第5章 Shell程序设计

实验目的

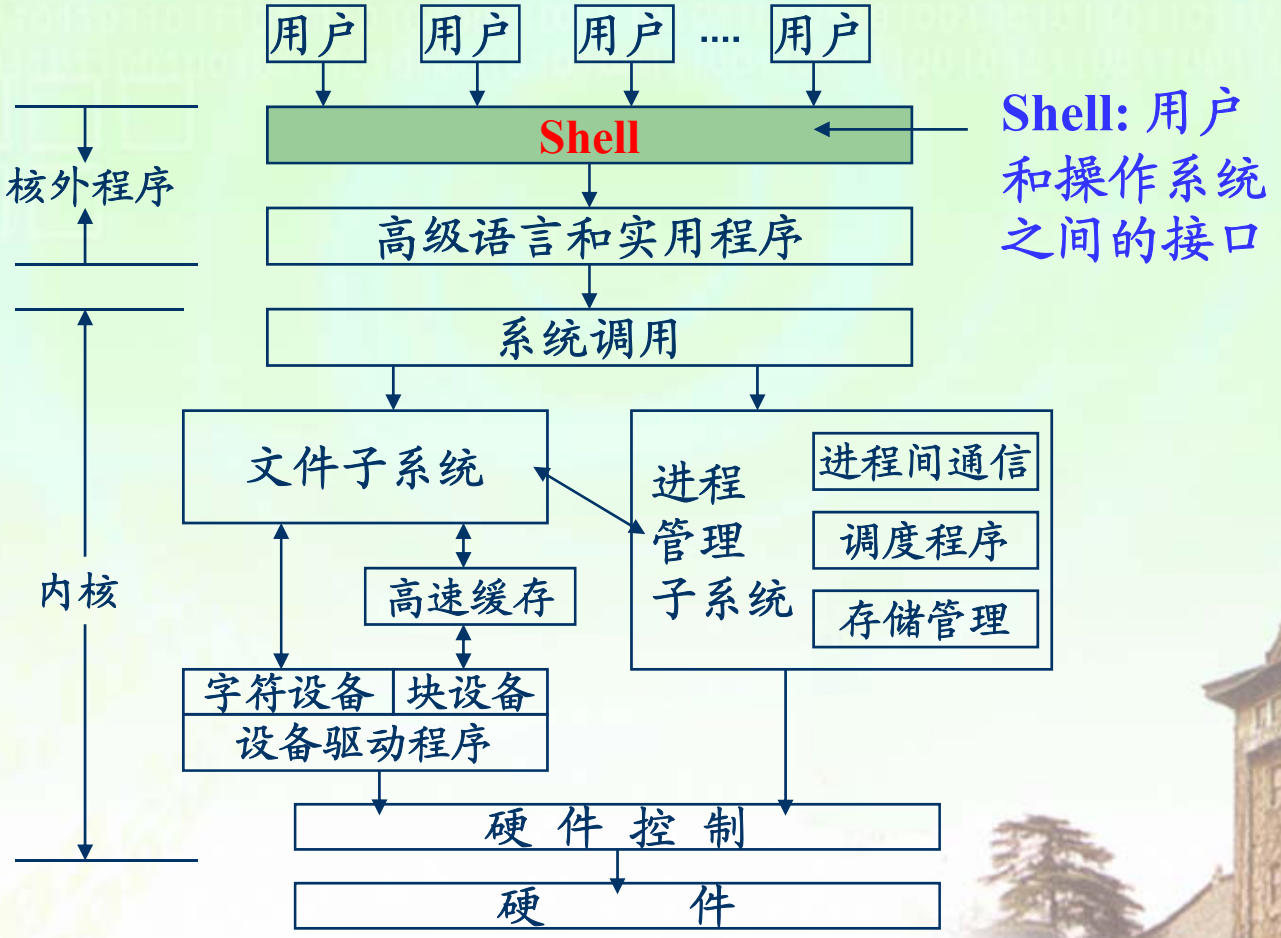
- 了解Shell在操作系统中的作用
- 理解I/O重定向和管道线
- 学会编写简单的Shell脚本程序
- 学会运行Shell命令文件

主要内容

- 背景知识
  - Shell基础
  - Shell编程
- 实验内容
  - 编写一个简单的Shell程序—Myshell
  - 基于Shell的网络管理

