**课程设计大纲**

|  |  |  |  |
| --- | --- | --- | --- |
| 课程编号 |  | 适用专业 | **计算机** |
| 课程名称 | **操作系统课程设计** | 课程性质 | **必修** |
| 英文名称 | **The course design of system software** | 学时/学分 | **64/2** |
| 选用教材 | [计算机操作系统(第3版)](http://www.amazon.cn/%E6%96%B0%E4%B8%96%E7%BA%AA%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%B1%BB%E6%9C%AC%E7%A7%91%E8%A7%84%E5%88%92%E6%95%99%E6%9D%90-%E8%AE%A1%E7%AE%97%E6%9C%BA%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F-%E6%B1%A4%E5%B0%8F%E4%B8%B9/dp/B005G0WYU8/ref=sr_1_2?ie=UTF8&qid=1394506563&sr=8-2&keywords=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 汤小丹、梁红兵、哲凤屏、 汤子瀛 西安电子科技大学出版社 | 考核方式 | **上机+课程报告** |
| 先修课程 | **操作系统** | 执笔人 | **叶琪** |

1. 教学基本目标

本设计的目的是实现操作系统和相关系统软件的设计，其中涉及进程编程、I/O操作、存储管理、文件系统等操作系统概念。

1. 教学基本内容

1．Shell编程：掌握Linux O.S.环境，掌握基本环境等

2．Linux系统调用：掌握系统调用的执行流程，增加一个新的系统调用。

3．Linux二级文件系统：编写程序实现模拟Linux文件系统

4. Linux设备驱动程序：编写程序实现Linux设备驱动程序

1. 教学进度安排
2. 课程设计要求说明 4学时
3. 软件设计 16学时
4. 上机编程 32学时
5. 调试 4学时
6. 检查考核 8学时
7. 课程设计报告要求

1．对进行认真分析，列出实验具体步骤，写出符合题目要求的程序清单，准备出调试程序使用的数据。

2．以完整的作业包的形式提交原始代码、设计文档和可运行程序。提交的软件应当包括：设计题目，程序清单，运行结果分析，所选取的算法及其优缺点，以及通过上机取得了哪些经验。程序清单要求格式规范，注意加注释（包含关键字、方法、变量等），在每个模块前加注释，注释不得少于20%。课程设计要求同时上交打印文档，设计报告包括设计题目，算法分析,关键代码及其数据结构说明，运行结果分析以及上机实践的经验总结。

1. 考核方式与成绩评定方法

平时成绩30%+70%系统完成情况

系统完成情况根据以下几方面情况评定：

1.分析与设计能力

2.系统实现能力

3.系统文档以及设计报告的规范性

4.是否在规定时间内完成设计

5.工作态度

通过项目答辩给出具体分数，答辩考核表：

|  |  |
| --- | --- |
| 要求 | 分数 |
| 实验报告与演示内容一致，报告内容不存在抄袭； |  |
| 完成系统设计要求所需的全部功能，程序代码无bug； |  |
| 正确、完整地演示系统，演示结果正确，内容表达清楚； |  |
| 能回答指导教师提出的问题，回答正确； |  |
| 其他 |  |
| 备注： | |
| 总成绩： | |

1. 参考书

1．[操作系统:精髓与设计原理(第7版)(英文版)](http://www.amazon.cn/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F-%E7%B2%BE%E9%AB%93%E4%B8%8E%E8%AE%BE%E8%AE%A1%E5%8E%9F%E7%90%86-%E6%96%AF%E6%89%98%E6%9E%97%E6%96%AF/dp/B00DUFKXB8/ref=sr_1_21?ie=UTF8&qid=1394506563&sr=8-21&keywords=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "_blank) 蒲晓蓉、周瑞、斯托林斯 (William Stallings)、 威廉•斯托林斯 (William Stallings) 电子工业出版社 (2013-07)

2. [计算机操作系统(第3版)](http://www.amazon.cn/%E6%96%B0%E4%B8%96%E7%BA%AA%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%B1%BB%E6%9C%AC%E7%A7%91%E8%A7%84%E5%88%92%E6%95%99%E6%9D%90-%E8%AE%A1%E7%AE%97%E6%9C%BA%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F-%E6%B1%A4%E5%B0%8F%E4%B8%B9/dp/B005G0WYU8/ref=sr_1_2?ie=UTF8&qid=1394506563&sr=8-2&keywords=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 汤小丹、梁红兵、哲凤屏、 汤子瀛 西安电子科技大学出版社 (2007-08)

3.[计算机操作系统教程(第3版)](http://www.amazon.cn/%E6%99%AE%E9%80%9A%E9%AB%98%E7%AD%89%E6%95%99%E8%82%B2-%E5%9B%BD%E5%AE%B6%E7%BA%A7%E8%A7%84%E5%88%92%E6%95%99%E6%9D%90%E2%80%A2%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%B3%BB%E5%88%97%E6%95%99%E6%9D%90-%E8%AE%A1%E7%AE%97%E6%9C%BA%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E6%95%99%E7%A8%8B-%E5%BC%A0%E5%B0%A7%E5%AD%A6/dp/B008KWW5GC/ref=sr_1_16?ie=UTF8&qid=1394506766&sr=8-16&keywords=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 张尧学、史美林、 张高 清华大学出版社 (2006-10)

