(京)新登字 158 号

内 容 提 要

本书是一本全面详尽介绍 Windows NT 并以该系统为主要范例的教科书。

本书共分六部分(14 章),第一部分介绍了有关操作系统的基本概念,以及操作系统运行的基本的软硬件环境。第二部分探讨了多道操作系统极为重要的基础——并行程序设计。第三部分讨论了作业和进程的调度以及死锁问题。第四部分中分别讲述各种实用的实存储器和虚拟存储技术及其最新发展。第五部分介绍了设备和文件管理中的有关问题。第六部分探讨了操作系统的结构,并介绍了 Windows NT, UNIX, CP/M, MP/M 等操作系统。本书作为计算机专业教材,内容丰富,通俗易懂,便于自学。可作为大、专院校计算机专业的教科书和参考书,也可作为电视大学的教材。

图书在版编目(CIP)数据

操作系统基础/屠立德,屠祁编著 .- 2 版 .- 北京:清华大学出版社,1995
ISBN 7-302-01914-2

Ⅰ.操… Ⅱ.①屠… ②屠… Ⅲ.操作系统-基本知识 Ⅳ.TP316

中国版本图书馆 CIP 数据核字 (95) 第 10941 号

出版者:清华大学出版社(北京清华大学校内,邮编 100084)
印刷者:北京市丰华印刷厂
发行者:新华书店总店北京科技发行所
开 本:787×1092 1/16 印张:17.5 字数:409 千字
版 次:1995 年 9 月第 2 版 1996 年 3 月第 2 次印刷
书 号:ISBN 7-302-01914-2/TP·877
印 数:12001—22000
定 价:16.00 元

前　　言

承蒙广大读者的关怀和支持,本书出版六年来已拥有十余万读者。他(她)们在使用本书过程中积累了丰富的经验和体会,为本书的改进和提高作出了极大的贡献。在此我们向广大的读者表示深深的敬意和感激之情。

六年来计算机技术获得很大发展。32 位的 386, 486 已广泛使用,网络和分布式处理已成为当前的发展潮流。于是在计算机世界,一个举世瞩目、崭新的 90 年代操作系统——Windows NT, 于 1993 年推出了。它采用了操作系统最新技术,提供了现代操作系统几乎所有功能: 多任务/ 多线程能力, 对称式多处理环境,内置式网络功能, Windows 图形用户界面……。

为反映操作系统当代最新发展技术和广大读者的经验与体会,我们对本书作了如下重大修订:

(1) 对 Windows NT 操作系统作了全面详尽的介绍,并作为本书主要范例。本书是第一本全面详尽介绍 Windows NT 的操作系统教科书。

(2) 每章后增加了大量习题,以帮助读者检验和加深对内容的理解。

(3) 结合作者长期从事软件工程教学和科研以及在第十一届亚运会计算机工程和其它大型软件开发实践中的体会,重写了第 11 章。介绍了在操作系统设计和结构方面的新成就和新进展。

(4) 其它各章都作了不同程度的修订,以反映该领域内的最新技术和方法。

本书是作者根据自己在多所院校的本科生和研究生中讲授操作系统课程以及科研中的经验,参考了国内、外操作系统方面的教材和操作系统技术的新发展,并考虑当代大学生的学习特点而编写的。在教材中力求使内容由浅入深,文字通俗易懂,便于讲授和自学。这是本书特点之一。其次在内容上,按理工科院校计算机专业的教学大纲要求进行编写。对于非本科的计算机专业的教学,目录中记有星号的部分可以删去不讲。本书在内容上不同于一般操作系统教科书的是给微型计算机以充分的注意,这是本书特点之二。

全书结构上符合目前国内一般讲授的体系。同时更能适应当前普遍要求减少课内讲授时数,培养和提高学生自学能力的发展趋势和要求。本书中各算法所采用的描述语言是类PASCAL 语言。在学习本课程前,读者应具备计算机原理、程序设计和数据结构等方面的知识。

本书第 3, 4, 7, 8, 10 和第 12 章由屠祁编写,其余部分由屠立德编写。由于时间仓促以及作者水平所限,错误和不妥之处在所难免,恳请读者批评指正。

作者于北京工业大学计算机系

目　　录

第一部分　概　　论

第 1 章　引论 …… 1
1.1　系统概述 …… 1
1.1.1　计算机的硬件组织 …… 1
1.1.1.1　微型计算机的典型组织 …… 1
1.1.1.2　大-小型计算机的硬件组织 …… 2
1.1.2　软件的层次与虚拟机的概念 …… 3
1.2　操作系统的形成和发展 …… 4
1.2.1　什么是操作系统 …… 4
1.2.2　操作系统的形成和发展 …… 4
1.3　多道程序设计的概念 …… 5
1.3.1　多道程序设计的引入 …… 5
1.3.2　多道程序设计的概念 …… 6
1.4　操作系统的功能和特性 …… 7
1.4.1　操作系统的功能 …… 7
1.4.2　操作系统的特性 …… 7
1.5　操作系统的类型 …… 8
1.5.1　多道批处理操作系统 …… 8
1.5.2　分时系统 …… 9
1.5.3　实时系统 …… 10
1.5.4　网络操作系统 …… 10
1.5.4.1　网络操作系统概述 …… 10
1.5.4.2　Windows NT 的内装网络简介 …… 11
1.6　微型计算机及其操作系统的发展趋势 …… 12
习题 …… 14
第 2 章　操作系统的运行环境 …… 16
2.1　硬件环境 …… 16
2.1.1　中央处理机(CPU) …… 16
2.1.1.1　特权指令 …… 16
2.1.1.2　处理机的状态 …… 17
2.1.1.3　程序状态字 PSW …… 17
2.1.2　主存储器 …… 18
2.1.2.1　存储器的类型 …… 18
2.1.2.2　存储分块 …… 19
2.1.2.3　存储保护 …… 19

2.1.3 缓冲技术 …… 20
2.1.4 中断技术 …… 21
2.1.4.1 中断的概念 …… 21
2.1.4.2 中断逻辑与中断寄存器 …… 21
2.1.4.3 中断类型 …… 22
2.1.4.4 中断响应与中断屏蔽 …… 23
2.1.4.5 中断处理 …… 24
2.1.5 时钟、时钟队列 …… 25
2.2 操作系统与其它系统软件的关系 …… 26
2.2.1 作业、作业步和进程的关系 …… 26
2.2.2 重定位的概念 …… 27
2.2.2.1 绝对地址、相对地址和逻辑地址空间 …… 27
2.2.2.2 静态重定位 …… 27
2.2.3 绝对装入程序和相对装入程序 …… 28
2.2.3.1 绝对装入程序 …… 29
2.2.3.2 相对装入程序——连接装入程序 …… 29
2.3 操作系统与人的接口 …… 30
2.3.1 作业控制语言 …… 31
2.3.2 联机作业控制——终端命令和图形用户接口(GUI) …… 32
2.3.2.1 终端命令 …… 32
2.3.2.2 图形用户接口(GUI) …… 33
* 2.4 固件——微程序设计概念 …… 34
2.4.1 微程序设计的概念 …… 34
2.4.2 微程序设计和操作系统 …… 35
习题 …… 35

第二部分 多道程序设计基础——并行程序设计

第 3 章 进程管理 …… 37
3.1 进程的概念 …… 37
3.1.1 进程的引入 …… 37
3.1.2 进程的定义 …… 38
3.2 进程的状态和进程控制块 …… 39
3.2.1 进程的状态及其变化 …… 39
3.2.2 进程控制块 …… 39
3.3 进程队列 …… 41
3.4 进程的管理 …… 42
3.4.1 进程的挂起和解除挂起 …… 42
3.4.2 进程的控制原语 …… 43

3.4.2.1 建立进程原语 …… 43
3.4.2.2 挂起进程原语 …… 44
3.4.2.3 解除挂起原语 …… 45
3.4.2.4 撤消进程原语 …… 45
3.4.2.5 改变进程优先数原语 …… 46
3.5 Windows NT 中的线程 …… 47
习题 …… 48
第 4 章 多道程序设计基础—— 并行程序设计 …… 50
4.1 顺序程序设计和并行程序设计概念 …… 50
4.1.1 顺序程序设计的特点 …… 50
4.1.2 并行程序设计 …… 51
4.1.2.1 并行程序设计的概念 …… 51
4.1.2.2 程序并行性的表示 …… 52
4.1.2.3 并行程序设计的特点 …… 53
4.2 进程间的同步与互斥 …… 54
4.2.1 临界段问题 …… 54
4.2.1.1 问题的提出 …… 54
* 4.2.1.2 软件解决办法 …… 55
4.2.2 同步与互斥的执行工具 …… 59
4.2.2.1 硬件指令 …… 59
4.2.2.2 信号量 …… 61
4.2.2.3 P, V 操作 …… 61
4.3 同步机构应用 …… 63
4.3.1 用信号量实现进程间的互斥 …… 63
4.3.2 信号量作为进程的阻塞和唤醒机构 …… 64
4.3.3 生产者和消费者问题 …… 65
4.3.4 阅读者/ 写入者问题 …… 66
4.4 进程间的通信 …… 67
* 4.5 管程的概念 …… 68
4.5.1 管程的定义 …… 69
4.5.2 五位就餐的哲学家问题 …… 69
4.6 Windows NT 中的同步与互斥机制 …… 71
习题 …… 72

第三部分 处理机管理

第 5 章 作业和进程的调度 …… 75
5.1 调度的层次和作业状态转换 …… 75
5.1.1 调度的层次 …… 75

5.1.2 作业状态 …… 75
5.2 作业的调度 …… 76
5.2.1 后备作业队列及作业控制块 JCB …… 76
5.2.2 作业调度及其功能 …… 77
5.3 进程调度 …… 77
5.4 选择调度算法时应考虑的问题 …… 78
5.5 调度算法 …… 79
5.5.1 先来先服务调度算法 FIFO …… 79
5.5.2 优先级调度算法 …… 79
5.5.3 时间片轮转算法 …… 79
5.5.4 短作业优先调度算法 …… 80
5.5.5 最短剩余时间优先调度算法 …… 81
5.5.6 最高响应比优先调度算法 …… 81
5.5.7 多级反馈队列调度算法 …… 81
5.6 Windows NT 可抢占动态优先级多级就绪队列调度算法 …… 83
习题 …… 84
第 6 章 死锁 …… 86
6.1 死锁问题的提出 …… 86
6.2 死锁的必要条件 …… 87
6.2.1 资源的概念 …… 87
6.2.2 死锁的必要条件 …… 88
6.3 死锁的预防 …… 89
6.3.1 预先静态分配法 …… 89
6.3.2 有序资源使用法 …… 89
* 6.4 死锁的避免和银行家算法 …… 90
6.4.1 银行家算法问题 …… 90
6.4.2 银行家算法 …… 91
6.4.3 银行家算法的优缺点 …… 93
* 6.5 死锁检测 …… 94
6.5.1 资源分配图 …… 94
6.5.2 资源分配图的化简 …… 94
6.5.3 资源分配图化简的实现 …… 95
6.5.3.1 矩阵表示法 …… 96
6.5.3.2 链表表示法 …… 97
6.5.3.3 检测算法的执行速度 …… 97
6.6 死锁的恢复 …… 98
习题 …… 99

第四部分　主存储器管理

第 7 章　实存储器管理技术 …… 101
7.1　引言 …… 101
7.1.1　主存储器的物理组织、多级存储器 …… 101
7.1.2　主存储器管理中的研究课题 …… 102
7.2　固定分区 …… 102
7.2.1　数据库 …… 103
7.2.2　存储分配算法 …… 104
7.2.3　存储保护与重定位 …… 104
7.2.4　优缺点 …… 105
7.3　可变分区的多道管理技术 …… 105
7.3.1　数据库 …… 106
7.3.2　分配和释放算法 …… 107
7.3.3　存储器的紧缩和程序的浮动 …… 108
7.3.3.1　碎片问题和存储器的紧缩 …… 108
7.3.3.2　程序浮动 …… 109
7.3.4　动态重定位的可变分区多道管理 …… 110
7.3.4.1　动态重定位 …… 110
7.3.4.2　动态重定位的硬件支持、软件算法 …… 111
7.3.4.3　IBM-PC 等微型计算机的存储管理与地址变换机构 …… 111
7.3.5　优缺点 …… 113
7.4　多重分区(多对界地址)管理 …… 113
7.5　覆盖技术 …… 113
7.5.1　覆盖的概念 …… 113
7.5.2　覆盖处理 …… 115
7.6　交换技术 …… 115
习题 …… 116
第 8 章　虚拟存储管理 …… 117
8.1　虚拟存储系统的基本概念 …… 117
8.2　分页存储管理 …… 119
8.2.1　分页存储管理的基本概念 …… 119
8.2.2　分页系统中的地址转换 …… 121
8.2.2.1　直接映象的页地址转换 …… 121
8.2.2.2　相关映象页地址转换 …… 122
8.2.2.3　相关映象和直接映象结合的页地址转换 …… 123
8.2.3　分页存储管理策略 …… 124
8.2.4　分页存储管理的软硬件关系和软件算法 …… 124

8.2.4.1 数据库 …… 124
8.2.4.2 分页存储管理中软硬件关系和缺页中断处理算法 …… 125
8.2.4.3 页表表目的扩充 …… 126
8.2.5 页的共享 …… 127
8.3 分段存储管理 …… 127
8.3.1 分段存储管理的基本概念 …… 127
8.3.2 分段管理中的地址转换 …… 128
8.3.3 段的动态连接 …… 129
8.3.3.1 连接间接字和连接中断 …… 130
8.3.3.2 编译程序的连接准备工作 …… 131
8.3.3.3 连接中断处理 …… 131
8.3.3.4 纯段和杂段(连接段) …… 131
8.3.4 虚拟存储管理中的存储保护问题 …… 132
8.3.5 分段存储管理的优缺点 …… 132
8.4 段页式存储管理 …… 133
8.4.1 段页式存储管理的基本概念 …… 133
8.4.2 段页式存储管理中的地址转换 …… 135
8.4.3 段页式存储管理算法 …… 136
8.4.4 段页式存储管理的优缺点 …… 136
8.5 页(和段)的置换算法和系统行为 …… 138
8.5.1 最佳置换算法 OPT …… 138
8.5.2 先进先出置换算法 FIFO …… 138
8.5.3 最近最少使用置换算法 LRU …… 139
8.5.4 最近未使用置换算法 NUR …… 139
8.5.5 分页环境中程序的行为特性 …… 140
8.5.5.1 局部性的概念 …… 140
8.5.5.2 分页环境中程序的行为特性 …… 141
8.5.5.3 减少访问离散性的程序结构 …… 141
8.5.6 工作集 …… 142
8.6 页架的分配算法 …… 143
8.6.1 提前分配 …… 143
8.6.2 最少页架数 …… 144
8.6.3 局部和全局分配 …… 144
8.6.4 分配算法 …… 144
8.6.5 页的大小 …… 145
* 8.7 高速缓冲存储器 …… 146
8.7.1 高速缓冲存储器的组织 …… 146
8.7.2 缓存块的编址形式 …… 147

8.7.3 缓存的工作过程 …… 148
8.8 Windows NT 的分页机构、页面调度算法和工作集、共享主存机制 …… 149
8.8.1 Windows NT 的二级页表地址变换机构 …… 149
8.8.2 页面调度算法和工作集 …… 149
8.8.3 共享主存机制——段对象、视口和映象文件 …… 150
习题 …… 150

第五部分 设备和文件管理

第 9 章 设备管理 …… 152
9.1 输入输出组织和输入输出处理机 …… 152
9.1.1 输入输出接口(IO 接口) …… 153
9.1.2 输入输出处理机(通道) …… 153
9.2 辅助存储器 …… 154
9.2.1 磁带的硬件特性及信息的组织 …… 154
9.2.2 磁鼓的硬件特性及信息的组织 …… 156
9.2.3 磁盘的硬件特性及信息的组织 …… 157
9.3 设备管理概述 …… 158
9.3.1 设备绝对号、相对号、类型号与符号名 …… 158
9.3.2 设备管理的任务 …… 159
9.4 设备分配策略 …… 160
9.4.1 设备控制块和设备等待队列 …… 160
9.4.2 独享设备的分配 …… 161
9.4.3 虚拟设备和 SPOOL 系统 …… 161
9.4.4 共享设备的分配和磁盘调度策略 …… 163
9.4.4.1 移动头磁盘存储器的操作 …… 163
9.4.4.2 查找优化的各种策略 …… 164
9.4.4.3 旋转优化 …… 165
9.5 输入输出管理程序 …… 165
9.5.1 输入输出进程 …… 166
9.5.2 设备管理程序 …… 166
9.5.3 输入输出调度程序 …… 167
9.6 Windows NT 一体化的输入输出系统 …… 167
习题 …… 168
第 10 章 文件系统 …… 169
10.1 文件系统概述 …… 169
10.1.1 引言 …… 169
10.1.2 文件的分类 …… 170
10.1.3 文件系统的功能和基本操作 …… 171

10.2 文件的逻辑组织和物理组织 …… 172
10.2.1 文件的逻辑组织 …… 172
10.2.2 文件的物理组织 …… 172
10.3 文件目录 …… 175
10.3.1 文件目录和文件描述符 …… 175
10.3.2 一级目录结构 …… 176
10.3.3 二级目录结构 …… 177
10.3.4 多级目录结构 …… 177
10.3.5 目录组织的改进——符号文件目录和基本文件目录 …… 179
10.4 辅存空间的分配和释放 …… 181
10.4.1 辅存空闲块的管理 …… 181
10.4.2 辅存空间的分配和释放 …… 182
10.5 文件的共享与文件系统的安全性 …… 184
10.5.1 文件的连接 …… 184
10.5.2 文件的存取控制 …… 185
10.5.3 文件的转储和恢复 …… 187
10.6 文件的使用与控制 …… 187
10.6.1 活动文件表和活动符号名表 …… 187
10.6.2 建立文件命令 …… 188
10.6.3 打开文件命令 …… 189
10.6.4 读文件命令 …… 189
10.6.5 写文件命令 …… 190
10.6.6 关闭文件命令 …… 190
10.6.7 撤消文件命令 …… 190
10.7 Windows NT 的多重文件系统 …… 190
习题 …… 191

第六部分 操作系统结构与范例

第 11 章 操作系统的结构和设计 …… 192
11.1 操作系统的设计 …… 192
11.1.1 设计的目标和原则 …… 192
11.1.2 操作系统的设计 …… 194
11.1.2.1 系统分析和总体功能设计阶段 …… 194
11.1.2.2 系统设计与结构设计阶段 …… 195
11.2 操作系统的结构 …… 195
11.2.1 模块接口法(单块式) …… 195
11.2.2 层次结构设计法 …… 197
11.2.3 客户/服务器方式 …… 198

习题 …… 199
第 12 章 Windows NT 操作系统 …… 200
12.1 Windows NT 操作系统概述 …… 200
12.2 Windows NT 的设计目标 …… 200
12.3 Windows NT 的系统模型 …… 201
12.4 Windows NT 的结构 …… 203
12.5 Windows NT 的基元成分——对象、进程和线程 …… 205
12.5.1 对象 …… 205
12.5.2 进程 …… 207
12.5.3 线程 …… 209
12.5.4 对象、进程和线程之间的关系 …… 211
12.5.5 进程管理程序 …… 212
12.6 内核 …… 213
12.6.1 内核调度程序与线程的状态转换 …… 213
12.6.2 中断和异常处理 …… 215
12.6.3 内核的同步与互斥机制——多处理器间的同步 …… 216
12.7 虚拟存储管理 …… 217
12.7.1 进程的虚拟地址空间 …… 217
12.7.2 NT 虚拟分页的地址变换机构 …… 218
12.7.3 页面调度策略和工作集 …… 219
12.7.3.1 页面调度策略 …… 219
12.7.3.2 工作集 …… 220
12.7.4 页架状态和页架数据库 …… 220
12.7.5 共享主存——段对象、视口和映象文件 …… 221
12.8 输入输出(I/O)系统 …… 222
12.8.1 输入输出(I/O)系统的结构 …… 222
12.8.2 统一的驱动程序模型 …… 223
12.8.3 异步 I/O 操作和 I/O 请求处理过程 …… 224
12.8.4 映象文件 I/O …… 225
12.9 Windows NT 的内装网络 …… 225
12.9.1 Windows NT 的内装网络的特色 …… 225
12.9.2 Windows NT 网络的体系结构 …… 226
12.10 对象管理程序 …… 227
12.11 进程通信及本地过程调用(LPC) …… 228
12.11.1 线程间的同步 …… 228
12.11.2 进程通信——本地过程调用(LPC) …… 229
12.12 Windows NT 的安全性 …… 230
12.13 综述 …… 231

第 13 章　UNIX 操作系统 …… 233
13.1　UNIX 操作系统概述 …… 233
13.2　系统结构 …… 234
13.3　进程管理 …… 235
13.3.1　程序状态字和通用寄存器 …… 235
13.3.2　进程和进程控制块 PCB …… 236
13.3.3　进程的控制 …… 241
13.4　文件系统 …… 243
13.4.1　UNIX 文件系统概述 …… 243
13.4.2　文件目录结构和文件(路径)名 …… 244
13.4.3　文件卷的动态装卸和安装 …… 245
13.4.4　文件的共享和联接 …… 245
13.5　设备管理和输入输出系统 …… 245
13.6　管道线 pipe 机构 …… 246
13.7　系统调用 …… 247
13.8　shell 语言简介 …… 248
13.8.1　shell 的一般用法 …… 248
13.8.2　shell 过程的用法 …… 250
第 14 章　CP/M 操作系统 …… 252
14.1　CP/M 操作系统概述 …… 252
14.1.1　概述 …… 252
14.1.2　CP/M 操作系统的功能和特性 …… 252
14.2　CP/M 的结构 …… 253
14.3　主存分配 …… 253
14.4　控制台命令处理程序 CCP …… 254
14.5　基本输入输出系统 BIOS …… 255
14.6　CP/M 文件系统 …… 256
14.6.1　CP/M 的文件组织和文件操作 …… 256
14.6.2　盘空间管理 …… 258
14.6.3　目录管理 …… 259
14.6.4　表块管理 …… 259
14.7　MP/M 操作系统 …… 259
14.7.1　MP/M 的结构 …… 260
14.7.2　主存管理 …… 260
14.7.3　进程的管理 …… 261
14.7.4　进程调度 …… 261
14.7.5　进程的同步与通信 …… 262
14.7.6　SPOOL 系统 …… 262
参考文献 …… 263

第 一 部 分

概　　论

第 1 章　　引　　论

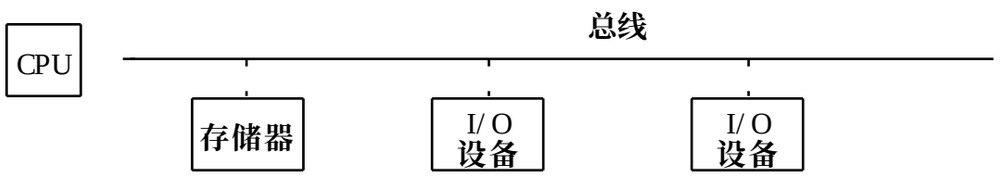
1.1　系 统 概 述

现在的一个完整的计算机系统,不论是大型机、小型机、甚至微型机和微处理机,都由两大部分组成:即计算机的硬件部分和计算机的软件部分。通常计算机的硬件部分是指计算机物理装置本身,它可以是电子的、电的、磁的、机械的、光的元件或装置,或者是由它们组成的计算机部件和计算机本体。总而言之,是指计算机系统中所有那些‘硬的’物理设施。也就是指计算机的各种处理机(如中央处理机,输入输出处理机和包含在该计算机系统中的其他处理机)、存储器、输入输出设备和通信装置。而软件部分是指计算机系统中的所有软件。术语‘软件’是相对于硬件而言的,它是指由计算机硬件执行以完成一定任务的所有程序及其数据。计算机的软件部分通常指各种语言的编译程序和解释程序、汇编程序、装入程序、连接编辑程序、连接装入程序、用户应用程序、数据库管理系统、数据通信系统和操作系统。那么计算机系统的各硬件部分是怎样连接和构成一个完整的计算机,各软件部分又是怎样的关系,硬件和软件之间又是怎样的关系呢?

1.1.1　计算机的硬件组织

1.1.1.1　微型计算机的典型组织

微型计算机也同一般计算机系统一样,由三个主要部分组成:处理机,存储器和输入输出(又称 I/O)设备,其组织结构关系如图 1.1 所示,由图可以看出微型计算机是以总线



为纽带来构成计算机系统。微处理机和存储器,存储器和输入输出设备,以及微处理机和输入输出设备之间都要经过总线来交换信息,无论哪个设备,如果需要使用总线与另一设备交换信息时,就必须先请求总线使用权,在获得总线使用权后才能进行通信。在通信双方使用总线期间,其它设备不能插入总线操作,这是其特点之一。其次,数据流的路线也有其特点,这主要表现在微处理机与输入输出设备交换数据时的两种不同的路线:当微处理机与慢速的输入输出设备(如打印机或终端等设备)交换数据时是不经过存储器的,而是直接从(或向)输入输出设备接口(控制器)中的数据寄存器中读(或写)。当微处理机与高速的输入输出设备(如磁盘)交换数据时,这些输入输出设备在控制器控制下首先将数据(通常是一组数据)送往存储区(或从存储区取数据),也就是说微处理机与高速输入输出设备交换数据时,必须经由存储器。这样两种不同的数据交换路线当然是由微型计算机的组织结构所决定的。

图 1.1 所示的单处理机的系统通常使用在比较简单的环境。目前许多微型计算机系统已具有很多复杂的、功能很强的处理能力,在这种情况下的微型计算机系统是多处理机的操作方式。在 IBM-PC 系统中,把单处理机的系统称为最小组成方式,或简称为最小方式系统,而把多处理机系统称为最大组成方式;或简称为最大方式系统。图 1.2 为最大方式系统的组织。图示系统中有一个 Intel 8088 芯片的 CPU 和一个(或两个)Intel 8089 芯片的输入输出处理机专门负责对输入输出的控制和管理。这两类处理机(CPU 和输入输出处理机)都同样地连在系统总线和输入输出总线(IO 总线)上,共同享用总线(但不准许同时使用总线)。

图 1.2　最大方式系统的组织

1.1.1.2　大-小型计算机的硬件组织

小型到大型的计算机系统多由中央处理机、输入输出处理机(又称通道)、存储器和输入输出设备组成,图 1.3 中是一个典型的中型计算机(IBM 370)的硬件组织。通常这类计算机都是非总线结构。由图中可以看出,存储器成为这类计算机组成中的中心部分。无论

中央处理机,还是诸多的输入输出处理机都与存储器相连,这些处理机执行的程序和数据都存放在存储器中(除 ROM 片中的微程序外),并从存储器中取来指令执行。中央处理机需要从或向输入输出设备(不管是高速设备还是慢速设备)交换数据时,它命令输入输出处理机来负责进行管理和控制。数据传输的路线都需经过存储器、输入输出处理机,也就是说中央处理机不能直接从输入输出设备中取(或存)数据,因为它们之间没有直接的数据线相连。

图 1.3　IBM 370 的系统结构

1.1.2　软件的层次与虚拟机的概念

一个计算机系统除了硬件部分以外还有许多的软件,这些软件通常可分为两大类,即系统软件和应用软件。系统软件用于计算机的管理、维护、控制和运行,以及对运行的程序进行翻译、装入等服务工作。系统软件本身又可分成三部分,即操作系统、语言处理系统和常用的例行服务程序。语言处理系统包括各种语言的编译程序,解释程序和汇编程序。服务程序的种类很多,通常包括库管理程序、连接编辑程序、连接装配程序、诊断排错程序、合并/排序程序以及不同的外部存储媒体间的复制程序等。应用软件是指那些为了某一类的应用需要而设计的程序、或用户为解决某个特定问题而编制的程序或程序系统,如航空订票系统就是一个例子。

计算机系统中的硬件和软件以及软件的各部分之间是怎样一种关系呢,或者说是怎样组织的呢? 如果用一句话来回答,它们的关系是层次结构的关系。计算机的硬件通常称为裸机,一个裸机的功能即使很强,但它往往是不方便于用户使用的,功能上相对来说也是有局限性的。而软件是在硬件基础之上对硬件的性能加以扩充和完善。举例来说,用户想要在裸机上运行他的程序时,他就必须用机器语言来编写程序。用户要在裸机上输入一个数据,他就要自己编写上千条指令的输入输出程序,这显然都使用户感到十分不便。再者如果我们给一个只有定点运算功能的裸机配上浮点运算的软件,则计算机就具有了浮点运算的功能,这就是用软件来扩充和完善硬件功能。至于软件之间的关系也是这样,一

部分软件的运行要以另一部分软件的存在并为其提供一定的运行条件作为基础,而新添加的软件可以看作是在原来那部分软件基础上的扩充与完善。因此一个裸机在每加上去一层软件后,就变成了一个功能更强的机器,我们通常把这“新的更强功能的机器”称之为“虚拟机”。图 1.4 中表示了计算机的硬件(裸机)和各类软件之间的层次关系,由图可以看

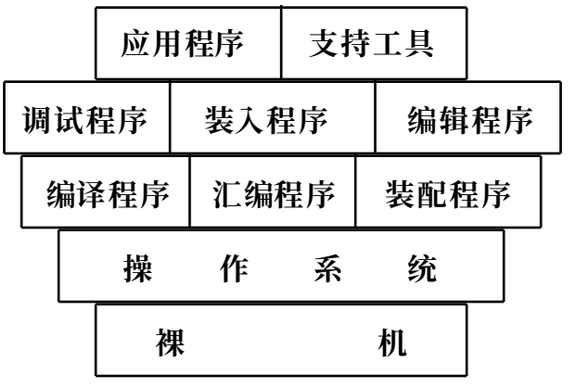


图 1．4　软件的层次

出,操作系统是紧挨着硬件层的第一层软件,它对硬件进行首次扩充。如果是多用户的操作系统,那么经它扩充后,一个实际的处理机就可以扩充成多个虚拟机器,使得每个用户都有一个虚拟计算机。操作系统同时又是其他软件的运行基础。

1.2　操作系统的形成和发展

1.2.1　什么是操作系统

由上可知,操作系统是系统软件中最基本的部分。其主要作用是:

(1) 管理系统资源。这些资源包括中央处理机、主存储器、输入输出设备、数据文件和网络等。

(2) 使用户能共享系统资源,并对资源的使用进行合理调度。

(3) 提供输入输出的便利,简化用户的输入输出工作。

(4) 规定用户的接口,以及发现并处理各种错误的发生。

虽然操作系统存在已几十年了,但至今世界上并没有一个被普遍接受的对操作系统的定义。通常把操作系统定义为“用以控制和管理系统资源,方便用户使用计算机的程序的集合”。

当前,世界上广为流行的观点是把操作系统看成为计算机系统的资源管理者,也就是说操作系统的主要任务是管理并调度计算机系统资源的使用。

1.2.2　操作系统的形成和发展

在 1946 年第一台计算机问世时,并没有操作系统,甚至没有任何软件。当时在使用方式上是单用户独占,即每次只能一个用户使用计算机,一切资源全部由该用户所占有。并且在一个作业运行过程中,以及在作业完成后转换到另一个作业都需要很多人工的干预。到了 50 年代,随着处理机速度的提高,这种手工操作所浪费的机时变得尖锐起来(例如一个作业在每秒一千次的机器上运行,需要三十分钟才能完成。而用手工来装入和卸下作业

等人工干预只需要三分钟。若机器速度提高十倍,则作业所需的运行时间缩短为三分钟,而手工操作时间仍然需要三分钟,这使得一半的机时被浪费了)。人们就想在作业转换过程中排除人工干预,使之自动转换。于是出现了早期的"批处理系统"。基本作法是把若干个作业合成一批,用一台或多台小型的卫星机把这批作业输入到磁带上,然后再把这盘磁带装到主机的磁带机上,由主机的监督程序——最早的操作系统雏型——把磁带上第一个作业调入主存中执行,该作业终止后(正常完成或非正常终止),再依次调入下一个作业……。

到了60年代,硬件技术取得了两个方面的重大进展:一是通道技术的引进,二是中断技术的发展,使得通道具有中断主机工作的能力。这就导致了操作系统进入了多道程序设计系统阶段。所谓通道是专门用来控制输入输出设备的处理机,称之为输入输出处理机(简称IO处理机)。通道比起主机来说,一般速度较慢,价格较便宜。它可以与中央处理机(简称CPU)并行工作。这样当需要传输数据时,CPU只要命令通道去完成就行了。当通道完成传输工作后,用中断机构向CPU报告完成情况,这样就可以把原来由CPU直接控制的输入输出工作转移给了通道,使得宝贵的CPU机时全部用来进行主要的数据处理工作。

但是如果仍然沿用过去的操作方式,主存中只存放一个用户作业在其中运行。那么,在CPU等待通道传输数据过程中,仍然因无工作可作而处于空闲状态。若是我们在主存中同时存放多个作业,那么CPU在等待一个作业传输数据时,就可转去执行主存中的其它作业,从而保证CPU以及系统中的其它设备得到尽可能充分的利用。在主存中同时存放多个作业,使之同时处于运行状态的系统称之为多道程序系统。由于在主存同时存放多个运行作业,就给系统带来了一系列复杂的问题,我们将在以后各章详细阐述之。这时操作系统的功能已变得十分丰富而完整。

所以粗略地来说,操作系统的发展是由手工操作阶段过渡到早期单道批处理阶段而具有其雏型,而后发展到多道程序系统时才逐步完善的。

今天操作系统的进一步发展,已进入了计算机网络和数据库的应用阶段。

1.3 多道程序设计的概念

1.3.1 多道程序设计的引入

现代计算机系统(包括部分个人计算机)一般都是基于多道程序设计技术。通常多道程序设计是指在主存中存放多道用户的作业,使之同时处于运行状态共享系统资源。为什么要采用多道程序设计技术呢?正如上面所提到的,由于通道技术的出现,CPU可以把直接控制输入输出工作转给通道。那么为什么CPU要把这工作转交给通道呢?最根本的原因是CPU同常用的输入输出设备(如打印机、读卡机等)之间速度的差距太大。如一台每分钟打印1200行的行式打印机打印一行要50毫秒,而百万次的计算机在此期间大致可执行数万条指令。如果由CPU直接控制打印机,那么在打印一行字符期间,CPU就不能进行其它工作,耽误数万条指令的执行。所以我们把直接控制输入输出工作转交给速度较慢,比较便宜的通道去做。为使CPU在等待一个作业的数据传输过程中,能运行其它作

业,我们在主存中同时存放多道作业。当一个在 CPU 上运行的作业要求传输数据时,CPU 就转去执行其它作业的程序。

1.3.2 多道程序设计的概念

所谓多道程序设计是指“把一个以上的作业存放在主存中,并且同时处于运行状态。这些作业共享处理机时间和外部设备等其它资源”。

对于一个单处理机的系统来说,“作业同时处于运行状态”这显然只是一个宏观的概念,其含义是指每个作业都已开始运行,但尚未完成。就微观上来说,在任一特定时刻,在处理机上运行的作业只有一个。

引入多道程序设计技术的根本目的是提高 CPU 的利用率,充分发挥并行性。这包括程序之间、设备之间、设备与 CPU 之间均并行工作。图 1.5 表示单道程序系统运行情况,图 1.6 表示两道程序系统运行情况。图中实线表示运行时间、水平虚线表示等待时间。

图 1.5 单道作业运行情况

图 1.6 两道作业运行情况

在单道程序系统中,没有任何并行情况存在。在任一特定时刻只有 CPU 或某一个设备在工作。据统计 CPU 的利用率只有 7% 左右。而在多道程序(设计)系统中,随着内存中程序道数的增加,可以提高系统中所有设备和 CPU 的并行性和利用率,尤其是提高了 CPU 的利用率(因为 CPU 是诸设备中最贵的设备)。从理论上来说,如果主存中的作业道数无限增加(实际上是不可能的,主存的容量是有限的),CPU 的利用率应能达到百分之百。

1.4 操作系统的功能和特性

1.4.1 操作系统的功能

对操作系统的功能有着各种不同的认识,比较广泛的看法是把操作系统看成是计算机系统的资源管理者。就是说操作系统主要负责管理系统资源,并调度对系统中各类资源的使用。具体来说,其主要功能有:

(1) 处理机管理:对系统中的各处理机及其状态进行登记、管理各程序对处理机的要求,并按照一定策略将系统中的各台处理机分给要求的用户作业(进程)使用。

(2) 存储器管理:用合理的数据结构形式记录系统中主存储器的使用情况,并按照一定策略在提出存储请求的各作业(进程)间分配主存空间,保护主存储器中的信息不被其它人员的程序有意无意地破坏或偷窃。

(3) 输入输出设备管理:记住系统中各类设备及其状态,按各类设备的特点和不同的策略把设备分给要求的作业(进程)使用。许多系统还十分注意优化设备的调度,以提高设备有效使用率。

(4) 信息管理:在当前的"信息社会"中,对信息的储存和管理受到所有计算机系统的重视,即使许多微型机以及个人计算机,尽管其对四大资源管理功能中其它的管理功能设计往往注意得不很充分。但对信息管理功能的设计一般都给以充分注意,都有较完善的文件系统。

操作系统中的信息管理功能主要涉及文件的逻辑组织和物理组织、目录的结构以及对文件的操作等,近来尤其注意对文件中信息的保护和保密措施。

以上是操作系统的四个主要功能,除此之外,每个操作系统还要提供:

(1) 中断管理系统:它与中断硬件一起处理系统中各种中断事件(见第 2 章)。

(2) 输入输出系统:系统提供的标准输入输出功能,以便用户调用。

(3) 错误处理功能:分析并处理系统中出现的有关错误。

1.4.2 操作系统的特性

多道程序(设计)系统具有很大的优点是它可以使 CPU 和输入输出设备以及其它系统资源得到充分利用,但也带来不少新的复杂问题。多道程序的操作系统具有一些明显的特性:

(1) 并行性:由于主存中存放有多道程序,并同时处于运行状态,即并行运行。一个程序在某个时刻可能在 CPU 上运行,也可能不在其上运行,而正在等待数据传输。

就整个系统来说,由于计算和输入输出操作并行,因此操作系统必须能控制、管理并调度这些并行的动作。除此之外,操作系统还要协调主存中各程序(进程)之间的动作以免互相发生干扰,造成严重后果。这也就是同步(见第 4 章)。总之操作系统要充分体现并行性。

(2) 共享性:在主存中并行运行的程序可要求共享所有的系统资源,因此,① 操作系统要管理并行程序对 CPU 的共享,即负责在并行程序间调度对 CPU 的使用;② 管理对

主存的共享;③ 管理对外部存储器的共享使用以及对系统中数据(或文件)正确的共享。维护数据的完整性。

多道程序系统由于其并行性和共享性的特点而给操作系统带来了许多复杂问题,这些问题将在以后各章中加以研究。

1.5 操作系统的类型

当前计算机已广泛深入人类生活的各个领域,从办公室自动化到为您看家、做饭,从工业控制到科学计算,真是无孔不入。自然,在如此广泛的使用领域中,人们对计算机的要求是十分不同的。于是对计算机上的操作系统的性能要求,使用方式也是十分不同的。因此对操作系统的类型进行分类的方法也很多。例如可以按照机器硬件的大小而分为大型机操作系统,小型机操作系统和微型机操作系统。由于大型机性能较强,附属设备较多,所以价格比较昂贵。机器主人关心的是如何使所有硬件设备得到充分使用,也就是说希望机器有较大的工作负荷,并要求机器能适应各种类型和性质的工作(如科学计算,数据处理,数据库管理和网络等)的通用性。所以很好地调度和管理系统资源就成为大型机操作系统的主要任务。而资源使用的有效性以及机器的吞吐量(指固定时间间隔中机器所完成的作业数),往往是大型机操作系统所追求的主要目标。

对于微型机之类小机器来说,相对比较便宜,因此对资源使用的有效性往往不像大机器那样重视。所以不要求其能适合多种使用方法,而常常只适合某种特定的使用方式。因此小机器的操作系统的设计就反映了这种特性。

另外一种被广泛使用的典型的分类方法是把操作系统分为四类:

(1) 多道批处理操作系统。

(2) 分时操作系统。

(3) 实时操作系统。

(4) 网络操作系统。

下面分别介绍这四类操作系统的特性。

1.5.1 多道批处理操作系统

多道批处理操作系统这个名词不同于 50 年代初的早期单道批处理系统。这二者的区别在于:

(1) 作业道数:单道批处理系统中只有一道作业在主存中运行。而多道批处理系统中同时有多道作业在运行。

(2) 作业处理方式:单道批处理系统是把多个用户作业形成一批,由卫星机将这些作业输入磁带中,然后主机再从该磁带中依次地将作业一个一个读入主存进行处理。作业完成后,将结果也都输出到另一磁带中去,当这批作业全部完成后,再由卫星机把此磁带上的结果通过相应的输出设备输出。处理完一批作业后再处理另一批作业。而在多道批处理系统中(包括网络中的远程批处理),作业可随时(不必集中成批)被接受进入系统,并存放在磁盘输入池中形成作业队列。而后操作系统按一定原则从作业队列中调入一个或多

个作业(视主存自由空间大小而定)进入主存运行。所以“批”的概念已不十分明显。这里所谓的“批处理”是指这样一种操作方式:即用户与他的作业之间没有交互作用,不能直接控制其作业的运行,一般称这种方式为脱机操作或批操作。与之相对应的概念是联机操作,这是指用户在控制台或终端前直接控制其作业的运行,也就是说用户和其作业之间有着交互作用。

多道批处理系统一般用于计算中心的较大的计算机系统中。由于它的硬件设备比较全,价格较高,所以此类系统十分注意 CPU 及其它设备的充分利用,追求高的吞吐量。故多道批处理系统的特点是其对资源的分配策略和分配机构,以及对作业和处理机的调度等功能均经过精心设计。各类资源管理功能既全且强。

1.5.2 分时系统

早期单道批处理和多道批处理系统的出现虽然导致了程序员和操作员两种工作的分工,程序员主要是编制、开发程序,操作员在机房运行程序,从而使得程序员不必进机房,但是在程序开发过程中,这种批处理的操作方式给程序员的开发工作带来了很大的不便。首先是因为程序运行时,一旦系统发现其中有错误时,就停止该程序运行,直到改正错误后,才能再次上机运行。而新开发的程序难免有不少错误或不适当之处需加以修改。这就要多次送到机房上机运行,以便进行调试,这就大大延缓了程序的开发进程。程序员早就希望自己能直接控制程序运行,随时改正错误,以及研究改变某些参数所产生的影响等(这在最初的手工操作阶段是可以做到的)。这就导致了人们去研究一种能够提供用户和程序之间有交互作用的系统。因此在 60 年代初期,美国麻省理工学院首先开始建立了第一个分时系统 CTSS。

所谓分时是指多个用户分享使用同一台计算机,也就是说把计算机的系统资源(尤其是 CPU 时间)进行时间上的分割,即将整个工作时间分成一个个的时间段,每个时间段称为一个时间片,从而可以将 CPU 工作时间分别提供给多个用户使用,每个用户依次地轮流使用时间片。分时系统具有以下特征:

(1) 多路性:一台计算机周围联上若干台终端,每个用户通过终端可以同时使用计算机。

(2) 交互性:分时系统中用户的操作方式是联机的,即用户通过终端可以直接控制程序运行,同其程序之间可以进行“会话”(交互作用)。这样程序员就可以把自己想法通过计算机进行检验,并得到进一步发展。计算机在计算和数据处理方面能力比人强,但在图象识别、思维、判断方面,人的能力更为卓越。这两方面结合起来可以得到更为完美的结果。

(3) 独占性:分时系统往往用来开发程序、处理数据等,在其上处理的作业一般不需要很多的连续的 CPU 时间。因此我们对 CPU 时间及其它系统资源按时间片进行分割,轮流分给终端用户使用。由于用户从键盘输入输出比较慢,有时还要停下来思考。而 CPU 处理速度很快。所以尽管 CPU 按时间片为多个、甚至几十个用户轮流服务,而每个用户的感觉仍然认为自己好像独占着计算机系统,丝毫也没有由于有多个用户共享而延缓其作业的处理速度之感觉。

由于分时系统的主要目的是及时地响应和服务于联机用户,因此分时系统设计的主

要目标是对用户响应的及时性。

1.5.3 实时系统

计算机不但广泛使用于科学计算,数据处理方面,也广泛用于工业生产中的自动控制,实验室中的实验过程控制、导弹发射的控制、票证预订管理等方面,通常我们称之为实时控制。“实时”是指对随机发生的外部事件作出及时的响应并对其进行处理,所谓外部是指来自与计算机系统相连接的设备所提出的服务要求和采集数据。这些随机发生的外部事件并非由于人来启动和直接干预而引起的。实时系统就是以此种方式工作的控制和管理系统。

实时系统通常包括实时过程控制和实时信息处理两种系统。

实时系统与批处理系统、分时系统有何不同呢?

(1) 无论批处理系统,还是分时系统,基本上都是多道程序系统,是属于处理用户作业的系统。系统本身没有要完成的作业,它只是起着管理调度系统资源,向用户提供服务的作用。这类系统可以说是“通用系统”。而许多实时系统则是“专用系统”,它为专门的应用而设计。在此种系统中,系统本身就包含有控制某实时过程和处理实时信息的专用应用程序。所以往往也就无所谓“作业”和“道”的概念,而只有固定的若干“任务”程序。

(2) 实时系统用于控制实时过程,所以要求对外部事件的响应要十分及时,迅速。外部事件往往以中断的方式通知系统,因此实时系统要有较强的中断处理机构、分析机构和任务开关机构。为了能迅速处理外部中断,较常用的中断处理程序及有关的系统数据基最好常驻主存储器中。

(3) 可靠性对实时系统十分重要,因为实时系统的控制、处理对象往往是重要的经济和军事目标,同时又是现场直接控制处理。任何故障往往会造成巨大损失。所以重要的实时系统往往采用双机系统,以保证系统的可靠性。

(4) 实时系统的设计常称之为“队列驱动设计”和“事件驱动设计”。其工作方式基本上是接受来自外部的消息(事件),分析这些消息,而后调用相应的消息(事件)处理程序进行处理。

(5) 上面讲到的是专用实时系统,但许多计算机系统常常把实时系统同批处理系统相结合为通用实时系统。在这些系统中,实时处理作为前台作业,批处理作为后台作业。前、后台作业的区别在于:只有前台作业不需要使用处理机时,后台的作业才能得到处理机的控制权。一旦前台的作业可以开始工作时,后台作业就需立即让出处理机供其使用。

虽然我们把操作系统分成以上三类。但实际的系统往往兼有多道批处理、分时处理和实时处理三者,或其中之二者的功能。在此种情况下,批处理作业往往是作为后台任务。

1.5.4 网络操作系统

1.5.4.1 网络操作系统概述

过去的所谓网络操作系统实际上往往是在原机器的操作系统之上附加上具有实现网络访问功能的模块。在网络上的计算机由于各机器的硬件特性不同、数据表示格式及其它方面要求的不同,在互相通信时为能正确进行并相互理解通信内容,相互之间应有许多约

定。这些约定称之为协议或规程。因此通常将网络操作系统定义为:“网络操作系统(NOS, Network Operating System)是使网络上各计算机能方便而有效地共享网络资源,为网络用户提供所需的各种服务的软件和有关规程的集合。”

网络操作系统除了应具有通常操作系统应具有的处理机管理、存储器管理、设备管理和文件管理外,还应具有以下两大功能:

(1) 提供高效、可靠的网络通信能力。

(2) 提供多种网络服务功能,如:

① 远程作业录入并进行处理的服务功能;

② 文件传输服务功能;

③ 电子邮件服务功能;

④ 远程打印服务功能。

总而言之,要为用户提供访问网络中计算机各种资源的服务。

那么网络软件具体应做些什么呢?让我们来考察一下一台计算机是如何请求网络服务的。假定一台机器上的某用户应用程序提出网络服务请求(通常是 I/O 请求)。将它的请求传送给另一台远程计算机并在其上执行请求,然后将结果返回到第一台机器。现在假定该请求是“从机器 A 上读文件 B 中的 N 个字节”。那么网络软件要做的工作是:

(1) 将这个请求按要求格式组装后,解决如何送到网络上的问题;

(2) 决定怎样到达机器 A?因为按网络的拓扑结构,到机器 A 的链路可能不止一条;

(3) 机器 A 理解什么样的通信软件;

(4) 为在网络中传送,必须改变请求形式(如把信息分为几个短的信息包);

(5) 当请求到达 A 机时,必须检查它的完整性,对它译码,然后送到本机操作系统中执行该请求;

(6) A 机对请求的应答必须经过编码以便通过网络送回去。

国际标准化组织为了对网络软件实行标准化并进行集成,定义了一个软件模型。这就是开放系统互连参考模型(OSI)。该模型定义了七个软件层,如图 1.7 所示。

按此模型一台机器上的每层都假定它与另一台机器上的同层“对话”(图上用虚线表示,称为虚拟通信)。模型中最下面四层又称通信子网。驻留于上三层的软件称为通信子网的用户。网络软件应实现各层应有的功能,并遵照各层间通信的协议(有关图 1.7 中各层功能请参阅有关网络方面的书籍)。

1.5.4.2 Windows NT 的内装网络简介

通常的网络操作系统是在传统操作系统之上附加上网络软件。但 90 年代最新的操作系统 Windows NT 不是这样,它把网络功能做在操作系统之中,而且是该操作系统执行体的输入输出(I/O)系统的一部分,NT 的 I/O 系统包含有五个部分(见第 12 章图 12.1):输入输出(I/O)管理程序;文件系统;缓冲存储管理系统;设备驱动程序;网络驱动程序。

NT 的内装网络是如何工作的呢?首先用户态软件(例如 Win 32 I/O API)通过调用本机 NT I/O 服务子程序发出 I/O 请求(如向对方写盘),于是 I/O 管理程序为它创建一个 I/O 请求包(IRP),并将请求传送给文件系统驱动程序之一——Windows NT 重定向

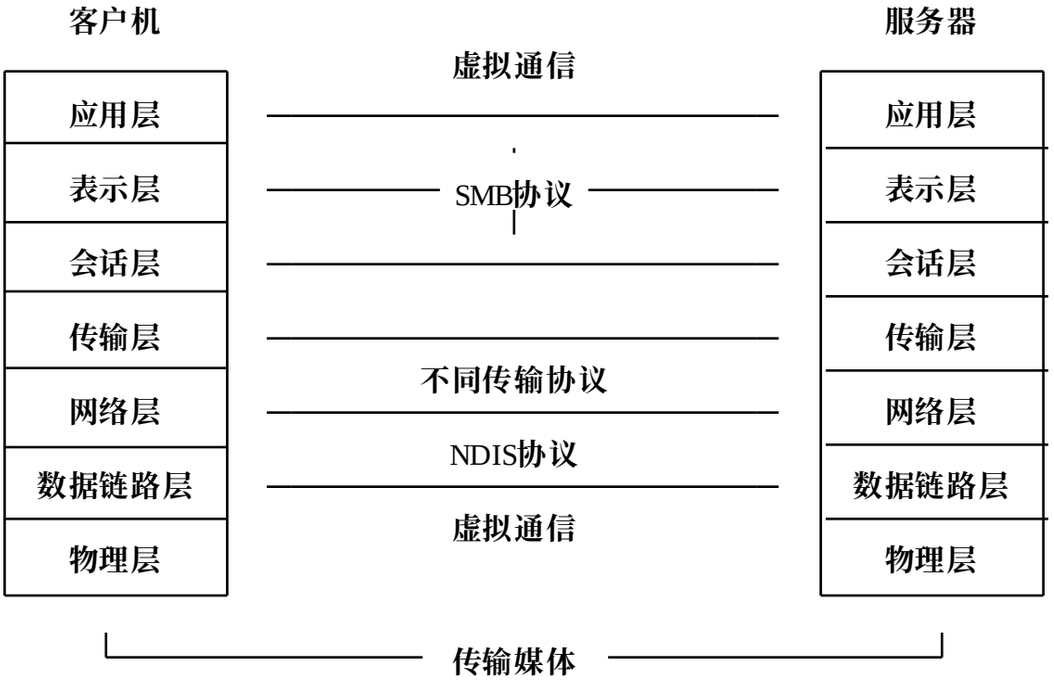


图 1.7 OSI参考模型

程序。重定向程序将请求包(IRP)提交给传输驱动程序,传输驱动程序处理请求包,并将其放在网络上。这样当请求到达 Windows NT 的目的地时,由传输驱动程序接收并复制数据到缓冲区,然后调用文件系统驱动程序,发出 I/O 命令写盘。

由上述内装网络工作过程可以看到 NT 的内装网络有如下特点:

(1) 将联网能力加入到操作系统中,使之成为操作系统功能的一个组成部分;

(2) NT 系统支持文件复制、电子邮件和远程打印,而无需要求用户在机器上再安装任何的网络服务器软件;

(3) 内装网络包含很多部件,但最主要的是重定向程序,服务器和传输驱动程序;

(4) 现存的多种网络,网络驱动程序和网络服务器(如 Novell, Banyan VINES, Sun NFS)在 NT 系统中能很容易地进行数据交换和交互,也就是说支持多个网络协议;

(5) 内装网络是开放式结构。不仅重定向程序、服务器和传输驱动程序都可以被动态地装入和卸出,而且很多不同的这种部件可以并存。详细内容请参阅 12.9 节。

1.6 微型计算机及其操作系统的发展趋势

随着 60 年代后期开始的集成电路的发展和性能/价格比的提高,小型机及微型计算机得到迅猛的发展,1970 年 PDP-11/20 作为第一个微型计算机引入了市场。几乎与此同时,Intel 4004 和 8008 单芯片 8 位机也出现了,在这之后 Motorola 6800 微型计算机在市场上流行起来,接着 Zilog 公司的 Z-80 也脱颖而出,微型机的发展情况如图 1.8 所示。当前比较著名并被广为使用的个人计算机(PC)和微型计算机有:长城 0520 和 IBM-PC, Apple-Ⅱ, M68020(32 位机), Intel-80386(32 位机), Z-8000 等。

由于早期微型计算机主要用于单用户情况下,故初期的微型计算机只有常驻主存的监督程序(Monitors)。因为微型计算机价格十分便宜(从 50 美元到数万美元之间),所以它们的操作系统的设计不像大型计算机的操作系统那样去追求充分发挥 CPU 及所有设

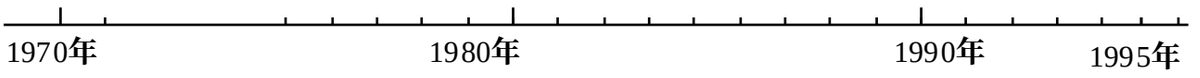


图 1.8　微型计算机发展情况

备的利用率,并且在若干用户之间共享硬件资源。因而它们的设计原则是有所不同的。在微型计算机中基本上不考虑配置批处理系统(见 1.5.1 节),通常是交互作用的使用方式,即实时系统和分时系统(当然在这些系统中也可以有批处理的操作方式)。

由于第一代微型机系统主要是由 8 位单片微处理器组成的微型机系统,其直接寻址范围为 64K 字节,速度也不够快,所以第一代微型机操作系统的功能还不太强,实际上只有信息管理(即文件系统),如 CP/M 操作系统。属于第一代微型机操作系统的还有:Zilog 公司为 Z-80 微型计算机配置的 RIO 单用户操作系统,Apple 公司的 SOS 单用户操作系统(为 AppleⅡ微型机),而 CDOS,RDOS 是在 CP/M 基础上改写成的。

微型机系统的发展迫使一些小型机微型化。如 Data General 公司研制了微型 NOVA 机,配置了四用户多道操作系统 DOS。DEC 公司研制出了由 LSI-11/03 和 LSI-11/23 等微处理器组成微型机系统,并配置 RT-11 和 TSX 等多用户操作系统。

随着微型机技术的发展,单片 16 位微处理器 Intel8086,Z-8000 和 M68000 的出现,使微型机进入新的时期。此时的操作系统如 8086 的 iRMX86,Z-8000 的 ZMOS,VENIX (UNIX 的扩充版)等都是属于多用户操作系统,1979 年期间 CP/M 被扩展成多用户操作系统 MP/M。

近来随着超级微型计算机 32 位机的出现,以及微型计算机应用的日益广泛,并广为应用在多机方式下并行工作的分布式系统中,许多微型计算机已具有以前的大型机和小型机系统中所具有的性能,它们本身就是一个多处理机系统。所以近来推出的许多微型计算机的操作系统已具有了与大型计算机操作系统同样的功能和特点,如采取多用户系统;虚拟存储技术并具有高速缓冲存储器等,表 1.1 列出了几个著名的微型计算机的操作系统的性能特点。由表中可以看出,这些计算机都具有多用户或多任务的多道并行程序设计技术;存储管理上采取了各种虚拟存储管理技术;在存储器的层次体系上增加了高速缓冲存储器技术;在设备管理上采取了多级中断组织和虚拟的输入输出技术(又称 SPOOL 系统),看来这已成为微型计算机操作系统的一个发展趋势。

在微型计算机操作系统的近代发展新趋势中,一个被世界关注并在全世界引起轰动效应的是于 1993 年由美国微软(Microsoft)公司推出的号称 90 年代操作系统的 Windows NT。它集中代表了微型机操作系统的发展方向。它是一个现代的、32 位可移植的操作系统。具有工作站和小型机上的操作系统所具有的强大功能;它支持对称的多处

理,使

表 1.1　著名微型机操作系统性能和特性

机　　型	操作系统名称和性能	存　储　技　术
Intel-80486 (也可在 80386 上运行)	Windows NT 多任务、多线程、对称多处理,内装网络	虚存空间 4000MB,分页,二级页表,32KB 或 64KB 高速缓存
Intel-80386	XENIX386　多用户,多处理机	虚存(分段/分页),有高速缓存
M68020	多用户、多处理机,虚拟输入输出(SPOOL)	虚存(分页)有 2KB 高速缓存
Z-8000		虚存(分页,分段)有高速缓存
IBM-PC	并发 CP/M-86　单用户,多任务 QASIS-16　实时,多任务,多用户 XENIX　分时,多用户	

用客户/服务器模式和内装网络,从而适合分布式处理的环境并具有了多用户的性质;它支持多种操作系统(如 MS-DOS,OS/2,POSIX(基于 UNIX 的国际标准)等应用程序的运行环境,因而被称为"操作系统之母";它支持多种文件系统和带有动态优先级的多任务/多线程环境;它支持 POSIX 及 TCP/IP 等协议的网络功能,从而是一个真正的网络操作系统;它具有良好的安全性,达到美国政府的 C_2 级安全标准;它采用了最新技术并用面向对象方法设计操作系统……。许多精妙的技术和设计使每个关心操作系统的人不能不为之吸引。本书将在第 12 章中全面详尽的对 Windows NT 进行介绍。

习　　题

1-1　微型计算机与大型计算机的硬件组织有何不同特点?

1-2　试述虚拟处理机的概念。

1-3　操作系统与系统中的其它软件以及与硬件是什么关系?

1-4　列举操作系统必须与之接口的所有实体,简要描述每种接口的性质。

1-5　定义、比较下列名词,并写出其反义词。

(1) 联机;　　(2) 分时;

(3) 实时;　　(4) 交互式计算

1-6　什么是操作系统,它的主要作用和功能是什么?

1-7　什么是多道程序设计技术,引入多道程序设计技术的起因和目的是什么?

1-8　试画出三道作业的运行情况。列举多道程序系统中存在哪些并行运行情况。

1-9　多道程序系统具有哪些特性,并设想一下这些特性对操作系统设计将带来什么影响?

1-10 为何要引入分时系统,分时系统具有什么特性?

1-11 比较批处理系统、分时系统和实时系统的特点。

1-12 什么是网络操作系统,它与通常的操作系统有何不同?

1-13 Windows NT 的内装网络有何特点?是如何工作的?属于 NT 的哪一部分?

1-14 以 Windows NT 为代表的当代微型机操作系统具有哪些先进技术和特点?

1-15 为什么把 Windows NT 称为真正的网络操作系统?

第 2 章　操作系统的运行环境

操作系统是管理、调度系统资源,方便用户使用的程序的集合。我们知道,一个程序在计算机上运行是要有一定的条件,或者说要有一定的环境。例如要有处理机,主存及输入输出设备和有关系统软件等。而操作系统作为系统的管理程序,为了实现其预定的各种管理功能,更需要有一定的条件,或称之为运行环境来支持其工作。由图 1.4 可以看出,操作系统的运行环境主要包括系统的硬件环境和由其它的系统软件形成的软件环境,同时操作系统与使用它的人之间也有相互作用。本章主要讨论操作系统对运行环境的特殊要求。

2.1　硬件环境

任何系统软件都是硬件功能的延伸,并且都是建立在硬件基础上的,离不开硬件设施的支持。而操作系统更是直接依赖于硬件条件,与硬件的关系尤为密切。操作系统同硬件的接口不像编译程序那样整齐,统一。对编译程序来说,一套指令系统就是其工作的硬件基础,而操作系统中除通道和中断技术比较集中外,它所要求的其它硬件环境则以比较分散的形式同各种管理技术相结合。所以本节只讨论各种管理技术均要用到的基本的硬件技术和概念,而将与操作系统的四大管理功能密切相关的硬件条件放在以后各章来讨论。

2.1.1　中央处理机(CPU)

操作系统作为一个程序要在处理机上执行。如果一个计算机系统只有一个处理机,我们称之为单机系统。如果有多个处理机(不包括通道)称之为多机系统。

2.1.1.1　特权指令

每个处理机都有自己的指令系统,对于微处理机来说,它的指令系统的功能相对来说比较弱。对于一个单用户,单任务方式下使用的微型计算机,它的指令系统中的全部指令,一个普通的非系统用户通常也都可使用。但是如果某微型计算机是使用于多用户或多任务的多道程序设计环境中,则它的指令系统中的指令必须区分成两部分:特权指令和非特权指令。所谓特权指令是指在指令系统中那些只能由操作系统使用的指令,这些特权指令是不允许一般的用户使用的,因为这些指令(如启动某设备指令,设置时钟指令,控制中断屏蔽的某些指令,清主存指令,和建立存储保护指令等)如果允许用户随便使用,就有可能

使系统陷入混乱。所以一个使用多道程序设计技术的微型计算机的指令系统必须要区分为特权指令和非特权指令。用户只能使用非特权指令,只有操作系统才能使用所有的指令(包括特权指令和非特权指令)。其指令系统没有特权和非特权之分的微型计算机是难以在多道环境下运行的。

那么 CPU 怎么知道当前是操作系统还是一般用户在其上执行呢,这有赖于处理机状态的标识。

2.1.1.2 处理机的状态

处理机有时执行用户程序,有时执行操作系统的程序。在执行不同程序时,根据运行程序对资源和机器指令的使用权限而将此时的处理机设置为不同状态。有些系统将处理机工作状态划分为核心状态,管理状态和用户程序状态(又称目标状态)三种。但多数系统将处理机工作状态较简单地划分为管态(一般指操作系统管理程序运行的状态)和目态(用户程序运行时的状态),也有称之为管理态和问题态。

当处理机处于管理态时可以执行全部指令(包括特权指令),使用所有资源,并具有改变处理机状态的能力。当处理机处于目态时,就只能执行非特权指令。

2.1.1.3 程序状态字 PSW

如何知道处理机当前处于什么工作状态,它能否执行特权指令,以及处理机何以知道它下次要执行哪条指令呢? 为了解决这些问题,所有的计算机(不管是大型计算机还是微型计算机)都有若干的特殊寄存器。如用一个专门的寄存器来指示下一条要执行的指令称程序计数器(PC)。同时还有一个专门的寄存器用来指示处理机状态的,称为程序状态字(PSW)。所谓处理机的状态通常包括条件码——反映指令执行后的结果特征的;中断屏蔽码——指出是否允许中断,有些机器(如 PDP-11)使用中断优先级;CPU 的工作状态——管态还是目态,用来说明当前在 CPU 上执行的是操作系统还是一般用户,从而决定其是否可以使用特权指令或拥有其它的特殊权力。

不同机器的程序状态字的格式,以及其包含的信息都不同,现以微型计算机 M68000 的程序状态字为例来加以介绍,图 2.1 中为其程序状态字 PSW(在 Intel8088 中称为 FLAG),其中各位为:

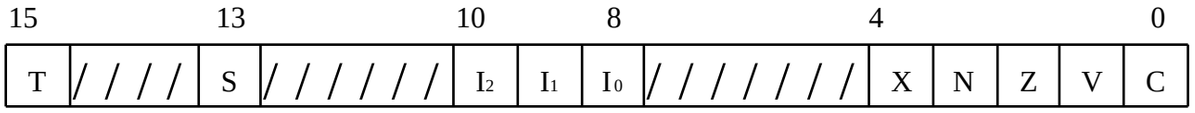


图 2.1 M 68000 的程序状态字

C 进位标志位; V 溢出标志位;

Z 结果为零标志位; N 结果为负标志位。

以上四位称为标准的条件位,几乎所有的微型计算机的 PSW 中都有此四位标志位。

I_0～I_2 这三位是中断屏蔽位,它建立 CPU 的中断优先级的值由 0 到 7,只接受优先级高于此值的那些中断。

S CPU 状态标志位,该位为 1 时则说明 CPU 处于管理态,为 0 时说明 CPU 处于用户态(目态)。

T　　陷阱(Trap)中断指示位，该位为 1 时，则在下一条指令执行后引起自陷中断，这主要用于连机调试排错以及用户程序请求系统服务。

M 68000 还有一个 32 位的程序计数器 PC。

其它微型计算机程序状态字(PSW)的格式大致上都差不多。但是我们前面已说过，一般使用于非多道环境的微型计算机是不分特权指令与非特权指令的，所以处理机也不分管理态与用户态，自然也就不需要上述的 S 标志位。

对于大型机来说，它的程序状态字中就包含有更多的信息。例如 IBM 370 的程序状态字，其格式如图 2.2 所示，我们也对其作一粗略的介绍：

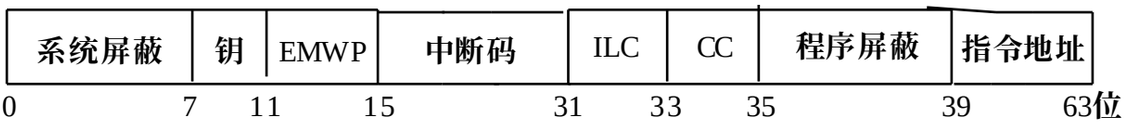


图 2.2　IBM370 的程序状态字PSW

其中第一字节是系统屏蔽码场，用以指出 CPU 是否接受特定通道的中断。而 PSW 中的第五字节中的右四位是程序屏蔽码场，用以说明 CPU 是否接受某种程序性中断。

PSW 中的第二字节的右四位是 EMWP 场，其中 E 位用以指示机器控制方式(基本控制方式和扩展控制方式)，M 位是机器校验方式位，W 位是等待状态位，P 位是处理机工作状态位。而 W 位和 P 位是与 CPU 状态密切相关的。P 位指出了 CPU 当前的工作状态。该位为 0 则表示 CPU 处于管态，为 1 表示 CPU 处于目态。W 位指出 CPU 当前正在运行某个程序(W= 0)还是处于停止状态并正等待中断信号的到来(W= 1)。

第三、第四字节为中断码场，其中含有最近接收到的中断的编码信息。

第二字节的前四位是钥场，是供存储保护用的(见 2.2 节)。ILC 场含有上一次被执行的指令长度，使程序员能回溯一条指令。CC 场是条件码场，含有条件码的当前值。

第六～八字节是指令地址场，指出将要被执行的下一条指令的地址。

2.1.2　主存储器

一个作业必须把它的程序和数据存放在主存中才能运行。在多道程序系统中，有好多道作业的程序和数据要放入主存。操作系统不但要管理这些程序，保护这些程序及其数据不致受到破坏，而且操作系统本身也要在主存中存放并运行。因此主存储器(又称内部存储器)以及与存储器管理有关的机构是支持操作系统运行的硬件环境的一个重要方面。

2.1.2.1　存储器的类型

在微型计算机中使用的半导体存储器有若干种不同的类型，但基本上可划分为两类：

一种是读写型的存储器；即既可以把数据存入其中任一地址单元，并且可在以后的任何时候把数据读出来，或者重新存入别的数据，这种类型的存储器常被称为 RAM (random-access memories)。

另一种是只读型的存储器；我们仅能从其中读出数据，但不能随意的用普通的方法向其中写入数据(向其中写入数据只能在制造该存储器芯片时进行)。这种类型的存储器常被称为 ROM(read-only memory)。作为其变型，还有 PROM 和 EPROM，这是一种可编程的只读存储器，它可由用户使用特殊 PROM 写入器向其中写入数据，而 EPROM 是可

用特殊的紫外线光照射此芯片,以“擦去”其中的信息位使之恢复原来的状态。

通常,在微型计算机中把一些常驻内存的模块以微程序形式固化在 ROM 中,如 IBM-PC 的基本系统中有 48KB 的 ROM,其中 8KB 的基本输入输出系统程序 BIOS 和 32KB 的 CBASIC 解释程序被固化于 40KB 的 ROM 中,另外 8KB 的插座,可供用户使用。而 RAM 主要用作存放随机存取的用户程序和数据。

2.1.2.2 存储分块

存储的最小单位称为“二进位”,它包含的信息为 0 或 1。存储器的最小编址单位是字节,一个字节一般包含八个二进位。在 PDP-11 系列中,两个字节组成为“字”,而在 IBM 系列中一个字是四个字节。

为了简化对存储器的分配和管理,在不少计算机系统中把存储器分成块。在为用户分配主存空间时,以块为最小单位。PDP-11 以 64 字节作为一块,而在 IBM 中是以 2K 字节为一块。

2.1.2.3 存储保护

存放在主存的用户程序和操作系统,以及它们的数据,很可能受到正在 CPU 上运行的某用户程序的有意或无意的破坏,这可能会造成十分严重的后果。例如该用户程序向操作系统区写入了数据,将会造成系统瘫痪。所以对主存中的信息加以严格的保护,使操作系统及其它程序不被错误的操作所破坏,是其正确运行的基本条件之一。下面介绍几种最常用的存储保护机构。

1. 界地址寄存器(界限寄存器)

界地址寄存器是被广泛使用的一种存储保护技术。这种机构比较简单,易于实现。其方法是在 CPU 中设置一对界限寄存器来存放该用户作业在主存中的下限和上限地址,分别称为下限寄存器和上限寄存器。也可将一个寄存器作为下限寄存器,另一寄存器作为长度寄存器(指示存储区长度)的方法来指出程序在内存的存放区域。每当 CPU 要访问主存时,硬件自动将被访问的主存地址与界限寄存器的内容进行比较,以判断是否越界。如果未越界,则按此地址访问主存,否则将产生程序中断——越界中断(存储保护中断)。

2. 存储键

在 IBM 370 系统中,除上述存储保护外,还有“存储保护键”机构来对主存进行保护。前面已经提到,在 IBM 中是把主存储器划分成 2KB 的存储块。为了存储保护的目的,每个存储块都有一个与其相关的由五位二进位组成的存储保护键(图2.3),这五位是附加

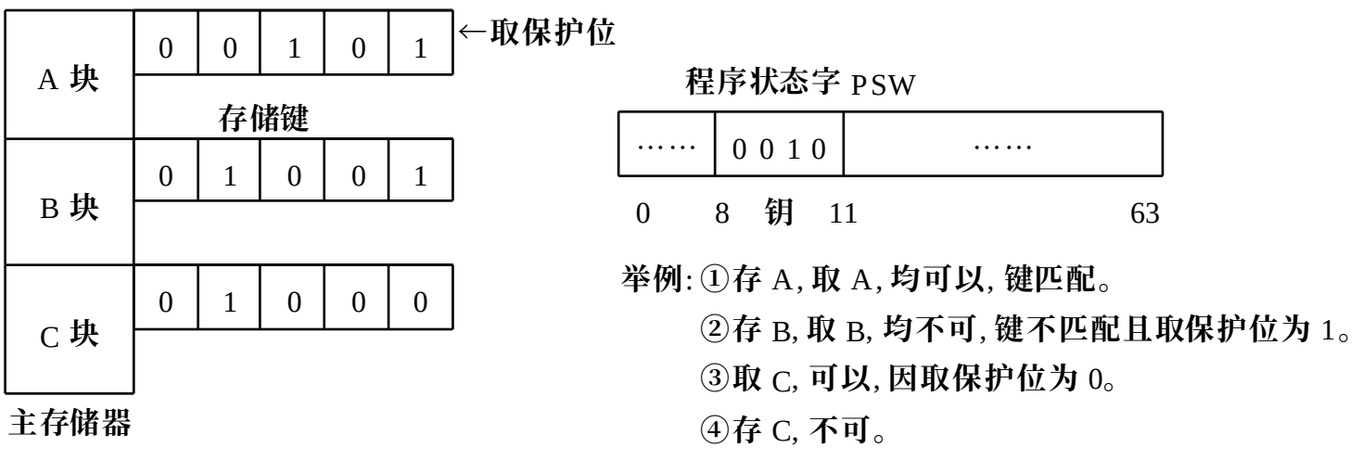


图 2.3 370 存储保护举例

在每个存储块上的,不属于编址的 2KB 块之内。五位中最左边的四位称为存储保护键(简

称存储键),可由操作系统使用特权指令将其设置为 0 到 15 之间的正整数存储键号。当一个用户作业被允许进入主存时,操作系统分给它一个唯一的不与其它作业相同的存储键号(1～15),并将分配给该作业的各存储块的存储键也设置成同样的键号。当作业被操作系统挑选到 CPU 上运行时,操作系统同时将它的存储键号放入程序状态字 PSW 的存储键(钥)场中。这样每当 CPU 访问主存时,都将该主存块的存储键与 PSW 中的钥进行比较。如果相匹配,则允许访问。通常,用户使用 1 到 15 之间的存储键号,而将 0 号键留给系统使用,俗称为"万能键"。所以当系统程序在 CPU 上运行时,CPU 状态为管态,其存储键为 0 号键,此时即使与要访问的存储块的存储键不相匹配,也可进行访问。故操作系统可以访问整个主存。

最右边的一位是"取保护位"。当该位为 0 时,即使存储键不匹配,也允许对该块内的数据进行读访问(取数据),但不许进行写访问。这是为了能在几个用户之间共享信息。此时,即使键不匹配,其他用户也可读该块中的信息(如图 2.3 中的 C 块)。当该位为 1 时,在键不匹配的情况下,禁止对该块进行任何访问,即不提供共享便利,只供用户自己访问(读、写均可以)。

2.1.3 缓冲技术

缓冲是外部设备在进行数据传输期间专门用来暂存这些数据的主存区域。例如,当从某输入设备输入数据时,通常是经过通道先把数据送入缓冲区中,然后 CPU 再把数据从缓冲区读入用户工作区中进行处理和计算。那么为什么不直接送入用户工作区,而要设置缓冲区来暂存呢? 最根本的原因是 CPU 处理数据速度与设备传输数据速度不相匹配,用缓冲区来缓解一下其间的速度矛盾。这就像我们先把货物装入集装箱,然后吊装进船舱的道理相似。如果把用户工作区直接作为缓冲区则有许多不便。首先当从工作区向(从)设备输出(入)时,工作区被长期占用而使用户无法使用。其次为了便利对缓冲区的管理,缓冲区往往是与设备相联系的,而不直接同用户相联系。再者也为了减少输入输出次数,以减轻对通道和输入输出设备的压力。缓冲区信息可供多个用户共同使用和反复使用。每当用户要求输入数据时,先从这些缓冲区中去找,如果已在缓冲区中,这就可减少输入输出次数。若以工作区作缓冲区用,就难以提供此种便利。

为了提高设备利用率,通常使用单个缓冲区是不够的。因为在单缓冲区情况下,设备向该缓冲区输入数据直到装满后,必须等待 CPU 将其取完,才能继续向其中输入数据。要是有两个缓冲区时,设备利用率可大为提高。

目前许多计算机系统广泛使用多缓冲区技术。以 PDP-11 的 UNIX 操作系统为例,整个系统有两个缓冲池。一个缓区池是为了磁盘之类的块设备而设置的,该池共有 15 个缓冲区,每个缓冲区大小为 514 个字节。另一个缓冲池是为慢速字符设备而设置的,该池共有 100 个缓冲区,每个缓冲区大小为 8 个字节。所有的缓冲区都用链指针链入不同的缓冲区队列(详见 UNIX 系统)。当需要缓冲区时,就向操作系统提出请求,操作系统分给一块相应的空闲缓冲区供其使用。

2.1.4 中断技术

前面已多次提到中断技术。中断对于操作系统的重要性就像机器中的齿轮一样，所以许多人把操作系统称为是由“中断驱动”的。

2.1.4.1 中断的概念

所谓中断是指 CPU 对系统中发生的异步事件的响应。异步事件是指无一定时序关系的随机发生的事件。如外部设备完成数据传输，实时控制设备出现异常情况等。“中断”这个名称是来源于：当这些异步事件发生后，打断了处理机对当前程序的执行，而转去处理该异步事件(即执行该事件的中断处理程序)。直到处理完了之后，再转回原程序的中断点继续执行。这种情况很像我们日常生活中的一些现象。例如你正在看书，此时电话响了(异步事件)，于是用书签记住正在看的那一页(中断点)，再去接电话(响应异步事件并进行处理)，接完电话后再从被打断那页继续向下看(返回原程序的中断点执行)。

最初，中断技术是作为向处理机报告“本设备已完成数据传输”的一种手段，以免处理机不断地测试该设备状态来判定此设备是否已完成传输工作。目前，中断技术的应用范围已大为扩大，作为所有要打断处理机正常工作并要求其去处理某一事件的一种常用手段。我们把引起中断的那些事件称为中断事件或中断源，而把处理中断事件的那段程序称为中断处理程序。一台计算机中有多少中断源，这要视各个计算机系统的需要而安排。就 IBM-PC 而论，它的微处理机 8088 就能处理 256 种不同的中断。

由于中断能迫使处理机去执行各中断处理程序，而这个中断处理程序的功能和作用可以根据系统的需要，想要处理的预定的异常事件的性质和要求、以及输入输出设备的特点进行安排设计。所以中断系统对于操作系统完成其管理计算机的任务实在是十分重要的，一般来说中断具有以下作用：

(1) 能充分发挥处理机的使用效率：因为输入输出设备可以用中断的方式同 CPU 通讯，报告其完成 CPU 所要求的数据传输的情况和问题，这样可以免除 CPU 不断地查询和等待，从而大大提高处理机的效率。

(2) 提高系统的实时处理能力：因为具有较高实时处理要求的设备，可以通过中断方式请求及时处理，从而使处理机立即运行该设备的处理程序(也是该中断的中断处理程序)。

所以目前的各种微型机，小型机及大型机均有中断系统。

2.1.4.2 中断逻辑与中断寄存器

如何接受和响应中断源的中断请求，这在总线结构的微型计算机中和非总线结构的大型计算机中是有些不同的，图 2.4 中所表示的是 IBM-PC 的中断源及中断逻辑。在 IBM-PC 中有可屏蔽的中断请求 INTR，这类中断主要是输入输出设备的 IO 中断。这种 IO 中断可以通过建立在程序状态字 PSW 中的中断屏蔽位加以屏蔽，此时即使有 IO 中断，处理机也不予以响应；另一类中断是不可屏蔽的中断请求，这类中断是属于机器故障中断，包括内存奇偶校验错以及掉电使得机器无法继续操作下去等中断源。它是不能被屏蔽的，一旦发生这类中断，处理机不管程序状态字中的中断屏蔽位是否建立都要响应这类中断并进行处理。

图 2.4　IBM-PC 中断逻辑和中断源

此外还有程序中的问题所引起的中断(如溢出,除法错都可引起中断)和软件中断等,由于 IBM-PC 中具有很多中断源请求,它们可能同时发生,因此由中断逻辑按中断优先级加以判定,究竟响应哪个中断请求。

而在大型计算机中为了区分和不丢失每个中断信号,通常对应每个中断源都分别用一个固定的触发器来寄存中断信号。并常规定其值为 1 时,表示该触发器有中断信号,为 0 时表示无中断信号。这些触发器的全体称为中断寄存器,每个触发器称为一个中断位。所以中断寄存器是由若干个中断位组成的。

中断信号是发送给中央处理机并要求它处理的,但处理机又如何发现中断信号呢?为此,处理机的控制部件中增设一个能检测中断的机构,称为中断扫描机构。通常在每条指令执行周期内的最后时刻扫描中断寄存器,询问是否有中断信号到来。若无中断信号,就继续执行下一条指令。若有中断到来,则中断硬件将该中断触发器内容按规定的编码送入程序状态字 PSW 的相应位(IBM 中是 16～31 位),称为中断码。

2.1.4.3　中断类型

无论是微型计算机或大型计算机,都有很多中断源,这些中断源按其处理方法以及中断请求响应的方式等方面的不同划分为若干中断类型,如 IBM-PC 的中断可分为可屏蔽中断(IO 中断),不可屏蔽中断(机器内部故障、掉电中断)、程序错误中断(溢出、除法错等中断)、和软件中断(Trap 指令或中断指令 INT_n)等。

而像 IBM 370 和 43 系列等大型机中把中断划分为五类(许多微型机也有类似的分类):

(1) 机器故障中断:如电源故障,机器电路检验错,内存奇偶校验错等。

(2) 输入输出中断:用以反映输入输出设备和通道的数据传输状态(完成或出错)。

(3) 外部中断:包括时钟中断,操作员控制台中断,多机系统中其它机器的通讯要求中断。

(4) 程序中断:程序中的问题引起的中断,如错误地使用指令或数据、溢出等问题,存储保护,虚拟存储管理中的缺页、缺段等。

(5) 访管中断:用户程序在运行中是经常要请求操作系统为其提供某种功能的服务

(如为其分配一块主存,建立进程等)。那么用户程序是如何向操作系统提出服务请求呢?用户程序和操作系统间只有一个相通的“门户”,这就是访管指令(在大型机中该指令的记忆码是SVC,所以常称SVC指令,在小型和微型计算机中的陷阱指令(Trap指令)也具有类似的功能),指令中的操作数规定了要求服务的类型。每当CPU执行访管指令(SVC指令,Trap指令和Z-8000的SC指令)即引起中断(称访管中断或陷阱中断)并调用操作系统相应的功能模块为其服务。

