

操作系统2022

# 第2讲 绪论 (2)

清华大学电子工程系

马洪兵

# 大纲

**01**

操作系统的用户接口

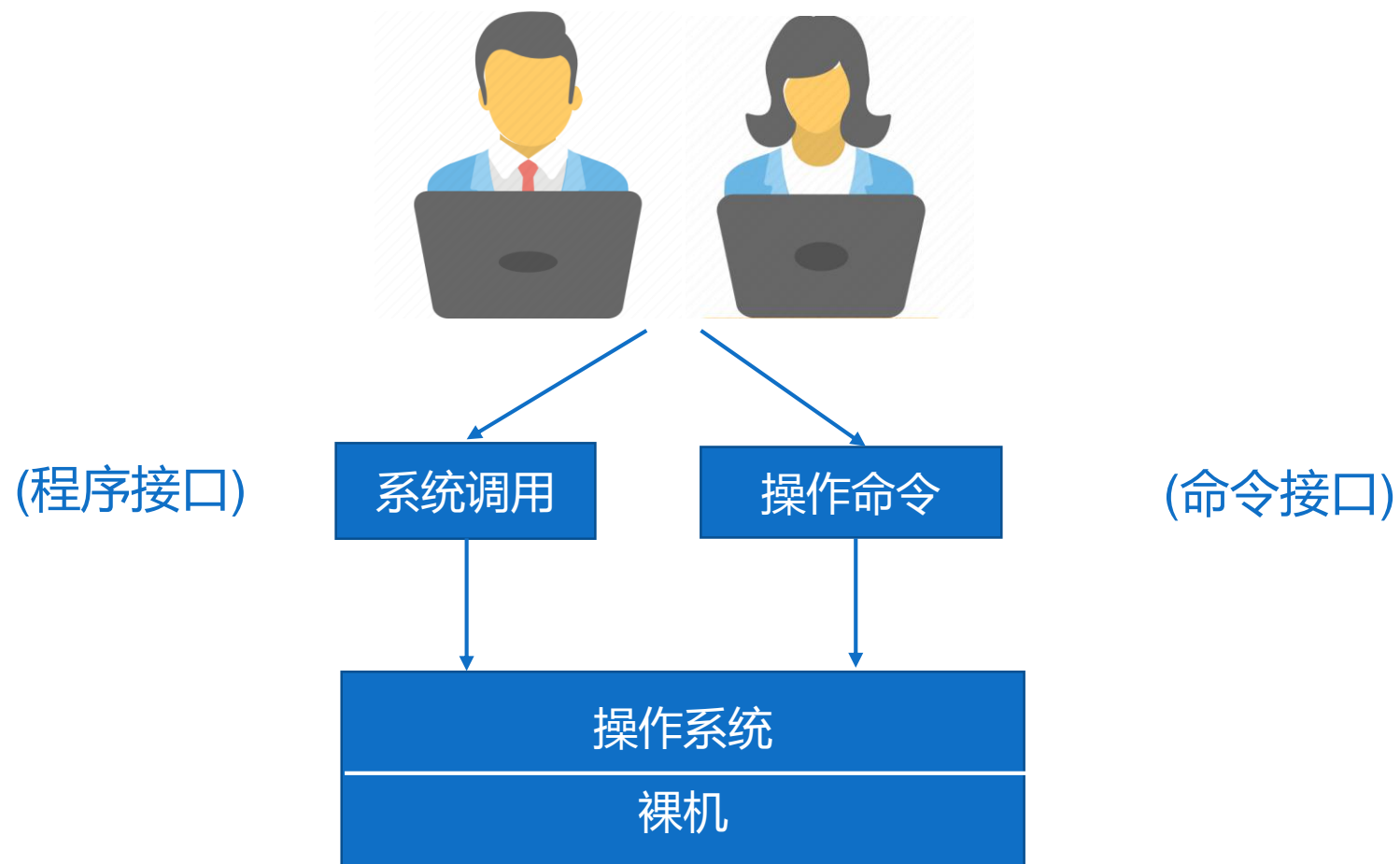
**02**

操作系统的设计与实现

**03**

操作系统体系结构

# 操作系统的用户接口



# 操作命令接口

---

- 操作命令接口是操作系统为用户操作控制计算机工作和提供服务的手段  
的集合
- 实现手段
  - 批处理系统提供的作业控制语言 (JCL)
  - 交互式系统的操作控制命令

# 作业控制语言(命令)

---

- 这种接口是专为批处理作业用户提供的，也称批处理用户接口，是一种脱机用户接口
- 操作系统提供了一个作业控制语言JCL (Job Control Language)，它由一组作业控制卡片，或作业控制语句，或作业控制操作命令组成
- 作业控制语言是一种语言，用来编写程序操作步骤的程序

# 作业控制语言(命令)

---

## ■ 工作方式

- 用户使用JCL语句，把运行意图（需要对作业进行的控制和干预）写在作业说明书上，将作业连同作业说明书一起提交给系统
- 批处理作业被调度执行时，系统调用JCL语句处理程序或命令解释程序对作业说明书进行解释处理，完成对作业的运行和控制
- 在批处理操作系统时代，用户采用脱机方式使用计算机，即用户将自己的程序、数据和用JCL编写的描述上机操作步骤的程序一起提交给计算中心，隔一段时间去机房取结果

# 操作控制命令

---

- 由一组命令及命令解释程序组成，是一种联机用户接口
- 工作方式为用户在键盘上键入一条命令后，系统立即转入命令解释程序，对该命令进行处理和执行
- 不同的用法和形式组成了不同的用户界面
  - 字符显示用户界面
  - 图形化用户界面

# 字符显示用户界面

---

- 主要通过命令语言来实现，又可分成两种方式：
  - 命令行方式
  - 脚本方式



# 命令行方式

- 命令行方式是以命令为基本单位来完成预定的工作任务
- 每个命令以命令行的形式输入并提交给系统
- 一个命令行由命令动词和一组参数构成，其一般形式

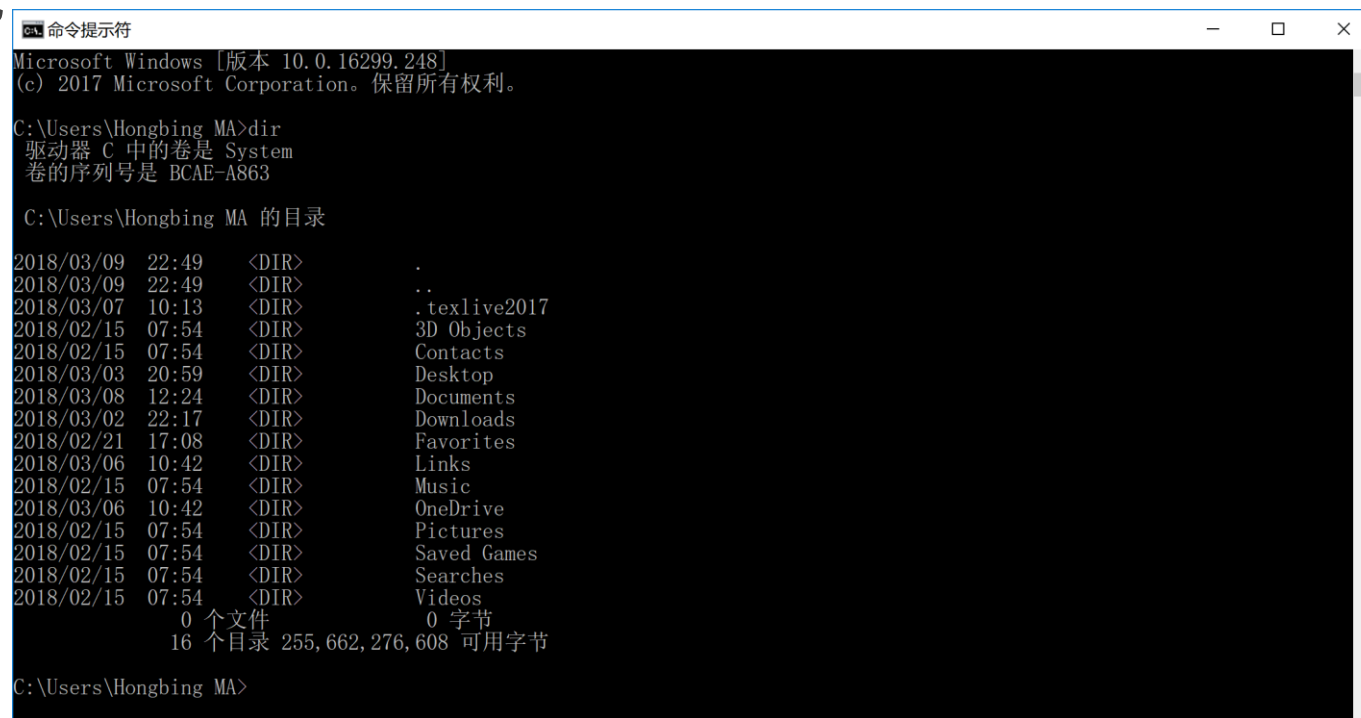
```
command arg1 arg2 ... argn
```

- command——命令名，又称命令动词
- arg1, arg2, ..., argn——命令参数

# 命令行方式

## ■ Windows

- cmd: 命令提示符, Windows的命令解释程序, 操作方式类似于DOS操作系统
- Windows PowerShell: 专为系统管理员设计的命令行 shell, 支持交互式提示和脚本环境



```
命令提示符
Microsoft Windows [版本 10.0.16299.248]
(c) 2017 Microsoft Corporation. 保留所有权利。

C:\Users\Hongbing MA>dir
驱动器 C 中的卷是 System
卷的序列号是 BCAE-A863

C:\Users\Hongbing MA 的目录

2018/03/09  22:49    <DIR>          .
2018/03/09  22:49    <DIR>          ..
2018/03/07  10:13    <DIR>          .texlive2017
2018/02/15  07:54    <DIR>          3D Objects
2018/02/15  07:54    <DIR>          Contacts
2018/03/03  20:59    <DIR>          Desktop
2018/03/08  12:24    <DIR>          Documents
2018/03/02  22:17    <DIR>          Downloads
2018/02/21  17:08    <DIR>          Favorites
2018/03/06  10:42    <DIR>          Links
2018/02/15  07:54    <DIR>          Music
2018/03/06  10:42    <DIR>          OneDrive
2018/02/15  07:54    <DIR>          Pictures
2018/02/15  07:54    <DIR>          Saved Games
2018/02/15  07:54    <DIR>          Searches
2018/02/15  07:54    <DIR>          Videos
                0 个文件             0 字节
                16 个目录 255,662,276,608 可用字节

C:\Users\Hongbing MA>
```

## Linux/Unix—shell



# 脚本方式

---

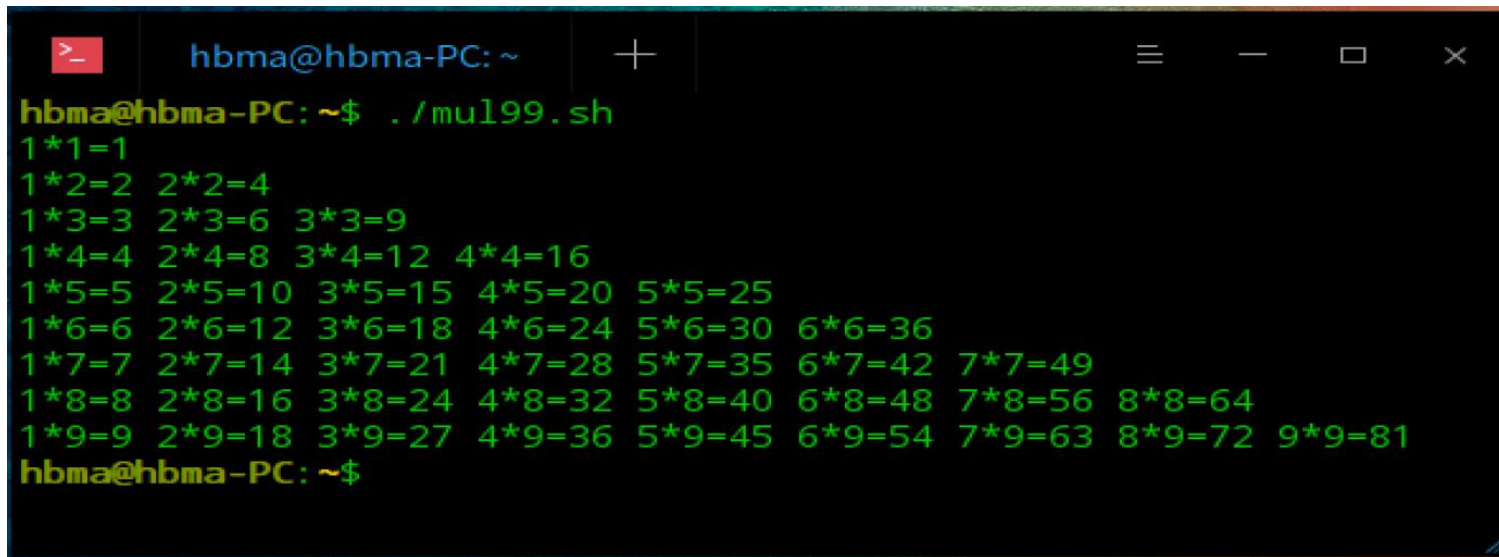
- 由批处理系统作业控制语言发展而来
- 规定一种特别的文件，用户可预先把一系列命令组织在该类文件中，一次建立，多次执行
- Windows称这种文件为**批命令文件**，具有特殊的文件扩展名.BAT
- Linux/Unix称这种文件为**脚本文件**
- 操作系统还提供一套控制子命令，增强命令接口的处理能力

