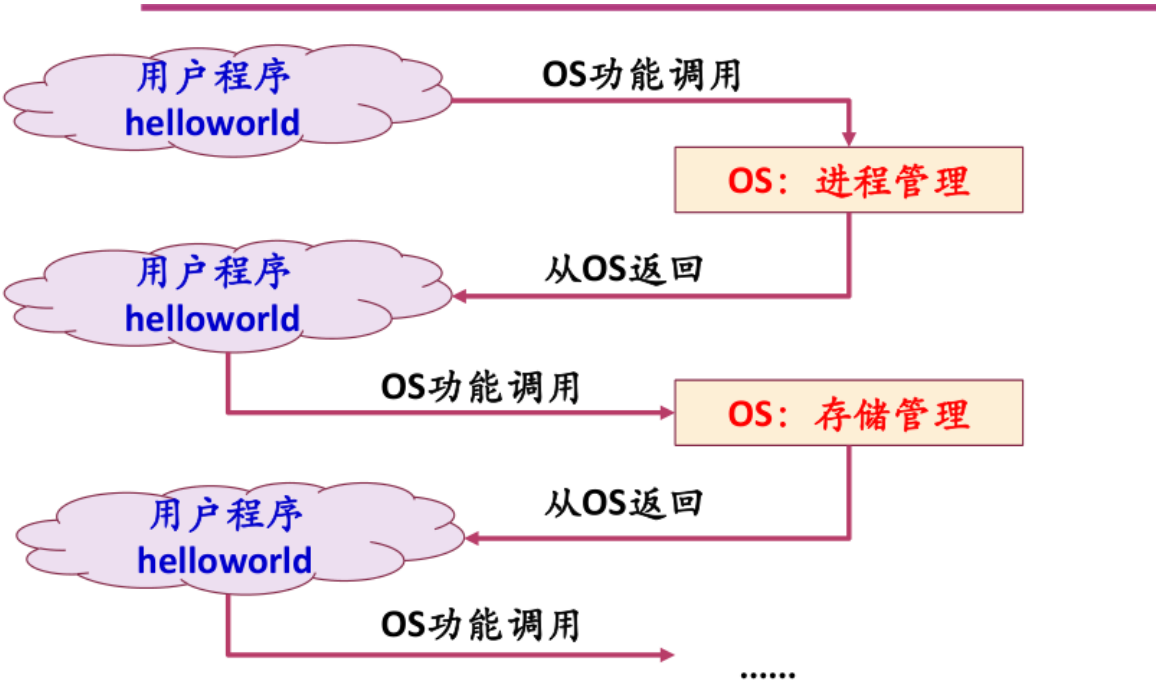
1. 操作系统概述

1.1 操作系统做了什么

以 hellowworld 程序执行为例：

1. 用户告诉操作系统执行 helloworld 程序（命令行键入，鼠标双击，etc
2. 操作系统接到用户请求后，找到 helloworld 程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址（文件格式？Windows: PE; Linux: ELF）
3. 操作系统：创建一个新的进程，并将 helloworld 可执行文件映射到该进程结构，表示由该进程执行 helloworld 程序
4. 操作系统：为helloworld程序设置CPU上下文环境，并跳到程序开始处（假设调度程序选中hello程序）
5. 执行helloworld程序的第一条指令，发生缺页异常
6. 操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行helloworld程序
7. helloworld程序执行puts函数（系统调用），在显示器上写一字符串
8. 操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程
9. 操作系统：控制设备的进程告诉设备的窗口系统它要显示字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区
10. 视频硬件将像素转换成显示器可接收的一组控制/数据信号
11. 显示器解释信号，激发液晶屏，屏幕上显示 “hello world”



1.2 操作系统的定义与作用

操作系统是什么

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

- 它们能以尽量有效、合理的方式组织和管理计算机的软硬件资源
- 合理地组织计算机的工作流程，控制程序的执行并向用户提供各种服务功能
- 使得用户能够灵活、方便地使用计算机，使整个计算机系统高效率运行

关键词：

- **有效**，系统效率，资源利用率：CPU利用率充足与否？I/O设备是否忙碌？
- **合理**，各种软硬件资源的管理是否公平合理：如果不公平、不合理，则可能会产生问题？
- **方便使用**，两种角度：用户界面 与 编程接口

操作系统的三个作用

1. 自底向上 → 资源的管理者 → 有效

- 硬件资源：CPU，内存，设备（I/O设备、磁盘、时钟、网络卡等）
- 软件资源：磁盘上的文件、各类管理信息等

怎么管理资源：跟踪记录资源的使用状况：

如：哪些资源空闲，分配给谁使用，允许使用多长时间等

- 确定资源分配策略——算法
  - 静态分配策略
  - 动态分配策略
- 实施资源的分配和回收
- 提高资源利用率
- 保护资源的使用
- 协调多个进程对资源请求的冲突

从资源管理的角度： 五大基本功能

- 进程/线程管理（CPU管理）
  进程线程状态、控制、同步互斥、通信、调度、……
- 存储管理
  分配/回收、地址转换、存储保护、内存扩充、……
- 文件管理
  文件目录、文件操作、磁盘空间、文件存取控制、……
- 设备管理
  设备驱动、分配回收、缓冲技术、……
- 用户接口
  系统命令、编程接口

2. 向用户提供各种服务 → 方便使用

- 在操作系统之上，从用户角度来看：
  操作系统为用户提供了一组功能强大、方便易用的命令或系统调用
- 典型的服务
  进程的创建、执行；文件和目录的操作；I/O设备的使用；各类统计信息；……

3. 对硬件机器的扩展 → 扩展能力

操作系统为硬件基础上的第一层软件，为应用程序提供了各种各样的接口，避免与简化与硬件相关的复杂繁琐的操作。

操作系统在应用程序与硬件之间建立了一个等价的扩展机器（虚拟机）以实现对硬件抽象，提高可移植性，比底层硬件更容易编程。

1.3 操作系统的主要特征

并发 (concurrency)

并发指处理多个同时性活动的能力，由于并发将会引发很多的问题：

- 活动切换、保护、相互依赖的活动间的同步
- 在计算机系统中同时存在多个程序运行，单CPU上
  - 宏观上：这些程序同时在执行
  - 微观上：任何时刻只有一个程序真正在执行，即这些程序在CPU上是轮流执行的

并行(parallel)：与并发相似，但多指不同程序同时在多个硬件部件上执行

共享 (sharing)

- 操作系统与多个用户的程序共同使用计算机系统中的资源（共享有限的系统资源）
- 操作系统要对系统资源进行合理分配和使用
- 资源在一个时间段内交替被多个进程所用

- 互斥共享（如打印机）
- 同时共享（如可重入代码、磁盘文件）
  问题：资源分配难以达到最优化，如何保护资源

虚拟 (Virtual)

- 一个物理实体映射为若干个对应的逻辑实体 - 分时或分空间
- 虚拟是操作系统管理系统资源的重要手段，可提高资源利用率

- CPU - - 每个进程的"虚处理机"
- 存储器 - - 每个进程都有独立的虚拟地址空间（代码 + 数据 + 堆栈）
- 显示设备 - - 多窗口或虚拟终端

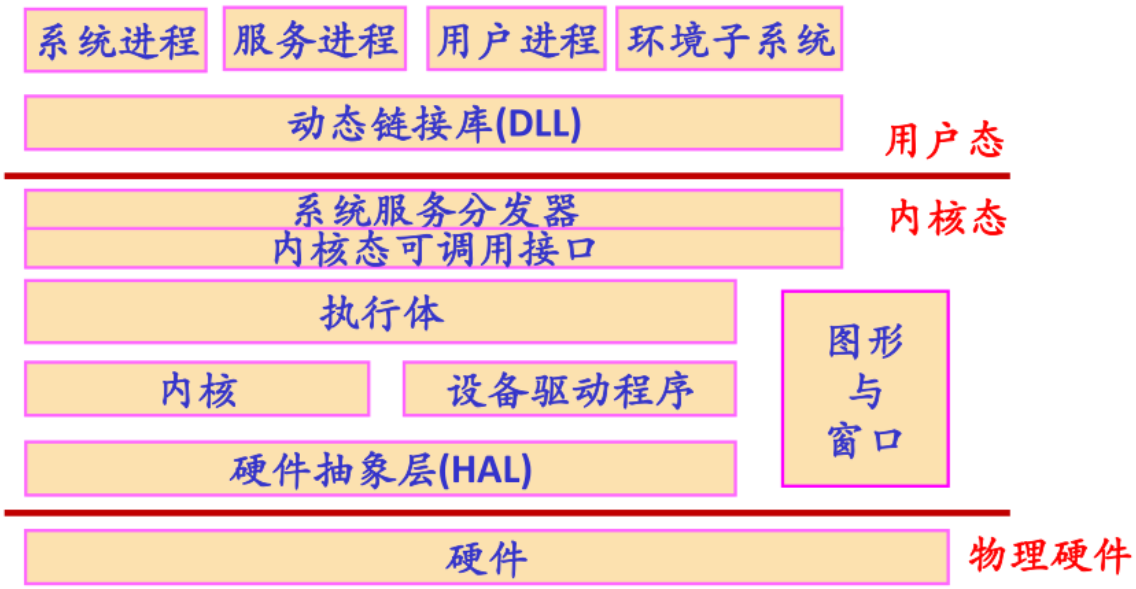
随机

操作系统必须随时对以不可预测的次序发生的事件进行响应并处理

- 进程的运行速度不可预知：多个进程并发执行，“走走停停”，无法预知每个进程的运行推进的快慢
- 难以重现系统在某个时刻的状态（包括重现运行中的错误）

1 .4 典型操作系统的架构

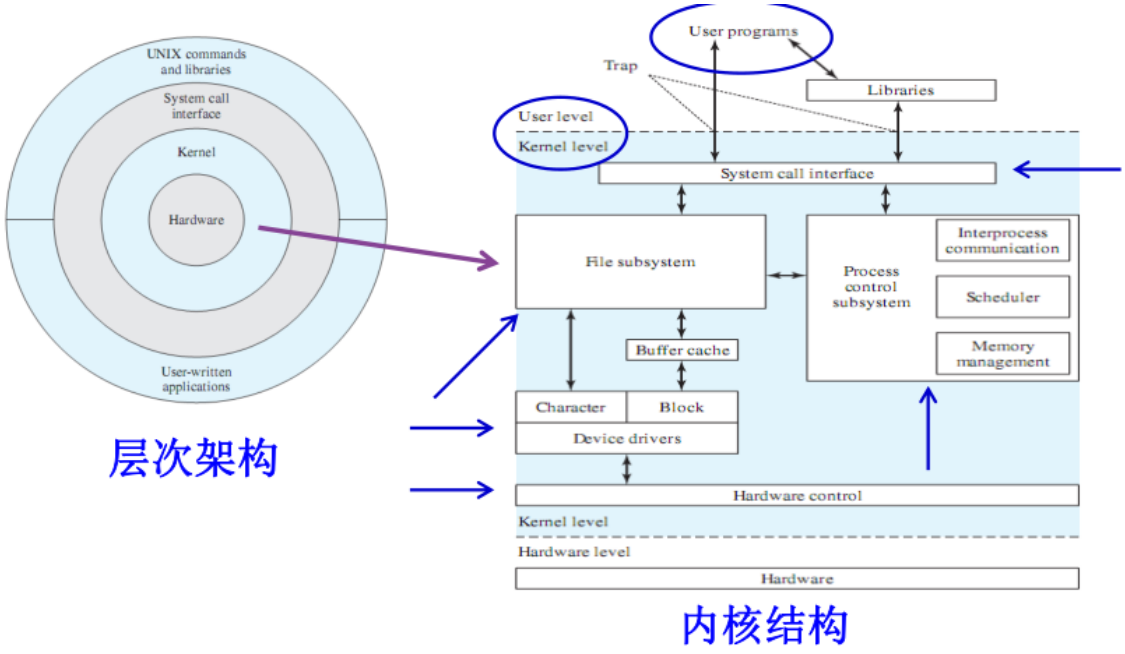
WINDOWS 操作系统的体系结构



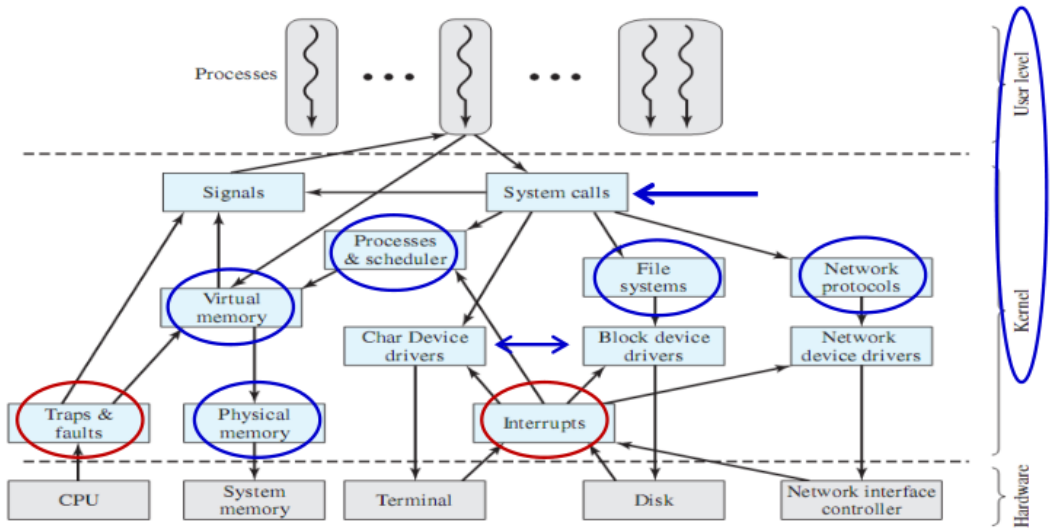
继续抽象为：

应用程序
* 系统功能调用
操作系统
计算机硬件

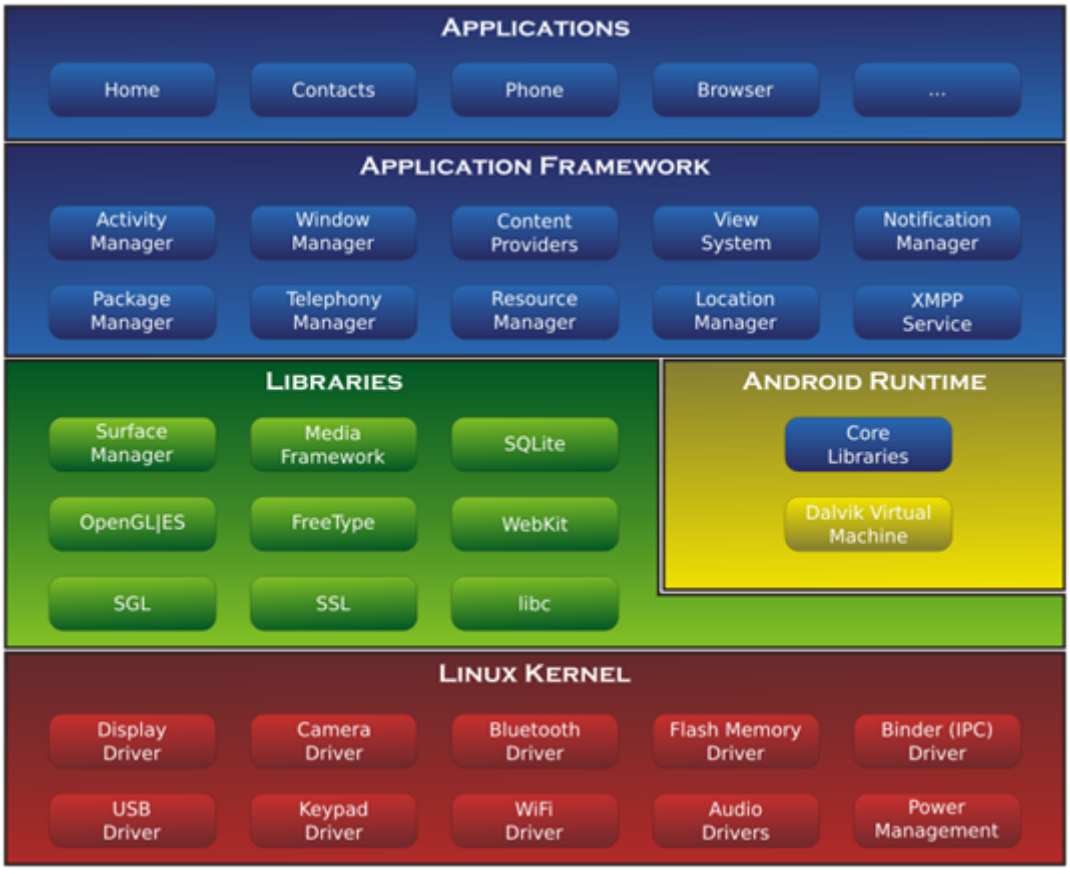
UNIX操作系统的体系结构



Linux 操作系统的体系结构



ANDROID架构



1.5 操作系统的分类

操作系统发展是随着计算机硬件技术、应用需求的发展、软件新技术的出现而发展的

目标：**充分利用硬件 + 提供更好的服务**

批处理操作系统

工作方式

1. 用户将作业交给系统操作员
2. 系统操作员将许多用户的作业组成一批作业，输入到计算机系统中，在系统中形成一个自动转接的连续的作业流
3. 启动操作系统
4. 系统自动、依次执行每个作业
5. 由操作员将作业结果交给用户

追求目标：提高资源利用率，增加作业处理吞吐量

批处理系统中的作业包括：

- 用户程序
- 数据
- 作业说明书（用作业控制语言编写）

成批：通常由若干个作业组成，用户提交作业后只能等待处理结果，不能干预自己作业的执行

批作业处理：对一批作业中的每个作业进行相同的处理：从磁带读入用户作业和编译链接程序，编译链接用户作业以生成可执行程序；启动执行；执行并输出结果

- **问题**：慢速的输入输出处理直接由主机来完成，输入输出时，CPU处于等待状态
- **解决方案**：卫星机：完成面向用户的输入输出（纸带或卡片），中间结果暂存在磁带或磁盘上

单道批处理系统（simple batch processing, uni-programming)

多道批处理系统（multiprogramming system）

SOIILING技术（Simultaneous Peripheral Operation On-Line，同时的外围设备联机操作，又称假脱机技术）：利用磁盘作缓冲，将输入、计算、输出分别组织成独立的任务流，使I/O和计算真正并行

- 用户作业加载到磁盘上的输入井
- 按某种调度策略选择几个搭配得当的作业，调入内存
- 作业运行的结果输出到磁盘上的输出井
- 运行结果从磁盘上的输出井送到打印机

打印机采用的就是SOIILING技术

分时系统(TIME-SHARING SYSTEM)

- 时间片 (time slice)：操作系统将CPU的时间划分成若干个片段，称为时间片
    - 操作系统以时间片为单位，轮流为每个终端用户服务，每次服务一个时间片
    - 其特点是利用人的错觉，使用户感觉不到计算机在服务他人
      追求目标：及时响应(依据是响应时间)
      响应时间：从终端发出命令到系统给予回答所经历的时间

通用操作系统

分时系统与批处理系统结合

- 原则：分时优先，批处理在后
    - “前台”：需要频繁交互的作业
    - “后台”：时间性要求不强的作业

实时操作系统

指使计算机能及时响应外部事件的请求，在规定的严格时间内完成对该事件的处理，并控制所有实时设备和实时任务协调一致地工作

分类：

- 第一类：实时过程控制：工业控制、航空、军事控制、...
- 第二类：实时通信（信息）处理：电讯（自动交换机）、银行、飞机订票、股市行情

追求目标：

- 对外部请求在严格时间范围内作出响应
- 高可靠性

特征：

- 关键参数 是 时间
  例子：工业过程控制系统——汽车装配线
- 硬实时系统：某个动作绝对必须在规定的时刻或时间范围完成 (汽车刹车等)
- 软实时系统：接受偶尔违反最终时限 (MP3视频播放等)

个人计算机操作系统

计算机在某一时间内为单用户服务

- 追求目标：界面友好，使用方便；丰富的应用软件

网络操作系统

基于计算机网络，在各种计算机操作系统上，按网络体系结构协议标准开发的软件

- 功能：网络管理，通信，安全，资源共享和各种网络应用
- 追求目标：相互通信，资源共享

分布式操作系统

分布式系统：或以计算机网络为基础，或以多处理机为基础，基本特征是处理分布在不同计算机上

- 分布式操作系统：是一个统一的操作系统，允许若干个计算机可相互协作共同完成一项任务。操作系统可将各种系统任务在分布式系统中任何处理机上运行，自动实现全系统范围内的任务分配、自动调度、均衡各处理机的工作负载
- 处理能力增强、速度更快、可靠性增强、具有透明性

嵌入式操作系统

- 嵌入式系统
  - 在各种设备、装置或系统中，完成特定功能的软硬件系统汽车、手机、电视机、MP3播放器
  - 它们是一个大设备、装置或系统中的一部分，这个大设备、装置或系统可以不是“计算机”
  - 通常工作在反应式或对处理时间有较严格要求环境中
- 嵌入式操作系统（Embedded Operating System）
  运行在嵌入式系统环境中，对整个嵌入式系统以及它所操作、控制的各种部件装置等等资源进行统一协调、调度、指挥和控制的系统软件

操作系统的另一神分类（ TANENBAUM）

大型机操作系统
服务器操作系统
多处理机操作系统
个人计算机操作系统
掌上计算机操作系统
嵌入式操作系统
传感器节点操作系统
实时操作系统
智能卡操作系统

2. 操作系统运行环境与运行机制 *

操作系统如何理解使用控制硬件

2.1 处理器状态

中央处理器（CPU）

处理器由运算器、控制器、一系列的寄存器以及高速缓存构成

- 两类寄存器：
  - 用户可见寄存器：高级语言编译器通过优化算法分配并使用之，以减少程序访问内存次数
  - 控制和状态寄存器：用于控制处理器的操作通常由操作系统代码使用
    - 用于控制处理器的操作
    - 在某种特权级别下可以访问、修改

    常见的控制和状态寄存器：
    - 程序计数器（PC：Program Counter），记录将要取出的指令的地址
    - 指令寄存器（IR：Instruction Register），记录最近取出的指令
    - 程序状态字（PSW：Program Status Word），记录处理器的运行状态如条件码、模式、控制位等信息

操作系统的需一保护

- 从操作系统的特征考虑：并发、共享
- 提出要求 → 实现保护与控制

需要硬件提供基本运行机制：

- 处理器具有特权级别，能在不同的特权级运行的不同指令集合
- 硬件机制可将OS与用户程序隔离

处理器的状态（模式MODE）

- 现代处理器通常将CPU状态设计划分为两种、三种或四种。
- 如何知道处理器运行在哪一种状态?

