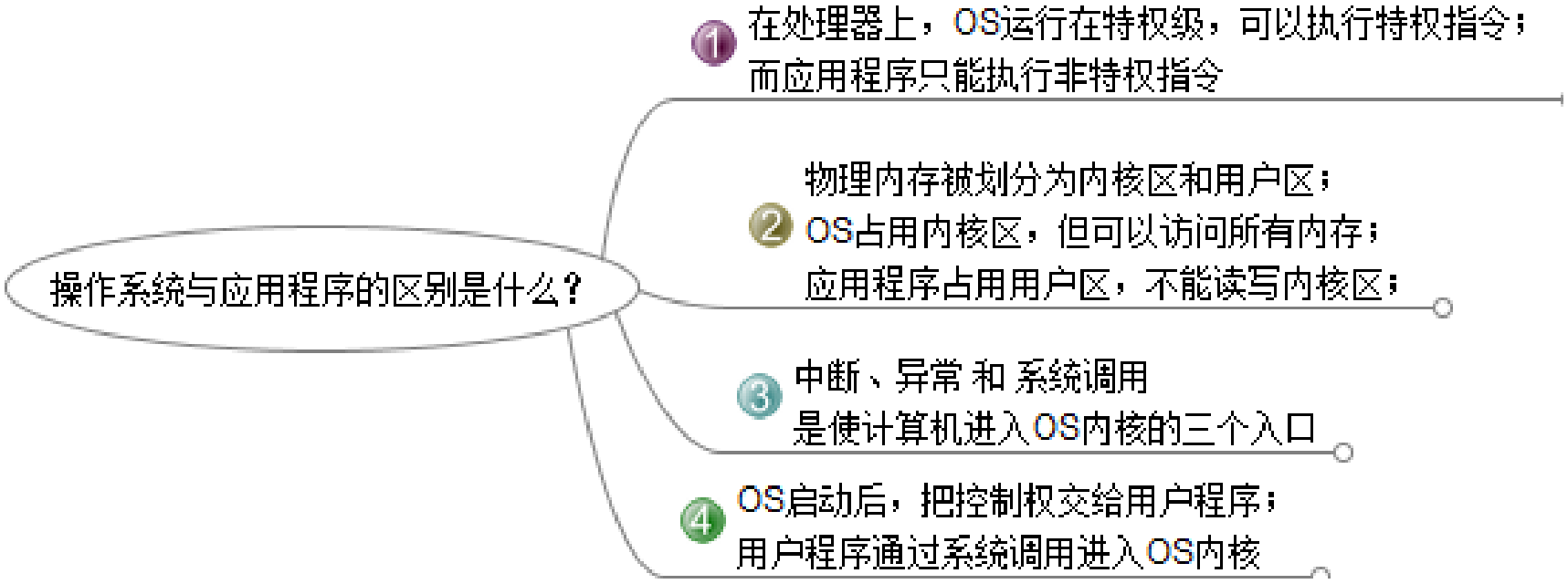
操作系统的运行环境

主讲教师：赵霞

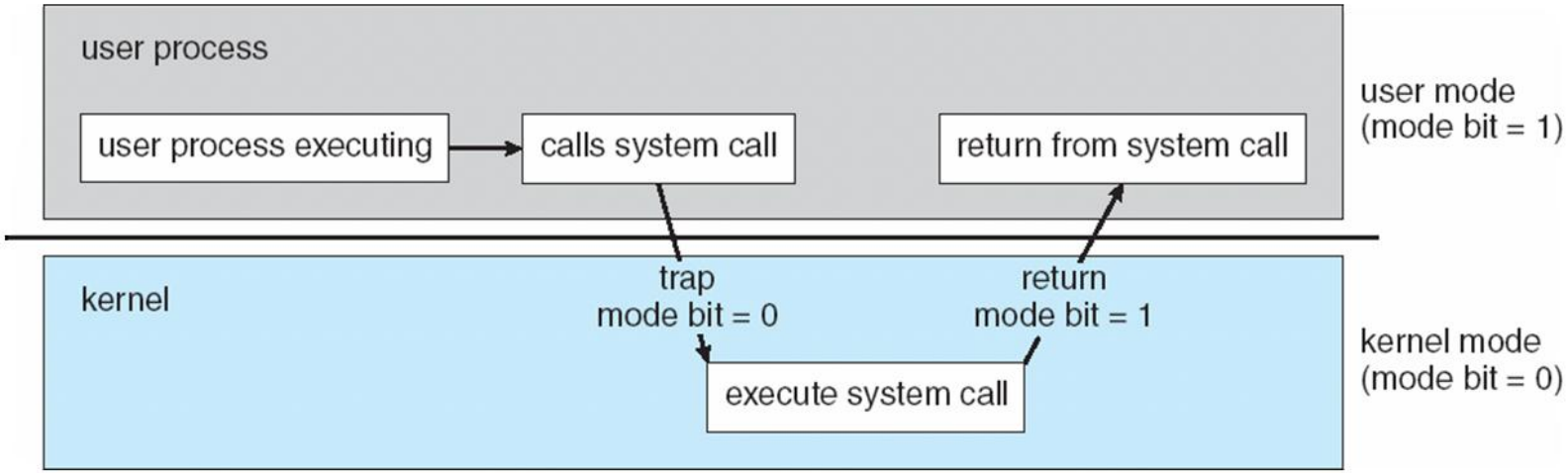
操作系统和应用程序的区别是什么？

- 操作系统作为计算机系统的管理者，具有特殊的地位
  - 直接控制硬件：处理器、内存、外围设备
- 从OS的运行环境
  - 理解关键区别
  - 理解OS内核机制



操作系统和应用程序的关系

- 用户进程 对应 应用程序，在用户态下执行
- 用户进程 通过 系统调用接口 由用户态 进入OS内核态
- 在内核态，OS的内核模块代码访问硬件设备，完成用户程序的请求



操作系统的运行环境

- 计算机系统的层次结构
- 中央处理器(CPU)
- 存储系统
- 中断与异常
- 操作系统运行模型
- 系统调用
- 人机界面

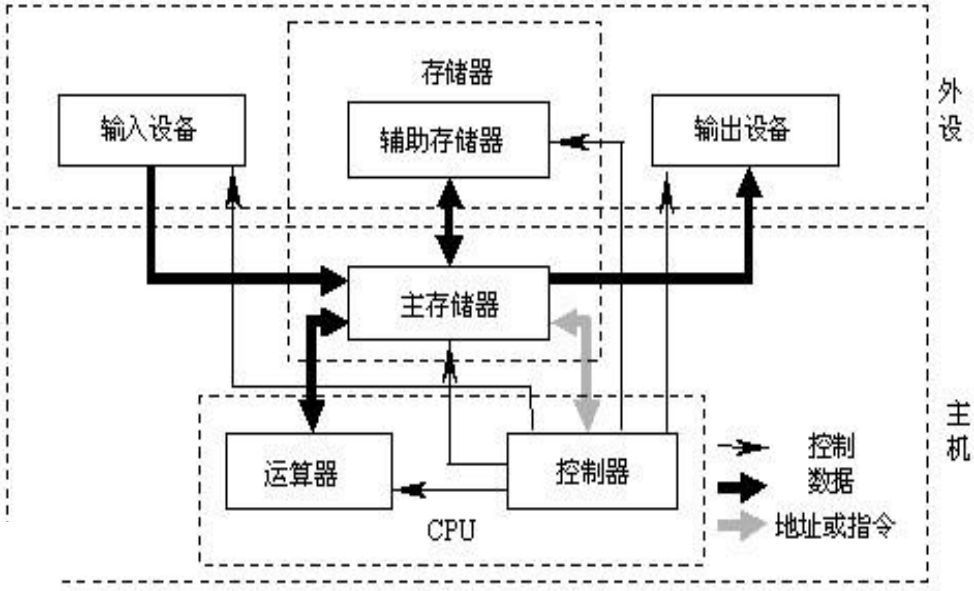
计算机的模型

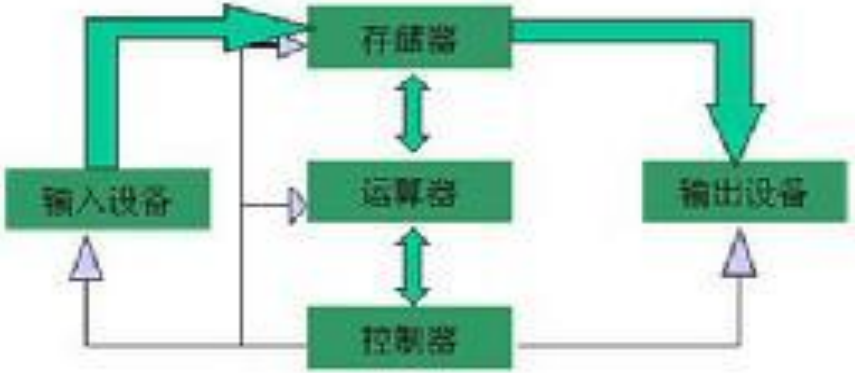
冯.诺依曼

（John von Neumann

1903-1957）

出生匈牙利的美国籍犹太人数学家

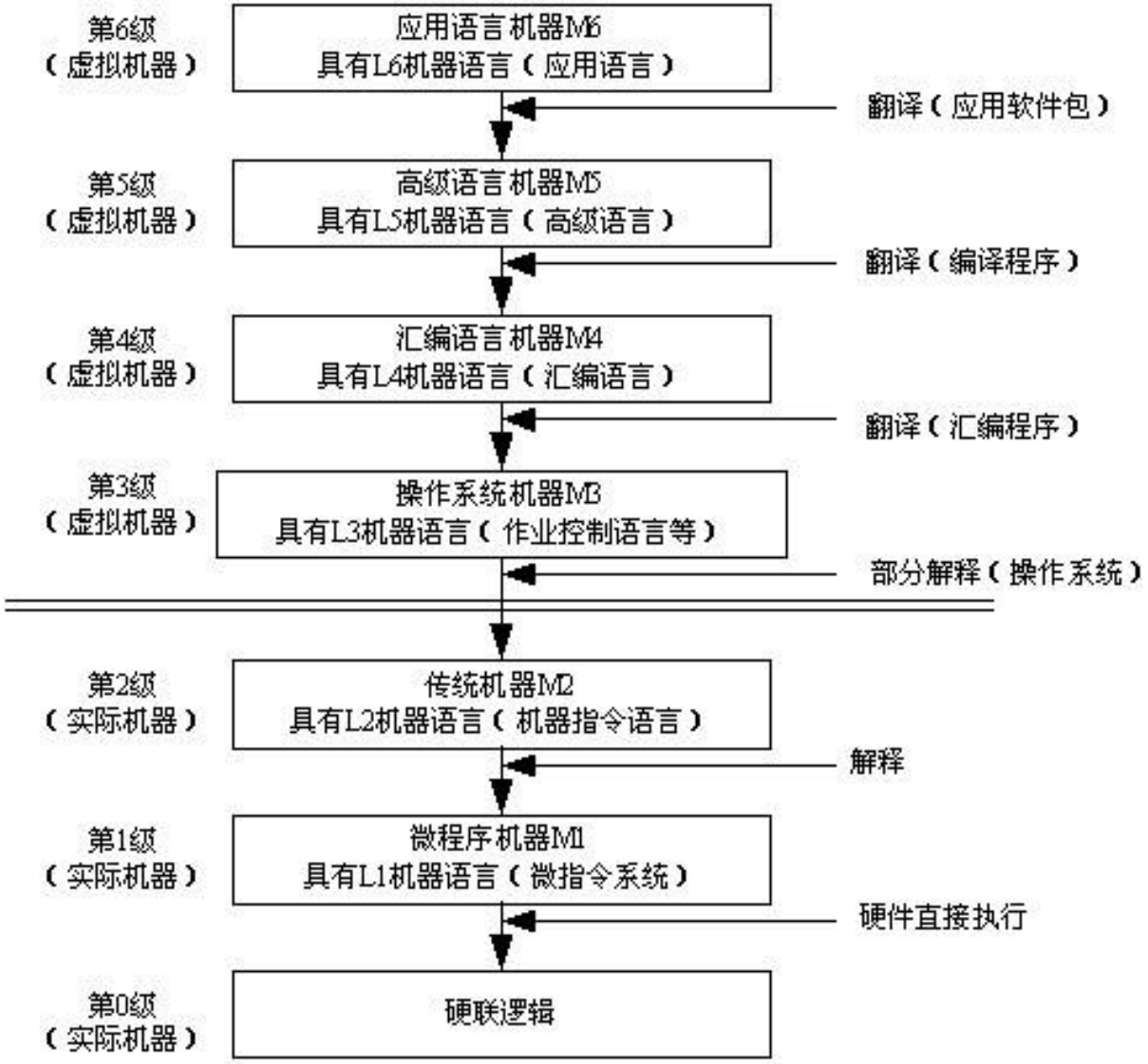




电子计算机基本结构图

功能模块　数据信号线　控制信号线

计算机系统的层次结构

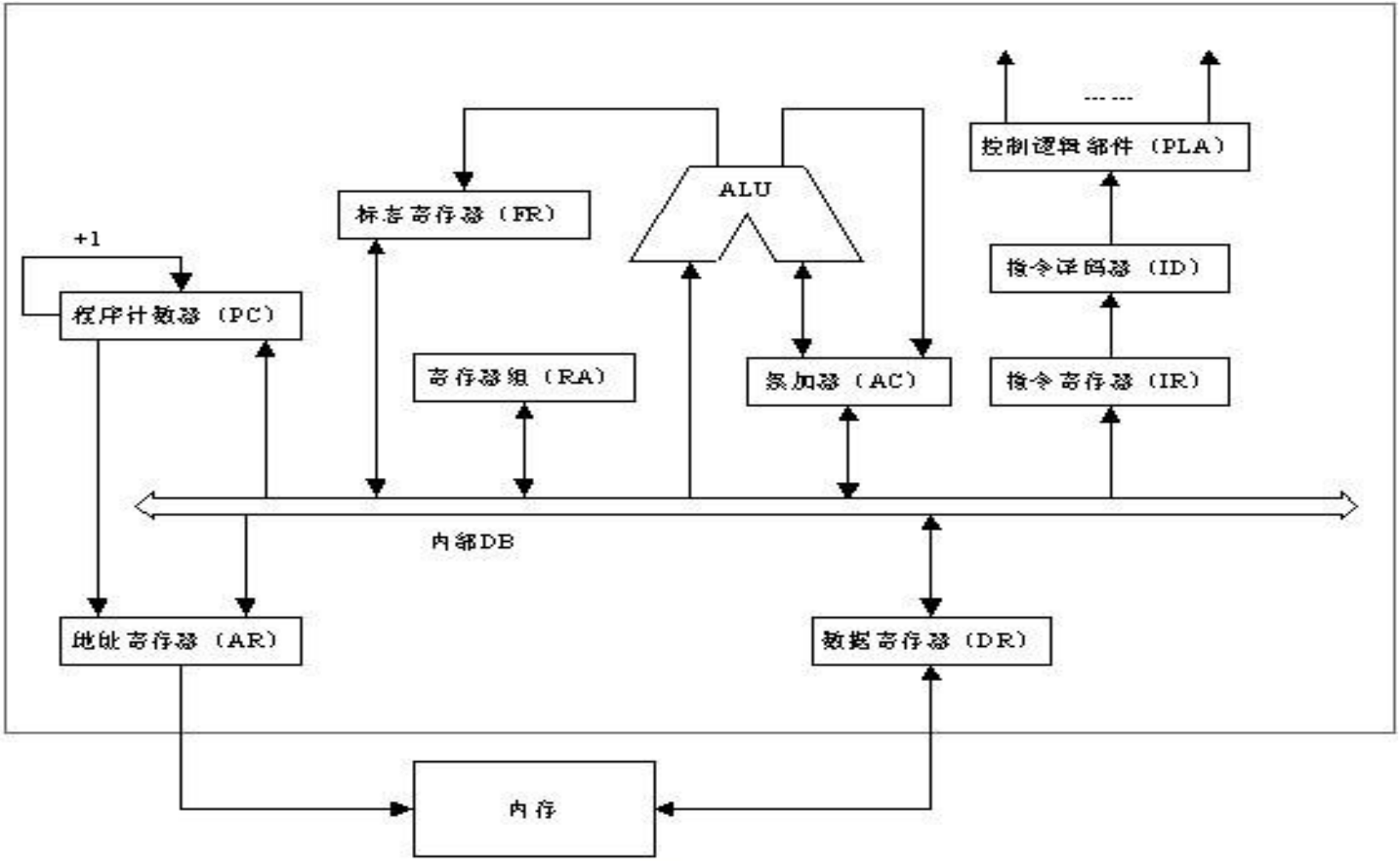

第6级
（虚拟机器）
应用语言机器M6
具有L6机器语言（应用语言）
翻译（应用软件包）
第5级
（虚拟机器）
高级语言机器M5
具有L5机器语言（高级语言）
翻译（编译程序）
第4级
（虚拟机器）
汇编语言机器M4
具有L4机器语言（汇编语言）
翻译（汇编程序）
第3级
（虚拟机器）
操作系统机器M3
具有L3机器语言（作业控制语言等）
部分解释（操作系统）
第2级
（实际机器）
传统机器M2
具有L2机器语言（机器指令语言）
解释
第1级
（实际机器）
微程序机器M1
具有L1机器语言（微指令系统）
硬件直接执行
第0级
（实际机器）
硬联逻辑

操作系统运行环境

- 计算机系统的层次结构
- 中央处理器(CPU)
  - CPU的构成与基本工作方式
  - 特权指令和非特权指令
  - 处理器状态
  - 程序状态字PSW
- 存储系统
- 中断和异常
- 操作系统运行模型
- 系统调用
- 人机界面