**实验一 Shell编程**

**一、实验目的**

学习如何编写一个Linux的外壳——Shell，即命令解释程序。学习怎样接收命令、解释命令、执行命令，特别是采用创建子进程的方式来执行一个程序，以及父进程如何继续子进程的工作。

**二、实验内容**

1、编写一个C语言程序作为Linux内核的shell命令行解释程序，所执行的结果需和系统命令行方式保持一致，从而使学生了解系统是怎样进行命令的解析和执行的。同时，还提供认识进程创建、进程等待、进程执行、前/后台执行、管道和I/O重定向含义的机会。

基本运行方式如下:

COMMAND命令 OPTIONS选项 ARGUMENTS参数 其中，

* CMMMAND用户输入的一命令：请求内核将某个二进制程序运行为一个进程；
* OPTIONS：指定命令的运行特性；
* ARGUMENTS：命令的作用参数，命令可同时带多个参数，多个之间以空白字符分隔；

例如：ls -ld /var /etc

shell应该解析命令行参数指针数组argv[argc]。使用Linux的系统调用fork()、wait()和execv()等完成。

2、对用户编写的shell增加后台运行功能。即用户可以使用”&”作为一个命令，表示该命令在后台启动。前台还可以处理其他命令。

COMMAND命令 &

3、修改程序，增加I/O重定向功能。即用户可以使用”<”、”>”和”|”符号改变程序/文件的输入和输出。

要求学生首先熟悉Linux的基本命令，命令行方式、参数等，再按要求完成程序功能，编写的shell，符合所使用机器安装Linux系统的shell风格。

此外，根据shell的基本语法和规则，结合编译知识，考虑试着在一个命令行中，同时实现“>、“<”、“|”、“&”等功能，或同时实现多管道、多重定向等功能。

**三、实验提示**

**1、 基本的Shell**

**（1）命令行工作方式**

每个shell都有自己语言的语法和语义。在标准Linux shell中，一个命令行具有如下形式：命令名 参数1 参数2……

shell的工作是找到与命令名对应的可执行程序（命令），为其准备参数，并使用参数执行命令。

shell程序需具有以下几种功能和健壮性：

⑴ 支持目录检索功能，即文件不存在，继续打印提示符

⑵ 支持以“&”结束的输入，进行并发执行(前台与后台)

⑶ 支持输入输出重定向，“<”，“>”为标志符

⑷ 支持以“|”进行进程间通信操作（管道功能）

⑸ 支持一定的错误输入处理，例如：多于空格的出现，输入命令不存在，空输入等等。

考虑shell为了完成工作必须采取以下步骤：

⑴ 打印提示符

存在一个默认的提示符，有时在硬化shell中，如单字符串%、#或>。当shell启动时，它查找在其上运行的机器名并将该字符串加到标准提示符前，如kiowa>。Shell也可以被设计为打印当前目录作为提示符的一部分，也即每次用户键入cd改变到不同的目录时，提示字符串也会相应变化。一旦提示字符串确定下来，每当shell准备接受一个命令行时便把它打印到stdout。

⑵ 得到命令行

为了得到一个命令行，shell执行一个阻塞读操作让执行shell的进程进入睡眠状态，直到用户键入一个命令行作为对提示符的相应。一旦用户键入命令行（以一个NEWLINE（“\n”）字符结束），命令行字符串就被返回到shell。

⑶ 解析命令

命令行的语法非常琐细。解析程序从命令行的左边开始扫描直到它遇到一个空白字符（如空格、制表符或者NEWLINE）。第一个单词是命令的名称，而后面的单词则是参数。

⑷ 查找文件

shell为每一个用户提供一组环境变量。虽然这些环境变量可以随时通过set命令修改，但它们首先在用户的.login文件中定义。PATH环境变量是一个有序的绝对路径列表。它指明了shell应该在什么地方寻找命令文件，如果.login具有一个类似以下的行

set path=(./bin/usr/bin)

那么shell将首先在当前目录中寻找（因为第一个完全路径名是用“.”代表当前目录），然后是/bin，最后是/usr/bin。如果（从命令行）在任何指定的目录中都没有找到与命令行同名的文件，那么shell将提示用户无法找到命令。

⑸ 准备参数

shell简单地将参数传递到命令，作为指向字符串的argv数组。

⑹ 执行命令

shell必须执行指定文件的可执行程序。UNIX shell总是设计为当它执行一个程序时保护原始进程不被破坏。也就是说，因为一个命令可以是任何可执行文件，那么执行shell的进程必须保护它自己以防止可执行文件包含致命的错误。

（2）Linux提供的重要函数：

用户键入的命令可能有误，即便找到要求的可执行文件也可能包含致命的错误，因此要求shell程序每接受一个命令后，均创建子进程，让子进程来执行命令文件。如果命令正确，并创建子进程成功，父进程等待子进程完成。如果命令不正确或可执行文件有错或创建子进程失败，则给出相应的错误提示，撤消子进程，回到shell，等待用户输入下一个命令。这样shell进程不会受到破坏。

编写模拟shell，需要调用UNIX风格的系统调用fork()、exec()和wait()等。

⑴ int fork(void)

创建一个新进程。调用fork的进程称为父进程，新进程是子进程。Fork系统调用为父子进程返回不同的值：子进程中返回0，父进程中返回子进程的PID，如创建不成功返回负数值。