  在程序状态字寄存器PSW中专门设置一位，根据运行程序对资源和指令的使用权限而设置不同的CPU状态

特权指令和非特权指令

硬件提供了不同的CPU状态，而操作系统需要两种CPU状态：

- 内核态(Kernel Mode)：运行操作系统程序
- 用户态(User Mode)：运行用户程序

两个状态执行不同的指令集合：

操作系统可以使用特权指令和非特权指令，用户程序只能使用非特权指令

- 特权(privilege)指令：只能由操作系统使用、用户程序不能使用的指令
- 非特权指令：用户程序可以使用的指令

下列哪些是特权指令？哪些是非特权指令？

- 特权指令：**启动I/O**、 **内存清零**、**修改程序状态字**、**设置时钟**、**允许/禁止中断**、**停机**
- 非特权指令：控制转移 、算术运算、访管指令（使得用户程序从用户态陷入操作系统内核态）、取数指令

实例：X86系列处理器

X86支持4个处理器特权级别，称之为特权环：R0、R1、R2和R3

希望在不同级别运行不同的程序

- 从R0到R3，特权能力由高到低
- R0相当于内核态；R3相当于用户态；R1和R2则介于两者之间
- 不同级别能够运行的指令集合不同
- 目前大多数基于x86处理器的操作系统只用了R0和R3两个特权级别

CPU状态之间的转换

- 用户态 → 内核态：唯一途径 → 中断/异常/陷入机制
- 内核态 → 用户态：只需设置程序状态字PSW
- 一条特殊的指令：陷入指令（又称访管指令）~~（因为内核态也被称为supervisor mode，即管理态~~

  提供给用户程序的接口，用于调用操作系统的功能（服务）
  例如：int，trap，syscall，sysenter/sysexit

2.2 中断异常机制介绍 （Interrupt Exception）

中断异常机制是操作系统的驱动力

中断/异常机制

- 中断/异常 对于操作系统的重要性就好比：汽车的发动机、飞机的引擎
- 可以说 操作系统是由“中断驱动”或者 “事件驱动”的
- 主要作用
  - 及时处理设备发来的中断请求
  - 可使OS捕获用户程序提出的服务请求
  - 防止用户程序执行过程中的破坏性活动
  - … … 等等

中断/异常 的概念

中断/异常即CPU对系统发生的某个事件作出的一种反应，事件的发生改变了
处理器的控制流

- 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：
    - 是随机发生的
    - 是自动处理的(由硬件来完成)
    - 是可恢复的（被打断的事件可以在以后某个时刻继续）

为什么引入中断与异常（历史背景

中断的引入是为了支持CPU和设备之间的并行操作。

早期计算机系统在没有中断机制时要负责对设备所有工作的管理。

有了中断机制后：

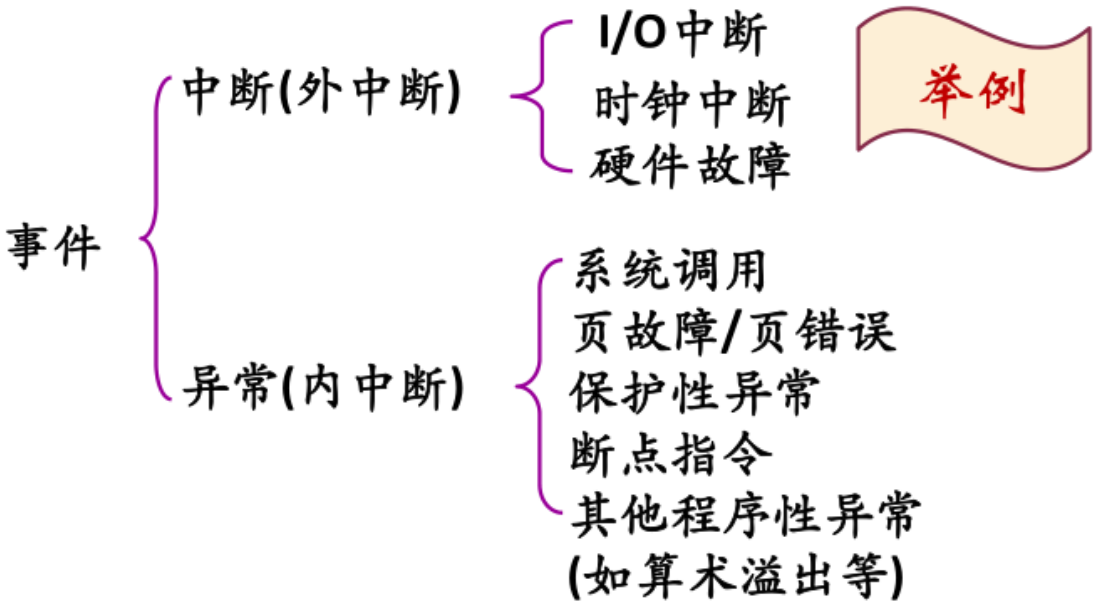
- 当CPU启动设备进行输入/输出后，设备便可以独立工作，CPU转去处理与此次输入/输出不相关的事情；当设备完成输入/输出后，通过向CPU发中断报告此次输入/输出的结果，让CPU决定如何处理以后的事情
- 异常的引入：表示CPU执行指令时本身出现的问题
    - 如算术溢出、除零、取数时的奇偶错，访存地址时越界或执行了“陷入指令” 等，这时硬件改变了CPU当前的执行流程，转到相应的错误处理程序或异常处理程序或执行系统调用

早期不区分中断异常，统称为中断，由于发生原因与处理过程不同，有了区分

中断与异常的划分

中断与异常可以统称为事件

中断：外部事件，正在运行的程序所不期望的
异常：由正在执行的指令引发



中断与异常的小结

类别	原因	异步/同步	返回行为
中断 Interrupt	来自I/O设备、其他硬件部件	异步	总是返回到下一条指令
陷入Trap	有意识安排的	同步	返回到下一条指令
故障Fault	可恢复的错误	同步	返回到当前指令
终止Abort	不可恢复的错误	同步	不会返回

2.3 中断与异常机制工作原理

工作原理

中断/异常机制是现代计算机系统的核心机制之一：硬件和软件相互配合而使计算机系统得以充分发挥能力

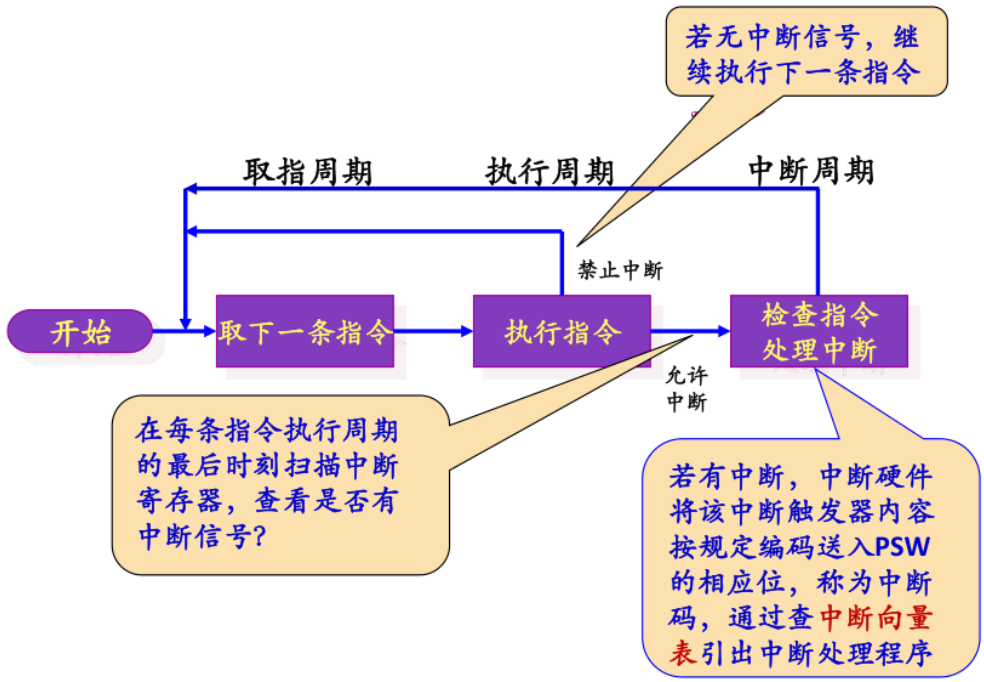
- 硬件该做什么事？ —— 中断/异常响应
  捕获中断源发出的中断/异常请求，以一定方式响应，将处理器控制权交给特定的处理程序
- 软件要做什么事？ —— 中断/异常处理程序
  识别中断/异常 类型并完成相应的处理

中断响应：

发现中断、接收中断的过程
由中断硬件部件完成

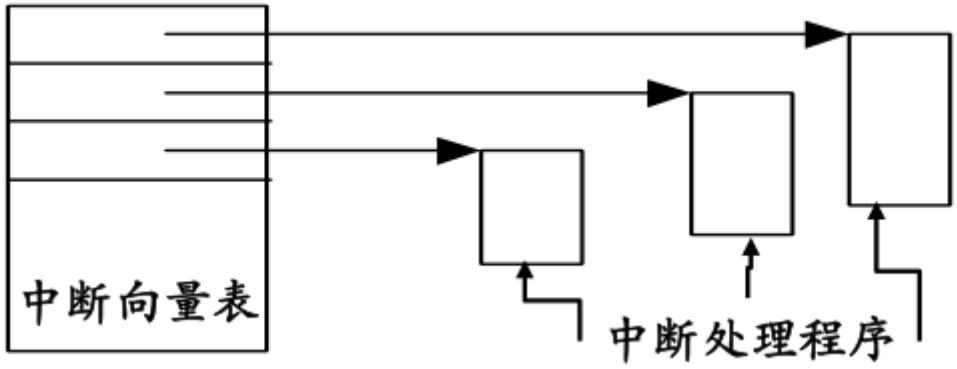
处理器控制部件中 设有 中断寄存器

CPU何时响应中断?



中断向量表

中断向量表每一行为一个中断向量，由中断向量组成。



中断向量：一个内存单元，存放中断处理程序入口地址和程序运行时所需的处理机状态字

执行流程按中断号/异常类型的不同，通过中断向量表转移控制权给中断处理程序

LINUX中的中断向量表

向量范围	用途
0～19	不可屏蔽中断和异常
20～31	Intel保留
32～127	外部中断（IRQ）
128（0x80）	用于系统调用的可编程异常
129～238	外部中断
239	本地APIC时钟中断
240	本地APIC高温中断
241～250	Linux保留
251～253	处理器间中断
254	本地APIC错误中断
255	本地APIC伪中断

不可屏蔽中断/异常

- 0 -- 除零
- 1 -- 单步调试
- 4 -- 算术溢出
- 6 --非法操作数
- 12 -- 栈异常
- 13 -- 保护性错误
- 14 -- 缺页异常

中断处理流程

1. 设备发中断信号
2. 硬件保存现场
3. 根据中断码查表
4. 把中断处理程序入口地址等推送到相应的寄存器
5. 执行中断处理程序

中断处理程序

设计操作系统时，为每一类中断/异常事件编好相应的处理程序，并设置好中断向量表

中断处理程序由软件提前设置好，硬件部件来执行

- 系统运行时若响应中断，中断硬件部件将CPU控制权转给中断处理程序：
  - 保存相关寄存器信息
  - 分析中断/异常的具体原因
  - 执行对应的处理功能
  - 恢复现场，返回被事件打断的程序

以设备输入输出中断为例：

硬件：

- 打印机给CPU发中断信号
- CPU处理完当前指令后检测到中断，判断出中断来源并向相关设备发确认信号
- CPU开始为软件处理中断做准备：
  - 处理器状态被切换到内核态
  - 在系统栈中保存被中断程序的重要上下文环境，主要是程序计数器PC、程 序状态字PSW
- CPU根据中断码查中断向量表，获得与该中断相关的处理程序的入口地址，并将PC设置成该地址，新的指令周期开始时，CPU控制转移到中断处理程序

软件：

- 中断处理程序开始工作
  - 在系统栈中保存现场信息
  - 检查I/O设备的状态信息，操纵I/O设备或者在设备和内存之间传送数据等等

硬件：

- 中断处理结束时，CPU检测到中断返回指令，从系统栈中恢复被中断程序的上下文环境 ，CPU状态恢复成原来的状态，PSW和PC恢复成中断前的值，CPU开始一个新的指令周期

软件做了什么工作：

通常分为两类处理：

- I/O操作正常结束
  - 若有程序正等待此次I/O的结果，则应将其唤醒
  - 若要继续I/O操作，需要准备好数据重新启动I/O
- I/O操作出现错误
  - 需要重新执行失败的I/O操作
  - 重试次数有上限，达到时系统将判定硬件故障

2.4 x86 的中断与异常机制

- 中断：x86中，中断指由硬件信号引发，分为可屏蔽和不可屏蔽中断
- 异常：x86中，异常指由指令执行引发，比如除零异常
  - 80x86处理器发布了大约20种不同的异常
  - 对于某些异常，CPU会在执行异常处理程序之前产生硬件出错码，并压入内核态堆栈
- 系统调用：异常的一种，用户态到内核态的唯一入口

x86处理对中断的支持

中断控制器（PIC或APIC）

- 负责将硬件的中断信号转换为中断向量，并引发CPU中断

x86分为实模式和保护模式，实模式与保护模式下中断向量表有不同的称呼

- 实模式：中断向量表 (Interrupt Vector)
  - 存放中断服务程序的入口地址
  - 入口地址 = 段地址左移4位 + 偏移地址
  - 不支持CPU运行状态切换
  - 中断处理与一般的过程调用相似
- 保护模式：中断描述符表 (Interrupt Descriptor Table)

  采用门 (gate) 描述符数据结构表示中断向量

  中断向量表/中断描述符表
  - 四种类型门描述符
    - 任务门(Task Gate)
    - 中断门(Interrupt Gate)：

      给出段选择符 (Segment Selector)、中断/异常程序的段内偏移量 (Offset)
      通过中断门后系统会自动禁止中断
    - 陷阱门(Trap Gate)：

      与中断门类似，但通过陷阱门后系统不会自动禁止中断
    - 调用门(Call Gate)

中断/异常的硬件处理过程：

- 确定与中断或异常关联的向量i
- 通过IDTR寄存器找到IDT表，获得中断描述符（表中的第i项）
- 从GDTR寄存器获得GDT的地址；结合中断描述符中的段选择符，在GDT表获取对应的段描述符；从该段描述符中得到中断或异常处理程序所在的段基址
- 特权级检查

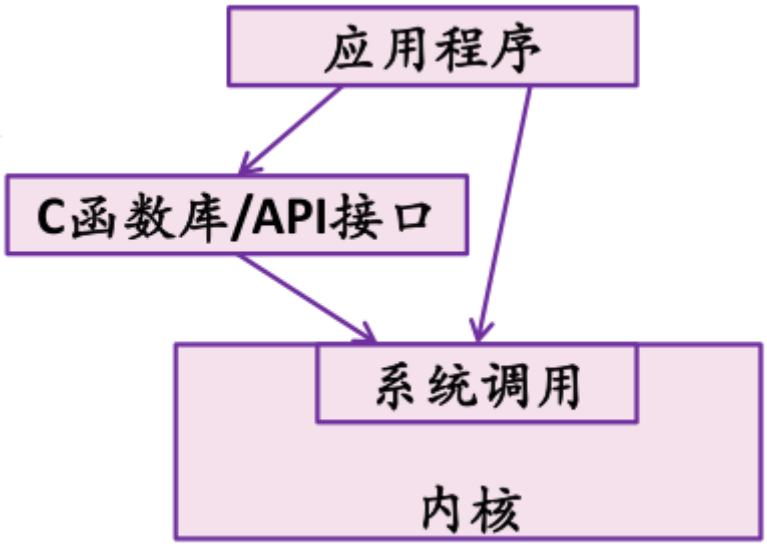
- 检查是否发生了特权级的变化，如果是，则进行堆栈切换(必须使用与新的特权级相关的栈)
- 硬件压栈，保存上下文环境；如果异常产生了硬件出错码，也将它保存在栈中
- 如果是中断，清IF位
- 通过中断描述符中的段内偏移量和段描述符中的基地址，找到中断/异常处理程序的入口地址，执行其第一条指令

2.5 系统的调用机制

- 系统调用是什么?

  系统调用：用户在编程时可以调用的操作系统功能
- 系统调用的作用
  - 系统调用是操作系统提供给编程人员的唯一接口
  - 使CPU状态从用户态陷入内核态
- 典型系统调用举例

  每个操作系统都提供几百种系统调用（进程控制、进程通信、文件使用、目录操作、设备管理、信息维护等)
- 系统调用与C库函数/API函数和内核函数的关系



系统调用机制的设计

- 中断/异常机制：支持系统调用服务的实现
- 选择一条特殊指令：陷入指令(亦称访管指令)
  引发异常，完成用户态到内核态的切换
- 系统调用号和参数：每个系统调用都事先给定一个编号(功能号)
- 系统调用表：存放系统调用服务例程的入口地址

参传递过程问题

怎样实现用户程序的参数传递给内核?
常用的3种实现方法：

- 由陷入指令自带参数：陷入指令的长度有限，且还要携带系统调用功能号，只能自带有限的参数
- **通过通用寄存器传递参数：这些寄存器是操作系统和用户程序都能访问的，但寄存器的个数会限制传递参数的数量** 大部分情况采用此种
- 在内存中开辟专用堆栈区来传递参数

当CPU执行到特殊的陷入指令时：

- 中断/异常机制：硬件保护现场；通过查中断向量表把控制权转给系统调用总入口程序
- 系统调用总入口程序：保存现场；将参数保存在内核堆栈里；通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数
- 执行系统调用例程
- 恢复现场，返回用户程序

2.6 x86 的 Linux 系统调用

以Linux为例，基于 x86处理器 的Linux调用是怎么实现的

- 陷入指令选择128号: `int $0x80`
- 门描述符

- 系统初始化时：对IDT表中的128号门初始化
- 门描述符的2、3两个字节：内核代码段选择符0、1、6、7四个字节：偏移量（指向system_call()）
- 门类型：15，陷阱门(执行系统调用的过程中允许执行中断)
- DPL：3，与用户级别相同，允许用户进程使用该门描述符

系统调用号示例：

```
#define __NR_exit       1
#define __NR_fork       2
#define __NR_read       3
#define __NR_write      4
#define __NR_open       5
#define __NR_close      6
#define __NR_waitpid    7
#define __NR_creat      8
#define __NR_link       9
#define __NR_unlink     10
#define __NR_execve     11
#define __NR_chdir      12
#define __NR_time       13
```

系统执行 INT $0x80指令

- 由于特权级的改变，要切换栈
  用户栈 → 内核栈
  CPU从任务状态段TSS中装入新的栈指针（SS：ESP），指向内核栈
- 用户栈的信息（SS：ESP）、EFLAGS、用户态CS 、EIP 寄存器的内容压栈（返回用）
- 将EFLAGS压栈后，复位TF，IF位保持不变
- 用128在IDT中找到该门描述符，从中找出段选择符装入代码段寄存器CS
- 代码段描述符中的基地址 + 陷阱门描述符中的偏移量 → 定位 system_call()的入口地址

中断发生后OS底层工作步骤

1. 硬件压栈：程序计数器等
2. 硬件从中断向量装入新的程序计数器等
3. 汇编语言过程保存寄存器值
4. 汇编语言过程设置新的堆栈
5. C语言中断服务程序运行（例：读并缓冲输入）
6. 进程调度程序决定下一个将运行的进程
7. C语言过程返回至汇编代码
8. 汇编语言过程开始运行新的当前进程

3. 进程线程模型

进程模型：什么是进程？操作系统在设计进程线程模型主要考虑哪些问题
线程模型：为什么引入线程？操作系统在支持线程方面都要做哪些工作？

3.1 进程的基本概念

- 多道程序设计：允许多个程序同时进入内存并运行，其目的是为了提高系统效率
- 并发环境：一段时间间隔内，单处理器上有两个或两个以上的程序**同时处于开始运行但尚未结束的状态，并且次序不是事先确定的**
- 并发程序：在并发环境中执行的程序

进程 (Process) 的定义

进程是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位，又称 任务（Task or Job）

- 程序的一次执行过程
- 是正在运行程序的抽象
- 将一个CPU变幻成多个虚拟的CPU
- 系统资源以进程为单位分配，如内存、文件、……每个具有独立的地址空间
- 操作系统将CPU调度给需要的进程

进程控制块PCB(Process Control Block)

- 进程控制块PCB 又称 进程描述符、进程属性
    - 操作系统用于管理控制进程的一个专门数据结构
    - 记录进程的各种属性，描述进程的动态变化过程
- PCB是系统感知进程存在的唯一标志：→ 进程与PCB是一一对应的
- 进程表：所有进程的PCB集合

PCB的内容应该包括什么?

