《操作系统原理》实验报告

实验名称	Linux 操作系统实例研究报告	实验序号	6
实验日期	2023/5/11	实验人	盖乐

一、实验题目

阅读教材第 18 章（Linux 案例），并在互联网上查阅相关资料，对照操作系统课程中所讲的原理（进程管理，存储管理，文件系统，设备管理），了解 Linux 操作系统实例

形成一份专题报告

可以是全面综述性报告

可以是侧重某一方面的报告（进程调度，进程间通信，存储管理，文件系统，安全）

二、相关原理与知识

2.1 进程基础知识

a. 进程的定义

进程是资源分配的最小单位，线程是 CPU 调度的最小单位

进程[1]（process），是指计算机中已执行的程序。进程曾经是分时系统的基本运作单位。在面向进程设计的系统（如早期的 UNIX, Linux 2.4 及更早的版本）中，进程是程序的基本执行实体;在面向线程设计的系统（如当代多数操作系统、Linux 2.6 及更新的版本）中，进程本身不是基本执行单位，而是线程的容器。

程序本身只是指令、数据及其组织形式的描述，相当于一个名词，进程才是程序（那些指令和数据）的真正执行实例，可以想像说是现在进行式。若干进程有可能与同一个程序相关系，且每个进程皆可以同步（循序）或异步（平行）的方式独立执行。现代计算机系统可在同一段时间内以进程的形式将多个程序加载到内存中，并借由时间共享（或称分时复用），以在一个处理器上表现出同时（平

行性）执行的感觉。同样的，使用多线程技术的操作系统或计算机体系结构，同样程序的平行线程，可在多 CPU 主机或网络上真正同时执行（在不同的 CPU 上）。进程不仅包括可执行程序的代码段，还包括一系列的资源，比如：打开的文件、内存、CPU 时间、信号量、多个执行线程流等等。而线程可以共享进程内的资源空间。

进程不仅包括可执行程序的代码段，还包括一系列的资源，比如:打开的文件、内存、CPU 时间、信号量、多个执行线程流等等。而线程可以共享进程内的资源空间。

b. 进程标识符（PID）

PID：Process Identifier, also known as process ID or PID, is a unique number to identify each process running in an operating system such as Linux, Windows, and Unix.

Unix 系统通过 pid 来标识进程， linux 把不同的 pid 与系统中每个进程或轻量级线程关联，而 unix 程序员希望同一组线程具有共同的 pid ，遵照这个标准 linux 引入线程组的概念。一个线程组所有线程与领头线程具有相同的 pid ，存入 tgid 字段， getpid（） 返回当前进程的 tgid 值而不是 pid 的值。

在 CONFIG_BASE_SMALL 配置为 0 的情况下，PID 的取值范围是 0 到 32767，即系统中的进程数最大为 32768 个。

c. **进程状态**

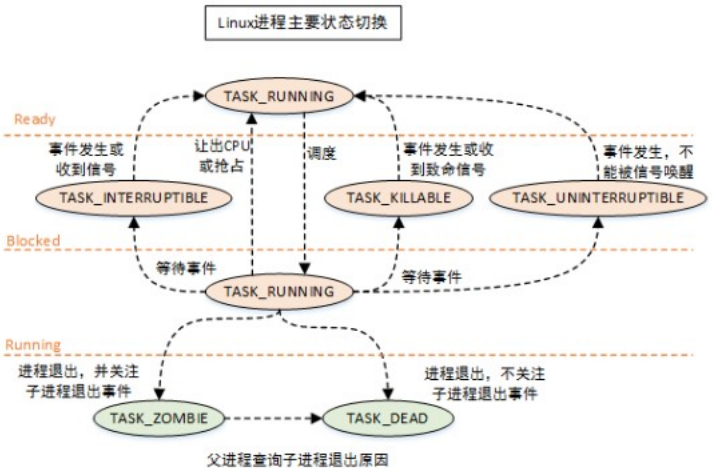
内核中主要定义的状态字段如下[3]：

```
1.  /*
2.   * Task state bitmask. NOTE! These bits are also
3.   * encoded in fs/proc/array.c: get_task_state().
4.   *
5.   * We have two separate sets of flags: task->state
```

```
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define __TASK_STOPPED 4
#define __TASK_TRACED 8
/* in tsk->exit_state */
#define EXIT_DEAD 16
#define EXIT_ZOMBIE 32
#define EXIT_TRACE (EXIT_ZOMBIE | EXIT_DEAD)
/* in tsk->state again */
#define TASK_DEAD 64
#define TASK_WAKEKILL 128 /** wake on signals that are deadly
**/
#define TASK_WAKING 256
#define TASK_PARKED 512
#define TASK_NOLOAD 1024
#define TASK_STATE_MAX 2048
/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED (TASK_WAKEKILL | __TASK_TRACED)
```

d. 进程状态图

系统中的每个进程都必然处于所列进程状态中的一种。



状态	描述
TASK_RUNNING	表示进程要么正在执行，要么正要准备执行（已经就绪），正在等待cpu时间片的调度
TASK_INTERRUPTIBLE	进程因为等待一些条件而被挂起（阻塞）而所处的状态。这些条件主要包括：硬中断、资源、一些信号……，一旦等待的条件成立，进程就会从该状态（阻塞）迅速转化成为就绪状态TASK_RUNNING
TASK_UNINTERRUPTIBLE	意义与TASK_INTERRUPTIBLE类似，除了不能通过接受一个信号来唤醒以外，对于处于TASK_UNINTERRUPIBLE状态的进程，哪怕我们传递一个信号或者有一个外部中断都不能唤醒他们。只有它所等待的资源可用的时候，他才会被唤醒。这个标志很少用，但是并不代表没有任何用处，其实他的作用非常大，特别是对于驱动刺探相关的硬件过程很重要，这个刺探过程不能被一些其他的东西给中断，否则就会让进城进入不可预测的状态
TASK_STOPPED	进程被停止执行，当进程接收到SIGSTOP、SIGTTIN、SIGTSTP或者SIGTTOU信号之后就会进入该状态
TASK_TRACED	表示进程被debugger等进程监视，进程执行被调试程序所停止，当一个进程被另外的进程所监视，每一个信号都会让进城进入该状态

e. 调度优先级

字段	描述
prio	用于保存动态优先级
static_prio	用于保存静态优先级，可以通过nice系统调用来进行修改
rt_priority	用于保存实时优先级
normal_prio	值取决于静态优先级和调度策略