如果说一个程序在执行中受到了中断,这些中断对于程序来说都是外界(程序之外)强迫其接受的,只有访管中断是它自愿要求的。

需要指出的是,中断类型的这种划分对于大型计算机的中断处理来说是有意义的,而对于微型计算机来说,则是意义不大的,这点将在下面几节中说明。

2.1.4.4 中断响应与中断屏蔽

目前多数微型处理机有着多级中断系统(如图2.4中的IBM-PC的中断逻辑所示),即可以有多根中断请求线(级)从不同设备(每个设备只处于其中一个中断级上,只与一根中断请求线相连在该设备的设备接口上)连接到中断逻辑。如M 68000有七级,PDP-11有四级,Z-8000有三级,Intel-8086有二级(包括IBM-PC),MCS-48只有一级。通常具有相同特性和优先级的设备可连到同一中断级(线)上,例如系统中所有的磁盘和磁带可以是同一级,而所有的终端设备又是另一级。

与中断级相关联的概念是中断优先级。在多级中断系统中,很可能同时有多个中断请求,这时CPU接受中断优先级为最高的那个中断(如果其中断优先级高于当前运行程序的中断优先级时),而忽略其中断优先级较低的那些中断。

如果在同一中断级中的多个设备接口中同时都有中断请求时,中断逻辑又怎么办呢?这时有两种办法可以采用:

(1) 固定的优先数:每个设备接口给安排一个不同的、固定的优先数顺序。在PDP-11中是以该设备在总线中的位置来定,离CPU近的设备,其优先数高于离CPU远的设备。

(2) 轮转法:用一个表,依次轮转响应,这是一个较为公平合理的方法。

CPU如何响应中断呢,这有两个方面的问题:

(1) 一是CPU何时响应中断:通常在CPU执行了一条指令以后,更确切地说是在指令周期最后时刻接受中断请求。在大型计算机中则是在此时扫描中断寄存器。

(2) 其次是如何知道提出中断请求的设备或中断源。因为只有知道中断源或中断设备是谁,才好调用相应的中断处理程序到CPU上执行,这也可以有两种方法:一是用软件指令去查询各设备接口。但这种方法比较费时。所以多数微型机对此问题的解决方法是使用一种称为“向量中断”的硬件设施。当CPU接受某优先级较高的中断请求时,该设备接口给处理机发送一个具有唯一性的“中断向量”,以标识该设备。“向量中断”设施在各计算机上实现的方法差别比较大。以PDP-11为例,它将主存的最低位的128个字保留作为中断向量表,每个中断向量占两个字。中断请求的设备接口为了标识自己,向处理机发送一个该设备在中断向量表中表目地址的地址指针。

在大型机中,中断优先级按中断类型划分,以机器故障中断的优先级最高;程序中断和访问管理程序中断次之;外部中断更次之;输入输出中断的优先级最低。

有时在 CPU 上运行的程序,由于种种原因,不希望其在执行过程中被别的事件所中断,这种情况称为中断屏蔽。在大型计算机中,通常在程序状态字 PSW 中设置中断屏蔽码以屏蔽某些指定的中断类型。在微型计算机中,如果其程序状态字中的中断禁止位建立后,则屏蔽中断(不包括不可屏蔽的那些中断)。如果程序状态字中的中断禁止位未建立,则可以接受其中断优先级高于运行程序的中断优先级的那些中断,另外在各设备接口中也有中断禁止位可用以禁止该设备的中断。

2.1.4.5　中断处理

微型计算机和大型计算机的中断处理大致相同,都是由计算机的硬件和软件(或固件)配合起来处理的。下面以 IBM-PC 的中断处理为例来研究中断处理过程。当处理机一旦接受某中断请求时,则首先由硬件进行如下操作(在微型计算机中称之为隐操作):

(1) 将处理机的程序状态字 PSW 压入堆栈;

(2) 将指令指针 IP(相当于程序代码段的段内相对地址)和程序代码段基地址寄存器 CS 的内容压入堆栈,以保存被中断程序的返回地址;

(3) 取来被接受的中断请求的中断向量地址(其中包含有中断处理程序的 IP,CS 的内容),以便转入中断处理程序;

(4) 按中断向量地址把中断处理程序的程序状态字取来,放入处理机的程序状态字寄存器中。

中断处理的硬件操作如图 2.5 所示,其主要作用是保存现场并转入中断处理程序进行中断处理。当中断处理完成后,要恢复被中断程序现场,以便返回被中断程序执行,即把原来压入堆栈的 PSW,IP,CS 的内容取回来。

图 2.5　中断处理

大型机的中断处理如图 2.6 所示。

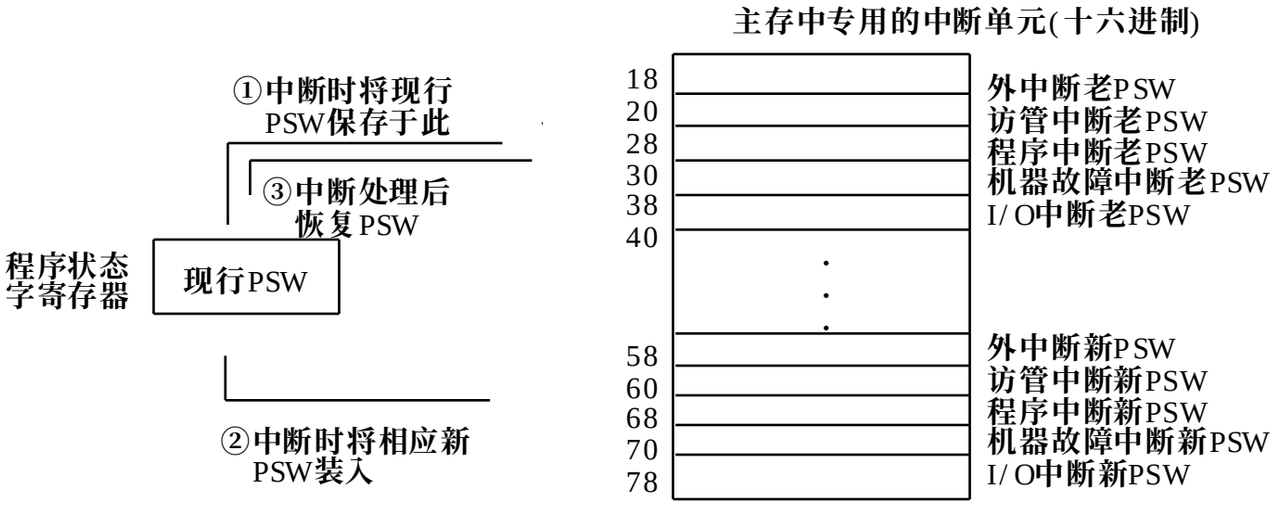


图 2.6 IBM 的中断处理过程

2.1.5 时钟、时钟队列

在计算机系统中(无论是微型计算机还是大型计算机)设置时钟是十分必要的。这是由于时钟可以为计算机完成以下的必不可少的工作:

(1) 在多道程序运行的环境中,它可以为系统发现一个陷入死循环(编程错误)的作业,从而防止机时的浪费;

(2) 在分时系统中,用间隔时钟来实现用户作业间按时间片轮转;

(3) 在实时系统中,按要求的时间间隔输出正确的时间信号给一个实时控制设备(如A/D 转换设备);

(4) 定时的唤醒那些要求延迟执行的各个外部事件(如定时为各进程计算优先数,银行系统中定时运行某类结帐程序等);

(5) 用作可编程的波特速率发生器;

(6) 记录用户使用各种设备的时间;

(7) 记录某外部事件发生的时间间隔;

(8) 提供用户和系统所需要的绝对时间,即年、月、日。

由上述时钟的这些作用可以看到,时钟是操作系统运行的必不可少的硬件设施,所以现在的微型机系统中均有时钟。在微型机系统中,通常只有一个间隔时钟(也作为绝对时钟),在大型机中时钟类型通常要多些。但不管是什么时钟、实际上都是硬件的时钟寄存器,按时钟电路所产生的脉冲数对这些时钟寄存器进行加 1 或减 1 的工作。下面简要介绍主要的时钟的工作特点:

1. 绝对时钟:记录当时的时间(年、月、日、时、分、秒),以便打印统计报表和日记使用。在 IBM 370 和 43 系列中,绝对时钟是一个 64 位的寄存器。它的操作是由操作员将当时的正确时间送入时钟寄存器作为其初值,以后每隔 1 微秒自动加 1。故其可记录的时间大约是 148 年。一般来说,绝对时钟比间隔时钟更准确。当计算机停机时,间隔时钟值不再被修改,而绝对时钟值仍然自动修改。在微型计算机如 IBM-PC 中,只有一个 16 位的寄存器,每次开机时由用户输入时间作为初值。

2. 间隔时钟,又称相对时钟,也是通过时钟寄存器来实现的,由操作人员置上时间间隔的初值(在 IBM-PC 中以秒为单位,在 IBM360 中以三百分之一秒为单位),以后每经过

一个单位的时间,时钟寄存器的值减 1。直到该值为负时,则触发一个时钟中断,并进行相应的处理。

尽管在有些大型机中除了有绝对时钟、相对时钟外,还可能有比较时钟和更准确的计时器,但时钟的数量终究是很少的(一个或几个)。然而往往有很多进程要求在某时间间隔后,或定时地唤醒它运行,也就是说要有自己的间隔时钟。为此我们可以通过软件为每个进程提供其需要的软时钟(或称虚拟时钟)。而时钟队列就是实现这种技术的一种方法。尽管具体实现方法各个系统可以不同,但基本原理相似。假定现在有四个作业,A 作业要求从现在起过 50 毫秒后运行,B 作业要求从现在起过 60 毫秒运行,C 和 D 作业都要求 65 毫秒后运行。此时钟队列我们用表的形式组织,如图 2.7 所示。

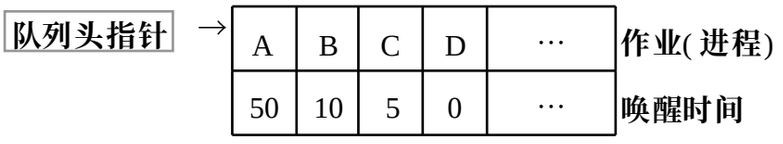


图 2.7　时钟队列组织

时钟队列头指针指出时钟队列在主存中的地址。时钟队列中的唤醒时间是采取时间增量的方法来登记。每当时钟经过一个毫秒时,时钟中断程序把时钟队列中的第一个作业的时间增量减 1。直到该值为零时,唤醒作业 A 运行。同时作业 B 成为时钟队列中的第一个作业(队列头指针指向作业 B)……,若作业 A 在 50 毫秒后还要运行,则按时间先后插入队列。若作业 D 是队尾,则 A 在 D 之后重排队,其时间增量为:50- 10- 5- 0= 35。即将作业再次运行的时间间隔减各作业时间增量之和。

2.2　操作系统与其它系统软件的关系

操作系统是整个计算机系统的管理者,是系统的控制中心。它不但控制、管理着其它各种系统软件,而且与其它系统软件共同支持用户程序的运行。可以说操作系统和这些软件构成一个以操作系统为中心的‘环境”,以便用户程序运行。既然是共同构成一个环境,那么操作系统的功能设计必然受到这些系统软件的功能强弱和完备与否的影响。所以也可把其它系统软件看作操作系统运行环境的一部分。本节只涉及操作系统中常用的一些软件技术。

2.2.1　作业、作业步和进程的关系

前面已经不加区分地多次提到用户、作业,程序等,所谓用户是指要计算机为他工作的人。而作业是用户要求计算机给以计算(或处理)的一个相对独立的任务。一个作业一般可以分成几个必须顺序处理的工作单位(或步骤),称为作业步。例如一个用高级语言写的用户作业,在计算机上运行要分成三步:先编译,第二步是将编译后的主程序中所用到的库程序和子程序都连接装配成一个完整的程序,第三步才是运行该装配好的程序。而一个作业步又可细分为若干个作业步任务——进程。而一个进程又可能要执行多个程序(见第 3 章),因此其具体关系如图 2.8 所示。

图 2.8　作业、作业步和进程的关系

2.2.2　重定位的概念

重定位概念是多道程序系统中最基本的概念。为了弄清什么是重定位，我们必须先区分绝对地址、相对地址和逻辑地址空间等概念。

2.2.2.1　绝对地址、相对地址和逻辑地址空间

绝对地址是指存储控制部件能够识别的主存单元编号(或字节地址)，也就是主存单元的实际地址。

相对地址是指相对于某个基准量(通常用零作基准量)编址时所使用的地址。相对地址常用于程序编写和编译过程中。由于程序要放入主存中才能执行。因此指令、数据都要与某个主存绝对地址发生联系——放入该主存单元。但是由于多道程序系统中，主存将存放多道作业。因此程序员在编写程序时，事先不可能了解自己的程序将放在主存中何处运行，也就是说他不可能用绝对地址来编写程序。因此往往用相对于某个基准地址来编写程序、安排指令和数据的位置。这时用的地址即是相对地址。所以相对地址是使用于程序编写和编译中的地址系统。

逻辑地址空间是指一个被汇编、编译或连接装配后的目标程序所限定的地址的集合。由于编译程序对一个源程序进行编译时，总是相对于某个基准地址(通常是零)来分配程序的指令和数据的地址，即相对地址而不是绝对地址(同程序员编写程序时一样)。我们把程序中这些相对地址的全体称为相对地址空间(或称逻辑地址空间，或用户地址空间)，这是相对于实际的主存地址空间(又称物理地址空间)而言。引入逻辑地址空间主要是为了在多道程序系统中研究如何把逻辑地址空间变换(或称映象)为实际的主存地址空间的子集，或者把某个相对地址映象为主存绝对地址。

2.2.2.2　静态重定位

在多道程序环境下，用户不可能决定自己使用的主存区，因而在编制程序时常按(以零作为基准地址)相对地址来编写。这样，当程序放入主存时，如果不把程序中与地址有关

的“项”变成新的实际地址,而是原封不动的装入,那么程序就不能正确执行(除非有动态地址变换机构,见 7.3.4 节)。如图 2.9 中,图(a)是一个简单的程序。第一条指令是把数据 A(放在相对地址 6 中)取到 1 号寄存器,第二条指令是把 B(放在相对地址 8 中)与 1 号寄存器内容相减,第三条指令是把 1 号寄存器内容送入相对地址 10 中去。如果这程序原封不动地装入主存中自 400 号单元起的存储区中[图 2.9(b)],就无法正确执行。因为第一条指令的含义是把绝对地址为 6 的单元中的内容取到 1 号寄存器,而不是把放在 406 单元的 A 取来。其它两条指令也有同样问题。因此要使程序装入主存后能正确执行,就必须修改程序中所有与地址有关的项,这就叫程序的重定位。故重定位是把程序中相对地址变换为绝对地址。

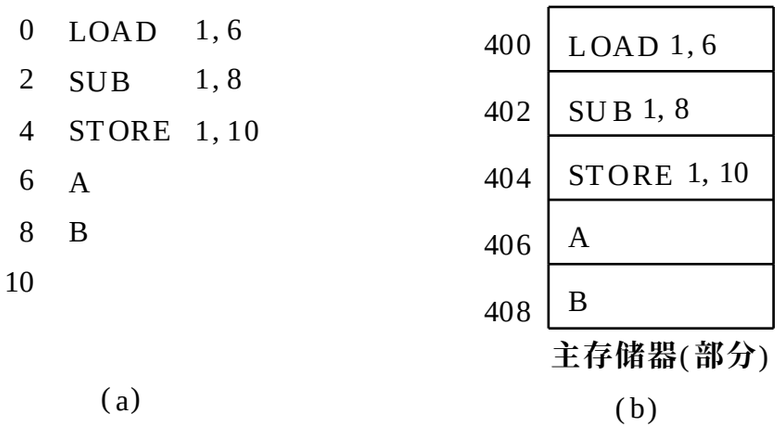


图 2.9 程序装入举例

对程序进行重定位的技术目前按重定位的时机区分为两种:

(1) 静态重定位:它是在程序装入主存时,由连接装入程序进行重定位。程序开始运行前,程序中各与地址有关的项均已重定位完毕(即已将程序中的相对地址转换为绝对地址了)。

(2) 动态重定位:重定位不是在程序装入过程中进行。在处理机每次访问主存时,由动态地址变换机构(硬件)自动进行,把相对地址转换为绝对地址。

本节只介绍静态重定位,关于动态重定位技术将在 7.3.4 节中介绍。

静态重定位是要把程序中所有与地址有关的项在程序运行前(确切地说是在程序装入主存时)修改好。程序中与地址有关的项都包括什么呢?主要有指令、数据和地址指针。其中数据项与地址的关系只表现该数据项存放的位置而不是数据项的内容。而地址指针其存放位置与内容都与地址有关,指令中含有操作码和操作数。其中操作数可能是直接数、寄存器号和操作数的存放地址。也就是说指令中的各项并非都与地址有关,只有指令本身存放的地址和操作数存放地址是与地址有关的。因此当操作系统为某目标程序分配了一个以 B 为起始地址的连续主存区后,重定位过程就是把每个与地址有关的项都加上“B- R”(设 R 是该程序编址时的基准地址,通常 R= 0)即可。

2.2.3 绝对装入程序和相对装入程序

假若运行学生作为练习的小的源程序,此时,编译后立即执行,不需要装入程序来做任何工作。但实际上许多用户的程序往往要调用许多过程和子程序。这些过程和子程序首先要同主程序装配起来,形成一个完整的大程序才能运行。这些过程和子程序很可能不

是同一次编译的。因此它们的地址空间之间不会已建立好某种正确关系,往往都是“可浮动”的相对地址空间。这就需要系统提供把这些过程和子程序找出来(从库中),并把它们同主程序装配起来的功能,这就是连接-装入程序(也叫连接程序,装入程序或连接编辑程序等)的功能。其次由于用户程序往往很大很复杂。编译完了后或者不想马上运行,或者要多次运行。于是下次运行时就要把上次已按相对地址编译(或装配)好的目标程序重新装入运行。这任务也要由装入程序来完成。因此,对于一个计算机系统来说,连接装入程序是不可缺少的,它与编译或汇编程序的功能密切相关。通常,连接装入程序可分为两类:绝对装入程序和相对装入程序。

2.2.3.1 绝对装入程序

在个人计算机(如 IBM-PC)中,用户能使用的主存起始地址是可以知道的。这种机器上的编译和汇编程序往往把源程序翻译成绝对地址形式的目标程序(以该机器的用户可用的起始地址作为基准地址)。因此当需要再次装入目标程序时,就十分简单,没有什么重定位问题。只要按其给出的起始地址,依次地将程序读入即可。

2.2.3.2 相对装入程序——连接装入程序

多数多道程序系统使用相对装入程序(或连接装入程序)。其主要功能是把主程序同被其调用的各子程序连接装配成一个大的完整的程序,并装入主存运行。但这里有两个具体问题。

(1) 由于装入程序要对诸程序进行重定位,而程序中有些项与地址无关(如直接数、寄存器号),有些项与地址有关(如指令存放地址、数据地址和地址指针等)。那么装入程序如何识别它们。通常有两种办法:

① 对程序中各数据项附加上指示字,以说明其是否需要重定位。

② 使用一个与该程序相关联的重定位表。依次给出那些要重定位的数据项。

无论哪种办法都对编译或汇编程序提出了附加要求,因为这些工作都要由它们在编译或汇编过程中来完成。

(2) 将主程序同各程序段连接起来。这个过程比较复杂。我们通过例子来说明其基本原理。假定有一个程序 P(如图 2.10 所示),它既可以被其它程序调用(通过用符号定义的入口点如 P, e, d),也可以调用别的程序模块。前一种情况称为内部定义符号,后者称为外部调用符号。一个源程序经编译或汇编后生成的可重定位目标模块必须明显地给出这些内部符号和外部符号,以供连接装入程序使用。因此与每个可重定位目标段相关联的除重定位表(又称重定位词典)或指示字外,还应有一张内部定义符号表和外部调用符号表(图 2.11)。在内部定义符号表中要依次给出每个内部符号名和它在本程序中的相对地址(当该段被连接装入程序重定位时,该相对地址就重定位成绝对地址)。外部调用符号表要包含本程序中所调用的全部外部符号名。当该程序进行重定位时,可把程序中的所有外部符号调用处,用间接寻址方法处理(间接地址指向外部调用符号表中的相应表目)。外部符号被定义后,将其绝对地址填入外部符号表即可。

以上是程序在连接前所必须做的准备工作。在进行具体连接和装入时,既可用一趟技术,也可用两趟技术来完成。两趟法有下列步骤:

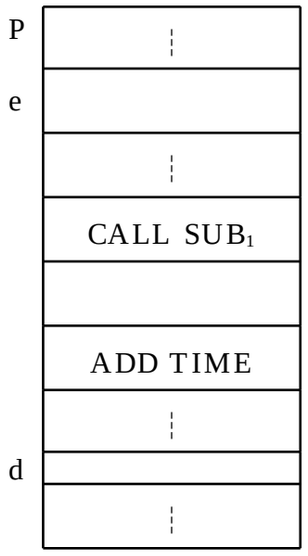


图 2.10　程序调用例子

内部定义符号:

P, e, d。

外部调用符号:

SUB_1,

TIME。

图 2.11　目标模块的结构(与图 2.10 对应)

第一趟

(1) 找到下一个要连接的程序段 P(该段应是可重定位的目标程序);

(2) 重定位并装入 P;

(3) 用 P 的内部定义符号修正全程符号表(全程符号表是被连接的各程序段所定义的符号全体);

(4) 对每个程序段都重复(1)—(3)步。

第二趟

(1) 找到要连接程序段 P 的外部调用符号表;

(2) 从全程符号表中找出该符号的地址填入程序 P 的外部符号表中;

(3) 对每一个被连接的程序段 P 重复(1),(2)步;

这个连接过程类似于标准的两趟汇编程序中所使用的处理过程。

2.3　操作系统与人的接口

无论在联机操作还是在脱机操作情况下,用户除了有程序要机器运行外,还应告诉机器如何来运行自己的程序。例如该程序要不要编译,用哪种语言的编译程序?是否要连接。编译时出现各种不同错误如何处理等。这在脱机批处理情况下更为需要。为此每个机器的厂家都提供给用户使用程序运行意图的说明手段——键盘命令和(或)作业控制语言。所以通常操作系统与用户的接口有两个方面:

(1) 用户程序:用户通过程序中的指令序列让机器为其完成要求的工作。在程序中用

户也可以要求操作系统的某些功能模块给以服务。这时它通过访管指令(SVC)进入操作系统。

(2) 作业控制说明:用户用作业控制语言编写作业说明卡(书)来告诉操作系统他对程序的运行意图。在联机操作情况下使用键盘命令和会话语言。由于各计算机系统的作业控制语言相差很大,至今没有形成统一的标准版本,所以在此只作一些简单介绍。

2.3.1 作业控制语言

目前存在两种形式的作业控制语言。一种相当于汇编语言(如 IBM370 的作业控制语言),一种类似于高级算法语言(如 1900 系列的 George 语言)。要想详细介绍任何一种作业控制语言,既无必要也不可能。本节只通过 IBM 的作业控制说明的例子来介绍其主要的常用语句。

一个 IBM 的脱机作业卡片叠包含三种类型的卡片:

(1) 作业控制卡:其上有作业控制命令的卡片。所有作业控制卡的第一、二列为"//";

(2) 源语句或目标语句卡:其中各卡片上为程序语句或机器指令;

(3) 数据卡:包含程序所用的数据(如果有数据时)。

下面是一个简单的例子:

```
//          JOB     NAME=Tu-Li,ACCOUNT=1073,
                    TIMELIMIT=30,
//STEP1     EXEC    C
//D1        DD      DSNAME=DATA1
                    DCB=(RECFM=FB,                  第一作业步,C 编译
                    LRECL=80,BLKSIZE=800
DATA1       DD      *
            ¦       输入数据
/*
//STEP2     EXEC    LINKER,COND=(4,LT,STEP1)        第二作业步,将 C 的输出与库
//D2        DD      DSNAME=Win                      程序相连接(仅当上一步没有
//D3        DD      DSNAME=DATA3                    4 级以上严重编译错误时才执
//SYSLIB    DD      DSNAME=C LIBRARY                行该作业步)
/*
//STEP3     EXEC    PRG=Win,MEMORY=100K
//OUTPUT    DD      UNIT=PRINTER                    第三作业步,执行用户程序
//INPUT     DD      *
            ¦       输入数据
/*
```

其中所用到的作业控制命令语句有:

(1) 作业标识命令:用以标识一个作业的开始。它作为一个作业卡片叠的第一张。一般格式为

//[作业名]␣JOB␣操作数(以逗点分隔)

[]中内容可缺省。␣表示空格。

操作数　用以说明该作业的一些参数。通常包括帐号,用户名,作业类别,估计的运行时间、主存空间要求和优先数等。

(2) 执行命令:用以标识一个作业步的开始,并指出要执行的程序名。其一般格式为:

```
                    PROC= 过程名
//〔作业步名〕␣EXEC␣ 程序名                  其它参数
                    PGM= 程序名
```

{　}中内容必须选其中一种。

例子中的三个执行语句分别标识了作业步 1,作业步 2 和作业步 3(即 STEP1, STEP2, STEP3)。第一个执行语句说明执行 C 编译程序。第二个执行语句说明执行 LINKER(连接)程序,并用 COND 参数说明执行的条件是 STEP1 作业步没有 4 级以上的编译错误。第三个执行语句说明执行第二作业步的输出文件 Win。而 MEMORY 参数说明主存大小的要求。

(3) 数据定义命令:用来描述作业所使用的数据文件(或称数据集)。其一般格式为:

//数据集名␣DD␣以逗点分开的操作数

操作数场包括数据文件的名字、属性、要求的输入输出设备以及所需的外部存储器空间等。以第一和第三作业步中的数据命令语句为例,UNIT 参数说明使用的输入输出设备。DCB 参数说明执行时数据集的物理形式。RECFM= FB 参数是说明数据集的记录格式是定长记录、成块形式(若干个记录合成磁带上一个数据块)。LRECL= 80 参数说明逻辑记录长为 80 个字节。BLOCKSIZE= 800 说明数据块大小为 800 字节。DSNAME 参数说明数据集的名字。* 参数指示此卡片后面的诸卡片即为数据卡。

(4) 定界命令:用以标识一个数据文件的结束。其一般格式为:/* 。

2.3.2　联机作业控制——终端命令和图形用户接口(GUI)

用户在分时系统的终端上工作,直接通过键盘打入命令来控制其作业的运行,这种方式称联机作业控制方式。它不需要像脱机批处理作业那样,除程序外还要提交一分作业控制说明来控制作业运行。但用户必须借助终端命令或图形用户接口(GUI)与分时操作系统通讯,把用户意图告诉系统,以完成计算任务。现将这两种方法加以简略介绍。

2.3.2.1　终端命令

终端命令也是一种语言,但不同于一般程序设计语言,也没有标准化。因此各个系统往往按照自己的设计构成一套命令。但各系统的命令按功能来说大致都包含以下几类命令:

系统访问命令;　　资源分配命令;

程序运行命令;　　前后台作业转换命令;

程序开发命令;　　系统管理命令。

文件操作命令;

系统访问命令是为用户提供进入和退出系统的手段。用户进入系统时还要打入帐号

和口令,以便系统检验其进入要求的合法性。

程序运行命令通常包括编译、装入目标程序,执行程序,停止运行,继续运行和运行服务程序等命令。

程序开发命令主要是指用户在终端前开发新的程序所使用的命令。一般来说,其开发步骤是首先利用编辑程序建立起正确的源程序文本并存入盘。其次,通过编译程序编译,得到目标程序文件,然后由连接程序将所需的各目标模块连接成一个可运行的程序文本,再进行运行及联机调试。PDP 中用 New 命令建立新的文件,用 Run $ EDT 命令进行编辑。

文件操作命令包括建立新的文件,删去文件,列出文件目录、改变文件名称,复制和比较文件等命令。

资源分配命令通常包括分配设备和主存等命令。但对于资源全部由系统控制的分时系统无需此类命令。在 PDP 系列中有设备分配和释放命令(ASSIGN 和 DEASSIGN 命令),检查系统和作业资源使用情况等命令。

前后台作业转换命令用于对终端用户的作业转换控制方式。在分时与批处理并存的系统中,称在终端上联机工作的作业为前台作业,而把批处理控制下的作业称为后台作业。有时终端用户的作业投入运行后,不需要用户在终端前进行干预,它可要求脱离终端而运行,为此需设置把作业从前台转为后台(转为批处理控制下的作业)或从后台转向前台的命令(重新要求变成联机作业)。

系统管理命令是为操作员在控制台发出一些命令,以生成一个合适的系统(系统功能可按需要在生成时加以裁剪),进行日常的管理和维护(记帐、统计、改变口令,清理磁盘目录等),以及关闭系统。

有关这方面详细的具体内容请参看机器的用户指南。

2.3.2.2 图形用户接口(GUI)

以终端命令和命令语言方式来控制程序的运行虽然有效,但给用户增加了很大的负担,即用户必须记住各种命令,并从键盘键入这些命令以及所需的数据,以控制他(她)们的程序的运行。随着大屏幕高分辨率图形显示和多种交互式输入输出设备(如鼠标、光笔、触摸屏、图形板控制杆等)的出现,于是要求改变这种“记忆并键入”的操作方式,以达到对用户更友好,使接口图形化的目标在 70 年代的国际会议上被多次讨论,并于 80 年代后期广泛推出。这种图形用户接口的目标是通过对出现在屏幕上的对象直接进行操作,以控制和操纵程序的运行。例如,用键盘和鼠标对菜单中的各种操作进行选择,从而命令程序执行用户选定的操作;用户也可以通过滑动滚动杆上的滑动块使列表盒(list Box)中的选择项表目上、下(或左、右)滚动,以使所要的选择项出现在屏幕上,并以鼠标点取的方式来选择操作对象(如文件);用户也可以用鼠标拖动屏幕上的对象(如某图形或图符)使其移动位置或旋转、放大和缩小。这种图形用户接口大大减少或免除用户的记忆工作量。其操作方式从原来的“记忆并键入”改为“选择并点取”。极大地方便了用户,受到普遍欢迎。目前图形用户接口(GUI)是最为常见的人机接口(或用户界面)形式。支持 GUI 的接口系统称为窗口系统,它已成为操作系统的一个重要组成部分。目前最为著名的窗口系统是 Windows 3.x(Windows NT 等系统使用)和 X 窗口系统(UNIX 系统使用)。

国际上为了促进 GUI 的发展于 1988 年制订了 GUI 标准, 该标准规定 GUI 由以下部件构成:

(1) 窗口: 在终端屏幕上划分出的一个矩形区域以实现用户与系统的交互, 窗口由标题条、菜单条、边框、控制按纽和用户区组成。应用程序可同时打开多个窗口, 窗口是彼此独立的。

(2) 菜单: 是一系列可选的命令和操作的集合。

(3) 列表盒: 用以显示一组较长或变长的选择项。由标题、包含列表的窗口和水平与垂直滚动杆三部分组成。

(4) 表目盒: 用以输入字符串。

(5) 对话盒: 有两类对话盒, 消息对话盒, 用以显示信息或请求信息; 通用对话盒, 用以输入文件名, 命令或选择项。

(6) 按纽: 与平常的按纽相似, 用以控制程序。

(7) 滚动杆: 通过滑动滑动块使选择项表目滚动以便快速浏览选择项。

有关图形用户界面详细情况, 请参阅有关的窗口系统资料。

* 2.4 固件——微程序设计概念

自从微程序设计技术进入了实用阶段以后, 硬件和软件之间的界面变得愈来愈不清楚了。硬件环境和软件环境自然也不好截然分开。许多原属软件的功能, 通过微程序设计技术可以转化为硬件, 也就是通常所说的固化, 故称这些具有软件功能的硬件为固件。由于现在绝大多数微型机和所有的大型机均采用微程序设计技术, 并且被广泛应用于操作系统设计中, 所以我们简单介绍微程序设计的概念。

2.4.1 微程序设计的概念

一台计算机可以看成是由两种线路组成的: 数据流线路和控制线路。数据流线路包括导线与存储元件。在它上面流动着表示数据的电信号。一般在数据流线路中包含着大量的分枝和通路。控制线路则译出计算机中的指令, 并在复杂的数据流线路中确定要用哪条数据通路。所以计算机的设计主要反映在数据流线路和控制线路的设计中。

早在 50 年代初, 剑桥大学数学系教授 M · V · Wilkes 就认识到计算机控制线路的操作是由一系列基本的动作组成, 其形式很像计算机中的程序, 并提出一种微程序设计的概念。人们把这些基本的动作称为微操作或微指令。一般说来, 每一条微指令都是计算机硬件中最基本的操作。例如:

将主存储器的缓冲寄存器中的一个字节送累加器;

清除加法器;

将指令地址寄存器增加一个固定值(通常为 1);

将加法器中内容送累加器。

控制器完成的每一个操作都要用类似上述微指令组成的微程序。而每执行一条机器指令将包括若干个控制器操作。例如一条简单的加法指令就需要一段微程序来执行取指

令操作、控制运算部件操作、控制逻辑部件操作等。所谓微程序设计是指把计算机控制器的操作用微指令编成程序(称微程序或微代码)来实现。

微程序设计的优点是:

(1) 机器控制线路的设计可以标准化。既方便又节省时间。

(2) 便于修改、维护、检查(微诊断)。这是因为用微程序控制,避免了过去在组合逻辑控制中所存在的微操作控制杂乱无章、很不规则的状况以及线路像一个凌乱的树形网络的这些缺点。从而带来了设计方便,修改、维护和检查容易的优点。

(3) 一组指令系统可以通过微程序以适合多种型号的计算机,实现了兼容,方便了用户使用,并为发展系列机创造了良好的条件。

(4) 一台计算机通过微程序可以包含若干组指令系统,这样可以实现仿真处理。而且还能使用户对"老"机器上的程序不需作任何修改就可在"新"机器上运行。

微程序设计的缺点是效率、性能都低于直接用硬件线路实现控制(组合逻辑控制)的计算机,成本也略高些。但其优点还是为广大计算机制造厂家所欢迎。所以目前大多数机器都采用微程序设计技术,尤其是微型机和个人计算机。

微程序并不是在主存中运行,而是在高速控制存储器(简称控存)中运行。60 年代中期把只读存储器(ROM)用作控存,这样 ROM 中的内容由制造厂家确定后就无法改变了,那时 ROM 中的微程序作为计算机的控制器使用。近年来,除 ROM 外还广泛使用可编程的控存(PROM, EPROM),它可通过控制台使用特殊指令将主存中的微程序写入 EPROM 中,这样机器的控制功能和指令系统就更加灵活了。控存一般和主存分开,不少机器将控存做在中央处理机中。IBM 4381 的控存是使用 4K～16K 字的 EPROM。

2.4.2 微程序设计和操作系统

在操作系统中有许多功能是要被经常地调用的。例如在交互作用的事务处理系统中,调度程序(它负责挑选下一个要在 CPU 上运行的作业)可能一秒钟要运行数百次。因此要求其运行效率要高。那么如果把调度程序做成微程序——固件来实现,则比用软件实现在速度上要快得多。除此之外,操作系统中的以下功能也常用微程序来实现:

中断管理。

表格、队列、链等数据结构的管理和维护。

控制对共享数据和资源的同步原语(见第四章)。

多道程序系统中的处理机调度。

过程的调用和返回。

允许进行"位"操作的那些特定的字的管理和维护。

用微程序来执行操作系统的功能不但可以改善系统性能,降低程序开发的成本,而且还能改善系统的保密性。

习　　题

2-1　操作系统的运行环境指什么?

2-2 现代计算机为什么设置目态/管态这两种不同的机器状态？现在的 Intel 80386 设置了四级不同的机器状态(把管态又分为三个特权级)，你能说出自己的理解吗？

2-3 什么叫特权指令？为什么要把指令分为特权指令和非特权指令？

2-4 说明以下各条指令是特权指令还是非特权指令，并说明理由：

(1) 启动磁带机；
(2) 求 π的 n 次幂；
(3) 停止 CPU；
(4) 读时钟；
(5) 清主存；
(6) 屏蔽一切中断；
(7) 修改指令地址寄存器内容。

2-5 CPU 如何判断可否执行当前的特权指令？

2-6 什么是程序状态字？主要包括什么内容？

2-7 存储保护的目的是什么？常用的存储保护机构有哪两种？指出它们的要点。

2-8 针对图 2.3 所示的主存各存储块的情况，请回答以下两种情况对 A，B，C 各块访问是否合法？

(1) 存储保护键的钥为“0000”；

(2) 存储保护键的钥为“0100”。

2-9 存储保护键的取“保护位”是做什么用的？如何起作用？

2-10 什么是双缓冲？详述什么是三缓冲模式的操作。在什么环境下，三缓冲是有效益的？

2-11 CPU 如何发现中断事件？发现中断事件后应做什么工作？

2-12 说明中断屏蔽的作用。

2-13 何谓中断优先级？为什么要对中断事件分级？

2-14 CPU 响应中断时，为什么要交换程序状态字？怎样进行？

2-15 什么是软时钟(虚拟时钟)？有何作用？

2-16 有四个作业 A，B，C，D，要求定时唤醒运行，其要求如下：

A 20 秒后运行，经过 40 秒后再次运行。

B 30 秒后运行。

C 30 秒后运行，经过 25 秒后再次运行。

D 65 秒后运行。

请建立相应的时钟队列。

2-17 列举出提出基地址加位移编址的原因。

2-18 什么叫重定位？有哪几种重定位技术？有何区别？

2-19 本书第 7 章的图 7.10 中，图(a)表示了一个作业的地址空间，该作业被连接装入程序装入主存中，起始地址为 10000(绝对地址)，请表示出该作业装入主存后的情况(存储空间足够作业装入)。

2-20 对比绝对地址装入程序与连接装入程序。

2-21 说明硬件、软件与固件的区别，固件对操作系统的意义何在？

2-22 硬件必须具备哪些条件后，操作系统才可能提供多道程序设计的功能？

第 二 部 分

多道程序设计基础——并行程序设计

第 3 章　进 程 管 理

3.1　进程的概念

3.1.1　进程的引入

进程的概念是操作系统中最基本、最重要的概念。它是在多道程序系统出现后，为了刻划系统内部出现的情况，描述系统内部各作业的活动规律而引进的一个新的概念。由于进程的概念是对程序的抽象，所以不十分直观，需要读者用心加以体会。

多道程序系统的特点首先是并行性。为了充分利用系统资源，在主存中同时存放多道作业运行，所以各作业之间是并行的。引进“中断”，无非也是想解决并行和同步问题。通道和中断技术还使全部外部设备和主机均可并行工作。在主存的所有用户程序；各种中断处理程序；各类设备管理程序；高级调度程序（又称作业调度程序，负责挑选作业进入系统运行，见第 5 章）；低级调度程序（又称进程调度程序，负责挑选就绪进程到处理机上运行）等都可并行运行。这种情况使得系统中各种程序具有的特性之一就是它们的并行性。

其次各程序由于同时存在于主存中。因此它们之间必定会存在着相互依赖、相互制约的关系。例如几个独立运行的用户程序，可能因竞争同一资源（如处理机、外部设备）而相互制约：获得资源者就能继续运行，而未获得者只好等待资源成为可用，通常称此类关系为间接制约关系，因为它是通过中间媒介——资源而发生的关系；另外一种制约关系称之为直接制约关系，这是由于各并行程序间需要相互协同而引起的。例如用户程序要求输入输出时，它就直接受到输入输出程序何时完成其要求的制约。所以制约性是这些程序的第二特性。

再者不论是系统程序还是用户程序，由于它们并行地在系统中运行，并且有着各种相

互制约关系,所以它们在系统内部所处的状态不断地改变。时而在处理机上运行;时而因等待某事件发生(如等待某个中断或等待某资源可用)而无法运行。于是我们说系统中各程序的另一特性是它们在系统中所处的状态不断变化的动态性。

由于在这样一个多道程序系统所带来的更为复杂的环境中,使程序具有了并行、制约和动态的特征。这就使得原来的程序概念已难以刻划和反映系统中的情况了。这是因为,第一:程序本身完全是一个静态的概念(程序是完成某个功能的指令的集合),而系统及其中的各程序实际上是处于不断变化的状态,程序概念反映不了这种动态性。其次程序概念也反映不了系统中的并行特性,例如,假设主存中有两个编译 Fortran 源程序的作业,它们的编译工作可以同时由一个 Fortran 编译程序完成。在这种情况下,如果用程序概念来理解,就会认为主存中只有一个编译程序在运行(被编译的两个源程序只是编译程序加工的数据),而无法说清主存中运行着的两个任务。就是说程序的概念刻划不了这种并行情况。

综上所述,静态的程序概念已不敷使用,需要引进一个新的概念——人们称之为"进程"。

3.1.2 进程的定义

进程(PROCESS)这个名词最早是由 MULTICS 系统于 1960 年提出的。直至目前关于进程的定义及其名称均不统一。在少数系统中把进程称为任务(TASK)。

对进程的定义有如下几种:

(1) 程序在处理机上的执行;

(2) 进程是一个可调度的实体;

(3) 进程是逻辑上的一段程序,它在每一瞬间都含有一个程序控制点指出现在正在执行的指令;

(4) 顺序进程是一个程序及其数据在处理机上顺序地执行时所发生的活动;

(5) 进程是这样的计算部分,它可以与别的进程并行运行。

上述这些描述是从不同的角度对进程所提出的看法。有些是相近的,有些可看作是互相补充。进程这一概念至今虽未形成公认的严格定义。但进程却已广泛而成功地被用于许多系统中。

我国对进程概念的描述,多数认为"进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动"(1978 年全国操作系统会议)。

从所有的对进程的描述来看,进程既是与程序有关而又与程序不同,是对程序进一步的抽象的描述。进程和程序之间一般认为有如下区别:

(1) 进程是程序的执行。故进程属于动态概念。而程序是一组指令的有序集合,是静态的概念;

(2) 进程既然是程序的执行,或者说是"一次运行活动"。因而它是有生命过程的,从投入运行到运行完成,或者说进程有诞生(建立进程)和死亡(撤消进程)。换言之,进程的存在是暂时的,而程序的存在是永久的。

(3) 进程是程序的执行,因此进程的组成应包括程序和数据。除此之外,进程还由记

录进程状态信息的“进程控制块”(见下节)组成;

(4) 一个程序可能对应多个进程。如有多个 Fortran 源程序同时进行编译工作,那么,该编译程序对每个源程序进行的编译,都可看作是该编译程序在不同数据(源程序)上的运行(进行编译——数据加工),所以根据进程定义应是多个不同的进程,这些进程都运行同一个编译程序;

(5) 一个进程可以包含多个程序。因为主程序执行过程中可以调用其它程序,共同组成“一个运行活动”。

3.2 进程的状态和进程控制块

3.2.1 进程的状态及其变化

进程在它存在过程中,由于系统中各进程并行运行及相互制约的结果。使得它们的状态不断发生变化。系统中不同的事件均可引起进程状态的变化。通常一个进程至少可划分为三种基本状态:

(1) 运行状态(记为 Running):当一个进程正在处理机上运行时,则称此进程处于运行状态;

(2) 就绪状态(记为 Ready):一个进程获得了除处理机外的一切所需资源,一旦得到处理机即可运行,则称此进程处于就绪状态;

(3) 等待状态(记为 Blocked,又称阻塞状态)一个进程正在等待某一事件(如等待某资源成为可用,等待输入输出完成或等待其它进程给它发来消息)发生而暂时停止运行。这时即使把处理机分配给该进程也无法运行,则称此进程处于等待状态(或称阻塞状态)。

在不同的系统中,出于调度策略的考虑,把进程的状态进一步细分为更多的状态。

进程在系统中,其状态可以变化。进程各状态间变化示于图 3.1。由图中可看出:

(1) 处于就绪状态的进程被进程调度程序选中后,就分配到处理机来运行。于是进程状态由就绪变为运行;

(2) 处于运行状态的进程在其运行过程中需等待某一事件发生后,才能继续运行。于是该进程由运行状态变为阻塞状态(等待状态);

(3) 处于运行状态的进程在其运行过程中,因分给它的处理机时间量(时间片)已用完而不得不让出处理机,于是进程由运行状态变为就绪状态;

图 3.1 进程状态变化图

(4) 处于阻塞状态的进程,若其等待的事件已经发生。于是进程由阻塞状态变为就绪状态。

3.2.2 进程控制块

进程由三部分组成:程序,数据和进程控制块。

进程控制块(简记为 PCB)是用以记录进程的有关信息的一块主存。它是由系统为每个进程分别建立的。由于进程在系统中的状态及其它情况(如占有的资源等)是不断变化的,所以必须把这些信息记录下来,以便操作系统的进程调度程序对进程进行调度。因此进程控制块的作用主要是:

(1) 记录进程的有关信息:进程控制块 PCB 的具体内容随不同系统而异,通常应包括这样几类信息:

进程的标识信息;

进程的状态信息;

进程的调度信息;

进程的通讯信息;

中断点现场保留区;

进程已占用的资源信息。

下面我们给出 PDP-11/40 上 UNIX 系统中的进程控制块的主要内容以供参考。

① 进程名　进程唯一的内部标识符(有些系统中除给出进程内部名外还给出进程外部名)。

② 父进程名　其父进程的内部标识符。

③ 状态　进程所处状态。

④ 资源信息　进程数据段在主存中的起始地址和大小。

⑤ 优先数　表示进程的重要性和优先程度。

⑥ 调度信息　包括该进程已使用的处理机时间和换入内存后的驻留时间或换到磁盘上的驻留时间。

本系统中的中断现场保存在堆栈中,所以 PCB 中未设置保留区。

(2) 标志进程的存在:操作系统根据系统中是否有该进程的进程控制块 PCB 而知道该进程的存在与否。系统在建立进程的同时就建立该进程的 PCB,在撤消一个进程时也就撤消其 PCB。所以说进程的 PCB 对进程来说是它存在的具体的物理标志和体现。PCB 对操作系统来说,是调度进程的主要的数据基。

PCB 在不同语言中可用不同的数据结构表示。在 PASCAL 中可用"记录"来表示。为了系统管理和控制进程方便,系统常将所有进程的 PCB 存放在主存中系统表格区,并按照进程的内部标号由小到大的顺序存放。而整个系统中的各进程的 PCB 的集合可用数组来表示。这时进程内部标号可与数组元素的下标变量一致起来。各系统预留的 PCB 空间往往是固定的,如 UNIX 系统中规定进程数不超过 50 个。

由此我们可以看出,进程的概念虽然是抽象的,但也可以用它的 PCB 与进程关联的程序和数据,物理地表征一个进程。图 3.2(a)表示与进程相关联的程序和数据集中放在一个内存区中。而图 3.2(b)表示其程序与数据放在不同的内存区中的情形。

3.3 进 程 队 列

系统中有许多进程。它们有的处于就绪状态,有的处于阻塞状态,而且阻塞的原因各不相同。因此为了调度和管理进程方便起见。常将各进程的进程控制块 PCB 用适当方法组织起来。一般来说,大致有以下几种方法:

(1) 把所有不同状态的进程的 PCB 组织在一个表格中,这种方法最为简单,适用于系统中进程数目不多的类型,如 UNIX 系统。其缺点是调度进程时,往往要查找整个 PCB 表;

(2) 分别把有着相同状态的进程的 PCB 组织在同一个表格中。于是分别有就绪进程表;运行进程表(多机系统中);各种等待事件的等待进程表。系统中的一些固定单元分别指出各表的起始地址(图 3.3);

图 3.3 PCB 的表格结构

(3) 分别把具有相同状态的所有进程的 PCB 按优先数排成一个或多个(每个优先级一个)队列。这就分别形成了就绪队列;等待在不同事件上的各等待队列(等待队列一般不按优先级组织,通常按其到达的先后次序排列),如等待打印机的进程队列,等待主存的进程队列等。采用队列形式时,每个进程的 PCB 中要增加一链指针的表目项,以指向队列中的下一个进程的 PCB 起始地址。同表格形式一样,系统要设置固定单元以指出各队列的头——队列中第一个进程 PCB 的起址(图 3.4)。

图 3.4　PCB 的队列结构

3.4　进程的管理

3.4.1　进程的挂起和解除挂起

前面介绍了进程的几种基本状态。但实际上,为了更好地管理和调度进程及适应系统的功能目标。在许多系统(如 Intel 8086 微型机的 IRMX 86 系统)中都有“挂起”和“解除挂起”(简称解挂)一个进程的功能。这是由于:

(1) 系统有时可能出故障或某些功能受到破坏。这时就需要暂时将系统中的进程挂起,以便系统把故障消除后,再把这些进程恢复原来状态。

(2) 用户检查自己作业的中间执行情况和中间结果时,因同预期想法不符而产生怀疑。这时用户要求挂起他的进程,以便进行某些检查和改正。

(3) 系统中有时负荷过重(进程数过多),资源数相对不足,从而造成系统效率下降。此时需要挂起一部分进程以调整系统负荷。等系统中负荷减轻后再将被挂起进程恢复运行。

图 3.5 表示了在具有挂起和解除挂起功能的系统中进程的状态。在此系统中,进程增

图 3.5　具有挂起功能的进程状态变化

加了两个新的状态:挂起就绪(记为 Readys)和挂起等待(记为 Blockeds)。为易于区分起见,把原来的就绪状态称为“活动就绪”(记为 Readya)和“活动等待”(记为 Blockeda)。

此时进程的状态变化如图 3.5 所示。如果一个进程原来处于运行状态或活动就绪状态,此时可因挂起命令而由原来状态变为挂起就绪状态,此时它不能参予争夺处理机,即进程调度程序不会把处于挂起就绪状态的进程挑选来运行。当处于挂起就绪状态的进程接到解除挂起命令后,它就由原状态变为活动就绪状态。如果一个进程原来处于活动阻塞状态,它可因挂起命令而变为挂起等待状态,直到解除挂起命令才能把它重新变为活动等待状态。处于挂起等待状态的进程,其所等待的事件(如正在等待输入输出工作完成,等待别的进程发给它一个消息)在该进程挂起期间并不停止这些事件的进行。因而当这些事件发生后(输入输出完成,消息已发送来了),该进程就由原来挂起阻塞状态变为挂起就绪状态。

挂起命令可由进程自己或其它进程发出,而解除挂起命令只能由其它进程发出。

3.4.2 进程的控制原语

为了对系统中的进程进行有效的管理,通常系统都提供了若干基本的操作,这些操作通常被称为原语。常用的进程控制原语有:

建立一个进程原语;

撤消一个进程原语;

挂起一个进程原语;

解除挂起进程原语;

改变优先数原语;

阻塞一个进程原语;

唤醒一个进程原语;

调度进程运行原语。

3.4.2.1 建立进程原语

一个进程如果需要时,它可以建立一个新的进程。被建立的进程称为子进程,而建立者进程称为父进程。所有的进程只能由父进程建立,不是自生自灭的。所以系统中有所谓“祖先”进程。从而系统中的所有进程就形成了进程间的层次(家族)体系——进程树或称进程族系。

各系统的建立进程原语就是供进程调用以建立子进程使用的。该原语的主要工作是为被建立进程建立起一个进程控制块 PCB,并填入相应的初始值。其主要操作过程是先向系统的 PCB 空间申请分给一个空闲的 PCB,而后根据父进程所提供的参数,将子进程的 PCB 表目初始化,最后返回一个进程内部名。通常父进程调用该原语时应提供以下参数:进程名(外部标识符)n;处理机的初始状态(或进程运行现场的初始值,主要指各寄存器和程序状态字初始值)S_0;优先数 k_0;父进程分给子进程的初始主存区 M_0 和其它资源清单(多种资源表)R_0 等。建立进程原语的工作大致描述为:

```
procedure Create (n, S0, k0, M0, R0)
begin
```

```
            /*  请求分配 PCB 空间
①           i:  = Get Internal Name(n);
            /*  初始化 PCB
②           Id(i):  = n;
③           Priority(i):  = k0;
④           Cpustate(i):  = S0;
⑤           Main Store(i):  = M0;
⑥           Resources(i):  = R0;
⑦           Status(i):  = `Readys'
⑧           Parent(i):  = CALLER
            /* 插入就绪队列
⑨           Insert(RL,i);
  end
```

程序中的第①语句是调用查找进程名过程“Get Internal Name”,参数为进程外部名 n。该过程查找 PCB 集合,如已有此同样外部名进程则返回出错消息,否则返回一个空闲的 PCB 内部标识号 i。第②语句是把进程外部名 n 登记到第 i 个 PCB 的相应外部名表目中。语句③是往 PCB 中登记优先数。语句④登记现场状态初始值 S_0 到相应的现场保留区中或 Cpustate 中。⑤,⑥分别记入主存和资源的初始占有情况,这是由父进程将自己的一部分资源分给子进程的。⑦是把进程初始状态置为“挂起就绪”。语句⑧中 CALLER 代表调用本过程的父进程之内部标识号,将它记入子进程 PCB 的父进程名这一栏。语句⑨也是调用插入过程 Insert,其中 RL 表示就绪队列,即把进程 i 插入就绪队列。

在 UNIX 系统中,该原语叫 Newproc。在有些高级语言中(如 PL/1)就有建立进程的语句(叫 TASK,属于 CALL 语句)供用户使用,在并行 PASCAL,MODUL-2 等语言中有更强的功能语句。

3.4.2.2　挂起进程原语

执行挂起命令的功能模块是挂起原语,调用挂起原语的进程只能挂起它自己或它的子孙。而不能挂起别的族系的进程,否则就认为命令非法。挂起命令的执行可以有多种方法:

(1) 把发本命令的进程挂起;

(2) 把具有指定标识符的进程挂起;

(3) 把某进程及其全部或部分(例如具有指定优先数的)子孙进程一起挂起。

这里介绍挂起指定标识符 n 的进程的操作过程,每调用一次挂起命令仅允许挂起一个进程。其命令的过程描述如下:

```
Procedure Suspend(n);
    begin
        i:  = Search Internal Name(n);
        CASE Status (i) of
         'Blockeda':  Status(i):  =  'Blockeds';
```

```
        'Readya': Status(i):  =  'Readys';
        'Running': begin STOP(i);
                              SCHEDULER
                   end
      end CASE
end
```

程序中第 1 语句是调用查找内部名过程以找出外部名为 n 的进程的内部名 i。然后按进程状态分别处理。当其状态为“运行”时，调用停止该进程的过程 STOP。并调用调度程序 SCHEDULER 以选一就绪进程到处理机(被挂起的运行进程释放出)上运行(毫无疑问，此程序适合于多机系统)。

3.4.2.3　解除挂起原语

调用解除挂起原语来恢复一个进程的活动状态是比较简单的。一个进程只能将自己的子孙进程解挂，而不能解挂别的族系进程。一个进程可以将自己挂起，却不能将自己解挂(为什么?)。现将解挂过程描述如下:

```
procedure Resume (n);
  begin
    i:  = Search Internal Name(n);
    if Status(i) = 'Readys' then
      begin Status(i):  =  'Readya';
      end    SCHEDULER
      else Status(i):  =  'Blockeda'
  end
```

程序中的 if 语句中指出，假若解挂后的状态是“活动就绪”，则由于此进程挂起了很长时间，其优先数可能改变，所以这时调用处理机调度程序来为高优先数进程抢占一个低优先数进程占用着的处理机。

3.4.2.4　撤消进程原语

当一个进程在完成其任务后，应将该进程撤消，以便及时释放出它所占用的资源。同挂起操作情况相同，撤消原语也可采取两种策略: 一是只撤消一个具有指定标识符的进程(其子进程);或者撤消它的一个子进程及该子进程的所有子孙进程。若采取前一种策略，进程树的层次关系就很容易切开、分离。这会导致在系统中留下一些孤立隔绝的进程，从而失去对它们行为的控制。因此我们采取第二种策略。被撤消进程的所有系统资源(主存、外设)全部释放出来归还系统。该原语的参数是被撤消进程的外部名。撤消原语一般由其父进程或祖先发出，不会自己撤消自己。考虑到被撤消进程可能正在某处理机上运行，因此撤消时还应调用处理机调度程序 SCHEDULER 以将处理机分给其它进程。当被撤消进程不是运行状态时，就不需要调用处理机调度程序。为标志这两种情况，在撤消原语中设置了一个标志位 Sched。若标志位的值为真，则调用处理机调度程序。下面给出该原语的大致描述:

```
procedure Destroy (n);
```

```
    begin
      Sched:  = false;
      i:  = Search Internal Name(n);
        KILL (i);
      if Sched= true then SCHEDULER;
    end(Destroy)
  procedure KILL(i);
    begin
      if Status(i)= 'Running 'then
      begin
        STOP(i);
        Sched:  = true
        end
      REMOVE(Queue(i),i);
      for all P∈progeny (i) do
        KILL(P);
      for all r∈Resources(i) do
        RELEASE (r);
        RELEASE (PCB(i));
    end (KILL)
```

在撤消进程原语中要调用 KILL 过程。KILL 过程所要求的参数是进程内部名 i。当进程 i 状态是运行,则调用 STOP 过程停止该进程。REMOVE(Queue(i),i)语句是调用从队列中移走一个进程的过程,这里是从进程 i 所在的队列中移走进程 i。然后递归调用 KILL 过程以杀死所有属于 i 的子进程 s,释放出进程 i 所有占有的资源。过程 RELEASE 是释放资源过程。也用该过程来释放进程 i 的 PCB 空间。

3.4.2.5 改变进程优先数原语

进程的优先数是表示进程的重要性及运行的优先性,供进程调度程序(在多机系统中为处理机调度程序)调度进程运行时使用。为了防止一些进程因优先数较低,而长期得不到运行(有时一个作业因优先数过低,以至于长期未能轮上运行。操作员甚至怀疑该作业丢失了)的情况。许多系统采用动态优先数,即进程的优先数不是固定不变的,而是按一定原则变化的。通常进程的优先数与以下因素有关:

(1) 作业开始时的静态优先数。作业的优先数取决于作业的重要程度;用户为作业运行所付出的价格和费用大小;作业的类型(联机作业的优先数大于脱机批处理作业的优先数)等因素;

(2) 进程的类型。一般系统进程的优先数大于用户进程的优先数;输入输出型进程(指主要是数据处理型的进程,输入输出量大而计算工作量小)的优先数大于 CPU 型的进程(指主要工作是在 CPU 上计算,输入输出工作量少的进程),这是为了充分发挥系统输入输出设备的效能;

(3) 进程所使用的资源量(CPU 机时、主存和其它资源)。随着使用 CPU 时间愈多,其优先数愈来愈低。对其它资源使用的情况的考虑也类似,但往往同系统中资源的配置及其使用的紧张情况有关。

(4) 进程在系统中等待时间。等待时间愈长,优先数就愈提高。

各系统出于不同考虑,有不同的优先数计算公式。这些公式主要来自于实践经验。以 UNIX 系统为例,其优先数的最小值为- 100,最大值为 127。其值越小,优先级越高。用户进程的优先数总是大于等于 100。对优先数大于等于 100 的进程,系统为其每秒重算一次优先数。计算公式如下:

$$P_{ri} = \mathrm{MIN}\ \left(127, 100 + \frac{P_{CPU}}{16} + P_{nice}\right)$$

式中, P_{CPU} ,对当前运行进程每 20 毫秒加 1,直至 255 为止。而对其它进程每秒减 10,直至小于 10 为止。所以这是一个与进程运行时间和等待时间有关的量;

P_{nice},这是一个与进程本身情况有关的参数,通常为正。用户是通过系统调用 nice 设置的。

由公式可以看出,一个进程占用 CPU 时间增加,它的优先数下降。而长期不被理睬的进程,其优先级将会相对提高。

改变某进程 n 的优先数的原语大致可描述如下:

```
procedure Change Priority(n);
  begin
    i:  = Search Internal Name (n);
    Pri(i):  = Calculate Priority(i);
    if Status(i) = 'Readya 'then
      begin
          Insert (RL, i, Pri);
            SCHEDULER
        end
    end
```

程序首先根据进程外部名 n 找出其内部名 i,然后调用计算优先数公式算出优先数并登记到进程 i 的 PCB 中。当该进程状态为活动就绪,则一方面将进程 i 按其优先数插入就绪队列的适当位置,并调用 SCHEDULER,以决定是否可能抢占某个处理机使用。

进程的其它控制原语(阻塞进程原语、唤醒进程原语和调度一个进程原语)将在以后有关章节中介绍。

3.5 Windows NT 中的线程

在单用户多任务的计算机中,如同在 Windows NT 中那样,除进程外,引入了一个新概念——线程(thread)。对象、线程和进程三者是构成 Windows NT 操作系统基本元成分。线程概念在微型机多任务系统中十分重要。

什么是线程?线程被定义为“进程内的一个执行单元”或“进程内的一个可调度实体”。一个线程有四个基本组成部分:

(1) 一个唯一的标识符,称为客户 ID;

(2) 一组处理器状态寄存器;

(3) 分别在用户态和核心态下使用的两个栈;

(4) 一个私用存储器。

为什么要引入线程概念?其主要目的在于加强并行性,以适应多任务情况下提高处理速度和系统性能的要求。除此之外,引入多线程机制也还带来其它好处,详细内容请参阅 12.5.3 节。

在具有多线程机制的操作系统中有以下特点:

(1) 参预竞争处理器的基本调度单位不是进程而是线程;

(2) 负责挑选线程到处理器上执行的调度程序称为线程调度程序,它是内核中的主要成分,调度线程是内核的主要功能之一;

(3) 一个线程可以创建它所需的线程;

(4) 一个进程至少要有一个可执行线程,进程可以有多个线程;

(5) 一个线程在它的生命期内可以处于六种不同的状态(参阅 12.6.1 节):

就绪状态;　　等待状态;

备用状态;　　转换状态;

运行状态;　　终止状态;

(6) 线程与进程一样,均是以对象来实现的;

(7) 用线程来实现并行性比用进程来实现并行性更方便更有效。

习　　题

3-1　为什么要引入进程概念?进程的基本特征是什么?它与程序有何区别?

3-2　定义以下术语:程序、过程、处理机、进程、用户、任务和作业。

3-3　为什么说 PCB 是进程存在的唯一标志?

3-4　建立进程的实质是什么?撤消进程原语完成哪些工作?

3-5　为什么要为用户应用程序和系统程序设置进程?你认为设置进程有哪些利弊?

3-6　为什么把阻塞队列中的进程按优先数排序是没有意义的?然而在什么情况下,这种做法是有用的?

3-7　在某些系统中,派生出来的进程在其父进程被撤消时也自动撤消;在另一些系统中,派生出来的进程独立于父进程工作,父进程撤消时子进程并不随之撤消,讨论二种方法的优缺点。

3-8　下述哪些情况是对的?

(1) 进程由自己创建;

(2) 进程由于自己阻塞;

(3) 进程由于自己挂起;

(4) 进程由于自己解除挂起;

(5) 进程由于自己唤醒;

(6) 进程由自己撤消。

3-9 动态优先数有哪些优点? 通常优先数与哪些因素有关?

3-10 试列举出进程状态转换的典型原因,详细列出引起进程调度的因素。

3-11 什么是线程? Windows NT 为什么要引入线程的概念? 有何好处?

3-12 Windows NT 的进程概念与传统操作系统的进程概念有何不同?

3-13 在 Windows NT 中为什么每个进程必须(至少)要有一个线程?

3-14 线程使用什么调度算法? 你能分析它为什么采用这种算法吗?

第 4 章　多道程序设计基础
——并行程序设计

早期的计算机是单用户的。现代计算机系统为了提高其资源利用率和计算机系统的效率,广泛地采用了支持多用户或多任务的环境。即主存中有多个用户或任务的作业(进程)在同时运行。由前章可知,这些进程在系统中的状态是动态地变化的;它们之间共享着系统资源;有着各种不同的制约关系;因而它们有时互相竞争系统资源,有时几个进程为了协同地完成一个共同的任务,需要不断交换各自工作的进展情况,向其它进程提出要求或回答,即进程之间需要不断地通信。总而言之,在多道环境下,系统具有了许多比单道环境下更为复杂的情况。多道环境下的程序设计也具有了许多单道环境下程序设计中所不存在的特点和问题。我们把多道环境下的程序设计叫做并行程序设计,因为它主要是以各程序(进程)间的并行运行为其特点。而把传统的程序设计方法叫做顺序程序设计。

4.1　顺序程序设计和并行程序设计概念

4.1.1　顺序程序设计的特点

大家所习惯的传统的程序设计方法就是顺序程序设计的方法。操作系统理论的出发点就是基于"顺序处理"——相继地一次处理一个事件。而顺序处理也是计算机的工作方法:处理机逐条地一次只执行一条指令;主存储块只能一次访问一个字或字节;外设一次只能传送一个数据块。顺序处理也是人们思考的方法,人们为了解决一个复杂的问题而把它分解成一些较为简单、易于分析的小问题、逐个地分析解决。一个复杂的程序也可以分为若干个程序段,依照某种次序逐个来执行。所以我们首先研究一下顺序程序处理执行的模式及其特点。

任何一个计算任务,总是首先要把程序和数据输入计算机,然后才能由计算机进行处理,最后把结果用适当的方式输出这样三个过程。在顺序处理过程中,计算机完成一个作业再执行下一个作业,每个作业全都经过这样三个过程。因此,如果用结点来代表各种操作(其中 I 代表输入操作,C 代表计算操作,O 代表输出操作),则计算机的顺序处理可以表示成如图 4.1 所示的模式。

分析一下计算机的顺序处理情况不难发现它具有以下特点:

图 4.1　顺序处理模式

(1) 当顺序程序在处理机上执行时,处理机的操作严格按照程序所规定的顺序执行,即每个操作必须在下一操作开始之前结束,这叫程序的顺序性。

(2) 程序运行时都处于一定的环境之中。所谓环境是指处理机能直接感知和加以改变的那一部分外界,如该程序使用的主存单元、累加器、各寄存器、指令计数器或程序状态字等。在顺序处理情况下,由于运行程序独占系统全部资源,所以在程序执行过程中,其所处的环境都是由程序本身确定(除初始环境外),即只有程序本身的动作才能改变其环境,不受任何其它程序等外界因素的影响。这叫程序环境的封闭性。

(3) 程序的两个动作之间允许暂时的间歇,这种间歇对程序的最终结果没有影响。也就是说程序执行的结果与它的执行速度无关。这叫程序结果的确定性,或其结果与执行速度、时间的无关性。

(4) 程序执行时,只要初始条件相同,则所得最终结果相同,因此可以用相同初始条件再重现前一次的计算情况,这叫计算的可再现性。这样当程序中有错误时,为了排除错误,往往可以重现错误,以便进行分析。

以上四个特性是顺序处理所特有的性质。在并行处理环境下,这些特性就受到了破坏,并因此具有了不同的性质。

4.1.2　并行程序设计

4.1.2.1　并行程序设计的概念

现代计算机为了提高计算机的运行速度和系统处理能力,在总体设计和逻辑设计中广泛采用并行操作技术,使多种硬件设备能并行工作。例如由于通道和中断技术的引入,通道能独立地控制外部设备操作,从而使 CPU 与通道以及与外部设备间均可以并行地工作。这是对硬件而言。对软件来说,由于多道程序的操作系统支持,不但在多机系统中它可同时执行多个不同的程序段(进程),即使在单机系统中,无论从逻辑上或是从宏观上来看,系统中各进程及其所对应的程序也是并行运行的。这里所说的并行运行是指这些程序段(如图 4.2 中的程序段 A, B, C, D)的运行在时间上是重叠的,并不是说这些程序在某一时刻都在处理机上运行。这在单机系统中是无法办到的。

多道程序系统中往往有多个作业,不但作业之间可以并行运行。而且并行运行也可存在于一个作业的各计算部分之间,因为对于多数作业来说,各操作之间往往只要求部分有序,即有些部分要求顺序地串行执行,有些部分则可以并行地执行。如要求计算机将两个矩阵求逆后相加,那么两个矩阵的求逆操作可以分别在一台或两台处理机上并行运行,而加法操作则必须在求逆操作之后串行地进行。所以有些用户为了提高其作业的处理速度而明白地指出其作业中的串行和并行部分。

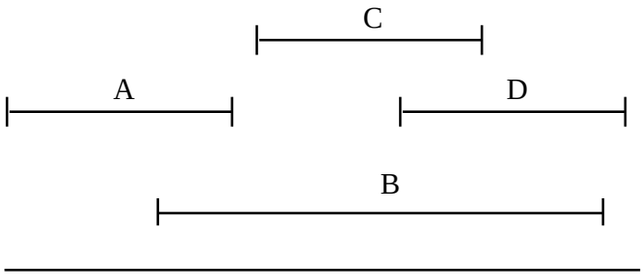


图 4.2　程序段的并行运行

在多道环境中,并行性不但存在于硬件之中,而且也存在于进程或程序之间。这样的并行情况就不能不给程序设计带来巨大的影响,带来许多不同于顺序程序设计中的问题,考虑各种并行性的程序设计方法称为并行程序设计。

4.1.2.2　程序并行性的表示

在一个程序的诸操作间,往往只要求部分有序,既有并行执行部分,又有串行执行部分。本节主要讨论程序中的诸操作间的串行、并行关系的图形和语言结构中的表示法。

进程中诸操作间的关系可用有向图表示为三类(图 4.3):

图 4.3　进程执行的先后次序关系

图中 S 表示系统启动运行,F 表示结束,P_i(i= 1,2…)代表各进程。

当给出一个算术表达式,我们可用这种图形清晰地显示出其并行、串行关系。如

$$(a-b)\times(c-d)+(e/f)**2$$

的串行、并行关系示于图 4.4。

算术表达式树的深度决定了其并行的程度。在矩阵运算和分类、合并中也可用并行运算来提高其处理速度。

程序并行性的语言表示已出现在一些并行语言中。这些语言有:

(1) 并行 PASCAL 语言:这是由丹麦计算机科学家 Brinch Hansen 于 1975 年对 PASCAL 语言所作的一种扩充。他建立并行 PASCAL 语言以便更适用来写操作系统。

(2) CSP/K 语言:这是由多伦多大学的计算机科学家 Holt 于 1978 年提出,它是对 PL/1 语言的扩充。

(3) MODULA 语言:这是由 Wirth 在 PASCAL 基础上提出的并行程序设计语言。

美国国防部对 Ada 语言进行扩充,以具有表示并行性的能力。

在 Hansen 正式提出并行 PASCAL 语言之前,著名的计算机科学家 Dijkstra 于 1965 年就提出了并行语句记号,并被 Hansen, Holt 等人所采用。其语言记号是:

COBEGIN

$S_1; S_2; \cdots; S_n$

COEND

COBEGIN/COEND 相当于一个括号,表示其中的所有语句 $S_1, S_2, \cdots, S_n$ 是可并行执行的。COBEGIN/COEND 可以嵌套。而语句 $S_1, S_2, \cdots, S_n$ 可为任何简单语句、复合语句和并行语句。编译时,编译程序将为每个并行语句设置一个进程。作为练习,读者可将图 4.4 用并行语言记号表示。

图 4.4　算术表达式求值

4.1.2.3　并行程序设计的特点

多道程序系统中广为存在的并行性不能不给程序设计带来巨大影响,带来许多新的问题。本节讨论并行程序设计究竟带来了什么新问题。

并行程序设计的最根本特点首先是并行性,这在前面已讨论了。由并行性必然要求有第二特点:共享性。在系统中存在的各进程不但共享硬资源:主存、CPU、外设。而且共享软资源:程序副本和数据集。

并行性和共享性是并行程序设计的根本特点。为了提高系统中各种资源的使用效率和对作业的处理能力。我们希望尽可能地提高系统的并行性和共享性。但另一方面,也正是由于并行性和共享性给操作系统带来了十分巨大的复杂性和许多新的问题。因此,使得并行程序设计具有不同于顺序程序设计的下述特点:

(1) 顺序运行的模式被打破,各进程可并行地、异步地在系统内运行,并以不同速度向前推进。

(2) 顺序程序设计中的程序环境封闭性被打破。这是由于系统中同时存在多个并行运行的进程,它们均可以异步地访问它所想要访问的共享变量。也就是说,某个进程不再能独占地访问主存单元、累加器、各寄存器、指令计数器或程序状态字等资源了,环境不再由单个程序来确定,而且程序(或进程)本身还将受到这一可以被其它进程改变的环境的

影响。

(3) 如果不能遵循并行程序设计的原则(即同步与互斥)正确地进行并行程序设计,那么程序运行的结果将是不确定的,而且错误现场是难以再现的,或者说将产生与进程进展速度或与时间有关的错误。从下面 4.2 节中读者可仔细体会和考察以上所说的特点。

4.2 进程间的同步与互斥

在多道程序系统中,有许多进程在系统中并行运行,这些进程间存在着不同的相互制约关系,这些关系可以归结为两种关系:同步关系与互斥关系。

所谓两个事件之间的"同步"是指两个事件的发生有着某种时序上的关系。而进程间的同步关系是指系统中往往有几个进程共同完成一个任务,因此它们之间必须协同动作,互相配合,甚至需要交换信息——进行进程间的通信。例如前面提到的两个矩阵求逆后相加的作业,主进程是加法进程,两个子进程分别是两个矩阵的求逆进程。这三个进程间要同步,就是说它们有一定的先后执行次序。只有求逆进程完成后,加法进程才能完成加法工作。但是系统中各进程是相对独立的,异步运行的。由于系统中情况非常复杂,因而各进程的进展速度是无法预知的(如进程的时间片可能被抢占,可能因等待事件而放弃等)。因此如何协调各进程之间动作以达到同步,这就是下一节要研究的问题——同步执行的工具。

互斥关系是进程间的另外一种关系。由于各进程要共享资源(包括硬资源和数据集)。而有些资源往往要求排它性地使用,或者说是互斥地使用。因此进程间往往要互相竞争,以使用这些互斥的资源。

互斥也可看作是一种特殊的同步关系。下面首先讨论有关互斥的问题。

4.2.1 临界段问题

4.2.1.1 问题的提出

考虑有两个进程 P_1 和 P_2,二者异步地增加代表某资源数量的一个公共变量 x。其工作过程如下:

P_1: …; x: = x+ 1; …

P_2: …; x: = x+ 1; …

设 C_1 和 C_2 是共享主存的双机系统中的两个处理机,它们分别有内部通用寄存器 R_1 和 R_2。假定 P_1 正在 C_1 上执行,而 P_2 正在 C_2 上执行,那么下面两个执行顺序中的任何一个,在整个时间过程内都有可能发生:

(1) P_1: …, R_1: = x; R_1: = R_1+ 1; x: = R_1; …

P_2: …, R_2: = x; R_2: = R_2+ 1; x: = R_2; …

t_0——————————————→t

(2) P_1: …, R_1: = x; R_1: = R_1+ 1; x: = R_1; …

P_2: …, R_2: = x; R_2: = R_2+ 1; x: = R_2; …

(赋值语句 x: = x+ 1 表示成机器内部动作由三部分组成: 将变量取到寄存器; 寄存器增 1; 将寄存器中内容送回变量中) 设 x 在时间 t_0 时有值为 v, 若按顺序 1 执行, 则 P_1, P_2 二进程执行后 x 的结果值为 v+ 1。若按顺序 2 执行, 则 x 的结果值为 v+ 2(若 P_1, P_2 分时使用同一个处理机时, 当 P_1 执行完 R_1: = x 后, 由于时间片到而被中断。由 P_2 执行, 也会得到同样结果)。显然顺序 2 的结果是正确的, 而顺序 1 的结果是错误的。如果 P_1 和 P_2 是航空订票系统中的两个售票进程, x 代表某班机售出的座位数, 那么这个系统将会给旅客和经理造成不愉快的后果。类似的情况在操作系统中是很多的, 问题在于 X 是一个互斥性使用的数据, 所以正确的解决办法是一次只能容许一个进程对共享变量进行写操作, 就是要互斥地对共享变量进行访问。人们把进程中访问共享变量的代码段称为临界段。因此上述的'R

: = x; R: = R+ 1; x: = R'就是进程 P_1 和 P_2 关于共享变量 x 的临界段(关于同一共享变量的各进程的临界段的代码可以不同, 本例中是相同的)。

由于关于同一变量的各临界段是分散在各有关进程的程序中, 而各进程是异步工作的, 其进展速度是不可预知的。那么怎样才能保证诸进程间互斥地执行临界段以访问共享变量呢。为此著名的计算机科学家 Dijkstra 于 1965 年提出了临界段设计原则如下:

(1) 每次至多只允许一个进程处于临界段之中;

(2) 若有多个进程同时要求进入它们的临界段时, 应在有限的时间内让其中之一进入临界段, 而不应相互阻塞, 以致于各进程都进不去临界段;

(3) 进程只应在临界段内逗留有限时间。

* 4. 2. 1. 2 软件解决办法

首先通过一个例子来研究临界段互斥执行的软件解决办法。

有两个并行进程 P_0 和 P_1, 互斥地共享单个资源(如磁带机或某共享数据)。P_0 和 P_1 是一个循环进程(指进程执行一个无限循环程序), 每次只使用资源为一个有限的时间间隔。

对此问题, 历史上曾有很多人提出过不同的软件解法:

解法一 用标志位 flag[i] 来标示进程 i 是否在临界段中执行, 当 flag[i]为真, 则表示它正在执行临界段。反之不在临界段中执行。故进程 P_i 的程序结构可描述如下:

```
var flag: array[0··1]of boolean;
Pi: begin
      repeat
      While flag[ j ] do skip;
          flag[i]: = true
      进程Pi 的临界段代码 CSi;
          flag[i]: = false;
      进程 Pi 的其它代码;
      forever
    end
```

此解法存在的问题是,当两个进程的标志位最初都为 false,这时刚好两个进程同时都想进入临界段,并且同时发现对方标志位为 false,于是两个进程同时进入了各自的临界段,这就违背了临界段设计原则 1,每次至多只允许一个进程进入临界段。

解法二　用一个指针 turn 来指示应该那个进程进入临界段。若 turn= i,则该进程 P_i 进入临界段。故进程 P_i 的程序结构如下:

```
Pi: begin
      repeat
        While turn≠i do skip;
      进程Pi 的临界段代码 CSi;
        turn: = j;
      进程 Pi 的其它代码;
      foreven
    end
```

此方法的问题在于强制两个进程以交替次序进入临界段,如果 P_i 进程的处理机速度要快得多,或者进程 P_j 的其它代码部分很长,那么进程 P_i 在进程 P_j 尚未进入临界段前要求再次进入临界段时,尽管此时临界段为空(无进程处于临界段之中),进程 P_i 也无法进入临界段,从而使临界资源的使用率不高。

解法三　此方法还是用 flag[i] 来表示进程 i 想要进入临界段,这是基于对解法一存在的问题:未表明自己想进入临界段的意向,而只检测对方是否已在临界段中,从而导致同时进入。为此改为如下的结构:

```
Pi: begin
      repeat
        flag[i]: = true;
        While flag[ j ] do skip;
    进程Pi 的临界段代码 CSi;
        flag[i]: = false;
    进程 Pi 的其它代码;
        forever
      end
```

但是此解法仍然存在问题,这就是两个进程同时想进入临界段,于是把各自的标志位都置为 true,并且同时检测对方的状态,发现对方也想进入(或已在临界段中),于是互相“谦让”而阻塞了自己,谁也进不了临界段。从而违反了临界段设计原则 2。

解法四　提出以上几个错误的解法,其目的在于揭示解决互斥问题中的困难及其不不断完善而至正确的过程。下面的解法是由荷兰数学家 Dekker 所提出的。

此方法是为每个进程设置一个标志位 flag[i],同样,当该位为真时,则表示该进程 P_i 要求进入临界段(或已在临界段中执行),反之为假。另外还设置了一个指针 turn 以指出应该由哪一个进程进入临界段;若 turn= i 则应该由进程 P_i 进入临界段。其程序如下:

```
var flag: array[0..1]of boolean;
        turn: 0..1;
```

初值为

flag[0]:=flag[1]:=false;

turn 的初值可任意。

于是进程 P_i 的程序结构为

```
Pi: begin
      repeat
        flag[i]:=true;
        while flag[j]
              do if turn=j
               then begin
                      flag[i]:=false;
                    while turn=j do skip;
                      flag[i]:=true;
                    end
        临界段代码 CSi
        turn:=j;
        flag[i]:=false;
        进程 i 的其它代码
        forever
        end
```

解法五　这是把 Dekker 算法扩充到 n 个进程共享一个公用变量的普遍情况的解法

```
var flag: array[0..n-1]of(idle,want-in,
                          in-cs);
    turn: 0..n-1;
```

变量 flag 数组各元素的初值均置为 false，而 turn 可在 0 到 n-1 中任意选一个值。于是进程 P_i 的程序结构如下：

```
Pi: begin
    var j: lnteger;
  repeat
      repeat
        flag[i]:=want-in;
        while turn≠i do if flag[turn]=idle
              then turn:=i;
        flag[i]:=in-cs;
        j:=0;
        while (j<n)and(j=i or flag[j]≠in-cs)
```

```
            do j:= j+1;
          until j≥n;
      进程 Pi 的临界段代码 CSi;
          flag[i]:= idle;
      进程 i 的其它代码;
      forever
    end
```

可以证明这个解法如同 Dekker 解法都是正确的。首先它保证了进程互斥地进入临界段,因为进程 P_i 进入临界段,仅当

flag[j]≠in-cs(对所有 j≠i 都成立)

因此只有 P_i 的 flag[i] = in-cs,也就是说只有 P_i 在临界段内。其次这个解法也不会造成进程间互相阻塞,这是因为:

(1) 任何进程 P_i 只要正在执行进入临界段代码(粗线框内),它的状态就不会是 idle,即

flag[i]≠idle

(2) 若某个进程 P_i 的状态是 flag[i] = in-cs 时,这并不意味着 turn= i。因为几个进程可能同时进入第一个 while 语句,并且都发现 flag[turn] = idle,于是这些进程都相继把 turn 指针改为指向自己;