Shell概述

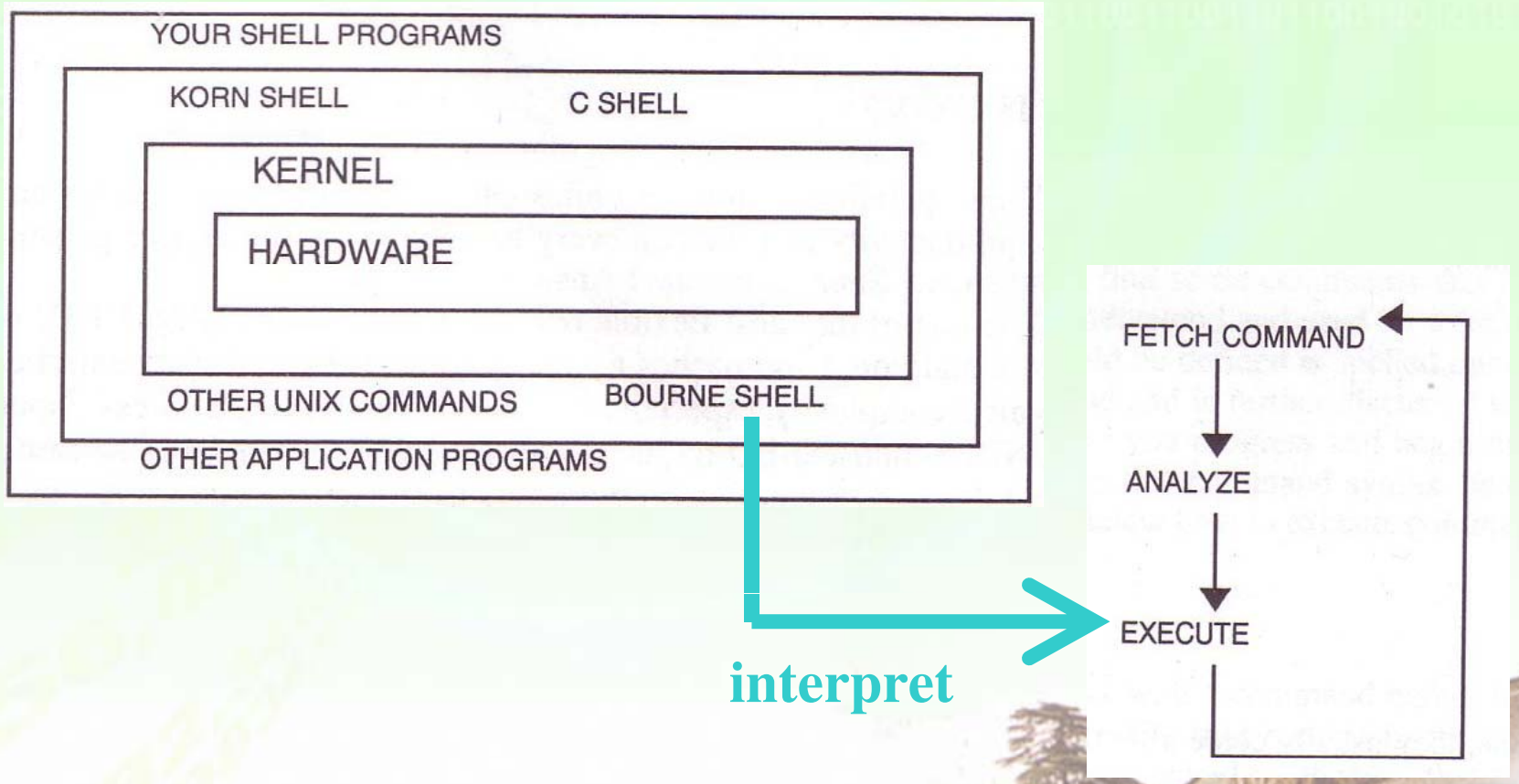
❖用户和操作系统之间的接口



Shell: 用户和操作系统之间的接口

Linux内核与Shell

- 作为核外程序而存在



Shell主要功能

- 命令解析
- 命令执行
- 前台执行和后台执行
- I/O重定向
- Shell管道
- 环境变量和环境文件
- Shell编程和Shell脚本

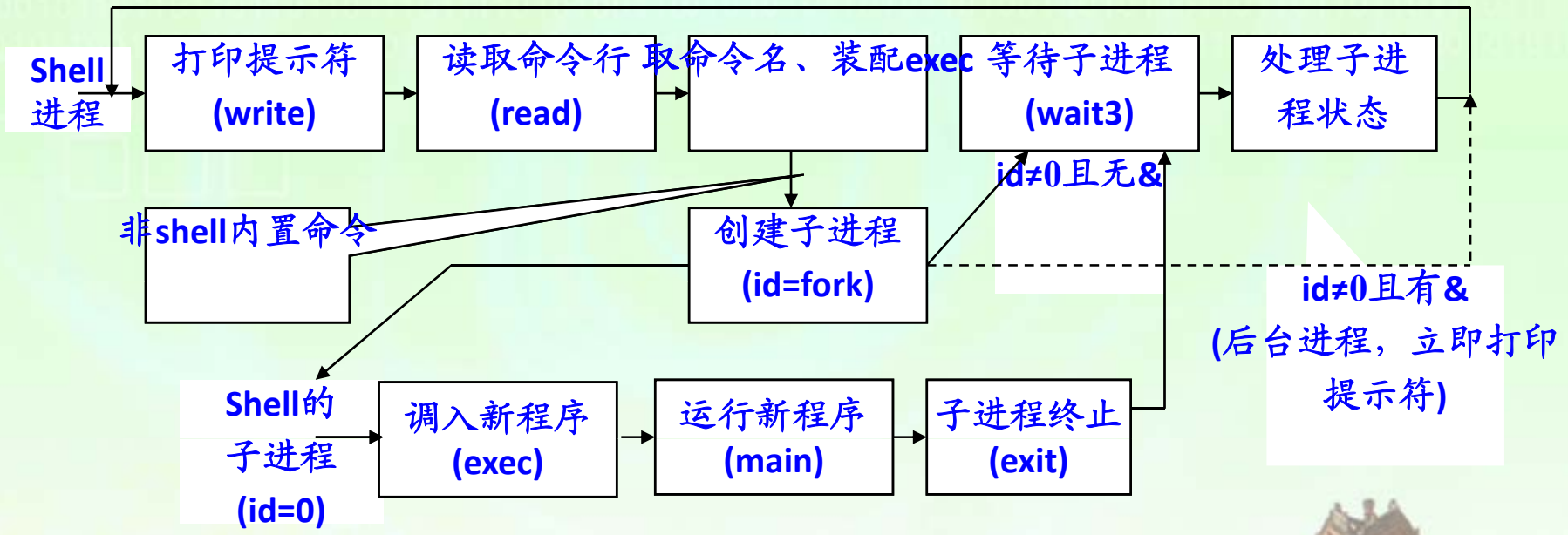
Shell分类

shell名称	描述	位置
ash	一个小的shell	/bin/ash
ash.static	一个不依靠软件库的ash版本	/bin/ash.static
bsh	ash的一个符号链接	/bin/bsh
bash	“Bourne Again Shell”。Linux中的主角，来自GNU项目	/bin/bash
sh	bash的一个符号链接	/bin/sh
csh	C shell, tcsh的一个符号链接	/bin/csh
tcsh	和csh兼容的shell	/bin/tcsh
ksh	Korn Shell	/bin/ksh

Shell的双重角色

- 命令解释程序
  - Linux的开机启动过程：进程树
  - Shell的工作步骤
    - ✓ 打印提示符；得到命令行；解析命令；查找文件；准备参数；执行命令
- 独立的程序设计语言解释器
  - KISS (Keep It Small and Stupid)
  - Reusable tools
  - Redirection and pipe

Shell的执行步骤


Shell进程
打印提示符(write)
读取命令行(read)
取命令名、装配exec
等待子进程(wait3)
处理子进程状态
id≠0且无&
非shell内置命令
创建子进程(id=fork)
id≠0且有&
(后台进程，立即打印提示符)
Shell的子进程(id=0)
调入新程序(exec)
运行新程序(main)
子进程终止(exit)

Shell程序

- 也称Shell script(Shell脚本)
  - 是一组命令
  - 举例

```
#!/bin/sh

ls -al
touch aa
cp aa bb
```

- Shell编程的基础知识
  - Linux环境
  - Linux命令
  - Shell程序结构

Shell脚本结构

```
#!/bin/sh
# first.sh
# This file looks through all the files in the current
# directory for the string POSIX, and then displays those
# files to the standard output

for file in *
do
     if grep –q POSIX $file
     then
          more $file
     fi
done

exit 0
```

Shell脚本的执行方法

- 方法1
  - $ sh script_file
- 方法2
  - chmod +x script_file (可选chown, chgrp)
  - ./script_file
- 方法3
  - source script_file, or
  - . script_file

Shell启动文件

- **sh**
  - /etc/profile — login shell, system wide
  - ~/.profile — login shell
  - ENV
- **csh**
  - /etc/csh.cshrc — always, system wide
  - /etc/csh.login — login shell, system wide
  - ~/.cshrc — always
  - ~/.login — login shell
  - ~/.logout — logout shell
  - /etc/csh.logout — logout shell, system wide
- **tcsh**
  - ~/.tcshrc — login shell
- **bash**
  - /etc/profile → ~/.bash_profile → ~/.bash_login → ~/.bash_profile
  - /etc/bash.bashrc → ~/.bashrc
  - BASH_ENV

Shell的特殊字符

字符	说明
*	Match any string of characters
?	Match any single alphanumeric character
[…]	Match any single character within []
[!...]	Match any single character not in []
~	Home directory
#	Start a shell comment
;	Command separator
&&	executes the first command, and then executes the second if first command success (exit code=0)
||	executes the first command, and then executes the second if first command fail (exit code≠0)
\	1) Escape character 2) Command continuation indicator
&	Background execution

Shell的特殊字符应用举例

- 如果当前目录下有如下文件
  - test1 test2 test3 test4 test-5 testmess

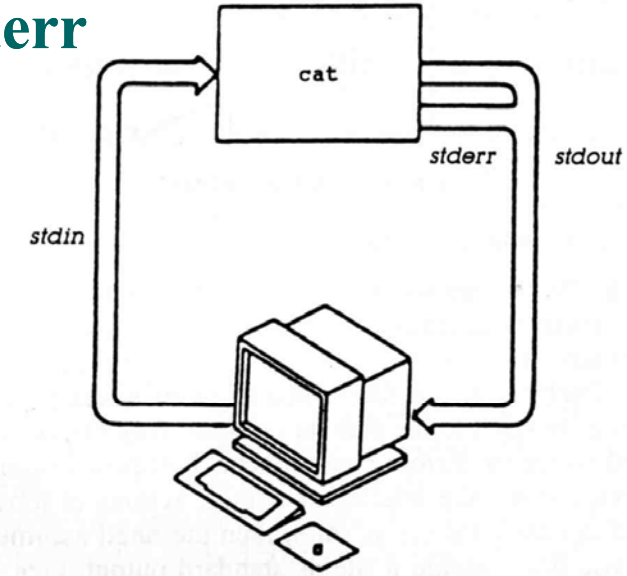
命令	结果
% ls test*	test1 test2 test3 test4 test-5 testmess
% ls test?	test1 test2 test3 test4
% ls test[123]	test1 test2 test3
% ls ~	List files under your home

