Linux调度系统全景指南(下篇)

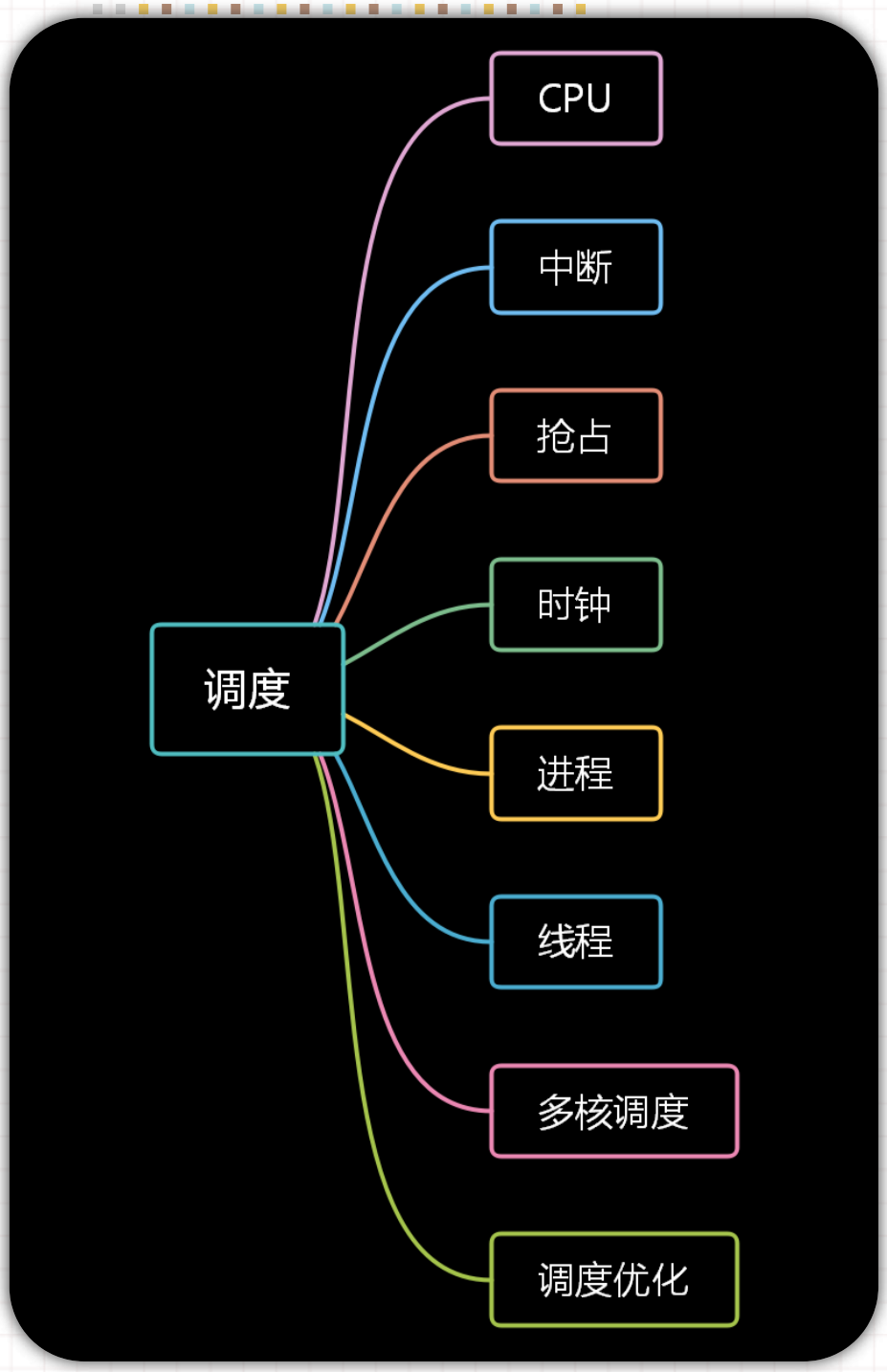
原创 Alex码农的艺术 极客重生 2021-02-26 03:50

收录于话题
#深入理解Linux系统 29 #原创文章 34

点击上方 蓝字 关注公众号，更多经典内容等着你

| 导语 本文主要是讲Linux的调度系统，由于全部内容太多，分三部分来讲，本篇是下篇（主要线程和进程），上篇请看（CPU和中断）：Linux调度系统全景指南(上篇)，调度可以说是操作系统的灵魂，为了让CPU资源利用最大化，Linux设计了一套非常精细的调度系统，对大多数场景都进行了很多优化，系统扩展性强，我们可以根据业务模型和业务场景的特点，有针对性的去进行性能优化，在保证客户网络带宽前提下，隔离客户互相之间的干扰影响，提高CPU利用率，降低单位运算成本，提高市场竞争力。欢迎大家相互交流学习！

目录



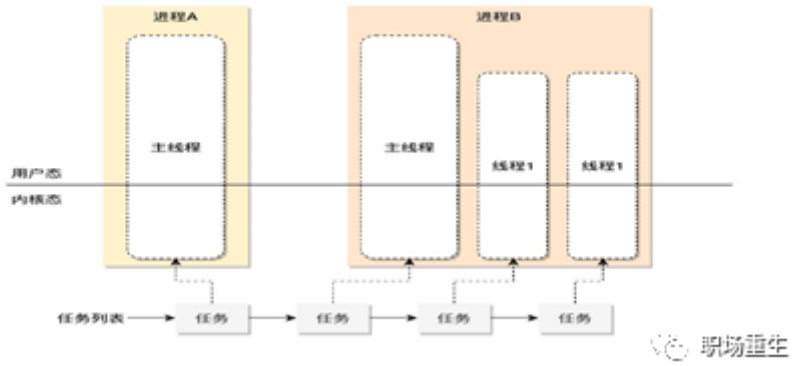
上篇请看（CPU和中断）：Linux调度系统全景指南(上篇)

中篇请看（抢占和时钟）：Linux调度系统全景指南(中篇)

进程

一般定义是操作系统对一个正在运行的程序的一种抽象， 是运行资源的管理单位（虚拟内存空间，文件句柄，全局变量，信号等运行资源），是操作系统资源分配的最小单位。在linux系统下，无论是

进程，还是线程，到了内核里面，我们统一都叫任务（Task），由一个统一的结构 task_struct 进行管理：



详细结构：

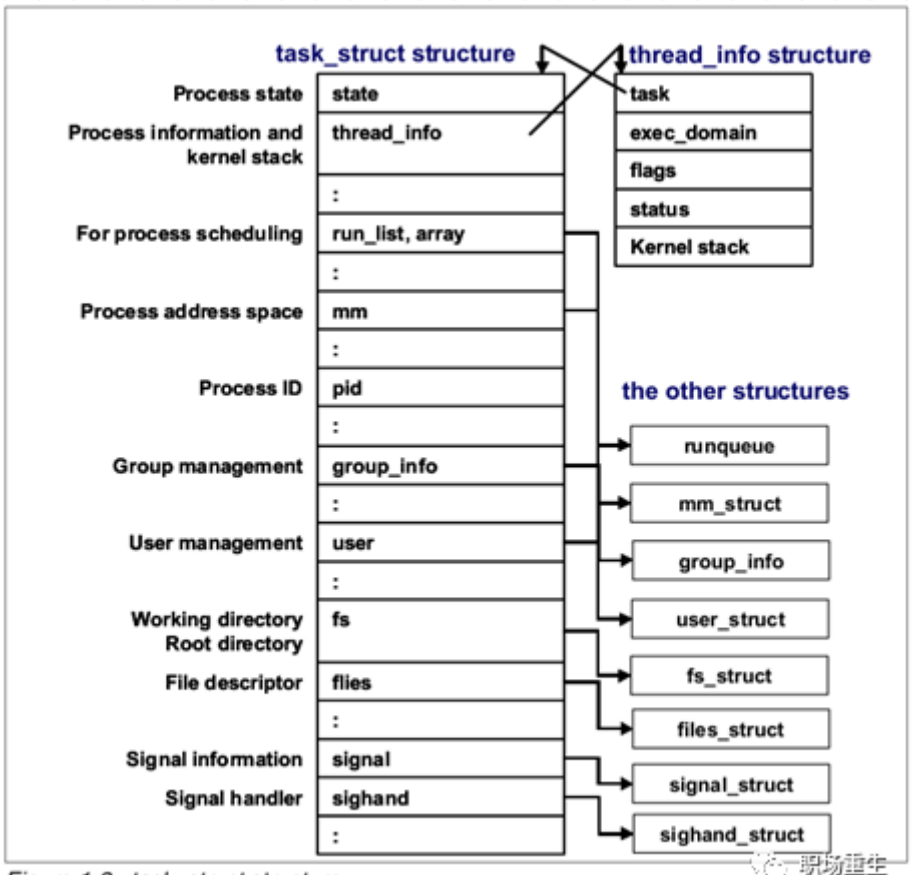
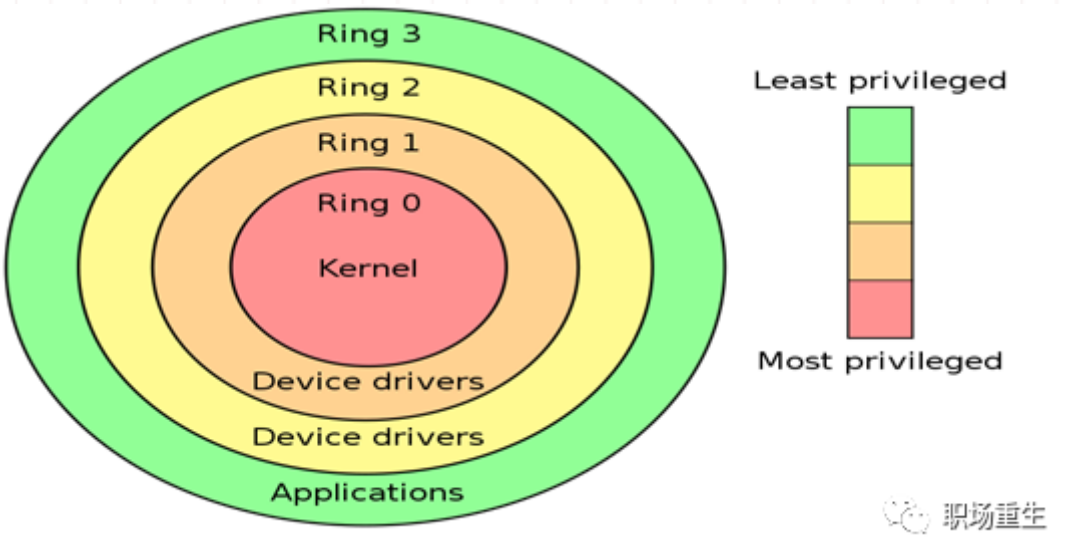


