操作系统

概述

> 部分内容引用自https://en.wikipedia.org/wiki/Operating_system

操作系统是一种管理硬件资源并为用户程序提供基础服务的系统软件。目前主流的操作系统包括Windows，Unix liked system（包括Linux based system，MacOS，BSD based system，Unix），Android，iOS。当然也有一些比较罕见的操作系统，比如GNU Hurd（一种微内核操作系统），Google新一代操作系统Fuchsia（目标是取代Android，基于Zircon内核）。

本文涉及的内容以X86和x86-64架构下的Linux Based System为基础。另外，由于操作系统和硬件密切相关，完全撇开硬件来谈操作系统是不现实的，因此文中也会在必要的地方提前介绍一些硬件和体系结构相关的内容。

从操作系统启动讲起

BIOS

BIOS全称Basic Input/Output System，它是系统通电之后运行的第一个程序，负责硬件检测、加载Boot Loader等工作，并且在操作系统启动后提供一些服务。

BIOS本质上是一段汇编编写的程序，这段程序被存储在主板上的ROM或者Flash中。在远古时代，BIOS被写入到ROM（Read Only Memory）中， 因此每次升级BIOS都必须将ROM从主板上移除并焊接上新的ROM；到了现代，BIOS被写入到Flash Memory上，可以反复读写——因此现代的主板可以由用户在终端进行更新，而不是返厂了。

通电后，CPU从`0x000fff0`开始（老）或利用特殊逻辑从母板ROM中（新）开始执行BIOS代码，然后开始P.O.S.T（Power On Self Test）过程，这个步骤称之为冷启动（热启动将会跳过自检）。自检结束后，BIOS将会扫描Option Rom，即额外的BIOS系统。一些比较复杂组件会有自己的BIOS来实现启动初始化和配置，比如Raid卡，显卡，高端SCSI控制器等。母板BIOS将会调用这些逻辑对特殊设备进行初始化。

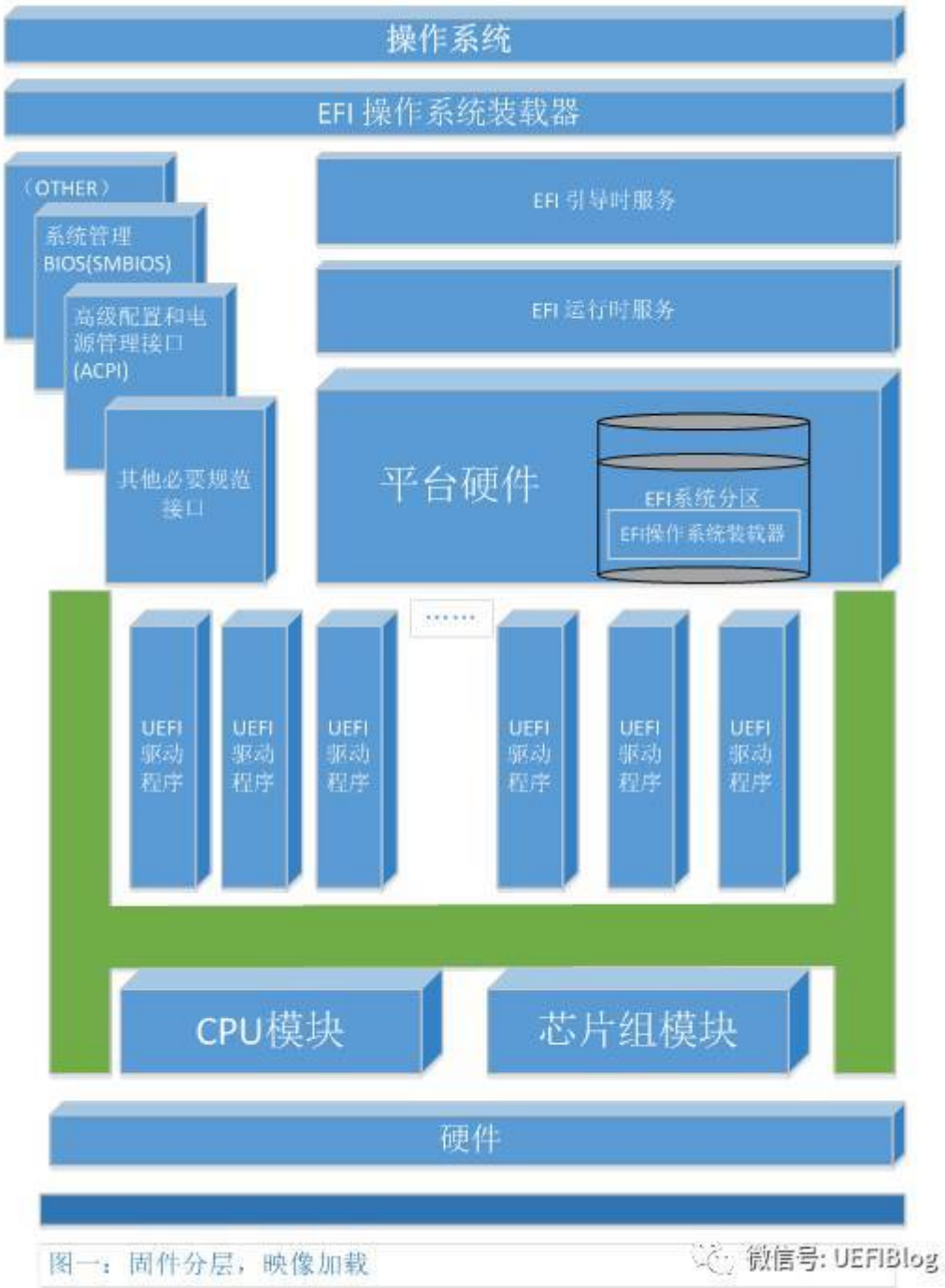
等待所有的自检和初始化完成后，BIOS将会使用`int 13h`开始启动逻辑（即19号中断处理程序）。该逻辑按顺序枚举所有的外部存储器（包括硬盘，DVD，USB存储器等），并从Boot Sector（存储器的第一个扇区）中检索分区信息。Boot Sector有不同的组织逻辑，最广为人知的是MBR（Master Boot Record）。Boot Sector标识了该硬盘是否是一个可启动的设备（Bootable Device）,并且含有Boot Loader的初始逻辑。BIOS将会加载Boot Sector中的Boot Loader至内存的`0x7c00`处，后续的文件系统识别，内核加载等工作将由Boot Loader完成。

