

操作系统

MRL Liu

2022 年 01 月 22 日

本文以《现代操作系统（原书第3版）》、小林 Coding《图解系统》为参考，结合 Linux 操作系统介绍操作系统领域面试相关的知识，主要分为内存管理、进程线程、文件系统三个部分，深入研究可参考其他资料。

第 1 章 操作系统基础

一、计算机硬件组成

操作系统的设计与运行和硬件资源密切相关，为此本小节简单梳理下计算机中的常见硬件设备。

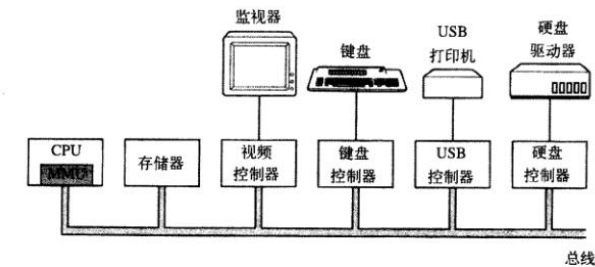
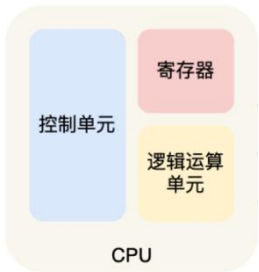


图1-6 简单个人计算机中的一些部件

1、中央处理器

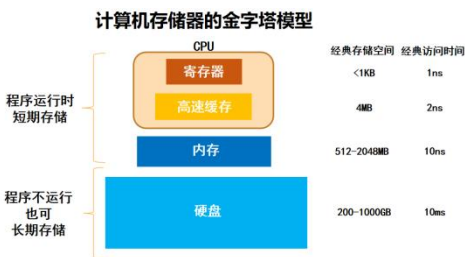
中央处理器（CPU）是计算机的大脑，负责从内存中取出指令并执行。不同产商的 CPU 对应不同的可执行的指令集。CPU 中有一些寄存器用来保存关键变量和临时数据，如程序计数器、堆栈指针、程序状态字（PSW）、存储器管理单元（MMU）。目前的 CPU 正在具有更多的晶体管和缓存，以及多个 CPU 处理芯片，即多核 CPU，使 CPU 从物理上支持操作系统的多进程和多线程。

CPU 的 32 位或 64 位是其位宽属性，达标了 CPU 一次可以计算多少个字节，计算的数值可以有多大。



2、存储器

存储器分很多类，性能好的存储器价格昂贵而且容量小，性能差的存储器价格便宜且容量大，为了满足不同的需求，计算机中使用不同的存储层次，存储器的金字塔如下：



其中寄存器和高速缓存大部分在 CPU 中，网卡等一些自治单元也存在一些高速缓存。内存是主存（也叫随机访

问存储器，Random Access Memory，简称 **RAM**），高速缓存中无法存储的数据首选内存；**硬盘**（也叫**磁盘**）是长期存储文件的硬件设备。

3、I/O 设备

I/O 设备一般分为两个部分：**设备**和**设备驱动程序**。**设备驱动程序**必须装载在操作系统中，**并在内核态运行**，I/O 设备中一般也有少量用于通信的寄存器。**要想访问 I/O 设备，就需要操作系统切换到内核态。**

4、总线

操作系统必须了解计算机内部的所有**总线**的配置和管理（本小节不要求读者记忆，仅理解即可）。总体来讲，计算机内部的总线可以分为以下 3 种：

- （1）地址总线：用于指定 CPU 将要操作的内存地址
- （2）数据总线：用于读写内存的数据
- （3）控制总线：用于发送和接收信号，比如中断、设备复位等信号。

如下是一张图：

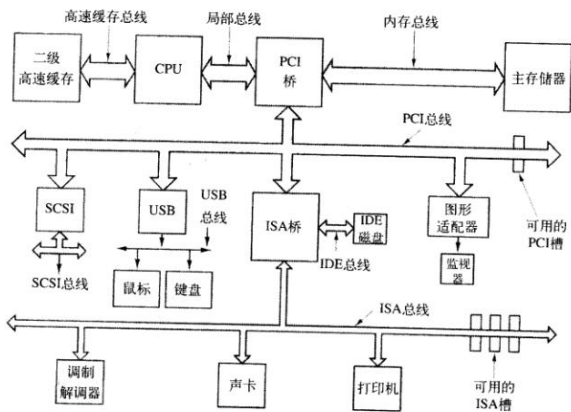


图1-12 大型Pentium系统的结构

CPU 通过局部总线与 PCI 桥连接，通过高速缓存总线与 L2 高速缓存连接，PCI 桥通过内存总线与主存储器连接。PCT（Peripheral Component Interconnect）总线，用于连接高速 I/O 设备，速度高达 528MB/s。SCSI（Small Computer System Interface）总线，用来连接高速硬盘、扫描仪等设备，速度高达 320MB/s。IDE（Integrate Drive Electronics）总线，用于连接磁盘。USB（Universal Serial Bus）总线，用来连接所有低速 I/O 设备，例如键盘、鼠标。ISA（Industry Standard Architecture）总线，用于连接老式 I/O 设备，速度高达 16.67MB/s

二、启动计算机和程序执行

1、计算机启动过程

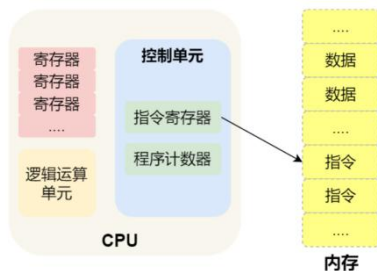
当计算机接通电源后，**CPU** 启动，其内部的 PC 寄存器的值被强制初始化为 **BIOS 程序**的入口地址，然后 CPU 将一个被提前固化在主板上的 BIOS 芯片上的 BIOS 程序加载到内存中，这就是所有计算机运行的第一个程序 **BIOS(Basic Input Output System，基本输入输出系统)**，

BIOS 程序首先帮助计算机进行硬件自检和初始化，检查本机是否存在软盘、U 盘、光盘等外部存储器，**有则尝试从外部存储器启动，没有则从硬盘启动**。硬盘的第一个分区默认都是**启动区**，BIOS 将硬盘中启动区的数据加载到内存中，启动区中的数据引导 CPU 去硬盘的相应位置加载**操作系统的内核代码指令**，此时，**BIOS 程序结束，操作系统程序开始运行。**

操作系统启动后，首先从 BIOS 获得了本机**硬件设备**的配置信息，然后尝试初始化他们的**设备驱动程序**，初始化成功化继续创建更多的**守护进程**，最后在终端上显示 **GUI 程序**的界面。

2、程序执行的过程

程序本质上就是计算机指令的集合，程序的运行本质就是 CPU 依次执行位于内存中的程序中的每一条指令。



CPU 先读取「**程序计数器**」的内存地址，从内存里面将对应的程序指令读取到指令寄存器中，若是计算指令交给逻辑运算单元运算，若是存储指令交付控制单元执行，然后程序计数器自增读取下一条指令，这个过程不断循环直至程序执行结束，这个过程叫做 **CPU 的指令周期**。具体如下：

第 1 步，CPU 读取「程序计数器」的值（指令的内存地址），然后 CPU 的「控制单元」操作「地址总线」指定需要访问的内存地址，接着通知内存设备准备数据，数据准备好后通过「数据总线」将指令数据传给 CPU，CPU 收到内存传来的数据后，将这个指令数据存到「指令寄存器」。

第 2 步，CPU 分析「指令寄存器」中的指令，确定指令的类型和参数，**如果是计算类型的指令，就把指令交给「逻辑运算单元」运算；如果是存储类型的指令，则交由「控制单元」执行；**

第 3 步，CPU 执行完指令后，「程序计数器」的值自增，表示指向下一条指令。自增的大小由 CPU 的位宽决定，如 32 位的 CPU，指令是 4 个字节，需要 4 个内存地址存放，因此「程序计数器」的值会自增 4。

三、操作系统的层次结构

操作系统的定义有很多种，但总体来说，它扮演着这样一个角色：

操作系统是介于硬件资源和应用程序之间的一个系统软件，它可以管理硬件资源，同时也可以管理应用程序。

所以总结出操作系统的主要功能有两点：

(1) **硬件资源管理**。计算机必要的硬件资源有 CPU、内存、硬盘和 I/O 设备。这些硬件资源是有限的，所以必须进行合理地资源分配和资源回收。

(2) **应用程序管理**。操作系统为应用程序提供硬件资源的服务，同时也要管理应用程序，即控制进程的生命周期、资源分配等。

目前最成熟的计算机操作系统 Linux 和 Windows 在安装后也会自带一些标准的应用程序，如文本编辑器、程序编译器等，这些应用程序位于操作系统内核之上。**操作系统的内核是操作系统最核心的程序部分**，用来控制硬件资源、进程管理等，一般的用户程序只能调用内核提供的接口来访问硬件资源，接受内核程序的控制。

1、Linux 系统提供的三种接口

Linux 系统的层次结构如图，Linux 操作系统位于硬件之上，对上层提供一些**系统调用接口**，这些**系统调用接口**被封装成**标准库函数**，程序员可以在这些库函数基础上开发应用；同时基于这些标准库函数，Linux 系统为普通用户提供了一些基础的**标准实用程序**，例如 shell、编辑器、编译器等。综上所述，Linux 系统提供了三类接口：**系统调用接口、库函数接口和用户接口**。



2、操作系统的两种运行模式

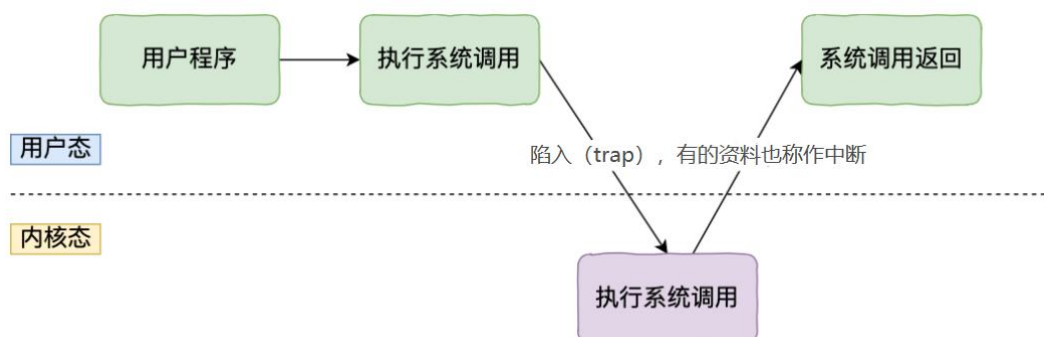
操作系统的内核具有很高的权限，而其上的应用程序却权限很好，因此多数计算机的操作系统将内存划分为两个区域：内核空间和用户空间。

不同内存区域上运行的代码分别对应操作系统的两种运行模式：**内核态**和**用户态**，其本质也是 CPU 的 2 种运行模式，CPU 的内部寄存器 **PSW** 中有一个二进制位可以控制这两种运行模式的切换。

在**内核态**下，CPU 可以执行整个指令集中的每一条指令，使用硬件的所有功能，此时的操作系统便在**内核态**下运行，可以访问所有硬件资源。

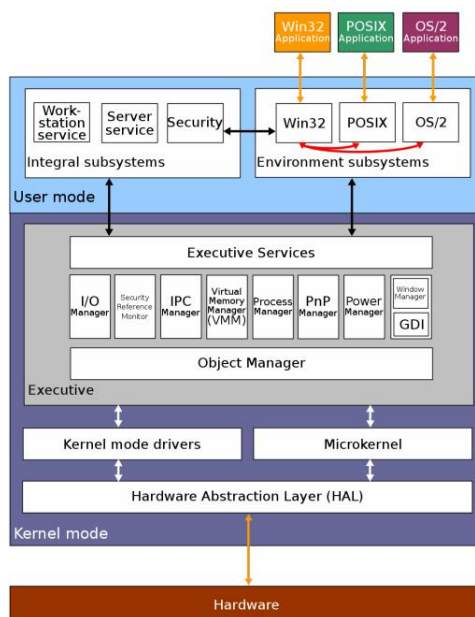
在**用户态**下，CPU 只运行执行整个指令集的一个子集，使用硬件的部分功能，此时的操作系统便在**用户态**下运行，只能访问部分 CPU 等资源，不能访问 I/O 设备。

用户程序只能在用户空间访问一个局部的内存空间，而内核程序的代码可以访问所有空间的代码。当用户程序使用**系统调用（system call）**调用内核代码，本质上也就是用户空间的代码需要访问内核空间，这个时候就是发生了**陷入（trap）**：TRAP 指令把 CPU 的**用户态**切换成**内核态**，然后启用操作系统的**内核态程序**，系统调用结束后切换回**用户态**。



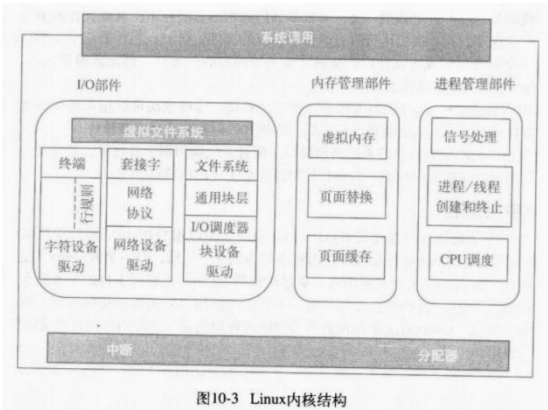
四、Linux 系统内核

不同的操作系统的内核的设计理念也不同。例如根据拥有最高权限的基础功能程序的多少，可以将内核分为宏内核、微内核和混合型内核等。**Windows** 系统的内核是混合型内核，如下：



Linux 操作系统的内核的设计理念是多任务（Mutitask）、对称多处理（SMP）、可执行文件链接格式（ELF）和宏内核（Monolithic Kernel）。其中 SMP 是指每个 CPU 核心的地位相等，不会有某个 CPU 单独服务某类程序；ELF 可以理解为程序编译汇编生成的目标代码需要通过链接器形成可执行文件。

- 这里我们只需要着重关注 Linux 内核，其可以分为三个主要部分：
- （1）**内存管理部件**：维护**虚拟内存**到**物理内存**的映射、负责维护（最近访问的）**页面缓存**和选择合适的**页面置换算法**。
 - （2）**进程管理部件**：使用进程调度算法（进程调度器）进行进程的创建和终止；处理各种**信号**（事件）；提供进程通信机制等。
 - （3）**虚拟文件系统**：初始化各种硬盘等存储设备的**文件系统**；所有的 **I/O 操作**都通过**设备驱动程序**封装为对一个文件的读写操作。
- 注意，有的资料会将 **I/O 系统**单独作为一部分。



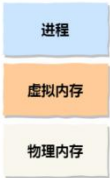
第 2 章 内存管理

内存（主存储器，RAM）是计算机中一种需要认真管理的重要资源，其用来在程序运行时存放数据。和硬盘中的文件不一样，内存中的数据是暂时的，当程序结束时，该程序对应的内存数据就会被释放。

一、虚拟地址

对于计算机来讲，任何类型的存储器的存储空间在计算机看来都是一段连续线性的「**物理地址**」。单片机是没有操作系统的，程序员编写的程序需要借助工具**烧录**进单片机的**存储器**，单片机才能运行程序。单片机的**CPU** 直接操作内存的「**物理地址**」，这种单片机几乎不能同时运行两个以上的程序。

操作系统的程序要尽可能避免访问绝对的「**物理地址**」，所以操作系统为创建的每个进程分配一套独立的「**虚拟地址**」。基于操作系统的所有进程中的程序可以访问的都是「**虚拟地址**」，「**虚拟地址**」在 CPU 执行指令时映射到「**物理地址**」。（请注意，这里讲的**物理地址**和**虚拟地址**都是内存管理的概念，是**内存**（主存储器，RAM），硬盘上一般不认为有虚拟地址）。

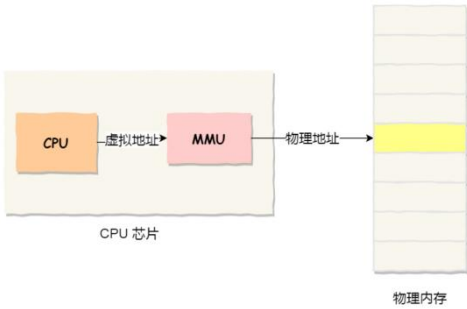


为什么不直接让所有程序使用物理内存而使用虚拟内存呢？

一句话，为了进行内存管理。详细讲就是，（1）**方便多道程序设计**。操作系统必须同时可以运行多个程序，但是再多的物理地址对于多个应用进程来讲都不够分，所以需要要将不同程序中使用频率低的物理地址回收。程序访问的虚拟地址始终不变，虚拟地址映射的物理地址动态调整，这样子虚拟地址和物理地址也不必是一一对应。（2）**安全性**。每个程序都可以直接访问所有的物理地址很不安全，理想的情况是每个程序只能访问特定区域的地址空间。

由「**虚拟地址**」构成的地址空间就是「**虚拟内存**」，相同地，由「**物理地址**」构成的地址空间就是「**物理内存**」。

「**虚拟内存**」比「**物理内存**」要大得多，CPU 中的内存管理单元 **MMU**（Memory Management Unit）会将程序指令访问的「**虚拟地址**」转换为实际的**物理地址**」进行执行。

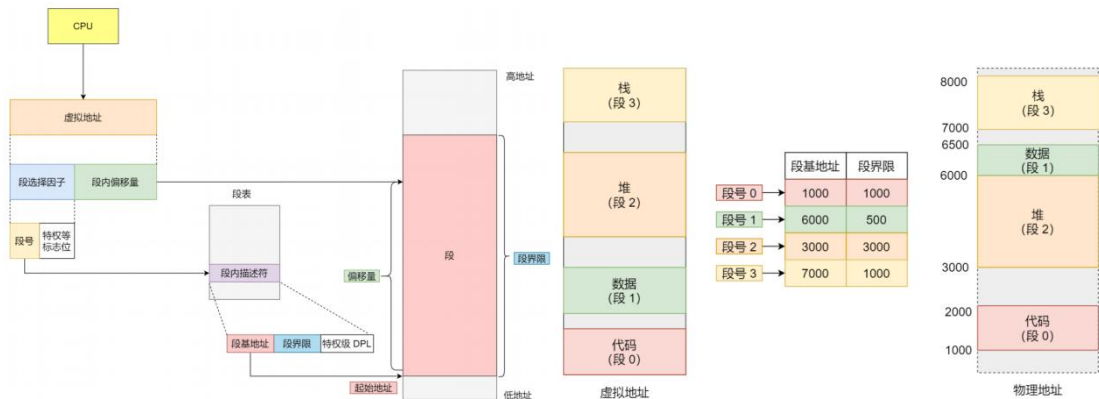


二、内存的分段分页

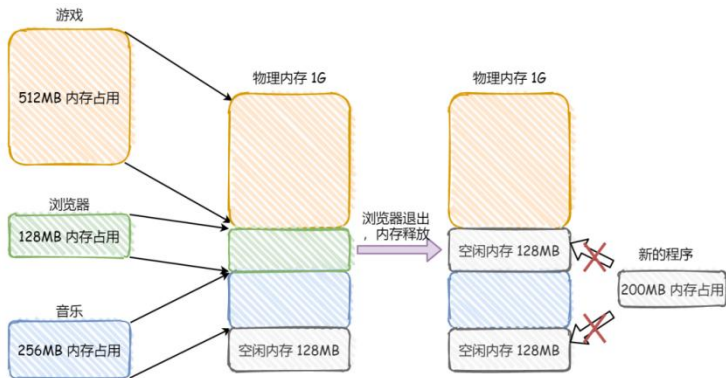
操作系统通过**分段分页**的方式来管理「**虚拟地址**」和「**物理地址**」之间的映射关系。

1、内存分段

分段机制将一个进程中的所有**程序指令**划分为若干个**逻辑分段**（Segmentation）：**代码段**、**数据段**、**堆段**和**栈段**。不同的段有不同的属性。在分段机制中，「**虚拟地址**」和「**物理地址**」之间通过「**段表**」进行映射。操作系统给每一个进程分配的「**虚拟地址**」由「**段选择子**」和「**段内偏移量**」组成。「**虚拟地址**」通过「**段表**」与物理地址进行映射的，分段机制会把程序的虚拟地址分成 4 个段，每个段在段表中有一个项，在这一项找到段的基地址，再加上偏移量，于是就能找到物理内存中的地址，如下图：

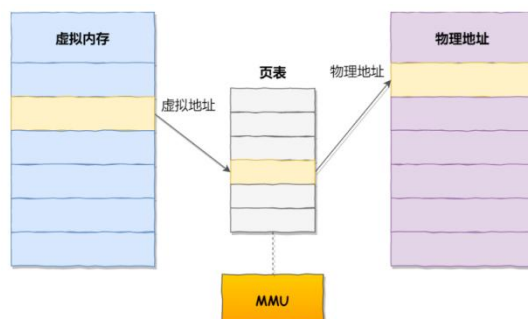


这种分段机制的好处是**可以产生连续的内存空间**，但是存在的显著问题就是**存在内存碎片**和**内存交换效率低**的问题。在多进程的系统中，内存碎片经常产生，内存碎片也分为外部内存碎片和内部内存碎片，解决外部内存碎片的常见方法就是进行内存交换（Swap）：将音乐程序占的 256MB 内存写入硬盘里再分配到已被占用的 512MB 内存下，新的 200MB 程序就有地方了，由于硬盘访问速度慢，如果交换内存的程序过大则效率较低会引起计算机卡顿。



2、内存分页

内存分段可以产生连续的内存空间但是会出现很多内存碎片,为了解决这个问题提出了**内存分页技术(Paging)**。物理内存一般划分为多个页框 (page frame), 虚拟内存也要划分为多个页面 (page), 虚拟内存的页面和物理地址的页框大小一致, Linux 中每一页为 4KB。在分页技术中, 「**虚拟地址**」和「**物理地址**」之间通过「**页表**」进行映射。



页表是存储在**内存**里的, **内存管理单元 (MMU)** 就做将**虚拟内存地址**转换成**物理地址**的工作。而当进程访问的虚拟地址在页表中查不到时, 系统会产生一个**缺页异常**, 进入系统内核空间分配物理内存、更新进程页表, 最后再返回用户空间, 恢复进程的运行。

分页和分段的不同点? 分页技术的优势?

