欢迎关注作者公众号





认识操作系统

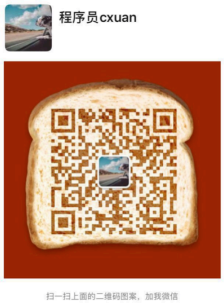
认识操作系统

- 操作系统
- 计算机硬件简介
  - CPU
  - 多线程和多核芯片
  - 内存
    - 寄存器
    - 高速缓存
    - 主存
    - 磁盘
  - I/O 设备
  - 总线
  - 计算机启动过程
- 操作系统博物馆
  - 大型机操作系统
  - 服务器操作系统
  - 多处理器操作系统
  - 个人计算机系统
  - 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
- 操作系统概念
  - 进程
  - 地址空间
  - 文件
  - 保护
  - shell
- 系统调用
  - 用于进程管理的系统调用
  - 用于文件管理的系统调用
  - 用于目录管理的系统调用
  - 其他系统调用
  - Win 32 API
- 操作系统结构
  - 单体系统
  - 分层系统
  - 微内核
  - 客户-服务器模式

进程和线程

- 进程

欢迎添加作者微信



- 进程模型
- 进程的创建
  - 系统初始化
  - 系统调用创建
  - 用户请求创建
  - 批处理创建
- 进程的终止
  - 正常退出
  - 错误退出
  - 严重错误
  - 被其他进程杀死
- 进程的层次结构
  - UNIX 进程体系
  - Windows 进程体系
- 进程状态
- 进程的实现

线程

- 线程的使用
  - 多线程解决方案
  - 单线程解决方案
  - 状态机解决方案
- 经典的线程模型
  - 线程系统调用
- POSIX 线程
- 线程实现
  - 在用户空间中实现线程
  - 在用户空间实现线程的优势
  - 在用户空间实现线程的劣势
- 在内核中实现线程
- 混合实现

进程间通信

- 竞态条件
- 临界区
- 忙等互斥
  - 屏蔽中断
  - 锁变量
  - 严格轮询法
  - Peterson 解法
  - TSL 指令
- 睡眠与唤醒
  - 生产者-消费者问题
- 信号量
  - 用信号量解决生产者 - 消费者问题
- 互斥量
  - Futexes
  - Pthreads 中的互斥量
- 管程

- 消息传递
  - 消息传递系统的设计要点
  - 用消息传递解决生产者-消费者问题
- 屏障
- 避免锁：读-复制-更新

调度

- 调度介绍
  - 进程行为
  - 何时调度
  - 调度算法的分类
  - 调度算法的目标
- 批处理中的调度
  - 先来先服务
  - 最短作业优先
  - 最短剩余时间优先
- 交互式系统中的调度
  - 轮询调度
  - 优先级调度
  - 多级队列
  - 最短进程优先
  - 保证调度
  - 彩票调度
  - 公平分享调度
- 实时系统中的调度
- 调度策略和机制
- 线程调度

内存

无存储器抽象

- 运行多个程序

一种存储器抽象：地址空间

- 地址空间的概念
  - 基址寄存器和变址寄存器
- 交换技术
  - 交换过程
- 空闲内存管理
  - 使用位图的存储管理
  - 使用链表进行管理

虚拟内存

- 分页
  - 存在映射的页如何映射
  - 未映射的页如何映射
- 页表
  - 页表项的结构
- 加速分页过程
  - 转换检测缓冲区
  - 软件 TLB 管理
- 针对大内存的页表

- 多级页表
- 倒排页表
- 页面置换算法
  - 最优页面置换算法
  - 最近未使用页面置换算法
  - 先进先出页面置换算法
  - 第二次机会页面置换算法
  - 时钟页面置换算法
  - 最近最少使用页面置换算法
  - 用软件模拟 LRU
  - 工作集页面置换算法
  - 工作集时钟页面置换算法
  - 页面置换算法小结

文件系统

- 文件
  - 文件命名
  - 文件结构
  - 文件类型
  - 文件访问
  - 文件属性
  - 文件操作
- 目录
  - 一级目录系统
  - 层次目录系统
  - 路径名
  - 目录操作
- 文件系统的实现
  - 文件系统布局
    - 引导块
    - 超级块
    - 空闲空间块
    - 碎片
    - inode
  - 文件的实现
    - 连续分配
    - 链表分配
    - 使用内存表进行链表分配
    - inode
  - 目录的实现
  - 共享文件
  - 日志结构文件系统
  - 日志文件系统
  - 虚拟文件系统
- 文件系统的管理和优化
  - 磁盘空间管理
    - 块大小
    - 记录空闲块

- 磁盘配额
- 文件系统备份
  - 物理转储和逻辑转储
- 文件系统的一致性
- 文件系统性能
  - 高速缓存
  - 块提前读
  - 减少磁盘臂运动
  - 磁盘碎片整理

I/O

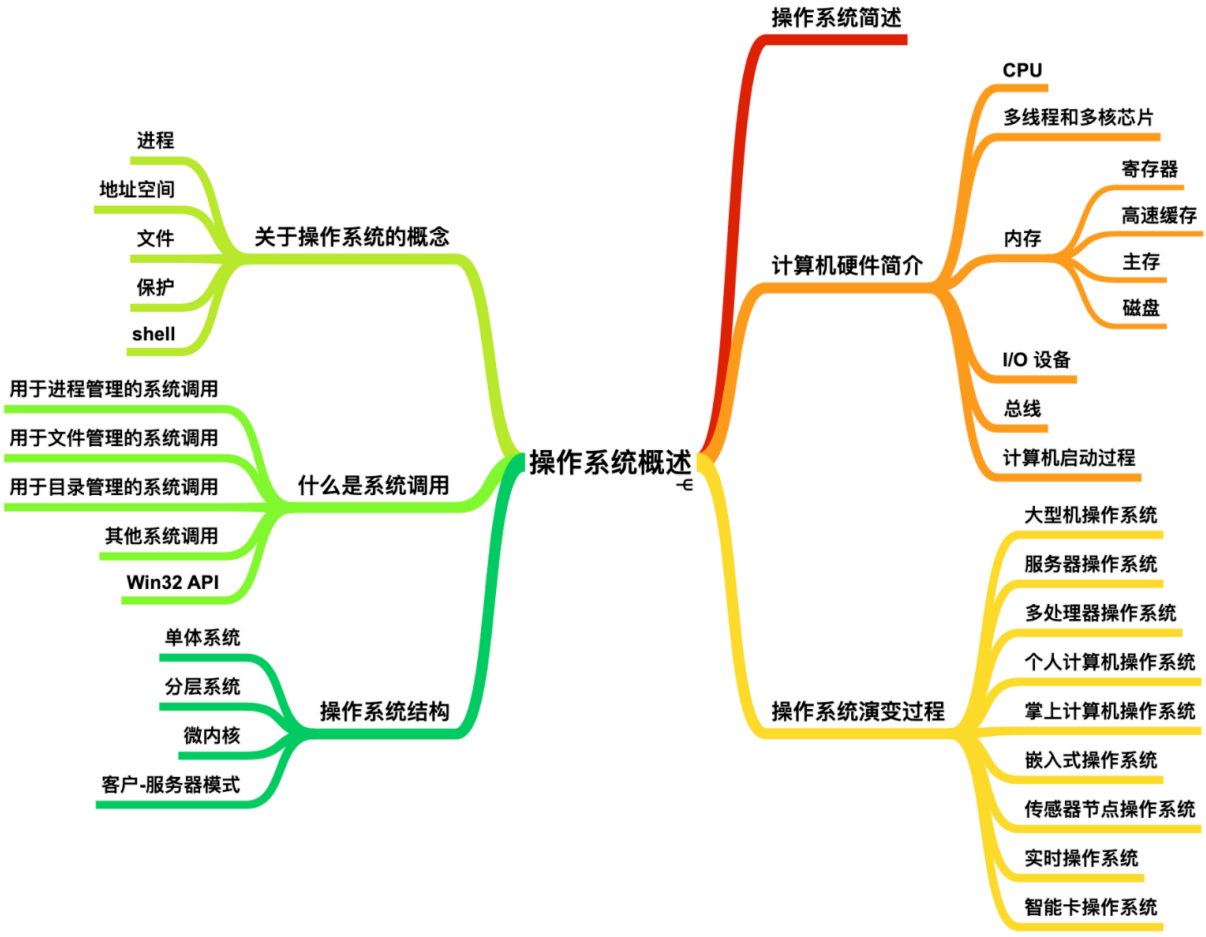
- I/O 设备
  - 块设备
  - 块设备的缺点
  - 字符设备
- 设备控制器
- 内存映射 I/O
  - 内存映射 I/O 的优点和缺点
- 直接内存访问
  - DMA 工作原理
- 重温中断
  - 精确中断和不精确中断
- IO 软件原理
- I/O 软件目标
  - 设备独立性
  - 错误处理
  - 同步和异步传输
  - 缓冲
  - 共享和独占
- 使用程序控制 I/O
- 使用中断驱动 I/O
- 使用 DMA 的 I/O
- I/O 层次结构
- 中断处理程序
- 设备驱动程序
- 与设备无关的 I/O 软件
  - 缓冲
  - 错误处理
  - 设备驱动程序统一接口
  - 分配和释放
  - 设备无关的块
- 用户空间的 I/O 软件
- 盘
- 盘硬件
  - 磁盘
  - RAID
  - 磁盘格式化
- 磁盘臂调度算法

        错误处理
        稳定存储器
    时钟
        时钟硬件
        时钟软件
        软定时器
死锁
    前言
    资源
        可抢占资源和不可抢占资源
        资源获取
    死锁
        资源死锁的条件
        死锁模型
    鸵鸟算法
    死锁检测和恢复
        每种类型一个资源的死锁检测方式
        每种类型多个资源的死锁检测方式
        从死锁中恢复
            通过抢占进行恢复
            通过回滚进行恢复
            杀死进程恢复
    死锁避免
        单个资源的银行家算法
    破坏死锁
        破坏互斥条件
        破坏保持等待的条件
        破坏不可抢占条件
        破坏循环等待条件
    其他问题
        两阶段加锁
        通信死锁
        活锁
        饥饿
    总结
操作系统面试题
    解释一下什么是操作系统
    解释一下操作系统的主要目的是什么
    操作系统的种类有哪些
    操作系统结构
        单体系统
        分层系统
        微内核
        客户-服务器模式
    什么是按需分页
    多处理系统的优势
    什么是内核

- 什么是实时系统
- 什么是虚拟内存
- 什么是进程和进程表
- 什么是线程，线程和进程的区别
- 使用多线程的好处是什么
- 什么是 RR 调度算法
- 导致系统出现死锁的情况
- RAID 的不同级别
- 什么是 DMA
- 多线程编程的好处是什么
- 什么是设备驱动程序
- 进程间的通信方式
    - 通信概念
    - 解决方案
- 进程间状态模型
- 调度算法都有哪些
    - 批处理中的调度
        - 先来先服务
        - 最短作业优先
        - 最短剩余时间优先
    - 交互式系统中的调度
        - 轮询调度
        - 优先级调度
        - 最短进程优先
        - 彩票调度
        - 公平分享调度
- 页面置换算法都有哪些
- 影响调度程序的指标是什么
- 什么是僵尸进程

关于操作系统，你必须知道的名词

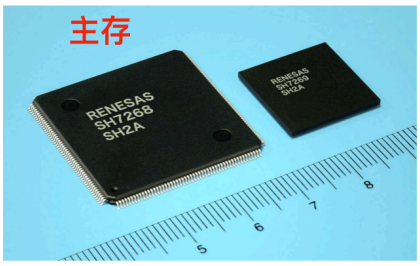
勘误



操作系统

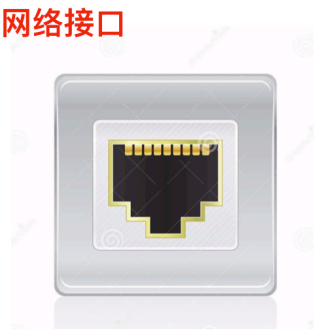
现代计算机系统由一个或多个处理器、主存、打印机、键盘、鼠标、显示器、网络接口以及各种输入/输出设备构成。







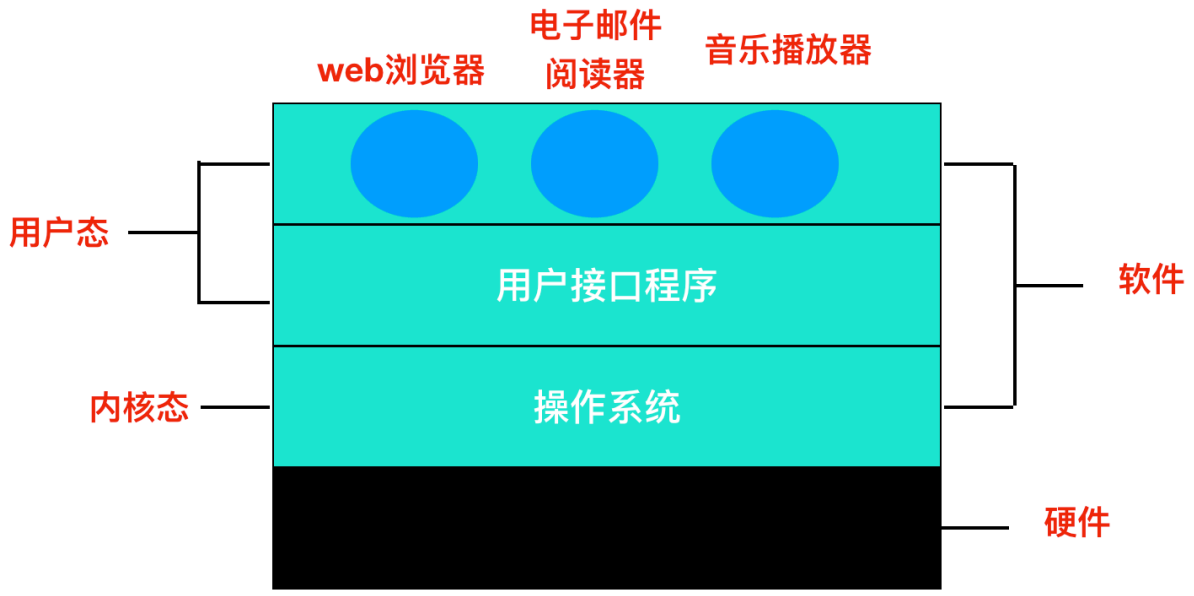






然而，程序员不会直接和这些硬件打交道，而且每位程序员不可能会掌握所有计算机系统的细节，这样我们就不用再编写代码了，所以在硬件的基础之上，计算机安装了一层软件，这层软件能够通过响应用户输入的指令达到控制硬件的效果，从而满足用户需求，这种软件称之为 `操作系统`，它的任务就是为用户程序提供一个更好、更简单、更清晰的计算机模型。

我们一般常见的操作系统主要有 **Windows、Linux、FreeBSD 或 OS X**，这种带有图形界面的操作系统被称为 `图形用户界面(Graphical User Interface, GUI)`，而基于文本、命令行的通常称为 `Shell`。下面是我们所要探讨的操作系统的部件



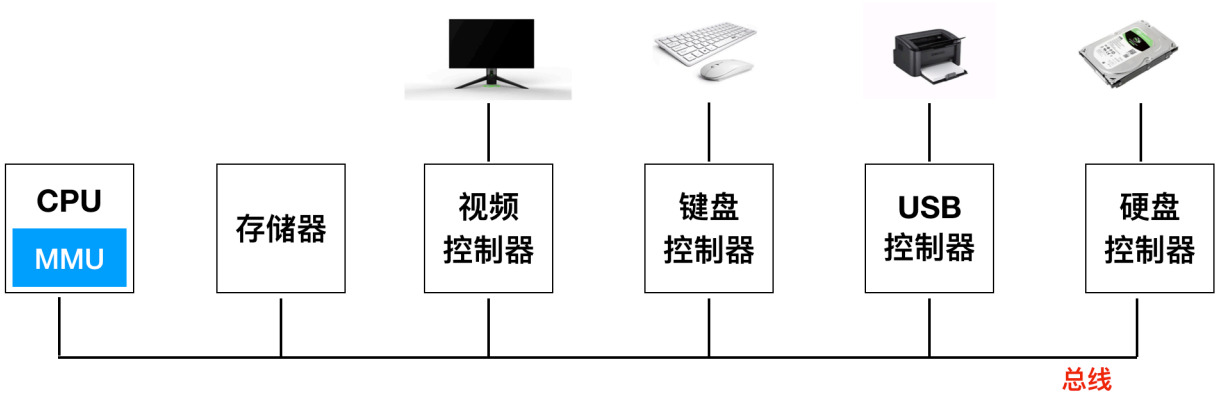
操作系统所处的位置

这是一个操作系统的简化图，最下面的是硬件，硬件包括芯片、电路板、磁盘、键盘、显示器等我们上面提到的设备，在硬件之上是软件。大部分计算机有两种运行模式：`内核态` 和 `用户态`，软件中最基础的部分是 `操作系统`，它运行在 `内核态` 中，内核态也称为 `管态` 和 `核心态`，它们都是操作系统的运行状态，只不过是不同的叫法而已。操作系统具有硬件的访问权，可以执行机器能够运行的任何指令。软件的其余部分运行在 `用户态` 下。

`用户接口程序(shell 或者 GUI)` 处于用户态中，并且它们位于用户态的最低层，允许用户运行其他程序，例如 Web 浏览器、电子邮件阅读器、音乐播放器等。而且，越靠近用户态的应用程序越容易编写，如果你不喜欢某个电子邮件阅读器你可以重新写一个或者换一个，但你不能自行写一个操作系统或者是中断处理程序。这个程序由硬件保护，防止外部对其进行修改。

计算机硬件简介

操作系统与运行操作系统的内核硬件关系密切。操作系统扩展了计算机指令集并管理计算机的资源。因此，操作系统因此必须足够了解硬件的运行，这里我们先简要介绍一下现代计算机中的计算机硬件。



简单个人计算机的组件

从概念上来看，一台简单的个人电脑可以被抽象为上面这种相似的模型，CPU、内存、I/O 设备都和总线串联起来并通过总线与其他设备进行通信。现代操作系统有着更为复杂的结构，会设计很多条总线，我们稍后会看到。暂时来讲，这个模型能够满足我们的讨论。

CPU

CPU 是计算机的大脑，它主要和内存进行交互，从内存中提取指令并执行它。一个 CPU 的执行周期是从内存中提取第一条指令、解码并决定它的类型和操作数，执行，然后再提取、解码执行后续的指令。重复该循环直到程序运行完毕。

每个 CPU 都有一组可以执行的特定指令集。因此，x86 的 CPU 不能执行 ARM 的程序并且 ARM 的 CPU 也不能执行 x86 的程序。由于访问内存获取执行或数据要比执行指令花费的时间长，因此所有的 CPU 内部都会包含一些 `寄存器` 来保存关键变量和临时结果。因此，在指令集中通常会有一些指令用于把关键字从内存中加载到寄存器中，以及把关键字从寄存器存入到内存中。还有一些其他的指令会把来自寄存器和内存的操作数进行组合，例如 add 操作就会把两个操作数相加并把结果保存到内存中。

除了用于保存变量和临时结果的通用寄存器外，大多数计算机还具有几个特殊的寄存器，这些寄存器对于程序员是可见的。其中之一就是 `程序计数器(program counter)`，程序计数器会指示下一条需要从内存提取指令的地址。提取指令后，程序计数器将更新为下一条需要提取的地址。

另一个寄存器是 `堆栈指针(stack pointer)`，它指向内存中当前栈的顶端。堆栈指针会包含输入过程中的有关参数、局部变量以及没有保存在寄存器中的临时变量。

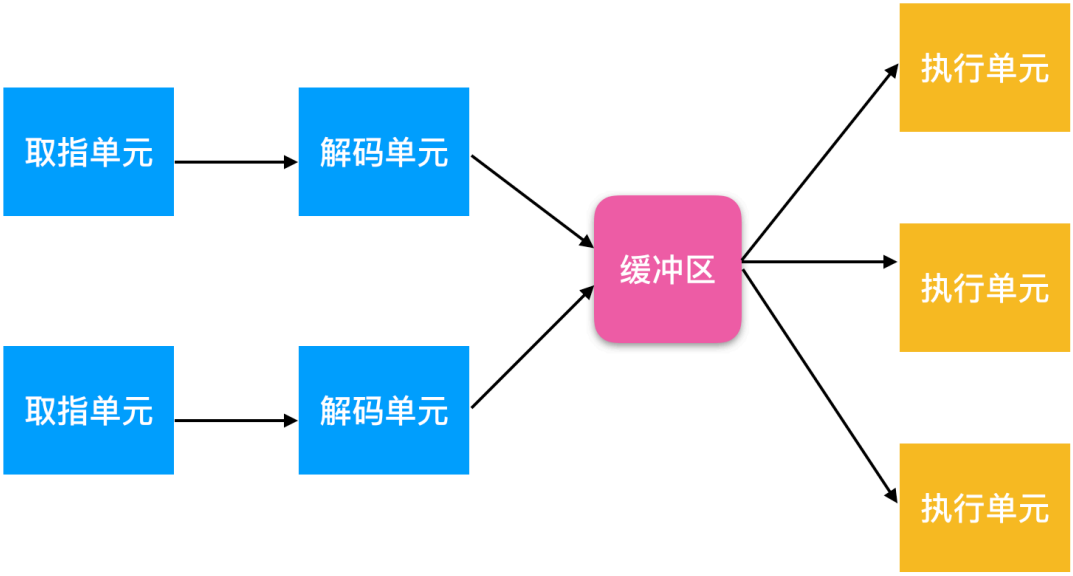
还有一个寄存器是 `PSW(Program Status Word)` 程序状态字寄存器，这个寄存器是由操作系统维护的8个字节(64位) long 类型的数据集合。它会跟踪当前系统的状态。除非发生系统结束，否则我们可以忽略 PSW 。用户程序通常可以读取整个PSW，但通常只能写入其某些字段。PSW 在系统调用和 I / O 中起着重要作用。

操作系统必须了解所有的寄存器。在 `时间多路复用(time multiplexing)` 的 CPU 中，操作系统往往停止运行一个程序转而运行另外一个。每次当操作系统停止运行一个程序时，操作系统会保存所有寄存器的值，以便于后续重新运行该程序。

为了提升性能， CPU 设计人员早就放弃了同时去读取、解码和执行一条简单的指令。许多现代的 CPU 都具有同时读取多条指令的机制。例如，一个 CPU 可能会有单独访问、解码和执行单元，所以，当 CPU 执行第 N 条指令时，还可以对 N + 1 条指令解码，还可以读取 N + 2 条指令。像这样的组织形式被称为 `流水线(pipeline)`，



比流水线更先进的设计是 `超标量(superscalar)` CPU，下面是超标量 CPU 的设计



在上面这个设计中，存在多个执行单元，例如，一个用来进行整数运算、一个用来浮点数运算、一个用来布尔运算。两个或者更多的指令被一次性取出、解码并放入缓冲区中，直至它们执行完毕。只要一个执行单元空闲，就会去检查缓冲区是否有可以执行的指令。如果有，就把指令从缓冲区中取出并执行。这种设计的含义是应用程序通常是无序执行的。在大多数情况下，硬件负责保证这种运算的结果与顺序执行指令时的结果相同。

除了用在嵌入式系统中非常简单的 CPU 之外，多数 CPU 都有 `两种模式` ，即前面已经提到的内核态和用户态。通常情况下， `PSW 寄存器` 中的一个二进制位会控制当前状态是内核态还是用户态。当运行在内核态时，CPU 能够执行任何指令集中的指令并且能够使用硬件的功能。在台式机和服务器上，操作系统通常以内核模式运行，从而可以访问完整的硬件。在大多数嵌入式系统中，一部分运行在内核态下，剩下的一部分运行在用户态下。

用户应用程序通常运行在用户态下，在用户态下，CPU 只能执行指令集中的一部分并且只能访问硬件的一部分功能。一般情况下，在用户态下，有关 I/O 和内存保护的所有指令是禁止执行的。当然，设置 PSW 模式的二进制位为内核态也是禁止的。

为了获取操作系统的服务，用户程序必须使用 `系统调用(system call)` ，系统调用会转换为内核态并且调用操作系统。 `TRAP` 指令用于把用户态切换为内核态并启用操作系统。当有关工作完成之后，在系统调用后面的指令会把控制权交给用户程序。我们会在后面探讨操作系统的调用细节。

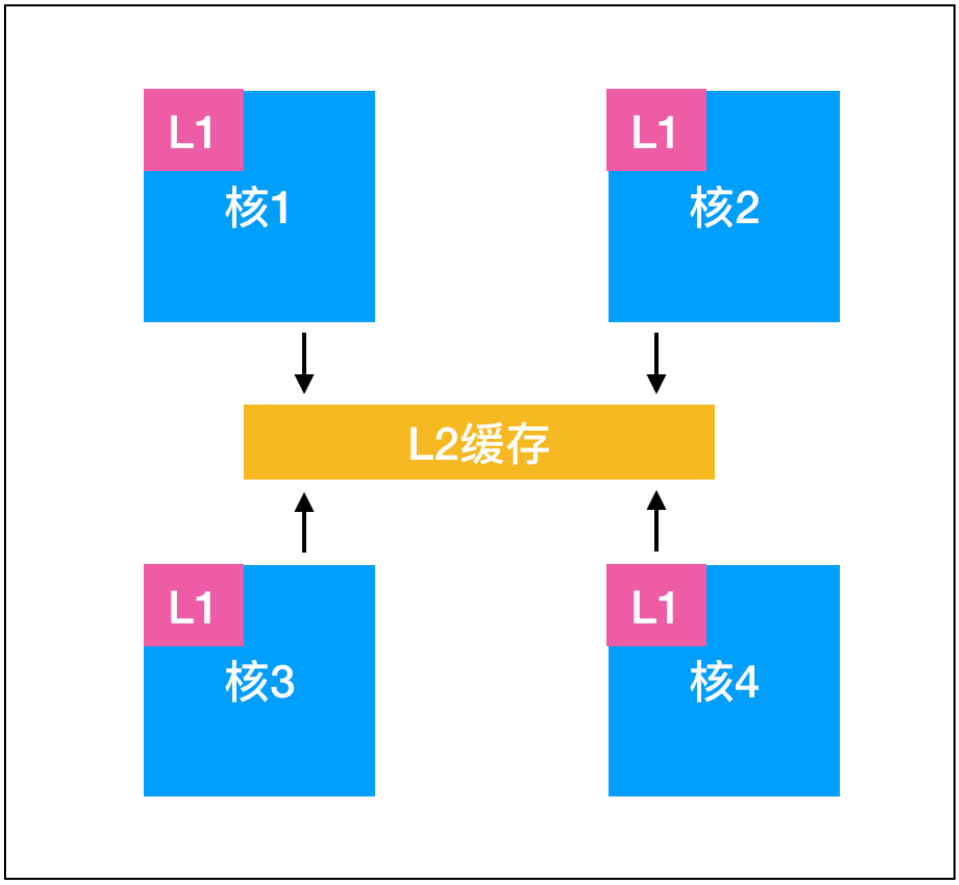
需要注意的是操作系统在进行系统调用时会存在陷阱。大部分的陷阱会导致硬件发出警告，比如说试图被零除或浮点下溢等你。在所有的情况下，操作系统都能得到控制权并决定如何处理异常情况。有时，由于出错的原因，程序不得不停止。

多线程和多核芯片

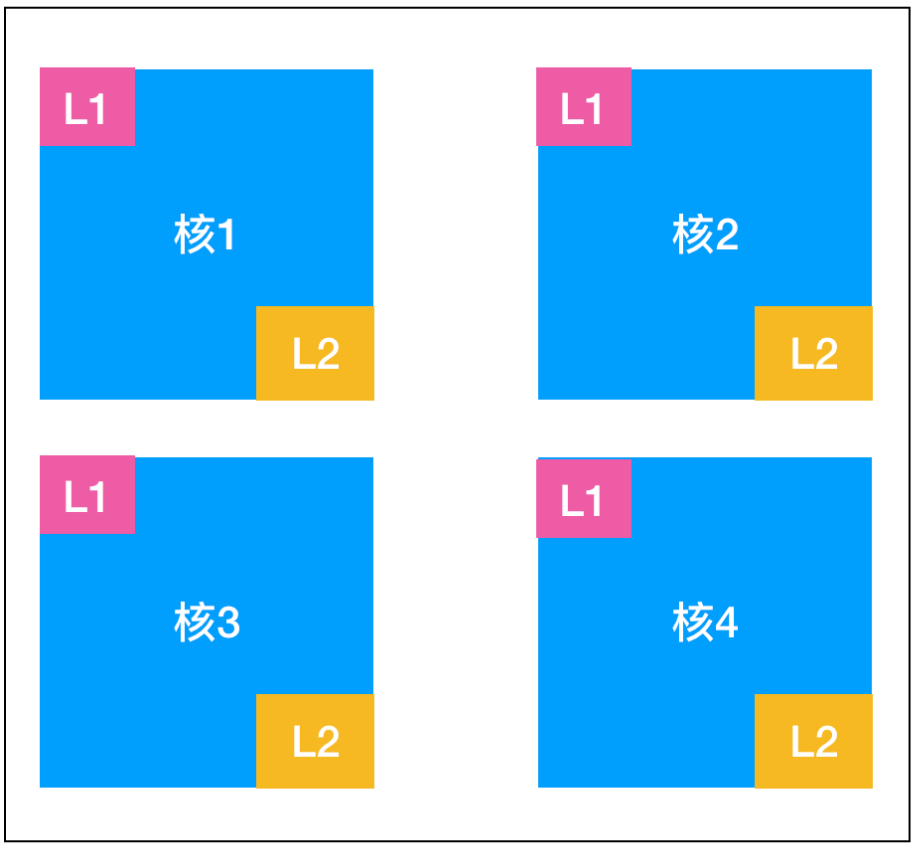
Intel Pentinum 4也就是奔腾处理器引入了被称为 `多线程(multithreading)` 或 `超线程(hyperthreading, Intel 公司的命名)` 的特性，x86 处理器和其他一些 CPU 芯片就是这样做的。包括 **SSPARC、Power5、Intel Xeon 和 Intel Core 系列** 。近似地说，多线程允许 CPU 保持两个不同的线程状态并且在 `纳秒级(nanosecond)` 的时间完成切换。线程是一种轻量级的进程，我们会在后面说到。例如，如果一个进程想要从内存中读取指令(这通常会经历几个时钟周期)，多线程 CPU 则可以切换至另一个线程。多线程不会提供真正的并行处理。在一个时刻只有一个进程在运行。

对于操作系统来讲，多线程是有意义的，因为每个线程对操作系统来说都像是一个单个的 CPU。比如一个有两个 CPU 的操作系统，并且每个 CPU 运行两个线程，那么这对于操作系统来说就可能是 4 个 CPU。

除了多线程之外，现在许多 CPU 芯片上都具有四个、八个或更多完整的处理器或内核。多核芯片在其上有效地承载了四个微型芯片，每个微型芯片都有自己的独立CPU。



带有共享 L2 缓存的 4 核芯片

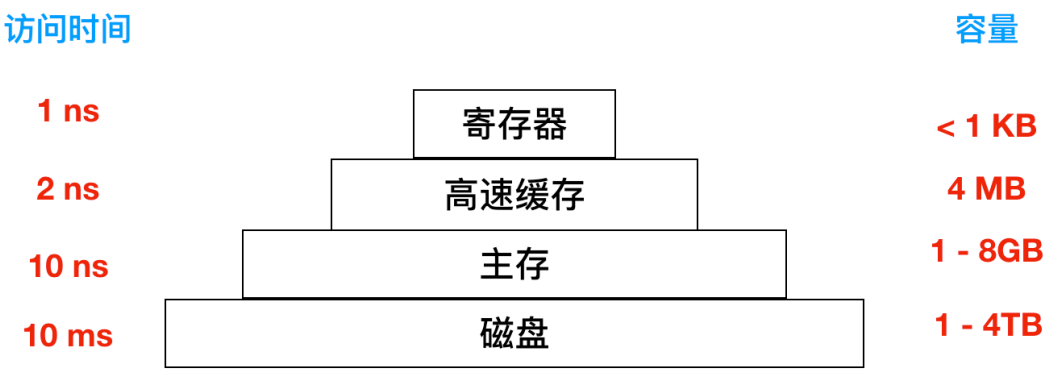


带有分离 L2 缓存的 4 核芯片

如果要说在绝对核心数量方面，没有什么能赢过现代 `GPU(Graphics Processing Unit)`，GPU 是指由成千上万个微核组成的处理器。它们擅长处理大量并行的简单计算。

内存

计算机中第二个主要的组件就是内存。理想情况下，内存应该非常快速(比执行一条指令要快，从而不会拖慢 CPU 执行效率)，而且足够大且便宜，但是目前的技术手段无法满足三者的需求。于是采用了不同的处理方式，存储器系统采用一种分层次的结构



存储层次结构

顶层的存储器速度最高，但是容量最小，成本非常高，层级结构越向下，其访问效率越慢，容量越大，但是造价也就越便宜。

寄存器

存储器的顶层是 CPU 中的 `寄存器` ，它们用和 CPU 一样的材料制成，所以和 CPU 一样快。程序必须在软件中自行管理这些寄存器（即决定如何使用它们）

高速缓存

位于寄存器下面的是 `高速缓存` ，它多数由硬件控制。主存被分割成 `高速缓存行(cache lines)` 为 64 字节，内存地址的 0 - 63 对应高速缓存行 0 ，地址 64 - 127 对应高速缓存行的 1，等等。使用最频繁的高速缓存行保存在位于 CPU 内部或非常靠近 CPU 的高速缓存中。当应用程序需要从内存中读取关键词的时候，高速缓存的硬件会检查所需要的高速缓存行是否在高速缓存中。如果在的话，那么这就是 `高速缓存命中(cache hit)` 。高速缓存满足了该请求，并且没有通过总线将内存请求发送到主内存。高速缓存命中通常需要花费两个时钟周期。缓存未命中需要从内存中提取，这会消耗大量的时间。高速缓存行会限制容量的大小因为它的造价非常昂贵。有一些机器会有两个或者三个高速缓存级别，每一级高速缓存比前一级慢且容量更大。

缓存在计算机很多领域都扮演了非常重要的角色，不仅仅是 RAM 缓存行。

> 随机存储器（RAM）： 内存中最重要的一种，表示既可以从中读取数据，也可以写入数据。当机器关闭时，内存中的信息会 `丢失` 。

大量的可用资源被划分为小的部分，这些可用资源的一部分会获得比其他资源更频繁的使用权，缓存经常用来提升性能。操作系统无时无刻的不在使用缓存。例如，大多数操作系统在主机内存中保留（部分）频繁使用的文件，以避免重复从磁盘重复获取。举个例子，类似于 `/home/ast/projects/minix3/src/kernel/clock.c` 这样的场路径名转换成的文件所在磁盘地址的结果也可以保存缓存中，以避免重复寻址。另外，当一个 Web 页面(URL) 的地址转换为网络地址(IP 地址)后，这个转换结果也可以缓存起来供将来使用。

在任何缓存系统中，都会有下面这几个噬需解决的问题

- 何时把新的内容放进缓存
- 把新的内容应该放在缓存的哪一行
- 在需要空闲空间时，应该把哪块内容从缓存中移除
- 应该把移除的内容放在某个较大存储器的何处

并不是每个问题都与每种缓存情况有关。对于 CPU 缓存中的主存缓存行，当有缓存未命中时，就会调入新的内容。通常通过所引用内存地址的高位计算应该使用的缓存行。

缓存是解决问题的一种好的方式，所以现代 CPU 设计了两种缓存。第一级缓存或者说是 `L1 cache` 总是位于 CPU 内部，**用来将已解码的指令调入 CPU 的执行引擎**。对于那些频繁使用的关键字，多数芯片有第二个 L1 cache 。典型的 L1 cache 的大小为 16 KB。另外，往往还设有二级缓存，也就是 `L2 cache` ，用来存放最近使用过的关键字，一般是兆字节为单位。L1 cache 和 L2 cache 最大的不同在于是否存在延迟。访问 L1 cache 没有任何的延迟，然而访问 L2 cache 会有 1 - 2 个时钟周期的延后。

> 什么是时钟周期？计算机处理器或 CPU 的速度由时钟周期来确定，该时钟周期是振荡器两个脉冲之间的时间量。一般而言，每秒脉冲数越高，计算机处理器处理信息的速度就越快。 时钟速度以 Hz 为单位测量，通常为兆赫（MHz）或千兆赫（GHz）。 例如，一个4 GHz处理器每秒执行4,000,000,000个时钟周期。
>
> 计算机处理器可以在每个时钟周期执行一条或多条指令，这具体取决于处理器的类型。 早期的计算机处理器和较慢的 CPU 在每个时钟周期只能执行一条指令，而现代处理器在每个时钟周期可以执行多条指令。

主存

在上面的层次结构中再下一层是 `主存`，这是内存系统的主力军，主存通常叫做 `RAM(Random Access Memory)`，由于 1950 年代和 1960 年代的计算机使用微小的可磁化铁氧体磁芯作为主存储器，因此旧时有时将其称为核心存储器。所有不能再高速缓存中得到满足的内存访问请求都会转往主存中。

除了主存之外，许多计算机还具有少量的非易失性随机存取存储器。它们与 RAM 不同，在电源断电后，非易失性随机访问存储器并不会丢失内容。 `ROM(Read Only Memory)` 中的内容一旦存储后就不会再被修改。它非常快而且便宜。（如果有人问你，有没有什么又快又便宜的内存设备，那就是 ROM 了）在计算机中，用于启动计算机的引导加载模块（也就是 bootstrap ）就存放在 ROM 中。另外，一些 I/O 卡也采用 ROM 处理底层设备控制。

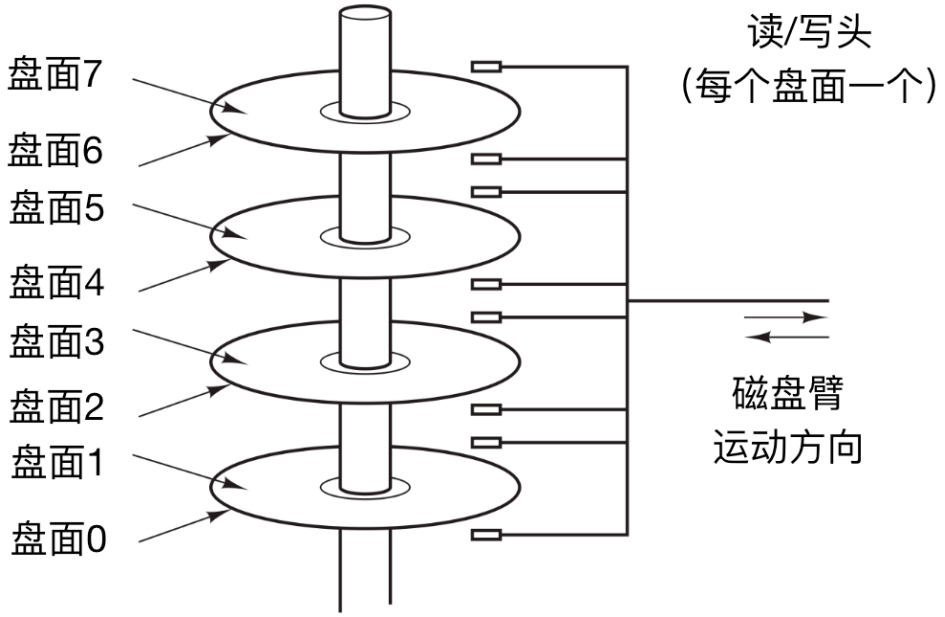
`EEPROM(Electrically Erasable PROM,)` 和 `闪存(flash memory)` 也是非易失性的，但是与 ROM 相反，它们可以擦除和重写。不过重写它们需要比写入 RAM 更多的时间，所以它们的使用方式与 ROM 相同，但是与 ROM 不同的是他们可以通过重写字段来纠正程序中出现的错误。

闪存也通常用来作为便携性的存储媒介。闪存是数码相机中的胶卷，是便携式音乐播放器的磁盘。闪存的速度介于 RAM 和磁盘之间。另外，与磁盘存储器不同的是，如果闪存擦除的次数太多，会出现磨损。

还有一类是 CMOS，它是易失性的。许多计算机都会使用 CMOS 存储器保持当前时间和日期。

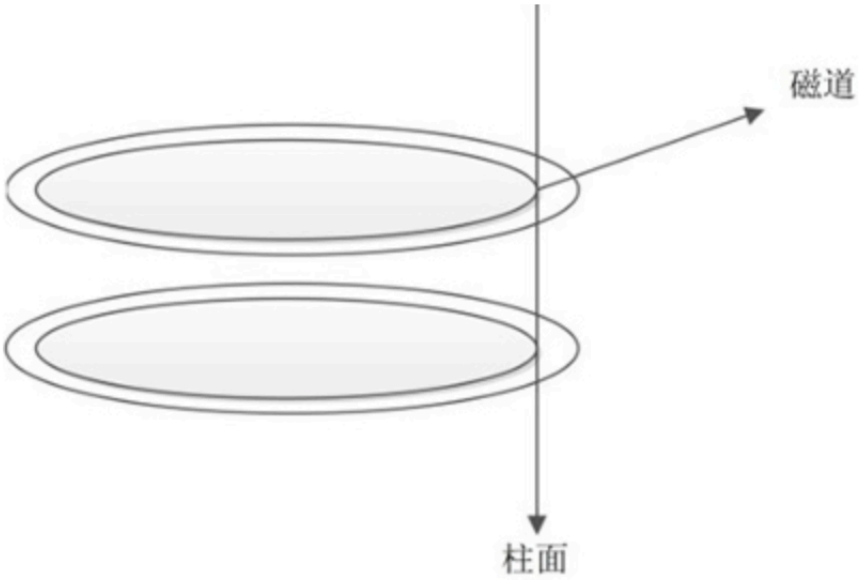
磁盘

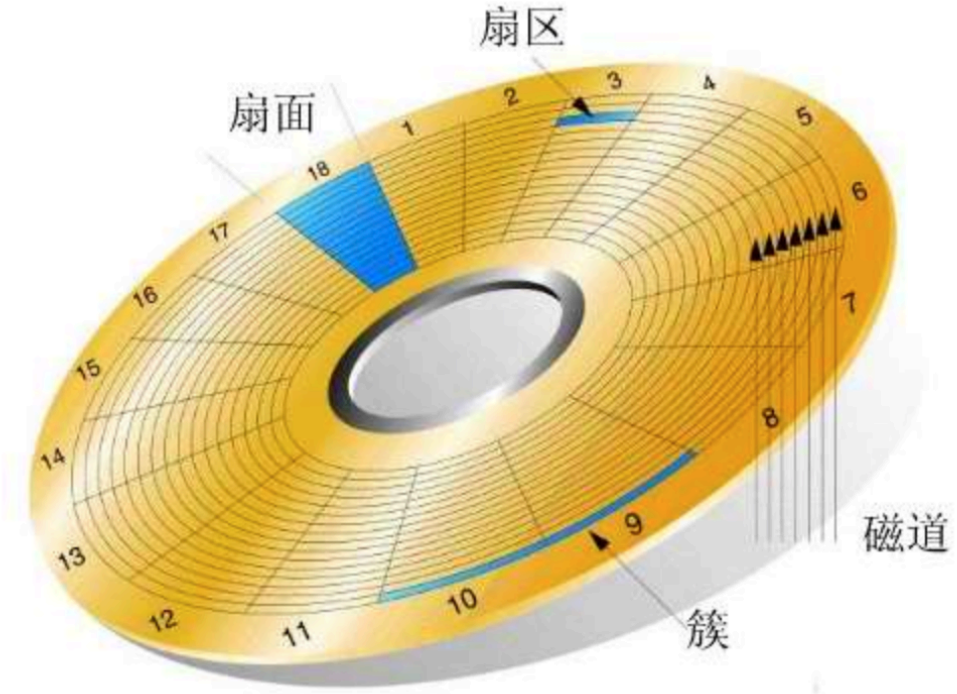
下一个层次是 `磁盘(硬盘)`，磁盘同 RAM 相比，每个二进制位的成本低了两个数量级，而且经常也有两个数量级大的容量。磁盘唯一的问题是随机访问数据时间大约慢了三个数量级。磁盘访问慢的原因是因为磁盘的构造不同



磁盘驱动器的构造

磁盘是一种机械装置，在一个磁盘中有一个或多个金属盘片，它们以 5400rpm、7200rpm、10800rpm 或更高的速度旋转。从边缘开始有一个机械臂悬横在盘面上，这类似于老式播放塑料唱片 33 转唱机上的拾音臂。信息会写在磁盘一系列的同心圆上。在任意一个给定臂的位置，每个磁头可以读取一段环形区域，称为 `磁道(track)` 。把一个给定臂的位置上的所有磁道合并起来，组成了一个 `柱面(cylinder)` 。

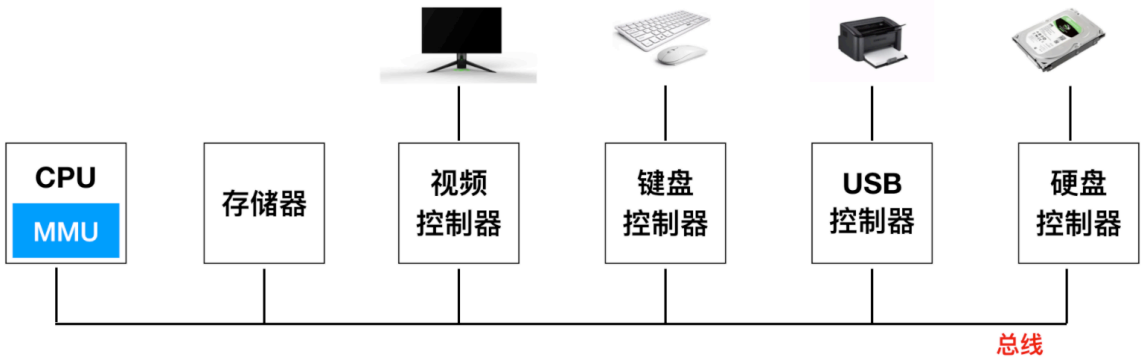




每个磁道划分若干扇区，扇区的值是 512 字节。在现代磁盘中，较外部的柱面比较内部的柱面有更多的扇区。机械臂从一个柱面移动到相邻的柱面大约需要 1ms。而随机移到一个柱面的典型时间为 5ms 至 10ms，具体情况以驱动器为准。一旦磁臂到达正确的磁道上，驱动器必须等待所需的扇区旋转到磁头之下，就开始读写，低端硬盘的速率是 `50MB/s`，而高速磁盘的速率是 `160MB/s`。

> 需要注意，`固态硬盘(Solid State Disk, SSD)` 不是磁盘，固态硬盘并没有可以移动的部分，外形也不像唱片，并且数据是存储在 `存储器（闪存）` 中，与磁盘唯一的相似之处就是它也存储了大量即使在电源关闭也不会丢失的数据。

许多计算机支持一种著名的 `虚拟内存` 机制，这种机制使得期望运行的存储空间大于实际的物理存储空间。其方法是将程序放在磁盘上，而将主存作为一部分缓存，用来保存最频繁使用的部分程序，这种机制需要快速映像内存地址，用来把程序生成的地址转换为有关字节在 RAM 中的物理地址。这种映像由 CPU 中的一个称为 `存储器管理单元(Memory Management Unit, MMU)` 的部件来完成。



缓存和 MMU 的出现是对系统的性能有很重要的影响，在多道程序系统中，从一个程序切换到另一个程序的机制称为 `上下文切换(context switch)`，对来自缓存中的资源进行修改并把其写回磁盘是很有必要的。

I/O 设备

CPU 和存储器不是操作系统需要管理的全部，`I/O` 设备也与操作系统关系密切。可以参考上面这个图片，I/O 设备一般包括两个部分：设备控制器和设备本身。控制器本身是一块芯片或者一组芯片，它能够控制物理设备。它能够接收操作系统的指令，例如，从设备中读取数据并完成数据的处理。

在许多情况下，实际控制设备的过程是非常复杂而且存在诸多细节。因此控制器的工作就是为操作系统提供一个更简单（但仍然非常复杂）的接口。也就是屏蔽物理细节。**任何复杂的东西都可以加一层代理来解决，这是计算机或者人类社会很普世的一个解决方案**

I/O 设备另一部分是设备本身，设备本身有一个相对简单的接口，这是因为接口既不能做很多工作，而且也已经被标准化了。例如，标准化后任何一个 SATA 磁盘控制器就可以适配任意一种 SATA 磁盘，所以标准化是必要的。`ATA` 代表 `高级技术附件(AT Attachment)`，而 SATA 表示 `串行高级技术附件(Serial ATA)`。

> AT 是啥？它是 IBM 公司的第二代个人计算机的 `高级` 技术成果，使用 1984 年推出的 6MHz 80286 处理器，这个处理器是当时最强大的。

像是高级这种词汇应该慎用，否则 20 年后再回首很可能会被无情打脸。

现在 SATA 是很多计算机的标准硬盘接口。由于实际的设备接口隐藏在控制器中，所以操作系统看到的是对控制器的接口，这个接口和设备接口有很大区别。

每种类型的设备控制器都是不同的，所以需要不同的软件进行控制。专门与控制器进行信息交流，发出命令处理指令接收响应的软件，称为 `设备驱动程序(device driver)`。每个控制器厂家都应该针对不同的操作系统提供不同的设备驱动程序。

为了使设备驱动程序能够工作，必须把它安装在操作系统中，这样能够使它在内核态中运行。要将设备驱动程序装入操作系统，一般有三个途径

- 第一个途径是将内核与设备启动程序重新连接，然后重启系统。这是 `UNIX` 系统采用的工作方式
- 第二个途径是在一个操作系统文件中设置一个入口，通知该文件需要一个设备驱动程序，然后重新启动系统。在重新系统时，操作系统回寻找有关的设备启动程序并把它装载，这是 `Windows` 采用的工作方式
- 第三个途径是操作系统能够在运行时接收新的设备驱动程序并立刻安装，无需重启操作系统，这种方式采用的少，但是正变得普及起来。热插拔设备，比如 USB 和 IEEE 1394 都需要动态可装载的设备驱动程序。

每个设备控制器都有少量用于通信的寄存器，例如，一个最小的磁盘控制器也会有用于指定磁盘地址、内存地址、扇区计数的寄存器。要激活控制器，设备驱动程序回从操作系统获取一条指令，然后翻译成对应的值，并写入设备寄存器中，所有设备寄存器的结合构成了 `I/O 端口空间`。

在一些计算机中，设备寄存器会被映射到操作系统的可用地址空间，使他们能够向内存一样完成读写操作。在这种计算机中，不需要专门的 I/O 指令，用户程序可以被硬件阻挡在外，防止其接触这些存储器地址（例如，采用基址寄存器和变址寄存器）。在另一些计算机中，设备寄存器被放入一个专门的 I/O 端口空间，每个寄存器都有一个端口地址。在这些计算机中，特殊的 `IN` 和 `OUT` 指令会在内核态下启用，它能够允许设备驱动程序和寄存器进行读写。前面第一种方式会限制特殊的 I/O 指令但是允许一些地址空间；后者不需要地址空间但是需要特殊的指令，这两种应用都很广泛。

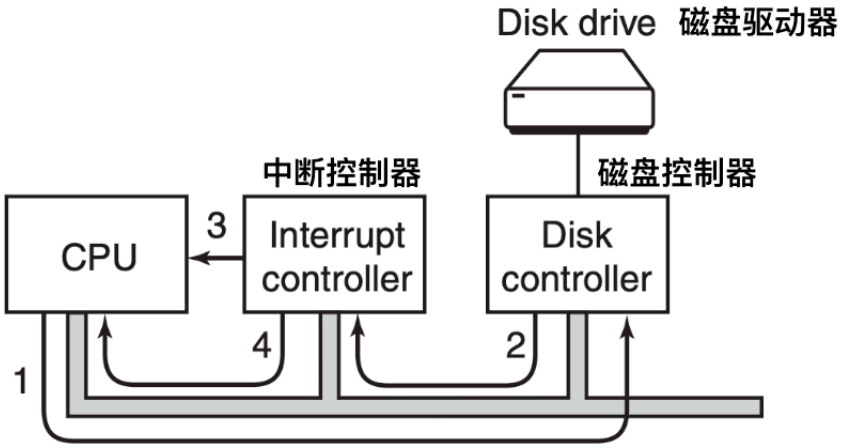
实现输入和输出的方式有三种。

- 在最简单的方式中，用户程序会发起系统调用，内核会将其转换为相应驱动程序的程序调用，然后设备驱动程序启动 I/O 并循环检查该设备，看该设备是否完成了工作（一般会有一些二进制位用来指示设备仍在忙碌中）。当 I/O 调用完成后，设备驱动程序把数据送到指定的地方并返回。然后操

作系统会将控制权交给调用者。这种方式称为 `忙等待(busy waiting)`，这种方式的缺点是要一直占据 CPU，CPU 会一直轮询 I/O 设备直到 I/O 操作完成。

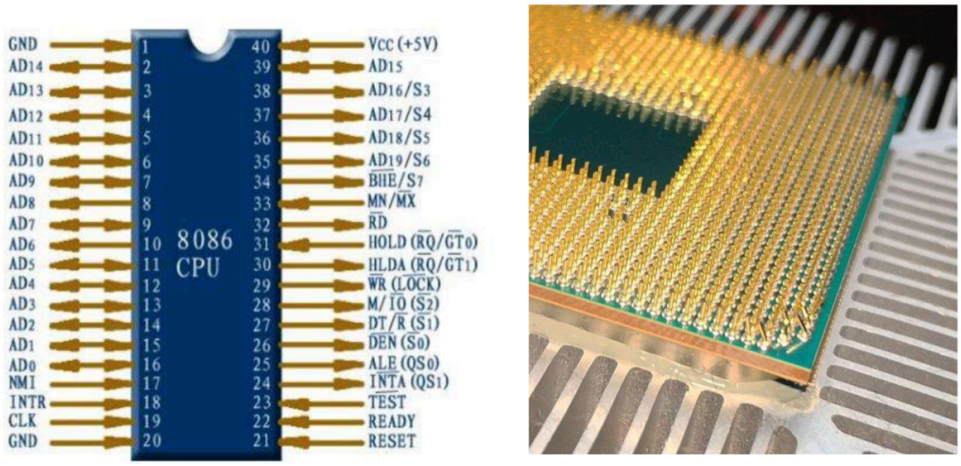
- 第二种方式是设备驱动程序启动设备并且让该设备在操作完成时发生中断。设备驱动程序在这个时刻返回。操作系统接着在需要时阻塞调用者并安排其他工作进行。当设备驱动程序检测到该设备操作完成时，它发出一个 `中断` 通知操作完成。

在操作系统中，中断是非常重要的，所以这需要更加细致的讨论一下。



启动设备并发出中断的过程

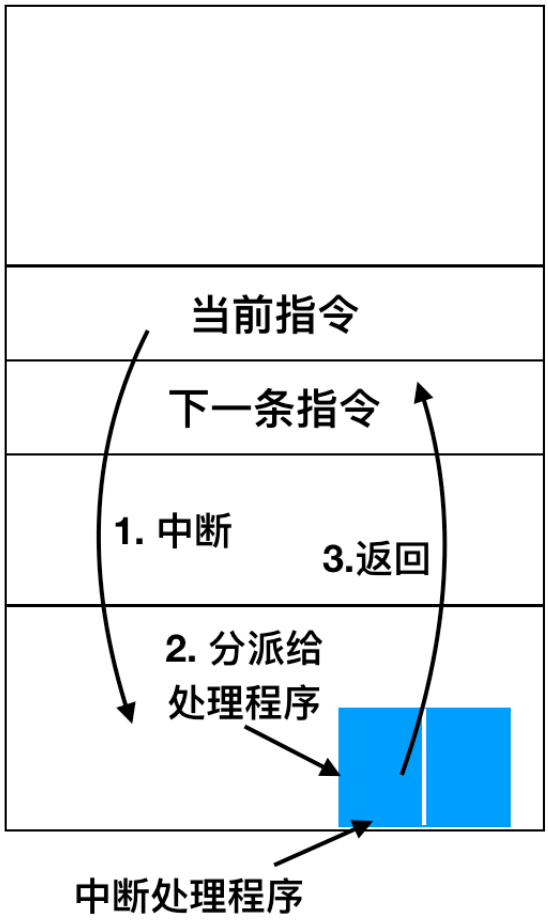
如上图所示，这是一个三步的 I/O 过程，第一步，设备驱动程序会通过写入设备寄存器告诉控制器应该做什么。然后，控制器启动设备。当控制器完成读取或写入被告知需要传输的字节后，它会在步骤 2 中使用某些总线向中断控制器发送信号。如果中断控制器准备好了接收中断信号（如果正忙于一个优先级较高的中断，则可能不会接收），那么它就会在 CPU 的一个引脚上面声明。这就是步骤3



CPU 引脚剖面图

在第四步中，中断控制器把该设备的编号放在总线上，这样 CPU 可以读取总线，并且知道哪个设备完成了操作（可能同时有多个设备同时运行）。

一旦 CPU 决定去实施中断后，程序计数器和 PSW 就会被压入到当前堆栈中并且 CPU 会切换到内核态。设备编号可以作为内存的一个引用，用来寻找该设备中断处理程序的地址。这部分内存称作 `中断向量(interrupt vector)` 。一旦中断处理程序（中断设备的设备驱动程序的一部分）开始后，它会移除栈中的程序计数器和 PSW 寄存器，并把它们进行保存，然后查询设备的状态。在中断处理程序全部完成后，它会返回到先前用户程序尚未执行的第一条指令，这个过程如下



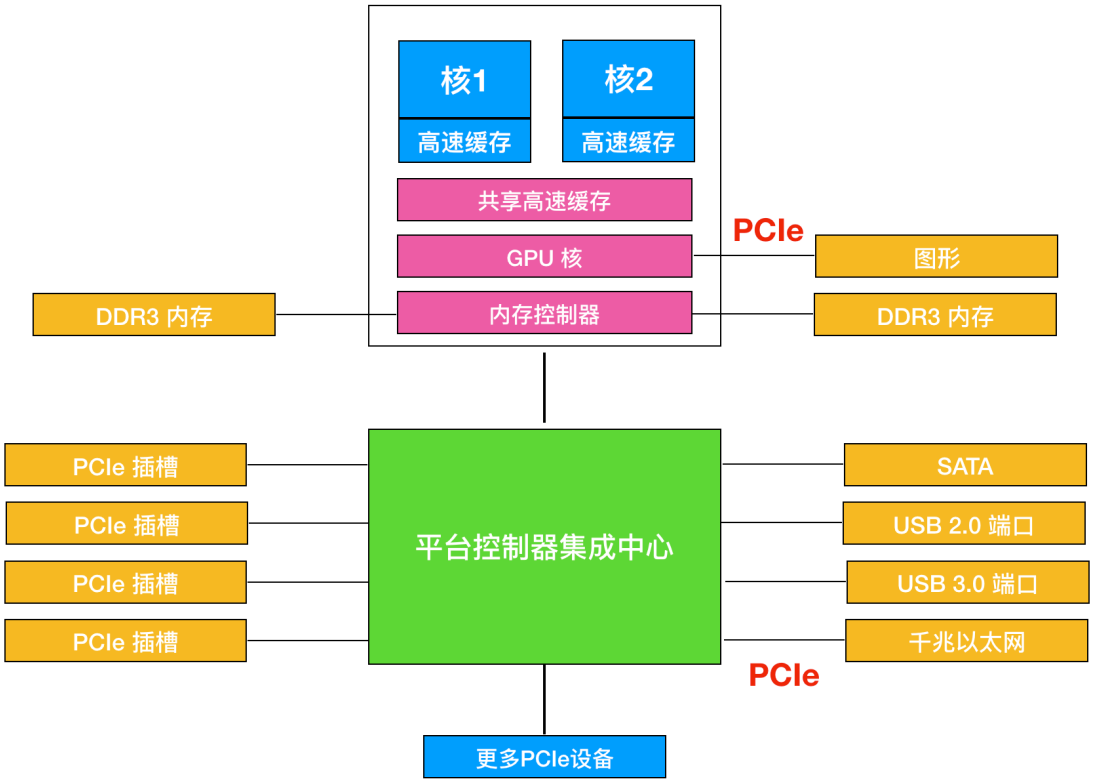
中断过程

- 实现 I/O 的第三种方式是使用特殊的硬件： `直接存储器访问(Direct Memory Access, DMA)` 芯片。它可以控制内存和某些控制器之间的位流，而无需 CPU 的干预。CPU 会对 DMA 芯片进行设置，说明需要传送的字节数，有关的设备和内存地址以及操作方向。当 DMA 芯片完成后，会造成中断，中断过程就像上面描述的那样。我们会在后面具体讨论中断过程

当另一个中断处理程序正在运行时，中断可能（并且经常）发生在不合宜的时间。 因此，CPU 可以禁用中断，并且可以在之后重启中断。在 CPU 关闭中断后，任何已经发出中断的设备，可以继续保持其中断信号处理，但是 CPU 不会中断，直至中断再次启用为止。如果在关闭中断时，已经有多个设备发出了中断信号，中断控制器将决定优先处理哪个中断，通常这取决于事先赋予每个设备的优先级，最高优先级的设备优先赢得中断权，其他设备则必须等待。

总线

上面的结构（简单个人计算机的组件图）在小型计算机已经使用了多年，并用在早期的 IBM PC 中。然而，随着处理器核内存变得越来越快，单个总线处理所有请求的能力也达到了上线，其中也包括 IBM PC 总线。必须放弃使用这种模式。其结果导致了其他总线的出现，它们处理 I/O 设备以及 CPU 到存储器的速度都更快。这种演变的结果导致了下面这种结构的出现。



一个大型的 x86 系统的结构

上图中的 x86 系统包含很多总线，**高速缓存、内存、PCIe、PCI、USB、SATA 和 DMI**，每条总线都有不同的传输速率和功能。操作系统必须了解所有的总线配置和管理。其中最主要的总线是 `PCIe(Peripheral Component Interconnect Express)` 总线。

Intel 发明的 PCIe 总线也是作为之前古老的 PCI 总线的继承者，而古老的 PCI 总线也是为了取代古董级别的 `ISA(Industry Standard Architecture)` 总线而设立的。数十 Gb/s 的传输能力使得 PCIe 比它的前身快很多，而且它们本质上也十分不同。直到发明 PCIe 的 2004 年，大多数总线都是并行且共享的。`共享总线架构(shared bus architeture)` 表示多个设备使用一些相同的电线传输数据。因此，当多个设备同时发送数据时，此时你需要一个决策者来决定谁能够使用总线。而 PCIe 则不一样，它使用专门的端到端链路。传统 PCI 中使用的 `并行总线架构(parallel bus architecture)` 表示通过多条电线发送相同的数据字。例如，在传统的 PCI 总线上，一个 32 位数据通过 32 条并行的电线发送。而 PCIe 则不同，它选用了 `串行总线架构(serial bus architecture)` ，并通过单个连接（称为通道）发送消息中的所有比特数据，就像网络数据包一样。这样做会简化很多，因为不再确保所有 32 位数据在同一时刻准确到达相同的目的地。通过将多个数据通路并行起来，并行性仍可以有效利用。例如，可以使用 32 条数据通道并行传输 32 条消息。

在上图结构中，CPU 通过 DDR3 总线与内存对话，通过 PCIe 总线与外围图形设备 （GPU）对话，通过 `DMI(Direct Media Interface)` 总线经集成中心与所有其他设备对话。而集成控制中心通过串行总线与 USB 设备对话，通过 SATA 总线与硬盘和 DVD 驱动器对话，通过 PCIe 传输以太网络帧。

不仅如此，每一个核

`USB(Univversal Serial Bus)` 是用来将所有慢速 I/O 设备（比如键盘和鼠标）与计算机相连的设备。USB 1.0 可以处理总计 12 Mb/s 的负载，而 USB 2.0 将总线速度提高到 480Mb/s ，而 USB 3.0 能达到不小于 5Gb/s 的速率。所有的 USB 设备都可以直接连接到计算机并能够立刻开始工作，而不像之前那样要求重启计算机。

`SCSI(Small Computer System Interface)` 总线是一种高速总线，用在高速硬盘，扫描仪和其他需要较大带宽的设备上。现在，它们主要用在服务器和工作站中，速度可以达到 640MB/s 。

计算机启动过程

那么有了上面一些硬件再加上操作系统的支持，我们的计算机就可以开始工作了，那么计算机的启动过程是怎样的呢？下面只是一个简要版的启动过程

在每台计算机上有一块双亲板，也就是母板，母板也就是主板，它是计算机最基本也就是最重要的部件之一。主板一般为矩形电路板，上面安装了组成计算机的主要电路系统，一般有 BIOS 芯片、I/O 控制芯片、键盘和面板控制开关接口、指示灯插接件、扩充插槽、主板及插卡的直流电源供电接插件等元件。

在母板上有一个称为 `基本输入输出系统(Basic Input Output System, BIOS)` 的程序。在 BIOS 内有底层 I/O 软件，包括读键盘、写屏幕、磁盘I/O 以及其他过程。如今，它被保存在闪存中，它是非易失性的，但是当BIOS 中发现错误时，可以由操作系统进行更新。

在计算机 `启动(booted)` 时，BIOS 开启，它会首先检查所安装的 RAM 的数量，键盘和其他基础设备是否已安装并且正常响应。接着，它开始扫描 PCIe 和 PCI 总线并找出连在上面的所有设备。即插即用的设备也会被记录下来。如果现有的设备和系统上一次启动时的设备不同，则新的设备将被重新配置。

蓝后，BIOS 通过尝试存储在 `CMOS` 存储器中的设备清单尝试启动设备

> CMOS是 `Complementary Metal Oxide Semiconductor（互补金属氧化物半导体）` 的缩写。它是指制造大规模集成电路芯片用的一种技术或用这种技术制造出来的芯片，是电脑主板上的一块可读写的 `RAM` 芯片。因为可读写的特性，所以在电脑主板上用来保存 BIOS 设置完电脑硬件参数后的数据，这个芯片仅仅是用来存放数据的。
>
> 而对 BIOS 中各项参数的设定要通过专门的程序。BIOS 设置程序一般都被厂商整合在芯片中，在开机时通过特定的按键就可进入 BIOS 设置程序，方便地对系统进行设置。因此 BIOS 设置有时也被叫做 CMOS 设置。

用户可以在系统启动后进入一个 BIOS 配置程序，对设备清单进行修改。然后，判断是否能够从外部 `CD-ROM` 和 USB 驱动程序启动，如果启动失败的话（也就是没有），系统将从硬盘启动，boots 设备中的第一个扇区被读入内存并执行。该扇区包含一个程序，该程序通常在引导扇区末尾检查分区表以确定哪个分区处于活动状态。然后从该分区读入第二个启动加载程序，该加载器从活动分区中读取操作系统并启动它。

然后操作系统会询问 BIOS 获取配置信息。对于每个设备来说，会检查是否有设备驱动程序。如果没有，则会向用户询问是否需要插入 `CD-ROM` 驱动（由设备制造商提供）或者从 Internet 上下载。一旦有了设备驱动程序，操作系统会把它们加载到内核中，然后初始化表，创建所需的后台进程，并启动登录程序或GUI。

操作系统博物馆

操作系统已经存在了大半个世纪，在这段时期内，出现了各种类型的操作系统，但并不是所有的操作系统都很出名，下面就罗列一些比较出名的操作系统

大型机操作系统

高端一些的操作系统是大型机操作系统，这些大型操作系统可在大型公司的数据中心找到。这些计算机的 I/O 容量与个人计算机不同。一个大型计算机有 1000 个磁盘和数百万 G 字节的容量是很正常，如果有这样一台个人计算机朋友会很羡慕。大型机也在高端 Web 服务器、大型电子商务服务站点上。

服务器操作系统

下一个层次是服务器操作系统。它们运行在服务器上，服务器可以是大型个人计算机、工作站甚至是大型机。它们通过网络为若干用户服务，并且允许用户共享硬件和软件资源。服务器可提供打印服务、文件服务或 Web 服务。Internet 服务商运行着许多台服务器机器，为用户提供支持，使 Web 站点保存 Web 页面并处理进来的请求。典型的服务器操作系统有 Solaris、FreeBSD、Linux 和 Windows Server 201x

多处理器操作系统

获得大型计算能力的一种越来越普遍的方式是将多个 CPU 连接到一个系统中。依据它们连接方式和共享方式的不同，这些系统称为并行计算机，多计算机或多处理器。他们需要专门的操作系统，不过通常采用的操作系统是配有通信、连接和一致性等专门功能的服务器操作系统的变体。

个人计算机中近来出现了多核芯片，所以常规的台式机和笔记本电脑操作系统也开始与小规模多处理器打交道，而核的数量正在与时俱进。许多主流操作系统比如 Windows 和 Linux 都可以运行在多核处理器上。

个人计算机系统

接下来一类是个人计算机操作系统。现代个人计算机操作系统支持多道处理程序。在启动时，通常有几十个程序开始运行，它们的功能是为单个用户提供良好的支持。这类系统广泛用于字处理、电子表格、游戏和 Internet 访问。常见的例子是 Linux、FreeBSD、Windows 7、Windows 8 和苹果公司的 OS X 。

掌上计算机操作系统

随着硬件越来越小化，我们看到了平板电脑、智能手机和其他掌上计算机系统。掌上计算机或者 `PDA(Personal Digital Assistant)，个人数字助理` 是一种可以握在手中操作的小型计算机。这部分市场已经被谷歌的 `Android` 系统和苹果的 `IOS` 主导。

嵌入式操作系统

嵌入式操作系统用来控制设备的计算机中运行，这种设备不是一般意义上的计算机，并且不允许用户安装软件。典型的例子有微波炉、汽车、DVD 刻录机、移动电话以及 MP3 播放器一类的设备。所有的软件都运行在 ROM 中，这意味着应用程序之间不存在保护，从而获得某种简化。主要的嵌入式系统有 Linux、QNX 和 VxWorks

传感器节点操作系统

有许多用途需要配置微小传感器节点网络。这些节点是一种可以彼此通信并且使用无线通信基站的微型计算机。这类传感器网络可以用于建筑物周边保护、国土边界保卫、森林火灾探测、气象预测用的温度和降水测量等。

每个传感器节点是一个配有 CPU、RAM、ROM 以及一个或多个环境传感器的实实在在的计算机。节点上运行一个小型但是真是的操作系统，通常这个操作系统是事件驱动的，可以响应外部事件。

实时操作系统

另一类操作系统是实时操作系统，这些系统的特征是将时间作为关键参数。例如，在工业过程控制系统中，工厂中的实时计算机必须收集生产过程的数据并用有关数据控制机器。如果某个动作必须要在规定的时刻发生，这就是 `硬实时系统` 。可以在工业控制、民用航空、军事以及类似应用中看到很多这样的系统。另一类系统是 `软实时系统` ，在这种系统中，虽然不希望偶尔违反最终时限，但仍可以接受，并不会引起任何永久性损害。数字音频或多媒体系统就是这类系统。智能手机也是软实时系统。

智能卡操作系统

最小的操作系统运行在智能卡上。智能卡是一种包含一块 CPU 芯片的信用卡。它有非常严格的运行能耗和存储空间的限制。有些卡具有单项功能，如电子支付；有些智能卡是面向 Java 的。这意味着在智能卡的 ROM 中有一个 Java 虚拟机（Java Virtual Machine, JVM）解释器。

操作系统概念

大部分操作系统提供了特定的基础概念和抽象，例如进程、地址空间、文件等，它们是需要理解的核心内容。下面我们会简要介绍一些基本概念，为了说明这些概念，我们会不时的从 `UNIX` 中提出示例，相同的示例也会存在于其他系统中，我们后面会进行介绍。

进程

操作系统一个很关键的概念就是 `进程(Process)` 。进程的本质就是操作系统执行的一个程序。与每个进程相关的是 `地址空间(address space)` ，这是从某个最小值的存储位置(通常是零)到某个最大值的存储位置的列表。在这个地址空间中，进程可以进行读写操作。地址空间中存放有可执行程序，程序所需要的数据和它的栈。与每个进程相关的还有资源集，通常包括 `寄存器(registers)` （寄存器一般包括 `程序计数器(program counter)` 和 `堆栈指针(stack pointer)` ）、打开文件的清单、突发的报警、有关的进程清单和其他需要执行程序的信息。你可以把进程看作是容纳运行一个程序所有信息的一个容器。

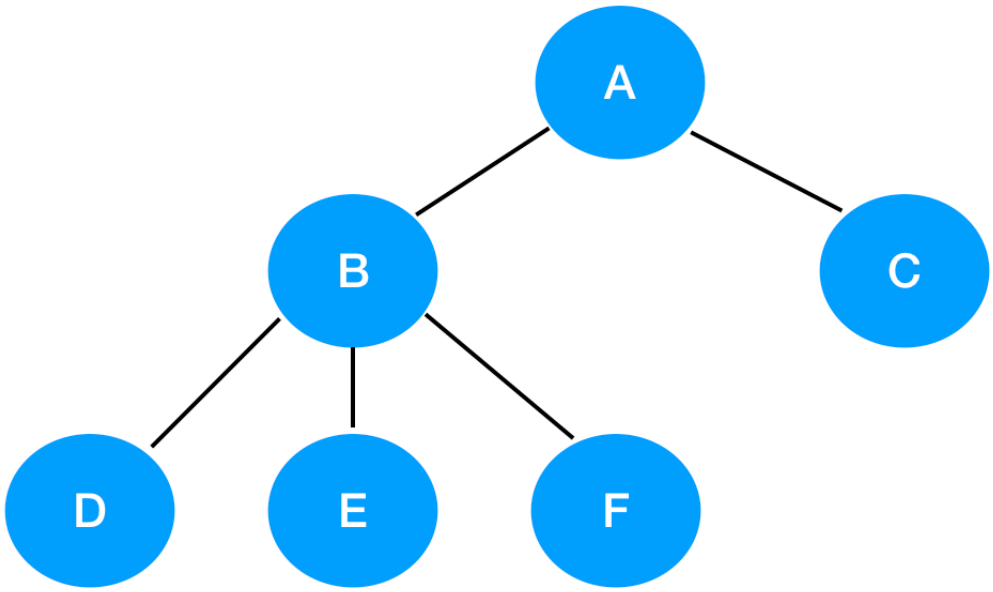
对进程建立一种直观感觉的方式是考虑建立一种多程序的系统。考虑下面这种情况：用户启动一个视频编辑程序，指示它按照某种格式转换视频，然后再去浏览网页。同时，一个检查电子邮件的后台进程被唤醒并开始运行，这样，我们目前就会有三个活动进程：视频编辑器、Web 浏览器和电子邮件接收程序。操作系统周期性的挂起一个进程然后启动运行另一个进程，这可能是由于过去一两秒钟程序用完了 CPU 分配的时间片，而 CPU 转而运行另外的程序。

像这样暂时中断进程后，下次应用程序在此启动时，必须要恢复到与中断时刻相同的状态，这在我们用户看起来是习以为常的事情，但是操作系统内部却做了巨大的事情。**这就像和足球比赛一样，一场完美精彩的比赛是可以忽略裁判的存在的**。这也意味着在挂起时该进程的所有信息都要被保存下来。例如，进程可能打开了多个文件进行读取。与每个文件相关联的是提供当前位置的指针（即下一个需要读取的字节或记录的编号）。当进程被挂起时，必须要保存这些指针，以便在重新启动进程后执行的 `read` 调用将能够正确的读取数据。在许多操作系统中，与一个进程有关的所有信息，除了该进程自身地址空间的内容以外，均存放在操作系统的一张表中，称为 `进程表(process table)` ，进程表是数组或者链表结构，当前存在每个进程都要占据其中的一项。

所以，一个挂起的进程包括：进程的地址空间（往往称作 `磁芯映像`， core image，纪念过去的磁芯存储器），以及对应的进程表项（其中包括寄存器以及稍后启动该进程所需要的许多其他信息）。

与进程管理有关的最关键的系统调用往往是决定着进程的创建和终止的系统调用。考虑一个典型的例子，有一个称为 `命令解释器(command interpreter)` 或 `shell` 的进程从终端上读取命令。此时，用户刚键入一条命令要求编译一个程序。shell 必须先创建一个新进程来执行编译程序，当编译程序结束时，它执行一个系统调用来终止自己的进程。

如果一个进程能够创建一个或多个进程（称为 `子进程`），而且这些进程又可以创建子进程，则很容易找到进程数，如下所示



进程树示意图

上图表示一个进程树的示意图，进程 A 创建了两个子进程 B 和进程 C，子进程 B 又创建了三个子进程 D、E、F。

合作完成某些作业的相关进程经常需要彼此通信来完成作业，这种通信称为 `进程间通信(interprocess communication)`。我们在后面会探讨进程间通信。

其他可用的进程系统调用包括：申请更多的内存（或释放不再需要的内存），等待一个子进程结束，用另一个程序覆盖该程序。

有时，需要向一个正在运行的进程传递信息，而该进程并没有等待接收信息。例如，一个进程通过网络向另一台机器上的进程发送消息进行通信。为了保证一条消息或消息的应答不丢失。发送者要求它所在的操作系统在指定的若干秒后发送一个通知，这样如果对方尚未收到确认消息就可以进行重新发送。在设定该定时器后，程序可以继续做其他工作。

在限定的时间到达后，操作系统会向进程发送一个 `警告信号(alarm signal)`。这个信号引起该进程暂时挂起，无论该进程正在做什么，系统将其寄存器的值保存到堆栈中，并开始重新启动一个特殊的信号处理程，比如重新发送可能丢失的消息。这些信号是软件模拟的硬件中断，除了定时器到期之外，该信号可以通过各种原因产生。许多由硬件检测出来的陷阱，如执行了非法指令或使用了无效地址等，也被转换成该信号并交给这个进程。

系统管理器授权每个进程使用一个给定的 `UID(User IDentification)` 。每个启动的进程都会有一个操作系统赋予的 UID，子进程拥有与父进程一样的 UID。用户可以是某个组的成员，每个组也有一个 `GID(Group IDentification)` 。

在 UNIX 操作系统中，有一个 UID 是 `超级用户(superuser)` ，或者 Windows 中的 `管理员(administrator)` ，它具有特殊的权利，可以违背一些保护规则。在大型系统中，只有系统管理员掌握着那些用户可以称为超级用户。

地址空间

每台计算机都有一些主存用来保存正在执行的程序。在一个非常简单的操作系统中，仅仅有一个应用程序运行在内存中。为了运行第二个应用程序，需要把第一个应用程序移除才能把第二个程序装入内存。

复杂一些的操作系统会允许多个应用程序同时装入内存中运行。为了防止应用程序之间相互干扰（包括操作系统），需要有某种保护机制。虽然此机制是在硬件中实现，但却是由操作系统控制的。

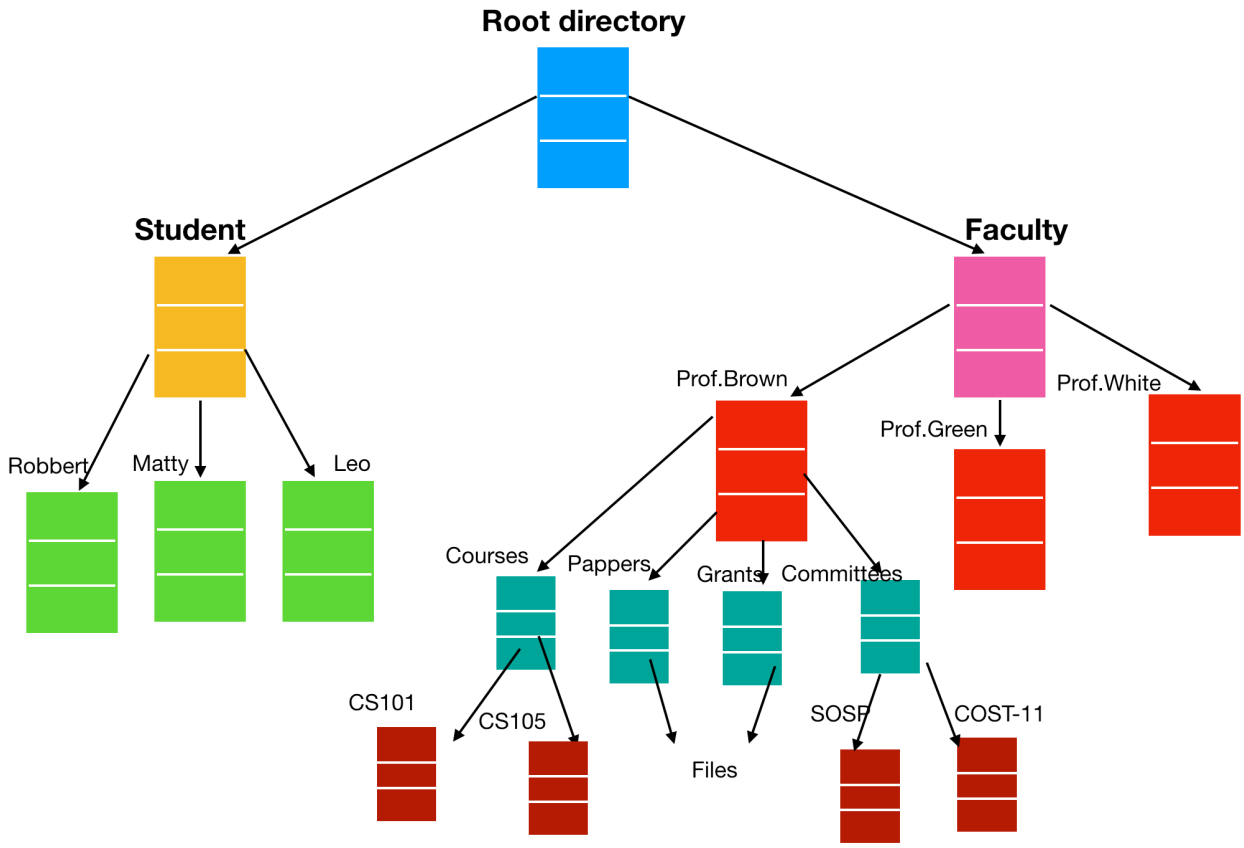
上述观点涉及对计算机主存的管理和保护。另一种同等重要并与存储器有关的内容是管理进程的地址空间。通常，每个进程有一些可以使用的地址集合，典型值从 0 开始直到某个最大值。一个进程可拥有的最大地址空间小于主存。在这种情况下，即使进程用完其地址空间，内存也会有足够的内存运行该进程。

但是，在许多 32 位或 64 位地址的计算机中，分别有 2^32 或 2^64 字节的地址空间。如果一个进程有比计算机拥有的主存还大的地址空间，而且该进程希望使用全部的内存，那该怎么处理？在早期的计算机中是无法处理的。但是现在有了一种 `虚拟内存` 的技术，正如前面讲到过的，操作系统可以把部分地址空间装入主存，部分留在磁盘上，并且在需要时来回交换它们。

文件

几乎所有操作系统都支持的另一个关键概念就是文件系统。如前所述，操作系统的一项主要功能是屏蔽磁盘和其他 I/O 设备的细节特性，给程序员提供一个良好、清晰的独立于设备的抽象文件模型。**创建文件、删除文件、读文件和写文件** 都需要系统调用。在文件可以读取之前，必须先在磁盘上定位和打开文件，在文件读过之后应该关闭该文件，有关的系统调用则用于完成这类操作。

为了提供保存文件的地方，大多数个人计算机操作系统都有 `目录(directory)` 的概念，从而可以把文件分组。比如，学生可以给每个课程都创建一个目录，用于保存该学科的资源，另一个目录可以存放电子邮件，再有一个目录可以存放万维网主页。这就需要系统调用创建和删除目录、将已有文件放入目录中，从目录中删除文件等。目录项可以是文件或者目录，目录和目录之间也可以嵌套，这样就产生了文件系统



大学院系文件系统

进程和文件层次都是以树状的结构组织，但这两种树状结构有不少不同之处。一般进程的树状结构层次不深（很少超过三层），而文件系统的树状结构要深一些，通常会到四层甚至五层。进程树层次结构是暂时的，通常最多存在几分钟，而目录层次则可能存在很长时间。进程和文件在权限保护方面也是有区别的。一般来说，父进程能控制和访问子进程，而在文件和目录中通常存在一种机制，使文件所有者之外的其他用户也能访问该文件。

目录层结构中的每一个文件都可以通过从目录的顶部即 `根目录(Root directory)` 开始的 `路径名(path name)` 来确定。绝对路径名包含了从根目录到该文件的所有目录清单，它们之间用斜杠分隔符分开，在上面的大学院系文件系统中，文件 CS101 的路径名是 `/Faculty/Prof.Brown/Courses/CS101` 。最开始的斜杠分隔符代表的是 `根目录 /` ，也就是文件系统的绝对路径。

> 出于历史原因，Windows 下面的文件系统以 `\` 来作为分隔符，但是 Linux 会以 `/` 作为分隔符。

在上面的系统中，每个进程会有一个 `工作目录(working directory)` ，对于没有以斜线开头给出绝对地址的路径，将在这个工作目录下寻找。如果 `/Faculty/Prof.Brown` 是工作目录，那么 `/Courses/CS101` 与上面给定的绝对路径名表示的是同一个文件。进程可以通过使用系统调用指定新的工作目录，从而变更其工作目录。

在读写文件之前，首先需要打开文件，检查其访问权限。若权限许可，系统将返回一个小整数，称作 `文件描述符(file descriptor)` ，供后续操作使用。若禁止访问，系统则返回一个错误码。

在 UNIX 中，另一个重要的概念是 `特殊文件(special file)` 。提供特殊文件是为了使 I/O 设备看起来像文件一般。这样，就像使用系统调用读写文件一样，I/O 设备也可以通过同样的系统调用进行读写。特殊文件有两种，一种是 `块儿特殊文件(block special file)` 和 `字符特殊文件(character special file)` 。块特殊文件指那些由可随机存取的块组成的设备，如磁盘等。比如打开一个块特殊文件，然后读取第4块，程序可以直接访问设备的第4块而不必考虑存放在该文件的文件系统结构。类似的，字符特殊文件用于打印机、调制解调起和其他接受或输出字符流的设备。按照惯例，特殊文件保存在 `/dev` 目录中。例如，/devv/lp 是打印机。

还有一种与进程和文件相关的特性是管道， `管道(pipe)` 是一种虚文件，他可以连接两个进程



如果 A 和 B 希望通过管道对话，他们必须提前设置管道。当进程 A 相对进程 B 发送数据时，它把数据写到管道上，相当于管道就是输出文件。这样，在 UNIX 中两个进程之间的通信就非常类似于普通文件的读写了。

保护

计算机中含有大量的信息，用户希望能够对这些信息中有用而且重要的信息加以保护，这些信息包括电子邮件、商业计划等，管理这些信息的安全性完全依靠操作系统来保证。例如，文件提供授权用户访问。

比如 UNIX 操作系统，UNIX 操作系统通过对每个文件赋予一个 9 位二进制保护代码，对 UNIX 中的文件实现保护。该保护代码有三个位子段，一个用于所有者，一个用于与所有者同组（用户被系统管理员划分成组）的其他成员，一个用于其他人。每个字段中有一位用于读访问，一位用于写访问，一位用于执行访问。这些位就是著名的 `rwx位` 。例如，保护代码 `rwxr-x--x` 的含义是所有者可以读、写或执行该文件，其他的组成员可以读或执行（但不能写）此文件、而其他人可以执行（但不能读和写）该文件。

shell

操作系统是执行系统调用的代码。编辑器、编译器、汇编程序、链接程序、使用程序以及命令解释符等，尽管非常重要，非常有用，但是它们确实不是操作系统的组成部分。下面我们着重介绍一下 UNIX 下的命令提示符，也就是 `shell` ，shell 虽然有用，但它也不是操作系统的一部分，然而它却能很好的说明操作系统很多特性，下面我们就来探讨一下。

shell 有许多种，例如 **sh、csh、ksh 以及 bash**等，它们都支持下面这些功能，最早起的 shell 可以追溯到 sh

用户登录时，会同时启动一个 shell，它以终端作为标准输入和标准输出。首先显示 `提示符(prompt)` ，它可能是一个 `美元符号($)` ，提示用户 shell 正在等待接收命令，假如用户输入

```
1   date
```

shell 会创建一个子进程，并运行 date 做为子进程。在该子进程运行期间，shell 将等待它结束。在子进程完成时，shell 会显示提示符并等待下一行输入。

用户可以将标准输出重定向到一个文件中，例如

```
date > file
```

同样的，也可以将标准输入作为重定向

```
sort <file1> file2
```

这会调用 sort 程序来接收 file1 的内容并把结果输出到 file2。

可以将一个应用程序的输出通过管道作为另一个程序的输入，因此有

```
cat file1 file2 file3 | sort > /dev/lp
```

这会调用 cat 应用程序来合并三个文件，将其结果输送到 sort 程序中并按照字典进行排序。sort 应用程序又被重定向到 /dev/lp ，显然这是一个打印操作。

系统调用

我们已经可以看到操作系统提供了两种功能：为用户提供应用程序抽象和管理计算机资源。对于大部分在应用程序和操作系统之间的交互主要是应用程序的抽象，例如创建、写入、读取和删除文件。计算机的资源管理对用户来说基本上是透明的。因此，用户程序和操作系统之间的接口主要是处理抽象。为了真正理解操作系统的行为，我们必须仔细的分析这个接口。

多数现代操作系统都有功能相同但是细节不同的系统调用，引发操作系统的调用依赖于计算机自身的机制，而且必须用汇编代码表达。**任何单 CPU 计算机一次执行执行一条指令**。如果一个进程在用户态下运行用户程序，例如从文件中读取数据。那么如果想要把控制权交给操作系统控制，那么必须执行一个异常指令或者系统调用指令。操作系统紧接着需要参数检查找出所需要的调用进程。操作系统紧接着进行参数检查找出所需要的调用进程。然后执行系统调用，把控制权移交给系统调用下面的指令。大致来说，系统调用就像是执行了一个特殊的过程调用，但是只有**系统调用能够进入内核态而过程调用则不能进入内核态**。

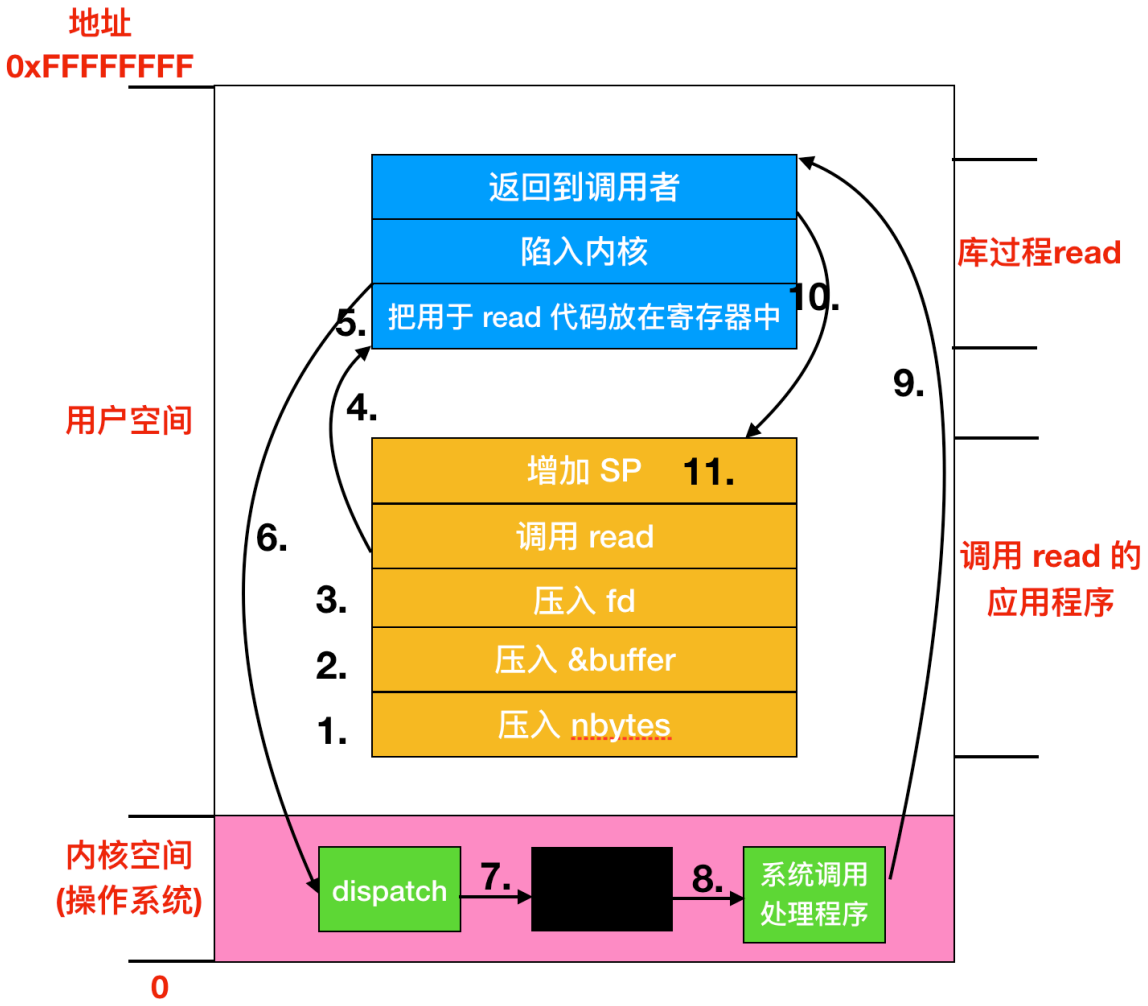
为了能够了解具体的调用过程，下面我们以 `read` 方法为例来看一下调用过程。像上面提到的那样，会有三个参数，第一个参数是指定文件、第二个是指向缓冲区、第三个参数是给定需要读取的字节数。就像几乎所有系统调用一样，它通过使用与系统调用相同的名称来调用一个函数库，从而从C程序中调用：read。

```
count = read(fd,buffer,nbytes);
```

系统调用在 count 中返回实际读出的字节数。这个值通常与 nbytes 相同，但也可能更小。比如在读过程中遇到了文件尾的情况。

如果系统调用不能执行，不管是因为无效的参数还是磁盘错误，count 的值都会被置成 -1，然后在全局变量 `errno` 中放入错误信号。程序应该进场检查系统调用的结果以了解是否出错。

系统调用是通过一系列的步骤实现的，为了更清楚的说明这个概念，我们还以 read 调用为例，在准备系统调用前，首先会把参数压入堆栈，如下所示



完成系统调用 read(fd, buffer, nbytes) 的 11 个步骤

C 和 C++ 编译器使用逆序（必须把第一个参数赋值给 printf(格式字符串)，放在堆栈的顶部）。第一个参数和第三个参数都是值调用，但是第二个参数通过引用传递，即传递的是缓冲区的地址（由 & 指示），而不是缓冲的内容。然后是 C 调用系统库的 read 函数，这也是第四步。

在由汇编语言写成的库过程中，一般把系统调用的编号放在操作系统所期望的地方，如寄存器（第五步）。然后执行一个 `TRAP` 指令，将用户态切换到内核态，并在内核中的一个固定地址开始执行第六步。TRAP 指令实际上与过程调用指令非常相似，它们后面都跟随一个来自远处位置的指令，以及供以后使用的一个保存在栈中的返回地址。

TRAP 指令与过程调用指令存在两个方面的不同

- TRAP 指令会改变操作系统的状态，由用户态切换到内核态，而过程调用不改变模式
- 其次，TRAP 指令不能跳转到任意地址上。根据机器的体系结构，要么跳转到一个单固定地址上，或者指令中有一 8 位长的字段，它给定了内存中一张表格的索引，这张表格中含有跳转地址，然后跳转到指定地址上。

跟随在 TRAP 指令后的内核代码开始检查系统调用编号，然后 `dispatch` 给正确的系统调用处理器，这通常是通过一张由系统调用编号所引用的、指向系统调用处理器的指针表来完成第七步。此时，系统调用处理器运行第八步，一旦系统调用处理器完成工作，控制权会根据 TRAP 指令后面的指令中返回给函数调用库第九步。这个过程接着以通常的过程调用返回的方式，返回到客户应用程序，这是第十步。然后调用完成后，操作系统还必须清除用户堆栈，然后增加 `堆栈指针(increment stackpointer)`，用来清除调用 read 之前压入的参数。从而完成整个 read 调用过程。

在上面的第九步中我们说道，控制可能返回 TRAP 指令后面的指令，把控制权再移交给调用者这个过程中，系统调用会发生阻塞，从而避免应用程序继续执行。这么做是有原因的。例如，如果试图读键盘，此时并没有任何输入，那么调用者就必须被阻塞。在这种情形下，操作系统会检查是否有其他可以运行的进程。这样，当有用户输入 时候，进程会提醒操作系统，然后返回第 9 步继续运行。

下面，我们会列出一些常用的 `POSIX` 系统调用，POSIX 系统调用大概有 100 多个，它们之中最重要的一些调用见下表

进程管理

调用	说明
pid = fork()	创建与父进程相同的子进程
pid = waitpid(pid, &statloc,options)	等待一个子进程终止
s = execve(name,argv,environp)	替换一个进程的核心映像
exit(status)	终止进程执行并返回状态

文件管理

调用	说明
fd = open(file, how,...)	打开一个文件使用读、写
s = close(fd)	关闭一个打开的文件
n = read(fd,buffer,nbytes)	把数据从一个文件读到缓冲区中
n = write(fd,buffer,nbytes)	把数据从缓冲区写到一个文件中
position = iseek(fd,offset,whence)	移动文件指针
s = stat(name,&buf)	取得文件状态信息

目录和文件系统管理

调用	说明
s = mkdir(nname,mode)	创建一个新目录
s = rmdir(name)	删去一个空目录
s = link(name1,name2)	创建一个新目录项 name2,并指向 name1
s = unlink(name)	删去一个目录项
s = mount(special,name,flag)	安装一个文件系统
s = umount(special)	卸载一个文件系统

其他

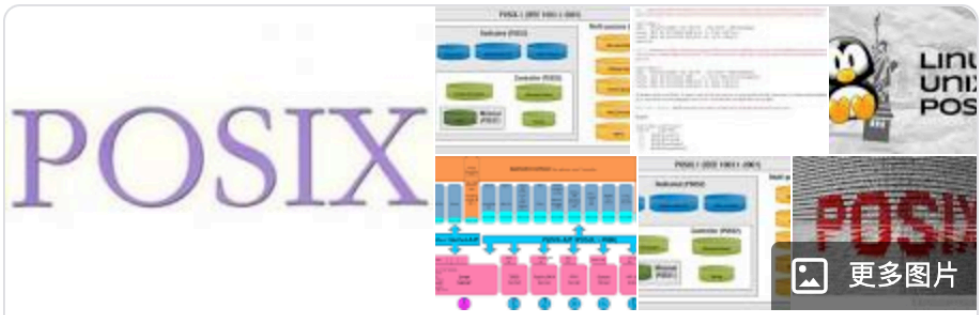
调用	说明
s = chdir(dirname)	改变工作目录
s = chmod(name,mode)	修改一个文件的保护位
s = kill(pid, signal)	发送信号给进程
seconds = time(&seconds)	获取从 1970 年1月1日至今的时间

上面的系统调用参数中有一些公共部分，例如 pid 系统进程 id，fd 是文件描述符，n 是字节数，position 是在文件中的偏移量、seconds 是流逝时间。

从宏观角度上看，这些系统调所提供的服务确定了多数操作系统应该具有的功能，下面分别来对不同的系统调用进行解释

用于进程管理的系统调用

在 UNIX 中，`fork` 是唯一可以在 POSIX 中创建进程的途径，它创建一个原有进程的副本，包括所有的文件描述符、寄存器等内容。在 fork 之后，原有进程以及副本（父与子）就分开了。在 fork 过程中，所有的变量都有相同的值，虽然父进程的数据通过复制给子进程，但是后续对其中任何一个进程的修改不会影响到另外一个。fork 调用会返回一个值，在子进程中该值为 0 ，并且在父进程中等于子进程的 `进程标识符(Process IDentified,PID)` 。使用返回的 PID，就可以看出来哪个是父进程和子进程。



可移植操作系统接口 (POSIX)

可移植操作系统接口是IEEE为要在各种UNIX操作系统上运行软件，而定义API的一系列互相关联的标准的总称，其正式称呼为IEEE Std 1003，而国际标准名称为ISO/IEC 9945。此标准源于一个大约开始于1985年的项目。POSIX这个名称是由理查德·斯托曼应IEEE的要求而提议的一个易于记忆的名称。 维基百科

相关标准： ISO/IEC 9945

最新版本： IEEE Std 1003.1-2017; 2017; 3 年前

组织： Austin Group (IEEE計算機學會, 国际开放标准组织, ISO / IEC JTC 1)

状态： Published

在多数情况下， 在 fork 之后，子进程需要执行和父进程不一样的代码。从终端读取命令，创建一个子进程，等待子进程执行命令，当子进程结束后再读取下一个输入的指令。为了等待子进程完成，父进程需要执行 `waitpid` 系统调用，父进程会等待直至子进程终止（若有多个子进程的话，则直至任何一个子进程终止）。waitpid 可以等待一个特定的子进程，或者通过将第一个参数设为 -1 的方式，等待任何一个比较老的子进程。当 waitpid 完成后，会将第二个参数 `statloc` 所指向的地址设置为子进程的退出状态（正常或异常终止以及退出值）。有各种可使用的选项，它们由第三个参数确定。例如，如果没有已经退出的子进程则立刻返回。

那么 shell 该如何使用 fork 呢？在键入一条命令后，shell 会调用 fork 命令创建一个新的进程。这个子进程会执行用户的指令。通过使用 `execve` 系统调用可以实现系统执行，这个系统调用会引起整个核心映像被一个文件所替代，该文件由第一个参数给定。下面是一个简化版的例子说明 fork、waitpid 和 execve 的使用

```
#define TRUE 1

/* 一直循环下去 */
while(TRUE){

/* 在屏幕上显示提示符 */
  type_prompt();

  /* 从终端读取输入 */
```

```
    read_command(command,parameters)

    /* fork 子进程 */
    if(fork() != 0){

      /* 父代码 */
      /* 等待子进程执行完毕 */
      waitpid(-1, &status, 0);
    }else{

      /* 执行命令 */
      /* 子代码 */
      execve(command,parameters,0)
    }
  }
```

一般情况下，execve 有三个参数：将要执行的文件名称，一个指向变量数组的指针，以及一个指向环境数组的指针。这里对这些参数做一个简要的说明。

先看一个 shell 指令

```
cp file1 file2
```

此命令把 file1 复制到 file2 文件中，在 shell 执行 fork 之后，子进程定位并执行文件拷贝，并将源文件和目标文件的名称传递给它。

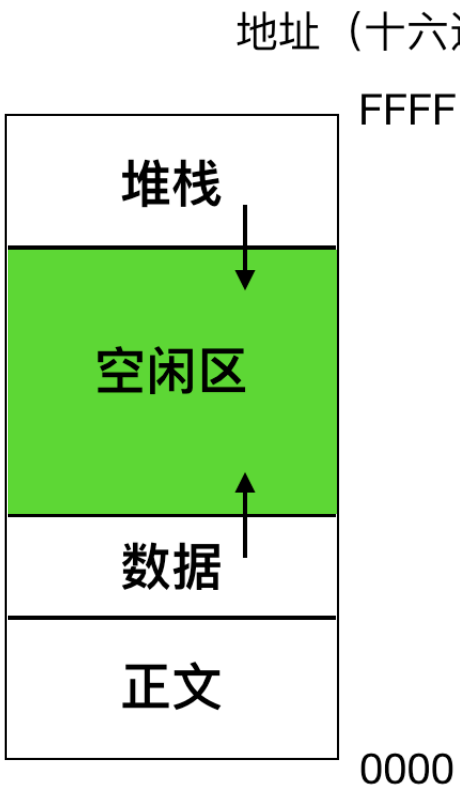
cp 的主程序（以及包含其他大多数 C 程序的主程序）包含声明

```
main(argc,argv,envp)
```

其中 argc 是命令行中参数数目的计数，包括程序名称。对于上面的例子，`argc` 是3。第二个参数 `argv` 是数组的指针。该数组的元素 i 是指向该命令行第 i 个字符串的指针。在上面的例子中，argv[0] 指向字符串 cp，argv[1] 指向字符串 file1，argv[2] 指向字符串 file2。main 的第三个参数是指向环境的指针，该环境是一个数组，含有 `name = value` 的赋值形式，用以将诸如终端类型以及根目录等信息传送给程序。这些变量通常用来确定用户希望如何完成特定的任务（例如，使用默认打印机）。在上面的例子中，没有环境参数传递给 execve ，所以环境变量是 0 ，所以 execve 的第三个参数为 0 。

可能你觉得 execve 过于复杂，这时候我要鼓励一下你，execve 可能是 POSIX 的全部系统调用中最复杂的一个了，其他都比较简单。作为一个简单的例子，我们再来看一下 `exit` ，这是进程在执行完成后应执行的系统调用。这个系统调用有一个参数，它的退出状态是 0 - 255 之间，它通过 waitpid 系统调用中的 statloc 返回给父级。

UNIX 中的进程将内存划分成三个部分：`text segment,文本区` ，例如程序代码，`data segment,数据区` ，例如变量，`stack segment` ，栈区域。数据向上增长而堆栈向下增长，如下图所示



上图能说明三个部分的内存分配情况，夹在中间的是空闲区，也就是未分配的区域，堆栈在需要时自动的挤压空闲区域，不过数据段的扩展是显示地通过系统调用 `brk` 进行的，在数据段扩充后，该系统调用指向一个新地址。但是，这个调用不是 POSIX 标准中定义的，对于存储器的动态分配，鼓励程序员使用 `malloc` 函数，而 malloc 的内部实现则不是一个适合标准化的主题，因为几乎没有程序员直接使用它。

用于文件管理的系统调用

许多系统调用都与文件系统有关，要读写一个文件，必须先将其打开。这个系统调用通过绝对路径名或指向工作目录的相对路径名指定要打开文件的名称，而代码 `O_RDONLY` 、 `O_WRONLY` 或 `O_RDWR` 的含义分别是只读、只写或者两者都可以，为了创建一个新文件，使用 `O_CREATE` 参数。然后可使用返回的文件描述符进行读写操作。接着，可以使用 close 关闭文件，这个调用使得文件描述符在后续的 open 中被再次使用。

最常用的调用还是 `read` 和 `write` ，我们再前面探讨过 read 调用，write 具有与 read 相同的参数。

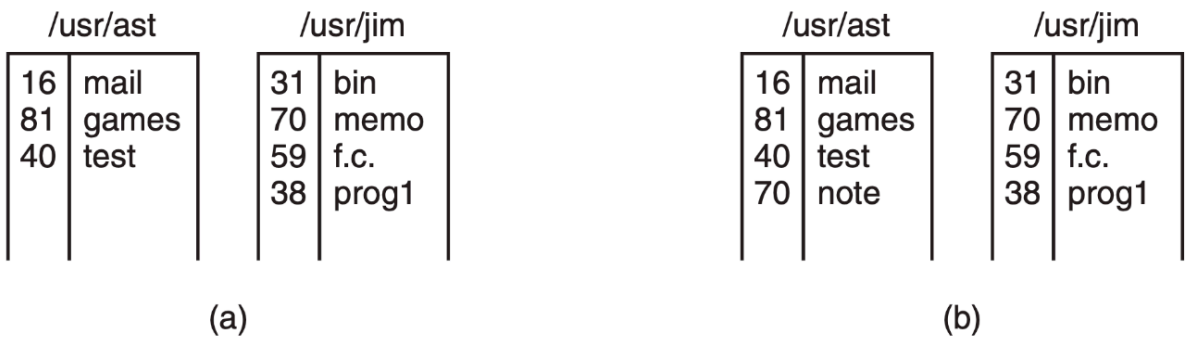
尽管多数程序频繁的读写文件，但是仍有一些应用程序需要能够随机访问一个文件的任意部分。与每个文件相关的是一个指向文件当前位置的指针。在顺序读写时，该指针通常指向要读出（写入）的下一个字节。 `Iseek` 调用可以改变该位置指针的值，这样后续的 read 或 write 调用就可以在文件的任何地方开始。

Iseek 有三个参数， `position = iseek(fd,offset,whence)` ，第一个是文件描述符，第二个是文件位置，第三个是说明该文件位置是相对于文件起始位置，当前位置还是文件的结尾。在修改了指针之后，Iseek 所返回的值是文件中的绝对位置。

UNIX 为每个文件保存了该文件的类型（普通文件、特殊文件、目录等）、大小，最后修改时间以及其他信息，程序可以通过 `stat` 系统调用查看这些信息。`s = stat(name,&buf)`，第一个参数指定了被检查的文件；第二个参数是一个指针，该指针指向存放这些信息的结构。对于一个打开的文件而言，fstat 调用完成同样的工作。

用于目录管理的系统调用

下面我们探讨目录和整个文件系统的系统调用，上面探讨的是和某个文件有关的系统调用。`mkdir` 和 `rmdir` 分别用于创建 `s = mkdir(nname,mode)` 和删除 `s = rmdir(name)` 空目录，下一个调用是 `s = link(name1,name2)` 它的作用是允许同一个文件以两个或者多个名称出现，多数情况下是在不同的目录中使用 link，下面我们探讨一下 link 是如何工作的



图中有两个用户 `ast` 和 `jim`，每个用户都有他自己的一个目录和一些文件，如果 ast 要执行一个包含下面系统调用的应用程序

```
1   link("/usr/jim/memo", "/usr/ast/note");
```

jim 中的 memo 文件现在会进入到 ast 的目录中，在 note 名称下。此后，`/usr/jim/memo` 和 `/usr/ast/note` 会有相同的名称。

> 用户目录是保存在 /usr，/user，/home 还是其他位置，都是由本地系统管理员决定的。

要理解 link 是如何工作的需要清楚 link 做了什么操作。UNIX 中的每个文件都有一个独一无二的版本，也称作 `i - number，i-编号`，它标示着不同文件的版本。这个 i - 编号是 `i-nodes,i-节点` 表的索引。每个文件都会表明谁拥有这个文件，这个磁盘块的位置在哪，等等。目录只是一个包含一组（i编号，ASCII名称）对应的文件。UNIX 中的第一个版本中，每个目录项都会有 16 个字节，2 个字节对应 i - 编号和 14 个字节对应其名称。现在需要一个更复杂的结构需要支持长文件名，但是从概念上讲一个目录仍是一系列（i-编号，ASCII 名称）的集合。在上图中，`mail` 的 i-编号为 16，依此类推。link 只是利用某个已有文件的 i-编号，创建一个新目录项（也许用一个新名称）。在上图 b 中，你会发现有两个相同的 70 i-编号的文件，因此它们需要有相同的文件。如果其中一个使用了 `unlink` 系统调用的话，其中一个会被移除，另一个将保留。如果两个文件都移除了，则 UNIX 会发现该文件不存在任何没有目录项（i-节点中的一个域记录着指向该文件的目录项），就会把该文件从磁盘中移除。

就像我们上面提到过的那样，`mount` 系统 `s = mount(special,name,flag)` 调用会将两个文件系统合并为一个。通常的情况是将根文件系统分布在硬盘（子）分区上，并将用户文件分布在另一个（子）分区上，该根文件系统包含常用命令的二进制（可执行）版本和其他使用频繁的文件。然后，用户就会插入可读取的 USB 硬盘。

通过执行 mount 系统调用，USB 文件系统可以被添加到根文件系统中，



(a) (b)

图 a 是安装前的系统文件，图 b 是安装后的系统文件。

如果用 C 语言来执行那就是

```
1    mount("/dev/sdb0","/mnt",0)
```

这里，第一个参数是 USB 驱动器 0 的块特殊文件名称，第二个参数是被安装在树中的位置，第三个参数说明将要安装的文件系统是可读写的还是只读的。

当不再需要一个文件系统时，可以使用 umount 移除之。

其他系统调用

除了进程、文件、目录系统调用，也存在其他系统调用的情况，下面我们来探讨一下。我们可以看到上面其他系统调用只有四种，首先来看第一个 chdir，chdir 调用更改当前工作目录，在调用

```
1    chdir("/usr/ast/test");
```

后，打开 xyz 文件，会打开 `/usr/ast/test/xyz` 文件，工作目录的概念消除了总是需要输入长文件名的需要。

在 UNIX 系统中，每个文件都会有保护模式，这个模式会有一个 `读-写-执行` 位，它用来区分所有者、组和其他成员。 `chmod` 系统调用提供改变文件模式的操作。例如，要使一个文件除了对所有者之外的用户可读，你可以执行

```
1    chmod("file",0644);
```

`kill` 系统调用是用户和用户进程发送信号的方式，如果一个进程准备好捕捉一个特定的信号，那么在信号捕捉之前，会运行一个信号处理程序。如果进程没有准备好捕捉特定的信号，那么信号的到来会杀掉该进程（此名字的由来）。

POSIX 定义了若干时间处理的进程。例如， `time` 以秒为单位返回当前时间，0 对应着 1970 年 1月 1 日。在一台 32 位字的计算机中，time 的最大值是 (2^32) - 1秒，这个数字对应 136 年多一点。所以在 2106 年，32 位的 UNIX 系统会发飙。如果读者现在有 32 位 UNIX 系统，建议在 2106 年更换位 64 位操作系统（偷笑～）。

Win 32 API

上面我们提到的都是 UNIX 系统调用，现在我们来聊聊 Win 32 中的系统调用。Windows 和 UNIX 在各自的编程方式上有着根本的不同。UNIX 程序由执行某些操作或执行其他操作的代码组成，进行系统调用以执行某些服务。Windows 系统则不同，Windows 应用程序通常是由事件驱动的。主程序会等待一些事件发生，然后调用程序去处理。最简单的事件处理是键盘敲击和鼠标滑过，或者是鼠标点击，或者

是插入 USB 驱动，然后操作系统调用处理器去处理事件，更新屏幕和更新程序内部状态。这是与 UNIX 不同的设计风格。

当然，Windows 也有系统调用。在 UNIX 中，系统调用（比如 read）和系统调用所使用的调用库（例如 read）几乎是一对一的关系。而在 Windows 中，情况则大不相同。首先，函数库的调用和实际的系统调用几乎是不对应的。微软定义了一系列过程，称为 `Win32应用编程接口(Application Programming Interface)`，程序员通过这套标准的接口来实现系统调用。这个接口支持从 Windows 95 版本以来所有的 Windows 版本。

Win32 API 调用的数量是非常巨大的，有数千个多。但这些调用并不都是在内核态的模式下运行时，有一些是在用户态的模型下运行。Win32 API 有大量的调用，用来管理视窗、几何图形、文本、字体、滚动条、对话框、菜单以及 GUI 的其他功能。为了使图形子系统在内核态下运行，需要系统调用，否则就只有函数库调用。

我们把关注点放在和 Win32 系统调用中来，我们可以简单看一下 Win32 API 中的系统调用和 UNIX 中有什么不同（并不是所有的系统调用）

UNIX	Win32	说明
fork	CreateProcess	创建一个新进程
waitpid	WaitForSingleObject	等待一个进程退出
execve	none	CraeteProcess = fork + servvice
exit	ExitProcess	终止执行
open	CreateFile	创建一个文件或打开一个已有的文件
close	CloseHandle	关闭文件
read	ReadFile	从单个文件中读取数据
write	WriteFile	向单个文件写数据
lseek	SetFilePointer	移动文件指针
stat	GetFileAttributesEx	获得不同的文件属性
mkdir	CreateDirectory	创建一个新的目录
rmdir	RemoveDirectory	移除一个空的目录
link	none	Win32 不支持 link
unlink	DeleteFile	销毁一个已有的文件
mount	none	Win32 不支持 mount
umount	none	Win32 不支持 mount，所以也不支持mount
chdir	SetCurrentDirectory	切换当前工作目录
chmod	none	Win32 不支持安全
kill	none	Win32 不支持信号
time	GetLocalTime	获取当前时间

上表中是 UNIX 调用大致对应的 Win32 API 系统调用，简述一下上表。`CreateProcess` 用于创建一个新进程，它把 UNIX 中的 fork 和 execve 两个指令合成一个，一起执行。它有许多参数用来指定新创建进程的性质。Windows 中没有类似 UNIX 中的进程层次，所以不存在父进程和子进程的概念。在进程创建之后，创建者和被创建者是平等的。`WaitForSingleObject` 用于等待一个事件，等待的事件可以是多种可能的事件。如果有参数指定了某个进程，那么调用者将等待指定的进程退出，这通过 `ExitProcess` 来完成。

然后是6个文件操作，在功能上和 UNIX 的调用类似，然而在参数和细节上是不同的。和 UNIX 中一样，文件可以打开，读取，写入，关闭。`SetFilePointer` 和 `GetFileAttributesEx` 设置文件的位置并取得文件的属性。

Windows 中有目录，目录分别用 `CreateDirectory` 以及 `RemoveDirectory` API 调用创建和删除。也有对当前的目录的标记，这可以通过 `SetCurrentDirectory` 来设置。使用 `GetLocalTime` 可获得当前时间。

Win32 接口中没有文件的链接、文件系统的 mount、umount 和 stat ，当然， Win32 中也有大量 UNIX 中没有的系统调用，特别是对 GUI 的管理和调用。

操作系统结构

下面我们会探讨操作系统的几种结构，主要包括**单体结构、分层系统、微内核、客户-服务端系统、虚拟机和外核**等。下面以此来探讨一下

单体系统

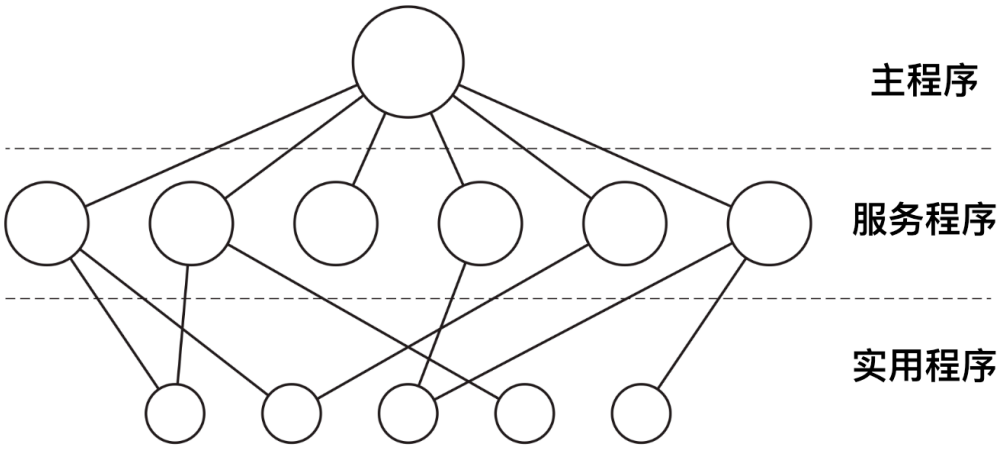
到目前为止，在大多数系统中，整个系统在内核态以单一程序的方式运行。整个操作系统是以程序集合来编写的，链接在一块形成一个大的二进制可执行程序。使用此技术时，如果系统中的每个过程都提供了前者所需的一些有用的计算，则它可以自由调用任何其他过程。在单体系统中，调用任何一个所需要的程序都非常高效，但是上千个不受限制的彼此调用往往非常臃肿和笨拙，而且单体系统必然存在单体问题，那就是只要系统发生故障，那么任何系统和应用程序将不可用，这往往是灾难性的。

在单体系统中构造实际目标程序时，会首先编译所有单个过程（或包含这些过程的文件），然后使用系统链接器将它们全部绑定到一个可执行文件中

对于单体系统，往往有下面几种建议

- 需要有一个主程序，用来调用请求服务程序
- 需要一套服务过程，用来执行系统调用
- 需要一套服务程序，用来辅助服务过程调用

在单体系统中，对于每个系统调用都会有一个服务程序来保障和运行。需要一组实用程序来弥补服务程序需要的功能，例如从用户程序中获取数据。可将各种过程划分为一个三层模型

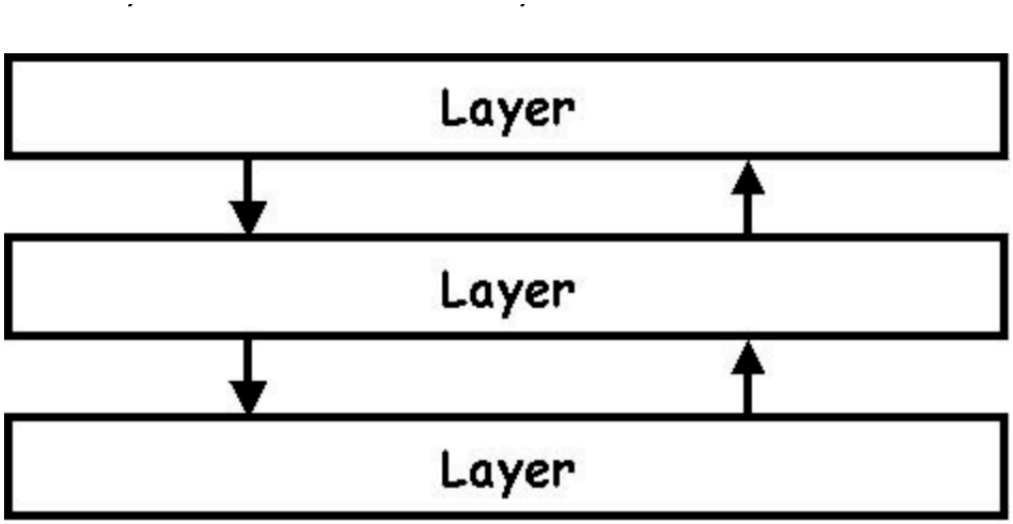


除了在计算机初启动时所装载的核心操作系统外，许多操作系统还支持额外的扩展。比如 I/O 设备驱动和文件系统。这些部件可以按需装载。在 UNIX 中把它们叫做 `共享库(shared library)` ，在 Windows 中则被称为 `动态链接库(Dynamic Link Library,DLL)` 。他们的扩展名为 `.dll` ，在 `C:\Windows\system32` 目录下存在 1000 多个 DLL 文件，所以不要轻易删除 C 盘文件，否则可能

就炸了哦。

分层系统

分层系统使用层来分隔不同的功能单元。每一层只与该层的上层和下层通信。每一层都使用下面的层来执行其功能。层之间的通信通过预定义的固定接口通信。



分层系统是由 `E.W.Dijkstar` 和他的学生在荷兰技术学院所开发的 THE 系统。

把上面单体系统进一步通用化，就变为了一个层次式结构的操作系统，它的上层软件都是在下层软件的基础之上构建的。该系统分为六层，如下所示

层号	功能
5	操作员
4	用户程序
3	输入/输出管理
2	操作员-进程通信
1	存储器和磁鼓管理
0	处理器分配和多道程序编程

处理器在 0 层运行，当中断发生或定时器到期时，由该层完成进程切换；在第 0 层之上，系统由一些连续的进程组成，编写这些进程时不用再考虑在单处理器上多进程运行的细节。内存管理在第 1 层，它分配进程的主存空间。第 1 层软件保证一旦需要访问某一页面，该页面必定已经在内存中，并且在页面不需要的时候将其移出。

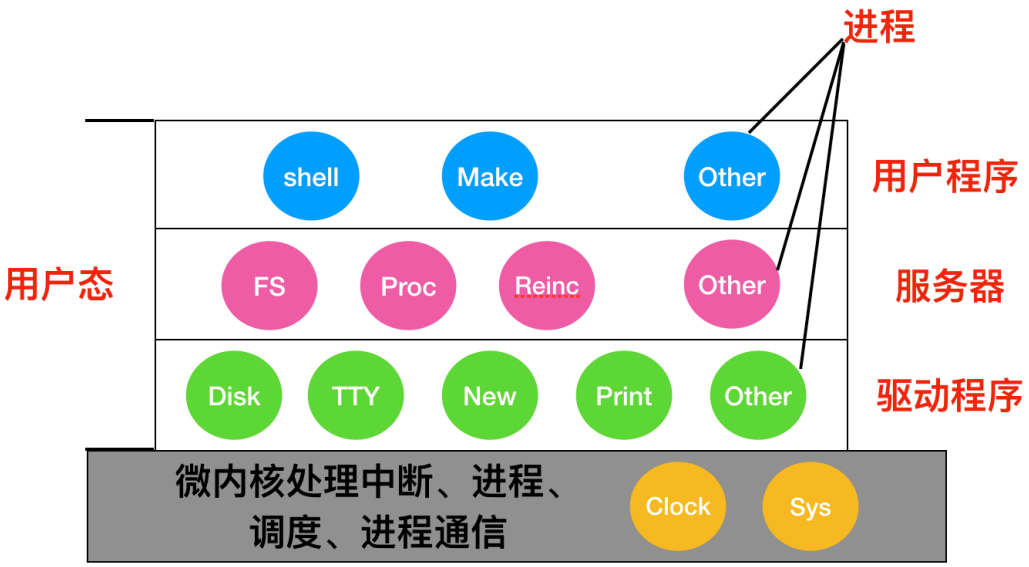
第 2 层处理进程与操作员控制台（即用户）之间的通信。第 3 层管理 I/O 设备和相关的信息流缓冲区。第 4 层是用户程序层，用户程序不用考虑进程、内存、控制台或 I/O 设备管理等细节。系统操作员在第 5 层。

微内核

在分层方式中，设计者要确定在哪里划分 `内核-用户` 的边界。传统上，所有的层都在内核中，但是这样做没有必要。事实上，尽可能减少内核态中功能可能是更好的做法。因为内核中的错误很难处理，一旦内核态中出错误会拖累整个系统。

所以，为了实现高可靠性，将操作系统划分成小的、层级之间能够更好定义的模块是很有必要的，只有一个模块 --- 微内核 --- 运行在内核态，其余模块可以作为普通用户进程运行。由于把每个设备驱动和文件系统分别作为普通用户进程，这些模块中的错误虽然会使这些模块崩溃，但是不会使整个系统死机。

`MINIX 3` 是微内核的代表作，它的具体结构如下



MINI 3 系统结构

在内核的外部，系统的构造有三层，它们都在用户态下运行，最底层是设备驱动器。由于它们都在用户态下运行，所以不能物理的访问 I/O 端口空间，也不能直接发出 I/O 命令。相反，为了能够对 I/O 设备编程，驱动器构建一个结构，指明哪个参数值写到哪个 I/O 端口，并声称一个内核调用，这样就完成了一次调用过程。

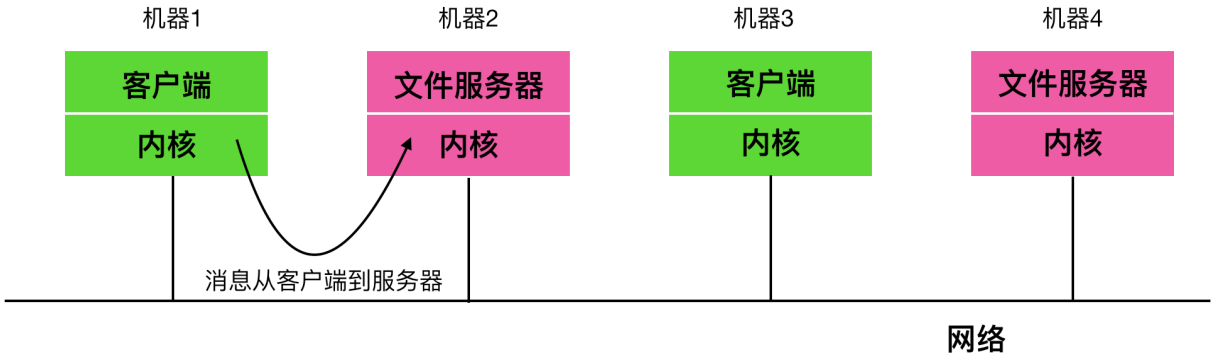
位于用户态的驱动程序上面是 `服务器` 层，包含有服务器，它们完成操作系统的多数工作。由一个或多个文件服务器管理着文件系统，进程管理器创建、销毁和管理进程。服务器中有一个特殊的服务器称为 `再生服务器(reincarnation server)`，它的任务就是检查服务器和驱动程序的功能是否正确，一旦检查出来错误，它就会补上去，无需用户干预。这种方式使得系统具有可恢复性，并具有较高的可靠性。

微内核中的内核还具有一种 `机制` 与 `策略` 分离的思想。比如系统调度，一个比较简单的调度算法是，对每个进程赋予一个优先级，并让内核执行具有最高优先级的进程。这里，内核机制就是寻找最高的优先级进程并运行。而策略（赋予进程优先级）可以在用户态中的进程完成。在这种模式中，策略和机制是分离的，从而使内核变得更小。

客户-服务器模式

微内核思想的策略是把进程划分为两类： `服务器`，每个服务器用来提供服务； `客户端`，使用这些服务。这个模式就是所谓的 `客户-服务器` 模式。

客户-服务器模式会有两种载体，一种情况是一台计算机既是客户又是服务器，在这种方式下，操作系统会有某种优化；但是普遍情况下是客户端和服务器在不同的机器上，它们通过局域网或广域网连接。



客户通过发送消息与服务器通信，客户端并不需要知道这些消息是在本地机器上处理，还是通过网络被送到远程机器上处理。对于客户端而言，这两种情形是一样的：都是发送请求并得到回应。

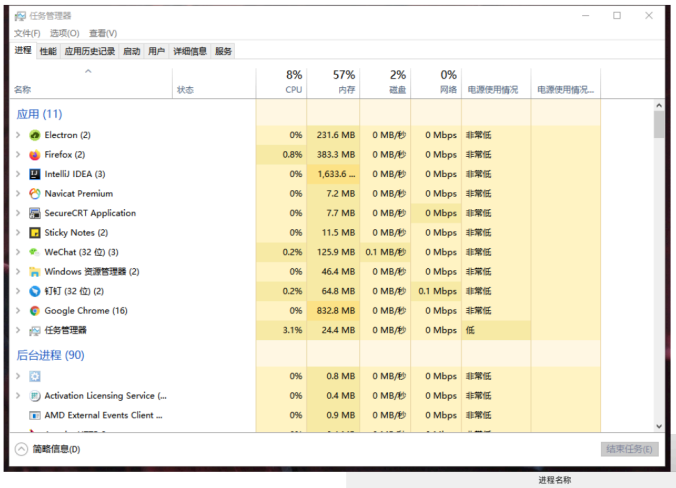
越来越多的系统，包括家里的 PC，都成为客户端，而在某地运行的大型机器则成为服务器。许多 web 就是以这种方式运行的。一台 PC 向某个服务器请求一个 Web 页面，服务器把 Web 页面返回给客户端，这就是典型的客服-服务器模式

进程和线程

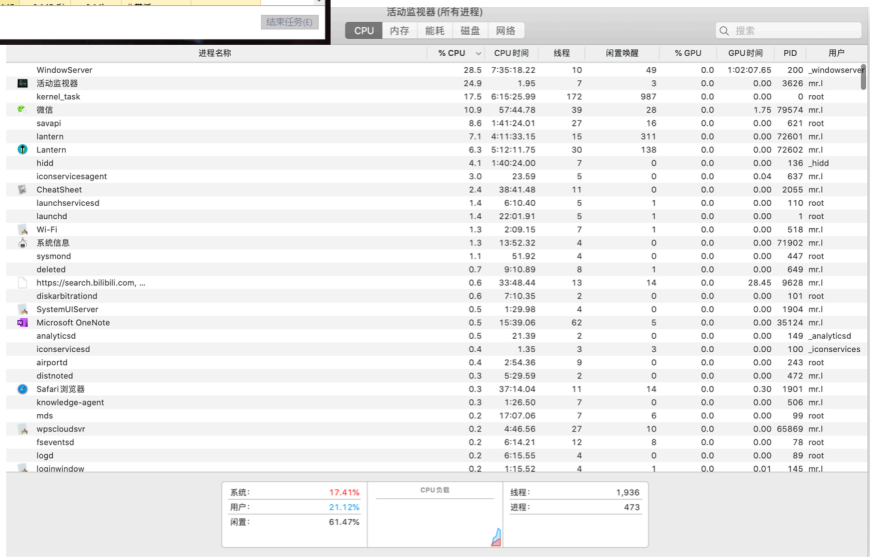


进程

操作系统中最核心的概念就是 `进程`，进程是对正在运行中的程序的一个抽象。操作系统的其他所有内容都是围绕着进程展开的。进程是操作系统提供的最古老也是最重要的概念之一。即使可以使用的 CPU 只有一个，它们也支持 `（伪）并发` 操作。它们会将一个单独的 CPU 抽象为多个虚拟机的 CPU。可以说：没有进程的抽象，现代操作系统将不复存在。



Windows 系统进程



UNIX 系统进程

所有现代的计算机会在同一时刻做很多事情，过去使用计算机的人（单 CPU）可能完全无法理解现在这种变化，举个例子更能说明这一点：首先考虑一个 Web 服务器，请求都来自于 Web 网页。当一个请求到达时，服务器会检查当前页是否在缓存中，如果是在缓存中，就直接把缓存中的内容返回。如果缓存中没有的话，那么请求就会交给磁盘来处理。但是，从 CPU 的角度来看，磁盘请求需要更长的时间，因为磁盘请求会很慢。当硬盘请求完成时，更多其他请求才会进入。如果有多个磁盘的话，可以在第一个请求完成前就可以连续的对其他磁盘发出部分或全部请求。很显然，这是一种并发现象，需要有并发控制条件来控制并发现象。

现在考虑只有一个用户的 PC。当系统启动时，许多进程也在后台启动，用户通常不知道这些进程的启动，试想一下，当你自己的计算机启动的时候，你能知道哪些进程是需要启动的么？这些后台进程可能是一个需要输入电子邮件的电子邮件进程，或者是一个计算机病毒查杀进程来周期性的更新病毒库。某个用户进程可能会在所有用户上网的时候打印文件以及刻录 CD-ROM，这些活动都需要管理。于是一个支持多进程的多道程序系统就会显得很有必要了。

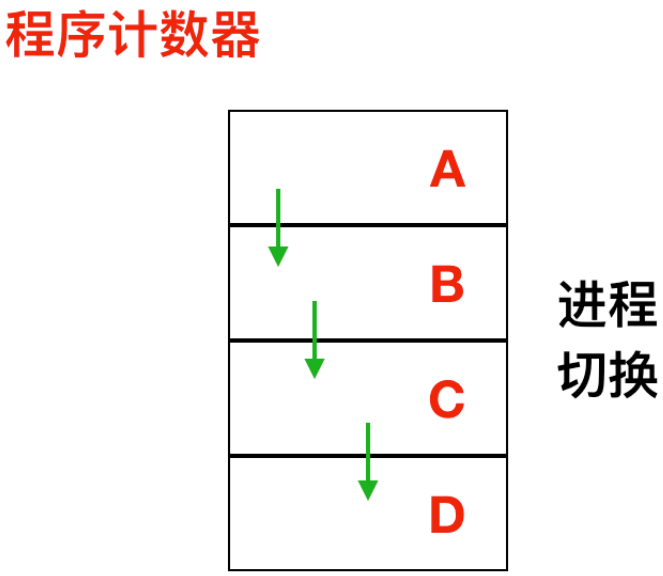
在许多多道程序系统中，CPU 会在 `进程` 间快速切换，使每个程序运行几十或者几百毫秒。然而，严格意义来说，在某一个瞬间，CPU 只能运行一个进程，然而我们如果把时间定位为 1 秒内的话，它可能运行多个进程。这样就会让我们产生 `并行` 的错觉。有时候人们说的 `伪并行(pseudoparallelism)` 就是这种情况，以此来区分多处理器系统(该系统由两个或多个 CPU 来共享同一个物理内存)

> 再来详细解释一下伪并行： `伪并行` 是指单核或多核处理器同时执行多个进程，从而使程序更快。通过以非常有限的时间间隔在程序之间快速切换CPU，因此会产生并行感。 缺点是 CPU 时间可能分配给下一个进程，也可能不分配给下一个进程。

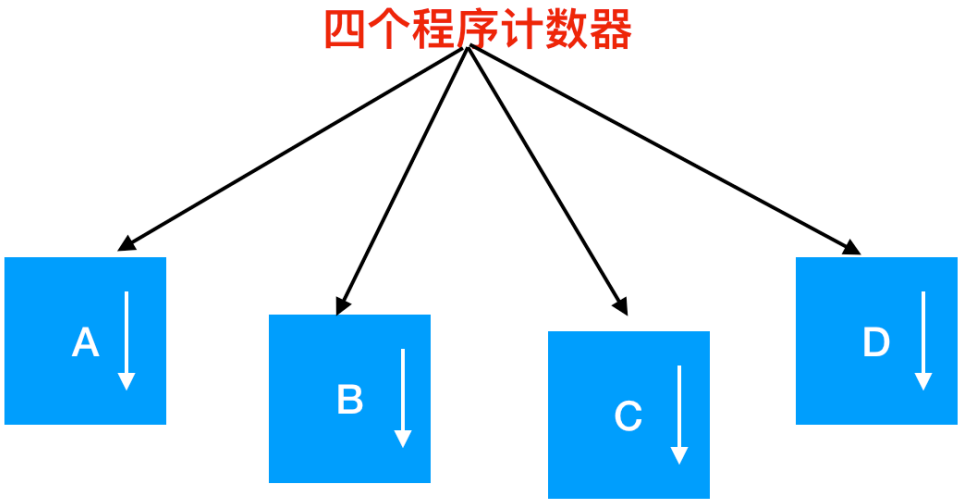
因为 CPU 执行速度很快，进程间的换进换出也非常迅速，因此我们很难对多个并行进程进行跟踪，所以，在经过多年的努力后，操作系统的设计者开发了用于描述并行的一种概念模型（顺序进程），使得并行更加容易理解和分析，对该模型的探讨，也是本篇文章的主题。下面我们就来探讨一下进程模型

进程模型

在进程模型中，所有计算机上运行的软件，通常也包括操作系统，被组织为若干 `顺序进程(sequential processes)` ，简称为 `进程(process)` 。一个进程就是一个正在执行的程序的实例，进程也包括程序计数器、寄存器和变量的当前值。从概念上来说，每个进程都有各自的虚拟 CPU，但是实际情况是 CPU 会在各个进程之间进行来回切换。

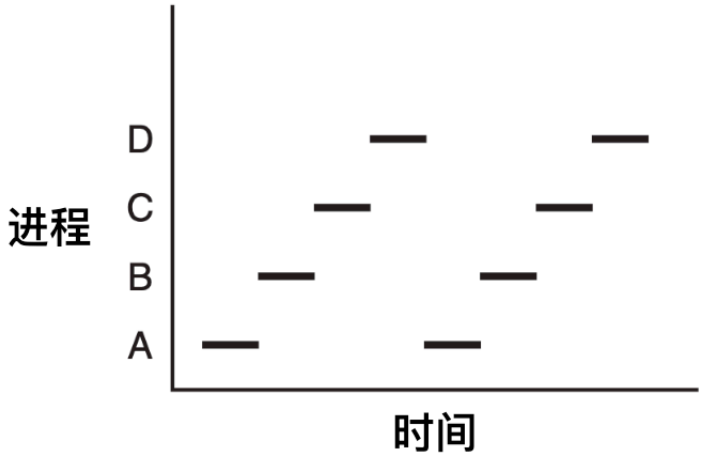


如上图所示，这是一个具有 4 个程序的多道处理程序，在进程不断切换的过程中，程序计数器也在不同的变化。



在上图中，这 4 道程序被抽象为 4 个拥有各自控制流程（即每个自己的程序计数器）的进程，并且每个程序都独立的运行。当然，实际上只有一个物理程序计数器，每个程序要运行时，其逻辑程序计数器会装载到物理程序计数器中。当程序运行结束后，其物理程序计数器就会是真正的程序计数器，然后再把它放回进程的逻辑计数器中。

从下图我们可以看到，在观察足够长的一段时间后，所有的进程都运行了，**但在任何一个给定的瞬间仅有一个进程真正运行**。



因此，当我们说一个 CPU 只能真正一次运行一个进程的时候，即使有 2 个核（或 CPU），**每一个核也只能一次运行一个线程**。

由于 CPU 会在各个进程之间来回快速切换，所以每个进程在 CPU 中的运行时间是无法确定的。并且当同一个进程再次在 CPU 中运行时，其在 CPU 内部的运行时间往往也是不固定的。进程和程序之间的区别是非常微妙的，但是通过一个例子可以让你加以区分：想想一位会做饭的计算机科学家正在为他的女儿制作生日蛋糕。他有做生日蛋糕的食谱，厨房里有所需的原谅：面粉、鸡蛋、糖、香草汁等。在这个比喻中，做蛋糕的食谱就是程序、计算机科学家就是 CPU、而做蛋糕的各种原谅都是输入数据。进程就是科学家阅读食谱、取来各种原料以及烘焙蛋糕等一系例了动作的总和。

现在假设科学家的儿子跑过来告诉他，说他的头被蜜蜂蜇了一下，那么此时科学家会记录出来他做蛋糕这个过程到了哪一步，然后拿出急救手册，按照上面的步骤给他儿子实施救助。这里，会涉及到进程之间的切换，科学家（CPU）会从做蛋糕（进程）切换到实施医疗救助（另一个进程）。等待伤口处理完毕后，科学家会回到刚刚记录做蛋糕的那一步，继续制作。

这里的关键思想是 `认识到一个进程所需的条件`，进程是某一类特定活动的总和，它有程序、输入输出以及状态。单个处理器可以被若干进程共享，它使用某种调度算法决定何时停止一个进程的工作，并转而为另外一个进程提供服务。另外需要注意的是，如果一个进程运行了两遍，则被认为是两个进程。那么我们了解到进程模型后，那么进程是如何创建的呢?

