**《Operating System》**

**实验指导书**

**计算机科学与技术系**

目 录

[实验1 Linux Basic Operation 1](#_Toc509821547)

[实验2 Process Creation 7](#_Toc509821548)

[实验3 Process Communication 16](#_Toc509821549)

[实验4 Thread Creation 22](#_Toc509821550)

[实验5 Semaphore and Process Concurrency 27](#_Toc509821551)

# 实验1 Linux Basic Operation

【实验目的】

1）上机学习 Linux文件操作。

2）理解Linux文件结构。

3）学会使用Linux操作系统下的Vi编辑器。

4）使用GCC编译源代码。

【条件要求】

1）认真阅读和掌握预备知识。

2）上机操作，完成实验内容。

【预备知识】

一、基础知识

1.Linux文件系统

1) 普通文件：包括文本文件，数据文件和可执行的二进制文件。

2) 目录文件：包含目录内文件的名称和一些简单的信息（如文件在磁盘中的位置）。

3) 特殊文件：与设备和进程之间的通信相关，每一个设备对应于一个文件。

2. 主要Linux目录内容

● /bin：二进制可执行文件。

● /dev：设备文件。

● /etc：系统管理和配置文件。

● /home：用户主目录的起始基点。

● /lib：标准程序设计库。

● /sbin：系统管理命令。

● /tmp：公用的临时文件存储点。

● /usr：应用程序、软件包的配置文件等。

● /var：某些大文件的溢出区。

3.路径和路径名

每一个文件都有一个路径名，路径名用于在文件系统中定位文件。每个路径名的最后是一个普通文件（文件）或者一个目录文件（目录）。

绝对路径是指从根目录开始定位一个文件，绝对路径总是从“/”开始。相对路径是指从工作目录开始定位一个文件。

当前目录下的文件可以直接访问，其他目录下的文件需要给出正确的路径名才能访问。在进行程序的编写时应尽量使用相对路径，这样有利于程序的分布。

4. 用户主目录和工作目录

1）用户主目录：当创建用户的时候，可以为用户分配一个特定的目录，这个目录称为该用户的主目录。用户每次登录系统，都会自动处于主目录下。

2）工作目录：用户当前所处的目录称为工作目录或当前目录。

5. 文件和目录名

Linux使用字母、数字、下划线和句点来命名文件和目录(根目录除外）。文件名对大小写敏感，并且不能出现空格。同一个目录下，不能出现同名的目录或文件，不同目录下不受此限制。可以使用扩展名来标识文件的类型，如“.c”，“.gz”，“.conf”，“.wav”和“.o”。file命令可用来查看文件类型。

注意：Linux中不使用扩展名来表示文件是否可以执行。

6．Vi编辑器概述

文本编辑器是所有操作系统中最常用的工具之一。用户在使用电脑的时候，往往需要创建自己的文件，无论是一般的文字文件、资料文件，还是编写源程序，这些工作都离不开编辑器。Linux操作系统也提供了一些文本编辑工具，如gedit，kwrite，OpenOffice等图形模式工具，如单行式编辑工具ed、全屏编辑器Vi、Vim（Vi的增强版本）及pico等文本模式工具。

Vi可以执行输出、删除、查找、替换及块操作等众多文本操作，但不足的是，它只能编辑普通的ASCII文件，不能对文件进行排版，即不能像Microsoft Office Word那样对字体、格式、段落等其他属性进行编排。

7. Vi编辑器的两种工作模式

Vi编辑器有两种基本工作模式：命令模式和文本输入模式。

在命令模式下，用户键入的字符被作为命令接收并解释执行，如复制、粘贴、删除、查找、保存及退出等命令。

在文本输入模式下，用户键入的字符被作为文件内容接收。这两种模式的转换过程如图1-1所示。

ZZ

命令

启动Vi

命令模式

文本输入模式

：

：提示符

其他命令

退出Vi

wq,q,q!

a/i/o/A/I/O/R

Esc

图1-1 Vi工作模式转换

启动Vi后首先进入命令模式，这时不能直接输入文件内容，必须在键盘上输入字符a（或i，o，A，I，O，R）后，进入文本输入模式。在文本输入模式下，通过键盘输入的内容被写到文件里面。当要执行复制、粘贴、保存等命令时，按Esc键，回到命令模式。在命令模式下输入相应的命令后，按回车键，命令即被执行。有些命令不能直接在命令行模式下输入，必须先输入一个“：”，出现“：”提示符再输入该命令，执行之后回到命令行模式。或者在“：”提示符后输入“退出”命令，退出Vi编辑器。

8. GCC编译器简介

Linux系统下经常使用的程序设计语言是C语言，可以使用vi编辑器编辑C语言程序，编辑好的C语言程序需要经过编译才能运行。在Linux下最常用的C语言程序编译工具就是GCC（GNU Compiler Collection），它是GNU项目中符合ANSI C标准的编译系统。