Shell输入/输出重定向

❖每个处理器都由个默认的文件描述符

名字	输入/输出	描述符号码
stdin	input	0
stdout	output	1
stderr	error output	2
User-defined	Input/output	3 ~ 19

- 正常情况下终端是stdout和stderr
- 键盘是stdin



Shell输入/输出重定向

❖ 重定向

- 改变stdin, stdout, stderr或任何用户自定义文件描述符的方向
  - ✓ 创建文件
  - ✓ 追加文件使用现有文件作为输入
  - ✓ 合并两个输出流
  - ✓ 使用部分shell命令作为输入

操作符	描述
<	Open the following file as stdin
>	Open the following file as stdout
>>	Append to the following file
<<del	Take stdin from here, up to the delimiter del
>&	Merge stdout with stderr
>>&	Append stdout to stderr
|	Pipe stdout into stdin
n>&-	Close file descriptor

Shell输入/输出重定向举例

- >:输出重定向

  $ ls –l > lsoutput.txt

- >>：追加

  $ ps >> lsoutput.txt

- 出错输出重定向 (2>)

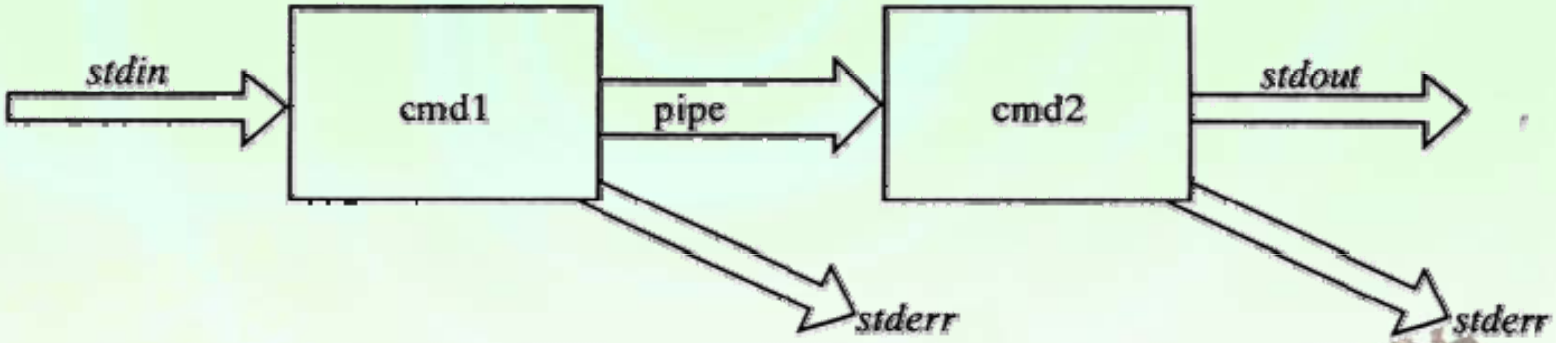
  $ kill –HUP 1234 > killout.txt 2> error.txt

- <：输入重定向

  $ more < killout.txt

Shell管道

- 将一个命令输出作为另一个命令的输入
- 两个命令将异步执行
  - ps | sort > passort.out



Shell内置命令

sh	csh	描述
	alias/unalias	command alias
ulimit	limit/unlimit	limit job's resource usage
cd	cd	change directory
echo	echo	write arguments on stdout
eval		evaluate and execute arguments
exec	exec	execute arguments
exit	exit	exit shell

Shell内置命令

sh	csh	描述
	goto	Goto label within shell program
	history	Display history list
jobs	jobs	List active jobs
%[job no.]	%[job no.]	Bring a process to foreground
kill	kill	Send a signal to a job
fg, bg	fg, bg	Bring a process to foreground/background
	stop	Stop a background process
	suspend	Suspend the shell
login	login, logout	Login/logout

Shell内置命令

sh	csh	说明
set/unset		Set/Unset shell's parameters
	set/unset	Set/Unset a local variable
export	setenv/unsetenv	Set/Unset a global variable
	nice	Change nice value
	nohup	Ignore hangups
	notify	Notify user when jobs status changes
trap	onintr	Manage execution signals
	dirs	print directory stack
	popd, pushd	Pop/push directory stack

Shell内置命令

sh	csh	说明
hash	rehash	Evaluate the internal hash table of the contents of directories
read		Read a line from stdin
shift	shift	Shift shell parameters
.	source	Read and execute a file
times	time	Display execution time
umask	umask	Set default file permission
test		Evaluation conditional expressions
expr	@	Display or set shell variables
wait	wait	Wait for background jobs to finish

主要内容

- 背景知识
  - Shell简介
  - Shell编程
- 实验内容
  - 编写一个简单的Shell程序—Myshell
  - 基于Shell的网络管理

Shell程序设计的语法

- 变量
- 条件测试
- 条件语句
- 重复语句
- 命令表和语句块
- 函数
- 其它

Shell环境变量

环境变量	说明
$HOME	当前用户的登陆目录
$PATH	以冒号分隔的用来搜索命令的目录清单
$PS1	命令行提示符，通常是”$”字符
$PS2	辅助提示符，用来提示后续输入，通常是”>”字符
$IFS	输入区分隔符。当shell读取输入数据时会把一组字符看成是单词之间的分隔符，通常是空格、制表符、换行符等。

参数变量和内部变量

环境变量	说明
$ #	传递到脚本程序的参数个数
$0	脚本程序的名字
$1, $2, …	脚本程序的参数
$*	一个全体参数组成的清单，它是一个独立的变量，各个参数之间用环境变量IFS中的第一个字符分隔开
$@	“$*”的一种变体，它不使用IFS环境变量

Shell变量赋值

	Bourne Shell	C Shell
Local variable	my=test	set my=test
Global variable	export my	setenv my test

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

Notice, there is no space between the "=" when assigning a variable

Shell变量访问

- % echo “$PAGER”
- % echo “${PAGER}”
  - Use {} to avoid ambiguity
- % temp_name=“haha”
- % temp=“hehe”
- % echo $temp
  - hehe
- % echo $temp_name
  - haha
- % echo ${temp}_name
  - hehe_name
- % echo ${temp_name}
  - Haha

参数变量和内部变量举例1

- 假设脚本名为myscript
  - 如果执行./myscript foo bar baz，结果如何？

```
#!/bin/sh
salutation="Hello"
echo $salutation
echo "The program $0 is now runnning"
echo "The 1st & the 2nd parameters were $1 & $2"
echo $*
exit 0
```

```
$ ./myscript foo bar baz
Hello
The program ./myscript is now runnning
The 1st & the 2nd parameters were foo & bar
foo bar baz
```

参数变量和内部变量举例2

- 假设脚本名为var3.sh
  - 执行sh ./var3.sh hello world earth，输入如何？

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

```
$ sh ./var3.sh hello world earth
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

参数变量和内部变量举例3

```
1  #!/bin/sh
2  echo "What is your name?"
3  read USER_NAME
4  echo "Hello $USER_NAME"
5  echo "File ${USER_NAME}_file will be
created"
6  touch "${USER_NAME}_file"
```

- 代码说明
  - 第5行使用${USER_NAME}_file来确保shell将使用变量$USER_NAME，而不是 $USER_NAME_file.
  - 第6行使用“${USER_NAME}_file”避免任何输入给$USER_NAME的空格.

Shell变量引用

- 当包含一个或多个空格时使用"..."
- 当$variable被双引用包含时，将被其值取代
- 当$variable被单引号（'....'）引用时，不会被替换。
- 在$variable前带有\时，将取消上述特殊含义
- 一般而言，字符串用双引号引用，以便避免被空格分隔，但允许使用$来替换

变量与字符串引用

字符串	作用
var=value set var=value	Assign value to variable
$var ${var}	Get shell variable
`cmd`	Substitution stdout
‘string’	Quote character without substitution
“string”	Quote character with substitution

- % varname=`/bin/date`
- % echo $varname
- % echo ‘Now is $varname’
- % echo “Now is $varname”

- % set varname2=`/bin/date`
- % echo $varname2
- % echo ‘Now is $varname2’
- % echo “Now is $varname2”

Tue Nov 13 14:54:57 CST 2007
Now is $varname
Now is Tue Nov 13 14:54:57 CST 2007

Shell变量引用举例