至此，BIOS的使命已经基本完成了，它所使用的资源都将被释放——至少在现代PC中是这样的。在上古时代，当IBM第一次开发出BIOS时，它还负责了操作系统和应用程序与硬件沟通的桥梁，而传递信息的方式就是著名的BIOS中断。比如：`int 10h`调用显示相关功能，`int 13h`进行磁盘操作等。但是这种办法在面对层出不穷的新硬件的时候实在是过于低效，因此这种方式在现代已经被完全放弃了。也就是说，BIOS在今天只负责硬件初始化和最开始的启动流程，而具体的设备访问直接由操作系统驱动接管，不再经过BIOS。

在今天，BIOS由于自身设计缺陷和低效已经几乎被完全放弃。不过主板制造商们通常会在自己的主板中保留对BIOS启动的支持，因此在主板选项中，我们会同时看到Legacy Boot和UEFI Boot两种启动方式，而Legacy Boot就是指的BIOS启动流程。

UEFI

UEFI全称Unified Extensible Firmware Interface，是BIOS的继任者。为了简便，除非特殊说明，下文中所说的BIOS都是实现了UEFI接口的BIOS（或直接称UEFI）。比起传统BIOS主要为启动系统和一些简单的硬件沟通而设计，UEFI可以被描述为是介于操作系统和硬件之间的一层软件接口 。它的大概结构如下图（有些描述方式个人觉得不是很合适）：



可以说支持UEFI的BIOS几乎就是一个小型的操作系统——它有自己的驱动程序，有自己的编程接口定义，它甚至还有自己的shell（UEFI shell）。有了UEFI接口，BIOS可以在启动时就获得诸如访问网络、使用显卡、访问文件系统等高级的能力。比如，从网络下载系统镜像并启动成为可能（网吧的无盘系统已经使用了这种技术），操作系统内核加载前就可以显示酷炫的动态效果（还是网吧用的多）。

UEFI的启动流程和传统BIOS类似，上电以后进行开机自检，然后执行各个设备独立的BIOS代码，最后枚举Bootable Device。具体的细节暂且不表，我们来关注一下在确定Bootable Device后，UEFI如何加载Boot Loader。在传统BIOS启动流程中，传统BIOS使用MBR来判断该设备是否可启动以及定位执行Boot Loader。但是由于MBR本身的缺陷，使用MBR的硬盘最多只能寻址2TB的空间，这显然跟不上时代的步伐。因此现代的系统中，我们在对硬盘进行分

区时都推荐使用GPT分区而不是DOS分区（MBR分区表）。GPT分区表可以提供更大的寻址空间和更好的UEFI启动支持，并且提供了一小部分针对DOS分区的兼容能力。UEFI提供了对DOS分区方式启动的兼容能力，启动方式和之前提到的类似。对于使用GPT分区的存储设备，UEFI会去寻找一个EFI类型的分区，该分区使用FAT32文件系统，该分区中的目录严格按照UEFI规范中的要求，存储了不同的体系结构下的Boot Loader，该文件为EFI字节码，可被UEFI平台执行，类似于shell文件。比如x86-64的Linux的Boot Loader路径为`<EFI Partition>/efi/BOOT/BOOTX64.EFI`。如果当前存储设备分区表中不存在EFI分区且MBR标识为不可启动，则该存储设备不可启动。

进程管理

概述

进程是可以被操作系统调度执行的单元，也是静态的程序（文件）在运行过程中对应的实体。在现代操作系统中往往同时运行着成百上千的进程，但与此同时，能够参与计算的CPU资源是有限的并且远远少于进程的数量。因此，如何让这些进程协调有序的运行是操作系统中一个非常重要的话题。

在开始讨论进程调度之前，我们需要明确几个概念：

1. 主板上每一个插槽在操作系统中称为一个`Socket`，每一个插槽可以插入一个处理器，每个处理器上可能有多个`Core`，而每个`Core`可以并行运行多个`Thread`。比如我们常说的8核CPU通常是指该CPU可以并行运行8个`Thread`，而不是真的有8个`Core`。再比如Intel的酷睿双核处理器则是指该CPU有2个`Core`，每个`Core`一般运行4个`Thread`
2. 由于Linux和Windows在对线程和进程的概念实现上略有不同，在这里我们以Linux为准——即不区分进程和线程的概念，他们都是通过系统调用`fork`产生的。

进程的两种状态

我们知道，Linux在运行时被分为用户态和内核态两种执行状态。这两种状态并不仅仅是单纯的软件概念，而是真正的对应CPU硬件的执行状态。在一些Intel CPU下，一共有4种执行不同的执行状态，分别被成为Ring0~Ring3（越低权限越高）。它们有不同的内存访问权限，但是Linux在实现的时候只利用了其中两种，分别被对应用户态和内核态。

综上，一个进程也具有两种运行状态——在用户态下和在内核态下的状态。通常情况下，进程运行于用户态；但是，当进程需要访问硬件资源（比如读写磁盘，输出到屏幕，读写网络等）、进入休眠等时，则需要将控制权交由内核，由内核来完成接下的操作，递交控制的方式就是调用特殊的函数（系统调用）。

