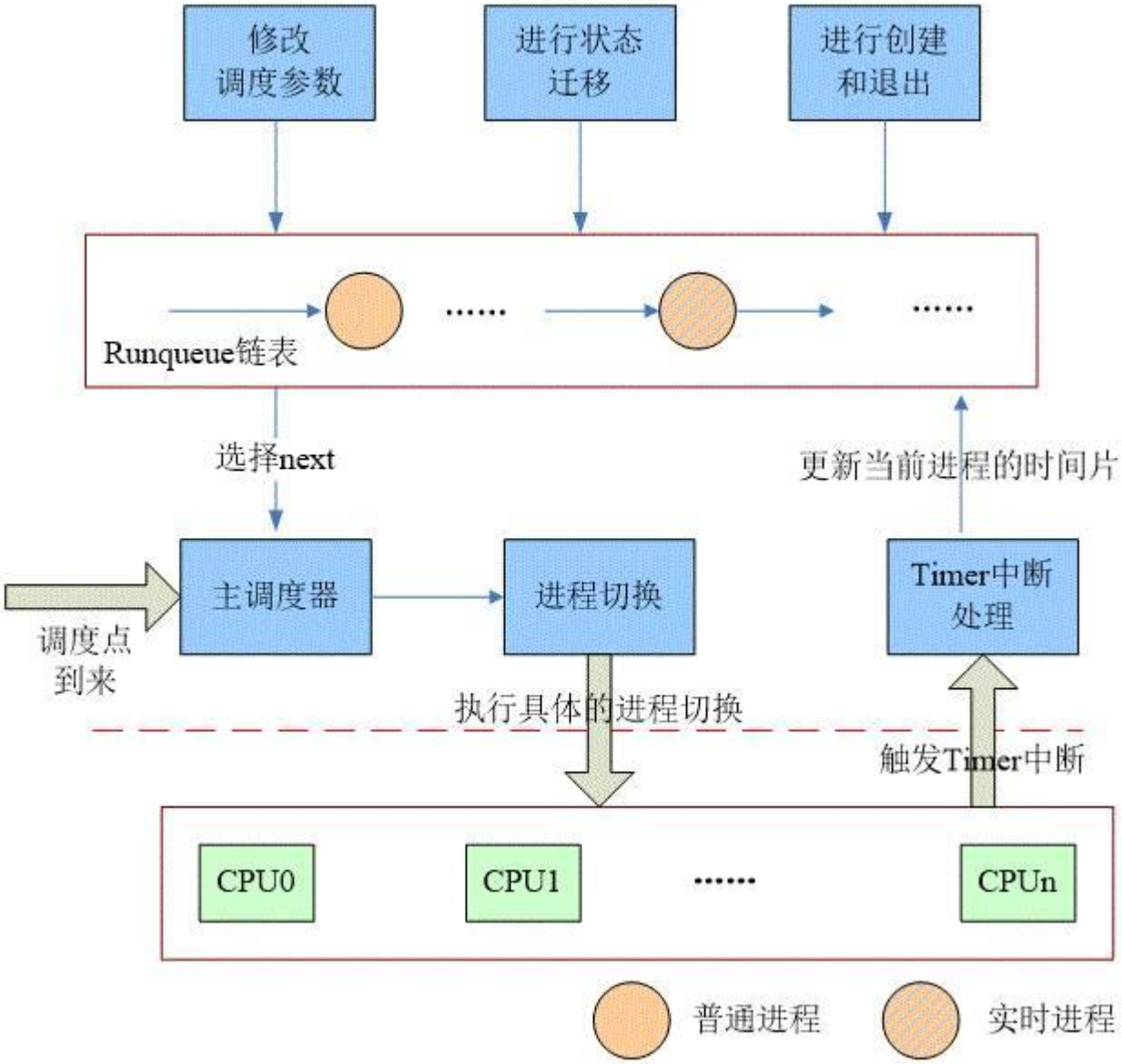
3.3 Linux操作系统概述

西安邮电大学

基本调度模型


修改
调度参数
进行状态
迁移
进行创建
和退出
Runqueue链表
选择next
更新当前进程的时间片
主调度器
进程切换
Timer中断
处理
调度点
到来
执行具体的进程切换
触发Timer中断
CPU0
CPU1
CPUn
普通进程
实时进程

调度基本模型

所谓调度就是从就绪队列中选择一个进程投入CPU运行。调度的主战场是就绪队列，核心是调度算法，实质性的动作是进程切换，对于以时间片调度为主的调度，时钟中断是驱动力，确保每个进程在CPU上运行一定的时间，在调度的过程中，用户还可以可以通过系统调用nice等调整优先级，比如降低自己的有优先级等，当然也涉及进程状态的转换。新创建的进程加入就绪队列，退出的进程从队列中删除。

