操作系统回顾

操作系统本身是一种软件，但其十分复杂，提供了以下几种抽象模型：

- 文件：对I/O设备的抽象
- 虚拟内存：对程序存储器的抽象
- 进程：对运行程序的抽象
- 虚拟机：对整个操作系统的抽象

0.引入

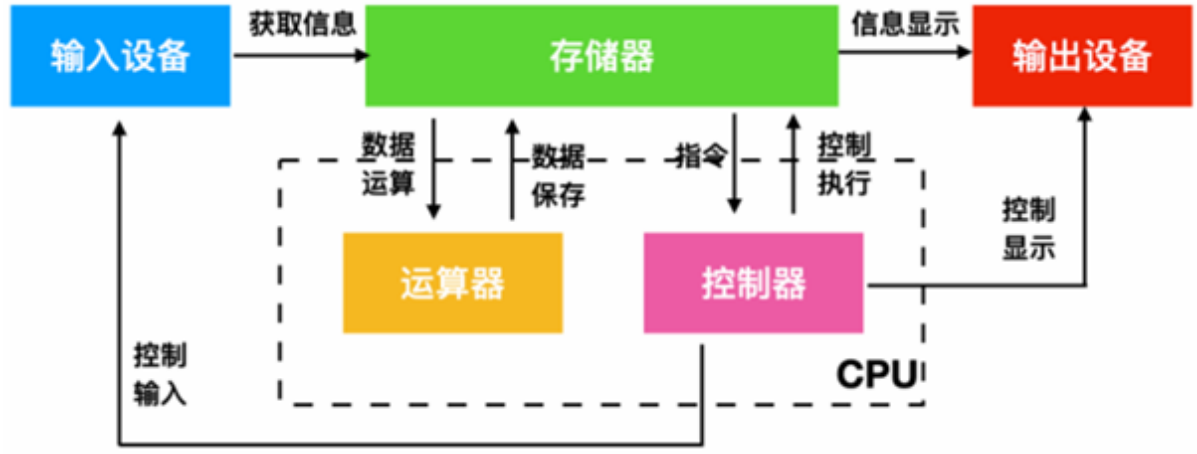
现代计算机系统由**一个或多个处理器、主存、打印机、键盘、鼠标、显示器、网络接口以及各种输入/输出设备构成的系统。**但实际上我们不可能直接操作这些硬件，因此需要在硬件的基础上安装一个软件来对这些硬件进行管理，为用户提供交互接口。这个软件就是操作系统。

