计算机操作系统

实验指导

上海大学计算机工程与科学学院

操作系统课程组

二〇一四年十一月

计算机操作系统（二）实验目录

实验（一） 操作系统的进程调度

一、目的与要求

1、目的

进程是操作系统最重要的概念之一，进程调度又是操作系统核心的主要内容。本实习要求学生独立地用高级语言编写和调试一个简单的进程调度程序。调度算法可任意选择或自行设计。例如，简单轮转法和优先数法等。本实习可加深对于进程调度和各种调度算法的理解。

2、 要求

（1） 设计一个有 n 个进程工行的进程调度程序。每个进程由一个进程控制块（PCB）表示。进程控制块通常应包含下述信息：进程名、进程优先数、进程需要运行的时间、占用 CPU 的时间以及进程的状态等，且可按调度算法的不同而增删。

（2） 调度程序应包含 2～3 种不同的调度算法，运行时可任意选一种，以利于各种算法的分析比较。

（3） 系统应能显示或打印各进程状态和参数的变化情况，便于观察诸进程的调度过程

二、示例

1、题目

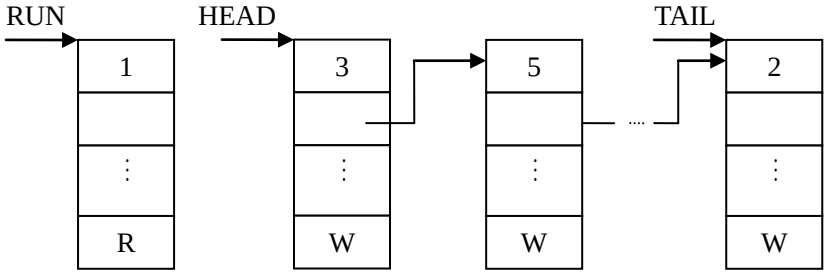
本程序可选用优先数法或简单轮转法对五个进程进行调度。每个进程处于运行 R(run)、就绪 W(wait)和完成 F(finish)三种状态之一，并假设起始状态都是就绪状态 W。为了便于处理，程序进程的运行时间以时间片为单位计算。各进程的优先数或轮转时间片数、以及进程需要运行的时间片数，均由伪随机数发生器产生。

进程控制块结构如下：

PCB

进程标识数
链指针
优先数/轮转时间片数
占用 CPU 时间片数
进程所需时间片数
进程状态

进程控制块链结构如下：



其中：RUN—当前运行进程指针；
HEAD—进程就绪链链首指针；
TAID—进程就绪链链尾指针。

2、 算法与框图

(1) 优先数法。进程就绪链按优先数大小从高到低排列，链首进程首先投入运行。每过一个时间片，运行进程所需运行的时间片数减 1，说明它已运行了一个时间片，优先数也减 3，理由是该进程如果在一个时间片中完成不了，优先级应该降低一级。接着比较现行进程和就绪链链首进程的优先数，如果仍是现行进程高或者相同，就让现行进程继续进行，否则，调度就绪链链首进程投入运行。原运行进程再按其优先数大小插入就绪链，且改变它们对应的进程状态，直至所有进程都运行完各自的时间片数。

(2) 简单轮转法。进程就绪链按各进程进入的先后次序排列，进程每次占用处理机的轮转时间按其重要程度登入进程控

制块中的轮转时间片数记录项（相当于优先数法的优先数记录项位置）。每过一个时间片，运行进程占用处理机的时间片数加 1，然后比较占用处理机的时间片数是否与该进程的轮转时间片数相等，若相等说明已到达轮转时间，应将现运行进程排到就绪链末尾，调度链首进程占用处理机，且改变它们的进程状态，直至所有进程完成各自的时间片。

(3) 程序框图如下**图 1** 所示。

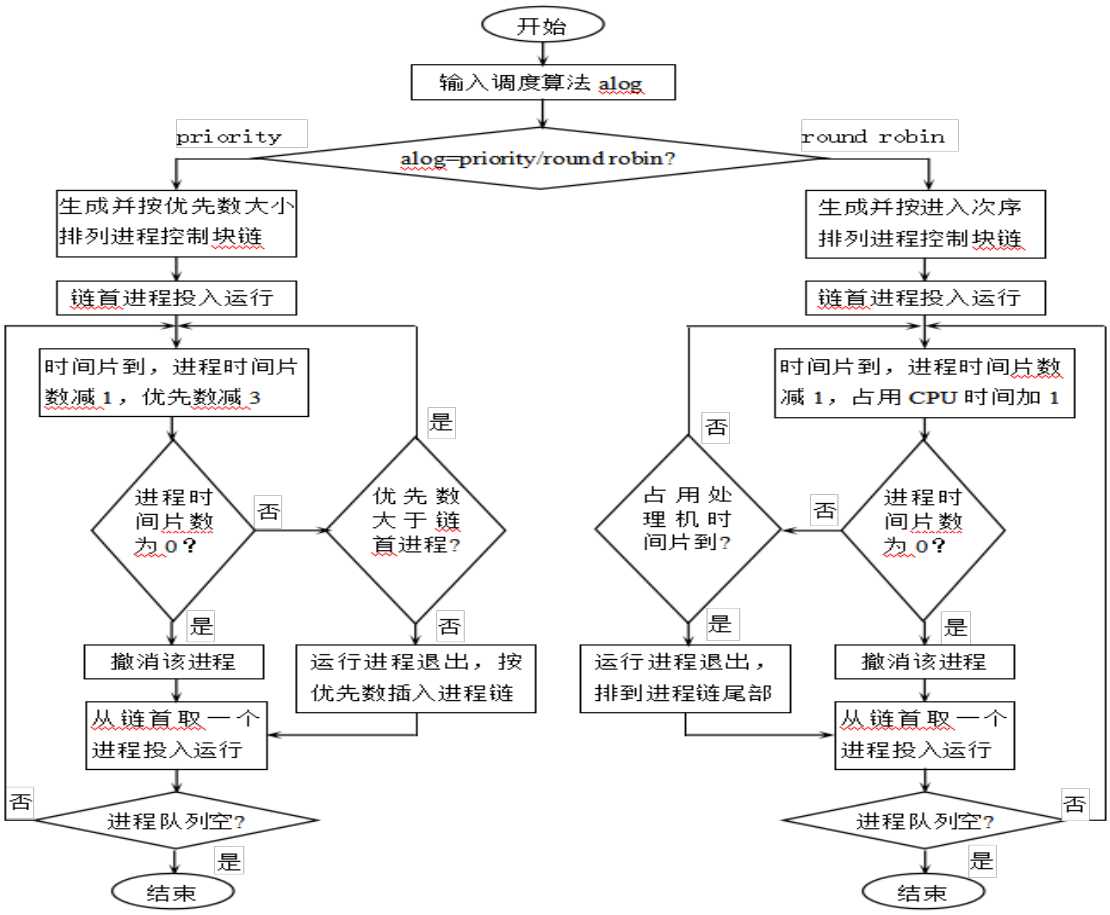


图 1　进程调度框图

3、 **程序运行结果格式**

（1）程序运行结果格式

```
TYPE THE ALGORITHM:PRIORITY
OUTPUT OF PRIORITY
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
RUNNING PROC.                    WAITING QUEUE
                                   3     4     1     5
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
ID                          1     2     3     4     5
PRIORITY                    9    38    30    29     0
CPUTIME                     0     0     0     0     0
ALLTIME                     3     3     6     3     4
STATE                       W     R     W     W     W
NEXT                        5     3     4     1     0
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
……
……
……
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
SYSTEM FINISHED
```

（2）说明：

程序启动后，屏幕上显示“TYPE THE ALGORTITHM”，要求用户打入使用何种调度算法。本程序只编制了优先数法（“priority”）和简单轮转法（“Round Robin”）两种。打入某一算法后，系统自动形成各进程控制块，实施该算法的进程调度算法，并打印各进程在调度过程中的状态和参数的变化。

4、 小结

本实习简单地模拟了进程调度的二种方法，在进程运行时间和进程状态变化方面都做了简化，但已能反映进程调度的实质。通过实习能加深对进程调度的理解和熟悉它的实施方法。

三、实习题

（1） 编制和调试示例给出的进程调度程序，并使其投入运行。

（2） 自行设计或改写一个进程调度程序，在相应机器上调试和运行该程序，其功能应该不亚于示例。

提示：可编写一个反馈排队法（FB 方法）的进程调度程序。该算法的基本思想是设置几个进程就绪队列，如队列 1……队列 i ，同一队列中的进程优先级相同，可采用先进先出方法调度。各队列的进程，其优先级逐队降低。即队列 1 的进程优先数最高，队列 i 的最低。而时间片，即以此占用 CPU 的时间正好相反，队列 1 的最短，队列 i 则最长。调度方法是开始进入的进程都在队列 1 中参加调度，如果在一个时间片内该进程完不成，应排入队列 2，即优先级要降低，但下一次运行的时间可加长（即时间片加长了）。以此类推，直至排到队列 i。调度时现在队列 1 中找，待队列 1 中已无进程时，再调度队列 2 的进程，一旦队列 1 中有了进程，又应返回来调度队列 1 的进程。这种方法最好设计成运行过程中能创造一定数量的进程，而不是一开始就生成所有进程。

提示：可综合各种算法的优先，考虑在各种不同情况下的实施方法，如上述 FB 算法。也可选用有关资料中报导的一些方法，加以分析、简化和实现。

（3）直观地评测各种调度算法的性能。

四、思考题

（1） 示例中的程序，没有使用指针型（pointer）数据结构，如何用指针型结构改写本实例，使更能体现 C 语言的特性。

（2） 如何在程序中真实地模拟进程运行的时间片？

（3） 如果增加进程的“等待”状态，即进程因请求输入输出等问题而挂起的状态，如何在程序中实现？

实验（二） 死锁观察与避免

一、目的与要求

1、目的

死锁会引起计算机工作僵死，造成整个系统瘫痪。因此，死锁现象是操作系统特别是大型系统中必须设法防止的。学生应独立的使用高级语言编写和调试一个系统动态分配资源的简单模拟程序，观察死锁产生的条件，并采用适当的算法，有效的防止死锁的发生。通过实习，更直观地了解死锁的起因，初步掌握防止死锁的简单方法，加深理解课堂上讲授过的知识。

2、要求

（1） 设计一个 n 个并发进程共享 m 个系统资源的系统。进程可动态地申请资源和释放资源。系统按各进程的申请动态地分配资源。

（2） 系统应能显示各进程申请和释放资源以及系统动态分配资源的过程，便于用户观察和分析。

（3） 系统应能选择是否采用防止死锁算法或选用何种防止算法（如有多种算法）。在不采用防止算法时观察死锁现象的发生过程。在使用防止死锁算法时，了解在同样申请条件下，防止死锁的过程。

二、示例

1、题目

本示例采用银行算法防止死锁的发生。假设有三个并发进程共享十个系统。在三个进程申请的系统资源之和不超过 10 时，当然不可能发生死锁，因为各个进程申请的资源都能满足。在有一个进程申请的系统资源数超过 10 时，必然会发生死锁。应该排除这二种情况。程序采用人工输入各进程的申请资源序列。如果随机给各进程分配资源，就可能发生死锁，这也就是不采用防止死锁算法的情况。假如，按照一定的规则，为各进程分配资源，就可以防止死锁的发生。示例中采用了银行算法。这是一种犹如“瞎子爬山”的方法，即探索一步，前进一步，行不通，再往其他方向试探，直至爬上山顶。这种方法是比较保守的。所花的代价也不小。

2、算法与框图

银行算法，顾名思义是来源于银行的借贷业务，一定数量的本金要应付各种客户的借贷周转，为了防止银行因资金无法周转而倒闭，对每一笔贷款，必须考察其最后是否能归还。研究死锁现象时就碰到类似的问题，有限资源为多个进程共享，分配不好就会发生每个进程都无法继续下去的死锁僵局。

银行算法的原理是先假定每一次分配成立，然后检查由于这次分配是否会引起死锁，即剩下的资源是不是能满足任一进程完成的需要。如这次分配是安全的（不会引起死锁），就实施这次分配，再假定下一次分配。如果不安全，就不实施，再作另一种分配试探，一直探索到各进程均满足各自的资源要求，防止了死锁的发生。

