make 的命令行参数设置的，那么 Makefile 中对这个变量的赋值会被忽略。如果你想在 Makefile 中设置这类参数的值，那么，你可以使用“override”指示符。其语法是：

override <variable> = <value>

override <variable> := <value>

当然，你还可以追加：

override <variable> += <more text>

对于多行的变量定义，我们用 define 指示符，在 define 指示符前，也同样可以使用 ovveride 指示符，如：

override define foo

bar

endef

六、多行变量

还有一种设置变量值的方法是使用 define 关键字。使用 define 关键字设置变量的值可以有换行，这有利于定义一系列的命令（前面我们讲过“命令包”的技术就是利用这个关键字）。

define 指示符后面跟的是变量的名字，而重起一行定义变量的值，定义是以 endef 关键字结束。其工作方

式和“=”操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以[Tab]键开头，所以如果你用 define 定义的命令变量中没有以[Tab]键开头，那么 make 就不会把其认为是命令。

下面的这个示例展示了 define 的用法：

```
define two-lines
echo foo
echo $(bar)
endef
```

七、环境变量

make 运行时的系统环境变量可以在 make 开始运行时被载入到 Makefile 文件中，但是如果 Makefile 中已定义了这个变量，或是这个变量由 make 命令行带入，那么系统的环境变量的值将被覆盖。（如果 make 指定了“-e”参数，那么，系统环境变量将覆盖 Makefile 中定义的变量）

因此，如果我们在环境变量中设置了“CFLAGS”环境变量，那么我们就可以在所有的 Makefile 中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。如果 Makefile 中定义了 CFLAGS，那么则会使用 Makefile 中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“全局变量”和“局部变量”的特性。

当 make 嵌套调用时（参见前面的“嵌套调用”章节），上层 Makefile 中定义的变量会以系统环境变量的方式传递到下层的 Makefile 中。当然，默认情况下，只有通过命令行设置的变量会被传递。而定义在文件中的变量，如果要向下层 Makefile 传递，则需要使用 exprot 关键字来声明。（参见前面章节）

当然，我并不推荐把许多的变量都定义在系统环境中，这样，在我们执行不用的 Makefile 时，拥有的是同一套系统变量，这可能会带来更多的麻烦。

八、目标变量

前面我们所讲的在 Makefile 中定义的变量都是“全局变量”，在整个文件，我们都可以访问这些变量。当然，“自动化变量”除外，如“$<”等这种类量的自动化变量就属于“规则型变量”，这种变量的值依赖于规则的目标和依赖目标的定义。

当然，我样同样可以为某个目标设置局部变量，这种变量被称为“Target-specific Variable”，它可以和“全局变量”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

其语法是：

<target ...> : <variable-assignment>

<target ...> : overide <variable-assignment>

<variable-assignment>可以是前面讲过的各种赋值表达式，如“=”、“:=”、“+=”或是“？=”。第二个语法是针对于 make 命令行带入的变量，或是系统环境变量。

这个特性非常的有用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的所有的规则中去。如：

```
prog : CFLAGS = -g
prog : prog.o foo.o bar.o
	$(CC) $(CFLAGS) prog.o foo.o bar.o

prog.o : prog.c
	$(CC) $(CFLAGS) prog.c
```

```
foo.o : foo.c
	$(CC) $(CFLAGS) foo.c

bar.o : bar.c
	$(CC) $(CFLAGS) bar.c
```

在这个示例中，不管全局的$(CFLAGS)的值是什么，在 prog 目标，以及其所引发的所有规则中（prog.o foo.o bar.o 的规则），$(CFLAGS)的值都是“-g”

九、模式变量

在 GNU 的 make 中，还支持模式变量（Pattern-specific Variable），通过上面的目标变量中，我们知道，变量可以定义在某个目标上。模式变量的好处就是，我们可以给定一种“模式”，可以把变量定义在符合这种模式的所有目标上。

我们知道，make 的“模式”一般是至少含有一个“%”的，所以，我们可以以如下方式给所有以[.o]结尾的目标定义目标变量：

```
%.o : CFLAGS = -O
```

同样，模式变量的语法和“目标变量”一样：

<pattern ...> : <variable-assignment>

<pattern ...> : override <variable-assignment>

override 同样是针对于系统环境传入的变量，或是 make 命令行指定的变量。

使用条件判断

使用条件判断，可以让 make 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

一、示例

下面的例子，判断$(CC)变量是否“gcc”，如果是的话，则使用 GNU 函数编译目标。

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
	$(CC) -o foo $(objects) $(libs_for_gcc)
else
	$(CC) -o foo $(objects) $(normal_libs)
endif
```

可见，在上面示例的这个规则中，目标“foo”可以根据变量“$(CC)”值来选取不同的函数库来编译程序。

我们可以从上面的示例中看到三个关键字：ifeq、else 和 endif。ifeq 的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。else 表示条件表达式为假的情况。endif 表示一个条件语句的结束，任何一个条件表达式都应该以 endif 结束。

当我们的变量$(CC)值是“gcc”时，目标 foo 的规则是：

```
foo: $(objects)
	$(CC) -o foo $(objects) $(libs_for_gcc)
```

而当我们的变量$(CC)值不是“gcc”时（比如“cc”），目标 foo 的规则是：

```
foo: $(objects)
	$(CC) -o foo $(objects) $(normal_libs)
```

当然，我们还可以把上面的那个例子写得更简洁一些：

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

foo: $(objects)
	$(CC) -o foo $(objects) $(libs)
```

二、语法

条件表达式的语法为：

<conditional-directive>
<text-if-true>
endif

以及：

<conditional-directive>
<text-if-true>
else
<text-if-false>
endif

其中<conditional-directive>表示条件关键字，如“ifeq”。这个关键字有四个。

第一个是我们前面所见过的“ifeq”

ifeq (<arg1>, <arg2>)
ifeq '<arg1>' '<arg2>'
ifeq "<arg1>" "<arg2>"
ifeq "<arg1>" '<arg2>'
ifeq '<arg1>' "<arg2>"

比较参数“arg1”和“arg2”的值是否相同。当然，参数中我们还可以使用 make 的函数。如：

ifeq ($(strip $(foo)),)
<text-if-empty>
endif

这个示例中使用了“strip”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。

第二个条件关键字是“ifneq”。语法是：

ifneq (<arg1>, <arg2>)
ifneq '<arg1>' '<arg2>'
ifneq "<arg1>" "<arg2>"
ifneq "<arg1>" '<arg2>'
ifneq '<arg1>' "<arg2>"

其比较参数“arg1”和“arg2”的值是否相同，如果不同，则为真。和“ifeq”类似。

第三个条件关键字是“ifdef”。语法是：

ifdef <variable-name>

如果变量<variable-name>的值非空，那到表达式为真。否则，表达式为假。当然，<variable-name>同样可以是一个函数的返回值。注意，ifdef 只是测试一个变量是否有值，其并不会把变量扩展到当前位置。

还是来看两个例子：

示例一：

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

示例二：

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

第一个例子中，“$(frobozz)”值是“yes”，第二个则是“no”。

第四个条件关键字是“ifndef”。其语法是：

ifndef <variable-name>

这个我就不多说了，和“ifdef”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“#”同样也是安全的。“else”和“endif”也一样，只要不是以[Tab]键开始就行了。

特别注意的是，make 是在读取 Makefile 时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，你最好不要把自动化变量（如“$@”等）放入条件表达式中，因为自动化变量是在运行时才有的。

而且，为了避免混乱，make 不允许把整个条件语句分成两部分放在不同的文件中。

使用函数

在 Makefile 中可以使用函数来处理变量，从而让我们的命令或是规则更为的灵活和具有智能。make 所支持的函数也不算很多，不过已经足够我们的操作了。函数调用后，函数的返回值可以当做变量来使用。

一、函数的调用语法

函数调用，很像变量的使用，也是以“$”来标识的，其语法如下：

$(<function> <arguments>)

或是

${<function> <arguments>}

这里，<function>就是函数名，make 支持的函数不多。<arguments>是函数的参数，参数间以逗号“,”分隔，而函数名和参数之间以“空格”分隔。函数调用以“$”开头，以圆括号或花括号把函数名和参数括起。感觉很像一个变量，是不是？函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“$(subst a,b,$(x))”这样的形式，而不是“$(subst a,b,${x})”的形式。因为统一会更清楚，也会减少一些不必

要的麻烦。

还是来看一个示例：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

在这个示例中，$(comma)的值是一个逗号。$(space)使用了$(empty)定义了一个空格，$(foo)的值是“a b c”，$(bar)的定义用，调用了函数“subst”，这是一个替换函数，这个函数有三个参数，第一个参数是被替换字串，第二个参数是替换字串，第三个参数是替换操作作用的字串。这个函数也就是把$(foo)中的空格替换成逗号，所以$(bar)的值是“a,b,c”。

二、字符串处理函数

$(subst <from>,<to>,<text>)

