- 主要内容
  - 用户环境
  - 操作系统的启动
  - 操作系统的生成
  - 用户界面
  - 操作界面
  - 系统调用
- 重点
  - 操作系统启动过程
  - 操作系统生成过程
  - 系统调用机制

1.用户环境和构造

- **用户环境**
  - 用户工作的软件和硬件环境。
- **用户环境构造**
  - 按照用户要求和硬件特性安装和配置操作系统。
    - 提供丰富灵活的操作命令和界面
    - 提供系统用户手册

2. 操作系统的启动

- 相关背景知识
  - 实模式和保护模式
  - BIOS（Basic I/O System ）
  - POST（Power On Self Test ）
  - CMOS
  - MBR （主引导记录）
  - 操作系统的安装
  - 多操作系统的启动选择菜单

实模式和保护模式

- **实模式（实地址模式）**
  - 程序按照8086寻址方法访问00000h—FFFFFh（1MB大小）
  - 寻址方式：物理地址（20位）=段地址左移4位+偏移地址。
  - CPU单任务运行
- **保护模式（内存保护模式）**
  - 寻址方式：段（ 32位）和偏移量（ 32位），寻址4GB
    - 段的属性：起始地址，存取属性，权限级别，…
      - 段描述符表，段描述符，段选择子
      - 段页式寻址机制(段，页）
      - 段寄存器：段选择子
      - 新增32位寄存器：GDR,LDR,CR0,CR1,CR2,…
  - 虚拟地址，进程，封闭空间
  - 应用程序和操作系统的运行环境都被保护
  - CPU支持多任务

- **实模式存取的1M空间**
  - 前面640K 【00000 -- 9FFFF】：基本内存
  - 中间128K 【A0000 -- BFFFF】：显卡显存
  - 末尾256K 【C0000 -- FFFFF】：**BIOS**
    - C0000 -- C7FFF：显示卡BIOS
    - C8000 -- CBFFF：IDE控制器BIOS
    - F0000 – FFFFF：最后64KB，系统BIOS

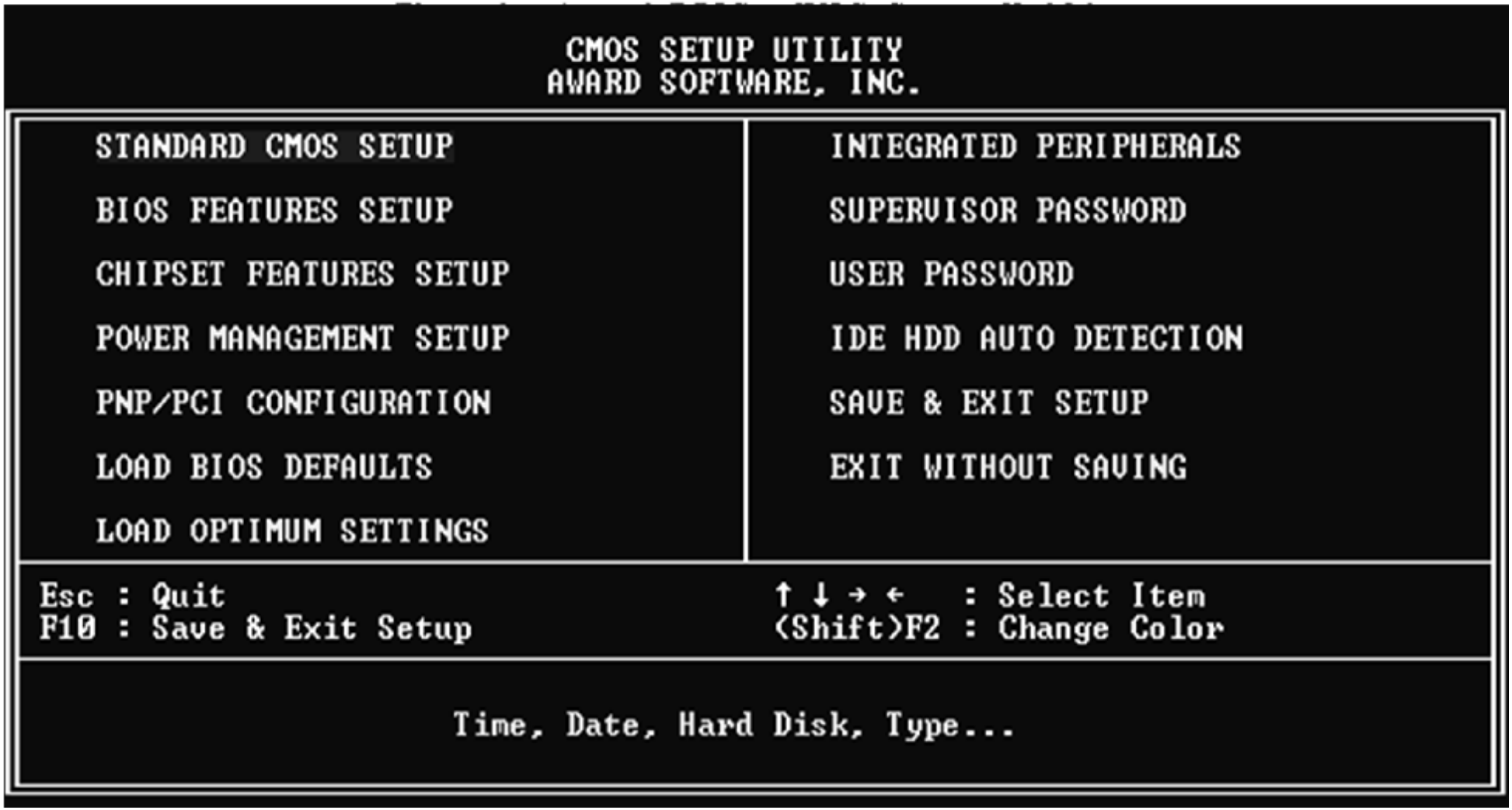
系统BIOS

- **（Basic I/O System ）(Fireware，固件)**
  - 基本输入/输出系统
  - 位置：F0000-FFFFF
  - 类型：
    - Award BIOS
    - AMI BIOS
    - Phoenix BIOS
  - 功能：
    - CMOS设置

PLCC BIOS 芯片

CMOS设置介绍

- **Complementary Metal-Oxide Semiconductor**
  - 互补金属氧化物半导体芯片



系统BIOS

- **（Basic I/O System ）(Fireware，固件)**
  - 基本输入/输出系统
  - 位置：F0000-FFFFF
  - 类型：
    - Award BIOS
    - AMI BIOS
    - Phoenix BIOS
  - 功能：
    - CMOS设置
    - 基本I/O设备中断服务

PLCC BIOS 芯片

BIOS中断

ⅠBIOS使用的中断类型号为10H ~ 1FH。

中断类型号	功能	中断类型号	功能
10H	显示器 I/O 调用	18H	磁带 BASIC 入口
11H	设备检验调用	19H	自举程序入口
12H	存储器检验调用	1AH	时间调用
13H	软盘 I/O 调用	1BH	ctrl－Break 控制
14H	异步通信口调用	1CH	定时处理
15H	磁带 I/O 调用	1DH	显示器参数表
16H	键盘 I/O 调用	1EH	软盘参数表
17H	打印机 I/O 调用	1FH	字符点阵结构参数表

系统BIOS

- **（Basic I/O System ）(Fireware，固件)**
  - 基本输入/输出系统
  - 位置：F0000-FFFFF
  - 类型：
    - Award BIOS
    - AMI BIOS
    - Phoenix BIOS
  - 功能：
    - CMOS设置
    - 基本I/O设备中断服务
    - POST(上电自检)
    - 系统自举

PLCC BIOS 芯片

POST概念

- **POST**
  - Power On Slef-Test（加电自检）
  - 初始化基本硬件
    - CPU，内存，显卡…
  - 自检正常不提示，错误则通过喇叭提示。
- **按下PowerOn或者Reset键**
  - 开始执行**FFFF0**处的指令

```
JUMP POST      ; POST位于系统BIOS内部
```

POST之后……

- 查找显卡BIOS，调用显卡 BIOS；
- 依次查找其它设备执行相应设备的BIOS；
- 显示启动画面
  - BIOS信息
  - 主板信息
  - 芯片组型号
  - ……
- 根据用户指定顺
- OS启动后，由

操作系统的启动

- 启动过程
  - 从加电到用户工作环境准备好的过程
    - **(1)初始引导**
    - (2)核心初始化
    - (3)系统初始化

1)初始引导

- 目的
  - 把OS核心装入内存并使之开始工作接管计算机系统
- 过程
  - 加电，JUMP POST
  - … BIOS中的**启动程序**运行
  - 启动程序：
    - 读取0面0道第1扇区内容（MBR）
    - 加载MBR中的**引导程序**。
  - 引导程序：
    - 根据相关参数，读取硬盘指定位置的文件到内存
    - 加载硬盘上OS**内核**，并初始化基本参数
  - OS内核：逐步加载OS剩余部分，最后完全控制计算机

常见引导程序：
(1）ntldr (WinXP以下)
(2）bootmgr(Vista以上含Win7)
(3）GRUB
(4）LILO

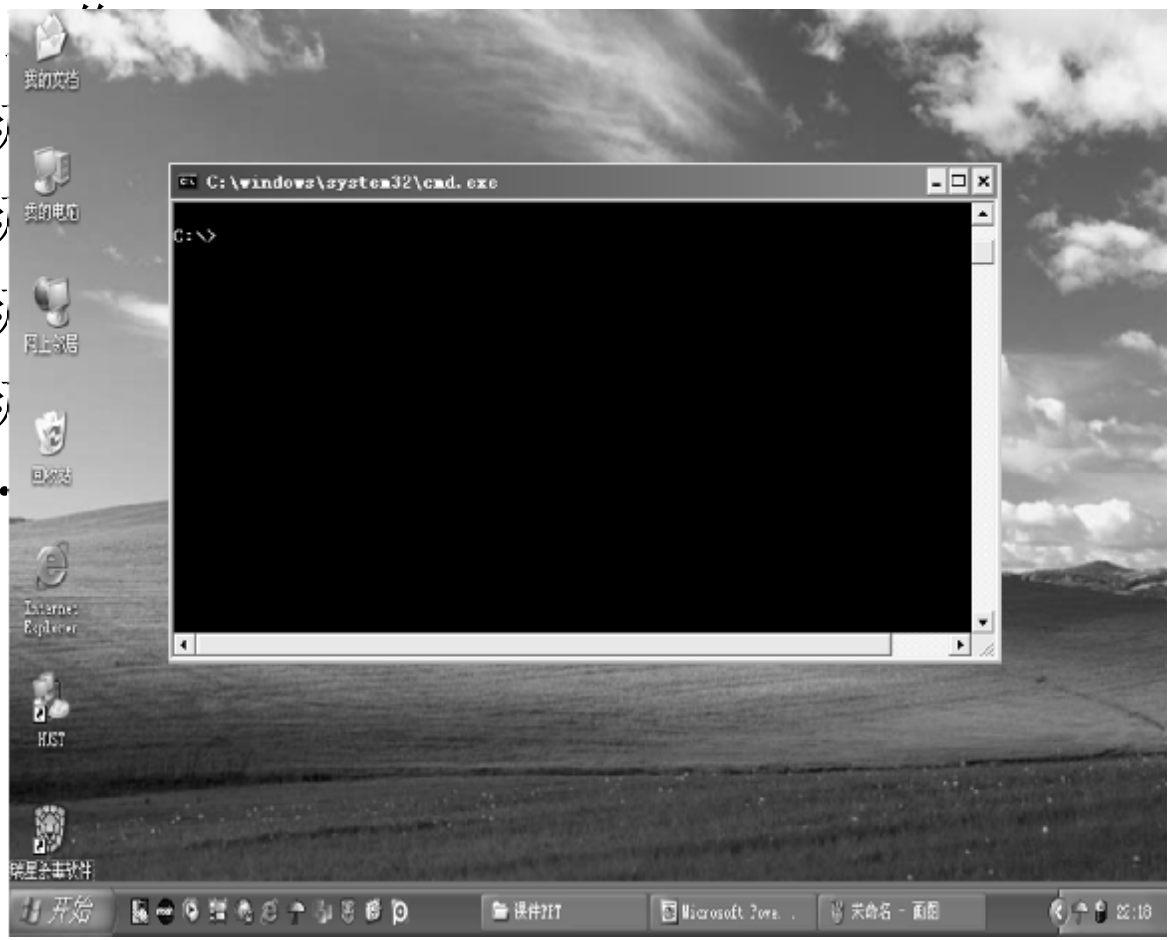
2)核心初始化

- 核心初始化
  - 目的：OS内核初始化系统的核心数据
  - 典型工作
    - 各种寄存器的初始化
    - 存储系统和页表初始化
    - 核心进程构建
    - ……

