

操作系统

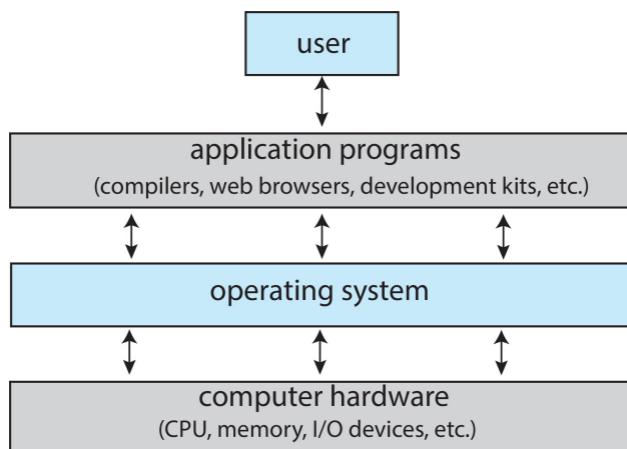
未经允许，禁止转载

基本原理部分

一、操作系统概论

1.1 操作系统的定义

1.1.1 计算机系统的组成



1.1.2 操作系统的作用

- 用户角度
 - 作为用户与计算机硬件系统之间的接口
 - 设计目标：有效性、方便性
- 计算机系统设计者角度
 - 作为计算机资源的管理者
 - 设计目标：提高资源的利用率

1.1.3 操作系统的定义

- 操作系统是计算机系统中的一个系统软件，是一些程序模块的集合
- Operating System = the kernel + system program

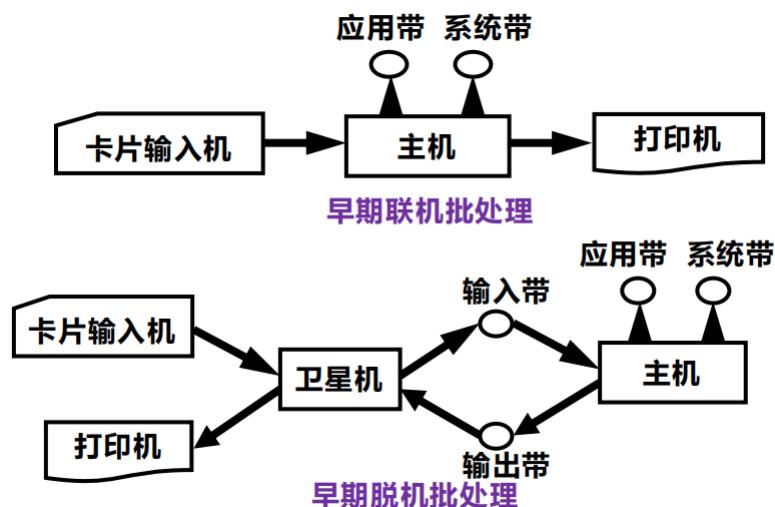
1.2 操作系统的形成与发展

1.2.1 无操作系统的计算机系统

- 程序员直接使用计算机硬件系统
- 单用户独占全机
- CPU等待人工操作（闲着）
 - 人工负责输入输出
 - 人工负责计算机的调度
 - 人工负责编排作业的运行顺序

1.2.2 单道批处理操作系统

- 以常驻内存的监控程序代替人工调度和作业编排，减少人工干预和等待时间，加快作业运行速度
- 硬件结构
 - 早期联机批处理
 - 作业的输入、计算、输出都在CPU控制下进行，CPU利用率低
 - 早期脱机批处理
 - 小型卫星机控制外部设备的输入输出（CPU只负责计算）



- 优点
 - 系统自动化程度高，吞吐量大，资源利用率高
 - 减少了CPU的空闲时间
 - 提高了I/O速度
- 缺点
 - CPU与外设串行（CPU仍然是要等待的）

- 作业周转时间长，用户无法实现对作业的控制

1.2.3 多道机处理操作系统

- 多道程序设计
 - 硬件支持：通道、中断、缓冲技术
 - 引入目的：CPU与外设并行执行，提高CPU利用率
 - 方法：在内存中同时存放若干道程序，使之在系统中同时处于交叉运行状态
 - | 程序道数不是无上限的
- 特点
 - 内存多道
 - 宏观上并行（不同的作业分别在CPU和外设上执行）
 - 微观上串行（在CPU上交叉执行）
- 目的
 - 提高CPU的利用率
 - 充分发挥系统设备的并行性
 - 程序之间
 - CPU与设备之间
 - 设备与设备之间
- 衡量批处理系统的性能指标
 - 资源利用率
 - 给定时间内系统中某一资源实际使用时间所占比率
 - 吞吐量
 - 单位时间内系统所处理的信息量，通常以每小时或每天所处理的作业个数进行度量
 - 周转时间
 - 从作业进入系统到退出系统所用时间
 - 平均周转时间
 - 系统运行的多个作业的周转时间的平均值
- 多道批处理系统的特点
 - 提高了系统的资源利用率
 - 提高了系统的吞吐量
 - 用户与作业之间无法交互
 - 作业平均周转时间较长
 - 适合处理计算量大的成熟的作业
- 典型系统
 - IBM OS/360

1.2.4 分时与多任务操作系统

- 批处理系统的缺点
 - 用户不能直接控制作业运行
 - 作业周转时间太长
- 产生分时系统的原因
 - 用户需求（交互+响应）

- 分时概念

多个用户分时使用CPU时间，即将CPU的单位时间划分为若干时间段（每个时间段称为一个时间片），各个用户按时间片轮流占用CPU

- 分时系统特点

- 同时性
- 独立性
- 交互性
- 及时性
 - 响应时间为2-3秒
 - 响应时间是衡量分时系统的主要指标
 - 影响响应时间的因素
 - 用户数目
 - 时间片

- 典型系统

- 第一个分时系统CTSS
- 最重要的分时系统
 - MULTICS
- UNIX/LINUX
- DOS/Windows
- macOS

- 批处理系统与分时系统比较

批处理系统

- 处理对象为作业
- 目标是提高系统资源利用率
- 适用于比较成熟大型作业
- 可在后台执行，不需要用户频繁干预

分时系统

- 处理对象为作业
- 目标是对用户请求快速响应
- 适用于短小作业
- 终端键入命令

1.2.5 实时系统

- 实时概念

- 系统对外部特定输入信号的响应时间足以控制发出实时信号的设备

- 实时系统分类

- 应用需求
 - 实时控制系统/实时信息处理系统
- 实时任务的截止时间
 - 硬实时/软实时

- 实时系统的特征

- 实时性
- 可靠性

- 可确定性
- 实时系统适合与一些不强调资源利用效率的专用系统
- 实时系统以数据或信息作为处理对象

1.2.6 嵌入式操作系统

- 以应用为中心、以计算机技术为基础、软硬件可裁剪的专用计算机系统
 - 基本都是实时操作系统（不全是），只有满足实际需求的有限功能
 - 特点
 - 软件要求固化存储
 - 高质量、高可靠性、高实时性
 - 典型系统
- Linux, FreeRTOS

1.2.7 智能移动终端操作系统

- 智能移动终端操作系统
 - 在受限的计算能力上提供丰富的用户体验
 - 在受限的供电能力上改善系统的能源效率
 - 在存储大量用户敏感信息场景下保证用户信息的隐私和安全
- 典型
 - Android, iOS

1.2.8 分布式系统

- 定义
 - A collection of independent computers appears to its users as a single coherent system
 - the one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages

1.2.9 操作系统的发展趋势

- 从封闭到开放、再到封闭
- 从专用到通用、再到专用（领域定制）
- 从简单到复杂

1.3 操作系统的基本特性

- 并发性

并发性：两个或多个事件在同一时间间隔内发生，它们都已经被启动执行，而且都还没有完成执行

并行性：两个或多个事件在同一时刻发生

- 共享性
 - 互斥共享
- 虚拟性
- 异步性

1.4 操作系统的主要功能

- 处理机管理
 - 进程控制/进程同步/进程通信/进程调度
- 存储器管理
 - 内存分配/内存保护/地址映射/内存扩充
- 文件管理
 - 文件存储空间的管理/目录管理/文件读写管理和保护
- 设备管理
 - 缓冲管理/设备分配

1.5 操作系统提供的服务

- 用户接口
 - 命令级接口
 - 程序级接口
- 执行程序
- I/O调用
- 文件系统操作
- 通信服务
- 错误检测和处理
- 资源分配
- 登录和记账
- 保护和安全

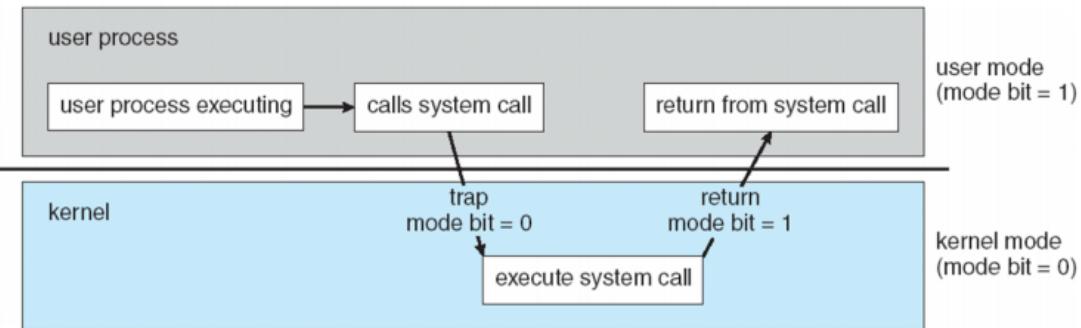
1.6 操作系统的用户接口

- 操作接口
 - 命令解释程序（命令行）
 - 两种形式
 1. 操作系统内核一部分（DOS）
 2. 特殊程序（Unix的Shell）
 - 作用：执行命令
 - 两种实现方式
 - 命令解释程序执行（DOS）
 - 系统程序实现（Unix的Shell）
 - GUI
 - 具有窗口界面的解释程序
 - 触屏界面
 - 手势、声音指令
- 编程接口
 - 系统调用
 - 操作系统内提供的一些子程序
 - 请求操作系统服务或资源
 - 常以API形式出现
 - 应用领域接口
 - Android/iOS应用接口
 - 汽车领域接口AUTOSAR

1.7 操作系统的运行方式

- 双重模式操作
 - 内核模式
 - 允许执行全部指令
 - 允许访问所有的寄存器和缓冲区
 - 用户模式
 - 只能执行非特权指令
 - 只能访问指定的寄存器和存储区
- 特权指令
 - 能引起损害的机器指令
 - 实例
 - I/O控制
 - 访问程序状态（PSW）
 - 存取和操作CPU状态
 - 启动各种外部设备
 - 设置时钟时间
 - 关中断
 - 清主存
 - 修改存储器

- 管理寄存器
- 改变用户方式到核心方式
- 停机指令
- 计算机硬件提供模式位



- 模式转换
 - 用户模式 >> 内核模式
 - 中断
 - 异常
 - 系统调用
 - 内核模式 >> 用户模式
 - 完成中断或异常的处理
 - 新进程或线程启动
 - 进程调度选择执行新的进程
 - 操作系统向进程发送信号

1.8 操作系统的结构

1.8.1 操作系统的机制和策略

- 目的
 - 降低操作系统设计的复杂性
- 设计原则：策略与机制分离
 - 策略：做什么
 - 机制：怎么做
- 操作系统可仅通过调整策略来适应不同应用的需求

1.8.2 操作系统复杂度管理方法

- 模块化
 - 分而治之
- 抽象
 - 接口与内部实现分离
- 分层
 - 按层次划分，减少模块之间的交互
- 层级

- 子系统递归形成大系统

1.8.3 操作系统的内核架构

- 简要结构
 - 功能简单的操作系统、应用程序和操作系统放置在同一个地址空间。无须底层硬件提供复杂的内存管理、特权级管理等功能
 - 模块化、层次化
 - 将操作系统按其功能划分为若干个具有一定独立性和大小的模块和子模块
 - 各模块间通过接口实现交互
 - 模块间调用关系无序
 - 模块间耦合紧密
- 宏内核架构
 - 操作系统内核的所有模块均运行在内核态（单内核），具备直接操作硬件的能力
 - 可加载内核模块LKM
 - 优点
 - 模块化
 - 易于实现
 - 增加可靠性
- 微内核架构
 - 特点
 - 操作系统从内核转移到“用户”空间
 - 以客户/服务器模式为基础
 - IPC进程间通信
 - 基本功能
 - 进程管理
 - 存储器管理
 - 进程通信管理
 - I/O设备管理
 - 实现方式
 - 核中只保留最基本的、最本质的操作系统功能
 - 用户空间包含所有的OS进程和用户应用进程
 - 每一OS功能均以单独的服务器进程的形式存在
 - 优点
 - 策略与机制进一步分离
 - 易于扩展
 - 易于移植
 - 服务与服务之间完全隔离
 - 可靠性
 - 安全性
 - 缺点
 - 性能（进程间通信开销大）
- 外核架构
 - 引入
 - 操作系统内核对硬件资源进行了抽象

- 过度抽象带来性能损失
- 通用抽象对具体应用往往不是最优选择
- 解决方案
 - 不提供硬件抽象
 - 不管理资源，只管理应用
- 多内核架构
- 混合内核架构
 - 宏内核 + 微内核
 - 将需要性能的模块重新放回内核态
- 微内核和外核的主要区别：微内核对硬件进行抽象，外核不提供硬件抽象，而是对应用进行抽象。
 - 微内核
 - 优势：策略与机制进一步分离，使得操作系统易于移植、易于扩展；服务与服务之间完全隔离，带来更高可靠性和安全性。
 - 劣势：性能不高
 - 外核
 - 优势：OS无抽象，能够在理论上提供最高性能；应用对计算有更精确的实时控制；LibOS在用户态更易调试
 - 劣势：对计算资源的利用效率主要由应用决定；定制化过多，维护难度增加；缺乏跨场景的通用性，应用生态差。
- 微内核较之其他操作系统结构性能较差的原因是大量系统服务放在用户空间，消息传递增加了系统开销。如果硬件性能能够提升，那么微内核的性能也能够有所提升。所以微内核性能的瓶颈在于硬件性能

二、进程、线程与调度

2.1 进程

2.1.1 进程的引入和概念

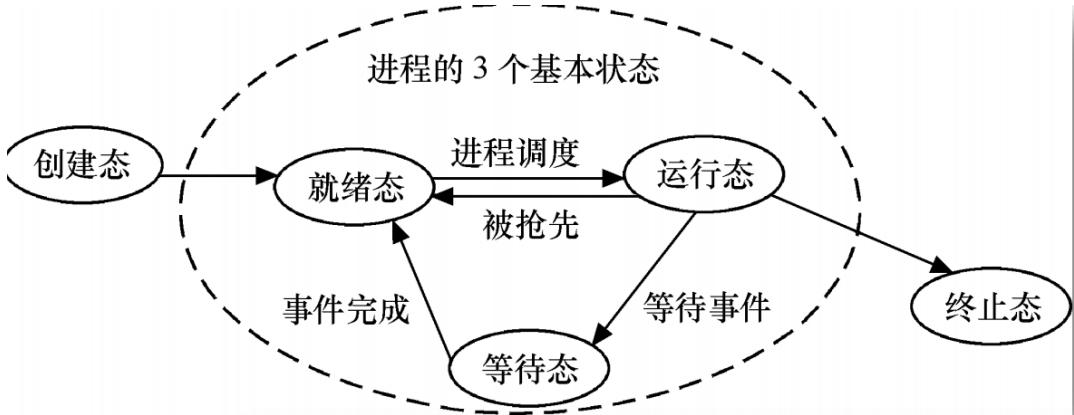
- 程序的顺序执行
 - 程序顺序执行的特点
 - 程序执行连续性
 - 程序环境封闭性
 - 程序结果可再现性
 - 优点
 - 调试方便
 - 缺点
 - 资源的独占性使得资源利用率非常低
- 程序的并发执行
 - 程序并发执行的特点
 - 增强了计算机系统的处理能力，提高了资源利用率

- 程序执行的间断性：并发执行的程序间产生了相互制约的关系
 - 1. 间接制约：竞争资源
 - 2. 直接制约：执行同一任务
- 缺点
 - 失去了程序的封闭性
 - 资源共享
 - 每次执行环境都不一样
 - 程序结果的不可再现性
- 程序和CPU执行活动不是一一对应的
 - 程序是静态的
 - CPU执行活动是动态的
- 进程的基本概念
 - 进程的定义
 - 进程是程序在一个数据集合上的运行过程，它是系统进行资源分配和调度的一个独立单位
 - 进程的特性
 - 动态性：进程是程序的一次执行，具有生命期
 - 独立性：进程是系统进行资源分配和调度的一个独立单位
 - 并发性：进程可以并发执行
 - 异步性：进程间的相互制约，使进程执行具有间隙
 - 结构性：进程具有结构
 - 进程的组成
 - 程序
 - 数据
 - 进程控制块（PCB）

2.1.2 进程的描述

- 进程的状态
 - 三种基本状态
 - 运行态
正在计算单元上运行进程所处状态
 - 就绪态
已经获得了除CPU core之外的全部资源并等待系统分配CPU core，一旦获得CPU core即可变为运行态的进程状态
(拥有除CPU core之外的全部资源)
 - 阻塞态/等候态
一个进程因等待某事件发生而不能运行时所处状态
(拥有除CPU core之外的部分其他资源)
 - 首尾状态
 - 创建态
 - 终止态

- 状态切换



- 运行态 → 就绪态
- 运行态 → 等待态
- 就绪态 → 运行态
- 等待态 → 就绪态
- 创建态 → 就绪态
 - 操作系统准备好再接纳一个进程时，把一个进程从创建态变为就绪态。
- 运行态 → 终止态
 - 进程已结束，但尚未撤消，以便其它进程去收集该进程的有关信息。

- 进程控制块PCB

- 进程存在的唯一标识
- 包含进程的描述信息和控制管理信息
 - 进程标识符：内部/外部
 - 进程当前状态
 - 进程调度信息
 - 进程优先级
 - 进程所在各种队列的指针
 - 进程存储管理信息
 - 所执行程序的内外存起始地址以及所采取的保护信息
 - 进程使用的资源信息
 - CPU现场保护区
 - 进程之间的家族关系

示例

```
//Linux中的PCB中的部分信息
struct task_struct {
    pid_t pid; //进程标识符(32767)
    struct thread_info *thread_info; //当前进程基本信息
    long state; //进程状态
    int prio; //进程优先级0-139
    int static_prio; //进程的静态优先级
    unsigned long rt_priority; //进程的实时优先级
    struct list_head tasks; //进程链表
    struct task_struct *parent; //指向父进程
    struct list_head children; //子进程链表
    struct files_struct *files; //打开文件列表
    struct mm_struct *mm, *active_mm; //指向进程拥有和执行
                                    //的虚拟内存描述符
    .....
};
```

- 进程的上下文切换
 - 进程的上下文包括进程运行时的寄存器状态，能够用于保存和回复一个进程在处理器上运行的状态
 - 当操作系统需要切换当前执行进程时，使用上下文切换机制
 - 上下文保存在对应的PCB中
 - 进程被调度时，从PCB中取出上下文并恢复；运行现场被恢复，将CPU控制权交给被选中进程
 - 上下文切换时，进程通过中断或系统调用进入内核
- 进程的内存空间布局
 - 进程具有独立的虚拟地址空间
 - 用户栈
 - 代码库
 - 用户堆
 - 数据与代码段
 - 内核部分
 - 内核栈
 - 内核代码与数据
- 进程的组织
 - 方式
 - 线性表
 - 链表
 - 哈希链表
 - 进程组与会话
 - 进程组
 - 进程的集合，由多个或一个进程组成
 - 父子进程默认属于一个进程组
 - 会话
 - 进程组的集合，由一个或多个进程组组成
 - 根据执行状态，分为前台进程组和后台进程组

2.1.3 进程的控制

- 进程控制
 - 系统使用一些具有特定功能的程序段来创建、撤销进程以及完成进程各状态间一系列转换等有效管理
 - 一般由操作系统完成
- 原语
 - 某些程序段的执行过程不允许中断
- 创建原语
 - 创建进程时机
 - 批处理系统中为每个作业创建一个进程
 - 分时系统中为每个用户创建一个进程
 - 提供服务：打印进程
 - 应用请求：已存在的进程创建子进程
 - 功能
 - 申请空白PCB
 - 为新进程分配资源
 - 初始化PCB
 - 将新进程插入就绪队列
- 撤销原语
 - 撤销进程时机
 - 进程完成任务，正常结束
 - 由于故障不能继续执行，异常结束
 - 外界干预
 - 功能
 - 根据被终止进程标识符查找被撤销的进程PCB
 - 终止该进程，重新调度
 - 终止该进程的全部子进程
 - 回收资源
 - 释放PCB
- 阻塞原语
 - 阻塞进程时机
 - 处于运行状态的进程等待某一事件发生
 - 等待I/O数据传输完成
 - 等待其他进程发送信息
 - 功能
 - 处于运行态的进程中断CPU，将其运行现场保存在PCB的CPU现场保护区
 - 将其状态置为阻塞态，并插入相应事件的等待队列
 - 转进程调度，选择一个就绪进程投入运行
 - 阻塞原语由进程自己执行
- 唤醒原语
 - 唤醒进程时机
 - 进程期待的事件到来
 - 功能

- 期待的事件是等待输入输出完成

输入/出完成后，由硬件提出中断请求，CPU响应中断，暂停当前进程的执行，转去中断处理，检查有无等待该输入/输出完成的进程

有则将该进程从等待队列抽出，并将其由阻塞态置为就绪态，插入就绪队列，结束中断处理

2.2 线程的引入与概念

2.2.1 线程的引入

- 原因
 - 应用中同时发生许多活动
 - 硬件发展，多CPU核，并行度提高
 - 进程创建开销大、资源
 - 进程享有独立的虚拟地址空间，进程间数据共享和同步的开销大
- 解决方法
 - 将拥有资源的基本单位与调度的基本单位分离

2.2.2 线程的概念

- 线程的概念
 - 进程内的一个可调度实体
- 线程的特性
 - 独立调度和分派的基本单位
 - 可并发（并行）执行
 - 动态性
 - 结构性
 - 线程控制块
 - 所有线程具有相同的地址空间（进程地址空间），共享进程资源
- 共享资源
 - 1、进程申请的堆内存
 - 2、进程打开的文件描述符
 - 3、进程的全局数据(可用于线程之间通信)
 - 4、进程ID、进程组ID
 - 5、进程目录
 - 6、信号处理器
- 独占资源
 - 1、线程ID
同一进程中每个线程拥有唯一的线程ID。
 - 2、寄存器组的值

程上时，必须将原有的线程的寄存器集合的状态保存，以便将来该线程在被重新切换到时能得以恢复。

3、线程堆栈

线程可以进行函数调用，必然会使使用大函数堆栈。

4、错误返回码

线程执行出错时，必须明确是哪个线程出现何种错误，因此不同的线程应该拥有自己的错误返回码变量。

5、信号屏蔽码

由于每个线程所感兴趣的信号不同，所以线程的信号屏蔽码应该由线程自己管理。但所有的线程都共享同样的信号处理器。

6、线程的优先级

由于线程需要像进程那样能够被调度，那么就必须要有可供调度使用的参数，这个参数就是线程的优先级。

- 线程的状态

- 就绪态
- 运行态
- 阻塞态/等待态

- 线程的创建

- 创建进程时，系统同时为进程创建第一个线程，即“初始化线程”
- 由初始化线程根据需要再去创建若干个线程

- 线程的终止

- 线程在完成工作后自愿退出
- 线程在运行中出现错误或被其他线程强行终止

- 线程和进程的比较

- 拥有的资源
 - 每个进程拥有独立的存储空间
- 调度
 - 进行进程的上下文切换需要较大的系统空间和时间的开销
 - 由于同一进程内的线程共享进程的资源，其上下文切换只把线程仅有的一小部分资源交换即可

由一个进程的线程向另一个进程的线程切换的时候，将引起进程上下文切换

- 并发性

- 引入线程后，系统并发执行程度更高

- 安全性

- 由于同一进程的多线程共享进程的所有资源，一个错误的线程可以任意改变另一个线程使用的数据而导致错误发生

2.2.3 线程的实现

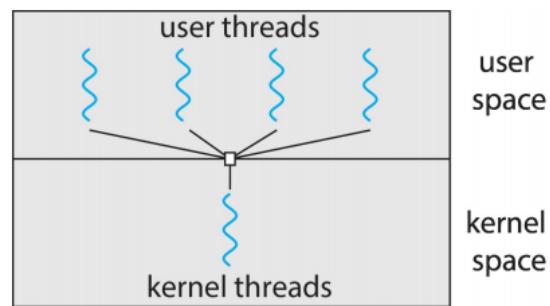
- 内核级线程

- 所有线程的创建、撤销、调度和管理都由OS依靠内核负责
- 在内核空间为每一个内核支持线程设置了一个线程控制块，内核根据线程控制块感知线程的存在并加以控制
- 缺点
 - 创建和管理的开销大

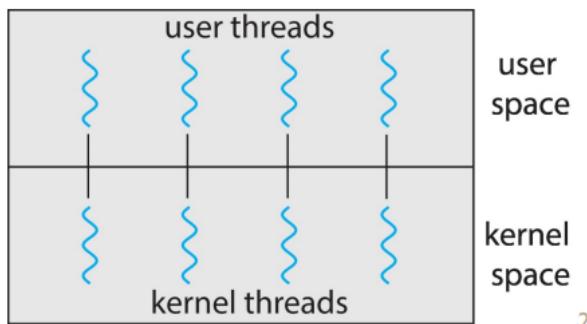
- 优点
 - 可调度一个进程的多个线程同时在多个计算单元上并行运行，从而提高程序运行的执行速度和效率
 - 当进程中的一个线程被阻塞时，进程中的其他线程仍可以被调度运行
 - 具有很小的数据结构和堆栈，线程切换较快，开销较小
- 实现
 - 创建进程时分配任务数据区空间
- 用户级线程
 - 所有的管理都在用户空间，与OS内核无关
 - 由于内核是单线程，处理机调度仍以进程为单位
 - 缺点
 - 一个线程的阻塞会导致进程其他
 - 只能有一个线程在CPU上运行，不能利用多核的好处
 - 优点
 - 线程切换开销较小
 - 线程调度算法与操作系统的调度算法无关
 - 不要求内核支持，可使用于任何操作系统

