清华大学计算机系列教材

计算机操作系统教程

第2版

习题解答与实验指导

张尧学 编著

清华大学出版社
http://www.tup.tsinghua.edu.cn

计算机操作系统教程

（第 2 版）

习题解答与实验指导

张尧学　编著

清华大学出版社

(京)新登字 158 号

内 容 简 介

本书是作者在清华大学计算机系多年教学和科研的基础上，配合清华大学计算机系列教材之一的《计算机操作系统教程》(第 2 版)而编写的相关习题解答和实验指导。全书分为两大部分：第一部分是《计算机操作系统教程》(第 2 版)中各章习题的参考解答和部分硕士研究生考试用习题及解答；第二部分为清华大学计算机系操作系统课程教学用实验指导及相应的程序设计与源代码分析。实验主要设计在 Linux 环境下用 C 语言编程完成，但也可在 UNIX System V 或其他更高版本的 UNIX 环境下完成。

本书既可作为计算机专业和其他相关专业操作系统课程的补充教材，也可供有关人员自学，或供操作系统等系统设计人员阅读和参考。

书　名：计算机操作系统教程(第 2 版)习题解答与实验指导
作　者：张尧学　编著
出版者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http://www.tup.tsinghua.edu.cn
印刷者：北京国马印刷厂
发行者：新华书店总店北京发行所
开　本：787×1092 1/16　印张：9.25　字数：210 千字
版　次：2000 年 8 月第 1 版　2000 年 8 月第 1 次印刷
书　号：ISBN 7-302-04004-4/TP・2350
印　数：0001～6000
定　价：11.00 元

序　　言

计算机技术的飞速发展正在引发新的一轮世界性技术革命。在经济发展越来越全球化、科技创新越来越国际化、知识经济已初见端倪的今天，任何一门技术或任何一个领域离开了计算机恐怕是不可想象的。然而，计算机技术发展之迅速、计算机及其相关 IT 产品市场竞争之激烈、计算机产业让人致富速度之迅猛也同样是人们始料不及的。在新世纪，任何想在技术领域有一番作为的人，恐怕都不得不面对计算机技术的挑战。

软件技术是计算机系统的灵魂与核心，而操作系统更是计算机系统的大脑。“想发财，学软件！”在一些国家已成为深入人心的广告词。在我国，科技创新、高科技产业化的浪潮也势必会以雷霆万钧之力推动软件技术的迅猛发展与普及。21 世纪的哪一行哪一业能够离开软件呢？

学习计算机软件技术，特别是计算机操作系统技术，除了需要刻苦努力外，还需要掌握软件和操作系统的原理与设计技巧。这些原理与技巧可以说是计算机界的前辈们一代接一代不停顿的努力所留下的知识与智慧的结晶，学习和掌握它们对于激发自己的创造力和想象力是很有帮助的。

如何学习和掌握操作系统技术的原理与实际技巧呢？除了听课和读书之外，最好的方法恐怕就是在实践中练习。例如，自己设计一个小型操作系统，多使用操作系统，多阅读和分析操作源代码等。当前非常流行的 Linux 操作系统的原始版事实上也是一位优秀的大学生的练习之作。除了上述练习方法之外，习题和实验也是很重要的实践之一。

本书就是一本配合《计算机操作系统教程》(第 2 版)的习题解答与实验指导书。本书除给出《计算机操作系统教程》(第 2 版)各章所附习题的参考答案外，还给出一些相应的综合试题及其参考答案；另外还设计了 4 个在 Linux 环境下或 UNIX System V 以上版本的 UNIX 环境下的小实验，包括进程控制、进程通信、内存管理以及文件系统设计等，并给出了这 4 个实验的参考编程解答。

本书的编写得到了清华大学计算机系网络系统组的博士生王晓春、马洪军以及宋建平和段小平等同志的大力帮助和支持。他们对本书中所给出的许多习题进行了解答和完善，尽管作者在讲课过程中已多次讲解过这些习题的解答思路。在这里，作者向这些同志表示衷心的感谢！

本书虽然给出了《计算机操作系统教程》(第 2 版)一书中习题的参考解答和相关实验指导，但由于作者的水平与知识所限，这些解答只是一种参考，里面完全可能存在错误和不妥之处，有待于有识之士的指教。此外，还希望读者不要局限于这些解答。

衷心希望本书能对学习计算机操作系统和计算机软件的人们有所帮助！

作　者

2000 年 3 月于清华园

目　　录

第一部分　习题解答 …… 1
第 1 章　绪论 …… 1
第 2 章　操作系统用户界面 …… 3
第 3 章　进程管理 …… 7
第 4 章　处理机调度 …… 20
第 5 章　存储管理 …… 26
第 6 章　进程和存储管理示例 …… 32
第 7 章　文件系统 …… 39
第 8 章　设备管理 …… 45
第 9 章　文件系统和设备管理示例 …… 49
综合试题 …… 56
操作系统综合练习试题 1 …… 56
操作系统综合练习试题 1 解答 …… 58
操作系统综合练习试题 2 …… 60
操作系统综合练习试题 2 解答 …… 62
操作系统综合练习试题 3 …… 65
操作系统综合练习试题 3 解答 …… 66

第二部分　实验 …… 69
系统调用函数说明、参数值及定义 …… 69
实验 1　进程管理 …… 76
实验 2　进程间通信 …… 78
实验 3　存储管理 …… 79
实验 4　文件系统设计 …… 81
实验 1 指导 …… 82
实验 2 指导 …… 90
实验 3 指导 …… 94
实验 4 指导 …… 103

第一部分　习题解答

第1章　绪　　论

1. 什么是操作系统的基本功能？

答：操作系统的职能是管理和控制计算机系统中的所有硬、软件资源，合理地组织计算机工作流程，并为用户提供一个良好的工作环境和友好的接口。操作系统的基本功能包括：处理机管理、存储管理、设备管理、信息管理（文件系统管理）和用户接口等。

2. 什么是批处理、分时和实时系统？各有什么特征？

答：批处理系统（batch processing system）：操作员把用户提交的作业分类，把一批作业编成一个作业执行序列，由专门编制的监督程序（monitor）自动依次处理。其主要特征是：用户脱机使用计算机、成批处理、多道程序运行。

分时系统（time sharing operation system）：把处理机的运行时间分成很短的时间片，按时间片轮转的方式，把处理机分配给各进程使用。其主要特征是：交互性、多用户同时性、独立性。

实时系统（real time system）：在被控对象允许时间范围内作出响应。其主要特征是：对实时信息分析处理速度要比进入系统快、要求安全可靠、资源利用率低。

3. 多道程序（multiprogramming）和多重处理（multiprocessing）有何区别？

答：多道程序（multiprogramming）是作业之间自动调度执行、共享系统资源，并不是真正地同时执行多个作业；而多重处理（multiprocessing）系统配置多个CPU，能真正同时执行多道程序。要有效使用多重处理，必须采用多道程序设计技术，而多道程序设计原则上不一定要求多重处理系统的支持。

4. 讨论操作系统可以从哪些角度出发，如何把它们统一起来？

答：讨论操作系统可以从以下角度出发：（1）操作系统是计算机资源的管理者；（2）操作系统为用户提供使用计算机的界面；（3）用进程管理观点研究操作系统，即围绕进程运行过程来讨论操作系统。

上述这些观点彼此并不矛盾，只不过代表了同一事物（操作系统）站在不同的角度来看待。每一种观点都有助于理解、分析和设计操作系统。

5. 写出 1.6 节中巡回置换算法的执行结果。

答：1.6 节中的巡回置换算法要求：

设　i = 1,2,3,4,5,6,7

　　p[i] = 4,7,3,1,2,5,6

当　k∈[1…n]

　　k = P[…, p[k], …]。

从而有如下解：

(1) 算法

```
local x, k          /* x, k 为局部变量 */
Begin  k←1          /* 初始化 k */
  while  k<=7  do
    x←k
    repeat
      print(x)
      x←p[x]
    until x=k
    k←k+1
  od
End
```

(2) 打印结果：

k=1 时，置换过程为(1 4 1)

k=2 时，置换过程为(2 7 6 5 2)

k=1 时，置换过程为(3 3)

k=1 时，置换过程为(4 1 4)

k=1 时，置换过程为(5 2 7 6 5)

k=1 时，置换过程为(6 5 2 7 6)

k=1 时，置换过程为(7 6 5 2 7)

6. 设计计算机操作系统与哪些硬件器件有关？

答：计算机操作系统的重要功能之一是对硬件资源的管理。因此设计计算机操作系统时应考虑下述计算机硬件资源：

(1) CPU 与指令的长度及执行方式；

(2) 内存、缓存和高速缓存等存储装置；

(3) 各类寄存器，包括各种通用寄存器、控制寄存器和状态寄存器等；

(4) 中断机构；

(5) 外部设备与 I/O 控制装置；

(6) 内部总线与外部总线；

(7) 对硬件进行操作的指令集。

第 2 章　操作系统用户界面

1. 什么是作业？作业步？

答：把在一次应用业务处理过程中，从输入开始到输出结束，用户要求计算机所做的有关该次业务处理的全部工作称为一个作业。作业由不同的顺序相连的作业步组成。作业步是在一个作业的处理过程中，计算机所做的相对独立的工作。例如，编辑输入是一个作业步，它产生源程序文件；编译也是一个作业步，它产生目标代码文件。

2. 作业由哪几部分组成？各有什么功能？

答：作业由三部分组成：程序、数据和作业说明书。程序和数据完成用户所要求的业务处理工作，作业说明书则体现用户的控制意图。

3. 作业的输入方式有哪几种？各有何特点？

答：作业的输入方式有 5 种：联机输入方式、脱机输入方式、直接耦合方式、SPOOLING(Simultaneous Peripheral Operations Online)系统和网络输入方式，各有如下特点：

(1) 联机输入方式：用户和系统通过交互式会话来输入作业。

(2) 脱机输入方式：又称预输入方式，利用低档个人计算机作为外围处理机进行输入处理，存储在后援存储器上，然后将此后援存储器连接到高速外围设备上和主机相连，从而在较短的时间内完成作业的输入工作。

(3) 直接耦合方式：把主机和外围低档机通过一个公用的大容量外存直接耦合起来，从而省去了在脱机输入中那种依靠人工干预来传递后援存储器的过程。

(4) SPOOLING 系统：可译为外围设备同时联机操作。在 SPOOLING 系统中，多台外围设备通过通道或 DMA 器件和主机与外存连接起来，作业的输入输出过程由主机中的操作系统控制。

(5) 网络输入方式：网络输入方式以上述几种输入方式为基础，当用户需要把在计算机网络中某一台主机上输入的信息传送到同一网中另一台主机上进行操作或执行时，就构成了网络输入方式。

4. 试述 SPOOLING 系统的工作原理。

答：在 SPOOLING 系统中，多台外围设备通过通道或 DMA 器件和主机与外存连接起来，作业的输入输出过程由主机中的操作系统控制。操作系统中的输入程序包含两个独立的过程，一个过程负责从外部设备把信息读入缓冲区，另一个过程是写过程，负责把缓冲区中的信息送入到外存输入井中。

在系统输入模块收到作业输入请求后，输入管理模块中的读过程负责将信息从输入装置读入缓冲区。当缓冲区满时，由写过程将信息从缓冲区写到外存输入井中。读过程和写过

程反复循环，直到一个作业输入完毕。当读过程读到一个硬件结束标志后，系统再次驱动写过程把最后一批信息写入外存并调用中断处理程序结束该次输入。然后，系统为该作业建立作业控制块 JCB，从而使输入井中的作业进入作业等待队列，等待作业调度程序选中后进入内存。

5. 作业说明书和作业控制块有何异同？

答：作业说明书主要包含三方面内容：作业的基本描述、作业控制描述和资源要求描述。作业基本描述主要包括用户名、作业名、使用的编程语言名、允许的最大处理时间等。而作业控制描述则大致包括作业在执行过程中的控制方式，例如是脱机控制还是联机控制、各作业步的操作顺序以及作业不能正常执行时的处理等。资源要求描述包括要求内存大小、外设种类和台数、处理机优先级、所需处理时间、所需库函数或实用程序等。

而作业控制块是作业说明书在系统中生成的一张表格，该表格登记该作业所要求的资源情况、预计执行时间和执行优先级等。从而，操作系统通过该表了解到作业要求，并分配资源和控制作业中程序和数据的编译、链接、装入和执行等。

6. 操作系统为用户提供哪些接口？它们的区别是什么？

答：操作系统为用户提供两个接口，一个是系统为用户提供的各种命令接口，用户利用这些操作命令来组织和控制作业的执行或管理计算机系统。另一个接口是系统调用，编程人员使用系统调用来请求操作系统提供服务，例如申请和释放外设等类资源、控制程序的执行速度等。

7. 作业控制方式有哪几种？调查你周围的计算机的作业控制方式。

答：作业控制的主要方式有两种：脱机方式和联机方式。

脱机控制方式利用作业控制语言来编写表示用户控制意图的作业控制程序，也就是作业说明书。作业控制语言的语句就是作业控制命令。不同的批处理系统提供不同的作业控制语言。

联机控制方式不同于脱机控制方式，它不要求用户填写作业说明书，系统只为用户提供一组键盘或其他操作方式的命令。用户使用操作系统提供的操作命令和系统会话，交互地控制程序执行和管理计算机系统。

8. 什么是系统调用？系统调用与一般用户程序有什么区别？与库函数和实用程序又有什么区别？

答：系统调用是操作系统提供给编程人员的唯一接口。编程人员利用系统调用，在源程序一级动态请求和释放系统资源，调用系统中已有的系统功能来完成那些与机器硬件部分相关的工作以及控制程序的执行速度等。因此，系统调用像一个黑箱子那样，对用户屏蔽了操作系统的具体动作而只提供有关的功能。它与一般用户程序、库函数和实用程序的区别是：系统调用程序是在核心态执行，调用它们需要一个类似于硬件中断处理的中断处理机制来提供系统服务。

9. 简述系统调用的实现过程。

答：用户在程序中使用系统调用，给出系统调用名和函数后，即产生一条相应的陷入指令，通过陷入处理机制调用服务，引起处理机中断，然后保护处理机现场，取系统调用功能号并寻找子程序入口，通过入口地址表来调用系统子程序，然后返回用户程序继续执行。

10. 为什么说分时系统没有作业的概念？

答：因为在分时系统中，每个用户得到的时间片有限，用户的程序和数据信息直接输入到内存工作区中和其他程序一起抢占系统资源投入执行，而不必进入外存输入井等待作业调度程序选择。因此，分时系统没有作业控制表，也没有作业调度程序。

11. 试述 UNIX 的主要特点。

答：UNIX 的主要特点是：

(1) UNIX 系统是一个可供多用户同时操作的交互式分时操作系统；

(2) 为了向用户提供交互式功能和使得用户可以利用 UNIX 系统的功能，UNIX 系统向用户提供了两种友好的界面或接口：系统调用和命令；

(3) UNIX 系统具有一个可装卸的分层树型结构文件系统，该文件系统使用方便、搜索简单；

(4) UNIX 系统把所有外部设备都当成文件，并分别赋予它们对应的文件名。从而，用户可以像使用文件那样使用任一设备而不必了解该设备的内部特性，这既简化了系统设计，又方便了用户；

(5) UNIX 系统核心程序的绝大部分源代码和系统上的支持软件都用 C 语言编写。且 UNIX 系统是一个开放式系统，即具有统一的用户接口，使得 UNIX 用户的应用程序可在不同的执行环境下运行。

正是由于 UNIX 具有上述这些特点，使得 UNIX 系统得到了广泛的应用和发展。

12. UNIX 操作系统为用户提供哪些接口？试举例说明。

答：UNIX 系统为用户提供两个接口，即面向操作命令的接口 Shell 和面向编程用户的接口：系统调用。常见的 Shell 命令如：login，logout，vi，emacs，cp，rm，ls，cc，link，adduser，chown，dbx，date 等；常见的系统调用如：ioctl，read，write，open，close，creat，execl，flock，stat，mount，fork，wait，exit，socket 等。

13. 在你周围装有 UNIX 系统的计算机上，练习使用后台命令、管道命令等 Shell 的基本命令。

答：例 1：用 Shell 语言编制一 Shell 程序，该程序在用户输入年、月之后，自动打印输出该年该月的日历：

```
echo "Please input the month:"
read month
echo "Please input the year:"
read year
```

```
cal $month $year
```

例 2：用 Shell 语言编制一 Shell 程序，当用户输入要搜索的字符串之后，该程序从指定文件中搜索出有关字符串，且该搜索过程后台执行：

```
echo "Please input the string to be searched:"
read str
echo "Please input the filename:"
read filename
grep $str $filename&
```

14. Shell 变量和参数的作用是什么？操作系统默认定义的 Shell 变量有哪些？

答：正是因为有了 Shell 变量和参数，加上控制语句和条件语句，从而使 Shell 可以像一般程序语言那样进行编程，所不同的只是 Shell 编程的对象是系统命令，可以使得 Shell 命令文件中的命令按 Shell 编程所规定的顺序执行。

操作系统默认定义的 Shell 变量及定义如表 E1.1 所示。

表 E1.1

Shell 变量	定　义
$#	参数个数
$*，或 $@	Shell 的全部参数
$-	指定给 Shell 的操作集
$?	Shell 程序中最后执行的命令的返回值
$$	Shell 进程号
$!	最后被执行的命令的进程号
$HOME	cd 命令的标准参数值(HOME 目录)
$PATH	查找 Shell 命令文件的路径
$MAIL	指定接收邮箱的打印接收信息
$PS1	显示提示符
$argv[0]，$argv[1]，…，$argv[n]	输入参数变量

第3章 进程管理

1. 有人说，一个进程是由伪处理机执行的一个程序，这话对吗？为什么？

答：对。

因为伪处理机的概念只有在执行时才存在，它表示多个进程在单处理机上并发执行的一个调度单位。因此，尽管进程是动态概念，是程序的执行过程，但是，在多个进程并行执行时，仍然只有一个进程占据处理机执行，而其他并发进程则处于就绪或等待状态。这些并发进程就相当于由伪处理机执行的程序。

2. 试比较进程和程序的区别。

答：(1) 进程是一个动态概念，而程序是一个静态概念，程序是指令的有序集合，无执行含义，进程则强调执行的过程。

(2) 进程具有并行特征(独立性，异步性)，程序则没有。

(3) 不同的进程可以包含同一个程序，同一程序在执行中也可以产生多个进程。

3. 我们说程序的并发执行将导致最终结果失去封闭性。这话对所有的程序都成立吗？试举例说明。

答：并非所有程序均成立。

如：

```
Begin
    local x
    x := 10
    print(x)
End
```

上述程序中 x 是内部变量，不可能被外部程序访问，因此这段程序的运行不会受外部环境影响。

4. 试比较作业和进程的区别。

答：一个进程是一个程序对某个数据集的执行过程，是分配资源的基本单位。作业是用户需要计算机完成某项任务，而要求计算机所做工作的集合。一个作业的完成要经过作业提交、作业收容、作业执行和作业完成 4 个阶段。而进程是已提交完毕的程序所执行过程的描述，是资源分配的基本单位。其主要区别关系如下：

(1) 作业是用户向计算机提交任务的任务实体。在用户向计算机提交作业之后，系统将它放入外存中的作业等待队列中等待执行。而进程则是完成用户任务的执行实体，是向系统申请分配资源的基本单位。任一进程，只要它被创建，总有相应的部分存在于内存中。

(2) 一个作业可由多个进程组成。且必须至少由一个进程组成，但反过来不成立。

(3) 作业的概念主要用在批处理系统中。像 Unix 这样的分时系统中，则没有作业概念。而进程的概念则用在几乎所有的多道程序系统中。

5. UNIX System V 中，系统程序所对应的正文段未被考虑成进程上下文的一部分，为什么？

答：因为系统程序的代码被用户程序所共享，因此如果每个进程在保存进程上下文时，都将系统程序代码放到其进程上下文中，则大大浪费了资源。因此系统程序的代码不放在进程上下文中，而是统一放在核心程序所处的内存中。

6. 什么是临界区？试举一临界区的例子。

答：临界区是指不允许多个并发进程交叉执行的一段程序。它是由于不同并发进程的程序段共享公用数据或公用数据变量而引起的。所以它又被称为访问公用数据的那段程序。例如：

```
getspace:
        Begin local g
            g = stack[top]
            top =top - 1
        End
release(ad):
          Begin
            top = top + 1
            stack[top] = ad
          End
```

7. 并发进程间的制约有哪两种？引起制约的原因是什么？

答：并发进程所受的制约有两种：直接制约和间接制约。

直接制约是由并发进程互相共享对方的私有资源所引起的。间接制约是由竞争共有资源而引起的。

8. 什么是进程间的互斥？什么是进程间同步？

答：进程间的互斥是指：一组并发进程中的一个或多个程序段，因共享某一公有资源而导致它们必须以一个不许交叉执行的单位执行，即不允许两个以上的共享该资源的并发进程同时进入临界区。

进程间的同步是指：异步环境下的一组并发进程因直接制约互相发送消息而进行互相合作、互相等待，是各进程按一定的速度执行的过程。

9. 试比较 P，V 原语法和加锁法实现进程间互斥的区别。

答：互斥的加锁实现是这样的：当某个进程进入临界区之后，它将锁上临界区，直到它退出临界区时为止。并发进程在申请进入临界区时，首先测试该临界区是否是上锁的，如果

该临界区已被锁住，则该进程要等到该临界区开锁之后才有可能获得临界区。

但是加锁法存在如下弊端：(1) 循环测试锁定位将损耗较多的 CPU 计算时间；(2) 产生不公平现象。

为此，P，V 原语法采用信号量管理相应临界区的公有资源，信号量的数值仅能由 P，V 原语操作改变，而 P，V 原语执行期间不允许中断发生。其过程是这样的：当某个进程正在临界区内执行时，其他进程如果执行了 P 原语，则该进程并不像 lock 时那样因进不了临界区而返回到 lock 的起点，等以后重新执行测试，而是在等待队列中等待由其他进程做 V 原语操作释放资源后，进入临界区，这时 P 原语才算真正结束。若有多个进程做 P 原语操作而进入等待状态之后，一旦有 V 原语释放资源，则等待进程中的一个进入临界区，其余的继续等待。

总之，加锁法是采用反复测试 lock 而实现互斥的，存在 CPU 浪费和不公平现像，P，V 原语使用了信号量，克服了加锁法的弊端。

10. 设在书 3.6 节中所描述的生产者-消费者问题中，其缓冲部分为 m 个长度相等的有界缓冲区组成，且每次传输数据长度等于有界缓冲区长度以及生产者和消费者可对缓冲区同时操作。重新描述发送过程 deposit(data)和接收过程 remove(data)。

答：设第 I 块缓冲区的公用信号量为 mutex[I]，保证生产者进程和消费者进程对同一块缓冲区操作的互斥，初值为 1。设信号量 avail 为生产者进程的私用信号量，初值为 m。信号量 full 为消费者进程的私用信号量，初值为 0。从而有：

```
deposit(data)
    Begin
        P(avail)
        选择一个空缓冲区 i
        P(mutext[I])
        送数据入缓冲区 i
        V(full)
        V(mutext[I])
    End

Remove(data)
    Begin
        P(full)
        选择一个满缓冲区 I
        P(mutext[ I ])
        取缓冲区 i 中的数据
        V(avail)
        V(mutext[ I ])
    End
```

11. 两进程 P_A，P_B 通过两 FIF() 缓冲区队列连接(如图 E1.1)，每个缓冲区长度等于传

送消息长度。

进程 P_A,P_B 之间的通信满足如下条件:

(a) 至少有一个空缓冲区存在时,相应的发送进程才能发送一个消息。

(b) 当缓冲队列中至少存在一个非空缓冲区时,相应的接收进程才能接收一个消息。

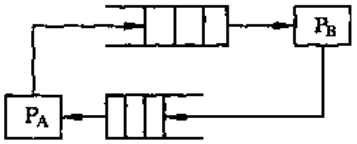


图 E1.1

试描述发送过程 send(i,m)和接收过程 receive(i,m)。这里 i 代表缓冲队列。

答:定义数组 buf[0],buf[1],bufempty[0],buffull[1]是 P_A 的私有信息量,buffull[0],bufempty[1]是 P_B 的私有信息量。

初始时:

```
bufempty[0] = bufempty[1] = n,(n 为缓冲区队列的缓冲区个数)
buffull[0] = buffull[1] = 0

send(I,m)
Begin
    local x
    P(bufempty[I])
    按 FIFO 方式选择一个空缓冲区
    buf[I](x)
    buf[I](x) = m
    buf[I](x)置满标记
    V(buffull[I])
End

Receive(I,m)
Begin
    Local x
    P(buffull[I])
    按 FIFO 方式选择一个装满数据的缓冲区 buf[I](x)
    m = buf[I](x)
    buf[I](x)置空标记
    V(bufempty[I])
End
PA 调用 send(0,m)和 receive(1,m)
PB 调用 send(1,m)和 receive(0,m)
```

12. 在和控制台通信的例中,设操作员不仅仅回答用户进程所提出的问题,而且还能独立地向各用户进程发出指示。对于这些指示,操作员不要求用户进程回答,但它们享有比其他消息优先传送的优先度。即如果 inbuf 中如有指示存在,系统不能进行下一次通信会话。试按上述要求重新描述 CCP 和 KCP,DCP。

答：KCP 描述如下：

设 T_Ready 和 T_Busy 分别为键盘 KP 和键盘控制进程 KCP 的私有信号量，其初值为 0 和 1。设 inbuf 为 inbuf 的共有信号量，初值为 1，表示其中没有控制消息。

初始化{清除所有 inbuf 和 echobuf}

```
Begin
    local x
    P(T_Ready)
    从键盘数据传输缓冲 x 中取出字符 m 记为 x.m
    if 为控制消息
    P(inbuf)
    Send(x.m)
    将 x.m 送入 echobuf
    V(T_Busy)
End
```

键盘控制进程 DCP：

```
repeat
    P(T_Busy)
    把键入字符放入数据传输缓冲
    V(T_Ready)
Until 终端关闭
```

显示其控制进程 DCP：

```
设 D_Ready 和 D_Busy 分别为 DP 和 DCP 的私有信号量且初值为 0 和 1.
初始化{清除输出缓冲 outbuf，echo 模式置 false}
Begin
    if outbuf 满
    then
        receive(k)
        P(D_Busy)
        把 k 送入显示器数据缓冲区
        V(D_Ready)
    else
    Else
        Echo 模式置 true
        Echobuf 中字符置入显示器数据缓冲区 fi
End
```

显示器动作 DP：

```
repeat
    if echo 模式
    then
```

```
        打印显示器数据缓冲区中字符
    else
        P(D_Ready)
        打印显示器数据缓冲区中消息
        V(D_Busy)
Until 显示器关机
```

设过程 Read(x)把 inbuf 中的所有字符读到用户进程数据区 x 处，过程 write(y)把用户进程 y 处的消息写到 outbuf 中. read(x)和 write(y)可分别描述如下：

```
read(x):            write(y)
    略                  略
```

13. 编写一个程序使用系统调用 fork 生成 3 个子进程，并使用系统调用 pipe 创建一管道，使得这 3 个子进程和父进程公用同一管道进行信息通信。

答：

```
main( )
{
    int i,r,p1,p2,fd[2];                          /* fd[2]为管道文件读写标识 */
    char buf[50],s[5];
    pipe(fd);                                     /* 创建管道 pipe( ) */
    while((p1 = fork(1) == -1);                   /* 创建子进程 1 */
    if(p1 == 0)                                   /* 在子进程 1 中执行 */
    {
        lockf(fd[1],1,0);                         /* 锁定写过程 */
        sprintf(buf,"child process P1 is seding message! \n");
        printf("child process P1! \n");
        write(fd[1],buf,50);                      /* 将 buf 中数据写入 pipe */
        sleep(5);                                 /* 睡眠等待父进程读出 */
        lockf(fd[1],0,0);                         /* 解锁 */
        exit(0);
    }
    else
    {
        while((p2 = fork( )) == -1);              /* 创建进程 2 */
        if(p2 == 0)                               /* 在子进程 2 中执行 */
        {
            lockf(fd[1],1,0);                     /* 锁定写过程 */
            sprintf(buf,"child process P2 is sending message! \n");  /* 数据入 buf */
            printf("child process P2! \n");
            write(fd[1],buf,50);                  /* buf 数据写入 pipe */
            sleep(5);                             /* 同步等待父进程读 */
            lockf(fd[1],0,0);                     /* 解锁 */
```

```
            exit(0);                                    /* 释放进程资源 */
        }
        else
        {
            while((p3 = fork( )) == -1);                /* 创建进程 3 */
            if(p3 == 0)                                 /* 在子进程 3 中 */
            {
                lock(fd[1],1,0);
                sprintf(buf,"child process P3 is sending message! \n");
                printf("child process P3! \n");
                write(fd[1],buf,50);
                sleep(5);
                lockf(fd[1],0,0);
                exit(0);
            }
            wait(0);                                    /* 父进程等待子进程先执行 */
            if(r = read(fd[0],s,50) == -1)              /* 读管道 pipe 内容到 s 中 */
                printf("can't read pipe\n");
            else
                printf("%s\n",s);
            wait(0);                                    /* 等待另一个子进程执行 */
            if(r = read(fd[0],s,50) == -1)
                printf("can't read pipe\n");
            else
                printf("%s\n",s);
            wait(0);                                    /* 等待最后一个子进程执行 */
            if(r = read( fd[0],s,50) == -1)
                printf("can't read pipe\n")
            else
                printf("%s\n",s);
            exit(0);
        }
    }
}
```

14. 设有 5 个哲学家，共享一张放有五把椅子的桌子，每人分得一把椅子。但是，桌子上总共只有五只筷子，在每人两边分开各放一只。哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。

条件：

(1) 只有拿到两只筷子时，哲学家才能吃饭。

(2) 如果筷子已在他人手上，则该哲学家必须等到他人吃完之后才能拿到筷子。

(3) 任一哲学家在自己未拿到两只筷子吃饭之前，决不放下自己手中的筷子。

试：

(1) 描述一个保证不会出现两个邻座同时要求吃饭的通信算法。

(2) 描述一个既没有两邻座同时吃饭，又没有人饿死(永远拿不到筷子)的算法。

在什么情况下，5 个哲学家全部吃不上饭？

答：(1) 设信号量 c[0]~c[4]，初始值均为 1，分别表示 I 号筷子被拿(I=0,1,2,3,4)，

```
send(I)：第 I 个哲学家要吃饭
Begin
  P(c[I])；
  P(c[I+1 mod 5])；
  Eat；
  V(c[I+1 mod 5])；
  V(c[I])；
End；
```