(3) 一旦第一个进程首先把 turn 指针改为指向它自己后,在那些尚未查询过"flag[turn] = idle 是否成立(即 if 语句)"的进程中再也不可能有新的进程把 turn 指针指向它自己(即不可能进入 then 语句);

(4) 若{$P_1, P_2, \cdots, P_m$}是同时进入第一个 while 语句在最早的那一批进程的集合,由于它们都发现 flag[turn] = idle,从而把指针都指向了自己,然后还把自己的状态改成 flag[i] = in-cs。但 turn 指针最终是指向最后一个改变 turn 指针值的进程 P_k($1 \le k \le m$)。对于所有属于这一集合的进程 $P_i \in \{P_1, P_2, \cdots, P_m\}$由于在执行第二个 while 语句时,发现除自己外还有别的进程的状态也是 flag[j]= in-cs,而被迫返回第一个 while 语句执行,于是只有 P_k 才能顺利的进入临界段,所以不会互相阻塞,并且是在有限的时间内就进入了临界段。

这解法的缺点是有些进程有可能被无限期的推迟进入临界段。下面的解法是上述解法的改进:

```
Pi: begin
    repeat
        repeat
          flag[i]:= want-in;
          j:= turn;
          while j≠i do if flag[ j ]≠idle
                       then j:= turn
                       else j:= j+1 mod n;
```

```
            flag[i]:= in-cs;
            j:= 0;
            while(j< n)and(j= i or flag[j]≠in-cs)
            do j:= j+ 1;
until (j≥n) and (turn= j or flag[turn]= idle);
turn:= i
```

进程 P_i 的临界段代码 CS_i;

```
j:= turn+ 1 mod n;
while(j≠turn)and(flag[ j ]= idle)
    do j:= j+ 1 mod n;
  turn:= j;
  flag[i]:= idle;
```

进程 P_i 的其它代码;

```
    forever
    end
```

不难看出,软件解决办法显得复杂而又不好理解,下面介绍临界段问题的硬件解决办法。

4.2.2 同步与互斥的执行工具

用软件方法来解决进程间的同步和互斥是有较大的局限性,并不是很理想的。为此在本节中提出若干以硬件技术为主的其它的解决办法,这些方法简单实用,使用广泛。

4.2.2.1 硬件指令

许多大型机(如 IBM 370 等)和微型计算机(如 Intel 8088, Z-8000, M68000 等)中都提供了专门的硬件指令,这些指令都允许对一个字中的内容进行检测和修正,或交换两个字的内容。特别要指出的是,这些操作都是在一个存储周期中完成,或者说是由一条指令来完成的。用这些指令就可以解决临界段问题了。因为临界段问题在多道环境中之所以存在,是由于多个进程共同访问、修改同一个公用变量。在单机系统中,由于中断的原因,使得一个进程在对一个公用变量先取来并检测其值,然后修改的这两个动作(通常要由 2～3 条指令来完成)中,可以插入其它进程对此公用变量的访问和修改,从而破坏了此公用变量数据的完整性和正确性。在多机系统中,该公用变量数据的完整性和正确性的破坏,并不是受中断的影响,而是多个处理机共享主存,因而使得某处理机可以插入另一处理机的两个存储访问周期之间,访问并修改此共享变量(要记住的是,中断的插入只能在两条指令之间,而不能在一条指令的执行过程中。因为在指令周期最后时刻 CPU 才扫瞄中断寄存器。而对于同一主存块的访问要求,即使两个处理机同时提出,存储控制逻辑也只能让其中之一先访问,即两个存储周期不会同时进行,但在一个处理机的两个存储周期间则可以插入另一处理机的存储周期)。现在我们用一条指令来完成检测和修改两个功能,这样中断和插入另一处理机的存储周期均不可能,所以不会影响此公用变量数据的完整性。

实现这种功能的硬件指令有两种:

(1) TS(Test-and-Set)指令。又称检测和设置指令,该指令的功能可用 PASCAL 语言描述如下:

```
function Test-and-Set(var flag: boolean): boolean
```

```
begin
  Test-and-Set:= flag;
    flag:= true;
  end
```

这条指令在微型计算机 Z-8000 中称为 TEST 指令，在 IBM 370 中称为 TS 指令。

(2) Swap 指令。又称交换指令，该指令的功能是交换两个字的内容，它可用 PASCAL 语言描述如下：

```
procedure Swap(var a, b: boolean)
    var temp: boolean;
    begin
      temp:= a;
        a:= b;
      b:= temp;
    end
```

在微型计算机 Intel 8086 或 8088 中，该指令称为 XCHG 指令。

用这些硬件指令可以简单而有效地实现互斥。其方法是为每个临界段或其它互斥资源设置一个布尔变量，例如称为 lock，当其值为 false 则临界段未被使用，反之则说明正有进程在临界段中执行。于是某进程用 TS 指令实现互斥的程序结构为(设为无限循环进程)：

```
repeat
    while TS (lock) do Skip;
      进程的临界段代码 CS;
      lock:= false;
      进程的其它代码;
forever
```

或者用 Swap 指令来实现互斥[①]，则

```
repeat
    key:= true;
    repeat
        Swap(lock, key);
```

① 此互斥功能在 Z-8000 中的汇编语言形式如下：

```
        Check TSET flag        检测 flag
              JR NE, Check     非零转移
              临界段 CS          (PSW 中 Z= 0)
              CLR flag         清 flag 位
```

在 Intel 8086 中的汇编语言形式如下：

```
               MOV AL 0        0→AL 寄存器
        Again: XCHG AL, flag       交换 AL 与 flag 内容
               TEST AL, AL
               JZ    Again          AL 为零时转移
               临界段 CS
               MOV flag, 1     退出临界段重建 AL 为 1
```

until key= false;

进程的临界段代码 CS;

lock: = false;

进程的其它代码;

forever

用上述硬件指令虽然可以有效地保证进程间互斥。但有一个缺点,就是当进程正在临界段中执行时,其它想进入临界段的进程必须不断地测试布尔变量 lock 的值,这就造成了处理机机时的浪费。我们常称这种情况为“忙等待”。在注①中给出了 Intel 8086 的非忙等待方式的执行方法。读者可对比前面的注①来阅读。

除此之外,也可用提高进程执行临界段时的中断优先级,不让中断来干扰的方法以达到互斥,如 PDP-11 的 UNIX 中所采用的方法那样。

4.2.2.2 信号量

信号量是由荷兰的计算机科学家 Dijkstra 于 1965 年提出的最早的同步方法。所谓信号量是一个仅能由同步原语对其进行操作的整型变量。Dijkstra 将这两个同步原语命名为“P 操作”和“V 操作”(P、V 来源于荷兰文的“发信号”和“等待”二词的第一个字母)。

信号量按其用途可分为:

(1) 二元信号量:它仅允许取值为“0”与“1”,主要用作互斥变量;

(2) 一般信号量:它允许取值为非负整数,主要用于进程间的一般同步问题(但在许多应用中允许其值为负整数,即对 Dijkstra 的原定义进行了扩充)。

4.2.2.3 P,V 操作

P,V 操作是对信号量进行的原语操作。Dijkstra 对这两个原语操作定义如下:

P(S):当信号量 S 大于零时将 S 值减 1。否则进程等待,直到其它进程对 S 进行 V 操作。

V(S):将信号量 S 的值加 1。

P 操作和 V 操作可用软件(或固件)和硬件来执行。无论 P 操作和 V 操作,它们的执行都必须是一个不可被中断的整体。

由于进程所用的等待方式不同,P 操作也可用不同的等待方式来执行。

当进程必须在信号量 S 上等待时,就将该进程的状态变为等待状态(或活动阻塞状态),并将该进程插入与此信号量有关的等待队列中,而后让出处理机给其它就绪进程。这是一种比较普遍采用的等待方式,尤其是在单机系统中。

---

① Again: MOV AL,0
XCHG AL,SEMAPHORE
TEST AL,AL
JNZ NEXT 非零转移
| }调用操作系统的管理程序,阻塞该进程
JMP Again
NEXT | }临界段 CS
MOV SEMAPHORE,1
| }调用操作系统的管理程序去唤醒等待的进程

当进程必须等待时,如果预计其等待时间不长,则在多机系统中,为了减少进程转换所引起的开销,可采取“忙等待”方式。

下面给出以上两种等待方式下的P,V操作执行的语言形式的定义。在这些定义中,我们所使用的信号量值可以为负。

(1) P,V操作的“忙等待”执行方式。

```
P(S): while S≤0 do skip;
        S:= S-1;
V(S): S:= S+1;
```

这种形式的P,V操作,如同前节附注中看到的那样,完全也可用硬件指令来形成。

(2) P,V操作的“阻塞等待”执行方式。

此种执行方式是把等待进程放入与此信号量S有关的阻塞队列,需要有该队列的头指针,因此要把信号量定义进一步进行扩充,其数据结构形式由“忙等待”方式中的整形变量,扩充成为记录形式:

```
type Semaphore= record
                  value: integer;
                  L: pointer to PCB;
                  end
```

故

```
P(S): S.value:= S.value-1;
      if S.value< 0
        then begin
               Insert(CALLER,S.L);
               Block(CALLER);
             end
V(S): S.value:= S.value+1;
      if S.value≤0
        then begin
               Remove(S.L,id);
               Wakeup(id);
             end
```

其中Insert, Block, Remove, Wakeup均为系统提供的过程。Insert(CALLER, S.L)是把调用者进程CALLER的进程控制块PCB插入信号量S的等待队列S.L。Block是把调用者进程CALLER的状态变为阻塞,并调用进程调度程序,以便选一新的就绪进程到处理机上运行。Remove(S.L,id)是从信号量S的阻塞队列S.L中,选一进程移出队列,并把该进程标识号(或其PCB地址)送入id中。Wakeup(id)是把该进程id的状态转换成活动就绪状态,并根据id进程的优先数高低决定是否需要抢占现行进程的处理机。

P,V操作的物理意义可这样来看待。当S>0时的信号量数值,表示该类资源的可用资源数。每执行一次P操作就意味着请求分配一个单位的该类资源给执行P操作的进

程,因此描述为 S: = S- 1。当 S≤0 时,表示已经没有此类资源可供分配了,因此请求资源的进程将被阻塞在相应的信号量 S 的等待队列中。此时 S 的绝对值等于在该信号量上等待的进程数。而执行一次 V 操作意味着进程释放出一个单位的该类可用资源,故描述为 S: = S+ 1。若 S≤0 表示信号量等待队列中有因请求该资源而被封锁的进程,因此就应把等待队列中的一个进程(往往是第一个)唤醒(即改变进程的阻塞状态),使之转到就绪队列中去。在多用户系统"ESOPE"中还引进了第三个原语 D(S),其作用是连续唤醒该阻塞队列中的所有进程。

4.3 同步机构应用

本节主要研究用 P,V 操作来实现互斥与同步的几个例子。读者可通过这些例子掌握并行程序设计中解决此类问题的方法。

4.3.1 用信号量实现进程间的互斥

信号量上的原语操作使临界段问题的解决比较简单明了了。

假定 mutex 是一个互斥信号量。由于每次只允许一个进程进入临界段,如果把临界段看作资源,那么它的可用单位数为 1,所以 mutex 的初值应为 1(见信号量物理意义)。下面用 mutex 这个信号量在 n 个进程间实现互斥。各进程的并行执行大致描述如下:

```
var mutex: Shared Semaphore;
begin
    mutex: = 1;
cobegin
  P1: …;
  P2: …;
      ┆
  Pi: begin
    repeat
        P(mutex);
    "进程 Pi 的临界段代码 CSi";
    V(mutex);
    "进程 Pi 的其它代码部分";
    forever
        ┆
  Pn: …;
  coend
end
```

当有一进程在临界段中时,则 mutex 的值为零,否则,mutex= 1。这样,互斥信号量起着相当于一把锁的作用。互斥的达到是由于只有一个进程能用 P 操作把 mutex 减为零。

当有第二个进程对 mutex 执行 P 操作时, mutex 变为- 1。于是满足 P 操作定义中 mutex < 0 的条件, 该进程进入等待队列, 直到第一个进程退出临界段时, 做 V 操作唤醒它。进程在 mutex 上做 V 操作的结果, 使 mutex 由- 1 又变为零, 于是唤醒在 mutex 信号量上等待的阻塞进程, 从而被唤醒进程可以进入临界段, ……。

需要指出的是, 如果想要在共享主存储器的几个处理机之间实现互斥地访问共享变量, 用信号量上的 P, V 操作是难以解决的, 最好还是使用硬件指令(TS 指令和 Swap 指令)或临界段的软件方法来解决这问题, 其道理请读者自己考虑。

4.3.2 信号量作为进程的阻塞和唤醒机构

当一个进程等待某事件时, 最好将它自己阻塞在某个信号量上以等待事件的完成。通常这可以分成两种情况。

(1) 在资源分配和释放情况中, 进程所等待的事件是其所要求的资源被其它进程释放出来成为可用。这时信号量代表该资源的数量, 而信号量上的 P, V 操作可以成为资源分配和释放的机构。此时, 若进程在非正的信号量上执行 P 操作, 该进程就会非自愿的被迫阻塞, 进行等待。关于用信号量上的 P, V 操作来作为分配资源和释放资源的机构比较简单, 读者可作为练习。

(2) 在进程间相互协同情况下, 使进程等待某事件来控制其执行顺序, 此时进程所等待的事件主要是等待相关的协作进程完成某个操作。当进程在等待这类事件时, 最好将它自己阻塞, 以等待该事件的发生或完成。但这种情况不像上述的资源分配情况, 有与每个资源相应的信号量可供使用。为解决这类阻塞的需要, 我们为每个进程都设置一个初值为零的信号量 Proc-sem(i), 其中 i 是进程的内部标识号。通常, 系统把所有进程的信号量 Proc-sem(i)组成一个信号量数组。有了这样的信号量后, 我们就可以把信号量作为进程阻塞和唤醒的机构。当一个进程要阻塞自己以等待事件完成(这种阻塞是进程自愿的)时, 则可以执行:

P(Proc-sem(进程内部标识号));

由于信号量 Proc-sem 的初值为 0, 所以 P 操作的结果为- 1, 于是进程就主动地封锁在它自己的信号量上了。

当一个相关的协同进程完成了它的操作后, 要唤醒等待它的进程 i_0, 于是它可以用 V 操作作为唤醒工具:

V(Proc-sem(i_0));

例如两个协同进程 A 和 B, 进程 A 要等待进程 B 的计算结果。则这两个进程的执行如下:

进程 A:	进程 B:
进行计算;	进行计算;
P(Proc-sem(A));	V(Proc-sem(A));
继续计算;	继续计算;

所有进程的阻塞都是由于进程自己在非正的信号量上执行 P 操作的结果。进程只能自己阻塞自己,不能阻塞其它进程。而进程的被唤醒只能是由其它进程执行 V 操作的结果。即由其它进程唤醒,而不能自己唤醒自己。

4.3.3 生产者和消费者问题

生产者和消费者问题及其同步技术是由 Dijkstra 于 1968 年提出的。在计算机系统中的许多问题都可以将它归结为生产者和消费者问题。例如:一个进程如果它使用资源(尤其是软资源——缓冲区中的数据,进程间通信的消息等),可把它称为消费者。而一个进程如果是制造并释放上述资源,则可把它称为生产者。它们二者之间的同步关系问题就称为生产者和消费者关系问题。在此类问题中,信号量可以作为资源计数器和同步进程的工具。下面介绍的一个简单的以缓冲存储器(区)为媒介的两进程间的通信例子,是由 Dijkstra 提出的:

一个生产者进程生产消息,然后把它放在缓冲存储区中。与此平行地,一个消费者进程从缓冲存储区中移走消息并处理它。假设该缓冲存储器是由 N 个相等大小的缓冲区组成,每个缓冲区能容纳一个记录,整个缓冲存储器的 N 个缓冲区构成如图 4.5 的环形缓冲区(链成缓冲区环)。显然此问题中的共享资源是缓冲区。但有两类缓冲区:空缓冲区和装有数据的满缓冲区。为便于对缓冲区的管理,分别用指针 C 指出满缓冲区头,指针 P 指出空缓冲区头。每当有生产者要求一个空缓冲区,则分给其指针 P 指出的缓冲区,同时指针 P 按箭头方向移动一个位置。消费者要求满缓冲区的操作,也类似地移动指针 C,并分给一个满缓冲区。我们假定系统中有等价的多个生产者和多个消费者进程。那么分配空缓冲区和移动指针 P 的操作应是互斥的临界段操作,对满缓冲区和指针 C 的操作也是互斥操作。所以共有两个临界资源。我们为每类资源设置一个信号量,则信号量 E 代表空缓冲区资源信号量,F 是满缓冲区信号量,M_e 是空缓冲区指针操作互斥信号量,M_f 是满缓冲区指针操作互斥信号量。于是两类进程的同步关系描述如下:

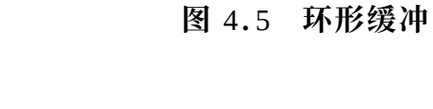


图 4.5 环形缓冲区

```
var E, F, Me, Mf: Shared Semaphore;
begin
      E: = N; F: = 0; Me: = Mf: = 1; (置初值)
cobegin
Producteres(诸生产者进程):
  begin
    repeat
    P(E);
    P(Me);
```

```
    “分给空缓冲区并调整指针 P 的临界段”;
      V(Me);
    “向缓冲区中装入数据”;
      V(F);
    forever
      end
  Consumeres(诸消费者进程):
      begin
        repeat
          P(F);
          P(Mf);
    “分给满缓冲区并调整指针 C 的临界段”;
          V(Mf);
    “从缓冲区中取出数据”;
          V(E);
        forever
      end
    coend
end
```

程序中有两点要请读者注意:

(1) 无论在生产者进程还是在消费者进程中,P 操作的次序都不能颠倒,否则将可能造成死锁(请考虑用一个互斥信号量来代替 M_e 和 M_f 的情况,见习题 4-20)。

(2) 若只有一个生产者和一个消费者时,互斥信号量可以不要。

4.3.4 阅读者/写入者问题

一个数据集(如文件或记录)如果被几个并行进程所共享。那么有些进程只是要求读这数据集的内容,而另一些进程则要求修改这数据集的内容。这种情况在文件系统、数据库中是很普遍的。那么在这种问题中有什么同步关系呢。通常我们把读进程称阅读者,而把要求修改数据的进程称为写入者,而把此类问题归结为“阅读者/写入者”问题。很显然,几个阅读者可以同时读此数据集,不需要互斥也不会产生任何问题,不存在破坏数据集中数据完整性、正确性的问题,但是一个写入者不能与其它进程(不管是写入者和阅读者)同时访问此数据集,它们之间必须互斥,否则将破坏此数据的完整性。设想一个银行管理系统中,一个分行向总帐目中写入存款数时(写入者),如结帐进程(阅读总帐目数据的阅读者)或其它分行的写入者同时并行对此数据操作,就会产生如同 4.2.1.1 节航空订票系统中的同样问题,即数据完整性被破坏,帐目是错误的。所以写操作必须互斥地进行。

解决此问题的最简单的方法是这样:当没有写进程正在访问共享数据集时,阅读者可以进入访问,否则必须等待。在我们解法中不考虑阅读者和写入者同时要求访问共享数据集时,优先照顾写入者的情况。其算法如下:

```
var mutex, wrt: Semaphore;
    readcount: integer;
    mutex: = wrt: = 1;
    readcount: = 0;
cobegin
Reader i: begin
          P(mutex);
          readcount: = readcount+ 1;
          if readcount= 1 then P (wrt);
          V(mutex);
          读数据集;
          P(mutex);
          readcount: = readcount- 1;
          if readcount= 0 then V (wrt);
          V(mutex);
          end
Writer i: begin
            P(wrt);
          写数据集;
            V(wrt);
            end
      coend
```

由于写入者要求进行写操作时,必须没有阅读者在进行读数据,所以要用 readcount 记录正在读的阅读者数。但对于所有阅读者来说,readcount 是一个共享变量,所以要互斥访问。用 mutex 作为访问 readcount 变量的互斥信号量。由于对共享数据集访问时要进行写操作互斥,所以用 wrt 作为互斥信号量。

当一个写入者正在临界段中时,如有几个阅读者要求访问共享数据集,则只有一个阅读者在 wrt 上等待,而其它 n- 1 个阅读者等待在 mutex 信号量上。对此算法的改进——优先让写入者进入,留作读者的练习。

4.4 进程间的通信

为提高计算机系统的资源利用率和作业的处理速度,一个作业通常被分为若干个可并行执行的进程。这些进程是异步的、相对独立地向前推进。但由于它们是协同地完成一个共同的作业,所以它们应保持一定的联系,以便协调地完成任务。这种联系就是指在进程间交换一定数量的信息——称为进程通信。进程通信不但存在于一个作业的诸进程间,而且也存在于共享有关资源的进程之间。

进程间通信时所交换的信息量可多可少。少者仅是一些状态和数据的变换,多者可交

换成千上百个数据。

前述的所有互斥和同步机构实际上均可看作是进程间通信的一种方式,只不过交换的信息量较少。有些作者把此种通信方式称为低级通信原语。而把交换信息量较多的通信方式称高级通信原语,本节主要研究高级通信方法。目前使用比较多的通信方法有两种:即直接通信和间接通信,或称消息缓冲区和信箱。所谓直接通信是指一个进程直接发送一个消息给接收者进程。所谓间接通信是指进程并不是把消息发送给接收者进程,而是把消息放在某个"信箱"之中,而该信箱是为通信双方(或多方)所知道的(操作系统中诸进程之间通信是为了完成共同的任务,因此通信诸方是彼此知道的,而且是程序员设计、安排好的)。所谓"消息"通常由消息头和消息正文构成,可用 PASCAL 描述如下:

```
type Msg= record
        Msgsender       消息发送者
        Msgnext         下一个消息,链指针
        Msgsize         整个消息的字节数
        Msgtext         消息正文
        end
```

在直接通信情况下,进程可以调用原语:

Send(P, Msg),向进程 P 发送一个消息。

Receive(P, Msg),接受来自进程 P 的一个消息。

通常一个进程可以与多个进程通信。它可以向多个有关的进程发消息,也可以接受来自各有关进程发来的消息。因此每个进程为了接受和处理这些消息,可以把发送给它的这些消息组织成消息队列。各消息之间用 Msgnext 链接起来,这个消息链的头指针通常放在每个进程的进程控制块 PCB 中。

发送原语 Send 的主要工作是请求分给一个消息缓冲区,然后将消息正文传送到该缓冲区中,向缓冲区中填写消息头,再将该消息缓冲区挂到进程 P 的消息链上。

接收原语 Receive 主要把消息链上的消息逐个读入该进程的接收区进行处理消息。

在间接通信情况下的此二原语为:

Send(A, Msg),发送一个消息到信箱 A。

Receive(A, Msg),从信箱 A 接收一个消息。

消息通信与消息队列在许多操作系统和软件开发环境(如 Windows NT)中得到广泛的应用。

* 4.5 管程的概念

前述的各种互斥手段,如测试和设置指令;P, V 在二元信号量上的操作,虽然都能有效地实现互斥,但都存在一些缺点:

(1) 临界段操作的代码以及进入和退出临界段时的"上锁"和"开锁"操作代码,均要由用户来编写,这加重了用户负担;

(2) 所有 P, V 操作都分散在用户编写的各程序代码中,由进程来执行。这样系统无

法有效地控制和管理这些 P, V 操作;

(3) 用户编程时难免会发生不正确地使用 P, V 操作的错误,这种错误会给系统带来不堪设想的后果(如死锁)。更麻烦的是,无论是编译程序或操作系统都无法发现和纠正此类错误。

例如用户编写出下列两种执行序列:

P(mutex);	V(mutex);
“临界段”;	“临界段”;
P(mutex);	P(mutex);

请读者想想,将会出现什么情况(其中 mutex 是互斥信号量)?!

于是出现了首先由 Dijkstra 提出,并由 Hansen 于 1974 年正式实现的另一种同步机构——管程。

4.5.1 管程的定义

由于对临界段的访问分散在各进程之中,这样不便于系统对临界资源的控制和管理,也无法发现和纠正分散在用户程序中的不正确地使用的 P, V 操作等问题。于是有人提出能否把这些分散在各进程中的临界段集中起来加以管理。怎样集中呢? 由于临界段是访问共享资源的代码段,所以 Dijkstra 提出为每个共享资源设立一个“秘书”来管理对它的访问。一切来访者都要通过秘书,而秘书每次仅允许一个来访者(进程)访问共享资源。这样既便于系统管理共享资源,又能保证互斥访问。

以后 Hansen 在并行 PASCAL 语言中,把“秘书”概念改名为管程(monitor),并将它作为该语言中的一个数据结构类型来用以描述操作系统。在该语言中,管程和进程都是操作系统的一个结构成分(详见有关此语言的著作)。管程是管理进程间同步的机制,它保证进程互斥地访问共享变量,并且提供了一个方便的阻塞和唤醒进程的机构。

管程由以下两部分构成:

(1) 局部于该管程的数据结构——共享变量。该共享变量表示其相应的共享资源的状态(通常系统为每个共享资源设置一个管程);

(2) 局部于该管程对上述数据结构进行规定的操作的若干个过程。

局部于管程内的数据结构只能被局部于管程内的过程所访问,不能被管程外的过程对其进行操作。反之,局部于管程内的过程只能访问管程内的数据结构。因此管程相当于围墙一样把共享变量(数据结构)和对它进行的若干操作过程围了起来。进程要访问共享资源(进入围墙使用某操作过程)就必须经过管程(围墙的门)才能进入,管程每次只允许一个进程进入管程内,即互斥地访问共享资源。

4.5.2 五位就餐的哲学家问题

限于篇幅不能详细介绍管程的概念,本节通过例子来帮助读者进一步理解管程的概念。

五位就餐的哲学家问题是由 Dijkstra 提出的。其问题是这样:有一张圆桌,共有五个

座位和五个盘子,在相邻的两个盘子中间有一把叉子。五位哲学家中的每一位都处于两种状态之一:“思考”问题状态和感到饥饿而去“吃通心粉”状态。吃通心粉需要同时使用两把叉子,我们规定每把叉子只能供其左右两边的就餐者使用。于是有可能出现某位哲学家想吃通心粉而又拿不到叉子的情况。为描述此情况,我们增加一个哲学家等待叉子成为可用的“饥饿”状态。我们的问题是管理这些临界资源——叉子,以及控制诸哲学家(进程)对叉子的使用。我们用管程来解决此问题。

根据问题显然有以下事件成立:

(1) 相邻两位哲学家不能同时进餐;

(2) 最多只能有两人同时用餐,即最大并行性为 2;

(3) 为排除随机因素影响,规定哲学家就餐时,首先拿起他左边的叉子,而后拿右边的叉子,不能同时拿起两边的叉子。餐后放下叉子,也以先左后右的同样顺序。

显然,管程要管理的是描述共享资源——叉子的数据结构,以及哲学家对叉子所能进行的操作过程:“拿起叉子”过程和“放下叉子”过程。我们将这两个过程记为“PickUp”和“PutDown”。

描述叉子的数据结构可以用表示各叉子状态的数组来描述,记为 Forks。但为使程序简便起见,我们把此数组改变一下。每个数组元素 Forks(i)不是表示叉子的状态,而是第 i 位哲学家可用的叉子数(其取值分别可以为 0, 1, 2)。数组的下标与每位哲学家的编号相一致。所以对于第 i 位哲学家来说,仅当 Forks(i) = 2 时,他才可以拿起叉子吃。否则必须等待在自己的条件变量上。各哲学家条件变量的集合用数组 Ready 来表示。程序中还用 Right(i)和 Left(i)这两个数组元素来表示第 i 位哲学家的右邻和左邻的哲学家号。

现将问题解法描述如下:

```
ForksControl: monitor
    type T = array[0..4] of integer;
         B= array[0..4] of boolean;
    var  Forks, Right, Left: T;
         Ready: B;        i: integer;
  procedure entry PickUp(var me: integer);
  begin
    if Forks(me)≠2 then
      wait(Ready(me));
    Forks (Right(me)):= Forks(Right(me))- 1;
    Forks (Left(me)):= Forks(Left(me))- 1;
  end
  procedure entry PutDown(var me: integer);
  begin
    Forks (Right(me)):= Forks(Right(me))+ 1;
    Forks (Left(me)):= Forks(Left(me))+ 1;
  if Forks (Right(me))= 2 then Signal (Ready(Right(me)));
```

```
    if Forks (Left(me))= 2 then Signal (Ready(Left(me)));
    end
        begin
                for i= 0 to 4 do
                    begin Forks(i):  = 2;
                        Right(i):  = [i+ 1]mod 5;
                        Left(i):  = [i+ 4]mod 5;
                        Ready (i):  = false;
                    end
      cobegin
Philosopher 0:  Process(var hung: boolean);
        begin
              repeat
        if hung= true then
          begin
            ForksControl · PickUp(0);
            "吃通心粉";
            ForksControl·PutDown(0);
          end
          "思考问题";
          forever
        end
    Philosopher 1:  Process;…;
                  ┆
    Philosopher 4:  Process;…;
      coend
```

赋初值

管程中的两个标准过程:WAIT 和 SIGNAL 是由并行 PASCAL 语言本身提供的,相当于 P,V 操作。

五位哲学家进程的代码是类似的,所以仅给出 0 号哲学家的程序。其他哲学家的程序只需修改调用管程体的过程中的参数为相应的哲学家代号即可。

4.6 Windows NT 中的同步与互斥机制

Windows NT 是支持对称多处理模式,也就是说,它是一个多机操作系统。为适应这种复杂系统内的同步、互斥以及进程通信的需要。Windows NT 根据不同情况提供了多种同步与互斥机制。现将其使用的各种同步与互斥机制及应用情况列出如下:

(1) 内核中的同步与互斥机制:实现多处理器之间的同步。主要保护线程调度时所要使用的调度数据基和其它的全局变量。使之互斥地访问这些全局变量,要求各线程互斥地

执行临界段。在 Windows NT 的内核中,根据不同情况采用不同机制:

① 提高临界段代码执行的中断优先级:以实现同一处理机(或单机情况下)中的互斥(请读者思考这种方法是怎样达到互斥的,详情请参阅 12.6.3 节)。

② 转锁(spin-lock)机制:这是一种如同 4.2.2.1 节中的 T-S 指令的机制。用于多处理器间的互斥(请读者思考:在多机情况下,为何第一种方法不能实现互斥,本方法又是如何达到互斥的,详情请参阅 12.6.3 节)。

(2) 线程之间的同步采用"等待和信号设置"机制:该功能由 NT 执行体的"对象管理程序"提供,详细情况请参阅 12.11.1 节。

(3) 进程通信机制:由"本地过程调用(LPC)"提供,根据通信进程间交换的信息量而分为:

① 小消息(少于 256 个字节)通信方法;

② 大消息(大于 256 个字节)通信方法;

③ 多通信端口机制。

以上三种通信机制的详细情况请参阅 12.11.2 节。

习　　题

4-1　顺序程序设计和并行程序设计的特点有何不同?

4-2　什么叫与时间有关的错误?表现在哪些方面?举例说明之。

4-3　什么叫临界段?临界段设计原则是什么?

4-4　若进程 A 和 B 在临界段上互斥,那么当 A 处于临界段内时不能打断它的执行,这说法对吗?为什么?

4-5　同步与互斥这两个概念有何区别?

4-6　信号量的物理意义是什么?应如何设置其初值?并说明信号量的数据结构。

4-7　信号量是一个初值为非负的整形变量,可在其上做加"1"和减"1"的操作。这说法对吗?如何改正之?

4-8　使用 cobegin/coend 改写下面的表达式以获得最大程度的并行性。

(3* a* b+ 4)/(c+ d)* * (e- f)

4-9　把下列并行计算改写成顺序计算序列。

```
a:  = b+ c;
cobegin
     d:  = b* c- x;
     e:  = (a/b)+ n* * 2
coend
```

4-10　为什么下面的并行计算程序是不正确的?

```
 cobegin
    a:  = b+ c;
    d:  = b* c- x;
```

```
        e: = (a/b)+ n* * 2
    coend
```

4-11 说明下面的说法是不正确的理由:当几个进程访问主存中的共享数据时,必须实行互斥以防止产生不确定的结果。

4-12 互斥原语可以用"忙等待"或"阻塞等待"两种方法加以实现。讨论每种方法的实用性和相对的优点。

4-13 说明单处理机系统中在实现互斥时,为什么用提高中断优先级的办法是很有用的?

4-14 Windows NT 在多处理模式下,用提高中断优先级以实现互斥可行吗?为什么?

4-15 为什么 V 操作必须是不可分割的?

4-16 下面是两个并发执行的进程,它们能正确执行吗?若不能正确执行,请举例说明,并改正之(x 是公共变量)。

```
cobegin
    var x: integer;
  procecc P1(进程 P1)
      var y, z: integer;
      begin
        x: = 1;
        y: = 0;
        if x≥1 then y: = y+ 1;
        z: = y
      end
  procecc P2(进程 P2)
      var t, u: integer;
      begin
        x: = 0;
        t: = 0;
        if x< 1 then t: = t+ z;
        u: = t
      end
coend
```

4-17 因修路使 A 地到 B 地的多路并行车道变为单车道,请问在此问题中,什么是临界资源?什么是临界段?

4-18 设有几个进程共享一互斥段,对于如下两种情况:

(1) 每次只允许一个进程进入互斥段;

(2) 最多允许 m 个进程(m< n)同时进入互斥段;所采用的信号量是否相同?信号量值的变化范围如何?

4-19　有一阅览室,读者进入时必须先在一张登记表上进行登记,该表为每一座位列一表目,包括座号和读者姓名。读者离开时要消掉登记信息,阅览室中共有 100 个座位,请问:

(1) 为描述读者的动作,应编写几个程序?设置几个进程?进程与程序间的对应关系如何?

(2) 用类 Pascal 语言和 P,V 操作写出这些进程间的同步算法。

4-20　在 4.3.3 节的生产者和消费者问题的同步算法中,如果用一个互斥信号量 M 来代替算法中的两个互斥信号量 M_e 和 M_f(即算法的所有 M_e 和 M_f 处都用 M 来代替,请问:

(1) 改变后的算法与原算法各有何优缺点?

(2) 在改变后的算法中将生产者和消费者进程的两个相邻 P 操作交换一下顺序,则将有可能产生死锁,请举例说明为什么?

(3) 在(2)中若交换 V 操作顺序有影响吗?

4-21　进程间通信有几种方法?发送和接收消息原语的功能是什么?

4-22　何谓多处理模式?比较非对称多处理与对称多处理的特点有何异同?

4-23　Windows NT 操作系统在其内核中采用了哪些互斥方法?有何特点?

第 三 部 分

处理机管理

第 5 章　作 业 和 进 程 的 调 度

5.1　调度的层次和作业状态转换

5.1.1　调度的层次

在大型通用系统中,可能有数百个批处理作业存放在磁盘的作业队列中,有数百个终端同主机相联接。因此如何从这些作业中挑选作业进入主存运行、如何在作业或进程间分配处理机等问题,无疑是操作系统的资源管理功能中的一个重要问题。本章主要讨论处理机分配问题,或称处理机调度。

一般来说,处理机调度可以分成三级:

(1) 高级调度:又称作业调度,其主要功能是按照某种原则从磁盘某些盘区的作业队列中选取作业进入主存,并为作业做好运行前的准备工作和作业完成后的善后工作。

(2) 中级调度:它决定哪些进程被允许参与竞争处理机资源。中级调度主要只是起到短期调整系统负荷的作用,以平顺系统的操作。其所使用的方法是通过“挂起”和“解除挂起”一些进程,来达到平顺系统操作的目的。

(3) 低级调度:又称进程调度,其主要功能是按照某种原则将处理机分配给就绪进程。执行低级调度功能的程序称为进程调度程序,由它实现处理机在进程间的转换。由于它的执行频率很高,一秒钟要执行很多次,因此它必须常驻主存。是操作系统内核的主要部分。进程调度策略的优劣和处理机在进程间转换时的速度对整个系统的性能有很大影响。

本章主要研究单处理机情况下的作业和进程调度。

5.1.2 作业状态

作业从提交给系统直到它完成后离开系统前的整个活动常划分为若干阶段。作业在每一阶段中所处的状况称为作业的状态。系统中的作业通常分为四种状态:

(1) 提交状态: 一个作业被提交给机房后或用户通过终端键盘向计算机中键入其作业时所处的状况为提交状态。

(2) 后备状态: 作业的全部信息都已通过输入机输入, 并由操作系统将其存放在磁盘的某些盘区(这些盘区专门存放输入作业故称为输入井) 中等待运行, 则称为后备状态。

(3) 运行状态: 作业一旦被作业调度程序选中而被送入主存中投入运行, 称之为运行状态。

(4) 完成状态: 作业完成其全部运行, 释放出其所占用的全部资源, 准备退出系统时的作业状况称为完成状态。

5.2 作业的调度

5.2.1 后备作业队列与作业控制块 JCB

系统中往往有成百个作业被收容在磁盘输入井中, 为了管理和调度作业, 就必须记录已进入系统的各作业的情况。因此同进程中的情况类似, 系统也为每个作业设置一个作业控制块(记为 JCB), 它记录了作业的有关信息。不同系统的 JCB 所包含的信息有所不同, 这取决系统对作业调度的要求。图 5.1 列出了 JCB 的主要内容。

作 业 名	
资 源 要 求	要求的运行时间
	最迟完成时间
	要求的主存量
	要求外设类型、台数
	要求的文件量和输出量
资 源 使 用 情 况	进入系统时间
	开始运用时间
	已运行时间
	主存地址
	外设台号
类 型 级 别	控制方式
	作业类型
	优先数
状 态	

图 5.1　作业控制块

JCB 是在作业进入系统时由 SPOOL 系统为其建立的。其内容由作业控制卡(说明书)中得到。同样 JCB 也是作业存在于系统的标志,作业进入系统时,则为之建立 JCB。当作业退出系统时,则其 JCB 也被撤消。

在磁盘输入井中的所有后备作业按作业类型(输入输出型,CPU 型)将它们组成一个或多个后备作业队列。所谓后备作业队列是由作业控制块 JCB 用表格或链指针组成的队列。作业队列可按优先数大小和(或)作业到达系统的时间顺序排列。

5.2.2　作业调度及其功能

所谓作业调度,即是按照某种调度算法从后备队列中挑选作业进入主存中运行。为此,作业调度还要为选中的作业分配资源,作好作业运行前的准备。完成作业调度功能的程序称为作业调度程序。通常作业调度程序要完成以下工作:

(1) 按照某种调度算法从后备作业队列中挑选作业。

(2) 为选中的作业分配主存和外设资源。因此作业调度程序在挑选作业过程中要调用存储管理程序和设备管理程序中的某些功能。

(3) 为选中的作业建立相应的进程。

(4) 构造和填写作业运行时所需的有关表格,如作业表(登记所有在主存中各作业的有关信息)等。

(5) 作业结束时完成该作业的善后处理工作,如收回资源,输出必要的信息,撤消该作业的全部进程(PCB)和作业控制块 JCB。

5.3　进 程 调 度

作业调度程序在挑选作业进入主存运行时,要为该作业建立相应的进程。在作业完成后要撤消该作业的全部进程。因此作业调度程序要调用操作系统内核所提供的有关的进程管理原语(见 3.4 节),如"建立进程"原语和"撤消进程"等原语。由于进程只能由其父进程建立,所以在一般系统中,作业调度程序都以进程的形式在系统中存在和活动,称为作业调度进程(UNIX 系统中的 1 号进程相当于作业调度进程)。作业调度进程可以说是系统中的祖先进程,由它完成作业调度的诸功能。

一个进程被建立后,系统为了便于对进程的管理,将系统中的所有进程按其状态,将其组织成不同的进程队列。于是系统中有运行进程队列(多机系统)、就绪进程队列和各种事件的进程等待队列(阻塞队列)。

进程调度的功能是从就绪队列中挑选一个进程到处理机上运行。负责进程调度功能的内核程序称为进程调度程序。

读者往往不易分清作业调度程序挑选作业进入主存运行和进程调度程序挑选一个进程到处理机上运行,这两种运行之间有何区别。所谓作业调度程序挑选作业进主存运行是个宏观的概念,实际上被选进主存运行的作业只是具有了竞争处理机的机会(将来真正在处理机上运行的是该作业的一个进程)。而进程调度程序才是真正让某个就绪进程到处理

机上运行。

由于进程调度程序负责在就绪进程间转接对处理机的控制,所以对它的调用相当频繁,每秒要执行很多次。它是操作系统核心(常称为内核)的重要组成部分。进程调度程序在系统中以原语形式存在,它是为进程在系统内的活动提供支持。

5.4 选择调度算法时应考虑的问题

目前比较普遍使用的几种调度算法,对于作业调度和进程调度大致上都是适用的,所以我们在下面的讨论中不严格区分哪种调度算法是属于两类调度(作业调度和进程调度)的哪一类。设计者根据系统设计要求加以选择。

在设计系统的调度程序时,首先要决定选择何种调度算法,依据此算法来编制相应的调度程序。而调度算法实际上就是系统所采取的调度策略,选择时所要考虑的因素很多。如系统各类资源的均衡使用;对用户公平并使用户满意;用户作业到达系统的时间;作业的优先数;对主存和外设的要求;以及整个系统的效率等。但这些因素之间往往相互矛盾,很难兼顾。例如计算中心为了提高系统的吞吐量要求每天运行尽可量多的作业,于是希望尽可能优先运行短作业;但为了提高处理机的使用效率,希望处理机总是处于忙的状态,则最好运行大的作业。这二者就相互矛盾。所以设计时应将那些对系统运行影响较大的关键因素作为调度算法考虑的主要依据。通常应考虑以下因素:

(1) 设计目标:系统的设计目标是选择算法的主要依据,目标不同,系统的设计要求自然不同。如批处理系统所追求的是充分发挥和提高计算机的效率;实时系统所关心的是不要丢失实时信息并及时给以处理;而分时系统则侧重于保证用户的请求及时给予响应;计算中心要求系统吞吐量要大等等。

(2) 资源利用率:这是评价系统性能优劣的重要指标。因此在确定算法时,在考虑设计目标的前提下应充分发挥各种资源的效能,最大限度地使它们忙碌。例如将科学计算型(CPU 型)作业和数据处理型(输入输出型)作业搭配运行就是一种方法。

为充分提高资源利用率还应注意照顾正在使用关键资源的进程,使之优先运行。

(3) 均衡地处理系统和用户的要求:由于系统和用户的要求往往是矛盾的,确定算法时也要设法给予缓和。一般说来,用户本能地希望尽快获得运行结果。但系统有时却不能立即满足用户要求,例如个别用户可能要求使用系统中的几乎全部外设,却只要求很少的主存。系统若满足这类用户的愿望,势必影响主存利用率,从而降低系统效率,所以一般都不得不推迟这种作业的运行时间,等到有要求内存多而外设少的作业与之搭配运行。

但是我们选择的算法也不应使一个作业的运行被无限制地推迟。这是用户无法忍受的。解决的办法是使进程优先数随等待时间而增加。

(4) 在使用优先级的系统中,每个进程都有一个优先数(或优先级),调度算法应优先运行高优先级进程。

(5) 在使用优先数的系统中,调度策略还分为“可抢占”和“不可抢占”两种方式。所谓“不可抢占”是指一旦某个进程分得处理机后,除非它因等待某事件发生或已完成其任务而主动让出处理机外,不得将处理机从该进程抢走给其它进程使用。所谓“可抢占”是指可

以将处理机从该进程抢占给其它进程使用,即使该进程仍然需要处理机。抢占策略通常使用于需要迅速响应高优先级进程的系统中。但抢占策略要增加系统两个方面的开销: 首先是增加了处理机在进程间转接的次数,消耗处理机机时。其次系统为提高抢占工作的效率,在主存中要存放多个作业(进程),以保证抢占后,该进程可立即投入运行,这又增加了不运行进程占用主存的开销。

5.5 调 度 算 法

5.5.1 先来先服务调度算法 FIFO

先来先服务算法是最简单的调度方法。其基本原则是按照作业到达系统或进程进入就绪队列的先后次序来选择。对于进程调度来说,一旦一个进程占有了处理机,它就一直运行下去,直到该进程完成其工作或者因等待某事件而不能继续运行时才释放出处理机。FIFO 策略是属于不可抢占策略。从表面上来说,对于所有进程和作业都是公平的,并且一个作业的等待时间是可以预先估计的。但是从另一方面来说这个方法也不见得公平,因为当一个大作业先到达系统时就会使许多小作业等待很长时间,提高了平均的作业周转时间,会使许多小作业的用户不满。

今天先来先服务算法已很少用作主要的调度策略,尤其是不能在分时和实时系统中用作主要的调度策略。但它常被结合在其它的调度策略中使用。例如,在使用优先级作为调度策略的系统中,往往对许多具有相同优先级的进程,使用先来先服务的原则。

5.5.2 优先级调度算法

按照进程的优先级大小来调度,使高优先级进程得到优先的处理的调度策略称为优先级调度算法。进程的优先级可以由系统自动地按一定原则赋给它。也可以由系统外部来进行安排,甚至可由用户支付高费用来购买优先级。

但在许多采用优先级调度的系统中,通常采用动态优先数策略。即一个进程的优先级不是固定的,往往随许多因素的变化而变化。尤其随作业(进程)的等待时间、已使用的处理机时间或其它资源的使用情况而定,为此系统还提供了“改变进程优先数”原语。

优先级调度方法又可分为:

(1) 非抢占的优先级调度法: 即一旦某个高优先级的进程占有了处理机,就一直运行下去,直到由于其自身的原因而主动让出处理机时(任务完成或等待事件)才让另一高优先级进程运行。

(2) 可抢占的优先级调度法: 任何时刻都严格按照高优先级进程在处理机上运行的原则进行进程的调度,或者说,在处理机上运行的进程永远是就绪进程队列中优先级最高的进程。如果在进程运行过程中,一旦有另一优先级更高的进程出现时(如一高优先级的等待状态进程因事件的到来而成为就绪),进程调度程序就迫使原运行进程让出处理机给高优先级进程使用或叫做抢占了处理机。在 UNIX 系统中,其进程调度算法就是属于“可抢占的优先级调度算法”,每个进程的优先数都是动态优先数,由“改变进程优先数”原语

为各进程每秒钟计算一次优先数。此算法主要用于进程调度。

5.5.3 时间片轮转算法

时间片轮转算法主要用于进程调度。采用此算法的系统,其进程就绪队列往往按进程到达的时间来排序。进程调度程序总是选择就绪队列中的第一个进程,也就是说按照先来先服务原则调度,但一旦进程占有处理机仅使用一个时间片。在使用完一个时间片后,进程还没有完成其运行,它也必须释放出(被抢占)处理机给下一个就绪的进程。而被抢占的进程返回到就绪队列的末尾重新排队等候再次运行。时间片轮转策略特别适合于分时系统中使用,当多个进程驻留在主存中时,在进程间转接处理机的开销一般是不大的。

时间片的大小对计算机系统的有效操作影响很大。所以在设计此算法时,时间片的选择究竟是大的好还是小的好;时间片的值是固定好还是可变值好;时间片的值是否对所有用户都相同呢还是随不同用户而不同。这些都应考虑。显然,如果时间片很大,大到一个进程足以完成其全部运行工作所需的时间。那么此时间片轮转策略就退化为先来先服务策略了。如果时间片很小,那么处理机在进程间的转接工作过于频繁,其处理机机时开销将会变得很大,而处理机真正用于运行用户程序的时间将会减少。通常最佳的时间片量值应能使分时用户得到好的响应时间。因此时间片量值应选得大于大多数分时用户的询问时间,即当一个交互进程正在执行时,我们给它的时间片相对来说略大些,使它足以产生一个输入输出要求。或者说时间片大小略大于大多数进程从计算到输入输出要求之间的间隔时间。这样可使用户进程工作在最高速度上,并且也减少了不必要的在进程间转接处理机的开销,提高了输入输出设备的利用率,同时也能提供较好的响应时间。

具体的最佳时间片值各个系统是不同的,而且随着系统负荷不同而有变化。关于时间片的更进一步考虑和时间片轮转的策略请看 5.5.7 节。

5.5.4 短作业优先调度算法

短作业优先调度算法主要用于作业调度,它是从作业的后备队列中挑选那些所需的运行时间(估计时间)最短的作业进入主存运行。这是一个非抢占的策略,它一旦选中某个短作业后,就保证该作业尽可能快地完成运行并退出系统。这样就减少了在后备队列中等待的作业数,同时也降低了作业的平均等待时间,提高了系统的吞吐量。但从另一方面来说,各个作业的等待时间的变化范围较大,并且作业(尤其是大作业)的等待时间难以预先估计。也就是说用户对他的作业什么时候能完成心里没有底,而在先来先服务策略中,作业的等待和完成时间是可以预期的。

短作业优先策略要求事先能正确地了解一个作业或进程将运行多长时间。但通常一个作业没有这方面的可供使用的信息,只能估计。在生产环境中,对于一个类似的作业可以提供大致的合理的估计。而在程序开发环境中,用户就难以知道他的程序大致将运行多长时间。

正因为此策略明显偏向短作业,而且作业的运行时间是估计的。所以用户可能把他的作业运行时间估计过低,以争取优先运行。为纠正这一情况,当一个作业超过其估计时间时,系统将停止这个作业,或对超时部分加价收费。

短作业优先策略和先来先服务策略都是非抢占的,因此均不适合于分时系统,这是由于它不能保证对用户及时地响应。

5.5.5 最短剩余时间优先调度算法

最短剩余时间优先调度算法是把最短作业优先算法使用于分时环境中的变型。其基本思想是让运行到作业完成时所需的运行时间最短的进程优先得到处理,其中包括新进入系统的进程。在最短作业优先策略中,一个作业一旦得到处理机就一直运行到完成(或等待事件)而不能被抢占(除非主动让出处理机)。而最短剩余时间优先策略是可以被一个新进入系统,并且其运行时间少于当前运行进程的剩余运行时间的进程所抢占。

本策略的优点是可以用于分时系统,保证及时响应用户要求。缺点是系统开销增加,首先要保存进程的运行情况记录,以比较其剩余时间大小。其次,抢占本身也要消耗处理机时间。毫无疑问,这个策略使短作业一进入系统就能立即得到服务,从而降低作业的平均等待时间。

5.5.6 最高响应比优先调度算法

Hansen 针对短作业优先调度策略的缺点提出了最高响应比优先调度策略。这是一个非抢占的调度策略。按照此策略每个作业都有一个优先数,该优先数不但是要求的服务时间的函数,而且是该作业为得到服务所花费的等待时间的函数。作业的动态优先数计算公式如下:

$$\text{优先数}=\frac{\text{等待时间}+\text{要求的服务时间}}{\text{要求的服务时间}}$$

要求的服务时间是分母,所以对短作业是有利的,它的优先数高,可优先运行。但是由于等待时间是分子,所以长作业由于其等待了较长时间,从而提高了其调度优先数,终于被分给了处理机。作业一旦得到了处理机,它就一直运行到作业完成(或因等待事件而主动让出处理机),中间不被抢占。

可以看出,“等待时间+要求的服务时间”就是系统对作业的响应时间。所以优先数公式中优先数值实际上也是响应时间与服务时间的比值,称为响应比。响应比高者得到优先调度。

5.5.7 多级反馈队列调度算法

短作业优先或最短剩余时间优先均是在估计的作业运行时间基础上进行调度。但在程序开发环境中或其它情况下,往往难以估计作业的运行时间。本节所研究的策略是时间片轮转调度算法的发展,不必估计作业运行时间的大小。但是本策略仍然基于以下考虑:

(1) 为提高系统吞吐量和降低作业平均等待时间而照顾短作业。

(2) 为得到较好的输入输出设备利用率和对交互用户的及时响应而照顾输入输出型作业。

(3) 在作业运行过程中,按作业运行情况来动态地考虑作业的性质(输入输出型作业还是 CPU 型作业)。并且要尽可能快地决定出作业当时的运行性质(以输入输出为主还

是以计算为主),同时进行相应的调度。

具体来说,多级反馈队列的概念如下(图 5.2):

图 5.2　多级反馈队列

(1) 系统中有多个进程就绪队列,每个就绪队列对应一个调度级别,各级具有不同的运行优先级。第一级队列的优先级最高,以下各级队列的优先级逐次降低。

(2) 各级就绪队列中的进程具有不同的时间片。优先级最高的第 1 级队列中的进程的时间片最小,随着队列的级别增加其进程的优先级降低了,但时间片却增加了。通常提高一级其时间片增加一倍。

(3) 各级队列均按先来先服务原则排序。

(4) 调度方法:当一个新进程进入系统后,它被放入第 1 级就绪队列的末尾。该队列中的进程按先来先服务原则分给处理机,并运行一个相应于该队列的时间片。假如进程在这个时间片中完成了其全部工作或因等待事件或等待输入输出而主动放弃了处理机,于是该进程或撤离系统(任务完成)或进入相应的等待队列,从而离开了就绪队列。若进程使用完了整个时间片后,其运行任务并未完成仍然要求运行(也没有产生输入输出要求)。于是该进程被抢占处理机,同时将它放入下一级就绪队列的末尾。

(5) 当第 1 级进程就绪队列为空后,调度程序才去调度第 2 级就绪队列中的进程。其调度方法同前。当第 1,第 2 队列皆为空时才去调度第 3 级队列中进程……。当前面各级队列皆为空时,才去调度最后第 n 级队列中的进程。第 n 级(最低级)队列中的进程是采用时间片轮转方法进行调度。

(6) 当比运行进程更高级别的队列中到来一个新的进程时,它将抢占运行进程的处理机,而被抢占的进程回到原队列的末尾。

多级反馈队列的调度操作已大致描述如上,它是根据进程执行情况的反馈信息而对进程队列进行组织并调度进程,故而得名。但对此调度算法仍有几点要加以说明:

(1) 照顾输入输出型进程是我们的宗旨,其目的在于充分利用外部设备,以及对终端交互用户及时地予以响应。为此通常输入输出型进程进入最高优先级队列,从而能很快得到处理机。另外一方面,第 1 级队列的时间片大小也应使之大于大多数输入输出型进程产生一个输入输出要求所需的运行时间。这样既能使输入输出型进程得到及时处理,且避免了不必要地过多的在进程间转接处理机操作,以减少系统开销。

(2) CPU 型(计算型)进程由于总是用尽时间片(有些计算型进程一直运行几小时也不会产生一个输入输出要求)而由最高级队列逐次进入低级队列。虽然运行优先级降低了,等待时间也较长,但终究将得到较大的时间片来运行,直至到最低一级队列中轮转。

(3) 在有些分时系统中,一个进程由于输入输出完成而要求重新进入就绪队列时,并不是将它放入最高优先级的就绪队列。而是让它进入因输入输出要求而离开的原来那一级就绪队列。这就需要对进程所在的就绪队列序号进行记录。这样做的好处是有些计算型进程,偶然产生一次输入输出要求,输入输出完成后仍然需要很长的处理机运行时间。为减少进程的调度次数和系统开销,就不要让它们从最高级队列逐次下降,而是直接放入原来所在队列。

但是一个大的程序中,不同的程序段有不同的运行特点。有时计算多,有时输入输出多。也就是说一个计算型进程,有时可以看成(或具有)输入输出型进程。为此在有些系统中,当进程每次由于输入输出完成而重新进入就绪队列时将它放入比原来高一级的就绪队列中。这样就能体现进程由计算型向输入输出型变化的情况和要求。

使用多级反馈队列调度算法的例子是著名的 Windows NT 操作系统的调度算法。

5.6 Windows NT 可抢占动态优先级多级就绪队列调度算法

前面 3.5 节中已提到,在 Windows NT 中的调度实体不是进程,而是线程。负责挑选合适线程到处理机上运行并对多处理机进行调度的调度程序是 NT 执行体的内核中的线程调度程序。该调度程序使用的调度算法大致有以下要点(详细的实现技术请参阅 12.6.1 节):

(1) 每个线程在其生命期内有六种状态:

就绪状态;

备用状态;

运行状态;

等待状态;

转换状态;

终止状态。

状态之间的转换关系请参阅第 12 章图 12.3。

(2) 为防止一个线程独占处理机的情况发生,系统采用可抢占策略。

(3) 调度程序使用的主要调度数据库是多优先级就绪队列。

(4) 系统划分为 32 个优先级(0～31 级)。每个优先级一个就绪队列。高序号队列优先级也高。序号为优先级等级。参阅第 12 章 12.4 节。

(5) 调度程序首先从高优先级队列挑选线程,如果高优先队列为空,才进入下一级队列挑选。

(6) 线程的初始优先级与其进程的初始优先级有关,而进程的基本优先级是在进程创建时设定的。

(7) 线程完整地使用完一个规定的时间段后,被中断抢占其处理机。而优先级降低一级,进入下一级队列……直至到线程的基本的初始优先级。

(8) 当一个线程由等待状态变为就绪状态时要提高优先级。提高优先级的幅度与它等待的事件有关,如等待键盘输入所提高的幅度要大于等待磁盘 I/O。

由以上情况可知,Windows NT 调度算法类似于多级反馈队列。可以看作是多级反馈队列的一个具体实现。

习　　题

5-1　区分以下三级调度程序:

(1) 作业调度程序;

(2) 中级调度程序;

(3) 进程调度程序。

5-2　下述问题应由哪一级调度程序负责?

(1) 在可获得处理机时,应将它分给哪个就绪进程?

(2) 在短期繁重负载下,应将哪个进程暂时挂起?

5-3　举例说明为什么 FIFO 算法对交互式用户并不是一种恰当的进程调度模式。

5-4　时间片大小的确定是关键而又困难的工作,假定进程间的切换时间为 s,在进程发出 I/O 请求前的平均执行时间为 t(t> > s)。讨论下述时间片 g 的设置所产生的影响。

(1) g 为无穷大;

(2) g 比零稍大一点;

(3) g= s;

(4) s< g< t;

(5) g= t;

(6) g> t。

5-5　指出下述各点为什么是不正确的?

(1) 最短剩余时间优先调度的平均响应时间总是比短作业优先调度小;

(2) 短作业优先是公平的;

(3) 越短的作业应该受到越好的服务;

(4) 由于最短作业优先调度是优先选择短作业,故可用于分时系统;

5-6　指出最短剩余时间优先调度的一些弱点,应如何改进以便得到更好的系统性能。

5-7　回答下述有关最高响应比优先策略的问题。

(1) 本策略如何防止无限延迟的出现;

(2) 本策略与其它调度策略相比,是如何减弱对于新的短作业的优先;

(3) 假定两个作业等待的时间相同,它们的优先级相同吗? 请说明理由。

5-8　多级反馈队列是如何实现下述调度目标的?

(1) 短作业优先;

(2) 输入/输出型作业优先;

(3) 根据作业性质动态地进行优先调度。

5-9　作业调度程序的主要功能有哪些?为什么把作业调度进程称为系统中的祖先进程。

5-10　何谓抢占处理机? 试比较可抢占的优先级调度法与非抢占的优先级调度法的要点及利弊。

5-11　Windows NT 的调度算法要点有哪些?下列线程的优先级顺序如何(初始具有相同优先级)?

(1) I/O 工作性质的线程;

(2) 交互式线程;

(3) 计算性质的线程。

第 6 章　死　　锁

6.1　死锁问题的提出

死锁问题是 Dijkstra 于 1965 年研究银行家算法时首先提出的，而后为 Havender (1968)，Lynch(1971)等人相继认识和发展。实际上死锁问题是一种具有普遍性的现象。不仅在计算机系统中存在，就是在日常生活和其它各个领域中也是广泛存在的。

所谓死锁是指计算机系统和进程所处的一种状态。在系统中，两个或多个进程无限期地等待永远不会发生的条件，我们称此系统处于死锁状态。

在多道程序系统中，实现资源共享是操作系统的基本目标，但不少资源是互斥地使用的，在这种情况下，比较容易发生死锁情况。例如，在只有一台打印机和一台输入机的情况下，假定甲进程正占用着输入机，而它还想得到打印机，但打印机正被乙进程占用着，乙进程在未释放打印机之前又要求已被甲进程占用的输入机，则这两个进程都无法运行，进入死锁状态。

图 6.1 是对死锁现象的一种非形式的说明。X 轴和 Y 轴分别表示甲进程和乙进程的进展(用完成的指令条数来度量)。我们称这个二维空间是单处理机情况下两进程的进展空间，从该空间的原点开始的任何一条阶梯形折线为二进程的共同进展路线，这条进展路

图 6.1　死锁的图形说明

线一般情况下只能前进,因为指令的执行是不能倒退的。

当共同进展路径是按图 6.1 中的路径 1 前进,这时不会发生死锁,因为乙进程在甲进程提出对输入机的要求前,已完成了对输入机和打印机的使用,并且释放了它们。反之在乙进程对打印机提出要求前,甲进程已完成了对输入机和打印机的使用,并且释放了它们。这种共同进展路径也不会使系统成为死锁状态。但是如果二进程的共同进展路径按照这样的路径向前进时,甲进程占有了输入机,乙进程占有了打印机,从而共同进展路径进入了危险区。危险区的右上角的顶点是死锁点。共同进展路径只要一进入危险区,就必定要到达死锁点从而使系统成为死锁。这是因为共同进展路径在危险区中前进时必定会到达危险区边缘(上边、右边或死锁点),设到达了危险区的右边。这时的情况是乙进程仍然占有打印机,但尚未提出占用输入机的要求。而甲进程占有着输入机,并且提出对打印机的要求。甲进程由于资源要求不能满足而被阻塞,这时只有乙进程使用处理机并取得进展,直到乙进程提出对输入机请求时也被阻塞。系统成为死锁状态。二进程的共同进展路径到达死锁点以后就无法前进了。从这里可以看到,产生死锁的原因是:

(1) 系统资源不足;

(2) 进程推进顺序不合适。

在早期的系统中,由于系统规模较小,结构简单,以及资源分配大多采用静态分配法,使得操作系统死锁问题的严重性未能充分暴露出来。但今天由于多道程序系统,以至于数据库系统的出现,系统中的共享性和并行性的增加,软件系统变得日益庞大和复杂等原因,使得系统出现死锁现象的可能性大大增加。尤其在实时系统中,航天飞行的实时控制,生命支持系统的监督控制中,死锁问题更显得十分严峻。

需要指出的是,目前有些计算机系统中,为了节省解决死锁问题的耗费,当死锁发生机会不太多时,宁肯冒发生死锁的风险,而不采取避免和排除死锁的措施。但在复杂的大型计算机系统中,尤其是实时系统是要认真对待死锁问题的。

6.2 死锁的必要条件

6.2.1 资源的概念

一个操作系统基本上是一个资源管理者,它负责分配不同类型的资源给进程使用。现代的操作系统所管理的资源类型十分丰富,并且可以从不同的角度出发对其进行分类。比如,我们可以把资源分为“可抢占的”和“不可抢占的”资源。所谓可抢占资源是指资源占有者进程虽然仍然需要使用该资源,但另一进程却强行把资源从占有者进程处抢来,归自己使用。而不可抢占资源是指只有在占有者进程不再需要使用该资源而主动释放资源外,其它进程不得在占有者进程使用资源过程中强行抢占。一个资源是否属于可抢占的资源,这完全决定于资源的性质。比如磁带驱动器在一个进程未使用完前,其它进程是无法抢占的,因而是属于不可抢占资源。另外像打印机、读卡机之类资源也是不可抢占资源。而主存和 CPU 却是可抢占资源。

如果从资源的使用方式上来说,可以分成“共享”资源和“独享”资源。所谓共享资源是指该资源可以为几个进程共同使用。而独享资源则只能为某个进程单独使用,不能同时为

多个进程共同使用。如 CPU、主存、磁盘等皆为共享资源,而磁带驱动器,读卡机、打印机等则为独享资源。如果把资源按其使用期限来分,则可以分为“可再次使用的永久资源”和“消耗性的临时资源”。一般来说,所有的硬资源和可再入的纯代码过程,都属于可再次使用的永久资源,它们可被进程反复使用。而在进程同步和通信情况中出现的消息、信号和数据也可看作为资源,它们是属于消耗性的临时资源,因为类似消息这类资源,在接受消息进程对其处理后,消息就被撤消了,不再存在了。

所有消耗性临时资源或可再次使用的永久资源都有一个共同的性质:即一个进程由于请求一个资源而未被满足,从而该进程被阻塞。因此我们对资源的定义是这样给出的“一个逻辑资源”(简称为资源)是指可以引起一个进程进入等待状态的事物”。

在研究资源分配时,我们必须搞清该资源是可以被几个进程同时(宏观上)使用呢?还是只能为一个进程使用。资源的不同使用性质正是引起系统死锁的原因。

6.2.2 死锁的必要条件

Coffman, Elphick 和 Shoshani(1971)指出,对于可再使用的永久资源来说,下述条件为发生死锁的必要条件:

(1) 互斥条件:一个资源一次只能被一个进程所使用;

(2) 不可抢占条件:一个资源仅能被占有它的进程所释放,而不能被别的进程强行抢占;

(3) 部分分配条件:一个进程已占有了分给它的资源,但仍然要求其它资源;

(4) 循环等待条件:在系统中存在一个由若干进程形成的环形请求链,其中的每一个进程均占有若干种资源中的某一种,同时每一个进程还要求(链上)下一个进程所占有的资源(图 6.2)。

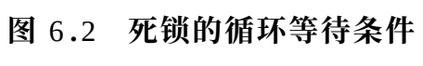


图 6.2 死锁的循环等待条件

要想防止死锁的发生,其根本办法是使上述必要条件之一(或多个条件)永不存在。换言之,就是破坏其必要条件使之永不成立。

第一个可能的途径是破坏互斥条件,即允许多个进程同时访问资源。但这受到资源本身的使用方法所决定,有些资源必须互斥访问,不能同时访问。如对公用数据访问必须是互斥的。又如几个进程同时使用打印机,一个进程打印一行,这种使用方式也是不可思议的,因此也必须互斥使用(不考虑 SPOOL 系统将物理的独享设备变为共享的虚拟设备的作用)。所以企图通过破坏互斥条件来防止死锁是不太实际的。

第二个可能的途径是破坏不可抢占条件,即强迫进程暂时把资源释放出来给其它进程。但这种强迫进程释放资源的方法只适用于 CPU 和主存,不适用于那些必须要求操作员装上私用数据介质的外部设备(如读卡机、打印机、磁带机等)。即使对于像主存和 CPU 之类的资源,可以进行抢占,但也会为抢占付出代价,不但要增加资源在进程间转移的机

时开销,而且还会降低资源的有效利用,所以也必须小心地加以控制。

第三个可能的途径是破坏部分分配条件。第四个可能的途径是破坏循环等待条件。这两者都是可能办到的,而且也被某些系统所采用。下面研究死锁预防的具体方法。

6.3 死锁的预防

死锁的预防中所要研究的问题是如何破坏死锁产生的必要条件,从而达到不使死锁发生的目的,这主要是通过破坏部分分配条件和循环等待条件来达到。

6.3.1 预先静态分配法

Havender(1968)提出的第一个策略就是预先静态分配法,这是针对部分分配条件的策略。即在作业调度程序选择作业时,仅当系统能满足作业运行时所需的全部资源时,才把该作业调入内存运行。同时在作业运行前,将其所需的全部资源一次性地分配给该作业,于是在作业运行过程中,不再会提出新的资源请求。

这个策略毫无疑问能够防止死锁的发生,但是它导致了严重的资源浪费。例如一个用户作业可能在作业完成时需要一台打印机打印结果数据,但必须在作业运行前就把打印机分配给它,而且在作业运行的整个过程中并不使用打印机。

因此一个更经常被使用的改进策略,是把程序分为几个相对独立的"程序步"来运行,并且资源分配以程序步为单位来进行,而不以整个进程为单位来静态地分配资源。这样可以得到较好的资源利用率,减少资源浪费,但增加了应用系统的设计和执行的开销。

这个策略的另一个缺点是由于其所需之全部资源不能在一次中得到全部满足,而可能使作业无限延迟。或者为了满足一个作业所需之全部资源,就必须逐渐积累资源。在积累过程中资源虽然空闲,也不能分给其它进程使用,造成了资源的浪费。

6.3.2 有序资源使用法

Havender 提出的第二个策略是有序资源使用法,这是针对循环等待条件的,即系统设计者把系统中所有资源类都分给一个唯一的序号,如输入机= 1,打印机= 2,穿孔输出机= 3,磁带机= 4,等等。并且要求每个进程均应严格按照递增的次序请求资源。亦即,只要进程提出请求资源 R_i,那么以后它只能请求排列在 R_i 后面的那些资源,而不能再要求序号低于 R_i 的那些资源。不难看出,由于对资源的请求作出了这种限制,在系统中就不可能形成几个进程对资源的环形请求链(图 6.2),破坏了循环等待条件。

这种方法由于不是采用预先静态分配方法,而是基本上基于动态分配方法,所以资源利用率较前一方法提高了,特别是小心地安排资源序号,把一些各作业经常用到的、比较普通的资源安排成低序号,把一些比较贵重或稀少的资源安排成高序号,便可能使最有价值的资源的利用率大为提高。因为高序号的资源往往等到进程真正需要时,才提出请求分配给进程。而低序号的资源,在进程即使暂不需要的情况下,但是进程需要使用高序号资源,所以在进程请求分配高序号资源时,不得不提前同时请求以后需要的低序号资源,从而造成资源空闲等待的浪费现象。

该策略虽然在许多操作系统中已经被执行,但也存在一些问题:

(1) 各类设备的资源序号一经安排,不宜经常地随意加以改动,起码应维持一个较长的时期(几月或数年)。在此期间,若要添置一些新设备,就必须重新改写已经存在的程序和系统。

(2) 资源序号的安排要反映大多数作业实际使用资源的正常顺序。对于与此序号相匹配的作业,资源能得到有效利用,否则,资源浪费现象虽然有改善,但仍然存在。

* 6.4 死锁的避免和银行家算法

死锁的避免与死锁预防的区别在于,死锁预防是严格地起码破坏死锁必要条件之一,使之不在系统中出现;而死锁的避免是不那么严格地限制必要条件的存在(因为死锁存在的必要条件成立,系统未必就一定发生死锁)。其目的是提高系统的资源利用率。万一当死锁有可能出现时,就小心地避免这种情况的最终发生。最著名的避免死锁算法是 Dijkstra 的银行家算法。

6.4.1 银行家算法问题

这个问题实际上也是个资源共享问题,是由 Dijkstra(1965)首先提出并解决的。问题本身是研究一个银行家如何将其总数一定的现金安全地借给若干个顾客,使这些顾客既能满足对资金的要求又能完成其交易,也使银行家可以收回自己的全部现金不至于破产。这个问题同操作系统中的资源分配问题十分相似。如操作系统在若干个并行进程间分配单位数量一定的某共享资源就是这样一个问题,既要使每个进程均能满足其对资源的要求,使之完成其运行任务,同时又要使整个系统不会产生死锁。例如系统共有五台磁带机,已有两台分给了进程 P,一台分给了进程 Q,这两个进程都还需要两台磁带机(假定系统对资源的分配方式是每次只分给进程一个资源单位,即一台磁带机),才能完成其全部运行任务。那么这两台磁带机如何分配,对系统是否会引起死锁关系很大。如果我们把剩下的两台磁带机全部分给其中的一个进程,例如进程 P,于是进程 P 就能完成运行,并最后释放出它所占有的四台磁带机,这样将其中两台分给进程 Q,Q 也终于完成其运行工作,整个系统是安全的。如果我们不是这样地分配剩下的两台磁带机,而是每个进程分给一台磁带机,那么两个进程都要等待对方的一台磁带机释放出来给它,在不可抢占资源情况下,系统成为死锁状态。由此可以看出,资源分配方法不同,或者说采取的分配算法不同,可以导致系统是否成为死锁的不同结果。银行家算法是一个能使系统避免死锁的一个算法。现将此问题描述如下。

一个银行家在若干个顾客间共享他的资金 capital,每个顾客必须在一开始就提前说明他所需借款总额,假如该顾客的借款总额不超过银行家的资金总数,银行家将接受顾客要求。

在顾客交易期间,他无论是向银行家错钱还是归还借款,都以每次一个单位(假定一个单位为一万元人民币)的方式,有时顾客在借到一个单位的借款前可能必须等待一个时期,但是银行家应保证顾客的等待时间是有限的。而顾客的当前借款总数不能超过其开始

时声明的所需最大借款数。

假如银行家能使他的当前的全部顾客在有限的时间内完成他们的交易(于是也归还了他们的借款),那么当前的状态是安全的。反之状态就是不安全的。

我们的任务就是找出一个算法以决定银行家当前的状态是否安全。如果有这样一个算法,每当有顾客提出借款要求时,银行家可以用此算法来决定是借给他还是让他等待别的顾客归还后再借。

假定每个顾客的状态用他的"当前借款总数"loan 和"将来的借款要求数"claim 来表征,显然

$$claim = need - loan$$

其中 need 是其所需的最大借款数。

银行家的状态是以他的总资金 capital 和现行剩余资金总数 cash 来表征:

$$cash = capital - \text{各顾客 } loan \text{ 之和}$$

由于此算法决定每个状态是否安全的过程不是很简单的,所以我们在下一节进一步举例说明其思想,并给出程序的描述。

6.4.2 银行家算法

如图 6.3 所示状态中有三个顾客 A, B, C,共享 10 个现金单位(设一现金单位为一万元),三个顾客共要求 20 个现金单位(图 6.3 中括号内为该顾客要求数 claim,括号外为顾客

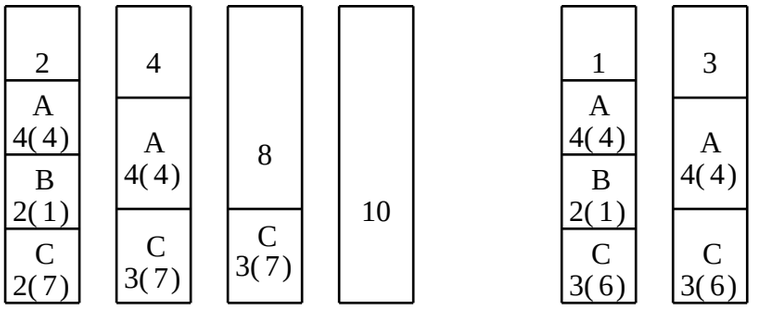


图 6.3 银行家算法状态

已借得的借款数 loan)。在图 6.3(一)的 a 中顾客 A 已借 4 个单位,仍要求借 4 个单位,顾客 B 已借 2 个单位,仍要求借 1 个单位,顾客 C 已借 2 个单位,仍要求借 7 个单位。故剩余现金总数 $cash = 10 - 4 - 2 - 2 = 2$。

本算法逐一检查各顾客中是否有"仍要求数"不超过剩余现金总数的顾客,从图 6.3(一)的 a 中可知,顾客 B 可以完成他的交易并归还其借款,于是系统状态变成图 6.3(一)的 b 所示……,依次进行,直至三个顾客全部完成,归还借款,所以状态(一)a 是安全的。

假如银行家从状态(一)a(图 6.3)出发把一个现金单位分给顾客 C,于是状态变成为(二)a(图 6.3),在这个状态中,顾客 B 可以是安全的,于是变成状态(二)b(图 6.3),在这个状态中,进程 A, C 均不能完成他们的工作,系统状态是不安全的。

银行家算法是从当前状态S出发,逐个检查各顾客中谁能完成其工作(即"仍要求数"不超过剩余现金总数)。然后假定其完成工作且归还全部借款,再进而检查谁又能完成工作,……如果所有顾客均能完成工作,则状态是安全的。

所谓当前状态S是指各顾客的状态与银行家状态(见前节)的总和。在PASCAL语言中,状态S可以用记录来表示。而各顾客的状态用数组表示,每个数组元素对应一个顾客的状态,而每个顾客状态又是一个记录。状态S是本程序所用的主要的数据库。以下给出本算法的程序。

```
The Banker's Algorithm
        type S= record
                transaction: array B of
                        record
                         claim, loan: integer
(顾客将来的借款要求数,当前借款总数)
                         completed: boolean
(顾客状态标志位,用以指示顾客是否已完成工作)
                        end
                capital, cash: integer
(银行家总资金数、现行剩余资金数)
                end
            B= 1.. number of customers(顾客数)
function Safe(current state: S): boolean
    var state: S
  procedure complete transactions(var state: S);
    var customer: B;    progress: boolean
    function completion possible(claim, cash: integer): boolean
      begin
        if claim> cash then
          begin
            completion possible: = false
          exit no
          end
      completion possible: = true;
      end (function completion possible)
      begin
      with state do
      repeat
      progress: = false(控制重复语句执行的本过程的局部变量)
      for every customer do
```

```
            with transaction (customer) do
              if not completed then
(假如该顾客的状态为尚未完成)
                if completion possible(
                claim, cash)then
(调用函数过程 completion possible, 假如结果为真,即该顾客有可能完成交易)
              begin
                cash:  = cash+ loan;(归还借款给银行家)
                Completed:  = true;(标志该顾客状态为已完成)
                progress:  = true;
              end
  until not progress
  end(procedure Complete transaction)
  begin(本算法的程序主体)
      state:  = current state;
      Complete transactions(state);
      if state.capital= state.cash then
          Safe:  = true
  end (function Safe)
```

上述算法中最后部分为程序主体。在主体中调用 Complete transactions 过程,对每个未完成的顾客检查是否有完成可能(for 语句)。这中间又要调用函数过程 Completion possible,如果至少有一个顾客有完成可能,于是该顾客归还借款,改状态标志为真,并且把 progress 也置为真。这样,repeat 语句继续执行,并对所有未完成顾客再进行检查完成的可能,……一直到 progress 为假,才停止执行 repeat 语句。然后检查银行家是否已收回全部资金。如果收回了全部资金,则系统为安全,否则为不安全。若系统状态为安全,我们可把资源按此算法分配给用户。

本节中所提出的银行家算法是基于 Habermann(1969)的算法,并作了某些简化,以便于读者阅读。主要的简化之处是本算法每次只考虑用户对一类资源所提出的要求,而 Habermann 算法可以同时考虑用户对多类资源所提出的要求。但如果读者有了对本节算法的了解后,再扩充为对多类资源的算法是不困难的,读者可将此作为练习,试做一下。

6.4.3 银行家算法的优缺点

由上面的银行家算法中可以明显地看出,作为资源分配的一种算法来说,它允许死锁必要条件中的互斥条件、部分分配条件和不可抢占条件存在,这样它比起预防死锁的几种方法(见 6.3 节)来说,限制条件放松了,资源利用程度提高了。同时由于系统可以满足、也可以拒绝进程提出的资源要求,当一个进程的资源要求将导致不安全状态时,系统就拒绝其要求,直到该资源要求不会导致不安全状态时,才满足此进程的资源要求(这主要由于其它进程释放出了资源)。这样系统总是处于安全状态。

但此算法有着以下一些缺点:

(1) 本算法要求被分配的每类资源的数量是固定不变的,但这是难以做到的,因为资源本身需要维护,可能损坏;

(2) 本算法要求用户数也保持固定不变,这在今天多道程序系统中也是难以做到的,用户数是经常变化的(尤其在分时环境中);

(3) 本算法保证所有用户的要求在有限时间内得到满足,但实时用户要求有更好的响应;

(4) 本算法要求用户事先说明他们的最大资源要求,但这个要求对用户来说不但是不方便的,而且往往也是困难的。

*6.5 死锁检测

死锁预防和死锁避免都是不让系统成为死锁,但往往为了破坏死锁的必要条件而对资源分配提出一些限制,从而降低资源利用率。有些系统往往着眼于提高资源利用率,允许死锁存在的必要条件发生,也未采取避免死锁算法。因而这些系统就有可能发生死锁。死锁检测就是实际地检查一个系统中是否存在死锁,并且标定出哪些进程和资源被牵涉在死锁中。因而死锁检测算法常被使用在允许前三个死锁必要条件(6.2节)存在的系统中。而死锁检测算法主要是检查系统中是否存在循环等待条件。

6.5.1 资源分配图

图论是被使用在许多领域中解决问题的强有力的工具。在操作系统中,人们按照图论的方法把进程和资源的关系用一个有向图来描绘,并通过研究图形的性质来解决死锁等有关问题。有关这方面的详细资料读者可参考文献[2]。

本节中我们介绍资源分配图,它主要描述进程、资源以及它们之间的关系。我们用大圆代表某类资源,大圆中的小圆代表该类资源的一个资源单位,大圆中有几个小圆就表示有几个资源单位。用方框代表进程。用有向边代表进程对资源的要求和资源的分配,如果有向边从资源出发指向进程,表示分给该进程一个资源单位;如果有向边从进程出发指向资源则表示该进程要求该资源的一个资源单位,图6.4表示了资源分配图。在图6.4(a)中,进程P_1要求资源R,但R已分给了进程P_2。在图6.4(b)中表示了一个小的死锁系统,P_1要求资源R_1,而R_1已分给了进程P_2,进程P_2还要求资源R_2,但R_2已分给了进程P_1。这就是循环等待条件的一个具体例子,所以循环等待条件就是资源分配图中的一个环路。

6.5.2 资源分配图的化简

最常用的检测死锁的方法就是对资源分配图进行化简。所谓化简是指一个进程的所有资源要求均能被满足的话,那么我们设想这个进程得到其所需之所有资源,从而该进程的工作就能不断取得进展,直至最后完成全部运行任务,并释放出全部资源。那么我们说,

图 6.4　资源分配图

该资源分配图可以被这个进程所化简。在资源图上化简的具体办法是移走所有从资源出发指向该进程的有向边,和从该进程出发指向资源的诸有向边(即假定所有被该进程要求的资源均得到满足,最后进程释放所有已分得之资源)。于是该进程成为一个孤立的节点(见图 6.5)。

假如一个资源分配图可以被其上的所有进程所化简,那么称该图是可化简的。假如该图不能被其上所有进程所化简,则称该图是不可化简的。

根据死锁理论可以证明如下的死锁定理:

当且仅当系统某状态 S 所对应的资源分配图是不可化简的,则 S 是死锁状态,而不可化简的那些进程即是被死锁的进程。反之,若状态 S 的对应资源分配图是可化简的,则状态 S 是安全的(参考文献[2])。

图 6.5 是可化简的,所以系统状态 S 是安全的,而图 6.4(b)是不可化简的,所以对应的系统状态是死锁的,P_1,P_2 是死锁进程。

根据死锁理论还可以证明,资源分配图化简的结果与化简顺序无关,最终结果是相同的。

详细的死锁检测算法及其程序请参考文献[2],本书不作介绍。

6.5.3　资源分配图化简的实现

为在计算机中具体实现对资源分配图的分析和判断,首先需要把资源分配图用程序设计语言中相应的数据结构来表示,而后给出化简算法的程序描述。要把资源分配图用相应数据结构表示,首先要对资源分配图作一些修改,前面我们规定进程要求某资源的每一个资源单位都要用一个从进程出发指向该资源的有向边来表示,这样如果进程要求同类资源的两个以上的资源单位,就要有两个以上的有向边。同样如果某资源的多个资源单位分配给一个进程,那就要有多个从资源出发指向进程的有向边。为此,我们把资源分配图上资源和进程之间的若干个分配有向边和请求有向边压缩成一个单边和一个与其相关联的给出其单位数的权的方法来表示,以节省存储空间。这样一个资源分配图既可用矩阵表

图 6.5　资源图的化简

示,亦可用表格来表示,现分述如下。

6.5.3.1　矩阵表示法

假定系统中有 n 个进程记为 $P_1, P_2, \cdots, P_n$,和 m 种类型的资源记为 $R_1, R_2, \cdots, R_m$,各类资源的资源数目分别为 $w_1, w_2, \cdots, w_m$。这样在任一时刻 t 的资源分配图可以用两个 n × m 的矩阵来表示:

(1) 分配矩阵

$$A(t)=\begin{vmatrix} a_{11} & a_{12} & \cdots\cdots & a_{1m} \\ a_{21} & a_{22} & \cdots\cdots & a_{2m} \\ a_{31} & a_{32} & \cdots\cdots & a_{3m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots\cdots & a_{nm} \end{vmatrix}$$

其中每个元素 $a_{ij}=$ 资源(R_j, P_i)($i=1,2,\cdots,n$, $j=1,2,\cdots,m$) 表示分配给进程 P_i 的 R_j 类资源单位数目。这个矩阵实际上代替了资源分配图中的所有资源分配的有向边及表示其单位数的权，每个元素 a_{ij} 的值就是权重。若 $a_{ij}=0$ 表示资源 R_j 和进程 P_i 间没有资源分配有向边，或者说资源 R_j 尚未分配给进程 P_i。

(2) 请求矩阵

$$B(t)=\begin{vmatrix} b_{11} & b_{12} & \cdots\cdots & b_{1m} \\ b_{21} & b_{22} & \cdots\cdots & b_{2m} \\ b_{31} & b_{32} & \cdots\cdots & b_{3m} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots\cdots & b_{nm} \end{vmatrix}$$

其中每个元素 $b_{ij}=$ 资源(P_i, R_j)($i=1,2,\cdots,n$, $j=1,2,\cdots,m$) 表示进程 P_i 请求资源 R_j 的数目。这个矩阵实际上代替了资源分配图中的所有进程对各类资源提出资源要求的有向边及表示其要求资源单位数的权，每个元素 b_{ij} 的值就是权重。若 $b_{ij}=0$ 表示进程 P_i 和资源 R_j 间没有资源要求有向边，或者说进程 P_i 不要求资源 R_j。

除此之外，系统为了便于对资源分配图进行化简，还要维持两个矩阵：

资源数目矩阵 $$W=(W_1, W_2, \cdots, W_m)$$

其中 W_j 表示资源 R_j 的总的资源单位数。

资源可用单位数矩阵 $$V(t)=(V_1, V_2, \cdots, V_m)$$

其中 V_j 表示资源 R_j 在时刻 t 时的可用资源单位数目。显然

$$V_j=W_j-\sum_i a_{ij}$$

或者

$$V_j=W_j-(\sum_i R_j, P_i)$$

6.5.3.2 链表表示法

把分配给某进程 P_i 的资源都链接到 P_i：

$$P_i\rightarrow(R_x, a_x)\rightarrow(R_y, a_y)\rightarrow\cdots\rightarrow(R_j, a_j)\rightarrow\cdots\rightarrow(R_z, a_z)$$

($i=1,2,\cdots,n$)

其中 R_j 是一个资源结点，a_j 是权，即 $a_j=$ 资源(R_j, P_j)

相似地，把请求资源 R_j 的进程也链接在一起：

$$R_j\rightarrow(P_u, b_u)\rightarrow(P_v, b_v)\rightarrow\cdots\rightarrow(P_i, b_i)\rightarrow\cdots\rightarrow(P_w, b_w)$$

其中 b_i 是进程 P_i 对资源 R_j 请求的权重，即 $b_i=$ 资源(P_i, R_j)($j=1,2,\cdots,m$)。

6.5.3.3 检测算法的执行速度

简单的检测方法是按次序审视进程请求矩阵(或链表),尽可能地进行化简,直到不可能化简为止。这种检测方法的检测速度是比较慢的,最坏的情况是当进程按 $P_1, P_2, \cdots, P_n$ 排序,而仅有的化简次序为 $P_n, \cdots, P_1$ 时。在这种情况下,当每个进程要请求所有的 m 种资源时,于是,审视进程的次数为 $n+(n-1)+\cdots+1=n(n+1)/2$。由于每次审视都要求检查这 m 种资源,这样,最坏情况的执行时间正比于 mn^2。

要是对这些要求附加上一些信息,则可以得到一个更为有效的算法。即对每个进程节点 P_1 附加上一个"等待计数" w_i,它是由在某一时刻引起该进程阻塞的资源数(不是资源单位数,是指资源种类数)所构成,另外我们还把进程对每类资源的请求链按请求数多少,由小到大进行排序。对资源分配链也按进程分得该类资源数多少,由小到大进行排序。该算法先把那些已满足了其全部资源请求的进程(即 $w_i=0$)记入一个表 L 中,然后依次假定该表 L 中每一进程完成运行并释放出所分得的所有资源。每释放出一个资源都检查是否有需要该资源的等待进程,若有,就修改其等待计数 w_i。若 $w_i=0$,则表示该进程所请求的资源已全部得到满足,于是也将该进程记入表 L 中。重复上述操作。最终若所有进程都被记入 L 表中,,则系统的初始状态是安全的,否则 S 是死锁状态,而未被归入 L 表中的那些进程是死锁进程。该算法的最大执行时间正比于 mn。现将算法描述如下:

```
        L:= {Pi©wi= 0};
    for all P∈L do
begin for all Rj ∈ ©(Rj, Pi©) 0 do
    begin
            vj:= vj+ ©(Rj, Pi);
            for all Pi 0≤ ©(Pi, Rj)©≤vj do
        begin
            wi:= wi- 1;
      IF wi= 0      then L:= L∪{Pi}
      end
    end
end

          Deadlock:= (L= {P1, P2, ⋯, Pn})
```

应当指出,上述算法只提供了死锁的必要条件,但不是充分条件。

6.6 死锁的恢复

一旦系统成为死锁,我们就要消除死锁,使系统从死锁中恢复。当前系统中所使用的死锁恢复办法有两种。第一种是强制性地从系统中撤消进程并剥夺它们的资源给剩下的进程使用。毫无疑问,被撤消进程前面已完成的工作全部损失了。有时为了使系统中有充分的资源可利用以消除死锁,常常要撤消好几个进程,这里存在一个按照什么原则进行撤消的问题。目前较实用而又简便的方法是撤消那些代价最小的进程。所谓一个进程的撤

消代价可以是:

(1) 进程的优先数;

(2) 根据系统中的会计过程所给出的运行代价,即重新启动它并运行到当前撤消点所需要的代价。

(3) 作业的外部代价,即与此进程相关的作业类型,如学生作业,行政管理作业,生产作业,研究作业和系统程序作业等,都可以有其相应的固定撤消代价。

总的来说,撤消法不是很理想的方法,它类似于用切掉手指的办法来治疗手指的一切病症。

第二种方法是使用一个有效的挂起和解除挂起机构来挂起一些进程,其实质是从被挂起进程那里抢占资源以解除死锁。许多学者认为在此领域的研究工作比起其他死锁恢复办法更为重要。例如,假如一个系统提供了检查点和重新启动的便利的话,我们可以暂时停止一个系统,而后从各进程的最近的检查点(系统保留了检查点的状态)逐次地重新启动。这既可从死锁中恢复,又使进程的损失最小(从检查点以后的损失)。但许多系统还没有此类功能,在 IBM 4300 系列等机器上提供了此类功能。

习　　题

6-1　何谓死锁? 给出一个仅涉及一个进程的死锁例子。

6-2　给出一个涉及三个进程和三个不同资源的死锁例子,并画出相应的资源分配图。

6-3　试述产生死锁的原因和必要条件是什么?