程序框图如**图 2、3、4。**

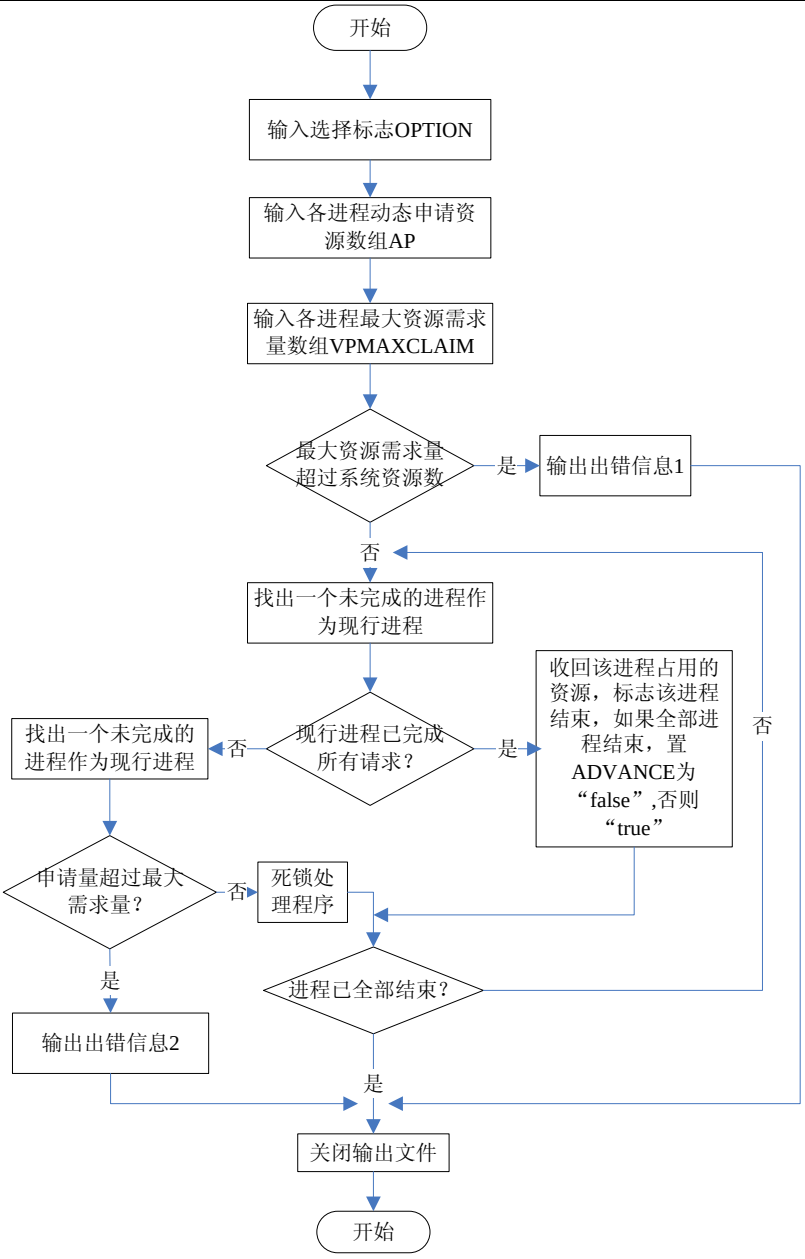

开始
输入选择标志OPTION
输入各进程动态申请资源数组AP
输入各进程最大资源需求量数组VPMAXCLAIM
最大资源需求量超过系统资源数
是
输出出错信息1
否
找出一个未完成的进程作为现行进程
现行进程已完成所有请求？
是
收回该进程占用的资源，标志该进程结束，如果全部进程结束，置ADVANCE为“false”,否则“true”
否
找出一个未完成的进程作为现行进程
否
申请量超过最大需求量？
否
死锁处理程序
是
进程已全部结束？
否
是
输出出错信息2
是
关闭输出文件
开始

图 2 防止死锁程序框图

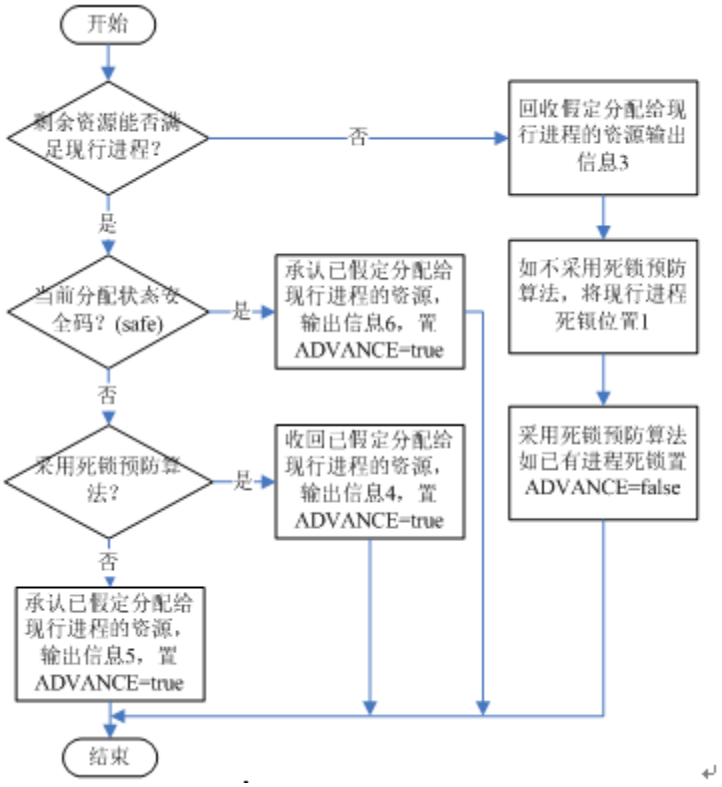


图 3 死锁处理程序框图

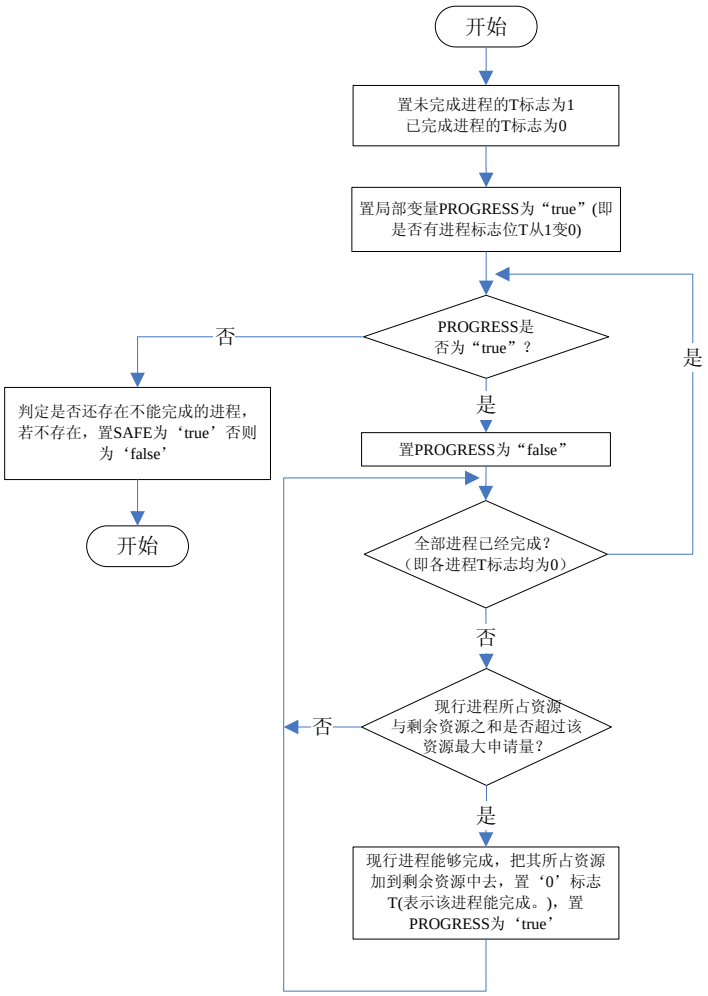


图 4　safe 函数框图

3、程序运行结果格式

（1）程序运行结果格式

```
INPUT:
OPTION =0
CLAIM OF PROCESS 1 IS: 1    2    3    -1    -1    0
CLAIM OF PROCESS 2 IS: 2    3    1    1     -2    0
CLAIM OF PROCESS 3 IS: 1    2    5    -1    -2    0
MAXCLAIM OF PROCESS 1 IS: 6
MAXCLAIM OF PROCESS 2 IS: 7
MAXCLAIM OF PROCESS 3 IS:8
THE SYSTEM ALLOCTION PROCESS IS AS FOLLOWS:
                PROCESS      CLAIM      ALLOCATION      REMAINDER
(1)             1            1          1               9
RESOURCE IS ALLOCATED TO PROCESS 1
(2)             2            2          2               7
RESOURCE IS ALLOCATED TO PROCESS 2
(3)             3            1          1               6
RESOURCE IS ALLOCATED TO PROCESS 3
(4)             1            2          3               4
RESOURCE IS ALLOCATED TO PROCESS 1
(5)             2            3          2               4
IF ALLOCATED,DEADLOCK MAY OCCUR
(6)             1            2          3               4
THE REMAINDER IS LESS THAN PROCESS 2 CLAIMS
(7)             3            0          0               10
PROCESS 3 HAS FINISHED, RETURN ITS RESOURSE
THE WHOLE WORK IS COMPLETED
*********************************************
```

(2) 程序中使用的数据结构和变量名说明如下：

OPTION　　选择标志

=0　选用“防止死锁”算法

=1　不用“防止死锁”算法

AP(I,J)　　资源请求矢量

$$=\begin{cases} n>0 & \text{第 I 进程第 J 次申请 n 个资源；} \\ n<0 & \text{第 I 进程第 J 次释放 n 个资源；} \\ n=0 & \text{第 I 进程在第 J 次终止。} \end{cases}$$

VPMAXCLAIM(I)　第 I 进程对资源的最大需求量；

VALLOCATION(I)　第 I 进程已分配到资源数；

VPSTATUS(I)　第 I 进程完成请求标志，为 1 时表示已完成各次请求；

VCOUNT(I)　第 I 进程请求次数计数器。其值表示该进程的第几次请求；

TOTAL　已分配的系统资源总数；

REMAINDER　剩余的系统资源数；

INQUIRY　当前运行进程号。

(3) 程序中定义的过程和函数说明如下：

FRONT 过程　初始化过程，装入所有初始数据，为各有关变量置初值，检查每个进程请求的资源总数是否超过系统所能提供的资源数。

PRINT 过程　输出一次分配结果。

RETRIEVE 过程　当测得资源不够分配或分配后可能产生死锁，回收已假定分配了的资源。

TERMINATION 过程　检查每个进程时候都已完成或者发生死锁。如果进程全部完成或发生死锁，则将全局变量 ADVANCE 置成 true，不然置成 false。

R_2 过程　为检查进程的分配资源数是否超过了它的最大申请量，或是释放的资源数是否超过占有数，这里是检查例外情况。

R_4 过程　　为各进程设置能执行完标志 T，对于还不能完成的进程将它的标志 T 置 1，能完成的进程标志 T 置 0.

SAFE 函数　　测试在当前分配状态下，会不会产生死锁，若不会，死锁函数值返回 true，否则返回 false。

ALLOCATE 过程　　按申请想当前进程分配或收回资源。

RETURN 过程　　收回当前进程的全部资源，并将此进程的 VPSTATUS 置 1。

（4）程序启动后要求输入各进程的资源请求序列和各种进程的最大申请资源数。同时，要求输入选择标志 OPTION。程序运行后，输出相应结果和有关信息。输出格式如下：（详见运行结果输出）

（I）　n_1　　n_2　　n_3　　n_4

其中：

I　　自然序号

n_1（PROCESS）　　进程号

n_2（CLAIM）　　本次申请资源数

n_3（ALLOCATION）　　已分配给该进程的资源数

n_4（REMAINDER）　　系统的总剩余资源数

4、小结

死锁的防止是比较复杂的。虽然可用于防死锁的方法或是用检测死锁然后予以恢复的方法来解决死锁问题，但花费的代价是很大的。

三、实习题

（1）　编制和调试**示例**给出的死锁观察与避免程序，使其投入运行，并用进程的各种资源请求序列测试死锁的形成和避免死锁的过程。

（2）　修改并调试一个使用有序资源使用法来预防死锁的模拟程序，并用进程的各种资源请求序列测试死锁的形成和预防死锁的过程。

提示：首先要把系统中所有资源类别按其紧缺程度排成一定的序号。要求进程必须严格按递增次序请求资源，即对低序号设备的要求未能满足之前，不准申请高序号的资源，这样就能有效地预防死锁的产生。

四、思考题

（1）　编制一个利用可在使用资源图的资源请求矩阵和分配矩阵的简化运算来检测死锁的模拟程序。

提示：可再使用资源图的分配矩阵为(A_{ij})，元素 A_{ij}=|（R_j，P_i）|；请求矩阵为(B_{ij})，元素 B_{ij}=|（P_i ，R_j）|，还需要一个可用资源向量（r_j），元素 $r_j = t_j - \sum_k |(R_j, P_k)|$。其中，P 表示进程，R 表示资源，t 表示某类资源的总数。

简化过程是：先设法满足请求边，使它变成分配边，然后把只有分配边而无请求边的节点的所有分配边撤销（相当于解放已全部满足某进程的所有资源）。再次检查能否将请求边变成分配边，再撤销已无请求边的所有分配边，直至撤销所有的边，此时，可再使用资源图完全可化简。如果状态是死锁，当且仅当它的可再使用资源图不是完全可化简的。也就是肯定有一些边无法撤消。**本实习要求用矩阵的运算来实现这过程。**

实验（三） 请求页式存储管理

一、目的与要求

1、目的

近年来，由于大规模集成电路（LSI）和超大规模集成电路（VLSI）技术的发展，使存储器的容量不断扩大，价格大幅度下降。但从使用角度看，存储器的容量和成本总受到一定的限制。所以，提高存储器的效率始终是操作系统研究的重要课题之一。虚拟存储技术是用来扩大内存容量的一种重要方法。学生应独立地用高级语言编写几个常用的存储分配算法，并设计一个存储管理的模拟程序，对各种算法进行分析比较，评测其性能优劣，从而加深对这些算法的了解。

2、 要求

为了比较真实地模拟存储管理，可预先生成一个大致符合实际情况的指令地址流。然后模拟这样一种指令序列的执行来计算和分析各种算法的访问命中率。

二、示例

1、题目 本示例是采用页式分配存储管理方案，并通过分析计算不同页面淘汰算法情况下的访问命中率来比较各种算法的优劣。另外也考虑到改变页面大小和实际存储器容量对计算结果的影响，从而可为算则好的算法、合适的页面尺寸和实存容量提供依据。

本程序是按下述原则生成指令序列的：

（1） 50%的指令是顺序执行的。

（2） 25%的指令均匀散布在前地址部分。

（3） 25%的指令均匀散布在后地址部分。

示例中选用最佳淘汰算法（OPT）和最近最少使用页面淘汰算法（LRU）计算页面命中率。公式为

$$命中率 = 1 - \frac{页面失效次数}{页地址流长度}$$

假定虚存容量为 32K，页面尺寸从 1K 至 8K，实存容量从 4 页至 32 页。

2、 算法与框图

（1） 最佳淘汰算法(OPT)。

这是一种理想的算法，可用来作为衡量其他算法优劣的依据，在实际系统中是难以实现的，因为它必须先知道指令的全部地址流。由于本示例中已预先生成了全部的指令地址流，故可计算出最佳命中率。

该算法的准则是淘汰已满页表中不再访问或是最迟访问的的页。这就要求将页表中的页逐个与后继指令访问的所有页比较，如后继指令不在访问该页，则把此页淘汰，不然得找出后继指令中最迟访问的页面淘汰。可见最佳淘汰算法要花费较长的运算时间。

