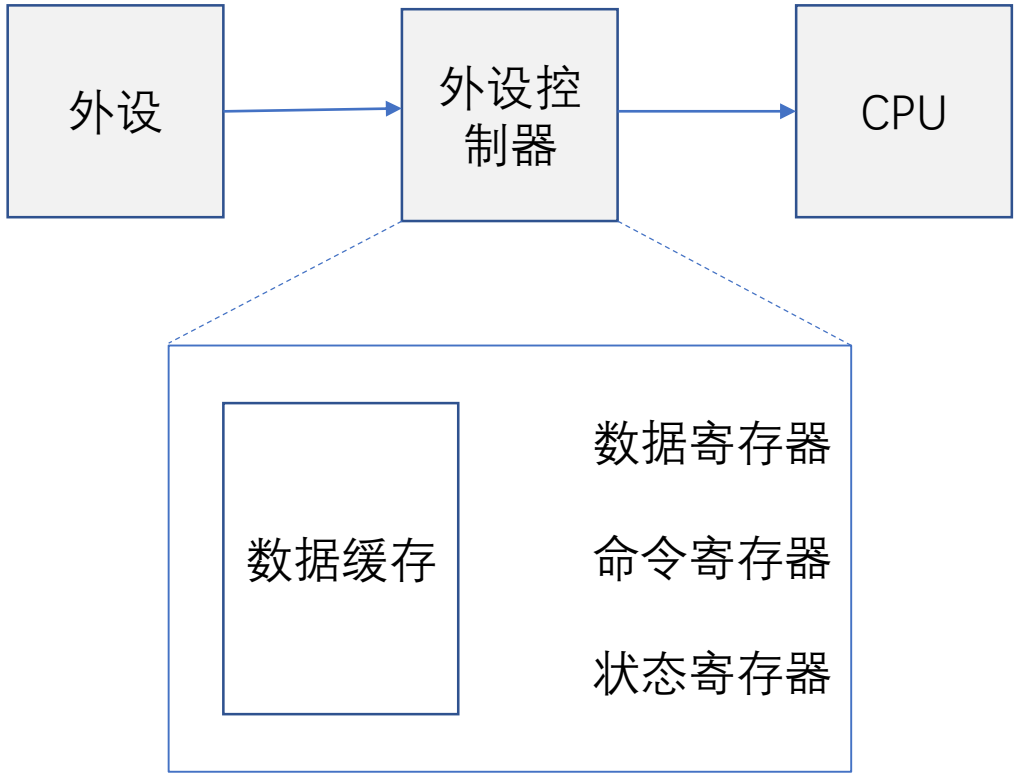
Linux IO系统架构

硬件架构

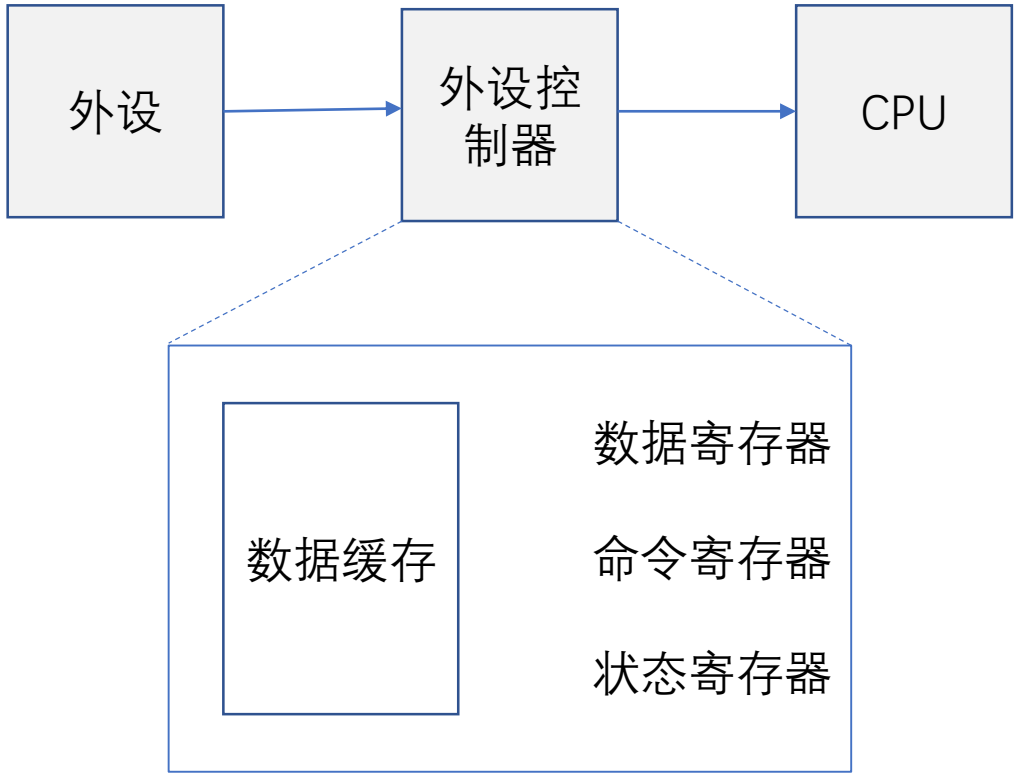
- 外设与计算机系统的一次交互过程
  - 外设控制器是外设与CPU的接口，将外设输入的信息转化为计算机能够处理的信息，以字节为单位存入数据缓存。供CPU读取。
  - 数据缓存中装有供CPU读取的新数据后，外设控制器会将状态寄存器中的ready比特置1，同时给出中断信号，提醒计算机系统读取新数据。
  - 计算机系统读取新数据后，CPU才可以向外设发送新的IO命令。