2.2.4 多线程模型

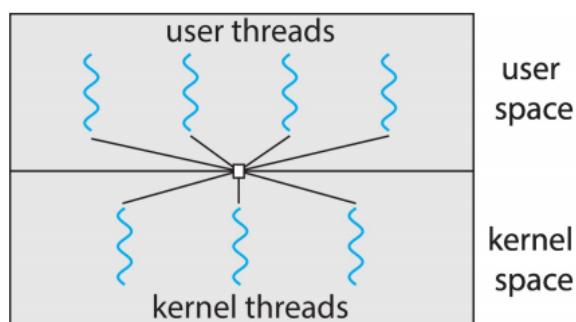
- 多对一



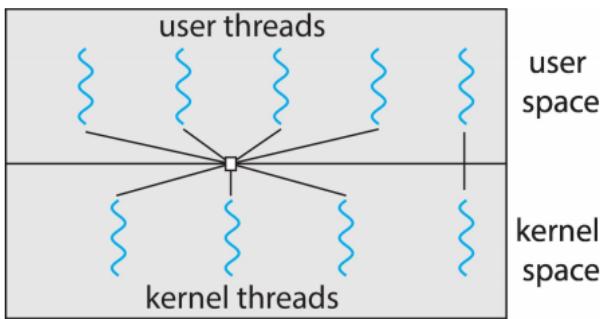
- 一对一



- 多对多



- 两级模型



2.2.5 线程库

- 为程序员提供创建和管理线程的API
- 实现方式
 - 在用户空间提供没有内核支持的库
 - 由操作系统直接支持的内核级的库

2.3 操作系统调度

2.3.1 基本概念

- 处理机调度级别
 - 高级调度（作业调度）
 - 低级调度
 - 中级调度
- 调度方式
 - 非剥夺方式
 - 实现简单

2.3.2 单核调度算法

- FCFS调度算法
 - 非剥夺
 - 既可用于作业调度，也可用于进程调度
 - 简单，但效率不高
 - 有利于长作业（进程），不利于短作业（进程），容易被垄断，使得平均等待时间很长
- SJF调度算法
 - 考虑作业（进程）运行时间
 - 在长期调度中频繁使用
 - 既可剥夺（最短剩余时间优先），也可非剥夺
 - 优点
 - 理论上最优算法，可以获得最小的平均等待时间，提高系统吞吐量
 - 缺点

- 对长作业不利，容易产生“饥饿”现象
- 高响应比优先调度算法
 - 优点
 - 兼顾了运行时间短和等待时间长的作业，优先运行短作业和等待时间足够长的长作业
 - 响应比=1+作业等待时间/作业估计运行时间
 - 缺点
 - 较复杂，系统开销大
- 优先级调度算法
 - 为每个进程设定一个优先级
 - 既可采用非剥夺（批处理），也可采用剥夺（实时）
 - 优先级设定
 - 进程类型：系统进程/用户进程
 - 进程对资源的需求
 - 申请资源较多的进程，优先级较低
 - 用户要求
 - 优先级与优先数
 - 优先级类型
 - 静态优先级
 - 进程创建时确定，整个运行期间保持不变
 - 不能反映进程特点，调度性能差
 - 容易导致“饥饿”，即不能调度低优先级进程
 - 动态优先级
 - 随着进程的推进或等待时间的增加而改变
- 时间片轮转调度算法
 - 用于分时系统
 - 剥夺
 - 时间片的确定
 - 既要保证系统各个用户进程及时地得到响应，又不要由于时间片太短而增加系统的开销，降低系统的效率
 - 时间片
 - 调度算法
 - 数据分析
 - 时间片用完后由运行态变为就绪态
- 多级队列调度算法
 - 综合考虑各种类型进程的需要
 - 终端型作业用户
 - 交互性好，响应时间短
 - 短批处理作业用户
 - 周转时间短，在较短时间内完成
 - 长批处理作业用户
 - 不会出现长期得不到处理的“饥饿”现象
 - 调度时考虑的问题

- 队列数目
- 每个队列采用的调度算法
- 初始状态时，每个进程处于哪个队列

2.3.3 实时系统调度

- 速率单调调度
 - 优先级：周期的倒数
- EDF调度算法
 - 优先级：截至时间
- LLF调度算法
 - 优先级：松弛度
 - 松弛度 = 结束时间 - 运行时间 - 当前时间
 - 松弛度为0的时候，必须抢占
 - 任务的紧急程度越高，为该任务赋予的优先级就越高

2.3.4 多核调度

- 多核调度
 - 任务同时在多个CPU核心上并行执行
 - 调度器的考虑
- 负载分担
- 协同调度
- 两级调度

三、进程通信与死锁

3.1 进程通信

3.1.1 并发进程的特点

- 资源共享引起的互斥关系
 - 并发进程互斥使用共享资源
 - eg. 两个进程均要独占打印机
 - 间接制约关系
 - 进程-资源-进程
- 协作完成同一个任务引起的同步关系

- 同步点
 - | eg. 生产者-消费者
- 直接制约关系
- 进程-进程

3.1.2 进程间通信

- 允许进程相互交换数据与信息
- 进程通信模式
 - 共享内存
 - 建立起一块供协作进程共享的内存区域，进程通过向此共享区域读或写入数据交换信息
 - 消息传递
 - 通过在协作进程间交换消息实现通信

3.2 进程同步

3.2.1 临界资源与临界区

- 临界资源
 - 一次仅允许一个进程使用的资源
- 临界区
 - 每个进程访问临界资源时必须互斥执行的程序
- 使用临界资源须遵循的原则
 - 互斥使用：不能同时有两个进程在临界区内执行
 - 有空让进：临界资源空闲时，应允许一个请求临界资源的进程进入临界区
 - 有限等待：不能使进入临界区的进程无限期地等待在临界区之外
 - 让权等待：等待进入临界区的资源应释放处理机后阻塞等待

3.2.2 Peterson算法

- 解决进程互斥进入临界区的软件方法
- 适用于**两个**进程交替执行临界区与剩余区
- 2个进程共享2个数据项
 - int turn;
 - 表示哪个进程可以进入临界区
 - boolean flag[2]
 - 表示哪个进程准备进入临界区

- 算法代码

```

// 进程Pi          // 进程Pj
void Pi() {        void Pj() {
    while(TURE) {    while(TURE) {
        flag[i]=TRUE;  flag[j]=TRUE;
        turn=j;        turn=i;
        while(flag[j]&&(turn==j));  while(flag[i]&&(turn==i));
        critical section  critical section
        flag[i]=FALSE;  flag[j]=FALSE;
        reminder section  reminder section
    }
}
}

```

- 退出while循环的条件是，要么另一个进程/线程不想使用临界区，要么此进程/线程拥有访问权限
- 问题
如果有两个无依赖变量，可能出现乱序执行

3.2.3 硬件同步

- 关中断
 - 进程刚进入临界区后，立即禁止所有中断；进程离开临界区之前打开所有中断
 - 原理：CPU只有在发生时钟中断或其他中断时才能进行进程切换
- Memory Barriers (内存屏障)
- 硬件指令
 - test and set()
 - compare and swap() (CAS)
- 原子变量

3.2.4 互斥锁

- 基于硬件的方法复杂且程序员无法访问
- 互斥锁是更高级的软件工具
- 进程间通过互斥锁实现互斥
- 原理
 - 为临界资源设置锁变量w
 - w=0 资源空闲可用
 - w=1 资源已被占用
 - 初始化及退出临界区，w置为0
 - 进入临界区，w置为1
- 自旋锁
 - 利用CAS
 - 优点
 - 没有上下文切换

3.2.5 信号量

- 信号量

- 进程同步工具
- 物理意义
 - 表示资源的实体
 - 意义与要表示的含义相关
- 结构

```
typedef struct {  
    int value;  
    struct process *L;  
} semaphore;
```

- value
 - 该类资源当前的可用数量
- L
 - 等待使用该类资源的阻塞进程队列的队首指针

- 操作

- 初始化
 - 将value初始化为该类资源的可用数量

value为负数时，绝对值表示在该信号量上等待的进程数目（等待队列）

- 原子操作P
 - 申请资源
- 原子操作V
 - 释放资源

- 利用信号量实现n个进程的互斥

- 互斥信号量mutex，初值为1
- 第i个进程的执行代码

```
1 | do{  
2 |     P(mutex)  
3 |     <critical section>  
4 |     V(mutex)  
5 |     <remainder section>  
6 | }while(1)
```

- 利用信号量实现进程间的同步

计算进程和打印进程同步，共用一个单缓冲

- 计算进程：信号量empty，表示缓冲区是否空，初值为1
- 打印进程：信号量full，表示缓冲区中是否有可供打印的计算结果，初始值为0

- 信号量分类

- 共用信号量
 - 互斥信号量，用于解决进程之间互斥进入临界区
- 私用信号量
 - 同步信号量，用于解决异步环境下进程间的同步

3.2.6 经典进程同步问题

- 生产者-消费者问题

- 问题描述

- 假定有一组生产者(M个) 和一组消费者(N个) 进程，通过一个有界环形缓冲区(k个缓冲块) 发生联系。生产者将生产的产品放入缓冲区，消费者从缓冲区取用产品。
 - 当缓冲区满时，生产者要等消费者取走产品后才能向缓冲区放下一个产品；当缓冲区空时，消费者要等生产者放一个产品入缓冲区后才能从缓冲区取下一个产品。

- 问题解决

- full 有数据的缓冲区个数
 - empty 空缓冲区个数
 - mutex 互斥访问临界区的信号量

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */

红框内P操作互换可能导致死锁
蓝框内V操作互换随意

- 读者-写者问题

- 问题描述

有一个多进程共享的数据区（可以是一个文件或者主存的一块空间），有一些只读取这个数据区的进程（reader）和一些只往数据区中写数据的进程（writer）：

- 任意多的读进程可以同时读数据区
- 一次只有一个写进程可以写数据区
- 若有写进程正在写，禁止任何进程读

- 问题解决

- 读写互斥信号量db：读写互斥和写写互斥访问信号量
- 计数器变量rc：记录同时读的读者数
- 读计数互斥信号量mutex：使读者互斥地访问共享变量rc

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
```

- 哲学家就餐问题

- 死锁问题

3.3 管程

- 引入管程的原因

- 各个进程自备P(S)和V(S)操作，加重了用户负担
 - 大量同步操作分散在各个进程中，系统管理复杂
 - 易产生死锁

- 管程定义

- 一个管程调用了一个数据结构和能为并发进程所执行（在该数据结构上）的一组数据，这组数据能同步进程和改变管程中的数据
 - 管程是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块（类）
 - 一次只能有一个进程能在管程内活动。从而提供互斥机制，保证数据的一致性

- 管程的结构
 - 管程名
 - 局部于管程的共享变量说明
 - 对该数据结构进行操作的一组过程
 - 对局部于过程的数据设置初始值的语句
- 管程的特点
 - 局部于管理的数据结构只能被局部于管程的过程所访问，任何管程外部的过程都不能访问它
 - 局部于管程的过程只能服务管程内的数据结构
 - 管程每次只允许一个进程进入管程，从而实现管程的互斥
- 管程的同步机制
 - 条件变量c
 - 同步原语wait(c)
 - 同步原语signal(c)

3.4 进程高级通信

- 高级通信
 - 消息缓冲
 - 信箱
 - 管道
 - 共享主存区
- 发送进程与接收进程状态
 - 阻塞
 - 非阻塞
- 通信方式组合
 - 非阻塞发送，非阻塞接收
 - | 信箱
 - 非阻塞发送，阻塞接收
 - 阻塞发送，阻塞接收
 - | 双向通信

3.4.1 消息缓冲通信

- 消息缓冲属于直接通信方式
 - 发送进程发送时要指明接收进程的名字
 - 接收进程接收消息时要指明发送进程的名字
- 消息缓冲通信有两种实现方法
 - 消息缓冲区
 - 消息队列通常放在接收进程的进程控制块中
 - 信箱
 - 消息缓冲（消息缓冲区）实现机制
 - 消息缓冲区

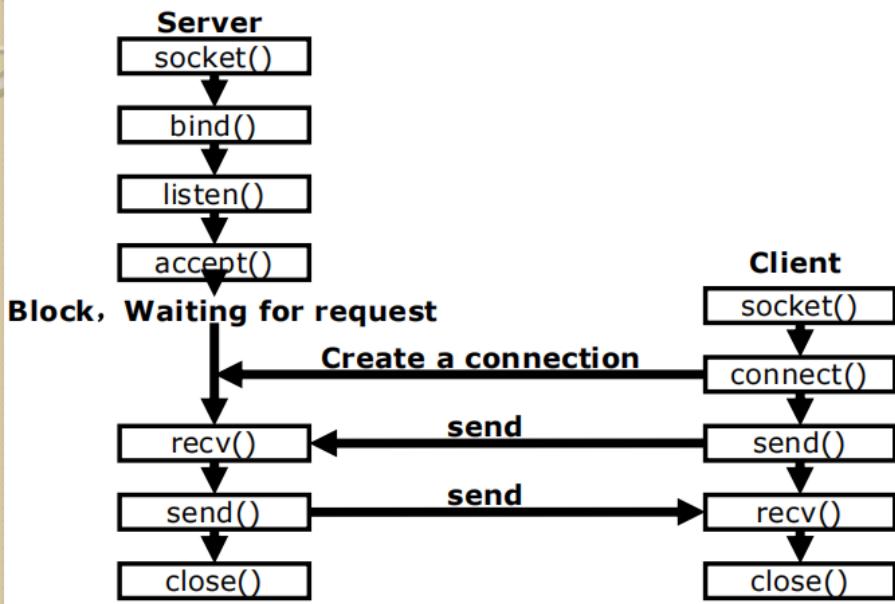
- 消息发送原语
 - send
 - 主要工作
 1. 请求分配一个消息缓冲区
 2. 将消息正文传送到该缓冲区中，并填入有关发送参数
 3. 将该消息缓冲区挂到接收进程消息链上
- 消息接受原语
 - receive
 - 主要工作
 1. 检查消息链上是否有消息
 2. 有则将消息接收到接收区
 3. 无则阻塞等待消息的到来
- PCB结构
 - 消息队列头指针
 - 实现同步与互斥机制信号量

