

纯C论坛杂志

2005年1月 第1期 总第3期



用GDB调试程序
命令行计算器的实现
深度探索编译器安全检查
推箱子游戏的自动求解

HPC
哈工大 纯 C/C++ 论坛
HIT Pure C/C++ Forum

目录

卷首语

信息时代的挑战与机遇

车万翔

操作系统

《操作系统概念》第二章 计算机系统结构

吕建鹏(译)

Linux 核心第 3 章 内存管理

毕昕等(译)

技术资料

用 GDB 调试程序

陈皓

编译原理

命令行计算器的实现

高立琦

C/C++

标准 C++ 类 std::string 的内存共享和 Copy-On-Write 技术

陈皓

深度探索编译器安全检查

杨新开(译)

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

谢煜波

算法

推箱子游戏的自动求解

hellwolf

编辑部通讯

投稿指南

编辑部

信息时代的挑战与机遇

哈尔滨工业大学 计算机科学与技术学院

车万翔(car@ir.hit.edu.cn)



欣慰地看到 2005 年第一期的《CSDN 社区电子杂志——纯 C 论坛杂志》如期而至，也很高兴有此机会在这里和各位道一声新年快乐！人们总是喜欢在这一特殊的日子回顾过去，展望未来。我是个俗人，当然也难免落此俗套。

回顾过去的百年，人们会情不自禁的想起许多科学技术上的伟大发明、发现，如内燃机、青霉素，新材料等等。然而，计算机的发明，特别是近十年互联网的蓬勃发展，更将二十一世纪定义成为“信息时代”。

我时常有“如果没有网络，世界将会怎样？”的疑问。想想互联网能为我们做什么吧！但我觉得如果换一种问法会更容易回答：互联网不能为我们做什么？答案是除了洗衣做饭②但是谁知道将来能不能呢？

刚刚过去的 2004 年，更是互联网应用全面发展的一年，尤其是以 Google 为代表的信息检索公司引发的搜索革命使人们相信，2004 年=“搜索年”。Google 的目标是让获得一切数字化信息都只需要举手之劳，雅虎和微软现在也加入了这一任务，越来越多的小企业也在加入进来。我们有理由相信，2005 年必将是信息检索应用全面爆发的一年。由此带给人们的是信息交换的便捷，同时也是人类思维的一种延伸。

然而，检索技术只是对信息的一种最基本的处理，人类的目标当然不仅仅局限于此。随着互联网的快速发展与广泛应用，人们越来越要求在开放、动态环境下实现灵活的、可信的、协同的、深层次的知识共享和利用，基于互联网的知识处理才是人们真正的梦想。互联网上的知识大部分是非结构或半结构的，它们以各种媒体形式存在，分布在数以亿计的网页上，每天以百万网页的数量级在增长、消失或改变内容，它充满了各种矛盾的事实、数据和观点。这些给知识处理带来了巨大的挑战，同时也是千载难逢的机遇。相信经过人们不懈的努力，互联网必将以其独特的魅力，深刻地影响和改变人类的发展历程和生活方式。

谨以此文献给已经、正在和即将为信息时代的发展做出贡献的人们！

车万翔，1980 年出生，哈尔滨工业大学计算机学院信息检索研究室(<http://ir.hit.edu.cn>)
博士研究生。研究方向为自然语言处理、信息检索。个人主页 <http://ir.hit.edu.cn/~car>

操作系统概念（第六版）

第二章 计算机系统结构

原著：Abraham Silberschatz, Peter Baer Galvin, GregGagne
翻译：吕建鹏(webmaster@tulipsys.com)

在研究计算机系统运行的细节之前，需要对计算机系统结构有一个总体的认识。我们将在本章中学习这个体系结构中的几个完全不同的部分，以此来完善我们的背景知识。本章主要关注计算机体系结构，所以，如果已经掌握了这些概念，你就可以略读或者跳过本章。最初的课题包括了系统启动、I/O 和存储器。

操作系统也必须确保计算机系统的正确运行。为确保用户程序不会干扰系统的正常运行，硬件必须要提供合适的机制来保证操作的正确性。在本章的后面，我们将描述基本的计算机体系结构，这是设计可用操作系统的基础知识。最后，我们对网络结构做一个概述。

1.1 计算机系统的运行

一个现代通用计算机系统由一个 CPU 和多个设备控制器组成，它们通过一条公共总线连接到一起，而这条公共总线提供了对共享存储器的访问能力（图 2.1）。每个设备控制器负责某种特定类型的设备（如磁盘驱动器、音频设备和视频显示器）。CPU 和设备控制器能够同时运行，并且相互竞争总线周期。为确保对共享存储器访问的有序性，需要提供一个存储控制器以同步对存储器的访问。

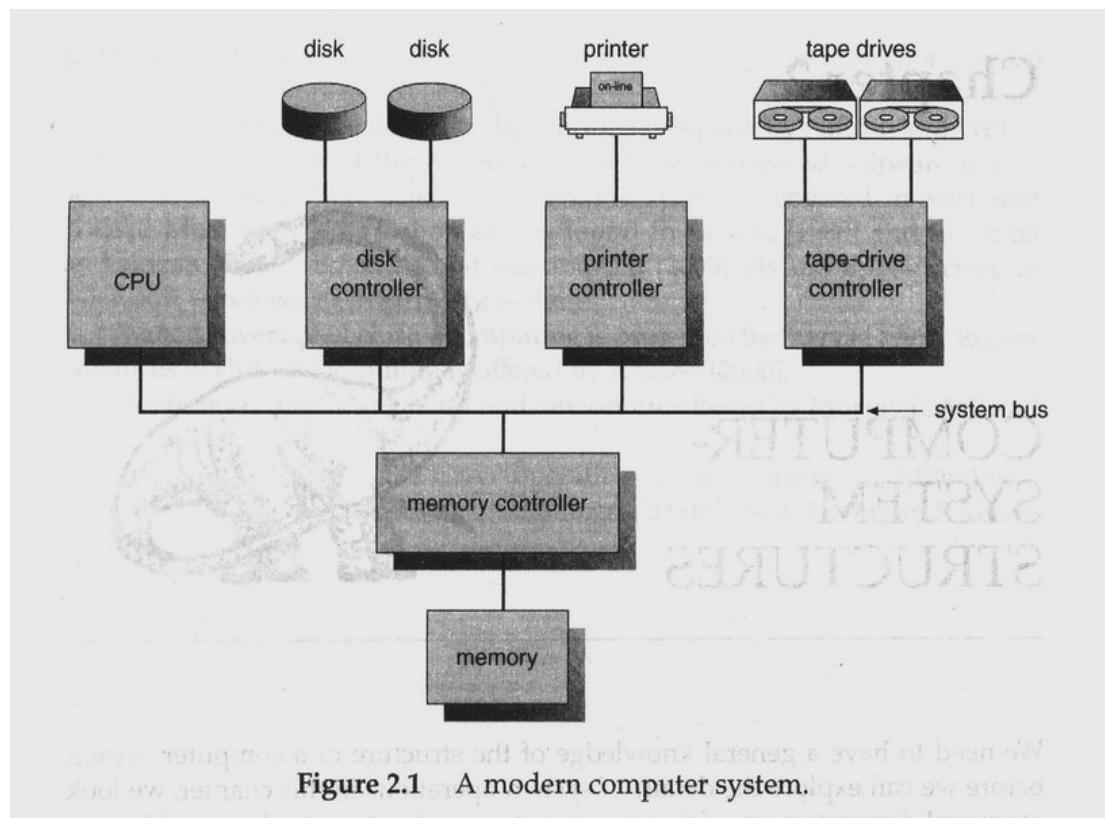


Figure 2.1 A modern computer system.

计算机开始运行（如开启电源或者重新启动）时需要首先运行一个**初始化程序**。这个初始化程序（或者说**引导程序**）往往很简单。它通常存储在计算机硬件中的只读存储器（如固件或 EEPROM）内。从 CPU 寄存器到设备控制器再到内存，引导程序初始化系统的各个方面。引导程序必须要知道如何装入操作系统并开始运行它。因此，引导程序必须要为操作系统内核分配内存空间并将其装入内存。操作系统此时才开始运行第一个进程（比如：“init”），然后等待事件的发生。

事件通常由硬件或软件**中断**触发产生。硬件随时会通过系统总线向 CPU 发送信号的方式触发一个中断。软件可能会运行一个特殊的操作触发一个中断，这个特殊的操作被称为**系统调用**（也称之为**监督程序调用**）。

现代操作系统是**中断驱动的**。如果没有进程运行、没有 I/O 设备运行并且没有用户响应，操作系统将停下来等待事件的发生。事件几乎总是通过中断或自陷发出信号产生。**自陷**（也称为**异常**）是一种由软件产生的中断，它由错误（如除以零或无效内存访问）或用户程序请求操作系统执行特殊的服务引起。操作系统的中断驱动的特性定义了系统的一般架构。针对每种类型的中断，操作系统中独立的代码段定义了应该执行什么样的操作。操作系统提供了中断服务程序，由它来负责处理中断。

当 CPU 接收到中断信号时，它会停止当前的工作并立即转向一个确定地点。这个地点通常存储了该中断服务程序的入口地址；处理完成后，CPU 恢复被中断的计算。图 2.2 表明了该操作的时序关系。

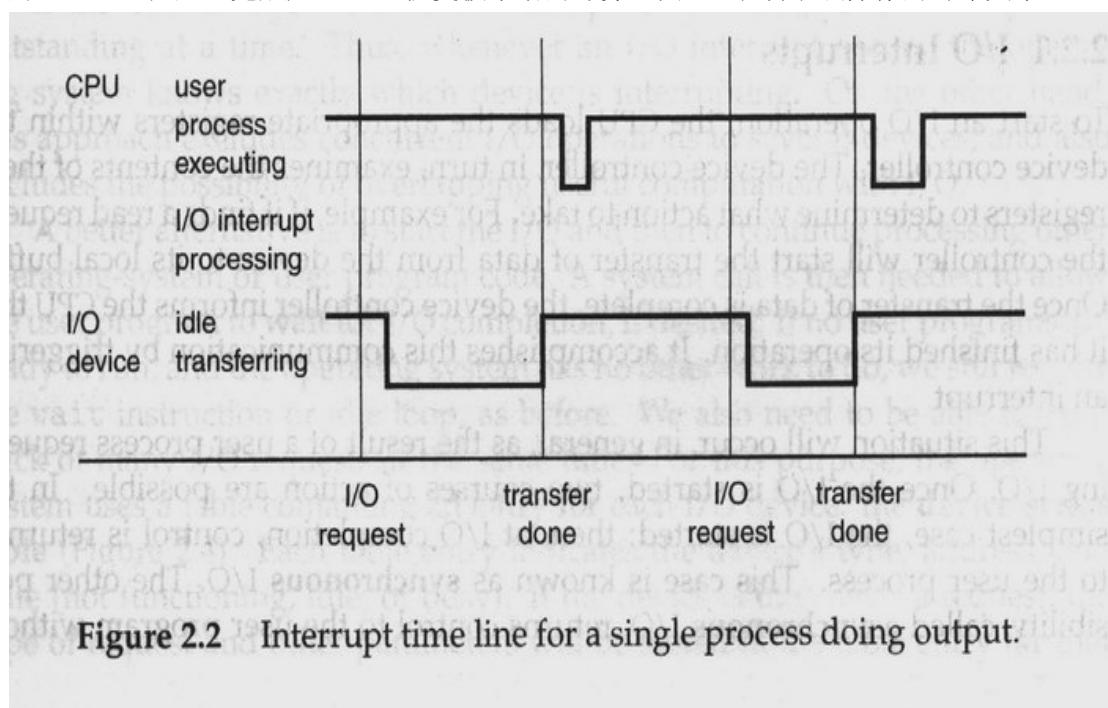


Figure 2.2 Interrupt time line for a single process doing output.

Figure 2.2 Interrupt time line for a single process doing output.

中断是计算机体系结构中的重要组成部分。每种计算机都有自己的中断机制，但是有些功能是共同的。中断必须要将控制移交给适当的中断服务程序。一个简单的方法是调用一个通用程序来检查中断信息；然后再调用具体的中断服务处理程序。然而，中断必须要得到快速处理，预定义中断数目是可行的，这样就可以使用一个指向中断处理程序的中断向量表。于是通过这个表间接调用中断程序，就不再需要中间程序了。通常，中断向量表存储在内存的低字节（前 100 位，大致如此）。这些位置存储了各种设备的中断服务进程地址。这个地址队列（或者说是**中断向量**）指向由中断请求给定的唯一的设备号，并向中断设备提供中断服务程序地址。MS-DOS 和 UNIX 在中断的工作方式上有所不同。

中断体系结构还必须保存被中断的指令的地址。许多老式的设计简单的将中断地址存储在一个确定地点或由设备号索引的地点。更新近的体系结构将返回地址存储在系统堆栈中。如果中断处理程序需要改变处理器状态（如通过修改寄存器值），它必须要显式的保存当前状态，然后在返回之前将其还原。在中断服务结束后，存储的返回地址将被装载到程序计数器中，此时被中断的计算重新开始，就像是中断没有发生过一样。

依据底层处理器提供的功能，有多种方式可以请求系统调用。不管用的是什么方式，它是进程请求操作系统服务的方法。系统调用往往采用自陷到中断向量指定地点的方式。通常可以执行通用的 trap 指令来产生一个自陷，而有些系统（如 MIPS R2000 家族）有一个专门的 syscall 指令。

1.2 I/O 结构

正如在第 2.1 节中所描述的，一个现代的通用计算机系统由一个 CPU 和多个设备控制器组成，这些设备控制器连接到一条公共总线上。每个设备控制器负责某种特定类型的设备。可能会连接有多个附属设备，这取决于于控制器。例如，小型计算机系统接口 (SCSI) 控制器具备七个或更多的设备挂载能力。一个设备控制器上有本地缓冲存储器和一系列特定用途的寄存器。设备控制器负责在它控制的外围设备和本地缓冲存储器之间传送数据。根据所控制设备的不同，设备控制器中本地缓冲器的大小也不尽相同。例如：磁盘控制器中缓存的大小应该等于磁盘最小的可寻址空间的大小（也就是一个扇区，通常是 512 字节），或者是它的倍数。

1.2.1 输入输出中断

要开始一个 I/O 操作，CPU 首先要给设备控制器中相应的寄存器赋值。然后设备控制器检查这些寄存器的内容以决定下一步的动作。例如，如果设备控制器发现一个读请求，它就开始把数据从设备传送到本地缓冲器中。一旦数据传输结束，设备控制器就会通知 CPU 该操作已经结束。通过触发一个中断来完成这种通信。

通常，在用户进程请求 I/O 时就会发生这种情况。一旦 I/O 操作开始，就会有两种可能的情况。最简单的一种，I/O 操作开始，然后，在 I/O 操作完成时将控制返回给用户进程。这种被认为是同步输入输出。另外一种可能被称之为异步输入输出，它并不等待输入输出结束，而是直接将控制返回给用户程序。于是，当其它的系统操作运行时输入输出可以继续进行（图 2.3）。

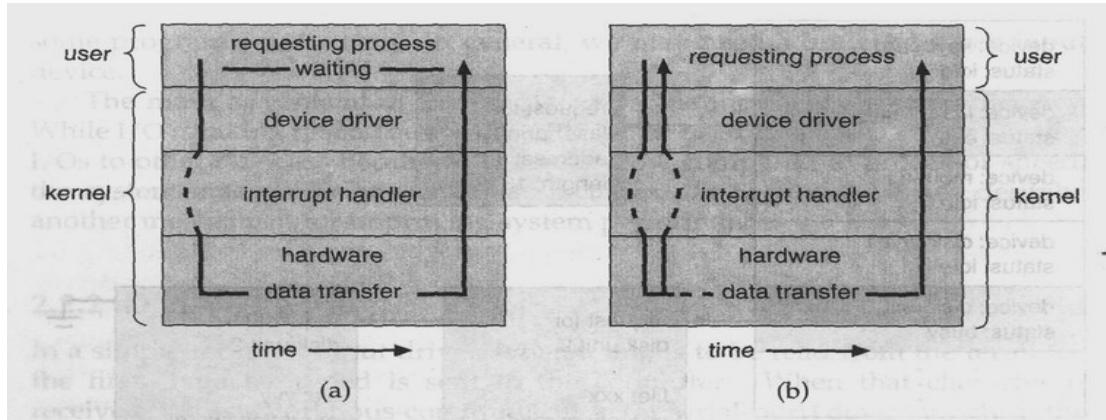


Figure 2.3 Two I/O methods: (a) synchronous, and (b) asynchronous.

Figure 2.3 Two I/O methods: (a) synchronous, and (b) asynchronous.

可以有两种方式来等待 I/O 操作的完成。有些计算机有一个特殊的 wait 指令，将 CPU 置于空闲等待状态直到产生下一个中断。没有这种指令的机器可能会有一个等待循环：

Loop: jmp Loop

这种死循环简单的持续运行直到发生一个中断，然后将控制提交给操作系统的另一个部分。这样的循环也可能需要轮流查询多个不支持中断结构的设备；这些设备简单的设置它们寄存器中的一个标志位，然后等待操作系统查询这个标志。

如果 CPU 总是等待 I/O 结束，那么它一次最多只能响应一个 I/O 请求。因此，不论一个 I/O 中断何时发生，

操作系统总是能够准确的知道正在响应中断的具体设备。从另一方面讲，这种方法不允许多个设备的并行 I/O 操作，也不允许对 I/O 的重叠操作。

一种更好的方法是开始 I/O 操作，然后继续处理其它的操作系统代码或用户程序代码。这就需要一个系统调用允许用户程序在需要的时候等待 I/O 完成。如果没有用户程序准备好，且操作系统无事可做，那么我们还是像以前那样使用 wait 指令或空闲循环。我们还需要具备同时监控多个 I/O 请求的能力。为此，操作系统使用一个表来为每一个 I/O 设备建立一个表项，这就是**设备状态表**（图 2.4）。每个表项指明了设备的类型、地址和状态（不是功能、空闲或忙碌）。如果这个设备正在处理一个请求，那么请求类型和其它的参数将存储在该设备所对应的表项中。因为也可能同时有其它的进程请求同样的设备，所以操作系统也要为每个 I/O 设备维护一个等待队列（一个等待的请求列表）。

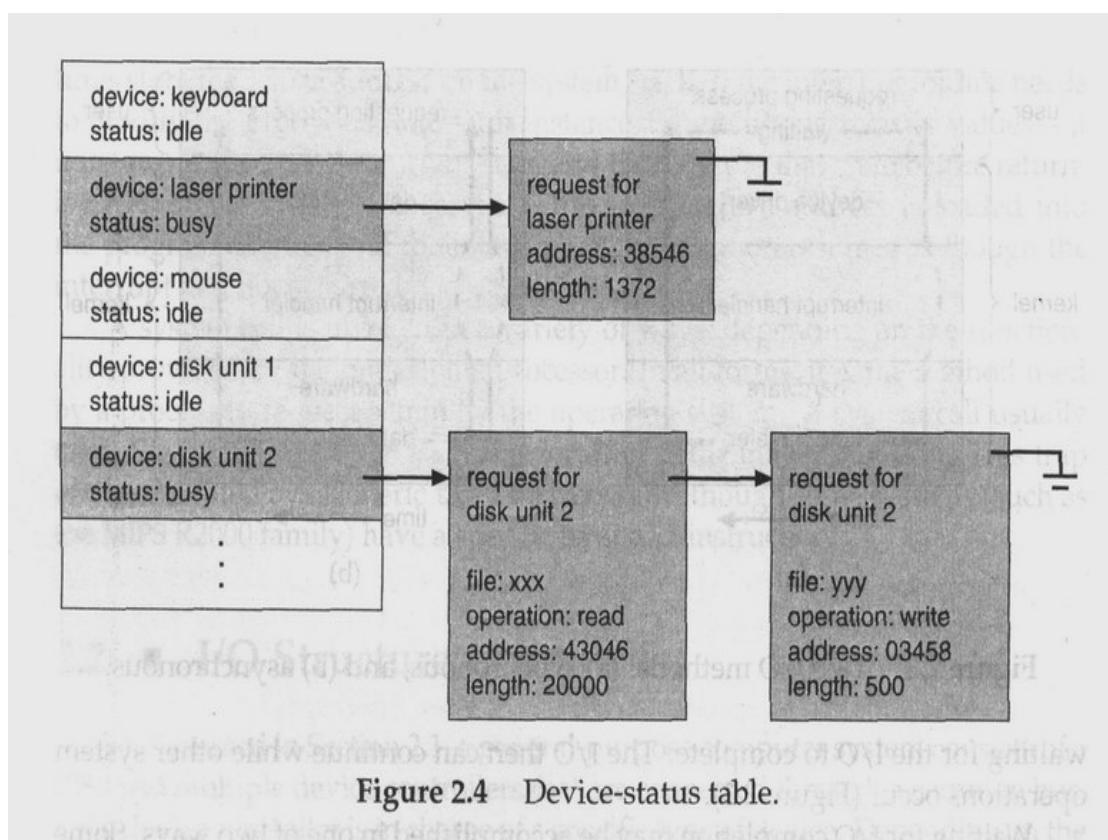


Figure 2.4 Device-status table.

Figure 2.4 Device-status table.

一个 I/O 设备需要服务时它会触发一个中断。当发生一个中断时，操作系统首先判断是哪个 I/O 设备发出了中断。然后查询 I/O 设备表确定其状态，并且修改相应表项来反映该中断。对于大多数设备来说，一个中断信号标志着 I/O 请求的完成。如果在该设备的等待队列中有其它的请求，那么操作系统就开始处理下一个请求。

最后，控制从 I/O 中断中返回。如果一个进程正在等待这个请求的完成（在设备状态表中有记录），那么我们现在就可以将控制返给该进程。否则，我们就返回给 I/O 中断发生前进行的工作：执行其它的用户程序（此用户程序开始了 I/O 操作并且该操作已经完成，但是这个程序没有等待该操作执行完毕）或者是等待循环（该程序开始了两个或更多个 I/O 操作并且正在等待某个结束，但是这个中断来自于另外的操作）。在分时系统中，操作系统可能会转向其它的等待（ready-to-run）进程。

具体的方案可能有所不同。许多交互式系统允许用户事先通过键盘键入数据——在请求数据前输入数据。这样，当设备状态段指明当前没有对该设备的输入请求时，中断可能会发生，以表明接收到了来自终端的字符串。如果允许提前输入，那么就必须提供一个缓冲区来存储事先输入的字符直到某些程序读取它们。通常可能需要为每个输入设备配备一个缓冲区。

异步输入输出的最大的优点是提高了系统效率。在 I/O 操作发生的同时，CPU 可以处理其它工作或是开始其它设备的 I/O 操作。因为与处理器速度相比 I/O 速度相当缓慢，所以异步输入输出可以提高系统效率。在 2.2.2 节，我们将描述提高系统性能的另外一种机制。

1.2.2 DMA 结构

在一个简单的终端输入设备中，从终端读取数据时，输入的第一个字符被发送到计算机中。当接收到这个字符，连接到终端线上的异步通信（或串行端口）设备就向 CPU 产生一个中断。当接收到来自终端的中断请求时，CPU 就会执行一些指令（如果 CPU 正在执行某个指令，那么中断就要等待该指令执行完毕）。保存这个中断指令的地址，并将控制转移给中断服务程序。

中断服务程序要保存一些 CPU 寄存器值，因为将来还需要用到这些数据。它要检查最近的输入操作有没有产生错误。然后从设备中读取字符并将其保存在一个缓冲器中。中断服务程序还必须调整指针和计数变量以保证将下一个输入字符存储在缓冲器中的下一个位置。下一步，中断服务程序将在内存中设置一个标记，以此向操作系统的其它部分表明接收到了新的输入。其它的部分则负责处理缓冲器中的数据，并将字符传送给请求数据的程序（2.5 节）。然后，中断处理程序恢复刚才保存的寄存器内容并将控制返回给被中断的指令。

如果向 9600 波特的终端中输入字符，那么终端大约能够以每毫秒一个字符（也就是每 1000 微秒一个字符）的速度接受和传输。一个优秀的中断服务程序输入一个字符可能需要 2 微秒，每秒钟留给 CPU 998 微秒计算（和为其它的中断服务）。根据这个不同点，异步 I/O 通常具有一个较低的中断优先权，允许首先处理其它的更重要的进程，甚至是一个进程抢占当前进程的中断。然而一个高速设备（比如：磁带、磁盘或者是通讯网络）可能以接近于内存的速度传输数据；举个例子，如果 CPU 响应每个中断需要 2 微秒，中断每 4 微秒到达一次，那么就没有留下多少时间用于执行进程。

为了解决这个问题，在高速设备中应用了直接内存访问（DMA）技术。在为设备设置好缓冲器、指针和计数器后，设备控制器直接或者是通过自身的缓冲器将整个数据块读取至内存，整个过程无需 CPU 干涉。每个数据块仅仅产生一个中断，而不像慢速设备每个字节（或字）就产生一个中断。

CPU 的基本操作是相同的。一个用户程序或者操作系统本身可能请求数据传输。操作系统从缓冲池中为数据传输指定一个缓冲器（一个空缓冲区用于输入，一个满缓冲区用于输出）。（根据设备类型，一个缓冲器典型的大小在 128 到 4,096 字节之间。）下一步，设备驱动程序（操作系统的一部分）将源地址、目标地址和传输数据长度设置到 DMA 控制器的寄存器中。然后，DMA 控制器命令开始 I/O 操作。当 DMA 控制器执行数据传输时，CPU 可以自由的执行其它的任务。因为存储器一次通常只能传输一个字，DMA 控制器从 CPU 中“窃取”了存储周期。在进行 DMA 传输时，周期挪用（cycle stealing）降低了 CPU 的执行速度。传输结束后 DMA 控制器就向 CPU 发出中断。

1.3 存储器结构

计算机程序必须在主存储器（也称之为随机读写存储器或简称 RAM）中才可以被执行。主存储器是唯一能够被处理器直接访问的大块存储空间（容量在数百万到数十亿字节之间）。它是一种半导体存储器，被称为动态随机读写存储器（DRAM），它由存储器字队列组成。每个字都有它自己的地址。通过调用 load 或 store 指令来对具体的地址进行读写。load 指令一次将一个字从主存储器移动到 CPU 的某个内部寄存器中；store 指令则将寄存器内容转移到主存储器中。除了直接调用 load 和 store 指令以外，CPU 自动的从主存储器中读取指令执行。

在采用冯诺伊曼体系结构的计算机系统中，一个典型的指令周期首先从内存中读取指令，然后将指令存储到**指令寄存器**中。该指令将被解码，也可能需要从内存中读取操作数并将操作数存储到一些内部寄存器中。要注意到，内存单元仅仅被视为内存地址流；并不知道内存地址是如何产生的（来自指令计数器、变址寻址、间接寻址、线性地址等等）或者它们是什么（指令还是数据）的地址。因此，我们可以忽视内存地址是怎样产生的。我们仅仅知道内存地址序列由正在运行的程序产生。

理想的，我们希望能够将程序和数据长久的保存在主存储器中。但实际上这是不可能的，主要有如下两个原因：

1. 主存储器通常太小而不能长久的存放所需的全部程序和数据。
2. 主存储器是一种易失性存储设备，会在掉电时丢失存储的信息，某些其它的原因也会造成数据丢失。

这样，大多数计算机提供了**辅助存储器**作为对主存储器的一个扩充。辅助存储器主要用来长久的保存大量数据。

磁盘是应用最常用的辅助存储器，它可以保存程序和数据。大多数程序（网页浏览器、编译器、文字处理软件、电子制表软件等）在载入内存前存储在磁盘上。有些程序利用磁盘存放原始数据和运算结果。因此，合适的磁盘管理程序对一个计算机系统来说是相当重要的，这个问题将在第十四章中讨论。

从更大的方面讲，前面提到的寄存器、主存储器和磁盘可能仅仅是许多存储系统的一部分。另外还有高速缓冲存储器、CD-ROM、磁带等。每个存储系统都提供了基本的数据存储能力，并且能够保持数据不会丢失。各种各样的存储系统之间的主要不同点在于速度、价格、尺寸和易失性。我们将在 2.3.1 节到 2.3.3 节讨论主存储器、磁盘和磁带，因为它们具备了所有重要的具备商用价值的存储设备的一般特性。将在第十四章中讨论软盘驱动器、硬盘驱动器、CD-ROM 和 DVD 等设备的具体特性。

1.3.1 主存储器

主存储器和处理器内部寄存器是唯一能够由 CPU 直接访问的存储器。有些机器指令将内存地址作为参数，但是却没有指令将磁盘地址作为参数。所以，指令运行时所需要的数据必须要存储在这些能够直接访问的存储设备中。如果数据不在内存中，那么 CPU 首先把它们转移到内存中，然后才能够对它们进行操作。

在 I/O 操作的情况下（就像 2.1 节提到的那样），每个 I/O 控制器拥有很多寄存器，这些寄存器用于存储即将被传输的命令和数据。通常，具体的 I/O 指令允许在这些寄存器和系统内存之间进行数据传输。为了更加便捷的访问 I/O 设备，许多计算机系统采用了**内存映象 I/O**。这样，内存的一部分地址范围被留出来并映射到设备寄存器中。对设备寄存器的访问就可以通过读写内存地址实现。这种技术适用于视频控制器等能够快速响应的设备。在 IBM 个人计算机中，显示屏上的每块区域都被映射到内存中相应的区域。在屏幕上显示文本就像往（相应的）内存映象中写入文本一样简单。

内存映象输入输出也适用于其它一些设备，比如用于将调制解调器和打印机连接到计算机上的串行和并行端口。CPU 通过对这些设备的寄存器进行读写来实现与它们的数据传输，这部分寄存器被称为**I/O 端口**。为了通过存储映象串行端口发送一长串字节，CPU 首先将数据字节写入数据寄存器，然后在控制寄存器中设置一个标志位来表明数据准备完毕。设备接收到该数据字节后将控制寄存器中的标志位清零以表明准备好接收下一个字节。然后，CPU 传输下一个字节。如果 CPU 以轮询方式检查每一个控制位，那么就有一个（周期）恒定的循环来检查设备是否准备好，这种方式被称为**程序控制输入输出 (PIO)**。如果 CPU 不对控制位进行轮询检查，而是通过接收中断（设备准备好接收下一个字节时会向 CPU 发出一个中断）来实现，那么这种数据传输被称为**中断驱动**。

CPU 中的寄存器的访问时间通常是一个 CPU 时钟周期。大多数 CPU 能够以每个时钟跳变一个或多个操作的速率完成对指令的解码并实现对寄存器内容的简单操作。主存储器就不同了，它通过存储总线进行数据传输。

内存的访问通常需要花费多个 CPU 时钟周期才能完成，这样，因为缺乏完成指令所需的数据，处理器通常需要停止运行（**stall**）。由于要频繁的访问内存，这种情形简直忍无可忍。解决方法是在 CPU 和主存储器间添加快速存储器。一个缓冲存储器可以调节 CPU 和内存间的速度差异，它被称为高速缓冲存储器，这将在 2.4.1 节讨论。

1.3.2 磁盘

磁盘为现代计算机系统提供了大容量的辅助存储空间。从理论上讲，磁盘相对来说要简单一些（图 2.5）。每个盘片是像 CD 一样的平坦的圆形物体。盘片直径通常在 1.8 到 5.25 英寸之间。盘片的两面覆盖了一层磁性材料。我们通过在盘片上进行磁性记录来存储信息。

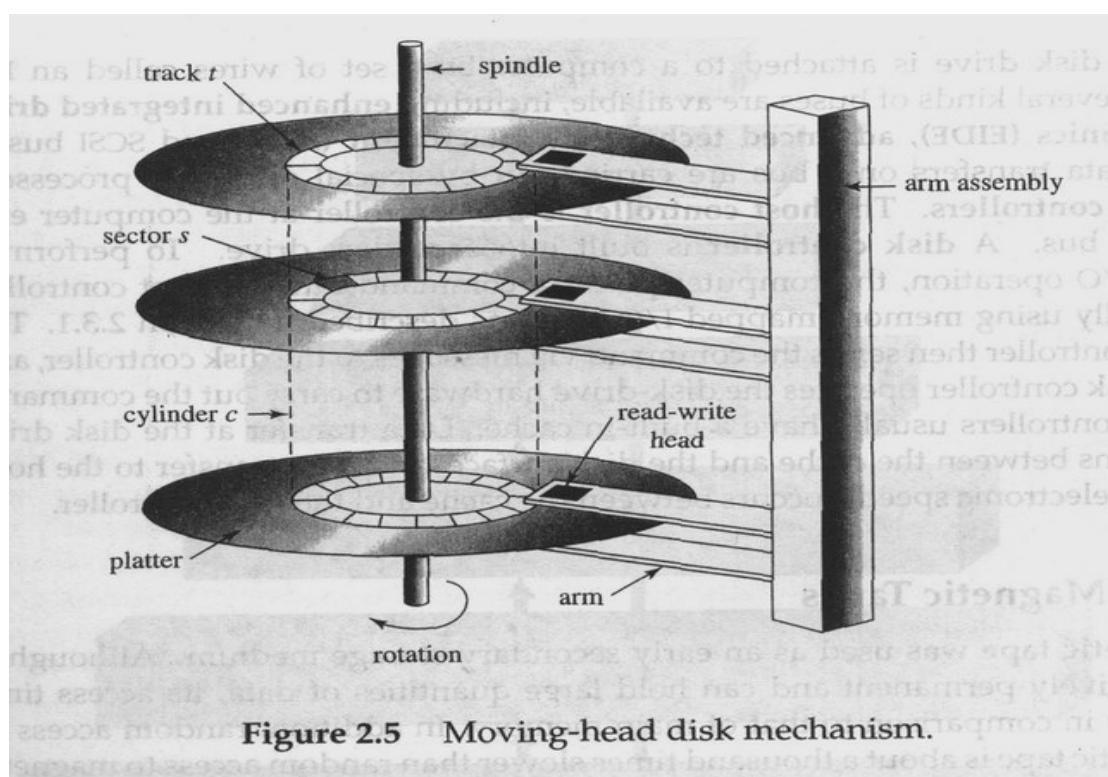


Figure 2.5 Moving-head disk mechanism.