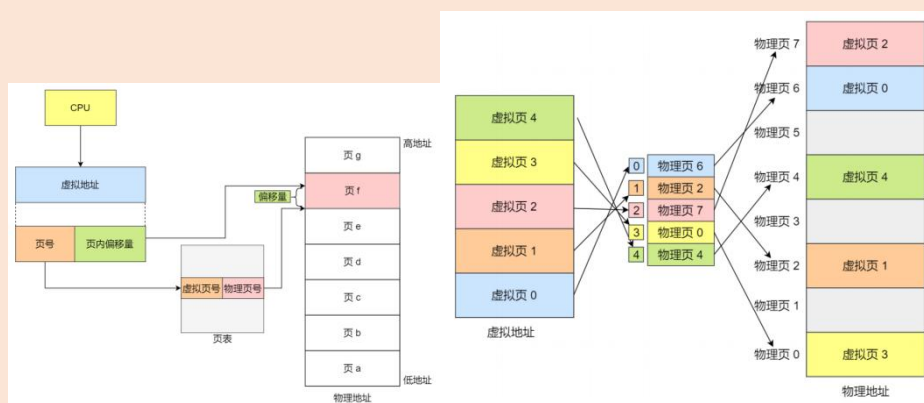
(1) 分页在一开始就将整个内存空间按页划分, 内存的分配和释放都是按照页为单位; 分段则是给每个进程内部按段划分, 所以分页技术不会产生间隙非常小的无法利用的内存, 只有空闲的页; 而分段技术可能由于空闲内存不足而无法分配给新进程。

(2) 分页技术允许空闲内存不足时可以将其他运行的进程中的空闲页或者访问频率少的页释放掉写入硬盘中 (换出, **swap out**), 需要该页的数据时再重新加载 (换入, **swap in**), 这样可以充分利用更多的内存页, 效率提升。

(3) 分页技术允许在加载进程中的程序指令时, 不需要一次性将所有程序加载到物理内存中, 完全可以先建立虚拟内存和物理内存的页映射, 在程序运行中, 需要用到对应虚拟内存页里面的指令和数据时, 再加载到物理内存里面去。

分页技术中如何进行虚拟地址和物理地址的映射?

在分页机制下, 虚拟地址分为两部分, **页号**和**页内偏移**。**页号**作为**页表**的**索引**, 页表包含**物理页**每页所在物理内存的**基地址**, 这个**基地址**与**页内偏移**的组合就形成了**物理内存地址**, 见下图。



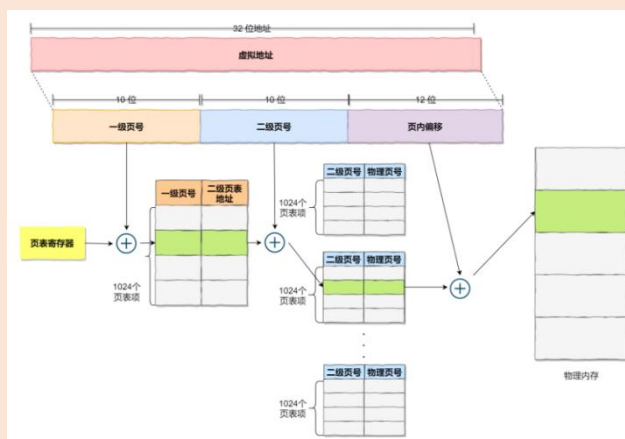
总结一下, 对于一个内存地址转换, 其实就是这样三个步骤:

- 把虚拟内存地址, 切分成页号和偏移量;
- 根据页号, 从页表里面, 查询对应的物理页号;
- 直接拿物理页号, 加上前面的偏移量, 就得到了物理内存地址。

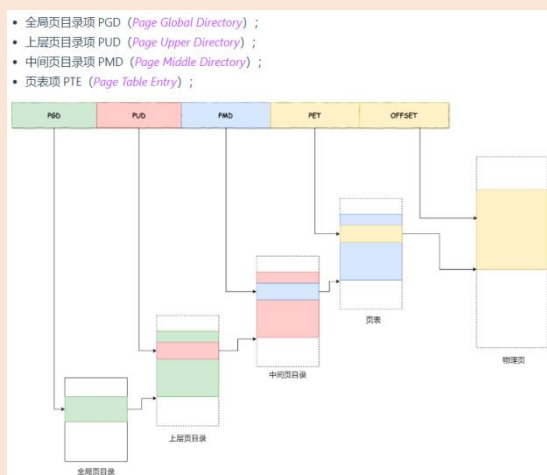
在 Linux 操作系统中, 系统为每个进程默认分配 4GB 的虚拟内存, 其中 1GB 作为内核空间, 3GB 作为用户空间。假设一个页是 4KB 大小, 则 4GB 共有大约 100 万个页, 假设每个页表项使用 4 个字节来存储, 那么 4GB 共需要 4MB

的内存来存储页表。假设有 100 个进程，则需要 400MB 内存来存储页表。

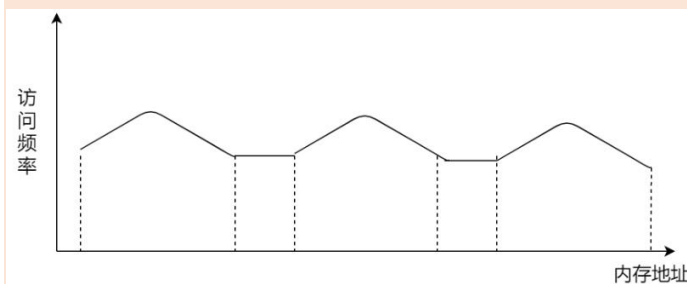
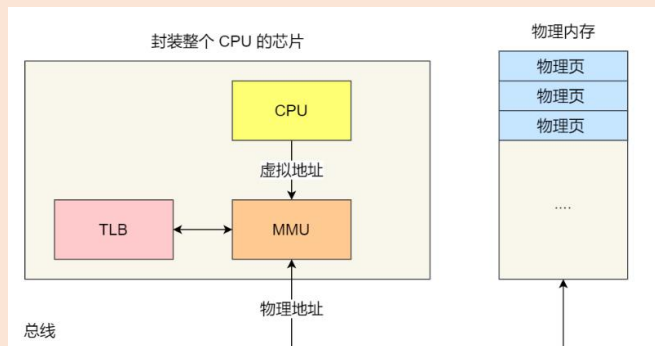
为了减少页表的占用，操作系统进一步采用了多级页表技术。多级页表技术将原有的页表进一步分为固定的 1024 个二级页表，一级页表存储二级页表目录，每个二级页表再包含固定的 1024 个页表项，这样就形成二级分页，如下图：



二级分页后其可以映射的页表项数量并没有大幅度降低 (1024×1024)，但是整个一级页表只有 4KB (4×1024)，二级页表可以在需要时被创建，这样就可以减少页表的内存占用。在 64 位系统中，使用的如下四级目录：



多级页表技术的成功在于一是按需创建的局部性原理，二是用时间换空间，减少了空间但是提升了时间。多级页表虚拟地址到物理地址的转换就多了几道转换的工序，这显然就降低了这俩地址转换的速度，也就是带来了时间上的开销。为了进一步解决这个问题，设计者在 CPU 中加入了一个专门存放程序最常访问的页表项的缓存区，即页表缓存 (Translation Lookaside Buffer, TLB)，也叫做转址旁路缓存、快表等。TCB 的命中率是非常高的，因为程序在一段时间内访问的内存是固定的。



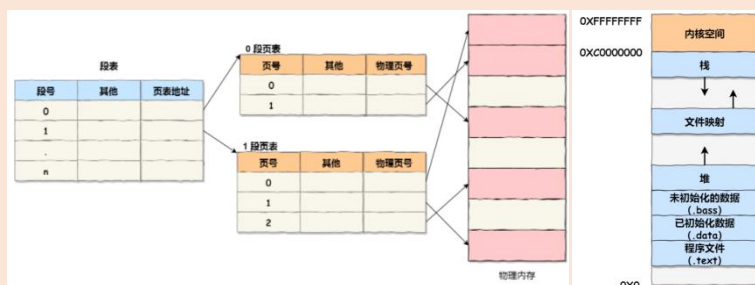
现代操作系统中如何进行内存管理？什么是段页式内存管理？

内存分页技术虽然改进了内存分段技术的不足，但是其并没有取代内存分段，而是和内存分段一起形成了现在操作系统的内存管理机制一段页式内存管理：

整个内存空间按页划分。Linux 操作系统默认为一个进程分配 4GB 的内存空间，然后进程的 4GB 内存空间中按

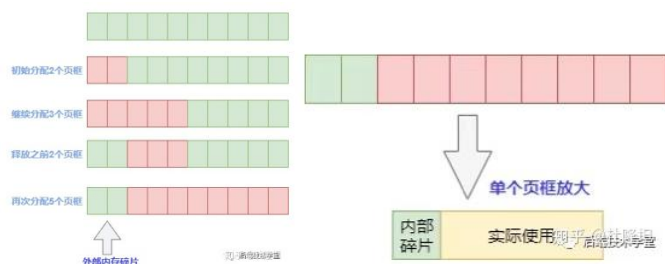
段划分为多个有逻辑意义的段，接着再把每个段包含了多个页，也就是对分段划分出来的连续空间，再划分固定大小的页。这样虚拟地址结构就由段号、段内页号和页内位移三部分组成。

用于段页式地址变换的数据结构是每一个程序一张段表，每个段又建立一张页表，段表中的地址是页表的起始地址，而页表中的地址则为某页的物理页号，如图所示：

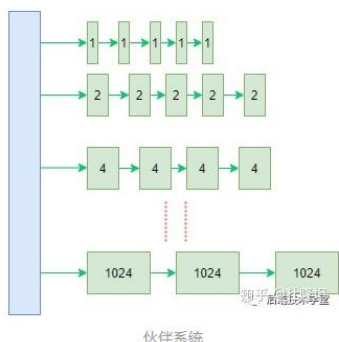


二、页面管理算法

为了允许多个进程分配的虚拟内存可以共享物理内存，操作系统需要监视物理内存的使用情况。Linux 系统中通过分段和分页机制，把虚拟内存划分被页面大小为 4K 的内存页，每一页的大小是固定的。当系统给进程分配虚拟内存时是分配固定数量的内存页，但是进程可能不需要用不完，就会产生很多空闲的内部碎片；在虚拟内存的多次分配回收中也会产生很多外部碎片。



所以操作系统需要一些页面管理算法。Linux 系统使用的是伙伴算法（Buddy System）。伙伴算法的原理也很简单，具体讲就是 Linux 系统将所有物理内存中的空闲页框分为 11 个块链表，每个块链表分别包含大小为 1，2，4，8，16，32，64，128，256，512 和 1024 个连续页框的页框块，所以 1024 的块链表每次可以申请 1024 个连续页框，对应 4MB 大小的连续内存。



假如系统需要申请 4 个页框，但是 4 的块链表没有空闲的页框块，伙伴系统会从 8 的块链表中取出一个长度为 8 的页框块，并将其拆分为 2 个连续 4 个页框块，一个拿去分配，另一个放入 4 的块链表中。系统释放页框的时候会检查，释放的这几个页框前后的页框是否空闲，能否组成下一级长度的块。

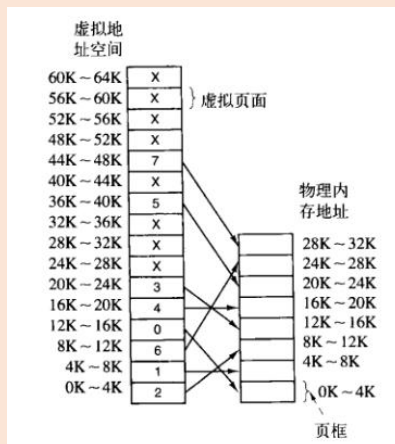
三、页面置换算法

对程序来讲，虚拟地址提供了更大的空间（比物理地址至少大一倍），但是实际执行的物理地址却并没有多，为什么虚拟地址还可以存放程序指令？

简单点说，操作系统给进程分配 4GB 的虚拟内存，并不代表真的有 4GB 空闲的物理内存可以给它用，操作系统只尽量保证在进程执行时其可以运行接近 4GB 的程序，为了达到这个效果，操作系统需要使用页面置换算法将程序使用频率低的数据保存到磁盘中，因为程序在某个时间段访问的内存空间是固定的。

所以一句话，借助了硬盘的空间。虚拟地址远多于内存中的物理地址，显然虚拟地址和物理地址间不可能存在一一对应，操作系统会将使用频率低的物理地址处的数据存入硬盘，这个操作叫做**页面置换**，然后空出来的物理地址就会分配给未映射的虚拟地址，这样内存中有限的物理地址会被尽可能地分配给经常使用的程序指令，从而提高内存的利用率。

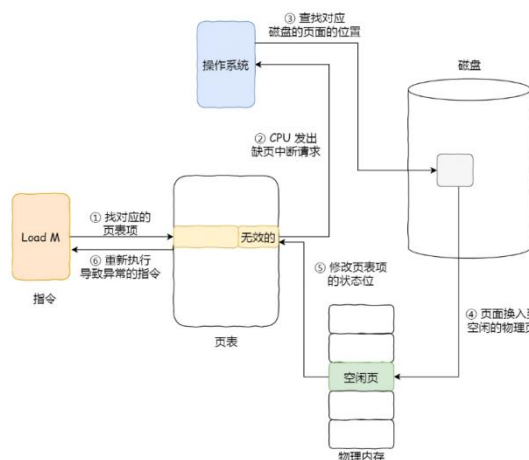
假如一台 16 位地址的计算机有 32KB 的物理内存，地址范围为 0-32KB，，假设这里的每个页框为 4KB，则一共可以分出 8 个**页框**。为了让这台计算机可以运行超过 32KB 的程序，操作系统将 32KB 的物理地址映射为 64KB 的虚拟地址，地址空间为 0-64KB，，每个**页面**为 4KB，则一共可以分出 16 个页面。



程序可以访问的所有内存都是虚拟内存。假如某个程序中的指令（MOV REG,0）要求访问虚拟地址 0，CPU 发送虚拟地址 0 给内存管理单元 MMU，MMU 返回该虚拟地址页面 0（0-4095）映射的物理地址页框 2（8192-12287），则 CPU 的指令变成了 MOV REG, 8192,可以执行。

通过 MMU，可以把 16 个虚拟地址页面映射到 8 个物理地址页框中的任何一个，如果程序访问了一个未映射的虚拟地址页面，MMU 发现该虚拟地址没有映射的物理地址，则发送指令给 CPU 使其陷入到操作系统，这也叫做**缺页中断**（page fault），操作系统程序找到物理地址中一个使用频率较低的页框将其内容写入磁盘，将要访问的虚拟地址页面读到刚才回收的页框，随后修改 MMU 中的映射关系，然后让 CPU 重新执行其指令。

在前文所述中，当发生缺页中断时，操作系统必须在内存中选择一个页面将其换出内存，以便为即将调入的页面腾出空间。缺页中断的处理流程如下：



如果要换出的页面在内存驻留期间被修改过则必须把它协会磁盘已更新该页面在磁盘上的副本。这类算法就是页面置换算法。页面置换算法的功能是，**当出现缺页异常，需调入新页面而内存已满时，选择被置换的物理页面，也就是说选择一个物理页面换出到磁盘，然后把需要访问的页面换入到物理页。**Linux 系统使用的是类似于以下时

钟置换算法。

(a) 最佳页面置换算法 (OPT)

置换在「未来」最长时间不访问的页面,但是实际系统中无法实现,因为程序访问页面时是动态的。我们是无法预知每个页面在「下一次」访问前的等待时间,因此作为实际算法效率衡量标准。

(b) 先进先出置换算法 (FIFO)

将页面以队列形式保存,先进入队列的页面先被置换出去,其性能较差。

(c) 最近最久未使用置换算法 (LRU)

根据页面未被访问时长用升序列表将页面排列,每次将最久未被使用的页面置换出去,其开销较大,实际应用少。

(d) 时钟置换算法 (Lock)

把所有的页面都保存在一个类似钟面的「环形链表」中,页面包含一个访问位。当发生缺页中断时,顺时针遍历页面,如果访问位为 1,将其改为 0,继续遍历,直到访问到访问位为 0 页面,进行置换。



(e) 最不常用算法 (LFU)

记录每个页面访问次数,当发生缺页中断时候,将访问次数最少的页面置换出去,此方法需要对每个页面访问次数统计,额外开销。

四、Linux 的虚拟地址空间

Linux 系统的内存管理主要采用的是内存分页技术。一台 32 位的 Linux 系统通常有 1GB 的**内核空间**。当创建一个新的进程时,系统为其分配 3GB 的虚拟内存作为该进程的**用户空间**。当该进程开始运行时,其执行一个 **exec** 系统调用,该进程的 3GB 的用户空间被重写。进程在用户态下运行时,其所有程序无法访问到 1GB 的内核空间;进程在内核态运行时,其程序可以访问到 1GB 的内核空间。



1、用户空间

进程通常只能访问其用户空间的虚拟地址，只有在执行内陷操作或系统调用时才能访问内核空间。按照访问属性（可读、可写、可执行），进程的用户空间可以分为如下 5 个区域：

代码段：用来存放可执行文件的操作指令，可执行程序在内存中的镜像。代码段需要防止在运行时被非法修改，所以只准许读取操作，它是不可写的。

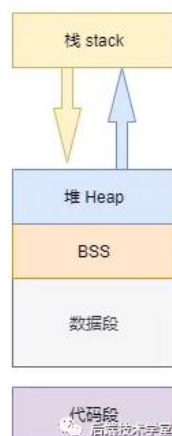
数据段：数据段用来存放可执行文件中已初始化的全局变量，换句话说就是存放程序静态分配的变量和全局变量。

BSS 段：包含了程序中未初始化的全局变量，在内存中 **bss** 段全部置零。

堆 heap：用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 **malloc** 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用 **free** 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。

栈 stack：栈是用户存放程序临时创建的局部变量，也就是函数中定义的变量（但不包括 **static** 声明的变量，**static** 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

上述几种内存区域中数据段、BSS 段、堆通常是被连续存储在内存中，在位置上是连续的，而代码段和栈往往会被独立存放。堆和栈两个区域在 **i386** 体系结构中栈向下扩展、堆向上扩展，相对而生。



2、内核空间

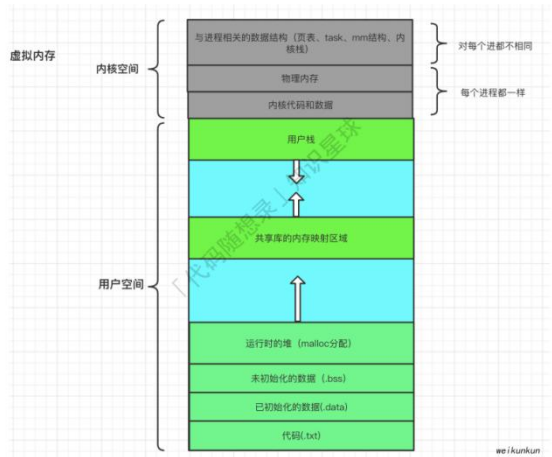
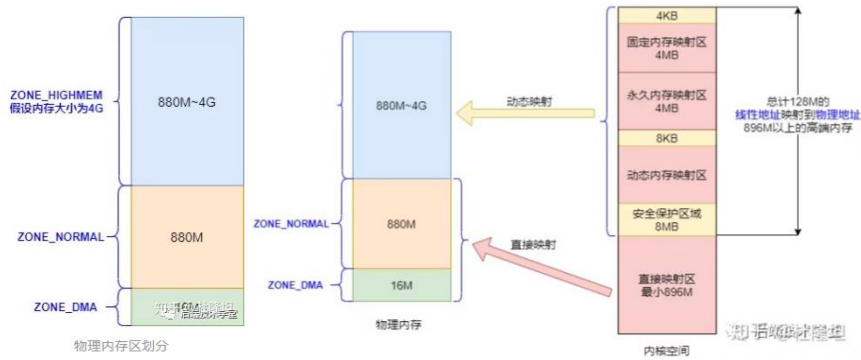
Linux 的内核空间是指高端内存地址空间，总计 **1G** 的容量，包括了内核镜像、物理页面表、驱动程序等运行在

内核空间。Linux 内核会将物理内存分为 3 个管理区，分别是：

ZONE_DMA。DMA 内存区域。包含 0MB~16MB 之间的内存页框，可以由老式基于 ISA 的设备通过 DMA 使用，直接映射到内核的地址空间。

ZONE_NORMAL。普通内存区域。包含 16MB~896MB 之间的内存页框，常规页框，直接映射到内核的地址空间。

ZONE_HIGHMEM。高端内存区域。包含 896MB 以上的内存页框，不进行直接映射，可以通过永久映射和临时映射进行这部分内存页框的访问。



第 3 章 进程与线程

一、进程的概念

什么是进程？进程和程序之间的关系是什么？

程序在运行前是存储在硬盘里的静态文件，运行时才被 **CPU** 加载到内存。**进程本质上就是一个正在运行的程序**，是对运行时程序的封装，其包含了该程序运行的所有资源和操作。简单说，进程就是一个程序的执行流程，内部保存程序运行所需的资源。

CPU、**程序**、**进程**和**地址空间**的关系可以按照以下比喻来理解：

CPU 的每个核是一个厨师；**程序**是保存在存储箱里的**食谱**，其平时一直放在硬盘里，做菜时拿出来放入内存；
程序的各种输入数据就是做菜的**各种原料**；程序运行后可以实现的**各种功能**就是最终做好的**菜品**；
（内存）地址空间是做菜的**厨房**；**进程**就是厨师阅读食谱、获取各种原料并做菜的一**系列动作的总和**，厨师烹饪每一道菜品都必须对应的厨房按照对应的食谱进行。

你知道操作系统的进程控制块/PCB 吗？你知道操作系统如何描述一个进程吗？

Linux 系统中一个进程被分配的「**虚拟内存**」可以分为三段：

- (1) **数据段**：存放全局变量、常数、以及动态分配的数据空间。根据实现的位置不同分为**用户数据段**和**系统数据段**。
- (2) **正文段**：存放程序代码。
- (3) **堆栈段**：存放函数返回地址、函数参数以及局部变量。



用户数据段在**用户空间**实现，系统数据段在**内核空间**实现。每一个进程的**系统数据段**存放在操作系统中的**进程表**，这个系统数据段也叫做**进程控制块 PCB (Process Control Block)**，也是操作系统唯一描述进程的单位。操作系统为每个进程都维护一个 **PCB**，用来描述当前存在的进程的基本情况和状态变化。

PCB 中都存储了什么信息？

(一) 进程标识信息。用来标识当前进程、进程间关系、多线程关系等。

进程标识符 (PID)：Linux 内核通过唯一标识每一个线程对应的任务进程，新创建的进程通常是前一个进程的 PID 加 1，PID 值有最大值 (32767)

父进程标识符 (PPID)：Linux 内核通过复制原有的进程来创建新的进程，为了描述这种创建的父子关系，子进程中会保留有其父进程标识符。

线程组标识符 (TGID)：Linux 内核支持多线程，通过 TGID 来标识每一个多线程组。一个多线程程序对应的进程只有一个主线程，当主线程创建其他线程时，就复制了主线程的 TGID，内核可以通过 TGID 判断该线程属于哪个线程组，即属于哪个进程，getpid() 返回的就是 TGID。