操作系统不应该受到用户的干扰，因此其运行于**内核态**下，而其余软件运行于**用户态**下。

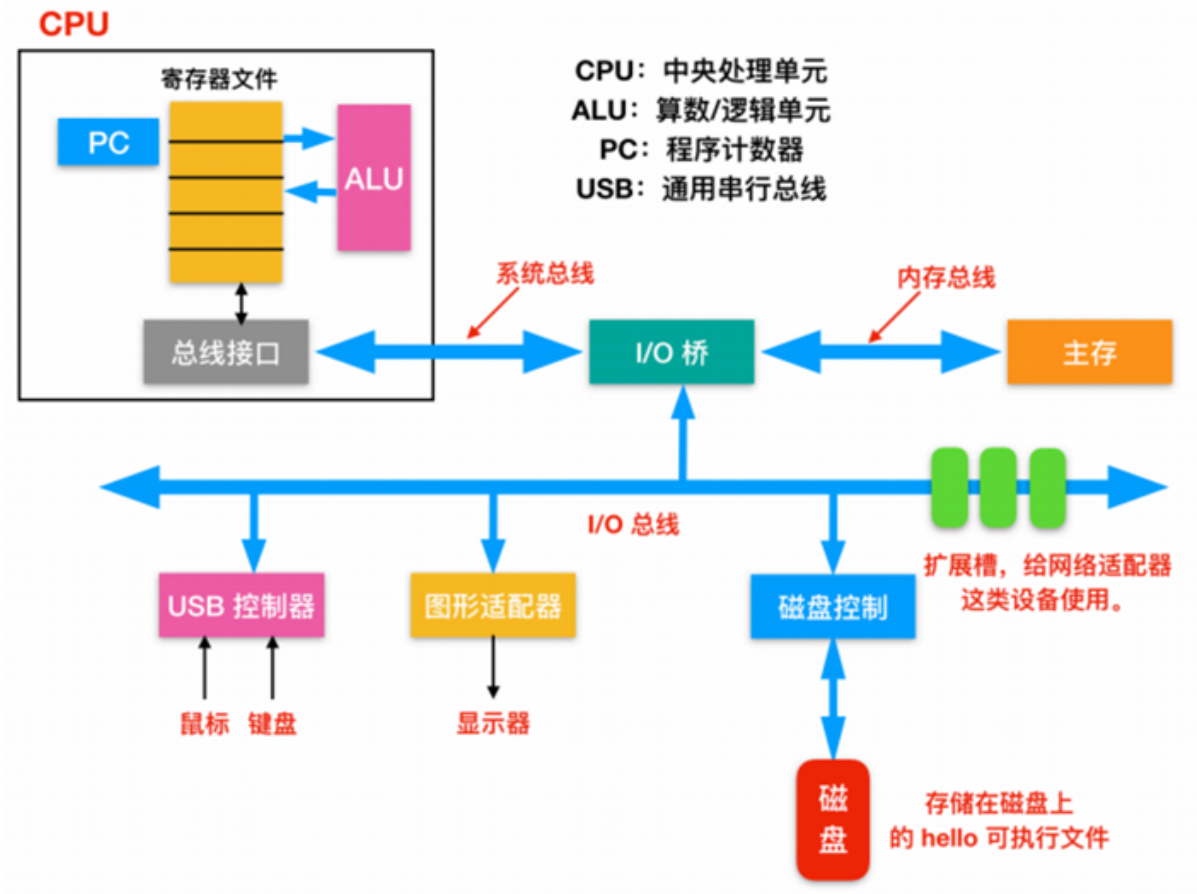
计算机硬件

计算机硬件：**运算器、控制器、存储器、输入设备、输出设备。**

- 冯诺伊曼体系结构（存储、控制）



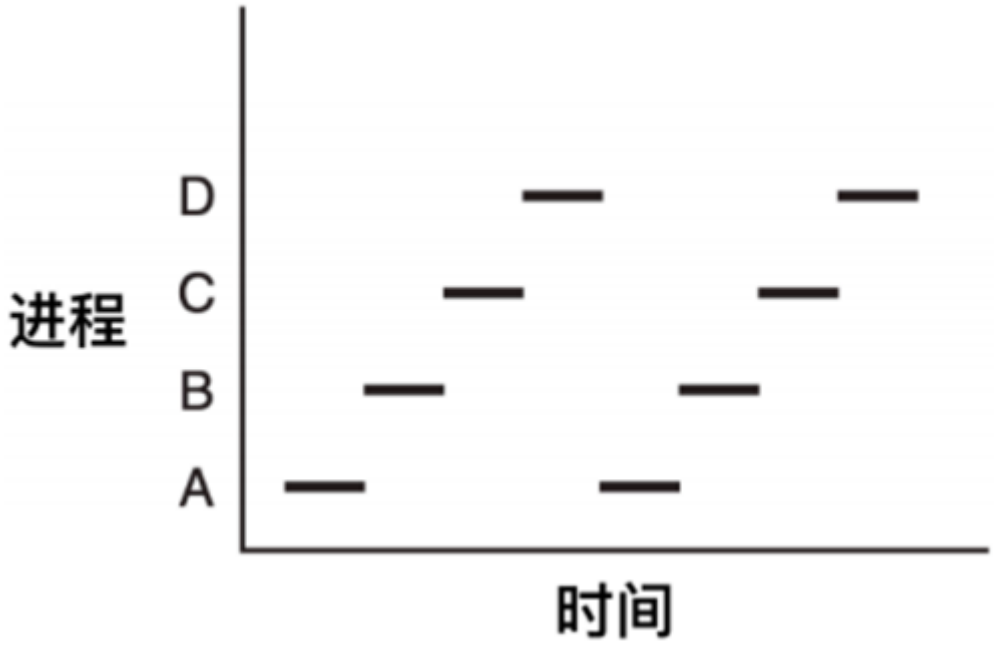
- 现代计算机体系结构（总线结构）



1. 进程和线程

1.1 进程

进程是对操作系统中正在运行的程序的一个抽象，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。在进程看来，自己拥有独有的虚拟CPU，但实际上是CPU在各个进程之间进行快速来回切换（时间片服务方式）。



1.1.1 进程创建

创建进程的方式包含：

1. 操作系统创建：系统初始化时，操作系统会创建若干进程。
2. 正在运行程序创建：执行创建新进程的系统调用(fork)
3. 用户请求创建新进程：例如用户使用某一软件，双击运行本质上就是创建了一个新进程
4. 初始化一个批处理工作。

1.1.2 进程终止

进程终止往往由以下情况触发：

- 自愿退出：程序执行完成了预定工作，放弃资源，终止进程(exit)
- 错误退出：程序执行过程出错，被操作系统终止
- 被其他进程终止：其他进程通过系统调用杀死某个进程(kill)

1.1.3 进程的层次结构

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（树状结构）。

linux中存在这种层次结构，但windows中没有层次概念，认为所有进程都是平等的。

1.1.4 进程状态

- 运行态：获得CPU资源，正在占用CPU进行执行
- 就绪态：可以运行，但是没有获得CPU资源，因此等待其他进程释放资源后运行
- 阻塞态：需要外部事件触发，否则无法运行。

1.1.5 进程的实现

操作系统为了执行进程间的切换，需要维护一张表，即**进程表**(process table)。每个进程占用一个进程表项，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。