事实上，GCC是一个功能非常强大的编译器，它的功能并不局限于编译C程序，还可以编译由C＋＋，Java，Fortran及Pascal等语言编写的程序，因为它可以通过不同的前端模块来支持各种语言。此外，GCC也是一种性能优越的多平台编译器，可以在多种硬体平台上编译出可执行程序，其执行效率与一般的编译器相比，要高出20%～30%。

GCC编译器能将C/C++语言源程序、汇编语言程序和目标程序编译、连接成可执行文件，如果没有给出可执行文件的名字，GCC将默认生成一个名为“a.out”的文件。在Linux系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。

二、文件相关命令

|  |  |  |  |
| --- | --- | --- | --- |
|  | command | Description | Example |
| 1 | Ls | List file or directory information | ls; ls –a ; ls /etc, ls Radio |
| 2 | mkdir | Create directory | mkdir music; mkdir –p music/pop ; mkdir /tmp/abc |
| 3 | cd | Change current directory | cd music cd /tmp/abc |
| 4 | pwd | List current directory | pwd |
| 5 | rmdir | Remove directory( this directory must be empty) | rmdir music; rmdir /tmp/abc |
| 6 | touch | Create a new empty file or upate the date of a existing file. | touch file1 |
| 7 | Rm | Remove file | rm file1 ; rm radio/rain.mp3, rm /tmp/file2 |
| 8 | cp | copy file or directory | cp file1 file2, cp /tmp/file1 /usr  cp –r /home/user1 /tmp |
| 9 | cat | view text file content | cat file1 |
| 10 | more | view text file content | more file1 |
| 11 | Less | view text file content | less file1 |
| 12 | head | print the first 10 lines of file | head file1 |
| 13 | Tail | Print the last 10 lines of file | tail file1 |
| 14 | Mv | Move file or change name of file or directory | mv file1 file2 , mv file1 /tmp |
| 15 | which | Look in the directories listed in the user's search path for the required file | which ls |
| 16 | find | Find file in file system | find /usr -name "\*.tmp"  find / -name "\*.tmp" -exec rm {} |
| 17 | slocate | Find file in a file indexed database | slocate ls |
| 18 | Tar | Compress and uncompress file and directory | tar –cjf user1.tar.bz2 /home/user1  tar -czf user1.tar.gz /home/user1  tar –xjf user1.tar.bz2  tar –czf user1.tar.gz |

三、GCC的使用方法

Gcc is GNC project C and C++ compiler in Linux. When you invoke GCC，it normally does preprocessing, compilation, assembly and linking. The first three stages apply to an individual source file, and end by producing an object file, linking combines all the object files into executable file.

For any given input file, the file name suffix determines what kinds of compilation is done. Filename.c stand for C source code.

The most commonly used option introduction:

gcc source.suffix –o sourc.o, the object file for souce.suffix in source.o.

if –o is not specified, the default is to put an executable file in a.out.

【实验内容】

1. create directory /home/test1 ,/root/test2
2. change current directory to /home/test1
3. list current directory
4. show all files(including hidden files) in the current directory
5. create file1 in /home/test1
6. copy file1 to /root/test2
7. delete file1
8. delete /home/test1
9. rename file2 in /root/test2 to file3
10. vi experiment 1

Create a file named linux, the content is following:

Linux is a free open-source operating system based on Unix.

Linux was originally created by Linus Torvalds

Linux is free to download, edit and distribute.

Linux is a free open-source operating system based on Unix.

(1)set number for this file

(2)find all “Linux” and change them to “Unix”

(3)save and quit

1. vi and gcc experiment 2

Type a C program named cTest4.c whose function is : calculate the value of 1+2+3+….+1000.

Requirement:

(1)use loop structure in C programming to accomplish the problem.

(2)Compile the source file and run it.

# 实验2 Process Creation

【实验目的】

1）学会使用Linux操作系统下的fork()和exec()。

2）学会使用Linux操作系统下的wait()。

【条件要求】

1）认真阅读和掌握预备知识。

2）上机操作，完成实验内容。

【预备知识】

一、fork（）函数

一个进程调用了fork以后,系统会创建一个子进程。这个子进程和父进程的不同之处在于进程ID和父进程ID，其他都一样。

当一个程序中调用fork函数后，内核会完成如下工作：

● 内核系统分配新的内存块和内核数据结构；

● 复制原来的进程到新的进程；

● 向运行进程集添加新的进程；

● 将进程返回给两个进程。

设原来的进程为父进程，调用fork生成的新进程为子进程，则子进程会执行父进程中fork函数后的代码。

fork系统调用使用格式：

● 头文件：

#include <sys/types.h> /\* 提供类型pid\_t的定义 \*/

#include <unistd.h>

● 函数原形为：

pid\_t　 fork(void);