中央处理器CPU-cpu构成

- 运算器、控制器、一系列的寄存器

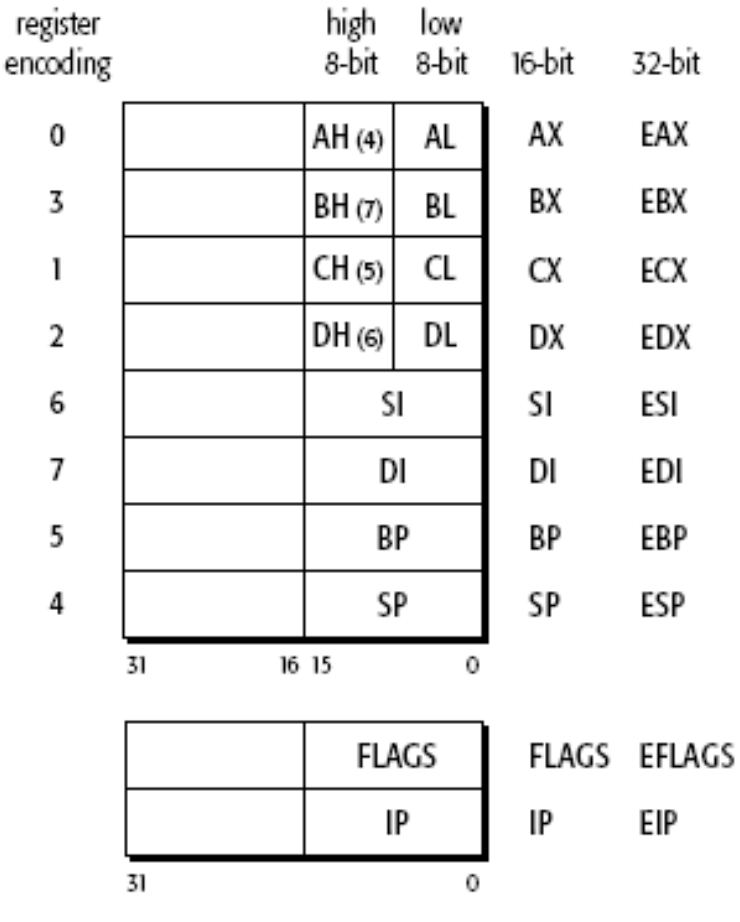


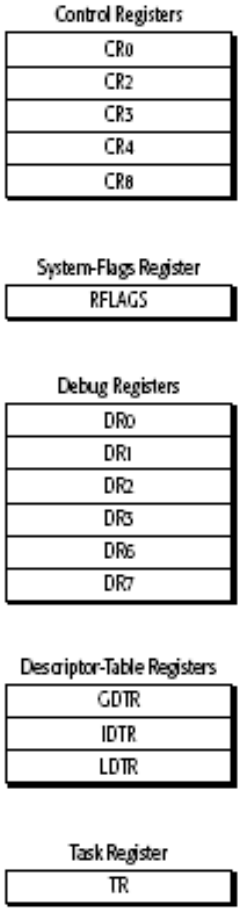
中央处理器CPU -cpu构成

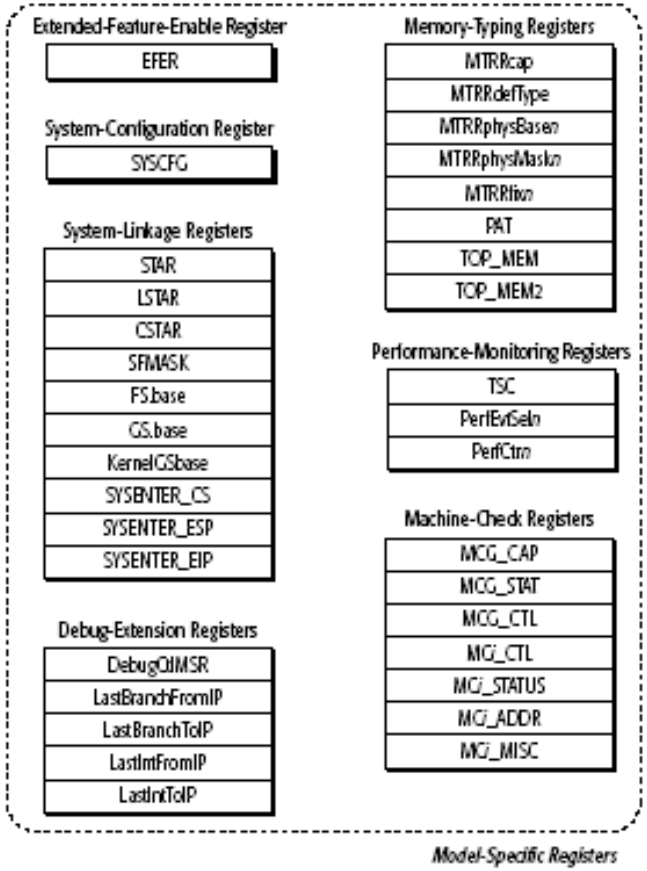
寄存器的种类

- 用户可见寄存器：由编译器进行寄存器分配
  - 数据寄存器：保存数据
  - 地址寄存器：存储数据及指令的物理地址、线性地址或有效地址
  - 条件码寄存器：保存CPU操作结果的各种标记位（在条件分支指令中被测试）
- 控制和状态寄存器：保存处理器的状态
  - 程序计数器：下一条指令地址
  - 指令计数器：最近取出的指令
  - 程序状态字：处理器的运行模式信息
  - 内存访问相关的寄存器

Intel X86的寄存器







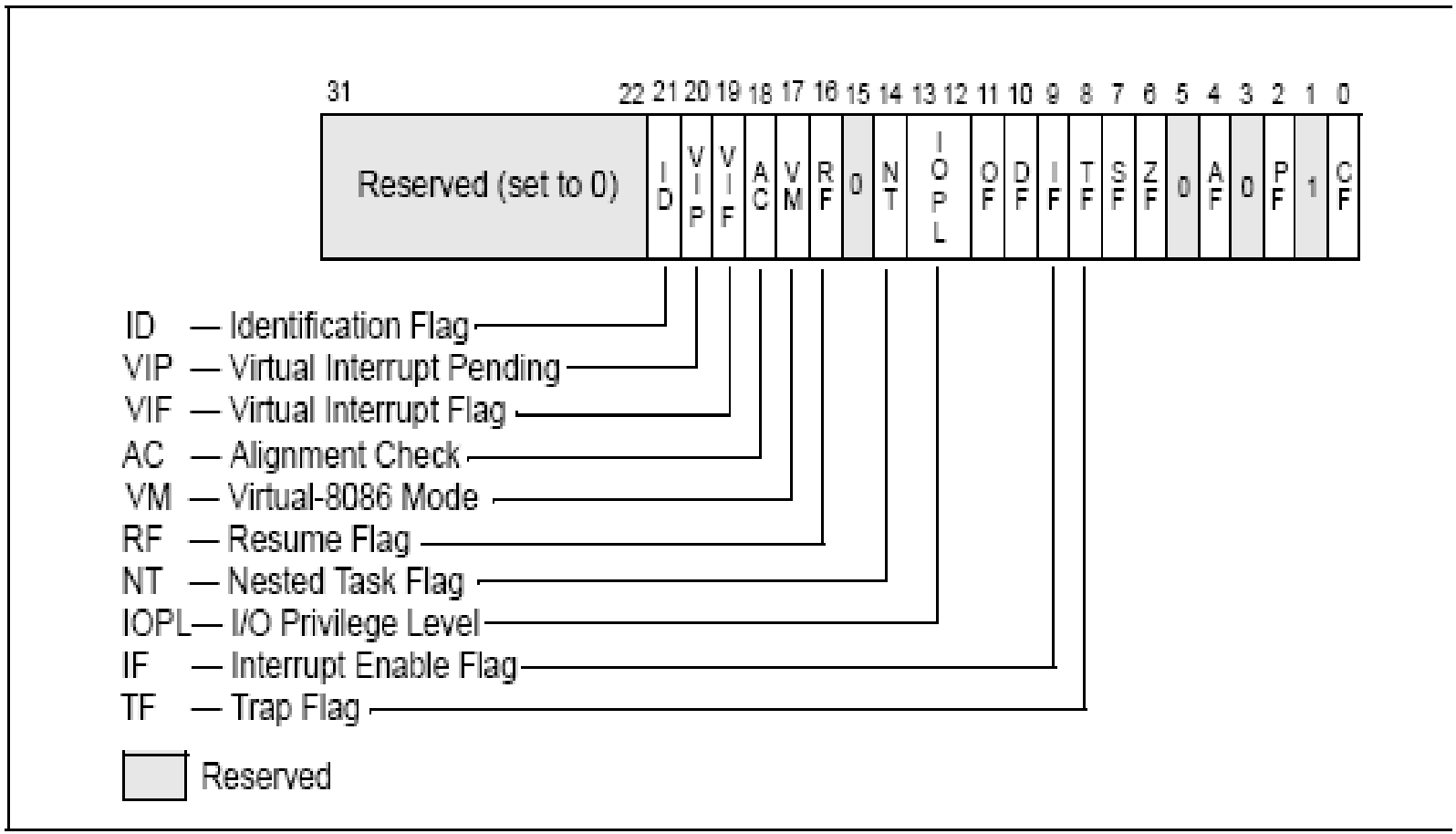
X86-32的通用寄存器

X86-64的系统寄存器

中央处理器CPU-程序状态字PSW

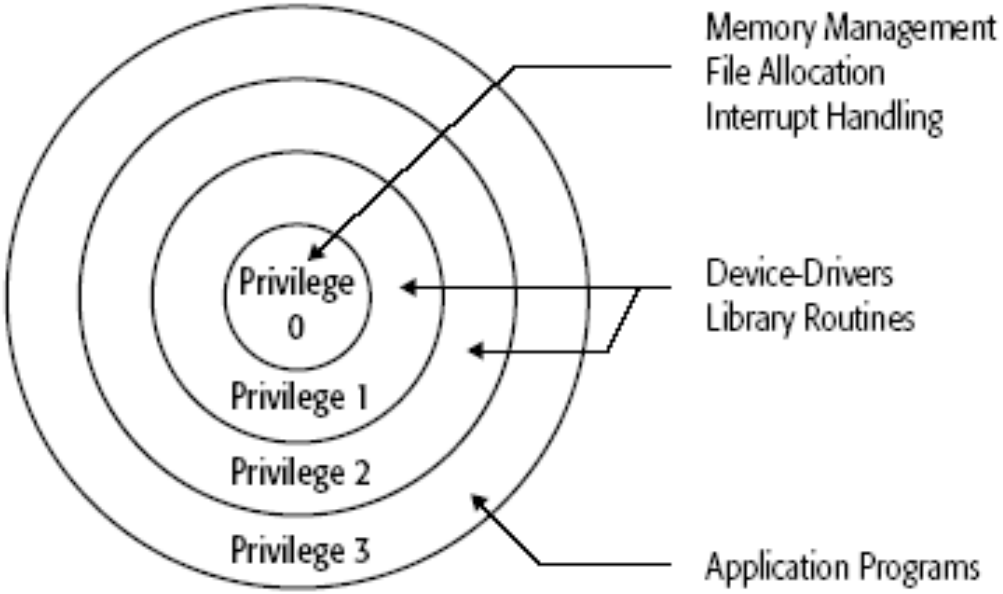
- 作用：指示处理器状态
- 状态代码
  - CPU的工作状态码
    - 指明当前CPU的工作状态是核心态还是用户态
  - 条件码
    - 反映指令执行后的结果特征
  - 中断屏蔽码
    - 指出是否允许中断
- 举例：Pentium 系列
  - CF:进位标志符
  - ZF:结果为零标志符
  - ……

System Flags in the EFLAGS Register



中央处理器CPU-特权指令和非特权指令

- 处理器的特权级
- 两类指令
  - 特权指令：允许操作系统使用，不允许一般用户使用（如修改程序状态字；设置中断屏蔽；启动I/O设备；清内存；设置时钟；停机等）
  - 非特权指令：一般用户可用的



X86的处理器特权级

中央处理器CPU-处理器状态

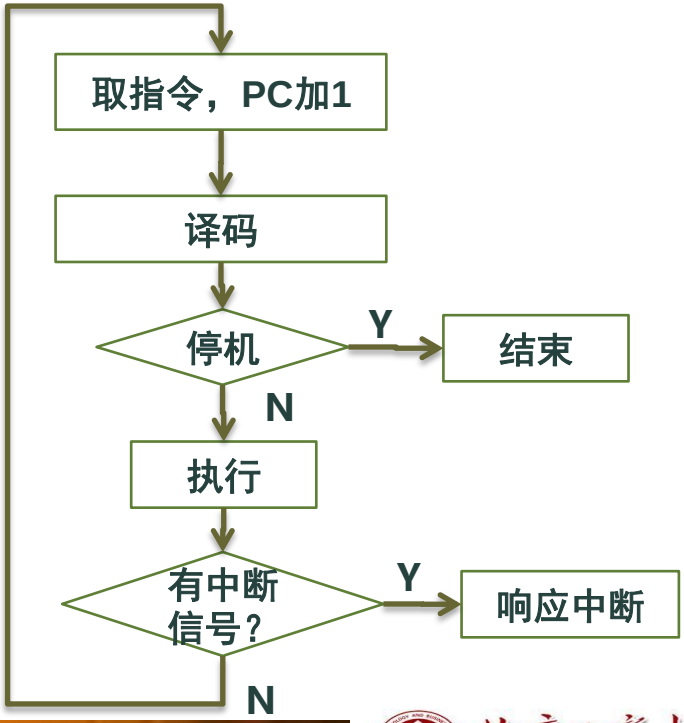
- 核心态：操作系统管理程序运行的状态
  - 能执行指令全集(包括特权，非特权指令)，具有改变CPU状态的能力，操作系统在核心态下运行
- 用户态：用户程序运行时的状态
  - 只能执行非特权指令，用户程序在用户态下运行
  - 如果在用户态下用户执行了核心态指令，则产生中断，由操作系统得到控制权，而特权指令被停止

中央处理器CPU - cpu工作方式

- 指令的类别
  - 访问存储器指令：负责处理器和存储器之间的数据传送
  - I/O指令：负责处理器和I/O模块之间的数据传送和命令发送
  - 算术逻辑指令（数据处理指令）：执行有关数据的算术和逻辑操作
  - 控制转移指令：指定一个新的指令的执行起点
  - 处理器控制指令：修改处理器状态，改变处理器工作方式

中央处理器CPU - cpu工作方式

- 指令执行的基本过程
  - 处理器每次从存储器中读取一条指令，在指令完成后，根据指令的类别自动将程序计数器的值变成下一条指令的地址
  - 取到的指令放在处理器的指令寄存器中，处理器解释并执行这条指令
    - 指令周期：单条指令处理的过程



操作系统运行环境

- 计算机层次结构
- 中央处理器(CPU)
- 存储系统
  - 存储器的类型
  - 存储器的层次结构
  - 存储保护
- 中断和异常
- 操作系统运行模型
- 系统调用
- 人机界面

存储系统-存储器的类型

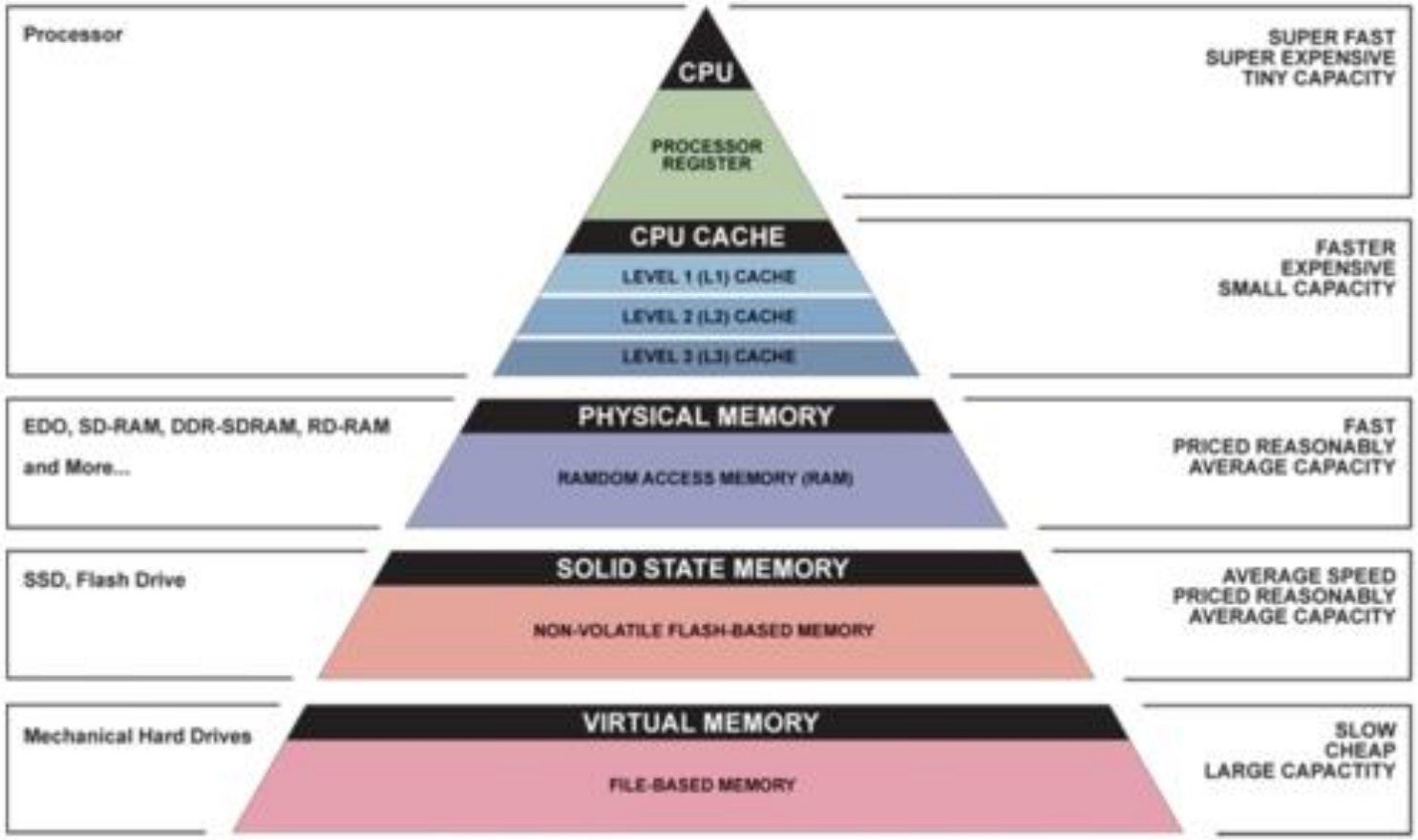
- 读写型存储器
  - 可以把数据存入其中任一地址单元，并且可在以后的任何时候把数据读出来，或者重新存入新的数据存储器
  - 随机访问存储器(RAM:Random Access Memory):存放随机存取程序的数据
- 只读型存储器
  - 只能从中读取数据，但不能随意地用普通的方法向其中写入数据
  - 只读存储器(ROM:Read-Only Memory)
    - PROM：可编程的只读存储器
    - EPROM：可用特殊的紫外线光照射写入此芯片，以擦去其中的信息位，使之恢复原来状态，然后用EPROM写入器写入数据

存储系统-存储器的类型

- 存储分块
  - 存储的最小单位“二进位”，包含的信息为0或1
  - 最小编址单位是字节
    - 一字节—8个二进位；
    - 一个字—2个字节；
    - 双字—4个字节
    - 1KB—1024个字节；1MB—1024个1KB；1GB—1024个1MB
- 主流PC内存：128MB~512MB； 4GB
- 服务器、工作站内存：512MB~4GB；512GB
- 外存：20GB~70GB；4TB

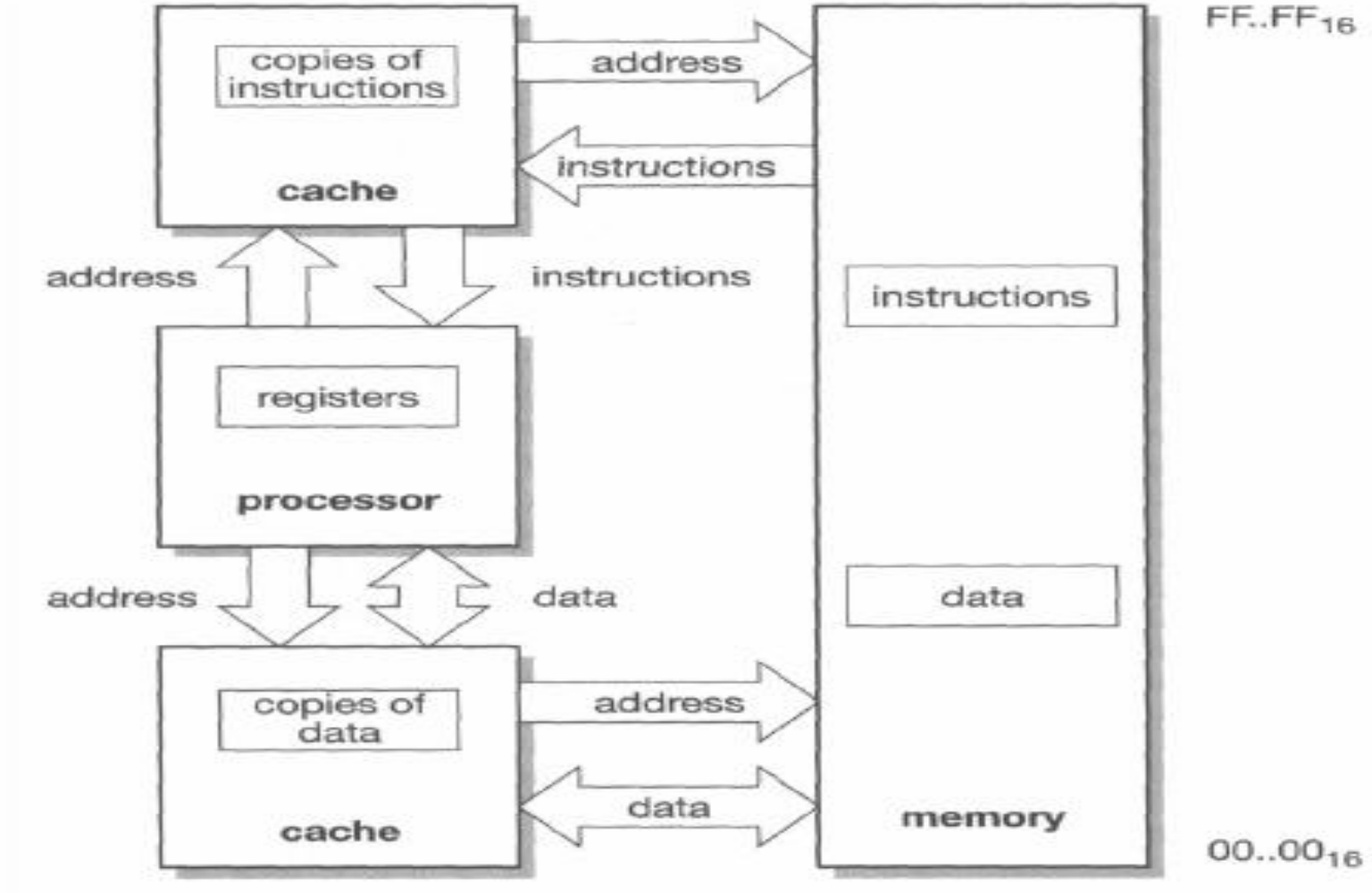
存储系统- 存储器的层次结构

- 存储器设计考虑的三个问题：
  - 容量、速度、成本：需权衡找出最优点



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

处理器-Cache-内存


copies of instructions
address
instructions
cache
address
instructions
instructions
registers
processor
address
data
data
copies of data
address
data
cache
memory
FF..FF16
00..0016