该过程能保证两邻座不同时吃饭，但会出现 5 个哲学家一人拿一只筷子，谁也吃不上饭的死锁情况。

(2) 解决的思路如下：让奇数号的哲学家先取右手边的筷子，让偶数号的哲学家先取左手边的筷子。

这样，任何一个哲学家拿到一只筷子以后，就已经阻止了他邻座的一个哲学家吃饭的企图，除非某个哲学家一直吃下去，否则不会有人会饿死。

```
send(I)：
Begin
  If I mod 2 == 0 then
  {
      P(c[I]，P(c[I+1] mod 5))
      Eat；
      V(c[I])；V(c[I+1 mod 5])
  }
  else
  {
      P(c[I+1 mod 5])
      P(c[I])
      Eat
      V(c[I+1 mod 5])
      V(c[I])
  }
End
```

15. 什么是线程？试述线程与进程的区别。

答：线程是在进程内用于调度和占有处理机的基本单位，它由线程控制表、存储线程上下文的用户栈以及核心栈组成。线程可分为用户级线程、核心级线程以及用户/核心混合型

线程等类型。其中用户级线程在用户态下执行，CPU 调度算法和各线程优先级都由用户设置，与操作系统内核无关。核心级线程的调度算法及线程优先级的控制权在操作系统内核。混合型线程的控制权则在用户和操作系统内核二者。

线程与进程的主要区别有：

(1) 进程是资源管理的基本单位，它拥有自己的地址空间和各种资源，例如内存空间、外部设备等；线程只是处理机调度的基本单位，它只和其他线程一起共享进程资源，但自己没有任何资源。

(2) 以进程为单位进行处理机切换和调度时，由于涉及到资源转移以及现场保护等问题，将导致处理机切换时间变长，资源利用率降低。以线程为单位进行处理机切换和调度时，由于不发生资源变化，特别是地址空间的变化，处理机切换的时间较短，从而处理机效率也较高。

(3) 对用户来说，多线程可减少用户的等待时间，提高系统的响应速度。例如，当一个进程需要对两个不同的服务器进行远程过程调用时，对于无线程系统的操作系统来说需要顺序等待两个不同调用返回结果后才能继续执行，且在等待中容易发生进程调度。对于多线程系统而言，则可以在同一进程中使用不同的线程同时进行远程过程调用，从而缩短进程的等待时间。

(4) 线程和进程一样，都有自己的状态，也有相应的同步机制，不过，由于线程没有单独的数据和程序空间，因此，线程不能像进程的数据与程序那样，交换到外存存储空间。从而线程没有挂起状态。

(5) 进程的调度、同步等控制大多由操作系统内核完成，而线程的控制既可以由操作系统内核进行，也可以由用户控制进行。

16. 使用库函数 clone( )与 pthread_create( )在 Linux 环境下创建两种不同执行模式的线程程序。

答： Linux 系统支持用户级线程和核心级线程两种执行模式，其库函数分别为 pthread_create( ) 和 clone( )。创建用户级线程和核心级线程的程序示例如下：

(1) 用户级线程编程示例：

```
#include <pthread.h>
void *ptest(void *arg)
{
    printf("This is the new thread! \n");
    return(NULL);
}

main( )
{
    pthread_t tid;
    printf("This is the parent process! \n");
    pthread_create(&tid,NULL,ptest,NULL);          /* 创建线程 */
    sleep(1);
```

```
    return;
}
```

该程序通过调用 pthread_create( )创建一个用户级线程,其指针为 tid,过程名为 ptest。

(2) 核心级线程编程示例:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#include <linux/unistd.h>

#define STACKSIZE 16384

#define CSIGNAL          0x000000ff      /* signal mask to be sent at exit */
#define CLONE_VM         0x00000100      /* set if VM shared between processes */
#define CLONE_FS         0x00000200      /* set if info shared between processes */
#define CLONE_FILES      0x00000400      /* set if files shared between processes */
#define CLONE_SIGHAND    0x00000800      /* set if signal handlers shared */

int show_same_vm;

void cloned_process_start_here(void * data)
{
        printf("child:\t got argument %d as fd\n", (int) data);
        show_same_vm = 5;
        printf ("child: \t vm = %d \n", show_same_vm);
        close((int)data);
}

int main ( )
{
        int fd, pid;

        fd = open (" /dev/null", O_RDWR);
        if ( fd < 0 ) {
                perror("/dev/null");
                exit(1);
        }
        printf (";mother: \t vm = %d\n",fd);

        show_same_vm = 10;
```

```
        printf ("mother: \t vm = %d\n", show_same_vm);

        pid = clone (cloned_process_starts_here, (void *) fd);
        if (pid < 0) {
            perror ("start_thread");
            exit(1);
        }

        sleep(1);
        printf ("mother: \t vm = %d\n", show_same_vm);
        if (write (fd, "c", 1) < 0)
            printf ("mother: \t child closed our file descriptor\n");
}

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#include <linux/unistd.h>

#define STACKSIZE 16384

#define CSIGNAL         0x000000ff      /* signal mask to be sent at exit */
#define CLONE_VM        0x00000100      /* set if VM shared between processes */
#define CLONE_FS        0x00000200      /* set if info shared between processes */
#define CLONE_FILES     0x00000400      /* set if files shared between processes */
#define CLONE_SIGHAND   0x00000800      /* set if signal handlers shared */

int clone (void (*fn) (void *), void *data)                    /* 创建核心线程 */
{
    long retval;
    void **newstack;
    /*
     *allocate new stack for subthread
     */
    newstack = (void **) malloc (STACKSIZE);
    if (!newstack)
        return -1;

    /*
     * Set up the stack for child function, put the (void *)
     * argument on the stack.
```

```
*/
newstack = (void **) (STACKSIZE + (char *) newstack);
*--newstack = data;

/*
  * Do clone ( ) system call. We need to do the low-level stuff
  * entirely in assembly as we're returning with a different
  * stack in the child process and we couldn't otherwise guarantee
  * that the program doesn't use the old stack incorrectly..
  *
  * Parameters to clone ( ) system call:
  *        %eax — _NR_clone, clone system call number
  *        %ebx — clone_flags, bitmap of cloned data
  *        %ecx — new stack pointer for cloned child
  *
  * In this example %ebx is CLONE_VM | CLONE_FS | CLONE_FILES |
  * CLONE_SIGHAND which shares as much as possible tetween parent and
  * child. (We or in the signal to be sent on child termination into clone_flags;
  * SIGCHLD makes the cloned process work like a "normal" unix child
  * process)
  *
  * The clone ( ) system call returns (in %eax) the pid of the newly
  * cloned process to the parent, and 0 to the cloned process. If
  * an error occurs, the return value will be the negative errno..
  * In the child process, we will doa "jsr" to the requested function
  * and then do a "exit ( ) " system call which will terminate the child.
  */
  __asm__volatile__(
    "int $0x80\n\t"          /* Linux/i386 system call */
    "testl %0, %0\n\t"       /* check return value */
    "jne 1f\n\t"             /* jump if parent */
    "call *%3\n\t"           /* start subthread function */
    "movl %2, %0\n\t"
    "int $0x80\n"            /* exit system call: exit subthread */
    "1: \t"
    : "=a" (retval)
    : "0" (__NR_clone), "i" (__NR_exit),
     "r" (fn),
     "b" (CLONE_VM | CLONE_FS | CLONE_FILES |
  CLONE_SIGHAND | CLONE_SIGCHLD),
     "c" (newstack));

  if (retval <0) {
```

```
        errno = -retval;
        retval = -1;
    }
    return retval;
}
```

第 4 章　处理机调度

1. 什么是分级调度？分时系统中有作业调度的概念吗？如果没有，为什么？

答：处理机调度问题实际上也是处理机的分配问题。显然只有那些参与竞争处理及所必需的资源都已得到满足的进程才能享有竞争处理机的资格。这时它们处于内存就绪状态。这些必需的资源包括内存、外设及有关数据结构等。从而，在进程有资格竞争处理机之前，作业调度程序必须先调用存储管理、外设管理程序，并按一定的选择顺序和策略从输入井中选择出几个处于后备状态的作业，为它们分配资源和创建进程，使它们获得竞争处理机的资格。另外，由于处于执行状态下的作业一般包括多个进程，而在单机系统中，每一时刻只能有一个进程占有处理机，这样，在外存中，除了处于后备状态的作业外，还存在处于就绪状态而等待得到内存的作业。我们需要有一定的方法和策略为这部分作业分配空间。因此处理机调度需要分级。

一般来说，处理机调度可分为 4 级：

(1) 作业调度：又称宏观调度，或高级调度。

(2) 交换调度：又称中级调度。其主要任务是按照给定的原则和策略，将处于外存交换区中的就绪状态或等待状态或内存等待状态的进程交换到外存交换区。交换调度主要涉及到内存管理与扩充。因此在有些书本中也把它归入内存管理部分。

(3) 进程调度：又称微观调度或低级调度。其主要任务是按照某种策略和方法选取一个处于就绪状态的进程占用处理机。在确立了占用处理机的进程之后，系统必须进行进程上下文切换以建立与占用处理机进程相适应的执行环境。

(4) 线程调度：进程中相关堆栈和控制表等的调度。

在分时系统中，一般不存在作业调度，而只有线程调度、进程调度和交换调度。这是因为在分时系统中，为了缩短响应时间，作业不是建立在外存，而是直接建立在内存中。在分时系统中，一旦用户和系统的交互开始，用户马上要进行控制。因此，分时系统中没有作业提交状态和后备状态。分时系统的输入信息经过终端缓冲区为系统直接接收，或立即处理，或经交换调度暂存外存中。

2. 试述作业调度的主要功能。

答：作业调度的主要功能是：按一定的原则对外存输入井上的大量后备作业进行选择，给选出的作业分配内存、输入输出设备等必要的资源，并建立相应进程，使该作业的相关进程获得竞争处理机的权利。另外，当作业执行完毕时，还负责回收系统资源。

3. 作业调度的性能评价标准有哪些？这些性能评价标准在任何情况下都能反映调度策略的优劣吗？

答：对于批处理系统，由于主要用于计算，因而对于作业的周转时间要求较高。从而作业的平均周转时间或平均带权周转时间被用来衡量调度程序的优劣。但对于分时系统来说，

平均响应时间又被用来衡量调度策略的优劣。

对于分时系统，除了要保证系统吞吐量大、资源利用率高之外，还应保证用户能够容忍的响应时间。因此，在分时系统中，仅仅用周转时间或带权周转时间来衡量调度性能是不够的。

对于实时系统来说，衡量调度算法优劣的主要标志则是满足用户要求的时限时间。

4. 进程调度的功能有哪些？

答：进程调度的功能有：

(1) 记录和保存系统中所有进程的执行情况；

(2) 选择占有处理机的进程；

(3) 进行进程上下文切换。

5. 进程调度的时机有哪几种？

答：进程调度的时机有：

(1) 正在执行的进程执行完毕。这时如果不选择新的就绪进程执行，将浪费处理机资源。

(2) 执行中进程自己调用阻塞原语将自己阻塞起来进入睡眠等待状态。

(3) 执行中进程调用了 P 原语操作，从而因资源不足而被阻塞；或调用了 V 原语操作激活了等待资源的进程队列。

(4) 执行中进程提出 I/O 请求后被阻塞。

(5) 在分时系统中时间片已经用完。

(6) 在执行完系统调用等系统程序后返回用户程序时，可看做系统进程执行完毕，从而调度选择一新的用户进程执行。

在 CPU 执行方式是可剥夺时，还有：

(7) 就绪队列中的某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

6. 进程上下文切换由哪几部分组成？描述进程上下文切换过程。

答：进程上下文切换由以下 4 个步骤组成：

(1) 决定是否作上下文切换以及是否允许作上下文切换。包括对进程调度原因的检查分析，以及当前执行进程的资格和 CPU 执行方式的检查等。在操作系统中，上下文切换程序并不是每时每刻都在检查和分析是否可作上下文切换，它们设置有适当的时机。

(2) 保存当前执行进程的上下文。这里所说的当前执行进程，实际上是指调用上下文切换程序之前的执行进程。如果上下文切换不是被那个当前执行进程所调用，且不属于该进程，则所保存的上下文应是先前执行进程的上下文，或称为“老”进程上下文。显然，上下文切换程序不能破坏“老”进程的上下文结构。

(3) 使用进程调度算法，选择一处于就绪状态的进程。

(4) 恢复或装配所选进程的上下文，将 CPU 控制权交到所选进程手中。

7. 为什么说在进程上下文切换过程中，上下文切换程序不能破坏“老”进程的上下文结构？

答：因为如果在进程上下文切换中破坏了老的进程上下文，等到CPU调度到该老进程执行时，就不能正确地恢复其停止执行前的状态了。

8. 假设有4道作业，它们的提交时间及执行时间由表E1.2给出。

表 E1.2

作业号	提交时刻(时)	执行时间(小时)
1	10：00	2
2	10：20	1
3	10：40	0.5
4	10：50	0.3

计算在单道程序环境下，采用先来先服务调度算法和最短作业优先调度算法时的平均周转时间和平均带权周转时间，并指出它们的调度顺序。

答：(1) 先来先服务调度：

顺序：
1. Ts1 = 10：00　Te1 = 12：00　T1 = 2.00　Tw1 = 0
2. Ts2 = 10：20　Te2 = 13：00　T2 = 1.00　Tw2 = 1.70
3. Ts3 = 10：40　Te3 = 13：30　T3 = 0.50　Tw3 = 2.30
4. Ts4 = 10：50　Te4 = 13：50　T4 = 0.30　Tw4 = 2.70

T = 0.25 * (2 +2.7 + 2.8 + 3) = 2.625 h

W= 0.25 * (4 +0 + 1.7/1 + 2.3/0.5 + 2.7/0.3) = 4.825

(2) 最短作业优先调度

顺序：
1. Ts4 = 10：50　Te4 = 10：80　T4 = 0.3　Tw4 = 0
2. Ts3 = 10：40　Te3 = 11：40　T3 = 0.5　Tw3 = 0.5
3. Ts2 = 10：20　Te2 = 12：40　T2 = 1　Tw2 = 1.3
4. Ts1 = 10：00　Te1 = 14：40　T1 = 2　Tw1 = 2.7

T = 0.25 * (0.3+1+2.3+4.7) = 2.075 h

W=0.25 * (4+0+1+1.3+2.7/2) = 1.9125

9. 设某进程所需要的服务时间 t=k * q，其中，k 为时间片的个数，q 为时间片长度且为常数。当 t 为一定值时，令 q 趋于 0，则有 k 趋于无穷，从而服务时间为 t 的进程响应时间 T 为 t 的连续函数。对应于时间片调度方式 RR、先来先服务方式 FCFS 和线性优先级调度方式 SRR，其响应时间函数分别为：

$Trr(t) = t * \mu/(\mu-\lambda)$

$Tfc(t) = 1/(\mu-\lambda)$

$Tsr(t) = 1/(\mu-\lambda) - (1 - t * \mu)/(\mu-\lambda')$

其中 $\lambda' = (1-b/a) * \lambda = r * \lambda$

取$(\lambda,\mu)=(50,100)$和$(\lambda,\mu)=(80,100)$，分别改变 r 的值，画出 Trr(t)、Tfc(t)和 Tsr(t)的时间变化图。

答：(1) 对$(\lambda,\mu)=(50,100)$，则

$Trr(t)=t$，　$Tfc(t)=t/50$，　$Tsr(t)=1/50-(1-100t)/(100-50r)$

$r\to 0$ 时，$Tsr(t)=1/100+t$；

$r\to 1$ 时，$Tsr(t)=2t$。

时间变化图如图 E1.2 所示。

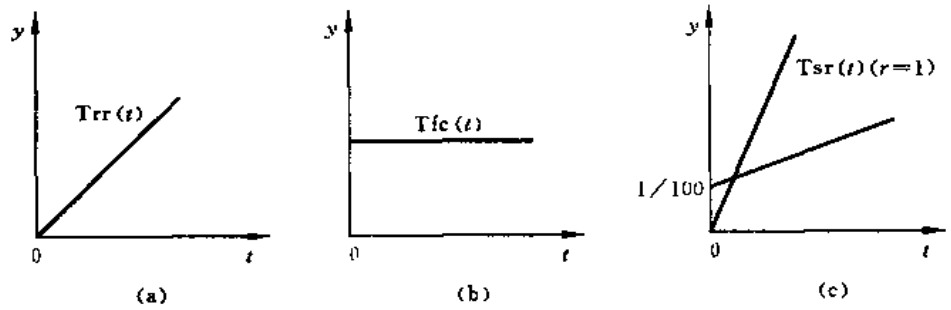


图　E1.2

只有 Tsr(t)受 r 值影响。且 r 增大时，Tsr(t)斜率增大，服务时间也增加。

(2) 对$(\lambda,\mu)=(80,100)$，有

$Trr(t)=5t$，　$Tfc(t)=1/20$，$Tsr(t)=1/20-(1-100t)/(100-80r)$

$r\to 0$ 时，$Tsr(t)\to 1/2s+t$；

$r\to 1$ 时，$Tsr(t)\to 5t$。

时间变化图如图 E1.3 所示。

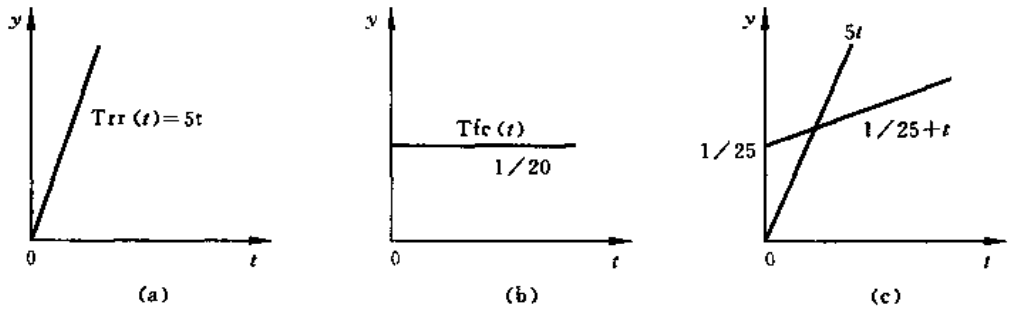


图　E1.3

Tsr(t)的斜率随 r 增大而增大，y 截距由 1/25 到 0 逐渐移动。

10. 什么是多处理机系统？并行处理系统、计算机网络、分布式系统和多处理机系统的操作系统之间有何区别？

答：从广义上说使用多台处理机协调工作，来完成用户所要求任务的计算机系统都是多处理机系统。狭义的多处理机系统是利用系统内的多个 CPU 来并行执行用户的几个程序，以提高系统的吞吐量；或用来进行冗余操作，以提高系统的可靠性。

并行处理机系统是利用多个功能单元(CPU)执行同一程序，多个处理机在物理位置上处于同一块电路板上。计算机网络系统则是通过物理通信媒介，包括有线和无线的，把现有的分散的计算机系统互相连接起来，以达到信息传递和资源共享的目的。分布式系统是以计算机网络为基础的，对用户来说是透明的。多处理机系统是指在同一计算机系统内共享内存的计算机系统。

11. 什么是实时调度？它与非实时调度相比，有何区别？

答：实时调度是为了完成实时处理任务而分配计算机处理器的调度方法。

实时处理任务要求计算机在用户允许的时限范围内给出计算机响应信号。实时处理任务可分为硬实时任务(hard real-time task)和软实时任务(soft real-time task)。硬实时任务要求计算机系统必须在用户给定的时限内处理完毕，软实时任务允许计算机系统在用户给定的时限左右处理完毕。

针对硬实时任务和软实时任务，计算机系统可以有不同的实时调度算法。这些算法采用基于优先级的抢先式调度策略，具体地说，大致有如下几类：

(1) 静态表驱动模式。该模式用于周期性实时调度，它在任务到达之前对各任务抢占处理机的时间进行分析，并根据分析结果进行调度。

(2) 静态优先级驱动的抢先式调度模式。该模式也进行静态分析。分析结果不是用于调度，只是用于给各任务指定优先级。系统根据各任务的优先级进行抢先式调度。

(3) 基于计划的动态模式。该模式在新任务到达后，将以前调度过的任务与新到达的任务一起统一计划，分配 CPU 时间。

(4) 动态尽力而为模式。该模式不进行任何关于资源利用率的分析，只检查各任务的时限是否能得到满足。

代表性的实时调度算法有两种。即时限式调度法(deadline scheduling)和频率单调调度法(rate monotonic scheduling)。

实时调度与非实时调度的主要区别是：

(1) 实时调度所调度的任务有完成时限，而非实时调度没有。从而，实时调度算法的正确与否不仅与算法的逻辑有关，也与调度算法调度的时限有关。

(2) 实时调度要求较快的进程或线程切换时间，而非实时调度的进程或线程的切换时间较长。

(3) 非实时调度强调资源利用率(批处理系统)或用户共享处理机(分时系统)，实时调度则主要强调在规定时限范围内完成对相应设备的控制。

(4) 实时调度为抢先式调度，而非实时调度则很少采用抢先式调度。

12. 写出图 4.11 所示周期性任务调度用的时限调度算法。

答：首先设置周期性任务进程的数据结构：

```
process  struct
         {
             int  p_num;          /* 进程号 */
             int  arr_time;       /* 进程到达时间 */
```

```
        int   exe_time;        /* 进程所需执行时间 */
        int   end_deadline;    /* 时限 */
        int   period;          /* 周期 */
        int   passed_time;     /* 该进程已占有处理机时间 */
    } pro[N],pro1[n];
local  int N,n,T1,T2;          /* N,n 为正整数,T1,T2 为进程周期 */
```

算法描述：