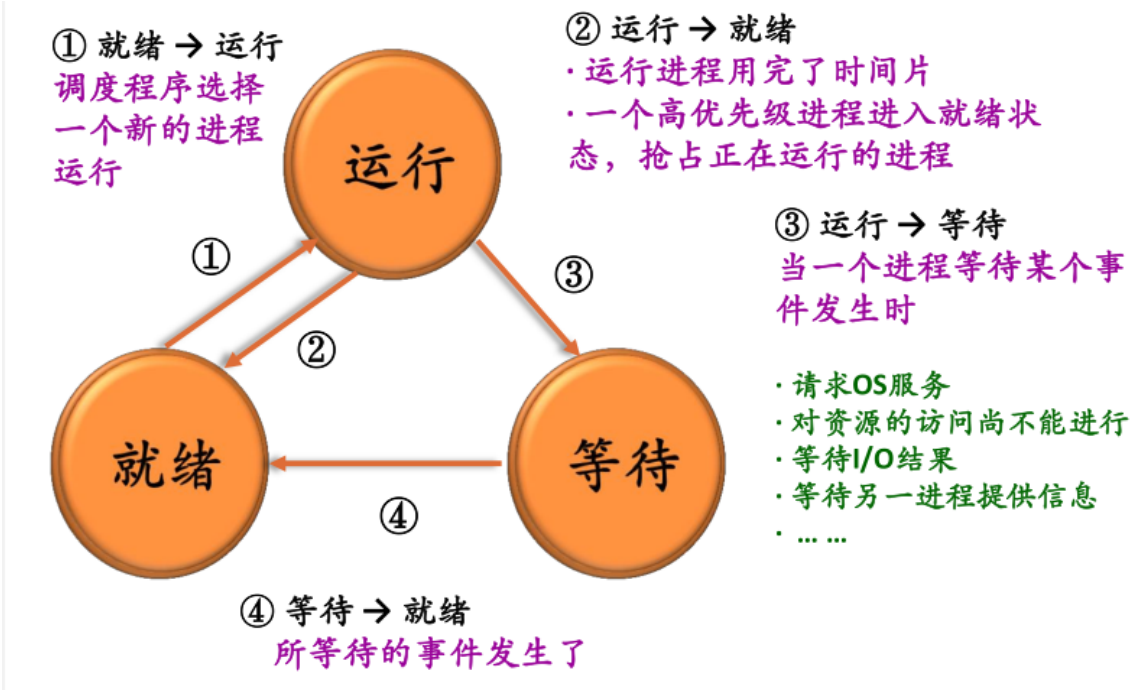
- 进程描述信息
    - 进程标识符(process ID)，唯一，通常是一个整数
    - 进程名，通常基于可执行文件名，不唯一
    - 用户标识符(user ID)
    - 进程组关系
- 进程控制信息
    - 当前状态
    - 优先级(priority)
    - 代码执行入口地址
    - 程序的磁盘地址
    - 运行统计信息(执行时间、页面调度)
    - 进程间同步和通信
    - 进程的队列指针
    - 进程的消息队列指针
- 所拥有的资源和使用情况
    - 虚拟地址空间的状况
    - 打开文件列表
- CPU现场信息
    - 寄存器值(通用寄存器、程序计数器PC、程序状态字PSW、栈指针)
    - 指向该进程页表的指针

3.2 进程状态及状态转换

三状态模型、五状态模型、七状态模型

进程的三种基本状态：

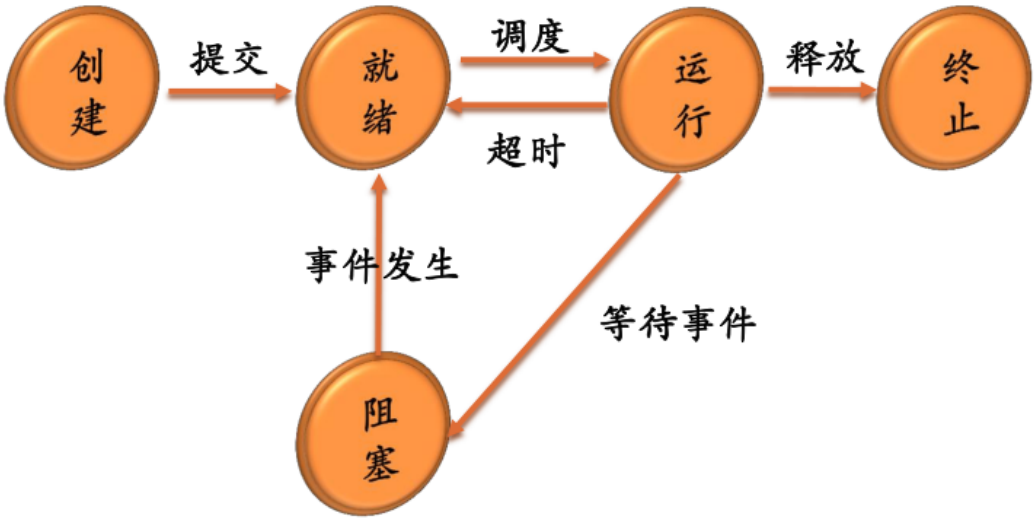
- 运行态（Running）:占有CPU，并在CPU上运行
- 就绪态（Ready）：已经具备运行条件，但由于没有空闲CPU，而暂时不能运行
- 等待态（Waiting/Blocked）：因等待某一事件而暂时不能运行 如：等待读盘结果。又称阻塞态、封锁态、睡眠态



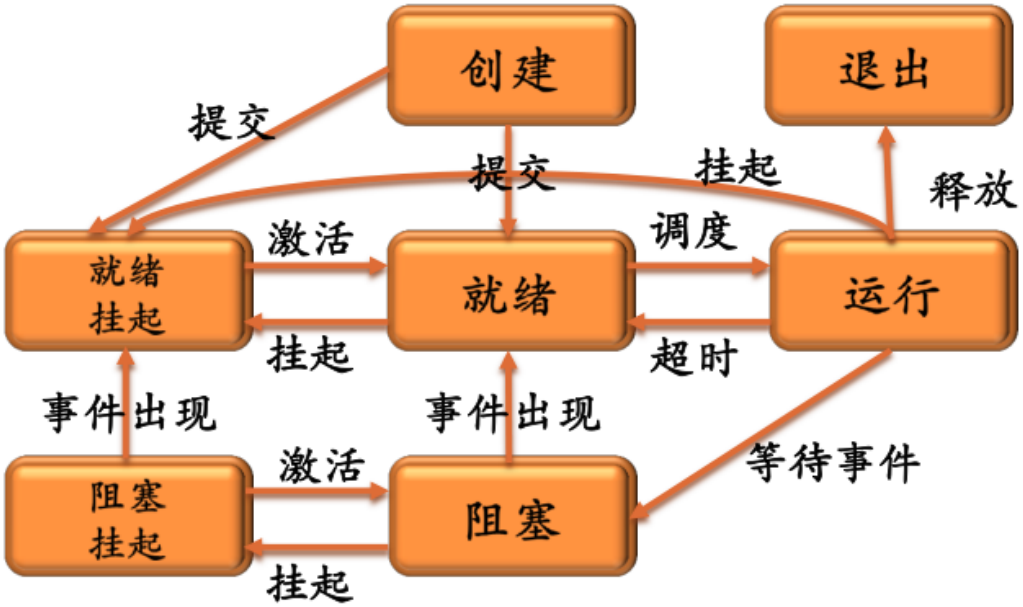
进程的其他状态

- 创建
  - 已完成创建一进程所必要的工作– PID、PCB
    但尚未同意执行该进程– 因为资源有限
- 终止：
  - 终止执行后，进程进入该状态
  - 可完成一些数据统计工作
  - 资源回收
- 挂起：
  - 用于调节负载
  - 进程不占用内存空间，其进程映像交换到磁盘上

五状态进程模型



七状态进程模型



进程队列

为使操作系统按不同状态对进程进行管理，设计了进程队列

- 操作系统为每一类进程建立一个或多个队列
- 队列元素为PCB
- 伴随进程状态的改变，其PCB从一个队列进入另一个队列

等待态

- 多个等待队列等待的事件不同
- 就绪队列也可以多个
- 单CPU情况下，运行队列中只有一个进程

3.3 进程控制

进程控制操作完成进程各状态之间的转换，由具有特定功能的原语完成

原语（primitive）：完成某种特定功能的一段程序，具有不可分割性或不可中断性，即原语的执行必须是连续的，在执行过程中不允许被中断

常见原语：进程创建原语、进程撤消原语、阻塞原语、唤醒原语、挂起原语、激活原语、改变进程优先级 ……

进程的创建

- 给新进程分配一个唯一标识以及进程控制块
- 为进程分配地址空间
- 初始化进程控制块：设置默认值 (如: 状态为 New，...)
- 设置相应的队列指针，如: 把新进程加到就绪队列链表中

  UNIX：fork/exec；WINDOWS：CreateProcess

进程的撤销：结束进程

- 收回进程所占有的资源

    - 关闭打开的文件、断开网络连接、回收分配的内存、……
- 撤消该进程的PCB

  UNIX：exit；WINDOWS：TerminateProcess

进程阻塞

处于运行状态的进程，在其运行过程中期待某一事件发生，如等待键盘输入、等待磁盘数据传输完成、等待其它进程发送消息，当被等待的事件未发生时，由**进程自己执行阻塞原语**，使自己由运行态变为阻塞态

UNIX：wait；WINDOWS：WaitForSingleObject

UNIX的几个进程控制操作

- fork()：创建新的进程，通过**复制调用进程**来建立新的进程，是最基本的进程建立过
- exec()：包括一系列系统调用，它们都是通过用一段新的程序代码覆盖原来的地址空间，实现进程**执行代码的转换**
- wait()：提供初级进程同步操作，能使一个进程等待另外一个进程的结束
- exit()：用来终止一个进程的运行

UNIX的FORK（）实现

- 为子进程分配一个空闲的进程描述符
- 分配给子进程唯一标识 pid
- 以一次一页的方式复制父进程地址空间

  Linux采用了写时复制技术COW（Copy-On-Write）加快创建进程
- 从父进程处继承共享资源，如打开的文件和当前工作目录等
- 将子进程的状态设为就绪，插入到就绪队列
- 对子进程返回标识符 0
- 向父进程返回子进程的 pid

3.4 关于进程相关概念的讨论

- 进程的分类：
    - 系统进程/用户进程
    - 前台进程/后台进程
    - CPU密集型进程/IO密集型进程
- 进程层次结构

  UNIX进程家族树：init为根
  Windows：地位相同

进程与程序的区别

- 进程更能准确刻画并发，而程序不能
- 程序是静态的，进程是动态的
- 进程有生命周期的，有诞生有消亡，是短暂的；而程序是相对长久的
- 一个程序可对应多个进程
- 进程具有创建其他进程的功能

- 进程地址空间：操作系统给每个进程都分配了一个地址空间
- 进程映像 (IMAGE)：对进程执行活动全过程的静态描述
  由进程地址空间内容、硬件寄存器内容及与该进程相关的内核数据结构、内核栈组成
- 用户相关：进程地址空间（包括代码段、数据段、堆和栈、共享库……）
- 寄存器相关：程序计数器、指令寄存器、程序状态、寄存器、栈指针、通用寄存器等的值
- 内核相关：
  - 静态部分：PCB及各种资源数据结构
  - 动态部分：内核栈（不同进程在进入内核后使用不同的内核栈）

上下文（CONTEXT）切换

将CPU硬件状态从一个进程换到另一个进程的过程称为上下文切换

- 进程运行时，其硬件状态保存在CPU上的寄存器中
  寄存器：程序计数器、程序状态寄存器、栈指针、通用寄存器、其他控制寄存器的值
- 进程不运行时，这些寄存器的值保存在进程控制块PCB中；当操作系统要运行一个新的进程时，将PCB中的相关值送到对应的寄存器中

3.5 线程的引入

为什么在进程中再派生线程?

- 应用的需要

  可以同时完成多个需求，实现并发
- 开销的考虑
  - 进程的开销高：进程相关的操作，创建进程、 撤消进程、 进程通信、进程切换 → 时间/空间开销大，限制了并发度的提高
  - 线程的开销小
    - 创建一个新线程花费时间少（撤销亦如此）
    - 两个线程切换花费时间少
    - 线程之间相互通信无须调用内核（同一进程内的线程共享内存和文件）
- 性能的考虑

  多个处理器情况下使用多个线程分别进行计算、I/O等，提高性能

进程的两个基本属性：

- 资源的拥有者：进程还是资源的拥有者
- CPU调度单位：线程继承了这一属性

线程：进程中的一个运行实体，是CPU的调度单位，有时将线程称为轻量级进程；在同一进程增加了多个执行序列（线程）

线程的属性

- 有标示符ID
- 有状态及状态转换 → 需要提供一些操作

- 不运行时需要保存的上下文
  有上下文环境：程序计数器等寄存器
- 有自己的栈和栈指针 √
- **共享所在进程的地址空间和其他资源**
- 可以创建、撤消另一个线程
  程序开始是以一个单线程进程方式运行的

3.6 线程机制的实现

- 用户级线程：UNIX
  - 在用户空间建立线程库：提供一组管理线程的过程
  - 运行时系统：完成线程的管理工作（操作、线程表）
  - 内核管理的还是进程，不知道线程的存在
  - 线程切换不需要内核态特权

  优点：
  - 线程切换快
  - 调度算法是应用程序特定的
  - 用户级线程可运行在任何操作系统上（只需要实现线程库）

  缺点：
  - 内核只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上
  - 大多数系统调用是阻塞的，因此，由于内核阻塞进程，故进程中所有线程也被阻塞
- 核心级线程：Windows
  - 内核管理所有线程管理，并向应用程序提供API接口
  - 内核维护进程和线程的上下文
  - 线程的切换需要内核支持
  - 以线程为基础进行调度
- 混合—两者结合方法：Solaris
  - 线程创建在用户空间完成
  - 线程调度等在核心态完成
    多个用户级线程，多路复用，多个内核级线程

4. 处理器调度

4.1 处理器调度的相关概念

CPU调度

CPU调度其任务是控制、协调进程对CPU的竞争。即按一定的调度算法从就绪队列中选择一个进程，把CPU的使用权交给被选中的进程。如果没有就绪进程，系统会安排一个系统空闲进
程或idle进程

- 系统场景
  - N个进程就绪、等待上CPU运行
  - M个CPU，M ≥ 1
  - 需要决策：给哪一个进程分配哪一个CPU?

CPU调度要解决的三个问题

WHAT（调度算法）：按什么原则选择下一个要执行的进程
WHEN（调度时机）：何时选择
HOW（调度过程，进程的上下文切换）： 如何让被选中的进程上CPU运行

CPU调度的时机

事件发生 → 当前运行的进程暂停运行 → 硬件机制响应后 → 进入操作系统，处理相应的事件 → 结束处理后：某些进程的状态会发生变化，也可能又创建了一些新的进程→ 就绪队列有调整 → 需要进程调度根据预设的调度算法从就绪队列选一个进程

典型的事件举例：

- 创建、唤醒、退出等进程控制操作
- 进程等待I/O、I/O中断
- 时钟中断，如：时间片用完、计时器到时
- 进程执行过程中出现abort异常
- 进程正常终止 或 由于某种错误而终止
- 新进程创建 或 一个等待进程变成就绪
- 当一个进程从运行态进入阻塞态
- 当一个进程从运行态变为就绪态

内核对中断/异常/系统调用处理后返回到用户态时

调度过程——进程切换

进程调度程序从就绪队列选择了要运行的进程：
这个进程可以是刚刚被暂停执行的进程，也可能是另一个新的进程

进程切换：是指一个进程让出处理器，由另一个进程占用处理器的过程

- 进程切换主要包括两部分工作：
    - 切换全局页目录以加载一个新的地址空间
    - 切换内核栈和硬件上下文，其中硬件上下文包括了内核执行新进程需要的全部信息，如CPU相关寄存器

切换过程包括了对原来运行进程各种状态的保存和对新的进程各种状态的恢复

上下文切换具体步骤

场景：进程A下CPU，进程B上CPU

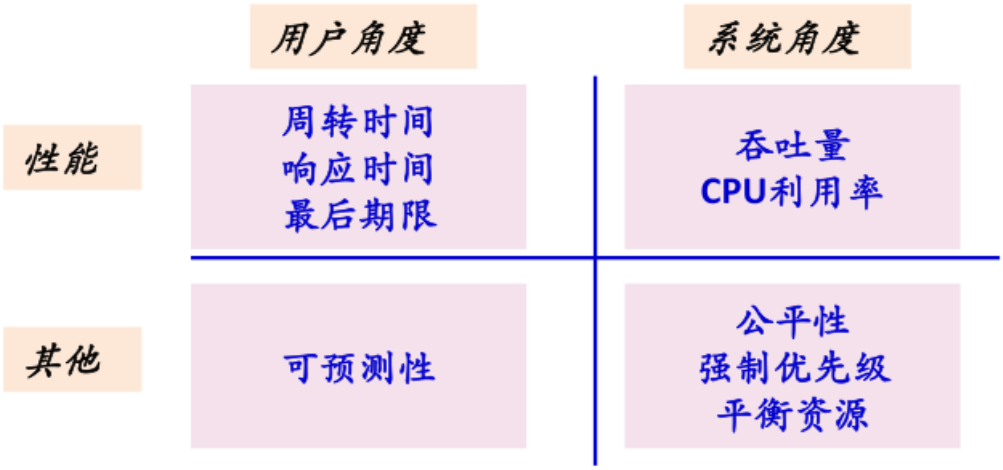
- 保存进程A的上下文环境（程序计数器、程序状态字、其他寄存器……）
- 用新状态和其他相关信息更新进程A的PCB
- 把进程A移至合适的队列（就绪、阻塞……）
- 将进程B的状态设置为运行态
- 从进程B的PCB中恢复上下文（程序计数器 、程序状态字、其他寄存器……）

上下文切换开销（COST)

- 直接开销：内核完成切换所用的CPU时间
    - 保存和恢复寄存器……
    - 切换地址空间（相关指令比较昂贵）
- 间接开销：高速缓存(Cache)、缓冲区缓存(Buffer Cache)和TLB(Translation Look-aside Buffer)失效

CPU调度算法的设计

- 什么情况下需要仔细斟酌调度算法?
  批处理系统 → 多道程序设计系统 → 批处理与分时的混合系统 → 个人计算机 → 网络服务器



调度算法衡量指标

- 吞吐量 Throughput — 每单位时间完成的进程数目
- 周转时间TT(Turnaround Time)：每个进程从提出请求到运行完成的时间
- 响应时间RT(Response Time)：从提出请求到第一次回应的时间
- 其他
  - CPU 利用率(CPU Utilization)：CPU做有效工作的时间比例
  - 等待时间(Waiting time)：每个进程在就绪队列(ready queue)中等待的时间

4.2 涉及调度算法要考虑的几个问题

设计调度算法时要考虑以下几个问题：

进程控制块PCB中需要记录哪些与CPU调度有关的信息

进程优先级及就绪队列的组织

进程优先级(数)

优先级 与 优先数 && 静态 与 动态

静态优先级：进程创建时指定，运行过程中不再改变
动态优先级：进程创建时指定了一个优先级，运行过程中可以动态变化，如：等待时间较长的进程可提升其优先级

进程就队列组织：按优先级排队

抢占式调度与非抢占式调度

指占用CPU的方式：

- 可抢占式Preemptive（可剥夺式）
  当有比正在运行的进程优先级更高的进程就绪时，系统可强行剥夺正在运行进程的CPU，提供给具有更高优先级的进程使用
- 不可抢占式Non-preemptive（不可剥夺式 ）
  某一进程被调度运行后，除非由于它自身的原因不能运行，否则一直运行下去

I/O密集型与CPU密集型进程

按进程执行过程中的行为划分：

- I/O密集型或I/O型(I/O-bound)
  频繁的进行I/O，通常会花费很多时间等待I/O操作的完成
- CPU密集型或CPU型或计算密集型(CPU-bound)
  需要大量的CPU时间进行计算

时间片

时间片长一点 or 短一点 && 固定 与 可变

Time slice 或 quantum

- 一个时间段，分配给调度上CPU的进程，确定了允许该进程运行的时间长度
- 如何选择时间片呢?
  - 进程切换的开销
  - 对响应时间的要求
  - 就绪进程个数
  - CPU能力
  - 进程的行为

4.3 批处理系统的调度算法

先来先服务（FCFS-First Come First Serve）

- First Come First Serve
- 先进先出 First In First Out (FIFO)
- 按照进程就绪的先后顺序使用CPU
- 非抢占
- 优缺点
  - 公平
  - 实现简单
  - 长进程后面的短进程需要等很长时间，不利于用户体验

最短作业优先（SJF-Shortest Job First）

- Shortest Job First
- 具有最短完成时间的进程优先执行
- 非抢占式
- 最短剩余时间优先：Shortest Remaining Time Next(SRTN)
- 进程具有更短的完成时间时，系统抢占当前进程，选择新就绪的进程执行

思路：先完成短的作业，改善短作业的周转时间

- 优缺点
  - 最短的平均周转时间
  - 不公平：源源不断的短任务到来，可能使长的任务长时间得不到运行 → 产生 “饥饿”现象 (starvation)
- 最短剩余时间优先（SRTN-Shortest Remaining Time Next）

- 最高相应比优先（HRRN-Highest Response Ratio Next）折衷权衡
- Highest Response Ratio Next
- 是一个综合的算法
- 调度时，首先计算每个进程的响应比R；之后，总是选择 R 最高的进程执行
- 响应比R = 周转时间 / 处理时间 =（处理时间 + 等待时间）/ 处理时间 = 1 +（等待时间 / 处理时间）
- 吞吐量、周转时间、CPU利用率、公平、平衡

4.4 交互式系统的调度算法

轮转调度（RR-Round Robin）

- 目标：
  - 为短任务改善平均响应时间
- 解决问题的思路
  - 周期性切换
  - 每个进程分配一个时间片
  - 时钟中断 → 轮换
- 如何选择合适的时间片?
  - 太长 --大于典型的交互时间
    - 降级为先来先服务算法
    - 延长短进程的响应时间
  - 太短 --小于典型的交互时间
    - 进程切换浪费CPU时间
- 优缺点
  - 公平
  - 有利于交互式计算，响应时间快
  - 由于进程切换，时间片轮转算法要花费较高的开销
  - 假设时间片 10ms，如果进程切换花费0.1ms，CPU开销约占1%
  - RR对不同大小的进程是有利的
    但是对于相同大小的进程呢?

