

# openEuler 进程调度

姓名：麻家乐 学号：521021910559

## 目录

### Chapter 1 进程基本概念

- 1、什么是进程
  - 1.1 进程的定义
  - 1.2 进程与线程
- 2、进程从创建到终止
  - 2.1 关系图
  - 2.2 进程的状态
- 3、进程调度简单介绍
  - 3.1 什么是进程调度
  - 3.2 为什么需要进程调度

### Chapter 2 struct task\_struct (该部分可以先跳过)

- 1、结构体总体介绍
  - 1.1 为什么要介绍 struct task\_struct
    - 1.2 struct task\_struct 重要成员组
- 3、进程与内核栈
- 4、进程的亲属关系
- 5、进程的标识符PID
- 6、进程的标记flags
- 7、进程的ptrace
- 8、进程的统计信息
- 9、进程的调度
- 10、struct task\_struct 总结

### Chapter 3 进程调度

- 1、进程的调度流程
  - 1.1 再次提及进程调度
  - 1.2 进程调度的流程
  - 1.3 一些补充
- 2、struct rq
- 3、schedule
- 4、\_\_schedule
- 5、Chapter3 总结

### Chapter 4 CFS (完全公平调度) 介绍

- 1.1 CFS基本原理概述
- 1.2 CFS相关概念
- 1.3 CFS算法设计核心
- 2、struct cfs\_rq
- 3、struct sched\_entity
- 4、struct sched\_class
- 5、struct sched\_class fair\_sched\_class
- 6、Chapter4 总结 & 完整思维导图

### Chapter 5 选做部分

# 结构体函数文件对照表

文件	序号	内容
/include/linux/sched.h	1	进程状态的宏定义
	2	struct task_struct
	3	flags宏定义
	4	struct load_weight
	5	struct sched_statistics
	6	struct cfs_rq
	7	struct sched_entity
	8	union thread_union
/kernel/sched/sched.h	1	struct rq
	2	struct cfs_rq
	3	struct sched_class
/kernel/sched/core.c	1	schedule( )
	2	__schedule( )
	3	pick_next_task( )
	4	context_switch( )
/include/linux/rbtree.h	1	struct rb_node
	2	struct rb_root
	3	struct rb_root_cached
/lib/rbtree.c	1	红黑树中函数实现的文件
/kernel/arch/.../thread_info.h	1	各个平台下的thread_info.h
/include/upai/linux/sched.h	1	5种调度器类的宏定义
/kernel/sched/fair.c	1	全部内容都是关于 fair_sched_class

结构体函数文件对照表

# Chapter1 进程基本概念

## 1 什么是进程

### 1.1 进程的定义

**【中文定义】**进程是计算机中的一个具有独立功能的程序关于某数据集合上的一次运行活动，是系统进行资源分配的基本单位。

**【英文定义】**Process is an instance of a computer program that is being executed.

### 1.2 进程与线程

**【线程的定义】**线程（thread）是操作系统能够进行运算调度的最小单位。

**【线程与进程】**进程是操作系统资源分配的基本单位，而线程是操作系统调度的基本单位。一个进程可以包含多个线程，这些线程共享该进程的所有资源，如内存空间、文件句柄等。

**【linux 中的 task\_struct】**在 linux 中并不区分进程和线程，都是用 task\_struct 来抽象。支持多线程的进程是由一组 task\_struct 来抽象，而这些 task\_struct 会共享一些数据结构。linux 用 thread ID 来唯一标识进程中的线程，对于单线程的进程，process ID 和 thread ID 是一样的，对于支持多线程的进程，每个线程有自己的 thread ID，但是所有的线程共享一个 PID。

**【注】**task\_struct 是一个庞大的结构体，本文将在 Chapter2 进行详细介绍

## 2、进程从创建到终止

## 2.1 关系图



图1-1 进程从创建到终止

## 2.2 进程的状态

从流程图中可以看出，进程有以下几个状态：

【创建状态】一个新进程被创建时的第一个状态

【就绪状态】进程已经准备好并分配到所需资源，此时，只要分配到CPU就能够立即运行

【阻塞状态】进程由于等待某些事件而暂停执行，如等待I/O操作完成

【执行状态】进程处于正在运行的状态

【终止状态】进程已经完成了它的任务，或者由于某些原因被操作系统强制终止

注：以上状态只是人们为了方便理解而抽象出的五种状态，事实上在linux中并非完全如此划分

## 3、进程调度简单介绍

### 3.1 什么是进程调度

所谓进程调度，就是指在处于就绪态的一堆进程中，按照一定的调度算法，选出一个进程并给它分配CPU时间让它运行，从而实现多进程的并发执行。

### 3.2 为什么需要进程调度

由于一个CPU同时只能处理一个进程，而就绪队列里面往往有大量的进程等待CPU资源，因此需要设计进程调度动态地从众多的就绪进程中选择一个最合适的进程来占用CPU资源。

进程调度的主要目的以及设计进程调度算法的宗旨是为了充分利用计算机系统中的CPU资源，让计算机能够又好又快省的完成各种任务。

## Chapter2 struct task\_struct

### 1、结构体总体介绍

#### 1.1 为什么要介绍 struct task\_struct

linux内核中使用 `task_struct` 结构来表示一个进程，这个结构体保存了进程的所有信息，所以它非常庞大，在讲解linux内核的进程调度前，我们有必要先初步了解这个 `task_struct` 中的一些成员

#### 1.2 struct task\_struct 主要成员组



图2-1 struct task\_struct 主要成员组

## 2、进程状态

### 2.1 思维导图

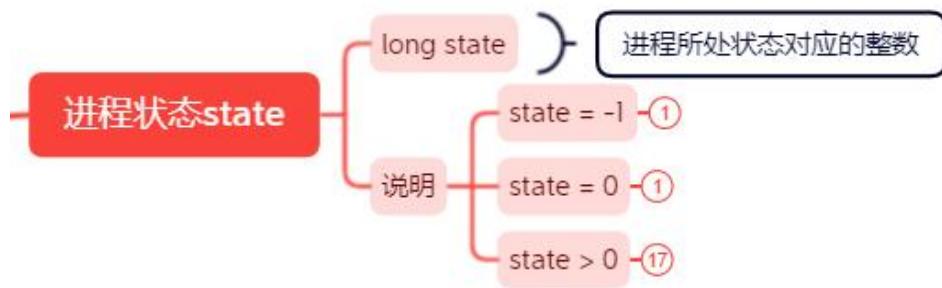


图2-2 进程状态思维导图

### 2.2 源代码

```

struct task_struct {
#ifndef CONFIG_THREAD_INFO_IN_TASK
/*
 * For reasons of header soup (see current_thread_info()), this
 * must be the first element of task_struct.
 */
struct thread_info thread_info;
#endif /* -1 unrunnable, 0 runnable, >0 stopped: */
volatile long state;

/* Used in tsk->state: */
#define TASK_RUNNING 0x0000
#define TASK_INTERRUPTIBLE 0x0001
#define TASK_UNINTERRUPTIBLE 0x0002
#define __TASK_STOPPED 0x0004
#define __TASK_TRACED 0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD 0x0010
#define EXIT_ZOMBIE 0x0020
#define EXIT_TRACE (EXIT_ZOMBIE | EXIT_DEAD)

```

图2-3 进程状态与宏定义源代码

### 2.3 详细介绍

task\_struct 中表示进程状态的变量是 state，它是一个 long 数据类型的变量。

当 state = 0 时，表示进程处于 TASK\_RUNNING 状态 (runnable)。这表示进程可以运行，或者正在运行，或者在运行队列中等待运行。至于如何判断是否正在运行，task\_struct 并没有给出相应的变量，不过可以根据CPU上的队列中的指针 \*curr 来判断。