一个读写头在每个盘片表面“飞行”。磁头与**磁盘取数臂**相连，磁盘取数臂作为一个单元控制所有的磁头。盘片表面被划分为圆形的**磁道**，磁道又被划分为**扇区**。在同一取数臂位置上的磁道集组成一个**柱面**。一个磁盘可能由数千个柱面组成，每个磁道又包含了数百个扇区。通用的磁盘驱动器的存储容量的度量单位为吉。

当磁盘运行时，驱动马达高速旋转。大多数驱动器的转速在每秒 60 到 200 转之间。磁盘速度分为两部分。**传送速率**是数据在驱动器和计算机之间的流动速率。**定位时间**（有时也被称为**随机存取时间**）由寻道时间和旋转等待时间组成。**寻道时间**是磁盘取数臂移动到指定柱面所需的时间。**旋转等待时间**则是指指定的扇区旋转到磁头所需的时间。典型的磁盘的数据传输速率为数兆每秒，寻道时间和旋转等待时间为数毫秒。

因为磁头在盘片上面很薄的一层气垫（其厚度以微米计）上飞行，所以就会有磁头碰到盘片的危险。虽然盘片由一层薄薄的保护膜覆盖着，但有时磁头仍会损伤磁表面。这种事故被称为**磁头碰撞**。磁头碰撞所造成的盘片损伤是不可修复的；这就需要更换磁盘了。

有种磁盘是**可移动的**，根据需要可以使用不同的磁盘。可移动磁盘通常有一个盘片，这个盘片放入驱动器

之前有一个塑料外套来保护它免收损害。软磁盘是一种廉价的可移动磁盘，它用一个软塑料外壳包裹着一个软盘片。软盘驱动器的磁头往往直接接触磁盘表面，所以这种驱动器的转速就要比硬盘慢很多，这样可以减少对磁盘表面的损耗。可移动磁盘的工作方式与硬盘非常相似，其存储容量由兆度量（注：原著认为移动磁盘的容量以吉为单位度量，这可能是个错误，因为这儿没有提到便携硬盘）。

软盘驱动器通过一组导线与计算机连接，这组导线被称为输入输出总线。现有多种总线标准，其中包括增强型 IDE 接口（EIDE）、ATA 和小型计算机系统接口（SCSI）。总线上的数据传输通过一个被称为控制器的特殊电子芯片实现。主控制器位于总线末端的计算机中。磁盘控制器构建在每个磁盘驱动器中。为了进行磁盘的 I/O 操作，计算机将命令发送到主控制器中，通常通过映象内存 I/O 端口，就像在 2.3.1 节中所描述的那样。然后主控制器通过消息将命令发送到磁盘控制器中，然后磁盘控制器再运行磁盘驱动硬件来执行该命令。磁盘控制器往往有一个内建的高速缓冲存储器。磁盘驱动器中的数据传输在高速缓冲存储器和磁盘表面进行，然后数据以电子速率传输到主控制器中。

1.3.3 磁带

磁带是一种在早期应用广泛的辅助存储器。虽然它能够长久的存储大量数据，但是与主存储器相比它的读写速度太慢了。另外，磁带的随机读写速度仅仅是磁盘的千分之一。所以磁带不适合作为辅助存储器使用。磁带通常主要用作备份，用来存储不常用的数据以及用于在系统间传输数据。

磁带缠绕在线轴上，通过在读写头前缠绕或反绕进行工作。将磁带移动到所需的位置需要耗费数分钟，但是一旦定位完毕，磁带驱动器的写入速度可与磁盘驱动器相当。根据具体的磁带驱动器，磁带的存储容量很不一样。有些磁带能够存储一张大容量磁盘 2 到 3 倍的数据。磁带和其驱动器通常根据（磁带的）宽度分类，这包括：4、8、19 毫米和 1/4、1/2 英寸。

1.4 存储器体系

根据速度和价格，计算机系统中应用的各种各样的存储器系统可以通过一个分级层次结构进行组织（图 2.6）。更高层的存储器价格更贵，速度更快。当我们往该层次下面转移时，通常每位的价格会降低，但读写周期会增加。这种权衡是有道理的；如果给定的一种存储系统与另外一种相比更快更便宜（其它的特性相同），那么就没有理由选择更贵更慢的存储器。事实上，包含纸带和磁心存储器在内的许多早期的存储器早就进了博物馆了，现在磁带和半导体存储器已经变得更快更便宜。在图 2.6 中，位于顶端的三层存储器可能采用半导体存储器构造。

除了速度和价格上的不同之外，各种存储系统在易失性和非易失性上的也有所不同。易失性存储器在掉电后会丢失数据。由于缺乏昂贵的电池和备用发电机，必需要将数据写入非易失性存储器中以保证数据的安全存储。在图 2.6 中所表示的层次结构中，电子磁盘上面的存储系统是易失性的，而下面的是非易失性的。电子磁盘可被设计为易失性的或非易失性的。在正常的操作期间，电子磁盘将数据存储在一个大型的 DRAM 阵列中，这是易失性的。但是许多电子磁盘驱动器隐含着一个磁硬盘和一个备用电池。如果外部电源被阻断，那么电子磁盘控制器将数据从 RAM 拷入磁盘中。当外部电源恢复时，控制器又将数据拷回到 RAM 中。

设计一个完整的存储系统必需权衡以下几个方面：尽可能的提供廉价的非易失性的存储器，只有在必要的情况下才使用昂贵的存储器。在两个组件的访问时间或速率差距很大的情况下可以使用高速缓冲存储器来提高系统性能。

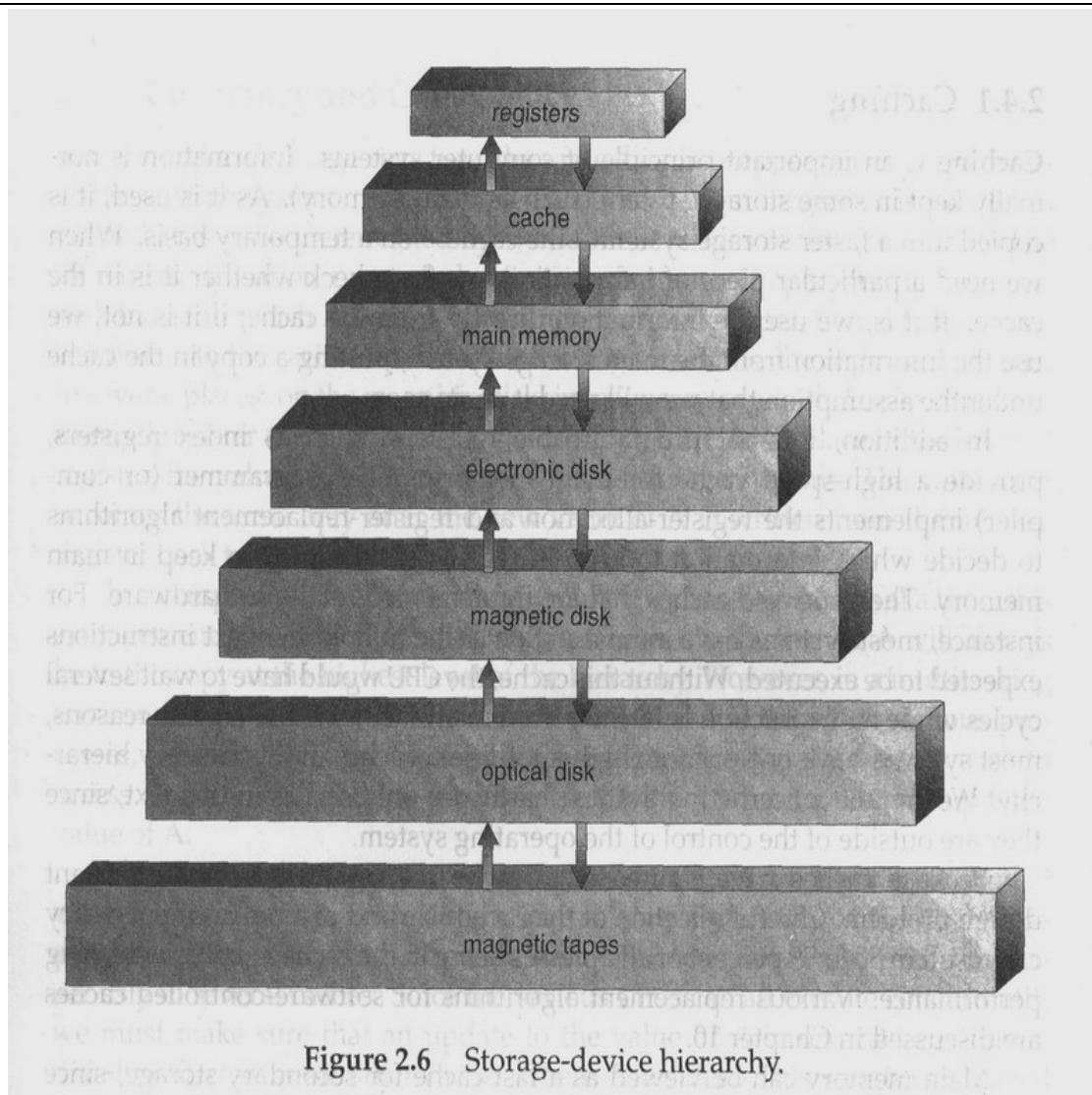


Figure 2.6 Storage-device hierarchy.

Figure 2.6 Storage-device hierarchy.

1.4.1 高速缓冲存储技术

高速缓冲存储技术是计算机系统中相当重要的一个概念。数据通常存储在一些存储系统(如主存储器)中。在使用数据时,首先要把它转移到更快速的存储器(高速缓冲存储器)中。当我们需要使用特定的数据片段时,首先检查它是否在高速缓冲存储器中。如果在,就直接读取;否则,就从主存储系统中读取,并向高速缓冲存储器中添加一个备份(这样是为了下次使用这个数据。当然,这只是个假设,我们未必会再次用到它)。

另外,变址寄存器等内部可编程寄存器为主存储器提供了高速缓冲存储能力。程序员(或编译器)通过实现寄存器分配和替换算法以决定哪些数据保存在寄存器中,哪些保存在内存中。也有些高速缓冲存储器完全以硬件实现。例如,大多数系统使用一个指令高速缓冲存储器来存储要执行的下一条指令。如果没有高速缓冲存储器,CPU就必须要在从主存储器中取指令时等待几个周期。类似的,大多数系统在其存储体系中拥有一个或多个高速数据缓冲存储器。因为这在操作系统的范畴之外,所以在本书中我们并不关心这种纯硬件的存储设备。

由于受到高速缓冲存储器的容量限制,高速缓冲存储器管理的设计是一个相当重要的问题。精心选择的高

速缓冲存储器的容量和替换策略能够使高速缓冲存储器的访问比达到 80% 到 99% 之间，在很大程度上提高了系统性能。在第十章中将讨论软件控制的高速缓冲存储器的替换算法。

可以将主存储器看成辅助存储器的高速缓冲存储器，因为辅助存储器的数据必须要拷到主存储器中才能使用，而主存储器中的数据必须要放到辅助存储器中才能安全存储。长期驻留在辅助存储器中的文件系统数据可能会出现在多个存储层次中。在最高层，CPU 可能会维护一个高速缓冲存储器来保存主存储器中的文件系统数据。电子式 RAM（也成为固态硬盘）可以用作高速存储设备，通过文件系统接口可以实现对它的访问。大容量的辅助存储器是磁盘。反过来，为了预防磁盘故障所造成的数据丢失，磁盘中存储的数据通常备份在磁带或可移动磁盘上。为了降低存储成本，有些系统能够将辅助存储器中的旧文件归档保存到第三方存储器中，如：小型盒式磁带机（jukebox）。

依据硬件设备和操作系统控制软件，数据可能直接或间接的在多个存储层间转移。例如，从高速缓冲存储器到 CPU 和寄存器的数据传输完全由硬件实现，而不受操作系统的干预。另一方面，磁盘到内存的数据传输通常由操作系统控制。

1.4.2 一致性

在存储器层次体系结构中，同样的数据可能会同时出现在多个层次中。例如：假设文件 B 中的一个整型数 A 被加 1，而文件 B 保存在磁盘中。首先，加法操作进程通过一个 I/O 操作将 A 调入主存。然后，又将 A 拷入 cache 和内部寄存器中。这样，A 的拷贝就会同时在多个地方出现：磁盘、主存、cache 和某个内部寄存器（图 2.7）。一旦加法操作在内部寄存器中完成，那么内部寄存器中的 A 的值就与其它存储器中的不同了。只有将 A 的新值从内部寄存器写回到磁盘后 A 的值才会相同。

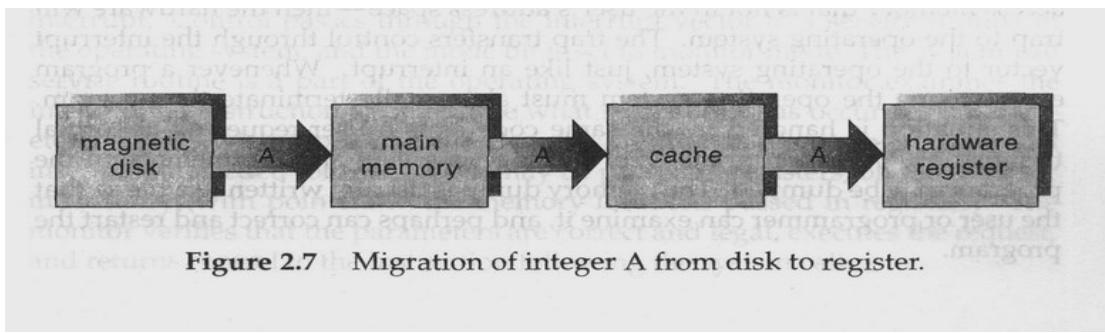


Figure 2.7 Migration of integer A from disk to register.

Figure 2.7

Migration of integer A from disk to register.

在单任务计算环境中，这种策略并不困难，因为总是要将 A 拷贝到最高层的存储器中进行访问。然而，在一个多任务的环境中，CPU 往返于多个进程。如果同时有几个进程要访问 A，那么每个进程都必须要获得最近更新的 A 的值，这一点必须要非常的小心。

在多处理机环境中就更为复杂了，除了要维护内部寄存器之外，每个 CPU 还有一个本地 cache。这样，多个 cache 中可能会同时保留一个 A 的拷贝。因为各个 CPU 并发运行，所以我们必须要确保某个 cache 中的 A 的更新要立即反映到其它的 cache 中。这被称为超高速缓存相关性，通常是一个硬件问题（在操作系统层面之下）。

在分布式环境中，这甚至更加复杂。在这样的环境中，相同文件的多个拷贝（或复制品）可以保存在分布在空间内的多个不同的计算机中。因为能够访问和更新这些拷贝，所以我们必须要确保一处的更新尽快的影响到其它的地方。有多种方法可以解决这个问题，我们将在第十六章中讨论。

1.5 硬件保护

早期的计算机是单用户、程序员操作的 (single-user programmer-operated) 系统。当一个程序员通过控制台操作计算机时，他就会完全控制整个系统。然而，随着操作系统的发展，这种控制权交给了操作系统。早期的操作系统被称为**常驻监督程序**，通过开启常驻监督程序，操作系统开始执行许多原先由程序员负责的功能(尤其是 I/O)。

另外，为了提高系统利用率，操作系统开始在多个程序间同时共享系统资源。利用假脱机技术，在某一进程的 I/O 操作进行的同时可以执行另外的程序；磁盘中的数据可同时由多个进程使用。多道程序设计允许同时在内存中保留多个程序。

这种共享在提高了利用率的同时带来了不少问题。在没有共享的系统中，一个程序错误仅仅影响到其自身。而在共享的情况下，一个程序错误可能会牵连多个进程。

例如，考虑简单的批处理操作系统 (1.2.1 节)，它只是提供了自动作业队列。在对列循环中，如果一个程序正在读取输入卡片，它会读取所需的全部数据。在没有干扰的情况下，将由下一个作业读取卡片，然后是再下一个，依次类推。这个循环可能会导致许多作业不能正确操作。

在多道程序系统中甚至会发生更多很微妙的错误，一个不正确的程序可能会修改程序或其它程序的数据，甚至是常驻监督程序自身。MS-DOS 和 Macintosh OS 中都可能发生这种错误。

若没有针对这种错误的保护，或者计算机每次只能执行一个程序或者怀疑所有的输出的正确性。优良设计的操作系统必须要确保一个错误（或恶意代码）不会导致其它程序错误执行。

通过硬件可以检测到许多程序错误。这些错误通常由操作系统处理。如果一个用户程序以某种方式出现故障——例如，试图执行非法指令或访问超出用户地址空间的内存——硬件将自陷给操作系统。自陷通过中断向量将控制交给操作系统。不论一个程序何时发生错误，操作系统必须要非正常的终止它。通过执行与正常的用户请求终止代码相同的代码可以处理这种状况。给出一条适当的错误消息，并可能会转储该程序的内存。内存转储通常将信息写到一个文件中，这样用户或程序员就可以对此进行检查并可能改正及重新运行这个程序。

1.5.1 双模式操作

为了确保操作的正确性，我们必须要保护操作系统和所有的程序以及它们的数据不受错误工作的程序的影响。对许多共享资源来说，保护是必须的。许多操作系统提供了硬件支持来允许我们区分不同的操作模式。

我们至少要区分两种不同的操作模式：**用户模式**和**监控模式**（也称为**管理模式**、**系统模式**、**特权模式**）。在计算机硬件中添加一个比特位（被称为**模式位**）来表示当前的模式：监控模式（0）或用户模式（1）。利用模式位，我们可以区别一个任务是以操作系统的执行还是用户的执行。就像我们将要看到的那样，这种体系结构上的改进在系统运行的许多其它方面也是非常有用的。

在系统引导时，硬件以监控模式开始运行。然后装入操作系统并以用户模式开启用户进程。无论自陷和中断何时发生，硬件都会从用户模式转向监控模式（将模式位的状态转为 0）。这样，不管操作系统何时获得计算机的控制权，它都处于监控模式。在将控制转给用户程序前系统总是要转为用户模式（通过将模式位设置为 1）。

双模式操作向我们提供了保护操作系统不受用户错误侵扰的方案，也保护了用户不受其他用户错误的侵扰。通过指定一些能够造成伤害的机器指令作为**特权指令**可以实现这种保护。在监控模式下，硬件允许执行特权指令。如果试图在用户模式下执行特权指令，那么硬件就会视该指令非法并自陷给操作系统。

特权指令的概念也为我们提供了与操作系统交互的方法，就是请求操作系统执行一些只能由操作系统才能执行的任务。每个这样的请求通过执行一条特权指令调用。这样的请求被称为系统调用（也被称为监督程序调用）。

用或操作系统功能调用)——就像是 2.1 节所描述的。

当系统调用执行时,硬件视之为软件中断。在操作系统中,控制通过中断向量转移给服务进程,模式位被设为监控模式。系统调用服务进程是操作系统的一部分。监控程序检测中断指令以确定发生了什么样的系统调用;程序参数则表明了用户程序所请求的服务类型。该请求的附加信息通过寄存器、堆栈或内存(将内存地址转移给寄存器)传递。监控程序检测参数的正确性,执行这个请求,然后将控制返回给系统调用。

硬件对双模式支持的缺乏会给操作系统带来严重的缺点。例如,MS-DOS 针对于 Intel 8088 体系结构,这种结构中没有模式位,所以也就没有双模式。用户程序以写数据的方式覆盖操作系统就能够毁坏整个系统,并且多个程序同时对某一设备进行写操作也会产生灾难性的后果。最新的 Intel CPU(如 Pentium 系列)提供了对双模式的支持。这样,新近的操作系统(如 Microsoft Windows 2000 和 IBM OS/2)利用了这一特性并为操作系统提供了更有力的保护。

1.5.2 I/O 保护

如果用户程序调用非法的 I/O 访问操作系统自身的内存地址或拒绝释放 CPU,那么该用户程序就会损坏系统的正常运行。我们能够利用多种机制来确保不会在系统中发生这样的事情。

为了阻止用户执行非法的 I/O 操作,我们将所有的 I/O 指令定义为特权指令。这样,用户不能够直接调用 I/O 指令;而必须要通过操作系统执行。为了实现对 I/O 的完全保护,我们必须确保用户程序不会在监控模式下获得对计算机的控制。

考虑计算机在用户模式下的执行情况。不论中断或自陷何时发生,它都会转入监控模式,跳转到中断向量指定的地址。如果一个用户程序在运行期间使用一个用户程序的地址覆盖了中断向量中原先的地址。那么,当相应的自陷或中断发生时,硬件通过(修改过的)中断向量将控制移交给用户程序!用户程序就可以在监控模式下获得对计算机的控制了。事实上,用户程序可以通过许多其它的方式在监控模式下获得对计算机的控制。另外,经常发现的新臭虫为绕过系统保护提供了机会。将在第十八章和第十九章中讨论这些问题。如此,为了完成 I/O 操作,用户程序执行系统调用来请求操作系统以其自己的方式执行 I/O 操作(图 2.8)。操作系统运行在监控模式,它会检查请求的有效性,并在(在请求有效地情况下)完成 I/O 请求。最后,操作系统返回给用户。

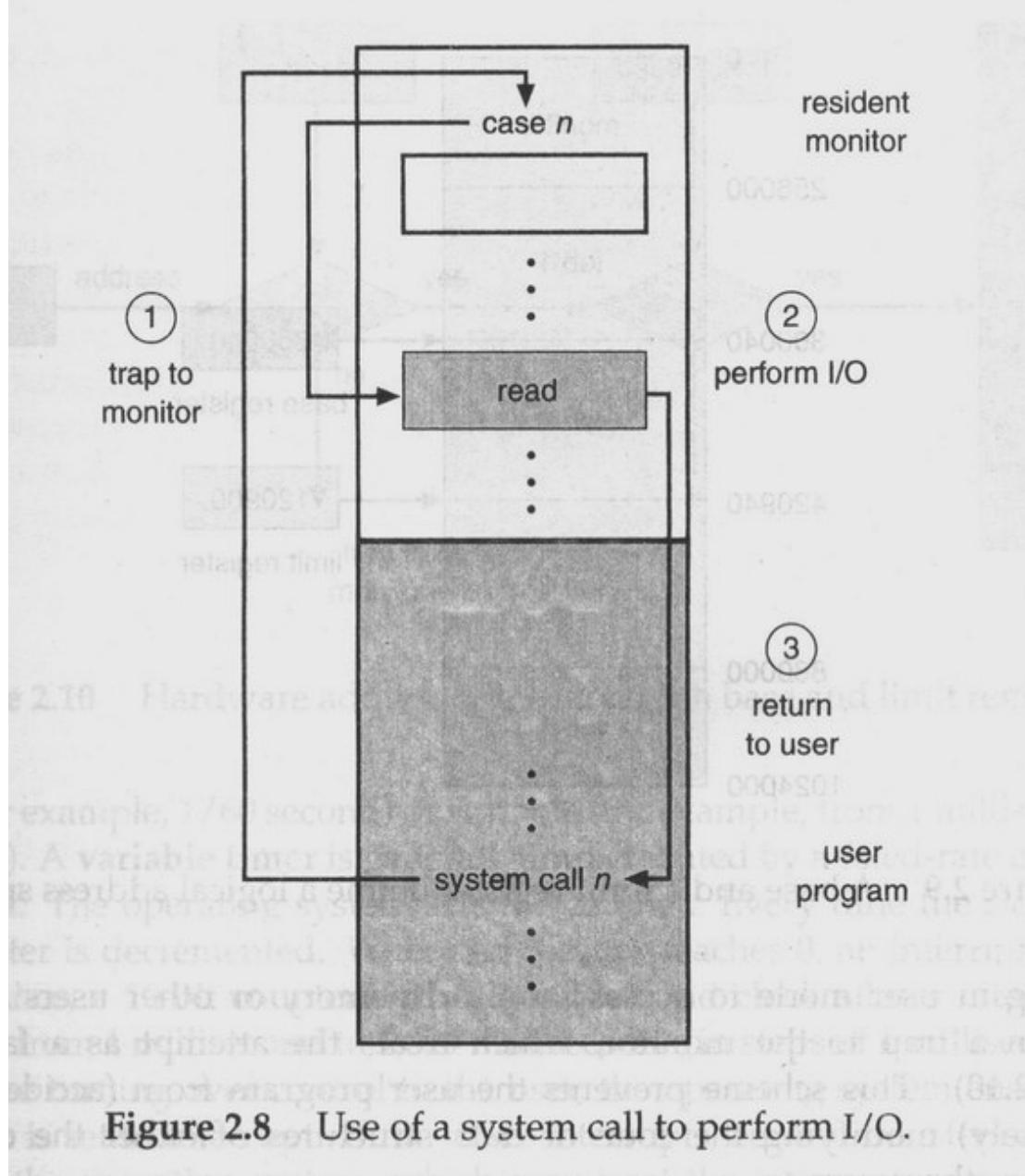


Figure 2.8 Use of a system call to perform I/O.

1.5.3 内存保护

为了确保操作的正确执行，我们必须保护中断向量免受用户程序的修改。另外，我们也必须保护操作系统中的中断服务进程不被修改。即使用户不能非法获得对计算机的控制权，对中断服务进程的修改也有可能破坏计算机系统和它的假脱机和缓冲处理的正确运行。

我们必需要认识到：至少要为中断向量和操作系统的中断服务进程提供保护。一般而言，我们要保护操作系统不会受到用户进程的访问，另外，也要保证用户进程不被其它的用户进程访问。这种保护必须要由硬件提供。就像我们将在第九章所描述的，有多种方法可以实现。这里，大概的介绍一下一种可能的实现方法。

为了分离每个进程的内存空间，我们要能够确定程序可访问的地址范围并保护该范围之外的内存空间。我

们可以通过使用两个寄存器来提供这种保护，一个记录基地址，一个记录界限，如图 2.9 所示。**基址寄存器**保存了最低的合法的物理内存地址；**界限寄存器**则保存了范围的大小。例如，如果基址寄存器的值为 300040，界限寄存器的值为 120900，那么程序能够访问到的合法的地址在 300040 到 420940 之间。

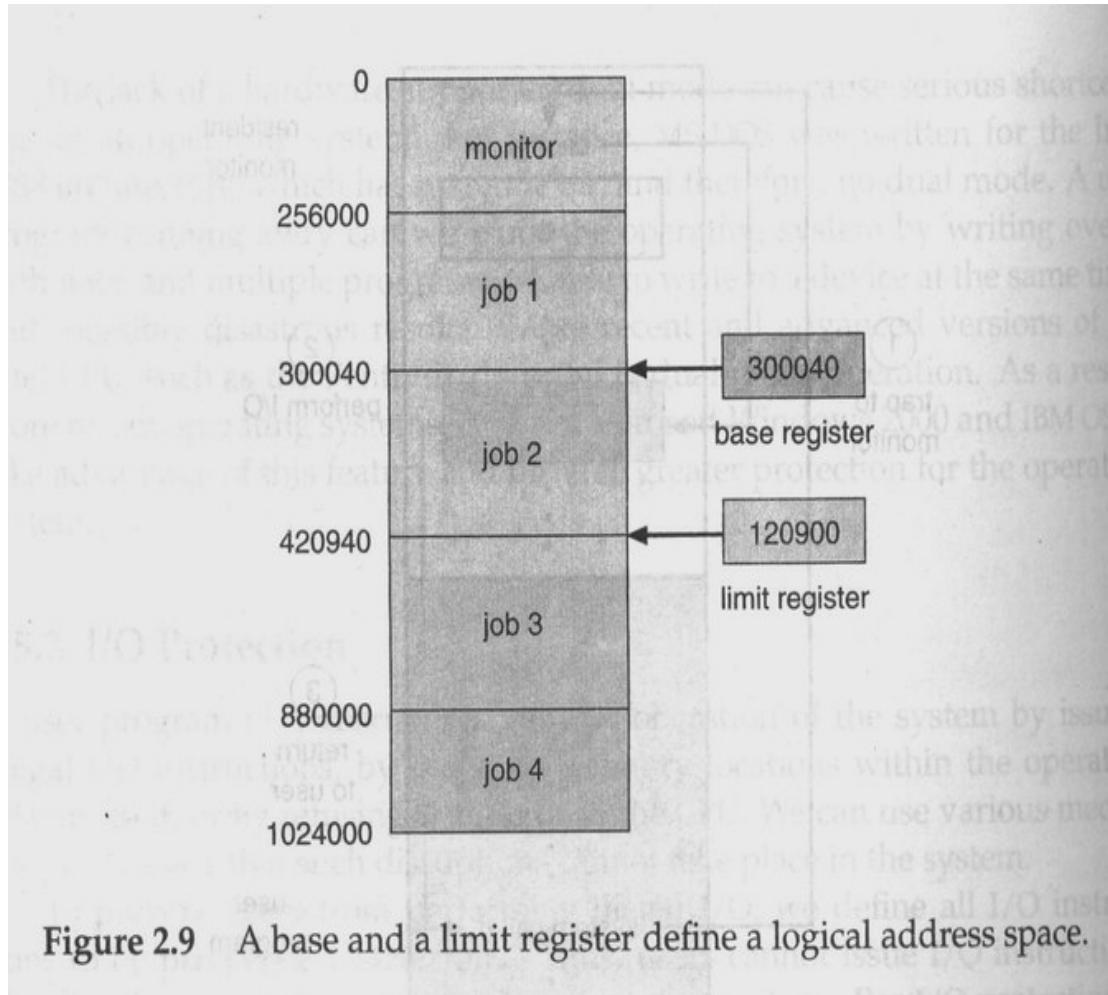


Figure 2.9 A base and a limit register define a logical address space.

Figure 2.9 A base and a limit register define a logical address space.

CPU 利用寄存器比较在用户模式下产生的每个地址就可以实现这种保护。运行在用户模式下的程序对监控进程或其他用户进程的内存的任何访问企图都将导致一个给监控程序的自陷，这被视为致命错误（图 2.10）。这种机制阻止了用户进程对操作系统和用户程序的代码或数据结构的修改（意外的或故意的）。

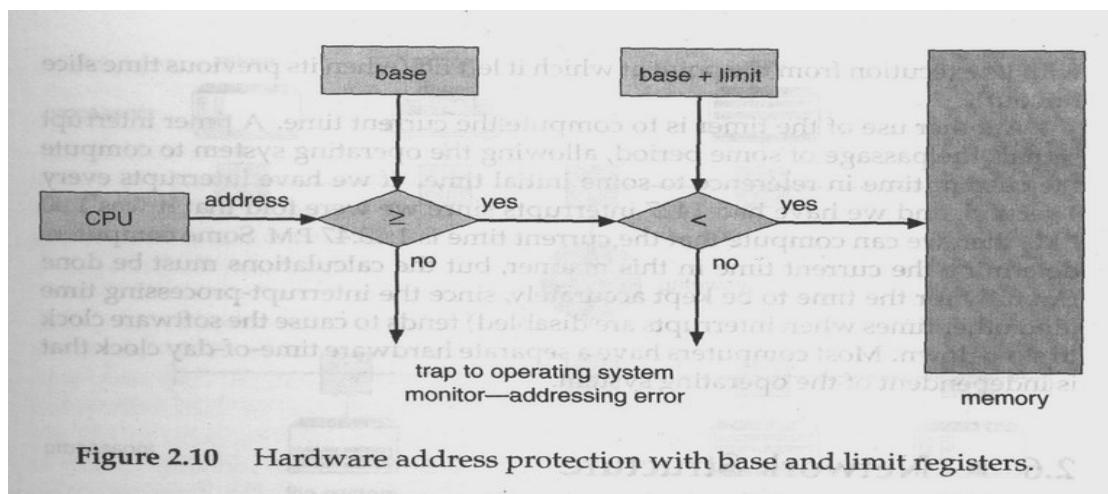


Figure 2.10 Hardware address protection with base and limit registers.

Figure 2.10 Hardware address protection with base and limit registers.