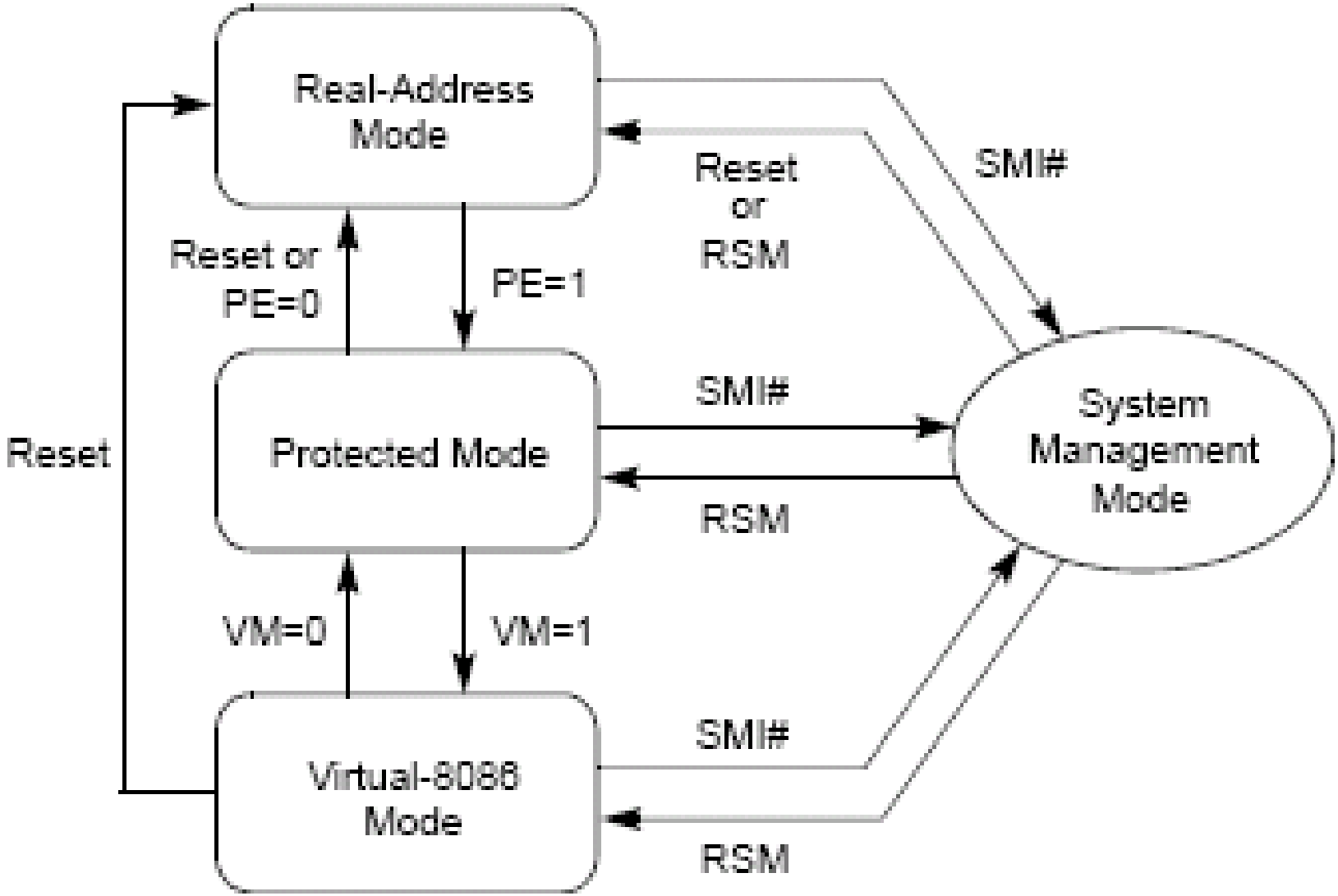
存储系统- 存储访问局部性原理

- 提高存储系统效能
- 原因：
  - 程序中会有很多重复存取指令集合的操作，同样数据存储也有类似情况
- 局部性原理
  - 较短时间内代码和数据能比较稳定地保持在一个存储器局部区域，处理器主要和存储器这个局部区域打交道
- 解决
  - 设计多级存储的体系结构，通过控制访问频率来提高存储系统效能

存储系统-存储保护

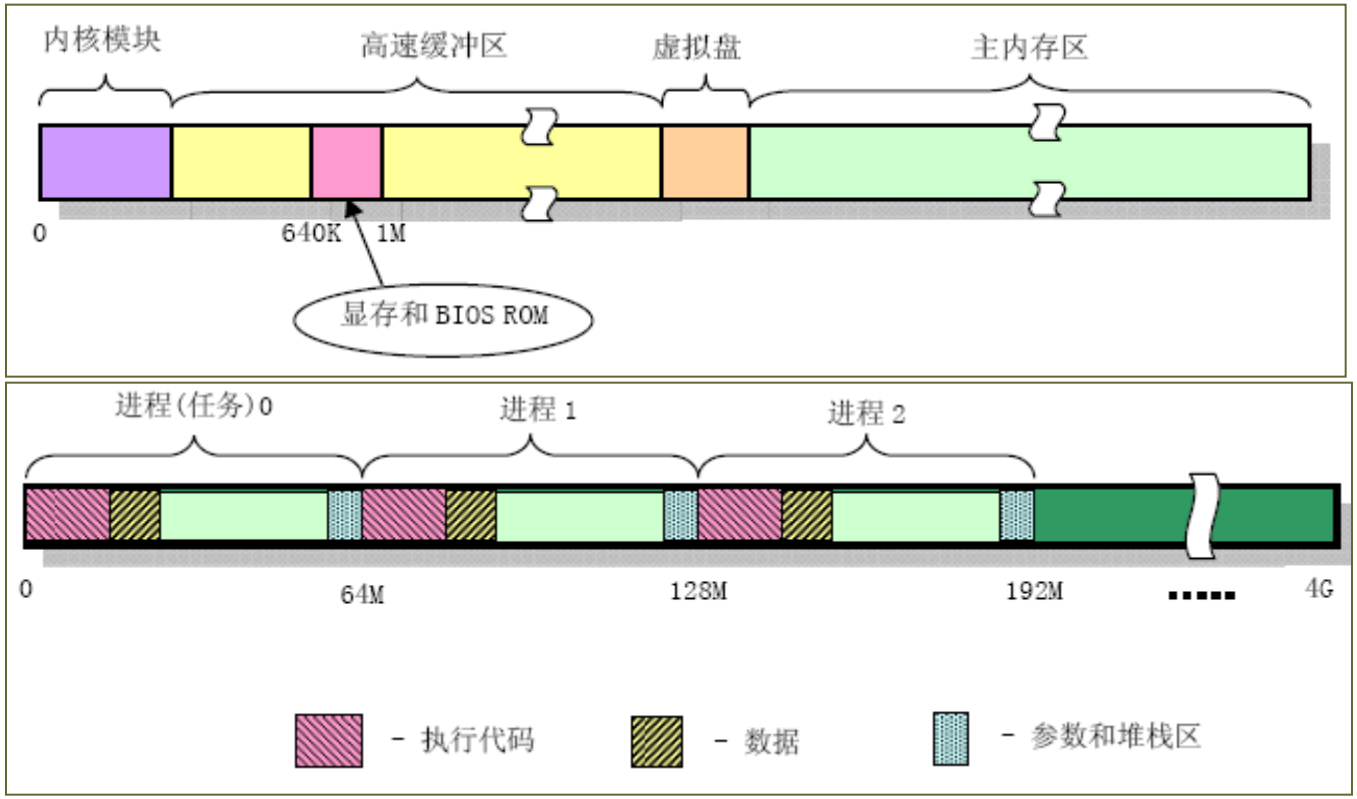
- 多用户，多任务操作系统：OS给每个运行进程分配一个存储区域
  - 问题：
    - 多个程序同时在同一台机器上运行，怎样才能互不侵犯？
- 存储保护的目的：
  - 防止用户程序破坏OS
  - 防止用户程序互相干扰