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		

1.2 线程

通常来说，进程创建后本身就包含一个线程（主线程），但实际中常常一个进程中包含了很多线程，出现同一地址空间中运行多个控制线程的情况。

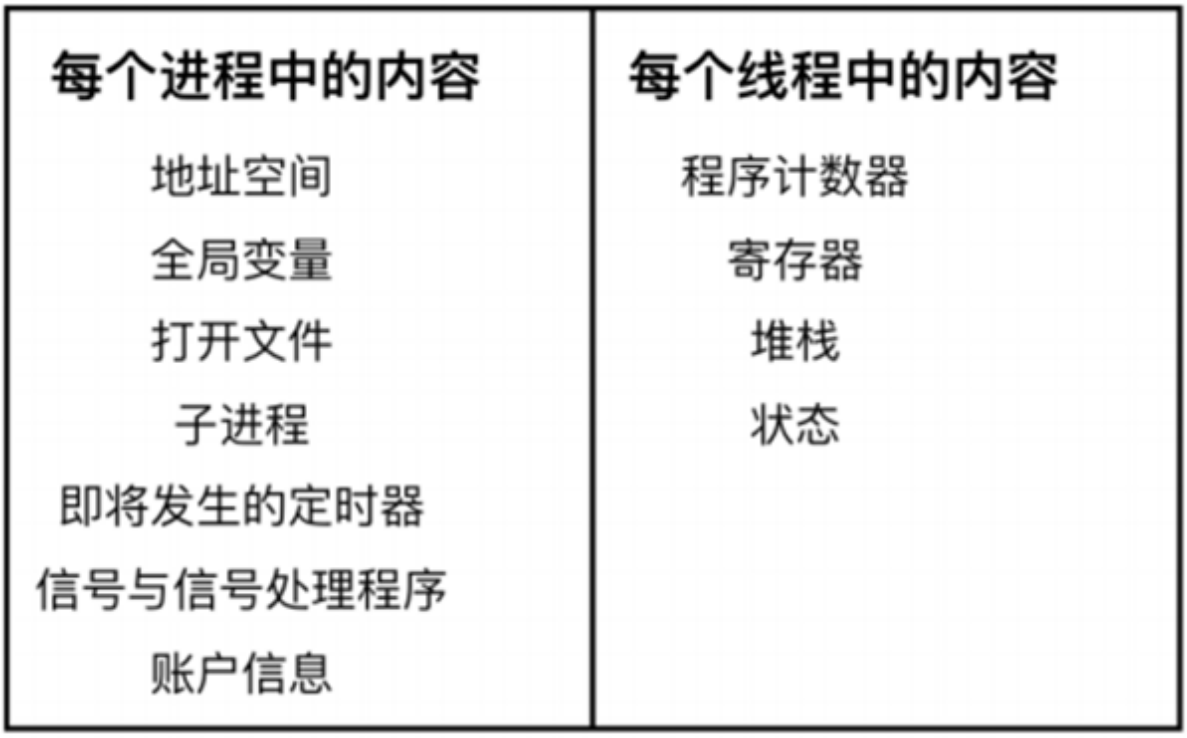
相比进程来说，线程具有以下独特特点：

- 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，而不同进程分别包含不同的地址空间和数据。
- 线程更加轻量级，其创建和销毁所需的资源消耗相比进程更小
- 多线程适合大量的计算和I/O处理，多线程可以使得在这些活动中彼此重叠进行，加快程序执行速度。

相比进程，线程不具备较强的独立性，同一个进程中的所有线程都会有完全一样的地址空间（**共享全局变量**），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。**

线程之间共享同一内存空间和CPU计算资源，但也具有以下线程独有的内容：程序计数器、寄存器、堆栈、状态信息

> 需要说明：多线程编程中，**堆栈**指的仅仅是**栈**（stack），而不是堆（heap）。



1.2.1 线程实现

主要有三种实现方式

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

2. 进程间通信

进程需要频繁和其他进程进行交流，这就涉及进程间通信方式，包含以下6种

2.1 信号 signal

通过向一个或多个进程发送`异步事件信号`来实现，信号可以从键盘或者访问不存在的位置等地方产生；信号通过 shell 将任务发送给子进程。

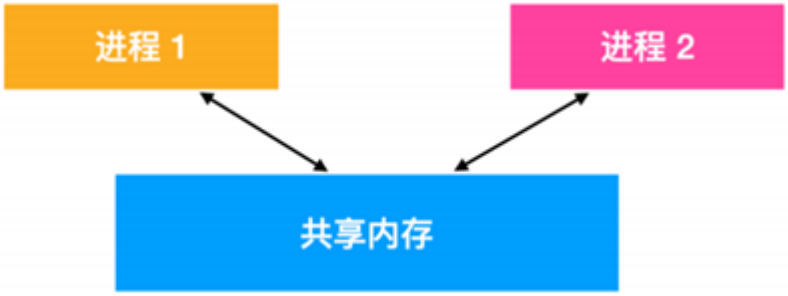
适用于事件通知和进程间的简单控制。

2.2 管道 pipe

在两个进程之间，可以建立一个通道，一个进程向这个通道里写入字节流，另一个进程从这个管道中读取字节流。当管道空时，读进程将阻塞；当管道满时，写进程将阻塞。

2.3 共享内存 shared memory

两个进程之间还可以通过共享内存进行进程间通信，其中两个或者多个进程可以访问公共内存空间。两个进程的共享工作是通过共享内存完成的，一个进程所作的修改可以对另一个进程可见。



2.4 先入先出队列 FIFO / 命名管道(Named Pipes)

命名管道的工作方式与常规管道非常相似。但未命名的管道没有备份文件，命名管道具有支持文件和独特 API ，命名管道在文件系统中作为设备的专用文件存在。当所有的进程通信完成后，命名管道将保留在文件系统中以备后用。命名管道具有严格的 FIFO 行为