```
#!/bin/sh
myvar="Hi there"
echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar
echo Enter some text
read myvar
echo '$myvar' is $myvar
exit 0
```

Shell变量引用举例输出结果

Output

```
#!/bin/sh
myvar="Hi there"
echo $myvar                          Hi there
echo "$myvar"                        Hi there
echo '$myvar'                        $myvar
echo \$myvar                         $myvar
echo Enter some text                 Enter some
text
read myvar                           Hello↵
echo '$myvar' is $myvar              $myvar is
Hello
exit 0
```

使用双引号的字符

- 双引号是Shell的重要组成部分

```
$ echo Hello      World
Hello World
$ echo "Hello    World"
Hello    World
```

- 如何显示：Hello　"World“
- 以下 命令可以吗？$ echo "Hello　　"World""“
- 正确方法：echo "Hello　\"World\""

Shell变量操作符

- 错误情形（ BadCond ）
  - 变量未设定或者值为空
- 正确情形（ GoodCond ）
  - 变量已设定并且值不为空

操作符	说明
${var:=value}	If BadCond, assign value to var
${var:+value}	If GoodCond, use value instead else null value is used but not assign to var
${var:-value}	If !GoodCond, use the value but not assign to var
${var:?value}	If !GoodCond, print value and shell exists

Shell变量操作符

操作符	说明
${#var}	String length
${var#pattern}	Remove the smallest prefix
${var##pattern}	Remove the largest prefix
${var%pattern}	Remove the smallest suffix
${var%%pattern}	Remove the largest suffix

```
#!/bin/sh

var="Nothing happened end closing end"

echo ${#var}
echo ${var#*ing}
echo ${var##*ing}
echo ${var%end*}
echo ${var%%end*}
```

Results:
32
happened end closing end
end
Nothing happened end closing
Nothing happened

条件测试

- 测试命令
  - test expression 或 [expression]
- test命令支持的条件测试
  - 字符串比较
  - 算术比较
  - 与文件有关的条件测试
  - 逻辑操作

测试命令使用方法

测试文件是否存在

```
if test -f fred.c
then
......
fi
```

Notices that spaces are required around the '[' and ']' characters

也可写成以下形式：

```
if [ -f fred.c ]
then
.....
fi
```

or

```
if [ -f fred.c]; then
.....
fi
```

字符串比较

字符串比较	结果
str1 = str2	两个字符串相同则结果为真
str1 != str2	两个字符串不相同则结果为真
-z str	字符串为空则结果为真
-n str	字符串不为空则结果为真

Spaces before and after the = sign is required.

```
$ str1 = "abc"
$ if [ $str = "abc“ ]; then
>   echo "The strings are equal"
> else
>   echo "The strings are not equal"
> fi
```

算术比较

算术比较	结果
expr1 –eq expr2	两个表达式相等则结果为真
expr1 –ne expr2	两个表达式不等则结果为真
expr1 –gt expr2	expr1 大于 expr2 则结果为真
expr1 –ge expr2	expr1 大于或等于 expr2 则结果为真
expr1 –lt expr2	expr1 小于 expr2 则结果为真
expr1 –le expr2	expr1 小于或等于 expr2 则结果为真

```
$ x = 5; y = 7
$ if [ $x -lt $y ]; then
> echo "x is less than y"
> fi
```

与文件有关的条件测试

文件条件测试	结果
-e file	文件存在则结果为真
-d file	文件是一个子目录则结果为真
-f file	文件是一个普通文件则结果为真
-s file	文件的长度不为零则结果为真
-r file	文件可读则结果为真
-w file	文件可写则结果为真
-x file	文件可执行则结果为真

```
$ mkdir temp
$ if [ -f temp ]; then
> echo "temp is a directory"
> fi
```

逻辑比较

逻辑操作	结果
! expr	逻辑表达式求反
expr1 –a expr2	两个逻辑表达式“And”（“与”）
expr1 –o expr2	两个逻辑表达式“Or”（“或”）

条件语句

- if语句
- case语句

if语句

- 形式

```
if [ expression ]
then
    statements
elif [ expression ]
then
    statements
elif …
else
    statements
fi
```

- 紧凑形式
  - ; (同一行上多个命令的分隔符)

if语句举例

```
#!/bin/sh
echo "Is it morning? (Answer yes or no)"
read timeofday
if [ $timeofday = "yes" ]; then
    echo "Good Morning"
else
    echo "Good afternoon"
fi
exit 0
```

```
if condition
then
    ...
else
    ...
fi
```

elif语句

❖做进一步检查

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
    echo "Good Morning"
elif [ $timeofday = "no" ]; then
    echo "Good afternoon"
else
    echo "Wrong answer! Enter yes or no"
    exit 1
fi
exit 0
```

It is advised to use "$timeofday" and not $timeofday. Why?

case语句

```
case variable in
      pattern [ | pattern] ...) statements ;;
      pattern [ | pattern] ...) statements ;;
       ....
esac
```

```
#!/bin/sh
echo "Is it morning? Enter yes or no";read timeofday
case  "$timeofday" in
  yes | y | Yes | YES)  echo "Good Morning";;
  n* | N*  )            echo "Good Afternoon";;
  *  )  echo "Sorry, answer not recognized"
        echo "Please answer yes or no"
        exit 1;;
esac
```

case语句举例

❖ 当执行append时，有多少种可能的情况出现？

```
#!/bin/sh
#脚本名: append
case $#
  1) cat >> $1 ;;
  2) cat >> $2 < $1 ;;
  *) echo 'usage: $0 [fromFile] toFile' ;;
esac
```

1. No parameter or more than 2
2. Only 1 parameter & the file exist
3. Only 1 parameter & the file not exist
4. Both file exist
5. 1st exist; 2nd not exist
6. 2nd exist; 1st not exist
7. Both files not exist

循环语句

- for语句
- while语句
- until语句
- select语句

for语句

```
#!/bin/sh
for foo in bar fud 43
do
  echo $foo
done
exit 0
```

```
#!/bin/sh
for i in 1 2 3 4 5
do
  echo "Number: $i"
done
```

```
for variable in values
do
  ....
done
```

```
#!/bin/sh
for file in $(ls f*h)
do
   mv "$file" _"$file"
done
exit 0
```

while语句

```
while condition
do
      ....
done
```

```
#!/bin/sh
echo "Enter \"hkIVE\""
read a
while [ "$a" != "hkIVE" ]
do
  echo "Error, try again"
  read a
done
exit 0
```

```
#!/bin/sh
foo=1
while [ "$foo" -le 5 ]; do
   echo "Here we go again"
   foo=$(($foo+1))
done
exit 0
```

What are the results?

until语句

```
until condition
do
    ....
done
```

```
#!/bin/sh
x=1
until [ "$x" -ge 20 ]
do
  echo $x
  x=$(($x+1))
done
exit 0
```

```
#!/bin/sh
until who | grep -q "$1"
do
   sleep 6
done
echo "*** $1 has just logged in ***"
```

What is the purpose of this script?
How to use it?

select语句

- 形式

```
select item in itemlist
do
    statements
done
```

- 作用
  - 生成菜单列表
- 举例：一个简单的菜单选择程序

```
#!/bin/sh
clear
select item in Continue Finish
do
   case "$item" in
      Continue) ;;
      Finish) break ;;
      *) echo "Wrong choice! Please select again!" ;;
   esac
done
```

- Question: 用while语句模拟?