最高优先级调度（HPF—Highest Priority First)

- 选择优先级最高的进程投入运行
- 通常：系统进程优先级 高于 用户进程
  前台进程优先级 高于 后台进程
  操作系统更偏好 I/O型进程
- 优先级可以是静态不变的，也可以动态调整
  - 优先数可以决定优先级
- 就绪队列可以按照优先级组织
  实现简单；不公平
- 优先级反置(Priority Inversion)，又称翻转、倒挂
  - 现象：一个低优先级进程持有一个高优先级进程所需要的资源，使得高优先级进程等待低优先级进程运行

- eg：设H是高优先级进程，L是低优先级进程， M是中优先级进程（CPU型）
  场景：L进入临界区执行，之后被抢占；H也要进入临界区，失败，被阻塞；M上CPU执行，L无法执行所以H也无法执行
- 影响
  - 系统错误
  - 高优先级进程停滞不前，导致系统性能降低
- 解决方案
  - 设置优先级上限
  - 优先级继承
  - 使用中断禁止

最短进程优先（Shortest Process Next）响应时间：公平、平衡

4.5 多级反馈队列调度算法（Multiple feedback queue)

Multilevel Feedback 是UNIX的一个分支BSD （加州大学伯克利分校开发和发布的）5.3版所采用的调度算法，是一个综合调度算法，折衷权衡

- 设置多个就绪队列，第一级队列优先级最高
- 给不同就绪队列中的进程分配长度不同的时间片，第一级队列时间片最小；随着队列优先级别的降低，时间片增大
- 当第一级队列为空时，在第二级队列调度，以此类推
- 各级队列按照时间片轮转方式 进行调度
- 当一个新创建进程就绪后，进入第一级队列
- 进程用完时间片而放弃CPU，进入下一级就绪队列
- 由于阻塞而放弃CPU的进程进入相应的等待队列，一旦等待的事件发生，该进程回到原来一级就绪队列（？）

再次被调度上CPU时对时间片的处理？：非抢占式

4.6 各种调度算法的比较

调度算法	占用CPU方式	吞吐量	响应时间	开销	对进程的影响	饥饿问题
FCFS	非抢占	不强调	可能很慢，特别是当进程的执行时间差别很大时	最小	对短进程不利；对I/O型的进程不利	无
Round Robin	抢占(时间片用完时)	若时间片小，吞吐量会很低	为短进程提供好的响应时间	最小	公平对待	无
SJF	非抢占	高	为短进程提供好的响应时间	可能较大	对长进程不利	可能
SRTN	抢占(到达时)	高	提供好的响应时间	可能较大	对长进程不利	可能
HRRN	非抢占	高	提供好的响应时间	可能较大	很好的平衡	无
Feedback	抢占(时间片用完时)	不强调	不强调	可能较大	对I/O型进程有利	可能

多处理器调度算法设计

- 不仅要决定选择哪一个进程执行
  - 还需要决定在哪一个CPU上执行
- 要考虑进程在多个CPU 之间迁移时的开销
  - 高速缓存失效、TLB失效
  - 尽可能使进程总是在同一个CPU上执行
    - 如果每个进程可以调度到所有CPU上，假如进程上次在CPU1上执行，本次被调度到CPU2，则会增加高速缓存失效、TLB失效；如果每个进程尽量调度到指定的CPU上，各种失效就会减少
- 考虑负载均衡问题

4.7 Windows的线程调度算法

典型系统所采用的调度算法

- UNIX 动态优先数法
- 5.3BSD 多级反馈队列法
- Linux 抢占式调度
- Windows 基于优先级的抢占式多任务调度
- Solaris 综合调度算法

Windows线程调度

调度单位是线程
采用基于动态优先级的、抢占式调度，结合时间配额的调整

- 就绪线程按优先级进入相应队列
- 系统总是选择优先级最高的就绪线程运行
- 同一优先级的各线程按时间片轮转进行调度
- 多CPU系统中允许多个线程并行运行

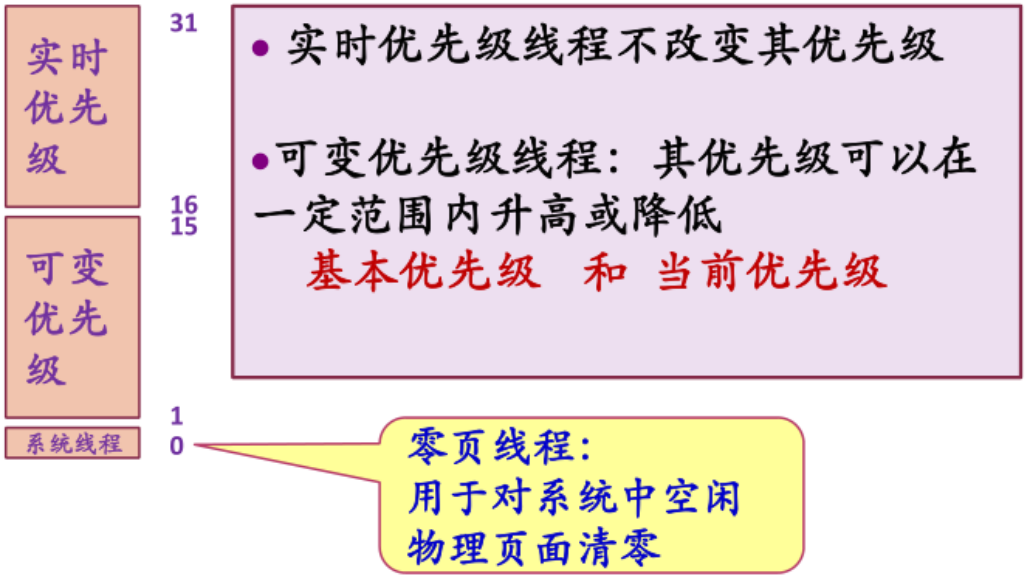
引发线程调度的条件：

- 一个线程的优先级改变了
- 一个线程改变了它的亲和(Affinity)处理机集合

- 线程正常终止 或 由于某种错误而终止
- 新线程创建 或 一个等待线程变成就绪
- 当一个线程从运行态进入阻塞态
- 当一个线程从运行态变为就绪态

线程优先级

Windows使用32个线程优先级，分成三类



- 时间配额不是一个时间长度值，而一个称为配额单位 (quantum unit)的整数
- 一个线程用完了自己的时间配额时，如果没有其他相同优先级的线程，Windows将重新给该线程分配一个新的时间配额，让它继续运行

时间配额的一种特殊作用

- 假设用户首先启动了一个运行时间很长的电子表格计算程序，然后切换到一个游戏程序(需要复杂图形计算并显示，CPU型)
- 如果前台的游戏进程提高它的优先级，则后台的电子表格计算进程就几乎得不到CPU时间了
- 但增加游戏进程的时间配额，则不会停止执行电子表格计算，只是给游戏进程的CPU时间多一些而已

调度策略

- 主动切换
- 抢占
- 时间配额用完

线程优先级提升与时同配额调整

- Windows的调度策略
    - 如何体现对某类线程具有倾向性?
    - 如何解决由于调度策略中潜在的不公平性而带来饥饿现象?
    - 如何改善系统吞吐量、响应时间等整体特征?

- 解决方案
    - 提升线程的优先级
    - 给线程分配一个很大的时间配额
- 针对可变优先级范围内(1至15)的线程优先级，下列5种情况，Windows 会提升线程的当前优先级：
    - I/O操作完成
    - 信号量或事件等待结束
    - 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窗口线程
    - 线程处于就绪态超过了一定的时间还没有运行—— “饥饿”现象

5. 同步互斥机制

5.1 进程的并发执行

并发是所有问题产生的基础
并发是操作系统设计的基础

- 并发
    - 进程的执行是间断性的
    - 进程的相对执行速度不可预测
- 共享：进程/线程之间的制约性
- 不确定性：进程执行的结果与其执行的相对速度有关，是不确定的

竞争条件（ RACE CONDITION)

两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序

5.2 进程互斥

- 由于各进程要求使用共享资源（变量、文件等），而这些资源需要排他性使用，各进程之间竞争使用这些资源，这一关系称为进程互斥
- 临界资源(critical resource)：系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源或共享变量
- 临界区(互斥区，critical section(region))：各个进程中对某个临界资源（共享变量）实施操作的程序片段

临界区（互斥区）的便用原则

- 没有进程在临界区时，想进入临界区的进程可进入
- 不允许两个进程同时处于其临界区中
- 临界区外运行的进程不得阻塞其他进程进入临界区
- 不得使进程无限期等待进入临界区

实现进程互斥的方案

- 软件方案：
    - Dekker解法

    - Peterson解法
- 硬件方案：
    - 屏蔽中断：

      “开关中断”指令
      执行“关中断”指令
      临界区操作
      执行“开中断”指令

        - 优缺点：

          简单，高效
          代价高，限制CPU并发能力（临界区大小）
          不适用于多处理器
          适用于操作系统本身，不适于用户进程
    - TSL(TEST AND SET LOCK)，XCHG(EXCHANGE)指令

小结

- 软件方法
    - 编程技巧
    - 硬件方法
    - 忙等待(busy waiting)
      进程在得到临界区访问权之前，持续测试而不做其他事情
      自旋锁 Spin lock (多处理器 √)
    - 优先级反转（倒置）

5.3 进程互斥的软件解决法

5.4 进程互斥的硬件解决法

5.5 进程同步

进程同步：synchronization

指系统中多个进程中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务：
具体地说，一个进程运行到某一点时，要求另一伙伴进程为它提供消息，在未获得消息之前，该进程进入阻塞态，获得消息后被唤醒进入就绪态

5.6 信号量及PV操作

一个特殊变量

- 用于进程间传递信息的一个整数值
- 定义如下：

```
struc semaphore{
    int count;
    queueType queue;

}
```

- 信号量说明：semaphore s;
- 对信号量可以实施的操作：初始化、P和V（P、V分别是荷兰语的test(proberen)和increment(verhogen)）

PV操作定义

```
P(s){
    s.count --;
    if (s.count < 0){
        //该进程状态置为阻塞状态；
        //将该进程插入相应的等待队列s.queue末尾；
        //重新调度；
    }
}//down, semWait

V(s){
    s.count ++;
    if (s.count < = 0){
        //唤醒相应等待队列s.queue中等待的一个进程；
        //改变其状态为就绪态，并将其插入就绪队列；
    }
}//up, semSignal
```

- P、V操作为原语操作(primitive or atomic action)
- 在信号量上定义了三个操作初始化(非负数)、P操作、V操作
- 最初提出的是二元信号量（解决互斥）之后，推广到一般信号量（多值）或计数信号量（解决同步）

用PV操作解决进程同互斥问题

- 分析并发进程的关键活动，划定临界区
- 设置信号量 mutex，初值为1
- 在临界区前实施 P(mutex)
- 在临界区之后实施 V(mutex)

5.7 生产者消费者问题

又称为有界缓冲区问题问题描述：

- 一个或多个生产者生产某种类型的数据放置在缓冲区中
- 有消费者从缓冲区中取数据，每次取一项
- 只能有一个生产者或消费者对缓冲区进行操作

避免忙等待 睡眠 与 唤醒 操作(原语)

要解决的问题：

- 当缓冲区已满时，生产者不会继续向其中添加数据；
- 当缓冲区为空时，消费者不会从中移走数据

```
void producer(void) {
    int item;
    while (TRUE) {
        item = produce_item();
        P(&empty);
```

```
        P(&mutex);
        insert_item(item);
        V(&mutex)
        V(&full);
    }
}
void consumer(void) {
    int item;
    while (TRUE) {
        P(&full);
        P(&mutex);
        item = remove_item();
        V(&mutex);
        V(&empty);
        consume_item(item);
    }
}
```

5.8 读者写者问题

- 问题描述：
  多个进程共享一个数据区，这些进程分为两组：
  - 读者进程：只读数据区中的数据
  - 写者进程：只往数据区写数据
- 要求满足条件：
  - 允许多个读者同时执行读操作
  - 不允许多个写者同时操作
  - 不允许读者、写者同时操作

第一类读者写者问题：读者优先

- 如果读者执行：
  - 无其他读者、写者，该读者可以读
  - 若已有写者等，但有其他读者正在读，则该读者也可以读
  - 若有写者正在写，该读者必须等
- 如果写者执行：
  - 无其他读者、写者，该写者可以写
  - 若有读者正在读，该写者等待
  - 若有其他写者正在写，该写者等待

```
//第一类解法
void reader(void) {
    while (TRUE) {
        …… P(w);
        读操作
        V(w);
        ……
    }
}
void writer(void) {
    while (TRUE) {
        …… P(w);
        写操作
```

```
            V(w);
            ……
        }
}
```

Linux 提供的读写锁

- 应用场景：如果每个执行实体对临界区的访问或者是读或者是写共享变量，但是它们都不会既读又写时，读写锁是最好的选择
- 实例：Linux的IPX路由代码中使用了读-写锁，用ipx_routes_lock的读-写锁保护IPX路由表的并发访问
- 要通过查找路由表实现包转发的程序需要请求读锁；需要添加和删除路由表中入口的程序必须获取写锁（由于通过读路由表的情况比更新路由表的情况多得多，使用读-写锁提高了性能）

5.9 管程(Monitor)的基本概念

为什么会出现管程

- 问题：信号量机制的不足：程序编写困难、易出错
- 解决： Brinch Hansen(1973)、Hoare（1974）
- 方案： 在程序设计语言中引入管程成分、一种高级同步机制

管程的定义

- 是一个特殊的模块
- 有一个名字
- 由关于共享资源的数据结构及在其上操作的一组过程组成
- 进程与管程：进程只能通过调用管程中的过程来间接地访问管程中的数据结构

管程要保证什么

- 作为一种同步机制，管程要解决两个问题
- 互斥：管程是互斥进入的——为了保证管程中数据结构的数据完整性管程的互斥性是由编译器负责保证的
- 同步：管程中设置条件变量及等待/唤醒操作以解决同步问题；可以让一个进程或线程在条件变量上等待（此时，应先释放管程的使用权），也可以通过发送信号将等待在条件变量上的进程或线程唤醒

应用管程时遇到的问题

设问：是否会出现这样一种场景，有多个进程同时在管程中出现

- 场景：
    - 当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权
    - 当后面进入管程的进程执行唤醒操作时（例如P唤醒Q），管程中便存在两个同时处于活动状态的进程
- 三种处理方法：
    - P等待Q执行
    - Q等待P继续 执行 Hoare √
    - 规定唤醒操作为管程中最后一个可执行的操作 Hansen，并发pascal

5.10 HOARE管程

- 因为管程是互斥进入的，所以当一个进程试图进入一个已被占用的管程时，应当在管程的入口处等待，为此，管程的入口处设置一个进程等待队列，称作入口等待队列
- 如果进程P唤醒进程Q，则P等待Q执行；如果进程Q执行中又唤醒进程R，则Q等待R执行；……，如此，在管程内部可能会出现多个等待进程
    - 在管程内需要设置一个进程等待队列，称为紧急等待队列，紧急等待队列的优先级高于入口等待队列的优先级

HOARE管程——条件变量的实现

- 条件变量——在管程内部说明和使用的一种特殊类型的变量
- var c:condition;
- 对于条件变量，可以执行wait和signal操作
    - wait(c)：如果紧急等待队列非空，则唤醒第一个等待者；否则释放管程的互斥权，执行此操作的进程进入c链末尾
    - signal(c)：如果c链为空，则相当于空操作，执行此操作的进程继续执行；否则唤醒第一个等待者，执行此操作的进程进入紧急等待队列的末尾

5.11 管程的应用

管程实现的两个主要途径：

- 直接构造 → 效率高
- 间接构造→ 用某种已经实现的同步机制去构造
  例如：用信号量及P、
  V操作构造管程

5.12 MESA管程

- Lampson和Redell，Mesa语言（1980）
- Hoare管程的一个缺点：两次额外的进程切换
- 解决：
    - signal → notify
    - notify：当一个正在管程中的进程执行notify(x)时，它使得x条件队列得到通知，发信号的进程继续执行

使用 NOTIFY要注意的问题

- notify的结果：位于条件队列头的进程在将来合适的时候且当处理器可用时恢复执行
- 由于不能保证在它之前没有其他进程进入管程，因而这个进程必须重新检查条件

  → 用while循环取代if语句
- 导致对条件变量至少多一次额外的检测（但不再有额外的进程切换），并且对等待进程在notify之后何时运行没有任何限制

改进 NOTIFY

- 对notify的一个很有用的改进
    - 给每个条件原语关联一个监视计时器，不论是否被通知，一个等待时间超时的进程将被设为就绪态
    - 当该进程被调度执行时，会再次检查相关条件，如果条件满足则继续执行

- 超时可以防止如下情况的发生：当某些进程在产生相关条件的信号之前失败时，等待该条件的进程就会被无限制地推迟执行而处于饥饿状态

引入broadcast

broadcast：使所有在该条件上等待的进程都被释放并进入就绪队列

- 当一个进程不知道有多少进程将被激活时，这种方式是非常方便的

  例子：生产者/消费者问题中，假设insert和remove函数都适用于可变长度的字符块，此时，如果一个生产者往缓冲区中添加了一批字符，它不需要知道每个正在等待的消费者准备消耗多少字符，而仅仅执行一个broadcast，所有正在等待的进程都得到通知并再次尝试运行
- 当一个进程难以准确判定将激活哪个进程时，也可使用广播

HOARE管程与MESA管程的比较

- Mesa管程优于Hoare管程之处在于Mesa管程错误比较少
- 在Mesa管程中，由于每个过程 在收到信号后都重新检查管程变量，并且由于使用了while结构，一个进程不正确的broadcast广播或发信号notify，不会导致收到信号的程序出错收到信号的程序将检查相关的变量，如果期望的条件没有满足，它会重新继续等待

管程小结

- 管程：抽象数据类型
  有一个明确定义的操作集合，通过它且只有通过它才能操纵该数据类型的实例
- 实现管程结构必须保证下面几点：
  1. 只能通过管程的某个过程才能访问资源；
  2. 管程是互斥的，某个时刻只能有一个进程或线程调用管程中的过程
- 条件变量：为提供进程与其他进程通信或同步而引入
  wait/signal 或 wait/notify 或 wait/broadcast

5.13 PTHREAD中的同步机制

pthread_cond_wait的执行分解为三个主要动作：

1. 解锁
2. 等待：当收到一个解除等待的信号（pthread_cond_signal或 pthread_cond_broad_cast）之后，pthread_cond_wait马上需要做的动作是：
3. 上锁

5.14 进程间通信IPC

5.15 典型操作系统中的IPC机制

为什么需要通信机制?

- 信号量及管程的不足
- 不适用多处理器情况
- 进程通信机制
  消息传递：send & receive原语
  - 适用于：分布式系统、基于共享内存的多处理机系统、单处理机系统可以解决进程间的同步问题、通信问题

基本通信方式

- 消息传递
- 共享内存
- 管道
- 套接字
- 远程过程调用

消息传递

1. 陷入内核
2. 复制消息
3. 消息入队
4. 复制消息

一组消息缓冲区：接收进程的PCB中消息队列指针

消息缓冲区结构：

- 消息头（消息类型、接收进程ID、发送进程ID、消息长度、控制信息）
- 消息体（消息内容）

管道通信方式PIPE

- 利用一个缓冲传输介质——内存或文件连接两个相互通信的进程
    - 字符流方式写入读出
    - 先进先出顺序
    - 管道通信机制必须提供的协调能力：互斥、同步、判断对方进程是否存在

典型操作系统中的IPC机制

- UNIX：管道、消息队列、共享内存、信号量、信号、套接字
- Linux：
    - 管道、消息队列、共享内存、信号量、信号、套接字
    - 内核同步机制：原子操作、自旋锁、读写锁、信号量、屏障、BKL
- Windows
    - 同步对象：互斥对象、事件对象、信号量对象
    - 临界区对象：互锁变量
    - 套接字、文件映射、管道、命名管道、邮件槽、剪贴板、动态数据交换、对象连接与嵌入、动态链接库、远程过程调用

原子操作

- 不可分割，在执行完之前不会被其他任务或事件中断
- 常用于实现资源的引用计数

```
typedef struct { volatile int counter; } atomic_t;
//原子操作API包括：
atomic_read(atomic_t * v);
atomic_set(atomic_t * v, int i);
void atomic_add(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
```

屏障（Barrier）

- 一种同步机制(又称栅栏、关卡)
- 用于对一组线程进行协调
- 应用场景：一组线程协同完成一项任务，需要所有线程都到达一个汇合点后再一起向前推进

6. 存储模型

6.1 基本概念-地址重定位(Relocation)

- 已知：程序装载到内存才可以运行通常，程序以可执行文件格式保存在磁盘上
  - 多道程序设计模型允许多个程序同时进入内存
  - 每个进程有自己的地址空间
  - 一个进程执行时不能访问另一个进程的地址空间
  - 进程不能执行不适合的操作
- 进程中的地址不是最终的物理地址
  在进程运行前无法计算出物理地址，因为：不能确定进程被加载到内存什么地方→→ 需要 地址重定位 的支持
  地址转换、地址变换、地址翻译、地址映射，Translation、Mapping

地址重定位

- 逻辑地址（相对地址，虚拟地址）：用户程序经过编译、汇编后形成目标代码，目标代码通常采用相对地址的形式，其首地址为0，其余地址都相对于首地址而编址。**不能用逻辑地址在内存中读取信息**
- 物理地址（绝对地址，实地址）：内存中存储单元的地址 可直接寻址
- 为了保证CPU执行指令时可正确访问内存单元，需要将用户程序中的逻辑地址转换为运行时可由机器直接寻址的物理地址，这一过程称为地址重定位

静态地址重定位与动态重定位

- 静态重定位：
  当用户程序加载到内存时，一次性实现逻辑地址到物理地址的转换，一般可以由软件完成
- 动态重定位：
  在进程执行过程中进行地址变换→ → 即逐条指令执行时完成地址转换，需要硬件部件支持