2.5 消息队列 Message Queue

消息队列是一种用于在不同进程之间传递消息的通信机制。在消息队列中，进程可以将消息发送到队列中，也可以从队列中接收消息。消息队列提供了一个由系统管理的、按照特定顺序存储消息的缓冲区，允许进程通过队列传递数据。

消息队列有两种模式，一种是`严格模式`， 严格模式就像是 FIFO 先入先出队列似的，消息顺序发送，顺序读取。还有一种模式是`非严格模式`，消息的顺序性不是非常重要。

2.6 套接字 Socket

套接字（Socket） 是一种计算机网络通信的基础接口，通常用于实现不同进程之间的网络通信。它提供了一种标准的机制，允许在计算机网络中不同的计算机或进程间交换数据。套接字既可以用于同一台机器上的进程间通信，也可以用于不同计算机之间的通信。

socket 提供端到端的双向通信。一个套接字可以与一个或多个进程关联。就像管道有命令管道和未命名管道一样，套接字也有两种模式，套接字一般用于两个进程之间的网络通信，网络套接字需要来自诸如`TCP（传输控制协议）`或较低级别`UDP（用户数据报协议）`等基础协议的支持。

- **流式套接字（SOCK_STREAM）**：使用 **TCP** 协议。
- **数据报套接字（SOCK_DGRAM）**：使用 **UDP** 协议。
- **原始套接字（SOCK_RAW）**：允许直接访问网络层协议，如 IP 和 ICMP。主要用于网络协议分析和自定义协议实现。
- **Unix 域套接字（SOCK_UNIX）**：在同一台机器上的进程间通信，不通过网络协议栈，提供更高效的通信。

2.7 其他

信号量（Semaphore）：信号量是一种同步机制，常用于控制多个进程对共享资源的访问。信号量本身并不直接传输数据，但可以用来协调进程间的顺序和访问共享资源的时机。

事件（Event）：一个进程可以设置事件的状态，另一个进程等待事件的发生，以便继续执行。适用于多进程或多线程间的同步，主要用于控制和同步。

3. CPU调度

对于CPU来说，会频繁的有很多进程或者线程来同时竞争 CPU 时间片。如果多个进程/线程都处于就绪状态，选择哪一个进程/线程，将 CPU 时间片分配给它就涉及到调度程序的工作，该程序使用的算法即为**调度算法。**

调度算法的分类：

- 批处理(Batch) : 商业领域
- 交互式(Interactive)： 交互式用户环境
- 实时(Real time)

3.1 批处理调度算法

3.1.1 先来先服务算法

排队思维，谁先来谁就先获得资源

3.1.2 最短作业优先

先做简单的

3.1.3 最短剩余时间优先

最短作业优先的抢占式版本

3.2 交互式系统调度算法

3.2.1 轮询调度（时间片轮转算法）

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。

3.2.2 优先级调度

每个进程都被赋予一个优先级，优先级高的进程优先运行。

3.2.3 多级队列调度

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。

3.2.4 最短进程优先

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。

3.2 实时系统中的调度

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。

4. 内存管理

4.1 地址空间

因为要使多个应用程序同时运行在内存中，所以必须要解决两个问题：`保护`和`重定位`。通常来说，采用地址空间(the address space)来实现

最简单的是使用`动态重定位(dynamic relocation)`技术：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。`动态重定位(dynamic relocation)`技术：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。

另一种方式是采用基址寄存器和变址寄存器。

4.1.1 基址寄存器和变址寄存器

使用基址寄存器和变址寄存器。

- 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。

4.1.2 交换技术

在程序运行过程中，经常会出现内存不足的问题，处理方案有两种，即：**交换**和**虚拟内存**

交换：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。

虚拟内存：允许应用程序部分的运行在内存中。

空闲内存管理

进行内存动态分配时，操作系统必须对其进行管理。有两种监控内存使用的方式：

1. 位图：将内存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）。
2. 空闲列表：维护一个记录已分配内存段和空闲内存段的链表。

   为创建的进程分配内存算法：

   - 首次适配(first fit)：沿着链表进行扫描，直到找个一个足够大的空闲区为止。
   - 下次适配(next fit)：和首次匹配的工作方式相同，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。
   - 最佳适配(best fit)：从头到尾寻找整个链表，找出能够容纳进程的最小空闲区。