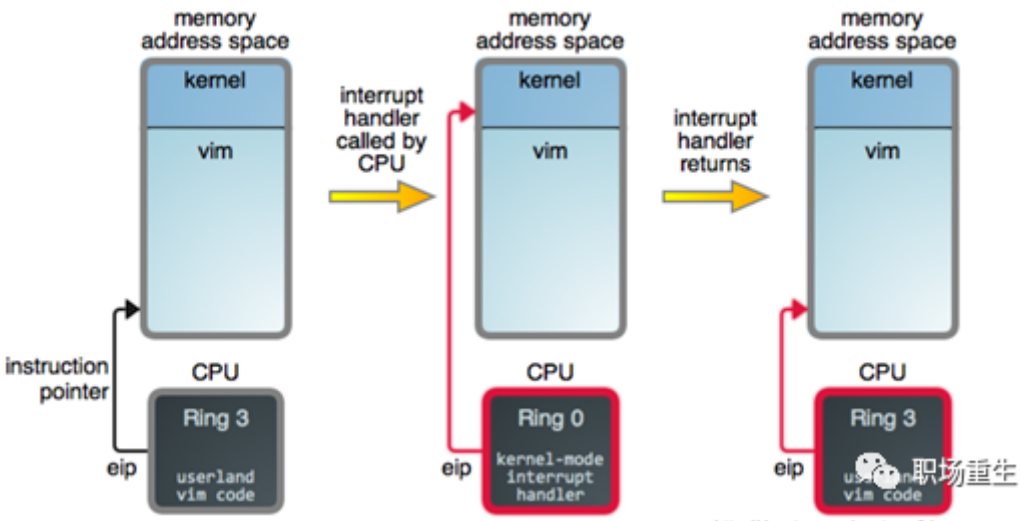
Figure 1-2 task_struct structure

大体分为下面几类：

进程运行空间

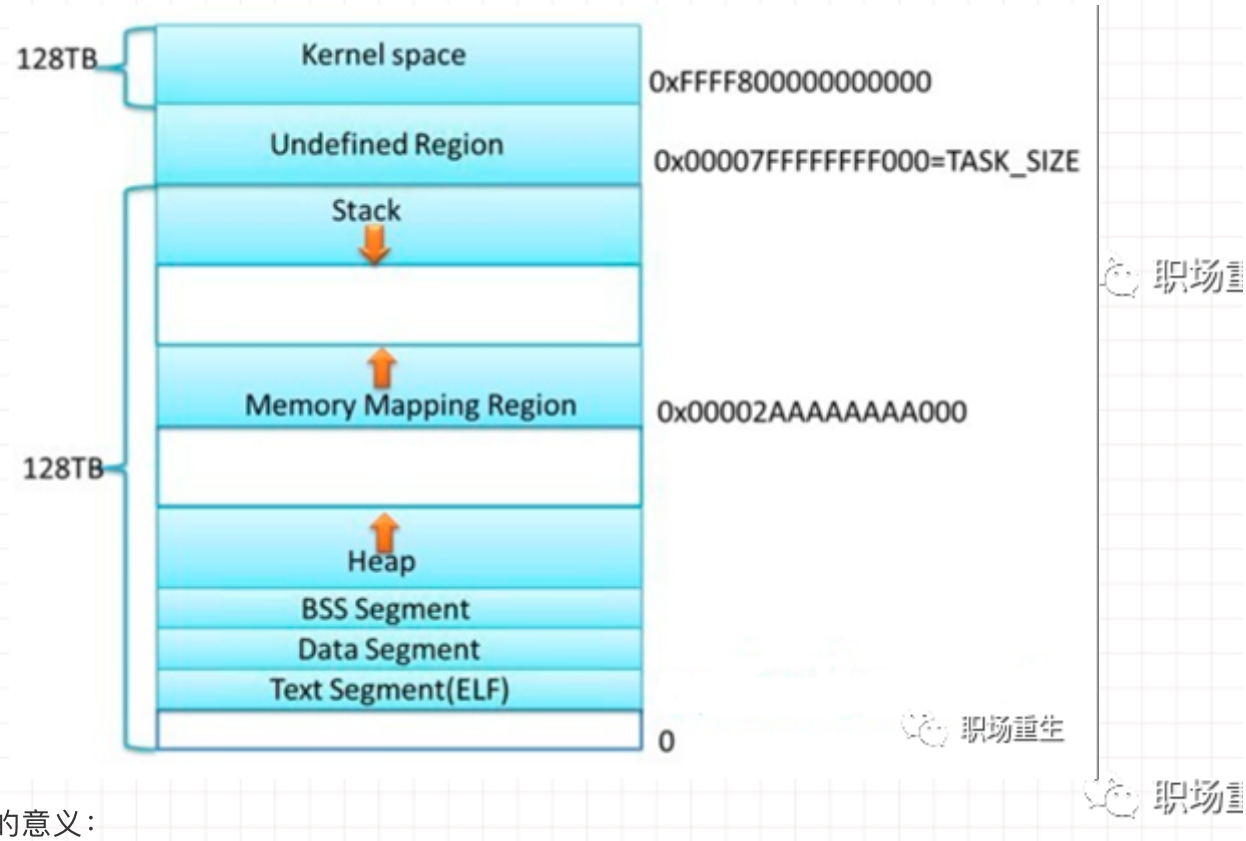
Linux 按照特权等级，把进程的运行空间分为内核空间和用户空间，分别对应着下图中，CPU 特权等级的 Ring 0 和 Ring 3。





- 内核空间（Ring 0）具有最高权限，可以直接访问所有资源；
- 用户空间（Ring 3）只能访问受限资源，不能直接访问内存等硬件设备，必须通过系统调用陷入到内核中，才能访问这些特权资源；
- 进程既可以在用户空间运行，又可以在内核空间中运行。进程在用户空间运行时，被称为进程的用户态，而陷入内核空间的时候，被称为进程的内核态。

进程内存空间（x86_64）



各个分区的意义：

内核空间：

在32位系统中，Linux会留1G空间给内核，用户进程是无法访问的，用来存放进程相关数据和内存数据，内核代码等；在64位系统里面，Linux会采用最低48位来表示虚拟内存，这可通过/proc/cpuinfo 来查看address sizes :

address sizes :

36 bits physical, 48 bits virtual，总的虚拟地址空间为256TB(2^48)，在这256TB的虚拟内存空间中，0000000000000000 - 00007fffffffffff(128TB) 为用户空间， ffff800000000000 - ffffffffffffffff(128TB)为内核空间, 剩下的是用户内存空间:

stack栈区：

专门用来实现函数调用-栈结构的内存块。相对空间下（可以设置大小，Linux 一般默认是8M，可通过 ulimit –s 查看），系统自动管理，从高地址往低地址，向下生长。

内存映射区：

包括文件映射和匿名内存映射， 应用程序的所依赖的动态库，会在程序执行时候，加载到内存这个区域，一般包括数据（data）和代码（text）;通过mmap系统调用，可以把特定的文件映射到内存中，然后在相应的内存区域中操作字节来访问文件内容，实现更高效的IO操作；匿名映射，在glibc中malloc分配大内存的时候会用到匿名映射。这里所谓的“大”表示是超过了MMAP_THRESHOLD设置的字节数，它的缺省值是 128 kB，可以通过 mallopt() 去调整这个设置值。还可以用于进程间通信IPC（共享内存）。

heap堆区：

主要用于用户动态内存分配，空间大，使用灵活，但需要用户自己管理，通过brk系统调用控制堆的生长，向高地址生长。

BBS段和DATA段：

用于存放程序全局数据和静态数据，一般未初始化的放在BSS段（统一初始化为0，不占程序文件的空间），初始化的放在data段，只读数据放在rodata段（常量存储区）。