PS：外设只能够串行工作，上一条IO命令完成后，才能够执行下一条IO命令。

硬件架构

- 计算机系统借由外设控制器中的数据寄存器，将数据缓存中的数据读入内存，一次一个字。



- 外设工作在
  - 中断模式： 数据缓存中的数据是中断处理程序存入内存的。比如，键盘中断处理程序从数据寄存器中读取用户输入的字符，然后将其存入内存（终端输入缓存）。
  - DMA模式： 数据缓存中的数据是DMA存入内存的。之后，DMA控制器向CPU发命令，告诉CPU数据内存就绪。中断处理程序无需执行数据搬迁操作。网卡和磁盘工作在DMA模式。

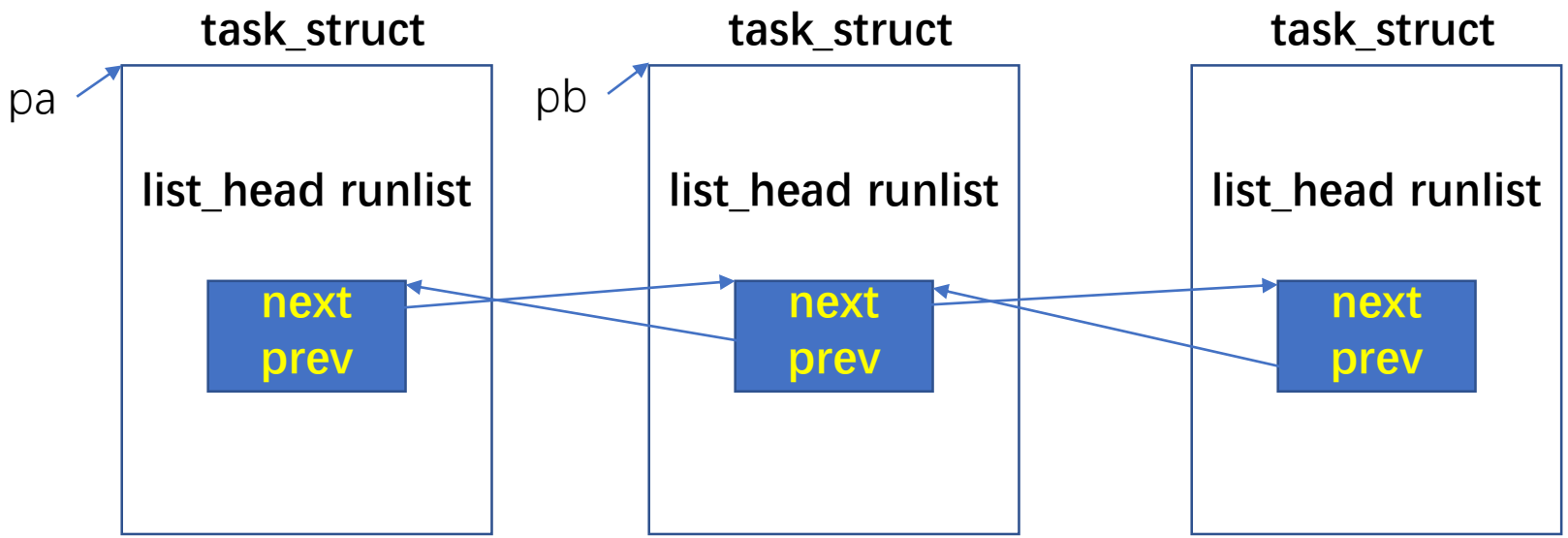
软件架构

- PA进程执行recv系统调用，读TCP链接
  - 执行recv入睡，等待在某个PORT上
  - 从内核缓存中读出来TCP包中的数据，存入自己的私有数据空间。
  - recv系统调用返回
- PA回用户态，执行应用程序处理内核送来的TCP包。