● 返回值为“pid\_t”。

对于父进程，fork函数返回了子程序的进程号，而对于子程序，fork函数则返回零。这样，对于程序，只要判断fork函数的返回值，就知道自己是处于父进程还是子进程中。如果调用不成功，则返回“-1”。

二、exec族函数

exec函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容。换言之，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何Linux下可执行的脚本文件。

exec系统调用有6种不同的使用格式，但在核心中只对应一个调用入口。它们有不同的调用格式和调用参数。这六种调用格式分别为：

#include <unistd.h>

1）int execl (const char \*path, const char \*arg0, ..., const char\*argn, (char \*)0);

2）int execv (const char \*path, char \*const \*argv);

3）int execle (const char \*path, const char \*arg0, ..., const char\*argn,(char \*0), const char \*envp[]);

4）int execve (const char \*path, char \*const \*argv, char \*const \*envp);

5）int execlp (const char \*file, const char \*arg0, ..., const char\*argn, (char \*)0);

6）int execvp (const char \*file, char \*const \*argv);

参数“path”指出一个可执行目标文件的路径名，参数“file”指出可执行目标文件的文件名。“arg0”作为约定同“path”一样指出目标文件的路径名，参数“arg1”到“argn”分别是该目标文件执行时所带的命令行参数。参数“argv”是一个字符串指针数组，由它指出该目标程序使用的命令行参数表，按约定第一个字符指针指向与“path”或“file”相同的字符串。最后一个指针指向一个空字符串，其余的指向该程序执行时所带的命令行参数。参数“envp”同“argv”一样也是一个字符指针数组，由它指出该目标程序执行时的进程环境，它也以一个空指针结束。

exec的6种格式在以下三点上有所不同：

1）“path”是一个目标文件的完整路径名，而“file”是目标文件名，它可以通过环境变量“PATH”来搜索。

2）由“path”或“file”指定的目标文件的命令行参数，是完整的参数列表或是通过一指针数组“argv”给出的。

3）环境变量是系统自动传递或者通过“envp”给出的。

表2-1说明了exec系统调用的6种不同格式对以上三点的支持。

表2-1 exec系统调用的6种不同格式

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | 参数形式 | 环境传送 | 路径搜索 |
| execl | 全部列表 | 自动 | 否 |
| execv | 指针数组 | 自动 | 否 |
| execle | 全部列表 | 不自动 | 否 |
| execve | 指针数组 | 不自动 | 否 |
| execlp | 全部列表 | 自动 | 是 |
| execvp | 指针数组 | 自动 | 是 |

与一般情况不同，exec函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段、数据段和堆栈等都已经被新的内容取代，只留下进程ID等一些表面上的信息仍保持原样。看上去还是旧的“躯壳”，却已经注入了新的“灵魂”。只有调用失败了，它们才会返回一个“-1”，从原程序的调用点接着往下执行。

现在我们应该明白Linux如何执行新程序：，每当有进程认为自己不能为系统和用户做出任何“贡献”了，它就可以发挥最后一点“余热”，调用任何一个exec，让自己以新的面貌“重生”；或者，更普遍的情况是，如果一个进程想执行另一个程序，它就可以fork出一个新进程，然后调用任何一个exec，这样看起来就好像通过执行应用程序而产生了一个新进程一样。

事实上第二种情况被应用得如此普遍，以至于Linux专门为其作了优化。我们已经知道，fork可以将调用进程的所有内容原封不动地拷贝到新产生的子进程中，这些拷贝的动作很消耗时间，而如果fork完之后马上就调用exec，这些辛辛苦苦拷来的东西又会被立刻抹掉。这看起来非常不划算，于是人们设计了一种名为“写时拷贝（copy-on-write）”的技术，使得fork结束后并不立刻复制父进程的内容，而是到了真正实用的时候才复制。这样，如果下一条语句是“exec”，它就不会作无用功了，因而也就提高了效率。

三、wait系统调用

wait系统调用可以完成父进程和子进程的同步。进程一旦调用了wait，就立即“阻塞”自己，由wait自动分析当前进程的某个子进程是否已经退出，如果让它找到了这样一个子进程，wait就会收集这个子进程的信息，并把它彻底销毁，然后返回；如果没有找到这样一个子进程，wait就会一直“阻塞“在这里，直到有一个出现为止。

wait系统调用使用格式为：

#include <sys/types.h> /\* 提供类型pid\_t的定义 \*/

#include <sys/wait.h>

pid\_t wait(int \*status)

参数“status“用来保存被收集进程退出时的一些状态，它是一个指向int类型的指针。如果我们对这个子进程是如何“死掉”的毫不在意，只想把这个“僵尸进程”消灭掉（事实上绝大多数情况下，我们都会这样想），就可以设定这个参数为“NULL”，就像这样：

pid = wait(NULL);