# 脚本方式

## ■例：输出九九乘法表（Linux shell脚本）

```
#!/bin/bash

for i in `seq 9`
do
    for j in `seq $i`
    do
        echo -n "$j*$i=${i*j} "
    done
    echo
done
```



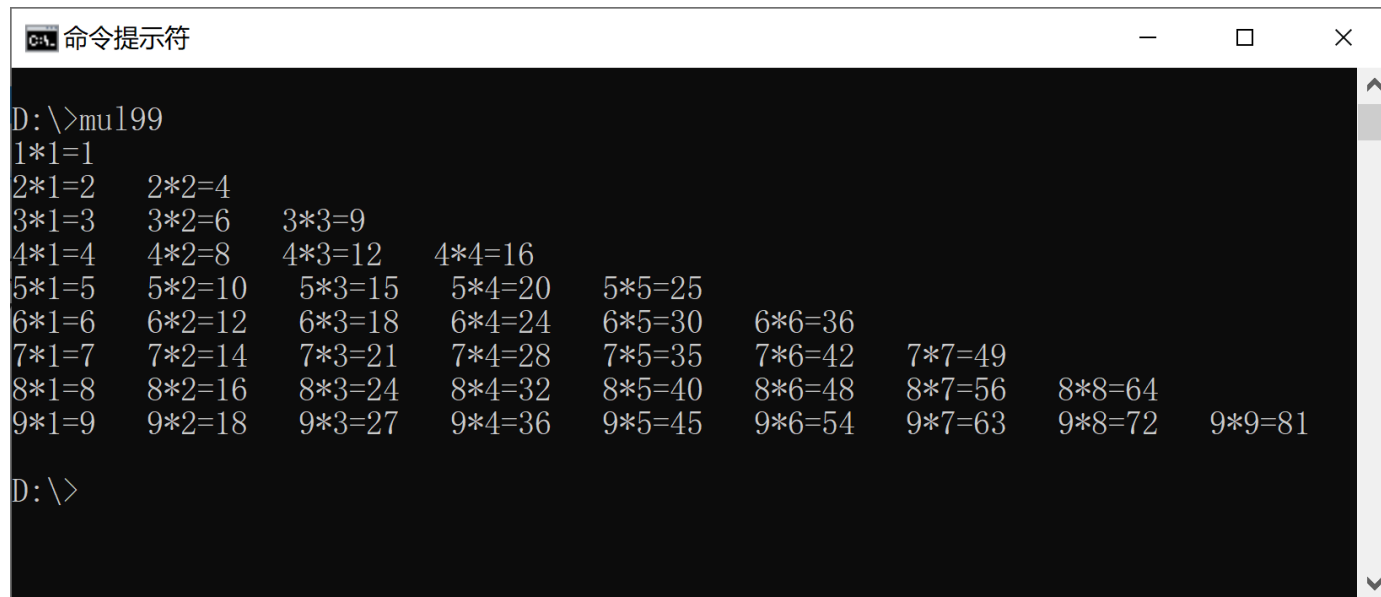
A terminal window titled 'hbma@hbma-PC: ~' showing the execution of a script named 'mul99.sh'. The script outputs a 9x9 multiplication table. The prompt is 'hbma@hbma-PC: ~\$ ./mul99.sh'. The output is as follows:

```
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
hbma@hbma-PC: ~$
```

# 脚本方式

## ■例：输出九九乘法表（Windows bat脚本）

```
@echo off
setlocal enabledelayedexpansion
for /l %%i in (1,1,9) do (
    for /l %%j in (1,1,%%i) do (
        set /a mul=%%i*%%j
        set /p =%%i*%%j=!mul! <nul
    )
)
```



The screenshot shows a Windows Command Prompt window titled "命令提示符" (Command Prompt). The prompt is at "D:\>". The user has entered the command "mul99". The output is a 9x9 multiplication table displayed in a grid format. The first row is "1\*1=1". The second row is "2\*1=2 2\*2=4". The third row is "3\*1=3 3\*2=6 3\*3=9". The fourth row is "4\*1=4 4\*2=8 4\*3=12 4\*4=16". The fifth row is "5\*1=5 5\*2=10 5\*3=15 5\*4=20 5\*5=25". The sixth row is "6\*1=6 6\*2=12 6\*3=18 6\*4=24 6\*5=30 6\*6=36". The seventh row is "7\*1=7 7\*2=14 7\*3=21 7\*4=28 7\*5=35 7\*6=42 7\*7=49". The eighth row is "8\*1=8 8\*2=16 8\*3=24 8\*4=32 8\*5=40 8\*6=48 8\*7=56 8\*8=64". The ninth row is "9\*1=9 9\*2=18 9\*3=27 9\*4=36 9\*5=45 9\*6=54 9\*7=63 9\*8=72 9\*9=81". The prompt is now "D:\>".

```
D:\>mul99
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
D:\>
```

# 命令解释程序

---

- 接受和执行用户输入的命令
- 当新的批作业被启动，或新的交互型用户登录时，系统就自动地执行命令解释程序，它负责读入控制卡或命令行，并作出相应解释和执行
- Windows——cmd、Windows PowerShell
- Linux/Unix——各种shell

# 命令解释程序

---

- 命令解释程序执行的命令有内部命令和外部命令之分
- **内部命令**实际上是命令解释程序的一部分，其中包含的是一些比较简单的系统命令，这些命令由命令解释程序识别并在命令解释程序内部完成运行，系统加载时内部命令就随同命令解释程序加载并驻留在系统内存中
- **外部命令**并不随系统一起被加载到内存中，而是在需要时才将其调进内存。通常外部命令的实体并不包含在命令解释程序中，但是其命令执行过程是由命令解释程序控制的。命令解释程序管理外部命令执行的路径查找、加载存放，并控制命令的执行



# 命令解释程序

---

## ■ 命令解释程序的处理过程

- 系统启动命令解释程序，输出命令提示符，等待键盘中断。用户打入命令并按回车换行，申请键盘中断
- CPU响应后，控制权交给命令解释程序，它读入命令缓冲区内容，分析命令、接受参数
- 若为内部命令立即转向命令处理代码执行。否则查找命令处理文件，装入主存，传递参数，将控制权交给其执行
- 命令处理结束后，再次输出命令提示符，等待下一条命令

# 图形用户接口

---

- 图形用户接口GUI (Graphics User Interface) 使用WIMP界面, 即窗口 (Window)、图标 (Icon)、菜单 (Menu) 和指点设备 (Pointing Device) 等技术, 将系统的功能、各种应用程序和文件用图形符号直观、逼真地表示出来, 用户可通过选择窗口、菜单、对话框和滚动条完成对它们的作业的各种控制 and 操作
- 图形化的用户界面的特点:
  - 所有程序以统一的窗口形式出现
  - 提供统一的菜单格式
  - 系统资源、系统命令、操作功能以图标表示
  - 统一的操作方法

# 图形用户接口

- 1973年施乐公司帕洛阿尔托研究中心（Xerox PARC）最先建构了WIMP
- 20世纪80年代苹果公司首先将图形用户界面引入微机领域，推出的Macintosh以其全鼠标、下拉菜单操作和直观的图形界面，引发了微机人机接口的历史性的变革



# 新一代用户接口

- 虚拟现实技术的研究和应用，多感知通道用户接口，自然化用户接口，智能化用户接口的研究
  - 头盔显示器
  - 数据手套



# 系统调用接口

---

- 操作系统和用户程序之间的接口是通过操作系统提供的一套系统调用(System Call)来定义的
- 对应用程序而言，操作系统内核的作用体现为可以供其调用的一组函数——系统调用
- 系统调用也称为系统服务(System Service)

# 系统调用的作用

---

- 为了管理硬件资源和为应用程序开发人员提供良好的环境，同时使应用程序具有更好的兼容性
- 为了安全问题，一些I/O操作的指令都被限制在只有内核模式可以执行，因此操作系统有必要为应用程序提供访问硬件设备的接口

# 系统调用的处理过程

---

- 系统调用与过程调用不同
- 为了保证OS不被用户程序破坏，不允许用户程序访问OS的系统程序和数据。那么，怎样得到操作系统提供的服务呢？
- 需要有一个类似于硬件中断处理的处理机构，当用户使用系统调用时，使系统进入核心态

# 系统调用的处理过程

---

- 为使CPU能够主动进入核心态，每种机器的指令集中都提供了陷入(Trap)指令，这个指令能够将系统转入核心态
- 系统调用由陷入指令实现
- 系统调用是操作系统提供给编程人员的唯一接口
- 利用系统调用，可以动态请求和释放系统资源，完成与硬件相关的工作以及控制程序的执行等
- 每个操作系统都提供几百种系统调用



# 系统调用的处理过程

- 在操作系统中，每个系统调用都对应一个事先给定的功能号，例如0、1、2、3
- 陷入指令中包括对应系统调用的功能号
- 在有些陷入指令中，还带有传给系统调用内部处理程序的有关参数

```
<arch\x86\include\asm\unistd_32.h>
1 #ifndef __ASM_X86_UNISTD_32_H
2 #define __ASM_X86_UNISTD_32_H
3 /*
4  * This file contains the system call numbers.
5  */
6 #define __NR_restart_syscall 0
7 #define __NR_exit 1
8 #define __NR_fork 2
9 #define __NR_read 3
10 #define __NR_write 4
11 #define __NR_open 5
12 #define __NR_close 6
13 #define __NR_waitpid 7
```

# 系统调用的处理过程

- 需要为实现系统调用功能的内核函数（子程序）编造入口地址表
- 陷入处理机构把陷入指令包含的功能号与入口地址表有关项对应，系统调用功能号驱动有关内核函数执行
- 在系统调用处理结束之后，用户程序需利用系统调用返回结果继续执行

```
<arch\x86\kernel\syscall_table_32.s>
```

```
ENTRY(sys_call_table)
.long sys_restart_syscall /* 0 - old "setup()" system call, used for restarting */
.long sys_exit
.long ptregs_fork
.long sys_read
.long sys_write
.long sys_open /* 5 */
.long sys_close
.long sys_waitpid
.long sys_creat
.long sys_link
.long sys_unlink /* 10 */
```

```
<arch\x86\kernel\entry_32.s>
```

```
syscall_call:
    call *sys_call_table(,%eax,4)
```

# 系统调用的处理过程

## ■保护现场：

- 进入系统调用处理前，陷入处理机构还需保存处理机现场
- 在系统调用处理结束之后，要恢复处理机现场，现场被保护在特定的内存区或寄存器中

```
<arch\x86\kernel\entry_32.s>
```

```
ENTRY(system_call)
```

```
    pushl %eax # save orig_eax
```

```
    SAVE_ALL
```

```
    GET_CURRENT(%ebx)
```

```
    cmpl $(NR_syscalls),%eax
```

```
    jae badsys
```

```
    testb $0x20,flags(%ebx) # PF_TRACESYS
```

```
    jne tracesys
```

```
<arch\x86\kernel\entry_32.s>
```

```
ENTRY(ret_from_sys_call)
```

```
    cli          # need_resched and signals atomic test
```

```
    cmpl $0,need_resched(%ebx)
```

```
    jne reschedule
```

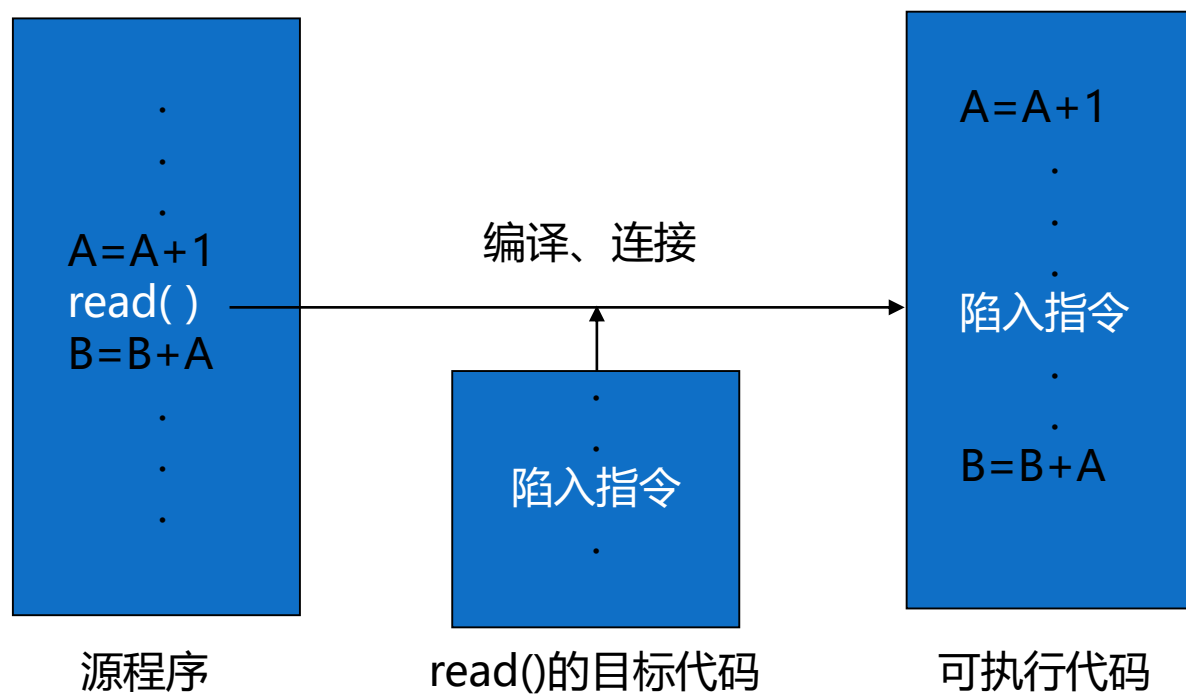
```
    cmpl $0,sigpending(%ebx)
```

```
    jne signal_return
```