**⑵ exec系列**

exec系统调用采用新程序覆盖调用它的进程的地址空间。exec把一个新的程序装入调用进程的内存空间，来改变调用进程的执行代码。当exec（）返回时，进程从新程序的第一条指令恢复执行。exec没有建立一个与调用进程并发执行的新进程，而是用新进程取代了老进程。

⑶ wait系列

wait系统调用可以使进程等待子进程终止。该函数将阻塞调用进程，直到该进程的某一个子进程结束运行（如果子进程收到一个信号而结束运行，该函数也会返回）。子进程的结束状态保存在status指向的整数中。如果status为空，就不会返回子进程的结束状态。如果在调用wait之前子进程结束运行，wait会立即返回子进程的结束状态，wait返回值为子进程的进程ID。若子进程不存在，则返回-1。

wait(int \*stat\_loc); /\*System V,BSD,and POSIX.1 \*/

wait3(stausp,options,rusagep); /\* BSD \*/

waitpid(pid,\*stat\_loc,options); /\* POSIX.1 \*/

waited(idtype,id,infop,options); /\* SVR4 \*/

以上三个系统调用联用时，形成shell用于执行命令的一个代码框架。

if(fork()==0){

//这是子进程

//执行于和父进程相同的环境变量中

execvp(full\_pathname,command->argv,0);

}else

{

//这是父进程――等待子进程终止

wait(status);

}

**⑷ int dup(int fd)**

把一个程序的标准输出连接到管道的写入端，把另一个程序的标准输入连接到管道的读出端。返回一个新的文件描述符，该描述符和fd指向同一个文件。新的文件描述符具有同样的存取模式，以及同样的读/写偏移量。用这个函数可以进行输入/输出重定向。

输出重定向步骤如下：

fid=open(file,O\_WRONLY|O\_CREAT);

close(1);

dup(fid);

close(fid);

标准输入(stdin)、标准输出(stdout)和标准错误(stderr)分别占用0、1和2。要求先关闭标准输出，从而使1成为当前系统内可用作文件描述符的最小数（注意，0被标准输入占用），然后调用dup，使管道文件与文件描述符1（即标准输出）连接。

**⑸ int pipe(int pipeID[2])**

管道是单处理器Linux系统的主要IPC机制。默认情况下，管道采取异步发送并阻塞接受操作。管道是先进现出缓冲区，它设计为带有与文件I/O接口十分相似的API。一个管道可能在任何给定时刻包含系统定义最大数量的字节，通常为4KB。一个进程可以通过将数据写入到管道的一端来发送数据，而另一进程可以通过管道的另一端读取来接受数据。管道在内核中由一个文件描述符代表。一个欲建立管道的进程通过以下形式来调用内核：

int pipeID[2];

……

pipe(pipeID);

该函数可以创建两个文件描述符，pipeID[0]用于读，pipeID[1]用于写。这两个文件描述符连接起来就是一个管道。

对于使用管道作为IPC（进程间通信）的两个或多个进程，这些进程的共同祖先在创建进程之前就必须创建管道。因为fork命令创建一个含有打开文件表拷贝的子进程，子进程继承父进程所创建的管道。为了使用管道，它只需要读写适当的文件描述符。

例如，假如一个父进程创建了管道。那么它可以创建一个子进程并通过使用如下代码与之通信。

Pipe(pipeID);

If(fork()==0){ /\* 子进程 \*/

……

read(pipeID[0],childBuf,len);

/\* 处理childBuf中的消息 \*/

……

}else{

……

/\* 发送一个消息到子进程 \*/

write(pipeID[1],msgTochild,len);

……

}

管道使得进程可以把信息从一个地址空间拷贝到另一个地址空间。管道的读写端可以通过与文件描述符相同的方式用于大多数系统调用。下例中解释了Linux 中如何使用管道来实现进程A和B（分别执行PROC\_AP和PROC\_B）之间的并发处理，其中A执行了一些计算，发送一个值x给B，接着执行第二阶段的计算，从B中读取值y，然后重复执行。同时B读取值x，执行第一阶段的计算，写入一个值到A，再执行更多的计算，然后重复执行。不打算使用管道的进程应该关闭以使可以检测到文件结束（EOF）条件。

int A\_to\_B[2],B\_to\_A[2];

main()

{

pipe(A\_to\_B);

pipe(B\_to\_A);

if (fork()==0) /\*第一个子进程\*/

{

close(A\_to\_B[0]);

close(B\_to\_A[1]);

execve("prog\_A.out",...);

exit(1); /\*错误终端子进程\*/

}

if(fork()==0) /\*\*另一个子进程/

{

close(A\_to\_B[1]);

close(B\_to\_A[0]);

execve("prog\_B.out",...);

exit(1);

}

/\*父进程代码\*/

wait(...);

wait(...);

}

proc\_A()

{

while(TRUE)

{

<Compute A1>;

write(A\_to\_B[1],x,sizeof(int)); /\*使用管道发送消息\*/

<Compute A2>;

read(B\_to\_A[0],y,sizeof(int)); /\*使用管道得到消息\*/

}

}

proc\_B()

{

while(TRUE)

{

read(A\_to\_B[0],x,sizeof(int)); /\*使用管道得到消息\*/

<Compute B1>;

write(B\_to\_A[1],y,sizeof(int)); /\*使用管道发送消息\*/

<Compute B2>;