实时优先级范围是 0 到 MAX_ RT_ PRIO-1 (即 99) ，而普通进程的静态优先级范围是从 MAX_ RT_ PRIO 到 MAX_ PRIO-1 (即 100 到 139) ， 值越大静态优先级越低，接下来将会对这些内容进行详细的介绍。

f. 调度策略相关字段

```
1. unsigned int policy;
2. const struct sched_class *sched_class;
3. struct sched_entity se;
4. struct sched_rt_entity rt;
5. cpumask_t cpus_allowed;
```

字段	描述
policy	调度策略
sched_class	调度类
se	普通进程的调用实体，每个进程都有其中之一的实体
rt	实时进程的调用实体，每个进程都有其中之一的实体
cpus_allowed	用于控制进程可以在哪里处理器上运行

g. 进程的分类

• I/O 密集型(IO-bound)

在进程占用 CPU 期间频繁有 IO 操作，出现 IO 阻塞等待情况，比如负责监听 socket 的进程，真正使用 CPU 进行计算的时间并不多。

• CPU 密集型(CPU-bound)

在进程占用 CPU 期间基本都在进行计算，很少进行 IO 操作，期间对 CPU 的真实使用率很高。

2.1 调度相关知识

a. scheduler 调度器

调度，就是按照某种调度的算法，从进程的就绪队列中选取进程分配 CPU，

主要是协调对 CPU 等的资源使用。进程调度的目标是最大限度利用 CPU 时间。

内核默认提供 5 类调度器，Linux 内核使用 struct sched_ class 对调度器进行抽象，一共分为以下 5 类:

• Stop 调度器，stop_sched_class}: 优先级最高的调度类，可以抢占其他所有进程，不能被其他进程抢占;

• Deadline 调度器， dl_sched_class): 使用红黑树，把进程按照绝对截止期限进行排序，选择最小进程进行调度运行;

• RT 调度器， rt_sched_class}: 实时调度器，为每个优先级维护一个队列;

• CFS 调度器， cfs_sched_class): 完全公平调度器，采用完全公平调度算法，引入虚拟运行时间概念;

• IDLE-Task 调度器，idle_sched_class: 空闲调度器，每个 CPU 都会有一个 idle 线程，当没有其他进程可以调度时，调度运行 idle 线程;

Linux 内核提供了一些调度策略供用户程序来选择调度器，其中 Stop 调度器和 IDLE-Task 调度器，仅由内核使用，用户无法进行选择:

•SCHED_ DEADLINE: 限期进程调度策略，使 task 选择 Deadline 调度器来调度运行;

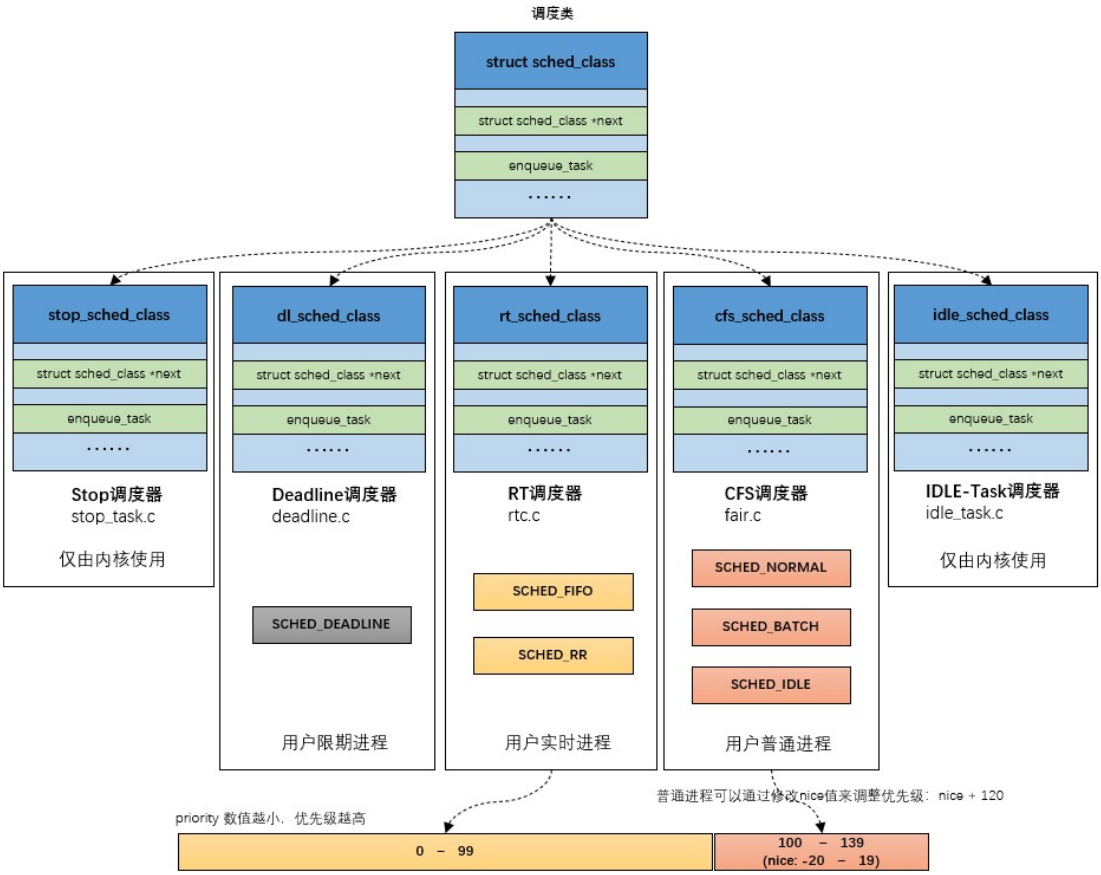
• SCHED_RR: 实时进程调度策略，时间片轮转，进程用完时间片后加入优先级对应运行队列的尾部，把 CPU 让给同优先级的其他进程;