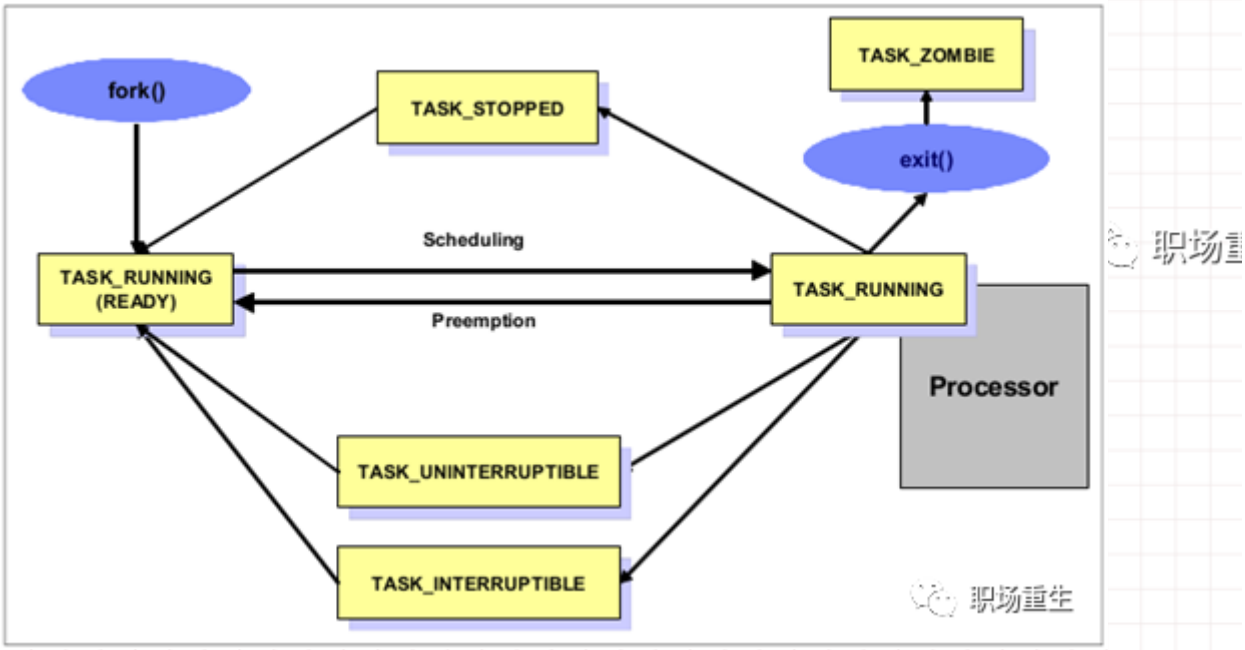
text段：

主要存放程序二进制代码。

进程调度

进程状态机

进程是一个动态的概念，是应用程序当前正在运行的一个实例, 在进程的整个生命周期中，它会处于不同的状态，并且在不同状态之间转化：



R (TASK_RUNNING)--执行状态

只有在该状态的进程才可能在CPU上运行。而同一时刻可能有多个进程处于可执行状态，这些进程的task_struct结构（进程控制块）被放入对 应CPU的可执行队列中（一个task最多只能出现在一个CPU的可执行队列中）。进程调度器的任务就是从各个CPU的可执行队列中分别选择一个task在该CPU上运行。很多操作系统教科书将正在CPU上执行的进程定义为RUNNING状态，而将可执行但是尚未被调度执行的进程定义为READY状态，这两种状态在linux下统一为 TASK_RUNNING状态。

S (TASK_INTERRUPTIBLE)--可中断的睡眠状态

处于这个状态的进程因为等待某个事件的发生（比如等待socket连接、等待信号量，等待锁），而被挂起。这些进程的task_struct结构被放入对应事件的等待队列中。当这些事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒。通过ps命令我们会看到，一般情况下，进程列表中的绝大多数进程都处于TASK_INTERRUPTIBLE状态（除非机器的负载很高）。毕竟CPU就一两个，进程动辄几十上百个，如果不是绝大多数进程都在睡眠，CPU又怎么响应得过来。

T (TASK_STOPPED or TASK_TRACED)--暂停状态或跟踪状态

向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK_STOPPED状态（除非该进程本身处于TASK_UNINTERRUPTIBLE状态而不响应信号）。（SIGSTOP与SIGKILL信号一样，是非常强制的。不允许用户进程通过signal系列的系统调用重新设置对应的信号处理函数。）向进程发送一个SIGCONT信号，可以让其从TASK_STOPPED状态恢复到TASK_RUNNING状态。

Z (TASK_DEAD - EXIT_ZOMBIE)--退出状态

进程成为僵尸进程。当父进程遗漏了用wait()函数等待已终止的子进程时，子进程就会进入一种无父进程的状态，此时子进程就是僵尸进程。

D (TASK_UNINTERRUPTIBLE)--不可中断的睡眠状态

TASK_INTERRUPTIBLE状态类似，进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指的并不是CPU不响应外部硬件的中断，而是指进程不响应异步信号。绝大多数情况下，进程处在睡眠状态时，总是应该能够响应异步信号的。否则你将惊奇的发现，kill -9竟然杀不死一个正在睡眠的进程了！于是我们也很好理解，为什么ps命令看到的进程几乎不会出现TASK_UNINTERRUPTIBLE状态，而总是 TASK_INTERRUPTIBLE状态。而TASK_UNINTERRUPTIBLE状态存在的意义就在于，内核的某些处理流程是不能被打断的。如果响应异步信号，程序的执行流程中就会被插入一段用于处理异步信号的流程（这个插入的流程可能只存在于内核态，也可能延伸到用户态），于是原有的流程就被中断了。

进程上下文切换

上下文切换(有时也称做进程切换或任务切换)：是指CPU从一个进程或线程切换到另一个进程或线程。简洁描述一下，上下文切换可以认为是内核（操作系统的核心）在 CPU 上对于进程（包括线程）进行以下的活动：

1. 挂起一个进程，将这个进程在CPU 中的状态（上下文）存储于内存中的某处；
2. 在内存中检索下一个进程的上下文并将其在CPU 的寄存器中恢复；
3. 跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程。

因此上下文是指某一时间点CPU寄存器和程序计数器的内容，广义上还包括内存中进程的虚拟地址映射信息。上下文切换只能发生在内核态中，上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的CPU时间，事实上，可能是操作系统中时间消耗最大的操作。Linux相比与其他操作系统（包括其他类Unix系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

进程优先级

1. 进程的优先级有动态优先级和静态优先级决定;
2. 它是决定进程在CPU的执行顺序的数字;
3. 优先级越高被CPU执行的概率越大;
4. 内核采用启发式算法决定是开启或者关闭动态优先级，可以通过修改nice级别直接修改进程的静态优先级，而获取更多CPU执行时间;
5. Linux支持的nice级别从19（最低优先级）到-20（最高优先级），默认只是0。只有root身份的用户才能把进程的nice级别调整为负数（让其具备较高优先级）。

时间片

- 时间片（timeslice）又称为“量子（quantum）”或“处理器片（processor slice）”是分时操作系统分配给每个正在运行的进程微观上的一段CPU时间（在抢占内核中是：从进程开始运行直到被抢占的时间）；
- 时间片由操作系统内核的调度程序分配给每个进程。首先，内核会给每个进程分配相等的初始时间片，然后每个进程轮番地执行相应的时间，当所有进程都处于时间片耗尽的状态时，内核会重新为每个进程计算并分配时间片，如此往复；