（2） 最近最少使用页淘汰算法(LRU)。

这是一种经常使用的方法，有各种不同的实施方案，这里采用的是不断调整页表链的方法，即总是淘汰页表链链首的页，而把新访问的页插入链尾。如果当前调用页已在页表内，则把它再次调整到链尾。这样就能保证最近使用的页，总是处于靠近链尾部分，而不常使用的页就移到链首，逐个被淘汰，在页表较大时，调整页表链的代价也是不小的。

（3） 程序框图如下**图 5** 示。

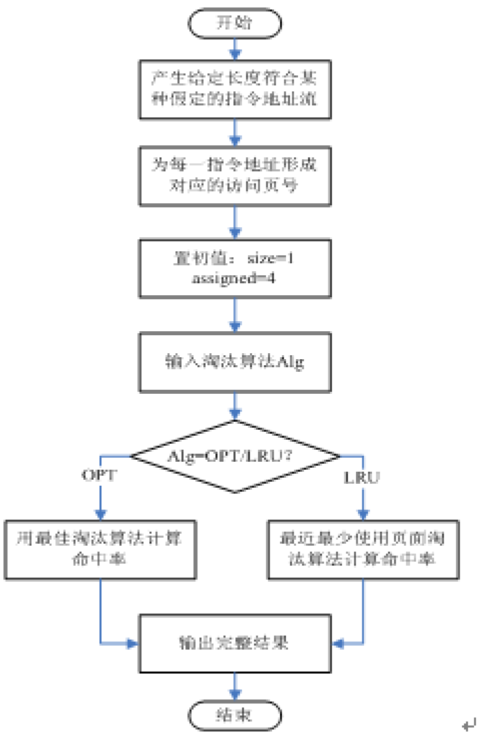


图 5 计算页面命中率框图

3、程序运行结果格式

(1) 程序运行结果格式

THE VIRTUAL ADDRESS STREAM AS FOLLOWS:

a[0] =16895　a[1]=16896　a[2]=16897　a[3]=16302

a[4]=25403　a[5]=13941　a[6]=13942　a[7]=8767

……

……

……

A[252]=23583　a[253]=20265　a[254]=20266　a[255]=20267

= =

The algorithm is:opt

PAGE NUMBER WITH SIZE 1k FOR EACH ADDRESS IS:

pageno[0]=17　pageno[1]=17　pageno[2]=17　pageno[3]=16

pageno[4]=25　pageno[5]=14　pageno[6]=14　pageno[7]=9

……

……

……

pageno[252]=24　pageno[253]=20　pageno[254]=20　pageno[255]=20

vmsize=32k　pagesize=1k

------------- --------------

page assigned	pages_in/total references
4	7.031250000000000E-1
6	7.578125000000000E-1
8	8.085937500000000E-1
10	8.554687500000000E-1
12	8.945312500000000E-1
14	9.140625000000000E-1
16	9.140625000000000E-1
18	9.140625000000000E-1
20	9.140625000000000E-1
22	9.140625000000000E-1
24	9.140625000000000E-1
26	9.140625000000000E-1
28	9.140625000000000E-1
30	9.140625000000000E-1

32 9.140625000000000E-1
PAGE NUMBER WITH SIZE 2k EACH ADDRESS IS:
……
……
……
PAGE NUMBER WITH SIZE 4k EACH ADDRESS IS:
……
……
……
PAGE NUMBER WITH SIZE 8k EACH ADDRESS IS:
……
……
……
End the result for opt
** **********
the algorithm is lru
……
……
……同上
End the result for lru
****

(2)示例中使用的有关数据结构、常量和变量说明如下：

length 被调试的指令地址流长度，可作为一个常量设定。

called 当前请求的页面号。

pagefault 页面失效标志，如当前请求页 called 已在页表内，则置 pagefault=false，否则为 true。

table 页表。table[i]=j,表示虚存的第 j 页在实存的第 i 页中。

used 当前被占用的实存页面数，可用来判断当前实存中是否有空闲页。

（3）本程序启动后，屏幕上显示“the algorithm is:”，用户可选择最佳淘汰算法（打入“OPT”）或者最近最少使用淘汰算法(打入“LRU”) 计算页面命中率。当然还可以加入各种其他的算法。

4、小结

(2) 编制评测各种算法性能的模拟程序是研制系统程序，尤其是操作系统所必须的。模拟的环境愈是真实，其结果愈是可靠，也就更有利于选择合适的方案。本实习虽属简单，但可作为一个尝试。

(3) 注意正整数的范围只能从 0……32767，限制程序中的虚存尺寸为 32K，实际如采用更大的虚存实存，更能说明问题。

三、实习题

（1） 编制和调试**示例**给出的请求页式存储管理程序，并使其投入运行。

（2） 增加 1～2 种已学过的淘汰算法，计算它们的页面访问命中率。试用各种算法的命中率加以比较分析。

提示：可选用 FIFO 方法，即先访问的页先淘汰，也可选用 LRU 方法中的其他方案。如在页表中设置标志位，按标志位值得变化来淘汰。也可用 LFU 方法，为页表中各页面设置访问计数器，淘汰访问频率最低的页（注意：当前访问的页不能淘汰）等等。

四、思考题

（1）设计一个界地址存储管理的模拟系统，模拟界地址方式下存储区的分配和回收过程。

提示：必须设置一个内存分配表，按照分配表中有关信息实施存储区的分配，并不断根据存储区的分配和回收修改该表。算法有首次匹配法，循环首次匹配法和最佳匹配法等。可用各种方法的比较来充实实习内容。可使用碎片收集和复盖等技术。

（2）自行设计或选用一种较为完善的内存管理方法，并加以实现。

提示：设计一个段页式管理的模拟程序或通过一个实际系统的消化和分析，编制一个程序来模拟该系统。

实验（四）文件操作与管理

一、目的与要求

1、目的

随着社会信息量的极大增长，要求计算机处理的信息与日俱增，涉及到社会生活的各个方面。因此，文件管理是操作系统的一个极为重要的组成部分。学生应独立地用高级语言编写和调试一个简单的文件系统，模拟文件管理的工作过程。从而对各种文件操作命令的实质内容和执行过程有比较深入的了解，掌握它们的实施方法，加深理解课堂上讲授过的知识。

2、要求

（1） 实际一个 n 个用户的文件系统，每个用户最多可保存 m 个文件。

（2） 限制用户在一次运行中只能打开 l 个文件。

（3） 系统应能检查打入命令的正确性，出错要能显示出错原因。

（4） 对文件必须设置保护措施，如只能执行，允许读、允许写等。在每次打开文件时，根据本次打开的要求，再次设置保护级别，即可有二级保护。

（5） 对文件的操作至少应有下述几条命令：

creat 建立文件。
delete 删除文件。
open 打开文件。
close 关闭文件。
read 读文件。
write 写文件。

二、示例

1. 题目

（1） 本实习设计一个 10 个用户的文件系统，每个用户最多可保存 10 个文件，一次运行中用户可打开 5 个文件。

（2） 程序采用二级文件目录，即设置了主文件目录（MFD）和用户文件目录（UFD）。前者应包含文件主（即用户）及他们的目录区指针；后者应给出每个文件主占有的文件目录，即文件名，保护码，文件长度以及他们存放的位置等。另外为打开文件设置了运行文件目录（AFD），在文件打开时应填入打开文件号，本次打开保护码和读写指针等。

（3） 为了便于实现，对文件的读写作了简化，在执行读写命令时，只修改读写指针，并不进行实际文件的读写操作。

2. 算法与框图

（1） 因系统小，文件目录的检索使用了简单的线性搜索，而没有采用 Hash 等有效算法。

（2） 文件保护简单实用了三位保护码，对应于允许读、允许写和运行执行，如下所示：

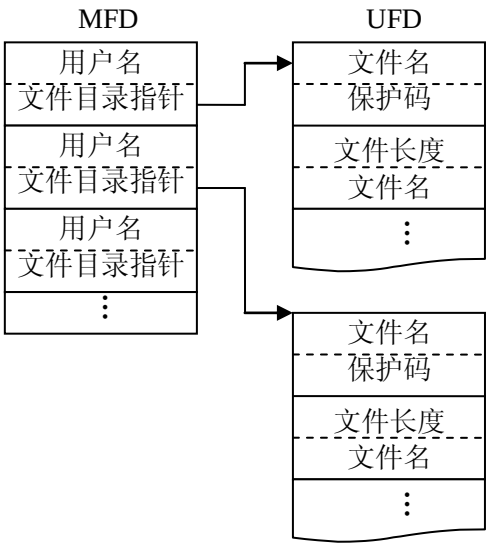
1 1 1

允许写 允许读 允许执行

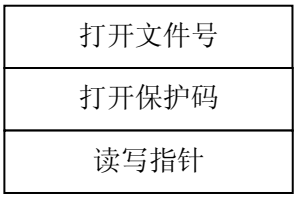
如对应位为 0，则不允许。

（3） 程序中使用的主要数据结构如下：

①主文件目录和用户文件目录



打开文件目录



（4） 程序框图如**图 6** 示。

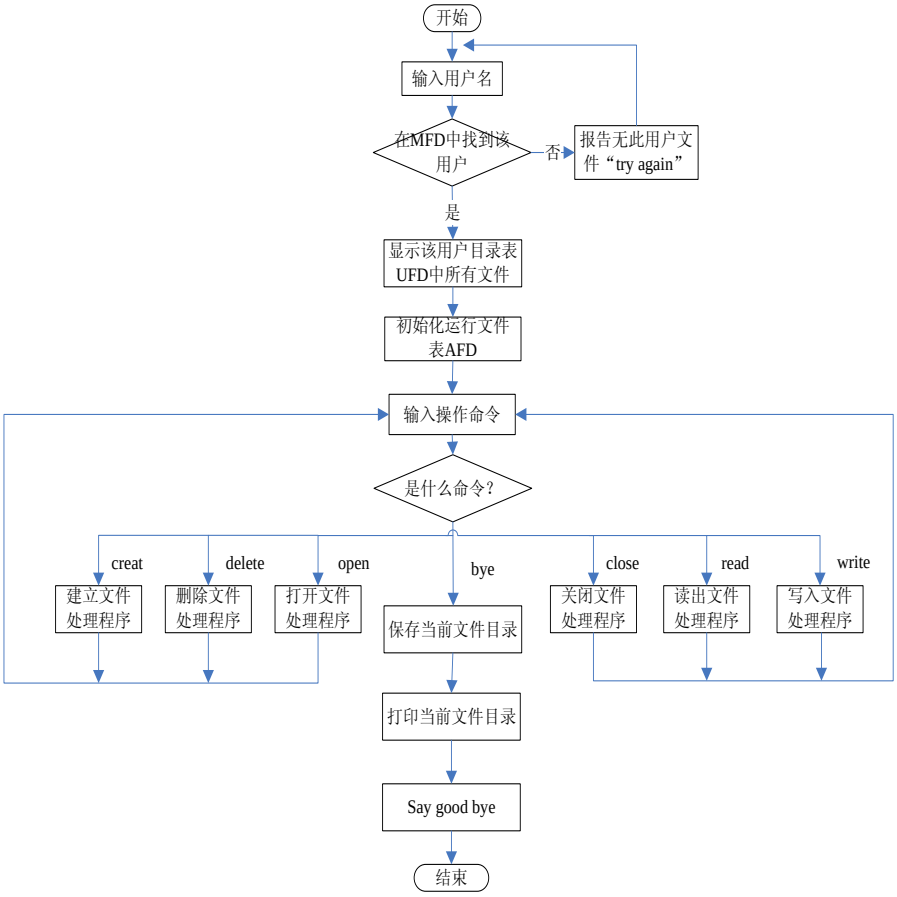


图 6 文件系统框图

3. 程序运行结果格式

(1) 程序运行结果格式

RUN

```
YOUR NAME ?     YOUIJIN
YOUR NAME IS NOT N THE USER NAME TABLE,TRY AGAIN.
YOUR NAME ?     YOUJIN
YOUR FILE DIRECTORY
FILE NAME       PROTECTION          CODE LENGTH
XUMAIN              111                 9999
F1                  111                  0
YOUJINYU            111                 100
******              000                  0
******              000                  0
******              000                  0
******              000                  0
******              000                  0
******              000                  0
COMMAND NAME? CREATER
COMMAND NAME GIVEN IS WRONG!
IT SHOULD BE ONE OF FOLLOWING : CREATE, DELETE, OPEN, CLOSE, READ, WRITE, BYE.TRY AGAIN
COMMAND NAME?CREATE
THE NEW FILE S NAME(LESS THAN 9 CHARS)?          F2
THE NEW FILE'S PROTECTION CODE?                  101
THE NEW FILE IS CREATED.
ENTER THE OPEN MODE?                             101
THIS FILE IS OPENED,ITS OPEN NUMBER IS            1
COMMAND NAME?       READ
OPEN   FILE   NUMBER?
ERROR MESSAGE:IT IS NOT ALLOWED TO READ THIS FILE   !!!
COMMAND NAME?WRITE
OPEN FILE NUMBER?                                 1
HOW MANY CHARACTERS TO BE WRITTEN INTO THAT FILE?      190
COMMAND NAME ?   OPEN
FILE NAME TO BE OPENED?                          F1
ENTER THE OPEN MODE?                             111
THIS FILE IS OPENED,ITS OPEN NUMBER IS            2
COMMAND NAME?     WRITE
OPEN FILE NUMBER?                                 2
……
……
……
COMMAND NAME?   CLOSE
THE OPENED FILE NUMBER TO BE CLOSED?              2
THIS FILE IS CLOSED.
COMMAND NAME?     BYE
NOW YOUR FILE DIRECTORY IS FOLLOWING:
XUMAN       111             9999
F1          111             1900
YOUJINYU    111             100
F2          101             190
*****       000             0
*****       000             0
*****       000             0
*****       000             0
GOOD BXE.
```