只有操作系统能够使用特殊的特权指令写入基址寄存器和界限寄存器。因为特权指令只能够在监控模式下执行，而且只有操作系统才能够在监控模式下运行，所以只有操作系统才能够写入基址寄存器和界限寄存器。这种机制允许监控程序修改寄存器值，而阻止用户进程修改寄存器。

运行在监控模式下的操作系统拥有对监控程序内存和用户内存的无限制的访问权力。这种规定允许操作系统将用户程序载入用户内存、在错误发生时释放内存、访问并修改系统调用参数等。

1.5.4 CPU 保护

除了对I/O和内存的保护，我们必须明确要由操作系统来维护控制。我们必须要阻止用户进程困在死循环中或不调用系统服务，以及永远不能将控制交还给操作系统。（We must prevent a user program from getting stuck in an infinite loop or not calling system services, and never returning control to the operating system.）为了达到这一目标，我们可以使用**计时器**。可以将寄存器设为在指定周期后向计算机发出中断。这个周期可以是固定的（例如，1/60 秒）也可以是可变的（例如，在 1 毫秒到 1 秒）。**可变计时器**通常由一个固定频率的时钟和一个计数器实现。操作系统设置这个计数器。每次时钟“滴嗒”时，计数器减 1。当计数器减至 0 时，就发生一个中断。例如，10 比特的计数器和 1 毫秒的时钟允许中断发生的间隔在 1 毫秒到 1,024 毫秒 ($2^{10}=1,024$) 之间，步调为 1 毫秒。

在将控制交给用户之前，操作系统要确保已经设置好了计时器。如果计时器产生中断，控制将自动的转移给操作系统，操作系统可能视其为致命错误，也可能给程序更多时间。很明显，修改计时器的指令为特权指令。

这样，我们就能够利用计数器来限定程序的运行时间。一个简单的方法是使用允许程序运行的时间长度来初始化计数器。例如，如果一个程序有着 7 分钟的时间限制，那么就将计数器初始化为 420。只要计数器是正的，控制就会交给用户程序。当计数器变为负值时，因为已经到了设定的时间限制，操作系统将终止程序的运行。

对计时器的更为广泛的应用是为了实现分时。在大多数简单的情况下，可以将计时器设置为每 N 毫秒产生一次中断，这里 N 是**时间片**，也就是每个用户在下一个用户获取对 CPU 的控制之前可以执行 N 毫秒。在每个时间片的末端调用操作系统来实现各种内部处理工作，比如：对时间记录加 N，该时间记录以指明（为了计算的需要）用户程序已运行的总时间。为了准备下一个程序的运行，操作系统也要保存寄存器、内部变量和缓冲器，并且要修改其它的一些参数。这个过程被称为**上下文转换**；将在第四章中讨论这个问题。上下文转换后，下一个程序从它上一次停下的地方（它的上一个时间片结束时）继续运行。

计时器的另外一个作用是计算当前时间。每隔一段时间计数器就发出一次中断信号，这就允许操作系统根据初始化时间计算出当前时间。如果设定的初始时间为下午 1:00，每秒钟产生一个时钟中断，那么 1427 次中断之后我们就可以计算出当前时间为下午 1:23:47。有些计算机以这种方式获取当前时间，但是因为中断处理时间会延缓软件时钟，所以为了保证时间的精确性，计算要非常的仔细。大多数计算机有一个单独的与操作系统分离的硬件日历钟（time-of-day clock）。

1.6 网络结构

局域网（LAN） 和 **广域网（WAN）** 是两种基本的网络结构。二者的主要区别在于地理位置分布范围的不同。局域网由在小地理范围（例如，独立的建筑物或邻近的建筑群）内分布的处理器构成。而广域网由跨越广阔的地理范围（如美国）内的相互独立的计算机构成。这些不同点意味着通信网络在速度和可靠性上的巨大差

异，而且也反映在分布式操作系统的设计上。

1.6.1 局域网

局域网于 70 年代早期作为大型计算机的一种替代品出现的。对于许多企业来说，使用一系列自主运行的小型机要比一台大型机更经济。因为小型机可能需要一些外围设备（如磁盘和打印机），而且单独的企业也可能需要某种形式的数据共享，所以自然而然的就需要把这些小型机连接到网络上。

局域网通常覆盖一个小地理范围（例如，独立的建筑物或邻近的多个建筑物），往往用在办公环境中。这种系统上的各个点离得很近，所以相对于广域网来说局域网的通信连接就会有较高的速度和较低的误码率。为了获取更高的速度和可靠性，就需要高质量的（昂贵的）电缆。对于较远的距离来说，使用高质量电缆的代价是非常高昂的，所以电缆不可以独占使用。

局域网大多使用双绞线和光缆连接。根据网络所采用的拓扑结构，大多为多路访问总线网络（multiaccess bus）、环形网络和星形网络。通信速度在 1 兆每秒（如 AppleTalk、红外和最新的 Bluetooth 局部无线网络）到 1 吉每秒（如以太网）之间。大多数的速度为 10 兆每秒，记为 **10BaseT Ethernet**。**100BaseT Ethernet** 需要更高质量的电缆，但它以 100 兆每秒的速度运行，这种网络日渐普及。基于光缆的 FDDI 的应用也逐步增多。FDDI 是一种基于令牌环的网络，它的速度为 100 兆每秒。

一个典型的局域网可能由许多不同的计算机（从巨型机到膝上计算机或 PDA）、多种共享的外围设备（如激光打印机和磁带驱动器）和提供对其它网络的访问的一个或多个网关（专门的处理器）构成（图 2.11）。**以太网**是一种通用的局域网结构。以太网采用多路访问总线结构，所以它没有中央控制器，这样很容易向网络上添加新主机。

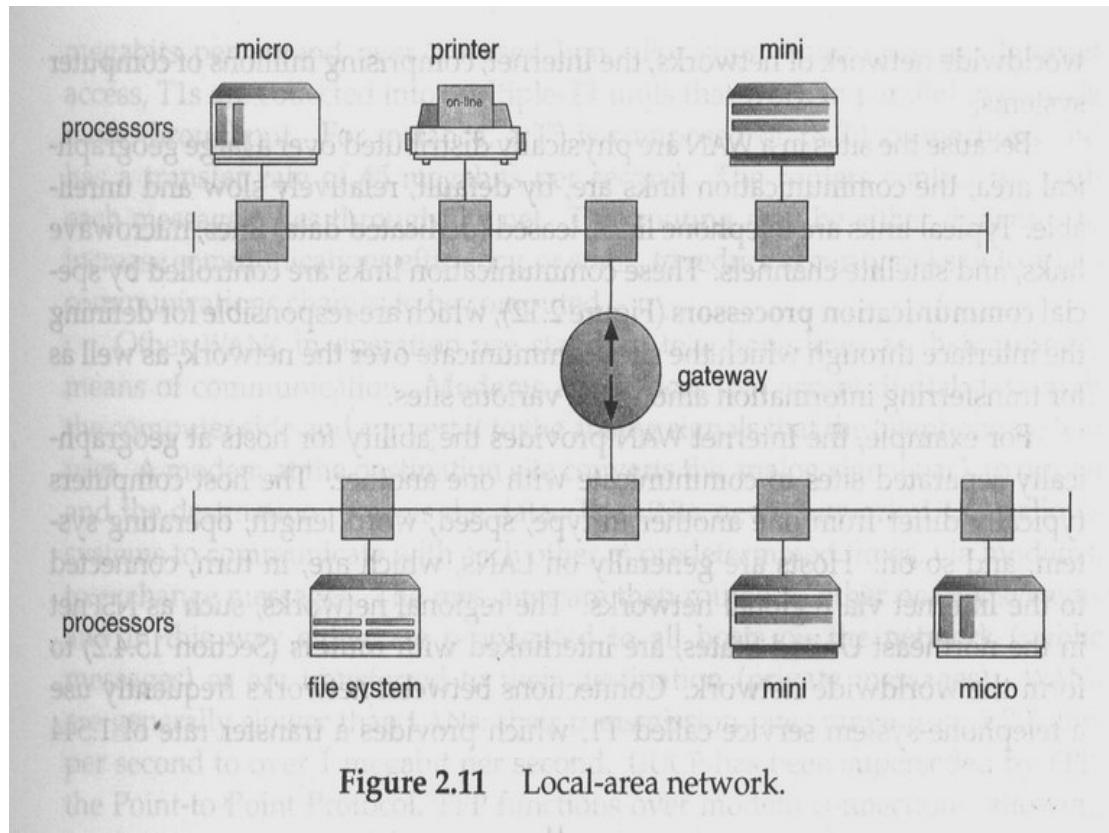


Figure 2.11 Local-area network.

Figure 2.11

Local-area network.

1.6.2 广域网

广域网诞生于 60 年代末，主要是作为一个学术研究项目。该项目旨在提供多节点间的高效连接，并且允许广泛的用户群体经济便利的共享数据。

Arpanet 是设计和开发的第一个广域网。从 1968 年开始，Arpanet 从一个只有四个节点的实验性网络逐步成长为由数百万台计算机系统组成的世界范围内的网络——Internet。

因为广域网中的每个节点分布在一个广阔的地理范围内，所以相对来说通信连接速度较慢可靠性较低。电话线、租用（专用数据）线路、微波和卫星信道是典型的连接方式。这些通信连接由专用的**通信处理器**控制（图 2.12），通信处理器负责定义节点在利用网络通信所用的接口，也负责在各个节点间传输数据。

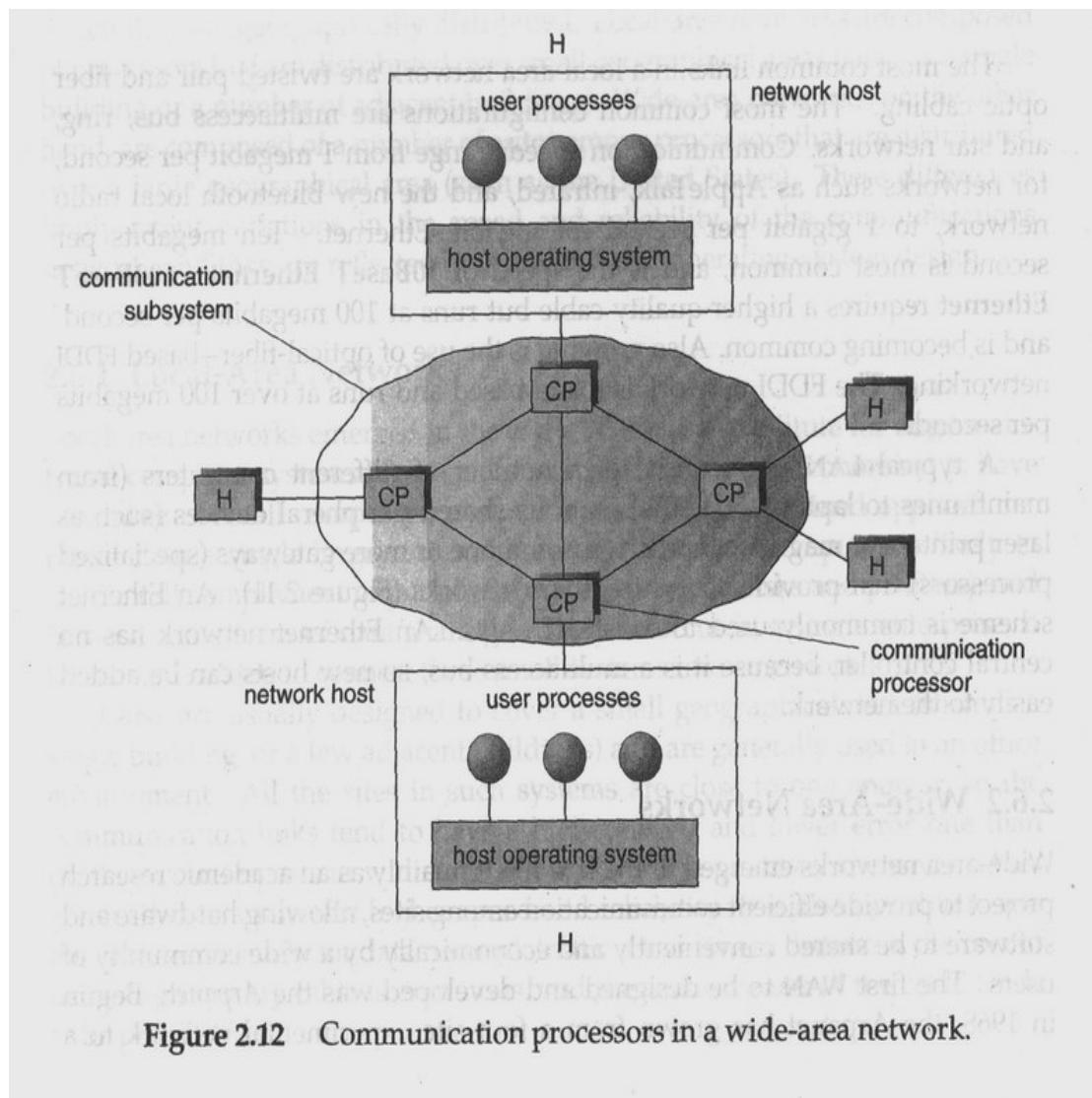


Figure 2.12 Communication processors in a wide-area network.

例如，Internet WAN 提供了在地理上独立的主机相互通信的能力。计算机主机间典型的不同点在于类型、速度、字长和操作系统等。通常在局域网上的主机可以通过区域网络（regional network）连接到 Internet（如

美国东北部的 NSFnet) 利用路由器与全球通信网连接。网络间的连接常常使用电话服务系统, 这被称为 T1, 它能够通过租用线路提供 1.544 兆每秒的传输速率。因为网络节点需要更快的 Internet 访问速率, 所以就将多个 T1 单元集中到一起并行工作以提供更高的吞吐量。例如, T3 是 28 个 T1 的集合, 它的传输速率为 45 兆每秒。路由器控制信息在网络中的传输路径。路由选择可以是动态的 (为了提高通信效率), 也可以是静态的 (为了降低安全风险或计算通信费用)。

其它的 WAN 使用标准的电话线作为主要的通信方式。调制解调器是一种从计算机接收数据并将其转换为电话系统所使用的模拟信号的设备。在目的节点, 调制解调器将模拟信号转换为数字信号并由节点接收数据。UNIX 新闻网络 UUCP 允许系统在预定时间通过调制解调器相互交换信息。然后, 消息被传送到邻近的系统中, 并且以这种方式, 或者将消息传送给网络上的每个主机 (公共信息), 或者传送给目标主机 (机密信息)。WAN 的数据传输速率通常要比 LAN 慢; 它的传输率在 1,200 比特每秒到高于 1 兆每秒之间。UUCP 已经被 PPP 取代。PPP 通过调制解调器连接, 允许家庭计算机与 Internet 完全连接。

1.7 摘要

多道程序系统和分时系统在单机上重叠 CPU 和 I/O 操作提高了计算机性能。这样的重叠需要 CPU 和 I/O 之间的数据传输通过轮询或中断驱动访问 I/O 端口的方式实现, 或者是使用 DMA 数据传输。

计算机要完成执行程序的任务, 程序必须要保存在主存中。主存是唯一能够由处理器直接访问的大容量存储空间。它是一个字或字节的队列, 容量在几十万到几亿之间。每个字都有它自己的地址。主存是易失性的存储设备, 掉电后会丢失存储的信息。大多数计算机系统支持辅助存储器作为主存的扩展。对辅助存储器的需求主要是为了能够长久的保存大量数据。最常用的辅助存储设备是磁盘, 它能够存储程序和数据。磁盘是一种支持随机存取的非易失性存储设备。磁带往往用于备份不常用的数据, 也能够作为媒介在多个系统间进行数据传输。

在计算机系统中, 依照速度和价格可以将广泛的存储器系统通过一个层次结构进行组织。更高层的存储器价格更贵, 但是速度更快。当我们往层次结构的下面转移时, 每位的价格通常会降低, 然而访问速度会提高。

操作系统必须要确保计算机系统能够正确操作。为了阻止用户程序干扰系统的正常运行, 硬件使用了两种模式: 用户模式和监控模式。有些指令 (如 I/O 指令和停机指令) 是特权指令, 它们只能在监控模式下执行。也必须要保护操作系统所驻留的内存免于用户的修改。利用一个计时器可以阻止死循环的运行。这些策略 (双模式、特权指令、内存保护、计时器中断) 是操作系统正确运行的基础, 将在第三章中详细讨论。

LAN 和 WAN 是两种基本的网络类型。LAN 通常使用昂贵的双绞线或光缆连接, 它允许在小地理范围内分布的处理机进行通信。WAN 通过电话线、租用线路、微波或卫星信道连接, 它允许分布在广阔的地理范围内的处理机进行通信。LAN 的典型传输速率高于 100 兆每秒, 而 WAN 的传输速率在 1,200 比特每秒到高于 1 兆每秒之间。

词汇

ATA: advanced technology attachment, ATA

传送速率: transfer rate

IDE: Integrated Drive Electronics

磁带: magnetic tape

半导体存储器: semiconductor memory

磁道: track

常驻监督程序: resident monitor

磁盘: magnetic disk

超高速缓存相关性: cache coherency

磁盘控制器: disk controller

程序控制输入输出: programmed I/O, PIO

磁盘取数臂: disk arm

初始化程序: initial program

磁头碰撞: head crash

磁心存储器: core memory	设备驱动程序: device driver
点对点协议: Point-to-Point Protocol, PPP	设备状态表: device-status table
电子磁盘: electronic disk	时间片: time slice
调制解调器: modem	输入输出总线: I/O bus
定位时间: positioning time	随机存取时间: random-access time
动态随机读写存储器: dynamic random-access memory, DRAM	随机读写存储器: random-access memory, RAM
端口: port	特权模式: privileged mode
非易失性存储器: nonvolatile storage	特权指令: privileged instruction
冯诺伊曼体系结构: von Neumann architecture	停机指令: halt instruction
辅助存储器: secondary storage	通信处理机: communication processor
高速缓冲存储技术: caching	同步输入输出: synchronous I/O
高速缓冲存储器: cache	系统调用: system call
高速缓冲存储器管理: cache management	系统模式: system mode
固态磁盘: solid-state disk	小型计算机系统接口: small computer system interface, SCSI
上下文转换: context switch	旋转等待时间: rotational latency
管理模式: supervisor mode	寻道时间: seek time
光缆分布式数据接口: Fiber Distributed Data Interface , FDDI	以太网: Ethernet
广域网: wide-area network, WAN	异步输入输出: asynchronous I/O
基址寄存器: base register	异常: exception
计时器: timer	易失性: volatility
监督程序调用: monitor call	易失性存储器: volatile storage
监控模式: monitor mode	引导程序: bootstrap program
界限寄存器: limit register	内存映象输入输出: memory-mapped I/O
局域网: local-area networks, LAN	用户模式: user mode
可变计时器: variable timer	增强型 IDE 接口: enhanced integrated drive electronics, EIDE
可移动的: removable	直接内存访问: direct memory access, DMA
控制器: controller	纸带: paper tape
路由器: router	指令寄存器: instruction register
模式位: mode bit	中断: interrupt
内存转储: memory dump	中断驱动: interrupt driven
盘片: platter	中断向量: interrupt vector
日历钟: time-of-day clock	主控制器: host controller
软(磁)盘: floppy disk	柱面: cylinder
扇区: sector	自陷: trap

【未完待续·责任编辑: iamxiaohan@hitbbs】

Linux 核心 (The Linux Kernel)

原著: David A Rusling¹

翻译²: 毕昕³、胡宁宁⁴、仲盛⁵、赵振平⁶、周笑波⁷、李群⁸、陈怀临⁹

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

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

Chapter 3 Memory Management

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.

Virtual memory does more than just make your computer's memory go further. The memory management subsystem provides:

Large Address Spaces

The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system.

Protection

Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

第 3 章 内存管理

内存管理子系统是操作系统中最重要的组成部份之一。从早期计算机开始，系统的实际内存总是不能满足需求，为解决这一矛盾，人们想了许多办法，其中虚存是最成功的一个。虚存让各进程共享系统内存空间，这样系统就似乎有了更多的内存。

虚存不仅使计算机的内存看起来更多，内存管理子系统还提供以下功能：

扩大地址空间

操作系统使系统看起来有远远大于它实际所拥有的内存空间。虚存能比系统的物理内存大许多倍。

内存保护

系统中每个进程都有它自己的虚拟地址空间。这些虚拟地址空间之间彼此分开，以保证应用程序运行时互不影响。另外，硬件虚存机制可以对内存部分区域提供写保护，以防止代码和数据被其它恶意的应用程序所篡改。

Memory Mapping

Memory mapping is used to map image and data files into a processes address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Fair Physical Memory Allocation

The memory management subsystem allows each running process in the system a fair share of the physical memory of the system.

Shared Virtual Memory

Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each processes virtual address space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes.

Shared memory can also be used as an Inter Process Communication (IPC) mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix System V shared memory IPC.

3.1 An Abstract Model of Virtual 3.1 一个抽象的虚存模型

Memory

Before considering the methods that Linux uses to support virtual memory it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location in memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

内存映射

内存映射被用于将映像和数据文件映射到一个进程的虚拟地址空间中，也就是将文件内容直接地连接到虚地址中。

公平分配内存

内存管理子系统可以使每一个在系统中运行的进程公平的共享系统的物理内存。

虚存共享

尽管虚存允许各进程有各自的(虚拟)地址空间，但有时进程间需要共享内存。例如，若干进程同时运行 Bash 命令。并非在每个进程的虚地址空间中都有一个 Bash 的拷贝，在内存中仅有一个运行的 Bash 拷贝供各进程共享。又如，若干进程可以共享动态函数库。

共享内存也能作为一种进程间的通信机制 (IPC)。两个或两个以上进程通过共享内存来交换数据这非常普遍。Linux 支持 Unix System V 的共享内存 IPC 机制。

3.1 一个抽象的虚存模型

在分析 Linux 实现虚存的方法前，让我们先来看一个没有由于过多细节而混乱的抽象模型。

当处理器执行一段程序时，它先从内存中读出一条指令并对它进行解码。解码时可能需要在内存中的某一地址存取数据。然后处理器执行这条指令并移向下一条。可见处理器总是不断地在内存中存取数据或指令。

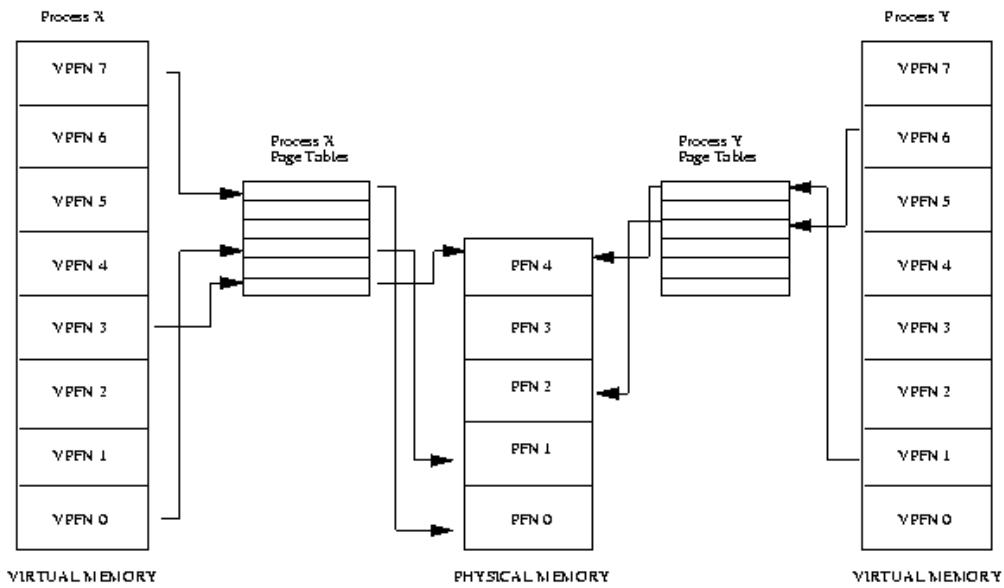


Figure 3.1: Abstract model of Virtual to Physical address mapping

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into handy sized chunks called pages. These pages are all the same size, they need not be but if they were not, the system would be very hard to administer. Linux on Alpha AXP systems uses 8 Kbyte pages and on Intel x86 systems it uses 4 Kbyte pages. Each of these pages is given a unique number; the page frame number (PFN).

In this paged model, a virtual address is composed of two parts; an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses page tables.

Figure 3.1 shows the virtual address spaces of two processes, process X and process Y, each with their own page tables. These page tables map each processes

在虚存系统中，所有地址都是虚地址而非物理地址。处理器根据操作系统维护的一组表格而把这些虚地址翻译成相应的物理地址。

为使这翻译的过程更容易，虚存和物理内存被划分成许多适当大小的块，这些块称之为“页”。这些页的大小都是一样的，当然不是必须这样，不过如果不这样的话操作系统会非常难管理它们。在 Alpha AXP 上的 Linux 系统中，每页有 8Kbyte，但在 Intel x86 系统中，每页有 4Kbyte。每一页又被分配了一个各不相同的数字，叫页号（PFN）。

在页模型中，一个虚地址由两部份组成：偏移量和虚页号。如果页的大小是 4Kbytes，那么虚地址的 0 至 11 位是偏移量，第 12 位以上是虚页号。每当处理器遇到虚地址时，它先取出偏移量和虚页号。然后，处理器把虚页号翻译成物理页号，再由偏移量得到正确的物理地址，最后存取数据。处理器需要使用页表来完成这整个过程。

图 3.1 显示了两个进程的虚存地址空间。进程 X 和进程 Y 分别有各自的页表。页表记录了各进程虚页和物理页之间的映射。如图：X 进程的虚存的第 0 页

virtual pages into physical pages in memory. This shows that process X's virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process Y's virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

- Valid flag. This indicates if this page table entry is valid,
- The physical page frame number that this entry is describing,
- Access control information. This describes how the page may be used. Can it be written to? Does it contain executable code?

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at Figures 3.1 and assuming a page size of 0x2000 bytes (which is decimal 8192) and an address of 0x2194 in process Y's virtual address space then the processor would translate that address into offset 0x194 into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However the processor delivers it, this is known as a page fault and the operating system is notified of the faulting virtual address and the reason for the page fault.

Assuming that this is a valid page table entry, the

<http://emag.csdn.net>

<http://purec.binghua.com>

被映射为物理内存的第 1 页, Y 进程的虚存的第 1 页被是非曲直射为物理内存的第 4 页。理论上, 页表中每条记录包含以下信息:

- 有效性标志。用以标识页表记录有效与否。
- 物理页号。
- 存取控制信息。描述这页应该怎样被使用。是否可写? 是否包含可执行代码?

页表中使用虚页号作为偏移量。虚页 5 将是表中的第 6 条记录 (0 是第一条记录)。

把一个虚地址翻译成物理地址时, 处理器必须先得出虚页号和偏移量。让页的大小总是 2 的幂, 这便于进行 mask 和移位操作。图 3.1 中, 假定页的大小是 0x2000 字节 (它是十进制的 8192), 在进程 Y 的地址空间中有一虚地址 0x2194。那么处理器将把这个地址翻译成偏移量为 0x194, 虚页号为 1。

处理器使用虚页号作为检索进程页表记录的索引。如果对应那偏移量的页表记录是有效的, 处理器就从中拿出物理页号。如果记录是无效的, 表明进程想存取一个不在物理内存中的地址。在这种情况下, 处理器不能翻译这个虚地址, 而必须把控制权传给操作系统, 让它处理。

当当前进程试图存取一个处理器无法翻译的虚地址时, 处理器如何通知操作系统这是与特定的处理器相关的。不过, 通常的做法是, 处理器会引发一个“页错误”, 并将产生页错误的虚地址和原因告诉给操作系统。

假设找到的是一有效的页表记录, 处理器就取出

processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process Y's virtual page frame number 1 is mapped to physical page frame number 4 which starts at 0x8000 ($4 * 0x2000$). Adding in the 0x194 byte offset gives us a final physical address of 0x8194.

By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order. For example, in Figure 3.1 process X's virtual page frame number 0 is mapped to physical page frame number 1 whereas virtual page frame number 7 is mapped to physical page frame number 0 even though it is higher in virtual memory than virtual page frame number 0. This demonstrates an interesting byproduct of virtual memory; the pages of virtual memory do not have to be present in physical memory in any particular order.

3.1.1 Demand Paging

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to only load virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not all of the database needs to be loaded into memory, just those data records that are being examined. If the database query is a search query then it does not make sense to load the code from the database program that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as demand paging.

When a process attempts to access a virtual address that is not currently in memory the processor cannot find a page table entry for the virtual page referenced. For example, in Figure 3.1 there is no entry in process X's page table for virtual page frame number 2 and so if process X attempts to read from an address within virtual page frame number 2 the processor cannot

物理页号并且乘以页的大小，得到内存中页的地址。最后，处理器加上偏移量得到它需要指令或数据的地址。

再次使用上面的例子，进程 Y 的虚存的第 1 页被映射到物理内存的第 4 页，它从 0x8000 ($4 * 0x2000$) 开始。加上偏移量 0x194 字节，我们得到最后的物理地址就是 0x8194。

由虚地址映射到物理地址时，虚存各页映射到系统内存中的顺序是任意的。例如，在图 3.1 中，进程 X 的虚存第 0 页被映射到内存第 1 页，而虚存第 7 页被映射到内存第 0 页，即使它的虚存页号比虚存页号 0 要大 (这里直译不太好理解，其实主要的意思就是虽然后的虚存页号比前者的虚存页号要大，但是映射到物理内存中的物理页号却比前者映射的物理页号要小，作者以此例来说明虚存各页映射到物理内存各页的顺序是任意的)。这说明了虚存的一个有趣现象，虚存各页在物理内存中不必有任何顺序。

3.1.1 按需装载页

由于物理内存比虚拟内存小很多，因此操作系统必须专注于物理内存的使用效率。节省物理内存的一个方法是只装载被当前执行程序使用的虚页。例如，有一个用来查询数据库的程序，此时，并非所有数据库中的数据都需要被装载进内存，只需要那些正在被访问的数据。如果正运行一条数据库查询命令，那么就不必载入添加新的数据记录的代码。这种只有在访问时才将对应的虚页载入内存的技术，叫做按需装载页。

当进程试图存取一个不在内存中的虚地址时，处理器不可能在页表中找到这一虚页的记录。例如，在图 3.1 中，进程 X 的虚存第 2 页没有对应的页表记录，如果尝试对这页进行读操作，那么处理器不能把虚地址翻译成物理地址。处理器就会通知操作系统发生了一个页错误。

translate the address into a physical one. At this point the processor notifies the operating system that a page fault has occurred.

If the faulting virtual address is invalid this means that the process has attempted to access a virtual address that it should not have. Maybe the application has gone wrong in some way, for example writing to random addresses in memory. In this case the operating system will terminate it, protecting the other processes in the system from this rogue process.

If the faulting virtual address was valid but the page that it refers to is not currently in memory, the operating system must bring the appropriate page into memory from the image on disk. Disk access takes a long time, relatively speaking, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual page frame number is added to the processes page table. The process is then restarted at the machine instruction where the memory fault occurred. This time the virtual memory access is made, the processor can make the virtual to physical address translation and so the process continues to run.

Linux uses demand paging to load executable images into a processes virtual memory. Whenever a command is executed, the file containing it is opened and its contents are mapped into the processes virtual memory. This is done by modifying the data structures describing this processes memory map and is known as memory mapping. However, only the first part of the image is actually brought into physical memory. The rest of the image is left on disk. As the image executes, it generates page faults and Linux uses the processes memory map in order to determine which parts of the image to bring into memory for execution.

3.1.2 Swapping

If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

If the page to be discarded from physical memory

<http://emag.csdn.net>

<http://purec.binghua.com>

如果页错误对应的虚地址是无效的，这意味着进程试图存取它不应该访问的虚地址。这也许是由于应用程序出了某些错误，例如试图在内存中任意进行写操作。在这种情况下，操作系统将终止这个错误进程，以保护其它进程。

如果页错误对应的虚地址是有效的，只是这页目前不在物理内存中，操作系统必须将对应的页从磁盘载入内存。相对来说，磁盘存取会花很多时间，所以进程必须等待相当一会儿直到页被读入。这时候，如果有其它进程能运行，操作系统将选择其中之一来运行。从磁盘中被取出来的页将被读入内存一空页中，并在进程页表中加入一条记录。然后，进程从产生页错的那条机器指令处重新启动。这次处理器能将虚地址翻译成物理地址了，因此进程能继续运行下去。

Linux 使用按需装载页将可执行的映像载入到进程的虚拟内存空间。一个命令被执行时，包含它的文件被打开，文件的内容被映射入进程的虚存。这一操作需修改描述这一进程内存映像的数据结构。这一过程称为内存映射。然而，只有映像的第一部份被实际载入物理内存，余下部份被留在磁盘上。当映像执行时，它将不断产生页错，Linux 使用进程的内存映像表来决定哪块映像应该被载入内存。

3. 1. 2 页交换 (Swapping)

当进程要装载一虚页进物理内存时，如果得不到可用的物理内存中的空页，操作系统必须从内存中淘汰别的页，为这页提供空间。

如果从内存中被丢弃的那页是从映像或数据文

came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.

However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a dirty page and when it is removed from memory it is saved in a special sort of file called the swap file. Accesses to the swap file are very long relative to the speed of the processor and physical memory and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.

If the algorithm used to decide which pages to discard or swap (the swap algorithm is not efficient then a condition known as thrashing occurs. In this case, pages are constantly being written to disk and then being read back and the operating system is too busy to allow much real work to be performed. If, for example, physical page frame number 1 in Figure 3.1 is being regularly accessed then it is not a good candidate for swapping to hard disk. The set of pages that a process is currently using is called the working set. An efficient swap scheme would make sure that all processes have their working set in physical memory.

Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older and more stale it becomes. Old pages are good candidates for swapping.

3.1.3 Shared Virtual Memory

Virtual memory makes it easy for several processes to share memory. All memory access are made via page tables and each process has its own separate page table. For two processes sharing a physical page of memory, its physical page frame number must appear in a page table entry in both of their page tables.

Figure 3.1 shows two processes that each share

<http://emag.csdn.net>

<http://purec.binghua.com>

件中来的，并且这页没有被修改过，那这页就不需要再被保存，可以直接丢掉。如果进程再需要那页，它可以重新被操作系统从映像或数据文件中读入内存。

但如果该页已被修改了，操作系统必须保存这页的内容以便它以后能再被访问。这类页叫作“脏页”，当它们被从内存中移出时，它们必须被保存在一种特殊的文件中，这种文件称为“交换文件”。相对于处理器和内存的速度，交换文件的存取时间是很长的，所以操作系统必须权衡是否需要把页写到磁盘上，还是保留在内存中以备后用。

如果用来决定那个页应当被淘汰或交换的交换算法的效率不高，那么“颠簸”现象就会发生。在这种情况下，页常常一会儿被写到磁盘上，一会儿又被读回来，操作系统忙于文件存取而不能执行真正的工。例如，图 3.1 中，如果内存第 1 页不断被访问，那它就不应该被交换到硬盘上。进程当前正在使用的页的集合被叫作工作集。有效的交换算法将保证所有进程的工作集都在内存中。

Linux 使用最近最少使用算法 (LRU) 来公平的从内存中选择被丢弃的页。在这个算法中，系统中的每个页都有一个年龄，这个年龄随着页被存取而变化。页被存取得越多便越年轻，被存取得越少就越老。老的页通常是被交换的好候选。

3.1.3 共享虚存

虚存使得若干进程更容易共享内存。进程所有的内存访问都要通过页表，并且各进程有各自独立的页表。当两个进程共享内存中的一页时，物理页号就会同时出现在每个进程的页表中。

图 3.1 中显示两进程共享物理第 4 页。对进程 X

physical page frame number 4. For process X this is virtual page frame number 4 whereas for process Y this is virtual page frame number 6. This illustrates an interesting point about sharing pages: the shared physical page does not have to exist at the same place in virtual memory for any or all of the processes sharing it.

3.1.4 Physical and Virtual 3.1.4 物理和虚拟地址模式

Addressing Modes

It does not make much sense for the operating system itself to run in virtual memory. This would be a nightmare situation where the operating system must maintain page tables for itself. Most multi-purpose processors support the notion of a physical address mode as well as a virtual address mode. Physical addressing mode requires no page tables and the processor does not attempt to perform any address translations in this mode. The Linux kernel is linked to run in physical address space.

The Alpha AXP processor does not have a special physical addressing mode. Instead, it divides up the memory space into several areas and designates two of them as physically mapped addresses. This kernel address space is known as KSEG address space and it encompasses all addresses upwards from 0xfffffc0000000000. In order to execute from code linked in KSEG (by definition, kernel code) or access data there, the code must be executing in kernel mode. The Linux kernel on Alpha is linked to execute from address 0xfffffc0000310000.

3.1.5 Access Control

The page table entries also contain access control information. As the processor is already using the page table entry to map a process's virtual address to a physical one, it can easily use the access control information to check that the process is not accessing memory in a way that it should not.

There are many reasons why you would want to restrict access to areas of memory. Some memory, such as that containing executable code, is naturally read only memory; the operating system should not allow a process to write data over its executable code. By

而言，那是虚存的第 4 页，对进程 Y 而言，那是虚存第 6 页。这说明共享页中的一个有趣的现象：被共享的物理页对应的虚存页号可以各不相同。

把操作系统运行在虚存中是不明智之举，如果操作系统还要为自己保存页表，那将是一场恶梦。因此，很多通用处理器同时支持虚拟地址模式和物理地址模式。物理地址模式不需要页表，处理器不必做任何地址翻译。Linux 内核被直接连在物理地址空间中运行。

Alpha AXP 处理器没有物理地址模式。相反，它把内存划分成若干区域并且指定其中两块为物理地址区。这段内核地址空间叫作 KSEG 地址空间，包括所有 0xfffffc0000000000 以上的地址。在 KSEG 那里执行的（按定义，称之为核心代码）或在那里存取数据的代码肯定是在内核模式下执行。在 Alpha 上的 Linux 内核被连接到从 0xfffffc0000310000 处开始执行。

3.1.5 存取控制

页表记录中也包含了存取控制信息。处理器使用页表记录来把虚地址翻译成物理地址的同时，它也很容易地使用其中的存取控制信息来检查进程是否在正确地访问内存。

在很多种情况下，你想要为内存的一段区域设置存取限制。一段内存，例如包含可执行的代码，应为只读内存；操作系统应该不允许进程在它的可执行的代码上写数据。相反的，包含数据的页能被写，但是当指令试图执行那段内存时，应该失败。大多数处理

contrast, pages containing data can be written to but attempts to execute that memory as instructions should fail. Most processors have at least two modes of execution: kernel and user. You would not want kernel code executing by a user or kernel data structures to be accessible except when the processor is running in kernel mode.

