版本号：v1.2 阶段：初稿

# 操作系统面试宝典

MRL Liu

（未经许可，不得传播）

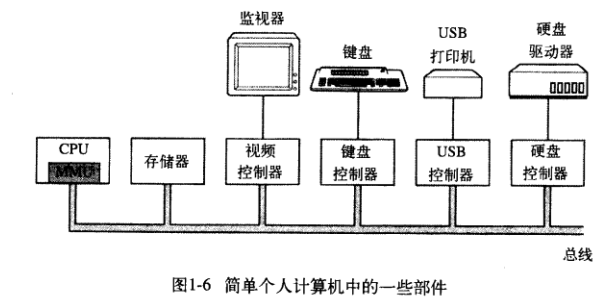
2022年01月22日

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

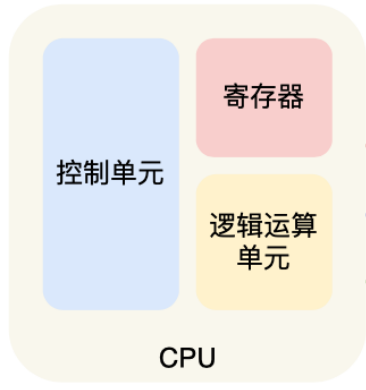
操作系统的设计与运行和硬件资源密切相关，为此本小节简单梳理下计算机中的常见硬件设备。



### 1、中央处理器

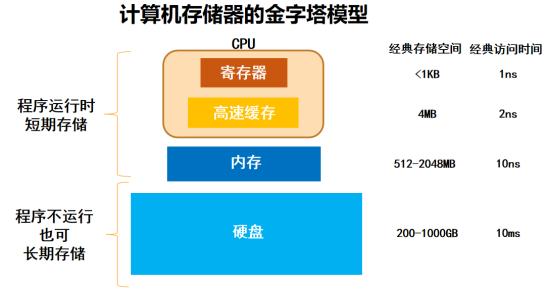
中央处理器（CPU）是计算机的大脑，负责从内存中取出指令并执行。不同产商的CPU对应不同的可执行的指令集。CPU中有一些寄存器用来保存关键变量和临时数据，如程序计数器、堆栈指针、程序状态字（PSW）、存储器管理单元（MMU）。目前的CPU正在具有更多的晶体管和缓存，以及多个CPU处理芯片，即多核CPU，使CPU从物理上支持操作系统的多进程和多线程。

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

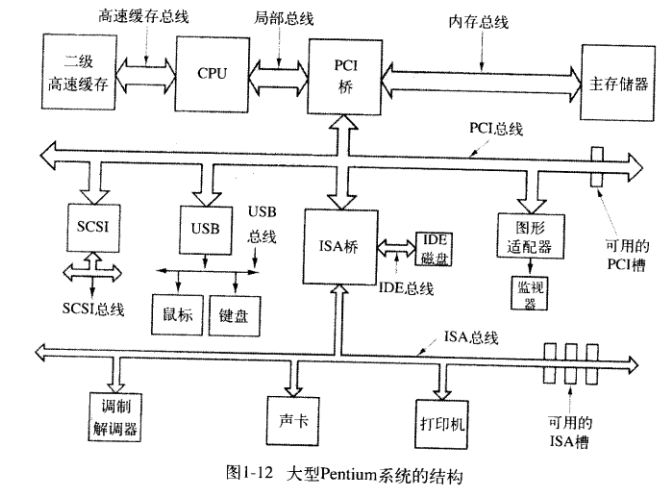
操作系统必须了解计算机内部的所有总线的配置和管理（本小节不要求读者记忆，仅理解即可）。总体来讲，计算机内部的总线可以分为以下3种：

（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：



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

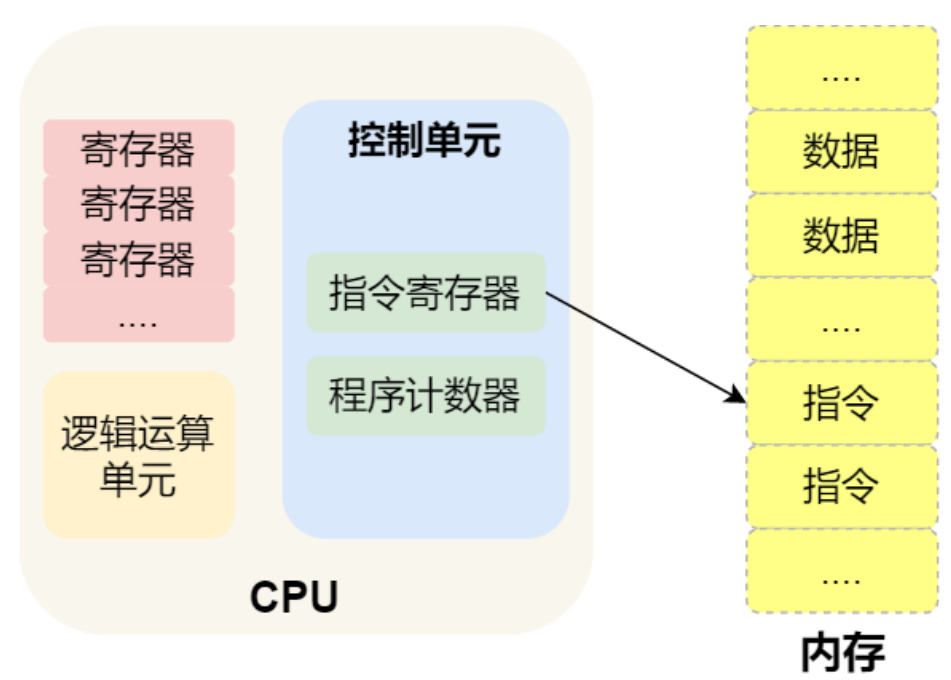
当计算机接通电源后，CPU启动，其内部的PC寄存器的值被强制初始化为BIOS程序的入口地址，然后CPU将一个被提前固化在主板上的BIOS芯片上的BIOS程序加载到内存中，这就是所有计算机运行的第一个程序BIOS（Basic Input Output System，基本输入输出系统），

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。

### 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。 |

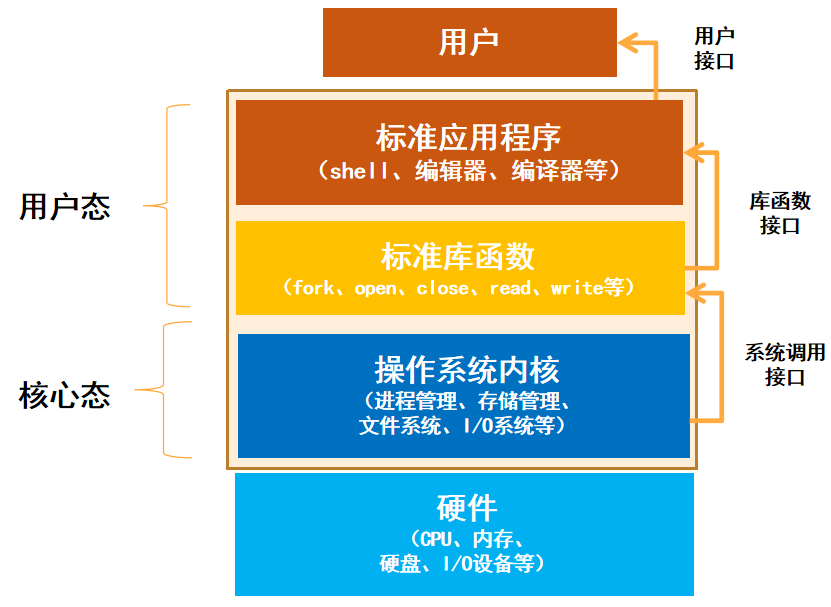
所以总结出操作系统的主要功能有两点：

|  |
| --- |
| （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。

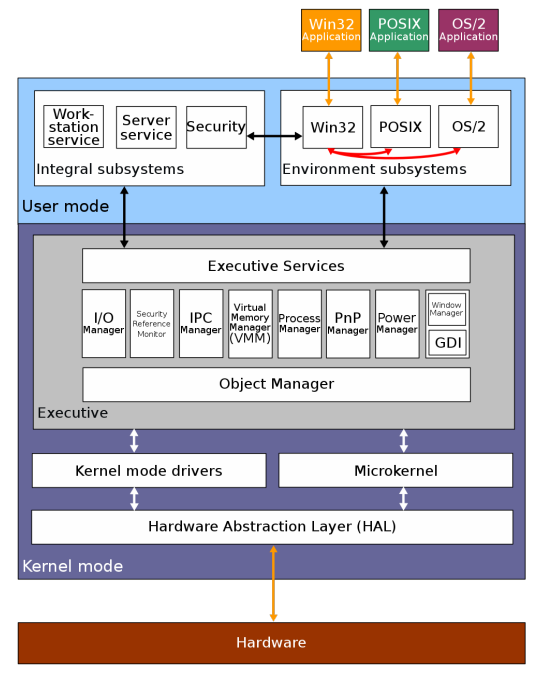
在用户态下，CPU只运行执行整个指令集中的一个子集，使用硬件的部分功能，此时的操作系统便在用户态下运行，只能访问部分CPU等资源，不能访问I/O设备。

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。



## 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**：

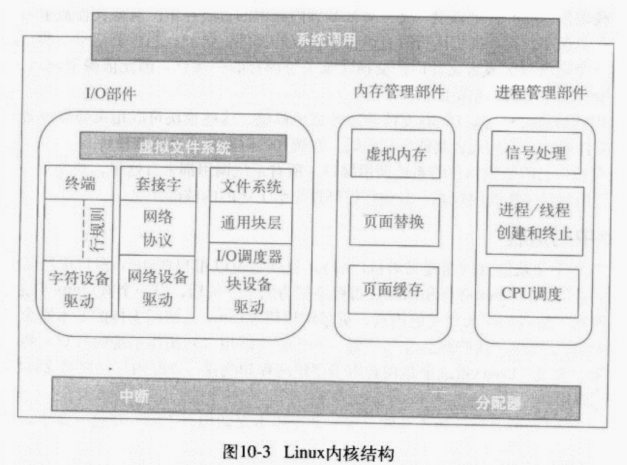


Linux操作系统的内核的设计理念是多任务（MutiTask）、对称多处理（SMP）、可执行文件链接格式（ELF）和宏内核（Monolithic Kernel）。其中SMP是指每个CPU核心的地位相等，不会有某个CPU单独服务某类程序；ELF可以理解为程序编译汇编生成的目标代码需要通过链接器形成可执行文件。