3)系统初始化

l 系统初始化

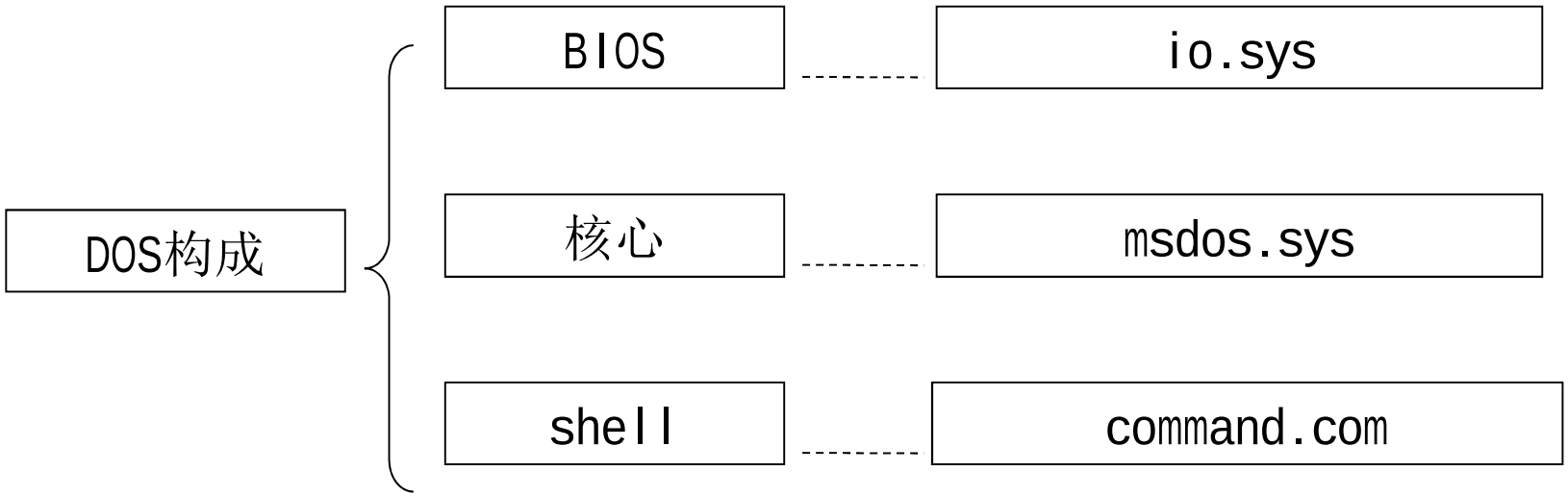
- n 为用户使用系统作准备，使系统处于**待命状态**。
- n 主要
  - u 初
  - u 初
  - u 初
  - u 初
  - u …



DOS启动实例

l DOS操作系统的构成

- **n** io.sys: 提供DOS与BIOS的调用接口
- **n** msdos.sys: 进程管理、存储管理、文件管理、解释系统调用
- **n** command.com:Shell命令和键盘命令解释及执行



- **DOS的启动过程**
  - POST
    - 加电后BIOS启动主机自检程序
  - 初始引导
    - BIOS从MBR读入**引导程序**，装入内存的特定位置
    - **引导程序**运行将io.sys及msdos.sys读入内存
    - DOS运行起来取代BIOS接管整个系统。
  - 核心初始化
    - 操作系统读入config.sys配置系统核心
  - 系统初始化
    - 读入Command.com，执行autoexec.bat，系统待命

- Windows的启动过程
  - POST
    - 加电后BIOS启动主机自检程序
  - 初始引导
    - BIOS从MBR读入引导程序，装入内存的特定位置
    - 引导程序启动DOS7.0，调入操作系统核心
    - WINDOWS开始接管系统
  - 核心初始化
    - 资源状态、核心数据等初始化；
  - 系统初始化
    - GUI界面生成，系统处于待命/消息接受状态

- **LINUX的启动过程**
  - POST → MBR → KERNEL映像 → KERNEL映像自解压并执行→内核初始化→内核启动 →
- **注释**
  - 当KERNEL映像被加载到内存之后，内核阶段就开始了。
  - KERNEL映像是一个zlib压缩过的内核映像，通常是一个zImage（小于 512KB）或 bzImage（大于 512KB）。
  - 在KERNEL映像最前面是一个可执行例程（实现少量硬件设置，并对映像其余部分解压放入高端内存中）。然后该例程会调用内核，并开始启动内核引导的过程。

GRUB 中的手工引导

假设硬盘分割如下：

hda1：Windows98（hd0,0）

hda2：Slackware Linux（hd0,1）

hdb1：Mandrake Linux（hd1,0）

Linux 中第一颗 IDE 硬盘是 hda

GRUB 将第一颗 IDE 硬盘视为 hd0

Linux 中第一颗 IDE 硬盘的第一个分割区是 hda1

GRUB 将第一颗 IDE 硬盘的第一个分割区表示为 hd0,0

GRUB 中的手工引导

激活 Slackware Linux

```
grub> root (hd0,1)
grub> kernel /boot/vmlinuz root=/dev/hda2 ro
grub> boot
```

root 指令会 mount 后面参数的硬盘分割区位置。

kernel 指令会加载其参数的系统核心

Linux 的核心通常是在/boot 目录下名为 vmlinuz 的档案

root=/dev/hda2 是告诉核心根目录的位置是在/dev/hda2

并要求挂载成只读(ro)

boot 指令以进行 Slackware 系统的开机

在 hdb1 上的 Mandrake Linux

```
grub> root (hd1,0)
grub> kernel /boot/vmlinuz root=/dev/hdb1 ro
grub> boot
```

假设硬盘分割如下：

hda1：Windows98 (hd0,0)

hda2：Slackware Linux (hd0,1)

hdb1：Mandrake Linux (hd1,0)

GRUB 中的手工引导

进入 Windows98:

```
grub> rootnoverify (hd0,0)
grub> chainloader +1
grub> makeactive
grub> boot
```

rootnoverity 指令让 GRUB 不要 mount (hd0,0)的分割区，只要知道待会是要激活此分割区上的操作系统

chainloader +1 的意思是指定此分割区上的第一个扇区来做激活

makeactive 指令是要在此分割区上设定 active 的旗标，只要是 Windows 的操作系统都是要这么做的。

假设硬盘分割如下：

hda1：Windows98 (hd0,0)

hda2：Slackware Linux (hd0,1)

hdb1：Mandrake Linux (hd1,0)

内核完成引导后，加载init程序

- **init进程是系统所有进程的起点**
  - 进程号永远是1。
  - init进程通过/etc/inittab脚本进行初始化
    - 不同运行级别（Runlevel）/etc/inittab脚本不同
  - 脚本文件/etc/inittab
    - Init进程根据/etc/inittab执行相应的脚本进行系统初始化
      - 设置键盘、字体、装载模块、设置网络等等。

- **/etc/inittab （Redhat Linux环境）**
  - 执行/etc/rc.d/rc.sysinit
  - 执行/etc/rc.d/rcX.d/[KS]
  - 执行/etc/ec.d/rc.local
  - 执行 /bin/login 程序
    - 启动 /etc/profile 文件
    - 启动用户目录下bash_profile或bash_login或~/.profile

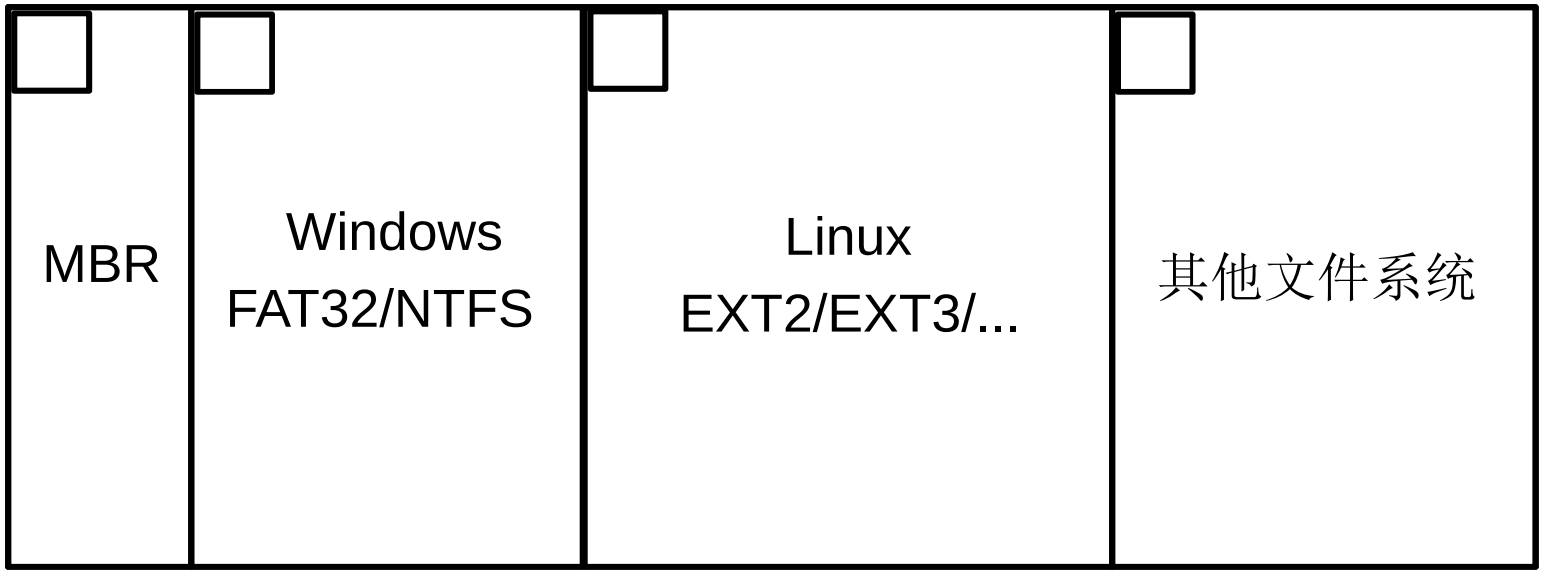
用户可以在该文件中添加自定义的初始化工作。

MBR:Main Boot Record

- **MBR**
  - 存放在主启动扇区（Main boot sector）
  - 存放和OS启动的相关信息
  - 512 BYTES
  - 结束：0xAA55h

MBR与bootloader以及硬盘分区/格式化

MBR:Main Boot Record



- •启动扇区（Boot Sector）: BootLoader
- •主启动扇区（Main boot sector）:BootLoader或更强功能的启动管理

•分区/次引导记录：Partition Boot Record，PBR

- **•提供菜单**：使用者可以选择不同的启动项目，这是多重启动的重要功能
- **•加载核心文件**：直接指向可启动的程序区段来开始操作系统；
- **•跳转其他Loader**：将启动管理功能转交给其他loader负责。

MBR的工作原理

- （1）POST → CMOS设置（硬盘启动） → 读取MBR →控制权交给MBR。
- （2）MBR读取分区表(Partition Table)，找到其中的活动分区（Active Partition），并确认其他的分区都不是活动分区。MBR读取活动分区的第一个分区（次引导记录PBR），并把它加载到内存中去。
- （3）PBR控制后面的引导过程。

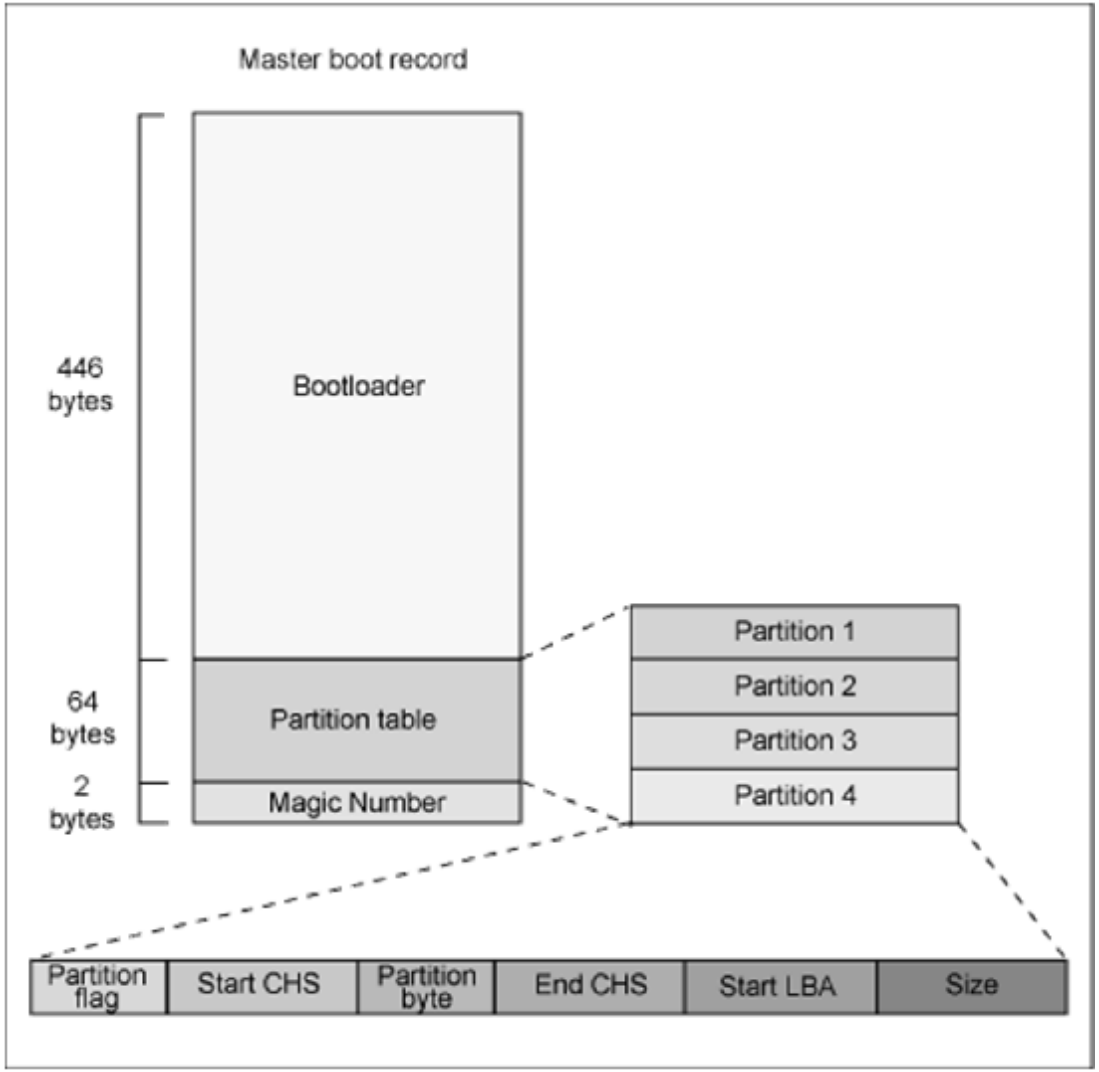
操作系统的安装过程

- 安装过程
  - 把OS映像拷贝到存储空间；
    - 拷贝/安装位置：硬盘
  - 写启动信息（boot code）到MBR扇区（Master Boot Record Sector）
    - 0面0道1扇区
    - 512字节