- 中断处理过程（执行者是被中断的现运行进程PB）
  - 中断上半段，确保 IO数据 进内存（在内核缓存）。
  - 中断下半段，将 IO数据处理成为应用程序 需要的数据。以网络接受为例：摘除报头，得TCP包。PORT号：定位阻塞在recv的进程PA，唤醒它。可以送给PA的数据在内核数据空间

Linux调度系统架构

list_head数据结构，以 task_struct 中的 runlist为例



task_struct * pa;
pa->runlist.next 指向 pb->runlist

求 pb 的值　　// list_entry( )做的事情

用task_struct中嵌入list_head数据成员的方式，可以形成进程双向队列。

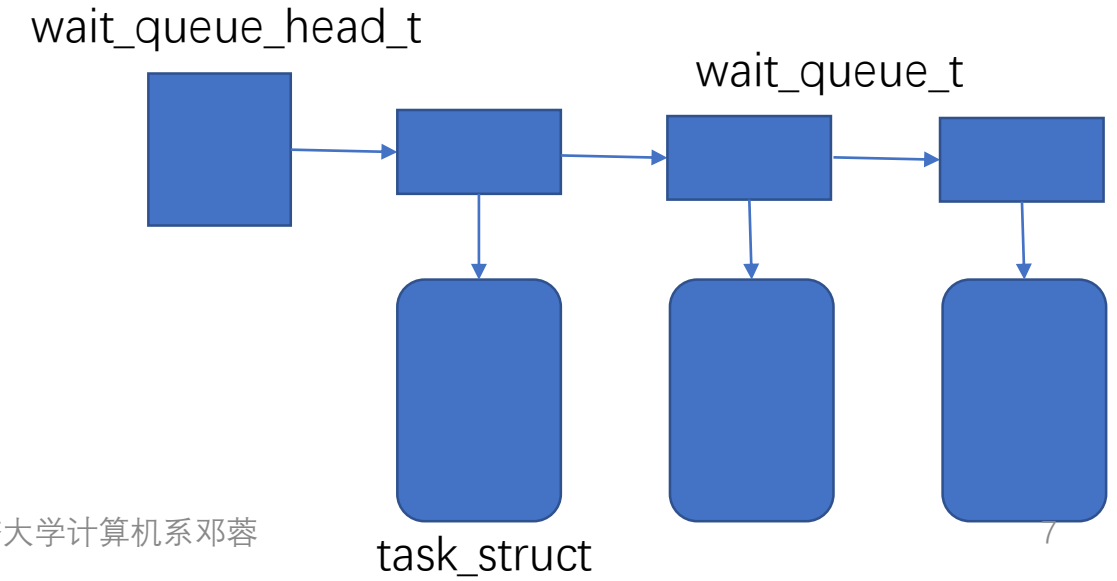
等待队列

等待队列由双向链表实现，其元素包括指向进程描述符的指针。每个等待队列都有一个等待队列头（*wait queue head*），等待队列头是一个类型为wait_queue_head_t的数据结构：

```
struct _ _wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct _ _wait_queue_head wait_queue_head_t;
```

等待队列链表中的元素类型为 wait_queue_t：

```
struct _ _wait_queue {
    unsigned int flags;
    struct task_struct * task;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct _ _wait_queue wait_queue_t;
```



现运行进程入睡

sleep_on()对当前进程进行操作:

```
void sleep_on(wait_queue_head_t *wq)
{
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);   // 为现运行进程做一个等待队列元素wait，挂上task_struct
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(wq,&wait);   // 把 wait 插入等待队列
    schedule();         // 放弃CPU
    remove_wait_queue(wq, &wait);    // 从等待队列中删除wait元素，del这个对象
}
```