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。

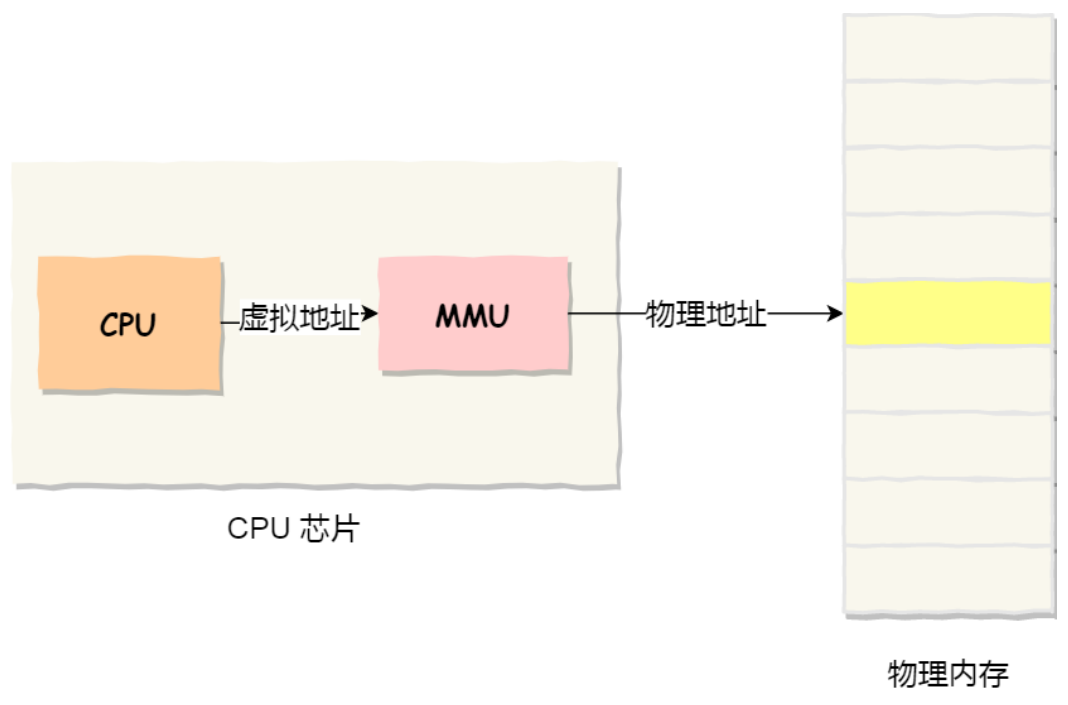
操作系统的程序**要尽可能避免访问绝对的**「物理地址」，所以操作系统为创建的每个进程分配一套独立的「虚拟地址」。基于操作系统的所有进程中的程序可以访问的都是「虚拟地址」，「虚拟地址」在CPU执行指令时映射到「物理地址」。（请注意，这里讲的物理地址和虚拟地址都是内存管理的概念，是内存（主存储器，RAM），硬盘上一般不认为有虚拟地址）



**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，所以就需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。

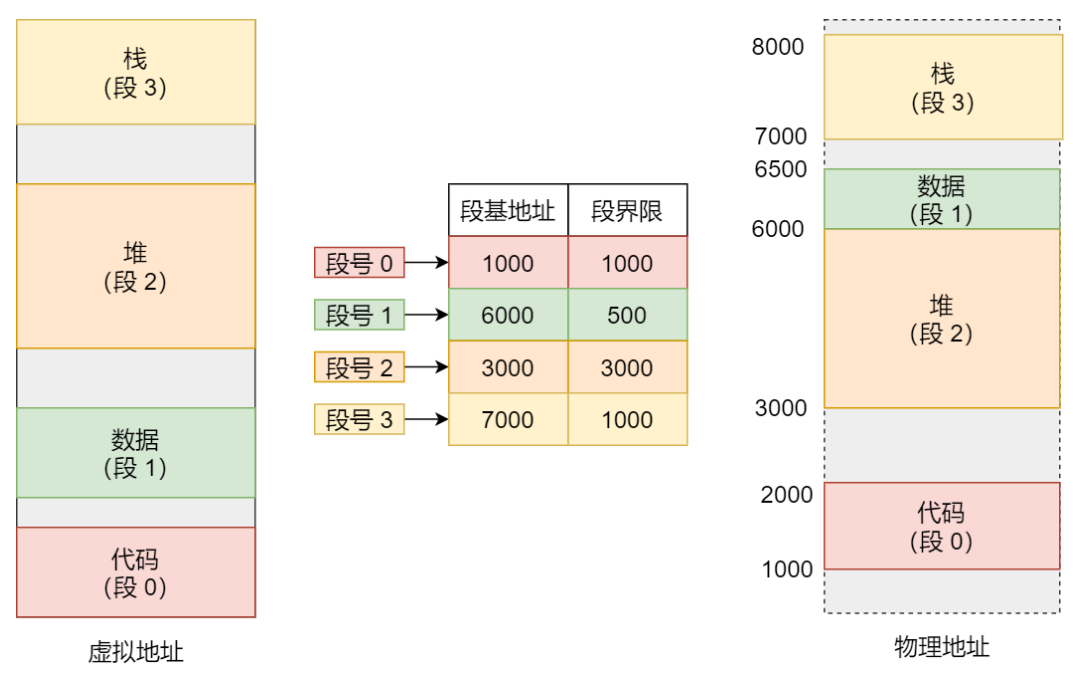
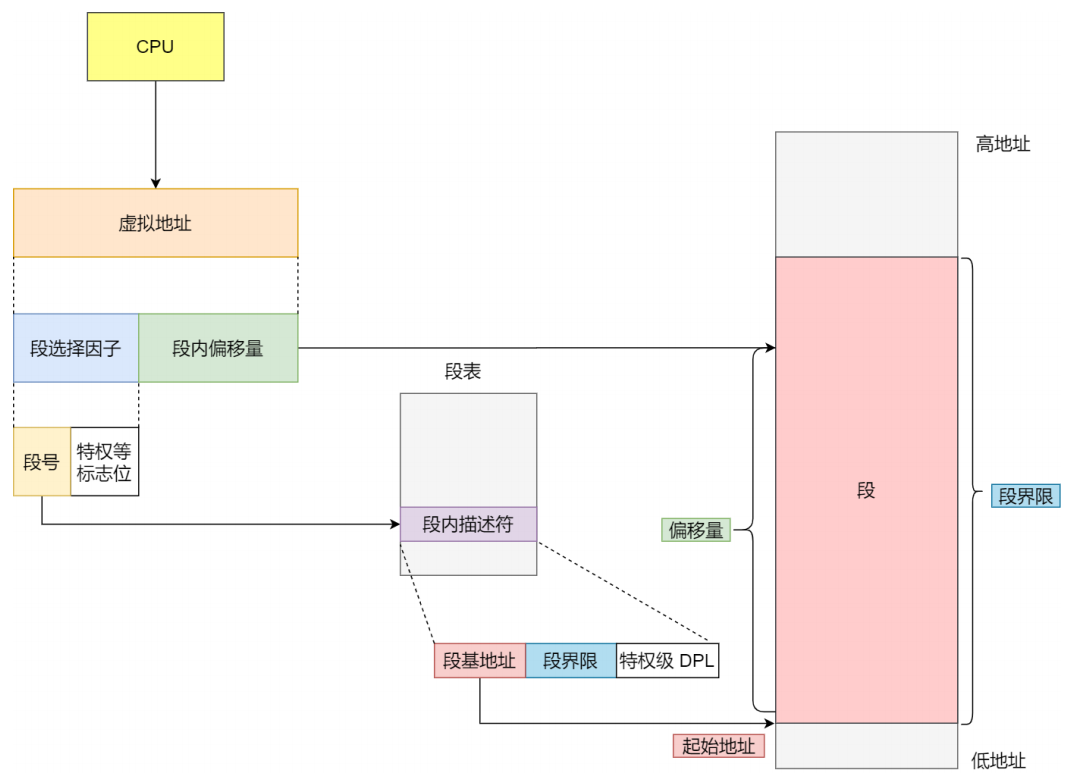