进程在内核中的表示

在内核中，操作系统里运行的每一个进程在内核中都由一个`struct task_struct`实例描述，该实例同时包含了进程的用户态信息和内核态信息，并且由全局变量`current`指出前CPU正在运行的进程所属的`struct task_struct`。所有的`task_struct`采用树形结构组织，也就是我们常说的进程树。我们先关注`task_struct`中的一些重要信息：`pid`，命名空间和内核栈。

首先来说说内核栈。总所周知，进程在用户空间态过程中主要由两种内存管理方式——堆和栈，栈用来管理函数调用时的内存和和信息，堆用来管理动态申请的内存。在内核态中依然沿用了这种方式，不过略微有一些区别。内核中堆内存的管理非常复杂，我们先只了解栈即可。内核栈实际上是以`struct task_struct`的一个成员出现的，其定义非常具有技巧性：

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
}
```

`thread_info`里包含了`task_struct`的指针和一些其它的信息。从这个联合体可以看出，`thread_info`总是位于内核栈的顶部，因此在内核态执行过程中，我们可以非常轻松的使用栈基址寄存器和栈大小计算出其位置并获取到相关信息。

接着是命名空间和PID，这两者可以说是密切相关的。命名空间是Linux中非常重要和强大功能，但它却很少被人们提及，它的一个重要应用就是在Linux中实现类似的`Docker`的容器服务。命名空间采用树状形态组织，当一个进程处在一个命名空间中时，它仅能感知到跟它处于同一个或更低级的命名空间的进程。这意味着，如果两个进程处在不同的命名空间里时，它们将无法互相感知和干扰，甚至具有不同的文件系统视图。子命名空间往往时随着新进程的产生而创建。命名空间的使用和`pid`脱不了关系——在子命名空间中出现的进程，也一定会在其父命名空间中出现。这意味着一个进程在不同的命名空间中具有不同的pid。

为了记录上述信息，我们不能在内核空间把PID像用户用空间一样简单解释为一个数字，而是采用结构结构体。但如果每一个`task_struct`都分配一个`struct pid`无疑会造成内存的浪费，因此内核中采用如下结构对`struct pid`进行管理：