(二) 处理机状态信息保护区。用于保存进程的运行信息。

用户可见寄存器：用户程序可以使用的数据，地址等寄存器

控制和状态寄存器：程序计数器，程序状态字

栈指针：过程调用、系统调用、中断处理和返回时需要用到它

(三) 进程控制信息。

调度和状态信息：用于操作系统调度进程使用

进程间通信信息：为支持进程间与通信相关的各种标识、信号、信件等，这些信息存在接收方的进程控制块中

存储管理信息：包含有指向本进程映像存储空间的数据结构

进程所用资源：说明由进程打开使用的系统资源，如打开的文件等

有关数据结构连接信息：进程可以连接到一个进程队列中，或连接到相关的其他进程的 PCB

Linux 中的 PCB 就是 **task_struct 结构体**，task_struct 也被称为 Linux 的进程描述符。

进程的特点可以总结为：

动态性：可动态的创建和结束进程

并发性：可以被独立的调度和占用处理机并发运行

独立性：不同进程的工作互不影响

制约性：进程间可以进行通信来产生制约

二、进程的状态

1、进程为什么要状态切换

单核的 CPU 在任意时刻只能运行一个程序（一个厨师在任意时刻只能按照一个食谱做一道菜），但是现在的复杂操作系统基本都是**多任务**的，即 CPU 可以同时运行多个程序。

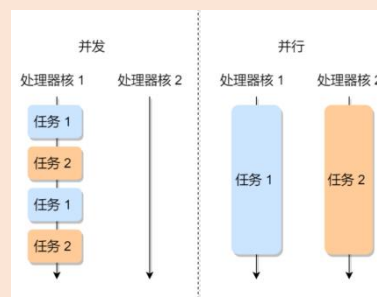
一个厨师在任意时刻只能按照一个食谱进行烹饪，每工作一段时间就切换到另一个食谱。这样子的好处是，对每个菜品来讲，每次它被烹饪时都独占了厨师的精力，对顾客来讲，这些菜品看起来是同时在烹饪的。

同样的原理，**单核 CPU** 在任意时刻只能运行一个程序，所以单核 CPU 在每个**时间片**（由程序计时器计数的一小段时间）内运行一个程序，时间片结束时切换到另一个程序。由于每个时间片足够小，**对每个程序来讲，每次运行都似乎独占了 CPU 的资源；对每个用户来讲，每个程序似乎都可以同时运行。**我们将这种机制称为 **CPU 的并发**。

注意以下 CPU 的并发和并行概念的区别：

CPU 的**单个核心**在**每个时间片**内分别执行多个进程，称为 **CPU 的并发**；

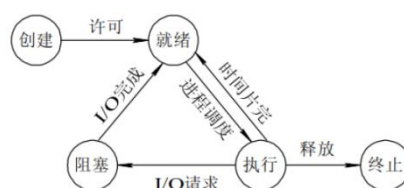
CPU 的**多个核心**在**同一个时间片**内同时执行多个进程，称为 **CPU 的并行**。



我们注意到，对于 CPU 的并发来讲，CPU 从一个进程切换到另一个进程时需要保存原来进程的状态信息（等会还要继续进行该进程）。

2、一般进程的状态切换

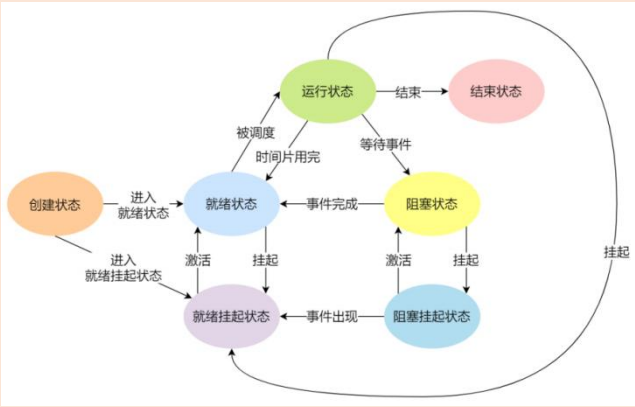
在前文描述中，我们已经知道，同一个**进程**存在多个状态，具体来讲，进程从创建到终止大致分为如下 5 种状态，即**进程的五态模型**：



状态类型	状态特点	状态切换条件
创建态	创建进程，调度程序为其分配内存空间等资源	如果成功获取除处理机以外的其他资源则自动进入就绪态。
就绪态	因为其他进程正在运行而暂时停止。	等待分配处理机资源，得到后可立即运行
执行态 (运行态)	该进程正在运行，处于独占用一个 CPU 核心的时间片	时间片用尽后自动进入就绪状态
阻塞态	该进程正在等待某一事件发生而暂时停止运行。	比如等待客户端连接或用户输入。
终止态	进程结束，资源被回收。	可能是任务结束或者遇到已知问题而主动结束也可能是遇到严重未知错误或被其他进程杀死而被动结束

请介绍下进程的挂起？/挂起和阻塞有什么区别？/进程的七态模型？

为了合理且充分的利用系统资源，当内存不足时，操作系统会把一个进程从内存转到外存，这个对进程的重要操作就是**挂起**。进程挂起就是进程在挂起状态时，意味着进程没有占用内存空间，处在挂起状态的进程映射在磁盘上。增加挂起操作后，进程的就绪状态分为活动就绪态（激活）和静止就绪态（挂起），阻塞状态分为活动阻塞态（激活）和静止阻塞态（挂起），就得到了**进程的七态模型**：



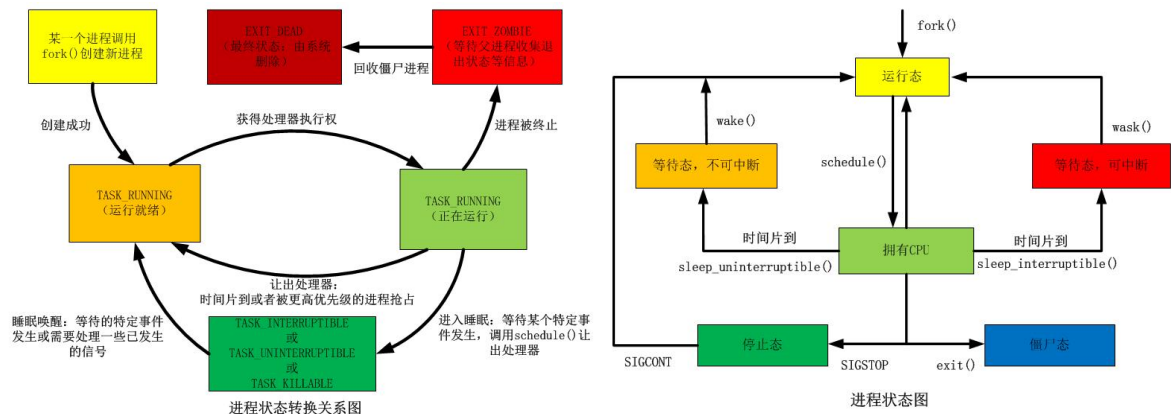
注意，在所有状态的切换中，**只有就绪态和运行态是双向转换的**，其他都是单向转换。

3、Linux 进程的状态切换

Linux 中的进程运行状态有以下几种主要状态：

- A、**运行状态 (TASK_RUNNING)**：程序当前正在运行，或者在运行队列中等待调度。
- B、**可中断的阻塞状态（等待状态） (TASK_INTERRUPTIBLE)**：进程处于阻塞（睡眠）状态，正在等待某些事件的发生或者能够占用某些资源。处在这种状态下的进程可以被信号中断。接收到信号或被显示的唤醒呼叫（如调用 wake_up 系列宏：wake_up、wake_up_interruptible 等）唤醒之后，进程将转变为运行（TASK_RUNNING）状态。
- C、**不可中断的阻塞状态（等待状态） (TASK_UNINTERRUPTIBLE)**：该状态类似于可中断的阻塞状态（TASK_INTERRUPTIBLE），但是它不会处理信号，把信号传递到这种状态下的进程不能改变它的状态。在一些特定的情况下（进程必须等待，知道某些不能被中断的事件发生），这种状态是很有用的。它不能被信号唤醒，只有它所等待的事件发生时，进程才能被显示唤醒呼叫唤醒。
- D、**暂停状态 (TASK_STOPPED)**：进程的执行被暂停，当进程收到 SIGSTOP、SIGTSTP、SIGTIN、SIGTTOU 等信号，就会进入暂停状态。
- E、**僵尸状态 (TASK_ZOMBIE)**：子进程运行结束，父进程未退出，并且未使用 wait 函数族（如使用 waitpid（）函数）等系统调用来回收子进程的退出状态。处在该状态下的子进程已经放弃了几乎所有的内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其父进程收集。
- F、**消亡状态（死亡状态） (TASK_DEAD)**：这是最终状态，这是一个已终止的进程，但还在内核的进程表中占有一个 task_struct 结构。父进程调用 wait 函数族回收之后，子进程由系统彻底删除，不可见。

它们之间的转换关系和调用各个函数的状态如下：

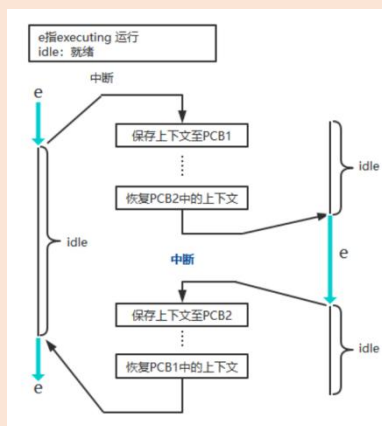


4、进程的上下文切换

什么是上下文切换？

上下文切换指的是操作系统停止当前运行进程（从运行状态改变成其它状态）并且调度其它进程（就绪态转变成运行状态）。操作系统必须在切换之前存储许多部分的进程上下文，必须能够在之后恢复他们，所以进程不能显示它曾经被暂停过，同时切换上下文这个过程必须快速，因为上下文切换操作是非常频繁的。那上下文指的是什么呢？指的是任务所有共享资源的工作现场，每一个共享资源都有一个工作现场，包括用于处理函数调用、局部变量分配以及工作现场保护的栈顶指针，和用于指令执行等功能的各种寄存器。

注意：这里所说的进程切换导致上下文切换其实不太准确，准确的说应该是任务的切换导致上下文切换，这里的任务可以是进程也可以是线程，准确的说线程才是 CPU 调度的基本单位，但是因为各个资料都这么解释上下文切换，所以上面也暂时这么介绍，只要读者心里有这个概念就好。



什么是中断？中断发生后操作系统最底层发生了什么？

一个进程在执行过程中可能会被中断无数次，从而发生状态变化，但是每次中断结束后，被中断的进程都要返回到与中断发生前完全相同的状态，这其中就要使用中断向量，中断向量是指中断服务程序的入口地址。中断发生后，其最底层发生如下：

- 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值；
- 4) 汇编语言过程设置新的堆栈；
- 5) C 中断服务例程运行（典型的读和缓冲输入）；
- 6) 调度程序决定下一个将运行的进程；
- 7) C 过程返回到汇编代码；
- 8) 汇编语言过程开始运行新的当前进程。

三、Linux 进程的创建、执行和终止

1、Linux 进程的创建

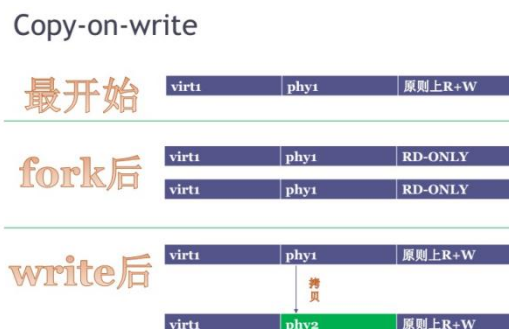
进程的创建主要发生在三类情景：操作系统初始化、用户手动创建新进程、正在运行的进程创建一个进程。
操作系统创建进程的一般过程如下：

- (1) 为新进程分配一个唯一的进程标识号，并申请一个空白的 PCB。
- (2) 尝试为新进程分配一块虚拟内存等资源，如果资源不足则该进程进入等待状态。
- (3) 虚拟内存资源分配成功后则初始化 PCB；
- (4) 如果进程的调度队列能够接纳新进程就插入就绪队列，等待被 CPU 调度运行。

Linux 系统中通过调用库函数接口 **fork()** 来复制当前进程创建一个新进程，调用 **fork** 函数的进程叫做**父进程**，新创建的进程叫做**子进程**。Linux 进程创建是**读时共享、写时复制**。

子进程与父进程的区别仅仅在于不同的 PID、PPID 和某些资源及统计量上。注意，Linux 中的 **fork()** 函数使用的是**写时复制**的技术，也就是**内核在创建进程时，其资源并没有被复制过来，所有的资源以只读的方式和父进程共享数据，一旦子进程需要写入数据时才进行资源复制**。这种**写时复制技术 COW(Copy-On-Write)**可以使 Linux 拥有快速执行的能力，因此这个优化是非常重要的。详细讲如下：

- (1) 在 **fork** 之前，一片内存区对应一份物理地址和一份虚拟地址，内存区的权限为 **RW**；
- (2) 在 **fork** 之后，父子进程看到的内存区虚拟地址相同，物理地址也相同，父子进程使用的其实是同一片物理内存，未发生内存拷贝，操作系统会将此内存区权限改为 **RO**；
- (3) 父或子进程对内存区执行写操作将触发 **PageFault**，操作系统此时会将内存区拷贝一份，父子进程看到的虚拟地址仍然一样，但是物理地址已经不同。



2、Linux 进程的执行

exec 函数族负责读取可执行文件并将其载入地址空间开始运行新创建的子进程。Linux 进程如何运行、运行多久，这就涉及到 CPU 的进程调度。

3、Linux 进程的阻塞

进程可以且只能自己阻塞自己，当程序必须等待某件事情发生后才能执行，进程就会等待，在以下情况下进程会等待（阻塞）：

- 请求并等待系统服务，无法马上完成
- 启动某种操作，无法马上完成
- 需要的数据没有到达。

4、Linux 进程的唤醒

进程只能被别的进程或操作系统唤醒，唤醒进程的原因有：

- 被阻塞进程需要的资源可被满足
- 被阻塞进程等待的事件到达
- 将该进程的 PCB 插入到就绪队列

5、Linux 进程的终止

进程的终止有三种情形：**程序执行完毕正常结束**、**程序错误异常结束**和**外界干预异常结束**（被其他进程关闭），有些是自愿退出的，有些是外界关于下非自愿退出的。终止过程的过程如下：

- （1）查找需要终止的进程 **PCB**，如果处于执行状态则立即终止进程；
- （2）如果还有子进程则**应将其**所有子进程关闭；
- （3）该进程所拥有的全部资源归还给父进程或操作系统；
- （4）将其从 **PCB** 所在队列中删除。

Linux 进程中关闭进程时可能出现**孤儿进程**或**僵尸进程**。

（1）孤儿进程

当父进程退出而它的子进程还在运行时，这些子进程就成为**孤儿进程**，孤儿进程将被 Init 进程（PID=1）收养，所以**孤儿进程**不会浪费系统资源。

（2）僵尸进程

父进程创建子进程后一般需要监控子进程的状态，如果子进程退出但是父进程没有读取其状态，则此时的子进程为僵尸进程。

具体说，在 Linux 系统中，一个进程使用 **fork** 创建子进程，如果子进程退出但是父进程并没有调用 **wait** 或 **waitpid** 来获取子进程的状态信息，那么子进程的进程描述符 **task_struct** 将会仍然存在，该子进程就是僵尸进程。僵尸进程会浪费系统资源。

僵尸进程产生原因：

- 1、子进程结束后向父进程发出 **SIGCHLD** 信号，父进程默认忽略了它；
- 2、父进程没有调用 **wait()**或 **waitpid()**函数来等待子进程的结束。

避免僵尸进程方法：

- 1、父进程调用 **wait()**或者 **waitpid()**等待子进程结束，这样处理父进程一般会阻塞在 **wait** 处而不能处理其他事情。
- 2、捕捉 **SIGCHLD** 信号，并在信号处理函数里面调用 **wait** 函数，这样处理可避免 1 中描述的问题。

（3）守护进程

启动操作系统时，通常会创建若干个进程，其中有些需要同用户交互并替用户完成某些工作，这时这些进程是**前台进程**，如果进程不需要与用户进行交互但是还有执行一些必要的功能，那么这些进程就是**后台进程**。

守护进程（daemon，也叫精灵进程）是一种特殊的**后台进程**，其**通常完全独立于控制终端，并且周期性地执行某种任务或等待处理某些事件**，操作系统中有很多守护进程，例如收发电子邮件的网络进程，在大部分事件下都处于休眠状态，但是当接收到电子邮件时就被唤醒。

四、进程间调度

什么是进程间调度？/什么时候进行进程间调度？/进程调度负责哪些事情？

在计算机系统是**多道程序设计**系统时，多个进程或线程间需要同时竞争 CPU。在操作系统中，负责完成这一工作的就是**调度程序**（scheduler），该类程序使用的算法就是**调度算法**（scheduling algorithm）。

什么时候进行进程调度？

（1）**就绪态→执行态**。进程创建成功时进入就绪态，此时它会被存入就绪队列，由调度程序决定什么时候执行；同时 Linux 中创建一个新进程后，调度程序还需要决定是运行父进程还是运行子进程。

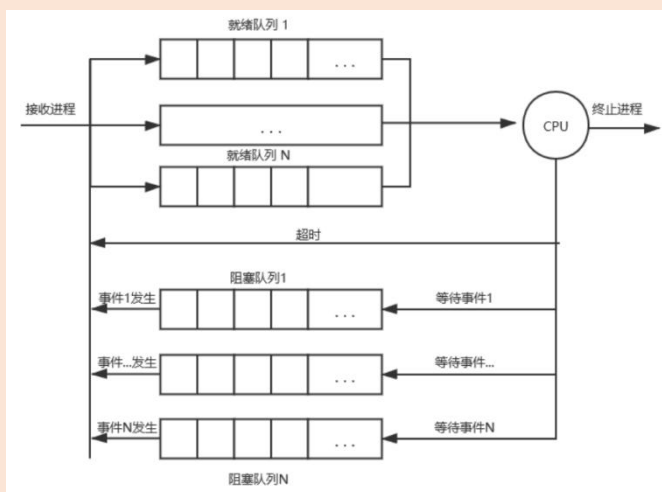
（2）**执行态→就绪态**。当前进程时间片用完后，需要从执行态转换为就绪态，当前进程退出执行时需要做出调度决策，需要决定下一个运行的是哪个进程。

（3）**执行态→阻塞态**。当一个进程阻塞在 I/O 和信号量或者由于其它原因阻塞时，必须选择另一个处于就绪态的进程运行。

（4）**阻塞态→就绪态**。某些被阻塞的等待该 I/O 的进程成为可运行的就绪进程时，是否让新就绪的进程运行，或者让中断发生时运行的进程继续运行，或者让某个其它进程运行，这就取决于调度程序的抉择了。

操作系统如何完成进程调度？

操作系统维护了一组状态队列，表示系统中所有进程的当前状态；不同的状态有不同的队列，有就绪队列、阻塞队列等，每个进程的 PCB 都根据它的状态加入到相应的队列中，当一个进程的状态发生变化时，它的 PCB 会从一个状态队列中脱离出来加入到另一个状态队列。同一个状态对应 N 个队列，是因为进程有优先级。



操作系统中的调度算法有哪些？

根据操作系统需要处理的任务类型不同，其对应的调度算法类型也不同。

（一）对于批处理任务系统（比如同时复制 1000 个文件），主要关注吞吐量、周转时间、CPU 利用率。

（1）**非抢占式的先来先服务**（first-come first-severd）算法：所有就绪态的进程在一个调度队列中排队，先请求 CPU 的进程先进入队列，阻塞后就绪的进程作为新进程加入队列，CPU 按照调度队列依次执行。

（2）**非抢占式的最短作业优先**（shortest job first）算法：先计算当前需要同时运行的每个进程的平均周转周期，即进程从进入队列到运行完成的平均耗时，然后选择对平均周转周期影响最大的进程作为最短作业优先执行，这样可以保证其他的进程等待的总体时间最少。该算法是基于预测性质的理想化算法，可能预测并不准确。

（3）**抢占式的最短剩余时间优先**（shortest remaining time first）算法：调度程序优先选择剩余时间最短的进程执行，也是通过理想化预测进程执行时间，可能并不准确，但是它是可以的打断当前正在执行的长时间的进程任务。

（二）对于交互任务系统（比如图形交互任务），主要关注响应时间、均衡性。

（1）**轮转调度**（round robin）：每个进程被分配一个时间片，每个进程在自己的时间片内运行，时间片设置得过长可能引起短交互请求响应时间过长，时间片设置得过短可能导致过多的进程切换操作，降低 CPU 效率，时间片一般为 20ms-50ms。

（2）**优先级调度**。在优先级调度算法中，每个优先级都有相应的队列，队列里面装着对应优先级的进程，首先在高优先级队列中进行轮转调度，当高优先级队列为空时，转而去低优先级队列中进行轮转调度，如果高优先级队列始终不为空，那么低优先级的进程很可能就会饥饿到很久不能被调度。

（3）**多级队列**。多级队列算法与优先级调度算法不同，优先级算法中每个进程分配的是相同的时间片，而在

多级队列算法中，不同队列中的进程分配给不同的时间片，当一个进程用完分配的时间片后就移动到下一个队列中，这样可以更好的避免上下文频繁切换。举例：有一个进程需要 100 个时间片，如果每次调度都给分配一个时间片，则需要 100 次上下文切换，这样 CPU 运行效率较低，通过多级队列算法，可以考虑最开始给这个进程分配 1 个时间片，然后被换出，下次分给它 2 个时间片，再换出，之后分给它 4、8、16、64 个时间片，这样分配的话，该进程只需要 7 次交换就可以运行完成，相比 100 次上下文切换运行效率高了不少，但顾此就会失彼，那些需要交互的进程得到响应的速度就会下降。

(4) **最短进程优先**。交互式系统中应用最短进程优先算法其实是非常适合的，每次都选择执行时间最短的进程进行调度，这样可以使任务的响应时间最短，不过根本无法非常准确地在当前可运行进程中找出最短的那个进程。基本都是根据进程过去的行为来预测执行时间。