从图可以看出，所有的CPU的所有的进程都存放在一个就绪队列中，从中选中一个进程进行调度的过程，是从这队列上的一种线性查找，因此其算法复杂度为O(n)，详细调度过程和代码分析请参加教材3.4节的进程调度，这是针对2.4版内核的代码分析，比较简单，看起来相对容易，本次课，主要讲解进程调度的演变过程，以指导大家阅读高版本的内核源代码。

调度的主战场-就绪队列

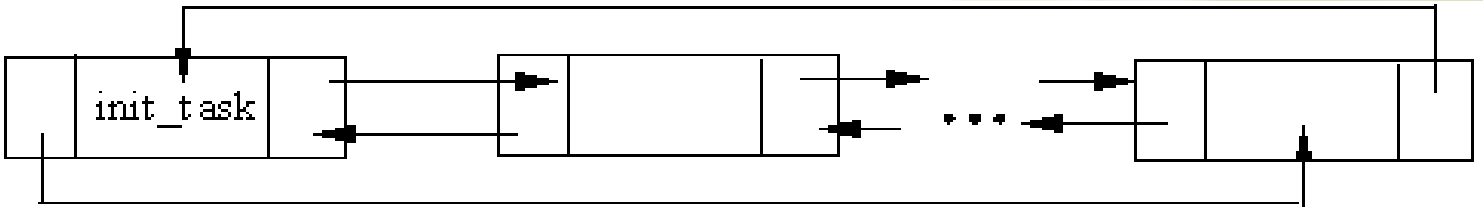
那么，在内核代码中，如何表示就绪队列？

把就绪状态的进程组成一个双向循环链表，也叫就绪队列（runqueue）

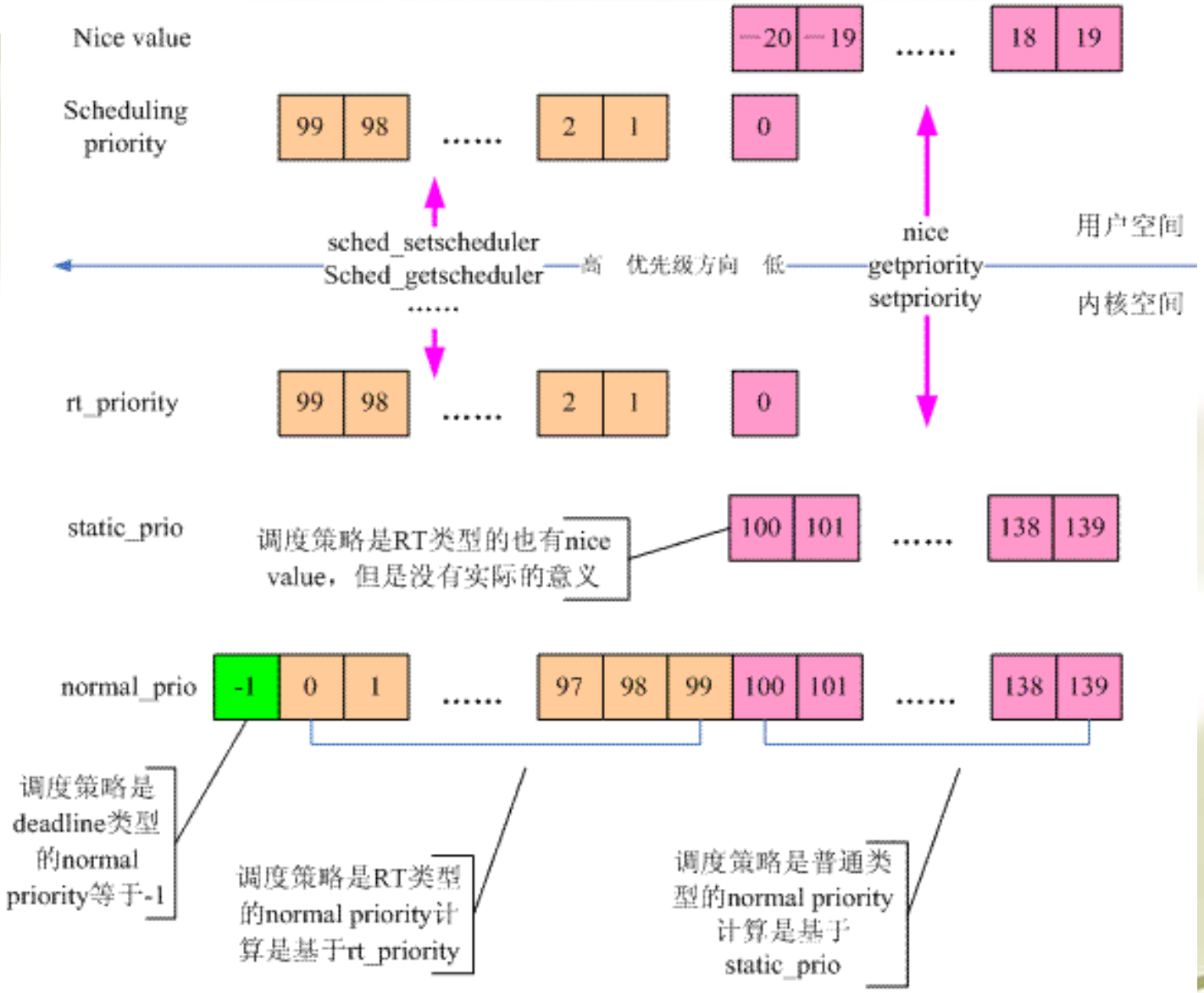
在task_struct结构中定义了队列结构。

```
struct task_struct{
                ...
                struct list_head run_list;
                ...
    };
```