```
Begin
  if (n 个进程 pro1[n]到达,且要求调度)
  then {
    初始化 pro1[n]
    和 T1,T2 分别比较 pro1[n].period
    if (pro1[n].period ≠ T1 且 pro1[n].period ≠ T2)
    then {返回错误信息:该进程周期错误;
    else { 比较 pro1[n]与 pro[N]中的各进程时限
         选择 pro1[n].end_deadline 或 pro[N].deadline 中时间最近者占据处理机
           保护当前进程现场,并将其放入 pro[N]队列
           将未选中的 pro1[n]的其他进程置入 pro[N]
         }
  }
  else if 当前进程执行完毕要求调度
       {
         比较 pro[N]中每个进程的 end_deadline
         选择 end_deadline 最小的进程占有处理机
       }
  else
      等待占有处理机的进程执行完毕
End
```

13. 设周期性任务 P_1,P_2,P_3 的周期 T_1,T_2,T_3 分别为 100,150,350；执行时间分别为 20,40,100。问：是否可用频率单调调度算法进行调度？

答： 根据频率单调调度公式，能进行周期性调度的进程应满足下式：

$$c1/T_1 + c2/T_2 + \cdots + cn/T_n \leqslant n\,(2^{1/n} - 1)$$

在本式中，$n = 3, T_1 = 100, T_2 = 150, T_3 = 350$，而 $c1,c2,c3$ 分别等于 20,40,100，

从而有：$20/100 + 40/150 + 100/350 = 0.2 + 0.267 + 0.286 = 0.753$ (1)

而 $3*(2\ 1/3 - 1) = 0.779$ (2)

比较上述(1)与(2)，有 (1)≤(2)。

本题给出的周期性任务可以用频率单调调度算法进行调度。

第5章 存储管理

1. 存储管理的主要功能是什么？

答：存储管理的主要功能包括以下几点：

(1) 在硬件的支持下完成统一管理内存和外存之间数据和程序段自动交换的虚拟存储器功能。

(2) 将多个虚存的一维线性空间或多维线性空间变换到内存的唯一的一维物理线性地址空间。

(3) 控制内外存之间的数据传输。

(4) 实现内存的分配和回收。

(5) 实现内存信息的共享与保护。

2. 什么是虚拟存储器？其特点是什么？

答：由进程中的目标代码、数据等的虚拟地址组成的虚拟空间称为虚拟存储器。虚拟存储器不考虑物理存储器的大小和信息存放的实际位置，只规定每个进程中相互关联信息的相对位置。每个进程都拥有自己的虚拟存储器，且虚拟存储器的容量是由计算机的地址结构和寻址方式来确定。

实现虚拟存储器要求有相应的地址转换机构，以便把指令的虚拟地址变换为实际物理地址；另外，由于内存空间较小，进程只有部分内容存放于内存中，待执行时根据需要再调指令入内存。

3. 实现地址重定位的方法有哪几类？

答：实现地址重定位的方法有两种：静态地址重定位和动态地址重定位。

(1) 静态地址重定位是在虚空间程序执行之前由装配程序完成地址映射工作。静态重定位的优点是不需要硬件支持，但是用静态地址重定位方法进行地址变换无法实现虚拟存储器。静态重定位的另一个缺点是必须占用连续的内存空间和难以做到程序和数据的共享。

(2) 动态地址重定位是在程序执行过程中，在CPU访问内存之前由硬件地址变换机构将要访问的程序或数据地址转换成内存地址。动态地址重定位的主要优点有：

① 可以对内存进行非连续分配。

② 动态重定位提供了实现虚拟存储器的基础。

③ 动态重定位有利于程序段的共享。

形式化描述：略。

4. 常用的内存信息保护方法有哪几种？它们各自的特点是什么？

答：常用的内存保护方法有硬件法、软件法和软硬件结合保护法三种。

上下界保护法是一种常用的硬件保护法。上下界存储保护技术要求为每个进程设置一

对上下界寄存器。上下界寄存器中装有被保护程序和数据段的起始地址和终止地址。在程序执行过程中，在对内存进行访问操作时首先进行访问地址合法性检查，即检查经过重定位之后的内存地址是否在上、下界寄存器所规定的范围之内。若在规定的范围之内，则访问是合法的；否则是非法的，并产生访问越界中断。

保护键法也是一种常用的软件存储保护法。保护键法为每一个被保护存储块分配一个单独的保护键。在程序状态字中则设置相应的保护键开关字段，对不同的进程赋予不同的开关代码以和被保护的存储块中的保护键匹配。保护键可以设置成对读写同时保护的或只对读写进行单项保护的。如果开关字段与保护键匹配或存储块未受到保护，则访问该存储块是允许的，否则将产生访问出错中断。

另外一种常用的硬软件内存保护方式是：界限存储器与 CPU 的用户态，核心态相结合的保护方式。在这种保护方式下，用户态进程只能访问那些在界限寄存器所规定范围内的内存部分，而核心态进程则可以访问整个内存地址空间。

5. 如果把 DOS 的执行模式改为保护模式，起码应做怎样的修改？

答：如果要把 DOS 的执行模式改成保护模式，起码要为每一个进程设置一对上下界寄存器。上下界寄存器中装有被保护程序和数据段的起始地址和终止地址。在程序执行过程中，在对内存进行访问操作时首先进行访问地址合法性检查，即检查经过重定位之后的内存地址是否在上、下界寄存器所规定的范围之内。若在规定的范围之内，则访问是合法的；否则是非法的，并产生访问越界中断。另外，还应该把指令的访问内存模式由访问实际物理地址改为由逻辑地址变换为物理地址的方式。

6. 动态分区式管理的常用内存分配算法有哪几种？比较它们各自的优缺点。

答：动态分区式管理的常用内存分配算法有最先适应法(FF)、最佳适应法(BF)和最坏适应法(WF)。

优缺点比较：

① 从搜索速度上看最先适应法最佳，最佳适应法和最坏适应法都要求把不同大小的空闲区按大小进行排队。

② 从回收过程来看，最先适应法也是最佳，因为最佳适应法和最坏适应法都必须重新调整空闲区的位置。

③ 最佳适应法找到的空闲区是最佳的，但是会造成内存碎片较多，影响了内存利用率，而最坏适应法的内存碎片最少，但是对内存的请求较多的进程有可能分配失败。

总之，三种算法各有所长，针对不同的请求队列，它们的效率和功能是不一样的。

7. 5.3 节讨论的分区式管理可以实现虚存吗？如果不能，需要怎样修改？试设计一个分区式管理实现虚存的程序流程图。如果能，试说明理由。

答：5.3 节讨论的分区式管理不能实现虚存。如果要实现虚存，可以在分区的基础之上对每个分区内部进行请求调页式管理。

程序流程图：略。

8. 简述什么是覆盖？什么是交换？覆盖和交换的区别是什么？

答：将程序划分为若干个功能上相对独立的程序段，按照程序的逻辑结构让那些不会同时执行的程序段共享同一块内存区的内存扩充技术就是覆盖。

交换是指先将内存某部分的程序或数据写入外存交换区，再从外存交换区中调入指定的程序或数据到内存中来，并让其执行的一种内存扩充技术。

与覆盖技术相比，交换不要求程序员给出程序段之间的覆盖结构，而且，交换主要是在进程或作业之间进行，而覆盖则主要在同一个作业或同一个进程内进行。另外，覆盖只能覆盖那些与覆盖程序段无关的程序段。

9. 什么是页式管理？静态页式管理可以实现虚存吗？

答：页式管理就是把各进程的虚拟空间划分为若干长度相等的页面，把指令按页面大小划分后存放在内存中执行或只在内存中存放那些经常被执行或即将被执行的页面，而那些不被经常执行以及在近期内不可能被执行的页面则存放于外存中，按一定规则调入的一种内存管理方式。

静态页式管理不能实现虚存，这是因为静态页式管理要求进程或作业在执行前全部被装入内存，作业或进程的大小仍受内存可用页面数的限制。

10. 什么是请求页式管理？试设计和描述一个请求页式管理时的内存页面分配和回收算法(包括缺页处理部分)。

答：请求页式管理是动态页式内存管理的一种，它在作业或进程开始执行之前，不把作业或进程的程序段和数据段一次性的全部装入内存，而只装入被认为是经常反复执行和调用的工作区部分。其他部分则在执行过程中动态装入。请求页式管理的调入方式是，当需要执行某条指令而又发现它不在内存时，或当执行某条指令需要访问其他数据或指令时，而这些指令和数据又不在内存中，从而发生缺页中断，系统将外存中相应的页面调入内存。

请求页式管理的内存页面分配和回收算法：

略。

11. 请求页式管理中有哪几种常用的页面置换算法？试比较它们的优缺点。

答：比较常用的页面置换算法有：

(1) 随机淘汰算法(random glongram)。即随机地选择某个用户页面并将其换出。

(2) 轮转法 RR(round robin)。轮转法循回换出内存可用区内一个可以被换出的页，无论该页是刚被换进或已经换进内存很长时间。

(3) 先进先出法 FIFO(first in first out)。FIFO 算法选择在内存驻留时间最长的一页将其淘汰。

(4) 最近最久未使用页面置换算法 LRU(least recently unused)。该算法的基本思想是：当需要淘汰某一页时，选择离当前时间最近的一段时间内最久没有使用过的页面先淘汰。

(5) 理想型淘汰算法 OPT(optimal replacement algorithm)。该算法淘汰在访问串中将来再也不出现的或是在离当前最远的位置上出现的页面。

RR 和 FIFO 都是基于 CPU 按线性顺序访问地址空间这一假设，但是实际上 CPU 在很多时候并非是按线性顺序访问地址空间的，因而它们的内存利用率不高。此外 FIFO 算法还存在着 Belady 现象。LRU 算法的完全实现是相当困难的，因此在实际系统中往往要采取 LRU 的近似算法，常用的近似算法有最不经常使用页面淘汰算法 LFU(least frequently used)和最近没有使用页面淘汰算法(NUR)。OPT 算法由于必须预先知道每一个进程的指令访问串，所以它是无法实现的。

12. 什么是 Belady 现象？试找出一个 Belady 现象的例子。

答：Belady 现象是指在使用 FIFO 算法进行内存页面置换时，在未给进程或作业分配足它所要求的全部页面的情况下，有时出现的分配的页面数增多，缺页次数反而增加的奇怪现象。

例：假设进程 P 共有 5 页，程序访问内存的顺序(访问串)为 1，2，3，4，1，2，5，1，2，3，4，5。

当内存工作区页面为 3 时：

页面数 = 3　缺页次数 = 9　缺页率 = 9 / 12 = 75%

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	1	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4
√	√	√	√	√	√	√			√	√	

当内存工作区页面为 4 时：

页面数 = 4　缺页次数 = 10　缺页率 = 10 / 12 = 83.3%

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
√	√	√	√			√	√	√	√	√	√

由上例可知，在 FIFO 算法产生了 Belady 现象时，工作区页面增加反而使得缺页率变大。

13. 描述一个包括页面分配与回收、页面置换和存储保护的请求式存储管理系统。

答：略。

14. 什么是段式管理？它与页式管理有何区别？

答：段式管理就是将程序按照内容或过程(函数)关系分成段，每段拥有自己的名字。一个用户作业或进程所包含的段对应于一个二维线性虚拟空间，也就是一个二维虚拟存储器。段式管理程序以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换成实际的内存物理地址。同页式管理时一样，段式管理也采用只把那些经常访问的段驻留内存，而把那些在将来一段时间内不被访问的段放入外存，待需要时自动调入相关段的方法实现二维虚拟存储器。

段式管理和页式管理的主要区别有：

(1) 页式管理中源程序进行编译链接时是将主程序、子程序、数据区等按照线性空间的一维地址顺序排列起来。段式管理则是将程序按照内容或过程(函数)关系分成段，每段拥有自己的名字。一个用户作业或进程所包含的段对应于一个二维线性虚拟空间，也就是一个二维虚拟存储器。

(2) 同动态页式管理一样，段式管理也提供了内外存统一管理的虚存实现。与页式管理不同的是：段式虚存每次交换的是一段有意义的信息，而不是像页式虚存管理那样只交换固定大小的页，从而需要多次的缺页中断才能把所需信息完整地调入内存。

(3) 在段式管理中，段长可根据需要动态增长。这对那些需要不断增加或改变新数据或子程序的段来说，将是非常有好处的。

(4) 段式管理便于对具有完整逻辑功能的信息段进行共享。

(5) 段式管理便于进行动态链接，而页式管理进行动态链接的过程非常复杂。

15. 段式管理可以实现虚存吗？如果可以，简述实现方法。

答：段式管理可以实现虚存。

段式管理把程序按照内容或过程(函数)关系分成段，每段拥有自己的名字。一个用户作业或进程所包含的段对应于一个二维线性虚拟空间(段号 s 与段内相对地址 w)，也就是一个二维虚拟存储器。段式管理以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换成实际的内存物理地址。只把那些经常访问的段驻留内存，而把那些在将来一段时间内不被访问的段放入外存，待需要时产生缺段中断，自动调入。

16. 为什么要提出段页式管理？它与段式管理及页式管理有何区别？

答：因为段式管理和页式管理各有所长。段式管理为用户提供了一个二维的虚拟地址空间，反映了程序的逻辑结构，有利于段的动态增长以及共享和内存保护等，这极大地方便了用户。而分页系统则有效地克服了碎片，提高了存储器的利用效率。从存储管理的目的来讲，主要是方便用户的程序设计和提高内存的利用率。所以人们提出了将段式管理和页式管理结合起来让其互相取长补短的段页式管理。

段页式管理与段式和页式管理相比，其访问时间较长。因此，执行效率低。

17. 为什么说段页式管理时的虚拟地址仍是二维的？

答：因为在段页式内存管理中，对每一段内的地址空间进行分页式管理只是为了克服在内存分配过程中产生的大量碎片，从而提高存储器的利用效率，它并没有改变段内地址空

间的一维结构，所以段页式内存管理中的虚拟地址仍然和段式内存管理中的虚拟地址一样，是二维结构的。

18. 段页式管理的主要缺点是什么？有什么改进办法？

答：段页式管理的主要缺点是对内存中指令或数据进行存取时，至少需要对内存进行三次以上的访问。第一次是由段表地址寄存器取段表始址后访问段表，由此取出对应段的页表在内存中的地址。第二次则是访问页表得到所要访问的指令或数据的物理地址。只有在访问了段表和页表之后，第三次才能访问真正需要访问的物理单元。显然。这将大大降低CPU 执行指令的速度。

改进办法是设置快速联想寄存器。在快速联想寄存器中，存放当前最常用的段号 s，页号 p 和对应的内存页面地址与其他控制项。当需要访问内存空间某一单元时，可在通过段表、页表进行内存地址查找的同时，根据快速联想寄存器查找其段号和页号。如果所要访问的段或页的地址在快速联想寄存器中，则系统不再访问内存中的段表、页表而直接把快速联想寄存器中的值与页内相对地址 d 拼接起来得到内存地址。

19. 什么是局部性原理？什么是抖动？你有什么办法减少系统的抖动现象？

答：局部性原理是指在几乎所有程序的执行过程中，在一段时间内，CPU 总是集中地访问程序中的某一个部分而不是对程序的所有部分具有平均的访问概率。

抖动是指当给进程分配的内存小于所要求的工作区时，由于内存外存之间交换频繁，访问外存的时间和输入输出处理时间大大增加，反而造成 CPU 因等待数据而空转，使得整个系统性能大大下降。

在物理系统中，为了防止抖动的产生，在进行淘汰或置换时，一般总是把缺页进程锁住，不让其换出，从而防止抖动发生。

防止抖动发生的另一个办法是设置较大的内存工作区。

第 6 章　进程和存储管理示例

1. 简述 UNIX 系统进程的概念。

答：在 UNIX 系统中，进程被赋予一些特定的含义和特性：

(1) 一个进程是对一个程序的执行。

(2) 一个进程的存在意味着在所谓的"Proc"数组(PCB 的常驻内存部分)中有一个非零的结构存在，它包含着相应的进程控制信息。

(3) 对于每个进程，有一个被称为 U 区(userblock)或 User 的数据结构。这个数据结构放置该进程的私用控制信息，且在进程被创建时，才会由系统分配相应的域(field)。

(4) 一个进程可以生成或删除其子进程。

(5) 一个进程是获得和释放各种资源的基本单位。

2. UNIX System V 进程上下文由哪几部分组成？为什么说核心程序不是进程上下文的一部分？进程页表也在核心区，它们也不是进程上下文的一部分吗？

答：UNIX System V 进程上下文由以下几部分组成：

Proc 结构，User 结构，用户栈和核心栈的内容，用户地址空间的正文段和数据段，硬件寄存器的内容，区表，页表。

核心页表被所有进程共享，所以不是进程上下文的一部分。

而进程页表是进程上下文的一部分。

3. 假定在用户态下执行的某个进程用完了它的时间片，由于时钟中断的原因，核心调度一个新进程去执行。请形式化地描述出新、旧进程的上下文切换过程。

答：见图 E1.4。

4. UNIX System V 的调度策略是什么？调度时应该封锁中断吗？如果不封锁，会发生什么问题？

答：UNIX System V 采用基于优先级的多级轮转反馈调度策略。在调度时应封锁中断，否则在调度过程中由于中断会使进程上下文的切换出现错误。

5. 试述进程 0 的作用。

答：进程 0 的作用有：创建用户进程(init 进程)、进行进程的调度和交换。

6. UNIX System V 在哪几种情况下发生调度。

答：在 UNIX System V 中引起进程调度的情况有 5 种：

(1) 当前执行进程申请内存等系统资源未得到满足，从而自己调用 sleep 过程，放弃处理机而进入睡眠状态。

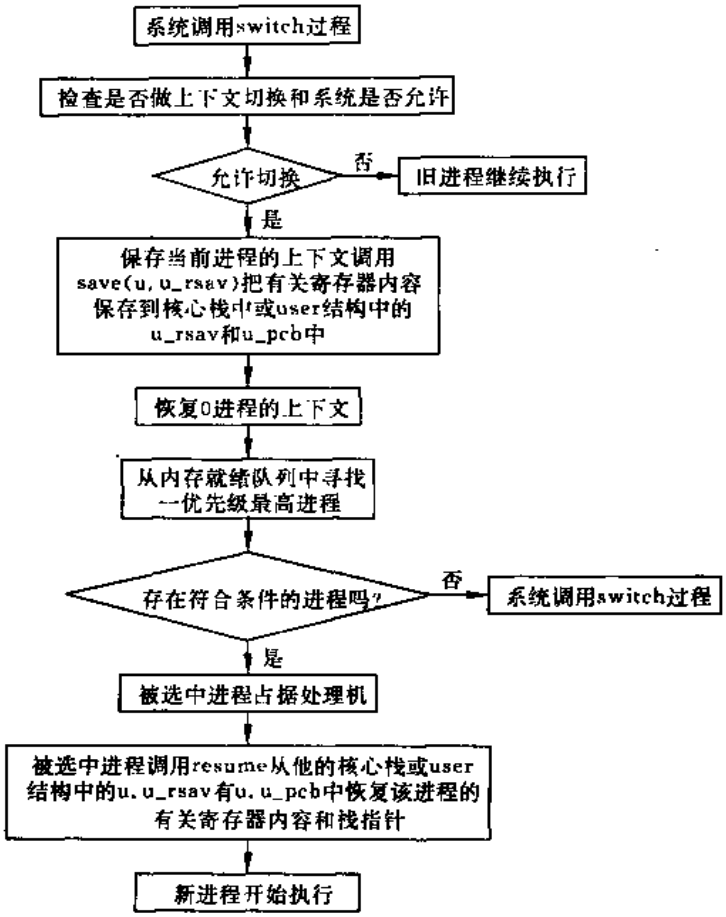


图 E1.4

(2) 为了与其他并发进程保持同步，调用了 wait 或 stop 过程等，从而主动放弃了处理机而进入睡眠状态。

(3) 当系统在从核心态转入用户态时，发现 runrun 调度标志被设置(即系统中某进程的优先级已高于当前执行进程的优先级)，系统调用优先级高的进程执行。

(4) 时间片用完，且当前进程的优先级低于其他就绪进程。

(5) 当前进程调用 exit，自我终止时。

7. 在系统调用 sleep 和 wakeup 时，要提高处理机的执行级(相当于优先级)来防止中断，为什么要这样做(提示：系统经常要从中断处理程序中唤醒睡眠进程)。

答： 系统调用 sleep 和 wakeup 时经常要做进程上下文切换，而系统又经常要从中断处理程序中唤醒睡眠进程，为了保证进程上下文切换的正确进行，防止在切换过程中响应中断，需要提高处理机的执行级。

8. 编写一个程序，利用 fork 调用创建一个子进程，并让该子进程执行一个可执行文件。

答：

```
#include <stdio.h>
main( ){
char *command;
char *prompt="$";

        while(printf("%s",prompt),gets(command)!=NULL){
                if(fork( )==0)
                    execlp(command,command,(char *)0);
                else
                    wait(0);
        }
}
```

9. 什么是软中断？

答：软中断是对硬中断的一种模拟，发送软中断就是向接收进程的 proc 结构中的相应项发送一个特定意义的信号。软中断必须等到接收进程执行时才能生效。

10. 进程在什么时候处理它接收到的软中断信号？进程接收到软中断信号后放在什么地方？

答：进程在再次被调度执行时先检查是否收到软中断，若进程接收到了软中断信号则优先处理软中断。进程把接收到软中断信号存放在 proc 结构的相应项中。

11. Shell 符号“》”将所执行命令的结果输出追加到一个指定的文件中。如果指定文件不存在，则该命令创建一个新文件并将输出写入其中。否则，它打开该文件并在该文件的数据尾部接着写入。编写实现“》”的 C 语言代码。

答：

```
#include<stdio.h>
main( ){
FILE *fp;
char  *filename;
char  *string;

if(fp=fopen(filename,"a")==0) fp=fopen(filename,"w");
        fputs(string,fp);
        fclose(fp);
}
```

12. 编写一程序，比较使用共享存储区和消息机制进行数据传输的速度。

答：略(参见实验 2)。

13. 描述 UNIX System V 中消息机制的通信原理。

答：UNIX System V 中消息机制使用消息队列和消息头两种基本数据结构，消息队列中每一表项由关键字、访问控制结构及操作状态信息组成，消息头描述消息的有关特征。在消息机制中，消息被格式化为类型与数据对，且允许不同的进程根据不同的消息类型接收。

消息机制提供 4 个系统调用 msgget，msgctl，msgsnd 和 msgrcv。系统调用 msgget 返回一个消息描述字 msgqid。msgqid 指定一个消息队列供其它三个系统调用使用。系统调用 msgctl 用来设置和返回与 msgqid 相关联的参数选项，以及用来删除消息描述符的相关选择项。系统调用 msgsnd 和 msgrcv 分别表示发送和接收一消息。

使用消息机制的通信双方先要建立相同的消息队列。通信时先得到自身的 msgqid，若要发送消息则把待发消息写入消息正文部分，并指定消息类型。最后调用 msgsnd 把消息发送到消息队列上。若要接收消息则调用 msgrcv 从消息队列中取出消息。

形式描述略。

14. *比较 UNIX System V 存储管理中的过程 malloc 和 memall 的功能与区别，并画出各自的程序流程图。*

答：在 UNIX System V 存储管理中使用过程 malloc 为进程分配页表区，而 memall 是为进程分配页面。Malloc 过程使用 mp 为链首的自由链式分区可用表 map 作为数据结构。

Memall 使用位示图和 mem 结构作为数据结构。图 E1.5 和 E1.6 分别为 malloc 和 memall 的流程图。

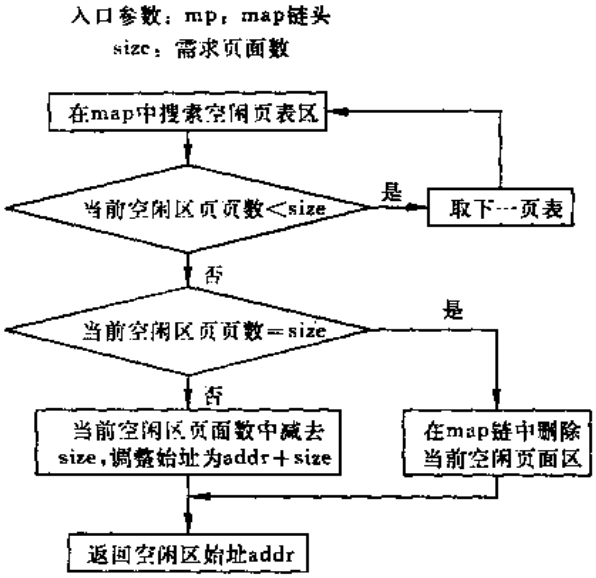


图 E1.5　malloc 流程图

15. *UNIX System V 为什么要采用交换和请求调页两种内存管理策略？交换和请求调页方式有何区别？*

答：交换是 UNIX System 早期版本引入的扩充内存技术，但交换不能实现虚拟存储，故 UNIX System V 引入请求调页内存管理策略以实现虚拟存储且保留了交换技术。

交换技术在交换时换入换出的是整个进程(除 Proc 外)，即使实现部分交换也不是按进

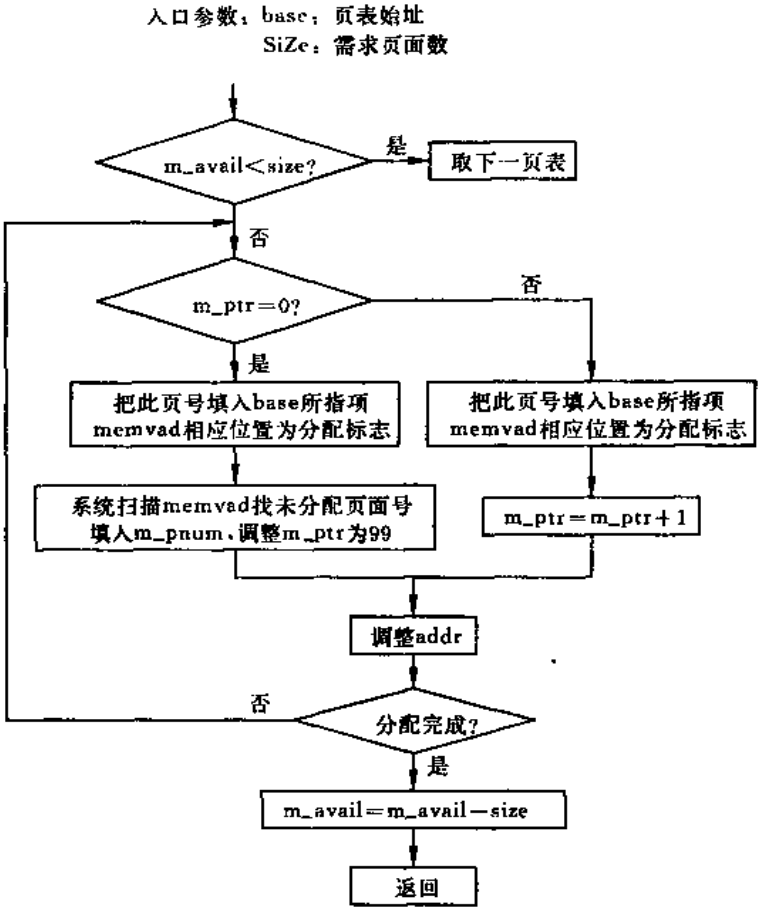


图 E1.6　memall 流程图

程执行的需要进行交换。交换根据程序和数据在内存或外存中驻留时间长短进行换入换出。交换不能实现虚拟存储。

请求调页只是把经常执行或正在执行的页调入内存，当执行到当前页不在内存时，系统调入此页。请求调页内存管理技术可以实现虚拟存储。

16. *在图 5.31 所示请求调页的调入处理过程中，有可能出现空闲页面链表中页面内容不同于外存设备上页面内容的情况，此时应取哪一个页面调入内存，为什么？*

答：在请求调页的调入处理过程中，若空闲页面链表中页面内容不同于外存设备上的页面内容，则应调入空闲页面链表中页面的内容。因为在系统运行过程中有些页被淘汰到空闲页面链表中，且其中的内容已被修改，但还没有被写入外存，当此空闲页面再分配给其他进程前要把它写入外存。若原进程再次要求调入此页就从空闲页面链表取出。这样可以提高系统的效率。

17. *简述核心页表和进程页表之间的关系。为什么各进程页表的基址寄存器中放入的是虚地址？*

答：核心页表和各进程页表在物理上是一块连续区。各进程页表都在核心虚存空间分

得一虚存区，也即核心页表指向的存储区存放进程页表。

用户进程无法直接访问存在于核心区的虚地址，所以各进程页表的基址寄存器中存放的是进程页表在核心页表中的偏移地址。

18. VAX-11 机为页表提供几个基址寄存器？为什么？

答：VAX-11 机为页表提供三个基址寄存器：SBR，P0BR，P1BR。因为 VAX-11 的虚存空间划分为进程空间和系统空间，进程空间又分 P0(程序区)和 P1(控制区)。

19. 图 5.32 中所述最不经常使用算法每次只淘汰一个页面。如果将出现缺页(V=0)的前后页面也预先换入的话，将改进算法的效率。试改进图 5.32 算法，使其可预先换入页面并形式化地描述之。

答：(图 E1.7)

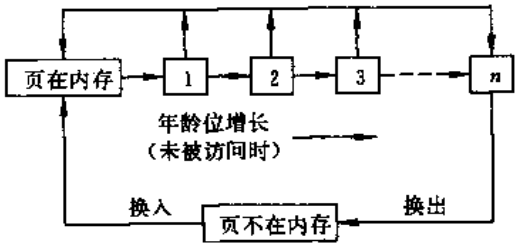


图 E1.7

形式化描述略。

20. 形式化地描述 memfree 和 mfree 过程。

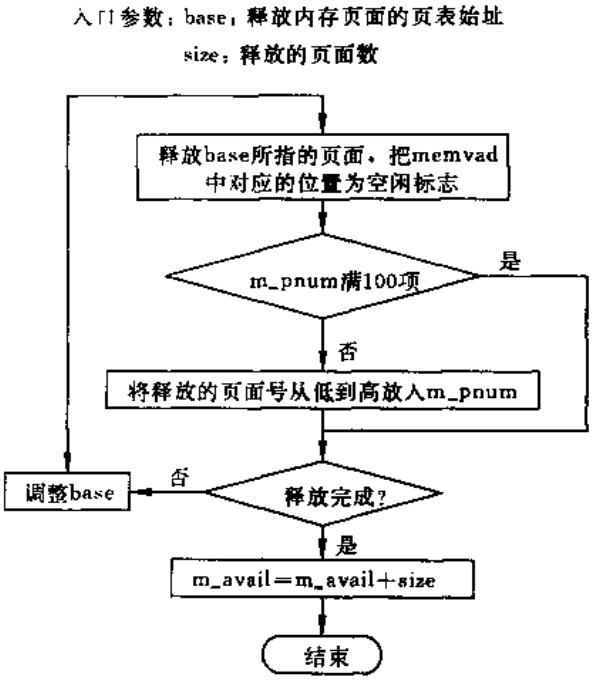


图 E1.8 memfree 过程

图 E1.8 为 memfree 的流程图，图 E1.9 为 mfree 的流程图。读者可参考 memfree 和 mfree 的流程图写出相应的形式化描述。

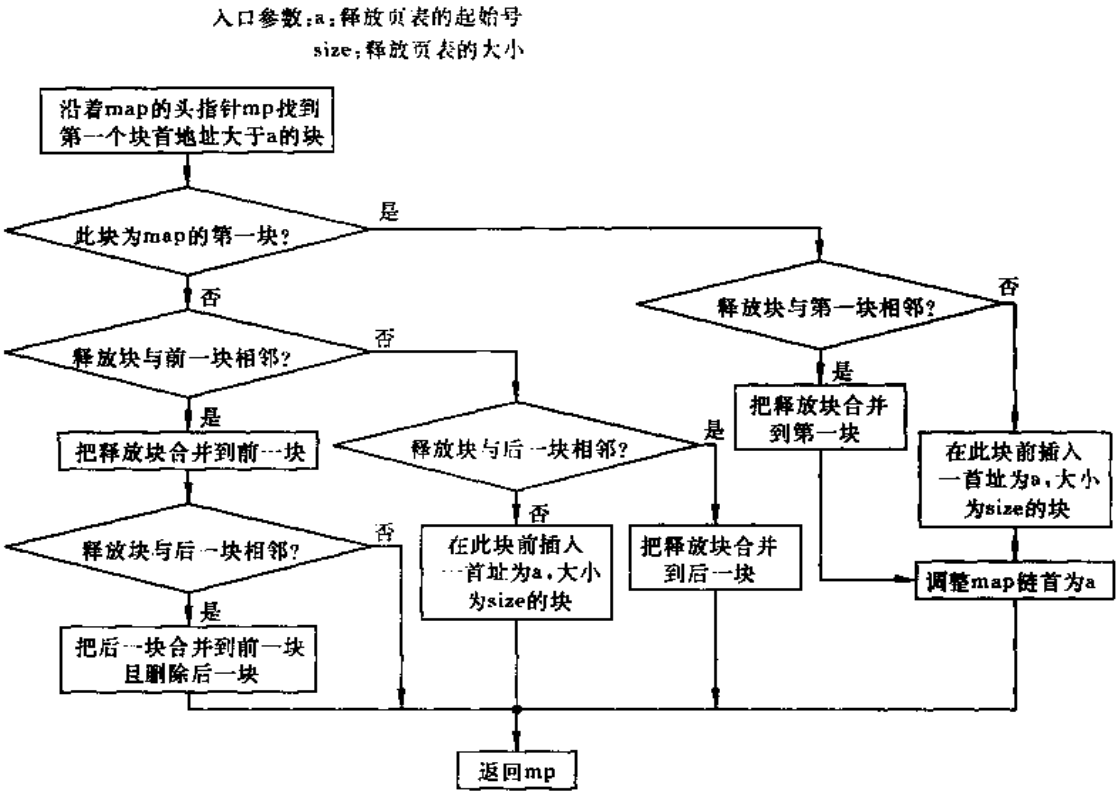


图 E1.9　mfree 过程

21. 小结 UNINX System V 中进程管理和存储管理部分的联系。

答： 进程管理包括进程创建、进程调度、进程执行和进程撤消。进程创建、调度和执行需要存储管理部分为进程分配或释放内存空间，进程的撤消需要存储管理部分回收分配给撤消进程的内存空间。存储管理系统必须决定哪个进程的哪个部分应该放在内存，并管理那些不在内存又属于同一进程虚空间的部分。

第7章 文件系统

1. 什么是文件、文件系统？文件系统有哪些功能？

答：在计算机系统中，文件被解释为一组赋名的相关字符流的集合，或者是相关记录的集合。

文件系统是操作系统中与管理文件有关的软件和数据。

文件系统的功能是为用户建立文件，撤销、读写修改和复制文件，以及完成对文件的按名存取和进行存取控制。

2. 文件系统一般按什么分类？可以分为哪几类？

答：文件系统一般按性质，用途，组织形式，文件中的信息流向或文件的保护级别等分类。

按文件的性质与用途可以分为系统文件，库文件和用户文件。按文件的组织形式可以分为普通文件，目录文件和特殊文件。按文件中的信息流向可以分为输入文件，输出文件和输入/输出文件。按文件的保护级别可以分为只读文件，读写文件，可执行文件和不保护文件。

3. 什么是文件的逻辑结构？什么是记录？

答：文件的逻辑结构就是用户可见的结构，可分为字符流式的无结构文件和记录式的有结构文件两大类。

记录是一个具有特定意义的信息单位，它由该记录在文件中的逻辑地址(相对位置)与记录名所对应的一组关键字，属性及其属性值所组成。

4. 设文件的结构为多重结构和转置结构的组合，且定义函数 decode(K,x) 和 retrive(K,x)如下：

函数 decode(K,x)为关键字 K 的搜索函数。其中 K 为待搜索关键字名，x 为顺序指针。当被搜索文件为多重结构时，指向关键字 K 的指针只有一个，即 e(K)，此时 x＝nil，且 decode(K,x)返回指针 e(K)。当被搜索文件为转置结构时，一般指向关键字 K 的指针有 n 个，即 e1(K)，…e_n(K)，n 为包含关键字 K 的记录个数。此时，若 x＝nil，decode(K,x)返回 e_1(K)；若 x＝e_n(K)，则 decode(K,x)返回 nil；否则，若 x＝e_i(K)，(1 <＝ i ＜ n)；则 decode(K,x) 返回 e_{i+1}(K)。

函数 retrive(K,x) 是记录搜索函数。其中 K 为指向关键字的指针，x 为记录顺序指针。如果 x＝nil 则 retrive(K,x)返回被搜索记录队列的第一个记录的逻辑地址；若 x 等于该记录队列的最后一个记录的话，则 retrive(K,x) 返回 nil；否则，retrive(K,x)返回 x 的下一个记录的逻辑地址。试问：

(1) 如果 decode(K,x)采用线性搜索法，怎样描述 decode(K,x)？

(2) 如果 retrive(K,x) 采用二分搜索法，应怎样排列记录队列，和怎样描述函数 retrive

(K,x)?

(3) 设函数 search(k)给出含有关键字 k 的所有记录的逻辑地址，试描述 search(k)。

答：

(1) decode(K,x)

```
{
  int i
  if (文件是多重结构) . return e(K)
  if (x==nil) return e1(K)
  if (x==e_n(K)) return nil
  for (i=1;i<n;i++)
    if (x==e_i(K)) return e_{i+1}(K)
}
```

(2) 应该对记录进行排序。

```
retrive(K,x)
{
  if (x==nil) return 第一个记录的逻辑地址
  if (x==最后一个记录) return nil
  if (x==中间记录) return 中间记录后的那个记录
  if (x<中间记录) 对第一个记录和中间记录之间的记录递归调用 retrive(K,x)
  对中间记录和最后一个记录之间的记录递归调用 retrive(K,x)
}
```

(3) search(k)