名称：字符串替换函数——subst。

功能：把字串<text>中的<from>字符串替换成<to>。

返回：函数返回被替换过后的字符串。

示例：

```
$(subst ee,EE,feet on the street),
```

把“feet on the street”中的“ee”替换成“EE”，返回结果是“fEEt on the strEEt”。

$(patsubst <pattern>,<replacement>,<text>)

名称：模式字符串替换函数——patsubst。

功能：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符“%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（可以用“\”来转义，以“\%”来表示真实含义的“%”字符）

返回：函数返回被替换过后的字符串。

示例：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

把字串“x.c.c bar.c”符合模式[%.c]的单词替换成[%.o]，返回结果是“x.c.o bar.o”

备注：

这和我们前面“变量章节”说过的相关知识有点相似。如：

“$(var:<pattern>=<replacement>)”

相当于

“$(patsubst <pattern>,<replacement>,$(var))”，

而“$(var: <suffix>=<replacement>)”

则相当于

“$(patsubst %<suffix>,%<replacement>,$(var))”。

例如有：objects = foo.o bar.o baz.o，

那么，“$(objects:.o=.c)”和“$(patsubst %.o,%.c,$(objects))”是一样的。

$(strip <string>)

名称：去空格函数——strip。

功能：去掉<string>字串中开头和结尾的空字符。

返回：返回被去掉空格的字符串值。

示例：

```
$(strip a b c )
```

把字串“a b c ”去到开头和结尾的空格，结果是“a b c”。

$(findstring <find>,<in>)

名称：查找字符串函数——findstring。

功能：在字串<in>中查找<find>字串。

返回：如果找到，那么返回<find>，否则返回空字符串。

示例：

```
$(findstring a,a b c)
$(findstring a,b c)
```

第一个函数返回“a”字符串，第二个返回“”字符串（空字符串）

$(filter <pattern...>,<text>)

名称：过滤函数——filter。

功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。

返回：返回符合模式<pattern>的字串。

示例：

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
      cc $(filter %.c %.s,$(sources)) -o foo
```

$(filter %.c %.s,$(sources))返回的值是“foo.c bar.c baz.s”。

$(filter-out <pattern...>,<text>)

名称：反过滤函数——filter-out。

功能：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。

返回：返回不符合模式<pattern>的字串。

示例：

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

$(filter-out $(mains),$(objects)) 返回值是“foo.o bar.o”。

$(sort <list>)

名称：排序函数——sort。

功能：给字符串<list>中的单词排序（升序）。

返回：返回排序后的字符串。

示例：$(sort foo bar lose)返回“bar foo lose” 。

备注：sort 函数会去掉<list>中相同的单词。

$(word <n>,<text>)

名称：取单词函数——word。

功能：取字符串<text>中第<n>个单词。（从一开始）

返回：返回字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返回空字符串。

示例：$(word 2, foo bar baz)返回值是“bar”。

$(wordlist <s>,<e>,<text>)

名称：取单词串函数——wordlist。

功能：从字符串<text>中取从<s>开始到<e>的单词串。<s>和<e>是一个数字。

返回：返回字符串<text>中从<s>到<e>的单词字串。如果<s>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串。

示例： $(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

$(words <text>)

名称：单词个数统计函数——words。

功能：统计<text>中字符串中的单词个数。

返回：返回<text>中的单词数。

示例：$(words, foo bar baz)返回值是“3”。

备注：如果我们要取<text>中最后的一个单词，我们可以这样：$(word $(words <text>),<text>)。

$(firstword <text>)

名称：首单词函数——firstword。

功能：取字符串<text>中的第一个单词。

返回：返回字符串<text>的第一个单词。

示例：$(firstword foo bar)返回值是“foo”。

备注：这个函数可以用 word 函数来实现：$(word 1,<text>)。

以上，是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能。这里，举一个现实中应用的例子。我们知道，make 使用“VPATH”变量来指定“依赖文件”的搜索路径。于是，我们可以利用这个搜索路径来指定编译器对头文件的搜索路径参数 CFLAGS，如：

```
override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))
```

如果我们的“$(VPATH)”值是“src:../headers”，那么“$(patsubst %,-I%,$(subst :, ,$(VPATH)))”将返回“-Isrc -I../headers”，这正是 cc 或 gcc 搜索头文件路径的参数。

三、文件名操作函数

下面我们要介绍的函数主要是处理文件名的。每个函数的参数字符串都会被当做一个或是一系列的文件名来对待。

$(dir <names...>)

名称：取目录函数——dir。

功能：从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（“/”）之前的部分。如果没有反斜杠，那么返回“./”。

返回：返回文件名序列<names>的目录部分。

示例： $(dir src/foo.c hacks)返回值是“src/ ./”。

$(notdir <names...>)

名称：取文件函数——notdir。

功能：从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（“/”）之后的部分。

返回：返回文件名序列<names>的非目录部分。

示例： $(notdir src/foo.c hacks)返回值是“foo.c hacks”。

$(suffix <names...>)

名称：取后缀函数——suffix。

功能：从文件名序列<names>中取出各个文件名的后缀。

返回：返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串。

示例：$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

$(basename <names...>)

名称：取前缀函数——basename。

功能：从文件名序列<names>中取出各个文件名的前缀部分。

返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字串。

示例：$(basename src/foo.c src-1.0/bar.c hacks)返回值是“src/foo src-1.0/bar hacks”。

$(addsuffix <suffix>,<names...>)

名称：加后缀函数——addsuffix。

功能：把后缀<suffix>加到<names>中的每个单词后面。

返回：返回加过后缀的文件名序列。

示例：$(addsuffix .c,foo bar)返回值是“foo.c bar.c”。

$(addprefix <prefix>,<names...>)

名称：加前缀函数——addprefix。

功能：把前缀<prefix>加到<names>中的每个单词后面。

返回：返回加过前缀的文件名序列。

示例：$(addprefix src/,foo bar)返回值是“src/foo src/bar”。

$(join <list1>,<list2>)

名称：连接函数——join。

功能：把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list2>中。

返回：返回连接过后的字符串。

示例：$(join aaa bbb , 111 222 333)返回值是“aaa111 bbb222 333”。

四、foreach 函数

foreach 函数和别的函数非常的不一样。因为这个函数是用来做循环用的，Makefile 中的 foreach 函数几乎是仿照于 Unix 标准 Shell（/bin/sh）中的 for 语句，或是 C-Shell（/bin/csh）中的 foreach 语句而构建的。它的语法是：

$(foreach <var>,<list>,<text>)

这个函数的意思是，把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，

最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是 foreach 函数的返回值。

所以，<var>最好是一个变量名，<list>可以是一个表达式，而<text>中一般会使用<var>这个参数来依次枚举<list>中的单词。举个例子：

```
names := a b c d
files := $(foreach n,$(names),$(n).o)
```

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为 foreach 函数的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

注意，foreach 中的<var>参数是一个临时的局部变量，foreach 函数执行完后，参数<var>的变量将不在作用，其作用域只在 foreach 函数当中。

五、if 函数

if 函数很像 GNU 的 make 所支持的条件语句——ifeq（参见前面所述的章节），if 函数的语法是：

$(if <condition>,<then-part>)

或是

$(if <condition>,<then-part>,<else-part>)

可见，if 函数可以包含“else”部分，或是不含。即 if 函数的参数可以是两个，也可以是三个。<condition>参数是 if 的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则<else-part>会被计算。

而 if 函数的返回值是，如果<condition>为真（非空字符串），那个<then-part>会是整个函数的返回值，如果<condition>为假（空字符串），那么<else-part>会是整个函数的返回值，此时如果<else-part>没有被定义，那么，整个函数返回空字串。

所以，<then-part>和<else-part>只会有一个被计算。

六、call 函数

call 函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用 call 函数来向这个表达式传递参数。其语法是：

$(call <expression>,<parm1>,<parm2>,<parm3>...)

当 make 执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是 call 函数的返回值。例如：

```
reverse =  $(1) $(2)
foo = $(call reverse,a,b)
```

那么，foo 的值就是“a b”。当然，参数的次序是可以自定义的，不一定是顺序的，如：

```
reverse =  $(2) $(1)
foo = $(call reverse,a,b)
```

此时的 foo 的值就是“b a”。

七、origin 函数

origin 函数不像其它的函数，他并不操作变量的值，他只是告诉你你的这个变量是哪里来的？其语法是：