【实验内容】

1、fork() 创建新进程，请解释结果中各个输出语句由哪个进程输出打印。

#include “stdio.h”

#include<sys/types.h> /\* 提供类型pid\_t的定义 \*/  
#include<unistd.h> /\* 提供函数的定义 \*/

main()

{

pid\_t id;

printf("hello Linux!\n");

id=fork();

if(id>0)

printf(“parent\n”);

else if(id==0)

printf(“child\n”);

else

printf("fork error\n");

}

2、fork() 创建进程，通过getpid（）得到父子进程ID。

#include<sys/types.h>  
#inlcude<unistd.h>  
main()  
{  
 pid\_t pid;   
 /\*此时仅有一个进程\*/  
 pid=fork();  
 /\*此时已经有两个进程在同时运行\*/  
 if(pid<0)  
 printf("error in fork!");  
 else if(pid==0)  
 printf("I am the child process, my process ID is %d\n",getpid());  
 else  
 printf("I am the parent process, my process ID is %d\n",getpid());  
}

$gcc fork\_test.c -o fork\_test  
$./fork\_test

结果示例如下：  
I am the parent process, my process ID is 1991  
I am the child process, my process ID is 1992

3、wait()阻塞原语，父进程阻塞，等待子进程结束。

#include <sys/types.h> /\* 提供类型pid\_t的定义 \*/  
#include <sys/wait.h>  
pid\_t wait(int \*status)

int main()

{int i;

if(fork()>0)

{

wait(NULL);

for(i=0;i<5;i++)

{ printf("AAAAAAAAAAAAAAAA\n");

sleep(2);

}

}

else

for(i=0;i<5;i++)

{

printf("BBBBBBBBBBBBBBBBB\n");

sleep(2);

}

return 1;

}

4、分析父子进程中各自value值的情况。

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int value=5; // 全局

int main(){

pid\_t pid;

pid=fork();

if(pid==0){

value += 15;

printf("Child: value = %d\n",value);

}

else if(pid>0){

wait(NULL);

printf("PARNET: value = %d\n",value);

exit(0); //父进程终止

}

}

5、分析该代码的输出结果。

#include <sys/types.h> /\* 提供类型pid\_t的定义 \*/  
#include <sys/wait.h>  
#include <unistd.h>  
#include <stdlib.h>  
main()  
{  
pid\_t pc,pr;  
pc=fork();  
if(pc<0) /\* 如果出错 \*/  
 printf("error ocurred!\n");  
else if(pc==0){ /\* 如果是子进程 \*/   
 printf("This is child process with pid of %d\n",getpid());  
 sleep(10); /\* 睡眠10秒钟 \*/  
}  
else{ /\* 如果是父进程 \*/  
 pr=wait(NULL); /\* 在这里等待 \*/  
 printf("I catched a child process with pid of %d\n",pr);  
}   
exit(0);  
}

6、分析execlp（）代码的输出结果。

#include <stdio.h>

#include <unistd.h>

int main()

{ printf("\* \* \* About to exec ls -l\n");

printf("\* \* \* AAAAAAAAA\*\*\*\*\*\n");

execlp( "/bin/ls" , "ls",NULL );

printf("\* \* \* BBBBBBBBBB\*\*\*\*\*\n");

return 1;

}

7、分析execvp（）代码的输出结果。

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t id=0；

char \*parameter[3];

parameter[0] = "ls";

parameter[1] = "-a";

parameter[2] = 0 ;

id=fork();

if(id==0)

{

execvp( "ls" , parameter );

printf("cannot execute the command!\n");

}

else

{

wait(NULL);

printf("\* \* \* ls is done. bye\n");

printf("\* \* \* BBBBBBBBBB\*\*\*\*\*\n");

}

return 1;

}

8、简单的命令解释器shell。

shell0.c

//shell0.c

#include “stdio.h”

#include “string.h”

int main()

{ char cmd[100];

char \*argv[2];

while(1)

{ printf(“%%”);

scanf(“%s”,cmd);

if(strcmp(cmd,“exit”)==0)

break;

argv[0]=cmd;

argv[1]=0;

if(fork()>0)

{

wait(NULL);

continue;

}

else

{

execvp(cmd,argv);

printf("cannot excute the command!\n");

exit(1);

}

}

return 1;

}

# 实验3 Process Communication

【实验目的】

1）了解管道的工作原理及通信方式。

2）使用管道进行一些简单的进程通信。

【条件要求】

1）认真阅读和掌握预备知识。

2）上机操作。

【预备知识】

一、管道概述

Linux下进程间通信（IPC）的几种主要手段有管道及有名管道、信号、消息队列、共享内存、信号量和套接口。其中管道和有名管道是最早的进程间通信机制之一。管道可用于具有亲缘关系进程（即具有公共祖先的进程）间的通信。有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。下面对管道的通信机制及相关API进行简单的介绍。