6.2 物理内存管理

空闲内存管理

数据结构

- 位图：每个分配单元对应于位图中的一位，0表示空闲，1表示占用（或者相反）
- 空闲区表、已分配区表：表中每一项记录了空闲区（或已分配区）的起始地址、长度、标志
- 空闲块链表

内存分配算法

- 首次适配 (first fit)：在空闲区表中找到第一个满足进程要求的空闲区
- 下次适配 (next fit)：从上次找到的空闲区处接着查找
- 最佳适配 (best fit)： 查找整个空闲区表，找到能够满足进程要求的最小空闲区

- 最差适配 (worst fit)：总是分配满足进程要求的最大空闲区

将该空闲区分为两部分，一部分供进程使用，另一部分形成新的空闲区

回收问题

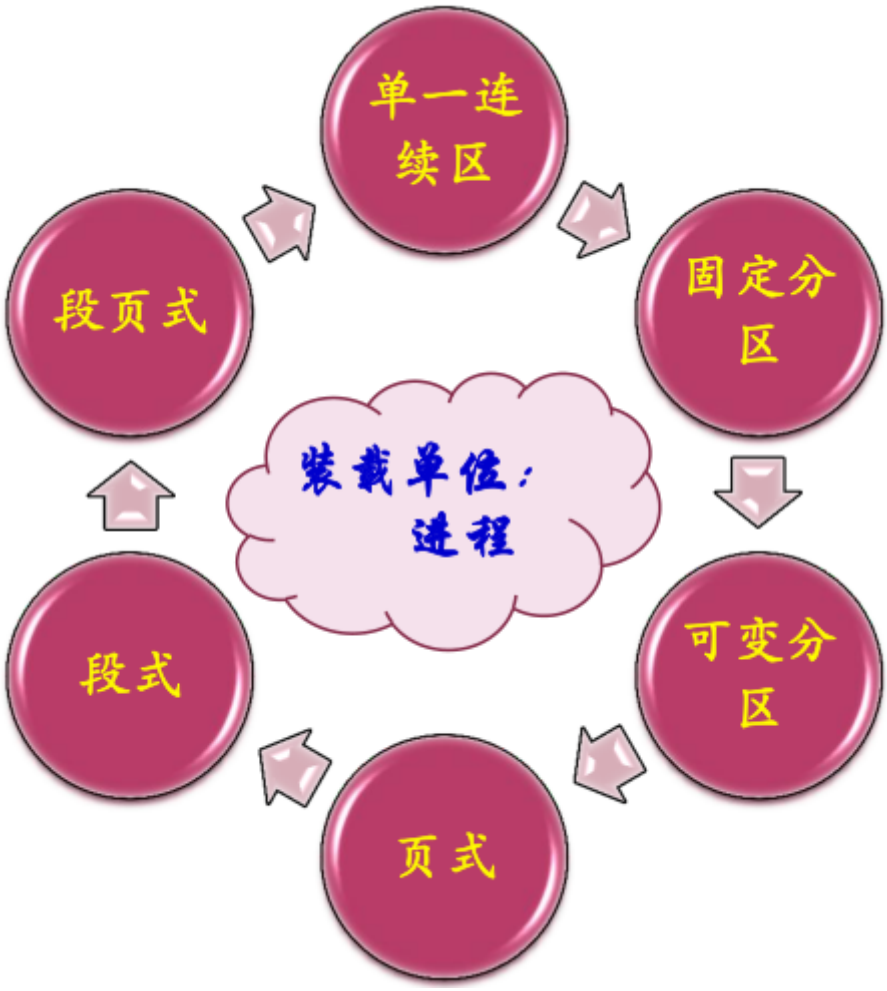
- 内存回收算法
    - 当某一块归还后，前后空闲空间合并，修改内存空闲区表
    - 四种情况：上相邻、下相邻、上下都相邻、上下都不相邻

6.3 伙伴系统

- 一种经典的内存分配方案
- 主要思想：将内存按2的幂进行划分，组成若干空闲块链表；查找该链表找到能满足进程需求的最佳匹配块
- 算法：
    - 首先将整个可用空间看作一块： 2^U
    - 假设进程申请的空间大小为 s，如果满足$2^{U-1} < s <= 2^U$，则分配整个块
      否则，将块划分为两个大小相等的伙伴，大小为2^{U-1}
    - 一直划分下去直到产生大于或等于 s 的最小块

6.4 基本内存管理方案

整个进程进入内存中一片连续区域



单一连续区

特点：一段时间内只有一个进程在内存简单，内存利用率低

固定分区

- 把内存空间分割成若干区域，称为分区
- 每个分区的大小可以相同也可以不同
- 分区大小固定不变
- 每个分区装一个且只能装一个进程

可变分区

- 根据进程的需要，把内存空闲空间分割出一个分区，分配给该进程
- 剩余部分成为新的空闲区

碎片问题解决

 碎片 → 很小的、不易利用的空闲区
→ 导致内存利用率下降
 解决方案 → 紧缩技术（memory compaction）
在内存移动程序，将所有小的空闲区合并为较大
的空闲区
又称：压缩技术，紧致技术，搬家技术
 紧缩时要考虑的问题
系统开销？ 移动时机

一个进程进入内存中若干不连续的区域

页式存储管理方案

设计思想

 用户进程地址空间被划分为大小相等的部分，称为页（page）或页面，从0开始编号
 内存空间按同样大小划分为大小相等的区域，称为页框（page frame），从0开始编号；也称为物理页面，页帧，内存块
 内存分配（规则）
以页为单位进行分配，并按进程需要的页数来分配；逻辑上相邻的页，物理上不一定相邻

 典型页面尺寸：4K 或 4M

划分是由系统自动完成的，对用户是透明的

相关数据结构及地址转换

 页表
 页表项：记录了逻辑页号与页框号的对应关系
 每个进程一个页表，存放在内存
 页表起始地址保存在何处?
 空闲内存管理
 地址转换（硬件支持）
CPU取到逻辑地址，自动划分为页号和页内地址；
用页号查页表，得到页框号，再与页内偏移拼接成
为物理地址

段式存储管理方案

设计思想
 用户进程地址空间：按程序自身的逻辑关系划分为若干个程序段，每个程序段都有一个段名
 内存空间被动态划分为若干长度不相同的区域，称为物理段，每个物理段由起始地址和长度确定
 内存分配（规则）：以段为单位进行分配，每段在内存中占据连续空间，但各段之间可以不相邻

逻辑地址： 段号 段内地址

相关数据结构及地址转换

 段表
 每项记录了段号、段首地址和段长度之间的关系
 每个进程一个段表，存放在内存
 段表起始地址保存在何处?
 物理内存管理
 地址转换（硬件）
CPU取到逻辑地址，用段号查段表，得到该段在内存的起始地址，与段内偏移地址计算出物理地址

段页式存储管理方案

产生背景
综合页式、段式方案的优点，克服二者的缺点
 设计思想
用户进程划分：
先按段划分，每一段再按页面划分
逻辑地址：
内存划分：同页式存储管理方案
内存分配：以页为单位进行分配

 数据结构及有关操作
 段表：记录了每一段的页表始址和页表长度
 页表：记录了逻辑页号与页框号的对应关系
每一段有一张页表，一个进程有多个页表
 空闲区管理：同页式管理
 内存分配、回收：同页式管理

小结

单一连续区	每次只运行一个用户程序，用户程序独占内存，它总是被加载到同一个内存地址上
固定分区	把可分配的内存空间分割成若干个连续区域，每一区域称为分区。每个分区的大小可以相同也可以不同，分区大小固定不变，每个分区装一个且只能装一个进程
可变分区	根据进程的需求，把可分配的内存空间分割出一个分区，分配给该进程
页式	把用户程序地址空间划分成大小相等的部分，称为页。内存空间按页的大小划分为大小相等的区域，称为内存块（物理页面，页框，页帧）。以页为单位进行分配，逻辑上相邻的页，物理上不一定相邻
段式	用户程序地址空间按进程自身的逻辑关系划分为若干段，内存空间被动态的划分为若干个长度不相同的区域（可变分区）。以段为单位分配内存，每一段在内存中占据连续空间，各段之间可以不连续存放
段页式	用户程序地址空间：段式；内存空间：页式；分配单位：页

6.5 内存不足时如何管理

内存紧缩技术（例如：可变分区）

覆盖技术 (overlaying): 主要用于早期的操作系统

解决的问题 → 程序大小超过物理内存总和

- 程序执行过程中，程序的不同部分在内存中相互替代
  → 按照其自身的逻辑结构，将那些不会同时执行的程序段共享同一块内存区域
  → 要求程序各模块之间有明确的调用结构
- 程序员声明覆盖结构，操作系统完成自动覆盖
- 缺点：对用户不透明，增加了用户负担

交换技术 swapping

- 设计思想
  内存空间紧张时，系统将内存中某些进程暂时移到外存，把外存中某些进程换进内存，占据前者所占用的区域（进程在内存与磁盘之间的动态调度）
  - 进程的哪些内容要交换到磁盘？会遇到什么困难？

    运行时创建或修改的内容：栈和堆
  - 在磁盘的什么位置保存被换出的进程？

    交换区：一般系统会指定一块特殊的磁盘区域作为交换空间（swap space），包含连续的磁道，操作系统可以使用底层的磁盘读写操作对其高效访问
  - 交换时机？

    只要不用就换出（很少再用）；内存空间不够或有不够的危险时换出，与调度器结合使用
  - 如何选择被换出的进程？

    考虑进程的各种属性；不应换出处于等待I/O状态的进程

- 如何处理进程空间增长?

虚拟存储技术 （virtual memory)

当进程运行时，先将其一部分装入内存，另一部分暂留在磁盘，当要执行的指令或访问的数据不在内存时，由操作系统自动完成将它们从磁盘调入内存的工作

- 虚拟地址空间 即为 分配给进程的虚拟内存
- 虚拟地址 是 在虚拟内存中指令或数据的位置，该位置可以被访问，仿佛它是内存的一部分

虚存与存储体系

- 把内存与磁盘有机地结合起来使用，从而得到一个容量很大的“内存”，即虚存
- 虚存是对内存的抽象，构建在存储体系之上，由操作系统协调各存储器的使用
- 虚存提供了一个比物理内存空间大得多的地址空间

存储保护

- 确保每个进程有独立的地址空间
- 确保进程访问合法的地址范围
- 确保进程的操作是合法的

防止地址越界，防止访问越权

地址越界 → 陷入操作系统

虚拟页式

虚拟存储技术 + 页式存储管理方案→ 虚拟页式存储管理系统

具体有两种方式

1. 请求调页（demandpaging）√
2. 预先调页（prepaging）

- 基本思想：以CPU时间和磁盘空间换取昂贵内存空间，这是操作系统中的资源转换技术
    - 进程开始运行之前，不是装入全部页面，而是装入一个或零个页面
    - 之后，根据进程运行的需要，动态装入其他页面
    - 当内存空间已满，而又需要装入新的页面时，则根据某种算法置换内存中的某个页面，以便装入新的页面

6.6 页表及页表项的设计

页表表项设计：通常由硬件设计的

- 页表由页表项组成
- 页框号、有效位、访问位、修改位、保护位
    - 页框号（内存块号、物理页面号、页帧号）
    - 有效位（驻留位、中断位）：表示该页是在内存还是在磁盘
    - 访问位：引用位
    - 修改位：此页在内存中是否被修改过
    - 保护位：读/可读写

关于页表

- 32位虚拟地址空间的页表规模?
  页面大小为4K；页表项大小为4字节，则：一个进程地址空间有 2^{20} 页
  其页表需要占 1024页（页表页）
- 64位虚拟地址空间
  页面大小为4K；页表项大小为8字节
  页表规模： 32,000 TB
- 页表页在内存中若不连续存放，则需要引入页表页的地址索引表 → 页目录（Page Directory）

引入反转(倒排)页表

- 地址转换
  从虚拟地址空间出发：虚拟地址 → 查页表 → 得到页框号 → 形成物理地址
  每个进程一张页表
- 解决思路
  - 从物理地址空间出发，系统建立一张页表
  - 页表项记录进程i的某虚拟地址(虚页号)与页框号的映射关系

反转(倒排)页表设计

- PowerPC、UltraSPARC和IA-64 等体系结构采用
- 将虚拟地址的页号部分映射到一个散列值
- 散列值指向一个反转页表
- 反转页表大小与实际内存成固定比例，与进程个数无关

6.7 地址转换过程及TLB的引入

TLB的引入

- 问题
  - 页表 → 两次或两次以上的内存访问
  - CPU的指令处理速度与内存指令的访问速度差异大，CPU的速度得不到充分利用
  - 如何加快地址映射速度，以改善系统性能?
- 解决
- 程序访问的局部性原理 → 引入快表(TLB)

快表是什么

- TLB — Translation Look-aside Buffers

  在CPU中引入的高速缓存（Cache），可以匹配CPU的处理速率和内存的访问速度
  一种随机存取型存储器，除连线寻址机制外，还有接线逻辑，能按特定的匹配标志在一个存储周期内对所有的字同时进行比较
- 相联存储器（associative memory）
  特点：按内容并行查找
- 保存正在运行进程的页表的子集(部分页表项)

6.8 页错误(Page Fault)

页错误页面错误、页故障、页面失效

- 地址转换过程中硬件产生的异常
  - 所访问的虚拟页面没有调入物理内存 → 缺页异常

- 页面访问违反权限（读/写、用户/内核）
- 错误的访问地址

缺页异常处理

- 是一种Page Fault
- 在地址映射过程中，硬件检查页表时发现所要访问的页面不在内存，则产生该异常——缺页异常
- 操作系统执行缺页异常处理程序：获得磁盘地址，启动磁盘，将该页调入内存
  - 如果内存中有空闲页框，则分配一个页框，将新调入页装入，并修改页表中相应页表项的有效位及相应的页框号
  - 若内存中没有空闲页框，则要置换内存中某一页框；若该页框内容被修改过，则要将其写回磁盘

6.9 软件相关策略

驻留集大小

给每个进程分配多少页框?

- 固定分配策略
  进程创建时确定
  可以根据进程类型（交互、批处理、应用类）或者基于程序员或系统管理员的需要来确定
- 可变分配策略
  根据缺页率评估进程局部性表现
  缺页率高→增加页框数
  缺页率低→减少页框数
  系统开销

置换问题

- 置换范围
  计划置换页面的集合是局限在产生缺页中断的进程，还是所有进程的页框?
- 置换策略
  在计划置换的页框集合中，选择换出哪一个页框?
  - 局部置换策略：仅在产生本次缺页的进程的驻留集中选择
  - 全局置换策略： 将内存中所有未锁定的页框都作为置换的候选

1. 当一个新进程装入内存时，给它分配一定数目的页框，然后填满这些页框
2. 当发生一次缺页异常时，从产生缺页异常进程的驻留集中选择一页用于置换
3. 不断重新评估进程的页框分配情况，增加或减少分配给它的页框，以提高整体性能

置换策略

- 决定置换当前内存中的哪一个页框
- 所有策略的目标→ 置换最近最不可能访问的页
- 根据局部性原理，最近的访问历史和最近将要访问的模式间存在相关性，因此，大多数策略都基于过去的行为来预测将来的行为
- 注意：置换策略设计得越精致、越复杂，实现的软硬件开销就越大
- 约束：不能置换被锁定的页框

页框锁定

为什么要锁定页面?

- 采用虚存技术后：开销 → 使进程运行时间变得不确定
- 给每一页框增加一个锁定位
- 通过设置相应的锁定位，不让操作系统将进程使用的页面换出内存，避免产生由交换过程带来的不确定的延迟
- 例如：操作系统核心代码、关键数据结构、I/O缓冲区，特别是正在I/O的内存页面

Windows中的VirtualLock和VirtualUnLock函数

清除策略

- 清除：从进程的驻留集中收回页框
- 虚拟页式系统工作的最佳状态：发生缺页异常时，系统中有大量的空闲页框
- 结论：在系统中保存一定数目的空闲页框供给比使用所有内存并在需要时搜索一个页框有更好的性能

  > 设计一个分页守护进程（paging daemon），多数时间睡眠着，可定期唤醒以检查内存的状态
  >
  > 如果空闲页框过少，分页守护进程通过预定的页面置换算法选择页面换出内存
  >
  > 如果页面装入内存后被修改过，则将它们写回磁盘分页守护进程可保证所有的空闲页框是“干净”的

- 当进程需要使用一个已置换出的页框时，如果该页框还没有被新的内容覆盖，将它从空闲页框集合中移出即可恢复该页面

  > 页缓冲技术：
  >
  > - 不丢弃置换出的页，将它们放入两个表之一：如果未被修改，则放到空闲页链表中，如果修改了，则放到修改页链表中
  > - 被修改的页定期写回磁盘(不是一次只写一个，大大减少I/O操作的数量，从而减少了磁盘访问时间)
  > - 被置换的页仍然保留在内存中，一旦进程又要访问该页，可以迅速将它加入该进程的驻留集合(代价很小)

6.10 页面置换算法

又称页面淘汰（替换）算法
最佳算法→先进先出→第二次机会→时钟算法→最近未使用→最近最少使用→最不经常使用→老化算法→工作集→工作集时钟

最佳页面置换算法(OPT)

- 设计思想：置换以后不再需要的或最远的将来才会用到的页面
- 作用：作为一种标准来衡量其他算法的性能

先进先出算法(FIFO)

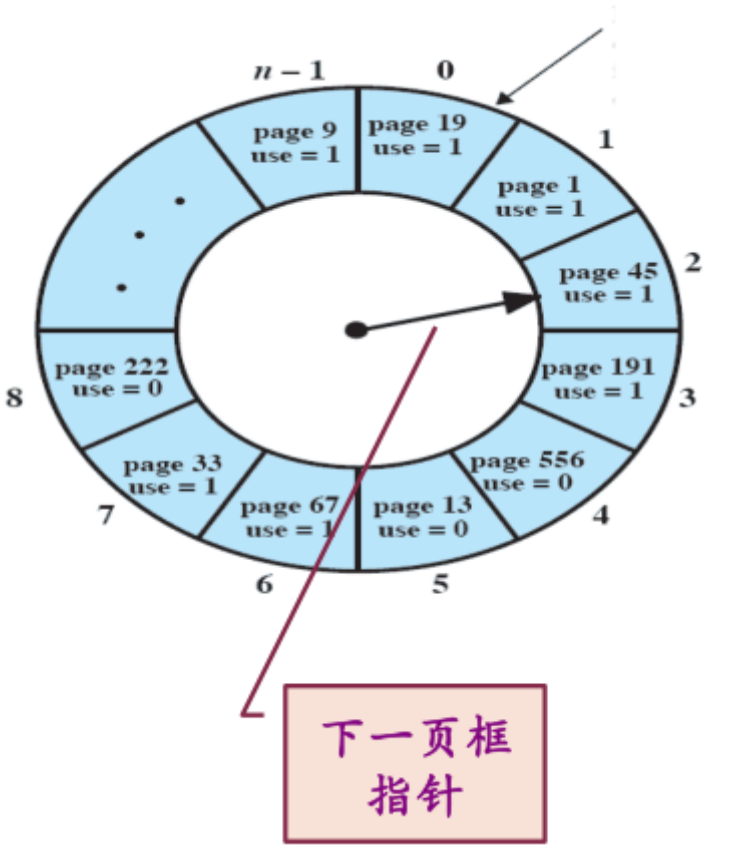
- 选择在内存中驻留时间最长的页并置换它
  对照：超市撤换商品
- 实现：页面链表法

第二次机会算法(SCR)

- SCR-Second Chance：按照先进先出算法选择某一页面，检查其访问位R，如果为0，则置换该页；如果为1，则给第二次机会，并将访问位置0

时钟算法(CLOCK)

1. 从指针的当前位置开始，扫描页框缓冲区，选择遇到的第一个页框（r=0；m=0）用于置换(本扫描过程中，对使用位不做任何修改)
2. 如果第1步失败，则重新扫描，选择第一个（r=0；m=1）的页框(本次扫描过程中，对每个跳过的页框，将其使用位设置成0)
3. 如果第2步失败，指针将回到它的最初位置，并且集合中所有页框的使用位均为0。重复第1步，并且，如果有必要，重复第2步。这样将可以找到供置换的页框



最近未便用算法(NRU，Not Recently Used)

- 选择在最近一段时间内未使用过的一页并置换
- 实现：设置页表表项的两位访问位（R），修改位（M）

  （如果硬件没有这些位，则可用软件模拟（做标记）
- 启动一个进程时，R、M位置0，R位被定期清零（复位）
- 发生缺页中断时，操作系统检查R，M：
  第1类：无访问，无修改
  第2类：无访问，有修改
  第3类：有访问，无修改
  第4类：有访问，有修改
- 算法思想：
  随机从编号最小的非空类中选择一页置换

最近最少使用算法（LRU，Least Recently Used

- 选择最后一次访问时间距离当前时间最长的一页并置换，即置换**未使用时间最长**的一页
- 性能接近OPT
- 实现：时间戳 或 维护一个访问页的栈→ 开销大

最不经常使用算法（NFU，Not Frequently Used）

- 选择访问次数最少的页面置换
- LRU的一种软件解决方案
- 实现：
  - 软件计数器，一页一个，初值为0
  - 每次时钟中断时，计数器加R
  - 发生缺页中断时，选择计数器值最小的一页置换
  - 改进（模拟LRU）：计数器在加R前先右移一位，R位加到计数器的最左端
- BELADY 现象

  例子：系统给某进程分配 m个页框，初始为空页面访问顺序为
  1 2 3 4 1 2 5 1 2 3 4 5
  采用FIFO算法，计算当 m=3 和 m=4 时的缺页中断次数
  m=3时，缺页中断9次；m=4时，缺页中断10次
  注：FIFO页面置换算法会产生异常现象（Belady现象），即：当分配给进程的物理页面数增加时，缺页次数反而增加

###

工作集算法

影响缺页次数的因数

- 页面置换算法
- 页面本身的大小 √
- 程序的编制方法 √
- 分配给进程的页框数量 √

> 颠簸（Thrashing，抖动）
> 虚存中，页面在内存与磁盘之间频繁调度，使得调度页面所需的时间比进程实际运行的时间还多，这样导致系统效率急剧下降，这种现象称为颠簸或抖动