```
$(origin <variable>)
```

注意，<variable>是变量的名字，不应该是引用。所以你最好不要在<variable>中使用“$”字符。Origin 函数会

以其返回值来告诉你这个变量的“出生情况”，下面，是 origin 函数的返回值:

“undefined”

如果<variable>从来没有定义过，origin 函数返回这个值“undefined”。

“default”

如果<variable>是一个默认的定义，比如“CC”这个变量，这种变量我们将在后面讲述。

“environment”

如果<variable>是一个环境变量，并且当 Makefile 被执行时，“-e”参数没有被打开。

“file”

如果<variable>这个变量被定义在 Makefile 中。

“command line”

如果<variable>这个变量是被命令行定义的。

“override”

如果<variable>是被 override 指示符重新定义的。

“automatic”

如果<variable>是一个命令运行中的自动化变量。关于自动化变量将在后面讲述。

这些信息对于我们编写 Makefile 是非常有用的，例如，假设我们有一个 Makefile 其包了一个定义文件 Make.def，在 Make.def 中定义了一个变量“bletch”，而我们的环境中也有一个环境变量“bletch”，此时，我们想判断一下，如果变量来源于环境，那么我们就把之重定义了，如果来源于 Make.def 或是命令行等非环境的，那么我们就不重新定义它。于是，在我们的 Makefile 中，我们可以这样写：

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

当然，你也许会说，使用 override 关键字不就可以重新定义环境中的变量了吗？为什么需要使用这样的步骤？是的，我们用 override 是可以达到这样的效果，可是 override 过于粗暴，它同时会把从命令行定义的变量也覆盖了，而我们只想重新定义环境传来的，而不想重新定义命令行传来的。

八、shell 函数

shell 函数也不像其它的函数。顾名思义，它的参数应该就是操作系统 Shell 的命令。它和反引号“`”是相同的功能。这就是说，shell 函数把执行操作系统命令后的输出作为函数返回。于是，我们可以用操作系统命令以及字符串处理命令 awk，sed 等等命令来生成一个变量，如：

```
contents := $(shell cat foo)
files := $(shell echo *.c)
```

注意，这个函数会新生成一个 Shell 程序来执行命令，所以你要注意其运行性能，如果你的 Makefile 中有一些比较复杂的规则，并大量使用了这个函数，那么对于你的系统性能是有害的。特别是 Makefile 的隐晦的规则可能会让你的 shell 函数执行的次数比你想像的多得多。

九、控制 make 的函数

make 提供了一些函数来控制 make 的运行。通常，你需要检测一些运行 Makefile 时的运行时信息，并且根据这些信息来决定，你是让 make 继续执行，还是停止。

$(error <text ...>)

产生一个致命的错误，<text ...>是错误信息。注意，error 函数不会在一被使用就会产生错误信息，所以如果你把其定义在某个变量中，并在后续的脚本中使用这个变量，那么也是可以的。例如：

示例一：

```
ifdef ERROR_001
$(error error is $(ERROR_001))
endif
```

示例二：

```
ERR = $(error found an error!)
.PHONY: err
err: ; $(ERR)
```

示例一会在变量 ERROR_001 定义了后执行时产生 error 调用，而示例二则在目录 err 被执行时才发生 error 调用。

$(warning <text ...>)

这个函数很像 error 函数，只是它并不会让 make 退出，只是输出一段警告信息，而 make 继续执行。

make 的运行

一般来说，最简单的就是直接在命令行下输入 make 命令，make 命令会找当前目录的 makefile 来执行，一切都是自动的。但也有时你也许只想让 make 重编译某些文件，而不是整个工程，而又有的时候你有几套编译规则，你想在不同的时候使用不同的编译规则，等等。本章节就是讲述如何使用 make 命令的。

一、make 的退出码

make 命令执行后有三个退出码：

0 —— 表示成功执行。

1 —— 如果 make 运行时出现任何错误，其返回 1。

2 —— 如果你使用了 make 的“-q”选项，并且 make 使得一些目标不需要更新，那么返回 2。

Make 的相关参数我们会在后续章节中讲述。

二、指定 Makefile

前面我们说过，GNU make 找寻默认的 Makefile 的规则是在当前目录下依次找三个文件——“GNUmakefile”、“makefile”和“Makefile”。其按顺序找这三个文件，一旦找到，就开始读取这个文件并执行。

当前，我们也可以给 make 命令指定一个特殊名字的 Makefile。要达到这个功能，我们要使用 make 的“-f”或是“--file”参数（“--makefile”参数也行）。例如，我们有个 makefile 的名字是“hchen.mk”，那么，我们可以这样来让 make 来执行这个文件：

```
make –f hchen.mk
```

如果在 make 的命令行是，你不只一次地使用了“-f”参数，那么，所有指定的 makefile 将会被连在一起传递给 make 执行。

三、指定目标

一般来说，make 的最终目标是 makefile 中的第一个目标，而其它目标一般是由这个目标连带出来的。这是 make 的默认行为。当然，一般来说，你的 makefile 中的第一个目标是由许多个目标组成，你可以指示 make，让其完成你所指定的目标。要达到这一目的很简单，需在 make 命令后直接跟目标的名字就可以完成（如前面提到的“make clean”形式）

任何在 makefile 中的目标都可以被指定成终极目标，但是除了以“-”打头，或是包含了“=”的目标，因为有这些字符的目标，会被解析成命令行参数或是变量。甚至没有被我们明确写出来的目标也可以成为 make 的终极目标，也就是说，只要 make 可以找到其隐含规则推导规则，那么这个隐含目标同样可以被指定成终极目标。

有一个 make 的环境变量叫“MAKECMDGOALS”，这个变量中会存放你所指定的终极目标的列表，如果在命令行上，你没有指定目标，那么，这个变量是空值。这个变量可以让你使用在一些比较特殊的情形下。比如下面的例子：

```
sources = foo.c bar.c
ifneq ( $(MAKECMDGOALS),clean)
include $(sources:.c=.d)
endif
```

基于上面的这个例子，只要我们输入的命令不是“make clean”，那么 makefile 会自动包含“foo.d”和“bar.d”这两个 makefile。

使用指定终极目标的方法可以很方便地让我们编译我们的程序，例如下面这个例子：