4.2 虚拟内存

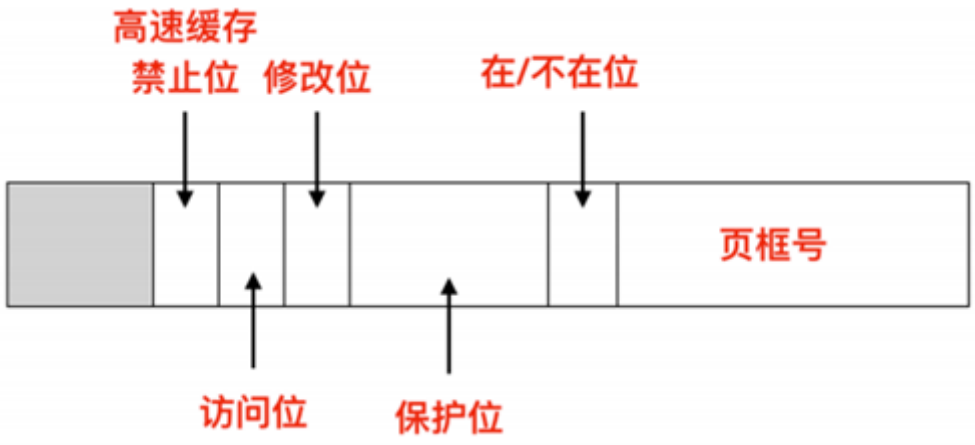
虚拟内存的基本思想：每个程序都有自己的地址空间，这个地址空间被划分为多个'页'。每一页都是连续的地址范围。这些页被映射到物理内存，但并不是所有的页都必须在内存中才能运行程序。当程序用到相应页时，再由操作系统负责将缺失的部分装入物理内存执行。

4.2.1 页表

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，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。

$$物理地址 = 页号 * 页大小$$

页表项的结构



不同计算机的页表项可能不同，但是一般来说都是 32 位的。

- 页框号(Page frame number)：页表到页框最重要的一步操作就是要把此值映射过去
- `在/不在`位：表明该页是否在内存中可以直接使用，如果不在，方位该页会引发缺页异常
- 保护位(Protection)：告诉我们哪一种访问是允许的，如：0 表示可读可写，1 表示的是只读。
- 修改位(Modified)：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。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
- 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。
- 高速缓存禁止位：通过这一位可以禁用高速缓存。

4.3 页面置换算法

4.3.1 最佳页面置换算法

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。

4.3.2 最近未使用页面置换算法

4.3.3 先进先出页面置换算法

4.3.4 时钟置换算法

4.3.5 最近最少使用页面置换算法

5. 文件系统

5.1 文件

文件是一种抽象机制，它提供了一种方式用来存储信息和读取信息。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。

文件结构

文件的构造有多种方式，常用的三种构造方式为：1.字节序列（文件是具有固定长度记录的序列） 2.记录序列 3.树（文件由一颗记录树构成，每个记录树都在记录中的固定位置包含一个`key`字段，以便于快速查找）

文件访问

序列访问：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。

随机访问文件：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。

文件操作

- Create：创建文件
- Delete：删除文件，释放内存空间
- Open：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。
- Close：关闭文件：关闭文件以释放表空间。
- Read：读取文件数据
- Write：向文件写入数据
- append：向文件追加数据
- seek：指定从何处开始获取数据
- get attributes：进程运行时通常需要读取文件属性
- 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
- rename：用户可以自己更改已有文件的名字

5.2 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

一级目录系统

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)`

层次目录系统

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。

路径名

绝对路径、相对路径：略

5.3 文件系统实现

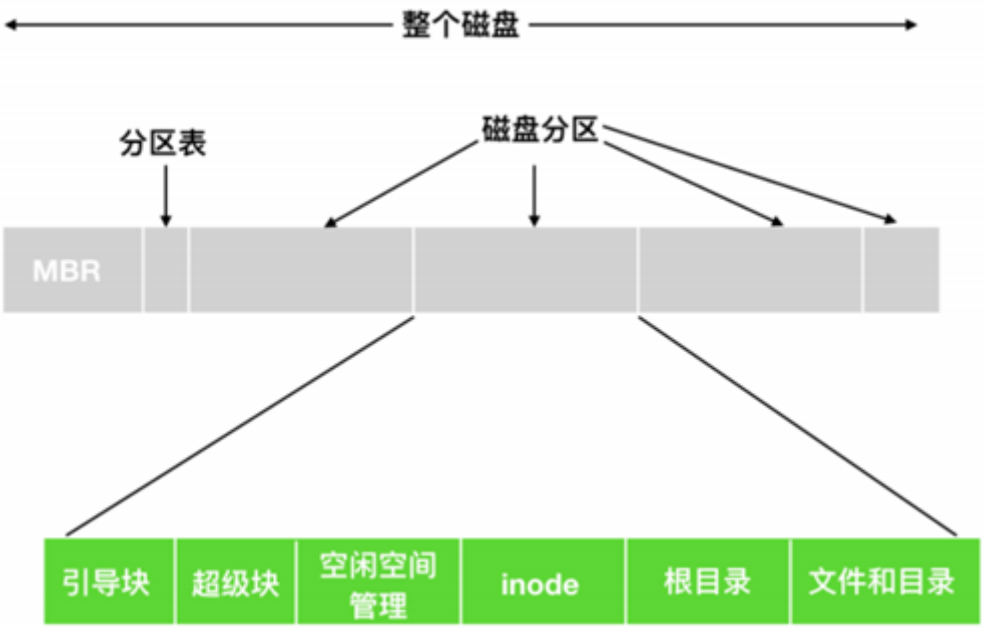
5.3.1 文件系统的布局

文件系统存储在磁盘中，通常磁盘能够划分出一到多个分区，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。

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。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。