}

}

**⑹ int open(const char \*path, int oflag)**

打开文件，path参数是一个字符串，包含要打开文件的路径，oflag是标志集合，控制文件的打开方式。可用标志有：

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\_CERAT 如果文件已存在，则该选项不执行任何操作。

**⑺ int close(int fd)**

关闭文件。如果关闭成功，返回0，发生错误返回-1。

**⑻ int access(const char \*path,int amode)**

测试路径是否存在。参数path中包含了需要检查的文件路径名，amode是下面几个常量进行或操作的结果：

R\_OK 测试文件是否可读；

W\_OK 测试文件是否可写；

X\_OK 测试文件是否可执行；

F\_OK 测试文件是否存在；

分离环境变量中的PATH参数可使用如下语句：

paths=(char \*)getevn(“PATH”);

PATH中路径用“：”分割，可分别取出每个路径，后面再加上文件的相对路径名组成绝对路径，便可使用access函数进行检测。

**⑼char getenv(char \*name)**

获得特定环境变量。参数name应当是要获取的环境变量的名字。如果该变量存在，就返回该变量的值；否则返回NULL。

read(),write()等等。

**2、解决问题指导**

**（1）基本步骤**

**⑴ 打印提示符。**

在打印提示符之前要使用get\_current\_dir\_name()得到当前路径。返回一个指向当前目录绝对路径的字符串指针，然后在stdout中输出。

**⑵** **解析用户输入的指令，包括命令和参数。**

命令和参数被保存在字符串数据argv[argc]中。在这里要区分命令和参数，其中argv[0]是命令，程序要识别一般命令并并执行；能识别“>”、“<”和“|”，根据不同的符号转入到不同的程序模块，执行不同的过程。普通的命令在主函数main()中执行即可，当遇到“>”和“<”是转移到子函数redirect()执行；当遇到“|”则转移到子函数pipe()执行。

**⑶** **寻找命令文件。**

子函数is\_fileexit()用来查找命令是否存在。Shell为每个用户提供了一组环境变量。这些变量定义在用户的.login文件中。其中路径变量PATH是一组绝对路径的列表，表明shell如何搜索命令文件。只要我们获取了路径变量，然后依次搜索各个路径，就可以确定用户输入的命令文件的位置了。Leave指令用来退出自己的shell回到linux的shell下。

**⑷** **执行命令。**

通过调用fork()创建一个子进程，在子进程中执行命令。在子进程中通过使用execv()函数来执行命令。

**⑸** **实现重定向的。**

首先使用open()函数创建一个文件描述符,然后将其复制到文件描述表第二项。这样该进程的所有输出都写到重定向的文件中。使用dup(fid)函数将标准输出重定向到fd\_out上（就是重定向文件中）。

**⑹** **管道的实现。**

通过函数pipe(pipeID)，则pipeID[0]是一个文件描述符，指向管道的读端；对应的，pipeID[1]是一个指向管道写端的指针。创建第一个子进程执行管道符前的命令，并将输出写到管道。然后，创建第二个进程执行管道符后的指令，并从管道读输入流。

**（2）程序结构分析**

**⑴ 可以分成如下几个主要的文件：**

main.c 基本调度shell 的执行流程

function.c 包括了shell 功能里的各个功能子模块

head.h 涵盖了要使用的头文件和基本的全局变量、数据结构

shellexe shell的可执行文件

d1,d2,testresult 为测试时，临时生成的文件

**⑵ 主要的函数组成如下：**

main ( ) 负责调度整个shell 执行的流程

init\_shell ( ) 简单初始化shell 执行前的signal

init\_command ( ) 初始化shell 执行命令前的全局变量初值

get\_string ( ) 从screen 读取一个字符串

set\_background( )检测是否存在&, 设置background 标志符

delete\_space ( ) 过滤掉输入字符串中的多余的空格，包括前后和连续的多个空格

split ( ) 将输入字符串，分拆成一个个单词

fill\_cmds ( ) 将单词存储到shellcmd 结构、infile、outfile文件中去

searchfile ( ) 分离环境变量PATH中设定的路径，调度scanfile ( )

scanfile ( ) 在当前路径和PATH设定的路径中，检索每一个命令是否存在

execute ( ) 设置infile,outfile，循环调度do\_execute ( )

do\_execute ( ) 执行每一个shellcmd中的命令

head.h 包含了要使用的头文件和所有的全局变量和数据结构**⑶ 程序调用流程：**



以上程序流程图仅作为编程参考。其中含有重复或不尽合理的地方。编程实现时，可以自己另外划分函数与模块功能。

**实验二 Linux系统调用**

**一、实验目的**

学习系统调用的执行流程，理解Linux系统调用，并且在内核中增加一个新的系统调用。

**二、实验内容**

1. 下载最新版本的Linux内核源代码，在Linux系统中解压缩，大致观察内核源代码的组成结构。然后编译这个版本的内核代码，并尝试用编译出的内核重新引导系统。
2. 添加一个新的系统调用，完成任意一个功能，重新编译和运行内核，使新的系统调用可用。
3. 编写一个用户态的程序，使用增加的系统调用，以证明它确实可以用。

**三、实验提示**

1. **内核及系统调用原理**

Linux中有许多系统调用，如open()、read()、fork()、pipe()、socket()等，我们可以像使用普通的C库函数一样去使用这些系统调用，向操作系统发出请求，操作系统会尽量满足我们的请求。操作系统在处理每个系统调用的时候必须暂时切入核心态，待完成用户请求后再返回用户态继续执行。