MBR（Master Boot Record）

- **MBR**
  - 存放和OS启动的相关信息
  - 512 BYTES
  - 结束：0xAA55h
- **安装多操作系统对MBR的影响**
  - MBR重写
  - MBR追加
  - 注意：安装顺序
- **查看和修改MBR内容**
  - DiskEdit （DOS）,WinHex,HxD,GDisk (Windows)
  - DD ：dd if=/dev/hda of=./mbr bs=512 count=1

- **MBR的结构：**
  - Boot Loader,446B
    - 000~1BD
  - Partition Table,64B
    - 该硬盘的分区表
    - 1BE~1FD
  - Magic number,2B
    - 1FE~1FF
    - 55AA
    - 表示MBR结束。

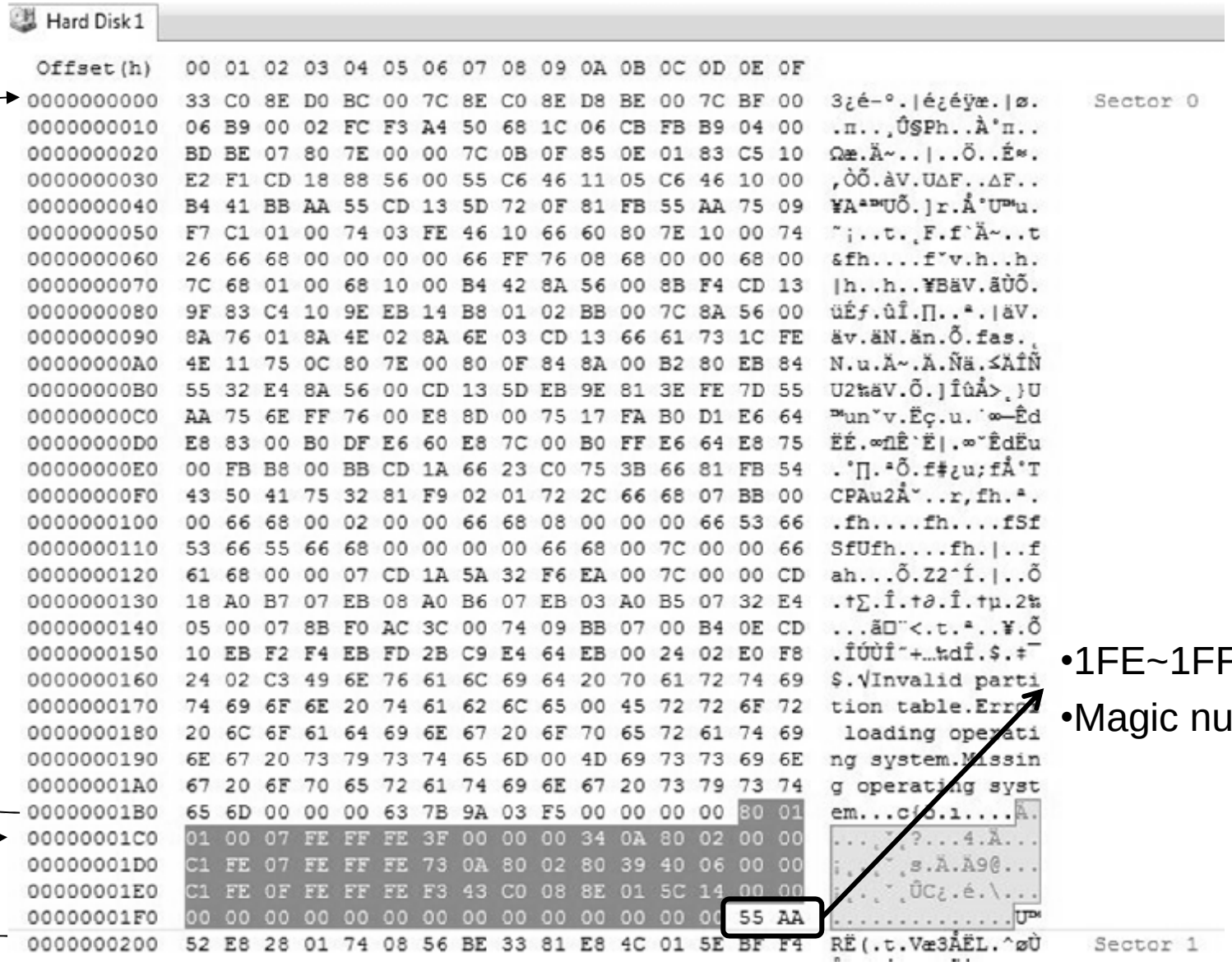


HxD打开硬盘MBR扇区

- 000~1BD
- Boot loader
- 446B

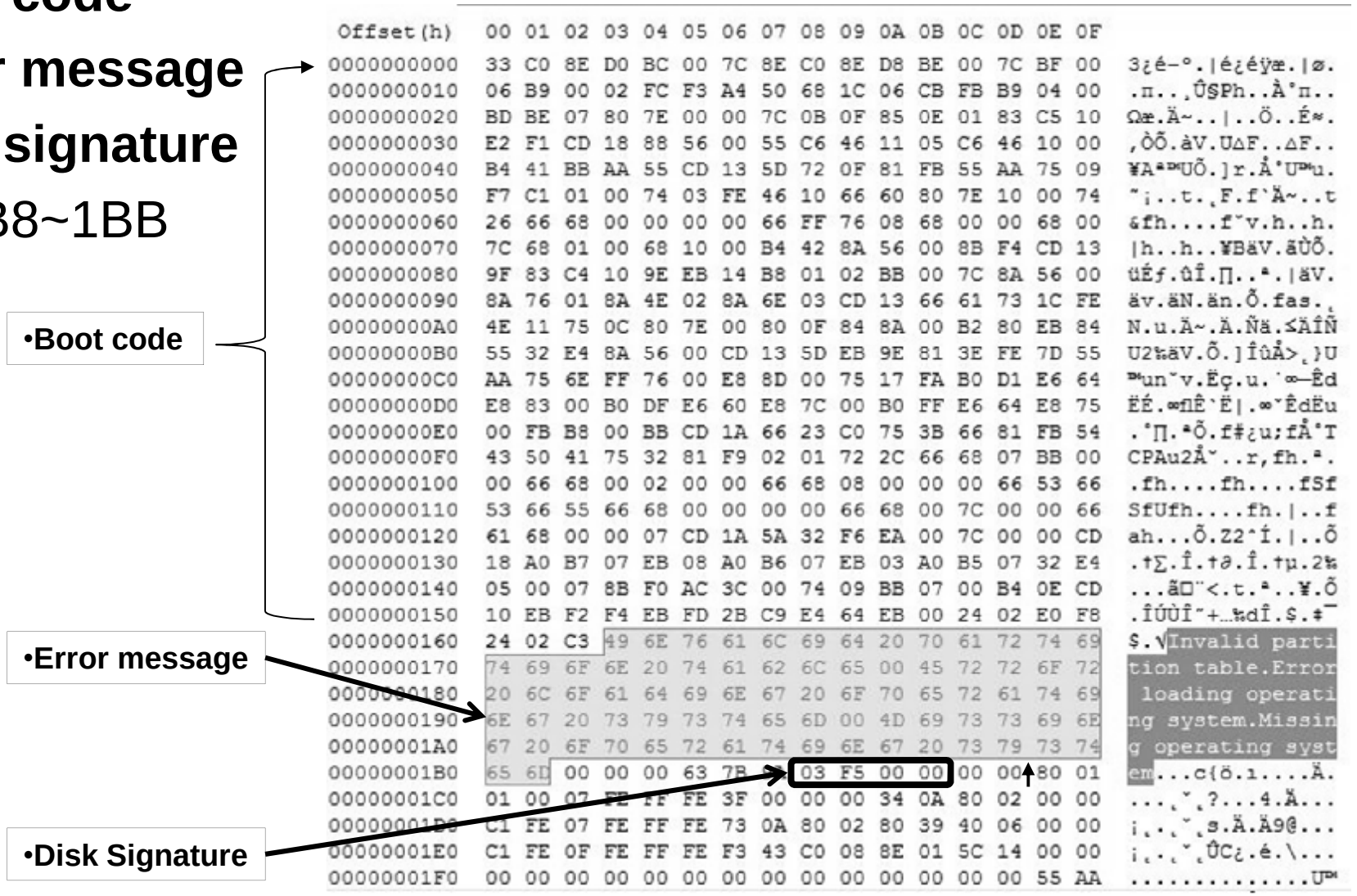
- 1BE~1FD
- Partition table
- 64B

- 1FE~1FF
- Magic number

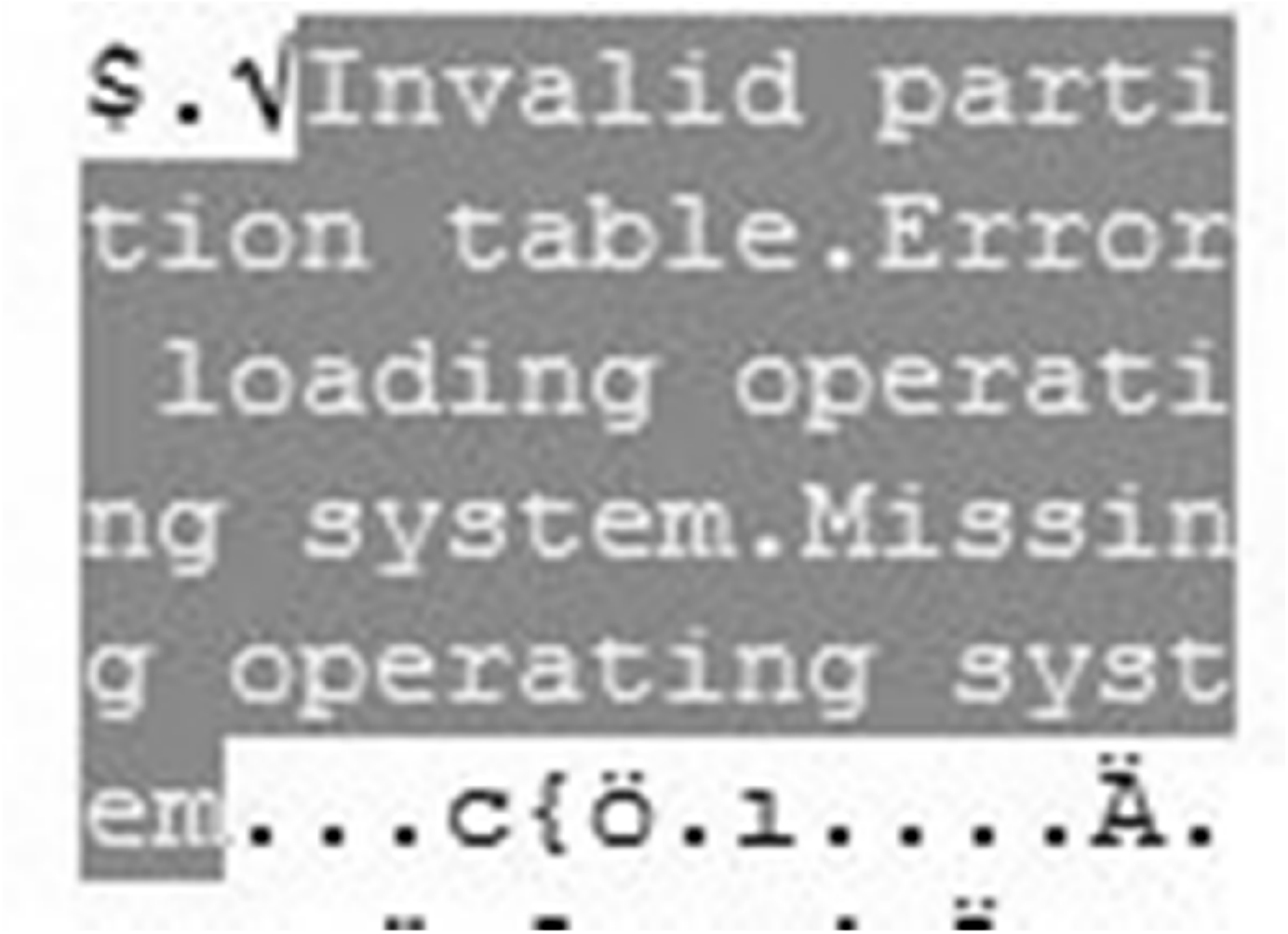


Boot Loader分3部分

- Boot code
- Error message
- Disk signature
  - n 1B8~1BB

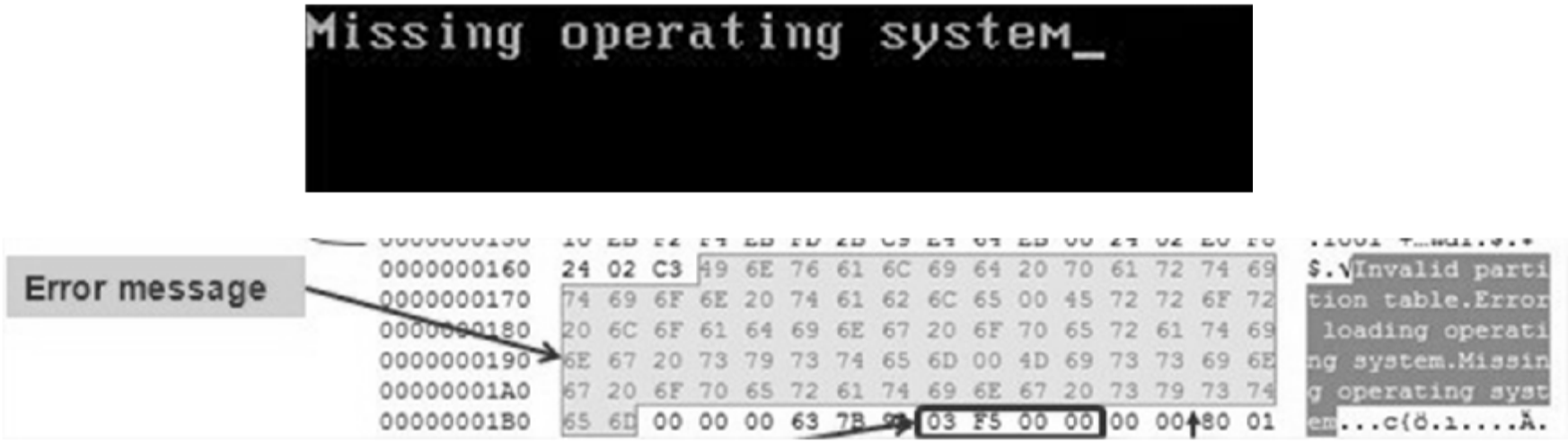


Error message



Error message

- **Windows 7的启动Error message**
  - **n** Invalid partition table;
  - **n** Error loading operating system;
  - **n** Missing operating system;



