Operating System Principle, OS

2025年秋. 操作系统原理

第03章 用户界面与系统调用

课 程 组：邹德清, 李珍, 李志, 苏曙光

企业教师：华为认证专家（鸿蒙方向）

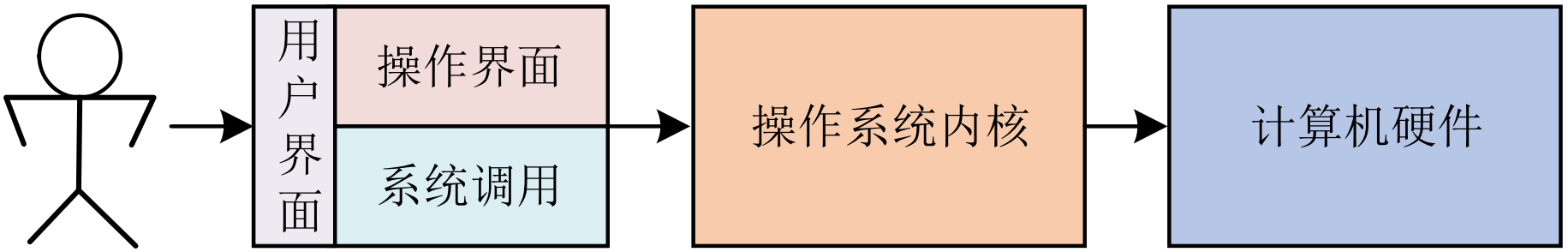
用户界面

- 主要内容
  - 操作界面
  - 系统调用
- 重点
  - 批处理和Shell脚本编程
  - 系统调用机制

3.1 用户界面

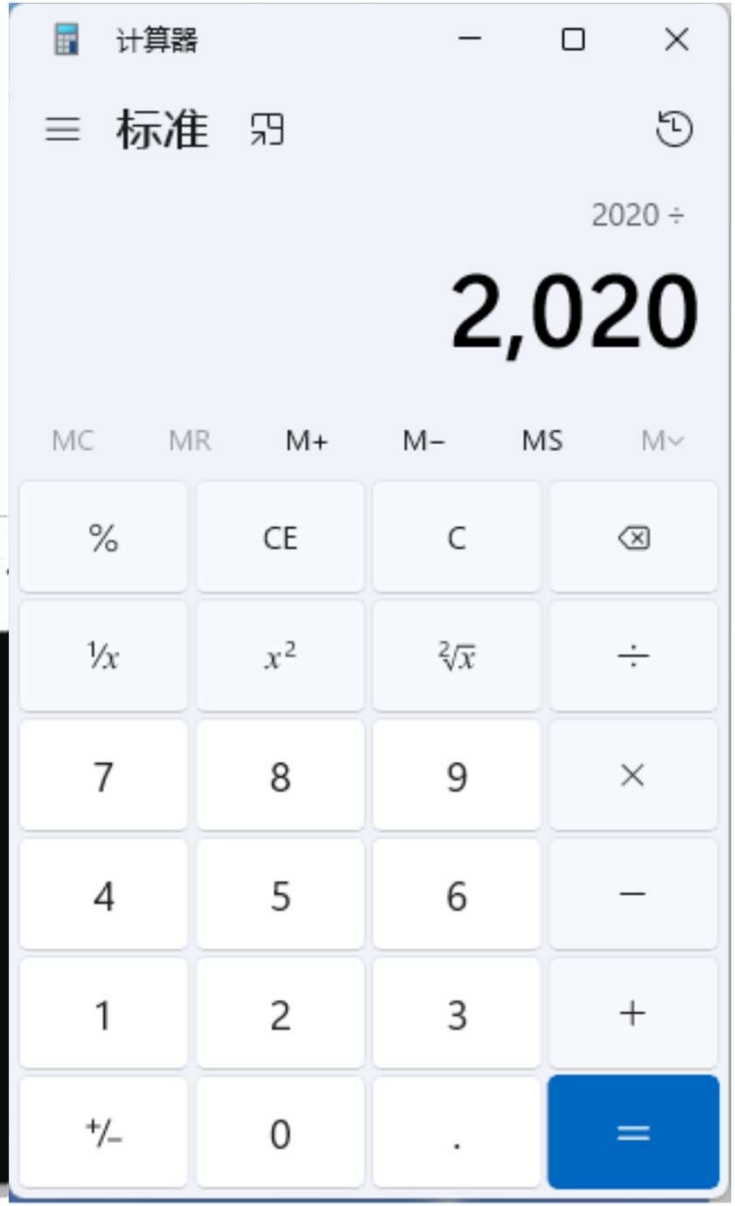
用户界面（User Interface）

- 用户界面的定义
  - 操作系统提供给用户控制计算机的机制(用户接口)
- 用户界面的类型
  - 操作界面
  - 系统调用 （System Call，系统功能调用,程序界面）



操作界面

- 操作界面的类型
  - 图形用户接口/GUI
  - 操作命令（普通命令）
  - 批处理与脚本程序



操作命令（普通命令）

- **DOS典型命令**
  - 文件管理
    - ◆COPY、COMP、TYPE、DEL、REN
  - 磁盘管理
    - ◆FORMAT、CHKDSK、DISKCOPY、DISKCOMP
  - 目录管理
    - ◆DIR、CD、MD、RD、TREE
  - 设备工作模式
    - ◆CLS、MODE
  - 日期、时间、系统设置
    - ◆DATE、TIME、VER、VOL
  - 运行用户程序
    - ◆MASM、LINK、DEBUG

操作命令（普通命令）

- **Linux典型命令**

命令	作用	命令	作用
ls	列举子目录和文件	find	查找文件
ps	列举进程	whereis	查找文件目录
top	列举进程	man	查看命令帮助信息
echo	输出字符串	cp	拷贝
cat	读文件的内容	inode	查看文件节点
cd	改变目录	tar	压缩和解压
chmod	改变文件属性	rm	删除文件和文件夹
mount	挂载文件系统	umount	卸载文件系统
insmod	安装模块	rmmod	卸载模块

操作界面

- 类型
  - 图形用户接口
  - 操作命令（普通命令）
  - 批处理与脚本程序：自动处理一批命令

DemoFor.bat

Windows/DOS

```
1  @echo off
2  for  %%C  in (HUST,SSE,2020) do (
3      echo %%C
4  )
5  pause
```

操作界面

- 类型
  - 图形用户接口
  - 操作命令（普通命令）
  - 批处理与脚本程序：自动处理一批命令

LINUX

```
#!/bin/bash
read -n 1 -p "Enter your choice(Y/N):" answer
echo
case "$answer" in
Y|y)
    echo "Yes";;
N|n)
    echo "No";;
```

```
ubuntu@VM-4-6-ubuntu:~/bash$ ./TestYesNO.sh
Enter your choice (Y/N): Y
Yes
```