一般在Linux系统中的/usr/src/linux\*.\*.\*（\*.\*.\*代表的是内核版本，如4.14）目录下就是内核源代码，如果没有类似目录，表示在安装Linux时没有选择安装内核源代码，这时可从Internet上免费下载。

Linux系统调用的执行原理和普通的C语言函数大不相同。当编写一个普通的C语言函数myfunc()时，这个函数将被编译成类似于下面所示的机器代码（以x86系列CPU、Visual C++中的编译器为例）：

myfunc proc near

函数内容

……

ret

myfunc end

可以看到，普通C语言函数在这里被编译成一个过程，在过程执行结束后，用一条ret指令返回调用处，而调用这个函数的语句将被编译成一条call myfunc指令。

在x86 CPU的执行过程中，一旦遇到call指令，CPU会把下一条指令的地址压入栈中，然后将程序计数器IP设为目标函数的地址，从目标函数处开始执行。而遇到ret指令，CPU将从栈中弹出一个值，并把程序计数器设为这个值，从这里开始执行。不难看出，无论是call，还是ret指令，都没有让CPU从用户态切换到核心态，或者从核心态转回用户态。而系统调用必须在用户态被调用，在核心态执行，这个过程中不可避免地要进行用户态/核心态的切换，因此系统调用不能像普通函数那样用call和ret指令实现。

在x86 CPU 中，用户态和核心态之间的切换，一般使用init指令，也就是中断指令。Linux系统调用正是通过中断实现的。Linux系统调用实际上是导出了可供用户态程序调用的内核函数的名称，它们不能像普通的C库函数一样调用，而必须通过0x80号中断切换到核心态调用。

Linux系统调用是很多的，所有这些系统调用共享了0x80号中断，那么当0x80号中断发生的时候，系统如何得知当前被调用的是哪个呢？在Linux内核中，维护了一张系统调用入口表，这个入口表的源码集中在syscals/syscalll\_64.tbl文件中。

接下来，就要实现自己的内核函数。可以单独在一个文件中添加该函数，也可以在已有的内核源代码中添加该函数。

**2、解决问题指导**

在编写核心态程序时，只能使用内核函数。在编写自己的代码之前，要搞清楚内核中相关的数据结构的定义。

以下的例子实现了枚举当前的所有进程，并且查询每个进程的gid、uid和pid，将所得的结果（进程信息数组，进程数）写在内核的系统日志中：

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/sched.h> /\*struct task\_struct\*/

asmlinkage int sys\_mysyscall()

{

int i=0;

struct task\_struct \*p;

p=&init\_task;

do

{

//获取该进程的信息

printk(“pid=%d ”,p->pid);

printk(“uid=%d ”,p->uid);

printk(“gid=%d ”,p->gid);

i++;

} while((p=p->next\_task) && (p!=&init\_task));

return i;

}

如何重新编译和引导新内核？

在使用新内核引导系统后，可以使用下面的测试程序，对新增的系统调用进行测试：

#define \_NR\_my\_syscall 223

/\*这里告诉编译器，my\_syscall系统调用的编号是223\*/

int errno;

#include <linux/unistd.h>

\_syscall0(int,my\_syscall)

/\*这里告诉编译器，请把my\_syscall作为一个系统调用来编译，而不是普通函数\*/

main()

{

My\_syscall(); /\*在main()函数里面使用了这个系统调用\*/

}

编译并运行这个用户态程序，并使用dmesg命令查看内核的系统日志。如果能够看到一个列表，显示了进程的pid等信息，那么说明实验成功了。

如果修改了内核目录中的文件内容，请在完成测试之后，把环境恢复到最初的状态。如果试图在已经被别人修改过的环境中工作，那么会出现一些奇怪的问题。这个时候最好的解决办法是删除当前的内核源码，并把以前下载的源码压缩包重新解压缩，创造一个最初状态的工作环境。

**3、编译Linux内核**

（1）Linux内核源码的组织结构

Linux内核作为一个特殊的程序，同样需要经过编译、链接之后才能运行，仅仅是它执行时拥有特殊的权限，位于特定的空间，并且不会也不可能依赖于其它软件罢了。

Linux发展至今，其内核的组织结构日渐清晰，层次日渐分明。一旦基本系统安装完毕，具有系统管理员的用户即可编译内核。一般说来，Linux系统内核的源代码放置在/usr/src/linux目录下，它将依赖于体系结构的代码和独立于体系结构的代码分离开来，以下是几个重要的源代码目录：

arch目录 该目录存放具体地依赖于体系结构实现的代码。在这个目录下，针对不同体系结构所移植的版本都有三个子目录：kernel、lib和mm。kernel子目录包含依赖于体系结构实现的一般内核功能，例如信号处理、时钟处理等。lib子目录包含库函数的本地实现，如果从依赖于体系结构的源代码编译，则运行得更快。mm子目录包含存储管理实现的代码。

drivers目录 拥有50%以上的内核源代码，所有的驱动程序源代码均位于该目录之下。

fs目录 存放所有的系统支持的文件系统的实现代码。

inlcude目录 一些重要的头文件。

ipc目录 处理进程间通信的全部所需的代码都放在ipc目录下。

init目录 存放所有系统的初始化代码，许多重要的文件，例如main.c就位于该目录下。该目录还包含了许多核心代码，例如实现fork()的代码和最常执行的代码----cpuidle()循环。