MBR程序的例子

```
; 功能：加电开机后显示“Hello, OS world!”
1  org 07C00h    ; 程序加载到7C00处
2  mov ax, cs ;
3  call DispStr      ; 调用显示字符串例程
4  jmp $             ; 无限循环
5  DispStr:
6      mov ax, BootMessage
7      mov bp, ax          ; es:bp = 串地址
8      mov cx, 16          ; cx = 串长度
9      mov ax, 1301h       ; ah = 13h, al = 01h
10     int 10h             ; BIOS 10h 号中断:显示字符串
11     ret
12 BootMessage: db "Hello, OS world!"
13 times 510-($-$$) db 0   ; 填充若干个0，使代码字节为510字节
14 dw 0xAA55   ; 结束标志
```

3. 操作系统的生成

- **操作系统的生成**
  - 满足特定硬件环境和用户需要，组装和构建操作系统过程。
- **操作系统生成的主要步骤**
  - 根据硬件环境/用户要求配置功能模块和构造参数
  - **构建（build）OS的映像**
- **操作系统的生成的前提**
  - 操作系统由可拆装模块构成
  - 有交互式配置工具
  - 有映像构建（build）工具

最新版本Linux内核4.4.5

Linux操作系统的生成

- 1、获取Linux内核的源代码（最好是当前版本）
- 2、选择和启动内核配置程序
- 3、根据需要配置内核模块和参数
- 4、重新编译新的内核
- 5、编译和安装模块
- 6、启动新内核

- 1、获取**Linux**内核的源代码
  - **n** http://www.kernel.org/

  # cd /usr/src

  # tar zxvf linux-2.6.38-12.tar.gz
- 2、选择和启动内核配置程序

  # cd /usr/src/linux-2.6

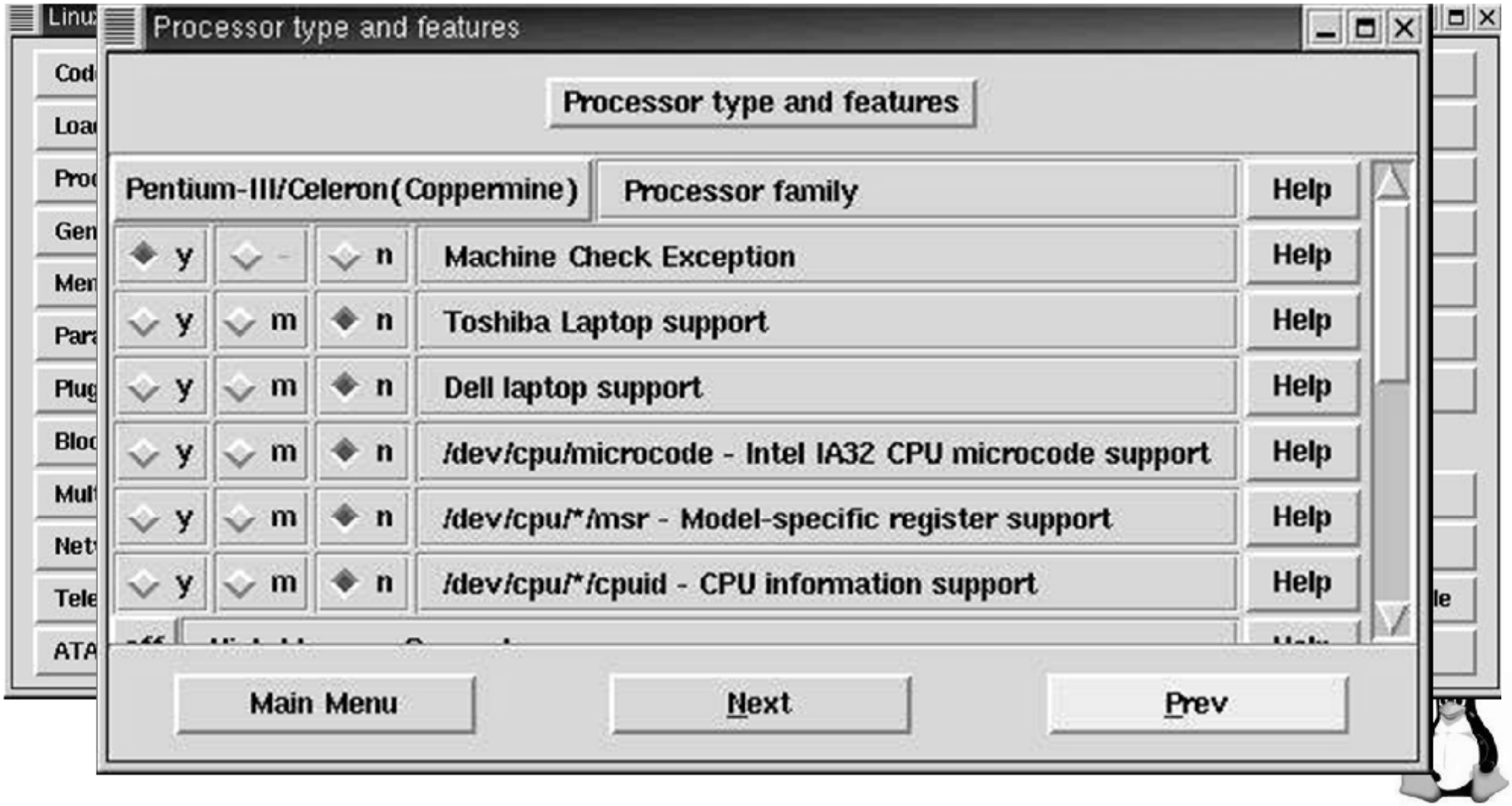
  #make config（文本界面，不推荐使用）

  #make xconfig（图形窗口模式，xWindow使用）

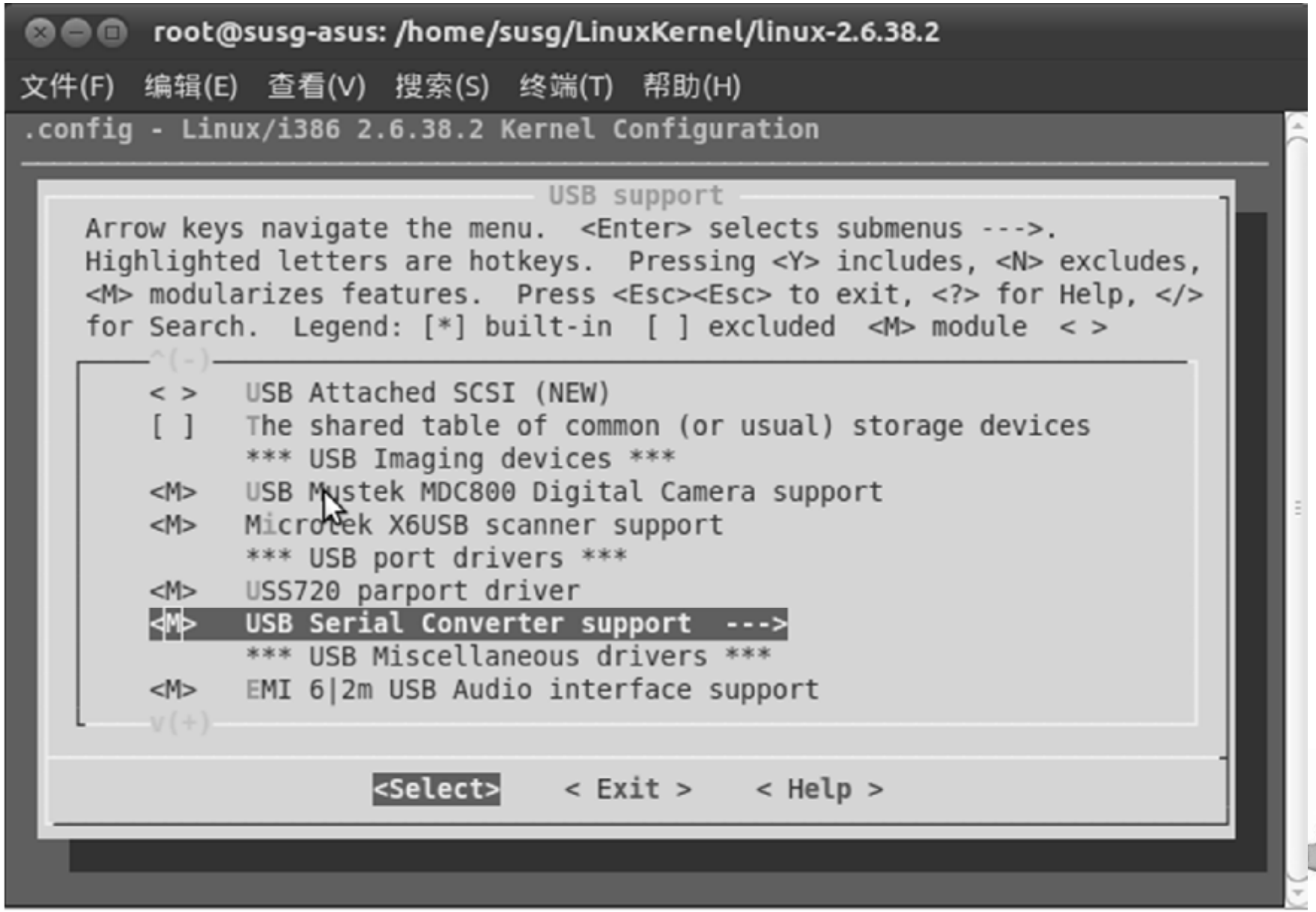
  #make menuconfig（文本选择界面，字符终端）

make xconfig

•Linux内核配置对话框



make menuconfig



3、根据需要配置内核模块和参数

- ① Loadable module support 设置对可加载模块支持。
  - Enable loadable module support (y)
  - Set version info on all module symbols (n)
  - Kernel module loader (y)
- ② **Processor type and features** 设置**CPU**的类型
  - Processor family 选择CPU类型
  - High Memory Support (n)
  - Math emulation (n)
  - MTTR support： (n)
  - Symmetric multi-processing support (n)

- **③ General setup　对普通的一些属性进行设置。**
  - Networking support：（y）
  - PCI support（y）
  - PCI access mode　PCI卡存取模式: BIOS、Direct和Any
  - Support for hot-pluggable devices　(n)
  - PCMCIA/CardBus support　(n)
- **④ Parallel port support　并口支持　（y）**
- **⑤ Plug and Play configuration：即插即用配置（y）**
- **⑥ Block devices 块设备支持的选项**
  - Normal PC floppy disk support 软盘支持（y）
  - Network block device support 网络块设备支持（y）

- ⑦ **Networking options** 选取**TCP/IP networking**选项
- ⑧ **Network device support** 网络设备支持的选项

  例如：使用Realtek 8139网卡

  - Ethernet（10 or 100Mbit） (y)
  - RealTeck RTL-8139 PCI Fast Ethernet Adapter support (y)
- ⑨ **Mice** 鼠标设置选项：串口、**PS/2**等类型鼠标
- ⑩ **File systems** 文件系统类型。
  - DOS FAT fs 选项：FAT16， FAT32
  - NTFS file system support
  - /proc file system support： （y）
- ⑪ **Sound** 声卡驱动，选项：声卡型号
- ⑫ **USB support** **USB**接口的支持，根据需要选择。

- **4、重新编译新的内核**

  # make dep　　生成依赖dependency信息

  # make clean　　清除旧的编译结果

  # make bzImage　. /arch/i386/boot/bzImage

- **5、编译和安装模块**

  # make modules

  # make modules_install

模块被编译且安装到 /usr/lib/<内核版本号> 目录下。

6、启动新内核

- cp bzImage /boot/bzImage
- LILO
  - 配置/etc/lilo.conf

```
image=/boot/bzImage
label=newLinux build by Zhang San Feb.28, 2012
```