进程的创建

操作系统需要一些方式来创建进程。下面是一些创建进程的方式

- 系统初始化（init）
- 正在运行的程序执行了创建进程的系统调用（比如 fork）
- 用户请求创建一个新进程
- 初始化一个批处理工作

系统初始化

启动操作系统时，通常会创建若干个进程。其中有些是 `前台进程(numerous processes)` ，也就是同用户进行交互并替他们完成工作的进程。一些运行在后台，并不与特定的用户进行交互，例如，设计一个进程来接收发来的电子邮件，这个进程大部分的时间都在休眠，但是只要邮件到来后这个进程就会被唤醒。还可以设计一个进程来接收对该计算机上网页的传入请求，在请求到达的进程唤醒来处理网页的传入请求。进程运行在后台用来处理一些活动像是 e-mail，web 网页，新闻，打印等等被称为 `守护进程(daemons)` 。大型系统会有很多守护进程。在 UNIX 中， `ps` 程序可以列出正在运行的进程， 在 Windows 中，可以使用任务管理器。

系统调用创建

除了在启动阶段创建进程之外，一些新的进程也可以在后面创建。通常，一个正在运行的进程会发出 `系统调用` 用来创建一个或多个新进程来帮助其完成工作。例如，如果有大量的数据需要经过网络调取并进行顺序处理，那么创建一个进程读数据，并把数据放到共享缓冲区中，而让第二个进程取走并正确处理会比较容易些。在多处理器中，让每个进程运行在不同的 CPU 上也可以使工作做的更快。

用户请求创建

在许多交互式系统中，输入一个命令或者双击图标就可以启动程序，以上任意一种操作都可以选择开启一个新的进程，在基本的 UNIX 系统中运行 X，新进程将接管启动它的窗口。在 Windows 中启动进程时，它一般没有窗口，但是它可以创建一个或多个窗口。每个窗口都可以运行进程。通过鼠标或者命令就可以切换窗口并与进程进行交互。

> 交互式系统是以人与计算机之间大量交互为特征的计算机系统，比如游戏、web浏览器，IDE 等集成开发环境。

批处理创建

最后一种创建进程的情形会在 `大型机的批处理系统` 中应用。用户在这种系统中提交批处理作业。当操作系统决定它有资源来运行另一个任务时，它将创建一个新进程并从其中的输入队列中运行下一个作业。

从技术上讲，在所有这些情况下，让现有流程执行流程是通过创建系统调用来创建新流程的。该进程可能是正在运行的用户进程，是从键盘或鼠标调用的系统进程或批处理程序。这些就是系统调用创建新进程的过程。该系统调用告诉操作系统创建一个新进程，并直接或间接指示在其中运行哪个程序。

在 UNIX 中，仅有一个系统调用来创建一个新的进程，这个系统调用就是 `fork` 。这个调用会创建一个与调用进程相关的副本。在 fork 后，一个父进程和子进程会有相同的内存映像，相同的环境字符串和相同的打开文件。通常，子进程会执行 `execve` 或者一个简单的系统调用来改变内存映像并运行一个新的程序。例如，当一个用户在 shell 中输出 sort 命令时，shell 会 fork 一个子进程然后子进程去执行 sort 命令。这两步过程的原因是允许子进程在 fork 之后但在 execve 之前操作其文件描述符，以完成标准输入，标准输出和标准错误的重定向。

在 Windows 中，情况正相反，一个简单的 Win32 功能调用 `CreateProcess` ，会处理流程创建并将正确的程序加载到新的进程中。这个调用会有 10 个参数，包括了需要执行的程序、输入给程序的命令行参数、各种安全属性、有关打开的文件是否继承控制位、优先级信息、进程所需要创建的窗口规格以及指向一个结构的指针，在该结构中新创建进程的信息被返回给调用者。除了 `CreateProcess` Win32 中大概有 100 个其他的函数用于处理进程的管理，同步以及相关的事务。下面是 UNIX 操作系统和 Windows 操作系统系统调用的对比

UNIX	Win32	说明
fork	CreateProcess	创建一个新进程
waitpid	WaitForSingleObject	等待一个进程退出
execve	none	CraeteProcess = fork + servvice
exit	ExitProcess	终止执行
open	CreateFile	创建一个文件或打开一个已有的文件
close	CloseHandle	关闭文件
read	ReadFile	从单个文件中读取数据
write	WriteFile	向单个文件写数据
lseek	SetFilePointer	移动文件指针
stat	GetFileAttributesEx	获得不同的文件属性
mkdir	CreateDirectory	创建一个新的目录
rmdir	RemoveDirectory	移除一个空的目录
link	none	Win32 不支持 link
unlink	DeleteFile	销毁一个已有的文件
mount	none	Win32 不支持 mount
umount	none	Win32 不支持 mount，所以也不支持mount
chdir	SetCurrentDirectory	切换当前工作目录
chmod	none	Win32 不支持安全
kill	none	Win32 不支持信号
time	GetLocalTime	获取当前时间

在 UNIX 和 Windows 中，进程创建之后，父进程和子进程有各自不同的地址空间。如果其中某个进程在其地址空间中修改了一个词，这个修改将对另一个进程不可见。在 UNIX 中，子进程的地址空间是父进程的一个拷贝，但是确是两个不同的地址空间；不可写的内存区域是共享的。某些 UNIX 实现是正是在两者之间共享，因为它不能被修改。或者，子进程共享父进程的所有内存，但是这种情况下内存通过 `写时复制(copy-on-write)` 共享，这意味着一旦两者之一想要修改部分内存，则这块内存首先被明确的复制，以确保修改发生在私有内存区域。再次强调，**可写的内存是不能被共享的**。但是，对于一个新创建的进程来说，确实有可能共享创建者的资源，比如可以共享打开的文件。**在 Windows 中，从一开始父进程的地址空间和子进程的地址空间就是不同的**。

进程的终止

进程在创建之后，它就开始运行并做完成任务。然而，没有什么事儿是永不停歇的，包括进程也一样。进程早晚会发生终止，但是通常是由于以下情况触发的

- 正常退出(自愿的)
- 错误退出(自愿的)
- 严重错误(非自愿的)
- 被其他进程杀死(非自愿的)

正常退出

多数进程是由于完成了工作而终止。当编译器完成了所给定程序的编译之后，编译器会执行一个系统调用告诉操作系统它完成了工作。这个调用在 UNIX 中是 `exit` ，在 Windows 中是 `ExitProcess` 。面向屏幕中的软件也支持自愿终止操作。字处理软件、Internet 浏览器和类似的程序中总有一个供用户点击的图标或菜单项，用来通知进程删除它锁打开的任何临时文件，然后终止。

错误退出

进程发生终止的第二个原因是发现严重错误，例如，如果用户执行如下命令

```
cc foo.c
```

为了能够编译 foo.c 但是该文件不存在，于是编译器就会发出声明并退出。在给出了错误参数时，面向屏幕的交互式进程通常并不会直接退出，因为这从用户的角度来说并不合理，用户需要知道发生了什么并想要进行重试，所以这时候应用程序通常会弹出一个对话框告知用户发生了系统错误，是需要重试还是退出。

严重错误

进程终止的第三个原因是由进程引起的错误，通常是由于程序中的错误所导致的。例如，执行了一条非法指令，引用不存在的内存，或者除数是 0 等。在有些系统比如 UNIX 中，进程可以通知操作系统，它希望自行处理某种类型的错误，在这类错误中，进程会收到信号（中断），而不是在这类错误出现时直接终止进程。

被其他进程杀死

第四个终止进程的原因是，某个进程执行系统调用告诉操作系统杀死某个进程。在 UNIX 中，这个系统调用是 kill。在 Win32 中对应的函数是 `TerminateProcess` （注意不是系统调用）。

进程的层次结构

在一些系统中，当一个进程创建了其他进程后，父进程和子进程就会以某种方式进行关联。子进程它自己就会创建更多进程，从而形成一个进程层次结构。

UNIX 进程体系

在 UNIX 中，进程和它的所有子进程以及子进程的子进程共同组成一个进程组。当用户从键盘中发出一个信号后，该信号被发送给当前与键盘相关的进程组中的所有成员（它们通常是在当前窗口创建的所有活动进程）。每个进程可以分别捕获该信号、忽略该信号或采取默认的动作，即被信号 kill 掉。

这里有另一个例子，可以用来说明层次的作用，考虑 `UNIX` 在启动时如何初始化自己。一个称为 `init` 的特殊进程出现在启动映像中 。当 init 进程开始运行时，它会读取一个文件，文件会告诉它有多少个终端。然后为每个终端创建一个新进程。这些进程等待用户登录。如果登录成功，该登录进程就执行一个 shell 来等待接收用户输入指令，这些命令可能会启动更多的进程，以此类推。因此，整个操作系统中所有的进程都隶属于一个单个以 init 为根的进程树。

```
1 init-+-apmd
       |-atd
       |-cron
       ...
5      |-dhclient
       |-firefox-bin-+-firefox-bin---2*[firefox-bin]
       |             |-java_vm---java_vm---13*[java_vm]
       |             `-swf_play
```

Windows 进程体系

相反，Windows 中没有进程层次的概念，Windows 中所有进程都是平等的，唯一类似于层次结构的是在创建进程的时候，父进程得到一个特别的令牌（称为句柄），该句柄可以用来控制子进程。然而，这个令牌可能也会移交给别的操作系统，这样就不存在层次结构了。而在 UNIX 中，进程不能剥夺其子进程的 `进程权` 。（这样看来，还是 Windows 比较 `渣` ）。

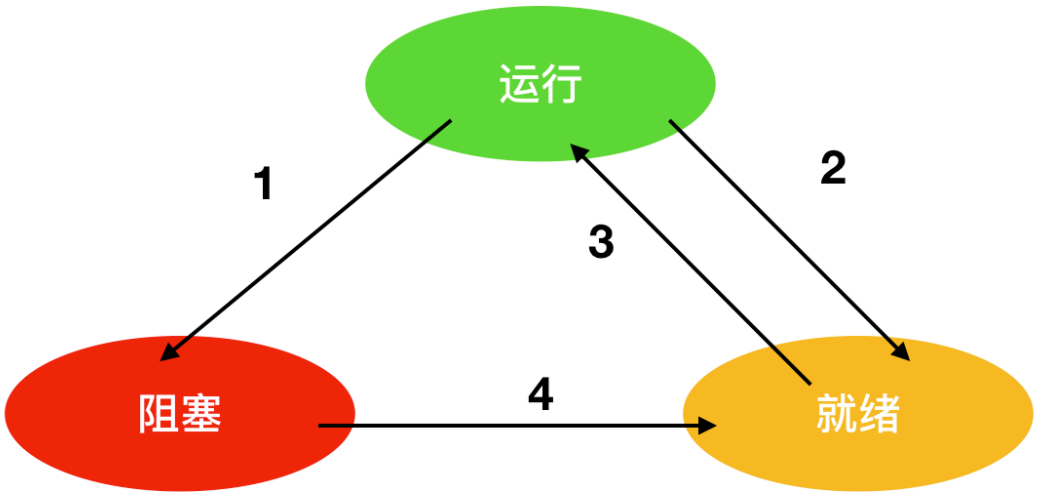
进程状态

尽管每个进程是一个独立的实体，有其自己的程序计数器和内部状态，但是，进程之间仍然需要相互帮助。例如，一个进程的结果可以作为另一个进程的输入，在 shell 命令中

```
1   cat chapter1 chapter2 chapter3 | grep tree
```

第一个进程是 `cat` ，将三个文件级联并输出。第二个进程是 `grep` ，它从输入中选择具有包含关键字 `tree` 的内容，根据这两个进程的相对速度（这取决于两个程序的相对复杂度和各自所分配到的 CPU 时间片），可能会发生下面这种情况， `grep` 准备就绪开始运行，但是输入进程还没有完成，于是必须阻塞 grep 进程，直到输入完毕。

当一个进程开始运行时，它可能会经历下面这几种状态



- 第一种是进程因为等待输入而阻塞
- 第二种是调度程序选择另一个进程
- 第三种是调度程序选择一个进程开始运行
- 第四种是出现有效的输入

进程间的状态切换图

图中会涉及三种状态

1. 运行态，运行态指的就是进程实际占用 CPU 时间片运行时
2. 就绪态，就绪态指的是可运行，但因为其他进程正在运行而处于就绪状态
3. 阻塞态，除非某种外部事件发生，否则进程不能运行

逻辑上来说，运行态和就绪态是很相似的。这两种情况下都表示进程 可运行，但是第二种情况没有获得 CPU 时间分片。第三种状态与前两种状态不同的原因是这个进程不能运行，CPU 空闲时也不能运行。

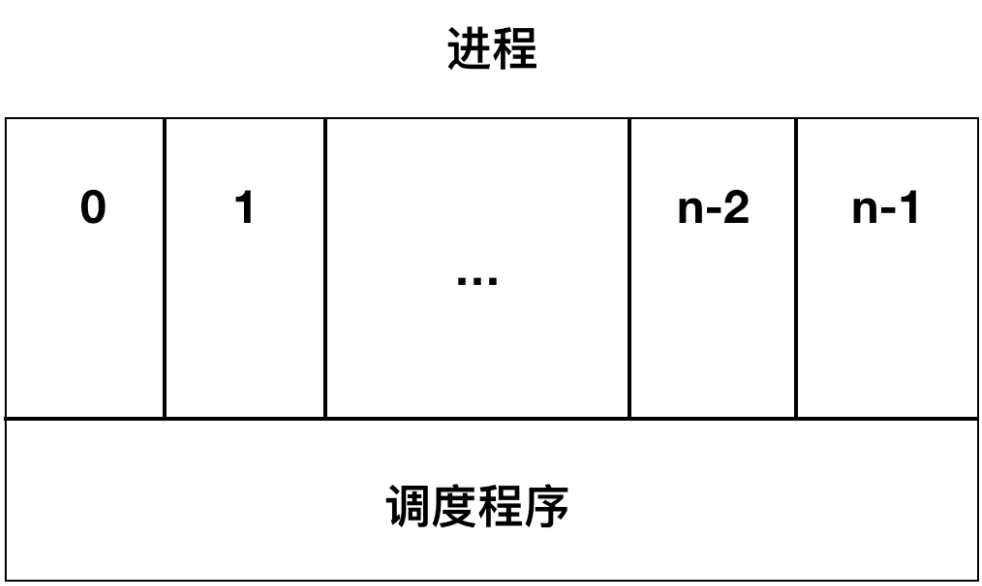
三种状态会涉及四种状态间的切换，在操作系统发现进程不能继续执行时会发生 状态1 的轮转，在某些系统中进程执行系统调用，例如 `pause`，来获取一个阻塞的状态。在其他系统中包括 UNIX，当进程从管道或特殊文件（例如终端）中读取没有可用的输入时，该进程会被自动终止。

转换 2 和转换 3 都是由进程调度程序（操作系统的一部分）引起的，进程本身不知道调度程序的存在。转换 2 的出现说明进程调度器认定当前进程已经运行了足够长的时间，是时候让其他进程运行 CPU 时间片了。当所有其他进程都运行过后，这时候该是让第一个进程重新获得 CPU 时间片的时候了，就会发生转换 3。

> **程序调度指的是，决定哪个进程优先被运行和运行多久，这是很重要的一点**。已经设计出许多算法来尝试平衡系统整体效率与各个流程之间的竞争需求。

当进程等待的一个外部事件发生时（如从外部输入一些数据后），则发生转换 4。如果此时没有其他进程在运行，则立刻触发转换 3，该进程便开始运行，否则该进程会处于就绪阶段，等待 CPU 空闲后再轮到它运行。

从上面的观点引入了下面的模型



基于进程的操作系统中最底层的是中断和调度处理，在该层之上是顺序进程

操作系统最底层的就是调度程序，在它上面有许多进程。所有关于中断处理、启动进程和停止进程的具体细节都隐藏在调度程序中。事实上，调度程序只是一段非常小的程序。

进程的实现

操作系统为了执行进程间的切换，会维护着一张表格，这张表就是 `进程表(process table)` 。每个进程占用一个进程表项。该表项包含了进程状态的重要信息，包括程序计数器、堆栈指针、内存分配状况、所打开文件的状态、账号和调度信息，以及其他在进程由运行态转换到就绪态或阻塞态时所必须保存的信息，从而保证该进程随后能再次启动，就像从未被中断过一样。

下面展示了一个典型系统中的关键字段

进程管理	存储管理	文件管理
寄存器	text segment 的指针	根目录
程序计数器	data segment 的指针	工作目录
程序状态字	stack segment 的指针	文件描述符
堆栈指针		用户ID
进程状态		组 ID
优先级		
调度参数		
进程ID		
父进程		
进程组		
信号		
进程开始时间		
使用的 CPU 时间		
子进程的 CPU 时间		
下次定时器时间		

典型的进程表表项中的一些字段

第一列内容与 `进程管理` 有关，第二列内容与 `存储管理` 有关，第三列内容与 `文件管理` 有关。

存储管理的 text segment 、 data segment、stack segment 更多了解见下面这篇文章

程序员需要了解的硬核知识之汇编语言(全)

现在我们应该对进程表有个大致的了解了，就可以在对单个 CPU 上如何运行多个顺序进程的错觉做更多的解释。与每一 I/O 类相关联的是一个称作 `中断向量(interrupt vector)` 的位置（靠近内存底部的固定区域）。它包含中断服务程序的入口地址。假设当一个磁盘中断发生时，用户进程 3 正在运行，则中断硬件将程序计数器、程序状态字、有时还有一个或多个寄存器压入堆栈，计算机随即跳转到中断向量所指示的地址。这就是硬件所做的事情。然后软件就随即接管一切剩余的工作。

当中断结束后，操作系统会调用一个 C 程序来处理中断剩下的工作。在完成剩下的工作后，会使某些进程就绪，接着调用调度程序，决定随后运行哪个进程。然后将控制权转移给一段汇编语言代码，为当前的进程装入寄存器值以及内存映射并启动该进程运行，下面显示了中断处理和调度的过程。

1. 硬件压入堆栈程序计数器等
2. 硬件从中断向量装入新的程序计数器
3. 汇编语言过程保存寄存器的值
4. 汇编语言过程设置新的堆栈
5. C 中断服务器运行（典型的读和缓存写入）
6. 调度器决定下面哪个程序先运行
7. C 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

一个进程在执行过程中可能被中断数千次，但关键每次中断后，被中断的进程都返回到与中断发生前完全相同的状态。

线程

在传统的操作系统中，每个进程都有一个地址空间和一个控制线程。事实上，这是大部分进程的定义。不过，在许多情况下，经常存在同一地址空间中运行多个控制线程的情形，这些线程就像是分离的进程。下面我们就着重探讨一下什么是线程

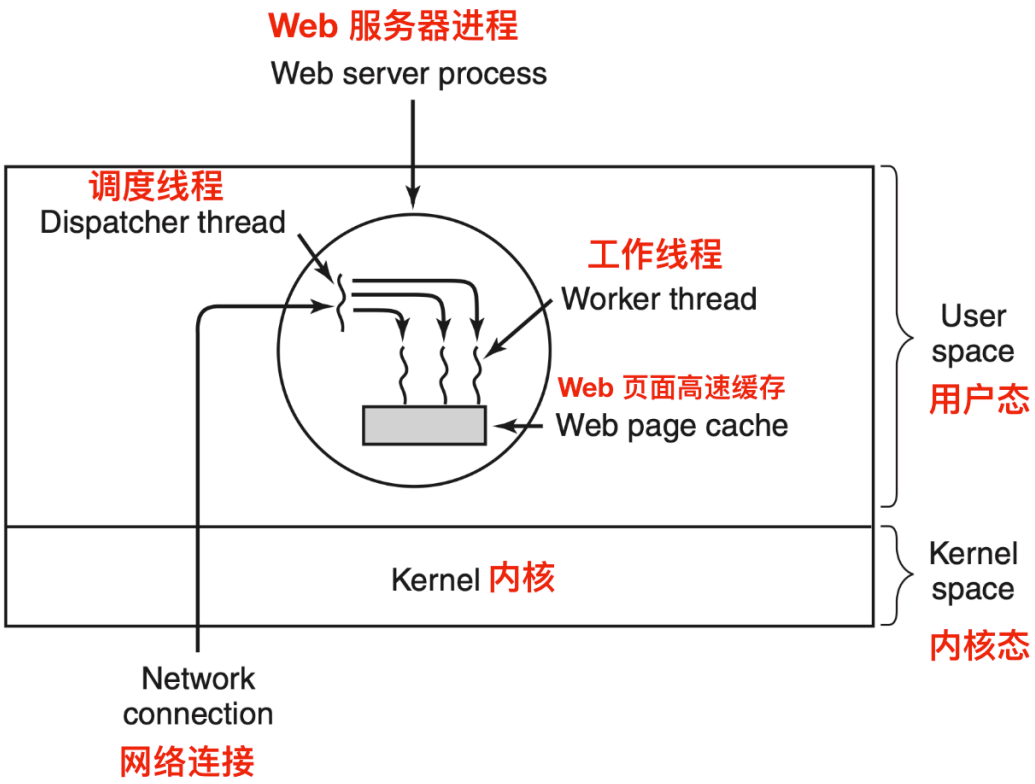
线程的使用

或许这个疑问也是你的疑问，为什么要在进程的基础上再创建一个线程的概念，准确的说，这其实是进程模型和线程模型的讨论，回答这个问题，可能需要分三步来回答

- 多线程之间会共享同一块地址空间和所有可用数据的能力，这是进程所不具备的
- 线程要比进程 `更轻量级` ，由于线程更轻，所以它比进程更容易创建，也更容易撤销。在许多系统中，创建一个线程要比创建一个进程快 10 - 100 倍。
- 第三个原因可能是性能方面的探讨，如果多个线程都是 CPU 密集型的，那么并不能获得性能上的增强，但是如果存在着大量的计算和大量的 I/O 处理，拥有多个线程能在这些活动中彼此重叠进行，从而会加快应用程序的执行速度

多线程解决方案

现在考虑一个线程使用的例子：一个万维网服务器，对页面的请求发送给服务器，而所请求的页面发送回客户端。在多数 web 站点上，某些页面较其他页面相比有更多的访问。例如，索尼的主页比任何一个照相机详情介绍页面具有更多的访问，Web 服务器可以把获得大量访问的页面集合保存在内存中，避免到磁盘去调入这些页面，从而改善性能。这种页面的集合称为 `高速缓存(cache)` ，高速缓存也应用在许多场合中，比如说 CPU 缓存。



一个多线程的 web 服务器

上面是一个 web 服务器的组织方式，一个叫做 `调度线程(dispatcher thread)` 的线程从网络中读入工作请求，在调度线程检查完请求后，它会选择一个空闲的（阻塞的）工作线程来处理请求，通常是将消息的指针写入到每个线程关联的特殊字中。然后调度线程会唤醒正在睡眠中的工作线程，把工作线程的状态从阻塞态变为就绪态。

当工作线程启动后，它会检查请求是否在 web 页面的高速缓存中存在，这个高速缓存是所有线程都可以访问的。如果高速缓存不存在这个 web 页面的话，它会调用一个 `read` 操作从磁盘中获取页面并且阻塞线程直到磁盘操作完成。当线程阻塞在硬盘操作的期间，为了完成更多的工作，调度线程可能挑选另一个线程运行，也可能把另一个当前就绪的工作线程投入运行。

这种模型允许将服务器编写为顺序线程的集合，在分派线程的程序中包含一个死循环，该循环用来获得工作请求并且把请求派给工作线程。每个工作线程的代码包含一个从调度线程接收的请求，并且检查 web 高速缓存中是否存在所需页面，如果有，直接把该页面返回给客户，接着工作线程阻塞，等待一个新请求的到达。如果没有，工作线程就从磁盘调入该页面，将该页面返回给客户机，然后工作线程阻塞，等待一个新请求。

下面是调度线程和工作线程的代码，这里假设 TRUE 为常数 1 ， buf 和 page 分别是保存工作请求和 Web 页面的相应结构。

调度线程的大致逻辑

```
while(TRUE){
  get_next_request(&buf);
  handoff_work(&buf);
}
```

工作线程的大致逻辑

```
while(TRUE){
  wait_for_work(&buf);
  look_for_page_in_cache(&buf,&page);
  if(page_not_in_cache(&page)){
    read_page_from_disk(&buf,&page);
  }
  return _page(&page);
}
```

单线程解决方案

现在考虑没有多线程的情况下，如何编写 Web 服务器。我们很容易的就想象为单个线程了，Web 服务器的主循环获取请求并检查请求，并争取在下一个请求之前完成工作。在等待磁盘操作时，服务器空转，并且不处理任何到来的其他请求。结果会导致每秒中只有很少的请求被处理，所以这个例子能够说明多线程提高了程序的并行性并提高了程序的性能。

状态机解决方案

到现在为止，我们已经有了两种解决方案，单线程解决方案和多线程解决方案，其实还有一种解决方案就是 `状态机解决方案` ，它的流程如下

如果目前只有一个非阻塞版本的 read 系统调用可以使用，那么当请求到达服务器时，这个唯一的 read 调用的线程会进行检查，如果能够从高速缓存中得到响应，那么直接返回，如果不能，则启动一个非阻塞的磁盘操作

服务器在表中记录当前请求的状态，然后进入并获取下一个事件，紧接着下一个事件可能就是一个新工作的请求或是磁盘对先前操作的回答。如果是新工作的请求，那么就开始处理请求。如果是磁盘的响应，就从表中取出对应的状态信息进行处理。对于非阻塞式磁盘 I/O 而言，这种响应一般都是信号中断响应。

每次服务器从某个请求工作的状态切换到另一个状态时，都必须显示的保存或者重新装入相应的计算状态。这里，每个计算都有一个被保存的状态，存在一个会发生且使得相关状态发生改变的事件集合，我们把这类设计称为 `有限状态机(finite-state machine)`，有限状态机杯广泛的应用在计算机科学中。

这三种解决方案各有各的特性，多线程使得顺序进程的思想得以保留下来，并且实现了并行性，但是顺序进程会阻塞系统调用；单线程服务器保留了阻塞系统的简易性，但是却放弃了性能。有限状态机的处理方法运用了非阻塞调用和中断，通过并行实现了高性能，但是给编程增加了困难。

模型	特性
单线程	无并行性，性能较差，阻塞系统调用
多线程	有并行性，阻塞系统调用
有限状态机	并行性，非阻塞系统调用、中断

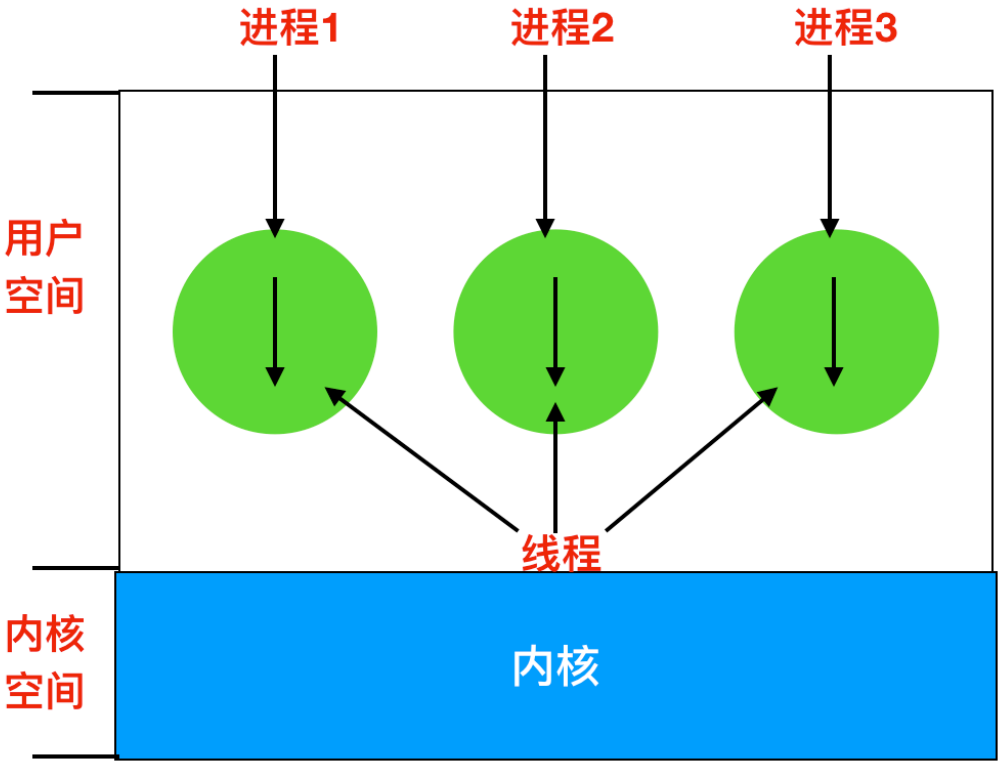
经典的线程模型

理解进程的另一个角度是，用某种方法把相关的资源集中在一起。进程有存放程序正文和数据以及其他资源的地址空间。这些资源包括打开的文件、子进程、即将发生的定时器、信号处理程序、账号信息等。把这些信息放在进程中会比较容易管理。

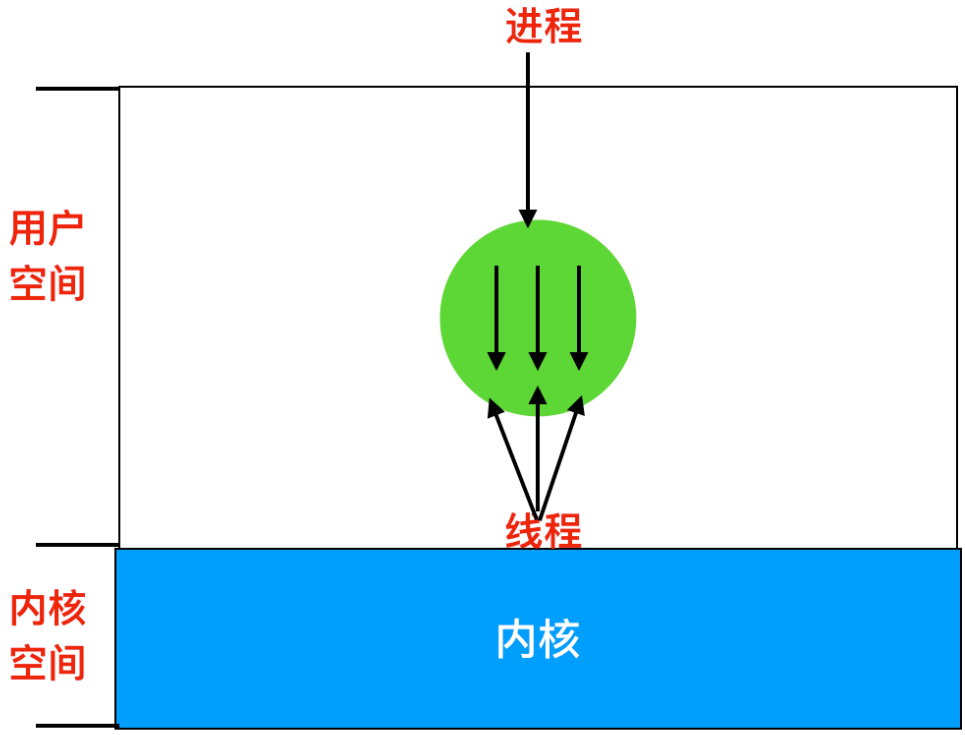
另一个概念是，进程中拥有一个执行的线程，通常简写为 `线程(thread)`。线程会有程序计数器，用来记录接着要执行哪一条指令；线程还拥有寄存器，用来保存线程当前正在使用的变量；线程还会有堆栈，用来记录程序的执行路径。尽管线程必须在某个进程中执行，但是进程和线程完完全全是两个不同的概念，并且他们可以分开处理。进程用于把资源集中在一起，而线程则是 CPU 上调度执行的实体。

线程给进程模型增加了一项内容，即在同一个进程中，允许彼此之间有较大的独立性且互不干扰。在一个进程中并行运行多个线程类似于在一台计算机上运行多个进程。在多个线程中，各个线程共享同一地址空间和其他资源。在多个进程中，进程共享物理内存、磁盘、打印机和其他资源。因为线程会包含有一些进程的属性，所以线程被称为 `轻量的进程(lightweight processes)`。 `多线程(multithreading)` 一词还用于描述在同一进程中多个线程的情况。

下图我们可以看到三个传统的进程，每个进程有自己的地址空间和单个控制线程。每个线程都在不同的地址空间中运行



下图中，我们可以看到有一个进程三个线程的情况。每个线程都在相同的地址空间中运行。



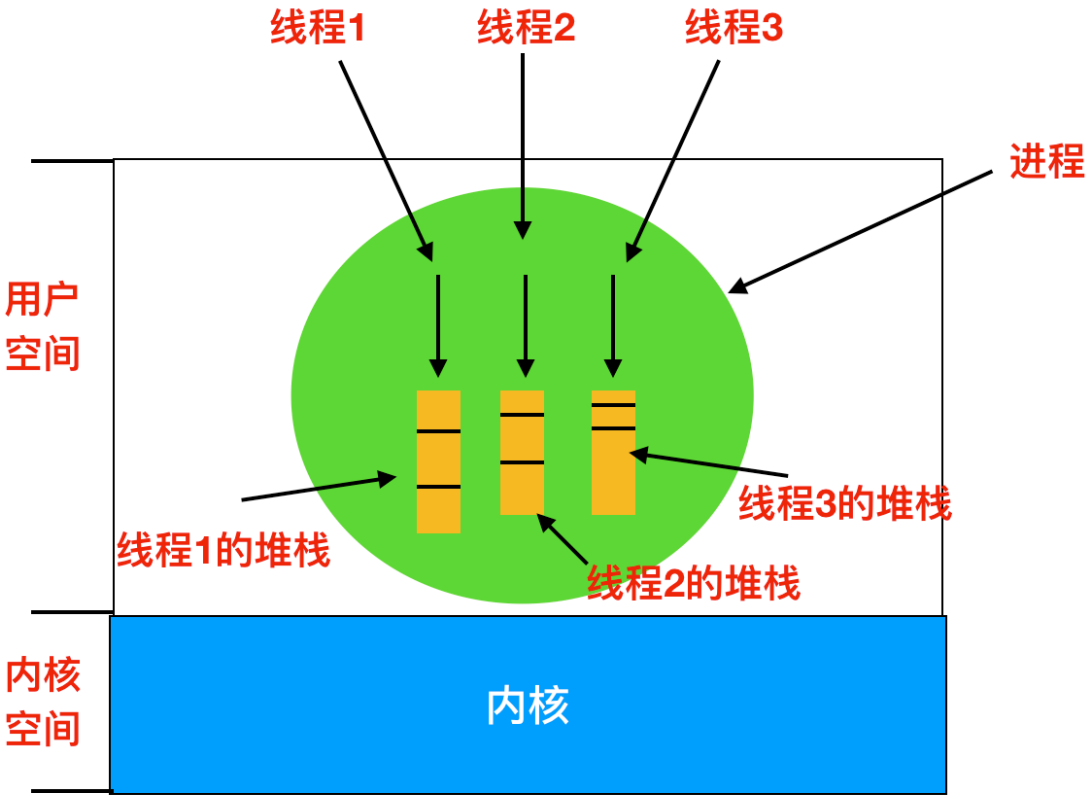
线程不像是进程那样具备较强的独立性。同一个进程中的所有线程都会有完全一样的地址空间，这意味着它们也共享同样的全局变量。由于每个线程都可以访问进程地址空间内每个内存地址，**因此一个线程可以读取、写入甚至擦除另一个线程的堆栈**。线程之间除了共享同一内存空间外，还具有如下不同的内容

每个进程中的内容	每个线程中的内容
地址空间	程序计数器
全局变量	寄存器
打开文件	堆栈
子进程	状态
即将发生的定时器	
信号与信号处理程序	
账户信息	

上图左边的是同一个进程中 `每个线程共享` 的内容，上图右边是 `每个线程` 中的内容。也就是说左边的列表是进程的属性，右边的列表是线程的属性。

和进程一样，线程可以处于下面这几种状态：**运行中、阻塞、就绪和终止（进程图中没有画）**。正在运行的线程拥有 CPU 时间片并且状态是运行中。一个被阻塞的线程会等待某个释放它的事件。例如，当一个线程执行从键盘读入数据的系统调用时，该线程就被阻塞直到有输入为止。线程通常会被阻塞，直到它等待某个外部事件的发生或者有其他线程来释放它。**线程之间的状态转换和进程之间的状态转换是一样的**。

每个线程都会有自己的堆栈，如下图所示



线程系统调用

进程通常会从当前的某个单线程开始，然后这个线程通过调用一个库函数（比如 `thread_create`）创建新的线程。线程创建的函数会要求指定新创建线程的名称。创建的线程通常都返回一个线程标识符，该标识符就是新线程的名字。

当一个线程完成工作后，可以通过调用一个函数（比如 `thread_exit`）来退出。紧接着线程消失，状态变为终止，不能再进行调度。在某些线程的运行过程中，可以通过调用函数例如 `thread_join`，表示一个线程可以等待另一个线程退出。这个过程阻塞调用线程直到等待特定的线程退出。在这种情况下，线程的创建和终止非常类似于进程的创建和终止。

另一个常见的线程是调用 `thread_yield`，它允许线程自动放弃 CPU 从而让另一个线程运行。这样一个调用还是很重要的，因为不同于进程，线程是无法利用时钟中断强制让线程让出 CPU 的。

POSIX 线程

为了使编写可移植线程程序成为可能，IEEE 在 IEEE 标准 1003.1c 中定义了线程标准。线程包被定义为 `Pthreads`。大部分的 UNIX 系统支持它。这个标准定义了 60 多种功能调用，一一列举不太现实，下面为你列举了一些常用的系统调用。

> **POSIX线程**（通常称为**pthreads**）是一种独立于语言而存在的执行模型，以及并行执行模型。它允许程序控制时间上重叠的多个不同的工作流程。每个工作流程都称为一个线程，可以通过调用POSIX Threads API来实现对这些流程的创建和控制。可以把它理解为线程的标准。
>
> POSIX Threads 的实现在许多类似且符合POSIX的操作系统上可用，例如 **FreeBSD**、**NetBSD**、**OpenBSD**、**Linux**、**macOS**、**Android**、**Solaris**，它在现有 Windows API 之上实现了 **pthread**。
>
> IEEE 是世界上最大的技术专业组织，致力于为人类的利益而发展技术。

线程调用	描述
pthread_create	创建一个新线程
pthread_exit	结束调用的线程
pthread_join	等待一个特定的线程退出
pthread_yield	释放 CPU 来运行另外一个线程
pthread_attr_init	创建并初始化一个线程的属性结构
pthread_attr_destory	删除一个线程的属性结构

所有的 Pthreads 都有特定的属性，每一个都含有标识符、一组寄存器（包括程序计数器）和一组存储在结构中的属性。这个属性包括堆栈大小、调度参数以及其他线程需要的项目。

新的线程会通过 `pthread_create` 创建，新创建的线程的标识符会作为函数值返回。这个调用非常像是 UNIX 中的 `fork` 系统调用（除了参数之外），其中线程标识符起着 `PID` 的作用，这么做的目的是为了和其他线程进行区分。

当线程完成指派给他的工作后，会通过 `pthread_exit` 来终止。这个调用会停止线程并释放堆栈。

一般一个线程在继续运行前需要等待另一个线程完成它的工作并退出。可以通过 `pthread_join` 线程调用来等待别的特定线程的终止。而要等待线程的线程标识符作为一个参数给出。

有时会出现这种情况：一个线程逻辑上没有阻塞，但感觉上它已经运行了足够长的时间并且希望给另外一个线程机会去运行。这时候可以通过 `pthread_yield` 来完成。

下面两个线程调用是处理属性的。 `pthread_attr_init` 建立关联一个线程的属性结构并初始化成默认值，这些值（例如优先级）可以通过修改属性结构的值来改变。

最后， `pthread_attr_destroy` 删除一个线程的结构，释放它占用的内存。它不会影响调用它的线程，这些线程会一直存在。

为了更好的理解 pthread 是如何工作的，考虑下面这个例子

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print_hello_world(vvoid *tid){
  /* 输出线程的标识符，然后退出 */
  printf("Hello World. Greetings from thread %d\n",tid);
  pthread_exit(NULL);
}

int main(int argc,char *argv[]){
  /* 主程序创建 10 个线程，然后退出 */
  pthread_t threads[NUMBER_OF_THREADS];
  int status,i;

  for(int i = 0;i < NUMBER_OF_THREADS;i++){
    printf("Main here. Creating thread %d\n",i);
    status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

    if(status != 0){
      printf("Oops. pthread_create returned error code %d\n",status);
      exit(-1);
    }
  }
  exit(NULL);
}
```

主线程在宣布它的指责之后，循环 `NUMBER_OF_THREADS` 次，每次创建一个新的线程。如果线程创建失败，会打印出一条信息后退出。在创建完成所有的工作后，主程序退出。

线程实现

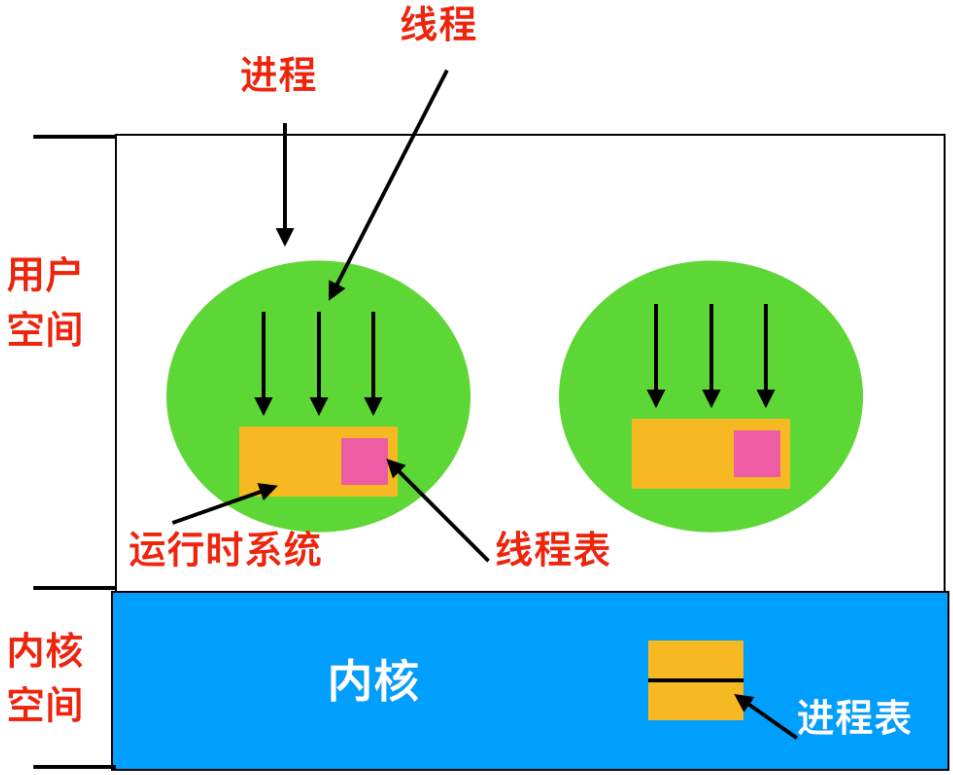
主要有三种实现方式

- 在用户空间中实现线程；
- 在内核空间中实现线程；
- 在用户和内核空间中混合实现线程。

下面我们分开讨论一下

在用户空间中实现线程

第一种方法是把整个线程包放在用户空间中，内核对线程一无所知，它不知道线程的存在。所有的这类实现都有同样的通用结构



在用户空间中实现多线程

线程在运行时系统之上运行，运行时系统是管理线程过程的集合，包括前面提到的四个过程：pthread_create, pthread_exit, pthread_join 和 pthread_yield。

> `运行时系统(Runtime System)` 也叫做运行时环境，该运行时系统提供了程序在其中运行的环境。此环境可能会解决许多问题，包括应用程序内存的布局，程序如何访问变量，在过程之间传递参数的机制，与操作系统的接口等等。编译器根据特定的运行时系统进行假设以生成正确的代码。通常，运行时系统将负责设置和管理堆栈，并且会包含诸如垃圾收集，线程或语言内置的其他动态的功能。

在用户空间管理线程时，每个进程需要有其专用的 `线程表(thread table)` ，用来跟踪该进程中的线程。这些表和内核中的进程表类似，不过它仅仅记录各个线程的属性，如每个线程的程序计数器、堆栈指针、寄存器和状态。该线程标由运行时系统统一管理。当一个线程转换到就绪状态或阻塞状态时，在该线程表中存放重新启动该线程的所有信息，与内核在进程表中存放的信息完全一样。

在用户空间实现线程的优势

在用户空间中实现线程要比在内核空间中实现线程具有这些方面的优势：考虑如果在线程完成时或者是在调用 `pthread_yield` 时，必要时会进程线程切换，然后线程的信息会被保存在运行时环境所提供的线程表中，然后，线程调度程序来选择另外一个需要运行的线程。保存线程的状态和调度程序都是 `本地过程` ，**所以启动他们比进行内核调用效率更高。因而不需要切换到内核，也就不需要上下文切换，也**

不需要对内存高速缓存进行刷新，因为线程调度非常便捷，因此效率比较高。

在用户空间实现线程还有一个优势就是**它允许每个进程有自己定制的调度算法**。例如在某些应用程序中，那些具有垃圾收集线程的应用程序（知道是谁了吧）就不用担心自己线程会不会在不合适的时候停止，这是一个优势。用户线程还具有较好的可扩展性，因为内核空间中的内核线程需要一些表空间和堆栈空间，如果内核线程数量比较大，容易造成问题。

在用户空间实现线程的劣势

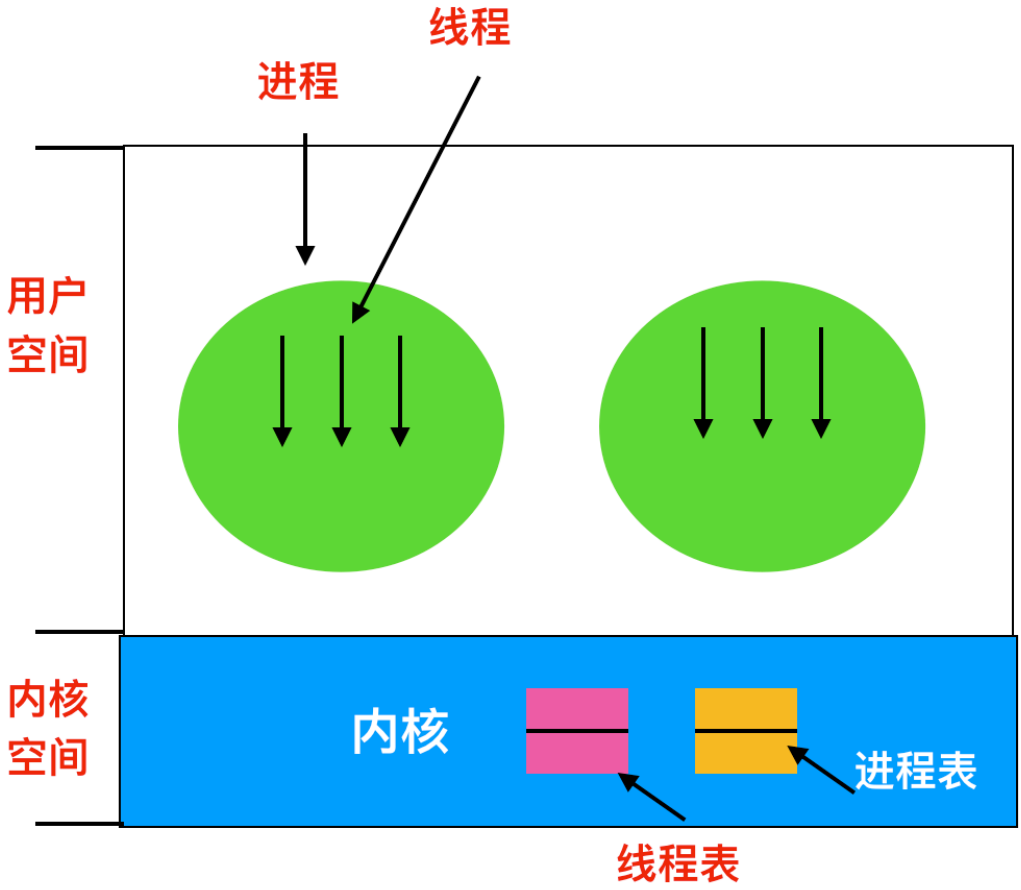
尽管在用户空间实现线程会具有一定的性能优势，但是劣势还是很明显的，你如何实现 `阻塞系统调用` 呢？假设在还没有任何键盘输入之前，一个线程读取键盘，让线程进行系统调用是不可能的，因为这会停止所有的线程。所以，**使用线程的一个目标是能够让线程进行阻塞调用，并且要避免被阻塞的线程影响其他线程**。

与阻塞调用类似的问题是 `缺页中断` 问题，实际上，计算机并不会把所有的程序都一次性的放入内存中，如果某个程序发生函数调用或者跳转指令到了一条不在内存的指令上，就会发生页面故障，而操作系统将到磁盘上取回这个丢失的指令，这就称为 `缺页故障` 。而在对所需的指令进行读入和执行时，相关的进程就会被阻塞。如果只有一个线程引起页面故障，内核由于甚至不知道有线程存在，通常会吧整个进程阻塞直到磁盘 I/O 完成为止，尽管其他的线程是可以运行的。

另外一个问题是，如果一个线程开始运行，该线程所在进程中的其他线程都不能运行，除非第一个线程自愿的放弃 CPU，在一个单进程内部，没有时钟中断，所以不可能使用轮转调度的方式调度线程。除非其他线程能够以自己的意愿进入运行时环境，否则调度程序没有可以调度线程的机会。

在内核中实现线程

现在我们考虑使用内核来实现线程的情况，此时不再需要运行时环境了。另外，每个进程中也没有线程表。相反，在内核中会有用来记录系统中所有线程的线程表。当某个线程希望创建一个新线程或撤销一个已有线程时，它会进行一个系统调用，这个系统调用通过对线程表的更新来完成线程创建或销毁工作。



在内核中实现多线程

内核中的线程表持有每个线程的寄存器、状态和其他信息。这些信息和用户空间中的线程信息相同，但是位置却被放在了内核中而不是用户空间中。另外，内核还维护了一张进程表用来跟踪系统状态。

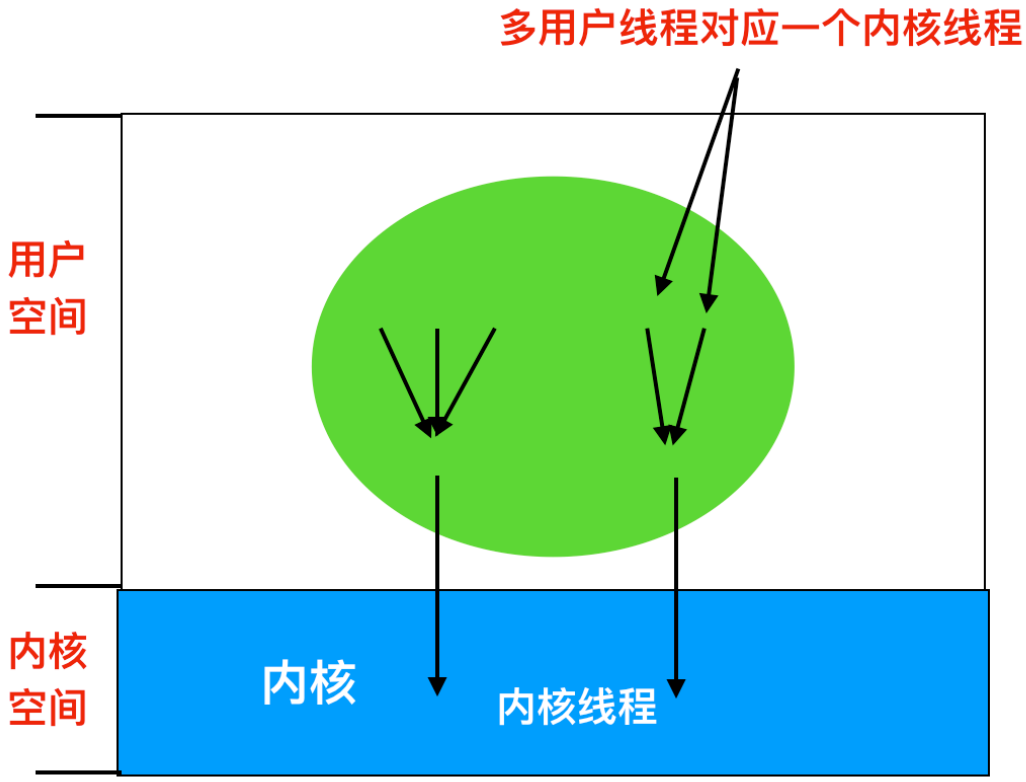
所有能够阻塞的调用都会通过系统调用的方式来实现，当一个线程阻塞时，内核可以进行选择，是运行在同一个进程中的另一个线程（如果有就绪线程的话）还是运行一个另一个进程中的线程。但是在用户实现中，运行时系统始终运行自己的线程，直到内核剥夺它的 CPU 时间片（或者没有可运行的线程存在了）为止。

由于在内核中创建或者销毁线程的开销比较大，所以某些系统会采用可循环利用的方式来回收线程。当某个线程被销毁时，就把它标志为不可运行的状态，但是其内部结构没有受到影响。稍后，在必须创建一个新线程时，就会重新启用旧线程，把它标志为可用状态。

如果某个进程中的线程造成缺页故障后，内核很容易的就能检查出来是否有其他可运行的线程，如果有的话，在等待所需要的页面从磁盘读入时，就选择一个可运行的线程运行。这样做的缺点是系统调用的代价比较大，所以如果线程的操作（创建、终止）比较多，就会带来很大的开销。

混合实现

结合用户空间和内核空间的优点，设计人员采用了一种 `内核级线程` 的方式，然后将用户级线程与某些或者全部内核线程多路复用起来



用户线程与内核线程的多路复用

在这种模型中，编程人员可以自由控制用户线程和内核线程的数量，具有很大的灵活度。采用这种方法，内核只识别内核级线程，并对其进行调度。其中一些内核级线程会被多个用户级线程多路复用。

进程间通信

进程是需要频繁的和其他进程进行交流的。例如，在一个 shell 管道中，第一个进程的输出必须传递给第二个进程，这样沿着管道进行下去。因此，进程之间如果需要通信的话，必须要使用一种良好的数据结构以至于不能被中断。下面我们会一起讨论有关 `进程间通信(Inter Process Communication, IPC)` 的问题。

关于进程间的通信，这里有三个问题

- 上面提到了第一个问题，那就是一个进程如何传递消息给其他进程。
- 第二个问题是如何确保两个或多个线程之间不会相互干扰。例如，两个航空公司都试图为不同的顾客抢购飞机上的最后一个座位。
- 第三个问题是数据的先后顺序的问题，如果进程 A 产生数据并且进程 B 打印数据。则进程 B 打印数据之前需要先等 A 产生数据后才能够进行打印。

需要注意的是，这三个问题中的后面两个问题同样也适用于线程

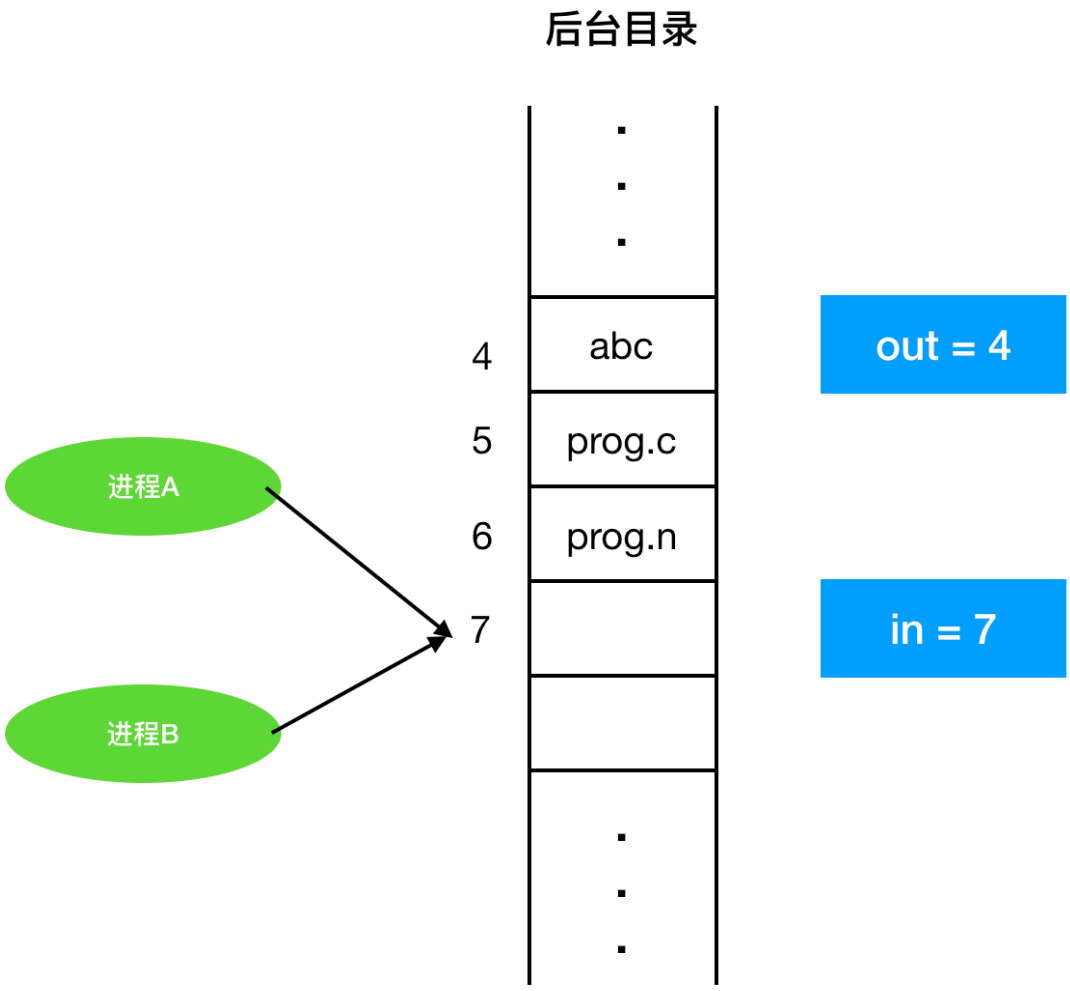
第一个问题在线程间比较好解决，因为它们共享一个地址空间，它们具有相同的运行时环境，可以想象你在用高级语言编写多线程代码的过程中，线程通信问题是不是比较容易解决?

另外两个问题也同样适用于线程，同样的问题可用同样的方法来解决。我们后面会慢慢讨论这三个问题，你现在脑子中大致有个印象即可。

竞态条件

在一些操作系统中，协作的进程可能共享一些彼此都能读写的公共资源。公共资源可能在内存中也可能在一个共享文件。为了讲清楚进程间是如何通信的，这里我们举一个例子：一个后台打印程序。当一个进程需要打印某个文件时，它会将文件名放在一个特殊的 `后台目录(spooler directory)` 中。另一个进程 `打印后台进程(printer daemon)` 会定期的检查是否需要文件被打印，如果有的话，就打印并将该文件名从目录下删除。

假设我们的后台目录有非常多的 `槽位(slot)` ，编号依次为 0，1，2，...，每个槽位存放一个文件名。同时假设有两个共享变量： `out` ，指向下一个需要打印的文件； `in` ，指向目录中下个空闲的槽位。可以把这两个文件保存在一个所有进程都能访问的文件中，该文件的长度为两个字。在某一时刻，0 至 3 号槽位空，4 号至 6 号槽位被占用。在同一时刻，进程 A 和 进程 B 都决定将一个文件排队打印，情况如下



`墨菲法则(Murphy)` 中说过，任何可能出错的地方终将出错，这句话生效时，可能发生如下情况。

进程 A 读到 in 的值为 7，将 7 存在一个局部变量 `next_free_slot` 中。此时发生一次时钟中断，CPU 认为进程 A 已经运行了足够长的时间，决定切换到进程 B 。进程 B 也读取 in 的值，发现是 7，然后进程 B 将 7 写入到自己的局部变量 `next_free_slot` 中，在这一时刻两个进程都认为下一个可用槽位是 7 。

进程 B 现在继续运行，它会将打印文件名写入到 slot 7 中，然后把 in 的指针更改为 8 ，然后进程 B 离开去做其他的事情

现在进程 A 开始恢复运行，由于进程 A 通过检查 `next_free_slot` 也发现 slot 7 的槽位是空的，于是将打印文件名存入 slot 7 中，然后把 in 的值更新为 8 ，由于 slot 7 这个槽位中已经有进程 B 写入的值，所以进程 A 的打印文件名会把进程 B 的文件覆盖，由于打印机内部是无法发现是哪个进程更新的，它的功能比较局限，所以这时候进程 B 永远无法打印输出，类似这种情况，**即两个或多个线程同时对一共享数据进行修改，从而影响程序运行的正确性时，这种就被称为竞态条件(race condition)**。调试竞态条件是一种非常困难的工作，因为绝大多数情况下程序运行良好，但在极少数的情况下会发生一些无法解释的奇怪现象。

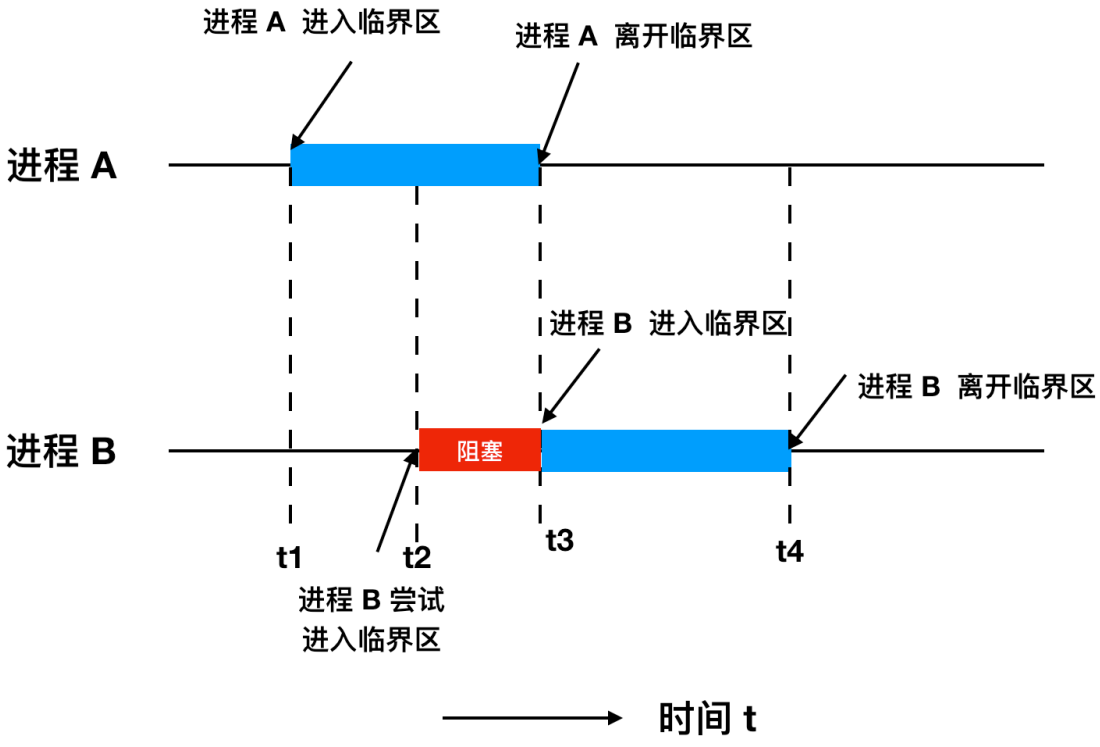
临界区

不仅共享资源会造成竞态条件，事实上共享文件、共享内存也会造成竞态条件、那么该如何避免呢？或许一句话可以概括说明：**禁止一个或多个进程在同一时刻对共享资源（包括共享内存、共享文件等）进行读写**。换句话说，我们需要一种 `互斥(mutual exclusion)` 条件，这也就是说，如果一个进程在某种方式下使用共享变量和文件的话，除该进程之外的其他进程就禁止做这种事（访问统一资源）。上面问题的纠结点在于，在进程 A 对共享变量的使用未结束之前进程 B 就使用它。在任何操作系统中，为了实现互斥操作而选用适当的原语是一个主要的设计问题，接下来我们会着重探讨一下。

避免竞争问题的条件可以用一种抽象的方式去描述。大部分时间，进程都会忙于内部计算和其他不会导致竞争条件的计算。然而，有时候进程会访问共享内存或文件，或者做一些能够导致竞态条件的操作。我们把对共享内存进行访问的程序片段称作 `临界区域(critical region)` 或 `临界区(critical section)` 。如果我们能够正确的操作，使两个不同进程不可能同时处于临界区，就能避免竞争条件，这也是从操作系统设计角度来进行的。

尽管上面这种设计避免了竞争条件，但是不能确保并发线程同时访问共享数据的正确性和高效性。一个好的解决方案，应该包含下面四种条件

1. 任何时候两个进程不能同时处于临界区
2. 不应对 CPU 的速度和数量做任何假设
3. 位于临界区外的进程不得阻塞其他进程
4. 不能使任何进程无限等待进入临界区



使用临界区的互斥

从抽象的角度来看，我们通常希望进程的行为如上图所示，在 t1 时刻，进程 A 进入临界区，在 t2 的时刻，进程 B 尝试进入临界区，因为此时进程 A 正在处于临界区中，所以进程 B 会阻塞直到 t3 时刻进程 A 离开临界区，此时进程 B 能够允许进入临界区。最后，在 t4 时刻，进程 B 离开临界区，系统恢复到没有进程的原始状态。

忙等互斥

下面我们会继续探讨实现互斥的各种设计，在这些方案中，当一个进程正忙于更新其关键区域的共享内存时，没有其他进程会进入其关键区域，也不会造成影响。

屏蔽中断

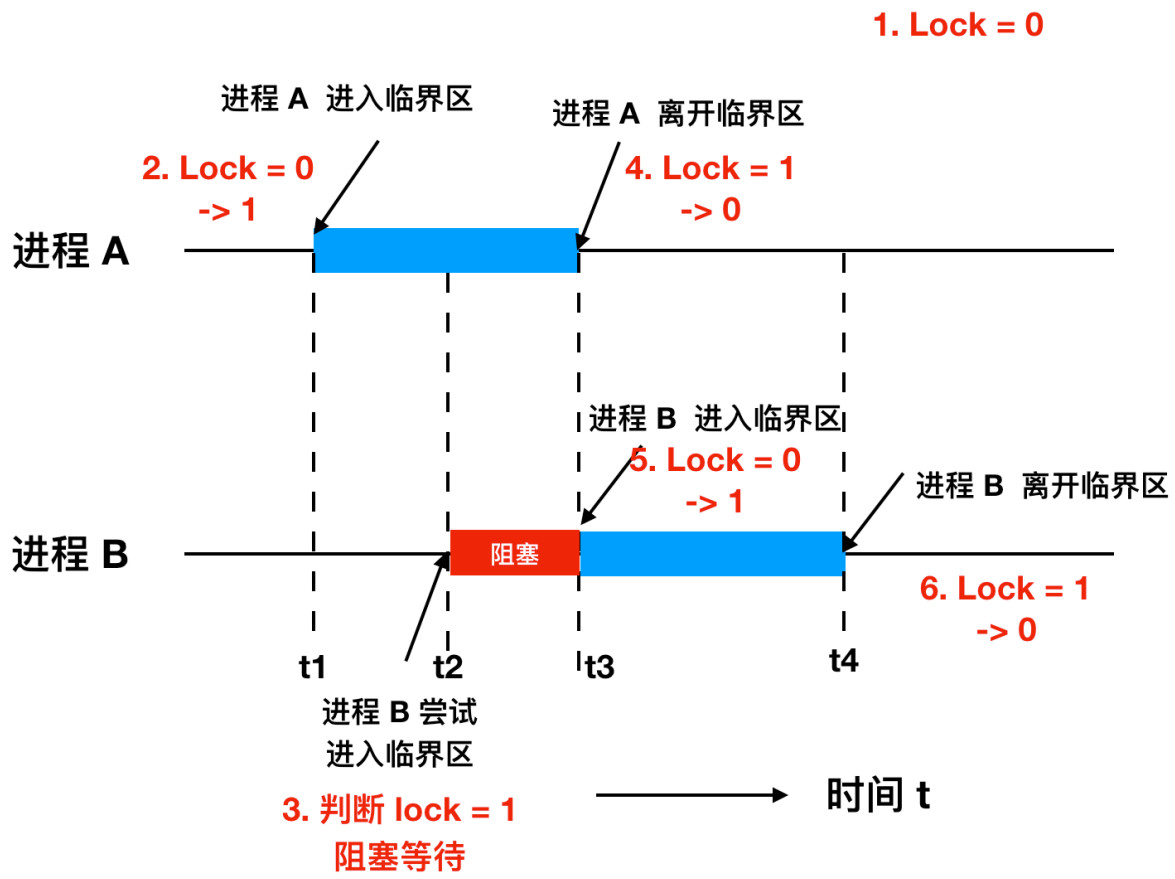
在单处理器系统上，最简单的解决方案是让每个进程在进入临界区后立即 `屏蔽所有中断` ，并在离开临界区之前重新启用它们。屏蔽中断后，时钟中断也会被屏蔽。CPU 只有发生时钟中断或其他中断时才会进行进程切换。这样，在屏蔽中断后 CPU 不会切换到其他进程。所以，一旦某个进程屏蔽中断之后，它就可以检查和修改共享内存，而不用担心其他进程介入访问共享数据。

这个方案可行吗？进程进入临界区域是由谁决定的呢？不是用户进程吗？当进程进入临界区域后，用户进程关闭中断，如果经过一段较长时间后进程没有离开，那么中断不就一直启用不了，结果会如何？可能会造成整个系统的终止。而且如果是多处理器的话，屏蔽中断仅仅对执行 `disable` 指令的 CPU 有效。其他 CPU 仍将继续运行，并可以访问共享内存。

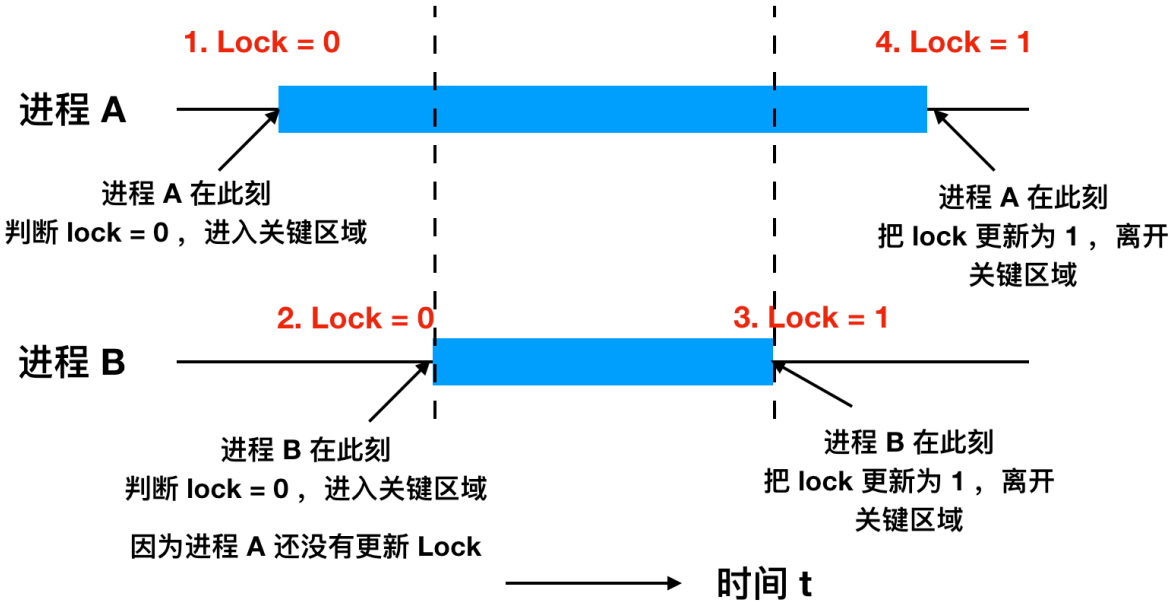
另一方面，对内核来说，当它在执行更新变量或列表的几条指令期间将中断屏蔽是很方便的。例如，如果多个进程处理就绪列表中的时候发生中断，则可能会发生竞态条件的出现。所以，屏蔽中断对于操作系统本身来说是一项很有用的技术，但是对于用户线程来说，屏蔽中断却不是一项通用的互斥机制。

锁变量

作为第二种尝试，可以寻找一种软件层面解决方案。考虑有单个共享的（锁）变量，初始为值为 0 。当一个线程想要进入关键区域时，它首先会查看锁的值是否为 0 ，如果锁的值是 0 ，进程会把它设置为 1 并让进程进入关键区域。如果锁的状态是 1，进程会等待直到锁变量的值变为 0 。因此，锁变量的值是 0 则意味着没有线程进入关键区域。如果是 1 则意味着有进程在关键区域内。我们对上图修改后，如下所示



这种设计方式是否正确呢？是否存在纰漏呢？假设一个进程读出锁变量的值并发现它为 0 ，而恰好在它将其设置为 1 之前，另一个进程调度运行，读出锁的变量为0 ，并将锁的变量设置为 1 。然后第一个线程运行，把锁变量的值再次设置为 1，此时，临界区域就会有两个进程在同时运行。



也许有的读者可以这么认为，在进入前检查一次，在要离开的关键区域再检查一次不就解决了吗？实际上这种情况也是于事无补，因为在第二次检查期间其他线程仍有可能修改锁变量的值，换句话说，这种 `set-before-check` 不是一种 `原子性` 操作，所以同样还会发生竞争条件。

严格轮询法

第三种互斥的方式先抛出来一段代码，这里的程序是用 C 语言编写，之所以采用 C 是因为操作系统普遍是用 C 来编写的（偶尔会用 C++），而基本不会使用 Java 、Modula3 或 Pascal 这样的语言，Java 中的 native 关键字底层也是 C 或 C++ 编写的源码。对于编写操作系统而言，需要使用 C 语言这种强大、高效、可预知和有特性的语言，而对于 Java ，它是不可预知的，因为它在关键时刻会用完存储器，而在不合适的时候会调用垃圾回收机制回收内存。在 C 语言中，这种情况不会发生，C 语言中不会主动调用垃圾回收回收内存。有关 C 、C++ 、Java 和其他四种语言的比较可以参考 **链接**

进程 0 的代码

```
while(TRUE){
  while(turn != 0){
    /* 进入关键区域 */
    critical_region();
    turn = 1;
    /* 离开关键区域 */
    noncritical_region();
  }
}
```

进程 1 的代码

```
while(TRUE){
  while(turn != 1){
    critical_region();
    turn = 0;
    noncritical_region();
  }
}
```

在上面代码中，变量 `turn` ，初始值为 0 ，用于记录轮到那个进程进入临界区，并检查或更新共享内存。开始时，进程 0 检查 turn，发现其值为 0 ，于是进入临界区。进程 1 也发现其值为 0 ，所以在一个等待循环中不停的测试 turn，看其值何时变为 1。连续检查一个变量直到某个值出现为止，这种方法称为 `忙等待(busywaiting)` 。由于这种方式浪费 CPU 时间，所以这种方式通常应该要避免。只有在有理由认为等待时间是非常短的情况下，才能够使用忙等待。用于忙等待的锁，称为 `自旋锁(spinlock)` 。

进程 0 离开临界区时，它将 turn 的值设置为 1，以便允许进程 1 进入其临界区。假设进程 1 很快便离开了临界区，则此时两个进程都处于临界区之外，turn 的值又被设置为 0 。现在进程 0 很快就执行完了整个循环，它退出临界区，并将 turn 的值设置为 1。此时，turn 的值为 1，两个进程都在其临界区外执行。

突然，进程 0 结束了非临界区的操作并返回到循环的开始。但是，这时它不能进入临界区，因为 turn 的当前值为 1，此时进程 1 还忙于非临界区的操作，进程 0 只能继续 while 循环，直到进程 1 把 turn 的值改为 0 。这说明，在一个进程比另一个进程执行速度慢了很多的情况下，轮流进入临界区并不是一个好的方法。

这种情况违反了前面的叙述 3 ，即 **位于临界区外的进程不得阻塞其他进程**，进程 0 被一个临界区外的进程阻塞。由于违反了第三条，所以也不能作为一个好的方案。

Peterson 解法

荷兰数学家 T.Dekker 通过将锁变量与警告变量相结合，最早提出了一个不需要严格轮换的软件互斥算法，关于 Dekker 的算法，参考 **链接**

后来， G.L.Peterson 发现了一种简单很多的互斥算法，它的算法如下

```
#define FALSE 0
#define TRUE  1
/* 进程数量 */
#define N     2

/* 现在轮到谁 */
int turn;

/* 所有值初始化为 0 (FALSE) */
int interested[N];

/* 进程是 0 或 1 */
void enter_region(int process){

  /* 另一个进程号 */
  int other;

  /* 另一个进程 */
  other = 1 - process;

  /* 表示愿意进入临界区 */
  interested[process] = TRUE;
  turn = process;

  /* 空循环 */
  while(turn == process
        && interested[other] == true){}

}

void leave_region(int process){

  /* 表示离开临界区 */
  interested[process] == FALSE;
}
```

在使用共享变量时（即进入其临界区）之前，各个进程使用各自的进程号 0 或 1 作为参数来调用 `enter_region`，这个函数调用在需要时将使进程等待，直到能够安全的临界区。在完成对共享变量的操作之后，进程将调用 `leave_region` 表示操作完成，并且允许其他进程进入。

现在来看看这个办法是如何工作的。一开始，没有任何进程处于临界区中，现在进程 0 调用 `enter_region`。它通过设置数组元素和将 turn 置为 0 来表示它希望进入临界区。由于进程 1 并不想进入临界区，所以 enter_region 很快便返回。如果进程现在调用 enter_region，进程 1 将在此处挂起直到 `interested[0]` 变为 FALSE，这种情况只有在进程 0 调用 `leave_region` 退出临界区时才会发生。

那么上面讨论的是顺序进入的情况，现在来考虑一种两个进程同时调用 `enter_region` 的情况。它们都将自己的进程存入 turn，但只有最后保存进去的进程号才有效，前一个进程的进程号因为重写而丢失。假如进程 1 是最后存入的，则 turn 为 1 。当两个进程都运行到 `while` 的时候，进程 0 将不会循环并进入临界区，而进程 1 将会无限循环且不会进入临界区，直到进程 0 退出位置。

TSL 指令

现在来看一种需要硬件帮助的方案。一些计算机，特别是那些设计为多处理器的计算机，都会有下面这条指令

```
1    TSL RX,LOCK
```

称为 `测试并加锁(test and set lock)`，它将一个内存字 lock 读到寄存器 `RX` 中，然后在该内存地址上存储一个非零值。读写指令能保证是一体的，不可分割的，一同执行的。在这个指令结束之前其他处理器均不允许访问内存。执行 TSL 指令的 CPU 将会锁住内存总线，用来禁止其他 CPU 在这个指令结束之前访问内存。

很重要的一点是锁住内存总线和禁用中断不一样。禁用中断并不能保证一个处理器在读写操作之间另一个处理器对内存的读写。也就是说，在处理器 1 上屏蔽中断对处理器 2 没有影响。让处理器 2 远离内存直到处理器 1 完成读写的最好的方式就是锁住总线。这需要一个特殊的硬件（基本上，一根总线就可以确保总线由锁住它的处理器使用，而其他的处理器不能使用）

为了使用 TSL 指令，要使用一个共享变量 lock 来协调对共享内存的访问。当 lock 为 0 时，任何进程都可以使用 TSL 指令将其设置为 1，并读写共享内存。当操作结束时，进程使用 `move` 指令将 lock 的值重新设置为 0 。

这条指令如何防止两个进程同时进入临界区呢？下面是解决方案

```
1   enter_region:
2         | 复制锁到寄存器并将锁设为1
3         TSL REGISTER,LOCK
4         | 锁是 0 吗?
5         CMP REGISTER,#0
6         | 若不是零，说明锁已被设置，所以循环
7         JNE enter_region
8         | 返回调用者，进入临界区
9         RET
10
11  leave_region:
12
13        | 在锁中存入 0
14        MOVE LOCK,#0
15        | 返回调用者
16        RET
```

我们可以看到这个解决方案的思想和 Peterson 的思想很相似。假设存在如下共 4 指令的汇编语言程序。第一条指令将 lock 原来的值复制到寄存器中并将 lock 设置为 1 ，随后这个原来的值和 0 做对比。如果它不是零，说明之前已经被加过锁，则程序返回到开始并再次测试。经过一段时间后（可长可短），该值变为 0 （当前处于临界区中的进程退出临界区时），于是过程返回，此时已加锁。要清除这个锁也比较简单，程序只需要将 0 存入 lock 即可，不需要特殊的同步指令。

现在有了一种很明确的做法，那就是进程在进入临界区之前会先调用 `enter_region`，判断是否进行循环，如果lock 的值是 1，进行无限循环，如果 lock 是 0，不进入循环并进入临界区。在进程从临界区返回时它调用 `leave_region`，这会把 lock 设置为 0 。与基于临界区问题的所有解法一样，进程必须在正确的时间调用 enter_region 和 leave_region，解法才能奏效。

还有一个可以替换 TSL 的指令是 `XCHG`，它原子性的交换了两个位置的内容，例如，一个寄存器与一个内存字，代码如下

```
enter_region:
    | 把 1 放在内存器中
    MOVE REGISTER,#1
    | 交换寄存器和锁变量的内容
    XCHG REGISTER,LOCK
    | 锁是 0 吗?
    CMP REGISTER,#0
    | 若不是 0 ，锁已被设置，进行循环
    JNE enter_region
    | 返回调用者，进入临界区
    RET

leave_region:
    | 在锁中存入 0
    MOVE LOCK,#0
    | 返回调用者
    RET
```

XCHG 的本质上与 TSL 的解决办法一样。所有的 Intel x86 CPU 在底层同步中使用 XCHG 指令。

睡眠与唤醒

上面解法中的 Peterson 、TSL 和 XCHG 解法都是正确的，但是它们都有忙等待的缺点。这些解法的本质上都是一样的，先检查是否能够进入临界区，若不允许，则该进程将原地等待，直到允许为止。

这种方式不但浪费了 CPU 时间，而且还可能引起意想不到的结果。考虑一台计算机上有两个进程，这两个进程具有不同的优先级，`H` 是属于优先级比较高的进程，`L` 是属于优先级比较低的进程。进程调度的规则是不论何时只要 H 进程处于就绪态 H 就开始运行。在某一时刻，L 处于临界区中，此时 H 变为就绪态，准备运行（例如，一条 I/O 操作结束）。现在 H 要开始忙等，但由于当 H 就绪时 L 就不会被调度，L 从来不会有机会离开关键区域，所以 H 会变成死循环，有时将这种情况称为 `优先级反转问题(priority inversion problem)`。

现在让我们看一下进程间的通信原语，这些原语在不允许它们进入关键区域之前会阻塞而不是浪费 CPU 时间，最简单的是 `sleep` 和 `wakeup`。Sleep 是一个能够造成调用者阻塞的系统调用，也就是说，这个系统调用会暂停直到其他进程唤醒它。wakeup 调用有一个参数，即要唤醒的进程。还有一种方式是 wakeup 和 sleep 都有一个参数，即 sleep 和 wakeup 需要匹配的内存地址。

生产者-消费者问题

作为这些私有原语的例子，让我们考虑 `生产者-消费者(producer-consumer)` 问题，也称作 `有界缓冲区(bounded-buffer)` 问题。两个进程共享一个公共的固定大小的缓冲区。其中一个是 `生产者(producer)`，将信息放入缓冲区， 另一个是 `消费者(consumer)`，会从缓冲区中取出。也可以把这个问题一般化为 m 个生产者和 n 个消费者的问题，但是我们这里只讨论一个生产者和一个消费者的情

况，这样可以简化实现方案。

如果缓冲队列已满，那么当生产者仍想要将数据写入缓冲区的时候，会出现问题。它的解决办法是让生产者睡眠，也就是阻塞生产者。等到消费者从缓冲区中取出一个或多个数据项时再唤醒它。同样的，当消费者试图从缓冲区中取数据，但是发现缓冲区为空时，消费者也会睡眠，阻塞。直到生产者向其中放入一个新的数据。

这个逻辑听起来比较简单，而且这种方式也需要一种称作 `监听` 的变量，这个变量用于监视缓冲区的数据，我们暂定为 count，如果缓冲区最多存放 N 个数据项，生产者会每次判断 count 是否达到 N，否则生产者向缓冲区放入一个数据项并增量 count 的值。消费者的逻辑也很相似：首先测试 count 的值是否为 0 ，如果为 0 则消费者睡眠、阻塞，否则会从缓冲区取出数据并使 count 数量递减。每个进程也会检查检查是否其他线程是否应该被唤醒，如果应该被唤醒，那么就唤醒该线程。下面是生产者消费者的代码

```
/* 缓冲区 slot 槽的数量 */
#define N 100
/* 缓冲区数据的数量 */
int count = 0

// 生产者
void producer(void){
  int item;

  /* 无限循环 */
  while(TRUE){
    /* 生成下一项数据 */
    item = produce_item()
    /* 如果缓存区是满的，就会阻塞 */
    if(count == N){
      sleep();
    }

    /* 把当前数据放在缓冲区中 */
    insert_item(item);
    /* 增加缓冲区 count 的数量 */
    count = count + 1;
    if(count == 1){
      /* 缓冲区是否为空？ */
      wakeup(consumer);
    }
  }
}

// 消费者
void consumer(void){

  int item;

  /* 无限循环 */
  while(TRUE){
    /* 如果缓冲区是空的，就会进行阻塞 */
    if(count == 0){
```

```
39          sleep();
40        }
41        /* 从缓冲区中取出一个数据 */
42        item = remove_item();
43        /* 将缓冲区的 count 数量减一 */
44        count = count - 1
45        /* 缓冲区满嘛?  */
46        if(count == N - 1){
47          wakeup(producer);
48        }
49        /* 打印数据项 */
50        consumer_item(item);
51      }
52
53    }
```

为了在 C 语言中描述像是 `sleep` 和 `wakeup` 的系统调用，我们将以库函数调用的形式来表示。它们不是 C 标准库的一部分，但可以在实际具有这些系统调用的任何系统上使用。代码中未实现的 `insert_item` 和 `remove_item` 用来记录将数据项放入缓冲区和从缓冲区取出数据等。

现在让我们回到生产者-消费者问题上来，上面代码中会产生竞争条件，因为 count 这个变量是暴露在大众视野下的。有可能出现下面这种情况：缓冲区为空，此时消费者刚好读取 count 的值发现它为 0 。此时调度程序决定暂停消费者并启动运行生产者。生产者生产了一条数据并把它放在缓冲区中，然后增加 count 的值，并注意到它的值是 1 。由于 count 为 0，消费者必须处于睡眠状态，因此生产者调用 `wakeup` 来唤醒消费者。但是，消费者此时在逻辑上并没有睡眠，所以 wakeup 信号会丢失。当消费者下次启动后，它会查看之前读取的 count 值，发现它的值是 0 ，然后在此进行睡眠。不久之后生产者会填满整个缓冲区，在这之后会阻塞，这样一来两个进程将永远睡眠下去。

引起上面问题的本质是 **唤醒尚未进行睡眠状态的进程会导致唤醒丢失**。如果它没有丢失，则一切都很正常。一种快速解决上面问题的方式是增加一个 `唤醒等待位(wakeup waiting bit)` 。当一个 wakeup 信号发送给仍在清醒的进程后，该位置为 1 。之后，当进程尝试睡眠的时候，如果唤醒等待位为 1 ，则该位清除，而进程仍然保持清醒。

然而，当进程数量有许多的时候，这时你可以说通过增加唤醒等待位的数量来唤醒等待位，于是就有了 2、4、6、8 个唤醒等待位，但是并没有从根本上解决问题。

信号量

信号量是 E.W.Dijkstra 在 1965 年提出的一种方法，它使用一个整形变量来累计唤醒次数，以供之后使用。在他的观点中，有一个新的变量类型称作 `信号量(semaphore)` 。一个信号量的取值可以是 0 ，或任意正数。0 表示的是不需要任何唤醒，任意的正数表示的就是唤醒次数。

Dijkstra 提出了信号量有两个操作，现在通常使用 `down` 和 `up` （分别可以用 sleep 和 wakeup 来表示）。down 这个指令的操作会检查值是否大于 0 。如果大于 0 ，则将其值减 1 ；若该值为 0 ，则进程将睡眠，而且此时 down 操作将会继续执行。检查数值、修改变量值以及可能发生的睡眠操作均为一个单一的、不可分割的 `原子操作(atomic action)` 完成。这会保证一旦信号量操作开始，没有其他的进程能够访问信号量，直到操作完成或者阻塞。这种原子性对于解决同步问题和避免竞争绝对必不可少。

> 原子性操作指的是在计算机科学的许多其他领域中，一组相关操作全部执行而没有中断或根本不执行。

up 操作会使信号量的值 + 1。如果一个或者多个进程在信号量上睡眠，无法完成一个先前的 down 操作，则由系统选择其中一个并允许该程完成 down 操作。因此，对一个进程在其上睡眠的信号量执行一次 up 操作之后，该信号量的值仍然是 0 ，但在其上睡眠的进程却少了一个。信号量的值增 1 和唤醒一个进程同样也是不可分割的。不会有某个进程因执行 up 而阻塞，正如在前面的模型中不会有进程因执行 wakeup 而阻塞是一样的道理。

用信号量解决生产者 - 消费者问题

用信号量解决丢失的 wakeup 问题，代码如下

```
/* 定义缓冲区槽的数量 */
#define N 100
/* 信号量是一种特殊的 int */
typedef int semaphore;
/* 控制关键区域的访问 */
semaphore mutex = 1;
/* 统计 buffer 空槽的数量 */
semaphore empty = N;
/* 统计 buffer 满槽的数量 */
semaphore full = 0;

void producer(void){

  int item;

  /* TRUE 的常量是 1 */
  while(TRUE){
    /* 产生放在缓冲区的一些数据 */
    item = producer_item();
    /* 将空槽数量减 1  */
    down(&empty);
    /* 进入关键区域  */
    down(&mutex);
    /* 把数据放入缓冲区中 */
    insert_item(item);
    /* 离开临界区 */
    up(&mutex);
    /* 将 buffer 满槽数量 + 1 */
    up(&full);
  }
}

void consumer(void){

  int item;

  /* 无限循环 */
  while(TRUE){
    /* 缓存区满槽数量 - 1 */
    down(&full);
    /* 进入缓冲区 */
```

```
        down(&mutex);
        /* 从缓冲区取出数据 */
        item = remove_item();
        /* 离开临界区 */
        up(&mutex);
        /* 将空槽数目 + 1 */
        up(&empty);
        /* 处理数据 */
        consume_item(item);
    }

}
```

为了确保信号量能正确工作，最重要的是要采用一种不可分割的方式来实现它。通常是将 up 和 down 作为系统调用来实现。而且操作系统只需在执行以下操作时暂时屏蔽全部中断：**检查信号量、更新、必要时使进程睡眠**。由于这些操作仅需要非常少的指令，因此中断不会造成影响。如果使用多个 CPU，那么信号量应该被锁进行保护。使用 TSL 或者 XCHG 指令用来确保同一时刻只有一个 CPU 对信号量进行操作。

使用 TSL 或者 XCHG 来防止几个 CPU 同时访问一个信号量，与生产者或消费者使用忙等待来等待其他腾出或填充缓冲区是完全不一样的。前者的操作仅需要几个毫秒，而生产者或消费者可能需要任意长的时间。

上面这个解决方案使用了三种信号量：一个称为 full，用来记录充满的缓冲槽数目；一个称为 empty，记录空的缓冲槽数目；一个称为 mutex，用来确保生产者和消费者不会同时进入缓冲区。`Full` 被初始化为 0 ，empty 初始化为缓冲区中插槽数，mutex 初始化为 1。信号量初始化为 1 并且由两个或多个进程使用，以确保它们中同时只有一个可以进入关键区域的信号被称为 `二进制信号量(binary semaphores)` 。如果每个进程都在进入关键区域之前执行 down 操作，而在离开关键区域之后执行 up 操作，则可以确保相互互斥。

现在我们有了一个好的进程间原语的保证。然后我们再来看一下中断的顺序保证

1. 硬件压入堆栈程序计数器等
2. 硬件从中断向量装入新的程序计数器
3. 汇编语言过程保存寄存器的值
4. 汇编语言过程设置新的堆栈
5. C 中断服务器运行（典型的读和缓存写入）
6. 调度器决定下面哪个程序先运行
7. C 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

在使用 `信号量` 的系统中，隐藏中断的自然方法是让每个 I/O 设备都配备一个信号量，该信号量最初设置为0。在 I/O 设备启动后，中断处理程序立刻对相关联的信号执行一个 `down` 操作，于是进程立即被阻塞。当中断进入时，中断处理程序随后对相关的信号量执行一个 `up` 操作，能够使已经阻止的进程恢复运行。在上面的中断处理步骤中，其中的第 5 步 `C 中断服务器运行` 就是中断处理程序在信号量上执行的一个 up 操作，所以在第 6 步中，操作系统能够执行设备驱动程序。当然，如果有几个进程已经处于就绪状态，调度程序可能会选择接下来运行一个更重要的进程，我们会在后面讨论调度的算法。

上面的代码实际上是通过两种不同的方式来使用信号量的，而这两种信号量之间的区别也是很重要的。 `mutex` 信号量用于互斥。它用于确保任意时刻只有一个进程能够对缓冲区和相关变量进行读写。互斥是用于避免进程混乱所必须的一种操作。

另外一个信号量是关于 `同步(synchronization)` 的。 `full` 和 `empty` 信号量用于确保事件的发生或者不发生。在这个事例中，它们确保了缓冲区满时生产者停止运行；缓冲区为空时消费者停止运行。这两个信号量的使用与 mutex 不同。

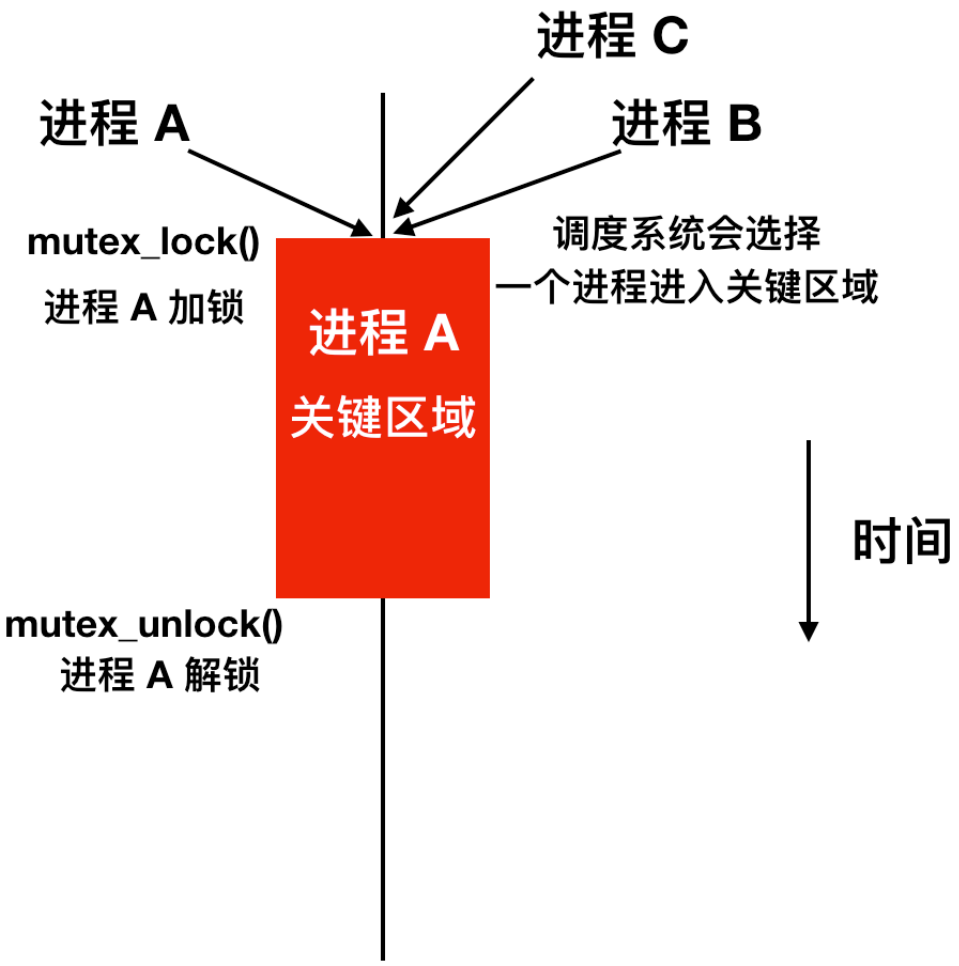
互斥量

如果不需要信号量的计数能力时，可以使用信号量的一个简单版本，称为 `mutex(互斥量)` 。互斥量的优势就在于在一些共享资源和一段代码中保持互斥。由于互斥的实现既简单又有效，这使得互斥量在实现用户空间线程包时非常有用。

互斥量是一个处于两种状态之一的共享变量： `解锁(unlocked)` 和 `加锁(locked)` 。这样，只需要一个二进制位来表示它，不过一般情况下，通常会用一个 `整形(integer)` 来表示。0 表示解锁，其他所有的值表示加锁，比 1 大的值表示加锁的次数。

mutex 使用两个过程，当一个线程（或者进程）需要访问关键区域时，会调用 `mutex_lock` 进行加锁。如果互斥锁当前处于解锁状态（表示关键区域可用），则调用成功，并且调用线程可以自由进入关键区域。

另一方面，如果 mutex 互斥量已经锁定的话，调用线程会阻塞直到关键区域内的线程执行完毕并且调用了 `mutex_unlock` 。如果多个线程在 mutex 互斥量上阻塞，将随机选择一个线程并允许它获得锁。



由于 mutex 互斥量非常简单，所以只要有 TSL 或者是 XCHG 指令，就可以很容易地在用户空间实现它们。用于用户级线程包的 `mutex_lock` 和 `mutex_unlock` 代码如下，XCHG 的本质也一样。

```
mutex_lock:
        | 将互斥信号量复制到寄存器，并将互斥信号量置为1
        TSL REGISTER,MUTEX
        | 互斥信号量是 0 吗?
        CMP REGISTER,#0
        | 如果互斥信号量为0，它被解锁，所以返回
        JZE ok
        | 互斥信号正在使用；调度其他线程
        CALL thread_yield
        | 再试一次
        JMP mutex_lock
        | 返回调用者，进入临界区
ok:     RET

mutex_unlcok:
        | 将 mutex 置为 0
        MOVE MUTEX,#0
        | 返回调用者
        RET
```

mutex_lock 的代码和上面 enter_region 的代码很相似，我们可以对比着看一下

TSL 进入临界区和离开临界区

```
enter_region:
        TSL REGISTER,LOCK           | 复制锁到寄存器并将锁设为1
        CMP REGISTER,#0             | 锁是 0 吗?
        JNE enter_region            | 若不是零，说明锁已被设置，所以循环
        RET                         | 返回调用者，进入临界区


leave_region:
        MOVE LOCK,#0                | 在锁中存入 0
        RET                         | 返回调用者
```

mutex 进入临界区和离开临界区

```
mutex_lock:
        TSL REGISTER,MUTEX          | 将互斥信号量复制到寄存器，并将互斥信号量置为1
        CMP REGISTER,#0             | 互斥信号量是 0 吗?
        JZE ok                      | 如果互斥信号量为0，它被解锁，所以返回
        CALL thread_yield           | 互斥信号正在使用；调度其他线程
        JMP mutex_lock              | 再试一次
ok:     RET                         | 返回调用者，进入临界区

mutex_unlcok:
        MOVE MUTEX,#0               | 将 mutex 置为 0
        RET                         | 返回调用者
```

上面代码最大的区别你看出来了吗?

- 根据上面我们对 TSL 的分析，我们知道，如果 TSL 判断没有进入临界区的进程会进行无限循环获取锁，而在 TSL 的处理中，如果 mutex 正在使用，那么就调度其他线程进行处理。所以上面最大的区别其实就是在判断 mutex/TSL 之后的处理。
- 在（用户）线程中，情况有所不同，因为没有时钟来停止运行时间过长的线程。结果是通过忙等待的方式来试图获得锁的线程将永远循环下去，决不会得到锁，因为这个运行的线程不会让其他线程

> 运行从而释放锁，其他线程根本没有获得锁的机会。在后者获取锁失败时，它会调用 `thread_yield` 将 CPU 放弃给另外一个线程。结果就不会进行忙等待。在该线程下次运行时，它再一次对锁进行测试。

上面就是 enter_region 和 mutex_lock 的差别所在。由于 thread_yield 仅仅是一个用户空间的线程调度，所以它的运行非常快捷。这样， `mutex_lock` 和 `mutex_unlock` 都不需要任何内核调用。通过使用这些过程，用户线程完全可以实现在用户空间中的同步，这个过程仅仅需要少量的同步。

我们上面描述的互斥量其实是一套调用框架中的指令。从软件角度来说，总是需要更多的特性和同步原语。例如，有时线程包提供一个调用 `mutex_trylock` ，这个调用尝试获取锁或者返回错误码，但是不会进行加锁操作。这就给了调用线程一个灵活性，以决定下一步做什么，是使用替代方法还是等候下去。

Futexes

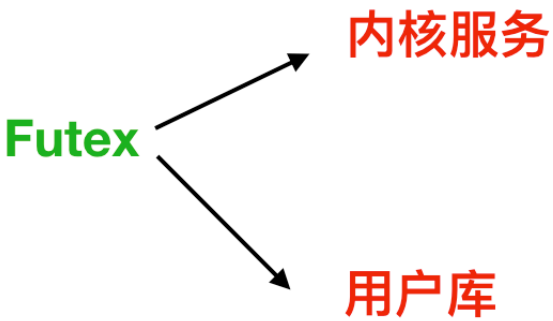
随着并行的增加，有效的 `同步(synchronization)` 和 `锁定(locking)` 对于性能来说是非常重要的。如果进程等待时间很短，那么 `自旋锁(Spin lock)` 是非常有效；但是如果等待时间比较长，那么这会浪费 CPU 周期。如果进程很多，那么阻塞此进程，并仅当锁被释放的时候让内核解除阻塞是更有效的方式。不幸的是，这种方式也会导致另外的问题：它可以在进程竞争频繁的时候运行良好，但是在竞争不是很激烈的情况下内核切换的消耗会非常大，而且更困难的是，预测锁的竞争数量更不容易。

有一种有趣的解决方案是把两者的优点结合起来，提出一种新的思想，称为 `futex` ，或者是 `快速用户空间互斥(fast user space mutex)` ，是不是听起来很有意思？

同步(synchronization) + 互斥(locking) =====> ？？？？

Futex !!!!!

futex 是 `Linux` 中的特性实现了基本的锁定（很像是互斥锁）而且避免了陷入内核中，因为内核的切换的开销非常大，这样做可以大大提高性能。futex 由两部分组成：**内核服务和用户库**。内核服务提供了了一个 `等待队列(wait queue)` 允许多个进程在锁上排队等待。除非内核明确的对他们解除阻塞，否则它们不会运行。



对于一个进程来说，把它放到等待队列需要昂贵的系统调用，这种方式应该被避免。在没有竞争的情况下，futex 可以直接在用户空间中工作。这些进程共享一个 32 位 `整数(integer)` 作为公共锁变量。假设锁的初始化为 1，我们认为这时锁已经被释放了。线程通过执行原子性的操作 `减少并测试(decrement and test)` 来抢占锁。decrement and set 是 Linux 中的原子功能，由包裹在 C 函数中

的内联汇编组成，并在头文件中进行定义。下一步，线程会检查结果来查看锁是否已经被释放。如果锁现在不是锁定状态，那么刚好我们的线程可以成功抢占该锁。然而，如果锁被其他线程持有，抢占锁的线程不得不等待。在这种情况下，futex 库不会 `自旋` ，但是会使用一个系统调用来把线程放在内核中的等待队列中。这样一来，切换到内核的开销已经是合情合理的了，因为线程可以在任何时候阻塞。当线程完成了锁的工作时，它会使用原子性的 `增加并测试(increment and test)` 释放锁，并检查结果以查看内核等待队列上是否仍阻止任何进程。如果有的话，它会通知内核可以对等待队列中的一个或多个进程解除阻塞。如果没有锁竞争，内核则不需要参与竞争。

Pthreads 中的互斥量

Pthreads 提供了一些功能用来同步线程。最基本的机制是使用互斥量变量，可以锁定和解锁，用来保护每个关键区域。希望进入关键区域的线程首先要尝试获取 mutex。如果 mutex 没有加锁，线程能够马上进入并且互斥量能够自动锁定，从而阻止其他线程进入。如果 mutex 已经加锁，调用线程会阻塞，直到 mutex 解锁。如果多个线程在相同的互斥量上等待，当互斥量解锁时，只有一个线程能够进入并且重新加锁。这些锁并不是必须的，程序员需要正确使用它们。

下面是与互斥量有关的函数调用

线程调用	描述
Pthread_mutex_init	创建一个互斥量
Pthread_mutex_destroy	撤销一个已存在的互斥量
Pthread_mutex_lock	获得一个锁或阻塞
Pthread_mutex_trylock	获得一个锁或失败
Pthread_mutex_unlock	释放一个锁

向我们想象中的一样，mutex 能够被创建和销毁，扮演这两个角色的分别是 `Phread_mutex_init` 和 `Pthread_mutex_destroy` 。mutex 也可以通过 `Pthread_mutex_lock` 来进行加锁，如果互斥量已经加锁，则会阻塞调用者。还有一个调用 `Pthread_mutex_trylock` 用来尝试对线程加锁，当 mutex 已经被加锁时，会返回一个错误代码而不是阻塞调用者。这个调用允许线程有效的进行忙等。最后， `Pthread_mutex_unlock` 会对 mutex 解锁并且释放一个正在等待的线程。

除了互斥量以外， `Pthreads` 还提供了第二种同步机制： `条件变量(condition variables)` 。mutex 可以很好的允许或阻止对关键区域的访问。条件变量允许线程由于未满足某些条件而阻塞。绝大多数情况下这两种方法是一起使用的。下面我们进一步来研究线程、互斥量、条件变量之间的关联。

下面再来重新认识一下生产者和消费者问题：一个线程将东西放在一个缓冲区内，由另一个线程将它们取出。如果生产者发现缓冲区没有空槽可以使用了，生产者线程会阻塞起来直到有一个线程可以使用。生产者使用 mutex 来进行原子性检查从而不受其他线程干扰。但是当发现缓冲区已经满了以后，生产者需要一种方法来阻塞自己并在以后被唤醒。这便是条件变量做的工作。

下面是一些与条件变量有关的最重要的 pthread 调用

线程调用	描述
Pthread_cond_init	创建一个条件变量
Pthread_cond_destroy	销毁一个条件变量
Pthread_cond_wait	阻塞以等待一个信号
Pthread_cond_signal	向另一个线程发信号来唤醒它
Pthread_cond_broadcast	向多个线程发信号来让它们全部唤醒

上表中给出了一些调用用来创建和销毁条件变量。条件变量上的主要属性是 `Pthread_cond_wait` 和 `Pthread_cond_signal` 。前者阻塞调用线程，直到其他线程发出信号为止（使用后者调用）。阻塞的线程通常需要等待唤醒的信号以此来释放资源或者执行某些其他活动。只有这样阻塞的线程才能继续工作。条件变量允许等待与阻塞原子性的进程。 `Pthread_cond_broadcast` 用来唤醒多个阻塞的、需要等待信号唤醒的线程。

> 需要注意的是，条件变量（不像是信号量）不会存在于内存中。如果将一个信号量传递给一个没有线程等待的条件变量，那么这个信号就会丢失，这个需要注意

下面是一个使用互斥量和条件变量的例子

```
#include <stdio.h>
#include <pthread.h>

/* 需要生产的数量 */
#define MAX 1000000000
pthread_mutex_t the_mutex;
/* 使用信号量 */
pthread_cond_t condc,condp;
int buffer = 0;

/* 生产数据 */
void *producer(void *ptr){

  int i;

  for(int i = 0;i <= MAX;i++){
```

```
      /* 缓冲区独占访问，也就是使用 mutex 获取锁 */
      pthread_mutex_lock(&the_mutex);
      while(buffer != 0){
        pthread_cond_wait(&condp,&the_mutex);
      }
      /* 把他们放在缓冲区中 */
      buffer = i;
      /* 唤醒消费者 */
      pthread_cond_signal(&condc);
      /* 释放缓冲区 */
      pthread_mutex_unlock(&the_mutex);
    }
    pthread_exit(0);

  }

  /* 消费数据 */
  void *consumer(void *ptr){

    int i;

    for(int i = 0;i <= MAX;i++){
      /* 缓冲区独占访问，也就是使用 mutex 获取锁 */
      pthread_mutex_lock(&the_mutex);
      while(buffer == 0){
        pthread_cond_wait(&condc,&the_mutex);
      }
      /* 把他们从缓冲区中取出 */
      buffer = 0;
      /* 唤醒生产者 */
      pthread_cond_signal(&condp);
      /* 释放缓冲区 */
      pthread_mutex_unlock(&the_mutex);
    }
    pthread_exit(0);

  }
```

管程

为了能够编写更加准确无误的程序，Brinch Hansen 和 Hoare 提出了一个更高级的同步原语叫做 `管程(monitor)` 。他们两个人的提案略有不同，通过下面的描述你就可以知道。管程是程序、变量和数据结构等组成的一个集合，它们组成一个特殊的模块或者包。进程可以在任何需要的时候调用管程中的程序，但是它们不能从管程外部访问数据结构和程序。下面展示了一种抽象的，类似 Pascal 语言展示的简洁的管程。不能用 C 语言进行描述，因为管程是语言概念而 C 语言并不支持管程。

```
monitor example
  integer i;
  condition c;

  procedure producer();
  ...
  end;

  procedure consumer();
  .
  end;
end monitor;
```

管程有一个很重要的特性，即在任何时候管程中只能有一个活跃的进程，这一特性使管程能够很方便的实现互斥操作。管程是编程语言的特性，所以编译器知道它们的特殊性，因此可以采用与其他过程调用不同的方法来处理对管程的调用。通常情况下，当进程调用管程中的程序时，该程序的前几条指令会检查管程中是否有其他活跃的进程。如果有的话，调用进程将被挂起，直到另一个进程离开管程才将其唤醒。如果没有活跃进程在使用管程，那么该调用进程才可以进入。

进入管程中的互斥由编译器负责，但是一种通用做法是使用 `互斥量(mutex)` 和 `二进制信号量(binary semaphore)` 。由于编译器而不是程序员在操作，因此出错的几率会大大降低。在任何时候，编写管程的程序员都无需关心编译器是如何处理的。他只需要知道将所有的临界区转换成为管程过程即可。绝不会有两个进程同时执行临界区中的代码。

即使管程提供了一种简单的方式来实现互斥，但在我们看来，这还不够。因为我们还需要一种在进程无法执行被阻塞。在生产者-消费者问题中，很容易将针对缓冲区满和缓冲区空的测试放在管程程序中，但是生产者在发现缓冲区满的时候该如何阻塞呢?

解决的办法是引入 `条件变量(condition variables)` 以及相关的两个操作 `wait` 和 `signal` 。当一个管程程序发现它不能运行时（例如，生产者发现缓冲区已满），它会在某个条件变量（如 full）上执行 `wait` 操作。这个操作造成调用进程阻塞，并且还将另一个以前等在管程之外的进程调入管程。在前面的 pthread 中我们已经探讨过条件变量的实现细节了。另一个进程，比如消费者可以通过执行 `signal` 来唤醒阻塞的调用进程。

> Brinch Hansen 和 Hoare 在对进程唤醒上有所不同，Hoare 建议让新唤醒的进程继续运行；而挂起另外的进程。而 Brinch Hansen 建议让执行 signal 的进程必须退出管程，这里我们采用 Brinch Hansen 的建议，因为它在概念上更简单，并且更容易实现。

如果在一个条件变量上有若干进程都在等待，则在对该条件执行 signal 操作后，系统调度程序只能选择其中一个进程恢复运行。

顺便提一下，这里还有上面两位教授没有提出的第三种方式，它的理论是让执行 signal 的进程继续运行，等待这个进程退出管程时，其他进程才能进入管程。

条件变量不是计数器。条件变量也不能像信号量那样积累信号以便以后使用。所以，如果向一个条件变量发送信号，但是该条件变量上没有等待进程，那么信号将会丢失。也就是说，**wait 操作必须在 signal 之前执行**。

下面是一个使用 `Pascal` 语言通过管程实现的生产者-消费者问题的解法

```
monitor ProducerConsumer
```

```
2       condition full,empty;
3       integer count;
4
5       procedure insert(item:integer);
6       begin
7           if count = N then wait(full);
8           insert_item(item);
9           count := count + 1;
10          if count = 1 then signal(empty);
11      end;
12
13      function remove:integer;
14      begin
15          if count = 0 then wait(empty);
16          remove = remove_item;
17          count := count - 1;
18          if count = N - 1 then signal(full);
19      end;
20
21      count := 0;
22  end monitor;
23
24  procedure producer;
25  begin
26        while true do
27        begin
28              item = produce_item;
29              ProducerConsumer.insert(item);
30        end
31  end;
32
33  procedure consumer;
34  begin
35        while true do
36        begin
37              item = ProducerConsumer.remove;
38              consume_item(item);
39        end
40  end;
```

读者可能觉得 wait 和 signal 操作看起来像是前面提到的 sleep 和 wakeup ，而且后者存在严重的竞争条件。它们确实很像，但是有个关键的区别：sleep 和 wakeup 之所以会失败是因为当一个进程想睡眠时，另一个进程试图去唤醒它。使用管程则不会发生这种情况。管程程序的自动互斥保证了这一点，如果管程过程中的生产者发现缓冲区已满，它将能够完成 wait 操作而不用担心调度程序可能会在 wait 完成之前切换到消费者。甚至，在 wait 执行完成并且把生产者标志为不可运行之前，是不会允许消费者进入管程的。

尽管类 Pascal 是一种想象的语言，但还是有一些真正的编程语言支持，比如 Java （终于轮到大 Java 出场了），Java 是能够支持管程的，它是一种 面向对象 的语言，支持用户级线程，还允许将方法划分为类。只要将关键字 `synchronized` 关键字加到方法中即可。Java 能够保证一旦某个线程执行该方法，就不允许其他线程执行该对象中的任何 synchronized 方法。没有关键字 synchronized ，就不能保

证没有交叉执行。

下面是 Java 使用管程解决的生产者-消费者问题

```
public class ProducerConsumer {
  // 定义缓冲区大小的长度
  static final int N = 100;
  // 初始化一个新的生产者线程
  static Producer p = new Producer();
  // 初始化一个新的消费者线程
  static Consumer c = new Consumer();
  // 初始化一个管程
  static Our_monitor mon = new Our_monitor();

  // run 包含了线程代码
  static class Producer extends Thread{
    public void run(){
      int item;
      // 生产者循环
      while(true){
        item = produce_item();
        mon.insert(item);
      }
    }
    // 生产代码
    private int produce_item(){...}
  }

  // run 包含了线程代码
  static class consumer extends Thread {
    public void run( ) {
      int item;
      while(true){
        item = mon.remove();
        consume_item(item);
      }
    }
    // 消费代码
    private int produce_item(){...}
  }

  // 这是管程
  static class Our_monitor {
    private int buffer[] = new int[N];
    // 计数器和索引
    private int count = 0,lo = 0,hi = 0;

    private synchronized void insert(int val){
      if(count == N){
        // 如果缓冲区是满的，则进入休眠
        go_to_sleep();
      }
```