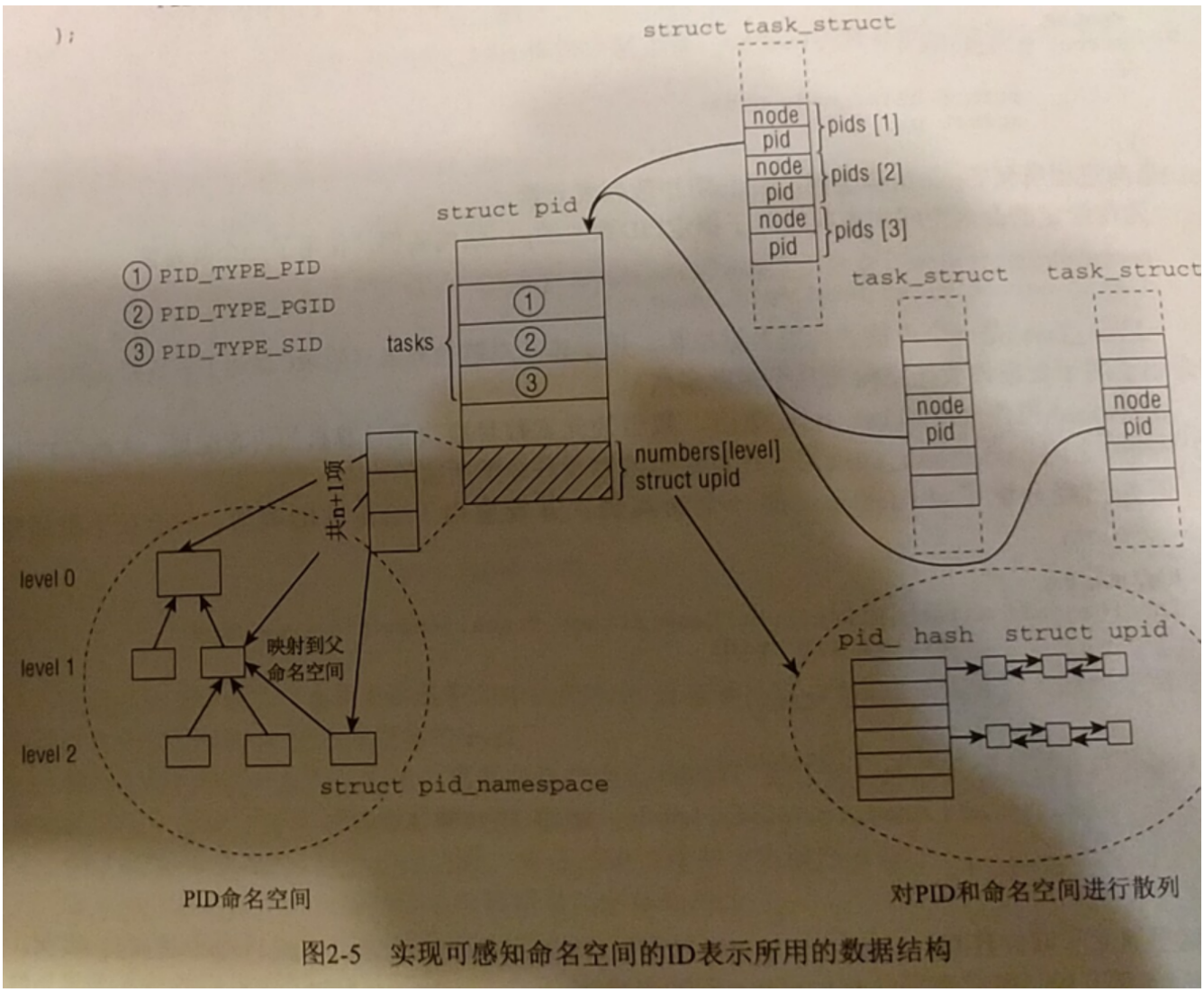


图2-5 实现可感知命名空间的ID表示所用的数据结构

进程的生命周期

在操作系统启动时，内核将会主动启动第一个用户态进程`init`进程，对应用户空间pid为0，该进程是之后所有进程的祖先。

要生成一个新的进程，在Linux中分为两步：

1. 使用`fork`或`vfork`或`clone`生成一个当前进程的副本
2. 调用`execve`函数族中的任意一个将子进程的地址空间完全替换，使之变为一个崭新的进程

在第一步中，虽然我们一共由3中选择来产生一个当前进程的副本，但实际上它们都调用了`do_fork`函数来完成真正的工作，这三个前端函数只是完成了用户空间参数提取和设置的工作。`fork`产生的副本将会完全拷贝当前进程，仅仅只有pid不同，其速度在写时拷贝技术出现之前非常慢；`vfork`是`fork`的轻量级版本，它与`fork`的主要区别是`vfork`不会拷贝父进程的地址空间，因此可以更快的产生新进程，并且`vfork`总是会让子进程先运行，这使得它被用来实现和`exec`函数族配合使用；`clone`一般被用于创建线程，在Linux中，线程实际上是共享一部分内存的进程。`do_fork`将会首先调用`copy_process`，然后确定子进程的PID，接着做一些`vfork`（如果是）和`ptrace`相关的工作，最后唤醒子进程。`copy_process`帮助`do_fork`完成了绝大部分拷贝工作，包括对`struct task_struct`的拷贝和初始化，它将根据标志位只拷贝需要拷贝的数据。

产生一个崭新进程的第二步是调用`exec`函数族，将旧的地址空间替换为新的（用户地址空间在内核中使用`struct mm_struct`表示）。整个过程包括释放原有进程资源、装载新的镜像、初始化堆栈、映射启动参数和环境变量。

进程的调度

Linux中进程调度目标是让每一个进程都能公平地分享CPU计算资源，在此之上，为了保证操作系统的可交互性，会尽可能的让交互性进程被尽快的响应（哪怕不能由很长的运行时间）。但由于CPU本身同一时间只能被一个进程使用，因此做到完全的公平是不可能的，因此只能保证在一个时间段内，所有可运行的进程都能得到相同的执行时间。