器至少有两种执行模式：内核模式和用户模式。你应当不想由一个用户来执行内核代码，或者让内核数据结构被内核代码之外的代码所访问。

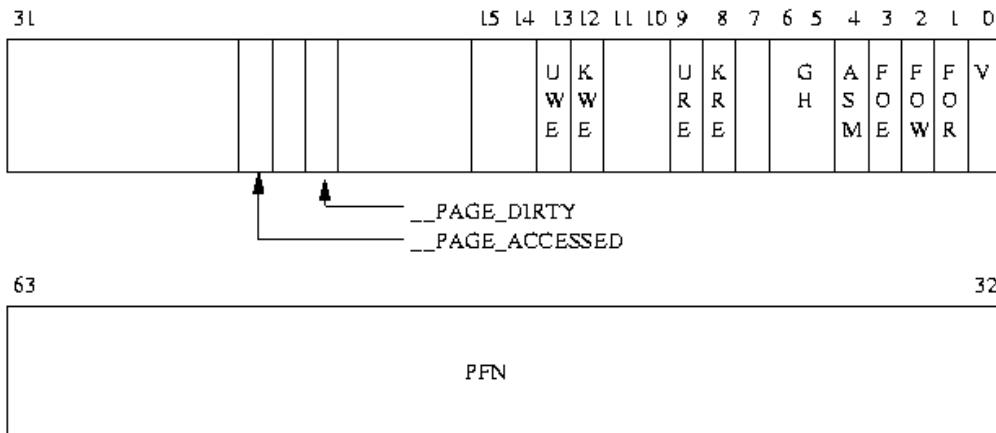


Figure 3.2: Alpha AXP Page Table Entry

The access control information is held in the PTE and is processor specific; figure 3.2 shows the PTE for Alpha AXP. The bit fields have the following meanings:

V

Valid, if set this PTE is valid,

FOE

“Fault on Execute”, Whenever an attempt to execute instructions in this page occurs, the processor reports a page fault and passes control to the operating system.

FOW

“Fault on Write” as above but page fault on an attempt to write to this page,

FOR

“Fault on Read”, as above but page fault on an attempt to read from this page,

ASM

Address Space Match. This is used when the operating system wishes to clear only some of the entries from the Translation Buffer,

KRE

Code running in kernel mode can read this page,

URE

<http://emag.csdn.net>

<http://purec.binghua.com>

存取控制信息被保存在 PTE 中，并且不同的处理器，PTE 的格式是不同的；图 3.2 显示的是 Alpha AXP 的 PTE。各位包含以下信息：

V

有效位。如果设置，表示这 PTE 是有效的。

FOE

执行错误。无论何时，试图在这页执行指令时，处理器将报告页错，并且把控制权传给操作系统。

FOW

写错误。同上面一样，不过页错发生在试图写这页时。

FOR

读错误。同上面一样，不过页错发生在试图读这页时。

ASM

地址空间匹配。当操作系统仅仅希望清除翻译缓冲区中若干记录时，这一位被使用。

KRE

在内核模式下运行的代码能读这页。

URE

Code running in user mode can read this page,

GH

Granularity hint used when mapping an entire block with a single Translation Buffer entry rather than many,

KWE

Code running in kernel mode can write to this page,

UWE

Code running in user mode can write to this page,

page frame number

For PTEs with the V bit set, this field contains the physical Page Frame Number (page frame number) for this PTE. For invalid PTEs, if this field is not zero, it contains information about where the page is in the swap file.

The following two bits are defined and used by Linux:

_PAGE_DIRTY

if set, the page needs to be written out to the swap file,

_PAGE_ACCESSED

Used by Linux to mark a page as having been accessed.

3.2 Caches

If you were to implement a system using the above theoretical model then it would work, but not particularly efficiently. Both operating system and processor designers try hard to extract more performance from the system. Apart from making the processors, memory and so on faster the best approach is to maintain caches of useful information and data that make some operations faster. Linux uses a number of memory management related caches:

Buffer Cache

The buffer cache contains data buffers that are used by the block device drivers.

These buffers are of fixed sizes (for example 512 bytes) and contain blocks of information that have either been read from a block device or are being written to it. A block device is one that can only be accessed by reading and writing fixed sized blocks of data. All hard disks are block devices. (*See fs/buffer.c*)

The buffer cache is indexed via the device

<http://emag.csdn.net>

<http://purec.binghua.com>

在用户模式下运行的代码能读这页。

GH

粒度性。指在映射一整块虚存时，是用一个翻译缓冲记录还是多个。

KWE

在内核模式下运行的代码能写这页。

UWE

在用户模式下运行的代码能写这页。

页号

在 V 位被置位的 PTE 中 (编者注：即有效的 PTE 中)，这域包含了对应的物理页号。对于无效的 PTE，如果这域不是零，它包含了页在交换文件中什么位置的信息。

以下两位是 Linux 定义并使用的：

_PAGE_DIRTY

如果被设置，则页需要被写到交换文件中。

_PAGE_ACCESSED

Linux 用它来标记这页是否曾经被访问。

3.2 缓存

如果你按照上面理论模型，可以实现一个工作的系统，但不会特别高效。操作系统和处理器的设计者都在努力提高系统性能。除提高处理器和内存的速度外，最好的途径是把有用的信息和数据保存在缓存中。Linux 就使用了很多与内存管理有关的缓存：

缓冲区

缓冲区包含了块设备驱动程序所使用的数据缓冲区。

这些缓冲区有固定的大小（例如 512 个字节），记录从一台块设备读或写的信息。一台块设备只能读写整块数据。所有的硬盘都是块设备。（参见 *fs/buffer.c*）

缓冲区以设备标识符和需要的块号做为索引来

identifier and the desired block number and is used to quickly find a block of data. Block devices are only ever accessed via the buffer cache. If data can be found in the buffer cache then it does not need to be read from the physical block device, for example a hard disk, and access to it is much faster.

Page Cache

This is used to speed up access to images and data on disk.

It is used to cache the logical contents of a file a page at a time and is accessed via the file and offset within the file. As pages are read into memory from disk, they are cached in the page cache. (*See mm/filemap.c*)

Swap Cache

Only modified (or dirty) pages are saved in the swap file.

So long as these pages are not modified after they have been written to the swap file then the next time the page is swapped out there is no need to write it to the swap file as the page is already in the swap file. Instead the page can simply be discarded. In a heavily swapping system this saves many unnecessary and costly disk operations. (*See swap.h; mm/swap_state.c; mm/swapfile.c*)

Hardware Caches

One commonly implemented hardware cache is in the processor; a cache of Page Table Entries. In this case, the processor does not always read the page table directly but instead caches translations for pages as it needs them. These are the Translation Look-aside Buffers and contain cached copies of the page table entries from one or more processes in the system.

When the reference to the virtual address is made, the processor will attempt to find a matching TLB entry. If it finds one, it can directly translate the virtual address into a physical one and perform the correct operation on the data. If the processor cannot find a matching TLB entry then it must get the operating system to help. It does this by signalling the operating system that a TLB miss has occurred. A system specific mechanism is used to deliver that exception to the operating system code that can fix things up. The operating system generates a new TLB entry for the address mapping. When the

迅速找到所需数据。块设备只能通过缓冲区进行存取操作。如果数据在缓冲区中，那么它就不需要再从块设备中被读取，例如硬盘，这样会使存取更快。

页缓存

它被用来加快磁盘上映像和数据的存取。

它被用来一次缓存文件的一页，存取操作通过文件名和偏移量来实现。当页从磁盘上被读进内存时，他们被缓存在页缓存中。*(参见 mm/filemap.c)*

交换缓存

只有修改了的页，即“脏页”，被保存在交换文件中。

只要页在被写进交换文件以后，没有再被修改，下次这页被换出内存时，不需要再把它写入交换文件，因为它已经存在于交换文件中了，它只需要被简单的丢弃。对一个要进行许多页面交换的系统，这将节省许多不必要的并且开销极大的磁盘操作。*(参见 swap.h, mm/swap_state.c, mm/swapfile.c)*

硬件缓存

一个常见的硬件缓存是处理器内部的页表记录的缓存。通常情况下，处理器并不总是直接读页表，而是用页表缓存保留用到的记录。这些缓存被叫做 Translation Look-aside Buffer，保存了系统中一个或多个进程页表的拷贝。

当翻译地址时，处理器先试图找到一个匹配的 TLB 记录。如果它找到了一个，它能直接把虚地址翻译成物理地址，并且对数据进行存取操作。如果处理器不能找到一个匹配的 TLB 记录，那就必须借助操作系统。它发信号给操作系统，报告有一个 TLB 疏漏发生。系统特定的机制将把这异常信号送给操作系统中可以解决此问题的代码。操作系统为映射的地址产生一个新的 TLB 记录。当异常被解决后，处理器将尝试再次翻译那个虚地址。因为现在那个地址在 TLB 中有一个有效的记录，因此这次的地址翻译一定成功。

exception has been cleared, the processor will make another attempt to translate the virtual address. This time it will work because there is now a valid entry in the TLB for that address.

The drawback of using caches, hardware or otherwise, is that in order to save effort Linux must use more time and space maintaining these caches and, if the caches become corrupted, the system will crash.

3.3 Linux Page Tables

使用缓冲区、硬件缓存或其它缓存等的缺点是 Linux 必须花费更多的时间和空间来维护这些缓存，如果缓存发生错误，系统将崩溃。

3.3 Linux 页表

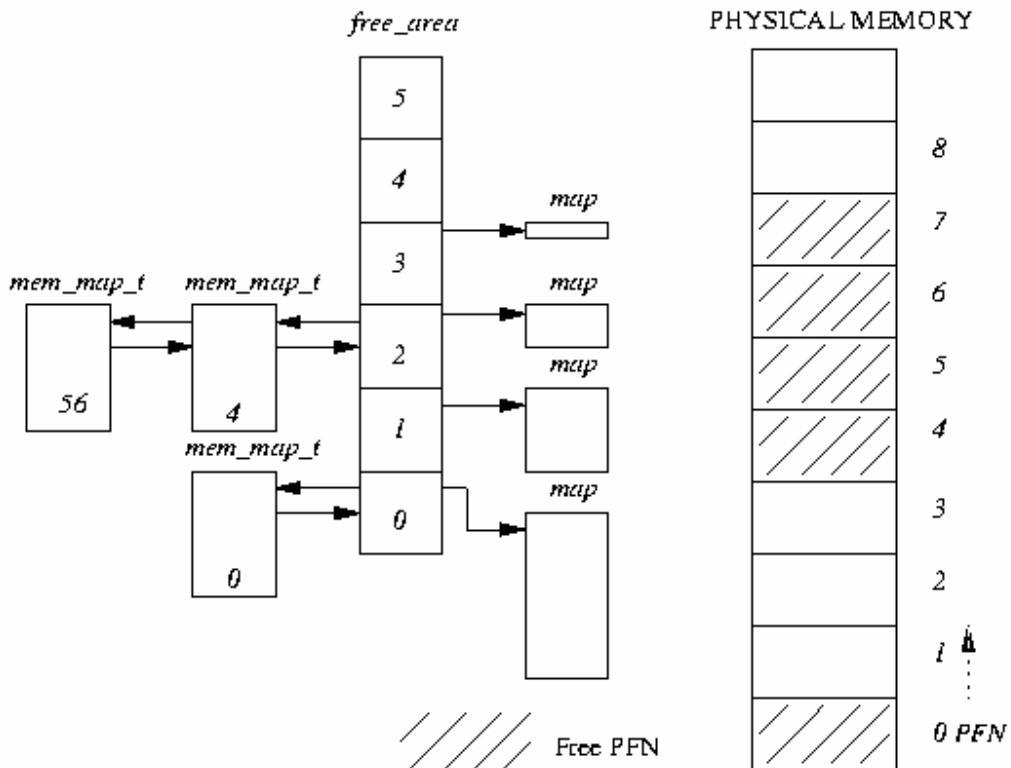


Figure 3.4: The `free_area` data structure

For example, in Figure 3.4 if a block of 2 pages was requested, the first block of 4 pages (starting at page frame number 4) would be broken into two 2 page blocks. The first, starting at page frame number 4 would be returned to the caller as the allocated pages and the second block, starting at page frame number 6 would be queued as a free block of 2 pages onto element 1 of the `free_area` array.

例如，在图 3.4 中，如果需要一个 2 页块，那么第一个空的 4 页块（从第 4 页起）将被分成两半。从第 4 页开始的 2 页块被返回给请求者；从第 6 页开始的 2 页块将被做为空闲块，在 `free_area` 队列的第一个元组中排队。（编者注：这里 `free_area` 队列的第一个元组其实是大小为二的所有空闲页块组成的一个链表）

3.4.2 Page Deallocation

3.4.2 页的回收

Allocating blocks of pages tends to fragment memory with larger blocks of free pages being broken down into smaller ones. The page deallocation code recombines pages into larger blocks of free pages whenever it can. In fact the page block size is important as it allows for easy combination of blocks into larger blocks. (*See free_pages() in mm/page_alloc.c*)

Whenever a block of pages is freed, the adjacent or buddy block of the same size is checked to see if it is free. If it is, then it is combined with the newly freed block of pages to form a new free block of pages for the next size block of pages. Each time two blocks of pages are recombined into a bigger block of free pages the page deallocation code attempts to recombine that block into a yet larger one. In this way the blocks of free pages are as large as memory usage will allow.

For example, in Figure 3.4, if page frame number 1 were to be freed, then that would be combined with the already free page frame number 0 and queued onto element 1 of the free_area as a free block of size 2 pages.

3.5 Memory Mapping

When an image is executed, the contents of the executable image must be brought into the processes virtual address space. The same is also true of any shared libraries that the executable image has been linked to use. The executable file is not actually brought into physical memory, instead it is merely linked into the processes virtual memory. Then, as the parts of the program are referenced by the running application, the image is brought into memory from the executable image. This linking of an image into a processes virtual address space is known as memory mapping.

页分配时容易将大块连续的内存分成很多小块。页回收代码须尽可能将小块的空闲页重新组合成大块的空闲页。事实上，页块的大小对内存的重新组合很重要。In fact the page block size is important as it allows for easy combination of blocks into larger blocks.
(参见 *mm/page_alloc.c* 中的 *free_pages()*)

当一页块被释放时，系统会检查它的相邻块或者同样大小的 buddy 块(编者注：sorry，实在没能力把握 buddy block 的准确意思)，看它们是否是空闲的。如果是，它们将结合成一个大小为原来两倍的整块。每次当两块内存被拼成了更大的空闲块时，页回收代码将尝试把它们继续与其它空闲块进行组合，以得到更大的空间。在此种方式下，与存储器使用量等量的空闲页块是可以的。

例如，在图 3.1 中，如果第 1 页被释放，那它将与已经是空闲的第 0 页组合，并作为一个大小为 2 的空闲页块，被放到 free_area 的保存大小为两页的空闲块的队中。

3.5 内存映射

当一映像被执行时，它的内容必须被读入进程的虚存地址空间。它所链接并使用的一些共享库也必须被读入虚存。这个可执行文件并非被实际读入内存，相反它只是被连接入进程的虚存。然后，当程序的一部份被运行中的应用程序调用时，系统才将这部份映像读入内存。像这样将映像连接到进程的虚地址空间叫做内存映射(memory mapping)。

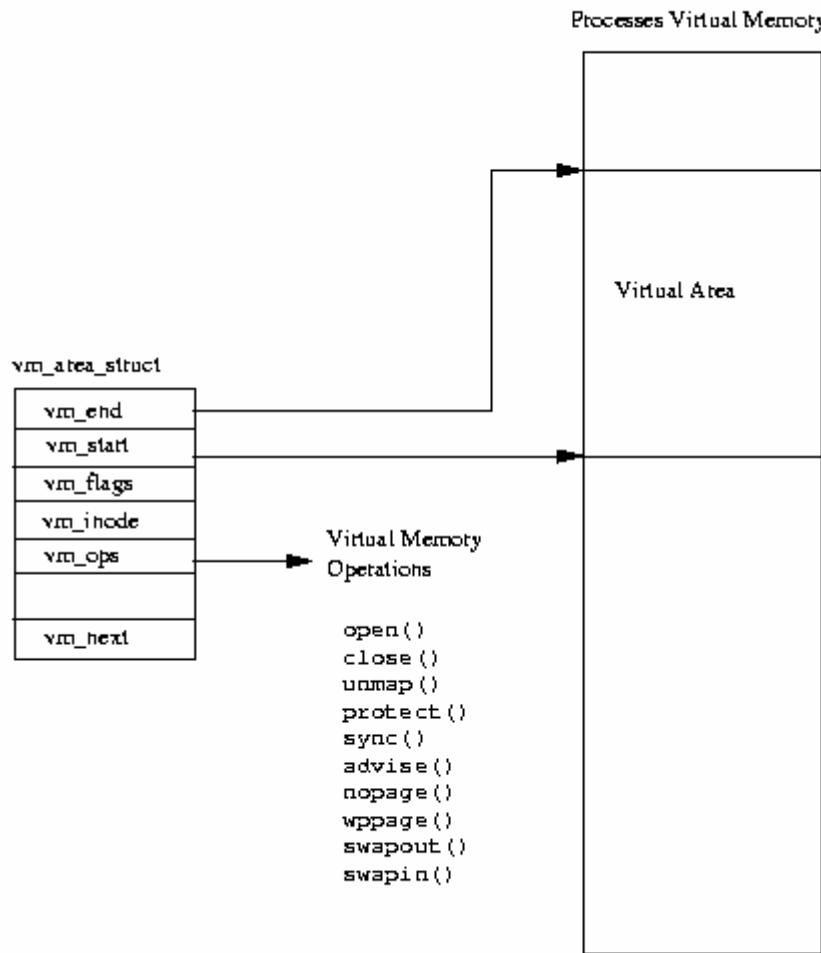


Figure 3.5: Areas of Virtual Memory

Every process's virtual memory is represented by an `mm_struct` data structure. This contains information about the image that it is currently executing (for example `bash`) and also has pointers to a number of `vm_area_struct` data structures. Each `vm_area_struct` data structure describes the start and end of the area of virtual memory, the process's access rights to that memory and a set of operations for that memory. These operations are a set of routines that Linux must use when manipulating this area of virtual memory. For example, one of the virtual memory operations performs the correct actions when the process has attempted to access this virtual memory but finds (via a page fault) that the memory is not actually in physical memory. This operation is the `nopage` operation. The `nopage` operation is used when Linux demands pages the pages of an executable image into memory.

每个进程的虚存空间由一个 `mm_struct` 数据结构表示。其中包含了当前正在执行的映像（例如 `bash`）的信息，以及很多指向 `vm_area_struct` 这一数据结构的指针。每个 `vm_area_struct` 数据结构描述了一段虚存区域的开始和结束，及进程对那段虚存的存取权限和允许的操作。这些操作是 Linux 在维护这段虚存必须使用的一套例程。例如，当进程试图存取虚存中某页，但发现这页并不在内存中时，应执行的正确操作是 `nopage` 操作（通过页错）。`nopage` 操作在 Linux 需要将一些可执行映像的页载入内存中时使用。

When an executable image is mapped into a process's virtual address space, a set of `vm_area_struct` data structures is generated. Each `vm_area_struct` data structure represents a part of the executable image; the executable code, initialized data (variables), uninitialized data and so on. Linux supports a number of standard virtual memory operations and as the `vm_area_struct` data structures are created, the correct set of virtual memory operations are associated with them.

3.6 Demand Paging

Once an executable image has been memory mapped into a process's virtual memory, it can start to execute. As only the very start of the image is physically pulled into memory, it will soon access an area of virtual memory that is not yet in physical memory. When a process accesses a virtual address that does not have a valid page table entry, the processor will report a page fault to Linux. (See `handle_mm_fault()` in `mm/memory.c`)

The page fault describes the virtual address where the page fault occurred and the type of memory access that caused it.

Linux must find the `vm_area_struct` that represents the area of memory that the page fault occurred in. As searching through the `vm_area_struct` data structures is critical to the efficient handling of page faults, these are linked together in an AVL (Adelson-Velskii and Landis) tree structure. If there is no `vm_area_struct` data structure for this faulting virtual address, this process has accessed an illegal virtual address. Linux will signal the process, sending a SIGSEGV signal, and if the process does not have a handler for that signal, it will be terminated.

Linux next checks the type of page fault that occurred against the types of accesses allowed for this area of virtual memory. If the process is accessing the memory in an illegal way, say writing to an area that it is only allowed to read from, it is also signalled with a memory error.

Now that Linux has determined that the page fault is legal, it must deal with it.

Linux must differentiate between pages that are in the swap file and those that are part of an executable

<http://emag.csdn.net>

<http://purec.binghua.com>

当一段可执行映像被映射入进程的虚存时，会产生一组 `vm_area_struct` 数据结构。每个 `vm_area_struct` 数据结构可表示可执行映像的一部份：可执行代码、初始化数据（变量）、未初始化数据等等。Linux 支持很多标准的虚存操作，当 `vm_area_struct` 数据结构产生时，系统会把正确的虚存操作集与他们相关联。

3.6 按需换页

当一部份可执行映像被映射入进程虚存后，它就可以开始执行了。可是这时只有映像的开始部份被实际读入内存，它将很快访问不在内存中的部份。当进程存取一个没有有效页表记录的虚地址时，那处理器将报告一个页错误给 Linux 系统。（参见 `mm/memory.c` 中的 `handle_mm_fault()`）

页错误描述页错发生的虚地址和引起页错的访存方式。

Linux 必须先找到代表页错发生区域的 `vm_area_struct`。由于搜索 `vm_area_struct` 数据结构对高效处理页错非常关键，所以所有 `vm_area_struct` 被连接成 AVL (Adelson-Velskii and Landis) 树结构（编者注：即平衡二叉树结构）。如果没有 `vm_area_struct` 代表这页错发生的虚地址，表示这进程企图访问一个非法的虚地址。Linux 将发送 SIGSEGV 信号给进程，如果进程没有这个信号的处理程序，那么该进程将被终止。

Linux 再检查引发这页错的存取操作是否是被允许的。如果进程在用一个非法的方式存取内存，例如，写一个只读区域，这也将引发一个内存错误信号。

如果 Linux 确定页错是合法的，那么 Linux 就必须处理它。

Linux 必须首先区别映像是在交换文件中还是在磁盘上。它是通过页错发生的虚地址所对应的页表

image on a disk somewhere. It does this by using the page table entry for this faulting virtual address. (*See do_no_page() in mm/memory.c*)

If the page's page table entry is invalid but not empty, the page fault is for a page currently being held in the swap file. For Alpha AXP page table entries, these are entries which do not have their valid bit set but which have a non-zero value in their PFN field. In this case the PFN field holds information about where in the swap (and which swap file) the page is being held. How pages in the swap file are handled is described later in this chapter.

Not all `vm_area_struct` data structures have a set of virtual memory operations and even those that do may not have a *nopage* operation. This is because by default Linux will fix up the access by allocating a new physical page and creating a valid page table entry for it. If there is a *nopage* operation for this area of virtual memory, Linux will use it.

The generic Linux *nopage* operation is used for memory mapped executable images and it uses the page cache to bring the required image page into physical memory.

However the required page is brought into physical memory, the processes page tables are updated. It may be necessary for hardware specific actions to update those entries, particularly if the processor uses translation look aside buffers. Now that the page fault has been handled it can be dismissed and the process is restarted at the instruction that made the faulting virtual memory access. (*See filemap_nopage() in mm/filemap.c*)

记录来识别的。（参见 `mm/memory.c` 中的 `do_no_page()`）

如果那页的页表记录是无效的，但非空，说明产生页错的那页当前存在于交换文件中。Alpha AXP 页表记录中，这样的记录的有效位未置位，但是 PFN 域不为零。在这种情况下，PFN 域含有的信息表示了这页被保存在哪个交换文件中的哪个位置。本章的后面部份将讲述怎样处理在交换文件中的页。

并非所有的 `vm_area_struct` 数据结构都有一组虚存操作，即使有，也不一定有 *nopage* 操作。这是因为，在缺损情况下，Linux 将分配一页新内存，并为这页增加一项页表记录，Linux 以此来修正这个访问错误。但如果这段虚存有 *nopage* 操作，Linux 将使用它。

通常，Linux 的 *nopage* 操作被用于把可执行映像通过页缓存读入物理内存。

当页被读入内存后，进程的页表将被更新。特别是如果处理器使用 TLB 缓冲区的话，它可能需要通过硬件操作来完成更新。页错被处理后，进程在发生虚拟内存访问错误的指令处（编者注：即产生页错的指令处）重新开始执行。（编者注：这里有个 TLB，可能大家不太熟悉，这里简单提一下：TLB 是 *translation look aside buffer* 的缩写，它是 CPU 在内部做的一块专用存储器一样的东东（当然还有一些其它控制逻辑），它主要用来缓存一小部分页表记录，这样它就可以专门用于加快虚拟地址到物理地址的转换，因为有了它之后，CPU 不需要每次都从外部存储器中读取页表记录了。故而，一些资料上常将其译为“快速地址转换表”。）
(参见 `mm/filemap.c` 中的 `filemap_nopage()`)

3.7 The Linux Page Cache

3.7 Linux 页缓存

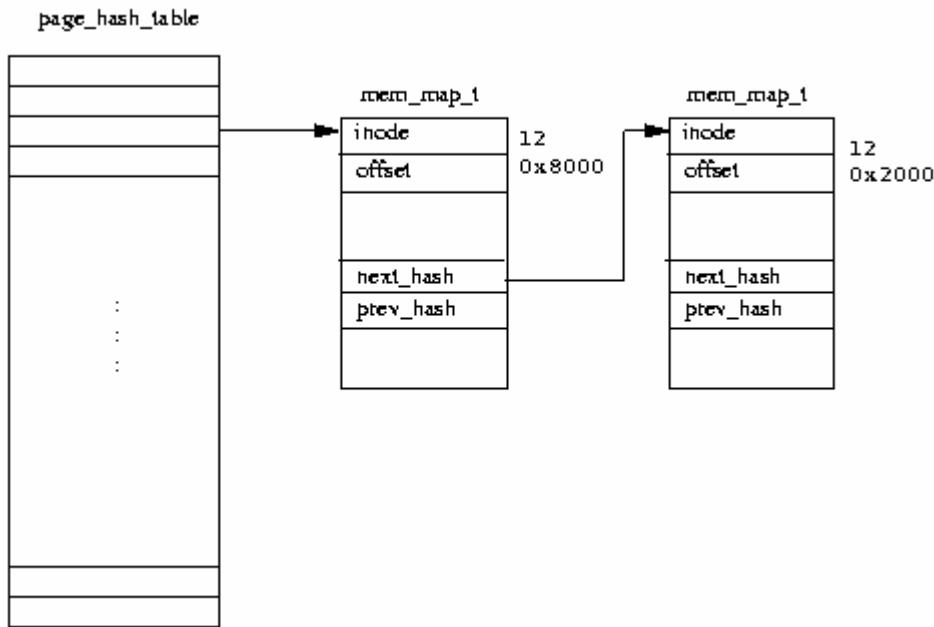


Figure 3.6: The Linux Page Cache

The role of the Linux page cache is to speed up access to files on disk. Memory mapped files are read a page at a time and these pages are stored in the page cache. Figure 3.6 shows that the page cache consists of the page_hash_table, a vector of pointers to mem_map_t data structures. (See *include/linux/pagemap.h*)

Each file in Linux is identified by a VFS inode data structure (described in Chapter filesystem-chapter) and each VFS inode is unique and fully describes one and only one file. The index into the page table is derived from the file's VFS inode and the offset into the file.

Whenever a page is read from a memory mapped file, for example when it needs to be brought back into memory during demand paging, the page is read through the page cache. If the page is present in the cache, a pointer to the mem_map_t data structure representing it is returned to the page fault handling code. Otherwise the page must be brought into memory from the file system that holds the image. Linux allocates a physical page and reads the file from the file on disk.

If it is possible, Linux will initiate a read of the next page in the file. This single page read ahead means that if the process is accessing the pages in the file serially, the next page will be waiting in memory for the process.

Linux 页缓存的作用是加快从磁盘上存取文件的速度。内存映射文件以每次一页的方式读出，这些页将被放在页缓存中。图 3.6 显示页缓存由 page_hash_table，以及一组指向 mem_map_t 的指针组成。(参见 *include/linux/pagemap.h*)

Linux 的每个文件由一 VFS inode 数据结构表示 (请参看“文件系统”一章)，并且每个 VFS inode 是唯一的并且描述一个且仅一个文件。页表中的索引从文件的 VFS inode 及其在文件中的偏移量派生而来。

当从内存映像文件中读取一页时，例如，按需装载一页回内存时，读操作将通过页缓存进行。如果页在缓存中，一个指向它的 mem_map_t 指针将被返回给处理页错的代码。否则，这页必须从含有这份映象的文件系统中被读入内存。Linux 需分配一页内存，并从磁盘文件中读取这页。(编者注：这两句话直译很拗口，其实它的意思就是如果这页不是在页缓存中，则系统必须把它从存放这页的磁盘文件中读取出来。)

如果可能，Linux 将开始读文件中的下一页。向前多读一页意味着如果进程是连续地访问文件，那么下一页将等在内存中。

Over time the page cache grows as images are read and executed. Pages will be removed from the cache as they are no longer needed, say as an image is no longer being used by any process. As Linux uses memory it can start to run low on physical pages. In this case Linux will reduce the size of the page cache.

3.8 Swapping Out and Discarding Pages

When physical memory becomes scarce the Linux memory management subsystem must attempt to free physical pages. This task falls to the kernel swap daemon (kswapd). (*See kswapd() in mm/vmscan.c*)

The kernel swap daemon is a special type of process, a kernel thread. Kernel threads are processes have no virtual memory, instead they run in kernel mode in the physical address space. The kernel swap daemon is slightly misnamed in that it does more than merely swap pages out to the system's swap files. Its role is make sure that there are enough free pages in the system to keep the memory management system operating efficiently.

The Kernel swap daemon (kswapd) is started by the kernel init process at startup time and sits waiting for the kernel swap timer to periodically expire.

Every time the timer expires, the swap daemon looks to see if the number of free pages in the system is getting too low. It uses two variables, `free_pages_high` and `free_pages_low` to decide if it should free some pages. So long as the number of free pages in the system remains above `free_pages_high`, the kernel swap daemon does nothing; it sleeps again until its timer next expires. For the purposes of this check the kernel swap daemon takes into account the number of pages currently being written out to the swap file. It keeps a count of these in `nr_async_pages`; this is incremented each time a page is queued waiting to be written out to the swap file and decremented when the write to the swap device has completed. `free_pages_low` and `free_pages_high` are set at system startup time and are related to the number of physical pages in the system. If the number of free pages in the system has fallen below `free_pages_high` or worse still `free_pages_low`, the

页缓存将随着映象的读取与执行而渐渐增长。当不再被需要, 或说不再被任何进程使用时, 这些页将从缓存中移出。Linux 使用内存时, 会尽量减少物理页的使用。在此种情况下, Linux 将减少页缓存的大小。

3.8 页的交换和释放

当空内存变得很少时, Linux 内存管理系统必须释放一些页。这任务由内核交换后台程序来完成 (kswapd)。(参见 `mm/vmscan.c` 中的 `kswapd()`)

内核交换后台程序是一种特殊的进程, 是一个内核线程。内核线程是没有虚存的进程, 它们在物理地址空间以内核模式运行。“内核交换后台程序”这个名称稍微有点不恰当, 因为它不仅仅是把页交换到系统的交换文件中。它这个角色是保证系统有足够的空闲内存而使内存管理系统可以高效地工作。

内核交换后台程序被内核 init 进程在初始化时启动, 并等待内核交换定时器周期性地到期时开始运行。

每次定时器到期, 内核交换后台程序就会检查系统中的空页数是否变得太低。交换程序使用两个变量, `free_pages_high` 和 `free_pages_low` 来决定是否它应该释放一些页。只要系统的空页数大于 `free_pages_high`, 内核交换后台程序不做任何事情; 它继续休息直到定时器再次到期。在做这项检查时, 交换程序计算了正在往交换文件中写的页数。它把这个值保存在 `nr_async_pages` 中, 每次有一页等待写入交换文件时, 此值加 1, 当操作结束后, 此值减 1。`free_pages_low` 和 `free_pages_high` 在系统开始时被设置, 并且与系统物理内存的页数有关。如果系统的空页数小于 `free_pages_high` 或甚至小于 `free_pages_low`, 内核交换后台程序将尝试 3 种方法以减少系统使用的页数:

kernel swap daemon will try three ways to reduce the number of physical pages being used by the system:

- Reducing the size of the buffer and page caches,
- Swapping out System V shared memory pages,
- Swapping out and discarding pages.

If the number of free pages in the system has fallen below free_pages_low, the kernel swap daemon will try to free 6 pages before it next runs. Otherwise it will try to free 3 pages. Each of the above methods are tried in turn until enough pages have been freed. The kernel swap daemon remembers which method it was using the last time that it attempted to free physical pages. Each time it runs it will start trying to free pages using this last successful method.

After it has free sufficient pages, the swap daemon sleeps again until its timer expires. If the reason that the kernel swap daemon freed pages was that the number of free pages in the system had fallen below free_pages_low, it only sleeps for half its usual time. Once the number of free pages is more than free_pages_low the kernel swap daemon goes back to sleeping longer between checks.

3.8.1 Reducing the Size of the Page and Buffer Caches

The pages held in the page and buffer caches are good candidates for being freed into the free_area vector. The Page Cache, which contains pages of memory mapped files, may contain unnecessary pages that are filling up the system's memory. Likewise the Buffer Cache, which contains buffers read from or being written to physical devices, may also contain unneeded buffers. When the physical pages in the system start to run out, discarding pages from these caches is relatively easy as it requires no writing to physical devices (unlike swapping pages out of memory). Discarding these pages does not have too many harmful side effects other than making access to physical devices and memory mapped files slower. However, if the discarding of pages from these caches is done fairly, all processes will suffer equally. (See `shrink_map()` in `mm/filemap.c`)

- 减少缓冲区和页缓存的大小
- 换出系统 V 的共享页
- 换出并释放一些页

如果系统的空页数小于 `free_pages_low`, 内核交换后台程序在它下次运行以前, 将尝试释放 6 页, 否则它将尝试释放 3 页。上面的方法将依次被使用直到有足够的页被释放。内核交换后台程序将记住上一次它是用什么方法释放内存的, 下一次将首先使用这个成功的方法。

在系统有足够的空页后, 交换程序将休息直到它的定时器到期。如果上次空页数小于 `free_pages_low`, 它只休息一半时间。直到空页数多于 `free_pages_low`, 内核交换后台程序才恢复休息的时间。

3.8.1 减少页缓存和缓冲区的大小

页缓存和缓冲区中的页是被释放到 `free_area` 数组里的最佳候选。页缓存保存着内存映像文件, 很可能包括了许多占据着系统内存但又没用的页。同样, 包含着从物理设备中读写的数据的缓冲区中, 也可能包含许多不需要的数据缓存。当系统的内存页快用完时, 从这些缓存丢弃页是相对容易的, 因为它们不需要写物理设备(不同于从内存交换页)。丢弃这些页除了使访问物理设备和内存映象文件的速度减慢一些以外, 没有其它的副作用。并且, 如果从缓存中对页的丢弃是公平的话, 那么对各进程的影响是相同的。(参见 `mm/filemap.c` 中的 `shrink_map()`)

Every time the Kernel swap daemon tries to shrink these caches it examines a block of pages in the mem_map page vector to see if any can be discarded from physical memory. The size of the block of pages examined is higher if the kernel swap daemon is intensively swapping; that is if the number of free pages in the system has fallen dangerously low. The blocks of pages are examined in a cyclical manner; a different block of pages is examined each time an attempt is made to shrink the memory map. This is known as the *clock* algorithm as, rather like the minute hand of a clock, the whole mem_map page vector is examined a few pages at a time.