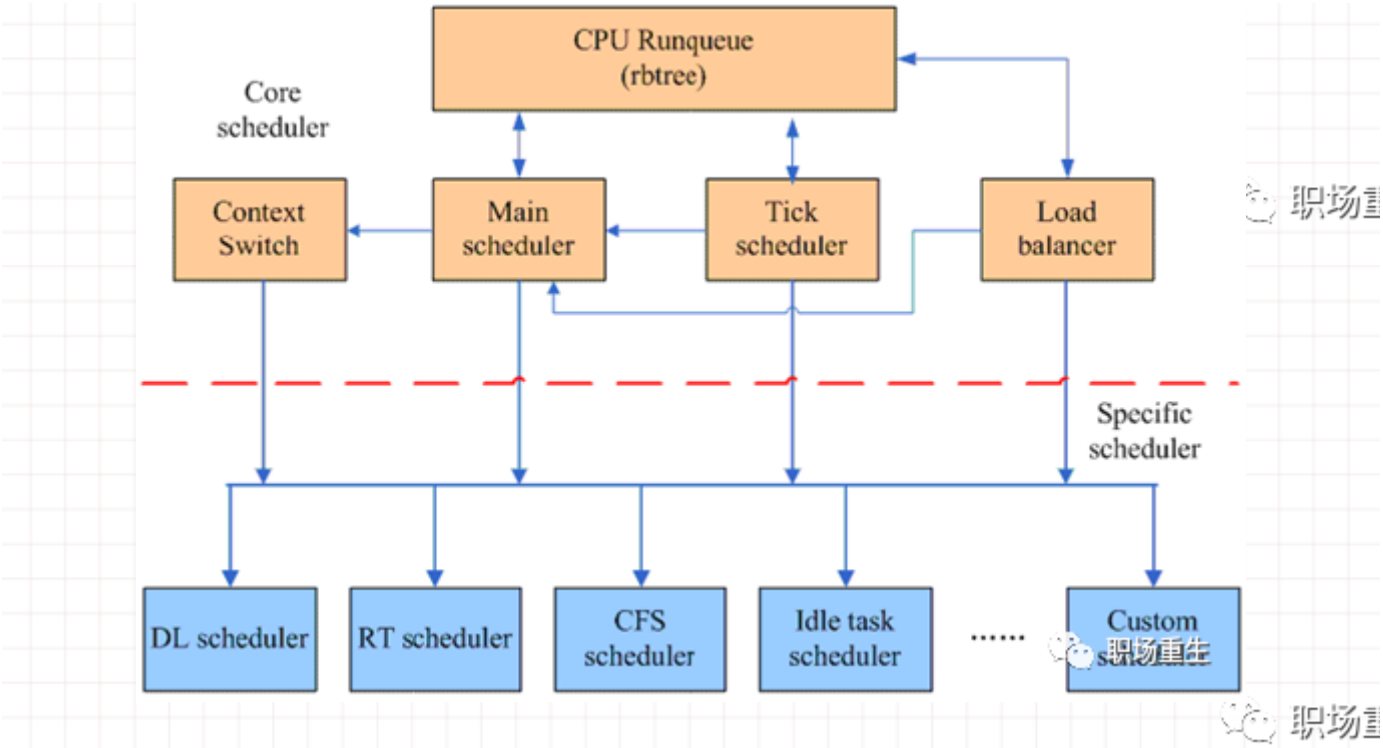
- 时间片设得太短会导致过多的进程切换，降低了CPU效率；而设得太长又可能引起对短的交互请求的响应变差，不同调度算法，对时间片管理不一样；
- 通常状况下，一个系统中所有的进程被分配到的时间片长短并不是相等的，尽管初始时间片基本相等（在Linux系统中，初始时间片也不相等，而是各自父进程的一半），系统通过测量进程处于“睡眠”和“正在运行”状态的时间长短来计算每个进程的交互性，交互性和每个进程预设的静态优先级（Nice值）的叠加即是动态优先级，动态优先级按比例缩放就是要分配给那个进程时间片的长短。一般地，为了获得较快的响应速度，交互性强的进程（即趋向于IO消耗型）被分配到的时间片要长于交互性弱的（趋向于处理器消耗型）进程。

调度框架

实际上进程是资源管理的单位，线程才是调度的单位，内核统称为任务调度。操作系统最重要的任务就是把系统中的task调度到各个CPU上去执行， 不同的任务有不同的需求，因此我们需要对任务进行分类：一种是普通进程，另外一种是实时进程。对于实时进程，毫无疑问快速响应的需求是最重要的，而对于普通进程，我们需要兼顾前三点的需求。相信你也发现了，这些需求是互相冲突的，对于这些time-sharing的普通进程如何平衡设计呢？这里需要进一步将普通进程细分为交互式进程（interactive processs）和批处理进程（batch process）。交互式进程需要和用户进行交流，因此对调度延迟比较敏感，而批处理进程属于那种在后台默默干活的，因此它更注重throughput的需求。当然，无论如何，分享时间片的普通进程还是需要兼顾公平，不能有人大鱼大肉，有人连汤都喝不上。为了达到这些设计目标，调度器必须要考虑某些调度因素，比如说“优先级”、“时间片”等。在Linux内核中，优先级就是实时进程调度的主要考虑因素。而对于普通进程，如何细分时间片则是调度器的核心思考点。过大的时间片会严重损伤系统的响应延迟，让用户明显能够感知到延迟，卡顿，从而影响用户体验。较小的时间片虽然有助于减少调度延迟，但是频繁的切换对系统的throughput会造成严重的影响。因为这时候大部分的CPU时间用于进程切换，而忘记了它本来的功能其实就是推动任务的执行。由于Linux是一个通用操作系统，它的目标是星辰大海，既能运行在嵌入式平台上，又能在服务器领域中获得很好的性能表现，此外在桌面应用场景中，也不能让用户有较差的用户体验。Linux任务调度算法核心就是解决调度优化问题：

（1）公平：对于time-sharing的进程，保证每个进程得到合理的CPU时间。
（2）高效：使CPU保持忙碌状态，即总是有进程在CPU上运行。
（3）响应时间：使交互用户的响应时间尽可能短。
（4）周转时间：使批处理用户等待输出的时间尽可能短。
（5）吞吐量：使单位时间内处理的进程数量尽可能多。

Linux调度器采用了模块化设计的思想，从而把进程调度的软件分成了两个层次，一个是core scheduler layer，另外一个是specific scheduler layer：



从功能层面上看，进程调度仍然分成两个部分，第一个部分是通过负载均衡模块将各个runnable task根据负载情况平均分配到各个CPU runqueue上去。第二部分的功能是在各个CPU的Main scheduler和Tick scheduler的驱动下进行单个CPU上的调度。调度器处理的task各不相同，有RT task，有normal task，有Deal line task，但是无论哪一种task，它们都有共同的逻辑，这部分被抽象成Core scheduler layer，同时各种特定类型的调度器定义自己的sched_class，并以链表的形式加入到系统中。这样的模块化设计可以方便用户根据自己的场景定义specific scheduler，而不需要改动Core scheduler layer的逻辑。

2个调度器

可以用两种方法来激活调度：

- 主调度器 ： 一种是直接的, 比如进程打算睡眠或出于其他原因放弃CPU；
- 周期性调度器：通过周期性的机制, 以固定的频率运行, 不时的检测是否有必要。

调度策略

linux内核目前实现了6种调度策略(即调度算法)，用于对不同类型的进程进行调度， 或者支持某些特殊的功能：

- SCHED_NORMAL和SCHED_BATCH调度普通的非实时进程；
- SCHED_FIFO和SCHED_RR和SCHED_DEADLINE则采用不同的调度策略调度实时进程；
- SCHED_IDLE则在系统空闲时调用idle进程；
- stop任务是系统中优先级最高的任务，它可以抢占所有的进程并且不会被任何进程抢占，其专属调度器类即stop-task；
- idle-task调度器类与CFS里要处理的SCHED_IDLE没有关系；
- idle任务会被任意进程抢占，其专属调度器类为idle-task；
- idle-task和stop-task没有对应的调度策略；
- 采用SCHED_IDLE调度策略的任务其调度器类为 **CFS**。

调度器类

- CFS (Completely_Fair_Scheduler)
- Real-Time Scheduler
- stop-task (sched_class_highest) Scheduler
- Deadline Scheduler: Earliest Deadline First (EDF) + Constant Bandwidth Server (CBS)
- Idle-task Scheduler

调度类顺序

- 优先级顺序：stop-task --> deadline --> real-time --> fair --> idle
- 在各调度器类定义的时候通过next指针定义好了下一级调度器类；
- stop-task是通过宏#define sched_class_highest (&stop_sched_class)指定的；
- 编译时期就已决定，不能动态扩展。

调度实体

这种一般性要求调度器不直接操作进程，而是处理可调度实体， 因此需要一个通用的数据结构描述这个调度实体，即seched_entity结构， 其实际上就代表了一个调度对象，可以是一个进程，也可以是一个进程组，linux中针对当前可调度的实时和非实时进程，定义了类型为seched_entity的3个调度实体:

- sched_dl_entity 采用EDF算法调度的实时调度实体；
- sched_rt_entity 采用Roound-Robin或者FIFO算法调度的实时调度实体；
- sched_entity 采用CFS算法调度的普通非实时进程的调度实体。

调度类算法

CFS（Completely Fair Scheduler）算法（完全公平调度器，对于普通进程）

- 设定一个调度周期（sched_latency_ns），目标是让每个进程在这个周期内至少有机会运行一次。就是每个进程等待CPU的时间最长不超过这个调度周期；
- 根据进程的数量，大家平分这个调度周期内的CPU使用权，由于进程的优先级即nice值不同，分割调度周期的时候要加权；
- 每个进程的经过加权后的累计运行时间保存在自己的vruntime字段里；
- 哪个进程的vruntime最小（红黑树pick_next）就获得本轮运行的权利。