  - 命令 #lilo　使配置生效
- GRUB (与发行版本有关)
  - 配置/boot/grub/grub.conf

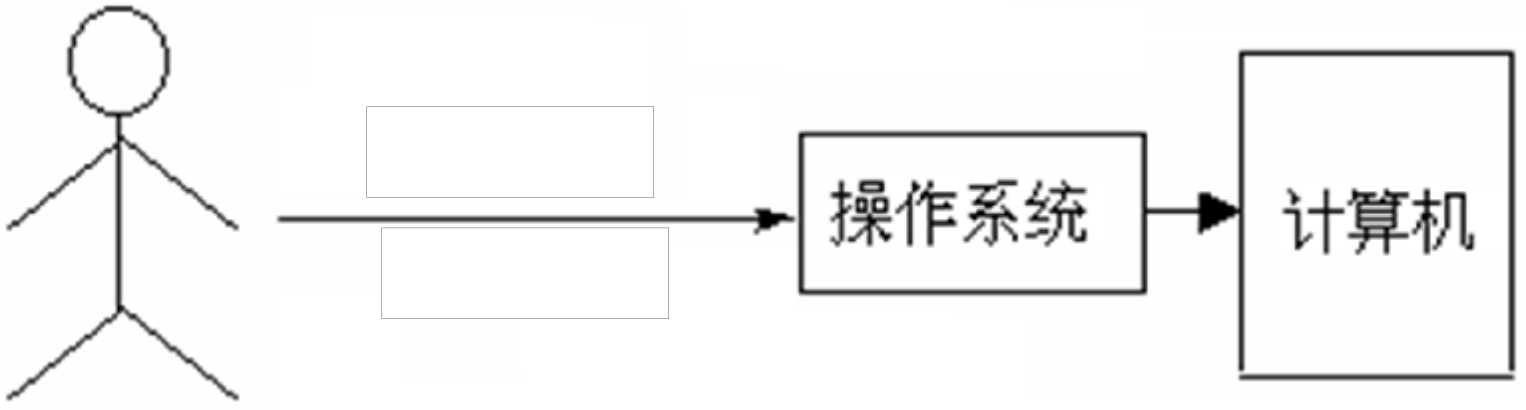
```
title newLinux build by Zhang San Feb.28, 2012
root (hd0,1)
kernel /boot/bzImage ro root=/dev/hda2
```

参考网址

- http://blog.csdn.net/xiaocainiaoshangxiao/article/details/21931801　ubuntu12.04的系统,Linux 2.6.30
- http://www.linuxidc.com/Linux/2011-01/31456.htm Fedora下内核编译
- http://www.2cto.com/os/201204/125945.html **fedora**内核更新（安装及卸载）

4. 操作系统的用户界面

- **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)



典型的操作界面


DEV_TOOL
文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)
后退
命令提示符
转到
compile - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
@echo off
A86.COM +P0 %1 %2
bin2hex.exe %2 %3 -s 0380 -o 0100
批处理命令/程序
Ln 1, Col 1
16 个对象
937 MB
我的电脑

- 操作界面
  - 图形用户接口（GUI，Graphic User Interface ）
    - 窗口，图标，菜单，按钮，鼠标(消息，事件)
  - 键盘命令（COMMAND）
    - 在控制台环境下接收键盘输入的命令
    - 类型
      - 普通命令
      - 批处理程序
      - shell
  - 作业控制语言（JCL，Job Control Language）

普通命令

l DOS典型命令

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

普通命令

l LINUX的典型命令

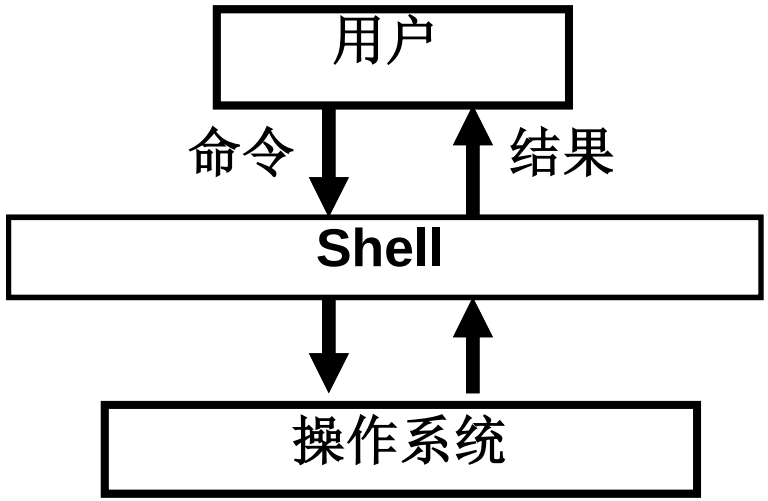
命令	基本功能		命令	基本功能
man	查看命令的帮助		tar	管理 TAR 类型的文件
cd	改变工作路径		chown, chgrp	设置文件/目录的拥有者
ls	列目录信息		chmod	改变文件属性
ps	显示系统中的进程及 ID		find	用于查找某个文件
mount	挂接文件系统		locate	查找文件
umount	卸掉文件系统		whereis	查看文件放在哪个目录

批处理

- 普通命令的集合，批执行,由command解释执行。

Shell

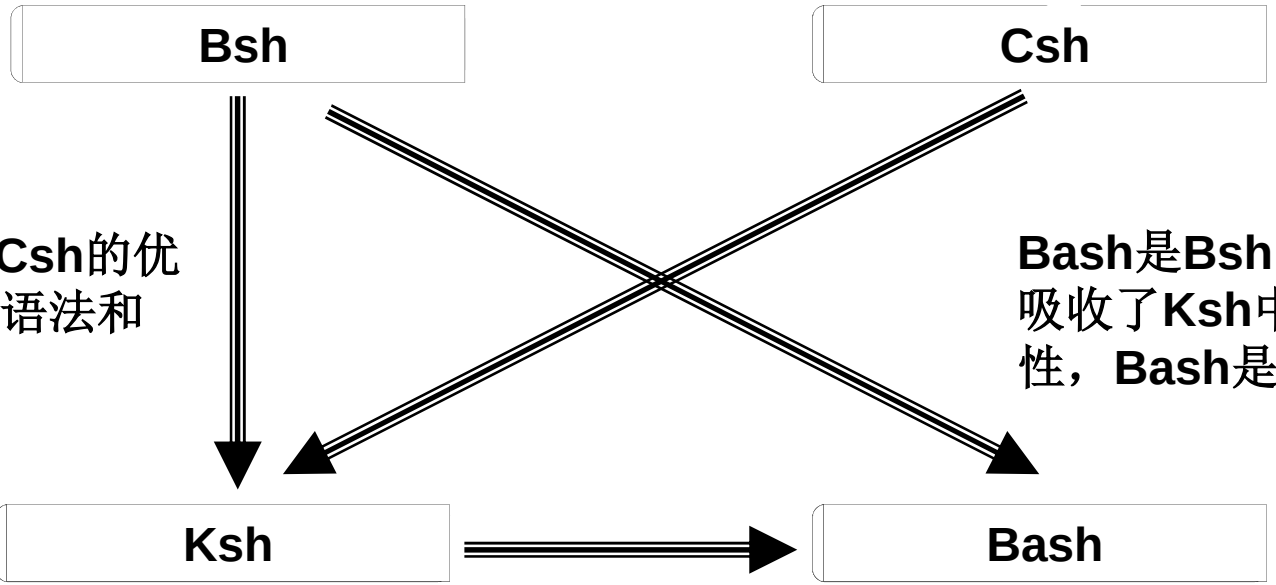
- **Shell是一个控制台方式的程序，通过类似程序的方式执行具有一定逻辑顺序的命令序列完成某个较复杂的功能和人机交互，最后返回结果。**
  - 注意：Shell本身不执行命令，仅仅是组织和管理命令
- **Shell是操作系统与用户进行交互操作的界面**



Shell的发展与分类

Bsh在20世纪70年代中期诞生于新泽西的AT&T贝尔实验室，具有较强的脚本编程功能

Csh在20世纪80年代早期诞生于加利福尼亚大学，使用C语言的语法，用户命令交互更加方便



Ksh结合Bsh和Csh的优势，兼有Bsh的语法和Csh的交互特性

Bash是Bsh的升级替代品，吸收了Ksh中的诸多优秀特性，Bash是开源软件

Bash的主要功能

- 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
- 提供作业控制功能
- 管道与重定向
- 具有将命令序列定义为功能键的功能
- **Shell**脚本编程

Bash的命令行编辑功能

操作键	功能
左右方向键	使用左右方向键可以使光标在当前命令行中的已有字符间进行任意的移动
退格键	删除命令行中光标左边的字符
Del	删除当前光标处的字符
Home	将光标快速移动到命令行的行首
End	将光标快速移动到命令行的行尾
Ctrl + u	删除当前光标到行首的内容
Ctrl + k	删除当前光标到行尾的内容

Bash的命令行补全功能

- 命令补全功能
  - 使用Tab键可在命令查找路径中查找匹配的命令，并进行命令拼写的补全
- 文件补全功能
  - 使用Tab键可对文件和目录名进行补全

Bash的命令历史与命令重复

- 命令历史功能的使用
  - **n** 使用上下方向键浏览已输入命令（历史命令）
- 历史命令的查看

  $ history
- 用户命令历史保存文件

  ~/.bash_history
- 命令历史的清除

  $ history -c
- 相关的环境变量

  HISTFILE，HISTSIZE

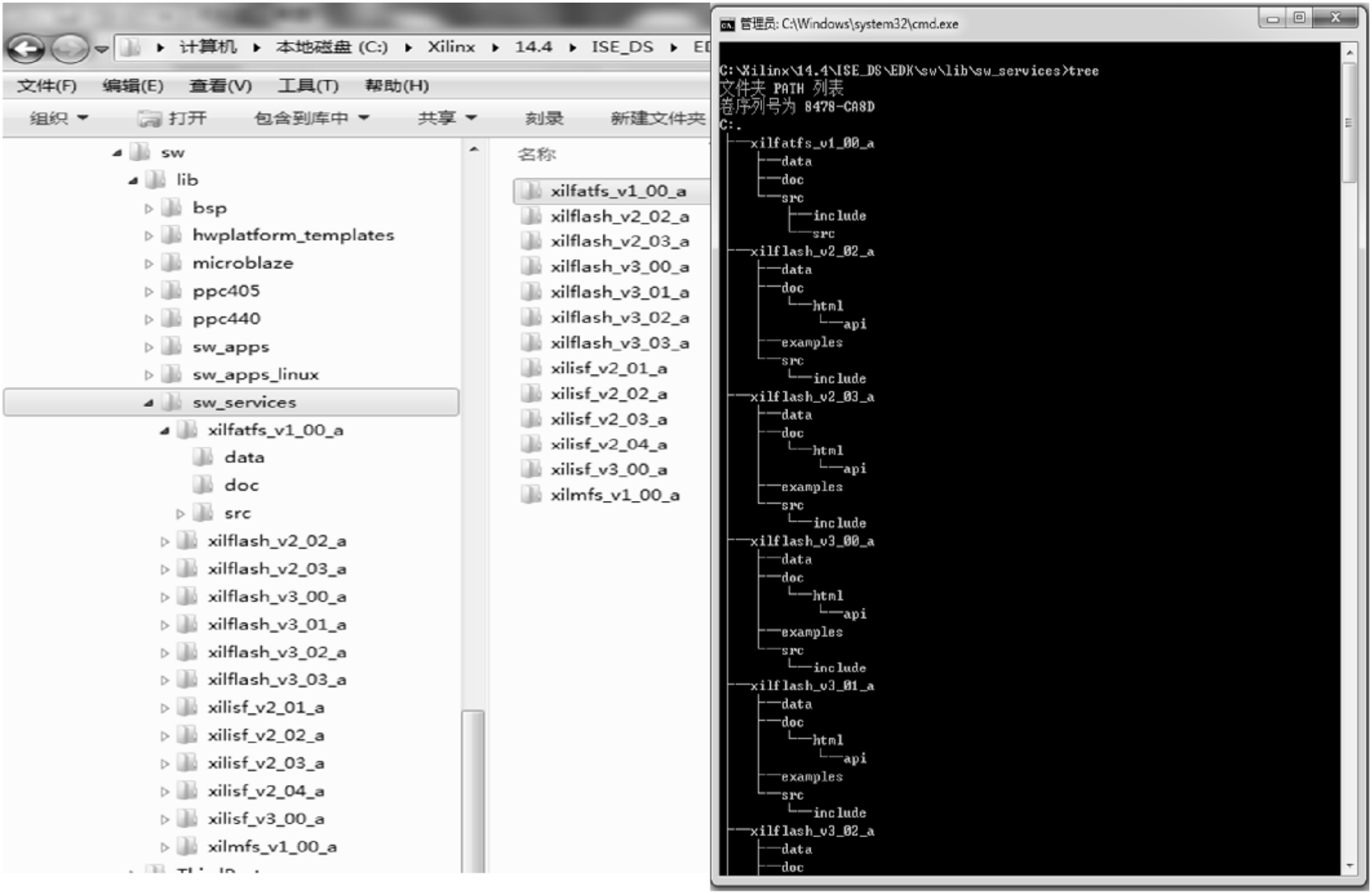
管道与重定向

- 标准输入输出
- 重定向操作
- 管道操作

重定向操作的例子（windows）

- **C:\>tree c: > c:/test2013.txt（或)**

  C:\>tree c:\ > c:/test2013.txt


计算机 ▸ 本地磁盘 (C:) ▸ Xilinx ▸ 14.4 ▸ ISE_DS ▸ ED
文件(F) 编辑(E) 查看(V) 工具(T) 帮助(H)
组织 打开 包含到库中 共享 刻录 新建文件夹
sw
lib
bsp
hwplatform_templates
microblaze
ppc405
ppc440
sw_apps
sw_apps_linux
sw_services
xilfatfs_v1_00_a
data
doc
src
xilflash_v2_02_a
xilflash_v2_03_a
xilflash_v3_00_a
xilflash_v3_01_a
xilflash_v3_02_a
xilflash_v3_03_a
xilisf_v2_01_a
xilisf_v2_02_a
xilisf_v2_03_a
xilisf_v2_04_a
xilisf_v3_00_a
xilmfs_v1_00_a
名称
xilfatfs_v1_00_a
xilflash_v2_02_a
xilflash_v2_03_a
xilflash_v3_00_a
xilflash_v3_01_a
xilflash_v3_02_a
xilflash_v3_03_a
xilisf_v2_01_a
xilisf_v2_02_a
xilisf_v2_03_a
xilisf_v2_04_a
xilisf_v3_00_a
xilmfs_v1_00_a
管理员: C:\Windows\system32\cmd.exe
C:\Xilinx\14.4\ISE_DS\EDK\sw\lib\sw_services>tree
文件夹 PATH 列表
卷序列号为 8478-CA8D
C:.
├─xilfatfs_v1_00_a
│ ├─data
│ ├─doc
│ └─src
│ ├─include
│ └─src
├─xilflash_v2_02_a
│ ├─data
│ ├─doc
│ │ └─html
│ │ └─api
│ ├─examples
│ └─src
│ └─include
├─xilflash_v2_03_a
│ ├─data
│ ├─doc
│ │ └─html
│ │ └─api
│ ├─examples
│ └─src
│ └─include
├─xilflash_v3_00_a
│ ├─data
│ ├─doc
│ │ └─html
│ │ └─api
│ ├─examples
│ └─src
│ └─include
├─xilflash_v3_01_a
│ ├─data
│ ├─doc
│ │ └─html
│ │ └─api
│ ├─examples
│ └─src
│ └─include
├─xilflash_v3_02_a
│ ├─data
│ ├─doc