```
49          // 向缓冲区插入内容
50          buffer[hi] = val;
51          // 找到下一个槽的为止
52          hi = (hi + 1) % N;
53          // 缓冲区中的数目自增 1
54          count = count + 1;
55          if(count == 1){
56            // 如果消费者睡眠，则唤醒
57            notify();
58          }
59        }
60
61        private synchronized void remove(int val){
62          int val;
63          if(count == 0){
64            // 缓冲区是空的，进入休眠
65            go_to_sleep();
66          }
67          // 从缓冲区取出数据
68          val = buffer[lo];
69          // 设置待取出数据项的槽
70          lo = (lo + 1) % N;
71          // 缓冲区中的数据项数目减 1
72          count = count - 1;
73          if(count = N - 1){
74            // 如果生产者睡眠，唤醒它
75            notify();
76          }
77          return val;
78        }
79
80        private void go_to_sleep() {
81          try{
82            wait( );
83          }catch(Interr uptedExceptionexc) {};
84        }
85      }
86
87    }
```

上面的代码中主要设计四个类，`外部类(outer class)` ProducerConsumer 创建并启动两个线程，p 和 c。第二个类和第三个类 `Producer` 和 `Consumer` 分别包含生产者和消费者代码。最后，`Our_monitor` 是管程，它有两个同步线程，用于在共享缓冲区中插入和取出数据。

在前面的所有例子中，生产者和消费者线程在功能上与它们是相同的。生产者有一个无限循环，该无限循环产生数据并将数据放入公共缓冲区中；消费者也有一个等价的无限循环，该无限循环用于从缓冲区取出数据并完成一系列工作。

程序中比较耐人寻味的就是 `Our_monitor` 了，它包含缓冲区、管理变量以及两个同步方法。当生产者在 insert 内活动时，它保证消费者不能在 remove 方法中运行，从而保证更新变量以及缓冲区的安全性，并且不用担心竞争条件。变量 count 记录在缓冲区中数据的数量。变量 `lo` 是缓冲区槽的序号，指出将要取出的下一个数据项。类似地，`hi` 是缓冲区中下一个要放入的数据项序号。允许 lo = hi,

含义是在缓冲区中有 0 个或 N 个数据。

Java 中的同步方法与其他经典管程有本质差别：Java 没有内嵌的条件变量。然而，Java 提供了 wait 和 notify 分别与 sleep 和 wakeup 等价。

通过临界区自动的互斥，管程比信号量更容易保证并行编程的正确性。但是管程也有缺点，我们前面说到过管程是一个编程语言的概念，编译器必须要识别管程并用某种方式对其互斥作出保证。**C、Pascal 以及大多数其他编程语言都没有管程**，所以不能依靠编译器来遵守互斥规则。

与管程和信号量有关的另一个问题是，这些机制都是设计用来解决访问共享内存的一个或多个 CPU 上的互斥问题的。通过将信号量放在共享内存中并用 `TSL` 或 `XCHG` 指令来保护它们，可以避免竞争。但是如果是在分布式系统中，可能同时具有多个 CPU 的情况，并且每个 CPU 都有自己的私有内存呢，它们通过网络相连，那么这些原语将会失效。因为信号量太低级了，而管程在少数几种编程语言之外无法使用，所以还需要其他方法。

消息传递

上面提到的其他方法就是 `消息传递(messaage passing)` 。这种进程间通信的方法使用两个原语 `send` 和 `receive` ，它们像信号量而不像管程，是系统调用而不是语言级别。示例如下

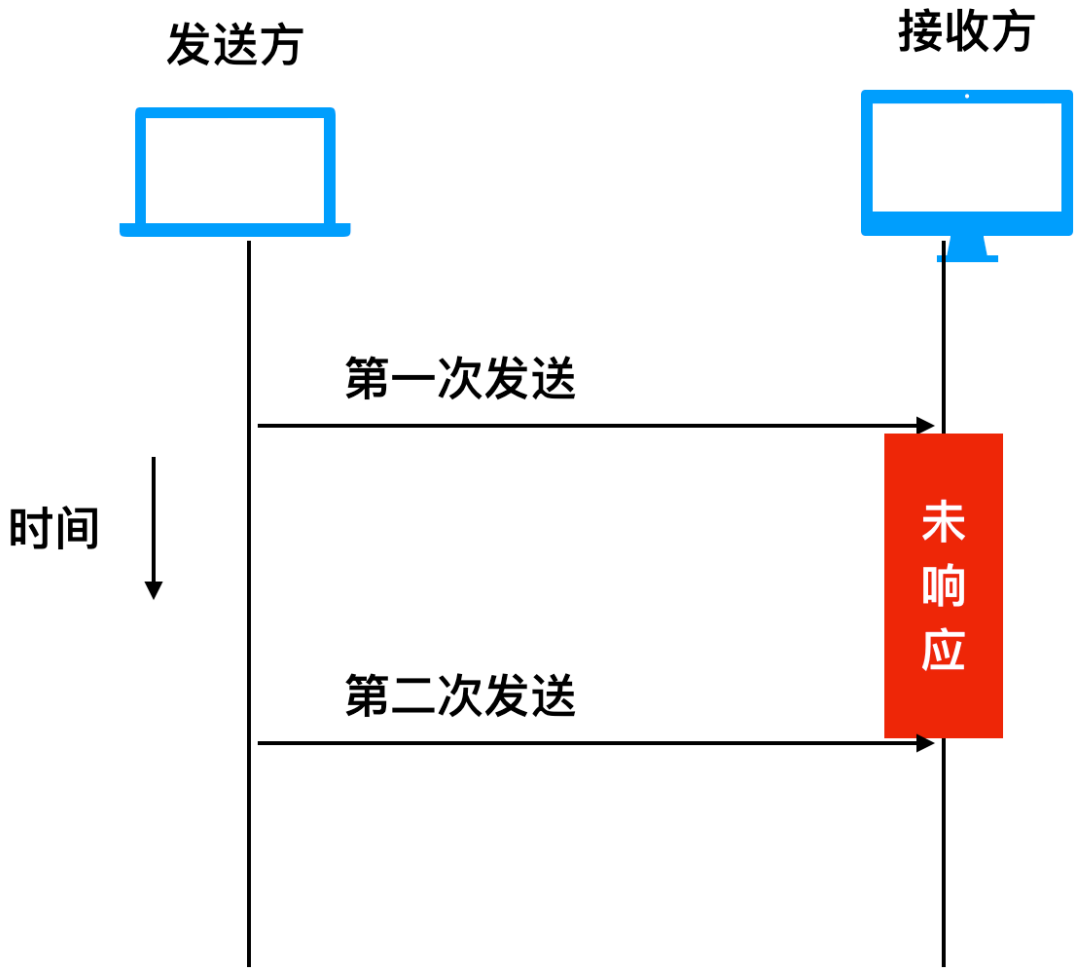
```
1   send(destination, &message);
2
3   receive(source, &message);
```

send 方法用于向一个给定的目标发送一条消息，receive 从一个给定的源接受一条消息。如果没有消息，接受者可能被阻塞，直到接受一条消息或者带着错误码返回。

消息传递系统的设计要点

消息传递系统现在面临着许多信号量和管程所未涉及的问题和设计难点，尤其对那些在网络中不同机器上的通信状况。例如，消息有可能被网络丢失。为了防止消息丢失，发送方和接收方可以达成一致：一旦接受到消息后，接收方马上回送一条特殊的 `确认(acknowledgement)` 消息。如果发送方在一段时间间隔内未收到确认，则重发消息。

现在考虑消息本身被正确接收，而返回给发送着的确认消息丢失的情况。发送者将重发消息，这样接受者将收到两次相同的消息。



对于接收者来说，如何区分新的消息和一条重发的老消息是非常重要的。通常采用在每条原始消息中嵌入一个连续的序号来解决此问题。如果接受者收到一条消息，它具有与前面某一条消息一样的序号，就知道这条消息是重复的，可以忽略。

消息系统还必须处理如何命名进程的问题，以便在发送或接收调用中清晰的指明进程。`身份验证(authentication)` 也是一个问题，比如客户端怎么知道它是在与一个真正的文件服务器通信，从发送方到接收方的信息有可能被中间人所篡改。

用消息传递解决生产者-消费者问题

现在我们考虑如何使用消息传递来解决生产者-消费者问题，而不是共享缓存。下面是一种解决方式

```
/* buffer 中槽的数量 */
#define N 100

void producer(void){

  int item;
  /* buffer 中槽的数量 */
  message m;

  while(TRUE){
    /* 生成放入缓冲区的数据 */
    item = produce_item();
    /* 等待消费者发送空缓冲区 */
```

```
      receive(consumer,&m);
      /* 建立一个待发送的消息 */
      build_message(&m,item);
      /* 发送给消费者 */
      send(consumer,&m);
    }

  }

  void consumer(void){

    int item,i;
    message m;

    /* 循环N次 */
    for(int i = 0;i < N;i++){
      /* 发送N个缓冲区 */
      send(producer,&m);
    }
    while(TRUE){
      /* 接受包含数据的消息 */
      receive(producer,&m);
      /* 将数据从消息中提取出来 */
      item = extract_item(&m);
      /* 将空缓冲区发送回生产者 */
      send(producer,&m);
      /* 处理数据 */
      consume_item(item);
    }

  }
```

假设所有的消息都有相同的大小，并且在尚未接受到发出的消息时，由操作系统自动进行缓冲。在该解决方案中共使用 N 条消息，这就类似于一块共享内存缓冲区的 N 个槽。消费者首先将 N 条空消息发送给生产者。当生产者向消费者传递一个数据项时，它取走一条空消息并返回一条填充了内容的消息。通过这种方式，系统中总的消息数量保持不变，所以消息都可以存放在事先确定数量的内存中。

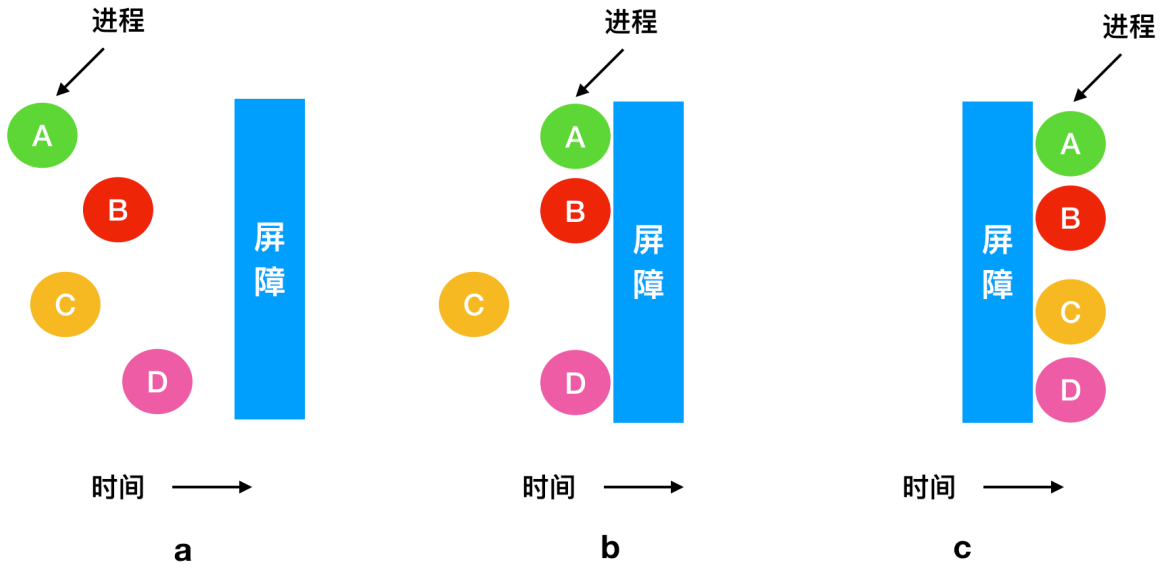
如果生产者的速度要比消费者快，则所有的消息最终都将被填满，等待消费者，生产者将被阻塞，等待返回一条空消息。如果消费者速度快，那么情况将正相反：所有的消息均为空，等待生产者来填充，消费者将被阻塞，以等待一条填充过的消息。

消息传递的方式有许多变体，下面先介绍如何对消息进行 `编址` 。

- 一种方法是为每个进程分配一个唯一的地址，让消息按进程的地址编址。
- 另一种方式是引入一个新的数据结构，称为 `信箱(mailbox)` ，信箱是一个用来对一定的数据进行缓冲的数据结构，信箱中消息的设置方法也有多种，典型的方法是在信箱创建时确定消息的数量。在使用信箱时，在 send 和 receive 调用的地址参数就是信箱的地址，而不是进程的地址。当一个进程试图向一个满的信箱发送消息时，它将被挂起，直到信箱中有消息被取走，从而为新的消息腾出地址空间。

屏障

最后一个同步机制是准备用于进程组而不是进程间的生产者-消费者情况的。在某些应用中划分了若干阶段，并且规定，除非所有的进程都就绪准备着手下一个阶段，否则任何进程都不能进入下一个阶段，可以通过在每个阶段的结尾安装一个 `屏障(barrier)` 来实现这种行为。当一个进程到达屏障时，它会被屏障所拦截，直到所有的屏障都到达为止。屏障可用于一组进程同步，如下图所示

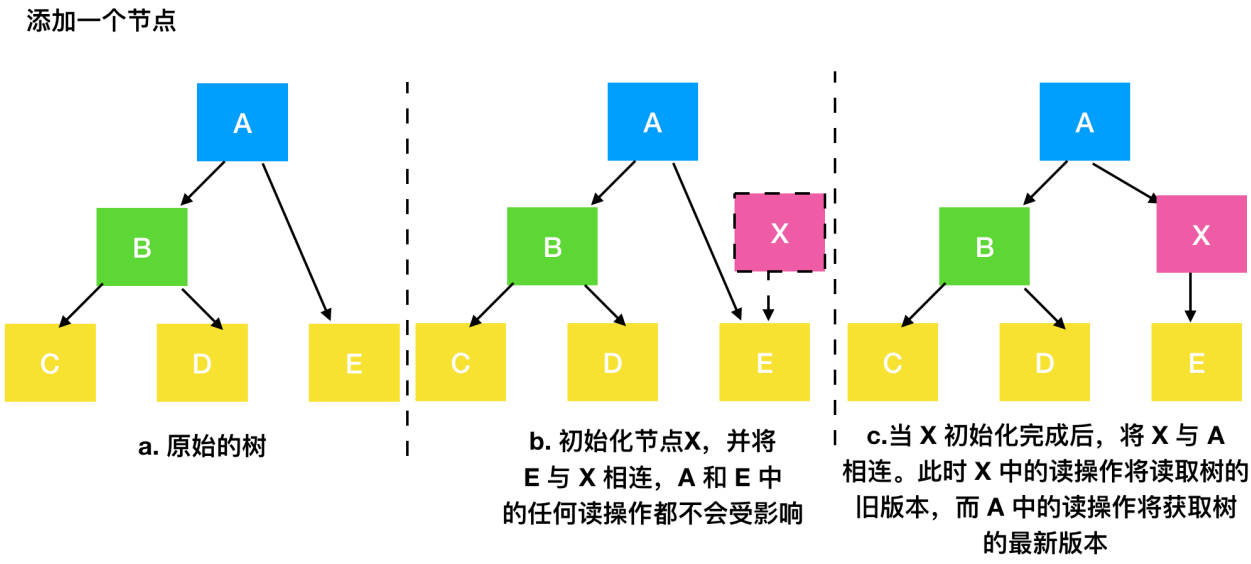


在上图中我们可以看到，有四个进程接近屏障，这意味着每个进程都在进行运算，但是还没有到达每个阶段的结尾。过了一段时间后，A、B、D 三个进程都到达了屏障，各自的进程被挂起，但此时还不能进入下一个阶段呢，因为进程 B 还没有执行完毕。结果，当最后一个 C 到达屏障后，这个进程组才能够进入下一个阶段。

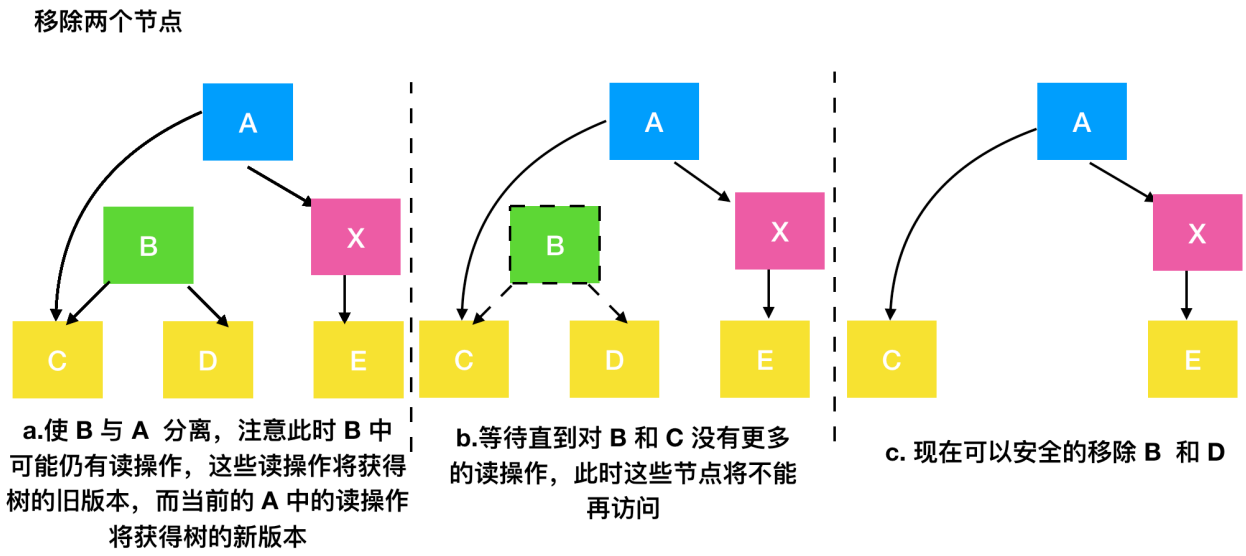
避免锁：读-复制-更新

最快的锁是根本没有锁。问题在于没有锁的情况下，我们是否允许对共享数据结构的并发读写进行访问。答案当然是不可以。假设进程 A 正在对一个数字数组进行排序，而进程 B 正在计算其平均值，而此时你进行 A 的移动，会导致 B 会多次读到重复值，而某些值根本没有遇到过。

然而，在某些情况下，我们可以允许写操作来更新数据结构，即便还有其他的进程正在使用。窍门在于确保每个读操作要么读取旧的版本，要么读取新的版本，例如下面的树



上面的树中，读操作从根部到叶子遍历整个树。加入一个新节点 X 后，为了实现这一操作，我们要让这个节点在树中可见之前使它"恰好正确"：我们对节点 X 中的所有值进行初始化，包括它的子节点指针。然后通过原子写操作，使 X 称为 A 的子节点。所有的读操作都不会读到前后不一致的版本



在上面的图中，我们接着移除 B 和 D。首先，将 A 的左子节点指针指向 C 。所有原本在 A 中的读操作将会后续读到节点 C ，而永远不会读到 B 和 D。也就是说，它们将只会读取到新版数据。同样，所有当前在 B 和 D 中的读操作将继续按照原始的数据结构指针并且读取旧版数据。所有操作均能正确运行，我们不需要锁住任何东西。而不需要锁住数据就能够移除 B 和 D 的主要原因就是 `读-复制-更新(Ready-Copy-Update,RCU)` ，将更新过程中的移除和再分配过程分离开。

调度

当一个计算机是多道程序设计系统时，会频繁的有很多进程或者线程来同时竞争 CPU 时间片。当两个或两个以上的进程/线程处于就绪状态时，就会发生这种情况。如果只有一个 CPU 可用，那么必须选择接下来哪个进程/线程可以运行。操作系统中有一个叫做 `调度程序(scheduler)` 的角色存在，它就是做这件事儿的，该程序使用的算法叫做 `调度算法(scheduling algorithm)` 。

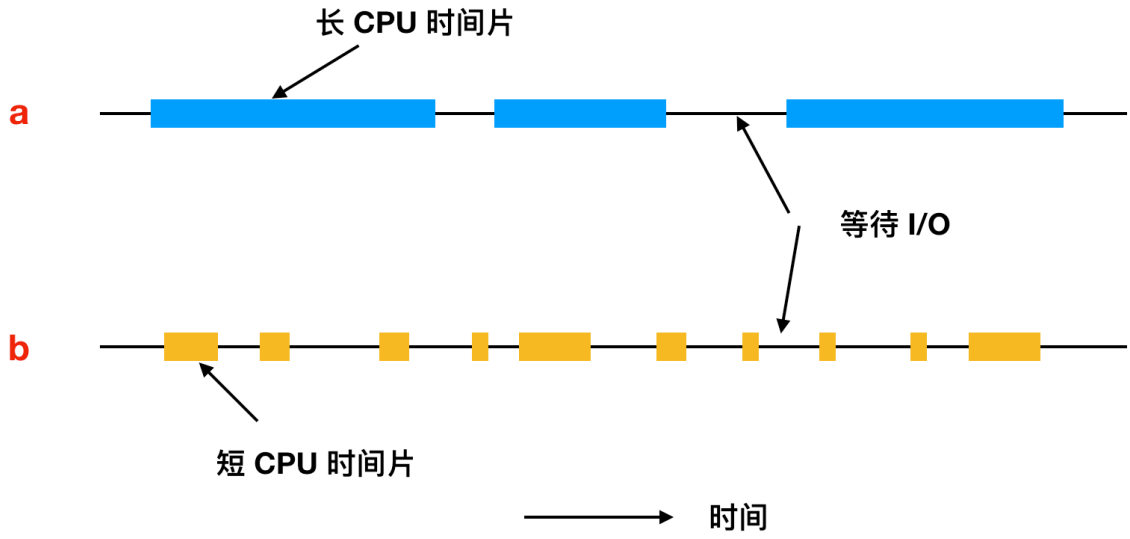
尽管有一些不同，但许多适用于进程调度的处理方法同样也适用于线程调度。当内核管理线程的时候，调度通常会以线程级别发生，很少或者根本不会考虑线程属于哪个进程。下面我们会首先专注于进程和线程的调度问题，然后会明确的介绍线程调度以及它产生的问题。

调度介绍

让我们回到早期以磁带上的卡片作为输入的批处理系统的时代，那时候的调度算法非常简单：依次运行磁带上的每一个作业。对于多道程序设计系统，会复杂一些，因为通常会有多个用户在等待服务。一些大型机仍然将 `批处理` 和 `分时服务` 结合使用，需要调度程序决定下一个运行的是一个批处理作业还是终端上的用户。由于在这些机器中 CPU 是稀缺资源，所以好的调度程序可以在提高性能和用户的满意度方面取得很大的成果。

进程行为

几乎所有的进程（磁盘或网络）I/O 请求和计算都是交替运行的



CPU 的使用和等待 I/O 交替出现

如上图所示，CPU 不停顿的运行一段时间，然后发出一个系统调用等待 I/O 读写文件。完成系统调用后，CPU 又开始计算，直到它需要读更多的数据或者写入更多的数据为止。当一个进程等待外部设备完成工作而被阻塞时，才是 I/O 活动。

上面 a 是 CPU 密集型进程；b 是 I/O 密集型进程进程，a 因为在计算的时间上花费时间更长，因此称为 `计算密集型(compute-bound)` 或者 `CPU 密集型(CPU-bound)`，b 因为I/O 发生频率比较快因此称为 `I/O 密集型(I/O-bound)`。计算密集型进程有较长的 CPU 集中使用和较小频度的 I/O 等待。I/O 密集型进程有较短的 CPU 使用时间和较频繁的 I/O 等待。注意到上面两种进程的区分关键在于 CPU 的时间占用而不是 I/O 的时间占用。I/O 密集型的原因是因为它们没有在 I/O 之间花费更多的计算、而不是 I/O 请求时间特别长。无论数据到达后需要花费多少时间，它们都需要花费相同的时间来发出读取磁盘块的硬件请求。

值得注意的是，随着 CPU 的速度越来越快，更多的进程倾向于 I/O 密集型。这种情况出现的原因是 CPU 速度的提升要远远高于硬盘。这种情况导致的结果是，未来对 I/O 密集型进程的调度处理似乎更为重要。这里的基本思想是，如果需要运行 I/O 密集型进程，那么就应该让它尽快得到机会，以便发出磁盘请求并保持磁盘始终忙碌。

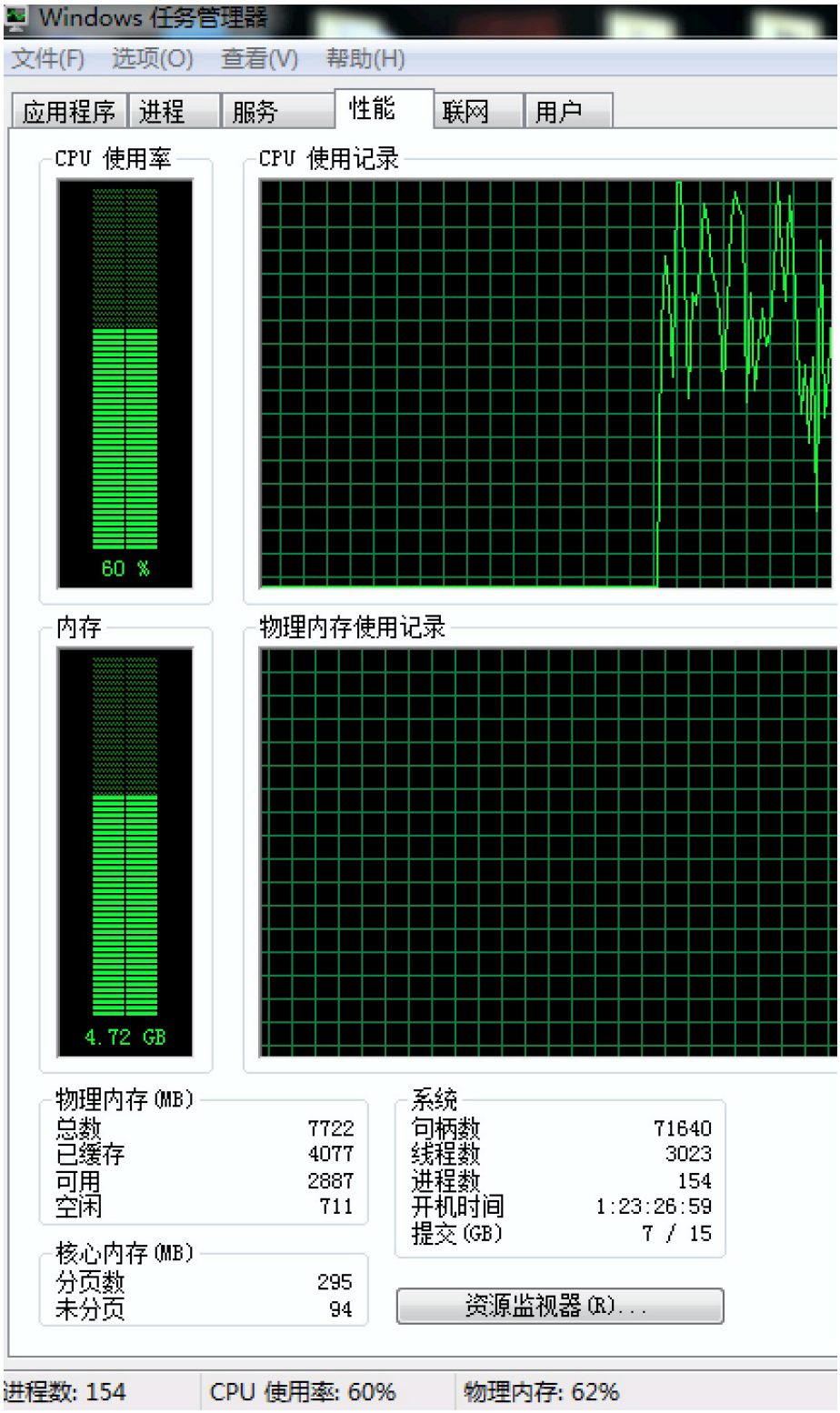
何时调度

第一个和调度有关的问题是 `何时进行调度决策`。存在着需要调度处理的各种情形。首先，在创建一个新进程后，需要决定是运行父进程还是子进程。因为二者的进程都处于就绪态下，这是正常的调度决策，可以任意选择，也就是说，调度程序可以任意的选择子进程或父进程开始运行。

第二，在进程退出时需要作出调度决定。因为此进程不再运行（因为它将不再存在），因此必须从就绪进程中选择其他进程运行。如果没有进程处于就绪态，系统提供的 `空闲进程` 通常会运行

什么是空闲进程

`空闲进程(system-supplied idle process)` 是 Microsoft 公司 windows 操作系统带有的系统进程，该进程是在各个处理器上运行的单个线程，它唯一的任务是在系统没有处理其他线程时占用处理器时间。System Idle Process 并不是一个真正的进程，它是 `核心虚拟` 出来的，多任务操作系统都存在。在没有可用的进程时，系统处于空运行状态，此时就是System Idle Process 在正在运行。你可以简单的理解成，它代表的是 CPU 的空闲状态，数值越大代表处理器越空闲，可以通过 Windows 任务管理器查看 Windows 中的 CPU 利用率



第三种情况是，当进程阻塞在 I/O 、信号量或其他原因时，必须选择另外一个进程来运行。有时，阻塞的原因会成为选择进程运行的关键因素。例如，如果 A 是一个重要进程，并且它正在等待 B 退出关键区域，让 B 退出关键区域从而使 A 得以运行。但是调度程序一般不会对这种情况进行考量。

第四点，当 I/O 中断发生时，可以做出调度决策。如果中断来自 I/O 设备，而 I/O 设备已经完成了其工作，那么那些等待 I/O 的进程现在可以继续运行。由调度程序来决定是否准备运行新的进程还是重新运行已经中断的进程。

如果硬件时钟以 50 或 60 Hz 或其他频率提供周期性中断，可以在每个时钟中断或第 k 个时钟中断处做出调度决策。根据如何处理时钟中断可以把调度算法可以分为两类。`非抢占式(nonpreemptive)` 调度算法挑选一个进程，让该进程运行直到被阻塞（阻塞在 I/O 上或等待另一个进程），或者直到该进程自动释放 CPU。即使该进程运行了若干个小时后，它也不会被强制挂起。这样会在时钟中断发生时不会进行调度。在处理完时钟中断后，如果没有更高优先级的进程等待，则被中断的进程会继续执行。

另外一种情况是 `抢占式` 调度算法，它会选择一个进程，并使其在最大固定时间内运行。如果在时间间隔结束后仍在运行，这个进程会被挂起，调度程序会选择其他进程来运行（前提是存在就绪进程）。进行抢占式调度需要在时间间隔结束时发生时钟中断，以将 CPU 的控制权交还给调度程序。如果没有可用的时钟，那么非抢占式就是唯一的选择。

调度算法的分类

毫无疑问，不同的环境下需要不同的调度算法。之所以出现这种情况，是因为不同的应用程序和不同的操作系统有不同的目标。也就是说，在不同的系统中，调度程序的优化也是不同的。这里有必要划分出三种环境

- `批处理(Batch)`
- `交互式(Interactive)`
- `实时(Real time)`

批处理系统广泛应用于商业领域，比如用来处理工资单、存货清单、账目收入、账目支出、利息计算、索赔处理和其他周期性作业。在批处理系统中，一般会选择使用非抢占式算法或者周期性比较长的抢占式算法。这种方法可以减少线程切换因此能够提升性能。

在交互式用户环境中，为了避免一个进程霸占 CPU 拒绝为其他进程服务，所以需要抢占式算法。即使没有进程有意要一直运行下去，但是，由于某个进程出现错误也有可能无限期的排斥其他所有进程。为了避免这种情况，抢占式也是必须的。服务器也属于此类别，因为它们通常为多个（远程）用户提供服务，而这些用户都非常着急。计算机用户总是很忙。

在实时系统中，抢占有时是不需要的，因为进程知道自己可能运行不了很长时间，通常很快的做完自己的工作并阻塞。实时系统与交互式系统的差别是，实时系统只运行那些用来推进现有应用的程序，而交互式系统是通用的，它可以运行任意的非协作甚至是有恶意的程序。

调度算法的目标

为了设计调度算法，有必要考虑一下什么是好的调度算法。有一些目标取决于环境（批处理、交互式或者实时）蛋大部分是适用于所有情况的，下面是一些需要考量的因素，我们会在下面一起讨论。



所有系统

在所有的情况中，公平是很重要的。对一个进程给予相较于其他等价的进程更多的 CPU 时间片对其他进程来说是不公平的。当然，不同类型的进程可以采用不同的处理方式。

与公平有关的是系统的强制执行，什么意思呢？如果某公司的薪资发放系统计划在本月的15号，那么碰上了疫情大家生活都很拮据，此时老板说要在14号晚上发放薪资，那么调度程序必须强制使进程执行 14 号晚上发放薪资的策略。

另一个共同的目标是保持系统的所有部分尽可能的忙碌。如果 CPU 和所有的 I/O 设备能够一直运行，那么相对于让某些部件空转而言，每秒钟就可以完成更多的工作。例如，在批处理系统中，调度程序控制哪个作业调入内存运行。在内存中既有一些 CPU 密集型进程又有一些 I/O 密集型进程是一个比较好的想法，好于先调入和运行所有的 CPU 密集型作业，然后在它们完成之后再调入和运行所有 I/O 密集型作业的做法。使用后者这种方式会在 CPU 密集型进程启动后，争夺 CPU ，而磁盘却在空转，而当 I/O 密集型进程启动后，它们又要为磁盘而竞争，CPU 却又在空转。。。。。。显然，通过结合 I/O 密集型和 CPU 密集型，能够使整个系统运行更流畅，效率更高。

批处理系统

通常有三个指标来衡量系统工作状态：**吞吐量、周转时间和 CPU 利用率**，`吞吐量(throughout)` 是系统每小时完成的作业数量。综合考虑，每小时完成 50 个工作要比每小时完成 40 个工作好。`周转时间(Turnaround time)` 是一种平均时间，它指的是从一个批处理提交开始直到作业完成时刻为止平均时间。该数据度量了用户要得到输出所需的平均等待时间。周转时间越小越好。

`CPU 利用率(CPU utilization)` 通常作为批处理系统上的指标。即使如此， CPU 利用率也不是一个好的度量指标，真正有价值的衡量指标是系统每小时可以完成多少作业（吞吐量），以及完成作业需要多长时间（周转时间）。把 CPU 利用率作为度量指标，就像是引擎每小时转动了多少次来比较汽车的性能一样。而且知道 CPU 的利用率什么时候接近 100% 要比什么什么时候要求得到更多的计算能力要有用。

交互式系统

对于交互式系统，则有不同的指标。最重要的是尽量 `减少响应时间` 。这个时间说的是从执行指令开始到得到结果的时间。再有后台进程运行（例如，从网络上读取和保存 E-mail 文件）的个人计算机上，用户请求启动一个程序或打开一个文件应该优先于后台的工作。能够让所有的交互式请求首先运行的就是一个好的服务。

一个相关的问题是 `均衡性(proportionality)` ，用户对做一件事情需要多长时间总是有一种固定（不过通常不正确）的看法。当认为一个请求很复杂需要较多时间时，用户会认为很正常并且可以接受，但是一个很简单的程序却花费了很长的运行时间，用户就会很恼怒。可以拿彩印和复印来举出一个简单的例子，彩印可能需要1分钟的时间，但是用户觉得复杂并且愿意等待一分钟，相反，复印很简单只需要 5 秒钟，但是复印机花费 1 分钟却没有完成复印操作，用户就会很焦躁。

实时系统

实时系统则有着和交互式系统不同的考量因素，因此也就有不同的调度目标。实时系统的特点是 `必须满足最后的截止时间` 。例如，如果计算机控制着以固定速率产生数据的设备，未能按时运行的话可能会导致数据丢失。因此，实时系统中最重要的需求是满足所有（或大多数）时间期限。

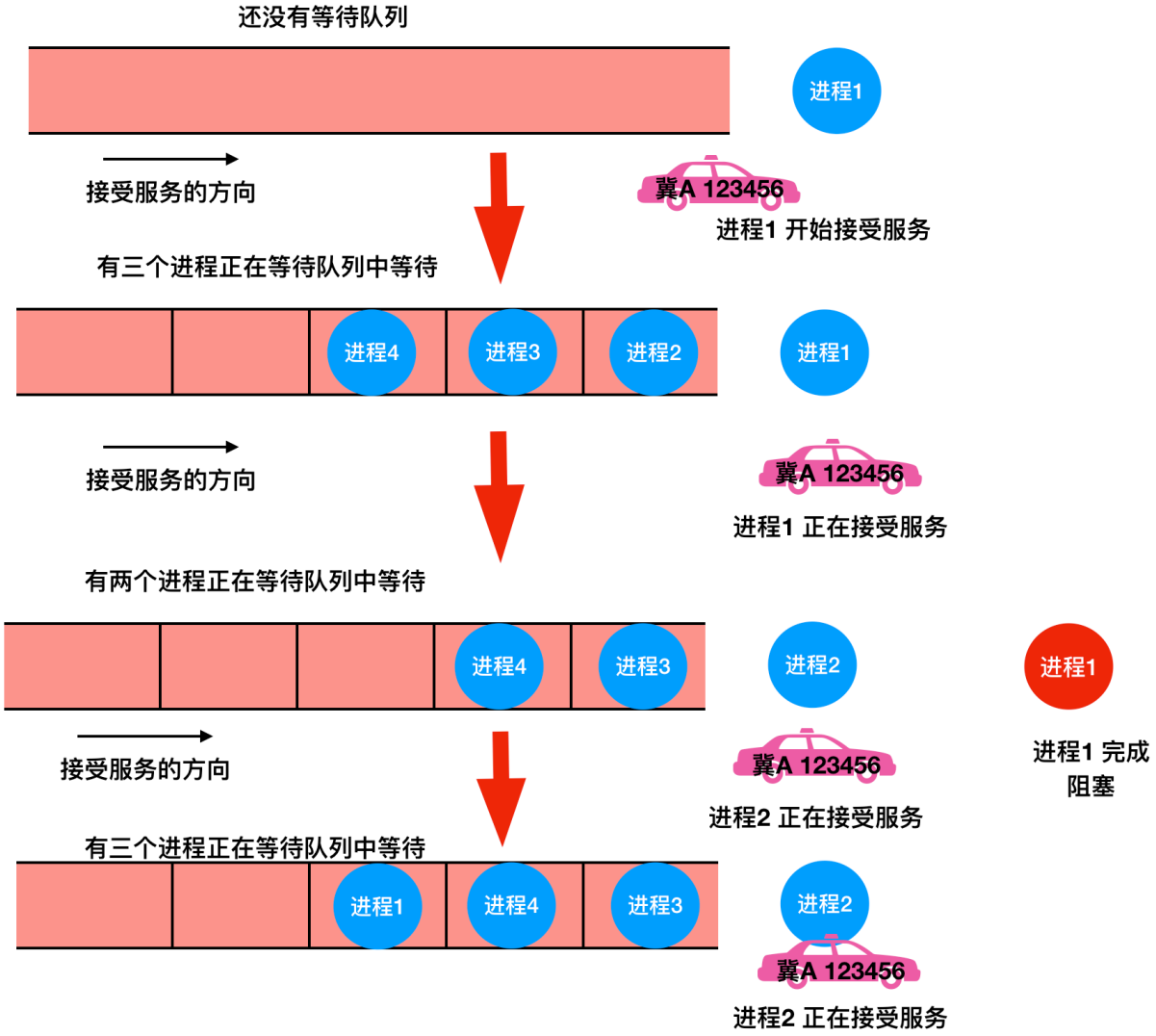
在一些实事系统中，特别是涉及到多媒体的， `可预测性很重要` 。偶尔不能满足最后的截止时间不重要，但是如果音频多媒体运行不稳定，声音质量会持续恶化。视频也会造成问题，但是耳朵要比眼睛敏感很多。为了避免这些问题，进程调度必须能够高度可预测的而且是有规律的。

批处理中的调度

现在让我们把目光从一般性的调度转换为特定的调度算法。下面我们会探讨在批处理中的调度。

先来先服务

很像是先到先得。。。可能最简单的非抢占式调度算法的设计就是 `先来先服务(first-come,first-serverd)` 。使用此算法，将按照请求顺序为进程分配 CPU。最基本的，会有一个就绪进程的等待队列。当第一个任务从外部进入系统时，将会立即启动并允许运行任意长的时间。它不会因为运行时间太长而中断。当其他作业进入时，它们排到就绪队列尾部。当正在运行的进程阻塞，处于等待队列的第一个进程就开始运行。当一个阻塞的进程重新处于就绪态时，它会像一个新到达的任务，会排在队列的末尾，即排在所有进程最后。

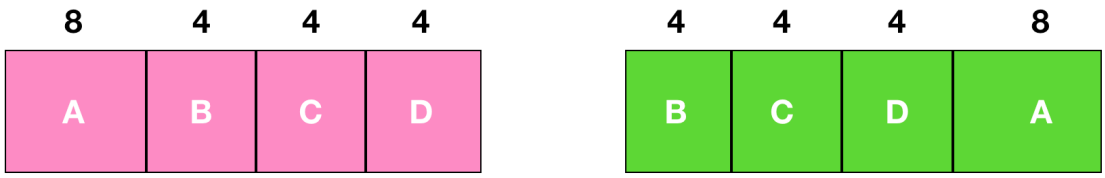


这个算法的强大之处在于易于理解和编程，在这个算法中，一个单链表记录了所有就绪进程。要选取一个进程运行，只要从该队列的头部移走一个进程即可；要添加一个新的作业或者阻塞一个进程，只要把这个作业或进程附加在队列的末尾即可。这是很简单的一种实现。

不过，先来先服务也是有缺点的，那就是没有优先级的关系，试想一下，如果有 100 个 I/O 进程正在排队，第 101 个是一个 CPU 密集型进程，那岂不是需要等 100 个 I/O 进程运行完毕才会等到一个 CPU 密集型进程运行，这在实际情况下根本不可能，所以需要优先级或者抢占式进程的出现来优先选择重要的进程运行。

最短作业优先

批处理中，第二种调度算法是 `最短作业优先(Shortest Job First)`，我们假设运行时间已知。例如，一家保险公司，因为每天要做类似的工作，所以人们可以相当精确地预测处理 1000 个索赔的一批作业需要多长时间。当输入队列中有若干个同等重要的作业被启动时，调度程序应使用最短优先作业算法



a

按原有次数运行 4 个作业

b

按最短作业优先次序进行

如上图 a 所示，这里有 4 个作业 A、B、C、D ，运行时间分别为 8、4、4、4 分钟。若按图中的次序运行，则 A 的周转时间为 8 分钟，B 为 12 分钟，C 为 16 分钟，D 为 20 分钟，平均时间内为 14 分钟。

现在考虑使用最短作业优先算法运行 4 个作业，如上图 b 所示，目前的周转时间分别为 4、8、12、20，平均为 11 分钟，可以证明最短作业优先是最优的。考虑有 4 个作业的情况，其运行时间分别为 a、b、c、d。第一个作业在时间 a 结束，第二个在时间 a + b 结束，以此类推。平均周转时间为 (4a + 3b + 2c + d) / 4 。显然 a 对平均值的影响最大，所以 a 应该是最短优先作业，其次是 b，然后是 c ，最后是 d 它就只能影响自己的周转时间了。

> 需要注意的是，在所有的进程都可以运行的情况下，最短作业优先的算法才是最优的。

最短剩余时间优先

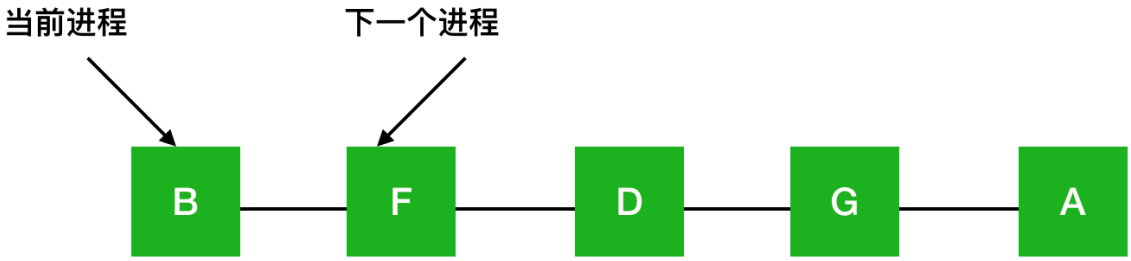
最短作业优先的抢占式版本被称作为 `最短剩余时间优先(Shortest Remaining Time Next)` 算法。使用这个算法，调度程序总是选择剩余运行时间最短的那个进程运行。当一个新作业到达时，其整个时间同当前进程的剩余时间做比较。如果新的进程比当前运行进程需要更少的时间，当前进程就被挂起，而运行新的进程。这种方式能够使短期作业获得良好的服务。

交互式系统中的调度

交互式系统中在个人计算机、服务器和其他系统中都是很常用的，所以有必要来探讨一下交互式调度

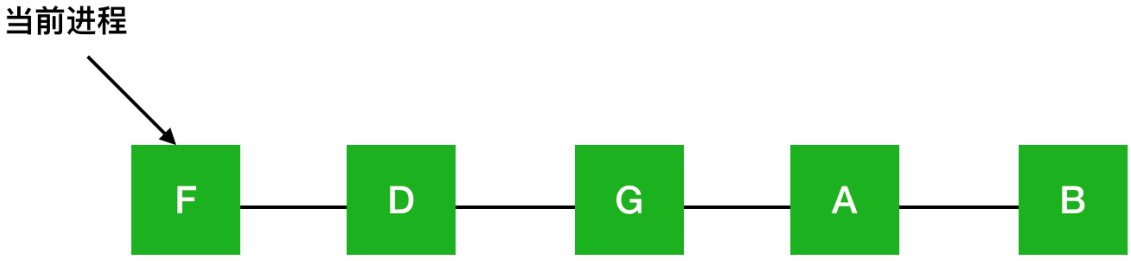
轮询调度

一种最古老、最简单、最公平并且最广泛使用的算法就是 `轮询算法(round-robin)` 。每个进程都会被分配一个时间段，称为 `时间片(quantum)` ，在这个时间片内允许进程运行。如果时间片结束时进程还在运行的话，则抢占一个 CPU 并将其分配给另一个进程。如果进程在时间片结束前阻塞或结束，则 CPU 立即进行切换。轮询算法比较容易实现。调度程序所做的就是维护一个可运行进程的列表，就像下图中的 a，当一个进程用完时间片后就被移到队列的末尾，就像下图的 b。



a

可运行进程列表



b

进程 B 用完时间片后的可运行列表

时间片轮询调度中唯一有意思的一点就是时间片的长度。从一个进程切换到另一个进程需要一定的时间进行管理处理，包括保存寄存器的值和内存映射、更新不同的表格和列表、清除和重新调入内存高速缓存等。这种切换称作 `进程间切换(process switch)` 和 `上下文切换(context switch)`。如果进程间的切换时间需要 1ms，其中包括内存映射、清除和重新调入高速缓存等，再假设时间片设为 4 ms，那么 CPU 在做完 4 ms 有用的工作之后，CPU 将花费 1 ms 来进行进程间的切换。因此，CPU 的时间片会浪费 20% 的时间在管理开销上。耗费巨大。

为了提高 CPU 的效率，我们把时间片设置为 100 ms。现在时间的浪费只有 1%。但是考虑会发现下面的情况，如果在一个非常短的时间内到达 50 个请求，并且对 CPU 有不同的需求，此时会发生什么？50 个进程都被放在可运行进程列表中。如果 CP画U 是空闲的，第一个进程会立即开始执行，第二个直到 100 ms 以后才会启动，以此类推。不幸的是最后一个进程需要等待 5 秒才能获得执行机会。大部分用户都会觉得对于一个简短的指令运行 5 秒中是很慢的。如果队列末尾的某些请求只需要几号秒钟的运行时间的话，这种设计就非常糟糕了。

另外一个因素是如果时间片设置长度要大于 CPU 使用长度，那么抢占就不会经常发生。相反，在时间片用完之前，大多数进程都已经阻塞了，那么就会引起进程间的切换。消除抢占可提高性能，因为进程切换仅在逻辑上必要时才发生，即流程阻塞且无法继续时才发生。

结论可以表述如下：将上下文切换时间设置得太短会导致过多的进程切换并降低 CPU 效率，但设置时间太长会导致一个短请求很长时间得不到响应。最好的切换时间是在 20 - 50 毫秒之间设置。

优先级调度

轮询调度假设了所有的进程是同等重要的。但事实情况可能不是这样。例如，在一所大学中的等级制度，首先是院长，然后是教授、秘书、后勤人员，最后是学生。这种将外部情况考虑在内就实现了 `优先级调度(priority scheduling)`

某学院等级制度

院长 > 教授 > 秘书 > 后勤人员 > 学生

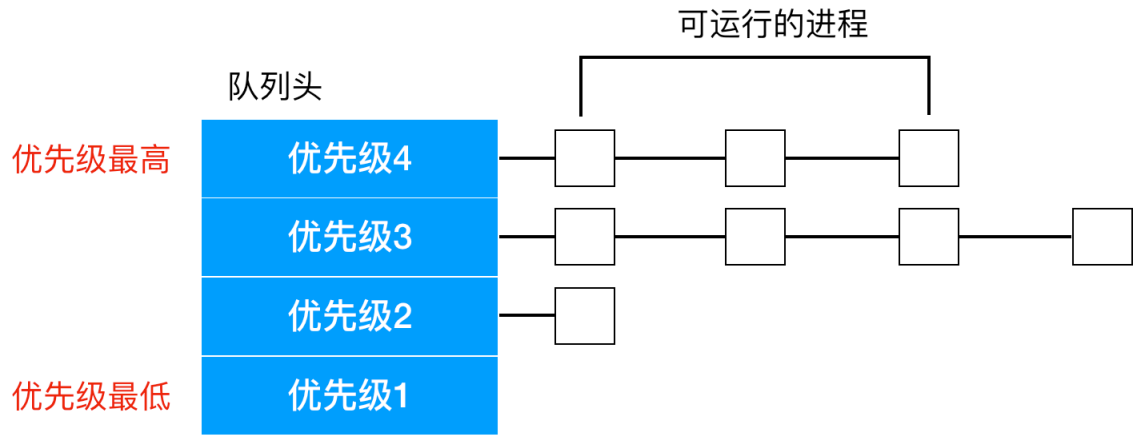
它的基本思想很明确，每个进程都被赋予一个优先级，优先级高的进程优先运行。

但是也不意味着高优先级的进程能够永远一直运行下去，调度程序会在每个时钟中断期间降低当前运行进程的优先级。如果此操作导致其优先级降低到下一个最高进程的优先级以下，则会发生进程切换。或者，可以为每个进程分配允许运行的最大时间间隔。当时间间隔用完后，下一个高优先级的进程会得到运行的机会。

可以静态或者动态的为进程分配优先级。在一台军用计算机上，可以把将军所启动的进程设为优先级 100，上校为 90，少校为 80，上尉为 70，中尉为 60，以此类推。UNIX 中有一条命令为 `nice`，它允许用户为了照顾他人而自愿降低自己进程的优先级，但是一般没人用。

优先级也可以由系统动态分配，用于实现某种目的。例如，有些进程为 I/O 密集型，其多数时间用来等待 I/O 结束。当这样的进程需要 CPU 时，应立即分配 CPU，用来启动下一个 I/O 请求，这样就可以在另一个进程进行计算的同时执行 I/O 操作。这类 I/O 密集型进程长时间的等待 CPU 只会造成它长时间占用内存。使 I/O 密集型进程获得较好的服务的一种简单算法是，将其优先级设为 `1/f`，f 为该进程在上一时间片中所占的部分。一个在 50 ms 的时间片中只使用 1 ms 的进程将获得优先级 50，而在阻塞之前用掉 25 ms 的进程将具有优先级 2，而使用掉全部时间片的进程将得到优先级 1。

可以很方便的将一组进程按优先级分成若干类，并且在各个类之间采用优先级调度，而在各类进程的内部采用轮转调度。下面展示了一个四个优先级类的系统

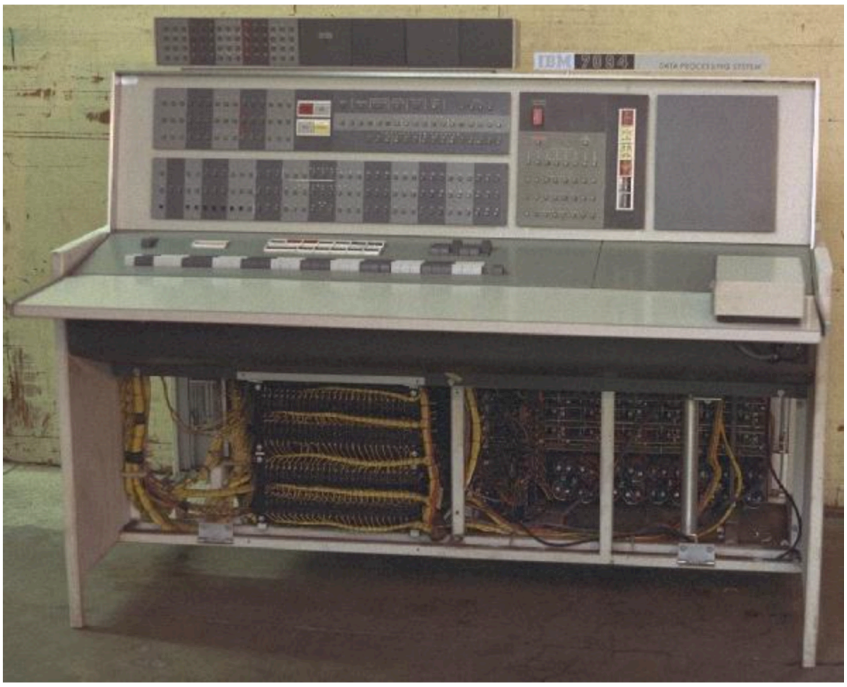


它的调度算法主要描述如下：上面存在优先级为 4 类的可运行进程，首先会按照轮转法为每个进程运行一个时间片，此时不理会较低优先级的进程。若第 4 类进程为空，则按照轮询的方式运行第三类进程。若第 4 类和第 3 类进程都为空，则按照轮转法运行第 2 类进程。如果不对优先级进行调整，则低优先级的进程很容易产生饥饿现象。

多级队列

最早使用优先级调度的系统是 `CTSS(Compatible TimeSharing System)`。CTSS 是一种兼容分时系统，它有一个问题就是进程切换太慢，其原因是 IBM 7094 内存只能放进一个进程。

> IBM 是哥伦比亚大学计算机中心在 1964 - 1968 年的计算机



CTSS 在每次切换前都需要将当前进程换出到磁盘，并从磁盘上读入一个新进程。CTSS 的设计者很快就认识到，为 CPU 密集型进程设置较长的时间片比频繁地分给他们很短的时间要更有效（减少交换次数）。另一方面，如前所述，长时间片的进程又会影响到响应时间，解决办法是设置优先级类。属于最高优先级的进程运行一个时间片，次高优先级进程运行 2 个时间片，再下面一级运行 4 个时间片，以此类推。当一个进程用完分配的时间片后，它被移到下一类。

最短进程优先

对于批处理系统而言，由于最短作业优先常常伴随着最短响应时间，所以如果能够把它用于交互式进程，那将是非常好的。在某种程度上，的确可以做到这一点。交互式进程通常遵循下列模式：等待命令、执行命令、等待命令、执行命令。。。如果我们把每个命令的执行都看作一个分离的作业，那么我们可以通过首先运行最短的作业来使响应时间最短。这里唯一的问题是如何从当前可运行进程中找出最短的那一个进程。

一种方式是根据进程过去的行为进行推测，并执行估计运行时间最短的那一个。假设每个终端上每条命令的预估运行时间为 `T0`，现在假设测量到其下一次运行时间为 `T1`，可以用两个值的加权来改进估计时间，即 `aT0+ (1- 1)T1`。通过选择 a 的值，可以决定是尽快忘掉老的运行时间，还是在一段长时间内始终记住它们。当 a = 1/2 时，可以得到下面这个序列

$$T_0,\quad \frac{T_0}{2}+\frac{T_1}{2},\quad \frac{T_0}{4}+\frac{T_1}{4}+\frac{T_2}{2},\quad \frac{T_0}{8}+\frac{T_1}{8}+\frac{T_2}{4}+\frac{T_3}{2}$$

可以看到，在三轮过后，T0 在新的估计值中所占比重下降至 1/8。

有时把这种通过当前测量值和先前估计值进行加权平均从而得到下一个估计值的技术称作 `老化(aging)`。这种方法会使用很多预测值基于当前值的情况。

保证调度

一种完全不同的调度方法是对用户做出明确的性能保证。一种实际而且容易实现的保证是：若用户工作时有 n 个用户登录，则每个用户将获得 CPU 处理能力的 1/n。类似地，在一个有 n 个进程运行的单用户系统中，若所有的进程都等价，则每个进程将获得 1/n 的 CPU 时间。

彩票调度

对用户进行承诺并在随后兑现承诺是一件好事，不过很难实现。但是存在着一种简单的方式，有一种既可以给出预测结果而又有一种比较简单的实现方式的算法，就是 `彩票调度(lottery scheduling)` 算法。

其基本思想是为进程提供各种系统资源（例如 CPU 时间）的彩票。当做出一个调度决策的时候，就随机抽出一张彩票，拥有彩票的进程将获得该资源。在应用到 CPU 调度时，系统可以每秒持有 50 次抽奖，每个中奖者将获得比如 20 毫秒的 CPU 时间作为奖励。

`George Orwell` 关于 **所有的进程是平等的，但是某些进程能够更平等一些**。一些重要的进程可以给它们额外的彩票，以便增加他们赢得的机会。如果出售了 100 张彩票，而且有一个进程持有了它们中的 20 张，它就会有 20% 的机会去赢得彩票中奖。在长时间的运行中，它就会获得 20% 的CPU。相反，对于优先级调度程序，很难说明拥有优先级 40 究竟是什么意思，这里的规则很清楚，拥有彩票 f 份额的进程大约得到系统资源的 f 份额。

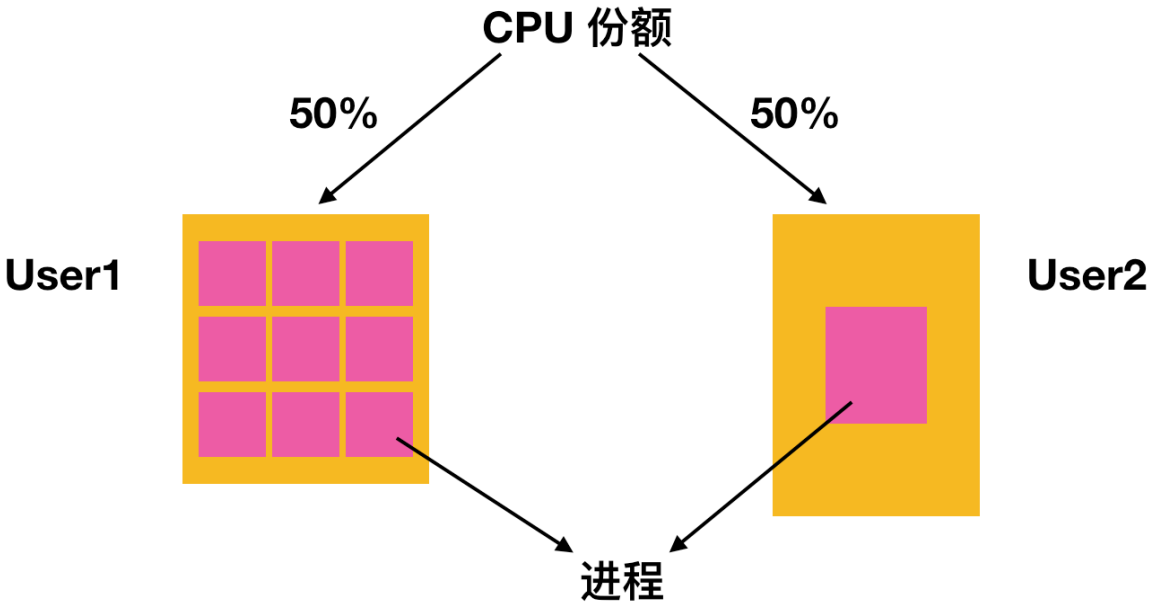
如果希望进程之间协作的话可以交换它们之间的票据。例如，客户端进程给服务器进程发送了一条消息后阻塞，客户端进程可能会把自己所有的票据都交给服务器，来增加下一次服务器运行的机会。当服务完成后，它会把彩票还给客户端让其有机会再次运行。事实上，如果没有客户机，服务器也根本不需要彩票。

> 可以把彩票理解为 buff，这个 buff 有 15% 的几率能让你产生 `速度之靴` 的效果。

公平分享调度

到目前为止，我们假设被调度的都是各个进程自身，而不用考虑该进程的拥有者是谁。结果是，如果用户 1 启动了 9 个进程，而用户 2 启动了一个进程，使用轮转或相同优先级调度算法，那么用户 1 将得到 90 % 的 CPU 时间，而用户 2 将之得到 10 % 的 CPU 时间。

为了阻止这种情况的出现，一些系统在调度前会把进程的拥有者考虑在内。在这种模型下，每个用户都会分配一些CPU 时间，而调度程序会选择进程并强制执行。因此如果两个用户每个都会有 50% 的 CPU 时间片保证，那么无论一个用户有多少个进程，都将获得相同的 CPU 份额。



实时系统中的调度

`实时系统(real-time)` 是一个时间扮演了重要作用的系统。典型的，一种或多种外部物理设备发给计算机一个服务请求，而计算机必须在一个确定的时间范围内恰当的做出反应。例如，在 CD 播放器中的计算机会获得从驱动器过来的位流，然后必须在非常短的时间内将位流转换为音乐播放出来。如果计算时间过长，那么音乐就会听起来有异常。再比如说医院特别护理部门的病人监护装置、飞机中的自动驾驶系统、列车中的烟雾警告装置等，在这些例子中，正确但是却缓慢的响应要比没有响应甚至还糟糕。

实时系统可以分为两类，`硬实时(hard real time)` 和 `软实时(soft real time)` 系统，前者意味着必须要满足绝对的截止时间；后者的含义是虽然不希望偶尔错失截止时间，但是可以容忍。在这两种情形中，实时都是通过把程序划分为一组进程而实现的，其中每个进程的行为是可预测和提前可知的。这些进程一般寿命较短，并且极快的运行完成。在检测到一个外部信号时，调度程序的任务就是按照满足所有截止时间的要求调度进程。

实时系统中的事件可以按照响应方式进一步分类为 `周期性(以规则的时间间隔发生)` 事件或 `非周期性(发生时间不可预知)` 事件。一个系统可能要响应多个周期性事件流，根据每个事件处理所需的时间，可能甚至无法处理所有事件。例如，如果有 m 个周期事件，事件 i 以周期 Pi 发生，并需要 Ci 秒 CPU 时间处理一个事件，那么可以处理负载的条件是

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

只有满足这个条件的实时系统称为 `可调度的` ，这意味着它实际上能够被实现。一个不满足此检验标准的进程不能被调度，因为这些进程共同需要的 CPU 时间总和大于 CPU 能提供的时间。

举一个例子，考虑一个有三个周期性事件的软实时系统，其周期分别是 100 ms、200 m 和 500 ms。如果这些事件分别需要 50 ms、30 ms 和 100 ms 的 CPU 时间，那么该系统时可调度的，因为 0.5 + 0.15 + 0.2 < 1。如果此时有第四个事件加入，其周期为 1 秒，那么此时这个事件如果不超过 150 ms，那么仍然是可以调度的。忽略上下文切换的时间。

实时系统的调度算法可以是静态的或动态的。前者在系统开始运行之前做出调度决策；后者在运行过程中进行调度决策。只有在可以提前掌握所完成的工作以及必须满足的截止时间等信息时，静态调度才能工作，而动态调度不需要这些限制。

调度策略和机制

到目前为止，我们隐含的假设系统中所有进程属于不同的分组用户并且进程间存在相互竞争 CPU 的情况。通常情况下确实如此，但有时也会发生一个进程会有很多子进程并在其控制下运行的情况。例如，一个数据库管理系统进程会有很多子进程。每一个子进程可能处理不同的请求，或者每个子进程实现不同的功能（如请求分析、磁盘访问等）。主进程完全可能掌握哪一个子进程最重要（或最紧迫），而哪一个最不重要。但是，以上讨论的调度算法中没有一个算法从用户进程接收有关的调度决策信息，这就导致了调度程序很少能够做出最优的选择。

解决问题的办法是将 `调度机制(scheduling mechanism)` 和 `调度策略(scheduling policy)` 分开，这是长期一贯的原则。这也就意味着调度算法在某种方式下被参数化了，但是参数可以被用户进程填写。让我们首先考虑数据库的例子。假设内核使用优先级调度算法，并提供了一条可供进程设置优先级的系统调用。这样，尽管父进程本身并不参与调度，但它可以控制如何调度子进程的细节。调度机制位于内核，而调度策略由用户进程决定，调度策略和机制分离是一种关键性思路。

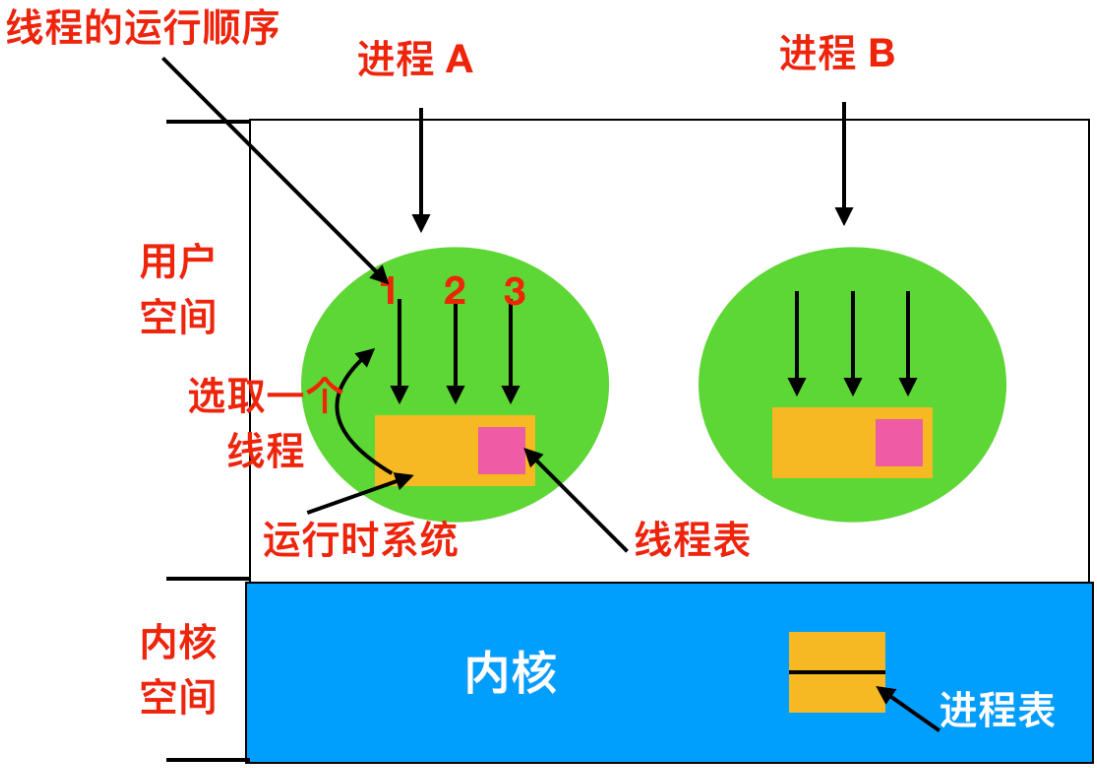
线程调度

当若干进程都有多个线程时，就存在两个层次的并行：进程和线程。在这样的系统中调度处理有本质的差别，这取决于所支持的是用户级线程还是内核级线程（或两者都支持）。

首先考虑用户级线程，由于内核并不知道有线程存在，所以内核还是和以前一样地操作，选取一个进程，假设为 A，并给予 A 以时间片控制。A 中的线程调度程序决定哪个线程运行。假设为 A1。由于多道线程并不存在时钟中断，所以这个线程可以按其意愿任意运行多长时间。如果该线程用完了进程的全部时间片，内核就会选择另一个进程继续运行。

在进程 A 终于又一次运行时，线程 A1 会接着运行。该线程会继续耗费 A 进程的所有时间，直到它完成工作。不过，线程运行不会影响到其他进程。其他进程会得到调度程序所分配的合适份额，不会考虑进程 A 内部发生的事情。

现在考虑 A 线程每次 CPU 计算的工作比较少的情况，例如：在 50 ms 的时间片中有 5 ms 的计算工作。于是，每个线程运行一会儿，然后把 CPU 交回给线程调度程序。这样在内核切换到进程 B 之前，就会有序列 A1,A2,A3,A1,A2,A3,A1,A2,A3,A1 。 如下所示

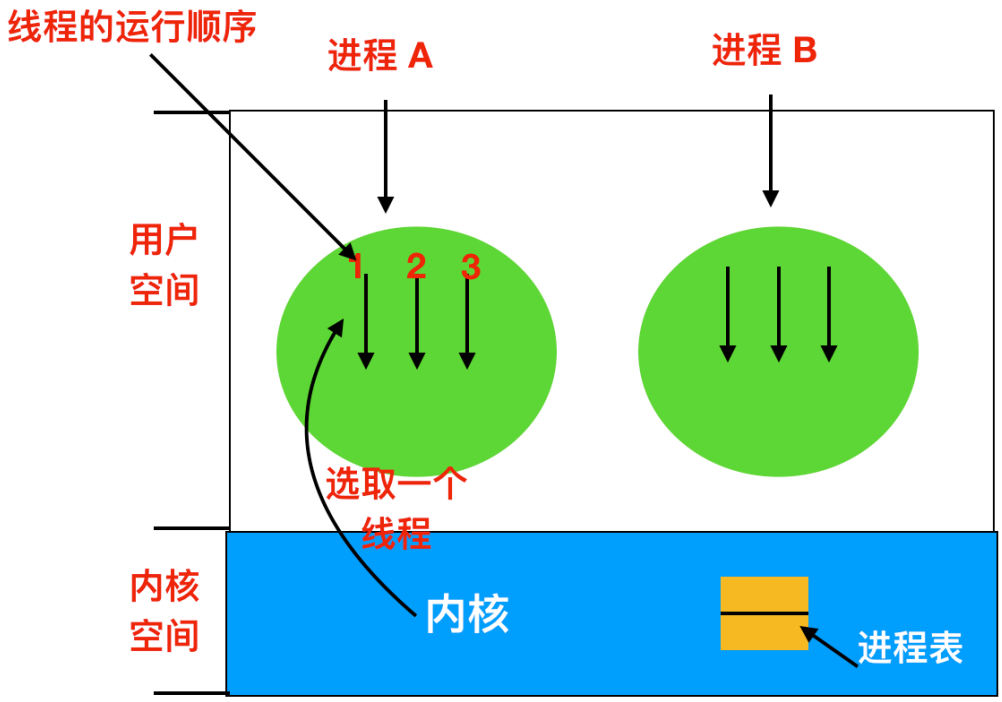


可能： A1,A2,A3,A1,A2,A3

不可能： A1,B1,A2,B2,A3,B3

运行时系统使用的调度算法可以是上面介绍算法的任意一种。从实用方面考虑，轮转调度和优先级调度更为常用。唯一的局限是，缺乏一个时钟中断运行过长的线程。但由于线程之间的合作关系，这通常也不是问题。

现在考虑使用内核线程的情况，内核选择一个特定的线程运行。它不用考虑线程属于哪个进程，不过如果有必要的话，也可以这么做。对被选择的线程赋予一个时间片，而且如果超过了时间片，就会强制挂起该线程。一个线程在 50 ms 的时间片内，5 ms 之后被阻塞，在 30 ms 的时间片中，线程的顺序会是 A1,B1,A2,B2,A3,B3。如下图所示



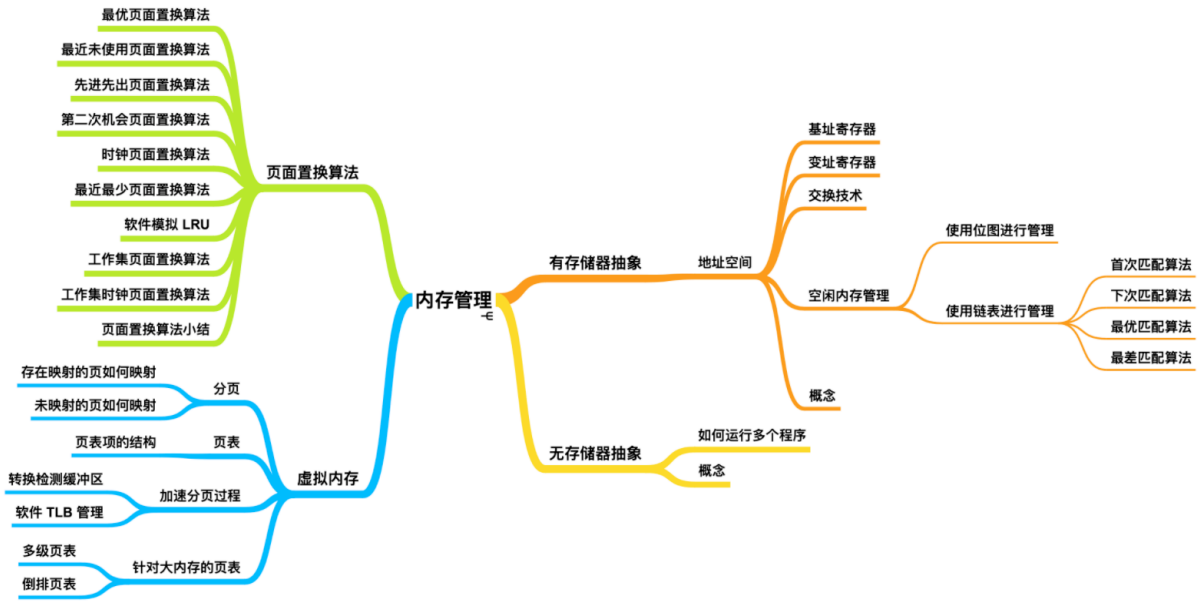
可能： A1,A2,A3,A1,A2,A3

也可能： A1,B1,A2,B2,A3,B3

用户级线程和内核级线程之间的主要差别在于 性能 。用户级线程的切换需要少量的机器指令（想象一下Java程序的线程切换），而内核线程需要完整的上下文切换，修改内存映像，使高速缓存失效，这会导致了若干数量级的延迟。另一方面，在使用内核级线程时，一旦线程阻塞在 I/O 上就不需要在用户级线程中那样将整个进程挂起。

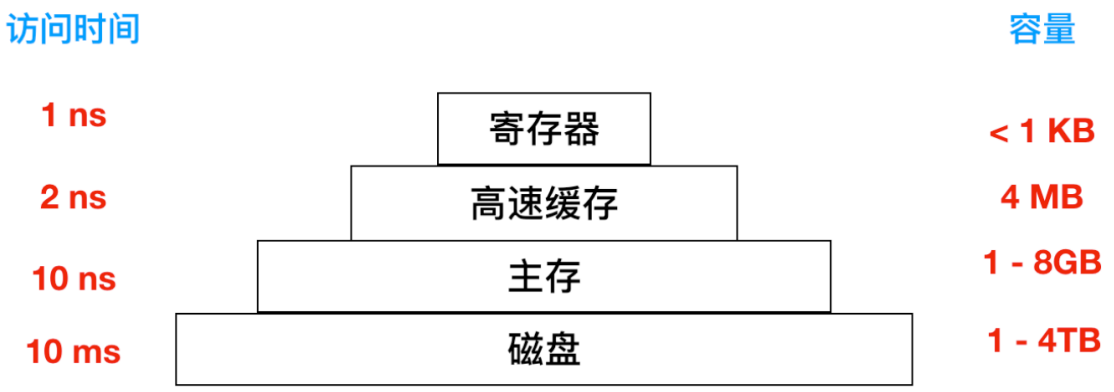
从进程 A 的一个线程切换到进程 B 的一个线程，其消耗要远高于运行进程 A 的两个线程（涉及修改内存映像，修改高速缓存），内核对这种切换的消耗是了解到，可以通过这些信息作出决定。

内存



`主存(RAM)` 是一件非常重要的资源，必须要认真对待内存。虽然目前大多数内存的增长速度要比 IBM 7094 要快的多，但是，程序大小的增长要比内存的增长还快很多。`不管存储器有多大，程序大小的增长速度比内存容量的增长速度要快的多` 。下面我们就来探讨一下操作系统是如何创建内存并管理他们的。

经过多年的研究发现，科学家提出了一种 `分层存储器体系(memory hierarchy)` ，下面是分层体系的分类



存储层次结构

位于顶层的存储器速度最快，但是相对容量最小，成本非常高。层级结构向下，其访问速度会变慢，但是容量会变大，相对造价也就越便宜。（所以个人感觉相对存储容量来说，访问速度是更重要的）

操作系统中管理内存层次结构的部分称为 `内存管理器(memory manager)` ，它的主要工作是有效的管理内存，记录哪些内存是正在使用的，在进程需要时分配内存以及在进程完成时回收内存。所有现代操作系统都提供内存管理。

下面我们会对不同的内存管理模型进行探讨，从简单到复杂，由于最低级别的缓存是由硬件进行管理的，所以我们主要探讨主存模型和如何对主存进行管理。

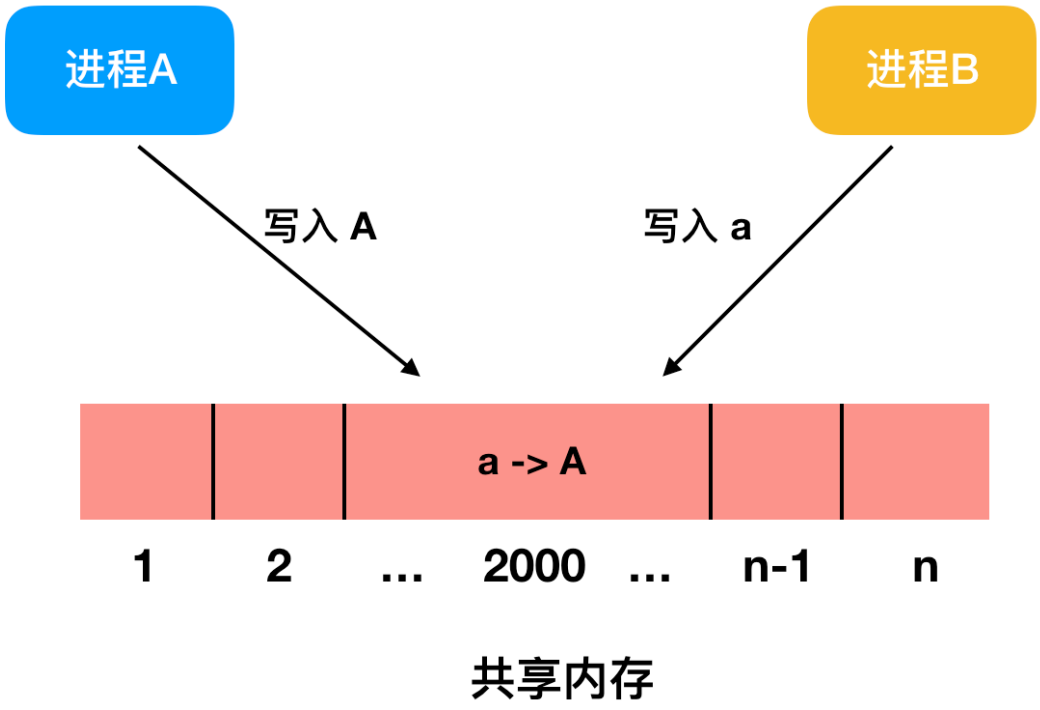
无存储器抽象

最简单的存储器抽象是 `无存储器` 。早期大型计算机（20 世纪 60 年代之前），小型计算机（20 世纪 70 年代之前）和个人计算机（20 世纪 80 年代之前）都没有存储器抽象。每一个程序都直接访问物理内存。当一个程序执行如下命令：

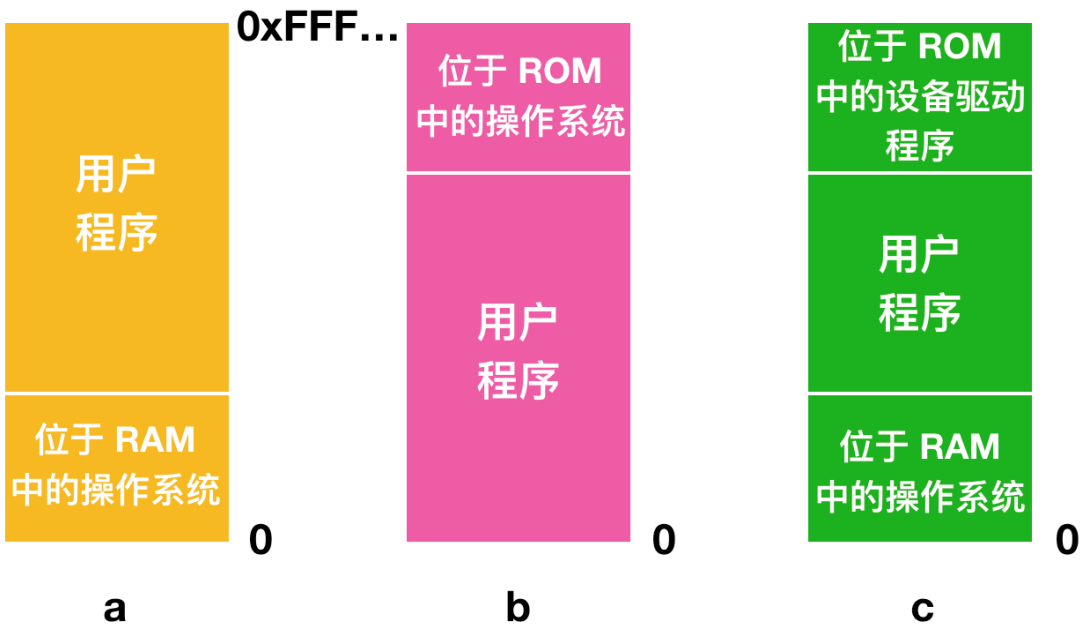
```
1    MOV REGISTER1, 1000
```

计算机会把位置为 1000 的物理内存中的内容移到 `REGISTER1` 中。因此呈现给程序员的内存模型就是物理内存，内存地址从 0 开始到内存地址的最大值中，每个地址中都会包含一个 8 位位数的内存单元。

所以这种情况下的计算机不可能会有两个应用程序 `同时` 在内存中。如果第一个程序向内存地址 2000 的这个位置写入了一个值，那么此值将会替换第二个程序 2000 位置上的值，所以，同时运行两个应用程序是行不通的，两个程序会立刻崩溃。



不过即使存储器模型就是物理内存，还是存在一些可变体的。下面展示了三种变体



在上图 a 中，操作系统位于 `RAM(Random Access Memory)` 的底部，或像是图 b 一样位于 `ROM(Read-Only Memory)` 顶部；而在图 c 中，设备驱动程序位于顶端的 ROM 中，而操作系统位于底部的 RAM 中。图 a 的模型以前用在大型机和小型机上，但现在已经很少使用了；图 b 中的模型一般用于掌上电脑或者是嵌入式系统中。第三种模型就应用在早期个人计算机中了。ROM 系统中的一部分成为 `BIOS (Basic Input Output System)`。模型 a 和 c 的缺点是用户程序中的错误可能会破坏操作系统，可能会导致灾难性的后果。

按照这种方式组织系统时，通常同一个时刻只能有一个进程正在运行。一旦用户键入了一个命令，操作系统就把需要的程序从磁盘复制到内存中并执行；当进程运行结束后，操作系统在用户终端显示提示符并等待新的命令。收到新的命令后，它把新的程序装入内存，覆盖前一个程序。

在没有存储器抽象的系统中实现 `并行性` 一种方式是使用多线程来编程。由于同一进程中的多线程内部共享同一内存映像，那么实现并行也就不是问题了。但是这种方式却并没有被广泛采纳，因为人们通常希望能够在同一时间内运行没有关联的程序，而这正是线程抽象所不能提供的。

运行多个程序

但是，即便没有存储器抽象，同时运行多个程序也是有可能的。操作系统只需要把当前内存中所有内容保存到磁盘文件中，然后再把程序读入内存即可。只要某一时刻内存只有一个程序在运行，就不会有冲突的情况发生。

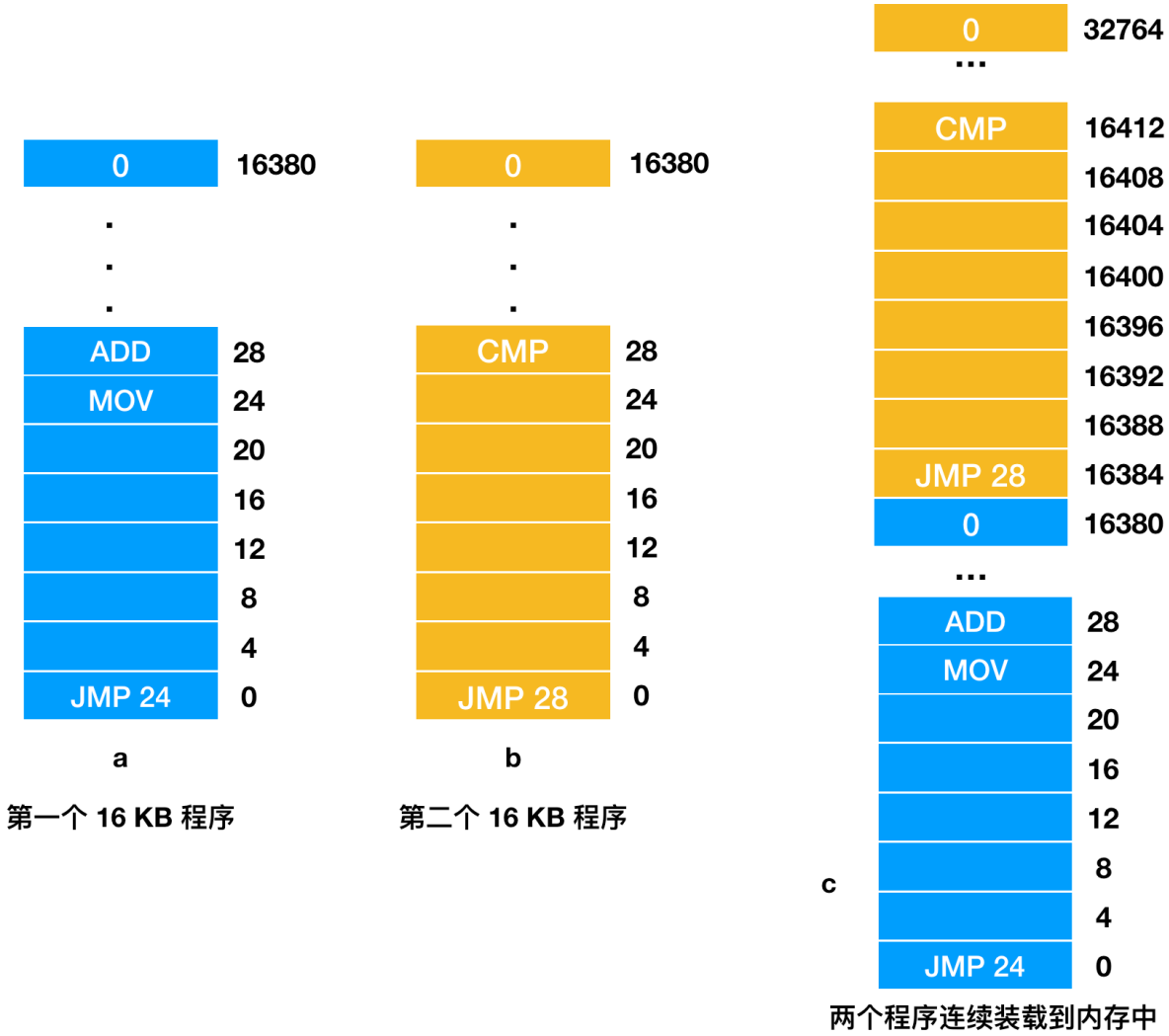
在额外特殊硬件的帮助下，即使没有交换功能，也可以并行的运行多个程序。IBM 360 的早期模型就是这样解决的

> System/360是 IBM 在1964年4月7日，推出的划时代的大型电脑，这一系列是世界上首个指令集可兼容计算机。



在 IBM 360 中，内存被划分为 2KB 的区域块，每块区域被分配一个 4 位的保护键，保护键存储在 CPU 的 `特殊寄存器(SFR)` 中。一个内存为 1 MB 的机器只需要 512 个这样的 4 位寄存器，容量总共为 256 字节 (这个会算吧) `PSW(Program Status Word, 程序状态字)` 中有一个 4 位码。一个运行中的进程如果访问键与其 PSW 中保存的码不同，360 硬件会捕获这种情况。因为只有操作系统可以修改保护键，这样就可以防止进程之间、用户进程和操作系统之间的干扰。

这种解决方式是有一个缺陷。如下所示，假设有两个程序，每个大小各为 16 KB



从图上可以看出，这是两个不同的 16KB 程序的装载过程，a 程序首先会跳转到地址 24，那里是一条 `MOV` 指令，然而 b 程序会首先跳转到地址 28，地址 28 是一条 `CMP` 指令。这是两个程序被 `先后` 加载到内存中的情况，假如这两个程序被同时加载到内存中并且从 0 地址处开始执行，内存的状态就如上面 c 图所示，程序装载完成开始运行，第一个程序首先从 0 地址处开始运行，执行 JMP 24 指令，然后依次执行后面的指令（许多指令没有画出），一段时间后第一个程序执行完毕，然后开始执行第二个程序。第二个程序的第一条指令是 28，这条指令会使程序跳转到第一个程序的 `ADD` 处，而不是事先设定好的跳转指令 CMP，由于这种不正确访问，可能会造成程序崩溃。

上面两个程序的执行过程中有一个核心问题，那就是都引用了绝对物理地址，这不是我们想要看到的。我们想要的是每一个程序都会引用一个私有的本地地址。IBM 360 在第二个程序装载到内存中的时候会使用一种称为 `静态重定位(static relocation)` 的技术来修改它。它的工作流程如下：当一个程序被加载到 16384 地址时，常数 16384 被加到每一个程序地址上（所以 `JMP 28` 会变为 `JMP 16412`）。虽然这个机制在 `不出错误` 的情况下是可行的，但这不是一种通用的解决办法，同时会减慢装载速度。更近一步来讲，它需要所有可执行程序中的额外信息，以指示哪些包含（可重定位）地址，哪些不包含（可重定位）地址。毕竟，上图 b 中的 JMP 28 可以被重定向（被修改），而类似 `MOV REGISTER1,28` 会把数字 28 移到 REGISTER 中则不会重定向。所以，`装载器(loader)` 需要一定的能力来辨别地址和常数。

一种存储器抽象：地址空间

把物理内存暴露给进程会有几个主要的缺点：第一个问题是，如果用户程序可以寻址内存的每个字节，它们就可以很容易的破坏操作系统，从而使系统 `停止运行` （除非使用 IBM 360 那种 lock-and-key 模式或者特殊的硬件进行保护）。即使在只有一个用户进程运行的情况下，这个问题也存在。

第二点是，这种模型想要运行多个程序是很困难的（如果只有一个 CPU 那就是顺序执行）。在个人计算机上，一般会打开很多应用程序，比如输入法、电子邮件、浏览器，这些进程在不同时刻会有一个进程正在运行，其他应用程序可以通过鼠标来唤醒。在系统中没有物理内存的情况下很难实现。

地址空间的概念

如果要使多个应用程序同时运行在内存中，必须要解决两个问题： `保护` 和 `重定位` 。我们来看 IBM 360 是如何解决的：第一种解决方式是用 `保护密钥标记内存块` ，并将执行过程的密钥与提取的每个存储字的密钥进行比较。这种方式只能解决第一种问题（破坏操作系统），但是不能解决多进程在内存中同时运行的问题。

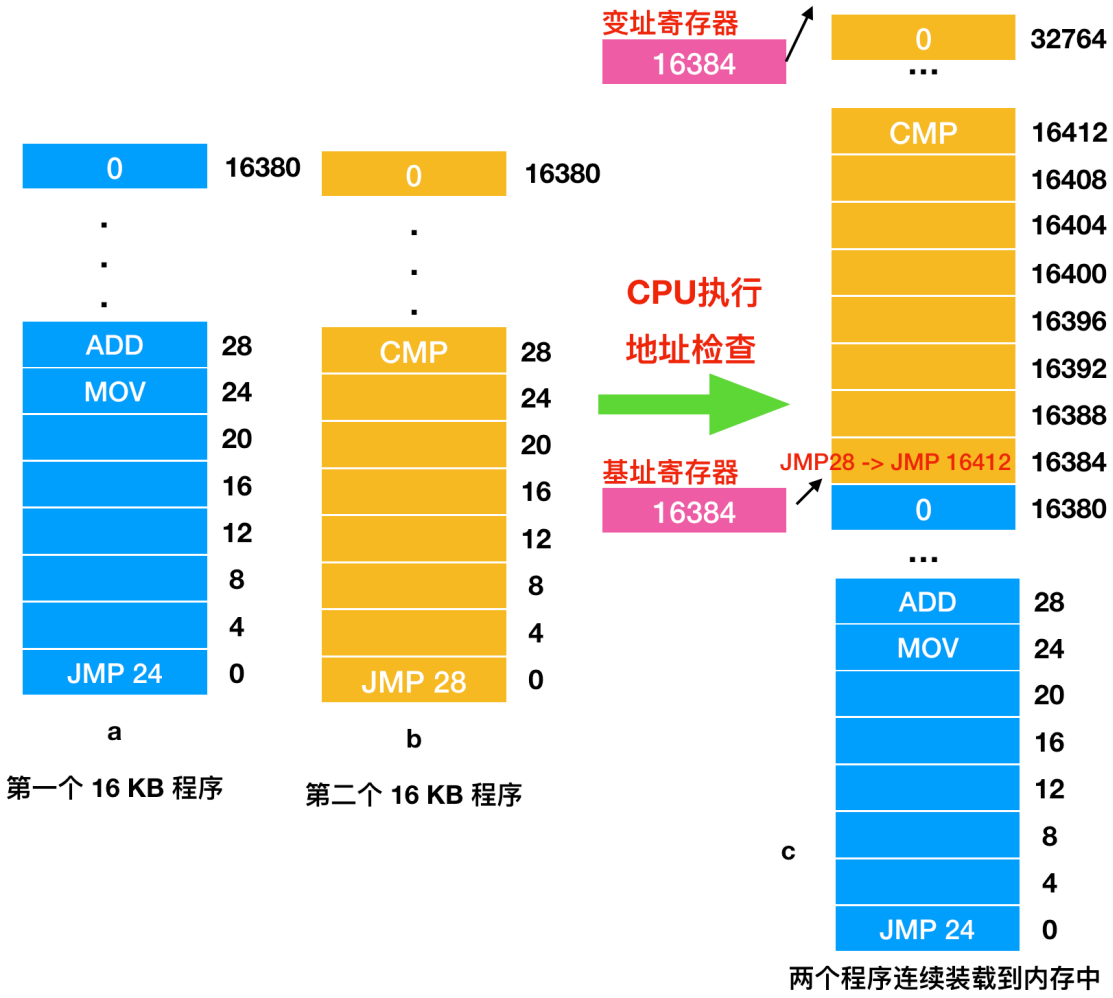
还有一种更好的方式是创造一个存储器抽象： `地址空间(the address space)` 。就像进程的概念创建了一种抽象的 CPU 来运行程序，地址空间也创建了一种抽象内存供程序使用。地址空间是进程可以用来寻址内存的地址集。每个进程都有它自己的地址空间，独立于其他进程的地址空间，但是某些进程会希望可以共享地址空间。

基址寄存器和变址寄存器

最简单的办法是使用 `动态重定位(dynamic relocation)` 技术，它就是通过一种简单的方式将每个进程的地址空间映射到物理内存的不同区域。从 `CDC 6600(世界上最早的超级计算机)` 到 `Intel 8088(原始 IBM PC 的核心)` 所使用的经典办法是给每个 CPU 配置两个特殊硬件寄存器，通常叫做 `基址寄存器(basic register)` 和 `变址寄存器(limit register)` 。当使用基址寄存器和变址寄存器时，程序会装载到内存中的连续位置并且在装载期间无需重定位。当一个进程运行时，程序的起始物理地址装载到基址寄存器中，程序的长度则装载到变址寄存器中。在上图 c 中，当一个程序运行时，装载到这些硬件寄存器中的基址和变址寄存器的值分别是 0 和 16384。当第二个程序运行时，这些值分别是 16384 和 32768。如果第三个 16 KB 的程序直接装载到第二个程序的地址之上并且运行，这时基址寄存器和变址寄存器的值会是 32768 和 16384。那么我们可以总结下

- 基址寄存器：存储数据内存的起始位置
- 变址寄存器：存储应用程序的长度。

每当进程引用内存以获取指令或读取、写入数据时，CPU 都会自动将 `基址值` 添加到进程生成的地址中，然后再将其发送到内存总线上。同时，它检查程序提供的地址是否大于或等于 `变址寄存器` 中的值。如果程序提供的地址要超过变址寄存器的范围，那么会产生错误并中止访问。这样，对上图 c 中执行 `JMP 28` 这条指令后，硬件会把它解释为 `JMP 16412` ，所以程序能够跳到 CMP 指令，过程如下



使用基址寄存器和变址寄存器是给每个进程提供私有地址空间的一种非常好的方法，因为每个内存地址在送到内存之前，都会先加上基址寄存器的内容。在很多实际系统中，对基址寄存器和变址寄存器都会以一定的方式加以保护，使得只有操作系统可以修改它们。在 `CDC 6600` 中就提供了对这些寄存器的保护，但在 `Intel 8088` 中则没有，甚至没有变址寄存器。但是，Intel 8088 提供了许多基址寄存器，使程序的代码和数据可以被独立的重定位，但是对于超出范围的内存引用没有提供保护。

所以你可以知道使用基址寄存器和变址寄存器的缺点，在每次访问内存时，都会进行 `ADD` 和 `CMP` 运算。CMP 指令可以执行的很快，但是加法就会相对慢一些，除非使用特殊的加法电路，否则加法因进位传播时间而变慢。

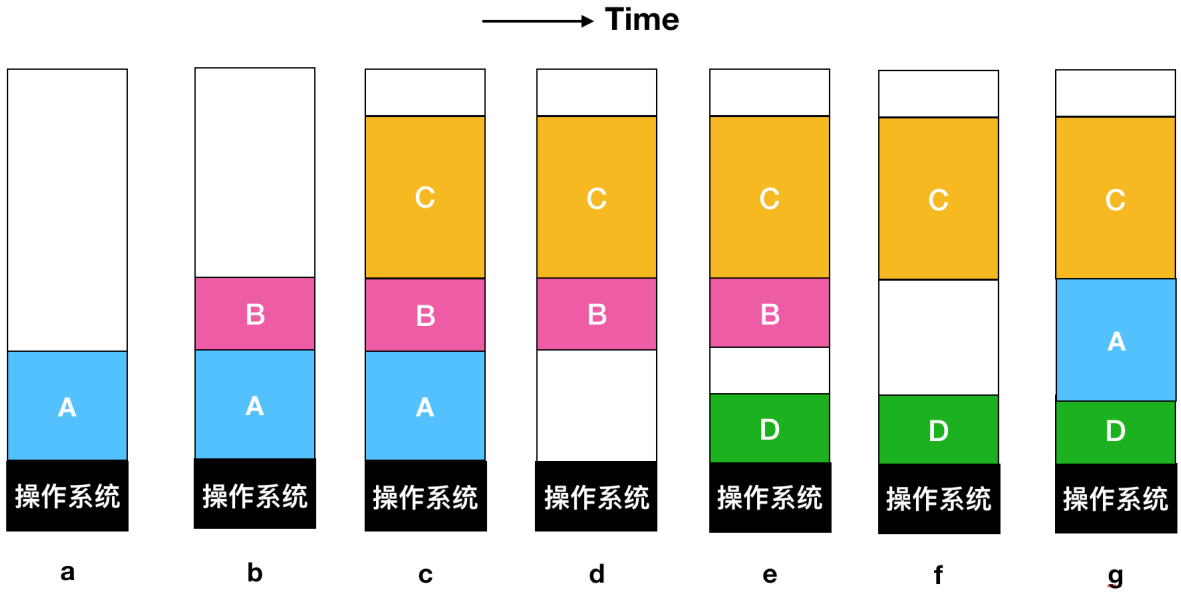
交换技术

如果计算机的物理内存足够大来容纳所有的进程，那么之前提及的方案或多或少是可行的。但是实际上，所有进程需要的 RAM 总容量要远远高于内存的容量。在 Windows、OS X、或者 Linux 系统中，在计算机完成启动（Boot）后，大约有 50 - 100 个进程随之启动。例如，当一个 Windows 应用程序被安装后，它通常会发出命令，以便在后续系统启动时，将启动一个进程，这个进程除了检查应用程序的更新外不做任何操作。一个简单的应用程序可能会占用 `5 - 10MB` 的内存。其他后台进程会检查电子邮件、网络连接以及许多其他诸如此类的任务。这一切都会发生在 `第一个用户` 启动之前。如今，像是 `Photoshop` 这样的重要用户应用程序仅仅需要 500 MB 来启动，但是一旦它们开始处理数据就需要许多 GB 来处理。从结果上来看，将所有进程始终保持在内存中需要大量内存，如果内存不足，则无法完成。

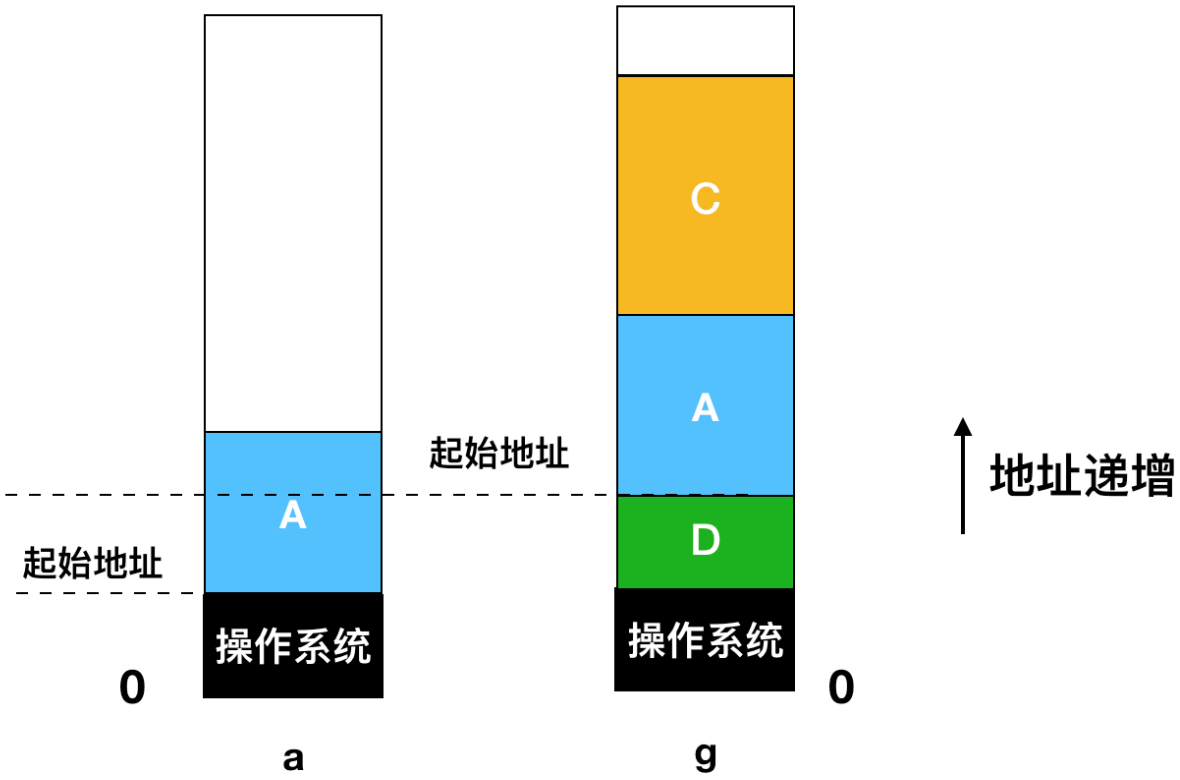
所以针对上面内存不足的问题，提出了两种处理方式：最简单的一种方式就是 交换(swapping) 技术，即把一个进程完整的调入内存，然后再内存中运行一段时间，再把它放回磁盘。空闲进程会存储在磁盘中，所以这些进程在没有运行时不会占用太多内存。另外一种策略叫做 虚拟内存(virtual memory)，虚拟内存技术能够允许应用程序部分的运行在内存中。下面我们首先先探讨一下交换

交换过程

下面是一个交换过程



刚开始的时候，只有进程 A 在内存中，然后从创建进程 B 和进程 C 或者从磁盘中把它们换入内存，然后在图 d 中，A 被换出内存到磁盘中，最后 A 重新进来。因为图 g 中的进程 A 现在到了不同的位置，所以在装载过程中需要被重新定位，或者在交换程序时通过软件来执行；或者在程序执行期间通过硬件来重定位。基址寄存器和变址寄存器就适用于这种情况。



交换在内存创建了多个 `空闲区(hole)`，内存会把所有的空闲区尽可能向下移动合并成为一个大的空闲区。这项技术称为 `内存紧缩(memory compaction)` 。但是这项技术通常不会使用，因为这项技术回消耗很多 CPU 时间。例如，在一个 16GB 内存的机器上每 8ns 复制 8 字节，它紧缩全部的内存大约要花费 16s。

有一个值得注意的问题是，当进程被创建或者换入内存时应该为它分配多大的内存。如果进程被创建后它的大小是固定的并且不再改变，那么分配策略就比较简单：操作系统会准确的按其需要的大小进行分配。

但是如果进程的 `data segment` 能够自动增长，例如，通过动态分配堆中的内存，肯定会出现问题。这里还是再提一下什么是 `data segment` 吧。从逻辑层面操作系统把数据分成不同的 `段(不同的区域)` 来存储：

- **代码段（codesegment/textsegment）：**

又称文本段，用来存放指令，运行代码的一块内存空间

此空间大小在代码运行前就已经确定

内存空间一般属于只读，某些架构的代码也允许可写

在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

- **数据段（datasegment）：**

可读可写

存储初始化的全局变量和初始化的 static 变量

数据段中数据的生存期是随程序持续性（随进程持续性） 随进程持续性：进程创建就存在，进程死亡就消失

- **bss段（bsssegment）：**

可读可写

存储未初始化的全局变量和未初始化的 static 变量

bss 段中数据的生存期随进程持续性

bss 段中的数据一般默认为0

- **rodata段：**

只读数据 比如 printf 语句中的格式字符串和开关语句的跳转表。也就是常量区。例如，全局作用域中的 const int ival = 10，ival 存放在 .rodata 段；再如，函数局部作用域中的 printf("Hello world %d\n", c); 语句中的格式字符串 "Hello world %d\n"，也存放在 .rodata 段。

- **栈（stack）：**

可读可写

存储的是函数或代码中的局部变量(非 static 变量)

栈的生存期随代码块持续性，代码块运行就给你分配空间，代码块结束，就自动回收空间

- **堆（heap）：**

可读可写

存储的是程序运行期间动态分配的 malloc/realloc 的空间

堆的生存期随进程持续性，从 malloc/realloc 到 free 一直存在

下面是我们用 Borland C++ 编译过后的结果

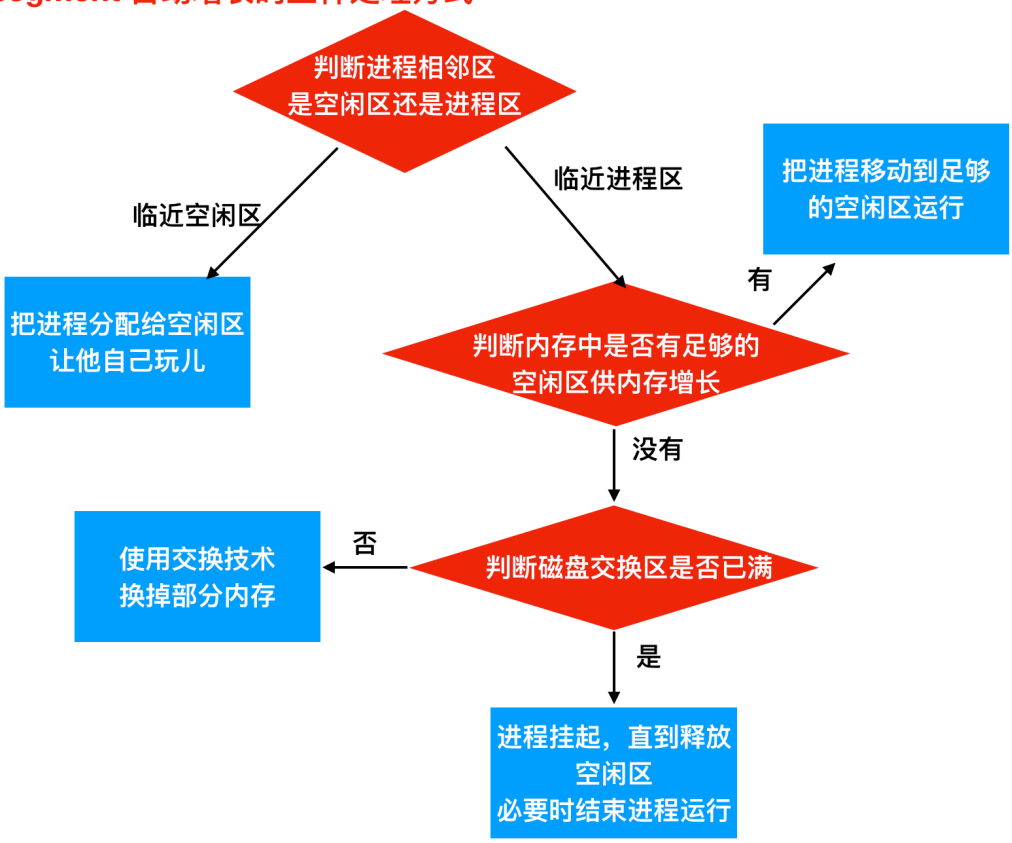
```
1   _TEXT segment dword public use32 'CODE'
2   _TEXT ends
3   _DATA segment dword public use32 'DATA'
4   _DATA ends
5   _BSS  segment dword public use32 'BSS'
6   _BSS  ends
```

> 段定义(segment) 是用来区分或者划分范围区域的意思。汇编语言的 segment 伪指令表示段定义的起始，ends 伪指令表示段定义的结束。段定义是一段连续的内存空间

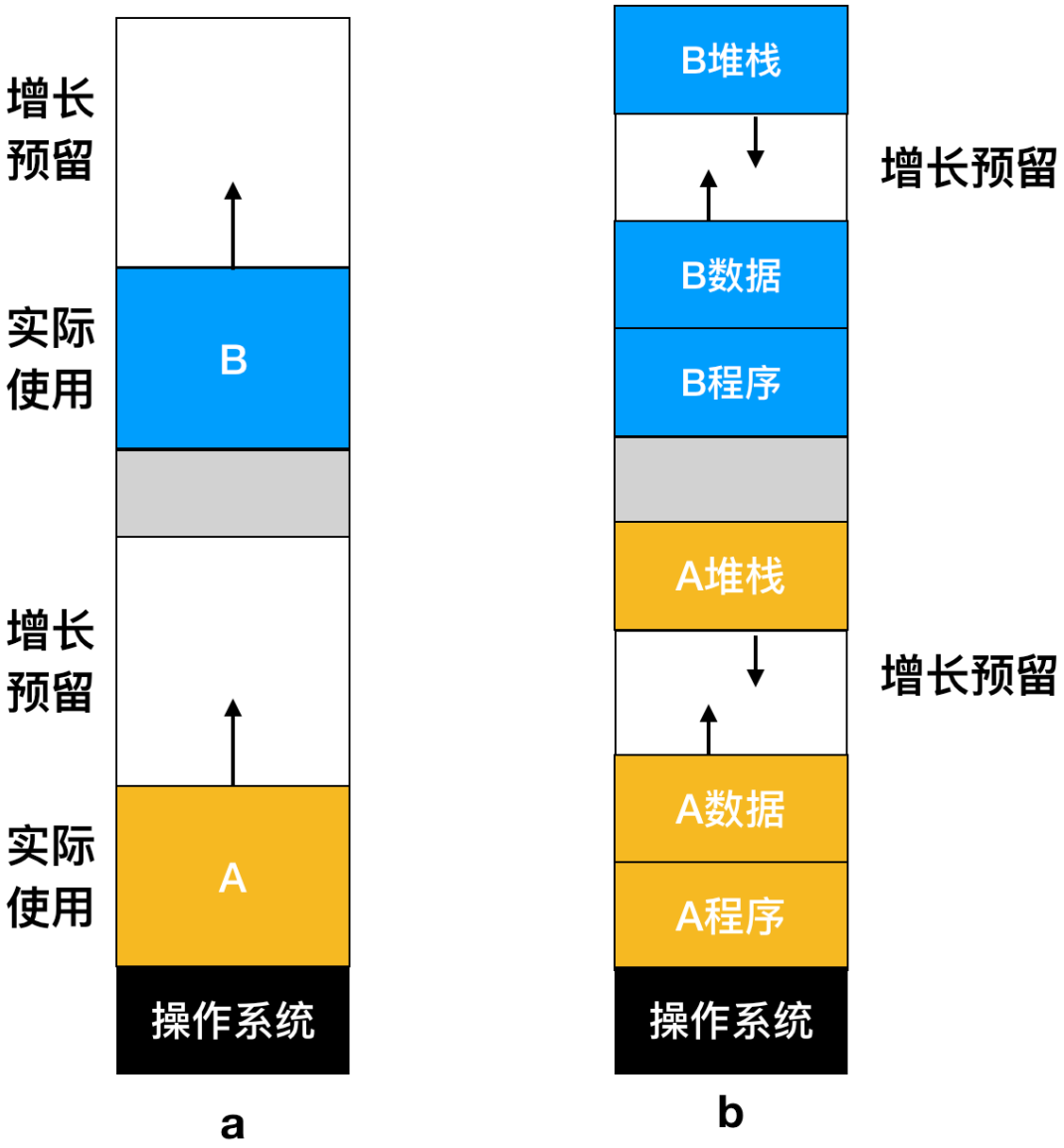
所以内存针对自动增长的区域，会有三种处理方式

- 如果一个进程与空闲区相邻，那么可把该空闲区分配给进程以供其增大。
- 如果进程相邻的是另一个进程，就会有两种处理方式：要么把需要增长的进程移动到一个内存中空闲区足够大的区域，要么把一个或多个进程交换出去，已变成生成一个大的空闲区。
- 如果一个进程在内存中不能增长，而且磁盘上的交换区也满了，那么这个进程只有挂起一些空闲空间（或者可以结束该进程）

内存针对 data segment 自动增长的三种处理方式



上面只针对单个或者一小部分需要增长的进程采用的方式，如果大部分进程都要在运行时增长，为了减少因内存区域不够而引起的进程交换和移动所产生的开销，一种可用的方法是，在换入或移动进程时为它分配一些额外的内存。然而，当进程被换出到磁盘上时，应该只交换实际上使用的内存，将额外的内存交换也是一种浪费，下面是一种为两个进程分配了增长空间的内存配置。



如果进程有两个可增长的段，例如，供变量动态分配和释放的作为 `堆(全局变量)` 使用的一个 `数据段(data segment)`，以及存放局部变量与返回地址的一个 `堆栈段(stack segment)`，就如图 b 所示。在图中可以看到所示进程的堆栈段在进程所占内存的顶端向下增长，紧接着在程序段后的数据段向上增长。当增长预留的内存区域不够了，处理方式就如上面的 `流程图(data segment 自动增长的三种处理方式)` 一样了。

空闲内存管理

在进行内存动态分配时，操作系统必须对其进行管理。大致上说，有两种监控内存使用的方式

- `位图(bitmap)`
- `空闲列表(free lists)`

下面我们就来探讨一下这两种使用方式

使用位图的存储管理

使用位图方法时，内存可能被划分为小到几个字或大到几千字节的分配单元。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）。一块内存区域和其对应的位图如下

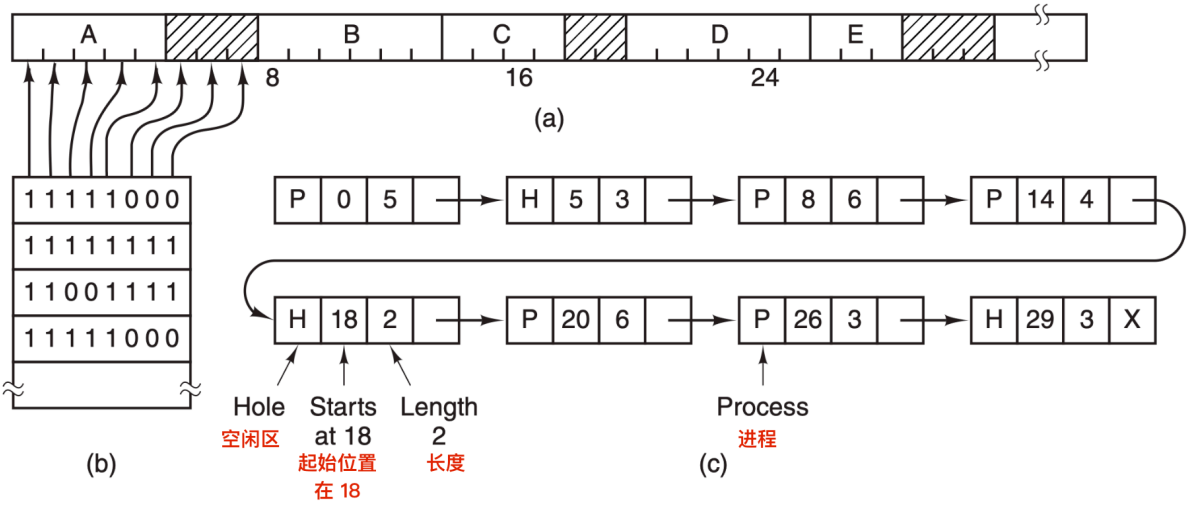


图 a 表示一段有 5 个进程和 3 个空闲区的内存，刻度为内存分配单元，阴影区表示空闲（在位图中用 0 表示）；图 b 表示对应的位图；图 c 表示用链表表示同样的信息

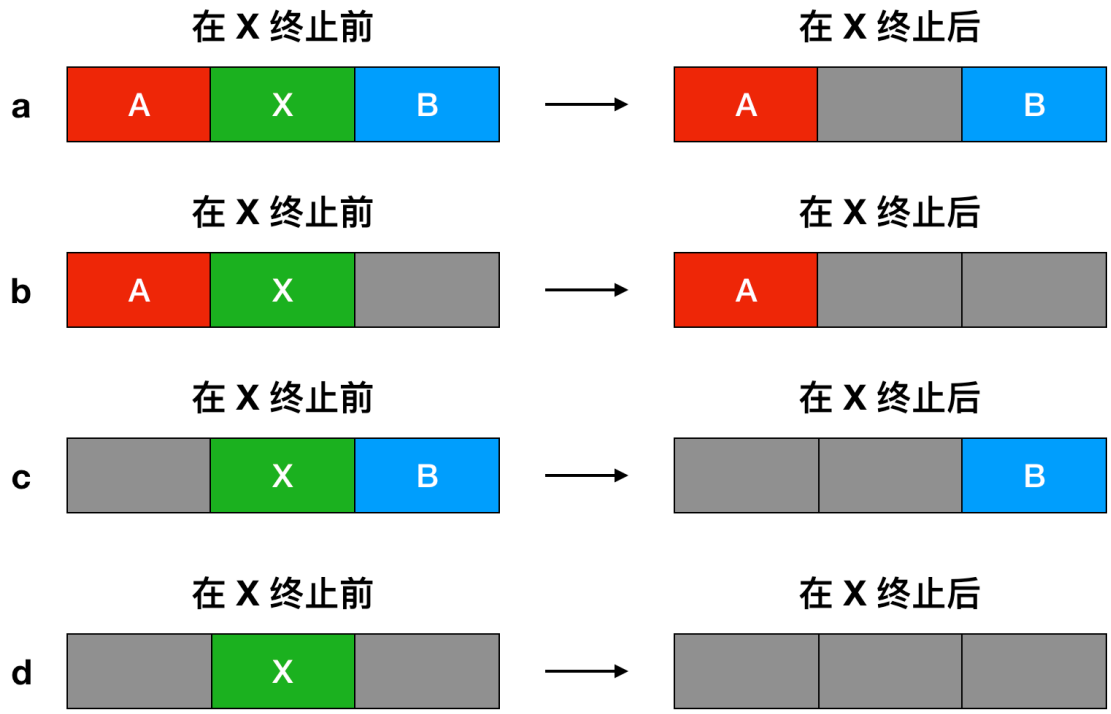
分配单元的大小是一个重要的设计因素，分配单位越小，位图越大。然而，即使只有 4 字节的分配单元，32 位的内存也仅仅只需要位图中的 1 位。`32n` 位的内存需要 n 位的位图，所以**1 个位图只占用了 1/32 的内存**。如果选择更大的内存单元，位图应该要更小。如果进程的大小不是分配单元的整数倍，那么在最后一个分配单元中会有大量的内存被浪费。

`位图` 提供了一种简单的方法在固定大小的内存中跟踪内存的使用情况，因为**位图的大小取决于内存和分配单元的大小**。这种方法有一个问题是，当决定为把具有 k 个分配单元的进程放入内存时，`内容管理器(memory manager)` 必须搜索位图，在位图中找出能够运行 k 个连续 0 位的串。在位图中找出制定长度的连续 0 串是一个很耗时的操作，这是位图的缺点。（可以简单理解为在杂乱无章的数组中，找出具有一大长串空闲的数组单元）

使用链表进行管理

另一种记录内存使用情况的方法是，维护一个记录已分配内存段和空闲内存段的链表，段会包含进程或者是两个进程的空闲区域。可用上面的图 c **来表示内存的使用情况**。链表中的每一项都可以代表一个 `空闲区(H)` 或者是 `进程(P)` 的起始标志，长度和下一个链表项的位置。

在这个例子中，`段链表(segment list)` 是按照地址排序的。这种方式的优点是，当进程终止或被交换时，更新列表很简单。一个终止进程通常有两个邻居（除了内存的顶部和底部外）。相邻的可能是进程也可能是空闲区，它们有四种组合方式。

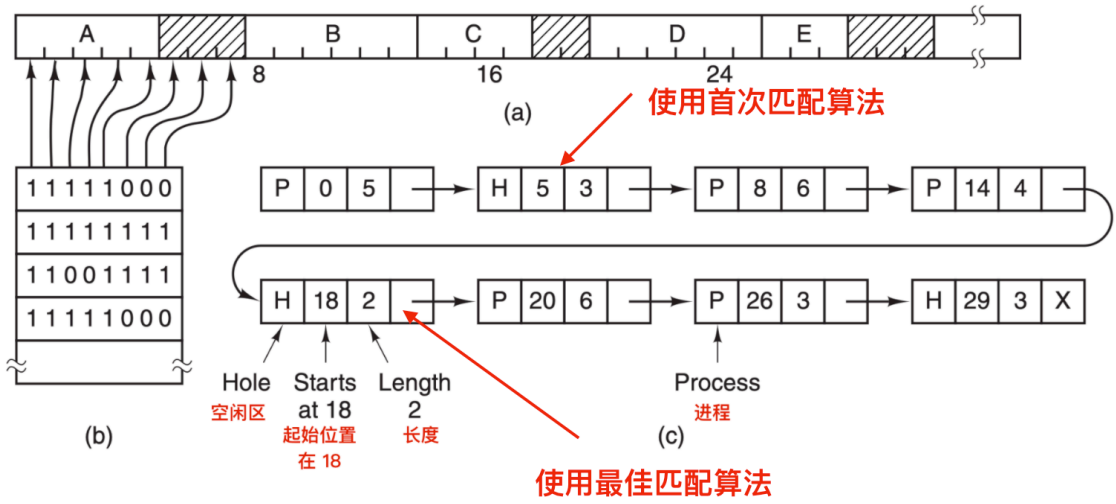


当按照地址顺序在链表中存放进程和空闲区时，有几种算法可以为创建的进程（或者从磁盘中换入的进程）分配内存。我们先假设内存管理器知道应该分配多少内存，最简单的算法是使用 `首次适配(first fit)` 。内存管理器会沿着段列表进行扫描，直到找个一个足够大的空闲区为止。除非空闲区大小和要分配的空间大小一样，否则将空闲区分为两部分，一部分供进程使用；一部分生成新的空闲区。首次适配算法是一种速度很快的算法，因为它会尽可能的搜索链表。

首次适配的一个小的变体是 `下次适配(next fit)` 。它和首次匹配的工作方式相同，只有一个不同之处那就是下次适配在每次找到合适的空闲区时就会记录当时的位置，以便下次寻找空闲区时从上次结束的地方开始搜索，而不是像首次匹配算法那样每次都会从头开始搜索。 `Bays(1997)` 证明了下次算法的性能略低于首次匹配算法。

另外一个著名的并且广泛使用的算法是 `最佳适配(best fit)` 。最佳适配会从头到尾寻找整个链表，找出能够容纳进程的最小空闲区。最佳适配算法会试图找出最接近实际需要的空闲区，以最好的匹配请求和可用空闲区，而不是先一次拆分一个以后可能会用到的大的空闲区。比如现在我们需要一个大小为 2 的块，那么首次匹配算法会把这个块分配在位置 5 的空闲区，而最佳适配算法会把该块分配在位置为 18 的空闲区，如下

使用链表如何分配一个大小为 2 的块?



那么最佳适配算法的性能如何呢?最佳适配会遍历整个链表,所以最佳适配算法的性能要比首次匹配算法差。但是令人想不到的是,最佳适配算法要比首次匹配和下次匹配算法浪费更多的内存,因为它会产生大量无用的小缓冲区,首次匹配算法生成的空闲区会更大一些。

最佳适配的空闲区会分裂出很多非常小的缓冲区,为了避免这一问题,可以考虑使用 `最差适配(worst fit)` 算法。即总是分配最大的内存区域(所以你现在明白为什么最佳适配算法会分裂出很多小缓冲区了吧),使新分配的空闲区比较大从而可以继续使用。仿真程序表明最差适配算法也不是一个好主意。

如果为进程和空闲区维护各自独立的链表,那么这四个算法的速度都能得到提高。这样,这四种算法的目标都是为了检查空闲区而不是进程。但这种分配速度的提高的一个不可避免的代价是增加复杂度和减慢内存释放速度,因为必须将一个回收的段从进程链表中删除并插入空闲链表区。

如果进程和空闲区使用不同的链表,那么可以按照大小对空闲区链表排序,以便提高最佳适配算法的速度。在使用最佳适配算法搜索由小到大排列的空闲区链表时,只要找到一个合适的空闲区,则这个空闲区就是能容纳这个作业的最小空闲区,因此是最佳匹配。因为空闲区链表以单链表形式组织,所以不需要进一步搜索。空闲区链表按大小排序时,首次适配算法与最佳适配算法一样快,而下次适配算法在这里毫无意义。

另一种分配算法是 `快速适配(quick fit)` 算法,它为那些常用大小的空闲区维护单独的链表。例如,有一个 n 项的表,该表的第一项是指向大小为 4 KB 的空闲区链表表头指针,第二项是指向大小为 8 KB 的空闲区链表表头指针,第三项是指向大小为 12 KB 的空闲区链表表头指针,以此类推。比如 21 KB 这样的空闲区既可以放在 20 KB 的链表中,也可以放在一个专门存放大小比较特别的空闲区链表中。

快速匹配算法寻找一个指定代销的空闲区也是十分快速的,但它和所有将空闲区按大小排序的方案一样,都有一个共同的缺点,即在一个进程终止或被换出时,寻找它的相邻块并查看是否可以合并的过程都是非常耗时的。如果不进行合并,内存将会很快分裂出大量进程无法利用的小空闲区。

虚拟内存

尽管基址寄存器和变址寄存器用来创建地址空间的抽象，但是这有一个其他的问题需要解决：`管理软件的不断增大(managing bloatware)`。虽然内存的大小增长迅速，但是软件的大小增长的要比内存还要快。在 1980 年的时候，许多大学用一台 4 MB 的 VAX 计算机运行分时操作系统，供十几个用户同时运行。现在微软公司推荐的 64 位 Windows 8 系统至少需要 2 GB 内存，而许多多媒体的潮流则进一步推动了对内存的需求。

这一发展的结果是，需要运行的程序往往大到内存无法容纳，而且必然需要系统能够支持多个程序同时运行，即使内存可以满足其中单独一个程序的需求，但是从总体上来看内存仍然满足不了日益增长的软件的需求（感觉和xxx和xxx 的矛盾很相似）。而交换技术并不是一个很有效的方案，在一些中小应用程序尚可使用交换，如果应用程序过大，难道还要每次交换几 GB 的内存？这显然是不合适的，一个典型的 `SATA` 磁盘的峰值传输速度高达几百兆/秒，这意味着需要好几秒才能换出或者换入一个 1 GB 的程序。

> SATA（Serial ATA）硬盘，又称串口硬盘，是未来 PC 机硬盘的趋势，已基本取代了传统的 PATA 硬盘。

那么还有没有一种有效的方式来应对呢？有，那就是使用 `虚拟内存(virtual memory)`，虚拟内存的基本思想是，每个程序都有自己的地址空间，这个地址空间被划分为多个称为 `页面(page)` 的块。每一页都是连续的地址范围。这些页被映射到物理内存，但并不是所有的页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间时，硬件会立刻执行必要的映射。当程序引用到一部分不在物理内存中的地址空间时，由操作系统负责将缺失的部分装入物理内存并重新执行失败的指令。

在某种意义上来说，虚拟地址是对基址寄存器和变址寄存器的一种概述。8088 有分离的基址寄存器（但不是变址寄存器）用于放入 text 和 data 。

使用虚拟内存，可以将整个地址空间以很小的单位映射到物理内存中，而不是仅仅针对 text 和 data 区进行重定位。下面我们会探讨虚拟内存是如何实现的。

虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中，当一个程序等待它的一部分读入内存时，可以把 CPU 交给另一个进程使用。

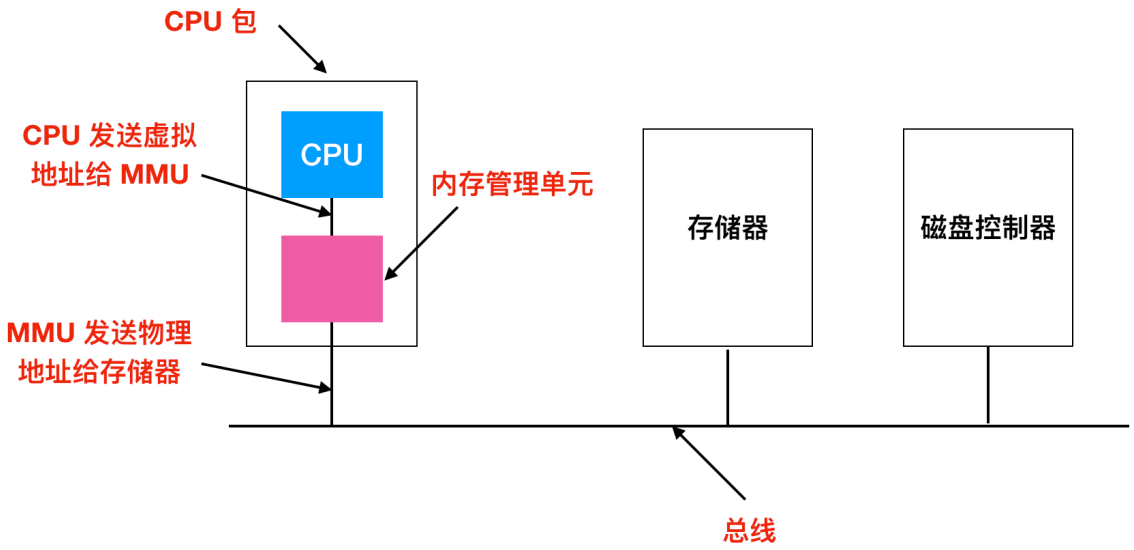
分页

大部分使用虚拟内存的系统中都会使用一种 `分页(paging)` 技术。在任何一台计算机上，程序会引用使用一组内存地址。当程序执行

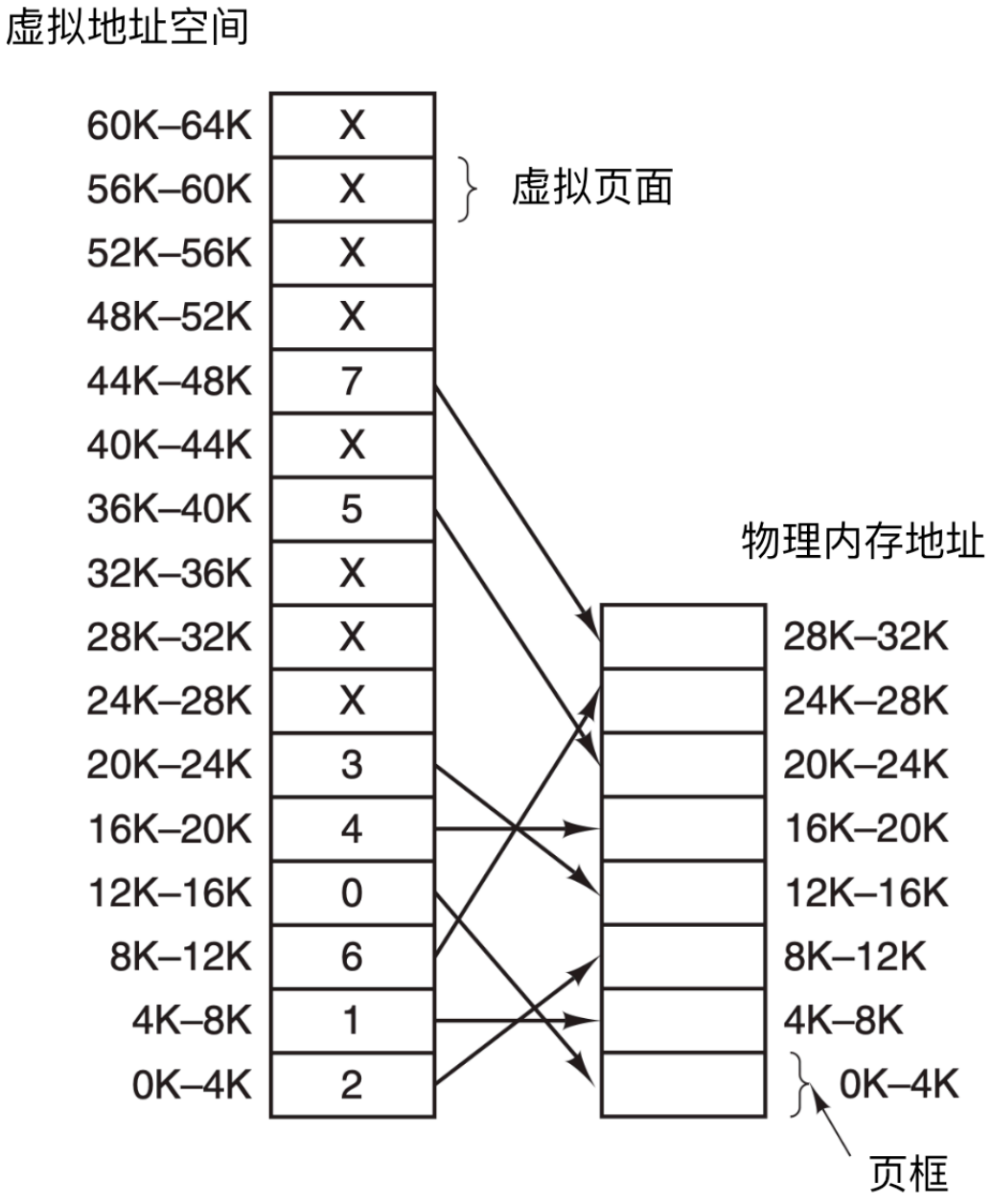
```
1    MOV REG,1000
```

这条指令时，它会把内存地址为 1000 的内存单元的内容复制到 REG 中（或者相反，这取决于计算机）。地址可以通过索引、基址寄存器、段寄存器或其他方式产生。

这些程序生成的地址被称为 `虚拟地址(virtual addresses)` 并形成 `虚拟地址空间(virtual address space)`，在没有虚拟内存的计算机上，系统直接将虚拟地址送到内存中线上，读写操作都使用同样地址的物理内存。**在使用虚拟内存时，虚拟地址不会直接发送到内存总线上**。相反，会使用 `MMU(Memory Management Unit)` 内存管理单元把**虚拟地址映射为物理内存地址**，像下图这样



下面这幅图展示了这种映射是如何工作的



页表给出虚拟地址与物理内存地址之间的映射关系。每一页起始于 4096 的倍数位置，结束于 4095 的位置，所以 4K 到 8K 实际为 4096 - 8191 ，8K - 12K 就是 8192 - 12287

在这个例子中，我们可能有一个 16 位地址的计算机，地址从 0 - 64 K - 1，这些是 `虚拟地址` 。然而只有 32 KB 的物理地址。所以虽然可以编写 64 KB 的程序，但是程序无法全部调入内存运行，在磁盘上必须有一个最多 64 KB 的程序核心映像的完整副本，以保证程序片段在需要时被调入内存。

存在映射的页如何映射

虚拟地址空间由固定大小的单元组成，这种固定大小的单元称为 `页(pages)` 。而相对的，物理内存中也有固定大小的物理单元，称为 `页框(page frames)` 。页和页框的大小一样。在上面这个例子中，页的大小为 4KB ，但是实际的使用过程中页的大小范围可能是 512 字节 - 1G 字节的大小。对应于 64 KB 的虚拟地址空间和 32 KB 的物理内存，可得到 16 个虚拟页面和 8 个页框。RAM 和磁盘之间的交换总是以整个页为单元进行交换的。

程序试图访问地址时，例如执行下面这条指令