(三) 对于实时任务系统（比如图形交互任务），主要关注满足截止时间、可预测性。

满足截止时间：需要在规定 deadline 前完成作业；

可预测性：可预测性是指在系统运行的任何时刻，在任何情况下，实时系统的资源调配策略都能为争夺资源的任务合理的分配资源，使每个实时任务都能得到满足。

调度算法分类：

硬实时：必须在 deadline 之前完成工作，如果 delay，可能会发生灾难性或发生严重的后果；

软实时：必须在 deadline 之前完成工作，但如果偶尔 delay 了，也可以容忍。

调度算法：

(1) 单调速率调度

采用抢占式、静态优先级的策略，调度周期性任务。每个任务最开始都被配置好了优先级，当较低优先级的进程正在运行并且有较高优先级的进程可以运行时，较高优先级的进程将会抢占低优先级的进程。在进入系统时，每个周期性任务都会分配一个优先级，周期越短，优先级越高。这种策略的理由是：更频繁的需要 CPU 的任务应该被分配更高的优先级。

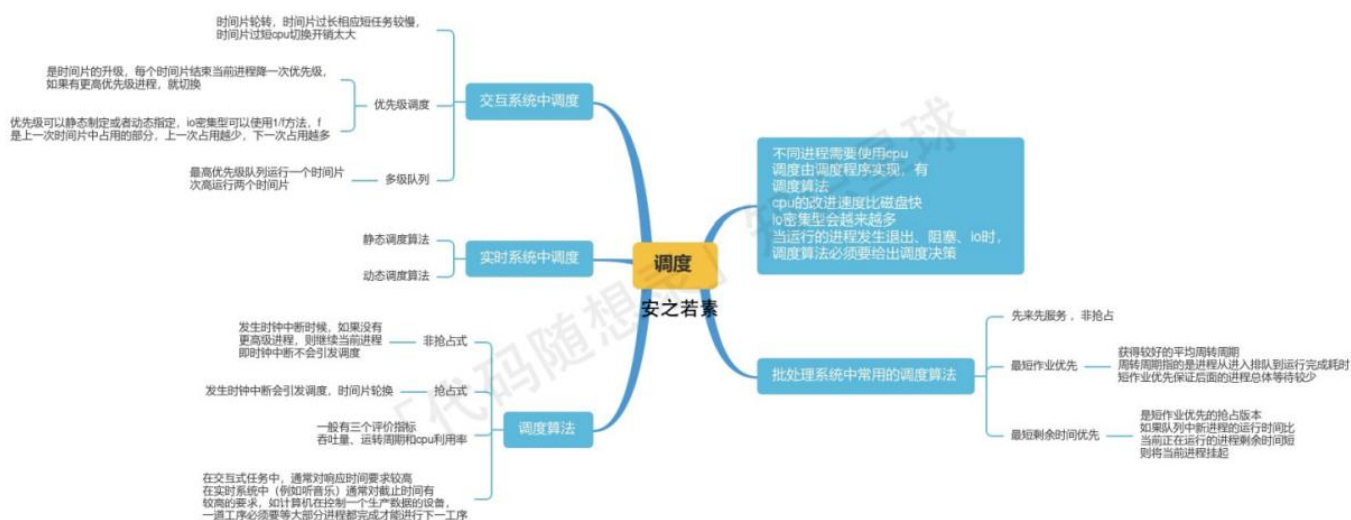
(2) 最早截止时间调度

根据截止时间动态分配优先级，截止时间越早的进程优先级越高。该算法中，当一个进程可以运行时，它应该向操作系统通知截止时间，根据截止时间的早晚，系统会为该进程调整优先级，以便满足可运行进程的截止时间要求。它与单调速率调度算法的区别就是一个是静态优先级，一个是动态优先级。

上述的抢占式指的是可以打断当前执行进程、非抢占式指的是不可打断当前执行进程。

Linux 系统使用如下的调度算法：(1) 实时先入先出算法 (2) 实时轮转算法。 (3) 分时

进程调度



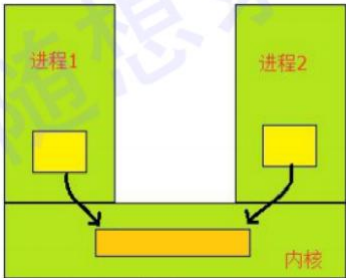
五、进程间通信

在很多应用中，进程间的通信可能是不可避免的，因为不同的进程间可能也要进行数据传输、资源共享、通知事件

或进程控制。为了实现这些功能，就需要操作系统提供**进程间通信的功能**。注意这里讨论的进程间通信都是**同一主机上的不同进程间的通信**，**不同主机间的进程通信**是网络编程中的**套接字（socket）**。

1、IPC 的原理

操作系统提供的**进程间通信** (IPC, InterProcess Communication) 就是在**内核空间开辟一块缓存区**，进程 A 把数据从用户空间拷到**内核缓冲区**，进程 B 再从**内核缓冲区**把数据读走。这个缓存区必须在**内核中**，因为操作系统的每个进程的用户空间都是相互隔离的，彼此数据都不可见，**所以进程之间想要进行通信必须通过内核交换数据**。



2、IPC 的 5 种技术

请注意，有的文献也将 socket 包含在进程通信中，socket 可以用在不同主机的进程间通信，是网络编程；以下 5 种方法只能用于同一主机间的不同进程间通信。

假设进程 A 要和进程 B 通信，进程间通信的技术有如下几种：

(1) **管道技术 (PIPE)**（也叫匿名管道）：是 IPC 中最基本的方式，管道将和另一个进程的通信看做是对一个文件的读写操作，`popen()`函数类似于 `fopen()`函数，`pipe()`函数类似于 `open()`函数，返回的是对象描述符。**缺陷是：管道必须在亲属进程(同一父进程创建出的相关进程)之间进行数据传输；一个管道只能进行单向通信。**

(2) **命名管道 (FIFO)**：命名管道可用于无亲属关系的进程间通信，其原理是通过函数 `mkfifo()/mknod()`在文件系统中创建一个有路径和名称的 FIFO 文件，不同的进程访问这个文件就可以相互通信。

(3) **消息队列 (MSG)**：消息队列是操作系统内核负责维护的一个链表结构，程序员可以创建一个消息队列并获得一个唯一的标识符 `key` 值，进程双方根据 `key` 值确定同一个消息队列进程通信，消息必须指定消息类型并封装在专用的 `msg` 结构中，通信双方可以根据不同的消息类型来获取不同的消息。

	创建	进程 A	进程 B
匿 名 管道	//创建匿名管道 <code>int fd[2];//0 读 1 写 pipe(fd);</code>	// 子进程 A 负责读 fd[0] <code>close(fd[1]); read(fd[0],&chr,1); printf("%c\n",chr); close(fd[0]);</code>	// 子进程 B 负责写 fd[1] <code>close(fd[0]); char chr='a'; write(fd[1],&chr,1); close(fd[1]);</code>
命 名 管道	// 创建命名管道 <code>mkfifo("fifo",0660);</code>	<code>char buf[256]; Int fd=open("fifo",O_RDONLY); //子进程读管道中的数据 read(fd,buf,10); buf[10]=0; printf("%s",buf); close(fd);</code>	<code>int fd=open("fifo",O_WRONLY); //父进程向管道写入数据 write(fd,"fifo test\n",10); close(fd);</code>
消 息 队列	// 创建消息队列的 key <code>#define MSG_KEY 111</code>	<code>int msqid=msgget(MSG_KEY,0); struct msgbuf buf; msgrcv(msqid,(void *)&buf,sizeof(struct msgbuf),1,0);</code>	<code>int mspid=msgget(MSG_KEY,IPC_CREAT 0666); struct msgbuf buf; buf.mtype=1; strcpy(buf.mtext,"Hello World!");</code>

		<pre>printf("child:rcv a msg is %s\n",buf.mtext);</pre>	<pre>msgsnd(mspid,(const void *)&buf,sizeof(struct msgbuf),0); printf("parent:snd a msg is %s\n",buf.mtext);</pre>
--	--	---	--

（4）**信号（Signal）**：一个进程只可以给它的同一进程组的进程发送信号，进程 A 可以向进程 B 发送信号，如果进程 B 此时没有被执行，内核程序会先保存该信号，进程 B 必须提前制定一个信号处理函数来获取所有收到的信号，当 A 的信号达到时，进程 B 立即切换到信号处理函数。我们常用的快捷键很多都使用信号量来实现。该通信方式一般只能用于特定信号的通知，消息的类型一般是固定的，不能传输要交换的任何数据。

信号和信号量的区别？

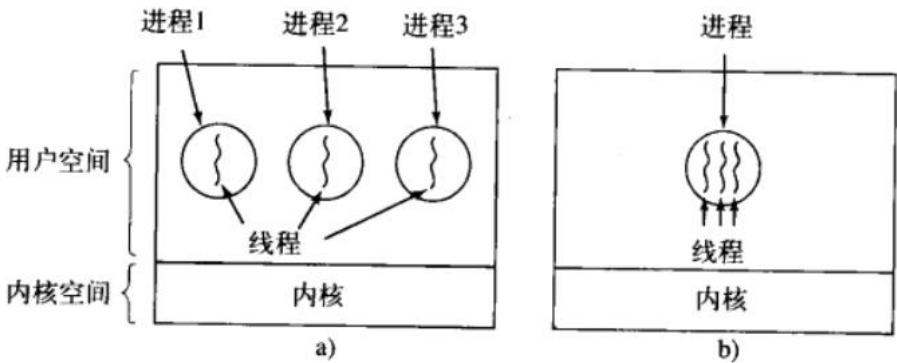
信号是进程通信方式；信号量是多线程/进程同步方式，其本质是一个特殊的内核程序中的原子化变量。

（5）**共享内存（SHM）**。是 IPC 中的最快的效率最高的方法之一。因为在内存中操作，所以一个进程向共享内存写入数据，共享的进程立刻就会察觉；允许多个进程同时通信，所以一般需要和信号量等同步手段配合使用。它的实现方式有很多种，主要的有 mmap 系统调用、Posix 共享内存以及 System V 共享内存等。通过这三种“工具”共享地址空间后，通信的目的自然就会达到。

六、线程

1、线程（thread）

线程概念也是随着操作系统不断完善的。传统操作系统中，每个进程都有一个对应的地址空间和一个控制线程。但是现代操作系统中，**同一个进程中可以并行运行多个线程，多个线程共享同一个进程的地址空间和其他资源等，在 Linux 系统中线程也叫做轻量级进程**。如下图：



传统操作系统 a) 现代操作系统 b)

（1）线程的优势

使用线程后就可以开发多线程程序，多线程的独特优势？（为什么需要多线程设计？）

- （1）**提高程序并发性**。多线程可以满足需要同时进行多种功能的应用需求。很多应用程序的功能实现需要同时有多种活动同时进行，而且有些活动可能需要阻塞，比如服务器等待客户端连接，服务器在等待客户端连接的同时，可能需要同时处理用户的输入，显然这种需求单进程/单线程是完成不了的。
- （2）**在并发性程序上比多进程开销更小**。多线程是轻量级进程，比单纯的多进程设计更快。多线程是同一个地址空间内的共享同一个地址空间和所有可用数据的并行实体，线程比进程更加轻量级，更容易创建和撤销，在许多系统中创建线程比创建进程要快 10~100 倍。
- （3）**数据通信、共享数据比多进程方便**。多线程可以大幅提升需要大量计算或大量 I/O 处理的应用的执行速度。即使一个应用不需要同时执行多种功能，但是只要其计算量大或者读写操作多，都可以考虑多线程。

使用线程的缺点：

- 1、依赖第三方线程包的库函数，实现上随着库函数迭代可能不稳定；
- 2、多线程程序的编写和调试都更加困难，维护成本高；
- 3、进程中的一个线程崩溃时会导致其所属进程中的所有线程崩溃；
- 4、对信号支持不好。

(2) 进程和线程的比较

相同点：

- (1) 进程和线程的**状态切换基本类似**，都具有就绪、阻塞、运行等状态。
- (2) 进程和线程在**同步与互斥上机制相似**。

不同点：

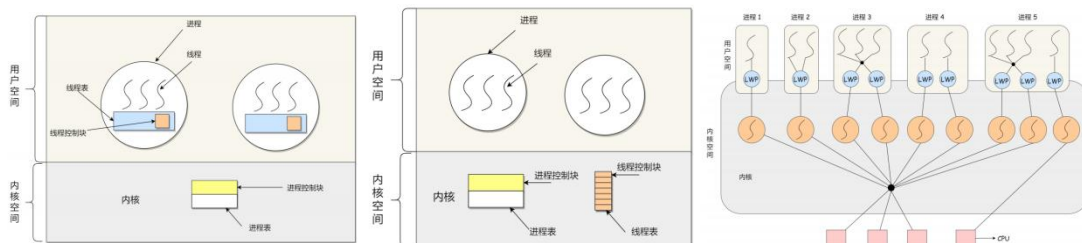
- (1) **功能不同**。进程是操作系统最小的资源分配的单位，线程是操作系统的最小的执行单位。
- (2) **资源不同**。操作系统只为每一个进程分配资源和回收资源；同一个进程中的所有线程共享该进程的资源，同时每个进程也独享一些资源：寄存器、堆栈、状态和程序计数器。如下：

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

- (3) **性能不同**。线程切换比进程切换的开销更小，适合程序的并行设计。这是因为进程切换一般需要切换到内核空间，线程一般不需要（Linux 特殊），线程独享的资源少。

(3) 线程的 3 种实现方式

有两种主要的方法来实现多线程机制：**用户级线程**和**内核级线程**。前者在操作系统的**用户空间**实现线程包，后者在操作系统的**内核空间**实现线程包。在这两种方式上，Linux 使用了轻量级进程（LightWeight Process），其在内核中支持用户线程。



线程类型	特点	优点	缺点
用户级线程	在用户空间实现线程包，在用户空间每个进程都有其独立的线程表	<ol style="list-style-type: none"> 1、移植性强，可用于不支持多线程的操作系统 2、线程切换快，用户空间的线程切换不需要操作系统在内核态和用户态转换，比内核级线程快一个数量级以上。 3、自由度高，允许开发者自定义每个进程中多线程的调度算法。 	<ol style="list-style-type: none"> 1、某个用户线程阻塞时该进程下的其他线程都会阻塞 2、某个用户线程运行后且不主动停止，其他线程就不能运行。
内核级线程	在内核空间实现线程包，在内核空间存在记录系统中所有线程的线程表	<ol style="list-style-type: none"> 1、某个内核线程阻塞时不会影响其他内核线程的执行。 2、每个内核线程都可以直接获取 CPU 资源，拥有更多的 CPU 使用时间 	<ol style="list-style-type: none"> 1、需要由内核来维护内核线程的信息，占用内核资源 2、内核线程的创建、终止和切换都是在内核空间中进行的，开销比较大。
轻量级进程 (LWP)	轻量级进程(LWP)是建立在内核之上并由内	代表系统就是 Linux 系统，综合了内核级线程和用户级线程的优点	

	核支持的用户线程，每一个轻量级进程都与一个特定的内核线程关联。	
--	---------------------------------	--

2、Linux 的线程

从用户空间看，Linux 中用户空间的线程使用一套通用的第三方 pthread 线程库，因此具有很好的移植性。一个进程可以拥有多个线程，其中每个线程共享该进程所拥有的资源。要注意的是，由于线程共享了进程的资源 and 地址空间，因此，任何线程对系统资源的操作都会给其它线程带来影响。由此可知，多线程中的同步是非常重要的问题。在多线程系统中，进程与线程的关系如下图所示：



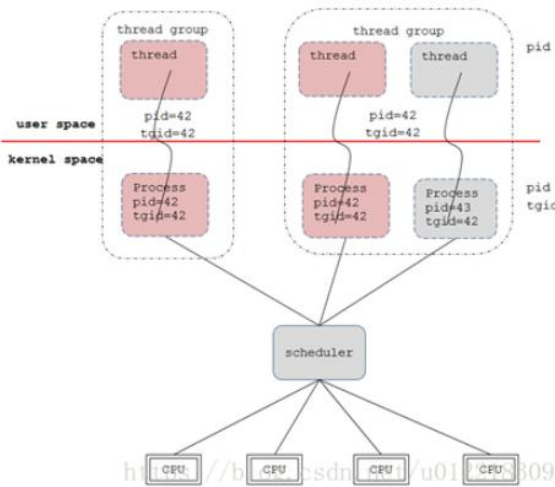
进程和线程之间的关系

从内核空间看，Linux 系统的每一个线程都是一个单独的任务，一个任务等同于一个内核级的进程，一个内核级的进程对应于一个线程组，对比如下：

Linux 创建进程时只有一种方式，即调用 fork 函数从一个父进程中复制而来。新创建的子进程在内核空间的进程表中创建一个新的 task_struct，其进程标识符 PID 和线程组标识符 TGID 都设为一个新值，然后在用户空间创建对应的控制线程。

Linux 创建线程时只有一种方式，即调用 pthread_create 函数从一个父线程中复制而来。新创建的子线程在内核空间的进程表中创建一个新的 task_struct，其进程标识符 PID 设为一个新值，线程组标识符 TGID 不变，然后在用户空间创建对应的用户级线程。

从上述内核层面的进程和线程的创建过程中，可以发现 **Linux 系统中的进程与线程的概念和实现非常接近，只是调用的函数不一样，设置的 task_struct 的参数不一样**，实际上确实如此，fork 和 pthread_create 两个函数在内部实现中都调用了 clone，只是传入的参数有区别，所以 **Linux 系统中线程和进程已经非常接近，线程也被称作轻量级进程**。Linux 系统中的每一个线程都对应一个内核空间的 task_struct，所以对于调度算法而言，**每一个线程都参与 CPU 资源的竞争**。



C++11 开始在语言层面支持多线程，提供了标准的线程库 std::thread，其基础操作如下：

```
#include<thread>
using namespace std;
```


// 创建并且执行线程对象

```
thread t1;// 创建一个空的线程对象;  
thread t2(f1,n+1);// 创建一个线程对象, 传入返回值为 void 的 f1 函数, f1 参数为 n+1;  
thread t3([]{});//创建一个线程对象, 传入 lambda 表达式
```

// 线程的阻塞设置

```
t1.join();// 阻塞主线程, 执行完后, 主线程继续执行, 一般用在主线程最后, 防止子线程没有执行完毕就退出  
t1.detach();// 不阻塞主线程, 同时主线程继续执行, 子线程变成守护线程, 被 C++运行时库接管。  
//一个线程只能调用一次 join()或 detach(),调用 detach()后不能再调用 join()
```

// 线程的操作

```
joinable()// 检查线程是否可被 join 或 detach()  
std::this_thread::getid()// 获取当前线程的 ID  
std::this_thread::yield()// 当前线程放弃执行  
sleep_until()// 线程休眠至某个指定时刻才被重新唤醒  
sleep_for()// 线程休眠某个指定的时间段才被重新唤醒
```

3、一个进程的最大线程数

Linux 系统中并没有专门的内核参数来限制单个进程的最大线程数, 但是其线程个数肯定受系统资源的限制, 是有上限的。Linux 系统中进程中创建的线程需要在用户空间的虚拟内存中分配空间, 所以其个数受**用户空间的虚拟内存**和**系统参数**限制。

创建一个线程默认情况下需要的内存大小是不同的, 可能是 8M, 也可能是 10M。(ulimit -a 查看)

- 32 位系统, 一个进程的虚拟空间是 4G, 内核态的虚拟空间只有 1G, 用户态的虚拟空间只有 3G, 如果创建线程时分配的栈空间是 10M, 那么一个进程最多只能创建 300 个左右的线程。

64 位系统, 用户态的虚拟空间可能大到有 128T, 理论上不会受虚拟内存大小的限制, 而会受系统的参数或性能限制, 例如下面这三个内核参数的大小, 都会影响创建线程的上限:

/proc/sys/kernel/threads-max, 表示系统支持的**最大线程数**, 默认值是 14553;

/proc/sys/kernel/pid_max, 表示**系统全局的 PID 号数值的限制**, 每一个进程或线程都有 ID, ID 的值超过这个数, 进程或线程就会创建失败, 默认值是 32768;

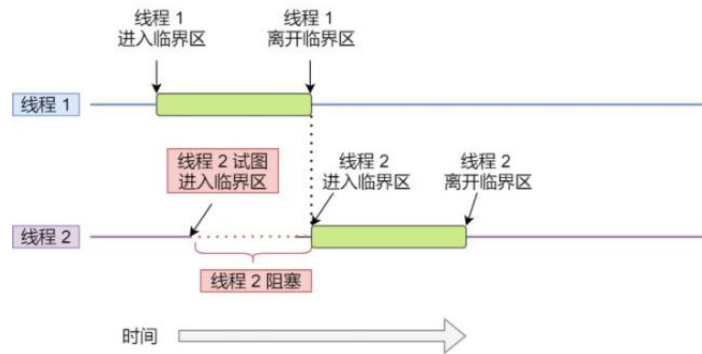
/proc/sys/vm/max_map_count, 表示限制**一个进程可以拥有的 VMA (虚拟内存区域) 的数量**, 具体什么意思我也没搞清楚, 反正如果它的值很小, 也会导致创建线程失败, 默认值是 65530。

七、多线程同步

CPU 在进程之间的调度可能导致进程在读写一些共享数据时出现问题。(例如进程 A 在读写某个共享变量 a 时恰好被 CPU 打断, 然后共享变量 a 被另一个进程 B 进行了修改, 这时进程 A 可能会产生错误的结果。)

1、同步和互斥的区别

两个或多个进程读写某些**共享数据**, 而最后的结果取决于运行的精确时序, 称为**竞争条件** (race condition), 包含**竞争条件**的程序大部分运行良好, 但是极少数情况会出现错误。想要避免竞争条件, 就需要确保一个进程在使用一个共享资源时其他资源不可以做同样的操作, 这就是**互斥** (mutual exclusion)。一个进程中会对共享内存进行访问的程序片段叫做**临界区域** (critical region) 或**临界区** (critical section)。



同一个进程中的多个线程共享进程内的资源，所以多个线程之间的工作也需要互斥，同时多线程也需要同步。所谓同步就是并发进程/线程在一些关键点上可能需要互相等待与互通消息，这种相互制约的等待与互通信息称为进程/线程同步。

同步和互斥的区别：

同步就好比：「操作 A 应在操作 B 之前执行」，「操作 C 必须在操作 A 和操作 B 都完成之后才能执行」等；
互斥就好比：「操作 A 和操作 B 不能在同一时刻执行」；

2、同步和互斥的实现

(1) 互斥锁

使用**加锁(lock)**和**解锁(unlock)**操作来控制对共享资源的访问，就可以解决不同进程或线程之间的**互斥**问题，这种锁也叫做**互斥锁**或**互斥量**。互斥锁的数据类型是：`pthread_mutex_t`。