在开始讨论之前我们需要先明确几个概念：

1. 调度是在每一个CPU核（指每一个可运行代码的核，即前文中的`Thread`）上都会发生的，因此每一个CPU核都会有一个调度队列，它是一个`per-CPU viriable`。
2. Linux中存在多种调度算法，但是一个进程只能被一种调度算法调度
3. 可运行进程以红黑树组织，方便调度器快速筛选出最需要被运行的进程
4. 见图（进程状态转换图）

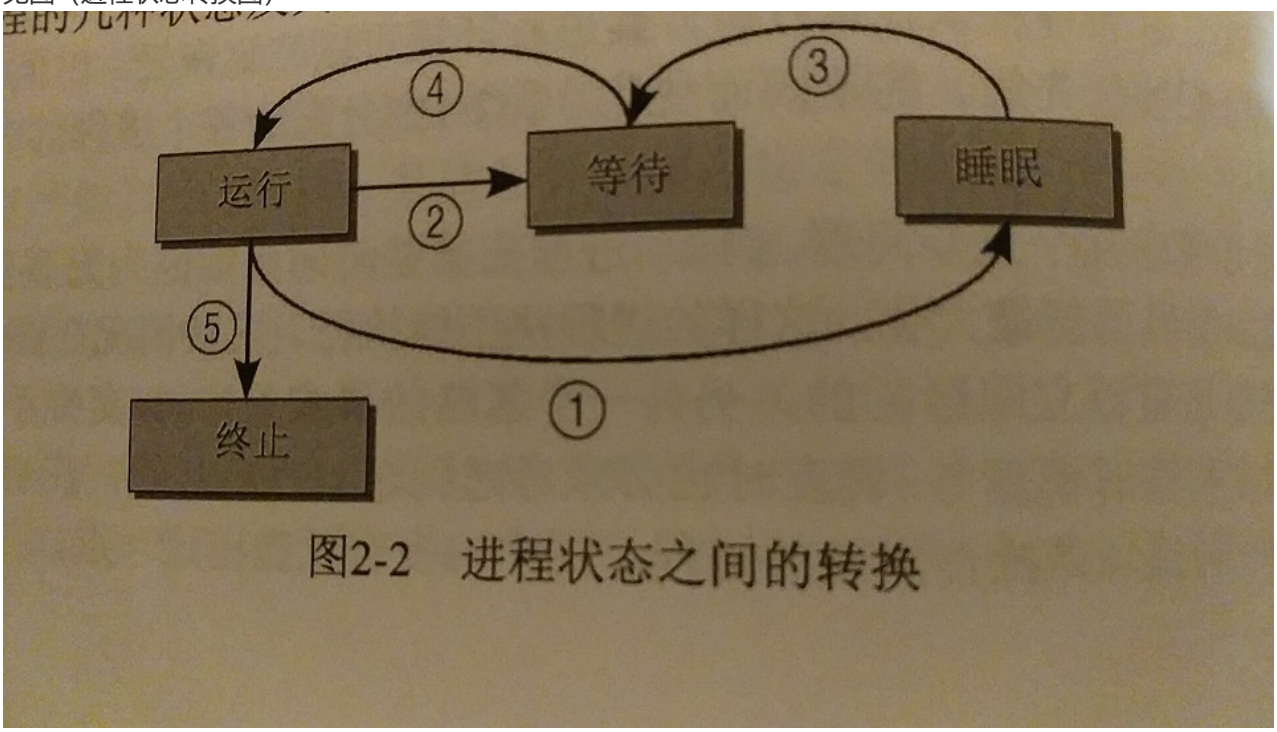
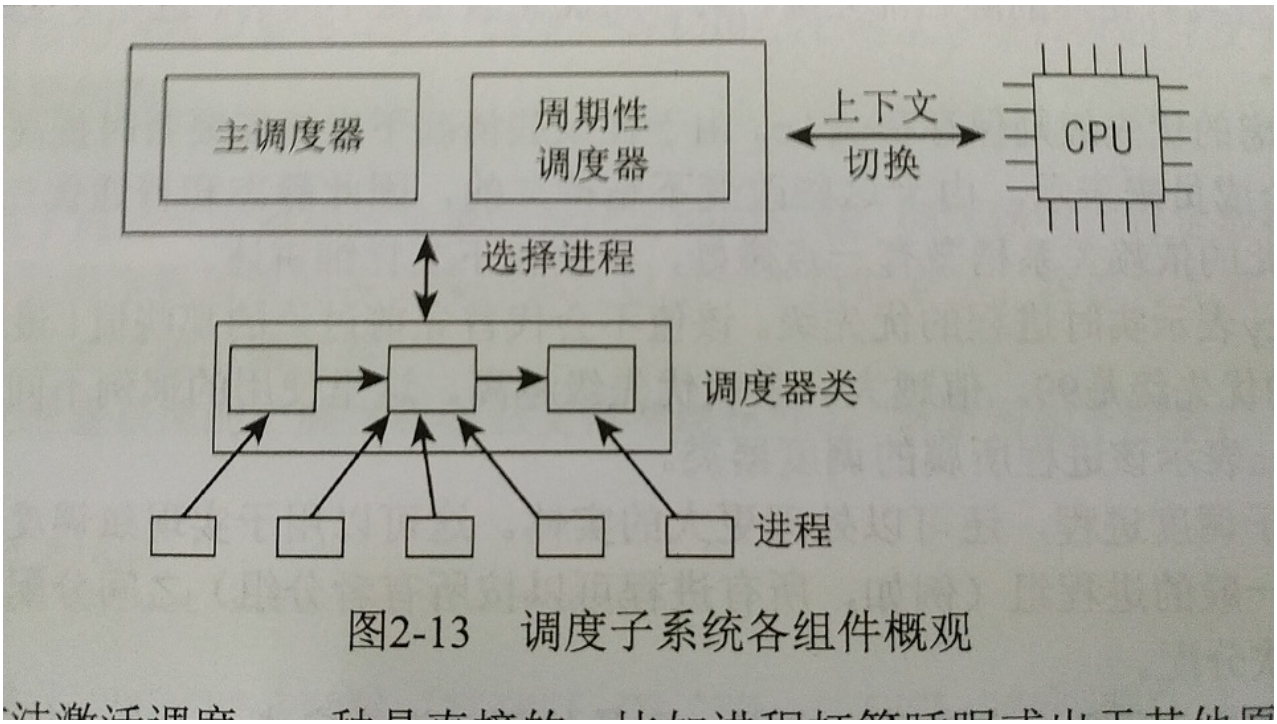


图2-2 进程状态之间的转换

5. 见图（调度器架构图），调度器被分为主调度器和周期调度器，调度器下可以有若干的调度类（用于代表不同的调度算法）



进程在Linux中被分为实时进程和普通进程，实时进程通常采用传统的时间片轮转和FIFO算法调度，并且比起普通进程具有更高的优先级；普通进程在2.6之后的版本都采用CFS，即完全公平调度算法调度。时间片轮转调度和FIFO调度算法的思想比较简单，这里就不再赘述。本文将着重讨论完全公平调度算法。

首先，为了评估一个进程有多需要被运行，Linux中定义了优先级的概念。优先级数字从0到139，其中0到99是专供实时进程使用的优先级，后100到139对应Linux中【-20,20】Nice值范围（越低越优先），供普通进程使用。先不说实时进程，普通进程的优先级如何体现在进程调度中呢？答案是通过权重值。Linux内核中有一张表专门用于存储Nice值和其对应权重的映射表，这个映射表里的权重值经过精心设计，使得Nice值每增加1，其权重值占总权重的比重将会下降10%。比如两个权重为1024的进程，应该各占50%的CPU时间，其中一个进程Nice加1，其权重变为820，经过计算，此时它将只能获得45%的CPU时间，比重比之原来下降了10%。

当然，仅有优先级是不够的，因为优先级一旦设置好，除非人为更改将不会发生变化——这会导致低优先级的进程永远无法被调度。CFS在权重的基础上进一步加强，使用了虚拟时间的概念，具体计算公式如下：