init_task（0号进程的PCB）为队头。



进程调度-进程优先级


Nice value
-20
-19
.....
18
19
Scheduling priority
99
98
.....
2
1
0
sched_setscheduler
Sched_getscheduler
.....
高
优先级方向
低
nice
getpriority
setpriority
用户空间
内核空间
rt_priority
99
98
.....
2
1
0
static_prio
调度策略是RT类型的也有nice value，但是没有实际的意义
100
101
.....
138
139
normal_prio
-1
0
1
.....
97
98
99
100
101
.....
138
139
调度策略是deadline类型的normal priority等于-1
调度策略是RT类型的normal priority计算是基于rt_priority
调度策略是普通类型的normal priority计算是基于static_prio

进程调度-进程的优先级

在进程调度算法中，优先级是一个很重要的因素，

我们从两个角度来看优先级

首先看用户空间，有两种优先级：

（1） 普通优先级（nice ）：从-20～19，数字越小，优先级越高，通过修改这个值可以改变普通进程获取cpu资源的比例。

（2）调度优先级（scheduling priority）。从1（最低）~99（最高），这是实时进程的优先级。当然，普通进程也有调度优先级（scheduling priority），被设定为0。

进程调度-内核空间优先级

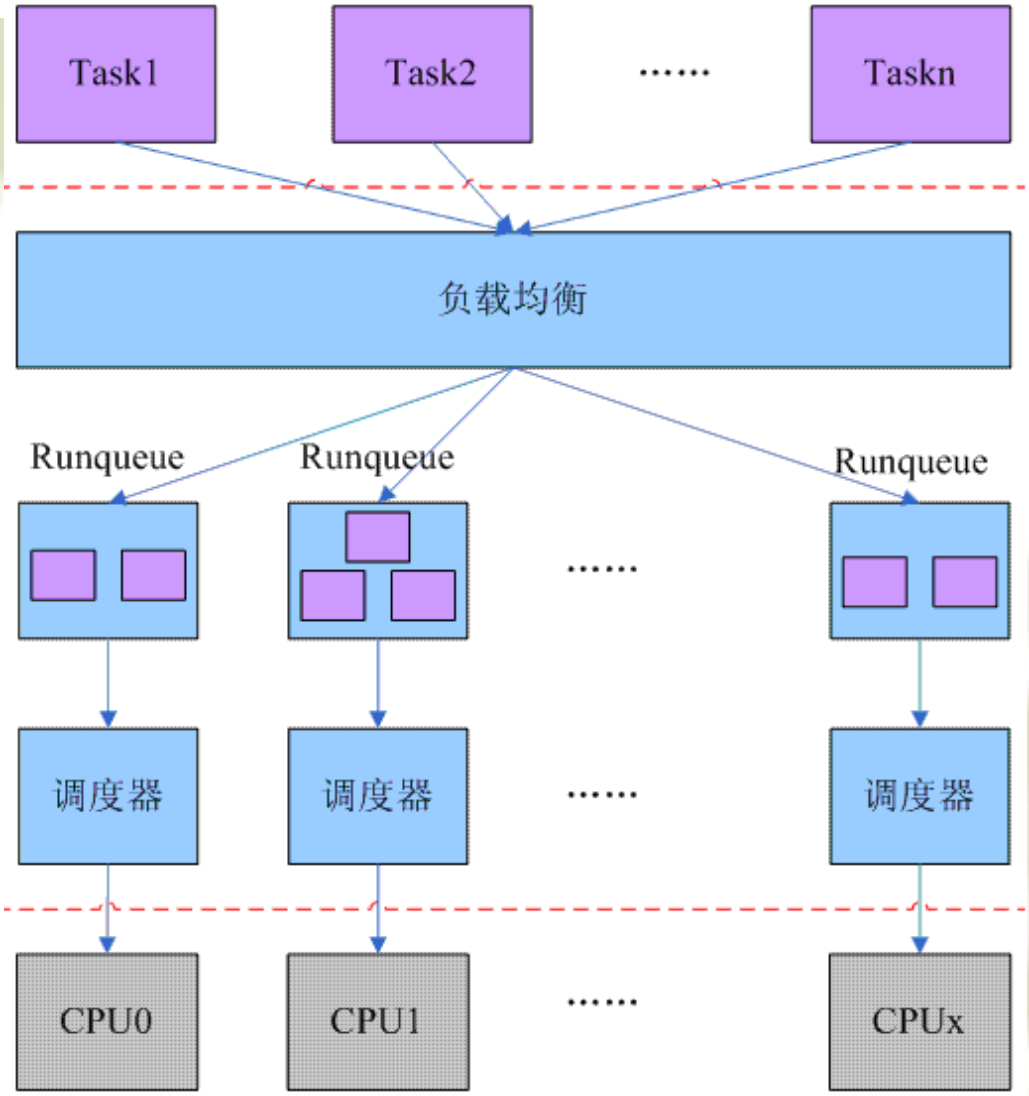
从内核空间：动态优先级（prio），静态优先级（static_prio），归一化优先级（normal_prio）和实时优先级（rt_priorit），这里特别介绍一下：