```
{
   调用 decode(k,x)获得指向关键字 k 的指针 K
   调用 retrive(K,x)返回所有记录
}
```

5. 设散列函数 $h(n)=(676\times l_1+26\times l_2+l_3)(\text{mod } t)$，$t=11$，$l_i$为关键字 n 的第 i 个英文字母序号。关键字表长为 11，关键字名为英文字母。先按线性散列法，再按平方散列法计算将任意 7 个关键字放入链表中所用的计算次数。这里令 a=2,c=1。

答： 设 7 个关键字为 zhl,ouy,lwj,yks,lxz,suy,hls. $h(n)=(676\times l_1+26\times l_2+l_3) \text{ mod } 11$.

(1) 线性散列 $h_i(k)=(h_1(k)+2i) \text{ mod } 11$

$h(zhl)=(676\times 19+26\times 8+12) \text{ mod } 11=9$

$h(ouy)=(676\times 15+26\times 21+25) \text{ mod } 11=8$

$h(lwj)=(676\times 12+26\times 23+10) \text{ mod } 11=8$

$h_2=(8+2\times 2) \text{ mod } 11=1$

$h(yks)=(676\times25+26\times11+19) \bmod 11=1$

$h_2=(1+2\times2) \bmod 11=5$

$h(lxz)=(676\times12+26\times24+26) \bmod 11=6$

$h(suy)=(676\times19+26\times21+25) \bmod 11=6$

$h_2=(6+2\times2) \bmod 11=10$

$h(hls)=(676\times8+26\times12+19) \bmod 11=8$

$h_2=(8+2\times2) \bmod 11=1$

$h_3=(8+2\times3) \bmod 11=3$

计算了 12 次。

(2) 平方散列

$h_i(k)=(h_1(k)+i^2) \bmod 11$

$h(zhl)=9$

$h(ouy)=8$

$h(lwj)=8$

$h_2=(8+2^2) \bmod 11=1$

$h(yks)=1$

$h_2=(1+2^2) \bmod 11=5$

$h(lxz)=6$

$h(suy)=6$

$h_2=(6+2^2) \bmod 11=10$

$h(hls)=8$

$h_2=(8+2^2) \bmod 11=1$

$h_3=(8+3^2) \bmod 11=6$

$h_4=(8+4^2) \bmod 11=2$

计算了 13 次。

6. 设关键字表按英文字母顺序排列，且两关键字之间的距离为 d，写出 $n=3$ 的三分搜索算法。

答： 设第一个关键字为 K_0，最后一个关键字为 K_n，要搜索的关键字为 K，则描述算法如下：

```
key findkey(K0,Kn,K)
{
  if (K==K0) return K0;
  if (K==Kn) return Kn;
  K1=K0+(Kn-K0)/3;
  K2= K0+2*(Kn-Kn)/3;
  if (K==K1) return K1;
  if (K==K2) return K2;
  if(K0<K<K1) return findkey(K0,K1,K);
```

```
    if(K1<K<K2) return findkey(K1,K2,K);
    return findkey(K2,Kn,K);
}
```

7. 文件的物理结构有哪几种？为什么说串联文件结构不适于随机存取？

答：文件的物理结构是指文件在存储设备上的存放方法。常用的文件物理结构有连续文件，串联文件和索引文件 3 种。

串联文件结构用非连续的物理块来存放文件信息。这些非连续的物理块之间没有顺序关系，链接成一个串联队列。搜索时只能按队列中的串联指针顺序搜索，存取方法应该是顺序存取的。否则，为了读取某个信息块而造成的磁头大幅度移动将花去较多的时间。因此，串联文件结构不适于随机存取。

8. 设索引表长度为 13，其中 0～9 项为之间寻址方式，后 3 项为间接寻址方式，试描述出给定文件长度 n(块数)后的索引方式寻址算法。

答：设每个物理块的大小为 k 字节，每个物理块可以放 m 个物理块号。输入为文件内的偏移地址 o_addr，输出为物理地址 p_addr，－1 表示寻址失败。描述算法如下：

```
addr find(o_addr)
{
  if((o_addr<0)||(o_addr>n*K-1))  return(-1)
  else if(o_addr<10*K) 直接寻址，获得 p_addr
  else 间接寻址，获得 p_addr
  return(p_addr)
}
```

9. 常用的文件存储设备的管理方法有哪些？试述主要优缺点。

答：文件存储设备的管理实质上是一个空闲块的组织和管理问题。有 3 种不同的空闲块管理方法。即空闲文件目录，空闲块链和位示图。

空闲文件目录管理方法就是把文件存储设备中的空闲块的块号统一放在一个称为空闲文件目录的物理块中，其中空闲文件目录的每个表项对应一个由多个空闲块构成的空闲区。该方法实现简单，适于连续文件结构的文件存储区的分配与回收。但是由于回收时不进行合并，所以使用该方法容易产生大量的小块空闲区。

空闲块链法把文件存储设备上的所有空闲块链接在一起，从链头分配空闲块，把回收的空闲块插入到链尾。该方法不占用额外的空间，但实现复杂。

位示图法是从内存中划出若干字节，每个比特位对应一个物理块的使用情况。如果该位为 0 则表示对应的是空闲块，为 1 则表示对应的物理块已分配出去。位示图法在查找空闲块时无需启动外设，但要占用内存空间。

10. 试述成组链法的基本原理，并描述成组链法的分配与释放过程。

答：成组链法首先把文件存储设备中的所有空闲块按 50 块一组分组。组的划分是从后

往前进行的。其中，每组的第一块用来存放前一组中各块的块号和总块数。第一组为 49 块。最后一组的物理块号与总块数只能放在管理文件存储设备用的文件资源表中。

分配和释放过程：

首先，系统在初始化时把文件资源表复制到内存，从而把文件资源表中放有最后一组空闲块块号与总块数的堆栈载入内存，并使得空闲块的分配与释放可以在内存中进行。用于空闲块分配与回收的堆栈有栈指针 Ptr，且 Ptr 的初值等于该组空闲块的总块数。当申请者申请 n 块空闲块时，按照后进先出的原则，分配程序在取走 Ptr 所指的块号之后，再做 Ptr＝Ptr－1 的操作。这个过程一直持续到所要求的 n 块空间都分配完毕或堆栈中只剩下最后一个空闲块的块号时。当堆栈中只剩下最后一个空闲块号时，系统启动设备管理程序，将该块中存放的下一组的块号与总块数读入内存之后再把该块分配给申请者。然后，系统重新设置 Ptr 指针，并继续为申请者分配空间。

文件存储设备的最后一个空闲块中设置有尾标识，以指示空闲块分配完毕。

如果用户进程不再使用有关文件并删除这些文件时，回收程序回收这些文件占用的物理块。成组链法的回收过程仍利用文件管理堆栈进行。在回收时，回收程序先做 Ptr＝Ptr＋1 操作，然后把回收的物理块号放入当前 Ptr 指针所指的位置。如果 Ptr 等于 50，则表示该组已经全部回收。此时，如果还有物理块需要回收的话，那么回收该块并启动 I/O 设备管理程序，把回收的 50 个块号与块数写入新回收的块中。然后将 Ptr 置 1 另起一个新组。

对空闲块的分配和释放必须互斥进行。

11. 什么是文件目录？文件目录中包含哪些信息？

答：一个文件的文件名和对该文件实施控制管理的说明信息称为该文件的说明信息，又称为该文件的目录。

文件目录中包含文件名、与文件名相对应的文件内部标识以及文件信息在文件存储设备上第一个物理块的地址等信息。另外还可能包含关于文件逻辑结构、物理结构、存取控制和管理等信息。

12. 二级目录和多级目录的好处是什么？符号文件目录表和基本文件目录表是二级目录吗？

答：二级目录和多级目录的好处是可以减少文件命名冲突和提高对目录表的搜索速度。

符号文件目录表和基本文件目录表是实现文件共享的一种方法，并不是二级目录。

13. 文件存取控制方式有哪几种？试比较它们各自的优缺点。

答：文件存取控制方式一般有存取控制矩阵、存取控制表、口令和密码术 4 种方式。

存取控制矩阵方式以一个二维矩阵来进行存取控制。而且矩阵的一维是所有的用户，另一维是所有的文件。对应的矩阵元素则是用户对文件存取控制权。存取控制矩阵的方法在概念上比较简单，但是当用户和文件较多时，存取控制矩阵将变得非常庞大，从而时间和空间的开销都很大。

存取控制表以文件为单位，把用户按某种关系划分为若干组，同时规定每组的存取限

制。这样，所有用户组对文件权限的集合就形成了该文件的存取控制表。存取控制表方式占用空间较小，搜索效率也较高，但要对用户分组，引入了额外的开销。

口令方式有两种。一种是当用户进入系统，为建立终端进程时获得系统使用权的口令。另一种口令方式是，每个用户在创建文件时，为每一个创建的文件设置一个口令，且将其置于文件说明中。当任一用户想使用该文件时，都必须首先提供口令。口令方式比较简单，占用的内存单元以及验证口令所费时间都非常少。不过，相对来说，口令方式保密性能较差。

密码方式在用户创建源文件并写入存储设备时对文件进行编码加密，在读出文件时对其进行译码解密。加密方式具有保密性强的优点。但是，由于加密解密工作要耗费大量的处理时间，因此，加密技术是以牺牲系统开销为代价的。

14. 设文件 SQRT 由连续结构的定长记录组成，每个记录的长度为 500 字节，每个物理块长 1000 字节，且物理结构是连续结构并采用直接存取方式；试按照图 7.23 所示文件系统模型，写出系统调用 Read (SQRT,5,15000)的各层执行结果。其中，SQRT 为文件名，5 为记录号，15000 为内存地址。

答：(简述)

第 1 层用户接口层，把系统调用转化成内部调用格式。

第 2 层符号文件系统层，把第 1 层提供的用户文件名转化成系统内部的唯一标识符 fd。

第 3 层基本文件系统层根据参数 fd 找到文件的说明信息。

第 4 层存取控制验证层根据存取控制信息和用户访问要求，检验文件访问的合法性。

第 5 层逻辑文件系统层根据文件逻辑结构找到第 5 个记录对应的逻辑地址 2000，并将其转换为相对块号 2。

第 6 层物理文件系统层根据文件的物理结构把相对块号 2 转换成物理地址如 1000000。

第 7 层文件存储设备分配模块和设备策略模块把物理块号转换成具体磁盘的柱面号、磁道号和扇区号，然后准备启动输入设备命令。

第 8 层启动输入输出层由设备处理程序执行读操作，把第 5 个记录读到内存地址 15000 处。

第8章 设备管理

1. 设备管理的目标和功能是什么?

答：设备管理的目标是：选择和分配输入/输出设备以便进行数据传输操作；控制输入/输出设备和CPU(或内存)之间交换数据；为用户提供一个友好的透明接口；提高设备和设备之间、CPU和设备之间，以及进程和进程之间的并行操作，以使操作系统获得最佳效率。

设备管理的功能是：提供和进程管理系统的接口；进行设备分配；实现设备和设备、设备和CPU等之间的并行操作；进行缓冲区管理。

2. 数据传送控制方式有哪几种？试比较它们各自的优缺点。

答：数据传送控制方式有程序直接控制方式、中断控制方式、DMA方式和通道方式4种。

程序直接控制方式就是由用户进程来直接控制内存或CPU和外围设备之间的数据传送。它的优点是控制简单，也不需要多少硬件支持。它的缺点是：CPU和外围设备只能串行工作；设备之间只能串行工作；无法发现和处理由于设备或其他硬件所产生的错误。

中断控制方式是利用向CPU发送中断的方式控制外围设备和CPU之间的数据传送。它的优点是大大提高了CPU的利用率且能支持多道程序和设备的并行操作。它的缺点是：由于数据缓冲寄存器比较小，如果中断次数较多，仍然占用了大量CPU时间；在外围设备较多时，由于中断次数的急剧增加，可能造成CPU无法响应中断而出现中断丢失的现象；如果外围设备速度比较快，可能会出现CPU来不及从数据缓冲寄存器中取走数据而丢失数据的情况。

DMA方式是在外围设备和内存之间开辟直接的数据交换通路进行数据传送。它的优点是除了在数据块传送开始时需要CPU的启动指令，在整个数据块传送结束时需要发中断通知CPU进行中断处理之外，不需要CPU的频繁干涉。它的缺点是在外围设备越来越多的情况下，多个DMA控制器的同时使用，会引起内存地址的冲突并使得控制过程进一步复杂化。

通道方式是使用通道来控制内存或CPU和外围设备之间的数据传送。通道是一个独立与CPU的专管输入/输出控制的机构，它控制设备与内存直接进行数据交换。它有自己的通道指令，这些指令受CPU启动，并在操作结束时向CPU发中断信号。该方式的优点是进一步减轻了CPU的工作负担，增加了计算机系统的并行工作程度。缺点是增加了额外的硬件，造价昂贵。

3. 什么是通道？试画出通道控制方式时的CPU、通道和设备的工作流程图。

答：通道是一个独立与CPU的专管输入/输出控制的机构，它控制设备与内存直接进行数据交换。它有自己的通道指令，这些指令受CPU启动，并在操作结束时向CPU发中断信号。

设备和通道的工作流程图如图 E1.10 所示。

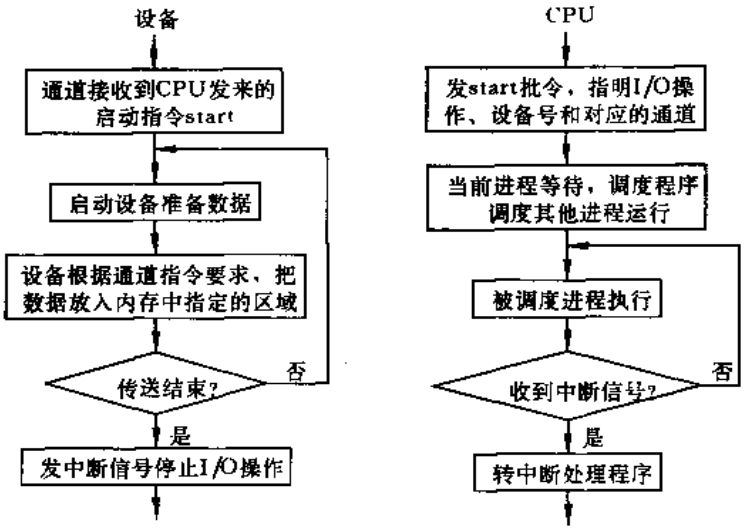


图 E1.10

4. 什么是中断？什么叫中断处理？什么叫中断响应？

答：中断是指计算机在执行期间，系统内发生任何非寻常的或非预期的急需处理事件，使得 CPU 暂时中断当前正在执行的程序而转去执行相应的事件处理程序，待处理完毕后又返回原来被中断处继续执行的过程。

CPU 转去执行相应的事件处理程序的过程称为中断处理。

CPU 收到中断请求后转到相应的事件处理程序称为中断响应。

5. 什么叫关中断？什么叫开中断？什么叫中断屏蔽？

答：把 CPU 内部的处理机状态字 PSW 的中断允许位清除从而不允许 CPU 响应中断叫做关中断。

设置 CPU 内部的处理机状态字 PSW 的中断允许位从而允许 CPU 响应中断叫做开中断。

中断屏蔽是指在中断请求产生之后，系统用软件方式有选择地封锁部分中断而允许其余部分的中断仍能得到响应。

6. 什么是陷阱？什么是软中断？试述中断、陷阱和软中断之间异同。

答：陷阱指处理机和内存内部产生的中断，它包括程序运算引起的各种错误，如地址非法、校验错、页面失效。存取访问控制错、从用户态到核心态的切换等都是陷阱的例子。

软中断是通信进程之间用来模拟硬中断的一种信号通信方式。

7. 描述中断控制方式时的 CPU 动作过程。

答：(1) 首先，CPU 检查响应中断的条件是否满足。如果中断响应条件不满足，则中断处理无法进行。

(2) 如果 CPU 响应中断，则 CPU 关中断。

(3) 保存被中断进程现场。

(4) 分析中断原因，调用中断处理子程序。

(5) 执行中断处理子程序。

(6) 退出中断，恢复被中断进程的现场或调度新进程占据处理机。

(7) 开中断，CPU 继续执行。

8. 什么是缓冲？为什么要引入缓冲？

答： 缓冲即是使用专用硬件缓冲器或在内存中划出一个区域用来暂时存放输入输出数据的器件。

引入缓冲是为了匹配外设和 CPU 之间的处理速度，减少中断次数和 CPU 的中断处理时间，同时解决 DMA 或通道方式时的数据传输瓶颈问题。

9. 设在对缓冲队列 em，in 和 out 进行管理时，采用最近最少使用算法存取缓冲区，即在把一个缓冲区分配给进程之后，只要不是所有其他的缓冲区都在更近的时间内被使用过，则该缓冲区不再分配出去。试描述过程 take_buf(type,number)和 add_buf(type,number)。

答：

对每个缓冲区设置一个时间标志位，其取值为该缓冲区上次放入队列时的系统时间。

```
take_buf(type,number)
{
  取出时间标志位最小的缓冲区
}

add_buf(type,number)
{
  把缓冲区放入队列，并获取当前系统时间赋给其时间标志位
}
```

10. 试述对缓冲队列 em，in 和 out 采用最近最少使用算法对改善 I/O 操作性能有什么好处？

答： 采用最近最少使用算法可以保留那些在最近一段时间内使用次数较多的缓冲区，而这些缓冲区继续被使用的可能性比较大，从而可以减少缓冲区分配和回收的次数，避免了频繁的分配、回收操作，所以可以改善 I/O 操作性能。

11. 用于设备分配的数据结构有哪些？它们之间的关系是什么？

答： 用于设备分配的数据结构有：设备控制表 DCT、系统设备表 SDT、控制器表 COCT 和通道控制表 CHCT。

SDF 整个系统一张，每个设备有一张 DCT，每个控制器有一张 COCT，每个通道有一张 CHCT。SDF 中有一个 DCT 指针，DCT 中有一个 COCT 指针，COCT 中有一个 CHCT

指针。

12. 设计一个设备分配的安全检查程序，以保证把某台设备分配给某进程时不会出现死锁。

答：参见教材 72 页避免死锁章节。

13. 什么是 I/O 控制？它的主要任务是什么？

答：I/O 控制是指从用户进程的输入/输出请求开始，给用户进程分配设备和启动有关设备进行 I/O 操作，并在 I/O 操作完成之后响应中断，直至善后处理为止的整个系统控制过程。

14. I/O 控制可用哪几种方式实现？各有什么优缺点？

答：I/O 控制过程可用三种方式实现：作为请求 I/O 操作的进程实现；作为当前进程的一部分实现；由专门的系统进程——I/O 进程完成。

第一种方式请求对应 I/O 操作的进程能很快占据处理机，但要求系统和 I/O 操作的进程应具有良好的实时性。第二种方式不要求系统具有高的实时性，但 I/O 控制过程要由当前进程负责。第三种方式增加了一个额外的进程开销，但用户不用关心 I/O 控制过程。

15. 设备驱动程序是什么？为什么要有设备驱动程序？用户进程怎样使用驱动程序？

答：设备驱动程序是驱动外部物理设备和相应 DMA 控制器或 I/O 控制器等器件，使之可以直接和内存进行 I/O 操作的子程序的集合。它们负责设置相应设备有关寄存器的值，启动设备进行 I/O 操作，指定操作的类型和数据流向等。

设备驱动程序屏蔽了直接对硬件操作的细节，为编程者提供操纵设备的友好接口。

用户进程通过调用设备驱动程序提供的接口来使用设备驱动程序。

第9章　文件系统和设备管理示例

1. 描述 UNIX 文件系统的空闲磁盘块分配算法 alloc 和释放算法 free.

答：

(1) alloc

```
    输入:欲分配的块数
    输出:分配成功的块数
Begin
    分配程序取走 Ptr 所指的块号                    /* Ptr 为块号指针 */
    分配成功的块数 b=0
    While(b < n )                                  /* n 为申请块号 */
        If    文件设备中没有空闲块
        Then  return(0)
        fi
        If    堆栈只剩最后一个空闲块
        Then  通过设备管理程序,将该块中存放的块号
              与总块数读入内存后将该块分配给申请者
              重置 Ptr
        fi
        作 Ptr=Ptr - 1
        b = b+1
    end
End
```

(2) free

```
    输入:文件首物理块号
    输出:是否成功
     begin
     指针 blk = 文件首物理块号
     ptr = 1
     while (blk<>nill)
          if  Ptr > 50
          then  启动 I/O 设备程序,把回收的 50 个块号和块
          数写入回收块中
          Ptr=1
          fi
          blk 所指的首物理块号放入当前指针 Ptr 所指的位置
          Ptr = Ptr +1
          blk 指向下一块物理块号
```

end
end

2. UNIX 文件系统为什么有磁盘 i 节点和内存 i 节点？为什么内存 i 节点的内容和磁盘 i 节点的内容不一样？

答：UNIX 系统中，磁盘 i 节点以静态形式存放文件说明信息。引入内存 i 节点是为了减少设备的启动次数以及提高操作速度，把磁盘 i 节点复制到内存特定区域。由于进程需用 i 节点中的逻辑结构和物理结构信息完成对文件信息的保护和共享，故 i 节点中多了当前文件状态信息。

3. 一个进程在发现所要搜索的 i 节点位于缓冲区中，且该 i 节点为上锁状态时，将在 iget 中睡眠。在该 i 节点变为开锁状态并唤醒该睡眠进程之后，该进程将重新开始搜索 i 节点。为什么？

答：该内存 i 节点可能被释放。

4. 描述一个算法，该算法把内存 i 节点的内容写回磁盘 i 节点中。

答：算法描述略

5. UNIX System V 允许一个路径名分量最长达 14 个字符。namei 把 1 个分量中多余的字节截掉。重新设计文件系统和重写 namei 以允许任意长度的分量名。

答：更改文件系统中 SFD 中文件名的长度以及加大 namei 中目录变量的字节即可。具体描述可参照教材中关于 namei 的描述。

6. UNIX 文件系统的目录结构是什么？

答：UNIX 文件系统的目录结构是树型层次结构。

7. 试述你自己用过的 UNIX 文件系统的系统调用。

答：如对文件操作，可用 open，read，write 等。

8. 为什么要有系统打开文件表？用户进程是怎样与文件系统联系的？创建一个文件时创建系统打开文件表吗？

答：用户打开表记录一个进程可以同时打开的文件数，UNIX System V 最多可达到 20。用户打开表的描述符返回给用户进程后成为文件描述符。与此相对应，用户对文件进行操作时，在系统内部需有相应数据结构来记录和控制打开文件的用户进程，以及记录和控制那些共享同一文件的用户进程。这个数据结构就是系统打开表。用户进程通过系统调用来完成与文件系统联系。创建文件时，需要在系统打开文件表的相应表项中生成相应数据，但不需要创建系统打开表。

9. 描述系统调用 write 的实行过程，并画出控制流向图。

答：

Write

输入：文件描述 fd

用户内存地址 buffer

欲写字节数 count

输出：写到文件中的字节数

Begin

在 User 结构中设置用户地址，字节偏移与字节计数等

由文件描述符 fd 得到系统打开文件表指针

检查文件的可存取性

由系统打开文件表得到内存 *i* 节点指针

锁定内存 *i* 节点

Repeat

将内存 buffer 中数据写入磁盘中

Until

写完 count 字节或文件中不再存在字符

将内存 *i* 节点解锁

返回写入字节总数

End

控制流向图(图 E1.11)

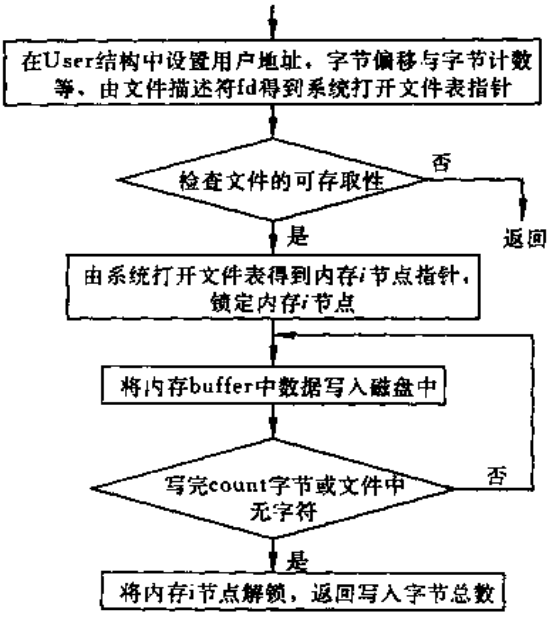


图 E1.11

10. UNIX 中的中断与陷阱的异同有哪些？处理方法上有哪些区别？为什么？

答：共同点：(1) 都是在系统执行期间发生的突发事件，要求 CPU 改变当前流程去处理突发事件。(2) 它们都是通过 SCB 找到相应的处理程序的入口地址，而后执行之。

不同点：(1) 中断是与整个系统或非当前执行进程有关的，其处理程序只能在系统的中

断栈上执行。而陷阱是与当前执行进程有关的事件，可以在当前执行进程的核心栈上执行处理。(2) 响应时，陷阱处理不做进程切换，其处理机状态字中优先级一般不变。而响应中断时，系统只响应那些优先级高于状态字中的中断，低级中断则被屏蔽。

中断和陷阱处理程序的进入与退出方式上，二者处理不相同。这是因为陷阱处理要求陷阱指令与处理子程序之间传递较多参数，另外中断处理子程序和陷阱处理子程序在不同的堆栈上执行。

11. 为什么陷阱处理时，处理机不用降低状态优先级，而中断处理时，需等到处理机状态优先级降低到低于中断优先级才开始？

答：陷阱处理时，不切换进程，处理机状态字中优先级没变，故与处理机状态优先级无关。而中断处理时，只有高于处理机状态优先级中断才能执行，优先级较低中断，必须等处理机状态优先级降低才能执行。

12. 什么是中断优先级？什么是处理机优先级？

答：为了按中断的轻重缓急处理响应中断，操作系统对不同的中断赋予不同的级别。这些级别被称为中断优先级。为了禁止中断或屏蔽中断，CPU 的处理状态字 PSW 中也设置有相应的级别。这些级别被称为处理机优先级。

13. UNIX System V 可处理哪几种陷阱？怎样传递参数？

答：UNIX System V 可处理 12 种不同的陷阱：

SYSCALL	更换到核心态方式
SEGFLT	转换无效
PROTFLT	访问违章
PRIVFLT	非法指令
RSADFLT	保留寻址方式
RSOPFLT	保留操作数
ARTHRP	算术自陷
TRCTRAP	跟踪自陷
BPTFLT	断点故障指令
XFCFLT	扩展功能调用指令
CMPTFLT	兼容方式
RESCHED	进程调度中断

参数传递：

核心态下的陷阱处理不需传递参数。用户态下的陷阱处理需要调用 trap 公共过程来传递参数，形如：trap(sp,type,code,pc,ps)。

14. 为什么要引进缓冲区的概念？UNIX System V 的缓冲区有哪几种？

答：引进缓冲区目的：为了匹配外设与 CPU 之间的处理速度，减少中断次数和中断处理时间，解决 DMA 或通道方式时数据传输瓶颈。

缓冲区分为：空闲缓冲区队列，设备缓冲区队列，设备 I/O 请求队列。

15. 试述空闲缓冲区队列，设备 b 链队列和设备 I/O 请求队列的关系。

答： 空闲缓冲区队列是系统所拥有的所有空闲缓冲区资源。一个空闲缓冲区同时挂在空闲 av 队列和设备 b 队列上。这样读数据时，首先在设备 b 队列上寻找相应的数据块，若此块不在 b 链，则从空闲 av 链中按最近最少使用算法取一空闲缓冲区，改写缓冲控制块的块号加入对应散列。若在，则可不必启动磁盘。系统释放某缓冲区时，把此缓冲区放在 b 链中，缓冲区中的数据只有往此缓冲区写数据时才释放。每个物理块设备都有一个 I/O 请求队列，设备 I/O 请求队列中的缓冲区属于设备 b 链，但不属于空闲 av 队列。设备 I/O 请求队列是由正在请求该块设备进行读写操作的缓冲区所组成的队列。

16. UNIX System V 采用散列法搜索缓冲区队列以提高搜索速度，试计算在公式 I＝(b_dev＋b) mod 64 的条件下，对一个具 200 个缓冲区的缓冲池来说，散列法比线性搜索法优越多少？

答： 假设数据的分布是均匀的。

散列法平均搜索的长度 len ＝(1 * 64＋2 * 64＋3 * 64＋4 * 8)/200＝2.08

线性法平均搜索的长度 len ＝ (n＋1)/2 ＝ 201/2 ＝100.5

当然，散列的分布不可能这么均匀，但是散列法比线性搜索法优越是显而已见的。

17. 描述缓冲区分配算法 geteblk。

答：

```
geteblk
    输出：从空闲 av 队列中所取的缓冲区
    Begin
        If  检查空闲 av 队列为空
            Then 睡眠等待
        fi
        从空闲 av 队列中取一缓冲区 buf
        If  空闲 av 队列中所取的缓冲区为延迟写
            Then 将该缓冲区的数据写回磁盘
        fi
        将空闲 av 队列中所取的缓冲区链入设备 b 链队列
        返回 buf
    end
```

18. UNIX 系统为什么要设置延迟写和异步写？它们各有什么优缺点？

答： 异步写的目的是提高写盘速度，延迟写的目的是为了让数据块在内存中待尽量多的时间，以减少不必要的 I/O 操作。优点：把一个缓冲区的数据往磁盘写时，如同步，进程因为等待写操作完成而进入睡眠，而且要在写操作完成后才释放缓冲区。如延迟写或异步写时，系统启动传输，不等写完成而返回，都加快了写盘速度，但延迟写还减少了不必要的 I/O

操作。缺点：延迟写没有立即把数据写入磁盘，当系统发生瘫痪时将产生磁盘数据错误。异步写是启动传输后，不等传输完成而返回，也可能发生数据错，但可能性小。

19. 什么是块设备驱动程序？

答： 块设备驱动程序是把一个逻辑设备号组成的文件系统地址转换成物理设备上特定的物理块号，并启动物理设备和控制器进行 I/O 传输工作。

20. 怎样在系统中建立设备特殊文件？试在你身边的微机或小型机上建立一个设备特殊文件。

答： 可用 mknod 命令建立一个字符设备文件 ttyb，命令如下：