命令组合语句

- 分号串联
  - command1 ; command2 ; …
- 条件组合
  - AND命令表

    格式：statement1 && statement2 && statement3 && …
  - OR命令表

    格式：statement1 || statement2 || statement3 || …

AND（&&）命令表

statement1 && *statement2* && *statement3* && . . .

从左边开始执行，每个语句都被被执行，如果当前语句执行返回结果为真，则继续执行右边的语句。该过程将持续到语句返回假。

```
$ test1=yes
$ test2=no
$ [ $test1 = yes ] && echo 1 && [ $test2 = yes ] && echo 2
1
$
```

AND命令表举例

```
#!/bin/sh
if [ -f $1 ] && [ -f $2 ]
then
     echo "Both $1 and $2 found"
else
     echo "File missing"
     exit 1
fi
exit 0
```

Script name is **check**. What are the result?

```
$ check /etc/passwd /etc/shadow
$ check /bin/bash /bin/ash
```

OR（||）语句

statement1 || statement2 || statement3 || ....

从左边开始执行，每个语句都被被执行，如果当前语句执行返回结果为真假，将继续执行右边的语句。该过程将持续到语句返回真。

```
#!/bin/sh
rm -f file_one
if [ -f file_one ] || echo "1" || echo "2"
then
        echo "in if"
else
        echo "in else"
fi
exit 0
```

What is the result?

函数

- 形式

```
func()
{
    statements
}
```

- 局部变量
  - local关键字
- 函数的调用

```
func para1 para2 …
```

- 返回值
  - return

函数举例1

```
#!/bin/sh
yes_or_no(){
   echo "Is your name $* ?"
   while true
   do
       echo -n "Enter yes or no: "
       read x
       case "$x" in
           y | yes ) return 0;;
           n | no ) return 1;;
           * )   echo "Answser yes or no"
       esac
   done
}
if yes_or_no "$1"
then
   echo "Hi $1, nice name"
else
   echo "Never mind"
fi
exit 0
```

如果文件名为“verify”?如何执行该脚本?

函数举例2

```
yesno()
{
    msg="$1"
    def="$2"
    while true; do
        echo" "
        $echol "$msg"
        read answer
        if [ "$answer" ]; then
            [redacted]
        else
            return $def
        fi
    done
}
```

```
case "$answer" in
    y|Y|yes|YES)
        return 0
        ;;
    n|N|no|NO)
        return 1
        ;;
    *)
        echo " "
        echo "ERROR: Invalid
response, expected \"yes\" or
\"no\"."
        continue
        ;;
esac
```

函数举例2的使用

- 调用函数yesno

```
if yesno “Continue installation? [n]” 1 ; then
    :
else
    exit 1
fi
```

变量范围

一般而言，除参数($1, $2, $@, etc)外，shell函数没有范围

```
#!/bin/sh
myfunc()
{
echo "myfunc() parameters : $@"
x=2
}
### Main script starts here
echo "Script parameters : $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

以“a b c”做参数时的运行结果？

```
Script parameters a b c
x is 1
myfunc() parameters: 1 2 3
x is 2
```

其他命令

- 杂项命令
  - break, continue, exit, return, export, set, unset, trap, “:”, “.”, …
- 捕获命令输出
- 算术扩展
- 参数扩展
- 即时文档

杂项命令

- break: 从for/while/until循环退出
- continue: 跳到下一个循环继续执行
- exit n: 以退出码”n”退出脚本运行
- return: 函数返回
- export: 将变量导出到shell，使之成为shell的环境变量
- set: 为shell设置参数变量
- unset: 从环境中删除变量或函数
- trap: 指定在收到操作系统信号后执行的动作
- “:”(冒号命令): 空命令
- “.”(句点命令)或source: 在当前shell中执行命令

break命令使用示例

- 从for/while/until循环退出

```
#!/bin/sh
for file
do
    if [ -d "$file" ]; then
        break;
    fi
done
echo the first directory is $file
```

continue命令使用示例

- 跳到下一个循环继续执行

```
#!/bin/sh
for file
do
    if [ -d "$file" ]; then
      continue
    fi
    mv "$file" "$(file)_file"
done
exit 0
```

continue命令使用说明

❖还有一个可选参数，用于指定从哪一级循环继续

```
Before 1
Before a
Before 2
Before a
Before 3
Before a
```

```
n/sh
 in 1 2 3

o Before $x
 y in a b c
  do
   echo Before $y
   continue 1
   echo After $y
  done
  echo After $x
done
```

```
#!/bin/sh
for x in 1 2 3
do
  echo Before $x
  for y in a b c
  do
   echo Before $y
   continue 2
   echo After $y
  done
  echo After $x
done
```

exec命令

- 调动另一个程序来替换当前shell

```
#!/bin/sh
echo "Before exec"
exec sh ./first.sh
echo "After exec"
exit 0
```

eval命令

- 计算参数
- 比较以下命令

```
$ foo=10
$ x=foo
$ y='$'$x
$ echo $y
$foo
$
```

```
$ foo=10
$ x=foo
$ eval y='$'$x
$ echo $y
10
$
```

:命令

- 空命令，相当于**true**
- 在**while**循环中
  - 实现无限循环，等价于**while true**

export命令

- 使变量在子shell中可以使用
- 考虑下例：demo运行的结果是什么？

Script: **showVar**

```
#!/bin/sh
echo $foo
echo $bar
```

Script: **demo**

```
#!/bin/sh
foo="Hello"
export bar="Hello"
sh showVar
```

expr命令

- 将参数作为表达式来计算

```
$ expr 8 + 6
14
$ x=`expr 12 / 4 `
$ echo $x
3
```

递归函数

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x(x-1)! & \text{otherwise} \end{cases}$$

```
#!/bin/sh
factorial()
{
      if [ "$1" -eq "0" ]; then
            echo 1
      else
             i=`expr $1 - 1`
             j=`factorial $i`
             k=`expr $1 \* $j`
             echo $k
       fi
}
while :
do
      echo -n "Enter a number:"
      read x
      factorial $x
done
```

echo –e 命令

❖ 允许使用转义符

```
$ echo -e "\tHello\nWorld"
        Hello
World
$
```

Escape	Description
\\	Backslash character
\a	Alert (ring the bell or beep)
\b	Backspace character
\f	Form feed character
\n	Newline character
\r	Carriage return
\t	Tab character
\v	Vertical tab character
777	The single character with octal value 0*777*

set命令

- 设置运行中的shell的变量

```
$ date
Sat Jan 20 02:00:23 HKT 2007
$
```

- 考虑以下命令在带参数与不带参数时的运行结果

```
#!/bin/sh
set $(date)
echo $2
```

```
$ ./script
Jan
$ ./script Hello World
Jan
$
```

shift命令

- 将所有参数前移一位， 因此$2 将变成 $1, $3变成$2, 等等。
- 常用来扫描参数

```
#!/bin/sh
while [ "$1" != "" ]
do
    echo "-- $1"
    shift
done
exit 0
```

```
$ ./script you are good
-- you
-- are
-- good
$
```

trap命令

- 指定接收到POSIX信号时所采取的动作
- 形式： **trap *command signal***

Signal	Description
HUP or 1	Hang up; usually sent when a terminal goes off
INT or 2	Interrupt; usually sent by pressing Ctrl-C
QUIT or 3	Quit; usually sent by pressing Ctrl-\
ABRT or 6	Abort; usually sent on some serious execution error
ALRM or 14	Alarm; usually used for handling time-out
TERM or 15	Terminate; usually sent by the system when it's shutting down

trap命令举例

```
#!/bin/sh
trap 'echo -e "\nHow dare you!" ' SIGINT
x=0
until [ $x -lt 0 ]
do
   echo -n "Enter an integer: "
   read x
done
```

```
$ sh script
Enter an integer: 3
Enter an integer: 4
Enter an integer:                Ctrl C
How dare you!
Ctrl C
How dare you!
-1
$
```

unset命令

- 从环境中删除变量或者函数