6-4　某系统有同类资源 m 个,被 n 个进程共享,请分别讨论当 $m > n$ 和 $m \leq n$ 时每个进程最多可以请求多少个这类资源,才能使系统一定不会发生死锁?

6-5　某系统中有六台打印机,N 个进程共享打印机资源,每个进程要求两台,试问 N 取哪些值时,系统才不会发生死锁?

6-6　设有两个进程 A 和 B 各自按以下顺序使用 P,V 操作并行运行(S_1 和 S_2 代表系统中一台打印机和一台扫瞄仪资源信号量):

A 进程	B 进程
⋮	⋮
$P(S_1)$	$P(S_2)$
⋮	⋮
$P(S_2)$	$P(S_1)$
⋮	⋮
$V(S_2)$	$V(S_1)$
⋮	⋮
$V(S_1)$	$V(S_2)$
⋮	⋮

(1) 分析各种推进速度可能引起的情况,并画出与图 6.1 类似的图形表示;

(2) 用死锁的必要条件说明产生死锁和不产生死锁的原因。

6-7　用银行家算法判断下述每个状态是否安全。如果一个状态是安全的，说明所有进程是如何能够运行完毕的。如果一个状态是不安全的，说明为什么可能出现死锁。

状态 A

	占有台数	最大需求
用户 1	2	6
用户 2	4	7
用户 3	5	6
用户 4	0	2
可供分配的台数		1

状态 B

	占有台数	最大需求
用户 1	4	8
用户 2	3	9
用户 3	5	8
可供分配的台数		2

第 四 部 分

主存储器管理

第 7 章　实 存 储 器 管 理 技 术

7.1　引　　言

主存储器(又称内部存储器,处理机存储器)的管理一直是操作系统最主要功能之一,受到人们高度重视。这是由于早先的存储器价格比较昂贵、主存容量有限。今天虽然主存价格已相当便宜,但主存容量仍然是计算机四大硬件资源中最关键而又最紧张的"瓶颈"资源。因此对主存的管理和有效使用仍然是今天操作系统十分重要的内容。许多操作系统之间最明显的区别特征之一往往是所使用的存储管理方法不同。如 OS/360-MFT 采用固定分区存储管理技术, OS/360-MVT 是采用可变分区存储管理技术, OS/2, Windows NT, DOS/VSE 是采用虚拟存储管理技术。

主存储器管理技术可分为两大类:实存储器管理和虚拟存储器管理。本章研究常用的几种实存储管理技术。第八章研究虚拟存储管理技术。

7.1.1　主存储器的物理组织、多级存储器

整个计算机系统的功能很大程度上取决于主存储器(以下简称主存)的结构组织和实现方法,它是同各种存储管理技术紧密相配合的。就主存的功能而言,首先是存放系统和用户程序的指令和数据,每一项信息都存放在主存的特定位置上。信息在主存是按"位"存放的。为了能对信息进行访问,要对这些位置进行编号,这些编号称为地址。以什么为单位进行编址呢?早期的计算机中,存储器是按字组织,每个字由若干"位"组成(不同计算机字长不同),每个字分配一个地址。目前多数计算机以字节为单位进行编址,每个字节由 8 位组成, IBM 还规定一个字为四个字节, PDP 中两个字节组成一个字。

为了能更多的存放并更快地处理用户信息,目前许多计算机把存储器分为三级(图7.1),用户的程序在运行时应存放在主存中,以便处理机访问。但是由于主存容量有限。所以把那些不马上使用的程序、数据放在外部存储器(又称次级存储)中。当用到时再把它们读入主存。图7.1中的三级存储器,从高速缓冲存储器(简称缓存)到外部存储器(以后简称外存),其容量愈来愈大,一般每级之间相差几个数量级(如M68020的缓存为2KB,IBM-4341的缓存为8KB,主存可达几兆)。而访问数据的速度则愈来愈慢,价格也愈来愈便宜,如IBM的缓存的最大传输速度为每双字120～225毫微秒,主存的传输速度每字1微秒。

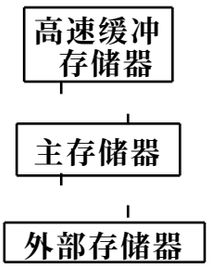


图7.1　多级存储组织

7.1.2　主存储器管理中的研究课题

早在单道程序阶段,人们就感到主存容量不敷需要,并研究了覆盖技术来解决用户作业空间大于实际的主存空间的矛盾。在多道程序系统出现后,主存容量不足的矛盾更为突出。由于多道程序共享主存,所以对主存的管理工作又出现了如何在各程序间分配主存空间的问题。同时还要考虑如何防止各程序有意无意地互相干扰和破坏的问题。再者无论是用户程序还是系统程序均由操作系统调度,并分给主存存放其数据和程序。正如在第2章2.2.2节所述,这些程序必须是相对编址的可浮动的程序。于是,程序被装入主存时就需重定位——把相对地址转换为主存中的绝对地址。

综上所述,目前关于主存储器管理的主要研究课题归纳为四个方面:

(1) 主存分配:这是存储管理研究的主要内容,也是本章讨论的重点。在本章中将研究各种主存分配算法,以及每种算法所要求的数据结构,但不涉及某个具体的存储管理系统的程序。读者只要掌握了算法,了解其数据结构,那么编写一个程序模块就很容易了。众所周知

算法+ 数据结构= 程序

(2) 地址映象或重定位:主要研究各种软件和硬件的地址转换技术和机构;

(3) 存储保护:研究如何保护各程序区中信息不被破坏和偷窃;

(4) 存储器扩充:这里所指的扩充并非指硬件设备上的扩充。而是用存储管理软件来实现逻辑上的扩充——即所谓的虚拟存储技术,这将在第8章研究。

7.2　固 定 分 区

固定分区存储管理技术的基本概念是把主存分成若干个固定大小的存储区(又称存储块)。每个存储区分给某一个作业使用。直到该作业完成后才把该存储区归还系统。

固定分区存储管理技术通常可分为单道作业和多道作业两种情况。在微型计算机中,目前较为简单的计算机系统(如Z-80,长城0520,IBM-PC等使用CP/M操作系统)的存储管理均采用单用户、单道作业固定分区的技术,而在早期的大、中型计算机(如IBM360)曾采用多道作业的固定分区的技术。下面分别简述如下:

(1) 对于单用户情况下,采用固定分区技术的主存分配情况如图7.2所示。通常将主

存分成两个(图 7.2b)或三个(图 7.2a)大的主存区。以 CP/M-80 为例,低地址区的 256 个字节作为系统区,主要放置中断向量(又称中断指针),系统缓冲区以及系统与用户程序的公用区。其它各区的分配如图示。为了防止用户有意或无意中破坏系统信息,可以使用界限寄存器来保护主存中的系统信息。

(2) 多道作业的固定分区技术如图 7.3 所示。通常主存区分成固定大小的若干块。除操作系统所占据的部分外,其余均分给用户使用,主存中不但分区的区数是固定不变的,而且每区的大小也是固定不变的。主存分配是分给每个作业一块足够大(应大于、等于作

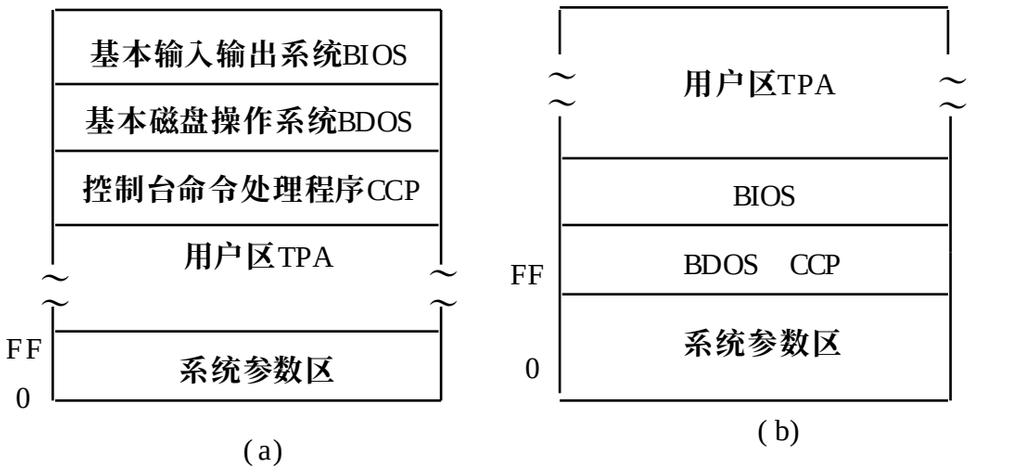


图 7.2 CP/M主存分配

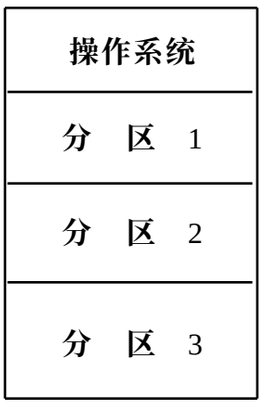


图 7.3 固定分区

业大小)的主存分区,不允许两个作业同时放于同一个分区中。由于分区往往大于作业,所以分区中常有未用的、剩下的空闲部分称为存储碎片(或分区内的碎片),它降低了主存的利用率。

7.2.1 数据库

在多道固定分区情况下,操作系统的存储分配模块和存储释放模块都要用到关于主存分区情况的说明信息,以及这些存储区的使用状况信息——即存储管理的数据库,常被称为存储分块表(记为 MBT 表),图 7.4 中给出了存储分块表所应包含信息的典型例子。

区 号 (存储键)	大小	位置	状态
1	8K	512K	正使用
2	32K	520K	正使用
3	32K	552K	未使用
4	128K	584K	未使用
5	512K	712K	正使用

图 7.4 存储分块表MBT

该例把主存分成五个分区(存储块)。在此存储分块表中,实际包含三项信息:

(1) 大小: 是指该存储块的大小,以字节为单位;

(2) 位置: 指该存储块在主存中的起始地址;

(3) 状态: 表明该存储块是否已被使用。

还有一项信息"区号"未包含在表格之内。在 IBM 系统中,区号实际上起存储键(见 2.1.2 节)作用,分给用户使用那个存储区,该区在存储分块表中的序号即作为分给用户的存储键使用。

存储分块表是存储分配和释放这两个模块的数据库。实际上在主存中并没有一个如同图 7.4 形状的表格,这里只是为了给出示意性的说明。在程序中这个表用相应语言的数据类型来表示。在 PASCAL 语言中此表用一维数组表示,数组元素是记录结构,数组的下标即是分区的序号(存储键)。用 PASCAL 语言的数据结构来描述图 7.4 如下:

```
type
    N:integer;
  Entry= record
        size:integer;
        location:integer;
        status:boolean;
        end
var MBT:array〔1..N〕of Entry;
```

按此描述,操作员在系统初始化时,输入相应数据,生成存储分块表。主存的分区个数和每个分区大小只能由操作员改变,用户不能改变此表。操作员根据这一时期用户作业大小的统计资料加以变化,不应随意变动。故存储分块表 MBT 应放在操作系统区内。

7.2.2 存储分配算法

固定分区和可变分区的存储分配算法一般有以下三种:

(1) 最佳适应法: 从所有未分配的分区中挑选一个最接近作业尺寸且大于或等于作业大小的分区分给要求的作业。从而使分区内浪费的未用部分(又称碎片)最少;

(2) 最先适应法: 即按分区序号从存储分块表的第一个表目起查找该表,把最先找到的且大于或等于作业大小的未分配分区分给要求的作业。这种分配算法的着眼点是在于缩短表格查找时间;

(3) 最坏适应法: 从所有未分配的分区中挑选最大的且大于和等于作业大小的分区分给要求的作业。此种方法可用于可变分区分配技术中。

对于固定分区存储管理技术所使用的分配算法的要求是查表时间要短,且分区内的碎片浪费要最少。所以最好是最佳适应法和最先适应分配算法的结合。为此存储分块表 MBT 中的各分区应按分区大小排序,最小分区放在表头。其分配和释放模块的框图作为练习,由读者完成。

7.2.3 存储保护与重定位

为防止其他用户的程序和系统信息不被在处理机上运行的用户程序所破坏,所以存储管理技术中必须有存储保护功能。最简单的方法可用一对“界地址寄存器”(见2.1.2节)。每当一个作业被作业调度程序调入主存时,操作系统的作业调度程序为其分配所需的主存空间,并将该作业的信息(如作业名,作业大小,在主存的起始地址等)登记入作业表中。当该作业(或进程)被分配给处理机运行时,操作系统的调度程序还同时从主存中的作业表(或进程的PCB)中,把该作业的大小和主存起始地址二项数据送入处理机的一对界地址寄存器中去。每当处理机要访问主存某单元时,系统硬件自动将该单元地址与界限寄存器的内容进行比较,以判断此次访问是否合法;如果该单元地址是在界限寄存器对所限定的地址范围内,则此次访问合法。否则将产生一个越界中断(属程序中断)通知系统处理。

固定分区中的重定位方法,是由连接装入程序来完成的,即采取静态重定位方法(见2.2.2节)。

7.2.4 优缺点

固定分区存储管理技术的最大优点是简单,要求的硬件支持只是一对界地址寄存器,软件算法也很简单。缺点是主存利用率不高。

7.3 可变分区的多道管理技术

因为固定分区主存利用率不高,使用起来不灵活(分区大小固定死了),所以出现了可变分区的管理技术。所谓可变分区是指主存事先并未划分成一块块分区,而是在作业进入主存时,按该作业的大小建立分区,分给作业使用。这种可变分区方法有何特点呢?首先分区个数是可变的,同时每个分区大小也是不固定的。在系统初始启动时,整个主存除操作系统区以外的其余主存区可以看成是整个一个分区[图7.5(a)]。随着作业一个个被调

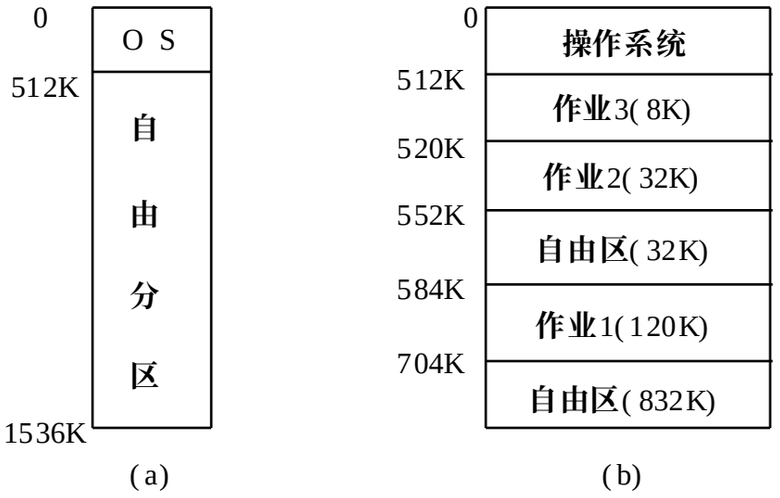


图 7.5 可变分区主存使用情况

入主存运行,并且分给它们一个相应于作业大小的主存分区使用,直到作业完成后才释放出其所占之主存分区。由于各作业大小和完成的时间是各不相同的,这样经过一段时间后,主存就由原来一个完整分区而变成了棋盘似的情况,被分割成了多个分区,这些分区中有些分区被作业占据使用,有些分区却是空闲的[图 7.5(b)]。这些空闲分区有时称为自由分区或碎片。所以可变分区方法的第二个特点是主存中分布着个数和大小都是变化的自由分区或碎片,这些自由分区有些可能相当大,有些则相当小。因此如何管理这些分区(已分配的和自由分区),如何实现可变分区存储分配算法,以及如何消除那些小得难以使用的碎片是我们要仔细加以研究的问题。微型计算机 PDP-11 的 UNIX 系统所使用的存储管理技术就属于此种类型。

7.3.1 数据库

可变分区存储管理中各功能模块要用的数据库可以有以下几种组织方法:

(1) 存储分块表: 此存储管理技术中仍然可以使用固定分区方法中所讲的存储分块表结构,但这表格存在两个缺点:

① 由于分区个数是变化的,所以表长不好确定,造成表格管理上的困难。如果给该表留的空间不足,则无法登记各分区情况。如果留的空间过大,造成浪费。

② 分配主存时,为查找一块合适的自由分区所需扫描的表目增加,因为整个表的长度增加了(其中分别包含有很多已分分区和自由分区),所以查找速度慢了。

(2) 分开设置两个存储管理表: 已使用分区表(记为 UBT)和自由分区表(记为 FBT)来分别登记和管理系统中的已分分区和自由分区。这样可以减少存储分配和释放时查找表格的长度,提高查找速度。这两个表格的形式如图 7.6 所示[此图对应于图 7.5(b)]。

区号(存储键)	大小	位置	状态
1	8K	512K	已分
2	32K	520K	已分
3	32K	552K	已分
4	120K	584K	已分
5	-	-	空表目
…	…	…	…

已分分区表 UBT

区号	大小	位置	状态
1	32K	552K	自由
2	-	-	空表目
3	-	-	空表目
4	-	-	空表目
5	-	-	空表目
…	…	…	…

自由分区表 FBT

图 7.6 可变分区的数据库

图中每个表仍然包含三种信息:“大小”、“位置”是指该分区的大小和在主存的起始地址,“状态”是分别指两个不同对象的状态(这两个对象是分区和表格本身的表目)。两个表格中共有三种状态: 状态“已分”和“自由”是指分区本身是否已被使用; 而状态“空表目”却是指表中(UBT 和 FBT)该表目的状态,所谓空表目是指该表目中没有登记分区的信息,当需要表目来登记分区信息时,可使用状态为“空表目”的表目。

这两种表中的各分区如何安排先后次序呢? 对自由分区表 FBT 来说,它的排序方法要同所使用的分配算法相适应。当采用最先适应法时可按分区位置排序。采用最佳适应法时,可按分区尺寸由小到大排序。采用最坏适应法时,可按分区尺寸由大到小排序。所以每当有作业释放分区时,对自由分区表还有表格维护(移动表目中的内容使自由分区按顺序排在表中)工作。对已分分区表 UBT 来说,不存在排序和维护表格问题。因为它的区号通常就是存储保护键,不能随意变动。在 PDP-11 的 UNIX 系统中用此种表格来管理所

有的自由分区,称主存可用资源表,该表共有 50 个表目,按地址排序,采用的分配算法为最先适应法。

(3) 自由存储块链:上述表格的语言表示一般可用数组。可变分区存储管理的数据库,实际上广泛地使用“链指针”来把所有的自由分区链结在一起,构成一条自由存储块链。图 7.7 中给出了一个自由存储块链的实例。其实现方法是把每个自由存储块的起始

图 7.7　自由存储块链

的若干个字节分为两部分:前一部分作为链指针,它指向下一自由存储块的起始地址,后一部分指出本自由存储块的大小。系统中用一固定单元作为自由存储链的头指针用以指出该链中的第一块自由存储块的起始地址。最后一块自由存储块的链指针中放着链尾标志(如 0)。链中各块的组织也要与使用的算法相适应。当使用最佳适应法时,链上各块应按分区大小,由小到大地链接起来。以自由存储块链作为数据库的方法,使得数据库的管理和维护均比较简单。

7.3.2　分配和释放算法

存储分配的三种算法已在 7.2.2 节中讨论了。在可变分区的存储管理中,无论数据库用哪种方式,当系统把一个自由分区分给作业时,该分区的大小可能大于作业大小。于是分区被分成两部分,前面部分按作业大小分给作业使用、剩下部分作为自由分区仍然放在自由存储块链(或自由存储块表)中。当一个作业释放一个分区时,需把与该分区相邻的所有自由分区合并成一个新的自由分区放入自由存储链(或表)中。本节作为一个例子给出算法的框图,其数据库按自由存储块链方式进行组织。

显然,系统中很可能有多个进程同时请求分配或释放主存块,于是它们可能同时调用存储管理的分配模块和释放模块,同时查找并修改存储管理数据库 UBT 和 FBT 两张表格或自由存储块链。所以为了保证存储管理数据库中数据的完整性和正确操作,对表格的访问应互斥进行。为此我们为该数据库——临界资源——设置一个互斥信号量 Mut,其初值为 1。另外为主存资源设置一个进程等待的信号量 Memory,其初值为零。下面给出“请求一主存块”和“释放一主存块”的两个程序模块框图如图 7.8,图 7.9 所示。

图 7.8　请求一主存块程序框图

7.3.3　存储器的紧缩和程序的浮动

7.3.3.1　碎片问题和存储器的紧缩

在可变分区存储管理中,由于各作业请求和释放主存块的结果,因此主存中经常可能出现大量离散型的碎片(小的自由分区)问题,这些碎片就像棋盘上的格子,小得难以利用。图 7.10 中就给出了这样一个例子。作业 6 要求进入主存运行。作业 6 的大小只有 60K,而主存各自由分区之和却有 82K,但由于每一碎片大小均小于该作业,因而作业 6 无法进入主存运行。这不但降低了多道程度,还造成了主存空间的大量浪费。

如何解决碎片问题呢? 可以从两个方面想办法:

(1) 把程序分成几部分装入不同的分区中去。也就是改变我们一直把程序作为一个连续的整体在主存中存放的要求。无疑这可以改变碎片问题,但带来了程序管理和执行上的复杂性。这将在多重分区和虚拟存储管理中研究。

(2) 把小碎片集中起来使之成为一个大分区。这自然也能解决碎片问题,但如何使这些碎片集中起来呢?实现的方法只能是移动各用户分区中的程序,使他们集中于主存的一端(顶部或底部),而使碎片集中于另一端,从而连成一个完整的大分区。这种技术通常称

图 7.9　释放一主存块程序框图

为存储器的“紧缩”或“澄清”,因为它类似于将水中油滴澄清成一片油层。

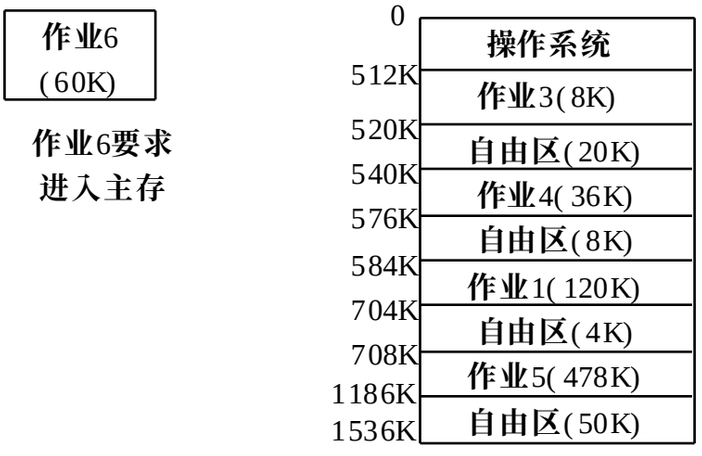


图 7.10　可变分区主存中碎片情况

7.3.3.2　程序浮动

要进行主存的压缩,就要将主存中用户程序移动(又称为浮动)。但移动程序会带来如同 2.2.2 节中所讲的同样问题——程序无法正确执行。如要使程序在移动后仍然能正确

执行,则程序中所有与地址有关的项要按照移动后的新的基地址重新进行程序的重定位工作(参看 2.2.2 节中静态重定位部分)。但这里不是没有困难的。首先这个重定位工作由谁来做?由操作系统来做。但操作系统无法识别程序中哪些项与地址有关,它没有重定位词典,也没有谁能给它以说明。所以在一般机器上操作系统无法承担此重定位工作。但像 Burroughs 5500 和 6700 等机器上为每个字增加两位用来指示值的类型(如 00= 整型,01= 浮点数,10= 字符,11= 地址指针),这些位是由硬件自动置上的,对一般用户程序是透明的。这样操作系统就能通过考查这些硬件附加位来识别与地址有关的项,并能进行相应的重定位工作。但这种方法不适用于没有这种硬件设施的大多数计算机系统。一般计算机系统也可以用另外一种移动程序的办法:由连接装入程序将所有需要移动的程序按紧缩后的新地址重新装入。但是这样重新装入的程序不能从程序断点开始继续运行。因为各寄存器中的内容无法判断是否与地址有关,所以无法对它们进行相应的修改。故必须从头开始重新运行这些程序。这带来两个问题,首先浪费了很多作业的已运行过的机时。其次是有些程序不允许重新运行(如工资管理程序重新运行的结果会使前面的一些人得到双份工资)。

要想比较好的解决碎片问题以及使程序可浮动的最好的办法是采用动态重定位技术。

7.3.4 动态重定位的可变分区多道管理

7.3.4.1 动态重定位

在 2.2.2 节,我们提出了重定位的概念,并且按重定位时机分为静态重定位和动态重定位。所谓动态重定位是指程序的重定位时机不是在程序执行前进行,而是在程序执行过程中才进行地址转换(由相对地址转换为主存绝对地址,这又称地址映象)。更确切地说是在每次访问主存单元前才进行地址转换。其具体过程如图 7.11 所示。

首先将用户按相对地址编址的目标程序[图 7.11(a)]原封不动地装入主存中分给该用户使用的分区[图 7.11(b),分区起始地址为 10000]中,所谓原封不动是指装入时,用户目标程序中与地址有关的各项均保持原来的相对地址不进行修改(如图中 LOAD 指令)。当该用户程序被调度到处理机上执行时,操作系统自动将该用户作业的起址(该分区起址)10000 由作业表中取来,并将分区起始地址减去用户目标程序(编址或编译时所使用的)的相对基地址[图 7.11(a)中该基地址为 0],然后将其减得值装入定位寄存器中。当处理机要访问主存时,地址转换硬件自动将程序中的相对地址与定位寄存器中的内容相加,并按相加的和作为主存绝对地址去访问数据。如本例中 LOAD 指令是要把相对地址为 1000 中的数据 0157 取到 1 号寄存器。当 CPU 执行此指令时,硬件自动将相对地址 1000 与定位寄存器中的 10000 相加而得 11000,然后以 11000 作为绝对地址把放在其中的数 0157 取到 1 号寄存器中。因此读者可以看到,动态重定位的时机是在执行指令过程中,每次访问内存前动态地进行。采取动态重定位后,由于目标程序装入主存后不需修改地址指针及所有与地址有关的项,因而程序可在主存中随意浮动而不影响其正确执行。这样,就可以方便地进行存储器紧缩,较好地解决了碎片问题。

图 7.11　动态地址转换及定位寄存器

7.3.4.2　动态重定位的硬件支持、软件算法

只有具有动态重定位硬件机构的计算机系统,才有可能采取动态重定位可变分区多道管理技术。即需要有一定的硬件支持条件。这硬件支持包括定位寄存器和加法器。为了实现存储保护技术仍然可以采取一对界地址寄存器。当用户程序按相对于零地址作为相对基地址来编写(或编译)目标程序时,可以把存储保护的界地址寄存器对中的基址寄存器作为定位寄存器使用。

动态重定位可变分区管理技术的分配算法和数据库基本上与可变分区存储管理中所介绍的相同。特殊之处是何时进行存储器紧缩,有两种不同的解决办法:

(1) 在某个分区被释放后立即进行紧缩。所以系统中总是只有一个连续的自由分区而无碎片。这对自由分区的表格管理和分配自由块都将变得非常容易,但是紧缩工作是很花费机时的。例如我们以每秒一百万字节的速度移动分区,那么紧缩一个 1024K 字节的存储器中的一个小分区平均要花费半秒钟的时间。

(2) 当"请求分配模块"找不到足够大的自由分区分给用户时再进行紧缩。这样紧缩的次数比上述方法要少得多,但表格管理复杂了。图 7.12 中给出了这种方法的存储分配法框图。图中略去了对互斥和等待信号量的操作,作为练习请读者自行添上。其释放模块框图同图 7.9 所示。

7.3.4.3　IBM-PC 等微型计算机的存储管理与地址变换机构

当前不少的微型计算机,如 IBM-PC, Intel 8086, 8088 和长城 0520 以及 PDP-11 等计算机所采用的存储管理技术,实际上都是属于多道可变分区的存储管理技术(尽管都称为分段管理,但并不是在本书中虚拟存储技术中所指的那类分段管理技术)。并且它们都具有动态重定位机构的硬件设施,因而也就具有了可对主存中的存储碎片进行紧缩的功能。除此之外,这些计算机还具有以指令中的 16 位地址长度(仅能表示 64K 范围)对一兆主存的地址空间进行寻址的能力。那么在这些机器中这些功能是怎样实现的呢,程序中的相

图 7.12　动态重定位可变分区分配算法

对地址又是如何变换成主存的绝对地址呢。

原来像 Intel 8086 等计算机的 CPU 具有 20 位地址总线,能直接访问 1MB 主存空间。但 CPU 内部有关地址的寄存器都是 16 位的,16 位地址的寻址能力只有 64KB,CPU 之所以能对 1 MB 主存进行寻址是由于它的作业的相对地址空间是采用段的结构(这里段的概念也不同于虚拟分段中段的定义),一个作业可分为四个段:代码段(程序代码部分)、数据段(数据和数据操作)、堆栈段(程序使用的堆栈区)、附加段(用作字符串操作,主存中两个数据区传送时使用)。这四个段分别用四个"段寄存器"(16 位)来存放各段在主存中的起始地址。这四个段寄存器名称分别为 CS(代码段)、DS(数据段)、SS(堆栈段)、ES(附加段)。这四个段的地址空间可以是连续的、分开的、部分重叠的、甚至是完全重叠的。因此一个作业的最大地址空间是 256K,因为每段最大地址空间是 64K。CPU 除了有上述四个段寄存器外还有一个 16 位的指令指针寄存器 IP,起程序计数器作用,用以存放指令的相对地址,即指令在代码段中的相对位移量(数据段中某数据的相对偏移量,又称有效地址,需根据寻址方式的不同将不同的通用寄存器的内容相加来得到,请参阅具体资料。堆栈的相对偏移量由栈指针给出。附加段的偏移量产生与数据段相同)。可是主存的绝对地址到底是怎样产生的呢。当由相对地址(即偏移量)变换为绝对地址(主存的实际物理地址)时,CPU 根据不同情况(是取指令,还是堆栈操作,或是数据操作)而将相应的段寄存器(CS,SS,或 DS)的内容左移四位后加上相应的相对偏移量(IP 的内容,栈指针、有效地址)即得到实际的主存绝对地址,进行相应的访问操作。用这样的方法就达到了访问 1 MB 主存的目的。由此可以看出,每段的起址必定是 16 的整数倍(左移四位)。

由于每次访问主存时(不管是取指令,还是取和存数据)均要进行把相对地址与段寄存器内容(相当于基址寄存器内容)相加,这正是一个动态重定位的过程,每个段寄存器都

是一个定位寄存器。由此也可以看出,这些机器所使用的技术也正是多重分区技术(见7.4节)。

7.3.5 优缺点

本存储管理技术的优点是可以消除碎片,因此能有效利用主存空间,提高多道程序系统的多道程度,从而也提高了对处理机和外设的利用率。

其缺点是需要动态重定位硬件机构支持,这提高了计算机成本,并降低了速度(虽然是十分轻微的)。其次紧缩工作要花费机时。

7.4 多重分区(多对界地址)管理

前面介绍的各种管理技术都是每个用户只占据主存的一个分区。存储保护只需使用一对界地址寄存器,故也称为单对界地址管理技术。这种管理技术存在一些弊病。首先在可变分区多道管理条件下为解决碎片问题,就要通过移动程序进行存储器"紧缩"。这种紧缩需要硬件支持。其次是单分区存储管理技术不便于在进程之间共享数据。例如在系统中如果有多个用户进程同时要求运行某语言的编译程序。在单分区情况下,就必须在每个用户分区中都要包含一个该语言的编译程序。因为不可能使每个要求编译源程序的用户区都同此编译程序区连在一起。所以每个用户区都应放一个编译程序,这将大大浪费主存空间。所以无论从作为解决碎片来考虑,还是从进程共享信息(这点特别重要)来考虑,都可看到单对界地址管理不是理想的方法。因此人们引进了多重分区(多对界地址寄存器)管理技术。所谓多重分区技术是系统中设置了多对(一般不超过3～4对)界地址寄存器,并且在为每个作业(或进程)分配主存时,可按界地址寄存器对的个数为其分配多个不相邻接的自由分区。

采用多重分区技术既可以改善碎片情况,又便于共享。如上述几个进程要求同时编译的情况,只要这些进程的多重分区中都包含装有该编译程序的分区即可,这样主存中只有一个编译程序的副本。

在实存管理技术中,多重分区的多重程度不宜过多,否则会增加管理的复杂性,一般不超过3～4对界地址寄存器。多重分区技术的进一步发展导致了段式虚拟存储管理技术的出现,这将在第八章中讨论。多重分区技术的实现细节类似段式虚存技术,故在此不再细述。

7.5 覆 盖 技 术

7.5.1 覆盖的概念

当用户作业大于主存可用空间时,该作业往往无法运行。这对研制大的程序系统是一个很大的限制。尤其是在多道程序环境中,为了提高并行性和多道程度,每个用户作业所使用的空间往往是有限的。为了能在小的空间中运行大的作业,许多机器(如PDP-11,IBM-PC)都采用了"覆盖"技术。所谓覆盖是指一个作业的若干程序段(或数据段)间,或

几个作业的某些部分间共享某主存空间。

覆盖技术通常和单用户连续分配,固定分区和可变分区等存储管理技术配合使用。它不但用于用户作业的运行中,而且还常常用于操作系统本身的运行中。实际上操作系统本身也有存储管理问题。大型操作系统规模很大,将它全部放在主存是不可能的。所以操作系统被分为两部分。一部分是操作系统中经常要用到的基本部分,它是常驻内存的,并且有固定位置。另一部分是比较不经常使用的部分,它们放在磁盘上,用到时才被调入主存。对于这部分程序的存放,系统为了减少本身所占的主存空间,所以也常采用覆盖技术。

覆盖技术的基本概念可用图 7.13 来加以说明。一个作业由一个主程序段 MAIN 和若干程序和数据段组成,其调用关系如图 7.13(a),我们把这树形关系中的主程序段

图 7.13　覆盖技术例子

MAIN 叫根段,而 A_0 与 B_0; A_1, A_2 与 B_1; A_3 与 A_4 称为同层模块。根段常驻主存,不被覆盖。其余部分均为覆盖部分。覆盖部分分成三层,每层为一个“覆盖段”,所以模块 A_0 与 B_0 组成覆盖段 0,由 A_1, A_2 与 B_1 组成覆盖段 1,由 A_3 与 A_4 组成覆盖段 2。组成覆盖段的每个模块称为“覆盖”。覆盖段所占用的主存大小按该段中最大覆盖分配。同一覆盖段的各个覆盖被调入执行时占用同一空间。要注意的是多个覆盖构成同一层的前提是在作业执行时,各个覆盖在逻辑上是互相独立的。即在同一时间,只有一个覆盖在执行或被引用,不能调用同层中的其它覆盖。

读者可以看到,如不采用覆盖技术,该作业要求 270K 主存,采用覆盖技术后只要求 160K 主存。

7.5.2 覆盖处理

我们以 PDP-11 的 RSX-22M 为例介绍覆盖处理。用户将其作业画出如图 7.13(a)所示的树形结构。然后用覆盖描述语言 ODL 来描述其覆盖结构,称为覆盖描述文件。如图 7.13 可用 ODL 语言中的命令表示为:

ROOT MAIN—(A_0—(A_1, A_2—(A_3, A_4)), B_0—(B_1, B_2));

END

其中,ROOT 说明 MAIN 是根段。同层各覆盖间用逗点分开,并括在同一括号内,允许的最大覆盖层数为 16 层。END 表示结束行。

用户将覆盖描述文件随同目标程序一起提交系统。系统根据覆盖描述文件形成一个覆盖数据结构。然后系统把覆盖运行子程序和覆盖数据结构加到根段中去。这样程序运行时,系统就能自动将所需覆盖装入主存运行。

7.6 交换技术

交换技术被广泛用于小型分时系统的存储管理中,并且常与单用户连续分配和固定(或可变)分区多道管理等技术配合使用。交换技术的着眼点仍然是解决主存不足的矛盾。尤其在分时系统中,主机与几十个或成百个终端用户相连,如果把这些用户作业全部放入主存中,既不可能也不应该,只能允许一个或若干个用户作业同时保留在主存中。那么作业如何被调度和运行呢?通常调度程序从进程就绪队列中挑选到某一进程运行时,就把该进程调入主存运行一个时间片。一旦时间片到或进程因等待事件而不能运行时,它不但让出 CPU,而且也要释放出其所占有的主存空间,并且把该进程的程序和数据以文件形式保存在外部存储器中,直到调度程序再次调度到它时,才重新进入主存运行。这时又要把它的程序和所需之数据送入主存。这种技术称之为交换技术,又叫“滚进滚出”。

这种交换技术最早用在美国麻省理工学院的兼容分时系统 CTSS 中,并和单用户连续分配技术配合使用。当时主存中只保留一个作业,现在的交换技术一般允许在主存中同时保留几个作业。并且为了充分利用机时,常常把分时作业和批处理作业相结合,以便在主存中无分时作业可运行时使处理机运行批处理作业。

交换技术的优点是可以保证合理的响应时间(采用分时),并且利用外存来解决主存的不足。但这些优点是以花费处理机的时间为代价(因为交换过程增加 CPU 机时开销)。

交换技术也可用在批处理系统的固定分区和可变分区的存储管理技术中。在 OS/360 中,如果高优先级的作业要求处理而又没有足够主存可供使用时,则从主存中移出一个或多个作业到外存中去,让高优作业先运行。当该作业完成后,再将原来移出去的作业重新装入主存中运行。但作业必须严格装入其移出前的原来分区中(除非系统有动态重定位机构)。

交换技术中一个值得注意的问题是如何减少交换的信息量,这是研究交换技术的核心问题。对于分时作业来说,作业一般较小,因此交换的代价较少。但对于大型作业来说,如果每次以整个作业为单位换进换出,那么付出的代价可能相当大。近来发展了一些减少

信息交换量的交换算法,读者可参考有关著作。

交换技术的发展导致了虚拟存储管理中的分页技术的出现。

习　　题

7-1　在多级存储系统中,常将它分为几级?各级存储器有何特点?

7-2　在多级存储系统中,在存储系统的各级之间移动数据和程序,势必带来一定量的系统开销。请问为什么这样的系统愈来愈受欢迎?原因何在?

7-3　比较固定分区和可变分区这两种存储管理技术在概念上、数据库及其实现上、碎片情况等方面有何不同?

7-4　请指出存储分配算法与数据库的组织有何关系?

7-5　进程由于主存空间不能满足要求时,应在代表主存资源的信号量 Smemory 上等待。如果等待队列是先进先出结构。那么当有其它进程释放主存资源时,它可以唤醒队列中的第一个等待进程,也可以从整个等待队列中挑选一个等待进程。请说明为什么后一个方法尽管复杂,但可获得比前者更高的系统吞吐量?

7-6　图 7.9 为释放一主存块的框图。为使框图简单起见,该程序的执行效率有明显不足之处。请指出其不足之处并改正之(画出更完善的框图)。

7-7　解释逻辑地址、绝对地址、地址转换等概念。

7-8　何谓程序浮动和主存拼接?要求什么硬件来支持?两种紧缩算法有何利弊?

7-9　什么是存储器的内部碎片、外部碎片和表格碎片?

7-10　什么是覆盖?在什么情况下覆盖是有用的?覆盖如何影响程序开发和程序的可修改性?

第 8 章　虚拟存储管理

8.1　虚拟存储系统的基本概念

前一章所讲的各种存储管理技术的特点是作业运行时,整个作业的逻辑地址空间必须全部装入主存(除覆盖外)。并且当作业尺寸大于主存可用空间时,该作业就无法运行。我们把前述的各种存储管理技术统称为“实存”管理技术。这些技术均不能满足 7.1.2 节中所提出的用存储管理软件来“扩充”存储器,使之能运行比主存可用空间大得多的作业的要求。

同“实存”相对应的另外一类存储管理技术称为“虚拟存储”常简称“虚存”管理技术。它首先由英国曼彻斯特大学提出的,1961 年该校在 Atras 计算机上实现了这一技术。70 年代以后,这一技术才广泛被使用。现在许多比较大型的计算机均采用了虚拟存储管理技术,如 PDP-11/45, IBM 370 系列, Honeywell 的 Multics 等计算机就是使用虚拟存储管理技术的。同时在微型计算机的存储管理技术中,也愈来愈广泛地使用虚拟存储管理技术,如采用分页技术的微型机有美国的 NS32032, M68020, 日本的 V70, 使用分段技术有 Intel 80286, 分段加分页的有 Intel 80386, Z-8000 等。

那么什么叫虚拟存储器呢? 顾名思义,所谓虚拟存储器是指一种实际上并不(以物理形式)存在的虚假的存储器。搞清虚拟存储器概念的关键是把被运行进程访问的地址同主存的实际地址区别开来。由 2.2.2 节可知,一个程序被编译连接后产生目标程序,该目标程序所限定的地址的集合称为逻辑地址空间。目标程序中指令和数据放置的位置称相对地址或逻辑地址。这二者是不同于 CPU 能直接访问的主存的绝对地址(实存地址)空间或实存地址空间的。前者是逻辑上而非物理上的存储空间(程序的指令和数据放置的逻辑上的空间),而后者是程序在执行时实际存放其指令和数据的物理空间。在讨论实存管理技术时,对这两种概念不需要严格区分。本章讨论虚拟存储器管理,则必须严格地把这两种概念区分开来。

大家知道,进程要执行程序,这也就是说进程要访问的是程序的指令和数据的逻辑地址。而进程必须在处理机上运行,它通过处理机才能访问指令和数据,这样指令和数据必须存放在处理机能直接访问的主存中,也就是说处理机所实际访问的是存放指令和数据的主存地址。由此可以看出,逻辑地址和主存地址二者之间概念上是不同的,但又是有联系的。在虚存的管理中,通常把一个运行进程访问的地址称之为“虚拟地址”,而把处理机

可直接访问的主存的地址称“实地址”。把一个运行进程可访问的虚拟地址的集合称为“虚拟地址空间”，把计算机的主存称为“实地址空间”。程序的指令和数据所在的虚拟地址，为了能使进程访问，必须要放入主存实地址中去，并要建立虚拟地址和实地址的对应关系，也就是要由虚拟地址转换到实地址，这种转换由动态地址映象机构来实现。

把运行进程所访问的虚拟地址空间与主存地址空间区分开来后，这两个地址空间的大小就是独立的了。这样就为进程的虚拟地址空间可以远大于主存实地址空间创造了条件，也就是为作业大小可远大于主存空间创造了条件。否则就如实存管理技术中那样，作业大小不能超过主存可用空间大小。

当然只是把两个地址空间分开还不够，另外一个相关问题是作业运行时其整个虚拟地址空间(实际上同以前的逻辑地址空间概念)是否必须都在主存之中。如果必须都在主存中，那么虚拟空间仍然不能大于实际空间，作业大小仍不能大于主存可用空间。幸运的是，实际上这种要求并不是必须的。因为程序即使不是全部在主存中，仍然可以正确执行。最早的计算机就是把指令逐条输入执行的。同时一个程序在某一次运行中，常有某些部分是不用的(如无错误发生时就不会调用出错处理程序，又如程序中有些部分是可在执行过程中随选的)，或用得很少(如程序中的启动和终止部分只用一次)，即使程序中经常用的部分，其使用情况在执行过程中也随着程序的处理过程而变化。所以完全可以只让最近要用到的那部分程序和数据装入主存。以后用到那部分时再把那部分调入而把不用部分调出主存。这工作由有虚拟存储功能的操作系统负责。

用户全部程序和数据所组成的虚拟空间放在哪里呢。通常用一个大容量的外部存储器(磁盘、磁鼓)来存放每个用户的虚拟空间的全部数据。所以实际上用户的虚拟地址空间并不能增到无限大，它受到两个条件的制约:

(1) 指令中的地址场长度的限制。因为进程访问的虚拟地址应限制在指令中地址场长度所能表示的范围内;

(2) 外部存储器大小的限制: 用户的虚拟空间不能超过外存中的作业存放空间。

由于虚拟存储管理技术可以使用户的虚拟空间大于主存的实际空间，所以从效果上来说等于扩充了内存。除此之外，虚拟存储管理技术还带来了提高主存利用率，提高多道程度，便于实现进程间数据共享和保密等优点。

综上所述，所谓虚拟存储器是一个地址空间，是进程访问的逻辑地址空间，而不是物理的主存空间。最大虚拟地址空间往往取决于指令中的地址长度限制(因为外存空间通常大于指令地址长度所限定的范围)。由虚拟地址到实地址的变换通常由地址映象机构执行。

决定作业虚拟地址空间中哪部分进入主存，哪部分放回外存的工作由操作系统负责。具体来说它包括三方面内容:

(1) 把哪部分装入主存;

(2) 放在主存什么位置;

(3) 主存空间不足时，把哪部分置换出主存。

实存管理中所讲的覆盖技术为何不算虚拟存储技术呢? 主要是覆盖技术中程序员要亲自组织覆盖结构及其执行次序等，程序员心中绝没有认为主存是“无限大”的感觉。而在

虚拟存储管理技术中,主存与外存间的数据交换则由系统自动进行,不需程序员操心,对他是透明的。所以用户有认为计算机主存是“无限大”的虚假的感觉。覆盖技术严格说来也不算是由软件来扩充了主存。

本章主要研究目前广为使用的三种虚拟存储管理技术:分页存储管理、分段存储管理、段页式存储管理。

8.2 分页存储管理

回想在可变分区多道管理时,由于要管理那些不同尺寸的已分配的和自由的分区所带来的困难,以及实存管理技术中存在的诸如碎片及紧缩问题,以及主存利用率低等缺点(和作业大小受主存空间限制的致命弱点)导致了人们考虑是否把主存分成许多同样大小的存储块,并以这种存储块作为存储分配单位来克服上述缺点。这是分页技术思想的由来。

8.2.1 分页存储管理的基本概念

分页存储管理技术中的基本作法是:

(1) 等分主存:把主存划分成相同大小的存储块,称为页架。对于一个特定的计算机系统而言,页架大小通常是固定不变的(如 NS32032 以 512B, Z8000 以 1KB, V70 以 4KB 大小为一页),并给各页架从零开始依次编以连续的页架号 0, 1, 2, …。

(2) 用户逻辑地址空间的分页:把用户的逻辑地址空间(虚拟地址空间)划分成若干个与页架大小相同的部分、每部分称为页。并给各页从零开始依次编以连续的页号 0, 1, 2, …。

(3) 逻辑地址的表示:用户的逻辑地址一般是从基地址“0”开始连续编址,即是相对地址。在分页系统中,每个虚拟地址(相对地址)用一个数对(p, d)来表示。其中 p 是页面号,d 是该虚拟地址在页面号为 p 的页中的相对地址,称为页内地址。若给定一个虚地址 A,页面大小为 L。则

$$p = \mathrm{INT}\ \frac{A}{L}\ ;\ d = [A]_{MOD\,L};$$

其中 INT 是向下整除的函数,MOD 是取余。例如 L = 1000B,则第 0 页对应的虚拟地址为 0～999,第一页为 1000～1999。设 A = 3456 则 p = 3, d = 456 故 A = 3456→(3, 456)。

(4) 主存分配原则:分页情况下,系统以页架为单位把主存分给作业或进程,并且分给一个作业或进程的各页架不一定是相邻和连续的。进程或作业的一个页面装入系统分给的某个页架中,所以页面与页架对应。一个作业的相邻的连续的几个页面,可被装入主存的任一页架中。

(5) 页表与页表地址寄存器:由于一个进程的各页并不全在主存中,只是将最近需要的几页放在主存中,同时这几页又可能被分散地装入各页架。因此当进程访问某个虚拟地址时,系统如何知道该页在不在主存中?若在主存,则又放在哪个页架中呢?为此系统为每个进程建立一个页面映象表(以下简称页表)又称 PMT 表。该页表(初步)应包括以下

信息(见图 8.1):

图 8.1　分页存储管理系统

① 页号(进程各页的序号)。由于页号从 0 开始连续编号,与页表的表目序号一致,故可把页号栏从表中略去。

② 页架号。指该页面装入主存的第几个页架。

③ 状态。表示该页是否在主存中。通常该位为 0 时表示该页不在主存,该位为 1 时表示该页在主存中。

当作业调度程序调入作业时,为该作业建立进程,系统为该进程建立页表。在撤消进程时清除其页表。除此之外,系统还应设置一个页表地址寄存器以指出当前运行的进程的页表起始地址和页表长。

(6) 分页系统中的地址结构:进程的虚拟地址是用一个数对(p, d)来表示,那么这个数对在机器指令的地址场中又如何表示呢?通常将该地址场分为两部分:一部分表示该地址所在页面的页号 p,另一部分表示页内地址 d,其格式示于图 8.2(以 IBM 370 为例)。至于页号和页内地址场各占几位,主要取决于页的大小。如 IBM 370 的指令地址场长度为 24 位,页的大小为 2K 字节时,则页内地址部分应占 11 位(从 21 位到 31 位),页的大小为 4K 字节时,则页内地址场占 12 位。

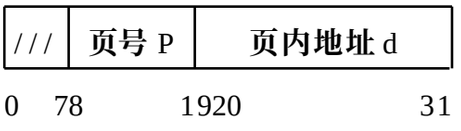
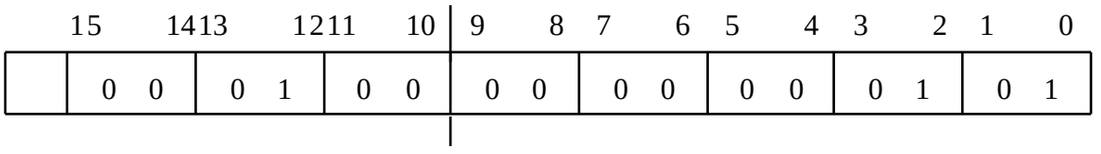


图 8.2　IBM 的地址格式

(7) 页面尺寸应是 2 的幂: 将逻辑地址转换成页号 p 和页内地址 d, 前面已介绍用除的方法。但如果每访问一个主存单元都做一次除法运算以得到页号 p 和页内地址 d, 那是不可思议的, 效率太低了。为此规定页的大小必须是 2 的幂。因为 2 的幂与机器内部的二进数制的表示法是一致的。这样给出一个逻辑地址(在机器指令中该地址是二进数表示的)后, 要决定其页号 p 和页内地址 d 就十分简单了。只要根据页的大小是 2 的几次幂, 就把地址场从第几位分开成两部分, 高位部分所表示的数即页号 p, 低位部位所代表的数即为页内地址 d。

例　页的大小为 1KB, 则逻辑地址 4101 的页号、页内地址可这样定:

1K= 1024= 2^{10}, 所以应从图示虚线处把地址分成两部分。而逻辑地址 4101= 2^{12}+ 2^{2} + 2^{0} 表示如下图, 所以高位部分(图中左面部分)的数为 4, 低位(右面)部分的数为 5, 故 P = 4, d= 5。所以 4101→(4, 5)。



采用页面大小是 2 的幂的优点是省去除法, 由硬件自动把地址场中的数拆成两部分来决定对应的页号和页内地址。

8.2.2　分页系统中的地址转换

8.2.2.1　直接映象的页地址转换

本节和以下几节主要研究几种执行页面地址映象的技术。

当进程被调度到处理机上运行时, 操作系统自动将该进程的页表起始地址(该地址通常在该进程的 PCB 内)装入页表地址寄存器中, 当该进程要访问某个虚拟地址时, 则分页的地址映象硬件自动按页面大小将地址场从某位起截成二部分: 页号和页内地址(p, d)。这时以页号 p 为索引查找页表, 查找工作由硬件自动进行。具体过程如图 8.3 所示: 加法器将页表地址寄存器内容与页号乘以页表表目长度(字节数)的积相加(硬件执行时自动把页号左移若干位即得乘积)得到该页在页表中的表目地址, 该表目中包含有该页所对应的 页架号p ′。然后地址映象硬件又自动将页架号p 与页内地址(也是页架内地址)如图 8.3 拼成了实际主存的绝对地址。实现了地址映象。

地址映象硬件实现并不复杂, 但有几个需要考虑的问题:

(1) 页表放在哪里? 整个系统的页表空间有多大? 通常页表放在主存中的系统表格区。可以将各个进程的页表分散放。但多数的系统都是在系统表格区开辟一个集中的页表空间。进程需要建立页表时, 则向系统请求分配页表空间(这本身也是个存储分配问题)。一个系统的页表空间的大小与主存大小有关。以 IBM 为例, 若主存为 16MB, 页大小为 2KB 时, 则页表表目最少要 8192 个。如果每个表目长为两个字节, 那么页表最少要占 16KB 的主存。

(2) 直接映象的分页系统对系统的效能有什么不利影响吗? 最大的问题是影响了处理机执行指令的速度, 它降低为原来的二分之一。因为 CPU 至少要访问两次主存才能存取到所要数据, 第一次查页表以找出对应的页架号, 第二次才真正访问所需数据。存取数

图 8.3　直接映象分页系统地址转换

据的速度降低这么多,这是令人难以忍受的。因此必须寻找一个更快的地址转换的方法——相关映象页地址转换法。通常将本节中所述的方法——通过放在一般主存储器中的页表进行地址转换的方法,称为直接映象页地址转换法。

8.2.2.2　相关映象页地址转换

为了提高页地址转换速度,一个显而易见的办法是把所有页表放在高速的半导体相关存储器(或称关联存储器)中。因为半导体存储器的数据存取速度比一般存储器高一个数量级。其地址转换过程如图 8.4 所示。当运行进程访问虚拟地址 v 时,该地址被硬件截

图 8.4　相关映象页地址转换

成两部分:页号 p和页内地址 d。这时硬件以页号 p 对相关存储器中页表的各表目同时进行比较,以查出页号为 p 的所在的表目(相关存储器的页表是把页号包含在表内作为一个栏目,不可略去),找到后则自动送出相应的页架号,与页内地址 d 拼成绝对地址,然后按

此地址访问内存。

本方法的实现从概念上讲是简单的,与直接映象页地址变换的区别是把页表放在高速相关存储器中,而不是放在普通的主存储器中,这样的页表被称为快表或相关页表,其访问数据的速度基本上接近原来速度(约降低 10%)。并且相关页表是各表目同时比较内容,所以不用加法器相加的方法来找所需表目。但由于整个系统的页表所占空间很大,所以需要很多半导体相关存储器。而半导体相关存储器比较贵,因此成本很高,不很实用。

8.2.2.3　相关映象和直接映象结合的页地址转换

此方法是将上述两种方法综合起来的。由于半导体相关存储器很贵,页表全部使用半导体相关存储器来存放不经济,于是想到把各作业的页表仍然放在主存的系统区内。除此之外、系统还使用一定数量(如 8～16 个)的高速半导体寄存器所组成的相关存储器。其中存放着正在运行进程的、当前最常用的部分页面的页号和它的相应页架号。其地址转换过程如图 8.5 所示。转换过程如下:当运行进程要访问虚拟地址 v,于是硬件将地址 v 截成页号 p 和页内地址 d。地址转换机构首先以页号 p 和相关存储器中各表目同时进行比较,以便确定该页是否在相关存储器中。若在其中,则相关存储器即送出相应的页架号与页内地址一起拼接成绝对地址,并按此地址访问主存。若该页不在相关存储器中,则使用直接映象方法查找进程的页表(见 8.2.2.1 节),找出其页架号 p 与页内地址拼成绝对地址,并访问主存。与此同时要将该页的页号、及对应的页架号一起送入相关存储器的空闲表目中去。如无空表目,通常把最先装入的那个页的有关信息淘汰掉,腾出表目位置。实际上直接映象和相关映象同时进行。当相关映象成功,就自动停止直接映象工作。

图 8.5　相关映象与直接映象结合的地址转换

由于程序执行中有局部性的特点,即刚被访问过的单元在很短时间内还将被访问(时间局部性)和刚被访问过的单元的邻近的单元也将被访问(空间局部性,见 8.5.5 节)。所以只要使用 8～16 个高速寄存器作相关存储器,那么从相关存储器中找到所需页号的概率为 90% 左右。使用这种方法来存取数据,速度约降低 10% 以下。例如微型计算机 Z-8000 的相关存储器有 16 个表目,命中率可达 96%。而微型计算机 NS32032 有 32 个表目,命中率达 97%。如果命中时只需一个时钟周期就完成地址变换,不命中时约需 20 个时钟周期。

8.2.3 分页存储管理策略

分页存储管理策略包括三个方面内容:

(1) 拿来策略:一个进程被调度到处理机上运行时可能没有或只有较少的页面在主存中,为了进程的顺利运行,首先要解决何时以及哪一页(或几页)将被装入主存。这通常有两种方法:

① 当进程访问某虚拟地址所在的页面不在主存中时(即页表中该页的状态位为"0")。这时硬件自动触发"缺页中断"(属程序中断),并由缺页中断程序将该页装入主存。故此拿来策略是当用到某页而又不在主存中时,才将此页装入,这叫请求分页。我们把用到的页不在主存中的情况称为缺页。

② 在进程访问前,预先把一页(或几页)装入主存。这主要是估计该页即将被访问的可能性很高,所以预先把它装入以提高进程运行速度。在 IBM 4300 系统的 OS/VS 和 DOS/VSE 中提供了一个汇编语言的宏命令,它可以在程序访问前将一个或多个页面装入主存,称为提前页,而不是通过缺页中断将页装入。适当使用此命令(尤其作业开始运行前)可减少缺页中断,提高系统效率。如果过分地滥用此命令,将会使分页管理退化为实存储器管理。

(2) 放置策略:它要解决一个被调入主存的页面装入主存哪个页架的问题。对分页系统来说,一般可以将新进入的页面放入任何一个可用的页架中。

(3) 置换策略:主要解决当进程请求调入一页,而主存中又无可用的自由页架的问题。这时需要从主存中更换出去一页(或称淘汰),以腾出一个页架给新进入的页面使用。置换策略是用来挑选被淘汰页的策略,这是分页系统中讨论的一个主要内容之一。

本节中所介绍的内容同样适用于分段和段页式存储管理。

8.2.4 分页存储管理的软硬件关系和软件算法

8.2.4.1 数据库

为了实现分页存储管理,操作系统必须建立和管理以下一些基本的数据库:

(1) 进程表:整个系统的 PCB 集合,它应包含以下信息:进程名;存储保护键;该进程的页表起始地址;页表长度(或作业大小);以及状态信息等内容。当某作业被调度进入主存时,操作系统将其信息登入该进程的 PCB;当进程运行时,操作系统的调度程序从此表中将页表起始地址和页表长度均装入页表地址寄存器中。

(2) 存储分块表:整个系统设置一个存储分块表,该表指出主存中各页架的状态是已

分配还是可用。

(3) 页面映象表(简称页表):为每个进程设置一个页表。页表内容见 8.2.4.3 节。

(4) 文件映象表:该表用以指出各进程的虚拟空间(以文件形式存于外存之中)的各页在外存中存放的地址和状态信息(见文件系统)。

8.2.4.2 分页存储管理中软硬件关系和缺页中断处理算法

在 8.1 节中我们指出,进程访问的是逻辑地址——即其对应程序的指令和数据在虚拟地址空间中的相对地址。进程通过处理机执行其程序。处理机只能访问在主存中的指令和数据——即实际物理地址。那么虚拟分页机构是如何实现从虚拟地址到实地址变换呢?或者说分页存储管理中软硬件如何共同完成地址变换?分页机构是由三部分共同完成由虚拟地址到实地址变换的:

(1) 分页地址变换硬件部分;

(2) 中断处理硬件部分;

(3) 软件部分——缺页中断处理程序。

其中第一部分是主要的经常起作用的,由它单独完成正常的地址变换工作,后两部分只有在发生缺页情况下才协助第一部分进行工作。现在通过考察一条指令的执行过程来了解三部分的工作情况和分页存储管理中软硬件协同工作的关系。

假定该条指令的操作码已放入译码器,而所需的数据或指令的地址已放入地址寄存器中。(这里要指出的是,这些地址均是进程的虚拟地址,因为虚页是原封不动地装入主存的。)此时该指令执行过程将按以下所指出的步骤进行。

(1) 分页地址变换硬件首先按步骤①～⑤工作:

① 将地址寄存器中虚拟地址由硬件自动截成 p(页号),d(页内位移)

↓

② 硬件自动查找页表中该页状态位

指示该页在主存否?

在↙ ↘不在

完成正常地址转换

③ 硬件将页表中该页对应的页架号与页内位移 d↓ 拼成绝对地址

⑤ 硬件使“缺页中断触发器”被触发

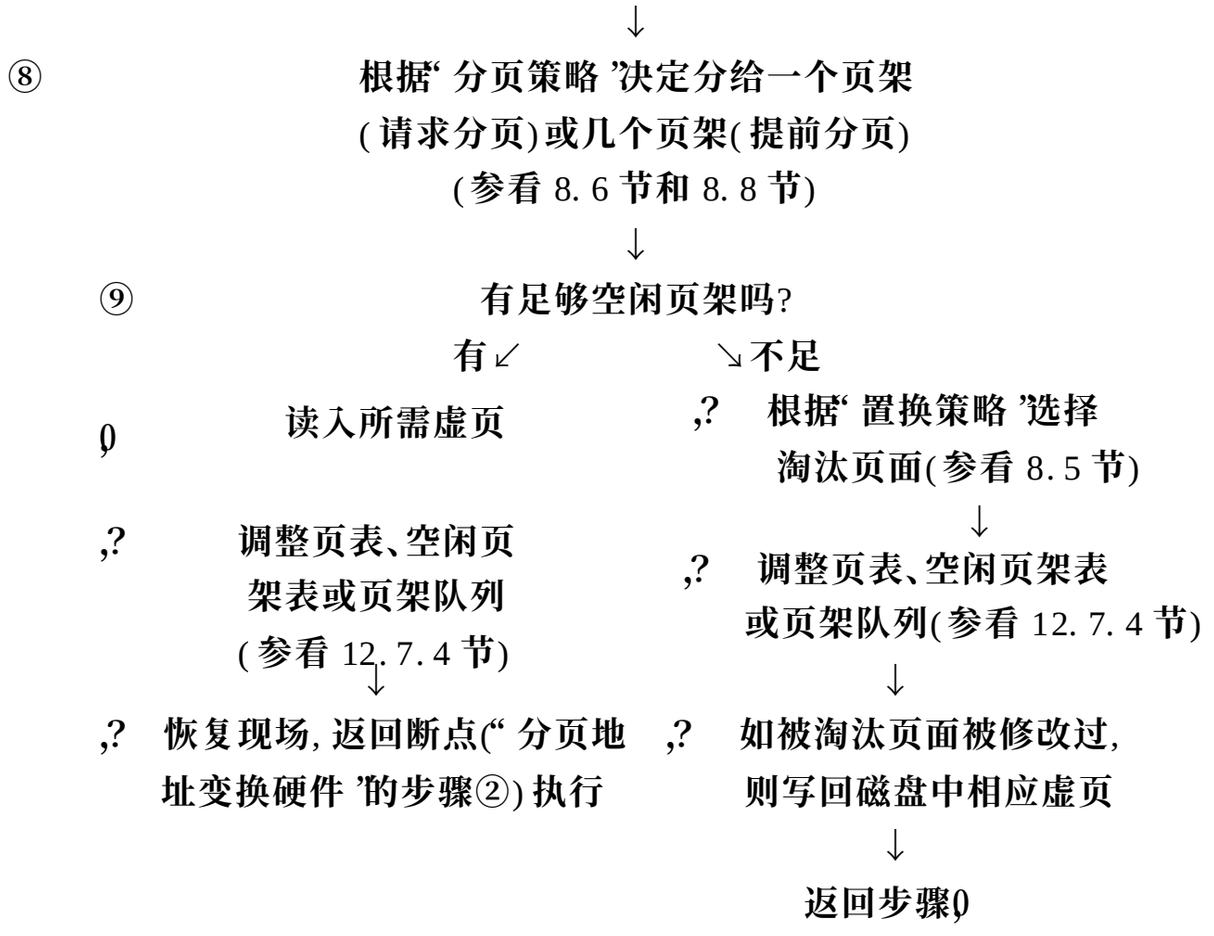
④ 硬件完成指令,并将下一条指令及数据地址放入译码器和地址寄存器。并返回步骤①

(2) 中断处理硬件完成下述步骤⑥的工作:

⑥ 当触发的缺页中断被接受后,中断处理硬件自动进行处理机状态的切换。即把旧的程序状态字(PSW)、指令指针(IP)、代码段基地址寄存器(CS)保留起来,并把相应中断向量指针放入 IP 与 CS 中(参看 2.1.4.5 节)。从而完成了将 CPU 控制转给了缺页中断处理程序的切换工作。

(3) 缺页中断处理程序的软件算法:缺页中断处理程序得到对处理机控制权而运行,其工作是处理“缺页中断”。其软件算法表示为步骤(7)～(14):

⑦ 保留断点现场



8.2.4.3 页表表目的扩充

在 8.2.1 节中所介绍的页表的每个表目只包含页号(可从页表中略去)、页架号、状态位三种信息。且状态位只表示一个页面在不在主存中。这些信息对分页管理是不够的。比如,怎样表示一个页面中所包含的数据是否被修改过;又如怎样解决这样一种情况所带来的问题:作业 A 有一个页面正在向主存的页架 p_1 中装入。此时作业 A 因等待页面输入输出完成而释放出处理机。处理机被分给了作业 B,而作业 B 因缺页,要求分给一个页架。但主存中已无空闲可用的页架,所以要淘汰一页。如果作业 A 的页架 p_1 的状态为“已分”,那么页面中断处理程序可能挑选 p_1 淘汰出去,但页架 p_1 正在装入作业 A 的页面。那么是否把页架 p_1 的状态标志成“可用”?如果这样,则页架 p_1 可能被分给作业 B,于是两个作业同时向同一页架中装入数据。怎么解决这个问题呢?为此,必须通过扩充页表表目的功能来解决这些问题。通常页表表目在逻辑上应包含如下信息(所谓“逻辑上”的含义是指有些特征位——修改位、访问位等,实际上并不一定放在页表中):

(1) 页号、页架号;

(2) 状态:以 IBM 4300 系列为例,一个页面可以有三种状态:

①“断开”状态。指该页不在主存,CPU 对它不能进行存取操作;

②“连接”状态。指该页已分配页架号,但正在进行数据的输入输出操作,此时 CPU 对该页不能进行访问,而通道可以对它进行访问;

③“可寻址”状态。指该页已分配页架号,CPU 可对该页进行访问。

(3) 修改位:该位为 0 时,表示该页面中的数据未被修改过。该位为 1 时,表示该页面中的数据已被修改过。

(4) 访问位:表示该页面在最近期间是否被 CPU 访问过。该位为 0 时,表示该页面未

被访问过;为 1 时,表示该页面最近被访问过。该位主要用于页置换策略中。

通常页面的访问位、修改位与每个存储块(页架)相关联。有的是把这两位放在存储分块表中,也有存放在主存块的附加位中,有的还将此两位也同时放入相关存储器(快表)中。

(5) 外存地址(外页地址):指出该页在磁盘等外存上的地址。

8.2.5 页的共享

在多道程序系统中,尤其在分时系统中,数据共享是很重要的。在分页系统中,诸共享进程应能共同访问共享程序的诸页。所以共享的方法是使这些共享进程的逻辑地址空间中的页指向相同的页架号(其中放有共享数据)。图 8.6 中表示了三个进程共享放在页架 3 中的数据。

页面共享中必须小心处理的一个问题是共享页中的信息保护问题。当一个进程正从共享页中读数据时,要防止另一进程修改该页中的数据。在大多数执行共享的系统中,将程序分为过程区和数据区。不可被修改的过程称纯过程或可再入过程,它是可以共享的。私用的可修改的数据和过程不可被共享。

附带指出,在分页系统中执行页的共享,比之分段和段页式存储管理中的共享要困难些,这是由于分页系统中将作业的地址空间划分成页面的作法对用户是透明的,同时作业的逻辑地址空间是线性连续的。所以当系统将作业的逻辑地址空间划分相同大小的页面时,被共享的程序部分不见得被包含在一个完整的页面中。这样如果共享页中有进程的私用数据时也被共享了,不利于保密。其次共享部分的起始单元在各进程的地址空间划分成页面过程中,在各自的页面中的相对偏移量(页内地址)不同也使共享更为困难。但通过改进连接装入程序还是很容易克服这种困难的。

图 8.6 页的共享

8.3 分段存储管理

前面介绍的各种存储管理技术中,用户的逻辑地址空间已被连接成一个一维的线性地址空间。而事实上一个作业常常由大量的标准和非标准的程序模块和数据段组成。如果作业运行前,将其所需的各程序段和数据段连接成一维的线性地址空间,这工作既费时间,又不便于作业的执行,尤其不便于共享。因此人们希望按照程序模块来划分段,并按这些段来分配主存。所谓段,可定义为一组逻辑信息的集合,如子程序,数组和数据区等。它们通常是以文件形式存在于文件系统中。

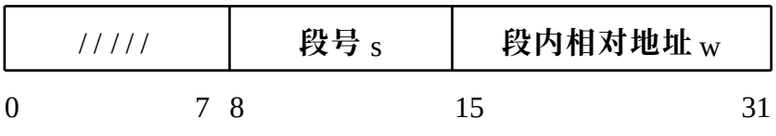
8.3.1 分段存储管理的基本概念

分段存储管理的基本概念可用下列几方面来表征:

(1) 进程的逻辑地址空间: 分段情况下要求每个进程的地址空间按照程序自身的自然逻辑关系分成若干段, 每个段有自己的段名(段名通常由程序员给出), 故分段存储情况下作业的逻辑地址空间是二维的。每一个虚拟地址均要求用两个成分: 段名和段内地址来描述(图 8.7)。分段在经过编译或汇编后, 系统为了管理的方便, 常常为每一段规定一个内部段名, 内部段名实际上是一个编号, 称为段号。每个段的地址空间都从“0”地址开始编址成连续的线性地址, 于是经过编译后的虚拟地址由这样二部分组成: 表示段名的唯一的段号和段内相对地址。

图 8.7 分段地址空间

(2) 程序的地址结构: 因为一个虚拟地址要用两个成分(s, w)来描述, 所以指令的地址场的结构形式应为图示形式。通常规定每个进程的段号为从“0”开始的连续正整数。



假定某机器指令的地址部分长为 24 位, 如果规定左边 8 位表示段号, 而右边 16 位表示段内相对地址。这样的地址结构就限定了一个作业最多的段数为 256 段, 最大的段长为 64KB。

(3) 主存分配: 段式管理的主存分配以段为单位, 每一段分配一块连续的主存分区, 一个进程的各段所分到的主存分区不要求是相邻连续的分区。

(4) 段表和段表地址寄存器: 同分页一样, 系统为每个进程建立一个段映象表(简称段表)以实现动态地址转换。段表中应包含以下内容: 段号、段的长度、段在主存中的起始地址、段的状态位、访问位、修改位、段的外存地址等。段表按段号从小到大的顺序排列, 并且包含该进程的全部段。当作业调度程序调入该作业时为其进程建立段表, 在撤消进程时清除此进程的段表。

段表中段号一栏可省略, 段长表示该段地址空间的大小。

类似地, 系统还要设立一个段表地址寄存器以指出运行进程的段表在主存中的起始地址和段表的长度。在每个进程被调度到 CPU 上运行前, 由系统从进程的 PCB 中将此两项内容装入段表地址寄存器中。

8.3.2 分段管理中的地址转换

段地址转换与分页情况基本相同,其大致过程如图 8.8 所示。进程运行时,由系统将该进程的段表地址和段表长装入段表地址寄存器中。当进程访问某逻辑地址(s,w)时,系统将段表地址寄存器中内容 b 与段号同段表表目长的乘积相加后,得到该段的表目入口地址。由此表目中查得段 s 在主存中的起始地址 s′,再将 s 与段内相对地址 w 相加,而得到欲访问单元的主存绝对地址,并进行访问。

图 8.8 分段地址转换

类似分页中的情况,CPU 执行指令的速度降低为原来的二分之一(因两次访问主存)。为了提高地址转换速度可以采用高速相关存储器技术。

8.3.3 段的动态连接

一个作业是由许多程序模块和数据段组成的,在以前所讲的存储管理技术中,无论是分页和实存管理技术,为了程序正确的执行,必须要由连接装配程序把一个作业所调用的各个子程序和数据段连接成一个可运行的目标模块——将其连接装配成一个一维的线性连续地址空间。在连接装配过程中,既要把所需之子程序从各个程序库中找出来,并且要重定位,把它们组成一个统一编址的相对地址空间。这个过程称之为运行程序的“静态连接过程”。

但是这种连接装配过程既复杂又费时间(一个复杂的大型程序可以由数百个子程序组成),有时连接过程所花费的机时比该作业的执行时间还长。还经常发生许多被连接好的模块在作业运行过程中根本不用,而造成连接时的机时和主存空间的浪费。所以最好能