```
mknod /dev/ttyb C 2 13.
```

21. 设备驱动程序和缓冲管理程序的关系是什么？

答： 缓冲区的设立是为了匹配内存和硬盘的速度，减少磁盘设备启动的次数。操作系统进行读写，首先查找缓冲区是否有所读写的数据；如在，则不必启动设备，否则缓冲管理程序需调用设备驱动程序来完成对缓冲区操作。故缓冲管理程序是设备驱动程序上一级，是调用与被调用关系。

22. 字符设备的缓冲区队列有哪些？对缓冲区队列操作有哪些？

答： 字符设备的缓冲区队列分为：空闲缓冲区队列和 I/O 字符缓冲区队列。操作分为：对空闲缓冲区队列操作和对 I/O 缓冲区队列操作，共 6 种：

(1) 从对空闲缓冲区队列分配一个缓冲区给驱动程序。

(2) 把一个缓冲区释放后放入空闲缓冲区队。

(3) 从 I/O 缓冲区队列中提取一个字符，并调整 I/O 缓冲区队列的字符计数。

(4) 把一个字符放入 I/O 缓冲区队列中的最后一个缓冲区的末尾。

(5) 从 I/O 缓冲区队列中每次移走一个缓冲区中的所有字符或 $n(n>1)$个字符。

(6) 往 I/O 缓冲区队列中每次送一个缓冲区的字符或 $n(n>1)$个字符。

23. 行规则程序功用是什么？

答： 行规则程序功能：

(1) 通过分析将输入字符串变成行。

(2) 处理删除键，以便用户能够从逻辑上部分地删除敲入的字符。

(3) 处理删除行的字符，从而使得在当前行上敲入的所有字符无效。

(4) 处理输出字符格式，例如把表格和空格符号变为空格字符序列。

(5) 把输入的字符回送到显示屏上。

(6) 为终端挂起，断线或响应用户敲入的 delete 键向进程发软中断信号。

(7) 允许不做格式变换的原方式。

24. 试画出在系统调用 write 的执行过程中，UNIX System V 的处理机流程。

答：（图 E1.12）

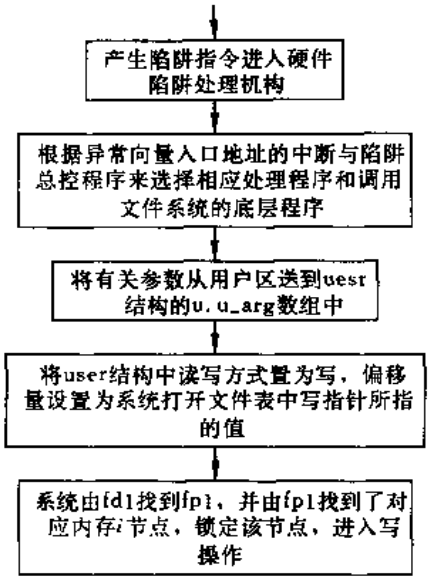


图 E1.12

综 合 试 题

操作系统综合练习试题 1

1.（10 分）试述分区式管理中的最先适应算法(FF)、最佳适用算法(BF)以及最坏适应算法(WF)的原理，并比较其优缺点。

2.（10 分）多项选择

(1) 虚存是：① 提高运算速度的设备　② 容量扩大了的内存
③ 实际不存在的存储器　④ 进程的地址空间及其内存扩大方法

(2) 临界区是：
① 一个缓冲区　② 一段共享数据区　③ 一段程序　④ 一个互斥资源

(3) 在 UNIX 系统中，用户通过_____读取磁盘文件中的数据？
① 作业申请表　② 原语　③ 系统调用　④ 中断

(4) UNIX 系统 V 的调度原理是基于：
① 时间电调度　② 先来先调度　③ 时间片＋优先级　④ 最短作业优先

3.（8 分）下列程序执行时，"parent：child exited"可能在"child leaving"前面打印吗？为什么？程序执行结果中 a＝？ 为什么？

```
{       ⋮
    a = 55;
    pid = frok ( );
    if (pid == 0) {
        sleep(5);
        a = 99;
        sleep(5);
        printf("child leaving\n");
        exit(0);
    }
    else
    {
        sleep(7);
        printf("a == %d\n", a);
        wait(0);
        printf("parent: child exited\n");
    }
    .......
```

}

4. (10 分)设有进程 A,B,C,分别调用过程 get,copy 和 put 对缓冲区 S 和 T 进行操作。其中 get 负责把数据块输入缓冲区 S,copy 负责从 S 中提取数据块并复制到缓冲区 T 中,put 负责缓冲区 T 中取出信息打印(如图 E1.13),描述 get,copy 及 put 的操作过程。

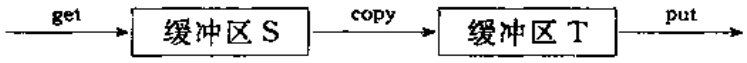


图 E1.13

5. (12 分)描述 UNIX 系统 V 中的缓冲区申请算法 getblk。说明为什么在相应的缓冲区标志了延迟写以后,要启动设备把该块内容写回磁盘,并分配另一个缓冲区给进程?

操作系统综合练习试题 1 解答

1. 最先适应法：把空闲区按首地址从低到高顺序排列，从表头开始搜索，直到找到空闲块大小不小于所需大小的空闲区。

最佳适应法：空闲区从小到大顺序排列，从表头开始搜索，直到找到空闲块大小不小于所需大小的空闲区。

最坏适应法：空闲区按从大到小的顺序排列，每次取第一个空闲区，如果其大小不小于所需大小，则分配，否则放弃。

优缺点：

	速度	空间	特点
FF	最好	中等	使用低地址区
BF	较慢	差(易产生零头)	最接近要求
WF	较慢	好	易于合并，零头少

2. ④，③，③，③。

3. 由于 UNIX 输出时采用缓冲区缓冲数据，尽管本程序采用了同步机制，缓冲区中的数据"parent: child exited"仍可能在"child leaving"前面打印输出。

4. 发送进程 Pa，接收进程 Pb，Pc。

Pa 的私用信号量 Sem(判断缓冲区 S 是否为空)，Pb 的私用信号量 Sfull(从缓冲区 S 接收数据)，Pa 的私用信号量 Tem(判断缓冲区 T 是否为空，Pc 的私用信号量 Tfull(从缓冲区 T 中接收数据)

初值：

Sem = 1，Sfull = 0，Tem = 1，Tfull = 0

描述：

```
Pa:get
   Begin
     P(Sem)
     把数据块写入 S
     V(Sfull)
   End

Pb:copy
   Begin
     P(Sfull)
     P(Tem)
     把数据从 S 中提取到 T 中
     V(Sem)
     V(Tfull)
   End

Pc:put
   Begin
     P(Tfull)
     从 T 中取数据打印
```

```
    V(Tem)
  End
```

5. getblk

```
  输入:逻辑设备号、逻辑块号
  输出:加锁的缓冲区
Begin
  while(未申请到缓冲区)
  do
    if(数据块在设备队列中,且该缓冲区被标记为忙)
    then 等待,进入相应的等待队列
    fi
    if(数据块在设备队列中,且被标记为空闲)
    then 将该块置为忙,并加锁返回
    fi
    if(数据块不在设备队列中且空闲缓冲区队列为空)
    then 等待,进入等待队列
    fi
    if(数据块不在设备队列,且空闲缓冲区队列不为空)
    then 从空闲队列中取一缓冲区
         if 该缓冲区为延迟写
         then 启动I/O,将该块内容写回磁盘
              重新申请缓冲区
         else 上锁该缓冲区,并返回
         fi
    fi
  od
End
```

操作系统综合练习试题 2

1. 填空

(1) 操作系统的主要特点是:(　　),(　　),(　　)。

(2) 在单 CPU 系统中,CPU 和(　　)是并行操作的。

(3) 作业控制方式有:(　　)和(　　)。

(4) 操作系统为编程人员提供的接口是:(　　),为一般用户提供的接口是:(　　)。

(5) UNIX 进程 0 的主要作用有(　　)和(　　)。

2. 名词解释

同步:

互斥:

目录与 i 点:

进程:

线程:

虚存:

3. 设 UNIX 系统 V 的调度优先数计算公式为:

$$p\_pri = p\_cpu/2 + 45 + p\_nice$$

其中 p_cpu 为被计算进程最近一次使用 CPU 后的 CPU 使用时间表述值,其初始值为 0。进程占有 CPU 时,p_cpu 周期性地加 1(每秒增加 60)。进程的 p_cpu 每秒衰减 p_cpu/2。

设 p_nice = 20,且 UNIX 系统 V 按 p_pri 值最小的进程优先获得 CPU 方式调度,进程 Pa,Pb,Pc,Pd 在初始时 p_cpu 为 0。处理机每秒发生一次调度。计算 0 到 5 秒之间,各进程在每秒发生调度时的 p_pri,p_cpu 以及占有 CPU 的进程。

4. 描述 UNIX 系统 V 的 i 节点释放过程 ifree,并指出该过程可能存在的问题。

5. UNIX 系统中,采用异步写与延迟写等方式将数据写回外存,试述异步写与延迟写的区别。

6. UNIX 系统 V 对进程的用户区和系统区采用不同的存储管理方式,即系统区采用分区式管理,然后将对应部分全部装入内存。而用户区部分则采用请求调页管理。问:

(1) 为什么要对系统区进行分区式管理?

(2) 设内存区的分配释放采用位示图方式,且位示图在内存的控制数组结构为

```
mem struct{
    int m_free;     /* 该组空闲页面数 */
    int m_ptr;      /* 位示图位置指针 */
```

```
    int m_avail;   /* 内存中的空闲页面数 */
    short m_pnum[NICMEM];
}
```

试描述内存页面分配过程 memall(base, size)。

操作系统综合练习试题 2 解答

1. 填空

(1) 执行并发、资源共享、用户随机。

(2) 输入/输出设备。

(3) 脱机控制和联机控制。

(4) 系统调用、命令界面。

(5) 交换、调度。

2. 同步：我们把异步环境下的一组并发进程因直接制约而相互发送消息，相互合作，相互等待，使得各进程按一定的速度向前推进的进程称为进程同步。具有同步关系的一组进程称为合作进程，合作进程间相互发送的信号称为消息或事件。

互斥：一组并发进程中的一个或多个程序段，因共享某一临界资源而导致它们必须以一个不允许交叉执行的单位执行。也就是说，不允许两个以上共享资源的并发进程同时进入临界区称为互斥。

进程互斥必须满足如下准则：

(1) 与各并发进程的执行速度无关，即各进程享有平等的、独立的竞争共有资源的权利。

(2) 不在临界区的进程不能阻止其他进程进入临界区。

(3) 当有若干进程申请进入临界区时，只能允许一个进程进入。

(4) 当一个进程申请进入临界区时，应在有限时间内进入。

目录与 i 节点：文件名和相应的文件标识号称为目录。也有系统把文件说明信息称为目录。i 节点在 UNIX 系统或 LINUX 系统中用来存放索引结构等文件说明信息和文件标识号。

进程：一个具有独立功能的程序对某个数据集在处理机上的执行过程和分配资源的基本单位。

线程：在进程的地址空间内，共享进程的各种资源，由寄存器和相应堆栈组成的处理机切换单位。

虚存：由指令的寻址方式所决定的进程寻址空间，以及根据该空间所形成的由内外存组成的存储介质。实现虚存必须满足以下 3 点：

(1) 有相应的地址变换机构，把虚拟(逻辑)地址变换为内存物理地址。

(2) 根据程序执行需要，自动选择指令进入内存。

(3) 有足够大的外存，存储进程的指令与数据，从而使得那些暂不访问的指令和数据都在外存。

3. 解：(表 E1.3)

表 E1.3

t	P_A pri	Cpu 计数	P_B pri	Cpu 计数	P_C pri	Cpu 计数	P_D Pri	Cpu 计数	
0	65	0	65	0	65	0	65	0	P_A 占据 cpu
1	80	60 30	65	0	65	0	65	0	P_B 占据 cpu
2	72	30 15	80	60 30	65	0	65	0	P_C 占据 cpu
3	68	15 7	72	30 15	80	60 30	65	0	P_D 占据 cpu
4	66	7 3	68	15 7	72	30 15	80	60 30	P_A 占据 cpu
5	80	63 31	66	7 3	68	15 7	72	30 15	P_B 占据 cpu

4. 解：在 UNIX System V 中，i 节点按从小到大方式排列，组成 i 节点管理数组。其中铭记 i 节点为 i 节点数组中的最高序号 i 节点。

在释放 i 节点时，如果该 i 节点数组有空位，则可将所释放的 i 节点号置入该空位中。

如果该 i 节点数组已满，且所释放的 i 节点号＞铭记 i 节点，则将该 i 节点开锁后不置入 i 节点数组。

若所释放的 i 节点号＜铭记 i 节点，则将铭记 i 节点号与所释放的 i 节点号交换，以便于再次分配。

算法描述：

```
ifree:
    Begin
    空闲 i 节点数+1
    if i 节点数组空,且数组未上锁
    then i 节点号入数组
    fi
    if  i 节点数组满
    then  比较铭记号与释放 i 节点的序号大小
          if 铭记号大
          then  i 节点锁定位置空闲
          else  把 i 节点的锁定位置空闲,并用该 i 节点号代替铭记 i 节点
          fi
          if    i 节点数组被锁定
          then  释放 i 节点后直接返回
          fi
    End
```

可能存在的问题：

由于i节点数组被锁定时，系统正在从磁盘块中读写相应的i节点号，因此，无法更改i节点数组的铭记i节点。此时，如果被释放的i节点序号小于铭记i节点号的话，由于i节点数组每次从铭记号开始，按从小到大方式读i节点号入数组进行分配，当被释放的i节点序号小于铭记i节点号，且由于i节点数组被锁定后无法进行铭记i节点换号时，就造成了该i节点无法进入i节点数组进行再分配，从而漏掉该i节点。

5. 解：UNIX系统采用异步写、延迟写和同步写三种方式，把数据从缓冲写回外存。延迟写是将装有待写数据的缓冲区放入空闲队列，在不启动写过程时就返回。待其他进程申请缓冲区时，如果分配到该缓冲区，则将其写回外存。

异步写是指系统启动I/O后，不等待传输完成就立即返回。

异步写与延迟写的主要区别是：

延迟写方式将待写数据在缓冲区中尽量多放一段时间，以减少I/O操作的次数。

异步写方式则主要是为了提高进程的执行速度，减少进程的等待时间。

6. 解：(1) 因为系统区主要装载系统程序和相关数据结构，而且这些系统程序和数据一旦装载入内存后就不再换出，这正适合分区式管理的特点。因此，UNIX对系统分区采用分区式管理。

(2) memall (base, size)的描述

```
memall(base, size)
Begin
  if m_avail 小于 size
      then 分配失败,返回
  fi
  while (m_free 小于 size)
      {
          将数组 m_pnum 中的 m_free 个页面填入页表
          将位示图中的相应位置为 0
          m_avail <- m_avail - m_free
          size  <-  size - m_free
          从位示图中 m_ptr 所指位开始读出 NICMEM 个页面入 m_pnum 数组
          m_free <- NICMEM
      }
  从 m_pnum 中取出 size 个页面填入页表
  将位示图中的相应位置为 0
  m_avail <- m_avail - m_free
  size  <-  size - m_free
  返回分配页面数
End
```

操作系统综合练习试题 3

1. 画出 UNIX System V 的进程状态转换图。

2. 名词解释：

系统调用、进程、并发与并行、临界区、同步、死锁、虚存、动态地址重定位、文件系统、中断。

3. UNIX 系统采用一般写，异步写和延迟写 3 种方式将缓冲区中内容写回磁盘。试述这 3 种方式各自的特点。

4. 当系统发生缺页时，试问所缺的页面数据可能在什么地方？画出相应的页面调入处理框图。

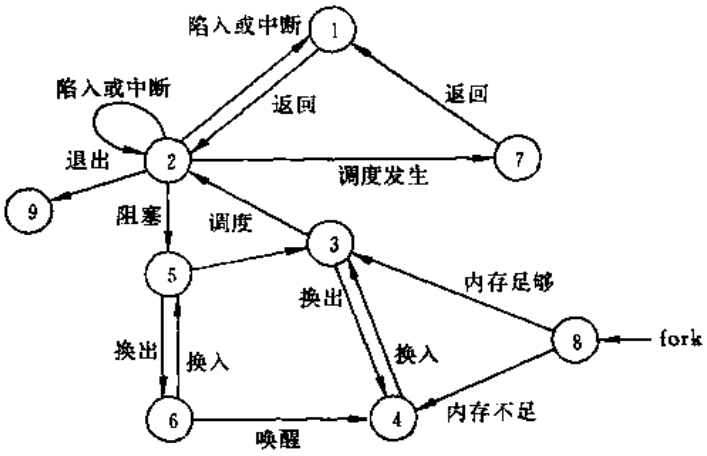
5. 试述成组链法的基本原理，并描述采用成组链法的磁盘块分配过程。

6. 父进程 PA 使用系统调用 fork( )创建了一个子进程 Pa，设 Pa 中有一局部变量 V，且 V 在 Pa 创建之前已被赋值。试问：如果在 Pa 中改变 V 的值，是否会改变 Pa 中 V 的值？

7. 动态分区式管理内存时常用的内存分配与释放算法有哪 3 种？分别简述这 3 种算法的基本原理并比较其优缺点。

操作系统综合练习试题 3 解答

1.（图 E1.14）



1. 用户态执行	6. 外存睡眠
2. 核心态执行	7. 临时状态
3. 内存就绪	8. 初始态
4. 外存就绪	9. 僵死态
5. 内存睡眠	

图 E1.14

2. 系统调用：操作系统提供给编程人员的唯一接口，其指令在核心态下执行。

进程：一个程序对数据集的执行过程。

并发与并行：一组在逻辑上互相独立的程序在执行过程中在执行时间上互相重叠的执行方式，为并发。一组程序按独立的、异常的速度执行为并行。

临界区：不允许多个并发进程交叉执行的一段程序。

同步：异步环境下的一组并发进程因直接制约而互相发送消息，进行合作，使各进程按一定的速度执行的过程。

死锁：一组并发进程因互相请求对方所拥有的资源，在无外力的条件下无法继续执行的状态。

虚存：一个进程的目标代码与数据的虚拟地址组成的空间为虚存，其大小受计算机地址结构限制，虚存的实现具有三个条件：内存中只存储那些经常执行的指令，指令的虚拟地址由硬件地址自动变换机构变换成物理地址，以及在外存中的数据和程序根据执行需要按需调入内存。

动态地址重定位：

(1) 设置 BR 和 VR；

(2) 将程序段首址装入 BR 中；

(3) 将所要访问的虚址送 VR；

(4) 地址变换机构把 VR 和 BR 内容相加得实际地址。

文件系统：与管理文件有关的程序和数据结构。

中断：计算机执行期间，系统发生非预期急需处理事件，使得 CPU 暂时中断当前正在执行的程序，转去执行相应的事件处理程序。

3. 一般写：启动设备写时进程睡眠，等到写结束后唤醒等待进程。

异步写：一次写两块，其中第一块为一般写，第二块则是不等待写结束即返回，从而提高访问外存的速度。

延迟写：等到分配装有该块数据的缓冲区时再将该块内容写入磁盘，从而增加数据在内存的驻留时间。

4. 数据块在缓冲区中，交换区中或在文件系统中。

算法描述如下：

```
if 所需要得到页面在内存中
    then 转地址变换,Exit 返回。fi
if 所需要得到页面在交换区中
    then 启动交换程序,将交换区中相应块调入内存。fi
if 待访问页面在缓冲区中
    then 从缓冲区中把该块数据移入内存。fi
if 所需访问页面在文件系统中
    then 启动文件系统,找出该块在文件系统中的位置,把该块读入内存。fi
```

5. 成组链法的基本原理：

(1) 把文件系统中的外存存储区的空闲块按 50 块一组从后往前划分；

(2) 第 n 组的总块数和块号作为数据存放在第 $n+1$ 组的最后一块中。最后一组的块数和块号放在文件资源表的堆栈中；

(3) 系统初启时将文件资源表中有关堆栈复制入内存；

(4) 文件系统空闲块的分配释放在读入内存的文件资源表管理堆栈上进行；

(5) 若堆栈中存储的块号已分配完毕只剩下最后一块时，锁住堆栈，不再进行空闲块分配。同时，启动外设将最后一块中所记载的块数和块号读入堆栈，待该块中的块数和块号全部读入堆栈后，对堆栈解锁，再继续分配；

(6) 若堆栈满，且又有一新的空闲块被释放时，锁住堆栈，且启动外设，将堆栈中所存放的块号和块数写入新释放的块中后再打开堆栈。

分配过程 alloc：

输入：待分配的块数 n，

输出：块号。

```
Begin
  if 堆栈 filsys 被锁定
```

```
    then 等待 filsys 被开锁
    fi
    ptr 所指块号入 val                    /* ptr 为堆栈指针,val 为存放输出块号的变量 */
    堆栈指针 ptr = ptr - 1
    if ptr = 0                            /* 堆栈中只剩最后一个块号 */
    then 锁住 filsys
         启动外设,从 ptr 所指块号中读入块数与块号
         sleep                            /* 等待读入完成事件唤醒 */
    else 返回 val 中块号
End
```

6. No(否)。

7. (略)。

第二部分　实　验

系统调用函数说明、参数值及定义

1. fork( )

创建一个新进程。

int fork( )

其中返回 int 取值意义如下：

0：创建子进程，从子进程返回的 id 值

大于 0：从父进程返回的子进程 id 值

－1：创建失败

2. lockf(files,function,size)：

用作锁定文件的某些段或者整个文件，本函数适用的头文件为：

```
#include <unistd.h>
```

参数定义：

```
int lockf(files,function,size)
int files,function;
long size;
```

其中：files 是文件描述符；function 是锁定和解锁；1 表示锁定，0 表示解锁。Size 是锁定或解锁的字节数，若用 0，表示从文件的当前位置到文件尾。

3. msgget(key,flag)；

获得一个消息的描述符，该描述符指定一个消息队列以便用于其他系统调用。

该函数使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义

```
int msgget(key,flag)
key_tkey;
int flag;
```

语法格式：msgqid＝msgget(key,flag)

其中：

msgqid 是该系统调用返回的描述符，失败则返回－1；

flag 本身由操作允许权和控制命令值相“或”得到。

如：IPC_CREAT |0400　　是否该队列应被创建；

　　IPC_EXCL |0400　　是否该队列的创建映是互斥的；等。

4. msgsnd(id,msgp,size,flag)：

发送一消息。

该函数使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义：

```
int msgsnd(id,msgp,size,flag)
int id,size,flag;
struct msgbuf * msgp;
```

其中：id 是返回消息队列的描述符；msgp 是指向用户存储区的一个构造体指针，size 指示由 msgp 指向的数据结构中字符数组的长度；即消息的长度。这个数组的最大值由 MSGMAX 系统可调用参数来确定。flag 规定当核心用尽内部缓冲空间时应执行的动作；若在标志 flag 中未设置 IPC_NOWAIT 位，则当该消息队列中的字节数超过一最大值时，或系统范围的消息数超过某一最大值时，调用 msgsnd 进程睡眠。若是设置 IPC_NOWAIT，则在次情况下，msgsnd 立即返回。

5. msgrcv(id,msgp,size,type,flag)：

接受一消息。

该函数调用使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义：

```
int msgrcv(id,msgp,size,flag)
int id,size,type,flag;
struct msgbuf * msgq;
struct msgbuf{long mtype;chat mtext[];};
```

语法格式：

```
count=msgrcv(id,msgp,size,type,flag)
```

其中：id 是消息描述符，msgp 是用来存放欲接收消息的拥护数据结构的地址；size 是 msgp 中数据数组的大小；type 是用户要读的消息类型：

type 为 0：接收该队列的第一个消息；

type 为正：接收类型 type 的第一个消息；

type 为负：接收小于或等于 type 绝对值的最低类型的第一个消息。

Flag 规定倘若该队列无消息，核心应当做什么事，如果此时设置了 IPC_NOWAIT 标志，则立即返回，若在 flag 中设置了 MSG_NOERROR，且所接收的消息大小大于 size，核心截断所接受的消息。

count 是返回消息正文的字节数。

6. msgctl(id,cmd,buf)：

查询一个消息描述符的状态，设置它的状态及删除一个消息描述符。

调用该函数使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义：

```
int msgctl(id,cmd,buf)
int id,cmd;
struct msqid_ds * buf;
```

其中：函数调用成功时返回 0，调用不成功时返回－1。

id 用来识别该消息的描述符；cmd 规定命令的类型。

IPC_STAT 将与 id 相关联的消息队列首标读入 buf。

IPC_SET 为这个消息序列设置有效的用户和小组标识及操作允许权和字节的数量。

IPC_RMID 删除 id 的消息队列。

buf 是含有控制参数或查询结果的用户数据结构的地址。

附：msgid_ds 结构定义如下：

```
struct msgid_ds
{struct ipc_perm msg_perm;      /* 许可权结构 */
short pad1[7];                  /* 由系统使用 */
ushort onsg_qnum;               /* 队列上消息数 */
ushort msg_qbytes;              /* 队列上最大字节数 */
ushort msg_lspid;               /* 最后发送消息的 PID */
ushort msg_lrpid;               /* 最后接收消息的 PID */
time_t msg_stime;               /* 最后发送消息的时间 */
time_t msg_rtime;               /* 最后接收消息的时间 */
```

```
time_t msg_ctime;                /* 最后更改时间 */
};
struct ipc_perm
{ushort uid;                     /* 当前用户 id */
ushort gid;                      /* 当前进程组 id */
ushort cuid;                     /* 创建用户 id */
ushort cgid;                     /* 创建进程组 id */
ushort mode;                     /* 存取许可权 */
{short pad1;long pad2}           /* 由系统使用 */
};
```

7. shmget(key,size,flag):

获得一个共享存储区。

该函数使用头文件如下:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

语法格式:

```
shmid=shmget(key,size,flag)
```

参数定义:

```
int shmget(key,size,flag)
key_t key;
int size,flag;
```

其中:size 是存储区的字节数,key 和 flag 与系统调用 msgget 中的参数含义相同。

附:

操作允许权	八进制数
用户可读	00400
用户可写	00200
小组可读	00040
小组可写	00020
其他可读	00004
其他可写	00002

控制命令	值
IPC_CREAT	0001000
IPC_EXCL	0002000

如:shmid=shmget(key,size,(IPC_CREAT|0400));

创建一个关键字为 key,长度为 size 的共享存储区。

8. shmat(id,addr,flag)：

从逻辑上将一个共享存储区附接到进程的虚拟地址空间上。

该函数调用使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义：

```
char * shmat(id,addr,flag)
int id,flag;
char * addr;
```

语法格式：virtaddr=shmat(id,addr,flag)

其中：id 是共享存储区的标识符，addr 是用户要使用共享存储区附接的虚地址，若 addr 是 0，系统选择一个适当的地址来附接该共享区。flag 规定对此区的读写权限，以及系统是否应对用户规定的地址做舍入操作。如果 flag 中设置了 shm_rnd 即表示操作系统在必要时舍去这个地址。如果设置了 shm_rdonly，即表示只允许读操作。viraddr 是附接的虚地址。

9. shmdt(addr)：

把一个共享存储区从指定进程的虚地址空间断开。

调用该函数使用头文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/mhm.h>
```

参数定义：

```
int shmdt(addr)
char * addr
```

其中，当调用成功时，返回 0 值，调用不成功，返回 -1，addr 是系统调用 shmat 所返回的地址。

10. shmctl(id,cmd,buf)：

对与共享存储区关联的各种参数进行操作，从而对共享存储区进行控制。

调用该函数使用头文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义：

```
int shmctl(id,cmd,buf)
```

```
int id,cmd;
struct shmid_ds * buf;
```

其中：调用成功时返回 0，否则返回－1。id 为被共享存储区的标识符。cmd 规定操作的类型。规定如下：

IPC_STAT： 返回包含在指定的 shmid 相关数据结构中的状态信息，并且把它放置在用户存储区中的 * but 指针所指的数据结构中。执行此命令的进程必须有读取允许权。

IPC_SET： 对于指定的 shmid，为它设置有效用户和小组标识和操作存取权。

IPC_RMID： 删除指定的 shmid 以及与它相关的共享存储区的数据结构。

SHM_LOCK： 在内存中锁定指定的共享存储区，必须是超级用户才可以进行此项操作。

Buf 是一个用户级数据结构地址。

附：

```
shmid_ds
{struct ipc_perm shm_perm;          /* 许可权结构; */
int shm_segsz;                      /* 段大小; */
int padl;                           /* 由系统使用; */
ushort shm_lpid;                    /* 最后操作的进程 id; */
ushort shm_cpid;                    /* 创建者的进程 id; */
ushort shm_nattch;                  /* 当前附界数; */
short pad2;                         /* 由系统使用; */
time_t shm_atime;                   /* 最后附接时间; */
time_t shm_dtime;                   /* 最后断接时间; */
time_t shm_ctime;                   /* 最后修改时间; */
}
```

11. signal(sig,function)：

允许调用进程控制软中断信号的处理。

头文件为：

```
#include<signal.h>
```

参数定义：

```
signal(sig,function)
int sig;
void ( * func)( );
```

其中：sig 的值是：

SIGHVP　　挂起

SIGINT　　键盘按 delete 键或 break 键

SIGQUIT　　键盘按 quit 键

SIGILL	非法指令
SIGIOT	IOT 指令
SIGEMT	EMT 指令
SIGFPE	浮点运算溢出
SIGKILL	要求终止进程
SIGBUS	总线错
SIGSEGV	段违例
SIGSYS	系统调用参数错
SIGPIPE	向无读者管道上写
SIGALRM	闹钟
SIGTERM	软件终结
SIGUSRI	用户定义信号
SIGUSR2	第二个用户定义信号
SIGCLD	子进程死
SIGPWR	电源故障

function 的解释如下：

SIG_DFL：缺省操作。对除 SIGPWR 和 SIGCLD 外所有信号的缺省操作是进程终结对信号 SIGQUIT，SIGILL，SIGTRAP，SIGIOT，SIGEMT，SIGFPE，SIGBUS，SIGSEGV 和 SIGSYS 它产生一内存映像文件。

SIG_IGN：忽视该信号的出现。

Function：在该进程中的一个函数地址，在核心返回用户态时，它以软中断信号的序号作为参数调用该函数，对除了信号 SIGILL，SIGTRAP 和 SIGPWR 以外的信号，核心自动地重新设置软中断信号处理程序的值为 SIG_DFL，一个进程不能捕获 SIGKILL 信号。

Linux 操作系统是一个向用户开放源码的免费的类 UNIX 操作系统。它为在校学生学习操作系统课程提供了一个看得见摸得着的范例。对于学生正确理解，掌握操作系统的基本知识具有重要意义。鉴于此，本操作系统课程涉及的实验均在 Linux 环境下进行。

这就要求大家：

(1) 熟悉 Linux 的操作和开发环境；

(2) 具有 C 语言知识(Linux 操作系统大约 90%的源码是用 C 语言编写)。

如果想安装 Linux 系统，可以利用 Linux 安装光盘进行安装，或向下列网址获取源码进行安装：

ftp://ftp.pk.edu.cn

ftp://ftp.lib.pku.edu.cn

ftp://ftp.ncic.cn.cn

实验1 进程管理

1. 实验目的

(1) 加深对进程概念的理解，明确进程和程序的区别。
(2) 进一步认识并发执行的实质。
(3) 分析进程争用资源的现象，学习解决进程互斥的方法。
(4) 了解 Linux 系统中进程通信的基本原理。

2. 实验预备内容

(1) 阅读 Linux 的 sched.h 源码文件，加深对进程管理概念的理解。
(2) 阅读 Linux 的 fork.c 源码文件，分析进程的创建过程。

3. 实验内容

(1) 进程的创建

编写一段程序，使用系统调用 fork ( )创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示字符“a”；子进程分别显示字符“b”和字符“c”。试观察记录屏幕上的显示结果，并分析原因。

(2) 进程的控制

修改已编写的程序，将每个进程输出一个字符改为每个进程输出一句话，在观察程序执行时屏幕上出现的现象，并分析原因。

如果在程序中使用系统调用 lockf ( )来给每一个进程加锁，可以实现进程之间的互斥，观察并分析出现的现象。

(3) ① 编制一段程序，使其实现进程的软中断通信。

要求：使用系统调用 fork ( )创建两个子进程，再用系统调用 signal ( )让父进程捕捉键盘上来的中断信号(即按 DEL 键)；当捕捉到中断信号后，父进程用系统调用 Kill ( )向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child Processll is Killed by Parent!
Child Process12 is Killed by Parent!