页面尺寸问题

- 确定页面大小对于分页的硬件设计非常重要而对于操作系统是个可选的参数
- 要考虑的因素：
  - 内部碎片
  - 页表长度
  - 辅存的物理特性
- Intel 80x86/Pentium：4096 或 4M
- 多种页面尺寸：为有效使用TLB带来灵活性，但给操作系统带来复杂性
- 最优页面大小：$P = \sqrt{2se}$

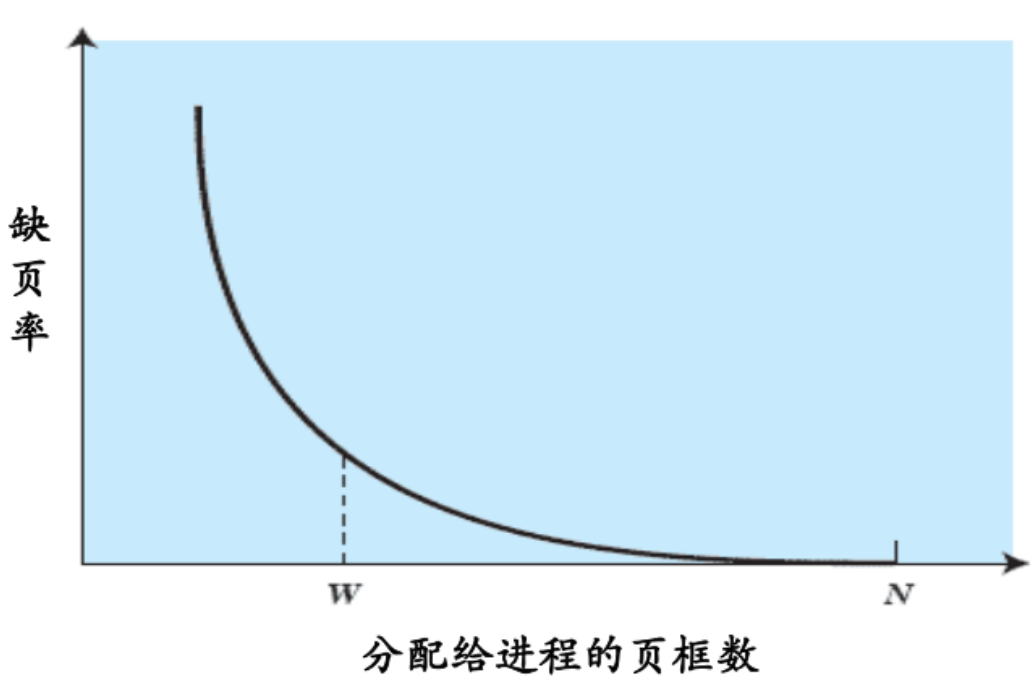
程序编制方法对缺页次数的影响

例子：分配了一个页框；页面大小为128个整数；矩阵A128X128按行存放

```
//程序编制方法1：
for j:=1 to 128
    for i:=1 to 128
        A[i,j]:=0;

//程序编制方法2：
for i:=1 to 128
    for j:=1 to 128
        A[i,j]:=0;
```

分配给进程的页框数与缺页率的关系



工作集 (Working Set) 模型

- 基本思想：
  根据程序的局部性原理，一般情况下，进程在一段时间内总是集中访问一些页面，这些页面称为活跃页面，如果分配给一个进程的物理页面数太少了，使该进程所需的活跃页面不能全部装入内存，则进程在运行过程中将频繁发生中断
  如果能为进程提供与活跃页面数相等的物理页面数，则可减少缺页中断次数
- 工作集：一个进程当前正在使用的页框集合
  工作集W（t，Δ）= 该进程在过去的Δ个虚拟时间单位中访问到的页面的集合
- 内容取决于三个因素：
  - 访页序列特性
  - 时刻t
  - 工作集窗口长度(Δ)：窗口越大，工作集就越大
- 基本思路：找出一个不在工作集中的页面并置换它
- 思路：
  - 每个页表项中有一个字段：记录该页面最后一次被访问的时间
  - 设置一个时间值T
  - 判断：根据一个页面的访问时间是否落在“当前时间-T”之前或之中决定其在工作集之外还是之内

- 实现：
  扫描所有页表项，执行操作
  1. 如果一个页面的R位是1，则将该页面的最后一次访问时间设为当前时间，将R位清零
  2. 如果一个页面的R位是0，则检查该页面的访问时间是否在“当前时间-T”之前
     - 如果是，则该页面为被置换的页面；
     - 如果不是，记录当前所有被扫描过页面的最后访问时间里面的最小值。扫描下一个页面并重复1、2

小结

算法	评价
OPT	不可实现，但可作为基准
NRU	LRU的很粗略的近似
FIFO	可能置换出重要的页面
Second Chance	比FIFO有很大的改善
Clock	实现的
LRU	很优秀，但很难实现
NFU	LRU的相对粗略的近似
Aging	非常近似LRU的有效算法
Working set	实现起来开销很大

6.11 其他相关技术

内存映射文件

- 基本思想
  进程通过一个系统调用(mmap)将一个文件(或部分)映射到其虚拟地址空间的一部分，访问这个文件就象访问内存中的一个大数组，而不是对文件进行读写
- 在多数实现中，在映射共享的页面时不会实际读入页面的内容，而是在访问页面时，页面才会被每次一页的读入，磁盘文件则被当作后备存储
- 当进程退出或显式地解除文件映射时，所有被修改页面会写回文件

写时复制技术

- 例如：两个进程共享三个页，每页都标志成写时复制
  新复制的页面对执行写操作的进程是私有的，对其他共享写时复制页面的进程是不可见的

7.文件系统

7.1 文件与文件系统

文件是什么

- 文件 是 对磁盘的 抽象
- 所谓文件 是指 一组带标识（标识即为文件名）的、在逻辑上有完整意义的信息项的序列
- 信息项：构成文件内容的基本单位（单个字节，或多个字节），各信息项之间具有顺序关系
- 文件内容的意义：由文件建立者和使用者解释

如何设计一个文件系统

- 操作系统角度：怎样组织、管理文件?
  - 文件的描述、分类
  - 文件目录的实现
  - 存储空间的管理
  - 文件的物理地址
  - 磁盘实际运作方式(与设备管理的接口)
  - 文件系统性能等等
- 用户角度：文件系统如何呈现在用户面前：
  - 一个文件的组织
  - 如何命名?
  - 如何保护文件?
  - 可以实施的操作？等等

文件系统

操作系统中统一管理信息资源的一种软件，管理文件的存储、检索、更新，提供安全可靠的共享和保护手段，并且方便用户使用

- 统一管理磁盘空间，实施磁盘空间的分配与回收
- 实现文件的按名存取：名字空间 映射 磁盘空间
- 实现文件信息的共享，并提供文件的保护、保密手段
- 向用户提供一个方便使用、易于维护的接口，并向用户提供有关统计信息
- 提高文件系统的性能
- 提供与I/O系统的统一接口

按文件性质和用途分类（UNIX）：

- 普通文件(regular)

  包含了用户的信息，一般为ASCII或二进制文件

- 目录文件(directory)

  管理文件系统的系统文件

- 特殊文件(special file，设备文件)

  字符设备文件：和输入输出有关，用于模仿串行I/O设备，例如终端，打印机，网卡等
  块设备文件：磁盘

- 管道文件；套接字

文件的逻辑结构

从用户角度看文件，由用户的访问方式确定

还可以组织成堆、顺序、索引、索 引顺序、散列等结构

典型的文件结构与文件存取

- 流式文件：构成文件的基本单位是字符
  文件是有逻辑意义、无结构的一串字符的集合
- 记录式文件：文件由若干个记录组成，可以按记录进行读、写、查找等操作
  每条记录有其内部结构

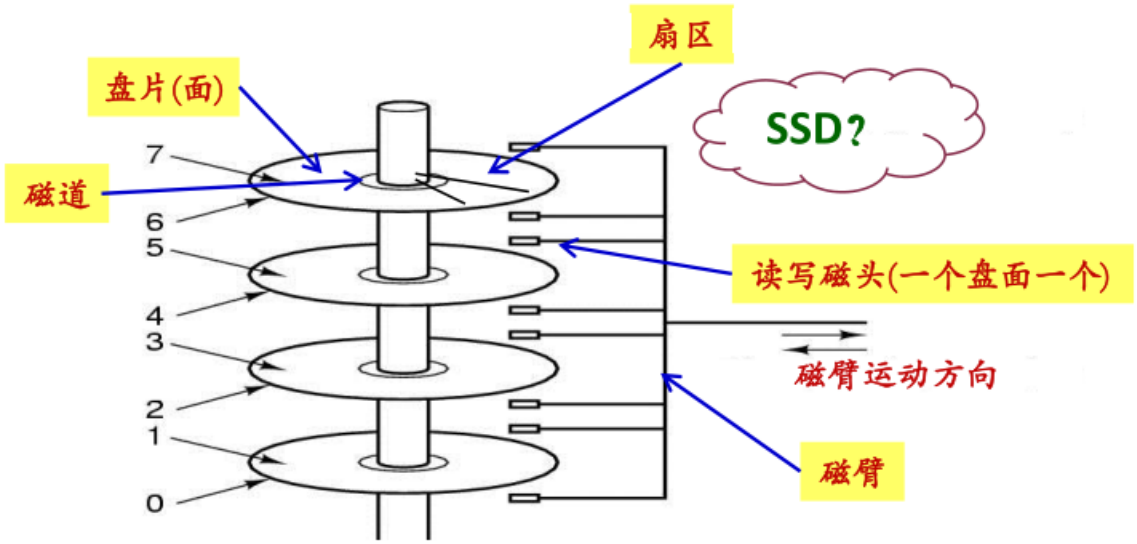
> 顺序存取(访问)
> 随机存取(访问)
> 提供读写位置(当前位置)
> 例如：UNIX的seek操作

7.2 文件的存储介质

存储介质与物理块

- 典型的存储介质
  磁盘(包括固态盘SSD)、磁带、光盘、U盘、……
- 物理块（块block、簇cluster）
  - 信息存储、传输、分配的独立单位
  - 存储设备划分为大小相等的物理块，统一编号

典型的磁盘结构



> 任何时刻只有一个磁头处于活动状态：输入输出数据流以位串形式出现
> 物理地址形式： 磁头号（盘面号）、磁道号（柱面号）、扇区号
> 扇区：标题(10字节)、数据(512字节)、ECC纠错信息(12-16字节)

磁盘访问

一次访盘请求：
读/写，磁盘地址（设备号，柱面号，磁头号，扇区号），内存地址（源/目）
完成过程由三个动作组成：

- 寻道（时间）：磁头移动定位到指定磁道
- 旋转延迟（时间）：等待指定扇区从磁头下旋转经过
- 数据传输（时间）：数据在磁盘与内存之间的实际传输

7.3 磁盘空间管理

有关数据结构

- 位图法
  - 用一串二进制位反映磁盘空间中分配使用情况，每个物理块对应一位，分配物理块为0，否则为1
  - 申请物理块时，可以在位示图中查找为1的位，返回对应物理块号
  - 归还时，将对应位转置1
- 空闲块表
  - 将所有空闲块记录在一个表中，即空闲块表
  - 主要两项内容：起始块号，块数
- 空闲块链表
  - 把所有空闲块链成一个链
  - 扩展：成组链接法

地址与块号的转换

> 已知块号，则磁盘地址：
> 柱面号 = [块号/（磁头数×扇区数）]
> 磁头号 = [（块号mod （磁头数×扇区数））/扇区数]
> 扇区号 = （块号mod （磁头数×扇区数））mod 扇区数
> 已知磁盘地址：
> 块号 = 柱面号×（磁头数×扇区数） + 磁头号×扇区数 + 扇区号

位图计算公式：
已知字号i、位号j： 块号 = i×字长 + j
已知块号：字号 = [块号/字长] 位号 = 块号 mod 字长

成组链接法—分配算法

分配一个空闲块，查L单元（空闲块数）：

- 当空闲块数 > 1 i = L + 空闲块数；
  从 i 单元得到一个空闲块号；
  把该块分配给申请者；
  空闲块数减1；
- 当空闲块数 = 1 取出L + 1单元内容（一组的第一块块号或0）；
  其值 = 0 无空闲块，申请者等待其值不等于零，把该块内容复制到专用块；
  该块分配给申请者；
  把专用块内容读到内存L开始的区域

成组链接法—回收算法

归还一块：查L单元的空闲块数；

- 当空闲块数<100 空闲块数加1；
  j: = L + 空闲块数；
  归还块号填入j单元。
- 当空闲块数 = 100，则把内存中登记的信息写入归还块中；
  把归还块号填入L + 1单元；
  将L单元置成1。

7.4 文件控制块及文件目录

文件属性

- 文件控制块（File Control Block）
  为管理文件而设置的数据结构，保存管理文件所
  需的所有有关信息（文件属性或元数据）
- 常用属性
  文件名，文件号，文件大小，文件地址，创建时间，最后修改时间，最后访问时间，保护，口令，创建者，当前拥有者，文件类型，共享计数，各种标志(只读、隐藏、系统、归档、ASCII/二进制、顺序/随机访问、临时文件、锁)

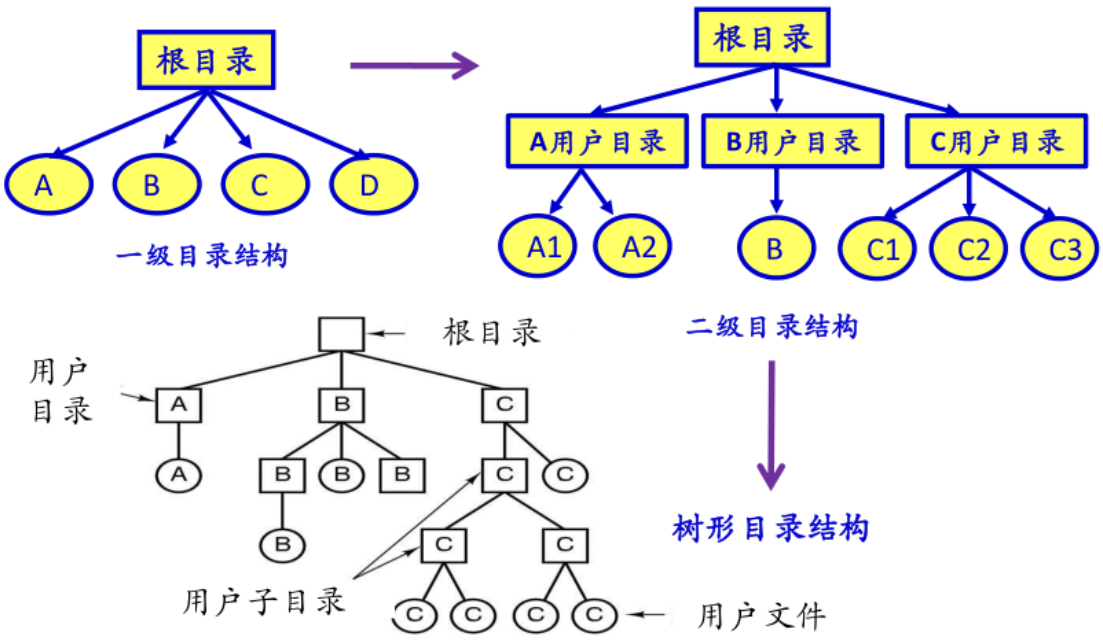
基本文件操作

Create、Delete、Open、Close、Read、Write、Append、Seek、Get Attributes、Set Attributes、Rename... ...

文件目录、目录项与目录文件

- 文件目录
  - 统一管理每个文件的元数据，以支持文件名到文件物理地址的转换
  - 将所有文件的管理信息组织在一起，即构成文件目录
- 目录文件：将文件目录以文件的形式存放在磁盘上
- 目录项
  - 构成文件目录的基本单元
  - 目录项可以是FCB，目录是文件控制块的有序集合

文件目录结构的演化



与目录相关的概念

- 路径名（文件名）
  – 绝对路径名：从根目录开始
  – 相对路径名：从当前目录开始
- 当前目录/工作目录

- 目录操作
  - 创建目录、删除目录
  - 读目录、写目录、改名、复制

7.5 文件的物理结构

- 文件在存储介质上的存放方式
- 主要解决两个问题：
  - 假设一个文件被划分成N块，这N块在磁盘上是怎么存放的？
  - 其地址(块号或簇号)在FCB中是怎样记录的？

连续（顺序）结构

文件的信息存放在若干连续的物理块中

- 优点
  - 简单
  - 支持顺序存取和随机存取
  - 所需的磁盘寻道次数和寻道时间最少
  - 可以同时读入多个块，检索一个块也很容易
- 缺点
  - 文件不能动态增长：预留空间：浪费 或 重新分配和移动
  - 不利于文件插入和删除
  - 外部碎片：紧缩技术

链式结构

一个文件的信息存放在若干不连续的物理块中，各块之间通过指针连接，前一个物理块指向下一个物理块

链接结构的一 个变形: 文件分配表 FAT

某文件的起始块号从何处得到？

- 优点
  - 提高了磁盘空间利用率，不存在外部碎片问题
  - 有利于文件插入和删除
  - 有利于文件动态扩充
- 缺点
  - 存取速度慢，不适于随机存取
  - 可靠性问题，如指针出错
  - 更多的寻道次数和寻道时间
  - 链接指针占用一定的空间

索引结构

- 一个文件的信息存放在若干不连续物理块中

  系统为每个文件建立一个专用数据结构—索引表，并将这些物理块的块号存放在该索引表中

  索引表就是磁盘块地址数组，其中第i个条目指向文件的第i块

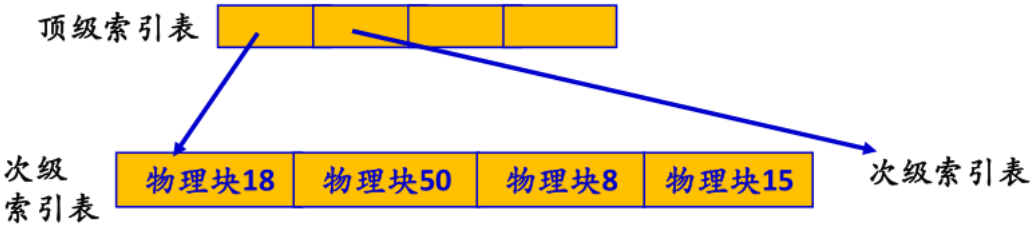
- 优点
  保持了链接结构的优点，又解决了其缺点

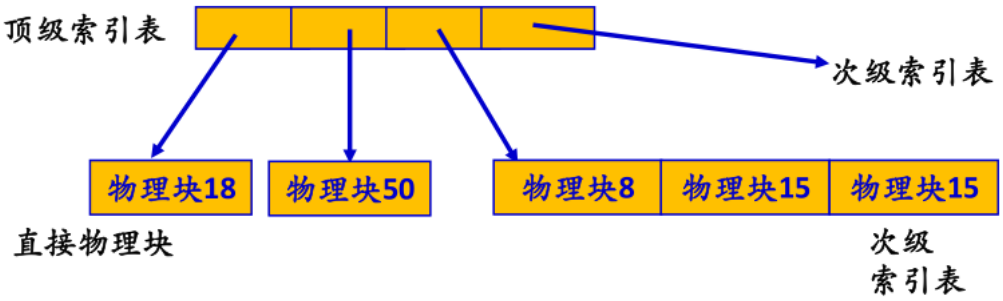
    - 既能顺序存取，又能随机存取
    - 满足了文件动态增长、插入删除的要求
    - 能充分利用磁盘空间
- 缺点
    - 较多的寻道次数和寻道时间
    - 索引表本身带来了系统开销，如：内存、磁盘空间，存取时间

索引表的组织方式

- 问题：索引表很大，需要多个物理块存放时怎么办?
- 链接方式：一个盘块存一个索引表，多个索引表链接起来
- 多级索引方式：将文件的索引表地址放在另一个索引表中
- 综合模式：直接索引方式 与 间接索引方式 结合

多级索引与综合模式





UNIX的三级索引结构

UNIX文件系统采用的是多级索引结构(综合模式)

- 每个文件的索引表有15个索引项，每项2个字节
- 前12项直接存放文件的物理块号（直接寻址）
- 如果文件大于12块，则利用第13项指向一个物理块，在该块中存放文件物理块的块号（一级索引表）

  假设扇区大小为512字节，物理块等于扇区块大小，一级索引表可以存放256个物理块号

  对于更大的文件还可利用第14和第15项作为二级和三级索引表

7.6 文件系统的实现

概述

实现文件系统需要考虑：磁盘上 与 内存中 的 内容布局

- 磁盘上
  - 如何启动操作系统?
  - 磁盘是怎样管理的？怎样获取磁盘的有关信息?
  - 目录文件在磁盘上怎么存放？普通文件在磁盘上怎么存放?
- 内存中
  当进程使用文件时，操作系统是如何支持的?
  文件系统的内存数据结构