采用什么时候用到哪一段再连接该段的方法。这种方法称为动态连接方法。所谓段的动态连接是指“在一个程序运行开始时,只将主程序段装配好并调入主存。其它各段的装配是在主程序段运行过程中逐步进行的。每当需要调用一个新段时,再将这个新段装配好,并与主程序段连接。在分段存储管理环境中,由于逻辑地址空间是二维的,每段有自己的段名,因而实现动态连接比较容易。而在分页环境中动态连接是难以办到的。

8.3.3.1　连接间接字和连接中断

引入“连接间接字”概念前,先回忆一下机器指令中直接和间接寻址指令两种方式。如图 8.9 所示,若第 400 号单元存放着数 1320,第 1320 号单元存放着数 1400。则在直接寻址时,指令 ADD 400 表示取出第 400 号单元的内容(即数 1320)作为操作数。在间接寻址时,指令 ADD* 400 表示第 400 号单元的内容 1320 被作为直接地址,取出第 1320 单元的内容作为操作数。在间接寻址方式中,把包含直接地址的字(图中 400 号单元)称为间接字。

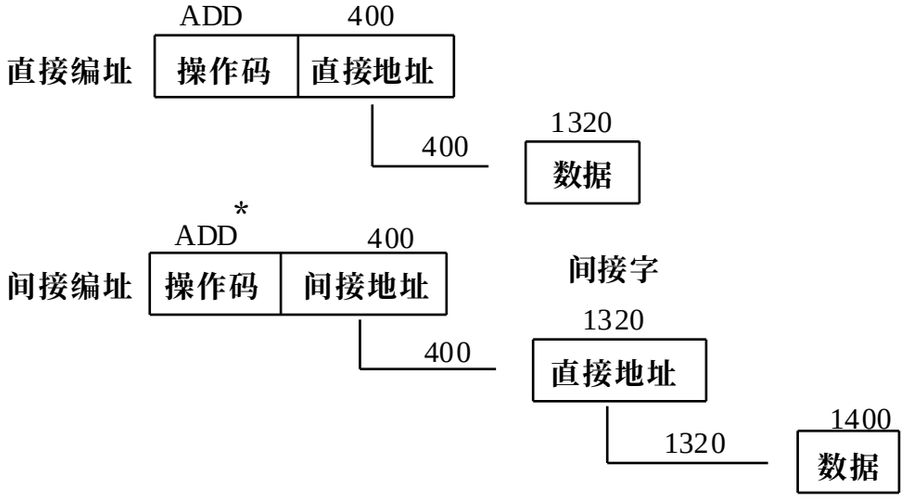


图8.9　直接和间接编址比较

通常在间接字中,除了包含直接地址外,还可以包含附加的状态位。在分段存储管理情况下,一个间接字可有如图 8.10 的格式:其中 L 为“连接标志位”。当 L= 1 时,表示该段尚未连接,当 L= 0 时,表示该段已进行了连接。这样的间接字称为连接间接字。

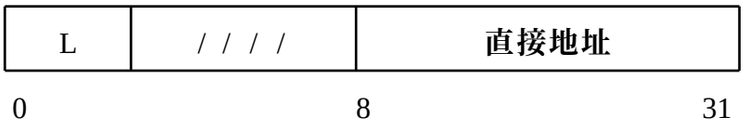


图 8.10　连接间接字格式

在具有段的动态连接功能的机器中,处理机在执行间接指令时,其硬件能自动对连接间接字中的连接标志位 L 进行判断。当 L= 1 时,硬件自动触发中断信号(称为连接中断),并且停止执行该间接指令,转去执行连接中断处理程序。等到连接中断处理程序处理完后(这时 L 已被中断处理程序改为“0”),再重新执行该间接指令;当 L= 0 时,就根据连接间接字中的直接地址去取数执行之。

8.3.3.2　编译程序的连接准备工作

假如用户用某种高级语言书写主程序 MAIN,如下所示:

```
        ┆
    CALL[P]〈Y〉;转子程序 P 内的 Y 入口点
        ┆
```

编译程序编译每个程序段时遵循以下原则:当被编译的程序段中的语句是访问本段的单元时,则该语句被编译成直接寻址指令格式;若是访问外段的单元时,则被编译成间接寻址指令格式,编译程序为之在本段中设置一连接间接字,并置连接间接字中的连接位 L = 1。连接间接字中的直接地址场本应包含直接数的地址,但由于要访问的外段与本段尚未连接,故编译程序在本段中要保留被访问段的符号名以便访问时进行连接。保留的方法是开辟一个存储单元来存放被访问段的符号名(见图 8.11),并使连接间接字的地址场指向该符号名所在的单元地址。

图 8.11　段的编译和连接示例

8.3.3.3　连接中断处理

处理机执行间接指令时,硬件判断 L = 1,则自动触发连接中断。并转操作系统的连接中断处理程序,其处理过程如下:

(1) 根据连接间接字找出欲访问段的符号名和段内地址。

(2) 在该作业的活动符号表(见 10.6.1 节)或系统的活动文件表中查找该段是否已在主存。

(3) 如果该段已在主存,则找出其段号,并把连接间接字中的直接地址场中原来的内容改为段号和段内地址。同时把连接标志位 L 清为零。返回断点执行原来的间接指令。

(4) 如果该段不在主存中,则为该段分配主存,然后将该段读入主存中(这一工作也可由缺段中断完成),为其分配一个段号,则重复上面步骤(3)的工作。

8.3.3.4　纯段和杂段(连接段)

如果一个被连接段是"纯过程(或称可再入的)"段,则在该过程段执行中,是不允许修改其中的数据的。但在上节所讲的连接中断处理过程中,实际上修改了连接段中的数据(修改连接间接字),这与纯过程段的概念相矛盾。所以在动态连接情况下,为保持连接段

是‘纯’的,在 IBM 360 系统和 Honeywell MULTICS 的 PL/1 语言的编译程序中采取这样一个办法:即另外设置一个杂段(或称连接段),将程序的所有可能变更的数据,包括连接间接字和某些临时变量、内部变量等,都放在杂段中。这样,对于每个源程序来说,编译程序至少必须产生两个分开的目标分段——纯段和杂段。

在 MULTICS 系统中编译后的情况如图 8.12 所示。系统使用了一个过程(段基址)寄存器、和一个连接(段基址)寄存器分别指出过程(纯)段与连接(杂)段的基地址。图中的 K_1 和 K_2 这两个常数由编译程序确定。连接过程同前。这样就保证了过程段是纯的。

图 8.12　编译后的过程段和连接段

8.3.4　虚拟存储管理中的存储保护问题

对存储器的保护包括两个方面的内容:

(1) 越界保护:实存管理技术中越界保护是通过一对界地址寄存器。而在分页和分段环境下是通过给出进程的最大页号(页表长度)和最大段号(段表长度)。当进程运行时,其页表长度和段表长度也同时被放入页表(段表)地址寄存器的左边部分(见图 8.3, 8.8)。当 CPU 访问某逻辑地址,硬件自动把页号(段号)与页表(段表)长度进行比较。在分段环境下还要将段内地址与段表中该段长度进行比较,如果合法,才进行地址转换,否则均产生越界中断信号。

(2) 存取控制保护:在分段情况下,段包含的是逻辑上完整的信息,需注意防止其中的信息被偷或被修改。因此往往要严格控制各种共享类型的用户的访问权限。目前,常用的存取控制类型包括:

① 读。允许用户对该段(页)中所包含的任何信息或整个段(页)的副本进行读操作;

② 写。允许用户修正该段(页)中所包含的任何信息,以至于撤消整个段(页);

③ 执行。用户可把该段(页)作为程序来运行。但通常不允许对数据段(页)进行这一类型的访问;

④ 增加。用户可在段(页)的末尾添加信息,但不允许修正已存在于段(页)中的信息。

各类用户的访问权限是以上存取控制类型的合理结合。如文件主可有全部四种类型的访问。对于同组用户可根据情况允许其读或写,而一般的用户可能拒绝一切类型的访问。

对共享段的访问还要注意读和写动作之间的互斥。

8.3.5 分段存储管理的优缺点

分段存储管理技术的优点为:

(1) 便于处理变化的数据结构: 在实际应用中, 有些表格或数据段其长度随输入数据多少而变化(如编译或汇编过程中的符号表), 事先很难知道该段有多大。在这种情况下, 要求能动态地扩大一个段。这在分段环境下是易于实现的, 因为在一个分段后面添加新的数据不会影响到地址空间中的其它部分(在一般一维线性地址空间中, 显然不能任意膨胀。因为这会影响被添加部分后面的编址关系。除非添加部分在整个地址空间的最后)。

但也没有必要允许每一个段都可随便增长, 往往只需要允许某些段可增长。为了标志哪些段可动态增长, 我们再次扩大段表功能 —— 在段表中增加一个增长标志位。如果该位为 0, 则表示该段不许增长。如果该位为 1, 则表示该段允许动态增长。在这种情况下, 由于往段中添加新的数据, 往往会触发越界中断, 操作系统的越界中断处理程序将根据动态增长位来判别增长是否合法, 如合法时则增加段的长度。

(2) 便于共享: 共享的实现是只要所有欲共享的作业的段表中, 有相应的表目指向被共享段在主存中的起始地址即可。

系统中有一个活动文件表, 它登记所有在主存中的共享段的有关信息, 并控制共享用户的访问权限。

(3) 提供了虚存的功能: 由于作业的各段(页)不必全在主存中, 这样可在小的主存分区内运行大的作业。提高了多道程度, 从而增加了主存和处理机的利用率。在分段情况下, 当一个新段要调入主存时, 为在主存中找到一个足够大的分区, 同样可能需要紧缩。而在分页情况下是不存在紧缩的问题的。

(4) 提供了动态连接的便利: 见 8.3.3 节。

(5) 便于控制存取访问: 见 8.3.4 节。

缺点有:

(1) 要为存储紧缩付出处理机机时的代价。
(2) 分段的最大尺寸受到主存大小的限制。
(3) 在外存中管理可变尺寸的分段比较困难。
(4) 与分页一样, 提高了硬件成本。

8.4 段页式存储管理

段页式存储管理是把分段和分页的两种技术结合的结果, 综合了二者的优点。IBM 370 和 Honeywell 6190 等机器均使用段页式存储管理技术。

8.4.1 段页式存储管理的基本概念

段页式存储管理技术的基本要点是:

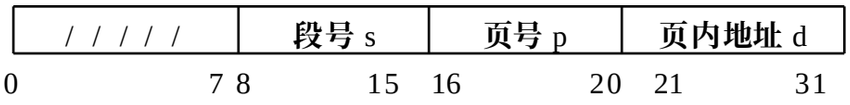
(1) 等分主存: 把整个主存分成大小相等的存储块, 称为页架。并从零起依次给各页架编以连续的序号, 称页架号。

(2) 进程的地址空间采用分段的方式: 即按程序的自然逻辑关系把进程的地址空间

分成若干段,而每一段有自己的外部段名和内部段号(同分段技术)。

(3) 每一段又采用分页方法:按照主存页架大小把每一段划分成若干页。每段都从零开始为自己段的各页依次编以连续的页号。

(4) 逻辑地址结构:一个逻辑地址用三个参数表示:段号 s;页号 p;页内地址 d。记为 v= (s, p, d)。因而指令中的地址部分应为如下形式(以 IBM 370 为例)。在 IBM 370 中,一个进程最多可有 256 段,每段最多可有 32 页,每页为 2KB(当每页为 4KB 时,每段可有 16 页)。



(5) 主存分配:主存以页架为单位分配给每个进程。

(6) 段表、页表、段表地址寄存器:为了进行地址转换、系统为每个进程建立一个段表,并且要为该进程段表中的每一个段建立一个页表。为指出运行进程的段表地址,系统中有一个段表地址寄存器来指出进程的段表起始地址和段表长度。其关系见图 8.13。各段的页表起始地址由段表内给出,并给出该段的页表长度。

图 8.13　段页式存储管理地址转换

图 8.14　进程表、段表、页表、页架的关系

8.4.2　段页式存储管理中的地址转换

段页式存储管理中地址转换如图 8.13 所示。若运行进程访问虚地址 v= (s, p, d), 在没有相关存储器情况下, 地址转换过程如下:

(1) 地址转换硬件将段表地址寄存器内容与指令地址场中的段号 s 相加(按段表的表目长进行适当移位后相加), 得到欲访问段 s 在该进程的段表中表目入口地址。

(2) 从该表的表目中得到该段的页表起始地址, 并将其与地址场中的页号 p 相加后

得到欲访问页 p 在该段的页表中的表目入口地址。

(3) 从该页表表目中取出其对应的页架号与指令地址场中的页内地址 d 拼成主存的绝对地址。

在使用相关存储器情况下,相关存储器的表目一般应包含如下内容:段号;页号;(该段号和页号所对应的)页架号;该段页表长;存取控制信息等。如同在分页和分段中情况一样,每次进行地址转换时,同时进行直接映象和相关映象的地址转换工作。当相关映象地址转换成功即自动停止直接映象工作。相关映象步骤如下:

(1) 以段号、页号为索引同时对相关存储器的各表目进行比较。

(2) 如果匹配,则比较页表长度和存取控制信息以判断访问的合法性。若不合法,则停止地址转换工作,发出相应的中断信号。

(3) 取出页架号与页内地址 d 拼成绝对地址,并按此地址访问主存。

(4) 若在相关存储器中查不到该表目,则通过直接映象来找出页架号,同时把有关信息装入相关存储器中。

图 8.14 中表示了段页式存储管理的进程表、段表和页表与主存中页架的关系。

8.4.3 段页式存储管理算法

与分页和分段中情况一样,在地址转换过程中,硬件和软件密切配合,其操作流程如图 8.15 所示。

现对图 8.15 作些说明:

(1)“s≤段表长吗?”这是由硬件自动将段号 s 与段表地址寄存器中的段表长进行比较。

(2)“段连接了吗?”这是根据连接间接字中的连接标志位 L 的值由硬件自动进行判别。

(3)“段在主存吗?”这是根据段表中状态位值由硬件判别。在分段存储管理情况下,“段在主存吗?”的含义是指该程序段(数据段)本身未在主存。而在段页式存储管理情况下,“段在主存吗?”的含义是指该段的页表是否已在主存中建立了。

(4)“p≤页表长吗?”是根据段表的表目中的页表长度(见 8.4.1 节)与 p 进行比较。在有相关存储器的情况下是将 p 与相关存储器中的该表目的“该段的页表长”(8.4.2 节)进行比较。

(5)“访问类型合法吗?”是将本次访问的类型与相关存储器中的存取控制信息进行比较。

(6)“页在主存吗?”是根据该段的页表中的相应页的状态位判定。

(7) 缺页中断处理部分与 8.2.4.2 节中(3)相同,故而略去。

在段页式环境下,分段的各页是根据需要一页一页地调入,所以那些不被访问的页不在主存中。

8.4.4 段页式存储管理的优缺点

段页式存储管理是分段技术和分页技术的结合。因而它具有它们的全部优点,包括:

图 8.15　段页式地址变换中软硬件作用关系

(1) 与分页和分段情况下一样,提供了虚拟存储器的功能;

(2) 因为以页架为单位分配主存,所以无紧缩问题,也没有页外的碎片存在;

(3) 便于处理变化的数据结构,段可动态增长;

(4) 便于共享,只要欲共享作业的段表中有相应表目指向该共享段在主存中的页表地址;

(5) 提供了动态连接的便利;

(6) 便于控制存取访问。

其主要缺点:

(1) 增加了硬件成本,因为需要更多的硬件支持;

(2) 增加了软件复杂性和管理开销;

(3) 同分页系统一样仍然存在页内碎片。

段页式存储管理技术对大、中型机器来说是使用得最广泛、最灵活的一种存储管理技术。

8.5 页(和段)的置换算法和系统行为

当发生缺页(或缺段),而主存中已没有空闲页架(或主存空闲块)时,则需要“选一页淘汰”。那么怎么挑选该页呢?通常把选取淘汰页(段)的方法叫页(段)的置换算法。这是多少年来引起广泛兴趣的一个研究课题,因为算法的好坏,直接影响系统的效能。若选用的算法不合适,可能会出现这样的现象:刚被淘汰出去的页,不久又要被访问,又需把它调入而将另一页淘汰出去,很可能又把刚调入的或很快要用的页淘汰出去了。如此反复频繁地更换页面,以致系统的大部分机时花在页面的调度和传输上了,系统的实际效率很低。这种现象称为“抖动”。

好的置换算法应能适当地降低页面更换的频率,尽可能避免系统“抖动”现象。常见的页面置换算法有:

(1) 最佳置换算法 OPT;

(2) 先进先出置换算法 FIFO;

(3) 最近最少使用置换算法 LRU;

(4) 最近未使用置换算法 NUR;

(5) 工作集。

下面讨论的算法虽以分页为例,但也适合分段和段页式存储管理情况,以及相关存储器(快表)中表目的置换算法。

8.5.1 最佳置换算法 OPT

最佳置换算法是由 Belady 于 1966 年提出的一种理论上的算法。其原则是“淘汰在将来再也不被访问,或者是在最远的将来才被访问的页”。但实际上这种算法无法实现,因为人们难以预知一个作业将要用到哪些页。所以这种算法只用来与其它算法进行比较。

正因为难以预知一个作业未来的访问页面的情况,所以以下的算法都是基于根据作业过去的访问页面的情况来推测其未来对页面访问的可能情况。由于各算法考虑的出发点不同,所以有不同的算法。

8.5.2 先进先出置换算法 FIFO

其基本原则是“选择最早进入主存的页面淘汰”。理由是最早进入的页面,其不再使用的可能性比最近调入的页面要大。

算法的实现比较简单,只要把进入主存的各页面按进入的时间次序用链指针链成队列,新进入的页面放在队尾。总是淘汰链头的那一页。除链指针外,也可用表格等方法来实现。

这种算法只是在按线性顺序访问地址空间时,才是理想的,否则效率不高。因为那些最早进入主存的页面常常有可能是最经常被使用的页(如常用子程序,常用的数组,循环

等)。先进先出置换算法的另一个缺点是,它有一种异常现象,这是 Belady 等人发现的。一般来说,对于任何一个页的访问顺序(或序列)和任何一种换页算法,如果分给的页架数增加,则缺页(所访问页不在主存)的频率应该减少。但这个结论并不普遍成立,Belady 等人发现,对于一些特定的页面访问序列,先进先出置换算法有随着分给的页架数增加,缺页频率也增加的异常现象,图 8.16 表示了这一异常现象:

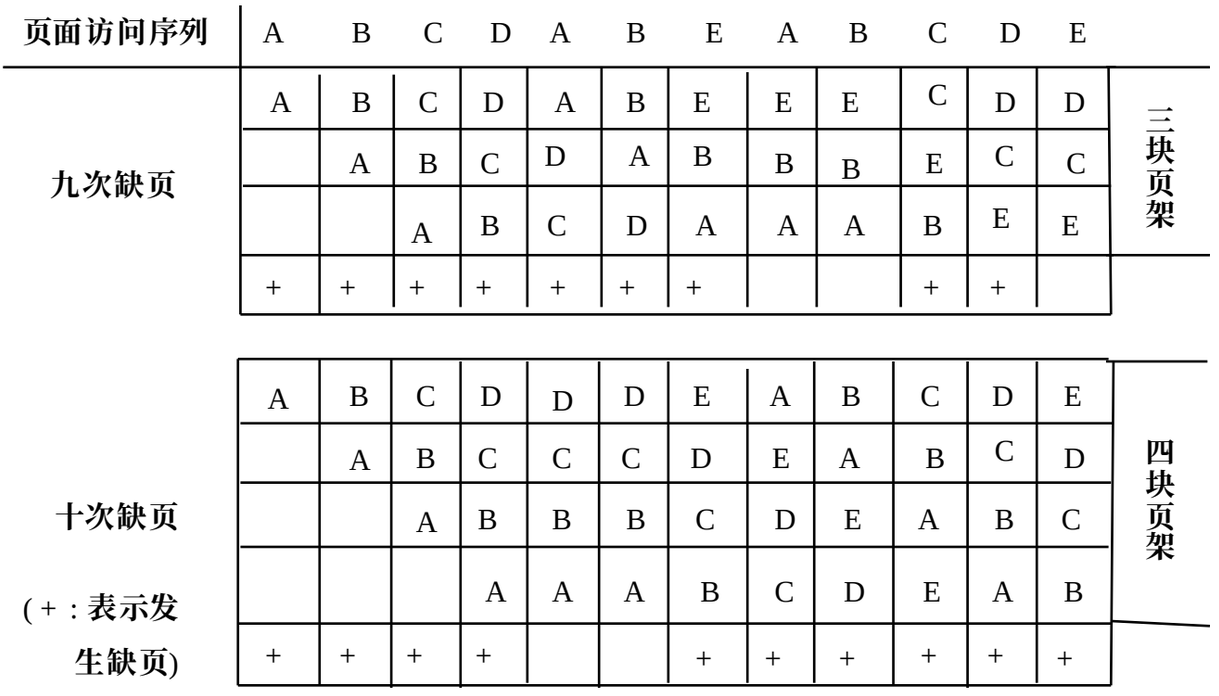


图 8.16 FIFO 的异常现象

异常现象本身对操作系统设计来说并不具有多大的意义。但无疑表明了这样一个事实,即操作系统是比较复杂的,操作系统的研究和设计中所反映出来的问题有时是同人的直觉相背离的。操作系统不只是实际技术和设计经验的总结,它更需要从理论上加以研究、发展。先进先出置换算法的异常现象可以通过栈式算法的蕴含特性加以解释。

8.5.3 最近最少使用置换算法 LRU

其基本原则是"选择最近一段时间内最长时间没有被访问过的页淘汰"。基本理由是认为过去一段时间里不曾被访问过的页,在最近的将来也可能不再会被访问。

该算法能比较普遍地适用于各种类型的程序。但是实现起来比较困难,因为要为每一页设置一特定单元来记录自上次访问后到现在的时间量 t,并选择 t 值最大的页淘汰。所以要对各页访问的历史情况时时加以记录和更新。如果修改、更新工作全由软件来做,系统的"非生产性"开销太大。如由硬件来做,则大大增加了硬件成本。所以实际使用的是这种算法的近似方法。

8.5.4 最近未使用置换算法 NUR

得到广泛使用的是最近最少使用算法的近似方法——最近未使用置换算法(NUR),它比较易于实现,开销也比较少。许多机器都采用此置换算法。在这些机器上,不但希望淘汰的页是最近未使用的页,而且还希望被挑选的页在主存驻留期间,其页面内的数据未

被修改过。为此,要为每页增设两个硬件位——访问位和修改位:

访问位= 0: 该页尚未被访问过

= 1: 该页已经被访问过

修改位= 0: 该页尚未被修改过

= 1: 该页已经被修改过

其具体工作过程如下:开始时,所有页的访问位、修正位都置为“0”。当访问某页时,将该页访问位置“1”。当某页的数据被修改,则该页的修正位置“1”。当要选一页淘汰时,挑选下面序列中属于低序号情况的页进行淘汰(为什么):

序号:	1	2	3	4
访问位:	0	0	1	1
修改位:	0	1	0	1

在多道程序系统中,主存中所有页架的访问位或修改位可能都会被置成了“1”,这时就难以决定淘汰哪一页。当前大多数系统的解决办法是避免出现这种情况。为此,周期性地(经一定时间间隔)把所有访问位重新清成“0”。当以后再访问页面时重新置“1”。这里一个主要问题是选择适当的清零时间间隔。如果间隔太大,可能所有页架的访问位均成为“1”。如果间隔太小,则访问位为“0”的页架又可能过多。

可以用栈式算法的蕴含特性证明LRU和NUR算法没有异常现象。

对于页的淘汰在IBM 4300系列中可分为局部淘汰和全局淘汰两种方法。前者仅在一个作业所占有的全部页面中挑选页面淘汰;而后者是对整个主存范围内的页面进行挑选。

并非主存中所有的页面均可选作淘汰页,由于各种不同的原因,有些页面必须永久地保留在主存中。如操作系统的常驻部分,系统表格等所在的页面就不应被换出。又如为了系统操作的完整性,与某些操作有关的页,在此操作期间也不应被换出。比如包含输入输出缓冲区的那些页,在输入输出操作期间和正在对缓冲区中的数据进行访问时,这些页就不应被换出。为标志这些页,IBM 4300系列设置了固定页标志,并分为长期固定和短期固定两种情况。

8.5.5 分页环境中程序的行为特性

8.5.5.1 局部性的概念

绝大多数的存储管理策略考虑的出发点是基于程序的特性——局部性概念。即进程对主存的访问远不是均匀的,而是高度地表现出局部性。它包含有两方面的内容:时间局部性和空间局部性。

(1) 时间局部性:是指某个位置最近被访问了,那么往往很快又要被再次访问。这一特性可通过程序中的循环,常用子程序,堆栈,常用变量这类程序结构来说明。

(2) 空间局部性:是指一旦某个位置最近被访问了,那么它附近的位置也要被访问。这一特性可通过程序中数组处理、顺序代码的执行,以及程序员倾向于将常用的变量存放在一起等特点来说明。

在操作系统环境中,程序的局部特性与其说是基于某种理论,还不如说是更多地基于

对程序特性的观察。

在分页环境中,程序访问的局部性表现为程序在某个时间内,对整个作业地址空间的各页的访问往往不是分散的、均匀的。而是比较集中于少数几页。随着时间推移,它又集中于另外的少数几页(可以与前面的几页的集合有相交部分)。而就一个页面而言,程序对一页中的各单元的访问也不是均匀的,也是集中于页中的较少部分。往往一个 1K 字的页中,大约只有 200 条左右的指令被访问。在程序局部特性的基础上,不少学者还对程序行为特性做了大量研究工作。

8.5.5.2 分页环境中程序的行为特性

由于分页是一个十分有价值的概念,世界上不少操作系统专家在过去的几十年中都在对分页环境下程序的行为特性进行了大量的研究和观察。尤其对进程占有的页架数与缺页频率关系的研究更使人感兴趣。图 8.17 表示了平均缺页时间(即发生两次缺页之间的平均时间间隔)与分给进程的页架数之间的关系。可以看出,此图形是单调递增的,即随着分给的页架数增加,而缺页间隔时间增大。但是此图形有一个拐点。拐点左边部分对应于进程只具有较少的页架,此时如果多分给其页架,则对进程的缺页间隔时间有明显改善。过了拐点后,即使多分给其页架,对进程的缺页间隔时间改善不大。一般把此拐点所对应的页架数看作是该进程的工作集的大小(见下节)。

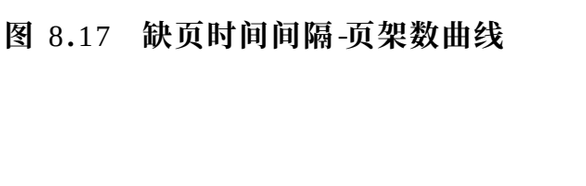


图 8.17 缺页时间间隔-页架数曲线

程序的这一特性对于分页环境下的存储管理十分重要。由于程序的"缺页时间间隔-页架数"曲线随程序而不同,所以许多人研究了该曲线的数学模型。Belady 曾提出过一个该曲线的数学模型如下:

$$G(x) = a\,x^{K}$$

其中 x 代表分给的主存大小,是个变量;a 为常数;K 为 1.5～3 之间的数值。

a,K 大小随程序而不同。这个模型在拐点以下同实际程序的曲线很一致。

8.5.5.3 减少访问离散性的程序结构

程序的行为特性与系统的缺页频率,以至整个系统的效率是有一定关系的。所以我们在进行程序设计的时候,如何编写高质量的程序,也应考虑程序访问的局部性如何提高,力求减少访问的离散性。

例如我们应注意数组在主存中存放顺序与使用顺序的一致性。假定在页面大小为 512 字的分页系统中,我们要把一个 512× 512 个元素的数组初始化为零,比较典型的一种编码方式如下(用 PASCAL 语言):

```
var A: array[1…512] of array [1…512] of integer;
    for j:= 1 to 512
do for i:= 1 to 512
    do A[i][j]:= 0;
```

这样的编码就不太好,因为数组在主存中是按行存放的,即 A[1],[1],A[1],[2],…,A[1],[512],A[2],[1],A[2],[2]…。而该程序使用的顺序是按列进行的,其使用顺序为 A[1],[1],A[2],[1]…A[512],[1],A[1],[2],A[2],[2],…。对于 512 字的页来说,每行刚好占一页。此程序是每页先初始化一个元素后进入下页,对下一个元素进行初始化。如果分给这个程序的数组只有一个页架,则将引起 512×512=262 144 次缺页。所以我们应修改编码如下:

```
var A: array [1…512] of array[1…512] of integer;
    for i:= 1 to 512
      do for j:= 1 to 512
        do A[i][j]:= 0;
```

这个程序只引起 512 次缺页,因为其存放顺序与使用顺序是一致的。

另外还应注意选择适当的数据结构以增加程序访问的局部性。如果从这个角度出发来探讨某数据结构的优劣,则堆栈是个好的数据结构类型,因为它的操作总是集中在栈顶,而链表相对差些。而一个 HASH 表的局部性也比较差。

同时我们也应注意加强编译程序和装入程序(Loader)的效能。例如编译程序最好能把程序代码和程序的数据分离开来,成为代码部分和数据部分这样两个不同的部分。这样易于保证代码部分是可再入的(纯的)、不被修改的(修改位常为 0),从而减少常使用的程序纯代码部分被换出的机会。而装入程序应将纯代码部分装入同一页(或几页)中,切不要把纯代码部分与非纯的代码或数据部分放入同一页中。这样可以减少那些常用子程序所在的页被换出主存。

8.5.6 工作集

根据程序行为的局部性理论,Denning 于 1968 年提出了工作集理论。工作集理论就其本质上来说是最近最少使用置换算法的发展。

在 8.5 节开始时,我们谈到置换算法的好坏直接决定了进程的缺页频率。当进程访问的页不在主存,这时就需要把该页调入主存。在调页过程中该进程需要等待(比较典型的情况是,从磁鼓上读入一页需要的时间是执行 20 000 条指令以上的时间),进程由就绪状态变为等待状态。如果缺页频率高,不但进程运行的进展很慢,大大增加了 CPU 非生产性机时开销,更增加了通道和外部设备的沉重负担,从而降低了系统效率,甚至引起系统抖动,直至瘫痪。因而如何降低缺页频率是一个很重要的研究课题。以前的各种置换算法并未直接从减少缺页频率方面来考虑,工作集正是从减少缺页频率方面来考虑的。所以从这个意义上来说,工作集也叫做工作集存储管理技术。

由于程序运行时,它对页的访问不是均匀的,而往往比较集中。在某段时间里,其访问范围可能局限在相对来说是比较少数的几页中。而在另一段时间里,其访问范围又可能局限在另一些相对较少的几页中(当然从这一个时间段到另一个时间段,进程所访问的页的集合的变化,往往是缓慢地逐步地过渡的)。因此如果能预知程序在某时间间隔中所要访问的那些页,并在该段时间前就把这些页调入主存,至该段时间终了时,再将其中在下一段时间里不再访问的哪些页调出主存。这样可以大大减少页的调入和调出工作,缩短等待

调页时间,降低缺页频率,从而能大幅度提高系统效率。

粗略地说工作集是进程在某段时间里实际上要访问的页的集合。程序要有效运行,其工作集必须在主存中。但是如何确定一个进程在某个时间的工作集呢?或者说,怎么知道一个进程在未来的某个时间段内要访问哪些页呢?实际上,计算机无法预知程序的行为,也就是无法预知要访问哪些页。我们仍然要依据程序过去的行为来估计它未来的行为(这种估计的依据就是程序行为的局部化特性,决定了工作集的变化是缓慢的)。所以把一个运行进程在 t- w 到 t 这个时间间隔内所访问的页的集合称为该进程在时间 t 的工作集,记为 W(t,w)。并把变量 w 称为"工作集窗口尺寸"。通常还把工作集中所包含的页面数目称为"工作集尺寸",记为©W(t,w)©¦。

可以看出,工作集 W 是二元函数。首先 W 是 t 的函数,即随着时间不同,工作集也不同。这包含两方面的含意:其一是不同时间的工作集所包含的页面数可能不同,也就是工作集尺寸不同。其二是不同时间的工作集中所包含的页面也可能不同(即有不相同的页面)。其次工作集 W 也是工作集窗口尺寸 w 的函数。工作集尺寸©W©是工作集窗口尺寸 w 的单调递增(确切说是非降)函数,且满足蕴含特性,即©W(t,w)©¦ ©W(t,w+ a)©¦其中 a> 0。

正确地选择工作集窗口尺寸的大小对工作集存储管理策略的有效工作是有很大影响的,这也是工作集理论研究中的重要领域。如果 w 选取过大,甚至把整个作业地址空间全部包含在内,就失去了虚存的意义。w 选取过小,则将引起频繁缺页,降低了系统效率。

所以不少学者认为,根据分页环境下程序行为特性,程序的工作集大小可粗略地看成对应于"缺页间隔时间——页架数"曲线的拐点。在程序执行期间,如果想要实际地确定程序当时的工作集尺寸是可能的。只要从某个时间 t_1 起,将所有的页面访问位全清零(用特权指令)。而后到时间 $t_2 = t_1 + w$ 时,记下全部访问位为"1"的页,这些页的集合可看作 t_2 时的工作集。

正确的策略并不是消除缺页现象,而应使缺页间隔时间保持在合理水平。当此间隔过小时,应增加其页架数。过大则应增大多道程度,减少分给进程的页架数,以提高整个系统的效率。所以工作集策略也可如图 8.17 中所示的那样,把缺页的间隔时间控制在合理的范围,使分给进程的页架数保持在上、下限之间。

8.6 页架的分配算法

当进程缺页而请求分配一个页架时,我们仅仅说分给一个空闲页架或淘汰出去一个页面,而未涉及任何细节。本节中进而探讨某些页架分配中的具体问题。

8.6.1 提前分配

在分页或段页式系统中,存在一个明显的问题就是当程序刚一被启动时所引起的大量的持续的缺页问题。解决此问题的方法是启动前,提前把该进程的最初的工作集装入主存。这有两方面的问题,一是系统有无能力主动装入某些指定的页面,而不是只能按缺页请求发生时才能将该页装入;二是将哪些页提前装入。对于前一问题,几乎所有分页系统

都有按指定页号提前装入该页的命令可供使用。对于后一问题,如果此系统是属于“滚进滚出”的分时系统,当进程被再启动时,我们可以提前将它上次滚出时的(或挂起进程被挂起时)工作集装入即可。这样就要求系统每当挂起一个进程或滚出一个进程时,要对它在主存中的页面(工作集中的各页)进行记录。这个方法也可使用于因等待事件(如等待输入输出)而阻塞的进程之中。

8.6.2 最少页架数

有人曾以为在请求分页系统中,进程在主存中即使只分给它一个页架,它也可以运行。这种看法是片面的。通过前面的讨论我们知道,系统要能有效地工作,每个进程的工作集应在主存中。姑且不论进程的工作集能否保证在主存,实际上即使从一条指令能否被完整地执行这样一个角度来考虑,进程也应分得起码的页架数——最少页架数。

进程应分得的最少页架数随着计算机的结构不同而不同,这主要取决于该机器的指令格式和寻址方式。例如对于 PDP-8 来说,它是单地址指令,如果是直接寻址方式,则最少要两个页架来放指令执行所需的页面。一页是指令,另一页是操作数。如果是间接寻址方式,则可能要三页。这样每个进程最少页架数为 3。而 PDP-11 则最少要求六页,因为对于某些传送指令长度可能超过一个字,所以它本身可能骑在两页上,再加上源地址和目标地址都是间接访问形式,为了能在逐次发生缺页中断后,把此六页全读进主存,重新启动这条指令的执行,所以每个进程最少要有六个页架。

8.6.3 局部和全局分配

在讨论页面淘汰策略时,我们研究了不同的一些淘汰算法,对于最近不使用(NUR)算法来说是淘汰那些访问位和修改位均为 0 的页,但并没有说明,这样的页是从整个主存中选择还是只从分给自己的那些页中去选择,我们已经说过,这有两种不同的方法,即全局分配(或称全局淘汰)和局部分配(或称局部淘汰)。所谓全局分配是从整个主存中去选择淘汰页。而局部分配是只从自己所已占有的页架中去选择淘汰页。

很显然,在局部分配情况下,一个进程所占有的页架数是不变的。它的缺页情况取决于它已分得的页架数和它自己的程序行为特性。而在全局分配情况下,进程可以选择别的进程的页面进行淘汰,而且往往是必然选择别的进程的页面,因为那些页面通常具有较长时间未被访问的特性。这样一个进程在主存的页面情况不但取决于自己的行为特性,也取决于其它进程的行为特性,这会使得同一个程序在主存中各次的重复执行情况十分不同,而且执行的速度是十分难以预料的。

8.6.4 分配算法

如果是局部分配策略,也就是说从进程自己所占的页架中选择淘汰页。那么我们如何把系统中全部可用页架分配给各个进程呢。当然最简单的方法是把系统中全部页架数 m,在几个进程中平均分配。例如 95 个页架给 6 个进程分,每个进程分得 15 个页架,剩下的 5 个页架作为公用页架缓冲池,它可在必要时按全局分配分给进程使用。

另外一个方配算法是考虑作业本身大小按比例进行分配。因为上述等分的分配算法

明显的不合理,对于只有 5K 的学生作业,上述每个进程分 15K 的方法,就会浪费 10K。而对于一个 64K 的较大的作业,15K 又嫌太少。按比例分配方法如下,设每个进程的虚存大小(即作业大小)为 s_i,则各进程的虚存之和

$$S=\sum s_i$$

假定主存可用页架数为 m,则进程 P_i 分得的页架数 a_i 为

$$a_i=\frac{s_i}{S}\times m$$

当然 a_i 应是整数,并且应大于对应于该计算机指令执行所必须的最少页架数。

上述两个分配算法中均没有考虑进程的优先级。在主存页架的分配中应该考虑进程优先级的因素,以保证优先级高的进程分得更多的页架,减少其缺页次数,提高它的处理速度。考虑优先级的具体执行方法,或者提高高优先级进程的分配比例,或者允许高优先级进程可以淘汰低优先级进程的页面,后者自然应使用在全局分配情况下。

实际上许多分页系统中,既将大部分页架按一定算法分给各进程及系统本身,另外还保留一部分页架作为全局分配和系统缓冲之用。

8.6.5 页的大小

分页系统中主存被分成固定大小的页架,然而页架应选多大为好?或者说页面尺寸以多大为好。在决定页面尺寸过程中通常要考虑以下因素:

(1) 较大的页面尺寸将增加页内碎片的消耗,同时大页面会使缺页频率增加(在分给作业的主存大小一定的情况下)。这样看来以小页面为好。

(2) 较小的页面尺寸将使整个主存的页架数增加,并导致需要更多的页表空间。

(3) 由于缺页时,从外存读入一页相对来说比较费时间,这是由于进程的虚拟页面大多存放于磁盘上,而从磁盘上读一个数据块所需的时间粗略地可看成两部分组成:一部分是将磁臂定位到所需的柱面的延迟时间并加上该盘块旋转到读写头下的旋转延迟时间,称为总的延迟时间。第二部分是该数据块传送到主存的时间。所以总的传送时间是二者之和,而延迟时间在其中约占(80～90)%。正因为磁盘的定位延迟时间占这么大的时间比例,为了尽量减少这种输入输出工作,所以倾向于大的页面尺寸。

综上所述,目前许多文章从理论上和实践上都认为页面尺寸以小些为好,表 8.1 列出了几种著名的微型计算机和大、中型计算机的页面尺寸。其中微型机 NS 32032 与小型机 VAX-11/780 的页面大小是具有典型性的。

表 8.1 几种著名计算机的页面尺寸

制造厂家	型　　号	页面大小	单　　位
NS 公司	NS 32032	512	B(8 位)
Zilog	Z 8000	1 K	B
日本电气	V 70	4 K	B
Motorola	MC68020	256～4K(8 种)	B

续表

制造厂家	型　　号	页面大小	单　　位
DEC	VAX11/780	128	字(32 位)
DEC	PDP10,20	512	字(36 位)
IBM	370/168	512 或 1 K	字(32 位)
IBM	360	1 K	字(32 位)
Honeywell	MULTICS	1 K	字(36 位)

* 8.7　高速缓冲存储器

为了提高存储器存取数据的速度,使之能与处理机的处理速度相匹配,许多计算机和微型计算机系统都采取了各种措施,使用高速缓冲存储器是其中的一个重要措施。现在许多系统都采用三级存储结构,即在中央处理机和处理机存储器(主存)之间增加了一个高速缓冲存储器(以下简称缓存)。缓存不参与存储器编址,对用户是透明的,所以又叫隐藏存储器(Cache 存储器)。

目前具有高速缓冲存储器的微型计算机很多,例如有 MC 68020, Z 8000 和 Intel 80386 等 32 位机,它们的缓存大小为 2K～32KB。不论大型计算机还是微型计算机,其高速缓冲存储器的基本组织和工作原理是相同的。(缓存技术并不属于虚拟存储技术,因用到本章的知识,故放在本章中进行讨论。)

8.7.1　高速缓冲存储器的组织

高速缓存的典型容量是 4K, 8K 和 16KB, 而 80386 的缓存为 32KB。其速度比较如表 8.2。

表 8.2　处理机、高速缓存、主存速度比较表

	主存	高速缓存	处理机周期
读双字指令	1100ns	120ns	150～300ns

可以看出,高速缓存的速度比主存高一个数量级,同处理机速度相匹配。

缓存的结构,以及缓存、主存与处理机关系如图 8.18 所示。由图可知,高速缓存由缓

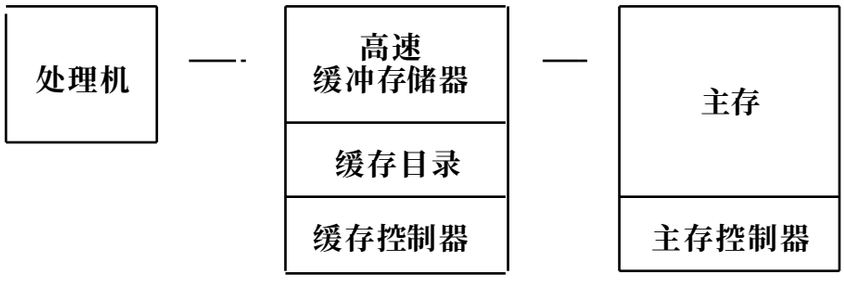
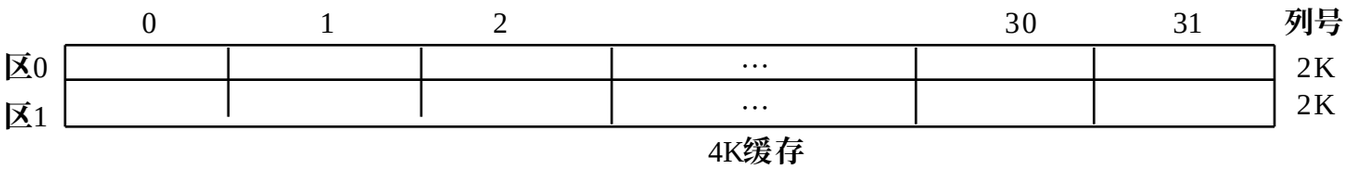


图 8.18　高速缓冲存储器的硬件结构

冲存储器,缓存目录和缓存控制器三部分构成。通常缓冲存储器和主存都分为若干块,假定每块大小为 64 个字节,(Intel 80386 的块大小为 32 个字节)。对于 4KB 的缓存,分为两个区(图 8.19),8KB 缓存分为四个区,16KB 的缓存分为八个区。每区 2KB。每 2KB 的区中包含有 32 个块,同一区中的各块用列号来标示。



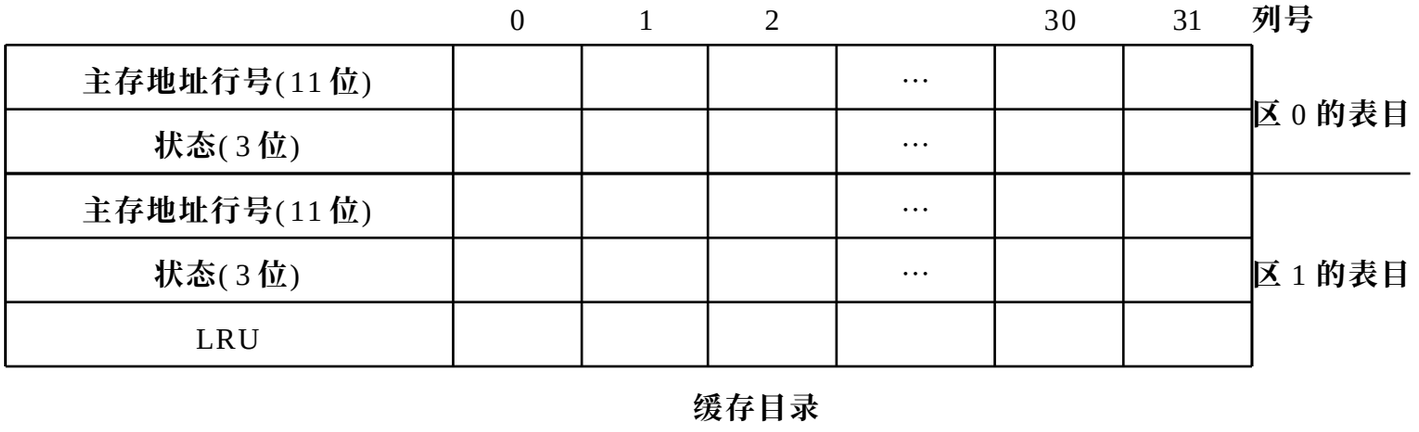


图 8.19　4K 缓冲存储器的结构形式

不论 4K,8K,16KB 的缓存,其缓存目录都只有 32 个表目(对应于缓存的 0～31 的列号)。4K 缓存目录的表目同样分为两个区(8K 分四个区,16K 分八个区)以对应于缓存的区号(或称行号)0～1。所以实际上缓存中的每一块都对应一个自己的固定的表目。目录中的三位状态位分别为:有效位、修改位、故障位。

有效位:该位为 1 时表示该表目所对应的缓存块中的数据已经是无效的。该位为 0 时,则对应块中的数据是有效的。

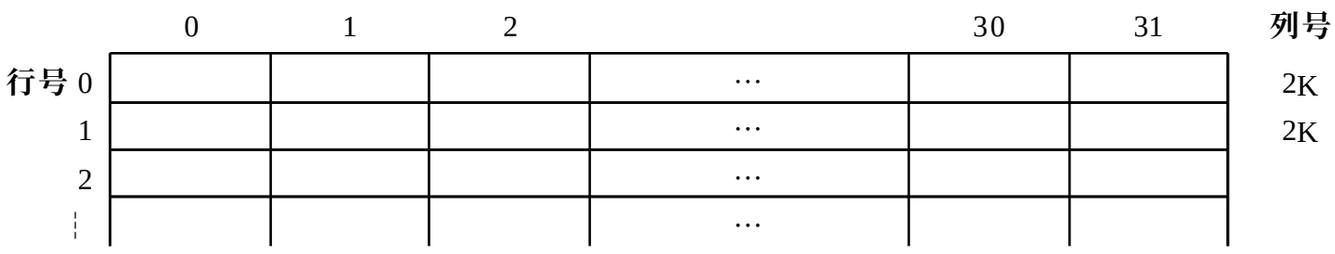
修改位:该位为 1 时表示该表目所对应的缓存块中的数据已经被修改过。

故障位:该位为 1 时表示该表目所对应的缓存块出了故障。

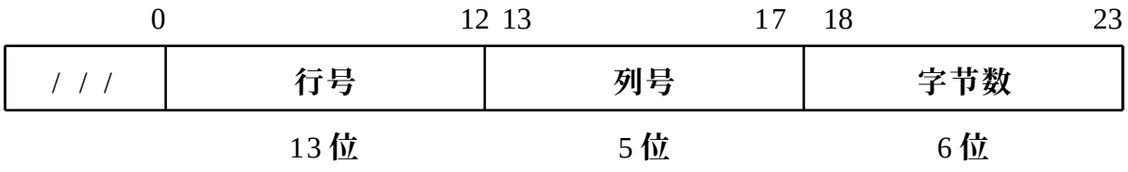
LRU 表示最近访问了两区中的哪一区,以作为选择淘汰块使用。

8.7.2　缓存块的编址形式

8.7.1 节已指出,主存和缓存均分成 64 个字节的块。因此我们可把主存储器看成如图 8.20(a)所示的形式。把每个主存地址按图 8.20(b)格式解释。



(a)



(b)

图 8.20　主存地址格式

附带介绍一下,在 80386 的编址形式为行号(17 位),列号(10)位,字节数(5 位)。以供读者参考。

8.7.3　缓存的工作过程

缓存的工作过程按操作类型描述如下:

(1)取指令:当处理机的指令处理功能部件要取指令(或数据)时,缓存控制器(图 8.18)就自动查找缓存目录,以确定包含该指令(或数据)的主存块是否在缓存之中。其方法是:

① 根据指令(数据)地址的 13～17 位以决定查找目录的表目号(列号)。

② 然后把该表目中分属区 0 和区 1 的主存地址行号先后与指令(数据)地址的 0～12 位(行号)进行比较。如果匹配,并且状态中的有效位为“0”,则把匹配表目所对应的缓存块中的内容直接送指令处理部件。并且把 LRU 位置成表示相应块最近已被访问(如 LRU = 0,表示区“0”的块是新近被访问的,= 1 表示区“1”的块是新近被访问的)。如两个区的相应表目(列号)中的主存地址行号都不与之匹配,则说明该块信息不在主存,于是从主存中把该块内容取到处理机的指令处理部件,同时该块内容也被送到缓存中的相应列的某块内。

(2) 写数据:当处理机要求存数据到某主存单元时,如包含此单元地址的主存块已在缓存中(由缓存控制器查缓存目录,过程同上),则处理机的指令处理部件把在缓存中的该块内容加以修改(写操作),并把缓存目录中相应表目(列号)的修改位置“1”。这里要注意的是 Windows NT 处理写数据的方法不同于 IBM 370,而是采用“惰性(lazy)”方法,即数据存入缓存后,并不马上去修改相应主存块中的内容,而是直到该缓存块内容被淘汰出缓存时,才把该缓存块写回主存相应块中(而 IBM 370 是用“立即存”方法,即对主存和缓存的相应块同时写)。若该块不在缓存中,则先把该块从主存读入缓存,然后再在缓存中修改。

(3) 通道向主存某单元写入数据时,该数据只被放在主存中。但缓存控制部件同时查找缓存目录。假如该主存单元所在的块在缓存中,则相应表目的状态中的有效位被置成“1”(表示该块数据无效)。

(4) 通道从主存读数据时,则查找缓存目录,假若包含该数据地址的块在缓存中,则从缓存中把该块送往通道。如不在缓存中,则从主存读出,但并不把该块放入缓存中。

最后,要指出的是,本节讲述缓冲存储器虽以 IBM 4300 系列为例。但目前 32 位的微处理器中无一例外地均在处理器与主存间设置了高速缓冲存储器,从而使 CPU 所需的

指令与数据绝大多数来自缓存,而不通过总线去访问主存。这就避免了处理机的等待,大大提高了系统的吐吞率。因而缓存已被看成是充分发挥 32 位微处理器潜在能力的一种有效方法。目前微处理机对缓存的命中率均在 98% 以上,这样高的命中率也是由于程序访问的局部性所带来的。

8.8 Windows NT 的分页机构、页面调度算法和工作集、共享主存机制

Windows NT 的虚拟存储管理十分出色。提供了卓越的功能和系统性能,采用了精妙的算法。

8.8.1 Windows NT 的二级页表地址变换机构

Windows NT 是 32 位操作系统,它为每个进程提供了一个很大的虚拟地址空间(2^{32} = 4000MB),虚拟地址空间的结构请参阅第 12 章图 12.5。每个进程有自己的虚拟地址空间,该地址空间平均分为两部分:系统存储区(2GB)、用户存储区(2GB)。

Windows NT 采用虚拟分页技术。其虚拟地址由三部分组成(参阅第 12 章图 12.6)

目录位移: 22～31 位,共 10 位

页表位移: 12～21 位,共 10 位

页内位移: 0～11 位,共 12 位

NT 的页面大小为 4K(2^{12})B。因此每个进程可以有 2^{20} = 1 048 576 个页面。如果一个进程的页表要常驻主存(通常的分页系统其页表是常驻主存的)的话,那么可能要使用最多达 1 兆多个页表表目(每个表目在 NT 中为四个字节),共需页表空间可达 1024 个页面(4 兆多字节)。这是一笔惊人的主存开销。为此,NT 采取了两个措施:

(1) 页表不常驻主存,可换进换出;

(2) 采用二级页表地址变换机构。即由系统的控制寄存器指出当前进程的页目录表地址,在页目录表的表目中指出二级页表的地址(页目录表的表目以及二级页表表目均占四个字节)。所以根据虚拟地址结构,每个进程的页目录表刚好 4K 大小(2^{10} = 1024 个表目)。每个二级页表也刚好 4K 大小(2^{10} = 1024 个表目)。具体地址变换过程请参阅12.7.2 节和图 12.6。

为了提高系统的访问速度(请问主存访问速度降低了多少),系统使用了高速缓冲存储器(32K 或 64KB)和快表。这使得系统性能大为提高,系统存取数据和指令的速度与处理器的处理速度完全匹配。

8.8.2 页面调度算法和工作集

页面调度算法包括取页策略、置页策略和淘汰(置换)策略。

取页策略:提前取页与请求取页相结合,并采取集群方法提前取页。NT 采取这一策略的理论根据是程序行为局部性理论。

置换策略: 采用局部置换策略;

使用 FIFO 置换算法;

使用“自动调整工作集”技术。

系统性能考虑:为了提高系统性能,避免不必要的费时的操作,采用了“惰性(lazy)技术”。详细情况请参阅 12.7.3 节和 12.7.4 节。

8.8.3 共享主存机制——段对象、视口和映象文件

Windows NT 采用了出色的共享主存机制,该机制有以下特点:

(1) 共享进程通过创建“段对象”作为共享的主存区域(主存区域是对象类)。

(2) 段对象的大小可远大于进程虚拟地址空间大小,达 2^{64} 个字节。这对很大的图象和多媒体应用情况十分有利。

(3) 由于进程的虚拟地址空间可能比段对象小得很多,所以系统提供“视口(view)”机制,使一个进程可逐个视口的访问段对象或访问所要的特殊视口。

(4) 通过映象文件 I/O 技术,使得进程访问段对象的速度,如同访问主存中的数组一样快。读者如需了解详细情况请参阅 12.7.5 节。

(5) 每个虚拟页面均有存取保护信息,实现访问监控。

习　　题

8-1　说明将进程的逻辑地址空间(虚拟地址空间)与物理地址空间分开是有利的。

8-2　何谓虚拟存储器?其容量通常由什么因素决定,虚拟存储器容量能大于主存容量与辅存容量之和吗?

8-3　比较下述几种虚拟存储映象技术的优缺点:

(1) 直接映象;

(2) 相关映象;

(3) 直接与相关相结合的映象。

8-4　请画出分页情况下地址变换过程,并指出页面尺寸为什么必须是 2 的幂?

8-5　说明在分页、分段和段页式虚拟存储技术中的存储保护是如何实现的?有无碎片问题?

8-6　说明请求分页情况下的装入策略、放置策略、置换策略。并指出 Windows NT 操作系统中是如何做的?

8-7　比较 FIFO 与 LRU 置换算法的优缺点。

8-8　什么叫动态连接?有何优点?指出动态连接过程。

8-9　如果存放页表区域也是分为大小相等的块(页架),每个进程的页表可能要存放在好多块中。请问这种情况下应如何构造页表?

8-10　为什么全局更换策略比局部更换策略对页面抖动更敏感(更易产生)?

8-11　下述每种硬件特性在虚拟存储中的使用情况及特点:

(1)地址变换机构;　　(4) 引用位;

(2) 关联存储器;　　(5) 修改位;

(3) 高速缓冲存储器;　　(6) 正在传输位。

8-12　讨论在分页系统中,程序设计风格如何影响性能? 请考虑下述各项:

(1) 自顶向下的方法;

(2) 少用 go to 语句;

(3) 模块化;

(4) 递归;

(5) 迭代法。

8-13　Windows NT 操作系统为什么要采用二级页表结构? 其地址变换有何特点?

8-14　Windows NT 的虚拟存储管理是如何使访问主存数据的速度与处理器的处理速度相匹配的?

8-15　Windows NT 的虚拟存储管理的‘自动调整工作集’技术是如何提高系统性能的?

8-16　说明什么是程序行为局部性概念? 其根据是什么?

8-17　Windows NT 的共享主存技术有何特点?

第 五 部 分

设备和文件管理

第 9 章　设 备 管 理

9.1　输入输出组织和输入输出处理机

现代计算机系统都具有种类繁多的外部设备(输入输出设备,通信设备,数据采集的传感设备),它们在整个计算机系统的成本中占相当大的比重。所以用好、管理好系统中的设备也是操作系统主要功能之一。

计算机的输入输出组织对微型计算机和大型计算机是有所不同的,图 9.1 中表示了微型计算机的基本的输入输出组织形式。数据的输入是由输入输出(以下简称 IO)设备,经由输入输出接口(以下简称 IO 接口)、通过数据总线传送到处理机的累加器之中。而数据的输出则刚好相反,由累加器通过数据总线再经过 IO 接口而传送到 IO 设备。而大型计算机的输入输出组织如图 1.3 所示,数据的传送是经由存储器,输入输出通道,设备控制器而到达设备的。取数据时的路线则刚好相反。在微型计算机中,如果是打印机,终端等慢速字符型设备,其与处理机的数据交换路线如同上述那样。但是对于那些像磁盘、磁带那样的高速大容量存储设备来说,其数据传送路线类似于大型计算机中的方式,是由处理机到存储器,然后经由设备接口,控制器(称 DMA 控制器)而到设备。取数时的路线则相反,这种数据存取方式称为直接存储访问,简称为 DMA(direct memory access)。

以下简略介绍主要的 IO 设备及其特点。

9.1.1　输入输出接口(IO 接口)

每个 IO 子系统都是由 IO 设备和 IO 接口组成。IO 设备是为计算机执行某种特定功能(打印,显示、存或取数据),而 IO 接口则是控制 IO 设备按 CPU 的命令工作,它也负责

图 9.1　微型机的输入输出组织

把计算机的数据格式转换成 IO 设备所要求的格式(或者相反),同时还负责发送中断请求并接受处理机发来的中断响应回答等工作,IO 接口还有为 IO 设备提供数据缓冲的功能。所以在 IO 接口中通常包含有数据缓冲寄存器和状态寄存器。IO 接口常与 CPU、存储器做在同一块板子上。

9.1.2　输入输出处理机(通道)

输入输出处理机又称通道,在大型机的结构中和通常的教科书中,术语“通道”专指专门用来负责输入输出工作的处理机(简称 IO 处理机)。比起中央处理机 CPU 来,通道是一个比 CPU 功能较弱、速度较慢、价格较为便宜的处理机。但是“通道”一词在微型计算机的有关著作中,常指与 DMA(直接存储器访问)或与 IO 处理机相连设备的单纯的数据传送通路,它并不具有处理机的功能,这是需要区分清楚的。

通道既然是指 IO 处理机,那么它同中央处理机 CPU 一样,有运算和控制逻辑,有累加器、寄存器,有自己的指令系统,它也在程序控制下工作,它的程序是由通道指令组成的,称通道程序。IO 处理机和 CPU 共享主存储器。在微型计算机中,IO 处理机并不是通道,它就称为 IO 处理机。但本节(以至于本书中)中所谓通道就是指 IO 处理机。

在大型计算机中常有多个通道,而在一般的微型计算机中则可以配置 1～2 个 IO 处理机(或更多)。这些 IO 处理机和中央处理机共享主存储器和总线(微型机中多采用总线结构)。在大型机中就可能出现几条通道和中央处理机同时争相访问主存储器的情况。为此给通道和中央处理机规定了不同的优先次序,当通道和中央处理机同时访问主存时,存储器的控制逻辑按优先次序予以响应。通常中央处理机被规定为最低优先级。而在微型计算机中,系统总线的使用是在中央处理机控制下,当 IO 处理机要求使用总线时,向中

央处理机发出请求总线的信号,中央处理机就把总线使用权暂时转让给 IO 处理机。以上两种情况,都可以想象为 IO 处理机从中央处理机那里“窃用”了存储周期和总线周期,统称为“窃用周期”。

通道既然是一个处理机,它就有它自己的指令系统(通道的指令系统比较简单,通常只有些数据传送指令;设备控制指令;转移指令等),并且通道也是在程序控制下工作的,这个程序叫通道程序。通道程序由中央处理机按数据传送的不同要求自动形成的,常常通道程序只包括少数几条指令(在大型计算机中,通道程序常由操作系统中的相应的设备管理程序按用户的输入输出请求自动生成)。CPU 生成的通道程序存放在主存储器中,并将该程序在主存中的起始地址通知 IO 处理机。在大型计算机中,通道程序的起始地址常存放在一个称为通道地址字(CAW)的主存固定单元中,而在微型计算机中,常将此起始地址存放在主存中的 CPU 与 IO 处理机的通信区中(如 Intel 8088)。每一条通道指令称为通道命令字 CCW。

通道作为处理机,也有说明处理机状态的程序状态字 PSW,但常被称为通道状态字 CSW。通道状态字 CSW 中包含有该通道及与之相连的控制器和设备的状态,以及数据传输的情况。

9.2 辅助存储器

又称外部存储器,常指磁带,磁鼓和磁盘(包括软磁盘)。

9.2.1 磁带的硬件特性及信息的组织

磁带和磁带机的原理类似于家庭中用的录音磁带和录音机。磁带是用来记录和存取信息的物理介质,它是在塑胶带制成的载体上涂抹一层薄薄的磁性材料。磁层厚度通常在 0.2～5 微米之间。这层磁性材料随着载体作高速回转或直线运动。磁带机是一个启停设备,平时处于停止状态。当要求从磁带上读取信息时,磁带机才被启动,并由磁头在磁带运动中对磁带进行读和写。磁头是由高导磁的软磁材料做成的电磁铁。在读、写磁头上分别绕有读写线圈,在磁头和磁层间有一个很窄的气隙口。

当要写入信息时,根据写入信息的不同(0 或 1),在写磁头上通以不同的脉冲电流。在磁层沿磁头相对运动的方向上形成记录信息的路径,称为磁道。磁道一般与磁头宽度一致,在每个磁道上串行地存储着一系列二进制代码。在每个磁道上都分别装有读写磁头。于是就可以对这些磁道上的信息进行并行读写。

磁带的宽度可以有几种不同的尺寸,常见的有 1,1/2 和 1/4 英寸。带的长度也是可变的,目前国产磁带有 600 米、750 米和 900 米三种。一般称磁带横方向的磁道数为其横向记录密度,常见的有 7 磁道、9 磁道和 16 磁道等。图 9.2 表示 9 磁道的磁带中的信息组织,在一个字节为 8 位的计算机中,通常磁带上的一个磁架包含一个字节,额外加一位作为纠错码或检验码用,纠错码是为提高可靠性而设置的。磁带的纵向也规定了记录密度,目前国产带的纵向记录密度,归零制时为 10 位/毫米,不归零制时为 20 位/毫米(所谓归零制是指写 1 时,在写磁头线圈中通以正脉冲电流,写 0 时通以负脉冲电流,每写完一位,

电流回到零,故称归零制)。

现将国产带的主要技术指标抄录如下:

带速　　2m/s

带宽　　1in, $\frac{1}{2}$in 和$\frac{1}{4}$in

带长　　600m, 750m 和 900m

横向密度　7 磁道, 9 磁道和 16 磁道

纵向密度　10bit/mm 或 20bit/mm

启停时间　少于 15ms

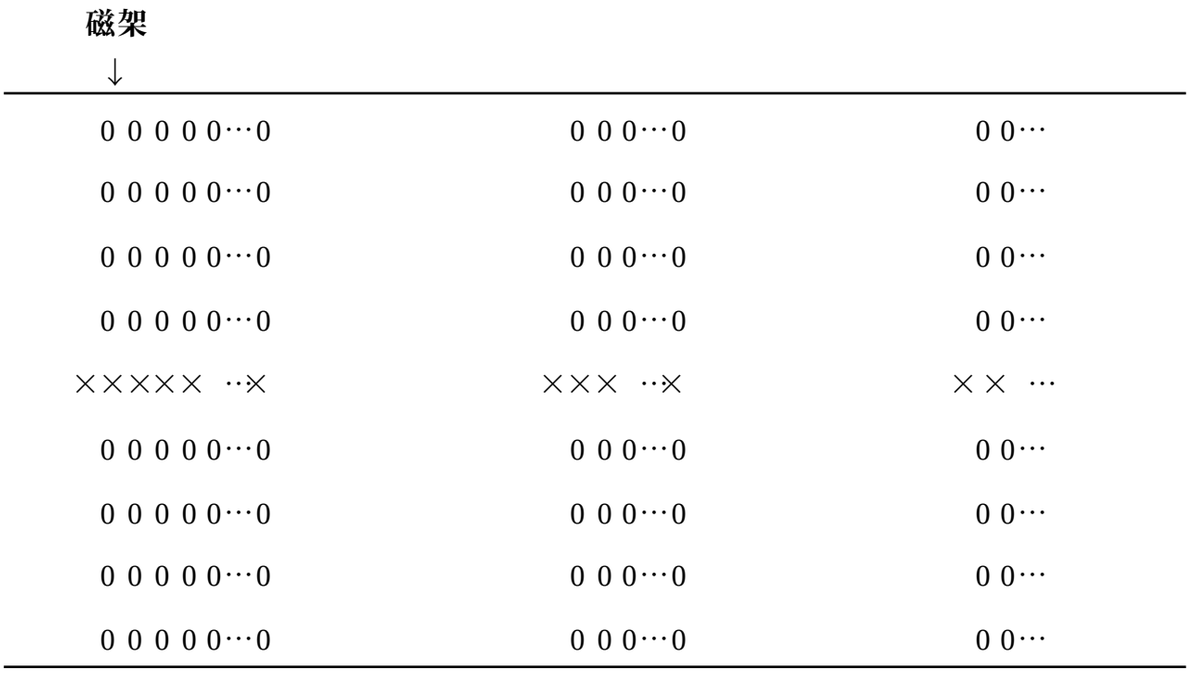


图 9.2　9 磁道磁带信息分布

0: 表示信息码,× 表示纠错码

磁带上信息的组织同磁带的使用方法有关,通常磁带的使用方法有两种: 一是把它作为顺序性设备; 另一是把它当成半顺序性设备。

1. 顺序性磁带的信息组织

所谓顺序性磁带是指磁带上信息的定位和存取严格按其上信息的物理位置顺序进行。

由于磁带是启停设备,平时磁带处于停止状态,当启动磁带机后,磁带才高速运动。只有磁带加速到正常速度时,读写头才能正确地进行信息的读写。所以为了读停止的磁带上的下一个记录,要有一定的时间来驱动磁带。这也就是为何磁带上两个相邻记录间要有一定的间隙,通常这个间隙从 1/4 英寸到 3/4 英寸。对于一个特定的设备的间隙是常数。这样为了提高磁带的利用率,减少间隙浪费的百分比,我们常常把磁带上的记录组成块的形式进行存放(即把多个逻辑记录放进一个物理记录块中)。通常块的长度为 800 到 8000B。块如果太大,一则块中信息的可靠性需要考虑,再则读进内存时将会占用太多的存储空间。

顺序性磁带上的信息也是以块为单位存取信息的,其信息组织如图 9.3 所示,图中的带头标和尾标用来标识带的起始端和末端,这两个标志通常用两条金属箔贴在磁带反面

以供光电系统识别。带标(或称块尾标)是一组供硬件识别的较短的特殊代码信息块。相当于有效信息块的间隔符。对于顺序性磁带来说,带上各信息块的长度可以是不等的,通常规定了最大和最小长度,一条通道命令通常只读、写一块信息。

图 9.3　顺序带的信息组织

2. 半顺序性磁带的信息组织

它与顺序性磁带的信息组织的差别主要在于信息块长度是相同的,一般为 512 个字或 1024 个字,一块称为一个信息组,并给每一组规定一个组号以标识和定位信息组。在磁带使用前,由磁带机将组号写在每个信息组前,其信息组织如图 9.4 所示。由于这种带上存在唯一标识信息组位置和组号,所以只要把要读、写的信息组号和读、写长度通知磁带机,它即能从该信息组开始,顺序地读、写信息。所以称为半顺序性。

图 9.4　半顺序带的信息组织

9.2.2　磁鼓的硬件特性及信息的组织

磁鼓是一种形状如鼓的设备,分为立式磁鼓和卧式磁鼓两种,主要根据转子轴是垂直于水平面,还是平行于水平面来分。磁转子是由鼓体上涂以磁层构成。其记录信息的原理与磁带相似。鼓表面被划分为许多条首尾相接的环形磁道,磁道密度一般为 4～5 道/厘米。每条磁道上对应有一个固定的读写头,鼓上的信息就是由这些磁头写上和读出的。由于磁鼓不停地绕轴高速旋转,所以每条磁道上的信息反复通过固定不动的相应的磁头,因而这些磁头能读写鼓上全部磁道的信息。磁鼓是属于可直接存取的存储设备。为使硬件能识别记录在磁道上信息的位置,通常规定磁鼓表面上的某一条母线为各磁道记录信息的起点。每条磁道从起点开始,沿圆周方向划分为若干个信息区作为记录信息的基本单位,这样的单位称为物理段或物理记录。每一物理记录中记录着连续的信息项。每条磁道上的物理记录从起点开始进行顺序编号: 0, 1, 2, ……。因而对磁鼓上的一个物理记录定位,需要有两个参数: 道号,物理记录号。

作为旋转设备来说,磁鼓的容量比磁盘为小,然而由于每个磁道有一个固定磁头,所以记录查找时间很小,数据传送速率较高,有些磁鼓为了进一步提高数据传送速率,每个磁道有两个或更多的读写磁头。

为使读者对磁鼓的技术指标有一个概念,下面列出 IBM 4300 系列配置的 2303 型磁鼓的主要技术性能:

磁道数　　800 道
每道容量　　4892B
每转所用时间　17.5ms/r
最大查找时间　17.5ms
平均查找时间　8.6ms
数据传送速率　0.303 百万字节/s

磁鼓目前已很少使用,但由于其与固定头磁盘的相似性,所以仍加以介绍。

9.2.3　磁盘的硬件特性及信息的组织

计算机中的磁盘,是由铝合金制成的金属圆盘,表面上(双侧)涂以磁性材料构成的。磁盘上一系列记录信息的同心圆称为磁道,通常每面有数百个磁道。磁盘盘面一般都划分成若干相等的扇形,这些扇形将每条磁道分割成许多相等的弧段,每个弧段被称为一个扇区或物理记录。虽然同样的扇区由于所处的磁道位置不同,其实际的物理长度不同(离圆心近的磁道上的扇区长度短于离圆心远的)。但它们所记录的信息量却是相等的。为了定位方便,将每个磁道上的扇区编以顺序的序号 0, 1, 2, …, n。

磁盘系统可分为两种基本类型:固定头磁盘和移动头磁盘。所谓固定头磁盘是指盘面上的每一条磁道都有一个读写头,固定头磁盘的操作类似于磁鼓。所谓移动头磁盘是指每个盘面只有一个读/写磁头,每执行一次盘的操作都须先移动磁头,使其对准所要找的磁道,这称为寻找操作。

通常在计算机系统中所说的磁盘,是由若干片盘所组成的磁盘迭(又称磁盘组),其结构如图 9.5 所示,各盘片均安装在一个高速旋转的枢轴上(转速可达 3600 转/分)。读写头安装在移动臂上(每个盘面均有一个读写头),移动臂可沿盘半径方向移动。在磁盘叠中,每片盘的两面均可用来记录信息。但为了可靠起见,磁盘叠顶部的上盘面和底部的下盘面不作为记录信息用。有些系统常将某一盘面作为同步伺服面,供控制定位使用。由于每个磁道是闭合的,所以每条磁道的起点也是终点,并用一个检索符表示出来。检索符(或称检索点)是磁盘体的特殊点,所有的磁道均与这个点同步。各个磁道上并没有自己的检索点,它为各个磁道所公用。检索点的检测是由磁盘机中专门的读出装置读出的,不要求由读写头进行操作。

为了对磁盘叠中的一个物理记录进行定位,需要三个参数:

(1) 柱面号:随着臂的移动,各盘面所有的读写头同时移动,并定位在同样的垂直位置的磁道上,这些磁道形成了一个柱面。通常由外向里给各柱面依次编以顺序号,0, 1, 2, …, l。

(2) 磁头号:将一个磁盘迭的全部有效盘面(除去最外层的二面)从上至下依次编以顺序号 0, 1, 2, …, H。称为磁头号,因此盘面号与磁头号是相对应的。

(3) 扇区号:将各磁道分成若干个大小相等的扇区,并编以序号 1, 2, …, n。

现以 IBM 4300 系列的 3344 型磁盘为例,将其主要性能指标抄录如下:

有效盘片数　　8 片
有效盘面数　　15 面,一个盘面作为伺服盘面

图 9.5　移动头磁盘

盘直径	14in
读写头数	2 个/每个盘面
柱面数	696
每个磁道容量	8368B
最大容量	≈7MB
每转所用时间	20.2ms/r
最大查找时间	50ms
平均查找时间	25ms
数据传送速率	885KB/s

9.3　设备管理概述

9.3.1　设备绝对号、相对号、类型号与符号名

一个计算机系统中可以配置有多种类型的外设,而每种相同类型的设备又可以有好多台。为了使用和管理方便,必须对每台设备加以标识命名。各系统中命名方法虽有不同,但基本思想是相似的。通常是将系统中的每一台设备按某种原则进行编号,这些编号就作为硬件区分和识别设备的代号,称为此设备的绝对号(或称绝对地址)。

在早期的单道程序情况下,用户独占系统的全部资源,包括系统的全部外部设备。因此用户程序要使用设备时,可通过设备绝对号使用它们。但在多道程序环境下,正象用户不能在程序中使用绝对地址一样,他也不能在程序中通过设备绝对号来使用外部设备。因为用户不可能知道哪台设备正被其他用户使用着,哪台设备是空闲的。所以为了提高设备利用率,避免多道程序系统中用户使用设备时发生矛盾,把调度和管理外部设备的工作交给操作系统去完成。当用户要使用外部设备时,只需要向系统说明所要使用的设备类型即可。至于实际使用的是该类设备的哪一台,则由操作系统根据当时该类各台设备的使用情况进行调度。为此,操作系统为每一类外部设备规定了一个编号,称为外部设备的类型号。用户在向系统说明他所需要的设备类型时就可以使用此类型号。在 UNIX 系统中,类型

号被称为主设备号,在该系统中的所有块设备的设备名由两部分构成,即主设备号和次设备号。每个部分各占一个字节。主设备号表示设备类型,次设备号表示同类设备中的第几台设备。

往往一个用户程序可能同时使用几台同类外部设备,并且每一台都可能多次使用。为使用户程序能向操作系统说明当时它要使用的是哪一类设备的哪一台,则必须同时提出设备的类型标识和台标识。类型标识就是类型号,而台标识的方法类似于存储管理中的相对地址,是用户自己规定并使系统能了解他要使用的是这几台同类设备中的哪一台。通常台标识用一个数来表示,该数被称为设备相对号,以区别系统为每台设备规定的设备绝对号。

在语言中常常不用类型号,而用设备的符号名代替,如在 PDP11 中,PC 是代表光电输入机或卡片阅读器,卡片穿孔机,LP 代表行式打印机,KL/DL 代表交互式终端。一般把这种用来代表某类设备的符号称为设备符号名。用符号名后跟随一数字来表示相对号。

9.3.2 设备管理的任务

设备管理的基本任务是按照用户的要求来控制外部设备的工作,以完成用户所希望的输入输出操作。为此操作系统的设备管理要有以下功能:

(1) 记住所有设备,控制器和通道的状态。通常把设备管理中完成这部分功能的程序叫做 I/O 交通控制程序。

(2) 按用户要求启动具体设备进行数据传输操作,并且处理设备的中断。通常完成这部分功能的程序叫做设备管理程序。操作系统中每类设备都有自己单独的设备管理程序。如果一类设备中有多台设备时,它们共同使用同一个设备管理程序。

(3) 在现代多道程序系统中,系统中的进程数总是多于 I/O 设备数,因此必将引起进程对设备的竞争,所以设备管理的另一个重要任务,便是按一定的算法在诸进程间调度和分配设备。完成这部分功能的程序叫做 I/O 调度程序。

通常在设计设备管理的各部分功能时,应力求达到以下的目标:

(1) 方便性:编制输入输出程序是相当复杂的,一个简单的输入输出动作很可能要涉及到成百上千条指令,因此如何使用户从直接编制自己的输入输出程序的繁重劳动中解放出来,由操作系统来负责输入输出工作,使系统形成一种对“用户友好”的环境是十分重要的。目前许多系统都提供了标准的输入输出控制系统供用户使用。

(2) 设备独立性:这在操作系统设计中是经常被提到的一个要求。一般来说,设备独立性包含两个方面的内容。第一个内容,一个程序应该与给定设备类型中的哪一台设备供其使用无关。即一个程序中不应使用绝对设备号(或称绝对设备地址),而应使用相对设备号,这样才能使程序不致因某台设备损坏或已分给其他用户而无法执行。例如一个程序不应关心具体哪台行式打印机作为它的输出设备。这意味着要由操作系统负责把用户的相对设备号同具体的设备(绝对号)相对应起来。第二个内容,要求用户程序尽可能地与设备类型无关。这样我们仅对作业进行极少的修改,就可以改变输入输出设备,比如可以将从卡片阅读机上的输入改为从光电输入机上输入信息。这个要求意味着程序不要对具体的物理设备进行操作,而是从“虚拟设备”上进行操作(此处的“虚拟设备”在不同系统中有不

同的名称,如在 MULTICS 中和 COBOL 语言中将其称为文件,在 IBM 中将其称为数据集,在 Atlas 中称其为数据流)。这也就是说使用在程序中的应是数据集(以 IBM 为例)而不是物理设备。程序员编制程序时从(或对)数据集输入(或输出)数据。而由操作系统来建立数据集和物理设备的联系,操作系统通常是根据用户提供在作业说明书(或作业控制卡)中的信息来建立它们间的联系。例如在一个典型的作业描述语句中可以把数据集同外部设备用下述方法联系起来:

OUTPUT 1= LPT

此语句的含意是数据集 OUTPUT 1 就是行式打印机(这与 COBOL 语言中的环境部分的文件控制节所使用的方法完全相同)。程序与设备类型的无关性只要通过改变作业描述语句即可达到,如上例中改为

OUTPUT 1= PTP

就表示程序从纸带机上输出。在程序中由于只访问数据集 OUTPUT 1,而未涉及具体物理设备,所以不需作任何修改,从而体现了程序与设备类型的无关性。

(3) 并行性:为了提高设备利用率和系统效率,设备管理的设计应能使各设备的数据传输与 CPU 运行能高度重叠,使各设备充分地并行工作。

(4) 有效性与均衡性:由于输入输出操作往往成为计算机系统中的"瓶颈"部分,因此设备管理设计应尽可能地使设备有效地工作。除此之外,设备管理设计应使各设备充分地保持忙碌,而且要避免各设备忙闲不均的现象,这样才能最大地发挥设备的潜力。

(5) 字符编码的独立性:各外部设备的字符编码方式会有所不同,为减轻用户编程负担,应使用统一的"内部字符码"。这就要求设备管理中应有适应于各设备的字符编码的变换机构。

9.4 设备分配策略

系统中要求设备为其服务的进程总是多于设备,所以往往有多个进程竞争某个设备的使用权,也就是说有多个进程等待设备,于是就有一个设备的调度问题,或者说设备分配的策略问题。

设备分配策略往往同设备的性质密切相关,通常按照设备的使用性质可将设备分为:① 独享设备;② 共享设备;③ 虚拟设备。

所谓独享设备是指这类设备被分给作业后,为作业所独占使用,不能由几个作业同时共同使用。通常这类设备在运行时常需要操作员干预,如为卡片阅读机装卡片,为磁带机装、卸磁带,从打印机上取下打印结果,所以难以共享。多数的低速设备,如光电阅读机、行式打印机、穿孔机、绘图仪以及磁带机等均属此类设备。所谓共享设备是指允许多个用户同时共同使用的设备。例如磁盘,磁鼓等设备,可由多个进程同时进行访问。显然共享设备有着较好的设备利用率,但其管理可能比较复杂。所谓虚拟设备是指通过假脱机(SPOOL)技术把独享设备变为可由多个用户共享的设备,以提高独享设备的利用率,假脱机技术的基本思想是在一台共享设备(如磁盘、磁鼓)上模拟独享设备。显然设备类型不同,设备分配策略也不同。

9.4.1 设备控制块和设备等待队列

为了对设备进行有效地管理,需要对每台设备的情况进行登记。通常称这些表为设备控制块(简称 UCB),每台设备均有自己的设备控制块。由于各系统的管理和调度算法不同,该表的内容和格式略有差异,图 9.6 表示设备控制块(UCB),控制器控制块(CUCB)和通道控制块(CCB)的主要内容。其中“设备标识符”是指该设备的绝对号(或绝对地址)。“设备状态”通常除指示该设备忙、闲外,还可指示该设备是否良好或有故障。“等待此设备的进程表”指出等待此设备的进程队列中第一个进程的进程控制块(PCB)地址。而等待设备的进程表通常又称为设备分配等待队列。其组织可以按先来先服务(FIFO)原则排列,也可按优先数大小排列。可以用队列结构,也可以用表格结构(参看 3.3 节)。

图 9.6 控制块

9.4.2 独享设备的分配

独享设备包括宽行打印机、光电输入机、穿孔输出机、磁带机等设备。其分配策略,在简单情况下,可按照先来先服务的原则进行。在具有优先数作业调度的系统中,亦可考虑优先数的因素。但不管用什么算法分配设备,都应考虑死锁问题,因为设备分配往往是产生死锁的主要原因。尤其是独享设备,它本身的使用性质已是构成死锁的必要条件之一,所以在考虑独享设备的分配算法中,应结合考虑有关的防止死锁和避免死锁的算法。