Intel处理器的操作模式

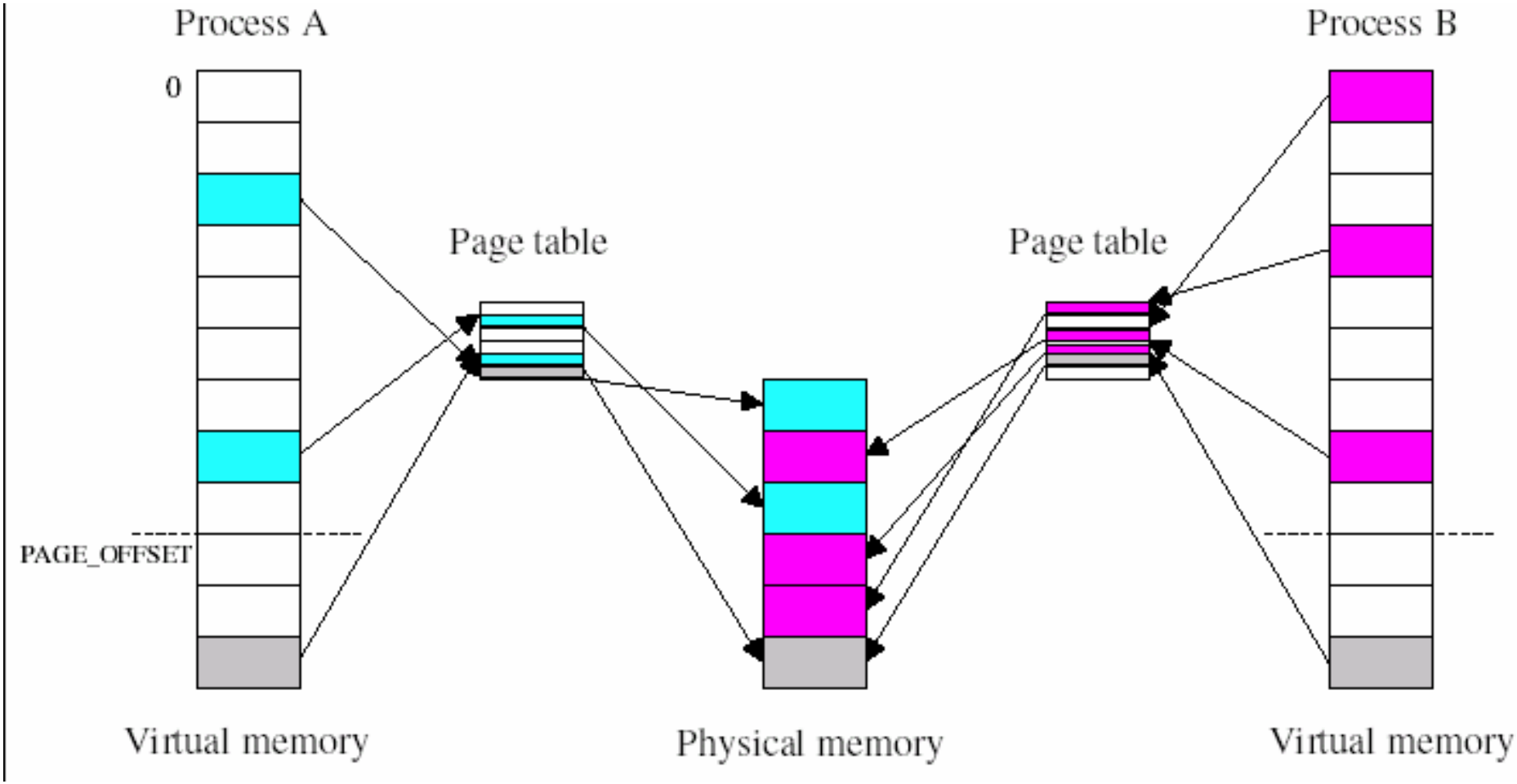


Linux0.11 的内存使用

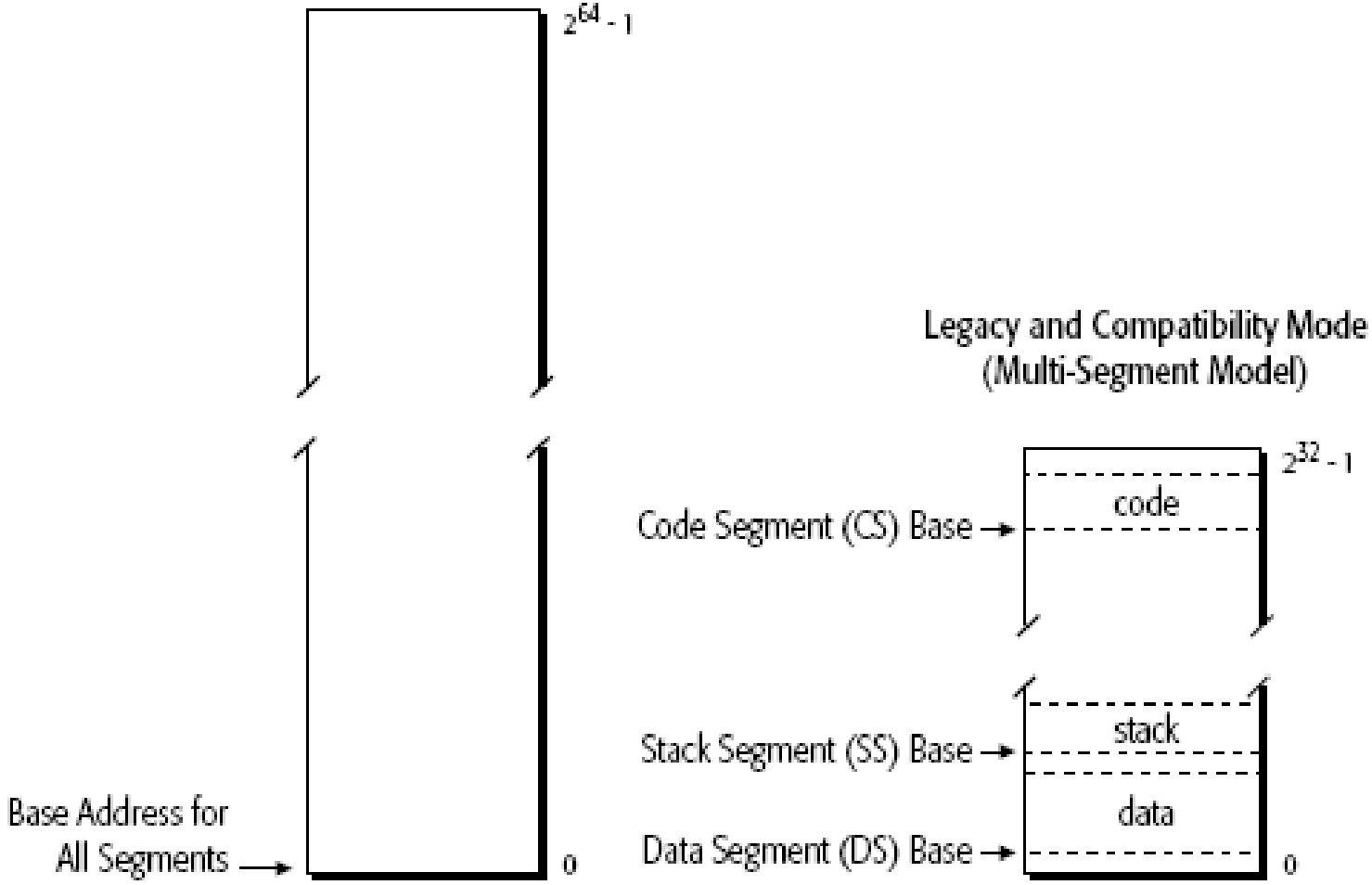
- 物理内存的布局
- 虚拟内存的划分
- 虚拟内存到物理内存的映射：页式管理



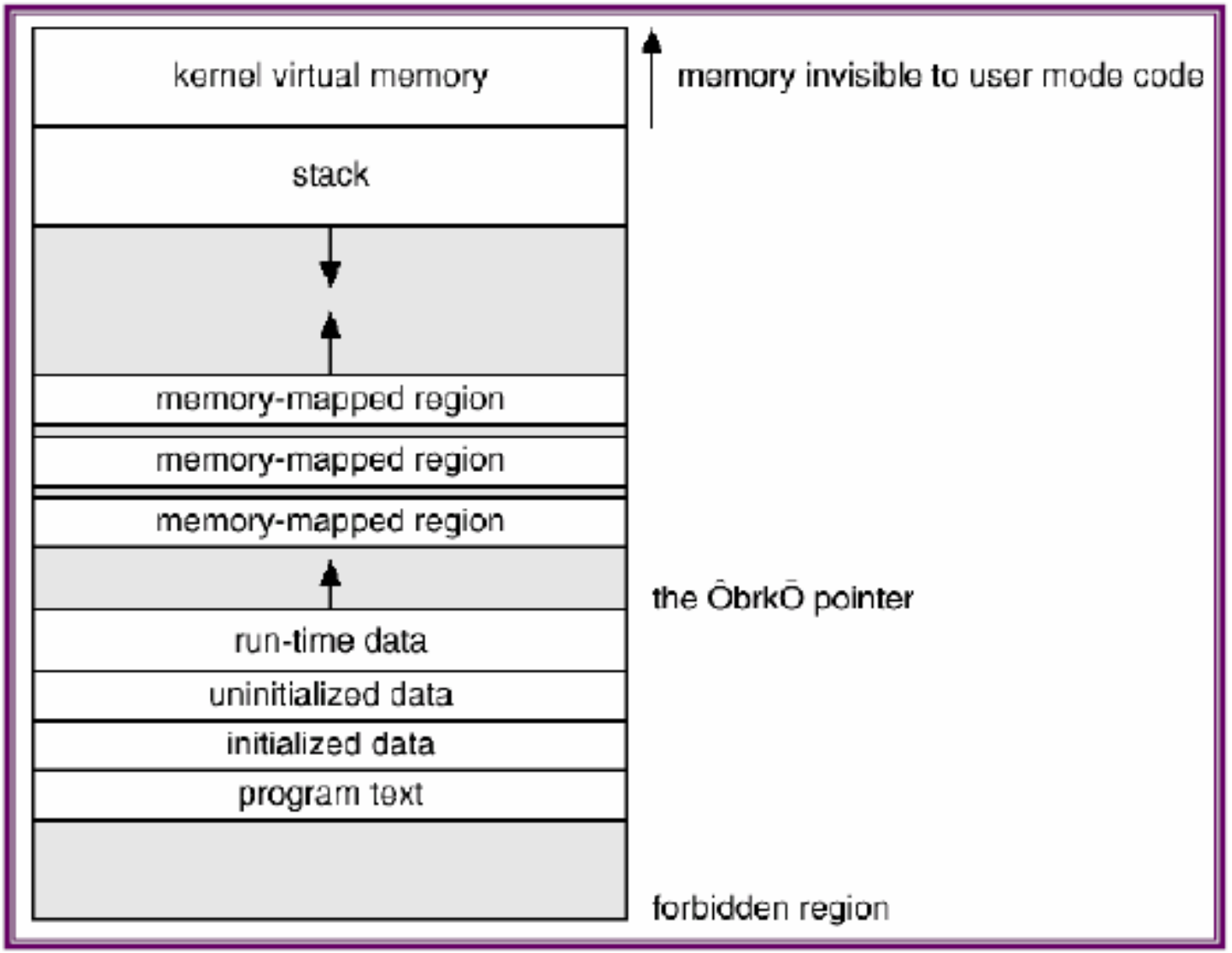
进程的虚拟内存和物理内存



虚拟地址空间和段的分配

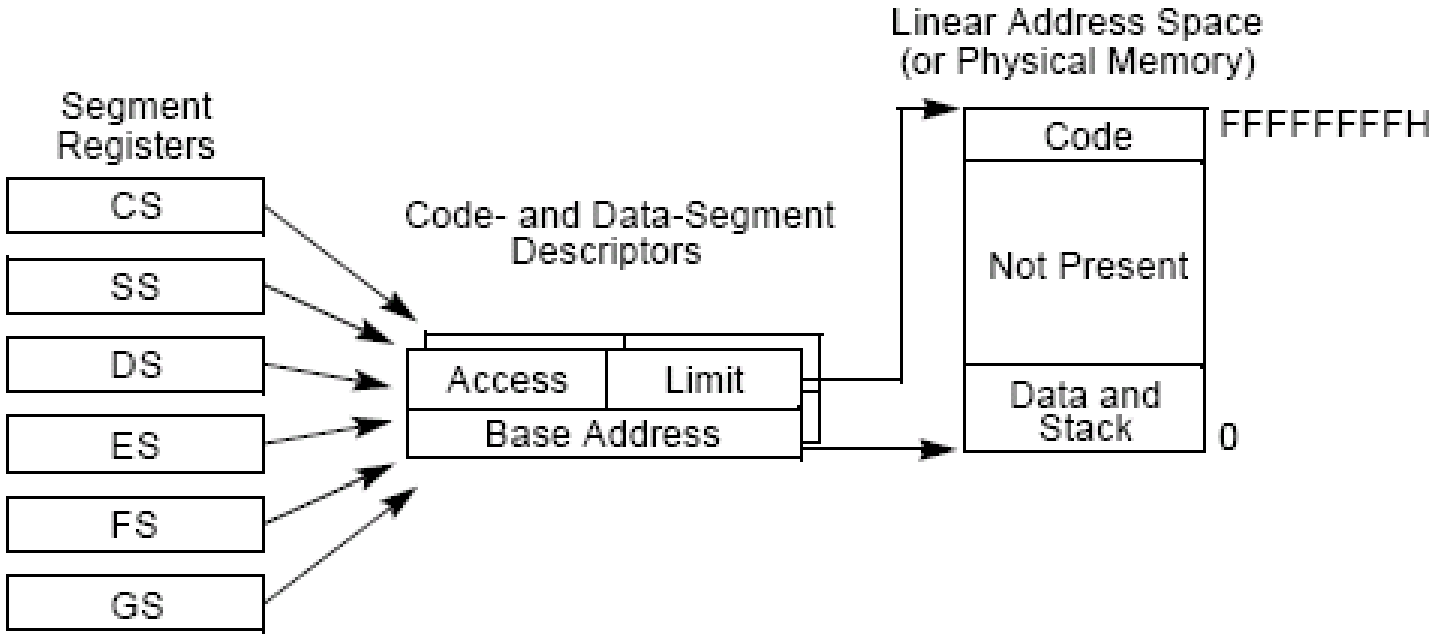


程序在虚拟内存中的布局



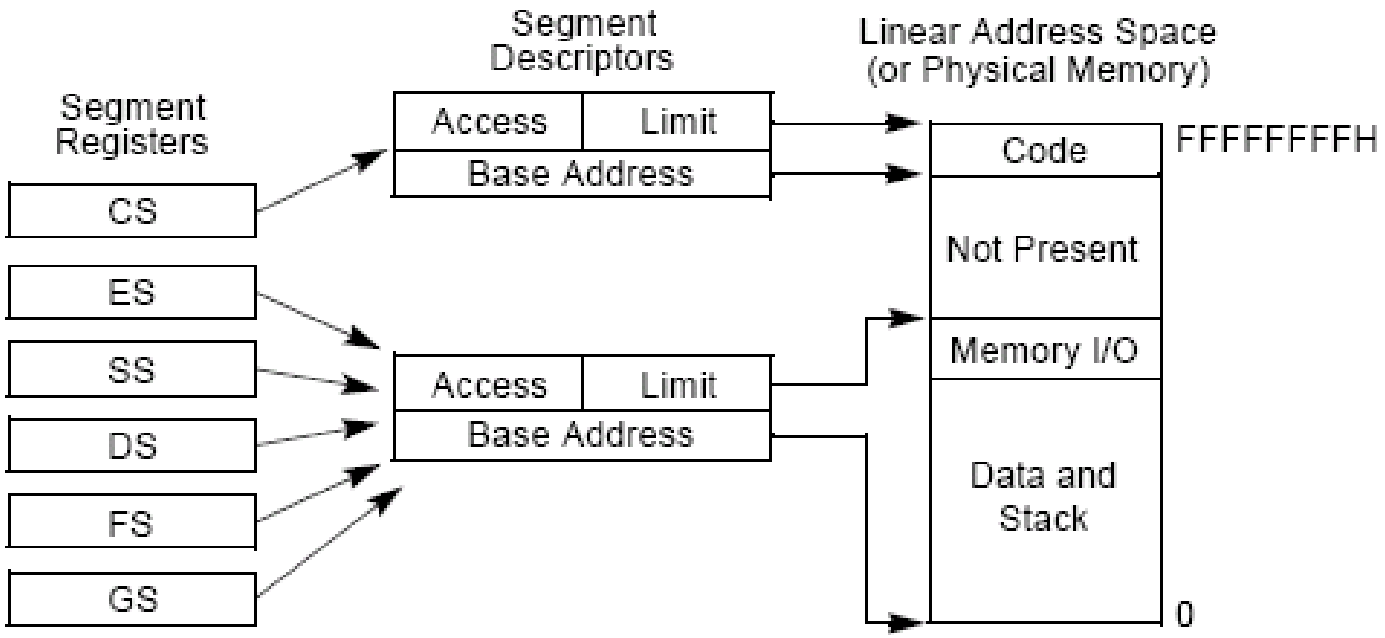
分段与保护-1

- 平面模式：没有实质的分段，也没有保护



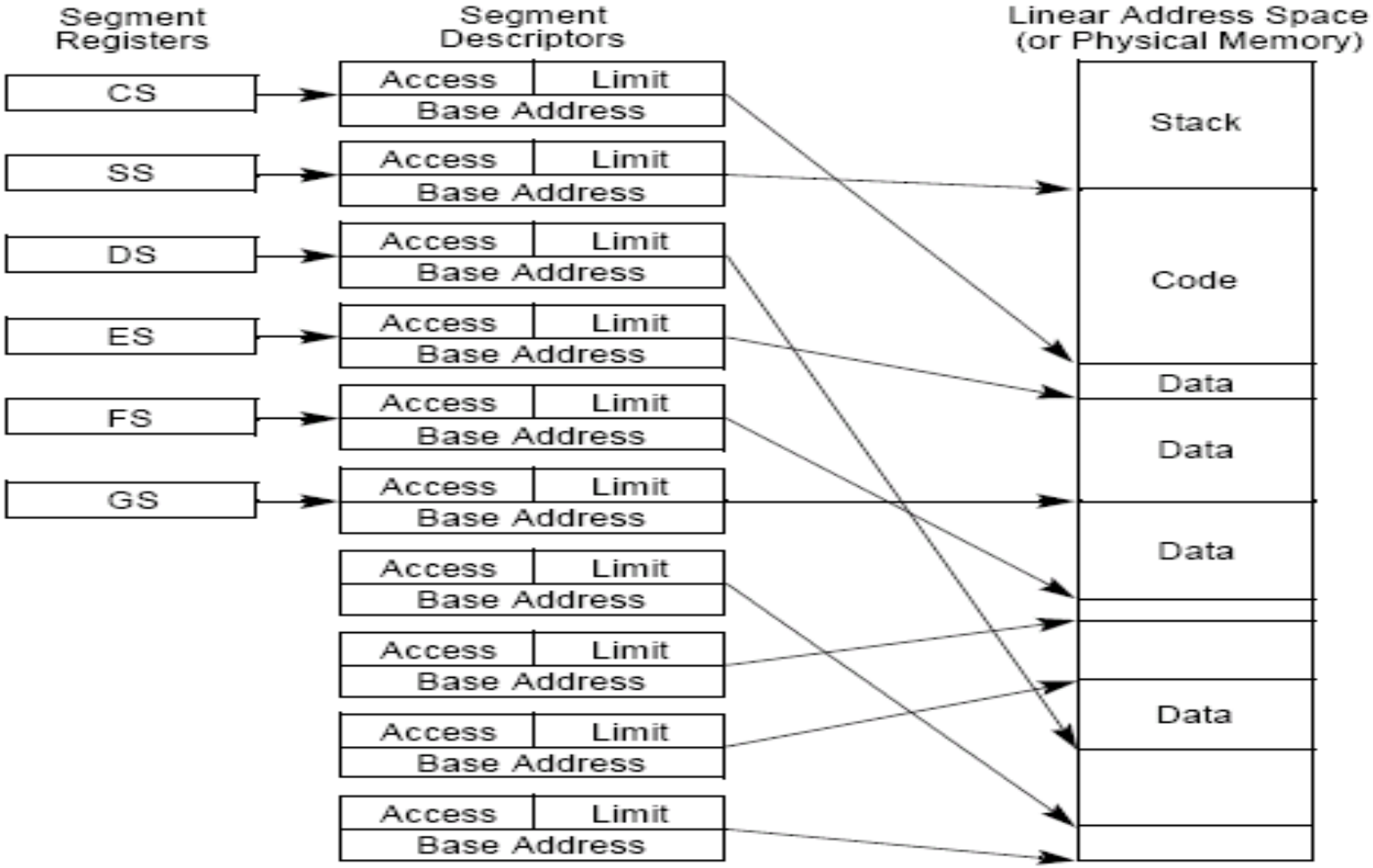
分段与保护-2

- 保护的平面模式：简单保护

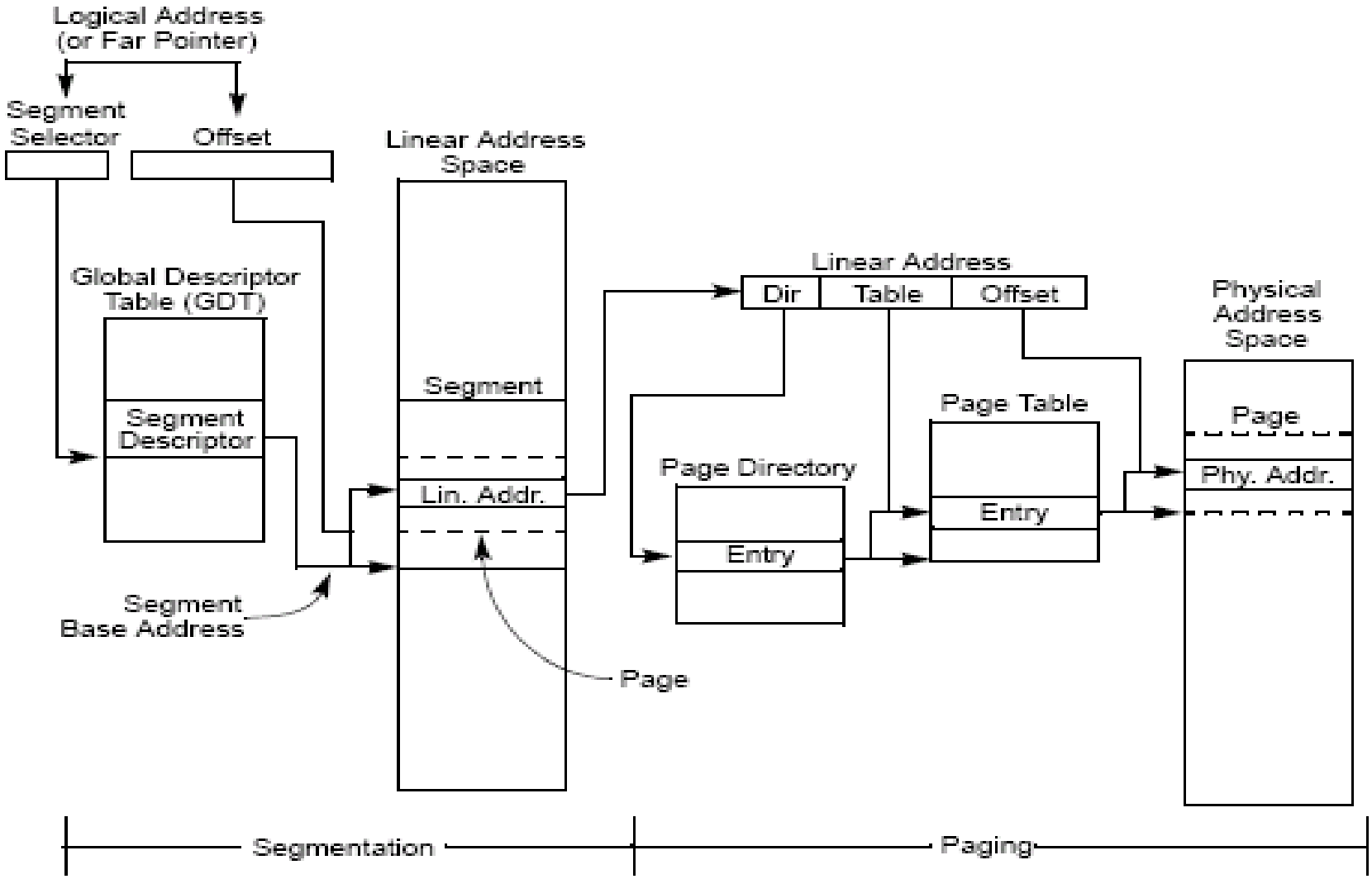


分段与保护-3

- 多段模式：完全的硬件保护



分段和分页


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

操作系统运行环境

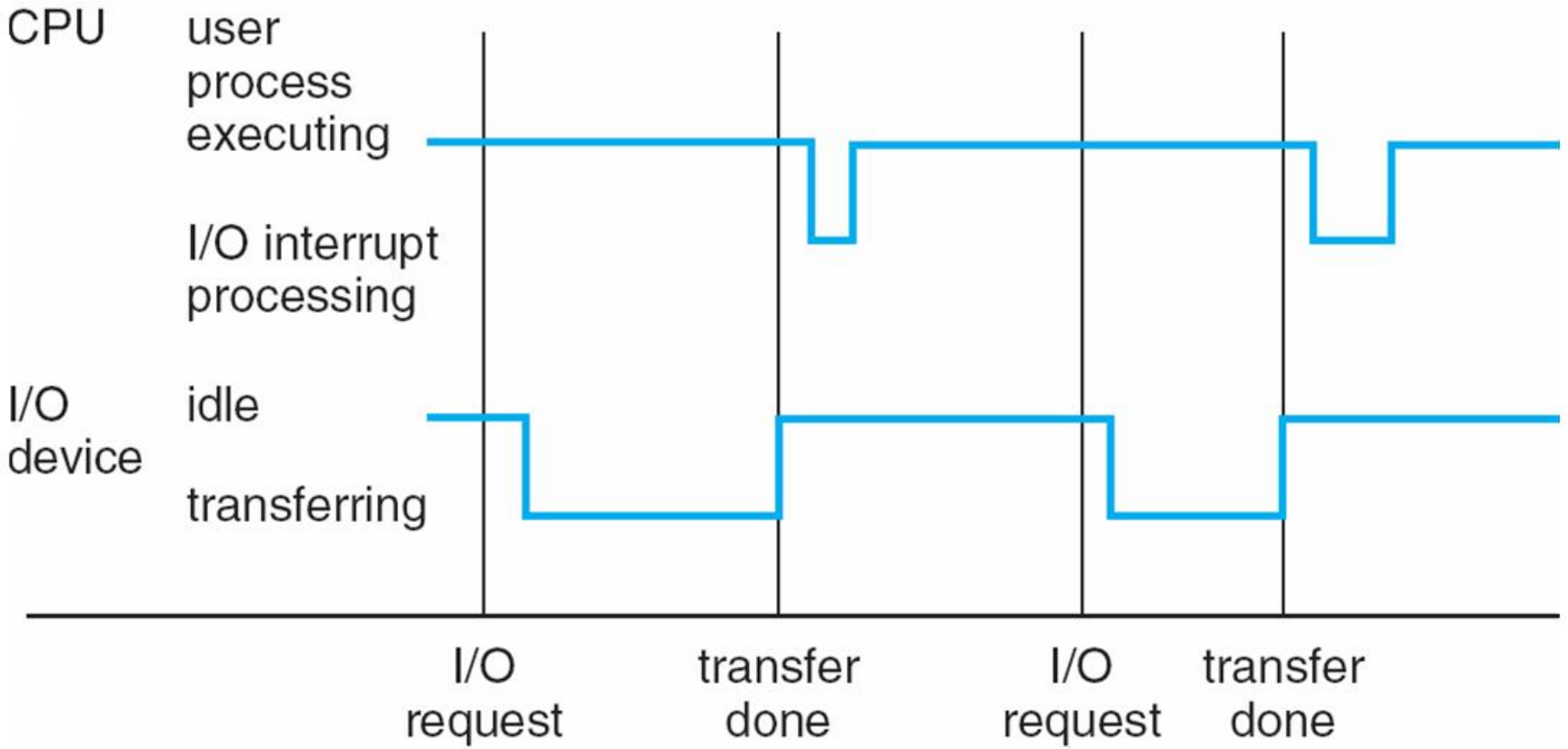
- 计算机层次结构
- 中央处理器(CPU)
- 存储系统
- 中断和异常
  - 中断和异常的概念
  - 中断/异常响应和处理
- 操作系统运行模型
- 系统调用
- 人机界面

中断与异常-中断概念

- 引入中断的作用
  - 中断是现代计算机系统中基本设施之一
  - 是实现多道程序的必要条件

- 引入中断的目的
  - 解决主机与外设的并行工作问题
  - 提高可靠性
  - 实现多机联系
  - 实现实时控制

中断处理的时序


CPU
user process executing
I/O interrupt processing
I/O device
idle
transferring
I/O request
transfer done
I/O request
transfer done

中断与异常-中断概念

- 中断定义：
  - CPU对系统中或系统外发生的异步事件作出的响应
    - 异步事件：发生时间没有系统约束的一类事件
  - CPU暂停正在执行的程序，保留现场后自动转去执行相应事件的处理程序，处理完成后返回断点，继续执行被打断的程序
- 特点
  - 中断是随机的
  - 中断是可恢复的
  - 中断是自动处理的