管道是Linux支持的最初Unix IPC形式之一，具有以下特点：

1）管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道。

2）只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）。

3）单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件。但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。

4）数据的读出和写入：一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。如图3-1所示。

fd[0] 读端

fd[1] 写端

图3-1 管道示意图

二、管道的创建

#include <unistd.h>  
int pipe(int fd[2])

pipe函数可以创建一个管道，并将其两端连接到两个文件描述符“fd[0]”和“fd[1]”。其中“fd[0]”为读文件描述符，“fd[1]”为写文件描述符。像一个打开的文件的内部情况一样，管道的内部实现隐藏在内核中，进程只能看见两个文件描述符。

如果管道创建成功，则返回“0”；如果不成功，返回“－1”并设置errno。当errno值为“EMFILE”时，表示因文件描述符不够用而创建失败；当errno值为“ENFILE”时，表示因系统中同时打开的文件数超出系统限定的范围而创建失败。

创建的管道的两端用于一个进程中，在实际应用中没有太大意义，因此，一个进程在由pipe创建管道后，一般再fork一个子进程，然后通过管道实现父子进程间的通信（因此也不难推出，只要两个进程存在亲缘关系——这里的亲缘关系指的是具有共同的祖先——都可以采用管道方式来进行通信）。

三、管道的读写规则

管道两端可分别用描述符“fd[0]”和“fd[1]”来描述，需要注意的是，管道的两端已经固定了任务：即一端只能用于读，由描述符“fd[0]”表示，称其为管道读端；另一端则只能用于写，由描述符“fd[1]”来表示，称其为管道写端。如果试图从管道写端读取数据，或者向管道读端写入数据都将导致错误发生。一般文件的I/O函数都可以用于管道，如close，read，write等等，但与文件的I/O有所不同，从管道中读写数据的时候要注意以下事项。

1.从管道中读取数据

1）当读一个写端存在的管道时，如果管道为空，则读进程会一直阻塞直到写进程将某些内容写入管道为止。

2）当读一个写端存在的管道时，如果管道非空，读进程请求读的字节数目大于“PIPE\_BUF”（常数“PIPE\_BUF”规定了内核中管道缓冲器的大小，它的值在“include/linux/limits.h”中定义，不同的内核版本可能会有所不同。Posix.1要求“PIPE\_BUF”至少为512字节，red hat 7.2中为4096字节），则返回管道中现有的数据字节数。

3）当读一个写端已被关闭的空管道时，read返回“0”，表示达到了文件结束处。

2.向管道中写入数据

1）当管道的读端存在时，写进程向管道中写入数据，如管道缓冲区没有足够的空闲区域容纳新的数据，则写进程阻塞直到读进程从管道缓冲区读走数据腾出空间。

2）只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的SIFPIPE信号，应用程序可以处理该信号，也可以忽略（默认动作则是应用程序终止该信号）。

四、管道的应用

管道可用于输入输出重定向，它将一个命令的输出直接定向到另一个命令的输入。比如，在某个shell下键入“who│wc –l”后，相应shell程序将创建who以及wc两个进程和这两个进程间的管道。who进程将其执行结果写入管道，wc进程从管道中读出数据后进行相应的处理。

管道还可用于具有亲缘关系的进程间通信，如父进程通过管道发送一些命令给子进程，子进程解析命令，并根据命令作相应处理。

五、管道的局限性

管道的主要局限性正体现在它的特点上：

1）只支持单向数据流。

2）只能用于具有亲缘关系的进程之间。

3）没有名字。

4）管道的缓冲区是有限的（管道只存在于内存中，在管道创建时，为缓冲区分配一个固定大小的页面）。

5）管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定数据的格式，比如把多少字节算作一个消息（或命令、记录）等等。

【实验内容】

1、输出PIPE\_BUF的大小。

//输出PIPE\_BUF的大小，Linux内核2.9.130以前一个缓冲区，之后为16个缓冲区。

#include<stdio.h>

#include <linux/limits.h>

int main()

{

printf("PIPE\_BUF %d(bytes)\n",PIPE\_BUF);

return 0;

}

2、子进程通过管道的写端将字符串写入管道，父进程通过管道读端将子进程写入管道的字符串读取出来。

#include <unistd.h>

int main(void)

{

int x, fd[2];

char buf1[40],buf2[40];

if(pipe(fd) < 0) /\*创建管道\*/

{

perror("could not make pipe");

exit(1);

}

x=fork();

if(x== -1)

{

perror("fork erro");

exit(1);

}

if(x==0)/\*子进程\*/

{

close(fd[0]);

sprintf(buf1,"hello");

write(fd[1],buf1,40);

exit(0);

}

else/\*父进程\*/

{

close(fd[1]);

wait(NULL);

read(fd[0],buf2,40); /\*父进程读管道字符\*/

printf("%s\n",buf2); /\*输出该字符串\*/

}

}

3、查看其执行结果并解释原因。

#include <stdio.h>

#include <unistd.h>

int main(int ac, char \*\*av)