父进程等待两个子进程终止后，输出如下的信息后终止：

Parent Process is Killed!

② 在上面的程序中增加语句 signal (SIGINT, SIG－IGN)和 signal (SIGQUIT, SIG－IGN)，观察执行结果，并分析原因。

(4) 进程的管道通信

编制一段程序,实现进程的管道通信。

使用系统调用 pipe ( )建立一条管道线;两个子进程 P1 和 P2 分别向管道各写一句话:

Child 1 is sending a message!

Child 2 is sending a message!

而父进程则从管道中读出来自于两个子进程的信息,显示在屏幕上。

要求父进程先接收子进程 P1 发来的消息,然后再接收子进程 P2 发来的消息。

4. 思考

(1) 系统是怎样创建流程的?

(2) 可执行文件加载时进行了哪些处理?

(3) 当首次调用新创建进程时,其入口在哪里?

(4) 进程通信有什么特点?

实验 2　进程间通信

1. 实验目的

Linux 系统的进程通信机构(IPC)允许在任意进程间大批量地交换数据。本实验的目的是了解和熟悉 Linux 支持的消息通信机制、共享存储区机制及信息量机制。

2. 实验预备内容

阅读 Linux 系统的 msg.c、sem.c 和 shm.c 等源码文件，熟悉 Linux 的三种通信机制。

3. 实验内容

(1) 消息的创建，发送和接收。

① 使用系统调用 msgget( )，msgsnd( )，msgrev( )及 msgctl( )编制一长度为 1K 的消息的发送和接收程序。

② 观察上面程序，说明控制消息队列系统调用 msgctl( )在此起什么作用？

(2) 共享存储区的创建、附接和断接。

使用系统调用 shmget( )，shmat( )，sgmdt( )，shmctl( )，编制一个与上述功能相同的程序。

(3) 比较上述(1)，(2)两种消息通信机制中数据传输的时间。

实验3 存储管理

1. 实验目的

存储管理的主要功能之一是合理地分配空间。请求页式管理是一种常用的虚拟存储管理技术。

本实验的目的是通过请求页式存储管理中页面置换算法模拟设计，了解虚拟存储技术的特点，掌握请求页式存储管理的页面置换算法。

2. 实验内容

(1) 通过随机数产生一个指令序列，共 320 条指令。指令的地址按下述原则生成：

① 50%的指令是顺序执行的；

② 25%的指令是均匀分布在前地址部分；

③ 25%的指令是均匀分布在后地址部分。

具体的实施方法是：

① 在[0,319]的指令地址之间随机选取一起点 m；

② 顺序执行一条指令，即执行地址为 m+1 的指令；

③ 在前地址[0,m+1]中随机选取一条指令并执行，该指令的地址为 m′；

④ 顺序执行一条指令，其地址为 m′+1；

⑤ 在后地址[m′+2,319]中随机选取一条指令并执行；

⑥ 重复上述步骤①～⑤，直到执行 320 次指令。

(2) 将指令序列变换成为页地址流

设：① 页面大小为 1K；

② 用户内存容量为 4 页到 32 页；

③ 用户虚存容量为 32K。

在用户虚存中，按每 K 存放 10 条指令排列虚存地址，即 320 条指令在虚存中的存放方式为：

第 0 条～第 9 条指令为第 0 页(对应虚存地址为[0,9])；

第 10 条～第 19 条指令为第 1 页(对应虚存地址为[10,19])；

⋮

⋮

第 310 条～第 319 条指令为第 31 页(对应虚存地址为[310,319])。

按以上方式，用户指令可组成 32 页。

(3) 计算并输出下述各种算法在不同内存容量下的命中率。

① 先进先出的算法(FIFO)；

② 最近最少使用算法(LRR)；

③ 最佳淘汰算法(OPT)：先淘汰最不常用的页地址；

④ 最少访问页面算法(LFR)；

⑤ 最近最不经常使用算法(NUR)。

其中③和④为选择内容。

$$命中率=1-\frac{页面失效次数}{页地址流长度}$$

在本实验中，页地址流长度为 320，页面失效次数为每次访问相应指令时，该指令所对应的页不在内存的次数。

3. 随机数产生办法

关于随机数产生办法，Linux 或 UNIX 系统提供函数 srand( )和 rand( )，分别进行初始化和产生随机数。例如：

```
  srand ( );
```

语句可初始化一个随机数；

```
a[0]=10 * rand( )/32767 * 319+1;
a[1]=10 * rand( )/32767 * a[0];
                ⋮
```

语句可用来产生 a[0]与 a[1]中的随机数。

实验 4　文件系统设计

1. 实验目的

通过一个简单多用户文件系统的设计，加深理解文件系统的内部功能及内部实现。

2. 实验内容

为 Linux 系统设计一个简单的二级文件系统。要求做到以下几点：
(1) 可以实现下列几条命令(至少 4 条)；

login	用户登录
dir	列文件目录
create	创建文件
delete	删除文件
open	打开文件
close	关闭文件
read	读文件
write	写文件

(2) 列目录时要列出文件名、物理地址、保护码和文件长度；
(3) 源文件可以进行读写保护。

3. 实验提示

(1) 首先应确定文件系统的数据结构：主目录、子目录及活动文件等。主目录和子目录都以文件的形式存放于磁盘，这样便于查找和修改。

(2) 用户创建的文件，可以编号存储于磁盘上。如 file0，file1，file2...并以编号作为物理地址，在目录中进行登记。

实验 1 指导

【实验内容】

1. 进程的创建

〈任务〉

编写一段程序，使用系统调用 fork( )创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符；父进程显示字符“a”，子进程分别显示字符“b”和“c”。试观察记录屏幕上的显示结果，并分析原因。

〈程序〉

```
#include <stdio.h>
main( )
{
   int p1,p2;
   while ((p1 == fork( )) == -1);          /* 创建子进程 p1 */
   if (p1 = 0)                             /* 子进程创建成功 */
       putchar('b');
   else
   {
   while ((p2 == fork( )) == -1);          /* 创建另一个子进程 */
   if (p2 == 0)                            /* 子进程创建成功 */
       putchar('c');
   else putchar('a');                      /* 父进程执行 */
   }
}
```

〈运行结果〉

```
bca(有时会出现 bac)
```

分析：从进程并发执行来看，输出 bac，acb 等情况都有可能。

原因：fork( )创建进程所需的时间要多于输出一个字符的时间，因此在主进程创建进程 2 的同时，进程 1 就输出了“b”，而进程 2 和主程序的输出次序是有随机性的，所以会出现上述结果。

2. 进程的控制

〈任务〉

修改已编写的程序，将每个进程的输出由单个字符改为一句话，再观察程序执行时屏幕上出现的现象，并分析其原因。如果在程序中使用系统调用 lockf( )来给每个进程加锁，可以实现进程之间的互斥，观察并分析出现的现象。

〈程序 1〉

```
#include <stdio.h>
main( )
{
   int p1,p2,i;
   while ((p1 == fork( )) == -1);
   if (p1 == 0)
       for (i=0; i<500; i++)
           printf("child %d/n", i);
   else
   {
       while ((p2 == fork( )) == -1);
       if (p2 == 0)
           for (i=0; i<500; i++)
               printf("son %d\n", i)
       else
           for(i=0; i<500; i++)
               printf("daughter %d\n",i);
   }
}
```

〈运行结果〉

```
child ...
son ...
daughter ...
daughter ...
或 child
... son
...child
... son
... daughter
... 等
```

分析：由于函数 printf( )输出的字符串之间不会被中断，因此，字符串内部的字符顺序输出时不变。但是，由于进程并发执行时的调度顺序和父子进程的抢占处理机问题，输出字符串的顺序和先后随着执行的不同而发生变化。这与打印单字符的结果相同。

〈程序 2〉

```
#include <stdio.h>
main( )
{
    int p1,p2,i;
       while ((p1 == fork( )) == -1);
    if (p1 == 0)
    {
        lockf(1,1,0);
        for (i=0; i<500; i++)    printf("child %d/n", i);
        lockf(1,0,0);
    }
    else
    {
        while ((p2 == fork( )) == -1);
        if (p2 == 0)
        {
            lockf(1,1,0);
            for (i=0; i<500; i++)    printf("son %d\n", i);
            lockf(1,0,0);
        }
        else
        {
            lockf(1,1,0);
            for(i=0; i<500; i++)     printf("daughter %d\n",i);
            lockf(1,0,0);
        }
    }
}
```

〈运行结果〉

大致与未上锁的输出结果相同，也是随着执行时间不同，输出结果的顺序有所不同。

分析：因为上述程序执行时，不同进程之间不存在共享临界资源（其中打印机的互斥性已由操作系统保证）问题，所以，加锁与不加锁效果相同。

3. 软中断通信

〈任务 1〉

编制一段程序，使用系统调用 fork( )创建两个子进程，再用系统调用 signal( )让父进程捕捉键盘上来的中断信号（即按 Del 键），当捕捉到中断信号后，父进程用系统调用 kill( )向两个子进程发出信号，子进程捕捉到信号后，分别输出下列信息后终止：

```
child process 1 is killed by parent!
child process 2 is killed by parent!
```

父进程等待两个子进程终止后，输出以下信息后终止：

parent process is killed!

〈程序〉

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void waiting( ), stop( );
int wait_mark;

main( )
{
    int p1,p2;
    while ((p1 == fork( )) == -1);          /* 创建进程 p1 */
    if (p1 > 0)
    {
        while ((p2 == fork( )) == -1);
        if (p2 > 0)
        {
            wait_mark = 1;
            signal(SIGINT, stop);           /* 接收'Del'信号,并转 stop */
            waiting(0);
            kill(p1, 16);                   /* 向 p1 发中断信号 16 */
            kill(p2, 17);                   /* 向 p2 发中断信号 17 */
            wait(0);                        /* 同步 */
            wait(0);
            printf("parent process is killed! \n");
            exit(0);
        }
        else
        {
            wait_mark = 1;
            signal(17, stop);
            waiting( );
            lockf(stdout, 1, 0);
            printf("child process 2 is killed by parent! \n");
            lockf(stdout, 0, 0);
            exit(0);
        }
    }
    else
```

```
    {
        wait_mark = 1;
        signal(16, stop);
        waiting( );
        lockf(stdout, 1, 0);
        printf("child process 1 is killed by parent! \n");
        lockf(stdout, 0, 0);
        exit(0);
    }
}
void waiting( )
{
    while (wait_mark != 0);
}
void stop( )
{
    wait_mark = 0;
}
```

〈运行结果〉

```
child process 1 is killed by parent!
child process 2 is killed by parent!
Parent process is killed!
```

分析

(1) 上述程序中，实用函数 signal( )都放在一段程序的前面部位，而不是在其他接收信号处。这是因为 signal( )的执行只是为进程指定信号量 16 或 17 的作用，以及分配相应的与 stop( )过程链接的指针。从而，signal( )函数必须在程序前面部分执行。

(2) 该程序段前面部分用了两个 wait(0)，为什么？请读者思考。

(3) 该程序段中每个进程退出时都用了语句 exit(0)，为什么？请读者思考。

〈任务 2〉

在上面任务 1 中，增加语句 signal(SIGINT, SIG_IGN)和语句 signal(SIGQUIT, SIG_IGN)，观察执行结果，并分析原因。这里，signal(SIGINT, SIG_IGN)和 signal(SIGQUIT, SIG_IGN)分别为忽略'Del'键信号以及忽略中断信号。

〈程序〉

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
int pid1,pid2;
```

```
int EndFlag = 0;
pf1 = 0;
pf2 = 0;
void IntDelete( )
{
    kill(pid1, 16);
    kill(pid2, 17);
    EndFlag = 1;
}
void Int1( )
{
    printf("child process 1 is killed ! by parent\n");
    exit(0);
}
void Int2( )
{
    printf("child process 2 is killed ! by parent\n");
    exit(0);
}

main( )
{
    int exitpid;
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
    while ((pid1 == fork( )) == -1);
    if (pid == 0)
    {
        signal(SIGUSR1, Int1);
        signal(SIGINT, SIG_IGN);
        pause( );
        exit(0);
    }
    else
    {
        while ((pid == fork( )) == -1);
        if (pid2 == 0)
        {
            signal(SIGUSR1, Int1);
            signal(SIGINT, SIG_IGN);
            pause( );
            exit(0);
        }
```

```
        else
        {
            signal(SIGINT, IntDelete);
            waitpid(-1, &exitcode, 0);
            printf("parent process is killed \n");
            exit(0);
        }
    }
}
```

〈运行结果〉

请读者将上述程序输入计算机后，执行并观察。

〈分析〉 由于忽略了中断与退出信号，程序会一直保持阻塞状态而无法退出。

4. 进程的管道通信

〈任务〉

编制一段程序，实现进程的管道通信。使用系统调用 pipe( )建立一条管道线。两个子进程 p1 和 p2 分别向管道各写一句话：

```
child 1 is sending message!
child 2 is sending message!
```

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。

〈程序〉

```
#include <unisted.h>
#include <signal.h>
#include <stdio.h>
int pid1, pid2;

main( )
{
    int fd[2];
    char OutPipe[100], InPipe[100];
    pipe(fd);
    while ((pid1 == fork( )) == -1);
    if (pid1 == 0)
    {
        lockf(fd[1], 1, 0);
        sprintf(OutPipe, "child 1 process is sending message!");
        write(fd[1], OutPipe, 50);
        sleep(5);
```

```
        lockf(fd[1], 0, 0);
        exit(0);
    }
    else
    {
        while ((pid2 = fork( )) == -1);
        if (pid2 == 0)
        {
            lockf(fd[1], 1, 0);
            sprintf(OutPipe, "child 1 process is sending message!");
            write(fd[1], OutPipe, 50);
            sleep(5);
            lockf(fd[1], 0, 0);
            exit(0);
        }
        else
        {
            wait(0);
            read(fd[0], InPipe, 50);
            printf("%s\n", InPipe);
            wait(0);
            read(fd[0], InPipe, 50);
            exit(0);
        }
    }
}
```

〈运行结果〉

```
child 1 is sending message!
child 2 is sending message!
```

〈分析〉

请读者自行完成。

实验 2 指导

【实验内容】

1. 消息的创建、发送和接收

〈任务〉

使用系统调用 msgget( ),msgsnd( ),msgrev( ),及 msgctl( )编制一长度为 1K 的消息发送和接收的程序。

〈程序设计〉

(1) 为了便于操作和观察结果,用一个程序作为“引子”,先后 fork()两个子进程,SERVER 和 CLIENT ,进行通信。

(2) SERVER 端建立一个 Key 为 75 的消息队列,等待其他进程发来的消息。当遇到类型为 1 的消息,则作为结束信号,取消该队列,并退出 SERVER 。SERVER 每接收到一个消息后显示一句“(server)received”。

(3) CLIENT 端使用 key 为 75 的消息队列,先后发送类型从 10 到 1 的消息,然后退出。最后的一个消息,即是 SERVER 端需要的结束信号。CLIENT 每发送一条消息后显示一句“(client)sent”。

(4) 父进程在 SERVER 和 CLIENT 均退出后结束。

〈程序〉

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#define MSGKEY 75                    /* 定义关键词 MEGKEY */
struct msgform                       /* 消息结构 */
{
    long mtype;
    char mtext[1030];                /* 文本长度 */
}msg;
int msgqid,i;
void CLIENT( )
{   int i;
    msgqid=msgget(MSGKEY,0777);
    for (i=10;i>=1;i--)
    {
        msg.mtype=i;
```

```
            printf("(client)sent\n");
            msgsnd(msgqid,&msg,1024,0);      /* 发送消息 msg 入 msgid 消息队列 */
        }
        exit(0);
}
void SERVER( ) .
{
      msgqid=msgget(MSGKEY,0777|IPC_CREAT);              /* 由关键字获得消息队列 */
      do
      {
            msgrcv (msgqid,&msg,1030,0,0);             /* 从 msgqid 队列接收消息 msg   */
            printf("(server)received\n");
       }while(msg.mtype!=1);                             /* 消息类型为 1 时,释放队列 */
       msgctl(msgqid,IPC_RMID,0);
       exit(0);
}
void main( )
{
       while((i=fork( ))==-1);
       if (! i) SERVER( );
       while((i=fork( ))==-1);
       if (! i) CLIENT( );
       wait(0);
       wait(0);
}
```

〈结果〉

从理想的结果来说,应当是每当 Clinet 发送一个消息后,Server 接收该消息,Clinet 再发送下一条。也就是说"(Clinet)sent"和"(server)received"的字样应该在屏幕上交替出现。实际的结果大多是,先由 Clinet 发送了两条消息,然后 Server 接收一条消息。此后 Clinet-Server 交替发送和接收消息。最后 Server 一次接收两条消息。Client 和 Server 分别发送和接收了 10 条消息,与预期设想一致。

〈分析〉

message 的传送和控制并不保证完全同步,当一个程序不在激活状态的时候,它完全可能继续睡眠,造成了上面的现象,在多次 send message 后才 receive message。这一点有助于理解消息传送的实现机理。

2. 共享存储区的创建,附接和断接

〈任务〉

使用系统调用 shmget( ),sgmat( ),smgdt( ),shmctl( ),编制一个与上述相同功能的程序。

（程序设计）

（1）为了便于操作和观察结果，用一个程序作为“引子”，先后 fork()两个子进程，SERVER 和 CLIENT ，进行通信。

（2）SERVER 端建立一个 Key 为 75 的共享区，并将第一个字节置为－1。作为数据空的标志。等待其他进程发来的消息。当该字节的值发生变化时，表示收到了信息，进行处理。然后再次把它的值设为－1。如果遇到的值为 0，则视为结束信号，取消该队列，并退出 SERVER。SERVER 每接收到一次数据后显示“(server)received”。

（3）CLIENT 端建立一个 Key 为 75 的共享区，当共享取得第一个字节为－1 时，Server 端空闲，可发送请求。CLIENT 随即填入 9 到 0。期间等待 Server 端的再次空闲。进行完这些操作后，CLIENT 退出。CLIENT 每发送一次数据后显示“(client)sent”。

（4）父进程在 SERVER 和 CLIENT 均退出后结束。

（程序）

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#define SHMKEY 75                          /* 定义共享区关键词 */
int shmid,i;
int *addr;

void CLIENT()
{    int i;
     shmid=shmget(SHMKEY,1024,0777);/* 获取共享区，长度 1024，关键词 SHMKEY */
     addr=shmat(shmid,0,0);       /* 共享区起始地址为 addr */
     for (i=9;i>=0;i--)
     {
          while(*addr!=-1);
          printf("(client)sent\n");       /* 打印(client)sent */
          *addr=i;                                    /* 把 i 赋给 addr */
      }
      exit(0);
}

void SERVER()
{
     shmid=shmget(SHMKEY,1024,0777|IPC_CREAT);          /* 创建共享区 */
     addr=shmat(shmid,0,0);                          /* 共享区起始地址为 addr */
     do
     {
          *addr=-1;
          while(*addr==-1);
```

```
            printf("(server)received\n");                    /* 服务进程使用共享区 */
        }while(*addr);
        shmctl(shmid,IPC_RMID,0);
        exit(0);
}

void main( )
{
        while((i=fork( ))==-1);
        if (!i) SERVER( );
        while((i=fork( ))==-1);
        if (!i) CLIENT( );
        wait(0);
        wait(0);
}
```

〈结果〉

运行的结果和预想的完全一样。但在运行的过程中，发现每当 client 发送一次数据后，server 要等待大约 0.1 秒才有响应。同样，之后 client 又需要等待约 0.1 秒才发送下一个数据。

〈分析〉

出现上述的应答延迟的现象是程序设计的问题。当 client 端发送了数据后，并没有任何措施通知 server 端数据已经发出，需要由 client 的查询才能感知。此时，client 端并没有放弃系统的控制权，仍然占用 CPU 的时间片。只有当系统进行调度时，切换到了 server 进程，再进行应答。这个问题，也同样存在于 server 端到 client 的应答过程之中。

3. 比较两种消息通信机制中的数据传输的时间

由于两种机制实现的机理和用处都不一样，难以直接进行时间上的比较。如果比较其性能，应更加全面地分析。

(1) 消息队列的建立比共享区的设立消耗的资源少。前者只是一个软件上设定的问题，后者需要对硬件操作，实现内存的映像，当然控制起来比前者复杂。如果每次都重新进行队列或共享的建立，共享区的设立没有什么优势。

(2) 当消息队列和共享区建立好后，共享区的数据传输，受到了系统硬件的支持，不耗费多余的资源；而消息传递，由软件进行控制和实现，需要消耗一定的 CPU 资源。从这个意义上讲，共享区更适合频繁和大量的数据传输。

(3) 消息的传递，自身就带有同步的控制。当等到消息的时候，进程进入睡眠状态，不再消耗 CPU 资源。而共享队列如果不借助其他机制进行同步，接收数据的一方必须进行不断的查询，白白浪费了大量的 CPU 资源。可见，消息方式的使用更加灵活。

实验 3 指导

【实验内容】

〈任务〉

设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。

(1) 先进先出的算法(FIFO)

(2) 最近最少使用算法(LRU)

(3) 最佳淘汰算法(OPT)

(4) 最少访问页面算法(LFU)

(5) 最近最不经常使用算法(NUR)

命中率＝(1－页面失效次数)/页地址流长度

〈程序设计〉

本实验的程序设计基本上按照实验内容进行。即首先用 Srand( )和 rand ( )函数定义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算法计算出相应的命中率。相关定义如下：

1. 数据结构

(1) 页面类型

```
typedef struct{
            int pn,pfn,counter ,time;
              }pl-type;
```

其中 pn 为页号，pfn 为面号，counter 为一个周期内访问该页面次数，time 为访问时间。

(2) 页面控制结构

```
pfc_struct {
          int     pn, pfn;
          struct    pfc_struct    * next;
            };
typedef    struct pfc_struct pfc_type;
pfc_type    pfc[total_vp],  * freepf_head , * busypf_head;
pfc_type  * busypf_tail;
```

其中 pfc[total_vp]定义用户进程虚页控制结构，

* freepf_head 为空页面头的指针，

* busypf_head 为忙页面头的指针，

* busypf_tail 为忙页面尾的指针。

2. 函数定义

(1) Void initialize( )：初始化函数，给每个相关的页面赋值。

(2) Void FIFO( )：计算使用 FIFO 算法时的命中率。

(3) Void LRU( )：计算使用 LRU 算法时的命中率。

(4) Void OPT( )：计算使用 OPT 算法时的命中率。

(5) Void LFU( )：计算使用 LFU 算法时的命中率。

(6) Void NUR( )：计算使用 NUR 算法时的命中率。

3. 变量定义

(1) int a[tatal_instruction]：指令流数据组。

(2) int page[total_instruction]：每条指令所属页号。

(3) int offset[total_instruction]：每页装入 10 条指令后取模运算页号偏移值。

(4) int total_pf：用户进程的内存页面数。

(5) int diseffect：页面失效次数。

4. 程序流程图

(略)

〈程序〉

```
#define TRUE 1
#define FALSE 0
#define INVALID -1
#define null 0

#define total_instruction 320                    /* 指令流长 */
#define total_vp 32                              /* 虚页长 */
#define clear_period 50                          /* 清零周期 */

typedef struct{                                  /* 页面结构 */
    int pn,pfn,counter,time;
}pl_type;
pl_type pl[total_vp];                            /* 页面结构数组 */
struct pfc_struct{                               /* 页面控制结构 */
      int pn,pfn;
      struct pfc_struct *next;
};
typdef struct pfc_struct pfc_type;
pfc_type pfc[total_vp], *freepf_head, *busypf_head, *busypf_tail;

int diseffect,␣␣a[total_instruction];
int page[total_instruction],␣␣offset[total_instruction];
```

```
void initialize( );
void FIFO( );
void LRU( );
void OPT( );
void LFU( );
void NUR( );

main ( )
{
  int S,i,j;

  srand(getpid( ) * 10);   /* 由于每次运行时进程号不同,故可用来作为初始化随机数队列的"种
                              子" */
  S=(float)319 * rand( )/32767+1;
  for (i=0;i<total_instruction;i+=4)                    /* 产生指令队列 */
  {
    a[i]=S;                                             /* 任选一指令访问点 */
    a[i+1]=a[i]+1;                                      /* 顺序执行一条指令 */
    a[i+2]=(float)a[i] * rand( )/32767;                 /* 执行前地址指令 m' */
    a[i+3]=a[i+2]+1;                                    /* 执行后地址指令 */
    S=(float)rand( ) * (318-a[i+2])/32767+a[i+2]+2;
  }
  for (i=0;i<total_instruction;i++)                     /* 将指令序列变换成页地址流 */
  {
    page[i]=a[i]/10;
    offset[i]=a[i]%10;
  }
  for (i=4;i<=32;i++)                                   /* 用户内存工作区从 4 个页面到 32 个页面 */
  {
    printf("%2d page frames",i);
    FIFO(i);
    LRU(i);
    OPT(i);
    LFU(i);
    NUR(i);
    printf("\n");
  }
}

void FIFO (total_pf)                                    /* FIFO(First in First Out) ALGORITHM */
int total_pf;                                           /* 用户进程的内存页面数 */
{  int i,j;
   pfc_type *p, *t;
```

```
  initialize(total_pf);                          /* 初始化相关页面控制用数据结构 */
  busypf_head=busypf_tail=NULL;                  /* 忙页面队列头,队列尾链接 */
  for (i=0;i<total_instruction;i++)
  {
  if (pl[page[i]].pfn == INVALID)                /* 页面失效 */
  {diseffect+=1;                                 /* 失效次数 */
  if (freepf_head==NULL)                         /* 无空闲页面 */
{ p=busypf_head->next;
pl[busypf_head->pn].pfn=INVALID;
  freepf_head=busypf_head;                       /* 释放忙页面队列中的第一个页面 */
freepf_head->next=NULL;
busypf_head=p;
}
p=freepf_head->next;                             /* 按 FIFO 方式调新页面入内存页面 */
freepf_head->next=NULL;
freepf_head->pn=page[I];
pl[page[i]].pfn=freepf_head->pfn;
if (busypf_tail==NULL)
        busypf_head=busypf_tail=freepf_head;
else
{busypf_tail->next=freepf_head);
busypf_tail=freepf_head;
}
freepf_head=p;
}
}
printf("FIFO:%6.4f",1-(float)diseffect/320);
}

void LRU(total_pf)          /* LRU(Last Recently Used)ALGORITHM */
    int total_pf;
{ int min,minj,i,j,present_time;

initialize(total_pf);present_time=0;
for(i=0;i<total_instruction;i++)
{  if (pl[page[i]].pfn=INVALID)                  /* 页面失效 */
{diseffect++;
  if (freepf_head=NULL)                          /* 无空闲页面 */
{min=32767;
for(j=0;j<total_vp;j++)
  if (min>pl[j].time && pl[j].pfn!=INVALID)
```

```
    {min=pl[j].time;minj=j;}
  freepf_head=&pfc[pl[minj].pfn];
  pl[minj].pfn=INVALID;
  pl[minj].time= -1;
  freepf_head->next=NULL;
  }
  pl[page[i]].pfn=freepf_head->pfn;
  pl[page[i]].time=present_time;
  freepf_head=freepf_head->next;
  }
  else
  pl[page[i]].time=present_time;
  present_time++;
  }
  printf("LRU: %6.4f",1-(float)diseffect/320);
  }


  void NUR(total_pf)

  int total_pf;
  {
  int i,j,dp,cont_flag,old_dp;
  pfc_type *t;

  initialize(total_pf);
  dp=0;
  for (i=0;i<total_instruction; i++)
  {
  if (pl[page[i]].pfn==INVALID)                    /*页面失效*/
  { diseffect++;
    if (freepf_head==NULL)                         /*无空闲页面*/
  { cont_flag=TRUE;old_dp=dp;
  while(cont_flag)
  if(pl[dp].counter==0 && pl[dp].pfn!=INVALID)
    cont_flag=FALSE;
  else
  { dp++;if(dp==tofal_vp) dp=0;
    if(dp==old_dp)
  for(j=0;j<total_vp;j++) pl[j].counter=0;
  }
  freept_head=&pfc[pl[dp].pfn];
  pl[dp].pfn=INVALID;
```

```
freepf_head->next=NULL;
}

pl[page[i]].pfn=freepf_head->pfn;
freepf_head=freepf_head->next;
}
else
        pl[page[i]].counter=1;
   if(i%clear_period==0)
     for(j=0;j<total_vp;j++)
     pl[j].counter=0;
}
printf("NUR:%6.4f",1-(float)diseffect/320);
}

void OPT (total_pf)              /*OPT (Optimal Replacement)ALGORITHM*/
int total_pf;
{ int i,j,max,maxpage,d,dist[total_vp];
pfc_type *t;

initialize(total_pf);
for(i=0;i<total_instruction;i++)
{
if (pl[page[i]].pfn==INVALID]
{ diseffect++;
if (freepf_head==NULL)
{for(j=0;j<total_vp;j++)
if(pl[j].pfn!=INVALID) dist[j]=32767;else dist[j]=0;
d=1;
for (j=i+1;j<total_instruction;j++)
{if (pl[page[j]].pfn!=INVALID)
dist[page[j]]=d;
d++;
}
max=-1;
for(j=0;j<total_vp;j++)
if(max<dist[j])
{max=dist[j];maxpage=j;}
freepf_head = &pfc[pl[maxpage].pfn];
freepf_head->next=NULL;
pl[maxpage].pfn=INVALID;
}
pl[page[i]].pfn=freepf_head->pfn;
```

```
freepf_head=freepf_head->next;
}
}
printf("OPT:%6.4f",1-(float)diseffect/320);
}

void LFU(total_pf)                          /*LFU(leat Frequently Used) ALGORITHM*/
int total_pf;
{ int i,j,min,minpage;
pfc_type *t;

initialize(total_pf);
for(i=0;i<total_instruction;i++)
{
if (pl[page[i]].pfn==INVALID)
{diseffect++;
if (freepf_head==NULL)
{min=32767;
  for(j=0;j<total_vp;j++)
{ if(min>pl[j].counter && pl[j].pfn!=INVALID)
  {min=pl[j].counter;minpage=j;}

pl[j].counter=0;
}
freepf_head=&pfc[pl[minpage].pfn];
pl[minpage].pfn=INVALID;
freepf_head->next=NULL;
}
pl[page[i]].pfn=freepf_head->pfn;
freepf_head=freepf_head->next;
}
else

pl[page[i]].counter++;
}
printf("LFU:%6.4f",1-(float)diseffect/320);
}
void initialize(total_pf)                   /*初始化相关数据结构*/
    int total_pf;                           /*用户进程的内存页面数*/
{ int i;
diseffect=0;
for(i=0;i<total_vp;i++)
{ pl[i].pn=i; pl[i].pfn=INVALID;             /*置页面控制结构中的页号,页面为空*/
```

```
pl[i].counter=0;pl[i].time=-1;                    /*页面控制结构中的访问次数为0,时间为-1*/
}
for(i=1;i<total_pf;i++)
{pfc[i-1].next=&pfc[i];pfc[i-1].pfn=i-1;}          /*建立pfc[i-1]和pfc[i]之间的链接*/
pfc[total_pf-1].next=NULL;pfc[total_pf-1].pfn=total_pf-1;
freepf_head=&pfc[0];                              /*空页面队列的头指针为pfc[0]*/
}
```