操作界面

- **Windows批处理例子：求1..100的和并输出**

Sum_1_100.bat

```
1  @echo off
2  for /l %%i in (1,1,100) do (
3      set /a sum = sum + %%i
4  )
5  echo %sum%
6  pause
```

- **批处理的特点**
  - 文本文件 | 文件名后缀*.BAT
  - 普通命令的集合，按批执行，由command解释执行
  - 支持变量替换、条件、转移、循环、注释等简单语法

操作界面

- **Windows批处理例子(带参数)：编译驱动程序**

CMD>MakeDrvr　DDK根路径　源程序所的在盘符　源程序的路径　发行/调试版

文件名：MakeDrvr.bat

```
1  #编译DDK驱动程序的批处理程序命令: MakeDrvr.bat
2  @echo off                          #关闭回显
3  if "%1"=="" goto usage             #若缺第1个参数，则提示用法
4  if "%3"=="" goto usage             #若缺第2个参数，则提示用法
5  if not exist %1\bin\setenv.bat goto usage #若参数1路径不存在相应文件，则提示用法
6  call %1\bin\setenv %1 %4           #利用call调用相应命令和参数：准备驱动程序编译环境
7  %2                                 #第2个参数是驱动源代码所在盘符，例如 D: ,切换盘符到D:
8  cd %3                              #第3个参数是驱动源代码所在路径。例如 D:\User，切换路径
9  build -b -w %5 %6 %7 %8 %9          # 执行build命令，编译驱动程序
10 goto exit                          # 编译完毕，退出批处理程序
11 :usage                             # 提示批处理程序的正确用法
12 echo usage MakeDrvr DDK_dir Driver_Drive Driver_Dir free/checked
13 [build_options]
14 echo eg MakeDrvr %%DDKROOT%% C: %%WDMBOOK%% free -cef
15 :exit
```

操作界面

- **Windows批处理程序编写参考**
  - http://wenku.baidu.com/view/08bd3c7687c24028915fc380.html?from=search

操作界面

- Linux脚本程序例子：安装或更新软件包

install.shell

文件名：Install.sh

```
1  #!/bin/bash
2  #创建临时文件
3  sudo mkdir /usr/temp
4  #解压安装包到临时文件
5  sudo echo "正在解压文件"
6  sudo unzip -qd /usr/temp /HUSTLibV30.zip
7  sudo echo "解压完成"
8  #拷贝安装文件
9  sudo cp -rf /usr/temp/HUSTLibV30/HUSTLib /usr/lib
10 #使配置文件生效
11 sudo ldconfig
12 #删除临时文件
13 sudo echo "正在删除临时文件"
14 sudo rm -rf /usr/temp
15 sudo echo "删除临时文件成功"
16 sudo echo "安装完成请重启"
```

操作界面

- Linux脚本程序例子：比较输入的两个整数的大小

文件名：Bigger.sh

```
#!/bin/bash
echo "Please Enter Two Number"
read a
read b
if test $a -eq $b
then
  echo "NO.1 = NO.2"
elif test $a -gt $b
then
  echo "NO.1 > NO.2"
else
  echo "NO.1 < NO.2"
fi
```

操作界面

- Linux脚本程序
  - 运行脚本程序的方法
    - ◆方法1：直接运行（用缺省版本Shell解释执行脚本）

      $./test.sh ↙
    - ◆方法2：在命令行上指定特定版本Shell执行脚本

      $bash ./test.sh ↙
    - ◆方法3：在脚本首行指定特定Shell执行当前脚本

test.sh

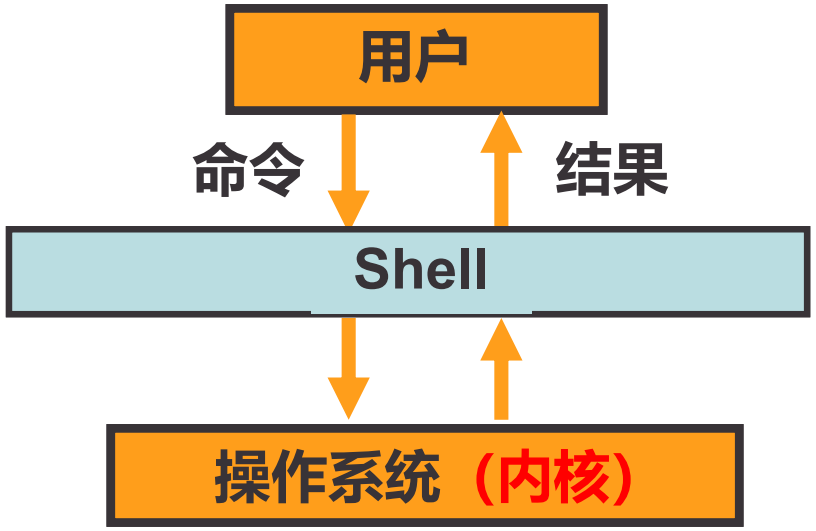
```
1 #!/bin/bash
2 cd ~
3 mkdir shell_test
4 cd shell_test
5 for ((i=0; i<10; i++)); do
6     touch test_$i.txt
7 done
```

操作界面

- **Linux**脚本程序参考
  - ttp://wenku.baidu.com/view/15822fc2fd0a79563c1e72be.html?from=search

Linux Shell (Windows CMD)