```
struct PCB {
    ...
    mq;      //消息队列队首指针
    mutex;   //消息队列互斥信号量
    sm;      //消息队列同步信号量
    ...
}
```

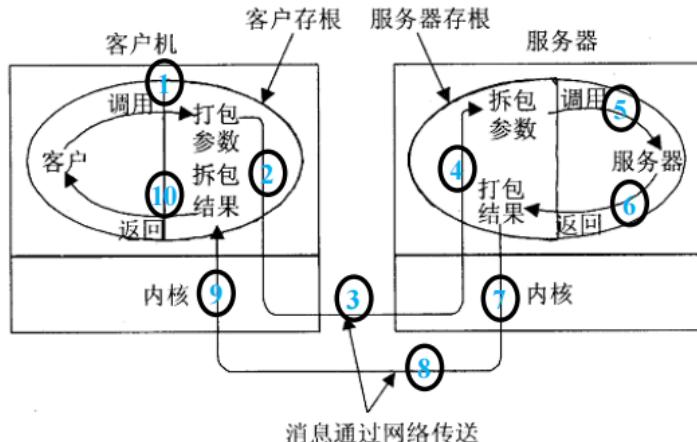
3.4.2 其他通信机制

- 管道
 - pipe文件：用于连接一个写进程和一个读进程的共享文件
 - 先进先出
 - 单向通信
- 共享主存区
 - 在主存中划出一块共享主存区，并将共享主存区连接到进程地址空间的某一部分
 - 各进程通过读和写共享主存区中的数据来实现通信
 - 最快捷的方式

- Socket



- RPC



3.5 死锁

3.5.1 基本概念

- 死锁概念
 - 多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若外力作用，所有进程都将无法向前推进
- 产生死锁的原因
 - 竞争资源（资源数量不足）
 - 资源类型
 - 可抢占资源：主存、CPU
 - 不可抢占资源：慢速设备/共享设备
 - 进程推进顺序非法
 - 请求和释放资源的顺序不当
 - 死锁产生的必要条件（同时具备）

- 互斥条件
 - 每个资源不可共享，它或者已经分配给一个进程，或者空闲
- 不可剥夺条件
 - 进程所获得的资源在未使用完之前，不可被其它进程强行剥夺，只能由获得资源的进程自行释放
- 保持和等待条件
 - 进程因请求资源而被阻塞等待时，对已经分配给它的资源保持不放
- 循环等待条件
 - 存在进程循环链，链中有两个或以上的进程等待链中下一个成员保持的资源

- 资源分配图

- 图例

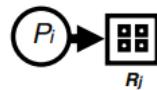
- ◆ 进程



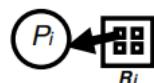
- ◆ 具有4个实例的资源



- ◆ P_i 请求资源 R_j 的一个实例



- ◆ P_i 拥有资源 R_j 的一个实例



- 判断方式

- 无环路，无死锁
 - 有环路
 - 每类资源只有一个实例，有死锁
 - 每类资源有多个实例，可能死锁

- 解决死锁的方法

- 不予理睬（鸵鸟算法）
 - 假定系统不会出现死锁
 - 保证系统永远不进入死锁
 - 死锁预防
 - 静态分配资源等策略
 - 死锁避免
 - 动态分配资源前，判断系统安全性
 - 允许系统进入死锁状态，并可从死锁状态中恢复
 - 死锁检测
 - 死锁恢复
 - 剥夺资源
 - 撤销进程

3.5.2 鸵鸟算法

- 假定死锁从不发生
- 原因
 - 死锁很少
 - 预防死锁代价高

3.5.3 死锁预防

- 破坏死锁产生的必要条件
 - 破坏互斥条件
 - 资源特性
 - 将独享设备改为共享设备
 - 破坏非剥夺条件
 - 当一个已经占有一些资源的进程提出申请新的资源的请求不能立即满足时，必须释放已经占有的全部资源，待需要再重新申请
 - 破坏保持和请求条件
 - 进程在开始运行前必须获得全部资源，若不能满足则必须等待
 - 破坏循环等待条件
 - 系统对全部资源按类分配编号，进程对资源的请求按顺序申请

3.5.4 死锁避免

- 安全状态

系统能按照某种进程顺序(P_1, P_2, P_3, \dots)来为每个进程分配资源，直至满足每个进程对资源的最大需求，使每个进程都能顺利完成
- 安全序列

(P_1, P_2, \dots, P_n)
- 不安全状态

不存在一个安全序列
- 原则
 - 系统处于安全状态 \Rightarrow 无死锁
 - 系统处于不安全状态 \Rightarrow 可能死锁
 - 要避免死锁，则不让系统进入不安全状态
- 银行家算法
 - 数据结构
 - n 表示进程数目， m 表示资源类型
 - 可用资源矩阵 Available
 - 最大需求矩阵 Max
 - 分配矩阵 Allocation
 - 需求矩阵 Need
 - $$\text{Need} = \text{Max} - \text{Allocation}$$
 - 算法
 - 设 $\text{Request}[j] = k$
 - 1. $\text{Request}[j] \leq \text{Need}[i,j]$, 则执行2

2. $\text{Request}[j] \leq \text{Available}[j]$, 则执行3
3. 假定将资源分配给 P_i , 则执行
 - $\text{Available}[j] = \text{Available}[j] - \text{Request}[j]$
 - $\text{Allocation}[i,j] = \text{Allocation}[i,j] + \text{Request}[j]$
 - $\text{Need}[j] = \text{Need}[j] - \text{Request}[j]$
4. 系统执行安全性算法
 - 安全, 则资源分配给 P_i
 - 不安全, 则 P_i 等待

3.5.5 死锁检测

- 检测时机
 - 每次资源请求时检测
 - 较早检测到
 - 浪费大量CPU
 - 间隔一段时间检测

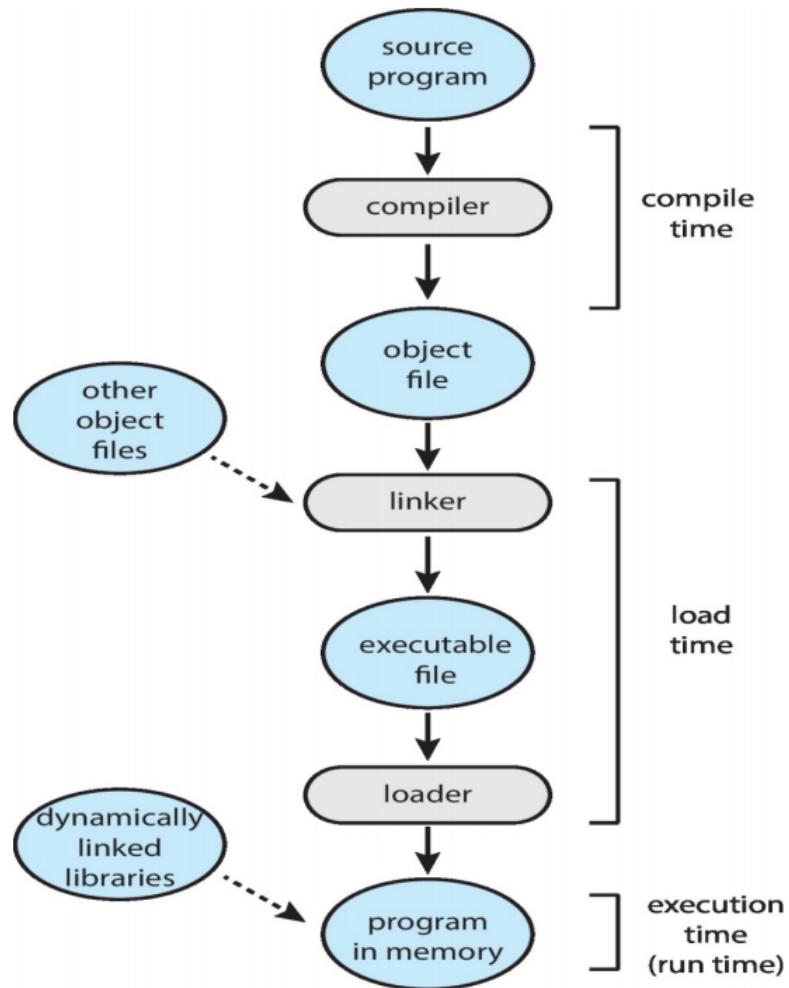
四、存储器管理

4.1 概述

- 存储器体系
 - 快速的昂贵的cache
 - 中速的、价格中等的内存
 - 实存
 - 虚存
 - 慢速的、大容量、低价格的磁盘存储器
- 存储器管理的目的
 - 方便用户使用
 - 提高存储器利用率
- 存储器管理的功能
 - 存储器分配
 - 地址转换
 - 存储器保护
 - 防止地址越界
 - 存取方式检查
 - 存储器共享
 - 允许多个进程共享一个内存区
 - 既可以是数据区, 也可以是程序区
 - 被共享的程序称为可重入程序, 不论执行多少遍都保持不变。又叫纯代码。

4.2 程序的装入与链接

- 编译链接



- 程序名空间
 - 程序中各种符号名的集合所限定的空间
- 地址空间
 - 地址空间：经编译或连接后目标代码所限定的地址域
 - 相对地址（逻辑地址、虚地址）：地址空间中的各个地址
- 存储空间
 - 存储空间：物理存储器中全部物理单元集合所限定的空间
 - 绝对地址（物理地址、实地址）：存储空间中的全部地址
- 地址重定位（地址映射、地址变换）
 - 将地址空间的逻辑地址转换为存储空间的物理地址
 - 定位方式（地址变换时间与技术不同）
 - 静态重定位
 - 在程序装入时，装入程序把用户程序地址空间中的指令和数据的相对地址全部转换成存储空间的绝对地址
 - 动态重定位
 - 由硬件地址转换机构——重定位寄存器实现
 - 转换工作在程序执行过程中进行
 - 优点
 - 执行期间可移动用户程序，移动后只需修改重定位寄存器即可

- 进程不必占用连续的内存空间
 - 便于多用户共享存储器中的进程
 - 主存利用率高
- 程序链接
 - 连接方式 (根据链接时间划分)
 - 静态链接
 - 装入时动态链接
 - 运行时动态链接
 - 静态链接
 - 在程序运行之前, 将所有编译生成的目标模块和所需库函数链接成一个完整的装配模块, 以后不再拆开
 - 采用静态重定位
 - 需解决的问题
 - 对相对地址进行修改
 - 变换外部调用符号
 - 物理上作为一个文件
 - 装入时动态链接
 - 编译后的目标模块, 在装入内存时边装入边链接
 - 采用静态重定位
 - 优点
 - 各目标模块分开存放, 便于修改和更新
 - 便于实现对目标模块的共享
 - 运行时动态链接
 - 将对某些模块的链接推迟到执行时才进行
 - 执行时未调用的目标模块不会被装入
- 程序的装入
 - 装入模块的装入方式
 - 绝对装入方式
 - 可重定位装入方式
 - 动态运行时装入方式
 - 绝对装入方式
 - 编译时, 若知道程序驻留在内存中的位置, 编译程序将产生绝对地址的目标代码
 - 绝对装入程序按照装入模块中的地址, 将程序和数据装入内存
 - 逻辑地址=实际内存地址
 - 绝对地址的获取
 - 程序员直接赋予
 - 编译或汇编时给出
 - 问题
 - 多道环境下编译程序不可能预知所编译模块在内存中的位置
 - 只适合单道程序设计
 - 可重定位装入方式
 - 多道程序环境下, 目标模块的起始地址通常从0开始, 程序中的其他地址均相对于该起始地址进行计算
 - 采用可重定位装入方式

- 静态重定位，转换工作在程序执行前一次完成，不能更改和移动
- 优点
 - 无需硬件支持，易于实现
 - 为每个进程分配一个连续的存储区
- 缺点
 - 进程占有连续的存储区
 - 程序执行期间不允许在主存移动
 - 不能共享存储器中的程序
- 动态运行时装入方式
 - 在将装入模块装入内存后，并不立即把装入模块中的相对地址转换为绝对地址，而是将地址转换推迟到程序真正要执行时才进行
 - 装入内存后所有地址都仍是相对地址

4.3 分区式存储管理

4.3.1 固定式分区

- 适合多道程序的最简单的存储管理方式
- 将主存预先划分为几个分区
 - 大小相等
 - 大小不等
- 进程到达时选择一个适合程序要求的空闲区
- 若没有可用的空闲分区，进程在分区队列中等待
 - 为每个分区分配独立的等待队列
 - 全部分区分配一个等待队列
- 实现
 - 设置主存使用情况表描述主存使用情况
 - 内存管理过程
- 优点
 - 简单易于实现
- 缺点
 - 主存利用不充分

4.3.2 可变式分区

- 根据进程大小动态地划分分区，使分区的大小正好等于进程大小
- 程序要求运行时，由系统从可用的空闲存储空间中划分一大小正好等于进程大小的存储区分配给程序
- 优点
 - 提高了存储器的使用效率
- 缺点
 - 分配与释放复杂

- 问题
 - 分区大小不变
 - 分区数目不定
- 管理方式
 - 分区说明表
 - 已分配区表
 - 未分配区表
 - 空闲区链
- 分配与回收
 - 分配
 - 从空闲区表中找一个足以容纳该进程的空闲区。若该分区较大，则一分为二，一部分分配给进程，一部分留在空闲区表中
 - 已分配区表中找一个空表目，填入新分配进程的信息
 - 回收
 - 回收分区登记在未分配区表中
 - 若有相邻接的空闲区，则合并后再登记
 - 将该进程占用的已分配区表目置空
- 动态分区分配算法
 - 首次适应法
 - 空闲区按照起始地址的大小从小到大排列
 - 从链首开始扫描空闲区链，直至找到一个足够大的空闲区
 - 空闲区一分为二
 - 若无满足要求的空闲区则分配失败（等待）
 - 优点
 - 优先利用低址空闲区，保留高址大空闲区
 - 缺点
 - 低址部分被不断划分，留下许多难以利用的、很小的空闲分区
 - 每次寻找均从低址部分开始，增加了查找空闲分区的开销
 - 改进
 - 循环首次适应算法
 - 最佳适应法
 - 扫描整个链表，将最接近进程需求量的空闲区分配给进程
 - 缺点
 - 每次查找整个链表，效率低
 - 浪费更多的存储空间，将主存划分为很多较小的无用的碎片
 - 改进
 - 空闲区按容量大小从小到大排列
 - 最差适应法
 - 扫描整个链表，直到找到该链中满足进程需求且为链中最大的空闲区为止
 - 空闲区一分为二

4.3.3 分区管理的地址重定位

- 固定式分区
 - 通常固定分区采用静态重定位，进程运行时使用主存绝对空间
- 可变式分区
 - 采用动态重定位，进程运行时使用相对空间

4.3.4 分区管理的存储器保护

- 上下限寄存器
 - 用于静态重定位
 - 下界寄存器内容 \leq 访问内存的低址 \leq 上界寄存器内容
- 基址+限长寄存器
 - 用于动态重定位
 - 基址（重定位）寄存器：分区首址
 - 限长寄存器：分区长度

4.3.5 分区管理的优缺点

- 优点
 - 实现了多道程序共享内存
 - 实现分区管理的系统设计相对简单，不需要更多的系统软硬件开销
 - 实现存储保护的手段比较简单
- 缺点
 - 主存利用不充分，存在碎片（零头）
 - 内零头：存在于进程存储空间内部（固定式）
 - 外零头：存在于整个主存空间（可变式）
 - 解决办法
 - 紧凑
 - 不占用连续的主存空间
 - 程序地址空间大于存储空间时，程序无法运行，即程序的地址空间受到实际存储空间的限制，无法对主存进行扩充
 - 解决办法
 - 覆盖与交换
 - 虚拟存储器

4.4 覆盖与交换技术

- 覆盖
 - 同一主存区可被不同的程序段重复使用
 - 原理：一个程序由若干功能独立的程序段组成，程序运行时，一些程序段不会同时执行，可以共用同一主存区
 - 覆盖区：可以共享的主存区
 - 覆盖段：程序执行时不要求同时装入主存的程序段（覆盖）组成一组，叫做覆盖段，并分配同一个主存区（覆盖区）

- 覆盖结构由用户实现，无需OS特殊支持
- 交换
 - 系统根据需要把主存中暂时不运行的某个或某些进程部分或全部移到外存，而把外存中的某个或某些进程移到相应的主存区，并使其投入使用
 - 时机
 - 进程用完时间片或等待输入输出
 - 进程要求扩充存储而得不到满足
 - 关键
 - 在外存保存副本，每次仅修改变化部分

4.5 页式存储管理

4.5.1 页式管理的实现原理

- 页面与物理块
 - 物理块（页框）
 - 将物理存储空间划分成大小相等的若干存储块
 - 大小为2的整次幂，大小在4KB到1GB之间
 - 页面
 - 将进程的逻辑地址空间划分成与物理块大小相同的若干片
- 页面和物理块的大小一定是相同的
- 地址计算
 - 逻辑地址空间大小时 2^m ，逻辑上连续（但是物理上不一定连续）
 - 每一页的大小是 2^n 个逻辑地址单元
 - 页号与页内位移计算公式：
 - 逻辑地址空间中的地址为A
 - 每一页的大小为 $L = 2^n$ ，则
 - 页号： $p = A \text{ div } L$
 - 页内地址： $d = A \text{ mod } L$

◦ 学会用二进制地址结构进行划分计算，即补齐m位，取低n位，高m-n位

页号	页内位移
p $m-n$	d n

- 地址结构
 - 页号(p)：用作页表的索引，包含了每一页在物理内存中的起始地址
 - 页内位移(d)
 - 页号与页面位移各占多少位，与页的大小和主存最大容量有关
 - 地址分离工作由硬件实现，不需要用户管理
- 采用分页技术不会产生外部碎片，但可能产生内部碎片（最后一页）
- 页表

- 记录进程的逻辑页与主存中物理块的对应关系，实现从页号到物理块号的地址映射
- 形成：存储分配时
- 进程地址空间的每一页对应一个表目，指出该逻辑页在主存中的物理块号
- 页表放在主存中，页表的主存始址与页表长度保存在进程控制块中

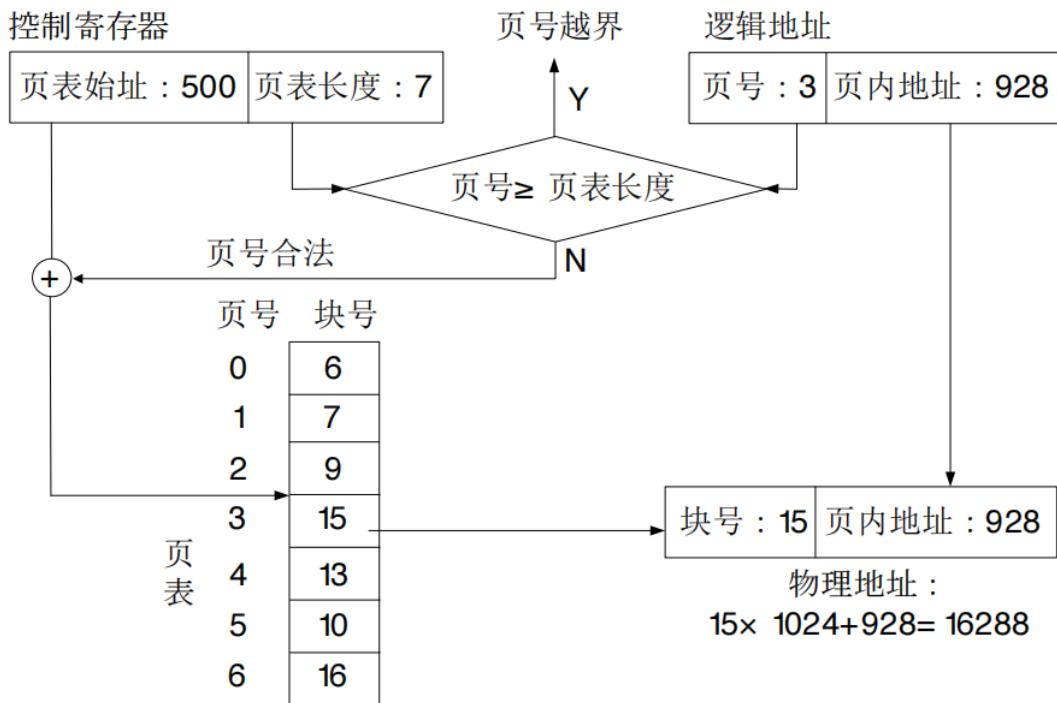
◦ 每个进程对应一个页表

- 控制寄存器
 - 页表基址寄存器：指向页表
 - 页表长度寄存器：页表长度
- 结构
 - 用索引/下标表示逻辑页号
 - 记录物理块号
- 每一页表项大小
 $(k - n) / 8$ 字节

◦ 需要 $k-n$ 位来存放所有物理页框编号

4.5.2 页式动态地址变换

- 变换过程
 - 将进程的页表始址和页表长度送入控制寄存器中
 - 比较程序计数器内的页号 p 与控制寄存器中的页表长度，若页号小于页表长度，则继续，否则产生地址越界，终止程序运行
 - 将程序计数器中的页号 p 与控制寄存器中的页表始址相加，得到该访问操作的页号在页表中的入口地址
 - 以该地址访问页表，得到该页所对应的主存块号 f
 - 将主存块号 f 与程序计数器中的页内偏移相拼接，得到该操作所在主存的物理地址： $f \times L + d$
- 页内存储空间连续，故相对位置是不变的，所以 d 不变
- 根据物理地址，完成指定操作



- 存在的问题
 - 执行一次访问操作至少要访问主存两次
 - 访问页表
 - 实现指定操作

4.5.3 快表和联想存储器

- 快表定义
 - 为了提高存取速度，在地址变换机构中设置的具有并行查找能力的专用高速缓冲寄存器组（32-1024个寄存器）。用来存放页表的一部分
- 快表结构
 - 页号：程序当前访问的地址空间的页号
 - 块号：该页所对应的主存块号
 - 访问位：指示该页最近是否被访问
 - 0：未被访问（优先被置换）
 - 1：访问过
 - 状态位：指示该寄存器是否被占用
 - 0：空闲
 - 1：占用
- 地址变换过程
 - 硬件地址转换机构在进行地址变换时，有两个变换过程：
 1. 用快表进行的快速变换过程
 2. 用主存页表进行的正常变换过程
 - 快表命中：一旦快表中与查找的页号相符合时（并行），则将快表中的块号与CPU给出的页内位移相拼接，得到访问主存的绝对地址，结束快表查找工作
 - 快表未命中：若利用快表进行变换时，没有找到要查找到要访问的页，则继续正常的变换过程
 - 进行快表更新

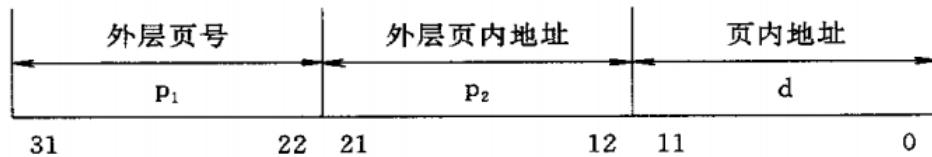
- 看状态位
- 看访问位
- TLB命中率
 - 页号在TLB中被查找到的百分比
 - TLB命中率提高可以提高地址变换效率

示例：假设查找TLB需要20ns，访问内存需要100ns，则内存数据访问时间为

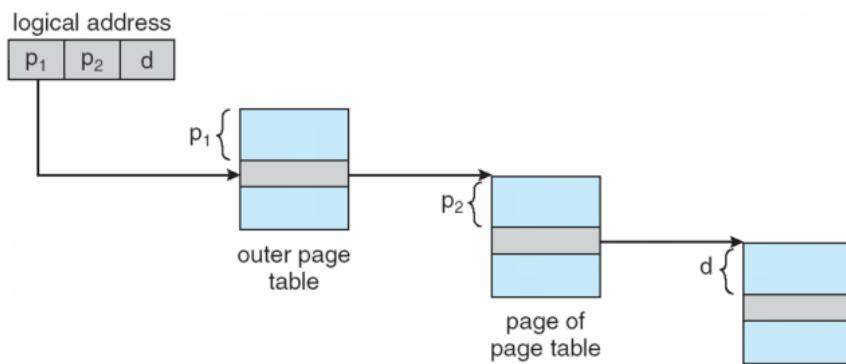
 - 页号位于TLB： $20+100=120\text{ns}$
 - 页号不在TLB： $20+100+100=220\text{ns}$
 - TLB命中率为80%：
 - $0.8 \times 120 + 0.2 \times 220 = 140\text{ns}$
 - TLB命中率为98%：
 - $0.98 \times 120 + 0.02 \times 220 = 122\text{ns}$

4.5.4 多级页表

- 引入
 - 问题：页表本身就占用很大的连续存储空间
 - 解决：页表不连续存储
 - 将页表再分页



- 两级页表



- 建立页表的时机
 - 页表的建立不再是进程装入主存时
 - 而是推迟到真正要访问页时，才为包含该页的页表分配空间和建立页表项

4.5.5 页式管理的主存分配

- 数据结构
 - 页表
 - 进程控制块

- 存储空间使用情况表
 - 记录存储空间的使用情况
 - 存储分快表/位示图
- 存储分块表
 - 记录存储器中各块的状态
 - 第一项指出当前主存空闲块数
 - 第二项是指向第一个空闲块指针
 - 分配
 - 检查存储分快表能否满足进程要求
 - 若不能满足，则进程等待
 - 若能满足，由存储分块表中第一项减去空闲块数，再由第二项空闲块指针找到所需各块，并为进程建立页表、修改存储分块表第二项的空闲块指针
 - 回收
 - 将进程占用主存块归还系统，并修改存储分快表有关各项
- 位示图
 - 使用一个位向量，磁盘中的每一块占用其中一位
 - 0为空闲，1为占用

4.5.6 页式管理的保护

- 越界保护
 - 页号与页表长度寄存器存放的值进行比较
- 访问保护
 - 在页表中为每个物理块设置保护位
 - 读写保护
 - 有效-无效保护

4.6 段式管理

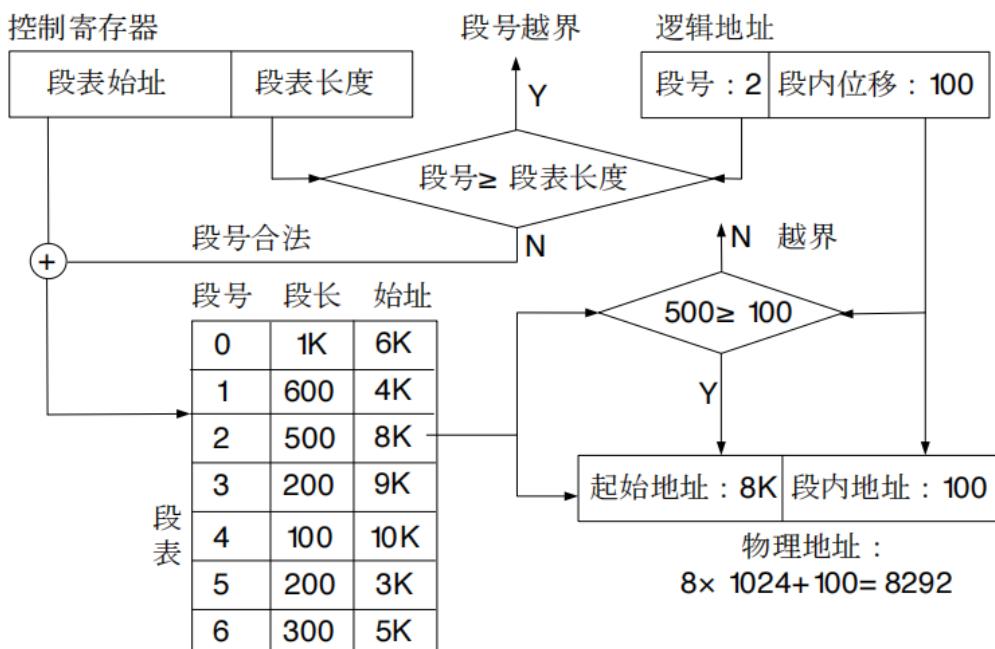
4.6.1 段式管理的实现原理

- 分段管理方式的引入
 - 满足编程和使用上的需求
 - 方便编程
 - 信息共享
- 分段
 - 按照程序自身的逻辑关系将作业的地址空间划分而成的若干部分
 - 每个段都有自己的名字
 - 主程序段MAIN
 - 子程序段X
 - 数据段D
 - 栈段S

- 每个段都占用从0开始编制的连续地址空间
- 二维地址空间结构
 - 段号：内部段号
 - 段内位移
- 段式存储分配管理以段为单位进行，为作业的每一个分段分配一个连续的存储空间，各段之间可以不连续
- 段表
 - 记录进程分段与物理存储空间的对应关系，实现从逻辑分段到物理内存的地址映射
 - 每个逻辑分段对应一个表目，指出该逻辑分段主存中的起始地址和段的长度
 - 段表存放于主存
 - 段表的主存始址与段表长度保存在进程控制块
 - 控制寄存器
 - 段表基址寄存器：指向段表
 - 段表长度寄存器：段表长度

4.6.2 段式动态地址变换

- 示意图



4.7 段页式管理

- 结合分段和分页
- 先将用户程序分为若干个段，再把每个段分成若干个页，并为每个段赋予一个段名
- 地址结构由三部分构成
 - 段号
 - 页号
 - 页内位移

- 地址变换
 - 三次访问内存
 - 快表

4.8 虚拟存储器

4.8.1 基本概念

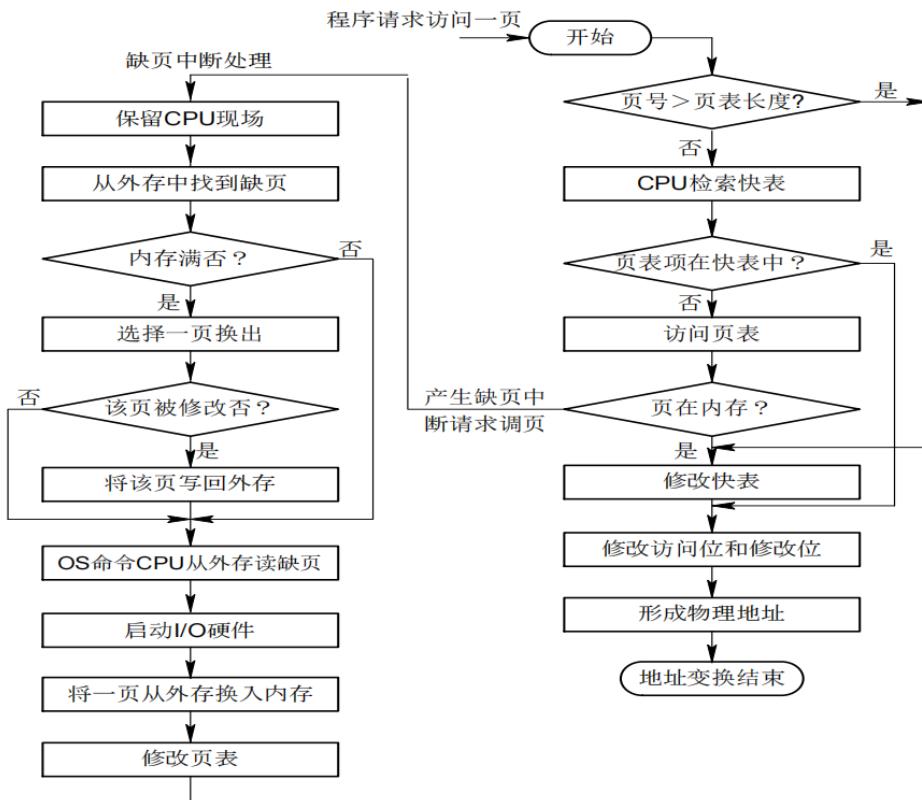
- 实存管理技术
 - 特点
 - 一次性
 - 驻留性
 - 缺点
 - 进程请求的内存空间超过内存总容量，进程不能全部存入内存，无法运行
 - 内存容量不足以容纳大进程，只能先运行小进程，大进程等待
- 虚存存储技术：虚拟存储器
 - 具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统
 - 逻辑容量 = 内存容量 + 外存容量
 - 运行速度接近于内存速度
 - 特征
 - 多次性
 - 对换性
 - 虚拟性
- 局部性原理
 - 程序在执行时将呈现出局部性规律，即在一较短时间内，程序的执行仅局限于某个部分；相应地，它所访问的存储空间也局限于某个区域
 - 类型
 - 时间局限性：循环
 - 空间局限性：互斥执行
- 结合多级页表可以节省存储空间

4.8.2 页式虚拟存储器管理

- 实现原理
 - 页式管理+交换技术
- 硬件支持- 修改页表
 - 有效位（状态位）
 - 指示某页是否在主存
 - 修改位
 - 指示该页被调入主存后是否被修改过
 - 访问位（引用位）
 - 用于页面置换

请求 页式 管理 地址 变换 过程

- 示意图



- 页面置换算法

- 抖动

- 刚被淘汰的页面马上又要调入，调入不久又被淘汰，如此反复。降低了系统处理效率

- 算法假设

- 一个进程分配的主存块数固定不变
(物理块是不变的)
 - 采用局部淘汰 (淘汰一页时，只考虑在本进程内部实现淘汰)

- 算法性能

- 缺页率
 $f = F/A$

- 理想调度算法

- 选择以后不再访问的页或经过很长时间后才可能访问的页进行淘汰

- 页面序列

- $1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$

- 置换算法

- 最佳算法 (OPT)

- 需要预先知道页面序列，因而是理想的

- 先进先出算法 (FIFO)

- 先被调入的先被淘汰
 - Belady异常：物理块增加，缺页次数反而增加了
 - 容易产生抖动

- 最近最久未用算法 (LRU)

- 软件实现：栈式页面置换算法

- 特性：这一类算法绝对不会产生Belady异常

- 最近最少使用算法 (LFU)

- 硬件实现：系统设置一个64位的硬件计数器C，每当一条指令引用后都自动计数

- 时钟页面置换算法 (CLOCK)

- 将进程所访问的页像时钟一样存放于一个循环链中
- 设置一个指针指向最早进入主存的页
- 产生缺页中断时，按照下列原则选择某一个页面进行淘汰
 1. 检查访问位，若为0，置换该页，指针+1
 2. 若为1，则将访问位置为0，指针+1，继续检查访问位

| 二次机会

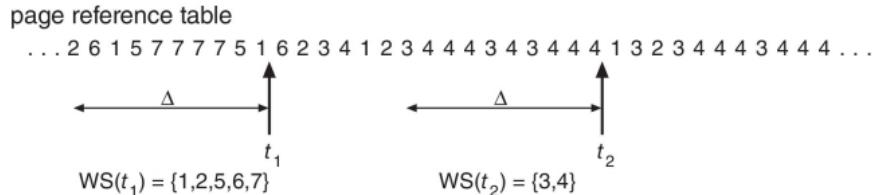
 3. 重复过程，直到找到访问位为0的页

4.8.3 页式管理设计中的问题

- 确定页面大小
 - 页面越大，无用程序装入主存就越多，从而使主存浪费严重
 - 页面越小，程序需要的页就越多，页表越大
- 物理块分配算法
 - 可变分配
 - 固定分配
 - 平均分配
 - 按比例分配
 - 按优先权分配
- 页面置换策略
 - 局部置换
 - 全局置换
- 物理页框分配+页面置换策略
 - 固定分配局部置换
 - 为每个进程分配多少物理块难以确定
 - 可变分配局部置换
 - 可变分配全局置换
- 调页时机
 - 预调页
 - 根据空间局部性，将不久之后会被访问的页面预先调入内存
 - 成功率仅有50%，主要用于进程的首次调入
 - 请求调页
 - 进程运行时，发现其所访问的程序页面不在内存，请求系统将所需页面调入内存
- 抖动与工作集
 - 抖动原因
 - 同时在系统中运行的进程太多，分配给进程的物理块太少，频繁调入调出
 - 工作集
 - 局部性原理
 - 在某段时间间隔内，进程实际所访问的页面的集合

- 定义

- 进程在时间t的工作集为 $W(t, \Delta)$
- 进程在时间间隔 $(t-\Delta, t)$ 中引用页面的集合
- 变量 Δ 为工作集的“窗口尺寸”
- 示例： $\Delta=10$



找不同的页面序号（元组）

- 抖动预防

- 采用局部置换策略
- 把工作集算法融入到处理机调度中
- “L=S”准则调节缺页率

L为缺页之间的平均时间，S为平均缺页服务时间

- 选择暂停的进程

- 页面共享

- 分时系统中，通常几个用户同时运行同一个程序（如编辑、编译等）
- 为了节约主存，共享页
- 不是所有页都可共享
 - 只读页，如程序文本，可以被共享
 - 可读写的数据页，不可共享
- 需要一个专门的数据结构记录共享页
 - 将被共享页锁在主存中，并在页表中增加引用计数项，仅当其引用计数为0时，才允许调出或释放共享页所占的空间

- 写时复制

- 当两个进程要读写相同内容的内存时，允许两个进程共享同一物理块，这些物理块标记为写时复制页
- 若两个进程对该块只读，则二者共享
- 若某一进程对该块写，则系统将该物理块复制到主存的另一个物理块中，并更新该进程的页表指向该物理块

五、文件系统

5.1 文件和文件系统

- 文件是具有符号名的相关信息的集合
- 文件的基本特征
 - 文件的内容为一组信息
 - 有结构
 - 无结构
 - 文件具有保护性
 - 长期保存，多次使用
 - 文件可以实现按名存取
 - 每个文件都有唯一的标识符，通过该标识符存取文件中的信息，无须了解文件在存储介质上的具体位置
- 文件的分类
 - 按用途
 - 系统文件
 - 库文件
 - 应用程序文件
 - 用户文件
 - 按保护方式
 - 只读文件
 - 读写文件
 - 无保护的文件
 - 按信息流向
 - 输入文件
 - 输出文件
 - 输入/输出文件
 - 按信息保存期限
 - 临时文件
 - 永久文件
 - 档案文件
 - 按文件所在位置
 - 磁盘文件
 - 硬盘文件
 - 软盘文件
 - 光盘文件
 - Unix系统中，为了方便用户操作文件，分为：
 - 普通文件（系统文件、应用程序文件、库文件、用户文件）
 - 目录文件
 - 特别文件（输入、输出、输入/输出型设备，及其他）

磁盘分为软盘和硬盘

硬盘分为机械硬盘和固态硬盘，一般所说硬盘指的是机械硬盘，有读写头

- 文件属性
 - 文件名（扩展名）
 - 文件标识符
 - 文件类型
 - 文件大小

- 文件的建立时间
- 文件的物理位置
- 文件的保护信息
- 文件系统
 - 为了方便用户使用软件资源，由OS提供的管理文件的软件机构
 - 文件系统既包括OS中用于文件管理的程序，也包括运行这些程序所需的各种数据结构
 - 一个理想的文件系统应当具备的结构
 - 管理磁盘、磁带等组成的文件存储器
 - 实现用户的按名存取，负责名字空间到存储空间的映射
 - 具有灵活多样的文件结构和存取方法
 - 向用户提供一套使用方便、简单的操作命令
 - 保证文件信息的安全性
 - 便于文件的共享

5.2 文件目录项和文件目录结构

- 文件目录
 - 文件的说明信息与控制信息
 - 作用
 - 实现按名存取
 - 便于文件共享和保密
 - 文件目录表
 - 一张记录所有文件的名字及其存放物理地址的映射表
 - 每个文件占据文件目录表中的一个表项
 - 文件目录项
 - 文件控制块（FCB）：文件存在的唯一标志
 - 存放文件的全部控制信息
 - 存放文件名和文件存储位置
- 文件控制块
 - 基本信息
 - 文件名：惟一性（不是可读的文件名，而是内部ID）
 - 文件类型
 - 文件物理位置
 - 设备名
 - 文件在外存上的起始盘块号
 - 文件长度：文件占用的盘块数和字节数
 - 文件逻辑结构
 - 文件物理结构
 - 存取控制信息
 - 文件主的存取权限
 - 其他用户存取权限
 - 使用信息
 - 文件建立时间

- 上一次修改时间
 - 当前使用信息
- 目录管理的要求
 - 实现按名存取
 - 提高对目录的检索速度
 - 文件共享
 - 允许文件重名
 - 文件目录结构
 - 一级目录结构
 - 为文件存储器上保存的全部文件建立一张目录表，每个文件在表中占有的一项
 - 创建文件/删除文件
 - 优点
 - 目录结构管理简单
 - 缺点
 - 不允许文件重名，否则出现二义性
 - 文件较多时，查找目录耗时太多
 - 限制了对文件的共享
 - 二级目录结构
 - 为每个用户建立独立的目录结构
 - 由一个主文件目录和若干用户文件目录组成
 - 主目录中的目录项记录各用户目录的名字及目录文件所在的磁盘地址
 - 创建新用户
 - 系统为新用户在主文件目录表中找一个空表目，并为其分配一个存放用户文件目录的区域，然后把用户名和该区域的起始地址填写该在对应表目中
 - 创建新文件
 - 用户建立一个新文件时，文件系统先按用户名在主目录中找到相应的文件目录的地址，然后在用户目录表中为其建立一个目录项，填上文件名以及文件的管理和控制信息