中断与异常-异常概念

- 异常
  - CPU执行指令本身过程中出现的无法继续执行指令的情况
  - CPU暂时终止当前的执行流程，转到相应的错误处理程序或陷入处理程序
- 陷入
  - 处理器执行一条“陷入”指令，由用户态转入核心态
- 最初，中断和异常统称为中断

中断与异常

外中断（中断）
- I/O中断、外部信号中断
- 定时器中断
- 调试中断

处理机、内存

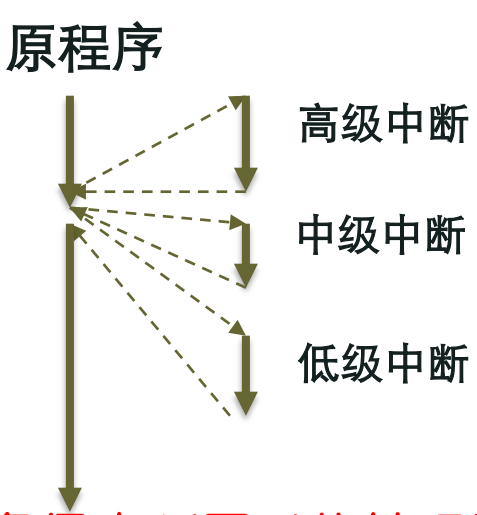
内中断（异常，陷入）
- 程序运行错误：地址非法、校验错、页面失效、存取访问控制错、算术操作溢出、时间片中断、用户程序执行特权指令、用户态到核心态切换

中断与异常的对比

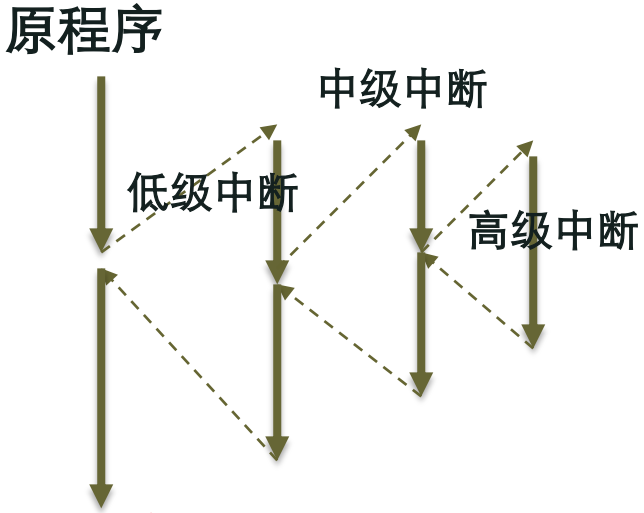
- 与现行指令的关系
  - 中断与现行指令无关，可以屏蔽
  - 异常由现行指令引起，不可以屏蔽
- 处理程序与当前进程的关系
  - 中断处理程序不是为当前进程服务
  - 异常处理程序为当前进程提供服务
- 响应时机
  - CPU在执行完一条指令后响应中断
  - CPU在执行指令中可响应异常
- 进程上下文不同
  - 中断处理程序在系统上下文中执行
  - 异常处理程序在各自的进程上下文中执行

中断的分级

- 中断优先级
  - 由硬件指定给中断源，软件可以定制
  - 表示引起中断的事件的重要性与紧迫程度
  - 是中断享有的响应权利
- 当有多个中断同时发生时，系统根据中断优先级决定响应中断次序
  - 优先响应 级别高 的中断
  - 级别高的中断可以打断级别低的中断的处理过程



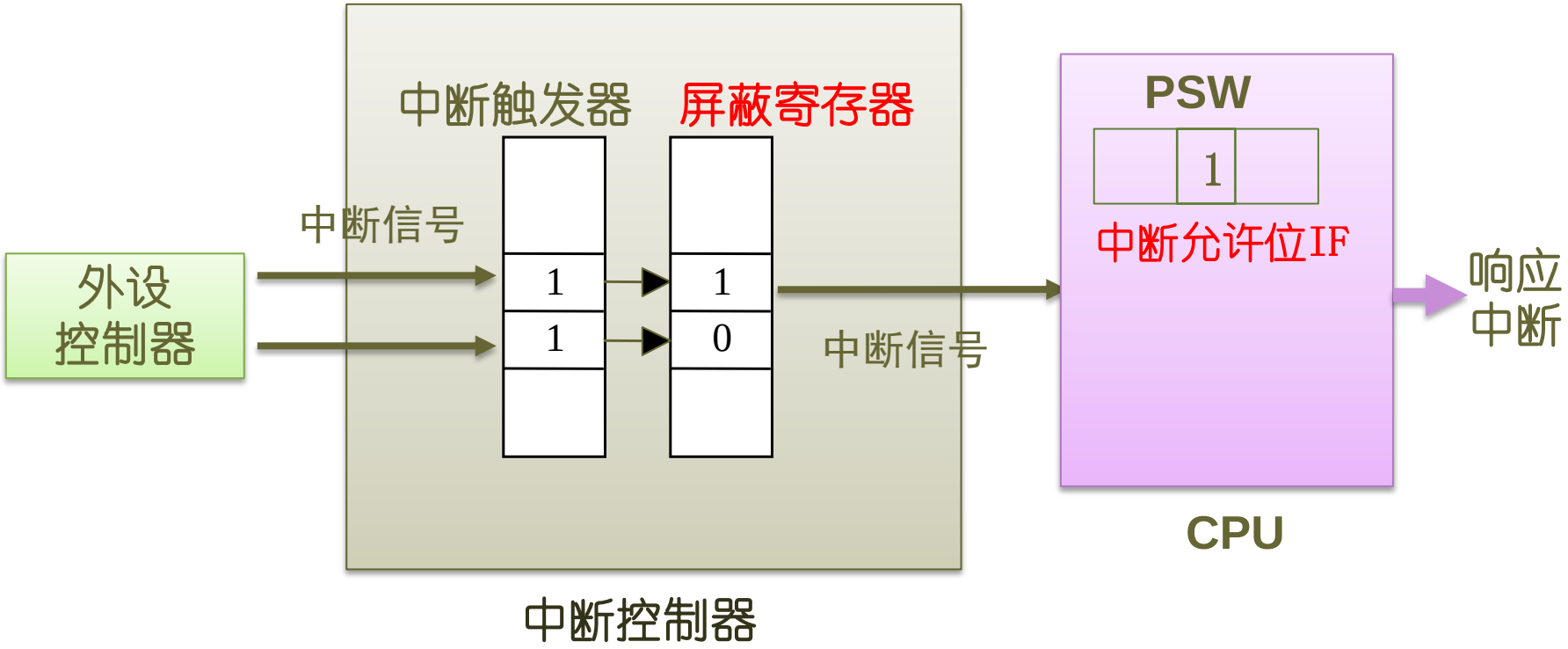
多级中断同时的处理过程



高级打断低级的嵌套处理过程

中断屏蔽

- 禁止中断出现或者禁止响应中断
- 操作系统可以设置中断屏蔽位调节中断响应顺序

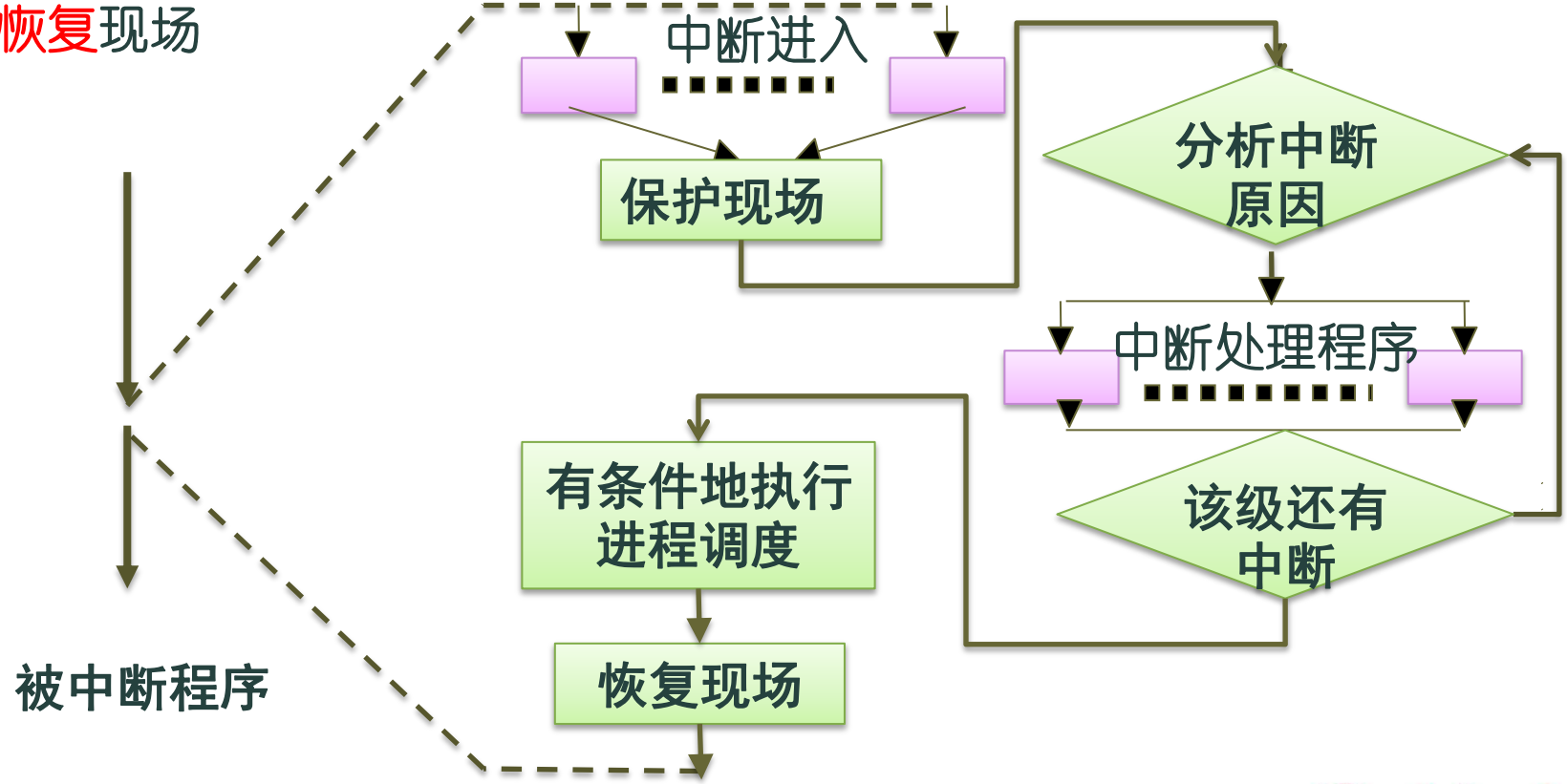


中断/异常响应和处理

- 中断响应
  - 发现中断、做出响应的过程
- 异常响应
  - 执行指令的时候，发现异常转入异常处理程序的过程
- 中断处理：从进入中断处理程序开始至恢复被中断程序执行的过程
  -

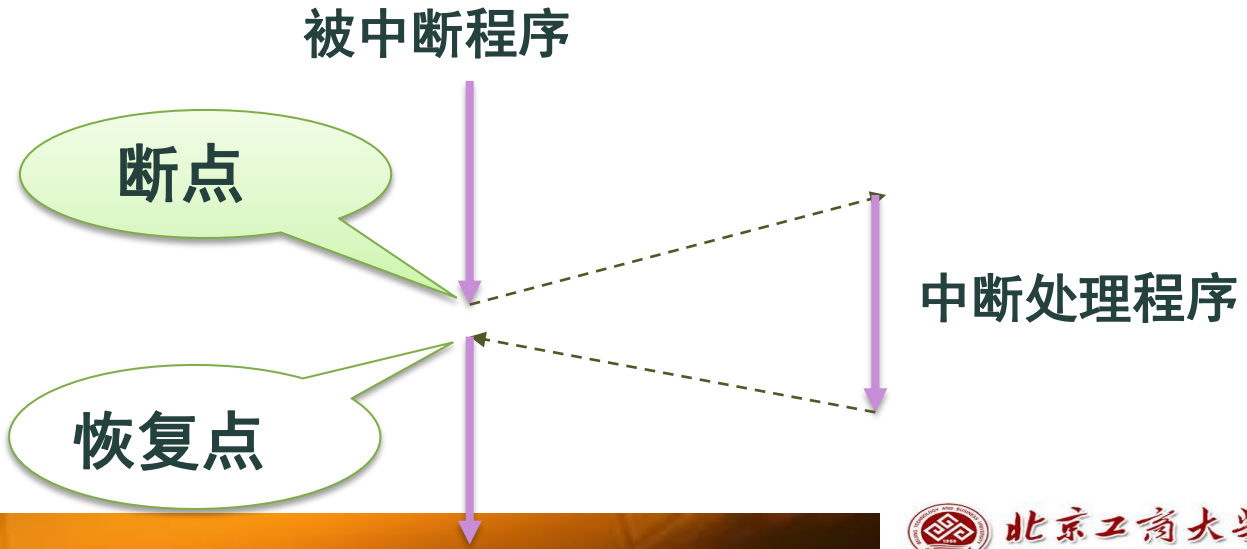
中断/异常处理过程

- 一般的处理过程
  - 进入中断处理程序，保存现场
  - 分析原因，调用具体中断处理程序（函数）
  - 执行中断处理程序
  - 恢复现场



断点和恢复点

- 断点
  - 检测到中断时，处理机刚执行完的那条指令地址
  - PC程序计数器的值
- 恢复点
  - 断点的逻辑后续指令地址
- 现场信息
  - 指中断那一刻确保被中断程序能继续运行的有关信息. 如PC，通用寄存器，特殊寄存器等.



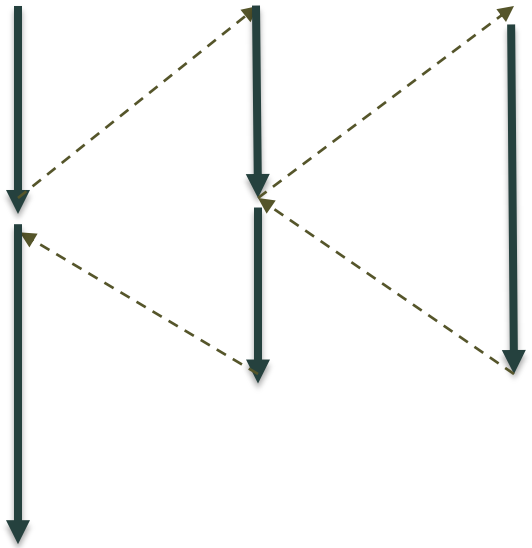
找到中断处理程序

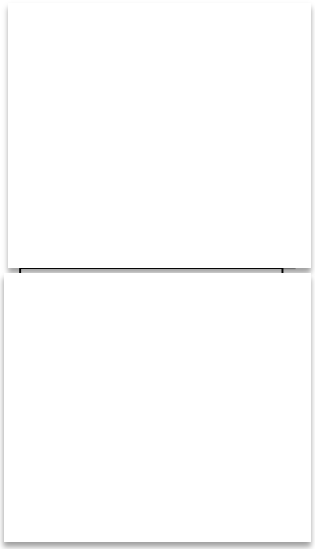
- 中断/异常向量及PS寄存器
  - 中断向量
    - 存放中断处理程序入口地址和程序运行所需处理机状态字的内存单元
  - PS寄存器
    - 描述了CPU的执行状态
    - 其中包含处理机优先级，表示CPU能够响应的最低中断优先级