```
$ foo="Hello World"
$ echo $foo
Hello World
$ unset foo
$ echo $foo

$
```

参数扩展

❖方法

```
#!/bin/bash
for (( i=1; $i<=4 ; i=$i+1 ))
do
   touch ${i}_tmp
done
```

参数扩展说明

Parameter Expansion	Description
`${param:-default}`	If `param` is null, set it to the value of `default`
`${#param}`	Gives the length of `param`
`${param%word}`	From the end, removes the smallest part of `param` that matches `word` and return the rest
`${param%%word}`	From the end, removes the longest part of `param` that matches `word` and return the rest
`${param#word}`	From the beginning, removes the smallest part of `param` that matches `word` and return the rest
`${param##word}`	From the beginning, removes the longest part of `param` that matches `word` and returns the rest

参数扩展处理举例

```
#!/bin/sh
unset foo
echo ${foo:-bar}
foo=fud
echo ${foo:-bar}
foo=/usr/bin/X11/startx
echo ${foo#*/}
echo ${foo##*/}
bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}
exit 0
```

运行结果

```
bar
fud
usr/bin/X11/startx
startx
/usr/local/etc
/usr
```

其他参数替换

Parameter Expansion	Description
`${param+default}`	Replace by `word` if `param` is set, and noting otherwise
`${param=word}`	Assigns `word` to the variable `param` if `param` is not already set and then is replaced by the value of `name`.
`${param?word}`	Replace by `$param` if `param` is set. If `param` is not set, `word` is displayed to the standard error channel and the shell is existed. If `word` is omitted, then a standard error message is displayed instead

Shell脚本编程示例1

- 显示给定目录下指定文件内容的Shell脚本
- 执行此Shell脚本时
  - 如果第1个位置参数是合法的目录，那么就把后面给出的各个位置参数所对应的文件显示出来
    - ✓ 若给出的文件名不正确，则显示出错信息
  - 如果第1个位置参数不是合法的目录，则显示目录名不对

Shell脚本编程示例1

- **shellprogram1**

```
dir=$1;shift
if[ -d $dir ]
then
      cd $dir
      for name
      do
        if [ -f $name ]
       then  cat $name
            echo " end of ${dir}/$name"
      else  echo " invaild file name:${dir}/$name"
      fi
    done
else echo " bad directory name:${dir}"
fi
```

Shell脚本编程示例2

- 从命令行输入简单的算术表达式并计算结果
  - shellprogram2

```
if [ $# = 3 ]
then
   case $2 in
+) let z=$1+$3;;
-) let z=$1-$3;;
/) let z=$1/$3;;
x | X) let z=$1*$3;;
*) echo " Warning-$2 invalied operator, only +、-、×、÷" operator allowed.
  exit;;
esac
echo " Answer is $z"
else
     echo " Usage-$0 value1 operator value2."
fi
```

Shell应用举例

❖检测某台机器是否关机

```
#!/bin/sh
# [Usage] isAlive.sh ccbsd1

Usage="[Usage] $0 host"
temp="$1.ping"
Admin="chwong"
count="20"
if [ $# != 1 ] ; then
  echo $Usage
else
  /sbin/ping -c ${count:=10} $1 | /usr/bin/grep 'transmitted' > $temp
  Lost=`awk –F" " '{print $7}' $temp | awk –F"%" '{print $1}' `