 - 访问文件
 - 先按用户名在主目录中找到相应文件目录项（地址信息）
 - 按文件名在用户文件目录中找到文件的相关信息
 - 完成指定的文件操作
 - 删除文件
 - 按用户名在主目录中找到相应文件目录项（地址信息）
 - 按文件名在用户文件目录中找到相应文件目录项，释放文件所占用的块并撤销该目录项
 - 当该用户的文件目录中的文件都被撤销时，将其所占目录表区释放，并将该用户在主目录表中的目录项撤销
 - 优点
 - 解决了文件的重名（用户名|文件名）问题和文件共享问题
 - 降低了文件查找时间
 - 缺点
 - 增加了系统开销
 - 对用户进行隔离
 - 多级目录结构

- 有一个主目录（根目录），在根目录下又有许多子目录和普通文件的记录，每个子目录下依次也有许多子目录或文件作为其目录项
- 优点
 - 层次结构清晰，解决重名问题
 - 便于管理和保护
 - 有利于文件分类
 - 提高文件检索速度
- 缺点
 - 管理开销大
 - 按路径名逐层查找文件，需多次访问磁盘，影响速度
- 绝对路径
 - 文件的固有名由根（主目录）到文件通路上所有目录与该文件的符号名拼接而成
- 相对路径
 - 当前目录（工作目录）
 - 文件名由“当前目录”到欲访问文件之间的所有符号名拼接而成

5.3 文件操作指令

- CREATE：创建一个文件
 - 在指定设备上为文件产生一个目录项，分配必要的外存空间，并设置文件相关属性
- DELETE：删除一个文件
 - 当一个文件不再需要，可用此命令将它删除。文件删除后，其对应的目录项不再存在，并回收该文件占用的存储空间
 - 不是擦除信息，而是标记为可用的；所以删除了也是有可能恢复的
- READ：读文件
 - 调用者必须指出要读取的文件名，数据大小以及存放数据的主存缓冲区地址
- WRITE：写文件
 - 将主存缓冲区中的数据写到指定的文件中，通常是写在当前位置
- APPEND：追加
 - 将文件追加到文件尾
- SEEK
 - 将文件的读写指针由当前位置重新定位到指定位置，以便随机存取文件中任意位置的数据
- OPEN：打开文件
 - 进程在使用前，必须先打开该文件
 - 打开：将文件的目录项复制到主存的一个专门区域，从而建立进程与文件的联系，加快进程对文件的存取速度
- CLOSE：关闭文件
 - 当进程对文件的所有存取完成后，该文件在主存专用区对应的目录项就不再需要，应该将它关闭，以便释放这部分主存空间，让以后需要打开的文件占用；
 - 若该目录项调入主存后修改过，还要复制到磁盘
 - 文件关闭后，不能再使用
- GET ATTRIBUTES：获取文件属性
- SET ATTRIBUTES：设置文件属性
- RENAME：文件重命名

5.4 文件逻辑结构和文件访问方式

- 文件结构
 - 文件的组织形式
 - 用户使用角度
 - 文件的逻辑结构：用户如何组织和使用文件
 - 系统使用角度
 - 文件的物理结构：文件的物理存储
- 文件系统的重要作用
 - 在用户逻辑文件和相应存储设备上的物理文件之间建立映射，实现二者的相互转换
- 文件逻辑结构的基本要求
 - 能够提高检索速度
 - 便于修改
 - 降低文件的存储费用
- 文件逻辑结构分类
 - 无结构的字符流
 - 有结构的记录式文件
 - 记录：一组相关联的数据项
 - 定长记录
 - 变长记录
 - 记录的组块与分解
 - 组块：把多个逻辑记录放在一个物理块中的工作
 - 分解：从一个物理块中将一个逻辑记录分离出来的工作
 - 由于信息交换是以物理块为单位，进行组块和分解操作时，必须使用主存缓冲区
- 文件访问方法
 - 顺序访问
 - 严格按照字符流或者记录排列的先后次序依次访问
 - 对文件的每一次访问都在前一次访问基础上进行
 - 系统设置两个位置指针指向其中要读写的字节位置或记录位置。根据要读写的字节数或者记录长度，系统自动修改指针位置
 - 通常对于变长记录采用顺序访问
 - 直接访问
 - 通常针对定长记录文件
 - 在请求对某个文件进行访问时，要指出访问的记录号
 - 按键访问
 - 按给定的字段值进行访问
 - 广泛应用于数据库系统

5.5 文件物理结构和文件存储介质

5.5.1 文件的物理结构

- 文件的物理结构
 - 指一个文件在文件存储器上存储方式及其与文件逻辑结构的关系
 - 说明
 - 物理块：将文件存储器的存储空间划分为若干个大小相等的块
 - 物理文件：文件存储器上的文件
 - 物理记录：存放文件记录的物理块
 - 文件物理结构
 - 连续
 - 链接
 - 索引
- 连续文件
 - 把逻辑上连续的文件信息存储在连续的物理块上的一种组织方式
 - 优点
 - 实现简单
 - 只需要记住文件的第一块所在位置及文件所包含的块数即可
 - 存取速度快
 - 支持顺序访问和随机访问
 - 缺点
 - 不灵活：要求在文件创建时，就给出文件的最大长度，文件不能动态扩展
 - 产生碎片：文件被删除时，文件存储空间可能出现许多小的无法利用的碎片
- 链接文件
 - 把逻辑上连续的信息文件存储在不连续的物理块中，存放信息的物理块中另设一个指针指向下一个物理块。文件最后一个物理块的指针通常为0，以指示该块是链尾
 - 优点
 - 不存在外部碎片
 - 有利于文件插入和删除
 - 有利于文件动态扩展
 - 缺点
 - 链接需要额外空间
 - 只能顺序访问
 - 存取速度较慢
 - 改进
 - 把指针字从文件的各物理块中取出，放在一张表中，此表叫做盘文件映射表
 - MS-DOS就使用这种方式分配和管理磁盘空间，并将该表叫做文件分配表FAT
 - 优点
 - 既能方便地实现顺序存取，而且也很容易实现随机存取
 - 缺点
 - 不缓存FAT，将导致大量寻道时间
 - 缓存FAT，占用太多内存
- 索引文件
 - 在文件目录表中为每个文件保留一个索引表块号，该索引块指出文件的逻辑块与物理块之间的映射关系
 - 优点
 - 允许文件动态修改，增加了使用的灵活性

- 允许用户按照要求，直接对文件进行存取
- 缺点
 - 索引表要占用额外空间
 - 降低了文件存取速度
 - 至少需要访问存储器二次
 - 一次访问索引表
 - 一次访问文件信息
- 多级索引
- 混合索引

5.5.2 文件的存储介质

- 文件存储设备
 - 磁带
 - 顺序存取设备
 - 磁盘、光盘
 - 直接存取设备
 - 特点
 - 存取容量大
 - 存取速度高
 - 以块为单位进行存取访问
- 文件存储介质的主要参数
 - 容量
 - 物理记录尺寸
 - 可拆卸性
 - 潜在时间
 - 寻找时间
 - 传输速度
- DOS文件卷的结构
 - 硬盘低级格式化
 - 将磁盘划分位若干磁道、扇区，有生产厂家完成
 - 扇区的头标记录其柱面号、磁头号和扇区号
 - 硬盘分区(FDISK命令)
 - 硬盘主引导扇区
 - 硬盘的0面0道1扇区，属于整个硬盘而不属于某个独立分区
 - 硬盘主引导程序：位于该扇区0~1BDH处
 - 硬盘分区表DPT：位于该扇区1BEH-1FDH处
 - 每个分区表占用16B，共4个分区表

16B意义如下：0为自举标志（80H为可引导分区，00H为不可引导分区）；1-3是分区的起始地址（磁头号扇区号柱面号）；4是分区类型（07H为NTFS分区）；5-7是分区的结束地址；8-11是分区首扇区的绝对扇区号；12-15是分区占用的总扇区数
 - 主引导扇区的有效标志：位于该扇区的1FEH-1FFH处，固定值为AA55H

- 分区规范规定
 - 一个硬盘可以有多个主分区，但最多只能有一个扩展分区
 - 一个扩展分区可以划分为多个逻辑分区，所以一个硬盘最多只能划分为4个主分区或者3个主分区和一个扩展分区
 - 扩展分区只作为数据盘使用
- 分区格式化 (FORMAT命令)
 - 对分区进行具体数据组织，即制作文件系统
- DOS卷的组成
 - 引导或保留扇区
 - 文件分配表1/文件分配表2
 - 根目录区
 - 文件数据区

5.6 文件存储器存储空间管理

- 文件存储器存储空间的基本分配单位为磁盘块/簇
- 常用的管理方法
 - 空白文件目录
 - 空闲块链表
 - 位映像表或位示图
- 空白文件目录
 - 空白文件：一个连续未用的空闲盘区块
 - 为所有空白文件建立的一张表，用来记录整个文件存储器的空闲未用空间，每个空白文件占用其中的一个表目
 - 适合于文件的静态分配（连续分配）
- 空闲块链表
 - 适合文件动态分配
 - 将所有的空闲块链接成一个链，在主存保留一个链头指针
 - 存储分配与回收
 - 直接从链头取出相应数量的空闲块分配
 - 释放的空闲块依次放入链头
 - 优点：管理简单
 - 缺点：工作效率低
 - UNIX系统常用
- 成组空闲块链表
 - 利用盘上的空闲块管理空闲块
 - 每个磁盘块记录尽可能多的空闲块而组成一组，各组之间也用链指针链在一起，在主存保留该链表的链头指针
 - 存储分配
 - 将链头指针所指的组块读入内存，从中找到一个空闲块进行分配；当未满足分配要求时，再从链中将记录下一组空闲块的盘块读入内存，再进行分配
 - 回收
 - 将释放的盘块记录在主存的组盘块中即可。若该块已满，除将该块内容写回磁盘外，可用刚释放块记录其它要释放块，并将其放到链头即可
- 位映像表

- 适合文件静态和动态分配的最简单方法
- 使用一个位向量，磁盘中每一块对应其中一位
- 0表示空闲，1表示占用
- 优点
 - 比较容易找到一个或几个空闲块
 - 尺寸固定，位映像表能够保存在主存中
- 转换工作
 - 将位映像表中的字位号转换成相应盘的物理地址
 - 转换
 - 磁盘的相对块号=字节号*n+位号
 - 反转换
 - 字节号=相对块号/n
 - 位号=相对块号%n

5.7 文件共享与保护

- 文件共享
 - 允许多个用户共同使用同一个文件
 - 既节省外存空间，又能减少I/O信息量和主存中的副本
 - 基于索引节点的共享
 - 索引节点：存放文件的物理地址和文件属性等信息
 - FCB：存放文件名和指向索引节点的指针
 - 链接计数
 - 存放在索引节点中
 - 记录链接到本索引节点的用户数目
 - 基于符号链的共享
 - 由系统创建一个LINK类型的新文件，新文件中只包含被链接文件的路径名
 - 只有文件主拥有指向被共享文件索引节点的指针，其他用户只用有路径
 - 文件拥有者删除共享文件后，其他用户试图通过符号链访问该共享文件将失败
 - 优点：可以链接任何文件
 - 缺点：耗费磁盘空间，增加启动磁盘的频率
- 文件保护
 - 防止文件的丢失和破坏
 - 解决方法：复制/恢复
 - 周期性的全量转存
 - 增量转存
 - 防止其他用户对文件的有意破坏或窃取
 - 增设防护措施
 - 防止由计算机病毒引起的文件破坏
 - 用户的鉴别——口令
 - 对文件加密
 - 物理鉴定
- 文件存取控制

- 一个用户对其文件所做的“谁能使用和如何使用”的规定，主要对文件共享加以限制
- 文件的存取控制
 - 防止核准的用户误用文件和未核准的用户存取文件
 - 文件目录中有关文件存取控制信息部分规定了各类用户对各个文件的存取权限
 - 存取控制方法
 - 保护域
 - 规定了进程对一组对象的存取权限
 - 一个域是一组（对象，存取权限）偶对，每一个偶对说明一个对象和允许对其施加的一组操作
 - 一个对象可以在多个保护域中
 - 存取控制表
 - 存取访问矩阵
 - 行：域
 - 列：目标对象
 - 元素：在域中执行的进程能对对象所施加的操作集合
 - 存取控制矩阵的实现
 - 按列存取：存取控制表ACL
 - 按行存取：存取权限表

5.8 文件系统组织结构

- 文件系统是用户和外存设备之间的接口
- 文件系统的层次结构
 - 应用程序接口
 - 检查由用户提供的命令句法的正确性和参数的合法性
 - 逻辑文件系统
 - 负责目录的管理和维护，按照命令给定的文件名查找目录
 - 创建文件：增加目录项
 - 打开文件：检查各级目录，找到相应的目录项
 - 读/写文件：检查文件是否已经以读/写方式打开
 - 文件组织模块
 - 负责将文件的读/写位置转换成文件中的相对块号，进而转换成在存储器中的物理块号
 - 管理磁盘空闲空间
 - 将上层传下来的命令转换成对基本文件系统的调用
 - 基本文件系统
 - 将上层传下来的命令和物理模块转换成对设备驱动程序的调用
 - I/O调度及控制模块
 - 由设备驱动和中断处理模块组成
 - 将上层传下来的命令转换成硬件设备的专用I/O指令和设备地址，控制设备完成和主存之间的信息传输
 - 负责多个读/写命令的排队和调度

5.9 存储区映射文件

- 允许进程分配虚存的一部分地址空间，将磁盘上的一个文件映射到该空间
- 对文件块的存取通过访问虚存的一个页实现，当访问的页不在主存时，产生缺页中断，将其读入主存
- 当需要对存储器映射文件存取时，才实际传输文件数据，使用与请求页式虚存管理相同的存取机制进行传输，不必使用文件命令读写文件
- OS提供映射文件的系统调用

六、设备管理

6.1 I/O硬件组成

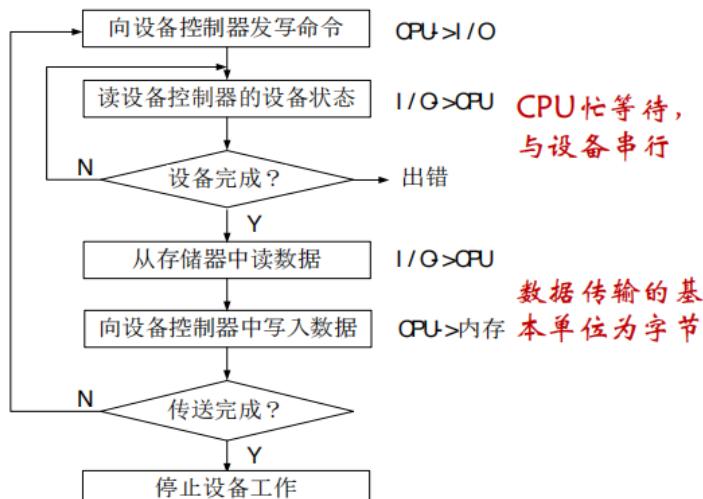
- I/O设备分类
 - 数据传输速率
 - 低速设备：键盘、鼠标
 - 中速设备：打印机
 - 高速设备：磁盘、光盘
 - 设备共享属性
 - 独占设备：慢速的字符设备
 - 共享设备：快速的块设备，资源利用率高
 - 虚拟设备：
在一类设备上模拟另一类设备的I/O技术
 - 将独占设备改造为共享设备（SPOOLing技术）
 - 将低速设备改造为高速设备（虚拟磁盘）
 - 数据传输单位
 - 块设备
 - 以块为单位传输信息
 - 传输速率高、可寻址
 - 磁盘、磁带
 - 字符设备
 - 以字符为单位传输信息
 - 传输速率低、不可寻址
 - 鼠标、键盘、打印机
 - 网络通信设备
- 设备控制器
 - I/O设备由机械和电子两部分组成
 - 机械部分是设备本身
 - 电子部分叫做设备控制器或适配器
 - 功能
 - 控制一个或多个I/O设备，以实现I/O设备和计算机之间的数据交换

- CPU与I/O设备之间的接口，接收CPU发来的命令，并控制I/O设备工作
 - 接收和识别命令
 - 数据交换
 - 标识和报告设备的状态
 - 地址识别
 - 数据缓冲
 - 差错控制

- I/O数据传输的控制方式

- 程序查询方式

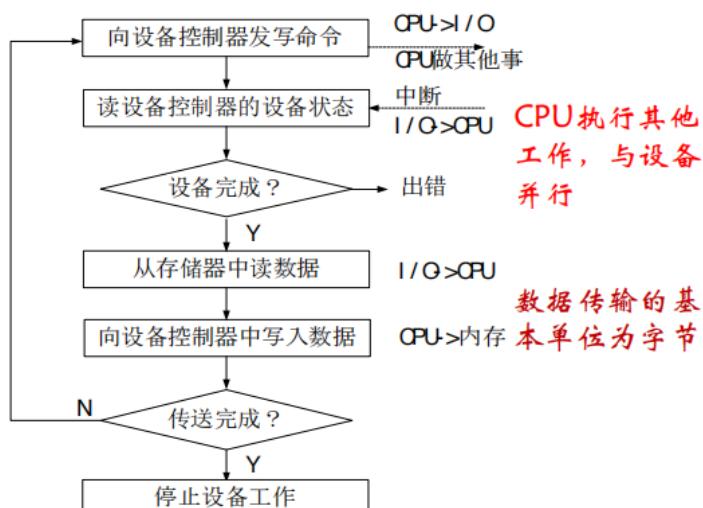
- 循环查询
 - CPU与设备完全串行



程序查询方式工作流程

- 程序中断方式

- 尽量减少主机对I/O控制的干预，将主机从烦杂的I/O控制事务中解脱



程序中断方式工作流程

- 直接存储器访问 (DMA)

- 从以字节为单位扩展到以数据块为单位
 - 特点

- 数据传输的基本单位是数据块
 - 传送数据从设备直接送入内存或相反
 - 仅在传送一个或多个数据块的开始和结束时，才需要CPU干预，整块数据的传送是在控制器的控制下完成的

- 实现
 - 磁盘地址
 - 主存的起始地址
 - 传送的字节数
- 通道方式
 - 引入
 - 进一步减少CPU的干预，将对一个数据块的干预减少为对一组数据块及其相关控制和管理的干预
 - 实现CPU、通道和I/O设备三者的并行
 - 实现方式
 - 接受CPU的委托，独立地执行自己的通道程序，管理和控制输入输出设备，实现外围设备与主存储器之间的成批数据传送。当CPU委托地I/O任务完成后，通道发出中断信号，请求CPU处理
 - 分类 (信息交换方式)
 - 字节多路通道
 - 以字节为单位传输信息，可以分时执行多个通道程序。当一个通道程序控制某台设备传送一个字节后，通道硬件就转去执行另一个通道程序，控制另一台设备传送一个字节地信息
 - 主要用来连接大量慢速地设备
 - 选择通道
 - 以成组方式工作，每次传送一批数据
 - 一段时间内只能执行一个通道程序，控制一台设备进行数据传输。当这台设备数据传输完成后，再选择与通道连接的另一台设备，执行它相应的通道程序
 - 常连接高速设备
 - 数组多路通道
 - 结合了选择通道传送速度高和字节多路通道能进行分时并行操作的优点
 - 它先为一台设备执行一条通道指令，然后自动转接，为另一台设备执行一条通道指令

6.2 I/O软件组成

6.2.1 I/O软件目标

- 提供设备的独立性（设备无关性）
 - 应用程序独立于具体使用的物理设备
 - 独立于设备的类型
 - 独立于同类设备的具体台号
 - 设备的统一命名
 - 设备名不应依赖于设备
 - 在OS中，通常规定用户程序中不直接使用物理设备名（或设备的物理地址），而使用逻辑设备名
 - 逻辑设备表示物理设备属性，它不指某个具体设备，而是对应一类设备
 - 由操作系统根据系统设备情况完成相应的映射

■ 逻辑设备表

逻辑设备名	物理设备名	驱动程序入口地址
/dev/tty	3	1024
/dev/printer	5	2046
⋮	⋮	⋮

(a)

逻辑设备名	系统设备表指针
/dev/tty	3
/dev/printer	5
⋮	⋮

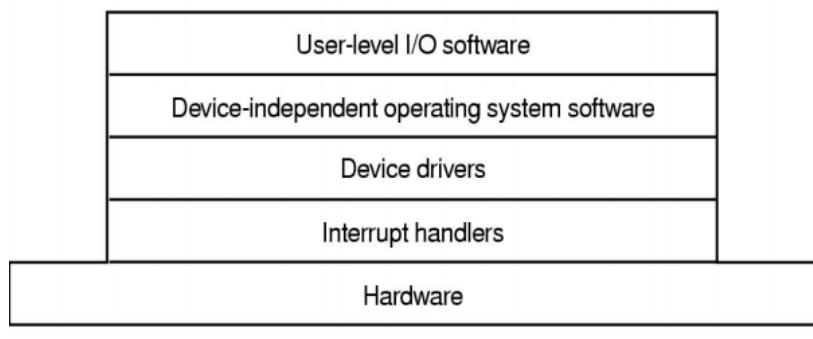
(b) 21

■ 优点

- 设备分配灵活
- 易于实现I/O重定向
- 设备独立软件
- 出错处理
 - 对于数据传输中的错误应尽可能在接近硬件层上处理
 - 仅当低层软件无能为力时，才将错误上交到高层处理
- 缓冲技术
 - 使数据的到达率和离去率相匹配，提高系统的吞吐率
 - 改善I/O设备和CPU之间速度不匹配的情况
 - 减少中断CPU的次数，提高CPU的利用率
 - 减少启动设备的次数，延长设备的寿命
 - 提高CPU和I/O设备之间的并行性
- 设备分配
 - 设备类型
 - 独占设备：静态分配
 - 共享设备：动态设备
 - 虚拟设备：共享设备，FCFS
 - 静态设备
 - 简单，但设备利用率低
 - 动态分配
 - 设备利用率高，但容易引起死锁

6.2.2 I/O软件的组成

- I/O软件的结构
 - 基本思想
 - 按分层思想构成
 - 较低层软件要使较高层的软件独立于硬件的特性
 - 较高层软件要向用户提供一个友好的、清晰的、简单的、功能更强的接口
 - 层次结构
 - 中断处理程序
 - 设备驱动程序
 - 独立于设备的软件
 - 用户空间的I/O软件



- 中断处理程序
 - 中断通常隐藏在操作系统内
 - 每个进程在启动一个I/O操作后阻塞起来，直到I/O操作完成产生一个中断，通过中断向CPU报告，CPU唤醒该进程，对设备进行中断处理
 - 中断处理工作完全由OS完成，用户进程根本不知道中断的产生和处理过程
 - 中断处理工作
 - 唤醒被阻塞的设备驱动程序
 - 保护被中断进程的CPU环境
 - 分析中断原因转入相应处理程序
 - 中断处理
 - 恢复被中断进程的现场
- 设备驱动程序
 - 与设备密切相关的代码放在设备驱动程序中，每个设备驱动程序处理一种设备类型
 - 接收来自与设备无关的上层软件的抽象请求，并执行这个请求
 - 在请求I/O的进程与设备控制器之间的一个通信和转换程序，它将进程的I/O请求经过转换后，传递给控制器，又把控制器中所记录的设备状态和I/O操作完成情况及时地反映给请求I/O的进程
 - 设备驱动程序的处理过程
 - 接收发出I/O请求的进程发来的命令和参数，并将命令中的抽象要求转换为具体要求
 - 检查用户I/O请求的合法性
 - 读出和检查设备的状态
 - 如果设备空闲则立即启动I/O设备完成指定的I/O操作
 - 如果设备处于忙碌状态，则将请求者的请求块挂在设备队列上等待
 - 传送必要的参数并设置设备工作方式
 - 启动I/O设备
 - 阻塞自己，直至中断到来时被唤醒，并根据中断类型调用相应的中断处理程序进行处理
 - 设备处理方式
 - 为每一类设备设置一个I/O进程
 - 为整个系统设置一个I/O进程
 - 不设置专门的设备处理进程，只为各类设备设置相应的设备处理（驱动）程序，供用户进程或系统进程调用
- 设备独立的软件
 - 基本任务
 - 实现所有设备都需要的功能，并向用户级软件提供一个接口
 - 设备命名
 - 将设备的逻辑名映射为物理设备名。即把设备的符号名映射到正确的设备驱动程序上

- 设备保护

- 防止无权存取设备的用户存取设备，保证有权使用的用户正确使用设备
- 为每一个设备设置正确的存取权
- 禁止用户直接访问设备（通过系统调用访问）

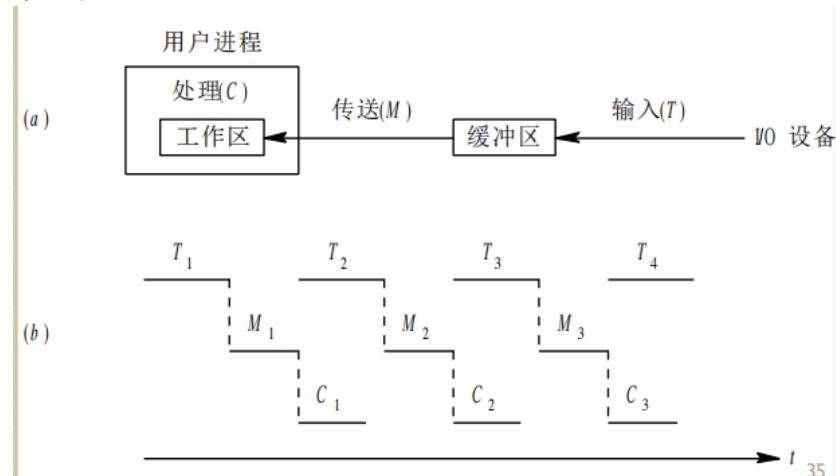
- 提供与设备无关的块尺寸

- 向较高层软件掩盖不同磁盘采用不同扇区尺寸的事实，并提供大小统一的块尺寸
- 较高层的软件只与抽象设备打交道，使用等长的逻辑块，独立于物理扇区尺寸

- 缓冲技术

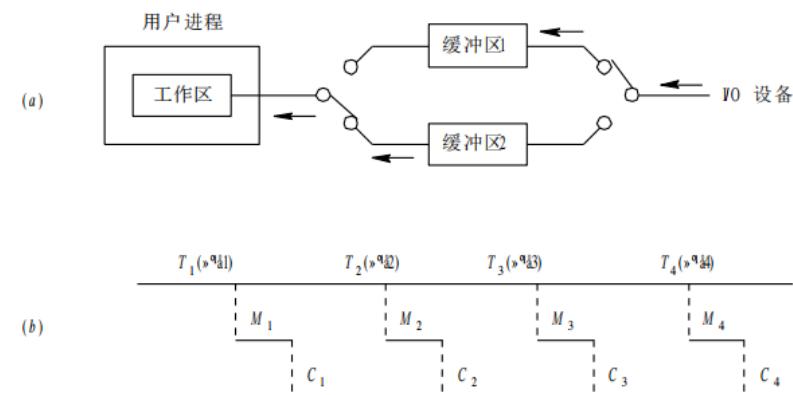
- 对设备缓冲区进行有效管理，提高I/O效率

- 单缓冲



- 假定从磁盘将一块数据输入到缓冲区的时间为T，OS将缓冲区的数据传送到用户区的时间为M，CPU对一块数据处理的时间为C，则
- 系统对每块数据的处理时间为 $\text{Max}(C, T) + M$

- 双缓冲



- 系统对每块数据的处理时间为 $\text{Max}(C+M, T)$

- 缓冲池

- 空缓冲区 (队列)
- 输入缓冲区 (队列)

- 输出缓冲区 (队列)

缓冲队列是临界资源

- 设备分配

- 分配程序

- 分配设备
 - 分配控制器
 - 分配通道

- 数据结构

- 设备控制表
 - 系统设备表
 - 控制器控制表
 - 通道控制表

- 考虑因素

- 设备固有属性: 独占, 共享, 虚拟
 - 设备分配算法: FCFS, Priority
 - 设备分配的安全性

- 出错处理

- 一般由设备驱动程序实现

- 用户空间的I/O软件

- 系统调用, 包括I/O系统调用, 通常由库过程实现

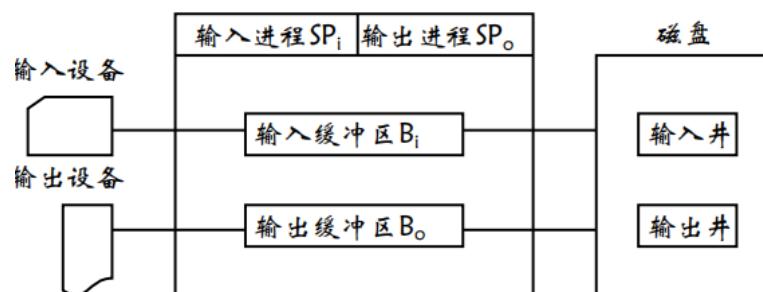
- 这些过程所做的工作只是将系统调用时所用的参数放在适当的位置, 实际由I/O过程实现真正的操作

- SPOOLing系统, 假脱机

- 联机情况下实现的同时外围操作
 - 将独占设备改造成可共享设备
 - 提高了独占设备利用率
 - 加快了进程的执行速度
(以空间换时间)

- 构成

- 输入井和输出井
 - 输入缓冲区和输出缓冲区
 - 输入进程SP_i和输出进程SP_o



- 工作原理

- 井管理程序

- 负责管理输入和输出缓冲区，记录每个缓冲区作用
- 当作业执行过程中要求启动某台设备输入或输出时，操作系统截获该请求并调出井管理程序，控制从相应的输入井读或向共享设备的输出井写。输入井和输出井上的信息是以文件的形式记录的
- 预输入程序
 - 负责将输入设备上的信息预先输入到可共享设备的缓冲区域（输入井）并将存放信息的位置记录下来待用
- 缓输出程序
 - 负责从磁盘缓冲区读信息向输出设备输出。当设备空闲或进程完成后，系统负责调用缓输出程序将各用户的信息依次从输出井送设备输出

【例：打印机】

- 打印机不能接收各进程的交叉数据流，为了使多用户进程能并发打印它们的输出，又使其输出不交叉在一起：
- 当进程要求输出打印时，系统并不为它分配打印机设备，而是分配可共享设备（如硬盘）的一部分空间，使得每个进程的输出被缓冲到一个独立的磁盘文件上
- 当进程完成执行时，再将相应的被缓冲的文件进行排队，准备打印输出
- 之后，Spooling系统一次一个地将排队地待打印文件复制给打印机进行实际地打印

【I/O软件工作实例】

- 用户进程读取一个文件中的一块信息
- 处理过程
 - 用户进程发出一个读文件的系统调用
 - 设备独立软件
 - 检查参数的正确性，若正确，检查内存缓存中有无要读的信息块
 - 有，从缓冲区中直接将信息返回用户
 - 无，执行物理I/O
 - 独立于设备的I/O软件设备的逻辑名转换成物理名，检查设备操作权限
 - 将I/O请求排队，用户进程阻塞等待磁盘操作的完成
 - 调用设备驱动程序，向I/O硬件泄放一个读请求
 - 分配缓冲区，准备接收数据，并向设备控制寄存器发启动读命令
 - 设备控制器控制设备，执行数据传输
 - 当磁盘将所需块读入缓冲区时，硬件产生一个中断
 - 系统响应中断后，转入中断处理程序
 - 中断处理程序从设备获取所需的状态信息，检查中断产生的原因
 - 若正常完成，将数据传输给指定的用户进程空间，唤醒等待该I/O完成的进程，将其放入就绪队列，等待调度
 - 若出错，则向该设备驱动程序发信号，若可重试，则再启动该设备重传一次；否则，向上报告错误
 - 用户进程继续运行

6.2.3 同步I/O和异步I/O

- 同步I/O：进程发出I/O请求后阻塞等待，直到数据传输完成后被唤醒，访问被传输的数据
- 异步I/O：允许进程发出I/O请求后继续运行
 - I/O完成后的通知方式：
 - 设备进程地址空间内的某个变量；
 - 通过触发信号或软件中断；
 - 进程执行流之外的某个回调函数
- 对于不必进行缓冲读写的快速I/O，使用同步I/O更有效；对于需要很长时间的I/O操作，可使用异步I/O操作

6.3 磁盘管理

6.3.1 磁盘结构

- HDD的物理结构
 - 磁盘由若干盘片组成，每片分为两个表面，表面上覆盖着磁性材料，用于记录信息
 - 磁盘表面又划分为磁道，磁道由一组同心圆构成（每条磁道存储的相同数目的二进制位，内层磁道密度较外层磁道密度高）
 - 每条磁道又分为若干个扇区
 - 磁盘的每个表面都定位一个读/写磁头，负责将信息读出或写入磁道
 - 磁盘系统的硬件分为两个部分：磁盘驱动器和磁盘控制器
- HDD的类型
 - 硬盘、固态硬盘
 - 单片盘、多片盘
 - 固定头磁盘
 - 每个磁道有一个磁头，磁道间的转换非常迅速，I/O速度快，但需要大量的磁头，设备成本很高，应用于大容量磁盘
 - 浮动头磁盘
 - 在一个磁盘表面只安装一个磁头，通过移动磁头可方便地存取不同磁道上的信息，要求专门的硬件装置来移动磁头，设备成本降低，但I/O速度较慢，应用于中小型磁盘
- HDD的访问时间
 - 磁盘上的信息通过多重编址定位
 - 定位信息
 - 硬盘：驱动器号、柱面号、磁头号及扇区号
 - 信息是按块存储的，每个块（称之为扇区）由硬件指定大小
 - 扇区大小一般为4KB
 - 为了存取磁盘上的一个扇区，磁盘的速度由三部分时间组成
 - 寻道时间：系统移动磁头至响应的磁道或柱面上的时间
 - 旋转时间：磁头到达指定磁道后，等待所需的扇区旋转到读/写头下的时间
 - 传输时间：数据在磁盘与主存之间实现数据传输所用时间
 - 服务一个磁盘请求的总时间是上述三者之和
- NVD的物理结构
 - NVD设备是电子设备

- 主要由控制器和用于存储数据的闪存NAND芯片组成
- 其他NVD技术包括DRAM等
- 固态磁盘 (SSD)
 - 比HDD更可靠
 - 无需寻道，访问速度比较快
 - 相对于HDD，容量较小，每MB价格较贵
- NVD限制
 - 可以以“页”（类似于扇区）来读和写，但不能覆盖数据，只能先擦除NAND单元
 - 擦除以几页大小的“块”增量发生，比读取（最快的操作）或写入（比读取的速度慢，但比擦除的速度快）花费更多的时间
 - 其生命期以“每日驱动器写数DWPD”为单位

6.3.2 磁盘调度

- HDD调度
 - 目标：使磁盘的平均寻道时间最少
 - 调度算法
 - 先来先服务FCFS
 - 优点：容易实现，公平合理
 - 缺点：完全不考虑队列中各个请求情况，致使磁头频繁地移动
 - 最短寻道时间优先SSTF
 - 在将磁头移向下一请求磁道时，总是选择移动距离最小的磁道
 - SSTF算法虽然比FCFS算法优越，但不是最优的算法
 - 可能导致饥饿问题
 - 扫描法SCAN
 - 读/写头开始由磁盘的一端向另一端移动时，随时处理所到达的任何磁道上的服务请求，直到移动到磁盘上的另一端为之；在磁盘的另一端上，磁头的方向反转，继续完成各磁道上的服务请求
 - 磁头总是连续不断地从磁盘的一端移动到另一端
 - 在使用SCAN前，不仅要直到磁头移动到最后位置，而且还需要直到磁头移动方向
 - 也叫电梯算法
 - 循环扫描法C-SCAN
 - 将磁头从一端移向另一端时，随时处理到达的请求。但是，当它到达另一端时，磁头立即返回到开始处。也即回程，不处理任何请求
 - 查询法LOOK
 - 扫描法和循环扫描法都是将磁头由磁盘的一端移向另一端，但实际上没有一个算法是这样实现。通常，磁头在向任何方向移动时都是只移到最远的一个请求的磁道上。一旦在前进的方向上没有请求到达，磁头就反向移动
 - 即在将磁头向前移动之前，先查询有无请求，若有，才移动，否则，立即反向
 - 循环查询法C-LOOK
 - 磁盘调度算法的比较
 - SSTF算法是公认的、最具吸引力的算法
 - SCAN和C-SCAN对于磁盘负载较重的系统更为合适
 - 任何调度算法性能优劣都与进程对磁盘的请求数量和方式紧密相关；
 - 当磁盘等待队列中的请求数量很少时，所有算法等效，这种情况下，最好采用FCFS算法

Linux实例

七、Linux进程管理

7.1 Linux进程的组成

7.1.1 进程定义

- 进程控制块/进程描述符
 - 进程是程序的一个执行过程。
 - 每个进程都有一个独立的地址空间
 - 在用户态运行时，进程映像包含有代码段、数据段和私有用户段等。
 - 在核心态运行时，访问内核的代码段和数据段，并使用各自的核心栈
 - 进程控制块部分信息

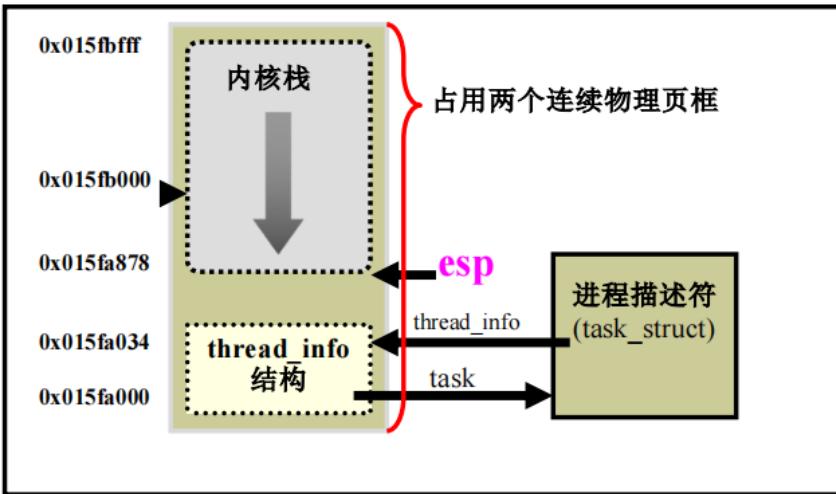
//Linux中的PCB中的部分信息