```
.PHONY: all
all: prog1 prog2 prog3 prog4
```

从这个例子中，我们可以看到，这个 makefile 中有四个需要编译的程序——“prog1”，“prog2”，“prog3”和“prog4”，我们可以使用“make all”命令来编译所有的目标（如果把 all 置成第一个目标，那么只需执行“make”），我们也可以使用“make prog2”来单独编译目标“prog2”。

即然 make 可以指定所有 makefile 中的目标，那么也包括“伪目标”，于是我们可以根据这种性质来让我们的 makefile 根据指定的不同的目标来完成不同的事。在 Unix 世界中，软件发布时，特别是 GNU 这种开源软件的发布时，其 makefile 都包含了编译、安装、打包等功能。我们可以参照这种规则来书写我们的 makefile 中的目标。

“all”

这个伪目标是所有目标的目标，其功能一般是编译所有的目标。

“clean”

这个伪目标功能是删除所有被 make 创建的文件。

“install”

这个伪目标功能是安装已编译好的程序，其实就是把目标执行文件拷贝到指定的目标中去。

“print”

这个伪目标的功能是例出改变过的源文件。

“tar”

这个伪目标功能是把源程序打包备份。也就是一个 tar 文件。

“dist”

这个伪目标功能是创建一个压缩文件，一般是把 tar 文件压成 Z 文件。或是 gz 文件。

“TAGS”

这个伪目标功能是更新所有的目标，以备完整地重编译使用。

“check”和“test”

这两个伪目标一般用来测试 makefile 的流程。

当然一个项目的 makefile 中也不一定要书写这样的目标，这些东西都是 GNU 的东西，但是我想，GNU 搞出这些东西一定有其可取之处（等你的 UNIX 下的程序文件一多时你就会发现这些功能很有用了），这里只不过是说明了，如果你要书写这种功能，最好使用这种名字命名你的目标，这样规范一些，规范的好处就是——不用解释，大家都明白。而且如果你的 makefile 中有这些功能，一是很实用，二是可以显得你的 makefile 很专业（不是那种初学者的作品）。

四、检查规则

有时候，我们不想让我们的 makefile 中的规则执行起来，我们只想检查一下我们的命令，或是执行的序列。于是我们可以使用 make 命令的下述参数：

“-n”

“--just-print”

“--dry-run”

“--recon”

不执行参数，这些参数只是打印命令，不管目标是否更新，把规则和连带规则下的命令打印出来，但不执行，这些参数对于我们调试 makefile 很有用处。

“-t”

“--touch”

这个参数的意思就是把目标文件的时间更新，但不更改目标文件。也就是说，make 假装编译目标，但不是真正的编译目标，只是把目标变成已编译过的状态。

“-q”

“--question”

这个参数的行为是找目标的意思，也就是说，如果目标存在，那么其什么也不会输出，当然也不会执行编译，如果目标不存在，其会打印出一条出错信息。

“-W <file>”

“--what-if=<file>”

“--assume-new=<file>”

“--new-file=<file>”

这个参数需要指定一个文件。一般是是源文件（或依赖文件），Make 会根据规则推导来运行依赖于这个文件的命令，一般来说，可以和“-n”参数一同使用，来查看这个依赖文件所发生的规则命令。

另外一个很有意思的用法是结合“-p”和“-v”来输出 makefile 被执行时的信息（这个将在后面讲述）。

五、make 的参数

下面列举了所有 GNU make 3.80 版的参数定义。其它版本和产商的 make 大同小异，不过其它产商的 make 的具体参数还是请参考各自的产品文档。

“-b”

“-m”

这两个参数的作用是忽略和其它版本 make 的兼容性。

“-B”
“--always-make”
认为所有的目标都需要更新（重编译）。

“-C <dir>”
“--directory=<dir>”
指定读取 makefile 的目录。如果有多个“-C”参数，make 的解释是后面的路径以前面的作为相对路径，并以最后的目录作为被指定目录。如：“make －C ~hchen/test －C prog”等价于“make －C ~hchen/test/prog”。

“—debug[=<options>]”
输出 make 的调试信息。它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。下面是<options>的取值：

a —— 也就是 all，输出所有的调试信息。（会非常的多）

b —— 也就是 basic，只输出简单的调试信息。即输出不需要重编译的目标。

v —— 也就是 verbose，在 b 选项的级别之上。输出的信息包括哪个 makefile 被解析，不需要被重编译的依赖文件（或是依赖目标）等。

i —— 也就是 implicit，输出所以的隐含规则。

j —— 也就是 jobs，输出执行规则中命令的详细信息，如命令的 PID、返回码等。

m —— 也就是 makefile，输出 make 读取 makefile，更新 makefile，执行 makefile 的信息。

“-d”
相当于“--debug=a”。

“-e”
“--environment-overrides”
指明环境变量的值覆盖 makefile 中定义的变量的值。

“-f=<file>”
“--file=<file>”
“--makefile=<file>”
指定需要执行的 makefile。

“-h”
“--help”
显示帮助信息。

“-i”
“--ignore-errors”
在执行时忽略所有的错误。

“-I <dir>”
“--include-dir=<dir>”
指定一个被包含 makefile 的搜索目标。可以使用多个“-I”参数来指定多个目录。

“-j [<jobsnum>]”
“--jobs[=<jobsnum>]”
指同时运行命令的个数。如果没有这个参数，make 运行命令时能运行多少就运行多少。如果有一个以上的“-j”参数，那么仅最后一个“-j”才是有效的。（注意这个参数在 MS-DOS 中是无用的）

“-k”
“--keep-going”
出错也不停止运行。如果生成一个目标失败了，那么依赖于其上的目标就不会被执行了。

“-l <load>”
“--load-average[=<load]”
“—max-load[=<load>]”
指定 make 运行命令的负载。

“-n”
“--just-print”
“--dry-run”
“--recon”
仅输出执行过程中的命令序列，但并不执行。

“-o <file>”
“--old-file=<file>”
“--assume-old=<file>”
不重新生成的指定的<file>，即使这个目标的依赖文件新于它。

“-p”
“--print-data-base”
输出 makefile 中的所有数据，包括所有的规则和变量。这个参数会让一个简单的 makefile 都会输出一堆信息。如果你只是想输出信息而不想执行 makefile，你可以使用“make -qp”命令。如果你想查看执行 makefile 前的预设变量和规则，你可以使用“make －p －f /dev/null”。这个参数输出的信息会包含着你的 makefile 文件的文件名和行号，所以，用这个参数来调试你的 makefile 会是很有用的，特别是当你的环境变量很复杂的时候。

“-q”
“--question”
不运行命令，也不输出。仅仅是检查所指定的目标是否需要更新。如果是 0 则说明要更新，如果是 2 则说明有错误发生。

“-r”
“--no-builtin-rules”
禁止 make 使用任何隐含规则。

“-R”
“--no-builtin-variabes”
禁止 make 使用任何作用于变量上的隐含规则。

“-s”
“--silent”
“--quiet”
在命令运行时不输出命令的输出。

“-S”
“--no-keep-going”
“--stop”
取消“-k”选项的作用。因为有些时候，make 的选项是从环境变量“MAKEFLAGS”中继承下来的。所以你可以在命令行中使用这个参数来让环境变量中的“-k”选项失效。

“-t”
“--touch”
相当于 UNIX 的 touch 命令，只是把目标的修改日期变成最新的，也就是阻止生成目标的命令运行。

“-v”
“--version”
输出 make 程序的版本、版权等关于 make 的信息。

“-w”
“--print-directory”
输出运行 makefile 之前和之后的信息。这个参数对于跟踪嵌套式调用 make 时很有用。

“--no-print-directory”
禁止“-w”选项。

“-W <file>”
“--what-if=<file>”
“--new-file=<file>”
“--assume-file=<file>”
假定目标<file>需要更新，如果和“-n”选项使用，那么这个参数会输出该目标更新时的运行动作。如果没有“-n”那么就像运行 UNIX 的“touch”命令一样，使得<file>的修改时间为当前时间。

“--warn-undefined-variables”
只要 make 发现有未定义的变量，那么就输出警告信息。

隐含规则

在我们使用 Makefile 时，有一些我们会经常使用，而且使用频率非常高的东西，比如，我们编译 C/C++ 的源程序为中间目标文件（Unix 下是[.o]文件，Windows 下是[.obj]文件）。本章讲述的就是一些在 Makefile 中的“隐含的”，早先约定了的，不需要我们再写出来的规则。

“隐含规则”也就是一种惯例，make 会按照这种“惯例”心照不喧地来运行，那怕我们的 Makefile 中没有书写这样的规则。例如，把[.c]文件编译成[.o]文件这一规则，你根本就不用写出来，make 会自动推导出这种规则，并生成我们需要的[.o]文件。

“隐含规则”会使用一些我们系统变量，我们可以改变这些系统变量的值来定制隐含规则的运行时的参数。如系统变量“CFLAGS”可以控制编译时的编译器参数。

我们还可以通过“模式规则”的方式写下自己的隐含规则。用“后缀规则”来定义隐含规则会有许多的限制。使用“模式规则”会更回得智能和清楚，但“后缀规则”可以用来保证我们 Makefile 的兼容性。

我们了解了“隐含规则”，可以让其为我们更好的服务，也会让我们知道一些“约定俗成”了的东西，而不至于使得我们在运行 Makefile 时出现一些我们觉得莫名其妙的东西。当然，任何事物都是矛盾的，水能载舟，亦可覆舟，所以，有时候“隐含规则”也会给我们造成不小的麻烦。只有了解了它，我们才能更好地使用它。

一、使用隐含规则

如果要使用隐含规则生成你需要的目标，你所需要做的就是不要写出这个目标的规则。那么，make 会试图去自动推导产生这个目标的规则和命令，如果 make 可以自动推导生成这个目标的规则和命令，那么这个行为就是隐含规则的自动推导。当然，隐含规则是 make 事先约定好的一些东西。例如，我们有下面的一个 Makefile：

```
foo : foo.o bar.o
	cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

我们可以注意到，这个 Makefile 中并没有写下如何生成 foo.o 和 bar.o 这两目标的规则和命令。因为 make 的“隐含规则”功能会自动为我们自动去推导这两个目标的依赖目标和生成命令。

make 会在自己的“隐含规则”库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，make 调用的隐含规则是，把[.o]的目标的依赖文件置成[.c]，并使用 C 的编译命令“cc -c $(CFLAGS) [.c]”来生成[.o]的目标。也就是说，我们完全没有必要写下下面的两条规则：

```
foo.o : foo.c
	cc -c foo.c $(CFLAGS)
bar.o : bar.c
	cc -c bar.c $(CFLAGS)
```

因为，这已经是“约定”好了的事了，make 和我们约定好了用 C 编译器“cc”生成[.o]文件的规则，这就是隐含规则。

当然，如果我们为[.o]文件书写了自己的规则，那么 make 就不会自动推导并调用隐含规则，它会按照我们写好的规则忠实地执行。

还有，在 make 的“隐含规则库”中，每一条隐含规则都在库中有其顺序，越靠前的则是越被经常使用的，所以，这会导致我们有些时候即使我们显示地指定了目标依赖，make 也不会管。如下面这条规则（没有命令）：

```
foo.o : foo.p
```

依赖文件“foo.p”（Pascal 程序的源文件）有可能变得没有意义。如果目录下存在了“foo.c”文件，那么我们的隐含规则一样会生效，并会通过“foo.c”调用 C 的编译器生成 foo.o 文件。因为，在隐含规则中，Pascal 的规则出现在 C 的规则之后，所以，make 找到可以生成 foo.o 的 C 的规则就不再寻找下一条规则了。如果你确实不希望任何隐含规则推导，那么，你就不要只写出“依赖规则”，而不写命令。

二、隐含规则一览

这里我们将讲述所有预先设置（也就是 make 内建）的隐含规则，如果我们不明确地写下规则，那么，make 就会在这些规则中寻找所需要规则和命令。当然，我们也可以使用 make 的参数“-r”或“--no-builtin-rules”

选项来取消所有的预设置的隐含规则。

当然，即使是我们指定了“-r”参数，某些隐含规则还是会生效，因为有许多的隐含规则都是使用了“后缀规则”来定义的，所以，只要隐含规则中有“后缀列表”（也就一系统定义在目标.SUFFIXES 的依赖目标），那么隐含规则就会生效。默认的后缀列表是：