归一化优先级（normal_prio）：它是根据静态优先级、调度优先级和调度策略来计算得到的动态优先级（prio）：运行时可动态调整

在task_struct结构中的表示如下：

```
struct task_struct {
......
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
......
    unsigned int policy;
......
}
```

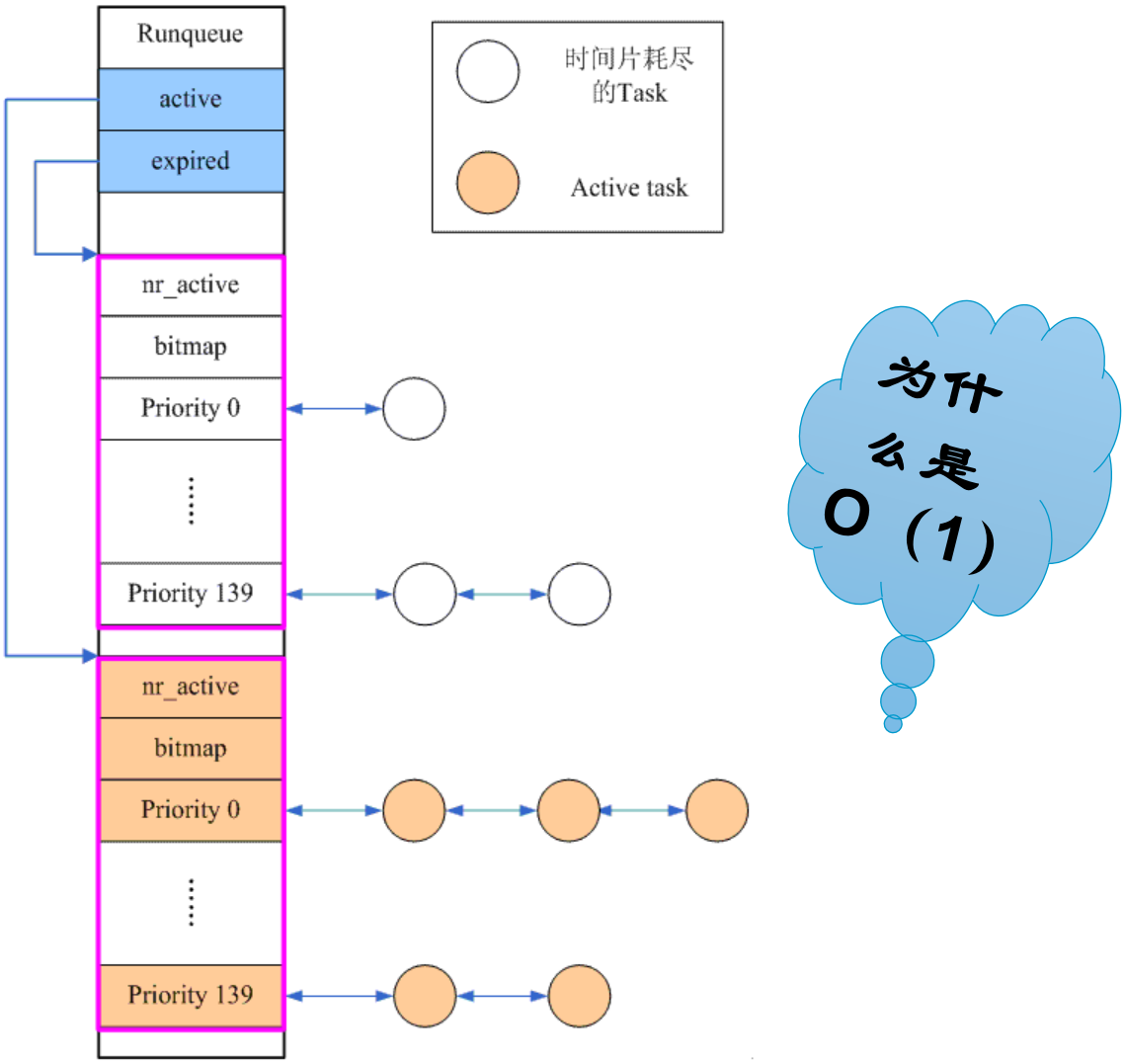
进程调度-O(1)调度


Task1
Task2
……
Taskn
负载均衡
Runqueue
Runqueue
Runqueue
……
调度器
调度器
……
调度器
CPU0
CPU1
……
CPUx