(2)本程序用交互方式工作。

启动程序后，系统查询：YOUR NAME?打入用户名，且已登入主目录时，系统就响应。注意，本程序中前一个编入登录用户名的程序，用户需实现在主目录中登入用户名。系统响应后会给出用户文件目录，然后显示：COMMAND NAME? 打入相应命令后就可建立、删除、读、写、打开和关闭文件，如命令打错，系统会指出并给用户提示。操作完成后应关闭文件，然后打入“BYE”命令退出文件系统。退出前系统再次打印当前文件目录。

4.小结

文件系统的管理有各种各样的方案和算法。由于实习时间和条件的限制，本示例只是简单的模拟了文件的几种操作命令，学生可从各种管理程序和数据库管理的实习中进一步深入探讨。

三、实习题

（1） 编制和调试**示例**给出的文件操作与管理程序，并使其投入运行。

（2） 增加 2～3 个文件操作命令，并加以实现。

提示：可以增加移动读写指针命令，如把指针移至某一起始位置或文件头；改变文件属性的命令，如更改文件名，改变文件保护级别等。

四、思考题

（1）编制一个通过屏幕选择命令的文件管理系统，每幅屏幕要为用户提供足够的选择信息，不需要打入冗长的命令。

提示：为了便于用户操作，微机大多采用按照屏幕的提示选择命令，这里可以使用高级语言编制通过屏幕显示选择文件操作命令的文件管理模拟程序。

（2）设计一个树形目录结构的文件系统，其根目录为 root，各分支可以是目录，也可以是文件，最后的叶子都是文件。

提示：可以参考 UNIX 操作系统的文件结构和管理方法。可采用多级保护，即把用户分成文件主，伙伴和普通用户三类，分别给予使用权。为了缩短搜索文件的路径，可设置工作目录（或值班目录），使能在当前使用的目录下查找文件，不必每次都从根目录开始查找。

（3）根据学校的各级机构，编制一个文件系统，要求上级机构能查阅和修改下级机构的文件，而下级机构只有在授权情况下才能查阅上级的文件，但不能修改，同一级的文件可以共享。

提示：学校机构可由如下**图 7 组成。**

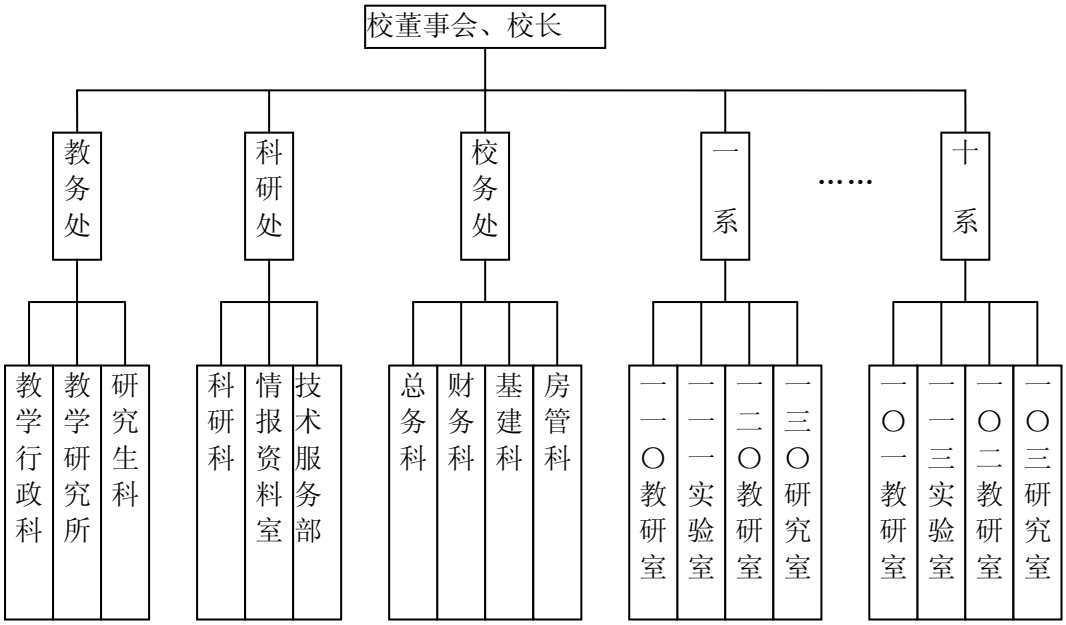


图 7　学校机构图

实验(五) Linux 文件实验

一. 实验目的

掌握操作系统中文件分类的概念。
了解 Linux 文件系统管理文件的基本方式和特点。
学会使用 Linux 文件系统的命令界面和程序界面的基本要领。

二. 实验准备

复习操作系统中有关文件系统的知识，熟悉文件的类型、i 节点、文件属性、文件系统操作等概念。
熟悉《实验指导》第五部分“文件系统的系统调用”。了解 Linux 文件系统的特点、分类。
阅读例程中给出的相应的程序段。

三. 实验方法

运行命令界面的各命令并观察结果。
用 vi 编写 c 程序（假定程序文件名为 prog1.c）
编译程序
$ gcc -o prog1.o prog1.c 或 $ cc -o prog1.o prog1.c
运行
$./prog1.o
观察运行结果并讨论。

四. 实验内容及步骤

1. 用 shell 命令查看 Linux 文件类型。

📖思考：Linux 文件类型有哪些？用什么符号表示。

2. 用 shell 命令了解 Linux 文件系统的目录结构。

执行
$ cd /lib
$ ls -l|more
看看/lib 目录的内容，这里都是系统函数。再看看/etc，这里都是系统设置用的配置文件；/bin 中是可执行程序；/home 下包括了每个用户主目录。

3. 用命令分别建立硬链接文件和符号链接文件。通过 ls -il 命令所示的 inode、链接计数观察它们的区别

找找一个其他目录中的文件，如：/home/zzl/mytest.c 执行
$ ln /home/zzl/mytest.c myt.c （建立硬链接文件）
$ ln -s /home/zzl/mytest.c myt2.c （建立符号链接文件）
📖思考：建立硬链接文件和建立符号链接文件有什么区别，体现在哪里？

4. 复习 Unix 或 Linux 文件目录信息 i 节点的概念。编程察看指定文件的 inode 信息。
【提示】以下是“获得 inode 信息实验”的例程。将程序稍加修改，进而实现题目要求
例程 8：获得 Inode 信息实验
/*利用 lstat() 系统调用获得 inode 信息。假定本程序文件名为 getiinfo.o 则执行程序，获得 filename 文件的 inode 信息的命令格式如下 ./getiinfo.o filename */

```
#include<sys/stat.h>
#include<sys/types.h>
#include<sys/sysmacros.h>
#include<stdio.h>
#include<time.h>
#include<unistd.h>
#include<string.h>
#include<errno.h>
```

```
#define TIME_STRING_LEN 50
char *time2String(time_t tm,char *buf)
{ struct tm *local;
local=localtime(&tm);
strftime(buf,TIME_STRING_LEN,"%c",local);
return buf;
}
int ShowFileInfo(char *file)
{ struct stat buf;
   char timeBuf[TIME_STRING_LEN];
if(lstat(file,&buf))
{ perror("lstat() error");
return 1;
}
printf("\nFile:%s\n",file);
printf("On device(major/minor):%d %d,inode number:%ld\n",
major(buf.st_dev),minor(buf.st_dev),buf.st_ino);
printf("Size:%ld\t Type: %07o\t Permission:%05o\n",buf.st_size,buf.st_mode & S_IFMT,buf.st_mode & ~(S_IFMT));
printf("Ower id:%d\t Group id:%d\t Number of hard links:%d\n", buf.st_uid,buf.st_gid,buf.st_nlink);
printf("Last access:%s\n",time2String(buf.st_atime,timeBuf));
printf("Last modify inode:%s\n\n",time2String(buf.st_atime,timeBuf));
return 0;
}
int main(int argc,char *argv[])
{ int i,ret;
for(i=1;i<argc;i++)
{ ret=ShowFileInfo(argv[i]);
if(argc-i>1) printf("\n");
}
return ret;
}
```

📖思考：Linux 文件的 inode 是不是很有特色？找一些这方面的资料，熟悉文件系统的实现方法，会让你的水平提升一个台阶的。

5. 再来一个更有趣的实验。修改父进程创建子进程的程序，用显示程序段、数据段地址的方法，说明子进程继承父进程的所有资源。再用父进程创建子进程，子进程调用其它程序的方法进一步证明子进程执行其它程序时，程序段发生的变化。

【提示】这个实验可参考例程 3 中“父进程创建子进程，子进程调用其它程序的例”以及下面例程 10 “显示程序段、数据段地址的程序”两个程序。设法在子进程运行的程序中显示程序段、数据段地址，以此说明：开始时子进程继承了父进程的资源，一旦子进程运行其它程序，就用该程序替换从父进程处继承来的程序段和数据段。

例程 10：显示程序段、数据段地址的程序

例程 10-1：

```
/*系统保持每一个用户进程的相关的虚拟地址,这些地址可以通过引用外部变量 etext,edata,end 来获得*/
#include<stdio.h>
extern int etext,edata,end;        /*对应文本的第一有效地址、初始化的数据*/
main()
{
printf("etext:%6x \t edata:%6x \t end:%6x \n",&etext,&edata,&end);
}
```

例程 10-2：

```
/*Print the address of program text, the adderess of data area, etc*/
#include<stdio.h>
#include<string.h>
#include<sys/types.h>
#include<stdlib.h>
#include<unistd.h>
#define SHW_ADR(ID,I) printf("The id %s \t is at adr:%8X\n",ID,&I);
extern int etext,edata,end;
```

```
char *cptr="Hello World.\n";
char buffer1[25];
main()
{     void showit(char *);
int i=0;
printf("Adr etext:%8x\t Adr edata:%8x Adr end:%8x\n\n",&etext,&edata,&end);
SHW_ADR("main",main);
SHW_ADR("showit",showit);
SHW_ADR("cptr",cptr);
SHW_ADR("buffer1",buffer1);
SHW_ADR("i",i);
strcpy(buffer1,"A demonstration\n");
write(1,buffer1,strlen(buffer1)+1);
for(;i<1;++i)
showit(cptr);
}
Void showit(char *p)
{
char *buffer2;
SHW_ADR("buffer2",buffer2);
if((buffer2=(char *)malloc((unsigned)(strlen(p)+1)))!=NULL)
{     strcpy(buffer2,p);
printf("%s",buffer2);
free(buffer2);
}
else{
printf("Allocation error.\n");
exit(1);
}
}
```

6. 编写一个涉及流文件的程序。要求：

- ◆ 以只读方式打开一个源文本文件
- ◆ 以只读方式打开另一个源文本文件
- ◆ 以只写方式打开目标文本文件
- ◆ 将两个源文件内容复制到目标文件
- ◆ 将目标文件改为指定的属性（其他人只读、文件主可读写）
- ◆ 显示目标文件

【提示】这个实验可参考例程 11 中“打开流文件进行行输入输出操作”的例程，当然还得自己加工。将程序再加修改为有两个源文件，一个目标文件，进而实现题目要求。

例程 11 打开流文件进行输入输出操作

```
/*Open the file and put the line into screen*/
/*Command format: ./command filename      */
#include<sys/types.h>
#include<stdio.h>
int main(int argc,char *argv[])
{
char s[1024];
FILE *fp;
if((fp=fopen(argv[1],"r"))!=(FILE*)0) /*打开参数 1 指定的文件*/
{
while((fgets(s,1024,fp))!=(char *)0)
puts(s); /*显示缓冲区*/
            }else
            {
fprintf(stderr,"file open error.\n");
exit(1);
}
exit(0);
```

}

🕮思考：你的程序用到哪那些设备文件操作？你对设备编程了吗？看来没有吧。文件在磁盘上，但文件操作很简单，这些都是操作系统提供的方便。这是不是叫 “设备无关性”、“设备独立性”呢？

五.研究并讨论

1. **硬链接文件和符号链接文件。有什么区别？系统如何处理的？举例说明。**
2. **从实验 6 的结果可以让我们了解父、子进程之间在资源共享方面是如何处理的？**
3. **查找资料讨论 Linux 的文件系统有什么特点？它是如何兼容各类文件系统的？**
4. **系统如何管理设备的?怎样体现“与设备无关”的思想方法？**

实验(六) FAT 文件系统实验

一. 实验目的：

从系统分析的角度出发，了解 FAT 文件系统的组织结构和文件的存储方式。进一步理解操作系统文件管理的基本思想。

二. 实验环境：

Windows 98 或 Windows 2000（使用 Debug.exe），FAT 文件系统 3 吋软盘。

三. 实验内容：

- ◆ 了解 3 吋软盘的 FAT 文件系统结构。
- ◆ 察看文件分配表的簇号链。
- ◆ 察看文件目录表中文件目录结构。
- ◆ 了解用簇号链映射的文件链式存储结构。
- ◆ 分析目录文件的组成。

四. 实验准备：

复习文件组成，文件描述目录信息内容及含义。

复习文件系统的存储结构概念。

复习文件存储空间管理方法。

五. 实验步骤：

1. 进入 DEBUG 环境，装入 FAT 文件系统结构。

- 执行命令： L 0 0 0 21↙

说明：将 0 号驱动器中，逻辑扇区号从 0 开始的共 21H 个扇区读入内存，放在DS:0000 为起始的地址中。

2. 观察 1.44M 软盘中 FAT12 文件系统结构。

- 执行命令： D 0000↙ （显示从 0 地址开始的内存）
- 连续执行 D 命令，每次显示 128 个字节，可见文件系统结构。

FAT 文件系统结构如下：

逻辑扇区号

Boot	FAT 1	FAT 2	FDT	数据区.....
0H	1H~9H	AH~12H	13H~20H	21H

其中：

Boot 引导程序

FAT 文件分配表

FDT 文件目录表

1.44M 软盘逻辑扇号与物理扇区的对应关系

逻辑扇号 0# —— 0 道 0 面 1 扇

逻辑扇号 1 H ~11H —— 0 道 0 面 2 ~18 扇

逻辑扇号 12 H ~23H —— 0 道 1 面 1 扇~18 扇

逻辑扇号 24 H ~35H —— 1 道 0 面 1 扇~18 扇

⋮ ⋮

- 逐个观察可用以下命令

命令	说明
D 0000↙	显示从 0 地址开始的 Boot 引导程序
D 200↙	显示从 200 地址开始的 FAT 文件分配表 1
D 2600↙	显示从 2600 地址开始的 FDT 文件表
L 8000 0 21 8↙	将逻辑扇区 21H开始的 8 个物理扇区装入DS:8000 起始的内存
D 8000↙	显示从 8000 H地址开始的数据区