```
sum_exec_vruntime += delta_exec * (NICE_0_LOAD) / curr->load.weight
```

其中`curr->load.weight`是前文中提到的权重值，`delta_exec`时本次调度周期内进程运行的真实时间。但是由于内核中无法使用浮点除法，所以一般采用乘`curr->load.inv_weight`的方式，`curr->load.inv_weight`是`weight`的逆。每一个进程的`task_struct`中都有一个`sum_exec_vruntime`的成员，用以统计进程自执行以来的总计虚拟运行时间。每次进程调度时，内核会利用上述公式更新该成员。该公式保证了，所有进程的虚拟运行时间总是以相同的速率增长的。

另外，在CFS调度队列里还会利用周期调度器来周期性地维护一个自然增长的虚拟时间`vruntime`，该时间总是匀速增长。一个进程的虚拟运行时间如果超出`vruntime`越多，说明它得到的CPU时间越多，将来就会越少被调度。设置该时间的原因是，如果一个进程因为睡眠或主动放弃在相当一段时间内没有被调度，它的虚拟运行时会和`vruntime`差距非常小或相等——这使得它能够在需要调度的时候被立刻执行。

最后，关于何时启动调度器进行调度。Linux为我们选取了若干调度点（如系统中断开始，系统中断返回，睡眠等），当进程执行到这些可调度点时，内核将会运行主调度器来确认当前进程是否会被抢占。另外，周期调度器也会被时钟中断周期性唤醒来确认是否需要抢占并维护一些调度相关的数据。

调度中的CPU亲和性

由于调度是以CPU核为单位进行的，为了避免出现一核工作，七核围观的情况，内核将会需要从繁忙的CPU调度一些进程到空闲的CPU。但其中又涉及到CPU高速缓存的问题，这是一个非常复杂平衡问题，包括如何选取合适的CPU，合适的可调度进程。

针对进程组的调度

算是针对进程调度的粗粒度版本。

内核抢占

关键在于内核可以阻止自己被再次抢占，而用户进程不可以。

进程间通信

常用的手段有网络通信，IPC信号量（semaphore），信号（signal），消息队列（message queue），mmap共享内存。最广泛的应该是网络通信、信号和mmap共享内存。

内存管理

内存的硬件视图

我们对内存最直观的感受来自于内存条。但事实上，和内存关系最紧密的部件莫过于CPU，因为它是内存的直接使用者（之一），CPU在大部分情况下都依赖内存进行计算。在主板上，每个CPU都有自己专属的内存插槽，比如对于单CPU双核PC，每一个核对应两个内存插槽，那么我们称之为“双核四通道PC”（通道对应英文为“Channel”）。内存在硬件上有两种组织形式——UMA和NUMA。对于UMA（`Uniform Memory Access`）来说，无论主板上有多少个CPU，多少个Channel（插槽位），每一个CPU访问任意一个Channel的内存，其损耗都是一致的；对于NUMA（None-Uniformed Memory Access）来说，CPU拥有自己专属的内存`Channel`，并且在访问这些`Channel`时会有性能上的提升——因此主板上的内存对于一个CPU来说并不完全相同。但这并不意味着CPU不能访问非专属的内存，只是性能比起专属内存会有一定损失。