进程调度-O(1)调度

O（n）调度器中只有一个全局的就绪队列（runqueue），严重影响了扩展性，因此在O（1）调度器中引入了每CPU 一个就绪队列的概念。系统中所有的就绪进程首先经过负载均衡模块挂入各个CPU的就绪队列上，然后由主调度器和周期性调度器驱动该CPU上的调度行为。

O(1)调度的优先级队列


Runqueue
active
expired
nr_active
bitmap
Priority 0
Priority 139
nr_active
bitmap
Priority 0
Priority 139
时间片耗尽的Task
Active task
为什么是O(1)

进程调度-O(1)调度

O（1）调度器的基本优化思路就是把原来就绪队列上的单链表变成多个链表，即每一个优先级的进程被挂入不同链表中。

优先级数组（struct prio_array）结构的具体表示如下：

```
struct prio_array {
    unsigned int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};
```

O（1）由于支持140个优先级，因此队列成员中有140个分别表示各个优先级的链表头，不同优先级的进程挂入不同的链表中。bitmap是表示各个优先级进程链表是空还是非空。nr_active表示这个队列中有多少个任务。在这些队列中，100～139是普通进程的优先级，其他的是实时进程的优先级。因此，在O（1）调度器中，实时进程和普通进程被区分开了，普通进程根本不会影响实时进程的调度。

进程调度-O(1)调度

就绪队列中有两个优先级队列：（struct prio_array）分别用来管理活跃队列active（也就是时间片还有剩余）和expired（时间片耗尽）的进程。随着系统的运行，活跃（active）队列的任务一个个的耗尽其时间片，挂入到时间片耗尽（expired）的队列。当活跃（active）队列的任务为空的时候，两个队列互换，开始一轮新的调度过程。

虽然在O（1）调度器中任务组织的形式发生了变化，但是其核心思想仍然和O（n）调度器一致的，都是把CPU资源分成一个个的时间片，分配给每一个就绪的进程。进程用完其额度后被抢占，等待下一个调度周期的到来。

主调度器（就是schedule函数）的主要功能是从该CPU的就绪队列中找到一个最合适的进程调度执行。其基本的思路就是从活跃（active）优先级队列中寻找。首先在当前活跃active队列的位图bitmap中寻找第一个非空的进程链表，然后从该链表中找到的第一个节点就是最适合下一个调度执行的进程。由于没有遍历整个链表的操作，因此这个调度器的算法复杂度是一个常量，从而解决了O（n）算法复杂度的问题。