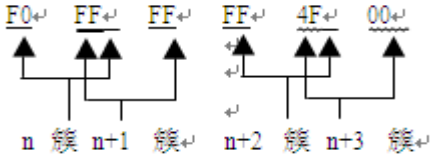
3. 分析文件分配表结构，了解用簇链映射的文件的链式存储结构。

- 执行命令：D 200↙

屏幕显示:

```
0AEA:0200  F0 FF FF FF 4F 00 05 60-00 07 80 00 09 F0 FF 0B    ....O..`........
0AEA:0210  B0 41 0D E0 00 0F 00 01-11 20 01 13 40 01 FF 6F    .A.......@..o
0AEA:0220  01 17 80 01 19 A0 01 1B-C0 01 FF EF 01 1F 00 02    ................
0AEA:0230  21 F0 FF 1A 44 02 25 60-02 FF 8F 02 29 A0 02 2B    !...D.%`....)..+
-
```

FAT 中每三个字节指示了两个簇，如:



若 n 为 0，且按 DEBUG 内存信息显示方式：000 簇中为 FF0、001 簇中为 FFF、002 簇中为 FFF、003 簇中为 004。

其中： 000 簇和 001 簇中包含了磁盘类型

002 簇对应了数据区 21H 逻辑扇，现在为 FFF，表示此乃文件最后一簇。

003 簇中为 004 表示下一簇号为 004。

簇链：文件的目录中指示了本文件在 FAT 中的首簇号；FAT 中每一簇的位置内指示的是该文件的下一簇号，以此类推，直到某一簇位置中用 FFF 表示文件最后一簇。

🕮思考：首簇号为 003 的文件共包括几个扇区？它分布在哪几个物理扇区中？

4. 观察 1.44M 软盘中文件目录表 FDT 以及文件目录结构

- 执行命令： L 0 0 0 21↙

说明：将逻辑扇区 0 H开始的共 21 H个物理扇区装入DS:0000 H起始的内存。

- 执行命令： D 2600↙

说明：显示从 2600 H地址开始的FDT文件表。

屏幕显示：

```
-d2600
0AEA:2600  30 31 30 41 31 46 30 41-32 44 31 28 00 00 00 00    010A1F0A2D1(....
0AEA:2610  00 00 00 00 00 00 AC 55-5F 2B 00 00 00 00 00 00    ......U_+......
0AEA:2620  41 6F 00 66 00 66 00 69-00 63 00 0F 00 9C 65 00    Ao.f.f.i.c....e.
0AEA:2630  2E 00 70 00 70 00 74 00-00 00 00 00 FF FF FF FF    ..p.p.t.........
0AEA:2640  4F 46 46 49 43 45 20 20-50 50 54 20 00 06 8C 4C    OFFICE  PPT  ...L
0AEA:2650  41 2C 90 2F 00 00 5C 4D-41 2C 91 00 00 F2 00 00    A,./..\MA,......
0AEA:2660  E5 31 36 42 34 30 30 30-20 20 20 20 00 6D D1 50    .16B4000    .m.P
0AEA:2670  41 2C 41 2C 00 00 58 54-41 2C D5 03 00 8A 00 00    A,A,..XTA,......
-
```

文件目录表中每 32 个字节是一个文件的目录项，以上显示了 4 个目录项。第 5~6 行是文件 office.ppt 的目录项：

目录项结构：

0 H ~7 字节　　文件名

8 H ~0A H 字节　扩展名

0B H 字节　　文件属性

0C H ~15 H 字节　保留

16 H ~19 H 字节　最后创建修改时间、日期

1A H ~1B H 字节　首簇号

文件属性：

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
未用	未用	归档	子目录	卷标	系统	隐含	只读

📖思考：

① 计算 1.44M 软盘根目录最多可以容纳多少文件？

② 上面屏幕显示的文件 office.ppt 的目录项中标示该文件的首簇号在何处？该文件是什么属性的？

③ 上面屏幕显示第 1~2 行目录项表示的是什么项目？

5. 观察 1.44M 软盘中文件目录表的长文件名目录结构

文件目录表的长文件名采用连续扩充目录项方法，结构如下:

内容	说明
长名最后项（可含 13 个字符）	第一字节的b_6位为 1
… …	
长名第二项（可含 13 个字符）	第一字节的b_5~b_0位为 00002
长名第一项（可含 13 个字符）	第一字节的b_5~b_0位为 00001
别名（短文件名）	

长文件名的目录项中长名字符占用 01~0A H、0E~19 H、1C~1D H 位（共 13 个字符，每个字符两位）。0B H 属性位为 0F H。每个目录项第一字节的 b5~b0 位为扩充目录项目序号，最后一项的第一字节增加 b6 位为 1，表示结束。如：某个目录项的第一字节为 46H，则说明当前是最后一项，且该长文件名占六个目录项。

使用长名的文件目录项中，别名用长名的开头 6 个字符大写加“~1”，若有相同的，用后面一个数字区别。文件属性、日期、时间、首簇号、长度等信息都在短文件名目录项中说明。

📖思考:

① 上面屏幕显示的 2~3 行是什么目录项？

② 若有一个文件名共长 34 个字符，要占多少目录项？试一试。

6. 自己动手做:

① 1观察测试软盘的 FDT 区，找到名为 BAK 的文件目录。该文件是什么类型的文件？文件放在磁盘的哪个位置？占用几个存储单位？调出其内容看看。

操作提示： 用 L 命令，调出 FDT 区。
察看属性域，判断类型。
察看首簇号，计算位置。
察看簇链。
再用 L 命令调出文件。
察看内容。

② 找到文件 123.TXT 的存储位置。调出文件的第二块将文件的第 512 字节开始的 128 个字节，改成“Happy New Year!”。再写回原文件。并用常规方式打开文件观察效果。

操作提示：再用 L 命令调出 BAK 文件。
察看内容，找到所需文件。
察看首簇号，计算位置。
察看簇链。
再用 L 命令调出文件。
用 F 命令修改字符串。
用 W 命令写回磁盘。

📖思考：写出操作步骤。

六. 总结：

① FAT 文件系统的结构是如何组织的？

② 系统怎样实现文件的逻辑块到物理块的映射？

七. 附录：

常用 DEBUG 命令

命令	格式
显示数据	D [address]
显示数据范围	D [address] [L length]
读数据	L address driver startsector sectornum
指定文件	N filspec
读文件	L address
比较	C [address1] [L length] address2
编辑数据	E address value
填充数据	F address “string”
十六进制算术运算	H value value
检索	S [address] [L length] value
数据传送	M [address] [L length] address
写磁盘	W address driver startsector sectornum
写文件	W address （长度用 R 命令放在 BX、CX 寄存器中）
设置寄存器参数	R register
读端口	I port
写端口	O port address bytes
输入汇编程序	A [address]
反汇编	U address
数据移动	M [address] [L length] address
执行程序	G [=address] [address]
单步执行	T [=address]
执行程序段	P [address][value]
退出	Q

实验（七） 内存分配和设备管理实验

一、实验目的

了解 Linux 管理设备的基本方式。

二、实验准备

复习设备管理基本原理。

三、实验内容

用 ls －l 命令观察设备文件的组织方式

$ ls －l /dev

$ ls /dev | wc > data.out (将设备文件名通过管道送到 wc 命令计算设备文件名的行数，结果重定向传送到文件 data.out 中，计算设备文件个数。)

$ cat data.out （显示结果）

📖思考：Linux 管理设备的方法与管理文件的方式有何异同?为什么用管文件的方式来管设备?有什么好处？

1. 参照例程 12 编程，显示设备文件的设备号信息。

例程 12 显示设备文件的设备号信息

```
#include<sys/types.h>
#include<sys/stat.h>
#include<sys/sysmacros.h>
#include<stdio.h>
#include<time.h>
#include<unistd.h>
#include<string.h>
#include<errno.h>
int main(int argc,char *argv[])
{
    int i;
    struct stat buf;
for(i=1;i<argc;i++)
{
printf("%s",argv[i]);
if(lstat(argv[i],&buf)<0)
{
error("lstat error");
continue;
}
printf(" dev=%d %d ",major(buf.st_dev),minor(buf.st_dev));
if(S_ISCHR(buf.st_mode)||S_ISBLK(buf.st_mode))
{
printf(" (%s)rdev=%d %d ",(S_ISCHR(buf.st_mode)),"character:block ",major(buf.st_rdev),minor(buf.st_rdev));
}
   printf("\n");
}
```

四、讨论

Linux 管理设备的特点。

实验（八） 编制一个自己的 Shell

一. 实验目的

综合所学知识，增加感性认识，加强对操作系统实现方式的理解。

二. 实验准备

复习操作系统相关的进程、进程通信、文件系统、输入输出系统等基本概念。熟悉本《实验指导》第六部分的系统调用的使用。

三. 实验内容

模拟 Linux 的 shell,实现一个命令解释器。从标准输入读入命令行并执行。每次只处理一个命令，不过可以是一个复杂命令，例如可以包括管道、输入输出重定向、后台执行等。

程序一共可分为十个模块，每个模块基本上是一个函数。各模块功能如下：

Head.h 一些宏定义和数据结构的定义；

Main.c 主过程；

Init.c 初始化模块，包括两个初始化，一个是启动 shell 时的初始化 init_once()，另一个是每运行完一条命令后的初始化 init_command()；

Get_comln.c 得到当前输入命令；

Get_simcom.c 对输入的命令进行语法分析，将命令分解为简单命令，得到简单命令的数组；

Get_word.c 得到下一个标识符；

Runj_com.c 执行一条简单命令；

Run_com.c 执行输入的命令；

Get_simarg.c 对一条简单命令进行语法分析，得到简单命令的名称和参数；

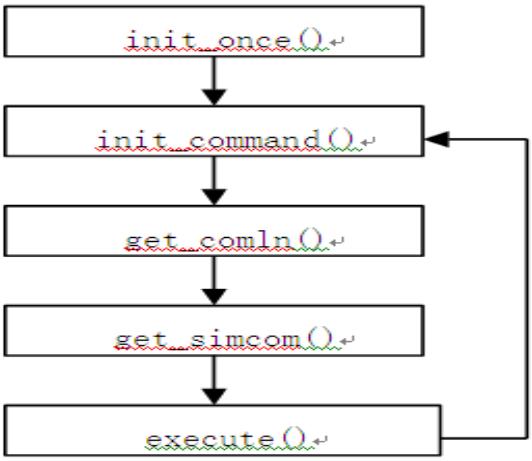
Check.c 查看字符串是否匹配，inputln[]数组中 inputlnptr 指向的字符串和 check 函数的参数字符串是否匹配。

头文件定义了一些在各个模块公用的数据结构。执行时从主过程 main 开始，在其中读入、解释并运行用户输入的命令。命令处理的基本过程如下图，其它函数用来实现一些辅助功能。

【提示】 本实验的设计可以在 Linux 中实现也可以在 Windows 环境下进行。若在 Linux 中实现，参考《实验指导》第五部分“Linux 编程系统调用”中有关“进程管理的系统调用”和“标准输入输出”的有关内容。若在 Windows 环境下进行，则请参考相关 C 函数库。

命令处理过程见以下流程图。

命令处理过程示意:



四. 思考并讨论

1. 操作系统的结构是如何实现的？

2. 系统设计中模块化、标准化思想如何使第三方比较方便地添加模块和驱动程序的？

第二部分　文件系统的系统调用

1. Linux 的文件系统的特点

Linux 的最大特点之一是它支持许多不同的文件系统。这使它很容易地与其他操作系统共存于同一台机器上。Linux 已经能够支持三十几种不同的文件系统，包括 ext、ext2、umsdos、msdos、proc、smb、ncp、iso9660、sysv、hpfs、affs、ufs、ntfs 等。

Linux 的文件系统的另一个特点是将 I/O 子系统结合到文件系统中。Linux 将对外围设备的 I/O 操作设计成与文件的 I/O 操作完全一样，因此无论处理什么设备，操作上都和文件的输入输出一样。所有设备都放在文件系统的/dev 目录下，这样使设备的输入输出有完全相同的接口，从而使操作和应用程序开发都很方便。

Linux 内核的虚拟文件系统（VFS）使系统能支持多个不同的文件系统,为此虚拟文件系统维护了一些描述整个虚拟文件系统信息和真实文件系统信息的数据结构。每一个 mount(装载)的文件系统都由一个 VFS 超级块来说明。VFS 超级块包含如下信息：

1) 设备。　文件系统所在的块设备号。

2) I 节点指针。　包括 mounted I 节点指针（指向装载的文件系统中第一个 I 节点）和 covered I 节点指针（所挂接的目录的 I 节点）。

3) 数据块大小。　以字节为单位的本文件系统的块的大小。

4) 超级块操作例程。　指向本文件系统所支持的一些超级块例程的指针。

5) 文件系统类型。　指向已经 mount 的文件系统的 file_system_type 数据结构的指针。

6) 特定文件系统的指针　指向本文件系统所需信息的指针。

2. 文件的分类

Linux 的文件除了正规文件、目录文件以外，还有其他几种文件类型。
包括：

普通文件
目录文件
硬链接文件
符号链接文件
套接字文件
有名管道文件
字符设备文件
块设备文件

其中：

普通文件　是分为文本文件和二进制文件，用户可以自己处理数据的逻辑边界。用 ls -l 列出其属性，最高位用字符“-”表示。

目录文件　是由目录列表组成的有结构的记录式文件。用 ls -l 列出其属性，最高位用字符“d”表示。

硬链接文件　用 ln 命令可以产生硬链接文件。用 ls -l 列出其属性，最高位用字符“l”表示。硬连接文件共享 i 节点，不能跨文件系统存在。

符号链接文件 与硬链接文件不同，符号文件指向一个文件，操作系统将它看作一个特殊文件。可以用 ln -s 命令产生。

套接字文件　套接字供 Linux 系统与其他机器联网时使用，一般用在网络端口上。文件系统利用套接字文件进行进程间通信。用 ls -l 列出文件列表时，套接字文件权限前第一个字母为 s。