通常情况下，我们的家用机都是UMA结构的，只有服务器才会采用比较昂贵和复杂的NUMA方案。

Linux的内存布局

在32位机器不启动PAE的情况下，内核最多可以寻址4GB大小的内存，启动PAE后内核可以额外多4个Bit的空间来寻址，这意味着内核可以使用最多64GB的内存。在64位机器下，理论上内核可以使用高达256EB的内存，但是CPU设计师们目前只使用其中的48bit来寻址，即最多256PB的内存。再加上硬件限制，目前市面上最高端的8路服务器也只能支持最多24T的内存（截至2018年）。16路服务器非常非常罕见，因为当CPU太多时反而会因为一些问题影响性能，它可以支持48TB的内存。

32位下的全局内存布局

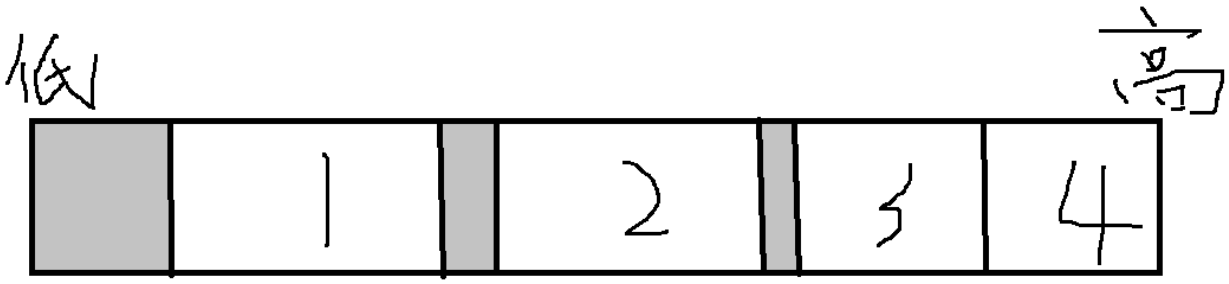
我们假设操作系统不启动PAE。内存布局会像下图：

这里的内存布局指的是虚拟内存的布局，代表的是一个进程在运行时观测到的内存布局。实际上在物理内存中，内核总是率先占据内存最开始的一部分空间。

64位下的全局内存布局

我们知道由于CPU设计者的限制，即使时在64位操作系统中，CPU也只能使用其后48位来进行寻址（硬件限制）。但是，在操作系统的虚地址拟空间中，我们依然需要使用64位的虚拟地址来寻址。那么如何处理这种情况呢？Linux的做法是使得前17位要么全为0，要么全为1。这就使得整个地址空间被天然地分为两个部分 `0x0` 到 `0x7fff ffff ffff ffff` 以及 `0xffff 8000 0000 0000` 到 `0xffff ffff ffff ffff`。我们可以这样理解，由于虚拟地址前16位无法被使用，因此Linux使其全为0或全为1，然后使用后48位中的最高位来将整个可用地址分为两截，即前17位全为0或全为1。这种操作可以非常方便的用符号位扩展实现。将整个虚拟空间划分为两半之后，分别交由用户空间和内核空间使用。如果用户尝试访问黑色部分中间地带，将会产生段错误。

内核空间内存布局

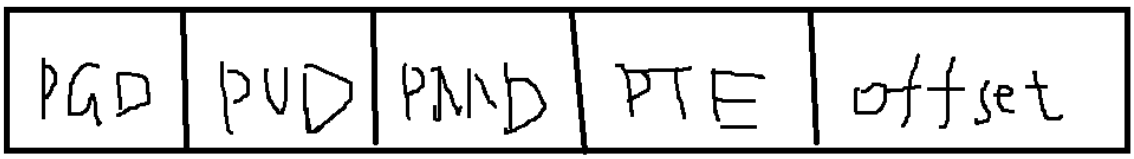


无论是在64位机器还是32位机器上，内核空间的*虚拟内存布局*都大致类似。上图中灰色区域代表中空地带，不会被内核使用的内存区域。另外，上图空间大小不代表其真实比例。

1号区域称之为直接映射区，通过简单的偏移量计算可以直接对应到物理内存上；2号区域称之为VMALLOC区域，该区域将使用页表来把物理上不连续的内存映射到该段连续的虚拟内存上；3号区域称之为持久映射区，属于高端内存的一部分，用于把内核无法直接映射的物理内存页映射至此处；4号区域称之为固定映射区，它可以映射到任意物理页上（也通过页表管理），但总是通过固定的公式来完成映射。由于该段的页映射关系不会被从TLB中换出，访问速度非常快，因此可以用于映射内核代码和固定内核模块的代码。

内核总是加载在物理内存的前几兆内存里。

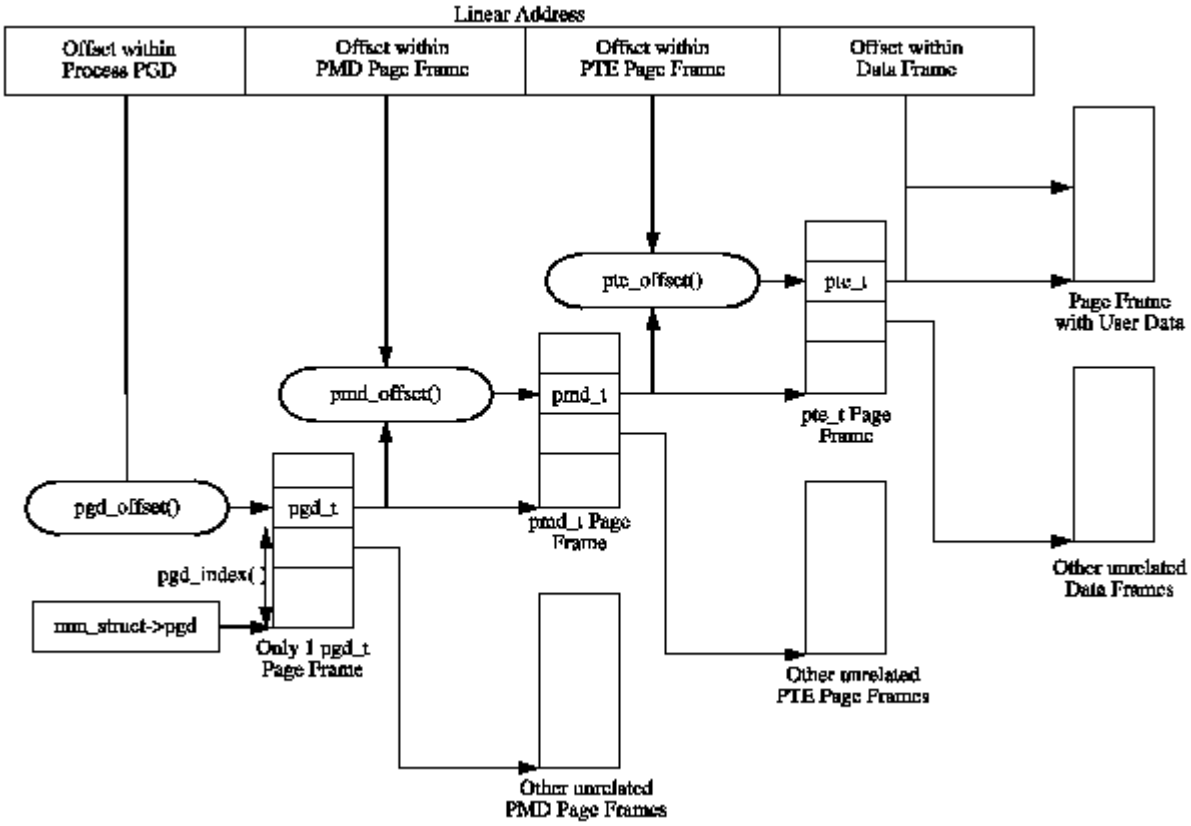
页表



多级页表查询的过程不再文字赘述，但有3点需要特殊说明：

1. TLB的存在可以极大加快页表查询速度
2. PTE的表项中除了页框偏移量，还包含一些页面的权限等额外信息