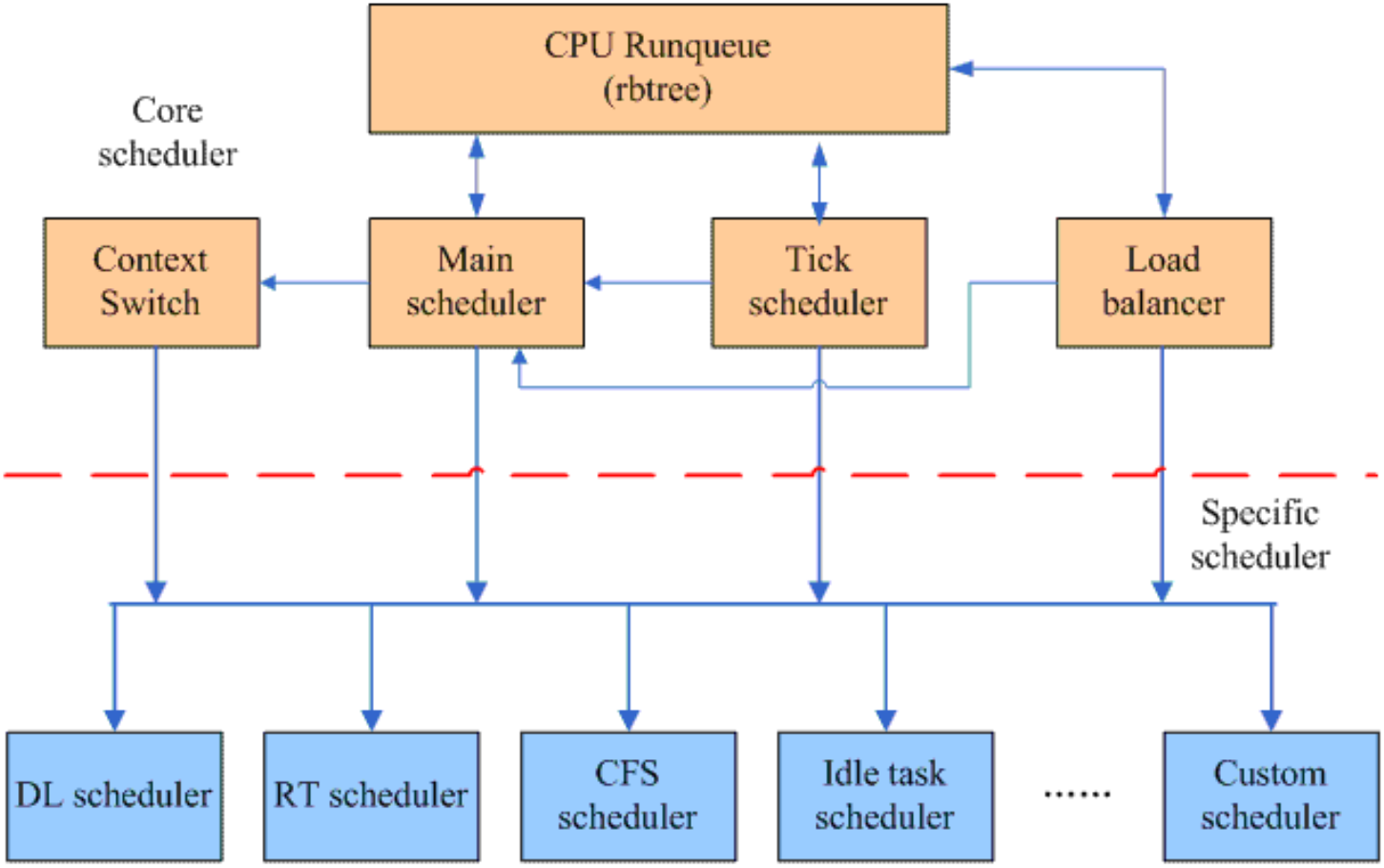
进程调度-O(1)调度

但是，O（1）调度器使用非常复杂的算法来判断进程是否是交互式进程以及进程的用户交互指数，即使这样，还会出现出现卡顿现象。如何解决？

能否不要被用户的具体需求捆绑，而又能支持灵活多变的需求。

再一次，我们想到了机制与策略分离机制。

调度模型-机制与策略分离



机制与策略分离-调度器类

这种机制功能层面上看，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就绪状态的任务根据负载情况平均分配到各个CPU 就绪队列上去。第二部分的功能是在各个CPU的主调度器（Main scheduler）和（周期性调度器）Tick scheduler的驱动下进行单个CPU上的调度。调度器处理的任务各不相同，有实时任务（ RT task ），普通任务（ normal task ），最后期限任务（Dead line task），但是无论哪一种任务，它们都有共同的）逻辑，这部分被抽象成核心调度器层（Core scheduler layer）

机制与策略分离-调度器类

同时各种特定类型的调度器定义自己的调度类（sched_class），并以链表的形式加入到系统中。这样的机制与策略分离的设计可以方便用户根据自己的场景定义特定的调度器，而不需要改动核心调度器层的逻辑。

先简单的介绍下struct sched_class部分成员作用。

机制与策略分离-调度器类

```
struct sched_class {
    const struct sched_class *next;
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);
    struct task_struct * (*pick_next_task)(struct rq *rq, struct task_struct
*prev, struct rq_flags *rf);
    /* ... */
};
```