唤醒进程

```
void wake_up(wait_queue_head_t *q)      // 唤醒 q 指向的 等待进程队列中的元素
{
    struct list_head *tmp;
    wait_queue_t *curr;

    list_for_each(tmp, &q->task_list) {
        curr = list_entry(tmp, wait_queue_t, task_list);
        if (curr->func(curr, TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
                       0, NULL) && curr->flags)
            break;                          // 如果队列中全是互斥的睡眠进程，只唤醒1个进程
    }                                       // 如果队列中全是非互斥的睡眠进程，全部唤醒
}                                           // 否则，唤醒队列中排在前面的所有非互斥进程
```

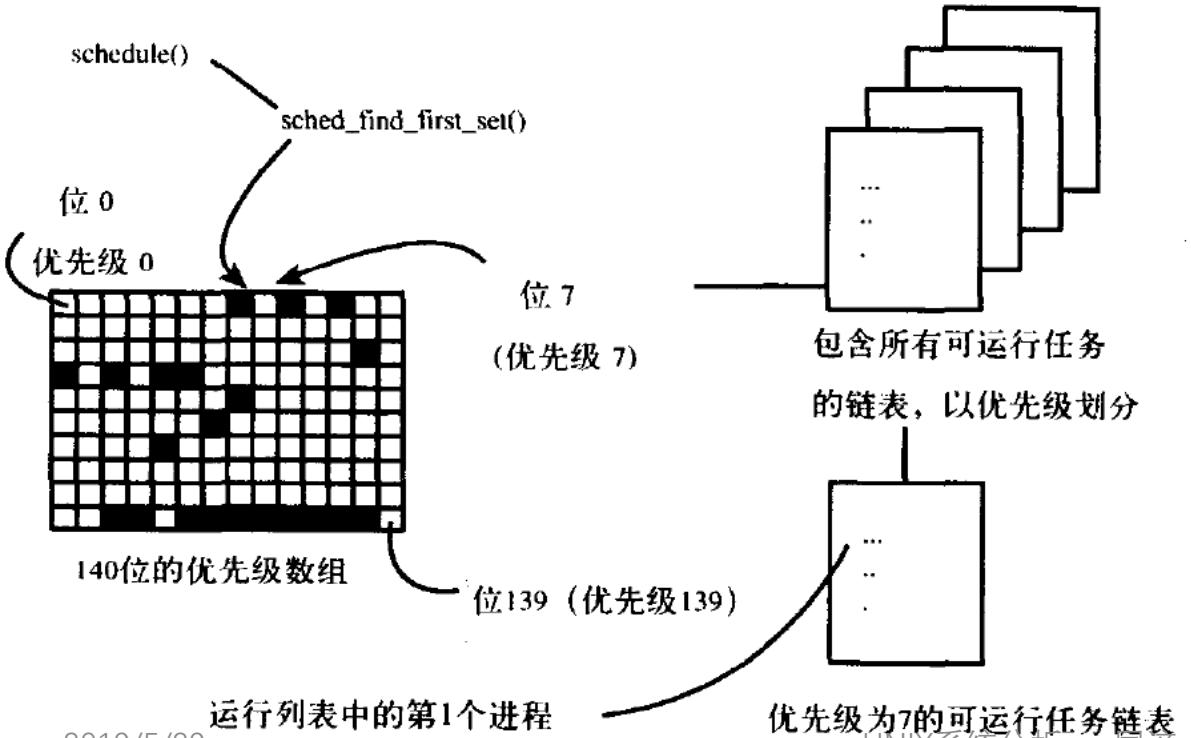
进程被唤醒后，插入就绪队列

就绪进程链表（序）　　数据结构

全局数组　prio_array_t

int	nr_active	链表中进程描述符的数量
unsigned long [5]	bitmap	优先权位图：当且仅当某个优先权的进程链表不为空时设置相应的位标志
struct list_head [140]	queue	140 个优先权队列的头结点

prio_array_t，可以让CPU在o(1)时间内找出最高优先级的进程



每个进程用task_struct 中的 runlist
链入就绪进程列表
进程优先数是几，就挂在下标是几的队列里

schedule在O(1)时间内取出最高优先级的就绪进程

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;

prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;        // 第 index 条就绪进程队列
next = list_entry(queue->next, struct task_struct, run_list);
                                   // 取队首进程
```

就绪进程链表（含所有TASK_RUNNING进程）又叫运行队列链表

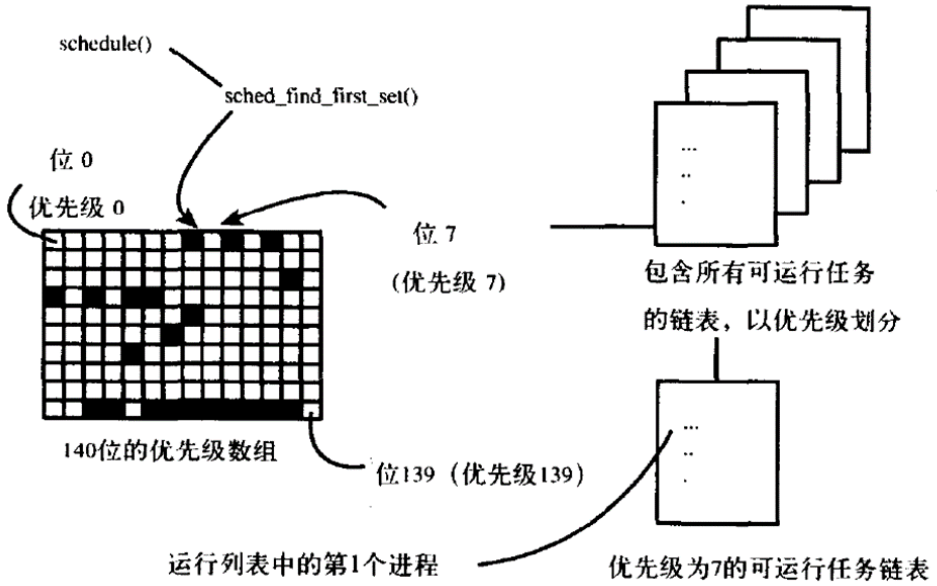
数据结构 runqueue（每个处理器有一个）

登记这颗CPU调度状态

```
struct runqueue {
spinlock_t              lock;                   /* 保护运行队列的自旋锁*/
unsigned long           nr_running;             /* 可运行任务数目 */
unsigned long           nr_switches;            /* 上下文切换数目*/
unsigned long           expired_timestamp;      /* 队列最后被换出时间 */
unsigned long           nr_uninterruptible;     /* 处于不可中断睡眠状态的任务数目*/
unsigned long long      timestamp_last_tick;    /* 最后一个调度程序的节拍 */
struct task_struct      *curr;                  /* 当前运行任务 */
struct task_struct      *idle;                  /* 该处理器的空任务*/
struct mm_struct        *prev_mm;               /* 最后运行任务的mm_struct结构体*/
struct prio_array       *active;                /* 活动优先级队列 */
struct prio_array       *expired;               /* 超时优先级队列 */
struct prio_array       arrays[2];              /* 实际优先级数组*/
struct task_struct      *migration_thread;      /* 移出线程 */
struct list_head        *migration_queue;       /* 移出队列*/
atomic_t                nr_iowait;              /* 等待I/O操作的任务数目 */
};
```

就绪进程队列的插入操作1（普通进程）

- 因为时间片，放弃CPU
- Fact：每颗CPU，有2个就绪进程队列。活动的（active）和 过期的（expired）。系统只会调度活动队列中的进程，不存在时，过期队列变活动队列。



- 每个进程有自己的时间片
  - TimeSlice，是进程在active队列中可以运行的总时间。用完，进程进过期队列，时间片重新充值。
  - Granularity，是进程可以连续运行的时间。用完，进程放弃CPU，去当前队列末尾。和队列中等优先级的其它进程Round Robin，共享CPU。

unsigned int time_slice;　　//时间片还剩多少

$$\text{base time quantum (in milliseconds)} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases} \quad (1)$$

表7_2

Description	Static priority	Nice value	Base time quantum
Highest static priority	100	-20	800 ms
High static priority	110	–10	600 ms
Default static priority	120	0	100 ms
Low static priority	130	+10	50 ms
Lowest static priority	139	+19	5 ms

时间片 和 静态优先级

int prio；

dynamic priority = max (100,
min (static_priority – bonus + 5, 139)) (2)

大的 bonus，会提升进程的动态优先级

表7_3

Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10

动态优先级 和 平均睡眠时间

交互性的判定：
dynamic priority <= 3 * static priority / 4 + 28 (3)
等价于 bonus - 5 >= static priority / 4 - 28

Interactive delta： static priority / 4 – 28
是一个阈值

Description	Static priority	Nice value	Base time quantum	Interactivedelta	Sleep time threshold
Highest static priority	100	-20	800 ms	-3	299 ms
High static priority	110	-10	600 ms	-1	499 ms
Default static priority	120	0	100 ms	+2	799 ms
Low static priority	130	+10	50 ms	+4	999 ms
Lowest static priority	139	+19	5 ms	+6	1199 ms

平均睡眠时间 和 交互性的判定

时钟中断处理程序 处理现运行进程的时间片

1、current->time_slice --

2、if(time_slice 是 0)

- current出列
- 重新计算prio
- current->time_slice = task_timeslice(current);
- 入列