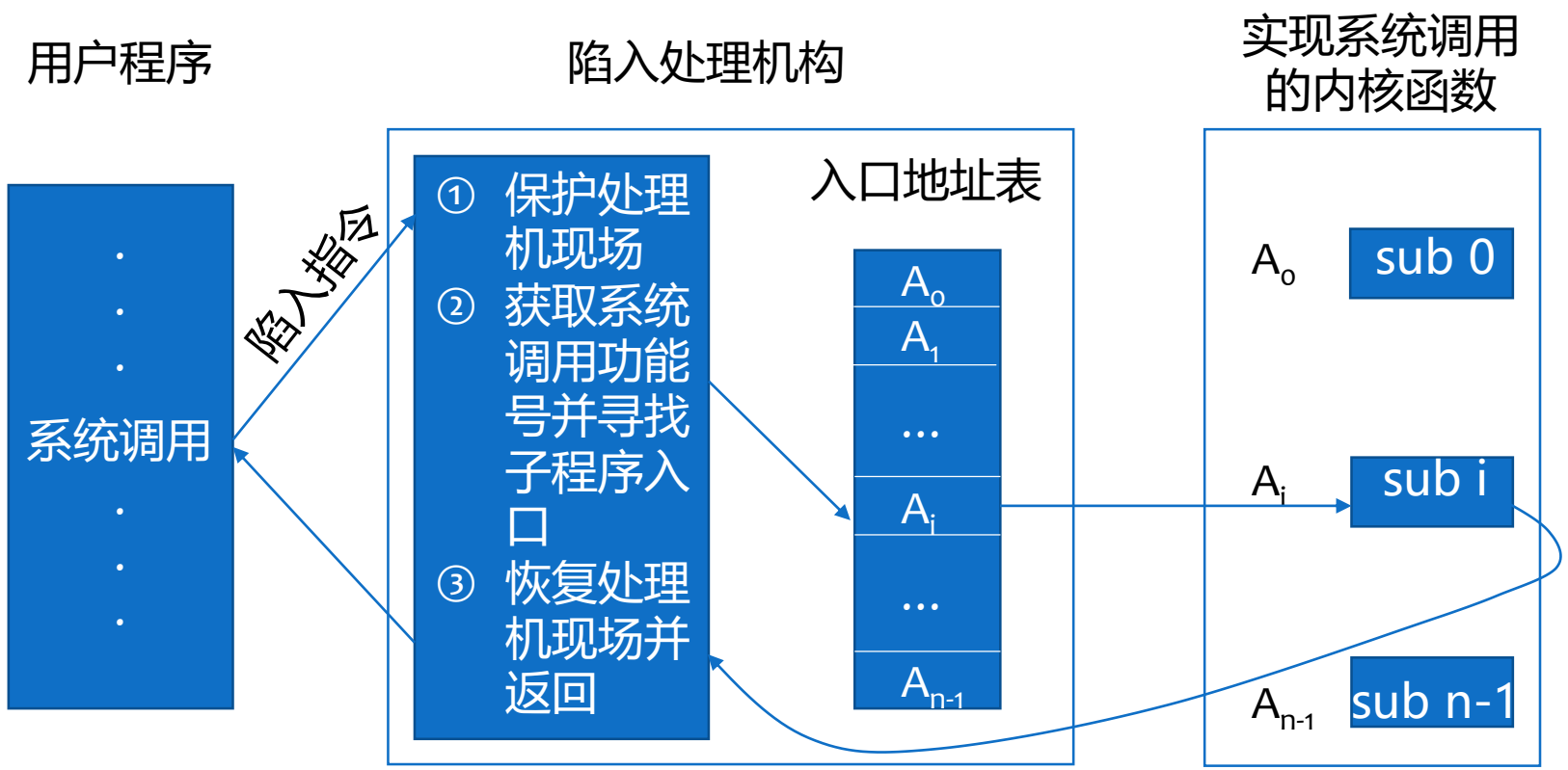
```
restore_all:
```

```
    RESTORE_ALL
```

# 系统调用的处理过程



# 系统调用的处理过程



# 系统调用的处理过程

---

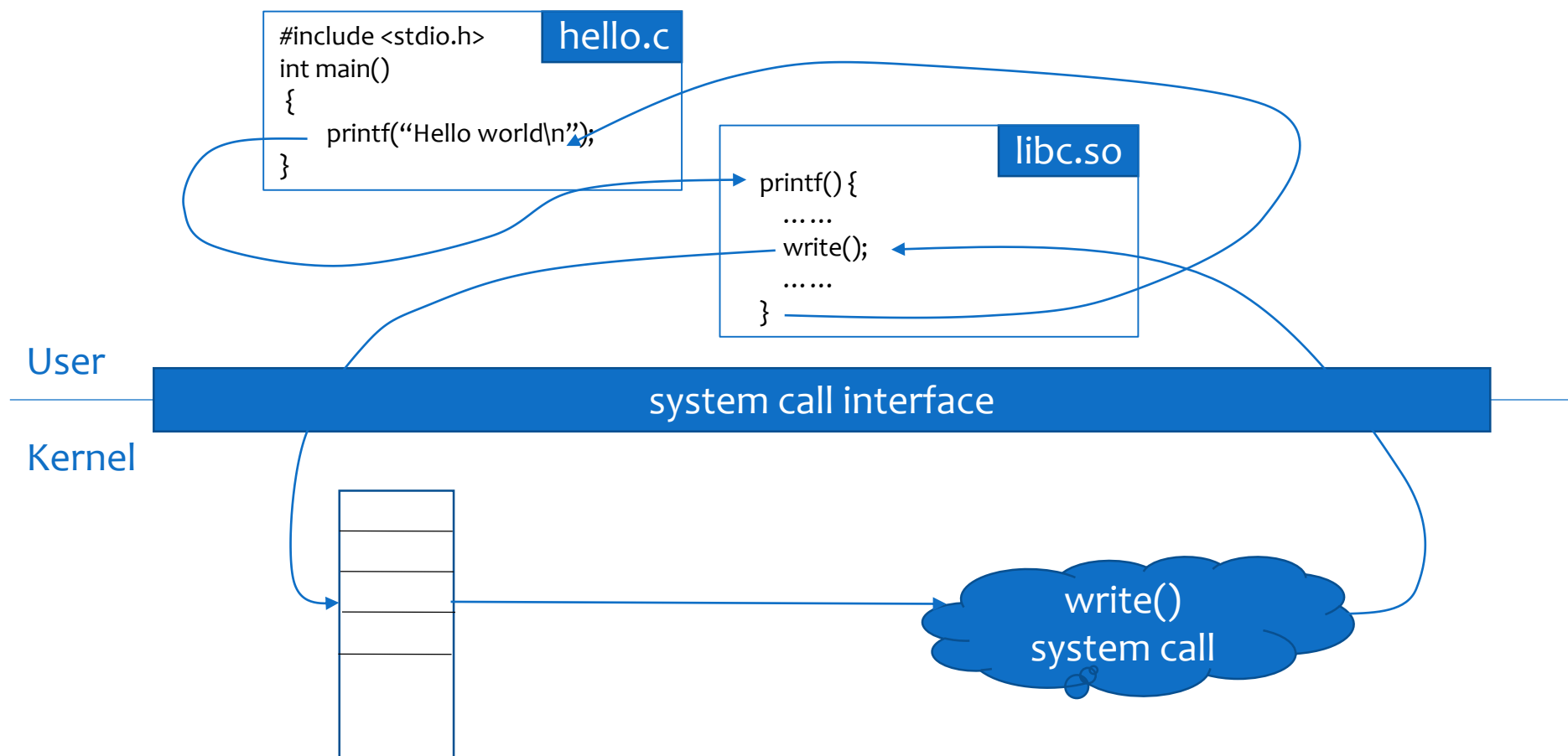
- 系统调用是用户程序进入内核的接口层，它本身并非内核函数，但是它由内核函数实现
- 进入内核后，不同的系统调用会找到各自对应的内核函数，这些内核函数也被称为系统调用的服务例程。例如，Linux的系统调用getpid实际调用的服务例程为sys\_getpid()

# 系统调用的封装

---

- 陷入指令是一条特殊指令，依赖操作系统实现的平台。例如在x86体系结构中，Linux利用int \$0x80作为陷入指令，不是用户在编程时应该使用的语句，因为这将使得用户程序难于移植
- 在标准C库函数中，为每个系统调用设置封装例程（库函数）。用户程序通过调用C函数库中相对应的封装例程执行系统调用

# 系统调用的处理过程





# 系统调用的封装

---

- Linux下用strace可以跟踪系统调用

```
hbma@192:~/oscource$ strace -e trace=write ./hello
write(1, "Hello world!\n", 13Hello world!
)          = 13
+++ exited with 0 +++
hbma@192:~/oscource$
```

# POSIX

---

- POSIX是Portable Operating System Interface(可移植操作系统接口)的缩写，缩写为 POSIX 是为了读音更像 UNIX
- POSIX由IEEE开发，由ANSI和ISO标准化
- POSIX的诞生和Unix的发展密不可分——由于各厂家对Unix的开发各自为政，造成了Unix的版本相当混乱，给软件的可移植性带来很大困难，对Unix的发展极为不利。为结束这种局面，IEEE开发了POSIX
- POSIX 并不局限于 UNIX。许多其它的操作系统，包括 Microsoft Windows，也支持 POSIX 标准

# POSIX

## ■ 进程管理

### Process management

Call	Description
<code>pid = fork( )</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &amp;statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

# POSIX

## ■ 文件管理

### File management

Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &amp;buf)</code>	Get a file's status information

# POSIX

## ■ 目录与文件系统管理

### Directory and file system management

Call	Description
<code>s = mkdir(name, mode)</code>	Create a new directory
<code>s = rmdir(name)</code>	Remove an empty directory
<code>s = link(name1, name2)</code>	Create a new entry, name2, pointing to name1
<code>s = unlink(name)</code>	Remove a directory entry
<code>s = mount(special, name, flag)</code>	Mount a file system
<code>s = umount(special)</code>	Unmount a file system

# POSIX

## ■其他

### Miscellaneous

Call	Description
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chmod(name, mode)</code>	Change a file's protection bits
<code>s = kill(pid, signal)</code>	Send a signal to a process
<code>seconds = time(&amp;seconds)</code>	Get the elapsed time since Jan. 1, 1970

# POSIX

---

- POSIX标准定义了构造系统所必须提供的一套过程，但是并没有规定它们是系统调用，还是普通的库函数调用或者其他形式
- POSIX过程调用映射到系统调用并不是一一对应的

# Windows API

---

- 也称为Win32 API
- 为方便用户编程，Windows以DLL的形式提供了API(Application Programming Interface, 应用程序编程接口)，用户可以通过调用API函数来使用Windows操作系统的系统调用
- 利用API间接调用系统调用的优点：
  - 方便
  - 增加应用程序的可移植性