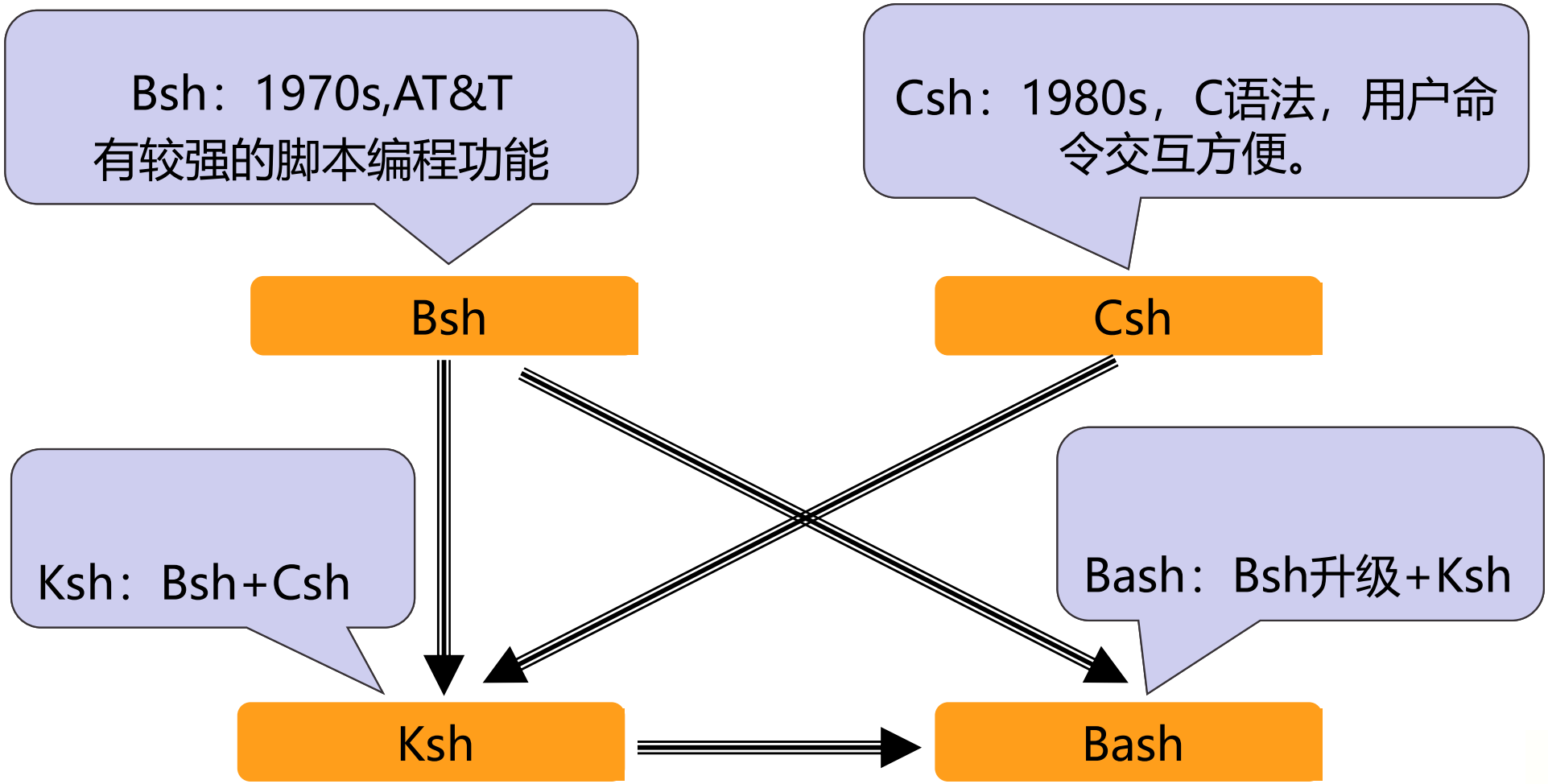
- Shell是操作系统与用户的交互机制（操作界面）



- 通过Shell(/控制台)执行用户命令
- 组织和管理用户命令的执行和结果展示

Linux Shell（类似Windows CMD）

- **Shell的发展与分类**



Linux Shell

- **Bash主要功能**
  - 命令行编辑功能
  - 命令和文件名补全功能
  - 命令历史功能
  - 命令别名功能
  - 作业控制功能
  - 将命令序列定义为功能键
  - 支持脚本程序
  - 重定向与管道

重定向的例子（Linux）

- **输出重定向：把命令的输出重定向到其它文件**
  - 将标准输出重定向到特定文件
    $ ls /etc/ > etc.log
  - 将标准输出重定向/追加到特定文件
    $ ls /etc/sysconfig/ >> etc.txt
  - 将错误输出重定向到特定文件
    $ ErrCmd 2> err.log
  - 将标准输出和错误输出重定向到特定文件
    $ AComand &> ErrFile
    $ AComand > AComand.out 2>&1

$ ErrCmd 2> /dev/null

重定向的例子（Linux）

- 输入重定向：把命令的输入来源改为其它文件

```
susg@susg311: ~
susg@susg311:~$ cat a.sh   // cat 输出文件内容
echo hello world !
susg@susg311:~$ /bin/bash < a.sh
hello world !
susg@susg311:~$ cat < a.sh
echo hello world !
susg@susg311:~$
```