超级块

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

文件系统中 `空闲块` 的信息，可以用位图或者指针链表的形式给出。

inode

inode(index node)，称作索引节点，是一个数组的结构，每个文件有一个 inode，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。包含了 模式/权限（保护）、所有者 ID、组 ID、文件大小、上次访问时间、最后修改时间、inode 上次修改时间等信息。

根目录

存放文件系统目录树的根部

文件和目录

存放了其他所有的目录和文件。

5.3.2 文件的实现

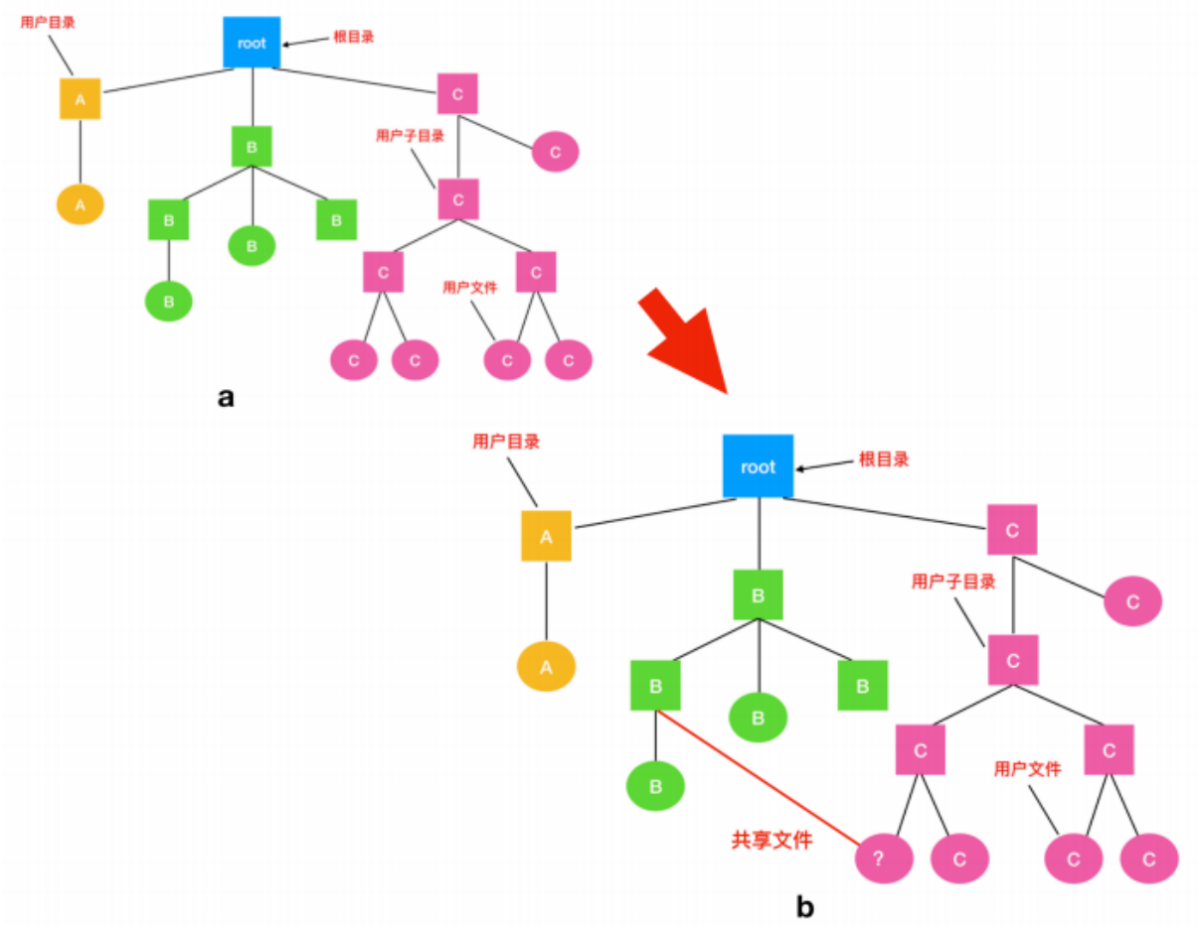
重点在于记录各个文件分别用到了哪些磁盘块。分配背后的主要思想是`有效利用文件空间`和`快速访问文件`，主要分配方案有如下三种：

- 连续分配：把每个文件作为一连串连续数据块存储在磁盘上；缺点：容易产生大量“碎片”
- 链表分配：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；缺点：难以进行随机访问；指针会占用一些字节，导致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，影响读写磁盘。
- 索引分配：使用内存中的表来进行链式分配，即取出每个磁盘块的指针字，把它们放在内存的一个表（文件分配表）中。