Each page being examined is checked to see if it is cached in either the page cache or the buffer cache. You should note that shared pages are not considered for discarding at this time and that a page cannot be in both caches at the same time. If the page is not in either cache then the next page in the mem_map page vector is examined.

Pages are cached in the buffer cache (or rather the buffers within the pages are cached) to make buffer allocation and deallocation more efficient. The memory map shrinking code tries to free the buffers that are contained within the page being examined.

If all the buffers are freed, then the pages that contain them are also be freed. If the examined page is in the Linux page cache, it is removed from the page cache and freed. (*See free_buffer() in fs/buffer.c*)

When enough pages have been freed on this attempt then the kernel swap daemon will wait until the next time it is periodically woken. As none of the freed pages were part of any process's virtual memory (they were cached pages), then no page tables need updating. If there were not enough cached pages discarded then the swap daemon will try to swap out some shared pages.

3.8.2 Swapping Out System V Shared Memory Pages

System V shared memory is an inter-process communication mechanism which allows two or more processes to share virtual memory in order to pass

<http://emag.csdn.net>

<http://purec.binghua.com>

每次内核交换后台程序尝试缩小这些缓存时，它先检查在 mem_map 页面数组中的页块，看是否有页可以从内存中释放。如果内核交换后台程序经常作交换操作，也就是系统空页数已经非常少了，它会先检查大一些的块。页块会被轮流检查；每次减少缓存时检查一组不同的页块。这被称作“时钟算法”，像钟的分针一样轮流检查 mem_map 页面数组中的页。

检查一页是看它是否在页缓存或缓冲区中。应该注意共享页在这时候不能被释放，并且一页不能同时在两个缓存中。如果页不在任何一个缓存中，那么就检查 mem_map 页面数组中的下一页。

页被缓存在缓冲区中（或页内的缓冲区被缓存）是为更有效地分配和回收缓存。缩减内存代码将尝试释放被检查页中的缓冲区。

如果所有的缓冲区都被释放了，那么对应它们的内存也就被释放了。如果被检查的页在 Linux 页缓存中，它将被从页缓存中移出并释放。（参见 *fs/buffer.c* 中的 *free_buffer()*）

当足够的页被释放后，内核交换后台程序将等到下一个周期再运行。因为释放的页都不是任何进程的虚存部份（他们是被缓存的页），所以没有页表记录需要更新。如果没有释放足够的缓存页，那么交换程序将试着释放一些共享页。

3.8.2 交换出系统 V 的共享页

系统 V 共享内存是一个进程内通信机制，它允许两个或多个进程共享虚拟内存，以便于在它们之间传递信息。进程间如何通过这种方式共享内存，将会在

information amongst themselves. How processes share memory in this way is described in more detail in Chapter IPC-chapter. For now it is enough to say that each area of System V shared memory is described by a shmid_ds data structure. This contains a pointer to a list of vm_area_struct data structures, one for each process sharing this area of virtual memory. The vm_area_struct data structures describe where in each processes virtual memory this area of System V shared memory goes. Each vm_area_struct data structure for this System V shared memory is linked together using the vm_next_shared and vm_prev_shared pointers. Each shmid_ds data structure also contains a list of page table entries each of which describes the physical page that a shared virtual page maps to.

The kernel swap daemon also uses a clock algorithm when swapping out System V shared memory pages.

Each time it runs it remembers which page of which shared virtual memory area it last swapped out. It does this by keeping two indices, the first is an index into the set of shmid_ds data structures, the second into the list of page table entries for this area of System V shared memory. This makes sure that it fairly victimizes the areas of System V shared memory. (*See shm_swap() in ipc/shm.c*)

As the physical page frame number for a given virtual page of System V shared memory is contained in the page tables of all of the processes sharing this area of virtual memory, the kernel swap daemon must modify all of these page tables to show that the page is no longer in memory but is now held in the swap file. For each shared page it is swapping out, the kernel swap daemon finds the page table entry in each of the sharing processes page tables (by following a pointer from each vm_area_struct data structure). If this processes page table entry for this page of System V shared memory is valid, it converts it into an invalid but swapped out page table entry and reduces this (shared) page's count of users by one. The format of a swapped out System V shared page table entry contains an index into the set of shmid_ds data structures and an index into the page table entries for this area of System V shared memory.

IPC 章中详细描述。现在，只要知道每一块系统 V 共享内存区域被一个 shmid_ds 数据结构描述就足够了。这个结构包含一个指向一组 vm_area_struct 数据结构的指针，每个进程都通过它共享这部分虚存。vm_area_struct 数据结构描述了每个进程在各自虚存的哪里共享系统 V 的这个区域。每个 vm_area_struct 由 vm_next_shared 和 vm_prev_shared 指针相互连接起来。每个 shmid_ds 数据结构还包括一组页表记录，每个页表记录描述了这些共享页是对应内存中的哪些物理页。

内核交换后台程序也使用时钟算法来换出系统 V 的共享页。

每次它运行时，它记得上次换出的是哪个共享内存区域中的哪一页。它将其记录在两个索引中，第一个是 shmid_ds 数据结构的索引，第二个是这段系统 V 共享内存的页表记录的索引。这保证它公平地对待系统 V 的所有共享页。（参见 *ipc/shm.c* 中的 *shm_swap()*）

由于给定的系统 V 共享内存物理页号在每一个共享此内存区域的进程的页表中都有记录，内核交换后台程序必须修改所有这些页表，显示页已不在内存中了，而被保存在交换文件中。对于每个换出的共享页，内核交换后台程序查找这共享页在各个进程中的页表记录（顺着每个 vm_area_struct 的指针）。如果这系统 V 共享页对应的页表记录是有效的，交换程序将把它改成无效，换出页表项，再将对应这页（共享页）的用户计数器减 1。被换出的系统 V 共享页的页表记录格式中含有一个 shmid_ds 数据结构的索引，以及一个这段系统 V 共享内存区域的索引。

If the page's count is zero after the page tables of the sharing processes have all been modified, the shared page can be written out to the swap file. The page table entry in the list pointed at by the shmid_ds data structure for this area of System V shared memory is replaced by a swapped out page table entry. A swapped out page table entry is invalid but contains an index into the set of open swap files and the offset in that file where the swapped out page can be found. This information will be used when the page has to be brought back into physical memory.

3.8.3 Swapping Out and Discarding Pages

The swap daemon looks at each process in the system in turn to see if it is a good candidate for swapping.

Good candidates are processes that can be swapped (some cannot) and that have one or more pages which can be swapped or discarded from memory. Pages are swapped out of physical memory into the system's swap files only if the data in them cannot be retrieved another way. (*See swap_out() in mm/vmscan.c*)

A lot of the contents of an executable image come from the image's file and can easily be re-read from that file. For example, the executable instructions of an image will never be modified by the image and so will never be written to the swap file. These pages can simply be discarded; when they are again referenced by the process, they will be brought back into memory from the executable image.

Once the process to swap has been located, the swap daemon looks through all of its virtual memory regions looking for areas which are not shared or locked.

Linux does not swap out all of the swappable pages of the process that it has selected; instead it removes only a small number of pages.

Pages cannot be swapped or discarded if they are locked in memory. (*See swap_out_vma() in mm/vmscan.c*)

The Linux swap algorithm uses page aging. Each page has a counter (held in the mem_map_t data structure) that gives the Kernel swap daemon some idea

<http://emag.csdn.net>

<http://purec.binghua.com>

如果各进程的页表修改过后，页的计数器变成 0，那么这个共享页就可以被写入交换文件了。shmid_ds 中指向系统V共享内存页的页表记录将被换出页表记录所替换。一个换出页表记录是无效的，但它包含一组打开的交换文件的索引，以及换出页面在交换文件中的偏移量。当这页面重新被载入物理内存时，这些信息会被使用到。

3.8.3 换出及释放的页

交换程序轮流检查系统中每一个进程，看它们是不是被用来交换的好候选。

好的候选是那些能被(有的不能)换出的进程以及那些能从内存中换出或释放若干页的进程。只有包含的数据不能从其它地方得到的页，才会从物理内存中交换到系统的交换文件中。(参见 *mm/vmscan.c* 中的 *swap_out()*)

可执行映像的许多内容是可以从映像文件中读出并且可以很容易的重新读出来的。例如，一段映像的可执行指令决不会被映象修改，所以决不会被写进交换文件。这些页可以简单的被丢弃；当他们再被进程调用时，他们将被从可执行映像中重新读入内存。

一旦确定了换出的进程，交换程序将检查它所有的虚拟内存区域，找出不是共享或被加锁的区域。

Linux 并不换出它所选择进程的所有可交换页；相反它仅移出其中的一小部份。

如果页在内存中被锁住了，它们就不能被换出或释放。(参见 *mm/vmscan.c* 中的 *swap_out_vma()*)

Linux 交换算法使用页的年龄(aging)。每页有一个记数器(保存在 mem_map_t 数据结构中)，告诉交换程序是否应将它移出。当页不使用时页会变老；当

whether or not a page is worth swapping. Pages age when they are unused and rejuvenate on access; the swap daemon only swaps out old pages. The default action when a page is first allocated, is to give it an initial age of 3. Each time it is touched, its age is increased by 3 to a maximum of 20. Every time the Kernel swap daemon runs it ages pages, decrementing their age by 1. These default actions can be changed and for this reason they (and other swap related information) are stored in the swap_control data structure.

If the page is old ($age = 0$), the swap daemon will process it further. Dirty pages are pages which can be swapped out. Linux uses an architecture specific bit in the PTE to describe pages this way (see Figure 3.2). However, not all dirty pages are necessarily written to the swap file. Every virtual memory region of a process may have its own swap operation (pointed at by the `vm_ops` pointer in the `vm_area_struct`) and that method is used. Otherwise, the swap daemon will allocate a page in the swap file and write the page out to that device.

The page's page table entry is replaced by one which is marked as invalid but which contains information about where the page is in the swap file. This is an offset into the swap file where the page is held and an indication of which swap file is being used. Whatever the swap method used, the original physical page is made free by putting it back into the `free_area`. Clean (or rather not dirty) pages can be discarded and put back into the `free_area` for re-use.

If enough of the swappable processes pages have been swapped out or discarded, the swap daemon will again sleep. The next time it wakes it will consider the next process in the system. In this way, the swap daemon nibbles away at each processes physical pages until the system is again in balance. This is much fairer than swapping out whole processes.

3.9 The Swap Cache

When swapping pages out to the swap files, Linux avoids writing pages if it does not have to. There are times when a page is both in a swap file and in physical memory. This happens when a page that was swapped out of memory was then brought back into memory when it was again accessed by a process. So long as the

被访问时，会变年轻。交换程序仅仅移出衰老的页。缺省状态下，当一页被分配时，起始年龄是 3，每次它被访问，它的年龄将增加 3，最大值为 20。每次内核交换后台程序运行时，它把所有页的年龄数减 1。这些缺省操作都能被改变，它们（以及一些其它的与交换相关的信息）被存储在 `swap_control` 数据结构中。

如果页是旧的($age = 0$)，交换程序就进一步处理它。脏页也可以被移出。Linux 用 PTE 中的特定位来标示(见 3.2 图)。然而，并非所有的脏页必须被写进交换文件。进程的每个虚存区域都可以有它们自己的交换操作(由 `vm_area_struct` 中的 `vm_ops` 指针指出)，这个特定的操作将被调用。否则，交换程序将在交换文件上分配一页，并将那页写到磁盘上。

页对应的页表记录将被改为无效，但包含了它在交换文件中的信息，它将指出是它存在于哪个交换文件，并且偏移量是多少。无论采取什么交换方法，原来的物理页将被放回 `free_area`。干净的(或者 not dirty)的页可以直接丢弃并放回 `free_area` 以备后用。

如果有足够的页被换出或释放，交换程序就又开始休息。下一次它运行时，它将检查系统中的下一个进程。这样，交换程序一点一点地将每个进程都移出几页，直到系统达到一个平衡，这比移出一整个进程来的公平。

3.9 交换缓存

当将页移入交换文件中时，如果不是必要，Linux 总是避免进行写页操作。有时一页既在交换文件中，又在内存中。这种情况是由于这页本来被移到了交换文件中，后又因为被调用，又被重新读入内存。只要在内存中的页没被写过，在交换文件中的拷贝仍然是有效。

page in memory is not written to, the copy in the swap file remains valid.

Linux uses the swap cache to track these pages. The swap cache is a list of page table entries, one per physical page in the system. This is a page table entry for a swapped out page and describes which swap file the page is being held in together with its location in the swap file. If a swap cache entry is non-zero, it represents a page which is being held in a swap file that has not been modified. If the page is subsequently modified (by being written to), its entry is removed from the swap cache.

When Linux needs to swap a physical page out to a swap file it consults the swap cache and, if there is a valid entry for this page, it does not need to write the page out to the swap file. This is because the page in memory has not been modified since it was last read from the swap file.

The entries in the swap cache are page table entries for swapped out pages. They are marked as invalid but contain information which allow Linux to find the right swap file and the right page within that swap file.

3.10 Swapping Pages In

The dirty pages saved in the swap files may be needed again, for example when an application writes to an area of virtual memory whose contents are held in a swapped out physical page. Accessing a page of virtual memory that is not held in physical memory causes a page fault to occur. The page fault is the processor signalling the operating system that it cannot translate a virtual address into a physical one. In this case this is because the page table entry describing this page of virtual memory was marked as invalid when the page was swapped out. The processor cannot handle the virtual to physical address translation and so hands control back to the operating system describing as it does so the virtual address that faulted and the reason for the fault. The format of this information and how the processor passes control to the operating system is processor specific.

The processor specific page fault handling code must locate the `vm_area_struct` data structure that describes the area of virtual memory that contains the

<http://emag.csdn.net>

<http://purec.binghua.com>

- 44 -

与处理器相关的页错处理代码必须找到引起页错的虚地址对应的 `vm_area_struct` 数据结构。在这个过程中，系统检索该进程所有的 `vm_area_struct`

2005 年 1 月第 1 期 总第 3 期

Linux 使用交换缓存来记录这些页。交换缓存是一个页表记录链表，每条记录对应一页。每条页表记录描述被换出的页在哪个交换文件中及其在文件中的位置。如果一个交换缓存记录非零，表示在交换文件中的那页没被修改过，如果页被修改了（被写），它的记录将被从交换缓存中移出。

当 Linux 需要移出一页内存到交换文件中时，它先查询交换缓存，如果这页有一个有效的记录，它就不需要把页写到交换文件中了。因为自从它上次从交换文件中读出后，在内存中没被修改过。

交换缓存中的记录是已被交换出的页的页表记录。它们被标为无效，但是告知了 Linux 页在哪个交换文件以及在交换文件的哪个位置。

3.10 移入页

保存在交换文件中的脏页可能会被再次调用。例如，一个应用程序要向已交换出物理页面的虚拟内存区上写入时。这样，存取不在内存中的虚页将引起页错。页错误是由处理器发信号给操作系统，告诉操作系统它不能把某个虚地址翻译成物理地址，由于描述这个虚页在被交换出时，页表记录已被标记为无效，处理器不能处理虚拟地址到物理地址的转换，于是，处理器把控制权交还给操作系统。同时告诉操作系统发生页错的虚拟地址及原因。消息的格式以及处理器怎样把控制权交给操作系统这是与处理器相关的。

faulting virtual address. It does this by searching the `vm_area_struct` data structures for this process until it finds the one containing the faulting virtual address. This is very time critical code and a processes `vm_area_struct` data structures are so arranged as to make this search take as little time as possible. (*See do_page_fault() in arch/i386/mm/fault.c*)

Having carried out the appropriate processor specific actions and found that the faulting virtual address is for a valid area of virtual memory, the page fault processing becomes generic and applicable to all processors that Linux runs on.

The generic page fault handling code looks for the page table entry for the faulting virtual address. If the page table entry it finds is for a swapped out page, Linux must swap the page back into physical memory. The format of the page table entry for a swapped out page is processor specific but all processors mark these pages as invalid and put the information necessary to locate the page within the swap file into the page table entry. Linux needs this information in order to bring the page back into physical memory. (*See do_no_page() in mm/memory.c*)

At this point, Linux knows the faulting virtual address and has a page table entry containing information about where this page has been swapped to. The `vm_area_struct` data structure may contain a pointer to a routine which will swap any page of the area of virtual memory that it describes back into physical memory. This is its swapin operation. If there is a swapin operation for this area of virtual memory then Linux will use it. This is, in fact, how swapped out System V shared memory pages are handled as it requires special handling because the format of a swapped out System V shared page is a little different from that of an ordinary swapped out page. There may not be a swapin operation, in which case Linux will assume that this is an ordinary page that does not need to be specially handled.

It allocates a free physical page and reads the swapped out page back from the swap file. Information telling it where in the swap file (and which swap file) is taken from the the invalid page table entry. (*See do_swap_page() in mm/memory.c; shm_swap_in() in*

数据结构直到找到为止。这段代码对时间的要求很高，所以 `vm_area_struct` 应被合理组织起来，以缩短查找所需的时间。(参见 `arch/i386/mm/fault.c` 中的 `do_page_fault()`)

系统执行完这些与处理器相关的操作并找到引起页错的虚地址所代表的有效内存区域后，处理页错的其它代码是通用并且与运行 Linux 的处理器无关的了。

页错处理代码寻找引发页错的虚地址对应的页表记录。如果页表记录指示这页在交换文件中，Linux 就必须把这页读回物理内存。页表记录的格式因处理器的不同而各不相同，但所有的处理器都会标记此页无效，并且都保存着有关这页在交换文件中的有用的信息。Linux 需要利用这些信息来把页重新载入内存。
(参见 `mm/memory.c` 中的 `do_no_page()`)

此时，Linux 知道了引起页错的虚地址及其对应的页表记录，并且拥有一个包含此页被交换到哪个交换文件中的信息的页表记录。`vm_area_struct` 数据结构可能包含一个指向一个例程的指针，这个例程能将此虚拟内存中的任何页交换到物理内存中去。这是 swapin 操作。如果此虚拟内存区域存在 swapin 操作，Linux 就会调用它。实际上，这是怎样换出系统 V 共享内存页的操作，这个操作需要特殊处理，因为系统 V 的页的格式与一般的页不同。这里也可能没有 swapin 操作，在这种情况下，Linux 将认为它是一普通的页，而不需要做任何特别的处理。

系统将在物理内存中分配一空页，并从交换文件中把交换同的页读回来。而关于页面在交换文件中的位置信息(以及在哪个交换文件中)是从无效的页表记录中取回的。(参见 `mm/memory.c` 中的 `do_swap_page()`、`ipc/shm.c` 中的 `shm_swap_in()`、

ipc/shm.c; swap_in() in mm/page_alloc.c)

If the access that caused the page fault was not a write access then the page is left in the swap cache and its page table entry is not marked as writable. If the page is subsequently written to, another page fault will occur and, at that point, the page is marked as dirty and its entry is removed from the swap cache. If the page is not written to and it needs to be swapped out again, Linux can avoid the write of the page to its swap file because the page is already in the swap file.

If the access that caused the page to be brought in from the swap file was a write operation, this page is removed from the swap cache and its page table entry is marked as both dirty and writable.

mm/page_alloc.c 中的 swap_in()

如果引起页错的不是写操作，那么这页将被留在交换缓存中，它的页表记录不会被标为“可写”。如果后来这页被写了，那么会产生另一个页错，这时，页被标成“dirty”，并且它的页表记录被从交换缓冲中删去。如果这页没被修改过，而它又需要被换出，Linux 将避免再把这页写到交换文件中，因为它已经在那儿了。

如果引起页面从交换文件中读出的操作是写操作，页将被从交换缓存中删除，它的页表记录将被标成“脏的(dirty)”和“可写(writable)”。

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

【责任编辑: cliff@hitbbs】

用 GDB 调试程序

作者：陈皓(haoel@hotmail.com)

1. GDB 概述

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具。或许，各位比较喜欢那种图形界面方式的，像 VC、BCB 等 IDE 的调试，但如果你是在 UNIX 平台下做软件，你会发现 GDB 这个调试工具有比 VC、BCB 的图形化调试器更强大的功能。所谓“寸有所长，尺有所短”就是这个道理。

一般来说，GDB 主要帮忙你完成下面四个方面的功能：

- 1、启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。
- 2、可让被调试的程序在你所指定的配置的断点处停住。(断点可以是条件表达式)
- 3、当程序被停住时，可以检查此时你的程序中所发生的事。
- 4、动态的改变你程序的执行环境。

从上面看来，GDB 和一般的调试工具没有什么两样，基本上也是完成这些功能，不过在细节上，你会发现 GDB 这个调试工具的强大，大家可能比较习惯了图形化的调试工具，但有时候，命令行的调试工具却有着图形化工具所不能完成的功能。让我们一一来看。

2. 一个调试示例

源程序：tst.c

```
1 #include <stdio.h>
2
3 int func(int n)
4 {
5     int sum=0,i;
6     for(i=0; i<n; i++)
7     {
8         sum+=i;
9     }
10    return sum;
11 }
12
13
14 main()
15 {
16     int i;
17     long result = 0;
18     for(i=1; i<=100; i++)
19     {
20         result += i;
21     }
22
23     printf("result[1-100] = %d \n", result );
24     printf("result[1-250] = %d \n", func(250) );
25 }
```

编译生成执行文件 (Linux 下):

```
hchen/test> cc -g tst.c -o tst
```

使用 GDB 调试：

```
hchen/test> gdb tst <----- 启动 GDB
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-suse-linux"...
(gdb) l <----- l 命令相当于 list, 从第一行开始例出原码。
1     #include <stdio.h>
2
3     int func(int n)
4     {
5         int sum=0,i;
6         for(i=0; i<n; i++)
7         {
8             sum+=i;
9         }
10        return sum;
(gdb) <----- 直接回车表示, 重复上一次命令
11    }
12
13
14    main()
15    {
16        int i;
17        long result = 0;
18        for(i=1; i<=100; i++)
19        {
20            result += i;
(gdb) break 16 <----- 设置断点, 在源程序第 16 行处。
Breakpoint 1 at 0x8048496: file tst.c, line 16.
(gdb) break func <----- 设置断点, 在函数 func() 入口处。
Breakpoint 2 at 0x8048456: file tst.c, line 5.
(gdb) info break <----- 查看断点信息。
Num Type      Disp Enb Address      What
1  breakpoint  keep y  0x08048496 in main at tst.c:16
2  breakpoint  keep y  0x08048456 in func at tst.c:5
(gdb) r <----- 运行程序, run 命令简写
Starting program: /home/hchen/test/tst

Breakpoint 1, main () at tst.c:17 <----- 在断点处停住。
17          long result = 0;
(gdb) n <----- 单条语句执行, next 命令简写。
18          for(i=1; i<=100; i++)
(gdb) n
20          result += i;
(gdb) n
18          for(i=1; i<=100; i++)
(gdb) n
20          result += i;
(gdb) c <----- 继续运行程序, continue 命令简写。
Continuing.
result[1-100] = 5050 <----- 程序输出。

Breakpoint 2, func (n=250) at tst.c:5
5          int sum=0,i;
(gdb) n
6          for(i=1; i<=n; i++)
(gdb) p i <----- 打印变量 i 的值, print 命令简写。
```

```
$1 = 134513808
(gdb) n
8                      sum+=i;
(gdb) n
6          for(i=1; i<=n; i++)
(gdb) p sum
$2 = 1
(gdb) n
8                      sum+=i;
(gdb) p i
$3 = 2
(gdb) n
6          for(i=1; i<=n; i++)
(gdb) p sum
$4 = 3
(gdb) bt      <----- 查看函数堆栈。
#0  func (n=250) at tst.c:5
#1  0x080484e4 in main () at tst.c:24
#2  0x400409ed in libc start main () from /lib/libc.so.6
(gdb) finish    <----- 退出函数。
Run till exit from #0  func (n=250) at tst.c:5
0x080484e4 in main () at tst.c:24
24      printf("result[1-250] = %d \n", func(250) );
Value returned is $6 = 31375
(gdb) c      <----- 继续运行。
Continuing.
result[1-250] = 31375   <-----程序输出。

Program exited with code 027. <-----程序退出，调试结束。
(gdb) q      <----- 退出 gdb。
hchen/test>
```

好了，有了以上的感性认识，还是让我们来系统地认识一下 gdb 吧。

3. 使用 GDB

一般来说 GDB 主要调试的是 C/C++的程序。要调试 C/C++的程序，首先在编译时，我们必须要把调试信息加到可执行文件中。使用编译器 (cc/gcc/g++) 的 -g 参数可以做到这一点。如：

```
> cc -g hello.c -o hello
> g++ -g hello.cpp -o hello
```

如果没有-g，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。当你用-g 把调试信息加入之后，并成功编译目标代码以后，让我们来看看如何用 gdb 来调试他。

启动 GDB 的方法有以下几种：

1、**gdb <program>**

program 也就是你的执行文件，一般在当前目录下。

2、**gdb <program> core**

用 gdb 同时调试一个运行程序和 core 文件，core 是程序非法执行后 core dump 后产生的文件。

3、**gdb <program> <PID>**

如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程 ID。gdb 会自动 attach 上去，并调试他。program 应该在 PATH 环境变量中搜索得到。

GDB 启动时，可以加上一些 GDB 的启动开关，详细的开关可以用 gdb -help 查看。我在下面只例举一些比较常用的参数：

-symbols <file>	从指定文件中读取符号表。
-----------------	--------------

-s <file>	
-se file	从指定文件中读取符号表信息，并把他用在可执行文件中。
-core <file>	调试时 core dump 的 core 文件。
-c <file>	
-directory <directory> -d <directory>	加入一个源文件的搜索路径。默认搜索路径是环境变量中 PATH 所定义的路径。

4. GDB 的命令概貌

启动 gdb 后，就你被带入 gdb 的调试环境中，就可以使用 gdb 的命令开始调试程序了，gdb 的命令可以使用 help 命令来查看，如下所示：

```
/home/hchen> gdb
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-suse-linux".
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

gdb 的命令很多，gdb 把之分成许多个种类。help 命令只是例出 gdb 的命令种类，如果要看种类中的命令，可以使用 help <class> 命令，如：help breakpoints，查看设置断点的所有命令。也可以直接 help <command> 来查看命令的帮助。

gdb 中，输入命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然，命令的前几个字符应该要标志着一个唯一的命令，在 Linux 下，你可以敲击两次 TAB 键来补齐命令的全称，如果有重复的，那么 gdb 会把其列出来。

示例一：在进入函数 func 时，设置一个断点。可以敲入 break func，或是直接就是 b func

```
(gdb) b func
Breakpoint 1 at 0x8048458: file hello.c, line 10.
```

示例二：敲入 b 按两次 TAB 键，你会看到所有 b 打头的命令：

```
(gdb) b
```

```
backtrace break bt  
(gdb)
```

用 GDB 调试程序：只记得函数的前缀，可以这样：

```
(gdb) b make <按 TAB 键>  
(再按下一次 TAB 键，你会看到)  
make a section from file      make environ  
make_abs_section               make_function_type  
make_blockvector                make_pointer_type  
make_cleanup                   make_reference_type  
make_command                   make_symbol_completion_list  
(gdb) b make_
```

GDB 把所有 make 开头的函数全部例出来给你查看。

示例四：调试 C++ 的程序时，有可以函数名一样。如：

```
(gdb) b 'bubble( M-?  
bubble(double,double)    bubble(int,int)  
(gdb) b 'bubble(
```

你可以查看到 C++ 中的所有的重载函数及参数。（注：M-? 和 “按两次 TAB 键” 是一个意思）

要退出 gdb 时，只用发 quit 或命令简称 q 就行了。

5. GDB 中运行 UNIX 的 shell 程序

在 gdb 环境中，你可以执行 UNIX 的 shell 的命令，使用 gdb 的 shell 命令来完成：

```
shell <command string>
```

调用 UNIX 的 shell 来执行<command string>，环境变量 SHELL 中定义的 UNIX 的 shell 将会被用来执行<command string>，如果 SHELL 没有定义，那就使用 UNIX 的标准 shell：/bin/sh。（在 Windows 中使用 Command.com 或 cmd.exe）

还有一个 gdb 命令是 make：

```
make <make-args>
```

可以在 gdb 中执行 make 命令来重新 build 自己的程序。这个命令等价于 “shell make <make-args>”。

6. 在 GDB 中运行程序

当以 gdb <program> 方式启动 gdb 后，gdb 会在 PATH 路径和当前目录中搜索<program>的源文件。如要确认 gdb 是否读到源文件，可使用 l 或 list 命令，看看 gdb 是否能列出源代码。

在 gdb 中，运行程序使用 r 或是 run 命令。程序的运行，你有可能需要设置下面四方面的事。

1、程序运行参数。

set args 可指定运行时参数。（如：set args 10 20 30 40 50）

show args 命令可以查看设置好的运行参数。

2、运行环境。

path <dir> 可设定程序的运行路径。

show paths 查看程序的运行路径。

set environment varname [=value] 设置环境变量。如：set env USER=hchen

show environment [varname] 查看环境变量。

3、工作目录。

cd <dir> 相当于 shell 的 cd 命令。

pwd 显示当前的所在目录。

4、程序的输入输出。

info terminal 显示你程序用到的终端的模式。

使用重定向控制程序输出。如：run > outfile

tty 命令可以指写输入输出的终端设备。如：tty /dev/ttys

7. 调试已运行的程序

两种方法：

1、在 UNIX 下用 ps 查看正在运行的程序的 PID (进程 ID)，然后用 gdb <program> PID 格式挂接正在运行的程序。

2、先用 gdb <program> 关联上源代码，并进行 gdb，在 gdb 中用 attach 命令来挂接进程的 PID。并用 detach 来取消挂接的进程。

8. 暂停 / 恢复程序运行

调试程序中，暂停程序运行是必须的，GDB 可以方便地暂停程序的运行。你可以设置程序的在哪行停住，在什么条件下停住，在收到什么信号时停住等等。以便于你查看运行时的变量，以及运行时的流程。

当进程被 gdb 停住时，你可以使用 info program 来查看程序的是否在运行，进程号，被暂停的原因。

在 gdb 中，我们可以有以下几种暂停方式：断点 (BreakPoint)、观察点 (WatchPoint)、捕捉点 (CatchPoint)、信号 (Signals)、线程停止 (Thread Stops)。如果要恢复程序运行，可以使用 c 或是 continue 命令。

一、设置断点 (BreakPoint)

我们用 break 命令来设置断点。正面有几点设置断点的方法：

```
break <function>
```

在进入指定函数时停住。C++中可以使用 class::function 或 function(type,type)格式来指定函数名。

```
break <linenum>
```

在指定行号停住。

```
break +offset
```

```
break -offset
```

在当前行号的前面或后面的 offset 行停住。offset 为自然数。

```
break filename:linenum
```

在源文件 filename 的 linenum 行处停住。

```
break filename:function
```

在源文件 filename 的 function 函数的入口处停住。

```
break *address
```

在程序运行的内存地址处停住。

```
break
```

break 命令没有参数时，表示在下一条指令处停住。

```
break ... if <condition>
```

...可以是上述的参数，condition 表示条件，在条件成立时停住。比如在循环体中，可以设置 break if i=100，表示当 i 为 100 时停住程序。

查看断点时，可使用 info 命令，如下所示：(注：n 表示断点号)

```
info breakpoints [n]
info break [n]
```

二、设置观察点 (WatchPoint)

观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停住程序。我们有下面的几种方法来设置观察点：

```
watch <expr>
```

为表达式（变量）expr 设置一个观察点。一量表达式值有变化时，马上停住程序。

```
rwatch <expr>
```

当表达式（变量）expr 被读时，停住程序。

```
awatch <expr>
```

当表达式（变量）的值被读或被写时，停住程序。

```
info watchpoints
```

列出当前所设置了的所有观察点。

三、设置捕捉点 (CatchPoint)

你可设置捕捉点来捕捉程序运行时的一些事件。如：载入共享库（动态链接库）或是 C++ 的异常。设置捕捉点的格式为：

```
catch <event>
```

当 event 发生时，停住程序。event 可以是下面的内容：

1、throw 一个 C++ 抛出的异常。（throw 为关键字）

2、catch 一个 C++ 捕捉到的异常。（catch 为关键字）

3、exec 调用系统调用 exec 时。（exec 为关键字，目前此功能只在 HP-UX 下有用）

4、fork 调用系统调用 fork 时。（fork 为关键字，目前此功能只在 HP-UX 下有用）

5、vfork 调用系统调用 vfork 时。（vfork 为关键字，目前此功能只在 HP-UX 下有用）

6、load 或 load <libname> 载入共享库（动态链接库）时。（load 为关键字，目前此功能只在 HP-UX 下有用）

7、unload 或 unload <libname> 卸载共享库（动态链接库）时。（unload 为关键字，目前此功能只在 HP-UX 下有用）

```
tcatch <event>
```

只设置一次捕捉点，当程序停住以后，应点被自动删除。

四、维护停止点

上面说了如何设置程序的停止点，GDB 中的停止点也就是上述的三类。在 GDB 中，如果你觉得已定义好的停止点没有用了，你可以使用 delete、clear、disable、enable 这几个命令来进行维护。

```
clear
```

清除所有的已定义的停止点。

```
clear <function>
```

```
clear <filename:function>
```

清除所有设置在函数上的停止点。

```
clear <linenum>
```

```
clear <filename:linenum>
```

清除所有设置在指定行上的停止点。

```
delete [breakpoints] [range...]
```

删除指定的断点，breakpoints 为断点号。如果不指定断点号，则表示删除所有的断点。range 表示断点号的范围（如：3-7）。其简写命令为 d。

比删除更好的一种方法是 disable 停止点，disable 了的停止点，GDB 不会删除，当你还需要时，enable 即可，就好像回收站一样。

```
disable [breakpoints] [range...]
```

disable 所指定的停止点, breakpoints 为停止点号。如果什么都不指定, 表示 disable 所有的停止点。
简写命令是 dis.