## 二、内存的分段分页

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。

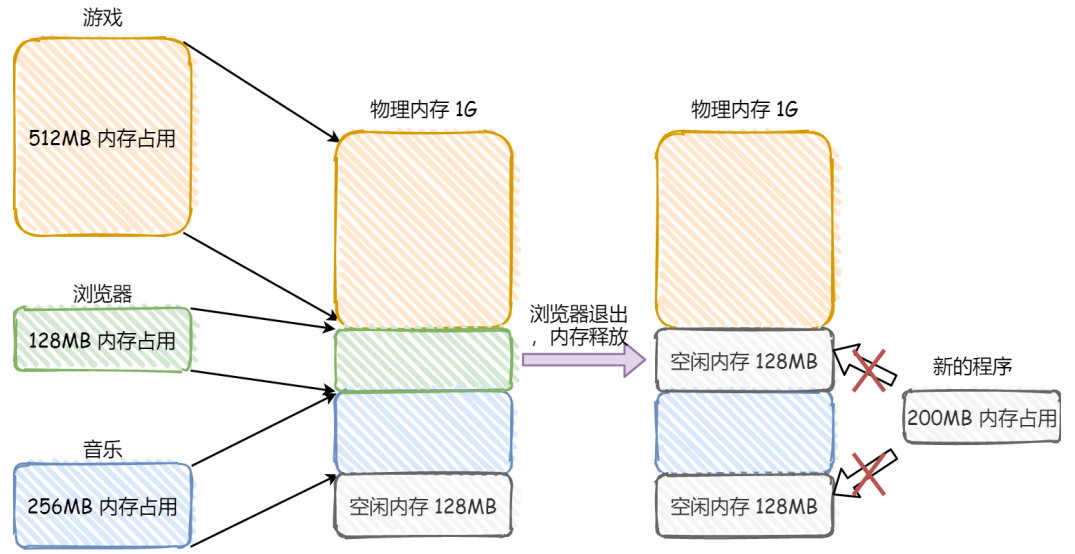
### 1、内存分段

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。



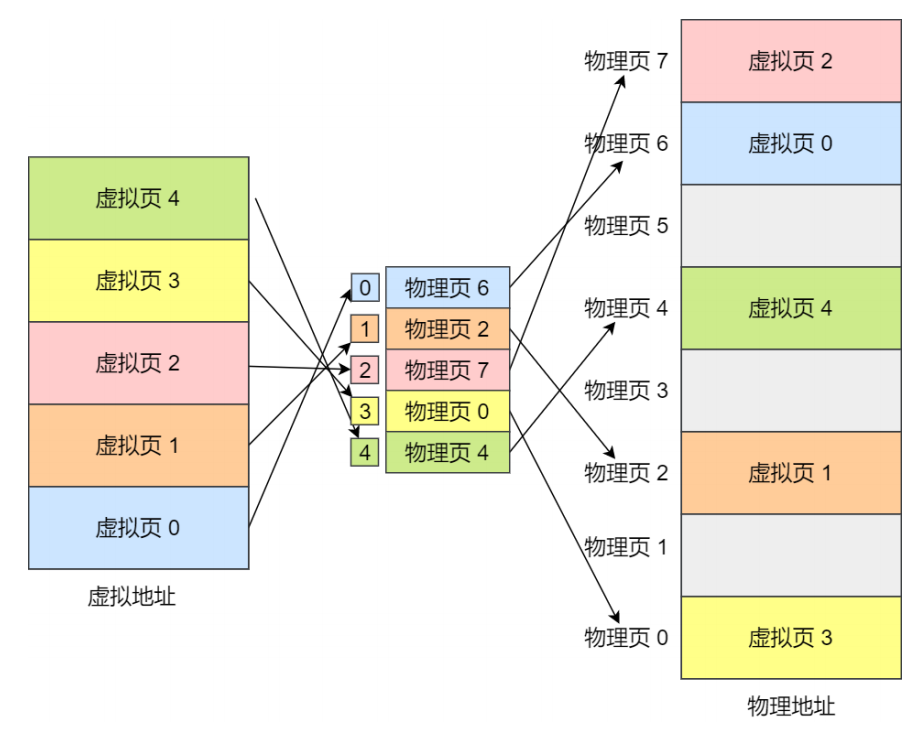
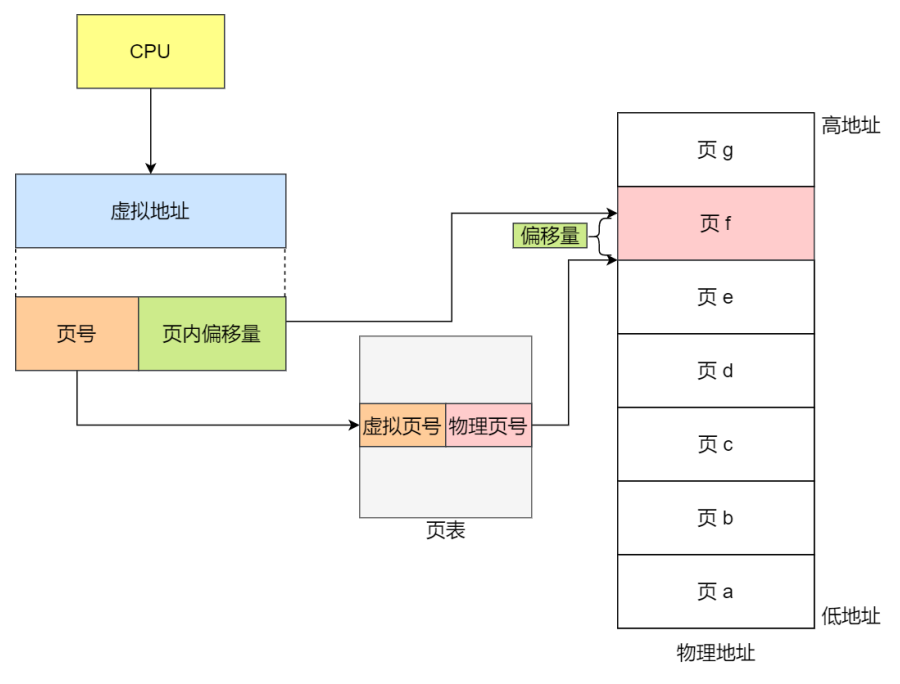
这种分段机制存在的显著问题就是存在内存碎片和内存交换效率低的问题。

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。



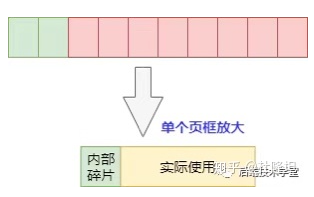
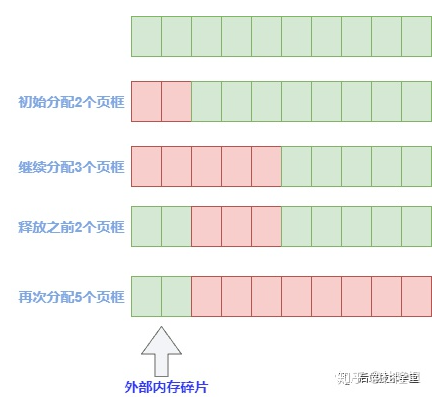
### 2、内存分页

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。

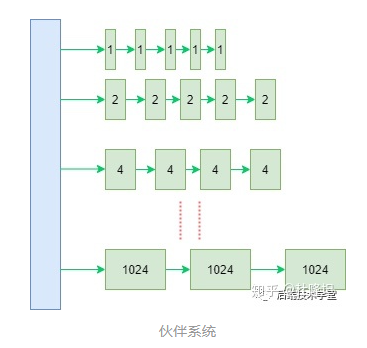


## 二、页面管理算法

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？

|  |
| --- |
| 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。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。Linux系统使用的是类似于以下时钟置换算法。

### 最佳页面置换算法（OPT）

置换在「未来」最⻓时间不访问的⻚⾯,但是实际系统中⽆法实现，因为程序访问⻚⾯时是动态的。我们是⽆法预知每个⻚⾯在「下⼀次」访问前的等待时间，因此作为实际算法效率衡ᰁ标准。

### 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。

### 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。

### 时钟置换算法

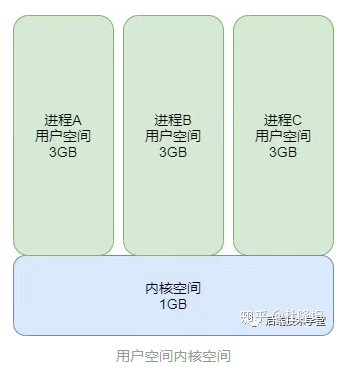
把所有的⻚⾯都保存在⼀个类似钟⾯的「环形链表」中，⻚⾯包含⼀个访问位。 当发⽣缺⻚中断时，顺时针遍历⻚⾯，如果访问位为1，将其改为0，继续遍历，直到访问到访问位为0⻚⾯，进⾏置换。

### （e）最不常⽤算法

记录每个⻚⾯访问次数，当发⽣缺⻚中断时候，将访问次数最少的⻚⾯置换出去，此⽅法需要对每个⻚⾯访问次数 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。

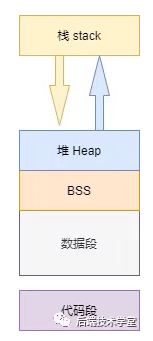
数据段：数据段用来存放可执行文件中已初始化的全局变量，换句话说就是存放程序静态分配的变量和全局变量。

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。



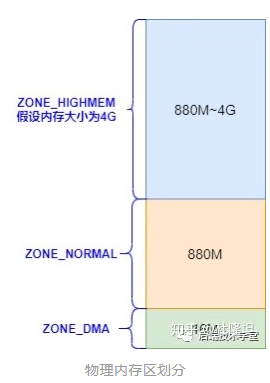
### 2、内核空间

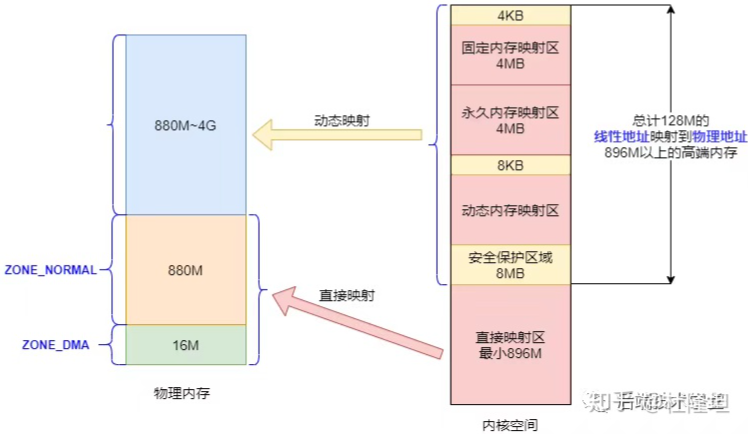
Linux的内核空间是指高端内存地址空间，总计1G的容量，包括了内核镜像、物理页面表、驱动程序等运行在内核空间。Linux内核会将物理内存分为3个管理区，分别是：