{

int x, fd[2],newfd;

if(ac!=3)

{

perror("usage: ./exam11b cmd1 cmd2\n");

exit(1);

}

if(pipe(fd) < 0) /\*创建管道\*/

{

perror("could not make pip");

exit(1);

}

x=fork();

if(x== -1)

{

perror("fork error");

exit(1);

}

if(x==0)/\*子进程重定向到标准输出并执行av[1]指定的程序\*/

{

close(fd[0]);

if(dup2(fd[1],1)== - 1)

{

perror("could not redirect stdout");

exit(1);

}

close(fd[1]);

execlp(av[1],av[1],NULL);

perror(av[1]);

exit(1);

}

else /\*父进程重定向到标准输入并执行av[2]指定的程序\*/

{

close(fd[1]);

if(dup2(fd[0],0)== - 1)

{

perror("could not redirect stdin");

exit(1);

}

close(fd[0]);

execlp(av[2],av[2],NULL);

perror(av[2]);

exit(1);

}

}

# 实验4 Thread Creation

【实验目的】

1）掌握Linux系统中线程创建的方法。

2）掌握Linux系统中线程间互斥的实现方法。

【条件要求】

1）认真阅读和掌握预备知识。

2）上机操作。

【预备知识】线程是进程的一条执行路线

一、进程与线程

Linux是一个多用户、多任务的操作系统。多用户是指多个用户可以在同一时间通过该操作系统使用计算机；多任务是指Linux可以同时执行几个任务。在Linux中，一个进程可以通过fork（）系统调用创建出多个进程并分别完成不同的任务；另外，也可以在一个进程中创建出多个线程来完成不同的工作。

线程技术早在20世纪60年代就被提出，在20世纪80年代中期开始应用到操作系统中。现在，多线程技术已经被许多操作系统所支持，包括Windows NT/2000和UNIX/Linux。

根据操作系统的定义，进程是系统资源管理和分配的最小单位，线程是程序执行的最小单位。

线程是在共享内存空间中并发的多道执行路径，它们共享一个进程的资源，如文件描述和信号处理。在两个普通进程(非线程)间进行切换时，内核准备从一个进程的上下文切换到另一个进程的上下文要花费很大的开销（上下文切换的主要任务是保存原执行进程CPU的状态，并加载新进程的保存状态，用新进程的内存映像替换原进程的内存映像）。从一个线程切换到另一个线程所花费的代价比进程要低。

线程能更充分地利用内存。线程可以看作是在进程内部执行的指令序列。线程和进程的最大区别在于线程完全共享相同的地址空间，运行在同一地址上。

二、线程的创建

我们在此讨论的是在POSIX中定义的线程，即pthread。在POSIX中，程序从main函数开始执行，这相当于单个进程中的单个线程。在运行过程中，可以通过调用pthread\_create函数创建新的线程。新创建的线程可以对应于程序中的一个函数。创建线程的函数如下：

#include <pthread.h>

int pthread\_create (pthread\_t \*thread\_id,   
const pthread\_attr\_t \*attr,  
void \*(\*start\_fcn) (void \*),  
void \*arg);

参数说明如下：

● thread\_id：第一个参数为指向线程标识符（线程id）的指针。

● attr：线程的属性。

● start\_fcn：新创建线程开始运行的地址。

● arg：传递给线程的参数。

●返回值：成功返回“0”，失败置错误码。

如果要把程序中的一个函数创建成一个线程来运行，可以将函数名（函数的起始地址）作为第三个参数“start\_fcn”调用“pthread\_create（）”，新创建的线程将从该函数开始执行，但新创建的线程与调用“pthread\_create（）”的线程哪一个会先执行是不确定的。

我们可以使用pthread\_join函数等待一个线程的结束：

#include <pthread.h>

int pthread\_join (pthread\_t thread\_id, void \*\*status\_ptr);

参数说明如下：

● thread\_id：被等待线程的标识符（线程id）。

● status\_ptr：用户定义的指针，它可以用来存储被等待线程的返回值。

这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

三、线程的互斥

由于一个进程的多个线程可以共享进程的资源，因此多个线程有可能会同时访问共享进程中的公共资源。如果它们同时访问的资源是不能同时访问的临界资源，则可能会出现错误的结果。这时，应该采取互斥机制保证每个线程互斥地访问临界资源。在POSIX中可以通过定义一个互斥变量并通过使用该互斥变量的加锁、解锁操作来实现互斥。

互斥变量用“pthread\_mutex\_t”数据类型表示，在使用互斥变量之前，必须首先对它进行初始化，可以把它置为常量“PTHREAD\_MUTEX\_INITIALIZER”。