```
if (! task->time_slice) {
        if (!TASK_INTERACTIVE(task) || EXPIRED_STARVING(rq))
                enqueue_task(task, rq->expired);
        else
                enqueue_task(task, rq->active);
}
```

- 置重新调度标记　TIF_NEED_RESCHED

3、if(time_slice 不是 0)

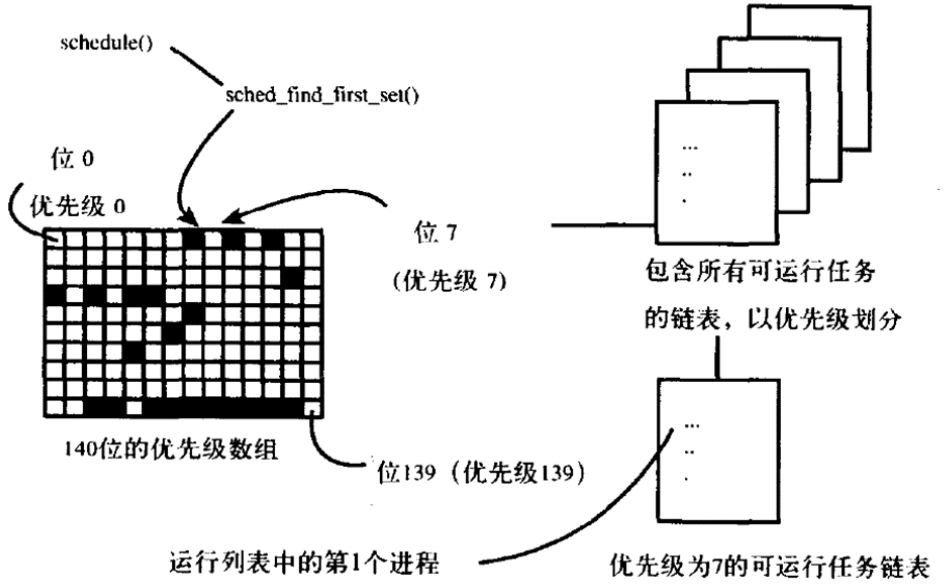
- if （ current是交互式进程 && time_slice%Granularity ==0 ~)