  if [ ${Lost:=0} -ge 50 ] ; then
    mail –s  "$1 failed" $Admin < $temp
  fi
  /bin/rm $temp
fi
```

主要内容

- 背景知识
  - Shell基础
  - Shell编程
- 实验内容
  - 编写一个简单的Shell程序—Myshell
  - 基于Shell的网络管理

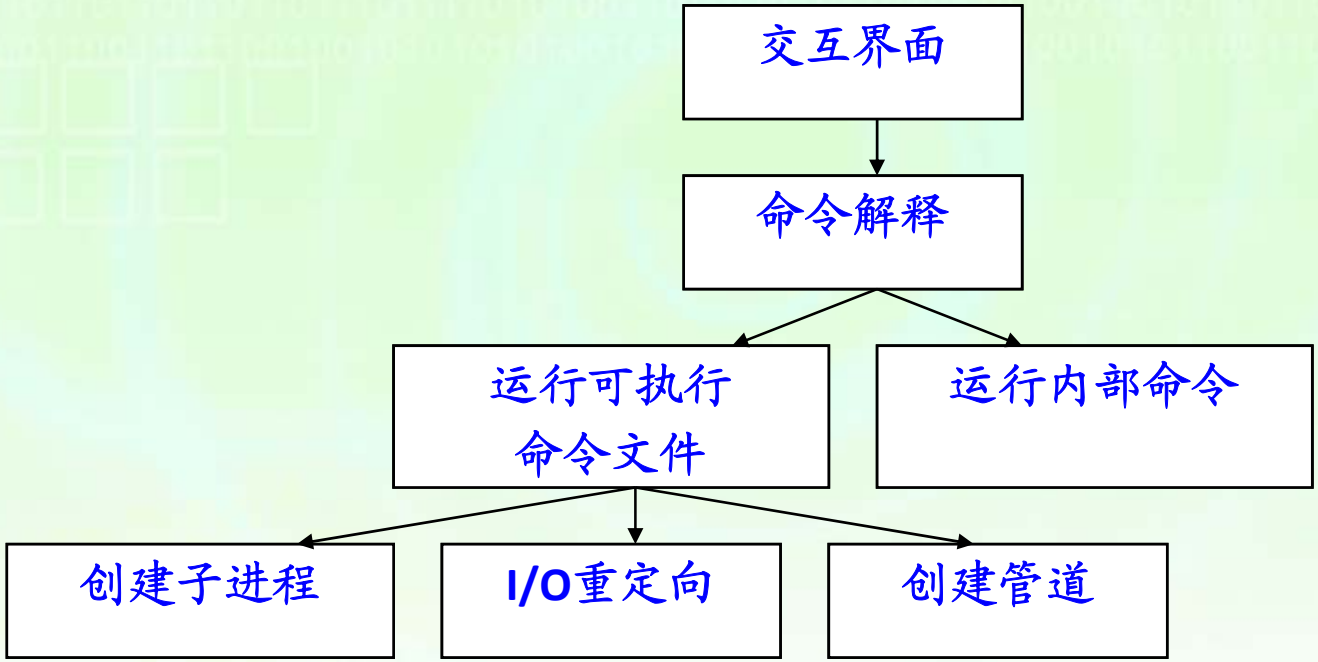
编写一个简单的Shell程序—Myshell

❖实验说明

- 设计并实现一个简单的交互式Shell，使其具备如下功能
  - ✓支持程序后台运行
  - ✓支持重定向
  - ✓支持管道线
  - ✓支持设定搜索路径
  - ✓支持内置命令：cd（切换目录），exit（退出Shell），path（设定搜索路径）

编写一个简单的Shell程序—Myshell

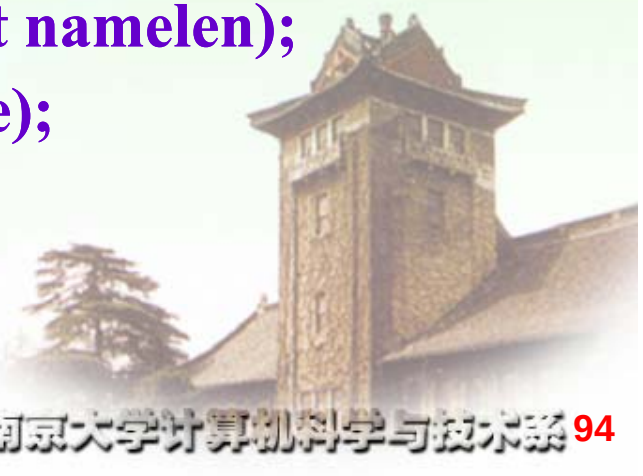
❖解决方案：总体框架



编写一个简单的Shell程序—Myshell

❖解决方案：交互界面显示

- 通常Shell会在命令提示符中根据用户的配置文件显示响应信息
- 只需要固定显示当前用户关联的登陆名、主机网络名、当前目录名即可。显示格式为:
  - ✓ [username@servername:pathname]#
- 涉及函数
  - ✓ char *getlogin(void);
  - ✓ int gethostname(char *name, size_t namelen);
  - ✓ char *getcwd(char *buf, size_t size);



编写一个简单的Shell程序—Myshell

❖解决方案：命令解析

- 为了得到命令行，MyShell执行一个阻塞型读操作
  - ✓执行MyShell进程时将会被阻塞，直到用户根据提示符输入又一个命令行
- 可通过gets( )获得用户输入的命令行
- 在获得用户的输入后，需要对输入进行解析，以获得命令和参数
- 在解析过程中，需要注意
  - ✓判断命令是否是内置命令
  - ✓是否包含“<”,“>”等字符，表明需要进行重定向
  - ✓是否包含“&”字符，表明命令要放入后台执行
  - ✓是否包含“|”字符，表明有多个命令，并要创建管道

编写一个简单的Shell程序—Myshell

❖解决方案：内置命令

- cd命令：可通过chdir函数更改当前目录
  - ✓int chdir(const char *pathname);
- path命令：用于设置MyShell的搜索路径
  - ✓在MyShell中，需要用一个全局变量gpath用于记录用户设置的搜索路径
  - ✓当用户设置新搜索路径时，MyShell对gpath变量进行更新
- exit命令：可通过exit函数退出程序
  - ✓void exit(int status);

编写一个简单的Shell程序—Myshell

❖解决方案：执行命令

- 如果用户输入的不是内置命令，MyShell需要在用户设置的路径中搜索命令，并测试该命令是否可执行
- 可以通过access( )函数进行测试
  - ✓int access(const char *pathname, int mode);
- 执行一个命令通常使用fork( )和exec( )
  - ✓pid_t fork(void);
  - ✓int execve(const char *pathname, char *const argv[], char *const envp[]);
- 如果需要将程序放在前台运行，则父进程需要等待子进程运行结束，可以通过执行waitpid( )函数等待子进程运行结束
  - ✓pid_t waitpid(pid_t pid, int *statloc, int options);

编写一个简单的Shell程序—Myshell

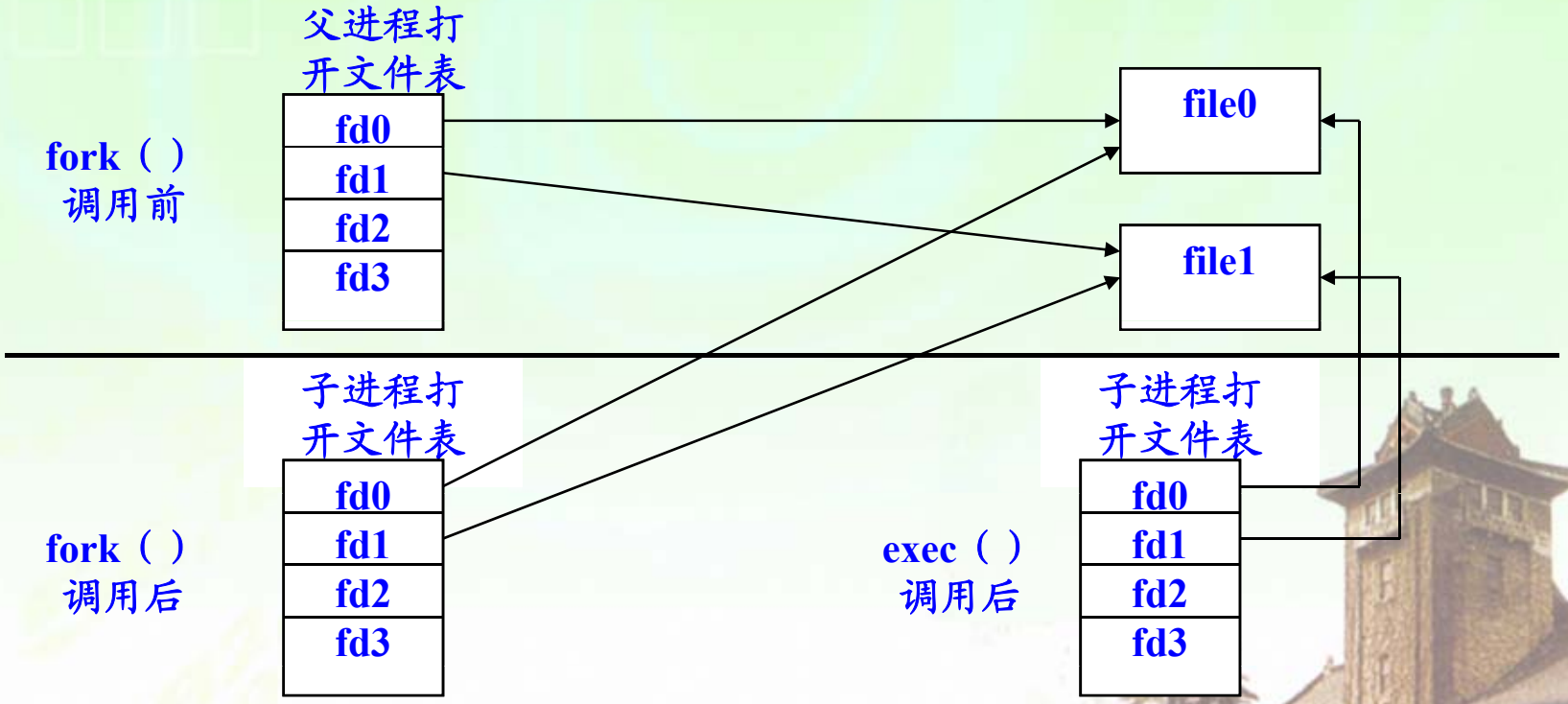
❖解决方案：执行命令代码框架