在有硬件通道的计算机系统中进行设备分配时,应考虑整个数据传输通路的分配,即不但要分配设备,还要分配相应的控制器和通道。对于一些慢速设备来说,往往同字节多路通道相连,通道分时地为各设备服务。或者设备同非分配型子通道相连接。在这种情况下,分配设备后,无需再分配数据通路。对分配型子通道和其它通道类型,都有分配整个数据通路问题。尤其是设备、控制器和通道之间采用多重通路情况下,应查找与设备相连的控制器和通道是否空闲,并选择一个空闲的控制器和通道分配给请求的进程使用。

9.4.3 虚拟设备和 SPOOL 系统

系统中独占型设备的数量是有限的,往往不能满足诸多进程的需求,成为系统中的“瓶颈”资源,使许多进程由于等待某些独占设备成为可用而被阻塞。而另一方面,分得独占设备的进程,在其整个运行期间,往往占有这些设备,却并不是经常地使用这些设备,因而使这些设备的利用率很低。为克服这种缺点,人们常通过共享设备来模拟独占型设备的动作,使独占型设备成为共享设备,从而提高了设备利用率和系统的效率,这种技术被称

为虚拟设备技术,实现这一技术的硬件和软件系统被称为 SPOOL(Simultaneous Peripheral Operation On Line)系统(或 Spooling 系统),或称为假脱机系统。

SPOOL 系统通常又由输入 SPOOL 和输出 SPOOL 两部分组成,现在我们以输出 SPOOL 为例来研究 SPOOL 系统的工作原理。

假定某系统的全部行式打印机采用了虚拟设备技术(即使用了 SPOOL 技术),当某进程要求打印输出时,输出 SPOOL 并不是把某台打印机分配给该进程,而是在某共享设备(磁盘或磁鼓)上的输出 SPOOL 存储区中,为其分配一块存储空间,同时为该进程的输出数据建立一个文件(文件名可缺省)。该进程的输出数据实际上并未从打印机上输出,而只是以文件形式输出,并暂时存放在输出 SPOOL 存储区中。这个输出文件实际上相当于虚拟的行式打印机。各进程的输出都以文件形式暂时存放在输出 SPOOL 存储区中并形成了一个输出队列,由输出 SPOOL 控制打印机进程,依次将输出队列中的各进程的输出文件最后实际地打印输出。由此可以看出 SPOOL 系统的特点:

(1) 用户进程并未真正分得打印机,或者说打印机并未分给某个进程独占地使用;

(2) 用户进程实际被分给的不是打印设备,而是共享设备中的一个存储区(或文件),即虚拟设备,实际的打印机由 SPOOL 调度依次(按某一策略)逐个地打印这些输出 SPOOL 存储区中的数据。所以说 SPOOL 技术的中心思想是通过共享设备使独享设备变为可共享的虚拟设备。每个用户以为自己已独占了一台打印机,实际上系统中并没有那么多打印机,只不过是磁盘(或磁鼓)上的一个存储区而已;

(3) 独享设备使用效率提高了,从而系统效率也提高了。

该输出 SPOOL 的程序结构大致可描述如下:

```
begin
  repeat
if“输出队列为空”then WAIT
  “从输出队列取来一个文件”;
  “打开该文件”;
    begin
      repeat
      “从磁盘中读一文件行”;
      “等待磁盘完成中断”;
      “打印该文件输出行”;
      “等待打印机完成中断”;
      until(end of file)
    end
  forever
 end
```

当输出请求队列为空时,SPOOL 调用标准过程 WAIT 将自己阻塞在本进程的等待信号量上。直到有某个进程要求输出时,通过 SIGNAL 过程将其唤醒。打开文件是文件系统的要求(见第 10 章)。本程序中主要是输出 SPOOL 的打印发送部分——由输出队列中

逐个打印输出。通常还应有接收部分——将用户进程的输出放入输出队列,并为之建立输出文件。这部分可与文件系统结合,在此从略。

目前不但大、中型计算机的操作系统中使用 SPOOL 的输入输出工作,而且在许多微型计算机上也使用了 SPOOL 技术。如 Intel 8086 的 MP/M 和 M68020 微型机的操作系统采用了 SPOOL 输出技术。在 SPOOL 系统设计中,为了弥补独享设备(如打印机)与共享设备间数据传输速度的差异,需要使用缓冲区技术,所以应注意同步与互斥的问题。有兴趣的读者可参阅有关资料。

输入 SPOOL 的工作原理基本与输出 SPOOL 相同。

9.4.4 共享设备的分配和磁盘调度策略

当前的计算机系统中,用户处理的信息量愈来愈大,往往将其处理的各种数据以文件形式存放在磁盘或磁鼓等共享设备中,以便以后访问。所以在多道程序系统中,用户对盘、鼓等设备的访问是极其频繁的,因而对磁盘和磁鼓等设备的使用是否适当,直接影响着系统的效率。本节主要研究各种磁盘调度(分配)策略,对系统效能以及满足用户访问磁盘要求等方面的影响。磁鼓可看作固定头磁盘的特例。

9.4.4.1 移动头磁盘存储器的操作

磁盘上的一个物理记录块要用三个参数来定位:柱面号、磁头号、扇区号。那么如何读写盘上的一个物理块(或扇区),其访问时间又由哪些因素决定的呢?由于所有读写头的磁臂是一起沿磁盘半径方向移动,每个读写头不能单独移动。同时要访问的物理块只有处于读写头之下才能对该块进行读写。所以要访问某特定的物理块时,首先要按给出的柱面号将读写头随整个磁臂移到指定的柱面上。这个动作叫查找操作,查找操作所需的时间叫查找时间(图 9.7)。其次盘上该物理块必须随着整个盘旋转到读写头下面,这部分的旋转时间叫旋转迟延时间(简称迟延时间)。第三部分操作是读、写头对该物理块中数据实际的访问,其所用的时间称为数据传输时间。由于这三部分操作均涉及机械运动,故对某特定物理块的访问时间约为 0.01～0.1s 之间。其中查找时间所占的比例最大,通常约占整个访问时间的 70%。

图 9.7 磁盘访问时间的组成

在有些系统中,如在 2305-I 型固定头磁盘中,为了提高数据传输速率和减少平均取数时间,其磁盘驱动器可以同时访问 1 或 2 个存储块。因此一个逻辑数据记录必须记在磁盘的两个盘面上,通常记录的奇字节记在盘的顶面,而偶字节记在同一盘的底面,两面的磁道并行地进行读写。这样可以减少一半的传输时间。在有些磁盘系统中,为了减小旋转延迟时间,将同一记录同时存放在同一磁道的两个扇区之中,彼此相距 180°,从而减少了一半的延迟时间。但更使人们感兴趣

的是如何通过合理调度对磁盘的访问,以降低查找时间。

9.4.4.2　查找优化的各种策略

磁盘的调度策略有很多,通常评价调度策略的优劣主要考虑:

① 吞吐量;

② 平均响应时间;

③ 响应时间的可预期性(或变化幅度)。

当前使用比较普遍的一些查找优化策略如下:

(1) 先来先服务策略(FCFS):顾名思义,各进程对磁盘请求的等待队列按提出请求的时间进行排序,并按此次序给予服务。这个策略看来对各进程是公平的,它也不管进程优先级多高,只要是新来到的访问请求,就被排在队尾。

当用户提出的访问请求比较均匀地遍布整个盘面,而不具有某种集中倾向时(通常是这样的),FCFS 策略导致了随机访问模式,这种策略下无法对访问进行优化。在对盘的访问请求比较多的情况下,此策略将降低设备服务的吞吐量和提高响应时间,但各进程得到服务的响应时间的变化幅度较小。

FCFS 策略在访问请求不是很多的情况下,是一个可以接受的策略,而且算法比较简单。

(2) 最短查找时间优先的策略(SSTF, Shortest-Seek-Time-First),它是选择请求队列中柱面号最接近于磁头当前所在的柱面的访问要求,作为下一个服务对象。此策略可以得到比较好的吞吐量(比 FCFS),和较低的平均响应时间。其缺点是对用户的服务请求的响应机会不是均等的,对中间磁道的访问请求得到最好的服务,对内、外两侧磁道的服务随偏离中心磁道的距离而愈远愈差,因而导致响应时间的变化幅度很大,在服务请求很多的情况下,对内、外边缘磁道的请求将会无限期地被迟延。因而有些请求的响应时间将不可预期。

(3) 扫描策略:它是由 Denning 首先提出的,其目的是为了克服 SSTF 策略的缺点。对于 SSTF 策略来说,只要某访问请求所在的柱面离磁头当前位置最近,而不管该柱面是在磁臂的前进方向上,还是相反。而扫描策略是选择请求队列中,按磁臂前进方向最接近于磁头当前所在柱面的访问要求作为下一个服务对象。也就是说,如果磁臂目前向内移动,那么下一个服务对象,应该是在磁头当前位置以内的柱面上的诸访问请求中之最近者,……这样依次地进行服务,直到没有更内侧的服务请求,磁臂才改变移动方向,转而向外移动,并依次服务于此方向上的访问请求。如此由内向外,由外向内,反复地扫描访问请求,依次给以服务。此策略基本上克服了 SSTF 策略的服务集中于中间磁道和响应时间变化比较大的缺点。而具有 SSTF 策略的优点,即吞吐量比较大,平均响应时间较小,但是由于是摆动式的扫描方法,两侧磁道被访问的频率仍然低于中间磁道,只是不像上述 SSTF 策略那样严重而已。

(4) N 步扫描与循环扫描策略:作为对上述扫描策略的改进提出了 N 步扫描策略与循环扫描策略。N 步扫描策略基本上与上述扫描策略相同,只是当它在磁臂向内或向外移动过程中,只服务于在磁臂改变方向前到达的访问要求,而不理会在磁臂单向移动过程中到达的那些新的访问要求。

循环扫描策略与基本扫描策略的不同之处在于循环扫描是单向反复地扫描。当磁臂向内移动时,它对本次移动开始前到达的各访问要求,自外向内地依次给以服务,直到对最内柱面上的访问要求满足后,磁臂直接向外移动,使磁头停在所有新的访问要求的最外边的柱面上。然后再对本次移动前到达的各访问要求依次给以服务。

这两个策略具有基本扫描策略的优点,并且消除了其缺点。根据模拟研究表明,在访问负荷较小的情况下,基本扫描策略是最好的。在中等以上的负荷情况下,循环扫描策略则产生最好的结果。

9.4.4.3 旋转优化

为了减少旋转延迟时间,对同一柱面上各磁道的物理块的多个访问请求也需重新排队,进行旋转优化,通常使用的旋转优化策略是"最短延迟时间优化"的策略,图 9.8 所示为一磁鼓的旋转优化的例子,但也同样适用于磁盘的旋转优化。假定有四个访问同一柱面的请求,按到达的时间次序为(图中简化为对同一磁道的四个物理记录的访问):

图 9.8 磁鼓的旋转优化

① 读物理记录 4;
② 读物理记录 3;
③ 读物理记录 2;
④ 读物理记录 1。

如果不进行旋转优化,那么读这四个物理记录大约需要磁鼓转三周的时间。如果按最小延迟时间优先策略,则对这四个访问请求应重新排序,那么新的服务次序应是读记录 1, 2, 3, 4。这样读四个物理记录时间为半周的旋转时间。

在一些系统中,查找优化和旋转优化均可用硬技术来实现。

9.5 输入输出管理程序

目前各计算机系统中对输入输出的管理和控制的方法各不相同,我们在本章开始时曾提到 IO 调度程序、IO 交通控制程序和 IO 设备管理程序的输入输出管理系统的结构并

不是唯一的。本章中我们研究输入输出管理程序的主要问题及大致的工作情况。

9.5.1 输入输出进程

每个系统中都有很多不同种类的设备,每类设备往往有很多台,为了标识和记录设备的状态以便对设备进行管理和调度。每一台设备、每一个控制器和每一条通道都有一个设备控制块(控制器控制块、通道控制块)简称 UCB(CCB, CHCB),其中记录着设备的标识符(或绝对号),设备状态、分配等待队列头指针和设备相连的控制器与通道。为了管理和控制设备的操作,完成用户提出的输入输出请求,每一类设备都有一个设备管理程序。在操作系统中,用户进程如何同这些设备管理程序发生作用呢,这通常也有不同的方法。

在 UNIX 系统中,把设备作为一种特殊文件来处理,因此对设备的访问变成为对文件的访问。所以对输入输出的管理是通过五个文件操作的功能调用来实现的,这五个操作是:打开一个文件、关闭一个文件、读文件、写文件和查找文件。

更普遍的方法,则是为每一台设备建立一个设备管理进程,以具体负责和管理该台设备的输入输出工作。这些进程运行的程序是各自的设备管理程序(同类的各台设备所对应的各设备管理进程,共享该类设备的设备管理程序)。这样用户进程要求访问(使用)某设备时,可以通过进程间通信的手段来实现。如进程 P 要求从磁带机上输入一批数据时,它可调用系统的"发消息原语"给磁带机进程发一个消息,在消息中可以提出其所需的操作,数据存放的主存地址,输入的数据量等。磁带机进程接到消息后,执行磁带机管理程序,进行相应的处理。

9.5.2 设备管理程序

设备管理进程运行的程序就是各类设备相应的设备管理程序,也就是说设备管理进程严格执行设备管理程序中规定的各种功能。一般来说,设备管理程序应有以下功能:

(1) 按受用户的输入输出请求,或者说接受用户(或系统)进程的消息。首先它从用户的输入输出请求(消息)中,根据该类设备的特性,汇集有用的参数形成"输入输出请求块"(又称 IORB)。然后将此请求块排在请求队列的队尾。该请求队列的首地址通常由与此设备有关的信号量中指出,而此信号量也可以是设备管理进程的 PCB 中的一个表目。通常在形成 IORB 过程中,还要检查输入输出请求的合法性(如参数是否合法)。

(2) 将请求队列中的第一个输入输出请求块拿来,并按照此 IORB 中的参数要求为之形成通道程序。

(3) 启动该设备工作(或执行通道程序),但这有两种情况:当与设备相连的是字节多路通道或非分配型子通道,则根据设备控制块状态是否忙来决定是否启动设备工作;当与设备相连的是分配型子通道或多重通路情况下,还要调用输入输出调度程序以分配相应的控制器和通道。

(4) 处理来自设备的中断。通常来自设备的中断包括数据传输完成的结束中断,传输错误中断,和设备故障中断。结束中断的处理是把设备,控制器和通道的控制块(UCB, CCB, CHCB)中状态置成"空闲"。然后看请求队列是否为空,如果为空,则设备管理进程封锁自己以等等用户的输入输出请求(消息)。如果队列非空,则处理下一个 IORB。如果

是传输错误中断,则其处理办法是向系统报告错误或进行重复执行处理。故障中断的处理是向系统报告故障。

因此设备管理程序可以大致描述如下:

```
repeat
        wait message;
     If message acceptable then
        begin
          start input(output);
          wait interrupt
        end
        produce answer message;
        send answer message;
  forever
```

9.5.3 输入输出调度程序

输入输出调度程序的主要功能是为用户的数据传输请求分配通道、控制器和设备,或者是设备上的存储空间(用户要使用独享设备时,则请求占有此设备。用户要使用虚拟设备和共享设备时,则请求分给共享设备上的存储空间)。但由于传输请求往往多于可用的设备,更多于可用的传输通路,因此必须作出选择先满足那个请求。也就是说要按照一定的调度策略来决定先满足那个用户(进程)的请求。在考虑调度策略时,对于独享设备还应考虑预防和避免死锁的问题。对于共享设备(包括虚拟设备)来说,则要考虑前几节中研究过的查找优化策略和旋转优先策略问题。

输入输出调度程序除了要考虑先满足谁的调度策略外,另一个更重要的任务是为用户请求分配设备和传输通路(相应的控制器和通道)。这样输入输出调度程序必须查找设备控制块表(所有同类设备的各 UCB 的集合组成该类设备表),以确定一个可用的设备。调度程序还要进一步查找该设备的 UCB,以确定该设备与哪些控制器相连,而每个控制器又与哪些通道相连(在多重通路情况下)。以便为该设备分配可用的控制器和通道,否则必须等待。

9.6 Windows NT 一体化的输入输出系统

Windows NT 的输入输出(I/O)系统是在所有操作系统中最具特色的。这是由于:

(1) 在高度抽象基础上建立起了统一一致的高层界面——I/O 设备的虚拟界面,即把所有读写数据看成直接送往虚拟文件的字节流。从而把通常操作系统中完全不同的四部分——文件系统、高速缓冲存储器、设备管理、网络管理——组织成为一个界面和操作一致的输入输出(I/O)系统。

(2) I/O 系统是个层次结构。构成 I/O 系统的部件有:I/O 管理程序、文件系统、缓冲存储管理器、设备驱动器、网络管理器。

(3) 统一的驱动程序模型。所有的驱动程序是统一的结构;用同一方式建立;表现出相同的外貌。

(4) 具有异步 I/O 操作。

(5) 具有映象文件 I/O 功能。

详细情况请参阅 12.8.1～12.8.4 节。

习　　题

9-1　什么叫通道和通道程序? 通道程序由谁来执行? 存放在什么地方?

9-2　什么叫窃用周期?

9-3　磁带上数据块之间的间隙是作什么用的,为什么要把磁带上的信息组成块?

9-4　磁盘和磁带上的信息均是如何定位的?

9-5　设备按其使用性质分成几类? 并举例说明。

9-6　什么叫 SPOOL 系统,它是如何工作的?

9-7　SPOOL 系统由输入 SPOOL 和输出 SPOOL 两部分组成,为什么现在的微型机上大多只提供输出 SPOOL 的功能?

9-8　对磁盘的访问时间由哪三部分组成? 哪部分时间最大?

9-9　如何减少“柱面定位时间”?

9-10　比较查找优化的四种算法的特点及优劣。

9-11　什么叫虚拟设备?实现虚拟设备的主要条件是什么?采用虚拟设备技术有何优点?

第 10 章　文 件 系 统

10.1　文件系统概述

10.1.1　引言

据联合国工业发展组织的调查材料表明,当前世界上的计算机,百分之八十以上用于数据处理。目前数据处理的范围已遍及人们生活的各个方面,处理的信息量也愈来愈多了,这些信息显然不能全部存放在主存中,所以人们很早就引进了外部存储器,用以保存大量的永久性的或暂时性的信息。为了便于对这些信息进行管理,所有在外部存储器中的信息均以文件的形式存放在其中。在较大的计算机系统中,有多达数万个文件。目前各个计算机系统都十分重视文件管理的功能,即使在小型甚至个人计算机中,它们的操作系统的其它功能往往不见得很强,但相对来说都具有较强的文件管理功能。UNIX 系统也以其文件系统使用方便而著称。可见文件系统在操作系统设计中的重要地位。

在早期,由于没有功能足够强的文件系统对外部存储器中的文件进行管理,所以对文件的使用是相当复杂和繁琐的工作。无论是用户还是系统本身,均需熟悉外存的物理特性。他们不但要按其物理地址(如磁带上的数据记录号,磁盘的柱面号,磁头号、物理记录块号等)存取信息,而且要准确地记住存在外存中的信息的物理位置和整个外存的信息分布情况,组织相应的输入和输出指令等。稍不小心,即会破坏已存入的内容。很显然,要求用户记住这样大量而复杂的信息分布情况,对用户是一种沉重的负担,对系统也带来了不安全因素。尤其是在多道程序出现以后,许多用户共享大容量的外部存储器的情况下,文件的安全和保密等问题更显得突出,这时用户还想自己去协调、管理那些信息既不可能也不允许,如同主存和外部设备一样,外部存储器也必须由系统来统一加以管理,这就是为什么现代的计算机系统的操作系统中都配备了文件系统(或称信息管理系统)。

什么叫文件和文件系统呢? 所谓文件是指具有符号名的数据项的集合。符号名是用户用以标识文件的。作为一个文件的例子很多,例如一个命名的源程序、目标程序、数据集合等均可作为一个文件,又如各种应用信息,如职工的工资表、人事档案表、设备表以及文件目录,系统程序和过程等,给以命名后也均可作为文件。

对于文件的基本构成单位目前有两种看法:

(1) 把文件看作是命名了的字符串的集合: 在 UNIX 系统中, 文件系统从物理上(而不是从用户对文件的逻辑上的看法)将每个文件仅仅看成是由一系列字符串组成, 而不把文件处理成物理记录的集合。

(2)把文件看作是命名了的相关记录的集合: 这是一种比较普遍的做法, 即使在 UNIX 系统中, 用户也往往把他的文件看成是相关记录的集合。例如一个命名为'学生登记表'的文件是每个学生情况的记录的集合。而记录是相关的数据项的集合, 数据项是相关的字符的集合。例如每个学生情况的记录是由姓名、性别、年龄等数据项组成, 而姓名、年龄、性别等数据项则由若干个字符组成。

所谓文件系统是指一个负责存取和管理辅助存储器上文件信息的机构。文件系统既要负责对用户的私人专用存储器上信息的访问, 也要负责提供给用户以有控制的方式访问共享的信息。后者在现今的计算机系统中已显得愈来愈重要了。

文件可保存在各类存储介质上, 诸如软磁盘、硬磁盘、磁鼓、磁带和主存等存储器中, 也可保存在卡片、纸带、打印纸等媒体上。但本章中研究的文件系统功能仅限于对辅助存储器上所存信息的管理。

10.1.2 文件的分类

为管理和控制文件方便起见, 常将系统中的文件分成若干类型, 根据各文件系统管理方法不同, 文件分类方法也不同。许多系统中还常把文件类型与文件名一起作为识别和查找文件的参数。通常对文件有以下几种分类方法。

1. 按用途分类

(1) 系统文件: 指与操作系统本身有关的一些信息(程序或数据)所组成的文件。

(2) 库文件: 是指由系统提供给用户调用的各种标准过程、函数和应用程序等。

(3) 用户文件: 由用户的信息(程序或数据)所组成的文件。

2. 按文件中数据分类

(1) 源文件: 是指从终端或输入设备输入的源程序和数据, 以及作为处理结果的输出数据的文件。

(2) 相对地址形式文件: 是指由各种语言编译程序所输出的相对地址形式的程序文件。

(3) 可执行的目标文件: 是指由连接装配程序连接后所生成的可执行的目标程序文件(用在非动态连接系统中)。

3. 按操作保护分类

(1) 只读文件: 仅允许对其进行读操作的文件。

(2) 读写文件: 有控制地允许不同用户对其进行读或写操作的文件。

(3) 不保护文件: 没有任何访问限制。

4. 按保存时间分类

(1) 临时文件: 批处理中从作业开始运行到作业结束, 或是在分时处理中从会话开始到终了期间所保存的暂时文件, 一旦这些作业终了, 其相应的临时文件也被系统自动撤消。

(2) 永久文件: 是指用户没有发出撤消该文件的命令前,一直需在系统中保存的文件。

5. 在 UNIX 系统中,文件的分类

(1) 普通文件: 一般的正文文件。

(2) 目录文件: 把文件目录本身也看成为文件。

(3) 特殊文件; 把每个输入输出设备看成是一个特殊文件。这样的做法可带来以下优点:

① 文件和设备的输入输出便于尽量统一;

② 文件名和设备名有相同的文法和意义;

③ 文件和设备服从同一保护机制。

10.1.3 文件系统的功能和基本操作

1. 文件系统的功能

一个文件系统应具有以下功能:

(1) 使用户能建立,修改和删除一个文件;

(2) 使用户能在系统控制下共享其他用户的文件,以便于用户彼此利用其他人的工作成果;

(3) 用户应能以方便其使用的方式来构造他的文件;

(4) 用户应能使用在文件间进行数据传输的命令;

(5) 用户应能用符号名对文件进行访问,而不应要求用户还得使用设备名来访问文件(与设备独立性要求一致);

(6) 为防止意外事故,文件系统应有转储和恢复文件的能力;

(7) 应提供可靠的保护和保密措施。

2. 文件操作功能

文件系统应方便于用户的使用。不应要求用户必须了解文件的物理组织(如存放的物理地址,在物理介质上怎样存放等)才能使用文件,应提供给用户按其逻辑组织形式来使用文件的便利,因为文件的逻辑组织形式,是在用户编制文件过程中所熟悉的。

一个文件系统至少要提供给用户以下的文件操作功能:

对整体文件而言:

(1) 打开(open)文件: 以准备对该文件进行访问;

(2) 关闭(close)文件: 结束对该文件的使用;

(3) 建立(create)文件: 构造一个新文件;

(4) 撤消(destroy)文件: 删去一个文件;

(5) 复制(copy)文件;

(6) 对文件改变其文件名(rename);

(7) 打印或显示文件内容(list)。

对文件中的数据项而言:

(1) 读(read)操作: 把文件中一个数据项输入给进程;

(2) 写(write)操作：进程输出一个数据项到文件中去；
(3) 修改(update)操作：修改一个已经存在的数据项；
(4) 插入(Insert)操作：添加一个新数据项；
(5) 删除(delete)操作：从文件中移走一个数据项。

10.2 文件的逻辑组织和物理组织

用户和系统程序人员常从不同的角度来看待同一个文件。用户对文件的观察和使用常以其编制时的组织方式来看待文件组织形式，我们常称此为文件的逻辑组织。而系统程序人员常按文件具体在辅助存储器中是如何存取、如何组织来看待文件的组织形式，我们称此为文件的物理组织。

10.2.1 文件的逻辑组织

文件的逻辑组织可分为两种形式。一种是记录式文件，即文件是由若干个相关的记录组成，并且对每个记录编以序号，分别称为记录 1、记录 2、……、记录 n，这种记录称为逻辑记录，其序号称为逻辑记录号。记录式文件按其各记录的长度是否相同又可分为等长记录文件和变长记录文件两种。前者文件的各记录长度均相等，后者文件的各记录的长度可不相等。与记录式文件相对应的另一种逻辑组织形式是非记录式(或称顺序式)文件，即文件是有序的相关数据项的集合。这种文件不再划分成记录。通常像“职工登记表”、“学生登记表”之类的文件，其逻辑组织形式为记录式文件，而源程序、目标程序之类的文件，其逻辑组织形式为顺序式或非记录式文件。

10.2.2 文件的物理组织

文件的物理组织是指一个逻辑文件(即用户文件)在辅助存储器上是如何存放的。我们知道在磁盘、磁鼓和磁带等辅助存储器上，是以物理块或物理记录来划分的，并且是以物理块或物理记录作为单位来分配辅存空间给用户的文件。因此逻辑文件的信息自然是保存在辅存的物理块和物理记录中。但由于各文件的逻辑记录长度是不同的，而物理块和物理记录的大小随设备不同而不同，因此逻辑记录大小与物理记录(或物理块)大小之间就不会有固定的对应关系，而逻辑记录和物理记录之间更不会有类似分页存储中页面与页架之间的一一对应关系。有时一个物理记录(或物理块)可存放几个逻辑记录，而有的情况下一个逻辑记录要用几个物理块才能放得下。除此之外，在像磁盘、磁鼓之类的共享存储器中，也会出现类似于主存储器中那样的碎片，使得一个文件的各记录无法存放在相邻的物理块中(如果不采取紧缩的办法)。正因为如此，所以文件的物理组织形式可以有很多种，常见有三种。

1. 顺序结构

一个逻辑文件的信息依次存于辅存的若干连续的物理块(或物理记录)中，则此文件的物理组织形式被称为是顺序结构的，这种物理组织形式通常用于磁带，纸带，穿孔卡片和打印输出等存储介质上，这些存储介质的性质就属于顺序存取设备。磁盘上的文件也可

以按顺序结构来组织。

一个定长记录的逻辑文件的顺序结构方式如图 10.1 所示，在存储介质上的物理位置是“下一个”逻辑记录跟随在前一个逻辑记录之后。而对于一个变长记录的逻辑文件的顺序结构如图 10.2 所示，它在存储媒体中存放按逻辑记录号排序，这同定长记录文件是一样的。但由于各记录长度不同，需要在每个记录前用一个单元来指示本记录的长度，以便于用户按记录长度对文件进行访问。

记录 1	记录 2	记录 3	记录 4	
←1→	→1→	→1→	←1→	

图 10.1　顺序结构的定长记录文件

l_1	记录 1	l_2	记录 2	l_3	记录 3	l_4	记录 4
	←l_1→		←l_2→		←l_3→		←l_4→

图 10.2　顺序结构的变长记录文件

对于这种顺序结构文件，用户应给出文件的最大长度，以便在用户建立文件时为其分配足够的外存空间。对顺序结构文件的操作主要是读、写操作，不允许对文件中间的某部分进行插入和删除操作，一般只允许在文件末端进行插入和删除。顺序结构文件只适用于只读的输入文件和只写的输出文件。

2. 链接(或串联)结构

对于文件的应用方式，如果用户基本上是顺序地访问文件中的信息，那么上述顺序结构的物理组织形式是适合的。但这种组织形式，如果在盘和鼓上实现，往往受到连续的空闲存储空间大小的限制，往往需要采用紧缩技术以保证足够的连续空闲空间，这增加了磁盘管理的复杂性。对于这种应用方式，即主要是顺序地访问文件中的信息，如果该文件要存放在磁盘和磁鼓中，那么其在盘、鼓上的物理组织形式可采用链接(或串联)结构。在这种物理组织中，一个文件不需要存放在存储媒体的连续的物理块中，类似主存中页架分配那样，一个文件可以散布在辅存的不连续的若干个物理块中。但每个物理块的最后一个单元中不能存放文件的信息，而要用来作为物理块之间的链接指针，它指向下一个物理块的位置(物理块的绝对块号)。指针链接的前后两物理块中的信息在逻辑组织上是连续的。文件的最后一个物理块的链接指针通常为“0”，表示该块为链尾。图 10.3 表示了此种结构形式。

链接结构消除了顺序结构方式中必须分给文件若干连续物理块的缺点。但是它也有其固有的缺点，首先，文件的各记录可能分散在整个盘上，这样即使是顺序地访问各记录的应用方式，也使得磁盘查找时间，在整个访问时间中所占比例较大。磁盘查找时间也比顺序结构方式要高得多。如果这种物理组织使用于非顺序地随机访问记录的应用方式下，为了查找某个记录在盘上的物理块地址，也必须从文件的第一块开始，沿整个链进行依次追寻，这样，查找时间在访问时间中所占比例更高(如要找最后一个记录，就必须查寻整个链)。因此链接结构形式是不适用于随机访问的应用方式的。其次是每个物理块都需要有

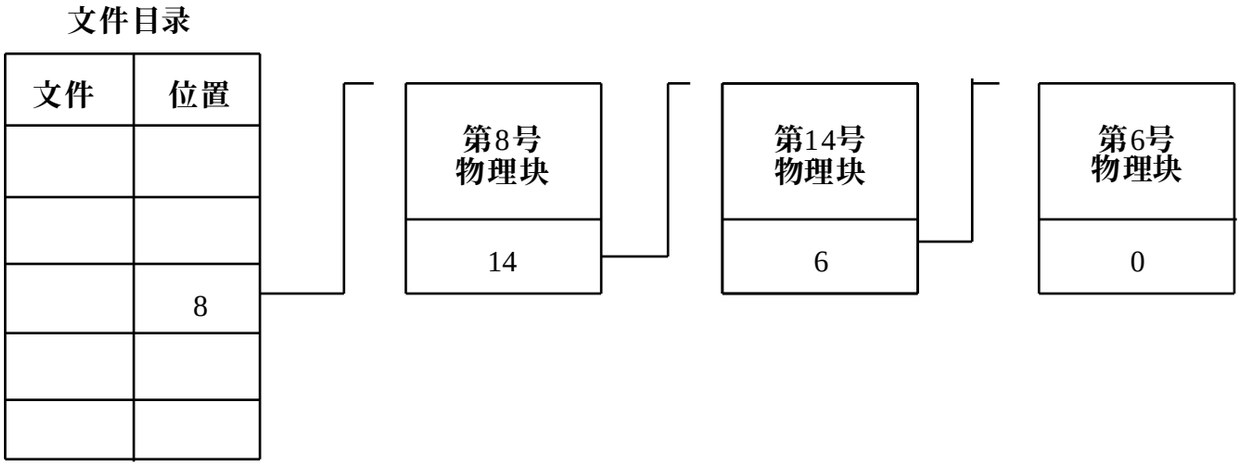


图 10.3　链接结构文件示意图

一个链接指针,这降低了磁盘空间的利用率。

作为对以物理块(扇区)为单位的链接结构的一种改进,是被称为以“区段”(或称为簇)为单位进行分配的链接结构形式。所谓“区段”是若干个连续的物理块(扇区)组成。系统为文件分配存储空间时,以区段为单位进行分配,并且寻找尽可能接近(按区段号,尽可能在同一柱面或相接近的柱面上)的区段分配给文件。同文件的各区段之间以上述方法进行链接。这样就提高了辅存的管理效率,降低了查找时间的开销。以区段为单位进行分配的方法也可使用于下面的索引结构中。

3. 索引结构(索引顺序结构)

上述两种结构形式比较适用于顺序访问的应用场合。但用户常常希望能随机地访问文件中的某一个记录或某些记录。比如说要求从“学生登记表”文件中,查找学生张三的各科成绩。对于这种使用方式的文件,其物理结构以采用索引结构(或称索引顺序结构)方式为好。所谓索引结构是系统为每个文件建立一张索引表如图 10.4 所示,索引表中包含两个主要内容:关键字和位置。关键字是包含在每个记录中的某个数据项,如“学生登记表”中的学生姓名,即可作为关键字。关键字要选择那些用户在检索记录时用作索引的主要数据项。通常在索引表中的关键字采用所对应的物理块中的最大关键字值,索引表按关键字的递增序列排列。位置是指该记录所在的物理块号。在磁盘空间中可不要求它们连续。对这种形式组织的文件,既可以按索引顺序进行顺序地访问,也可以按关键字进行直接地(随机地)访问某个记录。索引结构文件通常存在磁盘中。

图 10.4　索引结构

一个文件的记录个数有时可达上万个以上,如工厂的职工登记表所形成的文件,在该

文件中,每个职工的情况就是一个记录,其所构成的索引表可能会很长,以至于需要若干个物理块来存放。在这种情况下,可把文件索引表本身作为一个文件,并可采用链接结构方式存放在磁盘上。毫无疑问,由于索引表太大,对索引表本身的检索就会引起较大的开销。

为了克服检索大索引表所需开销过大和速度比较慢的缺点,还可以采取多级索引表的办法。图 10.5 表示了这种结构组织,主索引表指出各次级索引表的位置,次级索引表指出该文件按关键字顺序排列的包含有若干个记录的各个子集。

图 10.5　多级索引结构文件的组织

索引表及其所指出的文件的各数据子集的后面都留有若干个空白表目,以便插入新的记录。最后的一个空白表目作为溢出指针,当要插入更多记录以至于空白表目不敷使用时,需要把一部分记录放到溢出区中,而溢出指针指出其在溢出区的首地址。

对于这种结构的文件,用户每次请求访问一个记录需要三次访问外存,一次访问主索引表,一次访问次级索引表,而后顺序查找数据子集中的各记录的关键字,以找出要访问的记录。

10.3　文 件 目 录

10.3.1　文件目录和文件描述符

一个计算机系统中有成千上万个文件,为了便于对文件进行存取和管理,每个计算机系统都有一个目录,用以标识和找出用户与系统进程可以存取的全部文件。文件目录在文件系统中的作用类似于一本书的章节目录一样,只是文件目录的作用比书的章节目录更

强。

文件目录中应存放每个文件的有关信息,也就是说每个文件在文件目录中都应该有一个表目。由于目录本身是被查找和修改的对象,因此差不多所有的文件系统把目录也作为一个文件来处理。当然它是一个有特殊作用的文件,但有关文件系统的所有操作也同样能够适用目录文件。目录文件本身在目录中也要有个表目来描述它自己的有关信息。

文件目录中应包含文件的哪些信息呢? 最简单的文件目录表目起码应该包含文件的符号名(外部名)或内部描述符(内部标识号)和它的物理地址以建立起文件名和地址的对应关系。而较复杂的文件目录表目将存放每个文件的完整的文件描述符(或称文件控制块)。在这种情况下,文件目录就成为文件描述符的集合,一个文件描述符(或者说文件目录的表目)通常应包含以下内容:

① 文件的符号名;

② 文件的内部名;

③ 设备类型和设备号(设备地址);

④ 文件在辅存的起始地址(物理块号);

⑤ 文件在辅存的物理组织形式(顺序、链接、索引等);

⑥ 文件类型;

⑦ 文件大小以及逻辑组织形式的有关信息,如记录大小,记录个数等;

⑧ 文件共享的存取控制说明;

⑨ 用户名、帐号、同组用户名等;

⑩ 保存期限;

⑪ 文件建立的日期和时间;

⑫ 上次对文件进行修改的日期和时间。

由于文件系统中通常有很多文件,因此文件目录也就很大,所以一般不把文件目录放在主存中,而是放在辅存中,并且将文件目录及其所属文件存放在同一存储媒体或同一存储设备中。

10.3.2 一级目录结构

管理文件目录的最简单办法是采用一级目录结构。所谓一级目录结构是指把系统中的所有文件都建立在一张目录表中,整个目录组织是个线性表,结构比较简单。每当要建立一个新文件时,就在目录表中增加一个新的表目。撤消一个文件时,就在该目录中将此文件的表目中的信息清除即可。

但是一级目录结构有以下缺点:

(1) 由于系统中有成千上万个文件,每个文件一个表目,这样文件目录表将很大。如果要从目录表中查找一个文件,就要扫描整个目录表,使得查找目录时间增加。

(2) 不方便用户对文件的访问,为什么呢? 因为用户使用文件最方便的办法是通过文件的符号名(用户给文件命名的外部名)来访问,用户难以记住每个文件的唯一的内部标识名(标识号)。那么当两个以上的用户给他们的文件起了相同的外部符号名时,用户一旦要访问这种文件,系统就无从分辨他究竟要使用的是同名文件中的哪一个文件。因此一级

目录结构的文件系统必须规定用户不能给文件起相同的名字。如果要求同一用户不要给他的文件起相同的名字是可以办到的。但如果要求不同用户之间的文件也不能同名,这就难以办到了。所以说一级目录结构不能提供给用户这样的便利——可以给不同文件起相同的名字。再者文件系统应能使用户共享某些文件,有时由于工作中应用方便起见,不同用户给同一个共享文件起了不同的名字,这在一级目录结构的文件系统中,也是不可能的。总而言之,一级目录结构解决不了多用户环境下的命名冲突问题,所以一级目录结构主要用在单用户的操作系统中。

10.3.3 二级目录结构

这是一种最普通的、广泛使用的目录组织形式,它是一个如图 10.6 所示的树状结构。二级目录结构是一个主目录文件和它所管辖的若干个子目录组成。每个结点(第二级目录)在根结点(主目录)中都有一个表目来指出二级目录文件的位置,该树状结构的叶就是具体的数据文件。

图 10.6 二级文件目录

在多用户情况下,采用二级目录结构较为方便。在主目录中,每个用户都有一个表目,指出各次级目录——各用户自己的文件目录所在的位置。而各用户的目录(次级目录)才指出其所属的各具体的文件的位置和其它属性。

当一个新用户要建立一个文件时,系统为其在主目录表中分配一个表目,并为其分配一个存放二级目录的存储空间,同时要为新建立的文件在二级目录中分配一个表目,分配文件存储空间。当用户要访问一个文件时,先按用户名在主目录中找到该用户的二级目录,然后在二级目录中按文件名找出该文件的起始地址并进行访问。

10.3.4 多级目录结构

在较大的文件系统中,往往采用多级目录结构,这可以给大作业的用户带来更多的方便,它可以使每个用户按其任务的不同层次、不同领域建立多层次的分目录。多级目录结构组织形式同现实生活中的许多事物的组织形式一致。如学校由校、系、年级、班级四层次组成;一个行政机关由部、局、处、科等层次组成。这样,一个学校可以按其层次组织方式,采用多级目录结构形式,方便地管理其所属的全部文件。多级目录结构的一般形式如图

10.7 所示。由图上可以看到,从这个树形结构的每一个结点(目录)出来的分支既可以是叶(数据文件),也可以是又一个节点(次级目录)。图中圆代表数据文件,矩形代表目录文件,文件旁注的数字是系统分配给文件的唯一的内部标识符,目录中字母表示文件的符号名。例如用户 C 的目录文件其内部标识号为 4,外部标识符为 C。它有三个数据文件,它们的内部标识号分别为 8,9,10。外部标识符分别为 G,A,E。

文件系统采用二级或多级目录结构后,就能够为用户提供文件可以同名和一个文件可以有多个不同名字的这种便利。在这种情况下,当用户进程要访问多级目录中某个文件时,往往用该文件的"路径名"来标识文件。所谓文件的路径名是指从主目录(目录树形结构中的根)出发,一直到所要找的文件,把途经的各分支名(结点名)连接一起而形成的(此处暗含假定任何二个结点之间只有一条分支,即一条路径相连,否则不能保证该路径的唯一性),两个分支名(结点名)之间用分隔符分开。

在 UNIX 系统中用"/"作分隔符。例如/A/B/H 表示访问的是内部标识号为 15 的文件 H(图 10.7),路径名前面的第一个"/"代表根目录(主目录)。所以该例子表示从根目录中找到目录 A,再从目录 A 中找到目录 B,而后从目录 B 中找到 H。

图 10.7　多级目录组织

由于用文件路径名来寻找文件、标识文件,所以只要在同一结点(目录文件)中各文件表目的文件名不相同,那么路径名就能唯一的确定一个被查寻文件,尽管系统中可以有很多同符号名文件,也不会得到二义性的结果。

多级目录结构使得对文件的管理和使用都十分方便、灵活。尤其在 UNIX 系统中,它

提供了一个系统调用“mount”,使得用户能够将他自己的文件、文件目录连同其存储设备一起连接到用户当前正在访问的目录——当前目录上去。不用时,用户又可很方便地将私用的存储媒体、文件目录和文件折卸下来。这样整个文件系统就可以根据需要而随时安装和拆卸一部分文件。当然这必须是同一存储媒体上的所有文件及其目录树(目录的层次结构)是一个自包含集,即都在该存储设备上。一个给定目录的成员的一个任意子集不能存放在另一台设备上。

但是多级目录结构,沿着路径查找文件是可能会耗费查找时间的,一次访问可能要经过若干次间接查寻才能找到所要文件。读者尤其要明确的是,这种查寻是对辅存进行的。因为文件目录太大,不能全部装入主存中来查。因此往往要沿路径依次把各所需目录所在的物理块读入内存来查找。这显然很费时间,又增加了通道的压力。为此,各系统往往引进了“当前目录”或“值班目录”来克服此缺点。即由用户在一定时间内,指定某一级的一个目录作为“当前目录”,而后用户想要访问某个文件时,便不用给出文件的整个路径名,也不用从根目录开始查寻,而只需给出从当前目录到要查寻的文件间的路径名即可,从而减少查寻路径。UNIX 系统中,如果给出的路径名不是以“/”开头,就表示从当前目录开始查寻,如图 10.7 中用户指定文件 B(内部号 6)为当前目录,则路径名 H/L 规定了访问文件 24。目前在 IBM 4300 系列,MULTICS,以及 UNIX 中均是多级目录结构。在 ATLAS(英国剑桥大学)等计算机中采用二级目录结构。

10.3.5 目录组织的改进——符号文件目录和基本文件目录

前面讲到的目录组织,不管是一级目录组织和多级目录组织,基本上是文件描述符(见 10.3.1 节)的集合。文件描述符中,包括了许多关于每个文件的信息:如各文件的符号名,内部名,文件逻辑和物理组织形式,存取控制信息,用户的各种信息等等。通常一个文件描述符就要占很多空间,这样一个存放目录的盘物理块中放不了几个目录。如 UNIX 系统中,物理块大小为 512 字节时,其文件描述符占 48 个字节(索引节点大小加文件名),这样只能放 10 个目录表目。而文件系统在为用户查寻文件时,要把存放目录的物理块逐块读入内存中进行查寻。由于每个物理块中只包含很少几个目录,这样查寻到的概率比较低。为了找到一个文件在目录中的表目,就要多次读进目录所在的物理块进行查寻,这既降低查寻速度,又大大增加了输入、输出通道的压力。由于考虑到系统查寻目录时只使用文件符号名进行查找,而与文件描述符中其他信息无关,所以我们考虑是否可把文件符号名与文件描述符中其他信息分离开来使之成为两个部分:一部分称为符号文件目录(UNIX 中称之为目录),它包含文件符号名与文件内部标识号(或内部名),其中内部标识号在系统中应是唯一的,即不能有两个文件具有相同的内部标识号。另一部分称为基本文件目录(类似于 UNIX 中的索引节点表),它包含文件描述符中所有其它信息,并且各文件在基本文件目录中的表目是按文件的内部标识号由小到大进行排序的。这样就把原来的目录层次结构分为两套目录结构:基本文件目录和符号文件目录。

这样做的优点是明显的。首先查寻目录时命中率高,因为符号文件目录只包含很少的信息。每个物理块可包含更多目录表目,从而提高查寻效率。其次便于文件的共享连结(见下节)。

通常基本文件目录结构是一个一维的线性总表。而符号文件目录结构是一个多级的层次结构。如同前节所述，有个主符号目录，各用户的次级符号文件目录，次级符号文件目录又可以有他的子目录层，……如图 10.8 所示。当用户要访问某个文件时，仍然使用路径名从主符号目录(或当前目录)出发，在符号文件目录树中找出该文件的表目，从而得到该文件的唯一的内部标识号。并以此内部标识号为索引，查找基本文件目录表，从而得到该文件描述符的内容以供文件系统使用。

图 10.8　改进的文件目录组织

图 10.8 表示出了改进的文件目录组织。由于文件目录本身也作为文件处理，因此每个符号文件目录均在基本文件目录中有其相应的表目，而且有其自己的文件内部标识号，如主符号文件目录的内部标识号是 2。毫无例外，基本文件目录本身也是个文件，因此它自己在基本文件目录中也要有个表目。通常 ID= 1 的第一个基本文件目录的表目属于基本文件目录本身，其内部标识号为 1，而第二个表目是属于主符号文件目录，内部标识号为 2。

把图 10.8 中的符号文件目录画成一个目录树的话，则如图 10.9 所示。

文件目录分成基本文件目录和符号文件目录两个系统后，不同用户可以给同一个文件起不同的名字，只要这些文件符号名都对应同一个内部标识号即可，如图 10.8，图 10.9 中所示，在用户 Tu-Lide 的符号目录中称为 Rooms 的文件和用户 Tu-Qi 的符号目录中称

图 10.9　符号文件目录的层次结构

为 Classroom 的文件实际上是同一个文件,该文件的内部标识号是 5,于是文件 5 就有两个外部符号名,实现了一个文件可以起多个文件名的要求。

10.4　辅存空间的分配和释放

一个文件系统不但要对系统中已存在的文件进行管理,并且还要有效地管理和查寻目录,通过对目录的管理为用户提供快速、有效、方便地访问所需文件的便利。除此之外,当一个用户要求建立一个新的文件,和撤消一个老的文件时,还要为用户分配辅存中的文件存储空间或收回已撤消文件的辅存空间,这是文件系统的又一重要功能。

为了能对辅存空间进行有效地分配和释放,首先要管理好辅存中的空闲块资源。同内存管理中一样,对空闲块(或称自由块)管理方法的好坏,对分配和释放算法的效率影响很大。

10.4.1　辅存空闲块的管理

常用的管理方法有下述几种。

1. 空闲文件项

这是把每个空闲区看成是一个文件,并把它登记在文件目录中,如图 10.10 所示。分配时系统依次扫描整个目录表、找寻标志为空闲的表目,然后比较该空闲区大小是否符合要求。如符合,则挑选该区分配之。

这种方法存在一些明显的缺点:

(1) 增加了目录的尺寸:由于空闲区和真正的文件目录表目混杂在一起,无疑增加了目录的尺寸,这使得查寻文件时增加了扫描目录的时间,降低查找速度。同时也使得分配文件的存储空间时,降低了查找空闲区的速度。

(2) 增加了目录管理的复杂性: 因为在通常的简单目录结构中, 目录中各表目的排序是按文件起始地址从小到大排列的。这样当空闲区大于所申请空间的尺寸, 就会剩下一

序号	名　　字	逻辑记录的大小	逻辑记录的个数	第一物理块地址	物 理 块
1	人事文件	80	100	2	2～9
2	空　　闲	1000	4	10	10～13
3	工资文件	132	5	14	14
4	空　　闲	1000	5	15	15～19
	⋮				

图 10.10　空闲文件项例子

块, 因此又需要插入一个表目。释放文件空间时要合并前后的相邻空闲块。这两种情况都要引起目录中各表目内容按地址重新排序。

这种方法仅当系统中只有少量的大空闲区时才适用。

2. 空闲区映象表法

这种方法是把上述方法中的各空闲区从目录中抽出来形成一个空闲区映象表文件。该文件只在目录中占据一个表目(也可以不把此表放入目录中)。空闲区映象表文件中为每一个空闲区分配一个表目, 表目内容通常包括空闲区大小(以物理块数计)和起始块地址, 该表常按地址排序。

3. 空闲块链

是把所有的空闲块链接在一起, 系统设置一个空闲块链的头指针以指出第一个空闲块的位置(物理块号), 每一个空闲块均有一个链接指针(通常是该块的最后一个单元)指向下一个空闲块位置, 从而形成一条链, 最末一个空闲块应有链尾标志(如 NIL)。这种方法的好处在于节省了空闲区映象表文件所占空间, 而且在分配和释放文件空间时通常不需要查表。

4. 字位映象图(盘图)

这是一种比较通用的方法, 像 PDP11 的 DOS 和微型机 CP/M 操作系统, IBM 的 VM/370 系统均使用这种技术。它是以“位”(bit)的值为“1”或“0”来反映磁盘上的相应物理块是否已分配的一种方法, 通常又称为“盘图”。具体方法是用若干个连续的字节构成一张表, 其中每一位对应于盘上的一个物理块, 依次顺序为 0, 1, 2, ……。如果某位的值为“1”, 表示该对应的物理块已分配, 值为“0”表示对应的物理块为空闲, 如图 10.10 的 8 块可用 16 位的字位串表示其分配情况如下:

1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0

(图 10.10 中的第 0 块, 第 1 块由系统占用了。)

由于这种盘图的尺寸是固定的(相应于盘的大小), 并且每一位表示一个物理块, 所以有可能用不大的盘图把整个磁盘的使用情况反映出来, 这样完全可以把盘图整个存放在内存中, 从而方便了文件空间的分配和释放操作。(参阅 14.6.2 节)

10.4.2 辅存空间的分配和释放

在磁盘上分配和释放文件空间的问题,类似于多道程序系统中的可变分区的主存管理情况。最初,整个存储空间可连续地分给文件使用,但随着文件空间的分配和释放,盘上就会出现许多“碎片”。如果还想为文件分配空间,就不得不分给一个文件若干个不连续的“碎片”。

解决的办法是定期进行“压缩”,把这些碎片集中为一个空闲区或若干个大的空闲区。压缩的时机通常是关机前。有些系统也可以在操作进行过程中动态地进行压缩。压缩中往往对文件重新组织,使其存放在一个连续的区域中。

辅存的分配和释放算法本身是比较简单的,本节只就分配中的一些问题作些粗略的介绍。

1. 连续分配

如果系统采取连续分配方式,即文件被存放在辅存的连续存储区中。在这种分配算法中,用户必须在分配前说明被建文件所需的存储区大小。然后系统查找空闲区的管理表格,看看是否有足够大的空闲区供其使用。如果有,就分给其所需的存储区,如果没有,该文件就不能建立,用户进程必须等待。

连续分配的优点是查找速度比其它方法要快。目录中关于文件的物理存储区的信息也比较简单,只需要起始块号和文件大小。其主要缺点是有碎片问题,需要定期进行存储空间的紧缩。

很明显,此种分配算法不适合文件随时间动态增长和收缩的情况,也不适合用户事先不知道文件有多大的应用情况。

2. 非连续分配

对于动态增长和收缩的文件,以及用户不知道文件有多大的应用情况,往往采用非连续分配辅存空间。这种分配策略通常有以下两种方法:

(1) 以扇区为单位的链接分配:即按所建文件的动态要求,分给它若干个磁盘的扇区,这些扇区可能分布在整个盘上而不一定相邻接。属于同一个文件的各扇区按文件记录的逻辑次序用链接指针相连接。

当文件要增长时,就从空闲区表中分给其所需之扇区数,并链接到文件的链上去。反之当文件收缩时,将释放出的扇区放回空闲区表中。

非连续分配的优点是消除了碎片问题(只消除了外部碎片,这里的情形类似于主存管理中的分页策略),不需要采取压缩技术。但是检索逻辑上连续的记录时,查寻时间较长,同时还存在链的维护开销,和链指针占用存储空间的开销。

(2) 以区段(或簇)为单位分配:这是一种比较有效的被广为使用的策略,其实质上是连续分配和非连续分配的结合。通常扇区是磁盘和主存间信息交换的基本单位,所以常以扇区作为最小的分配单位。本策略则不是以扇区为单位进行分配,而是以区段(或称簇)为单位进行分配。所谓区段是由若干个(在一个特定系统中其数目是固定的)连续扇区所组成。一个区段往往是整条或几条磁道所组成,文件所属的各区段可以用链指针、索引表(见 10.2 节)等方法来管理。当为文件动态地分配一个新的区段时,该区段应尽量接近

文件的已有区段号,以减少查寻时间。

此策略的优点是对辅存的管理效率较高,并减少了文件访问的执行时间,所以被广为使用。

10.5 文件的共享与文件系统的安全性

文件共享与文件系统的安全性是文件系统中的一个重要问题,共享与安全性是一个问题的两个方面。所谓共享,就是在不同用户之间共同使用某些文件,这不仅是完成共同的任务所必需的,而且还能节省大量辅存空间和主存空间,减少输入输出操作,更主要的是它为用户的应用提供了极大的方便,节省了用户的劳动。所以共享是文件系统性能好坏的标志之一。

但为了系统的可靠、用户的安全起见,文件的共享必须是有控制的。在当前的各计算机系统中,既为用户提供了共享的便利,又充分注意到系统和数据(文件)的安全性和保密性。

文件的共享涉及两个方面问题,一是对各类欲共享文件的用户进行存取控制,二是系统如何实现共享。

10.5.1 文件的连接

如何共享文件呢? 由上节中可以知道,用户想要共享其他用户的文件,如果该用户允许共享,那么只要知道该文件的路径名,所有其他用户均可用此路径名访问该文件(如果文件保护允许这种访问)。因此通常共享文件方法可分两种情况。

(1) 由系统来实现对文件的共享: 当用户要共享系统文件(如库文件,标准过程文件等),或已经为其所知的用户(如 Tu-Lide)和他的文件(如 Rooms,见图 10.9)的路径时,则用户可以通过提供从根目录出发的路径名来共享地访问这些文件,而不必将这些文件的目录也填写进欲共享的用户的符号目录中。所以这种共享方法是指用户不在自己的符号文件目录中对要共享的文件(属于其他用户的)建立相应的表目,或者说不对想要共享的文件进行连接,而由系统从根目录起为其查寻该文件,并提供相应的操作。

(2) 对欲共享的文件进行连接: 当用户如果想要经常使用其他用户的文件时,则用上述方法来共享就显得太慢,太不方便了。于是往往在用户自己的符号文件目录中对欲共享的文件建立起相应的表目,这称之为连接。连接可以在目录树的结点之间进行,也可以在结点和叶之间进行。如 MULTICS 系统中允许所有这些方式的连接。但是连接时必须十分小心,因为连接后的目录结构已经不是树形结构了,而成为网络状结构了,路径不是唯一的了。尤其在删除目录树的分支时,也要考虑到是否有连接。否则有些目录的连接指针很可能指向一个已被删除的不再存在的目录表目,从而引起文件访问出错。在 UNIX 系统中规定连接只允许在结点和叶之间进行。并且通过系统调用"link"来实现连接。例如 B 用户已知 A 用户有一个文件 F。A,B 两个用户是同组用户。从包含用户 A 和用户 B 的目录出发,文件 F 的路径名为 A/F,B 用户欲用符号名 C 来共享文件 F,并对文件 F 进行连接,B 用户通过系统调用

link(A/ F, B/ C)

来实现连接, 该过程要两个输入参数: 已存在的该文件的文件名 A/ F, 和共享用户新给该文件命的别名 B/ C。于是 link 过程就在用户 B 的目录中建立一个新的表目:

文件名	内部标识号
C	A/ F 的内部标识号

并且在文件 F 所对应的目录表目中的“连接数”项加 1, 这就实现了连接。用户 B 想通过文件名 C 访问文件 F, 就不必再通过根目录进行, 而可以直接用内部标识号进行访问。在 UNIX 系统中, 最多允许对共享文件起 127 个别名。

10.5.2 文件的存取控制

系统中的文件既存在保护问题, 又存在保密问题, 所谓保护是指文件免遭文件主本人或其他用户由于错误的操作而使文件受到无意的破坏。所谓保密是指文件本身不得被未经文件主同意的用户访问。这二者都涉及每一个用户对文件的访问权限问题, 即所谓文件的存取控制问题。通常在实现存取控制时, 可以有许多不同的方案, 本节介绍几种主要的保护措施。

1. 存取控制矩阵

存取控制矩阵是一个二维矩阵, 一维列出该计算机的全部用户, 另一维列出在系统中的全部文件(如图 10.11)。如矩阵元素 A_{ij}的值是“1”, 表示用户 i 被允许访问文件 j, 反之当 A_{ij}的值是“0”, 则表示用户 i 不允许访问文件 j。

用户 \ 文件	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	0	0
3	0	1	0	1	0	1	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	0	0	0
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0
9	1	1	1	1	0	0	0	0	1	1
10	1	1	0	0	1	1	0	0	0	1

图 10.11 存取控制矩阵

图 10.11 的存取控制矩阵在概念上是简单的, 但在实现上却有些问题。在一个大企业中有几千个核准的用户和几万个联机文件, 则二维矩阵就要占据很大的存储空间, 图示的

每个元素仅取值为“0”和“1”,以表示允许不允许用户进行访问。如果还想进而表明每个用户对某个文件的不同的访问类型:如只读、只写、只执行和读写以及不许访问等情况,则就需要用编码来表示之,则每个矩阵元素要用 3～4 位来表示此编码。

2. 按用户分类的存取控制(存取控制表)

存取控制矩阵由于太大而往往无法实现,一个改进的办法是按用户对文件的访问权力的差别对用户进行分类,通常分为:

(1) 文件主:他是文件的创建者;

(2) 指定的用户:由文件主说明了可使用此文件的一些特殊的用户;

(3) 同组同户:与文件主同属于某一特定课题组的成员,同组用户与此文件是有关的;

(4) 一般用户。

另一个改进办法是把各个文件的存取控制权力合并到每个文件的文件描述符中去加以说明(而不像存取控制矩阵那样整个系统的存取控制集中在一个矩阵中)。在每个文件描述符中分别指出四类用户的名字(见 10.3 节),并且在文件描述符中指出每类用户的访问权力。

在 UNIX 系统中就是使用此种存取控制方式。它把用户分成三类:文件主、同组用户、一般用户。文件描述符中说明了三类用户的名字(只说明前二类)。每个文件的存取权力说明是用一个 9 位的二进位来表示,这 9 位分成三组,每类用户 3 位,如图 10.12,图中每位的值为“1”时,表示允许此种类型的访问,为“0”时表示不允许此类访问。图中,文件主拥有全部访问权力。而同组用户只允许读和执行,不允许对文件进行写操作,而拒绝一般用户访问此文件。MULTICS 系统也使用此种存取控制方式。

图 10.12 UNIX 中存取控制例图

3. 口令

另外一个简便的办法是用户为自己的每个文件规定一个口令,附在文件目录(文件描述符)中。凡请求访问该文件的用户必须先提供口令。如果文件主想让其他用户使用此文件,就告诉他口令即可。此方法的优点是保护的信息量少,节省空间。但有很多缺点,由于口令存在文件中,系统程序员可以得到所有文件的全部口令,可靠性差。另外文件主为了便于记住口令,所以一般口令均比较短,这样,不诚实的用户也可通过多次试验来掌握该口令。同时文件主将口令告诉别人后,无法再收回以拒绝某用户继续使用文件。如想更改口令,还要通知一切有关的人。

4. 密码

针对口令的缺点,另一个方法是对所有文件用密码来编码。通常简单的做法是当用户存入一个文件时,他利用一个代码键来启动一个随机数发生器,产生一系列随机数,由文件系统将这些相继的随机数依次加到文件的字节上去。译码时减去这些随机数,文件就还原了。而这种译码方法,文件主只告诉允许访问的用户,系统程序员是不知道的。此方法增加了文件编码和译码的开销。

10.5.3 文件的转储和恢复

文件系统中不论是硬件和软件都会发生损坏和错误。例如自然界的闪电,电网中电压的变化,火灾、水灾等均可造成磁盘损坏,电源中电压抖动会引起存储数据的奇偶错。粗野和不慎的操作也会引起软、硬件的破坏。这些损害甚至可能危及多道程序系统的中枢部分(所有的系统程序也均以文件形式出现在文件系统中)。所以为使至关重要的文件系统万无一失,应对保存在辅存中的文件采取一些保险措施。这些措施中的最简便方法是"定期转储",使一些重要的文件有多个副本。常用的转储方法有两种。

(1) 全量转储:把文件存储器中所有文件,定期(如每天一次)复制到磁带上,这种方法比较简单,但有以下缺点:

① 转储时系统必须停止向用户开放;

② 很费机时,全部转储工作可能要数小时;

③ 当发生故障时,只能恢复上次转储的信息,而丢失了从上次转储以来的新变化和增加的信息。

定期性全量转储的一个优点是转储期间系统可以重新组织介质上的用户文件。例如把盘上不连续存放的文件重新构造成连续文件。

(2) 增量转储:每隔一定时间,把所有被修改过的文件和新文件转储到磁带上。通常系统对这些改过的和新的文件做上标志,当用户退出时,将列有这些文件名的表传送给系统进程,由它转储这些文件。

文件被转储后,当系统出现故障,就可以用装入转储的文件来恢复系统。

10.6 文件的使用与控制

前面我们研究了文件管理中的一些问题:包括文件的物理和逻辑组织,文件的目录组织,以及文件共享方法和存取控制的各种文件保护方法。本节我们将进入文件系统的另一个功能领域——文件的使用。如果说前几节主要是介绍文件系统的内部组织情况,那么这一节就是研究或描述文件系统呈现在用户面前的面貌,它的外部特征。

通常系统为用户提供若干条广义指令,使用户能够灵活、方便、有效地使用和控制文件。最基本的广义指令包括建立文件和撤消文件,打开文件和关闭文件,读文件和写文件以及其它的控制和使用操作(见 10.1.3 节)。

在介绍具体操作前,我们先介绍有关活动文件表的概念。

10.6.1 活动文件表和活动符号名表

在文件系统提供给用户的所有操作中几乎都要包括一个最基本的动作——查寻文件目录。由于现在计算机系统的文件很多,文件目录很大,不可能将全部目录放在主存中,通常把文件目录作为文件来处理,并把它存放在某个外存的存储卷上。当要查找某个文件时,就要逐块地把目录所在的物理块读入主存中,才能逐个表目地进行查找。但是进程每发出读进一数据块的命令后,它要在 IO 通道排队等待,磁臂查寻优化和定位并把该块从外存传输进主存。这不但速度慢,而且,查寻如果过于频繁将大大增加通道压力。为此我们在文件目录组织中想了许多办法:如为了减少查找长度(也为方便用户使用)采取多级目录结构,为了增加查找命中率(也为了便于共享)而把目录分为基本文件目录和符号文件目录两套目录组织,以及文件的连接功能等。本节我们介绍另一个措施即活动文件表和活动符号名表。

我们从程序的局部性理论可知,一个文件被用户访问后,很可能要被多次访问。为了防止每次访问文件时都要从外存把目录读入主存来查找一番,各系统都提供了一个"打开文件"的操作,并且为整个系统设置了一个"活动文件表",以及为每个用户设置了一个"活动符号名表"。这些表都建立在内存中的系统表格区(在 UNIX 系统中活动文件表称为活动索引节点表。而活动符号名表由用户打开文件表和系统打开文件表组成,每个进程都有一个用户打开文件表,在该进程的 user 结构中)。活动文件表的表目内容基本上同基本文件目录的表目内容。活动符号名表的表目内容大致同符号文件目录的表目内容。但应增加该文件"在活动文件表中的表目指针"项。即

文件名	内部标识号	活动文件表中表目指针

当系统初始启动时,自动将基本文件目录中的第一个表目(内部标识号= 1,该表目是说明基本文件目录本身这一文件的有关信息的)和第二个表目(内部标识号= 2,该表目是说明主符号目录,或者说根符号目录文件的有关信息)复制到活动文件表中。

每当用户打开某个文件时,系统就将该文件在基本文件目录中的相应表目复制到活动文件表中,将符号文件目录中的相应表目复制到活动符号名表中。这样当用户再次访问该文件时,就不必再到辅存中去查找符号文件目录和基本文件目录了。就可以通过活动符号名表和活动文件表中得到该文件描述符的全部信息,从而去访问盘上的文件。当文件关闭时,将此文件的表目从活动文件表和活动符号名表中撤消,以节省表目空间给其它进程使用。

10.6.2 建立文件命令

当用户想要把他的一批信息建立为一个文件时,他就使用系统提供的建立文件的命令。用户在调用此命令时,通常要提供以下参数:

文件名:用户使用的外部符号名;

设备号:指出该文件建立在哪类设备和哪一个存储卷上。

文件属性和存取控制信息,此项内容与具体系统有关,差别较大,通常要包括文件类型、文件大小、记录大小,存取控制等信息。

文件系统完成此命令的主要步骤如下:

(1) 在基本文件目录中为其分配一个空表目,并返回一个内部标识号(通常相应于表目序号);

(2) 在符号文件目录中分配一个空表目,并填入文件符号名与内部标识号;

(3) 调用存储分配程序为文件分配辅存空间;

(4) 将其在基本文件目录中的相应表目置初值,并填入物理地址;

(5) 调用打开文件命令将有关表目登入活动文件表和活动符号名表。

10.6.3 打开文件命令

为了避免用户在每次访问文件时从外存中查找目录,在 10.6.1 节中已经谈到,系统提供了打开文件命令和活动文件表与活动符号名表。所以打开文件命令的功能,是为用户访问文件作好准备,建立起与该文件的联系,具体的作用是把欲访问文件的目录表目(或文件描述符)内容读入活动文件表和活动符号名表。此命令要求的参数通常应包括文件名和设备名。

系统完成此命令的主要步骤如下:

(1) 查找符号文件目录树,以找出该文件的表目。如果找不到就转错误处理程序。找到时则返回该文件的内部标识号;

(2) 在活动文件表和活动符号名表中为该文件分配一个表目;

(3) 将有关信息填入活动文件表和活动符号名表中,并将该文件的"当前用户数"加 1(该数据项在活动文件表中)。

文件被打开后,可以多次使用,直到文件被关闭为止,不需多次打开。在有些系统中,也可以通过读写命令中隐含地向系统提出打开文件要求,不必事先用显式地打开文件命令来打开文件。但如果在读、写命令中不包括隐式打开文件功能的系统,读写文件前,用户必须事先打开文件,否则认为是出错。

10.6.4 读文件命令

文件的读和写是文件系统中最基本而又最重要的操作。读文件是把文件中的数据从外存中的文件区读入内存用户区。该命令的主要参数为:

文件名和设备号;

在文件中欲读的起始逻辑记录号;

欲读的记录数或字节数;

数据应送至的主存起始地址。

完成该操作的主要步骤如下:

(1) 按文件名从活动符号名表和活动文件表中找出该文件的文件描述符内容(即目录表目内容);

(2) 按存取控制说明检查访问的合法性;

(3) 按文件描述符中指出的该文件的逻辑和物理组织形式(包括存放方式,记录大小,起始物理块号等)将欲读的逻辑记录号和记录个数转换成物理块号;

(4) 将所有这些参数按设备管理程序的接口形式进行转换,并将此访问要求转送给设备管理程序,以完成数据交换工作。

10.6.5 写文件命令

当用户要求插入、添加、更新一个记录时,可以使用写命令或其它专用的相应命令。除要求分配盘空间外其参数和操作类似于读文件命令,不再赘述。

10.6.6 关闭文件命令

若文件暂时不用,则用户必须将它关闭,即切断与该文件的联系。文件关闭后一般就不能存取,除非重新打开它。该命令的参数同打开文件命令。其主要步骤如下:

(1) 撤消在用户的活动符号名表中的相应表目内容;

(2) 在活动文件表中该文件的"当前用户数"减1。如减1后,此值为"0",则撤消此表目的内容;

(3) 若活动文件表表目内容已被修改过,则在撤消此表目内容前,应将此表目内容写回盘上的基本文件目录的相应表目中去。

10.6.7 撤消文件命令

当一个文件已经完成了它的使命,不再被需要时,可用撤消文件命令将此文件撤消。其所需的参数为文件名(本节中所述的文件名均为路径名)。其主要操作步骤如下:

(1) 清除用户符号文件目录中的相应表目;

(2) 释放该文件在外存的文件存储空间;

(3) 清除该文件在基本文件目录中的相应表目。

这里要注意的是:

(1) 撤消文件前应先关闭文件;

(2) 若此文件对另一文件进行了连接访问,则应将被连接文件中的"连接数"(在文件描述符中)减1;

(3) 只有当被撤消文件的"当前用户数"为"0"和"连接数"为"0"时,该文件才能被撤消。

10.7 Windows NT的多重文件系统

Windows NT有一个稳健的文件系统,其特点在于:

(1) 具有很强的文件系统恢复功能;

(2) 可以处理巨大的存储媒体(可达2^{64}个字节);

(3) 支持多种文件系统。包括DOS的FAT文件系统,OS/2的高性能文件系统(HPFS),CD-ROM文件系统(CDFS)以及NT文件系统(NTFS);

(4) 具有很强的安全特性;

(5) 文件名采用 Unicode 国际代码表示;

(6) 支持 POSIX 操作系统环境中如文件的连接,大小写敏感文件名;

(7) 可用动态连接库(DLL)对这些文件系统进行装入和卸出。

习　　题

10-1　说明下列术语:记录、文件、文件系统、特殊文件、目录文件、路径、文件描述符。

10-2　文件系统的功能是什么?有哪些基本操作?

10-3　文件按其用途和性质可分成几类,有何特点?

10-4　有些文件系统要求文件名在整个文件系统中是唯一的,有些系统只要求文件名在其用户的范围内是唯一的,请指出这两种方法在实现和应用这两方面有何优缺点?

10-5　什么是文件的逻辑组织?什么是文件的物理组织?文件的逻辑组织有几种形式?

10-6　文件的物理组织常见的有几种?它与文件的存取方式有什么关系?为什么?

10-7　多级索引顺序文件是如何组织的?举例说明之。

10-8　提出多级文件目录结构的原因是什么?

10-9　允许不同用户对各自的文件赋于了相同的名字(同名问题),以及同组的不同用户对他们共享的同一文件按自己的爱好赋于了不同的文件名(多名问题)。同名和多名的便利是文件系统应具有的,试问文件系统是如何解决同名和多名问题的?

10-10　请画出下面所列术语的对应关系。

基本文件目录	层次树形结构
符号文件目录	线性表结构
活动文件表	活动文件表目
活动符号表	活动符号表目

10-11　在一个层次文件系统中的路径名有时可能很长,假定绝大多数对文件的访问是用户对自己文件的访问,文件系统有何种方法来减少使用冗长的路径名?

10-12　何谓文件的连接?有何作用?

10-13　有些系统要求显式地打开文件,而有些系统把打开文件作为第一次访问某文件的隐式部分。为何显式打开文件更好些?

10-14　比较几种对文件存取控制方法的特点?

10-15　Windows NT 的输入输出系统都包括哪些部分?有哪些特点?

10-16　Windows NT 的输入输出管理程序主要做些什么工作?

10-17　Windows NT 的驱动程序由哪些主要的标准部件构成?对块设备和字节设备的数据传输有区别吗?是什么?

10-18　你对 Windows NT 的 I/O 系统的认识是什么?有何评价?

第 六 部 分

操作系统结构与范例

第 11 章　操 作 系 统 的 结 构 和 设 计

11.1　操作系统的设计

操作系统作为一个软件系统来说,由于其庞大与复杂性而著称。以 IBM 公司的 OS/360 系统为例:它由 4000 个模块组成,共约 100 万条指令,花费 5000 人年,经费达数亿美元。在 60 年代软件危机的年月中,由于其所陷入的困境,曾推动了软件工程技术的诞生。毫无疑问今天操作系统的开发应遵循软件工程的原则和方法。

11.1.1　设计的目标和原则

软件开发是当今世界最大的产业之一。软件产品是为了销售,所以产品的设计目标植根于市场需求,必须从分析市场需求出发才能确定相应的设计目标。对于操作系统和其它软件产品来说,通常具有以下的设计目标:

(1) 可维护性与可扩充性:软件是必定要被修改的,容易修改与否称之为可维护性。常分为三种维护情况

① 改错性维护。软件开发中免不了出错,有经验的程序员平均也有 1/200 的出错率(指 200 条语句中将会有一个错),改正已发现的错误称之为改错性维护。

② 适应性维护。修改软件使之能适应新的运行环境(包括硬件环境和软件环境)。通常所说的移植性是指整体规模上的适应性。

③ 完善性维护。修改软件使之增加新的功能。即通常所谓的可扩充性。

(2) 可靠性:软件可靠性包括两方面的含意。

① 正确性。软件应能正确地实现用户所要求的功能和性能。

② 稳健性。软件能按预定方式对任何意外(硬件故障和误操作等)作出适当处理。

(3) 可理解性:软件产品要经过测试,要被人维护,要被人阅读或交流。所以要易于被人理解才能方便容易地进行测试、维护和交流。

(4) 有效性:一个软件产品应有效地使用资源,尤其是不过分地使用时空资源(指CPU 时间和主存空间)。对操作系统来说,其有效性不但表现在计算机各种资源的充分利用,而且还要提高系统所做的有用工作(指为应用程序用户提供资源服务的生产性工作)的比例、降低系统本身在为用户提供服务中的非生产性开销比例。

为达到以上目标,在操作系统设计中应遵循以下软件工程原则:

(1) 抽象原则:所谓抽象是指提取事物的共性。要将客观世界中一个复杂的问题用软件来解决。首先要为问题建立一个求解的模型,在软件工程中常称为逻辑模型。毫无疑问,这需要对问题中各方面的事物进行概括和综合,按某种观点提取其共性,也就是抽象,以导出问题求解的逻辑模型。其次,抽象不是最终目的,抽象正是为了便于将复杂问题分解为一个个较为简单的部分,以便对每个简单的部分逐个地加以解决。所以抽象是解决复杂问题的有力工具,是分解的有效手段。

(2) 信息隐蔽和信息局部化:这是软件工程中两个相关联的原则。信息隐蔽是指"使得一个模块内的信息(过程和数据)对于不需要这些信息的模块来说,是不能访问的"。这有什么好处呢?首先便于修改,改善可维护性。使得修改一个模块的功能及其实现的方法时,不会影响其它模块也随之需作相应的修改。其次是使程序员只需了解其它模块能为其提供什么样的服务功能,不需了解这些功能如何实现的细节,从而可使程序员把精力集中在设计的主要方面。回想一下在 4.3.3 节中解决生产者和消费者问题的情况:每个进程均需要做 P 操作进入临界段,进入后要移动指针,要用 V 操作唤醒等待进程,以及退出临界段。这使每个与此有关的进程要知道许多细节(如几个信号量,信号量如何实现、缓冲队列如何实现、指针情况以及等待队列情况),而不能使程序员把精力集中于加工和产生信息这一主要工作方面。如果把该问题中的环形缓冲区改为线性的队列,那么就必须修改每个进程中的有关代码(如移动指针方法)。所以它的信息隐蔽是不好的(但 4.3.3 节中主要目的是讲解并行算法正确性,故可容忍此缺点)。比较正确的作法是设置一个缓冲区专用例程(如 4.5 节中管程概念),专为各进程提供放信息入缓冲区或从缓冲区取出信息。而各进程只要向该管程发消息请求管程的相应服务,这就做到了信息隐蔽(能说出是怎么做到信息隐蔽)。

信息局部化是指把一些关系密切的软件元素(如数据以及施加于该数据上的各种操作)物理地放得彼此靠近——放在一个模块中或一个程序包中。显然,局部化有助于实现信息隐蔽。

在面向对象的设计技术中的对象类(参看 12.4.1 节对象部分)具有很好的信息隐蔽和信息局部化特性。在面向对象的技术中称为封装性,其含意是把数据及其属性以及施加于数据上的操作,像信封一样全部封装在对象类里面,对象外部不能随便了解,只能通过对象提供的操作请求给予服务。

(3) 模块化:是把程序按模块独立性原则划分成若干个模块,使每个模块成为单入口、单出口单一功能的程序单元——模块,其主要目的是降低程序开发的复杂性,提高软

件的可维护性、可靠性和可理解性。

其它原则是一致性、完整性和确定性，不一一赘述。

11.1.2 操作系统的设计

经常听到读者以惊叹的口气问道："这么复杂的操作系统，他(她)们是怎么设计出来的呢？"本节的目的并不妄想指导操作系统设计，只是使本书的读者了解操作系统设计过程的概貌。

大型软件的开发过程遵循软件生存周期模型，该模型将软件生存期(life cycle，是指从软件开发任务提出到软件最终废弃不用这整个阶段)大致划分为五个阶段：

(1) 需求分析阶段；

(2) 设计阶段；

(3) 实现阶段；

(4) 测试阶段；

(5) 运行和维护阶段。

但由于在软件开发过程中，所采用的软件方法学的不同，第一、二两个阶段的具体实现差别较大。传统操作系统大多是采用基于结构分析和结构设计(SA-SD)方法。近来的操作系统较倾向于面向对象(Object-Oriented)的方法。在操作系统设计中，大致划分为以下五个阶段：

(1) 系统分析和总体功能设计阶段；

(2) 系统设计与结构设计阶段；

(3) 实现阶段；

(4) 测试阶段；

(5) 维护和运行阶段。

下面大致描述各阶段主要工作。

11.1.2.1 系统分析和总体功能设计阶段

操作系统的设计对软件设计人员来说是不同于一般大型软件的设计的。在通常的软件开发任务中，软件工程师们是为其它应用领域(如金融财会)问题去求得软件解法。但软件工程师对此领域问题几乎完全不熟悉。所以需求分析任务是繁重而又艰难的。但操作系统的设计者们面临一个自己十分熟悉的领域，他(她)们了解操作系统应具有的功能；操作系统是如何工作的以及操作系统的各种基本组成成分……。所以，需求分析工作要简单得多。但是设计操作系统是需要相当多的精力和经费的，所以这绝不是一个演练性质的任务。它必定是针对当前操作系统中的不足和市场需要所发生的变化而提出的设计任务。所以需求分析仍然是必须进行的。通常这个阶段要做这样一些工作：

(1) 需求分析：通过分析确定需求或修订需求语句。开发出系统目的、范围和所需功能。

(2) 信息分析：通过数据流图或实体关系图，标定在系统内的实体以及实体之间的关系。在此基础上再标定出系统对象(哪些实体描述和构成了某对象)。并建立起系统的静态结构——对象模型。

(3) 事件分析: 把系统看成是对外部事件的响应机, 以响应一系列外部事件。其目的是以外部观点来捕捉和分析系统的行为, 并标定对象的行为, 刻划对象的时序性质, 用系统或对象的状态转换图来建立系统的动态模型。

(4) 功能分析: 按对象的操作刻划出如何由输入得到输出的功能模型。这方面的图象工具是功能流图。

系统分析完成后, 建立起了系统的模型和系统规格说明书, 详细描述了系统的功能和性能要求。

11.1.2.2 系统设计与结构设计阶段

这部分工作是建立在前一阶段工作基础上的, 常包括以下几方面:

(1) 对前一阶段工作精细化: 在有些软件方法学中把这阶段称为细化阶段。如果说前一阶段是在系统的顶层或高层进行的系统分析并建立了系统模型和功能流图, 则这一阶段工作就要更精细化并向深层次进行。在此过程中调整和细化系统模型和各种图形和文字的描述文档。

(2) 调整和精细化功能分配: 如内核完成哪些功能, 提供哪些支持。每种功能实现的事件或活动链以及相关的全局性的数据结构设计。特别注意各事件的行为特性(顺序或并行)以及同步机制。

(3) 用快速原型技术建立模型以改善调度算法和功能实现机制的性能和效率。例如进程之间通信方式由于需要在进程间切换, 对效率影响到什么程度; 如何改善; 效果如何; 均需通过模型试验。

(4) 在层次结构的部分, 将功能模块分配在相应的层次。并检查各功能路径的工作是否能很好实现。

通常(1), (2)两步骤需反复进行。对实现阶段和测试阶段由于问题比较单一, 不再讨论。

11.2 操作系统的结构

随着计算机技术的飞速发展, 软件也变得愈来愈庞大而复杂。从而推动了软件开发技术由软件作坊向软件工程阶段发展。操作系统的设计技术和结构也如同其它大型软件技术一样地不断取得进展, 从而形成了操作系统的三个发展阶段。按操作系统结构特点划分的三个阶段为模块接口法(单块式)阶段、层次结构法阶段以及客户/服务器(client/server)式阶段。

11.2.1 模块接口法(单块式)

这是操作系统最早采用的一种结构设计方法。在 1968 年软件工程出现以前的早期操作系统(如 IBM 的操作系统)以及目前的一些小型操作系统(如 DOS 操作系统)均属此种类型。

什么是模块? 通常定义为“命名的一段程序语句”, 可理解为“子程序”。模块是构成软件的基本单位。软件工程技术出现后, 要求模块应是单入口、单出口和单一功能的。但以

前的模块常不具有这种特性,而是多入口、多出口,模块中具有多个功能。

操作系统中往往具有大量的模块,有些系统中模块多达几千个。操作系统为了完成用户所要求的服务,通常要若干个模块共同来完成,如为用户建立一个进程。则要调用“创建进程”模块,而创建进程模块还要调用“分配主存”模块为新进程分配进程控制块 PCB 的主存空间,以及调用“将一个进程插入队列”模块,以便把新创建的进程 PCB 插入就绪队列。所以模块间的调用关系是复杂的。在两个模块的相互调用过程中要有输入和输出参数的传送(如调用分配主存模块要传送给它输入参数——主存大小,分配主存模块要返回给调用者输出参数——主存地址或说明分配无法满足的状态参数),通常把这种情况称为模块间的接口关系。接口关系有简单和复杂之分。这取决于三个因素:

(1) 模块间调用的方式(直接引用和过程调用)。

(2) 模块间参数的作用(是起控制作用的开关量,还是作为数据使用)。

(3) 相互间传送参数的数量。

一个模块如果是多入口和多出口的,自然是有多个进入和离去该模块的渠道,该模块与其它模块联系就多,调用方式是直接引用(即不是通过过程调用语句,而是通过 go to 语句或转移指令直接进入模块内部)。一个模块如果具有多个功能,调用者必然要有参数传给该模块指出要用哪个功能,所以传送的参数是起控制作用的开关量。这种情况称为模块接口关系复杂,软件工程中术语称之为“模块间耦合紧密”。

以上介绍了什么是模块接口法,那么模块接口法有什么特点呢?