```
1    MOV REG, 0
```

会将虚拟地址 0 送到 MMU。MMU 看到虚拟地址落在页面 0 （0 - 4095），根据其映射结果，这一页面对应的页框 2 （8192 - 12287），因此 MMU 把地址变换为 8192 ，并把地址 8192 送到总线上。内存对 MMU 一无所知，它只看到一个对 8192 地址的读写请求并执行它。MMU 从而有效的把所有虚拟地址 0 - 4095 映射到了 8192 - 12287 的物理地址。同样的，指令

```
1    MOV REG, 8192
```

也被有效的转换为

```
1    MOV REG, 24576
```

虚拟地址 8192（在虚拟页 2 中）被映射到物理地址 24576（在物理页框 6 中）上。

通过恰当的设置 MMU，可以把 16 个虚拟页面映射到 8 个页框中的任何一个。但是这并没有解决虚拟地址空间比物理内存大的问题。

上图中有 8 个物理页框，于是只有 8 个虚拟页被映射到了物理内存中，在上图中用 `X` 号表示的其他页面没有被映射。在实际的硬件中，会使用一个 `在/不在(Present/absent bit)` 位记录页面在内存中的实际存在情况。

未映射的页如何映射

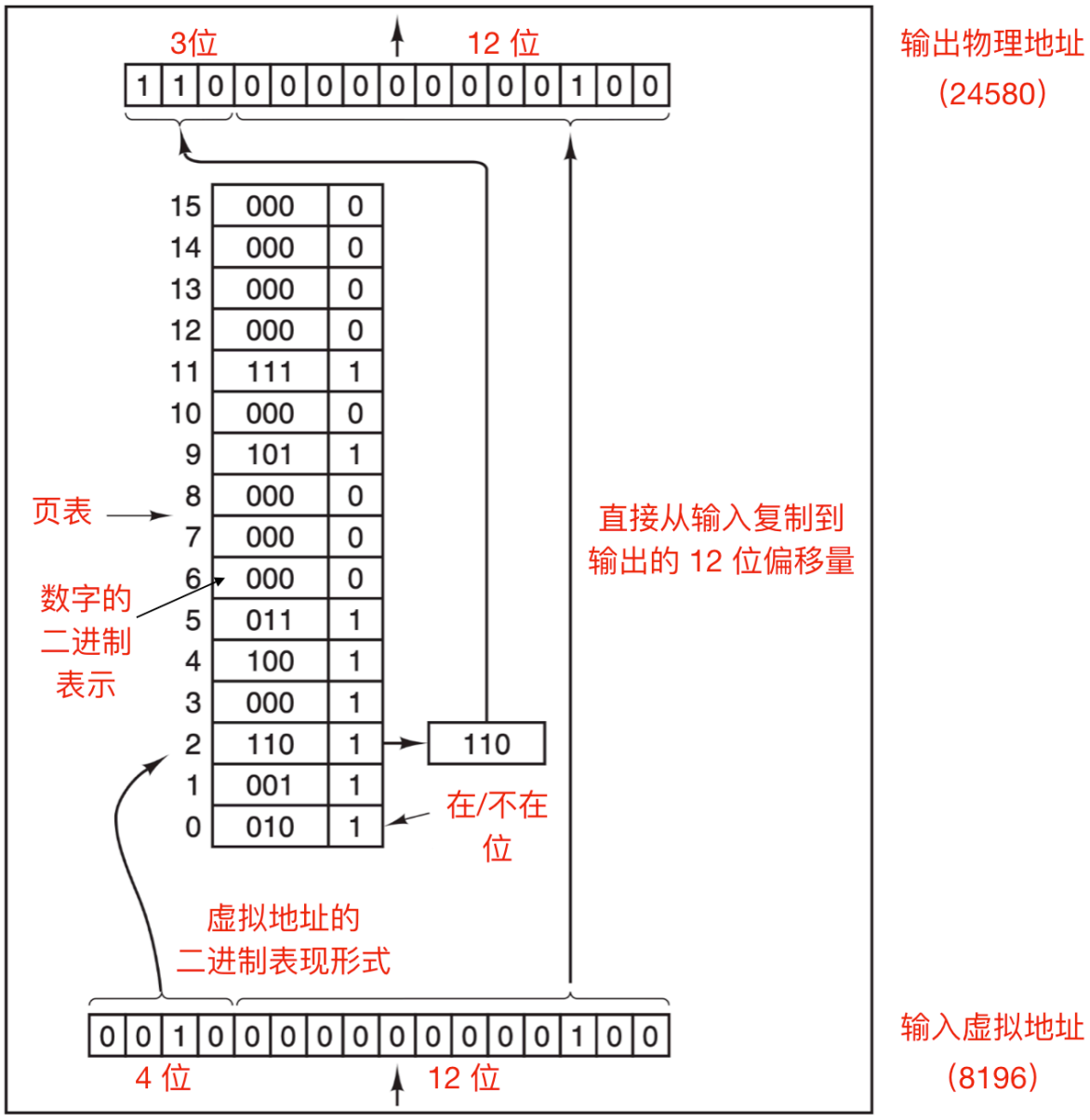
当程序访问一个未映射的页面，如执行指令

```
1    MOV REG, 32780
```

将会发生什么情况呢？虚拟页面 8 （从 32768 开始）的第 12 个字节所对应的物理地址是什么？MMU 注意到该页面没有被映射（在图中用 X 号表示），于是 CPU 会 `陷入(trap)` 到操作系统中。这个陷入称为 `缺页中断(page fault)` 或者是 `缺页错误` 。操作系统会选择一个很少使用的页并把它的内容写入磁盘（如果它不在磁盘上）。随后把需要访问的页面读到刚才回收的页框中，修改映射关系，然后重新启动引起陷入的指令。有点不太好理解，举个例子来看一下。

例如，如果操作系统决定放弃页框 1，那么它将把虚拟机页面 8 装入物理地址 4096，并对 MMU 映射做两处修改。首先，它要将虚拟页中的 1 表项标记为未映射，使以后任何对虚拟地址 4096 - 8191 的访问都将导致陷入。随后把虚拟页面 8 的表项的叉号改为 1，因此在引起陷阱的指令重新启动时，它将把虚拟地址 32780 映射为物理地址（4096 + 12）。

下面查看一下 MMU 的内部构造以便了解它们是如何工作的，以及了解为什么我们选用的页大小都是 2 的整数次幂。下图我们可以看到一个虚拟地址的例子



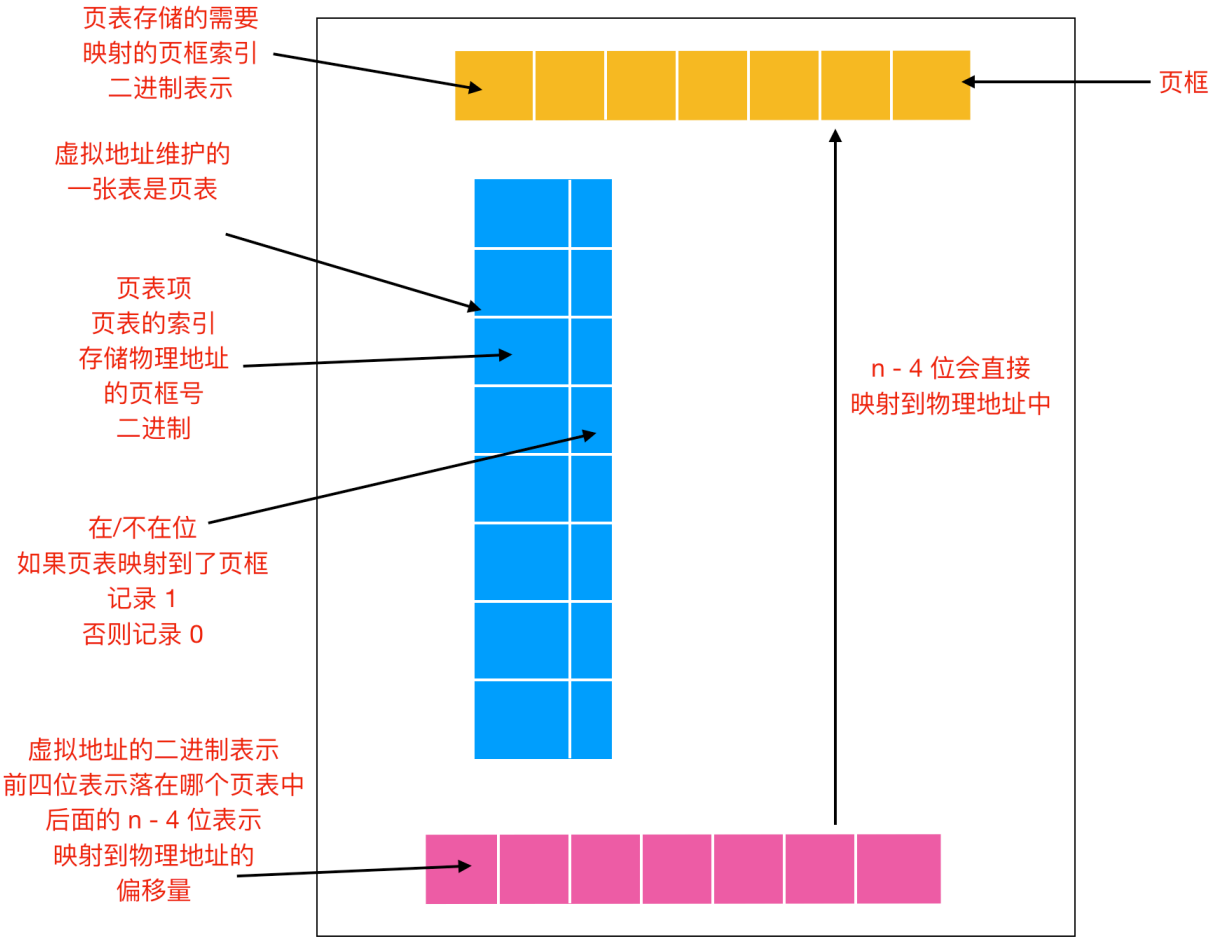
虚拟地址 8196 （二进制 0010000000000100）用上面的页表映射图所示的 MMU 映射机制进行映射，输入的 16 位虚拟地址被分为 4 位的页号和 12 位的偏移量。4 位的页号可以表示 16 个页面，12 位的偏移可以为一页内的全部 4096 个字节。

可用页号作为 `页表(page table)` 的索引，以得出对应于该虚拟页面的页框号。如果 `在/不在` 位则是 0 ，则引起一个操作系统陷入。如果该位是 1，则将在页表中查到的页框号复制到输出寄存器的高 3 位中，再加上输入虚拟地址中的低 12 位偏移量。如此就构成了 15 位的物理地址。输出寄存器的内容随即被作为物理地址送到总线。

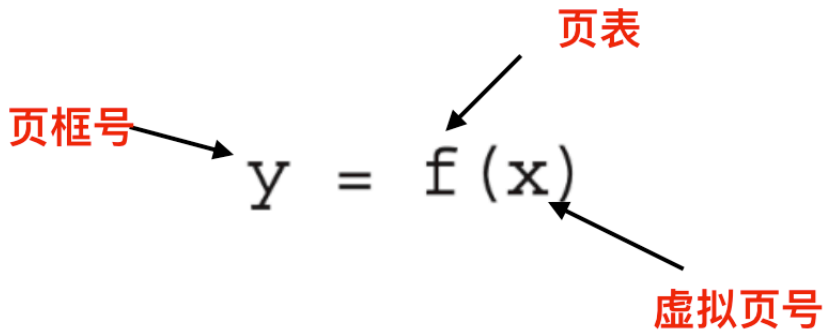
页表

在上面这个简单的例子中，虚拟地址到物理地址的映射可以总结如下：虚拟地址被分为 虚拟页号（高位部分） 和 偏移量（低位部分） 。例如，对于 16 位地址和 4 KB 的页面大小，高 4 位可以指定 16 个虚拟页面中的一页，而低 12 位接着确定了所选页面中的偏移量（0-4095）。

虚拟页号可作为页表的索引用来找到虚拟页中的内容。由页表项可以找到页框号（如果有的话）。然后把页框号拼接到偏移量的高位端，以替换掉虚拟页号，形成物理地址。



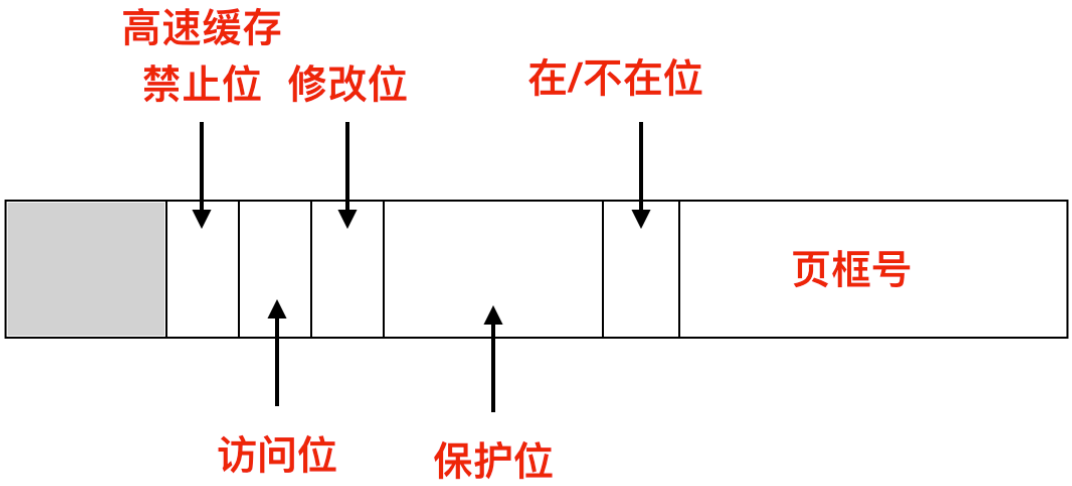
因此，页表的目的是把虚拟页映射到页框中。从数学上说，页表是一个函数，它的参数是虚拟页号，结果是物理页框号。



通过这个函数可以把虚拟地址中的虚拟页转换为页框，从而形成物理地址。

页表项的结构

下面我们探讨一下页表项的具体结构，上面你知道了页表项的大致构成，是由页框号和在/不在位构成的，现在我们来具体探讨一下页表项的构成



页表项的结构是与机器相关的，但是不同机器上的页表项大致相同。上面是一个页表项的构成，不同计算机的页表项可能不同，但是一般来说都是 32 位的。页表项中最重要的字段就是 `页框号(Page frame number)` 。毕竟，页表到页框最重要的一步操作就是要把此值映射过去。下一个比较重要的就是 `在/不在` 位，如果此位上的值是 1，那么页表项是有效的并且能够被 `使用` 。如果此值是 0 的话，则表示该页表项对应的虚拟页面 `不在` 内存中，访问该页面会引起一个 `缺页异常(page fault)` 。

`保护位(Protection)` 告诉我们哪一种访问是允许的，啥意思呢？最简单的表示形式是这个域只有一位，**0 表示可读可写，1 表示的是只读**。

`修改位(Modified)` 和 `访问位(Referenced)` 会跟踪页面的使用情况。当一个页面被写入时，硬件会自动的设置修改位。修改位在页面重新分配页框时很有用。如果一个页面已经被修改过（即它是 `脏` 的），则必须把它写回磁盘。如果一个页面没有被修改过（即它是 `干净` 的），那么重新分配时这个页框会被直接丢弃，因为磁盘上的副本仍然是有效的。这个位有时也叫做 `脏位(dirty bit)` ，因为它反映了页面的状态。

`访问位(Referenced)` 在页面被访问时被设置，不管是读还是写。这个值能够帮助操作系统在发生缺页中断时选择要淘汰的页。不再使用的页要比正在使用的页更适合被淘汰。这个位在后面要讨论的 `页面置换` 算法中作用很大。

最后一位用于禁止该页面被高速缓存，这个功能对于映射到设备寄存器还是内存中起到了关键作用。通过这一位可以禁用高速缓存。具有独立的 I/O 空间而不是用内存映射 I/O 的机器来说，并不需要这一位。

在深入讨论下面问题之前，需要强调一下：虚拟内存本质上是用来创造一个地址空间的抽象，可以把它理解成为进程是对 CPU 的抽象，虚拟内存的实现，本质是将虚拟地址空间分解成页，并将每一项映射到物理内存的某个页框。因为我们的重点是如何管理这个虚拟内存的抽象。

加速分页过程

到现在我们已经 `虚拟内存(virtual memory)` 和 `分页(paging)` 的基础，现在我们可以把目光放在具体的实现上面了。在任何带有分页的系统中，都会需要面临下面这两个主要问题：

- 虚拟地址到物理地址的映射速度必须要快
- 如果虚拟地址空间足够大，那么页表也会足够大

第一个问题是**由于每次访问内存都需要进行虚拟地址到物理地址的映射**，所有的指令最终都来自于内存，并且很多指令也会访问内存中的操作数。

> 操作数：操作数是计算机指令中的一个组成部分，它规定了指令中进行数字运算的量 。操作数指出指令执行的操作所需要数据的来源。操作数是汇编指令的一个字段。比如，MOV、ADD 等。

因此，每条指令可能会多次访问页表，如果执行一条指令需要 1 ns，那么页表查询需要在 0.2 ns 之内完成，以避免映射成为一个主要性能瓶颈。

第二个问题是所有的现代操作系统都会使用至少 32 位的虚拟地址，并且 64 位正在变得越来越普遍。假设页大小为 4 KB，32 位的地址空间将近有 100 万页，而 64 位地址空间简直多到无法想象。

对大而且快速的页映射的需要成为构建计算机的一个非常重要的约束。就像上面页表中的图一样，**每一个表项对应一个虚拟页面，虚拟页号作为索引**。在启动一个进程时，操作系统会把保存在内存中进程页表读副本放入寄存器中。

> 最后一句话是不是不好理解？还记得页表是什么吗？它是虚拟地址到内存地址的映射页表。页表是虚拟地址转换的关键组成部分，它是访问内存中数据所必需的。在进程启动时，执行很多次虚拟地址到物理地址的转换，会把物理地址的副本从内存中读入到寄存器中，再执行这一转换过程。

所以，在进程的运行过程中，不必再为页表而访问内存。使用这种方法的优势是 `简单而且映射过程中不需要访问内存` 。缺点是 `页表太大时，代价高昂` ，而且每次上下文切换的时候都必须 `装载整个页表` ，这样会造成性能的降低。鉴于此，我们讨论一下加速分页机制和处理大的虚拟地址空间的实现方案

转换检测缓冲区

我们首先先来一起探讨一下加速分页的问题。大部分优化方案都是从内存中的页表开始的。这种设计对效率有着巨大的影响。考虑一下，例如，假设一条 1 字节的指令要把一个寄存器中的数据复制到另一个寄存器。在不分页的情况下，这条指令只访问一次内存，即从内存取出指令。有了分页机制后，会因为要访问页表而需要更多的内存访问。由于执行速度通常被 CPU 从内存中取指令和数据的速度所限制，这样的话，两次访问才能实现一次的访问效果，所以内存访问的性能会下降一半。在这种情况下，根本不会采用分页机制。

> 什么是 1 字节的指令？我们以 8085 微处理器为例来说明一下，在 8085 微处理中，一共有 3 种字节指令，它们分别是 `1-byte(1 字节)` 、 `2-byte(2 字节)` 、 `3-byte(3 字节)` ，我们分别来说一下
>
> 1-byte：1 字节的操作数和操作码共同以 1 字节表示；操作数是内部寄存器，并被编码到指令中；指令需要一个存储位置来将单个寄存器存储在存储位置中。没有操作数的指令也是 1-byte 指令。
>
> 例如：MOV B,C 、LDAX B、NOP、HLT（这块不明白的读者可以自行查阅）
>
> 2-byte: 2 字节包括：第一个字节指定的操作码；第二个字节指定操作数；指令需要两个存储器位置才能存储在存储器中。
>
> 例如 MVI B, 26 H、IN 56 H
>
> 3-byte: 在 3 字节指令中，第一个字节指定操作码；后面两个字节指定 16 位的地址；第二个字节保存 `低位` 地址；第三个字节保存 `高位` 地址。指令需要三个存储器位置才能将单个字节存储在存储器中。
>
> 例如 LDA 2050 H、JMP 2085 H

大多数程序总是对少量页面进行多次访问，而不是对大量页面进行少量访问。因此，只有很少的页面能够被再次访问，而其他的页表项很少被访问。

> 页表项一般也被称为 `Page Table Entry(PTE)` 。

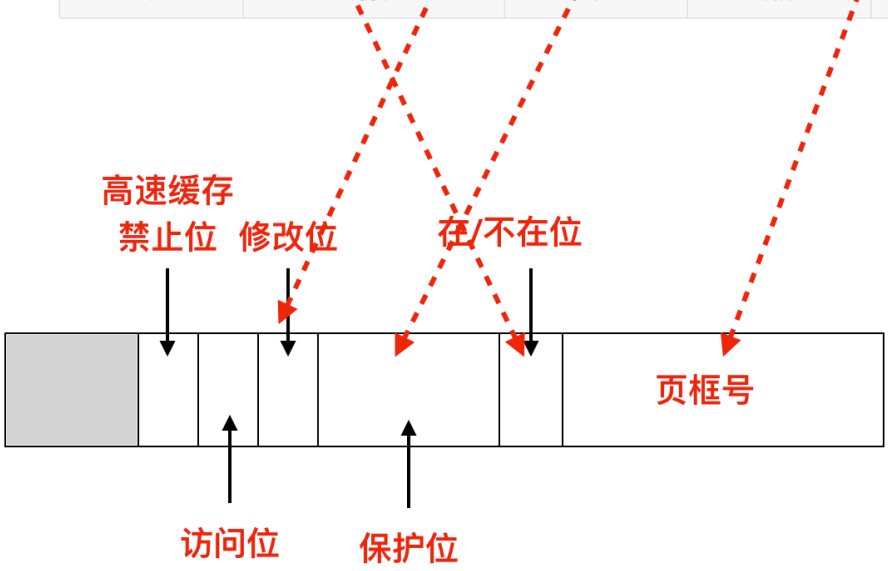
基于这种设想，提出了一种方案，即从硬件方面来解决这个问题，为计算机设置一个小型的硬件设备，能够将虚拟地址直接映射到物理地址，而不必再访问页表。这种设备被称为 `转换检测缓冲区(Translation Lookaside Buffer, TLB)` ，有时又被称为 `相联存储器(associate memory)` 。

有效位	虚拟页面号	修改位	保护位	页框号
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB 加速分页

TLB 通常位于 MMU 中，包含少量的表项，每个表项都记录了页面的相关信息，除了虚拟页号外，其他表项都和页表是一一对应的

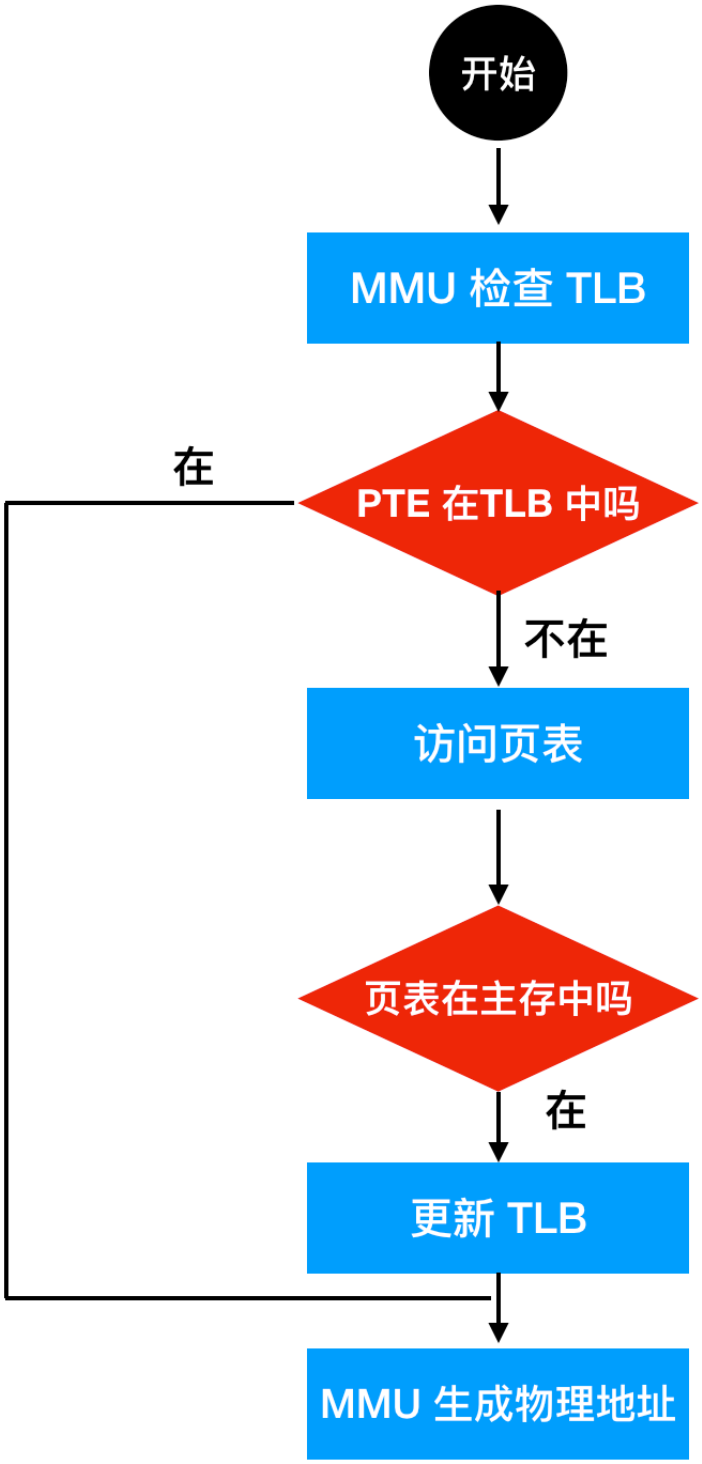
有效位	虚拟页面号	修改位	保护位	页框号
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75



是不是你到现在还是有点不理解什么是 TLB，TLB 其实就是一种内存缓存，用于减少访问内存所需要的时间，它就是 MMU 的一部分，TLB 会将虚拟地址到物理地址的转换存储起来，通常可以称为 `地址翻译缓存(address-translation cache)` 。TLB 通常位于 CPU 和 CPU 缓存之间，它与 CPU 缓存是不同的缓存级别。下面我们来看一下 TLB 是如何工作的。

当一个 MMU 中的虚拟地址需要进行转换时，硬件首先检查虚拟页号与 TLB 中所有表项进行并行匹配，判断虚拟页是否在 TLB 中。如果找到了有效匹配项，并且要进行的访问操作没有违反保护位的话，则将页框号直接从 TLB 中取出而不用再直接访问页表。如果虚拟页在 TLB 中但是违反了保护位的权限的话（比如只允许读但是是一个写指令），则会生成一个 `保护错误(protection fault)` 返回。

上面探讨的是虚拟地址在 TLB 中的情况，那么如果虚拟地址不再 TLB 中该怎么办？如果 MMU 检测到没有有效的匹配项，就会进行正常的页表查找，然后从 TLB 中逐出一个表项然后把从页表中找到的项放在 TLB 中。当一个表项被从 TLB 中清除出，将修改位复制到内存中页表项，除了访问位之外，其他位保持不变。当页表项从页表装入 TLB 中时，所有的值都来自于内存。



MMU 检查 TLB 过程

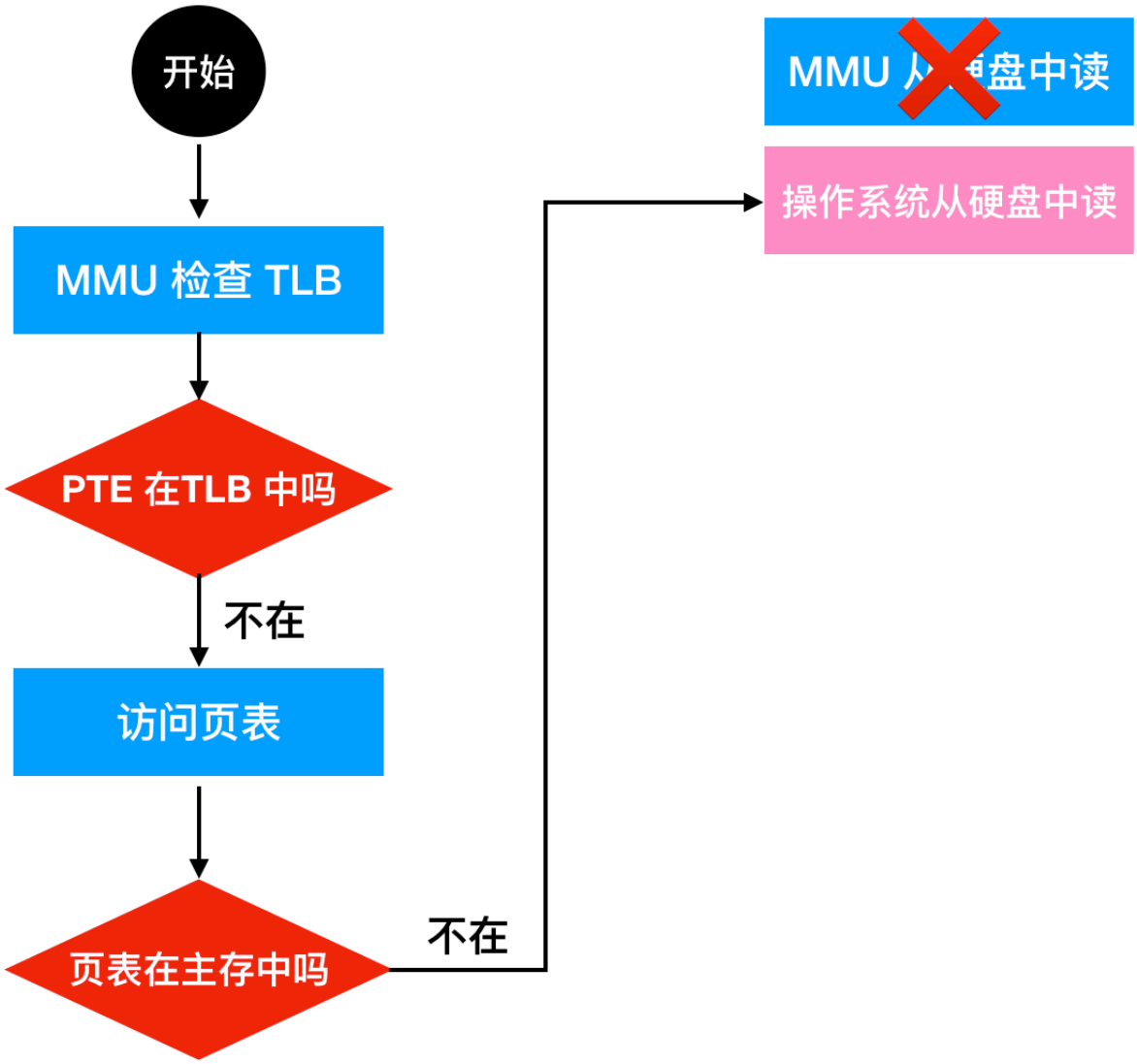
软件 TLB 管理

直到现在，我们假设每台电脑都有可以被硬件识别的页表，外加一个 TLB。在这个设计中，TLB 管理和处理 TLB 错误完全由硬件来完成。仅仅当页面不在内存中时，才会发生操作系统的 `陷入(trap)` 。

在以前，我们上面的假设通常是正确的。但是，许多现代的 `RISC` 机器，包括 SPARC、MIPS 和 HP PA，几乎所有的页面管理都是在软件中完成的。

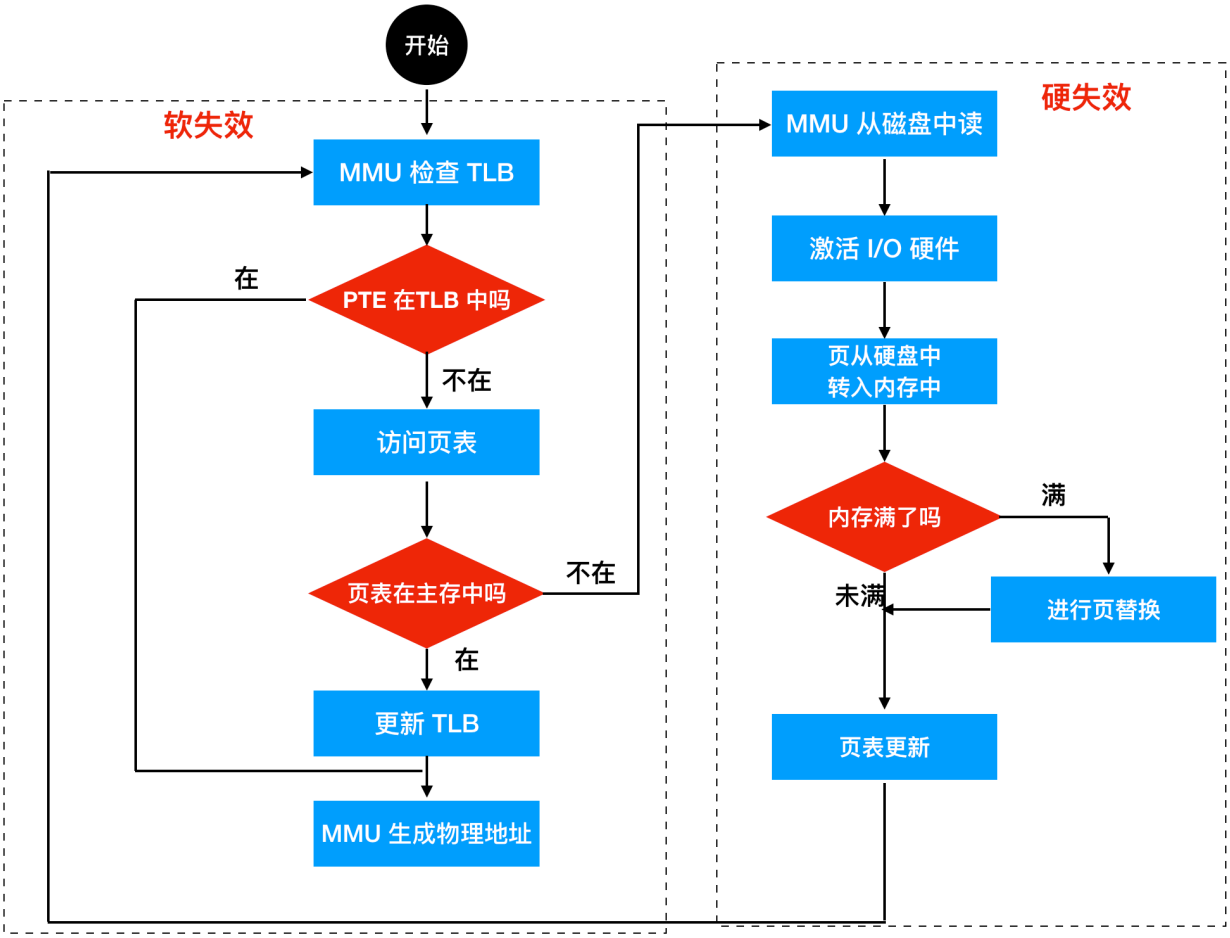
> 精简指令集计算机或 RISC 是一种计算机指令集，它使计算机的微处理器的每条指令（CPI）周期比复杂指令集计算机（CISC）少

在这些计算机上，TLB 条目由操作系统显示加载。当发生 TLB 访问丢失时，**不再是由 MMU 到页表中查找并取出需要的页表项，而是生成一个 TLB 失效并将问题交给操作系统解决**。操作系统必须找到该页，把它从 TLB 中移除（移除页表中的一项），然后把新找到的页放在 TLB 中，最后再执行先前出错的指令。然而，所有这些操作都必须通过少量指令完成，因为 TLB 丢失的发生率要比出错率高很多。



无论是用硬件还是用软件来处理 TLB 失效，常见的方式都是找到页表并执行索引操作以定位到将要访问的页面，在软件中进行搜索的问题是保存页表的页可能不在 TLB 中，这将在处理过程中导致其他 TLB 错误。改善方法是可以在内存中的固定位置维护一个大的 TLB 表项的高速缓存来减少 TLB 失效。通过首先检查软件的高速缓存，`操作系统` 能够有效的减少 TLB 失效问题。

TLB 软件管理会有两种 TLB 失效问题，当一个页访问在内存中而不在 TLB 中时，将产生 `软失效(soft miss)`，那么此时要做的就是把页表更新到 TLB 中（我们上面探讨的过程），而不会产生磁盘 I/O，处理仅仅需要一些机器指令在几纳秒的时间内完成。然而，当页本身不在内存中时，将会产生 `硬失效(hard miss)`，那么此时就需要从磁盘中进行页表提取，硬失效的处理时间通常是软失效的百万倍。在页表结构中查找映射的过程称为 `页表遍历(page table walk)`。



上面的这两种情况都是理想情况下出现的现象，但是在实际应用过程中情况会更加复杂，未命中的情况可能既不是硬失效又不是软失效。一些未命中可能更 `软` 或更 `硬` （偷笑）。比如，如果页表遍历的过程中没有找到所需要的页，那么此时会出现三种情况：

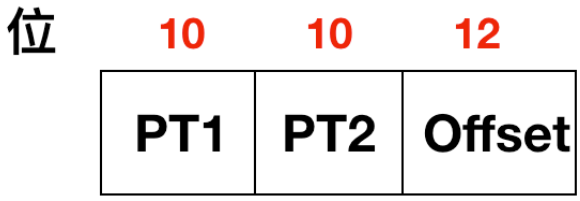
- 所需的页面就在内存中，但是却没有记录在进程的页表中，这种情况可能是由其他进程从磁盘掉入内存，这种情况只需要把页正确映射就可以了，而不需要在从硬盘调入，这是一种软失效，称为 `次要缺页错误(minor page fault)` 。
- 基于上述情况，如果需要从硬盘直接调入页面，这就是 `严重缺页错误(major page falut)` 。
- 还有一种情况是，程序可能访问了一个非法地址，根本无需向 TLB 中增加映射。此时，操作系统会报告一个 `段错误(segmentation fault)` 来终止程序。只有第三种缺页属于程序错误，其他缺页情况都会被硬件或操作系统以降低程序性能为代价来修复

针对大内存的页表

还记得我们讨论的是什么问题吗？（捂脸），可能讨论的太多你有所不知道了，我再提醒你一下，上面 `加速分页` 过程讨论的是**虚拟地址到物理地址的映射速度必须要快**的问题，还有一个问题是 **如果虚拟地址空间足够大，那么页表也会足够大**的问题，如何处理巨大的虚拟地址空间，下面展开我们的讨论。

多级页表

第一种方案是使用 `多级页表(multi)` ，下面是一个例子

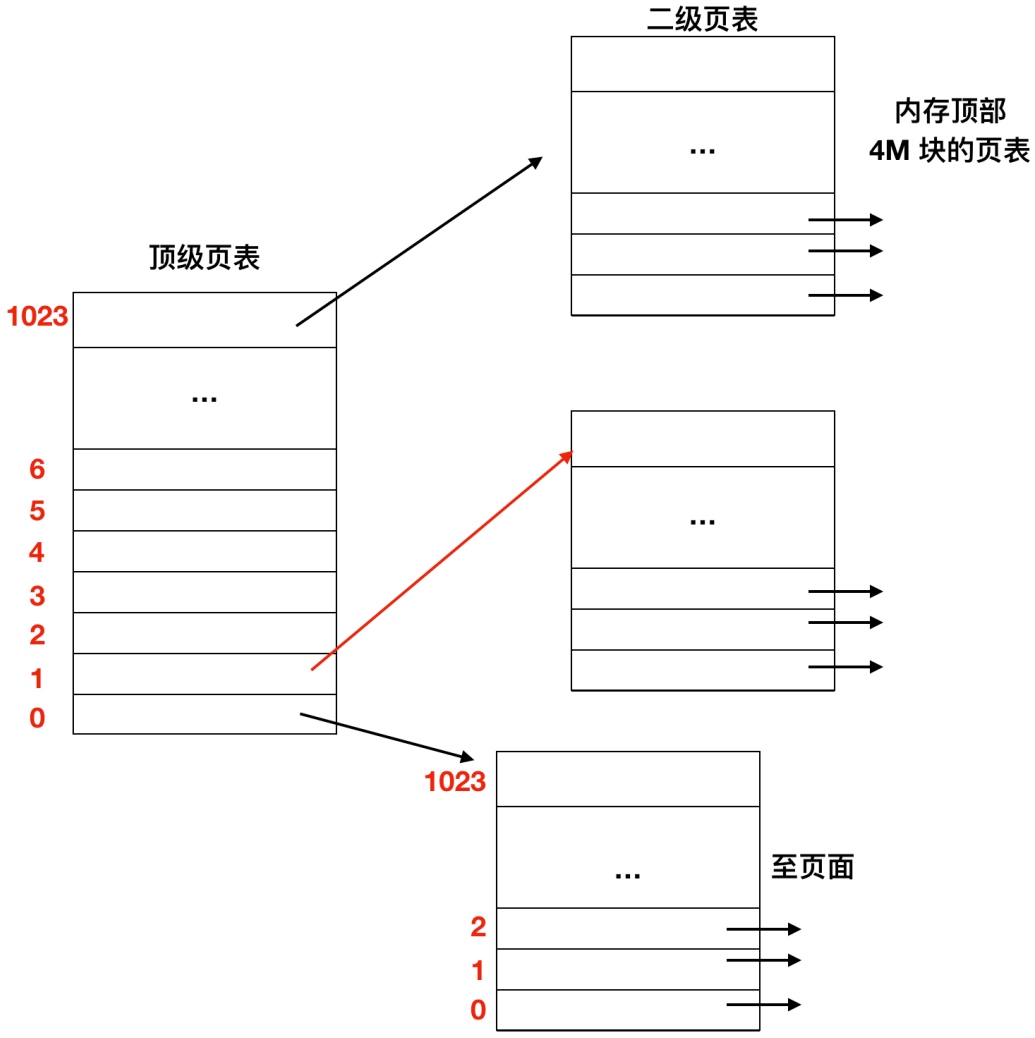


32 位的虚拟地址被划分为 10 位的 PT1 域，10 位的 PT2 域，还有 12 位的 Offset 域。因为偏移量是 12 位，所以页面大小是 4KB，公有 2^20 次方个页面。

引入多级页表的原因是避免把全部页表一直保存在内存中。不需要的页表就不应该保留。

多级页表是一种分页方案，它由两个或多个层次的分页表组成，也称为分层分页。级别1（level 1）页面表的条目是指向级别 2（level 2） 页面表的指针，级别2页面表的条目是指向级别 3（level 3） 页面表的指针，依此类推。最后一级页表存储的是实际的信息。

下面是一个二级页表的工作过程



在最左边是顶级页表，它有 1024 个表项，对应于 10 位的 PT1 域。当一个虚拟地址被送到 MMU 时，MMU 首先提取 PT1 域并把该值作为访问顶级页表的索引。因为整个 4 GB （即 32 位）虚拟地址已经按 4 KB 大小分块，所以顶级页表中的 1024 个表项的每一个都表示 4M 的块地址范围。

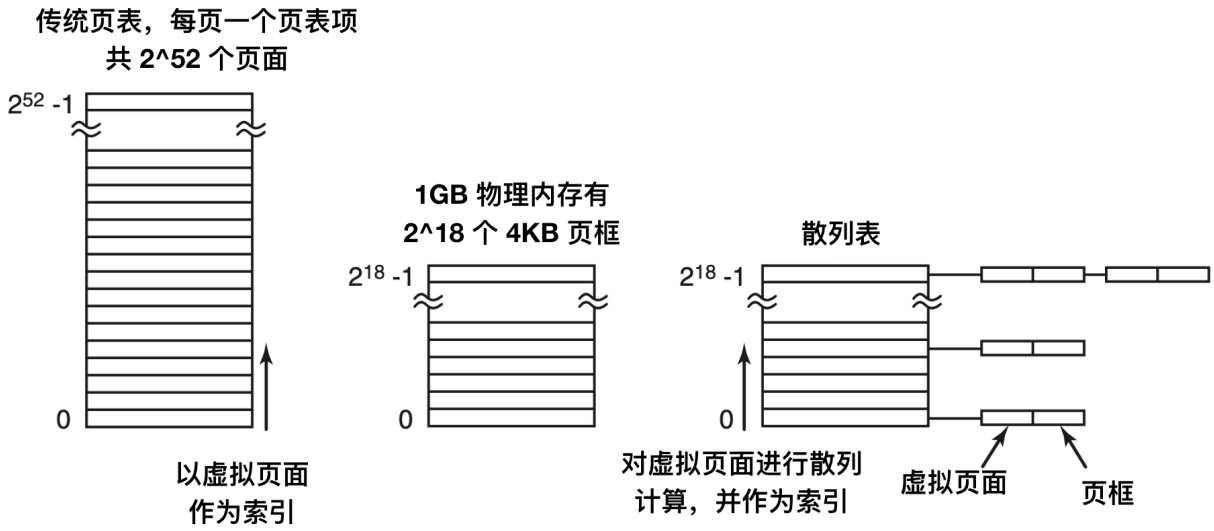
由索引顶级页表得到的表项中含有二级页表的地址或页框号。顶级页表的表项 0 指向程序正文的页表，表项 1 指向含有数据的页表，表项 1023 指向堆栈的页表，其他的 项（用阴影表示） 表示没有使用。现在把 PT2 域作为访问选定的二级页表的索引，以便找到虚拟页面的对应页框号。

倒排页表

针对分页层级结构中不断增加的替代方法是使用 倒排页表(inverted page tables) 。采用这种解决方案的有 PowerPC、UltraSPARC 和 Itanium。在这种设计中，实际内存中的每个页框对应一个表项，而不是每个虚拟页面对应一个表项。

虽然倒排页表节省了大量的空间，但是它也有自己的缺陷：那就是从虚拟地址到物理地址的转换会变得很困难。当进程 n 访问虚拟页面 p 时，硬件不能再通过把 p 当作指向页表的一个索引来查找物理页。而是必须搜索整个倒排表来查找某个表项。另外，搜索必须对每一个内存访问操作都执行一次，而不是在发生缺页中断时执行。

解决这一问题的方式时使用 TLB。当发生 TLB 失效时，需要用软件搜索整个倒排页表。一个可行的方式是建立一个散列表，用虚拟地址来散列。当前所有内存中的具有相同散列值的虚拟页面被链接在一起。如下图所示



如果散列表中的槽数与机器中物理页面数一样多，那么散列表的冲突链的长度将会是 1 个表项的长度，这将会大大提高映射速度。一旦页框被找到，新的（虚拟页号，物理页框号）就会被装在到 TLB 中。

页面置换算法

当发生缺页异常时，操作系统会选择一个页面进行换出从而为新进来的页面腾出空间。如果要换出的页面在内存中 已经被修改 ，那么必须将其写到磁盘中以使磁盘副本 保持最新状态 。如果页面没有被修改过，并且磁盘中的副本也已经是最新的，那么就 不需要 进行重写。那么就直接使用调入的页面覆盖需要移除的页面就可以了。

当发生缺页中断时，虽然可以随机的选择一个页面进行置换，但是如果每次都选择一个不常用的页面会提升系统的性能。如果一个经常使用的页面被换出，那么这个页面在短时间内又可能被重复使用，那么就可能会造成额外的性能开销。在关于页面的主题上有很多 页面置换算法(page replacement algorithms) ，这些已经从理论上和实践上得到了证明。

需要指出的是，页面置换问题在计算机的其他领域中也会出现。例如，多数计算机把最近使用过的 32 字节或者 64 字节的存储块保存在一个或多个高速缓存中。当缓存满的时候，一些块就被选择和移除。这些块的移除除了花费时间较短外，这个问题同页面置换问题完全一样。之所以花费时间较短，是因为丢掉的高速缓存可以从内存中获取，而内存没有寻找磁道的时间也不存在旋转延迟。

第二个例子是 Web 服务器。服务器会在内存中缓存一些经常使用到的 Web 页面。然而，当缓存满了并且已经引用了新的页面，那么必须决定退出哪个 Web 页面。在高速缓存中的 Web 页面不会被修改。因此磁盘中的 Web 页面经常是最新的，同样的考虑也适用在虚拟内存中。在虚拟系统中，内存中的页面可能会修改也可能不会修改。

下面我们就来探讨一下有哪些页面置换算法。

最优页面置换算法

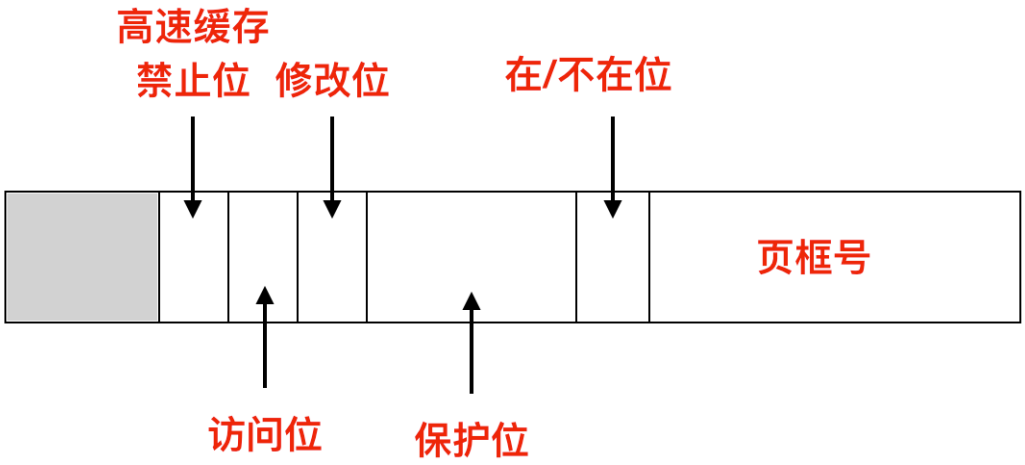
最优的页面置换算法很容易描述但在实际情况下很难实现。它的工作流程如下：在缺页中断发生时，这些页面之一将在下一条指令（包含该指令的页面）上被引用。其他页面则可能要到 10、100 或者 1000 条指令后才会被访问。每个页面都可以用在该页首次被访问前所要执行的指令数作为标记。

最优化的页面算法表明应该标记最大的页面。如果一个页面在 800 万条指令内不会被使用，另外一个页面在 600 万条指令内不会被使用，则置换前一个页面，从而把需要调入这个页面而发生的缺页中断推迟。计算机也像人类一样，会把不愿意做的事情尽可能的往后拖。

这个算法最大的问题时无法实现。当缺页中断发生时，操作系统无法知道各个页面的下一次将在什么时候被访问。这种算法在实际过程中根本不会使用。

最近未使用页面置换算法

为了能够让操作系统收集页面使用信息，大部分使用虚拟地址的计算机都有两个状态位，R 和 M，来和每个页面进行关联。**每当引用页面（读入或写入）时都设置 R，写入（即修改）页面时设置 M**，这些位包含在每个页表项中，就像下面所示



因为每次访问时都会更新这些位，因此由硬件来设置它们非常重要。一旦某个位被设置为 1，就会一直保持 1 直到操作系统下次来修改此位。

如果硬件没有这些位，那么可以使用操作系统的缺页中断和时钟中断机制来进行模拟。当启动一个进程时，将其所有的页面都标记为不在内存；一旦访问任何一个页面就会引发一次缺页中断，此时操作系统就可以设置 R 位(在它的内部表中)，修改页表项使其指向正确的页面，并设置为 READ ONLY 模式，然后重新启动引起缺页中断的指令。如果页面随后被修改，就会发生另一个缺页异常。从而允许

操作系统设置 M 位并把页面的模式设置为 `READ/WRITE` 。

可以用 R 位和 M 位来构造一个简单的页面置换算法：当启动一个进程时，操作系统将其所有页面的两个位都设置为 0。R 位定期的被清零（在每个时钟中断）。用来将最近未引用的页面和已引用的页面分开。

当出现缺页中断后，操作系统会检查所有的页面，并根据它们的 R 位和 M 位将当前值分为四类：

- 第 0 类：没有引用 R，没有修改 M
- 第 1 类：没有引用 R，已修改 M
- 第 2 类：引用 R，没有修改 M
- 第 3 类：已被访问 R，已被修改 M

尽管看起来好像无法实现第一类页面，但是当第三类页面的 R 位被时钟中断清除时，它们就会发生。时钟中断不会清除 M 位，因为需要这个信息才能知道是否写回磁盘中。清除 R 但不清除 M 会导致出现一类页面。

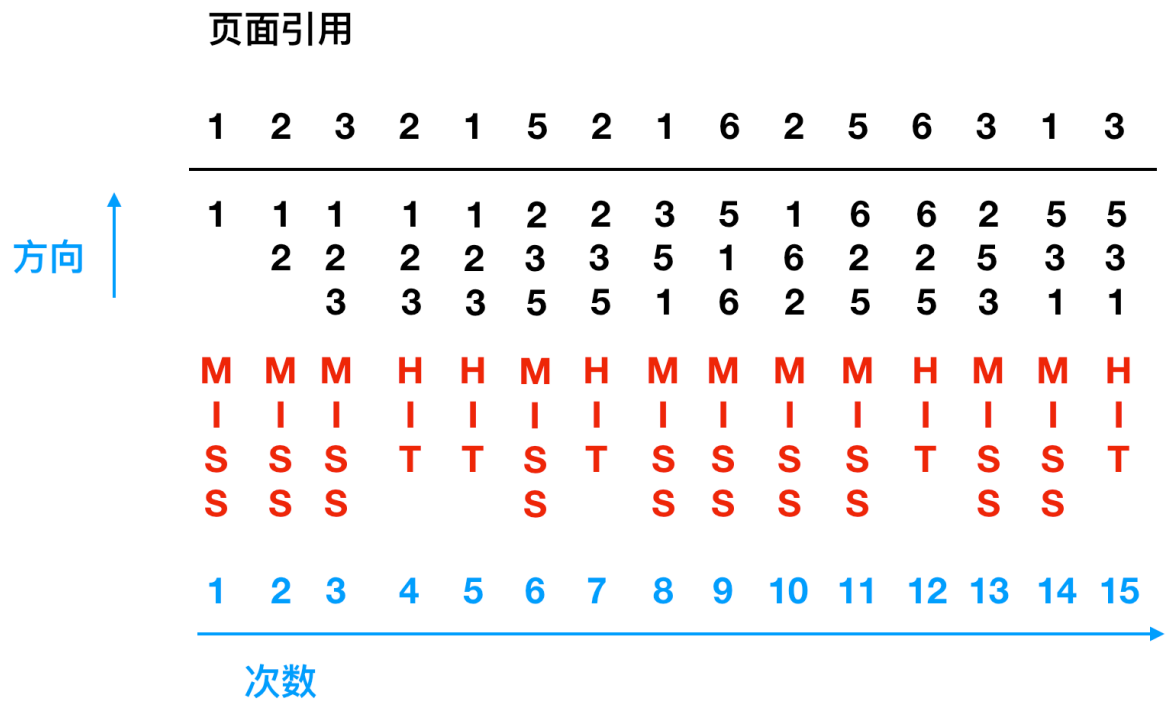
`NRU(Not Recently Used)` 算法从编号最小的非空类中随机删除一个页面。此算法隐含的思想是，在一个时钟内（约 20 ms）淘汰一个已修改但是没有被访问的页面要比一个大量引用的未修改页面好，NRU 的主要优点是**易于理解并且能够有效的实现**。

先进先出页面置换算法

另一种开销较小的方式是使用 `FIFO(First-In,First-Out)` 算法，这种类型的数据结构也适用在页面置换算法中。由操作系统维护一个所有在当前内存中的页面的链表，最早进入的放在表头，最新进入的页面放在表尾。在发生缺页异常时，会把头部的页移除并且把新的页添加到表尾。

> 还记得缺页异常什么时候发生吗？我们知道应用程序访问内存会进行虚拟地址到物理地址的映射，缺页异常就发生在虚拟地址无法映射到物理地址的时候。因为实际的物理地址要比虚拟地址小很多（参考上面的虚拟地址和物理地址映射图），所以缺页经常会发生。

先进先出页面可能是最简单的页面替换算法了。在这种算法中，操作系统会跟踪链表中内存中的所有页。下面我们举个例子看一下（这个算法我刚开始看的时候有点懵逼，后来才看懂，我还是很菜）

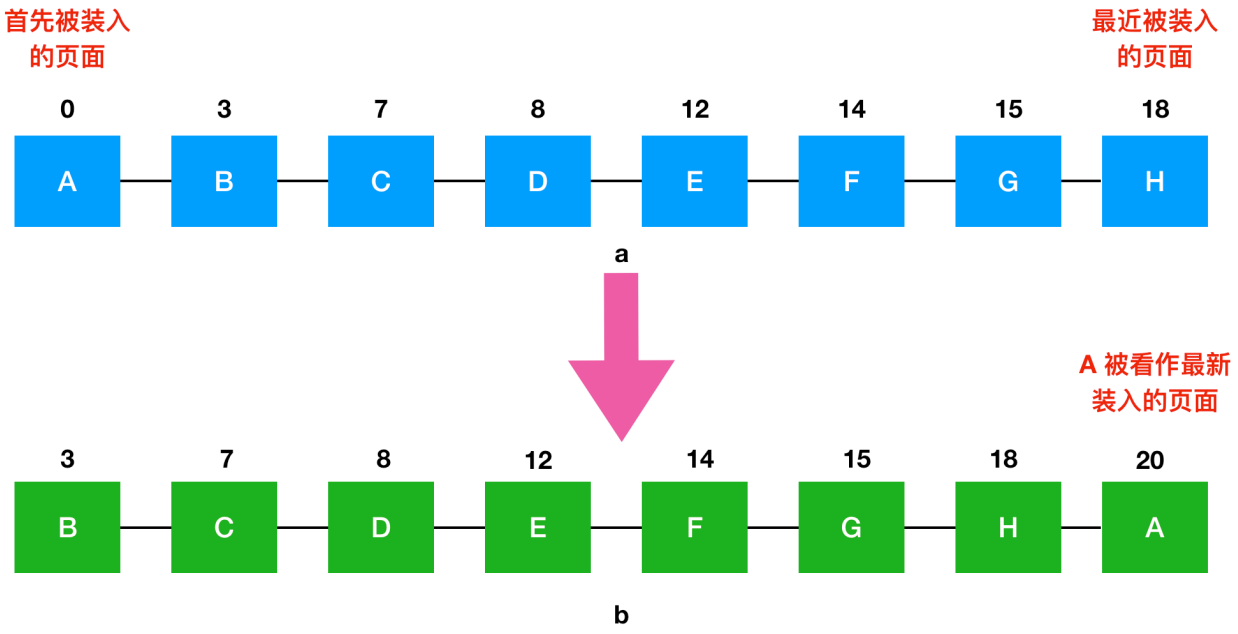


- 初始化的时候，没有任何页面，所以第一次的时候会检查页面 1 是否位于链表中，没有在链表中，那么就是 `MISS`，页面1 进入链表，链表的先进先出的方向如图所示。
- 类似的，第二次会先检查页面 2 是否位于链表中，没有在链表中，那么页面 2 进入链表，状态为 `MISS`，依次类推。
- 我们来看第四次，此时的链表为 `1 2 3`，第四次会检查页面 `2` 是否位于链表中，经过检索后，发现 2 在链表中，那么状态就是 `HIT`，并不会再进行入队和出队操作，第五次也是一样的。
- 下面来看第六次，此时的链表还是 `1 2 3`，因为之前没有执行进入链表操作，页面 `5` 会首先进行检查，发现链表中没有页面 5，则执行页面 5 的进入链表操作，页面 2 执行出链表的操作，执行完成后的链表顺序为 `2 3 5`。

第二次机会页面置换算法

我们上面学到的 FIFO 链表页面有个 `缺陷`，那就是出链和入链并不会进行 check `检查`，这样就会容易把经常使用的页面置换出去，为了避免这一问题，我们对该算法做一个简单的修改：我们检查最老页面的 `R 位`，如果是 0，那么这个页面就是最老的而且没有被使用，那么这个页面就会被立刻换出。如果 R 位是 1，那么就清除此位，此页面会被放在链表的尾部，修改它的装入时间就像刚放进来的一样。然后继续搜索。

这种算法叫做 `第二次机会(second chance)` 算法，就像下面这样，我们看到页面 A 到 H 保留在链表中，并按到达内存的时间排序。



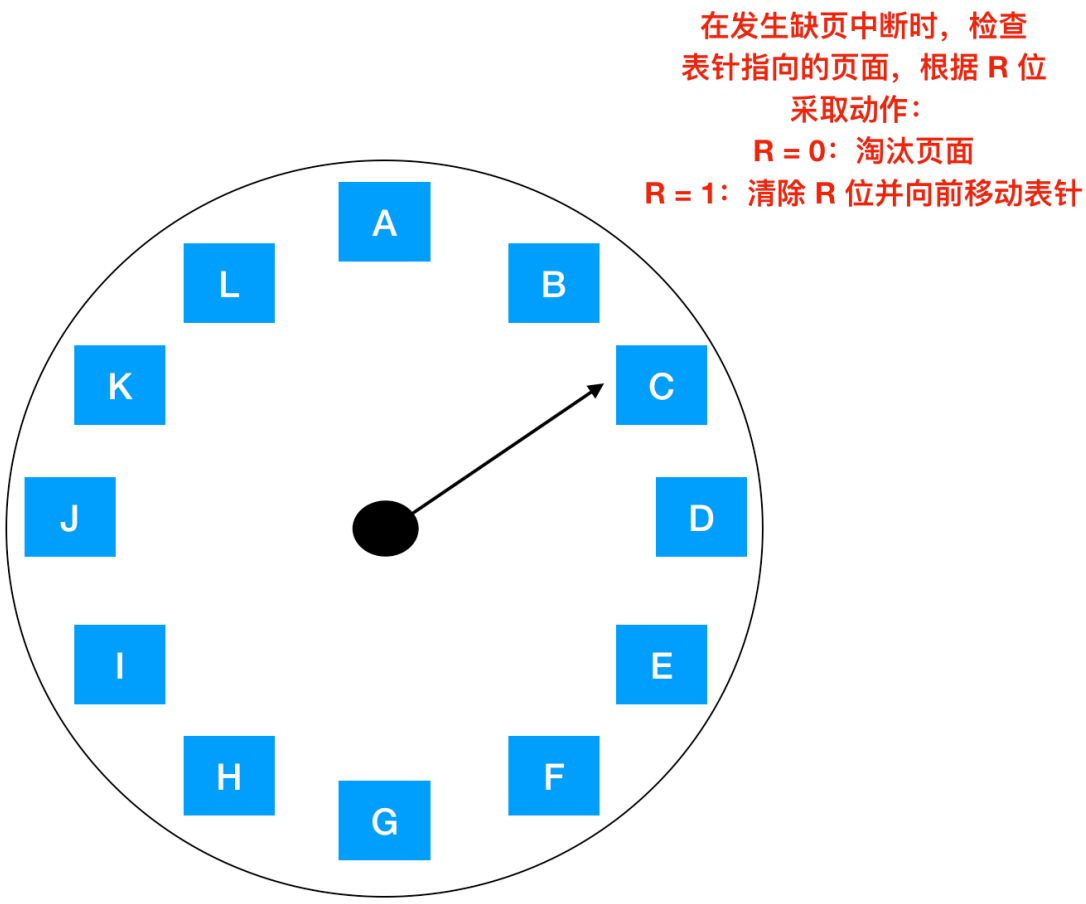
a）按照先进先出的方法排列的页面；b）在时刻 20 处发生缺页异常中断并且 A 的 R 位已经设置时的页面链表。

假设缺页异常发生在时刻 20 处，这时最老的页面是 A，它是在 0 时刻到达的。如果 A 的 R 位是 0，那么它将被淘汰出内存，或者把它写回磁盘（如果它已经被修改过），或者只是简单的放弃（如果它是未被修改过）。另一方面，如果它的 R 位已经设置了，则将 A 放到链表的尾部并且重新设置 `装入时间` 为当前时刻（20 处），然后清除 R 位。然后从 B 页面开始继续搜索合适的页面。

寻找第二次机会的是在最近的时钟间隔中未被访问过的页面。如果所有的页面都被访问过，该算法就会被简化为单纯的 `FIFO 算法` 。具体来说，假设图 a 中所有页面都设置了 R 位。操作系统将页面依次移到链表末尾，每次都在添加到末尾时清除 R 位。最后，算法又会回到页面 A，此时的 R 位已经被清除，那么页面 A 就会被执行出链处理，因此算法能够正常结束。

时钟页面置换算法

即使上面提到的第二次页面置换算法也是一种比较合理的算法，但它经常要在链表中移动页面，既降低了效率，而且这种算法也不是必须的。一种比较好的方式是把所有的页面都保存在一个类似钟面的环形链表中，一个表针指向最老的页面。如下图所示



时钟页面置换算法

当缺页错误出现时，算法首先检查表针指向的页面，如果它的 R 位是 0 就淘汰该页面，并把新的页面插入到这个位置，然后把表针向前移动一位；如果 R 位是 1 就清除 R 位并把表针前移一个位置。重复这个过程直到找到了一个 R 位为 0 的页面位置。了解这个算法的工作方式，就明白为什么它被称为 `时钟(clokc)` 算法了。

最近最少使用页面置换算法

最近最少使用页面置换算法的一个解释会是下面这样：在前面几条指令中频繁使用的页面和可能在后面的几条指令中被使用。反过来说，已经很久没有使用的页面有可能在未来一段时间内仍不会被使用。这个思想揭示了一个可以实现的算法：在缺页中断时，置换未使用时间最长的页面。这个策略称为 `LRU(Least Recently Used)` ，最近最少使用页面置换算法。

虽然 LRU 在理论上是可以实现的，但是从长远看来代价比较高。为了完全实现 LRU，会在内存中维护一个所有页面的链表，最频繁使用的页位于表头，最近最少使用的页位于表尾。困难的是在每次内存引用时更新整个链表。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常耗时的操作，即使使用 `硬件` 来实现也是一样的费时。

然而，还有其他方法可以通过硬件实现 LRU。让我们首先考虑最简单的方式。这个方法要求硬件有一个 64 位的计数器，它在每条指令执行完成后自动加 1，每个页表必须有一个足够容纳这个计数器值的域。在每次访问内存后，将当前的值保存到被访问页面的页表项中。一旦发生缺页异常，操作系统就检查所有页表项中计数器的值，找到值最小的一个页面，这个页面就是最少使用的页面。

用软件模拟 LRU

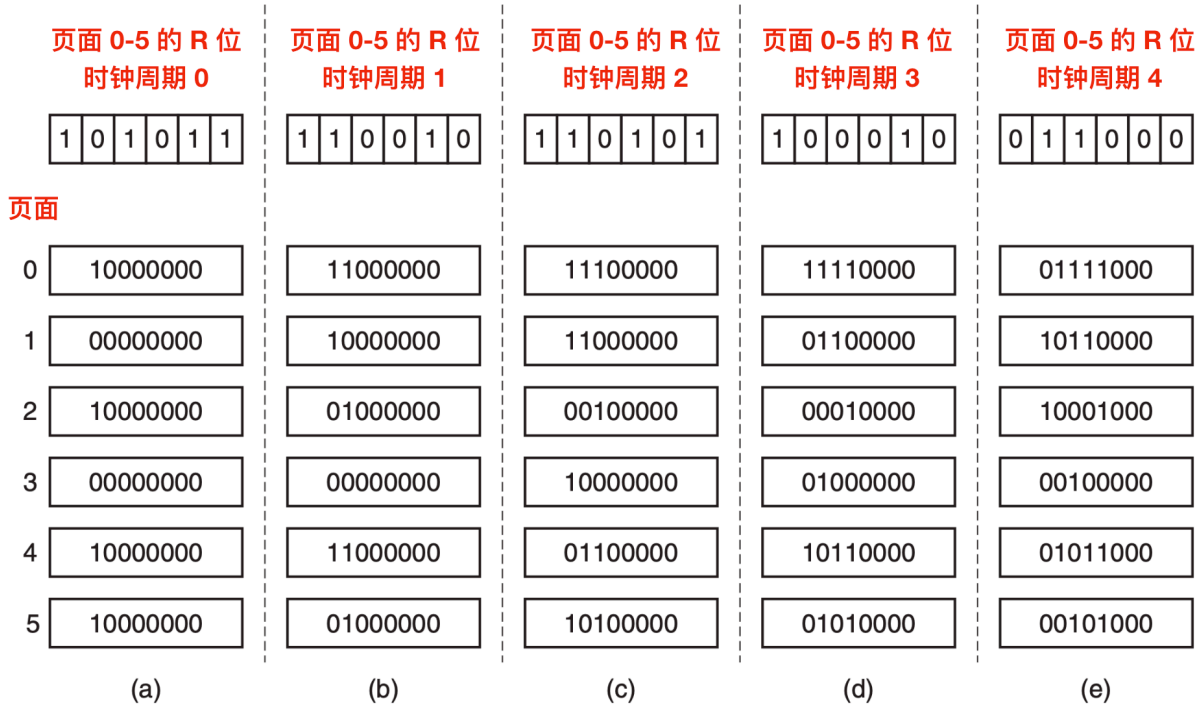
尽管上面的 LRU 算法在原则上是可以实现的，**但是很少有机器能够拥有那些特殊的硬件**。上面是硬件的实现方式，那么现在考虑要用 `软件` 来实现 LRU 。一种可以实现的方案是 `NFU(Not Frequently Used，最不常用)` 算法。它需要一个软件计数器来和每个页面关联，初始化的时候是 0 。在每个时钟中断时，操作系统会浏览内存中的所有页，会将每个页面的 R 位（0 或 1）加到它的计数器上。这个计数器大体上跟踪了各个页面访问的频繁程度。当缺页异常出现时，则置换计数器值最小的页面。

NFU 最主要的问题是它不会忘记任何东西，想一下是不是这样？例如，在一个多次（扫描）的编译器中，在第一遍扫描中频繁使用的页面会在后续的扫描中也有较高的计数。事实上，如果第一次扫描的执行时间恰好是各次扫描中最长的，那么后续遍历的页面的统计次数总会比第一次页面的统计次数 `小` 。结果是操作系统将置换有用的页面而不是不再使用的页面。

幸运的是只需要对 NFU 做一个简单的修改就可以让它模拟 LRU，这个修改有两个步骤

- 首先，在 R 位被添加进来之前先把计数器右移一位；
- 第二步，R 位被添加到最左边的位而不是最右边的位。

修改以后的算法称为 `老化(aging)` 算法，下图解释了老化算法是如何工作的。

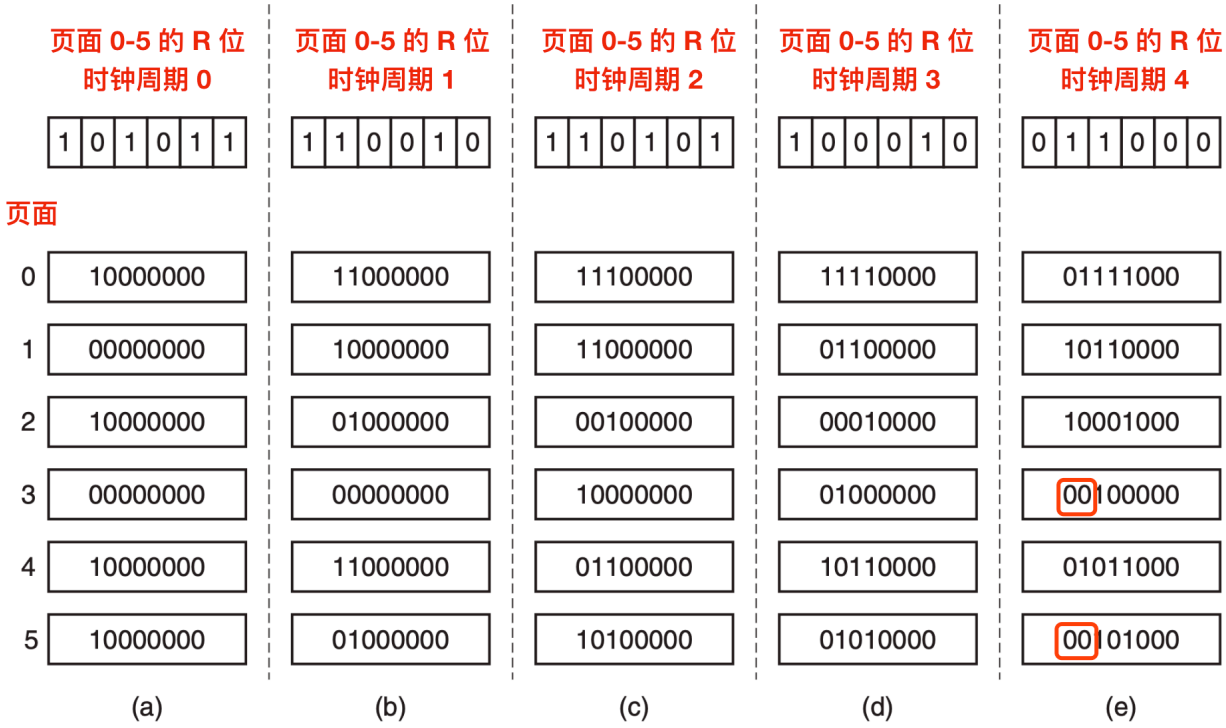


我们假设在第一个时钟周期内页面 0 - 5 的 R 位依次是 1，0，1，0，1，1，（也就是页面 0 是 1，页面 1 是 0，页面 2 是 1 这样类推）。也就是说，**在 0 个时钟周期到 1 个时钟周期之间，0，2，4，5 都被引用了**，从而把它们的 R 位设置为 1，剩下的设置为 0 。在相关的六个计数器被右移之后 R 位被添加到 `左侧` ，就像上图中的 a。剩下的四列显示了接下来的四个时钟周期内的六个计数器变化。

> CPU正在以某个频率前进，该频率的周期称为 `时钟滴答` 或 `时钟周期` 。一个 100Mhz 的处理器每秒将接收100,000,000个时钟滴答。

当缺页异常出现时，将 `置换（就是移除）` 计数器值最小的页面。如果一个页面在前面 4 个时钟周期内都没有被访问过，那么它的计数器应该会有四个连续的 0 ，因此它的值肯定要比前面 3 个时钟周期内都没有被访问过的页面的计数器小。

这个算法与 LRU 算法有两个重要的区别：看一下上图中的 `e` ，第三列和第五列



它们在两个时钟周期内都没有被访问过，在此之前的时钟周期内都引用了两个页面。根据 LRU 算法，如果需要置换的话，那么应该在这两个页面中选择一个。那么问题来了，我萌应该选择哪个？现在的问题是我们不知道时钟周期 1 到时钟周期 2 内它们中哪个页面是后被访问到的。因为在每个时钟周期内只记录了一位，所以无法区分在一个时钟周期内哪个页面最早被引用，哪个页面是最后被引用的。因此，我们能做的就是置换 `页面3`，**因为页面 3 在周期 0 - 1 内都没有被访问过，而页面 5 却被引用过**。

LRU 与老化之前的第 2 个区别是，在老化期间，计数器具有有限数量的位（这个例子中是 8 位），这就限制了以往的访问记录。如果两个页面的计数器都是 0，那么我们可以随便选择一个进行置换。实际上，有可能其中一个页面的访问次数实在 9 个时钟周期以前，而另外一个页面是在 1000 个时钟周期之前，但是我们却无法看到这些。在实际过程中，如果时钟周期是 20 ms，8 位一般是够用的。所以我们经常拿 20 ms 来举例。

工作集页面置换算法

在最单纯的分页系统中，刚启动进程时，在内存中并没有页面。此时如果 CPU 尝试匹配第一条指令，就会得到一个缺页异常，使操作系统装入含有第一条指令的页面。其他的错误比如 `全局变量` 和 `堆栈` 引起的缺页异常通常会紧接着发生。一段时间以后，进程需要的大部分页面都在内存中了，此时进程开始在较少的缺页异常环境中运行。这个策略称为 `请求调页(demand paging)`，因为页面是根据需要被调入的，而不是预先调入的。

在一个大的地址空间中系统的读所有的页面，将会造成很多缺页异常，因此会导致没有足够的内存来容纳这些页面。不过幸运的是，大部分进程不是这样工作的，它们都会以 `局部性方式(locality of reference)` 来访问，这意味着在执行的任何阶段，程序只引用其中的一小部分。

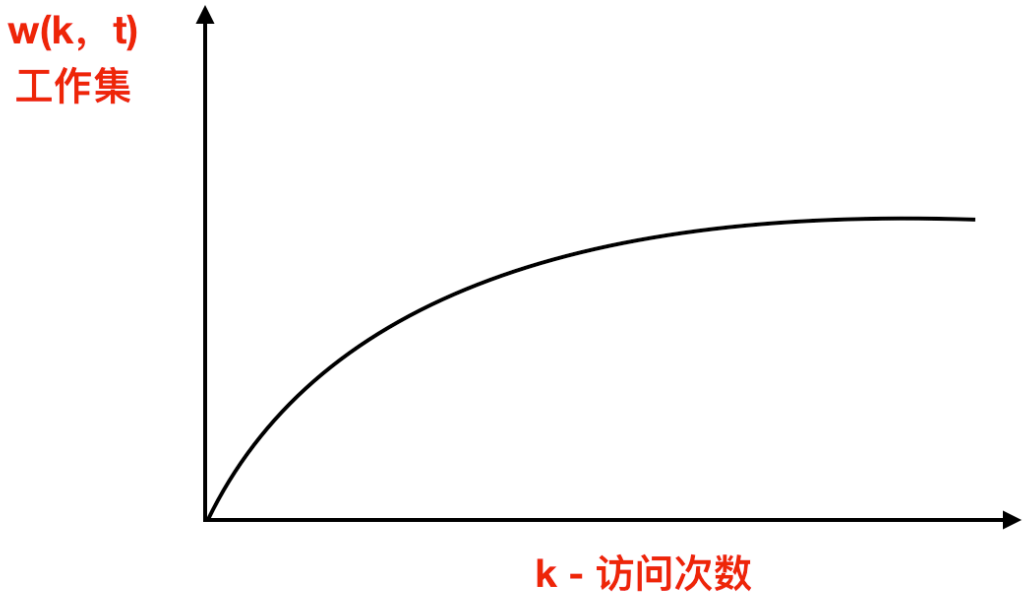
一个进程当前正在使用的页面的集合称为它的 `工作集(working set)`，如果整个工作集都在内存中，那么进程在运行到下一运行阶段（例如，编译器的下一遍扫面）之前，不会产生很多缺页中断。**如果内存太小从而无法容纳整个工作集，那么进程的运行过程中会产生大量的缺页中断，会导致运行速度也会变得缓慢**。因为通常只需要几纳秒就能执行一条指令，而通常需要十毫秒才能从磁盘上读入一个页面。如果一个程序每 10 ms 只能执行一到两条指令，那么它将需要很长时间才能运行完。如果只是执行几

条指令就会产生中断，那么就称作这个程序产生了 `颠簸(thrashing)` 。

在多道程序的系统中，通常会把进程移到磁盘上（即从内存中移走所有的页面），这样可以让其他进程有机会占用 CPU 。有一个问题是，当进程想要再次把之前调回磁盘的页面调回内存怎么办？从技术的角度上来讲，并不需要做什么，此进程会一直产生缺页中断直到它的 `工作集` 被调回内存。然后，每次装入一个进程需要 20、100 甚至 1000 次缺页中断，速度显然太慢了，并且由于 CPU 需要几毫秒时间处理一个缺页中断，因此由相当多的 CPU 时间也被浪费了。

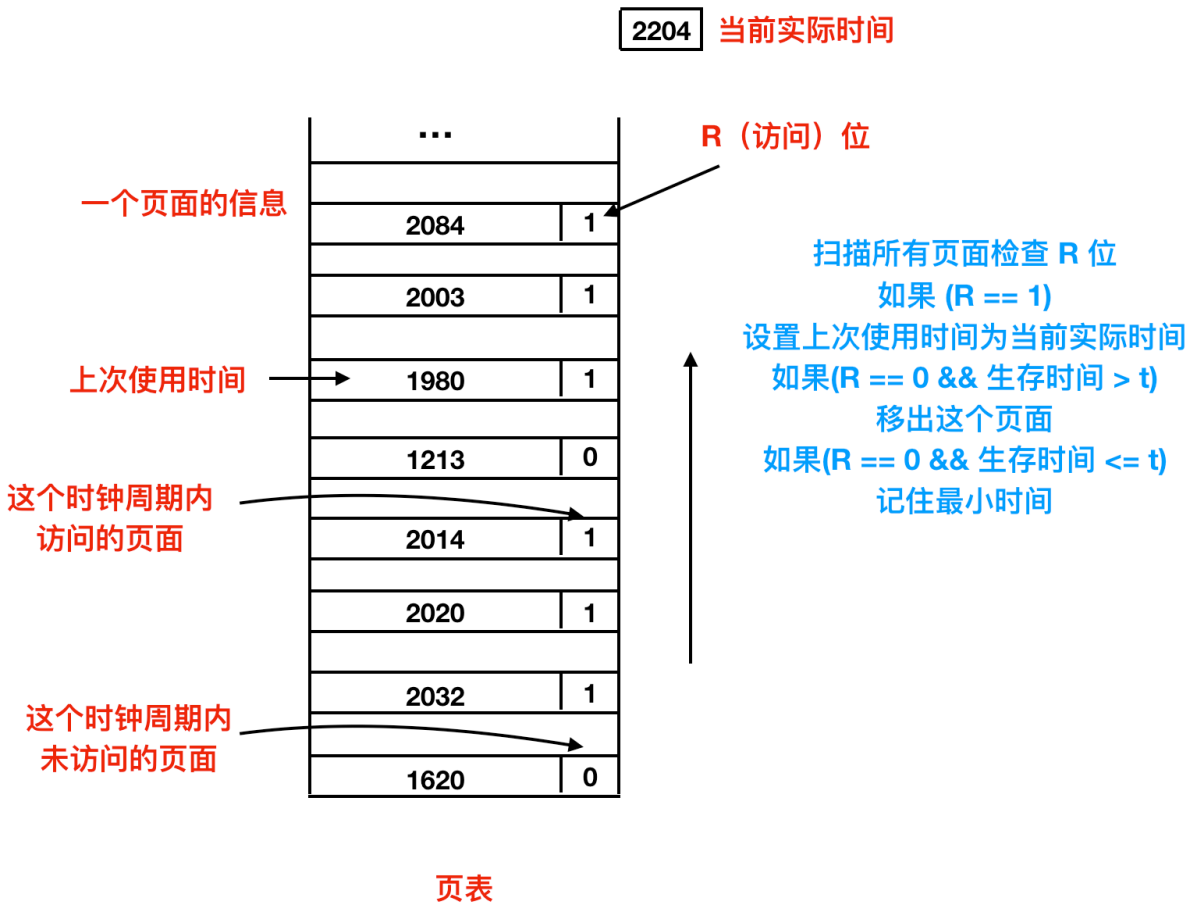
因此，不少分页系统中都会设法跟踪进程的工作集，确保这些工作集在进程运行时被调入内存。这个方法叫做 `工作集模式(working set model)` 。它被设计用来减少缺页中断的次数的。在进程运行前首先装入工作集页面的这一个过程被称为 `预先调页(prepaging)` ，工作集是随着时间来变化的。

根据研究表明，大多数程序并不是均匀的访问地址空间的，而访问往往是集中于一小部分页面。一次内存访问可能会取出一条指令，也可能会取出数据，或者是存储数据。在任一时刻 t，都存在一个集合，它包含所哟欧最近 k 次内存访问所访问过的页面。这个集合 `w(k,t)` 就是工作集。因为最近 k = 1次访问肯定会访问最近 k > 1 次访问所访问过的页面，所以 `w(k,t)` 是 k 的单调递减函数。随着 k 的增大， `w(k,t)` 是不会无限变大的，因为程序不可能访问比所能容纳页面数量上限还多的页面。



事实上大多数应用程序只会任意访问一小部分页面集合，但是这个集合会随着时间而缓慢变化，所以为什么一开始曲线会快速上升而 k 较大时上升缓慢。为了实现工作集模型，操作系统必须跟踪**哪些页面在工作集中**。一个进程从它开始执行到当前所实际使用的 CPU 时间总数通常称作 `当前实际运行时间` 。进程的工作集可以被称为在过去的 t 秒实际运行时间中它所访问过的页面集合。

下面来简单描述一下工作集的页面置换算法，基本思路就是找出一个不在工作集中的页面并淘汰它。下面是一部分机器页表



因为只有那些在内存中的页面才可以作为候选者被淘汰，所以该算法忽略了那些不在内存中的页面。每个表项至少包含两条信息：上次使用该页面的近似时间和 R（访问）位。空白的矩形表示该算法不需要其他字段，例如页框数量、保护位、修改位。

算法的工作流程如下，假设硬件要设置 R 和 M 位。同样的，在每个时钟周期内，一个周期性的时钟中断会使软件清除 `Referenced(引用)` 位。在每个缺页异常，页表会被扫描以找出一个合适的页面把它置换。

随着每个页表项的处理，都需要检查 R 位。如果 R 位是 1，那么就会将当前时间写入页表项的 `上次使用时间` 域，表示的意思就是缺页异常发生时页面正在被使用。因为页面在当前时钟周期内被访问过，那么它应该出现在工作集中而不是被删除（假设 t 是横跨了多个时钟周期）。

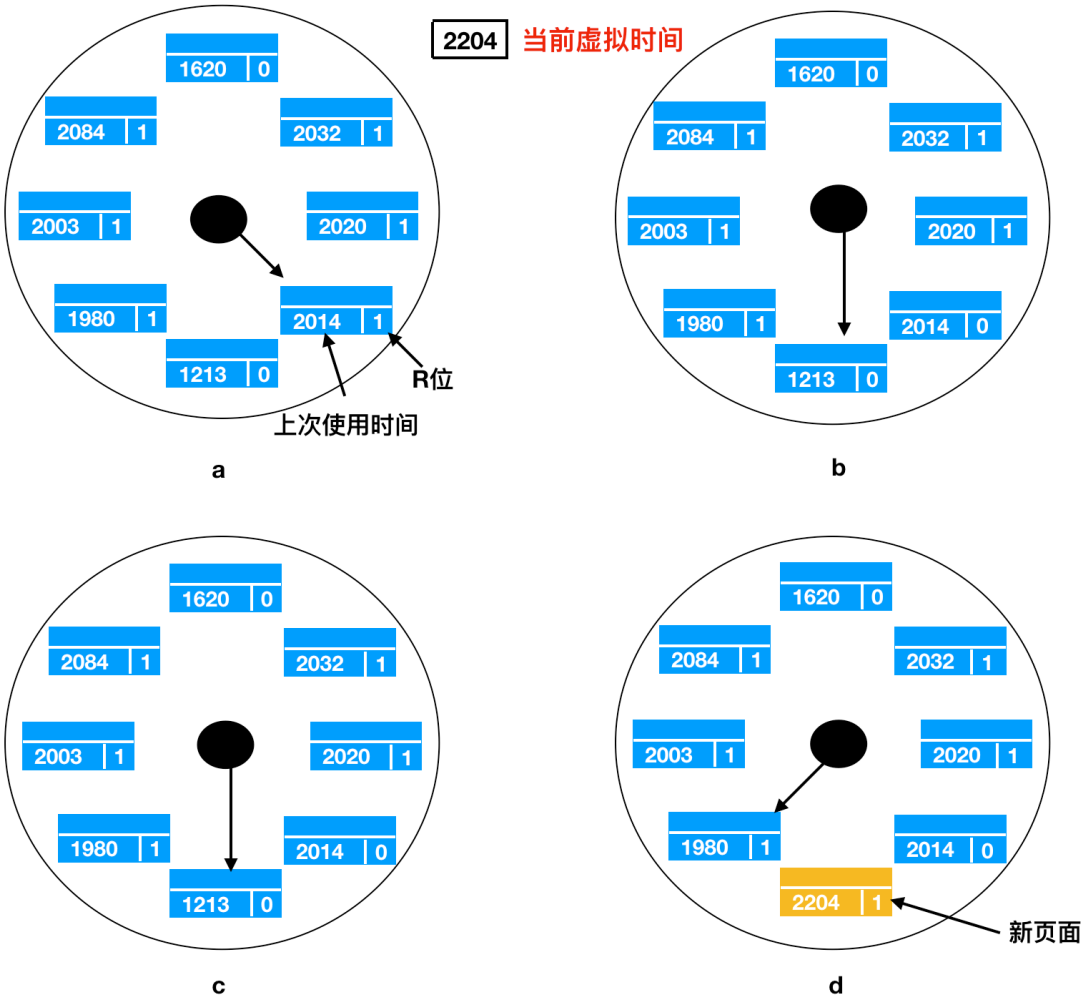
如果 R 位是 0，那么在当前的时钟周期内这个页面没有被访问过，应该作为被删除的对象。为了查看是否应该将其删除，会计算其使用期限（当前虚拟时间 - 上次使用时间），来用这个时间和 t 进行对比。如果使用期限大于 t，那么这个页面就不再工作集中，而使用新的页面来替换它。然后继续扫描更新剩下的表项。

然而，如果 R 位是 0 但是使用期限小于等于 t，那么此页应该在工作集中。此时就会把页面临时保存起来，但是会记 `生存时间最长（即上次使用时间的最小值）` 的页面。如果扫描完整个页表却没有找到适合被置换的页面，也就意味着所有的页面都在工作集中。在这种情况下，如果找到了一个或者多个 R = 0 的页面，就淘汰生存时间最长的页面。最坏的情况下是，在当前时钟周期内，所有的页面都被访问过了（也就是都有 R = 1），因此就随机选择一个页面淘汰，如果有的话最好选一个未被访问的页面，也就是干净的页面。

工作集时钟页面置换算法

当缺页异常发生后，需要扫描整个页表才能确定被淘汰的页面，因此基本工作集算法还是比较浪费时间的。一个对基本工作集算法的提升是基于时钟算法但是却使用工作集的信息，这种算法称为 `WSClock(工作集时钟)` 。由于它的实现简单并且具有高性能，因此在实践中被广泛应用。

与时钟算法一样，所需的数据结构是一个以页框为元素的循环列表，就像下面这样



工作集时钟页面置换算法的操作：a) 和 b) 给出 R = 1 时所发生的情形；c) 和 d) 给出 R = 0 的例子

最初的时候，该表是空的。当装入第一个页面后，把它加载到该表中。随着更多的页面的加入，它们形成一个环形结构。每个表项包含来自基本工作集算法的上次使用时间，以及 R 位（已标明）和 M 位（未标明）。

与时钟算法一样，在每个缺页异常时，首先检查指针指向的页面。如果 R 位被是设置为 1，该页面在当前时钟周期内就被使用过，那么该页面就不适合被淘汰。然后把该页面的 R 位置为 0，指针指向下一个页面，并重复该算法。该事件序列化后的状态参见图 b。

现在考虑指针指向的页面 R = 0 时会发生什么，参见图 c，如果页面的使用期限大于 t 并且页面为被访问过，那么这个页面就不会在工作集中，并且在磁盘上会有一个此页面的副本。申请重新调入一个新的页面，并把新的页面放在其中，如图 d 所示。另一方面，如果页面被修改过，就不能重新申请页面，因为这个页面在磁盘上没有有效的副本。为了避免由于调度写磁盘操作引起的进程切换，指针继续向前走，算法继续对下一个页面进行操作。毕竟，有可能存在一个老的，没有被修改过的页面可以立即使用。

原则上来说，所有的页面都有可能因为 `磁盘I/O` 在某个时钟周期内被调度。为了降低磁盘阻塞，需要设置一个限制，即最大只允许写回 n 个页面。一旦达到该限制，就不允许调度新的写操作。

那么就有个问题，指针会绕一圈回到原点的，如果回到原点，它的起始点会发生什么？这里有两种情况：

- 至少调度了一次写操作
- 没有调度过写操作

在第一种情况中，指针仅仅是不停的移动，寻找一个未被修改过的页面。由于已经调度了一个或者多个写操作，最终会有某个写操作完成，它的页面会被标记为未修改。置换遇到的第一个未被修改过的页面，这个页面不一定是第一个被调度写操作的页面，因为硬盘驱动程序为了优化性能可能会把写操作重排序。

对于第二种情况，所有的页面都在工作集中，否则将至少调度了一个写操作。由于缺乏额外的信息，最简单的方法就是置换一个未被修改的页面来使用，扫描中需要记录未被修改的页面的位置，如果不存在未被修改的页面，就选定当前页面并把它写回磁盘。

页面置换算法小结

我们到现在已经研究了各种页面置换算法，现在我们来一个简单的总结，算法的总结归纳如下

算法	注释
最优算法	不可实现，但可以用作基准
NRU(最近未使用) 算法	和 LRU 算法很相似
FIFO(先进先出) 算法	有可能会抛弃重要的页面
第二次机会算法	比 FIFO 有较大的改善
时钟算法	实际使用
LRU(最近最少)算法	比较优秀，但是很难实现
NFU(最不经常食用)算法	和 LRU 很类似
老化算法	近似 LRU 的高效算法
工作集算法	实施起来开销很大
工作集时钟算法	比较有效的算法

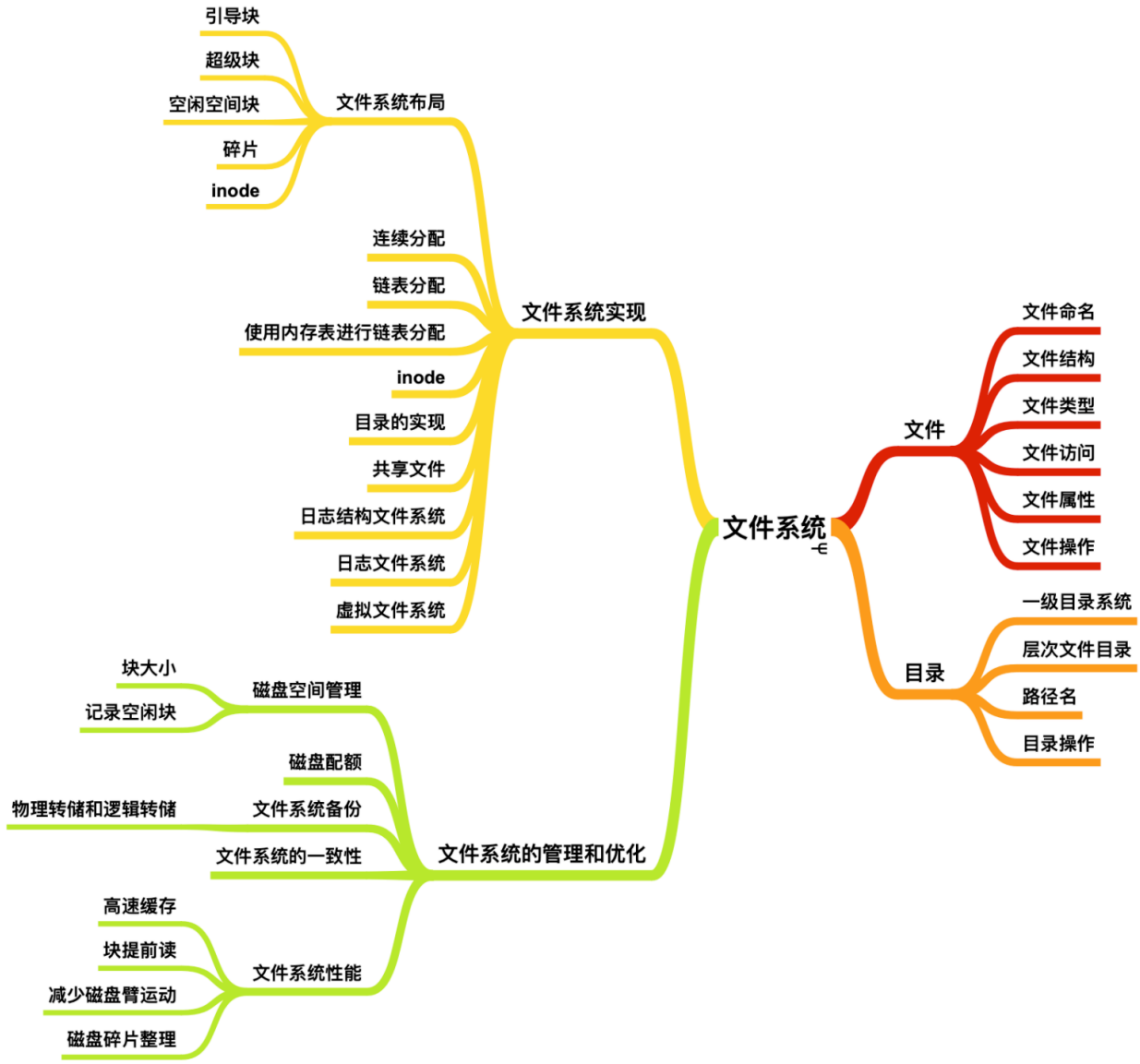
- `最优算法` 在当前页面中置换最后要访问的页面。不幸的是，没有办法来判定哪个页面是最后一个要访问的，`因此实际上该算法不能使用`。然而，它可以作为衡量其他算法的标准。
- `NRU` 算法根据 R 位和 M 位的状态将页面氛围四类。从编号最小的类别中随机选择一个页面。NRU 算法易于实现，但是性能不是很好。存在更好的算法。
- `FIFO` 会跟踪页面加载进入内存中的顺序，并把页面放入一个链表中。有可能删除存在时间最长但是还在使用的页面，因此这个算法也不是一个很好的选择。
- `第二次机会` 算法是对 FIFO 的一个修改，它会在删除页面之前检查这个页面是否仍在使用。如果页面正在使用，就会进行保留。这个改进大大提高了性能。
- `时钟` 算法是第二次机会算法的另外一种实现形式，时钟算法和第二次算法的性能差不多，但是会花费更少的时间来执行算法。
- `LRU` 算法是一个非常优秀的算法，但是没有 `特殊的硬件(TLB)` 很难实现。如果没有硬件，就不

能使用 LRU 算法。

- `NFU` 算法是一种近似于 LRU 的算法，它的性能不是非常好。
- `老化` 算法是一种更接近 LRU 算法的实现，并且可以更好的实现，因此是一个很好的选择
- 最后两种算法都使用了工作集算法。工作集算法提供了合理的性能开销，但是它的实现比较复杂。 `WSClock` 是另外一种变体，它不仅能够提供良好的性能，而且可以高效地实现。

总之，**最好的算法是老化算法和WSClock算法**。他们分别是基于 LRU 和工作集算法。他们都具有良好的性能并且能够被有效的实现。还存在其他一些好的算法，但实际上这两个可能是最重要的。

文件系统



所有的应用程序都需要 `存储` 和 `检索` 信息。进程运行时，它能够在自己的存储空间内存储一定量的信息。然而，存储容量受 `虚拟地址空间` 大小的限制。对于一些应用程序来说，存储空间的大小是充足的，但是对于其他一些应用程序，比如航空订票系统、银行系统、企业记账系统来说，这些容量又显得太小了。

第二个问题是，当进程终止时信息会丢失。对于一些应用程序（例如数据库），信息会长久保留。在这些进程终止时，相关的信息应该保留下来，是不能丢失的。甚至这些应用程序崩溃后，信息也应该保留下来。

第三个问题是，通常需要很多进程在同一时刻访问这些信息。解决这种问题的方式是把这些信息单独保留在各自的进程中。

因此，对于长久存储的信息我们有三个基本需求：

- **必须要有可能存储的大量的信息**
- **信息必须能够在进程终止时保留**
- **必须能够使多个进程同时访问有关信息**

`磁盘(Magnetic disk)` 一直是用来长久保存信息的设备。近些年来，`固态硬盘` 逐渐流行起来。



固态硬盘不仅没有易损坏的移动部件，而且能够提供快速的随机访问。相比而言，虽然磁带和光盘也被广泛使用，但是它们的**性能相对较差**，通常应用于备份。我们会在后面探讨磁盘，现在姑且把磁盘当作一种大小固定块的线性序列好了，并且支持如下操作

- 读块 k
- 写块 k



事实上磁盘支持更多的操作，**但是只要有了读写操作，原则上就能够解决长期存储的问题**。

然而，磁盘还有一些不便于实现的操作，特别是在有很多程序或者多用户使用的大型系统上（如服务器）。在这种情况下，很容易产生一些问题，例如

- 你如何找到这些信息?
- 你如何保证一个用户不会读取另外一个用户的数据?
- 你怎么知道哪些块是空闲的？等等问题

我们可以针对这些问题提出一个新的抽象 - `文件` 。**进程和线程的抽象、地址空间和文件都是操作系统的重要概念**。如果你能真正深入了解这三个概念，那么你就走上了成为操作系统专家的道路。

`文件(Files)` 是由进程创建的逻辑信息单元。一个磁盘会包含几千甚至几百万个文件，每个文件是独立于其他文件的。事实上，如果你能把每个文件都看作一个独立的地址空间，那么你就可以真正理解文件的概念了。

进程能够读取已经存在的文件，并在需要时重新创建他们。存储在文件中的信息必须是 `持久的` ，这也就是说，不会因为进程的创建和终止而受影响。一个文件只能在当用户明确删除的时候才能消失。尽管读取和写入都是最基本的操作，但还有许多其他操作，我们将在下面介绍其中的一些。

文件由操作系统进行管理，有关文件的构造、命名、访问、使用、保护、实现和管理方式都是操作系统设计的主要内容。从总体上看，操作系统中处理文件的部分称为 `文件系统(file system)` ，这就是我们所讨论的。

从用户角度来说，用户通常会关心文件是由什么组成的，如何给文件进行命名，如何保护文件，以及可以对文件进行哪些操作等等。尽管是用链表还是用位图记录内存空闲区并不是用户所关心的主题，而这些对系统设计人员来说至关重要。下面我们就来探讨一下这些主题

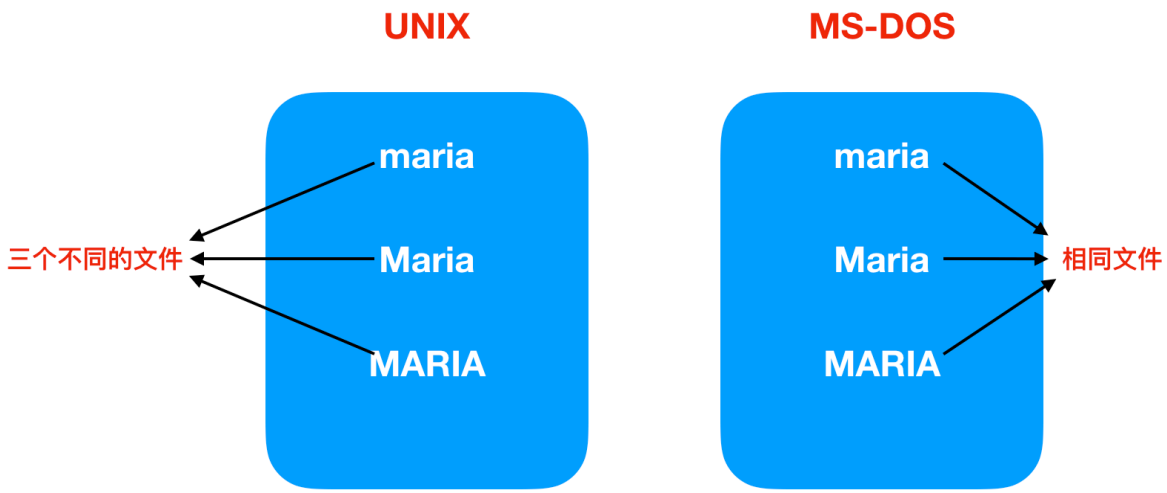
文件

文件命名

文件是一种抽象机制，它提供了一种方式用来存储信息以及在后面进行读取。可能任何一种机制最重要的特性就是管理对象的命名方式。在创建一个文件后，它会给文件一个命名。当进程终止时，文件会继续存在，并且其他进程可以使用 名称访问该文件 。

文件命名规则对于不同的操作系统来说是不一样的，但是所有现代操作系统都允许使用 1 - 8 个字母的字符串作为合法文件名。

某些文件区分大小写字母，而大多数则不区分。 UNIX 属于第一类；历史悠久的 MS-DOS 属于第二类（顺便说一句，尽管 MS-DOS 历史悠久，但 MS-DOS 仍在嵌入式系统中非常广泛地使用，因此它绝不是过时的）；因此，UNIX 系统会有三种不同的命名文件： maria 、 Maria 、 MARIA 。在 MS-DOS ，所有这些命名都属于相同的文件。



这里可能需要在文件系统上预留一个位置。Windows 95 和 Windows 98 都使用了 MS-DOS 文件系统，叫做 FAT-16 ，因此继承了它的一些特征，例如有关文件名的构造方法。Windows 98 引入了对 FAT-16 的一些扩展，从而导致了 FAT-32 的生成，但是这两者很相似。另外，Windows NT，Windows 2000，Windows XP，Windows Vista，Windows 7 和 Windows 8 都支持 FAT 文件系统，这种文件系统有些过时。然而，这些较新的操作系统还具有更高级的 本机文件系统(NTFS) ，有不同的特性，那就是基于 Unicode 编码的文件名。事实上，Windows 8 还配备了另一种文件系统，简称 ReFS(Resilient File System) ，但这个文件系统一般应用于 Windows 8 的服务器版本。下面除非我们特殊声明，否则我们在提到 MS-DOS 和 FAT 文件系统的时候，所指的就是 Windows 的 FAT-16 和 FAT-32。这里要说一下，有一种类似 FAT 的新型文件系统，叫做 exFAT 。它是微软公司对闪存和大文件系统开发的一种优化的 FAT 32 扩展版本。ExFAT 是现在微软唯一能够满足 OS X 读写操作的文件系统。

许多操作系统支持两部分的文件名，它们之间用 . 分隔开，比如文件名 prog.c 。原点后面的文件称为 文件扩展名(file extension) ，文件扩展名通常表示文件的一些信息。例如在 MS-DOS 中，文件名是 1 - 8 个字符，加上 1 - 3 个字符的可选扩展名组成。在 UNIX 中，如果有扩展名，那么扩展名的长度将由用户来决定，一个文件甚至可以包括两个或更多的扩展名，例如 homepage.html.zip ，html 表示一个 web 网页而 .zip 表示文件 homepage.html 已经采用 zip 程序压缩完成。一些常用的文件扩展名以及含义如下图所示

扩展名	含义
bak	备份文件
c	c 源程序文件
gif	符合图形交换格式的图像文件
hlp	帮助文件
html	WWW 超文本标记语言文档
jpg	符合 JPEG 编码标准的静态图片
mp3	符合 MP3 音频编码格式的音乐文件
mpg	符合 MPEG 编码标准的电影
o	目标文件（编译器输出格式，尚未链接）
pdf	pdf 格式的文件
ps	PostScript 文件
tex	为 TEX 格式化程序准备的输入文件
txt	文本文件
zip	压缩文件

在 UNIX 系统中，文件扩展名只是一种约定，操作系统并不强制采用。

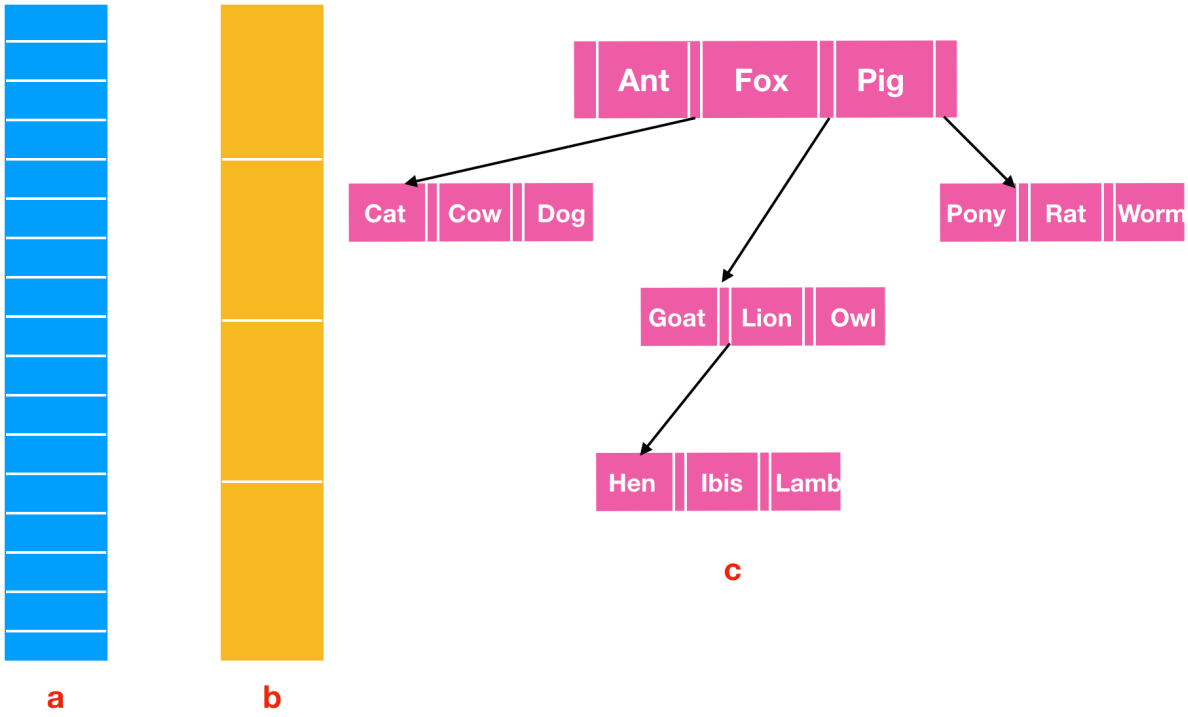
名为 `file.txt` 的文件是文本文件，这个文件名更多的是提醒所有者，而不是给计算机传递信息。但是另一方面，C 编译器可能要求它编译的文件以 `.c` 结尾，否则它会拒绝编译。然而，操作系统并不关心这一点。

对于可以处理多种类型的程序，约定就显得及其有用。例如 C 编译器可以编译、链接多种文件，包括 C 文件和汇编语言文件。这时扩展名就很有必要，编译器利用它们区分哪些是 C 文件，哪些是汇编文件，哪些是其他文件。因此，扩展名对于编译器判断哪些是 C 文件，哪些是汇编文件以及哪些是其他文件变得至关重要。

与 UNIX 相反，Windows 就会关注扩展名并对扩展名赋予了新的含义。`用户(或进程)` 可以在操作系统中注册 `扩展名`，并且规定哪个程序能够拥有扩展名。当用户双击某个文件名时，拥有该文件名的程序就启动并运行文件。例如，双击 file.docx 启动了 Word 程序，并以 file.docx 作为初始文件。

文件结构

文件的构造有多种方式。下图列出了常用的三种构造方式



三种不同的文件。 a) 字节序列 。b) 记录序列。c) 树

上图中的 a 是一种无结构的字节序列，操作系统不关心序列的内容是什么，操作系统能看到的就是 `字节(bytes)` 。其文件内容的任何含义只在用户程序中进行解释。UNIX 和 Windows 都采用这种办法。

把文件看成字节序列提供了最大的灵活性。用户程序可以向文件中写任何内容，并且可以通过任何方便的形式命名。操作系统不会为为用户写入内容提供帮助，当然也不会干扰阻塞你。对于想做特殊操作的用户来说，后者是十分重要的。所有的 UNIX 版本（包括 Linux 和 OS X）和 Windows 都使用这种文件模型。

图 b 表示在文件结构上的第一部改进。在这个模型中，文件是具有固定长度记录的序列，每个记录都有其内部结构。 把文件作为记录序列的核心思想是：**读操作返回一个记录，而写操作重写或者追加一个记录**。第三种文件结构如上图 c 所示。在这种组织结构中，文件由一颗 `记录树` 构成，记录树的长度不一定相同，每个记录树都在记录中的固定位置包含一个 `key` 字段。这棵树按 key 进行排序，从而可以对特定的 key 进行快速查找。

在记录树的结构中，可以取出下一个记录，但是最关键的还是根据 key 搜索指定的记录。如上图 c 所示，用户可以读出指定的 `pony` 记录，而不必关心记录在文件中的确切位置。用户也可以在文件中添加新的记录。但是用户不能决定添加到何处位置，添加到何处位置是由 `操作系统` 决定的。

文件类型

很多操作系统支持多种文件类型。例如，UNIX（同样包括 OS X）和 Windows 都具有常规的文件和目录。除此之外，UNIX 还具有 `字符特殊文件(character special file)` 和 `块特殊文件(block special file)` 。 `常规文件(Regular files)` 是包含有用户信息的文件。用户一般使用的文件大都是常规文件，常规文件一般包括 **可执行文件、文本文件、图像文件**，从常规文件读取数据或将数据写入时，内核会根据文件系统的规则执行操作，是写入可能被延迟，记录日志或者接受其他操作。

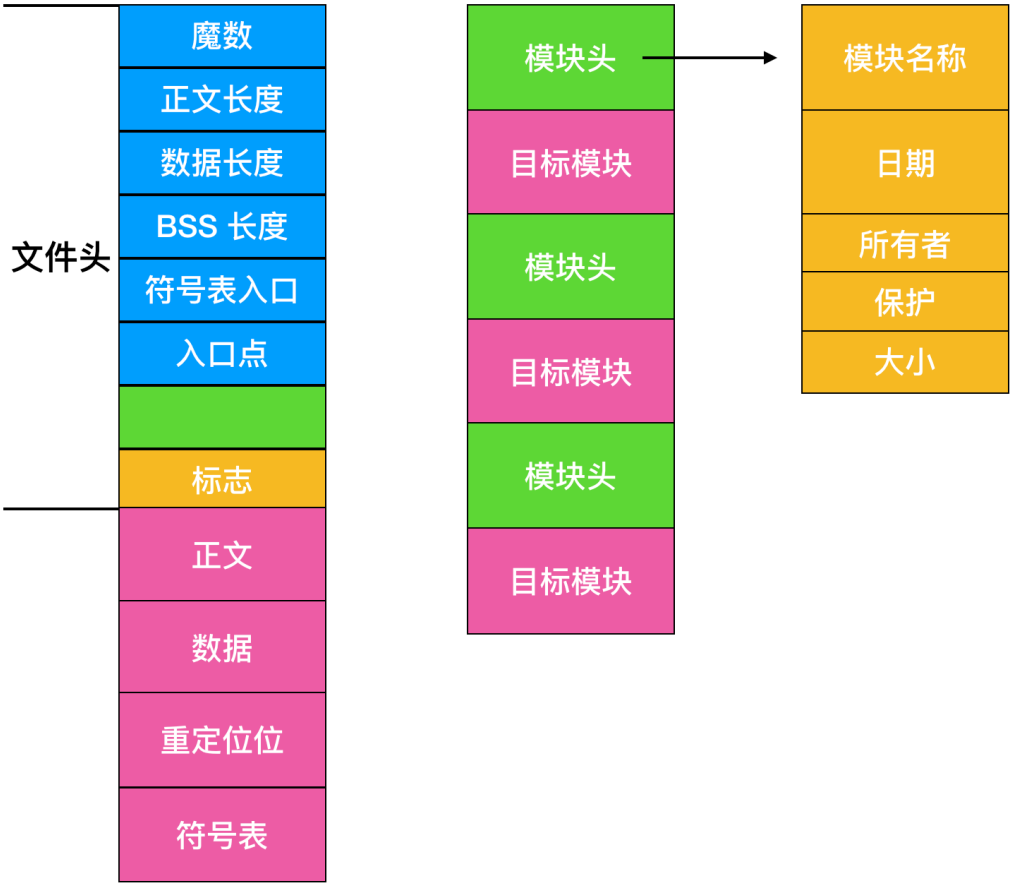
字符特殊文件和输入/输出有关，用于串行 I/O 类设备，如终端、打印机、网络等。块特殊文件用于磁盘类设备。我们主要讨论的是常规文件。

常规文件一般分为 `ASCII` 码文件或者二进制文件。ASCII 码文件由文本组成。在一些系统中，每行都会用回车符结束（ASCII码是13，控制字符 CR，转义字符 `\r` 。），另外一些则会使用换行符（ASCII码是10，控制字符LF，转义字符 `\n` ）。一些系统（比如 Windows）两者都会使用。

ASCII 文件的优点在于 `显示` 和 `打印` ，还可以用任何文本编辑器进行编辑。进一步来说，如果许多应用程序使用 ASCII 码作为输入和输出，那么很容易就能够把多个程序连接起来，一个程序的输出可能是另一个程序的输入，就像管道一样。



其他与 ASCII 不同的是二进制文件。打印出来的二进制文件是无法理解的。下面是一个二进制文件的格式，它取自早期的 UNIX 。尽管从技术上来看这个文件只是字节序列，但是操作系统只有在文件格式正确的情况下才会执行。



a）一个可执行文件　b）一个存档文件

这个文件有五个段：文件头、征文、数据、重定位位和符号表。文件头以 `魔数(magic number)` 为开始，表明这个文件是一个可执行文件（以防止意外执行非此格式的文件）。然后是文件各个部分的大小，开始执行的标志以及一些标志位。程序本身的正文和数据在 `文件头` 后面，他们被加载到内存中或者重定位会根据 `重定位位` 进行判断。符号表则用于 `调试` 。

二进制文件的另外一种形式是 `存档文件` ，它由已编译但没有链接的库过程（模块）组合而成。每个文件都以模块头开始，其中记录了**名称、创建日期、所有者、保护码和文件大小**。和可执行文件一样，模块头也都是二进制数，将它们复制到打印机将会产生乱码。

所有的操作系统必须至少能够识别一种文件类型：它自己的可执行文件。以前的 TOPS-20 系统（用于 DECsystem 20）甚至要检查要执行的任何文件的创建时间，为了定位资源文件来检查自动文件创建后是否被修改过。如果被修改过了，那么就会自动编译文件。在 UNIX 中，就是在 shell 中嵌入 `make` 程序。此时操作系统要求用户必须采用固定的文件扩展名，从而确定哪个源程序生成哪个二进制文件。

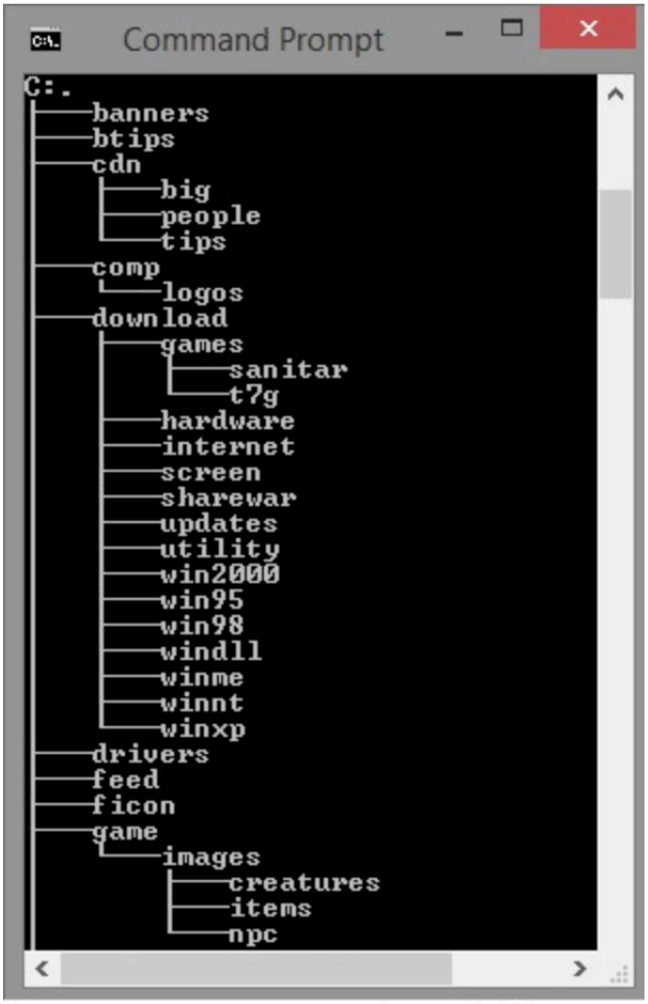
> 什么是 make 程序？在软件发展过程中，make 程序是一个自动编译的工具，它通过读取称为 `Makefiles` 的文件来自动从源代码构建可执行程序和库，该文件指定了如何导出目标程序。尽管集成开发环境和特定于语言的编译器功能也可以用于管理构建过程，但 Make 仍被广泛使用，尤其是在 Unix 和类似 Unix 的操作系统中使用。

当程序从文件中读写数据时，请求会转到 `内核处理程序(kernel driver)` 。如果文件是常规文件，则数据由文件系统驱动程序处理，并且通常存储在磁盘或其他存储介质上的某块区域中，从文件中读取的数据就是之前在该位置写入的数据。

当数据读取或写入到设备文件时，请求会被设备驱动程序处理。每个设备文件都有一个关联的编号，该编号标示要使用的设备驱动程序。设备处理数据的工作是它自己的事儿。

- `块设备` 也叫做块特殊文件，它的行为通常与普通文件相似：它们是字节数组，并且在给定位置读取的值是最后写入该位置的值。来自块设备的数据可以缓存在内存中，并从缓存中读取；写入可以被缓冲。块设备通常是可搜索的，块设备的概念是，相应的硬件可以一次读取或者写入整个块，例如磁盘上的一个扇区
- `字符设备` 也称为字符特殊文件，它的行为类似于管道、串行端口。将字节写入字符设备可能会导致它在屏幕上显示，在串行端口上输出，转换为声音。

`目录(Directories)` 是管理文件系统结构的系统文件。它是用于在计算机上存储文件的位置。目录位于 `分层文件系统` 中，例如 Linux，MS-DOS 和 UNIX。



图为 Windows / DOS 树命令的输出示例。

它显示所有本地和子目录（例如，cdn 目录中的 big 目录）。当前目录是 C 盘驱动器的 `根目录` 。之所以称为根目录，是因为该目录下没有任何内容，而其他目录都在该目录下 `分支` 。

文件访问

早期的操作系统只有一种访问方式： `序列访问(sequential access)` 。在这些系统中，进程可以按照顺序读取所有的字节或文件中的记录，但是不能跳过并乱序执行它们。顺序访问文件是可以返回到起点的，需要时可以多次读取该文件。当存储介质是磁带而不是磁盘时，顺序访问文件很方便。

在使用磁盘来存储文件时，可以不按照顺序读取文件中的字节或者记录，或者按照关键字而不是位置来访问记录。这种能够以任意次序进行读取的称为 `随机访问文件(random access file)` 。许多应用程序都需要这种方式。

随机访问文件对许多应用程序来说都必不可少，例如，数据库系统。如果乘客打电话预定某航班机票，订票程序必须能够直接访问航班记录，而不必先读取其他航班的成千上万条记录。

有两种方法可以指示从何处开始读取文件。第一种方法是直接使用 `read` 从头开始读取。另一种是用一个特殊的 `seek` 操作设置当前位置，在 seek 操作后，从这个当前位置顺序地开始读文件。UNIX 和 Windows 使用的是后面一种方式。

文件属性

文件包括文件名和数据。除此之外，所有的操作系统还会保存其他与文件相关的信息，如文件创建的日期和时间、文件大小。我们可以称这些为文件的 `属性(attributes)` 。有些人也喜欢把它们称作 `元数据(metadata)` 。文件的属性在不同的系统中差别很大。文件的属性只有两种状态： `设置(set)` 和 `清除(clear)` 。下面是一些常用的属性

属性	含义
保护	谁可以访问文件、以什么方式存取文件
密码（口令）	访问文件所需要的密码（口令）
创建者	创建文件者的 ID
所有者	当前所有者
只读标志	0 表示读/写，1 表示只读
隐藏标志	0 表示正常，1 表示不再列表中显示
系统标志	0 表示普通文件，1 表示系统文件
存档标志	0 表示已经备份，1 表示需要备份
ASCII / 二进制标志	0 表示 ASCII 文件，1 表示二进制文件
随机访问标志	0 表示只允许顺序访问，1 表示随机访问
临时标志	0 表示正常，1 表示进程退出时删除该文件
加锁标志	0 表示未加锁，1 表示加锁
记录长度	一个记录中的字节数
键的位置	每个记录中的键的偏移量
键的长度	键字段的字节数
创建时间	创建文件的日期和时间
最后一次存取时间	上一次访问文件的日期和时间
最后一次修改时间	上一次修改文件的日期和时间
当前大小	文件的字节数
最大长度	文件可能增长到的字节数

没有一个系统能够同时具有上面所有的属性，但每个属性都在某个系统中采用。

前面四个属性（保护，口令，创建者，所有者）与文件保护有关，它们指出了谁可以访问这个文件，谁不能访问这个文件。

> `保护（File Protection）`：用于保护计算机上有价值数据的方法。文件保护是通过密码保护文件或者仅仅向特定用户或组提供权限来实现。

在一些系统中，用户必须给出口令才能访问文件。`标志(flags)`是一些位或者短属性能够控制或者允许特定属性。

- `隐藏文件位(hidden flag)`表示该文件不在文件列表中出现。

- `存档标志位(archive flag)` 用于记录文件是否备份过，由备份程序清除该标志位；若文件被修改，操作系统则设置该标志位。用这种方法，备份程序可以知道哪些文件需要备份。
- `临时标志位(temporary flag)` 允许文件被标记为是否允许自动删除当进程终止时。

`记录长度(record-length)` 、 `键的位置(key-position)` 和 `键的长度(key-length)` 等字段只能出现在用关键字查找记录的文件中。它们提供了查找关键字所需要的信息。

不同的时间字段记录了文件的创建时间、最近一次访问时间以及最后一次修改时间，它们的作用不同。例如，目标文件生成后被修改的源文件需要重新编译生成目标文件。这些字段提供了必要的信息。

当前大小字段指出了当前的文件大小，一些旧的大型机操作系统要求在创建文件时指定文件呢最大值，以便让操作系统提前保留最大存储值。但是一些服务器和个人计算机却不用设置此功能。

文件操作

使用文件的目的是用来存储信息并方便以后的检索。对于存储和检索，不同的系统提供了不同的操作。以下是与文件有关的最常用的一些系统调用：

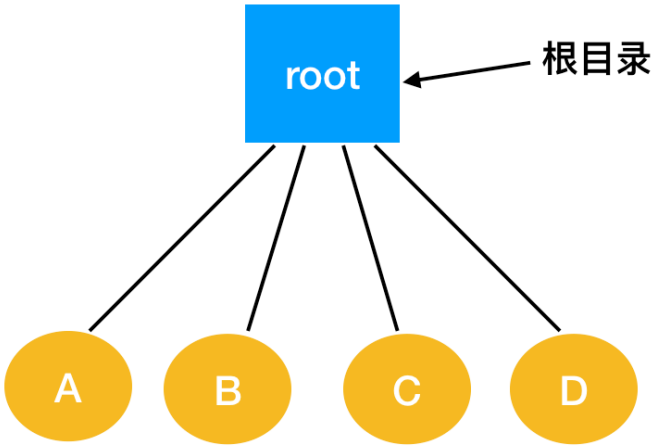
1. `Create` ，创建不包含任何数据的文件。调用的目的是表示文件即将建立，并对文件设置一些属性。
2. `Delete` ，当文件不再需要，必须删除它以释放内存空间。为此总会有一个系统调用来删除文件。
3. `Open` ，在使用文件之前，必须先打开文件。这个调用的目的是允许系统将属性和磁盘地址列表保存到主存中，用来以后的快速访问。
4. `Close` ，当所有进程完成时，属性和磁盘地址不再需要，因此应关闭文件以释放表空间。很多系统限制进程打开文件的个数，以此达到鼓励用户关闭不再使用的文件。磁盘以块为单位写入，关闭文件时会强制写入最后一 `块` ，即使这个块空间内部还不满。
5. `Read` ，数据从文件中读取。通常情况下，读取的数据来自文件的当前位置。调用者必须指定需要读取多少数据，并且提供存放这些数据的缓冲区。
6. `Write` ，向文件写数据，写操作一般也是从文件的当前位置开始进行。如果当前位置是文件的末尾，则会直接追加进行写入。如果当前位置在文件中，则现有数据被覆盖，并且永远消失。
7. `append` ，使用 append 只能向文件末尾添加数据。
8. `seek` ，对于随机访问的文件，要指定从何处开始获取数据。通常的方法是用 seek 系统调用把当前位置指针指向文件中的特定位置。seek 调用结束后，就可以从指定位置开始读写数据了。
9. `get attributes` ，进程运行时通常需要读取文件属性。
10. `set attributes` ，用户可以自己设置一些文件属性，甚至是在文件创建之后，实现该功能的是 set attributes 系统调用。
11. `rename` ，用户可以自己更改已有文件的名字，rename 系统调用用于这一目的。

目录

文件系统通常提供 `目录(directories)` 或者 `文件夹(folders)` 用于记录文件的位置，在很多系统中目录本身也是文件，下面我们会讨论关于文件，他们的组织形式、属性和可以对文件进行的操作。

一级目录系统

目录系统最简单的形式是有一个能够包含所有文件的目录。这种目录被称为 `根目录(root directory)` ，由于根目录的唯一性，所以其名称并不重要。在最早期的个人计算机中，这种系统很常见，部分原因是因为只有一个用户。下面是一个单层目录系统的例子

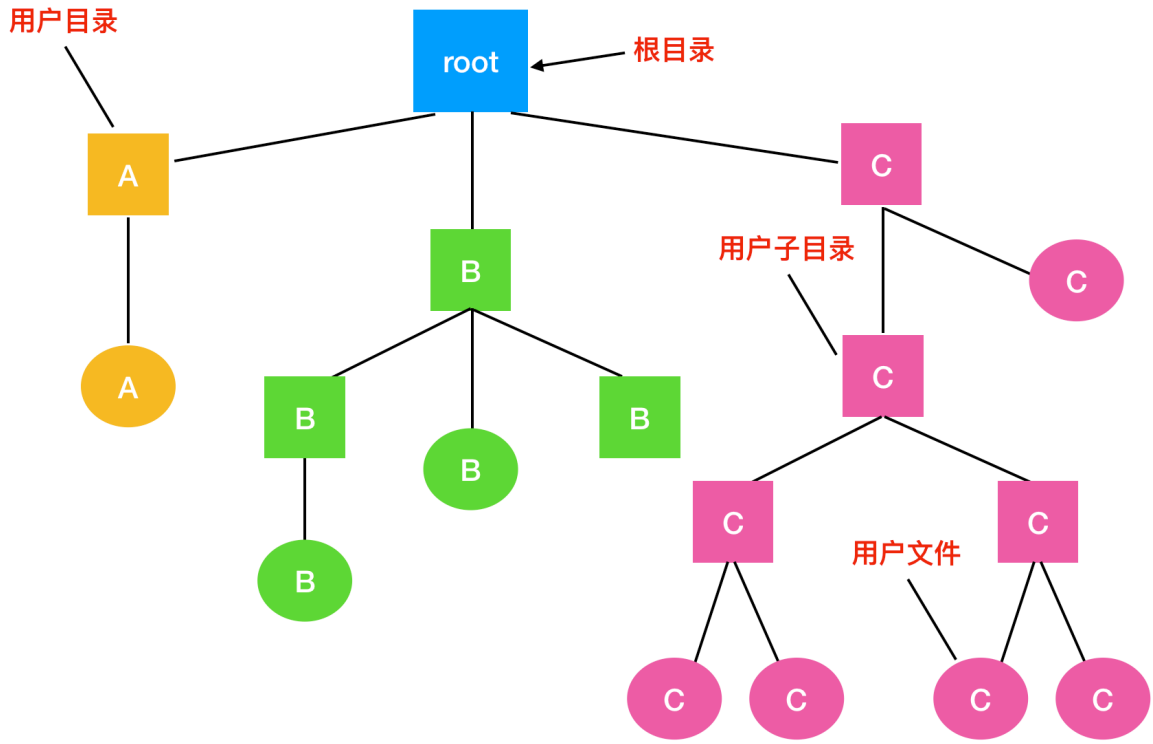


含有四个文件的单层目录系统

该目录中有四个文件。这种设计的优点在于简单，并且能够快速定位文件，毕竟只有一个地方可以检索。这种目录组织形式现在一般用于简单的嵌入式设备（如数码相机和某些便携式音乐播放器）上使用。

层次目录系统

对于简单的应用而言，一般都用单层目录方式，但是这种组织形式并不适合于现代计算机，因为现代计算机含有成千上万个文件和文件夹。如果都放在根目录下，查找起来会非常困难。为了解决这一问题，出现了 `层次目录系统(Hierarchical Directory Systems)` ，也称为 `目录树` 。通过这种方式，可以用很多目录把文件进行分组。进而，如果多个用户共享同一个文件服务器，比如公司的网络系统，每个用户可以为自己的目录树拥有自己的私人根目录。这种方式的组织结构如下



根目录含有目录 A、B 和 C，分别属于不同的用户，其中两个用户个字创建了 `子目录` 。用户可以创建任意数量的子目录，现代文件系统都是按照这种方式组织的。

路径名

当目录树组织文件系统时，需要有某种方法指明文件名。常用的方法有两种，第一种方式是每个文件都会用一个 `绝对路径名(absolute path name)` ，它由根目录到文件的路径组成。举个例子， `/usr/ast/mailbox` 意味着根目录包含一个子目录 `usr` ，usr 下面包含了一个 `mailbox` 。绝对路径名总是以 `/` 开头，并且是唯一的。在UNIX中，路径的组件由 `/` 分隔。在Windows中，分隔符为 `\` 。 在 MULTICS 中，它是 `>` 。 因此，在这三个系统中，相同的路径名将被编写如下