Realtime Scheduler（实时）

实时系统是这样的一种计算系统：当事件发生后，它必须在确定的时间范围内做出响应。在实时系统中，产生正确的结果不仅依赖于系统正确的逻辑动作，而且依赖于逻辑动作的时序。换句话说，当系统收到某个请求，会做出相应的动作以响应该请求，想要保证正确地响应该请求，一方面逻辑结果要正确，更重要的是需要在最后期限（deadline）内作出响应。如果系统未能在最后期限内进行响应，那么该系统就会产生错误或者缺陷。在多任务操作系统中（如Linux），实时调度器（realtime scheduler）负责协调实时任务对CPU的访问，以确保系统中所有的实时任务在其deadline内完成，为了满足实时任务的调度需求，Linux提供了两种实时调度器：POSIX realtime scheduler（后文简称RT调度）和deadline scheduler（后文简称DL调度器）。

- Linux 支持SCHED_RR和SCHED_FIFO两种实时调度策略。

  - **先进先出（SCHED_FIFO）：** 没有时间片，被调度器选择后只要不被抢占，阻塞，或者自愿放弃处理器，可以运行任意长的时间。
  - **轮转调度（SCHED_RR）：** 有时间片，其值在进程运行时会减少。时间片用完后，该值重置，进程置于队列末尾。
- 两种调度策略都是静态优先级，内核不为这两种实时进程计算动态优先级。
- 这两种实现都属于 *软实时*。
- 实时优先级的范围：**0 ~ MAX_RT_PRIO-1**
  - MAX_RT_PRIO默认值为 **100**
  - 故默认实时优先级范围：**0 ~ 99**。

实时进程的优先级范围[0~99]都高于普通进程[100~139]，始终优先于普通进程得到运行，为了防止普通进程饥饿，Linux kernel有一个RealTime Throttling机制，就是为了防止CPU消耗型的实时进程霸占所有的CPU资源而造成整个系统失去控制。它的原理很简单，就是保证无论如何普通进程都能得到一定比例（默认5%）的CPU时间，可以通过两个内核参数来控制：

- /proc/sys/kernel/sched_rt_period_us
  缺省值是1,000,000 μs (1秒)，表示实时进程的运行粒度为1秒。（注：修改这个参数请谨慎，太大或太小都可能带来问题）。
- /proc/sys/kernel/sched_rt_runtime_us
  缺省值是 950,000 μs (0.95秒)，表示在1秒的运行周期里所有的实时进程一起最多可以占用0.95秒的CPU时间。
  如果sched_rt_runtime_us=-1，表示取消限制，意味着实时进程可以占用100%的CPU时间（慎用，有可能使系统失去控制）。

Deadline Task Scheduling

- DL调度器是根据任务的deadline来确定调度的优先顺序的：deadline最早到来的那个任务最先调度执行。对于有M个处理器的系统，优先级最高的前M个deadline任务（即deadline最早到来的前M个任务）将被选择在对应M个处理器上运行。
- Linux DL调度器还实现了constant bandwidth server（CBS）算法，该算法是一种CPU资源预留协议。CBS可以保证每个任务在每个period内都能收到完整的runtime时间。在一个周期内，DL进程的“活”来的时候，CBS会重新补充该任务的运行时间。在处理“活”的时候，runtime时间会不断的消耗；如果runtime使用完毕，该任务会被DL调度器调度出局。在这种情况下，该任务无法再次占有CPU资源，只能等到下一次周期到来的时候，runtime重新补充之后才能运行。因此，CBS一方面可以用来保证每个任务的CPU时间按照其定义的runtime参数来分配，另外一方面，CBS也保证任务不会占有超过其runtime的CPU资源，从而防止了DL任务之间的互相影响。
- 为了避免DL任务造成系统超负荷运行，DL调度器有一个准入机制，在任务配置好了period、runtime和deadline参数之后并准备加入到系统的时候，DL调度器会对该任务进行评估。这个准入机制保证了DL任务将不会使用超过系统的CPU时间的最大值。这个最大值在sched_rt_runtime_us和kernel.sched_rt_period_us sysctl参数中指定。默认值是950000和1000000，表示在1s的周期内，CPU用于执行实时任务（DL任务和RT任务）的最大时间值是950000μs。对于单个核心系统，这个测试既是必要的，也是充分的。这意味着：既然接受了该DL任务，那么CPU有信心可以保证其在截止日期之前能够分配给它需要的runtime长度的CPU时间。

- deadline调度器是仅仅根据实时任务的时序约束进行调度的，从而保证实时任务正确的逻辑行为。虽然在多核系统中，全局deadline调度器会面临Dhall效应（把若干个任务分配给若干个处理器执行其实是一个NP-hard问题（本质上是一个装箱问题），由于各种异常场景，很难说一个调度算法会优于任何其他的算法），不过我们仍然可以对系统进行分区来解决这个问题。具体的做法是采用cpusets的方法把CPU利用率高的任务放置到指定的cpuset上。开发人员也可以受益于deadline调度器：他们可以通过设计其应用程序与DL调度器交互，从而简化任务的时序控制行为。
- 在linux中，DL任务比实时任务（RR和FIFO）具有更高的优先级。这意味着即使是最高优先级的实时任务也会被DL任务延迟执行。因此，DL任务不需要考虑来自实时任务的干扰，但实时任务必须考虑DL任务的干扰。

Stop_Sched_Class

stop_sched_class用于停止CPU, 一般在SMP系统上使用， 用以实现负载平衡和CPU热插拔. 这个类有最高的调度优先级,stop调度器类实现了Unix的stop_machine 特性, stop_machine 是一个通信信号：在SMP的情况下相当于暂时停止其他的CPU的运行, 它让一个 CPU 继续运行，而让所有其他CPU空闲。在单CPU的情况下这个东西就相当于关中断。一般来说，内核会在如下情况下使用stop_machine技术:

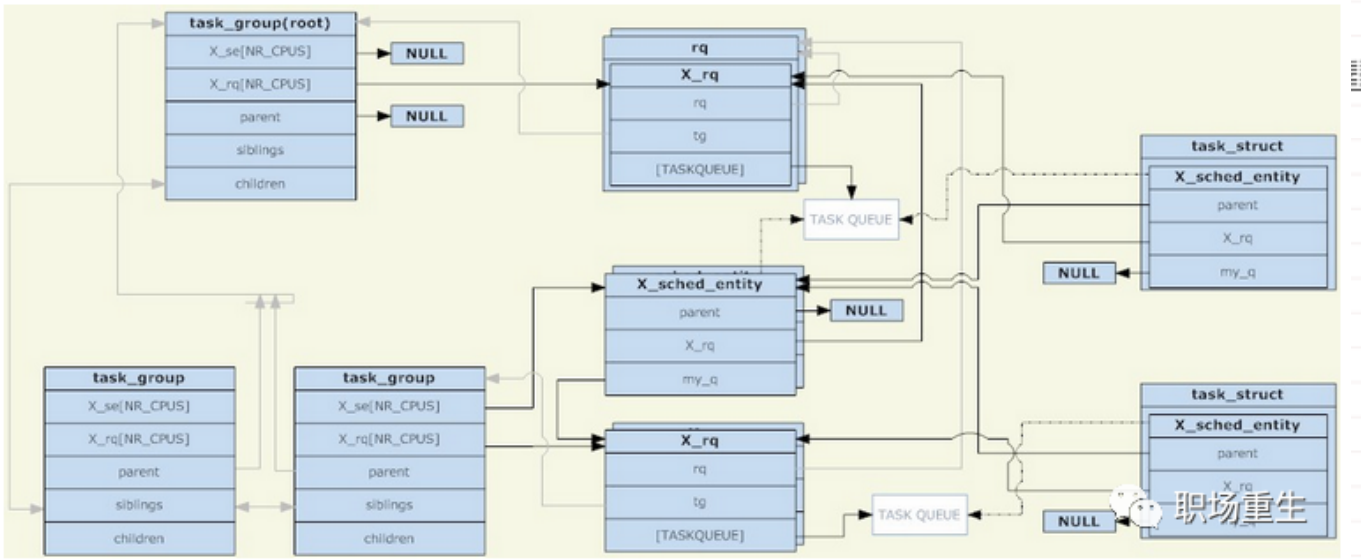
应用	描述
module install and remove	增加删除模块, 在不需要重启内核的情况下, 加载和删除模块
cpu hotplug	CPU的热插拔, 用以执行任务迁移的工作, cpu_stop_threads, 该任务由CPU绑定的migration内核线程来完成
memory hotplug	Memory的热插拔
ftrace	内核trace，debug功能, 参见kernel/trace/ftrace.c
hwlat_detector	检测系统硬件引入的latency，debug功能
Kernel Hotpatch	Ksplice可以在不到一秒时间里动态地应用内核补丁, 不需要重新引导

Idle_Sched_Class

idle任务会被任意进程抢占，其专属调度器类为idle-task，当CPU没有任务空闲时，默认的idle实现是hlt指令，hlt指令使CPU处于暂停状态，等待硬件中断发生的时候恢复，从而达到节能的目的。即从处理器C0态变到 C1态(见 ACPI标准)，让CPU置为WFI（Wait for interrupt）低功耗状态，以节省功耗，多cpu系统中每个cpu一个idle进程。