当 state = -1 时，表示进程不可运行 (unrunnable)。这通常是因为进程被冻结了或正在退出，此时需要通过 task\_struct 的标记符号 flags 的值来判断进程所处的状态（后面会提及）。

当 state > 0 时，表示进程正处于非运行状态 (stopped)。非运行状态是一系列状态的集合，它包括 TASK\_INTERRUPTIBLE、TASK\_UNINTERRUPTIBLE 等状态，详细见下图：

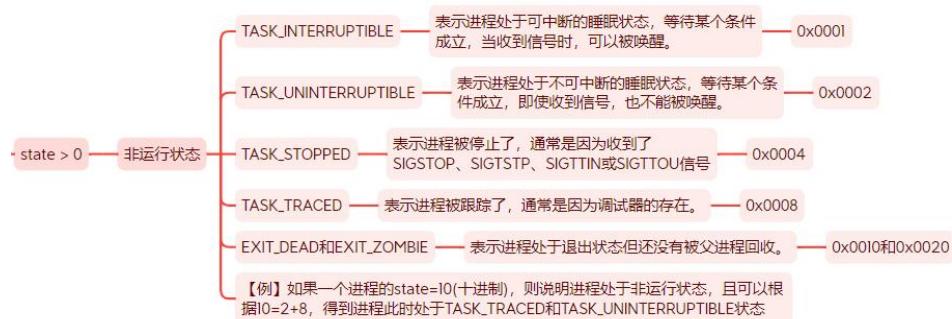


图2-4 stopped 状态

### 3、进程与内核栈

#### 3.1 思维导图



进程内核栈与进程描述符的关系如下图：

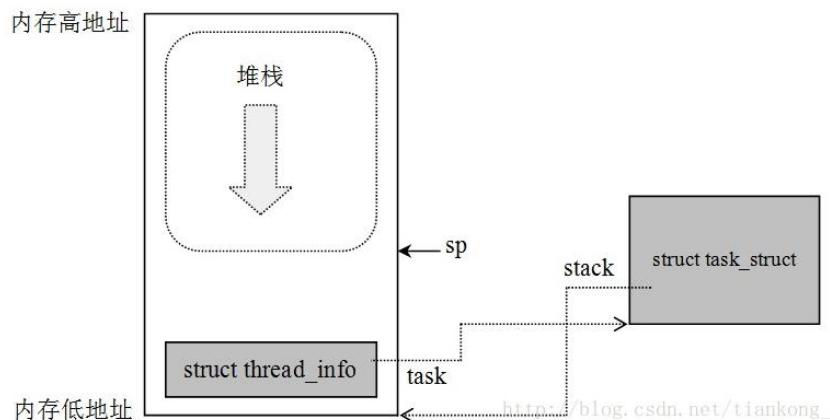


图2-5 进程与进程内核栈思维导图

#### 3.2 源代码

```
/*
 * This begins the randomizable portion of task_struct. Only
 * scheduling-critical items should be added above here.
 */
randomized_struct_fields_start

void *stack;
```

图2-6 进程与进程内核栈源代码

#### 3.3 详细介绍

内核在创建进程的时候，会为进程创建相应的堆栈。每个进程会有两个栈，一个用户栈，存在于用户空间，一个内核栈，存在于内核空间。当进程在用户空间运行时，CPU堆栈指针寄存器里面的内容是用户堆栈地址，使用用户栈；当进程在内核空间时，CPU堆栈指针寄存器里面的内容是内核栈空间地址，使用内核栈。

当进程因为中断或者系统调用而陷入内核态之行时，进程所使用的堆栈也要从用户栈转到内核栈。进程进入内核态后，先把用户态堆栈的地址保存在内核栈之中，然后设置堆栈指针寄存器的内容为内核栈的地址，这样就完成了用户栈向内核栈的转换。当进程从内核态恢复到用户态时，在内核态执行的最后将保存在内核栈里面的用户栈的地址恢复到堆栈指针寄存器即可。这样就实现了内核栈和用户栈的互转。

那么，我们知道从内核转到用户态时用户栈的地址是在陷入内核的时候保存在内核栈里面的，但是在陷入内核的时候，我们是如何知道内核栈的地址的呢？

关键在进程从用户态转到内核态的时候，进程的内核栈总是空的。这是因为，当进程在用户态运行时，使用的是用户栈，当进程陷入到内核态时，内核栈保存进程在内核态运行的相关信息，但是一旦进程返回到用户态后，内核栈中保存的信息无效，会全部恢复，因此每次进程从用户态陷入内核的时候得到的内核栈都是空的。所以在进程陷入内核的时候，直接把内核栈的栈底地址给堆栈指针寄存器就可以了。

讲了这么多关于进程内核栈理论方面的内容，那么linux具体是如何实现二者之间的切换的呢？

事实上，task\_struct 结构体为实现从用户态转到内核态创建了一个 \*stack 的指针，通过这个指针（经过一系列函数与计算）就能访问进程的内核栈。而为了实现从内核态转到用户态，linux 在设计内核栈时，事实上创建的是 union thread\_union，如下图所示：

```
union thread_union {
#ifndef CONFIG_ARCH_TASK_STRUCT_ON_STACK
    struct task_struct task;
#endif
#ifndef CONFIG_THREAD_INFO_IN_TASK
    struct thread_info thread_info;
#endif
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

图2-7 thread\_union 源代码

引入 union 而不是使用 struct 结构是为了让内核栈与 thread\_info 位于同一块内存。由于 thread\_info 中存在一个 struct task\_struct \*stack 指针，因此，内核态可以通过调用与其处于同一内存地址下的 thread\_info 中的 \*stack 指针来实现从内核态转到用户态。

```
struct thread_info {
    unsigned long flags;           /* low level flags */
    int preempt_count;             /* 0 => preemptable, <0 => BUG */
    struct task_struct *task;       /* main task structure */
    mm_segment_t addr_limit;      /* thread address space */
    __u32 cpu;                    /* current CPU */
    unsigned long thr_ptr;         /* TLS ptr */
};
```

图2-8 arch下thread\_info 源代码

需要注意的是，不同架构的 thread\_info 结构体包含的内容是不尽相同的。linux 支持的CPU架构有很多，本实验环境下可以通过访问arch文件夹查看。

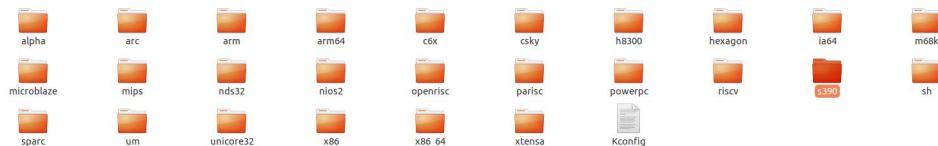


图2-9 支持的架构

事实上，有一些架构中的 thread\_info 并没有 \*stack 指针，例如在 arm64 平台上，thread\_info 结构体并不包含指向 task\_struct 的结构体指针。这是因为，在 arm64 平台上，内核态到用户态的转换是通过 el0 和 el1 两个特权级别来实现的。当进程从用户态切换到内核态时，会从 el0 切换到 el1，然后执行内核代码。当内核代码执行完毕后，会从 el1 切换回到 el0，然后返回用户态。

```

struct thread_info {
    unsigned long flags;          /* low level flags */
    mm_segment_t addr_limit;    /* address limit */
    #ifdef CONFIG_ARM64_SW_TTBR0_PAN
    u64 ttbr0;                /* saved TTBR0_EL1 */
    #endif
    int preempt_count;        /* 0 => preemptable, <0 => bug */
};

```

图2-10 arm64下thread\_info 源代码

## 4、进程的亲属关系

### 4.1 思维导图



图2-11 进程的亲属关系思维导图

### 4.2 源代码