```
1   Windows \usr\ast\mailbox
2   UNIX /usr/ast/mailbox
3   MULTICS >usr>ast>mailbox
```

不论使用哪种方式，如果路径名的第一个字符是分隔符，那就是绝对路径。

另外一种指定文件名的方法是 `相对路径名(relative path name)` 。它常常和 `工作目录(working directory)` （也称作 `当前目录(current directory)` ）一起使用。用户可以指定一个目录作为当前工作目录。例如，如果当前目录是 `/usr/ast` ，那么绝对路径 `/usr/ast/mailbox` 可以直接使用 `mailbox` 来引用。也就是说，如果工作目录是 `/usr/ast` ，则 UNIX 命令

```
1   cp /usr/ast/mailbox  /usr/ast/mailbox.bak
```

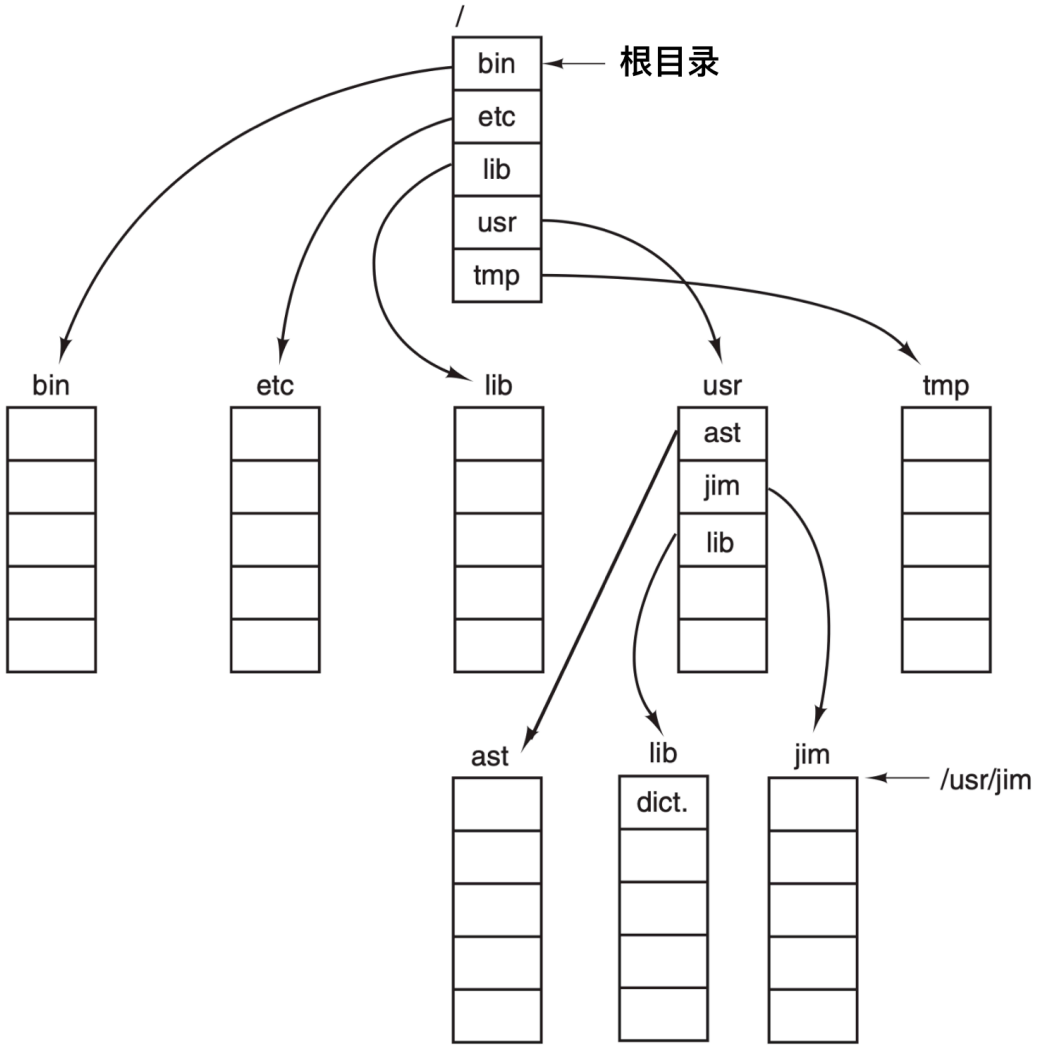
和命令

```
1   cp mailbox mailbox.bak
```

具有相同的含义。相对路径通常情况下更加方便和简洁。而它实现的功能和绝对路径安全相同。

一些程序需要访问某个特定的文件而不必关心当前的工作目录是什么。在这种情况下，应该使用绝对路径名。

支持层次目录结构的大多数操作系统在每个目录中有两个特殊的目录项 `.` 和 `..`，长读作 `dot` 和 `dotdot`。dot 指的是当前目录，dotdot 指的是其父目录（在根目录中例外，在根目录中指向自己）。可以参考下面的进程树来查看如何使用。



一个进程的工作目录是 `/usr/ast`，它可采用 `..` 沿树向上，例如，可用命令

```
cp ../lib/dictionary .
```

把文件 `usr/lib/dictionary` 复制到自己的目录下，第一个路径告诉系统向上找（到 usr 目录），然后向下到 `lib` 目录，找到 dictionary 文件

第二个参数 `.` 指定当前的工作目录，当 cp 命令用目录名作为最后一个参数时，则把全部的文件复制到该目录中。当然，对于上述复制，键入

```
cp /usr/lib/dictionary .
```

是更常用的方法。用户这里采用 `.` 可以避免键入两次 dictionary 。无论如何，键入

```
cp /usr/lib/dictionary dictionary
```

也可正常工作，就像键入

```
cp /usr/lib/dictionary /usr/lib/dictionary
```

一样。所有这些命令都能够完成同样的工作。

目录操作

不同文件中管理目录的系统调用的差别比管理文件的系统调用差别大。为了了解这些系统调用有哪些以及它们怎样工作，下面给出一个例子（取自 UNIX）。

1. `Create`，创建目录，除了目录项 `.` 和 `..` 外，目录内容为空。
2. `Delete`，删除目录，只有空目录可以删除。只包含 `.` 和 `..` 的目录被认为是空目录，这两个目录项通常不能删除
3. `opendir`，目录内容可被读取。例如，未列出目录中的全部文件，程序必须先打开该目录，然后读其中全部文件的文件名。与打开和读文件相同，在读目录前，必须先打开文件。
4. `closedir`，读目录结束后，应该关闭目录用于释放内部表空间。
5. `readdir`，系统调用 readdir 返回打开目录的下一个目录项。以前也采用 read 系统调用来读取目录，但是这种方法有一个缺点：程序员必须了解和处理目录的内部结构。相反，不论采用哪一种目录结构，readdir 总是以标准格式返回一个目录项。
6. `rename`，在很多方面目录和文件都相似。文件可以更换名称，目录也可以。
7. `link`，链接技术允许在多个目录中出现同一个文件。这个系统调用指定一个存在的文件和一个路径名，并建立从该文件到路径所指名字的链接。这样，可以在多个目录中出现同一个文件。有时也被称为 `硬链接(hard link)`。
8. `unlink`，删除目录项。如果被解除链接的文件只出现在一个目录中，则将它从文件中删除。如果它出现在多个目录中，则只删除指定路径名的链接，依然保留其他路径名的链接。在 UNIX 中，用于删除文件的系统调用就是 unlink。

文件系统的实现

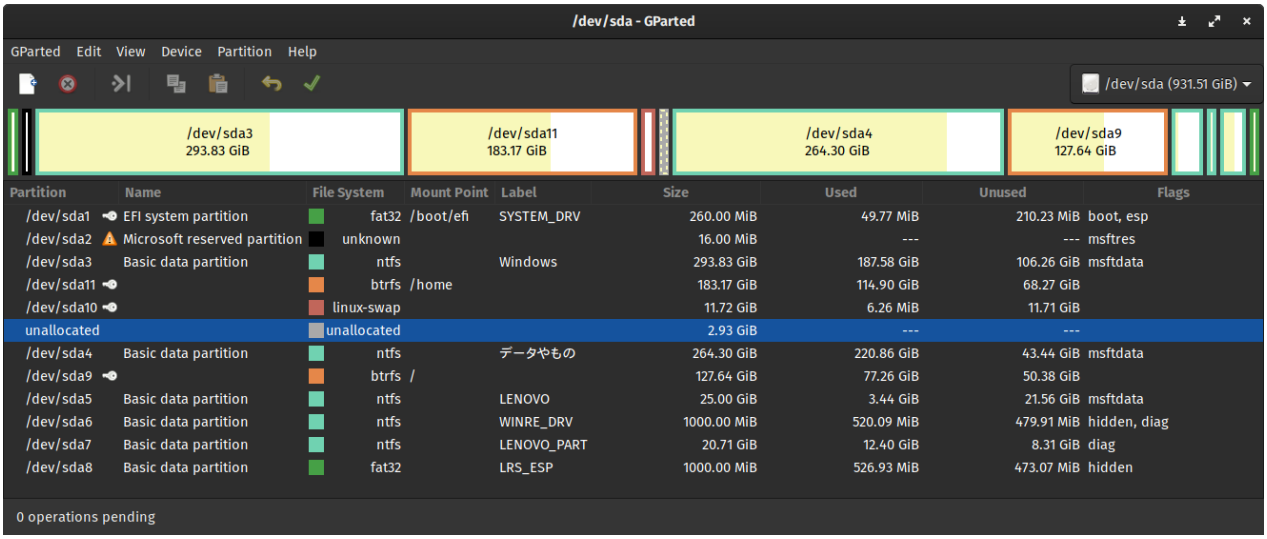
在对文件有了基本认识之后，现在是时候把目光转移到文件系统的 `实现` 上了。之前用户关心的一直都是文件是怎样命名的、可以进行哪些操作、目录树是什么，如何找到正确的文件路径等问题。而设计人员关心的是文件和目录是怎样存储的、磁盘空间是如何管理的、如何使文件系统得以流畅运行的问题，下面我们就来一起讨论一下这些问题。

文件系统布局

文件系统存储在 `磁盘` 中。大部分的磁盘能够划分出一到多个分区，叫做 `磁盘分区(disk partitioning)` 或者是 `磁盘分片(disk slicing)`。每个分区都有独立的文件系统，每块分区的文件系统可以不同。磁盘的 0 号分区称为 `主引导记录(Master Boot Record, MBR)`，用来 `引导(boot)` 计算机。在 MBR 的结尾是 `分区表(partition table)`。每个分区表给出每个分区由开始到结束的地址。系统管理员使用一个称为分区编辑器的程序来创建，调整大小，删除和操作分区。这种方式的一个缺点是很难适当调整分区的大小，导致一个分区具有很多可用空间，而另一个分区几乎完全被分配。

> MBR 可以用在 DOS 、Microsoft Windows 和 Linux 操作系统中。从 2010 年代中期开始，大多数新计算机都改用 GUID 分区表（GPT）分区方案。

下面是一个用 `GParted` 进行分区的磁盘，表中的分区都被认为是 `活动的(active)` 。



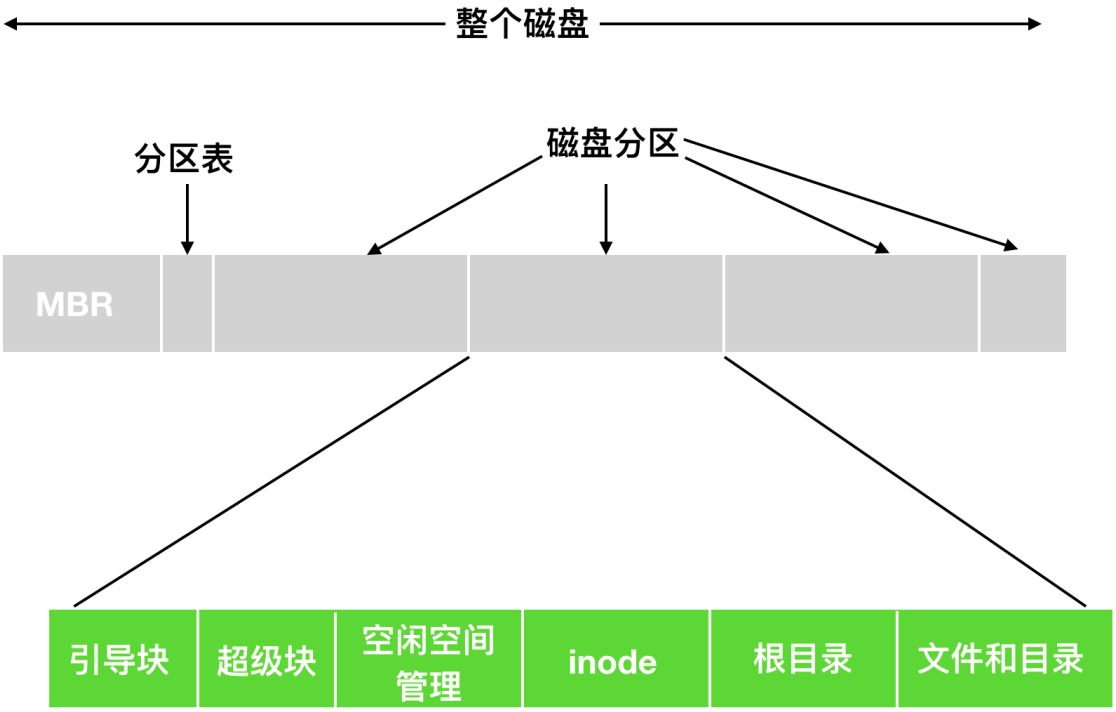
当计算机开始引 boot 时，BIOS 读入并执行 MBR。

引导块

MBR 做的第一件事就是 `确定活动分区` ，读入它的第一个块，称为 `引导块(boot block)` 并执行。引导块中的程序将加载分区中的操作系统。为了一致性，每个分区都会从引导块开始，即使引导块不包含操作系统。引导块占据文件系统的前 4096 个字节，从磁盘上的字节偏移量 0 开始。引导块可用于启动操作系统。

> 在计算机中，引导就是启动计算机的过程，它可以通过硬件（例如按下电源按钮）或者软件命令的方式来启动。开机后，电脑的 CPU 还不能执行指令，因为此时没有软件在主存中，所以一些软件必须先被加载到内存中，然后才能让 CPU 开始执行。也就是计算机开机后，首先会进行软件的装载过程。
>
> 重启电脑的过程称为 `重新引导(rebooting)` ，从休眠或睡眠状态返回计算机的过程不涉及启动。

除了从引导块开始之外，磁盘分区的布局是随着文件系统的不同而变化的。通常文件系统会包含一些属性，如下



文件系统布局

超级块

紧跟在引导块后面的是 `超级块(Superblock)`，超级块 的大小为 4096 字节，从磁盘上的字节偏移 4096 开始。超级块包含文件系统的所有关键参数

- 文件系统的大小
- 文件系统中的数据块数
- 指示文件系统状态的标志
- 分配组大小

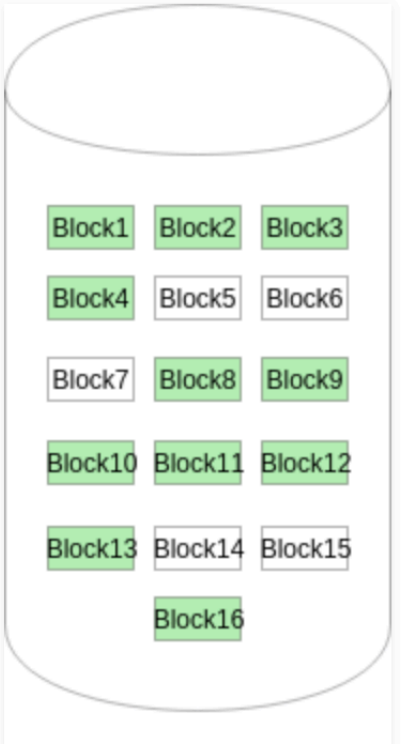
在计算机启动或者文件系统首次使用时，超级块会被读入内存。

空闲空间块

接着是文件系统中 `空闲块` 的信息，例如，可以用位图或者指针列表的形式给出。

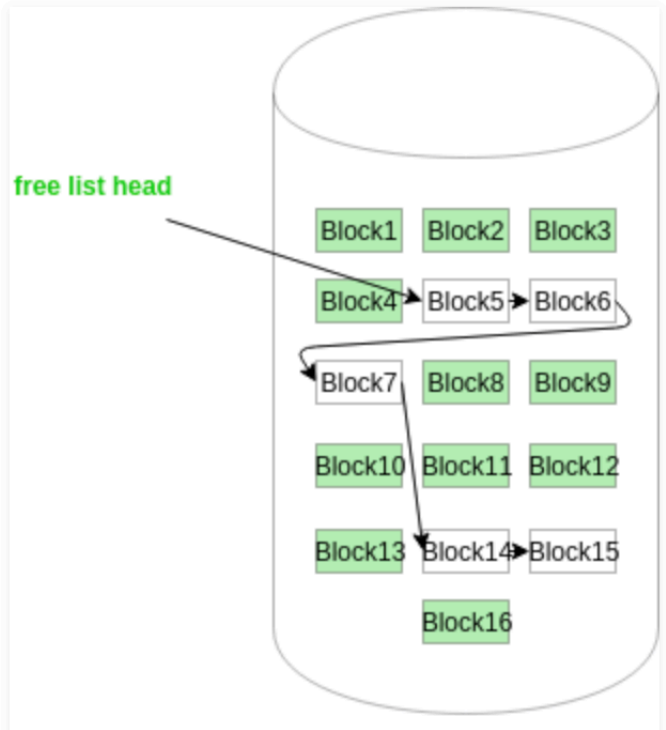
BitMap 位图或者 Bit vector 位向量

位图或位向量是一系列位或位的集合，其中每个位对应一个磁盘块，该位可以采用两个值：0和1，0表示已分配该块，而1表示一个空闲块。下图中的磁盘上给定的磁盘块实例（分配了绿色块）可以用16位的位图表示为：0000111000000110。



使用链表进行管理

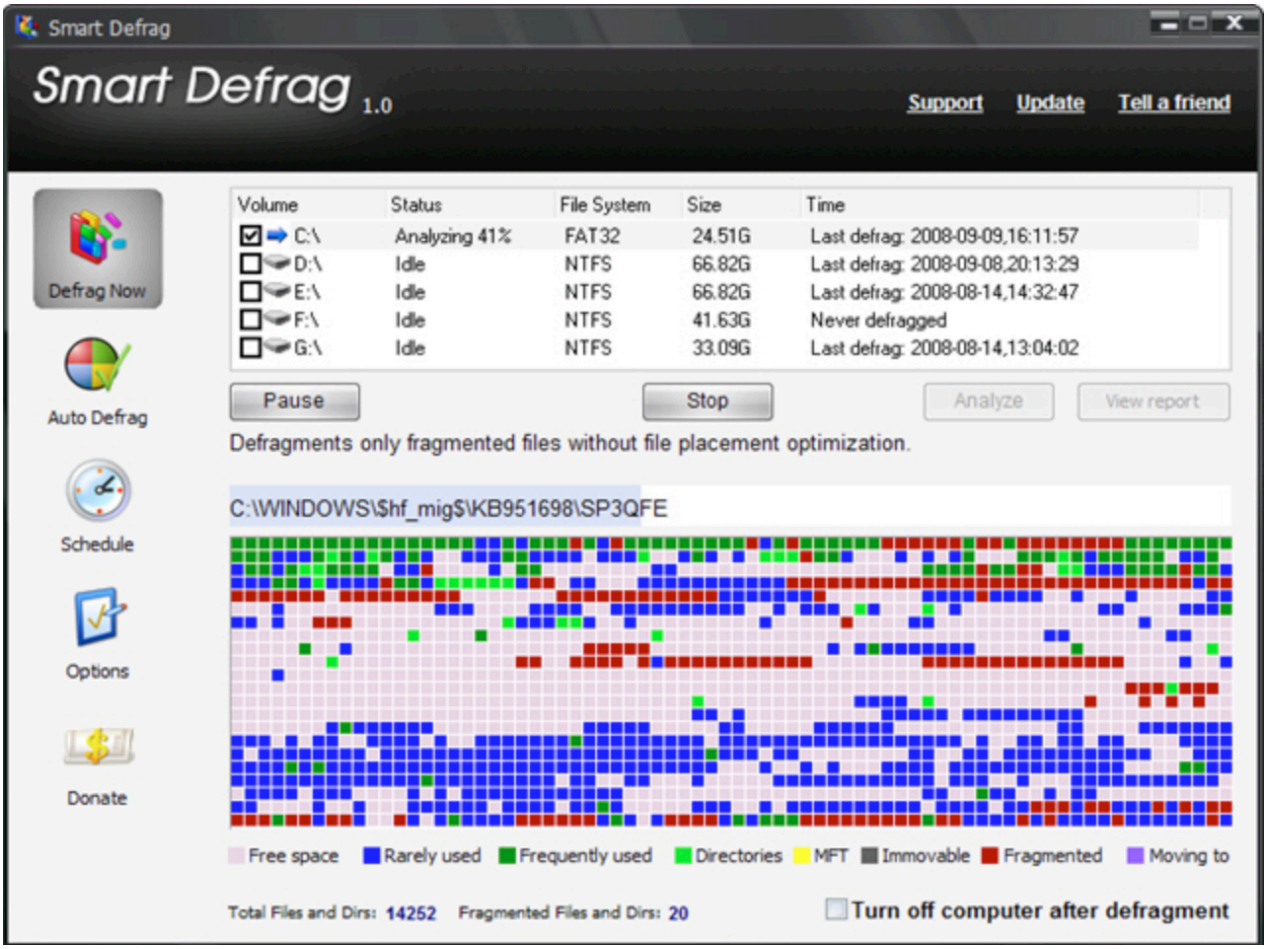
在这种方法中，空闲磁盘块链接在一起，即一个空闲块包含指向下一个空闲块的指针。第一个磁盘块的块号存储在磁盘上的单独位置，也缓存在内存中。



碎片

这里不得不提一个叫做 `碎片(fragment)` 的概念，也称为片段。一般零散的单个数据通常称为片段。磁盘块可以进一步分为固定大小的分配单元，片段只是在驱动器上彼此不相邻的文件片段。如果你不理解这个概念就给你举个例子。比如你用 Windows 电脑创建了一个文件，你会发现这个文件可以存储在任何地方，比如存在桌面上，存在磁盘中的文件夹中或者其他地方。你可以打开文件，编辑文件，删除

文件等等。你可能以为这些都在一个地方发生，但是实际上并不是，你的硬盘驱动器可能会将文件中的一部分存储在一个区域内，另一部分存储在另外一个区域，在你打开文件时，硬盘驱动器会迅速的将文件的所有部分汇总在一起，以便其他计算机系统可以使用它。



inode

然后在后面是一个 `inode(index node)`，也称作索引节点。它是一个数组的结构，每个文件有一个 inode，inode 非常重要，它说明了文件的方方面面。每个索引节点都存储对象数据的属性和磁盘块位置

有一种简单的方法可以找到它们 `ls -lai` 命令。让我们看一下根文件系统：

```
1152921504606781440 lrwxr-xr-x    1 root  wheel     25 Mar 17 09:35 home -> /Sys
tem/Volumes/Data/home
          57999767 drwxr-xr-x    2 root  wheel     64 Dec 14 06:10 opt
          57975873 drwxr-xr-x    6 root  wheel    192 Feb 15 15:22 private
1152921500311902342 drwxr-xr-x@  63 root  wheel   2016 Feb 15 15:20 sbin
1152921500312397161 lrwxr-xr-x@   1 root  admin     11 Feb 15 15:20 tmp -> priva
te/tmp
1152921500311879711 drwxr-xr-x@  11 root  wheel    352 Feb 15 15:20 usr
1152921500312397153 lrwxr-xr-x@   1 root  admin     11 Feb 15 15:20 var -> priva
te/var
```

inode 节点主要包括了以下信息

- 模式/权限（保护）
- 所有者 ID
- 组 ID
- 文件大小

- 文件的硬链接数
- 上次访问时间
- 最后修改时间
- inode 上次修改时间

文件分为两部分，索引节点和块。一旦创建后，每种类型的块数是固定的。你不能增加分区上 inode 的数量，也不能增加磁盘块的数量。

紧跟在 inode 后面的是根目录，它存放的是文件系统目录树的根部。最后，磁盘的其他部分存放了其他所有的目录和文件。

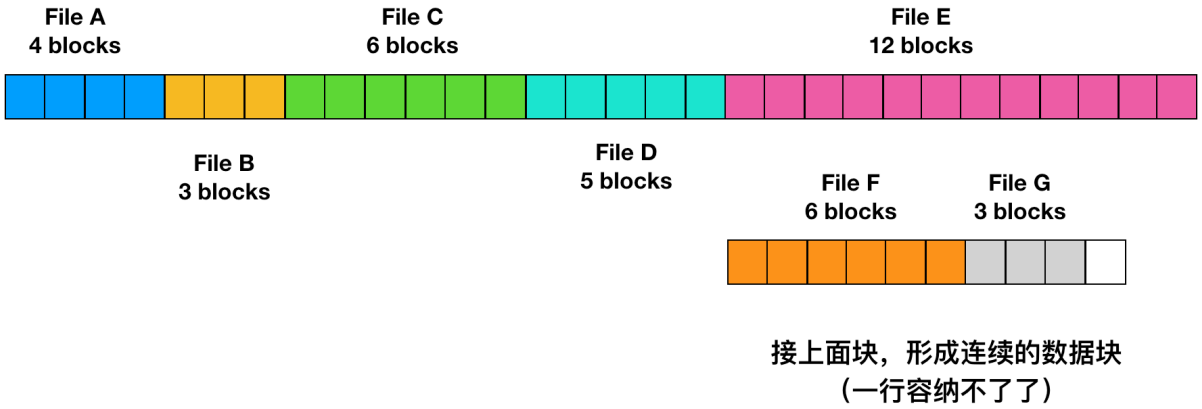
文件的实现

最重要的问题是记录各个文件分别用到了哪些磁盘块。不同的系统采用了不同的方法。下面我们会探讨一下这些方式。分配背后的主要思想是 `有效利用文件空间` 和 `快速访问文件` ，主要有三种分配方案

- 连续分配
- 链表分配
- 索引分配

连续分配

最简单的分配方案是把每个文件作为一连串连续数据块存储在磁盘上。因此，在具有 1KB 块的磁盘上，将为 50 KB 文件分配 50 个连续块。



使用连续空间存储文件

上面展示了 40 个连续的内存块。从最左侧的 0 块开始。初始状态下，还没有装载文件，因此磁盘是空的。接着，从磁盘开始处（块 0 ）处开始写入占用 4 块长度的内存 A 。然后是一个占用 6 块长度的内存 B，会直接在 A 的末尾开始写。

注意每个文件都会在新的文件块开始写，所以如果文件 A 只占用了 `3 又 1/2` 个块，那么最后一个块的部分内存会被浪费。在上面这幅图中，总共展示了 7 个文件，每个文件都会从上个文件的末尾块开始写新的文件块。

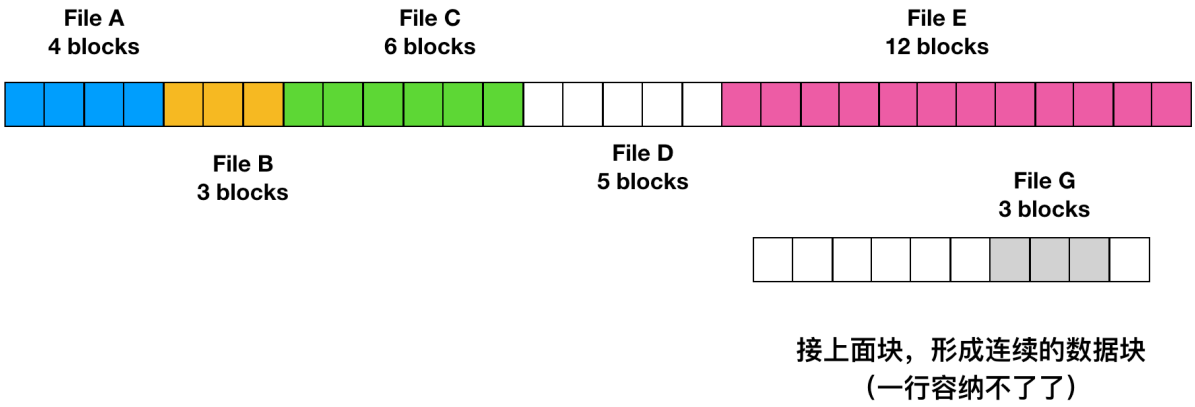
连续的磁盘空间分配有两个优点。

- 第一，连续文件存储实现起来比较简单，只需要记住两个数字就可以：一个是第一个块的文件地址和文件的块数量。给定第一个块的编号，可以通过简单的加法找到任何其他块的编号。
- 第二点是读取性能比较强，可以通过一次操作从文件中读取整个文件。只需要一次寻找第一个块。

后面就不再需要寻道时间和旋转延迟，所以数据会以全带宽进入磁盘。

因此，连续的空间分配具有 `实现简单` 、 `高性能` 的特点。

不幸的是，连续空间分配也有很明显的不足。随着时间的推移，磁盘会变得很零碎。下图解释了这种现象



接上面块，形成连续的数据块
（一行容纳不了了）

这里有两个文件 D 和 F 被删除了。当删除一个文件时，此文件所占用的块也随之释放，就会在磁盘空间中留下一些空闲块。磁盘并不会在这个位置挤压掉空闲块，因为这会复制空闲块之后的所有文件，可能会有上百万的块，这个量级就太大了。

刚开始的时候，这个碎片不是问题，因为每个新文件都会在之前文件的结尾处进行写入。然而，磁盘最终会被填满，**因此要么压缩磁盘、要么重新使用空闲块的空间**。压缩磁盘的开销太大，因此不可行；后者会维护一个空闲列表，这个是可行的。但是这种情况又存在一个问题，为空闲块匹配合适大小的文件，需要知道该文件的 `最终大小` 。

想象一下这种设计的结果会是怎样的。用户启动 word 进程创建文档。应用程序首先会询问最终创建的文档会有多大。这个问题必须回答，否则应用程序就不会继续执行。如果空闲块的大小要比文件的大小小，程序就会终止。因为所使用的磁盘空间已经满了。那么现实生活中，有没有使用连续分配内存的介质出现呢?

`CD-ROM` 就广泛的使用了连续分配方式。

> `CD-ROM (Compact Disc Read-Only Memory)` 即只读光盘，也称作只读存储器。是一种在电脑上使用的光碟。这种光碟只能写入数据一次，信息将永久保存在光碟上，使用时通过光碟驱动器读出信息。

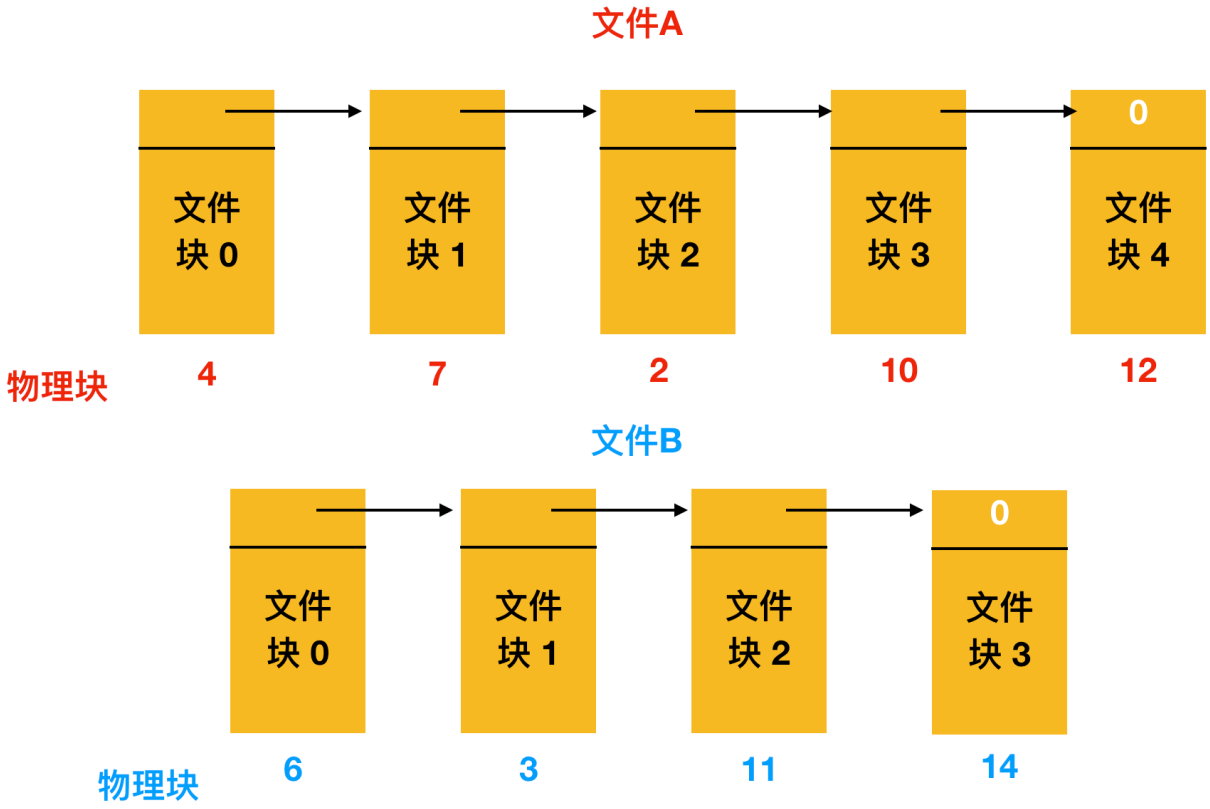


然而 DVD 的情况会更加复杂一些。原则上，一个 `90分钟` 的电影能够被编码成一个独立的、大约 4.5 GB 的文件。但是文件系统所使用的 `UDF(Universal Disk Format)` 格式，使用一个 30 位的数来代表文件长度，从而把文件大小限制在 1 GB。所以，DVD 电影一般存储在 3、4个连续的 1 GB 空间内。这些构成单个电影中的文件块称为 `扩展区(extends)` 。

就像我们反复提到的，**历史总是惊人的相似**，许多年前，连续分配由于其 `简单` 和 `高性能` 被实际使用在磁盘文件系统中。后来由于用户不希望在创建文件时指定文件的大小，于是放弃了这种想法。但是随着 CD-ROM 、DVD、蓝光光盘等光学介质的出现，连续分配又流行起来。从而得出结论， `技术永远没有过时性` ，现在看似很老的技术，在未来某个阶段可能又会流行起来。

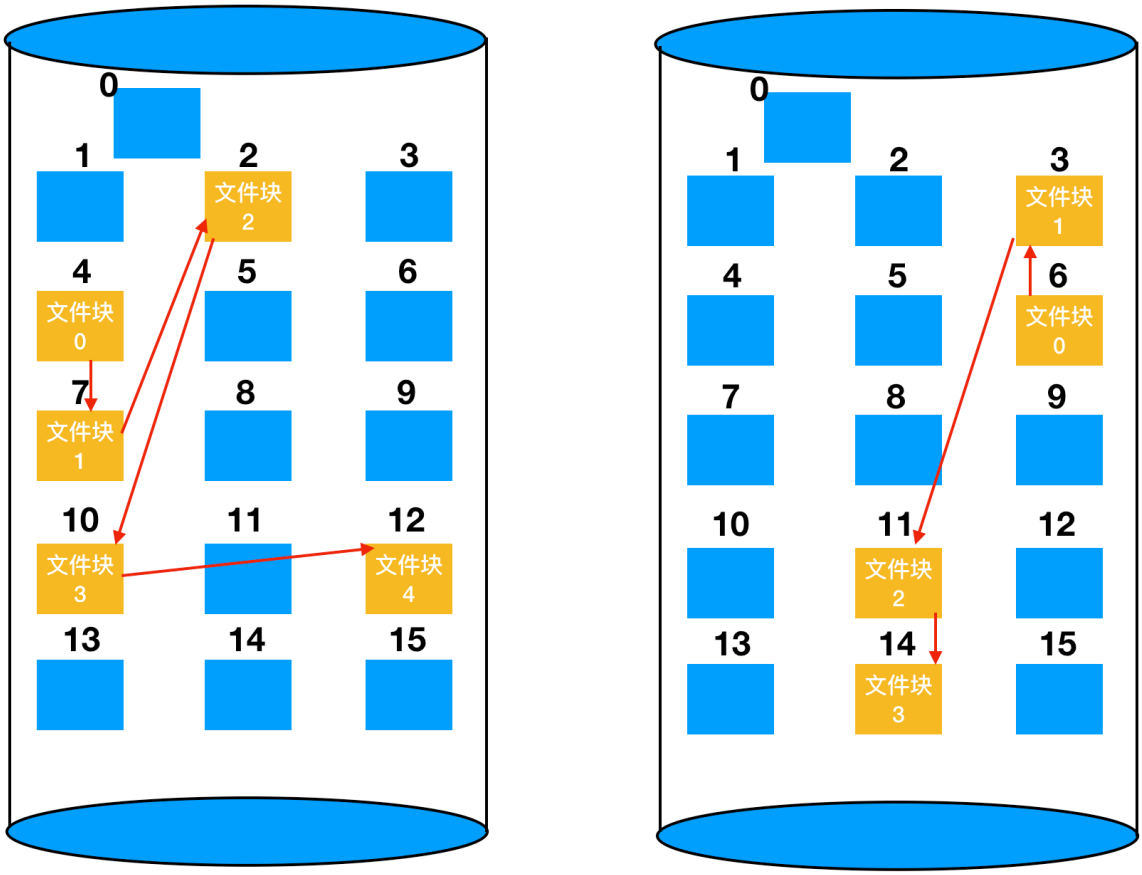
链表分配

第二种存储文件的方式是为每个文件构造磁盘块链表，每个文件都是磁盘块的链接列表，就像下面所示



以磁盘块的链表形式存储文件

每个块的第一个字作为指向下一块的指针，块的其他部分存放数据。如果上面这张图你看的不是很清楚的话，可以看看整个的链表分配方案



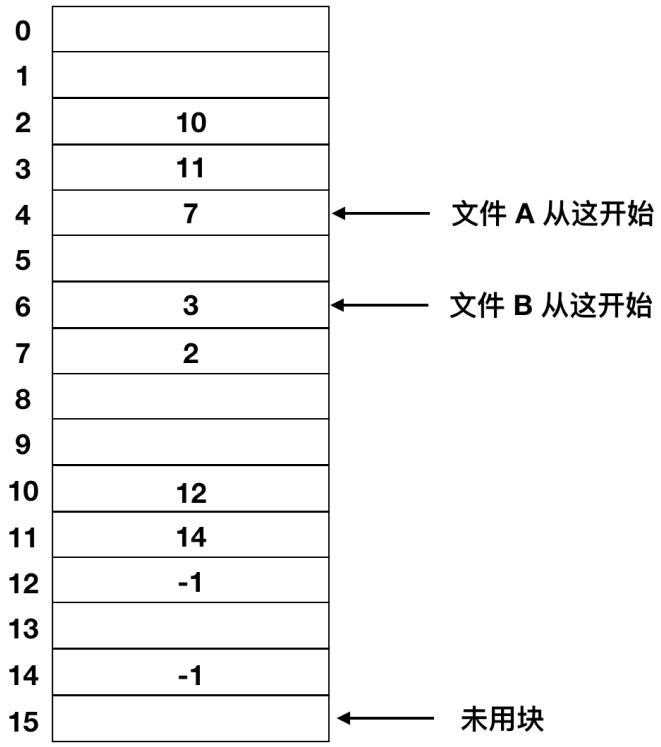
与连续分配方案不同，这一方法可以充分利用每个磁盘块。除了最后一个磁盘块外，不会因为磁盘碎片而浪费存储空间。同样，在目录项中，只要存储了第一个文件块，那么其他文件块也能够被找到。

另一方面，在链表的分配方案中，尽管顺序读取非常方便，但是随机访问却很困难（这也是数组和链表数据结构的一大区别）。

还有一个问题是，由于指针会占用一些字节，每个磁盘块实际存储数据的字节数并不再是 2 的整数次幂。虽然这个问题并不会很严重，但是这种方式降低了程序运行效率。许多程序都是以长度为 2 的整数次幂来读写磁盘，由于每个块的前几个字节被指针所使用，所以要读出一个完成的块大小信息，就需要当前块的信息和下一块的信息拼凑而成，因此就引发了查找和拼接的开销。

使用内存表进行链表分配

由于连续分配和链表分配都有其不可忽视的缺点。所以提出了使用内存中的表来解决分配问题。取出每个磁盘块的指针字，把它们放在内存的一个表中，就可以解决上述链表的两个不足之处。下面是一个例子



上图表示了链表形成的磁盘块的内容。这两个图中都有两个文件，文件 A 依次使用了磁盘块地址 **4**、**7**、 **2**、 **10**、 **12**，文件 B 使用了**6**、**3**、**11 和 14**。也就是说，文件 A 从地址 4 处开始，顺着链表走就能找到文件 A 的全部磁盘块。同样，从第 6 块开始，顺着链走到最后，也能够找到文件 B 的全部磁盘块。你会发现，这两个链表都以不属于有效磁盘编号的特殊标记（-1）结束。内存中的这种表格称为 `文件分配表(File Application Table,FAT)` 。

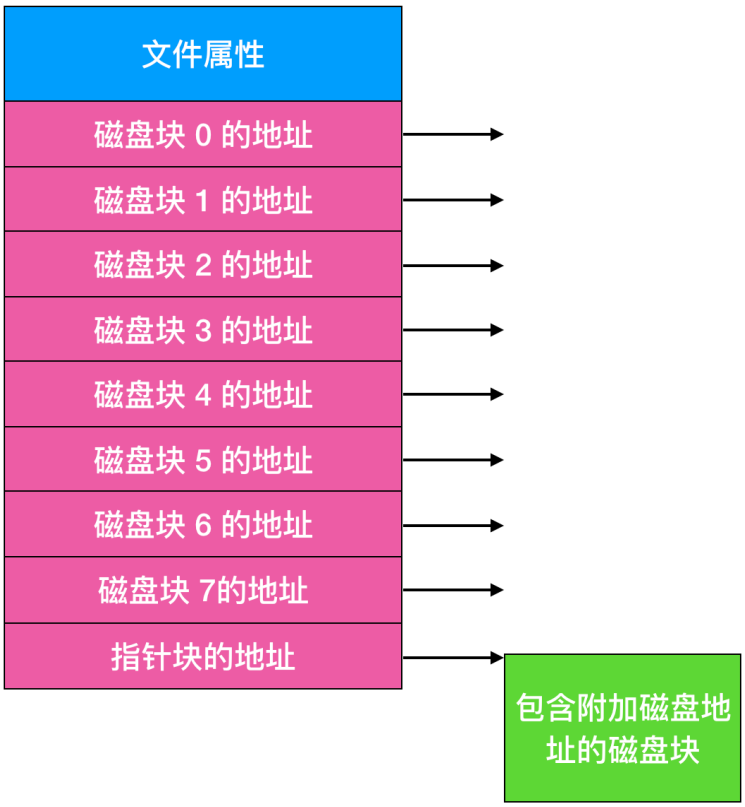
使用这种组织方式，整个块都可以存放数据。进而，随机访问也容易很多。虽然仍要顺着链在内存中查找给定的偏移量，但是整个链都存放在内存中，所以不需要任何磁盘引用。与前面的方法相同，不管文件有多大，在目录项中只需记录一个整数（起始块号），按照它就可以找到文件的全部块。

这种方式存在缺点，那就是**必须要把整个链表放在内存中**。对于 1TB 的磁盘和 1KB 的大小的块，那么这张表需要有 10 亿项。。。每一项对应于这 10 亿个磁盘块中的一块。每项至少 3 个字节，为了提高查找速度，有时需要 4 个字节。根据系统对空间或时间的优化方案，这张表要占用 3GB 或 2.4GB 的内存。FAT 的管理方式不能较好地扩展并应用于大型磁盘中。而这正是最初 MS-DOS 文件比较实用，并仍被各个 Windows 版本所安全支持。

inode

最后一个记录各个文件分别包含哪些磁盘块的方法是给每个文件赋予一个称为 `inode(索引节点)` 的数据结构，每个文件都与一个 `inode` 进行关联，inode 由整数进行标识。

下面是一个简单例子的描述。



给出 inode 的长度，就能够找到文件中的所有块。

相对于在内存中使用表的方式而言，这种机制具有很大的优势。即只有在文件打开时，其 inode 才会在内存中。如果每个 inode 需要 n 个字节，最多 k 个文件同时打开，那么 inode 占有总共打开的文件是 kn 字节。仅需预留这么多空间。

这个数组要比我们上面描述的 `FAT(文件分配表)` 占用的空间小的多。原因是用于保存所有磁盘块的链接列表的表的大小与磁盘本身成正比。如果磁盘有 n 个块，那么这个表也需要 n 项。随着磁盘空间的变大，那么该表也随之 `线性增长` 。相反，inode 需要节点中的数组，其大小和可能需要打开的最大文件个数成正比。它与磁盘是 100GB、4000GB 还是 10000GB 无关。

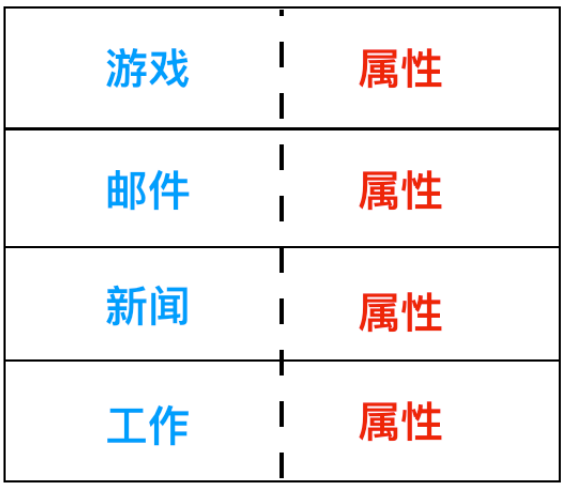
inode 的一个问题是如果每个节点都会有固定大小的磁盘地址，那么文件增长到所能允许的最大容量外会发生什么？一个解决方案是**最后一个磁盘地址不指向数据块**，而是**指向一个包含额外磁盘块地址的地址**，如上图所示。一个更高级的解决方案是：有两个或者更多包含磁盘地址的块，或者指向其他存放地址的磁盘块的磁盘块。Windows 的 NTFS 文件系统采用了相似的方法，所不同的仅仅是大的 inode 也可以表示小的文件。

> NTFS 的全称是 `New Technology File System` ，是微软公司开发的专用系统文件，NTFS 取代 FAT(文件分配表) 和 `HPFS(高性能文件系统)` ，并在此基础上进一步改进。例如增强对元数据的支持，使用更高级的数据结构以提升性能、可靠性和磁盘空间利用率等。

目录的实现

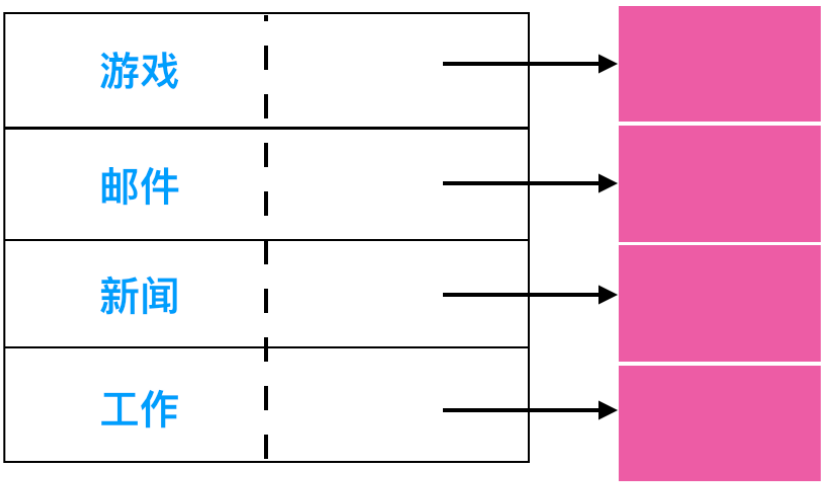
文件只有打开后才能够被读取。在文件打开后，操作系统会使用用户提供的路径名来定位磁盘中的目录。目录项提供了查找文件磁盘块所需要的信息。根据系统的不同，提供的信息也不同，可能提供的信息是整个文件的磁盘地址，或者是第一个块的数量（两个链表方案）或 inode的数量。不过不管用那种情况，目录系统的主要功能就是 **将文件的 ASCII 码的名称映射到定位数据所需的信息上**。

与此关系密切的问题是属性应该存放在哪里。每个文件系统包含不同的文件属性，例如文件的所有者和创建时间，需要存储的位置。一种显而易见的方法是直接**把文件属性存放在目录中**。有一些系统恰好是这么做的，如下。



在这种简单的设计中，目录有一个固定大小的目录项列表，每个文件对应一项，其中包含一个固定长度的文件名，文件属性的结构体以及用以说明磁盘块位置的一个或多个磁盘地址。

对于采用 inode 的系统，会把 inode 存储在属性中而不是目录项中。在这种情况下，目录项会更短：仅仅只有文件名称和 inode 数量。这种方式如下所示



到目前为止，我们已经假设文件具有较短的、固定长度的名字。在 MS-DOS 中，具有 1 - 8 个字符的基本名称和 1 - 3 个字符的可拓展名称。在 UNIX 版本 7 中，文件有 1 - 14 个字符，包括任何拓展。然而，几乎所有的现代操作系统都支持可变长度的扩展名。这是如何实现的呢?

最简单的方式是给予文件名一个长度限制，比如 255 个字符，然后使用上图中的设计，并为每个文件名保留 255 个字符空间。这种处理很简单，但是浪费了大量的目录空间，因为只有很少的文件会有那么长的文件名称。所以，需要一种其他的结构来处理。

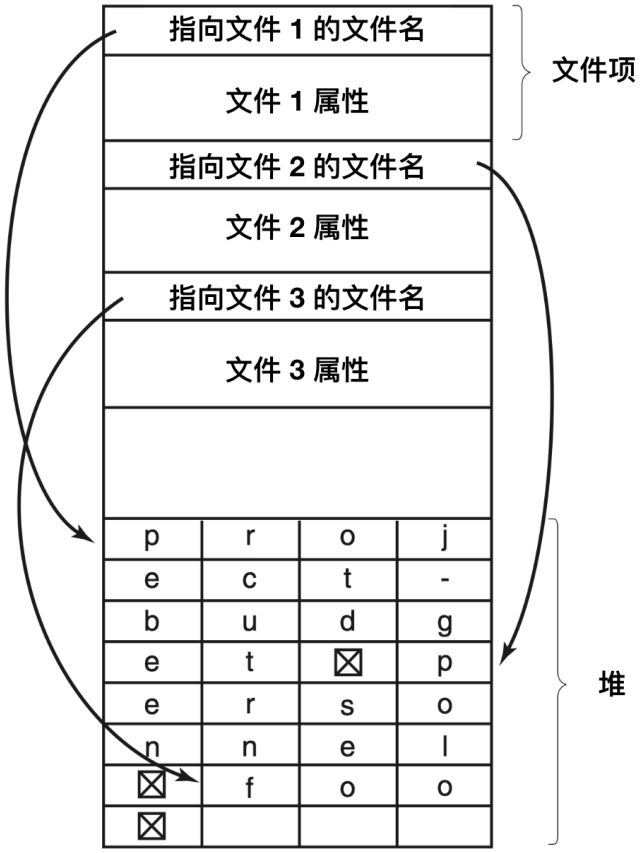
一种可选择的方式是放弃所有目录项大小相同的想法。在这种方法中，每个目录项都包含一个固定部分，这个固定部分通常以目录项的长度开始，后面是固定格式的数据，通常包括**所有者、创建时间、保护信息和其他属性**。这个固定长度的头的后面是一个任意长度的实际文件名，如下图所示



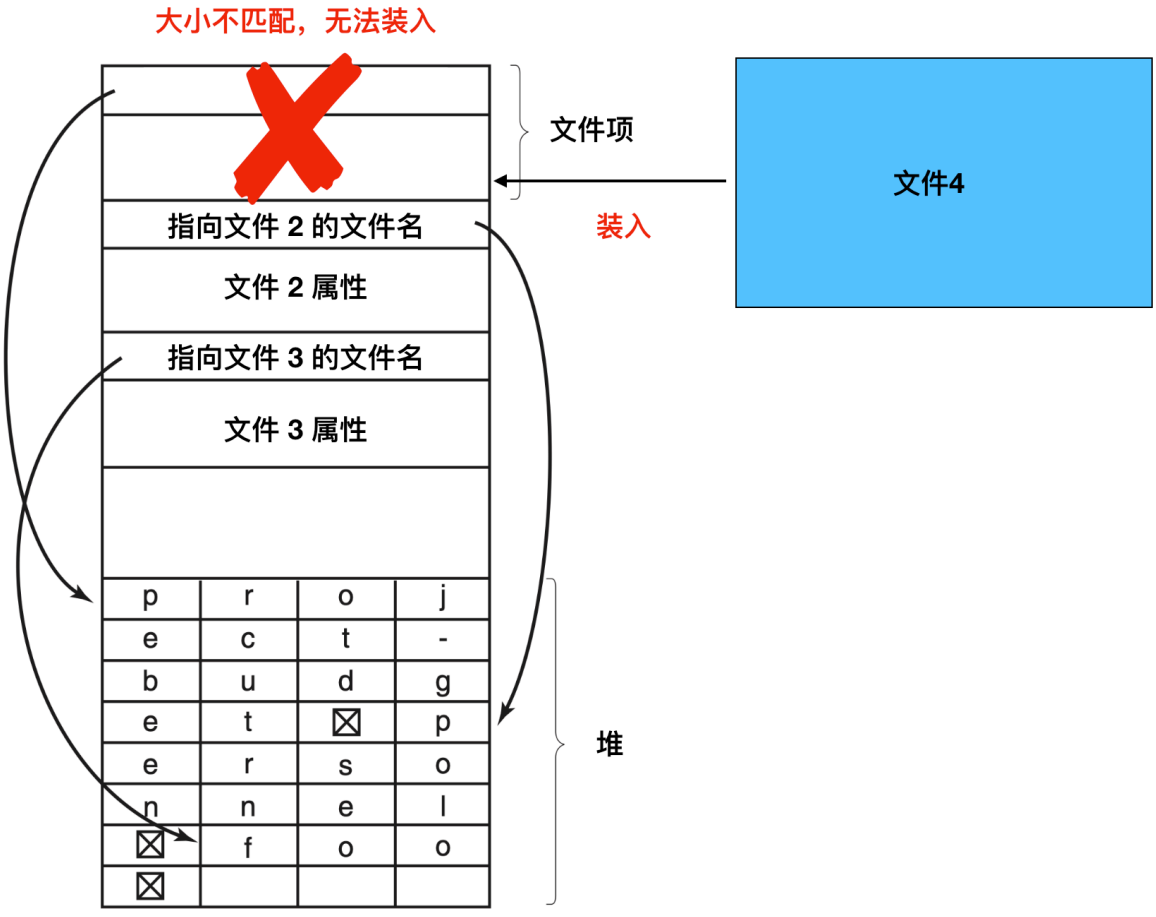
上图是 SPARC 机器使用正序放置。

> 处理机中的一串字符存放的顺序有 `正序(big-endian)` 和 `逆序(little-endian)` 之分。正序存放的就是高字节在前低字节在后，而逆序存放的就是低字节在前高字节在后。

这个例子中，有三个文件，分别是 `project-budget` 、 `personnel` 和 `foo` 。每个文件名以一个特殊字符（通常是 0 ）结束，用矩形中的叉进行表示。为了使每个目录项从字的边界开始，每个文件名被填充成整数个字，如下图所示



这个方法的缺点是当文件被移除后，就会留下一块固定长度的空间，而新添加进来的文件大小不一定和空闲空间大小一致。

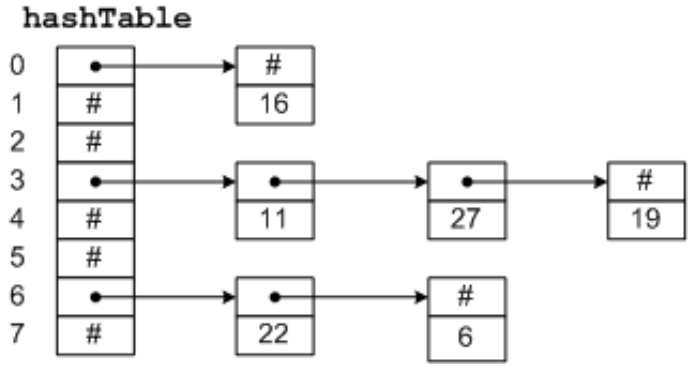


这个问题与我们上面探讨的连续磁盘文件的问题是一样的，由于整个目录在内存中，所以只有对目录进行 `紧凑拼接` 操作才可节省空间。另一个问题是，一个目录项可能会分布在多个页上，**在读取文件名时可能发生缺页中断**。

处理可变长度文件名字的另外一种方法是，使目录项自身具有固定长度，而将文件名放在目录末尾的堆栈中。如上图所示的这种方式。这种方法的优点是当目录项被移除后，下一个文件将能够正常匹配移除文件的空间。当然，必须要对 `堆` 进行管理，因为在处理文件名的时候也会发生缺页异常。

到目前为止的所有设计中，在需要查找文件名时，所有的方案都是线性的从头到尾对目录进行搜索。对于特别长的目录，线性搜索的效率很低。提高文件检索效率的一种方式是在每个目录上使用 `哈希表 (hash table)`，也叫做散列表。我们假设表的大小为 n，在输入文件名时，文件名被散列在 0 和 n - 1 之间，例如，它被 n 除，并取余数。或者对构成文件名字的字求和或类似某种方法。

无论采用哪种方式，**在添加一个文件时都要对与散列值相对 应的散列表进行检查**。如果没有使用过，就会将一个指向目录项的指针指向这里。文件目录项紧跟着哈希表后面。如果已经使用过，就会构造一个链表（这种构造方式是不是和 HashMap 使用的数据结构一样？），链表的表头指针存放在表项中，并通过哈希值将所有的表项相连。



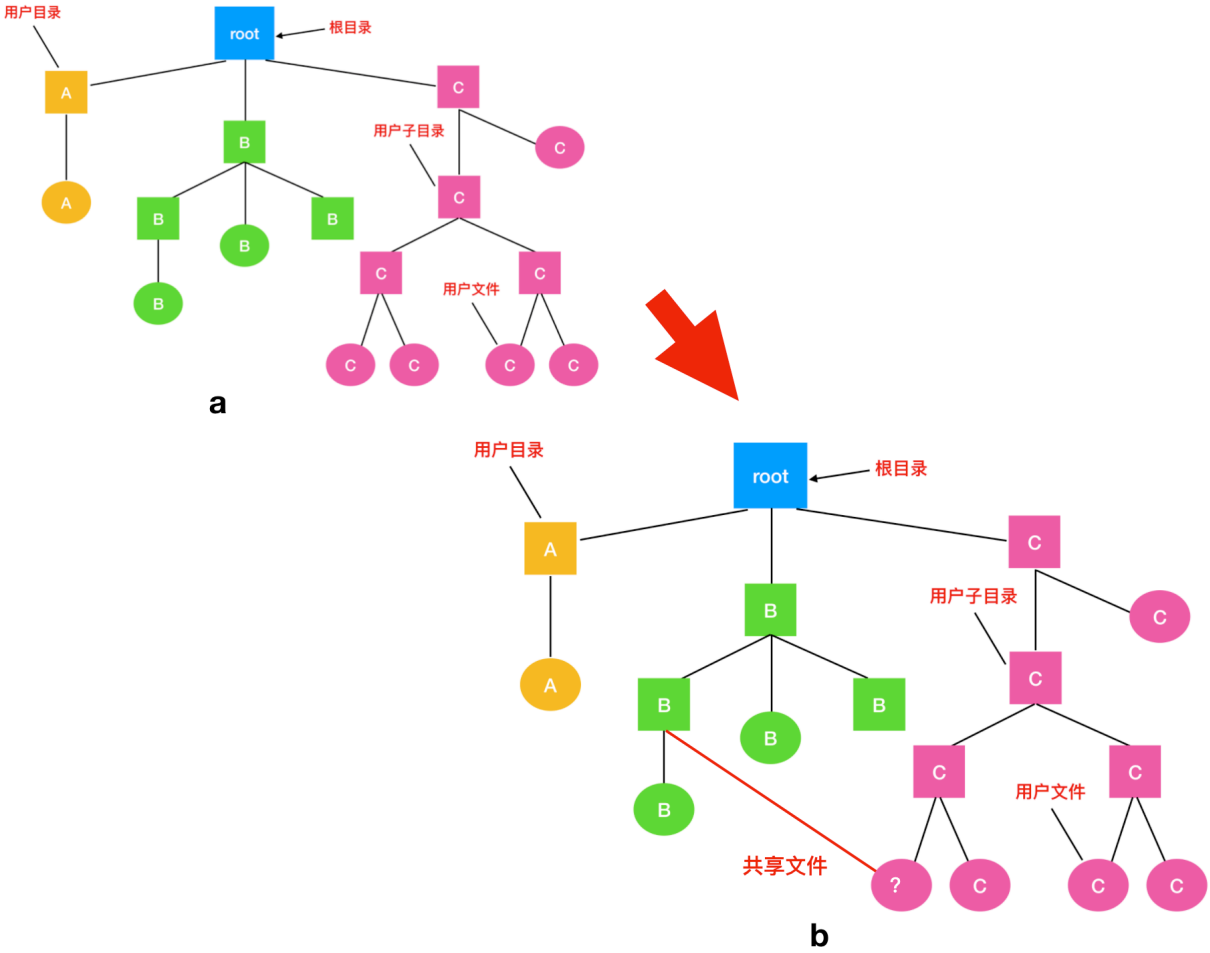
查找文件的过程和添加类似，首先对文件名进行哈希处理，在哈希表中查找是否有这个哈希值，如果有的话，就检查这条链上所有的哈希项，查看文件名是否存在。如果哈希不在链上，那么文件就不在目录中。

使用哈希表的优势是 `查找非常迅速`，缺点是 `管理起来非常复杂`。只有在系统中会有成千上万个目录项存在时，才会考虑使用散列表作为解决方案。

另外一种在大量目录中加快查找指令目录的方法是使用 `缓存`，缓存查找的结果。在开始查找之前，会首先检查文件名是否在缓存中。如果在缓存中，那么文件就能立刻定位。当然，只有在较少的文件下进行多次查找，缓存才会发挥最大功效。

共享文件

当多个用户在同一个项目中工作时，他们通常需要共享文件。如果这个共享文件同时出现在多个用户目录下，那么他们协同工作起来就很方便。下面的这张图我们在上面提到过，但是有一个更改的地方，就是 **C 的一个文件也出现在了 B 的目录下**。

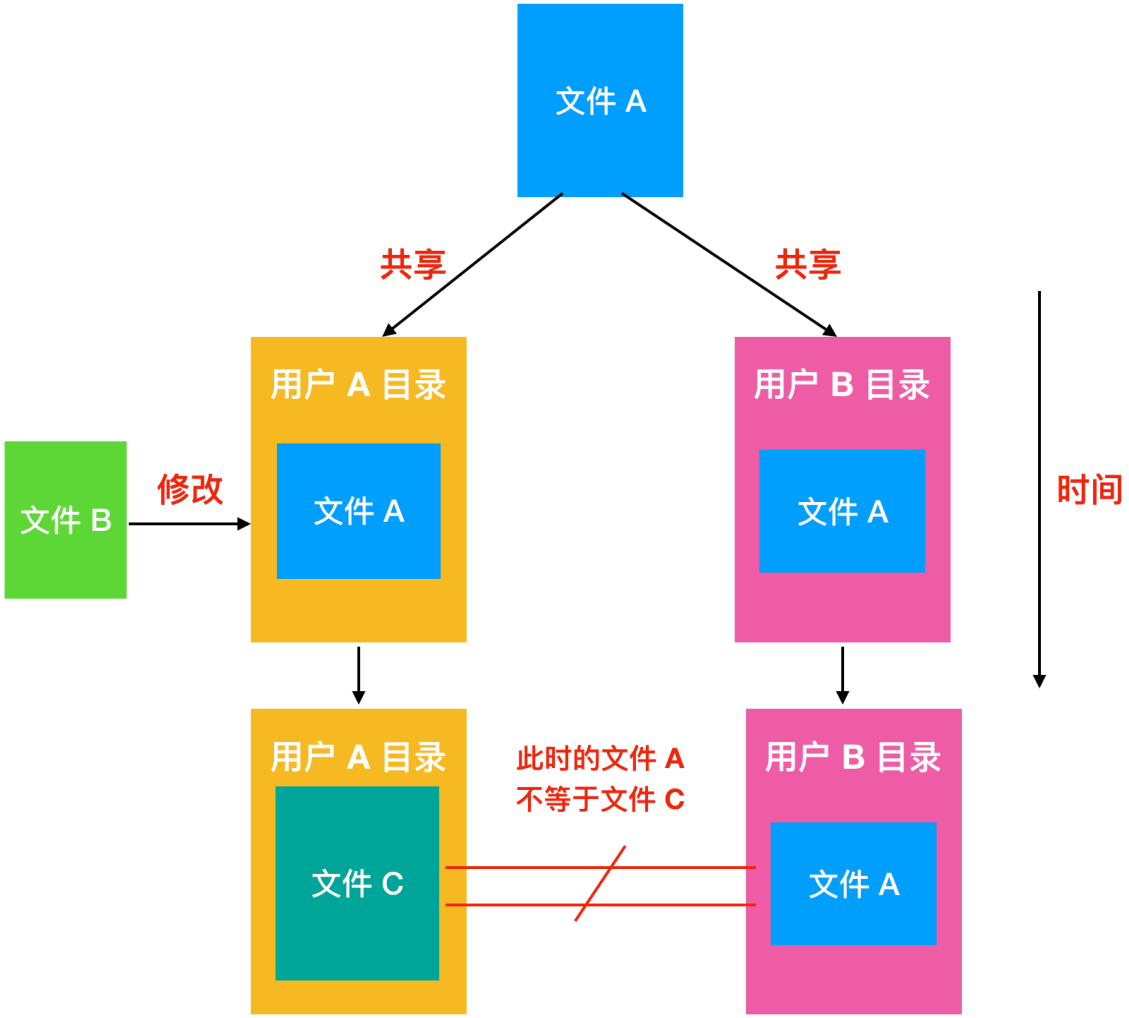


如果按照如上图的这种组织方式而言，那么 B 的目录与该共享文件的联系称为 `链接(link)` 。那么文件系统现在就是一个 `有向无环图(Directed Acyclic Graph, 简称 DAG)` ，而不是一棵树了。

> 在图论中，如果一个有向图从任意顶点出发无法经过若干条边回到该点，则这个图是一个 `有向无环图` ，我们不会在此着重探讨关于图论的东西，大家可以自行 google。

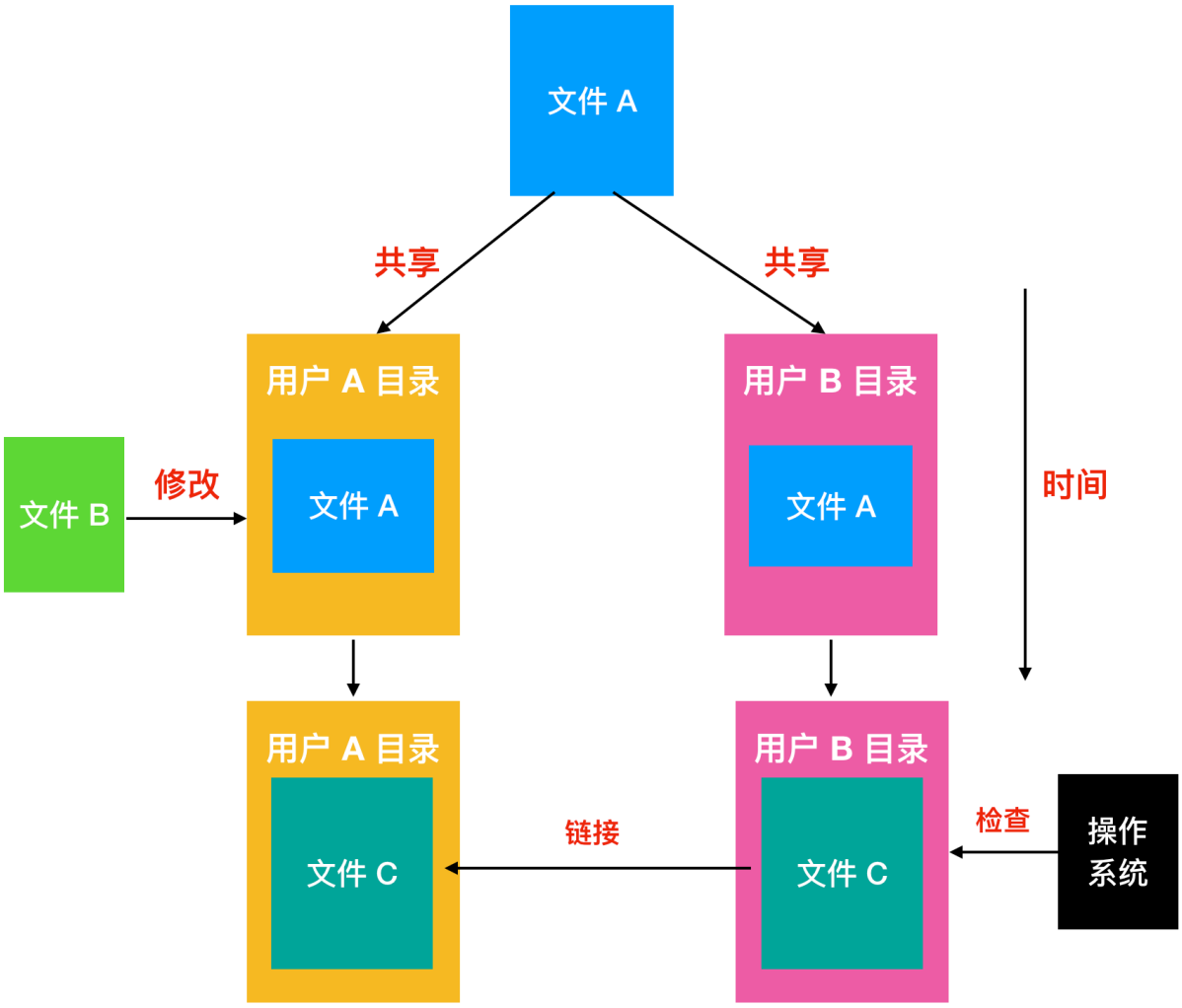
将文件系统组织成为有向无环图会使得维护复杂化，但也是必须要付出的代价。

`共享文件` 很方便，但这也会带来一些问题。如果目录中包含磁盘地址，则当链接文件时，**必须把 C 目录中的磁盘地址复制到 B 目录中**。如果 B 或者 C 随后又向文件中添加内容，则仅在执行追加的用户的目录中显示新写入的数据块。这种变更将会对其他用户不可见，从而破坏了共享的目的。

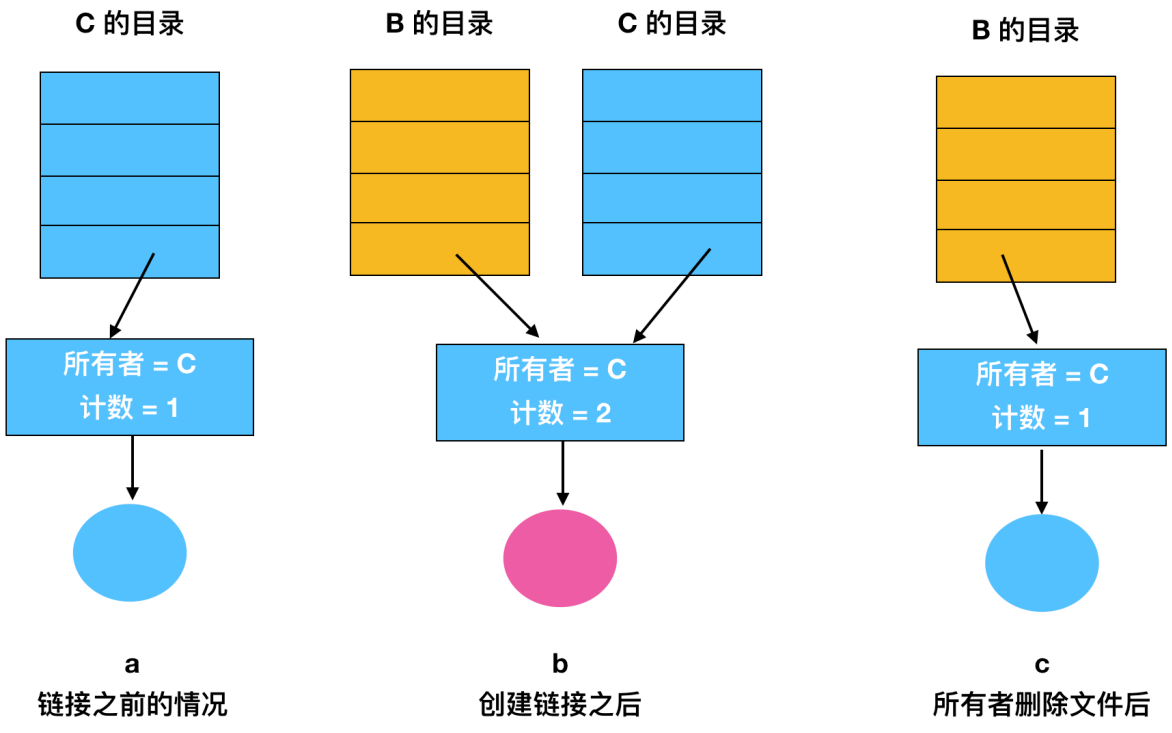


有两种方案可以解决这种问题。

- 第一种解决方案，磁盘块不列入目录中，而是会把磁盘块放在与文件本身相关联的小型数据结构中。目录将指向这个小型数据结构。这是 `UNIX` 中使用的方式（小型数据结构就是 inode）。
- 在第二种解决方案中，通过让系统建立一个类型为 `LINK` 的新文件，并把该文件放在 B 的目录下，使得 B 与 C 建立链接。新的文件中只包含了它所链接的文件的路径名。当 B 想要读取文件时，操作系统会检查 B 的目录下存在一个类型为 LINK 的文件，进而找到该链接的文件和路径名，然后再去读文件，这种方式称为 `符号链接(symbolic linking)`。



上面的每一种方法都有各自的缺点，在第一种方式中，B 链接到共享文件时，inode 记录文件的所有者为 C。**建立一个链接并不改变所有关系**，如下图所示。



第一开始的情况如图 a 所示，此时 C 的目录的所有者是 C ，当目录 B 链接到共享文件时，并不会改变 C 的所有者关系，只是把计数 + 1，所以此时 **系统知道目前有多少个目录指向这个文件**。然后 C 尝试删除这个文件，这个时候有个问题，如果 C 把文件移除并清除了 inode 的话，那么 B 会有一个目录项指向无效的节点。如果 inode 以后分配给另一个文件，则 B 的链接指向一个错误的文件。系统通过 inode 可知文件仍在被引用，但是没有办法找到该文件的全部目录项以删除它们。指向目录的指针不能存储在 inode 中，原因是有可能有无数个这样的目录。

所以我们能做的就是删除 C 的目录项，但是将 inode 保留下来，并将计数设置为 1 ，如上图 c 所示。c 表示的是只有 B 有指向该文件的目录项，而该文件的前者是 C 。如果系统进行记账操作的话，那么 C 将继续为该文件付账直到 B 决定删除它，如果是这样的话，只有到计数变为 0 的时刻，才会删除该文件。

对于 `符号链接` ，以上问题不会发生，只有真正的文件所有者才有一个指向 inode 的指针。链接到该文件上的用户只有路径名，没有指向 inode 的指针。当文件所有者删除文件时，该文件被销毁。以后若试图通过符号链接访问该文件将会失败，因为系统不能找到该文件。删除符号链接不会影响该文件。

符号链接的问题是**需要额外的开销**。必须读取包含路径的文件，然后要一个部分接一个部分地扫描路径，直到找到 inode 。这些操作也许需要很多次额外的磁盘访问。此外，每个符号链接都需要额外的 inode ，以及额外的一个磁盘块用于存储路径，虽然如果路径名很短，作为一种优化，系统可以将它存储在 inode 中。符号链接有一个优势，即只要**简单地提供一个机器的网络地址以及文件在该机器上驻留的路径**，就可以连接全球任何地方机器上的文件。

还有另一个由链接带来的问题，在符号链接和其他方式中都存在。如果允许链接，文件有两个或多个路径。查找一指定目录及其子目录下的全部文件的程序将多次定位到被链接的文件。例如，一个将某一目录及其子目录下的文件转存到磁带上的程序有可能多次复制一个被链接的文件。进而，如果接着把磁带读入另一台机器，除非转出程序具有智能，否则被链接的文件将被两次复制到磁盘上，而不是只是被链接起来。

日志结构文件系统

技术的改变会给当前的文件系统带来压力。这种情况下，CPU 会变得越来越快，磁盘会变得越来越大并且越来越便宜（但不会越来越快）。内存容量也是以指数级增长。但是磁盘的寻道时间（除了固态盘，因为固态盘没有寻道时间）并没有获得提高。

这些因素结合起来意味着许多系统文件中出现性能瓶颈。为此， `Berkeley` 设计了一种全新的文件系统，试图缓解这个问题，这个文件系统就是 `日志结构文件系统(Log-structured File System, LFS)` 。

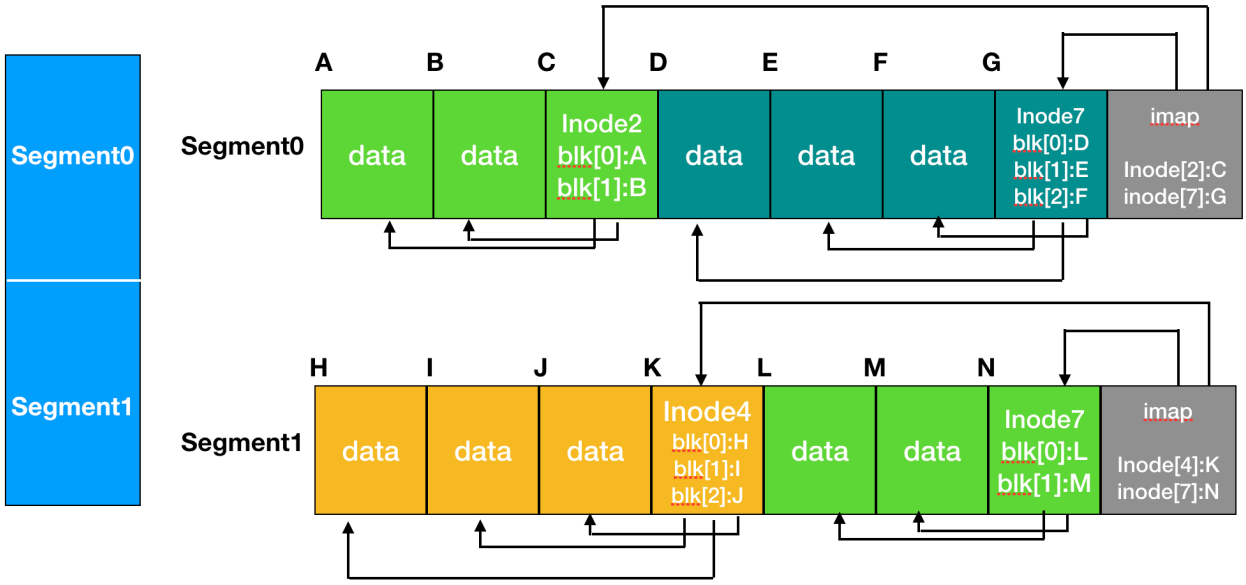
日志结构文件系统由 `Rosenblum` 和 `Ousterhout` 于90年代初引入，旨在解决以下问题。

- 不断增长的系统内存
- 顺序 I/O 性能胜过随机 I/O 性能
- 现有低效率的文件系统
- 文件系统不支持 RAID（虚拟化）

另一方面，当时的文件系统不论是 UNIX 还是 FFS，都有大量的随机读写（在 FFS 中创建一个新文件至少需要5次随机写），因此成为整个系统的性能瓶颈。同时因为 `Page cache` 的存在，作者认为随机读不是主要问题：随着越来越大的内存，大部分的读操作都能被 cache，因此 LFS 主要要解决的是减少对硬盘的随机写操作。

在这种设计中，inode 甚至具有与 UNIX 中相同的结构，但是现在它们分散在整个日志中，而不是位于磁盘上的固定位置。所以，inode 很定位。为了能够找到 inode ，维护了一个由 inode 索引的 `inode map(inode 映射)` 。表项 i 指向磁盘中的第 i 个 inode 。这个映射保存在磁盘中，但是也保存在缓存中，因此，使用最频繁的部分大部分时间都在内存中。

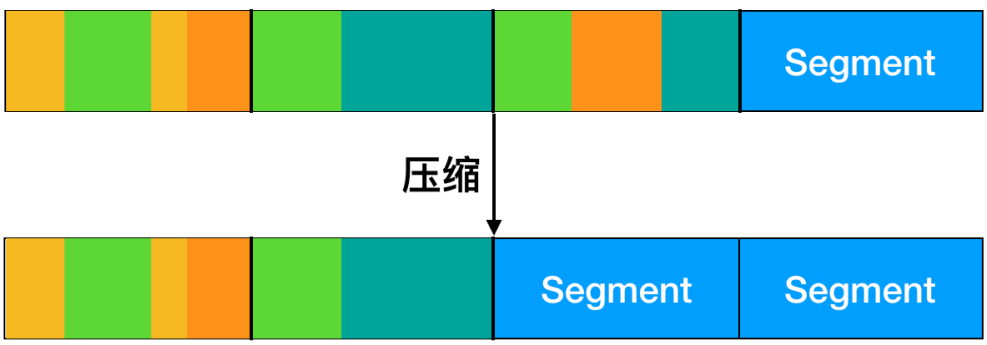
> 日志结构文件系统主要使用四种数据结构：Inode、Inode Map、Segment、Segment Usage Table。



到目前为止，所有写入最初都缓存在 `内存` 中，并且追加在 `日志末尾` ，所有缓存的写入都定期在单个段中写入磁盘。所以，现在打开文件也就意味着用映射定位文件的索引节点。一旦 inode 被定位后，磁盘块的地址就能够被找到。所有这些块本身都将位于日志中某处的分段中。

真实情况下的磁盘容量是有限的，所以最终日志会占满整个磁盘空间，这种情况下就会出现没有新的磁盘块被写入到日志中。幸运的是，许多现有段可能具有不再需要的块。例如，如果一个文件被覆盖了，那么它的 inode 将被指向新的块，但是旧的磁盘块仍在先前写入的段中占据着空间。

为了处理这个问题，LFS 有一个 `清理(clean)` 线程，它会循环扫描日志并对日志进行压缩。首先，通过查看日志中第一部分的信息来查看其中存在哪些索引节点和文件。它会检查当前 inode 的映射来查看 inode 否在在当前块中，是否仍在被使用。如果不是，该信息将被丢弃。如果仍然在使用，那么 inode 和块就会进入内存等待写回到下一个段中。然后原来的段被标记为空闲，以便日志可以用来存放新的数据。用这种方法，清理线程遍历日志，从后面移走旧的段，然后将有效的数据放入内存等待写到下一个段中。由此一来整个磁盘会形成一个大的 `环形缓冲区` ，写线程将新的段写在前面，而清理线程则清理后面的段。



日志文件系统

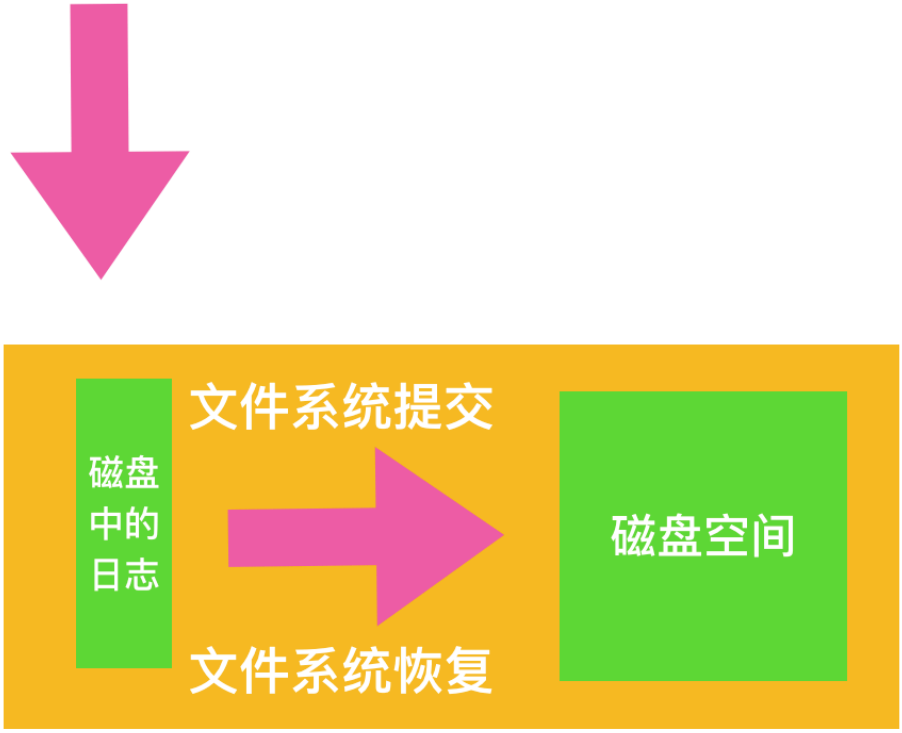
虽然日志结构系统的设计很优雅，但是由于它们和现有的文件系统不相匹配，因此还没有广泛使用。不过，从日志文件结构系统衍生出来一种新的日志系统，叫做 `日志文件系统` ，它会记录系统下一步将要做什么的日志。微软的 `NTFS` 文件系统、Linux 的 `ext3` 就使用了此日志。 `OS X` 将日志系统作为可供选项。为了看清它是如何工作的，我们下面讨论一个例子，比如 `移除文件` ，这个操作在 UNIX 中需要三个步骤完成：

- 在目录中删除文件
- 释放 inode 到空闲 inode 池
- 将所有磁盘块归还给空闲磁盘池。

在 Windows 中，也存在类似的步骤。不存在系统崩溃时，这些步骤的执行顺序不会带来问题。但是一旦系统崩溃，就会带来问题。假如在第一步完成后系统崩溃。inode 和文件块将不会被任何文件获得，也不会再分配；它们只存在于废物池中的某个地方，并因此减少了可利用的资源。如果崩溃发生在第二步后，那么只有磁盘块会丢失。 `日志文件系统` 保留磁盘写入期间对文件系统所做的更改的日志或日志，该日志可用于快速重建可能由于系统崩溃或断电等事件而发生的损坏。

> 一般文件系统崩溃后必须运行 `fsck（文件系统一致性检查）` 实用程序。

为了让日志能够正确工作，被写入的日志操作必须是 `幂等的(idempotent)` ，它意味着只要有必要，它们就可以重复执行很多次，并不会带来破坏。像操作 **更新位表并标记 inode k 或者块 n 是空闲的** 可以重复执行任意次。同样地，查找一个目录并且删除所有叫 `foobar` 的项也是幂等的。相反，把从 inode k 新释放的块加入空闲表的末端不是幂等的，因为它们可能已经被释放并存放在那里了。



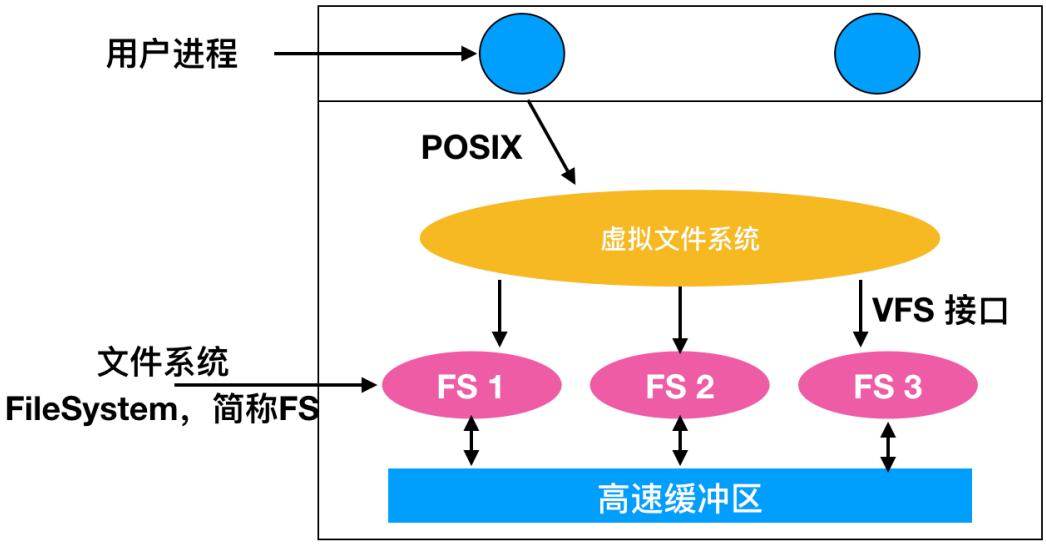
为了增加可靠性，一个文件系统可以引入数据库中 `原子事务(atomic transaction)` 的概念。使用这个概念，一组动作可以被界定在开始事务和结束事务操作之间。这样，文件系统就会知道它必须完成所有的动作，要么就一个不做。

虚拟文件系统

即使在同一台计算机上或者在同一个操作系统下，都会使用很多不同的文件系统。Windows 中的主要文件系统是 `NTFS 文件系统`，但不是说 Windows 只有 NTFS 操作系统，它还有一些其他的例如旧的 `FAT -32` 或 `FAT -16` 驱动器或分区，其中包含仍需要的数据，闪存驱动器，旧的 CD-ROM 或 DVD（每个都有自己的独特文件系统）。Windows 通过指定不同的盘符来处理这些不同的文件系统，比如 `C:`，`D:` 等。盘符可以显示存在也可以隐式存在，如果你想找指定位置的文件，那么盘符是显示存在；如果当一个进程打开一个文件时，此时盘符是隐式存在，所以 Windows 知道向哪个文件系统传递请求。

相比之下，UNIX 采用了一种不同的方式，即 UNIX 把多种文件系统整合到一个统一的结构中。一个 Linux 系统可以使用 `ext2` 作为根文件系统，`ext3` 分区装载在 `/usr` 下，另一块采用 `Reiser FS` 文件系统的硬盘装载到 `/home` 下，以及一个 ISO 9660 的 CD - ROM 临时装载到 `/mnt` 下。从用户的观点来看，只有一个文件系统层级，但是事实上它们是由多个文件系统组合而成，对于用户和进程是不可见的。

UNIX 操作系统使用一种 `虚拟文件系统(Virtual File System, VFS)` 来尝试将多种文件系统构成一个有序的结构。关键的思想是抽象出所有文件系统都共有的部分，并将这部分代码放在一层，这一层再调用具体文件系统来管理数据。下面是一个 VFS 的系统结构



还是那句经典的话，在计算机世界中，任何解决不了的问题都可以加个 `代理` 来解决。所有和文件相关的系统调用在最初的处理上都指向虚拟文件系统。这些来自用户进程的调用，都是标准的 `POSIX 系统调用`，比如 open、read、write 和 seek 等。VFS 对用户进程有一个 `上层` 接口，这个接口就是著名的 POSIX 接口。

VFS 也有一个对于实际文件的 `下层` 接口，就是上图中标记为 VFS 的接口。这个接口包含许多功能调用，这样 VFS 可以使每一个文件系统完成任务。因此，要创建一个可以与 VFS 一起使用的新文件系统，新文件系统的设计者必须确保它提供了 VFS 要求的功能。一个明显的例子是从磁盘读取特定的块，然后将其放入文件系统的缓冲区高速缓存中，然后返回指向该块的指针的函数。 因此，VFS具有两个不同的接口：上一个到用户进程，下一个到具体文件系统。

当系统启动时，根文件系统在 VFS 中注册。另外，当装载其他文件时，不管在启动时还是在操作过程中，它们也必须在 VFS 中注册。当一个文件系统注册时，根文件系统注册到 VFS。另外，在引导时或操作期间挂载其他文件系统时，它们也必须向 VFS 注册。当文件系统注册时，其基本作用是提供 VFS 所需功能的地址列表、调用向量表、或者 VFS 对象。因此一旦文件系统注册到 VFS，它就知道从哪里

开始读取数据块。

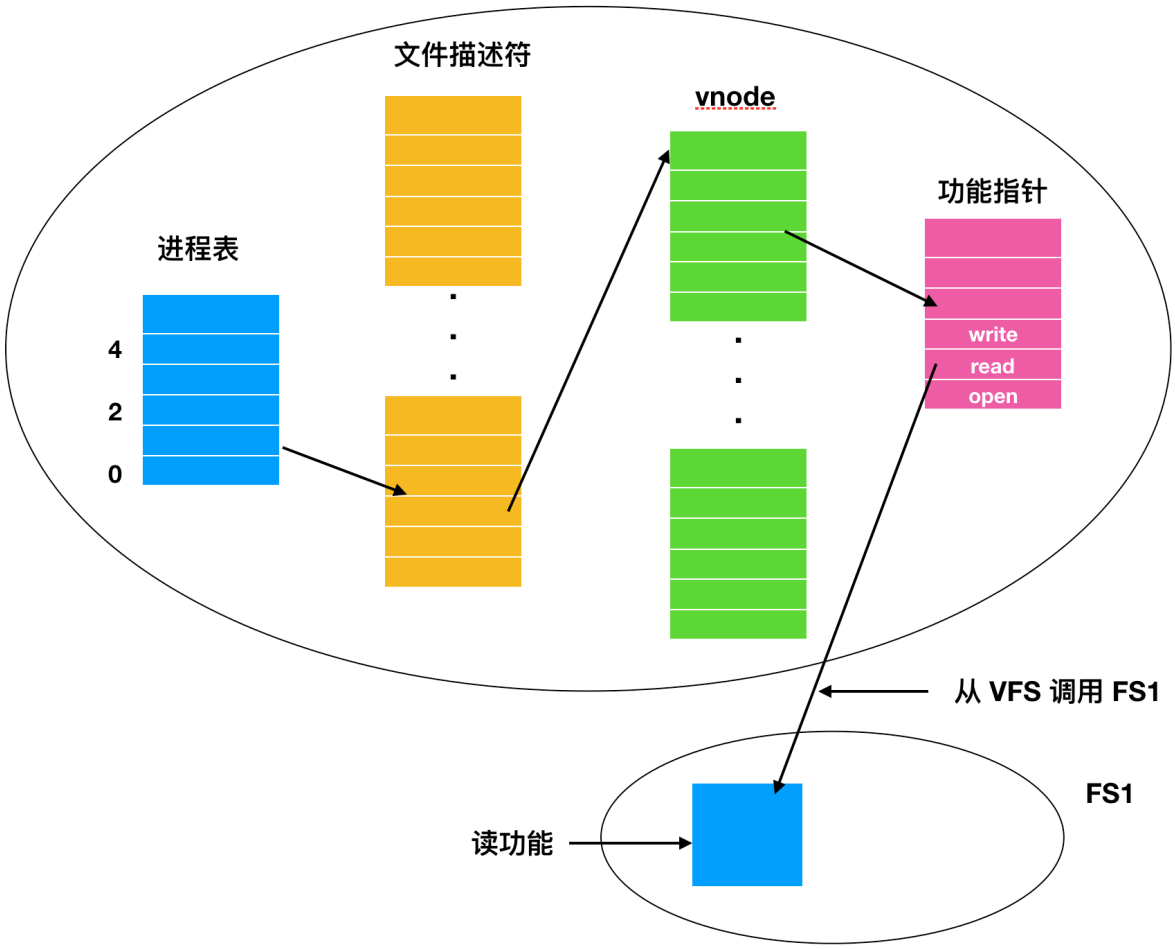
装载文件系统后就可以使用它了。比如，如果一个文件系统装载到 `/usr` 并且一个进程调用它：