D:\a.sh - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

a.sh

```
1  echo hello world !
```

length : 18 Ln : 1 Col : 19 Sel : 0 | 0 Dos\Windows UTF-8 INS

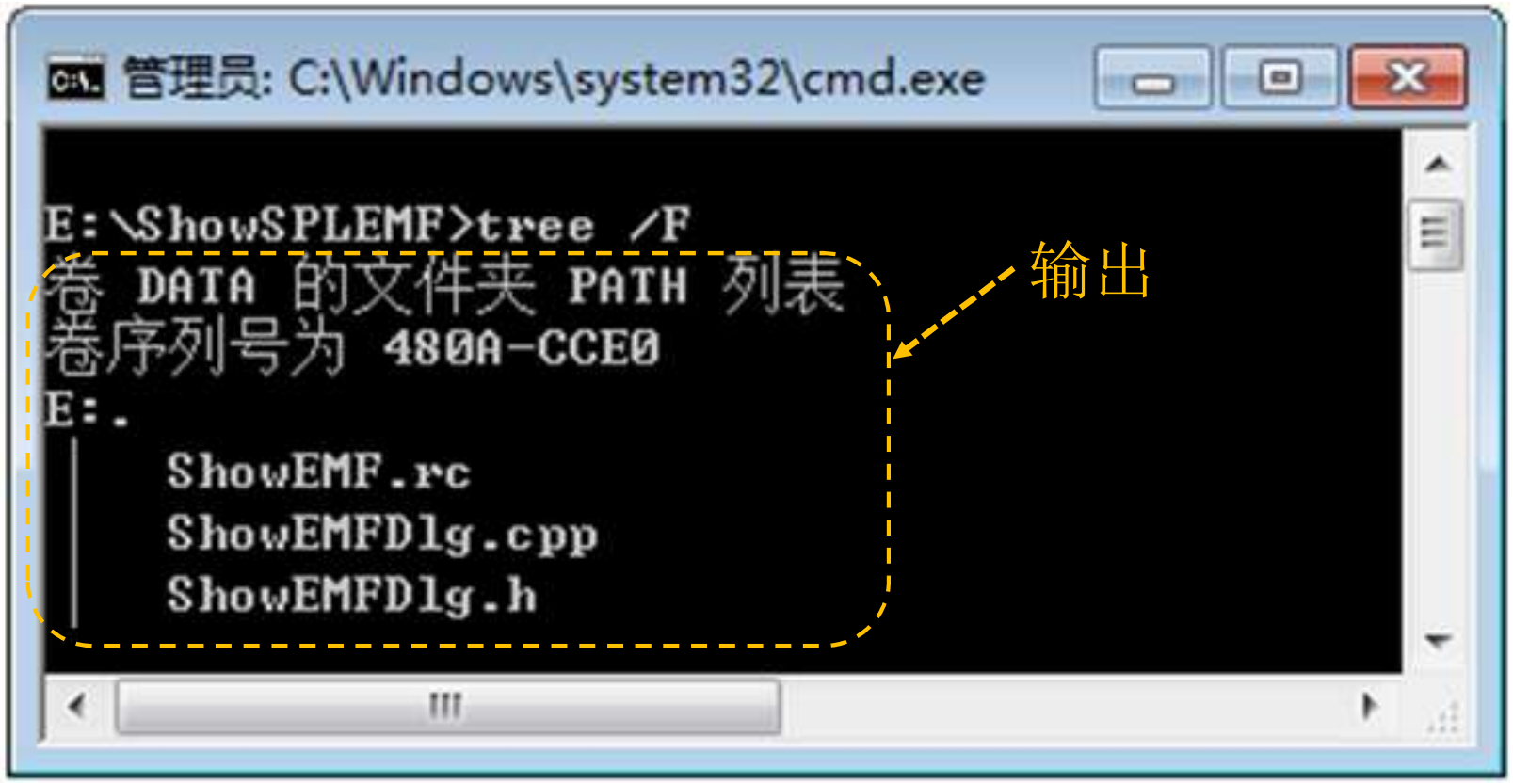
重定向操作（Linux）

- 把命令缺省输入来源或输出方向修改为其他文件/设备。

类别	操作符	说明
输入重定向	<	将命令输入由默认的键盘更改/重定向为指定的文件
输出重定向	>	将命令输出由默认的显示器更改/重定向为指定的文件
	>>	将命令输出重定向并追加到指定文件的末尾
错误重定向	2>	将命令的错误输出重定向到指定文件（先清空）。
	2>>	将命令的错误输出重定向到指定文件（追加到末尾）。
输出与错误组合重定向	&>	将命令的正常输出和错误输出重定向到指定文件。

重定向与管道

- 标准输入/输出设备




```
ubuntu@VM-4-6-ubuntu:~/bash$ ./TestYesNO.sh
Enter your choice (Y/N): N
No
```

输入

输出

标准输入/输出设备(缺省)

- 标准输入设备：键盘(stdin)
  - 程序的输入来源：文件0 ← stdin(缺省)

- 标准输出设备(含错误)：显示器(stdout和stderr)
  - 程序的输出去向：文件1/2 ← stdout/stderr(缺省)

输入/输出文件	设备	备注
标准输入文件	键盘stdin	缺省: → 文件0
标准输出文件	显示器stdout	缺省: → 文件1
标准错误输出文件	显示器stderr	缺省: → 文件2

```
ubuntu@VM-4-6-ubuntu:~/bash$ ./TestYesNO.sh
Enter your choice (Y/N): N
No
```

输入

输出

管道

- Linux管道的例子

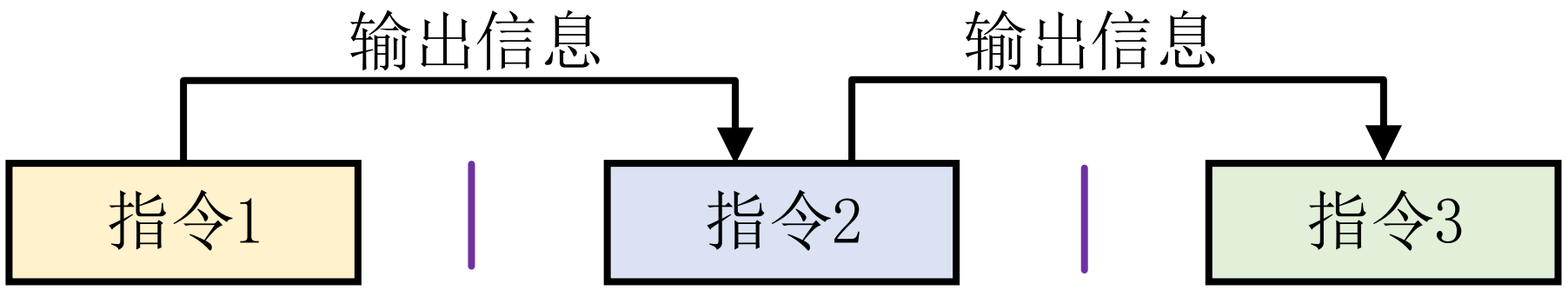
```
t@t-virtual-machine:~$ ls /etc
acpi                    hostname                ppp
adduser.conf            hosts                   profile
alsa                    hosts.allow             profile.d
alternatives            hosts.deny              protocols
anacrontab              hp                      pulse
apg.conf                ifplugd                 python3
apm                     init                    python3.10
apparmor                init.d                  rc0.d
apparmor.d              initramfs-tools         rc1.d
apport                  inputrc                 rc2.d
```

- ls /etc | wc -l

```
t@t-virtual-machine:~$ ls /etc | wc -l
223
```

管道

- Linux管道
  - 程序相连，一个程序的输出作为另一程序的输入
  - 管道操作符 |
    - “|”符用于连接左右两个命令，将|左边命令的执行结果（输出）作为|右边命令的输入



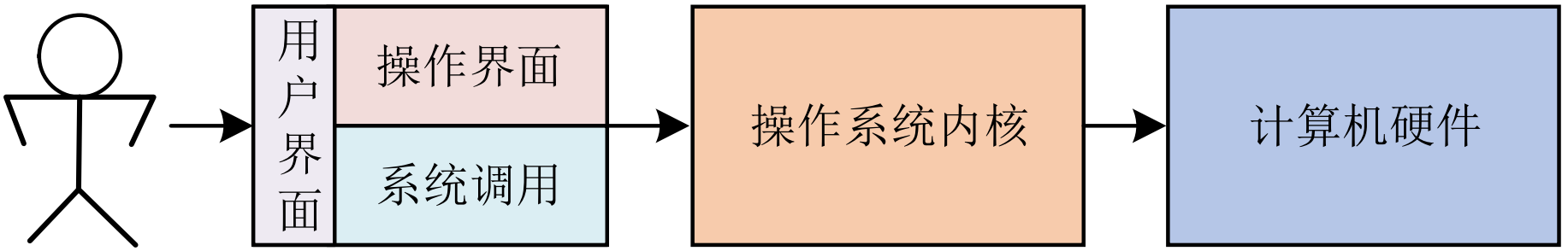
3.4 系统调用

应用程序

```
1   main()
2   {
3       int Var = 20;        // Var  DW   20   ;[1000:100]
4       Var = Var + 22;      // MOV  EAX, Var
5
6       int fd = open("\home\log.txt");
7       write(fd,"2023/03/09"); //写日志文件
8       close(fd);
9   }
10
11
12
```