```

/*
 * Pointers to the (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->real_parent->pid)
 */

/* Real parent process: */
struct task_struct __rcu *real_parent;

/* Recipient of SIGCHLD, wait4() reports: */
struct task_struct __rcu *parent;

/*
 * Children/sibling form the list of natural children:
 */
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;

```

图2-12 进程的亲属关系源代码

### 4.3 详细介绍

- \*real\_parent: 指向创建当前进程的父进程的task\_struct指针
- \*parent: 通常和real\_parent相同，但在某些情况下，例如进程被ptrace跟踪时，会被修改为跟踪进程。
- children: 链表，包含当前进程的所有子进程的task\_struct指针
- sibling: 链表，包含当前进程的所有兄弟进程的task\_struct指针
- \*group\_leader: 指向当前进程所属线程组的首个线程的task\_struct指针

## 5、进程的标识符 PID

### 5.1 思维导图



图2-13 进程的标识符思维导图

## 5.2 源代码

```
pid_t pid;
pid_t tgid;
```

图2-14 进程的标识符源代码

## 5.3 详细介绍

- pid Process ID，进程号，唯一标识一个进程的task\_struct结构体的数字。
- tgid Thread Group ID，线程组ID，标识一个进程中所有线程的task\_struct结构体的数字。
- 【重点】创建一个新的进程会有新的PID和TID，并且2个值相同；创建一个新的线程的时候，也会有新的PID，但它的tgid与创建它的进程的tgid一致，这样，内核就可以通过tgid知道某个task属于哪个线程组，也就知道属于哪个进程了。当使用ps命令或者getpid()等接口查询进程id时，内核返回的是tgid

```

1  USER VIEW
2  <-- PID 43 --> <----- PID 42 ----->
3          +-----+
4          | process |
5          _| pid=42 |
6          _/ | tgid=42 | \_ (new thread) _
7          _ (fork) _/ +-----+           \
8          /                               +-----+
9          +-----+                         | process |
10         | process |                     | pid=44 |
11         | pid=43 |                     | tgid=42 |
12         | tgid=43 |                     +-----+
13         +-----+
14  <-- PID 43 --> <----- PID 42 -----> <-- PID 44 -->
15                      KERNEL VIEW

```

图2-15 pid 与 tgid

## 6、进程的标记 flags

### 6.1 思维导图



图2-16 进程的标记思维导图

## 6.2 源代码

```

/* Per task flags (PF_*), defined further below: */
unsigned int flags;
unsigned int ptrace;

/*
 * Per process flags
 */
#define PF_IDLE          0x00000002 /* I am an IDLE thread */
#define PF_EXITING       0x00000004 /* Getting shut down */
#define PF_RELIABLE      0x00000008 /* Allocate from reliable memory */
#define PF_VCPU          0x00000010 /* I'm a virtual CPU */
#define PF_WQ_WORKER     0x00000020 /* I'm a workqueue worker */
#define PF_FORKNOEXEC   0x00000040 /* Forked but didn't exec */
#define PF_MCE_PROCESS   0x00000080 /* Process policy on mce errors */
#define PF_SUPERPRIV    0x00000100 /* Used super-user privileges */
#define PF_DUMPCORE     0x00000200 /* Dumped core */
#define PF_SIGNALLED    0x00000400 /* Killed by a signal */
#define PF_MEMALLOC      0x00000800 /* Allocating memory */
#define PF_NPROC_EXCEEDED 0x00001000 /* set_user() noticed that RLIMIT_NPROC was exceeded */
#define PF_USED_MATH     0x00002000 /* If unset the fpu must be initialized before use */
#define PF_NOFREEZE    0x00008000 /* This thread should not be frozen */
#define PF_FROZEN        0x00010000 /* Frozen for system suspend */
#define PF_KSWAPD       0x00020000 /* I am kswapd */
#define PF_MEMALLOC_NOFS 0x00040000 /* All allocation requests will inherit GFP_NOFS */
#define PF_MEMALLOC_NOIO 0x00080000 /* All allocation requests will inherit GFP_NOIO */
#define PF_LESS_THROTTLE 0x00100000 /* Throttle me less: I clean memory */
#define PF_KTHREAD       0x00200000 /* am a kernel thread */
#define PF_RANDOMIZE    0x00400000 /* Randomize virtual address space */
#define PF_SWAPWRITE     0x00800000 /* Allowed to write to swap */
#define PF_UCE_KERNEL_COREDUMP 0x01000000 /* Task in a redump process which is used in uce kernel recovery */
#define PF_UCE_KERNEL_RECOVERY 0x02000000 /* Task in uce kernel recovery state */
#define PF_NO_SETAFFINITY 0x04000000 /* Userland is not allowed to meddle with cpus_allowed */
#define PF_MCE_EARLY     0x08000000 /* Early kill for mce process policy */
#define PF_IO_WORKER    0x20000000 /* Task is an IO worker */
#define PF_FREEZER_SKIP  0x40000000 /* Freezer should not count it as freezable */
#define PF_SUSPEND_TASK 0x80000000 /* This thread called freeze_processes() and should not be frozen */

```

图2-17 进程的标记及其宏定义源代码

### 6.3 详细介绍

flags有很多种可能的取值，分别代表不同的状态，每种值所对应的状态如上图宏定义所示，此外，与state类似，flags对应的状态也是可以多种状态的叠加

## 7、进程的 ptrace

### 7.1 思维导图

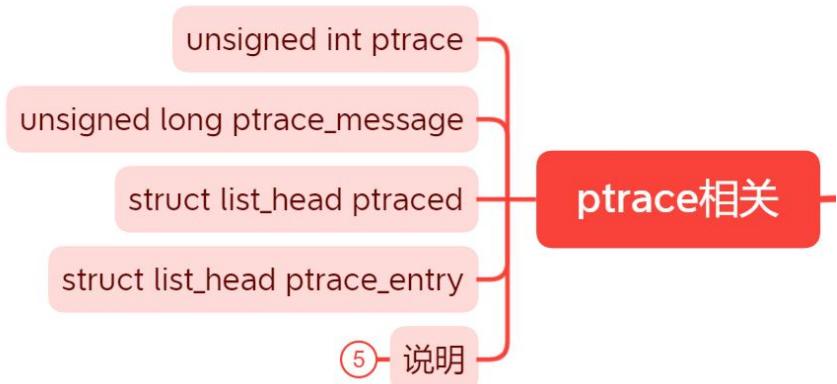


图2-18 进程的 ptrace 思维导图

### 7.2 源代码

```

/* Per task flags (PF_*), defined further below: */
unsigned int flags;
unsigned int ptrace;

/*
 * 'ptraced' is the list of tasks this task is using ptrace() on.
 *
 * This includes both natural children and PTRACE_ATTACH targets.
 * 'ptrace_entry' is this task's link on the p->parent->ptraced list.
 */
struct list_head ptraced;
struct list_head ptrace_entry;