组调度

linux内核实现了control group功能（cgroup，since linux 2.6.24），可以支持将进程分组，然后按组来划分各种资源。比如：group-1拥有30%的CPU和50%的磁盘IO、group-2拥有10%的CPU和20%的磁盘IO、等等。cgroup支持很多种资源的划分，CPU资源就是其中之一，这就引出了组调度，linux内核中，传统的调度程序是基于进程来调度的, 以进程为单位来瓜分CPU资源，如果我们想把进程进行分组，以进程组进行瓜分CPU资源，Linux实现了组调度架构来实现这个需求：



Linux组调度实现架构

- 在linux内核中，使用task_group结构来管理组调度的组。所有存在的task_group组成一个树型结构（与cgroup的目录结构相对应），task_group可以包含具有任意调度类别的进程（具体来说是实时进程和普通进程两种类别），于是task_group需要为每一种调度策略提供一组调度结构。这里所说的一组调度结构主要包括两个部分，调度实体和运行队列（两者都是每CPU一份的）。调度实体会被添加到运行队列中，对于一个task_group，它的调度实体会被添加到其父task_group的运行队列，因为被调度的对象有task_group和task两种，所以需要一个抽象的结构来代表它们。如果调度实体代表task_group，则它的my_q字段指向这个调度组对应的运行队列；否则my_q字段为NULL，调度实体代表task。在调度实体中与my_q相对的是X_rq（具体是针对普通进程的cfs_rq和针对实时进程的rt_rq），前者指向这个组自己的运行队列，里面会放入它的子节点；后者指向这个组的父节点的运行队列，也就是这个调度实体应该被放入的运行队列；
- 调度发生的时候，调度程序从根task_group的运行队列中选择一个调度实体。如果这个调度实体代表一个task_group，则调度程序需要从这个组对应的运行队列继续选择一个调度实体。如此递归下去，直到选中一个进程。除非根task_group的运行队列为空，否则递归下去一定能找到一个进程。因为如果一个task_group对应的运行队列为空，它对应的调度实体就不会被添加到其父节点对应的运行队列中；

组的调度策略

组调度的主要针对rt（实时调度）和cfs（完全公平调度）两种类别：

实时进程的组调度

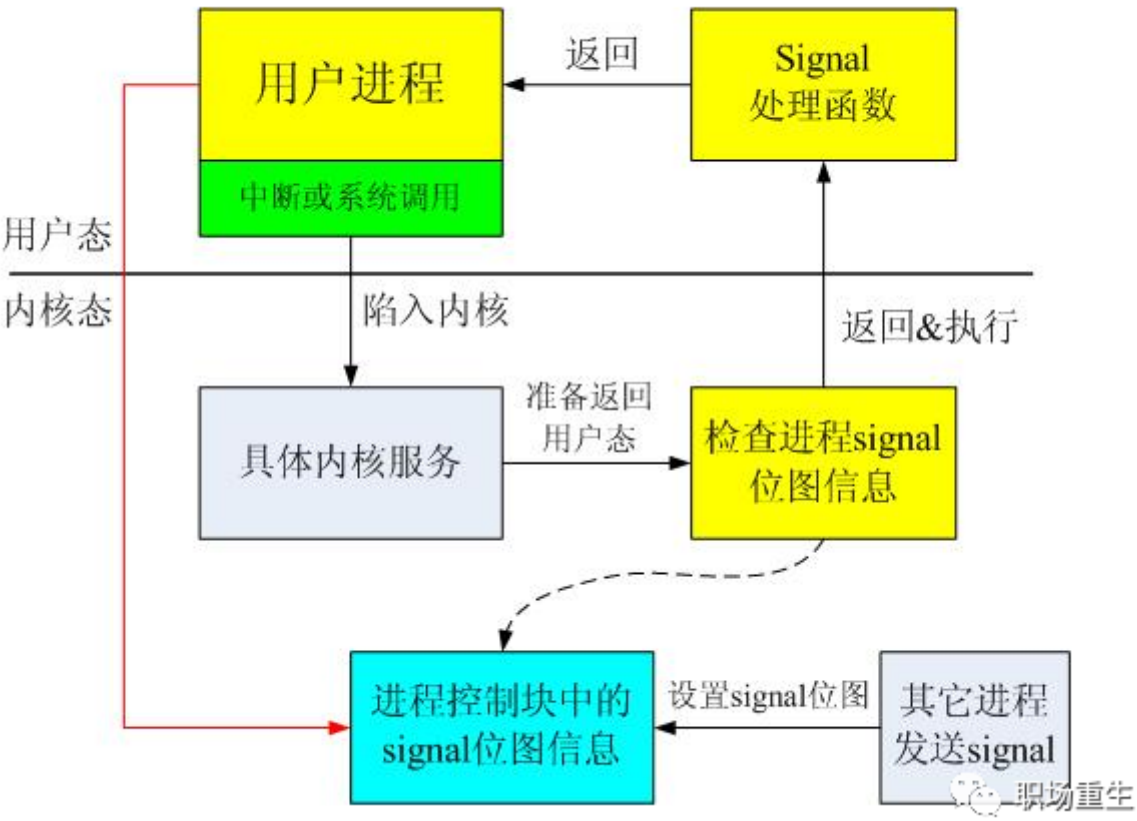
实时进程是对CPU有着实时性要求的进程，它的优先级是跟具体任务相关的，完全由用户来定义的。调度器总是会选择优先级最高的实时进程来运行，发展到组调度，组的优先级就被定义为“组内最高优先级的进程所拥有的优先级。

普通进程的组调度支持(Fair Group Scheduling)

2.6.23 引入的 CFS 调度器对所有进程完全公平对待。但是依然有个问题：设想当前机器有2个用户，有一个用户跑着9个进程，且都是CPU 密集型进程；另一个用户只跑着一个 X 进程，是交互性进程。从 CFS 的角度看，它将平等对待这 10 个进程，结果导致的是跑 X 进程的用户受到不公平对待，他只能得到约 10% 的 CPU 时间，让他的体验相当差。基于此，组调度的概念被引入[6]。CFS 处理的不再是一个进程的概念，而是调度实体(sched entity)，一个调度实体可以只包含一个进程，也可以包含多个进程。因此，上述例子的困境可以这么解决：分别为每个用户建立一个组，组里放该用户所有进程，从而保证用户间的公平性。该功能是基于控制组(control group, cgroup)的概念，需要内核开启 CGROUP 的支持才可使用。

信号处理

信号机制是进程之间相互传递消息的一种方法，信号全称为软中断信号，也有人称作软中断。从它的命名可以看出，它的实质和使用很像中断。所以，信号可以说是进程控制的一部分：



- 软中断信号（signal，又简称为信号）用来通知进程发生了异步事件。进程之间可以互相通过系统调用kill发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。注意，信号只是用来通知某进程发生了什么事件，并不给该进程传递任何数据；
- 当信号发送到某个进程中时，操作系统会中断该进程的正常流程，并进入相应的信号处理函数执行操作，完成后再回到中断的地方继续执行；
- 信号分类处理，第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处 理。第二种方法是，忽略某个信号，对该信号不做任何处理，就象未发生过

一样。第三种方法是，对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信 号的缺省操作是使得进程终止。进程通过系统调用signal来指定进程对某个信号的处理行为；

- 在进程表的表项中有一个软中断信号域，该域中每一位对应一个信号，当有信号发送给进程时，对应位置位。由此可以看出，进程对不同的信号可以同时保留，但对于同一个信号，进程并不知道在处理之前来过多少个；
-