```
struct task_struct {  
    pid_t pid;                      //进程标识符(32767)  
    struct thread_info *thread_info; //当前进程基本信息  
    long state;                     //进程状态  
    int prio;                       //进程优先级0-139  
    int static_prio;                //进程的静态优先级  
    unsigned long rt_priority;      //进程的实时优先级  
    struct list_head tasks;         //进程链表  
    struct task_struct *parent;     //指向父进程  
    struct list_head children;     //子进程链表  
    struct files_struct *files;    //打开文件列表  
    struct mm_struct *mm, *active_mm; //指向进程拥有和执行  
                                    //的虚拟内存描述符  
    .....  
};
```

- 进程基本信息

```
struct thread_info {  
    struct task_struct* task;        //进程描述符指针  
    unsigned long flags;           //重调度标识  
    struct exec_domain exec_domain;  
    int preempt_count;             //软中断计数器  
    __u32 cpu;  
    struct restart_block restart_block;  
};
```

- 进程核心栈、thread_info结构和进程描述符之间的关系



- Linux把每个进程的内核栈和基本信息thread_info存放在两个连续的页框（8KB）中。考虑到效率，内核让其第一个页框的起始地址是 2^{13} 的倍数
- 由于esp寄存器存放的是核心栈的栈顶指针，内核很容易从esp寄存器的值获得正在CPU上运行进程的thread_info结构的地址，进而获得PCB的地址
- 进程刚从用户态切换到内核态时，其内核栈为空，只要将栈顶指针减去8K，就能得到thread_info的地址

7.1.2 进程状态

- state表示进程生命期中的状态：

- 可运行态：进程正在或准备在CPU上运行的状态
- 可中断的等待状态：进程睡眠等待系统资源可用或收到一个信号后，进程被唤醒
- 不可中断的等待状态：进程睡眠等待一个不可中断的事件发生（很少使用）

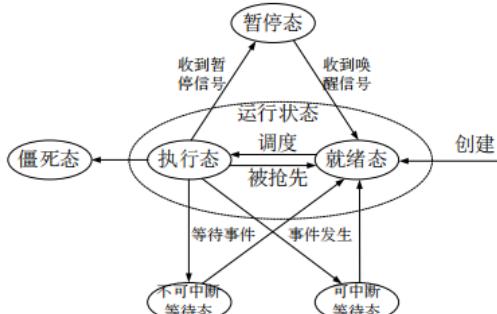
e.g. 进程打开一个设备文件时，其相应的设备驱动程序开始探测硬件设备。探测完之前不能被中断

- 暂停状态
- 跟踪状态

- exit_state表示进程的退出状态：

- 僵死态：进程已终止，等待父进程处理
- 死亡态：系统删除该进程

- Linux进程状态转换图



Linux进程状态转换图

7.2 Linux进程链表

- 传统进程链表list_head
 - 所有进程链表tasks
 - 链表头是0号进程(idle进程)
 - 可运行进程链表run_list
 - 双向链表
 - 子进程链表
 - 兄弟进程链表
 - 等待进程链表
 - 互斥等待临界资源的进程：一次唤醒一个
 - 非互斥等待的进程：唤醒所有进程
- 哈希链表
 - 内核定义了四类哈希表
 - PIDTYPE_PID链表：进程
 - PIDTYPE_TGID链表：线程组
 - 同一线程组中的所有轻量级进程的tgid值相同
 - PIDTYPE_PPID链表：进程组
 - PIDTYPE_SID链表：会话
 - 哈希表地址存入pid_hash中

7.3 Linux进程控制

7.3.1 进程创建

- 父子进程是2个完全独立的进程，拥有不同的pid与虚拟地址空间，但内存、寄存器、程序计数器等状态都完全一致
- Linux创建进程函数fork()、clone()、vfork()
 - 创建子进程函数fork()：创建成功后，子进程采用写时复制计数读共享父进程的全部地址空间
 - 创建轻量级进程函数clone()：实现对多线程应用程序的支持。共享进程在内核的很多数据结构，如页表、打开文件表等等
 - vfork()：阻塞父进程直到子进程退出或执行了一个新进程为止
- Linux创建内核线程函数kernel_thread()
 - 0号进程是一个内核线程，0号进程是所有进程的祖先进程，又叫idle进程或叫做swapper进程。每个CPU都有一个0号进程
 - 1号进程是由0号进程创建的内核线程init，负责完成内核的初始化工作。在系统关闭之前，init进程一直存在，它负责创建和监控在操作系统外层执行的所有用户态进程

7.3.2 进程撤销

- 进程终止
 - exit()系统调用只终止某一个线程
 - exit_group()系统调用能终止整个线程组
- 进程删除
 - 父进程先结束的子进程会成为孤儿进程，系统会强迫所有的孤儿进程成为init进程的子进程
 - init进程在用wait()类系统调用检查并终止子进程时，就会撤销所有僵死的子进程

7.4 Linux进程切换

- 上下文
 - 硬件上下文：装入CPU寄存器的一组数据
 - 软件上下文：Linux2.6以后实现
- 进程切换只发生在内核态
 - 在发生进程切换之前，用户态进程使用的所有寄存器值都已经被保存在进程的内核栈中
 - 之后，大部分寄存器值存放在PCB的类型为thread_struct的thread字段（进程硬件上下文）里，一小部分仍在内核栈中
- 进程切换步骤
 - 切换页目录表以安装一个新的地址空间
 - 切换内核栈和硬件上下文，由schedule()函数完成切换

7.5 Linux进程调度

- Linux调度
 - Linux2.4版本之前，调度设计较为简单，采用基于RR策略的运行队列
 - Linux2.4版本，O(n)调度
 - 选取所有任务中动态优先级最高的任务进行调度
 - 在运行时计算动态优先级，保证实时任务的优先级高于非实时任务优先级
 - 采用负载分担思想，所有任务（数量为n）存储于一个全局运行队列
 - 缺点：开销大，多核扩展性差
 - Linux2.6.0，O(1)调度
 - 采用两级调度思想
 - 采用抢占的基于优先级的调度算法
 - 两个独立的优先级范围
 - real-time: 0-99，静态优先级（实时进程）
 - nice: 1000-139，动态优先级（交互式和批处理进程）
 - 数值越低，优先级越高
- 普通进程的调度
 - 基本时间片
 - 给较高的优先级分配较长的时间片，给较低的优先级分配较短的时间片

- 静态优先数与时间片的关系：

静态优先数<120

- 基本时间片 = $\max((140 - \text{静态优先级}) * 20, \text{MIN_TIMESLICE})$

静态优先数>=120

- 基本时间片 = $\max((140 - \text{静态优先级}) * 5, \text{MIN_TIMESLICE})$

- 动态优先级

- 调度程序根据进程使用CPU的情况动态调整进程的优先级
- 提高长时间未获CPU进程的优先级
- 降低运行较长的进程的优先级
- 动态优先数计算方法：

$\max(100, \min(\text{静态优先数} - \text{bonus} + 5, 139))$

bonus取值范围是0-10, >5表示降低动态优先级, <5表示提升动态优先级, 其值取决于进程过去的行为

- 每个运行队列包括两个优先级队列

- 活动的：在其时间片中还有剩余时间的任务
- 到期的：一个任务耗尽时间片后，被认为到期了
- 当所有任务都耗尽其时间片（即活动队列为空），两个优先级队列相互交换

- 完全公平调度(CFS)

- 基本思想

- CFS调度程序并不采用严格规则来为一个优先级分配某个长度的时间片，而是为每个任务分配一定比例的CPU处理时间
- 每个task分配的具体比例是根据nice值（权重）来计算的
 - nice值得范围从-20到+19, 数值较低的nice值表示较高的优先级（权重较大）
 - 具有较低nice值的任务，会得到较高比例的处理器处理时间

- 运行时间计算

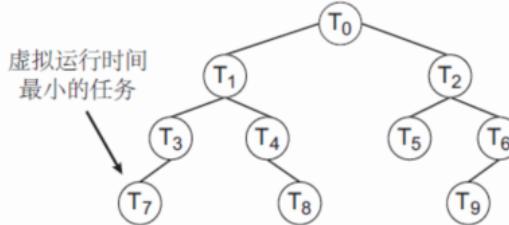
- 分配给进程的运行时间 = 调度周期 * 进程权重 / 所有进程权重之和
 - 调度周期：将所有处于TASK_RUNNING态进程都调度一遍的时间
 - 进程的权重越大，分配到的运行时间越多
- 虚拟运行时间virtual runtime(vruntime)
 - 记录进程已经运行的时间，但是并不是直接记录，而是要根据进程的权重将运行时间放大或者缩小一个比例
 - 记录运行时间的增长速度
 - $vruntime = \text{实际运行时间} * \text{NICE_0_LOAD} / \text{进程权重}$
 $= \text{实际运行时间} * 1024 / \text{进程权重}$

NICE_0_LOAD=1024, 表示nice值为0的进程权重

- 进程权重越大，运行同样的实际时间，vruntime增长得越慢
- 将分配给进程的运行时间带入虚拟运行时间
 - $vruntime = (\text{调度周期} * \text{进程权重} / \text{所有进程总权重}) * 1024 / \text{进程权重}$
 $= \text{调度周期} * 1024 / \text{所有进程总权重}$
 - 一个进程在一个调度周期内的vruntime值大小应该与该进程自己的权重无关，所有进程的vruntime大小相同

- 利用vruntime来选择运行的进程

- 谁的vruntime值较小就说明它以前占用cpu的时间较短，受到“不公平”对待，因此下一个运行进程就是它
- 这样既能公平选择进程，又能保证高优先级进程获得较多的运行时间
- CFS使用红黑树作为运行队列
 - 每个可运行的任务放置在红黑树上（一种平衡二叉查找树，其键基于虚拟运行时间）
 - 当一个任务编程可运行时，被添加到树上；当一个任务变成不可运行时，从树上删除
 - 得到较少处理时间的任务（vruntime较小）会偏向树的左侧；得到较多处理时间的树会偏向树的右侧
 - 根据二叉查找树的性质，最左侧的节点有较小的键值；从CFS调度程序角度而言，这也是具有较高优先级的任务



7.6 内核同步

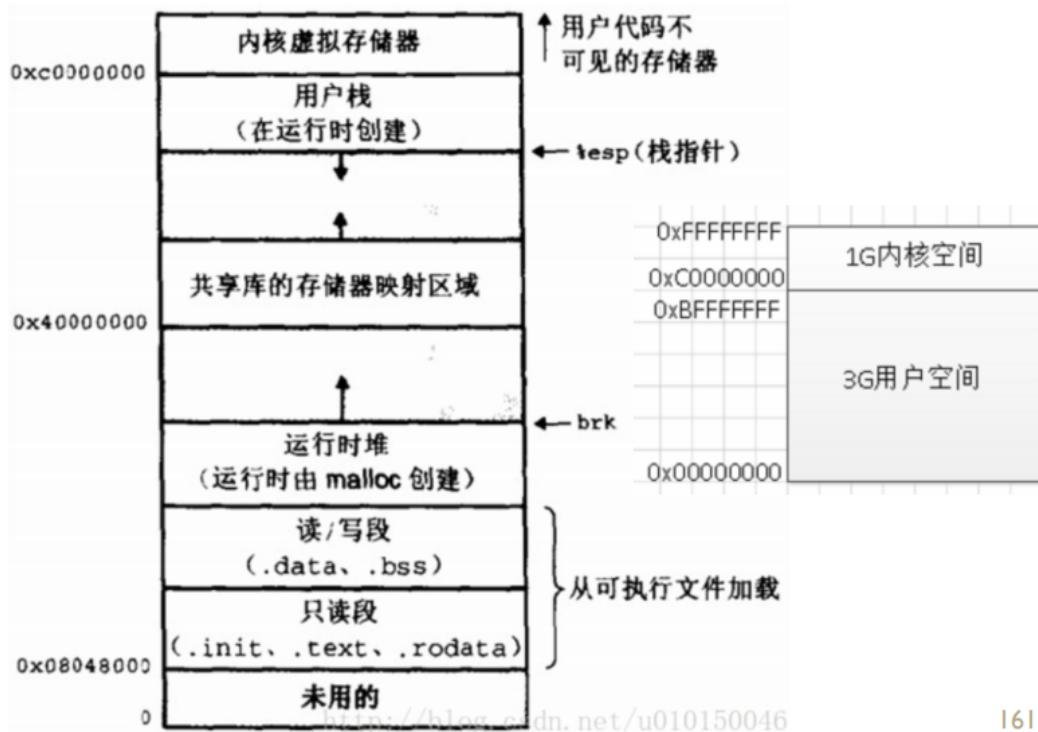
- 互斥访问内核共享数据结构
- 内核使用的同步技术
 - 每CPU变量
 - 把内核变量声明为每CPU变量，一个CPU不应该访问与其他CPU对应的变量
 - 为来自不同的CPU的并发访问提供保护
 - 问题：异步函数、内核抢占
 - 原子操作
 - 优化和内存屏障
 - 自旋锁
 - 用于多处理机环境的一种特殊锁
 - 当一个线程在获取锁的时候，如果锁已经被其他线程获取，那么该线程将循环等待，然后不断地判断锁是否能够被成功获取，直到获取到锁才会退出循环
 - 读-复制-更新
 - 信号量
 - 禁止本地中断
 - 禁止和激活可延迟函数

八、Linux存储器管理

8.1 进程地址空间管理

8.1.1 Linux虚拟地址空间布局

- Linux进程使用虚拟地址空间，由低地址到高地址分别为
 - 只读段：包括代码段、rodata段（C常量字符串和#define定义的常量）
 - 数据段：保存全局变量、静态变量的空间
 - 堆：动态内存（malloc/new分配）
 - 文件映射区域：如动态库、共享内存等映射物理空间的内存，一般是mmap函数所分配的虚拟地址空间
 - 栈：用于维护函数调用的上下文空间，一般为8M
 - 内核虚拟空间：用户代码不可见的内存区域，由内核管理（页表就存放在内核虚拟空间）



161

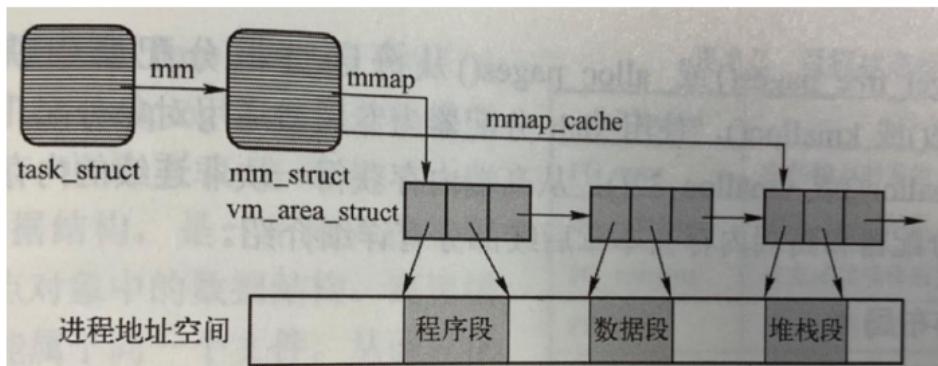
- 32位系统，每个进程的地址空间为4GB，是线性地址空间
 - 每个进程的私有地址空间（用户空间）是前3G，即0x08048000-0xbfffffff
 - 内核进程没有用户空间
 - 进程的公有地址空间（内核空间）是后1G，即0xc0000000-0xffffffff
 - 内核线性地址空间由所有进程共享，但只有运行在内核态的进程才能访问，用户空间可以通过系统调用切换到内核态访问内核空间
 - 进程运行在内核态时所产生的地址都属于内核空间
- 64位系统，每个进程的地址空间为 2^{48}
 - 不需要 2^{64} 这么大的寻址空间
 - 一般使用48位来表示虚拟地址空间，40位表示物理地址

8.1.2 进程地址空间结构

- 每个进程结构有一个mm_struct结构（虚拟内存描述符）管理该进程的地址空间