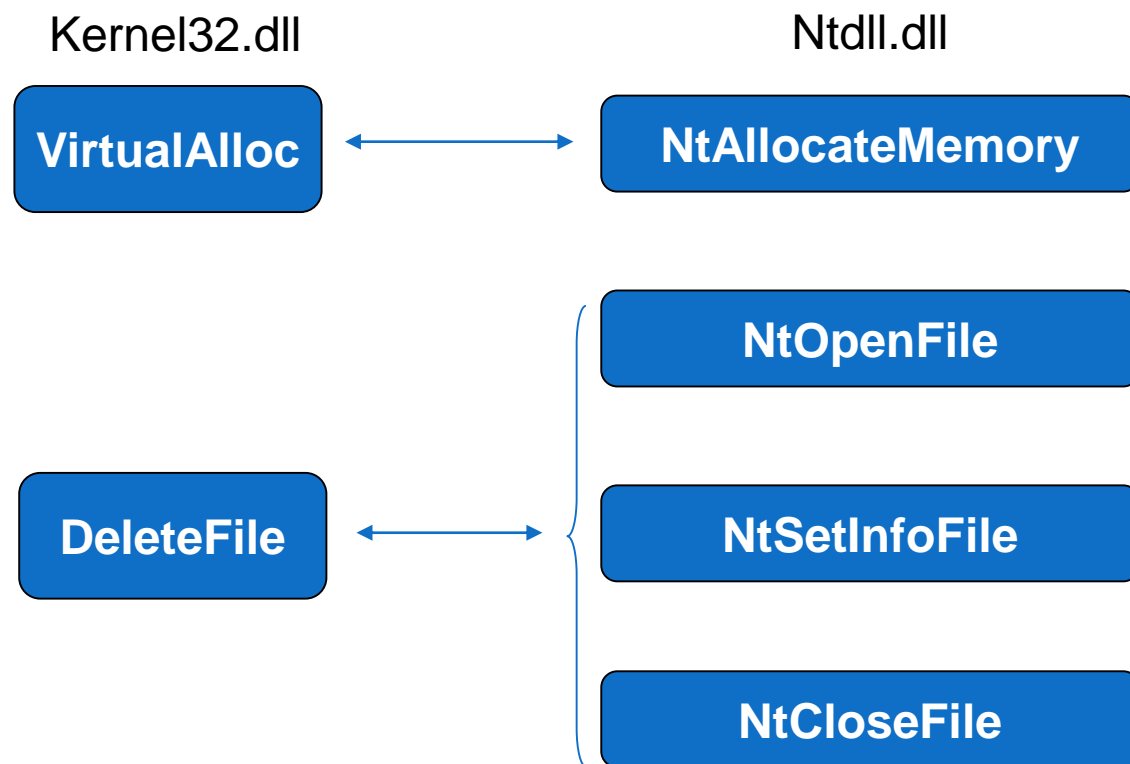


# Windows API

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

# Windows API

## ■ API调用与系统调用不存在一一对应的关系



# 大纲

01

操作系统的用户接口

02

操作系统的设计与实现

03

操作系统体系结构

# 操作系统的复杂性

## ■ IBM OS/360系统

- 由4000个模块组成
- 共有约100万条汇编语言指令
- 花费5000人年
- 经费达数亿美元
- 每个版本都仍然隐藏着无数的错误



# 操作系统结构设计

---

- 操作系统设计有着不同于一般应用系统设计的特征
  - 复杂程度高
  - 研制周期长
  - 正确性难以保证
- 解决途径
  - 良好的操作系统结构
  - 先进的开发方法和工程化的管理方法
  - 高效的开发工具

# 设计目标

---

- 系统设计的第一个问题是定义系统的目标和规格，系统设计受到硬件选择和系统类型的影响
- 不同的要求形成对不同环境的不同解决方案：
  - 批处理、分时
  - 单用户、多用户
  - 实时

# 设计目标

---

- 一般可以把需求分为两个基本类：
- 用户目标——容易使用、容易学习、可靠、安全、快速
- 系统目标——容易设计、实现和维护、灵活、可靠、没有错误且高效

# 操作系统的设计阶段

---

- **功能设计**：操作系统应具备哪些功能
- **算法设计**：选择和设计满足系统功能的算法和策略，并分析和估算其效能
- **结构设计**：选择合适的操作系统结构
  - 按照系统的功能和特性要求，选择合适的结构，使用相应的结构化设计方法将系统逐步分解、抽象和综合，使操作系统结构清晰、简单、可靠、易读、易修改，而且使用方便，适应性强



# 机制与策略的区分

---

- 机制(mechanism)——需要提供何种功能
- 策略(policy)——如何使用这些功能
- 例如：
  - 定时器是一种确保CPU保护的机制，对于特定用户将定时器设置成多长时间是个策略问题
  - 如何定义优先级则是机制问题，决定I/O密集型程序应该比CPU密集型程序有更高的优先级，还是相反则是策略问题

# 机制与策略的区分

---

- 策略和机制的区分对于灵活性来说很重要
- 策略可能会随时间和环境而有所改变，如果不区分策略和机制，在最坏的情况下，每次策略改变都可能需要底层机制的改变
- 因此，操作系统需要区分机制与策略，使策略的改变只需要重定义一些系统参数，而不导致底层机制的改变

# 系统实现

---

- 传统操作系统是用汇编语言来编写的，现代操作系统大都是都是用高级语言如C或C++来编写的
- C语言是开发操作系统的首选语言
- 高级语言编写操作系统的优点
  - 编写更快速
  - 更紧凑
  - 更易理解与调试
  - 更易快速移植到其它硬件平台上

# 系统实现

---

- 反对观点认为，采用高级语言实现的操作系统降低了速度、增加了存储要求
- 然而：
  - 现代编译器能对大型程序进行复杂的分析并采用高级优化技术以生成优化代码
  - 操作系统的重要性能改善主要是由于更好的数据结构和算法，而不是由于优秀的汇编语言代码
  - 在系统编写完成并能正常工作之后，可以找出瓶颈子程序，并用相应的汇编语言子程序代替

# 系统实现

---

- UNIX操作系统主要是用C来编写。早期的UNIX中，只有约900行代码是用汇编语言来编写的
- Windows、Linux、Mac OS等主流操作系统都是主要用C编写的

# 操作系统性能

---

- 可靠性
- 效率
- 可维护性

# 可靠性

---

- 计算机系统的用户都希望在一个稳定的、 安全可靠的、 不会产生错误的操作系统上工作
- 然而，现实中并不存在这种系统
  - 操作系统涉及大量的软件和硬件，至今还没有一种设计和实施技术能够保证它们永远不发生故障
  - 系统的使用环境复杂多变，病毒、 黑客攻击，系统操作员和一般用户的各种误操作等都会发生故障

# 可靠性策略

---

- 在设计和实现的各个环节采取各种技术，尽可能避免软件和硬件故障
- 在系统运行过程中，一旦出错要能及时检测出来，以减少对系统造成的损害
- 检测出错误后要能迅速找到造成错误的原因，确定故障位置并采取相应措施排除故障
- 尽可能对错误造成的损害进行修复，使系统恢复正常运行



# 效率

---

- 如今操作系统在一些关键业务处理过程中发挥着重要作用，效率是非常重要的指标
- 常用指标：
  - 吞吐量——单位时间内成功完成的作业（进程数）
  - 各种资源的使用效率——有效提供提供服务的时间占总工作时间的百分比
  - 分时系统的响应时间——用户发出请求到系统做出响应的的时间
  - 批处理的作业周转时间——作业进入系统到其完成并退出系统所经历的时间

# 可维护性

---

- 在系统投入运行后，经常需要进行维护，例如整理文件系统，更换硬件，进行故障排除等等
- 所有这些工作应该易于完成，为了做到这一点，一个好的操作系统应该提供较强的系统维护工具，而且要提供完整的帮助文档等

# 大纲

01

操作系统的用户接口

02

操作系统的设计与实现

03

操作系统体系结构

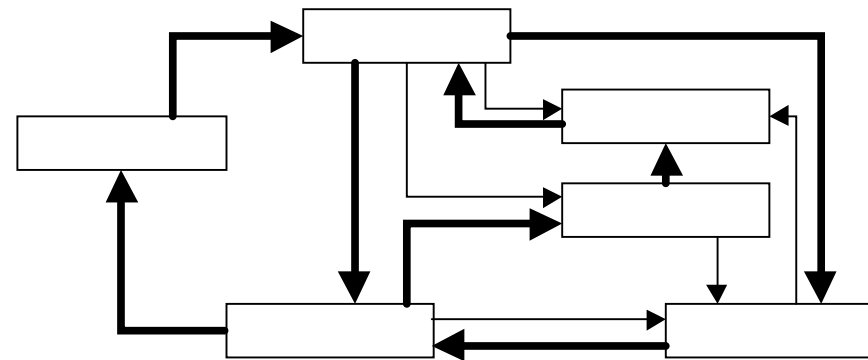
# 操作系统体系结构

---

- 操作系统是一种大型软件，为了研制操作系统，必须分析它的体系结构——也就是要弄清楚如何把这一大型软件划分成若干较小的模块以及这些模块间有着怎样的接口
- 常见的操作系统体系结构
  - 单体系统
  - 分层系统
  - 虚拟机结构
  - 微内核结构
  - 客户-服务器模式
  - 面向对象的操作系统

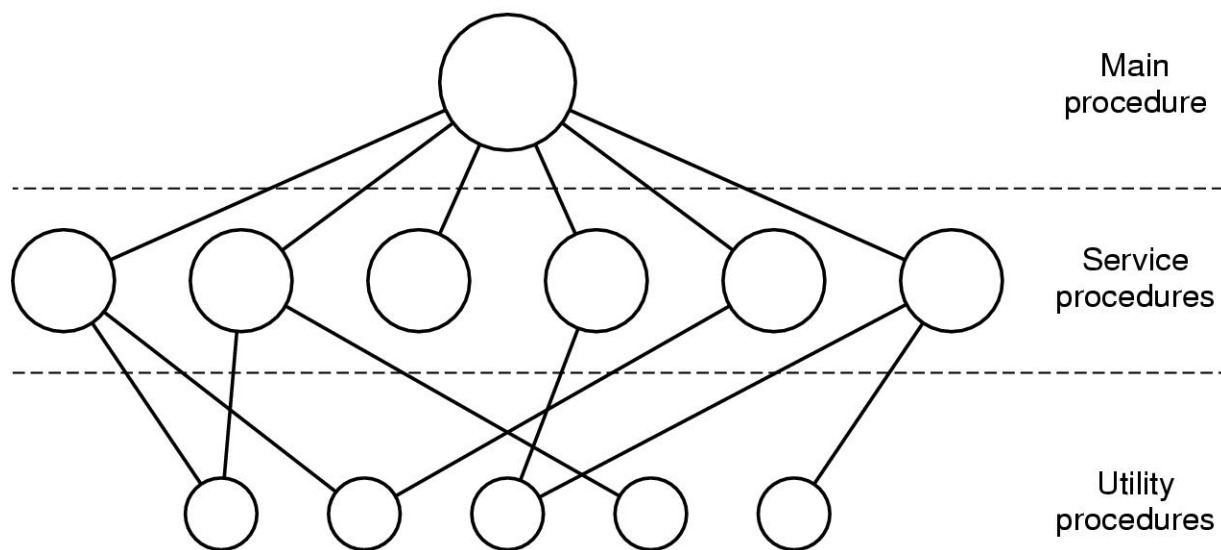
# 单体系统

- 也称为**单一内核结构**或**模块组合结构**：整个系统按功能进行设计和模块划分。系统是一个单一的、庞大的的软件系统，由众多服务过程（模块）组成，可以随意调用其他模块中的服务过程
- 优点
  - ◆ 具有一定灵活性，模块之间转接的灵活性使运行**高效率**；结构紧密，接口简单直接
- 缺点
  - ◆ 功能划分和模块接口难保正确和合理；模块之间的依赖关系（功能调用关系）**复杂**（调用深度和方向）