当一个线程进入临界区访问临界资源之前，需要调用“pthread\_mutex\_lock（）”对互斥变量进行加锁操作，如果此时该资源是空闲的，则加锁后开始访问临界资源；如果互斥变量已经被加锁（临界资源正在被其他线程访问），则当前线程阻塞直至互斥变量被解锁（其他线程释放了临界资源）；当一个线程访问完临界资源后释放时，调用“pthread\_mutex\_unlock（）”对互斥变量进行解锁操作。解锁后，如果有因为等待该资源而阻塞的线程，则唤醒该线程。

互斥变量的定义及初始化如下：

static pthread\_mutex\_t mtx=PHTREAD\_MUTEX\_INITIALIZER;

互斥变量的加锁、解锁函数如下：

#include <pthread.h>

● 加锁：int pthread\_mutex\_lock(pthread \_mutex\_t \*mutex);

● 解锁：int pthread\_mutex\_unlock(pthread \_mutex\_t \*mutex);

其中的“mutex”为互斥变量。

【实验内容】

1.编写一个包含两个线程的程序。程序运行时两个线程分别完成不同的工作，观察实验结果，解释原因。

#include <stdio.h>

#include <pthread.h>

void thread(void)

{

int i;

for(i=0;i<3;i++)

{

printf("This is a pthread.\n");

sleep(1);

}

}

int main(void)

{

pthread\_t id;

int i,ret;

ret=pthread\_create(&id,NULL,(void \*) thread,NULL);

if(ret!=0)

{

printf ("Create pthread error!\n");

exit (1);

}

for(i=0;i<3;i++)

{

printf("This is the main process.\n");

sleep(1);

}

pthread\_join(id,NULL);

return (0);

}

2.观察实验结果，解释原因。

#include <stdio.h>

#include <pthread.h>

void thread(void)

{ int i,sum=0;

for(i=1;i<=10;i++)

{ sum=sum+i;

sleep(1);

}

printf("The sum is %d\n",sum);

}

int main()

{ pthread\_t id，id1;

int i,ret;

char buf[80];

ret=pthread\_create(&id,NULL,(void \*) thread,NULL);

while(1)

{ printf("please input a string\n");

scanf("%s",buf);

printf("This string is %s\n",buf);

}

pthread\_join(id,NULL);

return (0);

}

编译：

# gcc –lpthread –o threadtest.out threadtest.c

# 实验5 Semaphore and Process Concurrency

【实验目的】

1）了解BACI并发运行环境。

2）使用BACI设计并发程序，深入理解并发概念。

【条件要求】

1）认真阅读和掌握预备知识。

2）上机操作。

【预备知识】

一、BACI简介

并行和同步是计算机科学中的重要课题。由于对并行和分布式计算的日益重视，理解并行和同步变得愈发重要。为了获得对于这些概念的感性认识，编写并行程序的实践经验将是必不可少的。BACI是帮助学生获得需要的“第一手”并行程序设计经验的选项之一。

BACI是Ben-Ari Concurrent Interpreter（Ben-Ari并行解释器）的缩写。编译器和解释器最初由M. Ben-Ari设计完成，主要基于Niklaus Wirth的Pascal编译器。最初版本的BACI编译器和解释器基于这些代码，并运行于一个PRIME大型机。经过大量的修改和扩充，目前版本已经移植到了PC上，支持的语言包括Turbo Pascal, Sun Pascal以及C。此外，编译器和解释器被分成了两个独立的程序。后来，BACI中还加入了C--编译器，这是一个有限的C++子集，它可以编译C--程序为解释器能够解释的中间代码(PCODE object code)。

与其他并行语言相比，BACI在学生熟悉的语法框架内提供了多种并行技术，所有具有C或Pascal程序设计经验的学生都可以在几小时内掌握BACI的使用，BACI已经被成功地运用于很多大学的课堂教学。

二、C--编译器语法

BACI的C--是C++的一个子集。换言之，C--符合C++语法，此外，它包括一些限制和新的数据类型，例如：

1）除了标准输入输出（cout，cin，endl）之外不支持其他文件操作。

2）C-- BACI只支持简单的C/C++类型：int和char。简单类型的常量修饰符（const）也被支持，所有的变量必须在代码块的开头定义。

3）支持字符串类型（string）。BACI同时包括内建的字符串处理函数，例如stringCopy, stringCompare, stringConcat等等。

4）支持简单类型和字符串数组，数组定义遵循常见的C语法。

5）支持过程和函数，采用标准的作用域规则，参数定义可以使用传值和传递引用的方式，执行开始于对“main()”的调用。

6）过程控制语句包括if-else，switch/case，for，while，do-while，break和continue，这些语句的句法和标准的C/C++相同。