lib目录 放置内核其它部分经常所需要的代码。

kernel目录 许多最常调用的内核函数放在该目录下。

mm目录 包含所有Linux实现虚拟内存管理的源代码。

net目录 存放所有提供网络支持的代码。

document目录 存放大量的内核代码相关文档，以及用户开发和维护手册。

（2）如何配置及编译Linux内核

在能够实际地编译内核之前，必须告诉编译程序需要哪些功能，还必须告诉它是将这些功能模块编到内核去还是将其配置成动态可加载的模块。下表为内核配置命令：

|  |  |
| --- | --- |
| 类型 | 命令（必须具有管理员权限） |
| 文本提示 | make config |
| 文本菜单 | make menuconfig |
| GUI（需运行X-Window） | make xconfig |

make config将打开一个字符模式的对话框，在终端上一个接一个地问问题直到回答了所有的问题。对每一问题有三种可能的答案：Yes、No和Module。Module告诉内核配置在运行时使用动态可装载模块，而不是静态地将功能连接到内核中。

make menuconfig 和make xconfig分别打开一个文本图形对话框和一个GUI对话框，其功能和make config基本相同，但是可以只配置自己关心的部分。另外，如果不希望对原始配置做过多的修改，则可以选用make oldconfig选项。

make config命令中有几个重要的选项，其意义如下：

---Code maturity level options：代码成熟等级。此处只有一项：prompt for development and/or incomplete code/drivers，如果要试验现在仍处于试验阶段的功能，就必须把该项选择为Y，否则可以把它选择为N。

---Loadable module support：对模块的支持。

---Processor type and features：选择支持的CPU类型

---General setup：对最普通的一些属性进行设置。一般使用默认设置就可以。

---Memory Technology Device（MTD）：MTD设备支持。

---Parallel port support：并口支持。

---Plug and Play configuration：即插即用设备。

---Block devices：块设备支持。

---Networking options：网络配置选项。

---Telephony Support：电话支持。

---SCSI support：SCSI设备的支持。

---Character devices：字符设备。

---File system：文件设备。

---Console drivers：控制台驱动。

---Sound：声卡驱动。

---USB support：USB支持。

一旦满意地配置好内核，就可以进行编译了。

/usr/src/linux目录下的makefile是构建Linux内核的顶层makefile。在该目录下只用以下几个命令（必须具有超级用户或者管理员权限）就可以将内核重新构建出来：

#make clean

#make config

#make depend

#make

其中，make clean入口将删除旧的可装载对象文件和其他临时文件。make config入口会导致configure的bash脚本的执行，如果前面已经配置好了内核，那么这条命令可以不用。make depend入口是根据各个文件之间的依赖关系，确定合适的编译顺序。make入口将编译所有内核源代码，生成内核可执行文件vmlinux。

另外，make boot可以压缩vmlinux文件来建立一个可启动的内核映像并将其安装到/usr/src/linux/arch/i386/boot/zImage中。如果使用LILO引导，也可以使用命令make bzImage。这时，会在该目录下生成一个名为bzImage的文件，将上述文件重命名后，拷贝到/etc/lilo所在的目录，在lilo.conf中增加新的引导选项，再次运行LILO，新的内核就可以被引导了。

**四、举例**

（1）系统调用函数

asmlinkage int sys\_mysetpriority (int which, int who, int niceval)

{

struct task\_struct \*g, \*p;

struct user\_struct \*user;

const struct cred \*cred = current\_cred();

int error = -EINVAL;

struct pid \*pgrp;

kuid\_t uid;

if (which > PRIO\_USER || which < PRIO\_PROCESS)

goto out;

/\* normalize: avoid signed division (rounding problems) \*/

error = -ESRCH;

if (niceval < MIN\_NICE)

niceval = MIN\_NICE;

if (niceval > MAX\_NICE)

niceval = MAX\_NICE;

rcu\_read\_lock();

read\_lock(&tasklist\_lock);

switch (which) {

case PRIO\_PROCESS:

if (who)

p = find\_task\_by\_vpid(who);

else

p = current;

if (p)

error = set\_one\_prio(p, niceval, error);

break;

case PRIO\_PGRP:

if (who)

pgrp = find\_vpid(who);

else

pgrp = task\_pgrp(current);

do\_each\_pid\_thread(pgrp, PIDTYPE\_PGID, p) {

error = set\_one\_prio(p, niceval, error);

} while\_each\_pid\_thread(pgrp, PIDTYPE\_PGID, p);

break;

case PRIO\_USER:

uid = make\_kuid(cred->user\_ns, who);

user = cred->user;

if (!who)

uid = cred->uid;

else if (!uid\_eq(uid, cred->uid)) {

user = find\_user(uid);

if (!user)

goto out\_unlock; /\* No processes for this user \*/

}

do\_each\_thread(g, p) {

if (uid\_eq(task\_uid(p), uid) && task\_pid\_vnr(p))

error = set\_one\_prio(p, niceval, error);

} while\_each\_thread(g, p);

if (!uid\_eq(uid, cred->uid))

free\_uid(user); /\* For find\_user() \*/

break;

}

out\_unlock:

read\_unlock(&tasklist\_lock);

rcu\_read\_unlock();

out:

return error;