标准输入输出（Linux）

输入输出文件	文件编号	默认设备
标准输入	0	键盘
标准输出	1	显示器
标准错误输出	2	显示器

重定向操作（Linux）

类别	操作符	说明
输入重定向	<	输入重定向是将命令中接收输入的途径由默认的键盘更改（重定向）为指定的文件
输出重定向	>	将命令的执行结果重定向输出到指定的文件中，命令进行输出重定向后执行结果将不显示在屏幕上
	>>	将命令执行的结果重定向并追加到指定文件的末尾保存
错误重定向	2>	清空指定文件的内容，并保存标准错误输出的内容到指定文件中
	2>>	向指定文件中追加命令的错误输出，而不覆盖文件中的原有内容
输出与错误组合重定向	&>	将标准输出与错误输出的内容全部重定向到指定文件

重定向操作的例子（Linux）

- **将命令输出重定向到文件**
  - **n**将标准输出重定向到文件

```
$ ls /etc/ > etcdir
```

  - **n**将标准输出重定向追加到文件

```
$ ls /etc/sysconfig/ >> etcdir
```

  - **n**将错误输出重定向到文件

```
$ nocmd 2> errfile
```

  - **n**将标准输出和错误输出重定向到文件

```
$ ls afile bfile &> errfile 或者
$ ls afile bfile > ab.out 2>&1
```

管道 |

- **管道操作符 |**
  - **n** “|”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

```
cmd1 | cmd2
```

- **在同一条命令中可以使用多个“|”符连接多条命令**

```
cmd1 | cmd2 | ... | cmdn
```

Shell脚本编程

- **Shell脚本程序（ Shell Script ）**
  - Shell命令语句的集合，用于实现特定的功能；
    - 所有命令将逐行执行（按逻辑）。
    - 凡是能够在shell下直接执行的命令，都可以在脚本中使用。
    - 脚本中还可以使用一些不能在shell提示符下直接执行的语句，这些语句只有在脚本中使用才有效。
  - Shell脚本程序保存在文本文件中；
  - Shell脚本程序由Shell环境解释执行；
  - 执行Shell脚本文件需要具有可执行属性（x）。

运行脚本的方法

- **直接运行（用缺省版本的shell运行脚本程序）**
- **在命令提示符下使用某个特定版本Shell执行该脚本**

  $bash first_script

  - 指定一个特定shell版本(此例是bash)来执行该脚本
  - first_script逐行执行脚本中的命令并依次输出结果。
  - 当脚本文件中的命令依次执行完毕，该临时子shell也自动结束运行，返回到用户原来使用的shell状态。
- **在脚本文件的开始部分指定一个将要使用的shell**
  - 在脚本开头增加一行：

    #!/bin/bash　　——#!必须顶格，后接shell全路径
  - 可从/etc/shell获知所有可用shell及其绝对路径。

基本Shell脚本编程

- 使用文本编辑器（如vi）建立Shell脚本文件
- 脚本可以包括的内容：
  - 1）脚本语句（命令的有序集合，主要内容）

    history –c
  - 2）脚本运行环境设置

    #!/bin/bash （首行且顶格）
  - 3）注释行，以#开始

    # Clean command history,清除用户命令历史
- 设置脚本文件为可执行属性

脚本程序的编写

- 支持的语法
  - 变量
  - 条件测试和判断语句
  - 循环
  - 函数
  - 调试方法
- 运行脚本文件的注意事项
  - 1. 增加文件的执行权限

    chmod u+x MyShell
  - 2. 指定路径./

    $./MyShell

•统计文本文件的行数

```
#!/bin/bash
#
let COUNTS=0

echo "Please enter a file:"
read FILE

if [ -e $FILE -a -f $FILE ]; then
  while read LINE
    do
      echo $LINE
      COUNTS=$(($COUNTS+1))
    done < $FILE
  echo "There are $COUNTS lines."
fi
```

课后作业2

- **在Fedora/Ubantu中编写一个shell**
  - 功能从文件中读取每一行**显示**并**统计总行数**
    - 在shell运行过程中指定文件
  - 源代码+DOC文档或PPT文档
  - 要求有电脑屏幕截图
  - 下周同一时间抽查若2名学生上台汇报（8分钟）。
    - 请提前一天将源代码或DOC/PPT压缩发到 OSCourse@163.com
    - **缺交按缺勤1次算。**
    - 课后作业1和课后作业2任选一个

5. 系统调用

编程时利用**ReadFile**从文件中读

- 从文件当前位置读取**4**个字节，存入**chBuffer**

```
#021        const int BUFSIZE = 4096;
#022        char chBuffer[BUFSIZE];


#046        DWORD dwReadSize = 0;
#047        bRet = ::ReadFile(hFile,chBuffer,4,&dwReadSize,NULL);
#048        if (bRet)
#049        {
#050            //
#051            OutputDebugString(_T("ReadFile 读文件成功\r\n"));
#052        }
```

- **ReadFile函数功能**
  - 文件系统操作/磁盘操作
  - 从文件当前位置读取若干字节
  - 操作系统内核实现
- **ReadFile函数原型**

```
BOOL ReadFile (
    HANDLE hFile,                   //文件指针
    LPVOID lpBuffer,                //数据缓冲
    DWORD nNumberOfBytesToRead,     //要读取的字节数
    LPDWORD lpNumberOfBytesRead,    //已读取的字节数
    LPOVERLAPPED lpOverlapped       //覆盖缓冲 )
```

DOS：利用INT 21h 09号子功能输出字符串

- 例：显示字符串

```
string db 'Hello,Everybody !'
                          ; 定义要显示的字符串
    ...
mov ah,09h                ; ah←09h 号子功能
mov dx,offset string      ; dx←字符串的偏移地址
int 21h                   ;
```

DOS 21h中断子功能列表（部分）

00	程序终止(同 INT 20H)
01	键盘输入并回显
02	显示输出
03	异步通讯输入
04	异步通讯输出
05	打印机输出
06	直接控制台 I/O
07	键盘输入(无回显)
08	键盘输入(无回显) 检测 Ctrl-Break
09	显示字符串

Linux例子

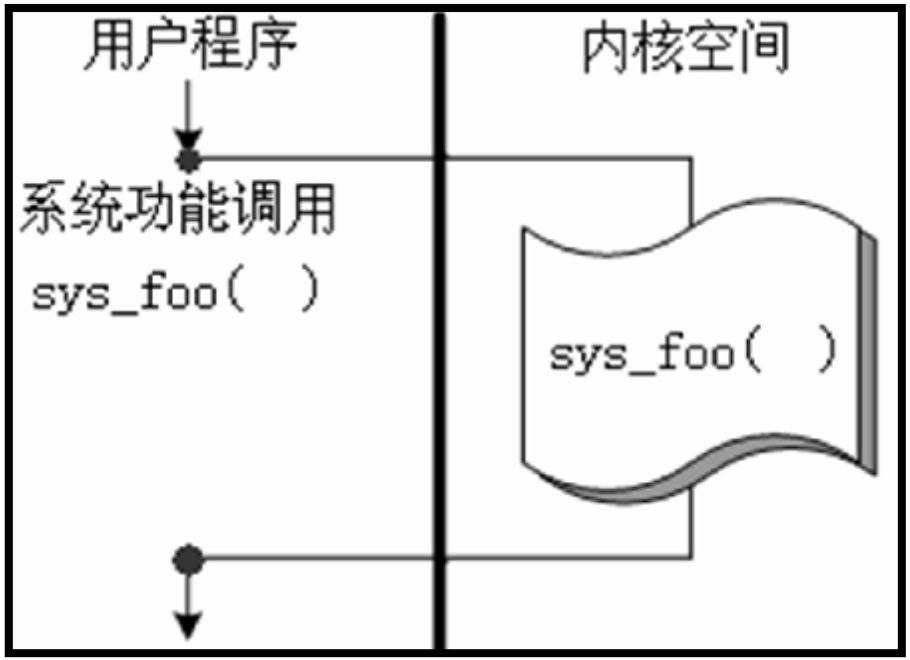
- 程序的功能
  - 打开一个文件
  - 显示文件内容
  - 关闭文件
- 系统函数列表
  - fopen
  - printf
  - fgetc
  - putchar
  - exit

```
1   #include <stdio.h>
2   main()
3   {
4       FILE *fp;
5       char ch;
6       if((fp=fopen("test","rt")) == NULL)
7       {
8           printf("\n Can't open file !");
9           exit(1);
10      }
11      ch = fgetc(fp);
12      while(ch != EOF)
13      {
14          putchar(ch);
15          ch = fgetc(fp);
16      }
17      fclose(fp);
18  }
```

系统调用

n 系统调用(System Call，System Service Call)

- n 操作系统内核为方便应用程序编程而提供的一系列服务/函数。这些服务/函数叫做系统调用/系统功能调用/System Call。



Windows的系统调用例子（部分）

```
NTSTATUS  WINAPI NtAcceptConnectPort(PHANDLE,ULONG,PLPC_MESSAGE,BOOLEAN,PLPC_SECTION_WRITE,PLPC_SECTION_READ);
NTSTATUS  WINAPI NtAccessCheck(PSECURITY_DESCRIPTOR,HANDLE,ACCESS_MASK,PGENERIC_MAPPING,PPRIVILEGE_SET,PULONG,PULONG,NTSTATUS*);
NTSTATUS  WINAPI NtAccessCheckAndAuditAlarm(PUNICODE_STRING,HANDLE,PUNICODE_STRING,PUNICODE_STRING,PSECURITY_DESCRIPTOR,ACCESS_MASK,PGENERIC_MAPPING,BOOLEAN,PULO
NG,PULONG,PBOOLEAN);
NTSTATUS  WINAPI NtAddAtom(const WCHAR*,ULONG,RTL_ATOM*);
NTSTATUS  WINAPI NtAdjustGroupsToken(HANDLE,BOOLEAN,PTOKEN_GROUPS,ULONG,PTOKEN_GROUPS,PULONG);
NTSTATUS  WINAPI NtAdjustPrivilegesToken(HANDLE,BOOLEAN,PTOKEN_PRIVILEGES,DWORD,PTOKEN_PRIVILEGES,PDWORD);
NTSTATUS  WINAPI NtAlertResumeThread(HANDLE,PULONG);
NTSTATUS  WINAPI NtAlertThread(HANDLE ThreadHandle);
NTSTATUS  WINAPI NtAllocateLocallyUniqueId(PLUID lpLuid);
NTSTATUS  WINAPI NtAllocateUuids(PULARGE_INTEGER,PULONG,PULONG);
NTSTATUS  WINAPI NtAllocateVirtualMemory(HANDLE,PVOID*,ULONG,SIZE_T*,ULONG,ULONG);
NTSTATUS  WINAPI NtCallbackReturn(PVOID,ULONG,NTSTATUS);
NTSTATUS  WINAPI NtCancelIoFile(HANDLE,PIO_STATUS_BLOCK);
NTSTATUS  WINAPI NtCancelTimer(HANDLE, BOOLEAN*);
NTSTATUS  WINAPI NtClearEvent(HANDLE);
NTSTATUS  WINAPI NtClose(HANDLE);
NTSTATUS  WINAPI NtCloseObjectAuditAlarm(PUNICODE_STRING,HANDLE,BOOLEAN);
NTSTATUS  WINAPI NtCompleteConnectPort(HANDLE);
NTSTATUS  WINAPI NtConnectPort(PHANDLE,PUNICODE_STRING,PSECURITY_QUALITY_OF_SERVICE,PLPC_SECTION_WRITE,PLPC_SECTION_READ,PULONG,PVOID,PULONG);
NTSTATUS  WINAPI NtContinue(PCONTEXT,BOOLEAN);
NTSTATUS  WINAPI NtCreateDirectoryObject(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES);
NTSTATUS  WINAPI NtCreateEvent(PHANDLE,ACCESS_MASK,const OBJECT_ATTRIBUTES *,BOOLEAN,BOOLEAN);
NTSTATUS  WINAPI NtCreateEventPair(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES);
NTSTATUS  WINAPI NtCreateFile(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES,PIO_STATUS_BLOCK,PLARGE_INTEGER,ULONG,ULONG,ULONG,ULONG,PVOID,ULONG);
NTSTATUS  WINAPI NtCreateIoCompletion(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES,ULONG);
```