三、并行结构

1.cobegin

cobegin块包括一组并发执行的进程列表，这样的块不允许嵌套，而且只能出现在主程序中。列表中的PCODE语句将被解释器以任意、随机的顺序执行，于是多次执行包括cobegin块的相同程序的结果将呈现不可预测性。主程序将挂起，直到cobegin块终结，此时主程序将恢复执行紧随块后的下一条语句。下面是一个例子：

cobegin

{

proc1( ... ); proc2( ... ); ... ; procN( ... );

}

2.semaphore

semaphore是一个BACI预定义类型，它是一个非负的整数变量，并且只能通过受限的方式访问。BACI也提供了一个它的子类型——binarysem，这是一个二元信号量，它的值只能是0或1。信号量函数包括：

● initialsem ( semaphore s, integer\_expression )：能够初始化BACI中两种信号量s的唯一的函数。

● p(semaphore s)和wait(semaphore s )：如果semaphore > 0，则递减1并返回，从而调用者能够继续运行；如果semaphore = 0，则递减1，p的调用者将休眠。

● v(semaphore s)和signal(semaphore s)：如果semaphore=0并且存在等待它的进程，则将随机地唤醒一个，并且semaphore递增1。无论如何v的调用者将继续。

3.monitors

BACI也支持有限的Hoare monitor概念，monitor是包括附加属性的C--块，所有monitor变量中的函数都对外可见，但monitor变量不可以在块外访问，而只能由monitor函数访问。

在BACI中，monitor只能在全局域中定义，并且不能嵌套。任意时刻只能有一个monitor块中的过程或函数被执行。这一特性使得能够使用monitor来实现互斥。

有三个可以被过程或函数使用的monitor机制：condition变量，waitc（条件等待）以及signalc（条件信号）。

● condition变量：只能被monitor的函数访问。这种变量不能有实际的值，它是进行P操作或V操作的标志。

● void waitc(condition cond, int prio)：monitor进程将被阻塞，并赋予指定的优先级(prio)以备重新唤醒。这个阻塞动作允许就绪的其他进程继续执行。

● void waitc(condition cond)：与waitc调用语义相同，但这一P操作将被赋予优先级10。

● void signalc(condition cond)：如果存在的话，唤醒等待cond的优先级最高的进程。

● void empty(condition cond)：当没有等待cond的进程时返回“1”，反之返回“0”。

4.其他并行结构

● atomic(原子的)关键字：如果某个函数被定义为“atomic”，则函数将是不可剥夺的。解释器将不会在进程切换时中断atomic函数。

● void suspend(void)：将调用进程置为休眠状态。

● void revive(int process\_id)：激活给定id的进程。

● int which\_proc( void )：返回当前进程的线程数。

● int random(int range)：返回0到“range-1”之间，包括端点的伪随机数。

四、怎样使用BACI

BACI C--编译器源文件的扩展名必须是“.cm”，在BACI中运行程序的步骤是：

1）编译“.cm”文件以获得PCODE文件(.pco)。命令格式为：

bacc [参数] 源文件名

常见的参数包括：

● –h：显示帮助信息。

● –c：产生“.pob”目标文件，以便进行后续的连接操作。

2）使用解释器解释执行PCODE(.pco)文件。命令格式为：

bainterp [参数] PCODE文件名

常见的参数包括：

● –d：调试模式，单步执行，可以设置断点。

● –e：在每一进程项旁边显示活动记录(AR)。

● –x：每一进程退出时显示AR。

● –t：通知进程结束。

● –h：显示帮助信息。

● –p：执行时显示PCODE指令。

【实验内容】

1. 观察实验结果，解释原因。

void func1()

{

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

cout<<"AAAAAAAAAAAAAA"<<endl;

cout<<"BBBBBBBBBBBBBB"<<endl;

}

void func2()

{

cout<<"$$$$$$$$$$$$$$"<<endl;

cout<<"CCCCCCCCCCCCCC"<<endl;

cout<<"DDDDDDDDDDDDDD"<<endl;

}

main()

{

cobegin{func1();func2();}

}

2、观察实验结果，解释原因。记录执行结果。看看x的结果是否等于100。

int x=0;

void func1()

{

int i=0;

for(i=0;i<50;i=i+1)

{ x=x+1;}

}

void func2()

{

int i=0;

for(i=0;i<50;i=i+1)

{ x=x+1;}

}

main()

{

cobegin{func1();func2();}

cout<<"x=="<<x <<endl;

}

3、观察实验结果，解释原因。记录执行结果。看看x的结果是否等于100。试说明原因。

int x=0;

semaphore s;

void func1()

{

int i=0;

for(i=0;i<50;i=i+1)

{

wait(s);

x=x+1; /\*临界区\*/

signal(s);

}

}

void func2()

{

int i=0;

for(i=0;i<50;i=i+1)

{

wait(s);

x=x+1; /\*临界区\*/

signal(s);

}

}

main()

{

initialsem(s,1);

cobegin{func1();func2();}

cout<<"x=="<<x <<endl;

}