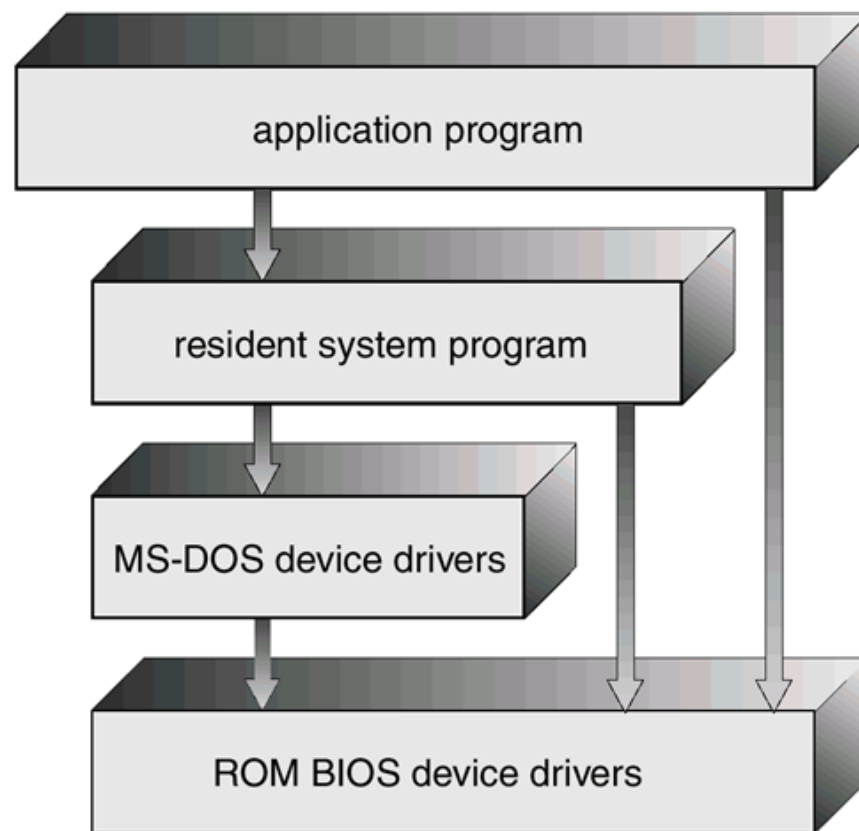
# 单体系统

## ■ 单体系统也应该有一定的结构



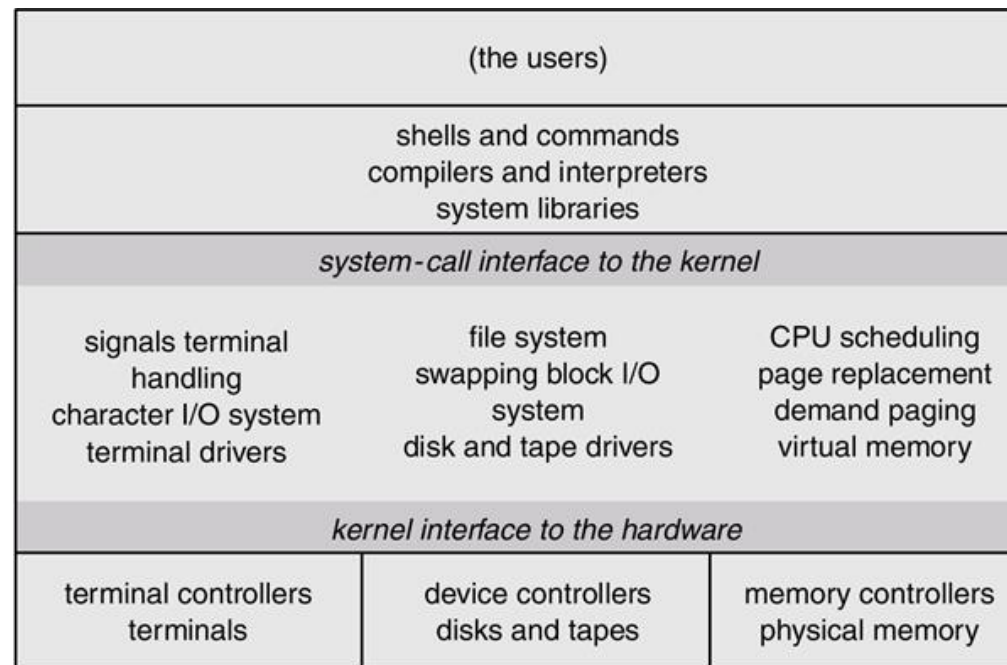
# MS-DOS

- 没有被划分成模块
- 尽管MS-DOS拥有一些结构，但它的接口和功能层次并没有很好的分离开来



# 早期Unix

- 最初受硬件功能的限制，由两个独立部分组成：系统程序、内核
- 内核
  - 包括物理硬件之上和系统调用接口之下的所有部分
  - 提供文件系统、CPU调度、内存管理和其它操作系统功能
  - 大多数的功能都结合放在这一层





# 分层系统

---

- 把操作系统的所有功能模块按功能的调用次序，分别排列成若干层，各层之间的模块只能是单向依赖或单向调用的关系
- 每层建立在较低层之上，每层只能利用较低层的功能和服务

# THE系统

- 按分层模型构造的第一个操作系统是E.W.Dijkstra和他的学生开发的THE系统（1968年）
- THE是荷兰语Technische Hogeschool Eindhoven缩写

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

# THE系统

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- 第0层：负责处理机的分配，当发生中断或定时器到时进行进程切换，从而提供了基本的多道程序环境
- 第1层：执行内存和磁鼓的管理，用来为进程分配内存空间和磁鼓上的空间。在内存用完时则在磁鼓上分配空间用作交换。在本层，进程不考虑它是在磁鼓上还是在内存中运行，能保证一旦某一页面需要访问时，它必定在内存中
- 第2层：处理每个进程和操作员控制台之间的通信。在本层，每个进程都有自己的操作员控制台

# THE系统

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- 第3层：进行输入输出管理，管理I/O设备，对信息流进行缓冲。在本层，每个进程和I/O设备打交道时无需考虑物理细节
- 第4层：用户程序层，用户程序不必考虑进程、内存、控制台或I/O设备等细节
- 第5层：系统操作员

# 分层原则

---

- 被调用功能在低层：如文件系统管理——设备管理——设备驱动程序
- 活跃功能在低层：提高运行效率
- 资源管理的公用模块放在低层：如缓冲区队列、堆栈操作
- 最低层的硬件抽象层：与机器特点紧密相关的软件放在最低层。如 Windows NT 中的 HAL——单处理、多处理
- 资源分配策略放在高层，便于修改或适应不同环境

# 分层系统

---

## ■优点：

- 功能明确，调用关系清晰（高层对低层单向依赖），有利于保证设计和实现的正确性
- 低层和高层可分别实现（便于扩充），高层错误不会影响到低层
- 便于修改、扩充，很容易增加或替换掉一层而不影响其它层次

## ■缺点：

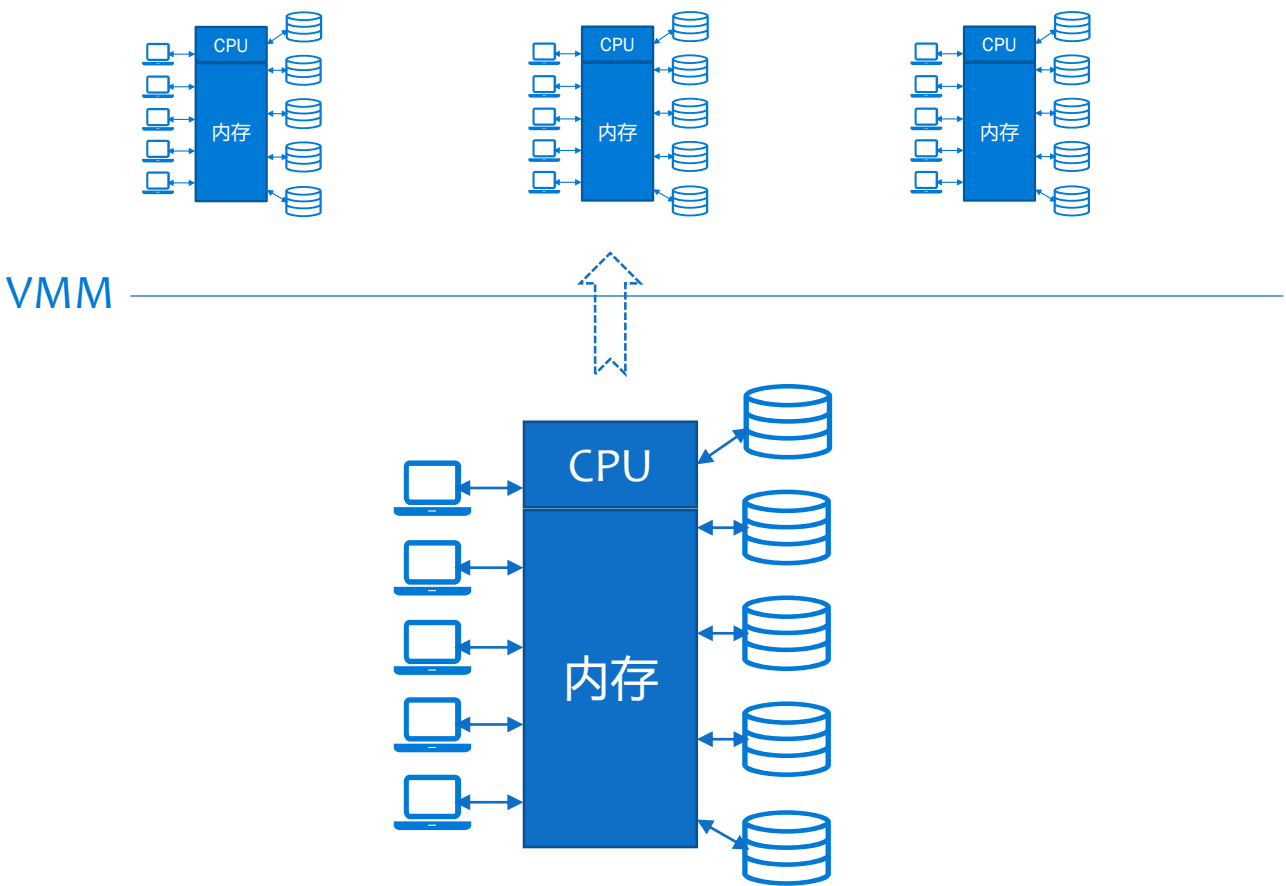
- 效率低——每层为系统调用增加了额外开销（参数或数据的修改与传递）

# 虚拟机结构

---

- 通过某种技术，使物理计算机作为共享资源从而创建虚拟机。虚拟机提供了与基本硬件相同的接口
- 通过利用CPU调度和虚拟内存技术，操作系统能创建一种幻觉，以至于进程认为有自己的处理器和自己的内存
- 每台虚拟机都与裸机相同，所以每台虚拟机可以运行一台裸机所能够运行的任何类型的操作系统。不同的虚拟机可以运行不同的操作系统

# 虚拟机结构





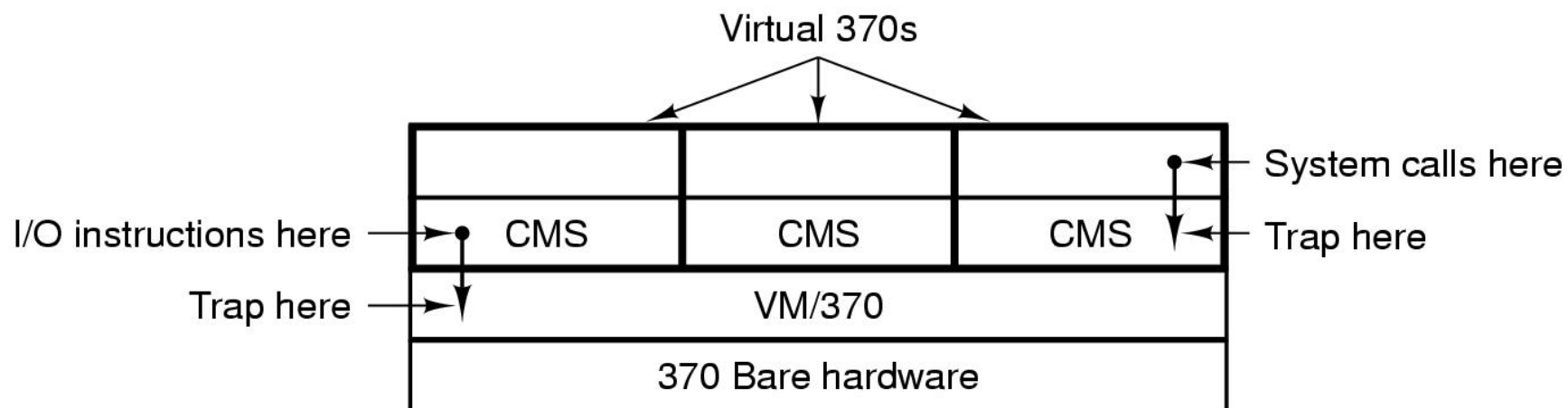
# IBM VM/370

---

- VM/370核心被称为**虚拟机监控器 (Virtual Machine Monitor)**，它在裸机上运行并且具备了多道程序设计功能。该系统向上层提供了若干台虚拟机
- 它不同于其他操作系统，这些虚拟机不是那种具有文件等优良特征的扩展计算机。与之相反，他们仅仅是裸机硬件的精确复制品，这个复制品包含了核心态/用户态、I/O、中断及其他真实机器所具有的全部内容

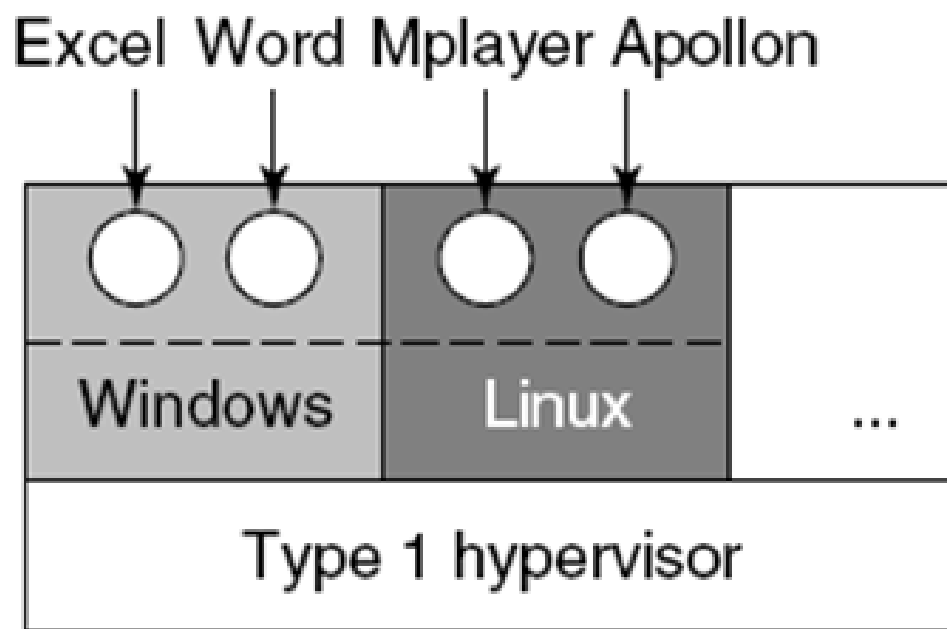
# IBM VM/370

- 由于每台虚拟机都与裸机相同，所以每台虚拟机可以运行一台裸机所能够运行的任何类型操作系统
- 程序在执行系统调用时，它的系统调用陷入其虚拟机中的操作系统。然后会话监控系统(CMS)发出硬件I/O指令，在虚拟机中执行为该系统调用所需的其它操作。这些I/O指令被VM/370捕获，作为对真实硬件模拟的一部分，VM/370随后就执行这些指令