• SCHED_FIFO:实时进程调度策略，先进先出调度没有时间片，没有更高优先级的情况下，只能等待主动让出 CPU;

• SCHED_NORMAL: 普通进程调度策略，使 task 选择 CFS 调度器来调度运行;

• SCHED_BATCH: 普通进程调度策略，批量处理，使 task 选择 CFS 调度器来调度运行;

• SCHED_IDLE:普通进程调度策略，使 task 以最低优先级选择 CFS 调度器来调度运行;



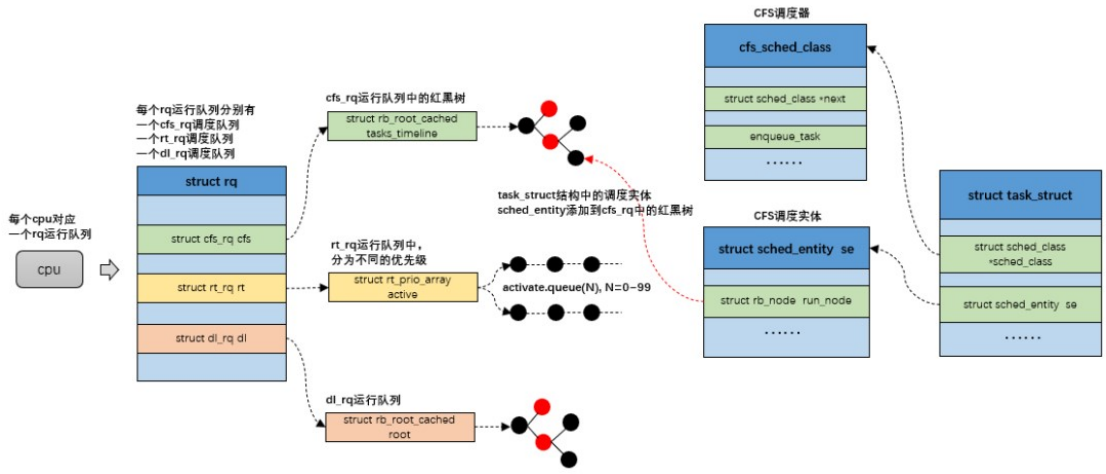
b. runqueue 运行队列

runqueue 运行队列是本 CPU 上所有可运行进程的队列集合。每个 CPU 都有一个运行队列，每个运行队列中有三个调度队列，task 作为调度实体加入到各自的调度队列中。

• struct cfs_rq cfs: CFS 调度队列

• struct rt_rq rt: RT 调度队列

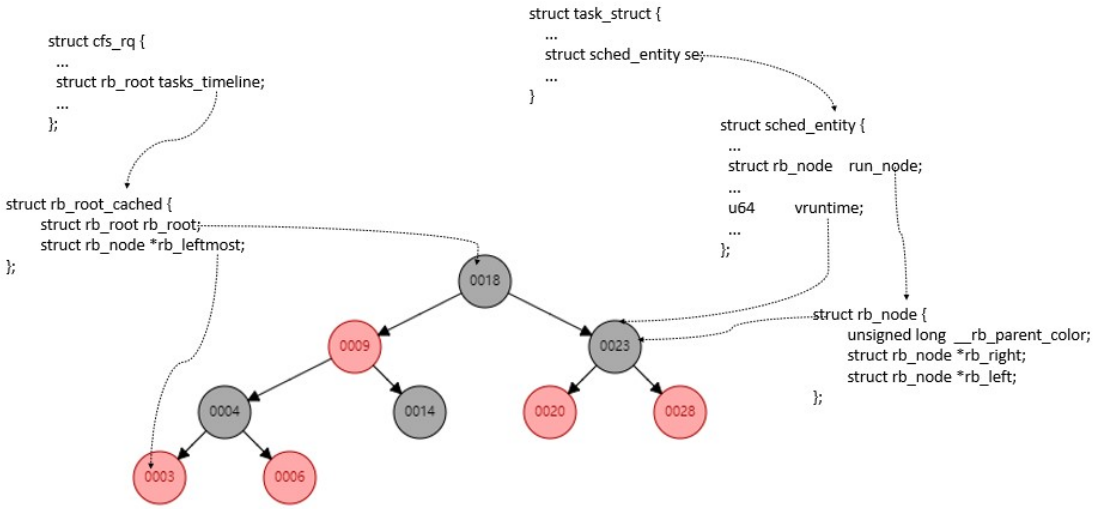
• struct dl_rq dl: DL 调度队列



cfs_rq:跟踪就绪队列信息以及管理就绪态调度实体，并维护一棵按照虚拟时间排序的红黑树。tasks_ timeline->rb_toot 是红黑树的根，tasks_timeline->rb_leftmost 指向红黑树中最左边的调度实体，即虚拟时间最小的调度实体。

sched_entity:可被内核调度的实体。每个就绪态的调度实体 sched_entity 包含插入红黑树中使用的节点 rb_node，同时 vruntime 成员记录已经运行的虚拟时间。

这些数据结构的关系如下图所示:



c. **调度算法**

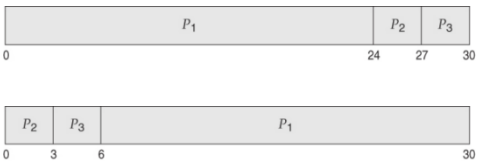
• 先来先服务 FCFS

先来先服务 FirstComeFirstService 可以说是最早最简单的调度算法，哪个进程先来就先让它占用 CPU,释放 CPU 之后第二个进程再使用，依次类推。

在下面的第一个甘特图中，进程 P1 先到达。三个进程的平均等待时间为 (0 + 24 + 27) / 3 = 17.0 ms

在下面的第二个甘特图中，相同的三个进程的平均等待时间为 (0 + 3 + 6) / 3 = 3.0 ms

三个突发的总运行时间相同，但在第二种情况下，三个中的两个完成得更快，而另一个进程稍有延迟。



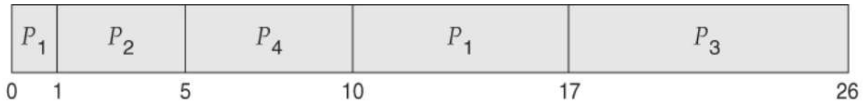
• 最短任务优先 SJF

最短任务优先 Shottest Job First 的思想是当多个进程同时出现时，优先安排执行时间最短的任务，然后是执行时间次短，以此类推。

SJF 可以解决 FCFS 中同时到达进程执行时间不一致带来的护航效应问题.