```
1    open("/usr/include/unistd.h",O_RDONLY)
```

当解析路径时， VFS 看到新的文件系统被挂载到 `/usr` ，并且通过搜索已经装载文件系统的超级块来确定它的超块。然后它找到它所转载的文件的根目录，在那里查找路径 `include/unistd.h` 。然后 VFS 创建一个 vnode 并调用实际文件系统，以返回所有的在文件 inode 中的信息。这个信息和其他信息一起复制到 vnode （内存中）。而这些其他信息中最重要的是指向包含调用 vnode 操作的函数表的指针，比如 read、write 和 close 等。

当 vnode 被创建后，为了进程调用，VFS 在文件描述符表中创建一个表项，并将它指向新的 vnode，最后，VFS 向调用者返回文件描述符，所以调用者可以用它去 read、write 或者 close 文件。

当进程用文件描述符进行一个读操作时，VFS 通过进程表和文件描述符确定 vnode 的位置，并跟随指针指向函数表，这样就调用了处理 read 函数，运行在实际系统中的代码并得到所请求的块。VFS 不知道请求时来源于本地硬盘、还是来源于网络中的远程文件系统、CD-ROM 、USB 或者其他介质，所有相关的数据结构欧如下图所示



从调用者进程号和文件描述符开始，进而是 vnode，读函数指针，然后是对实际文件系统的访问函数定位。

文件系统的管理和优化

能够使文件系统工作是一回事，能够使文件系统高效、稳定的工作是另一回事，下面我们就来探讨一下文件系统的管理和优化。

磁盘空间管理

文件通常存在磁盘中，所以如何管理磁盘空间是一个操作系统的设计者需要考虑的问题。在文件上进行存有两种策略：**分配 n 个字节的连续磁盘空间；或者把文件拆分成多个并不一定连续的块**。在存储管理系统中，主要有 分段管理 和 分页管理 两种方式。

正如我们所看到的，按 连续字节序列 存储文件有一个明显的问题，当文件扩大时，有可能需要在磁盘上移动文件。内存中分段也有同样的问题。不同的是，相对于把文件从磁盘的一个位置移动到另一个位置，内存中段的移动操作要快很多。因此，几乎所有的文件系统都把文件分割成固定大小的块来存储。

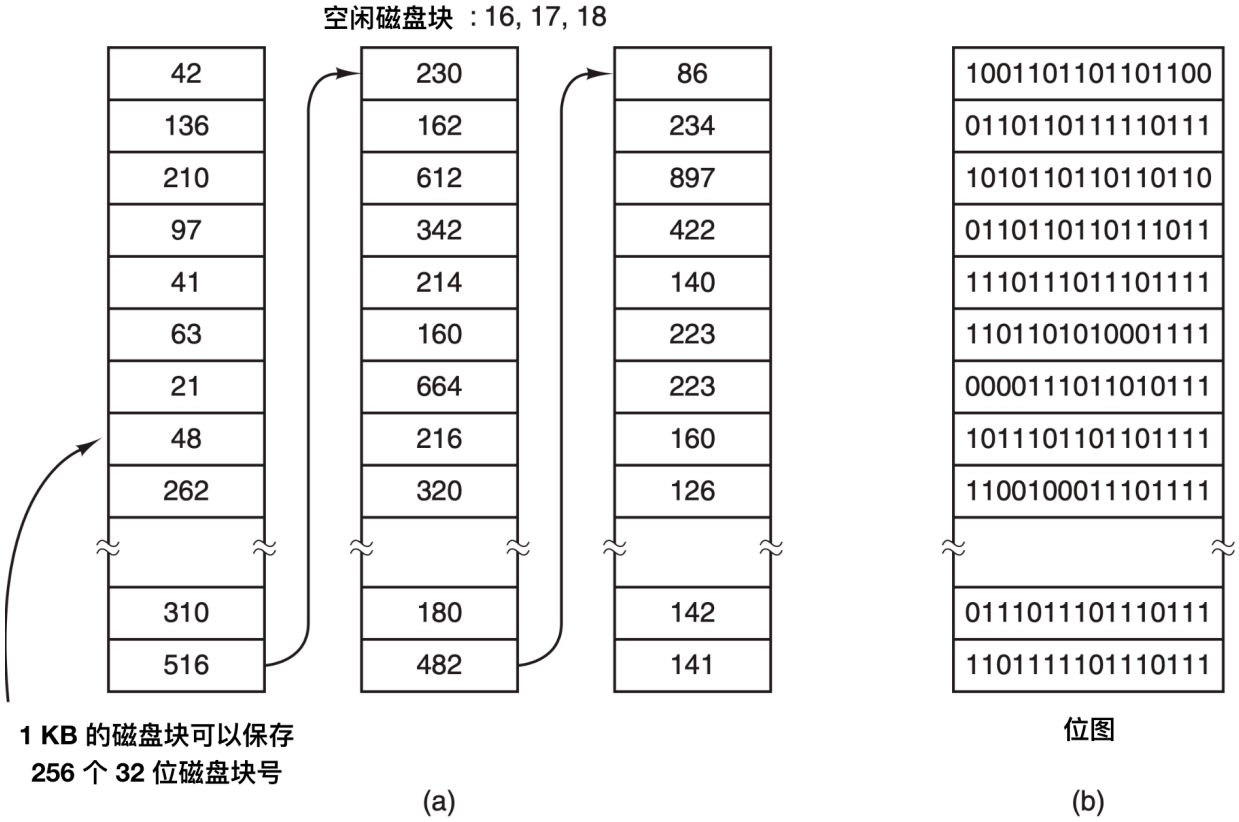
块大小

一旦把文件分为固定大小的块来存储，就会出现问题，块的大小是多少？按照**磁盘组织方式，扇区、磁道和柱面显然都可以作为分配单位**。在分页系统中，分页大小也是主要因素。

拥有大的块尺寸意味着每个文件，甚至 1 字节文件，都要占用一个柱面空间，也就是说小文件浪费了大量的磁盘空间。另一方面，小块意味着大部分文件将会跨越多个块，因此需要多次搜索和旋转延迟才能读取它们，从而降低了性能。因此，如果分配的块 太大 会浪费 空间 ；分配的块 太小 会浪费 时间 。

记录空闲块

一旦指定了块大小，下一个问题就是怎样跟踪空闲块。有两种方法被广泛采用，如下图所示



第一种方法是采用 `磁盘块链表` ，链表的每个块中包含极可能多的空闲磁盘块号。对于 1 KB 的块和 32 位的磁盘块号，空闲表中每个块包含有 255 个空闲的块号。考虑 1 TB 的硬盘，拥有大概十亿个磁盘块。为了存储全部地址块号，如果每块可以保存 255 个块号，则需要将近 400 万个块。通常，空闲块用于保存空闲列表，因此存储基本上是空闲的。

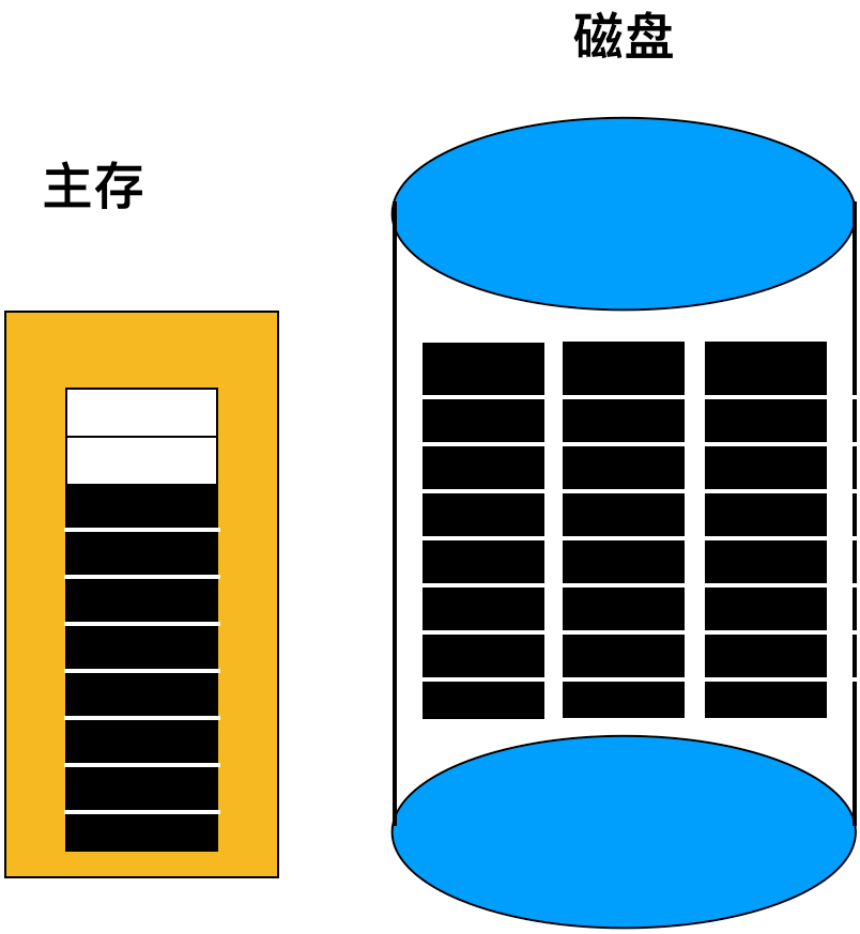
另一种空闲空间管理的技术是 `位图(bitmap)` ，n 个块的磁盘需要 n 位位图。在位图中，空闲块用 1 表示，已分配的块用 0 表示。对于 1 TB 硬盘的例子，需要 10 亿位表示，即需要大约 130 000 个 1 KB 块存储。很明显，和 32 位链表模型相比，位图需要的空间更少，因为每个块使用 1 位。只有当磁盘快满的时候，链表需要的块才会比位图少。

如果空闲块是长期连续的话，那么空闲列表可以改成记录连续分块而不是单个的块。每个块都会使用 8 位、16位、32 位的计数来与每个块相联，来记录连续空闲块的数量。最好的情况是一个空闲块可以用两个数字来表示：**第一个空闲块的地址和空闲块的计数**。另一方面，如果磁盘严重碎片化，那么跟踪连续分块要比跟踪单个分块运行效率低，因为不仅要存储地址，还要存储数量。

> 这种情况说明了一个操作系统设计者经常遇到的一个问题。有许多数据结构和算法可以用来解决问题，但是选择一个 `最好` 的方案需要数据的支持，而这些数据是设计者无法预先拥有的。只有在系统部署完毕真正使用使用后才会获得。

现在，回到空闲链表的方法，只有一个指针块保存在内存中。创建文件时，所需要的块从指针块中取出。当它用完时，将从磁盘中读取一个新的指针块。类似地，删除文件时，文件的块将被释放并添加到主存中的指针块中。当块被填满时，写回磁盘。

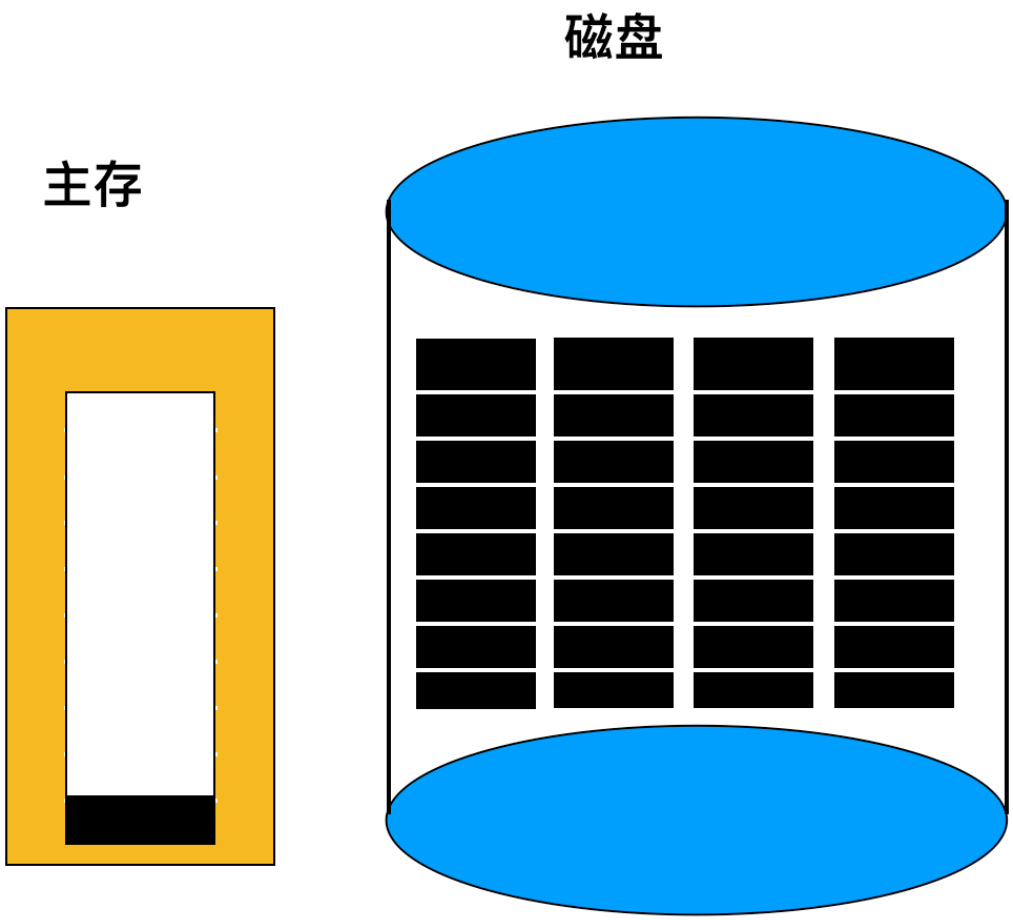
在某些特定的情况下，这个方法导致了不必要的磁盘 IO，如下图所示



a

在内存中一个被指向空闲磁盘处的指针几乎要满的磁盘块，以及磁盘上的三个指针块

上面内存中的指针块仅有两个空闲块，如果释放了一个含有三个磁盘块的文件，那么该指针块就会溢出，必须将其写入磁盘，那么就会产生如下图的这种情况。

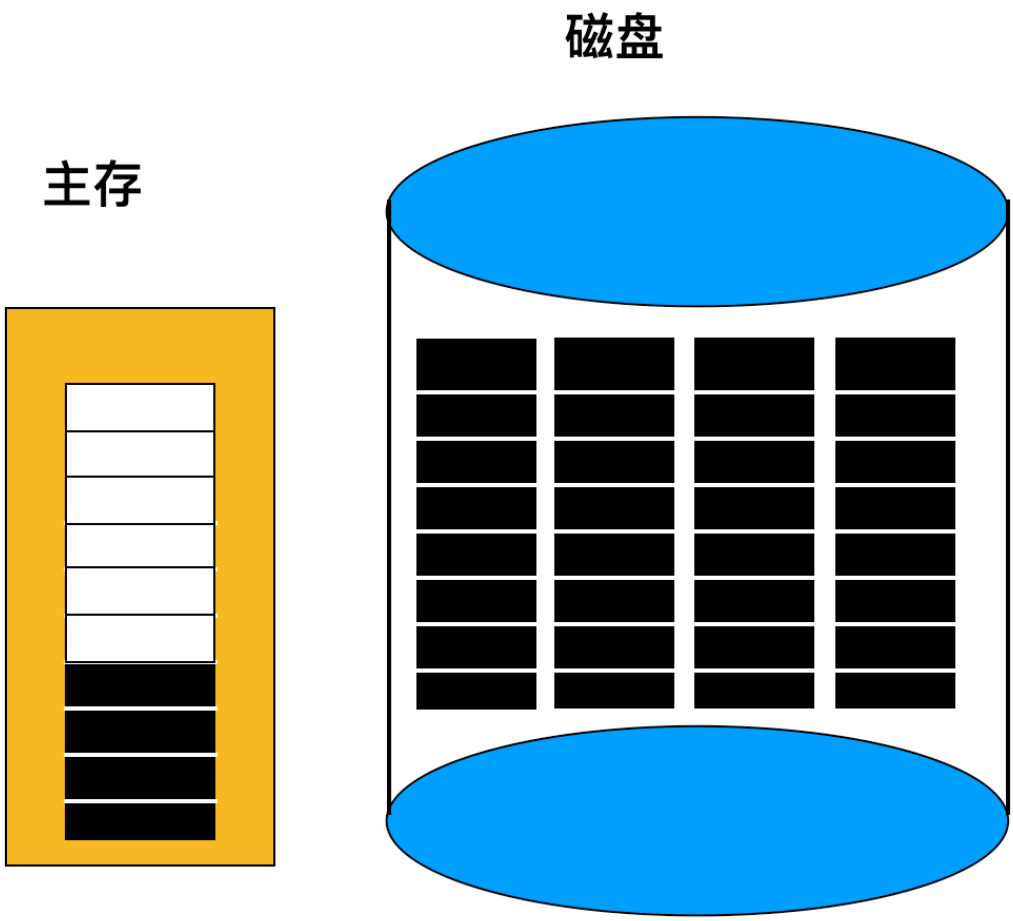


b

释放一个有三个块的文件的结果

如果现在写入含有三个块的文件，已满的指针不得不再次读入，这将会回到上图 a 中的情况。如果有三个块的文件只是作为临时文件被写入，在释放它时，需要进行另一次磁盘写操作以将完整的指针块写回到磁盘。简而言之，当指针块几乎为空时，一系列短暂的临时文件可能会**导致大量磁盘 I/O**。

避免大部分磁盘 I/O 的另一种方法是 拆分完整的指针块 。这样，当释放三个块时，变化不再是从 a - b，而是从 a - c，如下图所示



c

处理该三个块的文件的替代策略
带阴影的表项代表指向空闲磁盘块的指针

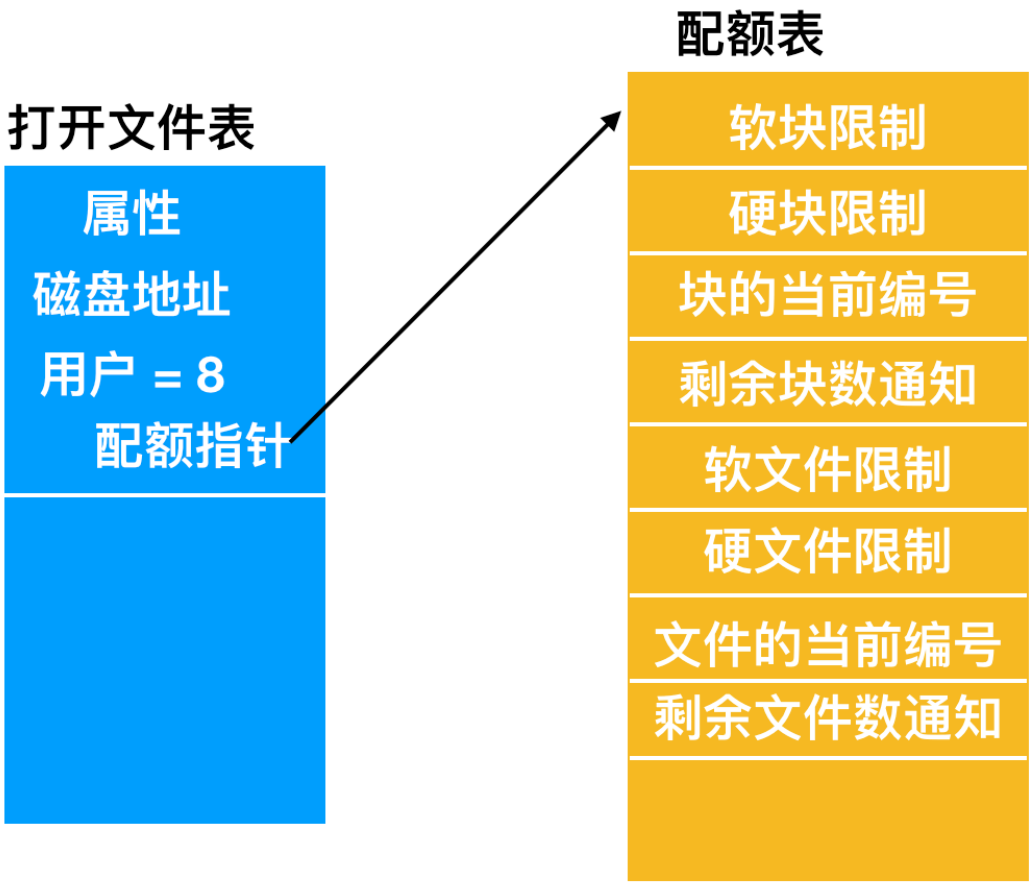
现在，系统可以处理一系列临时文件，而不需要进行任何磁盘 I/O。如果内存中指针块满了，就写入磁盘，半满的指针块从磁盘中读入。这里的思想是：要保持磁盘上的大多数指针块为满的状态（减少磁盘的使用），但是在内存中保留了一个半满的指针块。这样，就可以既处理文件的创建又同时可以处理文件的删除操作，而不会为空闲表进行磁盘 I/O。

对于位图，会在内存中只保留一个块，只有在该块满了或空了的情形下，才到磁盘上取另一个块。通过在位图的单一块上进行所有的分配操作，磁盘块会紧密的聚集在一起，从而 `减少了磁盘臂的移动` 。由于位图是一种固定大小的数据结构，所以如果内核是 `分页` 的，就可以把位图放在虚拟内存中，在需要时将位图的页面调入。

磁盘配额

为了防止一些用户占用太多的磁盘空间，多用户操作通常提供一种 `磁盘配额(enforcing disk quotas)` 的机制。系统管理员为每个用户分配**最大的文件和块分配**，并且操作系统确保用户不会超过其配额。我们下面会谈到这一机制。

在用户打开一个文件时，操作系统会找到 `文件属性` 和 `磁盘地址` ，并把它们送入内存中的打开文件表。其中一个属性告诉 `文件所有者` 是谁。任何有关文件的增加都会记到所有者的配额中。



配额表中记录了每个用户的配额

第二张表包含了每个用户当前打开文件的配额记录，即使是其他人打开该文件也一样。如上图所示，该表的内容是从被打开文件的所有者的磁盘配额文件中提取出来的。当所有文件关闭时，该记录被写回配额文件。

当在打开文件表中建立一新表项时，会产生一个指向所有者配额记录的指针。每次向文件中添加一个块时，文件所有者所用数据块的总数也随之增加，并会同时增加 `硬限制` 和 `软限制` 的检查。可以超出软限制，但硬限制不可以超出。当已达到硬限制时，再往文件中添加内容将引发错误。同样，对文件数目也存在类似的检查。

> 什么是硬限制和软限制？**硬限制是软限制的上限**。软限制是为会话或进程实际执行的限制。这允许管理员（或用户）将硬限制设置为允许它们希望允许的最大使用上限。然后，其他用户和进程可以根据需要使用软限制将其资源使用量自限制到更低的上限。

当一个用户尝试登陆，系统将检查配额文件以查看用户是否超出了文件数量或磁盘块数量的 `软限制` 。如果违反了任一限制，则会显示警告，保存的警告计数减 1，如果警告计数为 0，表示用户多次忽略该警告，因而将不允许该用户登录。要想再得到登录的许可，就必须与系统管理员协商。

如果用户在退出系统时消除所超过的部分，他们就可以再一次终端会话期间超过其软限制，但**无论什么情况下都不会超过硬限制**。

文件系统备份

文件系统的毁坏要比计算机的损坏严重很多。无论是硬件还是软件的故障，只要计算机文件系统被破坏，要恢复起来都是及其困难的，甚至是不可能的。因为文件系统无法抵御破坏，因而我们要在文件系统在被破坏之前做好 `数据备份` ，但是备份也不是那么容易，下面我们就来探讨备份的过程。

许多人认为为文件系统做备份是不值得的，并且很浪费时间，直到有一天他们的磁盘坏了，他们才意识到事情的严重性。相对来说，公司在这方面做的就很到位。磁带备份主要要处理好以下两个潜在问题中的一个

- 从意外的灾难中恢复

这个问题主要是由于外部条件的原因造成的，比如磁盘破裂，水灾火灾等。

- 从错误的操作中恢复

第二个问题通常是由于用户意外的删除了原本需要还原的文件。这种情况发生的很频繁，使得 Windows 的设计者们针对 `删除` 命令专门设计了特殊目录，这就是 `回收站(recycle bin)` ，也就是说，在删除文件的时候，文件本身并不真正从磁盘上消失，而是被放置到这个特殊目录下，等以后需要的时候可以还原回去。文件备份更主要是指这种情况，能够允许几天之前，几周之前的文件从原来备份的磁盘进行还原。

做文件备份很耗费时间而且也很浪费空间，这会引起下面几个问题。首先，是要**备份整个文件还是仅备份一部分呢**？一般来说，只是备份特定目录及其下的全部文件，而不是备份整个文件系统。

其次，对上次未修改过的文件再进行备份是一种浪费，因而产生了一种 `增量转储(incremental dumps)` 的思想。最简单的增量转储的形式就是 `周期性` 的做全面的备份，而每天只对增量转储完成后发生变化的文件做单个备份。

> 周期性：比如一周或者一个月

稍微好一点的方式是只备份最近一次转储以来更改过的文件。当然，这种做法极大的缩减了转储时间，但恢复起来却更复杂，因为**最近的全面转储先要全部恢复，随后按逆序进行增量转储**。为了方便恢复，人们往往使用更复杂的转储模式。

第三，既然待转储的往往是海量数据，那么在将其写入磁带之前对文件进行压缩就很有必要。但是，如果在备份过程中出现了文件损坏的情况，就会导致破坏压缩算法，从而使整个磁带无法读取。所以在备份前是否进行文件压缩需慎重考虑。

第四，对正在使用的文件系统做备份是很难的。如果在转储过程中要添加，删除和修改文件和目录，则转储结果可能不一致。因此，因为转储过程中需要花费数个小时的时间，所以有必要在晚上将系统脱机进行备份，然而这种方式的接受程度并不高。所以，人们修改了转储算法，记下文件系统的 `瞬时快照` ，即复制关键的数据结构，然后需要把将来对文件和目录所做的修改复制到块中，而不是到处更新他们。

磁盘转储到备份磁盘上有两种方案：**物理转储和逻辑转储**。 `物理转储(physical dump)` 是从磁盘的 0 块开始，依次将所有磁盘块按照顺序写入到输出磁盘，并在复制最后一个磁盘时停止。这种程序的万无一失性是其他程序所不具备的。

第二个需要考虑的是**坏块的转储**。制造大型磁盘而没有瑕疵是不可能的，所以也会存在一些 `坏块(bad blocks)` 。有时进行低级格式化后，坏块会被检测出来并进行标记，这种情况的解决办法是用磁盘末尾的一些空闲块所替换。

然而，一些块在格式化后会变坏，在这种情况下操作系统可以检测到它们。通常情况下，它可以通过创建一个由所有坏块组成的 `文件` 来解决问题，确保它们不会出现在空闲池中并且永远不会被分配。**那么此文件是完全不可读的**。如果磁盘控制器将所有的坏块重新映射，物理转储还是能够正常工作的。

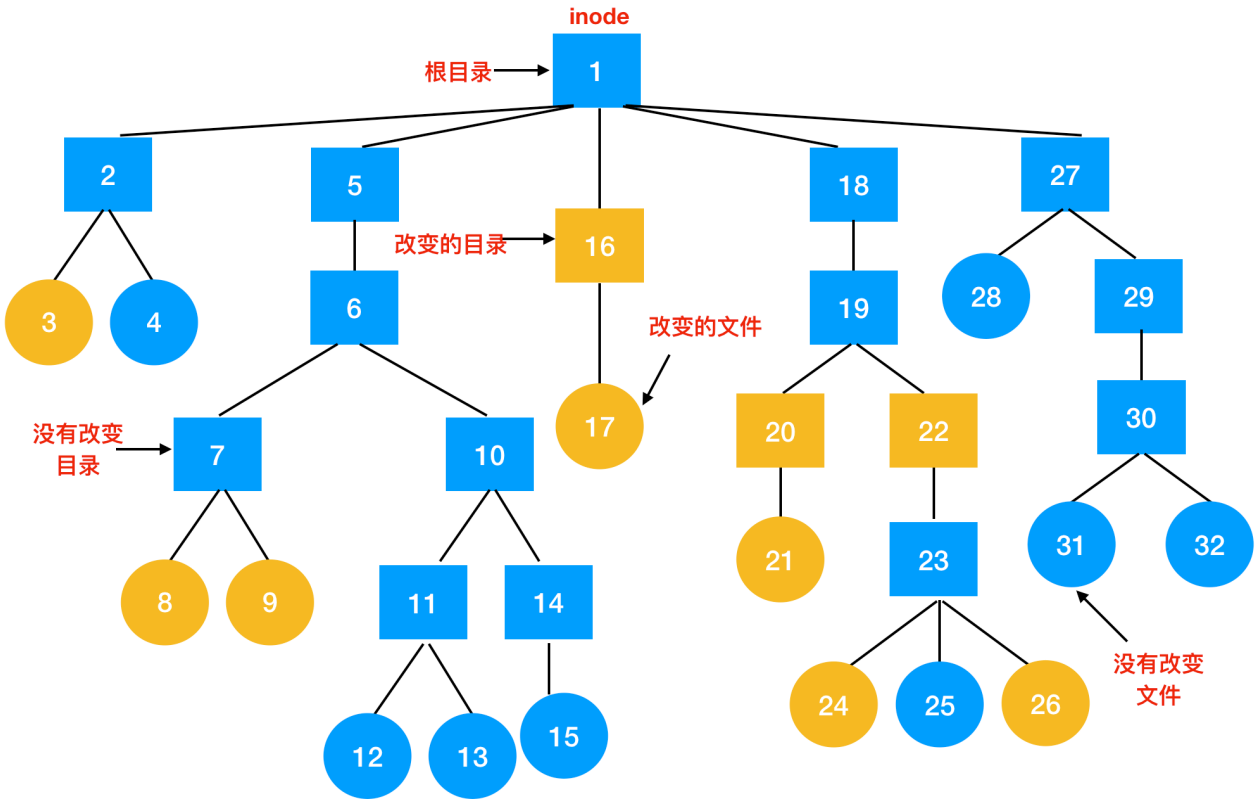
Windows 系统有 `分页文件(paging files)` 和 `休眠文件(hibernation files)` 。它们在文件还原时不发挥作用，同时也不应该在第一时间进行备份。

物理转储和逻辑转储

物理转储的主要优点是简单、极为快速（基本上是以磁盘的速度运行），缺点是 `全量备份` ，不能跳过指定目录，也不能增量转储，也不能恢复个人文件的请求。因此句**大多数情况下不会使用物理转储，而使用逻辑转储**。

`逻辑转储(logical dump)` 从一个或几个指定的目录开始，递归转储自指定日期开始后更改的文件和目录。因此，在逻辑转储中，转储磁盘上有一系列经过仔细识别的目录和文件，这使得根据请求轻松还原特定文件或目录。

既然逻辑转储是最常用的方式，那么下面就让我们研究一下逻辑转储的通用算法。此算法在 UNIX 系统上广为使用，如下图所示



待转储的文件系统，其中方框代表 `目录` ，圆圈代表 `文件` 。黄色的项目表是自上次转储以来修改过。每个目录和文件都被标上其 inode 号。

此算法会转储位于修改文件或目录路径上的所有目录（也包括未修改的目录），原因有两个。第一是能够在不同电脑的文件系统中恢复转储的文件。通过这种方式，转储和重新存储的程序能够用来在两个电脑之间 `传输整个文件系统` 。第二个原因是能够对单个文件进行 `增量恢复` 。

逻辑转储算法需要维持一个 inode 为索引的 位图(bitmap) ，每个 inode 包含了几位。随着算法的进行，位图中的这些位会被设置或清除。算法的执行分成四个阶段。第一阶段从 起始目录（本例为根目录） 开始检查其中所有的目录项。对每一个修改过的文件，该算法将在位图中标记其 inode。算法还会标记并递归检查每一个目录（不管是否修改过）。

在第一阶段结束时，所有修改过的文件和全部目录都在位图中标记了，如下图所示

a) 

理论上来说，第二阶段再次递归遍历目录树，并去掉目录树中任何不包含被修改过的文件或目录的标记。本阶段执行的结果如下

b) 

注意，inode 编号为 10、11、14、27、29 和 30 的目录已经被去掉了标记，因为它们所包含的内容 没有修改 。它们也不会转储。相反，inode 编号为 5 和 6 的目录本身尽管没有被修改过也要被转储，因为在新的机器上恢复当日的修改时需要这些信息。为了提高算法效率，可以将这两阶段的目录树遍历合二为一。

现在已经知道了哪些目录和文件必须被转储了，这就是上图 b 中标记的内容，第三阶段算法将以节点号为序，扫描这些 inode 并转储所有标记为需转储的目录，如下图所示

c) 

为了进行恢复，每个被转储的目录都用目录的属性（所有者、时间）作为前缀。

d) 

最后，在第四阶段，上图中被标记的文件也被转储，同样，由其文件属性作为前缀。至此，转储结束。

从转储磁盘上还原文件系统非常简单。一开始，需要在磁盘上创建空文件系统。然后恢复最近一次的完整转储。由于磁带上最先出现目录，所以首先恢复目录，给出文件系统的 框架(skeleton) ，然后恢复文件系统本身。在完整存储之后是第一次增量存储，然后是第二次重复这一过程，以此类推。

尽管逻辑存储十分简单，但是也会有一些棘手的问题。首先，既然空闲块列表并不是一个文件，那么在所有被转储的文件恢复完毕之后，就需要从零开始重新构造。

另外一个问题是关于 链接 。如果文件链接了两个或者多个目录，而文件只能还原一次，那么并且所有指向该文件的目录都必须还原。

还有一个问题是，UNIX 文件实际上包含了许多 空洞(holes) 。打开文件，写几个字节，然后找到文件中偏移了一定距离的地址，又写入更多的字节，这么做是合法的。但两者之间的这些块并不属于文件本身，从而也不应该在其上进行文件转储和恢复。

最后，无论属于哪一个目录，**特殊文件，命名管道以及类似的文件**都不应该被转储。

文件系统的一致性

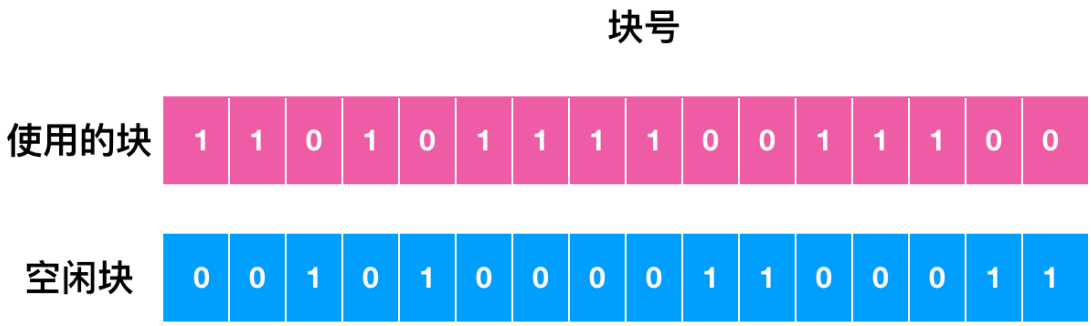
影响可靠性的一个因素是文件系统的一致性。许多文件系统读取磁盘块、修改磁盘块、再把它们写回磁盘。如果系统在所有块写入之前崩溃，文件系统就会处于一种 `不一致(inconsistent)` 的状态。如果某些尚未写回的块是索引节点块，目录块或包含空闲列表的块，则此问题是很严重的。

为了处理文件系统一致性问题，大部分计算机都会有应用程序来检查文件系统的一致性。例如，UNIX 有 `fsck`；Windows 有 `sfc`，每当引导系统时（尤其是在崩溃后），都可以运行该程序。

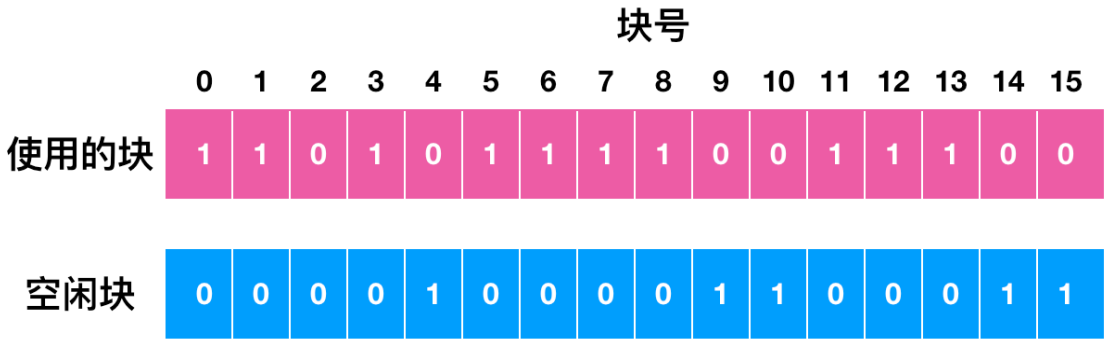
可以进行两种一致性检查：**块的一致性检查和文件的一致性检查**。为了检查块的一致性，应用程序会建立两张表，每个包含一个计数器的块，最初设置为 0 。第一个表中的计数器跟踪该块在文件中出现的次数，第二张表中的计数器记录每个块在空闲列表、空闲位图中出现的频率。

然后检验程序使用原始设备读取所有的 inode，忽略文件的结构，只返回从零开始的所有磁盘块。从 inode 开始，很容易找到文件中的块数量。每当读取一个块时，该块在第一个表中的计数器 + 1，应用程序会检查空闲块或者位图来找到没有使用的块。空闲列表中块的每次出现都会导致其在第二表中的计数器增加。

如果文件系统一致，则每一个块或者在第一个表计数器为 1，或者在第二个表计数器中为 1，如下图所示

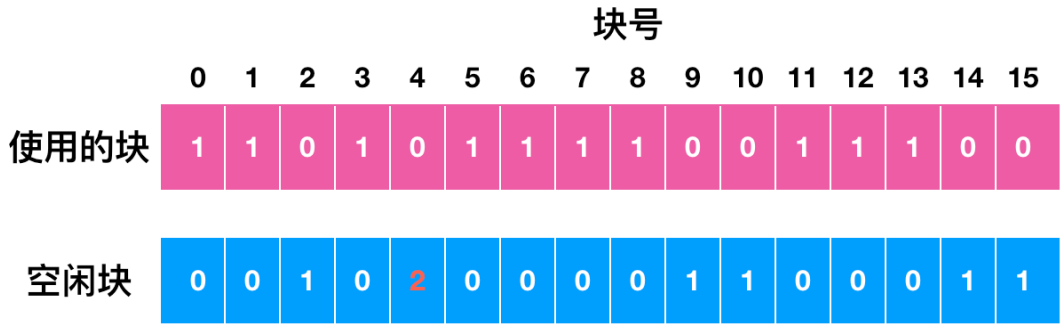


但是当系统崩溃后，这两张表可能如下所示



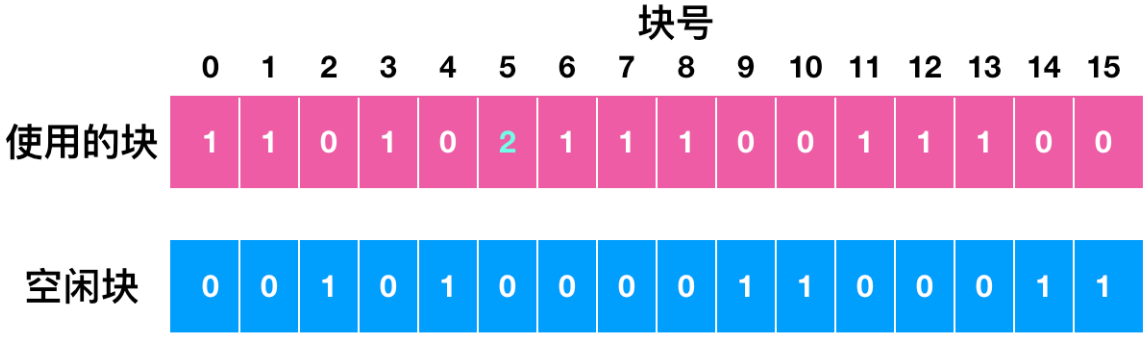
其中，磁盘块 2 没有出现在任何一张表中，这称为 `块丢失(missing block)`。尽管块丢失不会造成实际的损害，但它的确浪费了磁盘空间，减少了磁盘容量。块丢失的问题很容易解决，文件系统检验程序把他们加到空闲表中即可。

有可能出现的另外一种情况如下所示



其中，块 4 在空闲表中出现了 2 次。这种解决方法也很简单，只要重新建立空闲表即可。

最糟糕的情况是在两个或者多个文件中出现同一个数据块，如下所示



比如上图的磁盘块 5，如果其中一个文件被删除，块 5 会被添加到空闲表中，导致一个块同时处于使用和空闲的两种状态。如果删除这两个文件，那么在空闲表中这个磁盘块会出现两次。

文件系统检验程序采取的处理方法是，先分配一磁盘块，把块 5 中的内容复制到空闲块中，然后把它插入到其中一个文件中。这样文件的内容未改变，虽然这些内容可以肯定是不对的，但至少保证了文件的一致性。这一错误应该报告给用户，由用户检查受检情况。

除了检查每个磁盘块计数的正确性之外，文件系统还会检查目录系统。这时候会用到一张 `计数器表` ，但这时是一个文件（而不是一个块）对应于一个计数器。程序从根目录开始检验，沿着目录树向下查找，检查文件系统的每个目录。对每个目录中的文件，使其计数 + 1。

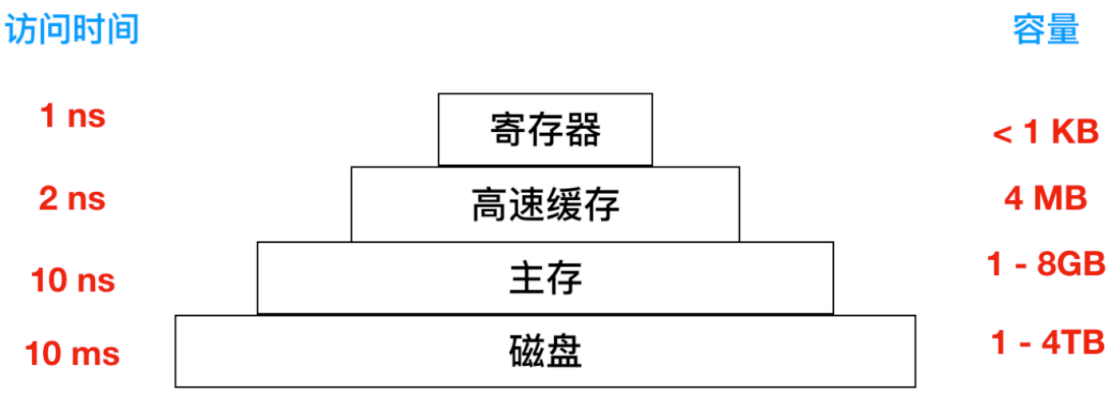
> 注意，由于存在硬连接，一个文件可能出现在两个或多个目录中。而遇到符号链接是不计数的，不会对目标文件的计数器 + 1。

在检验程序完成后，会得到一张由 inode 索引的表，说明每个文件和目录的包含关系。检验程序会将这些数字与存储在文件 inode 中的链接数目做对比。如果 inode 节点的链接计数大户目录项个数，这时即使所有文件从目录中删除，这个计数仍然不是 0 ，inode 不会被删除。这种错误不严重，却因为存在不属于任何目录的文件而浪费了磁盘空间。

另一种错误则是潜在的风险。如果同一个文件链接两个目录项，但是 inode 链接计数只为 1，如果删除了任何一个目录项，对应 inode 链接计数变为 0。当 inode 计数为 0 时，文件系统标志 inode 为 `未使用` ，并释放全部的块。这会导致其中一个目录指向一未使用的 inode，而很有可能其块马上就被分配给其他文件。

文件系统性能

访问磁盘的效率要比内存满的多，是时候又祭出这张图了



存储层次结构

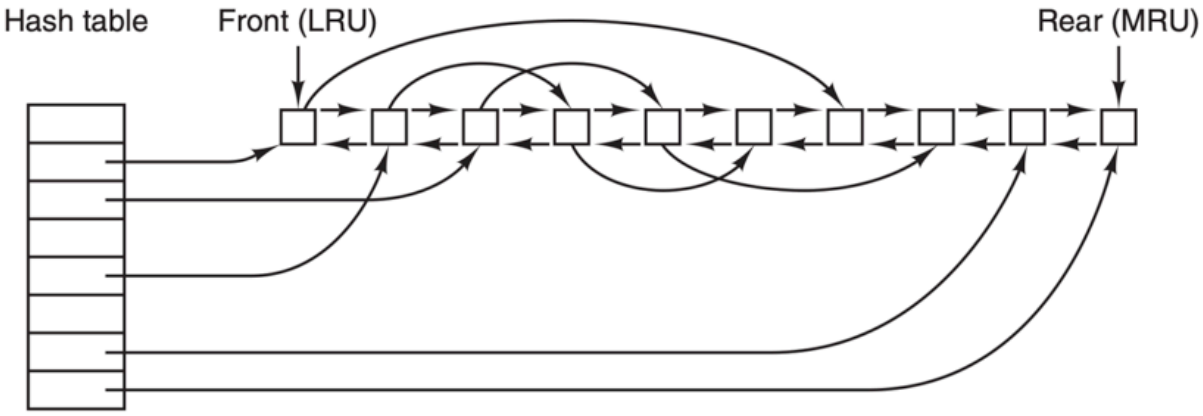
从内存读一个 32 位字大概是 10ns，从硬盘上读的速率大概是 100MB/S，对每个 32 位字来说，效率会慢了四倍，另外，还要加上 5 - 10 ms 的寻道时间等其他损耗，如果只访问一个字，内存要比磁盘快百万数量级。所以磁盘优化是很有必要的，下面我们会讨论几种优化方式

高速缓存

最常用的减少磁盘访问次数的技术是使用 `块高速缓存(block cache)` 或者 `缓冲区高速缓存(buffer cache)` 。高速缓存指的是一系列的块，它们在逻辑上属于磁盘，但实际上基于性能的考虑被保存在内存中。

管理高速缓存有不同的算法，常用的算法是：检查全部的读请求，查看在高速缓存中是否有所需要的块。如果存在，可执行读操作而无须访问磁盘。如果检查块不再高速缓存中，那么首先把它读入高速缓存，再复制到所需的地方。之后，对同一个块的请求都通过 `高速缓存` 来完成。

高速缓存的操作如下图所示



由于在高速缓存中有许多块，所以需要某种方法快速确定所需的块是否存在。常用方法是将设备和磁盘地址进行散列操作，然后，在散列表中查找结果。具有相同散列值的块在一个链表中连接在一起（这个数据结构是不是很像 HashMap?），这样就可以沿着冲突链查找其他块。

如果高速缓存 已满 ，此时需要调入新的块，则要把原来的某一块调出高速缓存，如果要调出的块在上次调入后已经被修改过，则需要把它写回磁盘。这种情况与分页非常相似，所有常用的页面置换算法我们之前已经介绍过，如果有不熟悉的小伙伴可以参考 **https://mp.weixin.qq.com/s/5-k2BJDgEp9symxcSwoprw** 。比如 **FIFO 算法、第二次机会算法、LRU 算法、时钟算法、老化算法等** 。它们都适用于高速缓存。

块提前读

第二个明显提高文件系统的性能是，在需要用到块之前，试图 提前 将其写入高速缓存，从而 提高命中率 。许多文件都是顺序读取。如果请求文件系统在某个文件中生成块 k，文件系统执行相关操作并且在完成之后，会检查高速缓存，以便确定块 k + 1 是否已经在高速缓存。如果不在，文件系统会为 k + 1 安排一个预读取，因为文件希望在用到该块的时候能够直接从高速缓存中读取。

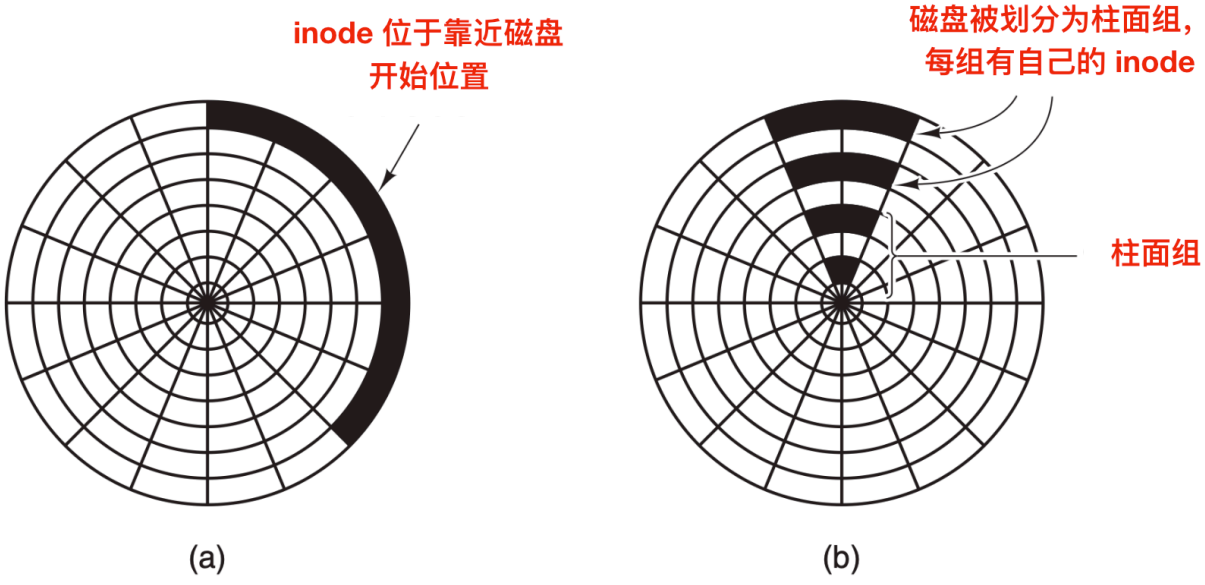
当然，块提前读取策略只适用于实际顺序读取的文件。对随机访问的文件，提前读丝毫不起作用。甚至还会造成阻碍。

减少磁盘臂运动

高速缓存和块提前读并不是提高文件系统性能的唯一方法。另一种重要的技术是**把有可能顺序访问的块放在一起，当然最好是在同一个柱面上，从而减少磁盘臂的移动次数**。当写一个输出文件时，文件系统就必须按照要求一次一次地分配磁盘块。如果用位图来记录空闲块，并且整个位图在内存中，那么选择与前一块最近的空闲块是很容易的。如果用空闲表，并且链表的一部分存在磁盘上，要分配紧邻的空闲块就会困难很多。

不过，即使采用空闲表，也可以使用 块簇 技术。即不用块而用连续块簇来跟踪磁盘存储区。如果一个扇区有 512 个字节，有可能系统采用 1 KB 的块（2 个扇区），但却按每 2 块（4 个扇区）一个单位来分配磁盘存储区。这和 2 KB 的磁盘块并不相同，因为在高速缓存中它仍然使用 1 KB 的块，磁盘与内存数据之间传送也是以 1 KB 进行，但在一个空闲的系统上顺序读取这些文件，寻道的次数可以减少一半，从而使文件系统的性能大大改善。若考虑旋转定位则可以得到这类方法的变体。在分配块时，系统尽量把一个文件中的连续块存放在同一个柱面上。

在使用 inode 或任何类似 inode 的系统中，另一个性能瓶颈是，读取一个很短的文件也需要两次磁盘访问：**一次是访问 inode，一次是访问块**。通常情况下，inode 的放置如下图所示



a）inode 放在磁盘开始位置；b）磁盘分为柱面组，每组有自己的块和 inode

其中，全部 inode 放在靠近磁盘开始位置，所以 inode 和它所指向的块之间的平均距离是柱面组的一半，这将会需要较长时间的寻道时间。

一个简单的改进方法是，在磁盘中部而不是开始处存放 inode ，此时，在 inode 和第一个块之间的寻道时间减为原来的一半。另一种做法是：将磁盘分成多个柱面组，每个柱面组有自己的 inode，数据块和空闲表，如上图 b 所示。

当然，只有在磁盘中装有磁盘臂的情况下，讨论寻道时间和旋转时间才是有意义的。现在越来越多的电脑使用 `固态硬盘(SSD)`，对于这些硬盘，由于采用了和闪存同样的制造技术，使得随机访问和顺序访问在传输速度上已经较为相近，传统硬盘的许多问题就消失了。但是也引发了新的问题。

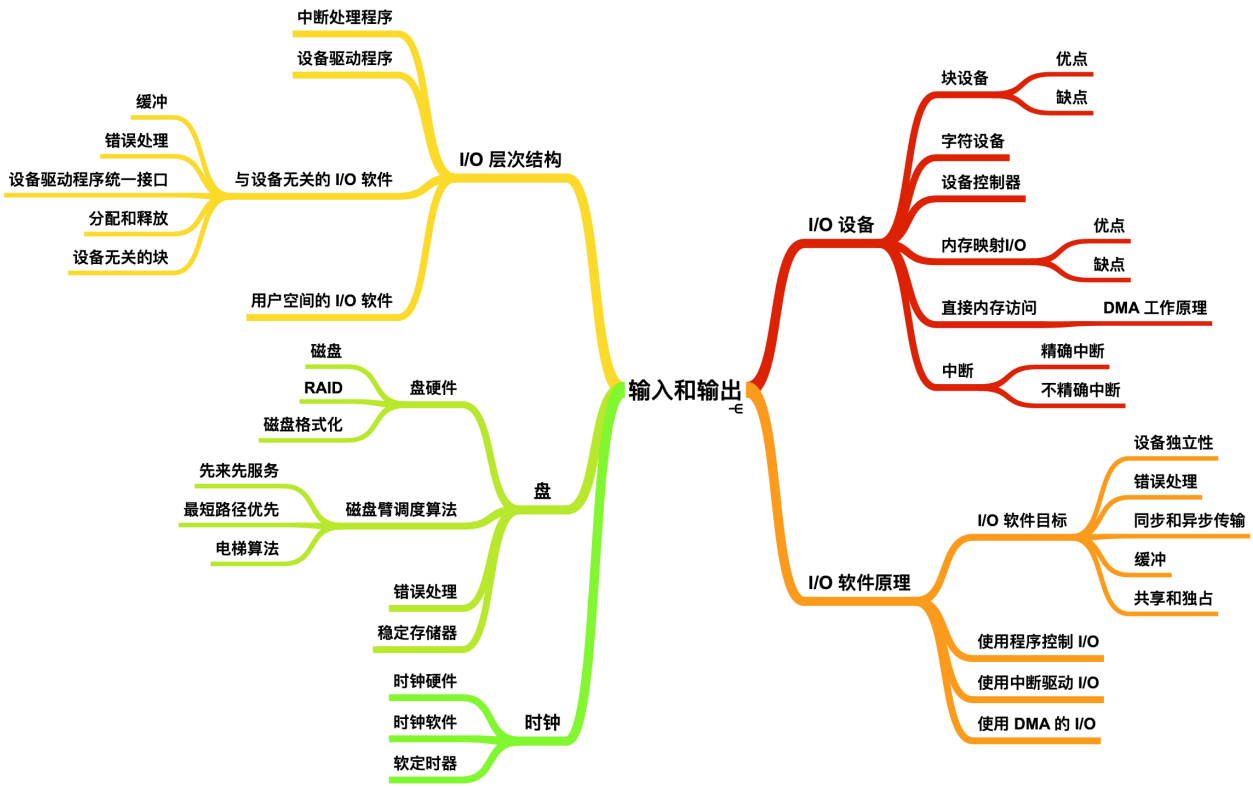
磁盘碎片整理

在初始安装操作系统后，文件就会被不断的创建和清除，于是磁盘会产生很多的碎片，在创建一个文件时，它使用的块会散布在整个磁盘上，降低性能。删除文件后，回收磁盘块，可能会造成空穴。

磁盘性能可以通过如下方式恢复：移动文件使它们相互挨着，并把所有的至少是大部分的空闲空间放在一个或多个大的连续区域内。Windows 有一个程序 `defrag` 就是做这个事儿的。Windows 用户会经常使用它，SSD 除外。

磁盘碎片整理程序会在让文件系统上很好地运行。Linux 文件系统（特别是 ext2 和 ext3）由于其选择磁盘块的方式，在磁盘碎片整理上一般不会像 Windows 一样困难，因此很少需要手动的磁盘碎片整理。而且，固态硬盘并不受磁盘碎片的影响，事实上，在固态硬盘上做磁盘碎片整理反倒是多此一举，不仅没有提高性能，反而磨损了固态硬盘。所以碎片整理只会缩短固态硬盘的寿命。

I/O



我们之前的文章提到了操作系统的三个抽象，它们分别是**进程、地址空间和文件**，除此之外，操作系统还要控制所有的 I/O 设备。操作系统必须向设备 `发送命令` ， `捕捉中断` 并 `处理错误` 。它还应该在设备和操作系统的其余部分之间提供一个简单易用的接口。操作系统**如何管理 I/O** 是我们接下来的重点。

不同的人对 I/O 硬件的理解也不同。对于电子工程师而言，I/O 硬件就是芯片、导线、电源和其他组成硬件的物理设备。而我们程序员眼中的 I/O 其实就是硬件提供给软件的 `接口` ，比如硬件接受到的命令、执行的操作以及反馈的错误。我们着重探讨的是如何对硬件进行编程，而不是其工作原理。

I/O 设备

什么是 I/O 设备？I/O 设备又叫做输入/输出设备，它是人类用来和计算机进行通信的外部硬件。输入/输出设备能够向计算机 `发送数据（输出）` 并从计算机 `接收数据（输入）` 。

`I/O 设备(I/O devices)` 可以分成两种： `块设备(block devices)` 和 `字符设备(character devices)` 。

块设备

块设备是一个能存储 `固定大小块` 信息的设备，它支持**以固定大小的块，扇区或群集读取和（可选）写入数据**。每个块都有自己的 `物理地址` 。通常块的大小在 512 - 65536 之间。所有传输的信息都会以 `连续` 的块为单位。块设备的基本特征是每个块都较为对立，能够独立的进行读写。常见的块设备有 **硬盘、蓝光光盘、USB 盘**

与字符设备相比，块设备通常需要较少的引脚。

块设备





块设备的缺点

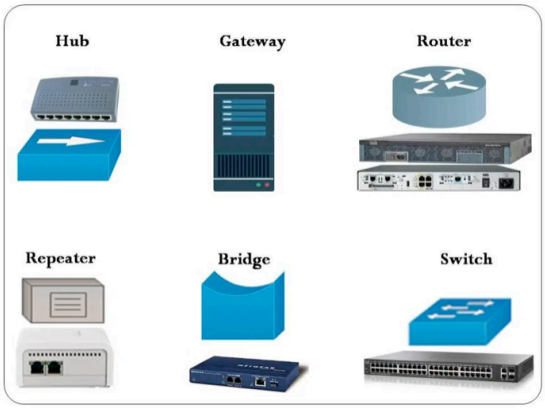
基于给定固态存储器的块设备比基于相同类型的存储器的字节寻址要慢一些，因为必须在块的开头开始读取或写入。所以，要读取该块的任何部分，必须寻找到该块的开始，读取整个块，如果不使用该块，则将其丢弃。要写入块的一部分，必须寻找到块的开始，将整个块读入内存，修改数据，再次寻找到块的开头处，然后将整个块写回设备。

字符设备

另一类 I/O 设备是 `字符设备` 。字符设备以 `字符` 为单位发送或接收一个字符流，而不考虑任何块结构。字符设备是不可寻址的，也没有任何寻道操作。常见的字符设备有 **打印机、网络设备、鼠标、以及大多数与磁盘不同的设备**。

字符设备





下面显示了一些常见设备的数据速率。

Device 设备	Data rate
Keyboard 键盘	10 bytes/sec
Mouse 鼠标	100 bytes/sec
56K modem 56K 调制解调器	7 KB/sec
Scanner at 300 dpi 300dpi 扫描仪	1 MB/sec
Digital camcorder 数字便携式摄像机	3.5 MB/sec
4x Blu-ray disc 4倍蓝光光盘	18 MB/sec
802.11n Wireless 802.11n 无线	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800 火线 800	100 MB/sec
Gigabit Ethernet 千兆以太网	125 MB/sec
SATA 3 disk drive SATA 3磁盘驱动	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus SCSI Ultra 5总线	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus 2 总线	2.5 GB/sec
SONET OC-768 network	5 GB/sec

设备控制器

首先需要先了解一下设备控制器的概念。

设备控制器是处理 CPU 传入和传出信号的系统。设备通过插头和插座连接到计算机，并且插座连接到设备控制器。设备控制器从连接的设备处接收数据，并将其存储在控制器内部的一些 `特殊目的寄存器 (special purpose registers)` 也就是本地缓冲区中。

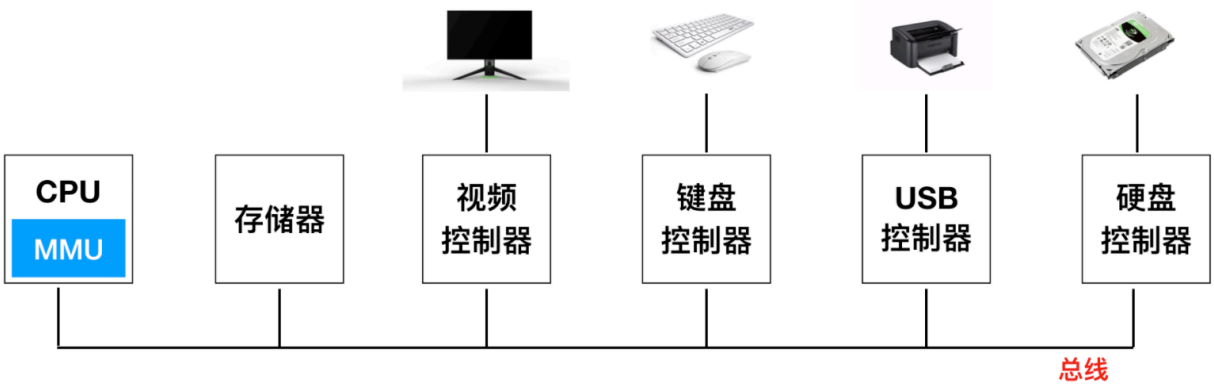
> 特殊用途寄存器，顾名思义是仅为一项任务而设计的寄存器。例如，cs，ds，gs 和其他段寄存器属于特殊目的寄存器，因为它们的存在是为了保存段号。 eax，ecx 等是一般用途的寄存器，因为你可以无限制地使用它们。 例如，你不能移动 ds，但是可以移动 eax，ebx。
>
> 通用目的寄存器比如有：eax、ecx、edx、ebx、esi、edi、ebp、esp
>
> 特殊目的寄存器比如有：cs、ds、ss、es、fs、gs、eip、flag

每个设备控制器都会有一个应用程序与之对应，设备控制器通过应用程序的接口通过中断与操作系统进行通信。设备控制器是硬件，而设备驱动程序是软件。

I/O 设备通常由 `机械组件(mechanical component)` 和 `电子组件(electronic component)` 构成。电子组件被称为 `设备控制器(device controller)` 或者 `适配器(adapter)` 。在个人计算机上，它通常采用 `可插入（PCIe）扩展插槽` 的主板上的芯片或印刷电路卡的形式。



机械设备就是它自己，它的组成如下



简单个人计算机的组件

控制器卡上通常会有一个连接器，通向设备本身的电缆可以插入到这个连接器中，很多控制器可以操作 2 个、4 个设置 8 个相同的设备。

控制器与设备之间的接口通常是一个低层次的接口。例如，磁盘可能被格式化为 2,000,000 个扇区，每个磁道 512 字节。然而，实际从驱动出来的却是一个串行的比特流，从一个 `前导符(preamble)` 开始，然后是一个扇区中的 4096 位，最后是一个 `校验和` 或 `ECC（错误码，Error-Correcting Code）` 。前导符是在对磁盘进行格式化的时候写上去的，它包括柱面数和扇区号，扇区大小以及类似的数据，此外还包含同步信息。

控制器的任务是把串行的位流转换为字节块，并进行必要的错误校正工作。字节块通常会在控制器内部的一个缓冲区按位进行组装，然后再对校验和进行校验并证明字节块没有错误后，再将它复制到内存中。

内存映射 I/O

每个控制器都会有几个寄存器用来和 CPU 进行通信。通过写入这些寄存器，操作系统可以命令设备发送数据，接收数据、开启或者关闭设备等。通过从这些寄存器中读取信息，操作系统能够知道设备的状态，是否准备接受一个新命令等。

为了控制 `寄存器` ，许多设备都会有 `数据缓冲区(data buffer)` ，来供系统进行读写。例如，在屏幕上显示一个像素的常规方法是使用一个视频 RAM，这一 RAM 基本上只是一个数据缓冲区，用来供程序和操作系统写入数据。

那么问题来了，CPU 如何与设备寄存器和设备数据缓冲区进行通信呢？存在两个可选的方式。第一种方法是，每个控制寄存器都被分配一个 `I/O 端口(I/O port)` 号，这是一个 8 位或 16 位的整数。所有 I/O 端口的集合形成了受保护的 I/O 端口空间，以便普通用户程序无法访问它（只有操作系统可以访问）。使用特殊的 I/O 指令像是

```
IN REG,PORT
```

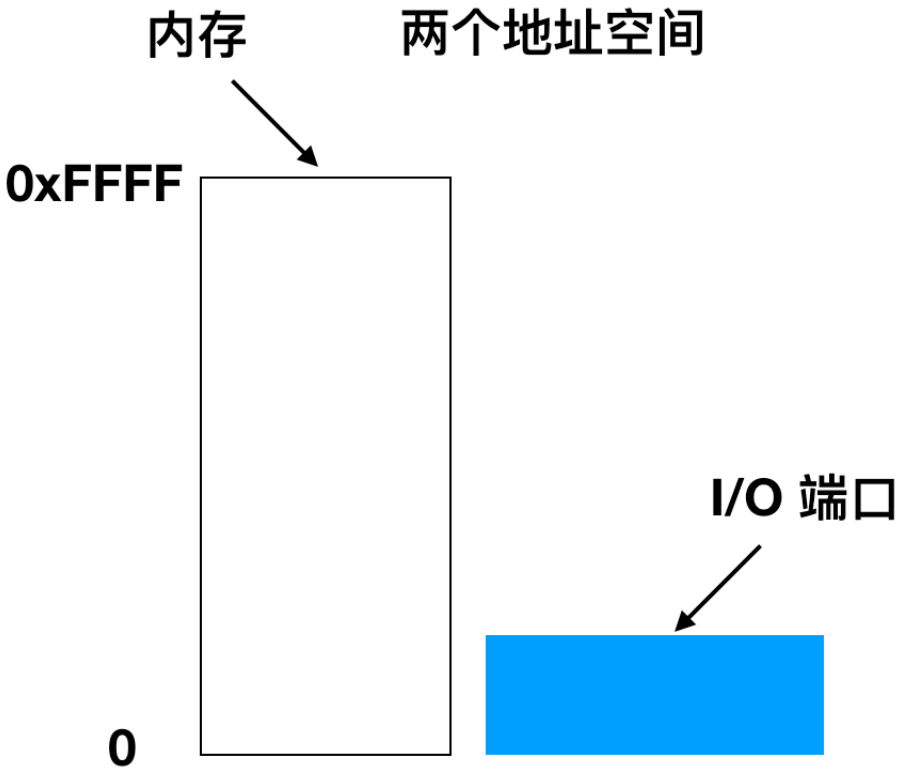
CPU 可以读取控制寄存器 PORT 的内容并将结果放在 CPU 寄存器 REG 中。类似的，使用

```
OUT PORT,REG
```

CPU 可以将 REG 的内容写到控制寄存器中。大多数早期计算机，包括几乎所有大型主机，如 IBM 360 及其所有后续机型，都是以这种方式工作的。

> 控制寄存器是一个处理器寄存器而改变或控制的一般行为 CPU 或其他数字设备。控制寄存器执行的常见任务包括中断控制，切换寻址模式，分页控制和协处理器控制。

在这一方案中，内存地址空间和 I/O 地址空间是不相同的，如下图所示



指令

```
IN R0,4
```

和

```
MOV R0,4
```

这一设计中完全不同。**前者读取 I/O端口 4 的内容并将其放入 R0，而后者读取存储器字 4 的内容并将其放入 R0**。这些示例中的 4 代表不同且不相关的地址空间。

第二个方法是 PDP-11 引入的，

> 什么是 PDP-11?



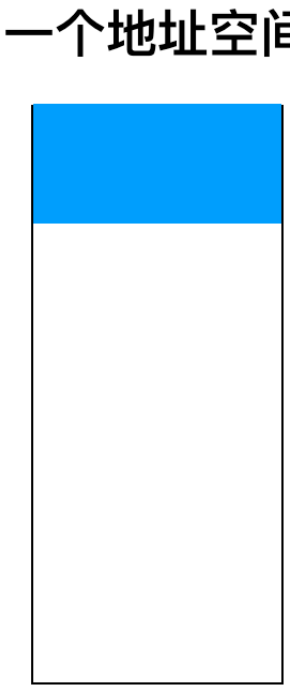
PDP-11

计算机

PDP-11为迪吉多计算机于1970到1980年代所销售的一系列16位迷你计算机。PDP-11是迪吉多计算机的PDP-8系列的后续机种。PDP-11有着许多创新的特色，而且比起其前代机种更容易撰写程序。当32位的后续扩展机型VAX-11推出时，PDP-11已经广受程序员的喜爱。

维基百科

它将**所有控制寄存器映射到内存空间**中，如下图所示

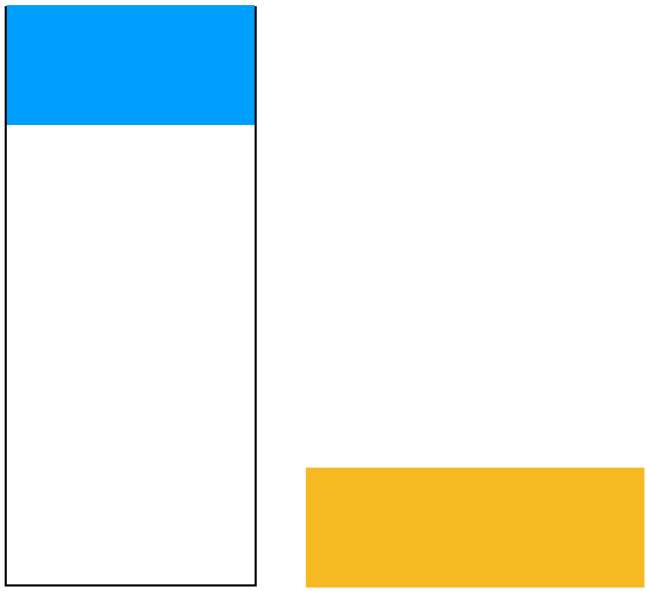


`内存映射的 I/O` 是在 CPU 与其连接的外围设备之间交换数据和指令的一种方式，这种方式是处理器和 IO 设备共享同一 `内存位置` 的内存，即处理器和 IO 设备使用内存地址进行映射。

在大多数系统中，分配给控制寄存器的地址位于或者靠近地址的顶部附近。

下面是采用的一种混合方式

两个地址空间



这种方式具有与内存映射 I/O 的数据缓冲区，而控制寄存器则具有单独的 I/O 端口。x86 采用这一体系结构。在 IBM PC 兼容机中，除了 0 到 64K - 1 的 I/O 端口之外，640 K 到 1M - 1 的内存地址保留给设备的数据缓冲区。

这些方案是如何工作的呢？当 CPU 想要读入一个字的时候，无论是从内存中读入还是从 I/O 端口读入，它都要将需要的地址放到总线地址线上，然后在总线的一条控制线上调用一个 `READ` 信号。还有第二条信号线来表明需要的是 I/O 空间还是内存空间。如果是内存空间，内存将响应请求。如果是 I/O 空间，那么 I/O 设备将响应请求。如果只有内存空间，那么每个内存模块和每个 I/O 设备都会将地址线和它所服务的地址范围进行比较。如果地址落在这一范围之内，它就会响应请求。绝对不会出现地址既分配给内存又分配给 I/O 设备，所以不会存在歧义和冲突。

内存映射 I/O 的优点和缺点

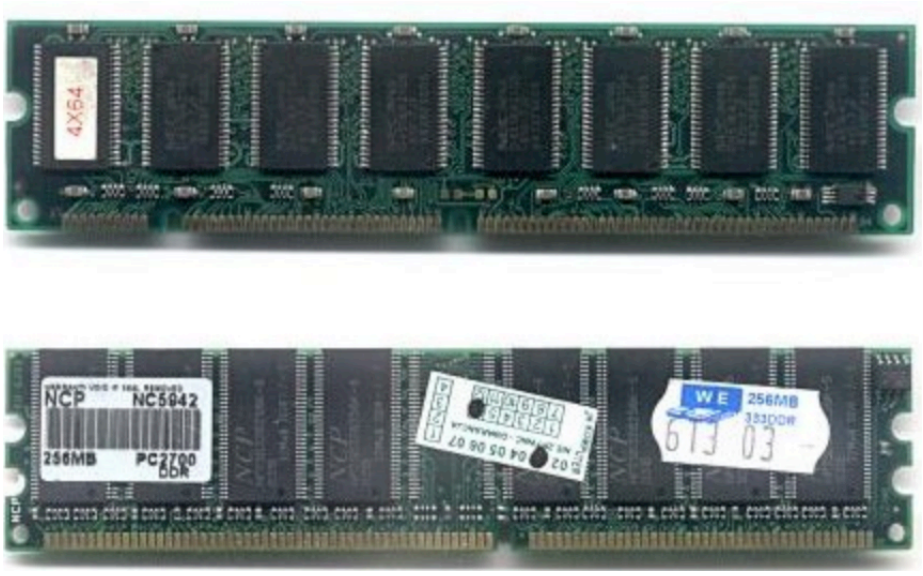
这两种寻址控制器的方案具有不同的优缺点。先来看一下内存映射 I/O 的优点。

- 第一，如果需要特殊的 I/O 指令读写设备控制寄存器，那么访问这些寄存器需要使用汇编代码，因为在 C 或 C++ 中不存在执行 `IN` 和 `OUT` 指令的方法。调用这样的过程增加了 I/O 的开销。在内存映射中，控制寄存器只是内存中的变量，在 C 语言中可以和其他变量一样进行寻址。
- 第二，对于内存映射 I/O ，不需要特殊的保护机制就能够阻止用户进程执行 I/O 操作。操作系统需要保证的是禁止把控制寄存器的地址空间放在用户的虚拟地址中就可以了。
- 第三，对于内存映射 I/O，可以引用内存的每一条指令也可以引用控制寄存器，便于引用。

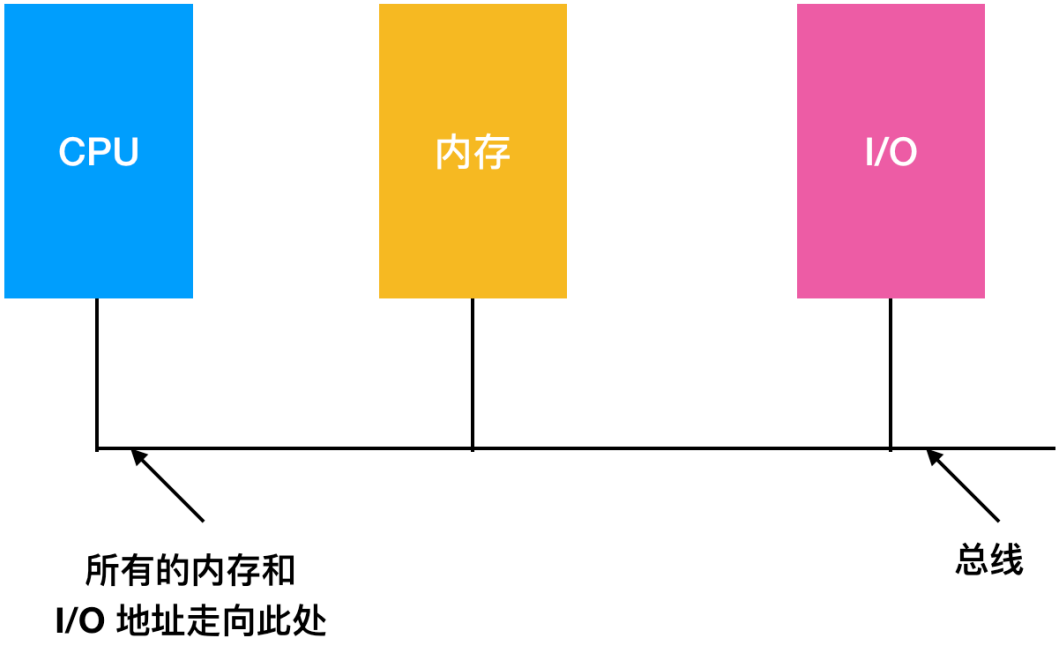
在计算机设计中，几乎所有的事情都要权衡。内存映射 I/O 也是一样，它也有自己的缺点。首先，大部分计算机现在都会有一些对于内存字的缓存。缓存一个设备控制寄存器的代价是很大的。为了避免这种内存映射 I/O 的情况，硬件必须有选择性的禁用缓存，例如，在每个页面上禁用缓存，这个功能为硬件和操作系统增加了额外的复杂性，因此必须选择性的进行管理。

第二点，如果仅仅只有一个地址空间，那么所有的 `内存模块(memory modules)` 和所有的 I/O 设备都必须检查所有的内存引用来推断出谁来进行响应。

> 什么是内存模块？在计算中，存储器模块是其上安装有存储器集成电路的印刷电路板。



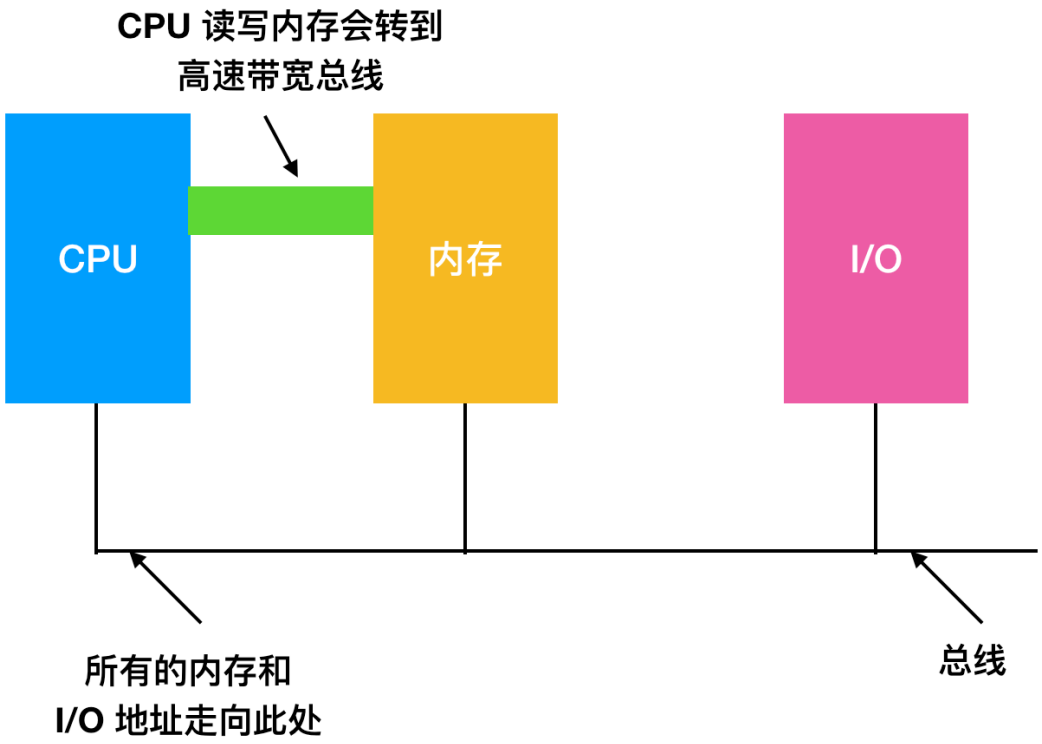
如果计算机是一种单总线体系结构的话，如下图所示



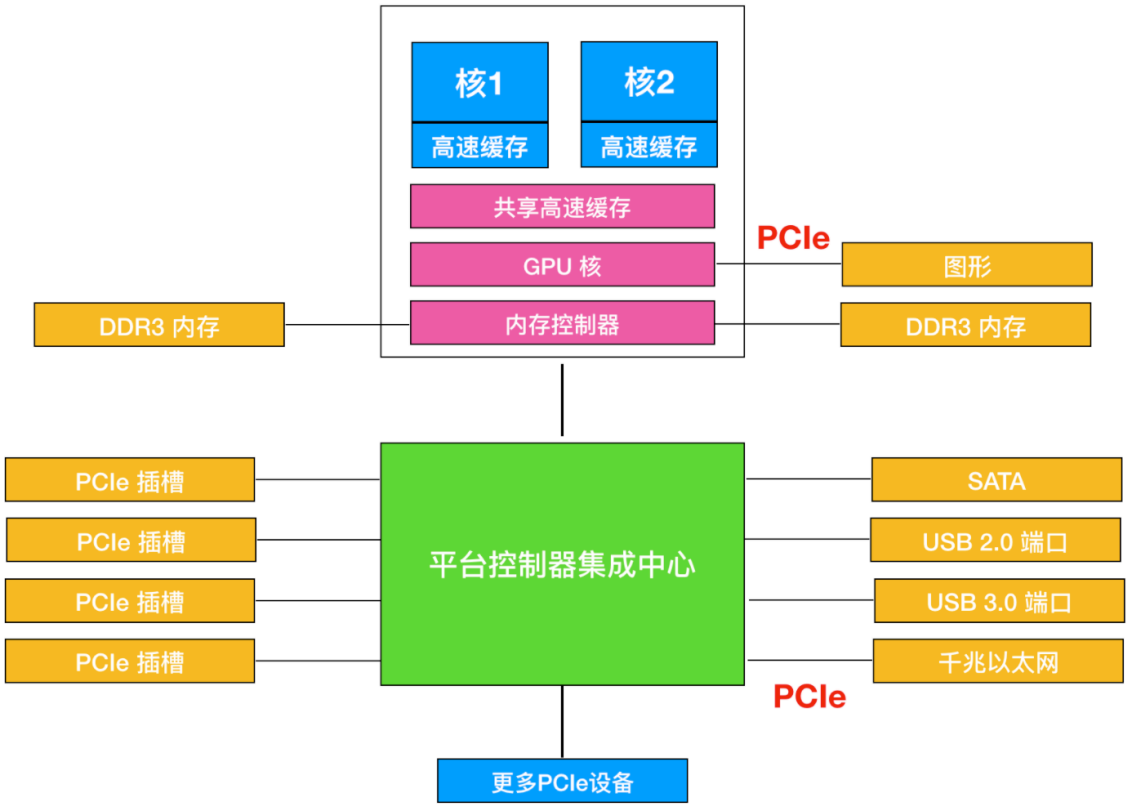
单总线体系结构

让每个内存模块和 I/O 设备查看每个地址是简单易行的。

然而，现代个人计算机的趋势是专用的高速内存总线，如下图所示



装备这一总线是为了优化内存访问速度，x86 系统还可以有多种总线（内存、PCIe、SCSI 和 USB）。如下图所示



一个大型的 x86 系统的结构

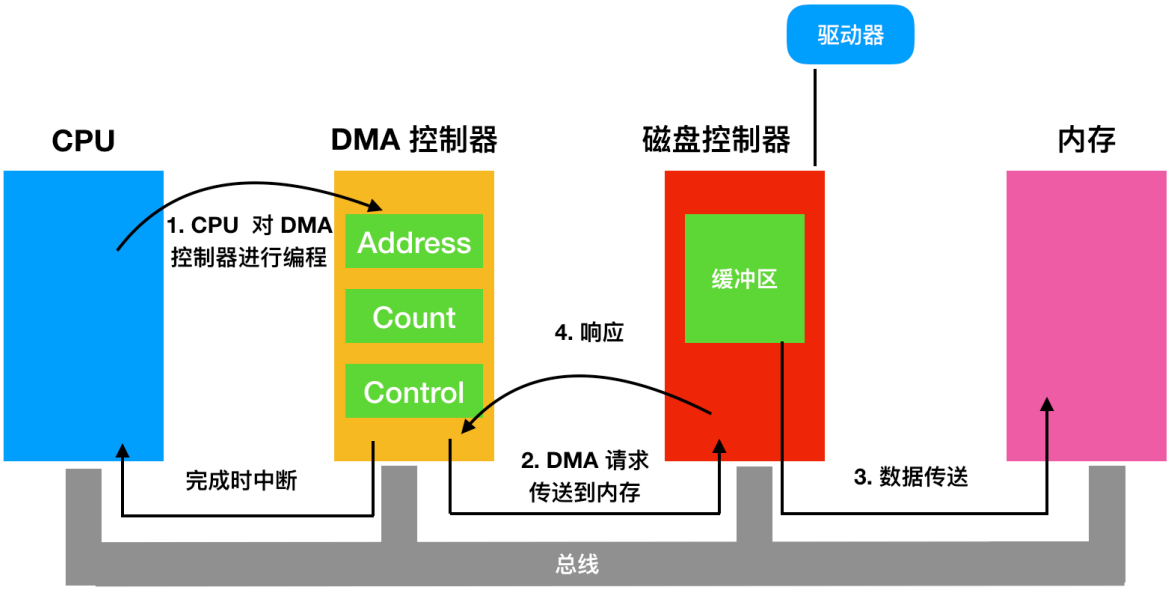
在内存映射机器上使用单独的内存总线的麻烦之处在于，I/O 设备无法通过内存总线查看内存地址，因此它们无法对其进行响应。此外，必须采取特殊的措施使内存映射 I/O 工作在具有多总线的系统上。一种可能的方法是首先将全部内存引用发送到内存，如果内存响应失败，CPU 再尝试其他总线。

第二种设计是在内存总线上放一个 探查设备 ，放过所有潜在指向所关注的 I/O 设备的地址。此处的问题是，I/O 设备可能无法以内存所能达到的速度处理请求。

第三种可能的设计是在内存控制器中对地址进行过滤，这种设计与上图所描述的设计相匹配。这种情况下，内存控制器芯片中包含在引导时预装载的范围寄存器。这一设计的缺点是需要在引导时判定哪些内存地址而不是真正的内存地址。因而，每一设计都有支持它和反对它的论据，所以折中和权衡是不可避免的。

直接内存访问

无论一个 CPU 是否具有内存映射 I/O，它都需要寻址设备控制器以便与它们交换数据。CPU 可以从 I/O 控制器每次请求一个字节的数据，但是这么做会浪费 CPU 时间，所以经常会用到一种称为 直接内存访问(Direct Memory Access) 的方案。为了简化，我们假设 CPU 通过单一的系统总线访问所有的设备和内存，该总线连接 CPU 、内存和 I/O 设备，如下图所示



DMA 传送操作

现代操作系统实际更为复杂，但是原理是相同的。如果硬件有 DMA 控制器 ，那么操作系统只能使用 DMA。有时这个控制器会集成到磁盘控制器和其他控制器中，但这种设计需要在每个设备上都装有一个分离的 DMA 控制器。单个的 DMA 控制器可用于向多个设备传输，这种传输往往同时进行。

不管 DMA 控制器的物理地址在哪，它都能够独立于 CPU 从而访问系统总线，如上图所示。它包含几个可由 CPU 读写的寄存器，其中包括一个内存地址寄存器，字节计数寄存器和一个或多个控制寄存器。控制寄存器指定要**使用的 I/O 端口、传送方向（从 I/O 设备读或写到 I/O 设备）、传送单位（每次一个字节或者每次一个字）以及在一次突发传送中要传送的字节数**。

为了解释 DMA 的工作原理，我们首先看一下不使用 DMA 该如何进行磁盘读取。

- 首先，控制器从 磁盘驱动器 串行地、一位一位的读一个块（一个或多个扇区），直到将整块信息放入控制器的内部缓冲区。
- 读取 校验和 以保证没有发生读错误。然后控制器会产生一个中断，当操作系统开始运行时，它会重复的从控制器的缓冲区中一次一个字节或者一个字地读取该块的信息，并将其存入内存中。

DMA 工作原理

当使用 DMA 后，这个过程就会变得不一样了。首先 CPU 通过设置 DMA 控制器的寄存器对它进行编程，所以 DMA 控制器知道将什么数据传送到什么地方。DMA 控制器还要向磁盘控制器发出一个命令，通知它从磁盘读数据到其内部的缓冲区并检验校验和。当有效数据位于磁盘控制器的缓冲区中时，DMA 就可以开始了。

DMA 控制器通过在总线上发出一个 `读请求` 到磁盘控制器而发起 DMA 传送，这是第二步。这个读请求就像其他读请求一样，磁盘控制器并不知道或者并不关心它是来自 CPU 还是来自 DMA 控制器。通常情况下，要写的内存地址在总线的地址线上，所以当磁盘控制器去匹配下一个字时，它知道将该字写到什么地方。写到内存就是另外一个总线循环了，这是第三步。当写操作完成时，磁盘控制器在总线上发出一个应答信号到 DMA 控制器，这是第四步。

然后，DMA 控制器会增加内存地址并减少字节数量。如果字节数量仍然大于 0 ，就会循环步骤 2 - 步骤 4 ，直到字节计数变为 0 。此时，DMA 控制器会打断 CPU 并告诉它传输已经完成了。操作系统开始运行时，它不会把磁盘块拷贝到内存中，因为它已经在内存中了。

不同 DMA 控制器的复杂程度差别很大。最简单的 DMA 控制器每次处理一次传输，就像上面描述的那样。更为复杂的情况是一次同时处理很多次传输，这样的控制器内部具有多组寄存器，每个通道一组寄存器。在传输每一个字之后，DMA 控制器就决定下一次要为哪个设备提供服务。DMA 控制器可能被设置为使用 `轮询算法` ，或者它也有可能具有一个优先级规划设计，以便让某些设备受到比其他设备更多的照顾。假如存在一个明确的方法分辨应答信号，那么在同一时间就可以挂起对不同设备控制器的多个请求。

许多总线能够以两种模式操作：**每次一字模式和块模式**。一些 DMA 控制器也能够使用这两种方式进行操作。在前一个模式中，DMA 控制器请求传送一个字并得到这个字。如果 CPU 想要使用总线，它必须进行等待。设备可能会偷偷进入并且从 CPU 偷走一个总线周期，从而轻微的延迟 CPU。这种机制称为 `周期窃取(cycle stealing)` 。

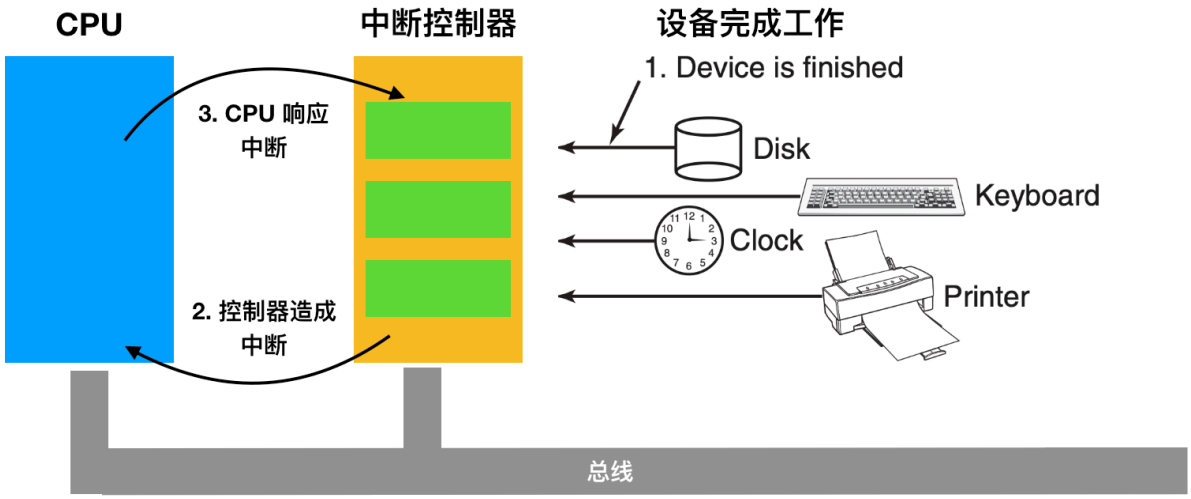
在块模式中，DMA 控制器告诉设备获取总线，然后进行一系列的传输操作，然后释放总线。这一操作的形式称为 `突发模式(burst mode)` 。这种模式要比周期窃取更有效因为获取总线占用了时间，并且一次总线获得的代价是可以同时传输多个字。缺点是如果此时进行的是长时间的突发传送，有可能将 CPU 和其他设备阻塞很长的时间。

在我们讨论的这种模型中，有时被称为 `飞越模式(fly-by mode)` ，DMA 控制器会告诉设备控制器把数据直接传递到内存。一些 DMA 控制器使用的另一种模式是让设备控制器将字发送给 DMA 控制器，然后 DMA 控制器发出第二条总线请求，将字写到任何可以写入的地方。采用这种方案，每个传输的字都需要一个额外的总线周期，但是更加灵活，因为它还可以执行设备到设备的复制，甚至是内存到内存的复制（通过事先对内存进行读取，然后对内存进行写入）。

大部分的 DMA 控制器使用物理地址进行传输。使用物理地址需要操作系统将目标内存缓冲区的虚拟地址转换为物理地址，并将该物理地址写入 DMA 控制器的地址寄存器中。另一种方案是一些 DMA 控制器将虚拟地址写入 DMA 控制器中。然后，DMA 控制器必须使用 MMU 才能完成虚拟到物理的转换。仅当 MMU 是内存的一部分而不是 CPU 的一部分时，才可以将虚拟地址放在总线上。

重温中断

在一台个人计算机体系结构中，中断结构会如下所示



中断是怎样发生的

当一个 I/O 设备完成它的工作后，它就会产生一个中断（默认操作系统已经开启中断），它通过在总线上声明已分配的信号来实现此目的。主板上的中断控制器芯片会检测到这个信号，然后执行中断操作。

如果在中断前没有其他中断操作阻塞的话，中断控制器将立刻对中断进行处理，如果在中断前还有其他中断操作 正在执行 ，或者有其他设备发出级别 更高 的中断信号的话，那么这个设备将暂时不会处理。在这种情况下，该设备会继续在总线上置起中断信号，直到得到 CPU 服务。

为了处理中断，中断控制器在地址线上放置一个数字，指定要关注的设备是哪个，并声明一个信号以中断 CPU。中断信号导致 CPU 停止当前正在做的工作并且开始做其他事情。地址线上会有一个指向 中断向量表 的索引，用来获取下一个程序计数器。这个新获取的程序计数器也就表示着程序将要开始，它会指向程序的开始处。一般情况下，陷阱和中断从这一点上看使用相同的机制，并且常常共享相同的中断向量。中断向量的位置可以硬连线到机器中，也可以位于内存中的任何位置，由 CPU 寄存器指向其起点。

中断服务程序开始运行后，中断服务程序通过将某个值写入中断控制器的 I/O 端口来确认中断。告诉它中断控制器可以自由地发出另一个中断。通过让 CPU 延迟响应来达到多个中断同时到达 CPU 涉及到竞争的情况发生。一些老的计算机没有集中的中断控制器，通常每个设备请求自己的中断。

硬件通常在服务程序开始前保存当前信息。对于不同的 CPU 来说，哪些信息需要保存以及保存在哪里差别很大。不管其他的信息是否保存，程序计数器必须要被保存，这对所有的 CPU 来说都是相同的，以此来恢复中断的进程。所有可见寄存器和大量内部寄存器也应该被保存。

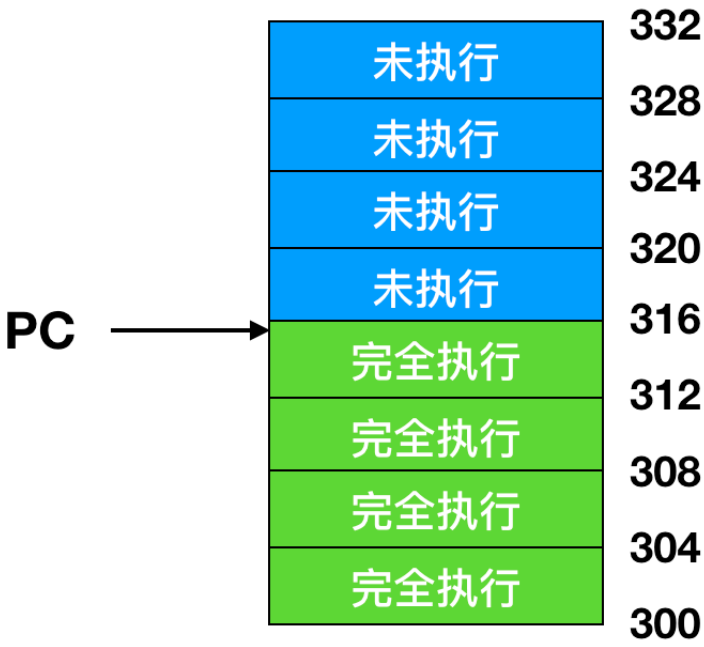
上面说到硬件应该保存当前信息，那么保存在哪里是个问题，一种选择是将其放入到内部寄存器中，在需要时操作系统可以读出这些内部寄存器。这种方法会造成的问题是：一段时间内设备无法响应，直到所有的内部寄存器中存储的信息被读出后，才能恢复运行，以免第二个内部寄存器重写内部寄存器的状态。

第二种方式是在堆栈中保存信息，这也是大部分 CPU 所使用的方式。但是，这种方法也存在问题，因为使用的堆栈不确定，如果使用的是 当前堆栈 ，则它很可能是用户进程的堆栈。堆栈指针甚至不合法，这样当硬件试图在它所指的地址处写入时，将会导致致命错误。如果使用的是内核堆栈，堆栈指针是合法的并且指向一个固定的页面，这样的机会可能会更大。然而，切换到内核态需要切换 MMU 上下文，并且可能使高速缓存或者 TLB 失效。静态或动态重新装载这些东西将增加中断处理的时间，浪费 CPU 时间。

精确中断和不精确中断

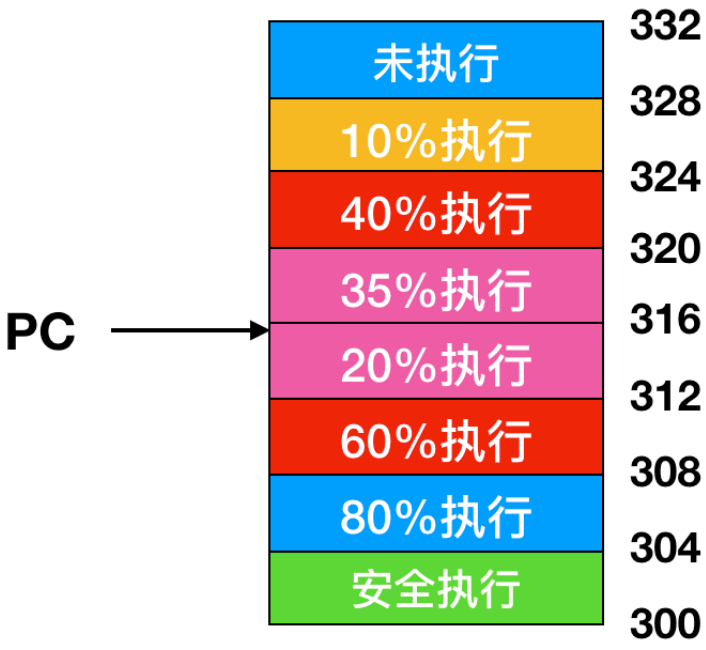
另一个问题是：现代 CPU 大量的采用 流水线 并且有时还采用 超标量(内部并行) 。在一些老的系统中，每条指令执行完毕后，微程序或硬件将检查是否存在未完成的中断。如果存在，那么程序计数器和 PSW 将被压入堆栈中开始中断序列。在中断程序运行之后，旧的 PSW 和程序计数器将从堆栈中弹出恢复先前的进程。

下面是一个流水线模型



在流水线满的时候出现一个中断会发生什么情况？许多指令正处于不同的执行阶段，中断出现时，程序计数器的值可能无法正确地反应已经执行过的指令和尚未执行的指令的边界。事实上，许多指令可能部分执行力，不同的指令完成的程度或多或少。在这种情况下，a程序计数器更有可能反应的是将要被取出并压入流水线的下一条指令的地址，而不是刚刚被执行单元处理过的指令的地址。

在超标量的设计中，可能更加糟糕



每个指令都可以分解成为微操作，微操作有可能乱序执行，这取决于内部资源（如功能单元和寄存器）的可用性。当中断发生时，某些很久以前启动的指令可能还没开始执行，而最近执行的指令可能将要马上完成。在中断信号出现时，可能存在许多指令处于不同的完成状态，它们与程序计数器之间没有什么关系。

使机器处于良好状态的中断称为 `精确中断(precise interrupt)` 。这样的中断具有四个属性：

- PC （程序计数器）保存在一个已知的地方
- PC 所指向的指令之前所有的指令已经完全执行
- PC 所指向的指令之后所有的指令都没有执行
- PC 所指向的指令的执行状态是已知的

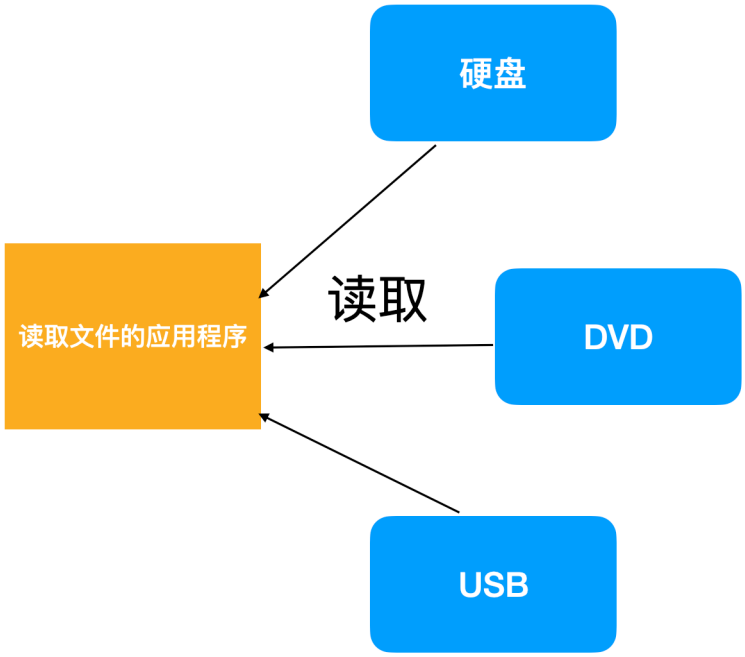
不满足以上要求的中断称为 `不精确中断(imprecise interrupt)` ，不精确中断让人很头疼。上图描述了不精确中断的现象。指令的执行时序和完成度具有不确定性，而且恢复起来也非常麻烦。

IO 软件原理

I/O 软件目标

设备独立性

现在让我们转向对 I/O 软件的研究，I/O 软件设计一个很重要的目标就是 `设备独立性(device independence)` 。啥意思呢？这意味着**我们能够编写访问任何设备的应用程序，而不用事先指定特定的设备**。比如你编写了一个能够从设备读入文件的应用程序，那么这个应用程序可以从硬盘、DVD 或者 USB 进行读入，不必再为每个设备定制应用程序。这其实就体现了设备独立性的概念。



再比如说你可以输入一条下面的指令