# 虚拟机结构

- VM/370的虚拟机监控程序也称为1型超级监控程序(type 1 hypervisor)



# 虚拟机结构

---

- 早期的X86处理器，在用户模式下执行特权指令只是忽略，无法像VM/370那样由客户OS陷入主机OS
- Stanford的Disco (1997) 引入二进制翻译技术，解决了这一问题，并在商业化产品例如Vmware、Xen得到应用，由此引入2型超级监控程序(type 2 hypervisor)

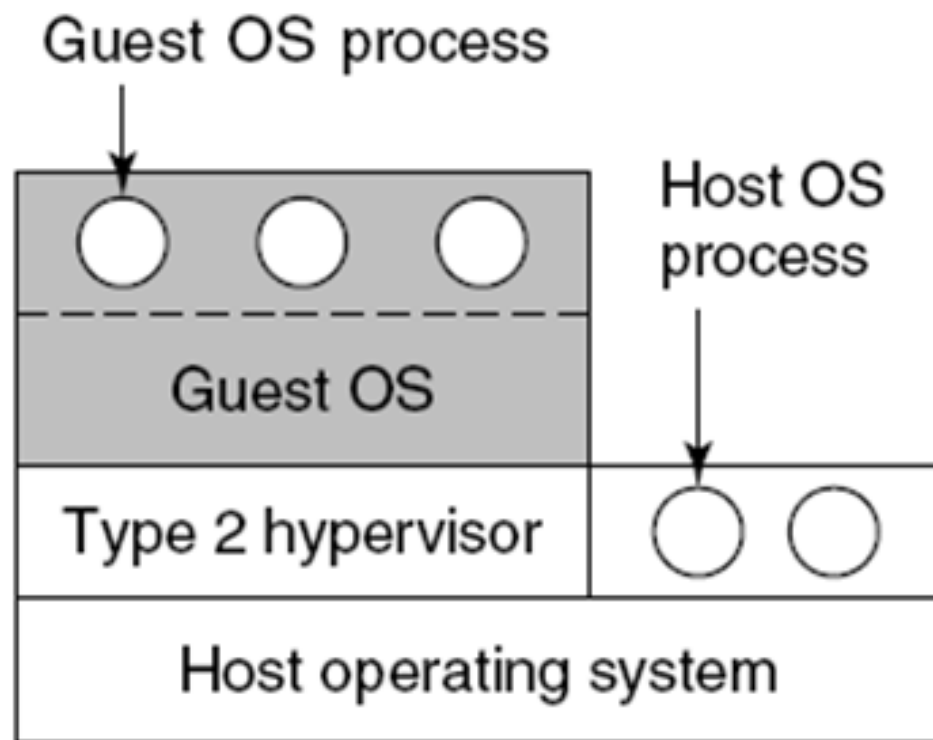
# 虚拟机结构

---

- 2型超级监控程序(type 2 hypervisor)作为一个应用程序运行在主机操作系统上
- 2型超级监控程序启动后，可以在虚拟磁盘上安装客户操作系统。虚拟磁盘实际上是主机操作系统中的大文件
- 客户操作系统在运行时，2型超级监控程序一块块地翻译客户操作系统的二进制程序，将特定的控制指令替换成超级监控程序调用

# 虚拟机结构

## ■ 2型超级监控程序的典型代表——VMWare



# 虚拟机结构的优点

---

- 每个虚拟机完全与其他虚拟机相隔离，由于各种系统资源完全被保护，所以不存在安全问题
- 没有直接资源共享
- 虚拟机系统是用于研究和开发操作系统的良好工具，虚拟机允许进行系统开发而不必中断正常的系统操作。利用虚拟机，系统开发可在虚拟机而不是真实的物理机器上进行

# 微内核结构

---

- 微内核结构的思想是，为了实现高可靠性，从操作系统中去掉尽可能多的东西，而只留一个最小的核心运行在核心态下，其他模块作为普通用户进行运行在用户态下
- **微内核**：运行在核心态的内核提供最基本的操作系统功能，包括中断处理、处理机调度、进程间通信。这些部分只提供了一个很小的功能集合，通常称为微内核
- **服务进程**：其它的OS服务都是由运行在用户模式下的进程完成，可作为独立的应用进程，称为服务进程。这些模块中的错误不会使整个操作系统崩溃



# 微内核结构

---

## ■ 优点

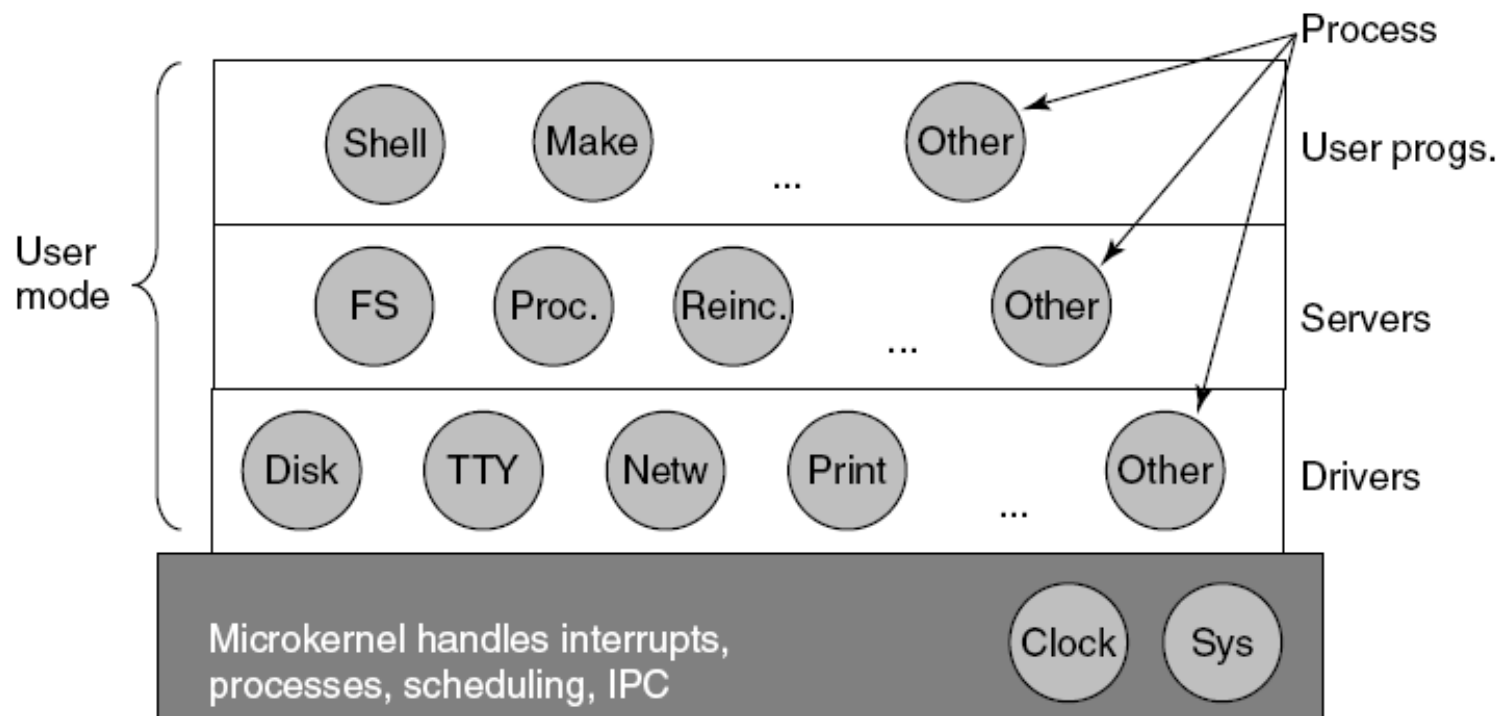
- 体现机制与策略分离
- 良好的扩充性：只需添加支持新功能的服务进程即可。而且所有新服务被增加到用户空间中，不需要修改内核
- 可靠性好：所有服务器以用户进程的形式运行，而不是运行在核心态，所以它们不直接访问硬件。假如在文件服务器中发生错误，文件服务器可能崩溃，但不会导致整个系统的崩溃

## ■ 缺点

- 消息传递比直接调用效率要低一些

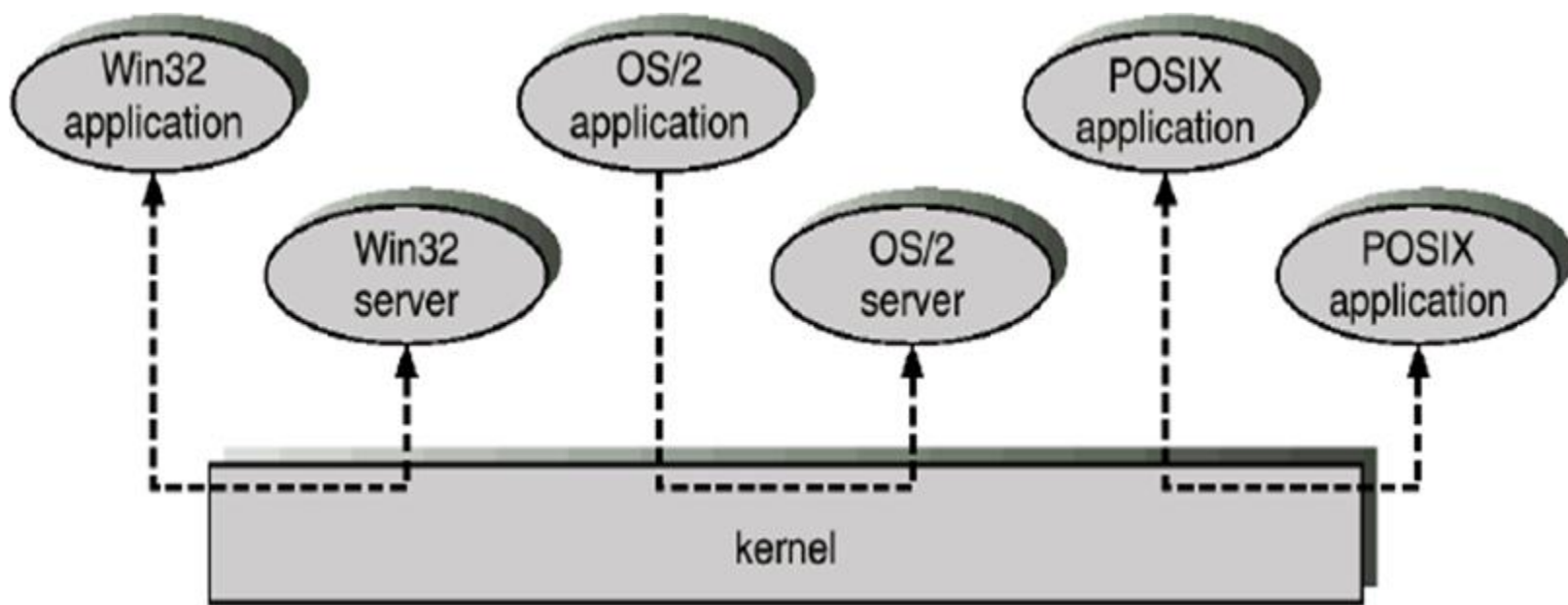
# 微内核结构

## ■ MINIX 3系统的结构



# 微内核结构

## ■早期Windows NT



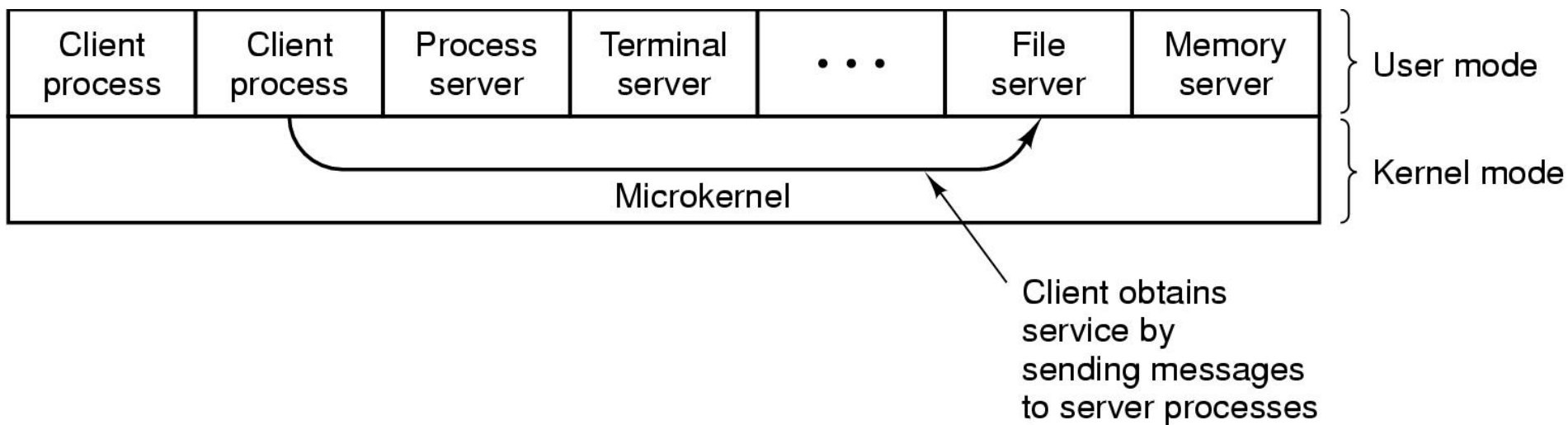
# 客户-服务器模型

---

- 微内核结构的变体，将进程分为两类：
  - 服务器：提供某种服务
  - 客户：使用这些服务
- 系统最底层是微内核
- 客户进程与服务器进程之间使用消息进行通信