信号分类：

（1） 与进程终止相关的信号。当进程退出，或者子进程终止时，发出这类信号。

（2） 与进程例外事件相关的信号。如进程越界，或企图写一个只读的内存区域（如程序正文区），或执行一个特权指令及其他各种硬件错误。

（3） 与在系统调用期间遇到不可恢复条件相关的信号。如执行系统调用exec时，原有资源已经释放，而目前系统资源又已经耗尽。

（4） 与执行系统调用时遇到非预测错误条件相关的信号。如执行一个并不存在的系统调用。

（5） 在用户态下的进程发出的信号。如进程调用系统调用kill向其他进程发送信号。

（6） 与终端交互相关的信号。如用户关闭一个终端，或按下break键等情况。

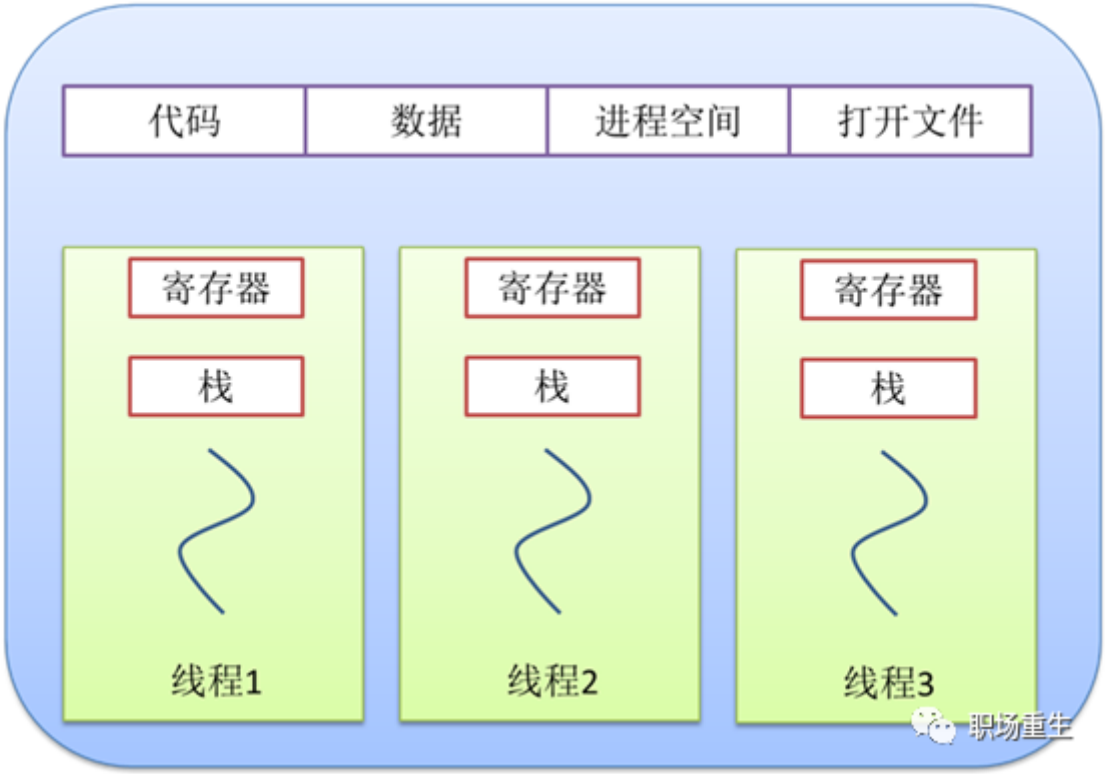
（7） 跟踪进程执行的信号。

多线程信号处理：

1. 不要在线程的信号掩码中阻塞不能被忽略处理的两个信号 SIGSTOP 和 SIGKILL。
2. 不要在线程的信号掩码中阻塞 SIGFPE、SIGILL、SIGSEGV、SIGBUS。
3. 确保 sigwait() 等待的信号集已经被进程中所有的线程阻塞。
4. 在主线程或其它工作线程产生信号时，必须调用 kill() 将信号发给整个进程，而不能使用 pthread_kill() 发送某个特定的工作线程，否则信号处理线程无法接收到此信号。
5. 因为 sigwait（）使用了串行的方式处理信号的到来，为避免信号的处理存在滞后，或是非实时信号被丢失的情况，处理每个信号的代码应尽量简洁、快速，避免调用会产生阻塞的库函数。

线程

线程（英语：thread）是操作系统能够进行运算调度的最小单位。大部分情况下，它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。在Unix System V及SunOS中也被称为轻量进程（lightweight processes），但轻量进程更多指内核线程（kernel thread），而把用户线程（user thread）称为线程。一个进程的组成实体可以分为两大部分：线程集和资源集。进程中的线程是动态的对象；代表了进程指令的执行。资源，包括地址空间、打开的文件、用户信息等等，由进程内的线程共享。

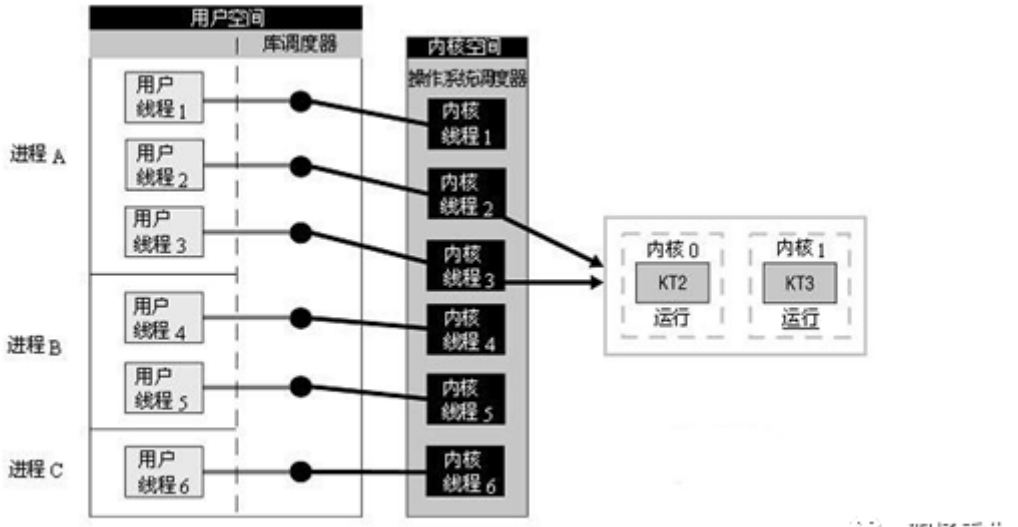


线程和进程的关系

根据操作系统内核是否对线程可感知，可以把线程分为**内核线程**和**用户线程**

名称	描述
用户级线程(User-LevelThread, ULT)	由应用程序所支持的线程实现, 内核意识不到用户级线程的实现
内核级线程(Kemel-LevelThread, KLT)	内核级线程又称为内核支持的线程

分类的标准主要是线程的调度者在核内还是在核外。前者更利于并发使用多处理器的资源，而后者则更多考虑的是上下文切换开销。Linux内核只提供了轻量进程的支持，限制了更高效的线程模型的实现，但Linux着重优化了进程的调度开销，一定程度上也弥补了这一缺陷。目前最流行的线程机制LinuxThreads所采用的就是“线程-进程”一对一模型（还存在多对一，多对多模型）用户级实现一个包括信号处理在内的线程管理机制。



Linux 线程实现采用内核级线程"一对一"模型

在linux系统下，无论是进程，还是线程，到了内核里面，我们统一都叫任务（Task），由一个统一的结构 task_struct 进行管理。一个进程由于其运行空间的不同，从而有内核线程和用户进程的区分。内核线程运行在内核空间，之所以称之为线程是因为它没有虚拟地址空间，只能访问内核的代码和数据； 而用户进程则运行在用户空间， 不能直接访问内核的数据但是可以通过中断，系统调用等方式从用户态陷入内核态，但是内核态只是进程的一种状态， 与内核线程有本质区别，用户进程运行在用户空间上，而一些通过共享资源实现的一组进程我们称之为线程组。Linux下内核其实本质上没有线程的概念，Linux下线程其实上是与其他进程共享某些资源的进程而已。但是我们习惯上还是称它们为线程或者轻量级进程。

内核线程

内核线程是直接由内核本身启动的进程。内核线程实际上是将内核函数委托给独立的进程，它与内核中的其他进程"并行"执行。内核线程经常被称之为内核守护进程，他们执行下列任务：

- 周期性地将修改的内存页与页来源块设备同步；
- 如果内存页很少使用，则写入交换区；
- 管理延时动作， 如 2 号进程接手内核进程的创建；
- 实现文件系统的事务日志；
- ...

内核线程主要有两种类型：

- 线程启动后一直等待，直至内核请求线程执行某一特定操作。
- 线程启动后按周期性间隔运行，检测特定资源的使用，在用量超出或低于预置的限制时采取行动。

内核线程由内核自身生成，其特点在于：

- 它们在CPU的内核态执行，而不是用户态；
- 它们只可以访问虚拟地址空间的内核部分（高于TASK_SIZE的所有地址），但不能访问用户空间。

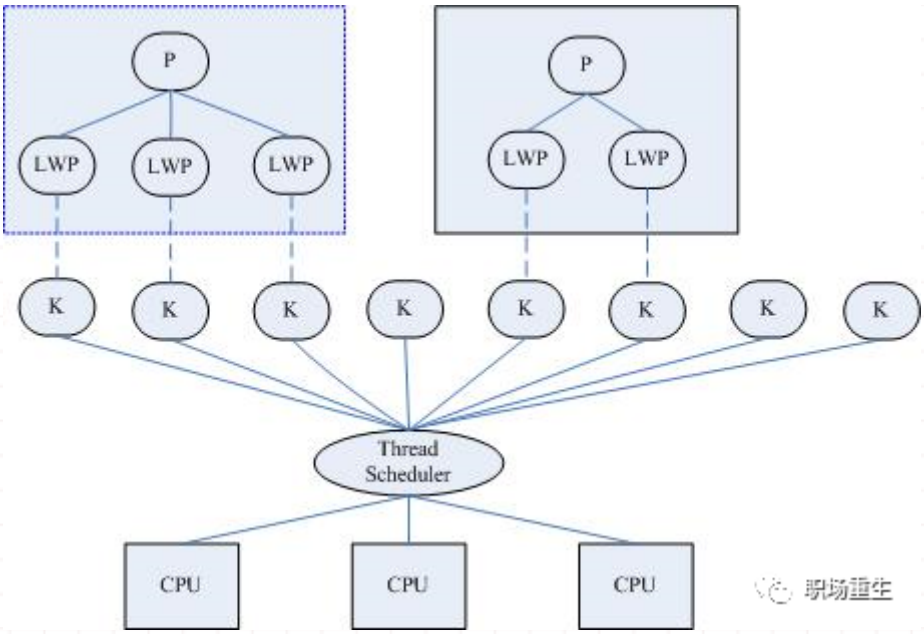
Linux在内核线程架构设计中，内核线程建立和销毁都是由操作系统负责、通过系统调用完成的。在内核的支持下运行，无论是用户进程的线程，或者是系统进程的线程，他们的创建、撤销、切换都是依靠内核实现的。线程管理的所有工作由内核完成，应用程序没有进行线程管理的代码，只有一个到内核级线程的编程接口. 内核为进程及其内部的每个线程维护上下文信息，调度也是在内核基于线程架构的基础上完成。内核线程驻留在内核空间，它们是内核对象。