.out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el。具体的细节，我们会在后面讲述。

还是先来看一看常用的隐含规则吧。

1、编译 C 程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.c”，并且其生成命令是“$(CC) －c $(CPPFLAGS) $(CFLAGS)”

2、编译 C++程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.cc”或是“<n>.C”，并且其生成命令是“$(CXX) －c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为 C++源文件的后缀，而不是“.C”）

3、编译 Pascal 程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.p”，并且其生成命令是“$(PC) －c $(PFLAGS)”。

4、编译 Fortran/Ratfor 程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.r”或“<n>.F”或“<n>.f”，并且其生成命令是:

“.f” “$(FC) –c $(FFLAGS)”

“.F” “$(FC) –c $(FFLAGS) $(CPPFLAGS)”

“.f” “$(FC) –c $(FFLAGS) $(RFLAGS)”

5、预处理 Fortran/Ratfor 程序的隐含规则。

“<n>.f”的目标的依赖目标会自动推导为“<n>.r”或“<n>.F”。这个规则只是转换 Ratfor 或有预处理的 Fortran 程序到一个标准的 Fortran 程序。其使用的命令是：

“.F” “$(FC) –F $(CPPFLAGS) $(FFLAGS)”

“.r” “$(FC) –F $(FFLAGS) $(RFLAGS)”

6、编译 Modula-2 程序的隐含规则。

“<n>.sym”的目标的依赖目标会自动推导为“<n>.def”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(DEFFLAGS)”。“<n.o>” 的目标的依赖目标会自动推导为“<n>.mod”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(MODFLAGS)”。

7、汇编和汇编预处理的隐含规则。

“<n>.o” 的目标的依赖目标会自动推导为“<n>.s”，默认使用编译品“as”，并且其生成命令是：“$(AS) $(ASFLAGS)”。“<n>.s” 的目标的依赖目标会自动推导为“<n>.S”，默认使用 C 预编译器“cpp”，并且其生成命令是：“$(AS) $(ASFLAGS)”。

8、链接 Object 文件的隐含规则。

“<n>”目标依赖于“<n>.o”，通过运行 C 的编译器来运行链接程序生成（一般是“ld”），其生成命令是：“$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)”。这个规则对于只有一个源文件的工程有效，同时也对多个 Object 文件（由不同的源文件生成）的也有效。例如如下规则：

```
x : y.o z.o
```

并且“x.c”、“y.c”和“z.c”都存在时，隐含规则将执行如下命令：

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

如果没有一个源文件（如上例中的 x.c）和你的目标名字（如上例中的 x）相关联，那么，你最好写出自己的生成规则，不然，隐含规则会报错的。

9、Yacc C 程序时的隐含规则。

“<n>.c”的依赖文件被自动推导为“n.y”（Yacc 生成的文件），其生成命令是：“$(YACC) $(YFALGS)”。（“Yacc”是一个语法分析器，关于其细节请查看相关资料）

10、Lex C 程序时的隐含规则。

“<n>.c”的依赖文件被自动推导为“n.l”（Lex 生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。（关于“Lex”的细节请查看相关资料）

11、Lex Ratfor 程序时的隐含规则。

“<n>.r”的依赖文件被自动推导为“n.l”（Lex 生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。

12、从 C 程序、Yacc 文件或 Lex 文件创建 Lint 库的隐含规则。

“<n>.ln”（lint 生成的文件）的依赖文件被自动推导为“n.c”，其生成命令是：“$(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。对于“<n>.y”和“<n>.l”也是同样的规则。

三、隐含规则使用的变量

在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你可以在你的 makefile 中改变这些变量的值，或是在 make 的命令行中传入这些值，或是在你的环境变量中设置这些值，无论怎么样，只要设置了这些特定的变量，那么其就会对隐含规则起作用。当然，你也可以利用 make 的“-R”或“--no－builtin-variables”参数来取消你所定义的变量对隐含规则的作用。

例如，第一条隐含规则——编译 C 程序的隐含规则的命令是“$(CC) －c $(CFLAGS) $(CPPFLAGS)”。Make 默认的编译命令是“cc”，如果你把变量“$(CC)”重定义成“gcc”，把变量“$(CFLAGS)”重定义成“-g”，那么，隐含规则中的命令全部会以“gcc －c -g $(CPPFLAGS)”的样子来执行了。

我们可以把隐含规则中使用的变量分成两种：一种是命令相关的，如“CC”；一种是参数相的关，如“CFLAGS”。下面是所有隐含规则中会用到的变量：

1、关于命令的变量。

AR

函数库打包程序。默认命令是“ar”。

AS

汇编语言编译程序。默认命令是“as”。

CC

C 语言编译程序。默认命令是“cc”。

CXX

C++语言编译程序。默认命令是“g++”。

CO

从 RCS 文件中扩展文件程序。默认命令是“co”。

CPP

C 程序的预处理器（输出是标准输出设备）。默认命令是“$(CC) －E”。

FC

Fortran 和 Ratfor 的编译器和预处理程序。默认命令是“f77”。

GET

从 SCCS 文件中扩展文件的程序。默认命令是“get”。

LEX

Lex 方法分析器程序（针对于 C 或 Ratfor）。默认命令是“lex”。

PC

Pascal 语言编译程序。默认命令是“pc”。

YACC

Yacc 文法分析器（针对于 C 程序）。默认命令是“yacc”。

YACCR

Yacc 文法分析器（针对于 Ratfor 程序）。默认命令是“yacc －r”。

MAKEINFO

转换 Texinfo 源文件（.texi）到 Info 文件程序。默认命令是“makeinfo”。

TEX

从 TeX 源文件创建 TeX DVI 文件的程序。默认命令是“tex”。

TEXI2DVI

从 Texinfo 源文件创建军 TeX DVI 文件的程序。默认命令是“texi2dvi”。

WEAVE

转换 Web 到 TeX 的程序。默认命令是“weave”。

CWEAVE

转换 C Web 到 TeX 的程序。默认命令是“cweave”。

TANGLE

转换 Web 到 Pascal 语言的程序。默认命令是“tangle”。

CTANGLE

转换 C Web 到 C。默认命令是“ctangle”。

RM

删除文件命令。默认命令是“rm －f”。

2、关于命令参数的变量

下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值，那么其默认值都是空。

ARFLAGS

函数库打包程序 AR 命令的参数。默认值是“rv”。

ASFLAGS

汇编语言编译器参数。（当明显地调用“.s”或“.S”文件时）。

CFLAGS

C 语言编译器参数。

CXXFLAGS
C++语言编译器参数。
COFLAGS
RCS 命令参数。
CPPFLAGS
C 预处理器参数。（ C 和 Fortran 编译器也会用到）。
FFLAGS
Fortran 语言编译器参数。
GFLAGS
SCCS “get”程序参数。
LDFLAGS
链接器参数。（如：“ld”）
LFLAGS
Lex 文法分析器参数。
PFLAGS
Pascal 语言编译器参数。
RFLAGS
Ratfor 程序的 Fortran 编译器参数。
YFLAGS
Yacc 文法分析器参数。

四、隐含规则链

有些时候，一个目标可能被一系列的隐含规则所作用。例如，一个[.o]的文件生成，可能会是先被 Yacc 的[.y]文件先成[.c]，然后再被 C 的编译器生成。我们把这一系列的隐含规则叫做“隐含规则链”。

在上面的例子中，如果文件[.c]存在，那么就直接调用 C 的编译器的隐含规则，如果没有[.c]文件，但有一个[.y]文件，那么 Yacc 的隐含规则会被调用，生成[.c]文件，然后，再调用 C 编译的隐含规则最终由[.c]生成[.o]文件，达到目标。

我们把这种[.c]的文件（或是目标），叫做中间目标。不管怎么样，make 会努力自动推导生成目标的一切方法，不管中间目标有多少，其都会执着地把所有的隐含规则和你书写的规则全部合起来分析，努力达到目标，所以，有些时候，可能会让你觉得奇怪，怎么我的目标会这样生成？怎么我的 makefile 发疯了？

在默认情况下，对于中间目标，它和一般的目标有两个地方所不同：第一个不同是除非中间的目标不存在，才会引发中间规则。第二个不同的是，只要目标成功产生，那么，产生最终目标过程中，所产生的中间目标文件会被以“rm -f”删除。

通常，一个被 makefile 指定成目标或是依赖目标的文件不能被当作中介。然而，你可以明显地说明一个文件或是目标是中介目标，你可以使用伪目标“.INTERMEDIATE”来强制声明。（如：.INTERMEDIATE ： mid ）

你也可以阻止 make 自动删除中间目标，要做到这一点，你可以使用伪目标“.SECONDARY”来强制声明（如：.SECONDARY : sec）。你还可以把你的目标，以模式的方式来指定（如：%.o）成伪目标“.PRECIOUS”的依赖目标，以保存被隐含规则所生成的中间文件。