```
enable [breakpoints] [range...]
```

enable 所指定的停止点, breakpoints 为停止点号。

```
enable [breakpoints] once range...
```

enable 所指定的停止点一次, 当程序停止后, 该停止点马上被 GDB 自动 disable。

```
enable [breakpoints] delete range...
```

enable 所指定的停止点一次, 当程序停止后, 该停止点马上被 GDB 自动删除。

五、停止条件维护

前面在说到设置断点时, 我们提到过可以设置一个条件, 当条件成立时, 程序自动停止, 这是一个非常强大的功能, 这里, 我想专门说说这个条件的相关维护命令。一般来说, 为断点设置一个条件, 我们使用 if 关键词, 后面跟其断点条件。并且, 条件设置好后, 我们可以用 condition 命令来修改断点的条件。(只有 break 和 watch 命令支持 if, catch 目前暂不支持 if)

```
condition <bnum> <expression>
```

修改断点号为 bnum 的停止条件为 expression。

```
condition <bnum>
```

清除断点号为 bnum 的停止条件。

还有一个比较特殊的维护命令 ignore, 你可以指定程序运行时, 忽略停止条件几次。

```
ignore <bnum> <count>
```

表示忽略断点号为 bnum 的停止条件 count 次。

六、为停止点设定运行命令

我们可以使用 GDB 提供的 command 命令来设置停止点的运行命令。也就是说, 当运行的程序在被停止住时, 我们可以让其自动运行一些别的命令, 这很有利行自动化调试。对基于 GDB 的自动化调试是一个强大的支持。

```
commands [bnum]
...
end
```

为断点号 bnum 指写一个命令列表。当程序被该断点停住时, gdb 会依次运行命令列表中的命令。

例如:

```
break foo if x>0
commands
printf "x is %d\n", x
continue
end
```

断点设置在函数 foo 中, 断点条件是 x>0, 如果程序被断住后, 也就是, 一旦 x 的值在 foo 函数中大于 0, GDB 会自动打印出 x 的值, 并继续运行程序。

如果你要清除断点上的命令序列, 那么只要简单的执行一下 commands 命令, 并直接在打个 end 就行了。

七、断点菜单

在 C++ 中, 可能会重复出现同一个名字的函数若干次 (函数重载), 在这种情况下, break <function> 不能告诉 GDB 要停在哪个函数的入口。当然, 你可以使用 break <function(type)> 也就是把函数的参数类型告诉 GDB, 以指定一个函数。否则的话, GDB 会给你列出一个断点菜单供你选择你所需要的断点。你只要输入你菜单列表中的编号就可以了。如:

```
(gdb) b String::after
[0] cancel
```

```
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
breakpoints.
(gdb)
```

可见，GDB 列出了所有 after 的重载函数，你可以选一下列表编号就行了。0 表示放弃设置断点，1 表示所有函数都设置断点。

八、恢复程序运行和单步调试

当程序被停住了，你可以用 continue 命令恢复程序的运行直到程序结束，或下一个断点到来。也可以使用 step 或 next 命令单步跟踪程序。

```
continue [ignore-count]
c [ignore-count]
fg [ignore-count]
```

恢复程序运行，直到程序结束，或是下一个断点到来。ignore-count 表示忽略其后的断点次数。continue, c, fg 三个命令都是一样的意思。

```
step <count>
```

单步跟踪，如果有函数调用，他会进入该函数。进入函数的前提是，此函数被编译有 debug 信息。很像 VC 等工具中的 step in。后面可以加 count 也可以不加，不加表示一条条地执行，加表示执行后面的 count 条指令，然后再停住。

```
next <count>
```

同样单步跟踪，如果有函数调用，他不会进入该函数。很像 VC 等工具中的 step over。后面可以加 count 也可以不加，不加表示一条条地执行，加表示执行后面的 count 条指令，然后再停住。

```
set step-mode
set step-mode on
```

打开 step-mode 模式，于是，在进行单步跟踪时，程序不会因为没有 debug 信息而不停住。这个参数有利于查看机器码。

```
set step-mod off
```

关闭 step-mode 模式。

```
finish
```

运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息。

```
until 或 u
```

当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。

```
stepi 或 si
nexti 或 ni
```

单步跟踪一条机器指令！一条程序代码有可能由数条机器指令完成，stepi 和 nexti 可以单步执行机器指令。与之同样有相同功能的命令是“display/i \$pc”，当运行完这个命令后，单步跟踪会在打出程序代码的同时打出机器指令（也就是汇编代码）

九、信号 (Signals)

信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号。尤其是 UNIX，比较重要应用程序一般都会处理信号。UNIX 定义了许多信号，比如 SIGINT 表示中断字符信号，也就是 Ctrl+C

的信号, SIGBUS 表示硬件故障的信号; SIGCHLD 表示子进程状态改变信号; SIGKILL 表示终止程序运行的信号, 等等。信号量编程是 UNIX 下非常重要的一种技术。

GDB 有能力在你调试程序的时候处理任何一种信号, 你可以告诉 GDB 需要处理哪一种信号。你可以要求 GDB 收到你所指定的信号时, 马上停住正在运行的程序, 以供你进行调试。你可以用 GDB 的 handle 命令来完成这一功能。

```
handle <signal> <keywords...>
```

在 GDB 中定义一个信号处理。信号<signal>可以以 SIG 开头或不以 SIG 开头, 可以用定义一个要处理信号的范围(如: SIGIO-SIGKILL, 表示处理从 SIGIO 信号到 SIGKILL 的信号, 其中包括 SIGIO, SIGIOT, SIGKILL 三个信号), 也可以使用关键字 all 来标明要处理所有的信号。一旦被调试的程序接收到信号, 运行程序马上会被 GDB 停住, 以供调试。其<keywords>可以是以下几种关键字的一个或多个。

```
nostop
```

当被调试的程序收到信号时, GDB 不会停住程序的运行, 但会打出消息告诉你收到这种信号。

```
stop
```

当被调试的程序收到信号时, GDB 会停住你的程序。

```
print
```

当被调试的程序收到信号时, GDB 会显示出一条信息。

```
noprint
```

当被调试的程序收到信号时, GDB 不会告诉你收到信号的信息。

```
pass
```

```
noignore
```

当被调试的程序收到信号时, GDB 不处理信号。这表示, GDB 会把这个信号交给被调试程序会处理。

```
nopass
```

```
ignore
```

当被调试的程序收到信号时, GDB 不会让被调试程序来处理这个信号。

```
info signals
```

```
info handle
```

查看有哪些信号在被 GDB 检测中。

十、线程 (Thread Stops)

如果你程序是多线程的话, 你可以定义你的断点是否在所有的线程上, 或是在某个特定的线程。GDB 很容易帮你完成这一工作。

```
break <linespec> thread <threadno>
break <linespec> thread <threadno> if ...
```

linespec 指定了断点设置在的源程序的行号。threadno 指定了线程的 ID, 注意, 这个 ID 是 GDB 分配的, 你可以通过 “info threads” 命令来查看正在运行程序中的线程信息。如果你不指定 thread <threadno> 则表示你的断点设在所有线程上面。你还可以为某线程指定断点条件。如:

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

当你的程序被 GDB 停住时, 所有的运行线程都会被停住。这方便你查看运行程序的总体情况。而在你恢复程序运行时, 所有的线程也会被恢复运行。那怕是主进程在被单步调试时。

9. 查看栈信息

当程序被停住了, 你需要做的第一件事就是查看程序是在哪里停住的。当你的程序调用了一个函数, 函数的地址, 函数参数, 函数内的局部变量都会被压入“栈”(Stack)中。你可以用 GDB 命令来查看当前的栈中的信息。

下面是一些查看函数调用栈信息的 GDB 命令:

```
backtrace
```

<http://emag.csdn.net>

<http://purec.binghua.com>

bt

打印当前的函数调用栈的所有信息。如：

```
(gdb) bt
#0  func (n=250) at tst.c:6
#1  0x08048524 in main (argc=1, argv=0xbfffff674) at tst.c:30
#2  0x400409ed in __libc_start_main () from /lib/libc.so.6
```

从上可以看出函数的调用栈信息：__libc_start_main --> main() --> func()

```
backtrace <n>
bt <n>
```

n 是一个正整数，表示只打印栈顶上 n 层的栈信息。

```
backtrace <-n>
bt <-n>
```

-n 表一个负整数，表示只打印栈底下 n 层的栈信息。

如果你要查看某一层的信息，你需要在切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前栈，如果你要查看栈下面层的详细信息，首先要做的是切换当前栈。

```
frame <n>
f <n>
```

n 是一个从 0 开始的整数，是栈中的层编号。比如：frame 0，表示栈顶，frame 1，表示栈的第二层。

```
up <n>
```

表示向栈的上面移动 n 层，可以不打 n，表示向上移动一层。

```
down <n>
```

表示向栈的下面移动 n 层，可以不打 n，表示向下移动一层。

上面的命令，都会打印出移动到的栈层的信息。如果你不想让其打出信息。你可以使用这三个命令：

select-frame <n> 对应于 frame 命令。

up-silently <n> 对应于 up 命令。

down-silently <n> 对应于 down 命令。

查看当前栈层的信息，你可以用以下 GDB 命令：

```
frame 或 f
```

会打印出这些信息：栈的层编号，当前的函数名，函数参数值，函数所在文件及行号，函数执行到的语句。

```
info frame
info f
```

这个命令会打印出更为详细的当前栈层的信息，只不过，大多数都是运行时的内内地址。比如：函数地址，调用函数的地址，被调用函数的地址，目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等。如：

```
(gdb) info f
Stack level 0, frame at 0xbfffff5d4:
eip = 0x804845d in func (tst.c:6); saved eip 0x8048524
called by frame at 0xbfffff60c
source language c.
Arglist at 0xbfffff5d4, args: n=250
Locals at 0xbfffff5d4, Previous frame's sp is 0x0
Saved registers:
    ebp at 0xbfffff5d4, eip at 0xbfffff5d8
```

```
info args
```

打印出当前函数的参数名及其值。

```
info locals
```

打印出当前函数中所有局部变量及其值。

```
info catch
```

打印出当前的函数中的异常处理信息。

10. 查看源程序

一、显示源代码

GDB 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g 的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。当程序停下来以后，GDB 会报告程序停在了那个文件的第几行上。你可以用 list 命令来打印程序的源代码。还是来看一看查看源代码的 GDB 命令吧。

```
list <linenum>
```

显示程序第 linenum 行的周围的源程序。

```
list <function>
```

显示函数名为 function 的函数的源程序。

```
list
```

显示当前行后面的源程序。

```
list -
```

显示当前行前面的源程序。

一般是打印当前行的上 5 行和下 5 行，如果显示函数是上 2 行下 8 行，默认是 10 行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。

```
set listsize <count>
```

设置一次显示源代码的行数。

```
show listsize
```

查看当前 listsize 的设置。

list 命令还有下面的用法：

```
list <first>, <last>
```

显示从 first 行到 last 行之间的源代码。

```
list , <last>
```

显示从当前行到 last 行之间的源代码。

```
list +
```

往后显示源代码。

一般来说在 list 后面可以跟以下这些的参数：

<linenum> 行号。

<+offset> 当前行号的正偏移量。

<-offset> 当前行号的负偏移量。

<filename:linenum> 哪个文件的哪一行。

<function> 函数名。

<filename:function> 哪个文件中的哪个函数。

<*address> 程序运行时的语句在内存中的地址。

二、搜索源代码

不仅如此，GDB 还提供了源代码搜索的命令：

```
forward-search <regexp>
```

```
search <regexp>
```

向前面搜索。

```
reverse-search <regexp>
```

全部搜索。

其中，<regexp>就是正则表达式，也主一个字符串的匹配模式，关于正则表达式，我就不在这里讲了，还请各位查看相关资料。

三、指定源文件的路径

某些时候，用-g 编译过后的执行程序中只是包括了源文件的名字，没有路径名。GDB 提供了可以让你指定源文件的路径的命令，以便 GDB 进行搜索。

```
directory <dirname ... >
dir <dirname ... >
```

加一个源文件路径到当前路径的前面。如果你要指定多个路径，UNIX 下你可以使用 “:”，Windows 下你可以使用 “;”。

```
directory
```

清除所有的自定义的源文件搜索路径信息。

```
show directories
```

显示定义了的源文件搜索路径。

四、源代码的内存

你可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟“行号”，“函数名”，“文件名:行号”，“文件名:函数名”，这个命令会打印出所指定的源码在运行时的内存地址，如：

```
(gdb) info line tst.c:func
Line 5 of "tst.c" starts at address 0x8048456 <func+6> and ends at 0x804845d <func+13>.
```

还有一个命令 (disassemble) 你可以查看源程序的当前执行时的机器码，这个命令会把目前内存中的指令 dump 出来。如下面的示例表示查看函数 func 的汇编代码。

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 <func>:    push   %ebp
0x8048451 <func+1>:   mov    %esp,%ebp
0x8048453 <func+3>:   sub    $0x18,%esp
0x8048456 <func+6>:   movl   $0x0,0xffffffffc(%ebp)
0x804845d <func+13>:  movl   $0x1,0xffffffff8(%ebp)
0x8048464 <func+20>:  mov    0xffffffff8(%ebp),%eax
0x8048467 <func+23>:  cmp    0x8(%ebp),%eax
0x804846a <func+26>:  jle    0x8048470 <func+32>
0x804846c <func+28>:  jmp    0x8048480 <func+48>
0x804846e <func+30>:  mov    %esi,%esi
0x8048470 <func+32>:  mov    0xffffffff8(%ebp),%eax
0x8048473 <func+35>:  add    %eax,0xffffffffc(%ebp)
0x8048476 <func+38>:  incl   0xffffffff8(%ebp)
0x8048479 <func+41>:  jmp    0x8048464 <func+20>
0x804847b <func+43>:  nop
0x804847c <func+44>:  lea    0x0(%esi,1),%esi
0x8048480 <func+48>:  mov    0xfffffffffc(%ebp),%edx
0x8048483 <func+51>:  mov    %edx,%eax
0x8048485 <func+53>:  jmp    0x8048487 <func+55>
0x8048487 <func+55>:  mov    %ebp,%esp
0x8048489 <func+57>:  pop    %ebp
0x804848a <func+58>:  ret
End of assembler dump.
```

11. 查看运行时数据

在你调试程序时，当程序被停住时，你可以使用 print 命令（简写命令为 p），或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式是：

```
print <expr>
print /<f> <expr>
```

<expr>是表达式，是你所调试的程序的语言的表达式（GDB 可以调试多种编程语言），<f>是输出的格式，比如，如果要把表达式按 16 进制的格式输出，那么就是/x。

一、表达式

print 和许多 GDB 的命令一样，可以接受一个表达式，GDB 会根据当前的程序运行的数据来计算这个表达式，既然是表达式，那么就可以是当前程序运行中的 const 常量、变量、函数等内容。可惜的是 GDB 不能使用你在程序中所定义的宏。

表达式的语法应该是当前所调试的语言的语法，由于 C/C++ 是一种大众型的语言，所以，本文中的例子都是关于 C/C++ 的。（而关于用 GDB 调试其它语言的章节，我将在后面介绍）

在表达式中，有几种 GDB 所支持的操作符，它们可以用在任何一种语言中。

@

是一个和数组有关的操作符，在后面会有更详细的说明。

::

指定一个在文件或是一个函数中的变量。

{<type>} <addr>

表示一个指向内存地址<addr>的类型为 type 的一个对象。

二、程序变量

在 GDB 中，你可以随时查看以下三种变量的值：

- 1、全局变量（所有文件可见的）
- 2、静态全局变量（当前文件可见的）
- 3、局部变量（当前 Scope 可见的）

如果你的局部变量和全局变量发生冲突（也就是重名），一般情况下是局部变量会隐藏全局变量，也就是说，如果一个全局变量和一个函数中的局部变量同名时，如果当前停止点在函数中，用 print 显示出的变量的值会是函数中的局部变量的值。如果此时你想查看全局变量的值时，你可以使用 “::” 操作符：

```
file::variable  
function::variable
```

可以通过这种形式指定你所想查看的变量，是哪个文件中的或是哪个函数中的。例如，查看文件 f2.c 中的全局变量 x 的值：

```
gdb) p 'f2.c'::x
```

当然，“::”操作符会和 C++ 中的发生冲突，GDB 能自动识别 “::” 是否 C++ 的操作符，所以你不必担心在调试 C++ 程序时会出现异常。

另外，需要注意的是，如果你的程序编译时开启了优化选项，那么在用 GDB 调试被优化过的程序时，可能会发生某些变量不能访问，或是取值错误码的情况。这个是很正常的，因为优化程序会删改你的程序，整理你程序的语句顺序，剔除一些无意义的变量等，所以在 GDB 调试这种程序时，运行时的指令和你所编写指令就有不一样，也就会出现你所想象不到的结果。对付这种情况时，需要在编译程序时关闭编译优化。一般来说，几乎所有的编译器都支持编译优化的开关，例如，GNU 的 C/C++ 编译器 GCC，你可以使用 “-gstabs” 选项来解决这个问题。关于编译器的参数，还请查看编译器的使用说明文档。

三、数组

有时候，你需要查看一段连续的内存空间的值。比如数组的一段，或是动态分配的数据的大小。你可以使用 GDB 的 “@” 操作符，“@” 的左边是第一个内存的地址的值，“@” 的右边则为你想查看内存的长度。例如，你的程序中有这样的语句：

```
int *array = (int *) malloc (len * sizeof (int));
```

于是，在 GDB 调试过程中，你可以以如下命令显示出这个动态数组的取值：

```
p *array@len
```

@的左边是数组的首地址的值，也就是变量 array 所指向的内容，右边则是数据的长度，其保存在变量 len 中，其输出结果，大约是下面这个样子的：

```
(gdb) p *array@len
$1 = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40}
```

如果是静态数组的话，可以直接用 print 数组名，就可以显示数组中所有数据的内容了。

四、输出格式

一般来说，GDB 会根据变量的类型输出变量的值。但你也可以自定义 GDB 的输出的格式。例如，你想输出一个整数的十六进制，或是二进制来查看这个整型变量的中的位的情况。要做到这样，你可以使用 GDB 的数据显示格式：

- x 按十六进制格式显示变量。
- d 按十进制格式显示变量。
- u 按十六进制格式显示无符号整型。
- o 按八进制格式显示变量。
- t 按二进制格式显示变量。
- a 按十六进制格式显示变量。
- c 按字符格式显示变量。
- f 按浮点数格式显示变量。

```
(gdb) p i
$21 = 101

(gdb) p/a i
$22 = 0x65

(gdb) p/c i
$23 = 101 'e'

(gdb) p/f i
$24 = 1.41531145e-43

(gdb) p/x i
$25 = 0x65

(gdb) p/t i
$26 = 1100101
```

五、查看内存

你可以使用 examine 命令（简写是 x）来查看内存地址中的值。x 命令的语法如下所示：

```
x/<n/f/u> <addr>
```

n、f、u 是可选的参数。

n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。

f 表示显示的格式，参见上面。如果地址所指的是字符串，那么格式可以是 s，如果地址是指令地址，那么格式可以是 i。

u 表示从当前地址往后请求的字节数，如果不指定的话，GDB 默认是 4 个 bytes。u 参数可以用下面的字符来代替，b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。当我们指定了字节长度后，GDB 会从指内存定的内存地址开始，读写指定字节，并把其当作一个值取出来。

<addr>表示一个内存地址。

n/f/u 三个参数可以一起使用。例如：

命令：x/3uh 0x54320 表示，从内存地址 0x54320 读取内容，h 表示以双字节为一个单位，3 表示三个单位，u 表示按十六进制显示。

六、自动显示

你可以设置一些自动显示的变量，当程序停住时，或是在你单步跟踪时，这些变量会自动显示。相关的GDB命令是 display。

```
display <expr>
display/<fmt> <expr>
display/<fmt> <addr>
```

expr 是一个表达式，fmt 表示显示的格式，addr 表示内存地址，当你用 display 设定好了一个或多个表达式后，只要你的程序被停下来，GDB 会自动显示你所设置的这些表达式的值。

格式 i 和 s 同样被 display 支持，一个非常有用命令是：

```
display/i $pc
```

\$pc 是 GDB 的环境变量，表示着指令的地址，/i 则表示输出格式为机器指令码，也就是汇编。于是当程序停下后，就会出现源代码和机器指令码相对应的情形，这是一个很有意思的功能。

下面是一些和 display 相关的 GDB 命令：

```
undisplay <dnums...>
delete display <dnums...>
```

删除自动显示，dnums 意为所设置好了的自动显式的编号。如果要同时删除几个，编号可以用空格分隔，如果要删除一个范围内的编号，可以用减号表示（如：2-5）

```
disable display <dnums...>
enable display <dnums...>
```

disable 和 enable 不删除自动显示的设置，而只是让其失效和恢复。

```
info display
```

查看 display 设置的自动显示的信息。GDB 会打出一张表格，向你报告当然调试中设置了多少个自动显示设置，其中包括，设置的编号，表达式，是否 enable。

七、设置显示选项

GDB 中关于显示的选项比较多，这里我只例举大多数常用的选项。

```
set print address
set print address on
```

打开地址输出，当程序显示函数信息时，GDB 会显出函数的参数地址。系统默认为打开的，如：

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
      at input.c:530
      530      if (lquote != def_lquote)
```

```
set print address off
```

关闭函数的参数地址显示，如：

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
      530      if (lquote != def_lquote)
```

```
show print address
```

查看当前地址显示选项是否打开。

```
set print array
set print array on
```

打开数组显示，打开后当数组显示时，每个元素占一行，如果不打开的话，每个元素则以逗号分隔。这个选项默认是关闭的。与之相关的两个命令如下，我就不再多说了。

```
set print array off
show print array
```

```
set print elements <number-of-elements>
```

这个选项主要是设置数组的，如果你的数组太大了，那么就可以指定一个<number-of-elements>来指定数据显示的最大长度，当到达这个长度时，GDB 就不再往下显示了。如果设置为 0，则表示不限制。

```
show print elements
```

查看 print elements 的选项信息。

```
set print null-stop <on/off>
```

如果打开了这个选项，那么当显示字符串时，遇到结束符则停止显示。这个选项默认为 off。

```
set print pretty on
```

如果打开 printf pretty 这个选项，那么当 GDB 显示结构体时会比较漂亮。如：

```
$1 = {  
    next = 0x0,  
    flags = {  
        sweet = 1,  
        sour = 1  
    },  
    meat = 0x54 "Pork"  
}
```

```
set print pretty off
```

关闭 printf pretty 这个选项，GDB 显示结构体时会如下显示：

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}
```

```
show print pretty
```

查看 GDB 是如何显示结构体的。

```
set print sevenbit-strings <on/off>
```

设置字符显示，是否按“\nnn”的格式显示，如果打开，则字符串或字符数据按\nnn 显示，如 “\065”。

```
show print sevenbit-strings
```

查看字符显示开关是否打开。

```
set print union <on/off>
```

设置显示结构体时，是否显式其内的联合体数据。例如有以下数据结构：

```
typedef enum {Tree, Bug} Species;  
typedef enum {Big tree, Acorn, Seedling} Tree forms;  
typedef enum {Caterpillar, Cocoon, Butterfly}  
    Bug_forms;  
  
struct thing {  
    Species it;  
    union {  
        Tree forms tree;  
        Bug_forms bug;  
    } form;  
};  
  
struct thing foo = {Tree, {Acorn}};
```

当打开这个开关时，执行 p foo 命令后，会如下显示：

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

当关闭这个开关时，执行 p foo 命令后，会如下显示：

```
$1 = {it = Tree, form = {...}}
```

```
show print union
```

查看联合体数据的显示方式

```
set print object <on/off>
```

在 C++ 中，如果一个对象指针指向其派生类，如果打开这个选项，GDB 会自动按照虚方法调用的规则显示输出，如果关闭这个选项的话，GDB 就不管虚函数表了。这个选项默认是 off。

```
show print object
```

查看对象选项的设置。

```
set print static-members <on/off>
```

这个选项表示，当显示一个 C++ 对象中的内容时，是否显示其中的静态数据成员。默认是 on。

```
show print static-members
```

查看静态数据成员选项设置。

```
set print vtbl <on/off>
```

当此选项打开时，GDB 将用比较规整的格式来显示虚函数表时。其默认是关闭的。

```
show print vtbl
```

查看虚函数显示格式的选项。

八、历史记录

当你用 GDB 的 print 查看程序运行时的数据时，你每一个 print 都会被 GDB 记录下来。GDB 会以\$1, \$2, \$3 这样的方式为你每一个 print 命令编上号。于是，你可以使用这个编号访问以前的表达式，如\$1。这个功能所带来的好处是，如果你先前输入了一个比较长的表达式，如果你还想查看这个表达式的值，你可以使用历史记录来访问，省去了重复输入。

九、GDB 环境变量

你可以在 GDB 的调试环境中定义自己的变量，用来保存一些调试程序中的运行数据。要定义一个 GDB 的变量很简单只需。使用 GDB 的 set 命令。GDB 的环境变量和 UNIX 一样，也是以\$起头。如：

```
set $foo = *object_ptr
```

使用环境变量时，GDB 会在你第一次使用时创建这个变量，而在以后的使用中，则直接对其赋值。环境变量没有类型，你可以给环境变量定义任一的类型。包括结构体和数组。

```
show convenience
```

该命令查看当前所设置的所有环境变量。

这是一个比较强大的功能，环境变量和程序变量的交互使用，将使得程序调试更为灵活便捷。例如：

```
set $i = 0
print bar[$i++]->contents
```

于是，当你就不必，print bar[0]->contents, print bar[1]->contents 地输入命令了。输入这样的命令后，只用敲回车，重复执行上一条语句，环境变量会自动累加，从而完成逐个输出的功能。

十、查看寄存器

要查看寄存器的值，很简单，可以使用如下命令：

```
info registers
```

查看寄存器的情况。(除了浮点寄存器)

```
info all-registers
```

查看所有寄存器的情况。(包括浮点寄存器)

```
info registers <regname ...>
```

查看所指定的寄存器的情况。

寄存器中放置了程序运行时的数据，比如程序当前运行的指令地址 (ip)，程序的当前堆栈地址 (sp) 等等。你同样可以使用 print 命令来访问寄存器的情况，只需要在寄存器名字前加一个\$符号就可以了。如：p \$eip。

12. 改变程序的执行

一旦使用 GDB 挂上被调试程序，当程序运行起来后，你可以根据自己的调试思路来动态地在 GDB 中更改当前被调试程序的运行线路或是其变量的值，这个强大的功能能够让你更好的调试你的程序，比如，你可以

在程序的一次运行中走遍程序的所有分支。

一、修改变量值

修改被调试程序运行时的变量值，在 GDB 中很容易实现，使用 GDB 的 print 命令即可完成。如：

```
(gdb) print x=4
```

x=4 这个表达式是 C/C++ 的语法，意为把变量 x 的值修改为 4，如果你当前调试的语言是 Pascal，那么你可以使用 Pascal 的语法：x:=4。

在某些时候，很有可能你的变量和 GDB 中的参数冲突，如：

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

因为，set width 是 GDB 的命令，所以，出现了“Invalid syntax in expression”的设置错误，此时，你可以使用 set var 命令来告诉 GDB，width 不是你 GDB 的参数，而是程序的变量名，如：

```
(gdb) set var width=47
```

另外，还可能有些情况，GDB 并不报告这种错误，所以保险起见，在你改变程序变量取值时，最好都使用 set var 格式的 GDB 命令。

二、跳转执行

一般来说，被调试程序会按照程序代码的运行顺序依次执行。GDB 提供了乱序执行的功能，也就是说，GDB 可以修改程序的执行顺序，可以让程序执行随意跳跃。这个功能可以由 GDB 的 jump 命令来完：

```
jump <linespec>
```

指定下一条语句的运行点。<linespec>可以是文件的行号，可以是 file:line 格式，可以是+num 这种偏移量格式。表示着下一条运行语句从哪里开始。

```
jump <address>
```

这里的<address>是代码行的内存地址。

注意，jump 命令不会改变当前的程序栈中的内容，所以，当你从一个函数跳到另一个函数时，当函数运行完返回时进行弹栈操作时必然会发生错误，可能结果还是非常奇怪的，甚至于产生程序 Core Dump。所以最好是同一个函数中进行跳转。

熟悉汇编的人都知道，程序运行时，有一个寄存器用于保存当前代码所在的内存地址。所以，jump 命令也就是改变了这个寄存器中的值。于是，你可以使用“set \$pc”来更改跳转执行的地址。如：

```
set $pc = 0x485
```

三、产生信号量

使用 singal 命令，可以产生一个信号量给被调试的程序。如：中断信号 Ctrl+C。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用 GDB 产生一个信号量，这种精确地在某处产生信号非常有利程序的调试。

语法是：signal <singal>，UNIX 的系统信号量通常从 1 到 15。所以<singal>取值也在这个范围。

single 命令和 shell 的 kill 命令不同，系统的 kill 命令发信号给被调试程序时，是由 GDB 截获的，而 single 命令所发出一信号则是直接发给被调试程序的。

四、强制函数返回

如果你的调试断点在某个函数中，并还有语句没有执行完。你可以使用 return 命令强制函数忽略还没有执行的语句并返回。

```
return
```

```
return <expression>
```

使用 `return` 命令取消当前函数的执行，并立即返回，如果指定了`<expression>`，那么该表达式的值会被认为是函数的返回值。

五、强制调用函数

```
call <expr>
```

表达式中可以是一函数，以此达到强制调用函数的目的。并显示函数的返回值，如果函数返回值是 `void`，那么就不显示。

另一个相似的命令也可以完成这一功能——`print`，`print` 后面可以跟表达式，所以也可以用他来调用函数，`print` 和 `call` 的不同是，如果函数返回 `void`，`call` 则不显示，`print` 则显示函数返回值，并把该值存入历史数据中。

13. 在不同语言中使用 GDB

GDB 支持下列语言：C, C++, Fortran, PASCAL, Java, Chill, assembly, 和 Modula-2。一般说来，GDB 会根据你所调试的程序来确定当然的调试语言，比如：发现文件名后缀为“.c”的，GDB 会认为是 C 程序。文件名后缀为“.C, .cc, .cp, .cpp, .cxx, .c++”的，GDB 会认为是 C++ 程序。而后缀是“.f, F”的，GDB 会认为是 Fortran 程序，还有，后缀为如果是“.s, .S”的会认为是汇编语言。

也就是说，GDB 会根据你所调试的程序的语言，来设置自己的语言环境，并让 GDB 的命令跟着语言环境的改变而改变。比如一些 GDB 命令需要用到表达式或变量时，这些表达式或变量的语法，完全是根据当前的语言环境而改变的。例如 C/C++ 中对指针的语法是`*p`，而在 Modula-2 中则是`p^`。并且，如果你当前的程序是由几种不同语言一同编译成的，那在调试过程中，GDB 也能根据不同的语言自动地切换语言环境。这种跟着语言环境而改变的功能，真是体贴开发人员的一种设计。

下面是几个相关于 GDB 语言环境的命令：

```
show language
```

查看当前的语言环境。如果 GDB 不能识别为你所调试的编程语言，那么，C 语言被认为是默认的环境。

```
info frame
```

查看当前函数的程序语言。

```
info source
```

查看当前文件的程序语言。

如果 GDB 没有检测出当前的程序语言，那么你也可以手动设置当前的程序语言。使用 `set language` 命令即可做到。

当 `set language` 命令后什么也不跟的话，你可以查看 GDB 所支持的语言种类：

```
(gdb) set language
The currently understood settings are:

local or auto    Automatic setting based on source file
c                 Use the C language
c++               Use the C++ language
asm               Use the Asm language
chill             Use the Chill language
fortran           Use the Fortran language
java              Use the Java language
modula-2          Use the Modula-2 language
pascal             Use the Pascal language
scheme             Use the Scheme language
```

于是你可以在 `set language` 后跟上被列出来的程序语言名，来设置当前的语言环境。

14. 后记

GDB 是一个强大的命令行调试工具。大家知道命令行的强大就是在于，其可以形成执行序列，形成脚本。UNIX 下的软件全是命令行的，这给程序开发提供了极大的便利，命令行软件的优势在于，它们可以非常容易的集成在一起，使用几个简单的已有工具的命令，就可以做出一个非常强大的功能。

于是 UNIX 下的软件比 Windows 下的软件更能有机地结合，各自发挥各自的长处，组合成更为强劲的功能。而 Windows 下的图形软件基本上是各自为营，互相不能调用，很不利于各种软件的相互集成。在这里并不是要和 Windows 做个什么比较，所谓“寸有所长，尺有所短”，图形化工具还是有不如命令行的地方。（看到这句话时，希望各位千万不要再认为我就是“鄙视图形界面”，和我抬杠了）

我是根据版本为 5.1.1 的 GDB 所写的这篇文章，所以可能有些功能已被修改，或是又有更为强劲的功能。而且，我写得非常仓促，写得比较简略，并且，其中我已经看到有许多错别字了（我用五笔，所以错字让你看不懂），所以，我在这里对我文中的差错表示万分的歉意。

文中所罗列的 GDB 的功能时，我只是罗列了一些常用的 GDB 的命令和使用方法，其实，我这里只讲述的功能大约只占 GDB 所有功能的 60% 吧，详细的文档，还是请查看 GDB 的帮助和使用手册吧，或许，过段时间，如果我有空，我再写一篇 GDB 的高级使用。

我个人非常喜欢 GDB 的自动调试的功能，这个功能真的很强大，试想，我在 UNIX 下写个脚本，让脚本自动编译我的程序，被自动调试，并把结果报告出来，调试成功，自动 checkin 源码库。一个命令，编译带着调试带着 checkin，多爽啊。只是 GDB 对自动化调试目前支持还不是很成熟，只能实现半自动化，真心期望着 GDB 的自动化调试功能的成熟。

如果各位对 GDB 或是别的技术问题有兴趣的话，欢迎和我讨论交流。本人目前主要在 UNIX 下做产品的开发，所以，对 UNIX 下的软件开发比较熟悉，当然，不单单是技术，对软件工程实施，软件设计，系统分析，项目管理我也略有心得。欢迎大家找我交流。

【责任编辑：sun@hitbbs】

命令行计算器的实现

——表达式分析的实现

哈尔滨工业大学计算机学院
作者: 高立琦 (lqgao@yahoo.com)

学习编译原理时, 我曾经实现过一个简易的计算器。这篇文章相当于制作过程的小结, 和诸位读者分享一下我的一点点经验或者体会。对我来说, 做这么一个命令行计算器有两点好处: 一是加深对算符优先文法的理解; 二是实现一个小工具, 方便平时简单的计算。用 Windows 自带的计算器, 如果遇到稍复杂的表达式, 就有点麻烦了; 用 MATLAB 吧, 好像又有点……自己动手写一个, 丰衣足食。这可不是从头开始造轮子哦, 因为我们的主要目的是学习实践么, 体验一下算符优先文法的神奇力量。当然了, 遇到复杂的运算, 别忘了使用 MATLAB 之类的工具。

言归正传, 本文主要介绍如何使用算符优先文法实现一个命令行计算器, 适合学习编译原理或对表达式分析感兴趣的读者。

1. 算符文法

首先, 我们得先了解一下算符优先文法——理论指导实践。限于篇幅, 我们不能详细讨论算符优先文法, 感兴趣的读者请查阅一下“编译原理”相关的书籍, 建议阅读参考文献。

算符文法的特点是不含右侧部分为空或者右侧有两个相邻的非终结符的产生式。例如:

$E \rightarrow EAE \mid (E) \mid -E \mid id$
 $A \rightarrow + \mid - \mid * \mid / \mid ^$

就不是一个算符文法, 因为 EAE 含有 2 个或者 2 个以上相邻的非终结符。

算符文法是 LR 文法的一种, 可以采用算符优先分析(Operator-Precedence Parsing)算法分析。分析时需要一个算符优先表, 即需要文法中所有符号的优先关系。一般地, 算符文法中定义三种优先级关系, $<= >$ 。

关系	含义
$a < b$	a 后于 b 规约
$a = b$	a 和 b 同等规约
$a > b$	a 先于 b 规约

注意这里提到的关系与算术中的关系 $<, =, >$ 不同, 算术中的任意 2 个实数一定满足三种关系中的一种, 而在算符文法中, 两个终结符不满足任何关系的情况却经常出现。

2. 文法设计

我们得先定义一下这个计算器的功能, 然后才能设计出文法。仔细想一下, 应该有这些功能吧:

- 支持浮点数运算;
- 支持四则运算(加减乘除)和基本逻辑运算(与或非);
- 支持括号, 改变优先级;
- 支持幂函数, 指数函数, 对数函数, 三角函数等初等函数;

- 支持变量；

先定义这些吧。功能定义好，文法也就容易了。注意是算符文法哦。