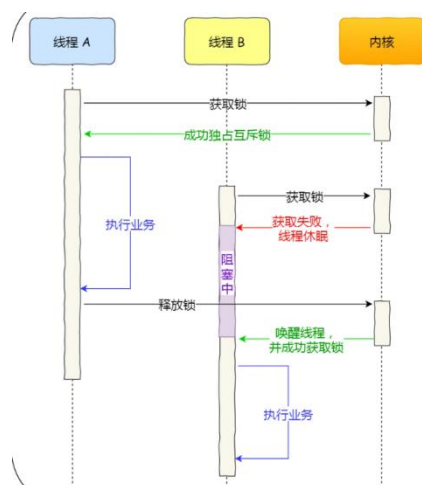
互斥锁的工作原理/流程：

互斥锁是一种互斥机制，本质上是一把锁，在访问共享资源前对加锁，在访问完成后释放锁。

在当前线程加锁后，解锁前，其他任何线程都不能访问该互斥锁锁住的临界区，进入休眠或者忙等待，这体现了互斥性和唯一性。

在当前线程解锁后，内核程序会在等待队列中取出一个等待线程，将其唤醒并让其获得该互斥锁进行执行。

根据互斥锁的实现方式，也可分为**忙等待锁**（**自旋锁**（`spin lock`））和**无等待锁**。**忙等待锁**中获取不到锁的线程就会一直进行 `while` 循环而不做任何事，一直占据 CPU 资源，直到锁可用。**无等待锁**则是线程没有锁的时候就放入锁的等待队列，然后将 CPU 资源让给其他线程。



互斥锁可以保证任何时刻只能有一个线程访问临界区，请注意把握临界区的选择，如果将线程中的大部分甚至全部代码都设置为临界区，那么线程的执行将趋近于串行执行，失去了并行执行的优势，降低并发效率。

自旋锁加锁和解锁的性能开销比互斥锁要小一点，但是长时间执行会占据 CPU 资源，所以**当临界区的代码执行时间非常短时，应该选用自旋锁，否则使用互斥锁（无等待锁）**。

C++11 通过 `#include <mutex>` 提供了互斥锁，该文件提供 4 种锁变量：

`std::mutex`，最基本的 `Mutex` 类。

`std::recursive_mutex`，递归 `Mutex` 类。

`std::time_mutex`，定时 `Mutex` 类。

`std::recursive_timed_mutex`，定时递归 `Mutex` 类。

上述这 4 种锁变量，其中基本都有 `lock()`、`unlock()` 和 `try_lock()` 等操作，这些操作都需要程序员自行调用。为了更加方便地对互斥量进行上锁和解锁，该文件又提供了 2 种更加方便地上锁的类：

`std::lock_guard` // 与 `Mutex RAII` 相关，上锁后，不能再手动解锁/上锁，只能在声明区结束时自动解锁。

`std::unique_lock` // 与 `Mutex RAII` 相关，上锁后，可以再次手动解锁或者上锁，声明区结束时自动解锁。

`unique_lock` 效率不如 `lock_guard`，但是 `unique_lock` 的颗粒度更细，可以在声明区中手动解锁和上锁。

使用 `unique_lock` 的用法如下：

```
#include <mutex>
std::mutex mtx; // 互斥锁
{
    std::unique_lock<std::mutex> locker(mtx); // 对声明区上锁
    // 执行临界区的代码（保证同一时刻只能有一个线程在执行这段代码）
    locker.unlock(); // 临时解锁
    // 执行非临界区的代码（同一时刻可能有多个线程在执行这段代码）
    locker.lock(); // 再次加锁
}
```

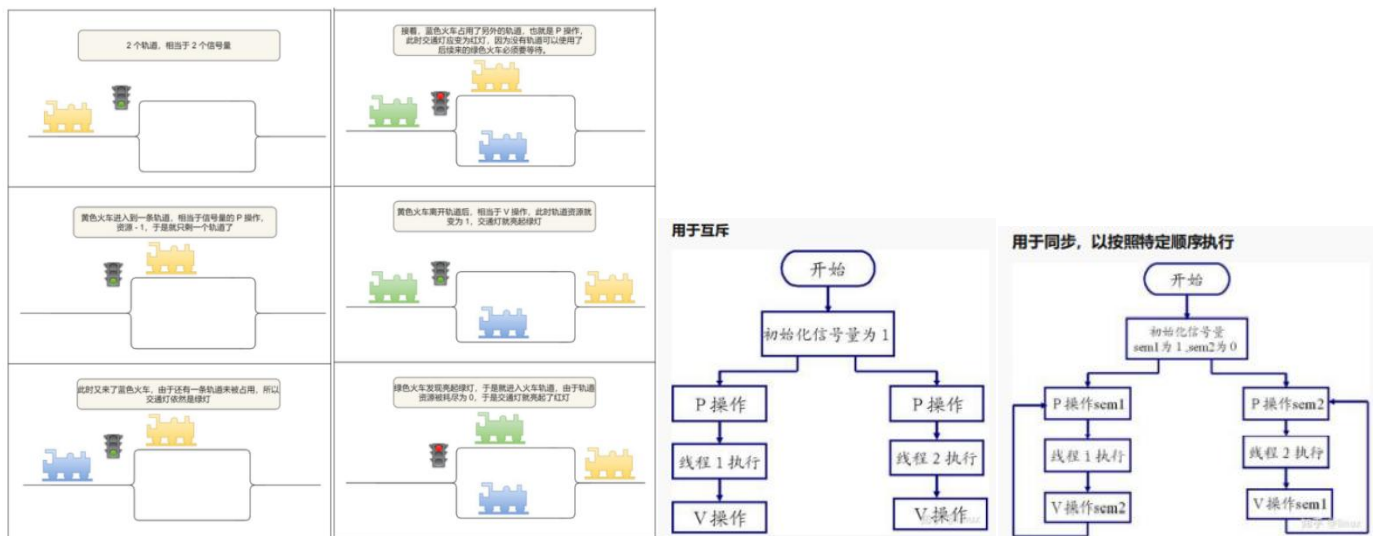
(2) 信号量

信号量是操作系统提供的一种协调共享资源的方法，它本质是一个特殊变量类型，一个**信号量**表示系统中某个共享资源的数量。**信号量** `sem` 有两个**原子操作** `P(sem)` 和 `V(sem)`：（原子操作即单一的不可分割的操作，原子操作要么不执行，要么全部执行，执行中不可中断）。

P 操作：将 `sem` 减 1，相减后，如果 `sem < 0`，则进程/线程进入阻塞等待，否则继续，表明 `P` 操作可能会阻塞；

V 操作：将 `sem` 加 1，相加后，如果 `sem <= 0`，唤醒一个等待中的进程/线程，表明 `V` 操作不会阻塞；

`P` 操作是用在进入临界区之前，`V` 操作是用在离开临界区之后，这两个操作是必须成对出现的。举个类比，2 个资源的信号量，相当于 2 条火车轨道，`PV` 操作如下图过程：



所以**信号量**也叫做**信号灯**，最简单的信号量是**二进制信号量**，即只能取 0 和 1，二进制信号量和互斥锁原理基本就一样。信号量不仅可以实现临界区的互斥访问控制，还可以线程间的事件同步。

(3) 条件变量

条件变量是利用线程间共享的全局变量进行同步的线程同步机制；在条件变量上等待的线程以睡眠的方式等待条件变量的满足；一个线程等待"条件变量的条件成立"挂起，另一个线程使"条件成立"；条件变量的使用总是和一个互斥锁结合在一起。

C++11 中条件变量的使用：

```
#include <condition_variable> // 条件变量
std::mutex mtx; // 创建一个互斥锁
std::condition_variable cond; // 创建一个条件变量
{
    std::unique_lock<std::mutex> locker(mtx); // 对声明区上锁
    cond.wait(locker); // 阻塞当前线程，释放锁，等待被唤醒
    cond.wait(locker, []{ return ready; }); // 如果条件满足，继续执行；否则阻塞当前线程并释放锁，等待被唤醒
    cond.wait_for(locker, time_duration, []{ return ready; }); // 如果条件满足或者超时则执行，否则当前线程并释放锁
    cond.notify_one(); // 唤醒等待队列中的第一个阻塞线程；不存在锁争用，能够立即获得锁，其余的线程不会被唤醒
    cond.notify_all(); // 唤醒等待队列中的所有阻塞线程，存在锁争用，只有一个线程能够获得锁，其余未获取锁的线程继续尝试获得锁
} // 自动解锁
```

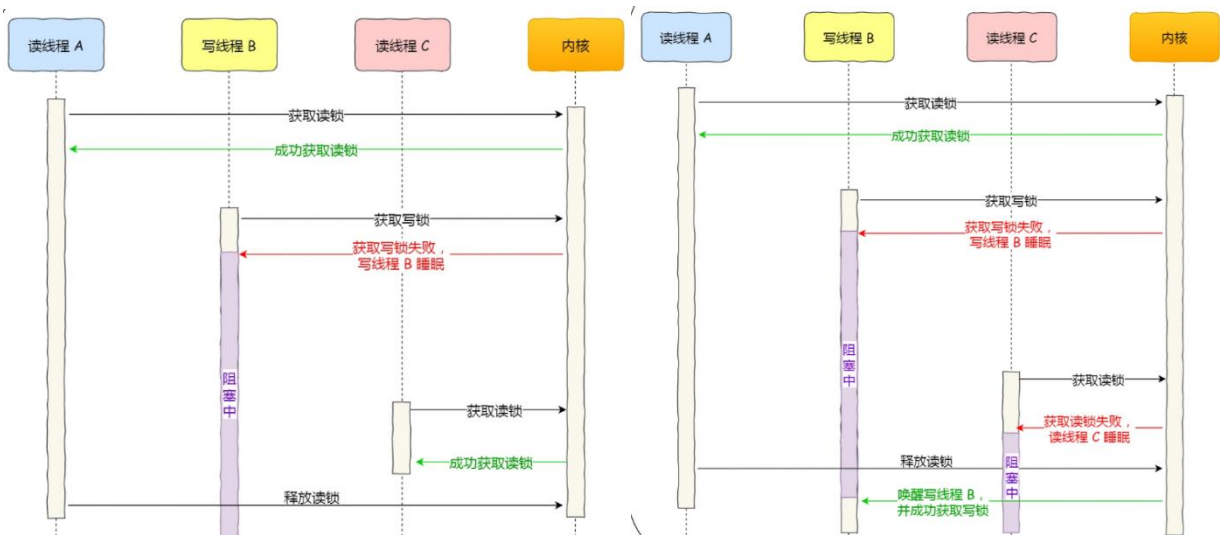
(4) 读写锁

在对数据的读写操作中，更多的是读操作，写操作较少，例如对数据库数据的读写应用为了满足当前能够允许多个读出，但只允许一个写入的需求，线程提供了**读写锁**来实现。**读写锁适用于能明确区分读操作和写操作的场景。互斥锁和自旋锁都是最基本的锁**，读写锁一般是根据场景来选择这两种锁其中的一个来实现的

读写锁的基本特点是**读读并发，读写、写写互斥**。根据实现的不同，读写锁可以分为「**读优先锁**」、「**写优先锁**」和「**公平读写锁**」。

读优先锁是优先服务读线程，其工作方式是：当读线程 A 先持有了读锁，写线程 B 在获取写锁的时候，会被阻塞，并且在阻塞过程中，后续来的读线程 C 仍然可以成功获取读锁，最后直到读线程 A 和 C 释放读锁后，写线程 B 才可以成功获取写锁。（下图左侧）

写优先锁是优先服务写线程，其工作方式是：当读线程 A 先持有了读锁，写线程 B 在获取写锁的时候，会被阻塞，并且在阻塞过程中，后续来的读线程 C 获取读锁时会失败，于是读线程 C 将被阻塞在获取读锁的操作，这样只要读线程 A 释放读锁后，写线程 B 就可以成功获取读锁。（下图右侧）



读优先锁对于读线程并发性更好，但如果一直有读线程获取读锁，那么写线程将永远获取不到写锁，造成写线程「饥饿」的现象。写优先锁可以保证写线程不会饿死，但是如果一直有写线程获取写锁，读线程也会被「饿死」。

「**公平读写锁**」不偏袒任何一方，公平读写锁比较简单的一种方式：**用队列把获取锁的线程排队，不管是写线程还是读线程都按照先进先出的原则加锁即可**，这样读线程仍然可以并发，也不会出现「饥饿」的现象。

（5）乐观锁和悲观锁

互斥锁、自旋锁、读写锁都属于**悲观锁**。

悲观锁做事比较悲观，它认为**多线程同时修改共享资源的概率比较高**，于是很容易出现冲突，它的工作方式是：每次访问共享资源前，先要上锁，操作完成后再解锁。

乐观锁做事比较乐观，它认为**多线程同时修改共享资源的概率比较低**，于是不容易出现冲突，它的工作方式是：先修改完共享资源，再验证这段时间内有没有发生冲突，如果没有其他线程在修改资源，那么操作完成，如果发现有其他线程已经修改过这个资源，就放弃本次操作。你会发现乐观锁全程并没有加锁，所以它也叫**无锁编程**。

放弃后如何重试和具体业务场景息息相关，虽然重试的成本很高，但是冲突的概率足够低的话，还是可以接受的。**乐观锁**虽然去除了加锁解锁的操作，但是一旦发生冲突，重试的成本非常高，所以**只有在冲突概率非常低，且加锁成本非常高的场景时，才考虑使用乐观锁**。

乐观锁的使用场景如：**版本管理工具** SVN、Git 和**在线文档**，它们的主要思想是，同一份共享的资源，先让用户编辑，然后提交的时候，通过资源版本号来判断是否产生了冲突，发生了冲突的地方，需要用户自己修改后再重新提交。

（6）原子操作

在多线程编程中，对某个数据资源的访问必须考虑多线程竞争问题，我们可以为每个数据资源都加上互斥锁，但是类里的每个变量都加上互斥锁实现麻烦还会影响性能。C++11 提供了系统层面的原子操作<atomic>。

atomic 对 int、char、bool 等数据结构进行了原子性封装，在多线程环境中，对 std::atomic 对象的访问不会造成竞争-冒险。利用 std::atomic 可实现数据结构的无锁设计，而且性能比手动加锁要高。

（7）死锁

如果一个进程/线程集合中的每一个进程都在等待只能由该进程/线程集合中的其他进程才能引发的事件，那么该进程集合就是**死锁（dead lock）**。完全避免死锁是非常困难的。

死锁发生的情景是一个线程/进程需要对多个锁（例如 ABC）进行加锁或者解锁操作；
尽可能避免死锁的方法：

- （1）一次性同时加锁，使用 C++ 提供的 std::lock()
- （2）固定顺序来获取锁，线程 1 中以 A、B、C 是顺序获取锁，线程 2 也按照 ABC 顺序。

4、Linux 的线程池实现

线程池（thread pool）是一种在 Linux 并发场景中经常用到的工具，它的作用是提前创建好若干个线程，避免在业务繁忙的高并发场景中临时频繁创建和销毁线程带来的性能损耗。线程池的基本需求是：

- 提前创建若干个线程，将其设置为休眠
- 当需要线程执行任务时，向线程池中添加一个任务，该线程池自动挑选一个空闲线程来执行任务。

实现线程池的难点在于解决并发问题：将多个任务同时分配给多个线程，需要避免多个线程同时竞争同一个任务。解决思路是使用一个任务队列，向队列添加任务时需要加锁，确保同一时刻只能被一个线程添加；从队列取出任务时需要加锁，确保线程池中的多个线程在同一时刻只能有一个线程取出任务，并且该线程在执行任务时需要解锁，最大程度保证并发性，这中间需要涉及多线程、互斥锁、条件变量。

这里只介绍其中的核心函数：

```

private:
using Task= std::function<void()>;// C++11允许使用using为模板定义别名, typedef可以给普通变量定义别名
struct Pool {
    // 数据结构
    std::queue<Task> taskQue;//任务队列, 每个元素为函数对象类型
    std::vector<std::thread> threadsArr;//工作线程数组, 每个元素为线程对象类型
    // 多线程同步
    std::mutex queue_mutex;// 互斥锁, 保证每一刻只有一个线程可以从任务队列中取出任务
    std::condition_variable condition;// 条件变量, 控制阻塞线程的唤醒与阻塞
    // 标志变量
    std::atomic_bool isClosed;// 原子布尔变量, 是否关闭, 可以像正常bool一样使用, 但是该变量具有原子性
};
std::shared_ptr<Pool> poolPtr;// 共享智能指针

// 函数模板, 添加一个任务
template<class F>
void AddTask(F&& task) {
    if(!this->poolPtr->isClosed){
        std::lock_guard<std::mutex> locker(poolPtr->queue_mutex);// 声明区加锁
        this->poolPtr->taskQue.emplace(std::forward<F>(task));// 向任务队列中添加一个任务
        this->poolPtr->condition.notify_one();// 唤醒等待队列中的一个阻塞线程
    }
    else{
        throw std::runtime_error("theadpool is closed but have task adding");
    }
}

// 构造函数, explicit表明必须显示调用
explicit ThreadPool(size_t threadCount):
// 初始化列表
poolPtr(std::make_shared<Pool>()) {
    this->poolPtr->isClosed=false;
    assert(threadCount > 0);
    //创建threadCount个线程对象
    for(size_t i = 0; i < threadCount; i++) {
        // 创建一个线程对象, 传入lambda表达式
        this->poolPtr->threadsArr.emplace_back(
            std::thread(
                // lambda表达式
                [pool = this->poolPtr] {
                    std::unique_lock<std::mutex> locker(pool->queue_mutex);//对以下声明块加锁
                    while(true) {
                        // 如果任务队列不为空
                        if(!pool->taskQue.empty()) {
                            // 出队, 获取一个任务, 使用了移动语义
                            auto task = std::move(pool->taskQue.front());
                            pool->taskQue.pop();
                            // 临时解锁, 让其他线程去竞争锁
                            locker.unlock();
                            // 执行任务
                            task();
                            // 继续加锁
                            locker.lock();
                        }
                        else if(pool->isClosed) break;// 如果线程池关闭, 跳出循环, 结束线程
                        // 此时任务队列为空, 且线程池没有关闭,
                        else pool->condition.wait(locker);//先阻塞当前线程, 然后解锁locker, 等待唤醒
                    }
                }
            )
        );
    }
}

// 析构函数
~ThreadPool() {
    // 判断该智能指针是否存在
    if(static_cast<bool>(this->poolPtr)) {
        {
            std::lock_guard<std::mutex> locker(this->poolPtr->queue_mutex);// 声明区加锁
            this->poolPtr->isClosed = true;// 设置线程池状态为关闭
        }
        this->poolPtr->condition.notify_all();//唤醒所有阻塞线程
        for(std::thread& t: this->poolPtr->threadsArr){
            if(t.joinable()){
                t.join();// 如果有线程没有执行完, 等待执行完毕
            }
        }
    }
}

```

4、经典同步问题

(1) 互斥同步问题: 生产者-消费者问题

生产者在生成数据后, 放在一个缓冲区中; 消费者从缓冲区取出数据处理; 任何时刻, 只能有一个生产者或消费者可以访问缓冲区。如何设计多线程的生产和消费代码?



分析问题：任何时刻只能有一个线程操作缓冲区，说明操作缓冲区是临界代码，需要互斥；缓冲区空时，消费者必须等待生产者生成数据，缓冲区满时，生产者必须等待消费者取出数据，说明生产者和消费者需要同步。所以需要三个信号量：

互斥信号量 **mutex**：用于互斥访问缓冲区，初始化值为 1；

资源信号量 **fullBuffers**：用于消费者询问缓冲区是否有数据，有数据则读取数据，初始化值为 0（表明缓冲区一开始为空）；

资源信号量 **emptyBuffers**：用于生产者询问缓冲区是否有空位，有空位则生成数据，初始化值为 **n**（缓冲区大小）；

实现代码：

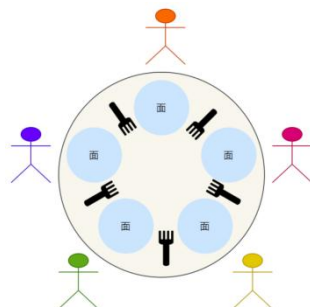
```
#define N 100
semaphore mutex = 1;           // 互斥信号量，用于临界区的互斥访问
semaphore emptyBuffers = N;     // 表示缓冲区「空槽」的个数
semaphore fullBuffers = 0;      // 表示缓冲区「满槽」的个数

// 生产者线程函数
void producer()
{
    while(TRUE)
    {
        P(emptyBuffers);        // 将空槽的个数 - 1
        P(mutex);               // 进入临界区
        将生成的数据放到缓冲区中;
        V(mutex);               // 离开临界区
        V(fullBuffers);         // 将满槽的个数 + 1
    }
}

// 消费者线程函数
void consumer()
{
    while(TRUE)
    {
        P(fullBuffers);         // 将满槽的个数 - 1
        P(mutex);               // 进入临界区
        从缓冲区里读取数据;
        V(mutex);               // 离开临界区
        V(emptyBuffers);        // 将空槽的个数 + 1
    }
}
```

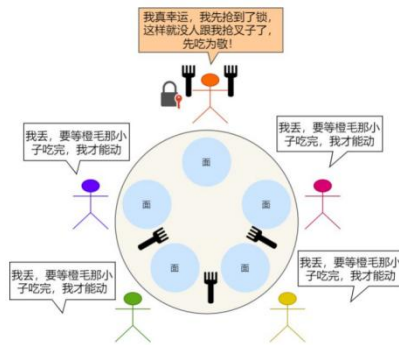
(2) 同步问题：哲学家就餐问题

5 个哲学家围绕着一张圆桌吃面，这个桌子只有 5 支叉子，每两个哲学家之间放一支叉子；哲学家围在一起先思考，思考中途饿了就会想进餐；每个哲学家要两支叉子才愿意吃面，也就是需要拿到左右两边的叉子才进餐；吃完后，会把两支叉子放回原处，继续思考。如何设计方案保证不同的哲学家思考和吃餐的动作有序进行？

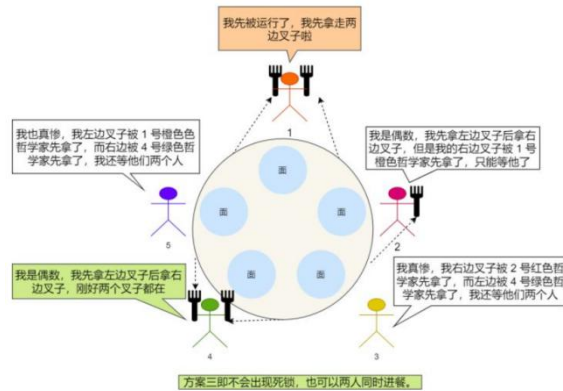


分析问题：哲学家们动作冲突的地方在吃饭时必须拿起左右 2 个叉子，将吃饭的动作作为临界区，吃饭前加锁，吃饭后解锁，就可以使用互斥信号量。

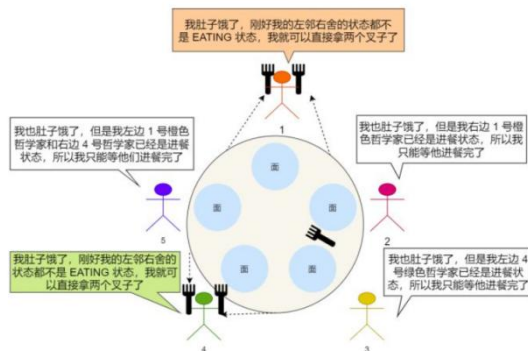
最简单的方案一：使用一个互斥信号量，哲学家吃饭时，其他人必须等待他解锁后才能吃饭。这样每次只能有一个哲学家吃饭。