〈结果〉

```
4  page frames FIFO: 0.4969  LRU: 0.5000  OPT: 0.5281  LFU: 0.4969  NUR: 0.5000
5  page frames FIFO: 0.5188  LRU: 0.5125  OPT: 0.5531  LFU: 0.5125  NUR: 0.5062
6  page frames FIFO: 0.5281  LRU: 0.5188  OPT: 0.5875  LFU: 0.5344  NUR: 0.5344
7  page frames FIFO: 0.5406  LRU: 0.5500  OPT: 0.6094  LFU: 0.5625  NUR: 0.5562
8  page frames FIFO: 0.5500  LRU: 0.5719  OPT: 0.6154  LFU: 0.5875  NUR: 0.5531
9  page frames FIFO: 0.5625  LRU: 0.5812  OPT: 0.6438  LFU: 0.6094  NUR: 0.5781
10 page frames FIFO: 0.5844  LRU: 0.5969  OPT: 0.6594  LFU: 0.6344  NUR: 0.5969
11 page frames FIFO: 0.5938  LRU: 0.6094  OPT: 0.6750  LFU: 0.6469  NUR: 0.6250
12 page frames FIFO: 0.6156  LRU: 0.6281  OPT: 0.7031  LFU: 0.6594  NUR:0.6594
13 page frames FIFO: 0.6375  LRU: 0.6344  OPT: 0.7312  LFU: 0.67169 NUR: 0.6500
14 page frames FIFO: 0.6844  LRU: 0.6625  OPT: 0.7469  LFU: 0.6937  NUR: 0.6500
15 page frames FIFO: 0.6844  LRU: 0.6812  OPT: 0.7656  LFU: 0.7125  NUR: 0.6875
16 page frames FIFO: 0.7062  LRU: 0.7062  OPT: 0.7750  LFU: 0.7188  NUR: 0.7094
17 page frames FIFO: 0.7094  LRU: 0.7125  OPT: 0.7844  LFU: 0.7281  NUR: 0.7250
18 page frames FIFO: 0.7188  LRU: 0.7281  OPT: 0.8000  LFU: 0.7375  NUR: 0.7344
19 page frames FIFO: 0.7281  LRU: 0.7531  OPT: 0.8062  LFU: 07562   NUR: 0.7531
20 page frames FIFO: 0.7281  LRU: 0.7656  OPT: 0.8219  LFU: 0.7812  NUR: 0.7594
21 page frames FIFO: 0.7812  LRU: 0.7781  OPT: 0.8312  LFU: 0.7906  NUR: 0.7906
22 page frames FIFO: 0.7875  LRU: 0.7937  OPT: 0.8406  LFU: 0.8062  NUR: 0.8125
23 page frames FIFO: 0.7960  LRU: 0.8094  OPT: 0.8562  LFU: 0.8187  NUR: 0.8187
24 page frames FIFO: 0.8000  LRU: 0.8219  OPT: 0.8656  LFU: 0.8312  NUR: 0.8219
25 page frames FIFO: 0.8344  LRU: 0.8312  OPT: 0.8719  LFU: 0.8500  NUR: 0.8344
26 page frames FIFO: 0.8625  LRU: 0.8438  OPT: 0.8781  LFU: 0.8625  NUR: 0.8594
27 page frames FIFO: 0.8625  LRU: 0.8652  OPT: 0.8844  LFU: 0.8688  NUR: 0.8781
28 page frames FIFO: 0.8750  LRU: 0.8656  OPT: 0.8906  LFU: 0.8750  NUR: 0.8812
29 page frames FIFO: 0.8844  LRU: 0.8781  OPT: 0.8969  LFU: 0.8812  NUR: 0.8812
30 page frames FIFO: 0.8875  LRU: 0.8875  OPT: 0.9000  LFU: 0.8875  NUR: 0.8906
31 page frames FIFO: 0.8875  LRU: 0.8906  OPT: 0.9000  LFU: 0.8969  NUR: 0.9000
32 page frames FIFO: 0.9000  LRU: 0.9000  OPT: 0.9000  LFU: 0.9000  NUR: 0.9000
```

〈分析〉

从上述结果可知,在内存页面数较少(4～5 页面)时,5 种算法的命中率差别不大,都是50%左右。在内存页面为 7～25 个页面之间时,5 种算法的访内命中率大致在 52%至 87%之间变化。但是,FIFO 算法与 OPT 算法之间的差别一般在 6～10 个百分点左右。在内存页

面为 25～32 个页面时，由于用户进程的所有指令基本上都已装入内存，从而命中率已增加较大。从而算法之间的差别不大。

比较上述 5 种算法，以 OPT 算法的命中率最高，NUR 算法次之，再就是 LFU 算法和 LRU 算法，其次是 FIFO 算法。

实验 4 指导

【实验内容】

〈任务〉

为 Linux 系统设计一个简单的二级文件系统。要求做到以下几点：

1. 可以实现下列几条命令：

login	用户登录
dir	列目录
create	创建文件
delete	删除文件
open	打开文件
close	关闭文件
read	读文件
write	写文件

2. 列目录时要列出文件名，物理地址，保护码和文件长度

3. 源文件可以进行读写保护

〈程序设计〉

1. 设计思想

本文件系统采用两级目录，其中第一级对应于用户账号，第二级对应于用户账号下的文件。另外，为了简单本文件系统未考虑文件共享、文件系统安全以及管道文件与设备文件等特殊内容。对这些内容感兴趣的读者，可以在本系统的程序基础上进行扩充。

2. 主要数据结构

(1) i 节点

```
struct inode {
struct inode *i_forw;
struct inode *i_back;
char i_flag;
unsigned int i_ino;                    /* 磁盘 i 节点标号 */
unsigned int i_count;                  /* 引用计数 */
unsigned short di_number;              /* 关联文件数，当为 0 时，则删除该文件 */
unsigned short di_mode;                /* 存取权限 */
unsigned short di_uid;                 /* 磁盘 i 节点用户 id */
unsigned short di_gid;                 /* 磁盘 i 节点组 id */
```

```
unsigned int di_addr [NADDR];          /* 物理块号 */
```

(2) 磁盘 i 节点

```
Struct dinode
    {
      unsigned short di_number;        /* 关联文件数 */
      unsigned short di_mode;          /* 存取权限 */

      unsigned short di_uid;
      unsigned short di_gid;
      unsigned long di_size;           /* 文件大小 */
      unsigned int di_addr [NADDR];    /* 物理块号 */
}
```

(3) 目录项结构

```
Struct direct
    {
      char d_name [DIRSIZ];            /* 目录名 */
      unsigned int d_ino;              /* 目录号 */
    }
```

(4) 超级块

```
Struct filsys
    {
      unsigned short s_isize;              /* i 节点块块数 */
      unsigned long s_fsize;               /* 数据块块数 */

      unsigned int s_nfree;                /* 空闲块块数 */
      unsigned short s_pfree;              /* 空闲块指针 */
      unsigned int s_free [NICFREE];       /* 空闲块堆栈 */

      unsigned int s_ninode;               /* 空闲 i 节点数 */
      unsigned short s_pinode;             /* 空闲 i 节点指针 */
      unsigned int s_inode [NICINOD];      /* 空闲 i 节点数组 */
      unsigned int s_rinode;               /* 铭记 i 节点 */

      char s_fmod;                         /* 超级块修改标志 */
    };
```

(5) 用户密码

```
Struct pwd
    {
      unsigned short P_uid;
```

```
    unsigned short P_gid;
    char passward [PWOSIZ];
  };
```

(6) 目录

```
Struct dir
  {
    struct direct direct [DIRNUM];
    int size;
  };
```

(7) 查找内存 i 节点的 hash 表

```
Struct hinode
  {
    struct inode *i_forw;
  };
```

(8) 系统打开表

```
Struct file
  {
    char f_flag;                          /* 文件操作标志 */
    unsigned int f_count;                 /* 引用计数 */
    struct inode *f_inode;                /* 指向内存 i 节点 */
    unsigned long f_off;                  /* 读/写指针 */
  };
```

(9) 用户打开表

```
Struct user
  {
    unsigned short u_default_mode;
    unsigned short u_uid;                 /* 用户标志 */
    unsigned short u_gid;                 /* 用户组标志 */
    unsigned short u_ofile [NOFILE];      /* 用户打开表 */
  };
```

3. 主要函数

(1) i 节点内容获取函数 iget( )(详细描述略)。

(2) i 节点内容释放函数 iput( )(详细描述略)。

(3) 目录创建函数 mkdir( )(详细描述略)。

(4) 目录搜索函数 namei( )(详细描述略)。

(5) 磁盘块分配函数 balloc( )(详细描述略)。

(6) 磁盘块释放函数 bfree( )(详细描述略)。

(7) 分配 i 节点区函数 ialloc( )(详细描述略)。

(8) 释放 i 节点区函数 ifree( )(详细描述略)。

(9) 搜索当前目录下文件的函数 iname( )(详细描述略)。

(10) 访问控制函数 access( )(详细描述略)。

(11) 显示目录和文件用函数_dir( )(详细描述略)。

(12) 改变当前目录用函数 chdir( )(详细描述略)。

(13) 打开文件函数 open( )(详细描述略)。

(14) 创建文件函数 create( )(详细描述略)。

(15) 读文件用函数 read( )(详细描述略)。

(16) 写文件用函数 write( )(详细描述略)。

(17) 用户登录函数 login( )(详细描述略)。

(18) 用户退出函数 logout( )(详细描述略)。

(19) 文件系统格式化函数 format( )(详细描述略)。

(20) 进入文件系统函数 install( )(详细描述略)。

(21) 关闭文件函数 close( )(详细描述略)。

(22) 退出文件系统函数 halt( )(详细描述略)。

(23) 文件删除函数 delete( )(详细描述略)。

4. 主程序说明

Begin

Step1	对磁盘进行格式化
Step2	调用 install( ),进入文件系统
Step3	调用_dir( ),显示当前目录
Step4	调用 login( ),用户注册
Step5	调用 mkdir( )和 chdir( )创建目录
Step6	调用 creat( ),创建文件 0
Step7	分配缓冲区
Step8	写文件 0
Step9	关闭文件 0 和释放缓冲
Step10	调用 mkdir( )和 chdir( )创建子目录
Step11	调用 creat( ),创建文件 1
Step12	分配缓冲区
Step13	写文件 1
Step14	关闭文件 1 和释放缓冲
Step15	调用 chdir 将当前目录移到上一级
Step16	调用 creat( ),创建文件 2
Step17	分配缓冲区
Step18	调用 write( ),写文件 2
Step19	关闭文件 2 和释放缓冲
Step20	调用 delete( ),删除文件 0
Step21	调用 creat( ),创建文件 3
Step22	为文件 3 分配缓冲区
Step23	调用 write( ),写文件 3

Step24　　关闭文件 3 并释放缓冲区
Step25　　调用 open( ),打开文件 2
Step26　　为文件 2 分配缓冲
Step27　　写文件 3 后关闭文件 3
Step28　　释放缓冲
Step29　　用户退出(logout)
Step30　　关闭(halt)
End

由上述描述过程可知,该文件系统实际是为用户提供一个解释执行相关命令的环境。主程序中的大部分语句都被用来执行相应的命令。

下面,我们给出每个过程的相关 C 语言程序。读者也可以使用这些子过程,编写出一个用 Shell 控制的文件系统界面。

〈程序〉

1. 编程管理文件 makefile

本文件系统程序用 makefile 编程管理工具进行管理。其内容如下。

```
**************************************************************************** /

/ ********************************************************
    makefile
******************************************************** /
filsys: main.o igetput.o iallfre.o ballfre.o name.o access.o log.o close.o creat.o delete.o dir.o
        open.o rdwt.o format.o install.o halt.o cc -o filsys main.o igetput.o iallfre.o ballfre.o
        name.o access.o log.o close.o creat.o delete.o dir.o open.o rdwt.o format.o install.o halt.o
main.o: main.c filesys.h
        cc -c main.c
igetput.o: igetput.c filesys.h
        cc -c igetput.c
iallfre.o: iallfre.c filesys.h
        cc -c iallfre.c
ballfre.o: ballfre.c filesys.h
        cc -c ballfre.c
name.o: name.c filesys.h
        cc -c name.c
access.o: access.c filesys.h
        cc -c access.c
log.o: log.c filesys.h
        cc -c log.c
close.o: close. c filesys.h
        cc -c close.c
creat.o: creat.c filesys.h
        cc -c creat.c
```

```
delete.o: delete.c filesys.h
        cc -c delete.c
dir.o: dir.c filesys.h
        cc -c dir.c
open.o: open.c filesys.h
        cc -c open.c
rdwt.o: rdwt.c filesys.h
        cc -c rdwt.c
format.o: format.c filesys.h
        cc -c format.c
install.o: install.c filesys.h
        cc -c install.c
halt.o: halt.c
        cc -c halt.c
```

2. 头文件 filesys.h

头文件 filesys.h 用来定义本文件系统中所使用的各种数据结构和常数符号。

```
/ ************************************************************************************
        filesys.h

        定义本文件系统中的数据结构和常数
 ************************************************************************************ /
 #define BLOCKSIZ 512
 #define SYSOPENFILE 40
 #define DIRNUM 128
 #define DIRSIZ 14
 #define PWDSIZ 12
 #define PWDNUM 32
 #define NOFILE 20
 #define NADDR 10
 #define NHINO 128      / * must be power of 2 * /
 #define USERNUM 10
 #define DINODESIZ 32

 / * filsys * /
 #define DINODEBLK 32
 #define FILEBLK 512
 #define NICFREE 50
 #define NICINOD 50
 #define DINODESTART 2 * BLOCKSIZ
 #define DATASTART (2+DINODEBLK) * BLOCKSIZ
```

```
/* di_mode */
#define DIEMPTY         00000

#define DIFILE          01000
#define DIDIR           02000

#define UDIREAD         00001   /* user */
#define UDIWRITE        00002
#define UDIEXICUTE      00004
#define GDIREAD         00010   /* group */
#define GDIWRITE        00020
#define GDIEXICUTE      00040
#define ODIREAD         00100   /* other */
#define ODIWRITE        00200
#define ODIEXICUTE      00400

#define READ            1
#define WRITE           2
#define EXICUTE         3

#define DEFAULTMODE     00777

/* i_flag */
#define IUPDATE 00002

/* s_fmod */
#define SUPDATE 00001

/* f_flag */
#define FREAD           00001
#define FWRITE          00002
#define FAPPEND         00004

/* error */
#define DISKFULL 65535

/* fseek origin */
#define SEEK_SET 0

                        /* 文件系统  数据结构 */
struct inode {
        struct inode *i_forw;
        struct inode *i_back;
```

```
    char i_flag;
    unsigned int i_ino;     /* 磁盘i节点标志 */
    unsigned int i_count;    /* 引用计数 */
    unsigned short di_number;    /* 关联文件数。当为0时,则删除该文件 */
    unsigned short di_mode;    /* 存取权限 */
    unsigned short di_uid;
    unsigned short di_gid;
    unsigned short di_size;    /* 文件大小 */
    unsigned int di_addr [NADDR];    /* 物理块号 */
};

struct dinode {
    unsigned short di_number;    /* 关联文件数 */
    unsigned short di_mode;    /* 存取权限 */
    unsigned short di_uid;
    unsigned short di_gid;
    unsigned long di_size;    /* 文件大小 */
    unsigned int di_addr [NADDR];    /* 物理块号 */
};

sruct direct {
    char d_name [DIRSIZ];
    unsigned int d_ino;
};

struct filsys {
    unsigned short s_isize;    /* i节点块块数 */
    unsigned long s_fsize;    /* 数据块块数 */
    unsigned int s_nfree;    /* 空闲块 */
    unsigned short s_pfree;    /* 空闲块指针 */
    unsigned int s_free [NICFREE];    /* 空闲块堆栈 */

    unsigned int s_ninode;    /* number of free inode in s_inode */
    unsigned short s_pinode;    /* pointer of the sinode */
    unsigned int s_inode [NICINOD];    /* 空闲i节点数组 */
    unsigned int s_rinode;    /* remember inode */

    char s_fmod;    /* 超级块修改标志 */
};

struct pwd {
    unsigned short p_uid;
    unsigned short p_gid;
```

```
        char password [PWDSIZ];
    };

struct dir {
        struct direct direct [DIRNUM];
        int size;      /* 当前目录大小 */
    };

struct hinode {
        struct inode * i_forw;      /* hash 表指针 */
    };
struct file {
        char f_flag;      /* 文件操作标志 */
        unsigned int f_count;      /* 引用计数 */

        struct inode *f_inode;      /* 指向内存 i 节点 */
        unsigned long f_off;      /* read/write character pointer */
    };

struct user {
        unsigned short u_default_mode;
        unsigned short u_uid;
        unsigned short u_gid;
        unsigned short u_ofile [NOFILE];      /* 用户打开文件表 */
        /* system open file pointer number */
    };

    /* 下为全局变量 */
extern struct hinode hinode [NHINO];
extern struct dir dir;      /* 当前目录(在内存中全部读入) */
extern struct file sys_ofile [SYSOPENFILE];
extern struct filsys filsys;      /* 内存中的超级块 */
extern struct pwd pwd [PWDNUM];
extern struct user user [USERNUM];
extern FILE *fd;      /* the file system column of all the system */
extern struct inode * cur_path_inode;
extern int user_id;

/* proptype of the sub roution used in the file system */
extern struct inode *iget( );
extern iput( );
extern unsigned int balloc( );
extern bfree( );
```

```
extern struct inode * ialloc( );
extern ifree( );
extern unsigned int namei( );
extern unsigned short iname( );
extern unsigned int access( );
extern _dir( )
extern mkdir( );
extern chdir( );
extern unsigned short open( );
extern creat( );
extern unsigned int read( );
extern unsigned int write( );
extern int login( );
extern logout( );
extern install( );
extern format( );
extern close( );
extern halt( );
```

3. 主程序 main( )(文件名 main.c)

主程序 main.c 用来测试文件系统的各种设计功能,其主要功能描述如程序设计中的第4部分。

程序:

```
#include <stdio.h>
#include "filesys.h"

struct hinode hinode [NHINO];


struct dir dir;
struct file sys_ofile [SYSOPENFILE];
struct filsys filsys;
struct pwd pwd [PWDNUM];
struct user user [USERNUM];
FILE *fd;
struct inode *cur_path_inode;
int user_id;

main( )
{
   unsigned short ab_fd1, ab_fd2, ab_fd3, ab_fd4;
   unsigned short bhy_fd1;
```

```
char * buf;

printf ("\nDo you want to format the disk \n");
if (getchar( )=='y')
  printf("\nFormat will erase all context on the disk \nAre You Sure!! \n");
if (getchar( )=='y')
  format( );

install( );
_dir( );

login(2118, "abcd");
user_id=0;

mkdir("a2118");
chdir("a2118");
wj_fd1=creat(2118,"ab_file0.c", 01777);
buf = (char * ) malloc (BLOCKSIZ * 6+5);
write(ab_fd1, buf, BLOCKSIZ * 6+5);
close(user_id, ab_fd1);
free(buf);

mkdir("subdir");
chdir("subdir");
wj_fd2=creat(2118,"file1.c", 01777);
buf=(char * ) malloc (BLOCKSIZ * 4+20);
write (ab_fd2, buf, BLOCKSIZ * 4+20);
close(user_id, ab_fd2);
free(buf);
chdir("..");
ab_fd3=creat(2118,"_file2.c", 01777);
buf=(char * ) malloc (BLOCKSIZ * 10+255);
write(ab_fd3, buf, BLOCKSIZ * 3+255);
close(ab_fd3);
free(buf);

delete("ab_file0.c");

ab_fd4=creat(2118, "ab_file3.c", 01777);
buf=(char * ) malloc (BLOCKSIZ * 8+300);
write (ab_fd4, buf, BLOCKSIZ * 8+300);
close(ab_fd4);
free(buf);
```

```
    ab_fd3=open(2118, "ab_file2.c", FAPPEND);
    buf=(char *) malloc (BLOCKSIZ * 3+100);
    write (ab_fd3, buf, BLOCKSIZ * 3+100);
    close (ab_fd3);


    free (buf);

    _dir( );
    chdir ("..");
      logout( );
      halt( );
}
```

4. 初始化磁盘格式程序 format( )(文件名 format.c)

```
#include <stdio.h>
#include "filesys.h"

format( )
{
    struct inode *inode;
    struct direct dir_buf [BLOCKSIZ / (DIRSIZ+2)];
    struct pwd passwd [BLOCKSIZ/(PWDSIZ+4)];
    /*
                                                    {
                                                      {2116, 03, "dddd"},
                                                      {2117, 03, "bbbb"},
                                                      {2118, 04, "abcd"},
                                                      {2119, 04, "cccc"},
                                                      {2220, 05, "eeee"},
                                                    };
    */
    struct filsys;
    unsigned int block_buf [BLOCKSIZ / sizeof (int)];
    char *buf;
    int i, j;
    /* creat the file system file */
    fd = fopen ("filesystem", "r+w+b");
    buf = (char *) malloc ((DINODEBLK+FILEBLK+2) * BLOCKSIZ * sizeof(char));
    if (buf==NULL)
    {
```

```
    printf ("\nfile system file creat failed!!! \n");
    exit (0);
}
fseek (fd, 0, SEEK_SET);


fwrite(buf,1, (DINODEBLK+FILEBLK+2) * BLOCKSIZ * sizeof(char),fd);

/* 0. initialize the passwd */
passwd[0].p_uid=2116; passwd[0].p_gid=03;
strcpy(passwd[0].password, "dddd");
passwd[1].p_uid=2117; passwd[1].p_gid=03;
strcpy(passwd[1].password, "bbbb");
passwd[2].p_uid=2118; passwd[2].p_gid=04;
strcpy(passwd[2].password, "abcd");
passwd[3].p_uid=2119; passwd[3].p_gid=04;
strcpy(passwd[3].password, "cccc");
passwd[4].p_uid=2220; passwd[4].p_gid=05;
strcpy(passwd[4].password, "eeee");

/* 1. creat the main directory and its sub dir etc and the file password */

inode =iget(0);      /* 0 empty dinode id */
inode->di_mode = DIEMPTY;
iput(inode);

inode=iget(1);      /* 1 main dir id */
inode->di_number=1;
inode->di_mode=DEFAULTMODE | DIDIR;
inode->di_size=3 * (DIRSIZ+2);
inode->di_addr[0]=0;      /* block 0# is used by the main directory */
strcpy(dir_buf[0].d_name, "..");
dir_buf[0].d_ino=1;
strcpy(dir_buf[1].d_name, ".");
dir_buf[1].d_ino=1;
strcpy(dir_buf[2].d_name, "etc");
dir_buf[2].d_ino=2;
fseek(fd, DATASTART, SEEK_SET);
fwrite(dir_buf, 1, 3 * (DIRSIZ+2), fd);
iput(inode);

inode=iget(2);      /* 2 etc dir id */
inode->di_number=1;
```

```
    inode->di_mode=DEFAULTMODE | DIDIR;
    inode->di_size=3 * (DIRSIZ+2);
    inode->di_addr[0]=1;      /* block 1# is used by the etc directory */
    strcpy (dir_buf[0].d_name, "..");
    dir_buf[0].d_ino=1;
    strcpy(dir_buf[1].d_name, ".");
    dir_buf[1].d_ino=2;
    strcpy(dir_buf[2].d_name, "password");
    dir_buf[2].d_ino=3;
    fseek(fd, DATASTART+BLOCKSIZ * 1, SEEK_SET);
    fwrite (dir_buf, 1, 3 * (DIRSIZ+2),fd);
    iput(inode);

    inode=iget(3);      /* 3 password id */
    inode->di_number=1;
    inode->di_mode=DEFAULTMODE | DIFILE;
    inode->di_size=BLOCKSIZ;
    inode->di_addr[0]=2;      /* block 2# is used by the password file */
    for (i=5; i<PWDNUM; i++)
    {
       passwd[i].p_uid=0;
       passwd[i].p_gid=0;
       strcpy(passwd[i].password,"                ");
    }
    fseek(fd, DATASTART+2 * BLOCKSIZ, SEEK_SET);


    fwrite(passwd,1, BLOCKSIZ, fd);
    iput(inode);

    /* 2. initialize the superblock */

    filsys.s_isize=DINODEBLK;
    filsys.s_fsize=FILEBLK;

    filsys.s_ninode=DINODEBLK * BLOCKSIZ/DINODESIZ-4;
    filsys.s_nfree=FILEBLK -3;

    for (i=0; i<NICINOD; i++)
    {
       /* begin with 4, 0,1,2,3, is used by main, etc, password */
       filsys.s_inode[i]=4+i;
    }
```

```
    filsys.s_pinode=0;
    filsys.s_rinode=NICINOD+4;

    block_buf[NICFREE-1]=FILEBLK+1;      /* FILEBLK+1 is a flag of end */
    for (i=0; i<NICFREE -1; i++)
      block_buf [NICFREE-2-i]=FILEBLK-i;
    fseek (fd, DATASTART+BLOCKSIZ * (FILEBLK-NICFREE-1), SEEK_SET);
    fwrite (block_buf, 1, BLOCKSIZ, fd);

    for (i=FILEBLK-NICFREE-1; i>2; i-=NICFREE)
    {
       for (j=0; j<NICFREE; j++)
       {
           block_buf[j]=i-j;
       }
       fseek(fd,DATASTART+BLOCKSIZ * (i-1), SEEK_SET);
       fwrite(block_buf, 1, BLOCKSIZ, fd);
    }

    j=1;
    for (i=i; i>2; i--)
    {
        filsys.s_free [NICFREE+i-j]=i;
    }

    filsys.s_pfree=NICFREE - j;
    filsys.s_pinode=0;

    fseek(fd, BLOCKSIZ, SEEK_SET);
    fwrite(&filsys, 1, sizeof(struct filsys), fd);
}
```

5. 进入文件系统程序 install( )(文件名 install.c)

```
#include <stdio.h>
#include <string.h>
#include "filesys.h"

install( )
{
   int i,j;

   /* 0. open the file column */
```

```
fd=fopen("filesystem", "w+r+b");
if (fd=NULL)
{
    printf("\nfilesys can not be loaded\n");
    exit(0);
}

/* 1. read the filsys from the superblock */
fseek (fd, BLOCKSIZ, SEEK_SET);
fwrite(&filsys, 1, sizeof(struct filsys), fd);

/* 2. initialize the inode hash chain */
for (i=0; i<NHINO; i++)
{
    hinode[i].i_forw=NULL;
}

/* 3. initjalize the sys_ofile */
for (i=0; i<SYSOPENFILE; i++)
{
    sys_ofile[i].f_count=0;
    sys_ofile[i].f_inode=NULL;
}

/* 4. initialize the user */
for (i=0; i<USERNUM; i++)
{
    user[i].u_uid=0;
    user[i].u_gid=0;
    for (j=0; j<NOFILE; j++)
    {
        user[i].u_ofile[j]=SYSOPENFILE+1;
    }
}

/* 5. read the main directory to initialize the dir */
cur_path_inode=iget(1);
dir.size=cur_path_inode->di_size/(DIRSIZ+2);
for(i=0; i<DIRNUM; i++)
{
    strcpy(dir.direct[i].d_name, "                ");
    dir.direct[i].d_ino=0;
}
```

```
    for (i=0; i<dir.size/(BLOCKSIZ/(DIRSIZ+2)); i++)
    {
        fseek(fd,DATASTART+BLOCKSIZ*cur_path_inode->di_addr[i], SEEK_SET);
        fread(&dir.direct[(BLOCKSIZ/(DIRSIZ+2))*i], 1, BLOCKSIZ, fd);
    }
    fseek(fd, DATASTART+BLOCKSIZ*cur_path_inode->di_addr[i], SEEK_SET);
    fread(&dir.direct[(BLOCKSIZ)/(DIRSIZ+2)*i], 1,
          cur_path_inode->di_size % BLOCKSIZ, fd);
}
```