用户界面(User Interface)

- 用户界面的定义
  - OS提供给用户控制计算机的机制，又称用户接口。
- 用户界面的类型
  - 操作界面
  - 系统调用（System Call，系统功能调用,程序界面）



例子1：Linux：两个整数相加：函数add（）

```
#include <stdio.h>
int add( int a, int b)
{
        return (a + b);
}

int main(void)
{
    int sum = add( 100, 300);
}
```

例子2：Linux：printf（）与exit（）

```
#include <stdio.h>

int main(void)

{

    printf( "Hello\n“ );

    exit( 0 );

}
```

- 特点：涉及显卡和进程操作

printf( )，exit( )与add( )比较

```
#include <stdio.h>

int main(void)

{

    printf( "Hello\n" );

    exit( 0 );
```

```
int fd = open("\home\log.txt");
write(fd,"2023/03/09"); //写日志文件
close(fd);
```

■ 特点：涉及显卡和进程操作
或硬件和内核资源操作

```
#include <stdio.h>
int add( int a, int b)
{
    return (a + b);
}

int main(void)
{
    int sum = add( 100, 300);
}
```

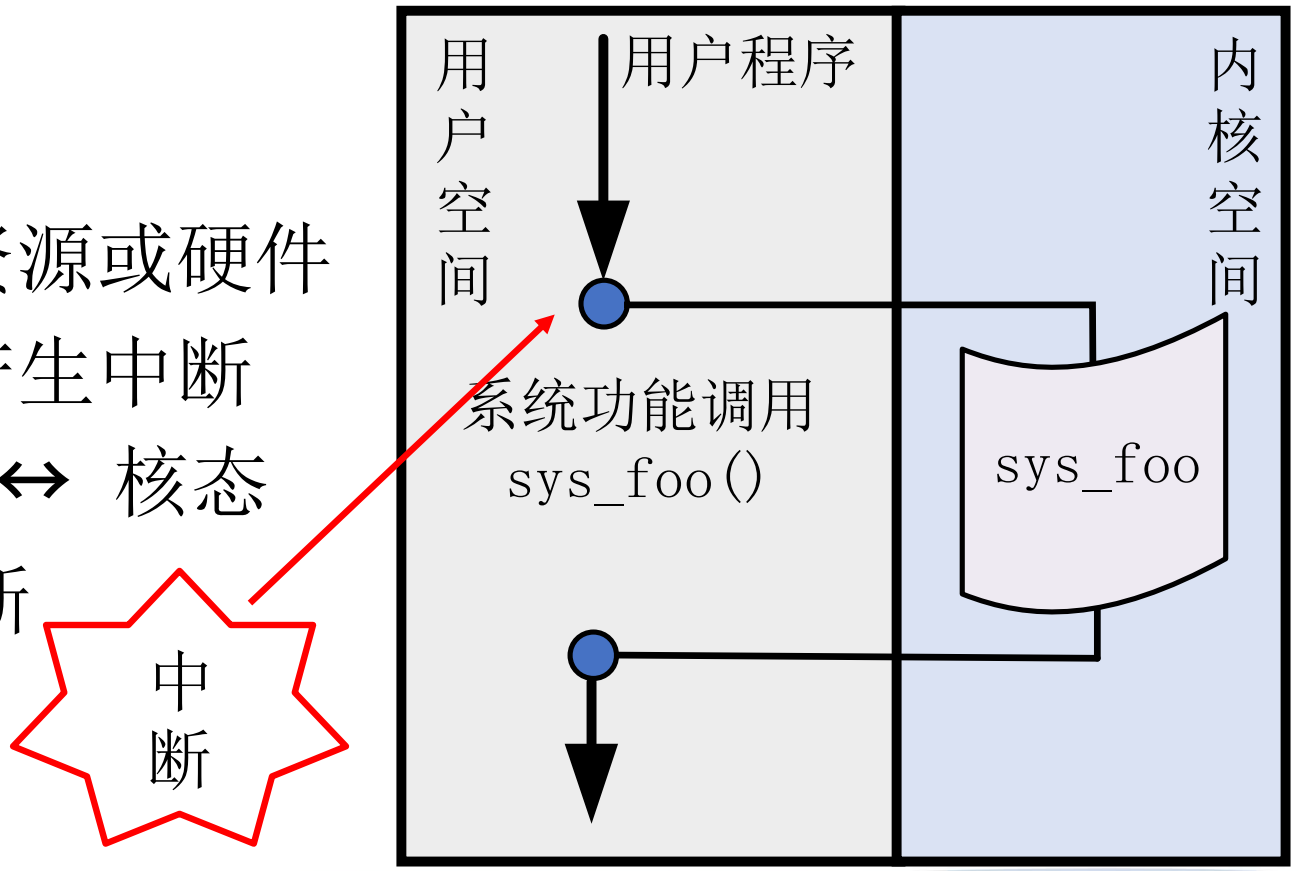
例子4：利用DOS 21H中断显示字符串（09号功能）

```
StrHello  DB   'Hello!'     ; 定义要显示的字符串
   ...
MOV DX,   StrHello          ; DX ← 字符串地址
MOV AH,   09H               ; AH ← 09h ： 功能编号
INT  21H                    ;DOS 21H中断
```

■ 特点：涉及外设（显卡）操作

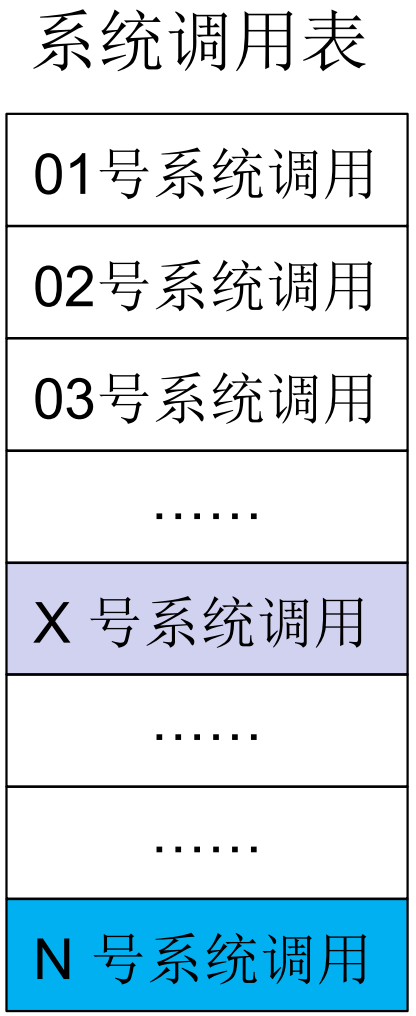
系统调用

- 系统调用(System Call，System Service Call)
  - 操作系统内核为应用程序提供的服务/函数。
    - 例：printf, exit, open, read, INT 21H(09)
  - 特点
    - 内核实现
    - 存取核心资源或硬件
    - 调用过程产生中断
      - 用户态 ↔ 核态
      - 自愿中断