```

expression → simple expression | id assignop simple expression
expression → simple_expression relop simple_expression
simple expression → term
simple expression → simple expression addop term
term → factor
term → term mulop factor
factor → id
factor → num
factor → ( expression )
factor → not factor

```

relop 表示==,<=,<,>,>=,!=

addop 表示|,+, -

mulop 表示*,/,&&

not 表示逻辑非,!

我们希望的全部功能就集中在这个文法中了。接下来我们就要想办法实现这个文法。

3. 词法分析部分的设计与实现

词法分析(Lexical Analysing)是编译过程的第一个阶段。简单说，就是一个预处理阶段，将输入字符串转换成机器可以识别、处理的过程。在我们的任务中，我们需要一个词法分析器，将输入的字符串识别为一个个可以分析的基本元素。我们会有哪些基本元素呢？

- 数
- 标识符(Identifier, 变量名、函数名都是标识符)
- 运算符号(+,-,*/,&&,...),

想来想去，好像就这些吧。让我们一个一个来考虑：

3.1 数的识别

计算器的基本操作对象是数。我们采用实数作为处理对象，存储格式使用双精度浮点数。识别数的文法：

```

digits → [0-9] +
optional_fraction → . digits | empty
optional_exponent → ( E (+|-|empty) digits ) | empty
num → digits optional_fraction optional_exponent

```

这是一个正则文法。部分实现代码如下：

```

int Expression::Digits()
{
    char c;
    int digits = 0;
    while( iLexem != iEndOfLexem )
    {
        c = *iLexem;
        if( c>='0' && c<='9' )
        {
            ++iLexem;
            digits = (digits<<3) + (digits<<1);      // digits *= 10;
            digits += int( c - '0' );
        }
        else

```

```

    {
        break;
    }
    return digits;
}
double Expression::OptionalFraction()
{
    char c;
    double frac = 0; // the fraction
    double e = 0.1;
    if( iLexem == iEndOfLexem ) return 0;
    c = *iLexem;
    if( c != '.' ) return 0;
    ++iLexem;
    while( iLexem != iEndOfLexem )
    {
        c = *iLexem;
        if( c>='0' && c<='9' )
        {
            ++iLexem;
            frac += e * int(c-'0');
            e = e/10;
        }
        else
            break;
    }
    return frac;
}

```

看起来应该容易，没有什么难度。

3.2 标识符的处理

在这个命令行计算器中，变量和函数名都是标识符。因此我们得想办法识别标识符。识别表示符的文法如下：

```

letter → [a-zA-Z]
digit → [0-9]
id → letter ( letter | digit )*

```

对于识别出来的标识符，我们还需要保存必要的信息。考虑到这个计算器支持变量，我们得想一些办法给每个变量“分配”空间——设置一个符号表。

符号表为一线性表，记录标识符的字符串和这个标识符的值(在本文中，符号表主要记录变量的信息)。

利用文法识别出来表示符后，首先要查表，看看“在职”；如果没有，则添加新项；有则返回索引号。具体代码请看函数 Identifier()。符号表的结构如图。

ans	10.0
a	3.1415
Variable1	-100
Variable	

3.3 运算符号的识别

有些特殊的运算符号需要进行识别，比如“==”，“<=”，“&&”等等。同标识符识别类似，程序中设置一个符号表，对于“可疑”的符号进行查表对照。

4. 算符优先分析算法

下面介绍核心问题。算符优先分析算法是基于终结符的三种关系的。

输入：字符串 w 和算符优先关系表

输出：如果 w 符合文法，算法相当于构建出一个分析树；否则，给出错误提示。

算法描述：

```
01# set ip to point to the first symbol of w$  
02# repeat forever  
03#     if $ is on top of the stack and ip points to $ then  
04#         return  
05#     else begin  
06#         let a be the topmost terminal symbol on the stack  
07#         and le be the symbol pointed to by ip;  
08#         if a<b or a=b then begin  
09#             push b onto the stack;  
10#             advance ip to the next input symbol;  
11#         end;  
12#         else if a>b then  
13#             repeat  
14#                 pop the stack  
15#             until the top stack terminal is related by < to the terminal most recently popped  
16#         else error()  
17#     end
```

如果两个终结符不符合三种关系中的任何一种，说明分析出错——输入的表达式是非法的。对我们来说，检查表达式的合法性不是目的，我们的目的是遍历分析树(Parse Tree)，执行对应的操作，实现对表达式的分析。

5. 一些细节

如果你也动手做一做，应该也会遇到这几个方面的问题。让我来先抛个砖。

5.1 关系表

关系表描述终结符之间的关系。举个简单的例子：

id	+	*	\$
id	>	>	>
+	<	>	<
*	<	>	>
\$	<	<	<

比如，+<*，表示+号的优先级低于*，当规约时，*要先于+规约。*>*>表示，同一个等级的符号，按照由左到右的顺序进行规约。一次类推。根据文法可以构建出来这样一张表。在分析时，这张表就是算法进行的导

航图。填表有一些算法，感兴趣的读者请参考相关书籍。

5.2 处理一元符号

比如非运算符号(!)就是一元符号，我们可以在填充符号表的时候，让 $! > \theta$, $\theta < !$ 就可以——因为这样(!)就有最高的级别了。不过负号(-)可是一个麻烦问题，因为负号同减号(-)在字符上相同。词法分析器将他们认为成同一个符号。如果也按照(!)处理方法处理， $a+b-c$ 的结果如何呢？最好的解决办法是在词法分析阶段将这对双胞胎兄弟识别开，虽然有点勉强词法分析器了。不过我们可以使一些小技巧，比如认为如果在(-)之前出现左括号或者操作符，则认为这个符号(-)是负号；否则是减号。这样以来，在语法分析阶段，它们两个就会各司其职。

5.3 翻译表达式

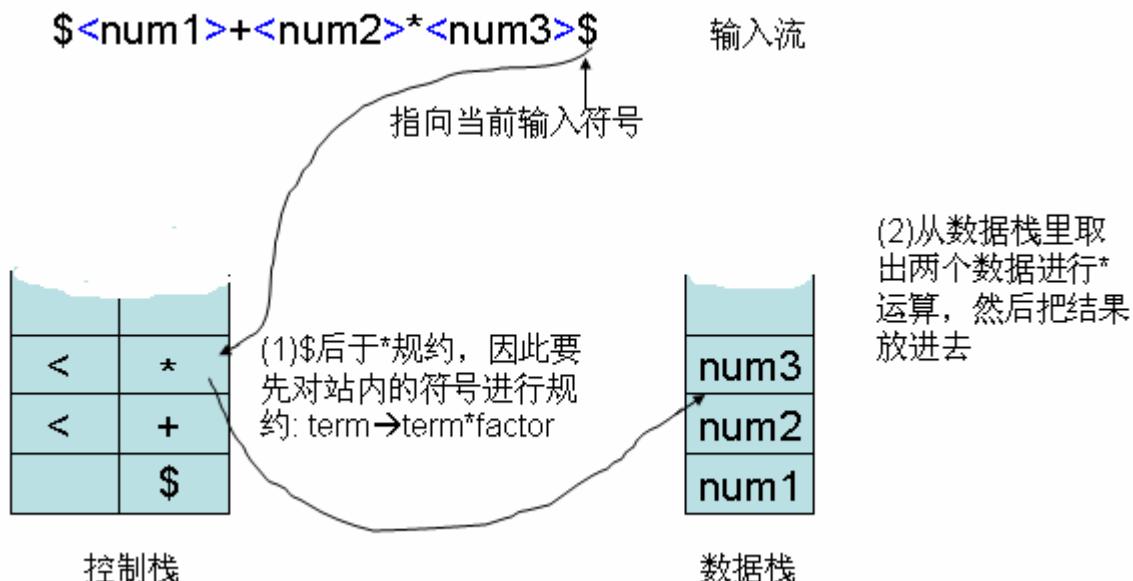


图 1 规约过程示意图

这可是激动人心的一步。因为计算机开始按照我们给出的表达式一步一步运算了，不是吗。在算法中，当遇到规约条件——当前符号后于规约栈顶符号时，算法会向栈内搜索。以加法为例，翻译过程其实就是将数据栈顶的两个数取出，做加法，然后再放回去。

部分程序的代码：

```
bool Expression::Translate(int tok, Expression::VALUE_TYPE &value)
{
    double n1, n2, r;
    string term;

    switch(tok)
    {
    case MINUS:
        n1 = -num.top(); num.pop();
        num.push(n1);
        break;
```

```
case NUM:
    num.push(Value.d);
    break;
case ID: // variables and functions
    if( !num.empty() ) n1 = num.top();
    term = TokenTable[value.i].lexem;
    if( term == "sin" ) { num.pop(); r = sin(n1); }
    else if( term == "cos" ) { num.pop(); r = cos(n1); }
    else if( term == "log" ) { num.pop(); r = log(n1); }
    else if( term == "exp" ) { num.pop(); r = exp(n1); }
    else if( term == "pi" ) { r = pi; }
    else
    {
        r = TokenTable[value.i].value.d;
    }
    num.push( r );
    break;
case ADD:
case SUB:
case MUL:
case DIV:
    n1 = num.top(); num.pop();
    n2 = num.top(); num.pop();
    switch( tok )
    {
        case ADD: r = n1 + n2;break;
        case SUB: r = n2 - n1;break;
        case MUL: r = n1 * n2;break;
        case DIV:
            if( fabs(n1) < 1e-8 ) return false; // divided by zero
            r = n2 / n1;break;
    }
    num.push(r);
    break;
.....
}
return true;
}
```

随着分析的一步步进行，表达式的值也逐渐的浮出水面，确切地说，是沉入栈底。当分析结束时，如果表达式正确的话，数据栈里面应该只有一个元素——我们想要的值。

6. 总结

本文主要介绍如何分析表达式，进而构造一个命令行计算器。分析表达式有多种方法，本文介绍的是采用算符优先分析算法。这个算法使用算符文法，是 LR 文法的一种。算符优先算法的特点是比较简单（相对 LR 分析算法来说），要求不是很严格，不能保证接收的一定是文法定义的语言。正是它的这些特点使它非常适合做表达式分析，容易实现，而且对表达式要求不太严格。不信您可以试试，看看输入表达式时少个右括号会怎么样？本文附有源代码，您可以自由使用。

运行程序后，您可以输入表达式或者赋值语句。例如

```
Command Line Calculator V0.1 By Liqi Gao.
Type ? to get help.
>a=100
ans = 100.00000
>b=20
ans = 20.00000
```

```
>a/b+10*sin(pi/2)
ans = 15.00000
>
```

你如果感兴趣，可以在这个程序的基础上增加新的功能。比如进制转换，位运算，自定义函数，多项式运算等等——还是把 MATLAB 打开吧⑨。

参考文献

Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*, 2001

【责任编辑：worldguy@hitbbs】

标准 C++ 类 std::string 的内存共享和 Copy-On-Write 技术

作者：陈皓(haoel@hotmail.com)

1. 概念

Scott Meyers 在《More Effective C++》中举了个例子，不知你是否还记得？在你还在上学的时候，你的父母要你不要看电视，而去复习功课，于是你自己关在房间里，做出一副正在复习功课的样子，其实你在干着别的诸如给班上的某位女生写情书之类的事，而一旦你的父母出来在你房间要检查你是否在复习时，你才真正捡起课本看书。这就是“拖延战术”，直到你非要做时候才去做。

当然，这种事情在现实生活中时往往会上出事，但其在编程世界中摇身一变，就成为了最有用的技术，正如 C++ 中的可以随处声明变量的特点一样，Scott Meyers 推荐我们，在真正需要一个存储空间时才去声明变量（分配内存），这样会得到程序在运行时最小的内存花销。执行到那才会去做分配内存这种比较耗时的工作，这会给我们的程序在运行时有比较好的性能。毕竟，20% 的程序运行了 80% 的时间。

当然，拖延战术还并不是这样一种类型，这种技术被我们广泛地应用着，特别是在操作系统当中，当一个程序运行结束时，操作系统并不会急着把其清除出内存，原因是有可能程序还会马上再运行一次（从磁盘把程序装入到内存是个很慢的过程），而只有当内存不够用了，才会把这些还驻留内存的程序清出。

写时才拷贝 (Copy-On-Write) 技术，就是编程界“懒惰行为”——拖延战术的产物。举个例子，比如我们有个程序要写文件，不断地根据网络传来的数据写，如果每一次 fwrite 或是 fprintf 都要进行一个磁盘的 I/O 操作的话，都简直就是性能上巨大的损失，因此通常的做法是，每次写文件操作都写在特定大小的一块内存中（磁盘缓存），只有当我们关闭文件时，才写到磁盘上（这就是为什么如果文件不关闭，所写的东西会丢失的原因）。更有甚者是文件关闭时都不写磁盘，而一直等到关机或是内存不够时才写磁盘，Unix 就是这样一个系统，如果非正常退出，那么数据就会丢失，文件就会损坏。

呵呵，为了性能我们需要冒这样大的风险，还好我们的程序是不会忙得忘了还有一块数据需要写到磁盘上的，所以这种做法，还是很有必要的。

2. 标准 C++ 类 std::string 的 Copy-On-Write

在我们经常使用的 STL 标准模板库中的 string 类，也是一个具有写时才拷贝技术的类。C++ 曾在性能问题上被广泛地质疑和指责过，为了提高性能，STL 中的许多类都采用了 Copy-On-Write 技术。这种偷懒的行为的确使使用 STL 的程序有着比较高要性能。

这里，我想从 C++ 类或是设计模式的角度为各位揭开 Copy-On-Write 技术在 string 中实现的面纱，以供各位在用 C++ 进行类库设计时做一点参考。

在讲述这项技术之前，我想简单地说明一下 string 类内存分配的概念。通常，string 类中必有一个私有成员，其是一个 char*，用户记录从堆上分配内存的地址，其在构造时分配内存，在析构时释放内存。因为是从堆上分配内存，所以 string 类在维护这块内存上是格外小心的，string 类在返回这块内存地址时，只返回 const char*，也就是只读的，如果你要写，你只能通过 string 提供的方法进行数据的改写。

2.1 特性

由表及里，由感性到理性，我们先来看一看 `string` 类的 Copy-On-Write 的表面特征。让我们写下下面的一段程序：

```
#include <cstdio>
#include <string>

using namespace std;

main()
{
    string str1 = "hello world";
    string str2 = str1;

    printf ("Sharing the memory:\n");
    printf ("\tstr1's address: %x\n", str1.c_str() );
    printf ("\tstr2's address: %x\n", str2.c_str() );

    str1[1]='q';
    str2[1]='w';

    printf ("After Copy-On-Write:\n");
    printf ("\tstr1's address: %x\n", str1.c_str() );
    printf ("\tstr2's address: %x\n", str2.c_str() );

    return 0;
}
```

这个程序的意图就是让第二个 `string` 通过第一个 `string` 构造，然后打印出其存放数据的内存地址，然后分别修改 `str1` 和 `str2` 的内容，再查一下其存放内存的地址。程序的输出是这样的（我在 VC6.0 和 g++ 2.95 都得到了同样的结果）：

```
> g++ -o stringTest stringTest.cpp
> ./stringTest
Sharing the memory:
    str1's address: 343be9
    str2's address: 343be9
After Copy-On-Write:
    str1's address: 3407a9
    str2's address: 343be9
```

从结果中我们可以看到，在开始的两个语句后，`str1` 和 `str2` 存放数据的地址是一样的，而在修改内容后，`str1` 的地址发生了变化，而 `str2` 的地址还是原来的。从这个例子，我们可以看到 `string` 类的 Copy-On-Write 技术。

2.2 深入

在深入这前，通过上述的演示，我们应该知道在 `string` 类中，要实现写时才拷贝，需要解决两个问题，一个是内存共享，一个是 Copy-On-Wirte，这两个主题会让我们产生许多疑问，还是让我们带着这样几个问题来学习吧：

- 1、 Copy-On-Write 的原理是什么？
- 2、 `string` 类在什么情况下才共享内存的？
- 3、 `string` 类在什么情况下触发写时才拷贝（Copy-On-Write）？
- 4、 Copy-On-Write 时，发生了什么？

5、Copy-On-Write 的具体实现是怎么样的？

喔，你说只要看一看 STL 中 string 的源码你就可以找到答案了。当然，当然，我也是参考了 string 的父模板类 basic_string 的源码。但是，如果你感到看 STL 的源码就好像看机器码，并严重打击你对 C++ 自信心，乃至产生了自己是否懂 C++ 的疑问，如果你有这样的感觉，那么还是继续往下看我的这篇文章吧。

OK，让我们一个问题一个问题地探讨吧，慢慢地所有的技术细节都会浮出水面的。

2.3 Copy-On-Write 的原理是什么？

有一定经验的程序员一定知道，Copy-On-Write 一定使用了“引用计数”，是的，必然有一个变量类似于 RefCnt。当第一个类构造时，string 的构造函数会根据传入的参数从堆上分配内存，当有其它类需要这块内存时，这个计数为自动累加，当有析构时，这个计数会减一，直到最后一个析构时，此时的 RefCnt 为 1 或是 0，此时，程序才会真正的 Free 这块从堆上分配的内存。

是的，引用计数就是 string 类中写时才拷贝的原理！

不过，问题又来了，这个 RefCnt 该存在在哪里呢？如果存放在 string 类中，那么每个 string 的实例都有各自的一套，根本不能共有一个 RefCnt，如果是声明成全局变量，或是静态成员，那就是所有的 string 类共享一个了，这也不行，我们需要的是一个“民主和集中”的一个解决方法。这是如何做到的呢？呵呵，人生就是一个糊涂后去探知，知道后和又糊涂的循环过程。别急别急，在后面我会给你一一道来的。

2.3.1 string 类在什么情况下才共享内存的？

这个问题的答案应该是明显的，根据常理和逻辑，如果一个类要用另一个类的数据，那就可以共享被使用的内存了。这是很合理的，如果你不用我的，那就不用共享，只有你使用我的，才发生共享。

使用别的类的数据时，无非有两种情况，1) 以别的类构造自己，2) 以别的类赋值。第一种情况时会触发拷贝构造函数，第二种情况会触发赋值操作符。这两种情况我们都可以在类中实现其对应的方法。对于第一种情况，只需要在 string 类的拷贝构造函数中做点处理，让其引用计数累加；同样，对于第二种情况，只需要重载 string 类的赋值操作符，同样在其中加上一点处理。

唠叨几句：

1) 构造和赋值的差别

对于前面那个例程中的这两句：

```
string str1 = "hello world";
string str2 = str1;
```

不要以为有“=”就是赋值操作，其实，这两条语句等价于：

```
string str1 ("hello world"); //调用的是构造函数
string str2 (str1); //调用的是拷贝构造函数
```

如果 str2 是下面的这样情况：

```
string str2; //调用参数默认为空串的构造函数: string str2("");
str2 = str1; //调用 str2 的赋值操作: str2.operator=(str1);
```

2) 另一种情况

```
char tmp[]="hello world";
string str1 = tmp;
string str2 = tmp;
```

这种情况下会触发内存的共享吗？想当然的，应该要共享。可是根据我们前面所说的共享内存的情况，两个 string 类的声明和初始语句并不符合我前述的两种情况，所以其并不发生内存共享。而且，C++现有特性也无法让我们做到对这种情况进行类的内存共享。

2.3.2 string 类在什么情况下触发写时才拷贝（Copy-On-Write）？

哦，什么时候会发现写时才拷贝？很显然，当然是在共享同一块内存的类发生内容改变时，才会发生 Copy-On-Write。比如 string 类的[]、=、+=、+、操作符赋值，还有一些 string 类中诸如 insert、replace、append 等成员函数。

修改数据才会触发 Copy-On-Write，不修改当然就不会改啦。这就是托延战术的真谛，非到要做的时候才去做。

2.3.3 Copy-On-Write 时，发生了什么？

我们可能根据那个访问计数来决定是否需要拷贝，参看下面的代码：

```
If ( RefCnt>0 ) {  
    char* tmp = (char*) malloc(strlen(_Ptr)+1);  
    strcpy(tmp, _Ptr);  
    _Ptr = tmp;  
}
```

上面的代码是一个假想的拷贝方法，如果有别的类在引用（检查引用计数来获知）这块内存，那么就需要把更改类进行“拷贝”这个动作。

我们可以把这个拷的运行封装成一个函数，供那些改变内容的成员函数使用。

2.2.4 Copy-On-Write 的具体实现是怎么样的？

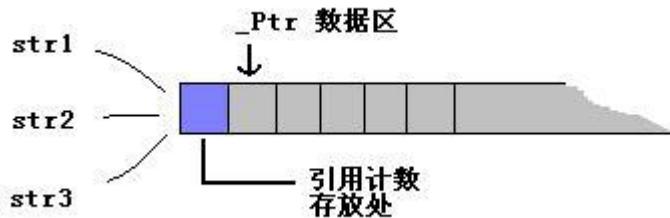
最后的这个问题，我们主要解决的是那个“民主集中”的难题。请先看下面的代码：

```
string h1 = "hello";  
string h2= h1;  
string h3;  
h3 = h2;  
  
string w1 = "world";  
string w2("");  
w2=w1;
```

很明显，我们要让 h1、h2、h3 共享同一块内存，让 w1、w2 共享同一块内存。因为，在 h1、h2、h3 中，我们要维护一个引用计数，在 w1、w2 中我们又要维护一个引用计数。

如何使用一个巧妙的方法产生这两个引用计数呢？我们想到了 string 类的内存是在堆上动态分配的，既然共享内存的各个类指向的是同一个内存区，我们为什么不在这块区上多分配一点空间来存放这个引用计数呢？这样一来，所有共享一块内存区的类都有同样的一个引用计数，而这个变量的地址既然是在共享区上的，那么所有共享这块内存的类都可以访问到，也就知道这块内存的引用者有多少了。

请看下图：



于是，有了这样一个机制，每当我们为 `string` 分配内存时，我们总是要多分配一个空间用来存放这个引用计数的值，只要发生拷贝构造或是赋值时，这个内存的值就会加一。而在内容修改时，`string` 类为查看这个引用计数是否为 0，如果不为零，表示有人在共享这块内存，那么自己需要先做一份拷贝，然后把引用计数减去一，再把数据拷贝过来。下面的几个程序片段说明了这两个动作：

```
//构造函数(分存内存)
string::string(const char* tmp)
{
    _Len = strlen(tmp);
    _Ptr = new char[ _Len+1];
    strcpy( _Ptr, tmp );
    _Ptr[ _Len]=0; // 设置引用计数
}
//拷贝构造(共享内存)
string::string(const string& str)
{
    if (*this != str) {
        this-> _Ptr = str.c_str(); //共享内存
        this-> Len = str.size();
        this-> _Ptr[_Len+1]++; //引用计数加一
    }
}

//写时才拷贝 Copy-On-Write
char& string::operator[](unsigned int idx)
{
    if (idx > Len || _Ptr == 0) {
        static char nullchar = 0;
        return nullchar;
    }
    _Ptr[_Len+1]--; //引用计数减一
    char* tmp = new char[_Len+1];
    strncpy( tmp, _Ptr, Len );
    _Ptr = tmp;
    _Ptr[_Len+1]=0; // 设置新的共享内存的引用计数
    return _Ptr[idx];
}
```

哈哈，整个技术细节完全浮出水面。

不过，这和 STL 中 `basic_string` 的实现细节还有一点点差别，在你打开 STL 的源码时，你会发现其取引用计数是通过这样的访问：`_Ptr[-1]`，标准库中，把这个引用计数的内存分配在了前面（我给出来的代码是把引用计数分配到了后面，这很不好），分配在前的好处是当 `string` 的长度扩展时，只需要在后面扩展其内存，而不需要移动引用计数的内存存放位置，这又节省了一点时间。

STL 中的 `string` 的内存结构就像我前面画的那个图一样，`_Ptr` 指着是数据区，而 `RefCnt` 则在 `_Ptr-1` 或是 `_Ptr[-1]` 处。

2.4 臭虫 Bug

是谁说的“有太阳的地方就会有黑暗”？或许我们中的许多人都很迷信标准的东西，认为其是久经考验，不可能出错的。呵呵，千万不要有这种迷信，因为任何设计再好，编码再好的代码在某一特定的情况下都会有 Bug，STL 同样如此，string 类的这个共享内存/写时才拷贝技术也不例外，而且这个 Bug 或许还会让你的整个程序 crash 掉！

不信？！那么让我们来看一个测试案例：

假设有一个动态链接库（叫 myNet.dll 或 myNet.so）中有这样一个函数返回的是 string 类：

```
string GetIPAddress(string hostname)
{
    static string ip;
    .....
    .....
    return ip;
}
```

而你的主程序中动态地载入这个动态链接库，并调用其中的这个函数：

```
main()
{
    //载入动态链接库中的函数
    hDll = LoadLibrary(...);
    pFun = GetProcAddress(hDll, "GetIPAddress");

    //调用动态链接库中的函数
    string ip = (*pFun) ("host1");
    .....
    .....

    //释放动态链接库
    FreeLibrary(hDll);
    .....
    cout << ip << endl;
}
```

让我们来看看这段代码，程序以动态方式载入动态链接库中的函数，然后以函数指针的方式调用动态链接库中的函数，并把返回值放在一个 string 类中，然后释放了这个动态链接库。释放后，输入 ip 的内容。

根据函数的定义，我们知道函数是“值返回”的，所以，函数返回时，一定会调用拷贝构造函数，又根据 string 类的内存共享机制，在主程序中变量 ip 是和函数内部的那个静态 string 变量共享内存（**这块内存区是在动态链接库的地址空间的**）。而我们假设在整个主程序中都没有对 ip 的值进行修改过。那么在当主程序释放了动态链接库后，那个共享的内存区也随之释放。所以，以后对 ip 的访问，必然会造成内存地址访问非法，造成程序 crash。即使你在以后没有使用到 ip 这个变量，那么在主程序退出时也会发生内存访问异常，因为程序退出时，ip 会析构，在析构时就会发生内存访问异常。

内存访问异常，意味着两件事：1) 无论你的程序再漂亮，都会因为这个错误变得暗淡无光，你的声誉也会因为这个错误受到损失。2) 未来的一段时间，你会被这个系统级错误所煎熬（在 C++世界中，找到并排除这种内存错误并不是一件容易的事情）。这是 C/C++程序员永远的心头之痛，千里之堤，溃于蚁穴。而如果你不清楚 string 类的这种特征，在成千上万行代码中找这样一个内存异常，简直就是一场噩梦。

备注：要改正上述的 Bug，有很多种方法，这里提供一种仅供参考：

```
string ip = (*pFun) ("host1").cstr();
```

3. 后记

文章到这里也应该结束了，这篇文章的主要有以下几个目的：

- 1) 向大家介绍一下写时才拷贝/内存共享这种技术。
- 2) 以 STL 中的 string 类为例，向大家介绍了一种设计模式。
- 3) 在 C++世界中，无论你的设计怎么精巧，代码怎么稳固，都难以照顾到所有的情况。智能指针更是一个典型的例子，无论你怎么设计，都会有非常严重的 BUG。
- 4) C++是一把双刃剑，只有了解了原理，你才能更好的使用 C++。否则，必将引火烧身。如果你在设计和使用类库时有一种“玩 C++就像玩火，必须千万小心”的感觉，那么你就入门了，等你能把这股“火”控制的得心应手时，那才是学成了。

【责任编辑：sun@hitbbs】

深度探索编译器安全检查

原文作者: Brandon Bray
译者: 杨新开 (vangxk@neusoft.com)

简介

安全是高质量软件重点关注的问题，最让人害怕、最多被误解的就是缓冲区溢出。现在，提及缓冲区溢出足以让人们停下来，仔细听。技术细节在传抄过程中逐渐丢失，大部分人关注的并不是最紧要、最基础的问题。为了解决这个问题，Visual C++ .NET 引入了安全检查来帮助开发者确定缓冲区溢出。

什么是缓冲区溢出？

缓冲区是一块内存，通常是数组的形式。当没有校验数组的长度时，可能会写出缓冲区的边界。如果这样的行为发生的地址比缓冲区的内存地址高，称为缓冲区溢出；类似的，如果这样的行为发生的地址比缓冲区的内存地址低，称为缓冲区下溢。缓冲区下溢的发生率明显少于缓冲区溢出，但是，正如本文的后面所描述，它确实发生过。向一个正在运行进程注入代码的缓冲区溢出被称为可以用缓冲区溢出。

一些文档化的函数，例如 strcpy, gets, scanf, sprintf, strcat 等，本身很容易受到缓冲区溢出的攻击，所以不推荐使用他们。一个简单的例子说明了这些函数的危险性(编者按：原文中的程序有个错误)：

```
int vulnerable1(char * pStr) {
    int nCount = 0;
    char strBuff[_MAX_PATH], *pBuff = strBuff;

    strcpy(pBuff, pStr);
    for(; pBuff; pBuff++)
        if (*pBuff == '\\') nCount++;
    return nCount;
}
```

这些代码有个明显的弱点——如果由 pStr 指向的缓冲区长度大于 _MAX_PATH，那么 pBuffer 参数可能溢出。如果包含一句 assert(strlen(pStr) < _MAX_PATH) 就能够在 debug 版本下发现这个错误，但是 release 版本不行。用这些容易受到攻击的函数被认为是坏的实践。技术上来讲更不容易受到攻击的相似的函数确实存在，如 strncpy, strncat 和 memcpy。这些函数的问题是开发者来验证缓冲区的长度，而不是编译器。下面的函数展示一个普遍的错误：

```
#define BUflen 16
void vulnerable2(void)
{
    wchar_t buf[BUflen];
    int ret;
    ret = MultiByteToWideChar(CP_ACP, 0, "1234567890123456789", -1, buf, sizeof(buf));
    printf("%d\n", ret);
}
```

这种情况下，字节的个数用来标示缓冲区的大小，而不是字符的个数，于是发生了缓冲区溢出。为了修正这个可攻击点，MultiByteToWideChar 的最后一个参数因该使用 sizeof(buf)/sizeof(buf[0])。vulnerable1 和 vulnerable2 都是很普遍的错误，并且可以很容易的预防。然而，如果由于代码 Reviewer 的疏忽，这些潜在的安全漏洞可能发布到产品中。这就是为什么 Visual C++ .NET 引入了安全检查，它可以阻止 vulnerable1 和 vulnerable2 函数

数中的缓冲区溢出向容易受到攻击的应用程序注入恶意代码。

X86 栈的分析

为了理解如何利用缓冲区溢出以及安全检查如何工作，必须完全理解堆栈的布局结构。在 X86 体系下，堆栈向下增长，意味着新创建数据的地址小于早期压入栈中元素的地址。每一个函数创建一个新的、有如下布局的栈帧，注意高内存地址在列表的顶部：

- Function parameters
- Function return address
- Frame pointer
- Exception Handler frame
- Locally declared variables and buffers
- Callee save registers

从以上布局很明显的可以看出，缓冲区溢出有可能覆盖掉比该缓冲区分配的早的变量，异常处理帧，栈帧指针，返回地址，函数参数。为了接管程序的执行，一个值必须写进数据中，该数据的值被装载进 EIP 寄存器中。函数的返回地址就是一个这样的数据。典型的利用缓冲区溢出是覆盖函数返回地址，让函数的返回指令将返回地址加载到 EIP 中。

数据元素按照如下方式存入栈中。函数调用之前将函数的参数压入栈中。参数从右到左被压入栈中。CALL 指令将函数的返回地址压入栈中，它存储 EIP 寄存器的当前值。栈帧指针是前一个 EB 寄存器的值，当没有发生 FPO 优化时，也压入栈中。因此，栈帧指针不总是在栈中。如果函数包括了 try/catch 或者任何其他形式的异常处理结构，编译器将在栈中包含异常处理信息。之后，是局部声明的变量和分配的缓冲区。这些分配的顺序可能根据优化实施而作改变。最后是调用者保存的寄存器如 ESI, EDI, EBX，如果他们在函数执行时被使用。

运行时检查

缓冲区溢出是 c、c++程序员普遍犯的错误，也是潜在的最危险的。Visual C++ .NET 提供了工具，它可以使开发者在开发阶段很容易发现这些错误并修正。Visual C++ 6.0 中的/GZ 开关在 Visual C++ .NET 中的/RTC1 中获得了新生。/RTC1 开关是/RTsu 的别名，其中 s 代表堆栈检查，u 代表未初始化变量检查。所有在堆栈中分配的缓冲区在边界处设置了标签。因此，缓冲区溢出、下溢可以被捕获。尽管小的缓冲区溢出可能不会改变程序的执行，它可以改变附近的数据，而这都不会被觉察到。

运行时检查对于那些不仅想写安全的代码、而且关心编写正确代码的基本问题的开发者很有帮助。然而运行时检查仅仅工作在 debug 版本下，该特性从没有设计为在产品代码中可用。但是，在产品代码中进行这样的检查有很明显的价值。做这些运行时检查需要一小部分的性能损失。结果，Visual C++ .NET 引入了/GS 开关。

/GS 开关做什么

/GS 开关在缓冲区和返回地址间提供了一个"Speed bump"或 cookie。如果一个溢出覆盖了返回地址，那么它也覆盖了放在缓冲区和他之间的 Cookie，新的堆栈布局如下：

- Function parameters
- Function return address
- Frame pointer
- Cookie
- Exception Handler frame
- Locally declared variables and buffers
- Callee save

Cookie 在以后会更详细的检查。随着这些安全检查的加入，函数的执行也改变了。首先，当一个函数执行时，第一条要执行的指令在函数的 prolog 中。至少，prolog 为堆栈中的局部变量分配空间，例如如下指令：

```
sub esp,20h
```

这条指令为函数中的局部变量预留了 32 字节。当函数使用/GS 开关编译时，函数 prolog 将预留另外的 4 个字节，三个如下另外的指令：