/* Ptrace state: */
unsigned long siginfo_t

```

ptrace\_message;

\*last\_siginfo;

图2-19 进程的 ptrace 源代码

### 7.3 详细介绍

- ptrace是一个系统调用，它可以让一个进程（跟踪者）观察和控制另一个进程（被跟踪者）的执行，并检查和修改被跟踪者的内存和寄存器。它主要用于实现断点调试和系统调用跟踪。要使用ptrace，首先需要将被跟踪者附加到跟踪者上。
- ptrace: 整数值，记录当前进程是否被跟踪，以及跟踪的模式和选项
- ptrace\_message: 用于传递ptrace事件的附加信息
- ptraced: 一个链表，包含当前进程跟踪的所有进程的task\_struct指针
- ptrace\_entry: 一个链表节点，用于将当前进程加入到跟踪者的ptraced链表中

## 8、进程的统计信息

### 8.1 思维导图

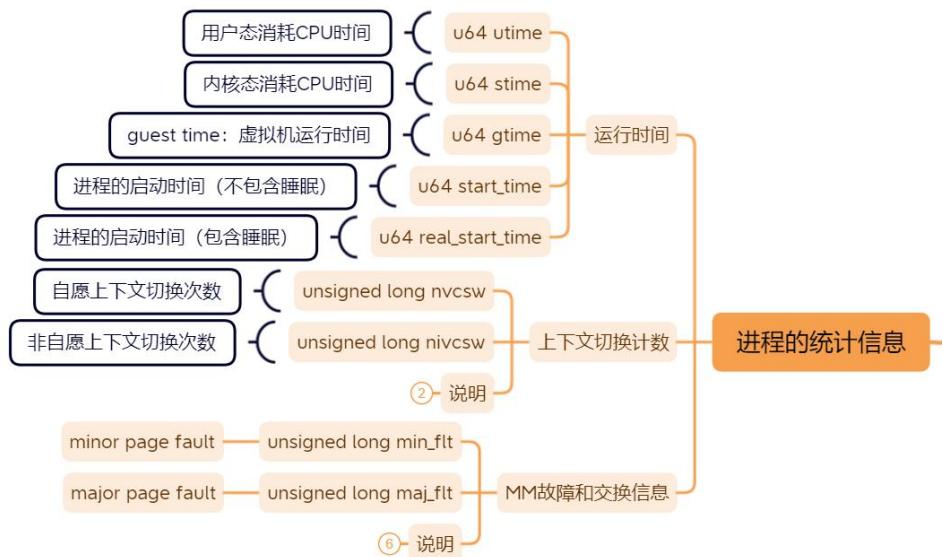


图2-20 进程的统计信息思维导图

### 8.2 源代码

```

/* Context switch counts: */
unsigned long          nvcsw;
unsigned long          nivcsw;

/* Monotonic time in nsecs: */
u64                   start_time;

/* Boot based time in nsecs: */
u64                   real_start_time;

/* MM fault and swap info: this can arguably be seen as either mm-specific or thread-specific: */
unsigned long          min_flt;
unsigned long          maj_flt;

u64
u64
#endif CONFIG_ARCH_HAS_SCALED_CPUTIME
u64
u64
u64
u64
struct prev_cputime
... ...
  
```

<pre> u64 u64 #endif CONFIG_ARCH_HAS_SCALED_CPUTIME u64 u64 u64 u64 struct prev_cputime ... ...   </pre>	<pre> utime; stime;  utimescaled; stimescaled;  gtime; prev_cputime;   </pre>
--	---

图2-21 进程的统计信息源代码

### 8.3 详细介绍

#### (一) Time

- utime: user time 用户态消耗CPU时间
- stime: 内核态消耗CPU时间
- gtime: 虚拟机运行时间
- start\_time: 进程的启动时间 (不包含睡眠)
- real\_start\_time: 进程的启动时间 (包含睡眠)
- real\_start\_time = start\_time + 总的睡眠时间

## (二) Context Switches

- nvcs: Number of Voluntary Context Switches, 自愿 (主动) 上下文切换, 指进程主动放弃CPU, 例如等待I/O操作完成。
- long nivcs: Number of Involuntary Context Switches, 非主动 (自愿) 上下文切换次数, 指进程被迫放弃CPU, 例如时间片用完或者被更高优先级的进程抢占。

## (三) Page Fault

当进程访问它的虚拟地址空间中的PAGE时, 如果这个PAGE目前还不在物理内存中, Linux会产生一个hard page fault中断。

此时, 系统需要做以下几件事, 然后进程才能访问这部分虚拟地址空间的内存。

- (1) 从慢速设备 (如磁盘) 将对应的数据PAGE读入物理内存
- (2) 建立物理内存地址与虚拟地址空间PAGE的映射关系。

- maj\_flt: major page fault, 也称为hard page fault, 指需要访问的内存不在虚拟地址空间, 也不在物理内存中, 因此, 系统需要完成上述的 (1) 与 (2) 。
- 注: 从swap回到物理内存也是hard page fault。
- min\_flt: minor page fault, 也称为soft page fault, 指需要访问的内存不在虚拟地址空间, 但是在物理内存中。此时, 系统只需要完成上述的 (2) 即可。
- 注: 通常多个进程访问同一个共享内存中的数据, 可能某些进程还没有建立起映射关系, 所以访问时会出现 soft page fault

## 9、进程的调度

### 9.1 思维导图

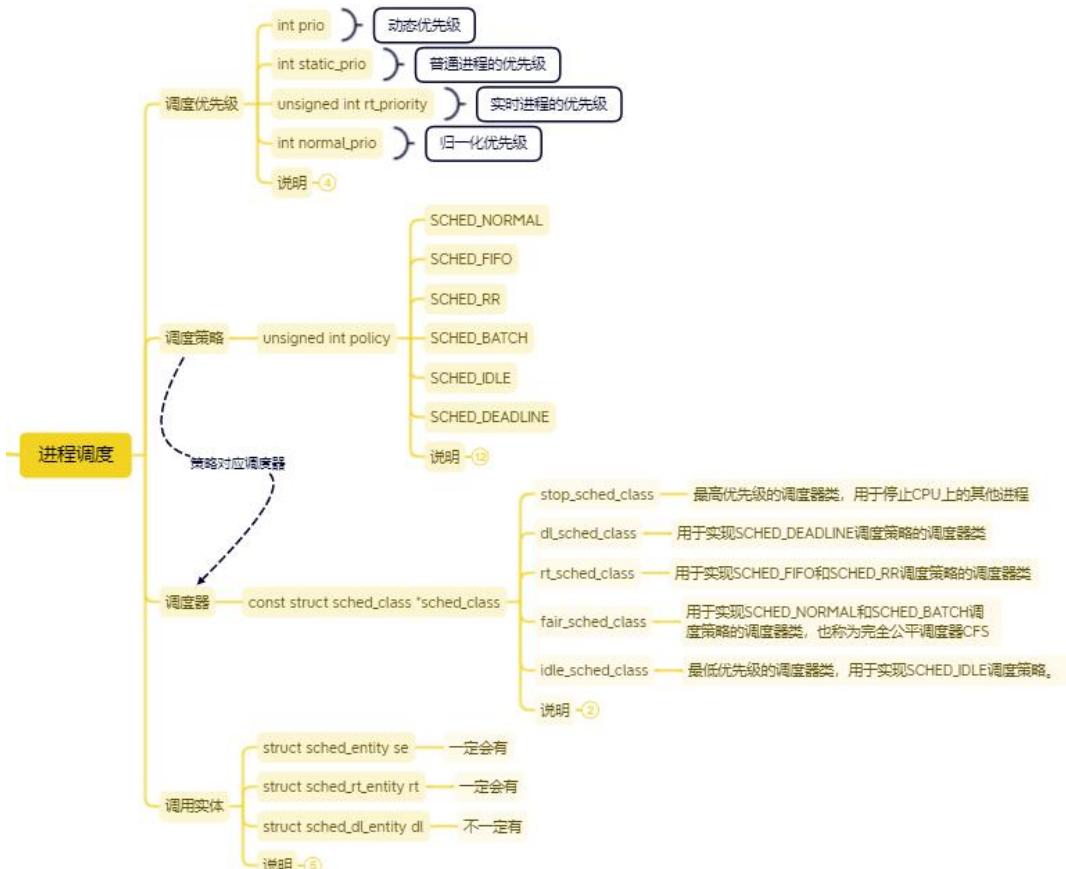


图2-22 进程的调度思维导图

## 9.2 源代码