5.3.3 目录的实现与文件共享

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。**

文件共享：



5.4 文件系统管理和优化

5.4.1 磁盘空间管理

文件通常存在磁盘中，因此涉及到如何管理磁盘空间。主要有`分段管理`、`分页管理`

段式：按逻辑划分存储空间的方式，磁盘被划分为多个大小不一、连续的段。

页式：将存储空间划分为固定大小的小块（页），然后将文件分割成大小相等的页来存储。

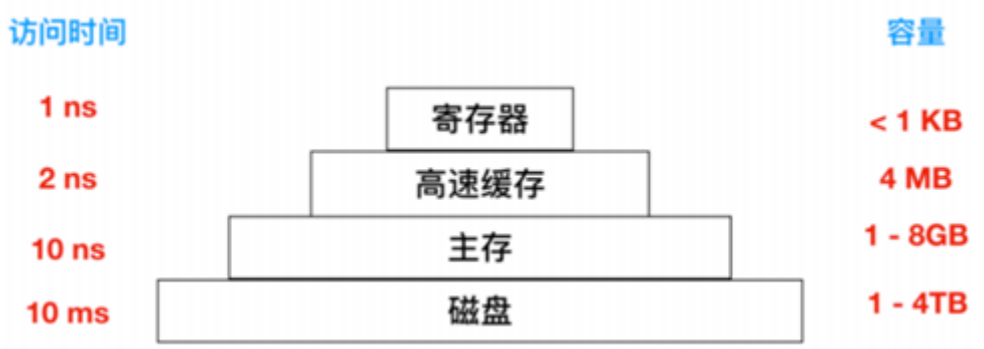
注意：段页式是（虚拟）内存管理，不是磁盘管理。内存管理需要高效的虚拟地址转换、动态分配和小粒度的内存管理，因此采用了**段页式管理**；而文件系统需要优化持久化存储、大容量数据的管理，并减少磁盘碎片，通常通过 **块/簇** 分配方式来提高性能和简化管理。因此，文件系统没有采用段页式管理技术。

把文件分为固定大小的块来存储，如果分配的块太大会浪费空间；分配的块太小会浪费时间。

一旦指定了块大小，就需要记录空闲块，有两种方法：1.磁盘块链表 2. 位图

5.4.2 文件系统性能

访问磁盘的效率要比内存慢的多，所以磁盘优化是很有必要的。



- 高速缓存：块高速缓存(block cache)、缓冲区高速缓存(buffer cache)。
- 块提前读：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。
- 减少磁盘臂运动：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。
- 磁盘碎片整理 （固态硬盘不受磁盘碎片的影响）

6. I/O管理

6.1 I/O 设备

I/O 设备：输入/输出设备，分为块设备(block devices) 和 字符设备(character devices)。

- 块设备：存储固定大小块信息（硬盘、蓝光光盘、USB 盘）
- 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（打印机、网络设备、鼠标等）。

设备控制器

设备控制器是处理 CPU 传入和传出信号的系统，每个设备控制器都会有一个应用程序（驱动程序）与之对应。

CPU与设备控制器内的寄存器通信，以控制设备工作，通信方式有：

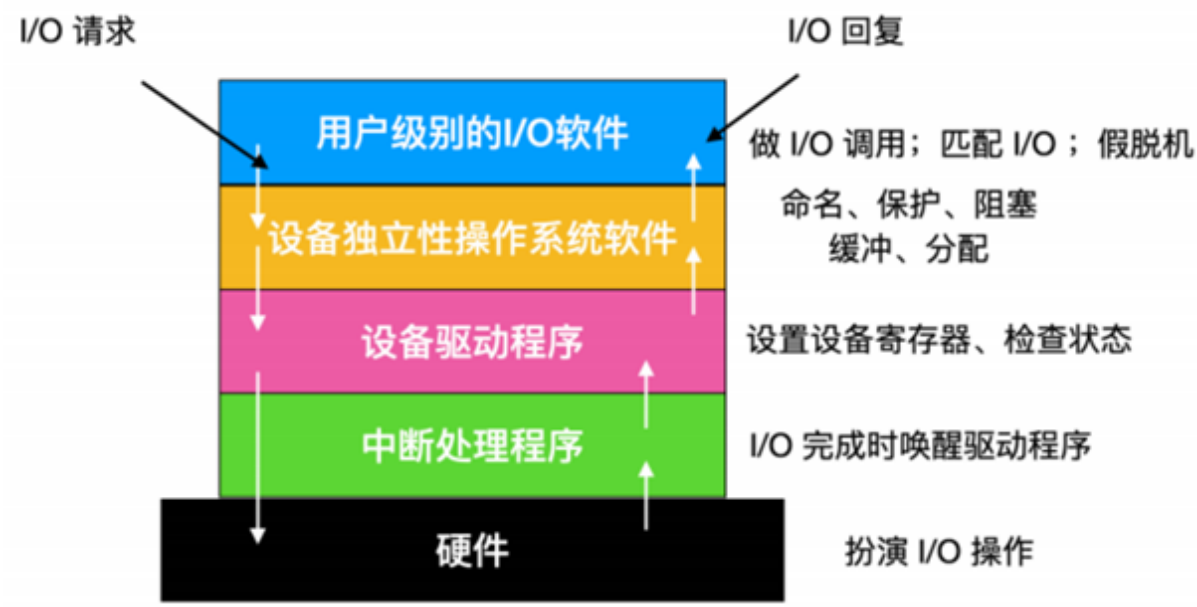
- 每个控制寄存器都被分配一个 `I/O 端口(I/O port)` 号，这是一个 8 位或 16 位的整数。以便普通用户程序无法访问它（只有操作系统可以访问）。
- 将**所有控制寄存器映射到内存空间**