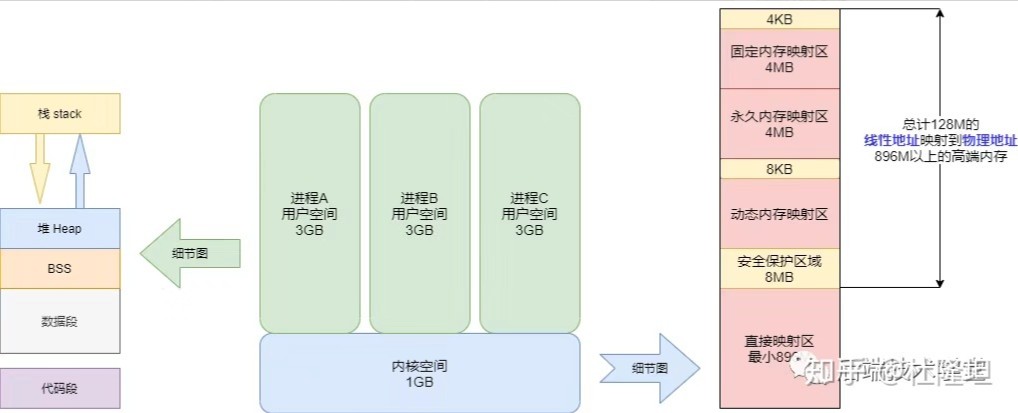
ZONE\_DMA。DMA内存区域。包含0MB~16MB之间的内存页框，可以由老式基于ISA的设备通过DMA使用，直接映射到内核的地址空间。

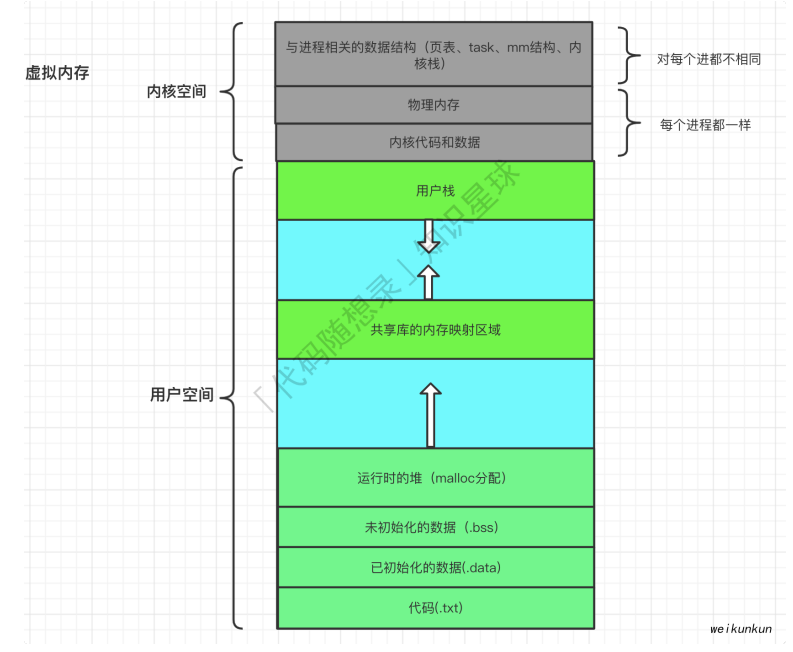
ZONE\_NORMAL。普通内存区域。包含16MB~896MB之间的内存页框，常规页框，直接映射到内核的地址空间。

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

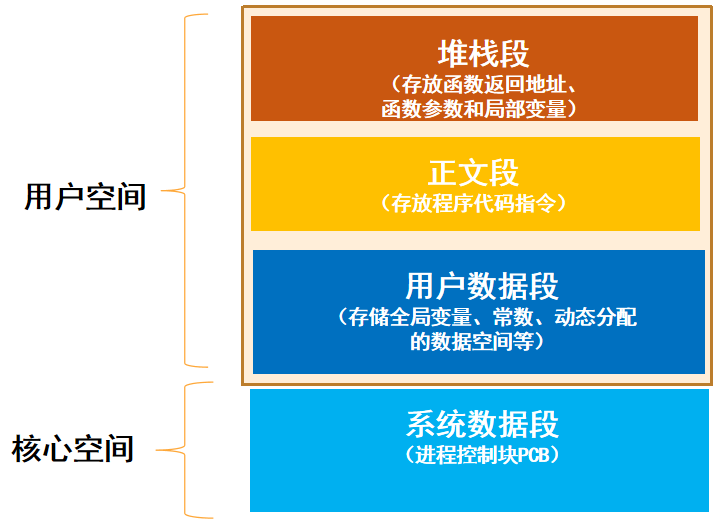
**程序**在运行前是存储在硬盘里的静态文件，运行时才被**CPU**加载到内存。**进程本质上就是一个正在运行的程序**，是对运行时程序的封装，其包含了该程序运行的所有资源和操作。

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在对应的厨房按照对应的食谱进行。 |

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），也是操作系统描述进程的单位。

Linux中的PCB就是task\_struct结构体，task\_struct也被称为Linux的进程描述符。task\_struct中有许多变量，涉及进程管理、存储管理、文件系统等，这里先介绍如下几个重要变量：

（1）**进程标识符**（PID）。Linux内核通过唯一的进程标识符来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）。

（2）**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。

（3）**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。

（4）**进程状态**（state）。进程当前的状态，取值为TASK\_RUNNING、TASK\_RUNNING、TASK\_INTERRUBTIBLE、TASK\_UNINTERRUPTIBLE、TASK\_KILLABLE等。

## 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，时间片时结束就切换到另一个程序。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**。

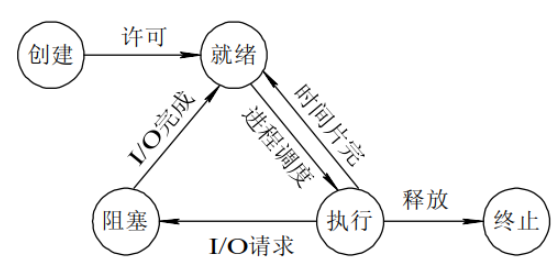
注意以下CPU的并发和并行概念的区别：

|  |
| --- |
| 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）。

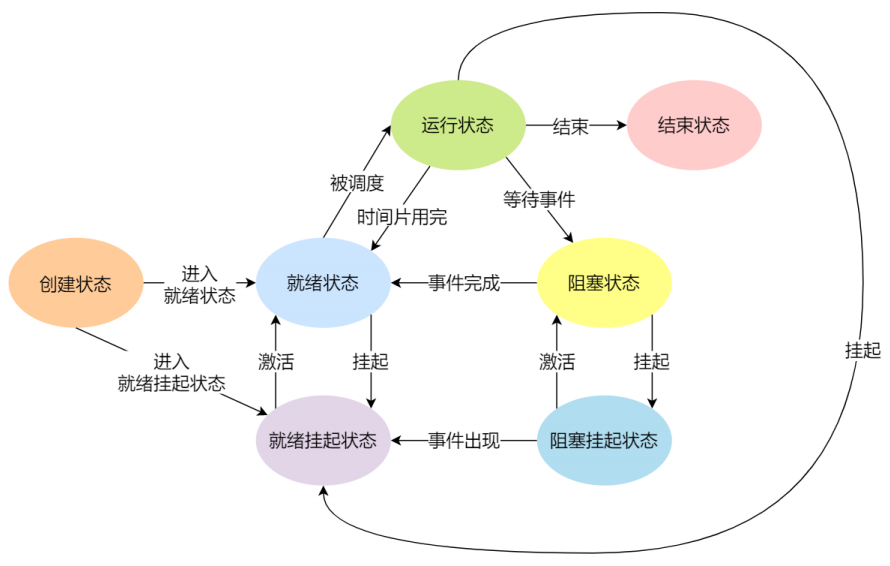
### 2、一般进程的状态切换

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 |

为了满足系统和用户观察、分析和及时修改进程的需要，还引入了一个对进程的重要操作——挂起，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**：



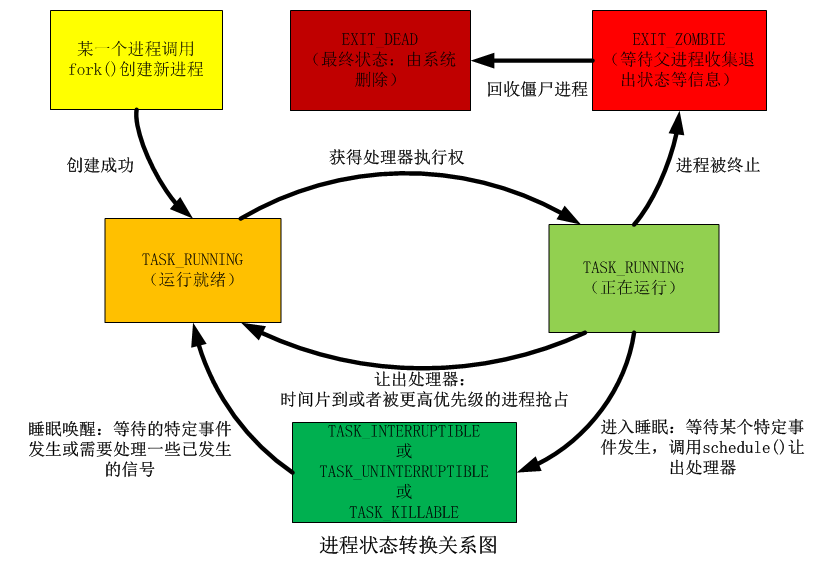
注意，在所有状态的切换中，**只有就绪态和运行态是双向转换的**，其他都是单向转换。