保存现场

- 现场保存/恢复过程示意
  - 现场区组织 成 栈 结构

被中断程序　低级中断1　高级中断2

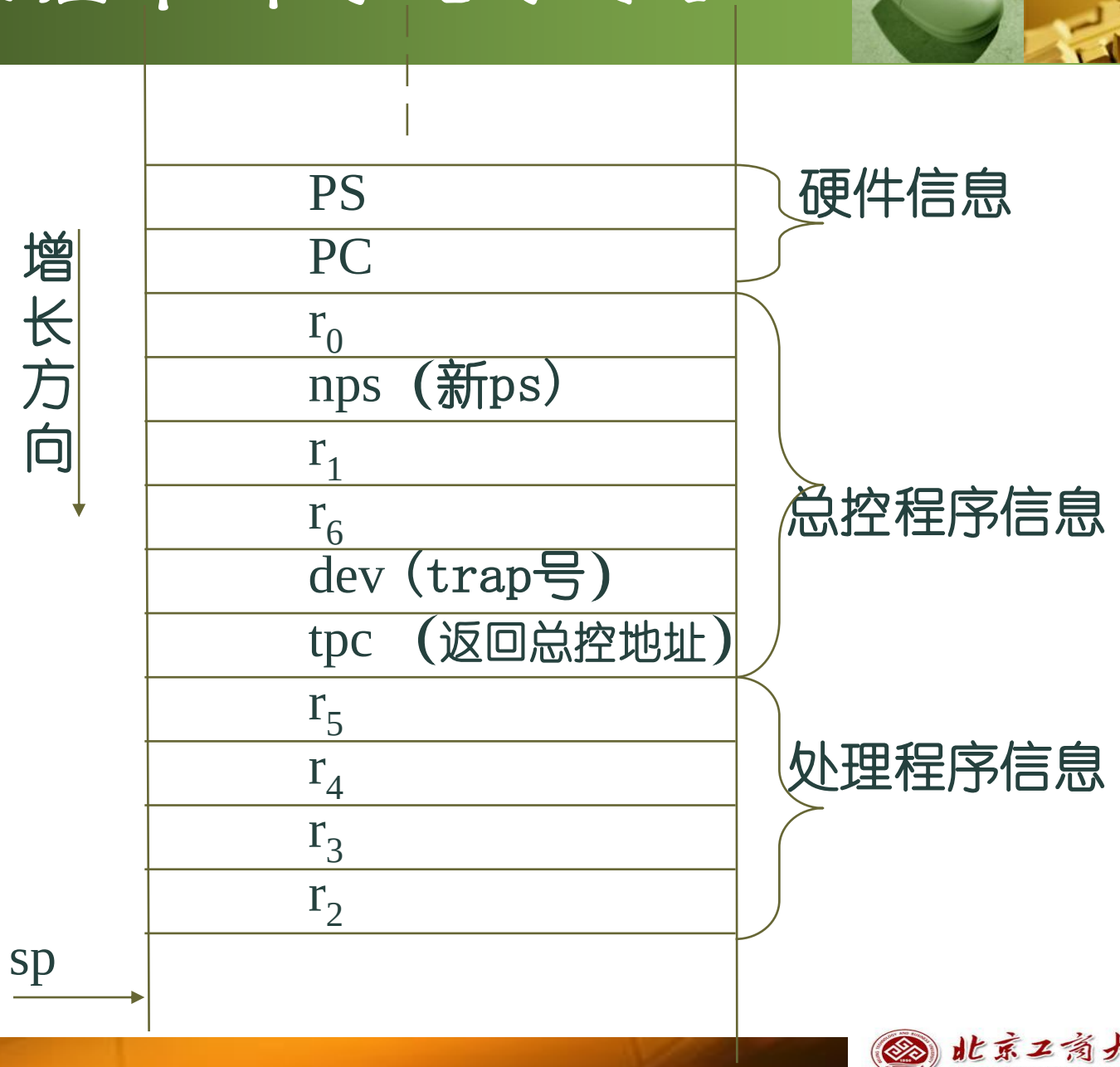




中断2的现场

中断1的现场

UNIX响应中断的现场内容



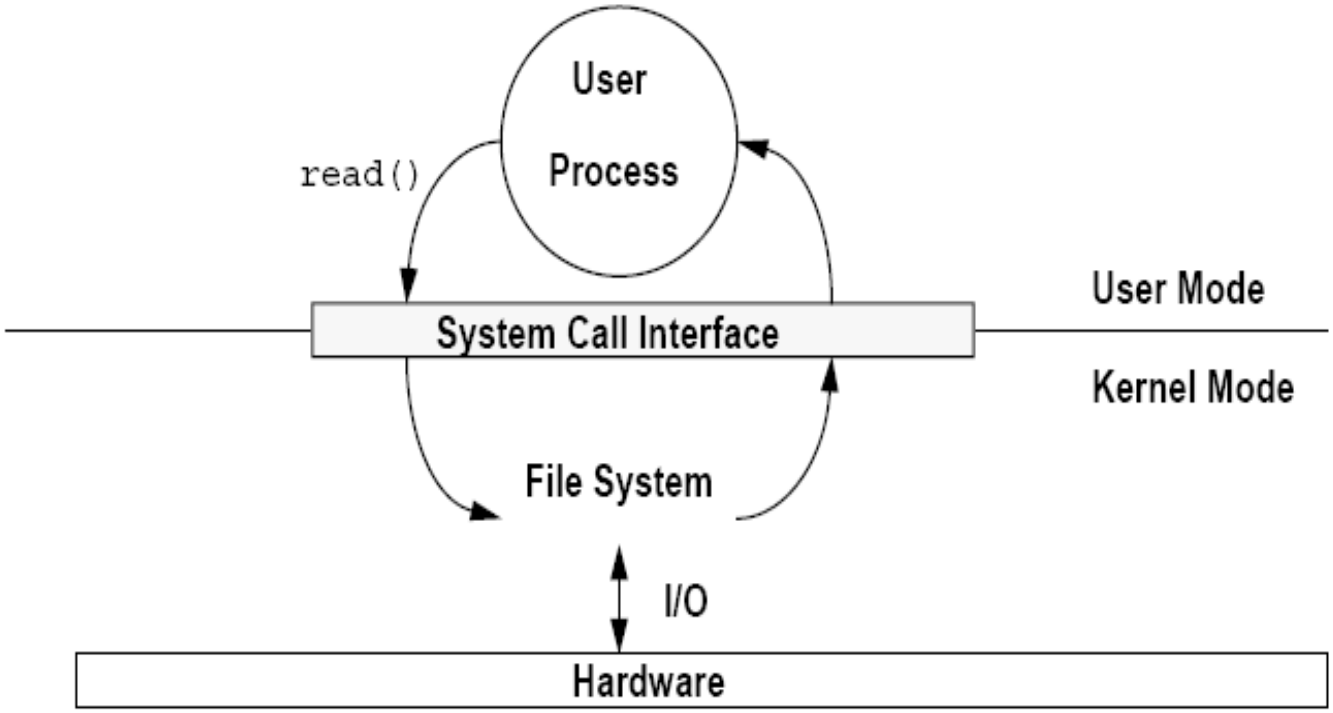
操作系统运行环境

- 计算机层次结构
- 中央处理器(CPU)
- 存储系统
- 中断与异常
- 操作系统运行模型
- 系统调用
- 人机界面

操作系统运行模型

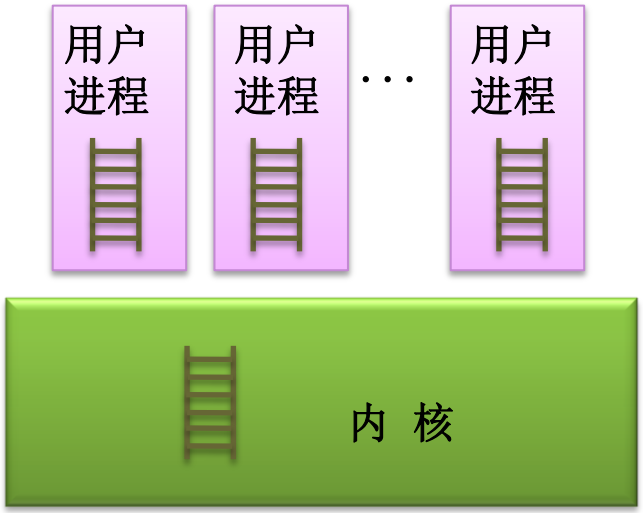
- 操作系统核心的主要功能模块
  - 系统初始化模块
    - 准备系统运行环境，为每个终端创建一个命令解释程序进程
  - 进程管理模块
    - 处理进程类系统调用（如进程创建/结束等）；进程调度
  - 存储管理模块
    - 配合进程管理分配进程空间；处理存储类系统调用（如动态增加进程空间）；虚存系统缺页异常时调入页面处理
  - 文件管理模块
    - 处理文件类系统调用
  - 外设管理模块
    - 负责外设驱动，中断处理

操作系统内核如何执行呢？



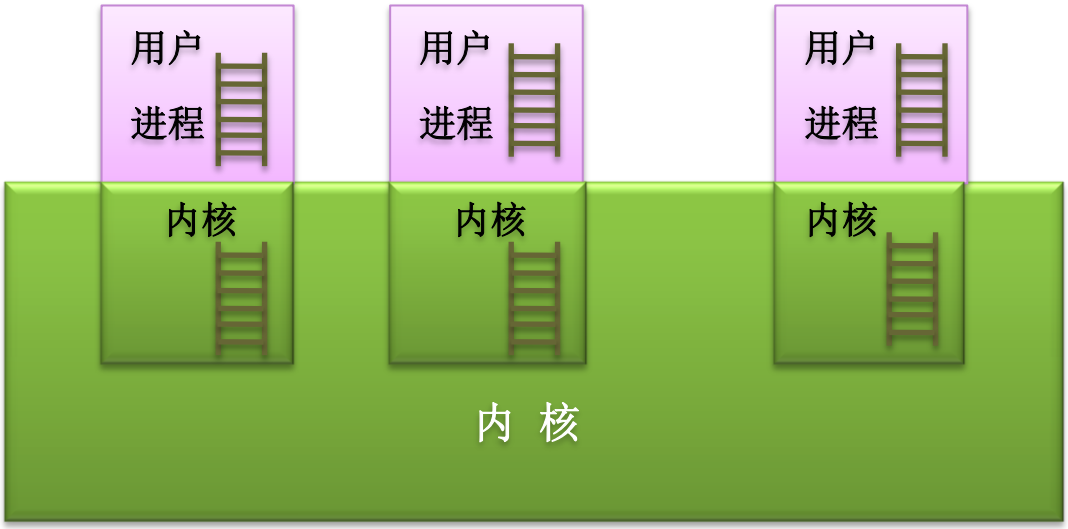
三种操作系统运行模型

- 独立运行的内核
  - 内核程序通过中断/异常机制启动运行
  - 用户程序与核心程序在分离的运行环境中运行
  - 核心程序作为一个独立的特殊执行体运行，有自己独立的运行栈



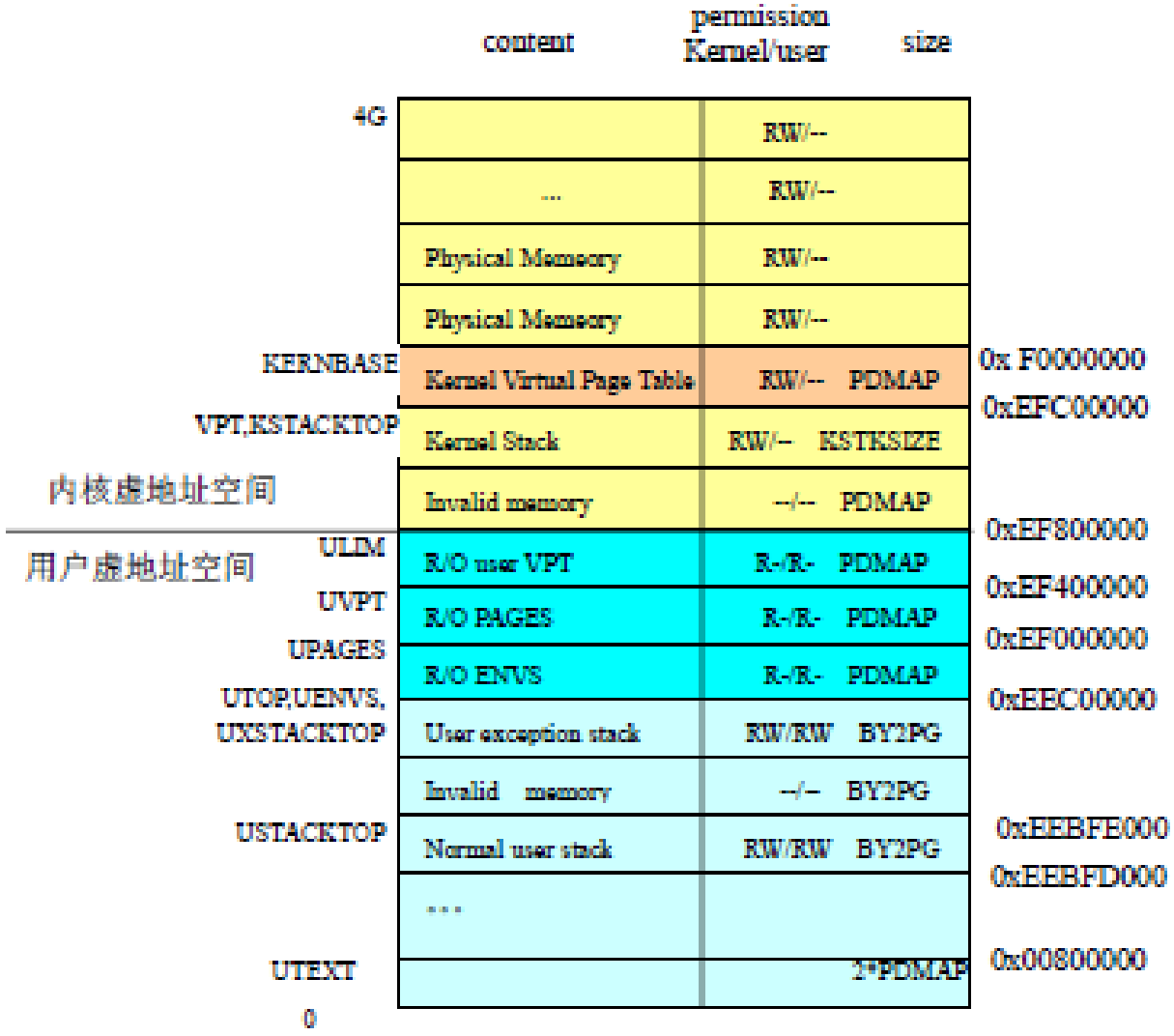
三种操作系统运行模型

- 嵌入用户进程执行模式
  - 操作系统核心程序通过中断/异常机制启动运行
  - 内核运行于被打断进程的核心栈上
  - 内核程序执行并发性好



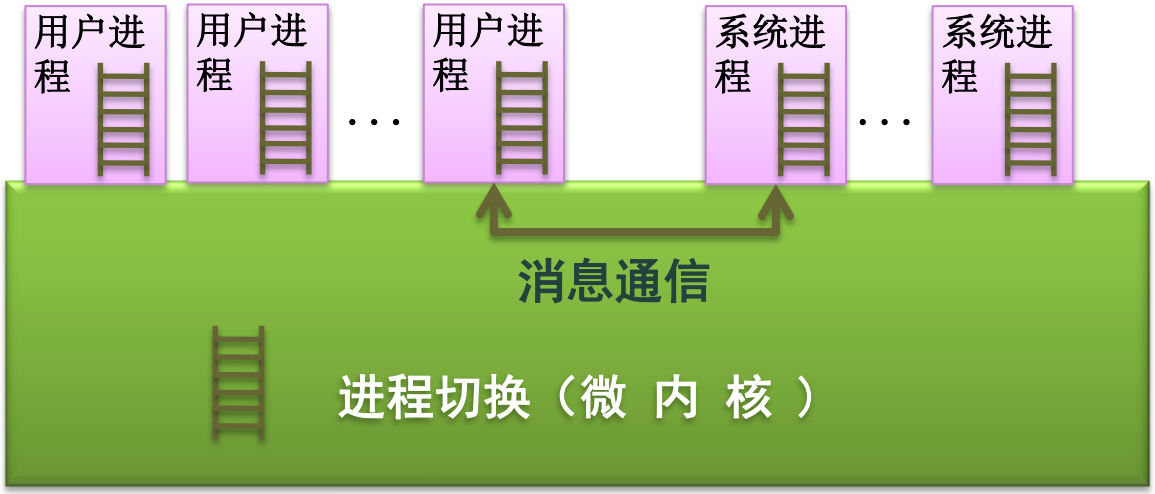
操作系统的地址空间布局

- 进程的地址空间划分为内核空间和用户空间



三种操作系统运行模型

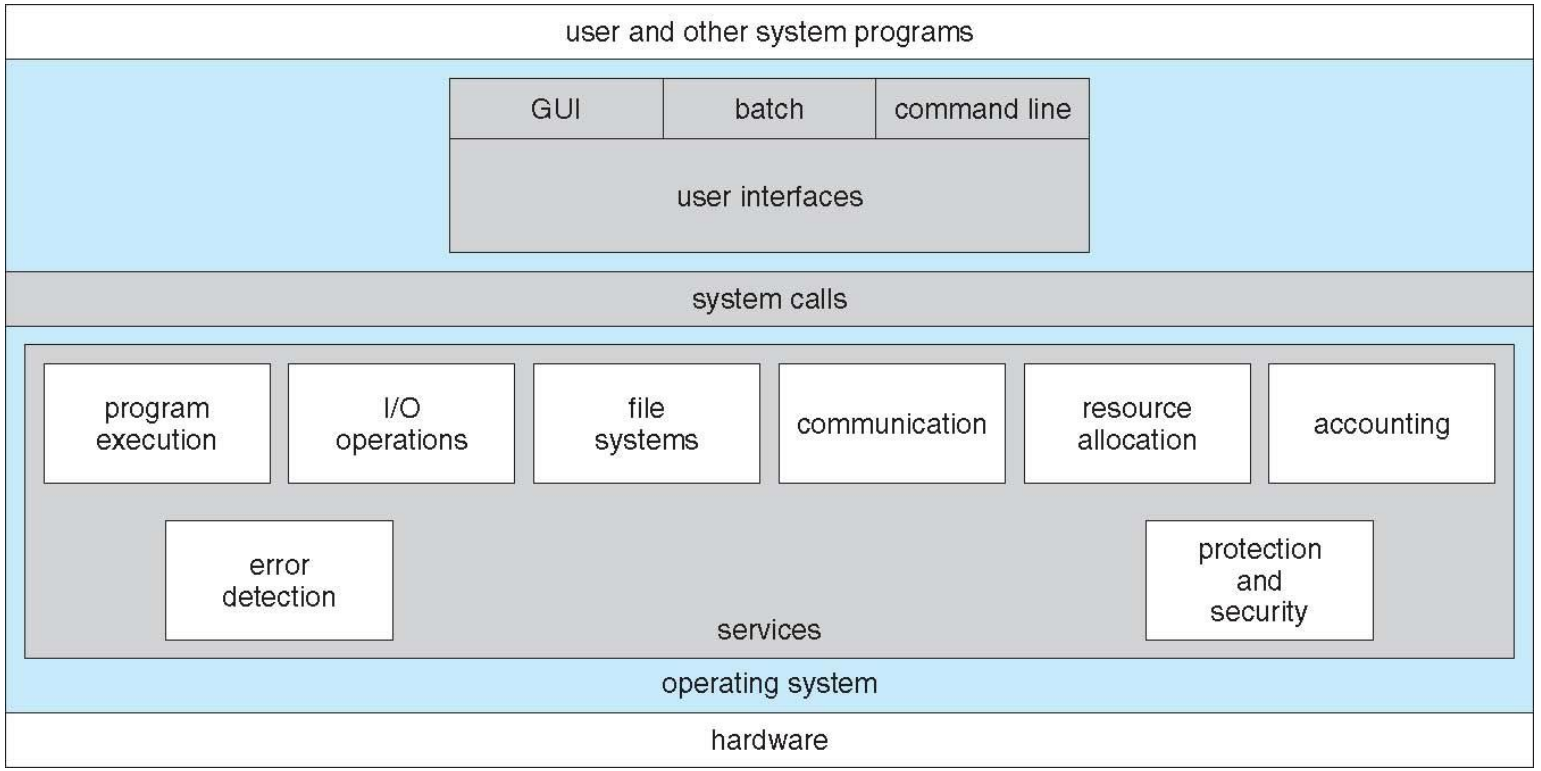
- 微内核模式
  - 内核只包含中断处理，系统调用总控，进程调度等功能
  - 其他功能由用户态运行的系统进程实现
  - 系统开销很大