}

（2）用户测试程序

#include <stdio.h>

#include <unistd.h>

#include <sys/syscall.h>

#include <sys/types.h>

#include <sys/resource.h>

#define SYS\_MYSETPRIORITY 335

int main ()

{

int res = -1;

res = getpriority(PRIO\_PROCESS, getpid());

printf("before set, the priority is %d\n", res);

syscall(SYS\_MYSETPRIORITY, PRIO\_PROCESS, getpid(), 10);

res = getpriority(PRIO\_PROCESS, getpid());

printf("after set, the priority is %d\n", res);

return 0;

}

**实验三 Linux二级文件系统**

**一、实验目的**

（1）本实验的目的是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和内部实现。

（2）结合数据结构、程序设计、计算机原理等课程的知识，设计一个二级文件系统，进一步理解操作系统。

（3）通过分对实际问题的分析、设计、编程实现，提高学生实际应用、编程的能力

**二、实验内容**

在任一OS下，建立一个大文件，把它假象成一张盘，在其中实现一个简单的模拟Linux文件系统。在现有机器硬盘上开辟100M的硬盘空间，作为设定的硬盘空间。编写一管理程序simdisk对此空间进行管理，以模拟Linux文件系统，要求：盘块大小1k，空闲盘块的管理：Linux位图法；结构：超级块, i结点区, 根目录区。

1.可以实现下列几条命令：

* info显示整个系统信息(参考Linux文件系统的系统信息)，文件可以根据用户进行读写保护。
* 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
* copy 拷贝文件
* write 写文件
* cd 进出目录

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

4．程序的总体流程为：

初始化文件目录；

输出提示符，等待接受命令，分析键入的命令；

对合法的命令，执行相应的处理程序，否则输出错误信息，继续等待新命令，直到键入EXIT退出为止。

**三、实验提示**

1.背景知识

（1）外存管理

文件系统是一个含有大量的文件及其属性，对文件进行操作、管理的软件，以及向用户提供使用文件的接口的一个集合。在逻辑上它的层次结构是这样的：

|  |  |
| --- | --- |
| 文件系统接口 | |
| 对对象的操作和管理的软件集合 | 逻辑文件系统 |
| 基本I/O管理程序（文件组织模块） |
| 基本文件系统（物理I/O层） |
| I/O控制层（设备驱动程序） |
| 对象及其属性说明 | |

作为产品的操作系统有各自的文件系统。比如MS的WINDOWS系列使用的是FAT16、FAT32或NTFS的文件系统、LINUX使用的是EXT2、EXT3文件系统等等。

（2）linux的EXT2文件系统

linux使用一个叫虚拟文件系统的技术从而可以支持多达几十种的不同文件系统，而EXT2是linux自己的文件系统。它有几个重要的数据结构，一个是超级块，用来描述目录和文件在磁盘上的物理位置、文件大小和结构等信息。inode也是一个重要的数据结构。文件系统中的每个目录和文件均由一个inode描述。它包含：文件模式（类型和存取权限）、数据块位置等信息。

一个文件系统除了重要的数据结构之外，还必须为用户提供有效的接口操作。比如EXT2提供的OPEN/CLOSE接口操作。

（3）用内存来模拟外存

真正的文件系统对外存进行管理，涉及到许多硬件、设备管理方面的底层技术，一方面这些技术不属于操作系统核心内容，一方面过多的内容不免造成实验者顾此失彼，所以这里推荐一种使用内存来模拟外存的方式，可以跳过这些硬件技术而直接把精力放在数据结构设计和操作算法设计上面。

假定pInode是一个指向inode结构的指针，而且它已经放入的需要放入的数值了，现在需要将其写入到特定位置。可用如下代码：

……

fd=fopen(“filesystem”,”w+b”)； //fd是FILE指针类型，w便是写方式，b表示二进制

fseek(fd, specific\_area,SEEK\_SET)；// fd是文件指针；specific\_area为整形，

// 为需要入pInode的位置

fwrite(pInode, sizeof(inode), 1,fd)； // 写入pInode信息

2、原理算法

本文件系统采用两级目录，其中第一级对应于用户账号，第二级对应于用户帐号下的文件。另外，为了简便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。

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。

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。

**实验四 Linux驱动程序**

**一、实验目的**

（1）本实验的目的是通过一个驱动程序的设计，加深理解文件系统的内部功能和内部实现。

（2）结合数据结构、程序设计、计算机原理等课程的知识，设计一个驱动程序，进一步理解操作系统。

（3）通过分对实际问题的分析、设计、编程实现，提高学生实际应用、编程的能力。

**二、实验内容**

1、驱动程序编写步骤

1）移植编写Linux touch驱动程序，了解Input系统，学习移植代码

2）编写OSS声卡系统、编写虚拟网卡驱动程序

**三、实验提示**

1、块设备

（1）基本概念

扇区(Sectors)：任何块设备硬件对数据处理的基本单位。通常，1个扇区的大小为512byte。   
 块(Blocks)：由Linux制定对内核或文件系统等数据处理的基本单位。通常，1个块由1个或多个扇区组成。   
 段(Segments)：由若干个相邻的块组成。是Linux内存管理机制中一个内存页或者内存页的一部分。 综合上描述：块设备驱动是基于扇区(sector)来访问底层物理磁盘，基于块(block)来访问上层文件系统。扇区一般是2的n次方大小，典型为512B，内核也要求块是2的n次方大小，且块大小通常为扇区大小的整数倍，并且块大小要小于页面大小，典型大小为512B、1K或4K。