在“隐含规则链”中，禁止同一个目标出现两次或两次以上，这样一来，就可防止在 make 自动推导时出现无限递归的情况。

Make 会优化一些特殊的隐含规则，而不生成中间文件。如，从文件“foo.c”生成目标程序“foo”，按道理，make 会编译生成中间文件“foo.o”，然后链接成“foo”，但在实际情况下，这一动作可以被一条“cc”的

命令完成（cc -o foo foo.c），于是优化过的规则就不会生成中间文件。

五、定义模式规则

你可以使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有"%"字符。"%"的意思是表示一个或多个任意字符。在依赖目标中同样可以使用"%"，只是依赖目标中的"%"的取值，取决于其目标。

有一点需要注意的是，"%"的展开发生在变量和函数的展开之后，变量和函数的展开发生在 make 载入 Makefile 时，而模式规则中的"%"则发生在运行时。

1、模式规则介绍

模式规则中，至少在规则的目标定义中要包含"%"，否则，就是一般的规则。目标中的"%"定义表示对文件名的匹配，"%"表示长度任意的非空字符串。例如："%.c"表示以".c"结尾的文件名（文件名的长度至少为 3），而"s.%.c"则表示以"s."开头，".c"结尾的文件名（文件名的长度至少为 5）。

如果"%"定义在目标中，那么，目标中的"%"的值决定了依赖目标中的"%"的值，也就是说，目标中的模式的"%"决定了依赖目标中"%"的样子。例如有一个模式规则如下：

%.o : %.c ; <command .....>

其含义是，指出了怎么从所有的[.c]文件生成相应的[.o]文件的规则。如果要生成的目标是"a.o b.o"，那么"%c"就是"a.c b.c"。

一旦依赖目标中的"%"模式被确定，那么，make 会被要求去匹配当前目录下所有的文件名，一旦找到，make 就会规则下的命令，所以，在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make 就会产生所有的模式目标，此时，make 关心的是依赖的文件名和生成目标的命令这两件事。

2、模式规则示例

下面这个例子表示了,把所有的[.c]文件都编译成[.o]文件.

```
%.o : %.c
      $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

其中，"$@"表示所有的目标的挨个值，"$<"表示了所有依赖目标的挨个值。这些奇怪的变量我们叫"自动化变量"，后面会详细讲述。

下面的这个例子中有两个目标是模式的：

```
%.tab.c %.tab.h: %.y
      bison -d $<
```

这条规则告诉 make 把所有的[.y]文件都以"bison -d <n>.y"执行，然后生成"<n>.tab.c"和"<n>.tab.h"文件。（其中，"<n>"表示一个任意字符串）。如果我们的执行程序"foo"依赖于文件"parse.tab.o"和"scan.o"，并且文件"scan.o"依赖于文件"parse.tab.h"，如果"parse.y"文件被更新了，那么根据上述的规则，"bison -d parse.y"就会被执行一次，于是，"parse.tab.o"和"scan.o"的依赖文件就齐了。（假设，"parse.tab.o"由"parse.tab.c"生成，和"scan.o"由"scan.c"生成，而"foo"由"parse.tab.o"和"scan.o"链接生成，而且 foo 和其[.o]文件的依赖关系也写好，那么，所有的目标都会得到满足）

3、自动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面，我们已经对自动化变量有所提涉，相信你看到这里已对它有一个感性认识了。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有

的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。

下面是所有的自动化变量及其说明：

$@

表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。

$%

仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix 下是[.a]，Windows 下是[.lib]），那么，其值为空。

$<

依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$?

所有比目标新的依赖目标的集合。以空格分隔。

$^

所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+

这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

$*

这个变量表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是"a.%.b"，那么，"$*"的值就是"dir/a.foo"。这个变量对于构造有关联的文件名是比较有较。如果目标中没有模式的定义，那么"$*"也就不能被推导出，但是，如果目标文件的后缀是 make 所识别的，那么"$*"就是除了后缀的那一部分。例如：如果目标是"foo.c"，因为".c"是 make 所能识别的后缀名，所以，"$*"的值就是"foo"。这个特性是 GNU make 的，很有可能不兼容于其它版本的 make，所以，你应该尽量避免使用"$*"，除非是在隐含规则或是静态模式中。如果目标中的后缀是 make 所不能识别的，那么"$*"就是空值。

当你希望只对更新过的依赖文件进行操作时，"$?"在显式规则中很有用，例如，假设有一个函数库文件叫"lib"，其由其它几个 object 文件更新。那么把 object 文件打包的比较有效率的 Makefile 规则是：

```
lib : foo.o bar.o lose.o win.o
        ar r lib $?
```

在上述所列出来的自动量变量中。四个变量（$@、$<、$%、$*）在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上"D"或"F"字样。这是 GNU make 中老版本的特性，在新版本中，我们使用函数"dir"或"notdir"就可以做到了。"D"的含义就是 Directory，就是目录，"F"的含义就是 File，就是文件。

下面是对于上面的七个变量分别加上"D"或是"F"的含义：

$(@D)

表示"$@"的目录部分（不以斜杠作为结尾），如果"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而如果"$@"中没有包含斜杠的话，其值就是"."（当前目录）。

$(@F)

表示"$@"的文件部分，如果"$@"值是"dir/foo.o"，那么"$(@F)"就是"foo.o"，"$(@F)"相当于函数"$(notdir $@)"。

"$(*D)"

"$(*F)"

和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，"$(*D)"返回"dir"，而"$(*F)"

返回"foo"

"$(%D)"

"$(%F)"

分别表示了函数包文件成员的目录部分和文件部分。这对于形同"archive(member)"形式的目标中的"member"中包含了不同的目录很有用。

"$(<D)"

"$(<F)"

分别表示依赖文件的目录部分和文件部分。

"$(^D)"

"$(^F)"

分别表示所有依赖文件的目录部分和文件部分。（无相同的）

"$(+D)"

"$(+F)"

分别表示所有依赖文件的目录部分和文件部分。（可以有相同的）

"$(?D)"

"$(?F)"

分别表示被更新的依赖文件的目录部分和文件部分。

最后想提醒一下的是，对于"$<"，为了避免产生不必要的麻烦，我们最好给$后面的那个特定字符都加上圆括号，比如，"$(<)"就要比"$<"要好一些。

还得要注意的是，这些变量只使用在规则的命令中，而且一般都是"显式规则"和"静态模式规则"（参见前面"书写规则"一章）。其在隐含规则中并没有意义。

4、模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的"%"，或是没有前后缀，直接就是一个"%"。因为"%"代表一个或多个字符，所以在定义好了的模式中，我们把"%"所匹配的内容叫做"茎"，例如"%.c"所匹配的文件"test.c"中"test"就是"茎"。因为在目标和依赖目标中同时有"%"时，依赖目标的"茎"会传给目标，当做目标中的"茎"。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行"茎"的传递时，我们需要知道这个步骤。例如有一个模式"e%t"，文件"src/eat"匹配于该模式，于是"src/a"就是其"茎"，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式"c%r"，那么，目标就是"src/car"。("茎"被传递）

5、重载内建隐含规则

你可以重载内建的隐含规则（或是定义一个全新的），例如你可以重新构造和内建隐含规则不同的命令，如：

%.o : %.c

```
	$(CC) -c $(CPPFLAGS) $(CFLAGS) -D$(date)
```

你可以取消内建的隐含规则，只要不在后面写命令就行。如：

%.o : %.s

同样，你也可以重新定义一个全新的隐含规则，其在隐含规则中的位置取决于你在哪里写下这个规则。朝前的位置就靠前。

六、老式风格的"后缀规则"

后缀规则是一个比较老式的定义隐含规则的方法。后缀规则会被模式规则逐步地取代。因为模式规则更强更清晰。为了和老版本的 Makefile 兼容，GNU make 同样兼容于这些东西。后缀规则有两种方式："双后缀"和"单后缀"。

双后缀规则定义了一对后缀：目标文件的后缀和依赖目标（源文件）的后缀。如".c.o"相当于"%o : %c"。单后缀规则只定义一个后缀，也就是源文件的后缀。如".c"相当于"% : %.c"。

后缀规则中所定义的后缀应该是 make 所认识的，如果一个后缀是 make 所认识的，那么这个规则就是单后缀规则，而如果两个连在一起的后缀都被 make 所认识，那就是双后缀规则。例如：".c"和".o"都是 make 所知道。因而，如果你定义了一个规则是".c.o"那么其就是双后缀规则，意义就是".c"是源文件的后缀，".o"是目标文件的后缀。如下示例：

```
.c.o:
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

后缀规则不允许任何的依赖文件，如果有依赖文件的话，那就不是后缀规则，那些后缀统统被认为是文件名，如：

```
.c.o: foo.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

这个例子，就是说，文件".c.o"依赖于文件"foo.h"，而不是我们想要的这样：