```
if  ((pid=fork( ))==0 ){ /*子进程*/
    execve(path, argv, envp);
}else{ /*父进程*/
    if ( forground )     /*前台运行*/
                waitpid( pid, &status, 0 ); /*等待子进程退出*/
}
```

编写一个简单的Shell程序—Myshell

❖解决方案：I/O重定向

- 已打开文件描述符将会在fork( )和exec( )调用中保持下来
  - ✓利用这一点可以实现文件的输入、输出重定向



父进程、子进程与打开文件表关系

编写一个简单的Shell程序—Myshell

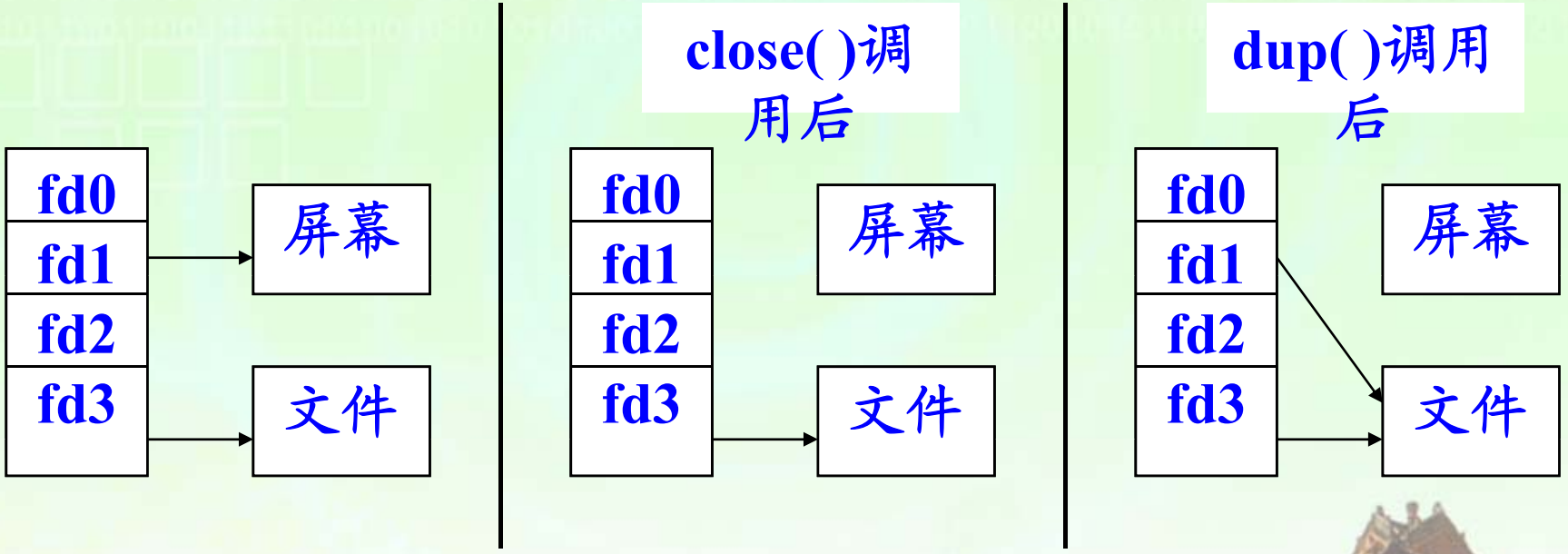
❖解决方案：I/O重定向

- 完成I/O重定向涉及到的函数
  - ✓int dup(int filedes);
  - ✓int dup2(int filedes, int filedes2);
- dup( )函数将指定的文件描述符filedes复制到当前第1个可用的文件描述符中
- 如果需要使用dup( )函数完成I/O重定向，需要先将标准输入或输出关闭，例如:

```
fid = open( foo, O_WRONLY|O_CREAT);
close(1);
dup(fid);
```

编写一个简单的Shell程序—Myshell

❖解决方案：I/O重定向



文件描述符变化过程

编写一个简单的Shell程序—Myshell

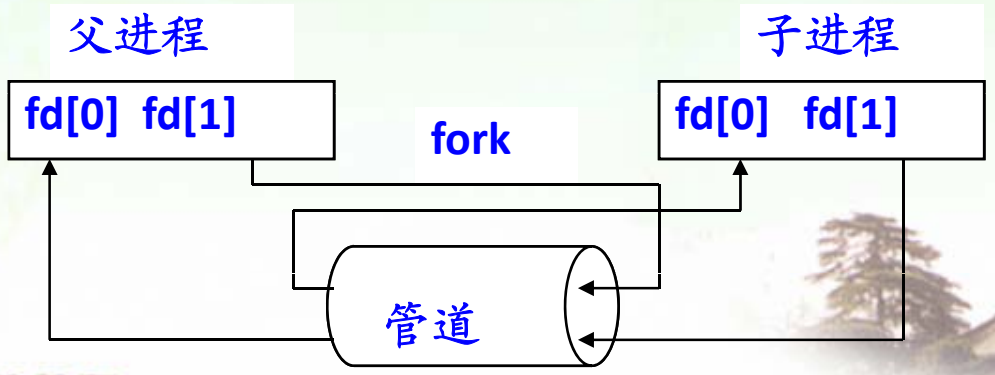
❖解决方案：I/O重定向

- 代码框架

```
if  ((pid=fork( ))==0 ){/*子进程*/
        /*实现输出重定向*/
        fid = open( foo, O_WRONLY|O_CREAT);
        close(1);
        dup(fid);
        execve(path, argv, envp);
}else{/*父进程*/
        if ( forground )        /*前台运行*/
        waitpid( pid, &status, 0 ); /*等待子进程退出*/
}
```

编写一个简单的Shell程序—Myshell

- 解决方案：管道
  - 通常，一个管道由一个进程创建，然后该进程调用fork( )，此后父、子进程之间就可以使用管道
  - 管道涉及到的函数
    - ✓ int pipe(int fd[2]);
  - 单个进程中的管道几乎没有任何用处
    - ✓ 通常，调用pipe( )的进程接着调用fork( )，创建从父进程到子进程或反之的管道

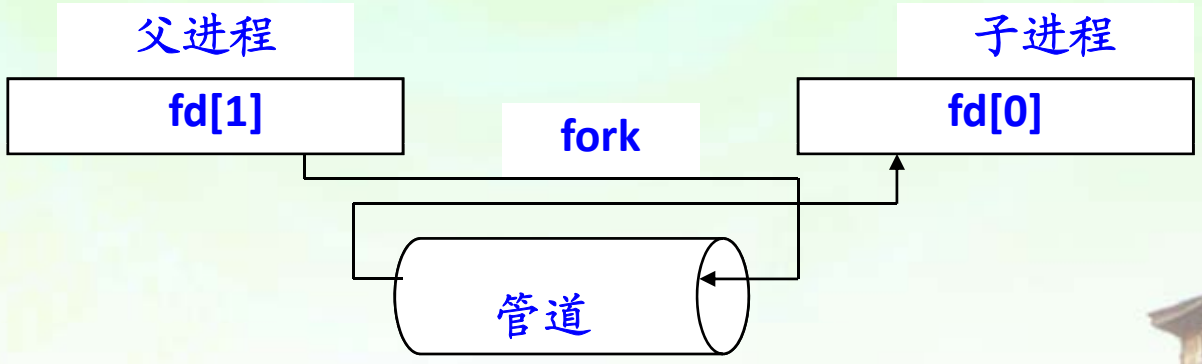


fork( )之后的半双工管道

编写一个简单的Shell程序—Myshell

❖解决方案：管道

- fork( )之后做什么取决于所需要的数据流的方向
  - ✓对于从父进程到子进程的管道，父进程关闭管道的读端（fd[0]），子进程则关闭写端（fd[1]）



从父进程到子进程的管道

编写一个简单的Shell程序—Myshell

❖解决方案：管道

- MyShell需要实现的管道是将进程1的标准输出变成进程2的标准输入，实现该功能需要的步骤为
  - ✓创建进程1
  - ✓在进程1中调用pipe( )创建管道，并得到管道描述符fd[0],fd[1]
  - ✓在进程1中创建进程2，使进程2成为进程1的子进程；
  - ✓进程1关闭描述符fd[0]
  - ✓进程1调用dup2(fd[1],1)，将管道描述符复制到标准输出
  - ✓进程1调用exec( )执行新程序；
  - ✓进程2关闭描述符fd[1]
  - ✓进程2调用dup2(fd[0],0)，将管道描述符复制到标准输入
  - ✓进程2执行exec( )执行新程序

编写一个简单的Shell程序—Myshell

❖解决方案：管道

- 管道处理代码框架

```
{
        ……
        int fd[2];
        pipe(fd);
        if ( fork( )>0 ) {/*父进程*/
                close(fd[0]);
                dup2(fd[1],1);
                exec(……);
        }else{
                close(fd[1]);
                dup2(fd[0],0);
                exec(……);
        }
        ……
}
```

主要内容

- 背景知识
  - Shell基础
  - Shell编程
- 实验内容
  - 编写一个简单的Shell程序—Myshell
  - 基于Shell的网络管理

基于Shell的网络管理

❖实验说明

- 通过Shell编程访问某个URL（如www.nju.edu.cn）N次
- 在网关节点上将指定MAC地址（如00:11:5B:A3:65:F8）的机器在某个网管上的进出

❖解决方案

- 对URL的访问可通过方法get实现，访问次数可通过while语句实现
- 对arp地址的获取可通过arp实现
- 对指定MAC地址的访问则可通过grep得到
- 对IP地址的提取可通过awk和sed工具实现

第5章 Shell程序设计