有名管道文件　有名管道文件也是通过文件系统进行通信的一种方法。命令 mknod 可用来创建一个有名管道。用 ls -l 列出文件列表时，有名管道文件权限前第一个字母为 p。

字符设备文件　设备文件一般存放在/dev 目录中。字符设备文件通过文件系统提供了一种与设备驱动程序通信的方法。每次的通信量是一个字符。用 ls -l 列出文件列表时，字符设备文件权限前第一个字母为 c。每个字符设备还包括两个数值，代表进行通信的主设备和从设备。

块设备文件　块设备文件与字符设备文件一样存放在/dev 目录中，用于同驱动程序通信。块设备也包含主从设备号。进行通信时一次传送一个数据块。用 ls -l 列出文件列表时，块设备文件权限前第一个字母为 b。

3. 文件描述字

当进程成功地打开一个文件时，操作系统通过系统调用返回一个文件描述字给调用它的进程。这个文件描述字是提供给其它函数或系统调用对文件读写的接口。系统内部用文件描述字来识别文件。每个进程至多同时使用 20 个文件描述字（0~19），其中 0、1、2 分别自动产生指定给 stdin、stdout、stderr,因此打开的第一个文件描述字是 3，接下来是 4，以此类推。

4. inode 信息

inode（index node）是 Unix 和 Linux 描述文件信息的数据结构。每一个文件（包括普通文件、目录文件、设备文件、……）都有一个 inode 与之对应。对文件的访问通过 inode 来实现。同时系统还提供了以下一组读取 inode 信息的系统调用。

- **inode** 结构包含以下信息：
  1) 设备信息：文件使用的设备。
  2) 状态（mode）信息：包含文件的类型及访问权限。
  3) 所有者信息：文件所有者的用户 ID、文件所有者所在的组 ID。
  4) 连接信息（link）：指向本 inode 的连接文件的个数。
  5) 文件的大小（size）：文件的字节数。
  6) 时间戳(timestamps)：上次文件被访问的时间（access time）、上次文件被修改的时间（mod time）。
  7) 数据块（datablocks）：含指向文件占用的磁盘数据块的指针。前 12 个是直接指针，后 3 个分别指向一级间接块、二级间接块、三级间接块。

- 可供进程访问的属性

  为了访问 inode，可供进程使用的属性被复制到 stat 结构中。stat 结构在<sys/stat.h>中定义。主要的域有：

```
dev_t       st_dev      /*设备号*/
ino_t       st_ino      /*索引节点号*/
umode_t     st_mode     /*文件类型和权限*/
nlink_t     st_link     /*链接计数*/
uid_t       st_uid      /*文件所有者的标识号 owner ID*/
gid_t       st_gid      /*文件所有者的组标识号 group ID*/
dev_t       st_rdev     /*设备文件的设备号*/
off_t       st_size     /*以字节为单位的文件的容量*/
time_t      st_atime    /*最后一次访问的时间*/
time_t      st_mtime    /*最后一次修改的时间*/
time_t      st_ctime    /*最后一次状态改变的时间*/
long        st_blksize /*文件系统 I/O 首选的块的大小*/
long        st_blocks   /*实际分配的块的个数*/
```

- 有关 **inode** 的系统调用

  系统还提供了以下一组读取 inode 信息的系统调用。stat 结构中的信息可以利用三个系统调用 fstat()、stat()或 lstat()之一来访问。

  函数调用包含文件：

```
#include  <sys/ stat.h >
#include  <sys/ types.h >
```

  函数调用原形：

```
int fstat(int fd, struct stat *sbuf);
```

  或

```
int stat(char *pathname , stat *sbuf);
```

或

```
int  lstat(char *pathname , stat *sbuf);
```

其中：

fd	指向打开文件描述信息的文件描述符。
Pathname	文件的路径名。
Sbuf	是 stat 结构的指针。

正确返回参数：0；

错误返回参数：-1；

说明：

如果第一个参数是文件的路径名而不是文件描述符，则 stat()调用 和 fstat()调用是相同的。

如果文件是一个符号链接，lstat()调用将给出链接文件自身的状态细节，stat()调用将依照链接给出所指向的文件的信息。

注：系统调用实例，可参见附录 A 中有关“获得 inode 信息实验”的源程序。

每个文件系统都有最大设备号和最小设备号。设备号的数据类型是 dev_t 。系统提供了两个宏 major 和 mijor 可以访问设备文件的最大设备号和最小设备号。

inode 结构中的设备参数除了 st_dev 还有 st_rdev。其中 st_dev 值对系统中每个文件名来说是指含有该文件名和对应的 inode 的文件系统的设备号；而 st_rdev 这个值只有字符设备文件和块设备文件才有，它表示了实际设备的设备号。

注：系统调用及宏操作实例，可参见附录 A 中有关“设备文件的设备号信息”的源程序。

5. 文件操作的系统调用

（1） 打开文件的系统调用 open()

打开文件和创建文件的方法是 open()。

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
```

函数调用原形：

```
int open(const char *pathname, int flag);
```

或

```
int open(const char *pathname, int flag,mode_t      mode);
```

其中：

pathname	是待打开或创建的文件名。
Flag	规定如何打开文件。包括：
O_RDONLY	只读方式打开。
O_WRONLY	只写方式打开。
O_RDWR	读写方式打开。
O_APPEND	写文件时赋加在文件末尾。
O_CREAT	若文件不存在，则创建该文件。
O_EXCL	若使用 O_CREAT 时，文件已存在，则产生产出错。
O_TRUNC	若文件已存在，写入之前先删除原有数据。
Mode	用于 flag 为 O_CREAT 时指定新文件的所有者、文件的用户组以及系

统中其他用户的访问权限位。包括：

S_IRUSR　　文件所有者的读权限位
S_IWUSR　　文件所有者的写权限位
S_IXUSR　　文件所有者的执行权限位
S_IRGRP　　文件用户组的读权限位
S_IWGRP　　文件用户组的写权限位
S_IXGRP　　文件用户组的执行权限位
S_IROTH　　文件其他用户的读权限位
S_IWOTH　　文件其他用户的写权限位
S_IXOTH　　文件其他用户的执行权限位

访问权限位按位逻辑加组合：

S_IRWXU　　定义为(S_IRUSR|S_IWUSR|S_IXUSR)
S_IRWXG　　定义为(S_IRGRP|S_IWGRP|S_IXGRP)
S_IRWXO　　定义为(S_IROTH|S_IWOTH|S_IXOTH)

可以用以下常量值设置 set_uid 和 set_gid：

S_ISUID　　设置 set_uid
S_ISGID　　设置 set_gid

正确返回参数：文件描述符；
错误返回参数：-1；
例：

```
open(“afile”,O_RDONLY);                    /* 以只读方式打开文件 afile */
open(“bfile”,O_RDWR | O_TRUNG);     /* 以读写方式打开文件 bfile，长度置为
                                          0 */
open(“cfile”,O_WRONLY | O_CREAT | O_EXCL,S_IRWXU | S_IXGRP);
     /* 创建文件 cfile 用于写操作，其权限设置为 r w x - - x - - - */
```

(2) 检测文件是否访问过　　access()

函数调用包含文件：

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

函数调用原形：

```
int access(char *pathname, int mode);
```

其中：

pathname　　是待检测的文件名。
Mode　　是包含在 unistd.h 文件中的值之一：

R_OK　　检测调用进程是否有过读操作
W_OK　　检测调用进程是否有过写操作
X_OK　　检测调用进程是否有过执行操作
F_OK　　检测指定的文件是否存在

正确返回参数：1；
错误返回参数：0；

(3) 创建新文件的系统调用

creat()

新文件可以用 creat()创建。
函数调用包含文件：

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

函数调用原形：

int creat(const char *pathname,mode_t mode);

该函数与下面函数等价：

open(pathname,O_WRONLY|O_CREAT|O_TRUNC,mode);

返回：如果创建正确则返回文件描述符，文件的打开方式为“只写”；否则返回 -1 。

说明：创建文件只用“只写”方式打开。如果创建的文件既要写，又要读，则必须连续使用 creat、close、open。也可以用以下 open 函数：

open(pathname,O_RDWR|O_CREAT|O_TRUNC,mode);

（4） 读文件的系统调用 read()

一旦用 O_RDONLY 或 O_RDWR 方式打开或建立了文件，就可以用 read()系统调用从该文件读取字节了。

函数调用包含文件：

#include <unistd.h>

函数调用原形：

ssize_t read(int fd ,void *buf,size_t nbytes);

正确返回参数：0 或字节数；

错误返回参数：-1；

其中：

fd	是想要读的文件的文件描述符。
Buf	是指向内存的指针。read()将从文件中读取的字节存放到这个内存块
Nbyte	从该文件复制到内存的字节个数。

说明：read()操作从文件的当前位置开始，该位置由包含在相应的文件描述中的偏移值（offset）给出。每一次读操作结束，偏移值被加上刚复制的字节数，以便下一次的 read()操作。

（5） 写文件的系统调用 write()

用 write()系统调用，可以将数据写到文件中。

函数调用包含文件：

#include<unistd.h>

函数调用原形：

ssize_t write(int fd , const void *buf , size_t nbytes);

正确返回参数：0 或字节数；

错误返回参数：-1；

其中：

fd	是想要写的文件的文件描述符。
buf	是指向内存的指针。write()从 buf 所指的内存块中将 nbytes 个字节写到 fd 所指示的文件中。
Nbytes	从内存复制到文件的字节个数。

说明：

每次写操作，都是将字节写到文件描述中偏移值指示的位置。写操作结束偏移值自动地加上实际写入的字节数，为下一次 write()做准备。

如果文件是用 O_APPEND 方式打开的，则 write()作用之前偏移值会自动移到

文件结束位置。

（6） 关闭文件的系统调用 close()

用 close()释放打开的文件描述符。

函数调用包含文件：

#include <unistd.h>

函数调用原形：

int close(int fd);

其中：

fd 为文件描述符。

正确返回参数：0；

错误返回参数：-1；

每次打开一个文件时，文件描述中的引用计数加一。用 close()关闭该文件时，文件描述中的引用计数减一。直到某次调用 close()时，使引用计数值为 0 了，系统才释放文件描述符及描述信息。

（7） 设置当前读写位置 lseek()

Linuxx 系统中可用 lseek()将当前文件的偏移值移到相关的位置,设置下一次 read()或 write()访问的位置。

函数调用包含文件：

#include <sys/types.h>

#include <unistdt.h>

函数调用原形：

off_t lseek(int fd, off_t offset, int whence);

其中：

fd	为文件描述符。
Offset	是一个相对值，被加到基地址上，给出新的偏移值。
Whence	指示偏移值的计算点。可以选择下列三个值之一：
	SEEK_SET 从文件的开始处计算偏移值。
	SEEK_CUR 从文件的当前偏移处开始处计算偏移值。
	SEEK_END 从文件的结束处计算偏移值。

6. 文件共享

Linux 系统中支持不同进程共享打开的文件。内核中使用三种数据结构描述打开的文件以及文件的共享关系。

1)进程打开文件表

每个进程的进程表中都有一个打开文件表，该表的每一个表项记录了一个进程打开的文件。每个进程最多可打开 20 个文件。进程打开文件表的表项主要包括：

文件描述符标志

指向系统打开文件表中相应表项的指针

2)系统打开文件表

系统打开文件表记录系统已打开的文件，每个表项对应了一个打开的文件，主要包括：

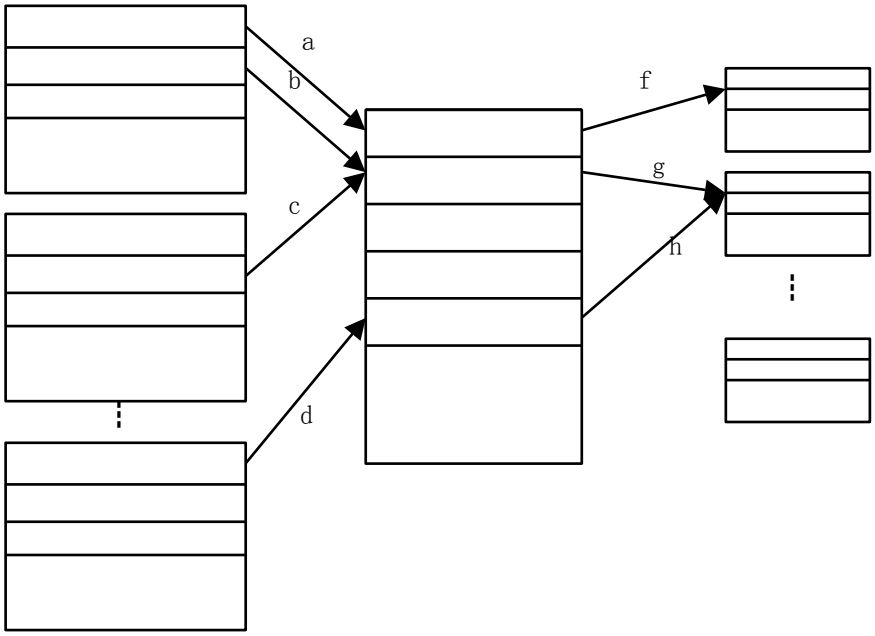
文件打开状态标志

文件当前偏移地址

打开文件的内存 vnode 表入口

3)内存中的索引节点表 vnode

是 Linux 中每一个打开的文件在内存中的索引节点表。主要包含了该文件的 inode 信息。当文件打开时，该 inode 信息从磁盘中读出，以便文件的信息可以被进程使用。



图中：•a、b 为同一个进程打开的不同文件，指向系统打开文件表中的各自的表项。

•d 和 b、c 为不同进程打开的文件，指向系统打开文件表中的各自的表项。

•b、c 为父进程和子进程的文件共享，或通过 dup()系统调用产生的文件共享，所以指向系统打开文件中表中的同一个表项。用系统打开文件表项的共享计数指示共享进程数。

•f、g、h 是系统打开文件表中的不同表项指向 vnode 的指针。其中 g、h 是不同进程以同一文件名或不同文件名方式打开的指向同一个文件的指针，用 vnode 中的共享计数指示。每关闭一个文件，计数减一，直到计数为零，将内存 inode 复制回磁盘。

（1） 复制文件描述字的系统调用 dup()

一个文件的描述符可以利用以下方式复制，所得到的文件描述符可以传递给其他进程从而达到文件的共享。

函数调用包含文件：

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

函数调用原形：