```
list_del(&current->run_list);
list_add_tail(&current->run_list,
        this_rq( )->active->queue+current->prio);
```

- 置重新调度标记　TIF_NEED_RESCHED

就绪进程队列的插入操作 2

- 唤醒进程时，计算它的优先级。插入这个优先级的就绪进程队列。尾部
  - 被中断处理程序唤醒的进程，插入
    - 响应中断的那个CPU队列
    - 执行系统调用的那个CPU队列
    - 除非，这两条队列就绪进程数量过多
- 如果进程因为响应中断被剥夺CPU，就绪队列位置不变，还在原队列首部



唤醒睡眠进程

1、锁住2条就绪队列：入睡时使用的 CPUa（在thread_info结构里） 和 中断处理程序使用的CPUb。
2、把进程挂在哪条就绪队列（CPU）？
- idle的　CPUa，CPUb，其它
- CPUa负载轻，CPUa
- 睡眠时间很短，CPUa
- CPUa移动至CPUb，有助于改善负载不均衡状态，CPUb

3、读当前时刻：now
4、重新计算优先级 prio
5、插在CPU的优先级是prio的就绪队列
6、标识需要重新调度
- 单处理器系统　置TIF_NEED_RESCHED标识
- 多处理器系统　如果目标处理器当前正在运行的进程优先级低，向它发中断信号，促其调度

7、将进程状态改为 TASK_RUNNING
8、解锁，返回

计算被唤醒的进程的优先权

1、sleep_time = now – p->timestamp // timestamp 是 进程入睡时的时间戳
2、交互式普通进程 & 从高优先级（uninterruptable）唤醒 & 睡了太久（阈值，看表7-2）
　　p->sleep_avg = 900； p->prio = effective_prio(p); return；
3、计算bonus（看表7-3 ） * sleep_time
4、sleep_time *= （10-bonus）
5、p->sleep_avg += sleep_time
6、p->prio = effective_prio(p); return；

4.2.5 计算优先级和时间片

在本章的开头，我们看到了如何利用优先级和时间片来影响调度程序做出决定。另外，我们还知道了I/O消耗型和处理器消耗型进程以及为什么提高I/O消耗型进程的优先级会有好处。现在，我们看实际的代码是如何实现这些设计的。

进程拥有一个初始的优先级，叫做nice值。该数值变化范围为−20到＋19，默认值为0。19优先级最低，−20最高。进程task_struct的static_prio域就存放着这个值。之所以起名为静态优先级(static priority)，是因为它从一开始由用户指定后，就不能改变。而调度程序要用到的动态优先级存放在prio域里。动态优先级通过一个关于静态优先级和进程交互性的函数关系计算而来。

effective_prio()函数可以返回一个进程的动态优先级。这个函数以nice值为基数，再加上−5到+5之间的进程交互性的奖励或罚分。举例来说，一个交互性很强的进程，即使它的nice值为10，它的动态优先级最终也有可能达到5。相反，一个温和的处理器吞噬者，虽然本来nice值一样是10，它最后的动态优先级却可能是12。交互性不强不弱的进程——位于I/O消耗型进程与处理器消耗型进程之间——不会得到优先级的奖励，同样也不会被罚分，所以它的动态优先级和它的nice值相等。

当然，调度程序不可能通过魔法来了解一个进程的交互性到底强不强。它必需通过一些推断来获取准确反映进程到底是I/O消耗型的还是处理器消耗型的。最明显的标准莫过于进程休眠的时间长短了。如果一个进程的大部分时间都在休眠，那么它就是I/O消耗型的。如果一个进程执行的时间比休眠的时间长，那它就是处理器消耗型的。这个标准可以向极端延伸：一个进程如果几乎所有的时间都用在休眠上，那么它就是一个纯粹的I/O消耗型进程，相反，一个进程如果几乎所有的时间都在执行，那么它就是纯粹的处理器消耗型进程。

为了支持这种推断机制，Linux记录了一个进程用于休眠和用于执行的时间。该值存放在task_struct的sleep_avg域中。它的范围从0到MAX_SLEEP_AVG。它的默认值为10毫秒。当一个进程从休眠状态恢复到执行状态时，sleep_avg会根据它休眠时间的长短而增长，直到达到MAX_SLEEP_AVG为止。相反，进程每运行一个时钟节拍，sleep_avg就做相应的递减，到0为止。

这种推断准确得让人吃惊。它的计算不仅仅基于休眠时间有多长，而且运行时间的长短也要被计算进去。所以，尽管一个进程休眠了不少时间，但它如果总是把自己的时间片用得一干二净，那么它就不会得到大额的优先级奖励——这种推断机制不仅会奖励交互性强的进程，它还会惩罚处理器耗费量大的进程，并且不会滥用这些奖惩手段。如果一个进程发生了变化，开始大量占用处理器时间，那么，它很快就会失去曾经得到的优先级提升。最后要说的是，这种衡量标准还有很快的反应速度。一个新创建的交互性进程的sleep_avg很快就会涨得很高。即便如此，由于奖励和罚分都加在作为基数的nice值上，所以用户还是可以通过改变进程的nice值来对调度程序施加影响。

另一方面，重新计算时间片相对简单了。它只要以静态优先级为基础就可以了。在一个进程创建的时候，新建的子进程和父进程均分父进程剩余的进程时间片。这样的分配很公平并且防止用户通过不断创建新进程来不停地攫取时间片。然而，当一个任务的时间片用完之后，就要根据任务的静态优先级重新计算时间片。task_timeslice()函数为给定任务返回一个新的时间片。时间片的计算只需要把优先级按比例缩放，使其符合时间片的数值范围要求就可以了。进程的优先级越高，它每次执行得到的时间片就越长。优先级最高的进程（其nice值是−20）能获得的最大时间片长度（MAX_TIMESLICE）是800毫秒。而优先级最低的进程（其nice值是+19）获得的最短时间片长度（MIN_TIMESLICE）为5毫秒或一个时钟滴答（参见第10章）。默认优先级（nice值为0）的进程得到的时间片长度为100毫秒，参见表4-1。

表4-1　调度程序时间片

进程类型	nice值	时间片长度
初始创建的进程	父进程的值	父进程的一半
优先级最低的进程	+19	5毫秒（MIN_TIMESLICE）
默认优先级的进程	0	100毫秒（DEF_TIMESLICE）
优先级最高的进程	−20	800毫秒（MAX_TIMESLICE）

调度程序还提供了另外一种机制以支持交互进程：如果一个进程的交互性非常强，那么当它时间片用完后，它会被再放置到活动数组而不是过期数组中。回忆一下，重新计算时间片是通过活动数组与过期数组之间的切换来进行的。一般进程在用尽它们的时间片后，都会被移至过期数组，当活动数组中没有剩余进程的时候，这两个数组就会被交换；活动数组变成过期数组，过期数组替代活动数组。这种操作提供了时间复杂度为O(1)的时间片重新计算。但在这种交换发生之前，交互性很强的一个进程有可能已经处于过期数组中了，当它需要交互的时候，它却无法执行，因为必须等到数组交换发生为止才可执行。将交互式的进程重新插入到活动数组可以避免这种问题。但该进程不会被立即执行，它会和优先级相同的进程轮流着被调度和执行。该逻辑在scheduler_tick()中实现，该函数会被定时器中断调用（在第10章中讨论）：