方案二：让偶数编号的哲学家「先拿左边的叉子后拿右边的叉子」，奇数编号的哲学家「先拿右边的叉子后拿左边的叉子」



方案三：用一个数组 **state** 来记录每一位哲学家在进程、思考还是饥饿状态（正在试图拿叉子），一个哲学家只有在两个邻居都没有进餐时，才可以进入进餐状态。



第 4 章 文件系统

文件系统是操作系统中负责管理持久数据的子系统，简单点说，文件系统负责把用户的文件存到磁盘硬件中。

文件是进程创建的信息逻辑单元。一个磁盘中一般含有数千甚至上万个文件，文件本质上也是一种地址空间。进程可以读取已存在的文件，也可以创建新的文件，文件是受操作系统管理的。文件是操作系统中一个非常重要的概念，特别是 Linux 系统的文件系统，充分体现了“一切皆文件”的设计哲学。

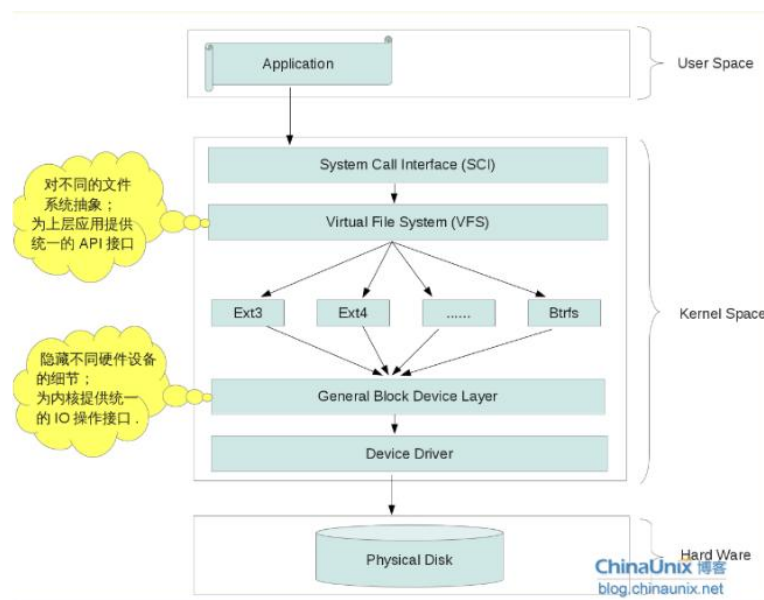
磁盘中的文件是一种持久化的数据。内存中的数据只能在程序运行时存储，所以计算机关闭电源后，内存中的数据都会消失，只有磁盘等外部存储器中的数据可以以文件的形式长期存在。

请注意，一个文件系统目录本质也是一种文件，但是为了做出区分，下文还以文件/目录的形式来叙述。

一、Linux 的文件系统结构

文件系统是对存储设备上的文件进行组织管理的机制，组织方式不同，就会形成不同的文件系统。

文件系统位于操作系统的内核空间，和大部分操作系统一样，Linux 系统支持多种类型的文件系统，如 Ext3、Ext4、Btrfs 等，大多数 Linux 发行版本默认使用 Ext4。同时 Linux 系统使用一个虚拟文件系统（VFS）层来支持不同类型的文件系统，以为上层服务提供统一的调用接口。如下图

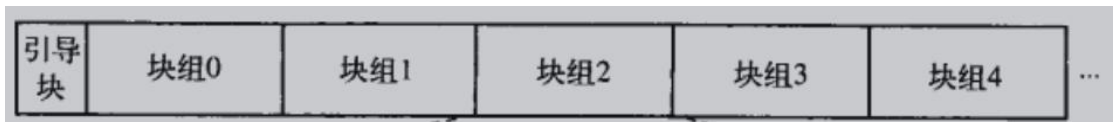


二、Linux 的文件系统实现

1、硬盘的特点

硬盘的优点是容量大、成本低、可以持久化存储数据，缺点是读取效率特别慢，几乎只有内存的十分之一，是计算机中读取速度最慢的设备之一。磁盘读写的最小单位是扇区，每个扇区的大小只有 512B。

磁盘在操作系统中会被映射为一整块的虚拟地址空间，整个虚拟地址可以被划分成不同的块组（硬盘分区）。块组可以是同一块硬盘分区得来，也可以是计算机添加了更多的外部存储器后产生。如下，引导块存放引导计算机启动的启动项，块组 0 可能就是操作系统文件所在的地址区域（C 盘）。



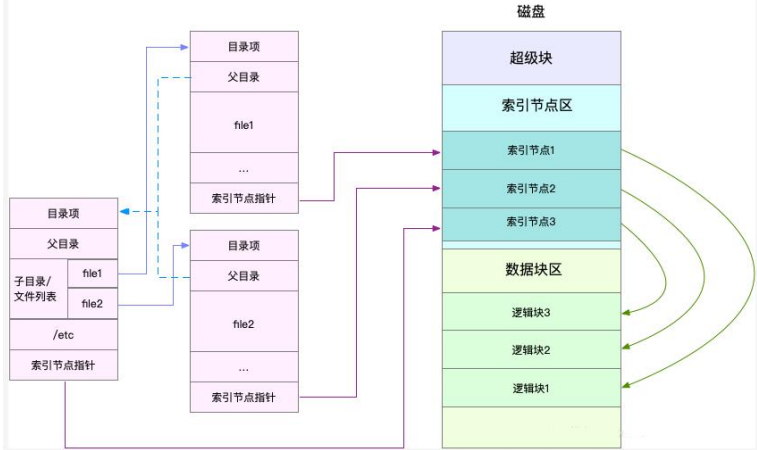
每个块组对应一个文件系统，不同的块组可以使用不同类型的文件系统。

每个块组在物理上包含多个扇区，块组上的文件系统将多个扇区看做一个个逻辑块（也叫数据块），文件系统每次读写的最小单位就是逻辑块，Linux 中的逻辑块大小为 4KB（一次性读写 8 个扇区，提高了读写效率）。

2、文件系统的四个要素

一个块组对应的文件系统可以划分为三个区域：超级块（super block）、索引节点区（index node block）、数据块区（data block）。

区域	存储的信息类型	主要特点
超级块	存储文件系统的初始化信息	例如索引节点的个数、空闲块链表的起始位置等，是整个文件系统的地图或心脏，程序初始化一个文件系统时首先读取的就是超级块中的数据，如果文件系统初始化失败，很可能就是超级块出现了问题。
索引节点区	存储所有文件的描述信息	存储多个索引节点（inode 号，文件大小、访问权限等）
数据块区	存储所有文件的具体内容	与每个索引节点一一映射



Linux 中文件系统的初始化：

（1）Linux 的系统程序默认在超级块中找到索引节点区的起始信息，根据 inode 号=0 索引到根目录的索引节点所在位置，在索引节点区找到该索引节点读取到内存中，然后为其创建一个目录项，此时内存中已经初始化一个根目录文件的缓存。

（2）根据根目录的目录项找到下一级目录或文件的名字与 inode 号，不断重复直至初始化完成，此时内存中保存了该目录的目录结构。

可以发现，文件系统的初始化就是从超级块和索引节点区中读取信息在内存中建立多个目录项的过程，所有目录项的数据会被保存到内存的缓存区中，这保证了文件系统的目录结构能够被缓存，所以用户访问文件系统的目录结构会非常快捷。

超级块和索引节点区的数据不会全部加载到内存，只有当需要的时候才将其加载进内存：当初始化文件系统时加载超级块数据进入内存；当访问某个文件时加载索引节点区的数据进入内存。

在初始化过程中，Linux 文件系统会为每个文件分配 2 个数据结构：索引节点（index node）和目录项（directory entry），它们的区别如下：

	存储数据	关系	存储位置
索引节点	Inode 编号、文件大小、访问权限、创建时间、修改时间、数据在磁盘中的位置	一个索引节点唯一标识一个文件	存储在磁盘中，占据磁盘空间，但为了加速文件的访问，通常也会把索引节点加载到内存中

目录项	文件名称、索引节点指针、与其他目录项的层级关系	一个索引节点可能对应多个目录项， 寄一个文件可能有多个别名	存储在内存中，占据缓存空间
-----	-------------------------	----------------------------------	---------------

目录和目录项的区别：目录是一个磁盘中的文件，存放在磁盘中，需要用索引节点唯一标识，普通文件保存文件数据，目录文件保存子目录或文件信息；目录项是由内核程序维护的一个数据结构，存放在内存中，多个目录项共同构成文件系统在内存中缓存的目录结构。

综上，Linux 系统中的文件系统的实现可以看做是由四个基本要素组成的：超级块、索引节点、数据块、目录项。

3、虚拟文件系统 VFS

虚拟文件系统 VFS（Virtual File System）对高层进程和应用程序隐藏了 Linux 支持的所有文件系统之间的区别，为上层服务提供统一的系统调用接口。下图则是 Linux 系统文件系统的架构图。



通过这张图可以看到，VFS 定义了一组所有文件系统都支持的数据结构和标准接口。这样，用户进程和内核中的其他子系统，就只需要跟 VFS 提供的统一接口进行交互。在 VFS 的下方，Linux 支持各种各样的文件系统，如 Ext4、XFS、NFS 等等。为了降低慢速磁盘对性能的影响，文件系统又通过页缓存、目录项缓存以及索引节点缓存，缓和磁盘延迟对应用程序的影响。

- 按照存储位置的不同，这些文件系统可以分为三类。
- 第一类是基于磁盘的文件系统，也就是把数据直接存储在计算机本地挂载的磁盘中。常见的 Ext4、XFS、OverlayFS 等，都是这类文件系统。
 - 第二类是基于内存的文件系统，也就是我们常说的虚拟文件系统。这类文件系统，不需要任何磁盘分配存储空间，但会占用内存。我们经常用到的 /proc 文件系统，其实就是一种最常见的虚拟文件系统。此外，/sys 文件系统也属于这一类，主要向用户空间导出层次化的内核对象。
 - 第三类是网络文件系统，也就是用来访问其他计算机数据的文件系统，比如 NFS、SMB、iSCSI 等。
- 不同的文件系统想要在 Linux 系统上运行，要先挂载到 VFS 目录树中的某个子目录(称为挂载点)，然后才能访问其中的文件。拿第一类，也就是基于磁盘的文件系统为例，在安装系统时，要先挂载一个根目录(/)，在根目录下再把其他文件系统(比如其他的磁盘分区、/proc 文件系统、/sys 文件系统、NFS 等)挂载进来。

把文件系统挂载到挂载点后，你就能通过挂载点，再去访问它管理的文件了。VFS 提供了一组标准的文件访问接口。这些接口以系统调用的方式，提供给应用程序使用。

目录	内容
bin	二进制（可执行）文件
dev	I/O设备文件
etc	各种系统文件
lib	库
usr	用户目录

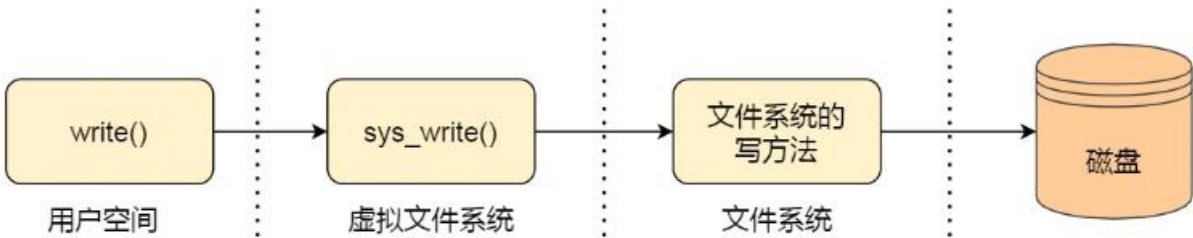
图10-23 大部分Linux系统中一些重要的目录

4、Linux 的文件系统调用

从程序员角度看文件的话，程序员一般需要通过系统调用函数来打开/操作一个文件，例如：

```
fd = open(name, flag); # 打开文件
...
write(fd,...);        # 写数据
...
close(fd);            # 关闭文件
```

系统调用 write()在操作系统的内核程序中会发生如下调用：



程序打开一个文件后，操作系统会为该程序所在的进程维护一个打开文件表，打开文件表存储了每个打开文件的文件描述符，同时维护打开文件的状态和信息，例如文件指针、文件打开计数器、文件磁盘位置、访问权限等。

用户进程对某个文件的读写如下：

当用户进程从文件读取 1 个字节大小的数据时，文件系统则需要获取字节所在的数据块，再返回数据块对应的用户进程所需的数据部分。

当用户进程把 1 个字节大小的数据写进文件时，文件系统则找到需要写入数据的数据块的位置，然后修改数据块中对应的部分，最后再把数据块写回磁盘。

所以说，文件系统的基本操作单位是数据块。

Linux 提供的系统调用很多与文件系统有关，提供的针对单个文件和目录相关的系统调用：

系统调用	描述
s=mkdir (path, mode)	建立新目录
s=rmdir (path)	删除目录
s=link (oldpath, newpath)	创建指向已有文件的链接
s=unlink (path)	取消文件的链接
s=chdir (path)	改变工作目录
dir=opendir (path)	打开目录
s=closedir (dir)	关闭目录
dirent=readdir (dir)	读取一个目录项
rewinddir (dir)	回转目录使其再次被读取

图10-29 与目录相关的一些系统调用。如果发生错误，那么返回值s是-1，dir是一个目录流，dirent是一个目录项。参数的含义是自解释的

系统调用	描述
fd = creat(name,mode)	创建新文件的一种方法
fd = open(file, how,...)	打开文件读、写或者读写
s = close(fd)	关闭一个已经打开的文件
n = read(fd, buffer, nbytes)	从文件中读取数据到一个缓冲区
n = write(fd, buffer, nbytes)	把数据从缓冲区写到文件
position = lseek(fd, offset, whence)	移动文件指针
s = stat(name, &buf)	获取一个文件的状态信息
s = fstat(fd, &buf)	获取一个文件的状态信息
s = pipe(&fd[0])	创建一个管道
s = fcntl(fd, cmd,...)	文件加锁及其他操作

图10-27 跟文件相关的一些系统调用。如果发生错误，那么返回值s是-1，fd是一个文件描述符，position是文件偏移。参数的含义是很清楚的

第 5 章 I/O 系统

操作系统的几个概念的区分：，操作系统本质上就是一个周而复始顺序执行的死循环，其不断运行中等待某种事件的发生

事件：操作系统中的消息

信号：操作系统用来管理进程时的异步事件，这个信号一般来自其他进程

中断：计算机中指出意外情况（多为 I/O 操作），需要 CPU 干预，CPU 会暂停手中工作，停下来专门处理中断，处理完毕后继续执行手中工作。中断分为软件中断和硬件中断。

陷入（trap）：本质是一种软件中断，用户程序执行系统调用时必须让 CPU 从用户态切换到内核态的信号。

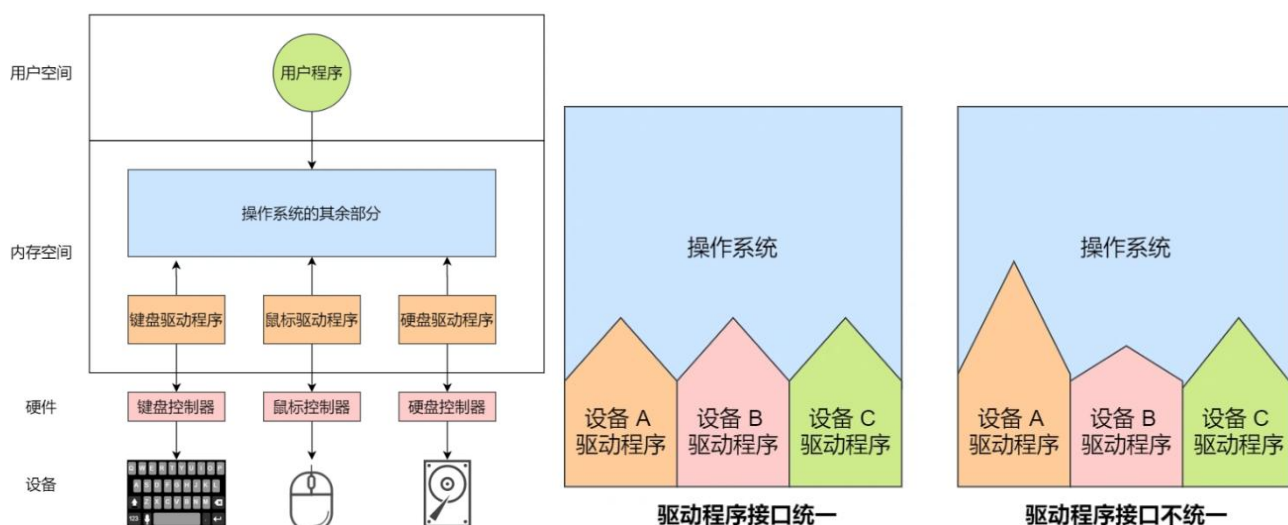
异常：CPU 在执行程序指令时发现程序指令有错误而发出的信号

一、I/O 设备管理

I/O 设备也叫做**输入/输出设备**。I/O 设备可以分为两大类：

- **块设备**：以**块**为单位发送或者接收固定大小的数据，**支持寻址**。例如**硬盘**、U 盘、光盘等。
 - **字符设备**：以**字符**为单位发送或者接受字符流，**不支持寻址**。例如**鼠标**、**键盘**、**网卡**、打印机等。
- 后文会讲到，CPU 驱动**块设备**的方式和**字符设备**的方式大不同。

1、设备控制器和设备驱动程序

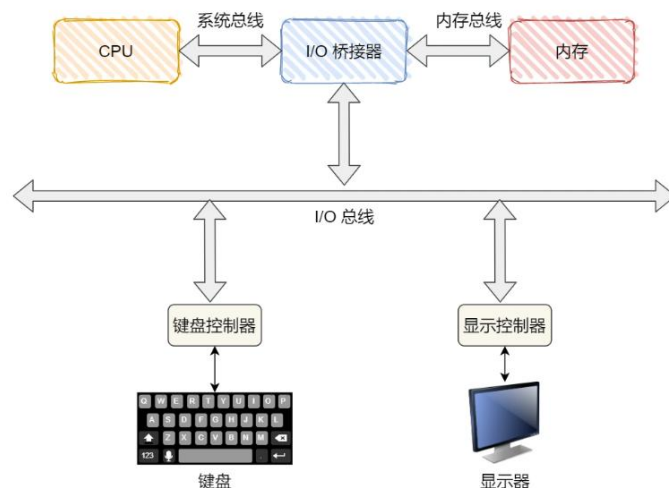


每一个 I/O 设备在**硬件**上都有一个对应的「**设备控制器**」（Device Controller），也叫适配器（adapter），「**设备控制器**」屏蔽了硬件设备的众多细节，减少 CPU 对 I/O 的干预。它本质是 I/O 设备的一种电子组件，由芯片、寄存器和电路组成。块设备传输的数据量很大，通常其设备控制器还集成一个可读写的**数据缓冲区**。

每一个 I/O 设备在软件上都有一个对应的「**设备驱动程序**」，「**设备驱动程序**」属于操作系统层面，它屏蔽了不同**设备控制器**的差异。I/O 设备提供商会针对不同操作系统开发专用的「**设备驱动程序**」，它们会实现该操作系统规定的 API 接口，这样不同的「**设备驱动程序**」就可以以相同或者近似的方式接入操作系统。设备驱动程序在接入 I/O 设备后就会默认安装或者由用户手动安装到操作系统上。

2、I/O 设备控制方式

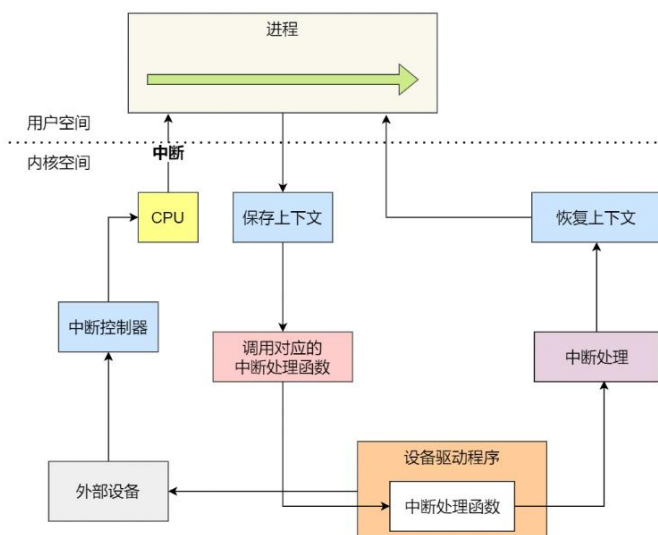
我们先看计算机中 CPU 和 I/O 设备的架构图：



在上图中，CPU 和每个 I/O 设备之间存在硬件上的 I/O 总线来连接，CPU 如何控制 I/O 设备的读写呢？

方式一：「**程序 I/O 方式**」。在早期计算机系统中，CPU 需要**轮训等待**（也叫**忙等待**）。CPU 定期执行每个 I/O 设备驱动程序，向每个 I/O 设备发出询问，检查其是否需要请求需要处理，该过程中会占据 CPU 资源，同时由于 CPU 的高速性和 I/O 设备的低速性，会造成 CPU 资源的大量浪费，现代操作系统中已经逐渐淘汰。

方式二：「**中断驱动 I/O 方式**」。当 I/O 设备需要处理读写任务时，就会触发**设备控制器**中的**中断控制器**，发送一个**中断请求**给 CPU，CPU 接收到**中断**后就会暂停当前的进程，调用**设备驱动程序**的**中断处理程序**来专门处理**中断请求**。



在 I/O 设备输入每个数据的过程中，由于无需 CPU 干预，因而可使 CPU 与 I/O 设备并行工作，仅当完成一个数据输入时，才需 CPU 花费极短的时间去做一些中断处理。这种驱动方式以**字节**为单位进行 I/O，适合**字符设备**，例如键盘、鼠标。

请注意，中断并不是处理 I/O 设备特有的，**中断**分为**软件中断**和**硬件中断**。软件中断是 CPU 主动执行一条用于中断的 CPU 指令，例如进行系统调用时。

我们以**用户在键盘输入信息**为例，介绍**中断**过程：

当用户在键盘上敲击字符时，**键盘控制器**会产生一种叫做扫描码的数据，将其缓存到**键盘控制器**的寄存器中，然后**键盘控制器**通过 I/O 总线向 CPU 发送**中断请求**。