机制与策略分离-调度器类

第一个：next：next指向下一个比自己低一个优先级调度类。

第二个字段：enqueue_task：指向入队的函数。

第三个字段：dequeue_task：指向出队的函数。

第四个：check_preempt_curr：当前cpu上正在运行的进程是否可被强占。

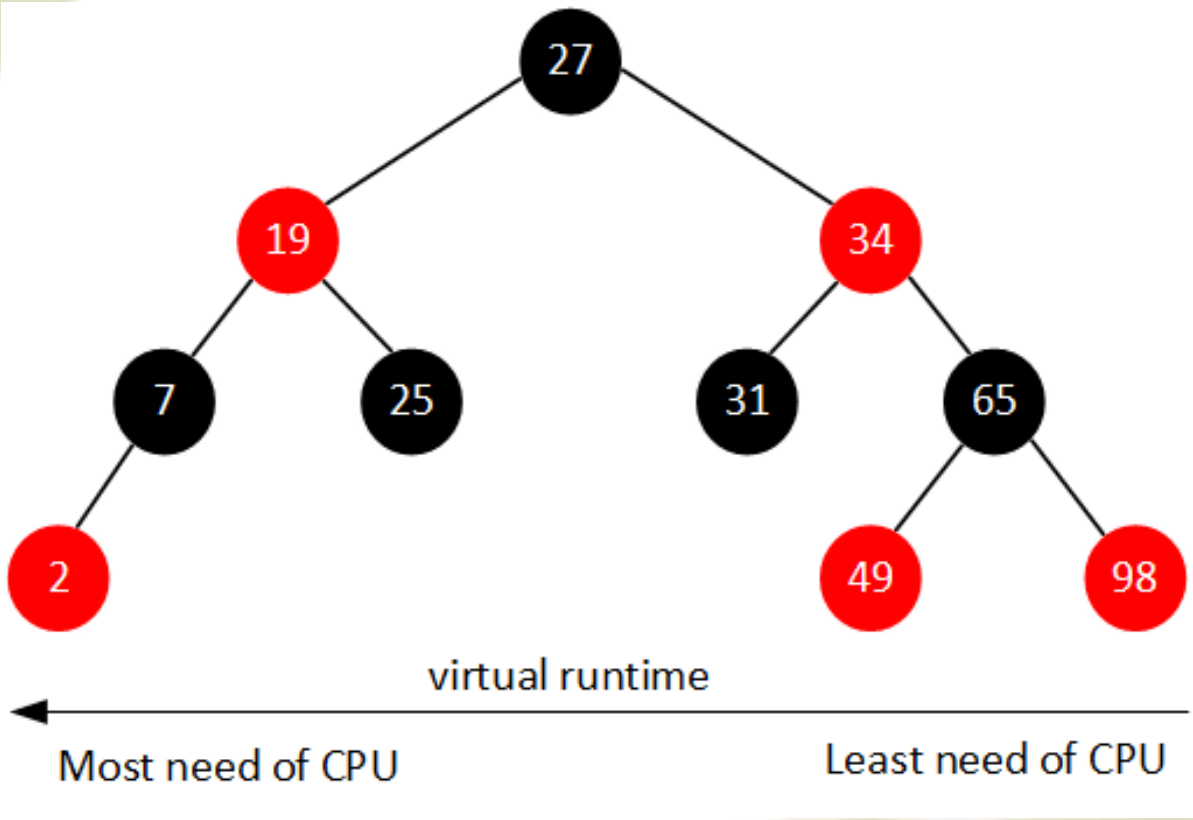
第五个：pick_next_task：从就绪队列中选择一个最适合运行的。这也算是调度器比较核心的一个操作。例如，我们依据什么挑选最适合运行的进程呢？这就是每一个调度器需要关注的问题。

完全公平调度-CFS

接下来，简述一下完全公平调度CFS。

CFS调度器的目标是保证每一个进程的完全公平调度。CFS调度器和以往的调度器不同之处在于没有时间片的概念，而是分配cpu使用时间的比例。理想状态下每个进程都能获得相同的时间片，并且同时运行在CPU上，但实际上一个CPU同一时刻运行的进程只能有一个。 也就是说，当一个进程占用CPU时，其他进程就必须等待。CFS为了实现公平，必须惩罚当前正在运行的进程，以使那些正在等待的进程下次被调度。