Windows的系统调用例子（部分）

```
NTSTATUS  WINAPI NtOpenDirectoryObject(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES);
NTSTATUS  WINAPI NtOpenEvent(PHANDLE,ACCESS_MASK,const OBJECT_ATTRIBUTES *);
NTSTATUS  WINAPI NtOpenEventPair(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES);
NTSTATUS  WINAPI NtOpenFile(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES,PIO_STATUS_BLOCK,ULONG,ULONG);
NTSTATUS  WINAPI NtOpenIoCompletion(PHANDLE,ACCESS_MASK,POBJECT_ATTRIBUTES);
NTSTATUS  WINAPI NtOpenKey(PHANDLE,ACCESS_MASK,const OBJECT_ATTRIBUTES *);
NTSTATUS  WINAPI NtOpenMutant(PHANDLE,ACCESS_MASK,const OBJECT_ATTRIBUTES*);
NTSTATUS  WINAPI NtOpenProcess(PHANDLE,ACCESS_MASK,const OBJECT_ATTRIBUTES*,const CLIENT_ID*);
NTSTATUS  WINAPI NtRaiseHardError(NTSTATUS,ULONG,PUNICODE_STRING,PVOID*,HARDERROR_RESPONSE_OPTION,PHARDERROR_RESPONSE);
NTSTATUS  WINAPI NtReadFile(HANDLE,HANDLE,PIO_APC_ROUTINE,PVOID,PIO_STATUS_BLOCK,PVOID,ULONG,PLARGE_INTEGER,PULONG);
NTSTATUS  WINAPI NtWaitLowEventPair(HANDLE);
NTSTATUS  WINAPI NtWriteFile(HANDLE,HANDLE,PIO_APC_ROUTINE,PVOID,PIO_STATUS_BLOCK,const void*,ULONG,PLARGE_INTEGER,PULONG);
NTSTATUS  WINAPI NtWriteFileGather(HANDLE,HANDLE,PIO_APC_ROUTINE,PVOID,PIO_STATUS_BLOCK,FILE_SEGMENT_ELEMENT,ULONG,PLARGE_INTEGER,PULONG);
```

……其余省略……

Linux的系统调用（部分）

进程控制：

fork	创建一个新进程
clone	按指定条件创建子进程
execve	运行可执行文件
exit	中止进程
_exit	立即中止当前进程
getdtablesize	进程所能打开的最大文件数
getpid	获取进程标识号
getppid	获取父进程标识号
getpriority	获取调度优先级

文件读写操作

fcntl	文件控制
open	打开文件
creat	创建新文件
close	关闭文件描述字
read	读文件
write	写文件
readv	从文件读入数据到缓冲数组中

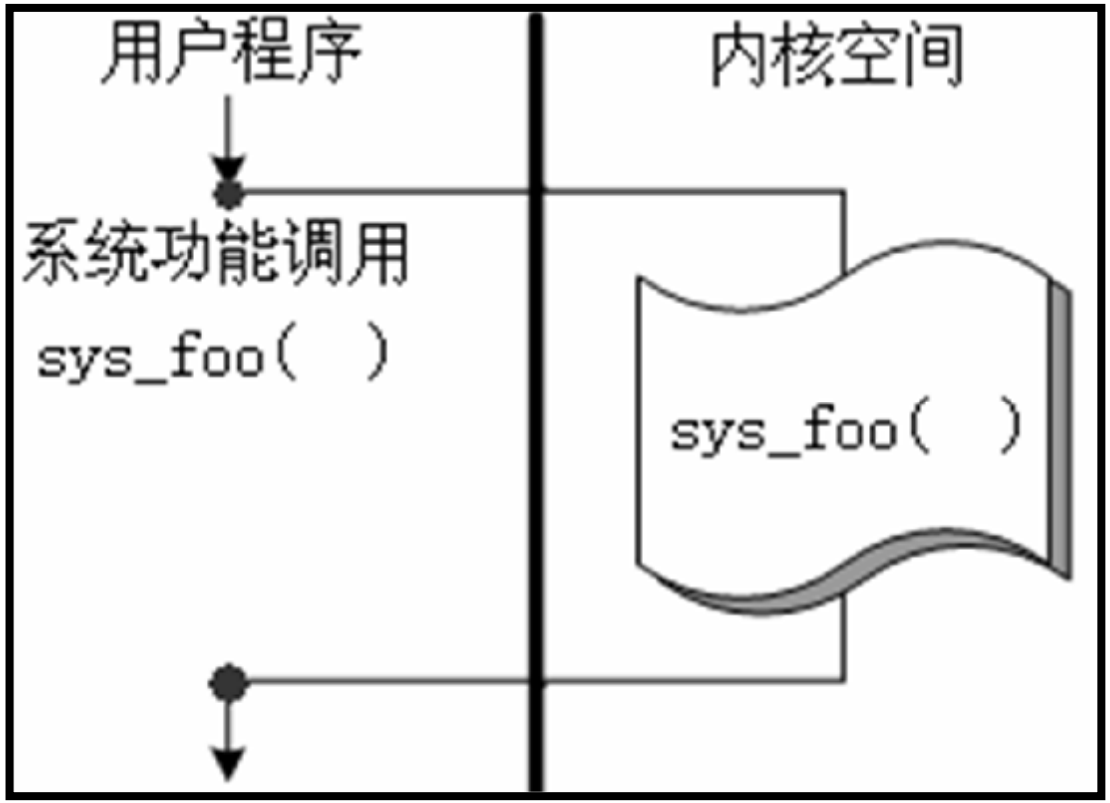
DOS的系统调用列表（部分）

00	程序终止(同 INT 20H)
01	键盘输入并回显
02	显示输出
03	异步通讯输入
04	异步通讯输出
05	打印机输出
06	直接控制台 I/O
07	键盘输入(无回显)
08	键盘输入(无回显) 检测 Ctrl-Break
09	显示字符串

39	建立子目录(MKDIR)
3A	删除子目录（RMDIR）
3B	改变当前目录(CHDIR)
3C	建立文件
3D	打开文件
3E	关闭文件
3F	读文件或设备
40	写文件或设备
41	删除文件
42	移动文件指针

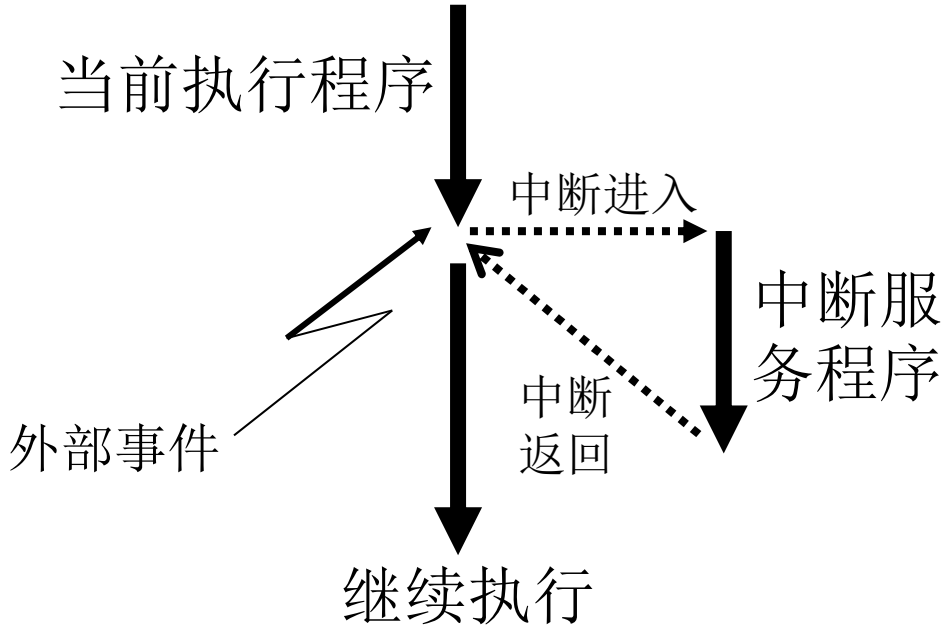
系统调用的特点

- 一般涉及核心资源的操作或底层功能的操作
- 由操作系统内核完成服务，运行于核态
- 调用过程会产生中断



回顾**中断机制**

- **中断定义**
  - 指CPU对突发的外部事件的反应过程或机制。
  - CPU收到外部信号（中断信号）后，停止当前工作，转去处理该外部事件，处理完毕后回到原来工作的中断处（断点）继续原来的工作。
- **引入中断的目的**
  - 实现并发活动
  - 实现实时处理
  - 故障自动处理



中断的一些概念

- **中断源和中断类型**
  - 引起系统中断的事件称为中断源
  - 中断类型
    - 强迫性中断和自愿中断
      - 强迫性中断：程序没有预期：例：I/O、外部中断
      - 自愿中断：程序有预期的。例：执行访管指令
    - 外中断（中断）和内中断（俘获）
      - 外中断：由CPU外部事件引起。例：I/O，外部事情。
      - 内中断：由CPU内部事件引起。例：访管中断、程序中断
    - 外中断：可屏蔽中断和不可屏蔽中断
      - 不可屏蔽中断：中断的原因很紧要，CPU必须响应
      - 可屏蔽中断：中断原因不很紧要，CPU可以不响应

中断响应过程

- **中断响应过程**
  - (1)识别中断源
  - (2)保护断点和现场
  - (3)装入中断服务程序的入口地址（ CS:IP ）
  - (4)进入中断服务程序
  - (5)恢复现场和断点
  - (6)中断返回： IRET

中断的一些概念

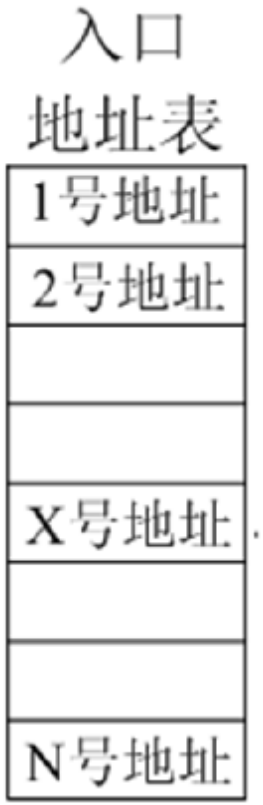
- 断点
  - 程序中断的地方，将要执行的下一指令的地址
  - CS:IP
- 现场
  - 程序正确运行所依赖的信息集合。
    - PSW（程序状态字）、PC、相关寄存器
    - 部分内存数据
- 现场的两个处理过程
  - 现场的保护：进入中断服务程序之前，栈
  - 现场的恢复：退出中断服务程序之后，栈

- **中断响应的实质**
  - 交换指令执行地址
  - 交换CPU的态
  - 工作
    - 现场保护和恢复
    - 参数传递（通信）

继续：系统调用的原理

l 系统调用的原理

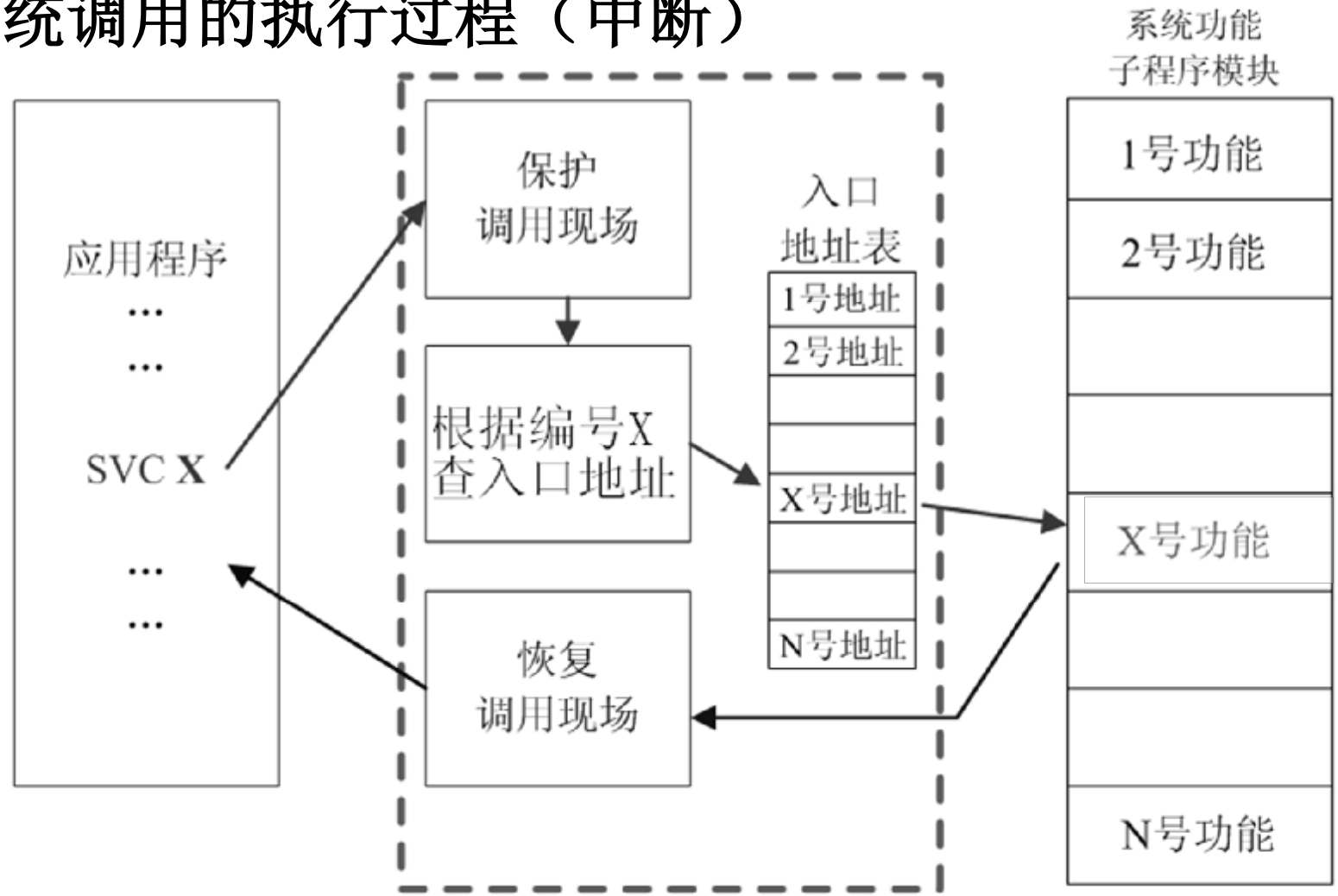
- n 系统调用的函数在**核态**执行。
- n 应用程序工作在**用户态**。
- n 应用程序通过**中断**机制才能进入核态。
  - u 自愿中断
  - u 每个系统调用具有唯一编号ID
  - u OS为系统调用维护入口地址（表）