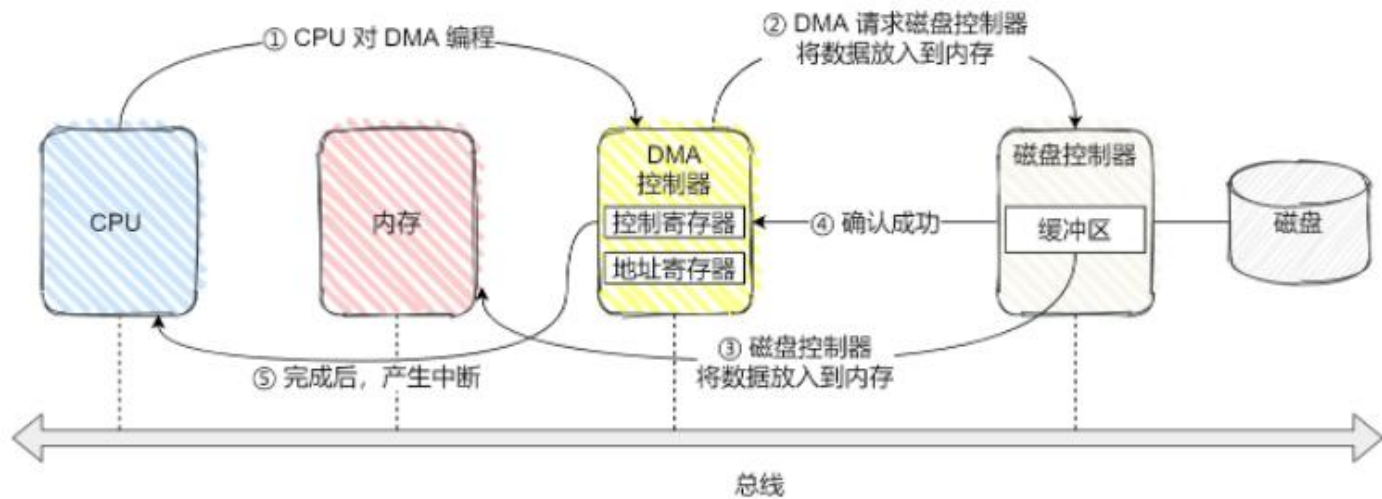
CPU 接收到**中断请求**后，操作系统会保存此时正在执行的进程的 CPU 上下文，然后调用**键盘驱动程序**中的**中断处理程序**：该程序会从**键盘控制器的缓冲区**读取扫描码，再根据扫描码找到用户在键盘输入的字符，将其翻译成对应字符对应的 ASCII 码，把 ASCII 码放到**内存**中的「**读缓冲区队列**」。

显示设备的**驱动程序**会定时从内存的「**读缓冲区队列**」读取数据放到「**写缓冲区队列**」，最后把「写缓冲区队列」的数据一个一个写入到显示设备的控制器的寄存器中的**数据缓冲区**，最后将这些数据显示在屏幕里。屏幕显示

出结果后，CPU 恢复被中断进程的上下文。

方式三：「直接存储器访问（DMA）I/O 方式」。中断 I/O 适合于键盘、鼠标等字符设备，对于硬盘等块设备的 I/O 是非常低效的，因为块设备传输的一般是文件形式的大量数据，每次传输以数据块为单位，而中断 I/O 以字节为单位进行中断。该控制方式依赖于一个叫做 DMA 控制器的硬件设备。

DMA（Direct Memory Access，直接存储器访问）控制器可以使块设备在 CPU 不参与的情况下完成将数据从块设备的缓存区放入到内存的缓存区中。DMA 的工作方式如下：



- CPU 对 DMA 控制器下发指令，告诉它想读取多少数据，读完的数据放在内存的某个地方就可以了；
- 接下来，DMA 控制器会向磁盘控制器发出指令，通知它从磁盘读数据到其内部的缓冲区中，接着磁盘控制器将缓冲区的数据传输到内存；
- 当磁盘控制器把数据传输到内存的操作完成后，磁盘控制器在总线上发出一个确认成功的信号到 DMA 控制器；
- DMA 控制器收到信号后，DMA 控制器发中断通知 CPU 指令完成，CPU 就可以直接用内存里面现成的数据了；

可以看到，CPU 当要读取磁盘数据的时候，只需给 DMA 控制器发送指令，然后返回去做其他事情，当磁盘数据拷贝到内存后，DMA 控制机器通过中断的方式，告诉 CPU 数据已经准备好了，可以从内存读数据了。仅仅在传送开始和结束时需要 CPU 干预。

3、I/O 设备的缓存管理

在进行 I/O 设备管理时，CPU 处理数据速度非常快，I/O 设备处理速度非常慢，为了缓和 CPU 和 I/O 设备之间速度不配的矛盾，操作系统的内核空间也使用了缓存区。

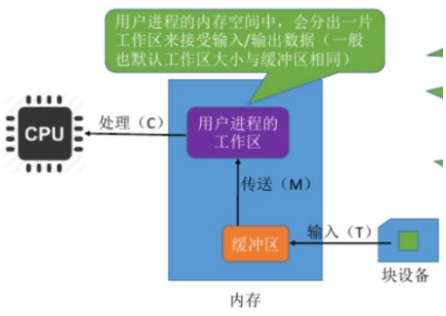
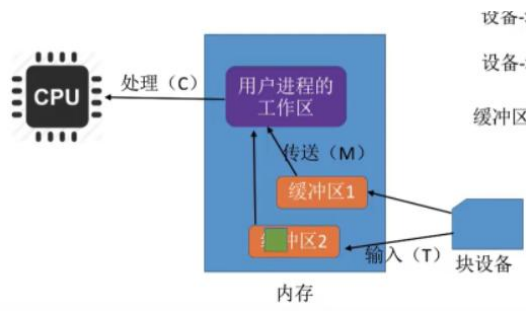
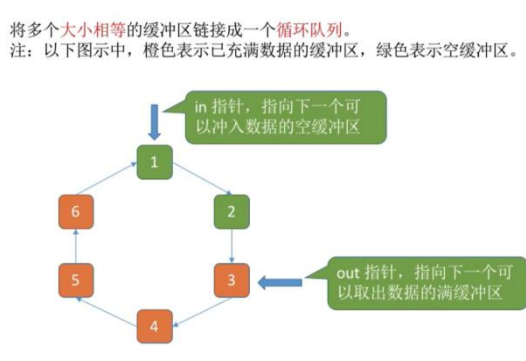
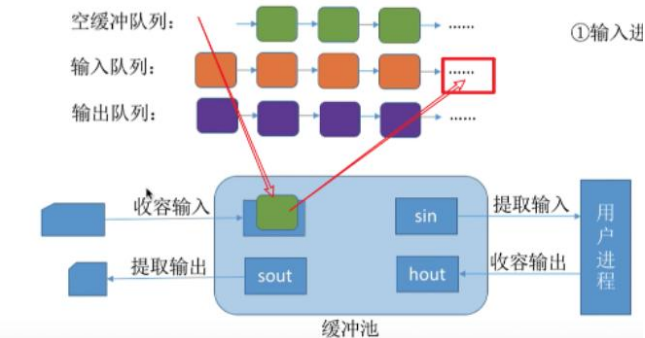
缓存（cache）是一个经常用到的概念，它也在很多地方出现：CPU 中存在缓存（各种寄存器）、I/O 设备的设备控制器中存在缓存（鼠标、键盘的寄存器、磁盘的缓存区）、内核空间中的内存也会开辟专门的缓存区，用户空间中的应用程序也可能存在缓存区。本小节我们只介绍操作系统的内核空间如何进行缓存管理。

内核空间中使用数据缓存区可以具有如下好处：

- ① 缓和 CPU 和 I/O 设备间速度不匹配的矛盾。
- ② 减少对 CPU 的中断频率，放宽对 CPU 中断响应时间的限制。
- ③ 提高 CPU 和 I/O 设备之间的并行性。

根据缓存管理的方式，内核缓存区可以分为如下：

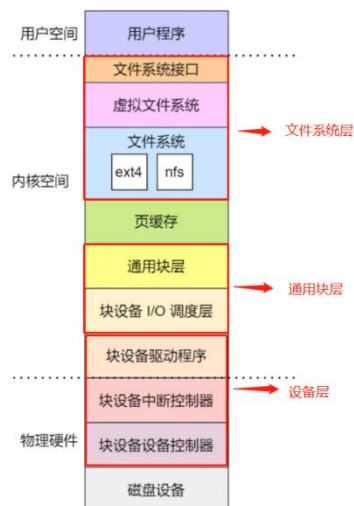
	类型	备注	
--	----	----	--

单缓存	每当用户进程发出一个 I/O 请求时，操作系统便在内存中为之分配一个缓冲区，只有当这个缓存区满时，CPU 才可以读取数据，CPU 全部提取完后 I/O 设备才可以继续写入。	单个缓存区的 读写不能同时进行 。	
双缓存	I/O 设备输入时，先写入第一个缓冲区，装满后再写入第二个缓冲区，此时 CPU 可以读第一个写好的缓冲区。	读写可以同时进行， 适合读写速度接近 的情景	
循环缓存	循环队列形式，通用用于输入进程和计算进程的关系中，输入进程不断向循环队列中添加数据，计算进程不断取出数据来计算。	适合读写速度差异较大的情景，一般是 某些进程专用 的缓存形式	<p>将多个大小相等的缓冲区链接成一个循环队列。 注：以下图示中，橙色表示已充满数据的缓冲区，绿色表示空缓冲区。</p> 
缓存池	缓存池中存在三个队列： 空缓冲队列、输入队列、输出队列 。同时存在四种工作缓冲区。缓冲区可以工作在收容输入、提取输入、收容输出、提取输出四种工作方式下。	缓存区的利用率较大，可供 多个普通进程共享	

5、Linux 中存储系统分层

操作系统的**文件系统**和**块设备**（磁盘、U 盘等）的 I/O 可以暂且统称为 **Linux 的存储系统**。可以把 Linux 存储系统的 I/O 由上到下可以分为三个层次，分别是**文件系统层**、**通用块层**、**设备层**。这三个层次的作用是：

- **文件系统层**：向上为**应用程序**统一提供了标准的**文件访问接口**，向下通过**通用块层**来存储和管理磁盘数据。
- **通用块层**：向上提供访问块设备**的标准接口**，向下管理不同的设备驱动程序。**Linux 中特有的块设备抽象层**，同时对来自上层的 I/O 请求进行 **I/O 调度**，按照合适的顺序执行 I/O 请求。
- **设备层**：负责执行最终**物理设备**的 I/O 操作。



二、I/O 模型

文件读写方式的各种差异，导致 I/O 的分类多种多样。最常见的有，缓冲与非缓冲 I/O、直接与非直接 I/O、阻塞与非阻塞 I/O、同步与异步 I/O 等。接下来，我们就详细看这四种分类。

1、缓冲 I/O 与非缓冲 I/O

根据**是否利用标准库缓存**，可以把文件 I/O 分为**缓冲 I/O**与**非缓冲 I/O**。**缓冲 I/O**，是指利用标准库缓存来加速文件的访问，而标准库内部再通过系统调度访问文件。**非缓冲 I/O**，是指直接通过系统调用来访问文件，不再经过标准库缓存。

注意，这里所说的“缓冲”，是指标准库内部实现的缓存。比方说，你可能见到过，很多程序遇到换行时才真正输出，而换行前的内容，其实就是被标准库暂时缓存了起来。无论缓冲 I/O 还是非缓冲 I/O，它们最终还是要经过系统调用来访问文件。我们知道，系统调用后，还会通过**页缓存**，来减少磁盘的 I/O 操作。

2、直接 I/O 与非直接 I/O

根据**是否利用操作系统的页缓存**，可以把文件 I/O 分为**直接 I/O**与**非直接 I/O**。

- **直接 I/O**：文件读写时，跳过操作系统的页缓存，直接跟文件系统交互来访问文件。
- **非直接 I/O**：文件读写时，先经过操作系统的页缓存，然后再由内核或额外的系统调用，真正写入磁盘。

什么是**页缓存**？

Linux 内核为了减少磁盘 I/O 次数，在系统调用后，会把用户数据拷贝到内核中缓存起来，这个内核缓存空间也就是「**页缓存**」，只有当缓存满足某些条件的时候，才发起磁盘 I/O 的请求。

使用页缓存有两个优势：

- (1) 内核会对在页缓存操作的 I/O 请求进行 I/O 调度，减少磁盘开销。
- (2) 内核在页缓存的 I/O 请求存在预读功能。

直接 I/O 和非直接 I/O 本质上还是要通过文件系统访问磁盘的文件。在数据库等场景中，裸 I/O 会跳过文件系统读写磁盘的情况，也就是我们通常所说的。想要实现直接 I/O，需要你在系统调用中，指定 `O_DIRECT` 标志。如果没有设置过，默认的是非直接 I/O。

直接 I/O 的常见应用场景如下：

- 应用程序已经实现了磁盘数据的缓存，那么可以不需要 PageCache 再次缓存，减少额外的性能损耗。在 MySQL 数据库中，可以通过参数设置开启直接 I/O，默认是不开启；

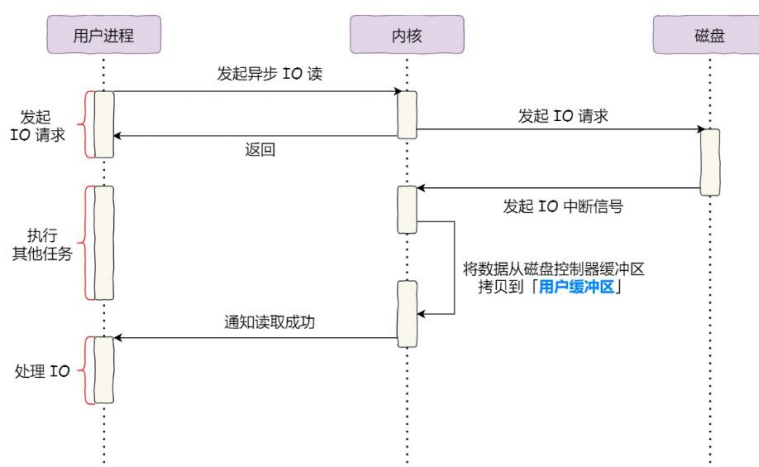
传输大文件。由于大文件难以命中 PageCache 缓存，而且会占满 PageCache 导致「热点」文件无法充分利用缓存，从而增大了性能开销，因此，这时应该使用直接 I/O。

3、同步 I/O 和异步 I/O

根据**是否等待读写的响应结果**，可以把**文件 I/O**分为**同步 I/O**和**异步 I/O**。

- **同步 I/O**：是指应用程序执行 I/O 操作后，要一直等到整个 I/O 完成后，才能获得 I/O 响应。
- **异步 I/O**：是指应用程序执行 I/O 操作后，不用等待，可以继续执行。等到这次 I/O 完成后，内核会用事件通知的方式，告诉应用程序。

异步 I/O不需要等待「**内核数据准备好**」和「**数据从内核空间拷贝到用户空间**」这两个过程。**异步 I/O**就只支持**直接 I/O**，因为它是将数据从**磁盘控制器缓冲区**拷贝到**用户缓存区**，中间不涉及**页缓存**。例如当我们调用 `aio_read` 之后，内核自动将数据从 I/O 设备的缓存区读取到内存缓存区，再从内存缓存区读取到应用程序的缓存区，这个过程由内核程序自动完成，应用程序并不需要主动拷贝数据。

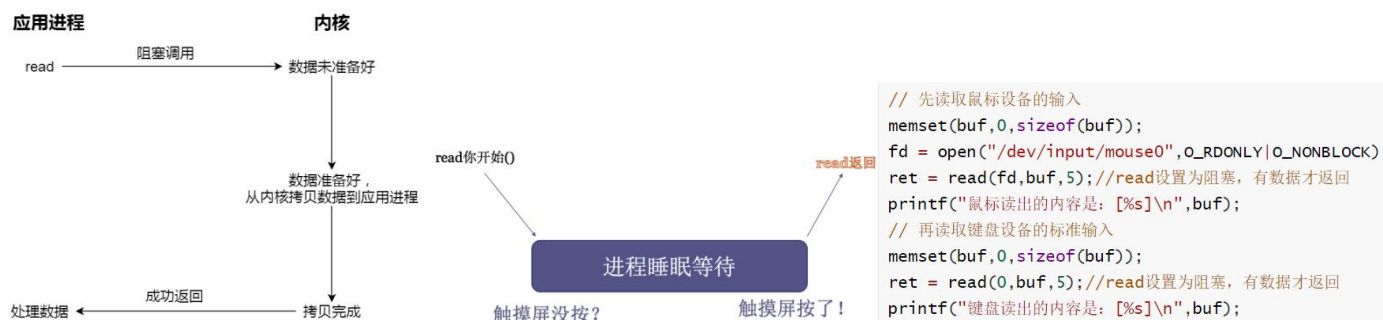


例如，在操作文件时，如果设置了 `O_SYNC` 或者 `O_DSYNC` 标志，就代表同步 I/O。如果设置了 `O_DSYNC`，就要等文件数据写入磁盘后，才能返回；而 `O_SYNC`，则是在 `O_DSYNC` 基础上，要求文件元数据也要写入磁盘后，才能返回。再比如，在访问管道或者网络套接字时，设置了 `O_ASYNC` 选项后，相应的 I/O 就是异步 I/O。这样，内核会再通过 `SIGIO` 或者 `SIGPOL`，来通知进程文件是否可读写。

4、阻塞 I/O、非阻塞 I/O、I/O 复用

根据**应用程序是否阻塞自身运行**，可以把文件 I/O 分为**阻塞 I/O**和**非阻塞 I/O**

- **阻塞 I/O**：是指执行的系统调用如果没有获得响应就会阻塞当前线程，被操作系统挂起，直到等待的事件发生为止。



【优点】：在阻塞等待过程中进程被挂起，不消耗 CPU 资源，提升程序性能

【缺点】：阻塞 I/O 只能串行，不适合大量并发场景

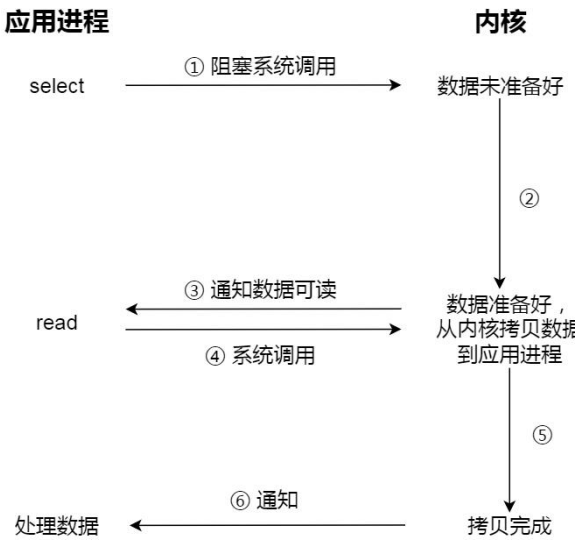
【应用场景】适合串行场景，例如各种编程语言提供的 `wait()`、`pause()`、`sleep()` 等函数。

- **非阻塞 I/O (也叫并发式 I/O)**：是指应用程序执行的系统调用则总是立即返回，不会阻塞当前的线程，而不管事件是否已经发生。如果事件没有立即发生，这些系统调用就返回-1，和出错的情况一样。此时程序员必须根据 `errno` 来区分这两种情况。



- 【常见优点】 适合用于并发场景
- 【常见缺点】 非阻塞 I/O 需要不断检查，可能会消耗大量 CPU 资源
- 【应用场景】 非阻塞 I/O 适用于并发场景，但一般不会单独使用，要和其他 I/O 通知机制一起使用，比如 I/O 复用和 SIGIO 信号。

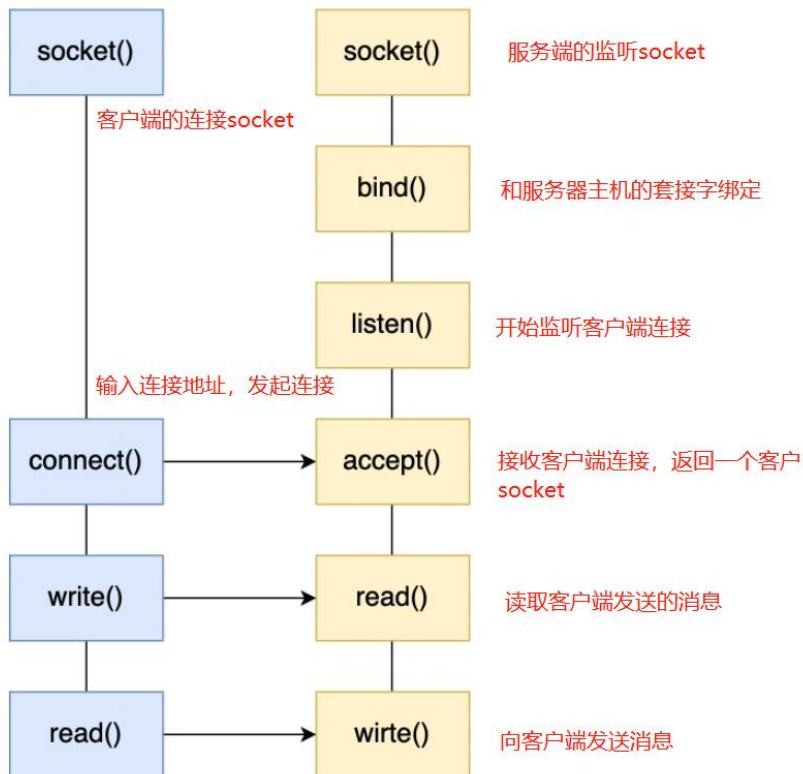
上述图示是**非阻塞 I/O**采用轮询方式询问内核程序中的 I/O 数据是否准备好，在并发场景中效率会大大降低，所以基于**非阻塞 I/O**的**I/O 复用**技术就出现了，它将**I/O 事件**分发出去，以**事件**形式通知应用程序，极大提供了对 CPU 的利用率，如下是一个**I/O 复用**模型 `select` 的过程：



- 请注意，**阻塞 I/O**、**非阻塞 I/O**、**I/O 复用**都是一种同步 I/O，因为它们在执行 `read` 调用时都需要等待「**数据从内核空间拷贝到用户空间**」。
- 阻塞 I/O** 需要等待的是「**内核数据准备好**」和「**数据从内核空间拷贝到用户空间**」这 2 个过程。
- 非阻塞 I/O** 和 **I/O 复用**需要等待的是「**数据从内核空间拷贝到用户空间**」这 1 个过程。
- 真正的**异步 I/O**不需要等待「**内核数据准备好**」和「**数据从内核空间拷贝到用户空间**」这 2 个过程。
- 比方说，访问管道或者网络套接字时，设置 `O_NONBLOCK` 标志，就表示用非阻塞方式访问；而如果不做任何设置，默认的就是阻塞访问。