6.2 I/O软件

I/O 软件设计有很多目标

1. 设备独立性：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。
2. 错误处理：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，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。
3. 同步和异步传输：同步传输中数据通常以块或帧的形式发送，异步传输中，数据通常以字节或者字符的形式发送。
4. 缓冲：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。
5. 共享与独占：有些 I/O 设备能够被许多用户共同使用。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。

6.3 I/O 层次结构



6.3.1 中断处理程序

中断处理程序是最靠近硬件的一层，处理硬件中断、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：

- 信号量实现
- 管程实现
- 发送消息

6.3.2 设备驱动程序

提供 I/O 设备到设备控制器转换的过程，需要进行接收和识别命令、进行数据交换、硬件地址识别、对设备数据进行差错检测

6.3.3 与设备无关的 I/O 软件

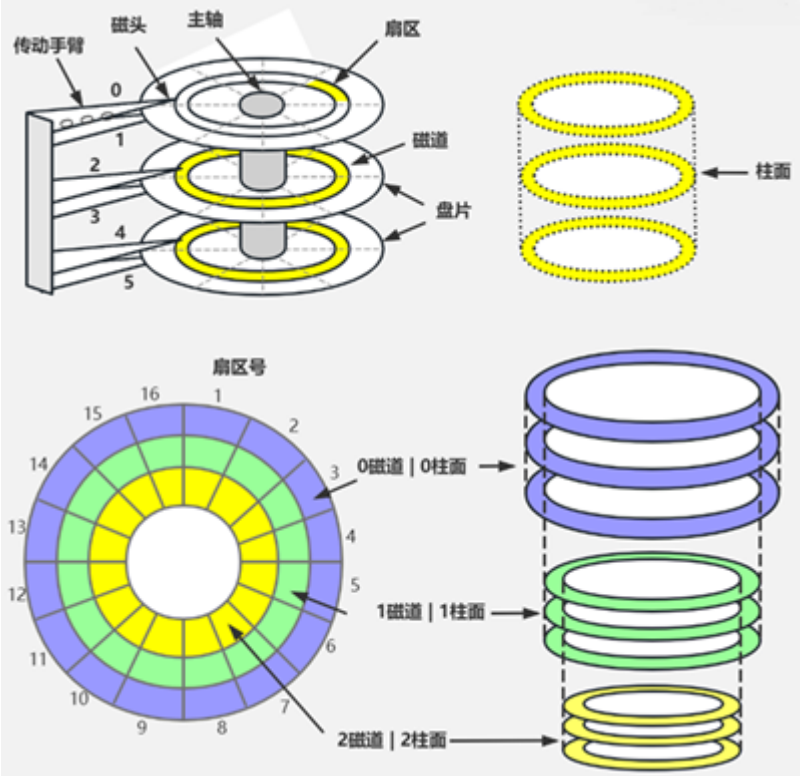
对所有设备执行公共的 I/O 功能，并且向用户层软件提供一个统一的接口。如缓冲、错误报告、设备驱动程序统一接口、分配和释放设备、提供设备无关的块大小。

6.3.4 用户级别的 I/O 软件

少量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。

7. 磁盘

7.1 磁盘硬件结构



7.2 RAID

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。

7.3 磁盘臂调度算法

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（对总时间的影响最大）
- 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

降低寻道时间的算法：

- 先来先服务

- 最短路径优先（谁距离我近先找谁）
- 电梯算法（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。）

7.4 错误处理

1. 在控制器中进行处理：将备用扇区之一重新映射；所有的扇区都向上移动一个扇区
2. 在操作系统层面进行处理

8. 死锁问题

8.1 资源

大部分的死锁都和资源有关，需要排他性使用的对象称为资源。资源主要分为 可抢占资源和不可抢占资源。

8.2 死锁

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。

死锁发生的条件：

1. 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
2. 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
3. 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，它只能由占有它的其它进程释放
4. 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。

8.3 死锁检测与恢复

死锁检测

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。

死锁恢复

- 通过抢占进行恢复
- 通过回滚进行恢复
- 杀死进程恢复

8.4 死锁避免-银行家算法

若在某一时刻，系统能按某种进程顺序，如$\{P_1,P_2,\cdots,P_n\}$，为每个进程分配其所需的资源，直至最大需求，使每个进程均可顺利完成，则称此时系统的状态为安全状态，称这样的一个进程序列$\{P_1,P_2,\cdots,P_n\}$为安全序列。安全序列的实质是：序列中的每一个进程$P_i(i=1,2,\cdots,n)$到运行完成尚需的资源量不超过系统当前剩余的资源量与所有在序列中排在它前面的进程当前所占有的资源量之和。

若在某一时刻，系统中不存在一个安全序列，则称系统处于不安全状态。

银行家算法的实质在于：要设法保证系统动态分配资源后不进入不安全状态，以避免可能产生的死锁。

每当进程提出资源请求且系统的资源能够满足该请求时，系统将判断满足此次资源请求后系统状态是否安全，如果判断结果为安全，则给该进程分配资源，否则不分配资源，申请资源的进程将阻塞。

8.5 其他

通信死锁

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。但该死锁使用 超时 进行避免。

活锁

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

对于进程来讲，最重要的就是资源，如果一直没有获得资源，那么进程会产生饥饿，这些进程会永远得不到服务。