```
int on_rq;
int prio;
int static_prio;
int normal_prio;
unsigned int rt_priority;

const struct sched_class *sched_class;
struct sched_entity se;
struct sched_rt_entity rt;
struct task_group *sched_task_group;
struct sched_dl_entity dl;

#ifdef CONFIG_PREEMPT_NOTIFIERS
/* List of struct preempt_notifier: */
struct hlist_head preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
unsigned int btrace_seq;
#endif

unsigned int policy;
int nr_cpus_allowed;
cpumask_t cpus_allowed;
```

图2-23 进程调度部分的源代码

## 9.3 详细介绍

### (一) priority

- `prio`: 进程动态优先级，是调度器使用的优先级，它依赖于`normal_prio`，调度器在运行期间可以根据倾向性调整该值
- `static_prio`, 普通进程的优先级，仅能通过`nice`修改，`nice`取值为[-20,19]， $static\_prio = nice + 120$ ，因此 `static_prio` 的取值范围为[100,139]。`static_prio`的值越小优先级越高。
- `rt_priority`, 实时进程的优先级，取值为[0,99]，【注意】`rt_priority`的值越大优先级越大！
- `normal_prio`, 归一化优先级/正常优先级，`normal_prio`值是根据调度器类型计算出来的，对于实时进程： $normal\_prio = 99 - rt\_priority$ ，对于非实时进程： $normal\_prio = static\_prio$

### (二) policy

在 `task_struct` 中，`policy`表示当前进程被调度时所采取的调度策略。调度策略有以下几种：

- `SCHED_NORMAL`: 普通非实时任务的调度策略，它使用完全公平调度 (CFS) 算法，根据任务的优先级 (`nice`值) 和运行时间来分配CPU时间。
- `SCHED_FIFO`, 实时调度策略，使用先进先出 (FIFO) 算法，即按照任务到达的顺序依次执行，直到任务完成或被更高优先级的任务抢占。该策略不使用时间片，所以同一优先级的任务不会相互抢占。
- `SCHED_RR`: 实时调度策略，使用轮转 (RR) 算法，即每个任务都有一个固定的时间片，在时间片用完后，就让出CPU给同一优先级的下一个任务。该策略可以保证同一优先级的任务都有公平的机会执行。
- `SCHED_BATCH`: 批处理调度策略，也使用CFS算法，但是抢占的机会比普通任务少，会减少被调度的次数。该策略适合那些批处理工作，比如大量的计算或数据处理。
- `SCHED_IDLE`: 空闲调度策略，也使用CFS算法，但是优先级最低，只有在没有其他任务可以运行时才会执行。该策略适合那些不紧急的后台任务，比如索引或备份。
- `SCHED_DEADLINE`: 基于截止时间的实时调度策略，使用EDF和CBS算法，支持资源预留。每个任务都有一个预算和一个周期，对应于一个重复的请求-服务模型。该策略可以保证每个任务在其截止时间之前完成其预算，否则会产生截止时间超时。该策略适用于有硬实时或软实时需求的任务。

从源代码中还可以看到：`policy` 是整型数据，事实上，`policy`可以取 0, 1, 2, 3, 5；这些数字分别对应的调度策略如下：