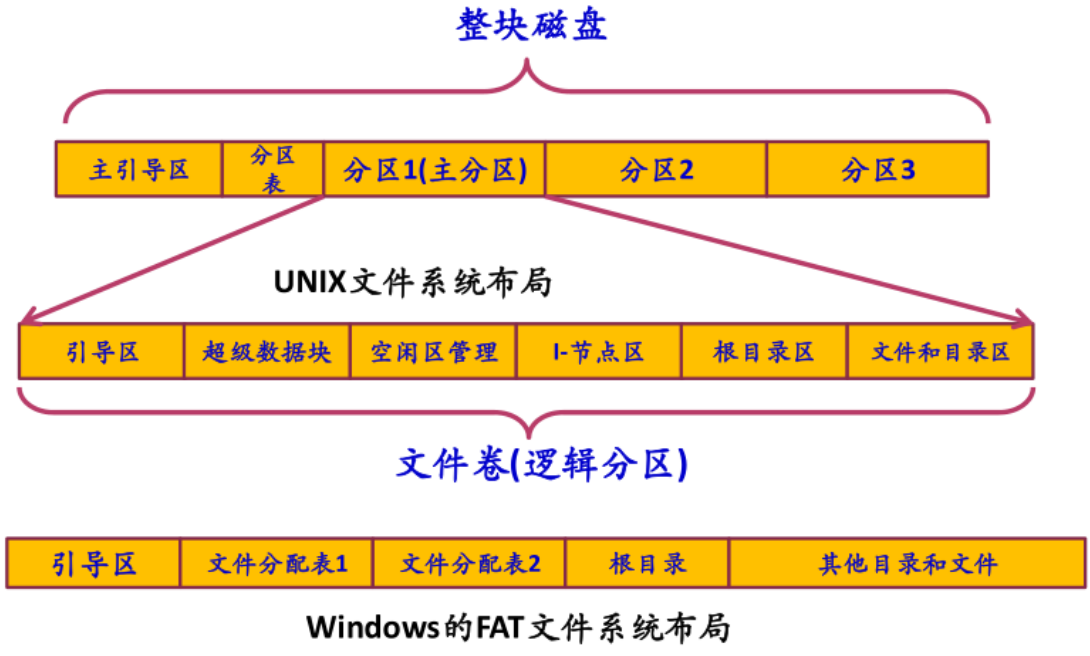
相关术语

- 磁盘分区(partition)：把一个物理磁盘的存储空间划分为几个相互独立的部分，称为分区
- 文件卷(volume)：磁盘上的逻辑分区，由一个或多个物理块(簇)组成
  - 一个文件卷可以是整个磁盘 或 部分磁盘 或 跨盘（RAID）
  - 同一个文件卷中使用同一份管理数据进行文件分配和磁盘空闲空间管理，不同的文件卷中的管理数据是相互独立的
  - 一个文件卷上：包括文件系统信息、一组文件（用户文件、目录文件）、未分配空间
  - 块（Block）或 簇（Cluster） ： 一个或多个（2的幂）连续的扇区，可寻址数据块
- 格式化(format)：在一个文件卷上建立文件系统，即建立并初始化用于文件分配和磁盘空闲空间管理的管理数据
  (元数据)

磁盘上的内容

- 引导区
  包括了从该卷引导操作系统所需要的信息，每个卷（分区）一个，通常为第一个扇区
- 卷（分区）信息
  包括该卷（分区）的块（簇）数、块（簇）大小、空闲块（簇）数量和指针、空闲FCB数量和指针……
- 目录文件(根目录文件及其他目录文件)
- 用户文件

磁盘上文件系统的布局



内存中所需的数据结构—以Unix为例

- 系统打开文件表
  - 整个系统一张
  - 放在内存：用于保存已打开文件的FCB
  - FCB(i节点)信息|引用计数|修改标记
- 用户打开文件表
  - 每个进程一个
  - 进程的PCB中记录了用户打开文件表的位置

  文件描述符|打开方式|读写指针|系统打开文件表索引

7.7 文件系统示例

Unix

文件目录检索

访问一个文件 → 两步骤：文件名→文件目录—FCB→磁盘

- 目录检索
  用户给出文件名 → 按文件名查找到目录项/FCB
  根据路径名检索：
  全路径名：从根开始 \A\B\C\File1
  相对路径：从当前目录开始 C\File1
- 文件寻址
  根据目录项/FCB中文件物理地址等信息，计算出文件中任意记录或字符在存储介质上的地址
- 如何加快目录检索?

  一种解决方案：
  目录项分解法：即把FCB分成两部分
  - 符号目录项：文件名，文件号
  - 基本目录项：除文件名外的所有字段

  例子：UNIX的I节点（索引节点 或 inode）

改进后的好处

目录文件改进后减少了访盘次数，提高了文件检索速度

Unix文件系统

- FCB = 目录项 + i节点
- 目录项：文件名 + i节点号
- 目录文件由目录项构成
- i节点：描述文件的相关信息
- 每个文件由一个目录项、一个i节点和若干磁盘块构成

FAT

Windows FAT16文件系统

- 簇大小：1、2、4、8、16、32或64扇区
- 文件系统的数据记录在“引导扇区”中
- 文件分配表FAT的作用
  描述簇的分配状态、标注下一簇的簇号等
- FAT表项：2字节

- 目录项：32字节
- 根目录大小固定

FAT文件系统 MBR

FAT文件系统 DBR FAT32

引导扇区-BIOS参数块 FAT32

引导扇区-扩展BIOS参数块(EBPB) FAT32

文件分配表FAT

- 可以把文件分配表看成是一个整数数组，每个整数代表磁盘分区的一个簇号
- 状态：未使用、坏簇、系统保留、被文件占用（下一簇簇号）、最后一簇（0xFFFF）
- 簇号从0开始编号，簇0和簇1是保留的

FAT 16目录项

FAT32文件系统

- FAT32的根目录区（ROOT区）不是固定区域、固定大小，而是数据区的一部分，采用与子目录文件相同的管理方式
- 目录项仍占32字节，但分为各种类型（包括：“.”目录项、“..”目录项、短文件名目录项、长文件名目录项、卷标项（根目录）、已删除目录项（第一字节为0xE5）等）
- 支持长文件名格式
- 支持Unicode
- 不支持高级容错特性，不具有内部安全特性

7.8 文件操作的实现

- 创建文件：建立系统与文件的联系，实质是建立文件的FCB
  - 在目录中为新文件建立一个目录项，根据提供的参数及需要填写相关内容
  - 分配必要的存储空间
- 打开文件：
  根据文件名在文件目录中检索，并将该文件的目录项读入内存，建立相应的数据结构，为后续的文件操作做好准备
  文件描述符/文件句柄

文件操作 建立文件

create（文件名，访问权限）

1. 检查参数的合法性，如：文件名是否符合命名规则；有无重名文件；
   合法→②，否则→报错、返回
2. 申请空闲目录项，并填写相关内容；
3. 为文件申请磁盘块；（？）
4. 返回

文件操作 打开文件

为文件读写做准备
给出文件路径名，获得文件句柄(file handle)或文件描述符(file descriptor)，需将该文件的目录项读到内存，fd=open（文件路径名，打开方式）

1. 根据文件路径名查目录，找到目录项 (或I节点号)；
2. 根据文件号查系统打开文件表，看文件是否已被打开；是 → 共享计数加1，否则 → 将目录项 (或I节点)等信息填入系统打开文件表空表项，共享计数置为1；
3. 根据打开方式、共享说明和用户身份检查访问合法性；
4. 在用户打开文件表中获取一空表项，填写打开方式等，并指向系统打开文件表对应表项返回信息：
   fd：文件描述符，是一个非负整数，用于以后读写文件

文件操作 指针定位

seek（fd, 新指针的位置）
系统为每个进程打开的每个文件维护一个读写指针，即相对于文件开头的偏移地址（读写指针指向每次文件读的开始位置，在每次读写完成后，读写指针按照读写的数据量自动后移相应数值）

1. 由fd查用户打开文件表，找到对应的表项；
2. 将用户打开文件表中文件读写指针位置设为新指针的位置，供后继读写命令存取该指针处文件内容

文件操作 读文件

read（文件描述符，读指针，要读的长度，内存目的地址）

1. 根据打开文件时得到的文件描述符，找到相应的文件控制块（目录项）
   确定读操作的合法性
   读操作合法→②，否则→出错处理
   问题：文件尚未打开?
2. 将文件的逻辑块号转换为物理块号
   根据参数中的读指针、长度与文件控制块中的信息，确定块号、块数、块内位移
3. 申请缓冲区
4. 启动磁盘I/O操作，把磁盘块中的信息读入缓冲区，再传送到指定的内存区（多次读盘）
5. 反复执行③、④直至读出所需数量的数据或读至文件尾

怎样实现系统调用rename（给文件重命名）

7.9 文件系统的管理

文件系统的可靠性：抵御和预防各种物理性破坏和人为性破坏的能力

- 坏块问题

文件系统备份

通过转储操作，形成文件或文件系统的多个副本

全量转储：定期将所有文件拷贝到后援存储器
增量转储：只转储修改过的文件，即两次备份之间的修改，减少系统开销

物理转储：从磁盘第0块开始，将所有磁盘块按序输出到磁带
逻辑转储：从一个或几个指定目录开始，递归地转储自给定日期后所有更改的文件和目录

文件系统的一致性

- 问题的产生：磁盘块 → 内存 → 写回磁盘块
  若在写回之前，系统崩溃，则文件系统出现不一致
- 解决方案：
  设计一个实用程序，当系统再次启动时，运行该程序，检查磁盘块和目录系统

磁盘块的一致性检查

UNIX一致性检查工作过程：
两张表，每块对应一个表中的计数器，初值为0
表一：记录了每块在文件中出现的次数
表二：记录了每块在空闲块表中出现的次数

文件系统的写入策略

考虑文件系统一致性和速度

- 通写（write-through）：内存中的修改立即写到磁盘
  缺点：速度性能差
  例： FAT文件系统
- 延迟写（lazy-write）：利用回写（write back）缓存的方法得到高速
  缺点：可恢复性差
- 可恢复写（transaction log）：采用事务日志来实现文件系统的写入
  既考虑安全性，又考虑速度性能
  例：NTFS

7.10 文件系统的安全性

文件保护机制

用户是谁？用户拥有什么？用户知道什么？

- 用于提供安全性、特定的操作系统机制
- 对拥有权限的用户，应该让其进行相应操作，否则，应禁止
- 防止其他用户冒充对文件进行操作

实现：

- 用户身份验证
- 访问控制

文件的访问控制

- 主动控制：访问控制表
  - 每个文件一个
  - 记录用户ID和访问权限
  - 用户可以是一组用户
  - 文件可以是一组文件
- 能力表(权限表)
  - 每个用户一个
  - 记录文件名及访问权限
  - 用户可以是一组用户
  - 文件可以是一组文件

Unix的文件访问控制

采用文件的二级存取控制，审查用户的身份、审查操作的合法性

- 第一级：对访问者的识别对用户分类：
  - 文件主（owner）
  - 文件主的同组用户（group）
  - 其他用户（other）
- 第二级：对操作权限的识别对操作分类：
  - 读操作（r）
  - 写操作（w）
  - 执行操作（x）
  - 不能执行任何操作（-）

7.11 文件系统的性能

文件系统的性能问题

- 磁盘服务→ 速度成为系统性能的主要瓶颈之一
  设计文件系统应尽可能减少磁盘访问次数
- 提高文件系统性能的方法：
  目录项(FCB)分解、当前目录、磁盘碎片整理块高速缓存、磁盘调度、提前读取、合理分配磁盘空间、信息的优化分布、RAID技术... ...

块高速缓存(BLOCK CACHE)

又称为文件缓存、磁盘高速缓存、缓冲区高速缓存是指：在内存中为磁盘块设置的一个缓冲区，保存了磁盘中某些块的副本

- 检查所有的读请求，看所需块是否在块高速缓存中
- 如果在，则可直接进行读操作；否则，先将数据块读入块高速缓存，再拷贝到所需的地方
- 由于访问的局部性原理，当一数据块被读入块高速缓存以满足一个I/O请求时，很可能将来还会再次访问到这一数据块

关于实现

- 块高速缓存的组织
- 块高速缓存的置换（修改LRU）

  该块是否不久后会再次使用
- 块高速缓存写入策略

  该块是否会影响文件系统的一致性

提前读取

思路：每次访问磁盘，多读入一些磁盘块

- 依据：程序执行的空间局部性原理
- 开销：较小(只有数据传输时间)
- 具有针对性

Windows的文件访问方式

- 不使用文件缓存

    - 普通方式
    - 通过Windows提供的FlushFileBuffer函数实现
- 使用文件缓存
    - 预读取。每次读取的块大小、缓冲区大小、置换方式
    - 写回。写回时机选择、一致性问题
- 异步模式
    - 不再等待磁盘操作的完成
    - 使处理器和I/O并发工作
- 用户对磁盘的访问通过访问文件缓存来实现
    - 由Windows的Cache Manager实现对缓存的控制
        - 读取数据的时候预取
        - 在Cache满时，根据LRU原则清除缓存的内容
        - 定期更新磁盘内容使其与Cache一致（1秒）
    - Write-back机制
      在用户要对磁盘写数据时，只更改Cache中的内容，由Cache Manager决定何时将更新反映到磁盘

合理分配磁盘空间

分配磁盘块时，把有可能顺序存取的块放在一起→ 尽量分配在同一柱面上，从而减少磁盘臂的移动次数和距离

磁盘调度

当有多个访盘请求等待时，采用一定的策略，对这些请求的服务顺序调整安排
→ 降低平均磁盘服务时间，达到公平、高效
公平：一个I/O请求在有限时间内满足
高效：减少设备机械运动带来的时间开销

> 一次访盘时间 = 寻道时间+旋转延迟时间+传输时间
> 减少寻道时间
> 减少延迟时间

磁盘调度算法

- 先来先服务(FCFS)：按访问请求到达的先后次序服务
    - 优点：简单，公平
    - 缺点：效率不高，相临两次请求可能会造成最内到最外的柱面寻道，使磁头反复移动，增加了服务时间，对机械也不利
- 最短寻道时间优先(Shortest Seek Time First)：优先选择距当前磁头最近的访问请求进行服务，主要考虑寻道优先
    - 优点：改善了磁盘平均服务时间
    - 缺点：造成某些访问请求长期等待得不到服务
- 扫描算法SCAN（电梯算法） 折中权衡距离、方向

  当设备无访问请求时，磁头不动；当有访问请求时，磁头按一个方向移动，在移动过程中对遇到的访问请求进行服务，然后判断该方向上是否还有访问请求，如果有则继续扫描；否则改变移动方向，并为经过的访问请求服务，如此反复
- 单向扫描调度算法C-SCAN 减少了新请求的最大延迟
    - 总是从0号柱面开始向里扫描
    - 按柱面(磁道)位置选择访问者

    - 移动臂到达最后一个柱面后，立即带动读写磁头快速返回到0号柱面
    - 返回时不为任何的等待访问者服务
    - 返回后可再次进行扫描
- N-step-SCAN策略 克服“磁头臂的粘性”
    - 把磁盘请求队列分成长度为N的子队列，每一次用SCAN处理一个子队列
    - 在处理某一个队列时，新请求添加到其他子队列中
    - 如果最后剩下的请求数小于N，则它们全都将在下一次扫描时处理
    - N值比较大时，其性能接近SCAN；当N = 1时，即FIFO
- FSCAN策略 克服“磁头臂的粘性”
    - 使用两个子队列
    - 扫描开始时，所有请求都在一个队列中，而另一个队列为空
    - 扫描过程中，所有新到的请求都放入另一个队列中
    - 对新请求的服务延迟到处理完所有老请求之后
- 旋转调度算法：根据延迟时间来决定执行次序的调度
    - 若干等待访问者请求访问同一磁头上的不同扇区
    - 若干等待访问者请求访问不同磁头上的不同编号的扇区
    - 若干等待访问者请求访问不同磁头上具有相同的扇区
    - 解决方案：
        - 对于前两种情况：总是让首先到达读写磁头位置下的扇区先进行传送操作
        - 对于第三种情况：这些扇区同时到达读写磁头位置下，可任意选择一个读写磁头进行传送操作

信息的优化分步

记录在磁道上的排列方式也会影响输入输出操作的时间

记录的成组与分解

- 记录的成组：把若干个逻辑记录合成一组存放一块的工作
- 进行成组操作时必须使用内存缓冲区，缓冲区的长度等于逻辑记录长度乘以成组的块因子
- 成组目的：提高了存储空间的利用率；减少了启动外设的次数，提高系统的工作效率
- 记录的分解：从一组逻辑记录中把一个逻辑记录分离出来

RAID（独立磁盘冗余阵列，Redundant Arrays of Independent Disks)

多块磁盘按照一定要求构成一个独立的存储设备
目标：提高可靠性和性能
考虑：磁盘存储系统 的 速度、容量、容错、数据灾难发生后的数据恢复

RAID技术的思想

数据是如何组织的?

- 通过把多个磁盘组织在一起，作为一个逻辑卷提供磁盘跨越功能
- 通过把数据分成多个数据块，并行写入/读出多个磁盘，以提高数据传输率（数据分条stripe）
- 通过镜像或校验操作，提供容错能力（冗余）
  最简单的RAID组织方式：镜像
  最复杂的RAID组织方式：块交错校验

RAID0 条带化

无冗余（即无差错控制）性能最佳

数据分布在阵列的所有磁盘上

- 有数据请求时，同时多个磁盘并行操作
- 充分利用总线带宽，数据吞吐率提高，驱动器负载均衡

RAID1 镜像

数据安全性最好

- 最大限度保证数据安全及可恢复性
- 所有数据同时存在于两块磁盘的相同位置
- 磁盘利用率50%

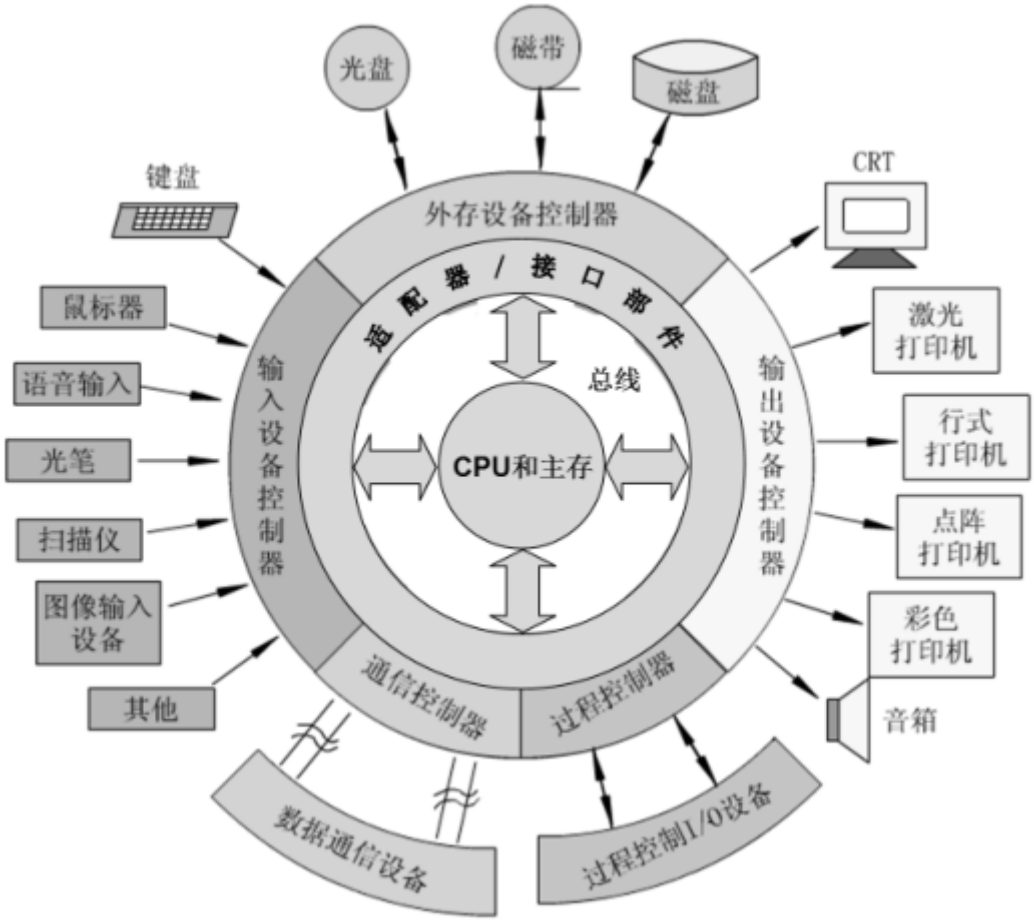
RAID4 交错块奇偶校验

- 带奇偶校验
- 以数据块为单位

8.IO系统

8.1 IO管理概述

计算机IO系统结构



IO的特点

- I/O性能经常成为系统性能的瓶颈
- 操作系统庞大复杂的原因之一：资源多、杂，并发，均来自I/O

- 速度差异很大
- 应用
- 控制接口的复杂性
- 传送单位
- 数据表示
- 错误条件
- 与其他功能联系密切，特别是文件系统

设备的分类—按数据组织分

- 块设备：以数据块为单位存储、传输信息传输速率较高、可寻址（随机读写）
- 字符设备：以字符为单位存储、传输信息传输速率低、不可寻址
- 存储设备（磁盘、磁带）
- 传输设备（网卡、Modem）
- 人机交互设备（显示器、键盘、鼠标）

设备的分类—从资源分配角度

- 独占设备：在一段时间内只能有一个进程使用的设备，一般为低速I/O设备（如打印机，磁带等）
- 共享设备：在一段时间内可有多个进程共同使用的设备，多个进程以交叉的方式来使用设备，其资源利用率高（如硬盘）
- 虚设备：在一类设备上模拟另一类设备，常用共享设备模拟独占设备，用高速设备模拟低速设备，被模拟的设备称为虚设备

  目的：将慢速的独占设备改造成多个用户可共享的设备，提高设备的利用率

  实例：SPOOLing技术，利用虚设备技术——用硬盘模拟输入输出设备

IO管理的目标和任务

- 按照用户的请求，控制设备的各种操作，完成I/O设备与内存之间的数据交换，最终完成用户的I/O请求
  - 设备分配与回收
    - 记录设备的状态
    - 根据用户的请求和设备的类型，采用一定的分配算法，选择一条数据通路
  - 执行设备驱动程序，实现真正的I/O操作
  - 设备中断处理：处理外部设备的中断
  - 缓冲区管理：管理I/O缓冲区
- 建立方便、统一的独立于设备的接口

  通用性：种类繁多、结构各异→设计简单、避免错误→采用统一的方式处理所有设备
  - 方便性：向用户提供使用外部设备的方便接口，使用户编程时不考虑设备的复杂物理特性
  - 统一性：对不同的设备采取统一的操作方式，即在用户程序中使用的是逻辑设备
    - 逻辑设备与物理设备
    - 屏蔽硬件细节（设备的物理特性、错误处理、不同I/O过程的差异性）
- 充分利用各种技术（通道，中断，缓冲，异步I/O等）提高CPU与设备、设备与设备之间的并行工作能力，充分利用资源，提高资源利用率