内核线程就是内核的分身，一个分身可以处理一件特定事情。Linux内核使用内核线程来将内核分成几个功能模块，像kworker，kswapd， ksoftirqd，migration , rcu_bh，rcu_schd, watchdog等(内核线程都用[] 括起来)。这在处理异步事件如异步IO，阻塞任务，延后任务处理时特别有用。内核线程的使用是廉价的，唯一使用的资源就是内核栈和上下文切换时保存寄存器的空间，内核线程只运行在内核态，不受用户态上下文的拖累，在多核系统中，很多内核线程都是per cpu运行粒度。

用户线程

Linux内核只提供了轻量级进程(LWP)的方式支持用户线程，限制了更高效的线程模型的实现，但Linux着重优化了进程的调度开销，一定程度上也弥补了这一缺陷。目前最流行的线程机制LinuxThreads所采用的就是线程-进程"一对一"模型，调度交给核心，而在用户级实现一个包括信号处理在内的线程管理机制。轻量级进程由clone()系统调用创建，参数是CLONE_VM，即与父进程是共享进程地址空间和系统资源。

LinuxThreads是用户空间的线程库，所采用的是"线程-进程"1对1模型(即一个用户线程对应一个轻量级进程，而一个轻量级进程对应一个特定的内核线程)，将线程的调度等同于进程的调度，调度交由内核完成， 有了内核线程，每个用户线程被映射或绑定到一个内核线程。用户线程在其生命期内都会绑定到该内核线程。调度器管理、调度并分派这些线程。运行时库为每个用户级线程请求一个内核级线程。而线程的创建、同步、销毁由核外线程库完成（LinuxThtreads已绑定到 GLIBC中发行）。 在LinuxThreads中，由专门的一个管理线程处理所有的线程管理工作。当进程第一次调用pthread_create()创建线程时就会先 创建(clone())并启动管理线程。后续进程pthread_create()创建线程时，都是管理线程作为pthread_create()的调用者的子线程，通过调用clone()来创建用户线程，并记录轻量级进程号和线程id的映射关系，因此用户线程其实是管理线程的子线程。LinuxThreads只支持调度范围为PTHREAD_SCOPE_SYSTEM的调度，默认的调度策略是SCHED_OTHER。 用户线程调度策略也可修改成SCHED_FIFO或SCHED_RR方式， 这两种方式支持优先级为0-99, 而SCHED_OTHER只支持0。



Linux 轻量级进程实现

LinuxThtreads的设计存在一些局限性，导致后面Linux实现了新的线程库NPTL，或称为 Native POSIX Thread Library，是 Linux 线程的一个新实现，它克服了 LinuxThreads 的缺点，同时也符合 POSIX 的需求。与 LinuxThreads 相比，它在性能和稳定性方面都提供了重大的改进。与 LinuxThreads 一样，NPTL 也实现了一对一的模型。

轻量级进程具有局限性：

- 首先，大多数LWP的操作，如建立、析构以及同步，都需要进行系统调用。系统调用的代价相对较高：需要在user mode和kernel mode中切换。
- 其次，每个LWP都需要有一个内核线程支持，因此LWP要消耗内核资源（内核线程的栈空间）。因此一个系统不能支持大量的LWP。

优点：

（1）运行代价：LWP只有一个最小的执行上下文和调度程序所需的统计信息。

（2）处理器竞争：因与特定内核线程关联，因此可以在全系统范围内竞争处理器资源。

（3）使用资源：与父进程共享进程地址空间。

（4）调度：像普通进程一样调度。

协程

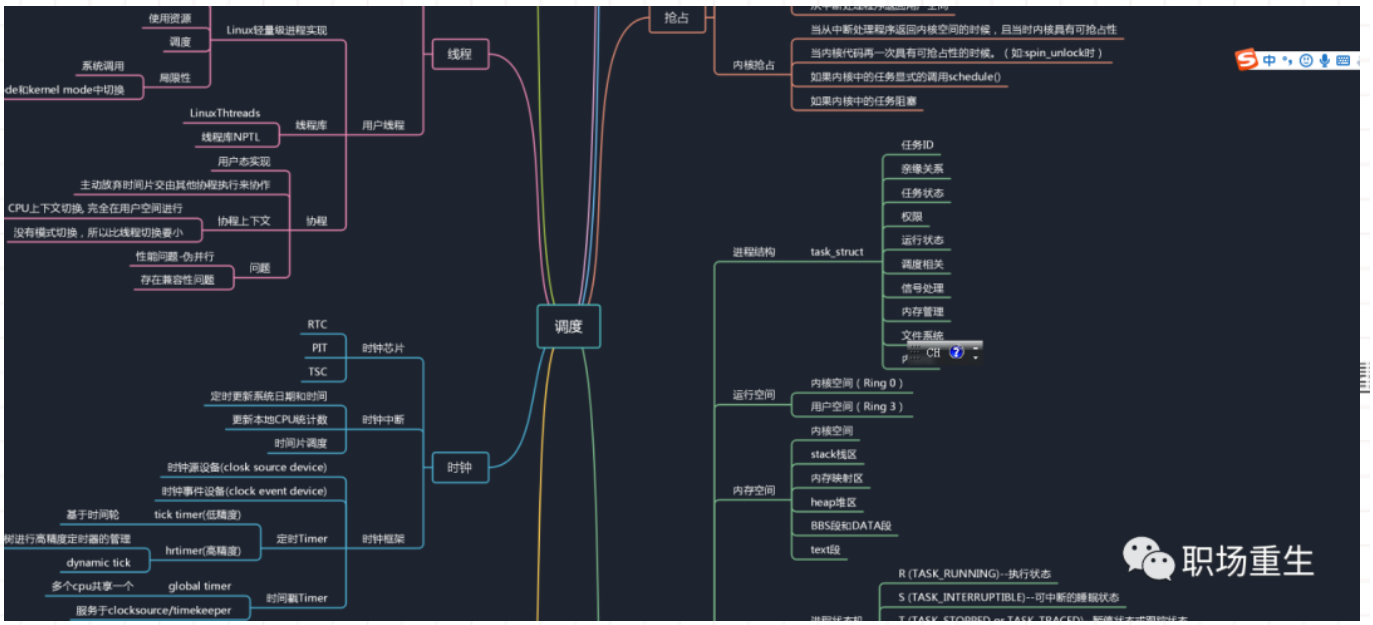
以上描述的不管是中断，进程，线程（内核线程，用户线程（轻量级进程实现））的调度都是由内核掌控，用户并不能直接干预，要在用户态实现对逻辑调度控制，需要实现类似用户级线程，用户级线程是完全建立在用户空间的线程库，用户线程的创建、调度、同步和销毁全部库函数在用户空间完成，不需要内核的帮助。因此这种线程是极其低消耗和高效的。协程本质上也是一种用户级线程实现，在一个线程（内核执行单元）内，协程通过主动放弃时间片交由其他协程执行来协作，故名协程。协程的一些关键点：

- 任何代码执行都需要上下文（CPU），协程也有自己的上下文，协程切换只涉及基本的CPU上下文切换，完全在用户空间进行，没有模式切换，所以比线程切换要小，开源libco 的协程切换的汇编代码，也就是二十来条汇编指令，一般切换代价： 协程 < 线程 < 系统调用 < 进程 ；

- 在多核多线程系统中，线程切换代价比较高（cache miss等），为了减少线程切换，希望在同一个线程内进行不同逻辑的伪并行（实际上还是串行），这样降低了代码逻辑切换代价，但这样并没有拥有真正并发带来的高性能，选择合适的使用场景；

- 协程存在兼容性问题，协程分为有栈协程和无栈协程实现，对不同系统和处理器要进行兼容，但不同系统对上下文实现，异常等不一样，这样就可能产生不兼容情况，一般用户态的代码都不关心底层差别，而使用协程后的代码兼容性变差。

本期结束，我们下期再见！



想要获取linux调度全景指南精简版，关注公众号回复“调度”即可获取。回复其他消息，获取更多内容；

◆ 往期推荐 ◆

Linux调度系统全景指南(上篇)

Linux调度系统全景指南(中篇)

如何成为一名大厂的优秀员工?

C++模版的本质

C++内存管理全景指南

云网络丢包故障定位全景指南

收录于话题 #深入理解Linux系统 29

< 上一篇
Linux调度系统全景指南(中篇)

下一篇 >
Linux调度系统全景指南(终结篇)