```

/*
 * Scheduling policies
 */
#define SCHED_NORMAL          0
#define SCHED_FIFO            1
#define SCHED_RR              2
#define SCHED_BATCH           3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE            5
#define SCHED_DEADLINE         6

```

图2-24 调度策略

### (三) sched\_class

上述介绍了6种调度策略，事实上，每一种调度策略都会有响应的调度器类，linux共设计了5种调度器类，按照优先级从高到低的顺序如下：

- stop\_sched\_class：最高优先级的调度器类，用于停止CPU上的其他进程
- dl\_sched\_class：(deadline) 用于实现 SCHED\_DEADLINE 调度策略的调度器类
- rt\_sched\_class：(real time) 用于实现 SCHED\_FIFO 和 SCHED\_RR 调度策略的调度器类
- fair\_sched\_class：用于实现 SCHED\_NORMAL 和 SCHED\_BATCH 调度策略的调度器类，也称为完全公平调度器CFS
- idle\_sched\_class：最低优先级的调度器类，用于实现 SCHED\_IDLE 调度策略

注：task\_struct 中 struct sched\_class \*sched\_class 是指向调度器类的指针，struct sched\_class 是linux内核抽象出来的一种调度类，是将各种调度器的共性的特征抽象出来封装成的类，关于它的具体信息我们将在下一章节介绍。

### (四) sched\_entity

- struct sched\_entity se：当采用 cfs 调度器类调度该进程时，调用的是该调度实体 se
- struct sched\_rt\_entity rt：当采用 rt 调度器类调度该进程时，调用的是该调度实体 rt
- struct sched\_dl\_entity dl：当采用 dl 调度器类调度该进程时，调用的是该调度实体 dl

至于stop与idle，由于CPU就绪队列中只有一个stop与idle类型的进程，所以调用时直接调用响应的进程即可，因此无需为这两种类型设计专门的调度实体。

此外，还需要说明的是，一个进程之所以有3个调度实体，本质上是为了在进程切换调度策略时，可以快速地将其在原先调度器的调度信息转化为当前调度器中的调度信息。

例如，当一个进程在SCHED\_NORMAL和SCHED\_FIFO之间切换时，就需要有一个se (sched\_entity) 和一个rt (sched\_rt\_entity) 来分别存储它在完全公平调度器类和实时调度器类的信息，比如虚拟运行时间、优先级、时间片等。这样，当进程切换调度策略时，就可以快速地恢复它在原来的调度器类中的状态，而不需要重新计算或分配。

## 10、struct task\_struct 总结

在 Chatper2 部分，我主要介绍了关于 struct task\_struct 结构体变量功能的介绍，将其划分为8个部分

- 进程状态【state 关于 0 进行划分成三种基本状态，非0时可再根据flags 或 数值进行进一步确定】
- 进程与内核栈【进程通过 \*stack 访问内核栈、内核栈通过与其同内存块的thread\_info中的 \*task 返回进程】
- 进程的亲属关系【父亲、孩子、兄弟、领队】
- 进程的标识符PID【创建进程：新的pid且tgid=pid、创建线程：新的pid且tgid=创建它的进程的tgid】
- 进程的标记flags【进程的各种状态，类似于日志】
- 进程的ptrace【Debug相关】
- 进程的统计信息【运行时间、上下文切换、缺页中断】
- 进程调度【优先级、调度策略、调度器类】

事实上，struct task\_struct 要比我所列出的这8个部分复杂的多，在这里不做进一步的赘述了，有兴趣的可以参考以下资料

【资料1】<https://blog.csdn.net/Quinn0918/article/details/70148159>

【资料2】[https://blog.csdn.net/m0\\_74282605/article/details/130034516](https://blog.csdn.net/m0_74282605/article/details/130034516)

【资料3】<http://husharp.today/2020/12/06/task-struct-01/>

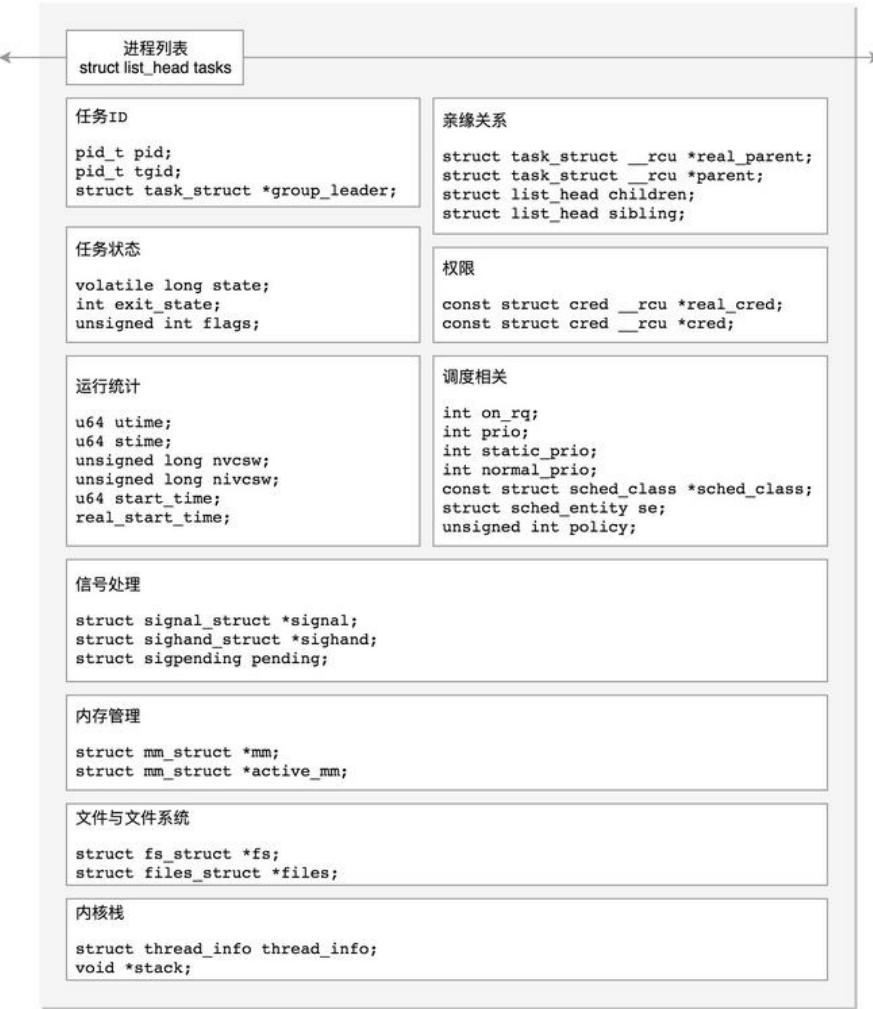


图2-25 资料3提供的图表

## Chapter3 进程调度

### 1、进程的调度流程

#### 1.1 再次提及进程调度

在Chatpter1中，我们曾提到过：由于一个CPU同时只能处理一个进程，而就绪队列里面往往有大量的进程等待CPU资源，因此需要设计进程调度动态地从众多的就绪进程中选择一个最合适的进程来占用CPU资源。

可能现在你还在因为 struct task\_struct 的庞大而困惑，但请把 Chapter2 中的一切都先放在一遍(只要明白它是一个代表进程的结构体就好了，其他的先不去管它)，你现在需要关注的是上一段中的这几个关键字：“CPU”、“就绪队列”、“进程”、“调度”。

事实上，这几个关键字就构成了进程调度的基本概念。它们分别对应着以下的结构体或函数：

- “队列”—— struct rq
- “进程”—— struct task\_struct
- “调度”—— schedule

#### 1.2 进程调度的流程

很多时候，我们往往因为过度地在意一些细节而使自己陷入困扰，比如你去linux内核里查看有关进程调度的代码时，刚开始就想让自己理清楚那几万行代码的作用是完全不现实的，反而会使自己的大脑一团糟。事实上，我们不用把进程的调度想得有多复杂，只需要先抓住主干再去延伸到分支便可。从整体分析进程的调度，其流程将会很简单：

(1) 当前CPU内有一个正在使用CPU资源的进程（记为current）以及一个等待使用CPU资源的一些进程组成的“就绪队列”。

(2) 硬件电路中有一个硬件定时器，它负责周期性的产生时钟中断（一般为10ms），我们称它为滴答定时器，可以认为，它就是操作系统的心脏。每当产生定时器中断的时候，CPU就会执行中断处理程序。

(3) 在滴答定时器的中断处理中，我们会通过 "schedule" 函数来判断 current 进程是否需要被抢占，若被抢占，"就绪队列" 中的哪一个进程将使用CPU资源。

以上就是最基本的进程调度的流程了，说白了就是在每次中断时调用 schedule 函数。

### 1.3 一些补充

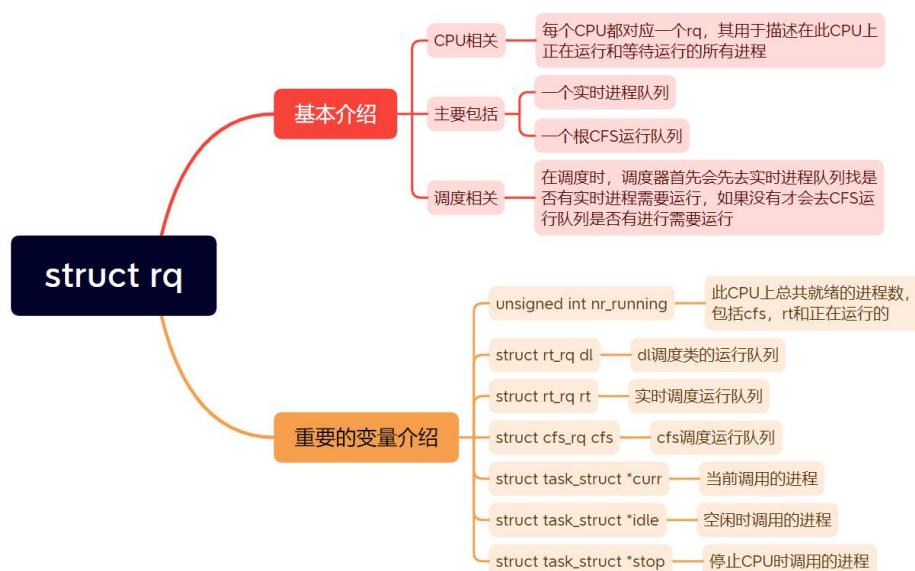
可能会有人会产生这样的疑问：好像上面所讲的只是提及了处于就绪队列的进程，这些进程都处于正在运行或等待运行的状态（即state = 0），那么其他状态的进程在哪里呢？

事实上，处在阻塞状态的进程并不在运行队列（就绪队列）中，而是处于一个等待队列中。这些进程都在等待某些事件的发生，例如等待I/O操作完成或等待信号量。只有当这些事件发生并完成时，内核会将进程从等待队列中唤醒并将其放回就绪队列中。

注：等待队列位于内核中。“内核”指的是一个提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件，内核常驻于内存，负责处理各种各样的核心任务，比如I/O、进程管理、内存管理等。（更多关于内核方面的知识请自行查阅相关资料）

## 2、struct rq

### 2.1 思维导图



### 2.2 源代码

```
struct rq {
    /* runqueue lock: */
    raw_spinlock_t      lock;