```
sort 输入 输出
```

那么上面这个 `输入` 就可以接收来自任意类型的磁盘或者键盘，并且 `输出` 可以写入到任意类型的磁盘或者屏幕。





计算机操作系统是这些硬件的媒介，因为不同硬件它们的指令序列不同，所以需要操作系统来做指令间的转换。

与设备独立性密切相关的一个指标就是 `统一命名(uniform naming)` 。设备的代号应该是一个整数或者是字符串，它们不应该依赖于具体的设备。在 UNIX 中，所有的磁盘都能够被集成到文件系统中，所以用户不用记住每个设备的具体名称，直接记住对应的路径即可，如果路径记不住，也可以通过 `ls` 等指令找到具体的集成位置。举个例子来说，比如一个 USB 磁盘被挂载到了 `/usr/cxuan/backup` 下，那么你把文件复制到 `/usr/cxuan/backup/device` 下，就相当于是把文件复制到了磁盘中，通过这种方式，实现了向任何磁盘写入文件都相当于是向指定的路径输出文件。

错误处理

除了 `设备独立性` 外，I/O 软件实现的第二个重要的目标就是 `错误处理(error handling)` 。通常情况下来说，错误应该交给 `硬件` 层面去处理。如果设备控制器发现了读错误的话，它会尽可能的去修复这个错误。如果设备控制器处理不了这个问题，那么设备驱动程序应该进行处理，设备驱动程序会再次尝试读取操作，很多错误都是偶然性的，如果设备驱动程序无法处理这个错误，才会把错误向上抛到硬件层面（上层）进行处理，很多时候，上层并不需要知道下层是如何解决错误的。这就很像项目经理不用把每个决定都告诉老板；程序员不用把每行代码如何写告诉项目经理。这种处理方式不够透明。

同步和异步传输

I/O 软件实现的第三个目标就是 `同步(synchronous)` 和 `异步(asynchronous，即中断驱动)` 传输。这里先说一下同步和异步是怎么回事吧。

同步传输中数据通常以块或帧的形式发送。发送方和接收方在数据传输之前应该具有 `同步时钟` 。而在异步传输中，数据通常以字节或者字符的形式发送，异步传输则不需要同步时钟，但是会在传输之前向数据添加 `奇偶校验位` 。下面是同步和异步的主要区别

比较条件	同步传输	异步传输
概念	块头序列开始	它分别在字符前面和后面使用开始位和停止位。
传输方式	以块或帧的形式发送数据	发送字节或者字符
同步方式	同步时钟	无
传输速率	同步传输比较快	异步传输比较慢
时间间隔	同步传输通常是恒定时间	异步传输时间随机
开销	同步开销比较昂贵	异步传输开销比较小
是否存在间隙	不存在	存在
实现	硬件和软件	只有硬件
示例	聊天室，视频会议，电话对话等。	信件，电子邮件，论坛

回到正题。大部分 `物理IO(physical I/O)` 是异步的。物理 I/O 中的 CPU 是很聪明的，CPU 传输完成后会转而做其他事情，它和中断心灵相通，等到中断发生后，CPU 才会回到传输这件事情上来。

> I/O 分为两种：物理I/O 和 `逻辑I/O(Logical I/O)` 。
>
> 物理 I/O 通常是从磁盘等存储设备实际获取数据。逻辑 I/O 是对存储器（块，缓冲区）获取数据。

缓冲

I/O 软件的最后一个问题是 `缓冲(buffering)` 。通常情况下，从一个设备发出的数据不会直接到达最后的设备。其间会经过一系列的校验、检查、缓冲等操作才能到达。举个例子来说，从网络上发送一个数据包，会经过一系列检查之后首先到达缓冲区，从而消除缓冲区填满速率和缓冲区过载。

共享和独占

I/O 软件引起的最后一个问题就是共享设备和独占设备的问题。有些 I/O 设备能够被许多用户共同使用。一些设备比如磁盘，让多个用户使用一般不会产生什么问题，但是某些设备必须具有独占性，即只允许单个用户使用完成后才能让其他用户使用。

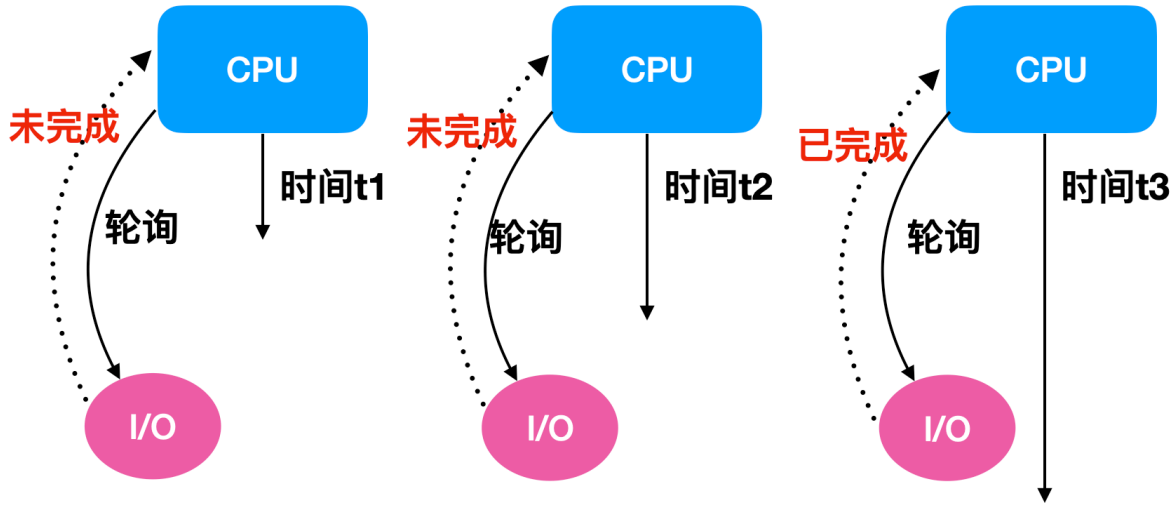
下面，我们来探讨一下如何使用程序来控制 I/O 设备。一共有三种控制 I/O 设备的方法

- 使用程序控制 I/O
- 使用中断驱动 I/O
- 使用 DMA 驱动 I/O

使用程序控制 I/O

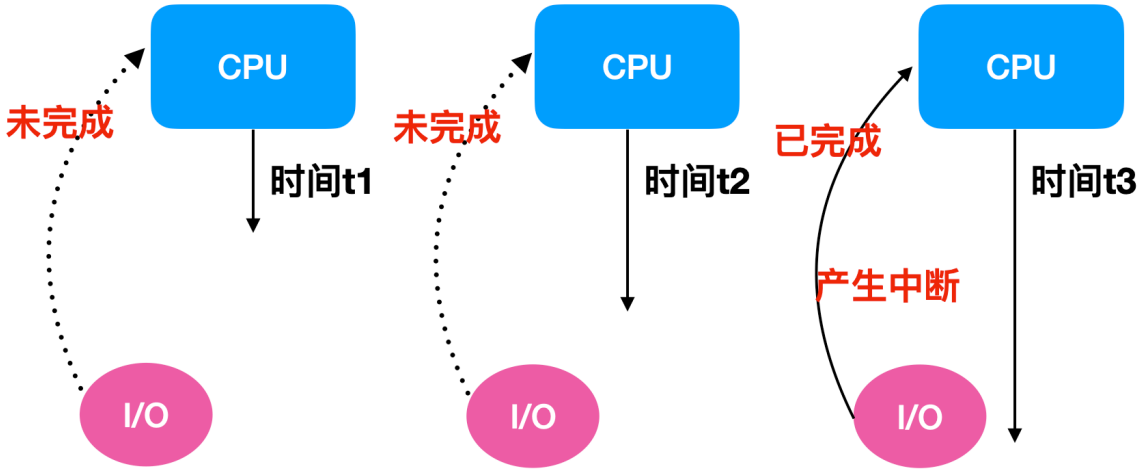
使用程序控制 I/O 又被称为 `可编程I/O`，它是指由 CPU 在驱动程序软件控制下启动的数据传输，来访问设备上的寄存器或者其他存储器。CPU 会发出命令，然后等待 I/O 操作的完成。由于 CPU 的速度比 I/O 模块的速度快很多，因此可编程 I/O 的问题在于，CPU 必须等待很长时间才能等到处理结果。CPU 在等待时会采用 `轮询(polling)` 或者 `忙等(busy waiting)` 的方式，结果，整个系统的性能被严重拉低。可编程 I/O 十分简单，如果需要等待的时间非常短的话，可编程 I/O 倒是一个很好的方式。一个可编程的 I/O 会经历如下操作

- CPU 请求 I/O 操作
- I/O 模块执行响应
- I/O 模块设置状态位
- CPU 会定期检查状态位
- I/O 不会直接通知 CPU 操作完成
- I/O 也不会中断 CPU
- CPU 可能会等待或在随后的过程中返回



使用中断驱动 I/O

鉴于上面可编程 I/O 的缺陷，我们提出一种改良方案，我们想要在 CPU 等待 I/O 设备的同时，能够做其他事情，等到 I/O 设备完成后，它就会产生一个中断，这个中断会停止当前进程并保存当前的状态。一个可能的示意图如下



尽管中断减轻了 CPU 和 I/O 设备的等待时间的负担，但是由于还需要在 CPU 和 I/O 模块之前进行大量的逐字传输，因此在大量数据传输中效率仍然很低。下面是中断的基本操作

- CPU 进行读取操作
- I/O 设备从外围设备获取数据，同时 CPU 执行其他操作
- I/O 设备中断通知 CPU
- CPU 请求数据
- I/O 模块传输数据

所以我们现在着手需要解决的就是 CPU 和 I/O 模块间数据传输的效率问题。

使用 DMA 的 I/O

DMA 的中文名称是直接内存访问，它意味着 CPU 授予 I/O 模块权限在不涉及 CPU 的情况下读取或写入内存。也就是 DMA 可以不需要 CPU 的参与。这个过程由称为 DMA 控制器（DMAC）的芯片管理。由于 DMA 设备可以直接在内存之间传输数据，而不是使用 CPU 作为中介，因此可以缓解总线上的拥塞。DMA 通过允许 CPU 执行任务，同时 DMA 系统通过系统和内存总线传输数据来提高系统并发性。

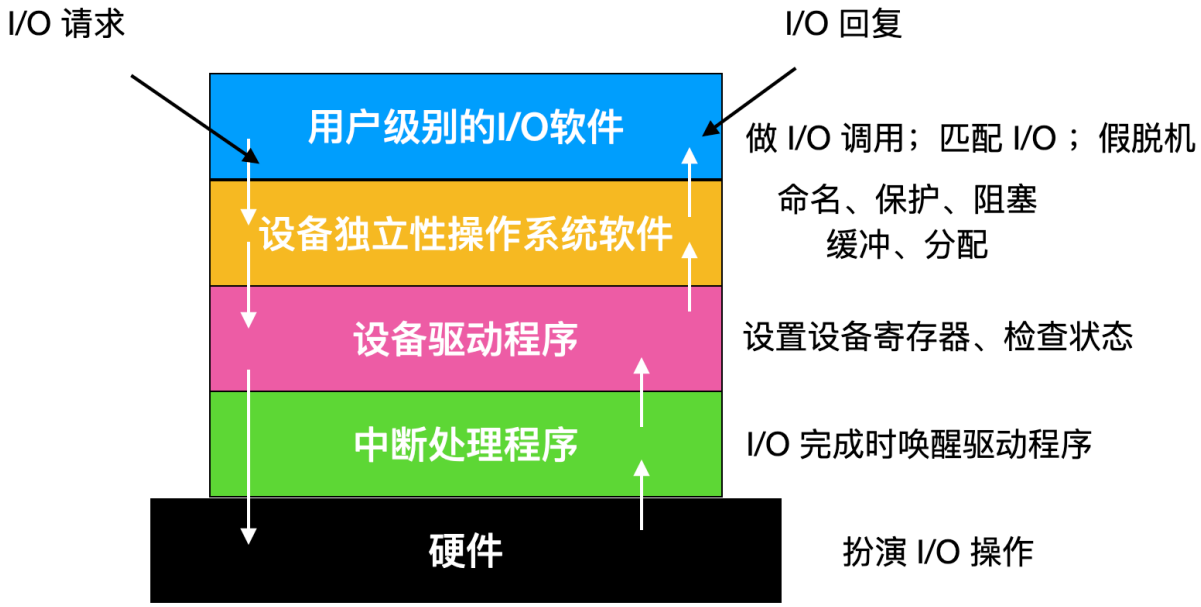
I/O 层次结构

I/O 软件通常组织成四个层次，它们的大致结构如下图所示



每一层和其上下层都有明确的功能和接口。下面我们采用和计算机网络相反的套路，即自下而上的了解一下这些程序。

下面是另一幅图，这幅图显示了输入/输出软件系统所有层及其主要功能。



下面我们具体的来探讨一下上面的层次结构

中断处理程序

在计算机系统中，中断就像女人的脾气一样无时无刻都在产生，中断的出现往往是让人很不爽的。中断处理程序又被称为 `中断服务程序` 或者是 `ISR(Interrupt Service Routines)`，它是最靠近硬件的一层。中断处理程序由硬件中断、软件中断或者是软件异常启动产生的中断，用于实现设备驱动程序或受保护的操作模式（例如系统调用）之间的转换。

中断处理程序负责处理中断发生时的所有操作，操作完成后阻塞，然后启动中断驱动程序来解决阻塞。通常会有三种通知方式，依赖于不同的具体实现

- 信号量实现中：在信号量上使用 `up` 进行通知；
- 管程实现：对管程中的条件变量执行 `signal` 操作
- 还有一些情况是发送一些消息

不管哪种方式都是为了让阻塞的中断处理程序恢复运行。

中断处理方案有很多种，下面是 《**ARM System Developer's Guide**

Designing and Optimizing System Software》列出来的一些方案

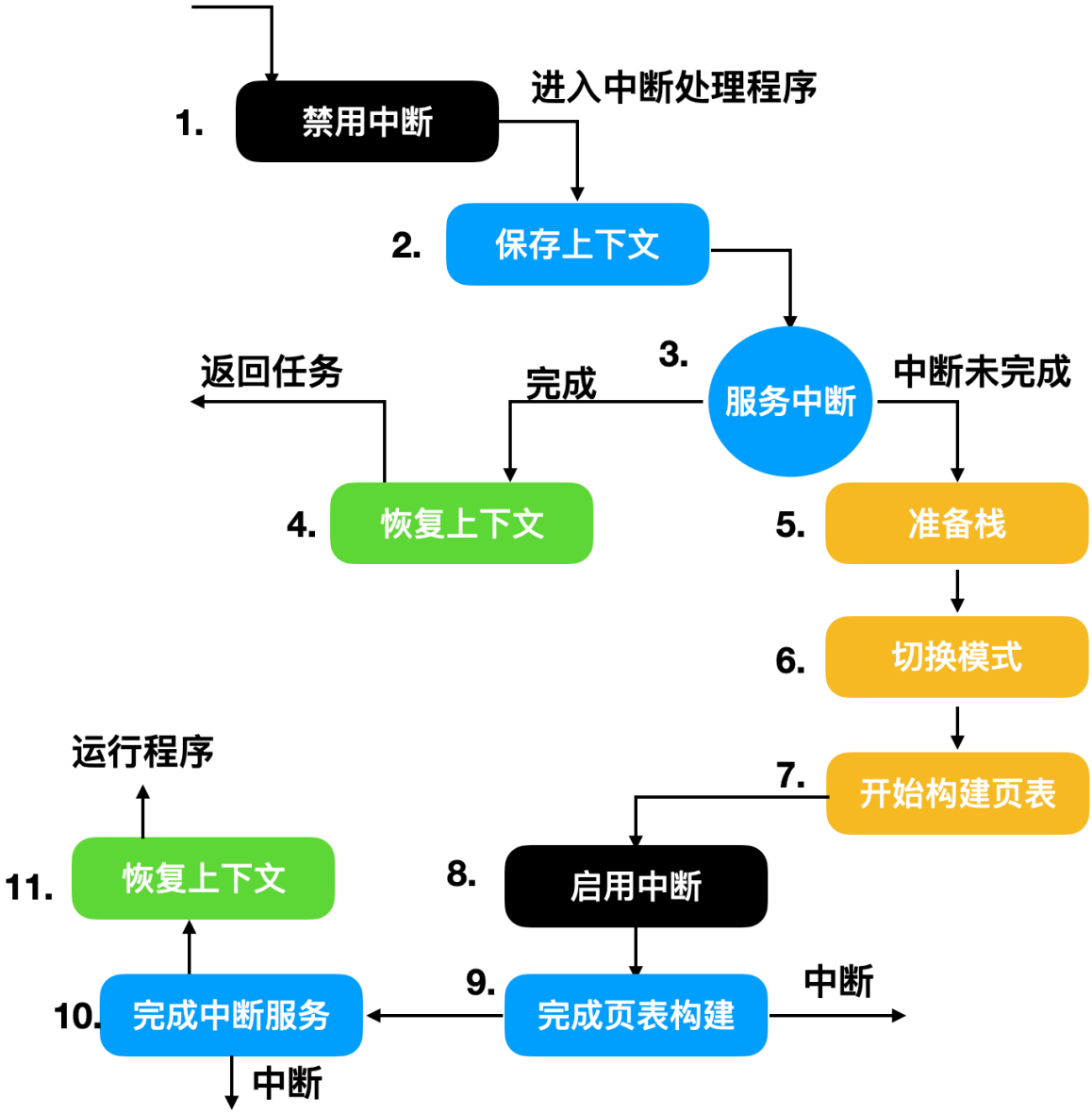
- `非嵌套` 的中断处理程序按照顺序处理各个中断，非嵌套的中断处理程序也是最简单的中断处理
- `嵌套` 的中断处理程序会处理多个中断而无需分配优先级
- `可重入` 的中断处理程序可使用优先级处理多个中断
- `简单优先级` 中断处理程序可处理简单的中断
- `标准优先级` 中断处理程序比低优先级的中断处理程序在更短的时间能够处理优先级更高的中断
- `高优先级` 中断处理程序在短时间能够处理优先级更高的任务，并直接进入特定的服务例程。
- `优先级分组` 中断处理程序能够处理不同优先级的中断任务

下面是一些通用的中断处理程序的步骤，不同的操作系统实现细节不一样

- 保存所有没有被中断硬件保存的寄存器
- 为中断服务程序设置上下文环境，可能包括设置 `TLB` 、 `MMU` 和页表，如果不太了解这三个概念，请参考另外一篇文章

- 为中断服务程序设置栈
- 对中断控制器作出响应，如果不存在集中的中断控制器，则继续响应中断
- 把寄存器从保存它的地方拷贝到进程表中
- 运行中断服务程序，它会从发出中断的设备控制器的寄存器中提取信息
- 操作系统会选择一个合适的进程来运行。如果中断造成了一些优先级更高的进程变为就绪态，则选择运行这些优先级高的进程
- 为进程设置 MMU 上下文，可能也会需要 TLB，根据实际情况决定
- 加载进程的寄存器，包括 PSW 寄存器
- 开始运行新的进程

上面我们罗列了一些大致的中断步骤，不同性质的操作系统和中断处理程序能够处理的中断步骤和细节也不尽相同，下面是一个嵌套中断的具体运行步骤

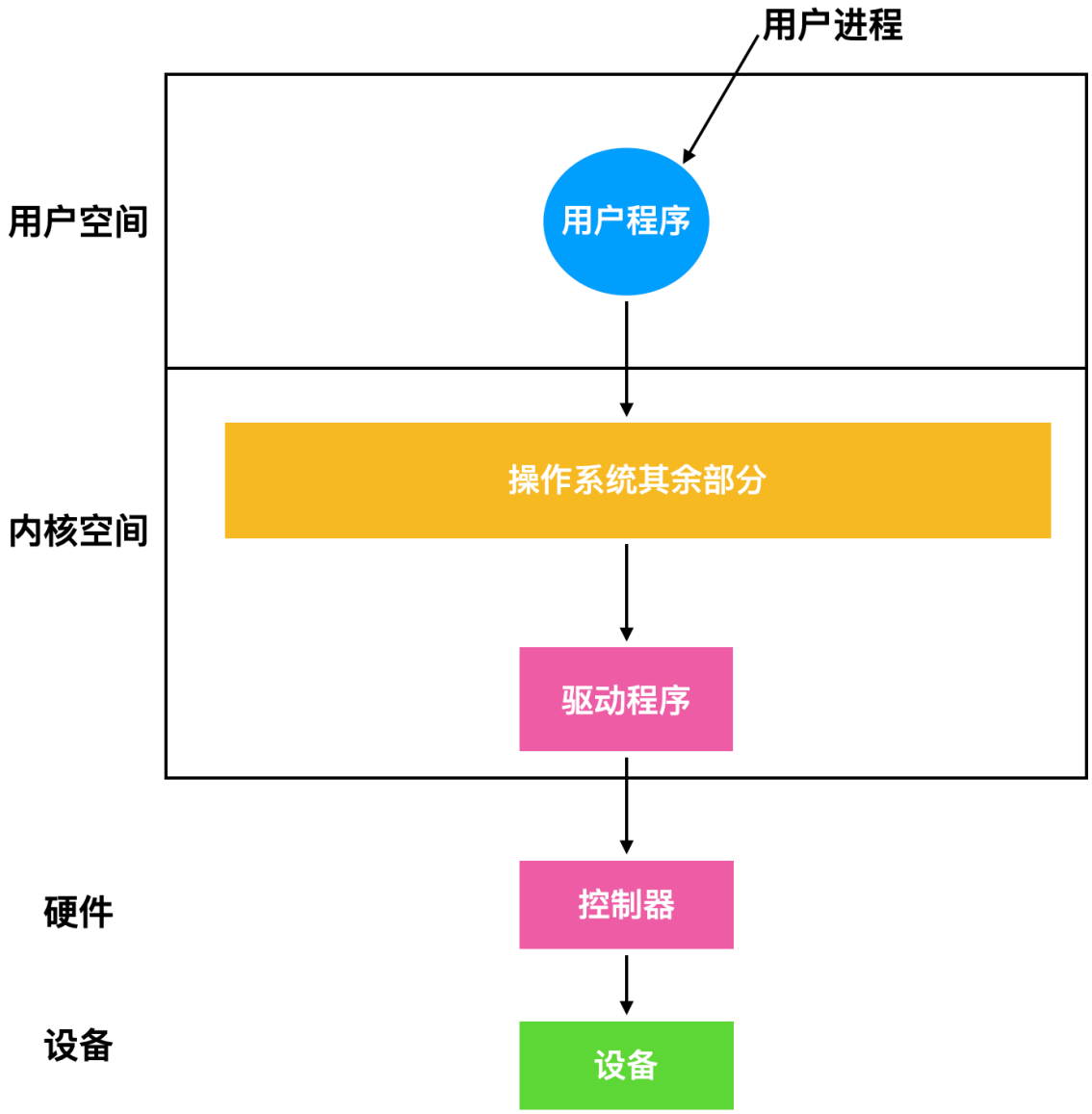


设备驱动程序

在上面的文章中我们知道了设备控制器所做的工作。我们知道每个控制器其内部都会有寄存器用来和设备进行沟通，发送指令，读取设备的状态等。

因此，每个连接到计算机的 I/O 设备都需要有某些特定设备的代码对其进行控制，例如鼠标控制器需要从鼠标接受指令，告诉下一步应该移动到哪里，键盘控制器需要知道哪个按键被按下等。这些提供 I/O 设备到设备控制器转换的过程的代码称为 `设备驱动程序(Device driver)` 。

为了能够访问设备的硬件，实际上也就意味着，设备驱动程序通常是操作系统内核的一部分，至少现在的体系结构是这样的。但是也可以构造 `用户空间` 的设备驱动程序，通过系统调用来完成读写操作。这样就避免了一个问题，有问题的驱动程序会干扰内核，从而造成崩溃。所以，在用户控件实现设备驱动程序是构造系统稳定性一个非常有用的措施。 `MINIX 3` 就是这么做的。下面是 MINI 3 的调用过程



然而，大多数桌面操作系统要求驱动程序必须运行在内核中。

操作系统通常会将驱动程序归为 `字符设备` 和 `块设备` ，我们上面也介绍过了

块设备是一个能存储固定大小块信息的设备，它支持以固定大小的块，扇区或群集读取和（可选）写入数据。
每个块都有自己的物理地址。通常块的大小在 512 - 65536 之间。
所有传输的信息都会以连续的块为单位。块设备的基本特征是每个块都较为对立，能够独立的进行读写。
常见的块设备有 硬盘、蓝光光盘、USB 盘

块设备



另一类 I/O 设备是字符设备。字符设备以字符为单位发送或接收一个字符流，而不考虑任何块结构。
字符设备是不可寻址的，也没有任何寻道操作。
常见的字符设备有 打印机、网络设备、鼠标、以及大多数与磁盘不同的设备。

字符设备



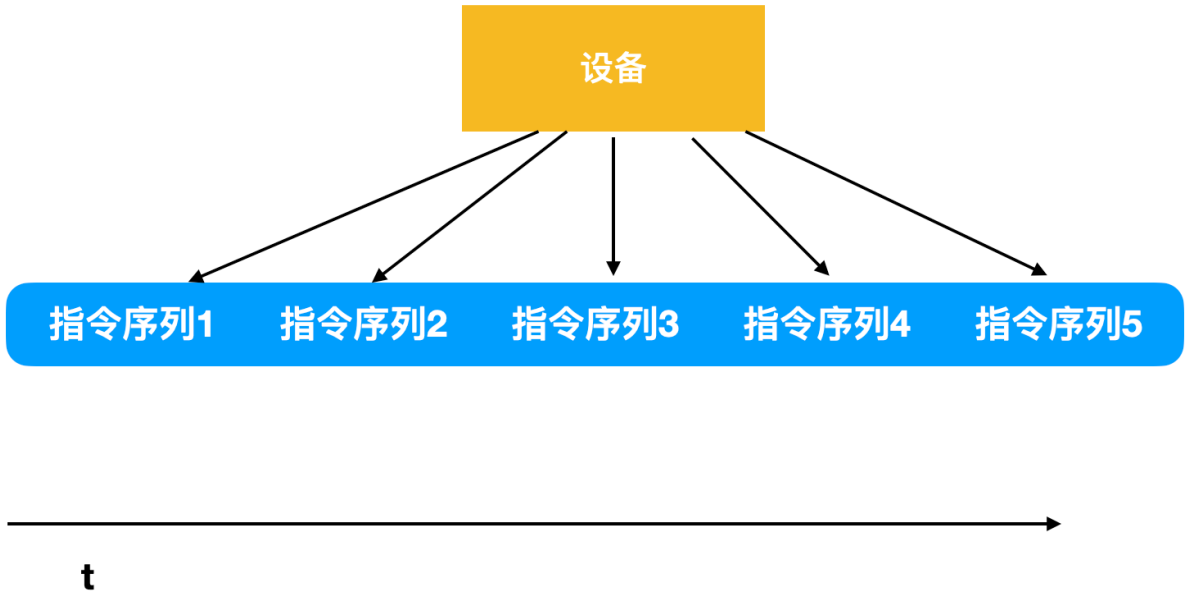
在 UNIX 系统中，操作系统是一个 二进制程序 ，包含需要编译到其内部的所有驱动程序，如果你要对 UNIX 添加一个新设备，需要重新编译内核，将新的驱动程序装到二进制程序中。

然而随着大多数个人计算机的出现，由于 I/O 设备的广泛应用，上面这种静态编译的方式不再有效，因此，从 MS-DOS 开始，操作系统转向驱动程序在执行期间动态的装载到系统中。

设备驱动程序具有很多功能，比如接受读写请求，对设备进行初始化、管理电源和日志、对输入参数进行有效性检查等。

设备驱动程序接受到读写请求后，会检查当前设备是否在使用，如果设备在使用，请求被排入队列中，等待后续的处理。如果此时设备是空闲的，驱动程序会检查硬件以了解请求是否能够被处理。在传输开始前，会启动设备或者马达。等待设备就绪完成，再进行实际的控制。**控制设备就是对设备发出指令**。

发出命令后，设备控制器便开始将它们写入控制器的 设备寄存器 。在将每个命令写入控制器后，会检查控制器是否接受了这条命令并准备接受下一个命令。一般控制设备会发出一系列的指令，这称为 指令序列 ，设备控制器会依次检查每个命令是否被接受，下一条指令是否能够被接收，直到所有的序列发出为止。



发出指令后，一般会有两种可能出现的情况。在大多数情况下，设备驱动程序会进行等待直到控制器完成它的事情。这里需要了解一下设备控制器的概念

> 设备控制器的主要主责是**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**。
>
> 设备控制器接收从 CPU 发送过来的指令，继而达到控制硬件的目的

设备控制器是一个 可编址 的设备，当它仅控制一个设备时，它只有一个唯一的设备地址；如果设备控制器控制多个可连接设备时，则应含有多个设备地址，并使每一个设备地址对应一个设备。

设备控制器主要分为两种：字符设备和块设备

设备控制器的主要功能有下面这些

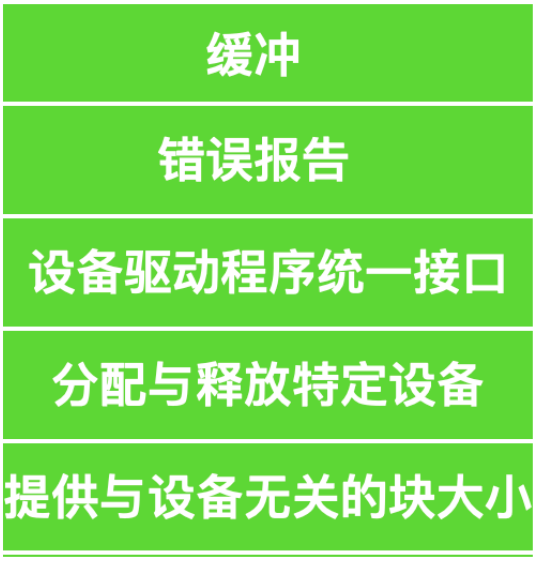
- 接收和识别命令：设备控制器可以接受来自 CPU 的指令，并进行识别。设备控制器内部也会有寄存器，用来存放指令和参数
- 进行数据交换：CPU、控制器和设备之间会进行数据的交换，CPU 通过总线把指令发送给控制器，或从控制器中并行地读出数据；控制器将数据写入指定设备。
- 地址识别：每个硬件设备都有自己的地址，设备控制器能够识别这些不同的地址，来达到控制硬件的目的，此外，为使 CPU 能向寄存器中写入或者读取数据，这些寄存器都应具有唯一的地址。
- 差错检测：设备控制器还具有对设备传递过来的数据进行检测的功能。

在这种情况下，设备控制器会阻塞，直到中断来解除阻塞状态。还有一种情况是操作是可以无延迟的完成，所以驱动程序不需要阻塞。在第一种情况下，操作系统可能被中断唤醒；第二种情况下操作系统不会被休眠。

设备驱动程序必须是 可重入 的，因为设备驱动程序会阻塞和唤醒然后再次阻塞。驱动程序不允许进行系统调用，但是它们通常需要与内核的其余部分进行交互。

与设备无关的 I/O 软件

I/O 软件有两种，一种是我们上面介绍过的基于特定设备的，还有一种是 设备无关性 的，设备无关性也就是不需要特定的设备。设备驱动程序与设备无关的软件之间的界限取决于具体的系统。下面显示的功能由设备无关的软件实现

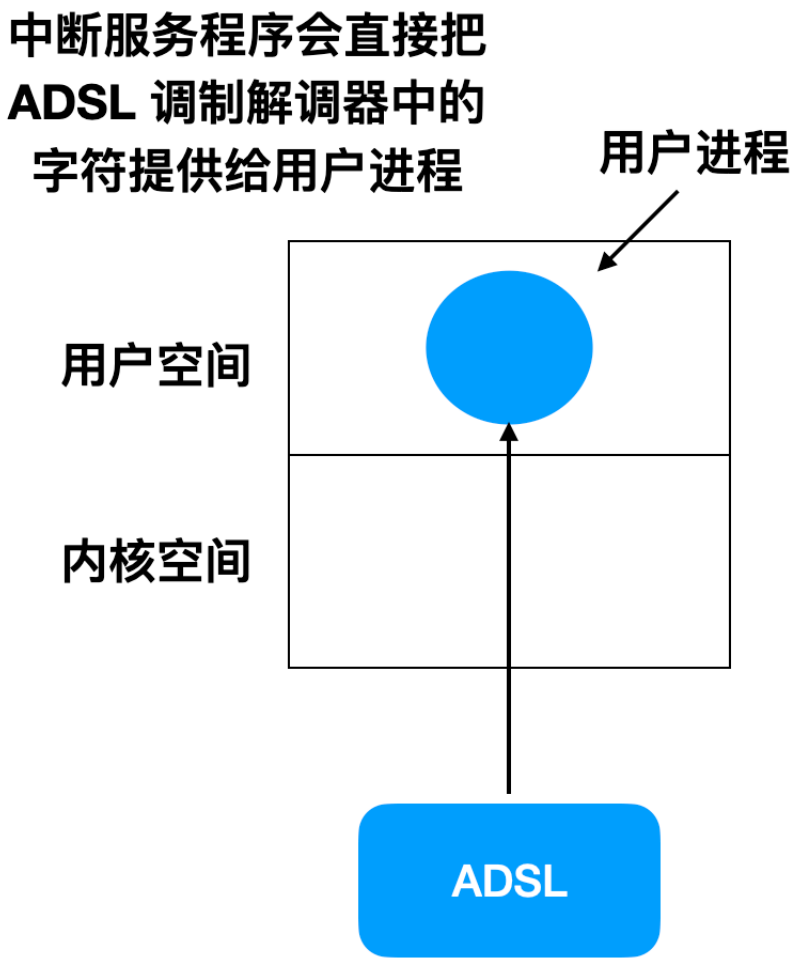


与设备无关的软件的基本功能是对所有设备执行公共的 I/O 功能，并且向用户层软件提供一个统一的接口。

缓冲

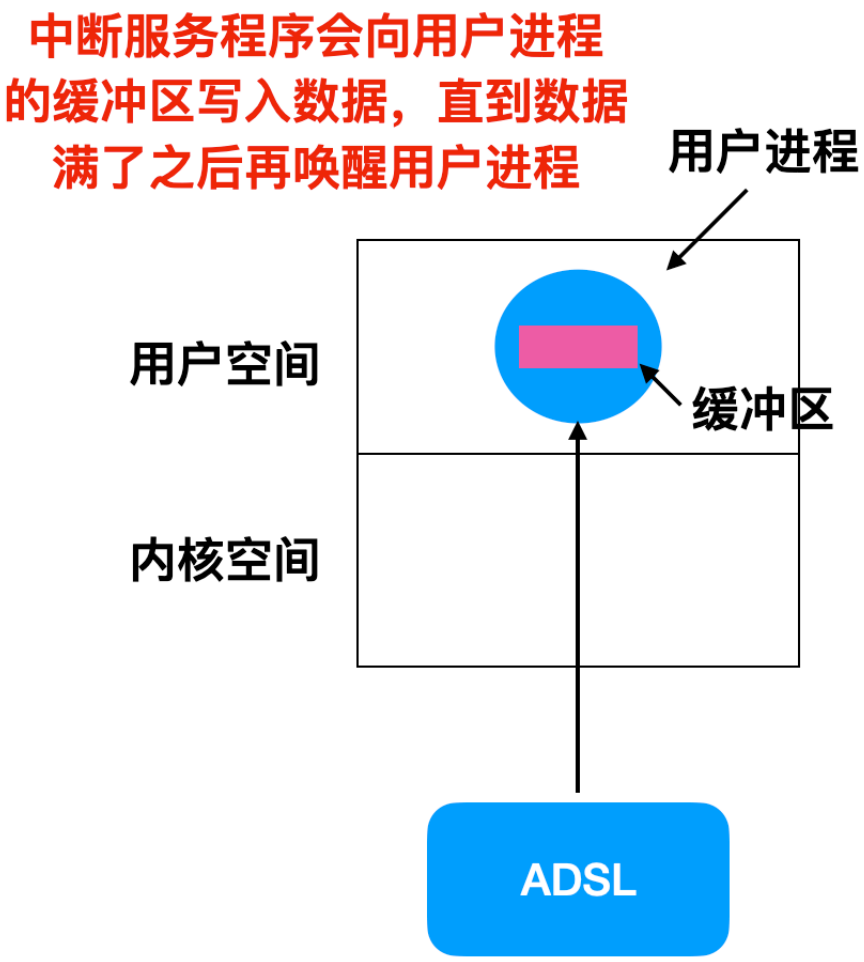
无论是对于块设备还是字符设备来说，缓冲都是一个非常重要的考量标准。下面是从 `ADSL(调制解调器)` 读取数据的过程，调制解调器是我们用来联网的设备。

用户程序调用 read 系统调用阻塞用户进程，等待字符的到来，这是对到来的字符进行处理的一种方式。每一个到来的字符都会造成中断。 `中断服务程序` 会给用户进程提供字符，并解除阻塞。将字符提供给用户程序后，进程会去读取其他字符并继续阻塞，这种模型如下



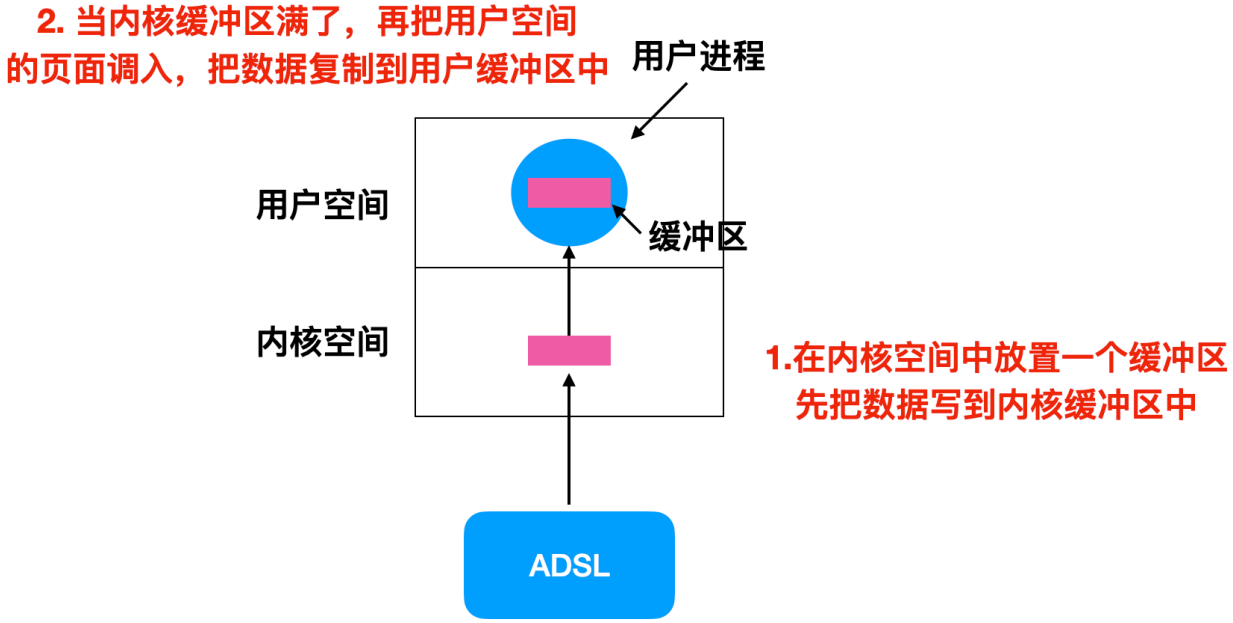
这一种方案是没有缓冲区的存在，因为用户进程如果读不到数据会阻塞，直到读到数据为止，这种情况效率比较低，而且阻塞式的方式，会直接阻止用户进程做其他事情，这对用户来说是不能接受的。还有一种情况就是每次用户进程都会重启，对于每个字符的到来都会重启用户进程，这种效率会严重降低，所以无缓冲区的软件不是一个很好的设计。

作为一个改良点，我们可以尝试在用户空间中使用一个能读取 n 个字节缓冲区来读取 n 个字符。这样的话，中断服务程序会把字符放到缓冲区中直到缓冲区变满为止，然后再去唤醒用户进程。这种方案要比上面的方案改良很多。



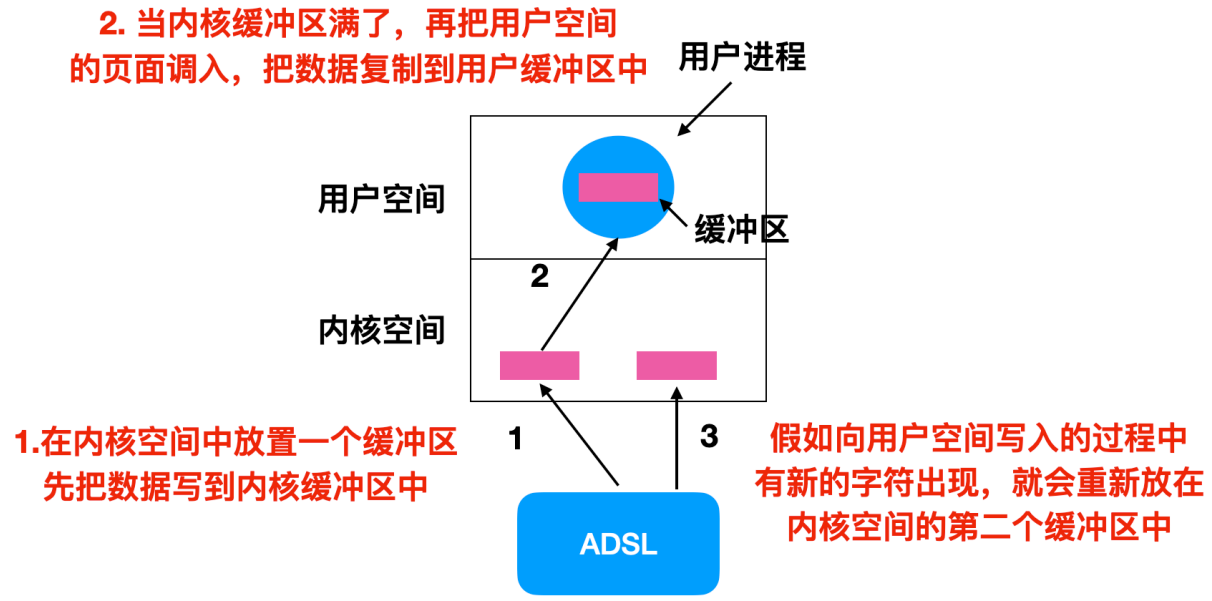
但是这种方案也存在问题，当字符到来时，如果缓冲区被调出内存会出现什么问题？解决方案是把缓冲区锁定在内存中，但是这种方案也会出现问题，如果少量的缓冲区被锁定还好，如果大量的缓冲区被锁定在内存中，那么可以换进换出的页面就会收缩，造成系统性能的下降。

一种解决方案是在 内核 中内部创建一块缓冲区，让中断服务程序将字符放在内核内部的缓冲区中。



当内核中的缓冲区要满的时候，会将用户空间中的页面调入内存，然后将内核空间的缓冲区复制到用户空间的缓冲区中，这种方案也面临一个问题就是假如用户空间的页面被换入内存，此时内核空间的缓冲区已满，这时候仍有新的字符到来，这个时候会怎么办？因为缓冲区满了，没有空间来存储新的字符了。

一种非常简单的方式就是再设置一个缓冲区就行了，在第一个缓冲区填满后，在缓冲区清空前，使用第二个缓冲区，这种解决方式如下

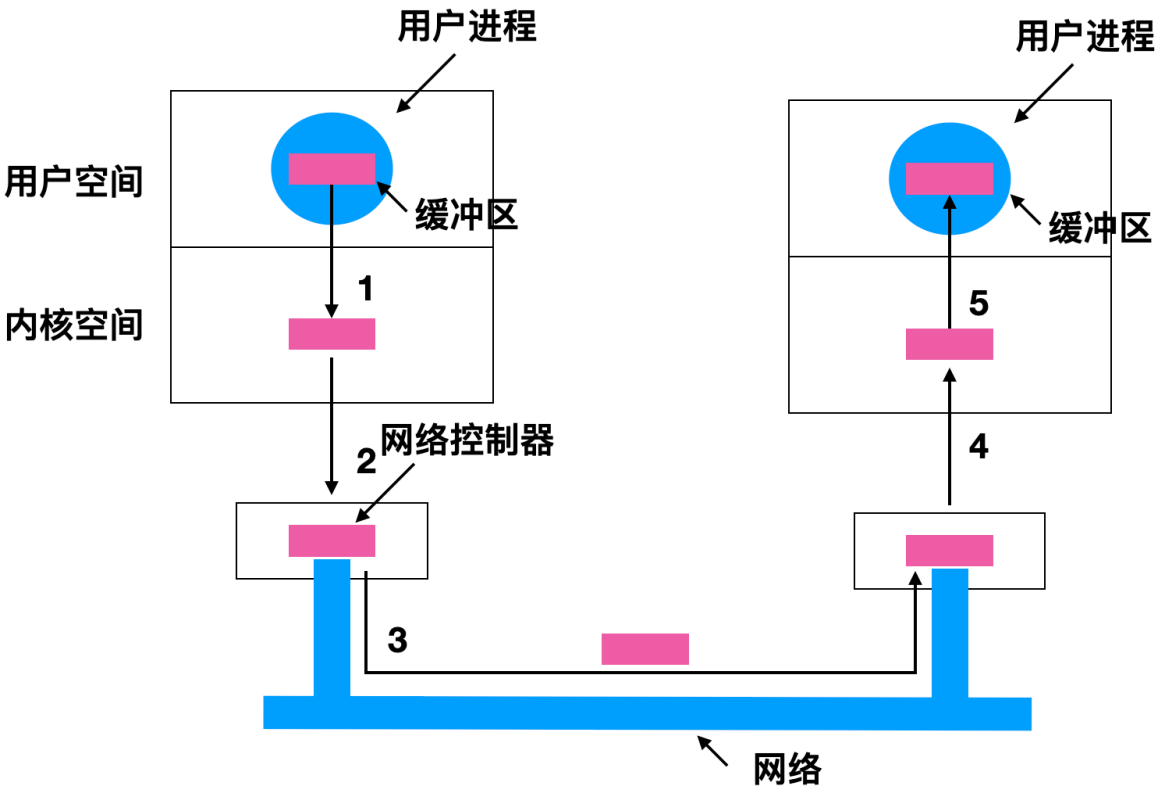


当第二个缓冲区也满了的时候，它也会把数据复制到用户空间中，然后第一个缓冲区用于接受新的字符。这种具有两个缓冲区的设计被称为 `双缓冲(double buffering)` 。

还有一种缓冲形式是 `循环缓冲(circular buffer)` 。它由一个内存区域和两个指针组成。一个指针指向下一个空闲字，新的数据可以放在此处。另外一个指针指向缓冲区中尚未删除数据的第一个字。在许多情况下，硬件会在添加新的数据时，移动第一个指针；而操作系统会在删除和处理无用数据时会移动第二个指针。两个指针到达顶部时就回到底部重新开始。

缓冲区对输出来说也很重要。对输出的描述和输入相似

缓冲技术应用广泛，但它也有缺点。如果数据被缓冲次数太多，会影响性能。考虑例如如下这种情况，



数据经过用户进程 -> 内核空间 -> 网络控制器，这里的网络控制器应该就相当于是 socket 缓冲区，然后发送到网络上，再到接收方的网络控制器 -> 接收方的内核缓冲 -> 接收方的用户缓冲，一条数据包被缓存了太多次，很容易降低性能。

错误处理

在 I/O 中，出错是一种再正常不过的情况了。当出错发生时，操作系统必须尽可能处理这些错误。有一些错误是只有特定的设备才能处理，有一些是由框架进行处理，这些错误和特定的设备无关。

I/O 错误的一类是程序员 `编程` 错误，比如还没有打开文件前就读流，或者不关闭流导致内存溢出等等。这类问题由程序员处理；另外一类是实际的 I/O 错误，例如向一个磁盘坏块写入数据，无论怎么写都写入不了。这类问题由驱动程序处理，驱动程序处理不了交给硬件处理，这个我们上面也说过。

设备驱动程序统一接口

我们在操作系统概述中说到，操作系统一个非常重要的功能就是屏蔽了硬件和软件的差异性，为硬件和软件提供了统一的标准，这个标准还体现在为设备驱动程序提供统一的接口，因为不同的硬件和厂商编写的设备驱动程序不同，所以如果为每个驱动程序都单独提供接口的话，这样没法搞，所以必须统一。

分配和释放

一些设备例如打印机，它只能由一个进程来使用，这就需要操作系统根据实际情况判断是否能够对设备的请求进行检查，判断是否能够接受其他请求，一种比较简单直接的方式是在特殊文件上执行 `open` 操作。如果设备不可用，那么直接 open 会导致失败。还有一种方式是不直接导致失败，而是让其阻塞，等到另外一个进程释放资源后，在进行 open 打开操作。这种方式就把选择权交给了用户，由用户判断是否应该等待。

> 注意：阻塞的实现有多种方式，有阻塞队列等

设备无关的块

不同的磁盘会具有不同的扇区大小，但是软件不会关心扇区大小，只管存储就是了。一些字符设备可以一次一个字节的交付数据，而其他的设备则以较大的单位交付数据，这些差异也可以隐藏起来。

用户空间的 I/O 软件

虽然大部分 I/O 软件都在内核结构中，但是还有一些在用户空间实现的 I/O 软件，凡事没有绝对。一些 I/O 软件和库过程在用户空间存在，然后以提供系统调用的方式实现。

盘

盘可以说是硬件里面比较简单的构造了，同时也是最重要的。下面我们从盘谈起，聊聊它的物理构造

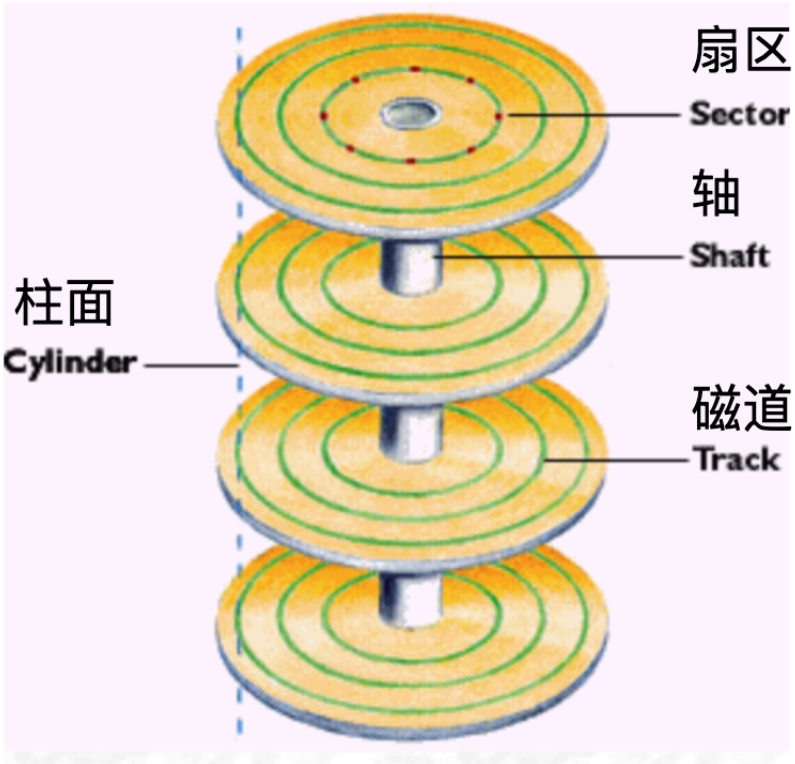
盘硬件

盘会有很多种类型。其中最简单的构造就是 `磁盘(magnetic hard disks)`， 也被称为 `hard disk,HDD` 等。磁盘通常与安装在磁臂上的磁头配对，磁头可将数据读取或者将数据写入磁盘，因此磁盘的读写速度都同样快。在磁盘中，数据是随机访问的，这也就说明可以通过任意的顺序来 `存储` 和 `检索` 单个数据块，所以你可以在任意位置放置磁盘来让磁头读取，磁盘是一种 `非易失性` 的设备，即使断电也能永久保留。

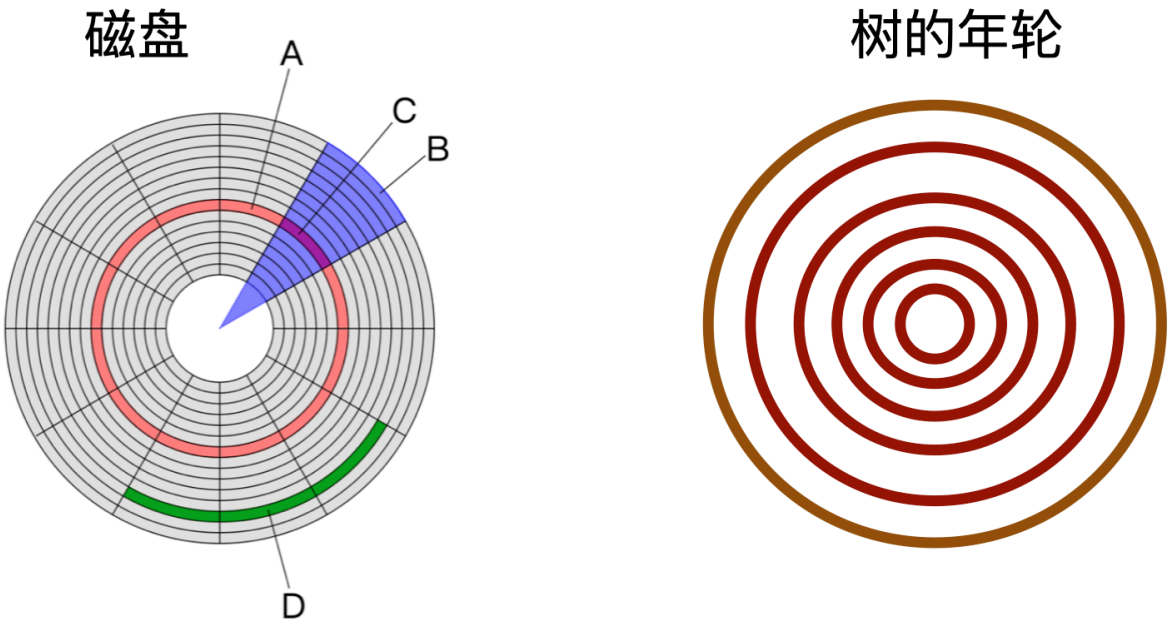
在计算机发展早期一般是用光盘来存储数据的，然而随着固态硬盘的流行，固态硬盘不包含运动部件的特点，成为现在计算机的首选存储方式。

磁盘

为了组织和检索数据，会将磁盘组织成特定的结构，这些特定的结构就是**磁道、扇区和柱面**



每一个磁盘都是由无数个同心圆组成，这些同心圆就好像树的年轮一样



> 部分树的年轮照片都要付费下载了，不敢直接白嫖，阔怕阔怕。

磁盘被组织成柱面形式，每个盘用轴相连，每一个柱面包含若干磁道，每个磁道由若干扇区组成。软盘上大约每个磁道有 8 - 32 个扇区，硬盘上每条磁道上扇区的数量可达几百个，磁头大约是 1 - 16 个。

对于磁盘驱动程序来说，一个非常重要的特性就是控制器是否能够同时控制两个或者多个驱动器进行磁道寻址，这就是 `重叠寻道(overlapped seek)` 。对于控制器来说，它能够控制一个磁盘驱动程序完成寻道操作，同时让其他驱动程序等待寻道结束。控制器也可以在一个驱动程序上进行读写草哦做，与此同时让另外的驱动器进行寻道操作，但是软盘控制器不能在两个驱动器上进行读写操作。

RAID

RAID 称为 `磁盘冗余阵列` ，简称 `磁盘阵列` 。利用虚拟化技术把多个硬盘结合在一起，成为一个或多个磁盘阵列组，目的是提升性能或数据冗余。

RAID 有不同的级别

- RAID 0 - 无容错的条带化磁盘阵列
- RAID 1 - 镜像和双工
- RAID 2 - 内存式纠错码
- RAID 3 - 比特交错奇偶校验
- RAID 4 - 块交错奇偶校验
- RAID 5 - 块交错分布式奇偶校验
- RAID 6 - P + Q冗余

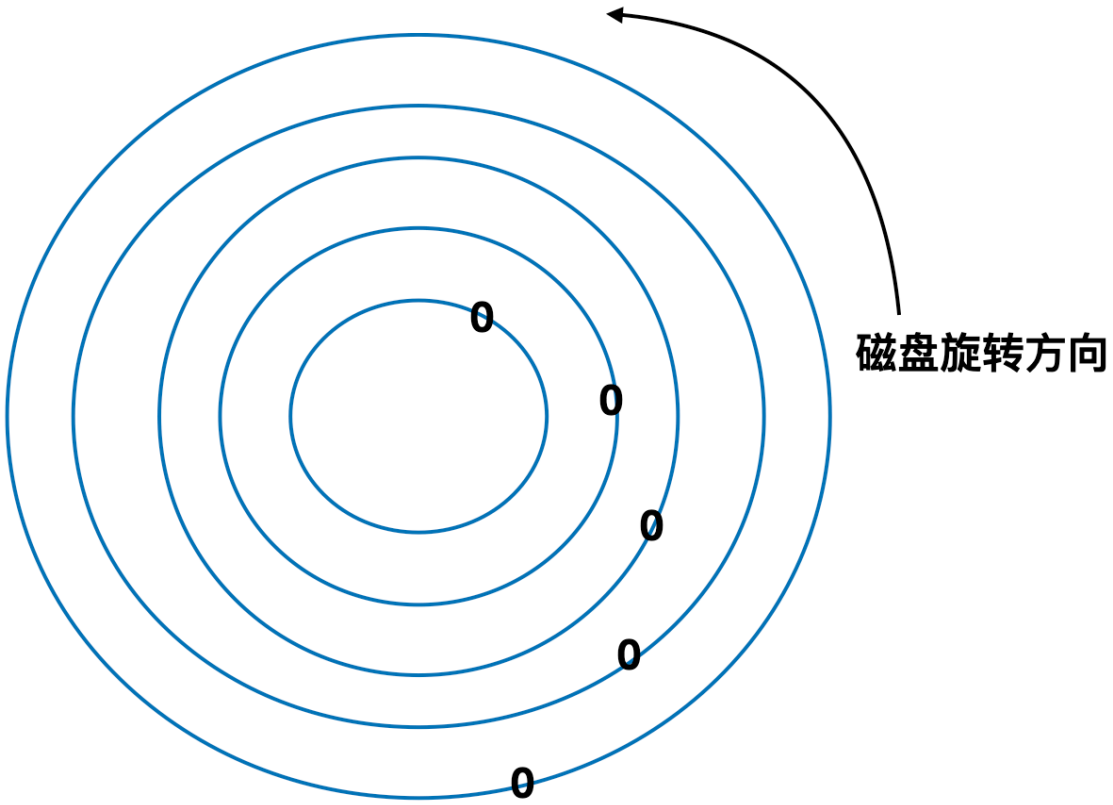
磁盘格式化

磁盘由一堆铝的、合金或玻璃的盘片组成，磁盘刚被创建出来后，没有任何信息。磁盘在使用前必须经过 `低级格式化(low-levvel format)` ，下面是一个扇区的格式



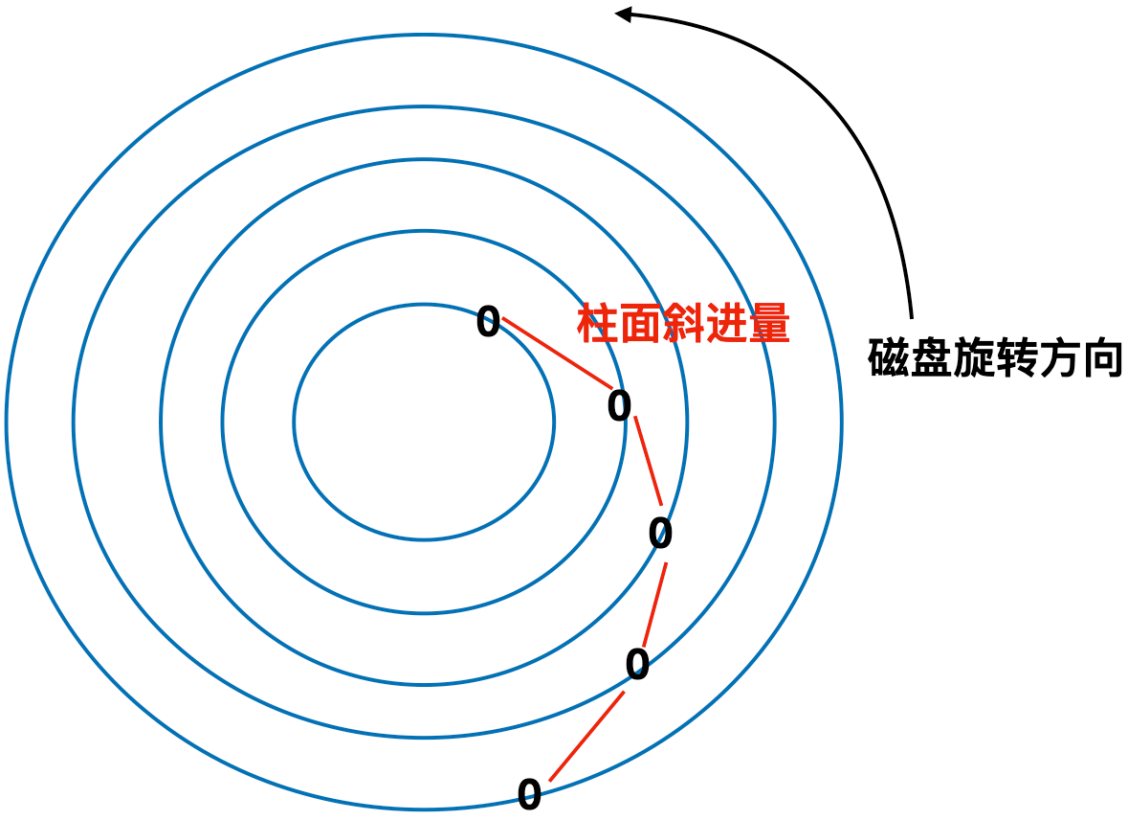
前导码相当于是标示扇区的开始位置，通常以位模式开始，前导码还包括 `柱面号` 、 `扇区号` 等一些其他信息。紧随前导码后面的是数据区，数据部分的大小由低级格式化程序来确定。大部分磁盘使用 512 字节的扇区。数据区后面是 ECC，ECC 的全称是 **error correction code** ， `数据纠错码` ，它与普通的错误检测不同，ECC 还可以用于恢复读错误。ECC 阶段的大小由不同的磁盘制造商实现。ECC 大小的设计标准取决于**设计者愿意牺牲多少磁盘空间来提高可靠性**，以及程序可以处理的 ECC 的复杂程度。通常情况下 ECC 是 16 位，除此之外，硬盘一般具有一定数量的备用扇区，用于替换制造缺陷的扇区。

低级格式化后的每个 0 扇区的位置都和前一个磁道存在 `偏移` ，如下图所示



这种方式又被称为 `柱面斜进(cylinder skew)` ，之所以采用这种方式是为了提高程序的运行性能。可以这样想，磁盘在转动的过程中会经由磁头来读取扇区信息，在读取内侧一圈扇区数据后，磁头会进行向外侧磁道的寻址操作，寻址操作的同时磁盘在继续转动，如果不采用这种方式，可能刚好磁头寻址到外侧，0 号扇区已经转过了磁头，所以需要旋转一圈才能等到它继续读取，通过柱面斜进的方式可以消除这一问题。

柱面斜进量取决于驱动器的几何规格。柱面斜进量就是两个相邻同心圆 0 号扇区的差异量。如下图所示



这里需要注意一点，不只有柱面存在斜进，磁头也会存在 `斜进(head skew)`，但是磁头斜进比较小。

磁盘格式化会减少磁盘容量，减少的磁盘容量都会由前导码、扇区间间隙和 ECC 的大小以及保留的备用扇区数量。

在磁盘使用前，还需要经过最后一道工序，那就是对每个分区分别执行一次 `高级格式化(high-level format)`，这一操作要设置一个引导块、空闲存储管理（采用位图或者是空闲列表）、根目录和空文件系统。这一步操作会把码放在分区表项中，告诉分区使用的是哪种文件系统，因为许多操作系统支持多个兼容的文件系统。在这一步之后，系统就可以进行引导过程。

当电源通电后，BIOS 首先运行，它会读取主引导记录并跳转到主引导记录中。然后引导程序会检查以了解哪个分区是处于活动的。然后，它从该分区读取 `启动扇区(boot sector)` 并运行它。启动扇区包含一个小程序来加载一个更大一点的引导器来搜索文件系统以找到 `系统内核(system kernel)`，然后程序被转载进入内存并执行。

> 这里说下什么是引导扇区：引导扇区是磁盘或者存储设备的保留扇区，其中包含用于完成计算机或磁盘引导过程所必要的数据或者代码。
>
> 引导扇区存储引导记录数据，这些数据用于在计算机启动时提供指令。有两种不同类型的引导扇区
>
> - Master boot record 称为主引导扇区
> - Volume boot record 卷启动记录
>
> 对于分区磁盘，引导扇区由主引导记录组成；
>
> 非分区磁盘由卷启动记录组成。

磁盘臂调度算法

下面我们来探讨一下关于影响磁盘读写的算法，一般情况下，影响磁盘快读写的时间由下面几个因素决定

- 寻道时间 - 寻道时间指的就是将磁盘臂移动到需要读取磁盘块上的时间
- 旋转延迟 - 等待合适的扇区旋转到磁头下所需的时间
- 实际数据的读取或者写入时间

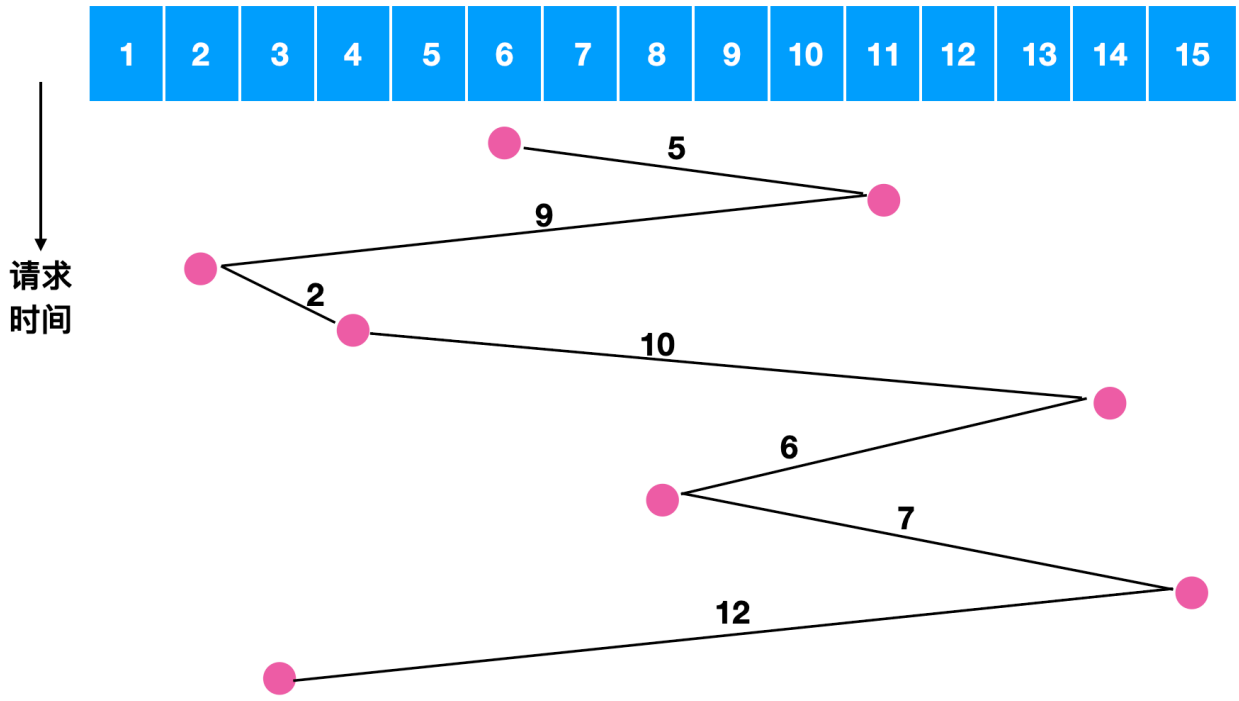
这三种时间参数也是磁盘寻道的过程。一般情况下，寻道时间对总时间的影响最大，所以，有效的降低寻道时间能够提高磁盘的读取速度。

如果磁盘驱动程序每次接收一个请求并按照接收顺序完成请求，这种处理方式也就是 `先来先服务(First-Come, First-served, FCFS)` ，这种方式很难优化寻道时间。因为每次都会按照顺序处理，不管顺序如何，有可能这次读完后需要等待一个磁盘旋转一周才能继续读取，而其他柱面能够马上进行读取，这种情况下每次请求也会排队。

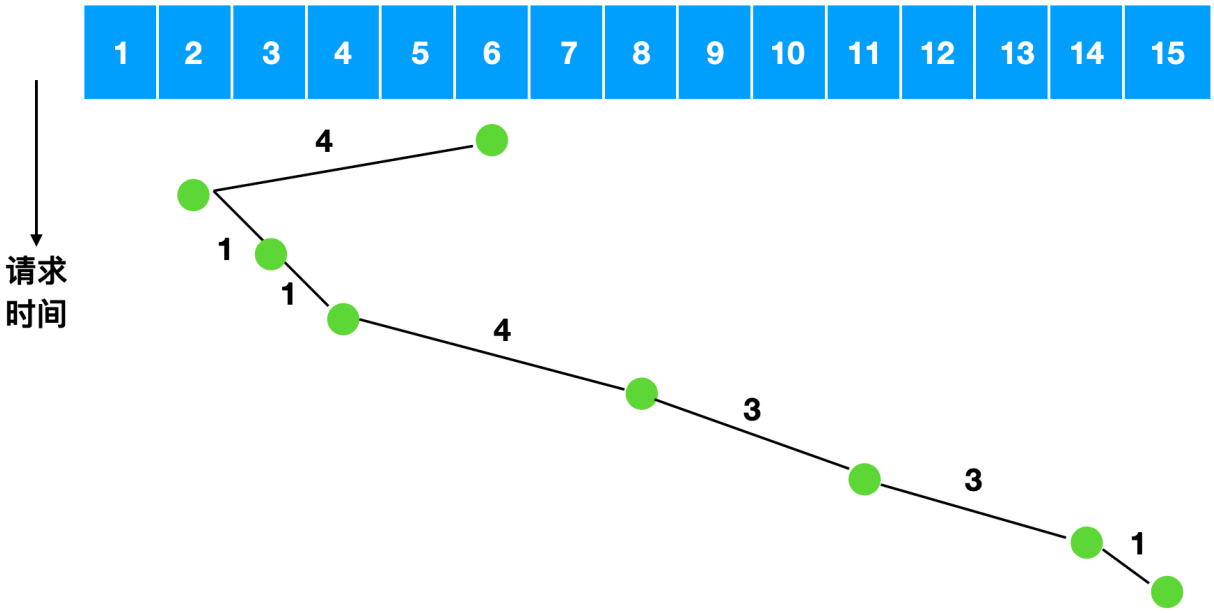
通常情况下，磁盘在进行寻道时，其他进程会产生其他的磁盘请求。磁盘驱动程序会维护一张表，表中会记录着柱面号当作索引，每个柱面未完成的请求会形成链表，链表头存放在表的相应表项中。

一种对先来先服务的算法改良的方案是使用 `最短路径优先(SSF)` 算法，下面描述了这个算法。

假如我们在对磁道 6 号进行寻址时，同时发生了对 11 , 2 , 4, 14, 8, 15, 3 的请求，如果采用先来先服务的原则，如下图所示



我们可以计算一下磁盘臂所跨越的磁盘数量为 5 + 9 + 2 + 10 + 6 + 7 + 12 = 51，相当于是跨越了 51 次盘面，如果使用最短路径优先，我们来计算一下跨越的盘面



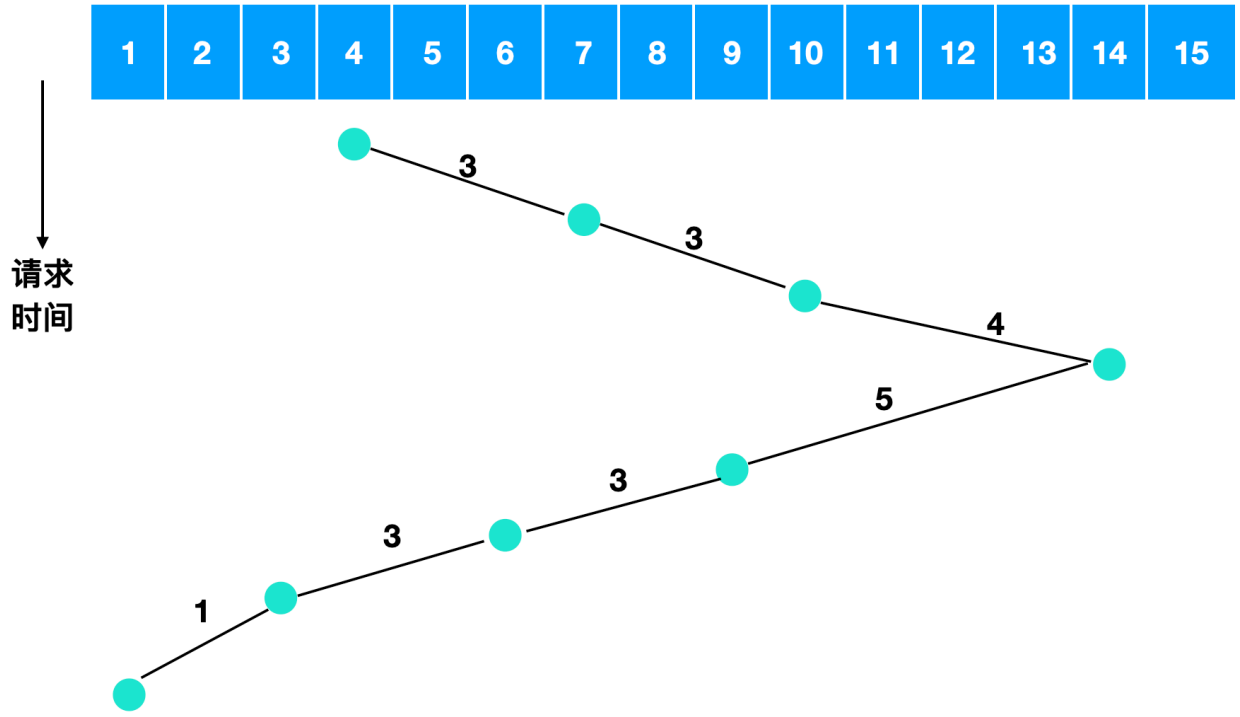
跨越的磁盘数量为 4 + 1 + 1 + 4 + 3 + 3 + 1 = 17 ，相比 51 足足省了两倍的时间。

但是，最短路径优先的算法也不是完美无缺的，这种算法照样存在问题，那就是 `优先级` 问题，

这里有一个原型可以参考就是我们日常生活中的电梯，电梯使用一种 `电梯算法(elevator algorithm)` 来进行调度，从而满足协调效率和公平性这两个相互冲突的目标。电梯一般会保持向一个方向移动，直到在那个方向上没有请求为止，然后改变方向。

电梯算法需要维护一个 `二进制位` ，也就是当前的方向位： `UP(向上)` 或者是 `DOWN(向下)` 。当一个请求处理完成后，磁盘或电梯的驱动程序会检查该位，如果此位是 UP 位，磁盘臂或者电梯仓移到下一个更高跌未完成的请求。如果高位没有未完成的请求，则取相反方向。当方向位是 `DOWN` 时，同时存在一个低位的请求，磁盘臂会转向该点。如果不存在的话，那么它只是停止并等待。

我们举个例子来描述一下电梯算法，比如各个柱面得到服务的顺序是 4，7，10，14，9，6，3，1 ，那么它的流程图如下



所以电梯算法需要跨越的盘面数量是 3 + 3 + 4 + 5 + 3 + 3 + 1 = 22

电梯算法通常情况下不如 SSF 算法。

一些磁盘控制器为软件提供了一种检查磁头下方当前扇区号的方法，使用这样的控制器，能够进行另一种优化。如果对一个相同的柱面有两个或者多个请求正等待处理，驱动程序可以发出请求读写下一次要通过磁头的扇区。

> 这里需要注意一点，当一个柱面有多条磁道时，相继的请求可能针对不同的磁道，这种选择没有代价，因为选择磁头不需要移动磁盘臂也没有旋转延迟。

对于磁盘来说，最影响性能的就是寻道时间和旋转延迟，所以一次只读取一个或两个扇区的效率是非常低的。出于这个原因，许多磁盘控制器总是读出多个扇区并进行高速缓存，即使只请求一个扇区时也是这样。一般情况下读取一个扇区的同时会读取该扇区所在的磁道或者是所有剩余的扇区被读出，读出扇区的数量取决于控制器的高速缓存中有多少可用的空间。

磁盘控制器的高速缓存和操作系统的高速缓存有一些不同，磁盘控制器的高速缓存用于缓存没有实际被请求的块，而操作系统维护的高速缓存由显示地读出的块组成，并且操作系统会认为这些块在近期仍然会频繁使用。

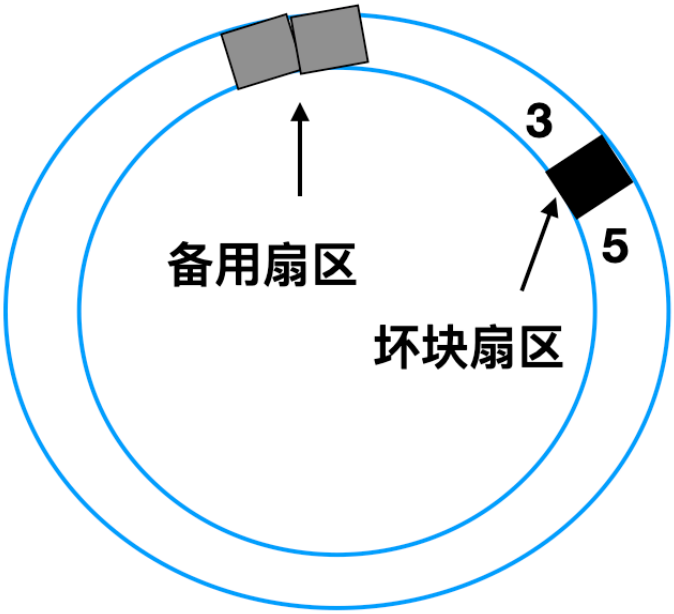
当同一个控制器上有多个驱动器时，操作系统应该为每个驱动器都单独的维护一个未完成的请求表。一旦有某个驱动器闲置时，就应该发出一个寻道请求来将磁盘臂移到下一个被请求的柱面。如果下一个寻道请求到来时恰好没有磁盘臂处于正确的位置，那么驱动程序会在刚刚完成传输的驱动器上发出一个新的寻道命令并等待，等待下一次中断到来时检查哪个驱动器处于闲置状态。

错误处理

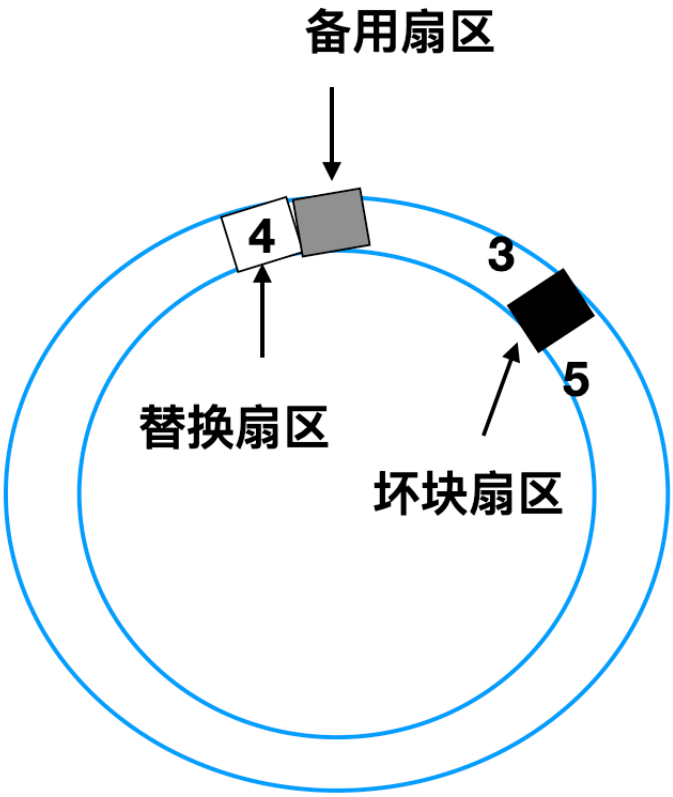
磁盘在制造的过程中可能会有瑕疵，如果瑕疵比较小，比如只有几位，那么使用坏扇区并且每次只是让 ECC 纠正错误是可行的，如果瑕疵较大，那么错误就不可能被掩盖。

一般坏块有两种处理办法，一种是在控制器中进行处理；一种是在操作系统层面进行处理。

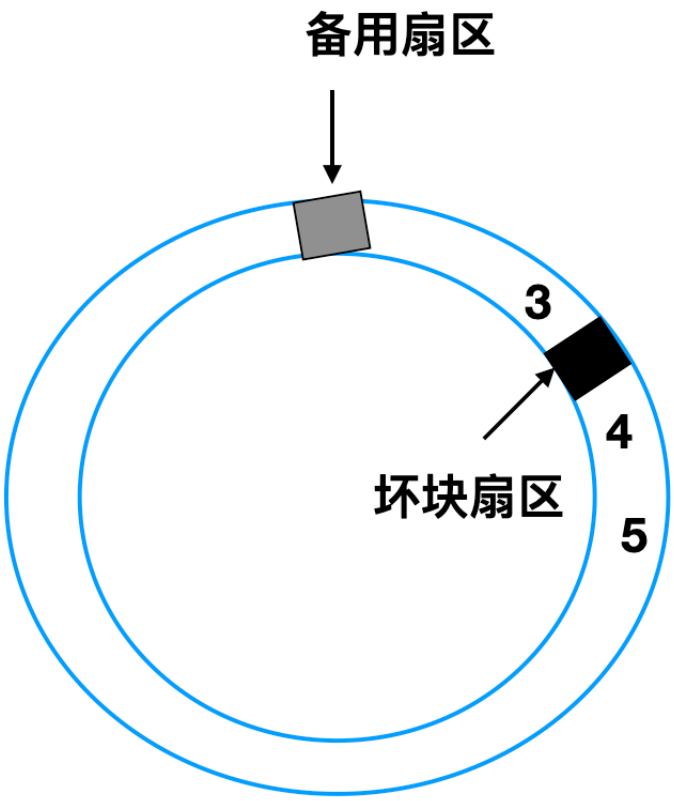
这两种方法经常替换使用，比如一个具有 30 个数据扇区和两个备用扇区的磁盘，其中扇区 4 是有瑕疵的。



控制器能做的事情就是将备用扇区之一重新映射。



还有一种处理方式是将所有的扇区都向上移动一个扇区



上面这这两种情况下控制器都必须知道哪个扇区，可以通过内部的表来跟踪这一信息，或者通过重写前导码来给出重新映射的扇区号。如果是重写前导码，那么涉及移动的方式必须重写后面所有的前导码，但是最终会提供良好的性能。

稳定存储器

磁盘经常会出现错误，导致好的扇区会变成坏扇区，驱动程序也有可能挂掉。RAID 可以对扇区出错或者是驱动器崩溃提出保护，然而 RAID 却不能对坏数据中的写错误提供保护，也不能对写操作期间的崩溃提供保护，这样就会破坏原始数据。

我们期望磁盘能够准确无误的工作，但是事实情况是不可能的，但是我们能够知道的是，一个磁盘子系统具有如下特性：当一个写命令发给它时，磁盘要么正确地写数据，要么什么也不做，让现有的数据完整无误的保留。这样的系统称为 `稳定存储器(stable storage)` 。 稳定存储器的目标就是不惜一切代价保证磁盘的一致性。

稳定存储器使用两个一对相同的磁盘，对应的块一同工作形成一个无差别的块。稳定存储器为了实现这个目的，定义了下面三种操作：

- `稳定写(stable write)`
- `稳定读(stable read)`
- `崩溃恢复(crash recovery)`

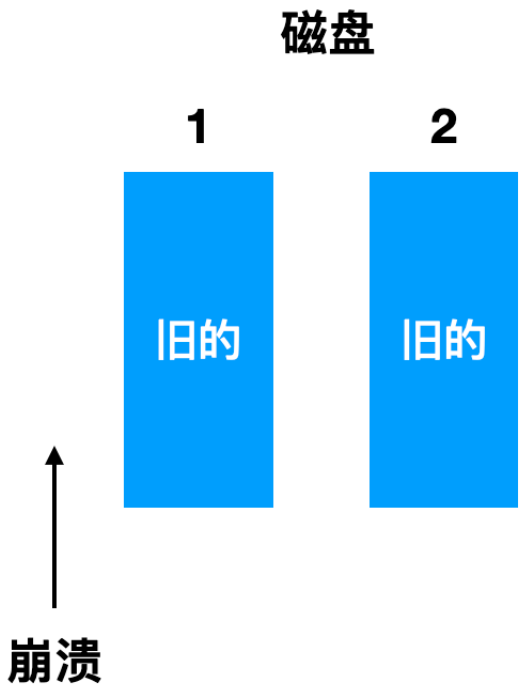
稳定写指的就是首先将块写到比如驱动器 1 上，然后将其读回来验证写入的是否正确，如果不正确，那么就会再次尝试写入和读取，一直到能够验证写入正确为止。如果块都写完了也没有验证正确，就会换块继续写入和读取，直到正确为止。无论尝试使用多少个备用块，都是在对你驱动器 1 写入成功之后，才会对驱动器 2 进行写入和读取。这样我们相当于是对两个驱动器进行写入。

稳定读指的就是首先从驱动器 1 上进行读取，如果读取操作会产生错误的 ECC，则再次尝试读取，如果所有的读取操作都会给出错误的 ECC，那么会从驱动器 2 上进行读取。这样我们相当于是对两个驱动器进行读取。

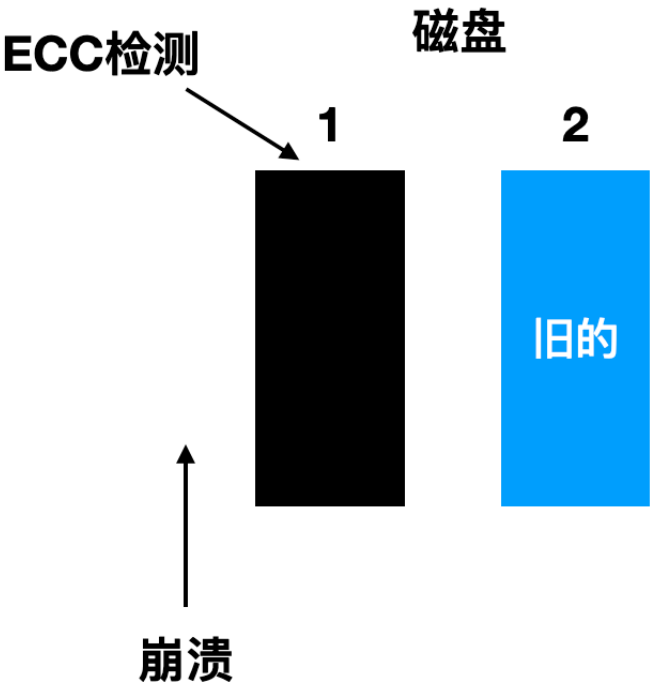
崩溃恢复指的是崩溃之后，恢复程序扫描两个磁盘，比较对应的块。如果一对块都是好的并且是相同的，就不会触发任何机制；如果其中一个块触发了 ECC 错误，这时候就需要使用好块来覆盖坏块。

如果 CPU 没有崩溃的话，那么这种方式是可行的，因为稳定写总是对每个块写下两个有效的副本，并且假设自发的错误不会再相同的时刻发生在两个对应的块上。如果在稳定写期间出现 CPU 崩溃会怎么样？这就取决于崩溃发生的精确时间，有五种情况，下面来说一下

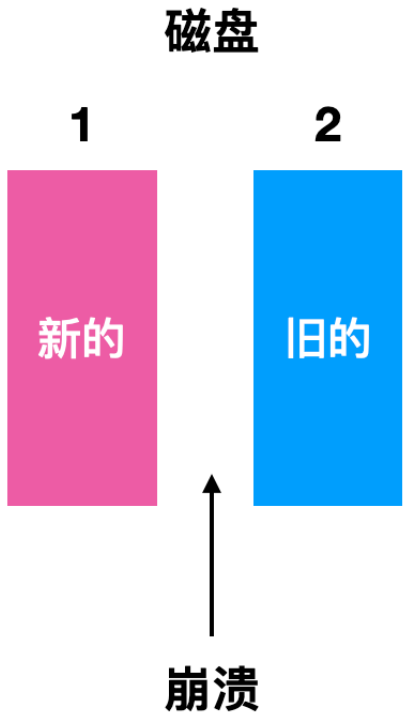
- 第一种情况是崩溃发生在写入之前，在恢复的时候就什么都不需要修改，旧的值也会继续存在。



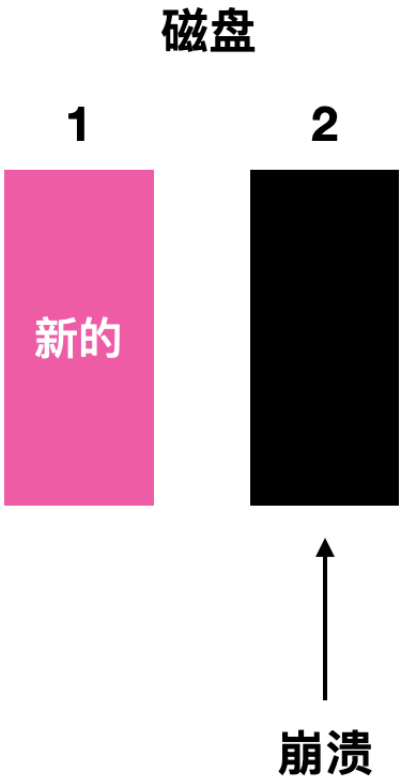
- 第二种情况是 CPU 崩溃发生在写入驱动器 1 的时候，崩溃导致块内容被破坏，然而恢复程序能够检测出这一种错误，并且从驱动器 2 恢复驱动器 1 上的块。



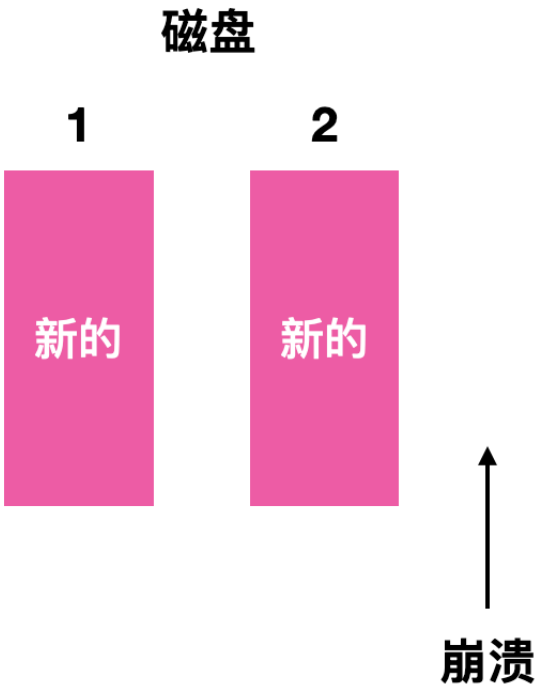
- 第三种情况是崩溃发生在磁盘驱动器 1 之后但是还没有写驱动器 2 之前，这种情况下由于磁盘 1 已经写入成功



- 第四种情况是崩溃发生在磁盘驱动 1 写入后在磁盘驱动 2 写入时，恢复期间会用好的块替换坏的块，两个块的最终值都是最新的



- 最后一种情况就是崩溃发生在两个磁盘驱动写入后，这种情况下不会发生任何问题



这种模式下进行任何优化和改进都是可行的，但是代价高昂，一种改进是在稳定写期间监控被写入的块，这样在崩溃后进行检验的块只有一个。有一种 `非易失性 RAM` 能够在崩溃之后保留数据，但是这种方式并不推荐使用。

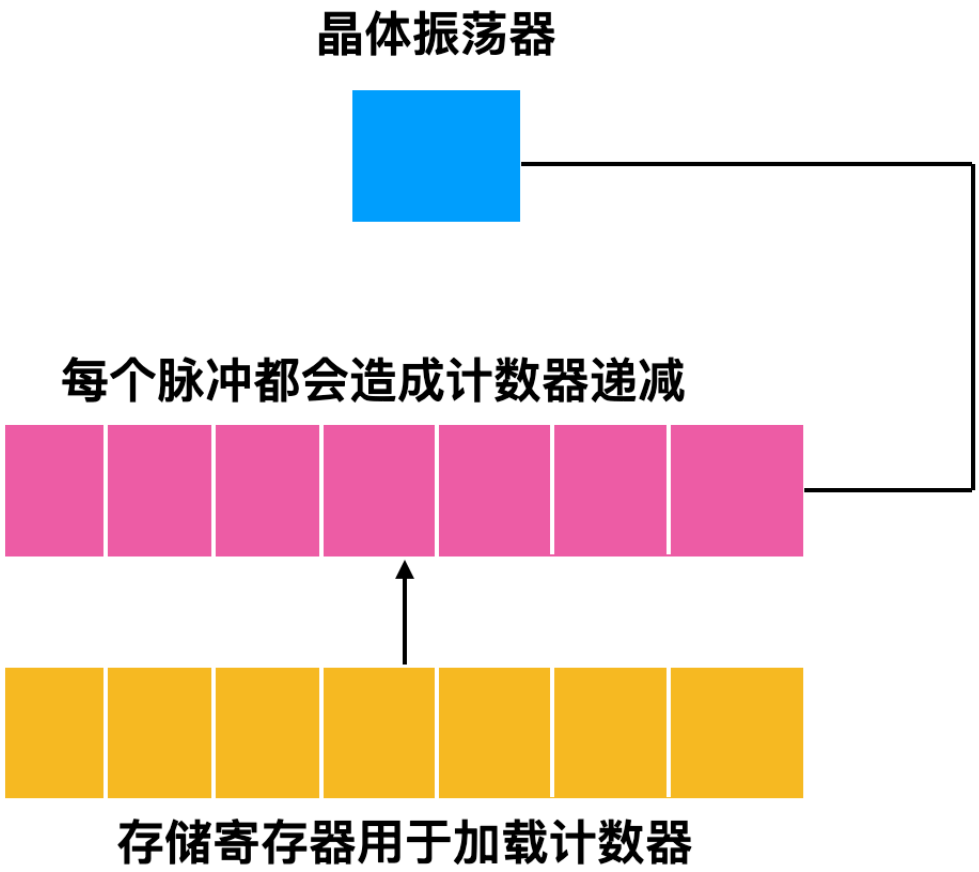
时钟

`时钟(Clocks)` 也被称为 `定时器(timers)`，时钟/定时器对任何程序系统来说都是必不可少的。时钟负责维护时间、防止一个进程长期占用 CPU 时间等其他功能。 `时钟软件(clock software)` 也是一种设备驱动的方式。下面我们就来对时钟进行介绍，一般都是先讨论硬件再介绍软件，采用由下到上的方式，也是告诉你，底层是最重要的。

时钟硬件

在计算机中有两种类型的时钟，这些时钟与现实生活中使用的时钟完全不一样。

- 比较简单的一种时钟被连接到 110 V 或 220 V 的电源线上，这样每个 `电压周期` 会产生一个中断，大概是 50 - 60 HZ。这些时钟过去一直占据支配地位。
- 另外的一种时钟由晶体振荡器、计数器和寄存器组成，示意图如下所示



这种时钟称为 `可编程时钟` ，可编程时钟有两种模式，一种是 `一键式(one-shot mode)`，当时钟启动时，会把存储器中的值复制到计数器中，然后，每次晶体的振荡器的脉冲都会使计数器 -1。当计数器变为 0 时，会产生一个中断，并停止工作，直到软件再一次显示启动。还有一种模式时 `方波(square-wave mode)` 模式，在这种模式下，当计数器变为 0 并产生中断后，存储寄存器的值会自动复制到计数器中，这种周期性的中断称为一个时钟周期。

时钟软件

时钟硬件所做的工作只是根据已知的时间间隔产生中断，而其他的工作都是由 `时钟软件` 来完成，一般操作系统的不同，时钟软件的具体实现也不同，但是一般都会包括以下这几点

- 维护一天的时间
- 阻止进程运行的时间超过其指定时间
- 统计 CPU 的使用情况
- 处理用户进程的警告系统调用
- 为系统各个部分提供看门狗定时器
- 完成概要剖析，监视和信息收集

软定时器

时钟软件也被称为可编程时钟，可以设置它以程序需要的任何速率引发中断。时钟软件触发的中断是一种硬中断，但是某些应用程序对于硬中断来说是不可接受的。

这时候就需要一种 `软定时器(soft timer)` 避免了中断，无论何时当内核因为某种原因呢在运行时，它返回用户态之前都会检查时钟来了解软定时器是否到期。如果软定时器到期，则执行被调度的事件也无需切换到内核态，因为本身已经处于内核态中。这种方式避免了频繁的内核态和用户态之前的切换，提高了程序运行效率。

软定时器因为不同的原因切换进入内核态的速率不同，原因主要有

- 系统调用
- TLB 未命中
- 缺页异常
- I/O 中断
- CPU 变得空闲

死锁



前言

计算机系统中有很多 `独占性` 的资源，在同一时刻只能每个资源只能由一个进程使用，我们之前经常提到过打印机，这就是一个独占性的资源，**同一时刻能有两个打印机同时输出结果，否则会引起文件系统的瘫痪**。所以，操作系统具有授权一个进程单独访问资源的能力。

两个进程独占性的访问某个资源，从而等待另外一个资源的执行结果，会导致两个进程都被阻塞，并且两个进程都不会释放各自的资源，这种情况就是 `死锁(deadlock)` 。

死锁可以发生在任何层面，在不同的机器之间可能会发生死锁，在数据库系统中也会导致死锁，比如进程 A 对记录 R1 加锁，进程 B 对记录 R2 加锁，然后进程 A 和 B 都试图把对象的记录加锁，这种情况下就会产生死锁。

下面我们就来讨论一下什么是死锁、死锁的条件是什么、死锁如何预防、活锁是什么等。

首先你需要先了解一个概念，那就是资源是什么

资源

大部分的死锁都和资源有关，在进程对设备、文件具有独占性（排他性）时会产生死锁。我们把这类需要排他性使用的对象称为 `资源(resource)` 。资源主要分为 **可抢占资源和不可抢占资源**

可抢占资源和不可抢占资源

资源主要有可抢占资源和不可抢占资源。 `可抢占资源(preemptable resource)` 可以从拥有它的进程中抢占而不会造成其他影响，内存就是一种可抢占性资源，任何进程都能够抢先获得内存的使用权。

`不可抢占资源(nonpreemtable resource)` 指的是除非引起错误或者异常，否则进程无法抢占指定资源，这种不可抢占的资源比如有光盘，在进程执行调度的过程中，其他进程是不能得到该资源的。

死锁与不可抢占资源有关，虽然抢占式资源也会造成死锁，不过这种情况的解决办法通常是在进程之间重新分配资源来化解。所以，我们的重点自然就会放在了不可抢占资源上。

下面给出了使用资源所需事件的抽象顺序



如果在请求时资源不存在，请求进程就会强制等待。在某些操作系统中，当请求资源失败时进程会自动阻塞，当自资源可以获取时进程会自动唤醒。在另外一些操作系统中，请求资源失败并显示错误代码，然后等待进程等待一会儿再继续重试。

请求资源失败的进程会陷入一种**请求资源、休眠、再请求资源**的循环中。此类进程虽然没有阻塞，但是处于从目的和结果考虑，这类进程和阻塞差不多，因为这类进程并没有做任何有用的工作。

请求资源的这个过程是很依赖操作系统的。在一些系统中，一个 `request` 系统调用用来允许进程访问资源。在一些系统中，操作系统对资源的认知是它是一种特殊文件，在任何同一时刻只能被一个进程打开和占用。资源通过 `open` 命令进行打开。如果文件已经正在使用，那么这个调用者会阻塞直到当前的占用文件的进程关闭文件为止。

资源获取

对于一些数据库系统中的记录这类资源来说，应该由用户进程来对其进行管理。有一种管理方式是使用 `信号量(semaphore)` 。这些信号量会初始化为 1 。互斥锁也能够起到相同的作用。

> 这里说一下什么是 `互斥锁(Mutexes)`：
>
> 在计算机程序中， `互斥对象(mutex)` 是一个程序对象，它允许多个程序共享同一资源，例如文件访问权限，但并不是同时访问。需要锁定资源的线程都必须在使用资源时将互斥锁与其他线程绑定（进行加锁）。当不再需要数据或线程结束时，互斥锁设置为解锁。

下面是一个伪代码，这部分代码说明了信号量的资源获取、资源释放等操作，如下所示

```
typedef int semaphore;
semaphore aResource;

void processA(void){

  down(&aResource);
  useResource();
  up(&aResource);

}
```

上面显示了一个进程资源获取和释放的过程，但是一般情况下会存在多个资源同时获取锁的情景，这样该如何处理？如下所示

```
typedef int semaphore;
semaphore aResource;
semaphore bResource;

void processA(void){

  down(&aResource);
  down(&bResource);
  useAResource();
  useBResource();
  up(&aResource);
  up(&bResource);

}
```

对于单个进程来说，并不需要加锁，因为不存在和这个进程的竞争条件。所以单进条件下程序能够完好运行。

现在让我们考虑两个进程的情况，A 和 B ， 还存在两个资源。如下所示

```
typedef int semaphore;
semaphore aResource;
semaphore bResource;

void processA(void){

  down(&aResource);
  down(&bResource);
  useBothResource();
  up(&bResource);
```

```
    up(&aResource);

  }

  void processB(void){

    down(&aResource);
    down(&bResource);
    useBothResource();
    up(&bResource);
    up(&aResource);

  }
```

在上述代码中，两个进程以相同的顺序访问资源。在这段代码中，一个进程在另一个进程之前获取资源，如果另外一个进程想在第一个进程释放之前获取资源，那么它会由于资源的加锁而阻塞，直到该资源可用为止。

在下面这段代码中，有一些变化

```
  typedef int semaphore;
  semaphore aResource;
  semaphore bResource;

  void processA(void){

    down(&aResource);
    down(&bResource);
    useBothResource();
    up(&bResource);
    up(&aResource);

  }

  void processB(void){

    down(&bResource); // 变化的代码
    down(&aResource); // 变化的代码
    useBothResource();
    up(&aResource); // 变化的代码
    up(&bResource); // 变化的代码

  }
```

这种情况就不同了，可能会发生同时获取两个资源并有效地阻塞另一个过程，直到完成为止。也就是说，可能会发生进程 A 获取资源 A 的同时进程 B 获取资源 B 的情况。然后每个进程在尝试获取另一个资源时被阻塞。

在这里我们会发现一个简单的获取资源顺序的问题就会造成 `死锁` ，所以死锁是很容易发生的，所以下面我们就对死锁做一个详细的认识和介绍。

死锁

如果要对死锁进行一个定义的话，下面的定义比较贴切

如果一组进程中的每个进程都在等待一个事件，而这个事件只能由该组中的另一个进程触发，这种情况会导致死锁。

简单一点来表述一下，就是每个进程都在等待其他进程释放资源，而其他资源也在等待每个进程释放资源，这样没有进程抢先释放自己的资源，这种情况会产生死锁，所有进程都会无限的等待下去。

换句话说，死锁进程结合中的每个进程都在等待另一个死锁进程已经占有的资源。但是由于所有进程都不能运行，它们之中任何一个资源都无法释放资源，所以没有一个进程可以被唤醒。这种死锁也被称为 `资源死锁(resource deadlock)` 。资源死锁是最常见的类型，但不是所有的类型，我们后面会介绍其他类型，我们先来介绍资源死锁

资源死锁的条件

针对我们上面的描述，资源死锁可能出现的情况主要有

- 互斥条件：每个资源都被分配给了一个进程或者资源是可用的
- 保持和等待条件：已经获取资源的进程被认为能够获取新的资源
- 不可抢占条件：分配给一个进程的资源不能强制的从其他进程抢占资源，它只能由占有它的进程显示释放
- 循环等待：死锁发生时，系统中一定有两个或者两个以上的进程组成一个循环，循环中的每个进程都在等待下一个进程释放的资源。

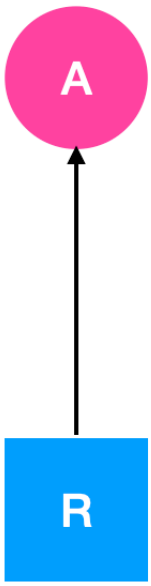
发生死锁时，上面的情况必须同时会发生。如果其中任意一个条件不会成立，死锁就不会发生。可以通过破坏其中任意一个条件来破坏死锁，下面这些破坏条件就是我们探讨的重点

死锁模型

Holt 在 1972 年提出对死锁进行建模，建模的标准如下：

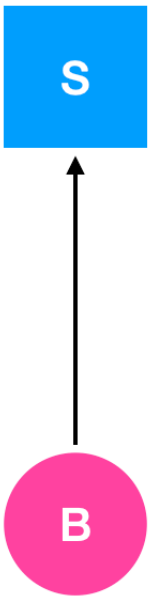
- 圆形表示进程
- 方形表示资源

从资源节点到进程节点表示资源已经被进程占用，如下图所示

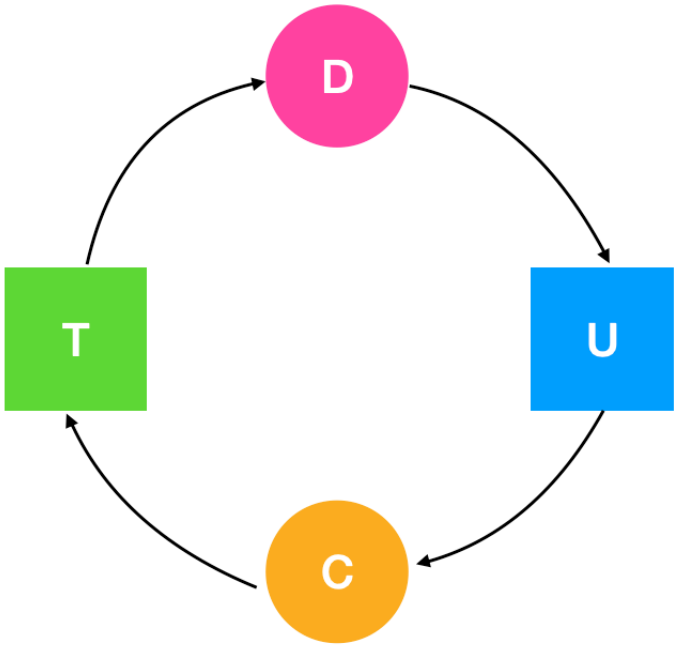


在上图中表示当前资源 R 正在被 A 进程所占用

由进程节点到资源节点的有向图表示当前进程正在请求资源，并且该进程已经被阻塞，处于等待这个资源的状态



在上图中，表示的含义是进程 B 正在请求资源 S 。Holt 认为，死锁的描述应该如下



这是一个死锁的过程，进程 C 等待资源 T 的释放，资源 T 却已经被进程 D 占用，进程 D 等待请求占用资源 U ，资源 U 却已经被线程 C 占用，从而形成环。

总结一点：**吃着碗里的看着锅里的容易死锁**

那么如何避免死锁呢？我们还是通过死锁模型来聊一聊

假设有三个进程 (A、B、C) 和三个资源(R、S、T) 。三个进程对资源的请求和释放序列如下图所示

进程A	进程B	进程C
请求 R	请求 S	请求 T
请求 S	请求 T	请求 R
释放 R	释放 S	释放 T
释放 S	释放 T	释放 R

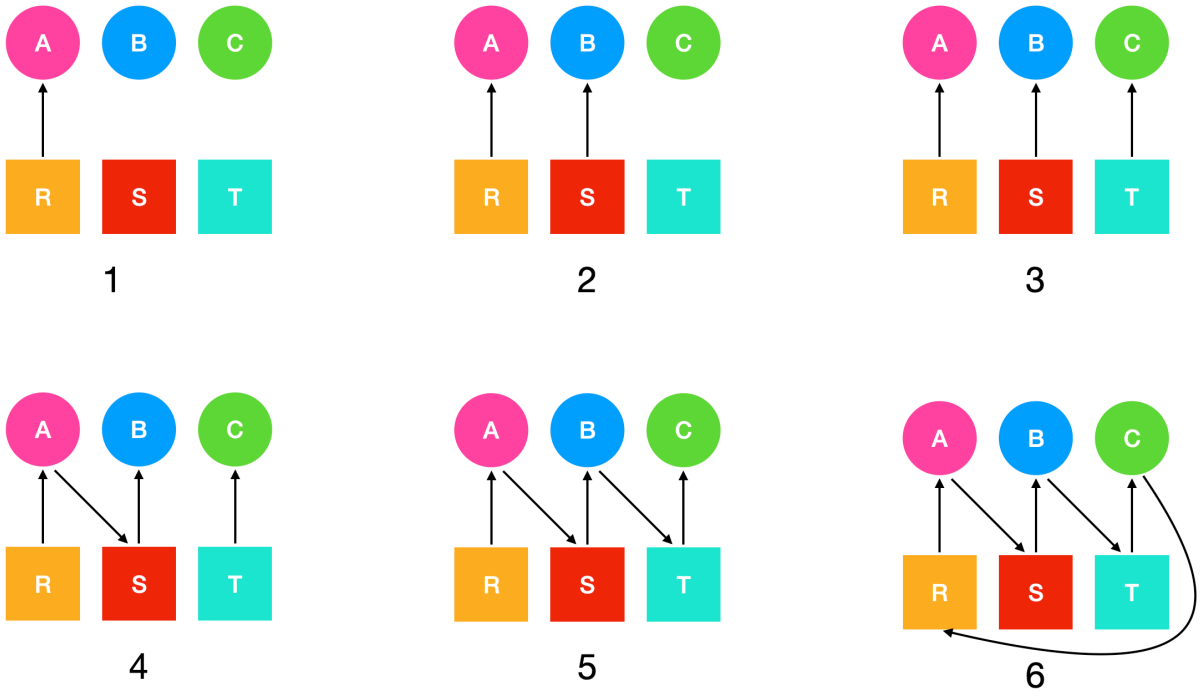
操作系统可以任意选择一个非阻塞的程序运行，所以它可以决定运行 A 直到 A 完成工作；它可以运行 B 直到 B 完成工作；最后运行 C。

这样的顺序不会导致死锁（因为不存在对资源的竞争），但是这种情况也完全没有 `并行性` 。进程除了在请求和释放资源外，还要做计算和输入/输出的工作。当进程按照顺序运行时，在等待一个 I/O 时，另一个进程不能使用 CPU。所以，严格按照串行的顺序执行并不是最优越的。另一方面，如果没有进程在执行任何 I/O 操作，那么最短路径优先作业会优于轮转调度，所以在这种情况下串行可能是最优越的

现在我们假设进程会执行计算和 I/O 操作，所以轮询调度是一种合理的 `调度算法` 。资源请求可能会按照下面这个顺序进行

1. A 请求 R
2. B 请求 S
3. C 请求 T
4. A 请求 S
5. B 请求 T
6. C 请求 R

下图是针对上面这六个步骤的资源分配图。



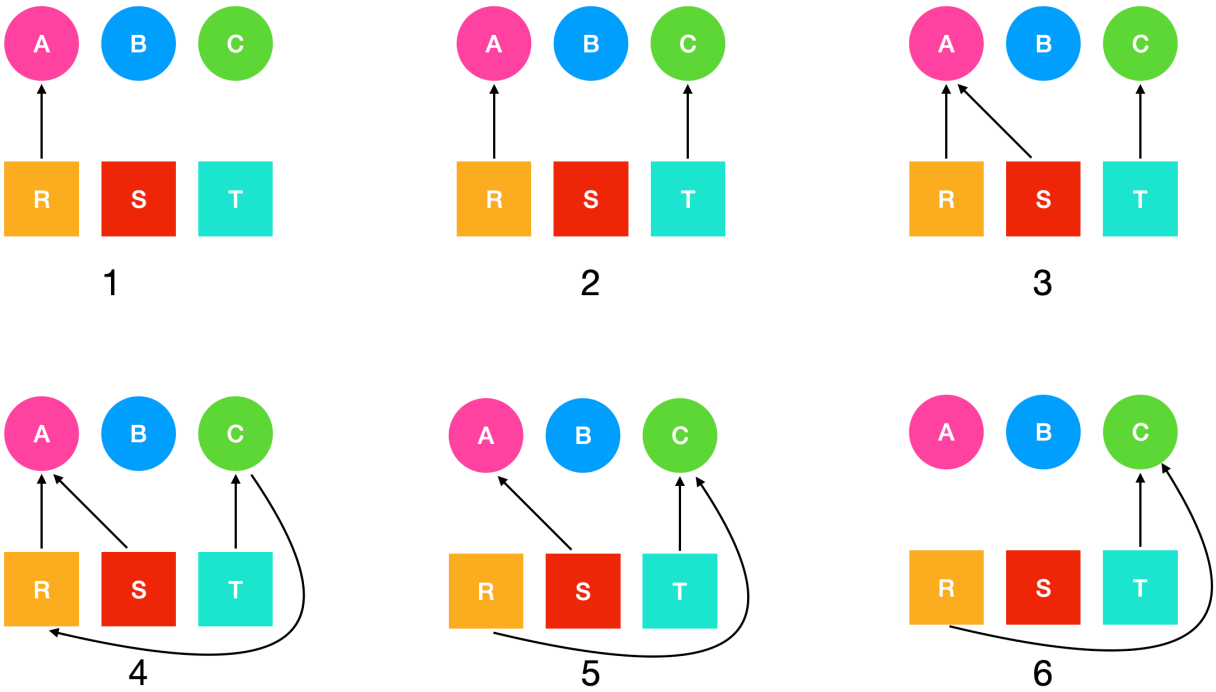
> 这里需要注意一个问题，为什么从资源出来的有向图指向了进程却表示**进程请求资源**呢？笔者刚开始看也有这个疑问，但是想了一下这个意思解释为进程占用资源比较合适，而进程的有向图指向资源表示进程被阻塞的意思。

在上面的第四个步骤，进程 A 正在等待资源 S；第五个步骤中，进程 B 在等待资源 T；第六个步骤中，进程 C 在等待资源 R，因此产生了环路并导致了死锁。

然而，操作系统并没有规定一定按照某种特定的顺序来执行这些进程。遇到一个可能会引起死锁的线程后，操作系统可以干脆不批准请求，并把进程挂起一直到安全状态为止。比如上图中，如果操作系统认为有死锁的可能，它可以选择不把资源 S 分配给 B ，这样 B 被挂起。这样的话操作系统会只运行 A 和 C，那么资源的请求和释放就会是下面的步骤

1. A 请求 R
2. C 请求 T
3. A 请求 S
4. C 请求 R
5. A 释放 R
6. A 释放 S

下图是针对上面这六个步骤的资源分配图。



在第六步执行完成后，可以发现并没有产生死锁，此时就可以把资源 S 分配给 B，因为 A 进程已经执行完毕，C 进程已经拿到了它想要的资源。进程 B 可以直接获得资源 S，也可以等待进程 C 释放资源 T 。

有四种处理死锁的策略：

- 忽略死锁带来的影响（惊呆了）
- 检测死锁并回复死锁，死锁发生时对其进行检测，一旦发生死锁后，采取行动解决问题
- 通过仔细分配资源来避免死锁
- 通过破坏死锁产生的四个条件之一来避免死锁

下面我们分别介绍一下这四种方法

鸵鸟算法

最简单的解决办法就是使用`鸵鸟算法(ostrich algorithm)`，把头埋在沙子里，假装问题根本没有发生。每个人看待这个问题的反应都不同。数学家认为死锁是不可接受的，必须通过有效的策略来防止死锁的产生。工程师想要知道问题发生的频次，系统因为其他原因崩溃的次数和死锁带来的严重后果。如果死锁发生的频次很低，而经常会由于硬件故障、编译器错误等其他操作系统问题导致系统崩溃，那么大多数工程师不会修复死锁。

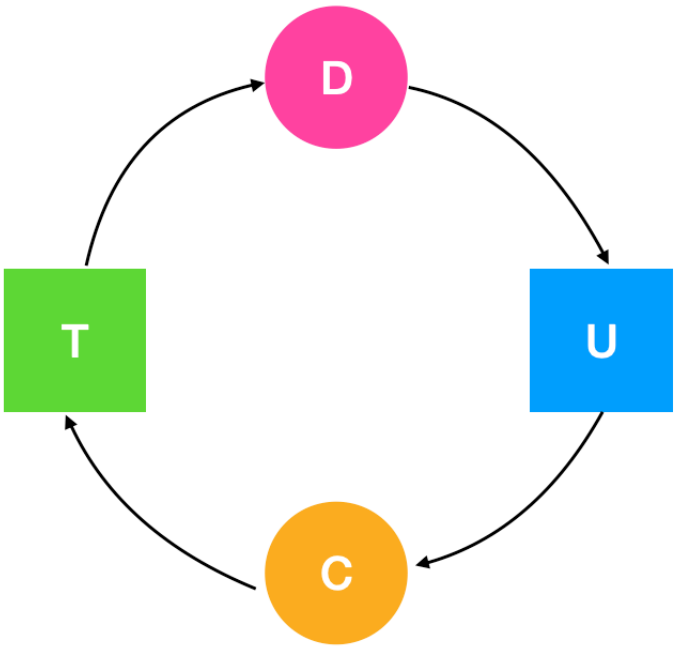
死锁检测和恢复

第二种技术是死锁的检测和恢复。这种解决方式不会尝试去阻止死锁的出现。相反，这种解决方案会希望死锁尽可能的出现，在监测到死锁出现后，对其进行恢复。下面我们就来探讨一下死锁的检测和恢复的几种方式

每种类型一个资源的死锁检测方式

每种资源类型都有一个资源是什么意思？我们经常提到的打印机就是这样的，资源只有打印机，但是设备都不会超过一个。

可以通过构造一张资源分配表来检测这种错误，比如我们上面提到的



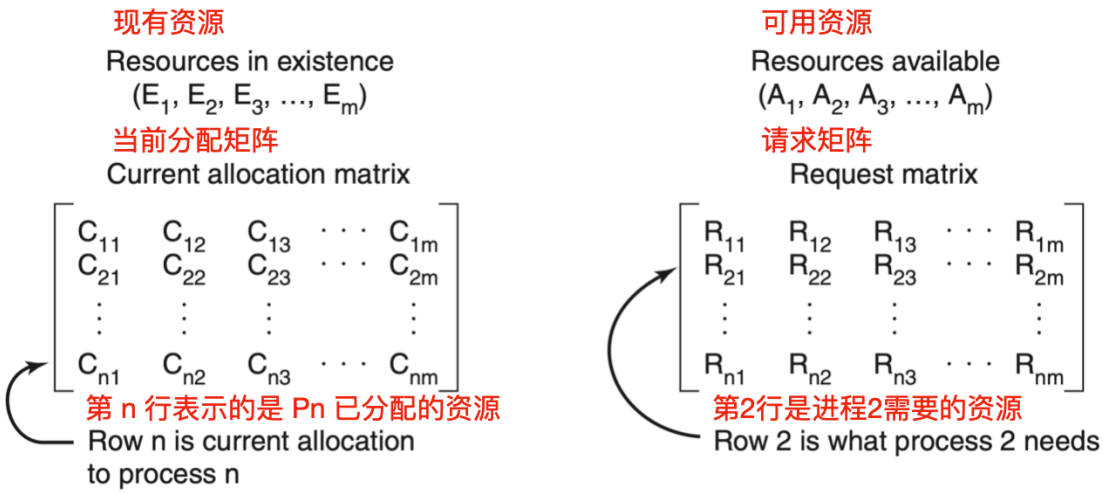
如果这张图包含了一个或一个以上的环，那么死锁就存在，处于这个环中任意一个进程都是死锁的进程。

每种类型多个资源的死锁检测方式

如果有多种相同的资源存在，就需要采用另一种方法来检测死锁。可以通过构造一个矩阵来检测从 P1 -> Pn 这 n 个进程中的死锁。

现在我们提供一种基于矩阵的算法来检测从 P1 到 Pn 这 n 个进程中的死锁。假设资源类型为 m，E1 代表资源类型1，E2 表示资源类型 2 ，Ei 代表资源类型 i (1 <= i <= m)。E 表示的是 `现有资源向量(existing resource vector)`，代表每种已存在的资源总数。

现在我们就需要构造两个数组：C 表示的是 `当前分配矩阵(current allocation matrix)` ，R 表示的是 `请求矩阵(request matrix)` 。Ci 表示的是 Pi 持有每一种类型资源的资源数。所以，Cij 表示 Pi 持有资源 j 的数量。Rij 表示 Pi 所需要获得的资源 j 的数量



一般来说，已分配资源 j 的数量加起来再和所有可供使用的资源数相加 = 该类资源的总数。

死锁的检测就是基于向量的比较。每个进程起初都是没有被标记过的，算法会开始对进程做标记，进程被标记后说明进程被执行了，不会进入死锁，当算法结束时，任何没有被标记过的进程都会被判定为死锁进程。

上面我们探讨了两种检测死锁的方式，那么现在你知道怎么检测后，你何时去做死锁检测呢？一般来说，有两个考量标准：

- 每当有资源请求时就去检测，这种方式会占用昂贵的 CPU 时间。
- 每隔 k 分钟检测一次，或者当 CPU 使用率降低到某个标准下去检测。考虑到 CPU 效率的原因，如果死锁进程达到一定数量，就没有多少进程可以运行，所以 CPU 会经常空闲。

从死锁中恢复

上面我们探讨了如何检测进程死锁，我们最终的目的肯定是想让程序能够正常的运行下去，所以针对检测出来的死锁，我们要对其进行恢复，下面我们会探讨几种死锁的恢复方式

通过抢占进行恢复

在某些情况下，可能会临时将某个资源从它的持有者转移到另一个进程。比如在不通知原进程的情况下，将某个资源从进程中强制取走给其他进程使用，使用完后又送回。这种恢复方式一般比较困难而且有些简单粗暴，并不可取。

通过回滚进行恢复

如果系统设计者和机器操作员知道有可能发生死锁，那么就可以定期检查流程。进程的检测点意味着进程的状态可以被写入到文件以便后面进行恢复。检测点不仅包含 `存储映像(memory image)` ，还包含 `资源状态(resource state)` 。一种更有效的解决方式是不要覆盖原有的检测点，而是每出现一个检测点都要把它写入到文件中，这样当进程执行时，就会有一系列的检查点文件被累积起来。

为了进行恢复，要从上一个较早的检查点上开始，这样所需要资源的进程会回滚到上一个时间点，在这个时间点上，死锁进程还没有获取所需要的资源，可以在此时对其进行资源分配。

杀死进程恢复

最简单有效的解决方案是直接杀死一个死锁进程。但是杀死一个进程可能照样行不通，这时候就需要杀死别的资源进行恢复。

另外一种方式是选择一个环外的进程作为牺牲品来释放进程资源。

死锁避免

我们上面讨论的是如何检测出现死锁和如何恢复死锁，下面我们探讨几种规避死锁的方式

单个资源的银行家算法

银行家算法是 Dijkstra 在 1965 年提出的一种调度算法，它本身是一种死锁的调度算法。它的模型是基于一个城镇中的银行家，银行家向城镇中的客户承诺了一定数量的贷款额度。算法要做的就是判断请求是否会进入一种不安全的状态。如果是，就拒绝请求，如果请求后系统是安全的，就接受该请求。

比如下面的例子，银行家一共为所有城镇居民提供了 15 单位个贷款额度，一个单位表示 1k 美元，如下所示



城镇居民都喜欢做生意，所以就会涉及到贷款，每个人能贷款的最大额度不一样，在某一时刻，A/B/C/D 的贷款金额如下

居民	已使用额度	总额度
A	4	6
B	5	8
C	3	5
D	1	4

空闲：2

上面每个人的贷款总额加起来是 13，马上接近 15，银行家只能给 A 和 C 进行放贷，可以拖着 B 和 D、所以，可以让 A 和 C 首先完成，释放贷款额度，以此来满足其他居民的贷款。这是一种 安全 的状态。

如果每个人的请求导致总额会超过甚至接近 15 ，就会处于一种不安全的状态，如下所示

居民	已使用额度	总额度
A	4	6
B	6	8
C	2	5
D	2	4

空闲：1

这样，每个人还能贷款至少 2 个单位的额度，如果其中有一个人发起最大额度的贷款请求，就会使系统处于一种死锁状态。

> 这里注意一点：不安全状态并不一定引起死锁，由于客户不一定需要其最大的贷款额度，但是银行家不敢抱着这种侥幸心理。

银行家算法就是对每个请求进行检查，检查是否请求会引起不安全状态，如果不会引起，那么就接受该请求；如果会引起，那么就推迟该请求。

类似的，还有多个资源的银行家算法，读者可以自行了解。

破坏死锁

死锁本质上是无法避免的，因为它需要获得未知的资源和请求，但是死锁是满足四个条件后才出现的，它们分别是

- 互斥
- 保持和等待
- 不可抢占
- 循环等待

我们分别对这四个条件进行讨论，按理说破坏其中的任意一个条件就能够破坏死锁

破坏互斥条件

我们首先考虑的就是**破坏互斥使用条件**。如果资源不被一个进程独占，那么死锁肯定不会产生。如果两个打印机同时使用一个资源会造成混乱，打印机的解决方式是使用 `假脱机打印机(spooling printer)` ，这项技术可以允许多个进程同时产生输出，在这种模型中，实际请求打印机的唯一进程是打印机守护进程，也称为后台进程。后台进程不会请求其他资源。我们可以消除打印机的死锁。

后台进程通常被编写为能够输出完整的文件后才能打印，假如两个进程都占用了假脱机空间的一半，而这两个进程都没有完成全部的输出，就会导致死锁。

因此，尽量做到尽可能少的进程可以请求资源。

破坏保持等待的条件

第二种方式是如果我们能阻止持有资源的进程请求其他资源，我们就能够消除死锁。一种实现方式是让所有的进程开始执行前请求全部的资源。如果所需的资源可用，进程会完成资源的分配并运行到结束。如果有任何一个资源处于频繁分配的情况，那么没有分配到资源的进程就会等待。

很多进程**无法在执行完成前就知道到底需要多少资源**，如果知道的话，就可以使用银行家算法；还有一个问题是这样**无法合理有效利用资源**。

还有一种方式是进程在请求其他资源时，先释放所占用的资源，然后再尝试一次获取全部的资源。

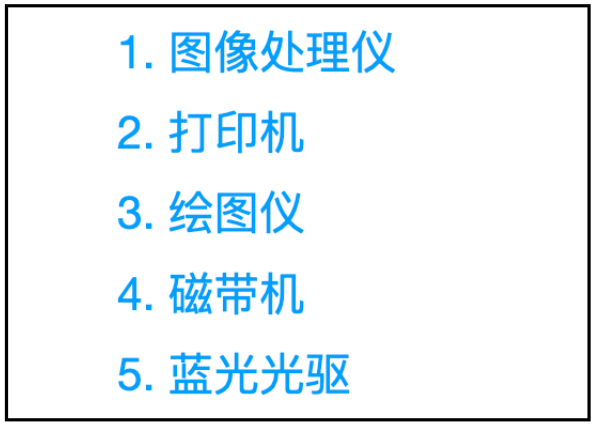
破坏不可抢占条件

破坏不可抢占条件也是可以的。可以通过虚拟化的方式来避免这种情况。

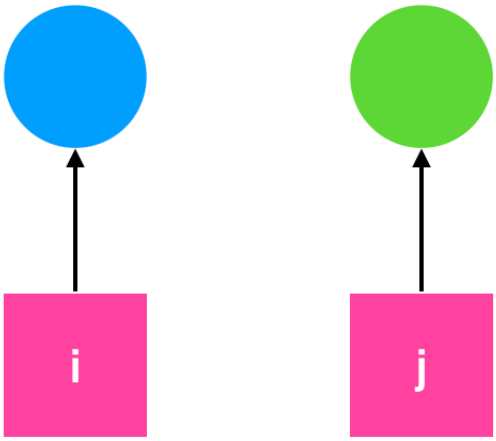
破坏循环等待条件

现在就剩最后一个条件了，循环等待条件可以通过多种方法来破坏。一种方式是制定一个标准，一个进程在任何时候只能使用一种资源。如果需要另外一种资源，必须释放当前资源。对于需要将大文件从磁带复制到打印机的过程，此限制是不可接受的。

另一种方式是将所有的资源统一编号，如下图所示



进程可以在任何时间提出请求，但是所有的请求都必须按照资源的顺序提出。如果按照此分配规则的话，那么资源分配之间不会出现环。



尽管通过这种方式来消除死锁，但是编号的顺序不可能让每个进程都会接受。

其他问题

下面我们来探讨一下其他问题，包括 **通信死锁、活锁是什么、饥饿问题和两阶段加锁**

两阶段加锁

虽然很多情况下死锁的避免和预防都能处理，但是效果并不好。随着时间的推移，提出了很多优秀的算法用来处理死锁。例如在数据库系统中，一个经常发生的操作是请求锁住一些记录，然后更新所有锁定的记录。当同时有多个进程运行时，就会有死锁的风险。

一种解决方式是使用 `两阶段提交(two-phase locking)` 。顾名思义分为两个阶段，一阶段是进程尝试一次锁定它需要的所有记录。如果成功后，才会开始第二阶段，第二阶段是执行更新并释放锁。第一阶段并不做真正有意义的工作。

如果在第一阶段某个进程所需要的记录已经被加锁，那么该进程会释放所有锁定的记录并重新开始第一阶段。从某种意义上来说，这种方法类似于预先请求所有必需的资源或者是在进行一些不可逆的操作之前请求所有的资源。

不过在一般的应用场景中，两阶段加锁的策略并不通用。如果一个进程缺少资源就会半途中断并重新开始的方式是不可接受的。

通信死锁

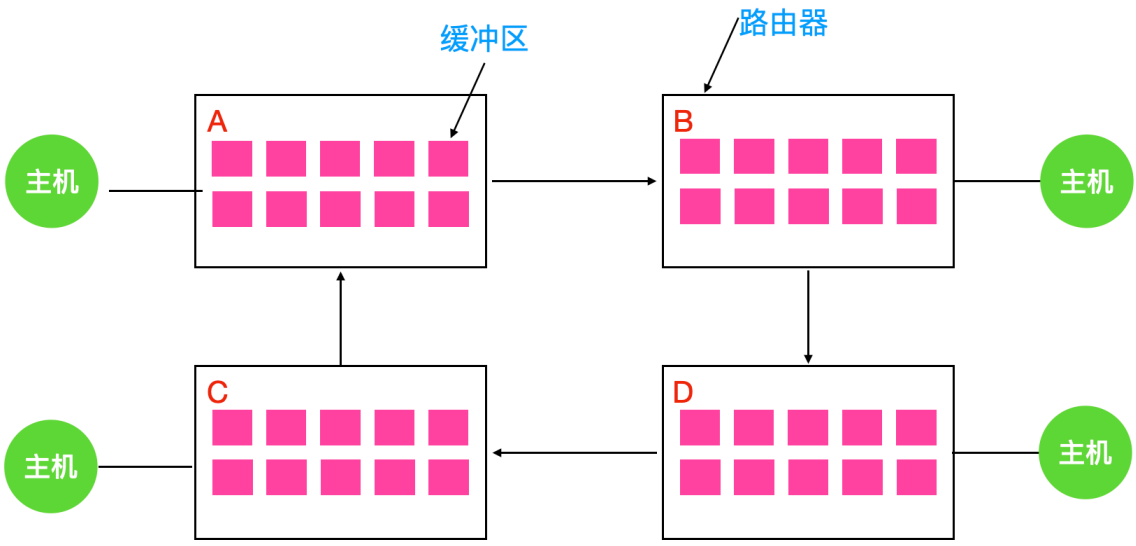
我们上面一直讨论的是资源死锁，资源死锁是一种死锁类型，但并不是唯一类型，还有通信死锁，也就是两个或多个进程在发送消息时出现的死锁。进程 A 给进程 B 发了一条消息，然后进程 A 阻塞直到进程 B 返回响应。假设请求消息丢失了，那么进程 A 在一直等着回复，进程 B 也会阻塞等待请求消息到来，这时候就产生 `死锁` 。

尽管会产生死锁，但是这并不是一个资源死锁，因为 A 并没有占据 B 的资源。事实上，通信死锁并没有完全可见的资源。根据死锁的定义来说：每个进程因为等待其他进程引起的事件而产生阻塞，这就是一种死锁。相较于最常见的通信死锁，我们把上面这种情况称为 `通信死锁(communication deadlock)` 。

通信死锁不能通过调度的方式来避免，但是可以使用通信中一个非常重要的概念来避免： `超时(timeout)` 。在通信过程中，只要一个信息被发出后，发送者就会启动一个定时器，定时器会记录消息的超时时间，如果超时时间到了但是消息还没有返回，就会认为消息已经丢失并重新发送，通过这种方式，可以避免通信死锁。

但是并非所有网络通信发生的死锁都是通信死锁，也存在资源死锁，下面就是一个典型的资源死锁。

当一个数据包从主机进入路由器时，会被放入一个缓冲区，然后再传输到另外一个路由器，再到另一个，以此类推直到目的地。缓冲区都是资源并且数量有限。如下图所示，每个路由器都有 10 个缓冲区（实际上有很多）。



假如路由器 A 的所有数据需要发送到 B ， B 的所有数据包需要发送到 D，然后 D 的所有数据包需要发送到 A 。没有数据包可以移动，因为在另一端没有缓冲区可用，这就是一个典型的资源死锁。

活锁

你会发现一个很有意思的事情，死锁就跟榆木脑袋一样，不会转弯。我看过古代的一则故事：

> 更具体些的故事是这样的：春秋时，鲁国曲阜有个年轻人名叫尾生，为人正直，乐于助人，和朋友交往很守信用，受到四乡八邻的普遍赞誉。后来，尾生迁居梁地（今陕西韩城南）。他在那里认识了一位年轻漂亮的姑娘。两人一见钟情，君子淑女，私订终身。但是姑娘的父母嫌弃尾生家境贫寒，坚决反对这门亲事。为了追求爱情和幸福，姑娘决定背着父母私奔，随尾生回到曲阜老家去。那一天，两人约定在韩城外的一座木桥边会面，双双远走高飞。黄昏时分，尾生提前来到桥上等候。不料，六月的天气说变就变，突然乌云密布，狂风怒吼，雷鸣电闪，滂沱大雨倾盆而下。不久山洪暴发，滚滚江水裹挟泥沙席卷而来，淹没了桥面，没过了尾生的膝盖。城外桥面，不见不散，尾生想起了与姑娘的信誓旦旦；四顾茫茫水世界，不见姑娘踪影。但他寸步不离，死死抱着桥柱，终于被活活淹死。再说姑娘因为私奔念头泄露，被父母禁锢家中，不得脱身。后伺机禽夜逃出家门，冒雨来到城外桥边，此时洪水已渐渐退去。姑娘看到紧抱桥柱而死的尾生，悲恸欲绝。她抱着尾生的尸体号啕大哭。阴阳相隔，生死一体，哭罢，便相拥纵身投入滚滚江中……？

如果说死锁很 `痴情` 的话，那么 `活锁` 用一则成语来表示就是 `弄巧成拙` 。

某些情况下，当进程意识到它不能获取所需要的下一个锁时，就会尝试礼貌的释放已经获得的锁，然后等待非常短的时间再次尝试获取。可以想像一下这个场景：当两个人在狭路相逢的时候，都想给对方让路，相同的步调会导致双方都无法前进。

现在假想有一对并行的进程用到了两个资源。它们分别尝试获取另一个锁失败后，两个进程都会释放自己持有的锁，再次进行尝试，这个过程会一直进行重复。很明显，这个过程中没有进程阻塞，但是进程仍然不会向下执行，这种状况我们称之为 `活锁(livelock)` 。

饥饿

与死锁和活锁的一个非常相似的问题是 `饥饿(starvvation)` 。想象一下你什么时候会饿？一段时间不吃东西是不是会饿？对于进程来讲，最重要的就是资源，如果一段时间没有获得资源，那么进程会产生饥饿，这些进程会永远得不到服务。

我们假设打印机的分配方案是每次都会分配给最小文件的进程，那么要打印大文件的进程会永远得不到服务，导致进程饥饿，进程会无限制的推后，虽然它没有阻塞。

总结

死锁是一类通用问题，任何操作系统都会产生死锁。当每一组进程中的每个进程都因等待由该组的其他进程所占有的资源而导致阻塞，死锁就发生了。这种情况会使所有的进程都处于无限等待的状态。

死锁的检测和避免可以通过安全和不安全状态来判断，其中一个检测方式就是银行家算法；当然你也可以使用鸵鸟算法对死锁置之不理，但是你肯定会遭其反噬。

也可以在设计时通过系统结构的角度来避免死锁，这样能够预防死锁；也可以破坏死锁的四个条件来破坏死锁。资源死锁并不是唯一性的死锁，还有通信间死锁，可以设置适当的超时时间来完成。

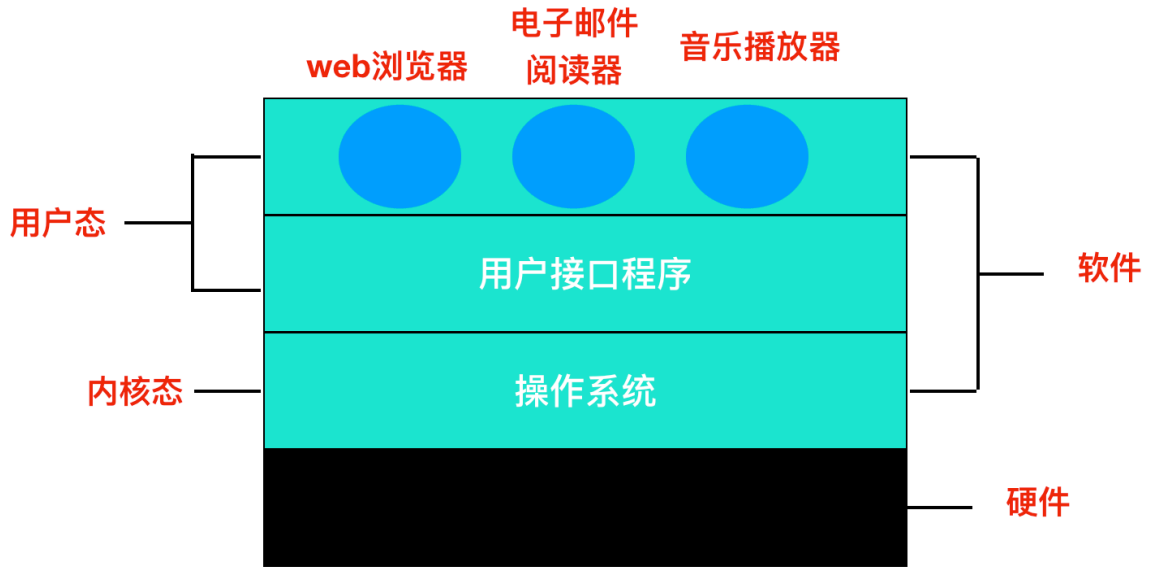
活锁和死锁的问题有些相似，它们都是一种进程无法继续向下执行的状态。由于进程调度策略导致尝试获取进程的一方永远无法获得资源后，进程会导致饥饿的出现。

操作系统面试题



解释一下什么是操作系统

操作系统是运行在计算机上最重要的一种 软件 ，它管理计算机的资源和进程以及所有的硬件和软件。它为计算机硬件和软件提供了一种中间层



操作系统所处的位置

通常情况下，计算机上会运行着许多应用程序，它们都需要对内存和 CPU 进行交互，操作系统的目的就是为了保证这些访问和交互能够准确无误的进行。

解释一下操作系统的主要目的是什么

操作系统是一种软件，它的主要目的有三种

- 管理计算机资源，这些资源包括 CPU、内存、磁盘驱动器、打印机等。
- 提供一种图形界面，就像我们前面描述的那样，它提供了用户和计算机之间的桥梁。
- 为其他软件提供服务，操作系统与软件进行交互，以便为其分配运行所需的任何必要资源。

操作系统的种类有哪些

操作系统通常预装在你购买计算机之前。大部分用户都会使用默认的操作系统，但是你也可以升级甚至更改操作系统。但是一般常见的操作系统只有三种：**Windows、macOS 和 Linux**。

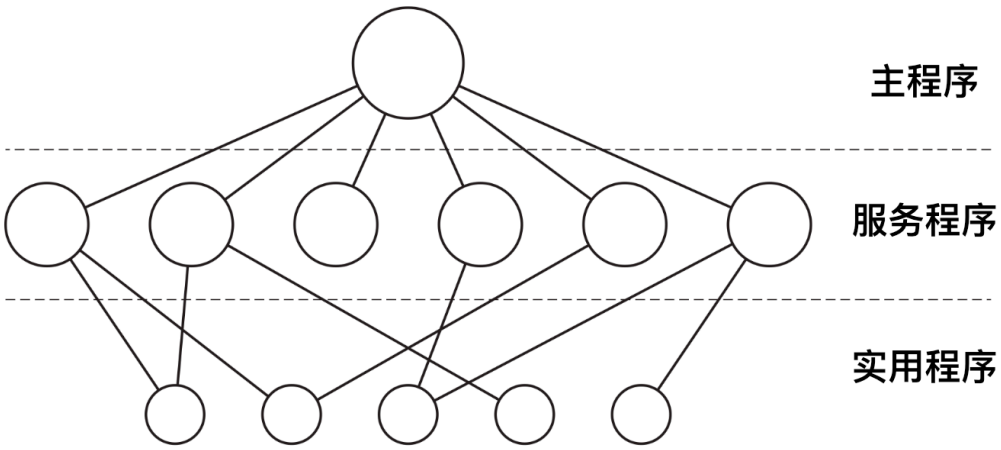
操作系统结构

单体系统

在大多数系统中，整个系统在内核态以单一程序的方式运行。整个操作系统是以程序集合来编写的，链接在一块形成一个大的二进制可执行程序，这种系统称为单体系统。

在单体系统中构造实际目标程序时，会首先编译所有单个过程（或包含这些过程的文件），然后使用系统链接器将它们全部绑定到一个可执行文件中

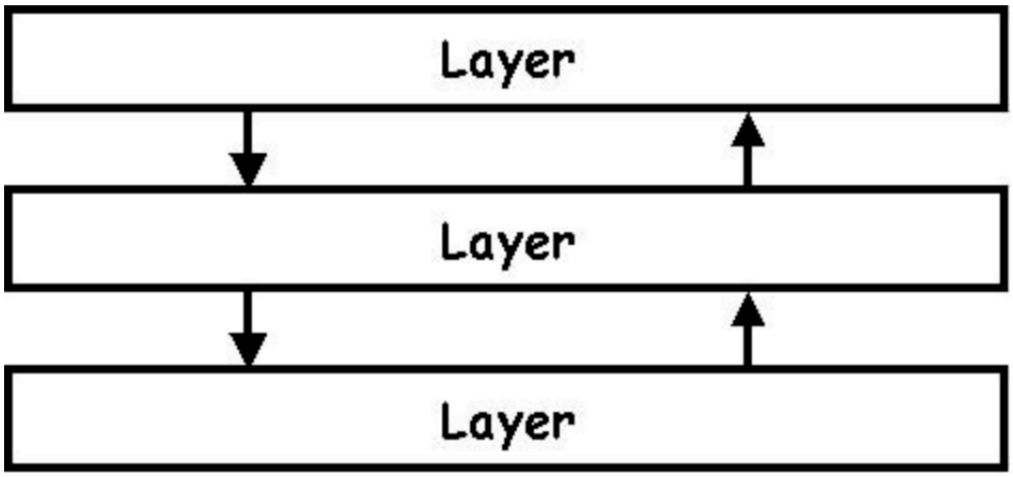
在单体系统中，对于每个系统调用都会有一个服务程序来保障和运行。需要一组实用程序来弥补服务程序需要的功能，例如从用户程序中获取数据。可将各种过程划分为一个三层模型



除了在计算机初启动时所装载的核心操作系统外，许多操作系统还支持额外的扩展。比如 I/O 设备驱动和文件系统。这些部件可以按需装载。在 UNIX 中把它们叫做 `共享库(shared library)`，在 Windows 中则被称为 `动态链接库(Dynamic Link Library,DLL)`。他们的扩展名为 `.dll`，在 `C:\Windows\system32` 目录下存在 1000 多个 DLL 文件，所以不要轻易删除 C 盘文件，否则可能就炸了哦。

分层系统

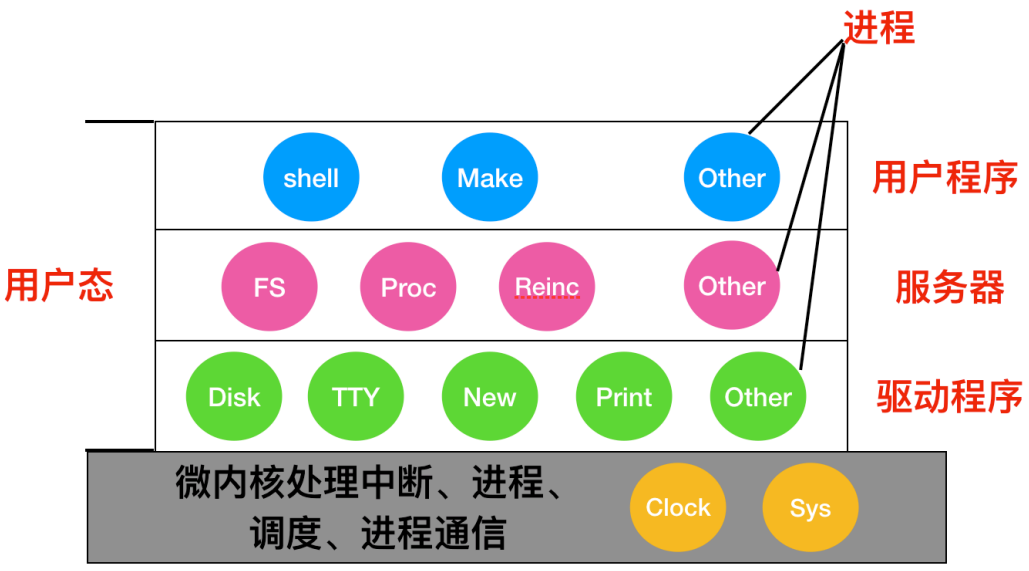
分层系统使用层来分隔不同的功能单元。每一层只与该层的上层和下层通信。每一层都使用下面的层来执行其功能。层之间的通信通过预定义的固定接口通信。



微内核

为了实现高可靠性，将操作系统划分成小的、层级之间能够更好定义的模块是很有必要的，只有一个模块 --- 微内核 --- 运行在内核态，其余模块可以作为普通用户进程运行。由于把每个设备驱动和文件系统分别作为普通用户进程，这些模块中的错误虽然会使这些模块崩溃，但是不会使整个系统死机。

`MINIX 3` 是微内核的代表作，它的具体结构如下



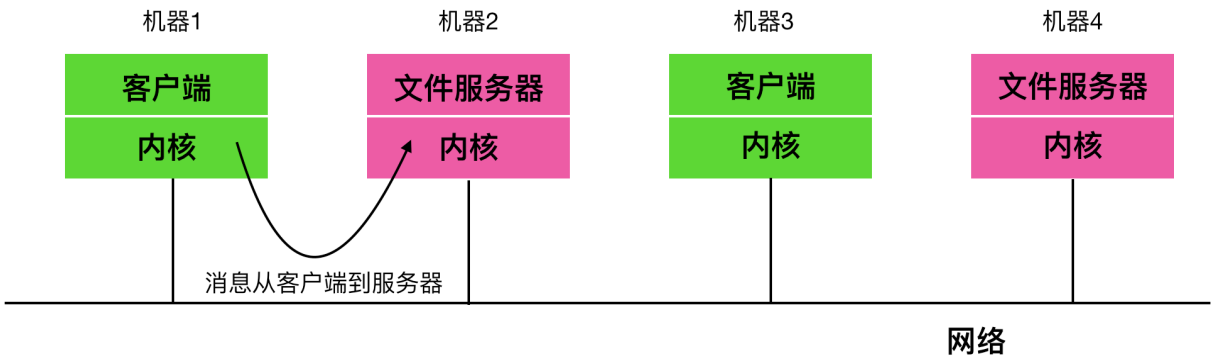
MINI 3 系统结构

在内核的外部，系统的构造有三层，它们都在用户态下运行，最底层是设备驱动器。由于它们都在用户态下运行，所以不能物理的访问 I/O 端口空间，也不能直接发出 I/O 命令。相反，为了能够对 I/O 设备编程，驱动器构建一个结构，指明哪个参数值写到哪个 I/O 端口，并声称一个内核调用，这样就完成了一次调用过程。

客户-服务器模式

微内核思想的策略是把进程划分为两类： 服务器 ，每个服务器用来提供服务； 客户端 ，使用这些服务。这个模式就是所谓的 客户-服务器 模式。

客户-服务器模式会有两种载体，一种情况是一台计算机既是客户又是服务器，在这种方式下，操作系统会有某种优化；但是普遍情况下是客户端和服务器在不同的机器上，它们通过局域网或广域网连接。



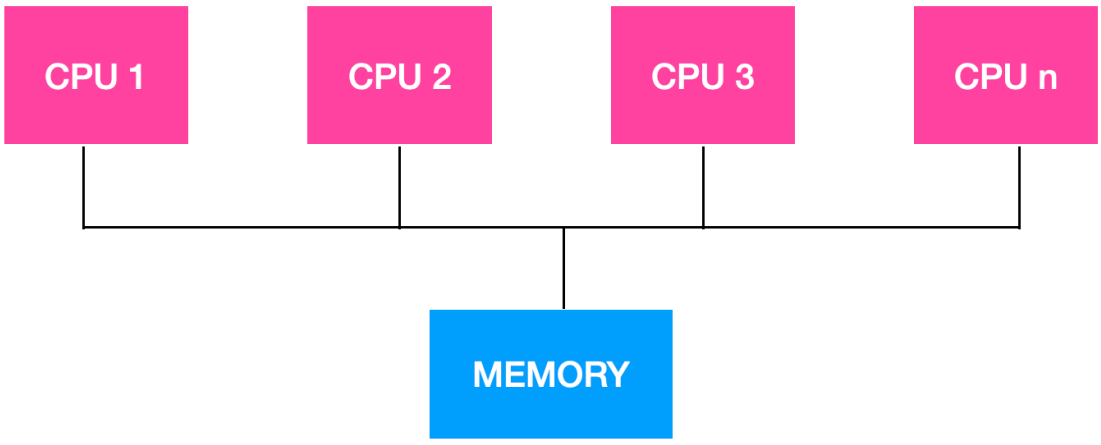
客户通过发送消息与服务器通信，客户端并不需要知道这些消息是在本地机器上处理，还是通过网络被送到远程机器上处理。对于客户端而言，这两种情形是一样的：都是发送请求并得到回应。

什么是按需分页

在操作系统中，进程是以页为单位加载到内存中的，按需分页是一种 虚拟内存 的管理方式。在使用请求分页的系统中，只有在尝试访问页面所在的磁盘并且该页面尚未在内存中时，也就发生了 缺页异常 ，操作系统才会将磁盘页面复制到内存中。

多处理系统的优势

随着处理器的不断增加，我们的计算机系统由单机系统变为了多处理系统，多处理系统的吞吐量比较高，多处理系统拥有多个并行的处理器，这些处理器共享时钟、内存、总线、外围设备等。



多处理系统由于可以共享资源，因此可以开源节流，省钱。整个系统的可靠性也随之提高。

什么是内核

在计算机中，内核是一个计算机程序，它是操作系统的核心，可以控制操作系统中所有的内容。内核通常是在 boot loader 装载程序之前加载的第一个程序。

这里还需要了解一下什么是 `boot loader` 。

> boot loader 又被称为引导加载程序，它是一个程序，能够将计算机的操作系统放入内存中。在电源通电或者计算机重启时，BIOS 会执行一些初始测试，然后将控制权转移到引导加载程序所在的 `主引导记录(MBR)` 。

什么是实时系统

实时操作系统对时间做出了严格的要求，实时操作系统分为两种：**硬实时和软实时**

`硬实时操作系统` 规定某个动作必须在规定的时刻内完成或发生，比如汽车生产车间，焊接机器必须在某一时刻内完成焊接，焊接的太早或者太晚都会对汽车造成永久性伤害。

`软实时操作系统` 虽然不希望偶尔违反最终的时限要求，但是仍然可以接受。并且不会引起任何永久性伤害。比如数字音频、多媒体、手机都是属于软实时操作系统。

你可以简单理解硬实时和软实时的两个指标：**是否在时刻内必须完成以及是否造成严重损害**。

什么是虚拟内存

`虚拟内存` 是一种内存分配方案，是一项可以用来辅助内存分配的机制。我们知道，应用程序是按页装载进内存中的。但并不是所有的页都会装载到内存中，计算机中的硬件和软件会将数据从 RAM 临时传输到磁盘中来弥补内存的不足。如果没有虚拟内存的话，一旦你将计算机内存填满后，计算机会对你说



呃，不，**对不起，您无法再加载任何应用程序，请关闭另一个应用程序以加载新的应用程序**。对于虚拟内存，计算机可以执行操作是查看内存中最近未使用过的区域，然后将其复制到硬盘上。虚拟内存通过复制技术实现了 **妹子，你快来看哥哥能装这么多程序** 的资本。复制是自动进行的，你无法感知到它的存在。

什么是进程和进程表

`进程` 就是正在执行程序的实例，比如说 Web 程序就是一个进程，shell 也是一个进程，文章编辑器 typora 也是一个进程。

操作系统负责管理所有正在运行的进程，操作系统会为每个进程分配特定的时间来占用 CPU，操作系统还会为每个进程分配特定的资源。

操作系统为了跟踪每个进程的活动状态，维护了一个 `进程表` 。在进程表的内部，列出了每个进程的状态以及每个进程使用的资源等。

http://courses.cs.vt.edu/csonline/OS/Lessons/Processes/index.html 这个网站上面有一个关于进程状态轮转的动画，做的真是太好了。

什么是线程，线程和进程的区别

这又是一道老生常谈的问题了，从操作系统的角度来回答一下吧。

我们上面说到进程是正在运行的程序的实例，而线程其实就是进程中的单条流向，因为线程具有进程中的某些属性，所以线程又被称为轻量级的进程。浏览器如果是一个进程的话，那么浏览器下面的每个 tab 页可以看作是一个个的线程。

下面是线程和进程持有资源的区别

每个进程中的内容	每个线程中的内容
地址空间	程序计数器
全局变量	寄存器
打开文件	堆栈
子进程	状态
即将发生的定时器	
信号与信号处理程序	
账户信息	

线程不像进程那样具有很强的独立性，线程之间会共享数据

创建线程的开销要比进程小很多，因为创建线程仅仅需要 `堆栈指针` 和 `程序计数器` 就可以了，而创建进程需要操作系统分配新的地址空间，数据资源等，这个开销比较大。

使用多线程的好处是什么

多线程是程序员不得不知的基本素养之一，所以，下面我们给出一些多线程编程的好处

- 能够提高对用户的响应顺序
- 在流程中的资源共享
- 比较经济适用
- 能够对多线程架构有深入的理解

什么是 RR 调度算法

`RR(round-robin)` 调度算法主要针对分时系统，RR 的调度算法会把时间片以相同的部分并循环的分配给每个进程，RR 调度算法没有优先级的概念。这种算法的实现比较简单，而且每个线程都会占有时间片，并不存在线程饥饿的问题。

导致系统出现死锁的情况

死锁的出现需要同时满足下面四个条件

- `互斥(Mutual Exclusion)`：一次只能有一个进程使用资源。如果另一个进程请求该资源，则必须延迟请求进程，直到释放该资源为止。
- `保持并等待(Hold and Wait)`：必须存在一个进程，该进程至少持有一个资源，并且正在等待获取其他进程当前所持有的资源。
- `无抢占(No Preemption)`：资源不能被抢占，也就是说，在进程完成其任务之后，只能由拥有它的进程自动释放资源。
- `循环等待(Circular Wait)`：必须存在一组 {p0，p1，..... pn} 的等待进程，使 p0 等待 p1 持有的资源，p1 等待由 p2 持有的资源， pn-1 正在等待由 pn 持有的资源，而 pn 正在等待由 p0 持有的资源。

RAID 的不同级别

RAID 称为 `磁盘冗余阵列`，简称 `磁盘阵列`。利用虚拟化技术把多个硬盘结合在一起，成为一个或多个磁盘阵列组，目的是提升性能或数据冗余。

RAID 有不同的级别

- RAID 0 - 无容错的条带化磁盘阵列
- RAID 1 - 镜像和双工
- RAID 2 - 内存式纠错码
- RAID 3 - 比特交错奇偶校验

- RAID 4 - 块交错奇偶校验
- RAID 5 - 块交错分布式奇偶校验
- RAID 6 - P + Q冗余

什么是 DMA

DMA 的中文名称是 直接内存访问 ，它意味着 CPU 授予 I/O 模块权限在不涉及 CPU 的情况下读取或写入内存。也就是 DMA 可以不需要 CPU 的参与。这个过程由称为 DMA 控制器（DMAC）的芯片管理。由于 DMA 设备可以直接在内存之间传输数据，而不是使用 CPU 作为中介，因此可以缓解总线上的拥塞。DMA 通过允许 CPU 执行任务，同时 DMA 系统通过系统和内存总线传输数据来提高系统并发性。

多线程编程的好处是什么

对不起，我忍不住想偷笑



说直白点，为什么单线程能够处理的却要用多线程来处理？当然是为了提高程序的 ~~装逼~~ 并行能力了。多线程 在某些情况下 能够使你程序运行的更快，这也是为什么多核 CPU 会出现，但是多核 CPU 的出现会导致数据的一致性问题，不过这些问题程序员就能解决。另一个角度来说，多线程编程能够提高程序员的编程能力和编程思维。同时也能提高程序员的管理能力，你如果把每条线程流当作罗老师时间管理的女主一样，能够及时协调好所有P友的关系，那你也是超神程序员了，所以，是谁说程序员不会做管理的？Doug Lea 大佬牛逼！！！

> ps：Doug Lea 大佬开发的 JUC 工具包，此处不加狗头。

什么是设备驱动程序

在计算机中，设备驱动程序是一种计算机程序，它能够控制或者操作连接到计算机的特定设备。驱动程序提供了与硬件进行交互的软件接口，使操作系统和其他计算机程序能够访问特定设备，不用需要了解其硬件的具体构造。

进程间的通信方式

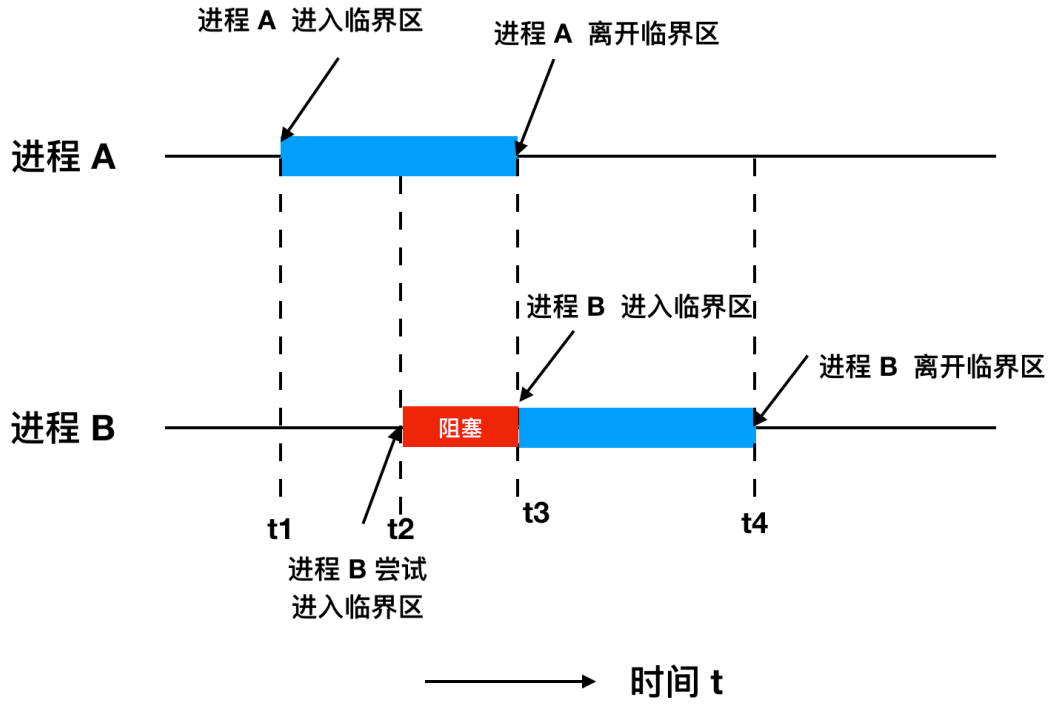
通信概念

进程间的通信方式比较多，首先你需要理解下面这几个概念

- 竞态条件：即两个或多个线程同时对一共享数据进行修改，从而影响程序运行的正确性时，这种就被称为 `竞态条件(race condition)` 。
- 临界区：不仅 `共享资源` 会造成竞态条件，事实上共享文件、共享内存也会造成竞态条件、那么该如何避免呢？或许一句话可以概括说明：**禁止一个或多个进程在同一时刻对共享资源（包括共享内存、共享文件等）进行读写**。换句话说，我们需要一种 `互斥(mutual exclusion)` 条件，这也就是说，如果一个进程在某种方式下使用共享变量和文件的话，除该进程之外的其他进程就禁止做这种事（访问统一资源）。

  一个好的解决方案，应该包含下面四种条件

  1. 任何时候两个进程不能同时处于临界区
  2. 不应对 CPU 的速度和数量做任何假设
  3. 位于临界区外的进程不得阻塞其他进程
  4. 不能使任何进程无限等待进入临界区



  使用临界区的互斥

- 忙等互斥：当一个进程在对资源进行修改时，其他进程必须进行等待，进程之间要具有互斥性，我们讨论的解决方案其实都是基于忙等互斥提出的。

解决方案

进程间的通信用专业一点的术语来表示就是 `Inter Process Communication, IPC` ，它主要有下面几种通信方式



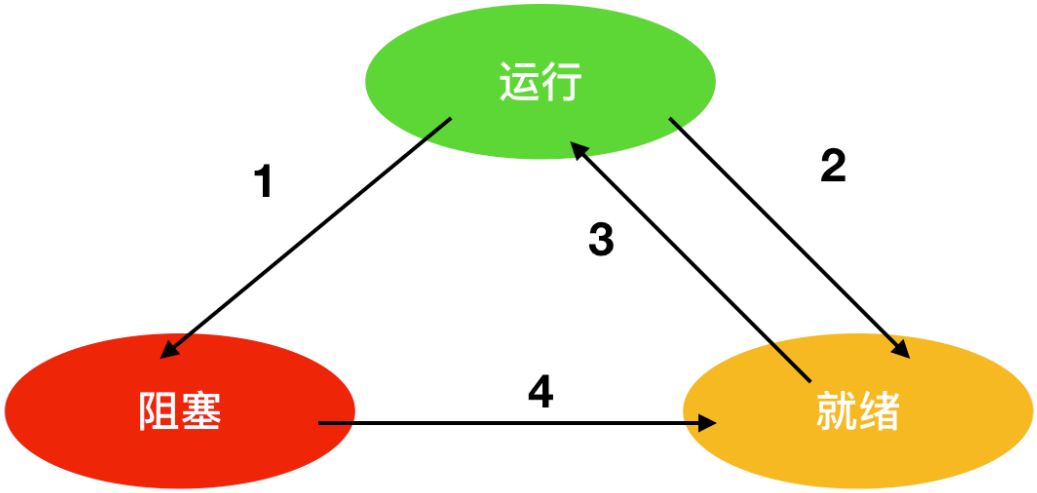
- 消息传递 ： 消息传递是进程间实现通信和同步等待的机制，使用消息传递，进程间的交流不需要共享变量，直接就可以进行通信；消息传递分为发送方和接收方
- 先进先出队列 ： 先进先出队列指的是两个不相关联进程间的通信，两个进程之间可以彼此相互进程通信，这是一种全双工通信方式
- 管道 ： 管道用于两个相关进程之间的通信，这是一种半双工的通信方式，如果需要全双工，需要另外一个管道。
- 直接通信 ： 在这种进程通信的方式中，进程与进程之间只存在一条链接，进程间要明确通信双方的命名。
- 间接通信 ： 间接通信是通信双方不会直接建立连接，而是找到一个中介者，这个中介者可能是个对象等等，进程可以在其中放置消息，并且可以从中删除消息，以此达到进程间通信的目的。
- 消息队列 ： 消息队列是内核中存储消息的链表，它由消息队列标识符进行标识，这种方式能够在不同的进程之间提供全双工的通信连接。
- 共享内存 ： 共享内存是使用所有进程之间的内存来建立连接，这种类型需要同步进程访问来相互保护。

进程间状态模型

```
1   cat chapter1 chapter2 chapter3 | grep tree
```

第一个进程是 `cat`，将三个文件级联并输出。第二个进程是 `grep`，它从输入中选择具有包含关键字 `tree` 的内容，根据这两个进程的相对速度（这取决于两个程序的相对复杂度和各自所分配到的 CPU 时间片），可能会发生下面这种情况， `grep` 准备就绪开始运行，但是输入进程还没有完成，于是必须阻塞 grep 进程，直到输入完毕。

当一个进程开始运行时，它可能会经历下面这几种状态



- 第一种是进程因为等待输入而阻塞
- 第二种是调度程序选择另一个进程
- 第三种是调度程序选择一个进程开始运行
- 第四种是出现有效的输入

进程间的状态切换图

图中会涉及三种状态

1. `运行态`，运行态指的就是进程实际占用 CPU 时间片运行时
2. `就绪态`，就绪态指的是可运行，但因为其他进程正在运行而处于就绪状态
3. `阻塞态`，除非某种外部事件发生，否则进程不能运行

逻辑上来说，运行态和就绪态是很相似的。这两种情况下都表示进程 `可运行`，但是第二种情况没有获得 CPU 时间分片。第三种状态与前两种状态不同的原因是这个进程不能运行，CPU 空闲时也不能运行。

三种状态会涉及四种状态间的切换，在操作系统发现进程不能继续执行时会发生 `状态1` 的轮转，在某些系统中进程执行系统调用，例如 `pause`，来获取一个阻塞的状态。在其他系统中包括 UNIX，当进程从管道或特殊文件（例如终端）中读取没有可用的输入时，该进程会被自动终止。

转换 2 和转换 3 都是由进程调度程序（操作系统的一部分）引起的，进程本身不知道调度程序的存在。转换 2 的出现说明进程调度器认定当前进程已经运行了足够长的时间，是时候让其他进程运行 CPU 时间片了。当所有其他进程都运行过后，这时候该是让第一个进程重新获得 CPU 时间片的时候了，就会发生转换 3。

> **程序调度指的是，决定哪个进程优先被运行和运行多久，这是很重要的一点**。已经设计出许多算法来尝试平衡系统整体效率与各个流程之间的竞争需求。

当进程等待的一个外部事件发生时（如从外部输入一些数据后），则发生转换 4。如果此时没有其他进程在运行，则立刻触发转换 3，该进程便开始运行，否则该进程会处于就绪阶段，等待 CPU 空闲后再轮到它运行。

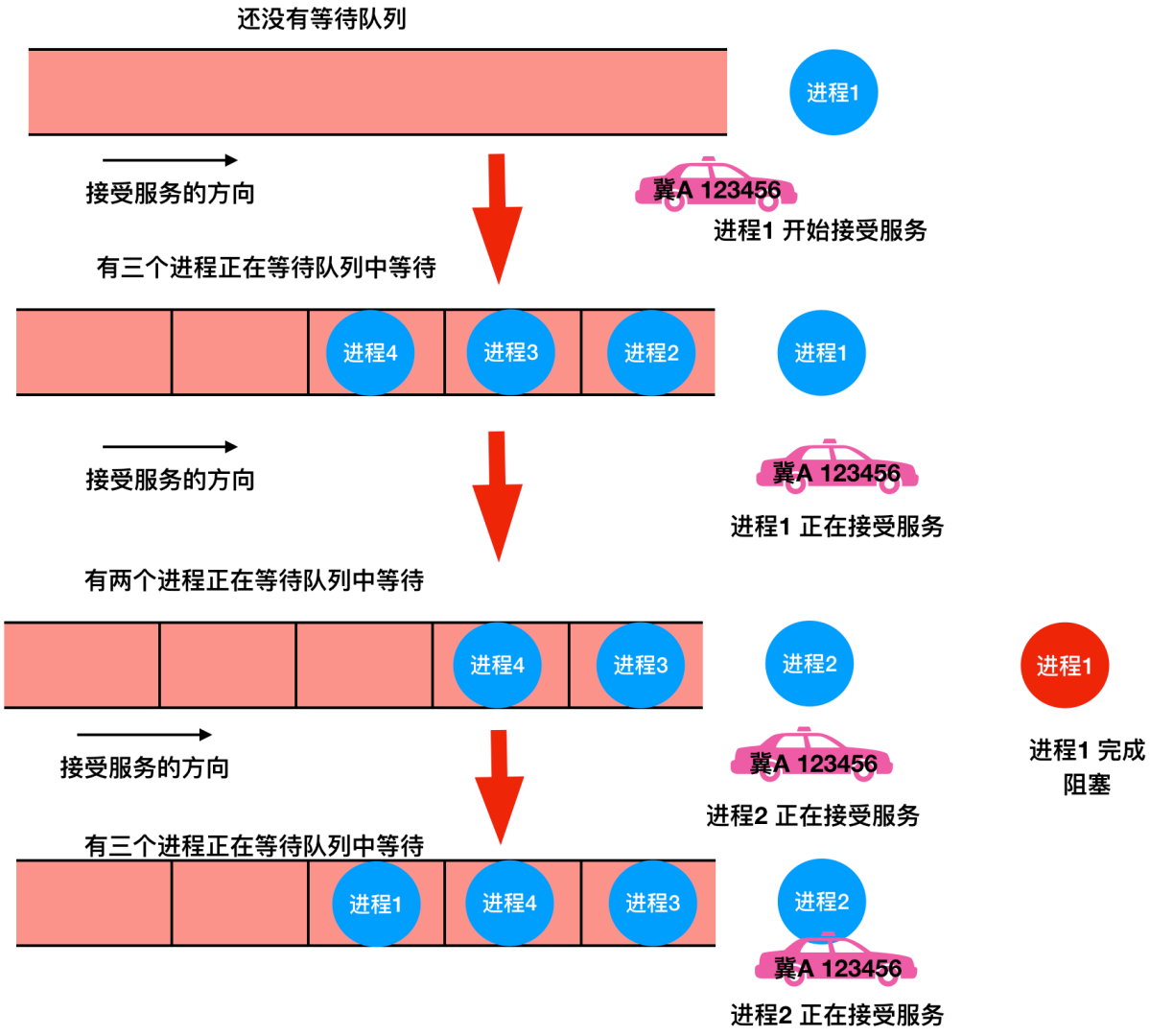
调度算法都有哪些

调度算法分为三大类：批处理中的调度、交互系统中的调度、实时系统中的调度

批处理中的调度

先来先服务

很像是先到先得。。。可能最简单的非抢占式调度算法的设计就是 `先来先服务(first-come,first-serverd)` 。使用此算法，将按照请求顺序为进程分配 CPU。最基本的，会有一个就绪进程的等待队列。当第一个任务从外部进入系统时，将会立即启动并允许运行任意长的时间。它不会因为运行时间太长而中断。当其他作业进入时，它们排到就绪队列尾部。当正在运行的进程阻塞，处于等待队列的第一个进程就开始运行。当一个阻塞的进程重新处于就绪态时，它会像一个新到达的任务，会排在队列的末尾，即排在所有进程最后。

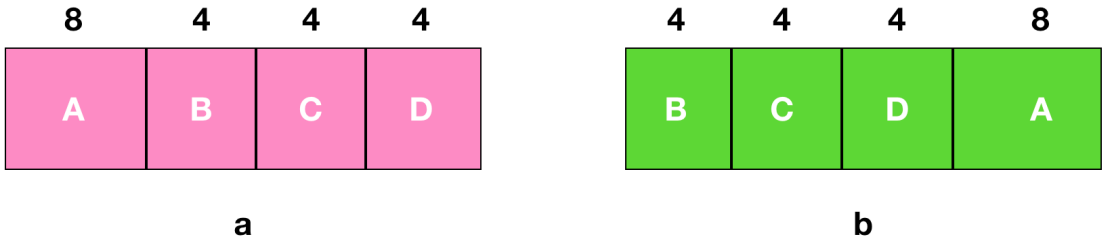


这个算法的强大之处在于易于理解和编程，在这个算法中，一个单链表记录了所有就绪进程。要选取一个进程运行，只要从该队列的头部移走一个进程即可；要添加一个新的作业或者阻塞一个进程，只要把这个作业或进程附加在队列的末尾即可。这是很简单的一种实现。

不过，先来先服务也是有缺点的，那就是没有优先级的关系，试想一下，如果有 100 个 I/O 进程正在排队，第 101 个是一个 CPU 密集型进程，那岂不是需要等 100 个 I/O 进程运行完毕才会等到一个 CPU 密集型进程运行，这在实际情况下根本不可能，所以需要优先级或者抢占式进程的出现来优先选择重要的进程运行。

最短作业优先

批处理中，第二种调度算法是 `最短作业优先(Shortest Job First)`，我们假设运行时间已知。例如，一家保险公司，因为每天要做类似的工作，所以人们可以相当精确地预测处理 1000 个索赔的一批作业需要多长时间。当输入队列中有若干个同等重要的作业被启动时，调度程序应使用最短优先作业算法



如上图 a 所示，这里有 4 个作业 A、B、C、D，运行时间分别为 8、4、4、4 分钟。若按图中的次序运行，则 A 的周转时间为 8 分钟，B 为 12 分钟，C 为 16 分钟，D 为 20 分钟，平均时间内为 14 分钟。

现在考虑使用最短作业优先算法运行 4 个作业，如上图 b 所示，目前的周转时间分别为 4、8、12、20，平均为 11 分钟，可以证明最短作业优先是最优的。考虑有 4 个作业的情况，其运行时间分别为 a、b、c、d。第一个作业在时间 a 结束，第二个在时间 a + b 结束，以此类推。平均周转时间为 (4a + 3b + 2c + d) / 4 。显然 a 对平均值的影响最大，所以 a 应该是最短优先作业，其次是 b，然后是 c，最后是 d 它就只能影响自己的周转时间了。

> 需要注意的是，在所有的进程都可以运行的情况下，最短作业优先的算法才是最优的。

最短剩余时间优先

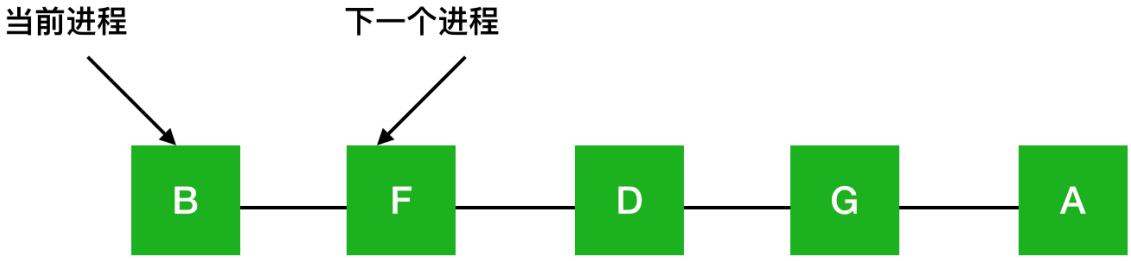
最短作业优先的抢占式版本被称作为 `最短剩余时间优先(Shortest Remaining Time Next)` 算法。使用这个算法，调度程序总是选择剩余运行时间最短的那个进程运行。当一个新作业到达时，其整个时间同当前进程的剩余时间做比较。如果新的进程比当前运行进程需要更少的时间，当前进程就被挂起，而运行新的进程。这种方式能够使短期作业获得良好的服务。

交互式系统中的调度

交互式系统中在个人计算机、服务器和其他系统中都是很常用的，所以有必要来探讨一下交互式调度

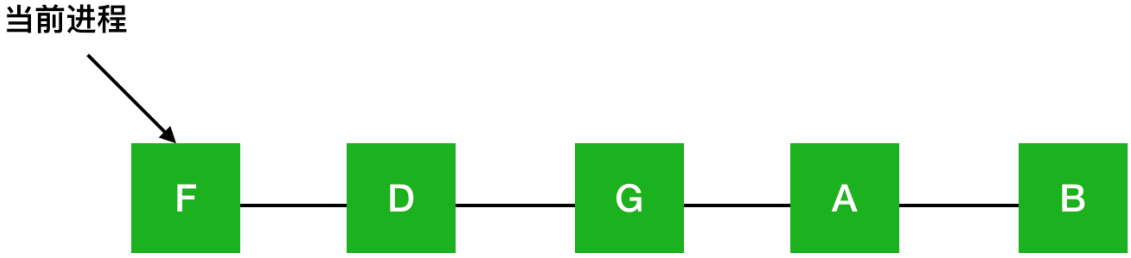
轮询调度

一种最古老、最简单、最公平并且最广泛使用的算法就是 `轮询算法(round-robin)` 。每个进程都会被分配一个时间段，称为 `时间片(quantum)` ，在这个时间片内允许进程运行。如果时间片结束时进程还在运行的话，则抢占一个 CPU 并将其分配给另一个进程。如果进程在时间片结束前阻塞或结束，则 CPU 立即进行切换。轮询算法比较容易实现。调度程序所做的就是维护一个可运行进程的列表，就像下图中的 a，当一个进程用完时间片后就被移到队列的末尾，就像下图的 b。



a

可运行进程列表



b

进程 B 用完时间片后的可运行列表

优先级调度

事实情况是不是所有的进程都是优先级相等的。例如，在一所大学中的等级制度，首先是院长，然后是教授、秘书、后勤人员，最后是学生。这种将外部情况考虑在内就实现了 `优先级调度(priority scheduling)`



它的基本思想很明确，每个进程都被赋予一个优先级，优先级高的进程优先运行。

但是也不意味着高优先级的进程能够永远一直运行下去，调度程序会在每个时钟中断期间降低当前运行进程的优先级。如果此操作导致其优先级降低到下一个最高进程的优先级以下，则会发生进程切换。或者，可以为每个进程分配允许运行的最大时间间隔。当时间间隔用完后，下一个高优先级的进程会得到运行的机会。

最短进程优先

对于批处理系统而言，由于最短作业优先常常伴随着最短响应时间，一种方式是根据进程过去的行为进行推测，并执行估计运行时间最短的那一个。假设每个终端上每条命令的预估运行时间为 `T0`，现在假设测量到其下一次运行时间为 `T1`，可以用两个值的加权来改进估计时间，即 `aT0+ (1- 1)T1`。通过选择 a 的值，可以决定是尽快忘掉老的运行时间，还是在一段长时间内始终记住它们。当 a = 1/2 时，可以得到下面这个序列

$$T_0, \quad \frac{T_0}{2}+\frac{T_1}{2}, \quad \frac{T_0}{4}+\frac{T_1}{4}+\frac{T_2}{2}, \quad \frac{T_0}{8}+\frac{T_1}{8}+\frac{T_2}{4}+\frac{T_3}{2}$$

可以看到，在三轮过后，T0 在新的估计值中所占比重下降至 1/8。

有时把这种通过当前测量值和先前估计值进行加权平均从而得到下一个估计值的技术称作 `老化(aging)`。这种方法会使用很多预测值基于当前值的情况。

彩票调度

有一种既可以给出预测结果而又有一种比较简单的实现方式的算法，就是 `彩票调度(lottery scheduling)` 算法。他的基本思想为进程提供各种系统资源的 `彩票`。当做出一个调度决策的时候，就随机抽出一张彩票，拥有彩票的进程将获得资源。比如在 CPU 进行调度时，系统可以每秒持有 50 次抽奖，每个中奖进程会获得额外运行时间的奖励。

> 可以把彩票理解为 buff，这个 buff 有 15% 的几率能让你产生 `速度之靴` 的效果。

公平分享调度

如果用户 1 启动了 9 个进程，而用户 2 启动了一个进程，使用轮转或相同优先级调度算法，那么用户 1 将得到 90 % 的 CPU 时间，而用户 2 将之得到 10 % 的 CPU 时间。

为了阻止这种情况的出现，一些系统在调度前会把进程的拥有者考虑在内。在这种模型下，每个用户都会分配一些CPU 时间，而调度程序会选择进程并强制执行。因此如果两个用户每个都会有 50% 的 CPU 时间片保证，那么无论一个用户有多少个进程，都将获得相同的 CPU 份额。

页面置换算法都有哪些

算法	注释
最优算法	不可实现，但可以用作基准
NRU(最近未使用) 算法	和 LRU 算法很相似
FIFO(先进先出) 算法	有可能会抛弃重要的页面
第二次机会算法	比 FIFO 有较大的改善
时钟算法	实际使用
LRU(最近最少)算法	比较优秀，但是很难实现
NFU(最不经常食用)算法	和 LRU 很类似
老化算法	近似 LRU 的高效算法
工作集算法	实施起来开销很大
工作集时钟算法	比较有效的算法

- `最优算法` 在当前页面中置换最后要访问的页面。不幸的是，没有办法来判定哪个页面是最后一个要访问的，`因此实际上该算法不能使用`。然而，它可以作为衡量其他算法的标准。
- `NRU` 算法根据 R 位和 M 位的状态将页面氛围四类。从编号最小的类别中随机选择一个页面。NRU 算法易于实现，但是性能不是很好。存在更好的算法。
- `FIFO` 会跟踪页面加载进入内存中的顺序，并把页面放入一个链表中。有可能删除存在时间最长但是还在使用的页面，因此这个算法也不是一个很好的选择。
- `第二次机会` 算法是对 FIFO 的一个修改，它会在删除页面之前检查这个页面是否仍在使用。如果页面正在使用，就会进行保留。这个改进大大提高了性能。
- `时钟` 算法是第二次机会算法的另外一种实现形式，时钟算法和第二次算法的性能差不多，但是会花费更少的时间来执行算法。
- `LRU` 算法是一个非常优秀的算法，但是没有 `特殊的硬件(TLB)` 很难实现。如果没有硬件，就不能使用 LRU 算法。
- `NFU` 算法是一种近似于 LRU 的算法，它的性能不是非常好。
- `老化` 算法是一种更接近 LRU 算法的实现，并且可以更好的实现，因此是一个很好的选择
- 最后两种算法都使用了工作集算法。工作集算法提供了合理的性能开销，但是它的实现比较复杂。`WSClock` 是另外一种变体，它不仅能够提供良好的性能，而且可以高效地实现。

最好的算法是老化算法和WSClock算法。他们分别是基于 LRU 和工作集算法。他们都具有良好的性能并且能够被有效的实现。还存在其他一些好的算法，但实际上这两个可能是最重要的。

影响调度程序的指标是什么

会有下面几个因素决定调度程序的好坏

- CPU 使用率：

CPU 正在执行任务（即不处于空闲状态）的时间百分比。

- 等待时间

这是进程轮流执行的时间，也就是进程切换的时间

- 吞吐量

单位时间内完成进程的数量

- 响应时间

这是从提交流程到获得有用输出所经过的时间。

- 周转时间

从提交流程到完成流程所经过的时间。

什么是僵尸进程

僵尸进程是已完成且处于终止状态，但在进程表中却仍然存在的进程。僵尸进程通常发生在父子关系的进程中，由于父进程仍需要读取其子进程的退出状态所造成的。

关于操作系统，你必须知道的名词

1. `操作系统（Operating System，OS）`：是管理计算机硬件与软件资源的系统`软件`，同时也是计算机系统的`内核与基石`。操作系统需要处理**管理与配置内存、决定系统资源供需的优先次序、控制输入与输出设备、操作网络与管理文件系统等基本事务**。操作系统也提供一个让用户与系统交互的操作界面。

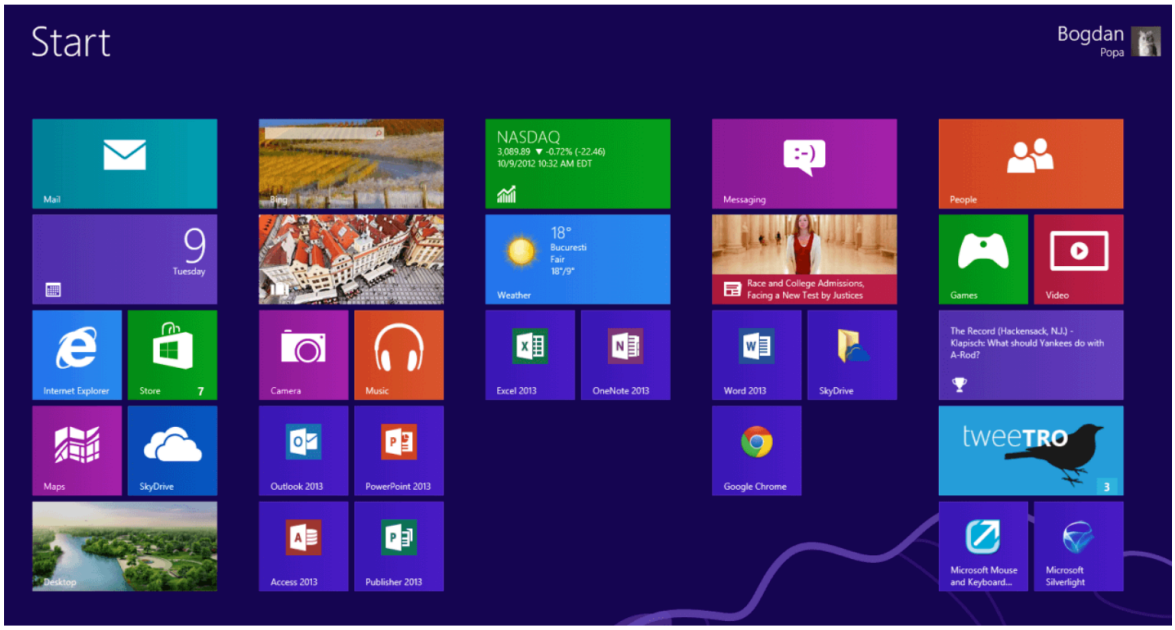


2. `shell`：它是一个程序，可从键盘获取命令并将其提供给操作系统以执行。 在过去，它是类似 Unix 的系统上唯一可用的用户界面。 如今，除了命令行界面（CLI）外，我们还具有图形用户界面（··）。