• 抢占式最短任务优先 PSJF

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5



SJF 算法最具启发的地方在于优先调度执行时间短的任务来解决护航效应，但是该算法属于非抢占式调度，对于先后到达且执行时间差别较大的场景也束手无策。

当向 SJF 算法增加抢占调度时能力就大大增强了，这就是 PSJF(Preemptive Shortest Job First)算法。

PSJF 算法对于进程 A 来说却不友好，进程 A 在被抢占之后只能等待 B 和 C 运行完成后，此时如果再来短任务 DEF 都会抢占 A，就会产生饥饿效应。

PSJF 算法是基于对任务运行时间长短来进行调度的，但在调度前任务的运行时间是未知的，因此首要问题是通过历史数据来预测运行时间。

• 时间片轮转 RR

时间片轮转 RR(Round-Robin)将 CPU 执行时间按照时钟脉冲进行切割称为时间切片 Time-Slice，一个时间切片是时钟周期的数倍，时钟周期和 CPU 的主频呈倒数关系。

比如 CPU 的主频是 1000hz，则时钟周期 TimeClick=1ms，Time Slice = n*Time Click，时间切片可以是 2ms/4ms/10ms 等。

在一个时间片内 CPU 分配给一个进程，时间片耗尽则调度选择下一个进程，如此循环。

进程往往需要多个循环获取多次时间片才能完成任务，因此需要操作系统记录上一次的数据，等进程下一次被分配时间片时再拿出来，这就是传说中的上下文

Contextc。

进程上下文被存储和拿出的过程被称为上下文切换 ContextSwitch，上下文切换是比较消耗资源的，因此时间片的划分就显得很重要:

• 时间片太短，上下文频繁切换，有效执行时间变少。

• 时间片太大，无法保证多个进程可以雨露均沾，响应时间较大。

RR 算法在保障了多个进程都能占用 CPU,属于公平策略的一种，但是 RR 算法将原本可以一次运行完的任务切分成多个部分来完成，从而拉升了周转时间。

• 多级反馈队列 MLFQ

多级反馈队列(Multi-Level Feedback Queue)尝试去同时解决响应时间和周转时间两个问题，具体做法:

•设置了多个任务队列，每个队列对应的优先级不同，队列内部的优先级相同。

• 优先分配 CPU 给高优先级的任务，同优先级队列中的任务采用 RR 轮询机制。

• 通过对任务运行状态的追踪来调整优先级，也就是所谓的 Feedback 反馈机制。

• 任务在运行期间有较多 IO 请求和等待，预测为交互进程，优先级保持或提升。

任务在运行期间一直进行 CPU 计算，预测为非交互进程，优先级保持或下降。

• 最初将所以任务都设置为高优先级队列，随着后续的运行状态再进行调整。

• 运行期间有 IO 操作则保持优先级。

运行期间无 IO 操作则把任务放到低一级的队列中。

• 不同队列分配不同的时间片。

• 高优先级队列往往是 IO 型任务，配置较小的时间片来保障响应时间。

低优先级队列往往是 CPU 型任务，配置较长时间片来保障任务一直运行。

在实际应用中仍然会有一些问题:

1)饥饿问题

• CPU 密集型的任务随着时间推移优先级会越来越低，在 IO 型进程多的场景下很容易出现饥饿问题，一直无法得到调度。

任务是 CPU 密集型还是 IO 密集型可能是动态变化的，低优先级队列中的 IO 型任务的响应时间被拉升，调度频率下降。

2）作弊问题

• 基于 MLFQ 对 IO 型任务的偏爱，用户可能为 CPU 密集型任务编写无用的 IO 空操作，从而人为提升任务优先级，相当于作弊。

针对上述问题 MLFQ 还需增加几个修正:

• 周期性提升所有任务的优先级到最高，避免饥饿问题。

• 调度程序记录任务在某个层级队列中消耗的时间片，如果达到某个比例，无论期间是否有 IO 操作都降低任务的优先级，通过计时来确定真正的 IO 型任务。

MLFQ 的算法思想在 1962 年被提出，其作者也获得了图灵奖，可谓是影响深远。

三、实验过程

对 Linux 的进程调度器进行研究:

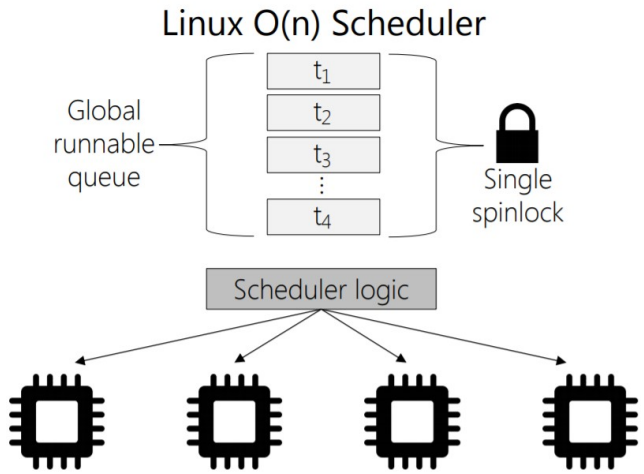
Linux 的进程调度器是不断演进的，先后出现过三种里程碑式的调度器:

调度器种类	内核版本
O(n)调度器	linux0.11 - 2.4
O(1)调度器	linux2.6
CFS调度器	linux2.6至今

O(n)属于早期版本在 pick next 过程是需要遍历进程任务队列来实现，O(1)版本性能上有较大提升可以实现 O(1)复杂度的 pick next 过程。

CFS 调度器可以说是一种 O(logn)调度器，但是其算法思想相比前两种有些不同，并且设计实现上也更加轻量，一直被 Linux 内核沿用至今。

3.1 O(n)调度器



O(n)调度器采用一个全局队列 runqueue 作为核心数据结构，具备以下特点:

- 多个 cpu 共享全局队列，并非每个 cpu 有单独的队列
- 实时进程和普通进程混合且无序存放，寻找最合适进程需要遍历
- 就绪进程将被添加到队列，运行结束被删除
- 全局队列增删进程任务时需要加锁
- 进程被挂载到不同 CPU 运行，缓存利用率低

```
1.  struct task_struct {
2.  long counter;
```

```
3.  long nice;
4.  unsigned long policy;
5.  int processor;
6.  unsigned long cpus_runnable, cpus_allowed;
7.  }
```

counter 值在调度周期开始时被赋值，随着调度的进行递减，直至 counter=0 表示无可用 CPU 时间，等待下一个调度周期。

O(n)调度器中调度权重是在 goodness 函数中完成计算的:

```
1.  static inline int goodness(struct task_struct * p, int this_cpu, struct
2.  mm_struct *this_mm)
3.  {
4.      int weight;
5.      weight = -1;
6.      /*进程可以设置调度策略为 SCHED_YIELD 即“礼让”策略,这时候它的权值为-1,权值相当
7.  低*/
8.      if (p->policy & SCHED_YIELD)
9.          goto out;
10.     /*
11.     * Non-RT process - normal case first.
12.     */
13.
14.     /*对于调度策略为 SCHED_OTHER 的进程，没有实时性要求，它的权值仅仅取决于
15.     * 时间? 的剩余和它的 nice 值，数值上 nice 越小，则优先级越高，总的权值 = 时间片剩
    余 + (20 - nice)
16.     * */
17.     if (p->policy == SCHED_OTHER) {
18.     /*
19.     * Give the process a first-approximation goodness value
20.     * according to the number of clock-ticks it has left.
```

```
21.     *
22.     * Don't do any other calculations if the time slice is
23.     * over..
24.     */
25.         weight = p->counter;
26.         if (!weight)
27.             goto out;
28. #ifdef CONFIG_SMP
29.     /* Give a largish advantage to the same processor... */
30.     /* (this is equivalent to penalizing other processors) */
31.     if (p->processor == this_cpu)
32.         weight += PROC_CHANGE_PENALTY;
33. #endif
34.     /* .. and a slight advantage to the current MM */
35.     if (p->mm == this_mm || !p->mm)
36.         weight += 1;
37.     weight += 20 - p->nice;
38.     goto out;
39.     }
40.     /*
41.     *对于实时进程，也就是 SCHED_FIFO 或者 SCHED_RR 调度策略，
42.     *具有 个实时优先级，总的权值仅仅取决于该实时优先级，
43.     *总的权值 = 1000 + 实时优先级。
44.     * */
45.     weight = 1000 + p->rt_priority;
46. out:
47.     return weight;
48. }
```

从代码可以看到:

• 当进程的剩余时间片 Counter 为 0 时，无论静态优先级是多少都不会被选中

• 普通进程的优先级=剩余时间片 Counter+20-nice

• 实时进程的优先级=1000+进程静态优先级

• 实时进程的动态优先级远大于普通进程，更容易被选中

• 剩余时间片 counter 越多说明进程 IO 较多，分配给它的没用完，被调度的优先级需要高一些

O(n)调度器对实时进程和普通进程采用不同的调度策略:

• 实时进程采用的是 SCHED_RR 或者 SCHED_FIFO,高级优先&同级轮转或者顺序执行

• 普通进程采用的是 SCHED_OTHER

• 进程采用的策略在 task_struct 中 policy 体现

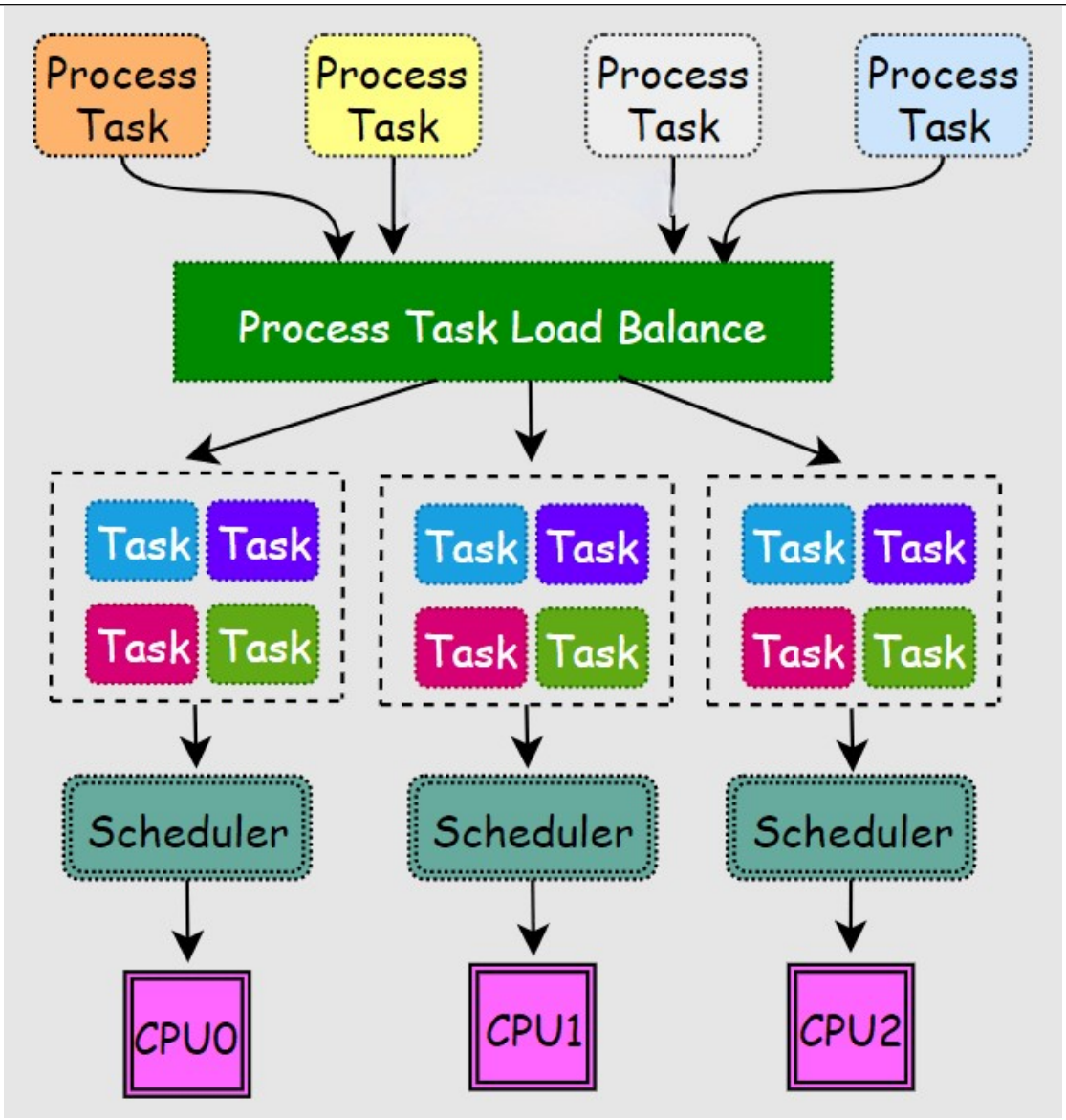
在 runqueue 中搜索下一个合适的进程是基于动态优先级来实现的，动态优先级最高的就是下一个被执行的进程。