    /*
     * nr_running and cpu_load should be in the same cacheline because
     * remote CPUs use both these fields when doing load calculation.
     */
    unsigned int        nr_running;

    struct cfs_rq       cfs;
    struct rt_rq        rt;
    struct dl_rq        dl;

    struct task_struct  *curr;
    struct task_struct  *idle;
    struct task_struct  *stop;
```

图3-2 struct rq 重要部分源代码

### 2.3 详细介绍

每个CPU都会有属于自己的队列rq，该队列包含了所有正在CPU上进行的以及等待该CPU分配资源的进程。根据进程的性质（优先级等），又将这些进程进行分组。结构体struct rq中就记录了分组的情况。

dl是deadline调度类的运行队列，优先级很高；rt是实时调度运行队列，该队列中的进程都是实时进程，优先级较高[0-99]；cfs是cfs调度运行队列，该队列的进程都是普通进程，优先级较低[100-139]。

\*curr是指向当前使用CPU进程的指针，\*idle指向的是一个空闲的idle线程，Linux系统初始化时会为每个cpu创建一个idle线程，当没有其他进程需要运行的时候，便运行idle线程；\*stop指向的是stop调度类的进程。

### 3、schedule

#### 3.1 源代码

```
asm linkage __visible void __sched schedule(void)
{
    struct task_struct *tsk = current;
    sched_submit_work(tsk);
    do {
        preempt_disable();
        __schedule(false);
        sched_preempt_enable_no_resched();
    } while (need_resched());
    sched_update_worker(tsk);
}
```

图3-3 schedule 源代码

#### 3.2 代码分析

- (1) struct task\_struct \*tsk = current; // 定义一个指向当前进程的指针tsk
- (2) sched\_submit\_work(tsk); // 将当前进程放入调度队列
- (3) do { ... } while (need\_resched()); // 循环执行直到不需要调度
- (4) preempt\_disable(); // 禁止抢占
- (5) \_\_schedule(false); // 选择下一个进程运行，schedule 函数真正的核心
- (6) sched\_preempt\_enable\_no\_resched(); // 允许抢占但不进行调度
- (7) sched\_update\_worker(tsk); // 更新当前进程的调度状态，包括调度策略、优先级等信息

schedule() 函数只是个外层的封装，实际调用的还是\_\_schedule() 函数。\_\_schedule() 接受一个参数，该参数为 bool 型，false 表示非抢占，自愿调度，而 true 则相反。

通过源代码可以发现，在调用\_\_schedule() 前是需要关闭抢占的。实际上，在\_\_schedule中会去检查当前进程的抢占计数(位于schedule\_debug函数)，确保此次调度是在关闭抢占的情况下进行的，且不能是在中断或原子上下文发生的调用。至于关闭中断，该操作将在\_\_schedule内部完成。

在调度过程完成之后，即\_\_schedule 返回后，需要重新打开抢占。

### 4、\_\_schedule

#### 4.1 变量定义与前期准备

```
struct task_struct *prev, *next;
unsigned long *switch_count;
struct rq_flags rf;
struct rq *rq;
int cpu;

cpu = smp_processor_id();           // 获取当前CPU的ID
rq = cpu_rq(cpu);                 // 获取当前CPU的运行队列
prev = rq->curr;                  // 获取当前队列中正在运行的进程

schedule_debug(prev);              // BUG检查，判断当前 preempt count 是不是 1
local_irq_disable();               // 禁止本地CPU中断
```

图3-4 变量定义与前期准备

#### 4.2 当前运行的进程主动发起调度

```

    if (!preempt && prev->state) {
        if (unlikely(signal_pending_state(prev->state, prev))) {
            prev->state = TASK_RUNNING;
        } else {
            deactivate_task(rq, prev, DEQUEUE_SLEEP | DEQUEUE_NOCLOCK);
            prev->on_rq = 0;
        }
    }
}

```

图3-5 当前运行的进程主动发起调度

preempt 用于判断本次调度是否为抢占调度，如果发生了调度抢占 (preempt=1)，那么直接跳过，不参与判断，直接调用 pick\_next\_task。抢占调度通常都是处于运行态的任务发起的抢占调度。

如果本次调度不是抢占调度 (preempt=0)，并且该进程的state不等于 TASK\_RUNNING (0)，也就是不是运行态，处于其他状态，代表此次调度是该进程主动请求调度，主动调用了schedule函数，比如该进程进入了阻塞态。

如果该进程还有未处理信号，那就需要先处理信号。此时需要将当前进程的状态设置回 TASK\_RUNNING，这种情况下，当前进程会重新参与调度，有很大概率选取到的下一个进程依旧是当前进程，从而不执行实际的进程切换。

若该进程没有未处理的信号，由于进程不是运行态，那么就不能在CFS就绪队列中了，于是就调用 deactivate\_task 函数将陷入阻塞态的进程移除CFS就绪队列，并将进程调度实体的 on\_rq 成员置0，表示不在CFS就绪队列中了。

#### 4.3 选择下一个将要执行的任务

```

next = pick_next_task(rq, prev, &rf);
clear_tsk_need_resched(prev);
clear_preempt_need_resched();

```

图3-6 选择下一个将要执行的任务

选择下一个将要执行的任务的核心函数是pick\_next\_task()，该函数所对应的源代码如下所示：

```

static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /*
     * Optimization: we know that if all tasks are in the fair class we can
     * call that function directly, but only if the @prev task wasn't of a
     * higher scheduling class, because otherwise those loose the
     * opportunity to pull in more work from other CPUs.
     */
    if (likely((prev->sched_class == &idle_sched_class ||
               prev->sched_class == &fair_sched_class) &&
               rq->nr_running == rq->cfs.h_nr_running)) {
        p = fair_sched_class.pick_next_task(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto again;
        /* Assumes fair_sched_class->next == idle_sched_class */
        if (unlikely(!p))
            p = idle_sched_class.pick_next_task(rq, prev, rf);
    }
    return p;
}

```

图3-7 pick\_next\_task 源代码第一部分

linux 认为：当前进程属于 cfs 调度器类或 idle 调度器类是大概率事件。因此，pick\_next\_task 首先先进行判定当前进程是否属于 cfs 调度器类或 idle 调度器类。

若否，说明当前进程属于更高级别的调度器类，因此直接调过该部分代码；若是，则进而判断此时CPU运行队列中是否只存在属于cfs调度器类的进程 (nr\_running = 所有进程数量；cfs.h\_nr\_running = cfs调度器类的所有进程数量，若只存在属于cfs调度器类的进程，则 nr\_running = cfs.h\_nr\_running)。

若否，说明队列中存在比普通进程更高级别的进程，比如实时进程，此时同样直接调过该部分代码；若是，则说明下一个进程大概率属于cfs调度器类（只有当cfs.h\_nr\_running=0时才会进行空闲进程）。

满足了上述的两个条件后，选择下一个进程的方式就可以直接调用 cfs 调度器类的子函数 fair\_sched\_class.pick\_next\_task()。该函数将在 Chapter4 中进行详细讨论

假如 fair\_sched\_class.pick\_next\_task() 函数的返回值为 RETRY\_TASK，将指示调用者从高优先级调度类里面选取目标进程（该部分在again后面，所以使用goto again 直接跳到 again 后面），若返回 NULL，将指示调用者从低优先级调度类（即 idle）里面获取目标进程，所以使用 idle 调度器类的子函数 idle\_sched\_class.pick\_next\_task()选择下一个进程。

```

again:
    for_each_class(class) {
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
    /* The idle class should always have a runnable task: */
    BUG();
}

```

图3-8 pick\_next\_task 源代码第二部分

如果确定下一个进程不在 cfs\_rq 中选，那就需要依据优先级 (stop -> deadline -> realtime -> cfs\_rq -> idle) 对所有的调度器类进行遍历，找到一个进程之后就返回该进程。

#### 4.4 执行进程切换与收尾工作

```

if (likely(prev != next)) {
    rq->nr_switches++;