### 3、Linux进程的状态切换

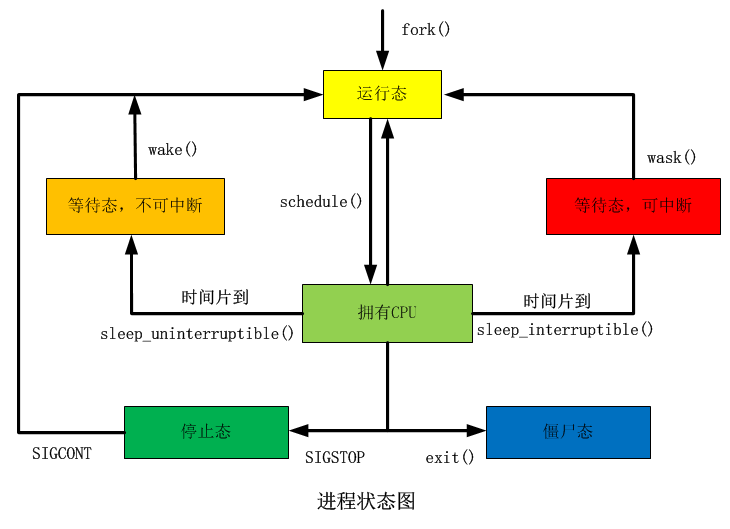
Linux中的进程运行状态有以下几种主要状态：

|  |
| --- |
| 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、SIGTTIN、SIGTTOU等信号，就会进入暂停状态。  E、**僵尸状态(TASK\_ZOMBLE)**：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如下图所示：



调用各个函数的状态如下：



## Linux进程的创建、执行和终止

### 1、Linux进程的创建

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。 |

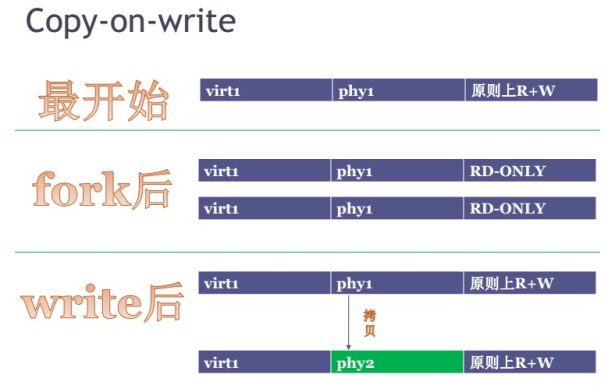
Linux系统中通过调用库函数接口**fork()**来复制当前进程创建一个新进程，调用fork函数的进程叫做父进程，新创建的进程叫做子进程。

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页的技术，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）。终止进程的过程如下：

|  |
| --- |
| 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

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）。

批处理系统中的调度（比如同时复制1000个文件）：

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。

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。

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。

交互式系统中的调度：

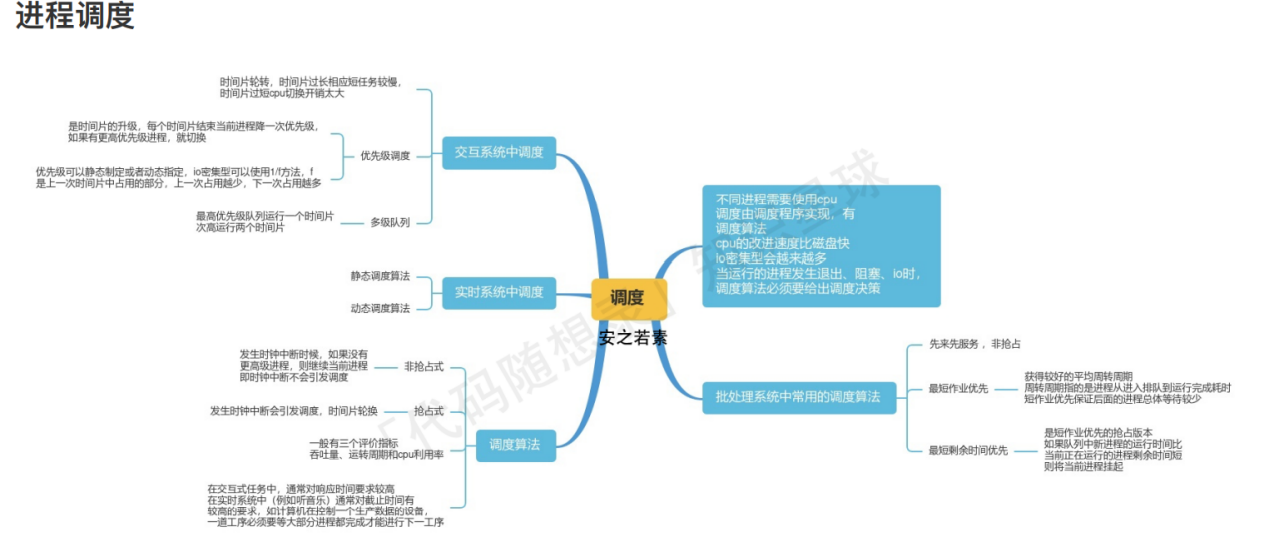
轮转调度（round robin）：每个进程被分配一个时间片，每个进程在自己的时间片内运行，时间片设置得过长可能引起短的交互请求响应时间过长，时间片设置得过短可能导致过多的进程切换操作，降低CPU效率，时间片一般为20ms-50ms。

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

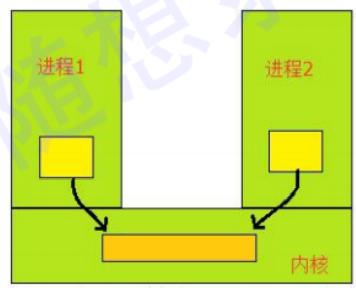


## 五、进程间通信

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。



### IPC的5种技术

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）**信号量（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，进程B必须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。

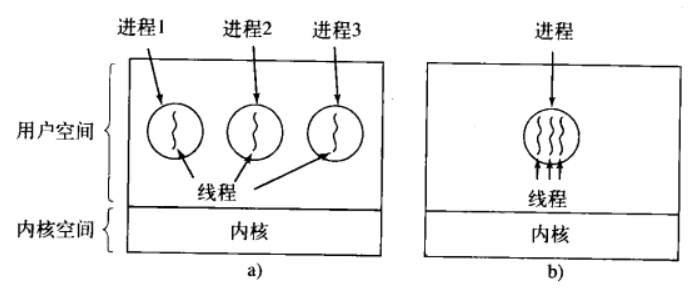
（4）**消息队列（MSG）**：消息队列是操作系统内核负责维护的一个链表结构，程序员可以创建一个消息队列并获得一个唯一的标识符key值，进程双方根据key值确定同一个消息队列进程通信，消息必须指定消息类型并封装在专用的msg结构中，通信双方可以根据不同的消息类型来获取不同的消息。

（5）**共享内存（SHM）**。是IPC中的最快的效率最高的方法之一。允许两个及以上的进程访问同一个进程，一个进程向共享内存写入数据，共享的进程立刻就会察觉，一般需要和信号量配合使用。