(1) 模块之间调用关系如图 11.1 所示,也就是对模块间调用关系没有限制,可以任意调用。所以有人也把模块接口法称为无序调用法。

(2) 模块之间耦合紧密。早期的操作系统中对模块独立性未给予充分的注意,使模块间接口关系多而复杂。

由于以上两个特点,近来许多人把模块接口法称为单块式。因为紧密的耦合和错综复杂的调用关系使得这些模块如同形成了一个坚实的整体。单块式的名称勾划出了这种方法的本质。这种结构方式给操作系统设计带来如下缺点:

(1) 系统的结构关系不清晰:这种方法使得各模块之间构成了如图 11.1 所示的复杂的网络关系,这种网络是一个相当复杂的有向图,无规律地相互调用,相互依赖,使得人们难于对结构作出清晰的了解和判断。也难于对系统作局部修改,因为很可能会造成牵一发而动全身地连锁式修改有关诸模块。因而使系统的易懂性,可适应性变差。

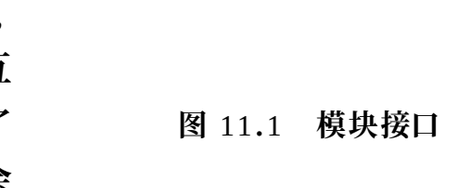


图 11.1 模块接口

(2) 使系统的可靠性降低:因为这种复杂的网络关系,使得各块之间可互相调用,以至于构成了循环。这样的情况,弄得不好就易于使系统陷于死锁。又由于在操作系统的总体功能设计和功能分配阶段,以非常粗略和比较模糊的方式把系统划分为若干个功能模块,并规定了模块间的接口,然后各模块齐头并进地进行设计。这样,更加强了系统循环调用的危险性,很难保证每个模块设计的正确性。使得系统的可靠性降低。

11.2.2 层次结构设计法

显然,要清除模块接口法的缺点就必须减少各模块之间毫无规则地互相调用,相互依赖的关系,特别是清除循环现象。层次结构设计方法正是从这点出发的,它力求使模块间调用的无序性变为有序性。因此所谓层次结构设计方法,就是把操作系统的所有功能模块,按功能流图的调用次序,分别将这些模块排列成若干层,各层之间的模块只能是单向依赖或单向调用(如只允许上层或外层模块调用下层或内层模块)关系。这样不但操作系统的结构清晰,而且不构成循环。图 11.2 表示了这种调用关系。

图 11.2 层次结构关系

在一个层次结构的操作系统中,若不仅各层之间是单向调用的,而且每一层中的同层模块之间不存在互相调用的关系,则称这种层次结构关系为全序的层次关系,如图 11.2 中实线调用关系所示。但是在实际的大型操作系统中,要建成一个全序的层次结构关系是十分困难的,往往无法完全避免循环现象(请注意,循环调用和循环等待不一定就发生死锁,循环等待在死锁研究中只是个必要条件而不是充分条件,所以尽管存在循环现象,只要小心加以控制是可以避免死锁的),此时我们应使系统中的循环现象尽量减少。例如我们可以让各层之间的模块是单向调用的,但允许同层之间的模块可以互相调用,可以有循环调用现象,这种层次结构关系称为半序的,如图 11.2 中虚线调用关系所示。

层次结构法的优点是它既具有模块接口法的优点——把复杂的整体问题分解成若干个比较简单的相对独立的成分,即把整体问题局部化,使得一个复杂的操作系统分解成许许多多功能单一的模块。同时它又具有模块接口法不具有的优点,即各模块之间的组织结构和依赖关系清晰明了。这不但增加了系统的可懂性和可适应性,而且还使我们对操作系统的设计的每一步都建立在可靠的基础上。因为层次结构是单向依赖的,上一层各模块所提供的功能(以及资源)是建立在下一层的基础上的。或者说上一层功能是下一层功能的扩充和延续。最内层是硬件基础——裸机,裸机的外层是操作系统的最下面(或内层)的第一层。按照分层虚拟机的观点,每加上一层软件就构成了一个比原来机器功能更强的虚拟机,也就是说进行了一次功能扩充。而操作系统的第一层是在裸机基础上进行的第一次扩充后形成的虚拟机,以后每增加一层软件就是在原机器上的又一次扩充,又成为一个新的虚拟机。因此只要下层的各模块设计是正确的,就为上层功能模块的设计提供了可靠基础,从而增加了系统的可靠性。

这种结构的优点还在于很容易对操作系统增加或替换掉一层可以不影响其它层次。便于修改、扩充。

层次结构的操作系统的各功能模块应放在哪一层,系统一共应有多少层,这是一个很自然地会提出的问题。但对这些问题通常并无一成不变的规律可循,必须要依据总体功能设计和结构设计中的功能流图和数据流图进行分层,大致的分层原则如下:

(1) 为了增加操作系统的可适应性,并且方便于将操作系统移植到其它机器上,必须把与机器特点紧密相关的软件,如中断处理,输入输出管理等放在紧靠硬件的最低层。这样经过这一层软件扩充后的虚拟机,硬件的特性就被隐藏起来了,方便了操作系统的移植。例如第 14 章中的 CP/M 与 MP/M 操作系统就采用层次结构。为了便于修改移植,它把与硬件有关和与硬件无关的模块截然分开,而把与硬件有关的 BIOS(管理输入输出设备)放在最内层。所以当硬件环境改变时只需要修改这一层模块就可以了。

(2) 对于一个计算机系统来说,往往具有多种操作方式(例如既可在前台处理分时作业,又可在后台以批处理方式运行作业,也可进行实时控制),为了便于操作系统从一种操作方式转变到另一种操作方式,通常把三种操作方式共同要使用的基本部分放在内层,而把随三种操作方式而改变的部分放在外层:如批作业调度程序和联机作业调度程序,键盘命令解释程序和作业控制语言解释程序等,这样操作方式改变时仅需改变外层,内层部分保持不变。

(3) 当前操作系统的设计都是基于进程的概念,进程是操作系统的基本成分。为了给进程的活动提供必要的环境和条件,因此必须要有一部分软件——系统调用的各功能,来为进程提供服务,通常这些功能模块(各系统调用功能)构成操作系统内核,放在系统的内层。内核中又分为多个层次,通常将各层均要调用的那些功能放在更内层。

11.2.3 客户/服务器方式

操作系统结构技术的发展是与整个计算机技术的发展相联系的。当前计算机技术发展的突出特点是要求广泛的信息和其它资源的共享。这一要求促使网络技术的普遍应用和发展。由于网络技术逐渐成熟并实用化,再加以数据库联网已是计算机应用的新趋势。所以为用户提供一个符合企业信息处理应用要求的分布式的系统环境是十分必要的。事实上,在一个企业或部门中,数据一般总是在它产生的各个现场上就近地被存储、管理、加工、组织和使用。只有少量的数据或加工后的信息才是供全局共享或为某些其它局部所使用的。所以分布式处理才是真正合乎客观实际和新的应用需要的潮流。如果操作系统是采用客户/服务器结构,它将非常适宜于应用在网络环境下,应用于分布式处理的计算环境中。所以说客户/服务器模式是第三代的操作系统(前述的两种模式分别是操作系统的第一代和第二代模式)。

客户/服务器结构模式的操作系统有卡内基·梅隆大学研制的 Mach 操作系统和美国微软公司(Microsoft)研制的 Windows NT 操作系统。它们的共同特点是操作系统由下面两大部分组成:

(1) 运行在核心态的内核(Mach 中称为微内核,Windows NT 中称为 NT 执行体):它提供所有操作系统基本都具有的那些操作,如线程(Thread)调度(参看第 12 章

Windows NT)、虚拟存储、消息传递、设备驱动以及内核的原语操作集和中断处理等。这些部分通常采用层次结构并构成了基本操作系统。

(2) 运行在用户态的并以客户/服务器方式活动的进程层: 这意味着除内核部分外, 操作系统所有的其它部分被分成若干个相对独立的进程, 每一个进程实现一组服务, 称为服务器进程(用户应用程序对应的进程, 虽然也以客户/服务器方式活动于该层, 但不作为操作系统的功能构成成分看待)。这些服务器进程可以是各种应用程序接口 API(参阅第 12 章 Windows NT)或文件系统以及网络等(Mach 操作系统)。服务器进程的任务是检查是否有客户提出要求服务的请求, 在满足客户进程的请求后将结果返回。而客户可以是一个应用程序, 也可以是另一个服务器进程。客户进程与服务器进程之间的通信是采用发送消息进行的。这是因为每个进程属于不同的虚拟地址空间, 它们之间不能直接通信, 必须通过内核进行, 而内核则是被映象到每个进程的虚拟地址空间内的, 它可以操纵所有进程。客户进程发出消息, 内核将消息传给服务器进程。服务器进程执行相应的操作, 其结果又通过内核用发消息方式返回给客户进程, 这就是客户/服务器的运行模式。

这种模式的优点在于, 它将操作系统分成若干个小的并且自包含的分支(服务器进程), 每个分支运行在独立的用户态进程中。相互之间通过规范一致的方式接口—— 发送消息, 从而把这些进程链结起来。这种模式的直接好处是:

(1) 可靠性好: 由于每个分支是独立的自包含的(分支之间耦合最为松散), 所以即使某个服务器失败或产生问题, 也不会引起系统其它服务器和系统其它组成部分的损坏或崩溃。

(2) 易扩充性: 便于操作系统增加新的服务器功能, 因为它们是自包含的, 且接口规范。同时修改一个服务器的代码不会影响系统其它部分, 可维护性好。

(3) 适宜于分布式处理的计算环境: 因为不同的服务器可以运行在不同的处理器或计算机上, 从而使操作系统自然地具有分布式处理的能力。

习　　题

11-1　操作系统的设计目标有哪些? 为什么?

11-2　软件工程的原则有哪些?

11-3　信息隐蔽和信息局部化有何重要意义? 面向对象的设计技术中是如何做到信息隐蔽的?

11-4　Windows NT 被称为是第三代操作系统, 那么这三代操作系统的结构有哪些类型? 有什么特点?

11-5　模块接口法为什么称为单块式结构?

11-6　什么叫模块? 为什么要设计单入口、单出口和单一功能的模块?

11-7　模块接口的复杂程度取决于哪三个因素? 哪种情况最好?

11-8　层次结构法有何优点?

11-9　试述分层原则有哪些?

11-10　什么叫客户/服务器模式? 有何好处? Windows NT 为什么采用这种模式?

第 12 章　Windows NT 操作系统

12.1　Windows NT 操作系统概述

Windows NT 是微软(Microsoft)公司于 1993 年推出的一个崭新的 32 位操作系统。该系统自 1989 年 初确定系统设计目标和设计规范以来,数千万 PC 机用户就翘首以待。直至推出之时,在世界上所引起的轰动效应,在操作系统的历史上可以说是实属罕见。这固然有赖于微软公司在 DOS 和 Windows 上取得的巨大成功,更归功于 Windows NT 以其最新的技术、完善的功能和绝妙的性能而深深地打动了计算机世界,而被誉为“ 90 年代的操作系统 ”、“ 所有操作系统之母 ”。

Windows NT 是用 C 和 C^{++} (少部分用汇编语言)语言编写的通用操作系统。它采用操作系统的最新技术:例如深受欢迎的 Windows 图形用户界面技术;支持多操作系统运行环境;对称多处理能力;内装网络功能;多重文件系统与异步 I/O 以及采用面向对象的软件开发技术等。它提供了现代操作系统的几乎所有功能:例如多任务能力;多处理系统;虚拟资源管理;统一成一体化的 I/O 系统;网络通信功能等,它具有很高的系统性能……。无疑这是一个被人们所期待的绝佳的现代操作系统。下面让我们首先看一下这个操作系统的设计目标是如何确定的?

12.2　Windows NT 的设计目标

对任何软件开发来说,设计目标是软件设计的根本出发点,所以首先确定设计目标是十分重要的。但设计目标又是根据什么确定下来的? 在商品社会中,推动技术发展的动力是市场需求。一个软件产品只有受到用户的欢迎、爱用,它才有生命力。畅销才能占有巨大市场,长久不衰。微软公司在分析市场需求发展的基础上制订了下述设计目标。

1. 可扩充性

软件在它的“ 生存周期 ”过程中,随着时间推移必将要改变的(在软件工程中称为“ 可维护性 ”),微软公司考虑到在 NT 的生命期内,必将有新的技术和新的需求,所以要求操作系统必须易于扩充,并随着市场需求的变化而易于改动。

2. 可移植性

可移植性是与可扩充性密切相关的(两者均属可维护性),可扩充性使操作系统很容易被增强。而可移植性使整个操作系统以尽可能少的改动,便能移植到一个具有不同的处理器或不同配置的计算机上(可扩充性和可移植性是一个程度概念,问题不在于能不能扩充和移植,而在于进行扩充和移植时有多大的难度)。

3. 可靠性

可靠性是 Windows NT 的第三个目标,它的可靠性包括了两种不同的、却又相关的概念:

(1) 操作系统应该是稳健的(robust,软件工程中把可靠性定义为正确性和稳健性两方面),也就是说操作系统对一切意外出错情况和硬件故障都能按预定方式作出处理,不会引起系统不良后果。

(2) 操作系统应主动地保护自身及其用户免遭用户程序偶然或有意的破坏。

4. 兼容性

兼容性是指该操作系统能执行为其它操作系统(包括本系统的早期版本)所编写的程序的能力。当然在确定兼容性时还要考虑是应用程序二进制兼容(需要一个仿真程序将原来机器的指令集转换到另一个机器的指令集即可得到二进制兼容)还是源码级兼容。Windows NT 对微软公司的老版本(包括 MS-DOS ,16 位 Windows ,OS/2, LAN Manager)提供使用仿真器的二进制兼容。对 POSIX (面向计算环境的可移植操作系统接口, Portable Operating System Interface for Computing Environments, 美国政府已把遵守 POSIX.1 作为购买操作系统时的要求。)的应用程序提供源码级兼容。

5. 高的系统性能

为了获得最佳性能,对于许多计算量大的应用程序(如图形软件包,仿真软件包,金融分析软件包)为了能给用户良好的响应时间,需要快速的处理。为此除了硬件要具有良好的性能外,操作系统也必须快而且有效。

12.3 Windows NT 的系统模型

本节中所介绍的是 Windows NT (以下简称 NT)在进行操作系统设计和结构中,所依据的主要的概念模型,他们称之为操作系统模型,并定义为“把操作系统所提供的特性、服务以及系统所执行的任务统一成一体的一个概括性的构架”。

Windows NT 的模型是由客户/服务器(Client/Server)模型、对象(object) 模型、对称多处理(SMP)模型,这样三种模型的组合。

1. 客户/服务器模型

客户/服务器模型是 90 年代的操作系统结构。其思想是把操作系统分成若干进程。每个进程实现单个的一套服务,例如:主存服务、进程生成服务、处理器调度服务。每个服务器运行在用户态,它执行一个循环,以检查是否有客户已请求某项服务。而客户可以是另外的操作系统成分,也可以是应用程序。客户通过发送一个消息给服务器进程来请求提供一项服务。运行在核心态的操作系统内核把该消息传送给服务器;而后该服务器执行有关

操作,操作完成后;内核用另一消息把结果返回给客户。

NT 的设计者们认为客户/服务器模型有以下好处:

(1) 提供多种操作系统运行环境的支持(这是 NT 设计目标之四——兼容性的要求)。为此要提供多种应用程序编程接口 (API)以及使这些编程接口 API 的过程及其基础执行环境与本机器的环境完全相容。怎样才能较好地实现这一要求?曾设想了多种方案(包括用层次结构方法)均遇到了困难,只有客户/服务器模型才产生满意的结果。而且可以简化基本操作系统——NT 执行体,因为它把每个 API 放到独立的服务器中,这样即可避免与执行体的冲突和重复,又有利于以后扩充增加新的 API。

(2) 改进了可靠性,这是由于:

① 每个服务器是以分配给它的主存分区的独立进程的方式运行,因而防止了受其它进程的影响。另外由于服务器运行在用户态,它们不能直接访问硬件或修改执行体的存储区。

② 每个服务器以独立的用户进程方式运行,因此单个服务器出现故障不会引起操作系统其它部分崩溃等问题。

(3) 适宜于分布式计算模型。由于网络上的计算机是以客户/服务器模型为基础的,并且也是用消息来通信。本地服务器(或客户)可以容易地给远程计算机上的客户应用程序(或服务器)发送消息。

2. 对象模型

现代软件工程学为开发软件提供了多种技术。而传统的设计技术是结构分析和结构设计(SA-SD)技术,包括 Jackson 面向数据结构的设计方法均是采取自顶向下的进行分解和不断精细化的方法,并由此导出模块的层次结构图。在这种设计中,系统均有个"主程序",以控制、管理和调度下层模块。而面向对象(又称为 O-O 的方法,Object-Oriented)的软件开发技术是当前最新、最受关注的技术,它与传统的基于功能分解的技术是全然不同的概念。O-O 技术认为问题世界是由诸实体(又称为对象,Object)构成,开发软件是要找出实体并描述它,同时还要研究和描述实体间的关系。这种开发技术得到了广泛的应用并获得成功。而操作系统设计者们关心 O-O 技术,是认为 O-O 技术很适合开发操作系统。因为有人把操作系统称之为没有"顶"的程序,实际上操作系统像其它大的软件系统一样,要找出一个操作系统的单个"主程序"是困难的。而 O-O 软件开发方法学不是试图去设计一个这样的自顶向下的系统,而把设计目标集中到找出软件为完成其工作所必须处理的对象上面。"对象"是个抽象的数据结构,它符合软件工程原则——抽象、信息隐蔽和信息局部化,因而便于开发高质量软件。对象是现代软件系统中最好的构成部件。

NT 操作系统使用对象模型有以下优点:

(1) 操作系统访问和操纵其资源是一致的。因为操作系统无论生成、删除和引用一个事件对象、资源对象、进程对象都是用相同的方法——通过使用对象句柄;

(2) 所有的对象采用同样的保护方法,因此简化了安全措施。

3. 对称多处理模型

当今的微型计算机已迅速向多处理器模式过渡,而 Windows NT 操作系统支持单处理机和多处理机的运行模式。

所谓多处理模式是指一台计算机中具有两个以上的处理机,可同时执行进程(NT 为线程)。每个处理机上同时都可以有一个进程(或线程)在执行。

多处理模式可分为非对称多处理和对称多处理。

非对称多处理(ASMP)操作系统模式又称为主从式。它选择一台处理机执行操作系统,负责对其它处理机的调度,负责对主存、外设和其它资源的管理。而其它处理机不执行操作系统,只执行用户的进程(线程)。所有的处理机共用主存。非对称多处理模式的主要缺点是万一主处理机坏了,系统就必须停下来。而且主处理机往往负荷过重,成为系统瓶颈。

对称多处理(SMP)系统允许操作系统在任何一个处理机上运行。也就是说各处理机的地位是平等的,它们都可以既执行操作系统又执行用户进程(或线程),共同负责管理系统主存、外设和其它资源。各处理机共用主存(对称和非对称多处理系统均是公用主存,因为是一个主存把多个处理机连接起来的)。对称多处理系统有很多优点。首先它能较好地挖掘多处理机能力,因为非对称式往往主处理机很忙,而其它处理机因等待它的调度而空闲,这就降低了系统吞吐量。其次是一个处理机失效不会影响其它处理机的正常运行,系统可靠。

Windows NT 是采用对称多处理模式,它有以下特点:

(1) 操作系统可在任一空闲的处理机上运行,也可以同时在各处理机上运行。而且当一个高优先级线程需要时,可以抢占操作系统所有代码(除内核部分外。内核部分是执行线程调度和处理中断的。)即强迫把操作系统给一个处理机。

(2) 一个进程的多个线程可以同时在多个处理机上执行(即分别在几个处理机上执行一个程序的不同部分)。

(3) 服务器进程可使用多个线程同时在不同的处理机上处理多个客户进程的服务请求。

12.4 Windows NT 的结构

Windows NT 的结构框图如图 12.1 所示,它的结构可以分为两部分:

(1) 系统用户态部分(Windows NT 保护子系统);

(2) 系统核心态部分(NT 执行体)。

图 12.1 中的黑粗线以上部分被称为保护子系统,是因为每一个服务器驻留在单独的进程中,它的主存由 NT 执行体的虚拟存储系统所保护,不受其它进程影响。下面分别介绍这两部分的结构特点。

1. 保护子系统

这部分是由诸客户进程,诸服务器进程所构成的成分,其结构关系为客户/服务器模型。

Windows NT 有两类保护子系统:环境子系统和集成子系统。

每个环境子系统是一个用户态服务器,为特定的操作系统提供一个 API。它为客户进程提供的服务是这样的,当一个应用程序调用其相应的某个 API(本系统提供 Win32,

图 12.1　Windows NT 框架

POSIX, OS/ 2, 16 位 Windows 和 MS-DOS 等子系统的编程接口 API)时, 一个消息通过执行体的本地过程调用(LPC)工具, 发送给完成该 API 程序的服务器——环境子系统。子系统执行 API 例程、并通过 LPC 将结果返回应用程序进程。

集成子系统是完成重要操作系统功能的服务器。这包括安全子系统, 网络软件中的若干部件(工作站服务和网络服务器服务, 都是用户态进程)。

用户态的每一个环境子系统都可以支持多个客户应用程序同时运行。而在多个子系统运行时, 所能看到的只有 Win32 子系统本身, 就好像 Win 32 运行所有的应用程序那样(Win 32 系统构成本机环境)。

当用户试图进入系统时, 首先必须进行登录, 由安全子系统对用户进行是否允许其进入和权限的检查与控制。安全子系统维护着一个有关用户帐号信息的数据库, 内容包括帐号名、保密字、用户权限等信息。任何非法用户或非法操作, 都被安全子系统拒之门外。

2. NT 执行体(executive)

在图 12.1 中黑粗线以下部分称为核心态的是NT 的执行体。其结构是层次式与微内核的结合。与 UNIX 相比, NT 执行体的内核是相当小的。NT 执行体基本上是一个完整的操作系统, 它由一组部件构成, 这些部件形成了层次结构。在图 12.1 中的系统服务实际上是 NT 执行体为用户态的进程提供的一个接口。层次结构的第一层是由系统服务下面的几个部件构成。这几个部件可以通过一组精心设计的内部界面互相调用。

NT 内核是第二层,它类似于 Mach 的微内核。最底下的一层是硬件抽象层(HAL),它将 NT 执行体的其余部分与运行机器的硬件特性隔离开来。而 I/O 系统内部又分成若干层,如图 12.2 所示,它们是由 I/O 管理程序、文件驱动程序和设备驱动程序构成层次。

下面介绍 NT 执行体各组成部件的主要作用:

(1) 对象管理程序:生成、管理及删除执行体对象。

(2) 安全引用监视程序:监视操作系统资源;执行运行对象的保护和审查;实施安全方针。

(3) 进程管理程序:它生成和终止进程及线程;执行暂停和恢复线程的执行;存储和检索有关 NT 进程和线程的信息。

图 12.2 分层驱动程序

(4) 本地过程调用功能(LPC):在同一台计算机中的客户进程和一个服务器进程之间传送消息,是远程过程调用的优化版本。

(5) 虚拟存储管理(VM) 程序:为每个进程提供专有地址空间;对进程地址空间进行保护;负责页面调度。

(6) 内核:对中断和异常作出响应;调度线程;提供一组基本对象和接口。

(7) I/O 系统包括下列子部件:

I/O 管理程序;

文件系统;

网络重定向程序和网络服务器;

设备驱动程序;

高速缓冲存储管理程序。

(8) 硬件抽象层(HAL)。

以上这些组成部件中的前六个部件都要实现两组函数:

系统服务:可以由环境子系统和其它执行体部件调用。

内部例程:只可由在执行体内的组成部件使用。

12.5 Windows NT 的基元成分——对象、进程和线程

12.5.1 对象

对象(Object)是个抽象数据结构,在 Windows NT 中用以表示所有资源——一个比传统操作系统远为广义的资源概念。

对象是数据和有关操作的封装体(传统的设计方法是采用数据和操作分离的模式),它包括数据、数据的属性以及可以施加于数据上的操作等三个成分。作为例子,下面给出一个用 C++ 表示的队列对象类:

```
Class queue{
    //以下是数据成员的说明
    int q[LENGTH]; //队列存储区
    int front, rear; //队头、队尾下标          数据及属性
Public
    //以下是函数成员的说明
    void init (void); //队列初始化
    void put (int i); //入队
    int get (void); //出队                     数据上的操作
    int empty (void); //判队列是否空
    };
    //以下是函数定义,略去
```

对象将数据和操作封装起来,使外界无法了解其内部细节以及是如何实现的,从而体现了很好的信息隐藏特性。因此无论是完善扩充对象的功能,还是修改对象的实现,其影响仅局限于对象内部,不会影响外界。这就大大增强了软件的易维护性和易于构造软件。

事物抽取其共性可以划分为类。实际上具有相同特性的对象也可归为一个对象类。软件设计中定义的是对象类(或称为类 Class),如前面给出的是队列类(Class 是类定义符)定义。而对象则是对象类一个具体实现的示例(Instance)。如队列 A、队列 B 这两个对象是队列对象类的实现,它们的队列大小和首、尾指针可以不同。

那么 Windows NT 中定义了多少种对象类? 表 12.1 列出 NT 执行体中定义的对象类(读者不妨把对象类理解为资源类)。

表 12.1 执行体对象类

对 象 类	定 义 于	描 述
进程	进程管理程序	略(参见 12.5.2 节)
线程	进程管理程序	进程一个可执行项
区域	主存管理程序	共享主存的一个区域
文件	I/O 管理程序	一个打开文件或 I/O 设备的示例
事件	执行体支持服务	系统事件已发生的通告
事件对	执行体支持服务	通告已把一个信息拷贝给 Win32 服务或反向拷贝已发生
信号量	执行体支持服务	能使用资源的线程数计数器
时间器	执行体支持服务	记录已用时间的计数器
对象目录	对象管理程序	一个对象名管理的方法
简要表	执行体支持服务	测量执行时间分布的一种机制
符号连接	对象管理程序	间接访问对象名的方法
关键字	配置管理程序	数据库中访问记录的索引
端口	本地过程调用程序	进程间传递消息的终点
存取令牌	安全子系统	登录用户安全信息的 ID
多用户终端程序	执行体支持服务	对 Win32 和 OS/2 环境提供互斥

对象类是描述资源类型的,由表 12.1 中可以看到在 Windows NT 中的资源概念是广义的,资源类型是很广泛的。

NT 执行体实现两种对象:

(1) 执行体对象:由执行体的各组成部件实现的对象(参看表 12.1 中对象类定义的部件),能被子系统或 NT 执行体创建和修改,其对象类列于表 12.1 中。

(2) 内核对象:由内核实现的一个更基本的对象集合,称为控制对象集合,包括:内核过程对象、异步过程调用对象、延迟过程调用对象、中断对象、电源通知对象、电源状态对象、调度程序对象等。这些对象对用户态的进程和线程来说是不可见的,它们仅在 NT 执行体内创建和使用。内核对象提供了仅能由操作系统的内核来完成的基本功能(如改变系统时间表的能力等)。很多执行体对象包含了一个或多个内核对象。

除此之外,用户进程和子系统也可以创建它们所需的对象类(如窗口,菜单等)。

在 Windows NT 中每个对象有两部分:

(1) 对象头:其中包含的信息列于表 12.2,其中的信息是由对象管理程序控制的,并用来管理对象。

表 12.2 标准对象头属性

对象名	使对象为共享进程可见
对象目录	提供对象名层次结构
安全描述体	决定谁能使用该对象和对它做什么
配额帐	列出进程打开对象句柄被征收的资源帐
打开句柄计数器	记录该对象句柄被打开的次数
打开句柄数据库	列出打开该对象句柄的进程
永久/暂时状态	该对象不用时可否被删
内核/用户模式	指出该对象在用户态是否可行
对象类型指针	指向对象类型

(2) 对象体:由执行体各组成部件控制自己创建的对象的对象体,对象体的格式和内容随对象类而不同。对象体中列出的各对象类的属性如表 12.3 所示。

表 12.3 对象类属性

对象类名	
存取类型	指一个线程可请求的存取类型(读、写、终止、挂起)
同步能力	线程是否能等待该类
可分页/不可分页	该类对象可否被换出主存
方法	对象管理程序自动调用的一个或多个程序过程

12.5.2 进程

在 Windows NT 中进程被定义为表示操作系统所要做的工作,是操作系统用于组织其必须完成的诸项工作的一种手段。NT 中的进程由以下四个部分组成:

(1) 一个可执行的程序:它定义了初始代码和数据。

(2) 一个私用地址空间: 即进程的虚拟地址空间。

(3) 系统资源: 由操作系统分给进程, 并且是进程执行时所必须的一个资源的集合。由于资源是用对象表示的, 所以进程的资源集用局限于进程的对象表来描述。

(4) 至少有一个执行线程。

进程的概念是大家所熟悉的, 因此我们主要研究 NT 的进程概念与传统操作系统有何不同。其主要不同点如下:

(1) 进程是作为对象来实现的(参见表 12.1), 因此从广义角度来说, 进程也是可共享的资源(多个用户进程可共享服务器进程提供的服务)。NT 定义了一个进程对象类, 表 12.4 列出了进程对象类的对象体所包含的属性。它定义了进程对象的数据及其属性和施加于其上的操作(服务)。但是细心的读者可能会发现, 描述进程组成的两个主要部分——进程地址空间和局限于进程的对象表未包括在表 12.4 中。其原因是这两部分是附属的, 是用户态进程不能修改的, 不可见的。

表 12.4 进程对象类

对象类	进　　程
对象体属性	进程 ID 存取令牌 基本优先级 默认处理器族 配额限制 执行时间 I/O 记数器 VM 操作记数器 异常情况/调试端口 退出状态
服务	创建进程 打开进程 查询进程信息 置进程信息 当前进程信息 终止进程 分配/释放虚拟主存 读/写虚拟主存 保护虚拟主存 锁定/解锁虚拟主存 查询虚拟主存 刷新虚拟主存

表 12.4 中的对象体属性说明如下:

进程 ID: 系统中进程的唯一标识;

存取令牌: 包含该进程所对应的已登录用户的安全信息的对象;

基本优先级: 该进程的线程的基线执行优先级;

默认处理器族: 进程的线程所能运行的默认处理器族;

配额限制: 一个用户进程所能使用的分页和非分页的系统主存, 页面调度文件空间和处理器时间的最大数量;

执行时间: 进程的所有线程的执行时间总量;

I/O 记数器: 记录其所有线程所执行的 I/O 操作数和种类的变量;

VM 操作记数器: 记录其所有线程所进行的虚拟存储操作数量和种类的变量;

异常情况/调试端口: 进程间通信的信道。线程发生异常时, 进程管理程序将向信道发消息。

退出状态: 进程终止原因。

读者要分清的是, 所有进程都是对象, 但是对象不一定就是进程。

(2) NT 进程要求一个独特的组成成分——至少一个执行线程。这可以说是一个新概念, 在传统操作系统中是没有的。详细情况在 12.5.3 节中讨论。

(3) NT 进程的组成中没有进程控制块 PCB。那么有关进程的信息登记在哪里呢? 在进程对象的对象体中以及局限于进程的对象表中等处都有进程的信息。

(4) 传统操作系统中进程是竞争处理机、参预处理机调度的基本单位。但在 NT 中, 调度和执行的基本单位是线程而不是进程。所以对进程也就没有通常意义上的状态变化的划分(不是进程没有某种状态的变化, 而是没有必要去对它进行区分)。

(5) 一个 NT 进程可以有多个线程在其地址空间内执行。

(6) 进程是由进程创建的。每当用户的应用程序启动时, 相应的环境子系统进程调用执行体的进程管理程序为之建立一个进程, 并返回一个句柄。然后进程管理程序又调用对象管理程序为之建立一个进程对象。

当系统初启动时, 系统为每个环境子系统建立一个服务器进程。

(7) 进程管理程序不维护进程的父/子或其它关系。

(8) 进程和线程都具有内含的同步机制。

12.5.3 线程

线程有这样一些定义: 进程内的一个执行单元; 进程内的一个可调度实体。

但是怎么理解这两个定义的确切含义呢? 如果把进程理解为在逻辑上表示了操作系统所做的作业, 那么线程表示完成该作业的许多可能的子任务之一。实际上, 在用户的应用活动中, 多任务情况经常出现。例如, 一个用户使用屏幕编辑程序在进行某项目的开发工作, 操作系统将这个编辑程序的调用表示为一个进程。假设在编辑过程中用户要求搞清某数据的组成及其属性, 从而查找项目的数据字典, 这是一个子任务; 周期性地保存一个编辑文档, 这又是一个子任务; 编辑程序本身也是个子任务; ……, 这些子任务均可表示为编辑进程中独立的线程并且它们可以并发地进行操作。

一个线程有以下四个基本组成部分:

(1) 一个唯一的标识符, 称之为客户 ID。

(2) 描述处理器状态的一组状态寄存器的内容(相当于“老程序状态字”的意思)。

(3) 两个栈,分别用于用户态和核心态下执行时使用。

(4) 一个私用存储器。

问题是 Windows NT 为什么要推出线程这一概念,到底有什么好处? 难道不能用多进程来实现并行性吗? 问题正是在这里,因为用两个进程来实现并行性并不总是有效的。以 UNIX 为例,当一个进程创建一个子进程时,系统必须将父进程地址空间的所有内容拷贝到新进程的地址空间中去。这对大地址空间来说,这操作很费时,更何况两进程还需建立共享数据。如果用多线程进程来实现并行性要有利得多,因为这些线程共享进程的同一地址空间、对象句柄以及其它资源,所以没有用进程来实现并行性所存在的缺点,此外,还有以下优点:

(1) 通过线程可方便而有效地实现并行性,进程可创建多个线程来执行同一程序的不同部分,如一个编译进程可创建预处理线程和编译线程这样两个线程。

(2) 创建线程比创建进程要快,而且只需很少的开销。因为所有线程除栈和寄存器内容外共享同一主存,不需特殊的数据传送机制(如发消息),一个线程只需简单地把输出写入主存,另一线程可以读出作为输入。而且进程的资源线程都可用。

(3) 创建多线程进程,对多个客户同时提出服务请求时的回答也十分有利。因为服务器程序只被装入主存一次,就可使每个客户的服务请求分别由一个独立的服务器线程,通过执行适当的服务器功能,并行地为客户进行处理。

NT 的线程有些什么特点呢?

(1) 线程也是作为对象来实现。NT 定义了线程对象类如表 12.5 所示;

(2) 每个 NT 进程创建时只有一个线程,需要时这个线程可以创建其它线程;

(3) 线程调用系统服务是采用陷阱(trap)方式(参阅 12.6.2 节);

表 12.5 线程对象类

对象类	线 程
对象体属性	客户 ID 线程描述表 动态优先级 基本优先级 线程处理器族 线程执行时间 报警状态 挂起记数 模仿令牌 终止端口 线程退出状态
服务	创建线程 打开线程 查询线程信息 当前线程

续表

对象类	线　　程
服务	终止线程 取得描述表 置描述表 挂起 重新开始 报警线程 检测线程警报 寄存器终止端口

(4) 线程是调度的基本单位,线程之间竞争处理机。为防止一个线程独占处理机,采用可抢占的优先级调度算法(参阅 12.6.1 节);

(5) 线程在它的生命期间有六种状态的变化,每一时刻处于六种状态之一(参阅 12.6.1 节);

(6) 由于 NT 的调度程序只在线程(而不是进程)中挑选合适的线程到处理机上运行,所以每个进程在可以执行前,至少必须具有一个线程。NT 进程只有在它的一个线程被调度执行才被激活;

(7) 用户态进程的线程主要在用户态处理器模式下运行。当用户态线程通过系统陷阱(trap)调用系统服务访问操作系统时,处理器将其执行模式切换到核心态。操作系统在完成服务后,将控制权交还用户程序时,再将线程切换回用户态。

12.5.4　对象、进程和线程之间的关系

对象、进程和线程是组织和构造 Windows NT 操作系统的三个基元成分。它们之间互相交叉的关系,往往使人不易搞清楚。对象是一个抽象的数据结构,NT 用它来描述资源。从这个意义上来说,对象是构成操作系统的三个基元成分中非活动的成分。可是它经常使人产生它也是活动的幻觉! 例如下面的说法是完全正确的,即"NT 将所有的资源都看成对象,这就使得 NT 的内部服务大部分是对象服务"。"对象服务"的提法给人一种对象在活动的感觉,而这一提法恰巧是由于有些对象(如进程,线程)是实实在在的活动成分,需要为之提供服务。而且对象作为抽象数据而封装在其内部的操作函数所提供的操作,也给人活动成分的感觉。但是从操作系统这一角度来认识,对象是构成操作系统的非活动成分。认清这一点才不至于把对象混同于进程、线程,难分彼此。

进程和线程则是构成操作系统的两个活动成分,这二者之间的关系更是难以搞清的。让我们来看看它们的关系:

(1) 线程是进程的一个组成部分;

(2) 进程的多个线程都在进程的地址空间活动;

(3) 资源分配的对象是进程。换句话说,资源是分给进程的,而不是分给线程的,并且系统还为分给进程的资源规定了配额;

(4) 调度的基本单位是线程。也就是说,处理机是分给线程的,真正在处理机上执行

的是线程(从某种意义上说,进程不干活,它只通过线程工作)。线程在执行中需要资源时,系统分给它,但从进程的配额中扣除;

(5) 线程在执行过程中,需要协作同步。在不同进程的线程间的同步要用消息通信办法。消息通信过程中要用到进程的资源——端口(端口也是对象,端口中有消息队列)。因此通信的是线程,用的消息队列是进程的资源(服务器进程还可创建多端口,以便多线程通信)。

列出以上关系供读者理解进程和线程之间关系。它们之间的关系可作这样比喻:进程相当于董事长,线程相当于部门执行经理。也有人认为与通常操作系统中的作业与进程关系类似,但似不甚恰当。

12.5.5 进程管理程序

NT 执行体的进程管理程序的任务是创建与终止进程和线程,挂起线程的执行,存储和检索 NT 进程和线程的信息。下面主要考察其创建进程和线程的功能。

众所周知, Windows NT 是支持多种操作系统的运行环境。这一功能的实现是通过环境子系统,而环境子系统要实现这功能需要完成两个主要任务:一个是模拟子系统的客户应用程序的运行环境;另一重要任务是实现客户应用程序所要求的、适应原环境(如 UNIX)的进程结构。这既包括进程本身的结构,也包括进程之间的关系(如 NT 所不维护的进程父子关系)。

所以进程结构实际上有 NT 本机进程和各环境中进程结构之分。NT 执行体的进程管理程序创建和维护的是 NT 本机进程结构。由 NT 本机进程转换到环境子系统的进程结构,这是由环境子系统来实现的。

于是在 Windows NT 中进程和线程的创建过程如下:

(1) 客户进程用创建进程原语(如 fork ( ): POSIX 子系统或 Createprocess( ): Win32 子系统)创建进程;

(2) 客户进程通过发消息给相应的服务器进程(某环境子系统);

(3) 服务器进程调用 NT 执行体的进程管理程序为之创建一个 NT 本机进程。在此过程中,进程管理程序调用 NT 执行体的对象管理程序为该进程创建一个进程对象;

(4) 进程创建后(请注意, NT 把进程创建视为对象创建,仅此而已),进程管理程序返回一个句柄给进程对象;

(5) 环境子系统取得该句柄,并生成客户应用程序期望的适合本环境的返回值;

(6) 环境子系统又调用 NT 的进程管理程序为已创建的新进程创建一个线程。

总的说来, NT 执行体中的进程只不过是对象管理程序所创建和删除的对象。从这个意义上说,进程管理程序的主要工作是定义了存放在进程对象的对象体中的属性(参看表 12.4),并提供检索和改变这些属性的系统服务。

进程管理程序允许不同的环境子系统以不同的初始属性来创建进程。

在表 12.4 中的服务,多数是不言自明的。其中“ 当前进程 ”服务,允许进程不通过对象服务程序而快速为自己获得一句柄。而“ 终止进程 ”则停止进程和线程,关闭所有对象句柄,删除进程的虚拟地址空间。进程管理程序提供了有关进程和线程的服务。

12.6 内　　核

内核是NT真正的中心，Helen Custer将内核比喻为轮轴，一切围绕内核转。NT内核提供了一组精心定义的操作系统原语和机制。通过使用内核原语，NT执行体可以构成许多更高级的功能。内核执行的主要任务有：

(1) 调度线程的执行；

(2) 当中断和异常发生时，将控制转移到相应的中断和异常处理程序；

(3) 执行低级的多处理器同步；

(4) 在电源失效后，实现系统的恢复过程。

与NT执行体的其它部分不同，内核永远驻留内存，内核的执行是不可被抢占的，并且总运行在核心态。

12.6.1 内核调度程序与线程的状态转换

内核调度程序的主要功能是选择一个适当的线程到处理器执行并进行描述表切换。

引起调度程序重新调度的时机是：

(1) 当线程变为就绪时；

(2) 当线程时间段(片)结束或线程终止时；

(3) 当调度程序或执行体改变线程优先级时；

(4) 当执行体或应用程序改变正在运行线程的处理器族(多处理器中的一个子集)时。

线程的描述表和描述表切换的过程随处理器结构不同而异。典型的描述表切换要求保存原执行线程的现场数据并装入新执行线程的现场数据，这些数据是指：

(1) 程序记数器(PC)；

(2) 处理器状态寄存器(PSW)；

(3) 其它寄存器内容；

(4) 用户和内核栈指针(线程的两个栈)；

(5) 线程运行的地址空间指针(进程页表目录)。

一个线程在生命期的状态是变化的，它在任一时刻的状态是以下六种状态之一(参看图12.3)。当一个线程创建一个新的线程时，这个线程开始了它的“生命期”，一旦被初始化，便将经历以下六种状态：

(1) 就绪状态：线程已具备执行的条件等待CPU执行。调度程序只从就绪线程池中挑选线程进入备用状态。

(2) 备用状态：被调度程序选定为某一特定处理器的下一个执行对象。当条件合适时调度程序为该线程执行描述表切换。系统中每个处理器上只能有一个处于备用状态的线程。

(3) 运行状态：一旦调度程序对线程执行完描述表切换，线程进入运行状态。

(4) 等待状态：以下情况线程进入等待状态。

① 线程等待同步对象,同步它的执行;

② 因 I/O 系统而等待;

③ 环境子系统导致线程将自己挂起。

当线程的等待状态结束(如同步对象已成为有信号状态),就变为就绪状态。

(5) 转换状态:如果线程已准备好执行,但由于资源成为不可用(如其内核栈所在页被换出了主存)从而成为转换状态。当资源成为可用时,线程便由转换状态成为就绪状态。

(6) 终止状态:线程完成了它的执行。

线程六种状态的变化及其变化原因如图 12.3 所示。

图 12.3　线程状态

内核的线程调度程序是采用可抢占的动态优先级调度算法,调度程序按线程的优先级调度线程的执行顺序,先调度高优先级的线程。

最初线程从创建它的进程处得到优先级。线程可将继承的进程基本优先级改为稍高或稍低的优先级。线程执行过程中优先级可发生变化。调度程序调度时的主要数据库是多优先级就绪队列(类似于 5.5.7 节多级反馈队列)如图 12.4 所示。NT 执行体支持 32 个优先级,将它们分为两类:实时优先级和可变优先级。实时线程的优先级从 16 到 31,是实时程序所用的高优先级线程。线程中大多数线程属于可变优先级线程,其优先级从 1 到 15(优先级 0 为系统所保留)。

调度程序从最高优先级队列开始向下找,直到找到一个线程为止,高优先级队列为空后才再向下找。当线程执行完一个完整的时间段后,被中断而抢占处理器,而被抢占的线程的优先级降低一级而进入下一就绪队列,直至到该线程的基本优先级。而一个线程从等待状态变为就绪时要提高优先级,提高的幅度与等待的事件有关。如等待键盘输入所提高的幅度要大于等待磁盘 I/O。这样,交互式线程处于高优先级;I/O 工作性质的线程处于中间优先级;计算性工作的线程处于低优先级;当系统中没有任何事件时,内核提供一个总在执行的线程,称为空闲线程,优先级最低。

图 12.4　多级就绪队列

12.6.2　中断和异常处理

NT 中的中断主要是由硬件引起的,是随机发生的异步事件。

而异常是某一特别指令执行的结果,是同步情况,如主存存取错、被零除等。

1. 陷阱处理程序

当中断和异常发生并被系统捕捉到后,系统将执行线程从用户态切换到核心态,将处理器控制权交给操作系统中存放在主存固定地址的陷阱处理程序。而陷阱处理程序首先要保存执行线程被中断的断点现场,在此过程中要屏蔽中断;其次它确定所发生的情况,并将控制转交给相应的处理程序,如图 12.5 所示。

图 12.5　中断和异常调度

NT 的中断类型和中断优先级如表 12.6 所示。

表 12.6　中断优先级

中断请求优先级	中断类型
高级	机器检查和总线错
电源级	电源故障
处理器间	从另一处理器来的请求
中断级	
时钟级	间隔时钟
设备级 n	最高 I/O 设备级
⋮	⋮
设备级 1	最低 I/O 设备级
调度/DPC 级	线程调度和延迟过程调用(DPC)处理
APC 级	异步过程调用(APC)处理
低级	普通线程执行

NT 的中断服务也是使用“中断调度表”以查找处理特定中断的中断处理程序。

而 NT 的“异常”与一般所说的“程序中断事件”类似,“异常”是指以下情况:

主存存取错　　整数溢出
浮点被零除　　调试程序断点
页面读取错　　非法指令
调试单步执行　　整数被零除
浮点上溢/下溢　　数据类型错
特权指令　　页保护错
超出分页文件限额　　浮点保留的操作数

系统服务调用类似于 IBM 的访管指令 SVC 和 MIPS 的 Syscall 以及 Intelx86 处理器上的 int2Eh,当用户态线程采用陷阱(trap) 方法调用某系统服务时,内核用“系统服务调度表”查找该系统服务处理程序,并进行相应处理。

12.6.3　内核的同步与互斥机制——多处理器间的同步

内核中涉及许多对全局数据库的修改,众所周知,这些修改应互斥地进行,也就是说要互斥地执行临界段[如对内核调度程序数据库和延迟过程调用队列(又称 DPC 队列)修改的代码]。那么 NT 在内核如何执行互斥呢?

第一种方法是提高临界段代码执行的中断优先级。这种方法在 UNIX 中也使用,它是在单机系统中有效地实行互斥的一种方法。因为在传统操作系统中,打断进程对临界段代码的执行只有中断请求。中断一旦被接受,系统就有可能调用其它进程进入临界段并修改此全局数据库。所以用提高临界段中断优先级方法就可以屏蔽了其它中断,保证了临

界段的执行不被打断,从而实现了互斥。

在 NT 中具体执行方法是这样的:在内核使用全局临界资源之前,内核暂时屏蔽那些也要使用该资源的中断,也就是说把该临界段执行时的中断优先级提高到这些潜在中断源中的最高级。换句话说,NT 在临界段执行时并不屏蔽一切中断,只屏蔽有可能也将使用同一临界资源的中断。例如内核"调度/DPC"级中断(NT 的"调度/DPC"是中断级中的第三级、是比较低的中断级)才能使调度程序运用,而只有调度程序才修改调度程序数据基,所以内核中所有使用调度程序的部分把中断优先级提高到"调度/DPC"级别,从而屏蔽了其它的调度/DPC 中断。

第二种方法是使用转锁(spin-lock)。用提高临界段执行的中断优先级方法来实现互斥,在单处理器情况下虽然很有效,但是如果在多处理器情况下,用这样方法是无法保证互斥的。因为在一个处理器上提高中断优先级并不能阻止其它处理器上的中断。所以必须采用别的方法,这方法就是在 4.2.2.1 节中所讲的方法。NT 是用 T-S 指令的方法来实现对临界段执行上锁,并把这种机制称为转锁(spin-lock)。这里要强调的是 T-S 指令在一条指令中实现测试锁变量值并设置锁变量值这样两个操作,这是能在多处理器间实现互斥的关键。

NT 为保证尽快地执行临界段代码,而采取了以下措施:

(1) 占用转锁的线程不被抢占处理器。

(2) 临界段必须迅速存取临界资源。在临界段中不得与其它代码进行复杂的交互作用。

(3) 临界段代码所在的页不能被换出主存;不能涉及可分页的数据(指可换出主存的页面);不能调用外部过程(包括系统服务);不能产生中断或异常。

我们知道信号量上的 P,V 操作也可以实现互斥(信号量初值为 1),但在多处理模式下,这种 P,V 操作对互斥是无效的、不能实现的。

12.7 虚拟存储管理

12.7.1 进程的虚拟地址空间

Windows NT 的虚拟存储管理程序(Virtual Memory(VM) Manager)是 NT 执行体的主要组成部件之一,它是 NT 的基本存储管理系统。NT 所支持的面向不同应用环境(支持多操作系统应用环境)的各环境子系统中的存储管理也都是基于 NT 的 VM 虚拟存储管理系统的。

VM 实现了一个复杂而有效的虚拟存储系统,它为每个进程提拱了一个很大的虚拟地址空间(4GB)。为做到这一点,NT 坚决抛弃了所有基于 Intel 芯片的早期个人计算机(从 8086 到 80286)都使用的分段模式所造成的每段 64K 的局限性,因为这个局限使得 32 位计算机难以发挥它的全部性能。同时 NT 的设计者认为,虚拟空间的线性模式远比分段模式更与主存的实际结构(单字节的存储单元的连续集合)相吻合,所以决定采用"请求分页的虚拟存储管理技术"。

Windows NT 运行在 32 位的 386 以上的微型机上,所以每一个 NT 的进程都有 4GB(2^{32}个字节,称为 40 千兆)大的虚拟地址空间,其地址空间的使用情况如图 12.6 所示。由图可知,进程 4GB 的地址空间被等分成两部分。虚拟地址空间高地址的 2GB 空间保留给系统使用,而低地址的 2GB 空间才是用户存储区,可被用户态和核心态的线程访问 。

系统区又分为三部分,最上部的固定页面区(称为非页交换区)用以存放永不被换出内存的页面,这些页面中存放系统中需常驻内存的代码(如实现页面调度的代码)。而第二部分称为页交换区,用于存放非常驻内存的系统代码和数据 。第三部分称为直接映射区是比较特殊的,首先这一区域的寻址是由硬件直接变换;其次这些页面常驻内存,永不"失效"(见 12.7.4 节的页架状态)。因此存取这一区域的数据特别快。用以存放 Windows NT 内核中需频繁使用、响应速度快的那些代码,如调度线程执行的代码。

图 12.6　虚拟地址空间

NT 中使用的页面尺寸为 4KB(2^{12})大小。

12.7.2　NT 虚拟分页的地址变换机构

虚拟分页技术的实现应包括地址变换机构与页面调度策略两方面,本节集中研究 NT 使用的地址变换机构。

NT 的地址变换机构不同于传统的页面地址变换机构,它是采用一种称为两级页表结构的技术,这也是比较特殊的,具体实现如图 12.7 所示。第一级表叫页目录,每个进程

图 12.7　二级页表地址变换机构

一个页目录。每个页目录均包含 1024 个表目,每个页目录表目指出其第二级页表所在的页架号(所以称页表地址)。第二级页表中也包含有 1024 个表目,每个表目大小是 4 个字

节,因此第二级页表均为 4KB 大小(恰好占一页空间)。这些特点从图 12.7 中的虚拟地址结构中完全可以看出。在系统中有控制寄存器指出进程的页目录地址——即页目录所在的页架号。地址变换过程是这样的,当给出虚拟地址后,按图示被自动分为三部分:目录位移(10 位,由第 22 位到 31 位);页表位移(10 位,由第 12 位到 21 位);页内位移(12 位,由第 0 位到 11 位)。地址变换机构将寄存器中的页目录地址与目录位移(左移二位,因表目长为 4 个字节)拼成页目录中表目所在地址,该表目中包含有其页表地址(页表所在页架号),再将此页表地址与页表位移(同理左移二位)拼成页表中目标表目地址,其中包含有该页所在的页架号。最后将此页架号与页内位移拼成主存绝对地址,从而访问主存。

至此,读者心中必然有个问题,Windows NT 为什么要采用两级页表结构?

道理很简单,因为每个进程的虚拟地址空间太大了,它的 32 位地址使得一个进程有 2^{32} 个可能的虚地址。每页大小为 4KB(2^{12} 个字节),那么每个进程的地址空间可有 1048 576页(即有 2^{20} 个页面)。这也就是说每个进程的页表可有 2^{20} 个表目,每个表目占 4 个字节,那么一个进程的全部页表就可能要占 1024 个页面(2^{20} 乘 2^{2} 再除以页面大小 2^{12})。要知道每个进程都有独立的地址空间,这将是多么大的主存开销。所以 Windows NT 为避免把所有主存都消耗在页表上,它的虚拟存储管理程序不把这些页表放在主存,而是根据需要把这些页表换入、换出主存。

采用两级页表结构带来一个主存访问速度变慢的问题。因为每从主存访问一个数据要三次访问主存才行。首先查页目录(在主存)要访问一次主存;其次查页表(也在主存)又访问一次主存;最后才从主存中存取数据。但是实际上 Windows NT 的访问主存速度不但不慢,相对来说还比较快。这是为什么?因为它采取了两个有力的措施:

(1) 使用快表:即使用高速相关存储器来存放经常使用的页表表目[NT 称快表为变换查找缓冲区(TLB)];

(2) 使用高速缓冲存储器:在微处理器和主存间设置 32KB 或 64KB 的高速缓冲存储器,大部分的指令和数据取自高速缓存(命中率 98%)。所以存取数据和指令速度相当高,达到与处理器速度完全相匹配。

12.7.3 页面调度策略和工作集

12.7.3.1 页面调度策略

页面调度策略包括取页策略、置页策略和淘汰(置换)策略。

取页策略分“提前取页”和按进程需要的“请求取页”两种策略。而 Windows NT 是采用既按进程需要时的请求取页,又采取集群方法把一些页面提前装入主存。集群方法提前取页的含意是,当一个线程发生缺页时,不但把它所需的页装入主存,而且把该页附近的一些页也一起装入主存。这样做的主要根据就是程序行为的局部特性,因此装入一簇虚页会减少缺页的数量。尤其在一个线程开始执行时,请求取页会造成频繁缺页,降低系统性能。而集群方法提前取页使缺页情况大大减少。

置页策略是指把虚页放在哪个页架。这问题在线性存储结构中比较简单,只要找到一个未分配的页架即可。

置换策略或淘汰策略有以下两个要点:

(1) 采用局部置换策略,它为每一个进程分配一个固定数量的页面(但可动态地调整这个数量)。发生缺页时,从本进程的范围内替换页。

(2) 使用的置换算法是 FIFO,即把驻留时间最长的页面淘汰出去。采用 FIFO 算法的主要出发点是该算法实现起来最简单。

12.7.3.2 工作集

Windows NT 的虚存管理程序(VM)为每一个进程分给固定数量的页面,并且可动态地调整这个数量。那么这个数量如何确定?又如何动态调整呢?这个数量就用每个进程的工作集来确定,并且根据主存的负荷和进程的缺页情况动态地调整其工作集。具体作法是这样的:一个进程在创建时就指定了一个最小工作集,该工作集大小是保证进程运行时应在主存中的页面的数量。但是在主存负荷不太大(页面不太满)时,虚存管理程序(VM)还允许进程拥有尽可能多的页面作为其最大工作集。当主存负荷发生变化时,如空闲页架不多了,虚存管理程序使用"自动调整工作集"的技术来增加主存中可用的自由页架。方法是检查主存中的每一个进程,将它当前工作集大小与其最小工作集进行比较。如大于最小值,则从它们的工作集中移去一些页面作为主存自由页面,并可为其它进程所使用。若主存自由页面仍然太小,则继续这样做,直到每个进程的工作集都达到最小值。当然本方法要求 VM 跟踪每个进程当前的页面。

当每个工作集已达最小值的进程,虚存管理程序 VM 跟踪该进程的缺页数量,根据内存中自由页面数量适当增加其工作集大小。

12.7.4 页架状态和页架数据库

主存中页架可有以下六种状态之一:

(1) 有效状态:某进程正使用该页。

(2) 清零状态:页架处于空闲并已被清零[①]。

(3) 空闲状态:页架空闲但尚未被清零。

(4) 备用状态:页架从工作集中移出了,页表表目已将其记为无效状态,但有一个过渡标志。当进程缺页并且需该页内容,则不必从盘上读入主存,重新将该页分给它,并置为有效。

(5) 修改状态:类似于备用状态,只是该页中的数据已被修改(写操作)过,并且该页上的内容尚未写入磁盘中的该虚页中去。

(6) 坏页状态:该页架产生了奇偶校验错或其它硬件错,不能再用。

在传统的操作系统中,为虚页分配页架时要查找自由页架表,该表是主要数据库之一。而在本系统中是将所有相同状态(除有效状态外)的页架均用链指针连接在一起而形成了空闲表、清零表、备用表、修改表和坏页表并作为页架分配数据库。如某进程需一个清零页架,则从清零表中取出第一页。如该表为空,则从空闲表中取来第一页清零后分给之。如果二类表均为空,则使用备用表中的页。当以上三类表中的页架数低于最小允许值时,则把修改状态的页中内容写回盘后将其移入备用表队列。如果修改表中的页面也太少,则

① 清零是出于页面中信息的安全性,防止已移出进程地址空间的页面中的信息被窃取。

就调整各进程工作集的大小。

VM 为了提高性能,采用了惰性(lazy)技术,尽可能地避免不必要的费时的操作,只有当需要时才执行这些操作。如一个已修改过的页面移出进程工作集之后,并不立即写回盘,而是先放入备用表中以备该进程再次需要,直到必要时才写回盘。

12.7.5 共享主存——段对象、视口和映象文件

存储管理的一个重要功能就是允许一些进程在需要时能有效地共享主存。通常共享主存有两种情况:

(1) 两个(或多个)进程共享同一数据文件;

(2) 两个(或多个)进程共享同一主存缓冲区以便于进行数据交换或通信。

所有基于分页的操作系统中,解决主存共享的方法都是把这些进程的页表中相关的虚页使其指向同样的页架(这些页架中存放着共享文件或缓冲区)。

问题是如何使共享进程知道共享的数据文件和主存缓冲区在那些页架中? 对于共享数据文件这种情况,传统操作系统的文件系统提供活动文件表,可供进程查找共享文件在主存中的有关信息。对于共享主存缓冲区这种情况,缓冲区位置只好用某种进程通信方式通知对方。而 Windows NT 的解决办法却是不同的。在 NT 的 Win 32 中是把它们作为被保护的‘对象’来实现的——这就是用‘段对象’作为文件映象对象。所谓段对象代表一个可由两个或更多进程共享的主存区域。一个进程中的一个线程可创建一个段对象,并给它起个名字——对象名,以便其它进程中的线程能打开这个段对象的句柄 。打开段对象的一个句柄后,线程就能把这个段或该段中的若干部分映象到它们自己的虚拟地址空间中去,实现了共享(查找对象名以得到对象的信息由对象管理程序负责)。

NT 允许一个段对象可以相当大(甚至大到几千个页面)。有时某进程只需要段对象中的一部分。进程所需的那部分叫做该段的一个视口(view),一个视口提供了进入这个共享的主存区的一个窗口,不同进程可以映象一个段对象的不同的视口。

用映象一个段对象的一些视口可使进程访问很大的一些主存,否则该进程也许就没有足够的虚拟地址空间(进程虚拟地址空间为 2GB,而段对象可以远大于此)来映象它们。例如某公司可能有一个很大的数据库含有职工的数据,该数据库程序创建一个段对象来包含整个职工数据库。当一个用户进程查询该数据库时,该数据库程序就把该段对象的视口映象到进程的虚拟地址空间,从中获得数据。查完该视口,然后映象该段对象的另一视口,……。实际上 NT 通过段对象和视口的方法每次给一个区域‘开一个窗口’,从而可从数据库的每一部分获得数据,而又不会超出进程虚拟地址空间。NT 虚存管理程序通过段对象的不同视口的方法,就能使进程像在主存中访问一个大数组(视口类似于数组元素子集)那样存取该文件。这被称为映象文件 I/O 活动。当发生缺页时,VM 管理程序就自动把该页放入主存,可进行像正常的页面调度那样操作。NT 执行体使用映象文件把可执行图象装入主存,这对多媒体应用情况十分有利。

Win32 应用程序可使用映象文件方便地完成随机输入/输出到大文件。方法是该应用程序创建 Win 32 映象文件对象(它对应于一个 NT 的段对象)以包含该文件,然后读写到文件中的随机主存区,VM 管理程序在文件所需部分自动分页。

段对象同其它对象一样,由对象管理程序创建对象头部,并进行管理。由虚拟存储管理程序定义对象体如图 12.8 中所示。其中基本段/非基本段属性指明为基本段时,该段在所有共享进程中的虚拟地址应相同(参见 8.2.5 节)。

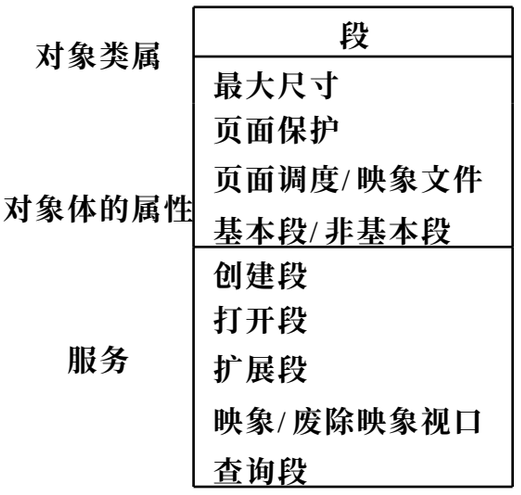


图 12.8　段对象

由上可知,NT 实现共享主存的方法是很有特点的:

(1) 用创建段对象的方法来实现主存共享。对象是 NT 用以表示共享资源的有效方法。

(2) 段对象允许大于进程的虚拟地址空间(2^{32}个字节)可达 2^{64}个字节。NT 还允许用视口的方法来一部分一部分的进行段对象访问。映象文件和段对象以及视口的方法提高了系统性能。

12.8　输入输出(I/O)系统

传统上一直认为输入输出(或设备管理)系统是操作系统设计中最为逊色的领域。确实,负责具体进行输入输出的设备是如此的五花八门、种类繁杂,小到传感器、鼠标,大到磁盘、磁带、绘图仪、甚至航天器、人造卫星。它们的性能全然不同,以常规的 I/O 设备而言,按其各种特性,有字符设备与块设备不同;有独享设备与共享设备不同;……,总之从数据格式、数据组织到使用方法都有很大差别。因此过去的操作系统设计者们感到,面对如此众多的特定方法,难以规范化。

可是 Windows NT 操作系统的设计者们采取并开辟了与其它操作系统全然不同的设计思路:传统操作系统设计中一直致力于研究和确定操作系统中各种事物的不同特性和不同方面。如存储管理、处理机管理、设备管理以及各种设备的管理等都是强调以特殊的方法和机制来处理,这就使系统十分复杂,界面全然不同,交互十分不便。Windows NT 采用和强调了软件工程中抽象的原则,在设计中全力找出各种事物的共同性,用一致的模型、方法和界面来规范化,如用客户/服务器模型规范各用户进程之间的关系。尤其突出的是,在寻找事物的共同性思想指导下,建立起了广义的资源概念,并统一地用对象模型来描述并规范化(当然也在共同性前提下注意其个性),这就使系统复杂性降低,界面和操作一致,交互容易、方便而有效。同样,NT 在输入输出系统设计方面建立起了一个统一一致的高层界面——I/O 设备的虚拟界面——即把所有读写数据看成直接送往虚拟文件的字节流。

12.8.1　输入输出(I/O)系统的结构

12.4 节中已指出 I/O 系统是个层次结构模型,其结构如图 12.1 和 12.2 所示。从图 12.1 中可以看出,NT 的 I/O 系统由一组负责处理各种设施的输入和输出部件构成。这

些构成 I/O 系统层次关系的部件有:

(1) I/O 管理程序(I/O Manager):它实现与设备无关的输入输出,并建立 NT 执行体 I/O 的模型,它并不进行实际的 I/O 处理。它的工作是:

① 建立一个代表 I/O 操作的 I/O 请求包(IRP,I/O Request Packet),将 IRP 传给适当的驱动程序,并在 I/O 操作完成后处理其结果,撤消 IRP;

② I/O 管理程序允许一个驱动程序通过它调用其它驱动程序。如文件驱动程序调用磁盘驱动程序;

③ 管理 I/O 请求要用的缓冲区。

(2) 文件系统:它们也当成设备驱动程序看待,接收面向文件的 I/O 请求,将之翻译成针对某特定设备的 I/O 请求。

NT 支持多种文件系统,包括 DOS 的 FAT 文件系统;OS/2 的高性能文件系统(HPFS);CD-ROM 文件系统(CDFS)以及 NT 文件系统(NTFS)。NTFS 具有许多新的特性:具有很强的文件系统恢复功能;可以处理巨大的存储媒体(可达 2^{64} 个字节);具有安全特性;文件名采用 Unicode 国际代码表示;支持 POSIX 操作系统环境中如文件的连接,大小写敏感文件名;可用动态连接库(DLL)对这些文件系统进行装入和卸出并适宜于将来的扩展。

(3) 缓冲存储管理器(Cache Manager):管理高速缓冲存储器,并在主存、磁盘和高速缓存之间进行信息块的调度,以改善系统性能和系统文件 I/O 的功能。

(4) 设备驱动器(Device Driver):它们直接对特定物理设备或网络进行输入输出的操作。设备驱动程序接收上层传来的 I/O 操作请求包 IRP,执行 IRP 指定的操作,然后将结果返回上层,或者通过 I/O 管理程序将它传给另一个驱动程序作进一步的处理。

(5) 网络重定向器(Network Redirector)和网络服务器(Network Server):它们是文件系统驱动器,通过它们可以访问在 LAN Manager 网络上的文件。重定向器接收对远程文件的请求,并将它们定向到位于其它机器上的 LAN Manager 服务器。

12.8.2 统一的驱动程序模型

12.8.1 节中描述的 I/O 系统是由五个类型的部件组成的。该结构中有四类驱动程序:文件系统、缓冲存储管理器、设备驱动器、网络重定向器。要把这四类功能和特性均不相同的部件组合在一起,并建成一个一体化的统一的系统,必须要进行高层的抽象,以建立一个统一的逻辑模型。

NT 建立起的逻辑模型吸收了 UNIX 对 I/O 定义的观念,认为"所有的读写数据都看成直接送往虚拟文件(UNIX 把设备也看成文件,把文件定义为字符流)的字节流。虚拟文件用文件描述符(参见 10.3.1 节)表示,处理虚拟文件就像处理一个真正的文件一样。由操作系统判定这个虚拟文件是设备、网络、管道(pipe)还是真文件。"由此建立起一个统一的驱动程序模型。它有以下特点:

(1) 所有的驱动程序是统一的结构,用同一方式建立,表现出相同的外貌。

(2) 每一个驱动程序都由标准成套(或组合)部件组成。这些部件有:

一个初始化程序;

一组调度程序;

一个启动 I/O 程序;

一个中断服务程序,处理设备中断;

一个中断服务 DPC 程序;

一个完成例程,当低层驱动程序完成一个 IRP 处理时,通知分层驱动程序;

一个撤消 I/O 例程。如果一个 I/O 操作可撤消,驱动程序可定义一个或多个撤消例程。

一个卸载例程。释放资源以便驱动程序卸出主存;

一个出错记录例程。

(3) 每个驱动程序都是一个独立的部件,可由动态连接库(DLL)动态地装入和卸出操作系统。

(4) I/O 系统是包驱动。所有的 I/O 请求,表示为一个一致的形式,叫做 I/O 请求包(IRP, I/O Request Packet)。每个 IRP 是一个数据结构,用以控制在每一步骤上如何处理 I/O 操作。

(5) IRP 由两部分组成:一个称作头部的固定部分和一个或多个栈存储单元。

头部包含的信息是请求类型和大小;请求同步 I/O 还是异步 I/O;有缓冲区 I/O 的缓冲区指针;请求进展过程中变动的状态信息等。

栈存储单元包含一个功能代码、功能定义的参数(如写调度程序)以及指向调用程序文件对象的指针。

(6) 处理方式由驱动程序接受 IRP ,执行 IRP 指定的操作,并在完成操作后,将 IRP 传回 I/O 管理程序或通过 I/O 管理程序再调用其它驱动程序以求进一步的处理。

12.8.3 异步 I/O 操作和 I/O 请求处理过程

在大多数操作系统中,都是采取输入输出同步操作。什么意思呢? 当用户程序调用一个 I/O 服务时,程序等待传输完成(即等待 I/O 设备在完成数据传输后返回给程序一个状态码)后,立即存取这些数据再进行下面的处理,这称为同步操作。在单用户的机器上,程序等待时,处理器也没有多少事。对现代高速处理器来说,往往因一个 I/O 请求而耽误了几千条代码的执行,这是很可惜的。而异步操作服务是允许应用程序在发出 I/O 请求后,在设备传输数据的同时,应用程序继续可以进行其它工作,如在屏幕上编辑程序或画某个图形等。因此当调用 NT 执行体的 I/O 服务时,开发人员必须选择是同步操作还是异步操作。一般来说,对于快速操作或者可预测时间的操作,使用 I/O 同步操作有效。而对于需要很长时间或无法确定所需时间的操作,采用异步 I/O 是有利的。

I/O 请求的处理过程随采用的是同步操作还是异步操作而略有不同。

同步 I/O 请求的处理过程分为以下三个步骤进行:

(1) 按用户请求,I/O 管理程序为之形成 I/O 请求包 IRP ,并将 I/O 请求包 IRP 送驱动程序[如果 I/O 请求是对大容量的设备,如光盘驱动器、磁盘驱动器、磁带驱动器和网络服务器等,则使用分层处理模式(参见图 12.2)。如果请求是对面向字节的设备,如打印机、键盘、视频监视器和鼠标器等,则可以用单层独立处理。但多层驱动程序比单层驱动

程序使用得更普遍。多层处理时，IRP 首先送文件驱动程序。单层处理时，IRP 送设备驱动程序]，驱动程序启动 I/O 操作(假定此时为单层处理)；

(2) 设备完成 I/O 操作，并发出中断请求，设备驱动程序中的中断处理程序服务于中断；

(3) I/O 管理程序完成 I/O 请求。

异步 I/O 请求的处理是在上述的第一步与第二步之间增加了一步：I/O 管理程序将控制返回发出 I/O 请求的用户调用程序。这样，调用程序可在 I/O 系统处理第二步和第三步的同时继续执行其它的工作。但调用程序为了知道何时数据传输已完成，它必须和第三步的完成相同步。

12.8.4 映象文件 I/O

映象文件 I/O 是指把驻留在盘上的文件看成是虚拟主存中的一部分，程序可把文件作为一个大数组来存取而无需缓冲数据或执行磁盘 I/O。

映象文件 I/O，在执行图象操作以及大量文件 I/O 时，能提高执行速度。因为映象文件 I/O 是向主存中写入，这自然要比写盘快得多，而且 VM 虚存管理程序还优化了磁盘的存取。

12.9 Windows NT 的内装网络

由于 Windows NT 内装网络具有非凡特色，故单辟一节对这方面加以介绍，作为对 12.8 节中网络功能的补充。

12.9.1 Windows NT 的内装网络的特色

特色之一：真正的网络操作系统。

网络操作系统概念提出已有近 20 年的历史了，但至今，网络操作系统几乎全都是在一个多任务的操作系统上附加一个网络软件层。例如 SUN 公司在基于 UNIX 的系统上建立了 NFS，DECnet 的服务器建立在 VMS 系统上，而 LAN Manager 是将其服务器建立在单用户多任务的 OS/2 上而构成了网络操作系统。今天，微软公司推出了 Windows NT 才使人看到了真正的网络操作系统，令人耳目一新。它的网络功能不再是操作系统的一个附加层，而成为嵌入操作系统的一个有机的组成部分——内装网络。

内装网络是什么含义呢？首先由图 12.1 中可以看到，Windows NT 的网络平台是作为 NT 执行体的 I/O 系统中一个组件而嵌入系统内部的。这使得 Windows NT 无需安装其它网络软件，即可为用户提供文件共享、打印机共享、电子邮件和网络 DDE 等功能。因为在 Windows NT 的标准版本中，已经包含了一个采用 SMB（Server Message Block）协议的对等服务(peertopeer)的网络产品[该产品被微软公司称为工作组(Workgroup)]。其次内装网络意味着在 NT 中的网络组件将直接利用 Windows NT 的内部系统功能。这是由于在 12.8.1 节中所指出的，NT 把网络重定向器和网络服务器设计成一个文件系统驱动器，并运行于核心态，它可以直接调用 NT 执行体的其它部件的功能。例

如可以调用缓冲存储管理器的功能,以优化其数据传输性能。

特性之二:NT 的网络部分与 LAN Manager, MS-NET 间有高度可互操作性和网络级的兼容性。这种互操作性和兼容性可分以下两种情况:

(1) 如果 NT 作为 LAN Manager (或 MS-NET)的服务器,则可对网上的任何不同操作系统下的 LAN Manager (或 MS-NET)的客户提供与 OS/2 服务器(或 MS-NET 服务器)等同的服务。

(2) 如果 NT 作为 LAN Manager (或 MS-NET)的客户,可访问网上的任何 LAN Manager (或 MS-NET)服务器。

特性之三:与其它网络系统的互操作性。

(1) 作为客户可访问其它厂商的服务器(如 Novell NetWare , Banyan VINES , SUN NES 等)。

(2) Windows NT 上的应用程序可直接访问网上的非微软公司的文件系统(如 UNIX, VMS, Apple Macintosh 系统等)。

特性之四:提供方便地建立分布式应用程序的机制。

NT 提供了方便地建立和运行客户/服务器模型的分布式应用程序的机制,主要包括远程过程调用 RPC (Remote Procedure Call)和命名管道(Named Pipes)以及多种应用程序接口 API 。

特性之五:开放性好。

NT 的 I/O 系统的各种驱动程序均可由动态连接库 DLL 在系统运行期间可动态地装入和卸出。

12.9.2 Windows NT 网络的体系结构

下面以国际标准化组织提出的开放系统互连参考模型为参照,提出 NT 网络的协议体系与模型如图 12.9 所示,并对图上各部分加以介绍。

图 12.9 NT 网络协议体系与模型

(1) 位于应用层的“命名管道”是 Net BIOS 的更高层接口,它在两个系统之间提供一个抽象的、可靠的和易于使用的数据通路。命名管道是支持客户/服务器应用的重要机制。

邮件槽(mailslot)则用于支持网络邮件功能。

(2) 重定向器是解释网络 I/O 请求并生成对下层协议的调用,以实现网络 I/O 功能。重定向器用于客户/服务器的 SMB 的客户方,与 SMB 服务器方的服务器同处于会话层。该层是实现文件共享、打印机共享等功能的最顶层系统界面。

(3) 为支持重定向器和服务器,定义了统一的传输界面 TDI (Transport Driver Interface)。

(4) 传输层和网络层是由传输驱动模块(或称协议)所构成。NT 提供包含以下传输驱动模块:

NetBEUI　用以提供图中的 NetBIOS 界面;

TCP/IP　它是 UNIX 网络中的传输层与网络层上的协议,它可以使 NT 访问 Internet 网以及建立在 TCP/IP 上的分布式应用系统。

此外已开发的和正在开发的传输驱动模块有:

Novell 的 IPX/SPX;

DEC 的 DECnet;

Apple 的 Apple Talk;

Xerox 的 XNS。

有了上述传输驱动模块(协议),NT 网络就可以与其他厂商的网络产品实现互连。

(5) 在链路层上定义了 NDIS(Network Driver Interface Specification),供其他厂商依据 NDIS 环境规范开发其网络硬件驱动器。目前,已包含了十多种 NDIS 驱动器以支持(Ethernet),Token Ring 等物理层产品(网络适配器)。

12.10　对象管理程序

对象管理程序的主要功能是创建、管理、删除用来表示操作系统资源的对象。操作系统通过对象管理程序对资源进行统一的管理,使用共同的代码操纵它们。

进程、线程和 NT 执行体的其它部件在需要时可创建对象。NT 的对象管理程序在接到创建对象的系统服务要求后,要做以下工作:

(1) 为对象分配主存;

(2) 给对象一个附加安全描述体,以指出允许谁使用对象以及谁被允许进行的操作;

(3) 创建和维护对象目录表目;

(4) 创建一个对象句柄并返回调用者。

NT 对于对象的管理、组织和操作的模型是基于文件系统的模型。

对于对象管理的一个主要方面是对对象名空间的管理。每个对象有对象名,对象名是供共享进程查询对象句柄使用的。为了管理对象名空间,对象管理程序维护一个对象目录系统,每个对象在对象目录中均有相应的表目,整个对象目录是个树形结构。树的根用反斜杠“\”表示。叶节点是对象,中间节点是对象目录名。对象目录也是个对象,其内容如图

12.10 所示。对象名空间对所有进程都是可知道、可访问的。一个进程在创建对象或打开对象句柄 时指定对象名,此后进程使用对象句柄访问对象,它比使用对象名要快(因为可以跳过查找对象名工作)。对象句柄是一个指向由进程指定的对象表(该对象表是进程资源集的描述,对象表表目序号为句柄)的索引,对象表的表目中包含有认可的存取权限;由该进程创建的进程能否继承对象的句柄的信息以及指向对象的指针。当一个进程打开一个共享对象的句柄时还需指出它的操作类型。对象管理程序根据对象的安全描述体审查是否允许该进程使用以及是否允许其进行此类操作。

图 12.10　对象目录对象

图 12.11　对象名空间

对象名空间类似文件系统的符号名空间。访问一个对象也使用路径名,路径名的表示方法也与文件系统相同。而且也允许一个对象目录(中间节点)与其它分支上的某个对象(叶节点)进行符号连接,以便于访问。

NT 也提供对象名空间的安装和卸出。如图 12.11 所示在软驱 Floppy0 的盘有一个文件系统的名字空间 B,可以整个地安装到系统的对象名空间 A 上。

12.11　进程通信及本地过程调用(LPC)

12.11.1　线程间的同步

线程同步指一个线程主动停止执行并等待其它线程执行一些操作的能力。但一个线程希望与之同步的对象,除线程外还可有以下对象:

进程对象;
文件对象;
事件对象;
事件对对象;
信号量对象;
计时器对象;
变异(mutant)对象。

任何时候同步对象都处于两种状态之一:有信号状态和无信号状态。各种同步对象的有信号状态定义见表 12.7。

表 12.7　有信号状态定义

对象类型	置成有信号状态的时机	对等待线程的影响
进程	最后一个线程终止	全部释放(即解除等待,下同)
线程	线程终止	全部释放
文件	I/O 操作完成	全部释放
事件	线程设置事件	全部释放
事件对	专用客户或服务器线程设置事件	其它专用线程被释放
信号量	信号量减到 0	全部释放
定时器	设置的时间到或时间间隔到	全部释放
变异体	线程释放变异体	一个线程被释放

线程为要与一个对象同步,将调用对象管理程序提供的系统服务之一——等待,该服务功能将一个句柄传到同步对象。线程可以等待一个或几个同步对象。每当内核将一个对象设置为有信号状态时,它将检查是否有线程在等待该对象。若有则解除一个或多个线程的等待状态,(即所谓释放)使它们继续执行。

12.11.2　进程通信——本地过程调用(LPC)

NT 的客户进程与服务器进程之间的通信采用消息传送。而消息的传送必须经过 NT 执行体的本地过程调用 LPC。LPC 提供了三种不同的消息传送方法:

(1) 将消息传给和服务器进程相连的端口对象;

(2) 将消息指针传给与服务器进程相连的端口对象,并把消息存放在共享的主存区域中;

(3) 通过一个共享主存区域将消息传给一个特定的服务器。

第一种方法适用于小消息传送。系统为每个端口对象设置有一个固定大小的消息队列,作为消息通信之用。这方法与 4.4 节中所讲的方法类似。

第二种方法适用于传送大消息。当客户进程要传送大于 256 个字节的消息时,就必须通过共享的主存区域进行。消息大小受进程所分得的主存配额限制。消息传送过程如下:当客户进程要传送大消息时,就由它自己创建一个称为主存区域的对象——主存共享的对象,LPC 机制使得该地址空间为客户进程和服务器进程均可见,然后客户进程把大消息存放在该主存区域对象中;再向服务器进程的端口对象的消息队列中传送一个小消息指出所传送消息的大小和消息所在的地址指针。