```
sub esp,24h
mov eax,dword ptr [__security_cookie (408040h)]
xor eax,dword ptr [esp+24h]
mov dword ptr [esp+20h],eax
```

prolog 包含了一个提取 cookie 拷贝的指令，接着一条指令是将 cookie 和返回地址进行 XOR 操作，最后将 cookie 存储在紧挨着返回地址的下面。从以上来看，函数象正常一样执行。当函数返回时，最后要执行的是函数的 epilog，它和 prolog 正好相反。如果没有安全检查，它将回收堆栈空间、返回，就像如下指令：

```
add esp,20h
ret
```

当使用/GS 编译时，安全检查也放在 epilog 中：

```
mov ecx,dword ptr [esp+20h]
xor ecx,dword ptr [esp+24h]
add esp,24h
jmp __security_check_cookie (4010B2h)
```

查询堆栈的 cookie 的拷贝，然后和返回地址进行 XOR 操作，ECX 寄存器应该包含和存储在 __security_cookie 变量中的原始 cookie 相同的内容。接着回收堆栈空间，然后不是 RET 指令，而是执行 JMP 指令，跳转到 __security_check_cookie 例程。

__security_check_cookie 例程是很直观的。如果 cookie 没有被改变，它执行 RET 指令并结束函数调用。否则，它调用 report_failure 函数，report_failure 接着调用 __security_error_handler(_SECERR_BUFFER_OVERRUN, NULL)。这些函数都定义在 C 运行库 (CRT) 的源文件 seccook.c 中。

错误处理器

使这些安全检查起作用需要 CRT 的支持。当安全检查失败时，程序的控制需要交给 __security_error_handler，以下是它的处理概要：

```
void __cdecl __security_error_handler(int code, void *data)
{
    if (user handler != NULL) {
        __try {
            user_handler(code, data);
        } except (EXCEPTION_EXECUTE_HANDLER) {}
    } else { //...prepare outmsg...
        __crtMessageBoxA(      outmsg,      "Microsoft      Visual      C++      Runtime      Library",
MB_OK|MB_ICONHAND|MB_SETFOREGROUND|MB_TASKMODAL);
    }
    _exit(3);
}
```

缺省情况下，安全检查失败的应用程序弹出显示信息为" Buffer overrun detected!"的对话框，当关闭对话框后终止应用程序。CRT 库提供给开发者定制不同的、能够更好的处理缓冲区溢出的处理器功能。函数 `_set_security_error_handler` 通过在变量 `user_handler` 中存储用户提供的处理器的方式来安装 handler。以下例子说明：

```
void __cdecl report_failure(int code, void * unused)
{
    if (code == SECERR_BUFFER_OVERRUN)
        printf("Buffer overrun detected!\n");
}
void main()
{
    set security error handler(report failure);
    // More code follows
}
```

缓冲区溢出发生时，将会向控制台窗口打印一条消息，而不是弹出消息窗口。尽管用户处理器没有显示的终止程序，但是当处理器返回时，`_security_error_handler` 通过调用 `_exit(3)` 来终止程序。函数 `_security_error_handler` 和 `_set_security_error_handler` 都在 CRT 的 `secfail.c` 文件中

讨论在用户处理器中应该怎么做是有用的。普遍的动作时抛出异常。然而，因为异常信息存储在堆栈中，抛出异常会将控制传递给异常栈。为了防止这种行为发生，`_security_error_handler` 函数中调用用户函数时使用 `try/_except` 来捕捉所有异常，然后终止程序。开发者不应该调用 `DebugBreak` 因为它会导致异常，也不应该调用 `longjmp`。用户处理器应该做的是报告错误，尽可能创建一个日志以便修正这个问题。

有时，开发者或许想重写函数 `_security_error_handler`，而不是用函数 `_set_security_error_handler` 来达到同样的目的。重写容易出错，主处理器非常重要，如果没有正确的实现将导致危险的结果。

Cookie 的值

Cookie 是一个和指针同样大小的随机数，意味着在 X86 体系下，cookie 是 4 个字节长。它的值存储在 CRT 全局变量 `_security_cookie` 中。它的值由在 CRT `seccinit.c` 文件中的函数 `_security_init_cookie` 来初始化。Cookie 的随机性来自于 CPU 处理器的计数器。每一个影像文件（使用/GS 编译的 DLL、EXE）在装载时有一个不同的 cookie。

当时用/GS 编译器开关编译程序时可能有两个问题。第一、不包含 CRT 支持的程序

将缺少随机的 cookie，因为 CRT 初始化时调用 `_security_init_cookie`。如果在装载时 cookie 没有被初始化，如果有缓冲区溢出，应用程序还是有可能被攻击。为了解决这个问题，应用程序在启动时需要显示的调用 `_security_init_cookie`。第二、调用文档化的函数来初始化的

遗留的应用程序可能遇到不可预期的安全检查失败。例如下面的例子：

```
DllEntryPoint(...) {
    char buf[ MAX_PATH]; // A buffer that triggers security checks
    ...
    _CRT_INIT();
    ...
}
```

问题是 `_CRT_INIT` 改变了已经存在的用来安全检查的 cookie 的值。因为 cookie 的值在函数退出时和原来的值不同，安全检查认为发生了缓冲区溢出。解决办法是避免在调用 `_CRT_INIT` 之前声明缓冲区。现在可以使用 `_alloca` 在堆栈上分配缓冲区作为回避方法，因为如果使用函数 `_alloca` 分配缓冲区，编译器不会产生安全检查。这种回避方法不保证在以后的 Visual C++ 版本中适用。

性能影响

必须平衡程序的性能和安全检查。Visual C++编译器组致力于将性能影响降低到最小。大多说情况下，性能影响不超过 2%。实际上，经验显示对大多说的应用程序、包括高性能的服务器端程序来讲，性能影响是微乎其微的。

使性能不受影响的最重要的因素是只有那些容易受到攻击的函数才执行安全检查。现在，容易受到攻击的函数为在堆栈中分配缓冲区的函数。字符串缓冲区如果分配多于四个字节、缓冲区中每个元素时 1 到 2 个字节，就容易受到攻击。小缓冲区不容易受到攻击并且限制进行安全检查的函数的数量就限制了代码的增长。大部分的可执行程序因为适用/GS 编译引起的代码增长时微乎其微的。

例子

因此，/GS 开关并没有修正缓冲区溢出，但是它可以阻止攻击者利用缓冲区进行攻击。当时用/GS 开关编译 vulnerable1 、vulnerable2 时，溢出就不会被利用。缓冲区溢出发生在最后一个动作的函数可以免于被攻击。因为如果溢出发生在函数执行的早期，安全检查或者没有机会执行、或者安全检查本身已被攻击，象如下例子。

例子 1

```
class Vulnerable3 {
public:
    int value;
    Vulnerable3() { value = 0; }
    virtual ~Vulnerable3() { value = -1; }
};

void vulnerable3(char * pStr) {
    Vulnerable3 * vuln = new Vulnerable3;
    char buf[20];
    strcpy(buf, pStr);
    delete vuln;
}
```

这种情况下，在栈中分配了含有虚函数的对象的指针。因为对象含有虚函数，对象包含一个 vtable 指针。供给者能提供一个恶意的 pStr 并溢出 buf。函数返回前，delete 操作符调用 vuln 的虚函数。调用需要在 vtable 中查找析构函数，它已经被接管了。在函数返回前，程序已经北接管，所以安全检查根本没有检测到缓冲区溢出发生。

例子 2

```
void vulnerable4(char *bBuff, in cbBuff) {
    char bName[128];
    void (*func) () = MyFunction;
    memcpy(bName, bBuff, cbBuff);
    (func) ();
}
```

这种情况下，函数容易受到指针修改攻击。当编译器为这两个局部变量分配空间时，它把 func 变量放在 bName 之前。因为这种布局优化器可以提升代码的效率。很不幸，这允许攻击者一个为 bBuff 提供恶意的值。攻击者同样可以提供 cbBuff 的值，它标示着 bBuff 的大小。函数忽略了验证 cbBuff 小于等于 128。调用 memcpy 导致了缓冲区溢出，覆盖了 func 的值。因为在 vulnerable4 返回之前首先调用 func，在进行安全检查之前，代码已经被攻击了。

例子 3

```
int vulnerable5(char * pStr) {
    char buf[32];
```

<http://emag.csdn.net>

<http://purec.binghua.com>

```
char * volatile pch = pStr;
strcpy(buf, pStr);
return *pch == '\0';
}
int main(int argc, char* argv[]) {
    __try { vulnerable5(argv[1]); }
    __except(2) { return 1; }
    return 0;
}
```

这个程序展示了一个特别难的问题，因为它使用了结构化异常处理。如前面提及，使用异常处理的函数将把异常处理信息，例如合适的异常处理函数，放在栈中。本例中，main 函数的异常处理帧因为函数 vulnerable5 的缺陷而可能被攻击。攻击者利用溢出 buf 来破坏 pch、main 函数的异常处理帧。因为 vulnerable5 函数后来引用了 pch，如果攻击者覆盖它的值为 0，这将导致访问异常。在堆栈展开的过程中，操作系统在异常处理帧中查找处理该异常的异常处理函数。因为异常处理帧已经被破坏，操作系统可能将程序的控制权交给由攻击者提供的任意代码。安全检查将不能够检查这样的缓冲区溢出，因为函数没有正常返回。

近来一些很流行的攻击都是利用异常处理。其中最流行的 Code Red 病毒出现在 2001 年夏。Window XP 已经创建了一个环境，在该环境下，通过异常处理进行攻击将会更难，因为异常处理函数的地址不在栈中，而且调用异常处理函数之前清除所有的寄存器的值。

例子 4

```
void vulnerable6(char * pStr) {
    char buf[_MAX_PATH];
    int * pNum;
    strcpy(buf, pStr);
    sscanf(buf, "%d", pNum);
}
```

不象以前其他的例子，当时用/GS 开关编译此函数时，攻击者不能简单的通过缓冲区溢出来获得程序的控制权。如果想获得程序的控制权需要两阶段的攻击。pNum 将被分配在 buf 之前使得它可以被 pStr 提供的任意的值重写。攻击者将不得不选择四个字节进行重写。如果缓冲区重写超过了 cookie，存储在 user_handle 中的用户提供的处理器或者或者存储在 security_cookie 中的默认处理器会接管程序的运行。如果没有覆盖 cookie，攻击者将选择函数的返回地址作为不包含安全检查的函数。这种情况下，程序正常的执行，从函数中正常返回，没有意识到缓冲区溢出；一段时间后，程序悄悄的被接管。

容易受到攻击的代码可能受到另外的攻击如，发生在堆中的缓冲区溢出，它不能够被/GS 开关检查出来。数组索引越界攻击，它对数组中的某一个位置进行存取，而不是对数组进行连续写入，这样的问题/GS 开关也不能检查出来。一个未检查的数组索引可以访问内存的任意部分，而不会修改 cookie 的内容。另外一种未检查的索引是有符号、无符号的不匹配。如果索引是有符号整数，简单的验证索引小于数组的大小也是不够的。最后，/GS 开关不能够检查缓冲区下溢。

结论

很明显，缓冲区溢出是应用程序的一个非常关键的缺陷。没有比写出紧凑、安全的代码更重要。尽管大多数观点认为，少量的缓冲区溢出很难被发现。在编写安全代码方面，/GS 开关对开发者是很有帮助的。但是，它解决不了代码中的缓冲区溢出的问题。尽管安全检查在某种程度上阻止了缓冲区被利用，但程序仍然终止了，一种拒绝服务的攻击，特别是服务器端代码。使用/GS 开关是一种安全的方法来减少在没有意识到的情况下，受到缓冲区溢出的攻击。

尽管存在能够标记可能受到攻击的代码的工具，例如本文所讨论的，但是他们都是由缺点的。被好的代码 Review 人员检查过得好的代码比什么都更可靠。Michael Howard 和 David LeBlanc 的< Writing Secure Code > 在编写高度安全的应用程序方面，提供了很多其他的、可以降低被攻击的方法。

原文参考：

http://msdn.microsoft.com/visualc/default.aspx?pull=/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksind_epth.asp

【审阅： sun@hitbbs】

对《浅析 C 语言函数传递机制及对变参函数的处理》 一文中有关“浮点参数压栈”一段描述的更正

哈尔滨工业大学并行计算实验室
谢煜波

在第二期杂志(《纯 C 论坛·电子杂志》2004.11)中，笔者的一篇题为《浅析 C 语言函数传递机制对参数函数的处理》一文在有关“浮点参数压栈”问题的描述上有不当之处，在此特对原文所述内容进行修正。

在第二期杂志第 88 页最后一段，原文描述的是 C 语言在对浮点参数压栈时，总是先用 flds 指令把 float 型的数据载入浮点寄存器，然后用 fstpl 把这个数再存回到堆栈中，即从 float 到 CPU 的 extend double，再由 CPU 中的 extend double 到 double，亦即，float 型参数在传递进入函数内部的时候都通过浮点寄存器转换成 double 进行压栈。

原来的这段描述在对于 printf(.....) 这一类未在函数的参数表中指明参数确切类型的函数参数传递过程中才有效，比如，对于 printf，由于其声明类似于：printf(const char *s, ...)，可见，在这个声明的参数表中没有指出确切的参数类型，而是使用的变参形式的参数表，这种情况下，比如：

```
void f()
{
    float a = 0.5 ;
    printf( "%f" , a ) ;
}
```

其中的 a 就被编译器转换成 double 压栈，具体的压栈方式如原文中所述，下面就是我们对上面一段语句的反汇编结果：

```
f:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $0x3f000000, -4(%ebp)
    subl $4, %esp
    flds -4(%ebp)
    leal -8(%esp), %esp
    fstpl (%esp)
    pushl $.LC0
    call printf
    addl $16, %esp
    leave
    ret
```

从中可以清楚的看出，这与我们原文的描述是一致的。

但是，如果被调用的函数的参数表明确的指出了参数的类型，比如，对于函数 void f(float a); 由于明确指出了其参数是一个 float 型的参数，故而，在调用的时候，C 编译器将不再能过浮点寄存器将其转换成 double 型的参数，而是直接压栈。请看下面一段代码：

```
void f( float a )
{
}
void k()
{
    float b = 0.5 ;
    f( b ) ;
}
```

把上面这段代码反汇编一下，能得到如下的反汇编结果：

```
k:
```

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $0x3f000000, -4(%ebp)
subl $12, %esp
pushl -4(%ebp)
call f
addl $16, %esp
leave
ret
```

可以看见，这里已经没有了前面出现的 flds, fstpl 指令，而直接用 pushl 指令将参数值压入了堆栈中。

现在，我们在将前面的代码改一下：

```
voif f( float a , ... ) /* 这里使用省略的参数表 */
{
}
void k()
{
    float b = 0.5 ;
    f( b , b ) ;
}
```

同样将其反汇编，我们能得到如下的反汇编结果：

```
k:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $0x3f000000, -4(%ebp)
subl $4, %esp
flds -4(%ebp)
leal -8(%esp), %esp
fstpl (%esp)
pushl -4(%ebp)
call f
addl $16, %esp
leave
ret
```

这里就非常显然了，你看，首先它通过 flds,fstpl 指令压入了一个参数，然后又通过 pushl 压入最后一个压数，由于 C 语言函数参数的默认压栈顺序是从右到左，故而这最后一个通过 pushl 指令直接压栈的应是参数表中的第一个参数，而 f(float a , ...) 声明中的第一个参数是被明确说明是 float 型的，因此，它通过 pushl 指令直接压栈，而后面的参数是由的省略参数的形式，故而，编译器通过 flds,fstpl 指令将 float 参数转换成了 double 参数压栈。

相信通过现在的描述，您应当清楚许多了吧。

参考资料：

《C/C++深层探索》姚新颜，人民邮电出版社

推箱子游戏的自动求解

作者: hellwolf(hellwolf@seu.edu.cn)

1. 简介

推箱子，又称搬运工，是一个十分流行的单人智力游戏。玩家的任务是在一个仓库中操纵一个搬运工人，将 N 个相同的箱子推到 N 个相同的目的地。推箱子游戏出现在计算机中最早起源于 1994 年台湾省李果兆开发的仓库世家，又名仓库番，箱子只可以推，不可以拉，而且一次只能推动一个。它的规则如此简单，但是魅力却是无穷的。但是人毕竟思考的深度和速度有限，我们是否可以利用计算机帮助我们求解呢？

2. 游戏的基础部件

首先，我选择了 C++ 来做这个程序。这一步并没有太大的难度，不少编程爱好者肯定也写过不少的小游戏。所以这里我只简要的为后面的叙述必要的铺垫。

我将推箱子游戏的数据和操作封装为一个 BoxRoom 类，下面是后面的自动求解算法用到的成员函数和数据结构。

(1)

```
//移动一格,如果有箱子就推
short      MovePush(Direction);
//移动到一点,并返回一个移动的路径
short      Goto(Position p, MovePath& path);
```

以上两个是用来让搬运工移动的，它们返回人走过的步数，失败则返回-1。其中 Goto 用到了 MovePath 结构。

```
typedef    vector<Direction>      MovePath;
```

关于 Goto 算法，即最短路径算法可以参考《CSDN 开发高手》2004 年第 10 期的《PC 游戏中的路径搜索算法讨论》。考虑到推箱子的地图规模不大所以我采用了经典的 Dijkstra 算法。这在数据结构的教科书中应该可以找到，故不多着笔墨。

(2)

为了记忆已经搜索过的状态，BoxRoom 还提供了记录和保存状态的函数：

```
void      SaveState(BoxRoomState& s) const;
void      LoadState(const BoxRoomState& s);
```

其中的 BoxRoomState 结构将在后文中讨论。

(3)

用来检测是否已经胜利。

```
bool      IsFinished() const{
    return m_nbox == std::count(m_map.begin(), m_map.end(), EM BOX TARGET);
}
```

3. 自动求解算法的框架

人工智能的精髓从某种意义上就是穷举，但是如何有效的穷举就是一个好的智能算法所要解决的问题。已知的事实是推箱子问题是 NP-Hard 的。第一个问题是，我们所要搜索的空间是相当的巨大，“傻傻”的搜索是相当费时的，我们所要做的就是动用各种手段减少不必要的搜索来节省时间。还有一个问题是这么大的搜索空间，我们如何利用有限的空间有效的保存，并且快速的判断出某个状态已经搜索过。

上面多次提到了搜索空间，那么我们如何来描述推箱子问题的搜索空间呢。

上面提到 BoxRoom 类的 SaveState 和 LoadState 函数用到了 BoxRoomState。它描述的就是问题空间中的节点。首先，它的实现要尽量节省空间，因为自动求解过程中要记录相当数量的状态。我用了 boost::dynamic_bitset，因为标准库的 bitset 的有一个弱点就是不能动态的决定其位数，而我们又不想让 BoxRoom 模板化。

```
class BoxRoomState{
    friend class BoxRoom;
    boost::dynamic_bitset<> m_extracted_map;
    Position m_manpos;
    short m_totlestep;
    //比较状态是否等价
    //等价：如果状态 A 中能够在箱子保持不动的情况下达到状态状态 B，那么 A<=>B
    //性质：自反性，传递性，对称性
    //
    //注意：其充要条件比较难表示，所以我们暂时只能用其充分条件！所以严格的说这里不符合==的定义，也就是说！operator == ()不代表!=
public:
    bool operator==(const BoxRoomState& oth) const{
        return m_manpos == oth.m_manpos && m_extracted_map == oth.m_extracted_map;
    }
    inline int GetTotlestep() const{return m_totlestep;}
    inline void SetTotlestep(int s){m_totlestep = s;}
};
```

SetTotlestep 似乎有些奇怪（你甚至可以把它改为 1 而不考虑其合理性），提供它纯粹是为了算法的需要。注释已经说明了如何判断两个状态是否等价，特别提到了这只是充分条件而非必要条件。提供一个加强的充分条件（如果是充要条件那将更加完美）将能够进一步减小搜索的空间。

这些状态之间的转移就是边，这样就构成了一个有向图。对一般的有向图的搜索是十分麻烦的，因为这样容易造成回路。考虑这样一种情况，把一个箱子向左推一格和向右推一格再向左推两格达到的状态明显是等价的，对后一种情况继续搜索所需要的步数明显大于前者，所以这一支可以去掉。也就是说，我们只保留状态 A->状态 B 所需要的路径中人走过的步数最少的一个（我的算法只解决最优移动，当然也有很多人需要最优化推动）。如此一来，我们就得到了更特殊的有向图--树。

对树的搜索，大家应该相当熟悉。一般可以分为深度优先搜索和广度优先搜索。由于要得到（步数）最优解，我采用的算法的基本思路属于广度优先搜索。

算法的框架：

```
//表示解中的一次有效移动：表示走到一个箱子旁，并推动他
struct ValidStep{
    Position p;
    Direction d;
    ValidStep():p(-1),d(EAST){}
    ValidStep(int pp, Direction dd):p(pp),d(dd){}
};

typedef vector<ValidStep> SolveResult;

int SolveBoxRoom(BoxRoom room, SolveResult& path){
    //保存根状态
    SolveState startstate(room);
```

```
SolveSearchTree searchtree(startstate);
SolveState 包含 BoxRoomState, 它在 SolveSearchTree 中保存, 这在后面将作讨论。
    //步数的限制, 每次递增, 这样保证得到解的是步数最优解
    int      limit= room.GetTotleststep();
    bool     no_solution;
    do{
        SolveState curstate = startstate;
        int       curdepth = -1;
        //保存每一层已经搜索到的节点的 index
        vector<int> indexlist(1,0);
        no_solution = true;
        limit++;
        do{
            ++curdepth;
            //一开始初状态还没有展开
            if(curdepth != 0){
                //第一次搜索到这一层, 让 indexlist[curdepth] = -1
                if((int)indexlist.size() <= curdepth) indexlist.push_back(-1);
                searchtree.getnextchild(curdepth
indexlist[curdepth-1],indexlist[curdepth],curstate);
                //这一层已经无法得到可用的节点了
                if(indexlist[curdepth] == -1){
                    //已经到头了
                    if(curdepth <= 1)break;
                    //什么? 什么都没做? 废了这一支
                    if(no_solution)
                        searchtree.set_disabled(curdepth - 1,indexlist[curdepth - 1]);
                    //没有到头, 向上回溯
                    curdepth-=2;continue;
                }
            }
            //已经超过深度的限制, 换同一深度的其他节点
            if(limit < curstate.roomstate.GetTotleststep()){
                no solution = false;
                --curdepth;continue;
            }
            room.LoadState(curstate.roomstate);
            if(curstate.isfinished){
                SolveResult result;
                for(int i = curdepth; i > 0; i = curstate.depth){
                    result.push_back(curstate.laststep);
                    searchtree.getfather(curstate.depth,curstate.depthindex,curstate);
                }
                path.insert(path.end(),result.rbegin(),result.rend());
                return room.GetTotleststep();
            }
            //展开一个节点, 如果还没有展开过
            if( searchtree.have not been expanded(curdepth,indexlist[curdepth])){
                //展开这个节点
                BoxRoom::BoxRoomState tmpstate;
                room.SaveState(tmpstate);
                for(Position i = 0; i < room.GetSize(); ++i){
                    if(room.IsNotBox(i))continue;
                    //表示四个方向
                    for(int j = 0; j < 4; ++j){
                        Position nman = i - room.GetOffset(static_cast<Direction>(j));
                        if(room.Goto(nman) != -1){
                            if((room.MovePush(static cast<Direction>(j)) != -1)
                                && IsBoxRoomDead(room)){
                                SolveState ss(room);
                                ss.laststep = ValidStep(nman,static_cast<Direction>(j));
                            }
                        }
                    }
                }
            }
        }
    }
    //注意 IsBoxRoomDead, 事实证明这个函数的好坏能够大的影响我们的搜索范围从而影响我们求解的速度。
    if((room.MovePush(static cast<Direction>(j)) != -1)
        && IsBoxRoomDead(room)){
        SolveState ss(room);
        ss.laststep = ValidStep(nman,static_cast<Direction>(j));
    }
}
```

```

        searchtree.insert(curdepth, indexlist[curdepth], ss);
    }
    room.LoadState(tmpstate);
}
}
searchtree.set_expanded(curdepth, indexlist[curdepth]);
no_solution = false;
}
}while(true);
}while(!no_solution);
//求解失败
return -1;
}

```

4. 状态树和 Hash 表

注意到上面的代码中：

```
SolveState startstate(room);
SolveSearchTree searchtree(startstate);
```

我用了 `SolveState`，`SolveSearchTree` 两个类来保存和组织状态。

上面提到，我们的算法要判断那些状态已经出现过，那么如何高效的搜索就是一个问题了。状态直接在树中保存的话，那么搜索起来将耗费大量的时间。我们先看一下 `SolveState` 的定义：

```

struct SolveState{
    BoxRoom::BoxRoomState roomstate;
    int hash;
    int depth;
    int depthindex;
    bool isfinished;
    ValidStep laststep;
    SolveState(const BoxRoom& room);
    bool operator == (const SolveState& oth) const{
        return hash == oth.hash && roomstate == oth.roomstate;
    }
    inline int GetTotlestep() const{ return roomstate.GetTotlestep(); }
    inline void SetTotlestep(int s){ roomstate.SetTotlestep(s); }
};

```

它只是 `BoxRoom::BoxRoomState` 类型的一个包装。`roomstate` 保存了状态值；为了从节点映射回树，`depth`、`depthindex` 保存了状态在树中的必要信息；`isfinished` 为了避免多次调用 `BoxRoom::IsFinished()`；`laststep` 表示从父状态到该状态，搬运工应该如何移动。还有一个成员 `hash`，一看名字就知道，它和 hash 表有关，是的，它是这个状态的 hash 值，也就是说我们将节点的存储和节电间的关系的表示分开来实现了。来看 `SolveSearchTree` 你就会明白了：

```

class SolveSearchTree{
    class StateLib{
        typedef vector<SolveState> HashNode;
        vector<HashNode> m_hash_table;
    public:
        StateLib(int rank):m_hash_table(HASH_SIZE(rank)) {}
        int Add_state(const SolveState& ns);
        SolveState& Get_state(int stateindex);
    }statelib;

    struct Node{
        int stateindex;//状态在 StateLib 中的索引值
    };
};

```

```

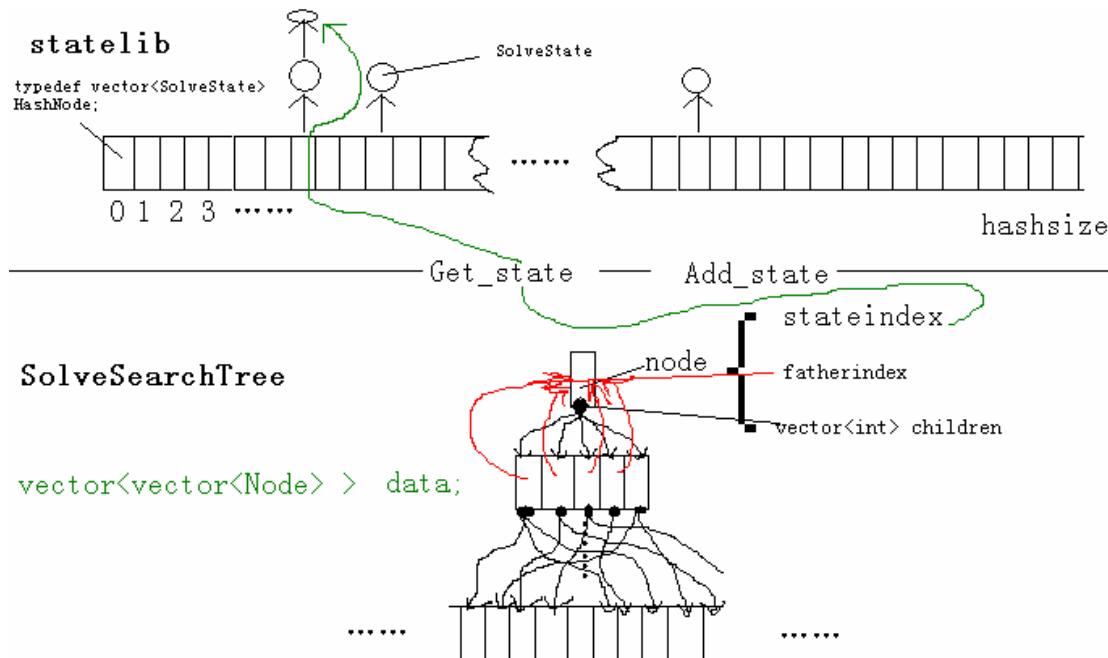
vector<int> children; //所有孩子在下一层数据中的 index 的节点表
int fatherindex;
bool is_expanded;
bool is_disabled;
Node():is_expanded(false),is_disabled(false){}
};

//类似于广义表的方式作为树的表示方式。data[n]代表树的第 n 层，data[n][m]代表第 n 层的第 m 个成员
vector<vector<Node>> data;
SolveState dummystate;

public:
    SolveSearchTree(SolveState& r);
    void insert(int fatherdepth,int fatherindex,SolveState& );
    void getnextchild(int fatherdepth,int fatherindex,int& lastchildindex,
SolveState& );
    void getfather(int childepth,int childindex,SolveState& );
    bool have_not_been_expanded(int depth,int index) const{return !(data[depth][index].is_expanded);}
    void set_expanded(int depth,int index){data[depth][index].is_expanded = true;}
    bool have_not_been_disabled(int depth,int index) const{return !(data[depth][index].is_disabled);}
    void set_disabled(int depth,int index);
};

```

如图所示：



通过用 stateindex 调用 Get_state 我们可以得到唯一一个 SolveState，通过 Add_state 加入新的状态，这时 hash 表的威力就显示出来了：

```

#define HASH_RANK 16
#define HASH_SIZE(rank) (1 << rank)
//对一个值求模使它小于 HASH_SIZE
#define HASH_MOD(hash) (hash & ( (1 << HASH_RANK) - 1))
.....
int Add state(const SolveState& ns){
    HashNode& data = m_hash_table[ns.hash];
    HashNode::iterator iter = find(data.begin(),data.end(),ns);
    long h1;
    if( iter == data.end() ){

```

```

        data.push_back(ns);
        h1 = (long)data.size() - 1;
        return (h1 << HASH_RANK) + ns.hash;
    }else{
        if(ns.GetTotleststep() < (*iter).GetTotleststep()){
            h1 = (long)(iter - data.begin());
            (*iter).SetTotleststep(ns.GetTotleststep());
            (*iter).laststep = ns.laststep;
            return -(h1 << HASH_RANK) + ns.hash;
        }
        //Magic Number, 表示无法加入这个新状态, 因为已经存在步数更优的等价状态
        //因为 hash!=0, 所以说下面的(h1<< HASH_RANK) +ns.GetHash() 肯定不会等于 0
        return 0;
    }
}
SolveState& Get_state(int stateindex){
    HashNode& data = m_hash_table[HASH_MOD(stateindex)];
    return data[stateindex >> HASH_RANK];
}

```

stateindex 用位操作来提高速度, 它的思想不难理解。通过 hash 表我们可以大大减少搜索状态的时间, 那么 hash 值又是什么呢? 我选择了一个相当简单的方法:

```

SolveState::SolveState(const BoxRoom& room):hash(0){
    room.SaveState(roomstate);
    isfinished = room.IsFinished();
    //求 hash 值
    for(int i = 0;i < room.GetSize(); ++i){
        if(room.IsBox(i)){
            hash += i*(i+1)*(i+2);
            hash = HASH_MOD(hash);
        }
    }
}

```

呵呵, 简单吧, 肯定有更好的 hash 值, 但这里我偷个懒罢了。

树的 insert 操作要负责对等价节点的处理, 保证等价节点只保留一个步数最优的:

```

void SolveSearchTree::insert(int fatherdepth,int fatherindex,SolveState& ss){
    int newchildstateindex = statelib.Add_state(ss);
    //这个状态已经存在, 而且以前的步数更优
    if(newchildstateindex == 0) return;
    //这个状态不是新状态, 但它比以前的步数更优
    if(newchildstateindex < 0){
        newchildstateindex = -newchildstateindex;
        SolveState& ts = statelib.Get_state(newchildstateindex);
        set_disabled(ts.depth,ts.depthindex);
    }
    if((int)data.size() <= fatherdepth + 1) data.push_back(vector<Node>());
    Node childnode;
    childnode.stateindex = newchildstateindex;
    childnode.fatherindex = fatherindex;
    data[fatherdepth+1].push_back(childnode);
    int newchilddepthindex = (int)data[fatherdepth+1].size() - 1;

    SolveState& ts = statelib.Get_state(newchildstateindex);
    ts.depth      = fatherdepth + 1;
    ts.depthindex = newchilddepthindex;
    data[fatherdepth][fatherindex].children.push_back(newchilddepthindex);
}

```

树的 getnextchild 操作要跳过已经废除的枝:

```

void SolveSearchTree::getnextchild(int fatherdepth, int fatherindex, int& lastchildindex,
SolveState& rt){

```

```
vector<int>& childindex = data[fatherdepth][fatherindex].children;
vector<int>::iterator iter;
if(lastchildindex == -1){
    iter = childindex.begin();
} else{
    iter = find(childindex.begin(), childindex.end(), lastchildindex) + 1;
}
do{
    if(iter == childindex.end()){
        lastchildindex = -1;
        rt = dummysstate; return;
    } else{
        lastchildindex = *iter;
        if(data[fatherdepth+1][lastchildindex].is_disabled){
            ++iter;
            continue;
        }
        rt = statelib.Get_state(data[fatherdepth+1][lastchildindex].stateindex); return;
    }
} while(true);
}
```

5. 死锁检测

万事俱备，只欠东风。我们还差一个 IsBoxRoomDead 函数。死锁就是一旦把箱子推动到某些位置，一些箱子就再也无法推动或者无法推到目的点，比如四个箱子成 2×2 摆放。推箱子高手对何种情况引起死锁非常敏感，这样他们预先就知道决不能让某些局面形成，这也是高手高于常人的原因之一。

当然我不是推箱子的高手，所以我只给出了两个简单的判断规则：

规则一：

```
#B # # B# ## B# BB BB BB  
# B# #B # BB #B ### BB BB
```

其中 B 表示箱子，# 表示墙。如果出现了上面的任何一种情况，那么将一定死锁

规则二：

边缘上的箱子的个数大于边缘上的目标的数量。比如如下的情况：

```
#####  
# T T B B B #
```

T 表示目标。

可能要令你失望的是，我的程序只解决了这两种显而易见的死锁情形的判断，V_V!。网上葛永高人 (<http://notabdc.vip.sina.com>) 有一个推箱子自动求解的程序，它的程序有相当先进的死锁检测算法，但可惜的是没有给出源代码。所以这一部分只能我也就不能再详细展开了。

6. 结语

这个程序目前还不是很完备，我的实验证明，它的复杂度和箱子的个数有很大的关系，当箱子很多的时候还不能很好的解出。这篇文章的目的只是给出一个算法的框架，使它能够解决一些问题了，全当抛砖引玉。

【责任编辑：xiong@hitbbs】

<http://emag.csdn.net>

<http://purec.binghua.com>

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	
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	
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	

purec@126.com

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稿，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酬, 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