```
struct task_struct *task;
struct runqueue *rq;

task = current;
rq = this_rq();

if (!--task->time_slice) {
        if (!TASK_INTERACTIVE(task) || EXPIRED_STARVING(rq))
                enqueue_task(task, rq->expired);
        else
                enqueue_task(task, rq->active);
}
```

首先，这段代码减小进程时间片的值，再看它是否为0。如果是的话就说明进程的时间片已经耗尽，需要把它插入到一个数组中，所以该代码先通过TASK_INTERACTIVE()宏来查看这个进程是不是交互型的进程。该宏主要基于进程的nice值来判定它是不是一个“交互性十足”的进程。nice值越小（优先级越高），越能说明该进程交互性越高。一个nice值为19的进程交互性表现得再强，它也不可能被重新加入到活动数组中去。而一个nice值为－20的进程要想不被重新加入，只有拼命的占用处理器才行。一个拥有默认优先级的进程，需要表现出一定的交互性才能被重新加入，但这通常很容易就能满足。接着，EXPIRED_STARVING()宏负责检查过期数组内的进程是否处于饥饿状态——是否已经有相对较长的时间没有发生数组切换了。如果最近一直没有发生切换，那么再把当前的进程放置到活动数组会进一步拖延切换—过期数组内的进程会越来越饿。只要不发生这种情况，进程就会被重新放置在活动数组里。否则，进程会被放入过期数组里，这也是最普通的一种操作。

调度时机　不可剥夺的内核 V6

- 中断返回用户态
- if（中断前是用户态 & runrun标识是1）

  swtch（）　　// 放弃CPU，挑优先级最高的进程运行
- 存在的问题
  - 现运行进程PA正在执行系统调用。
    - 被中断。中断处理程序唤醒等待IO操作结束的PB进程何时能够运行?
    - 时间片到了，PA会立即放弃CPU?

调度时机1　可剥夺的Linux内核

- 中断返回

- if（还有上半段没跑完 || 还有下半段没跑完 || 内核运行在临界区
  || TIF_NEED_RESCHED == 0 ）
  恢复现场、IRET;

  else
  schedule( );

表4-10：preempt_count的字段

位	描述
0～7	抢占计数器(max value = 255)
8～15	软中断计数器 (max value = 255)
16～27	硬中断计数器 (max value = 4096)
28	PREEMPT_ACTIVE 标志

调度时机2　可剥夺的Linux内核

- 进程进入临界区时，有时需要执行preempt_disable( )**禁剥夺**，保证临界区是原子操作。
  - preempt_count ++

- 进程退出临界区时，执行 preempt_enable（）。

1. preempt_count --;
2. if（ TIF_NEED_RESCHED != 0)
   preempt_schedule( );

大概率，if条件不成立，进程退出临界区，执行临界区外的代码

表4-10：preempt_count的字段

位	描述
0～7	抢占计数器(max value = 255)
8～15	软中断计数器 (max value = 255)
16～27	硬中断计数器 (max value = 4096)
28	PREEMPT_ACTIVE 标志

- preempt_schedule( )

```
if( current->thread->preempt_count == 0 )
    schedule( );
```

中断返回

1、现运行进程需要让出CPU嘛？
2、让出CPU，安全嘛？（现运行进程可以被抢占嘛）