系统调用表和调用形式

- 系统调用表
  - 全部系统调用的入口列表
    - 有序排列
  - 系统调用号：系统调用的唯一编号
- 系统调用的一般调用形式
  - 访管指令：`SVC X`
    - SVC = SuperVisor Call
    - X = 系统调用的编号



系统调用表和调用形式

- DOS的系统调用表(部分)

■ 01：键盘输入有回显	■ 39：建立子目录
■ 02：屏幕输出1个字符	■ 3A：删除子目录
■ 03：显示输出	■ 3B：改变当前目录
■ 04：串口输入1字符	■ 3C：创建文件
■ …	■ …
■ 08：键盘输入无回显	■ 3F：读文件
■ 09：显示字符串	■ 40：写文件
■ …	■ 4C：结束程序

系统调用的调用方式

- 例：DOS：9号系统调用（屏幕输出字符串）(输出Hello)

```
1  DATA SEGMENT
2    STR1 DB 'HOW DO YOU DO?',0DH,0AH,'$'
3  DATA ENDS
4
5  CODE SEGMENT
6    ASSUME CS:CODE,DS:DATA
7  START:
8          MOV AX,DATA
9          MOV DS,AX
10         MOV DX,OFFSET STR1 ;字符串
11         MOV AH,9
12         INT 21H;输出字符串
13
14         MOV AH,4CH
15         INT 21H
16 CODE ENDS
17   END START
```

- DOS系统调用(部分)
  - 01：键盘输入有回显
  - 02：屏幕输出1个字符
  - 03：显示输出
  - …
  - 08：键盘输入无回显
  - 09：显示字符串
  - …
  - 4C：结束程序

系统调用的调用方式

- 例：DOS：1号系统调用(键盘输入字符)(键盘输入1/2?)

```
1  InputKey:
2        MOV   AH, 1
3        INT   21H          ;等待输入一个字符
4        CMP   AL, '1'
5        JE    ONE          ;如果输入'1
6        CMP   AL, '2'
7        JE    TWO          ;如果输入'2
8        CMP   AL, '3'
9        JE    THREE        ;如果输入'3
10       JMP   InputKey     ;如果不是1,
11 ONE:    ......
12 TWO:    ......
13 THREE:  ......
```

- DOS系统调用(部分)
  - 01：键盘输入有回显
  - 02：屏幕输出1个字符
  - 03：显示输出
  - …
  - 08：键盘输入无回显
  - 09：显示字符串
  - …
  - 4C：结束程序

系统调用的调用方式

- DOS中系统调用的调用方式
  - DOS： INT 21H（利用AH存放系统调用的编号）

01：键盘输入有回显	39：建立子目录
02：屏幕输出1个字符	3A：删除子目录
03：显示输出	3B：改变当前目录
04：串口输入1字符	3C：创建文件
…	…
08：键盘输入无回显	3F：读文件
09：显示字符串	40：写文件
…	4C：结束程序

系统调用表和调用形式

- Linux的系统调用(部分)

■ 01：结束进程	■ 14：创建节点
■ 02：创建进程	■ 15：修改文件属性
■ 03：读文件	■ 39：创建目录
■ 04：写文件	■ 40：删除目录
■ 05：打开文件	■ 42：创建管道
■ 06：关闭文件	■ ……
■ ……	■ 48：安装信号处理
■ 12：改变目录	■ 52：解挂设备

系统调用的调用方式

- 例：Linux：4号系统调用(写文件) （屏幕输出字符串）

```
;输出字符串：Hello World!
MOV EBX, 1          ;EBX=1=stdout
MOV ECX, MSG        ;字符串首地址
MOV EDX, 13         ;字符串长度
MOV EAX, 4          ;系统调用编号（4 = write）
INT 80h             ;中断：输出字串
MSG: DB "Hello World!"
MOV EAX,1           ;系统调用编号（1 = exit ）
INT 80h             ;中断：结束进程
```