（2）块设备在Linux中的结构

首先，分析块设备驱动是以何种方式对块设备进行访问的？在Linux中，驱动对块设备的输入或输出(I/O)操作，都会向块设备发出一个请求，在驱动中用request结构体描述。但对于一些磁盘设备而言请求的速度很慢，这时候内核就提供一种队列的机制把这些I/O请求添加到队列中(即：请求队列)，在驱动中用request\_queue结构体描述。在向块设备提交这些请求前内核会先执行请求的合并和排序预操作，以提高访问的效率，然后再由内核中的I/O调度程序子系统(即：上图中的I/O调度层)来负责提交I/O请求，I/O调度程序将磁盘资源分配给系统中所有挂起的块I/O请求，其工作是管理块设备的请求队列，决定队列中的请求的排列顺序以及什么时候派发请求到设备，关于更多详细的I/O调度知识这里就不深加研究了。

  其次，块设备驱动又是怎样维持一个I/O请求在上层文件系统与底层物理磁盘之间的关系呢？在通用块层中，通常用一个bio结构体来对应一个I/O请求，它代表了正在活动的以段(Segment)链表形式组织的块IO操作，对于它所需要的所有段又用bio\_vec结构体表示。

  再次，块设备驱动又是怎样对底层物理磁盘进行访问方式。Linux提供了一个gendisk数据结构体，用他来表示一个独立的磁盘设备或分区。在gendisk中有一个类似字符设备中file\_operations的硬件操作结构指针，是block\_device\_operations结构体。

2、Wifi设备驱动

（1）sdio接口层解析

     SDIO总线 和 USB总线 类似，SDIO也有两端，其中一端是HOST端，另一端是device端。所有的 通信 都是 由HOST端 发送 命令 开始的，Device端只要能解析命令，就可以相互通信。

  CLK信号:HOST给DEVICE的 时钟信号，每个时钟周期传输一个命令。

  CMD信号：双向 的信号，用于传送 命令 和 反应。

  DAT0-DAT3 信号:四条用于传送的数据线。

  VDD信号:电源信号。

  VSS1，VSS2:电源地信号。

 （2）SDIO热插拔原理：

     方法：设置一个 定时器检查 或 插拔中断检测

     硬件：假如GPG10（EINT18）用于SD卡检测

          GPG10 为高电平 即没有插入SD卡

          GPG10为低电平  即插入了SD卡

 （3）SDIO命令

      SDIO总线上都是HOST端发起请求，然后DEVICE端回应请求。sdio命令由6个字节组成。

* Command:用于开始传输的命令，是由HOST端发往DEVICE端的。其中命令是通过CMD信号线传送的。
* Response:回应是DEVICE返回的HOST的命令，作为Command的回应。也是通过CMD线传送的。
* Data:数据是双向的传送的。可以设置为1线模式，也可以设置为4线模式。数据是通过DAT0-DAT3信号线传输的。

　　SDIO的每次操作都是由HOST在CMD线上发起一个CMD，对于有的CMD，DEVICE需要返回Response，有的则不需要。

       对于读命令，首先HOST会向DEVICE发送命令，紧接着DEVICE会返回一个握手信号，此时，当HOST收到回应的握手信号后，会将数据放在4位的数据线上，在传送数据的同时会跟随着CRC校验码。当整个读传送完毕后，HOST会再次发送一个命令，通知DEVICE操作完毕，DEVICE同时会返回一个响应。

       对于写命令，首先HOST会向DEVICE发送命令，紧接着DEVICE会返回一个握手信号，此时，当HOST收到回应的握手信号后，会将数据放在4位的数据线上，在传送数据的同时会跟随着CRC校验码。当整个写传送完毕后，HOST会再次发送一个命令，通知DEVICE操作完毕，DEVICE同时会返回一个响应。

sd命令格式

     以IO\_SEND\_OP\_COND命令为例包含以下部分：

     S（开始位） 总为0

     D（方向位） 1 从host到 device （0  从device到host）

     命令索引：  通过值000101B来

     填充位    0

     IO\_OCR      运转条件寄存器所支持的VDD的最小值和最大值

     CRC7        7位CRC校验数据

     E（结束位）  总为1

     MMC命令总共有40多个，分为class0 ~class7共8类，class0的所有卡必须支持。驱动程序通过发送cmd1、cmd41来区分sd卡和mmc卡，如果发送cmd1返回成功，则为mmc卡，否则发送cmd41返回成功，则为sd卡。

     cmd0   初始化mmc卡

（4））Sdio接口驱动

     首先我们来探讨几个重要的数据结构：该结果位于core核心层，主要用于 核心层与主机驱动层 的数据交换处理。/include/linux/mmc/host.h

struct mmc\_host 用来描述 卡控制器

struct mmc\_card 用来描述 卡

struct mmc\_driver 用来描述 mmc 卡驱动

struct sdio\_func  用来描述 功能设备

struct mmc\_host\_ops 用来描述 卡控制器操作接口函数功能，用于从 主机控制器层向 core 层注册操作函数，从而将core 层与具体的主机控制器隔离。也就是说 core 要操作主机控制器，就用这个 ops 当中给的函数指针操作，不能直接调用具体主控制器的函数。