三、I/O 多路复用

想要在不同主机间的进程间通信就必须使用 `socket` 编程。



`socket()`创建一个 `listened_socket`，需要指定网络协议为 IPv4 还是 IPv6

`bind()`给 `listened_socket` 绑定一个套接字（IP 地址和端口号）:绑定 IP 地址就是和对应网卡的 IP 地址绑定，一台主机可能有多个网卡；绑定端口号就是选用一个空闲端口号作为当前程序的唯一标识。

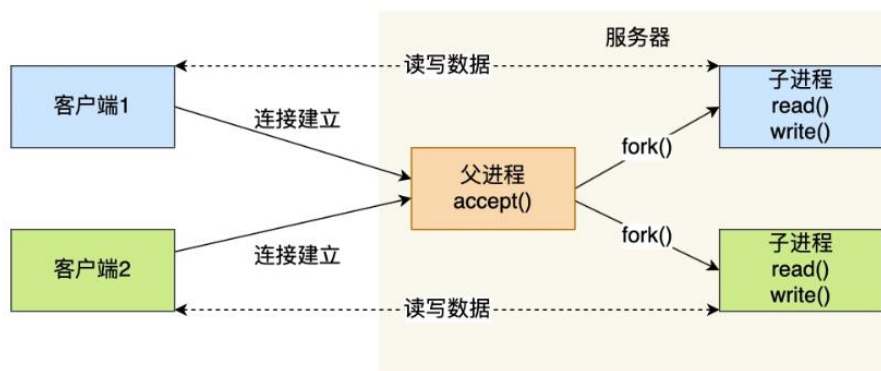
`listen()`启动当前网络程序来监听绑定的端口号，这个可以作为判定一个网络程序是否启动的条件

`accept()`从内核程序中来获取客户端的连接，默认采用阻塞 I/O，即没有客户端连接返回就会一直阻塞。

当客户端程序调用 `connect()`函数来连接指定地址的主机时，客户端和服务端的 TCP 三次握手就开始了，

当内核程序接收到一个客户端连接时，返回一个已经完成连接的 `Socket`。

上述模型是最基本的网络通信模型，它默认采用了同步阻塞 I/O，基本只能用来一对一通信。

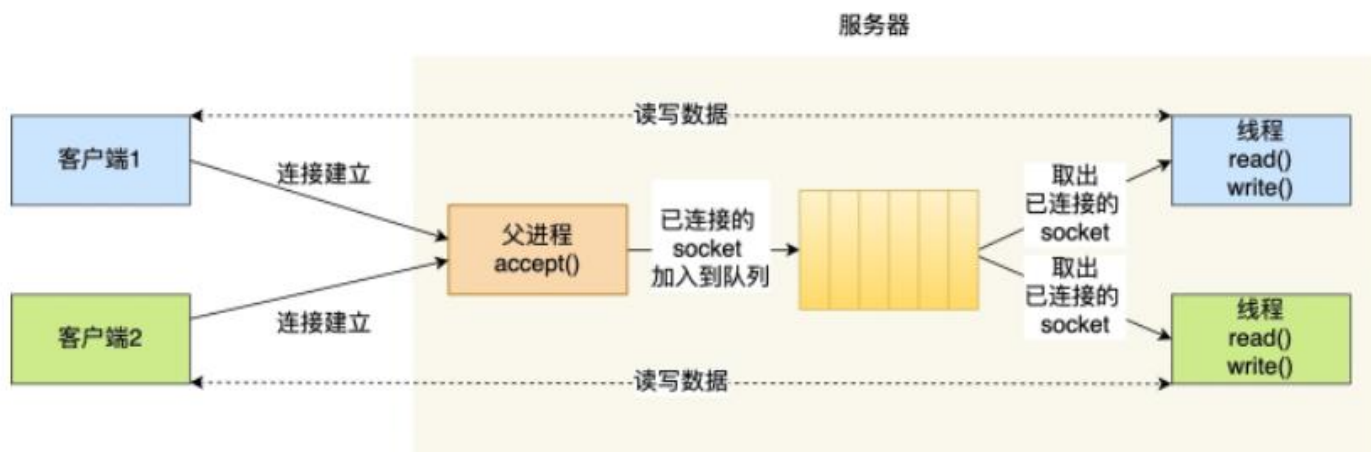


1、多进程模型

多进程模型就是为每个客户端分配一个进程来处理请求。服务器的主进程负责监听客户的连接，一旦与客户端连接完成，`accept()`函数就会返回一个「已连接 `Socket`」，这时就通过 `fork()`函数创建一个子进程，子进程专心处理和「已连接 `Socket`」的通信。

该模型的不足：创建进程的数量是有上限的（一般不大于 100 个）；频繁创建进程和销毁进程会严重增加系统开销；如果子进程没有做好回收就会成为僵尸进程，最终耗尽系统资源。

2、多线程模型



主进程将接收到的「已连接 Socket」放入一个全局队列，然后使用 `pthread_create()` 来创建线程处理和每个「已连接 Socket」的通信，为了避免频繁地创建和销毁线程，可以使用线程池技术（提前创建若干个线程），同时为了避免多线程竞争，需要互斥锁机制。

该模型的不足：创建线程的数量是有上限的（一般不大于 100 个）。

3、I/O 复用

I/O 复用模型可以在一个进程中维护多个「已连接 Socket」的通信，原理就是多路复用技术：

一个进程虽然任时刻只能处理一个请求，但是处理每个请求的事件时，耗时控制在 1 毫秒以内，这样 1 秒内就可以处理上千个请求，把时间拉长来看，多个请求复用了个进程，这就是多路复用，这种思想很类似一个 CPU 并发多个进程，所以也叫做时分多路复用。

阻塞 I/O 需要多进程/线程才能用于并发场景，普通的非阻塞 I/O 需要在应用程序中不断检查才能用于并发场景，两者在并发场景中都不高效。I/O 复用是并发场景中最常使用的 I/O 通知机制。I/O 复用的原理是：

- 应用程序通过 I/O 复用函数向内核程序注册一组事件集合；
- 内核程序不断检查事件集合，通过 I/O 复用函数把其中就绪的事件通知给应用程序。

本质上 IO 复用将普通的非阻塞 I/O 在应用程序中的轮询通过 I/O 复用函数交给了内核程序，所以应用程序可以在一个进程/线程中同时处理多个 I/O 操作。Linux 上常用的 I/O 复用函数是 `select`、`poll` 和 `epoll_wait`。

需要指出的是，I/O 复用函数本身是阻塞的，它们能提高程序效率的原因在于它们可以在同一个线程/进程中具有同时监听多个 I/O 事件的能力，所以 IO 复用可以解决阻塞 I/O 不能用于大量并发的缺陷。

【常见优点】非常适合用于并发场景

【常见缺点】实现复杂。

【应用场景】网络 socket。

(1) select 和 poll

`select` 和 `poll` 实现多路复用的方式是：

- （一）将已连接 socket 都放到一个文件描述符集合，
- （二）然后调用 `select/poll` 函数，进入阻塞等待。（该函数将文件描述符集合拷贝到内核程序里，让内核通过遍历来检查是否有网络事件产生，当检查到有事件产生后，将此已连接 socket 标记为可读或可写，接着再把整个文件描述符集合拷贝回用户程序里）
- （三）用户程序需要遍历 `select/poll` 函数返回的集合，分类型处理各种情况。

`select` 和 `poll` 的区别：

`poll` 和 `select` 并没有太大的本质区别，都是使用「线性结构」存储进程关注的 socket 集合，`select` 使用固定长度的 `Bitmap`，表示文件描述符集合，该集合默认最大值是 1024；`poll` 使用动态数组以链表形式来表示文件描述符集合，突破了 `select` 的文件描述符个数限制，当然还会受到系统文件描述符限制。

`select` 和 `poll` 的缺点：

`select` 和 `poll` 的程序需要进行 2 次「遍历」文件描述符集合（一次是在内核态里，一次是在用户态里）；还会 2 次「拷贝」文件描述符集合（从用户空间传入内核空间，由内核修改后，再传出到用户空间），其时间复杂度为 $O(n)$ 。当客户端数量上升时，性能的损耗也会快速增长。

(2) epoll

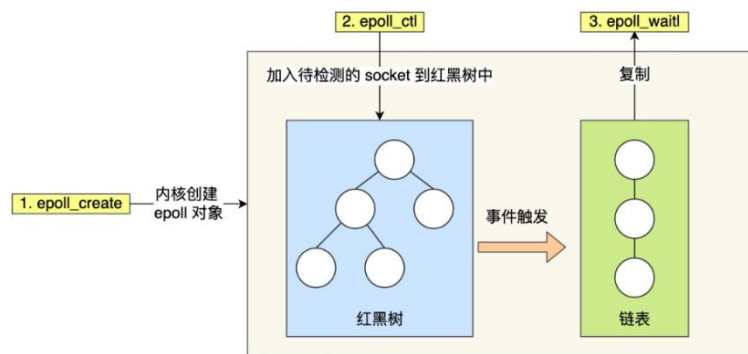
`epoll` 是 Linux 特有的 IO 复用函数，其实现和使用与 `select` 或者 `poll` 有较大差异,其性能也更加高效。`epoll` 机制需要多个系统调用。

- (一) 使用 `epoll_create()` 创建 `epoll` 模型，返回一个 `epoll` 文件描述符，用来唯一标识要访问的内核事件表
- (二) 使用 `epoll_ctl()` 向 `epoll` 模型中注册一个关联 `socket` 的 `epoll` 事件（`socket` 需要设置为非阻塞 I/O）
- (三) 调用 `epoll_wait()` 来阻塞监听。`epoll_wait` 函数如果检测到事件，就将所有就绪的事件从内核程序的内核事件表中复制用户程序。
- (四) 遍历返回的就绪事件，分情况处理不同的 `epoll` 事件。

`epoll` 通过如下两个方面很好解决了 `select/poll` 的问题：

第一点，`epoll` 在内核程序中使用红黑树来跟踪进程所有待检测的 `socket`，红黑树是个高效的数据结构，增删查一般时间复杂度是 $O(\log n)$ ，而且其每次操作都只需要传入一个待检测的 `socket`，减少了内核和用户空间大量的数据拷贝和内存分配。

第二点，`epoll` 在内核程序中使用事件驱动机制来返回就绪的 `socket`，内核维护了一个链表来记录就绪事件，当用户调用 `epoll_wait()` 函数时，只会返回所有的就绪事件。



`epoll` 支持两种事件触发模式：LT 模式（Level Trigger，水平触发）和 ET 模式（Edge Trigger，边缘触发）。

LT: `epoll` 的默认工作模式。`epoll_wait` 每个事件会通知多次，应用程序可以不立即处理，只要满足事件条件，比如内核中有数据需要读，就一直不断地把这个事件传递给用户

ET: `epoll` 的高效工作模式。`epoll_wait` 每个事件只会通知一次，应用程序必须要立即处理，只有第一次满足条件时才触发，之后就不会再传递同样的事件了。

LT 模式是默认的工作模式，这种模式下 `epoll` 相当于一个效率较高的 `poll`。ET 模式在很大程度上降低了同一个 `epoll` 事件被重复触发的次数，因此效率要比 LT 模式高。`select/poll` 只有水平触发模式，`epoll` 默认的触发模式是水平触发，但是可以根据应用场景设置为边缘触发模式。

总结：`epoll` 是几乎是大规模并行网络程序设计的代名词，一个线程里可以处理大量的 `tcp` 连接，`cpu` 消耗也比较低。很多框架模型，`nginx`, `node.js` 底层均使用 `epoll` 实现。

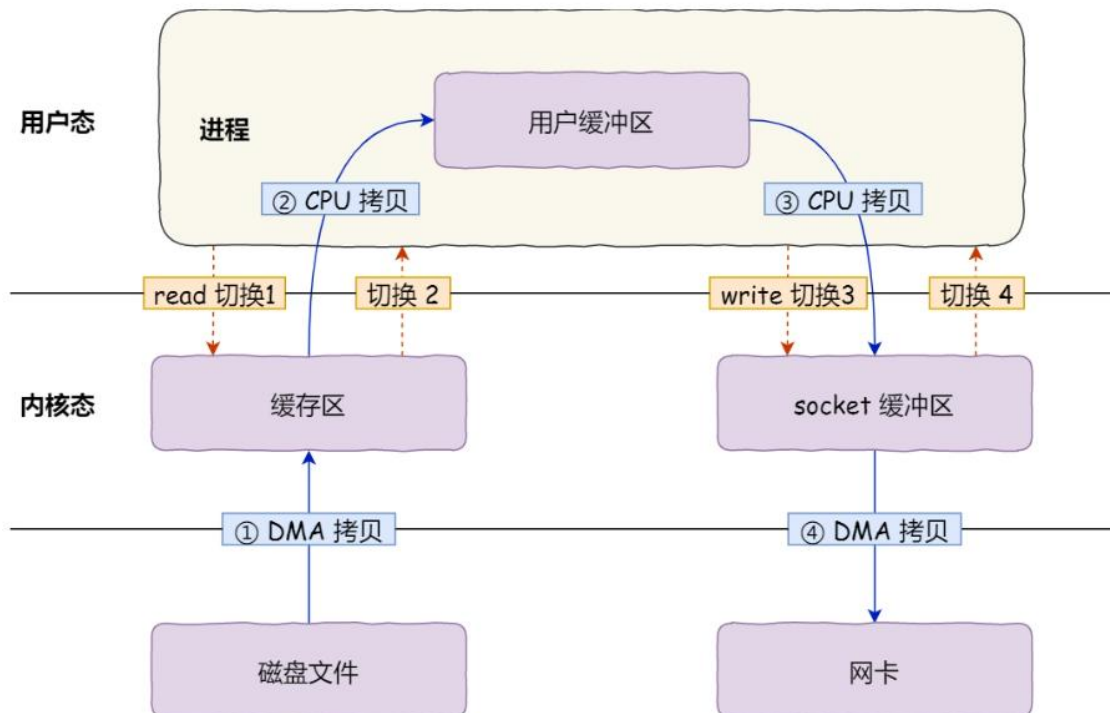
三、文件传输中的零拷贝

在 I/O 设备控制方式中，我们简单介绍了 DMA 技术，它用于磁盘等块设备，自动将数据从磁盘中拷贝到内存的缓存区，或者将数据从内存拷贝到磁盘中，我们暂且将这种拷贝称为 DMA 拷贝。

假设现在在服务端要提供文件传输功能：读取磁盘上的文件，然后通过网络协议发送给客户端。两个文件 I/O 的系统调用如下：

```
read(file, tmp_buf, len);
write(socket, tmp_buf, len);
```

代码很简单，但是背后发生了很多事情：

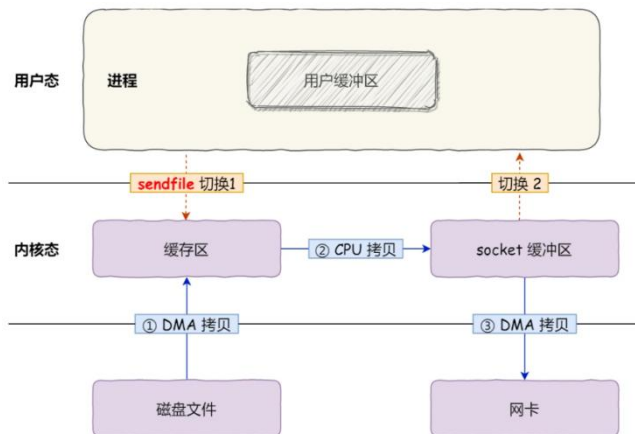


可以看到，两次系统调用共发生了 4 次「**用户态和内核态的上下文切换**」和 4 次「**数据拷贝**」。可以发现，我们发送一份数据，内部却拷贝了 4 次，多余的数据拷贝的时间消耗，在高并发场景下容易被放大和累计，影响性能。

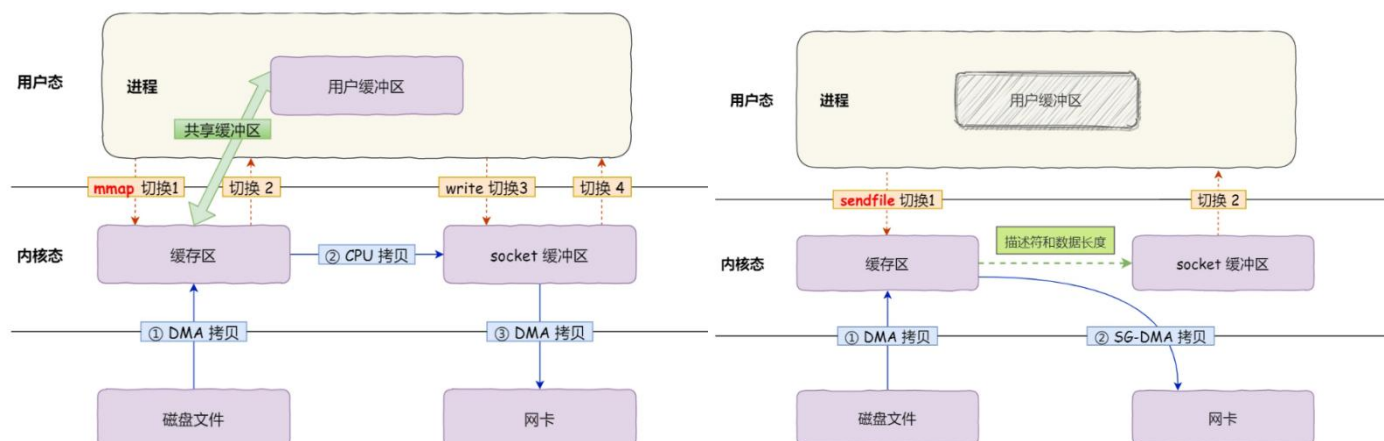
首先在文件传输情景下，用户缓冲区是没必要存在的，因为用户程序并不会修改文件；其次如果可以减少内存中的缓存区，则也会减少「**数据拷贝**」的次数

这种优化思路的实现就是零拷贝技术，零拷贝技术的实现通常存在如下 2 种方式：

(1) 使用系统调用函数 `mmap+write`。用 `mmap()` 替换 `read()` 系统调用函数，`mmap()` 系统调用函数会直接把内核缓冲区里的数据「映射」到用户空间，这样，就减少了用户缓存区，减少了一次拷贝。



(2) 使用系统调用函数 `sendfile`。减少一次系统调用，其次如果网卡支持 SG-DMA 技术 (The Scatter-Gather Direct Memory Access)，可以进一步减少一个缓冲区。



这就是所谓的零拷贝（Zero-copy）技术，因为我们没有在内存层面去拷贝数据，也就是说全程没有通过 CPU 来搬运数据，所有的数据都是通过 DMA 来进行传输的。

总结下，零拷贝技术用在文件传输场景下，可以全程不需要 CPU 参与，只通过 DMA 控制，传统文件传输方式，需要 4 次「**用户态和内核态的上下文切换**」和 4 次「**数据拷贝**」，零拷贝技术只需要 2 次「**用户态和内核态的上下文切换**」和 2 次「**数据拷贝**」，文件传输的性能提高至少一倍以上。很多著名的开源项目例如 Kafka 和 Nginx 都使用了零拷贝技术。

接下来介绍磁盘中的**页缓存/磁盘高速缓存**：

CPU 和磁盘之间的读写就是之前说的 CPU 和 I/O 设备的读写，其中间需要添加一个**缓冲区**，对于磁盘来讲，这个缓存区就是**页缓存**（PageCache），也就是**磁盘高速缓存**。请注意，**页缓存**是在逻辑上属于磁盘，在物理上属于内存。

页缓存用来缓存最近被访问的数据，当空间不足时淘汰最久未被访问的缓存。所以，读磁盘数据的时候，优先在**页缓存**找，如果数据存在则可以直接返回；如果没有，则从磁盘中读取，然后缓存 页缓存中。同时，对于机械磁盘来讲，**页缓存也会采取预读功能**，加入 I/O 请求读取 32KB 的数据，但是 CPU 会在页缓存中读取 64KB 的数据，将前 32KB 的数据返回。

当**传输大文件**（GB 级别的文件）的时候，**页缓存会不起作用**，这是因为每当用户访问大文件时，如果内核程序就把它们载入页缓存中，页缓存空间将会很快被这些大文件占满，针对大文件的传输，不应该使用 PageCache，也就是说不应该使用零拷贝技术，因为可能由于 PageCache 被大文件占据，而导致「热点」小文件无法利用到 PageCache，这样在高并发的环境下，会带来严重的性能问题。

所以，零拷贝技术使用了页缓存，是一种非直接 I/O，它适合传输热点小文件,不适合传输大文件。传输大文件不使用页缓存，而且采用异步模式，综上，高并发场景下的文件传输：

传输小文件：非直接 I/O+零拷贝技术

传输大文件：直接 I/O+异步 I/O

（3）反应堆模式和 Proactor

前面我们已经知道 epoll 是 Linux 大规模并行网络程序设计的必用的 I/O 复用技术，但是 epoll 模型只提供了几个系统调用函数，程序员直接使用基本也就是面向过程开发，为了提高开发效率，主流框架采用面向对象思想将 epoll 模型进一步封装，产生了 Reactor 模式（反应堆模式），也叫做 Dispatcher 模式（分配器模式）。

这里的反应指的是「对事件反应」，也就是来了一个事件，Reactor 就有相对应的反应/响应。

Reactor 模式主要由 Reactor 和处理资源池这两个核心部分组成，它俩负责的事情如下：

Reactor 负责监听和分发事件，事件类型包含连接事件、读写事件；

处理资源池负责处理事件，如 read -> 业务逻辑 -> send；

Reactor 模式是灵活多变的，可以应对不同的业务场景，灵活在于：

Reactor 的数量可以只有一个，也可以有多个；

处理资源池可以是单个进程 / 线程，也可以是多个进程 / 线程；

剩下的 3 个方案都是比较经典的，且都有应用在实际的项目中：

单 Reactor 单进程 / 线程；

单 **Reactor** 多线程 / 进程；

多 **Reactor** 多进程 / 线程；

方案具体使用进程还是线程，要看使用的编程语言以及平台有关：

Java 语言一般使用线程，比如 **Netty**；

C 语言使用进程和线程都可以，例如 **Nginx** 使用的是进程，**Memcache** 使用的是线程。

接下来，分别介绍这三个经典的 **Reactor** 方案。

迭代日志表

迭代版本	迭代工作	迭代日期
V1.0	建立初稿文档，完成第 1 章和第 2 章的整理。	2022-01.22-01.23
V1.1	添加 Linux 操作系统的基础知识、内存管理、进程线程、文件系统的初步笔记。	2022-01.24-01.28
V1.2	结合面试题，添加 Linux 操作系统的硬盘、死锁、信号、的笔记。	2022-01.29-01.31