```
int dup(int fd);
```

或 int dup2(int fd,int fd2);

返回参数：最小的可用文件描述符。

另一个复制文件描述符的方式是 fcntl()系统调用。

dup(fd)等同于系统调用

fcntl(fd,F_DUPFD,0)

dup2(fd,fd2) 等同于系统调用

```
close(fd2);
fcntl(fd,F_DUPFD,fd2);
```

(2) 链接文件的系统调用 link()

当文件被首次创建时，从给定的路径名自动地产生一个文件的链接。所以文件的一个目录项称为一个链接（link）。此后，对这个文件的硬链接可以通过 link()系统调用产生。

每建立一个文件链接，文件的索引节点(inode)中的链接计数加一。每删除一个链接文件，链接计数减一，直到计数为零，系统回收该索引节点(inode)以及分配给该文件的磁盘数据块。

函数调用包含文件：

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

函数调用原形：

```
int link(char *origfile,char *linkname);
```

正确返回参数：0

错误返回参数：-1

其中:

origfile 是已有的文件

linkname 是新创建的目录链接

说明：

要建立硬链接文件，用户必须对 origfile 有读的权限，对 linkname 所在的目录有写和执行的权限。只有超级用户有权创建对目录的硬链接。

(3) 移去链接文件的系统调用 unlink()

这个系统调用移去指定的文件硬链接并将它的索引节点的链接计数减一。如果索引节点的链接计数已经变成零，则索引节点和文件数据块全部释放。

执行此操作，必须具有对含有该文件目录的目录的可写和可执行权限。

函数调用包含文件：

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

函数调用原形：

```
int unlink(char *pathname);
```

正确返回参数：0

错误返回参数：-1

(4) 建立符号文件的系统调用 symlink()

符号链接比硬连接更灵活。它可以链接位于另一个文件系统的文件，几乎所有与文件相关的系统调用，都会自动引用符号链接所指向的文件，除了 chown(),lstat(),readlink(),rename(),unlink()。

- 系统调用 symlink()可以用来建立符号链接。

函数调用包含文件：

#include <unistd.h>

函数调用原形：

int symlink(const char *origfile,const char *linkname);

说明：

调用成功 linkname 将称为指向 origfile 的符号链接。

- 要确定符号链接文件所指向的目标文件可以使用 readlink()系统调用。

函数调用包含文件：

#include <unistd.h>

函数调用原形：

int readlink(const char *linkname,const char *buf,size_t bufSize);

说明：

函数会在 buf 指向的缓冲区返回符号链接文件所指向的目标文件。成功时函数返回文件名的长度，失败时返回 -1。

7. 创建设备文件和管道的系统调用

（1） 创建特殊文件 mknode()

mknode()函数可以用来，包括有名管道、字符设备、块设备等。

函数调用包含文件：

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<unistd.h>

函数调用原形：

int mknode(const char *pathname,mode_t mode,dev_t dev);

其中：

pathname 指定该设备的名字。 (

Mode 指定文件类型和存取权限。文件类型可以是，S_IFIFO(管道文件),S_IFBLK（块文件）,S_IFCHR（字符设备）。

Dev 指示新创建的设备的主设备号和从设备号。 若是 S_IFIFO 文件次项参数不用）

（2） 用来创建无名管道 Pipe()函数

无名管道用于父进程和子进程或兄弟进程之间的通信。

函数调用包含文件：

#include<unistd.h>

函数调用原形：

int pipe(int fds[2]);

说明：

该系统调用在 fds 中返回两个文件描述符，fds[0]用于只读，fd[1]用于只写。

8. 文件目录结构

（1） 获得当前目录 getcwd()、**get_current_dir_name()**、**getwd()**

函数调用包含文件：

#include<unistd.h>

函数调用原形：

char *getcwd(char *buf,size_t size);

或 char *get_current_dir_name(void);

或 char *getwd(char *buf);

说明：函数 getcwd()把当前目录的绝对路径名拷贝到 buf 指示的缓冲区中。Buf 的大小为 size 个字节。当 buf 为 NULL 时，getcwd（）会调用 malloc()分配所需的缓冲区。

程序中应负责用 free()将缓冲区释放

（2） 设置当前目录 chdir()、fchdir()

函数调用包含文件：

#include<unistd.h>

函数调用原形：

int chdir(const char *path);

或 int fchdir(int fd);

其中：

path	希望成为当前目录的目录
fd	文件描述符

成功返回:0

错误返回:		
	ENOTDIR	所指定的路径中有错
	EACCESS	对目录没有执行权
	EBADF	fd 不是有效的文件描述符

（3） 改变根目录 chroot()

系统只有一个根目录,但是每个进程可以有自己的根目录。这样可以防止一些不安全的进程存取整个文件系统。改变了根目录，并不改变进程的当前目录。进程仍可以通过相对当前目录的路径存取其它目录的文件。

函数调用包含文件：

#include<unistd.h>

函数调用原形：

int chroot(const char *path);

其中:

path　　是作为根目录的路径

(4) 创建目录　　mkdir()

函数调用包含文件:

#include<sys/stat.h>

#include<sys/types.h>

#include<fcntl.h>

#include<unistd.h>

函数调用原形:

int mkdir(const char *dirname,mode_t mode);

其中:

dirname　　指定的要创建的目录名。

ode　　该目录的存取权限。最终的目录还受到 umask 的影响,即 mode & umask。

说明:如果 dirname 已经存在或 dirname 中某部分无效。函数返回-1。

新创建的目录的 uid 是进程的有效 uid。gid 的设置或者按系统安装的父目录的 gid,或者进程的有效 gid 将成为新目录的 gid。

(5) 删除目录　　rmdir()

函数调用包含文件:

#include<unistd.h>

函数调用原形:

int rmdir(const char *dirname);

其中:

dirname　　指定的要删除的目录名。

说明:只有空目录才能被删除,否则 rmdir()将返回 -1,同时 errno 等于 ENOTEMPTY。

(6) 目录访问　　opendir()、readdir()、rewindir()、closedir()

应用程序经常需要了解包含在一个目录中的文件的信息。 Linux 可以同时支持多种文件系统。为了使应用程序不涉及具体文件系统的目录格式,Linux 提供了一组系统调用函数帮助应用程序按照一种抽象方式处理目录。

← 为了打开一个目录,可以使用 opendir()系统调用。

函数调用包含文件:

#include<sys/types.h>

#include<diret.h>

函数调用原形:

DIR *opendir(const char *pathname);

正确返回:目录指针。

错误返回:0;

这个系统调用打开指定的目录。如果成功，返回一个目录指针。该目录指针被传递给有关的系 readdir()统调用 readdir()和 closedir()。

← readdir()系统调用返回一个指向 direct 结构的指针。该结构含有指定目录下的一个文件链接的内容，inode 号和文件名。重复调用 readdir()可以在目录中顺序访问所有的链接，直到最后返回一个 0 值。

函数调用包含文件：

```
#include<sys/types.h>
#include<direct.h>
```

函数调用原形：

```
struct dirent *readdir(DIR *dp);
```

正确返回：direct 的指针。

错误返回：0；

dirent 结构如下：

```
struct dirent {
ino_t    d_ino;       /*inode 节点号*/
char     d_name[NAME_MAX+1] ;          /*文件名*/
    }
```

← 要从头读取打开的目录的内容，可以使用 rewindir()系统调用。它将复位 direct 结构的数据，这样下次再用 readdir()将返回该目录的第一项。

← closedir()系统调用关闭打开的目录。

函数调用包含文件：

```
#include<sys/types.h>
#include<direct.h>
```

函数调用原形：

```
int closedir(DIR *dp);
```

正确返回：0；

错误返回：-1；

9. 改变文件属性

（1） 改变文件名称　　rename()

一个文件或目录可以由 rename()系统调用修改名称。

函数调用包含文件：

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

函数调用原形：

```
#include<stdio.h>
int rename(const char *oldname,const char *newname) ;
```

正确返回：0；

错误返回：-1；

说明：用户应对包含 oldname 和 newname 的目录具有可写的权限和可执行的权限。

（2） 改变文件读取权限 chmode()、fchmode()

用于改变文件权限位的系统调用 chmode()、fchmode()可以改变文件所有者、用户组和其它人的读、写和执行位。

chmod（）用来改变给定的 pathname 的文件权限位，而 fchmod()用来改变给定的 fd 的文件权限位。

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/stat.h>
```

函数调用原形：

```
int chmod(char *pathname,mode_t mode) ;
```

或

```
int fchmod(int fd,mode_t mode) ;
```

正确返回：0；

错误返回：-1；

如： fchmod(fd,S_ISUID|S_IRWXU|S_IXGRP|S_IXOTH);

等同于 fchmod(fd,04711);

文件和目录的默认模式是由 umask 来设置的，umask 用数字定义其值。用户进入系统时，该值被放在文件系统中，创建的文件若未指定模式，就用 umask 来自动设置其许可。当改变一个文件的权限位时，所规定的位，将自动地由当前的 umask 的值按以下公式加以修改。

Mode & (~umask)

Umask 的值也可以用 umask()设置。

函数调用包含文件：

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

函数调用原形：

```
mode_t umask(mode_t mask);
```

正确返回：0；

错误返回：-1；

这个函数用新的 mask 的值代替 umask 的当前值。

（3） 改变文件的所有者 **chown()、fchown()**

chown()和 fchown()系统调用用来改变文件所有者识别号或它的用户组识别号。

函数调用包含文件：

```
#include<sys/types.h>
#include<unistd.h>
```

函数调用原形：

int fchown(int fd,uid_t owner,gid_t group) ;
int chown(const char *pathname,uid_t owner,gid_t group) ;

正确返回：0；

错误返回：-1；

fchown()给 fd 所指定的文件描述符赋予新的所有者识别号和组识别号。chown()给路径名为 pathname 的文件赋予新的所有者识别号和组识别号。

Linux 中只有 root 帐号可以使用 chown()和 fchown()系统调用。

（4） 改变或设置打开文件的属性 fcntl()

fcntl()系统调用用于更改正在使用中的文件的属性，如复制文件描述字、设定 close_on_exec 标志等。

函数调用包含文件：

#include<unistd.h>
#include<fcntl.h>

函数调用原形：

int fcntl(int fd,int cmd) ;
或 int fcntl(int fd,int cmd,long arg) ;
或 int fcntl(int fd,int cmd,struct flock *lock) ;

常用第二种格式。

其中：

fd 文件的述字。

cmd 为命令,共 5 种格式

1) F_DUPFD 拷贝 fd 到 arg，如果需要首先关闭 fd。
2) F_GETFD 读取并返回 close_on_exec 标志。如果低位字节为 0，执行 exec()之后，fd 指向的文件仍保持在打开状态；反之，在执行 exec()之后，该文件会自动关闭。
3) F_SETFD 设置 close_on_exec 标志为 arg 指定的值。
4) F_GETFL 读取并返回已打开文件的标志状态。
5) F_SETFL 设置由 arg 指定的标志。可设置的标志有 O_APPEND、O_NOBLOCK 和 O_ASYNC。标志设置通过按位 OR 来进行，因此其它标志不受影响。

第三部分　标准输入输出

标准输入输出的系统调用都存在于标准输入输出库中。标准输入输出库是 C 语言编写的，所以不光应用于 Linux，也可以应用于其它系统。而基于文件系统的输入输出与操作系统关系很大，不能应用于其它系统。基于文件系统的输入输出都是围绕文件描述符展开的，打开一个文件，返回的是文件描述符，并被应用于输入输出操作。而标准输入输出是围绕“流”（stream）进行的，当调用标准输入输出创建一个文件时，就将一个“流”和文件关联在一起了。

1. 标准输入输出的基本操作

在 Linux 系统中，文件和设备都被认为是数据流。在对流进行操作前，需要将它打开。操作完成后，需要通过操作系统清空缓冲区、保存数据。最后应关闭“流”。

只有一个流与某文件或设备关联起来，才可以对这个流进行各种操作，这就是流的打开操作。打开操作成功的话，系统将返回一个 FILE 结构的指针，以后的操作都可以借助这个指针，通过调用系统函数来完成了。

执行程序时自动打开了三个流，它们是标准输入、标准输出和标准错误输出。相应的 FILE 结构指针为 stdin、stdout、stderr。它们和 STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO 文件描述符对应的文件是相同的。

（1） 流的打开操作　　fopen()、freopen()、fdopen()

以下三个系统调用都可以打开一个标准输入输出流。

函数调用包含文件：

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

函数调用原形：

```
FILE *fopen(const char *pathname,const char *type);
```

或 `FILE *freopen(const char *pathname,const char *type,FILE *fp);`

或 `FILE *fdopen(int fildes,const char *type);`

正确返回：文件结构的指针；

错误返回：空指针；

这三个系统调用的不同之处在于：

fopen 打开一个特定的文件，文件名由 pathname 指出。

freopen 在特定的流上打开一个特定的文件。它先关闭 FILE *fp 指定的已打开的流，再打开 pathname 指定的流。一般用于打开特定文件代替标准输入流、标准输出流、标准错误输出流。

fdopen 将一个流对应到某个已打开的文件上，fildes 就是这个文件的描述符。已打开文件的模式应与 type 定义的模式相同。这个已打开文件一般是管道文件或网络通信管道，这些文件无法通过标准输入输出函数打开，只能通过特定设备函数得到文件描述符，再用 fdopen 函数将一个流与这个文件关联起来。