## 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。多线程可以大大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。 |

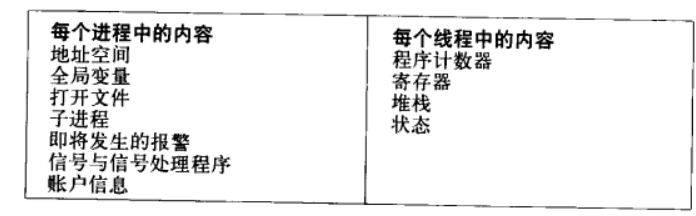
#### 进程和线程的比较

**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：**

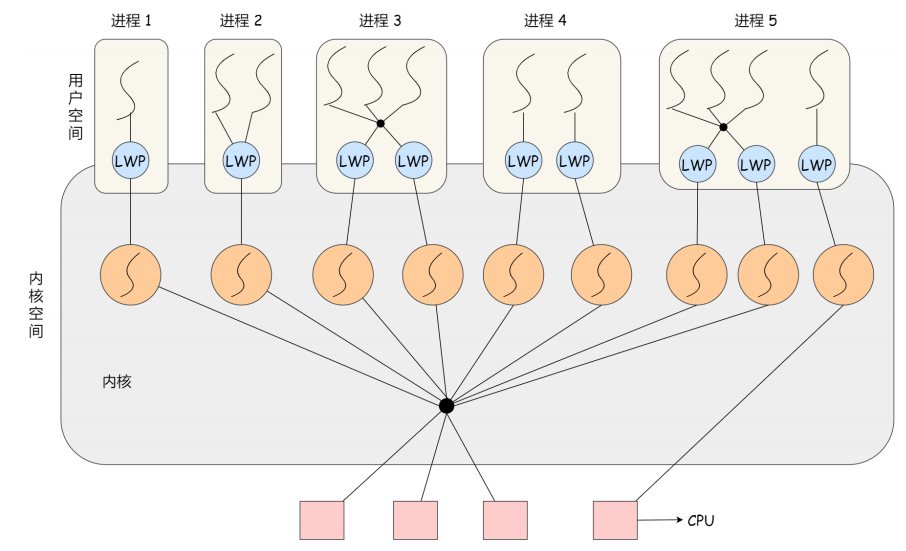
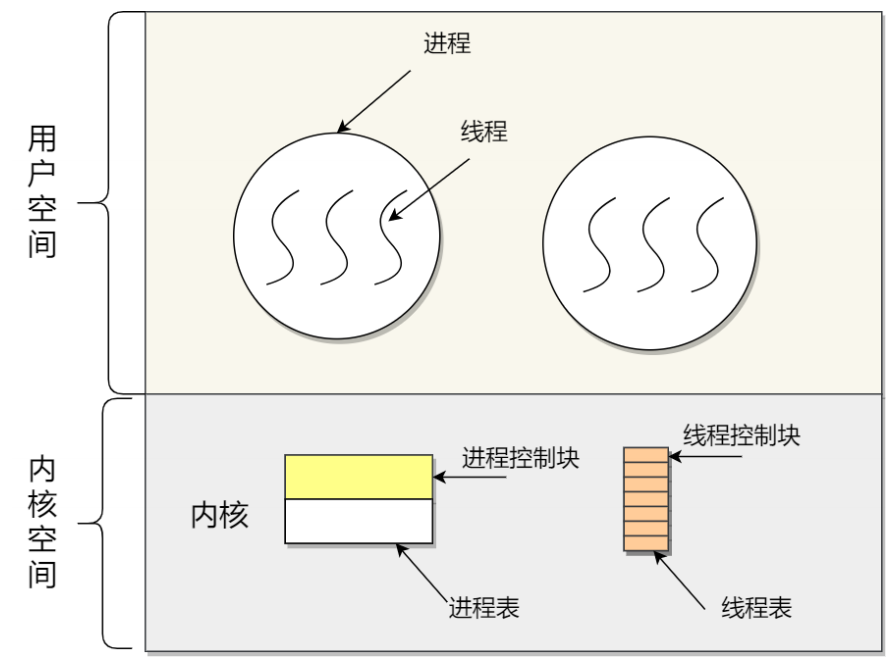
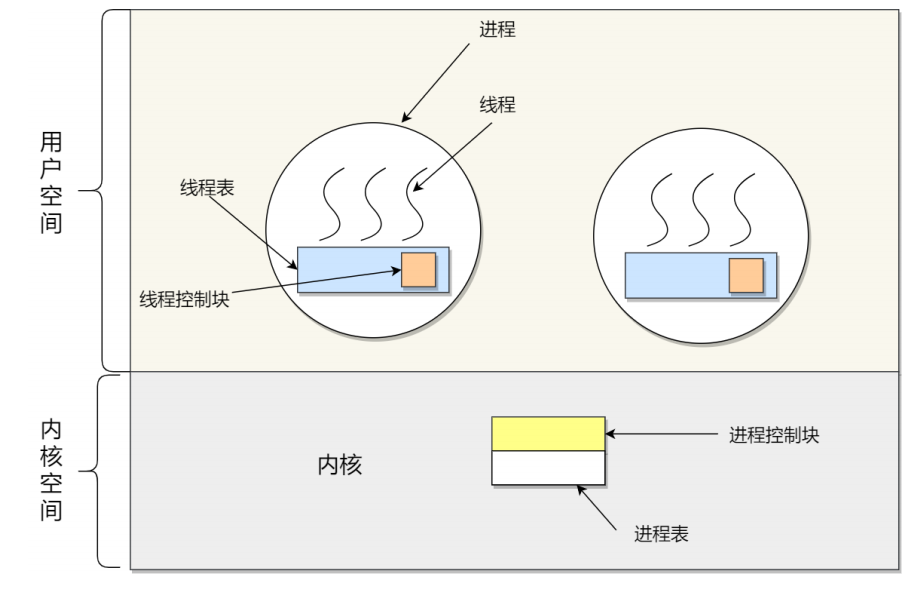
1. **功能不同。**进程是操作系统最小的资源分配的单位，线程是操作系统的最小的执行单位。
2. **资源不同。**操作系统只为每一个进程分配资源和回收资源；同一个进程中的所有线程共享该进程的资源，同时每个进程也独享一些资源：寄存器、堆栈、状态和程序计数器。如下：



（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

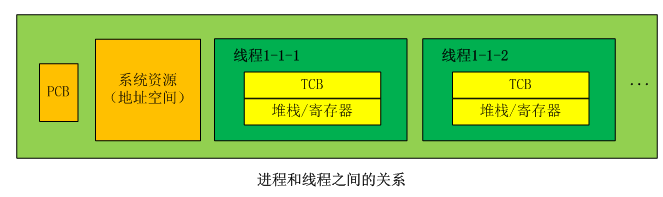
有两种主要的方法来实现多线程机制：**用户级线程**和**内核级线程**。前者在操作系统的**用户空间**实现线程包，后者在操作系统的**内核空间**实现线程包。在这两种方式上，Linux使用了轻量级进程（LightWeight Process），其在内核中支持用户线程。



|  |  |  |  |
| --- | --- | --- | --- |
| **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 | 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。 | 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 | 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** | 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。 |
| 轻量级线程（LWP） | 轻量级进程(LWP)是建立在内核之上并由内核支持的用户线程，每一个轻量级进程都与一个特定的内核线程关联。 | 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 |  |

### 2、Linux的线程

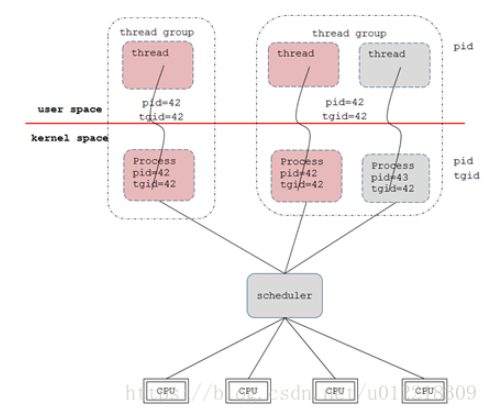
从用户空间看，Linux中用户空间的线程使用一套通用的第三方pthread线程库，因此具有很好的移植性。一个进程可以拥有多个线程，其中每个线程共享该进程所拥有的的资源。要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其它线程带来影响。由此可知，多线程中的同步是非常重要的问题。在多线程系统中，进程与线程的关系如下图所示：



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**。

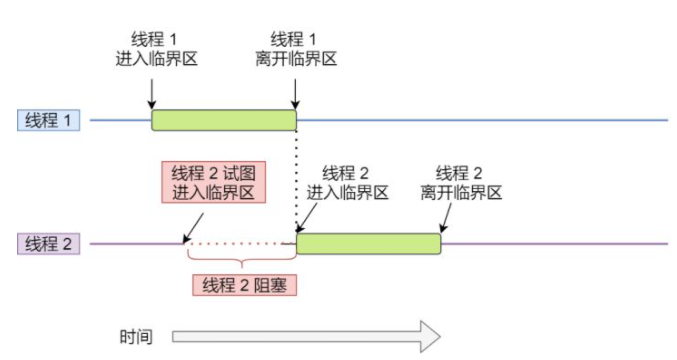


## 七、同步与互斥

**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

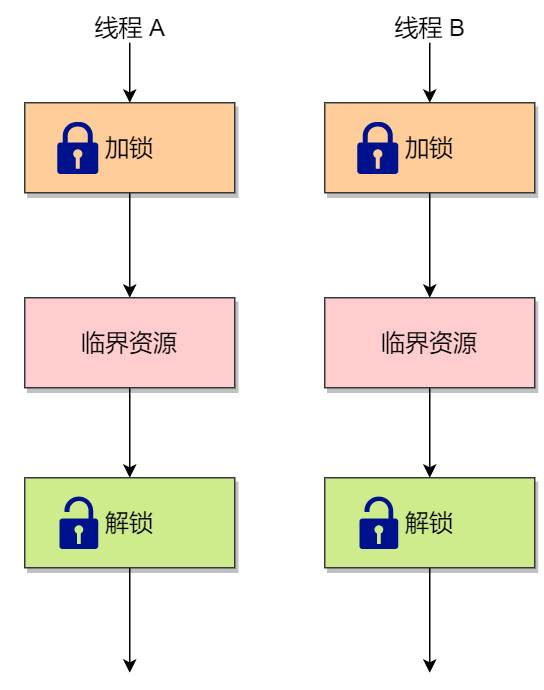
实现同步和互斥主要有如下两种方案：

#### 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：**任何想进入临界区的线程，必须先执行加锁操作。若加锁操作顺利通过，则线程可进入临界区；在完成对临界资源的访问后再执行解锁操作，以释放该临界资源。对互斥锁进⾏加锁后，任何其他试图再次对互斥锁加锁的线程将会被阻塞，直到锁被释放。

根据回斥锁的实现方式，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。



如果⼀个进程集合中的每⼀个进程都在等待只能由该进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）。

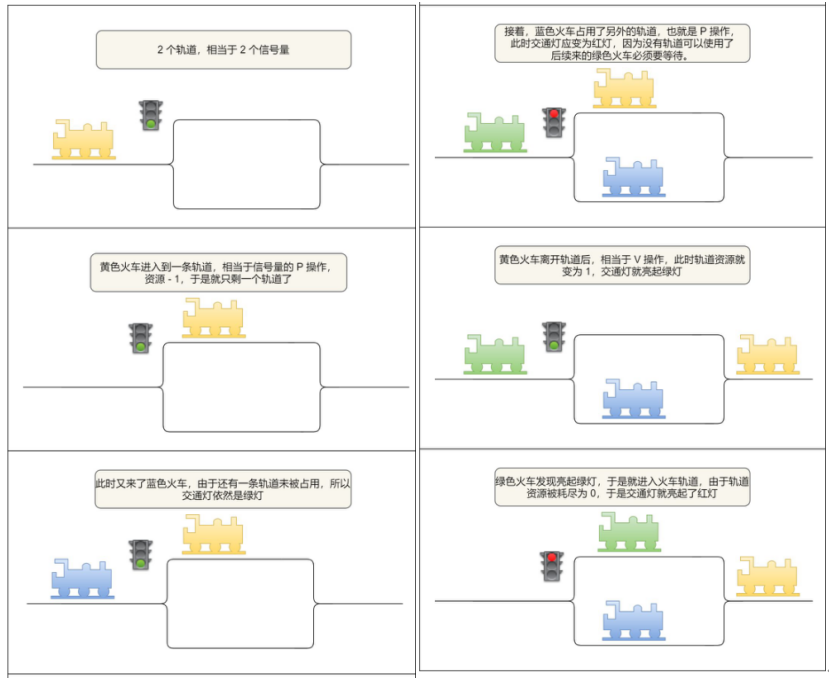
#### 信号量

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）。

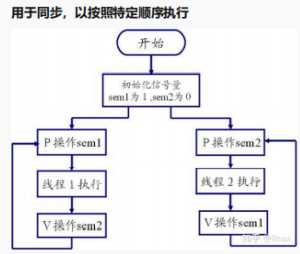
P操作：将sem减1，相减后，如果sem<0，则进程/线程进入阻塞等待，否则继续，表明P操作可能会阻塞；