```
struct mm_struct {           //虚拟内存描述符
    struct vm_area_struct *mmap; //指向虚拟内存区域
                                //链表表头
    struct rb_root mm_rb;      //指向红-黑树的根
    pgd_t *pgd;               //指向页目录表
    atomic_t mm_users;         //次使用计数器
    atomic_t mm_count;         //主使用计数器
    int map_count;             //进程拥有的VMA个数
    struct list_head mmlist;   //内存描述符双向链表
    unsigned long start_code, end_code; //可执行代码
                                    //占用的地址区间
    unsigned long start_brk, end_brk; //堆起始和结束地址
    .....
}
```

- 每个分配的内存块由一个vm_area_struct结构表示，所有的vm_area_struct结构构成一个单链表/红黑树，开始于mm_struct的mmap



8.1.3 虚拟内存区域

- 虚拟内存区域的组织
 - 单链表
 - 把所拥有的各个虚拟内存区域按照地址递增顺序链接在一起
 - 默认情况下，一个进程最多可以拥有65536个不同的虚拟内存区域
 - 红黑树
 - 虚拟内存区域较多时，为提高效率使用红黑树管理虚拟内存区域
 - 单链表+红黑树
 - 当插入或删除一个虚拟内存区域时，内核通过红黑树搜索其相邻节点，并用搜索结果快速更新单链表
 - 红黑树用来快速确定含有指定地址的虚拟内存区域
 - 单链表用来扫描整个虚拟内存区域集合
- 虚拟内存区域访问权限

- 每个虚拟内存区域是由一组连续的页组成，具有相同的访问方式
- 在进程访问该区域中的某个页时，才为其建立页表，且由虚拟内存区域描述符 vm_area_struct 的 vm_flags 字段给出其允许访问方式等标识
- 分配/释放虚拟内存区域
 - 进程创建或映射文件时，分配虚拟内存区域
 - do_mmap(file,addr,len,long prot,flag,offset);
 - 进程完成或取消文件映射时，释放虚拟内存区域
 - do_munmap(mm,start,len);

8.1.4 创建进程的地址空间

- 内核调用 copy_mm() 函数建立新进程的页表和内存描述符，以此创建其地址空间
- 通常每个进程都有独立的地址空间
 - clone() 系统调用提供创建线程的功能
 - 通过传递一组标志，决定父任务和子任务之间发生多少共享

标 志	含 义
CLONE_FS	共享文件系统信息
CLONE_VM	共享内存空间
CLONE_SIGHAND	共享信号处理程序
CLONE_FILES	共享打开文件集合

- exit_mm() 函数释放进程的地址空间

8.1.5 堆的管理

- 每个进程都拥有一个特殊的虚拟内存区域——堆(heap)
- 堆用于满足进程的动态内存请求
- malloc/free

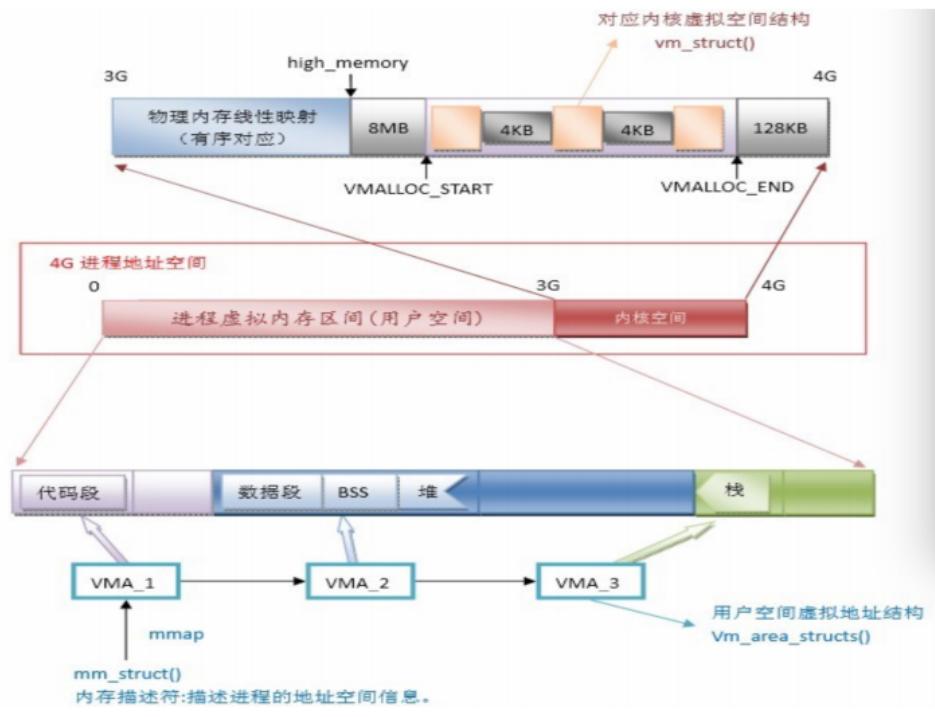
8.2 物理内存管理

8.2.1 物理内存布局

- 页框大小为4KB
- 页框0由BIOS使用，存放加电自检期间检查到的系统硬件配置
- 从0x000a0000到0x000fffff的物理地址通常留给BIOS例程
- Linux跳过RAM的第一个MB的空间，以避免将内核装入一组不连续的页框中
- Linux内核安装在RAM的从物理地址0x00100000开始的地方

8.2.2 物理内存布局

- 内存空间的3个管理区
 - ZONE_DMA: 包含低于16MB的常规内存页框，用于堆老式的基于ISA的DMA支持
 - ZONE_NORMAL: 包含高于16MB且低于896MB的常规内存页框
 - ZONE_HIGHMEM: 包含从896MB开始的高端物理页框，内核不能直接访问这部分页框，在64位体系结构上，该区总是空的
- ZONE_DMA+ZONE_NORMAL属于直接映射区
 - 虚拟地址=3G+物理地址，
从该区域分配内存不会除法页表操作来建立映射关系
 - 是为了保证能够申请到物理地址上连续的内存区域
- ZONE_HIGHMEM属于动态映射区
 - 128M (1024M-896M) 虚拟地址空间可以动态映射到(X-896)M(其中X为物理内存大小)
的物理内存，从该区域分配内存需要更新页表来建立映射关系
 - 但是动态映射区带来了很高的灵活性（比如动态建立映射，缺页时才去加载物理页）
- 高端内存
 - 借助128MB高端内存地址空间可以访问所有物理内存
 - 基本思想
 - 借一段地址空间，建立临时地址映射，用完后释放，达到这段地址空间可以循环使用，
访问所有物理内存
 - 内核将高端内存划分为3部分
 - VMALLOC_START-VMALLOC_END
 - KMAP_BASE-FIXADDR_START
 - FIXADDR_START-4G
 - 高端内存映射的3种方式
 - 映射到“内核动态映射空间”
 - vmalloc(), 给内核分配一个非连续的内存区
 - 永久内核映射
 - 允许内核建立高端页框到内核空间的长期映射
 - 使用内核页表中的一个专门页表实现
 - 临时内核映射



- 内存管理区的数据结构zone

```
struct zone {
    unsigned long free_pages;      // 管理区中的空闲页框数
    struct free_area free_area[MAX_ORDER]; // 伙伴系统
                                         // 中的11个空闲页框链表
    struct list_head active_list;   // 活动页框列表
    struct list_head inactive_list; // 非活动页框列表
    unsigned long nr_active;       // 活动页框列表中页框数
    unsigned long nr_inactive;    // 非活动页框列表中页框数
    spinlock_t lru_lock;          // 自旋锁
    struct page *zone_mem_map;    // 指向管理区中
                                 // 首页框描述符的指针
    .....
};
```

8.2.3 物理页框管理

- 物理页框大小为4KB
- 内核记录每个物理页框的当前状态
 - 哪些页框属于进程
 - 哪些页框是内核代码或内核数据页
 - 哪些页框是空闲页框
 - 使用一个类型为struct page的页框描述符来记录页框的当前信息

- 所有页框描述符存放在mem_map数组中

```

struct page {                                //页框描述符
    unsigned long flags;          //页框状态标志
    atomic_t _count;             //页框的引用计数
    atomic_t _mapcount;          //页框对应的页表项数目
    unsigned long private;        //空闲时由伙伴系统使用
    struct address_space *mapping; //用于页高速缓存
    pgoff_t index;               //在页高速缓存中以页为单位偏移
    struct list_head lru;         //链入活动/非活动页框链表
    void *virtual;               //页框所映射的内核虚地址
};
```

- 分区页框分配器
 - 负责处理对连续物理页框的分配请求
 - 管理区分配器
 - 负责接收动态内存的分配与释放请求
 - 保留的每CPU页框高速缓存
 - 为提高性能，系统为每个CPU预先分配的一些页框，以满足本地CPU发出时的对单个页框的请求
 - 伙伴系统
 - 管理连续的空闲内存页框

8.2.4 伙伴系统

- 伙伴系统
 - 从物理上连续的大小固定的内存上分配
 - 内存按2的整数次幂大小进行分配
 - 如果请求大小不是2的整数次幂，则调整到下一个更大的整数次幂
 - 如果需要的空间比当前可用空间小，则当前可用空间一分为二，直至满足需要空间
 - 如果需要的空间比当前可用空间大，则合并可用空间

8.2.5 slab分配

- slab分配
 - 为只有几十或几百个字节的小内存区分配内存
 - slab是由一个或多个物理上连续的页组成的空间
 - 当需要创建一个新的slab时，slab分配器通过分区页框分配器为该slab获得一组连续的空闲页框
 - slab分配器为不同类型的对象生成不同的高速缓存cache，每个cache存储相同类型的对象。一个新创建的cache没有包含任何slab，也没有空闲对象。但cache并非由各个对象直接构成

成，而是由一连串slab构成，每个slab包含了若干个相同类型的对象

- 分配方法
 - 创建cache时包括标记为空闲的若干对象，对象数量与slab大小相关
 - 当需要内核数据结构对象时，可直接从cache获取，并将该对象标记为使用
- Linux的slab分配
 - slab的状态
 - 满：slab中的所有对象标记为使用
 - 空：slab中的所有对象标记为空闲
 - 部分：slab中的对象部分标记为空闲，部分标记为使用
 - slab分配器首先从部分空闲的slab开始分配，如果没有则从空的slab进行分配，如果没有空slab则从连续物理块上分配新的slab，并把它赋给一个cache，然后再从新的slab分配空间
- 优点
 - 没有因碎片引起的内存浪费
 - 内存请求可以快速得到满足

8.3 地址转换

- 32位处理机普遍采用二级页表模式，为每个进程分配一个页目录表
- 页表一直推迟到访问页时才建立，以节约内存
- 页表项

页表项中的字段	说明
Present标志	为1，表示页（或页表）在内存；为0，则不在内存。
页框物理地址(20位)	页框大小为4096，占去12位。 $20+12=32$
Accessed 标志	页框访问标志，为1表示访问过
Dirty标志	每当对一个页框进行写操作时就设置这个标志
Read/Write标志	存取权限。Read/Write或Read
User/Supervisor标志	访问页或页表时所需的特权级
PCD和PWT标志	设置PCD标志表示禁用硬件高速缓存，设置PWT表示写直通
Page Size 标志	页目录项的页大小标志。置1，页目录项使用2MB/4MB的页框
Global标志	页表项使用。在Cr4寄存器的PGE标志置位时才起作用

8.4 请求调页与缺页异常处理

- 请求调页
 - 请求调页增加了系统中的空闲页框的平均数
 - 页面置换策略是LFU
- 缺页调入
 - 该页从未被进程访问过，且没有相应的内存映射，从指定文件中调页
 - 该页属于非线性内存映射文件，从磁盘读入页
 - 该页已被进程访问过，但其内容被临时保存到磁盘交换区，从交换区调入
 - 该页在非活动页框链表中，直接使用

- 该页正在由其他进程进行I/O传输过程中，等待传输完成

8.5 盘交换区空间管理

- 每个盘交换区都由一组4KB的页槽组成
- 盘交换区的第一个页槽用于存放该交换区的有关信息，有相应的描述符
- 存放在磁盘分区中的交换区只有一个子区，存放在普通文件中的交换区可能有多个子区，原因是磁盘上的文件不要求连续存放
- 内核尽力把换出的页存放在相邻的页槽中，减少访问交换区时磁盘的寻道时间

九、Linux文件系统

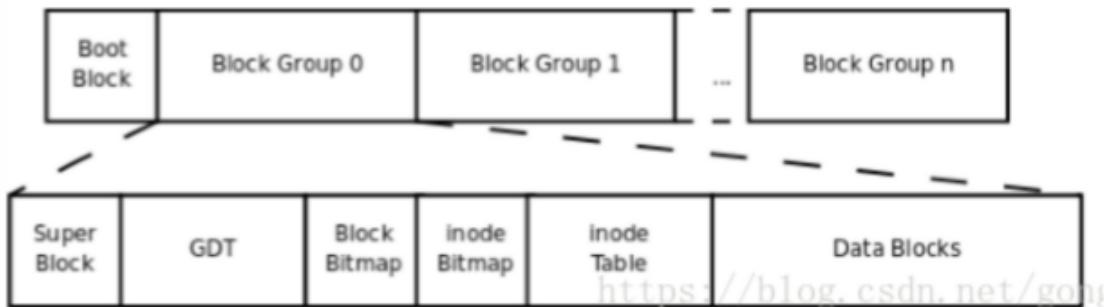
- Unix文件系统的特点
 - Unix的文件目录采用树形结构
 - Unix将其管理的文件看成一个无结构的字节流
 - Unix把外部设备看成与普通文件一样的特殊文件
- Linux文件系统
 - 类Unix系统
 - 继承了Unix文件系统的特色
- Linux最初采用的是Minix的文件系统。当Linux成熟时，引入了扩展文件系统(ext)，ext文件系统支持255个字符的文件名和最大2GB的文件
- 同时增加了虚拟文件系统，不同类型文件系统通过VFS提供的统一接口进行映射，使得上层应用不依赖于具体文件系统实现

9.1 ext2的磁盘涉及的数据结构

9.1.1 Linux文件卷的布局

- 一个ext2磁盘文件卷由若干个大小相等的磁盘块组成。包括一个引导块和n个块组

每个块组由超级块、块组描述符、数据块位图、文件的索引节点位图、索引节点区和文件数据区组成



- 块
 - 用来存储数据的单元，大小相同
 - block按照0, 1, 2, 3的顺序进行编号
 - ext2支持的block大小有1024B、2048B和4096B，其大小在创建文件系统的时候通过参数指定
 - block的大小和数量在格式化完成之后不能再改变（除非重新格式化）
 - 每个block内最多只能放置一个文件的数据
- 引导块
 - ext2文件卷的第一个盘块，不使用
 - 用作Linux的引导块或保留不用
 - 用于读入并启动操作系统
- 块组
 - 每个块组包含相邻磁道的磁盘块
 - 每个块组大小相等且顺序排列
 - 使用“块组描述符”记录块组信息
 - 每个块组中均存放超级块和所有块组信息，但OS只使用块组0中的相应信息
- 超级块
 - 存放整个文件卷的资源管理信息
 - 操作系统只是用块组0中的超级块和块组描述符
 - 采用e2fsck程序对文件系统状态进行一致性检查时，将块组0中的超级块和块组描述符复制给其他块组
- 数据块位图
 - 记录文件数据区各个盘块的使用情况
 - 存放在一个单独的块中
- 索引节点位图
 - 记录索引节点区各个索引节点的使用情况
 - 存放在一个单独的块中
- 索引节点区
 - 存放文件的索引节点
 - 一个索引节点存放一个文件的管理和控制信息
- 文件数据区
 - 存放普通文件和目录文件
- 块组个数与大小
 - 一个块组中的盘块和索引节点的位图必须存放在一个单独的磁盘块中
 - 若s是文件卷的总块数，b是以字节为单位的块大小，每组中至多可以有8b个盘块。因此可划分的块组总数大约是s/(8b)。由于块大小可以是1024、2048或4096B。因此，一个块的位图，分别可以描述8192、16384或32768个盘块的状态

e.g.假定ext2文件卷的大小为32GB, 盘块的大小为4KB。在这种情况下, 每个4KB的盘块位图最多可以描述32K个磁盘块, 即一个块组最大不超过128MB。因此, 最多可以划分为256个块组

9.1.2 超级块

- 包含文件的主要管理和控制信息
 - block和inode的总量未使用与已使用的inode/block的数量
 - block与inode的大小(block为1, 2, 4K, inode为128bytes)
 - filesystem的挂载时间, 最近一次写入数据的时间、最近一次检验磁盘(fsck)的时间\
 - 其他一些文件系统的相关信息

```
struct ext2_super_block {  
    _le32 s_inodes_count;      //索引节点的总数  
    _le32 s_blocks_count;     //盘块的总数  
    _le32 s_free_blocks_count; //空闲块计数  
    _le32 s_free_inodes_count; //空闲索引节点数  
    _le32 s_log_block_size;   //盘块的大小  
    _le32 s_blocks_per_group; //每组中的盘块数  
    _le32 s_inodes_per_group; //每组索引节点数  
    _le16 s_inode_size;       //磁盘上索引节点结构的大小  
    .....  
};
```

9.1.3 块组描述符

- 每个块组都有自己的块组描述符, 记录了该块组管理的一些重要信息
 - 盘块位图的块号
 - 索引节点位图的块号
 - 块组中空闲盘块数
 - 块组中的空闲索引节点个数
 - 组中目录个数

```
struct ext2_group_desc {  
    _le32 bg_block_bitmap; //盘块位图的块号  
    _le32 bg_inode_bitmap; //索引节点位图的块号  
    _le32 bg_inode_table; //索引节点区  
                           //第一个盘块块号  
    _le16 bg_free_blocks_count; //组中空闲块个数  
    _le16 bg_free_inodes_count; //组中空闲索引节  
                               //点个数  
    _le16 bg_used_dirs_count; //组中目录个数  
    .....  
};
```

9.1.4 文件目录与索引节点结构

- 文件目录
 - 采用树形目录结构管理所有文件
 - 把通常的文件目录项分成简单目录项和索引节点两部分
 - 简单目录项包含了文件名和索引节点号等，可以提高文件目录的检索速度
 - 文件的管理控制信息存放在索引节点中
 - 系统只保留一个索引节点，就可以实现多条路径共享文件，减少信息冗余
 - 目录项结构

```
struct ext2_dir_entry_2{    _le32 inode; //索引节点号    _le16 rec_len; //目录项长度    _u8 name_len; //实际文件名长度    _u8 file_type; //文件类型    char name[255]; //文件名是4B整数倍        //变长数组}
```
 - 文件类型
 - ◆ 0, 不可知
 - ◆ 1, 普通文件
 - ◆ 2, 目录文件
 - ◆ 3, 字符设备文件
 - ◆ 4, 块设备文件
 - ◆ 5, 有名管道
 - ◆ 6, 套接字文件
 - ◆ 7, 符号链接
 - 目录项位置

	inode	rec_len	name_len	file_type	name
0	21	12	1 2 . \ 0 0 0		
12	22	12	2 2 . . \ 0 0		
24	53	16	5 2 h o m e 1 \ 0 0 0		
40	67	28	3 2 u s r \ 0		
52	0	16	7 1 o l d f i l e \ 0		
68	34	12	4 2 s b i n		

- 索引节点
 - 索引节点存放在索引节点区
 - 索引节点区由连续磁盘块组成
 - 第一个索引块的块号存放在块组描述符的bg_inode_table字段中
 - 一个索引节点的大小为128B
 - 索引节点存放文件的管理和控制信息
 - 该文件的访问模式(rwx)
 - 该文件的所有者与组(owner/group)
 - 该文件的大小
 - 该文件创建或状态改变的时间
 - 最近一次的读取时间

- 最近修改内容的时间
- 定义文件特性的标志
- 该文件真正内容的指向