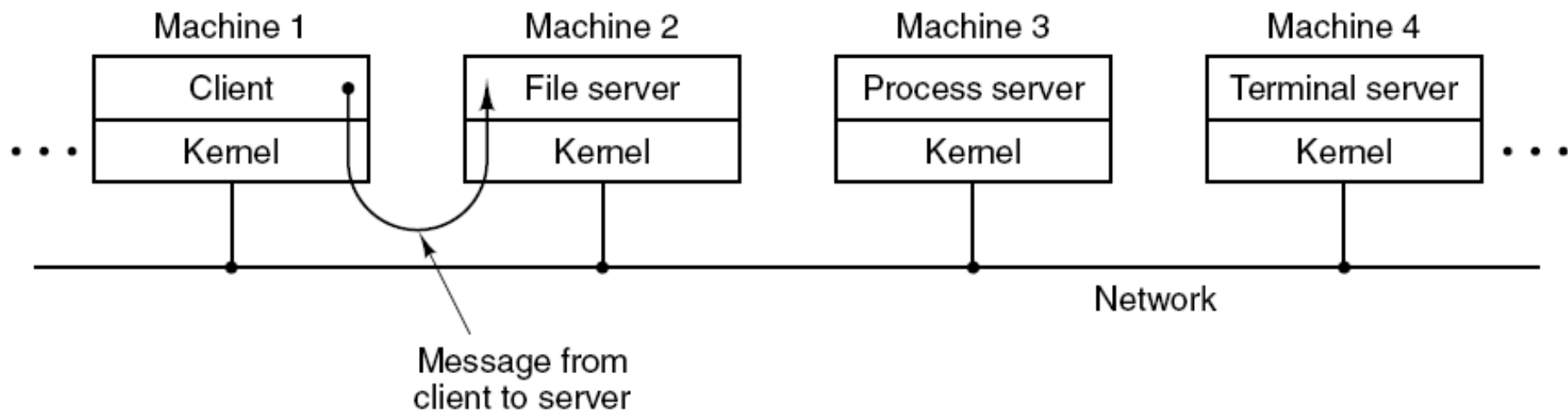
# 客户-服务器模型

## ■ 服务器和客户进程运行在相同的机器上



# 客户-服务器模型

- 服务器和客户进程运行在不同的机器上



# 面向对象的操作系统

---

- **面向对象**概念起源于20世纪60年代末期。近年来，随着计算机软、硬件技术的发展，面向对象技术在数据库、程序设计语言、以及操作系统和计算机网络通信等几乎所有的软件领域都受到了极大重视和广泛研究
- 在操作系统领域中，由于面向对象技术除了在设计方法上更接近于设计人员脑子中的“思维形象”之外，它还具有隐蔽数据以及由消息激活对象等特性，从而它比传统技术更容易应用于分布式操作系统的设计与实现

# 面向对象的操作系统

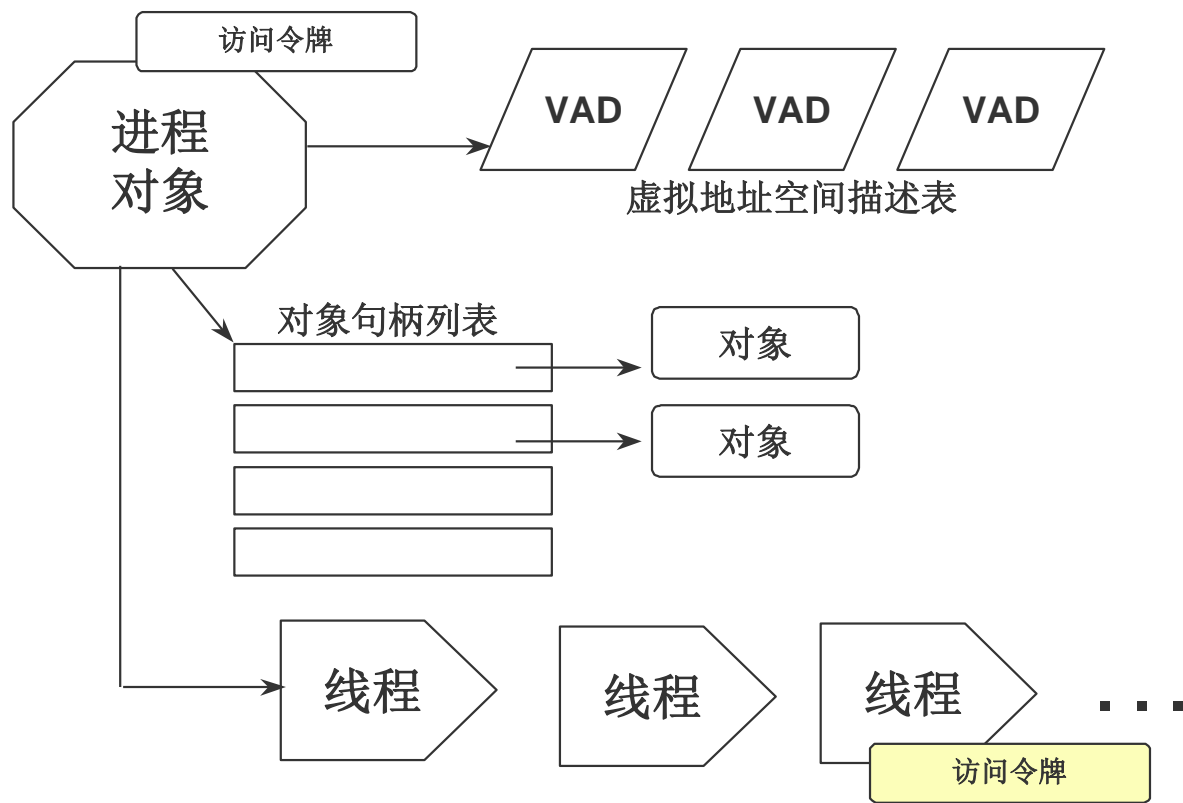
---

- 面向对象的操作系统得到广泛重视和研究的另一个重要因素是它适合于超大规模的、开放式分布环境
- 由于面向对象技术采用对象间发送消息来驱动对象完成特定功能方法，并且对象的定义不受距离和系统的限制，因此，面向对象的概念被广泛地用于分布式操作系统或网络操作系统



# 面向对象的操作系统

- 例如，Windows的核心态组件使用了面向对象的设计原则。出于可移植性以及效率因素的考虑，大部分代码使用了基于C语言的对象实现

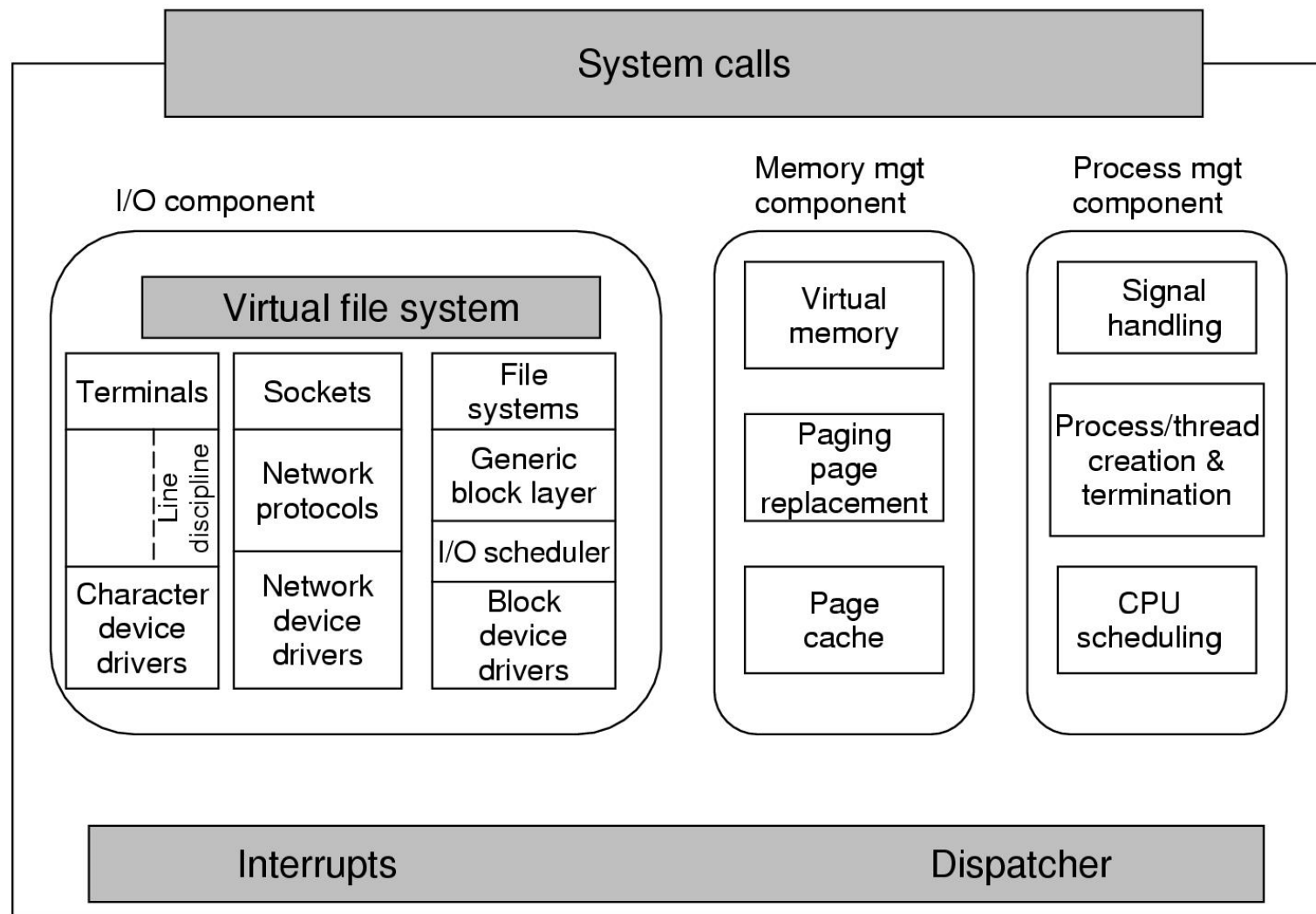


# 面向对象的操作系统

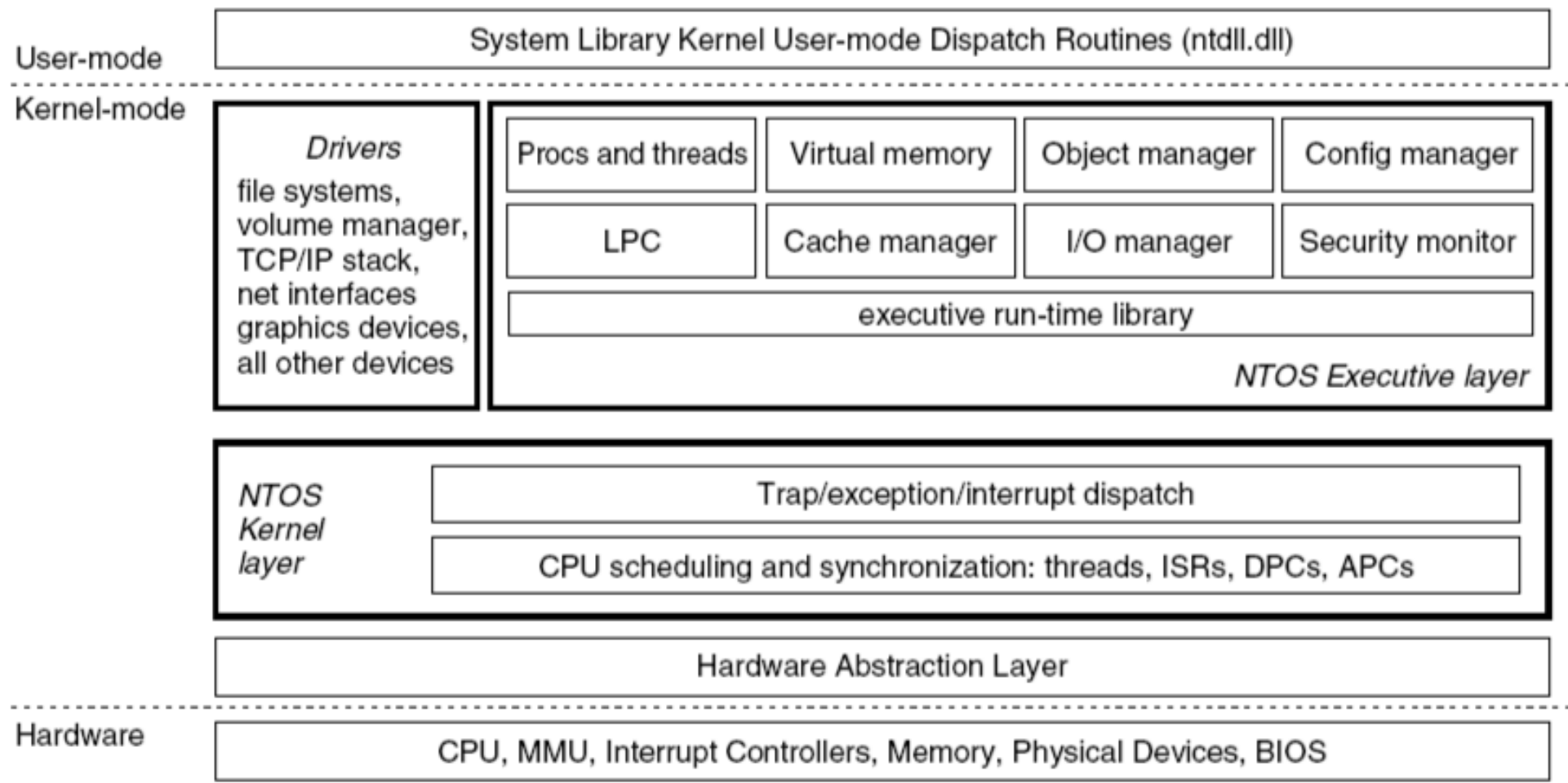
---

- 对象的封装实现了接口和实现的分离，即机制和策略的分离
- 派生机制有利于基于类架构的代码复用
- 便于以更细的粒度配置系统，可以使系统更灵活，更容易扩充
- 易于保证系统的可靠性和正确性