第三种方法是针对一个子系统(服务器进程)可能有许多个通信端口的情况。限于篇幅不加探讨。

由于客户进程每次对服务器进程的服务申请都必须使用消息通信机制,所以消息传送机制的实现效率对整个系统的性能有重大影响。消息通信中的主要问题是消息传送需要切换运行线程的环境,一切消息传送至少需要两次切换,为减少这方面的开销,NT 的

设计者采取了许多措施使消息传送尽量减少(如成批处理客户的 API 调用)。为提高效率甚至使用了汇编语言。

12.12 Windows NT 的安全性

Windows NT 是一个安全性很强的操作系统。确实,目前安全性是操作系统的重要问题,美国国防部已鉴别出保证操作系统的安全特征,并把这些特征划分为七个安全级别。NT 已达 C2 级,C2 级要达到如下目标:

(1) 安全登录功能:在允许用户存取系统前,要求他们输入一个唯一的登录标识符和口令来证实自己。

(2) 自选存取控制:允许资源的属主决定谁能存取此资源及能对它做什么。属主通过用户或用户组给予存取权限来实现。

(3) 监视:要求有一种能力来监测并记录与安全性有关的重要事件或任何创建、存取、删除系统资源的企图。它使用登录标识符来记录执行过此动作的用户。

(4) 主存保护:防止任何人在一个数据结构已经释放回操作系统后读取别人写的信息。在主存被使用之前,它要被重新初始化。

NT 采用了下述措施来保证安全性:

1. 登录进程和安全子系统

NT 要求每个用户建立帐号,每个用户帐号有一个安全特征表,安全子系统用此表来验证用户,用户使用系统前要在该帐号上登记,询问帐号名和密码。

2. 存取令牌

在安全子系统认定登录是真实的之后,它就构造一个始终附着于用户进程的“对象”——“存取令牌”,在进程要使用系统资源时,作为进程的正式标识卡,令牌上有存取控制表。

3. 存取控制表

所有文件、线程、事件和存取令牌在内的所有对象,在它们创建时,都被分配安全描述体,其主要特征是一个用于对象的保护表。

存取令牌识别一个进程及其线程,而安全描述体枚举哪些进程或进程组能够存取一个对象。

4. 主存保护

NT 还在主存使用方面采取了以下措施来提高系统的安全性:

(1) 每个进程有单独地址空间,硬件不允许线程访问另一进程的虚拟地址。

(2) 两种运行状态——核心态,用户态。不允许用户态访问系统代码和数据。

(3) 以页面为基础的保护机制。每个虚拟页面有一组与它相关的标识,它们决定在核心态和用户态下的访问类型。

(4) 以对象为基础的主存保护。每次,一个进程打开一个段对象的句柄或映象一个视口到段对象时,NT 的安全引用监控程序检查该进程是否被授权访问此对象。

5. 客户/服务器模型

客户/服务器模型也可起到有效的主存保护作用。由于每个子系统是分立的,不受其它子系统的影响,所以每个子系统可以独立创建、维护数据结构。而且由于子系统是用户态应用程序,不能修改 NT 执行体的数据结构或是调用内部操作系统过程,它们只能调用系统服务来对 NT 执行体进行存取。这都增加了安全性。

12.13 综　　述

广大的操作系统的爱好者和读者常喜欢从操作系统的结构、功能和性能以及用户友好性等方面来评述操作系统的特性和优劣。

从结构上来说, Windows NT 比 UNIX 的结构清晰、漂亮而且规范。NT 的用户态层是进程的活动层,在这个层次上进程之间是用客户/服务器模式建立相应之间的关系的。在其内层,核心态层是 NT 执行体——一个基本的操作系统。这是一个漂亮的具有微内核的层次结构(半序)。结构非常清淅,这给学习者带来了易理解性,肯定将为读者所欢迎。

从功能和性能上来说,NT 是一个 90 年代的、模块化的、32 位可移植的操作系统。它具有工作站和小型机上的操作系统所具有的强大功能,包括一个稳健的多重的文件系统;动态优先级的多任务/多线程环境;支持对称的多处理;安全性级别达 C2 级;与 DCE[①] 兼容的远程过程调用(RPC),支持 POSIX 及 TCP/IP 协议的网络功能。与 UNIX 相比, NT 对每一个它所支持的平台都能兼容一致地提供所有这些功能,这些平台包括了 CISC (如 Intel 系列的机器) 和 RISC[②] (如 DEC Alpha, MIPS R4x00, Intergraph Clipper 等系列的机器)。Windows NT 支持多种操作系统环境(所以被称为“多头蛇操作系统”、“变色龙操作系统”),也就是说它能够运行多种应用程序(32 位 Windows ,MS-DOS/Windows ,基于字符的 OS/2 以及 POSIX)。NT 支持多种符合工业标准的通信协议(如 NETBIOS 和 TCP/IP)和机制(如套接字和 DCE 远程过程调用)。

Windows 开放服务结构(WOSA),是一个可用于企业范围的应用程序实现互用的开放结构框架。NT 作为一个开放式系统,是非常符合追求开放策略的企事业所要求的计算结构。

NT 是一个被设计成在网络工作站上运行的,客户/服务器模式的,分布式计算的现代操作系统。它包含了网络和多线程功能以支持分布式客户/服务器程序。所以可以说 NT 提供了多用户的计算模式,而不是传统操作系统的主机/哑终端的模式。

从用户友好性来说, Windows 图形用户界面是如此地使广大计算机用户喜爱以至于着迷的事实,充分说明了 NT 的用户友好性是卓越的。另外 X 窗口系统也是一个面向网络的图形用户界面系统, UNIX 的供应商都支持 X 终端,所以支持 X 窗口系统对许多 UNIX 用户来说是很重要的。目前 DEC 和 NCD 等几家厂商已宣布推出了面向 NT 的 32 位 X 服务器应用程序。在 NT 上运行 X 服务器应用程序可使用户得到最强的软件功能

---

① DCE :分布式计算环境的英文缩略语,它是由开放软件基金会(OSF)开发并发放许可证的大型软件系统。

② CISC 和 RISC 是硬件平台的两种发展潮流。CISC 是复杂指令系统计算机简称, RISC 是精细化指令系统计算机的简称。二者各有特点,竞争剧烈。

—— 即得到 X 终端，同时也是一个面向其他分布式客户/服务器应用程序的高级平台。

任何事物均有两面性。一个事物的优点，往往也是其缺点所在。Windows NT 的优点无疑是它的强功能和通用性，但这正意味着它的缺点。可是微小的缺点是挡不住它的光辉，Windows NT 如旭日东升在 90 年代的地平线上。用不了几年，在计算机操作系统领域必将如日中天，读者将会经常与它打交道，希望本书将会对你有所帮助。

第 13 章　UNIX 操作系统

13.1　UNIX 操作系统概述

UNIX 操作系统是一个当今世界上十分流行、应用十分广泛的操作系统。在小型计算机和微型计算机领域中,得到了广泛的应用。UNIX 操作系统是一个通用的、交互式的分时系统。它是由美国贝尔(Bell)实验室的 D. M. Ritchie 和 K. Thompson 于 1969 年首先在 PDP-7 上实现的。该系统吸取了当时许多操作系统的成功经验,如美国麻省理工学院(MIT)研制的 CTSS 和 MULTICS 等系统的优点。UNIX 系统的设计者原来都曾参加 MULTICS 系统的设计工作。MULTICS 是一个大而复杂的系统,它是为一个大型的通用计算机而设计的。UNIX 吸收了这些系统中成功的设计,改造和删除了与基本功能关系不大的部分,大大压缩了系统的规模,使之能以不多的代码,在一台小型以至于微型的计算机上完成许多大型机的操作系统功能。

UNIX 操作系统的第一个版本是在 PDP-7 计算机上用汇编语言编写的。1972 年才用 C 语言(这是一种比 BASIC 语言高级的语言)改写而成,改写后的 UNIX 系统提高了兼容性和可读性(易懂性)。UNIX 首先是在大学中流行起来的,到目前,已经过了多次修改,产生过很多版本,(现在已产生了第七版)。但它的根本思想没有什么变化,只是在功能上进行了一些有意义的扩充。

标准的 UNIX 是一个多道的分时系统,主要是为美国的 DEC 公司的 PDP-11/34, 40, 45 和 70 计算机而设计的,也适用于 DEC 公司的 VAX 计算机。它支持 40 个分时用户同时工作,其主要特点是:

(1) 具有分层的、可装卸的文件卷系统。该系统具有完整的文件保护的功能,提供了设备独立性的特点和使用户程序设计简单化的便利。

(2) 任何一个程序或程序组可以不加任何改变地既可在前台(交互式)也可在后台(非交互式)异步地运行。

(3) 把文件、目录和设备均统一地看作为文件,统一地进行处理。所有文件均看作为无结构(无记录)的流式的字符串序列,这给用户提供了一个简单、统一的接口,使得用户对文件和对设备的操作是统一的。

(4) 提供了输入输出缓冲技术。主存分配和释放以及磁盘空间分配和释放的程序模

块是共同的。所有这些技术都由操作系统自动实现,对用户来说是透明的。而且由 C 语言编译程序产生的所有程序中的过程代码是可再入的和可共享的纯代码。

(5) 提供了功能完备、使用灵活的命令语言即 shell 语言。这是 UNIX 系统与用户的接口,它既是终端用户交互作用时使用的命令语言,又是方便的程序设计语言。用户可通过 Shell 语言来使用 UNIX 系统提供的各种程序设计工具。尤其方便的是,它允许一个程序的输出直接作为另一个程序的输入,这就十分方便于程序的开发,使得新开发的大程序可由许多已有的小程序合成。

(6) 标准的 UNIX 系统提供了许多程序包的集合。包括正文编辑程序,可编程的命令语言(shell 语言)解释程序,连接装配程序,文件格式化(包括数学公式)程序,调试程序,正文处理程序,汇编程序,十几种程序设计语言的编译程序,分类应用程序,状态查询,用户间通信,以及管理和维护程序,标准的系统和用户子程序库和游戏程序包等。

小型的 UNIX 系统 MINI-UNIX 操作系统基本上是 UNIX 的子集,它是为像 PDP-11/10, 20, 30 等计算机设计的,它能支持 4 个用户和 13 个并行进程同时工作。

UNIX/V7 是 UNIX 第七版,是为 PDP-11/45 和 70 设计的,它允许一个文件的大小可达百万字节,系统具有更强的兼容性,C 语言的功能也得到了扩充,正文处理能力和照相排版软件的功能都有了更大的提高,shell 语言功能也更强了,包含了行变量,结构程序设计,陷阱处理以及从一个机器到另一个机器的文件转移能力。

总的来看,UNIX 操作系统的设计目标是:

(1) 简单性:它着眼于提供给用户一些基本的必须的功能。并不追求一些大型通用操作系统所希望达到的"无所不包、无所不能"的功能要求,所以 UNIX 系统比较简单,但功能又足够的强而且十分方便使用。

(2) 通用性:它力求用一种方法,同一机构来服务于几个不同的目的。例如对文件、设备和进程间的消息缓冲区的读和写操作都是同样的系统调用,同样的程序模块。又如同样的命名,别名和存取保护机构都适用于数据文件、目录文件和设备。再如处理机陷阱(trap)和软中断使用的也是同样的机构等。

(3) 程序开发的方便性:它创建了一个十分方便于程序开发的环境,使得一个新开发的大任务可由许多已有的小程序合成,而不必都重新设计。

13.2 系 统 结 构

UNIX 系统的结构大致如图 13.1 所示,它由内核和用户层组成。内核是 UNIX 的心脏,它划分为 44 个源代码文件,包括两个汇编文件,28 个 C 语言文件,14 个 C 语言的全局变量文件。两个汇编文件分为 33 个汇编子程序,共约占 1000 行代码,主要用 于系统初启,中断处理等与硬件细节密切相关的部分。28 个 C 语言文件分为 190 个子程序,约占 10000 行代码。这些源代码文件是作为独立编写和编译的单位。由于内核的 233 个模块基本上是以全局变量为中心的无序模块结构,所以其调用关系十分复杂,整个内核可以按其功能划分为:

① 存储管理;

② 进程管理;

③ 进程通信;

④ 中断,陷阱与系统调用;

⑤ 输入输出管理;

⑥ 文件系统。

在 UNIX 核心之外运行的是用户态的程序,这包括用户程序、大量的应用程序和各种程序包子系统,它们在 shell 命令语言解释程序管理之下,作为进程的用户态的一部分来运行。所有用户进程(或者说进程的用户态)只能用陷阱的方式,调用内核提供的 41 个系统调用程序来请求内核为之服务。而用户只能通过 shell 语言在终端上或后台得到服务。

下面分别介绍 UNIX 系统的主要功能的实现情况。

图 13.1 UNIX 系统的结构

13.3 进 程 管 理

13.3.1 程序状态字和通用寄存器

PDP-11 的机器的程序状态字 PS 的格式如图 13.2。

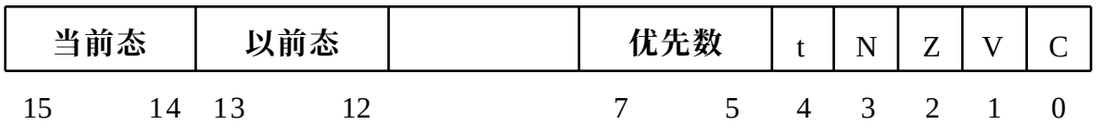


图 13.2 PDP-11 的程序状态字

图中 C：进位位;

V：溢出位;

Z：结果为零位;

N：结果为负位;

t：陷阱位;

5～7 位：中断优先级,取值范围 0..7;

12～13 位 表示原来处理机所处的状态‘00’表示核心态(系统态或管态),‘11’表示用户态(目态);

14～15 位：表示当前处理机所处的状态。

PDP-11 的处理机可以处于两种不同的工作状态:即核心态(系统态)和用户态。当处理机运行操作系统本身的各程序时,其工作状态为核心态。除此之外,处理机的状态均为用户态。

不同的处理机状态决定了:

(1) 当存储访问时,虚拟地址(相对地址)变为主存物理地址时使用段寄存器的不同集合。

(2) 使用不同的栈顶指针寄存器 r_6。

(3) 是否可以使用某些特权指令,如‘halt’指令等。

PDP-11/40 有 9 个通用寄存器,每个寄存器的长度为 16 位。其编号为 r_0～r_7:

其中 r_0, r_1：传递过程调用时的输入、输出参数;

r_2, r_3, r_4：用作过程的局部变量(当过程执行时)的存放;

r_5：环境指针;

r_6：堆栈指针。用户态,核心态各一个,所以有两个 r_6 寄存器;

r_7：程序计数器(PC),即指令地址寄存器。

13.3.2 进程和进程控制块 PCB

在 UNIX 系统中,每个用户在自己的‘虚拟计算机’上运行。虚拟计算机的当前状态称为一个“映象”(image),一个映象包含:

(1) 存储映象(虚拟地址空间);

(2) 通用寄存器的值;

(3) 打开的文件的状态;

(4) 当前记录;

(5) 以及其它信息。

一个存储映象可划分为三个逻辑段(如图 13.3 所示):

(1) 可再入的过程段,它是从虚拟地址空间的相对地址 0 开始。该段可共享;

(2) 数据段: 在过程段之后为数据段,可动态地向高地址方向扩展;

(3) 堆栈段: 它由虚拟地址空间的最高地址开始,当有栈元素推入时,栈指针向低地址方向移动。

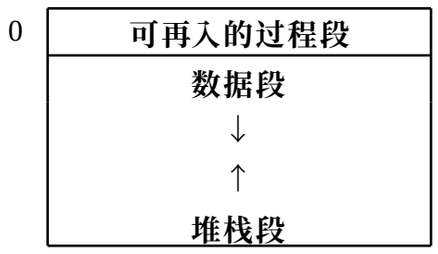


图 13.3 存储映象

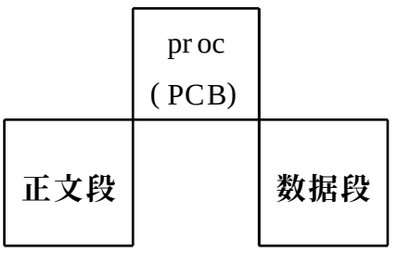


图 13.4 一个进程的组成

那么什么是进程呢,在 UNIX 中进程就定义为"一个映象的执行"。

进程的映象是一个抽象的、逻辑上的数据结构,是它的虚拟计算机的状态。但进程的映象,或者说一个进程,在物理上可看作由三部分组成,如图 13.4 所示,下面分述图示的三个部分。

1. 进程控制块 PCB

如同其它操作系统一样, UNIX 操作系统也是建立在进程的概念之上,并以进程作为基本的组织概念,整个计算机系统是所有进程集合的活动。系统为了管理这些进程的活动,必须为每个进程设立一个进程控制块(PCB)来记录各个进程的状态以及进程映象中的一些数据,进程控制块也是进程存在的标志,是操作系统的重要数据基。通常 PCB 中包含的信息量很多, PCB 所占的空间很大。

UNIX 系统为了节省 PCB 所占的主存空间,把每个进程的 PCB 分为两部分:

(1) 常驻主存部分,称 proc 结构,其中包含有进程调度时必须使用的一些主要信息;

(2) 非常驻主存部分,称 user 结构,这里登记有更多的进程运行时才要用到的信息,它随用户的程序和数据部分而换进和换出主存。

整个系统有一个进程表,称为 proc 数组。每个进程的 proc 结构(PCB 的一部分)均为此数组的一个元素,称为进程表(或 proc 数组)的一个表目,该数组的元素个数是一定的,UNIX 允许系统中最多有 50 个进程。每个 proc 结构中包含有进程名(标识号),它的各段的位置,以及状态和优先级等调度信息。

2. 数据段

它分为三部分,如图 13.5 所示。其底部为进程数据区(ppda)共 1024 个字节。它又分为两部分,最底部的 289 个字节是属于 PCB 的一部分的 user 结构,它含有进程的更多的信息。另一部分是该进程的系统栈,从顶向下扩展,这部分(ppda)虽然在物理上随数据段一起换进、换出主存,但它们是属于进程的核心空间的一部分,也就是说进程的用户程序部分不能访问这一部分,只有该进程的系统程序部分(处理机的当前状态为核心态时,即进程请求操作系统服务而执行操作系统的某个程序时)才能访问进程数据部分 ppda。数据段的顶部是用户栈区,物理上是从顶向下扩展。数据段的中间部分是可读可写的用户的程序和数据区,也可以有非可再入的(不纯的)程序正文。

3. 正文段

包括该进程所执行的所有可共享的、可再入的纯代码和常数。一个进程的正文段也可以不存在。

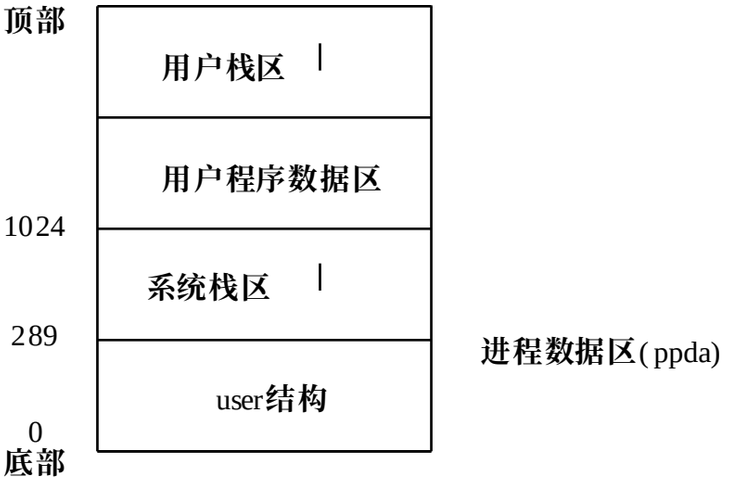


图 13.5　进程数据段结构

操作系统为了管理系统中的所有这些仅读的可共享的纯段(正文段),设置了一个正文表称为 text 数组。在主存中的所有正文段均在此正文表(text 数组)中有一个表目。该表目中包含有该正文段在主存和磁盘上的地址,该段大小和使用此段的进程数目等信息。该表大小是一定的,UNIX 系统中最多允许在主存中同时存放有 40 个正文段(即 text 数组元素为 40 个)。

表 13.1 列出了 UNIX 系统的 proc 结构、user 结构与 text 结构的内容。

表 13.1　UNIX 系统的 proc、text、user 结构

	proc 结构	
char	p-stat	: 定义进程的七种状态之一
	null	: 该表目未分配
	ssleep	: 高优先级睡眠(优先数< 0)
	swait	: 低优先级睡眠(优先数> 0)
	srun	: 就绪
	sidl	: 进程正被创建
	szomb	: 进程结束
	sstop	: 因被跟踪而暂停
char	p-flag	: 进程特征
	sload	: 在主存否的标志位
	ssys	: 正在调度中的进程标志位
	slock	: 该进程不许被换出的锁住位
	sswap	: 该进程正被换出标志位
	strc	: 被跟踪标志位
	swted	: 另一被跟踪标志位
char	p-pri	: 优先数

char	p-time	: 在主存或磁盘上的驻留时间(秒)
int	p-addr	: 进程数据段 ppda 在主存或盘上的起始地址(块号)
int	p-size	: 数据段大小(块为单位)
int	p-wchan	: 睡眠原因
int	p-textp	: 指向 text 结构的指针
char	p-sig	: 收到的软中断号码
char	p-uid	: 用户标识
chap	p-cpu	: 用以计算进程优先数
chap	p-nice	: 用以计算进程优先数
int	p-ttyp	: 对应终端的 tty 结构地址
int	p-pid	: 进程号
int	p-ppid	: 父进程号

text 结构

int	x-daddr	: 正文段在磁盘上地址
int	x-caddr	: 正文段在主存中地址
int	x-size	: 正文段块数
char	x-count	: 使用此正文段的进程数
char	x-ccount	: 使用此正文段内存中的进程数
int	x-iptr	: 含有此正文段文件的 i 节点指针

user 结构

各个进程的 user 结构的虚拟起始地址都是 u= 14000,其地址空间对应核心空间第 7 页。其物理地址可以映射到主存中各进程的 user 结构地址。

int	u-rsav[z] u-qsav[z] 保留 r5,r6 u-ssav[z]	
int	u-procp	: 指向本进程的 proc 结构的指针
int	u-uisa[16]	: 16 字的数组,存放本进程用户空间地址寄存器内容
int	u-uisd[16]	: 16 字的数组,存放本进程用户空间说明寄存器内容
int	u-tsize	: 本进程正文段的块数
int	u-dsize	: 本进程数据段的块数
int	u-ssize	: 本进程栈区的块数
char	u-error	: 出错号码

续表

char	u-base	: 文件传输中表示主存地址
char	u-conut	: 文件传输中表示传送字节数
char	u-offset[2]	: 文件传输中表示文件内相对位移量(字节)
int	u-cdir	: 当前目录项 i 节点指针
char	u-dbuf:	: 保留当前用到的文件的路径名分量
char	u-dirp	: 指向当前目录文件名的指针
struct	u-dent	: 存一个文件目录项。该结构中包括 i 节点号, 该文件路径名分量
char	u-uid	: 有效用户号
char	u-gid	: 有效用户组号
int	u-ofile	: 用户打开文件表, 共 15 个表目元素的数组
int	u-signal	: 软件中断处理程序入口表, 是 20 个元素的数组
int	u-arg[5]	: 当前系统调用中, 用户传入的自变量
char	u-intflg	: 系统调用执行完成否
int	u-sep	: 指令空间与数据空间分离否
int	u-pdir	: 父目录项 i 节点指针

整个系统中的进程控制情况如图 13.6 所示。

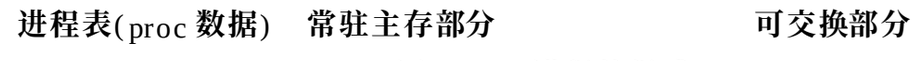


图 13.6　进程的组成

需要指出的是 UNIX 的系统核心的所有程序都是作为进程的一部分而运行, 它们被包括在每一个进程的虚拟地址空间中。所以每一个进程都包含着两个部分: 系统程序部分和用户程序部分。当一进程在用户态运行时, 执行的是用户程序。此时若发生中断或陷阱,

则转入核心态,执行系统核心程序。所以在 UNIX 系统中,有时把进程 i 的用户程序部分称为用户进程 i,而把其中的核心程序部分叫作系统进程 i。实际上用户进程 i 和系统进程 i 都是进程 i 的一部分,这两个进程(用户和系统进程 i)的 PCB 是同一个(进程 i 的 PCB)。只是这两个进程所执行的程序不同,映射到不同的物理地址空间,使用不同的堆栈。一个系统进程的地址空间中包含所有的系统核心程序和各进程的进程数据区 ppda ,所以各进程的系统进程除 ppda 不同外,其余部分全是相同的(其物理的地址空间也相同)。而各进程的用户进程部分则是各不相同的。

13.3.3 进程的控制

UNIX 中提供了许多有关进程控制和管理的功能,主要有下述几种。

1. 进程的建立原语

父进程可以通过调用系统原语 fork 来建立一个新的进程,称为子进程。该原语为新建立进程分配一个 proc 表目,并调用 newproc 过程将 proc 表目初始化。在 UNIX 中子进程共享父进程的所有打开文件和当前目录,但并不共享父进程的主存。所以初始化工作主要复制父进程的 proc 表目中的某些项目以共享其打开的文件,正文段和当前目录表的表目(i 结点),为子进程申请主存并把父进程所有可写的数据段部分复制到主存中去。

2. 进程的执行原语 exec

子进程被建立后共享父进程的文件和正文段、数据段。为了扩充子进程的执行能力,进程可以通过系统调用 exec 来执行一个新的文件。这使得该进程以在此文件中指出的新的正文段和数据段来调换当前的正文段和数据段。这个操作仅改变进程执行的程序,而并不改变进程本身。

3. 进程的同步和通信

UNIX 中对可写的公用变量通常只允许核心程序(即处理机是核心状态)访问,也就是说由系统进程访问。那么这些系统进程间如何实现临界段互斥执行呢。其主要的方法是:

(1) UNIX 的调度策略规定在核心程序执行期间发生中断或陷阱时,不进行进程的转换调度。只有在用户程序执行过程中被中断或陷阱、并且在处理中断后返回时,才可以进行进程的调度以转让处理机。所以在系统进程执行期间不会插入其它进程的操作。

(2) 提高临界段等代码段的处理机中断优先级以屏蔽某些类别的中断,用来防止在核心程序内插入中断和陷阱处理程序时,可能改变进程的状态或修改公共数据等情况。

系统进程还可利用系统调用 sleep 和 wakeup 原语实现进程间同步。sleep 原语使调用者进程以指定的原因和优先数睡眠(阻塞),而 wakeup 原语则唤醒在指定原因上睡眠的所有进程。这两个原语,用户进程是不能使用的,但用户进程之间可以利用核心程序提供的软中断(信号)方法来实现某种通信。它是某一个进程向另一个与之相关的进程发送一个 0～19 之间的数(放在接收进程的 proc 结构 p-sig 中)。接收者进程在退出陷阱处理程序、时钟中断程序,和 sleep 原语时均要检查是否收到某个软中断信号,如已收到,则执行预先规定好的程序。UNIX 系统的进程通信功能是很弱的,为了满足协同进程间的通信需要而设置了这一最简单的通信功能。

4. 进程调度与交换

在一般的操作系统中,进程调度程序和交通控制程序是一个把 CPU 在各就绪进程间进行调度的转接站,是被某些原语调用的最基本的原语。它并不属于某个进程的一部分。而在 UNIX 系统中进程调度的功能是由一个专门的进程——0# 进程来负责,由于 UNIX 系统的进程调度的主要功能是响应分时用户,这主要由 0# 进程来负责。所以 0# 进程包括两部分任务:

(1) 把进程的映象从主存换出到磁盘;

(2) 分配处理机。

UNIX 系统的进程调度是按照其优先级的高低进行调度的。系统进程的优先级高于用户进程的优先级,其最初的优先级取决于进程所等待的事件,事件优先级的排列次序为:磁盘事件,终端事件,时钟事件和用户进程事件。用户进程的优先级是基于其所使用了的处理机时间的多少而动态地变化。优先级高的进程优先得到处理机。每个进程分得的时间片一般为一秒。UNIX 每秒为每个用户进程计算一次优先数,优先数愈小,优先级愈高。

在分时系统中,各交互进程需要经常在主存和磁盘之间进行交换。一个在主存中非运行状态的进程,其进程的映象将被从主存中换出而存入磁盘的对换区中。而一个被选中运行的磁盘上的就绪进程,其映象可以被换入主存。在这两种情况下,均有主存或磁盘空间的分配和释放问题。在 UNIX 中主存的磁盘空间的分配和释放模块是相同的,所使用的分配算法是“最先适应法”,其自由存储块数据库的组织是按地址排序的。

一个进程在运行过程中如果要求增加分给它的主存块数时,系统就重新分给它一块足够大的主存空间,并把旧的主存区中的内容复制到新分给的主存区中同时释放旧的主存区。如果进程要求增加主存空间而系统不能满足其要求时,就将该进程换出主存而放入磁盘对换区中,直到以后有足够主存时才将它重新换入主存。

所有进程的对换工作全是由 0# 进程负责,其对换过程如下:当 0# 进程决定要从磁盘上换入进程时,它首先扫描系统的 PCB 表(proc 数组)以找出在磁盘上驻留时间最长的就绪进程并将其换入主存。为此要为其分配主存。如果主存不够,0# 进程要从进程表中查找在主存驻留时间最长,并正等待慢速事件的那些进程(象输入输出之类)换出,以腾出更多的主存空间。进程被换入主存后,就同主存中的进程一起争夺对 CPU 的控制权。

5. 进程的终止

在 UNIX 中,一个进程只能通过系统调用 exit (也包括调用 exit 的 rexit 原语)原语来自己终结自己。父进程可以要求子进程自我终止,但不能直接终止其子进程。所以一个进程可在以下情况通过调用 exit 原语终止自己:

(1) 自愿地调用 exit 终止自己;

(2) 父进程通过 trap 指令调用 kill 原语通知同一终端上所有进程令其终止自己;

(3) 由于来自程序本身的错误(如非法指令)而导致通过软中断的信号而不得不被迫终止该进程。

进程被终止时,关闭所有文件,将当前目录项的访问计数减 1,释放正文段。将进程数据 ppda 写入盘对换区,释放数据段空间,唤醒父进程和 1 进程,调用 swtch 放弃处理机。

13.4 文件系统

13.4.1 UNIX 文件系统概述

UNIX 系统中把文件看成是由一串字符组成的无结构的(无记录结构,但用户自己可以按需要进行结构)信息的有序集合。每个文件由用户指定一个文件名以进行标识。文件大小按字符数计算,通常,一个文件最大不能超过百万字节。文件常被存储在大容量磁盘或磁带上。

UNIX 文件系统将文件分成三类:

(1) 普通文件: 即通常的正文文件。这是一个无结构的以 512 个字节为一块,顺序存取的字符序列。在文件的组织形式上无明显的索引文件、连续文件,链接(串连)文件之分。在文件的访问形式上也无顺序和随机访问之别。用户可通过系统调用 seek 原语预先确定下次文件读写的开始位置。用户也可按自己需要来结构自己的文件,只是文件系统不承认也不了解这种结构。

(2) 目录文件: 如同所有的文件系统一样,UNIX 也把目录本身看成是文件,称为目录文件。一个目录文件包含有多个目录项,每个目录项的格式如图 13.7 所示。目录项的长度为 16 个字节,文件名最多不超过 14 个字符,另外两个字节用作该文件的 i 节点指针称为 i 节点的索引号 ino 。这样一个 512 字节的物理盘块可以存放 32 个目录项。

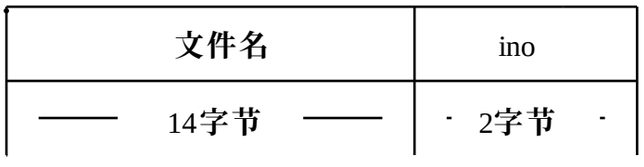


图 13.7 目录项格式

如同其他文件系统一样,一个驻留在盘上的文件,除了有文件正文本身外,还要在某个目录文件中有一个文件目录项(如同第 10 章中所述的符号文件目录项)和一个 i 结节(如同第 10 章所述的基本文件目录项)。一个 i 节点包含有该文件的更多的信息,主要有:

① 文件主标识;

② 文件主的同组用户标识;

③ 文件的保护信息;

④ 该文件本身在盘或带上的地址(物理块号);

⑤ 文件大小(字节数);

⑥ 文件建立的时间上次使用的时间和上一次修改的时间;

⑦ 与该文件进行连接(link)的进程数目;

⑧ 文件的类型。

一个物理设备上的所有文件的 i 节点放在一起形成 i 节点表,i 节点表放在该文件卷的 i 节点表区中。但 i 节点表本身不视为文件,而是数据结构。

(3) 特别文件:UNIX 把每个 IO 设备看成是一个特别文件,与普通的文件一样处理,这样可以使得文件与设备的 IO 尽可能统一。

13.4.2 文件目录结构和文件(路径)名

如上所述,一个文件除其本身内容外还要包括一个目录项和一个 i 节点。文件系统的目录结构是一个多层次的树形结构。它有一个根(root) 目录,是整个目录结构的基础。从根目录往下的各次级节点,既可以又是一个目录文件,也可以是一个普通文件或特殊文件。如同其它文件系统一样,目录的层次结构是通过符号文件目录(在此为目录文件) 实现的,而基本文件目录(在此为 i 节点表) 则是线性结构。

每个文件有用户给起的文件外部名(目录项中给出),还有唯一的内部名,该内部名由三部分组成:主设备号,次级设备号和 i 节点号。主设备号对应于一组设备驱动程序入口点的索引号(设备类型号),次级设备号对应于一组该类设备中某一台的索引号,i 节点号是指该设备上的文件卷中 i 节点表区的索引号。

如同其它文件系统一样,UNIX 访问某文件时也使用路径名。文件的路径名是由目录结构中各层次分量顺序排列构成的,各分量之间用“ / ”隔开。文件查找通常由根开始,例如 / user1/ Mat/ com 文件的查找,先由根目录中找 user1 目录,然后从 user1 目录中找 Mat 目录,而后又从 Mat 目录中找出表目 com (最左边的“ / ”代表根目录)。然而,如果每次查找均从根目录开始,不但麻烦,而且增加系统开销。不过,由于一个用户在一段时间内所涉及的文件一般有一定范围,所以可以指定一个当前目录(值班目录)。如果路径名的最左边不是以“ / ”打头,则表示从当前目录查找起,这可以大大节省查找时间。用户可以根据需要通过系统调用 chdir 原语来随时改变当前目录。

为了方便和加速对文件的访问,在一般文件系统中有活动文件表和活动符号文件表来记录被打开文件在基本文件目录和符号文件目录中相应的表目内容。与此相类似,在 UNIX 系统中,主存有一个活动 i 节点表(该表有 100 个表目) 来登记所有已被打开的文件的 i 节点内容,以免每次访问文件时都要从文件卷(盘) 上的 i 节点表中去查找该 i 节点。除此之外系统还有一个打开文件表(最多可登记 100 个表目) 来登记系统中所有已打开文件和 pipe 文件。每个用户也都有一个打开文件表(最多可达 15 个表目,该表在各进程的进程数据区 ppda 的 user 结构中) 来登记该进程所打开的活动文件,其中每一项只登记该文件在系统打开文件表中的入口索引指针。在系统打开文件表中,每一项登记有:文件访问计数;指向此文件在主存 i 节点表的入口指针;读、写字符的位置(偏移量) 指针;对文件进行读、写和 pipe 文件这三种操作类型的标志。

至此,我们可以看到一个被打开的文件需要如下资源:

① 在盘上某个目录文件中要有一个目录项(16 个字节);

② 在盘的文件卷 i 节点表区中有一个 i 节点项(32 个字节);

③ 若干个盘块以存放文件本身的内容;

④ 内存活动 i 节点表中有一个 i 节点项;

⑤ 一个系统打开文件表中的登记项;

⑥ 一个用户打开文件表中的登记项。

附带指出,盘 i 节点和主存活动 i 节点的内容基本相同,但由于使用情况的不同也有微小区别。这区别在于活动 i 节点中要登记该节点所在的设备号以及该节点在该设备中

的 i 节点号 ino。

13.4.3 文件卷的动态装卸和安装

UNIX 的文件卷是指在同一设备上的所有的文件、这些文件的层次结构的目录和该设备的 i 节点表的一个自包含集。UNIX 文件系统的一个最大特点是文件卷可以方便地动态地装卸。系统初启时,只有一个文件存储设备,即根设备,其名字常驻于系统。其它的存储设备可以通过系统调用‘smount’原语安装上去。安装上去的设备的层次目录结构被装到整个系统的层次目录结构的某一节点上(该节点所对应的文件是专为安装新文件卷而创建的空文件),从而两者形成一个新的目录树系统。与之相反的是也可把安装上去的文件卷完整地卸下来,从而恢复安装前的状态。

UNIX 为了文件卷的安装和卸下而设置了一个安装表以反映文件卷动态装卸情况。

13.4.4 文件的共享和联接

UNIX 中为各用户共享文件提供了如下的方式:

(1) 父进程用 fork 原语创建的子进程自动地共享父进程的所有打开的文件。

(2) 不同进程之间可以用系统调用 link 原语来连接非目录文件,从而可以直接共享该非目录文件。而且不同用户还可对此文件使用不同的文件名,连接原语形式为:

link (oldname, newname)

其中 oldname 是该文件原来的文件名, newname 是给该文件起的别名。与此相应的还有取消连接原语 unlink 。

(3) UNIX 提供了一种无名文件,在相关进程间建立传输通道,称之为 pipe (管道),数据可在两个进程间直接传递。

有关对文件和目录的操作请看 13.7 节系统调用。

13.5 设备管理和输入输出系统

UNIX 的输入输出系统由两部分组成:

(1) 成块(或结构)的输入输出系统;

(2) 字符(或非结构)的输入输出系统。

前者以物理块(512 个字节)为单位存取,后者以字符为单位存取。在 UNIX 中对输入输出的管理主要通过五种系统调用:打开文件(open) ;关闭文件(close);读(read)、写(write) 文件和定位查找(seek)。这同文件系统中是相同的。

不管块设备(磁盘、磁带)还是字符设备的输入输出系统均有缓冲区以缓和处理机与输入输出设备之间传送速度的不匹配。对于块设备来说共有 15 个缓冲区,每个区 512 个字节。每个缓冲区还有一个缓冲区控制块(缓冲区首部),用来对缓冲区进行管理,并记录有该缓冲区使用情况的状态信息。对于字符设备来说,系统提供 100 个缓冲区,每个缓冲区 6 个字节。

输入输出系统的系统调用形式如下:

(1) 打开一个文件:

fd= open (filename, mode)

其中 mode 是指访问类型是读、写还是两者皆有; fd 是被打开文件的文件描述符。

(2) 读和写一个文件:

nobytesread= read (fd, buffer, nobytesdesired)

nobyteswritten= write(fd, buffer, nobytesdesired)

其中 buffer 是使用的缓冲区; nobytesdesired 是想要传送的字符数; fd 是文件描述符; 等号左侧为实际传送的字符数。

(3) 读写指针定位: 在 UNIX 中所有的读和写全是顺序的, 如果想要直接访问文件中某一位置则可以用 seek 原语来调整读写指针位置:

seek(fd, offset, offsettype)

其中 offset 是读、写的偏移量; offsettype 是说明 offset 是个相对量还是绝对量, 是以字符还是以块为单位。

(4) 关闭一个文件:

close(fd)

13.6 管道线 pipe 机构

UNIX 系统最重要的贡献之一是关于 pipe (管道线)的概念。所谓 pipe , 是连接两个进程之间的一个打开的共享文件。一个进程(写者)可以从 pipe 一端写入数据, 而另一个进程(读者)可以从 pipe 另一端读出(取走)数据。所以 pipe 的作用类似于两个进程间利用消息缓冲区进行消息通信, 只不过消息缓冲区通常很小, 而 pipe 可以进行大批的数据传输。所以 pipe 不但可以作为进程间的同步通信工具, 而且可以用来进行大量的数据传输。由于数据传输都是经过内存缓冲区(见前一节设备管理), 所以可以利用延迟写(所谓延迟写, 是指该缓冲区内容不是在向缓冲区读入动作完成后, 立即写回盘文件区, 而是直到该缓冲区被再次分配时才写回盘。这样, 在此期间如需要缓冲区内的信息时可直接从缓冲区查找。所以发延迟写命令时, 实际上并未启动外设向盘中进行数据传输) 功能直接从缓冲区中读, 而不必实际启动外设。其次 pipe 可以用二级存储作为数据缓冲区, 这样就不必考虑主存空间的开销。

pipe 既然是读、写两个进程间的数据传输, 所以这两进程间要互斥以防止两个进程同时对相应于 pipe 文件的活动 i 节点, 系统打开文件表的表目进行修改, 也为了每次只允许一个进程对 pipe 文件进行读、写操作(这与同步关系中的读、写问题相同, 见 4.3.4 节)。同时 还要考虑两进程间的同步, 因为 pipe 文件一次存放量不多于 4096 个字节, 所以写进程只能超前读进程最多为 4096 个字节。若超前多于 4096 个字节, 写者进程等待; 若 pipe 文件读空, 读者进程等待; 若整个 pipe 文件传输量大于 4096 个字节, 则分批传送。pipe 机构中的这些互斥、同步、缓冲区管理完全由系统自动进行, 对用户是透明的。用户在应用 shell 语言时, 可以十分方便而简捷地使用 pipe 机构所提供的功能, 直接把一个程序的输出作为另一个程序的输入, 而不必经过一个中间文件作为媒介。详细情况请参阅

13.8 节 shell 语言部分。

13.7 系 统 调 用

如同所有操作系统一样,用户进程在执行过程中常要求操作系统的服务,因而通过访管指令 SVC 而进入操作系统,所以访管指令是用户程序和操作系统间的唯一通路,用户进入操作系统是由访管指令引起的访管中断进行管理。而在 UNIX 系统中,用户程序(进程的用户态或用户进程)与系统程序(该进程的核心态或系统进程)间的唯一通路是陷阱指令 trap,用户请求系统服务要使用 trap 指令,并由陷阱处理程序来进行管理,以提供用户所要求的服务。UNIX 为用户提供了 41 种功能的服务,这 41 个系统调用说明如下。

1. 与进程管理有关的系统调用

fork	建立子进程
exec	执行一个文件
wait	等待子进程信息
rexit	终止本进程
sslep	睡眠一段时间(睡眠时间,秒)
ssig	设置软中断(软中断号,程序入口地址)
kill	发送软中断给同一终端上各进程
ptrace	跟踪子进程
sbreak	改变数据区大小

2. 与文件管理有关的系统调用

creat	建立文件(同时打开)
link	连接文件
open	打开文件
close	关闭文件
read	读文件
write	写文件
seek	读写指针定位
unlink	取消连接
smount	安装系统文件卷
sumount	拆卸文件卷
pipe	建立 pipe 文件
chdir	更改当前目录
chmod	修改文件属性
chown	改变文件的用户和同组用户名
mknod	建立一个特殊文件或目录文件
stat	按文件名取主、外存中的 i 节点内容
fstat	按文件内部名取主、外存的 i 节点内容

dup	为文件再取一个内部名
sgnc	文件系统转储

3. 其它系统调用

getuid	取当前进程的有效用户号、实际用户号
setuid	置当前进程的有效用户号、实际用户号
getgid	取当前进程组号(有效组号、实际组号)
setgid	置当前进程组号(同上)
getpid	取当前进程号
gtime	取日历时间
stime	设置日历时间
gtty	取终端设备表 tty
stty	置终端设备表 tty
times	读本进程运行时间
nice	设置 proc 的 p-nice 项
getswit	读出 sw 寄存器内容
profil	建立程序运行统计表

13.8 shell 语言简介

shell 是 UNIX 操作系统的命令语言,同时又是该命令语言的解释程序的简称。shell 作为语言来说,它既是终端上的用户与 UNIX 操作系统会话的语言,又可作为程序设计的语言,所以 shell 是用户与系统之间的接口,而且是一种比较高级、易被用户理解和使用的程序设计语言,它为用户提供了使用方便、功能强、又容易扩充的程序设计环境。

13.8.1 shell 的一般用法

1. 一般命令

一般命令是由一个命令名(附录中给出了 UNIX 的主要命令的命令名及其功能)和用空格分隔的多个参数名所组成(参数名可为空)。例如用户打入如下命令

```
pwp
```

该命令就显示出该用户的当前目录的路径名(由根目录开始的完整路径名)。又如

```
Is    - 1
```

Is 是命令名,它打印当前目录中的所有文件名。- 1 是参数,它要求该命令不但要打印当前目录中的文件名,而且还要给出每个文件的存取控制、联接数、文件主、文件大小和生成日期。

2. 后台命令

有时用户希望运行一个程序或 UNIX 命令,同时还想再通过终端做些别的工作。例如某用户想要得到当前工作目录中所有的目录和文件的清单,并且要把它们放到另一个文件“ dir ”中。在执行这个任务时,他若还想开始另外一个文件的编辑,这时可以用后台命

令使前一个任务转为后台方式运行,方法是在命令的末尾附加一个符号“&”。例如上述任务,可以打入

用户→　　Is> dir &↙

UNIX→　　76　　这是进程号

→　　$　　可以开始你的编辑工作

用户→　　ed　　letter↙

其中“>”是输出重新定向符,表示 Is 命令的输出送入文件“dir”。“↙”是回车符,“$”是 UNIX 准备就绪,你可以开始工作的提示符。上述命令序列使 UNIX 为用户创建了一个新的进程,该进程运行第一条命令并把 shell 的控制返回终端用户并继续做其它工作。但不是所有命令都可以在后台运行的,例如编辑文件的命令。附带要指出的是,后台进程(例中是进程号 76)是用户进程的子进程,它与父进程并行运行。

3. 输入输出重新定向

UNIX 的标准输入设备和输出设备均是终端设备,所以除非由用户作了改变,否则标准输入来自终端,而标准输出是返回到终端,有时你需要从另外的介质上进行输入和输出工作,此时 shell 提供了输入输出重新定向的功能。此功能允许用户指定一个文件代替输入或输出设备,并用符号“<”、“n”、“>”、“m”分别来表示输入和输出的重新定向,例如

Is- 1> file

表示把命令 Is 的输出送到名为 file 的文件中去,如果文件 file 原来不存在,则 shell 建立此文件。如果文件已存在,则新内容取代旧的。而

Is- 1m file

是把 Is 的输出附加到文件 file 的末尾。又如

WC< file

表示命令 WC 是从文件 file 输入,而不是从终端上输入,然后给出该文件 file 的行数,字数、字符数。

4. 管道线和过滤器

经常有这样的情况,要求把一个命令执行后(输出)生成的数据,传送到另一个命令中去作为其输入。这在一般操作系统中的执行办法是,建立一个临时文件,将第一个命令的执行结果输出到该临时文件,然后再将此临时文件作为第二条命令的输入文件。也就是说必须要建立一个临时文件来传送数据。而 UNIX 系统提供了管道线 pipe 机构来执行两个进程间的数据传送,shell 语言用符号“©¦”来代表 pipe 操作符来实现上述数据传送工作,即使得一个命令的输出是另一个命令的输入。例如

Is- 1©¦sort- r

等价于

Is- 1> file; sort- r< file

这里 Is 和 sort 是两个并行的子进程。此命令功能是把 Is 命令输出的目录名和文件名传送给一个排序子程序 sort,参数- r 表示反向排序。

管道线可以包括多于两个的命令,例如

Is©¦sort©¦WC

过滤器是这样一种命令,它读入标准输入,按某种方式转换,然后作为输出打印结果。例如排序命令 sort 就是一个过滤器,它接受标准输入,根据你提供的参数去排序,并且输出结果。

5. 元字符的引用

许多命令接受文件名作为参数, shell 提供一种机构,能够生成与某种图式匹配的文件名清单。实现这个功能的元字符是“ * ”。例如

Is- 1* A

列出当前目录表中所有用 A 结尾的文件名字。通常元字符“ * ”有如下三种图式:

* 单一的元字符可匹配任何字符串、空串;

? 匹配该符号位置上的任何一个字符;

[…] 匹配其中任一个字符,如用横杠“ — ”相连,则匹配它们之间任一字符,例如:

[a- f]*

将选择以 a 到 f 范围内的字母打头的文件名。

13.8.2 shell 过程的用法

有时我们可能要不断地打入一个包含若干条命令的命令序列,以完成某个特定的任务。如果我们要多次反复地使用这组命令序列时,就应该使用 shell 过程(或称 shell 文件),这样每当我们要使用该功能时,就可用打入该 shell 过程名(或 shell 文件名)就能执行该过程而不必再逐条地打入这个命令序列。

1.shell 过程

一个 shell 过程是由能执行一个特定任务的一条或多条命令组成。shell 过程本身是个文件,所以又称 shell 文件,也有文件名。该文件也是通过编辑程序产生的,由用户在终端前打入命令并同编辑文件一样地进行编辑。它与文件不同之处在于只要打入文件 名就能执行这个文件。调用 shell 过程可以提供参数,一个 shell 文件最多可使用九个参数,我们可以用位置参数 $1, $2, …, $9 来访问相应的参数。

shell 过程被建立后,我们能用两种方法执行它:

(1) 使用“ sh ”命令,后面跟我们建立的 shell 过程名。例如 sh file[参数…]这实际上是递归地调用 shell。

(2) 使用 chmod 命令将我们建立的 shell 过程(shell 文件) 的存取控制改为“ 可执行 ”后,则不使用 shell 命令就能执行它(即不需在过程名前加“ sh ”)。例如 file [参数…]。

2. shell 变量

shell 提供字符串赋值的变量。变量以字母开头,由字母、数字和下线所组成。变量可以如下方式赋值,例如

user= wan

把值 wan 赋给变量 user。

对于经常使用的字符串可以用变量加以缩写,例如

b= /user/ fred/ bin

用变量 b 代替路径名/ user/ fred/ bin。

3. 控制流 if

条件转移的一般形式是　　if 命令表 1　　then 命令表 2　　else 命令表 3

if　　它检测后面命令表 1 中最后一条一般命令的值, 如执行成功(返回 0)则执行命令表 2, 否则执行命令表 3。

除控制流 if 用以控制转移外, 还有控制流 case 根据条件控制应执行的命令表, shell 还提供控制流 for 和 whil 2 以控制循环。在此不一一介绍了。

附　　录

UNIX 的主要命令表(shell 命令)

命令名	功　　能	命令名	功　　能
ar	调库维护程序以删除、取代、读出库文件。	pr	打印文件
/etc/mknod	建立特别文件	rm	消除文件
chgrp	改变用户组	sort	排序或合并文件
chmod	改变存取控制权	find	寻找文件
chown	改变文件主	pwd	给出当前工作目录路径名
chri	消除 i 节点	who	给出系统中所有注册用户名字
cmp 或 comm	比较两个文件	write 或 wall	写给某用户或所有用户
cat	串联文件	WC	计算文件中行数、字数、字节数
/etc/mkfs	构造一个文件系统	tee	管道安装
cp	复制文件	ps	列出所有进程或与终端有关进程
file	确定文件类型	kill	结束一进程
df	磁盘空闲空间	stty	设置终端
/etc/umount	拆卸文件系统	sleep	中止执行一段时间
grep	在文件中进行匹配的查寻	nice	以低优先权运行一条命令
nchech	根据 i 节点产生路径名	date	打印或设置日期
tty	取终端名	mail	发送与接收信件
/etc/mount	安装文件系统	dump	增量文件系统转储
Is	列出目录内容	restore	增量文件系统恢复
mv	移动文件		

第 14 章　CP/M 操 作 系 统

14.1　CP/M 操作系统概述

14.1.1　概述

CP/M(Control Program/Monitor)是为 8 位微型计算机设计的一个操作系统。CP/M 与 UNIX 这两个操作系统是当今世界上较为流行、使用较为广泛的两个操作系统,被许多不同类型的厂商所选用。CP/M 是由美国数字研究公司(Digital Research Company)于 1976 年研制成的,用于 Intel 8080 及 Z-80 微处理机上作为监控程序,它是一个单用户单任务批处理的操作系统,其系统软件完全适应于 70 年代后期出现的个人计算机,它为用户提拱了较强的使用功能和灵活方便的使用环境,所以得到了广泛应用。近年来随着微型计算机以爆炸性的速度向前发展,相继出现了 16 位机和 32 位机,这就迫使数字研究公司迅速将 CP/M 应用到 16 位微型计算机上,以便与其它操作系统(如 UNIX, MS -DOS 等)竞争,因此 CP/M 操作系统版本不断更新,迄今为止 CP/M 版本计有: CP/M-86, 并发 CP/M-86, MP/M, CP/NET 等。

CP/M-86 是 1981 年设计的,它是为 16 位微型计算机设计的单用户单作业的操作系统。

并发 CP/M-86 是在 CP/M-86 基础上设计的新版本,可同时执行几个实时应用程序。

MP/M 是与 CP/M-86 兼容的多用户多作业磁盘操作系统,它提供实时和分时处理的能力。

CP/NET 是计算机网络操作系统,是 MP/M 与 CP/M 组合在一起的系统,主机由 MP/M 操作,而从机与主机的通信由 CP/M 来完成。

14.1.2　CP/M 操作系统的功能和特性

CP/M 作为一个单用户单作业的操作系统来说,在典型的操作系统所具有的四大资源管理功能中,它具有较完善的文件管理和设备管理的功能,而在存储管理和处理机管理上则十分简单。尽管如此,它已为其用户提供了方便而又足够强的功能。同时 CP/M 采用

层次结构,可靠、易懂,便于用户学习。除此之外,CP/M 具有很强的兼容性,它十分便于移植、修改,适应性很强。这是由于 CP/M 的设计者将它分为两部分:不变部分和可变部分。不变部分是用高级语言 PL/M 写的,主要是磁盘操作系统 BDOS;可变部分是用汇编语言写的,主要内容是输入输出设备管理,这是要根据具体机器的配置来编写的。这样的分法使 CP/M 很容易移植到别的机器上,只要把可变部分根据具体机器的配置加以修改即可。这些优点使得 CP/M 受到用户广泛的欢迎。

14.2 CP/M 的结构

CP/M-80 与 CP/M-86 都由三个主要的子系统构成:

控制台命令处理程序 CCP;

基本输入输出系统 BIOS;

基本磁盘操作系统 BDOS。

控制台命令处理程序 CCP 负责解释用户输入的命令,并将这些命令传送给操作系统的有关部分,主要将大量调用 BDOS 中的功能模块,而后从控制台(CRT 或 TTY)给用户以回答,所以 CCP 是用户与 CP/M 操作系统其它主要部分的外层接口。

基本磁盘操作系统 BDOS 包括各种管理磁盘的实用程序。如对磁盘文件管理和调用,对磁盘空间的分配和释放等。所以它是一般操作系统中的文件管理系统(包括盘空间管理)。

基本输入输出系统 BIOS 是操作系统直接与机器硬件联系的部分,它的功能是提供软盘存取和系统其它设备的数据传输和输入输出操作成功否的状态信息数据的传送。

CP/M 操作系统采用层次结构,主要分为三层,如图 14.1 所示。

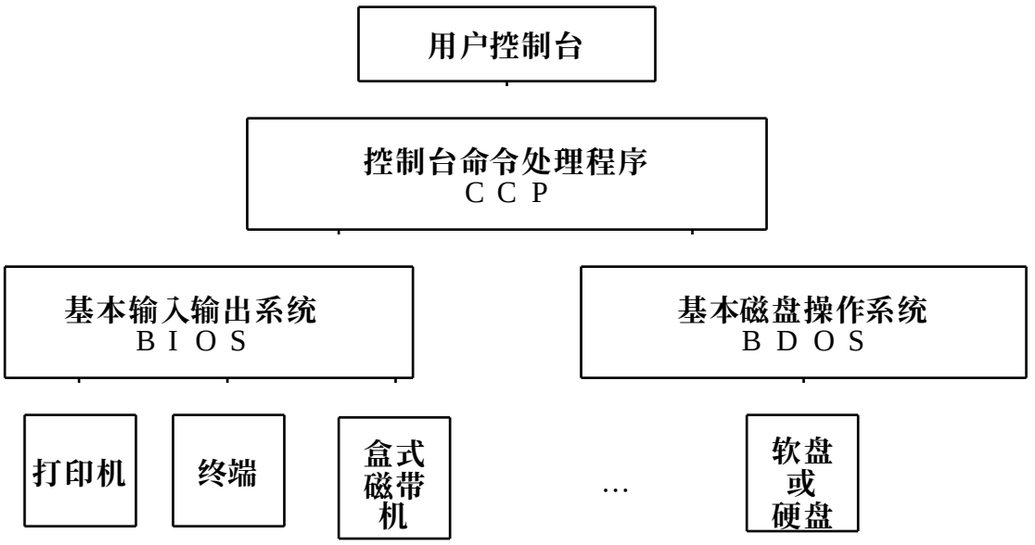


图 14.1 CP/M 的结构

14.3 主存分配

CP/M 是一个实存系统而不是虚存系统,它的内存分配如图 14.2 所示。

CP/M -80 最前面的 256 个字节(第 0 页)保存系统参数和作为缓冲,操作系统放在高地址部分,其余部分称为临时程序区 TPA,可用来存放用户程序,它由 0 页后面开始存放。

CP/M-86 的最前面(低地址)1K 空间是系统参数区,它包括中断向量,系统缓冲区以及系统与 TPA 公用区。在系统参数区之上是 CCP 和基本磁盘操作系统 BDOS,再往上是 BIOS ,这三部分构成了系统常驻部分,并以名为 CPM · SYS 的文件形式存在软盘上。

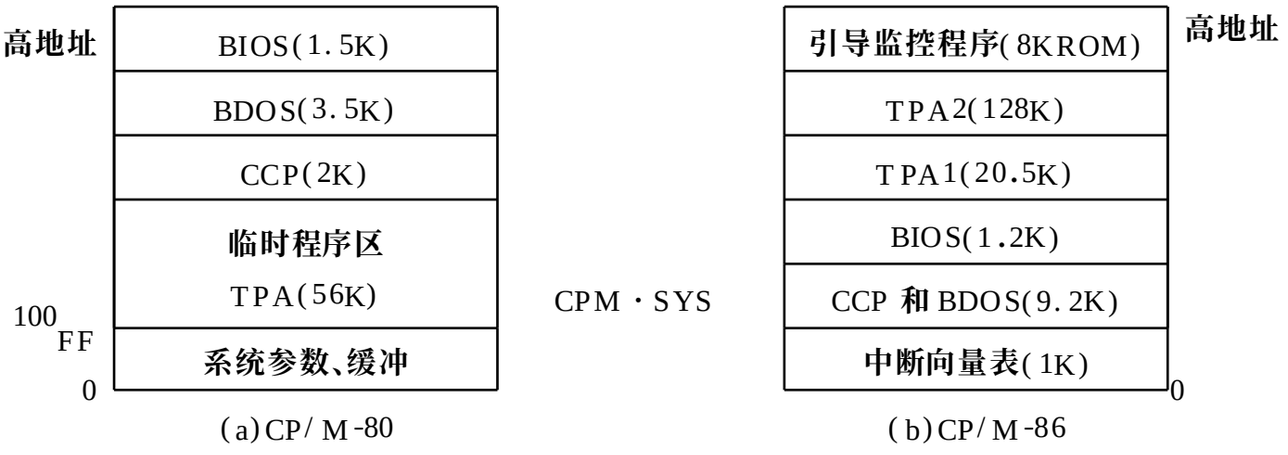


图 14.2 CP/M 的主存安排

14.4 控制台命令处理程序 CCP

控制台命令处理程序检查用户打入的命令行,进行一些简单的合法性检查,然后调用 BDOS 和 BIOS 的相应功能以完成该命令的要求。

CP/M 主要有如下命令:

ASM	汇编一个程序
DDT	动态查错手段,它可以显示主存内容;从某地址开始执行程序;在说明的地址处建立检查点;反汇编;在两个主存区之间传送主存内容;从盘中读一文件到主存;交换主存内容;执行一定数量的指令后停止并显示寄存器内容;显示寄存器内容;交换寄存器内容;显示目录
DUMP	用十六进制显示一个文件
ED	编辑一个文件(包括定位到文件开始处或结尾处);传送或删除一个字符;找出或更换一个字符串;删除或添加一行;打印某些行;复制库文件;停止编辑和返回 CP/M 等功能
ERA	删除一个文件
LOAD	装入一个程序
MOVCPM	构造一个 CPM 的新的副本

REN	重新给一个文件命名
SAVE	保存一个文件
STAT	显示软盘、文件、设备等状态
SUBMIT	批处理命令
DIR	显示目录
TYPE	打印文件
FORMAT	磁盘格式化
PIP	执行不同文件的应用操作。包括复制文件;连结两个文件;传送文件到控制台打印;把大写字母转换成小写字母或相反;在文件的每一行前加行号;打印文件时使每 n 行为一页,m 列作为一栏等
SYSGEN	生成 CP/M
USER	建立用户号

CP/M 的 CCP 分为两部分:内部命令和外部命令。内部命令是操作系统中常驻主存部分,随操作系统一起进入 RAM;内部命令包括: DIR, ERA, RENAME, SAVE 和 TYPE。外部命令做成文件存于磁盘上,使用时临时调入 RAM 运行。

14.5 基本输入输出系统 BIOS

基本输入输出系统 BIOS 主要管理外部设备的输入输出工作,实际上是设备驱动(管理)程序层,这一层是操作系统与硬件部分的接口,是 CP/M 中根据具体设备的不同而进行变化的可变部分。而 BDOS 和 CCP 是与硬件无关的不变部分。因此 CP/M 适应性强,当硬件环境变化时只需改变第三层 BIOS 即可。

为了使用户编程时方便,CP/M 的设备管理提供了设备独立性的功能,用户在编程时使用设备的逻辑名,而不是直接对物理设备进行操作。用户的逻辑设备转换成物理设备的工作由操作系统自动进行。这使 CP/M 与 MP/M(多用户多道作业的操作系统)兼容更为方便。如果 CP/M 使用输入输出说明字节 IOBYTE 功能时,用户的逻辑设备不但与物理设备台号无关,而且还与设备类型无关,即一个逻辑设备可按 IOBYTE 中的内容变换成相应的不同类型的物理设备。

CP/M 提供的逻辑设备有:

(1) CON:用户与 CP/M 之间的低速通信设备。

CONIN	从控制台输入设备一次输入一个字符
CONOUT	从控制台输出设备一次输出一个字符
CONST	检查控制台输入设备的字符是否已准备完毕等待传输

(2) RDR: 从大容量存储设备中输入的逻辑阅读设备。

(3) PUN: 逻辑穿孔设备,从大容量存储设备中进行输出。

(4) LST: 逻辑列表设备,通常是输出到打印机,但有时也输出到大容量存储设备(如盘)。

基本输入输出系统 BIOS 可以从控制台(通常是键盘)读入一个字符;向控制台(屏幕

或打印机)写一个字符; 从逻辑阅读设备读一字符, 向逻辑穿孔设备写一字符, 向打印机写一字符; 向(或从)控制台写(或读)一字符串; 建立或读输入输出状态, 判定控制台是否就绪。由此可见 BIOS 仅管理最基本的输入输出操作。

基本磁盘操作系统 BDOS 管理全部磁盘的输入输出工作, 用户不能通过命令直接使用 BDOS 功能。BDOS 的主要功能是: 恢复磁盘系统; 选择盘; 打开一个文件; 关闭一个文件; 查找文件目录; 查找文件的下一个数据段; 删除一个文件; 顺序地读一个文件; 顺序写一个文件、建立一个文件、对文件重新起名; 决定哪个盘驱动器是可用的; 现行盘驱动器是哪一个; 写盘的保护和决定哪个盘被保护; 建立文件属性; 读盘参数块的地址; 随机地对盘进行读和写; 计算文件大小; 读盘分配向量地址等。实际上 BDOS 完成 CP/M 命令的系统调用的主要功能, 所以又称系统调用层。BDOS 又是主要完成通常操作系统中的文件管理的功能。

14.6 CP/M 文件系统

CP/M 的文件系统在结构上分为三层: 命令处理; 文件管理和设备驱动。而在文件管理中 CP/M 与 MP/M 则又分两个子层, 其结构如图 14.3 所示。

其中命令处理主要由控制台命令解释程序 CCP 负责, 其主要过程是当键入命令后, 命令处理程序首先查找内部命令, 查到后执行此命令子程序。若查不到, 则再去磁盘目录表中查找(在命令后加扩展名 COM)。找到后将此命令调入 RAM0100 地址处执行, 否则, 显示出错信息。而文件管理层涉及文件系统主要部分, 分别在以下几节中讨论。

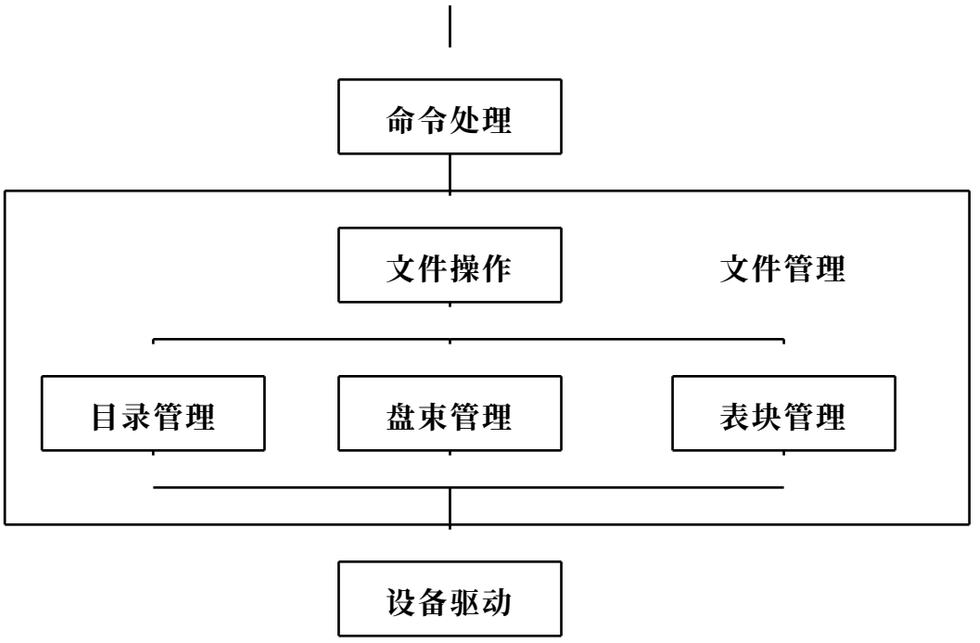


图 14.3 CP/M, MP/M 文件系统结构

14.6.1 CP/M 的文件组织和文件操作

相对于 UNIX 系统来说, CP/M 的文件逻辑组织是一个有记录结构的顺序文件。其记录称为逻辑记录, 逻辑记录的大小是 128 个字节(对应于盘扇区的大小), 所以在 CP/M

中把文件定义为逻辑记录的集合。

文件的物理组织(在盘上存放的形式)是这样:盘的每个扇区大小为 128 个字节,一个逻辑记录对应于一个扇区。由文件的逻辑记录号映象为物理扇区是通过在该文件描述符或文件控制块(FCB)中的映象表进行的,该映象表包含了所有属于该文件的扇区。

CP/M 中一个文件最大不能超过 256KB(略大于一张 IBM 3740 8 英寸盘的容量)。CP/M 为一个文件的每 16KB 都设置一个文件控制块 FCB,所以一个文件可以有多达 16 个 FCB。每个 FCB 占 33 个字节,它们被存放在磁盘的目录区(目录区放在磁盘文件区的开始处)。当文件被使用时,该文件的 FCB 被装入主存的临时程序区 TPA。当文件被关闭时,该文件的 FCB 被复制回磁盘。FCB 的格式如表 14.1 所示。

表 14.1 FCB 的格式

场标识	字节位置	意义
ET	0	所选驱动器编号,0 代表选现用盘;1 代表选 A 盘;2 代表选 B 盘;……
FN	1～8	文件名
FD	9～11	文件扩展名
EX	12	同一文件所占目录项编号
	13～14	未使用
RC	15	文件所占记录数
DM	16～31	该文件所在的束号,即逻辑记录映象表
NR	32	下一个要读写的记录

一个连续文件的相邻记录在盘上存放时通常并不是放在相邻的物理扇区中。因为这样会使连续地读、写相邻逻辑记录的速度降低。这是由于每读一个物理记录(扇区)后系统还要做某些辅助工作(例如判定读、写的正确性,以及一些内务操作等)。如果下一个逻辑上连续的记录在物理上也连续地存于相邻的扇区时,当要读写下一个物理记录时,它已经越过了读写头的位置,必须再转一圈才能读写该物理记录。为此 CP/M 采取把两个相邻的逻辑记录间隔地安排在两个扇区中,其间隔为五个扇区。对于每个磁道 26 个扇区的磁盘来说,其逻辑记录和物理扇区的对应关系如图 14.4 所示。

逻辑记录号	物理扇区号																									
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
第一圈访问	1						2						3						4						5	
第二圈访问					6						7						8						9			
第三圈访问			10						11						12						13					
第四圈访问		14						15						16						17						18
第五圈访问						19						20						21						22		
第六圈访问				23						24						25						26				

图 14.4 逻辑记录与物理扇区的对应

CP/M 的文件操作是文件管理的第一层,文件操作时所使用的数据库是文件目录和文件控制块 FCB,主要的文件操作有:

(1) 建立文件:在磁盘目录表中寻找一个空目录项,将 FCB 中的文件名写入此目录

项。

(2) 删除文件: 将该文件的目录项和盘区释放。

(3) 查寻文件: 将目录表中的目录项每四个一组地读进 RAM 里的磁盘缓冲区(通常位于 0080～00FF),逐个目录项与 FCB 中的文件名进行比较是否匹配。

(4) 打开文件: 将该文件目录项的全部内容写进 RAM 中的 FCB。

(5) 关闭文件: 将 FCB 中内容写回盘目录表。

(6) 文件改名: 将目录项中该文件改成新名。

(7) 顺序读一个记录: 将文件中该记录 i 读入 RAM 中的磁盘缓冲区,将 FCB 中下一记录号改为 i+ 1,为读下一记录做好准备。

(8) 顺序写一记录: 将 RAM 中磁盘缓冲区中的 128 字节写入磁盘中该文件的第 i 个记录,并将 FCB 中下一记录号改为 i+ 1。

14.6.2 盘空间管理

软盘(以 8 英寸盘为例)通常包含 77 个磁道(磁道号 0～76),每个磁道有 26 个扇区,每个扇区大小为 128 个字节,故一个磁盘包含 256 256 个字节。0 道和 1 道保留给 CP/M 本身和一个小的引导装入程序使用。2 道的 16 个扇区作为该盘上所有文件的目录表,其它均作为用户文件区。磁盘空间分配如图 14.5 所示,整个磁盘空间分成系统区和文件区两个部分。

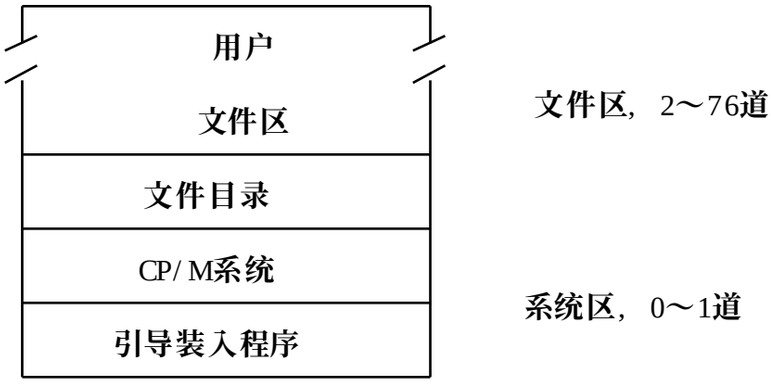


图 14.5 8英寸盘的分配

为了对磁盘空间进行管理,CP/M 对每个盘驱动器都使用一个 243 位的磁盘分配映象位示图(即以前所说的盘图或位示图),该图用以指出磁盘的哪些部分已分给文件使用,以及哪些部分尚未分配。CP/M 的磁盘管理是以“盘束”为单位的。所谓盘束是指 8 个为一组的逻辑上连续的扇区。所以盘束是一个逻辑单位而不是指 8 个物理上连续的扇区集合的这样一个物理单位(因为两个逻辑扇区间要有间隔)。所以位表上的每一位指相应的盘束的状态是已分配还是未分配。如已分配,则该位为 1,否则为 0。图 14.6 为分配映象位示图,由于一个盘的位示图的 243 位用十六进制表示为 F2,因此每个盘束号可用两位十六进制数表示,从 00 到 F2。

每当要求建立一个新的文件时,BDOS 的文件管理就查找该位示图,将第一个为零的盘束(图 14.6 中为 75)分给文件,并将该盘束号记入该文件的 FCB 中。

第 1 位数字	第 2 位 数 字															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

续表

第 1 位数字	第 2 位数字															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 14.6　分配映象位示图

14.6.3　目录管理

每一张磁盘都有一个目录表,它是一个一级线性表组织,放在文件区的开始处。8 英寸盘的目录表共有 64 个目录项,每个目录项的内容即文件控制块 FCB 的前 32 个字节。所以该目录表占 2K 个字节。

在 CP/M 和 MP/M 中,目录管理和盘束管理密切相关。当写入磁盘文件时,先从位示图中找到一个未分配的盘束,将文件的一部分记录写入这个盘束后,将该盘束号登记在该文件的目录项中。读文件时(从磁盘上)按目录项所指出的盘束号、将该文件读进 RAM。

14.6.4　表块管理

在 CP/M 和 MP/M 中,重要的表有:

(1) 系统参数表: 存放系统调用和目录查寻等参数。

(2) 磁盘参数表: 存放与磁盘有关的参数。

(3) 磁盘状态表: 存放当前磁盘的运行状态。

在系统调用和文件管理中将经常访问这些表。

14.7　MP/M 操作系统

MP/M 是 CP/M 系列的多用户多作业的操作系统,MP/M 是 Multi-Programming monitor for Microcomputer 的缩写,主要用于分时和实时处理。MP/M 有一个支持多个协同的顺序进程的操作系统内核和一个与 CP/M 兼容的文件管理,该文件管理具有终端管理的功能。

MP/M 系统使用了虚拟的输出 SPOOL 技术以打印各用户的输出(整个系统只有一台打印机)。

14.7.1 MP/M 的结构

它类似于 CP/M 的结构,由以下几部分组成:

(1) XIOS (BIOS 和扩充的 IOS): 它是与硬件的接口。

(2) XDOS(BIOS 和扩充的磁盘操作系统 DOS): 它包括文件操作、进程调度、队列管理、存储管理等。

(3) CLI: 命令行解释程序。

(4) TMP: 终端消息处理程序。

14.7.2 主存管理

MP/M 的主存分配如图 14.7 所示,其中:

图 14.7 MP/M 主存分配

(1) SYS·DAT: 系统数据区为 256 个字节,占主存最高一页,其中包括:

驻留系统进程链的链头地址;

XDOS 数据区头地址(XDOS 状态表 XST 的首址);

主存体分配表,主存段数;

断点向量表;

……

(2) CON·DAT: 控制台数据区,对应每个控制台终端有 256 个字节。

(3) USE·STK: 用户系统堆栈是任选的。

(4) 绝对 TPA: 每个用户主存段,包含一个绝对 TPA,TPA 的前 256 个字节称为系统参数区,与 CP/M 的相同。

MP/M 的存储管理为了克服 8 位微型计算机只能直接寻址 64K 的能力,采用选体分配的方法来扩充存储空间。MP/M 最多允许 8 个用户体,8 个用户体的物理地址空间范围

为 0000～7FFFH, 系统体的物理地址空间为 8000H～FFFFH。当用户进程被选中运行时, MP/M 将该用户体与系统体连接构成 64K 的存储空间。MP/M 驻留在系统体内。

14.7.3 进程的管理

MP/M 为每个进程建立一个进程描述块 PD(相当于进程控制块 PCB), 它由 52 个字节组成, 其中包括:

STATUS	进程状态字
PRIORITY	优先数
STKPTR	堆栈指针
NAME	进程名
CONSLE	进程所用的控制台号
MEMSEG	进程所用的主存段号
B	进程工作单元
THREAD	进程总链链接字
DISK$DMA	对磁盘输入输出的 DMA 地址
DISK$SLCT	使用的磁盘号/用户码
DRVACT	由进程存取的 16 位盘驱动器向量
REGISTERS	寄存器保存区
SEARCH	系统暂存字节

进程描述块 PD 中的进程状态字 STATUS 指出进程的 13 种状态, 每个进程可取这些状态之一, 这些状态为: 就绪、等某队列消息、等某队列缓冲区、等外设可用、等某标志建立、延迟某时间片后运行、终止进程、重新设置优先数、进程调度、控制台挂钩、控制台调度、重新挂钩控制台。

进程的基本状态有: 运行状态、就绪状态和挂起状态(即阻塞状态)。

MP/M 还对应进程的不同状态, 设置了以下七种类型的进程链:

(1) 就绪进程链: 按优先数排序。

(2) 延迟进程链: 按时间片增量方式组织, 链的排序按延迟时间片多少, 由少到多排序。时间片增量填入 PD 的 B 字节段。

(3) 进程调度链: 进程由阻塞链转到就绪链的过渡链, 新建进程也挂在此链。

(4) 查询外设进程链: 对所有外设的查询在此链等待, 以 FIFO 排序。

(5) 等控制台进程链: 当进程要使用某控制台进行输入输出时, 而该控制台被其它进程占用, 则该进程挂到对应控制台等待。MP/M 可支持 16 个控制台, 因此对应有 16 条控制台等待链。

(6) 等待队列缓冲区/消息进程链: 按优先数排序。

(7) 系统进程总链: 所有系统进程, 不管处于何种状态均挂在此链上。

14.7.4 进程调度

MP/M 的进程调度由进程调度程序负责。调度的原则是将处理机分给具有最高优先

数的就绪进程,优先数相同的进程则采用时间片轮转。进程的优先数是采用静态优先数。

系统配备了一个实时钟,每当时间片(20ms)到,就产生一次时钟中断。由进程调度程序调度就绪进程运行。实时钟还用于进程的延迟和定时调度。

14.7.5 进程的同步与通信

对于进程间的通信、同步和互斥均采用消息队列。每个队列有一个队列控制块 QCB,它包括以下信息:

QL	把各队列链接成队列链的链指针
NAME	队列名称
MSGLEN	消息长度(互斥队列时为零)
NMBMSG	最大消息数(互斥队列时为 1)
DQPH	消息队列的等待进程队列头指针,指向第一个进程的 PD 地址(读消息进程)
NQPH	消息队列缓冲区的等待进程队列头指针,这些进程要向缓冲区写入信息
MSG$IN	消息队列尾指针
MSG$OUT	消息队列头指针
MSG$CNT	消息队列中的消息个数
BUFFER	该消息队列大小,它等于消息长度与最大消息数的乘积(MSGLEN* NMBMSG)

MP/M 还设置了一条把各队列链接起来的队列链,其链首地址存放在 XST 表(XDOS 状态表)中。当用户要访问某队列时,按队列名称查找此链,从而找出该队列的 QCB,访问此队列。

MP/M 把打印机、磁盘驱动和语法分析模块看作临界资源。为了达到互斥使用的目的,MP/M 为它们各设置了一个互斥队列,其数据结构同上,但消息队列大小 BUFFER 只有两个字节,用以存放占用本互斥队列的进程的 PD 地址。消息个数 MSG$CNT 相当于二元信号量,其值为 1 时,表示该临界资源可用,为 0 时表示已被占用。进程要想使用此临界资源时,用'测试和设置'指令对 MSG$CNT 进行操作。若 MSG$CNT 的值为零,则该进程阻塞,将自己挂在 DQPH 等待队列上。反之进程使用该资源。当进程释放资源时,将 MSG$CNT 恢复为 1,并唤醒等待者进程。

14.7.6 SPOOL 系统

MP/M 的 SPOOL 是一个假脱机输出模块,它不具备假脱机输入功能。MP/M 的 Spool 命令允许用户将 ASCII 文件在列表设备上脱机输出。命令格式为

Spool 文件名 1,文件名 2,…

Spool 命令一个一个地将文件发送到 Spool 队列等待,直到由列表设备(通常为行式打印机)处理。Spool 队列按 FIFO 排序。

参 考 文 献

[1] Harvey M Deitel. An Introduction to Operating Systems. Addion-Wesley, 1983

[2] Alan C Shaw . The Logical Design of Operating Systems. Prentice-Hall, 1974

[3] Avi Silberschatz. Operating System Concepts. Addion-Wesley, 1983

[4] Per Brinch Hanse. Operating System Principles . Prentice-Hall, 1973

[5] Holt R C. Structured Concurrent Programming With Operating Systems Application. Addion-Wesley, 1978

[6] Lister A M. Fundamentals of Operating Systems. Macmillan Press Ltd, 1977

[7] Lions J. A Commentary on The UNIX Operating System. The University of New South Wales, 1977

[8] Yu-Cheng Liu, Glenn A. Gibson . Microcomputer Systems: The 8086/8088 Family. Prentice-Hall, 1984

[9] John F Wakerly . Microcomputer Architecture And Programming . Wiley, 1981

[10] Randall W Jensen, Charles C Tonies. Software Engineering, Prentice-Hall, 1979

[11] Helen Custer 著 . 程渝荣译. Windows NT 技术内幕. 清华大学出版社, 1993

[12] 马迪尼克 S E, JJ 多诺万著. 石伍锁等译. 操作系统. 科学出版社, 1981

[13] 张尤腊, 仲萃豪, 金庭湖, 曹东启编. 计算机操作系统. 科学出版社, 1979

[14] 王鸿武编. 操作系统. 湖南科技出版社, 1980

[15] 汤子瀛, 杨成忠编. 操作系统原理. 国防工业出版社, 1981