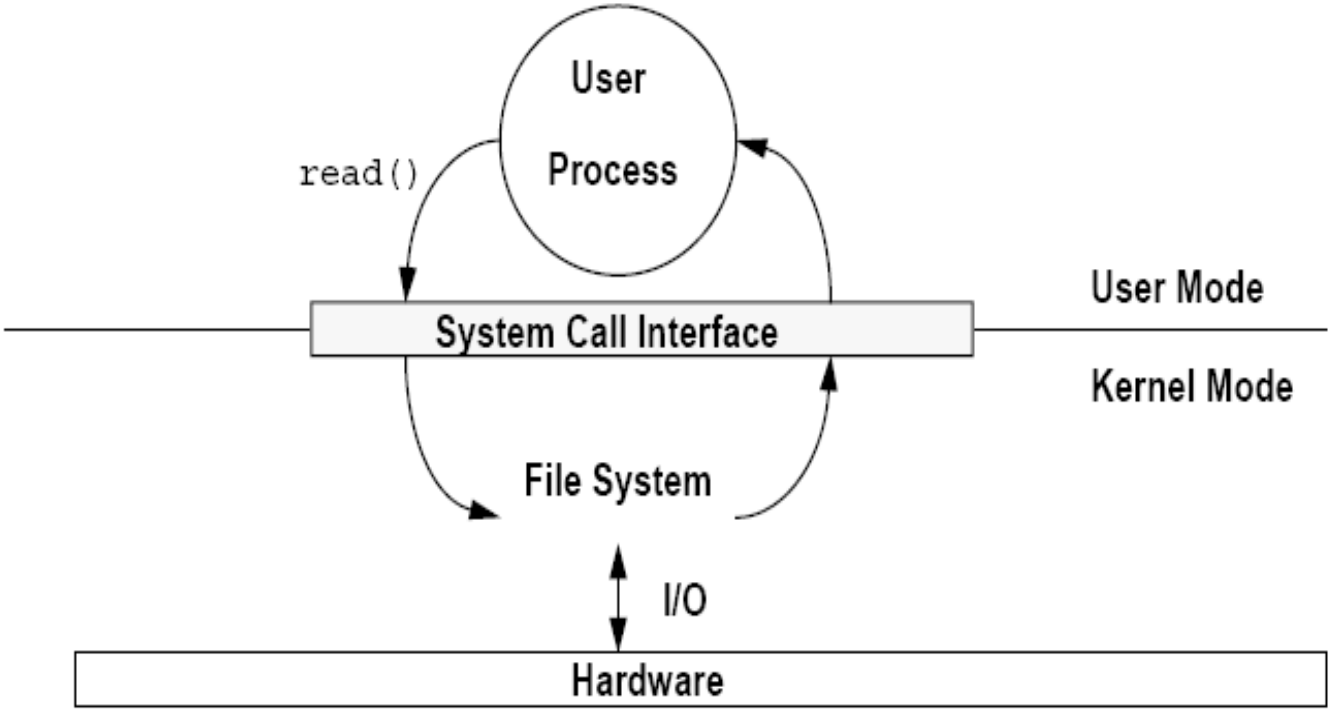
操作系统运行环境

- 计算机层次结构
- 中央处理器(CPU)
- 存储系统
- 中断与异常
- 操作系统运行模型
- 系统调用
- 人机界面

OS提供的服务

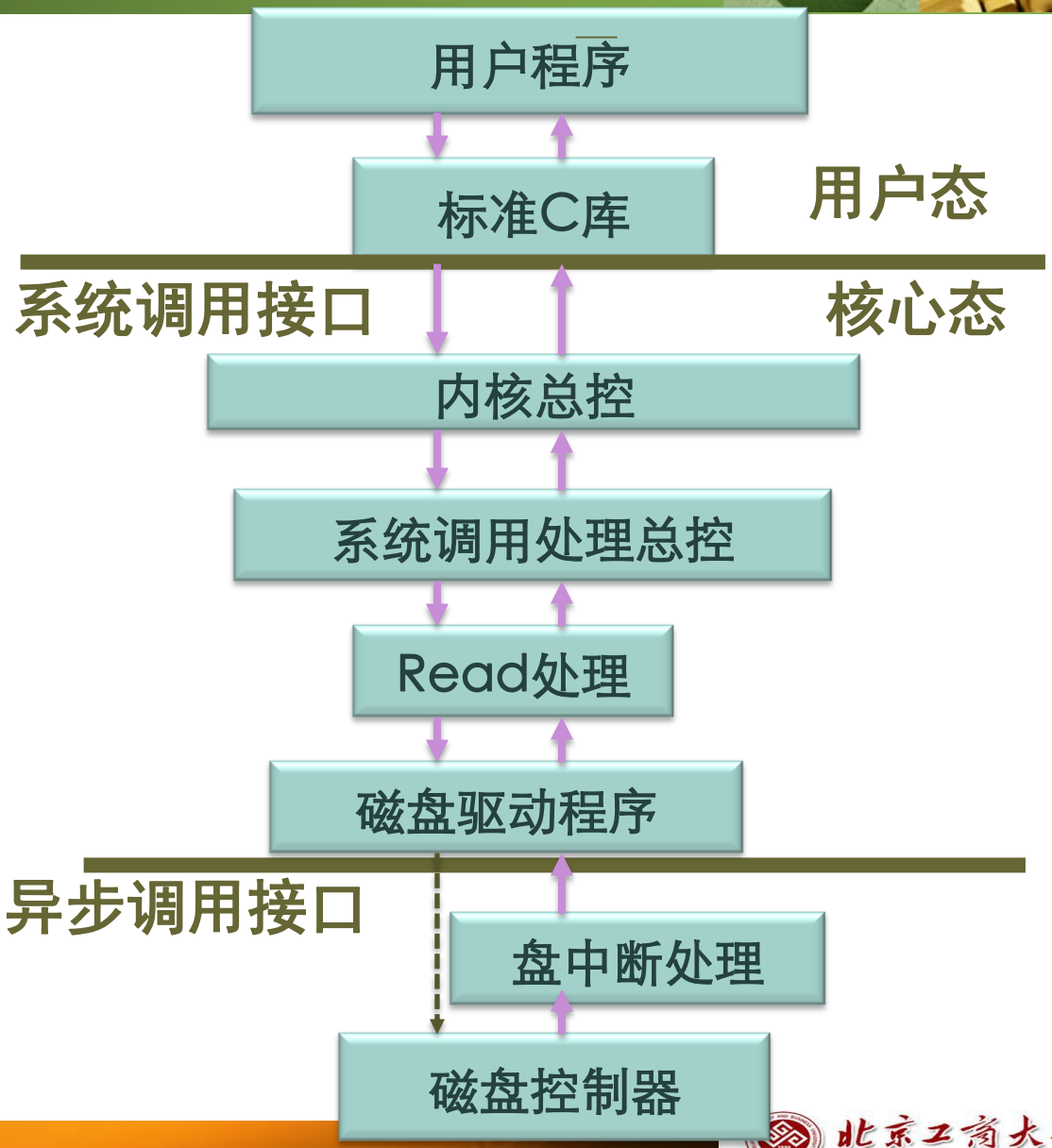

user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

系统调用


User
Process
read()
System Call Interface
User Mode
Kernel Mode
File System
I/O
Hardware

系统调用的实现

- Read系统调用的处理过程



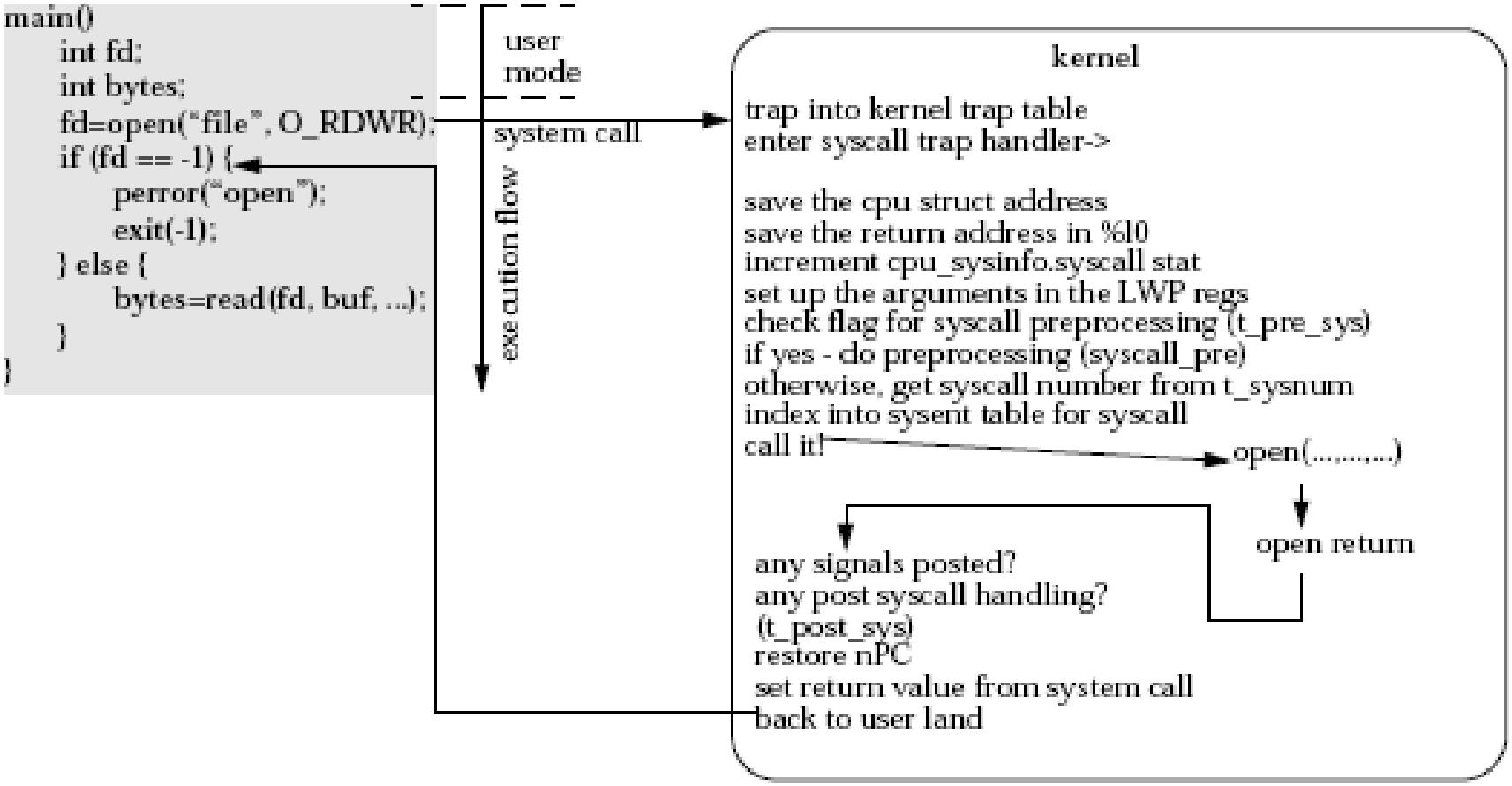
系统调用

- 系统调用是操作系统内核和用户态运行程序之间的接口
- 操作系统提供系统调用接口，让用户程序通过该接口使用系统提供的各种功能
- 系统调用可以看成是用户程序在程序级请求操作系统为之服务的一种手段
- 通过系统调用，用户程序陷入内核

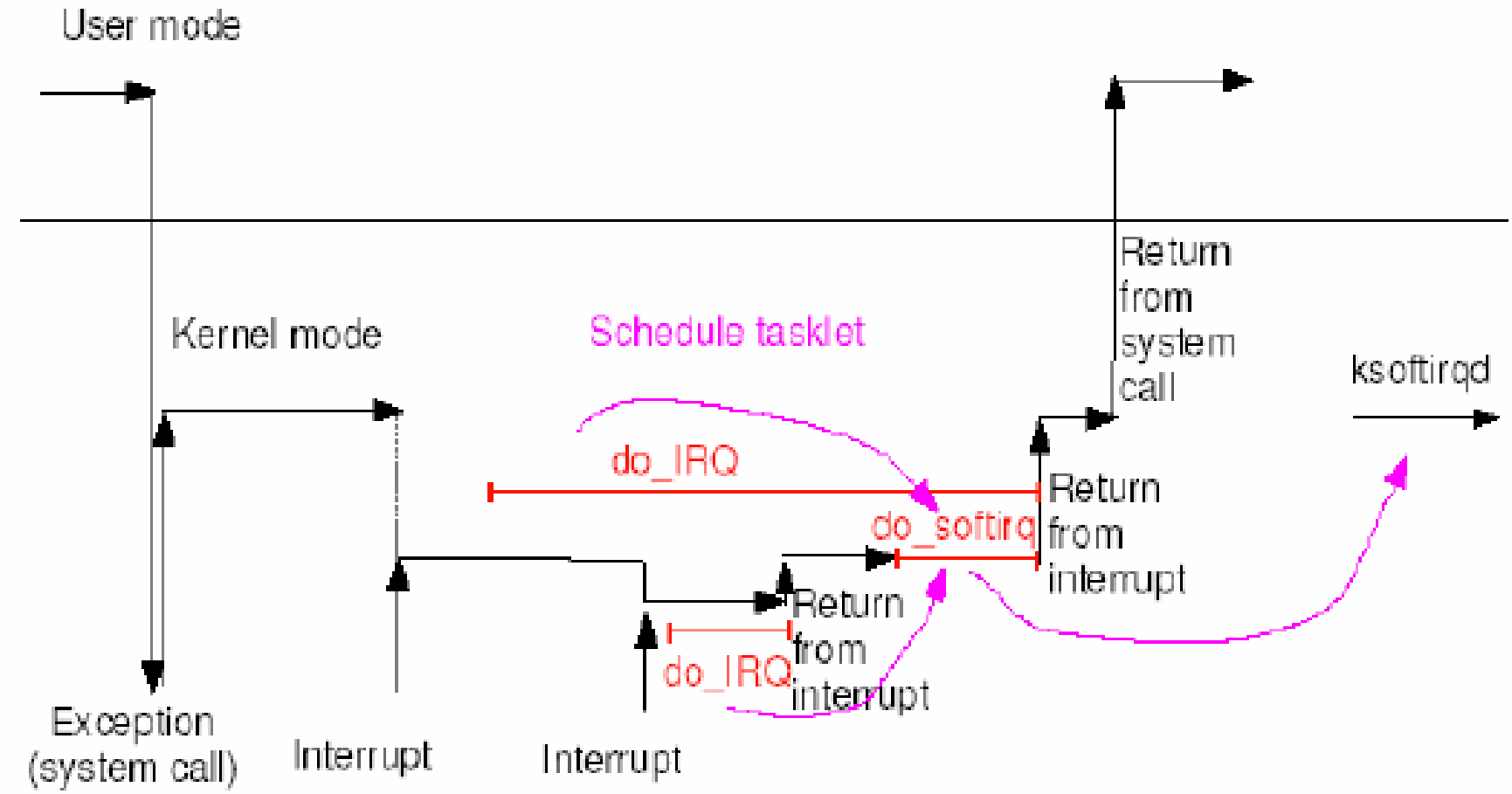
用自陷指令实现系统调用

- 一条产生异常的机器指令，导致处理器状态从用户态切换到核心态
  - 也称为：陷入指令，访管指令
- UNIX中的Trap指令，硬件完成以下步骤
  - PS内容压入现场栈
  - PC内容压入现场栈
  - 从中断向量034单元中取出内容装入PS
  - 从中断向量036单元中取出内容装入PC

系统调用的实现


main()
int fd;
int bytes;
fd=open("file", O_RDWR);
if (fd == -1) {
perror("open");
exit(-1);
} else {
bytes=read(fd, buf, ...);
}
}
user mode
system call
execution flow
kernel
trap into kernel trap table
enter syscall trap handler->
save the cpu struct address
save the return address in %l0
increment cpu_sysinfo.syscall stat
set up the arguments in the LWP regs
check flag for syscall preprocessing (t_pre_sys)
if yes - do preprocessing (syscall_pre)
otherwise, get syscall number from t_sysnum
index into sysent table for syscall
call it!
open(.....)
open return
any signals posted?
any post syscall handling?
(t_post_sys)
restore nPC
set return value from system call
back to user land

Linux中的系统调用和中断处理



主要系统调用实例

- 进程管理:
  - 创建进程　pid=fork()
  - 终止进程　exit(status)
  - 等待子进程结束　pid=waitpid(pid,…)
  - 替换进程映像　s=execve(name,…)
- 存储管理
  - 动态申请　malloc()
  - 释放存储空间　free()

主要系统调用实例

- 文件管理:
  - 创建文件 fd=creat(name,…)
  - 打开文件 fd=open(name,…)
  - 读文件 n=read(fd,buffer,nbyte)
  - 写文件 n=write(fd,buffer,nbyte)
  - 移动文件指针 pos=lseek(fd,offset,…)
  - 关闭文件 s=close(fd)
- 其它
  - 设置/获得时间等

Windows和Linux的系统调用

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

操作系统运行环境

- 计算机层次结构
- 中央处理器(CPU)
- 存储系统
- 中断与异常
- 操作系统运行模型
- 系统调用
- 人机界面

命令语言

- 通讯语言/作业控制语言
  - 控制作业流程的用户界面
  - 由语言解释器解释执行命令
- UNIX启动一命令解释程序的过程
  - 系统启动时1号进程为每个终端生成一个tty进程，让其运行登录程序
  - 用户输入ID及口令，验证完用户后，转去执行shell命令解释器
  - 由解释器处理用户输入命令

命令语言

- 命令解释程序shell的工作
  - 通过发“从终端读”系统调用接收输入
  - 直接处理一些控制语句，简单命令
  - 对不识别的命令关键字，到PATH环境变量所指目录中找到执行代码文件
  - 创建子进程去运行该程序.（如果命令关键字代表一个script程序文件，则产生子进程去执行该文件头行中说明的解释器，并解释执行该文件中的语句）
  - 等子进程结束后取下一输入命令

命令语言

- 系统主要的实用程序
  - 编辑器
    - 供用户建立和修改文本文件，提供一组内部编辑命令
  - 编译器和装配器
    - 实现编译源程序、链接模块、装配目标程序等功能
  - 文件及文件系统相关的实用程序
    - 文件的拷贝，打印，文件系统装卸等实用程序
  - 显示系统进程，资源状态的实用程序
  - 用户管理
    - 用户加入删除，口令修改

命令语言

- 用户命令与系统调用的关系
  - 用户命令对应的实用程序(包括命令解释器本身)在用户态运行，
  - 而系统调用处理程序在核心态运行
  - 实用程序在运行过程中可能会调用系统调用
  - 用户命令是用户操作接口
  - 系统调用是用户与操作系统之间的编程接口

图形化的用户界面

- 用图符，菜单来替换命令行表示
- 参数则通过窗口
- 提示用户选择或输入
- 命令解释器变成了图形化的程序管理器
- 应用程序都提供图形使用界面

小结

- 计算机层次结构
  - 操作系统是软件和硬件之间的接口
- 中央处理器(CPU)
  - 处理器执行指令的过程
  - OS利用特权指令控制计算机（特权级和非特权级）
  - 程序状态字（PSW）记录处理器状态
- 存储系统
  - 空间与时间的权衡：存储访问的局部性原理
  - 安全：存储保护问题
  - 虚拟内存空间 Vs. 物理内存空间

小结

- 中断与异常
  - 中断是实现多道程序的必要条件
  - 中断处理过程包含着哲理和处理技巧
  - 中断是理解OS核心机制的关键
- 操作系统运行模型
  - 从宏观上把握OS的核心设计思想
  - 关键问题：用户栈和内核栈的布局
    - 栈是 OS中标识 执行线索（context） 的实体
- 系统调用
  - 深入分析一个系统调用的处理过程，是理解OS的有效途径
- 人机界面
  - 学会使用Linux

课后阅读与思考

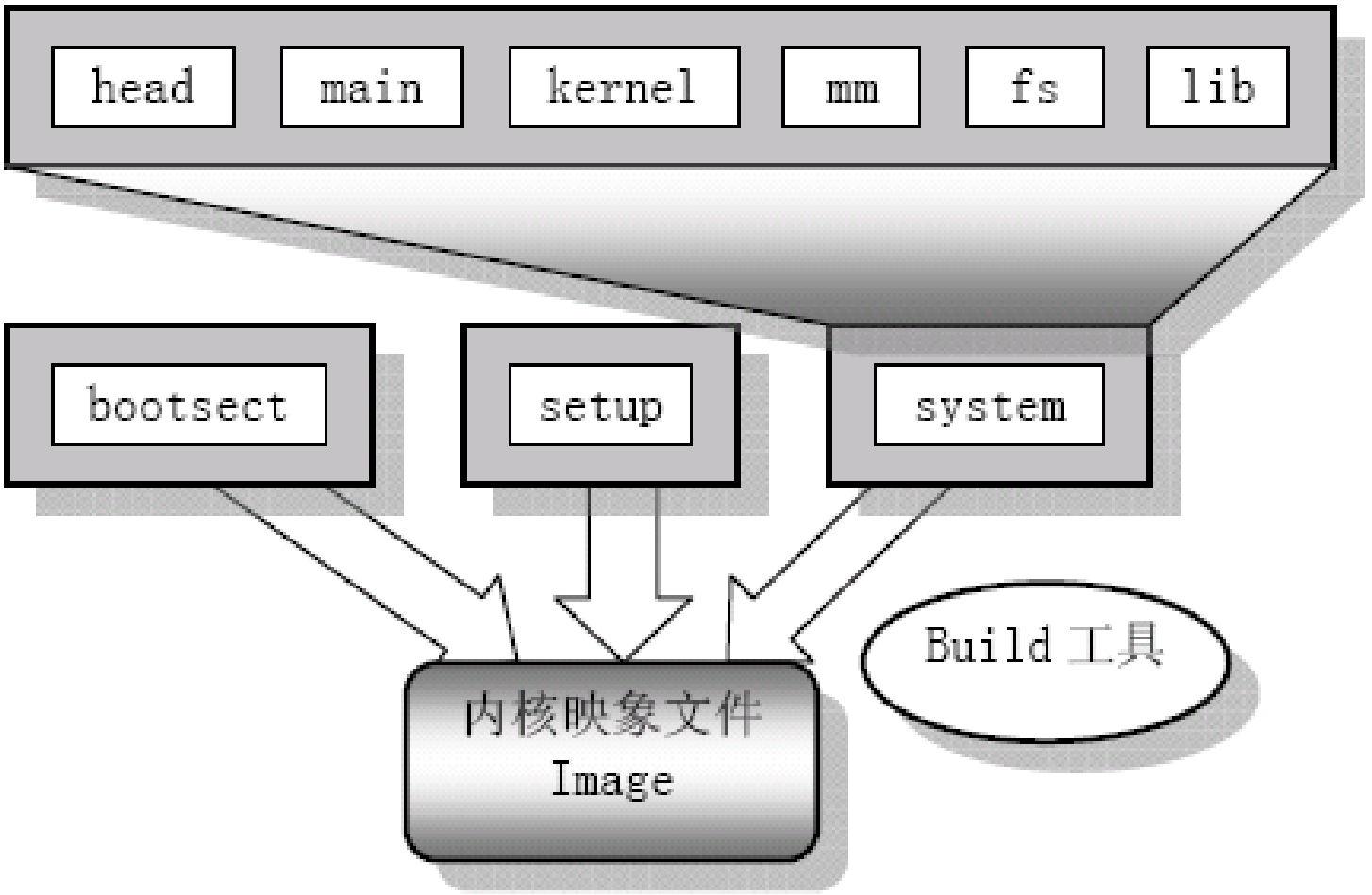
- 教材
  - 第2章
- Operating System Concepts (6th edition)
  - Chapter 2 Computer System Structure
  - Chapter 3 Operating System Structure
- Modern Operating System (2nd edition)
  - Section 1.4,1.5,1.6,1.7