O(n)调度器设计和实现上存在一些问题，但是其中的很多思想为后续调度器设计指明了方向，意义深远。

3.2 O(1)调度器

Linux 2.6.0 采纳了 Red Hat 公司 Ingo Molnar 设计的 O(1)调度算法，该调度算法的核心思想基于 Corbato 等人提出的多级反馈队列算法。

O(1)调度器引入了多个队列，并且增加了负载均衡机制，对新出现的进行任务分配到合适的 cpu-runqueue 中:



为了实现 O(1)复杂度的 pick-next 算法，内核实现代码量增加了一倍多，其有以下几个特点:

• 实现了 per-cpu-runqueue,每个 CPU 都有一个就绪进程任务队列

• 引入活跃数组 active 和过期数组 expire，分别存储就绪进程和结束进程

•采用全局优先级:实时进程 0-99,普通进程 100-139，数值越低优先级越高，更容易被调度

• 每个优先级对应一个链表，引入 bitmap 数组来记录 140 个链表中的活跃任务情况

任务队列的数据结构:

```
1.  struct runqueue {
2.      spinlock_t lock;
```

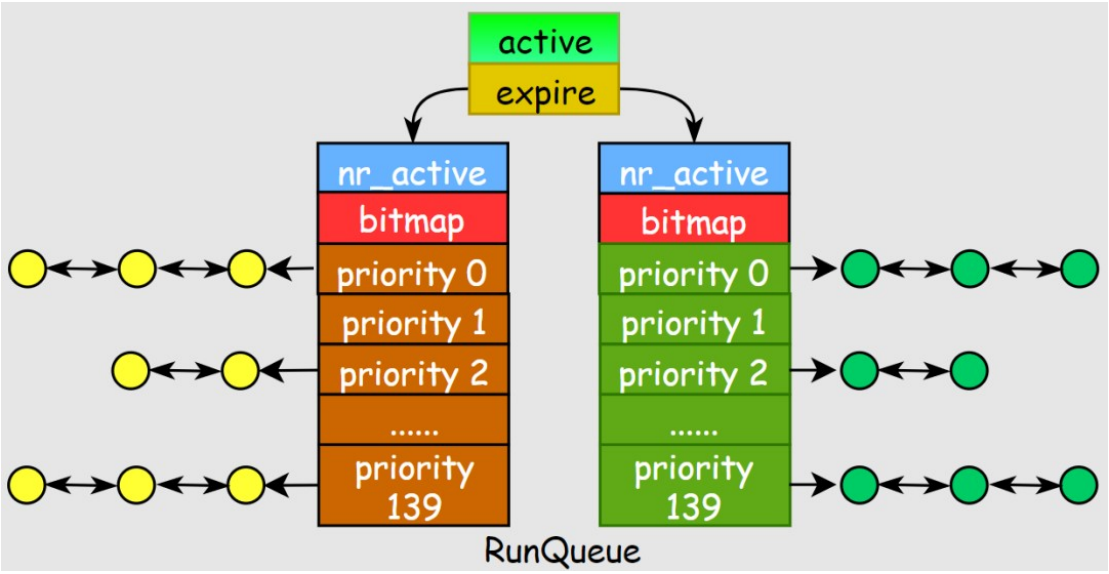
```
    unsigned long nr_running;
    unsigned long long nr_switches;
    unsigned long expired_timestamp, nr_uninterruptible;
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;
    ......
};
```

active 和 expired 是指向 prio_array_t 的结构体指针

arrays 是元素为 prio_array_t 的结构体数组

prio_array_t 结构体的定义:

```
#define BITMAP_SIZE ((((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))
typedef struct prio_array prio_array_t;
struct prio_array {
    unsigned int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};
```



O(1)调度器对 pick-next 的实现:

• 在 runqueue 结构中有 active 和 expire 两个数组指针，active 指向就绪进程的结构，从 active-bitmap 中寻找优先级最高且非空的数组元素，这个数组是元素是进程链表，找该链表中第 1 个进程即可。