- 特点：利用EAX寄存器存放系统调用的编号

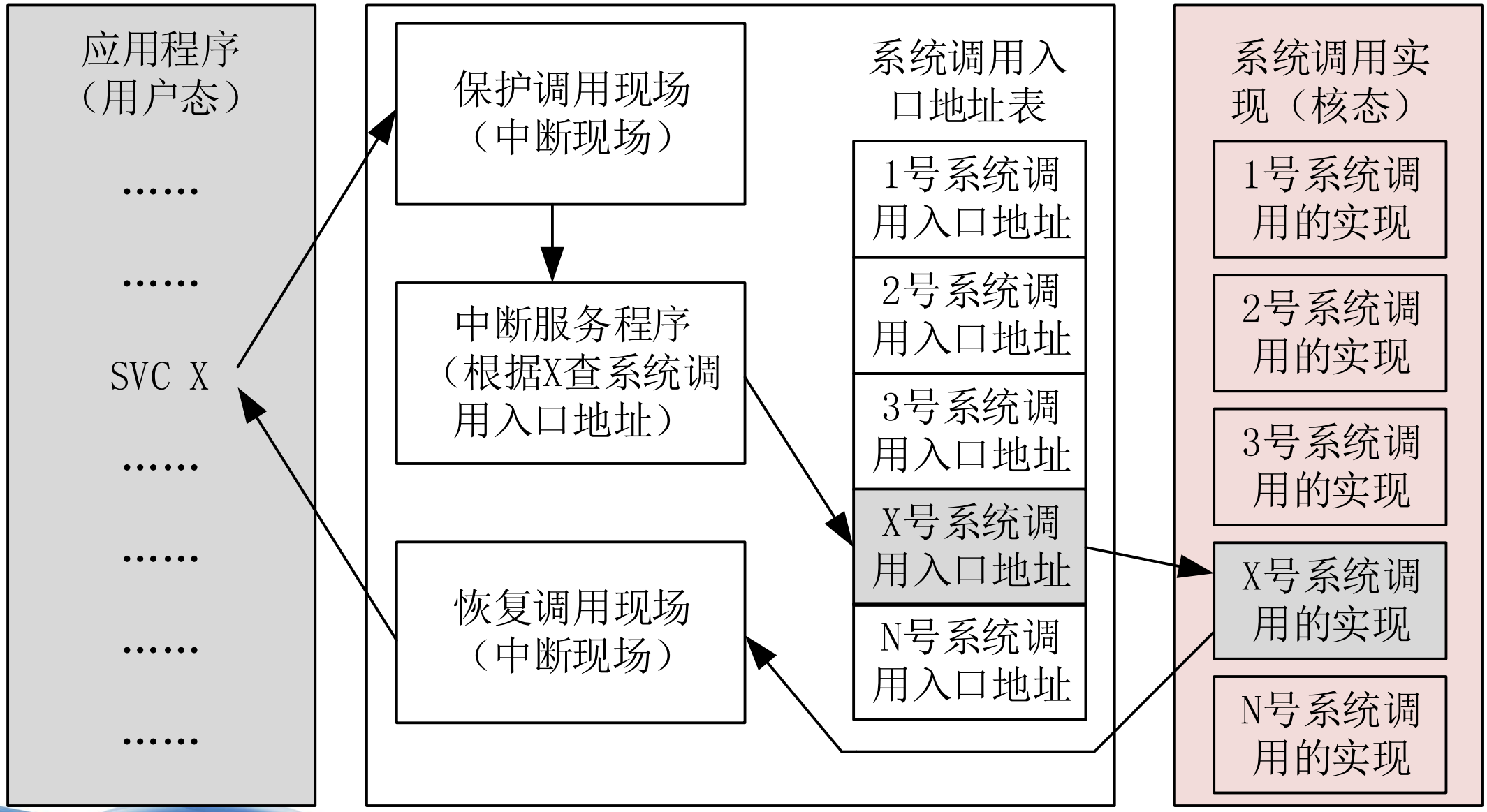
系统调用的调用方式（小结）

- 系统调用的一般调用形式
  - SVC N
- 具体OS系统调用的调用形式
  - DOS： INT 21H + AH
  - Linux：INT 80H + EAX
  - 注意：
    - INT XXH = SVC指令
    - AH/EAX = 系统调用的编号：N

```
MOV AH,09h ;输出字符串
INT 21h
```

```
MOV EAX, 4 //写文件
INT 80h
```

系统调用的执行过程（中断的过程）



隐式系统调用

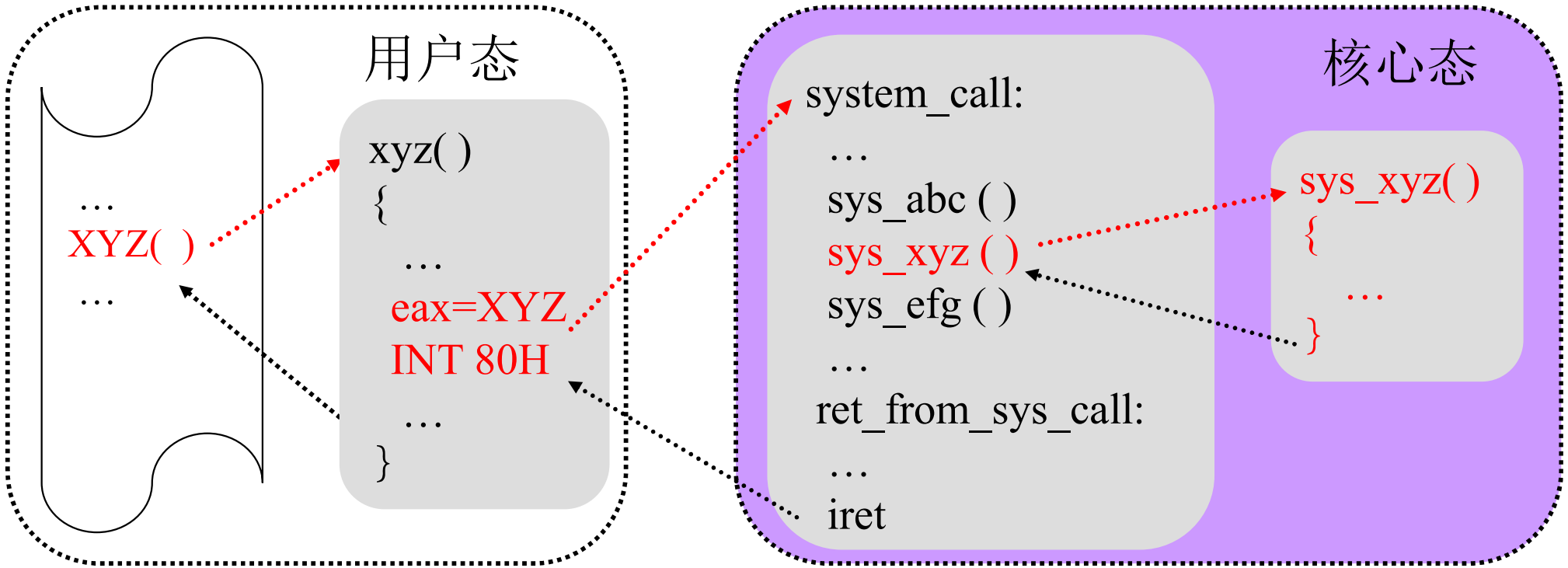
```
#include  <stdio.h>
int main(void)
{
    printf( "Hello World\n " );
    exit( 0 );
}
```

- 特点
  - 系统API函数
  - 在高级语言中使用
  - 包含INT 80h指令（软中断）

```
int main() {;省略了部分代码
    __asm__ (
            "POPL   %ESI;"
            "MOVL   $1, %EBX;"
            "MOVL   %esi, %ECX;"
            "MOVL   $12, %EDX;"
            "MOVL   $4, %EAX;"
            "INT    $0x80;"
            "MOVL   $0, %EBX;"
            "MOVL   $1, %EAX;"
            "INT    $0x80;"
            ".string \"Hello World\\n\";");
}
```

3.4 Linux系统调用原理和实现

Linux系统调用的工作原理



应用程序使用隐式方式调用系统调用xyz( )

→

xyz( )在Libc中编译为含有INT 0x80指令的代码。

→

system_call是0x80号中断服务程序的一部分，指定各系统调用的入口。例sys_xyz( )

→

具体实现各个系统调用。例：sys_xyz ()

KYLIN 银河麒麟

案例：read系统调用的工作原理

- 例：**int read(int fd，char * buf, int n)** // unistd.h

```
1  int read(int fd, char *buf, int n)
2  {
3      long res;
4      __asm__ volatile (
5          "int$0x80"
6          :"=a" (__res)
7          :"0" (__NR_read), "b" ((long)(fd)), "C" ((long)(buf)),
8          "d" ((long)(n)));
9      if (__res>=0)
10         return int __res;
11     errno=- res;
12     return -1;
13 }
```