课后作业

- 使用思维导图软件（如FreeMind），以“操作系统的运行环境”为主题，制作思维导图
- 要求
  - 一级节点与教材上的一级小节题目对应
  - 沿每个分支上按照主题细化出小节点
  - 知识点选取原则
    - 围绕每个叶子节点的主题
    - 关键知识点
    - 重要的内容，细化成小节点
  - 知识点来源
    - 课堂、教材
    - 自学的知识

- OS内核启动过程

Linux内核的可执行映像



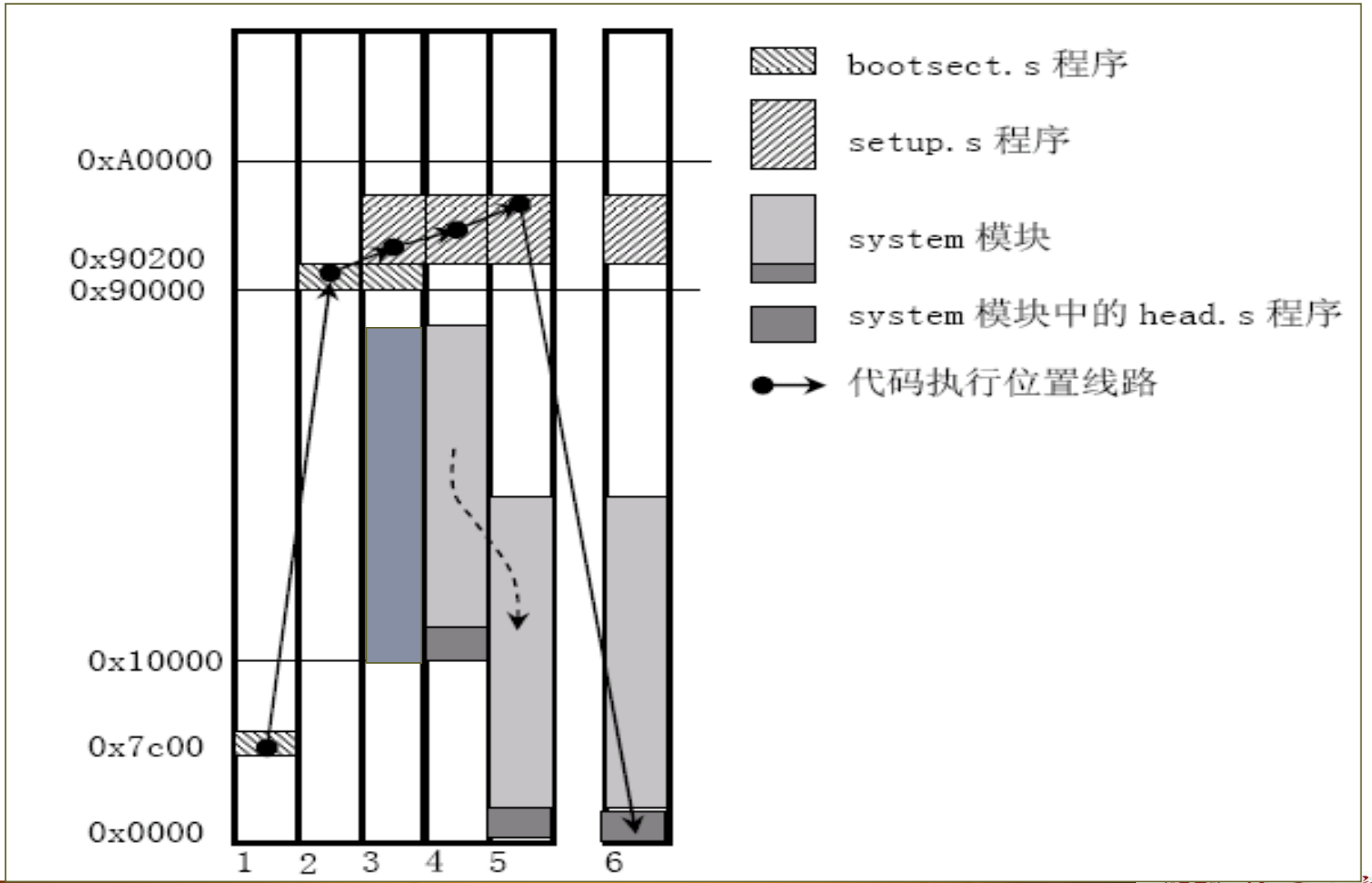
Linux的启动流程

- 开机：
  - 实模式：0xffff0（ROM-BIOS）开始，自动执行程序
  - BIOS：系统检测，初始化中断向量，读入启动设备的第一个磁盘引导扇区，到0x7C00处
- Boot/bootsect.s
- Boot/setup.s
- system模块

Bootsects功能

- 将自己搬到0x90000处
- 从磁盘把setup.s 加载到0x90200处
- 从磁盘把system加载到0x10000处
- 跳转到setup

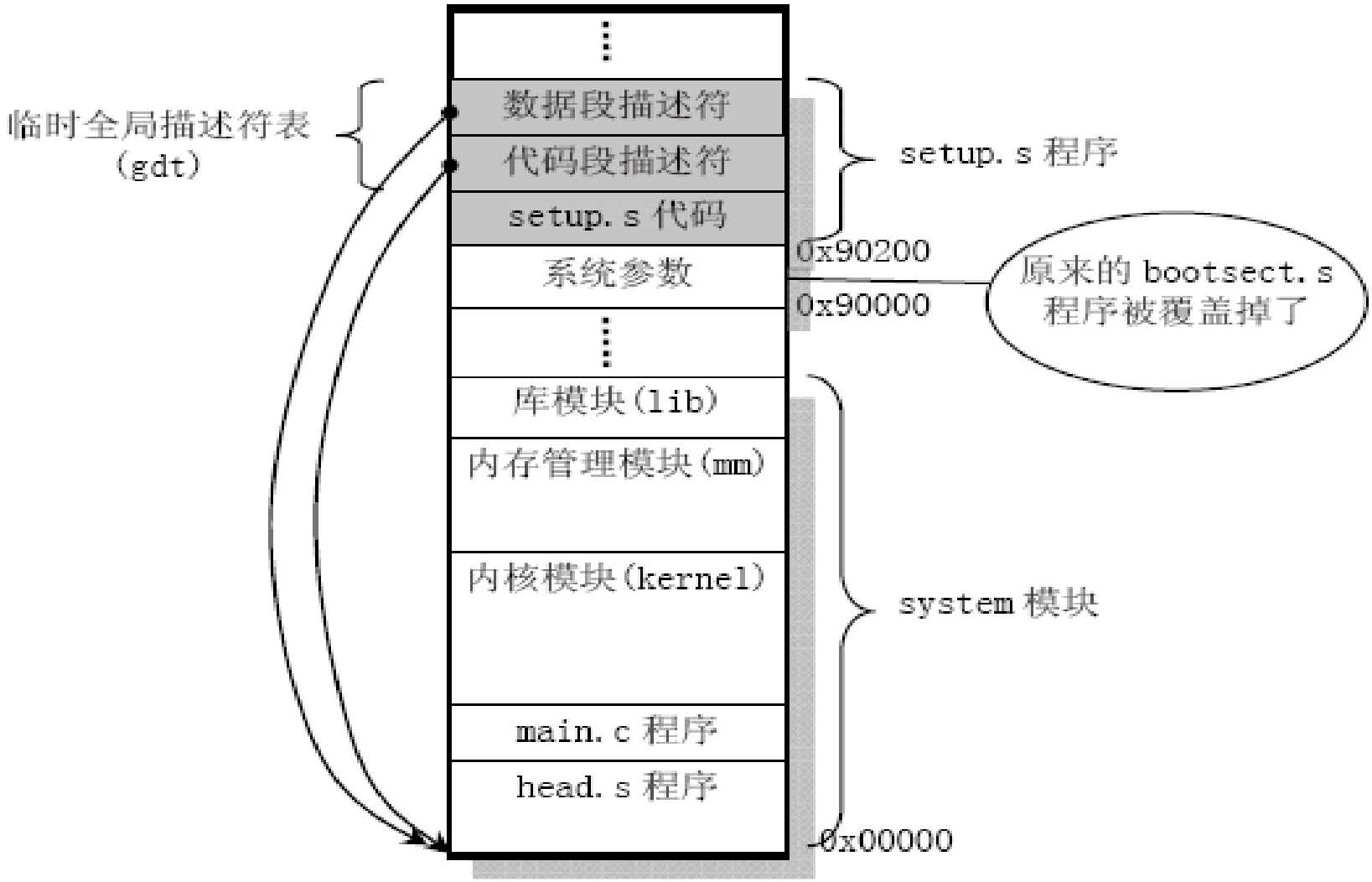
内核代码在内存中的位置及执行过程



Setup功能

- 读取计算机系统数据
  - 利用ROM BIOS 中断读取计算机系统数据，保存到0x90000开始的位置
- 把system从原来位置移到0x0000处
- 加载Idtr,gdtr ，开启A20 地址线，设置中断控制器8259A，硬件中断改为0x20-0x2f
- 设置CPU的控制寄存器CR0，开启32位保护模式
- 跳转到system的head.s

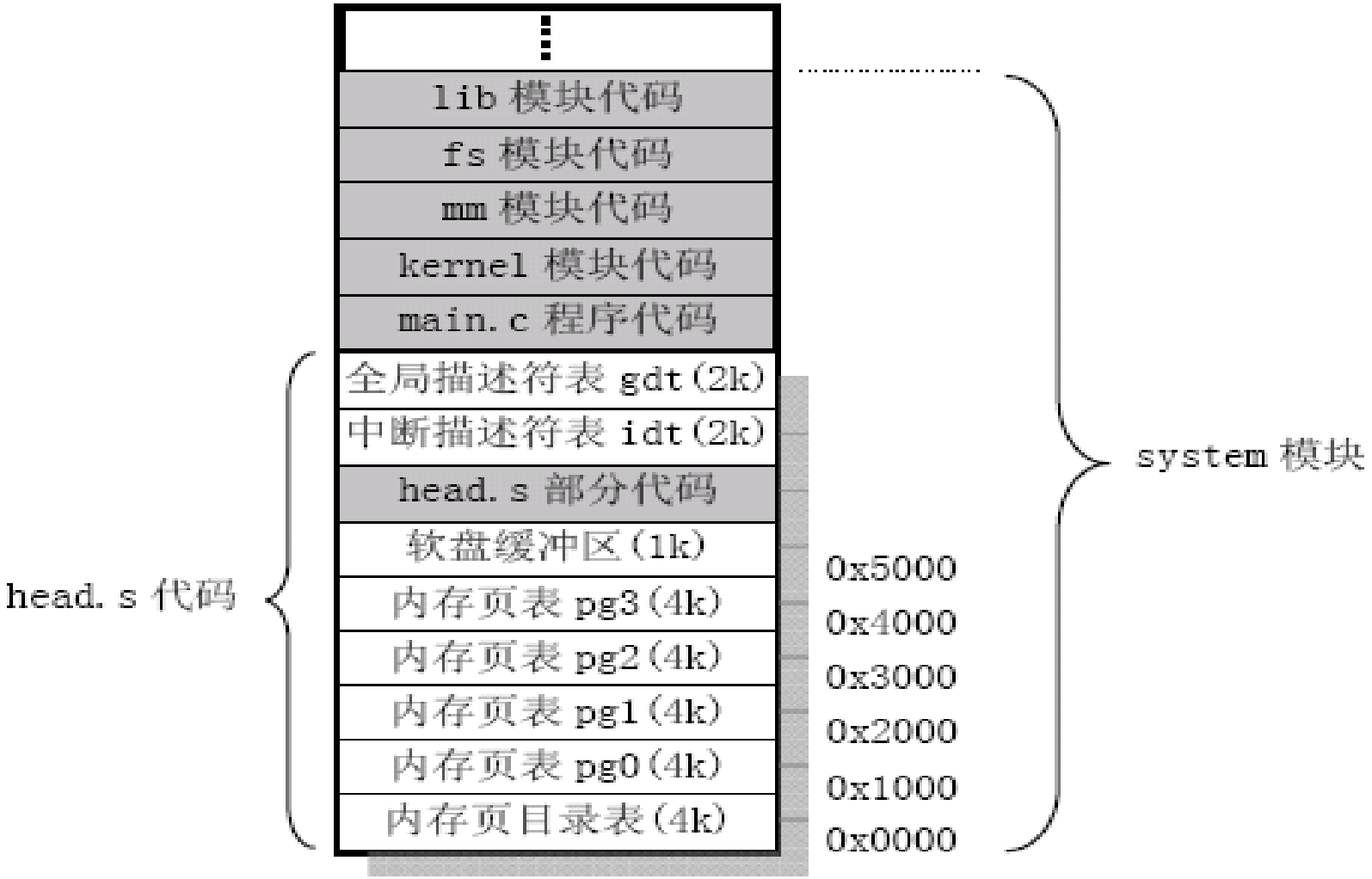
Setup执行后的内存



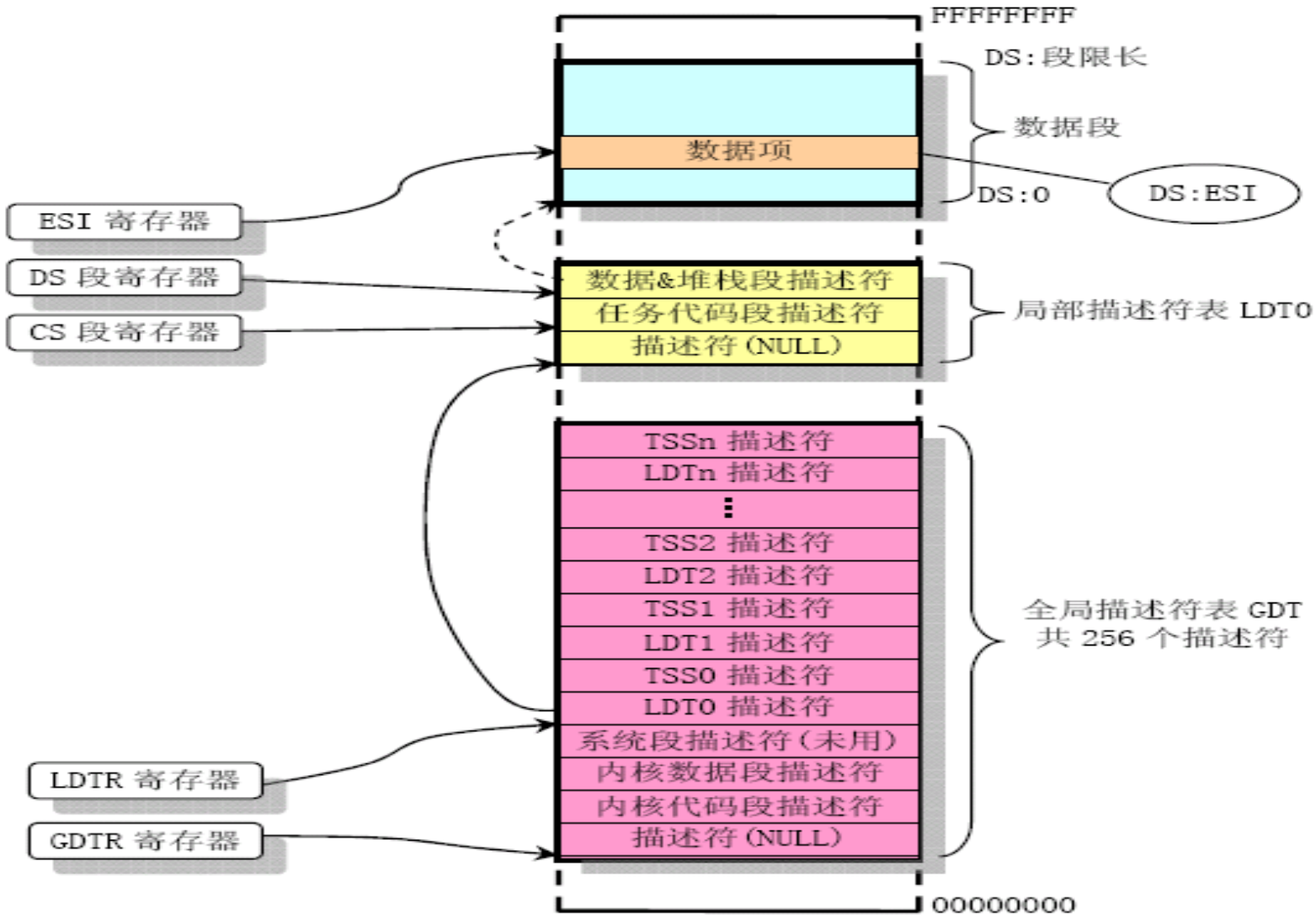
Head.s 功能

- 从绝对地址0x0 开始执行
- 加载各个数据段寄存器，重新设置idt，gdt
- 检测
  - A20 线是否开启
  - 检测协处理器，置CR0 相应位
- 设置分页机制
  - 页目录放在0x0开始的地方
  - 设置其中的4个页表，寻址16M
- 用返回指令，利用堆栈执行main（）程序

Head.s 之后的内存

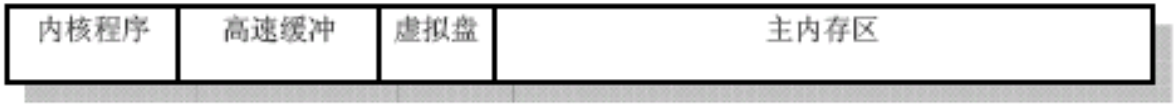

lib 模块代码
fs 模块代码
mm 模块代码
kernel 模块代码
main.c 程序代码
全局描述符表 gdt(2k)
中断描述符表 idt(2k)
head.s 部分代码
软盘缓冲区(1k)
内存页表 pg3(4k)
内存页表 pg2(4k)
内存页表 pg1(4k)
内存页表 pg0(4k)
内存页目录表(4k)
0x5000
0x4000
0x3000
0x2000
0x1000
0x0000
head.s 代码
system 模块

Linux 0.11的描述符表



系统初始化（main.c）

- 设置系统的根文件设备号以及一些内存全局变量
  - 主存开始地址、系统内存容量、设备缓冲区地址、虚拟盘地址



- 硬件初始化
  - 陷阱门、块设备、字符设备、tty
  - Task0
  - 开中断
- 切换到用户态，调用fork创建第一个用户进程，执行init()
- Init中运行控制台程序，创建子进程，运行shell

课后思考问题

- 在单道操作系统上，操作系统怎么运行？
- 如果没有“进程”，OS和应用程序怎么运行？
- 在一个程序里，怎么表示一个概念（对象）？
  - 定义一个数据结构
  - 定义一些操作
  - 那么，OS中的“进程”该是一个多么复杂的数据结构和操作的集合呢？
- 建议：
  - 先理解进程的概念和原理
  - 再考虑如何在操作系统中实现进程的概念