```
%.o: %.c foo.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

后缀规则中，如果没有命令，那是毫无意义的。因为他也不会移去内建的隐含规则。

而要让 make 知道一些特定的后缀，我们可以使用伪目标".SUFFIXES"来定义或是删除，如：

```
.SUFFIXES: .hack .win
```

把后缀.hack 和.win 加入后缀列表中的末尾。

```
.SUFFIXES:            # 删除默认的后缀
.SUFFIXES: .c .o .h   # 定义自己的后缀
```

先清楚默认后缀，后定义自己的后缀列表。

make 的参数"-r"或"-no-builtin-rules"也会使用得默认的后缀列表为空。而变量"SUFFIXE"被用来定义默认的后缀列表，你可以用".SUFFIXES"来改变后缀列表，但请不要改变变量"SUFFIXE"的值。

七、隐含规则搜索算法

比如我们有一个目标叫 T。下面是搜索目标 T 的规则的算法。请注意，在下面，我们没有提到后缀规则，原因是，所有的后缀规则在 Makefile 被载入内存时，会被转换成模式规则。如果目标是"archive(member)"的函数库文件模式，那么这个算法会被运行两次，第一次是找目标 T，如果没有找到的话，那么进入第二次，第二次会把"member"当作 T 来搜索。

1、把 T 的目录部分分离出来。叫 D，而剩余部分叫 N。（如：如果 T 是"src/foo.o"，那么，D 就是"src/"，N 就是"foo.o"）
2、创建所有匹配于 T 或是 N 的模式规则列表。
3、如果在模式规则列表中有匹配所有文件的模式，如"%"，那么从列表中移除其它的模式。
4、移除列表中没有命令的规则。
5、对于第一个在列表中的模式规则：

1）推导其"茎"S，S 应该是 T 或是 N 匹配于模式中"%"非空的部分。

2）计算依赖文件。把依赖文件中的"%"都替换成"茎"S。如果目标模式中没有包含斜框字符，而把 D 加在

第一个依赖文件的开头。

3）测试是否所有的依赖文件都存在或是理当存在。（如果有一个文件被定义成另外一个规则的目标文件，或者是一个显式规则的依赖文件，那么这个文件就叫"理当存在"）

4）如果所有的依赖文件存在或是理当存在，或是就没有依赖文件。那么这条规则将被采用，退出该算法。

6、如果经过第 5 步，没有模式规则被找到，那么就做更进一步的搜索。对于存在于列表中的第一个模式规则：

1）如果规则是终止规则，那就忽略它，继续下一条模式规则。

2）计算依赖文件。（同第 5 步）

3）测试所有的依赖文件是否存在或是理当存在。

4）对于不存在的依赖文件，递归调用这个算法查找他是否可以被隐含规则找到。

5）如果所有的依赖文件存在或是理当存在，或是就根本没有依赖文件。那么这条规则被采用，退出该算法。

7、如果没有隐含规则可以使用，查看".DEFAULT"规则，如果有，采用，把".DEFAULT"的命令给 T 使用。一旦规则被找到，就会执行其相当的命令，而此时，我们的自动化变量的值才会生成。

使用 make 更新函数库文件

函数库文件也就是对 Object 文件（程序编译的中间文件）的打包文件。在 Unix 下，一般是由命令"ar"来完成打包工作。

一、函数库文件的成员

一个函数库文件由多个文件组成。你可以以如下格式指定函数库文件及其组成：

```
archive(member)
```

这个不是一个命令，而一个目标和依赖的定义。一般来说，这种用法基本上就是为了"ar"命令来服务的。如：

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

如果要指定多个 member，那就以空格分开，如：

```
foolib(hack.o kludge.o)
```

其等价于：

```
foolib(hack.o) foolib(kludge.o)
```

你还可以使用 Shell 的文件通配符来定义，如：

```
foolib(*.o)
```

二、函数库成员的隐含规则

当 make 搜索一个目标的隐含规则时，一个特殊的特性是，如果这个目标是"a(m)"形式的，其会把目标变成"(m)"。于是，如果我们的成员是"%.o"的模式定义，并且如果我们使用"make foo.a(bar.o)"的形式调用 Makefile 时，隐含规则会去找"bar.o"的规则，如果没有定义 bar.o 的规则，那么内建隐含规则生效，make 会去找 bar.c 文件来生成 bar.o，如果找得到的话，make 执行的命令大致如下：

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
```

rm -f bar.o

还有一个变量要注意的是"$%"，这是专属函数库文件的自动化变量，有关其说明请参见"自动化变量"一节。

三、函数库文件的后缀规则

你可以使用"后缀规则"和"隐含规则"来生成函数库打包文件，如：

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

其等效于：

```
(%.o) : %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

四、注意事项

在进行函数库打包文件生成时，请小心使用 make 的并行机制（"-j"参数）。如果多个 ar 命令在同一时间运行在同一个函数库打包文件上，就很有可以损坏这个函数库文件。所以，在 make 未来的版本中，应该提供一种机制来避免并行操作发生在函数打包文件上。
但就目前而言，你还是应该不要尽量不要使用"-j"参数。

后序

终于到写结束语的时候了，以上基本上就是 GNU make 的 Makefile 的所有细节了。其它的产商的 make 基本上也就是这样的，无论什么样的 make，都是以文件的依赖性为基础的，其基本是都是遵循一个标准的。这篇文档中 80%的技术细节都适用于任何的 make，我猜测"函数"那一章的内容可能不是其它 make 所支持的，而隐含规则方面，我想不同的 make 会有不同的实现，我没有精力来查看 GNU 的 make 和 VC 的 nmake、BCB 的 make，或是别的 UNIX 下的 make 有些什么样的差别，一是时间精力不够，二是因为我基本上都是在 Unix 下使用 make，以前在 SCO Unix 和 IBM 的 AIX，现在在 Linux、Solaris、HP-UX、AIX 和 Alpha 下使用，Linux 和 Solaris 下更多一点。不过，我可以肯定的是，在 Unix 下的 make，无论是哪种平台，几乎都使用了 Richard Stallman 开发的 make 和 cc/gcc 的编译器，而且，基本上都是 GNU 的 make（公司里所有的 UNIX 机器上都被装上了 GNU 的东西，所以，使用 GNU 的程序也就多了一些）。GNU 的东西还是很不错的，特别是使用得深了以后，越来越觉得 GNU 的软件的强大，也越来越觉得 GNU 的在操作系统中（主要是 Unix，甚至 Windows）"杀伤力"。

对于上述所有的 make 的细节，我们不但可以利用 make 这个工具来编译我们的程序，还可以利用 make 来完成其它的工作，因为规则中的命令可以是任何 Shell 之下的命令，所以，在 Unix 下，你不一定只是使用程序语言的编译器，你还可以在 Makefile 中书写其它的命令，如：tar、awk、mail、sed、cvs、compress、ls、rm、yacc、rpm、ftp……等等，等等，来完成诸如"程序打包"、"程序备份"、"制作程序安装包"、"提交代码"、"使用程序模板"、"合并文件"等等五花八门的功能，文件操作，文件管理，编程开发设计，或是其它一些异想天开的东西。比如，以前在书写银行交易程序时，由于银行的交易程序基本一样，就见到有人书写了一些交易的

通用程序模板，在该模板中把一些网络通讯、数据库操作的、业务操作共性的东西写在一个文件中，在这些文件中用些诸如"@@@N、###N"奇怪字串标注一些位置，然后书写交易时，只需按照一种特定的规则书写特定的处理，最后在 make 时，使用 awk 和 sed，把模板中的"@@@N、###N"等字串替代成特定的程序，形成 C 文件，然后再编译。这个动作很像数据库的"扩展 C"语言（即在 C 语言中用"EXEC SQL"的样子执行 SQL 语句，在用 cc/gcc 编译之前，需要使用"扩展 C"的翻译程序，如 cpre，把其翻译成标准 C）。如果你在使用 make 时有一些更为绝妙的方法，请记得告诉我啊。

回头看看整篇文档，不觉记起几年前刚刚开始在 Unix 下做开发的时候，有人问我会不会写 Makefile 时，我两眼发直，根本不知道在说什么。一开始看到别人在 vi 中写完程序后输入"!make"时，还以为是 vi 的功能，后来才知道有一个 Makefile 在作怪，于是上网查啊查，那时又不愿意看英文，发现就根本没有中文的文档介绍 Makefile，只得看别人写的 Makefile，自己瞎碰瞎搞才积累了一点知识，但在很多地方完全是知其然不知所以然。后来开始从事 UNIX 下产品软件的开发，看到一个 400 人年，近 200 万行代码的大工程，发现要编译这样一个庞然大物，如果没有 Makefile，那会是多么恐怖的一样事啊。于是横下心来，狠命地读了一堆英文文档，才觉得对其掌握了。但发现目前网上对 Makefile 介绍的文章还是少得那么的可怜，所以想写这样一篇文章，共享给大家，希望能对各位有所帮助。