```
#!/bin/sh
#echoes its shell script file name and the values of odd arguments

echo $0
while [ $# -gt 0 ]
do
 result=`expr $# % 2`
 if [ result -eq 0 ]
 then
  shift
 else
  echo \$$#
  shift
 fi
done
~
~
~
~
~
~
~
~
```

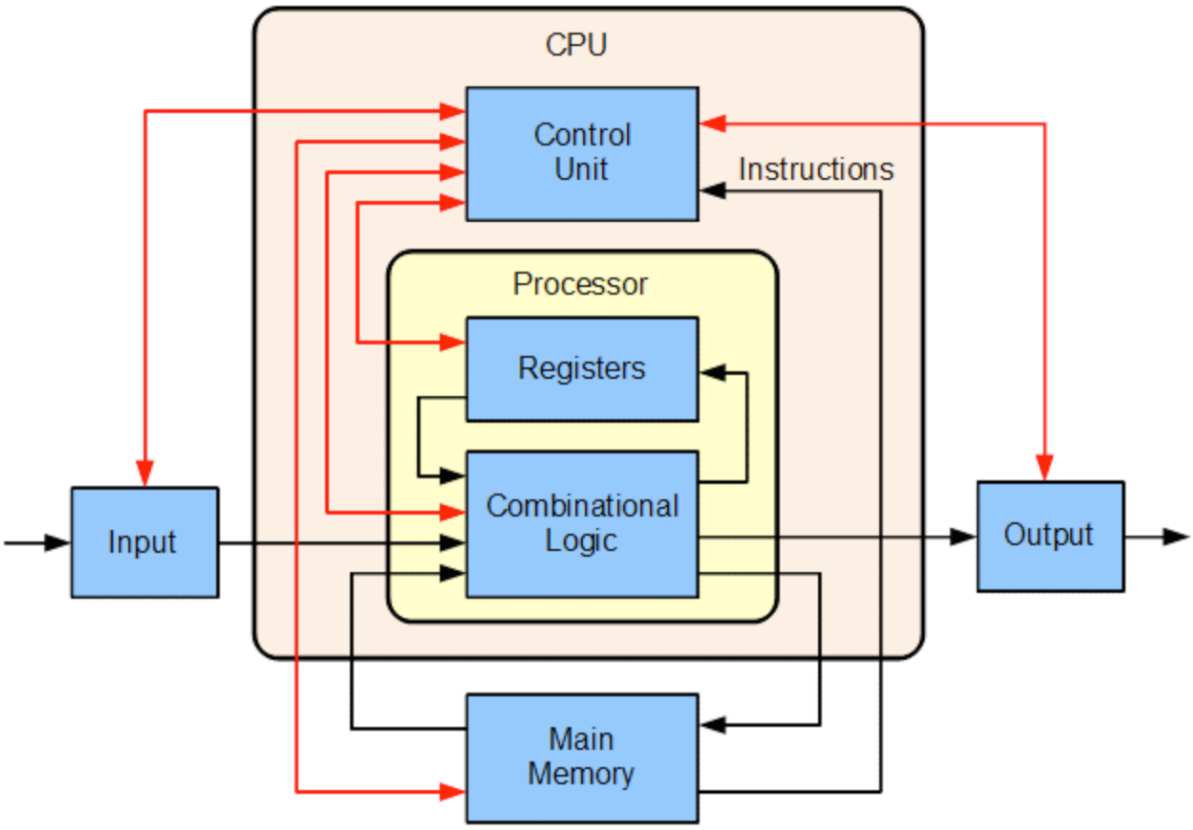
3. `GUI (Graphical User Interface)`：是一种`用户界面`，允许用户通过图形图标和音频指示符与电子设备进行交互。



4. `内核模式(kernel mode)`：通常也被称为 `超级模式（supervisor mode）`，在内核模式下，正在执行的代码具有对底层硬件的完整且不受限制的访问。 它可以执行任何 CPU 指令并引用任何内存地址。 内核模式通常保留给操作系统的最低级别，最受信任的功能。 内核模式下的崩溃是灾难性的； 他们将停止整个计算机。 超级用户模式是计算机开机时选择的自动模式。

5. `用户模式(user node)`：当操作系统运行用户应用程序（例如处理文本编辑器）时，系统处于用户模式。 当应用程序请求操作系统的帮助或发生中断或系统调用时，就会发生从用户模式到内核模式的转换。在用户模式下，模式位设置为1。 从用户模式切换到内核模式时，它从1更改为0。

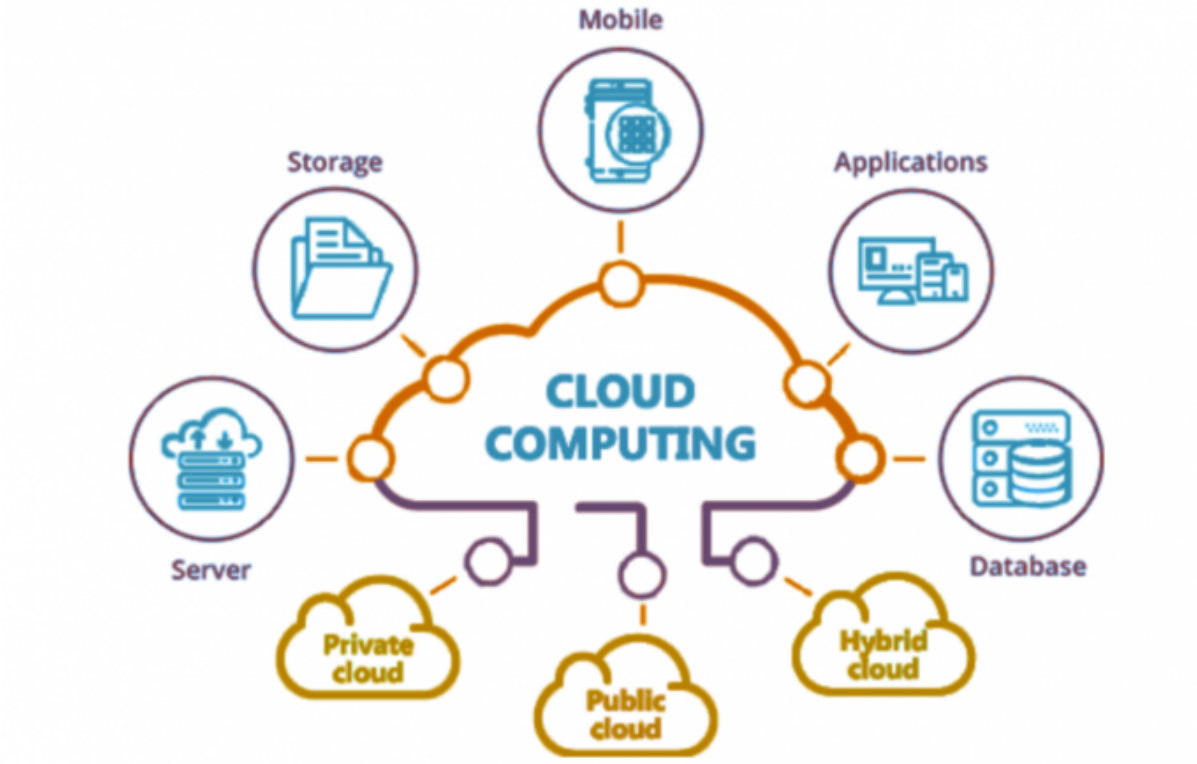
6. `计算机架构(computer architecture)`： 在计算机工程中，计算机体系结构是描述计算机系统功能，组织和实现的一组规则和方法。它主要包括指令集、内存管理、I/O 和总线结构



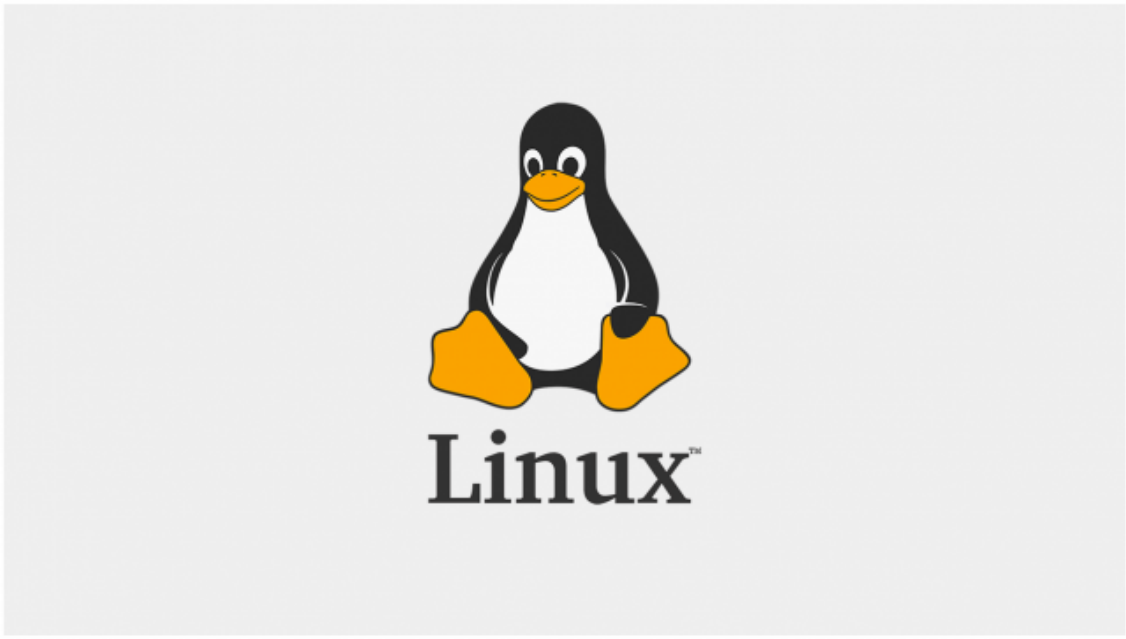
7. `SATA(Serial ATA)`：串行 ATA (Serial Advanced Technology Attachment)，它是一种电脑总线，负责主板和大容量存储设备（如硬盘及光盘驱动器）之间的数据传输，主要用于个人电脑。
8. `复用(multiplexing)`：也称为共享，在操作系统中主要指示了时间和空间的管理。对资源进行复用时，不同的程序或用户轮流使用它。 他们中的第一个开始使用资源，然后再使用另一个，依此类推。
9. `大型机(mainframes)`：大型机是一类计算机，通常以其大尺寸，存储量，处理能力和高度的可靠性而著称。它们主要由大型组织用于需要大量数据处理的关键任务应用程序。



10. `批处理(batch system)`：批处理操作系统的用户不直接与计算机进行交互。 每个用户都在打孔卡等脱机设备上准备工作，并将其提交给计算机操作员。 为了加快处理速度，将具有类似需求的作业一起批处理并成组运行。 程序员将程序留给操作员，然后操作员将具有类似要求的程序分批处理。
11. `OS/360`： OS/360，正式称为IBM System / 360操作系统，是由 IBM 为 1964 年发布的其当时新的System/360 大型机开发的已停产的批处理操作系统。
12. `多处理系统(Computer multitasking)`：是指计算机同时运行多个程序的能力。多任务的一般方法是运行第一个程序的一段代码，保存工作环境；再运行第二个程序的一段代码，保存环境；……恢复第一个程序的工作环境，执行第一个程序的下一段代码。
13. `分时系统(Time-sharing)`：在计算中，分时是通过多程序和多任务同时在许多用户之间共享计算资源的一种系统
14. `相容分时系统(Compatible Time-Sharing System)`：最早的分时操作系统，由美国麻省理工学院计算机中心设计与实作。
15. `云计算(cloud computing)`：云计算是计算机系统资源（尤其是数据存储和计算能力）的按需可用性，而无需用户直接进行主动管理。这个术语通常用于描述 Internet 上可供许多用户使用的数据中心。 如今占主导地位的大型云通常具有从中央服务器分布在多个位置的功能。 如果与用户的连接相对较近，则可以将其指定为边缘服务器。

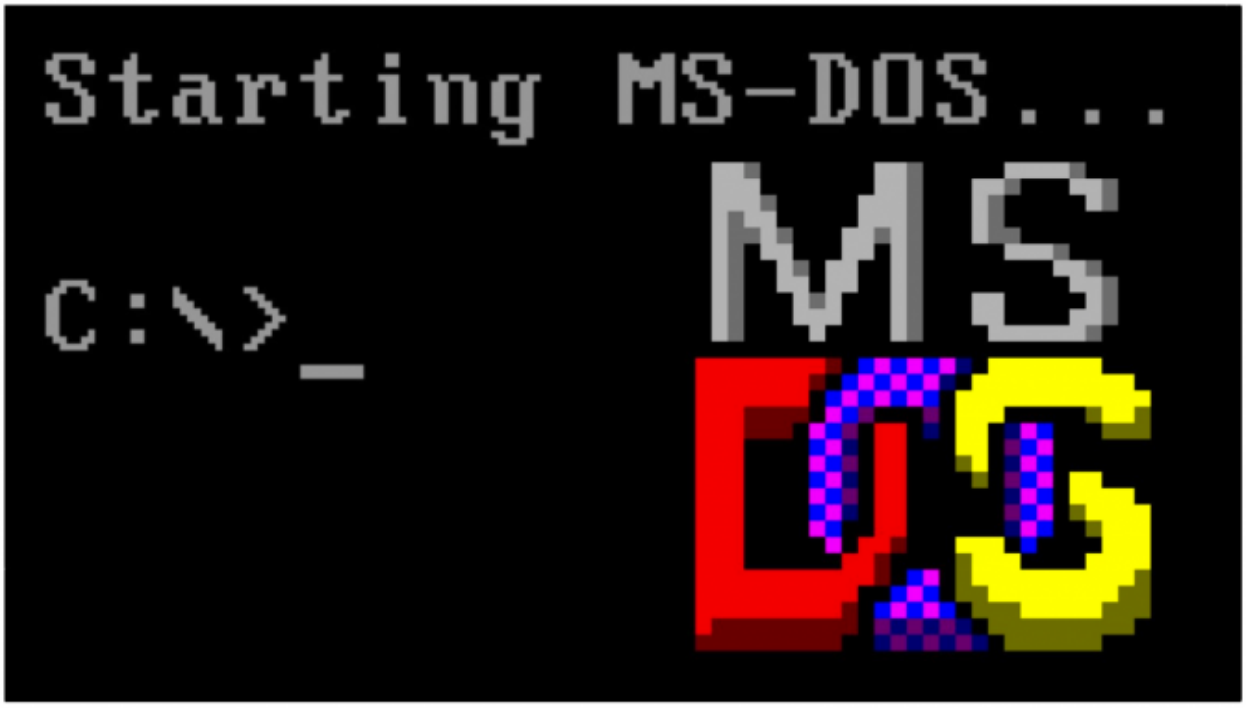


16. `UNIX 操作系统`：UNIX 操作系统，是一个强大的多用户、多任务操作系统，支持多种处理器架构，按照操作系统的分类，属于分时操作系统。
17. `UNIX System V`：是 UNIX 操作系统的一个分支。
18. `BSD(Berkeley Software Distribution)`：UNIX 的衍生系统。
19. `POSIX`：可移植操作系统接口，是 IEEE 为要在各种 UNIX 操作系统上运行软件，而定义API的一系列互相关联的标准的总称。
20. `MINIX`：Minix，是一个迷你版本的类 UNIX 操作系统。
21. `Linux`：终于到了大名鼎鼎的 Linux 操作系统了，太强大了，不予以解释了，大家都懂。



22. `DOS (Disk Operating System)`：磁盘操作系统（缩写为DOS）是可以使用磁盘存储设备（例如软盘，硬盘驱动器或光盘）的计算机操作系统。

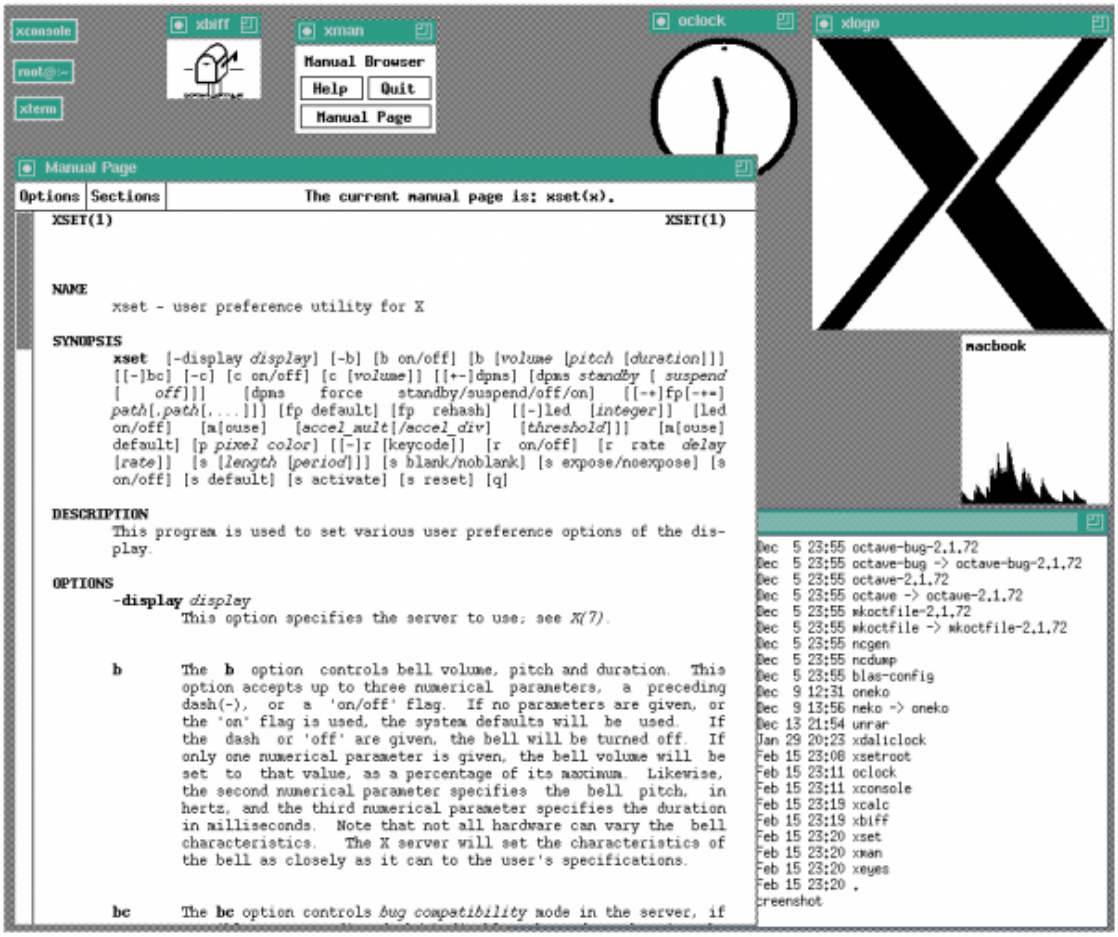
23. `MS-DOS(MicroSoft Disk Operating System)` ：一个由美国微软公司发展的操作系统，运行在Intel x86个人电脑上。它是DOS操作系统家族中最著名的一个，在Windows 95以前，DOS是IBM PC及兼容机中的最基本配备，而MS-DOS则是个人电脑中最普遍使用的DOS操作系统。



24. `MacOS X`，怎能少的了苹果操作系统？macOS 是苹果公司推出的基于图形用户界面操作系统，为 Macintosh 的主操作系统



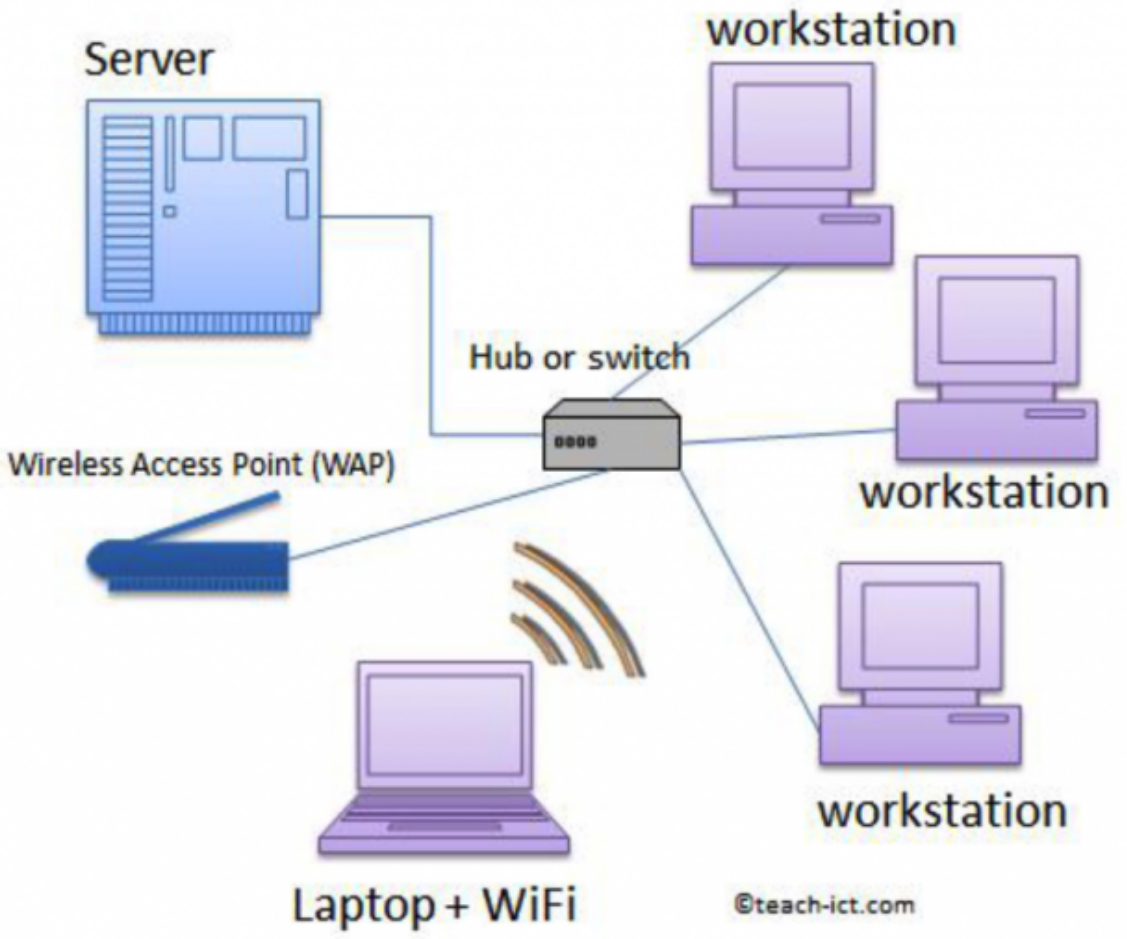
25. `Windows NT(Windows New Technology)`：是美国微软公司 1993 年推出的纯 32 位操作系统核心。
26. `Service Pack(SP)`：是程序的更新、修复和（或）增强的集合，以一个独立的安装包的形式发布。许多公司，如微软或Autodesk，通常在为某一程序而做的修补程序达到一定数量时，就发布一个Service Pack。
27. `数字版权管理（DRM）`：他是工具或技术保护措施（TPM）是一组访问控制技术，用于限制对专有硬件和受版权保护的作品的使用。
28. `x86`：x86是一整套指令集体系结构，由 Intel 最初基于 Intel 8086 微处理器及其 8088 变体开发。采用内存分段作为解决方案，用于处理比普通 16 位地址可以覆盖的更多内存。32 位是 x86 默认的位数，除此之外，还有一个 x86-64 位，是x86架构的 64 位拓展，向后兼容于 16 位及 32 位的 x86架构。
29. `FreeBSD`：FreeBSD 是一个类 UNIX 的操作系统，也是 FreeBSD 项目的发展成果。
30. `X Window System`：X 窗口系统（X11，或简称X）是用于位图显示的窗口系统，在类 UNIX 操作系统上很常见。



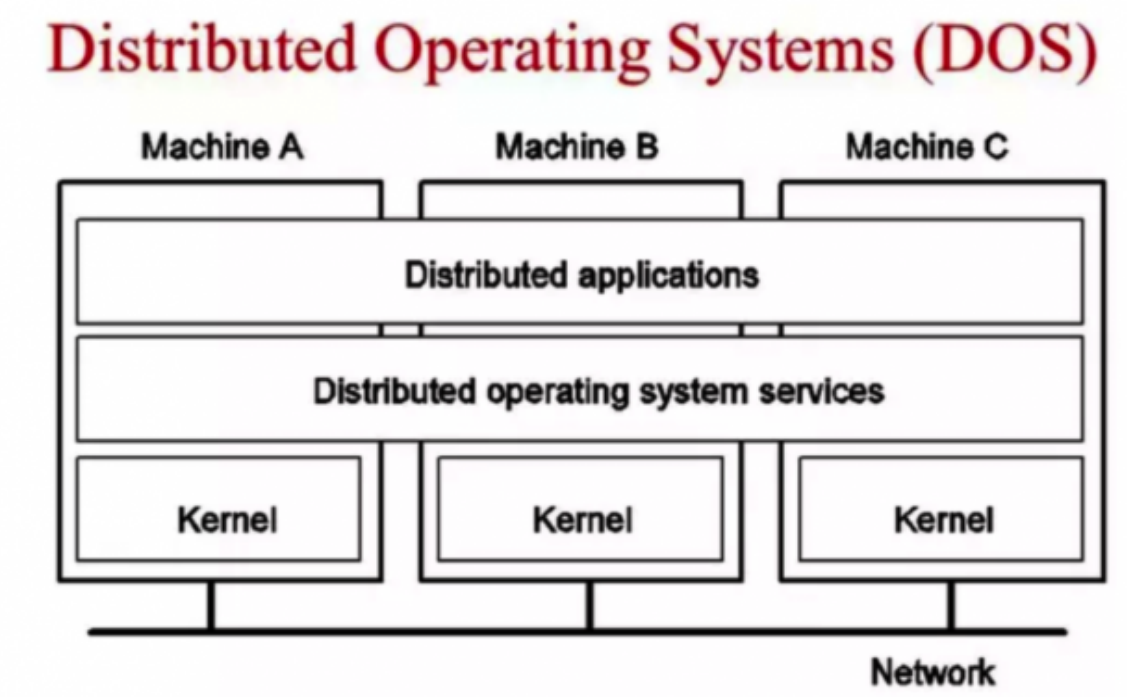
31. `Gnome`：GNOME 是一个完全由自由软件组成的桌面环境。它的目标操作系统是Linux，但是大部分的 BSD 系统亦支持 GNOME。



32. `网络操作系统(network operating systems)`：网络操作系统是用于网络设备（如路由器，交换机或防火墙）的专用操作系统。



33. `分布式网络系统(distributed operating systems)`：分布式操作系统是在独立，网络，通信和物理上独立计算节点的集合上的软件。 它们处理由多个CPU服务的作业。每个单独的节点都拥有全局集合操作系统的特定软件的一部分。



34. `程序计数器(Program counter)`：程序计数器 是一个 CPU 中的 `寄存器`，用于指示计算机在其程序序列中的 `位置` 。

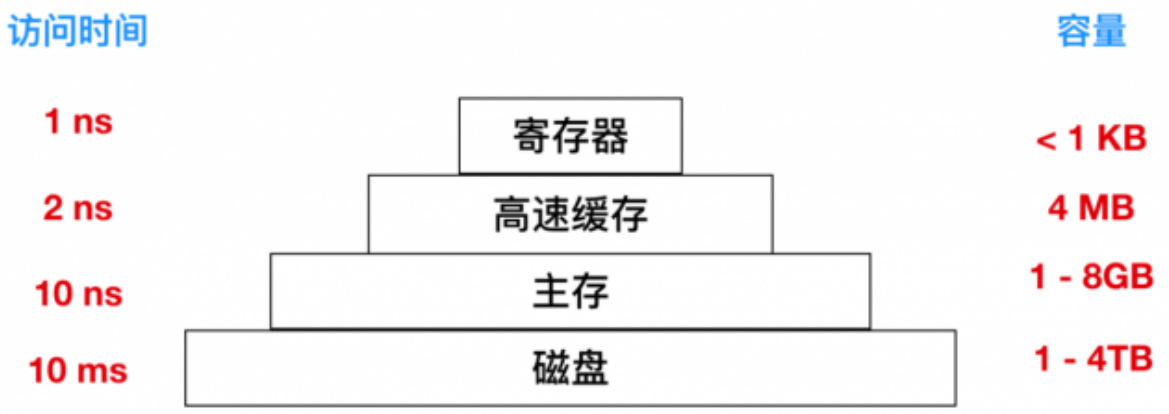
35. `堆栈寄存器(stack pointer)`： 堆栈寄存器是计算机 CPU 中的寄存器，其目的是`跟踪调用堆栈`。
36. `程序状态字(Program Status Word)`：它是由操作系统维护的8个字节（或64位）长的数据的集合。它跟踪系统的当前状态。
37. `流水线(Pipeline)`：在计算世界中，管道是一组串联连接的数据处理元素，其中一个元素的输出是下一个元素的输入。 流水线的元素通常以并行或按时间分割的方式执行。 通常在元素之间插入一定数量的缓冲区存储。



38. `超标量(superscalar)`： 超标量 CPU 架构是指在一颗处理器内核中实行了指令级并发的一类并发运算。这种技术能够在相同的CPU主频下实现更高的 CPU 流量。
39. `系统调用(system call)`：指运行在用户空间的程序向操作系统内核请求需要更高权限运行的服务。系统调用提供用户程序与操作系统之间的接口。大多数系统交互式操作需求在内核态运行。如设备 IO 操作或者进程间通信。
40. `多线程(multithreading)`： 是指从软件或者硬件上实现多个线程并发执行的技术。具有多线程能力的计算机因为有硬件支持而能够在同一时间执行多个线程，进而提升整体处理性能。
41. `CPU 核心(core)`： 它是 CPU 的大脑，它接收指令，并执行计算或运算以满足这些指令。一个 CPU 可以有多个内核。
42. `图形处理器(Graphics Processing Unit)`： 又称显示核心、视觉处理器、显示芯片或绘图芯片；它是一种专门在个人电脑、工作站、游戏机和一些移动设备（如平板电脑、智能手机等）上运行绘图运算工作的微处理器。

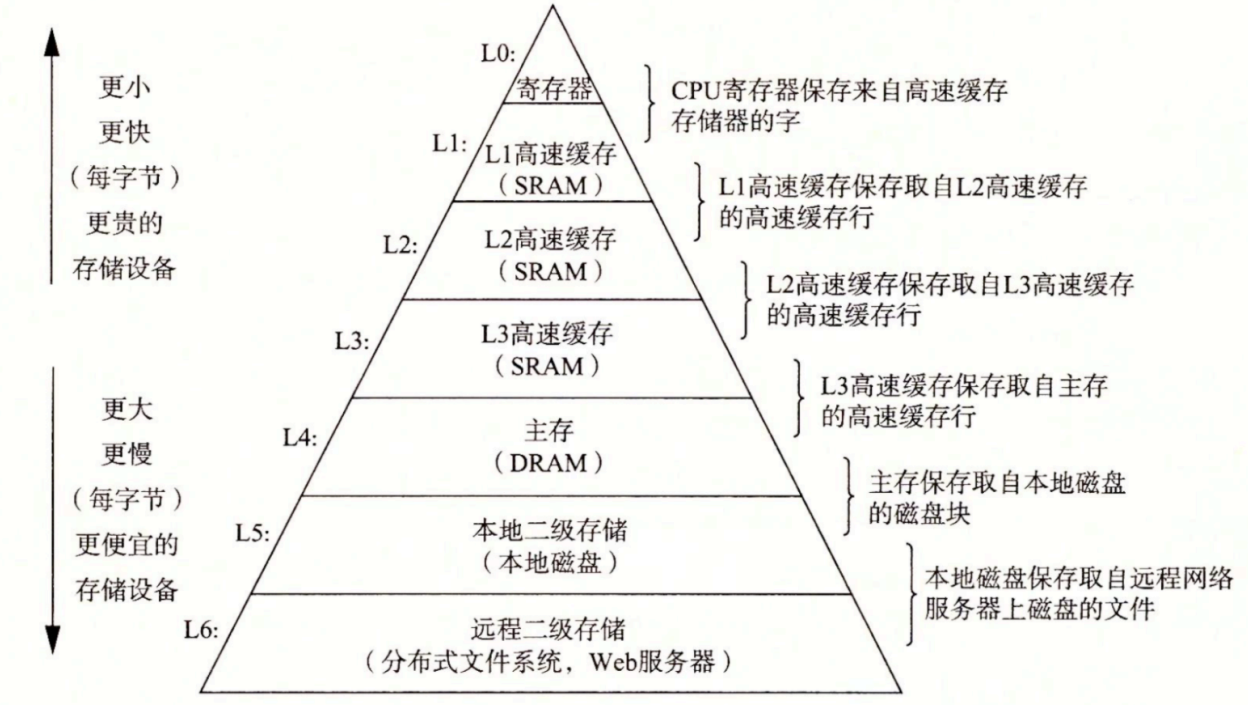


43. 存储体系结构：顶层的存储器速度最高，但是容量最小，成本非常高，层级结构越向下，其访问效率越慢，容量越大，但是造价也就越便宜。



存储层次结构

44. 高速缓存行(cache lines)：其实就是把高速缓存分割成了固定大小的块，其大小是以突发读或者突发写周期的大小为基础的。
45. 缓存命中(cache hit)：当应用程序或软件请求数据时，会首先发生缓存命中。 首先，中央处理单元（CPU）在其最近的内存位置（通常是主缓存）中查找数据。 如果在缓存中找到请求的数据，则将其视为缓存命中。



一个存储器层次结构的示例

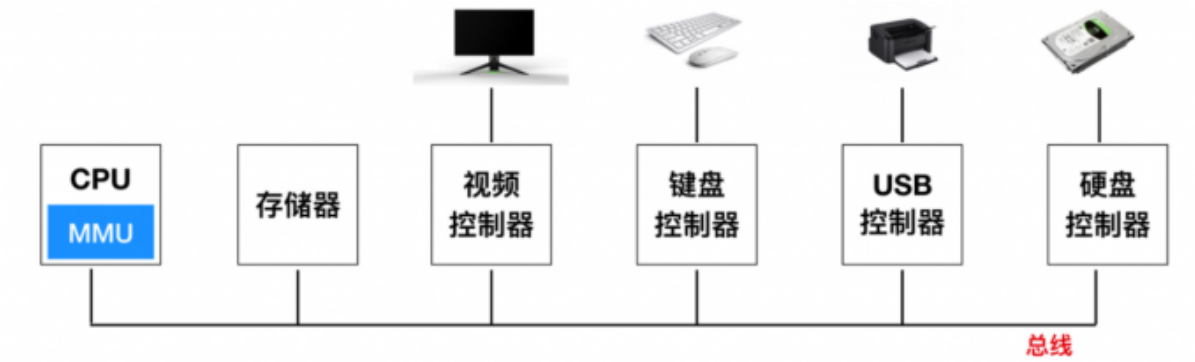
46. L1 cache：一级缓存是 CPU 芯片中内置的存储库。 L1缓存也称为 主缓存，是计算机中 最快 的内存，并且最接近处理器。
47. L2 cache：二级缓存存储库，内置在 CPU 芯片中，包装在同一模块中，或者建在主板上。 L2 高速缓存提供给 L1 高速缓存，后者提供给处理器。 L2 内存比 L1 内存慢。
48. L2 cache：三级缓存内置在主板上或CPU模块内的存储库。 L3 高速缓存为 L2 高速缓存提供数据，其内存通常比 L2 内存慢，但比主内存快。 L3 高速缓存提供给 L2 高速缓存，后者又提供给 L1 高速缓存，后者又提供给处理器。
49. RAM(Random Access Memory)：随机存取存储器，也叫主存，是与 CPU 直接交换数据 的内部存储器。它可以随时读写，而且速度很快，通常作为操作系统或其他正在运行中的程序的 临时

数据存储介质。RAM工作时可以随时从任何一个指定的地址写入（存入）或读出（取出）信息。它与 ROM 的最大区别是数据的`易失性`，即一旦断电所存储的数据将随之丢失。RAM 在计算机和数字系统中用来暂时存储程序、数据和中间结果。

50. `ROM (Read Only Memory)`：只读存储器是一种半导体存储器，其特性是**一旦存储数据就无法改变或删除**，且内容不会因为电源关闭而`消失`。在电子或电脑系统中，通常用以存储不需经常变更的程序或数据。
51. `EEPROM (Electrically Erasable PROM)`：电可擦除可编程只读存储器，是一种可以通过电子方式多次复写的半导体存储设备。
52. `闪存(flash memory)`： 是一种电子式可清除程序化只读存储器的形式，允许在操作中被多次擦或写的存储器。这种科技主要用于一般性数据存储，以及在电脑与其他数字产品间交换传输数据，如储存卡与U盘。
53. `SSD(Solid State Disks)`：固态硬盘，是一种主要以闪存作为永久性存储器的电脑存储设备。

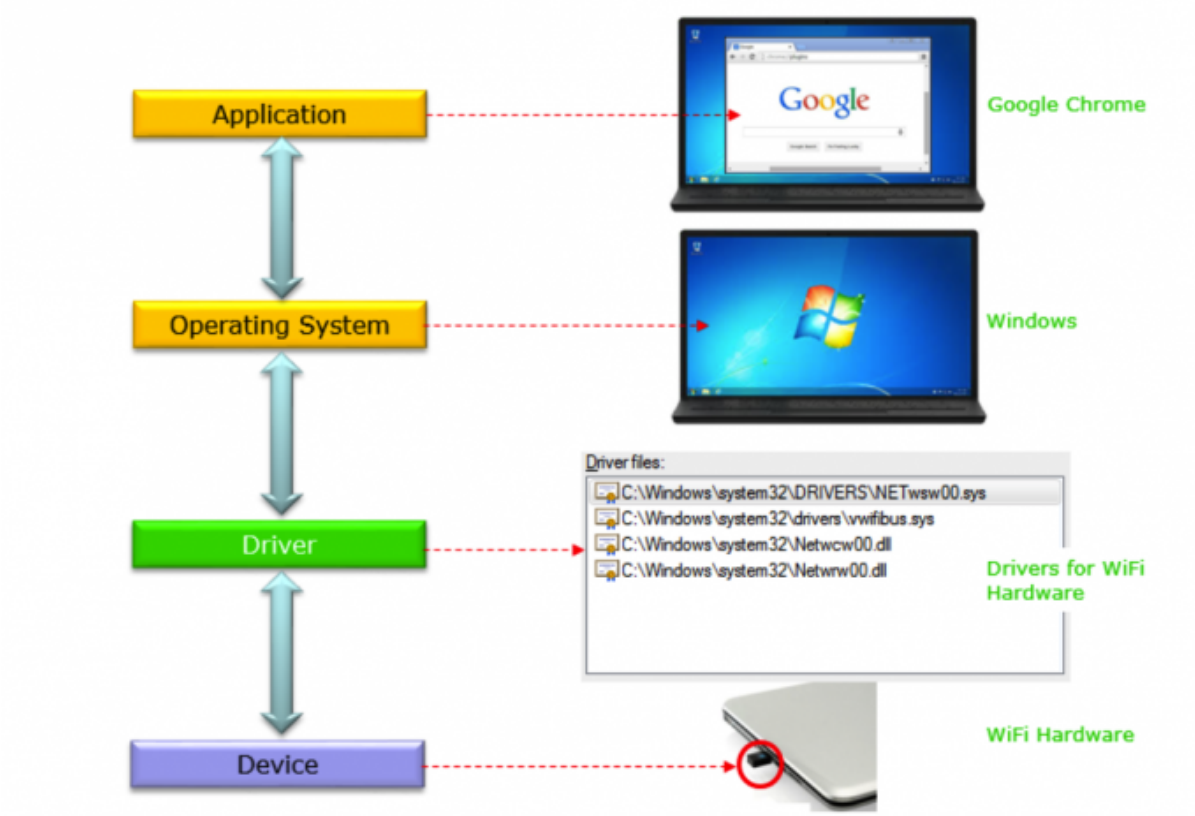


54. `虚拟地址(virtual memory)`： 虚拟内存是计算机系统`内存管理`的一种机制。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。与没有使用虚拟内存技术的系统相比，使用这种技术的系统使得大型程序的编写变得更容易，对真正的物理内存（例如 RAM）的使用也更有效率。
55. `MMU (Memory Management Unit)`：内存管理单元，有时称作分页内存管理单元。它是一种负责处理中央处理器（CPU）的内存访问请求的计算机硬件。它的功能包括**虚拟地址到物理地址的转换（即虚拟内存管理）、内存保护、中央处理器高速缓存的控制等**。



56. `context switch`：上下文切换，又称环境切换。是一个存储和重建 CPU 状态的机制。要交换 CPU 上的进程时，必需先行存储当前进程的状态，然后再将进程状态读回 CPU 中。

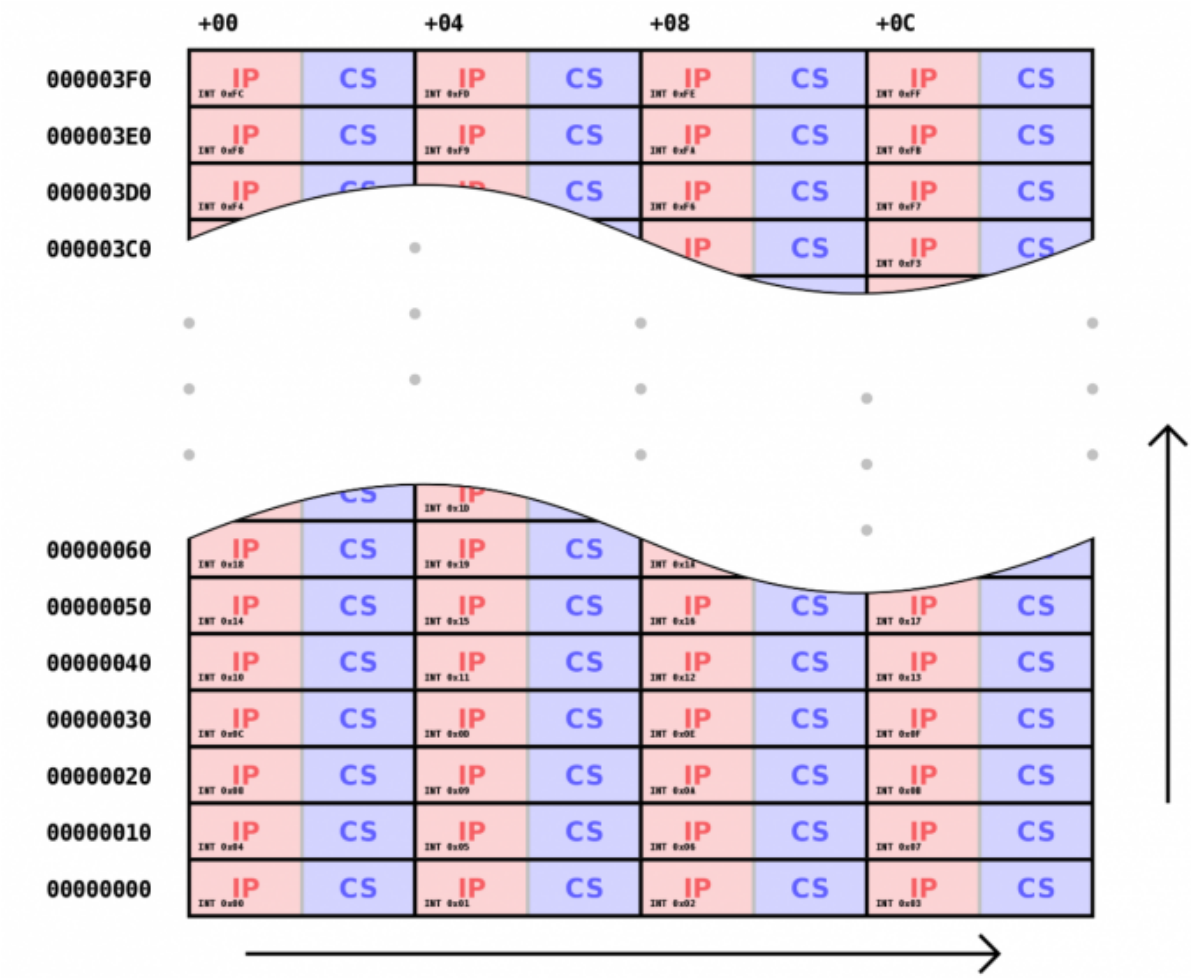
57. `驱动程序(device driver)`：设备驱动程序，简称驱动程序（driver），是一个允许高级别电脑软件与硬件交互的程序，这种程序创建了一个硬件与硬件，或硬件与软件沟通的接口，经由主板上的总线或其它沟通子系统与硬件形成连接的机制，这样使得硬件设备上的数据交换成为可能。



58. `忙等(busy waiting)`：在软件工程中，忙碌等待`也称自旋`，是一种以进程反复检查一个条件是否为真的条件，这种机制可能为检查键盘输入或某个锁是否可用。

59. `中断(Interrupt)`：通常，在接收到来自外围硬件（相对于中央处理器和内存）的异步信号，或来自软件的同步信号之后，处理器将会进行相应的硬件 / 软件处理。发出这样的信号称为进行`中断请求（interrupt request，IRQ）`。硬件中断导致处理器通过一个`运行信息切换（context switch）`来保存执行状态（以程序计数器和程序状态字等寄存器信息为主）；`软件中断则`通常作为 CPU 指令集中的一个指令，以可编程的方式直接指示这种运行信息切换，并将处理导向一段中断处理代码。中断在计算机多任务处理，尤其是即时系统中尤为有用。

60. `中断向量(interrupt vector)`：中断向量位于中断向量表中。`中断向量表（IVT）`是将中断处理程序列表与中断向量表中的中断请求列表相关联的数据结构。 中断向量表的每个条目（称为中断向量）都是中断处理程序的地址。



61. `DMA (Direct Memory Access)`：直接内存访问，直接内存访问是计算机科学中的一种内存访问技术。它允许某些电脑内部的硬件子系统（电脑外设），可以独立地直接读写系统内存，而不需中央处理器（CPU）介入处理 。

62. `总线(Bus)`：总线（Bus）是指计算机组件间规范化的交换数据的方式，即以一种通用的方式为各组件提供数据传送和控制逻辑。

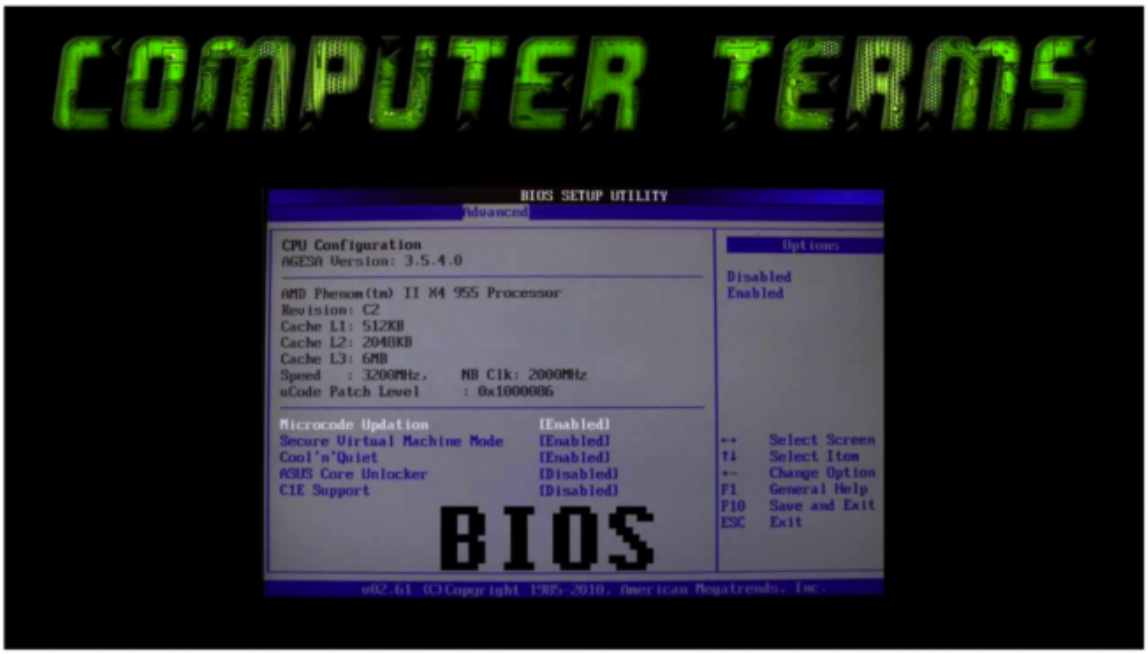
63. `PCIe (Peripheral Component Interconnect Express)`：官方简称PCIe，是计算机总线的一个重要分支，它沿用现有的PCI编程概念及信号标准，并且构建了更加高速的串行通信系统标准。

64. `DMI (Direct Media Interface)`：直接媒体接口，是英特尔专用的总线，用于电脑主板上南桥芯片和北桥芯片之间的连接。

65. `USB(Universal Serial Bus)`：是连接计算机系统与外部设备的一种 `串口总线` 标准，也是一种输入输出接口的技术规范，被广泛地应用于个人电脑和移动设备等信息通讯产品，并扩展至摄影器材、数字电视（机顶盒）、游戏机等其它相关领域。



66. `BIOS(Basic Input Output System)`：是在通电引导阶段运行硬件初始化，以及为操作系统提供运行时服务的固件。它是开机时运行的第一个软件。



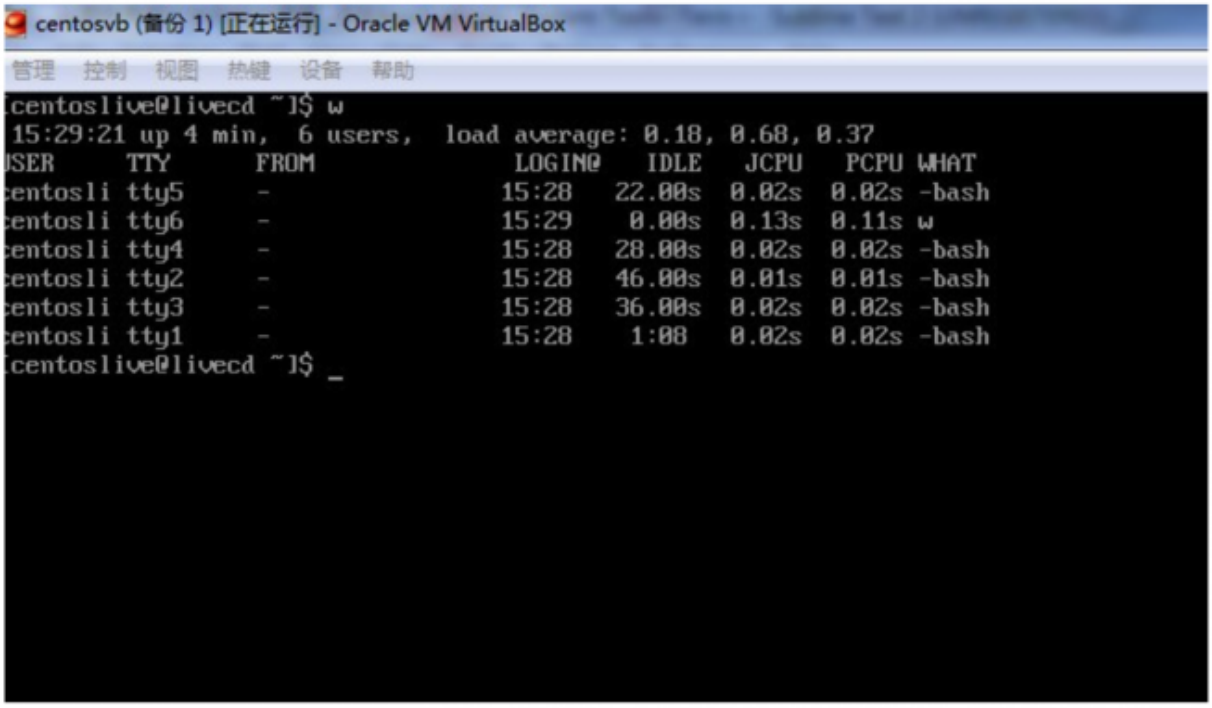
67. `硬实时系统(hard real-time system)`：硬实时性意味着你必须绝对在每个截止日期前完成任务。 很少有系统有此要求。 例如核系统，一些医疗应用（例如起搏器），大量国防应用，航空电子设备等。

68. `软实时系统(soft real-time system)`：软实时系统可能会错过某些截止日期，但是如果错过太多，最终性能将下降。 一个很好的例子是计算机中的声音系统。

69. `进程(Process)`：程序本身只是指令、数据及其组织形式的描述，进程才是程序（那些指令和数据）的真正运行实例。若进程有可能与同一个程序相关系，且每个进程皆可以同步（循序）或异步的方式独立运行。

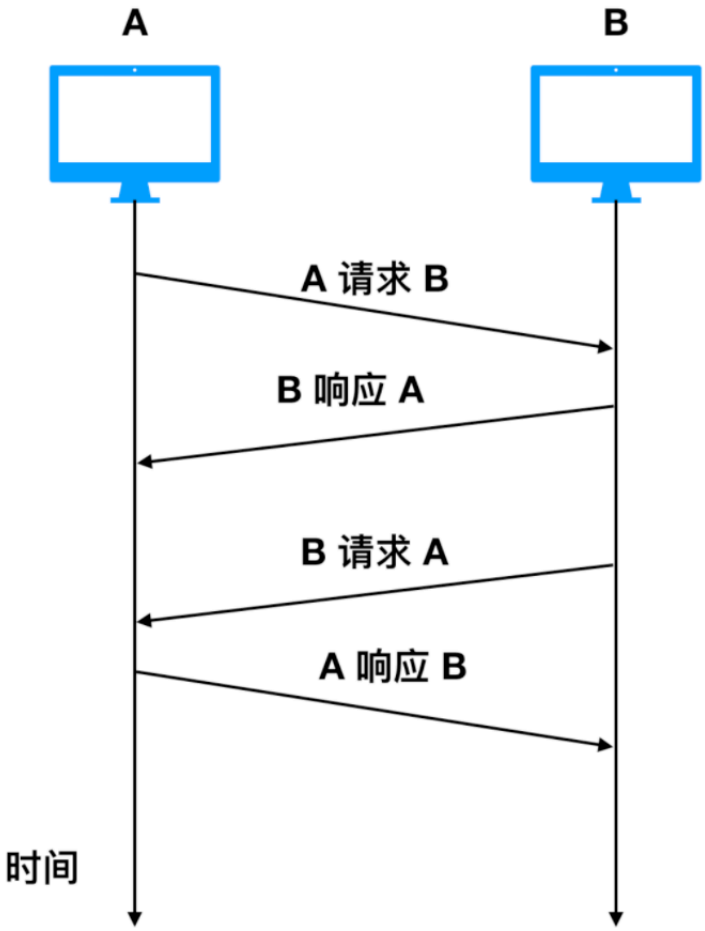
70. `地址空间(address space)`：地址空间是内存中可供程序或进程使用的有效地址范围。 也就是说，它是程序或进程可以访问的内存。 存储器可以是物理的也可以是虚拟的，用于执行指令和存储数据。

71. 进程表(process table)：进程表是操作系统维护的数据结构，该表中的每个条目（通常称为上下文块）均包含有关进程的信息，例如进程名称和状态，优先级，寄存器以及它可能正在等待的信号灯。
72. 命令行界面(command-line interpreter)：是在图形用户界面得到普及之前使用最为广泛的用户界面，它通常不支持鼠标，用户通过键盘输入指令，计算机接收到指令后，予以执行。

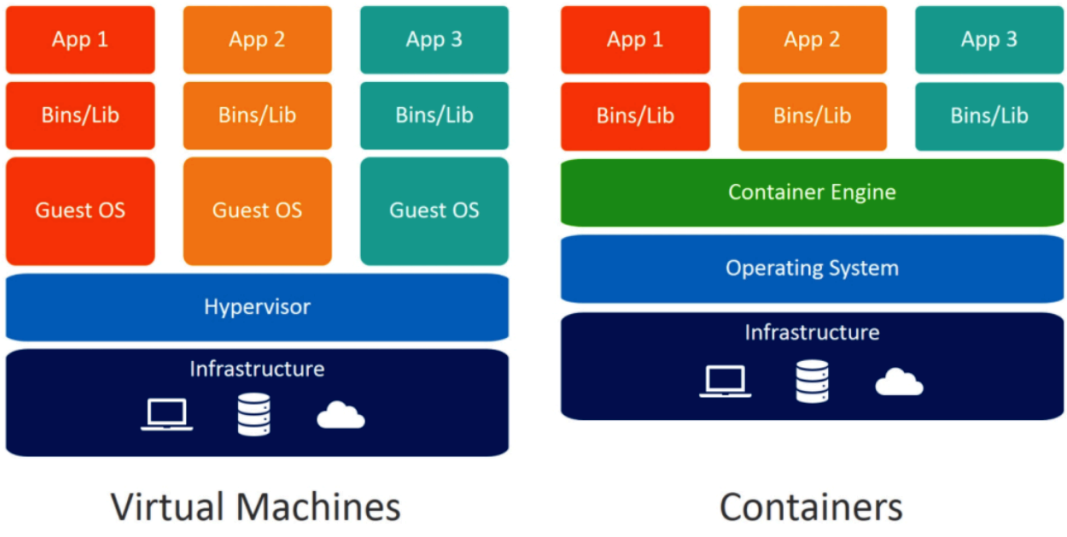


73. 进程间通信(interprocess communication)： 指至少两个进程或线程间传送数据或信号的一些技术或方法。
74. 超级用户(superuser)： 也被称为管理员帐户，在计算机操作系统领域中指一种用于进行系统管理的特殊用户，其在系统中的实际名称也因系统而异，如 root、administrator 与supervisor。
75. 目录(directory)：在计算机或相关设备中，一个目录或文件夹就是一个装有数字文件系统的虚拟 容器 。在它里面保存着一组文件和其它一些目录。
76. 路径(path name)： 路径是一种电脑文件或目录的名称的通用表现形式，它指向文件系统上的一个唯一位置。
77. 根目录(root directory)：根目录指的就是计算机系统中的顶层目录，比如 Windows 中的 C 盘和 D 盘，Linux 中的 / 。
78. 工作目录(Working directory)：它是一个计算机用语。用户在操作系统内所在的目录，用户可在此目录之下，用相对文件名访问文件。
79. 文件描述符(file descriptor)： 文件描述符是计算机科学中的一个术语，是一个用于表述指向文件的引用的抽象化概念。
80. inode：索引节点的缩写，索引节点是 UNIX 系统中包含的信息，其中包含有关每个文件的详细信息，例如节点，所有者，文件，文件位置等。
81. 共享库(shared library)：共享库是一个包含目标代码的文件，执行过程中多个 a.out 文件可能会同时使用该目标代码。
82. DLLs (Dynamic-Link Libraries)：动态链接库，它是微软公司在操作系统中实现 共享函数库 概念的一种实现方式。这些库函数的扩展名是 .DLL、.OCX（包含ActiveX控制的库）或者.DRV（旧式的系统驱动程序）。
83. 客户端(clients)：客户端是访问服务器提供的服务的计算机硬件或软件。
84. 服务端(servers)： 在计算中，服务器是为其他程序或设备提供功能的计算机程序或设备，称为 服务端

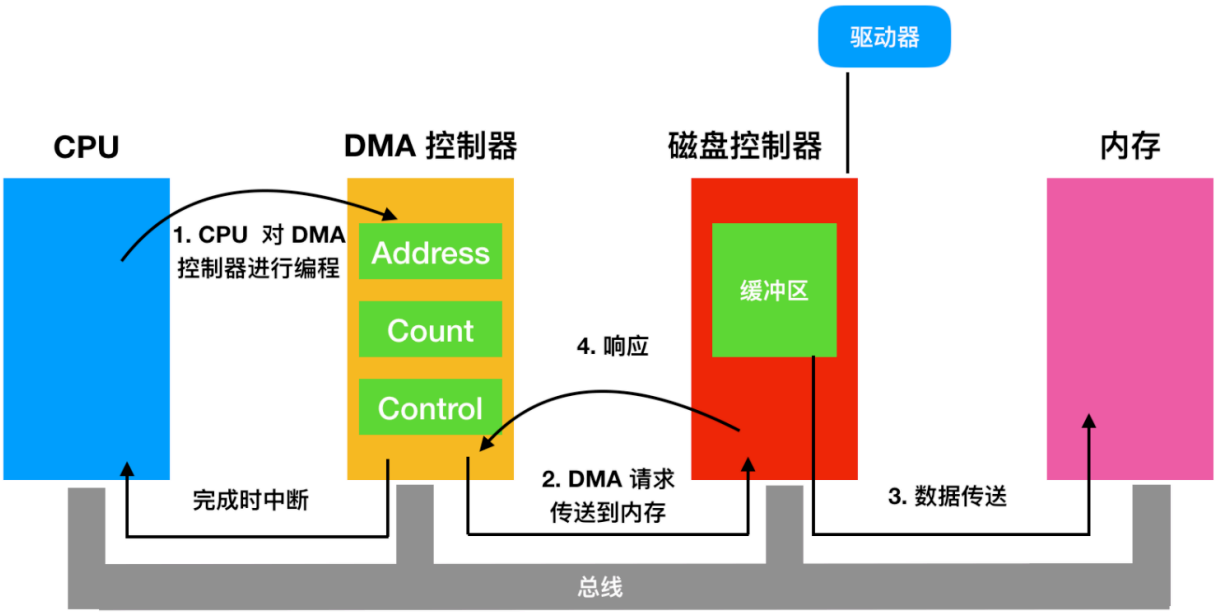
85. `主从架构(client-server)`：主从式架构也称`客户端/服务器`架构、`C/S`架构，是一种网络架构，它把客户端与服务器区分开来。每一个客户端软件的实例都可以向一个服务器或应用程序服务器发出请求。有很多不同类型的服务器，例如文件服务器、游戏服务器等。



86. `虚拟机(Virtual Machines)`：在计算机科学中的体系结构里，是指一种特殊的软件，可以在计算机平台和终端用户之间创建一种环境，而终端用户则是基于虚拟机这个软件所创建的环境来操作其它软件。



87. `Java 虚拟机(Jaav virtual Machines)`：Java虚拟机有自己完善的硬体架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。JVM屏蔽了与具体操作系统平台相关的信息，使得Java程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。
88. `目标文件(object file)`：目标文件是包含 `目标代码` 的文件，这意味着通常无法直接执行的可重定位格式的机器代码。 目标文件有多种格式，相同的目标代码可以打包在不同的目标文件中。目标文件也可以像共享库一样工作。
89. `C preprocessor`： C 预处理å器是 C 语言、C++ 语言的预处理器。用于在编译器处理程序之前预扫描源代码，完成头文件的包含，宏扩展，条件编译，行控制等操作。
90. `设备控制器(device controller)`： 设备控制器是处理 CPU 传入信号和传出信号的系统。设备通过插头和插座连接到计算机，并且插座连接到设备控制器。
91. `ECC(Error-Correcting Code)`： 指能够实现错误检查和纠正错误技术的内存。
92. `I/O port`：也被称为输入/输出端口，它是由软件用来与计算机上的硬件进行通信的内存地址。
93. `内存映射I/O(memory mapped I/O, MMIO)`：内存映射的 I/O 使用相同的地址空间来寻址内存和 I/O 设备，也就是说，内存映射I/O 设备共享同一内存地址。
94. `端口映射I/O(Port-mapped I/O ,PMIO)`： 在 PMIO中，内存和I/O设备有各自的地址空间。 端口映射I/O通常使用一种特殊的CPU指令，专门执行I/O操作。
95. `DMA (Direct Memory Access)`： 直接内存访问，它是计算机系统的一项功能，它允许某些硬件系统能够独立于 CPU 访问内存。如果没有 DMA，当 CPU 执行输入/输出指令时，它通常在读取或写入操作的整个过程中都被完全占用，因此无法执行其他工作。使用 DMA 后，CPU 首先启动传输信号，然后在进行传输时执行其他操作，最后在完成操作后从 DMA 控制器（DMAC）接收中断。完成执行。



DMA 传送操作

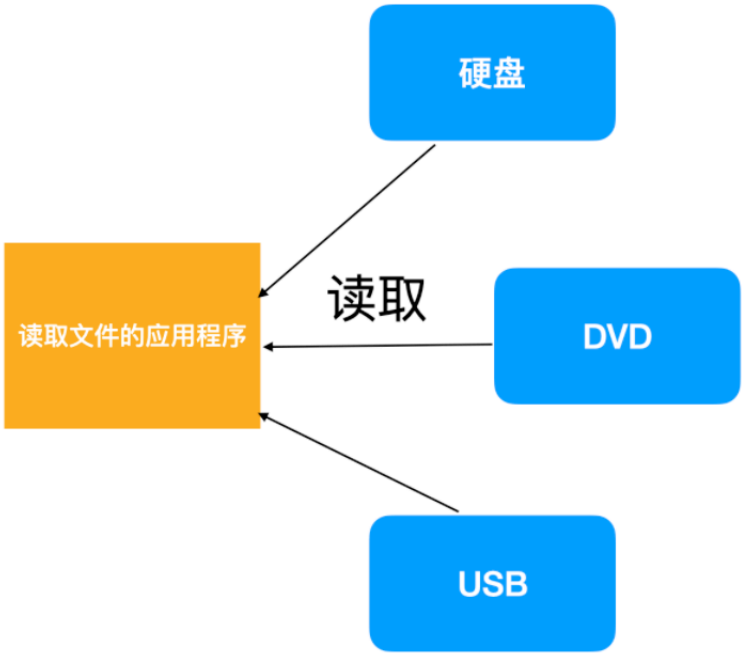
96. `周期窃取(cycle stealing)`：许多总线能够以两种模式操作：**每次一字模式和块模式**。一些 DMA 控制器也能够使用这两种方式进行操作。在前一个模式中，DMA 控制器请求传送一个字并得到这个字。如果 CPU 想要使用总线，它必须进行等待。设备可能会偷偷进入并且从 CPU 偷走一个总线周期，从而轻微的延迟 CPU。它类似于直接内存访问（DMA），允许I / O控制器在无需 CPU 干预的情况下读取或写入RAM。
97. `突发模式(burst mode)`： 指的是设备在不进行单独事务中重复传输每个数据所需的所有步骤的

情况下，重复传输数据的情况。

98. `中断向量表(interrupt vector table)`： 用来形成相应的中断服务程序的入口地址或存放中断服务程序的首地址称为中断向量。 中断向量表是中断向量的集合，中断向量是中断处理程序的地址。
99. `精确中断(precise interrupt)`：精确中断是一种能够使机器处于良好状态下的中断，它具有如下特征

- PC （程序计数器）保存在一个已知的地方
- PC 所指向的指令之前所有的指令已经完全执行
- PC 所指向的指令之后所有的指令都没有执行
- PC 所指向的指令的执行状态是已知的

100. `非精确中断(imprecise interrupt)`：不满足以上要求的中断，指令的执行时序和完成度具有不确定性，而且恢复起来也非常麻烦。
101. `设备独立性(device independence)`：**我们编写访问任何设备的应用程序，不用事先指定特定的设备**。比如你编写了一个能够从设备读入文件的应用程序，那么这个应用程序可以从硬盘、DVD 或者 USB 进行读入，不必再为每个设备定制应用程序。这其实就体现了设备独立性的概念。



102. `UNC(Uniform Naming Convention)`：UNC 是统一命名约定或统一命名约定的缩写，是用于命名和访问网络资源（例如网络驱动器，打印机或服务器）的标准。 例如，在 MS-DOS 和 Microsoft Windows 中，用户可以通过键入或映射到类似于以下示例的共享名来访问共享资源。

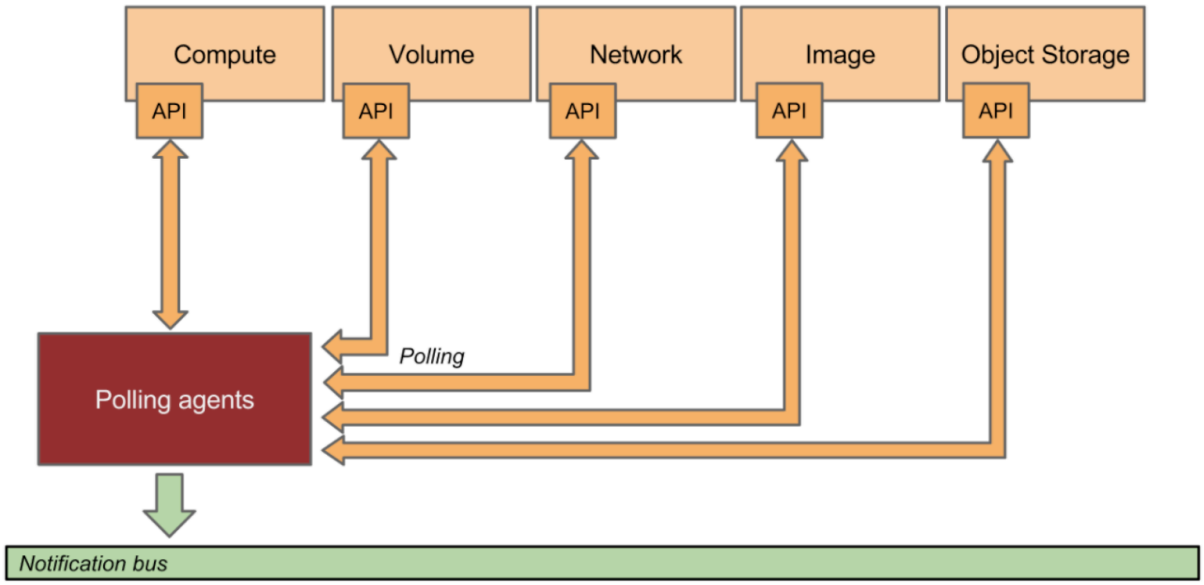
```
\\computer\path
```

然而，在 UNIX 和 Linux 中，你会像如下这么写

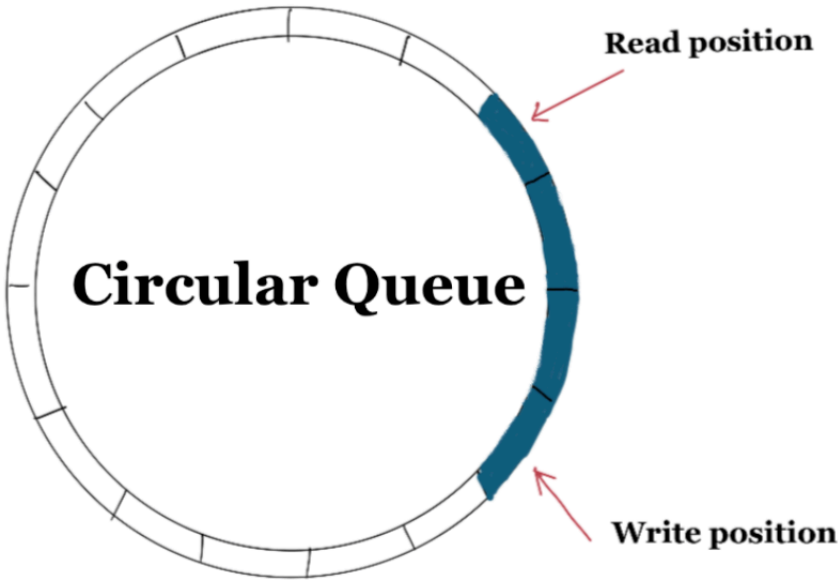
```
//computer/path
```

103. `挂载(mounting)`：挂载是指操作系统会让存储在硬盘、CD-ROM 等资源设备上的目录和文件，通过文件系统能够让用户访问的过程。
104. `错误处理(Error handling)`： 错误处理是指对软件应用程序中存在的错误情况的响应和恢复过程。

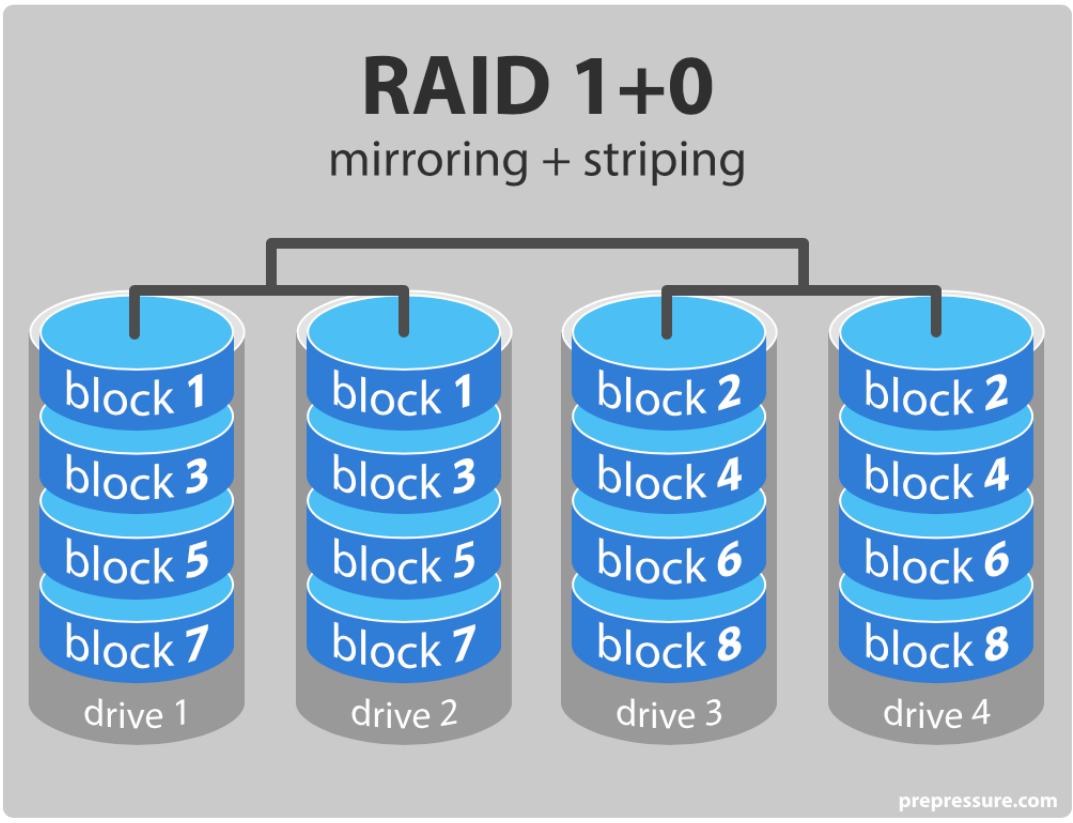
105. `同步阻塞(synchronous)`： 同步是阻塞式的，CPU 必须等待同步的处理结果。
106. `异步响应(asynchronous)`： 异步是由中断驱动的，CPU 不用等待每个操作的处理结果继而执行其他操作
107. `缓冲区(buffering)`： 缓冲区是内存的临时存储区域，它的出现是为了加快内存的访问速度而设计的。对于经常访问的数据和指令来说，CPU 应该访问的是缓冲区而非内存
108. `Programmed input-output,PIO`：它指的是在 CPU 和外围设备（例如网络适配器或 ATA 存储设备）之间传输数据的一种方法。
109. `轮询(polling)`： 轮询是指通过客户端程序主动通过对每个设备进行访问来获得同步状态的过程。



110. `忙等(busy waiting)`：当一个进程正处在某临界区内，任何试图进入其临界区的进程都必须等待，陷入忙等状态。连续测试一个变量直到某个值出现为止，称为忙等。
111. `可重入(reentrant)`： 如果一段程序或者代码在任意时刻被中断后由操作系统调用其他程序或者代码，这段代码调用子程序并能够正确运行，这种现象就称为可重入。也就是说当该子程序正在运行时，执行线程可以再次进入并执行它，仍然获得符合设计时预期的结果。
112. `主设备编号(major device number)、副设备编号(minor device number)` ： 所有设备都有一个主，副号码。 主号码是更大，更通用的类别（例如硬盘，输入/输出设备等），而次号码则更具体（即告诉设备连接到哪条总线）。
113. `多重缓冲区(double buffering)`： 它指的是使用多个缓冲区来保存数据块，每个缓冲区都保留数据块的一部分，读取的时候通过读取多个缓冲区的数据进而拼凑成一个完整的数据。
114. `环形缓冲区(circular buffer)`： 它指的是首尾相连的缓冲区，常用来实现数据缓冲流。

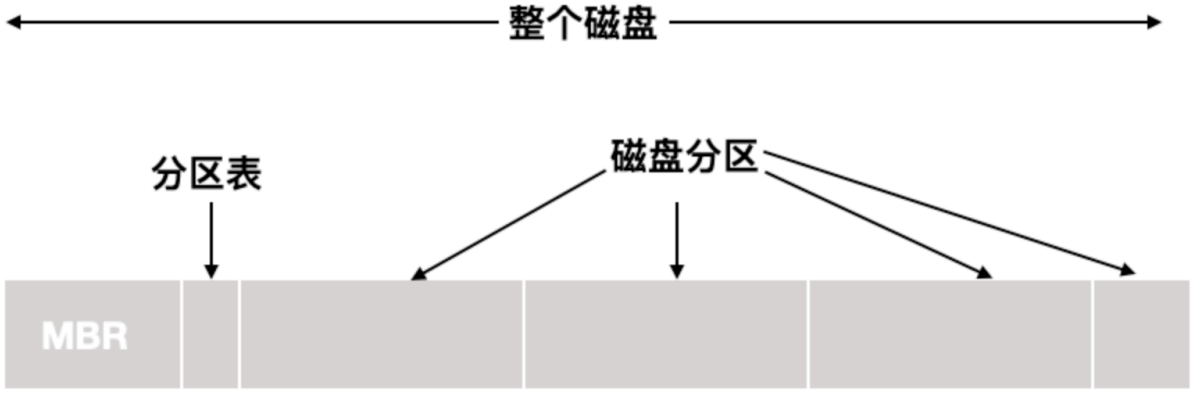


115. `假脱机(Spooling)` ：假脱机是多程序的一种特殊形式，目的是在不同设备之间复制数据。 在现代系统中，通常用于计算机应用程序和慢速外围设备（例如打印机）之间的中介。
116. `守护进程(Daemon)` ： 在计算机中，守护程序是作为后台进程运行的计算机程序，而不是在交互式用户的直接控制下运行的程序。
117. `逻辑块寻址(logical block addressing, LBA)` ：逻辑块寻址是一种通用方案，用于指定存储在计算机存储设备上的数据块的位置。
118. `RAID` ：全称是 Redundant Array of Inexpensive Disks ，廉价磁盘或驱动器的冗余阵列，它是一种数据存储虚拟化的技术，将多个物理磁盘驱动器组件组合成一个或多个逻辑单元，以实现数据冗余，改善性能。



119. `MBR(Master Boot Record)` ：主引导记录（MBR）是任何硬盘或软盘的第一扇区中的信息，

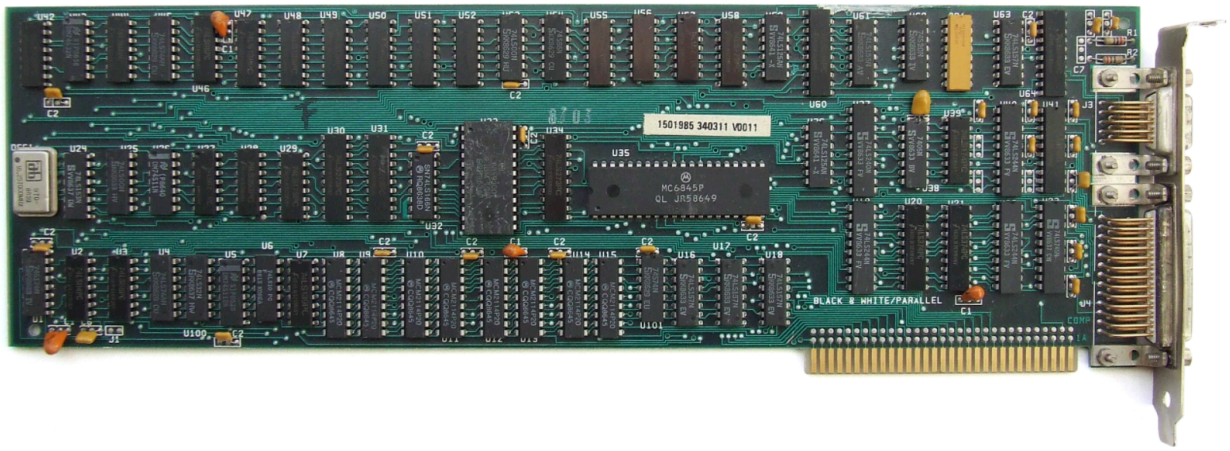
用于标识操作系统的放置方式和位置，以便可以将其加载到计算机的主存储器或随机存取存储器中。



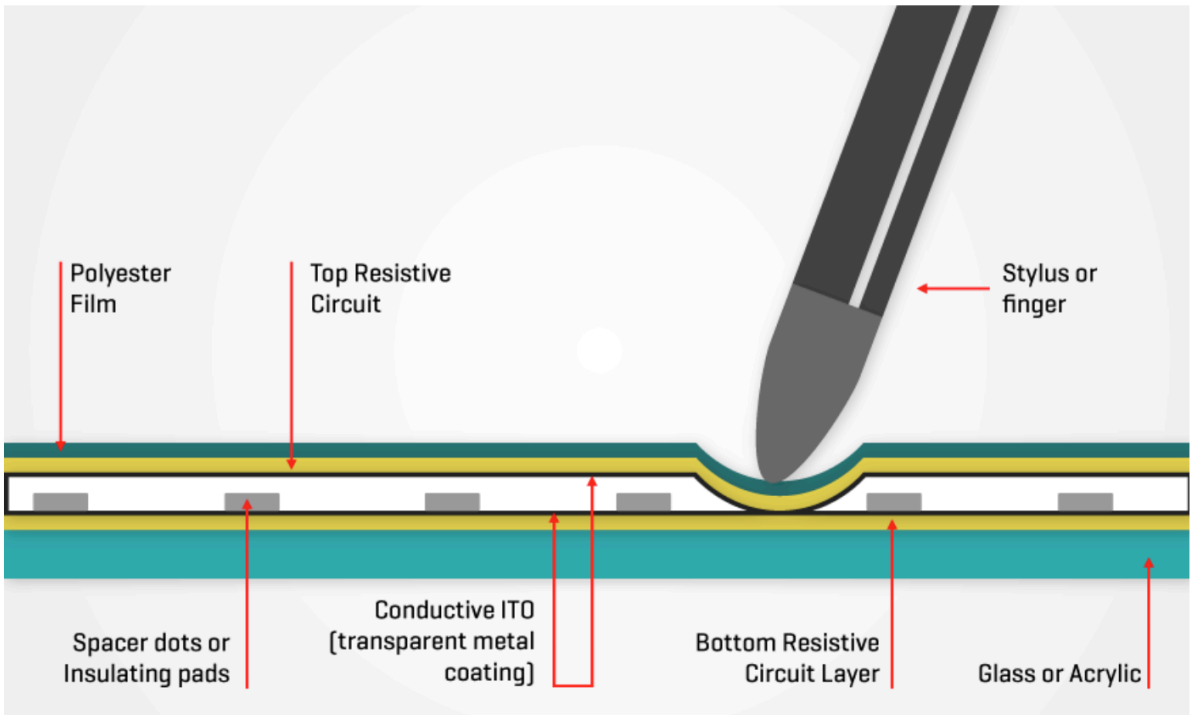
120. `FCFS (First-Come, First-Served)`： 先进先出的调度算法，也就是说，首先到达 CPU 的进程首先进行服务。
121. `SSF (Shortest Seek First)` 最短路径优先算法，这是对先进先出算法的改进，这种算法因为减少了总的磁臂运动，从而缩短了平均响应时间。
122. `稳定存储(stable storage)`： 它是计算机存储技术的一种分类，该技术可确保任何给定的写操作都具有原子性。
123. `时钟(Clocks)`：也被称为 timers。通常，时钟是指调节所有计算机功能的时序和速度的微芯片。芯片中是一个晶体，当通电时，晶体会以特定的频率振动。 任何一台计算机能够执行的最短时间是一个时钟或时钟芯片的一次振动。
124. `QR Code`： 二维码的一种，它的全称是快速响应矩阵图码，能够快速响应。一般应用于手机读码操作，国内火车票上的二维码就是 QR 码



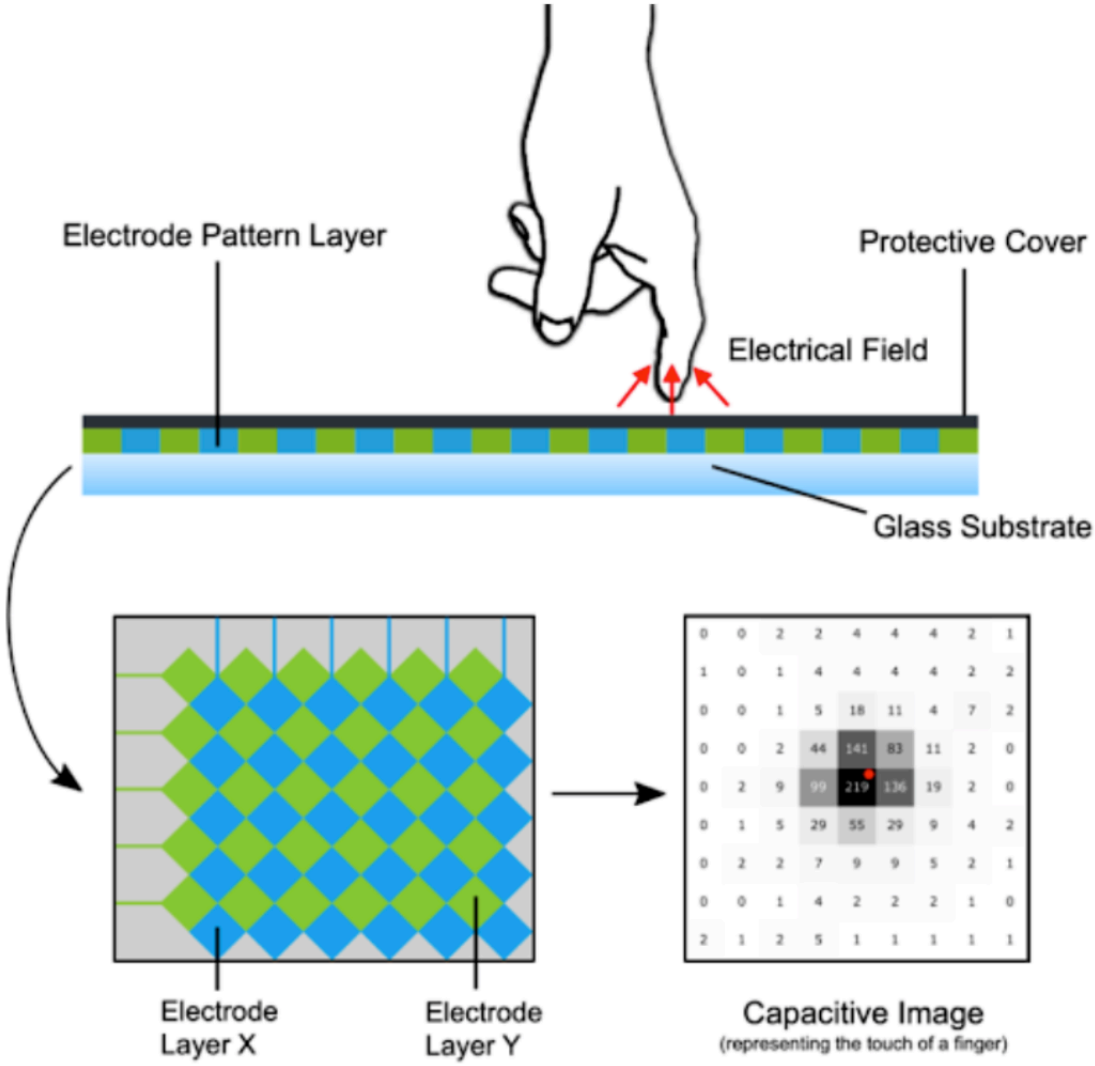
125. `显卡(Video card)`，是个人电脑最基本组成部分之一，用途是将计算机系统所需要的显示信息进行转换驱动显示器，并向显示器提供逐行或隔行扫描信号，控制显示器的正确显示，是连接显示器和个人电脑主板的重要组件，是 `人机对话` 的重要设备之一。



126. `GDI (Graphics Device Interface)`：图形接口，是微软视窗系统提供的应用程序接口，也是其用来表征图形对象、将图形对象传送给诸如显示器、打印机之类输出设备的核心组件。
127. `设备上下文(device context)`：设备上下文是 Windows 数据结构，其中包含有关设备（例如显示器或打印机）的图形属性的信息。 所有绘图调用都是通过设备上下文对象进行的，该对象封装了用于绘制线条，形状和文本的 Windows API。 设备上下文可用于绘制到屏幕，打印机或图元文件。
128. `位图(bitmap)`：在计算机中，位图是从某个域（例如，整数范围）到位的映射。也称为位数组或位图索引。
129. `电阻式触摸屏(Resistive touchscreens)`：电阻式触摸屏基于施加到屏幕上的压力来工作。电阻屏由许多层组成。 当按下屏幕时，外部的后面板将被推到下一层，下一层会感觉到施加了压力并记录了输入。 电阻式触摸屏用途广泛，可以用手指，指甲，手写笔或任何其他物体进行操作。



130. `电容式触摸屏(capacitive touchscreen)`：电容式触摸屏通过感应物体（通常是指尖上的皮肤）的导电特性来工作。 手机或智能手机上的电容屏通常具有玻璃表面，并且不依赖压力。 当涉及到手势（如滑动和捏合）时，它比电阻式屏幕更具响应性。 电容式触摸屏只能用手指触摸，而不能用普通的手写笔，手套或大多数其他物体来响应。



131. `死锁(deadlock)`：死锁常用于并发情况下，`死锁` 是一种状态，死锁中的每个成员都在等待另一个成员（包括其自身）采取行动。

相信你一定看过这个图



132. `可抢占资源(preemptable resource)`：可以从拥有它的进程中抢占而并不会产生任何副作用。
133. `不可抢占资源(nonpreemptable resource)`：与可抢占资源相反，如果资源被抢占后，会导致进程或任务出错。
134. `系统检查点(system checkpointed)`：系统检查点是操作系统（OS）的可启动实例。检查点是计算机在特定时间点的快照。
135. `两阶段加锁(two-phase locking, 2PL)`：经常用于数据库的并发控制，以保证可串行化

这种方法使用数据库锁在两个阶段：

- 扩张阶段：不断上锁，没有锁被释放
- 收缩阶段：锁被陆续释放，没有新的加锁

136. `活锁(Livelock)`：活锁类似于死锁，不同之处在于，活锁中仅涉及进程的状态彼此之间不断变化，没有进展。举一个现实世界的例子，当两个人在狭窄的走廊里相遇时，就会发生活锁，每个人都试图通过移动到一边让对方通过而礼貌，但最终却没有任何进展就左右摇摆，因为他们总是同时移动相同的方式。
137. `饥饿(starvation)`：在死锁或者活锁的状态中，在任何时刻都可能请求资源，虽然一些调度策略能够决定一些进程在某一时刻获得资源，但是有一些进程永远无法获得资源。永远无法获得资源的进程很容易产生 `饥饿` 。

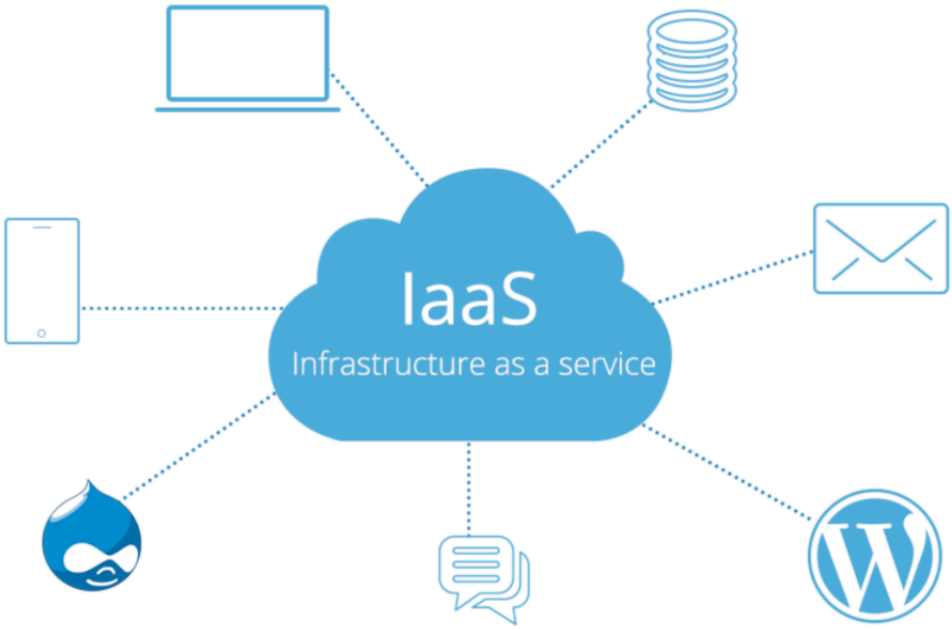
138. 沙盒(sandboxing)：沙盒是一种软件管理策略，可将应用程序与关键系统资源和其他程序隔离。它提供了一层额外的安全保护，可防止恶意软件或有害应用程序对你的系统造成负面影响。
139. VMM (Virtual Machine Monitor)：也被称为 hypervisor，在同一个物理机器上创建出来多态虚拟机器的假象。



140. 虚拟化技术(virtualization)： 是一种资源管理技术，将计算机的各种实体资源（CPU、内存、磁盘空间、网络适配器等），进行抽象、转换后呈现出来并可供分割、组合为一个或多个电脑配置环境。
141. 云(cloud)：云是目前虚拟机最重要、最时髦的玩法。
142. 解释器(interpreter)： 解释器是一种程序，能够把编程语言一行一行解释运行。每次运行程序时都要先转成另一种语言再运行，因此解释器的程序运行速度比较缓慢。它不会一次把整个程序翻译出来，而是每翻译一行程序叙述就立刻运行，然后再翻译下一行，再运行，如此不停地进行下去。
143. 半虚拟化(paravirtualization)： 半虚拟化的目的不是呈现出一个和底层硬件一摸一样的虚拟机，而是提供一个软件接口，软件接口与硬件接口相似但又不完全一样。
144. 全虚拟化(full virtualization)：全虚拟化是硬件虚拟化的一种，允许未经修改的客操作系统隔离运行。对于全虚拟化，硬件特征会被映射到虚拟机上，这些特征包括完整的指令集、I/O操作、中断和内存管理等。
145. 客户操作系统(guest operating system)：客户操作系统是安装在计算机上操作系统之后的操作系统，客户操作系统既可以是分区系统的一部分，也可以是虚拟机设置的一部分。客户操作系统为设备提供了备用操作系统。
146. 主机操作系统(host operating system)： 主机操作系统是计算机系统的硬盘驱动器上安装的主要操作系统。 在大多数情况下，只有一个主机操作系统。
147. 应用编程接口(Application Programming Interface,API)：应用程序编程接口（API）是软件组件或系统的编程接口，它定义其他组件或系统如何使用它。
148. 虚拟机接口(Virtual Machine Interface, VMI)：它是一个高速接口，同一主机上的虚拟机（VM）可用于相互之间以及主机内核模块之间进行通信。
149. 输入输出内存管理单元(Input-output memory management unit, I/O MMU)：在计算机中，输入输出内存管理单元（IOMMU）是将直接内存访问（DMA）I / O 总线连接到主存的内存管理单元（MMU）。
150. 设备穿透(device pass through)：它允许将物理设备直接分配给特定虚拟机。
151. 设备隔离(device isolation)：保证设备可以直接访问其分配到的虚拟机的内存而不影响其他

虚拟机的完整性。

152. `基础设施即服务(IAAS (Infrastructure As A Service))`：基础架构即服务（IaaS）是一种即时计算基础架构，可通过 Internet 进行配置和管理。 它是四种云服务类型之一，另外还有软件即服务（SaaS），平台即服务（PaaS）和无服务器。



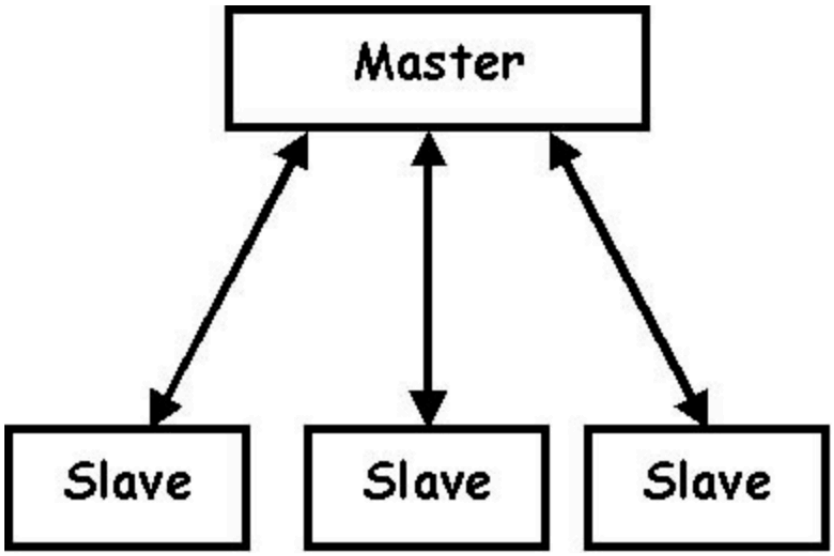
153. `平台即服务(PAAS (Platform As A Service))`：平台即服务（PaaS）或应用程序平台即服务（aPaaS）或基于平台的服务是云计算服务的一种，它提供了一个平台，使客户可以开发，运行和管理应用程序，而无需构建和维护该应用程序。



154. `软件即服务(SAAS(Software As A Service))`：它是一个提供特定软件服务访问的平台，是一种软件许可和交付模型，在该模型中，软件是基于订阅许可的，并且是集中托管的。

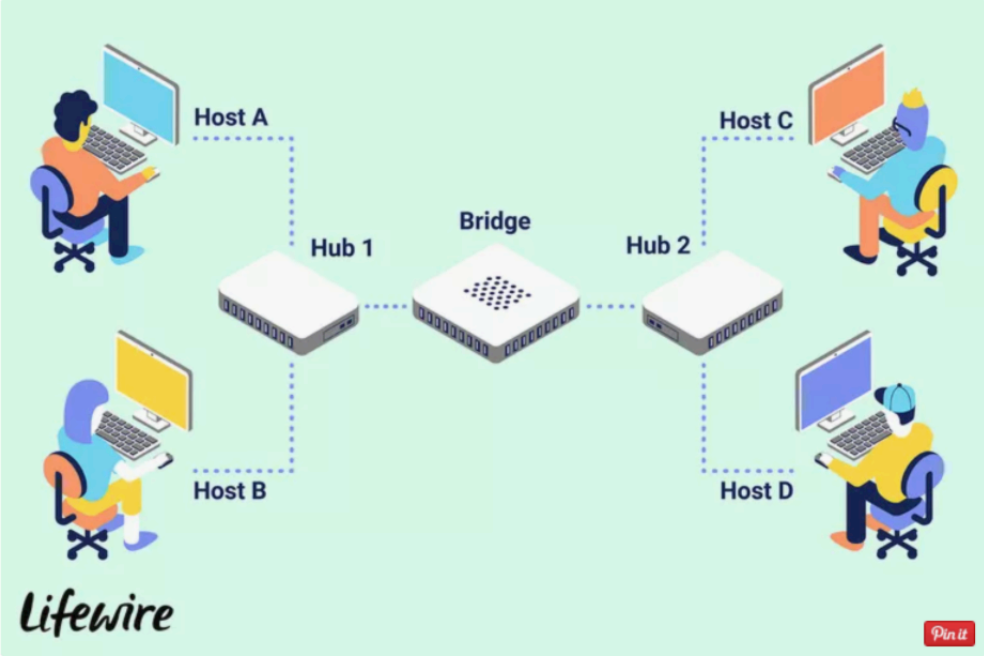


155. `实时迁移(live migration)`： 实时迁移是指在不断开客户端或应用程序连接的情况下，在不同的物理机之间移动正在运行的虚拟机或应用程序的过程，一般经常采用的方式是内存预复制迁移
156. `写入时复制(copy on write)`：写入时复制是一种计算机程序设计领域的优化策略。其核心思想是，如果有多个 `调用者（callers）` 同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份 `专用副本（private copy）` 给该调用者，而其他调用者所见到的最初的资源仍然保持不变
157. `主从模型(master-slave)`：主/从是一种不对称通信或控制的模型，其中一个设备进程控制一个或多个其他设备或进程并充当其通信中心。 在某些系统中，从一组合格的设备中选择一个主设备，而其他设备则充当从设备的角色。

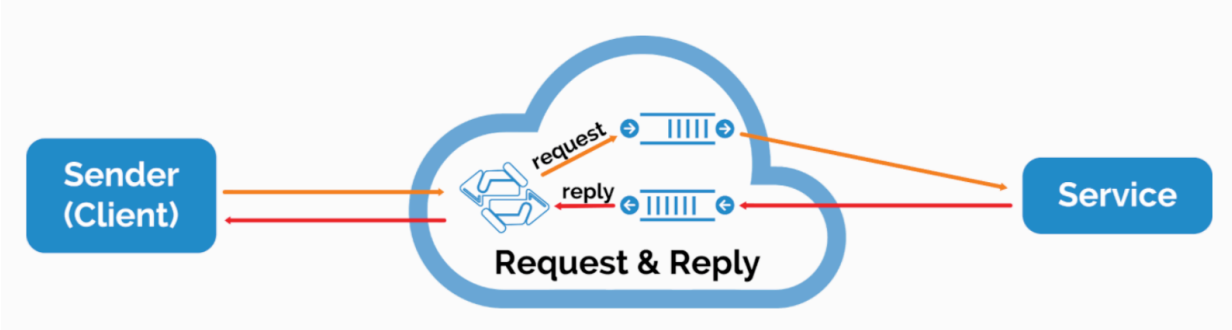


158. `分布式系统(distributed system)`：分布式系统，也称为分布式计算，是一种具有位于不同机器上的多个组件的系统，这些组件可以通信和协调动作，以便对最终用户显示为单个一致的系统。

159. `局域网(LANs, Local Area Networks)`：局域网（LAN）是一种计算机网络，可将住宅，学校，实验室，大学校园或办公大楼等有限区域内的计算机互连。
160. `广域网(WAN,Wide Area Network)`：又称**广域网**、**外网**、**公网**。是连接不同地区局域网或城域网计算机通信的远程网。通常跨接很大的物理范围，所覆盖的范围从几十公里到几千公里，它能连接多个地区、城市和国家，或横跨几个洲并能提供远距离通信，形成国际性的远程网络。
161. `以太网(Ethernet)`：以太网是一种计算机局域网的技术，它规定了包括物理层的连线、电子信号和介质访问层协议的内容。
162. `桥接器(bridge)`：当指代计算机时，网桥是连接两个 LAN（局域网）或同一 LAN 的两个网段的设备。与路由器不同，网桥是独立于协议的。他们转发数据包时无需分析和重新路由消息。



163. `主机(host)`：在网络硬件中，主机又被称为网络主机，网络主机是连接到计算机网络的计算机或其他设备。主机可以充当服务器，向网络上的用户或其他主机提供信息资源，服务和应用程序。主机被分配至少一个网络地址。
164. `路由器(router)`：路由器是在计算机网络之间转发数据包的联网设备。通过互联网发送的数据（例如网页或电子邮件）以数据包的形式出现。
165. `面向连接的服务(Connection-oriented service)`：面向连接的服务是一种在数据通信开始之前在通信实体之间建立专用连接的服务。要使用面向连接的服务，用户首先建立一个连接，使用它，然后释放它。TCP 就是一种面向连接的服务，在发送数据包之前需要经过握手操作。
166. `无连接的服务(Connectionless service)`：无连接服务是两个节点之间的数据通信，其中发送方在不确保接收方是否可以接收数据的情况下发送数据。此处，每个数据包都具有目标地址，并且与其他数据包无关地独立路由。UDP 就是一种无连接的服务，发送数据包不需要经过握手连接。
167. `服务质量(quality of service, QoS)`：服务质量是对服务整体性能的描述或度量，尤其是网络用户看到的性能。
168. `确认包(acknowledgement packet)`：在数据网络，电信和计算机总线中，确认（ACK）是作为通信协议一部分在通信过程，计算机或设备之间传递以表示确认或消息接收的信号。
169. `请求-响应服务(request-reply service)`：请求-响应是计算机彼此通信的基本方法之一，其中第一台计算机发送对某些数据的请求，第二台计算机对请求进行响应。



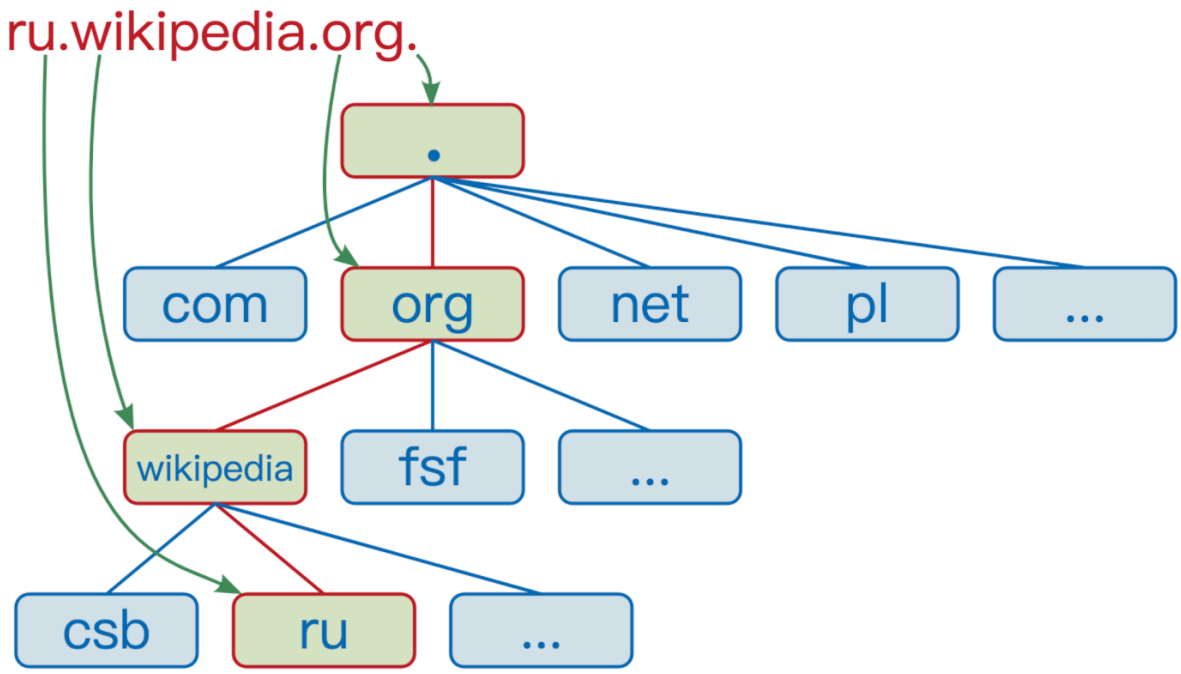
170. `协议栈(protocol stack)`：所有现代网络都使用所谓的协议栈把不同的协议一层一层叠加起来。每一层解决不同的问题。



171. `IP地址`：标示互联网上每一台主机有两种方式，一种是 IPv4，一种是 IPv6。
172. `超链接(hyperlink)`：超链接是可以单击以跳到新文档或当前文档中新部分的单词，短语或图像。 几乎在所有网页中都可以找到超链接，从而允许用户单击页面之间的方式。 文本超链接通常为蓝色并带有下划线。
173. `Web 页面(Web page)`：网页是一个适用于万维网和网页浏览器的文件。
174. `Web浏览器`：Web浏览器（通常称为浏览器）是一种用于访问 Internet 上的信息的软件应用程序。 当用户请求特定网站时，Web 浏览器从 Web 服务器检索必要的内容，然后在用户的设备上显示结果网页。
175. `漏洞(vulnerability)`：漏洞是一种系统不安全级别的错误。
176. `漏洞利用(exploit)`：漏洞利用是计算机安全术语，指的是利用程序中的某些漏洞，来得到计算机的控制权。
177. `病毒(virus)`：计算机病毒是一种计算机程序，在执行时会通过修改其他计算机程序并插入自己的代码来自我复制。复制成功后，可以说受影响的区域已被计算机病毒 `感染` 。



178. `CIA(Confidentiality,Integrity,Availability)`：安全系统的三个指标，即机密性、完整性和可用性。
179. `黑客(cracker)`：黑客是指经常通过网络闯入他人计算机系统的人。 绕过计算机程序中的密码或许可证； 或以其他方式故意破坏计算机安全性。 黑客可能会出于恶意，出于某些利他目的或原因，或者是因为存在挑战而牟取暴利。 表面上已经进行了一些破解和输入，以指出站点安全系统中的弱点。
180. `端口扫描(portscan)`：端口扫描程序是一种旨在探测服务器或主机是否存在开放端口的应用程序。 管理员可以使用这种应用程序来验证其网络的安全策略，攻击者可以使用这种应用程序来识别主机上运行的网络服务并利用漏洞。
181. `僵尸网络(botnets)`：僵尸网络是指骇客利用自己编写的分布式拒绝服务攻击程序将数万个沦陷的机器，即骇客常说的傀儡机或 `肉鸡` 。
182. `域(domain)`：网域名称，简称域名、网域，是由一串用点分隔的字符组成的互联网上某一台计算机或计算机组的名称，用于在数据传输时标识计算机的电子方位。



183. `盐(solt)`：在密码学中，盐是随机数据，用作哈希数据，密码或密码的单向函数的附加输入。
184. `逻辑炸弹(logic bomb)`： 是一些嵌入在正常软件中并在特定情况下执行的恶意程式码。这些特定情况包括更改档案、特别的程式输入序列、特定的时间或日期等。恶意程式码可能会将档案删除、使电脑主机当机或是造成其他的损害。
185. `定时炸弹(time bomb)`：在计算机软件中，定时炸弹是已编写的计算机程序的一部分，因此它会在达到预定的日期或时间后开始或停止运行。
186. `登陆欺骗(login spoofing)`：登录欺骗是用于窃取用户密码的技术。它会向用户显示一个普通的登录提示，提示用户名和密码，这实际上是一个恶意程序，通常在攻击者的控制下称为特洛伊木马。
187. `后门程序(backdoor)`：软件后门指绕过软件的安全性控制，从比较隐秘的通道获取对程序或系统访问权的黑客方法。
188. `防火墙(firewall)`：防火墙在计算机科学领域中是一个架设在互联网与企业内网之间的信息安全系统，根据企业预定的策略来监控往来的传输。

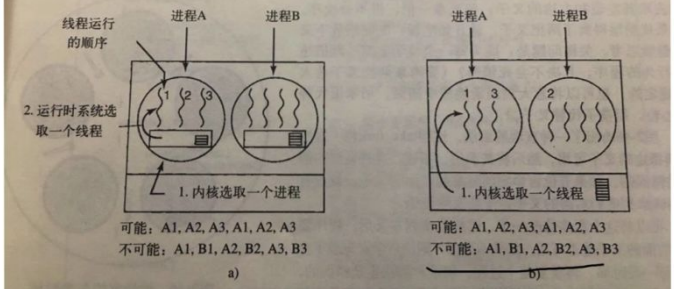
勘误

对了，如果你刷《现代操作系统》第四版的时候，有三个问题需要注意一下

部时间片，内核就会选择另一个进程运行。

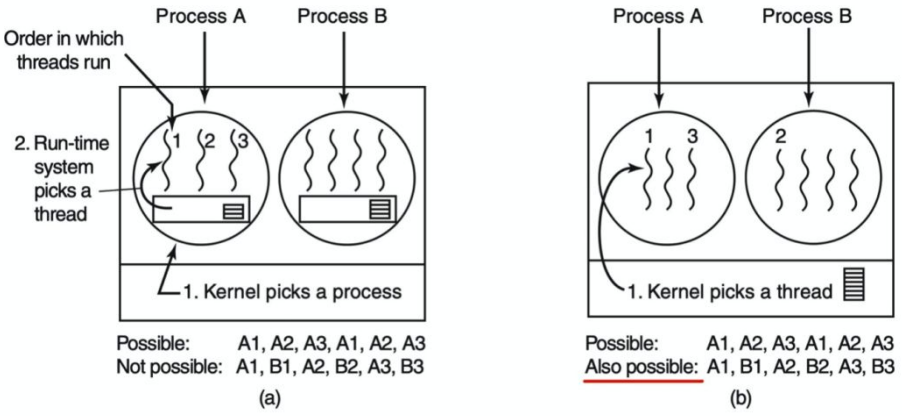
在进程A终于又一次运行时，线程A1会接着运行。该线程会继续耗费A进程的所有时间，直到它完成工作。不过，该线程的这种不合群的行为不会影响到其他的进程。其他进程会得到调度程序所分配的合适份额，不会考虑进程A内部所发生的事。

现在考虑A线程每次CPU计算的工作比较少的情况，例如，在50ms的时间片中有5ms的计算工作于是，每个线程运行一会儿，然后把CPU交回给线程调度程序。这样在内核切换到进程B之前，就会有序列A1，A2，A3，A1，A2，A3，A1，A2，A3，A1。这种情形可用图2-44a表示。



-44 a) 用户级线程的可能调度，有50ms时间片的进程以及每次运行5ms CPU的线程；b) 与a) 有

现代操作系统 第四版，p93页

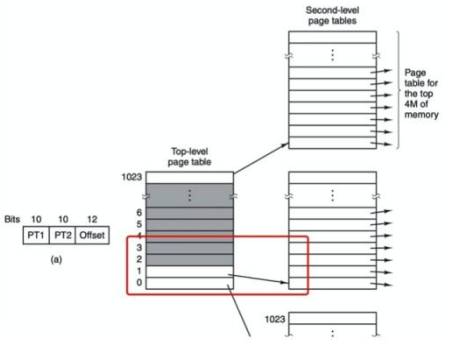


知乎 @cxuan

table for the data, and entry 1023 points to the page table for the stack. The other (shaded) entries are not used. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

As an example, consider the 32-bit virtual address 0x00403004 (4,206,596 decimal), which is 12,292 bytes into the data. This virtual address corresponds to *PT1* = 1, *PT2* = 3, and *Offset* = 4. The MMU first uses *PT1* to index into the top-

206 MEMORY MANAGEMENT CHAP. 3



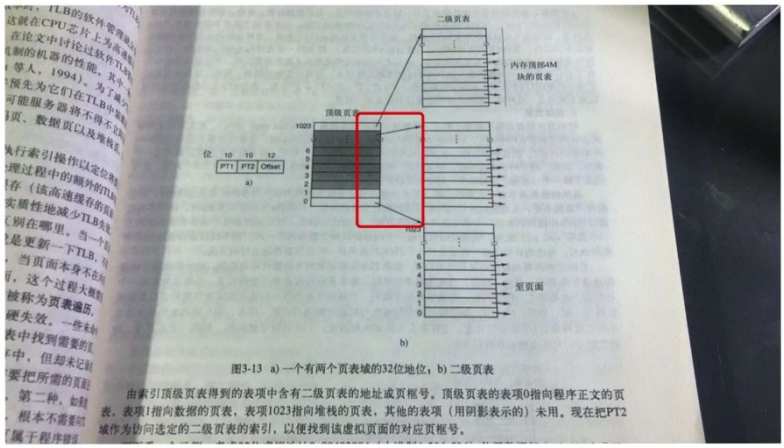
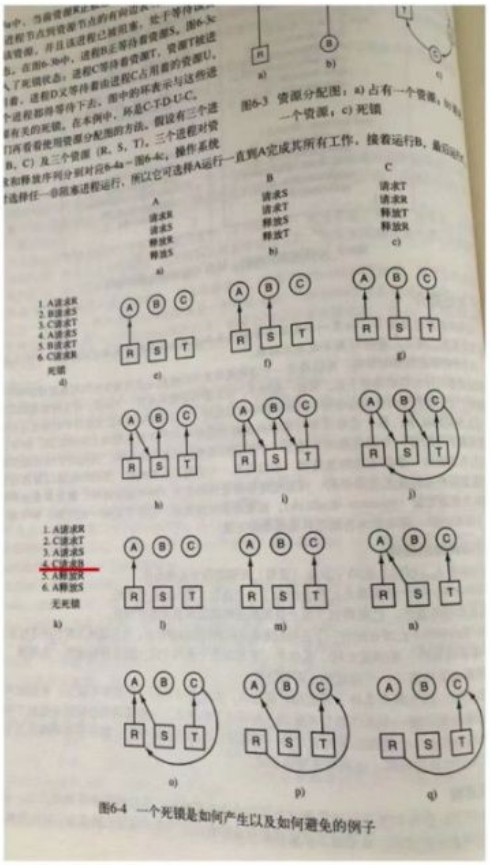


图3-13 a)一个有两个页表域的32位地位；b) 二级页表

由索引顶级页表得到的表项中含有二级页表的地址或页框号。顶级页表的表项0指向程序正文的页表，表项1指向数据的页表，表项1023指向堆栈的页表，其他的表项（用阴影表示的）未用。现在把PT2域作为访问选定的二级页表的索引，以便找到该虚拟页面的对应页框号。

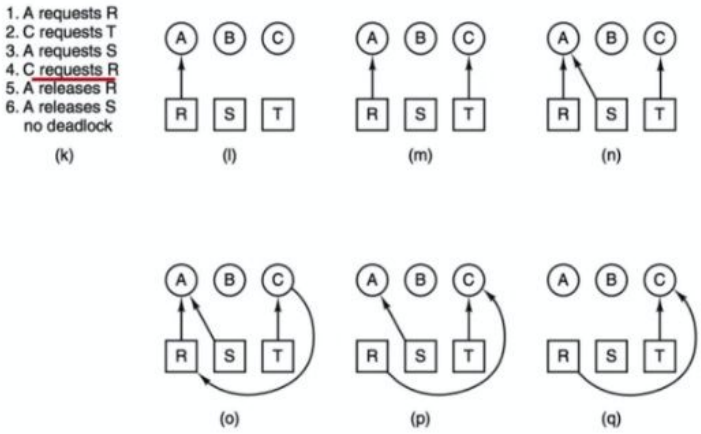
P115页

知乎 @cxuan



《现代操作系统》第四版 P250

C 请求 B
应该为 C 请求 R



知乎 @程序员cxuan

知错能改，善莫大焉。



写文不易，如果认可还请读者支持