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。



### 死锁和读写锁

如果⼀个进程集合中的每⼀个进程都在等待只能由该进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**。完全避免死锁是非常困难的。

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。Linux系统中的POSIX定义的读写锁的数据类型是：pthread\_rwlock\_t。

读写锁的特点：

1. 如果有其它线程读数据，则允许其它线程执⾏读操作，但不允许写操作
2. 如果有其它线程写数据，则其它线程都不允许读、写操作。

读写锁分为读锁和写锁，规则如下：

1. 如果某线程申请了读锁，其它线程可以再申请读锁，但不能申请写锁。
2. 如果某线程申请了写锁，其它线程不能申请读锁，也不能申请写锁。

### 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](https://www.zhihu.com/search?q=%E7%BC%93%E5%86%B2%E5%8C%BA&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2062905174}" \t "https://www.zhihu.com/question/_blank)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？



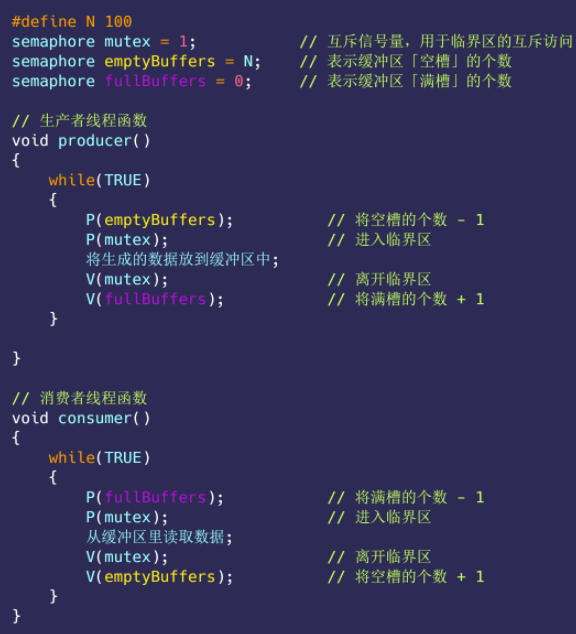
分析问题：任何时刻只能有一个线程操作缓冲区，说明操作缓冲区是临界代码，需要互斥；缓冲区空时，消费者必须等待生产者生成数据，缓冲区满时，生产者必须等待消费者取出数据，说明生产者和消费者需要同步。所以需要三个信号量：

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）；

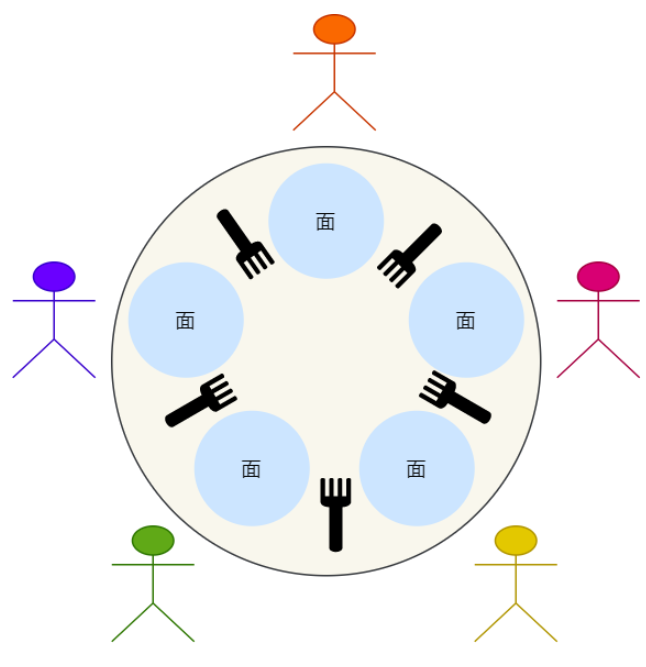
资源信号量emptyBuffers：用于生产者询问缓冲区是否有空位，有空位则生成数据，初始化值为n（缓冲区大小）；

实现代码：



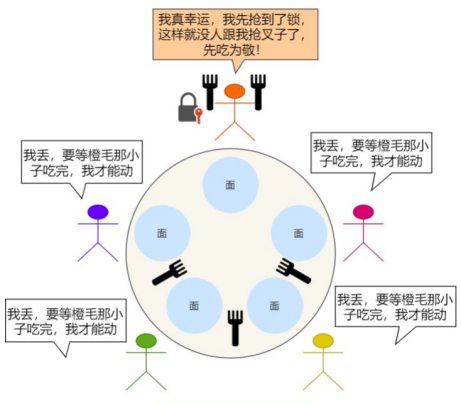
#### （2）同步问题：哲学家就餐问题

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？

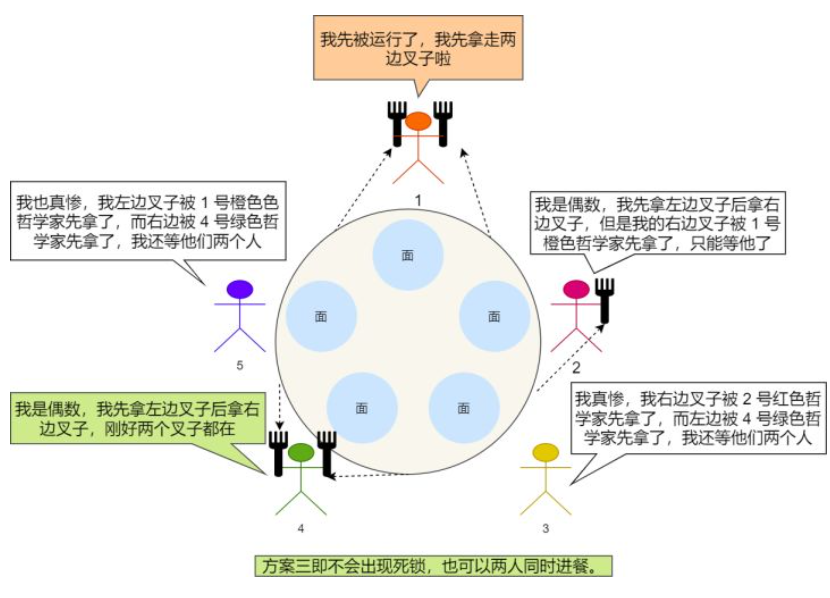


分析问题：哲学家们动作冲突的地方在吃饭时必须拿起左右2个叉子，将吃饭的动作作为临界区，吃饭前加锁，吃饭后解锁，就可以使用互斥信号量。

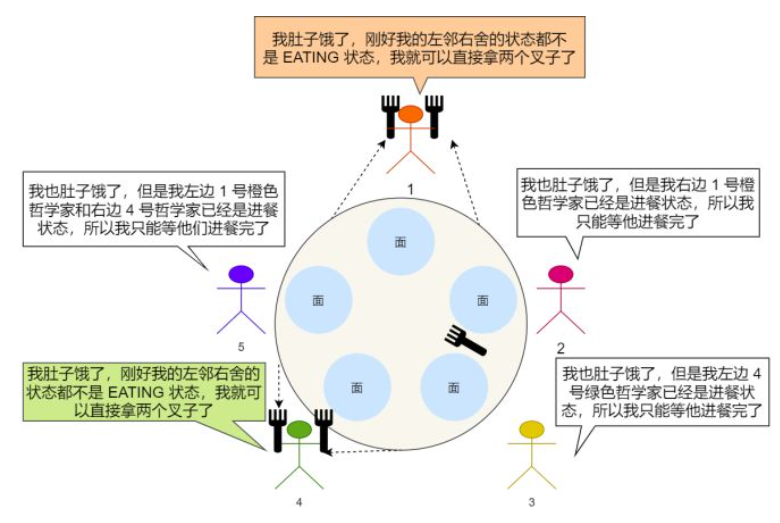
最简单的方案一：使用一个互斥信号量，哲学家吃饭时，其他人必须等待他解锁后才能吃饭。这样每次只能有一个哲学家吃饭。



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

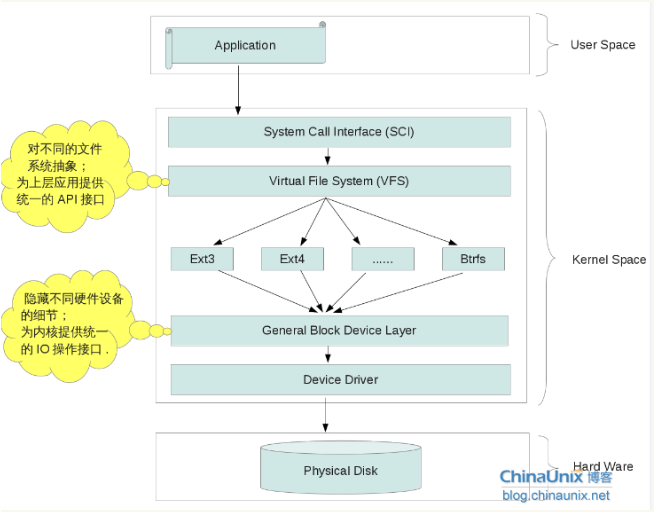
文件是操作系统中一个非常重要的概念，特别是Linux系统的文件系统，充分体现了“一切皆文件”的设计哲学。

内存中的数据只能在程序运行时存储，数据想要长期存储只能以文件的形式保存在磁盘等外部存储器中。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。

一个文件系统中的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



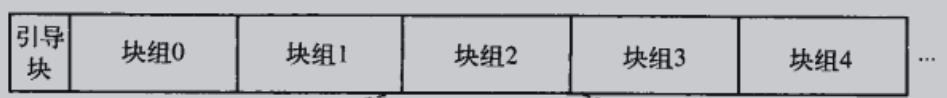
## 二、Linux的文件系统实现

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。

### 1、文件系统的四个要素

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。接下来分别介绍这四个概念。

磁盘在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），如下引导块存放引导计算机启动的启动项所在位置，块组0可能就是操作系统文件所在的地址区域（C盘）。