  性能CPU与I/O的速度差别大→减少由于速度差异造成的整体性能开销→尽量使两者交叠运行
  - 并行性
  - 均衡性（使设备充分忙碌）
- 保护：设备传送或管理的数据应该是安全的、不被破坏的、保密的

8.2 IO硬件组成

I/O设备一般由机械和电子两部分组成

1. 机械部分是设备本身（物理装置）
2. 电子部分又称设备控制器(或适配器)
    - (端口) 地址译码
    - 按照主机与设备之间约定的格式和过程接受计算机发来的数据和控制信号 或 向主机发送数据和状态信号
    - 将计算机的数字信号转换成机械部分能识别的模拟信号，或反之
    - 实现设备内部硬件缓冲、数据加工等提高性能或增强功能

设备接口—控制器的作用

- 操作系统将命令写入控制器的接口寄存器（或接口缓冲区）中，以实现输入 / 输出，并从接口寄存器读取状态信息或结果信息
- 当控制器接受一条命令后，可独立于CPU完成指定操作，CPU可以另外执行其他计算；命令完成时，控制器产生一个中断，CPU响应中断，控制转给操作系统；通过读控制器寄存器中的信息，获得操作结果和设备状态
- 控制器与设备之间的接口常常是一个低级接口
- 控制器的任务：把串行的位流转换为字节块，并进行必要的错误修正：首先，控制器按位进行组装，然后存入控制器内部的缓冲区中形成以字节为单位的块；在对块验证检查和并证明无错误时，再将它复制到内存中

IO端口地址

- I/O端口地址：接口电路中每个寄存器具有的、唯一的地址，是个整数
- 所有I/O端口地址形成I/O端口空间(受到保护)
- I/O指令形式与I/O地址是相互关联的， 主要有两种形式：
    - 内存映像编址（内存映像I/O模式）
    - I/O独立编址（I/O专用指令）

IO独立编址

- 分配给系统中所有端口的地址空间完全独立，与内存地址空间无关
- 使用专门的I/O指令对端口进行操作
- 优点
    - 外设不占用内存的地址空间
    - 编程时，易于区分是对内存操作还是对I/O端口操作
- 缺点：I/O端口操作的指令类型少，操作不灵活

  例子：8086/8088，分配给I/O端口的地址空间64K，0000H~0FFFFH，只能用in和out指令进行读写操作

内存映像编址

- 分配给系统中所有端口的地址空间与内存的地址空间统一编址
- 把I/O端口看作一个存储单元，对I/O的读写操作等同于对内存的操作
- 优点
    - 凡是可对内存操作的指令都可对I/O端口操作

    - 不需要专门的I/O指令
    - I/O端口可占有较大的地址空间
- 缺点：占用内存空间

内存映射IO的优点

- 不需要特殊的保护机制来阻止用户进程执行I/O操作

  操作系统必须要做的事情：避免把包含控制寄存器的那部分地址空间放入任何用户的虚拟地址空间之中

- 可以引用内存的每一条指令也可以引用控制寄存器

  例如，如果指令TEST可以测试一个内存字是否为0，那么它也可以用来测试一个控制寄存器是否为0

内存映射IO的缺点

- 对一个设备控制寄存器不能进行高速缓存
- 考虑以下汇编代码循环，第一次引用PORT_4将导致它被高速缓存，随后的引用将只从高速缓存中取值并且不会再查询设备，之后当设备最终变为就绪时，软件将没有办法发现这一点，结果循环将永远进行下去
- 为避免这一情形，硬件必须针对每个页面具备选择性禁用高速缓存的能力，操作系统必须管理选择性高速缓存，所以这一特性为硬件和操作系统两者增添了额外的复杂性

8.3 IO控制方式

1. 可编程I/O（轮询/查询）
   由CPU代表进程给I/O模块发I/O命令，进程进入忙等待，直到操作完成才继续执行
2. 中断驱动I/O
   为了减少设备驱动程序不断地询问控制器状态寄存器的开销
   I/O操作结束后，由设备控制器主动通知设备驱动程序
3. DMA

IO部件的演化

1. CPU直接控制外围设备
2. 增加了控制器或I/O部件，CPU使用非中断的可编程I/OCPU开始从外部设备接口的具体细节中分离出来
3. 与2相同，但采用了中断方式CPU无需花费等待执行一次I/O操作所需的时间，效率提高
4. I/O部件通过DMA直接控制存储器可以在没有CPU参与的情况下，从内存中移出或者往内存中移入一块数据，仅仅在传送开始和结束时需要CPU干预
5. I/O部件增强为一个单独的处理器，有专门为I/O设计的指令集；CPU指导I/O处理器执行内存中的一个I/O程序。I/O处理器在没有CPU干涉的情况下取指令并执行这些指令
6. I/O部件有自己的局部存储器(其本身就是一台计算机)使用这种体系结构可以控制许多I/O设备，并且使需要CPU参与程度降到最小（通常用于控制与交互终端的通信，I/O处理器负责大多数控制终端的任务）

8.4 IO软件的组成

分层的设计思想

- 把I/O软件组织成多个层次
- 每一层都执行操作系统所需要的功能的一个相关子集，它依赖于更低一层所执行的更原始的功能，从而可以

隐藏这些功能的细节；同时，它又给高一层提供服务
- 较低层考虑硬件的特性，并向较高层软件提供接口
- 较高层不依赖于硬件，并向用户提供一个友好的、清晰的、简单的、功能更强的接口

IO软件层次

1. 用户进程层执行输入输出系统调用，对I/O数据进行格式化，为假脱机输入/输出作准备
2. 独立于设备的软件实现设备的命名、设备的保护、成块处理、缓冲技术和设备分配
3. 设备驱动程序设置设备寄存器、检查设备的执行状态
4. 中断处理程序负责I/O完成时，唤醒设备驱动程序进程，进行中断处理
5. 硬件层实现物理I/O的操作

设备独立性（设备无关性）

好处：设备分配时的灵活性，易于实现I/O重定向

用户编写的程序可以访问任意I/O设备，无需事先指定设备

- 从用户角度：用户在编制程序时，使用逻辑设备名，由系统实现从逻辑设备到物理设备（实际设备）的转换，并实施I/O操作
- 从系统角度：设计并实现I/O软件时，除了直接与设备打交道的低层软件之外，其他部分的软件不依赖于硬件

8.5 IO相关技术

引入缓冲技术解决什么问题?

操作系统中最早引入的技术
→ 解决CPU与I/O设备之间速度的不匹配问题：**凡是数据到达和离去速度不匹配的地方均可采用缓冲技术**
→ 提高CPU与I/O设备之间的并行性
→ 减少了I/O设备对CPU的中断请求次数，放宽CPU对中断响应时间的要求

缓冲技术实现

- 缓冲区分类
  - 硬缓冲：由硬件寄存器实现（例如：设备中设置的缓冲区）
  - 软缓冲：在内存中开辟一个空间，用作缓冲区
- 缓冲区管理
  单缓冲
  双缓冲
  缓冲池（多缓冲，循环缓冲）：统一管理多个缓冲区，采用有界缓冲区的生产者/消费者模型对缓冲池中的缓冲区进行循环使用

UNIX SYSTEM V 缓冲技术

- 采用缓冲池技术，可平滑和加快信息在内存和磁盘之间的传输
- 缓冲区结合提前读和延迟写技术对具有重复性及阵发性I/O进程、提高I/O速度很有帮助
- 可以充分利用之前从磁盘读入、虽已传入用户区但仍在缓冲区的数据（尽可能减少磁盘I/O的次数，提高系统运行的速度）
- 缓冲池：200个缓冲区(512字节或1024字节)
- 每个缓冲区由两部分组成：缓冲控制块或缓冲首部 + 缓冲数据区

- 系统通过缓冲控制块来实现对缓冲区的管理
- 空闲缓冲区队列(av链) 队列头部为bfreelist
- 设备缓冲队列（b链）链接所有分配给各类设备使用的缓冲区，按照散列方式组织

说明：

- 逻辑设备号和盘块号分别标志出文件系统和数据所在的盘块号，是缓冲区的唯一标志
- 状态标识缓冲区的当前状态：忙/闲、上锁/开锁、是否延迟写、数据有效性等
- 两组指针（av和b）用于对缓冲池的分配管理

每个缓冲区同时在av链和b链：

- 开始：在空闲av链(缓冲区未被使用时)
- 开始IO请求：在设备IO请求队列和设备b链
- IO完成：在空闲av链和设备b链
- 当进程想从指定的盘块读取数据时，系统根据盘块号从设备b链（散列队列）中查找，如找到缓冲区，则将该缓冲区状态标记为“忙”，并从空闲av队列中取下，并完成从缓冲区到内存用户区的数据传送
- 如果在设备b链中未找到时，则从空闲av链队首摘取一个缓冲区，插入设备I/O请求队列；并从原设备b链中取下，插入由读入信息盘块号确定的新的设备b链中
- 当数据从磁盘块读入到缓冲区后，缓冲区从设备I/O请求队列取下；当系统完成从缓冲区到内存用户区的数据传送后，要把缓冲区释放，链入空闲av链队尾
- 当数据从磁盘块读入到缓冲区，并传送到内存用户区后，该缓冲区一直保留在原设备b链中，即它的数据一直有效。若又要使用它，则从空闲av链中取下，使用完后插入到空闲av链队尾。若一直未使用它，则该缓冲区从空闲av链队尾慢慢升到队首，最后被重新分配，旧的盘块数据才被置换

  系统对缓冲区的分配是采用近似LRU算法

8.6 IO设备管理

设备管理有关的数据结构

- 描述设备、控制器等部件的表格：系统中常常为每一个部件、每一台设备分别设置一张表格，常称为设备表或部件控制块。这类表格具体描述设备的类型、标识符、状态，以及当前使用者的进程标识符等
- 建立同类资源的队列：系统为了方便对I/O设备的分配管理，通常在设备表的基础上通过指针将相同物理属性的设备连成队列（称设备队列）
- 面向进程I/O请求的动态数据结构：每当进程发出I/O请求时，系统建立一张表格（称I/O请求包），将此次I/O请求的参数填入表中，同时也将该I/O有关的系统缓冲区地址等信息填入表中。I/O请求包随着I/O的完成而被删除
- 建立I/O队列：如请求包队列

设备独占的分配

- 在申请设备时，如果设备空闲，就将其独占，不再允许其他进程申请使用，一直等到该设备被释放，才允许被其他进程申请使用
  考虑效率问题，并避免由于不合理的分配策略造成死锁
- 静态分配：在进程运行前, 完成设备分配；运行结束时，收回设备
  缺点：设备利用率低
- 动态分配：
  在进程运行过程中，当用户提出设备要求时，进行分配，一旦停止使用立即收回
  优点：效率好；
  缺点：分配策略不好时, 产生死锁

分时共享设备的分配

- 所谓分时式共享就是以一次I/O为单位分时使用设备，不同进程的I/O操作请求以排队方式分时地占用设备进行I/O
- 由于同时有多个进程同时访问，且访问频繁，就会影响整个设备使用效率，影响系统效率。因此要考虑多个访问请求到达时服务的顺序，使平均服务时间越短越好

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
- 一般，设备驱动程序的任务是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
- 每一个控制器都设有一个或多个设备寄存器，用来存放向设备发送的命令和参数。设备驱动程序负责释放这些命令，并监督它们正确执行
- 在设备驱动程序的进程释放一条或多条命令后，系统有两种处理方式，多数情况下，执行设备驱动程序的进程必须等待命令完成，这样，在命令开始执行后，它阻塞自已，直到中断处理时将它解除阻塞为止；而在其它情况下，命令执行不必延迟就很快完成
- 设备驱动程序与外界的接口
    - 与操作系统的接口
      为实现设备无关性，设备作为特殊文件处理。用户的I/O请求、对命令的合法性检查以及参数处理在文件系统中完成。在需要各种设备执行具体操作时，通过相应数据结构转入不同的设备驱动程序
    - 与系统引导的接口（初始化，包括分配数据结构，建立设备的请求队列）
    - 与设备的接口
- 设备驱动程序接口函数
    - 驱动程序初始化函数（如向操作系统登记该驱动程序的接口函数，该初始化函数在系统启动时或驱动程序安装入内核时执行）
    - 驱动程序卸载函数
    - 申请设备函数
    - 释放设备函数
    - I/O操作函数
      对独占设备，包含启动I/O的指令；对共享设备，将I/O请求形成一个请求包，排到设备请求队列，如果请求队列空，则直接启动设备
    - 中断处理函数
      对I/O完成做善后处理，一般是唤醒等待刚完成I/O请求的阻塞进程，使其能进一步做后续工作；如果存在I/O请求队列，则启动下一个I/O请求

一种典型的实现方案

- I/O进程：专门处理系统中的I/O请求和I/O中断工作
- I/O请求的进入
    - 用户程序：调用send将I/O请求发送给I/O进程；调用block将自己阻塞，直到I/O任务完成后被唤醒
    - 系统：利用wakeup唤醒I/O进程，完成用户所要求的I/O处理
- I/O中断的进入
    - 当I/O中断发生时，内核中的中断处理程序发一条消息给I/O进程，由I/O进程负责判断并处理中断
- I/O进程
    - 是系统进程，一般赋予最高优先级。一旦被唤醒，它可以很快抢占处理机投入运行

- I/O进程开始运行后，首先关闭中断，然后用receive去接收消息
  两种情形：
  - 没有消息，则开中断，将自己阻塞
  - 有消息，则判断消息类型（I/O请求或I/O中断）
    a. I/O请求 准备通道程序，发出启动I/O指令，继续判断有无消息
    b. I/O中断，进一步判断正常或异常结束
    正常：唤醒要求进行I/O操作的进程
    异常：转入相应的错误处理程序

8.7 IO性能问题

- 使CPU利用率尽可能不被I/O降低
- 使CPU尽可能摆脱I/O

解决方式：

- 减少或缓解速度差距 → 缓冲技术
- 使CPU不等待I/O → 异步I/O
- 让CPU摆脱I/O操作 → DMA、通道

异步传输

Windows提供两种模式的I/O操作：异步和同步

- 异步模式：用于优化应用程序的性能
  - 通过异步I/O，应用程序可以启动一个I/O操作，然后在I/O请求执行的同时继续处理
  - 基本思想：填充I/O操作间等待的CPU时间
- 同步I/O：应用程序被阻塞直到I/O操作完成

同步传输IO流程

- 在I/O处理过程中，CPU处于空闲等待状态
- 而在处理数据的过程中，不能同时进行I/O操作

异步传输IO的基本思想

- 系统实现
  - 通过切换到其他线程保证CPU利用率
  - 对少量数据的I/O操作会引入切换的开销
- 用户实现
  - 将访问控制分成两段进行
  - 发出读取指令后继续做其他操作
  - 当需要用读入的数据的时候，再使用wait命令等待其完成
  - 不引入线程切换，减少开销

9. 死锁

9.1 死锁的基本概念

死锁的定义

- 一组进程中，每个进程都无限等待被该组进程中另一进程所占有的资源，因而永远无法得到的资源，这种现象称为进程死锁，这一组进程就称为死锁进程
- 如果死锁发生，会浪费大量系统资源，甚至导致系统崩溃

> 参与死锁的所有进程都在等待资源
>
> 参与死锁的进程是当前系统中所有进程的子集

为什么会出现死锁

资源数量有限、锁和信号量错误使用
资源的使用方式： “申请--分配--使用--释放”模式

- 可重用资源：可被多个进程多次使用
  - 可抢占资源与不可抢占资源
  - 处理器、I/O部件、内存、文件、数据库、信号量
- 可消耗资源：只可使用一次、可创建和销毁的资源
  - 信号、中断、消息

活锁与饥饿

- 活锁
  - 先加锁
  - 再轮询 → 既无进展也没有阻塞
- 饥饿：资源分配策略决定

产生死锁的必要条件

- 互斥使用(资源独占)：一个资源每次只能给一个进程使用
- 占有且等待(请求和保持，部分分配)：进程在申请新的资源的同时保持对原有资源的占有
- 不可抢占(不可剥夺)
  资源申请者不能强行的从资源占有者手中夺取资源，资源只能由占有者自愿释放
- 循环等待
  存在一个进程等待队列 {P1 , P2 , … , Pn},其中P1等待P2占有的资源，P2等待P3占有的资源，…，Pn等待P1占有的资源，形成一个进程等待环路

9.2 资源分配图(RAG)

用有向图描述系统资源和进程的状态
二元组G=（V，E）
V：结点的集合，分为P(进程)，R(资源)两部分
P = {P1, P2, … , Pn}
R = {R1, R2, … , Rm}
E：有向边的集合，其元素为有序二元组
(Pi, Rj) 或 (Rj, Pi)

资源分配图画法说明

系统由若干类资源构成，一类资源称为一个资源类；每个资源类中包含若干个同种资源，称为资源实例

资源类：用方框表示
资源实例：用方框中的黑圆点表示
进程：用圆圈中加进程名表示
分配边：资源实例 → 进程
申请边：进程 → 资源类

死锁定理

- 如果资源分配图中没有环路，则系统中没有死锁，如果图中存在环路则系统中可能存在死锁
- 如果每个资源类中只包含一个资源实例，则环路是死锁存在的充分必要条件

资源分配图化简

化简步骤：

- 找一个非孤立、且只有分配边的进程结点去掉分配边，将其变为孤立结点
- 再把相应的资源分配给一个等待该资源的进程即将该进程的申请边变为分配边

重复

9.3 死锁预防

破坏产生死锁的四个必要条件

- 不考虑此问题（鸵鸟算法）
- 不让死锁发生
    - 死锁预防，静态策略：设计合适的资源分配算法，不让死锁发生
    - 死锁避免，动态策略：以不让死锁发生为目标，跟踪并评估资源分配过程，根据评估结果决策是否分配
    - 让死锁发生：死锁检测与解除

死锁预防定义：在设计系统时，通过确定资源分配算法，排除发生死锁的可能性

- 具体的做法是：防止产生死锁的四个必要条件中任何一个条件发生
    - 破坏“互斥使用/资源独占”条件
        - 资源转换技术：把独占资源变为共享资源
        - SPOOLing技术的引入
          解决不允许任何进程直接占有打印机的问题设计一个“守护进程/线程”负责管理打印机，进程需要打印时，将请求发给该daemon，由它完成打印任务
    - 破坏“占有且等待”条件
        - 实现方案1：要求每个进程在运行前必须一次性申 请它所要求的所有资源，且仅当该进程所要资源均可满足时才给予一次性分配
          问题：资源利用率低；“饥饿”现象
        - 实现方案2：在允许进程动态申请资源前提下规定，一个进程在申请新的资源不能立即得到满足而变为等待状态之前，必须释放已占有的全部资源，若需要再重新申请
    - 破坏“不可抢占”条件
        - 当一个进程申请的资源被其他进程占用时，可以通过操作系统抢占这一资源(两个进程优先级不同)
        - 局限性：适用于状态易于保存和恢复的资源CPU、内存
    - 破坏“循环等待”条件
        - 通过定义资源类型的线性顺序实现
        - 实施方案：资源有序分配法，把系统中所有资源编号，进程在申请资源时必须严格按资源编号的递增次序进行，否则操作系统不予分配
        - 实现时要考虑什么问题呢?
        - 例子：解决哲学家就餐问题

为什么资源有序分配法不会产生死锁?

9.4 死锁避免

安全状态、不安全状态

- 死锁避免定义：
  在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统发生死锁或可能发生死锁，则不予分配，否则予以分配
- 安全状态：
  如果系统中存在一个由所有进程构成的安全序列P1，...，Pn，则称系统处于安全状态

安全序列

一个进程序列P_1，$\ldots$，P_n是安全的，如果对于每一个进程$P_i(1 \leq i \leq n)$：它以后还需要的资源量不超过系统当前剩余资源量与所有进程$P_j(j < i)$，当前占有资源量之和则称系统处于安全状态

安全状态一定没有死锁发生

不安全状态：系统中不存在一个安全序列，不安全状态一定导致死锁

9.5 银行家算法

Dijkstra提出(1965)仿照银行发放贷款时采取的控制方式而设计的一种死锁避免算法

应用条件：

1. 在固定数量的进程中共享数量固定的资源
2. 每个进程预先指定完成工作所需的最大资源数量
3. 进程不能申请比系统中可用资源总数还多的资源
4. 进程等待资源的时间是有限的
5. 如果系统满足了进程对资源的最大需求，那么，进程应该在有限的时间内使用资源，然后归还给系统

9.6 死锁检测与解除

死锁检测：

- 允许死锁发生，但是操作系统会不断监视系统进展情况，判断死锁是否真的发生
- 一旦死锁发生则采取专门的措施，解除死锁并以最小的代价恢复操作系统运行

检测时机：

- 当进程由于资源请求不满足而等待时检测死锁
  缺点：系统开销大
- 定时检测
- 系统资源利用率下降时检测死锁

一个简单的死锁检测算法

- 每个进程、每个资源指定唯一编号
- 设置一张资源分配表记录各进程与其占用资源之间的关系
- 设置一张进程等待表记录各进程与要申请资源之间的关系

死锁的解除

重要的是以最小的代价恢复系统的运行
方法如下：

- 撤消所有死锁进程
- 进程回退（Roll back）再启动
- 按照某种原则逐一撤消死锁进程，直到...
- 按照某种原则逐一抢占资源（资源被抢占的进程必须回退到之前的对应状态），直到...

9.7 哲学家聚餐问题

问题描述：

- 有五个哲学家围坐在一圆桌旁，桌中央有一盘通心粉，每人面前有一只空盘子，每两人之间放一只筷子
- 每个哲学家的行为是思考，感到饥饿，然后吃通心粉
- 为了吃通心粉，每个哲学家必须拿到两只筷子，并且每个人只能直接从自己的左边或右边去取筷子（筷子的互斥使用、不能出现死锁现象）

为防止死锁发生可采取的措施

- 最多允许4个哲学家同时坐在桌子周围
- 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子
- 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之