3. 由于系统初始化时页表还没有建立，因此内核采用最简单的位图来访问内存，在页表完成建立之后替换

页表的查询过程如下图：

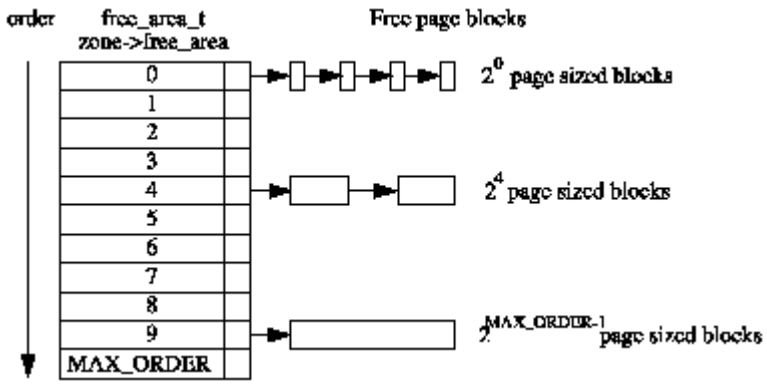


内核中的堆内存管理

内核同用户程序一样，在运行过程中也需要使用动态分配的内存。在内核中使用了两种机制来负责这一工作：

1. 著名的伙伴系统，用于页级别的内存分配
2. Slab分配器API，用于细粒度的内存分配和对象缓存

伙伴系统



伙伴算法在实际应用中有非常好的效果，但是它不能解决Linux长期以来的一个弊病——内存碎片化。为此，Linux借鉴磁盘整理的思路提出了反碎片的一个方案。该方案将页分为不可移动页，可回收页和可移动页。不可移动页意味着该物理页无法被移动（复制）到其他位置；可回收页意味着该页的内容不能被移动到其它位置，但是可以被删除并重新生成；可移动页意味着该页可以被自由地移动到任意位置。

同样，内存移动可能会造成CPU高速缓存失效的问题，另外在NUMA系统中，内存页的移动还会造成额外的性能损耗。

Slab

很多情况下，我们并不需要一整页那么大的内存，而只是需要几十个字节的空间来初始化一个对象，此时Slab分配器API就派上了用场。Slab API可以理解为是内核态下的`malloc`。首先我们需要明确几点：

1. Slab底层依然需要从伙伴系统中获取页
2. Slab负责已经申请的页的管理，用户不需要参与
3. Slub和Slob分配器是Slab的变种，用于适应不同的操作系统使用场景，比如Slub分配器在经常分配超过1页内存的场景下有非常好的性能表现，但对于前端使用者来说，通过`kmalloc`或者`kmem_cache_alloc`调用分配器时是感受不到这些差别的

slab分配器的内核架构如下图：

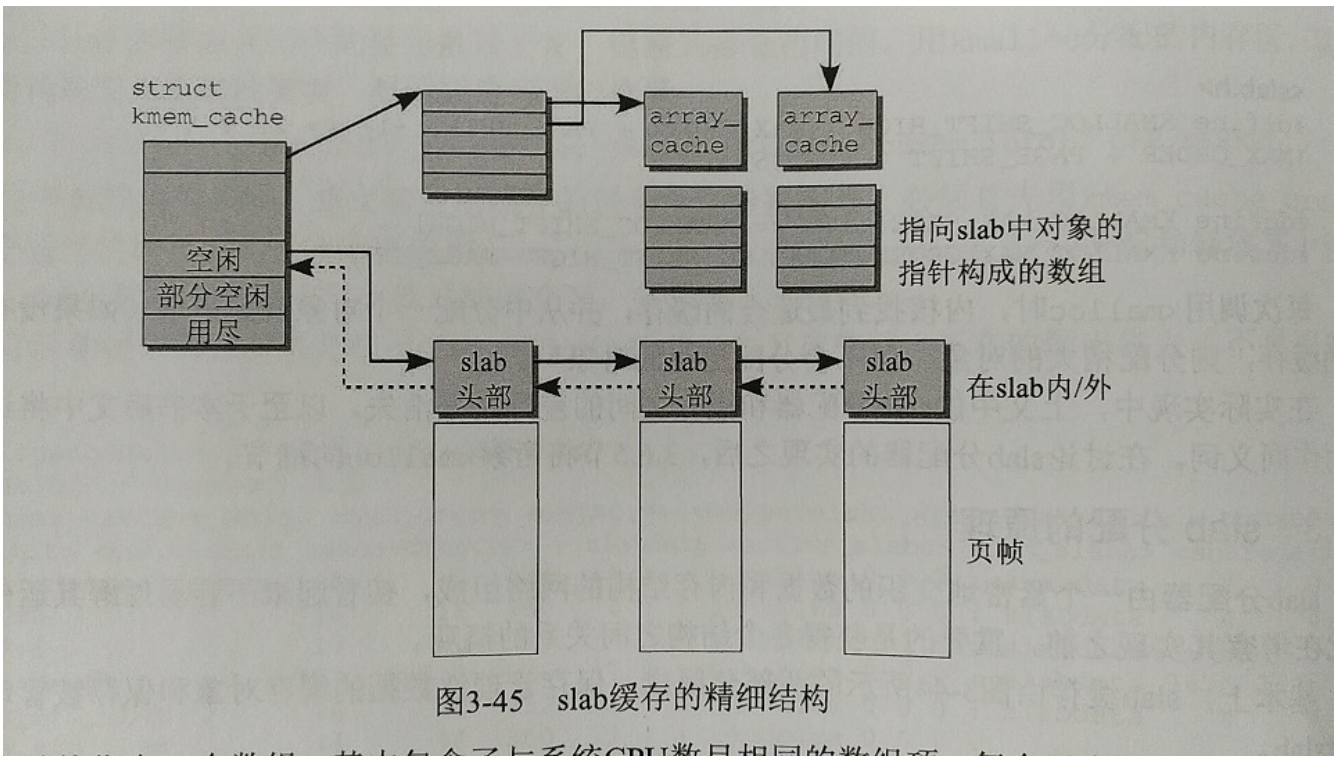


图3-45　slab缓存的精细结构

每一个`struct kmem_cache`对应一类对象缓存，这些对象可以是`struct task_struct`，可以是固定大小的32kb内存块。在用户空间的对象内存池可以设计的非常简单，甚至简单到就是一块预先分配好的固定大小的内存块，然后只需要在特定大小的空间上构造对象即可。Slab采用了类似的思路，但是作为内核重要的内存管理手段，它考虑的细节要多得多。比如设立`per-CPU`缓存`array_cache`，该缓存队列可以最大程度的利用CPU高速缓存。

关于Slab还有一个非常常见的话题——Slab着色。事实上，Slab着色并不是真的要给每一个对象选一个颜色，而是指通过偏移量的方式，让每个Slab中的对象拥有不同的偏移量，从而让不同Slab中的对象落入不同的CPU缓存行中。这样做的目的是为了在调取Slab对象时可以最大程度地利用高速缓存。

用户空间的内存布局

Linux中常见的可执行文件格式是ELF格式，内核会分析改格式，并提取出必要的内容，然后将这些文件内容使用mmap机制映射至内存中。具体请参见ELF。

用户空间中的堆内存管理

用户空间的堆内存通过glibc的`malloc/free`函数组进行申请和释放，具体的细节参见MallocInternals

应试和常见编程技巧

无锁数据结构

线程池的实现

异步编程