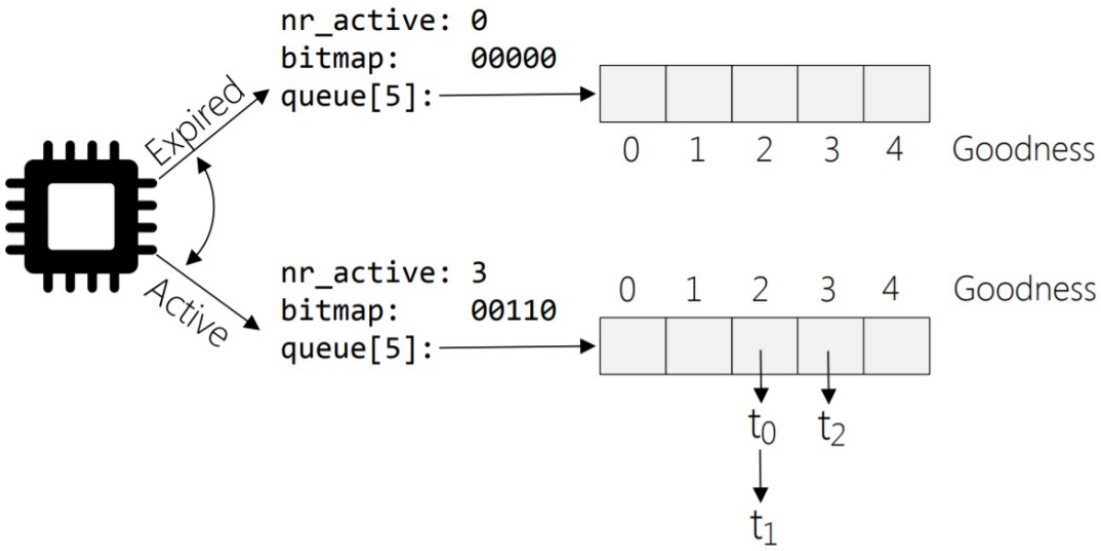
```
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);
```

• 当 active 的 nr_ active=0 时表示没有活跃任务，此时进行 active 和 expire 双指针互换，速度很快。

```
array = rq->active;
if (unlikely(!array->nr_active)) {
    /*
    * Switch the active and expired arrays.
    */
    rq->active = rq->expired;
    rq->expired = array;

    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = MAX_PRIO;
```

```
12. }
```



O(1)和 O(n)调度器确定进程优先级的方法不一样，0(1)借助了 sleep_avg 变量记录进程的睡眠时间，来识别 IO 密集型进程，计算 bonus 值来调整优先级:

```
1.  #define NICE_TO_PRIO(nice) (MAX_RT_PRIO + (nice) + 20)
2.  #define NS_TO_JIFFIES(TIME) ((TIME) / (1000000000 / HZ))
3.  #define CURRENT_BONUS(p)
4.      (NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS /MAX_SLEEP_AVG)
5.  static int effective_prio(task_t *p)
6.  {
7.      int bonus, prio;
8.      if (rt_task(p))
9.          return p->prio;
10.     bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;
11.     prio = p->static_prio - bonus;
12.     if (prio < MAX_RT_PRIO)
13.         prio = MAX_RT_PRIO;
14.     if (prio > MAX_PRIO-1)
15.         prio = MAX_PRIO-1;
16.     return prio;
17. }
```

O(1)调度器为了实现复杂场景 IO 密集型任务的识别，做了大量的工作仍然无法到达 100%的准确，但不可否认 O(1)调度器是一款非常优秀的产品。

3.3CFS 调度器

O(1)调度器本质上是 MLFQ 算法的思想，随着时间的推移也暴露除了很多问题，主要集中在 O(1)调度器对进程交互性的判断上积重难返。

无论是 O(n)还是 O(1)都需要去根据进程的运行状况判断它属于 IO 密集型还是 CPU 密集型，再做优先级奖励和惩罚，这种推测本身就会有误差，而且场景越复杂判断难度越大。

在内核中引入 scheduling class 的概念，将调度器模块化，系统中可以有多种调度器，使用不同策略调度不同类型的进程:

- DL Scheduler 采用 sched_deadline 策略
- RT Scheduler 采用 sched_rr 和 sched_fifo 策略
- CFS Scheduler 采用 sched_normal 和 sched_batch 策略
- IDEL Scheduler 采用 sched_jidle 策略

这样一来，CFS 调度器就不关心实时进程了，专注于普通进程就可以了。

CFS(Completely Fair Scheduler)完全公平调度器，从实现思想上和之前的 O(1)/O(n)不一样。

3.3.1 优先级和权重

O(1)和 O(n)都将 CPU 资源划分为时间片，采用了固定额度分配机制，在每个调度周期进程可使用的时间片是确定的，调度周期结束被重新分配。

CFS 摒弃了固定时间片分配，采用动态时间片分配，本次调度中进程可占用的时间与进程总数、总 CPU 时间、进程权重等均有关系，每个调度周期的值都可能会不一样。

CFS 调度器从进程优先级出发，它建立了优先级 prio 和权重 weight 之间的映射关系，把优先级转换为权重来参与后续的计算:

```
1. const int sched_prio_to_weight[40] = {
2. /* -20 */ 88761, 71755, 56483, 46273, 36291,
3. /* -15 */ 29154, 23254, 18705, 14949, 11916,
4. /* -10 */ 9548, 7620, 6100, 4904, 3906,
```

```
/* -5 */ 3121, 2501, 1991, 1586, 1277,
/* 0 */ 1024, 820, 655, 526, 423,
/* 5 */ 335, 272, 215, 172, 137,
/* 10 */ 110, 87, 70, 56, 45,
/* 15 */ 36, 29, 23, 18, 15,
};
```

普通进程的优先纸范围是[100,139], prio 整体减小 120 就和代码左边的注释对上了，也就是 nice 值的范围[-20,19]，因此 sched_prio=0 相当于 static_ prio=120。

比如现有进程 A sched_prio=0，进程 B sched_prio=-5, 通过 sched_prio_to_weight 的映射:

- 进程 A weight=1024，进程 B weight = 3121
- 进程 A 的 CPU 占比= 1024/(1024+3121)= 24.7%
- 进程 B 的 CPU 占比= 3121/(1024+3121)= 75.3%
- 假如 CPU 总时间是 10ms，那么根据 A 占用 2.47ms，B 占用 7.53ms

在 CFS 中引入 sysctl_sched_latency(调度延迟)作为一个调度周期，真实的 CPU 时间表示为:

$$\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{n=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency}$$

显然这样根据权重计算后的各个进程的运行时间是不等的，也就违背了"完全公平"思想，于是 CFS 引入了虚拟运行时间(virtual runtime)。

3.3.2 虚拟运行时间

每个进程的物理运行时间时肯定不能一样的，CFS 调度器只要保证的就是进程的虚拟运行时间相等即可。

那虚拟运行时间该如何计算呢?

$$\text{virtual\_time} = \text{wall\_time} * \text{nice\_0\_weight}/\text{sched\_prio\_ to\_weigh}$$

比如现有进程 A sched_prio=0， 进程 B sched_prio=-5:

- 调度延迟=10ms, A 的运行时间=2.47ms B 的运行时间=7.53ms,也就是

wall_time

- nice_0 _weight 表示 sched_prio=0 的权重为 1024
- 进程 A 的虚拟时间: 2.47*1024/1024=2.47ms
- 进程 B 的虚拟时间: 7.53*1024/3121=2.47ms

经过这样映射，A 和 B 的虚拟时间就相等了。

上述公式涉及了除法和浮点数运算，因此需要转换成为乘法来保证数据准确性，再给出虚拟时间计算的变形等价公式:

$$virtual\_time = (wall\_time * nice\_0\_weight * 2^{32}/sched\_prio\_to\_weigh) >> 32$$

令 $inv\_weight = 2^{32}/sched\_prio\_to\_weigh$

则 $virtual\_time = (wall\_time * 1024 * inv\_weight) >> 32$

由于 sched_prio_to_weigh 的值存储在数组中，inv_weight 同样可以:

```
const u32 sched_prio_to_wmult[40] = {
/*-20 */ 48388, 59856, 76040, 92818, 118348,
/*-15 */ 147320, 184698, 229616, 287308, 360437,
/*-10 */ 449829, 563644, 704093, 875809, 1099582,
/* -5 */ 1376151, 1717300, 2157191, 2708050, 3363326,
/*  0 */ 4194304, 5237765, 6557202, 8165337, 10153587,
/*  5 */ 12820798, 15790321, 19976592, 24970740, 31350126,
/* 10 */ 39045157, 49367440, 61356676, 76695844, 95443717,
/* 15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
```

经过一番计算，各个进程的虚拟运行时间一致了，似乎我们理解了"完全公平"的思想。

虚拟运行时间与优先级的衰减因子有关，也就是 inv_weight 随着 nice 值增大而增大，同时其作为分母也加速了低优先级进程的衰减。

- nice=0 虚拟运行时间=物理运行时间
- nice>0 虚拟运行时间>物理运行时间
- nice<0 虚拟运行时间<物理运行时间

简言之: CFS 将物理运行时间在不同优先纸进程中发生了不同的通胀。

摒弃了固定时间片机制也是 CFS 的亮点，系统负载高时大家都少用一些 CPU,系统负载低时大家都多用一些 CPU，让调度器有了一定的自适应能力。

3.3.3 pick-next 和红黑树

那么这些进程应该采用哪种数据结构来实现 pick-next 算法呢?

CFS 调度器采用了红黑树来保存活跃进程任务，红黑树的增删查复杂度是 O(logn),但是 CFS 引入了一些额外的数据结构，可以免去遍历获得下一个最合适的进程。

红黑树的 key 是进程已经使用的虚拟运行时间，并且把虚拟时间数值最小的放到最左的叶子节点，这个节点就是下一个被 pick 的进程了。

前面已经论证了，每个进程的虚拟运行时间是一样的，数值越小表示被调度的越少，因此需要更偏爱一些，当虚拟运行时间耗尽则从红黑树中删除，下个调度周期开始后再添加到红黑树上。

四、实验总结

1.O(n)调度器采用全局 runqueue，导致多 cpu 加锁问题和 cache 利用率低的问题

2. O(1)调度器为每个 cpu 设计了一个 runqueue,并且采用 MLFQ 算法思想设置 140 个优先级链表和 active/expire 两个双指针结构

3.CFS 调度器采用红黑树来实现 O(logn)复杂度的 pick-next 算法，摒弃固定时间片机制，采用调度周期内的动态时间机制

4. O(1)和 O(n) 都在交互进程的识别算法上下了功夫，但是无法做到 100%准确

5. CFS 另辟蹊径采用完全公平思想以及虚拟运行时间来实现进行的调度，但并非完美，在某些方面可能不如 O(1)

五、参考文献

[1]进程.https://zh.wikipedia.org/wiki/行程

[2] Linux 进程描述符 task_struct 结构体详解--Linux 进程的管理与调度 https://blog.csdn.net/gatieme/article/details/51383272

[3] Linux 进程管理之 task_struct 结构体

https://blog.csdn.net/qpy_Jp/article/details/7292563

[4] Linux 进程: task_struct 结构体成员

https://www.cnblogs.com/iujuan/p/11715853.html

[5] Linux 进程描述符 task_ struct 结构体详解-Linux 进程的管理与调度

https://cloud.tencent.com/ developer/atticle/1339556

[6] Linux 进程调度器-基础 https://www.cnblogs.com/LovenWang/p/ 12249106.html

[7]吐血整理 | 肝翻 Linux 进程调度所有知识点 https://www

eefocus.com/embedded/ 498554

[8] The Linux Completely Fair Scheduler (CFS)

https://cs334.cs.vassar.edu/slides/04_Linux CFS.pdf

[9] Scheduling: Case Studies http://wwcecs.harvard.edu/-cs161/notes/

scheduling-case-studiespdf

[10] O(n)、O(1)和 CFS 调度器

http://www.wowotech.net/process_management/scheduler-history.html