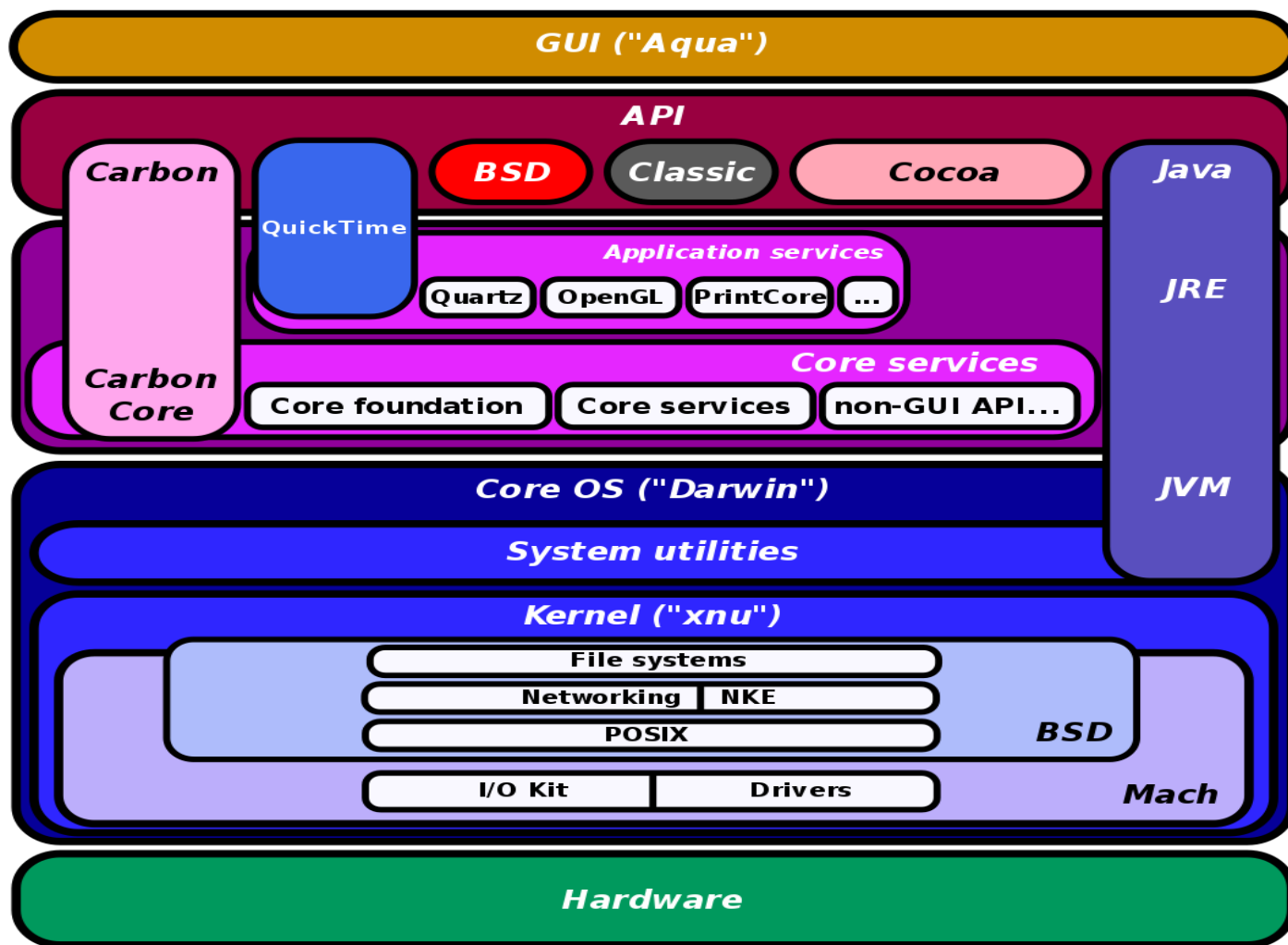
# Linux体系结构



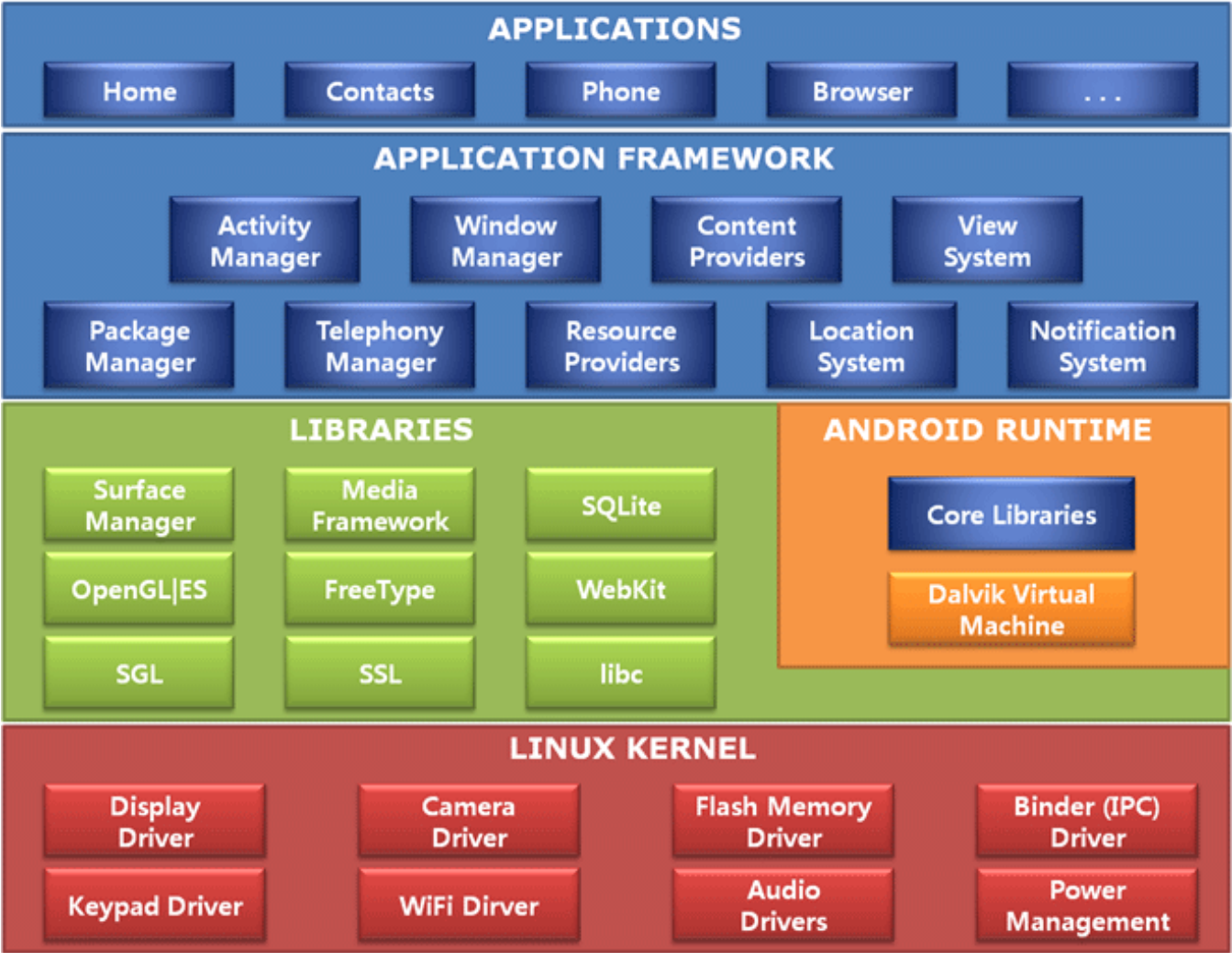
# Windows体系结构



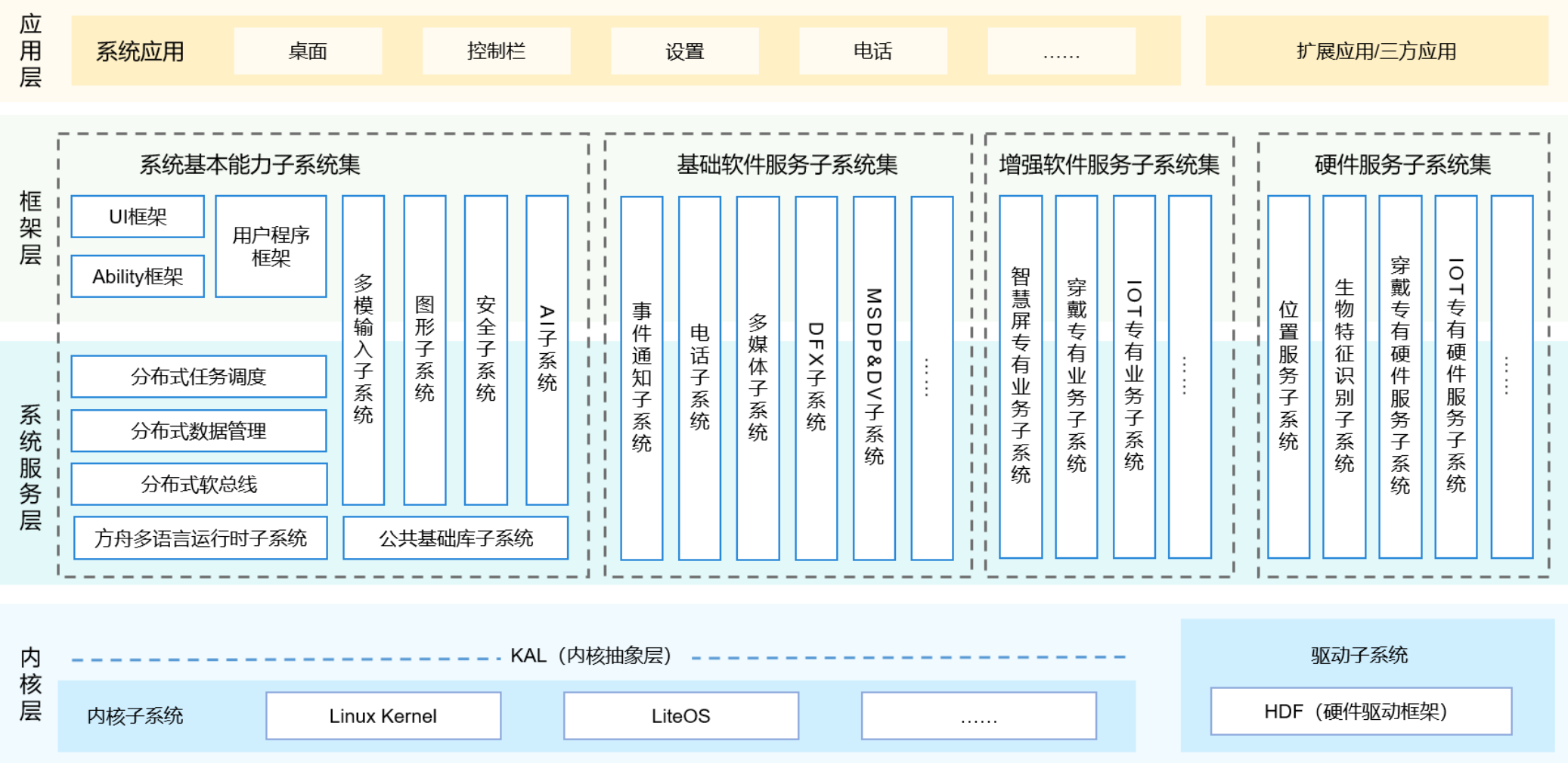
# Mac OS体系结构



# Android体系结构



# HarmonyOS体系结构





谢谢