__NR_Read = 3

```
char buffer[100];
int fd = open("TestFile.log");
int byteRead = read(fd, buffer, sizeof(buffer));
int byteWrite = write(fd, buffer, sizeof(buffer));
```

KYLIN 银河麒麟

案例：read系统调用的工作原理

- 80h中断服务程序

```
1  void sched_init(void)
2  {
3      ......
4      set_intr_gate(0x20,&timer_interrupt);
5      ......
6      set_system_gate(0x80,&system_call);
7  }
```

案例：read系统调用的工作原理

- 系统调用编号的声明

  格式： **#define __NR_CallName ID**

```
1  //unistd.h @Linux 1.0
2  #define __NR_setup          0
3  #define __NR_exit           1
4  #define __NR_fork           2
5  #define __NR_read           3
6  #define __NR_write          4
7  #define __NR_open           5
8  ......
9  #define __NR_fchdir       133
10 #define __NR_bdflush      134
```

案例：read系统调用的工作原理

- 例：**int read(int fd，char * buf, int n)** // unistd.h

```
1  int read(int fd, char *buf, int n)
2  {
3      long __res;
4      __asm__ volatile (
5          "int$0x80"
6          :"=a" (__res)
7          :"0" (__NR_read), "b" ((long)(fd)), "c" ((long)(buf)),
8          "d" ((long)(n)));
9      if (__res>=0)
10         return int __res;
11     errno=- res;
12     return -1;
13 }
```

__NR_Read = 3

```
char buffer[100];
int fd = open("TestFile.log");
int byteRead = read(fd, buffer, sizeof(buffer));
int byteWrite = write(fd, buffer, sizeof(buffer));
```

KYLIN 银河麒麟

案例：read系统调用的工作原理

- 80h中断服务程序

```
1   system_call:
2       push %ds
3       push %es
4       ......
5       mov %dx,%fs
6       call sys_call_table(,%eax,4)
7       pushl %eax # 把系统调用号入栈。
8       movl _current,%eax
9       cmpl $0,state(%eax) # state
10      jne reschedule
11      cmpl $0,counter(%eax) # counter
12      je reschedule
13  ret_from_sys_call:
14      ......
15      movl signal(%eax),%ebx
16      movl blocked(%eax),%ecx
17      notl %ecx # 每位取反。
18      ......
19      call _do_signal # 调用信号处理程序
20      popl %eax
21      ......
22      pop %es
23      pop %ds
24  iret
```

案例：read系统调用的工作原理

- 80h中断服务程序（部分）

```
1   system_call:
2       push %ds
3       push %es
4       ......
5       mov %dx,%fs
6       call sys_call_table(,%eax,4)
7       pushl %eax # 把系统调用号入栈。
8       movl _current,%eax
9       cmpl $0,state(%eax) # state
10      jne reschedule
11      cmpl $0,counter(%eax) # counte
```

__NR_Read = 3

案例：read系统调用的工作原理

- sys_call_table的数据结构

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
    sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
    sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
    sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
    sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
    sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
    sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
    sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
    sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
    sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
    sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
    sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
    sys_setreuid, sys_setregid
};
```

案例：read系统调用的工作原理

- 例：**int sys_read( )** // read_write.c @Linux 1.0

```
1  asmlinkage int sys_read(unsigned int fd,char * buf,
2                          unsigned int count)
3  {
4      int error;
5      struct file * file;
6      struct inode * inode;
7      if (fd>=NR_OPEN || !(file=current->filp[fd]) ||
8                         !(inode=file->f_inode))
9          return -EBADF;
10     if (!(file->f_mode & 1))
11         return -EBADF;
12     if (!file->f_op || !file->f_op->read)
13         return -EINVAL;
14     if (!count)
15         return 0;
16     return file->f_op->read(inode,file,buf,count);
17 }
```

案例：read系统调用的工作原理

```
1  int sys_read(unsigned int fd,char * buf,int count)
2  {
3      struct file * file;
4      struct m_inode * inode;
5      if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
6          return -EINVAL;
7      if (!count)
8          return 0;
9      inode = file->f_inode;
10     if (inode->i_pipe)
11         return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
12     if (S_ISCHR(inode->i_mode))
13         return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
14     if (S_ISBLK(inode->i_mode))
15         return block_read(inode->i_zone[0],&file->f_pos,buf,count);
16     if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
17         if (count+file->f_pos > inode->i_size)
18             count = inode->i_size - file->f_pos;
19         if (count<=0)
20             return 0;
21         return file_read(inode,file,buf,count);
22     }
23     return -EINVAL;
24 }
```

案例：read系统调用的工作原理

- 例：**int sys_write( )** // read_write.c @Linux 1.0

```
1  asmlinkage int sys_write(unsigned int fd,char * buf,
2                           unsigned int count)
3  {
4      int error;
5      struct file * file;
6      struct inode * inode;
7      if (fd>=NR_OPEN || !(file=current->filp[fd]) ||
8                         !(inode=file->f_inode))
9          return -EBADF;
10     if (!(file->f_mode & 2))
11         return -EBADF;
12     if (!file->f_op || !file->f_op->write)
13         return -EINVAL;
14     if (!count)
15         return 0;
16     return file->f_op->write(inode,file,buf,count);
17 }
```

系统调用表sys_call_table的其他版本

- **sys_call_table数据结构**

```
.data
   ENTRY(sys_call_table)
   .long SYMBOL_NAME(sys_ni_syscall)   // 0
   .long SYMBOL_NAME(sys_exit)         // 1
   .long SYMBOL_NAME(sys_fork)         // 2
   .long SYMBOL_NAME(sys_read)         // 3
   .long SYMBOL_NAME(sys_write)        // 4
   .long SYMBOL_NAME(sys_open)         // 5
    ……
```

Ref：/usr/src/linux/arch/i386/kernel/entry.S

3.6 实验：增加系统调用（Linux）

第一次上机：预习，自习，机房或寝室完成

- 在LINUX（优麒麟）中增加新的系统调用
  - 1、编写新的系统调用函数（指函数实现部分）
  - 2、注册新的系统调用（声明系统调用和相应编号）
  - 3、编译新LINUX内核
  - 4、编译和安装模块
  - 5、启动新的LINUX内核
  - 6、[在新内核中]编写应用测试新的系统调用
- 建议环境
  - 优麒麟/UBUNTU/Fedora/自备电脑
  - 开源内核 > 5.0