系统调用的实现形式

- 调用N号系统调用，使用指令：`SVC N`
  - N：系统调用的编号
  - SVC：SuperVisor Call，访管指令
  - SVC是中断指令

系统调用的执行过程（中断）



- 具体OS中系统调用的实现
  - DOS： INT 21H
  - LINUX：INT 80H
  - Windows 2K： INT 2EH （所有NT内核的系列）
  - Windows XP ：SYSENTER （WindowsXP及以上）
  - 注意：
    - 上述指令都是中断指令，等同于“SVC”指令
    - 中断之前通过设置参数指定系统功能调用编号N。

各种OS系统调用的例子

- DOS的例子
- 例：显示字符串（系统调用编号**09h**）

```
string db 'Hello,Everybody !'
                          ; 定义要显示的字符串
    ...
mov ah,09h                ; 指定编号：ah←09h
mov dx,offset string      ; dx←字符串的偏移地址
int 21h                   ; 系统调用
```

```
INT 21H = SVC ;
09h = N
```

I 例：输入字符（系统调用编号01h），判断按键Y/N？

```
GetKey:   mov ah,01h          ; 指定编号：ah←01h
          int 21h             ; 系统调用
          cmp al, 'Y'         ; 处理出口参数al
          je  YesKey          ; 是 "Y"
          cmp al, 'N'
          je  NoKey           ; 是 "N"
          jne GetKey
            ...
YesKey:     ...（处理Y键）  ...
NoKey:      ...（处理N键）  ...
```

DOS的系统调用列表（部分）

00	程序终止(同 INT 20H)
01	键盘输入并回显
02	显示输出
03	异步通讯输入
04	异步通讯输出
05	打印机输出
06	直接控制台 I/O
07	键盘输入(无回显)
08	键盘输入(无回显) 检测 Ctrl-Break
09	显示字符串

39	建立子目录(MKDIR)
3A	删除子目录（RMDIR）
3B	改变当前目录(CHDIR)
3C	建立文件
3D	打开文件
3E	关闭文件
3F	读文件或设备
40	写文件或设备
41	删除文件
42	移动文件指针

各种OS系统调用的例子

- Linux的例子 （在屏幕上输出Hello World）

```
mov eax, 4      ;系统调用编号：4号（文件写）
mov ebx, 1      ;ebx送1表示stdout
mov ecx, msg ;字符串的首地址送入ecx
mov edx, 14     ;字符串的长度送入edx
int 80h         ;输出字串
mov eax,1       ;系统调用编号：1号（进程结束）
int 80h         ;结束
msg: db "Hello World!"
```

```
INT 80H = SVC ;
4 = N
```

l 在unistd.h文件中定义系统调用的编号（前缀 __NR_）

```
#define __NR_exit              1
#define __NR_fork              2
#define __NR_read              3
#define __NR_write             4
#define __NR_open              5
#define __NR_close             6
#define __NR_waitpid           7
#define __NR_creat             8
#define __NR_link              9
```

各种OS系统调用的例子

- **Win2K**
- 例子：获取本机系统信息（进程信息，编号**0x97**）

```
NtQuerySystemInformationNo = 0x97;
_asm
    {
        mov eax, NtQuerySystemInformationNo
        lea edx, [esp+4]
        int 2eh
        ret 10h
     }
```

```
INT 2eH = SVC ;
97h = N
```

//0x64 openfile; 0xed writefile; 0x29 createprocess

- 系统调用的隐式调用方式
  - 通过API（Application Program Interface）方式调用
    - 例 open，write，printf，return等函数都是API

```
file* fp = open(“c:/test.txt”); //打开文件
return 0;
```

    - 高级语言（C，Vb，Delphi，Java）中使用
    - 编译时转化为相应的显示调用（INT指令）

隐式调用转化为显示调用

- 编译时隐式调用转化为显示调用（**INT指令**）
- 例子：**printf, exit**

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

```
1  /* shellcode.c */
2  void main()
3  {
4      __asm__ __volatile__("jmp forward;"
5                  "backward:"
6                      "popl   %esi;"
7                      "movl   $4, %eax;"
8                      "movl   $1, %ebx;"
9                      "movl   %esi, %ecx;"
10                     "movl   $12, %edx;"
11                     "int    $0x80;"
12                     "movl   $1, %eax;"
13                     "movl   $0, %ebx;"
14                     "int    $0x80;"
15                 "forward:"
16                     "call   backward;"
17                     ".string \"Hello World\\n\";");
18 }
```

4→write

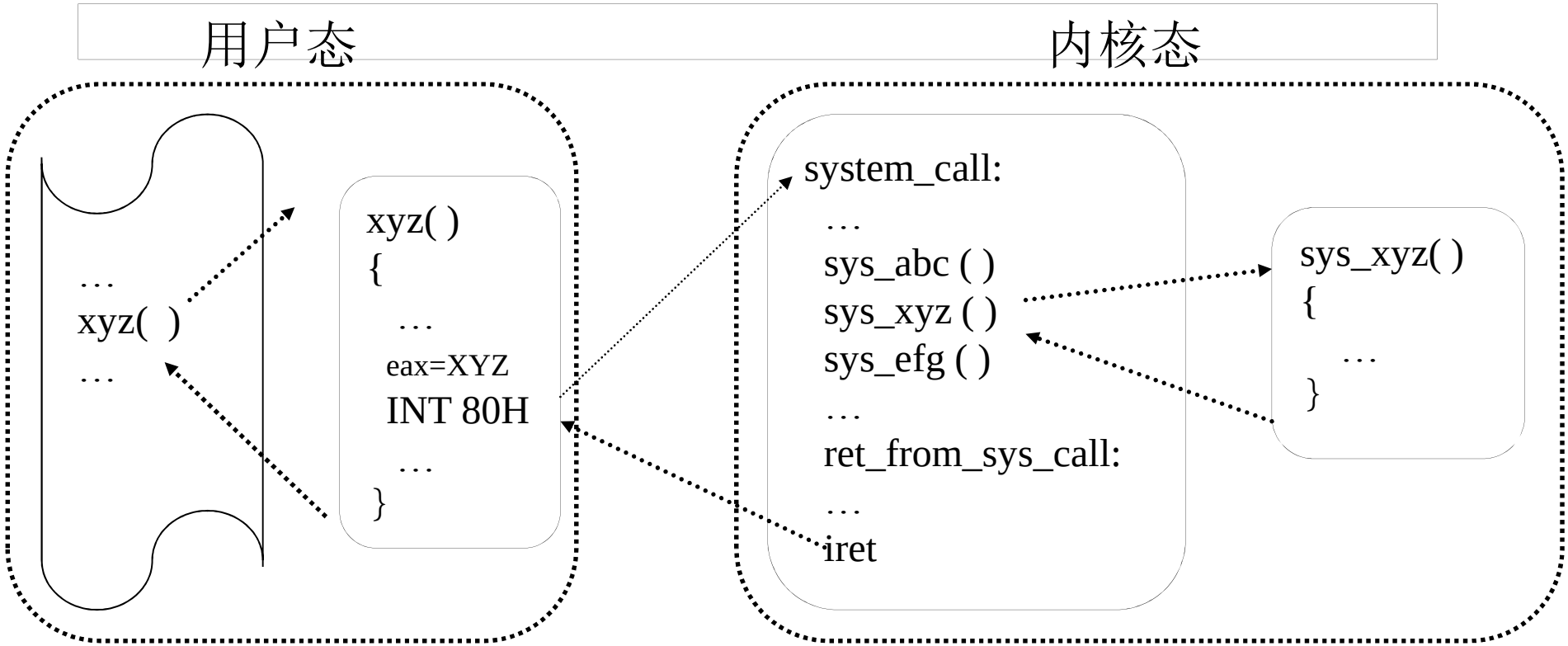
1→exit

5. Linux系统调用的内部机制

LINUX系统调用的实质

- 系统调用是Linux内核的出口，给用户程序调用的一组“特殊”接口。
- 系统调用采用API方式向用户提供，遵循 POSIX标准
- 系统调用通过软中断(INT 80H)向内核发出服务请求

Linux系统调用的工作原理



应用程序中调用系统调用相应的API函数xyz( )

→

xyz( )在Libc中实现，含有INT 0x80指令

→

system_call是0x80号中断的服务程序，其中包括所有系统调用函数的入口：sys_xyz( )

→

根据中断时指定的系统调用编号（函数名xyz）选择相应sys_xyz( )

80H中断的服务程序system_call( )

- **system_call( )**

```
{
    //.................
    //寄存器%eax存放有系统调用编号，
    //以其为索引遍历sys_call_table，查找对应的服务子程序。
    call *SYMBOL_NAME(sys_call_table)(%eax, 4)
}
```

SYS_CALL_TABLE的结构

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

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

系统调用编号的声明

- **Linux-source-2.6.38**
  - **n** ./arch/x86/include/asm/unistd_32.h
- 系统调用编号的声明

  格式： **#define __NR_CallName ID**

```
#define __NR_perf_event_open    336
#define __NR_recvmmsg           337
#define __NR_fanotify_init      338
#define __NR_fanotify_mark      339
#define __NR_prlimit64          340
#define __NR_mycall             341
#define __NR_addtotal           342
#define __NR_three              343
#ifdef __KERNEL__
```

系统调用函数的声明

- **Linux-source-2.6.38**

  n ./arch/x86/kernel/syscall_table_32.S

- 系统调用函数的声明

  .long sys_XXXX

  注意：1） XXX是函数名，有前缀sys_修饰。

  2）添加位置是末尾，且注意添加顺序。

```
.long sys_rt_tgsigqueueinfo     /* 335 */
.long sys_perf_event_open
.long sys_recvmmsg
.long sys_fanotify_init
.long sys_fanotify_mark
.long sys_prlimit64             /* 340 */
.long sys_mycall
.long sys_addtotal
.long sys_three
```

系统调用函数的定义

- 注意格式 **asmlinkage**
  - **n** ./kernel/sys.c（ Linux-source-2.6.38）

```
asmlinkage int sys_mycall(int number)
{
 printk("这是我添加的第一个系统调用");
 return number;
}
asmlinkage int sys_addtotal(int number)
{
 int i=0,enddate=0;
 printk("这是我添加的第二个系统调用");
 while(i<=number)
 enddate+=i++;
 return enddate;
}
asmlinkage int sys_three()
{
 printk("这是我添加的第三个系统调用");
 return 0;
}
```

系统调用函数的调用方法

- 早先的版本
  - 先声明（N：参数个数； FuncName不带sys_前缀）

```
#define  _syscallN ( type, FuncName ,
                type1, arg1,
                type2, arg2,  ……
                typeN, argN )
```

  - 后调用： type = FuncName (arg1, arg2, …)
- 较新的版本
  - 直接调用 （FuncName 不带sys_前缀）

```
type = FuncName(__NR_funcname, arg1,  arg2, …)
```

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（声明系统调用函数和编号）
  - 3、编译新LINUX内核
  - 4、编译和安装模块
  - 5、启动新的LINUX内核
  - 6、编写应用程序测试新的系统调用
- 建议环境
  - UBANTU/Fedora
  - 开源内核2.6.38或其它

1、编写新的系统调用函数（指函数实现部分）

- 函数命名规则：“**sys_**”标志。
- 在./kernel/sys.c中实现

```
asmlinkage int sys_mycall(int number)
{
        return number + 100 ;
}
```

l 2、注册新的系统调用

n 定义系统调用的编号： __**NR**_调用名。

./arch/x86/include/asm/unistd_32.h

本例如此添加：

```
#define    __NR_mycall      341
```

n 声明系统调用函数： .long sys_XXX

./arch/x86/kernel/syscall_table_32.S

注意：1） XXX是函数名，必须用前缀sys_修饰。

2）添加位置现有函数的末尾，特别注意添加顺序。

本例如此添加：

```
.long sys_mycall
```

3、编译新的LINUX内核

- 重建内核

```
#make dep
#make clearn
#make bzImage-2.6.38-mykernel
```

- 内核映象: ./arch/i386/boot/bzImage-2.6.38-mykernel
- 拷贝到: /boot/bzImage-2.6.38-mykernel

4、配置新的LINUX内核

- 修改/boot/grub/grub.cfg或/boot/grub/menu.lst

  linux /boot/bzImage-2.6.38-mykernel
  initrd /boot/initrd.img-2.6.38

- reboot

5、编写应用程序测试新的系统调用

- 在应用程序中使用新添加的系统调用mycall

```
main( )
{
        printf("%d n", mycall(341,5));  // "105"
}
```

```
asmlinkage int sys_mycall(int number)
{
        return number + 100 ;
}
```