```
struct ext2_inode {
    __le16 i_mode;          //文件类型和访问权限
    __le16 i_uid;           //拥有者的标识符
    __le32 i_size;          //以字节为单位的文件长度
    __le16 i_gid;           //组标识符
    __le16 i_links_count;   //硬链接计数
    __le32 i_block[15];     //索引表, 15×4B
    __le32 i_blocks;        //文件的数据块数
    __le32 i_file_acl;      //文件访问控制表ACL
    .....
};
```

- 索引表i_block：是一个有15个元素的数组，每个元素占4B

- 数组的15个元素有4种类型，三级混合索引
 - 直接索引，占前12项（0-11）
 - 一次间接索引，占第12项
 - 二次间接索引，占第13项
 - 三次间接索引，占第14项
- 满足小型、中型、大型、巨型文件的存储需求
 - 小型文件：文件占用物理块数 ≤ 12
 - 中型文件： $12 < \text{文件占用物理块数} \leq 12 + b/4$ 块，

| b为一个盘块以字节为单位的大小，哎呀 $b/4$ 就是一个盘块能记录多少个盘块号嘛

- 大型文件： $12 + b/4 < \text{文件占用物理块数} \leq 12 + b/4 + (b/4)^2$
- 巨型文件： $12 + b/4 + (b/4)^2 < \text{文件占用物理块数} \leq 12 + b/4 + (b/4)^2 + (b/4)^3$

9.2 ext2的主存数据结构

- 安装ext2文件系统时，存放在ext2文件卷的磁盘数据结构中的大部分信息被复制到主存RAM中，从而使内核避免以后的多次重复的磁盘读写操作
- 频繁使用的超级块和组描述符总是缓存在主存高速缓冲区
- “动态”模式
 - 只要相关的对象（索引节点、数据块和相应的位图）正在使用，其数据就保存在高速缓存中
 - 当相应的文件被关闭或删除时，页框回收算法从高速缓存中删除相关数据并将其写回磁盘

9.2.1 超级块和索引节点对象

- 当安装ext2文件系统时，用类型ext2_sb_info结构对象缓冲ext2_super_block结构，以便内核能找出与该文件系统相关的内容
- 超级块
 - 磁盘结构：ext2_super_block
 - 内存结构：ext2_sb_info

```
struct ext2_sb_info {  
    unsigned long s_inodes_per_block; //每块包含索引节点数  
    unsigned long s_blocks_per_group; //每组盘块数  
    unsigned long s_inodes_per_group; //每组索引节点数  
    unsigned long s_itb_per_group; //每组索引节点区块数  
    unsigned long s_gdb_count; //每组的组描述符的盘块数  
    unsigned long s_desc_per_block; //每组的组描述符数  
    unsigned long s_groups_count; //整个文件系统的组数  
    struct ext2_super_block *s_es; //指向超级块所在  
                                   //缓冲区的指针  
    unsigned short s_loaded_inode_bitmaps; //索引节点位图  
    .....  
};
```

140

- 当ext2文件的索引节点对象被初始化时，用类型ext2_inode_info结构对象缓冲磁盘的索引节点ext2_inode结构
- 索引节点

```
o 磁盘结构：ext2_inode  
o 内存结构：ext2_inode_info  
struct ext2_inode_info {  
    _u32 i_block[15]; //文件数据块的索引表  
    _u32 i_flags; //文件标志  
    _u32 i_file_acl; //文件访问控制表  
    _u32 i_dir_acl; //目录访问控制表  
    _u32 i_block_group; //索引节点所在块组的索引  
    _u32 i_next_alloc_block; //预分配文件逻辑块号  
    _u32 i_next_alloc_goal; //预分配磁盘物理块号  
    _u32 i_prealloc_block; //预分配第一个数据块  
    _u32 i_prealloco_count; //预分配数据块总数  
    .....  
};
```

9.2.2 位图高速缓存

- ext2中，每一个块组需要两个磁盘块，分别用作描述文件数据区和索引节点区使用情况的位图，即数据区位图块和索引节点位图块
- 随着磁盘块容量增加，将所有节点均保存于RAM中不再显示
- 使用大小为ext2_max_group_loaded的两个高速缓存，分别存放最近访问的大部分索引节点位图和数据块位图

9.3 ext2磁盘空间管理

- 磁盘块和索引节点的分配和回收
- 文件的数据块和其索引节点尽量在同一个块组中
- 文件和它的目录项尽量在同一个块组中
- 父目录和子目录尽量在同一个块组中
- 每个文件的数据块尽量连续存放
- 采用位图实现磁盘块和磁盘索引节点的分配和回收

9.3.1 磁盘索引节点的管理

- 当用户创建一个新文件时，系统为其分配一个磁盘索引节点存放文件的管理控制信息
 - ext2_new_inode()
- 当用户要求从磁盘上删除一个文件或目录时，系统在回收该文件或目录占用的磁盘索引节点之间，先回收文件或目录占用的数据块
 - ext2_free_inode()

9.3.2 空闲磁盘块的分配与回收

- 内核为普通文件分配一个磁盘块
 - ext2_get_block()
- 当进程要删除或截短一个文件时，要释放文件不再占用的磁盘块
 - ext2_truncate()

十、Linux虚拟文件系统

- 借助VFS，Linux可以透明地安装具有其他操作系统的文件系统格式的磁盘或分区

- VFS与具体文件系统的关系

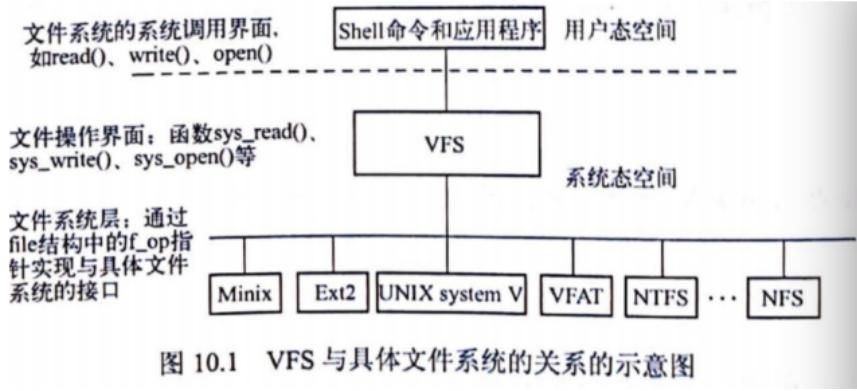


图 10.1 VFS 与具体文件系统的关系的示意图

- VFS

- 主要思想：引入一个通用的文件模型，该模型能够表示其支持的所有文件系统
- VFS涉及的所有数据结构在系统运行时才能在内存建立，其磁盘上没有存储
- 作用
 - 对各种文件系统的数据结构进行抽象，以统一的数据结构进行管理
 - 接收用户层的系统调用
 - 支持多种具体文件系统之间的相互访问
 - 接收内核其他子系统的操作请求

10.1 虚拟文件系统涉及的数据结构

10.1.1 超级块对象

- 代表一个已安装的文件系统，存放该文件系统的管理和控制信息
- Linux为每个安装好的文件系统都在内存中建立一个超级块对象
- VFS超级块是各种文件系统在已安装时建立的，并在卸载时自动删除，其数据结构时super_block
- 所有超级块对象都以双向循环链表的形式链接在一起
- 与超级块关联的方法是超级块操作表，这些操作是由struct_super_operations描述

```

struct super_block {
    struct list_head s_list; // 系统超级块双向链
    struct file_system_type *s_type; // 文件系统类型
    struct super_operations *s_op;
    struct dentry *s_root // 根目录的目录项对象
    struct list_head s_inode; // 索引节点链表
    struct list_head s_files; // 文件对象链表
    void *s_fs_info; // 指向一个具体文件系统在内存
                      // 的超级块结构X_sb_info
    .....
}

```

10.1.2 索引节点对象

- 对于具体文件系统，代表一个文件，对应于存放在磁盘上的FCB
- 文件的管理和控制信息都包含在索引节点inode结构中
- 同一个文件系统中，每个文件的索引节点号都是唯一的
- 与索引节点关联的方法由struct inode_operations来描述

```
struct inode {  
    struct list_head i_sb_list; //同一个超级块的索引  
                                //节点链表的指针  
    struct list_head i_dentry; //指向引用这个索引节点  
                                //的目录项表的指针  
    unsigned long i_ino;     //磁盘索引节点号  
    atomic_t i_count;       //该对象的引用计数  
    nlink_t i_nlink;        //硬链接计数  
    loff_t i_size;          //文件的字节长度  
    struct timespec i_atime; //文件的最近访问时间  
    .....  
}
```

10.1.3 目录项对象

- 对应一个目录项，是文件路径的组成部分，存放目录项与对应文件进行链接的信息
- 每个文件除了一个inode结构外，还对应一个目录项dentry结构
- 对于进程查找文件路径名中的每个分量，内核都为其建立一个目录项对象
- 目录项代表的是逻辑意义上的文件，描述的是文件逻辑上的属性，目录项对象在磁盘上并没有对应的映像
- inode代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统，其inode在磁盘上有对应的映像
- 目录项对象在磁盘上没有映像，所有目录项对象存放在名为dentry_cache的slab分配器的高速缓冲区中
 - 相当于索引节点高速缓冲控制器
 - 一旦目录项对象被使用，可以很快引用相应的索引节点对象
- 与目录项关联的方法由struct dentry_operations描述
- 一个索引节点可能对应多个目录项对象

```

struct dentry {
    atomic_t d_count;      // 目录项对象引用计数
    struct inode *d_inode; // 指向文件的inode对象
    struct dentry *d_parent; // 指向父目录项对象
    struct list_head d_alias; // 属于同一inode的
                             // dentry链表
    struct dentry_operations *d_op; // 访问目录项
                                   // 的方法
    .....
}

```

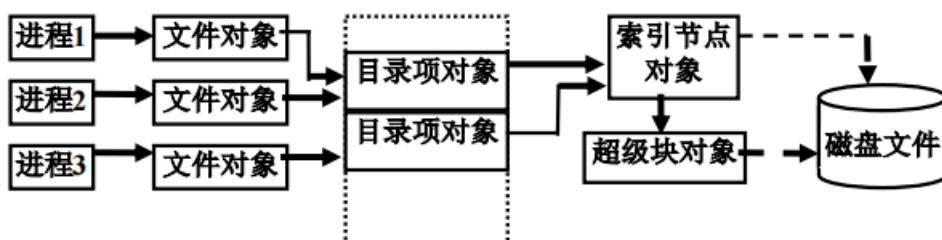
10.1.4 文件对象

- 记录了进程与打开的文件之间的交互信息
- 为了描述进程与其打开的一个文件进行交互而引入
- 文件对象在磁盘上没有对应的映像，在文件打开时创建，由file结构描述
- 一个文件对应一个目录项对象
- file结构形成一个双链表，称为系统打开文件表
 - 文件对象
 - 文件对应目录项
 - 文件对象引用计数
 - 文件读写位置指针
 - 文件方法
- 与文件对象关联的方法由struct file_operations来描述

```

struct file {
    struct list_head f_list;      // 文件对象链表
    struct dentry *f_dentry; // 文件对应目录项对象指针
    atomic_t f_count;      // 文件对象的引用计数
    loff_t f_pos;          // 文件的当前读写位置
    struct file_operations *f_op; // 访问文件对象的方法
    struct address_space *f_mapping; // 指向该映射文件
                                    // 的地址空间对象指针
    .....
}

```



10.1.5 与进程打开文件相关的数据结构

- 进程描述符task_struct
 - fs字段存放的fs_struct结构的信息：描述安装的文件系统相关信息
 - files字段存放的file_struct结构：描述当前打开文件的信息
 - 用户打开文件表，是进程的私有数据
 - 文件描述符
 - 读写指针
 - 系统打开表入口指针
- 超级块是对一个文件系统的描述
- 索引节点是对一个文件物理属性的描述
- 目录项是对一个文件逻辑属性的描述
- 进程所在的文件系统由fs_struct描述
- 一个进程（用户）打开的文件由file_struct描述
- 整个系统打开的文件由file结构描述

10.2 文件系统的注册与安装

- Linux系统支持的所有文件系统在使用前必须进行注册，完成注册之后，VFS为内核提供一个调用相应文件系统的方法的接口
- 之后，再用mount命令将该文件系统安装到根目录系统的某个目录结点上

10.2.1 文件系统注册

- 生成一个file_system_type类型的结构，填写相应的内容
- 调用register_filesystem()完成注册
- 所有已注册的各种文件系统类型的file_system_type()结构形成一个文件系统类型链表

10.2.2 文件系统安装

- 向Linux内核注册一个文件系统只是通知内核可以支持的文件系统，必须将该文件系统相应的硬盘分区安装在系统目录树的某一目录（安装点）下
- 内核将安装点与被安装的文件系统信息保存在vfsmount结构中，形成一个链式安装表
- vfsmount是一次安装的实例，包含了操作该特定文件系统所必须的信息
- 一个超级块对应一个特定文件系统，超级块建立了vfs与特定文件系统之间的关联
- 同种文件系统类型的super_block挂在fs_supers链表上
- 根文件系统
 - 最顶层的文件系统叫做“根文件系统”
 - Linux在启动的时候，要求用户必须指定一个“根设备”，内核在初始化阶段，将“根设备”安装到“根安装点”上，从而有了根文件系统。这样，文件系统才算准备就绪。
- 其他文件系统由用户通过mount命令来安装

10.3 VFS系统调用的实现

10.3.1 文件的打开与关闭

- open()
- close()

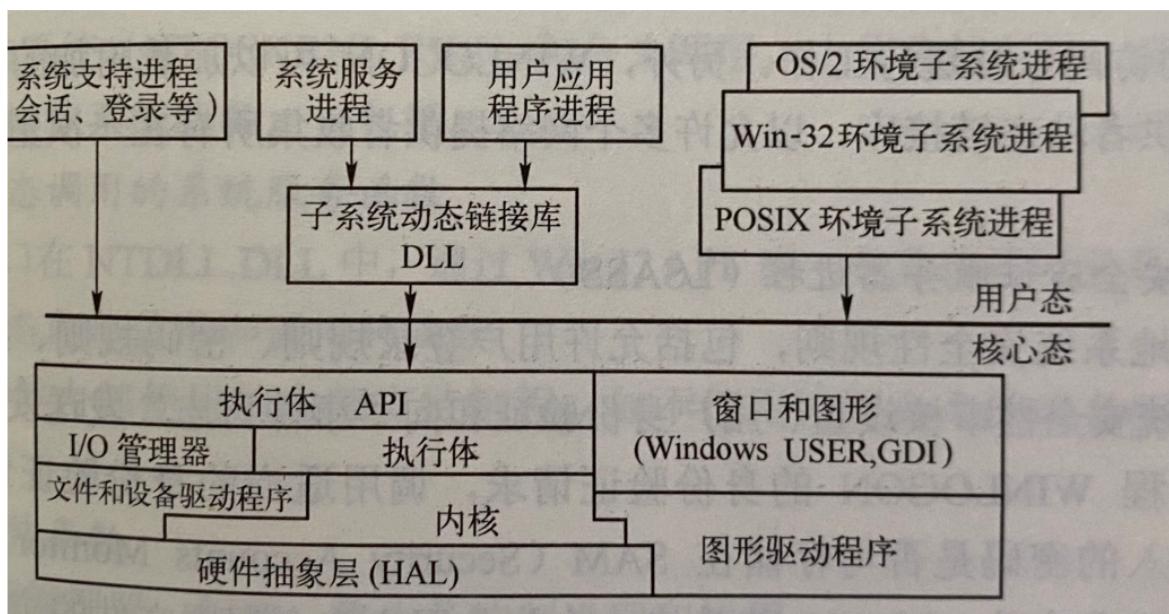
10.3.2 文件的读写

- read()
- write()

Windows实例

十四、Windows操作系统模型

14.1 Windows的体系结构



14.1.1 用户态进程

- 系统支持进程
 - idle进程
 - id为0，每个CPU都有一个相应的线程，用来统计空闲CPU时间
 - 系统进程
 - 运行在核心态的“系统进程”的宿主，id为2，负责执行I/O请求、进程的换入和换出，将内存脏页写入外存等
- 会话管理器SMSS
 - 系统中创建的第一个用户态进程，负责系统的初始化
 - 创建本地过程调用LPC端口对象和两个线程，等待客户请求
 - 创建系统环境变量
 - 打开已知的动态链接库
 - 加载Win32子系统的部分（WIN32K.exe）
 - 启动登录进程（WINLOGIN），启动子系统进程
 - 会话管理器主进程等待WIN32子系统进程（CSRSS，为用户提供GUI）和登录进程的进程句柄，直至系统终止
- Windows登录进程WINLOGIN
 - 处理用户的登录和注销
 - 捕获到合法用户名和密码后，发送至本地安全身份验证服务器LSASS进行验证
 - 合法则代表用户创建并激活一个登录shell进程（Explorer.exe）
- 本地安全身份验证服务器进程LSASS
 - 负责本地系统安全性规则，包括用户登录规则，密码规则，权限列表等
 - 身份验证成功，将生成包含用户安全配置文件的访问令牌对象，并返回给登录进程
- 系统服务进程
 - 系统引导时，自动创建和启动
 - Spooler、事件日志服务、RPC、网络组件等作为服务实现
- 环境子系统进程
 - Win32
 - 向应用程序提供运行环境的调用接口
 - 创建控制台窗口、创建/删除进程/线程、输入信息、提供GUI等
 - POSIX、OS/2
 - 调用Win32子系统中的服务显示I/O
- 用户应用程序
 - 不能直接调用操作系统服务，所有请求必须由用户态的动态链接库检查合法后，转换成系统内部相应的API调用

14.1.2 子系统动态链接库

- NTDLL.dll
 - 特殊的用于子系统动态链接的系统支持库
 - 两类函数
 - 作为Windows执行体系统服务的接口提供给用户态调用时使用，通过Win32API访问
 - 执行检查，转换成系统调用
 - 为子系统、子系统动态链接库及其他本机映像使用的内部支持函数

- 映像加载、堆管理、Win32子系统通信、

14.1.3 核心态的系统组件

- 执行体
 - 在内核之上，包含基本的操作系统服务
 - 可供用户态调用的系统服务函数
 - 接口在NTDLL.dll中
 - 通过Win32API或环境子系统访问
 - 仅供核心态内部使用和调用的函数
 - 各组件功能函数
 - 进程和线程管理器
 - 虚拟内存管理器
 - 安全监视器
 - 高速缓存管理器
 - I/O管理器
 - 对象管理器
 - 本地过程调用LPC
 - 经过优化的RPC，实现在同一台计算机上的客户进程和服务器进程之间的消息传递
 - 即插即用管理PnP
 - 电源管理
 - 网络管理、交互进程通信、系统注册的配置管理
- 内核
 - 执行Windows最基本的操作
 - 功能
 - 线程安排和调度
 - 中断和异常调度
 - 多处理机同步
 - 提供执行体使用的一组例程和基本对象
 - 运行在核心态，除中断服务例程ISR以外，正在运行的线程不能抢占内核
 - 内核对象
 - 内核提供一组严格定义的低级的基本内核对象，帮助控制、处理和支持执行体对象的创建
 - 大多数执行体对象都封装了一个或多个内核对象
 - 控制对象
 - 控制操作系统的各个基本功能
 - 内核进程对象
 - 异步过程调用对象APC
 - 延迟过程调用对象DPC

- 由I/O系统使用的对象
- 调度对象
 - 用来同步进程和线程操作并影响进程调度
- 内核线程
- 互斥体事件
- 内核事件对
- 信号量
- 定时器
- 提供独立于硬件的支持
- 硬件抽象层HAL
 - 可加载的核心态模块HAL.dll，直接操作硬件，为Windows运行在硬件平台提供低级接口
 - 隐藏各种与硬件有关的细节，使上层免受特殊硬件平台的影响，系统可移植性好
- 设备驱动程序
 - 可加载的核心态模块
 - 不直接操作硬件，通过调用硬件抽象层与硬件交互
- GUI

14.2 Windows操作系统的特点

- 可移植性
 - 采用分层模型：硬件抽象层HAL
 - 将操作系统的实现机制和策略分离
 - 机制：系统能完成任务的方法、能实现的功能
 - 策略：按照什么算法选择哪个任务、什么时候执行
 - 分离
 - 每个环境子系统建立各自的OS策略层
 - NT执行体建立适合所有子系统的相对基础的策略层
 - 内核层完全不考虑策略
- 支持对称多处理和可伸缩性
 - 多处理共享内存
 - 用户任务和操作系统代码可以被调度运行在任何一个处理机上
 - NTOSKRNL.exe在单/多处理机上有所不同
- 核心组件使用面向对象原则
 - 融合了分层模型和客户/服务器模型
 - 用户应用程序进程与服务器进程之间通过NT执行体中提供的消息传递工具进行通信

14.3 Windows的系统机制

14.3.1 陷阱处理程序

- 操作系统用来处理意外事件的硬件机制
- 处理过程
 - 当硬件或软件检测到异常或发生中断时，暂停正在处理的事情，将控制权交给内核的陷阱处理程序
 - 关中断并将足够多的机器状态记录到被中断线程的核心栈
 - 若被中断线程处于用户态，则系统切换到该线程的核心态，并在核心态创建一个陷阱帧，保存中断线程的运行现场
 - 该程序检测异常和中断类型，将控制转交给处理相应程序的代码
- 中断
 - 异步事件
 - 硬件中断：由I/O设备、处理器时钟或定时器产生
 - 软件中断：内核自身、设备驱动程序
 - 可以开或关中断
- 异常
 - 同步事件
 - 某一特定指令执行的结构，相同条件下，异常可以重复出现
- 软件和硬件都可以产生中断和异常

14.3.2 中断调度

- 设备驱动程序与内核分别提供设备或其他类型中断的中断处理例程
- Windows有32各中断请求级别IRQL，又叫中断优先级
 - 硬件中断
 - 软件中断
 - 用户线程



- I/O请求中断处理
 - 中断调度程序首先提高处理器的IRQL至中断源统一级别，以屏蔽低级中断源，执行结束后，再将处理器的IRQL调整回至中断发生前级别
 - 处理
 - 当设备中断产生时，调用中断处理例程ISR，根据IRQL处理硬件中断
 - 当IRQL降低到DPC级别以下时，产生DPC中断，调度DPC例程，完成中断处理
 - 同步I/O，直接返回发起者线程；异步I/O，在发起者线程中插入一个APC对象以便通知I/O完成
 - 之后需调用IoCompleteRequest在进程地址空间将数据从系统缓冲区复制到用户缓冲区

- DPC被内核用来执行一些相对于当前高优先级的任务不那么紧急的任务
 - 有时内核在进行系统嵌套调用时，检测到应该进行线程调度，为了保证调度的正确性，内核用DPC来延迟调度的产生
 - 在硬件中断服务例程中，可以把一些相对不那么紧急的事情放到一个DPC对象中，从而缩短处理器停留在高IRQL的时间
 - 当IRQL降低到DPC级别以下时，产生DPC中断，依次执行DPC队列中的每个例程，直至DPC队列为空
 - DPC队列是系统范围的
- APC为用户程序和系统代码提供了一种在特定用户线程描述表中执行代码的方法
 - 每个线程都有自己的APC队列，当一个线程被调度时，会立刻执行其APC过程
 - 分类
 - 核心态APC：可以中断线程、在线程表中执行该过程，无需目标线程“允许”
 - 执行体、环境子系统、设备驱动程序
 - 用户态APC：需要目标线程“允许”
 - 报警
 - APC非常适合于实现各种异步通知事件
 - APC队列是线程范围的
 - e.g. 异步I/O完成通知事件

14.3.3 异常调度

- 异常是由运行程序产生的同步事件
- 简单的异常可由陷阱处理程序解决，其余异常均由内核调度模块的“异常调度程序”提供服务
- 内核定义异常
 - 内存访问违约
 - 被零除
 - 整数溢出
 - 浮点异常
 - 调度程序断点

14.3.4 系统服务调度

- 检查调度服务程序
 - 检查参数，并将调用参数从用户态堆栈复制到核心栈
 - 使用参数查找“系统调度服务表”中的系统服务信息
 - 每个线程都有一个指向系统服务表的指针

14.4 对象管理器

14.4.1 对象结构

- Windows使用对象模型提供服务
- 对象管理器管理操作系统内的所有对象
- 对象是一种内核数据结构
 - 执行体对象
 - 执行体的各种组件实现的对象
 - 进程管理器
 - 内存管理器
 - I/O管理器
 - 对象管理器
 - 内核对象
 - 内核实现的初级对象，仅供执行体使用

执行体对象-内核对象 1:n

- 当系统刚启动时，并没有任何对象，对象是后来创建的
- 对象头和对象体
 - 对象由对象头和对象体组成
 - 所有类型的对象头结构都是相同的，而且对象体部分各不相同
 - 对象管理器控制对象头，提供一个雄安的通用服务器用于操作对象头中存储的属性
 - 各执行体组件控制其创建对象的对象体
- 对象类型与类型对象
 - 每种类型的对象只有一个类型对象
 - 每种类型的对象都在其对象体中存在一个指向其类型对象的指针
 - Windows支持31中类型对象

14.4.2 管理对象

- 对象的组织
 - Windows中的内核对象独立存在于系统地址空间中
 - 系统利用目录对象将所有这些对象组织起来
 - 目录对象是由一个由37个数组元素组成的哈希树
 - 系统将对象名称进行一定的计算得出一个哈希值
- 对象的访问
 - 系统通过对对象名唯一地标识一个对象
 - 对象管理器通过对对象名发现和检索一个特定对象
 - 内核中知道内核对象的地址可以直接访问该内核对象
 - 进程通过对对象名创建或打开对象时，获得一个对象的句柄，进程通过句柄访问对象
 - 句柄值是进程独立的，一个进程中的句柄值在另一个进程中是无效的
 - 句柄值是一个进程句柄表的索引，每个进程都有一个进程句柄表

14.5 对象之间的同步

- 内核同步
 - 在内核执行的不同阶段，必须保证每次有且仅有一个处理机在临界区执行
 - 内核临界区就是修改全局数据结构的代码段
 - 内核引入自旋锁实现多处理机互斥访问内核临界区
 - 在Intel处理机上，自旋锁是通过“测试与设置”TestSet硬件指令实现的
 - 拥有自旋锁的线程不会被剥夺处理机，其独占处理机执行
- 执行体同步
 - 自旋锁的限制
 - 被保护的资源必须被快速访问，并且不与其他代码进行复制的交互
 - 临界区代码不能换出内存，不能引用可分页数据，不能调用外部程序（包括系统服务），不能产生中断或异常
 - 用户态同步机制
 - 调度程序对象
 - 内核以内核对象形式给执行体提供的用户态同步机制
 - 进程、线程、事件、信号量、互斥体、可等待的定时器、I/O完成端口或文件等同步对象
 - 每个同步对象有两种状态：“有信号”，“无信号”
 - 线程、进程终止时有信号
 - Windows定义了统一的同步机制，等待调度程序对象为有信号状态
 - WaitForSingleObject()

十五、Windows进程和线程管理

15.1 Windows进程和线程

15.1.1 进程对象

- 执行体进程块EPROCESS

- 表示进程的基本属性