现在我终于写完了，看了看文件的创建时间，这篇技术文档也写了两个多月了。发现，自己知道是一回事，要写下来，跟别人讲述又是另外一回事，而且，现在越来越没有时间专研技术细节，所以在写作时，发现在阐述一些细节问题时很难做到严谨和精练，而且对先讲什么后讲什么不是很清楚，所以，还是参考了一些国外站点上的资料和题纲，以及一些技术书籍的语言风格，才得以完成。整篇文档的提纲是基于 GNU 的 Makefile 技术手册的提纲来书写的，并结合了自己的工作经验，以及自己的学习历程。因为从来没有写过这么长，这么细的文档，所以一定会有很多地方存在表达问题，语言歧义或是错误。因些，我迫切地得等待各位给我指证和建议，以及任何的反馈。

最后，还是利用这个后序，介绍一下自己。我目前从事于所有 Unix 平台下的软件研发，主要是做分布式计算/网格计算方面的系统产品软件，并且我对于下一代的计算机革命——网格计算非常地感兴趣，对于分布式计算、P2P、Web Service、J2EE 技术方向也很感兴趣，同时，对于项目实施、团队管理、项目管理也小有心得，希望同样和我战斗在“技术和管理并重”的阵线上的年轻一代，能够和我多多地交流。我的 MSN 是：haoel@hotmail.com（常用），QQ 是：753640（不常用）。（注：请勿给我 MSN 的邮箱发信，由于 hotmail 的垃圾邮件导致我拒收这个邮箱的所有来信）

我欢迎任何形式的交流，无论是讨论技术还是管理，或是其它海阔天空的东西。除了政治和娱乐新闻我不关心，其它只要积极向上的东西我都欢迎！

最最后，我还想介绍一下 make 程序的设计开发者。

首当其冲的是： **Richard Stallman**

开源软件的领袖和先驱，从来没有领过一天工资，从来没有使用过 Windows 操作系统。对于他的事迹和他的软件以及他的思想，我无需说过多的话，相信大家对这个人并不比我陌生，这是他的主页：http://www.stallman.org/ 。这里只贴上一张他的近照：

计算机、音乐、蝴蝶就是他的最爱

第二位是：Roland McGrath

个人主页是：http://www.frob.com/~roland/ ，下面是他的一些事迹：

1） 合作编写了并维护 GNU make。

2） 和 Thomas Bushnell 一同编写了 GNU Hurd。

3） 编写并维护着 GNU C library。

4） 合作编写并维护着部分的 GNU Emacs。

在此，向这两位开源项目的斗士致以最真切的敬意。

（全文完）

【责任编辑：worldguy@hitbbs】

论坛视点

纯 C 论坛网友

编者按：“论坛视点”来自“纯 C 论坛”网站，或为独到见解，或为技术讨论。本期选取的几则有启发性的帖子，以飨读者。欢迎各位朋友提意见或建议。

贴子主题：全新的操作系统概念

楼主

请问大家能不能想象一种操作系统没有进程，也没有文件系统，应用程序不是指令的序列，而是数据、代码和时间片的集合，磁盘上数据与代码的封装。

nydgg

楼主所说的这种操作系统其实早已存在！只是大家没有理解楼主的意思。让我来解释一下先：

首先来重提一下进程的定义：进程是程序在某一数据集上的一次执行，是一个动态的过程。楼主的操作系统中不存在进程这个东西，但是楼主的操作系统中有程序和数据的概念，根据进程的定义可知楼主的操作系统与其它操作的本质区别在于：在楼主的操作系统中，程序并不执行！由此可知，楼主的操作系统是指存放在磁盘上的，不被执行的代码和数据的集合。

另外大家需要注意的是，楼主的操作系统中并无文件系统，这个概念可以这样理解：文件系统是用于存取存和管理储设备上可计算机表示的数据的，也就是说，在楼主的操作系统里，并没有提供对数据进行管理和存取的机制。即，楼主的操作系统是保存在存放在磁盘上的，不可被存取的，也不被执行的代码和数据的集合。

还有，楼主的操作系统里有时间片这个概念。时间片即一段规定的时间长度，由此我不们难得出楼主的操作系统的最终定义：存放在磁盘上的，经过一定时间长度的，不可被存取的，也不会执行的代码和数据集合。

事情至此已豁然开朗，楼主的操作系统是什么？—— 一块装了 Win98 的陈年报废硬盘耳。

这种操作系统我早就有了，诸位兄弟都有吗？

wkf

你这个思路的应该不仅仅局限于 OS 这个领域，它应该是涉及到编程模型，OS,编译,计算机体系结构多个领域的东东。简单地说，就是我们现在普遍使用的编程模型有些缺陷！我们都知道，程序被汇编成汇编语言的时候，语句间就订好了先后顺序，不同的指令间可能有数据相关或控制相关，也可能完全无关。目前冯式结构的计算机运行时需要 PC 来指定下一个指令地址的，所以，就限制了一些即使完全无关的指令也必须按照先后顺序来运行（严格一点地讲，现在的超标量处理器和 EPIC 之类的编译器能够在小范围的解决这一限制）。但是实际上，保证程序正确运行的最小约束条件是什么呢？应该是是维护一致的数据流关系。也就是说，目前的这种程序指令被事先定义成具有先后顺序的方法其实有点保守，正确的编程方法应该是完全面向数据流的编程。就像楼主所讲，把一个程序分别写成很多个小的线程，每个线程维护一个数据流，线程间通信可能要用某种方法减到最小而使得性能不至于由于同步之类的开销而降低很多。

基于数据流的研究很早以前就有人提出，像数据流机这个词可能大家都听说过。后来因为想法太创新，当时半导体制造业无法满足他的要求，而目前人们现在还不知道如何用数据流进行高效的并行编程。有的人在研究打破现有的串行编程模型，有的人在研究改进编译器，让用户在不知道的情况下，把他写的串行程序编译成并行的数据流程序，还有的人直接研究新型处理器结构。像德州大学奥斯汀分校正在进行的 TRIPS 项目，他的思路就非常有创意，主要思想就是只把生产者和消费者指令分到一堆儿，每堆里没有其他多余的指令，每一小堆指令维持了一组或几组数据流关系（可以认为是小线索），然后，又设计了一些线索间通信的硬件结构在 CPU 级上直接支持线索间通信。这个项目涉猎众多：处理器设计，编译器设计，OS 设计和应用程序的设计。可以说，把整个计算机基础结构翻了底朝天。Doug Burger 也明确地说过，冯式结构有些过时，人们现在需要更新的思想！

原文地址：
http://purec.binghua.com/bbs/dispbbs.asp?boardid=8&star=1&replyid=3567&id=722&skin=0&page=1
欢迎参与讨论！

投稿指南

《纯 C 论坛杂志》编辑部成立于公元 2004 年 9 月 28 日。2005 年 1 月开始，杂志更名为《CSDN 社区电子杂志——纯 C 论坛杂志》。新的一年，新的开始，我们有幸和 CSDN 社区电子杂志的编辑一同合作，一同打造属于 CS 人的杂志。本刊定位为相对底层、较为纯粹的计算机科学及技术的研究，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，希望以此形成本刊独特的技术风格。

本刊目前为双月刊，于每单月 28 号通过网络发行。任何人均可从 **CSDN 电子社区电子杂志**(http://emag.csdn.net)或**纯 C 论坛网站**(http://purec.binghua.com)下载本刊。读者不需要付费。

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@126.com
编译原理	worldguy@hitbbs	
算法理论与数据结构	xiong@hitbbs	
计算机语言（C/C++）	sun@hitbbs hitool@hitbbs	
汇编语言	ogg@hitbbs	
数据库原理	pineapple@hitbbs	
网络与信息安全	true@hitbbs	
计算机病毒	swordlea@hitbbs	
人工智能及信息处理	car@hitbbs	
操作系统	iamxiaohan@hitbbs	

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本人所有。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，投稿请寄本刊通用联系信箱（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，版权归作者本人所有。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）或者 CSDN 社区电子杂志(http://emag.csdn.net)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。是否提供源代码，由作者决定。

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。

本刊编辑部联系地址：

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！

《CSDN 社区电子杂志——纯 C 论坛杂志》编辑部
2005.1.28 修订

2005 年 1 月号 勘误表

第 89 页

正文第 **10** 行：“下面就是我们对上面一段语句的返汇编结果：……”中应为“反汇编”

[感谢 **CrazyWind @ PureC BBS]**

第 33 页

在中**"3.5** 内存印射**"**应改为**"3.5** 内存映射**"**

[感谢 **TOTO @ CSDN]**

第 89 页 补充说明

“对《浅析 **C** 语言函数传递机制及对变参函数的处理》的一点更正”文中对浮点数参数传递的描述适用于编译时没有进行优化的情况；对于优化的情况，常数参数或者结果为常数的表达式参数传递还是直接进行 **push** 的。另外，第一期杂志中“剖析 **Intel IA32** 架构下 **C** 语言及 **CPU** 浮点数机制”一文中关于浮点数直接比较的问题，其实只要正确设置浮点数精度标志位（设置精度为 **53** 位），**27** 页上面的两个程序的结果是完全一样的。

[感谢 **hit-007 @ PureC BBS]**