完全公平调度-红黑树

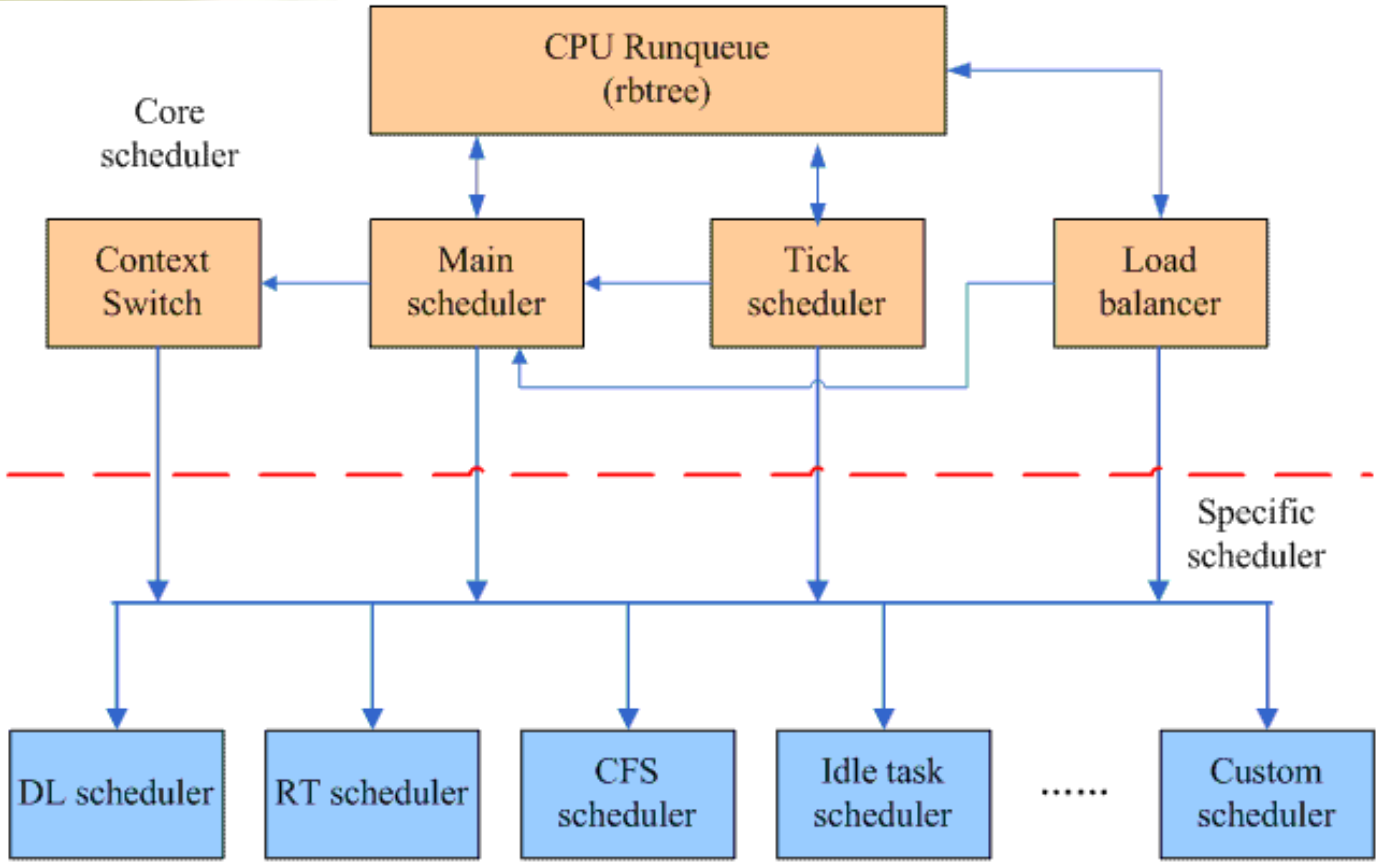

27
19
34
7
25
31
65
2
49
98
virtual runtime
Most need of CPU
Least need of CPU

完全公平调度-CFS

具体实现时，CFS通过每个进程的虚拟运行时间（vruntime）来衡量哪个进程最值得被调度。CFS中的就绪队列是一棵以虚拟时间为键值的红黑树（如图所示），虚拟时间越小的进程越靠近整个红黑树的最左端。因此，调度器每次选择位于红黑树最左端的那个进程，该进程的虚拟时间最小。

虚拟运行时间是通过进程的实际运行时间和进程的权重（weight）计算出来的。在CFS调度器中，将进程优先级这个概念弱化，而是强调进程的权重。一个进程的权重越大，则说明这个进程更需要运行，因此它的虚拟运行时间就越小，这样被调度的机会就越大。

Linux调度器小结


CPU Runqueue
(rbtree)
Core
scheduler
Context
Switch
Main
scheduler
Tick
scheduler
Load
balancer
Specific
scheduler
DL scheduler
RT scheduler
CFS
scheduler
Idle task
scheduler
.....
Custom
scheduler

Linux调度器小结

（针对上图讲）进程（或者叫任务）调度器是操作系统一个很重要的部件，它的主要功能就是把系统中的任务调度到各个CPU上去执行，以满足如下的性能需求：

1、对于分时的进程，调度器必须是公平的

2、快速的进程响应时间

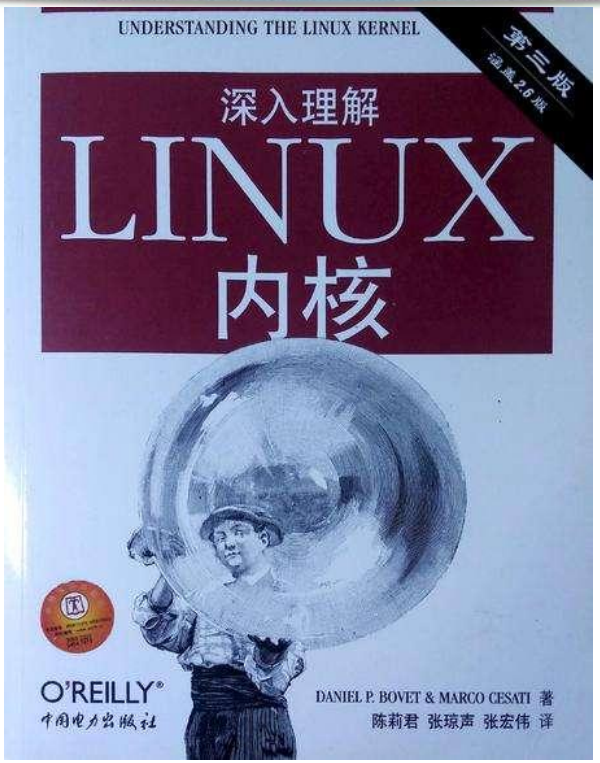
3、系统有高的吞吐量

4、功耗要小

当然，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。

为了达到这些目标，调度器必须综合考虑各种因素，进一步更详细的探讨请大家阅读相关参考书和源代码。

参考资料



深入理解Linux内核 第三版第七章

http://www.wowotech.net/ ，蜗窝科技网站关于进程调度，有一系列的文章，非常详细的介绍了调度的方方面面。

本章结束后，要求大家务必阅读调度的源代码，可以先看低版本的代码，从原理上入门，然后，按照兴趣或者需求看高版本的代码。

谢谢大家！

THANK YOU