```
//Windows
typedef struct _EPROCESS {
    KPROCESS Pcb;           //内核进程块
    PPEB Peb;              //进程环境块

    HANDLE UniqueProcessID; //进程编号，PID

    LARGE_INTEGER CreateTime; //进程创建时间
    LARGE_INTEGER ExitTime;  //进程退出时间

    KSPIN_LOCK HyperSpaceLock; //自旋锁

    .....
};
```

- 内核进程块KPROCESS

- 进程控制块
- 包含了内核调度线程的必要信息，如基本优先级、默认时间片、进程状态、进程页目录表
- 进程环境块PEB
 - 位于进程的用户态地址空间
 - 包含了映像加载程序需要的信息、线程使用的堆信息等

15.1.2 线程对象

- WIndows应用程序以独立进程方式运行，每个进程可包括一个或者多个线程
- Windows实现了一对一的映射，每个用户线程映射到相关的内核线程
- Windows也提供了对fiber库的支持，该库提供了多对多模型
- 线程是内核支持的进程内的处理机调度执行的一个实体。它具有如下特点：
 - 一个线程ID，用于唯一标识线程
 - 一组寄存器集合，代表CPU状态
 - 一个用户栈和一个内核栈，分别供线程在用户模式下和内核模式下运行
 - 一个私有存储区域，为各种运行时库和动态链接库使用
- WIndows线程的数据结构
 - ETHREAD：执行线程块
 - 包括线程进程的指针和线程开始控制的子程序的地址，以及相应的KTHREAD
 - KTHREAD：内核线程块
 - 包括线程的调度和同步信息，以及内核栈（线程在内核模式下运行时使用）和TEB
 - TEB：线程环境块
 - 包括线程相关信息、用户模式栈和用于线程特定数据的数组
- 线程对象的服务
 - 创建线程CreateThread
 - 线程退出ExitThread
 - 终止某个线程TerminateThread
 - 改变线程优先级SetThreadPriority

15.2 线程调度

- 进程优先级
 - 空闲优先级 (4)
 - 系统处于空闲状态时执行的程序，如屏幕保护程序
 - 普通优先级 (7/9)
 - 能升高或降低普通进程优先级
 - 高优先级 (13)
 - 只在需要时才使用，如Task manager以高优先级运行
 - 实时优先级 (24)
 - 用于核心态系统程序
- 线程优先级
 - 采用基于优先级的、抢占算法调度线程
 - 抢占确保最高优先级的线程总是运行
 - 线程优先级
 - 可变类型：1-15
 - 实时类型：16-31
 - 特殊线程：0
 - Win32 API定义的优先级类型
 - REALTIME_PRIORITY_CLASS
 - HIGH_PRIORITY_CLASS
 - ABOVE_NORMAL_PRIORITY_CLASS
 - NORMAL_PRIORITY_CLASS
 - BELOW_NORMAL_PRIORITY_CLASS
 - IDLE_PRIORITY_CLASS
 - 相对优先级
 - TIME_CRITICAL
 - HIGHEST
 - ABOVE_NORMAL
 - NORMAL
 - BELOW_NORMAL
 - LOWEST
 - IDLE
 - 每个线程的优先级是基于其所属优先级类型的相对优先级
 - 每个线程在其所属的类型中有一个基础优先级，一般为类型中的NORMAL相对优先级的值
 - 进程通常属于NORMAL_PRIORITY_CLASS
 - 可变优先级
 - 线程时间片用完，降低优先级，但不会降至基础优先级之下
 - 线程从等待操作释放时，调度程序提升其优先级
- 单处理机下线程调度
 - 线程调度时机
 - 主动切换
 - 线程因等待某个事件等主动放弃处理机
 - 抢占
 - 被抢占线程放在就绪队列队首
 - 实时优先级线程被抢占时，时间配额重置为一个完整时间片

- 可变优先级线程被抢占时，时间配额不变
- 时间配额用完
 - 确定是否需要降低该线程的优先级
 - 确定是否需要调度一个线程
- 运行结束
- 线程状态
 - 就绪态
 - 就绪
 - 备用态
 - 被选中作为下一个要执行的线程，正等待描述表切换
 - 运行态
 - 运行
 - 等待态
 - 阻塞等待
 - 传输态
 - 类似就绪，但是核心栈被调到外存。当核心栈被调回主存时，变为就绪态
 - 终止态
 - 初始化状态

15.3 对称多处理机 (SMP) 上的线程调度

15.3.1 几个与调度有关的概念

- 亲合关系
 - 每个线程都有一个亲合掩码，描述该线程可在哪些处理机上运行
 - 默认时，所有进程的亲合掩码为系统中所有可用处理机的集合
- 线程的首选处理机和第二处理机
 - 首选处理机
 - 线程运行时偏好的处理机
 - 基于进程控制块的索引值在线程创建时随机选择
 - 第二处理机
 - 线程最近在运行的处理机

15.3.2 线程调度程序的数据结构

- 32个就绪队列
 - 每个优先级对应一个
- 32位线程就绪队列位图
 - 每一位指示一个优先级就绪队列中是否有线程等待运行

- 32位处理机空闲位图
 - 每一位指示一个处理机是否处于空闲状态

15.3.3 多处理机的线程调度算法

- 有空闲处理机
 - 首选处理机---->第二处理机---->正在运行线程调度程序的处理机---->按处理机编号选第一个空闲处理机
- 无空闲处理机
 - 是否可抢占一个处于运行状态或备用状态的线程
 - 可以，首选处理机---->第二处理机---->按处理机掩码选择编号最大的处理机
 - 备用低优先级---->运行低优先级
- 为特定处理机选择线程
 - 不是简单地从就绪队列队首选取线程
 - 寻找满足下列4个条件之一的线程
 - 上一次运行在该处理机上
 - 首选处理机是该处理机
 - 处于就绪状态的时间超过2个时间配额
 - 优先级大于等于24
- 优先级高的就绪线程可能不处于运行状态
 - 在多处理机系统中，由于线程的亲合关系，系统并不总是选择优先级高的线程抢先优先级低的线程所占用的处理机

15.4 线程优先级提升

- 系统会提升线程的优先级，以改善性能
 - I/O操作完成后的线程
 - 信号量或事件等待结束的线程
 - 前台进程中的线程完成一个等待操作
 - 由于窗口活动而唤醒GUI线程
 - 线程处于就绪状态超过一定时间，仍未能进入运行状态（处理机饥饿）

15.5 Windows的线程同步

15.5.1 同步对象

- 事件对象
 - 相当于一个“触发器”，用于通知线程某个事件是否出现，包括有信号和无信号两个状态
- 互斥体对象
 - 互斥访问共享资源
- 信号量对象

- 就是资源信号量，初始值可在0到指定最大值之间设置

15.5.2 内核同步

- 在内核的不同阶段，必须保证每次有且仅有一个处理机在临界区执行
- 内核临界区就是修改全局数据结构的代码段
- 内核引入自旋锁实现多处理机互斥访问内核临界区
- 在Intel处理机上，自旋锁是通过“测试与设置”TestSet硬件指令实现的
- 拥有自旋锁的线程不会被剥夺处理机，其独占处理机执行

15.5.3 执行体同步

- 自旋锁的限制
 - 被保护的资源必须被快速访问，并且不与其他代码进行复杂的交互
 - 临界区代码不能换出内存，不能引用可分页数据，不能调用外部程序（包括系统服务），不能产生中断或异常
- 用户态同步机制
 - 调度程序对象
 - 内核以内核对象形式给执行体提供的用户态同步机制
 - 进程、线程、事件、信号量、互斥体、可等待的定时器、I/O完成端口或文件等同步对象
 - 每个同步对象有两种状态：“有信号”、“无信号”
 - Win32应用程序中的一个线程可以等待一个或多个同步对象变为有信号状态实现同步
 - 线程、进程终止时有信号
 - Windows定义了统一的同步机制，等待调度程序对象为有信号状态
 - WaitForSingleObject()

十六、Windows的存储器管理

16.1 存储器管理的基本概念

16.1.1 概述

- 存储管理器是执行体的一个组件
- 提供基本服务
 - 存储器管理需要的系统服务
 - 分配、释放、保护虚存和物理主存，写时复制，虚拟页信息等
 - 以Win32API或核心态设备驱动程序接口形式提供
 - 提供运行在核心态系统线程上的例程
 - 平衡工作集管理器
维护空闲主存数量不低于某一界限并及时调整进程的工作集
 - 进程/堆栈交换器
执行换入/换出操作

- 更改页写入器
将被修改页写回磁盘
- 废弃段线程
高速缓存和页文件的扩大与缩写
- 零页线程
维护系统有足够的零填充的空闲页存在

16.1.2 进程地址空间的布局

- 在32位的地址空间上，允许每个用户进程占有4G的线程虚拟地址空间
 - 低2GB为进程的私有地址空间
 - 每个进程的私有代码和数据
 - 高2GB为进程公有的操作系统空间
 - 系统范围的代码和数据
 - NTOSKRNL.exe, HAL, 引导程序
- Windows企业版有一个引导选项，允许用户拥有3GB的地址空间

16.1.3 进程私有空间的分配

- x86模型中，用全局描述符表(GDT)和局部描述符表(LDT)实现操作系统的公共空间和进程私有虚拟地址空间的分配
- 进程私有空间分配
 - 一部分映射物理内存
 - 使用虚拟地址描述符VAD记录进程空间
 - 一部分映射硬盘上的交换文件
 - 多个进程共享存储区 (区域对象)
- 虚拟地址描述符VAD
 - 当进程要求分配一块连续虚存时，系统并不立即构造页表，而是为其建立一个VAD结构，记录地址空间相关信息
 - 被分配的地址域的起始地址
 - 该域是共享的还是私有的
 - 该域的存取保护
 - 是否可继承
 - 进程页表的构建一直推迟到访问页时才建立
 - 一个进程的一组VAD结构构成一棵自平衡二叉树，便于快速查找
- 区域对象
 - 被称为文件映射对象，可被多个进程共享的存储区
 - 一个文件区域对象可被多个进程打开
 - 可利用区域对象映射磁盘上的文件 (包括页文件, 可执行文件)
 - 主要作用：
 - 执行体利用区域对象将一个可执行的映像装入主存。访问这个文件对象就像访问内存中的一个大数组，而不需要读/写操作
 - 高速缓冲管理区使用区域对象访问一个被缓冲文件的数据

- 进程使用区域对象将一个大于进程地址空间的文件映射到进程整个部分或部分地址空间中
- 映射类型
 - 数据文件映射
 - 可执行文件映射
 - 共享的高速缓存映射
- 映射过程
 - 先为被访问的文件创建一个区域对象，进程只需在自己的地址空间保留一部分空间，来映射该区域对象的一部分（视口）
- 区域对象的结构
 - 对象头
 - 对象体
 - 最大尺寸
区域对象的最大长度；如果映射到一个文件，则为文件大小；最大尺寸可达 2^{64} B
 - 页保护方式：创建区域时，分配给该区域的所有页的保护方式
 - 页文件（交换区）/映射文件：指出区域是否被创建为空，或是加载一个文件
 - 基准的/非基准的：基准则要求共享该区域的所有进程在相同的虚拟地址空间出现
 - 系统提供的对象服务
 - 检索和更改区域对象体中属性
- 页文件（交换区）
 - 作为主存补充的磁盘上的那部分空间

e.g.计算机有128MB物理主存，同时在磁盘上有256MB的页文件，那么就认为计算机拥有384MB主存

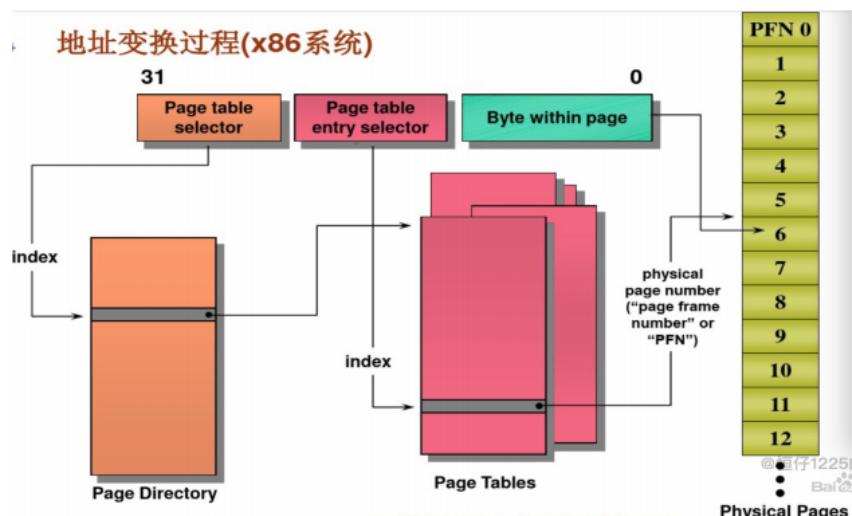
 - Windows支持多个页文件，当系统启动时，打开页文件，系统为每个页文件都维持一个打开的句柄
 - 一旦打开页文件，在系统运行区间不能删除
- Win32子系统实现文件映射的过程
 - CreateFile()
创建/打开一个被映射文件
 - CreateFileMapping()
创建一个与被映射文件大小相等的区域对象
 - MapViewOfFile()
将区域对象的一个视口映射到进程保留的某部分地址空间，之后进程就可以像访问主存一样访问文件
 - UnMapViewOfFile()
访问后接触被映射的视口
- 虚存的分配
 - 进程私有2G地址空间的地址域可能是
 - 空闲的：还没被使用过
 - 被保留：已预留虚存，还没分配物理主存
 - 被提交：已分配物理主存或交换区
 - 主存分配方法：两阶段
 - 先保留地址域，后提交物理主存
 - 也允许保留和提交同时实现
 - 保留地址空间

- 线程创建大的动态数据结构时，可以采用保留地址空间的方法在虚地址空间预留一块区域，以防止进程的其他线程占用这段连续的虚拟地址空间，并用一个虚拟地址描述符记录
- 试图访问保留的虚拟地址空间会造成访问冲突，由系统进行缺页处理
- 提交
 - 在保留的地址空间中分配物理内存，并建立虚实映射

16.2 Windows地址转换

16.2.1 地址转换涉及的数据结构

- 采用虚拟页式管理，页面大小默认为4KB
- 采用二级页表结构：页目录表、页表
 - 页表和页目录表的结构相同，是由页表项（PTE）或页目录项构成的数组
- 虚拟地址变换过程

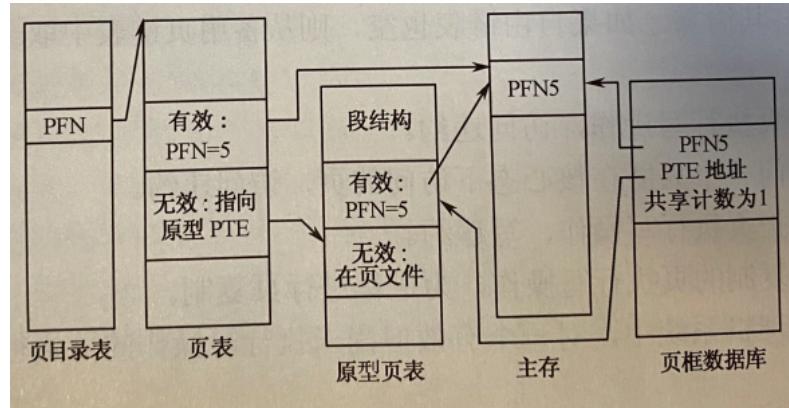


- 页框数据库
 - 页框数据库用于跟踪物理主存的使用
 - 是一个数组，其索引号从0到主存的页框总数-1
 - 内存页框有八种状态
 - 活动（有效）：是进程工作集的一部分
 - 转换：一个页框正处于I/O操作进行中
 - 备用：已不属于工作集，页表项仍然指向该页框，但被标记为正在转移的无效PTE
 - 更改：已不属于工作集，修改后未写磁盘，页表项仍指向该页框，被标记为正在转移的无效PTE
 - 更改不写入：更改但不写入磁盘
 - 空闲：不属于任何一个工作集
 - 零初始化：清零的初始页框
 - 坏页框
 - 页框链表
 - 为快速定位，形成6个页框链表
 - 零初始化
 - 空闲
 - 备用

- 更改
- 更改不写入
- 坏的
- 活动页框由进程页表管理
- 转换页框不在链表中

16.2.2 页错误处理

- 无效页处理
 - 当被访问的页无效时，产生无效页错误
 - 处理
 - 访问一个未知页，其页表项为0或者页表不存在（首次访问）
 - 系统为包含该地址的页创建一个页表
 - 从该进程的VAD树种查找包含该地址的VAD，填充页表项
 - 所访问的页不在主存，在外存页文件或映像文件中
 - 系统分配一个物理页框，将所需要页从磁盘中读出，并放入工作集中
 - 所访问的页在备用链表或更改链表
 - 将该页从指定链表中移出，放入进程或系统工作集
 - 访问一个请求零初始化的页
 - 页调度器查看零页表项是否空。是则从自由链表中取一页将其清零；若自由链表也为空，则从备用页链表中取一页将其清零，放入工作集中
 - 页访问违约，与访问权限不符
 - 对一个只读页执行写操作
 - 从用户态访问一个只能在核心态下访问的页
 - 对一个写保护页执行写操作
 - 对一个写时复制页执行写操作
 - 为进程进行页复制
 - 在多处理机系统中，对一个有效但未执行写操作的页执行写操作
 - 在页表项中将修改位置1
- 原型页表项
 - 当一个页框被两个或多个进程共享时，存储器管理器依靠一个称为“原型页表PTE”的结构记录这些被共享的页框
 - 引入
 - 尽可能减少对各进程页表的影响
 - 原型页表项位于页表和页框数据库之间，不负责地址转换，不会直接出现在页表中
 - 段结构
 - 区域对象有原型页表
 - 当进程首次访问区域对象视口中的页时，利用原型页表填写进程页表
 - 当共享页框变为有效时，进程页表项和原型页表项都指向该物理页框
 - 当共享页框变为无效时，进程页表中的页表项由一个特殊的页表项来填充，这个特殊的页表项指向描述该页的原型页表项



16.3 页调度策略

- 采用请求调页和集群方式
 - 产生缺页中断时，根据程序局部性原理，将所缺的页及其前后的一些页装入主存
- 置页策略
 - 产生缺页中断时，存储器管理器必须确定将调入的虚拟页放在物理内存的具体位置
- 置换策略
 - 在多处理器系统中，采用局部先进先出置换策略
 - 在单处理器系统中，更接近于最近最久未使用策略
- 进程工作集
 - 进程正在使用的物理页面的集合
 - 系统根据内存情况允许进程工作集规模增大或缩小
- 系统工作集
 - 为可换页的系统代码和数据分配一定数量的页框
- 平衡工作集管理器
 - 是一个系统线程，用来调整进程和系统工作集

十七、Windows的文件系统

17.1 文件系统概述

- FAT
 - FAT12
 - FAT16
 - FAT32
 - 12、16和32分别为描述磁盘块簇地址使用的位数
 - ExFAT
- NTFS
 - 使用64位的磁盘地址，理论上可以支持的最大分区位 2^{64} B
- FAT文件系统
 - 物理结构：链接

- FAT12
 - 文件卷最大为32MB，簇大小512B~8KB
- FAT16
 - 文件卷最大4GB（理论值），簇大小512B~64KB
- FAT32
 - 文件卷最大8TB（理论值），簇大小4KB~32KB
 - 描述磁盘块簇地址的实际有效位是2B
 - Windows2000限制文件卷最大为32GB
- NTFS卷结构
 - 分区引导扇区
 - 最多占用16个扇区，包含了卷的布局、文件系统结构、引导代码等
 - 主控文件表区
 - NTFS卷的管理控制中心，包含了卷上所有文件、目录及空闲未用盘簇信息
 - 文件数据区
- NTFS文件系统
 - 物理结构：索引顺序
 - 利用主控文件表中的索引表实现虚拟簇号VCN与逻辑簇号LCN之间的映射
 - 通过磁盘的逻辑簇号LCN引用文件在磁盘上的物理位置
 - 通过虚拟簇号VCN引用文件中的数据

17.2 主控文件表

17.2.1 主控文件表的结构

- 主控文件表 (MFT)
 - NTFS卷的管理控制核心
 - 包含
 - 系统引导程序
 - 用于定位和恢复卷中所有文件的数据结构
 - 记录整个卷的分配状态的位图（元数据）等
 - 元数据
- MFT结构
 - 由若干个记录构成，记录大小为1KB
 - 每个记录描述一个文件或目录
 - 前16个记录为NTFS元数据文件保留
 - 以"\$"开头的文件名
 - 当一个文件或目录太大的时候，可能占用多个记录
 - 文件基记录（第1个记录）
 - 存放同一个文件属性
 - 扩展记录

17.2.2 主控文件表的记录结构

- 记录结构
 - 一个记录头+若干 (属性, 属性值) 对
- 记录头
 - 用于有效性检查的魔数
 - 文件生成时的顺序号
 - 文件引用计数
 - 记录中实际使用的字节数
- 属性
 - 有名属性/无名属性 (文件内容)
 - NTFS负责读写有名属性, 应用程序读写实际文件数据 (无名属性)
 - 常驻属性/非常驻属性
 - 存放在文件记录中的属性为常驻属性
 - 属性头总是常驻内存
- 小文件
 - 所有属性常驻MFT
 - (\$DATA,文件内容)属性可以包含文件的所有数据

小文件的MFT记录

标准信息	文件名	安全描述体	文件数据
------	-----	-------	------

- 小目录
 - 所有属性常驻内存, 索引根属性包含其所有文件和子目录的目录项 (文件引用索引)
 - 文件目录项包括: 64位文件引用号时间, 大小等信息, 以提高目录浏览速度

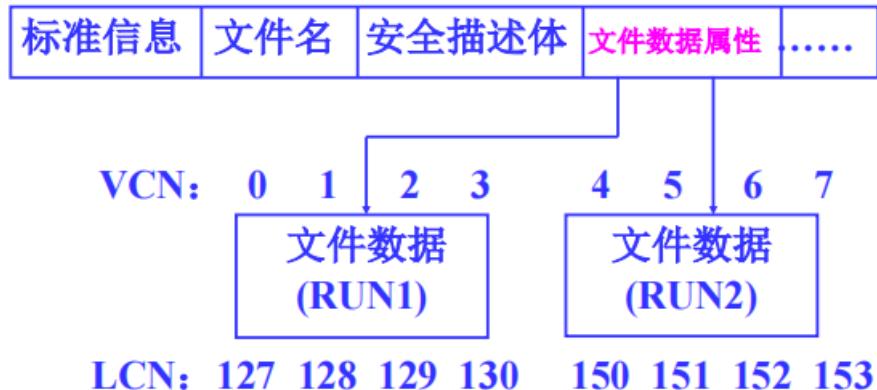
小目录的MFT记录

标准信息	目录名	安全描述体	索引根 (文件索引)			空
			文件1	文件2	文件3	

- 大文件
 - MFT+单独存储文件内容的区域
 - 存放文件数据的区域 (称为一个运行run) 或一个扩展extent
 - 文件数据属性
 - 逻辑簇号LCN: 对分区的第一个簇到最后一个簇进行编号, NTFS使用逻辑簇号对簇进行定位
 - 虚拟簇号VCN: 将文件所占用的簇从头到尾进行编号的, 虚拟簇号不要求在物理上是连续的

- 属性头中包含LCN与VCN的映射关系

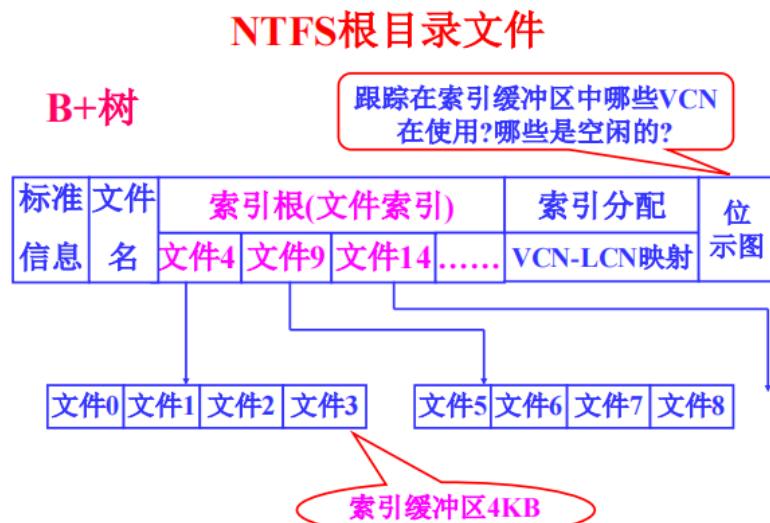
开始的VCN	开始的LCN	簇数
0	127	4
4	150	4



17.3 NTFS文件的引用和索引

- 大目录

- 一部分索引在索引根属性里，剩余的索引存储在索引缓冲区的非常驻run中
- 文件名实际存储在固定4KB大小的索引缓冲区中，每个索引缓冲区可容纳20到30个文件目录项
- 索引缓冲区采用B+树实现
 - 索引根属性包含B+树的第一级并指向包含下一级的索引缓冲区



- 文件的引用

- 每个文件有一个64位的文件引用号，系统通过文件引用号引用文件
- 文件引用由文件号和文件顺序号组成
 - 文件号对应于该文件在MFT中的索引位置
 - 文件顺序号是文件记录被重复使用的次数，是为了进行内部一致性检查而设计的