6. 退出程序 halt( )(文件名 halt.c)

```
#include <stdio.h>
#include "filesys.h"

halt( )
{
    struct inode *inode;
    int i, j;

    /* 1. write back the current dir */
    chdir ("..");
    iput(cur_path_inode);

    /* 2. free the u_ofile and sys_ofile and inode */
    for (i=0; i<USERNUM; i++)
    {
        if (user[i].u_uid !=0)
        {
            for (j=0; j<NOFILE; j++)
            {
                if (user[i].u_ofile[j] !=SYSOPENFILE+1)
                {
                    close (user[i].u_ofile[j]);
                    user[i].u_ofile[j]=SYSOPENFILE+1;
                }
            }
        }
    }

    /* 3. write back the filesys to the disk */
    fseek (fd, BLOCKSIZ, SEEK_SET);
    fwrite (&filsys, 1, sizeof(struct filsys),fd);
```

```
    /* 4. close the file system column */
    fclose (fd);

    /* 5. say GOOD BYE to all the user */
    printf ("\nGood Bye. See You Next Time. Please turn off the switch\n");
    exit (0);
```

7. 获取释放 i 节点内容程序 iget( )/iput( )(文件名 igetputc)

```
#include <stdio.h>
#include "filesys.h"

struct inode * iget (dinodeid)      /* iget( ) */
unsigned int dinodeid;
{
    int existed=0, inodeid;
    long addr;
    struct inode *temp, *newinode;

    inodeid=dinodeid % NHINO;
    if (hinode [inodeid]. i_forw == NULL) existed = 0;
    else
    {
        temp=hinode [inodeid].i_forw;
        while (temp)
        {
            if (temp-> i_ino == inodeid)
            /* existed */
            {
                existed=1;
                temp->i_count ++;
                return temp;
            }
            /* not existed */
            else
                temp =temp->i_forw;
        };
    }
        /* not existed */
        /* 1. calculate the addr of the dinode in the file sys column */
        addr = DINODESTART + dinodeid * DINODESIZ;
        /* 2. malloc the new inode */
        newinode =(struct inode * ) malloc (sizeof (struct inode));
```

```
        /* 3. read the dinode to the inode */
        fseek (fd, addr, SEEK_SET);
        fread(&(newinode->di_number), DINODESIZ, 1, fd);
        /* 4. put it into hinode [inodeid] queue */
        newinode->i_forw=hinode [inodeid].i_forw;
        newinode->i_back=newinode;
        newinode->i_forw->i_back=newinode;
        hinode [inodeid].i_forw=newinode;
        /* 5. initialize the inode */
        newinode->i_count=1;
        newinode->i_flag=0;      /* flag for not update */
        newinode->i_ino=dinodeid;

        return newinode;
}

iput(pinode)                    /* iput ( ) */
struct inode *pinode;
{
    long addr;
    unsigned int block_num;
    int i;

    if (pinode->i_count>1)
    {
        pinode->i_count --;
        return;
    }
    else
    {

        if (pinode->di_number !=0)
        {
            /* write back the inode */
            addr =DINODESTART + pinode->i_ino * DINODESIZ;
            fseek(fd, addr, SEEK_SET);
            fwrite(&pinode->di_number, DINODESIZ,1,fd);
        }
        else
        {
            /* rm the inode & the block of the file in the disk */
            block_num=pinode->di_size/BLOCKSIZ;
            for (i=0; i<block_num; i++)
```

```
            {
                balloc(pinode->di_addr[i]);
            }
            ifree(pinode->i_ino);
        };

        /* free the inode in the memory */
        if (pinode->i_forw==NULL)
            pinode->i_back->i_forw=NULL;
        else
        {
            pinode->i_forw->i_back=pinode->i_back;
            pinode->i_back->i_forw=pinode->i_forw;
        };
        free (pinode);
    };
}
```

8. i 节点分配和释放函数 ialloc( )和 ifree( )(文件名 iallfre.c)

```
#include <stdio.h>
#include "filesys.h"

static struct dinode block_buf [BLOCKSIZ/DINODESIZ];

struct inode * ialloc ( )          /* ialloc */
{
    struct inode * temp_inode;
    unsigned int cur_di;
    int i, count, block_end_flag;

    if (filsys.s_pinode == NICINOD)      /* s_inode empty */
    {
        i=0;
        count = 0;
        block_end_flag=1;
        filsys.s_pinode=NICINOD-1;
        cur_di=filsys.s_rinode;
        while ((count <NICINOD) || (count <= filsys.s_ninode))
        {
            if (block_end_flag)

            {
```

```
                fseek (fd,DINODESTART+cur_di * DINODESIZ);
                fread (block_buf, 1, BLOCKSIZ, fd);
                block_end_flag=0;
                i=0;
            }
            while (block_buf[i].di_mode == DIEMPTY)
            {
                cur_di ++;
                i++;
            }
            if (i==NICINOD)
                block_end_flag=1;
            else
            {
                filsys.s_inode[filsys.s_pinode--]=cur_di;
                count ++;
            }
        }
        filsys.s_rinode=cur_di;
    }
    temp_inode=iget(filsys.s_inode [filsys.s_pinode]);
    fseek (fd, DINODESTART+filsys.s_inode [filsys.s_pinode] * DINODESIZ, SEEK_SET);
    fwrite (&temp_inode->di_number, 1, sizeof (struct dinode), fd);
    filsys.s_pinode ++;
    filsys.s_ninode --;
    filsys.s_fmod=SUPDATE;
    return temp_inode;
}

ifree(dinodeid)                    /* ifree */
unsigned dinodeid;
{
    filsys.s_ninode ++;
    if (filsys.s_pinode != NICINOD)     /* not full */
    {
        filsys.s_inode[filsys.s_pinode]=dinodeid;
        filsys.s_pinode ++;
    }
    else /* full */
    {
        if (dinodeid <filsys.s_rinode)
        {
            filsys.s_inode [NICINOD]=dinodeid;
```

```
            filsys.s_rinode=dinodeid;
        }
    }
}
```

9. 磁盘块分配与释放函数 balloc( )与 bfree( )(文件名 ballfre.c)

```
#include <stdio.h>
#include "filesys.h"

static unsigned int block_buf[BLOCKSIZ];

unsigned int balloc( )
{
    unsigned int free_block, free_block_num;
    int i;

    if (filsys.s_nfree==0)
    {
        printf ("\nDisk Full!!! \n");
        return DISKFULL;
    };

    free_block=filsys.s_free[filsys.s_pfree];
    if (filsys.s_pfree==NICFREE-1)
    {
        fread(block_buf, 1, BLOCKSIZ, fd);
        free_block_num=block_buf [NICFREE];     /* the total block num in the group */
        for (i=0; i<free_block_num; i++)
        {
            filsys.s_free [NICFREE-1-i]=block_buf[i];
        }
        filsys.s_pfree=NICFREE-free_block_num;
    }
    else filsys.s_pfree ++;

    filsys.s_nfree --;
    filsys.s_fmod=SUPDATE;

    return free_block;
}

bfree (block_num)
```

```
unsigned int block_num;
{
    int i;

    if (filsys.s_pfree==0)      /* s_free full */
    {
        block_buf[NICFREE]=NICFREE;
        for (i=0; i<NICFREE; i++)
        {
            block_buf[i]=filsys.s_free[NICFREE-1-i];
        }
        filsys.s_pfree=NICFREE-1;
    }

    fwrite(block_buf, 1, BLOCKSIZ, fd);
    filsys.s_nfree ++;
    filsys.s_fmod=SUPDATE;
}
```

10. 搜索函数 namei( )和 iname( )(文件名 name.c)

```
#include <string.h>
#include <stdio.h>
#include "filesys.h"

unsigned int namei (name)                    /* namei */
char *name;
{
    int, i, notfound=1;

    for (i=0; ((i<dir.size)&&(notfound));i++)
        if ((! strcmp(dir.direct[i].d_name, name)) && (dir.direct[i].d_ino != 0))
            return i;      /* find */
    /* not find */
    return NULL;
};

unsigned short iname (name)                  /* iname */
char *name;
{
    int i, notfound=1;

    for (i=0; ((i<DIRNUM)&&(notfound)); i++)
```

```
        if (dir.direct[i].d_ino==0)
        {
            notfound=0;
            break;
        }
    if (notfound)
    {
        printf("\nThe current directory is full!!! \n");
        return 0;
    }
    else
    {
        strcpy(name, dir.direct[i].d_name);
        return i;
    }
}
```

11. 访问控制函数 access( )(文件名 access.c)

```
#include <stdio.h>
#include "filesys.h"

unsigned int access (user_id, inode, mode)
unsigned int user_id;
struct inode *inode;
unsigned short mode;
{

    switch(mode)
    {
        case READ:
                    if (inode->di_mode & ODIREAD) return 1;
                    if ((inode->di_mode & GDIREAD) &&

                        (user[user_id].u_gid==inode->di_gid)) return 1;
                    if ((inode->di_mode & UDIREAD) &&
                        (user[user_id].u_uid==inode->di_uid)) return 1;
                    return 0;
        case WRITE:
                    if (inode->di_mode & ODIWRITE) return 1;
                    if ((inode->di_mode & GDIWRITE) &&
                        (user[user_id].u_gid==inode->di_gid)) return 1;
                    if ((inode->di_mode & UDIWRITE) &&
```

```
                    (user[user_id].u_uid==inode->di_uid)) return 1;
                return 0;
        case EXICUTE:
                if (inode->di_mode & ODIEXICUTE) return 1;
                if ((inode->di_mode & GDIEXICUTE) &&
                    (user[user_id].u_gid==inode->di_gid)) return 1;
                if ((inode->di_mode & UDIEXICUTE) &&
                    (user[user_id].u_uid==inode->di_uid)) return 1;
                return 0;
        defualt:
                return 0;
    }
}
```

12. 显示列表函数 dir( )和目录创建函数 mkdir( )等(文件名 dir.c)

```
#include <stdio.h>
#include <string.h>
#include "filesys.h"

_dir( )                 /* _dir */
{
    unsigned int di_mode;
    int i, one;
    struct inode *temp_inode;

    printf("\n CURRENT DIRECTORY :\n");
    for (i=0; i<dir.size; i++)
    {
        if (dir.direct[i].d_ino != DIEMPTY)
        {
            printf("%DIRSIZs", dir.direct[i].d_name);
            temp_inode=iget(dir.direct[i].d_ino);
            di_mode=temp_inode->di_mode;
            for (i=0; i<9; i++)
            {
                one =di_mode % 2;
                di_mode=di_mode /2;

                if (one) printf ("x");
                else printf ("-");
            }
            if (temp_inode->di_mode && DIFILE==1)
```

```
            {
                printf ("%ld\n", temp_inode->di_size);
                printf("block chain:");
                for (i=0; i<temp_inode->di_size/BLOCKSIZ+1; i++)
                    printf("%4d", temp_inode->di_addr[i]);
                printf("\n");
            }
            else printf ("<dir>\n");
            iput(temp_inode);
        }
    }
}

mkdir (dirname)                         /* mkdir */
char *dirname;
{
    int dirid, dirpos;
    struct inode *inode;
    struct direct buf[BLOCKSIZ/(DIRSIZ+2)];
    unsigned int block;

    dirid=namei(dirname);
    if (dirid != NULL)
    {
        inode=iget(dirid);
        if(inode->di_mode & DIDIR)
            printf("\n%s directory already existed!! 1\n");
        else
            printf("\n%s is a file name, &can't creat a dir the same name", dirname)
        iput(inode);
        return;
    }

    dirpos=iname(dirname);
    inode=ialloc( );
    inode->i_ino=dirid;
    dir.direct[dirpos].d_ino=inode->i_ino;
    dir.size ++;

    /* fill the new dir buf */
    strcpy (buf[0].d_name, ".");
    buf[0].d_ino=dirid;
    strcpy(buf[1].d_name, "..");
```

```
    buf[1].d_ino =cur_path_inode->i_ino;

    block=balloc( );
    fseek(fd, DATASTART+block * BLOCKSIZ, SEEK_SET);
    fwrite(buf,1,BLOCKSIZ,fd);

    inode->di_size=2 * (DIRSIZ+2);
    inode->di_number=1;
    inode->di_mode=user[user_id].u_default_mode;
    inode->di_uid=user[user_id].u_uid;
    inode->di_gid=user[user_id].u_gid;
    inode->di_addr[0]=block;

    iput(inode);

    return;
}

chdir (dirname)                          /* chdir */
char *dirname;
{
    unsigned int dirid;
    struct inode *inode;
    unsigned short block;
    int i,j,low=0, high=0;

    dirid=namei(dirname);
    if (dirid==NULL)
    {
        printf("\n%s does not existed\n", dirname);
        return;
    }
    inode=iget (dirid);
    if (! access (user_id, inode, user[user_id].u_default_mode))
    {
    printf("\nhas not access to the directory %s", dirname);
    iput(inode);
    return;
    }

    /* pack the current directory */
    for (i=0; i<dir.size; i++)
    {
```

```
        for (;j<DIRNUM;j++)
            if (dir.direct[j].d_ino==0) break;
        memcpy(&dir.direct[i], &dir.direct[j], DIRSIZ+2);
        dir.direct[j].d_ino=0;
    }
    /* write back the current directory */
    for (i=0; i<cur_path_inode->di_size/BLOCKSIZ+1, i++)
    {
        bfree (cur_path_inode->di_addr[i]);
    }
    for (i=0; i<dir.size; i+=BLOCKSIZ/(DIRSIZ+2))
    {
        block=balloc( );
        cur_path_inode->di_addr[i]=block;
        fseek(fd, DATASTART+block*BLOCKSIZ, SEEK_SET);
        fwrite(&dir.direct[i], 1, BLOCKSIZ, fd);
    }
    cur_path_inode->di_size=dir.size*(DIRSIZ+2);
    iput(cur_path_inode);

    cur_path_inode=inode;
    /* read the change dir from disk */
    j=0;
    for (i=0; i<inode->di_size/BLOCKSIZ+1; i++)
    {
        fseek(fd,DATASTART+inode->di_addr[i]*BLOCKSIZ, SEEK_SET);
        fread(&dir.direct[j], 1, BLOCKSIZ, fd);
        j+=BLOCKSIZ/(DIRSIZ+2);
    };

    return;

}
```

13. 文件创建函数 creat( )(文件名 creat.c)

```
#include <stdio.h>
#include "filesys.h"

creat (user_id, filename, mode)
unsigned int user_id;
char *filename;
unsigned short mode;
```

```
{
    unsigned int di_ith, di_ino;
    struct inode *inode;
    int i,j;

    di_ino=namei(filename);
    if (di_ino != NULL)        /* already existed */
    {
        inode=iget(di_ino);
        if (access (user_id, inode, mode)==0)
        {
            iput (inode);
            printf ("\creat access not allowed \n");
            return;
        }
        /* free all the block of the old file */
        for (i=0; i<inode->di_size / BLOCKSIZ+1; i++)
        {
            bfree (inode->di_addr[i]);
        }
        /* to do: add code here to update the pointer of the sys_file */
        for (i=0; i<SYSOPENFILE; i++)
            if (sys_ofile [i].f_inode == inode)
            {
                sys_ofile[i].f_off=0;
            }
        for (i=0; i<NOFILE; i++)
            if (user[user_id].u_ofile[i]==SYSOPENFILE+1)
            {
                user [user_id].u_uid=inode->di_uid;
                user [user_id].u_gid=inode->di_gid;
                for (j=0; j<SYSOPENFILE; j++)
                    if (sys_ofile [j].f_count=0)
                    {
                        user [user_id].u_ofile[i]=j;
                        sys_ofile[j].f_flag=mode;
                    }
                return i;
            }
    }
    else /* not existed before */
    {
        inode = ialloc( );
```

```
        di_ith=iname (filename);

        dir.size ++;

        dir.direct[di_ith].d_ino=inode->i_ino;
        inode->di_mode=user[user_id].u_default_mode;
        inode->di_uid=user[user_id].u_uid;
        inode->di_gid=user[user_id].u_gid;
        inode->di_size=0;
        inode->di_number=0;

        for (i=0; i<SYSOPENFILE; i++)
            if (sys_ofile[i].f_count==0)
            {
                break;
            }

        for (j=0; j<NOFILE; i++)
            if (user[user_id].u_ofile[j]==SYSOPENFILE +1)
            {
                break;
            }

        user[user_id].u_ofile[j]=i;
        sys_ofile[i].f_flag=mode;
        sys_ofile[i].f_count=0;
        sys_ofile[i].f_off=0;
        sys_ofile[i].f_inode=inode;

        return j;
    }

}
```

14. 打开文件函数 open( )(文件名 open.c)

```
#include <stdio.h>
#include "filesys.h"

unsigned short open(user_id, filename, openmode)
int user_id;
char *filename;
unsigned short openmode;
```

```
{
    unsigned int dinodeid;
    struct inode *inode;
    int i,j;

    dinodeid=namei(filename);
    if (dinodeid != NULL)      /* no such file */
    {
        printf ("\nfile does not existed!!! \n");
        return NULL;
    }
    inode=iget(dinodeid);
    if (! access(user_id, inode, openmode))      /* access denied */
    {
        printf("\nfile open has not access!!!");
        iput(inode);
        return NULL;
    }

    /* alloc the sys_ofile item */
    for (i=1; i<SYSOPENFILE; i++)
        if (sys_ofile[i].f_count==0) break;
    if (i==SYSOPENFILE)
    {
        printf("\nsystem open file too much\n");
        iput (inode);
        return NULL;
    }
    sys_ofile[i].f_inode=inode;
    sys_ofile[i].f_flag=openmode;
    sys_ofile[i].f_count=1;
    if (openmode & FAPPEND)
        sys_ofile[i].f_off=inode->di_size;
    else
        sys_ofile[i].f_off=0;

    /* alloc the user open file item */
    for (j=0; j<NOFILE; j++)
        if (user[user_id].u_ofile[j]==0) break;
    if (j==NOFILE)
    {
        printf("\nuser open file too much!!! \n");
        sys_ofile[i].f_count=0;
```

```
        iput(inode);
        return NULL;
    }
    user[user_id].u_ofile[j]=1;

    /* if APPEND, free the block of the file before */
    if (openmode & FAPPEND)
    {
        for (i=0; i<inode->di_size /BLOCKSIZ +1; i++)
            bfree (inode->di_addr[i]);
        inode->di_size=0;
    }
    return j;

}
```

15. 关闭文件系统函数 close( )(文件名 close.c)

```
#include <stdio.h>
#include "filesys.h"

close (user_id, cfd)                /* close */
unsigned int user_id;
unsigned short cfd;
{
    struct inode *inode;

    inode=sys_ofile [user[user_id].u_ofile[cfd]].f_inode;
    iput (inode);
    sys_ofile [user[user_id].u_ofile[cfd]].f_count --;
    user [user_id].u_ofile [cfd]=SYSOPENFILE + 1;
}
```

16. 删除文件函数 delete( )(文件名 delete.c)

```
#include <stdio.h>
#include "filesys.h"

delete (filename)
char *filename;
{
    unsigned int dinodeid;
    struct inode *inode;
```

```
    dinodeid=namei(filename);
    if (dinodeid != NULL)
    inode = iget(dinodeid);
    inode->di_number --;
    iput(inode);
}
```

17. 读写文件函数 read( )与 write( )(文件名 rdwt.c)

```
#include <stdio.h>
#include "filesys.h"

unsigned int read (fd, buf, size)
int fd;
char *buf;
unsigned int size;
{
    unsigned long off;
    int block, block_off, i, j;
    struct inode *inode;
    char *temp_buf;

    inode=sys_ofile[user[user_id].u_ofile[fd]].f_inode;
    if (! (sys_ofile[user[user_id].u_ofile[fd]].f_flag & FREAD))
    {
        printf ("\nthe file is not opened for read\n");
        return 0;
    }

    temp_buf=buf;

    off = sys_ofile[user[user_id].u_ofile[fd]].f_off;
    if ( (off+size) > inode->di_size) size=inode->di_size-off;

    block_off=off % BLOCKSIZ;
    block=off/BLOCKSIZ;

    if (block_off+size<BLOCKSIZ)

    {
        fseek(fd, DATASTART + inode->di_addr[block] * BLOCKSIZ + block_off, SEEK_
        SET);
        fread(buf, 1, size, fd);
```

```
        return size;
    }

    fseek(fd, DATASTART+inode->di_addr[block] * BLOCKSIZ+block_off, SEEK_SET);
    fread(temp_buf, 1, BLOCKSIZ-block_off, fd);

    temp_buf += BLOCKSIZ-block_off;
    j=(inode->di_size-off-block_off) / BLOCKSIZ;
    for (i=0; i<(size-block_off) /BLOCKSIZ; i++)
   {
        fseek(fd, DATASTART+inode->di_addr[j+i] * BLOCKSIZ, SEEK_SET);
        fread(temp_buf, 1, BLOCKSIZ, fd);
        temp_buf += BLOCKSIZ;
    }

    block_off = (size-block_off) % BLOCKSIZ;
    block=inode->di_addr[off+size/BLOCKSIZ+1];
    fseek(fd, DATASTART+block * BLOCKSIZ, SEEK_SET);
    fread(temp_buf, 1, block_off, fd);

    sys_ofile[user[user_id].u_ofile[fd]].f_off += size;

    return size;
}

unsigned int write (fd, buf, size)                    /* write */
int fd;
char *buf;
unsigned int size;
{
    unsigned long off;
    int block, block_off, i, j;
    struct inode *inode;
    char *temp_buf;

    inode=sys_ofile[user[user_id].u_ofile[fd]].f_inode;

    if (! (sys_ofile[user[user_id].u_ofile[fd]].f_flag & FWRITE))
    {
        printf("\nthe file is not opened for write\n");
        return 0;
    }
```

```
    temp_buf=buf;

    off=sys_ofile[user[user_id].u_ofile[fd]].f_off;
    block_off=off % BLOCKSIZ;
    block=off/BLOCKSIZ;

    if (block_off+size<BLOCKSIZ)
    {
        fseek(fd, DATASTART + inode->di_addr[block] * BLOCKSIZ + block_off, SEEK_
        SET);
        fwrite(buf, 1, size, fd);
        return size;
    }

    fseek(fd, DATASTART+inode->di_addr[block] * BLOCKSIZ+block_off, SEEK_SET);
    fwrite(temp_buf, 1, BLOCKSIZ-block_off, fd);

    temp_buf += BLOCKSIZ-block_off;
    for (i=0; i<(size-block_off)/BLOCKSIZ; i++)
    {
        inode->di_addr[block+1+i]=balloc( );
        fseek(fd,DATASTART+inode->di_addr[block+1+i] * BLOCKSIZ, SEEK_SET);
        fwrite(temp_buf, 1, BLOCKSIZ, fd);
        temp_buf += BLOCKSIZ;
    }

    block_off = (size-block_off) % BLOCKSIZ;
    block=inode->di_addr[off+size/BLOCKSIZ+1]=balloc( );
    fseek(fd,DATASTART+block * BLOCKSIZ,SEEK_SET);
    fwrite(temp_buf,1,block_off, fd);

    sys_ofile[user[user_id].u_ofile[fd]].f_off += size;

    return size;
}
```

18. 注册和退出函数 login( )和 logout( )(文件名 log.c)

```
#include <stdio.h>
#include "filesys.h"

int login (uid, passwd)
unsigned short uid;
```

```
char *passwd;
{
    int i,j;

    for (i=0; i<PWDNUM; i++)
    {
        if ((uid==pwd[i].p_uid)&&(strcmp(passwd,pwd[i].password)))
        {
            for (j=0; j<USERNUM; i++)
                if (user[j].u_uid==0) break;
            if (j==USERNUM)
            {
                printf("\ntoo much user in the system, waited to login\n");
                return 0;
            }
            else
            {
                user[j].u_uid=uid;
                user[j].u_gid=pwd[i].p_gid;
                user[j].u_default.mode=DEFAULTMODE;

            }
            break;
        }
    }
    if (i==PWDNUM)
    {
        printf("\incorrect password\n");
        return 0;
    }
    else
        return 1;

}

int logout (uid)                    /* logout */
unsigned short uid;
{
    int i,j,sys_no;
    struct inode *inode;

    for (i=0; i<USERNUM; i++)
        if (uid == user[i].u_uid) break;
```

```
    if (i==USERNUM)
    {
        printf("\nno such a file\n");
        return NULL;
    }

    for (j=0; j<NOFILE; j++)
    {
        if (user[i].u_ofile[j] != SYSOPENFILE+1)
        {
            /* iput the inode free the sys_ofile and clear the user_ofile */
            sys_no=user[i].u_ofile[j];
            inode=sys_ofile[sys_no].f_inode;
            iput(inode);
            sys_ofile[sys_no].f_count --;
            user[i].u_ofile[j]=SYSOPENFILE+1;
        }
    }

    return 1;

}
```

[结果]

对上述 makefile 文件进行编译后可得执行文件“filsys”。在 Linux 或 UNIX System V 以上版本环境下，运行 filsys，可对上述文件系统程序进行测试。其结果如下：

```
/ ********************************************************************************
      result:
            the output of the filesystem run by the test program
********************************************************************************/

$ filsys <CR>

$ format
$ Do you want to format the disk?     y
$ Format will erase all context on the disk. Are You Sure!!     y

$ install

$ dir
  CURRENT DIRECTORY:
    .                    d xxx xxx xxx <dir> block chain:1
    ..                   d xxx xxx xxx <dir> block chain:2
```

```
$ login
  please input your uid:2118
  password:
  ok! 2118's user id is: 0

$ mkdir a2118
$ chdir a2118

$ creat(2118, "file0.c", 01700)
  the file file0.c fd:0
$ write (0, buf, 3077)
$ close (0,0)

$ mkdir subdir
$ chdir subdir

$ creat(2118, "file1.c", 01700)
  the file file1.c fd:0
$ write (0, buf, 2068)

$ chdir ..

$ creat(2118, "file2.c", 01700)
  the file file2.c fd:1
$ write(1,buf,1791)

$ dir
  CURRENT DIRECTORY:
    .          d xxx xxx xxx <dir>block chain:1
    ..         d xxx xxx xxx <dir>block chain:2
    file0.c    f xxx --- --- 3077     block chain:4 5 6 7 8 9 10
    subdir     d xxx xxx xxx <dir>bloke chain:11
    file2.c    f xxx --- --- 1791     block chain: 17 18 19 20

$ delete file0.c

$ creat(2118, "file3.c", 01700)
  the file file3.c fd:2
$ write (2, buf, 4396)
$ close (0,2)
```

```
$ dir
  CURRENT DIRECTORY:
    .              d xxx xxx xxx <dir>block chain:1
    ..             d xxx xxx xxx <dir>block chain:2
    subdir         d xxx xxx xxx <dir>block chain:11
    file2.c        f xxx --- --- 1791      block chain: 17 18 19 20
    file3.c        f xxx --- --- 4396      block chain:4 5 6 7 8 9 10 21 22 23


$ close(0,0)
$ close(0,1)

$ open(2118, "file2.c", 03)
  the file file2.c fd:0
$ write(0, buf, 1636)
$ close(0,0 )

$ dir
  CURRENT DIRECORY:
    .              d xxx xxx xxx <dir>block chain:1
    ..             d xxx xxx xxx <dir>block chain:2
    subdir         d xxx xxx xxx <dir>block chain:11
    file2.c        f xxx --- --- 3427      block chain: 17 18 19 20 24 25 26
    file3.c        f xxx --- --- 4396      block chain, 4 5 6 7 8 9 10 21 22 23

$ chdir ..

$ logout (2118)
  no user in the file system.

$ halt
  the file system now is halt! good bye.
```

注意，本文件系统程序未使用命令交互解释工具 Shell，读者可从文程序中观察到这一点。

清华大学计算机系列教材

1. 计算机操作系统教程（第2版）	张尧学 等
2. 计算机操作系统教程（第2版）习题解答与实验指导	张尧学
3. PASCAL程序设计（第二版）	郑启华
4. PASCAL程序设计习题与选解（新编）	郑启华
5. IBM PC汇编语言程序设计（第2版）	沈美明 等
6. IBM PC汇编语言程序设计例题习题集	温冬婵 等
7. IBM PC汇编语言程序设计实验教程	沈美明
8. 计算机图形学（新版）	孙家广 等
9. 微型计算机技术及应用——从16位到32位	戴梅萼
10. 微型计算机技术及应用——习题与实验题集	戴梅萼
11. 微型计算机技术及应用——微型机软件硬件开发指南	戴梅萼
12. 计算机组成与结构（第3版）	王爱英 等
13. 计算机组成原理实验指导书与习题集	王 诚 等
14. 计算机系统结构（第二版）	张晨曦 等
15. 数据结构（第二版）	严蔚敏 等
16. 数据结构题集	严蔚敏 等
17. 图论与代数结构	戴一奇 等
18. 数字逻辑与数字集成电路	王尔乾 等
19. 数字系统设计自动化	蔡懿慈 等
20. 计算机图形学基础	唐泽圣 等
21. 编译原理	吕映芝 等
22. 数据结构（用面向对象方法与C++描述）	殷人昆 等
23. 计算机网络与 Internet 教程	张尧学 等
24. 多媒体技术基础	林福宗
25. 多媒体技术基础实验指南	谢青珊 等
26. 数理逻辑与集合论（第2版）	石纯一 等
27. 数理逻辑与集合论（第2版）精要与题解	王 宏 等

ISBN 7-302-04004-4

9 787302 040040

定价：11.00 元

责任编辑：马殿禄 / 封面设计：傅瑞学