三个系统调用中的 char *type 表示了流的打开模式：

type	读打开	写打开	文件长度截为零	新建文件	开始读写位置
“r”或“rb”	是	否	否	否	文件开头
“r+”或“r+b”或“rb+”	是	是	否	否	文件开头
“w”或“wb”	否	是	是	是	文件开头
“w+”或“w+b”或“wb+”	是	是	是	是	文件开头
“a”或“ab”	否	是	否	是	文件末尾
“a+”或“a+b”或“ab+”	是	是	否	是	文件末尾

其中“r”为允许读操作、“w”为允许写操作、“a”为允许从文件尾继续写操作、“b”为允许标准输入输出系统区分文本文件和二进制文件。

（2） 流的清洗操作　　fflush()、fpurge()

流的清洗操作是指将输入输出缓冲区的内容写入一个文件或刷新缓冲区。以下两个系统调用都可以实现清洗。

函数调用包含文件：

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

函数调用原形：

```
int fflush(FILE *fp);
```

或 `int fpurge(FILE *fp);`

正确返回：0；

错误返回：EOF；

其中 FILE *fp 就是已经打开的流。它们的区别是：

fflush 将缓冲区内容保存在磁盘上，并清空缓冲区。

fpurge 将缓冲区中所有数据清除掉。

（3） 流的关闭操作　　**fclose()**

用户在应用程序打开的流必须在程序结束时关闭。否则可能造成数据丢失。

流的关闭操作用以下系统调用：

函数调用包含文件：

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

函数调用原形：

```
int fclose(FILE *fp);
```

正确返回：0；

错误返回：EOF；

（4） 流缓冲区属性的设定　　setbuf()、setbuffer()、setlinebuf()、setvbuf()

流缓冲区的属性包括大小和类型。缓冲区类型有以下三种：

全缓冲：缓冲区满时才真正执行输入输出，即将缓冲区内容保存到磁盘文件或输出到终端上。

行缓冲：缓冲区中输入一个换行符时，才真正执行输入输出。这种缓冲允许一次一个字符地在终端上输出。一般与终端设备相连的缓冲采用这种方式。

无缓冲：一接受到字符，就执行输入输出。错误标准输出都采用这种方式。

用户打开流时系统就为其设置了默认的缓冲属性。用户也可以利用以下系统函数设定自己想要的缓冲属性。

函数调用包含文件：

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

函数调用原形：

```
int setbuf(FILE *fp,char *buf);
```

或 int setbuffer(FILE *fP,char *buf,size_t size);
或 int setlinebuf(FILE *fp);
或 int setvbuf(FILE *fp,char *buf,int mode,size_t size);
正确返回:0;
错误返回:非 0;
其中：

FILE *fP	已打开的流。
char *buf	用户自己设定的缓冲区。
size_t size	缓冲区大小。
int mode	流德类型，可以取_IOFBF、_IOLBT、_IONBF 分别代表全缓冲、行缓冲和无缓冲。

说明： 调用这些函数时，必须先打开流，而且最好还没有进行其它操作。因为其它操作是与缓冲区的性质密切相关的。

(5) 流的定位操作 ftell()、fseek()、rewind()、fgetpos()、fsetpos()

用户打开流以后，每次读写操作，读写指针都会移动一个单元。对于块设备，用户可以通过系统调用了解指针所处的位置，并可以将指针移到任何一个位置。完成流定位的系统调用有以下几个。

← 对于二进制文件，读写指针从文件开头，以字节为单位开始计数。通过 ftell()函数可以得到这个值。

函数调用包含文件：

#include<stdio.h>

函数调用原形：

long ftell(FILE *fp);

正确返回:读写指针的位置;
错误返回:-1;

← fseek()函数用来定位二进制文件。但必须指定偏移量的大小 offset 和偏移量的使用方法 whence。Whence 可以取值 SEEK_SET（从文件头开始计算）、SEEK_CUR(从文件当前位置计算)、SEEK_END(从文件结尾开始计算)。

函数调用包含文件：

#include<stdio.h>

函数调用原形：

long fseek(FILE *fp,long offset,int whence);

正确返回:0;
错误返回:非零;

← rewind()函数用来将读写指针移到文件的开头。系统默认这个系统调用总是成功的，所以没有返回参数。

函数调用包含文件：

#include<stdio.h>

函数调用原形：

```
void rewind(FILE *fp);
```

无返回值；

← 用户可以使用抽象的数据结构 fpos_t 来存放读写指针的位置。fgetpos()可以得到读写指针的位置。

函数调用包含文件：

#include<stdio.h>

函数调用原形：

```
int fgetpos(FILE *fp,fops_t *pos);
```

正确返回:0;

错误返回:非零;

← fsetpos（）可以定位读写指针的位置。

函数调用包含文件：

#include<stdio.h>

函数调用原形：

```
int fsetpos(FILE *fp,const fops_t *pos);
```

正确返回:0;

错误返回:非零;

2. 非格式化输入输出操作

非格式化输入输出分为：字符型输入输出、一行型输入输出、直接型输入输出。

字符型输入输出操作

（1） 字符型输入　　getc()、fgetc()、getchar()

字符型输入输出每次可以读写一个字符。标准输入输出函数会自动处理缓冲区问题。

字符型输入的操作有以下三个函数调用：

函数调用包含文件：

#include<stdio.h>

函数调用原形：

```
int getc(FILE *fp);
```

或 `int fgetc(FILE *fp);`

或 `int getchar(void);`

文件末尾或出错返回：EOF（C 语言中值为 －1）；

其它情况返回：下一个要读入的字符；

说明：

getchar（）的作用相当于调用函数 getc(stdin),将标准输入流作为参数传进去。

函数 getc 可以当作宏，而 fgetc 只能作为普通函数调用。

（2） 字符型输出 putc()、fputc()、putchar()

字符型输出的操作与输入函数是对应的，有以下三个函数调用：

函数调用包含文件：

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

函数调用原形：

```
int putc(int c,FILE *fp);
```

或 `int fputc(int c,FILE *fp);`

或 `int putchar(int c);`

正确返回：字符 c；

错误返回：EOF;

说明：

putchar（c）的作用相当于调用函数 putc(c,stdout),将标准输出作为参数传进去。

函数 putc 可以当作宏，而 fputc 只能作为普通函数调用。

（3） 判断文件尾和文件出错 ferror()、 feof()

函数调用包含文件：

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

函数调用原形：

```
int ferror(FILE *fp);
int feof(FILE *fp);
```

正确返回：非零；

错误返回：0;

（4） 将字符推回缓冲区 ungetc()

函数调用包含文件：

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

函数调用原形：

```
int ungetc(int c,FILE *fp);
```

正确返回：字符 c；

错误返回：EOF;

一行型输入输出操作

使用这种方式，每次可以读写一行。系统提高如下两个函数处理一行的输入。

（1） 一行型输入 fgets()、gets()

函数调用包含文件：

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

函数调用原形：

```
char* fgets(char *buf,int n,FILE *fp);
char* gets(char *buf);
```

其中：

buf　　接受输入的缓冲区的地址

fp　　流结构的指针

正确返回：buf；

错误返回：空指针；

说明：

fgets 需要明确接受的字节数 n。读完一行，将这一行的内容包括行结尾标志，一起保存在 buf 中，用空字符 0 来结束。因此，这一行内容包括行结尾在内，不能超过 n-1 个。如果超过 n-1 个字符，系统接受这一行的一部分，其余在下次调用 fgets 时读入。

gets 是从标准输入设备输入的。

（2） 一行型输出　　fputs()、puts()

函数调用包含文件：

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

函数调用原形：

```
char* fputs(const char *str,FILE *fp);
```

或 char* puts(const char *str);

其中：

char *str　　要输出的字符串

FILE *fp　　流结构的指针

正确返回：非负数；

错误返回：EOF；

说明：

fputs 把以字符 0 结尾的字符串输出到指定的流文件中，但结尾的字符 0 并不输出。

puts 把字符 0 结尾的字符串输出标准输出设备上，结尾的字符 0 也不输出，最后加一个换行符。

例:使用行输入输出的程序。打开命令行指定的流文件（命令行参数用 argv[i]表示,其中 i 为参数序号），从文件中输入一行，在屏幕上输出一行。

```
#include<sys/types.h>
#include<stdio.h>
int main(int argc,char *argv[])
```

```
{
    char s[1024];
    FILE *fp;
    if((fp=fopen(argv[1],"r"))!=(FILE*)0)      /*打开流文件，判是否错*/
    {
        while((fgets(s,1024,fp))!=(char )0) /*从文件输入一行*/
        puts(s);                                    /*再在屏幕输出一行*/
    }
    else
    {
        fprintf(stderr,"file open error.\n");
        exit(1);
    }
    exit(0);
}
```

直接型输入输出操作

字符型输入输出操作用在二进制文件的处理中。输入输出以结构为单位，一次可以处理若干个记录。

(1) 直接型输入操作　　fread ()

函数调用包含文件：

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

函数调用原形：

```
size_t fread(void *ptr,size_t size,size_t nobj,FILE *fp);
```

其中：

```
void *ptr    指向若干结构的指针
size_t siz 结构大小（一般可用 sizeof()获得）
nobj             要处理的结构的个数
FILE *fp         流结构的指针
```

正确返回：实际读出的数目

(2) 直接型输出操作　　fwrite ()

函数调用包含文件：

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

函数调用原形：

```
size_t fwrite(void *ptr,size_t size,size_t nobj,FILE *fp);
```

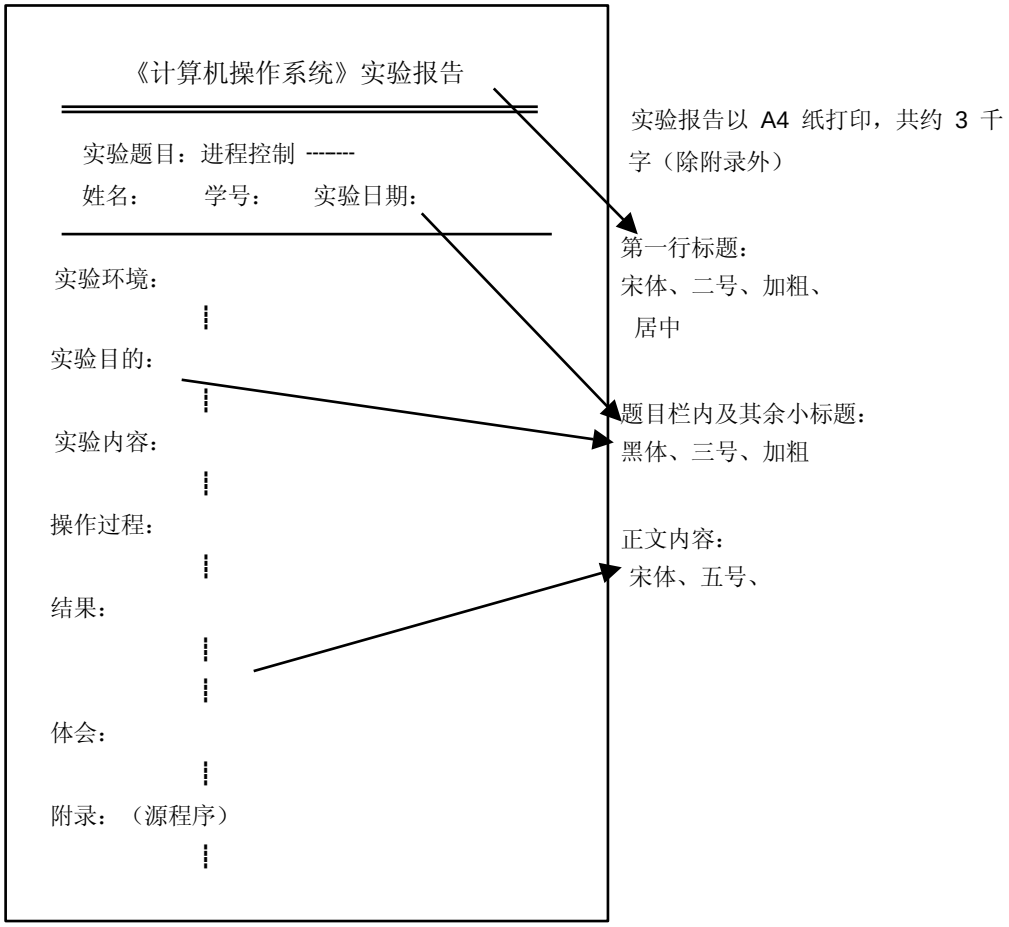
其中：

```
void *ptr    指向若干结构的指针
size_t siz 结构大小（一般可用 sizeof()获得）
```

nobj	要处理的结构的个数
FILE *fp	流结构的指针

正确返回：实际写入的数目。

附录 A 实验报告格式


《计算机操作系统》实验报告
实验题目：进程控制 -------
姓名： 学号： 实验日期：
实验环境：
实验目的：
实验内容：
操作过程：
结果：
体会：
附录：（源程序）
实验报告以 A4 纸打印，共约 3 千字（除附录外）
第一行标题：
宋体、二号、加粗、
居中
题目栏内及其余小标题：
黑体、三号、加粗
正文内容：
宋体、五号、

附录 B 参考资料

计算机操作系统教（第三版）	张尧学、史美林	清华大学出版社
操作系统教程—原理和实例分析	孟静	高等教育出版社
现代操作系统（第三版）	塔嫩鲍姆 著、陈向群等译	机械工业出版社
计算机操作系统教程	陆松年	电子工业出版
Red Hat Linux 大全	David Pitts 等姚彦忠、赵小杰等译	机械工业出版社
操作系统及实验教程	李善平、郑扣根	机械工业出版社社
Linux 上的 C 编程	怀石工作室	中国电力出版社
Linux 管理员指南	何田、宋建平、李舒、阎瑞雪	清华大学出版社
Linux 编程指南	徐严明等	科学出版社
Linux C 高级程序员指南	毛曙福	国防工业出版社
Linux 技术内幕	Moshe Bar 著、王超译	清华大学出版社
Linux 基础及应用习题分析与实验指导	谢蓉	中国铁道出版社