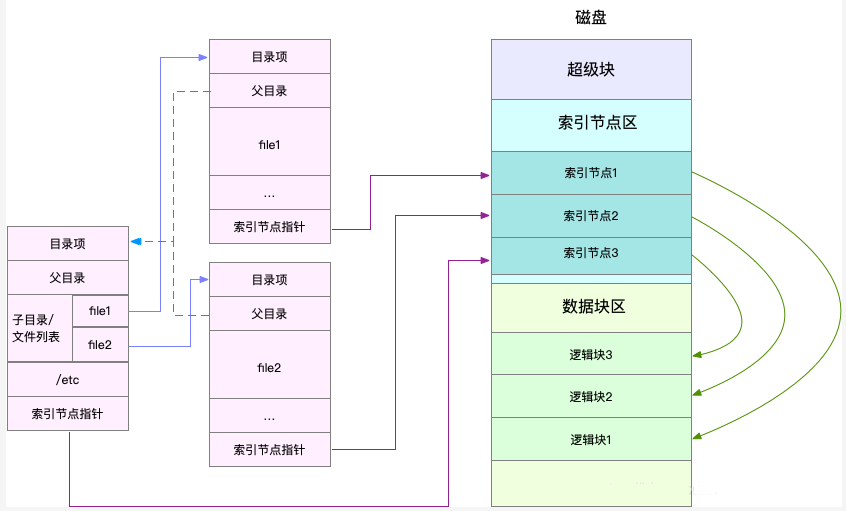
如果计算机添加了新的外部存储器，则会出现更多的块组，所以每个不同的块组都可能使用了不同类型的文件系统。

现在我们研究一个块组对应的文件系统的结构，通常来讲，一个文件系统的块组可以划分为三个部分：

（1）超级块（super block）：存储整个文件系统的信息，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。

（2）索引节点区：存储多个索引节点。索引节点（index node，简称inode）是一个数据结构块，描述了每个文件或目录对应的元数据（inode号，文件大小、访问权限等），每个文件或目录唯一对应一个索引节点，每个索引节点都被唯一编号，根目录的inode号是0，可以看做是文件/目录的唯一标识或指代。

（3）数据块区：存储每个文件的内容，与每个索引节点一一映射。



Linux系统在初始化一个文件系统后会得到该系统下各个文件的目录结构。在这个过程中，Linux首先从根目录（inode号为0）读取，系统程序首先在超级块中根据inode号索引到根目录的索引节点所在位置，找到该索引节点读取到内存中，然后为其创建一个目录项，然后根据该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。初始化过程中，所有目录项的数据会被保存到缓存中，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时非常快捷。

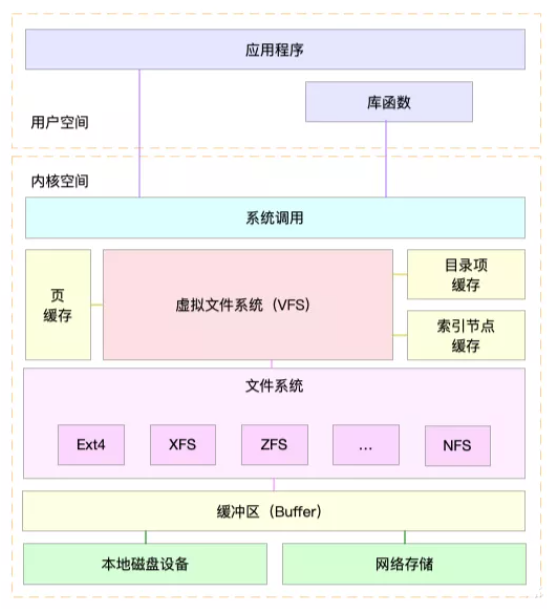
在这个过程中，我们看到了一个新的概念：目录项。

目录项（directory entry，简称为dentry），其本质也是一个描述文件或目录信息的数据结构，其与索引节点的区别是：

1. 目录项在系统程序读取索引节点后动态创建后存储到缓存中，其并非长期存储，而索引节点是保存在磁盘中长期存储。
2. 文件/目录与索引节点是一对一，与目录项是一对多，即一个文件/目录可能对应多个目录项但是只会对应一个索引节点。这是因为一个文件可以被链接到其他文件名，所以可能不止有一个文件名或路径名。

### 2、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。

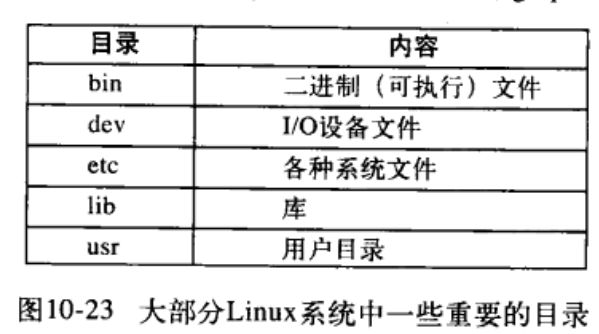
第一类是基于磁盘的文件系统，也就是把数据直接存储在计算机本地挂载的磁盘中。常见的 Ext4、XFS、OverlayFS等，都是这类文件系统。

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。

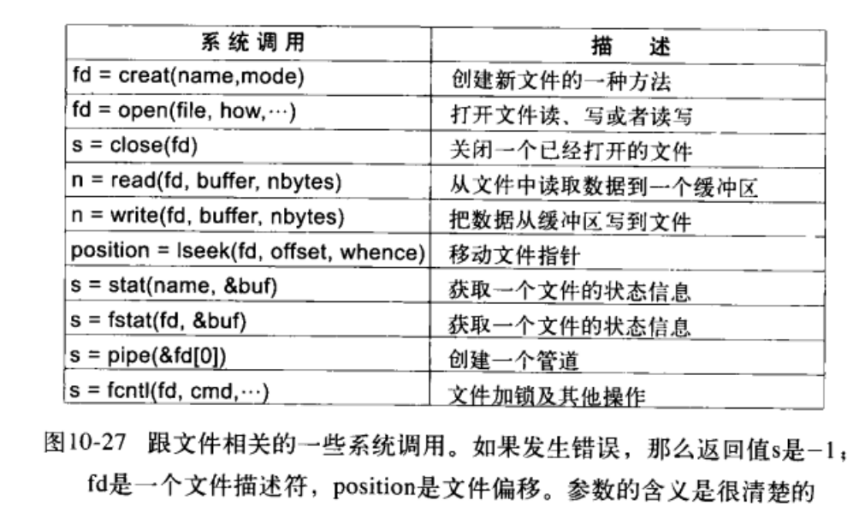
不同的文件系统想要在Linux系统上运行，要先挂载到VFS目录树中的某个子目录(称为挂载点)，然后才能访问其中的文件。拿第一类，也就是基于磁盘的文件系统为例，在安装系统时，要先挂载一个根目录(/)，在根目录下再把其他文件系统(比如其他的磁盘分区、/proc 文件系统、/sys文件系统、NFS等)挂载进来。

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。



### 3、Linux的文件系统调用

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的系统调用



与目录相关的系统调用：



## 三、I/O系统

操作系统的几种事件概念：，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

信号：是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：由外部事件引起，例如I/O设备完成某个操作时发送一个信号。中断由可屏蔽中断和不可屏蔽中断，前者CPU可以忽略，后者CPU不可忽略。

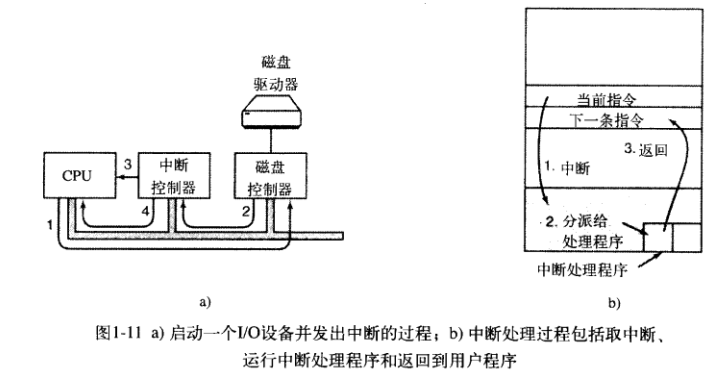
异常：CPU在执行程序指令时发现程序指令有错误而发出的信号

陷入（trap）：用户程序执行系统调用时必须让CPU切换到内核态的信号

用户程序发出一个系统调用，操作系统内核将其翻译为一个对应设备驱动程序的过程调用，设备驱动程序驱动I/O设备完成功能。I/O设备实现输入和输出的方式：

忙等待（busy waiting）。I/O设备驱动程序在一个循环中不断检查该设备是否完成操作，该过程中会占据CPU资源，CPU需要轮训。

中断。I/O设备驱动程序只有在该设备完成操作时发出一个中断，此刻设备驱动程序返回。



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，是指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正好相反，文件读写时，先要经过系统的页缓存，然后再由内核或额外的系统调用，真正写入磁盘。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。

不过要注意，直接I/O、非直接I/O，本质上还是和文件系统交互。如果是在数据库等场景中，还会看到，跳过文件系统读写磁盘的情况，也就是我们通常所说的裸I/O。

### 3、阻塞I/O和非阻塞I/O

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

所谓阻塞I/O，是指应用程序执行I/O操作后，如果没有获得响应，就会阻塞当前线程，自然就不能执行其他任务。

所谓非阻塞 I/O，是指应用程序执行I/O操作后，不会阻塞当前的线程，可以继续执行其他的任务，随后再通过轮询或者事件通知的形式，获取调用的结果。

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。

### 4、同步I/O和异步I/O

根据是否等待响应结果，可以把文件I/O分为同步和异步I/O，所谓同步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。所谓异步I/O，是指应用程序执行I/O操作后，不用等待完成和完成后的响应，而是继续执行就可以。等到这次I/O完成后，响应会用事件通知的方式，告诉应用程序。

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 SIGPOLL，来通知进程文件是否可读写。

# 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 |