    RCU_INIT_POINTER(rq->curr, next);
    ++switch_count;

    trace_sched_switch(preempt, prev, next);

    rq = context_switch(rq, prev, next, &rf);
} else {
    rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
    rq_unlock_irq(rq, &rf);
}

```

图3-9 执行进程切换与收尾工作

这段代码是一个进程调度的另一个关键步骤。此时我们已经通过第三部分获得了下一进程，若当前进程与下一个进程不相同，就会增加 nr\_switches 计数器（上下文交换次数），然后调用 context\_switch() 函数进行上下文切换。否则，就会更新 clock\_update\_flags 标志位，并释放锁。

很明显，该步骤中最重要的一个函数就是 context\_switch() 函数，该函数的源代码如下，本文不对该代码进行进一步解释，有了解需要的请自行查阅相关资料。

```

1 static __always_inline struct rq *
2 context_switch(struct rq *rq, struct task_struct *prev,
3                 struct task_struct *next, struct rq_flags *rf)
4 {
5     prepare_task_switch(rq, prev, next); // 切换的准备工作，包括更新统计信息、设置 next->on_cpu 为 1 等。
6     arch_start_context_switch(prev); // arch 架构相关的上下文切换操作
7
8     // 根据 next 是否有用户空间，分别进行内核态到用户态和用户态到内核态的切换
9     if (!next->mm) { // to kernel
10         enter_lazy_tlb(prev->active_mm, next);
11         next->active_mm = prev->active_mm;
12         if (prev->mm) // from user
13             mmgrab(prev->active_mm);
14         else
15             prev->active_mm = NULL;
16     } else { // to user
17         membarrier_switch_mm(rq, prev->active_mm, next->mm);
18         switch_mm_irqs_off(prev->active_mm, next->mm, next);
19         if (!prev->mm) // from kernel
20             rq->prev_mm = prev->active_mm;
21         prev->active_mm = NULL;
22     }
23
24     rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP); // 更新时钟标志位
25     prepare_lock_switch(rq, next, rf); // 准备锁切换
26     switch_to(next, prev); // 切换进程
27     barrier(); // 内存屏障
28     finish_task_switch(prev); // 完成上下文切换，并返回新的运行队列。
29 }

```

图3-10 context\_switch 源代码

## 5、Chapter3 总结

Chapter3 主要介绍了进程调度的大体流程，详细介绍了与进程调度直接相关的运行队列（rq）以及调度函数（schedule 以及 \_\_schedule）。

相信阅读到这里的你已经对进程调度已经有了基本概念，明白了进程调度本质上是在每个滴答定时器中断时，使用调度函数在运行队列中寻找下一个使用CPU资源的进程并执行进程切换。在下一章中，我将详细介绍运行队列中的cfs队列以及与调度相关的调度器和其中的cfs调度器。

## Chapter4 CFS（完全公平调度）介绍

### 1、什么是CFS

#### 1.1、CFS基本原理概述

采用cfs策略的进程队列中的每一个进程都会设置一个虚拟时钟——virtual runtime(vruntime)。如果一个进程得以执行，随着执行时间的不断增长，其vruntime也将不断增大，没有得到执行的进程vruntime将保持不变。

而调度器将会选择最小的vruntime那个进程来执行。这就是所谓的“完全公平”。不同优先级的进程其vruntime增长速度不同，优先级高的进程vruntime增长得慢，所以它可能得到更多的运行机会。

#### 1.2、CFS相关概念

- schedule: 总的调度函数
- \_\_schedule: 调度函数中的核心部分，也是真正的调度函数
- sched\_class: 总的调度器类，fair\_sched\_class是它的一个实例化
- fair\_sched\_class: 采用CFS算法（策略）的调度器类
- struct rq: CPU运行（就绪）队列，每个rq队列都包含一个cfs\_rq
- struct cfs\_rq: 采用CFS算法（策略）的运行队列，里面所有的进程都是普通进程，优先级均在[100 - 139]
- struct task\_struct: 进程对应的结构体
- struct sched\_entity: 进程中对应于CFS算法（策略）的调度实体，保存了进程在被CFS调度器调度时的调度信息

#### 1.3、CFS算法设计核心

cfs调度器每次都会选择最小的vruntime那个进程作为下一个使用CPU资源的进程，那么vruntime该如何计算呢？又如何实现优先级高的进程vruntime增长得慢呢？

事实上，linux采用如下公式计算 vruntime，优先级越大意味着权重越大，因此采用倒数的方式可以实现优先级优先级高的进程 vruntime 增长得慢。

$$vruntime = realtime * \frac{NICE\_0\_LOAD}{weight}$$

其中，realtime 表示的是真实运行时间，NICE\_0\_LOAD 表示的是Nice 值为0对应的权重，weight 表示的是当前进程的权重。

### 2、struct cfs\_rq

#### 2.1、思维导图

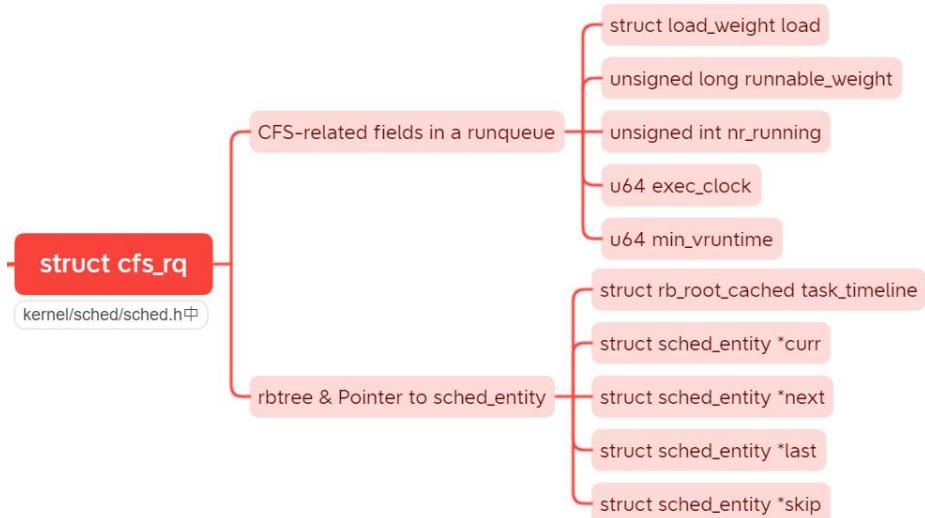


图4-1 cfs\_rq 思维导图

## 2.2、源代码

```

struct cfs_rq {
    struct load_weight      load;
    unsigned long           runnable_weight;
    unsigned int            nr_running;
    unsigned int            h_nr_running;      /* SCHED_{NORMAL,BATCH,IDLE} */

    u64                  exec_clock;
    u64                  min_vruntime;

    struct rb_root_cached   tasks_timeline;

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    struct sched_entity     *curr;
    struct sched_entity     *next;
    struct sched_entity     *last;
    struct sched_entity     *skip;
}

```

图3-10 cfs\_rq 源代码

## 2.3、详细介绍

- load: cfs\_rq上所有调度实体的负载权重之和
- runnable\_weight: cfs\_rq上所有调度实体的运行权重之和

注1: 对应单个调度实体, task se 的 load weight 和 runnable weight 是相等的, 但是对于调度组 group se 而言, 这两个值是不同的。

注2: runnable\_weight的计算涉及到了调度组的概念以及PELT算法, 本文暂时不作进一步说明, 有需要的请参考下方链接

参考: [https://blog.csdn.net/Rong\\_Toa/article/details/108598418](https://blog.csdn.net/Rong_Toa/article/details/108598418)

- nr\_running: cfs\_rq中第一层的调度实体的数量, 也就是红黑树中的节点数。这些调度实体可能是进程或者调度组, 但是不包括调度组内的子调度实体
- h\_nr\_running: cfs\_rq中所有调度实体的数量, 也就是红黑树中的节点数加上每个调度组节点对应的子cfs\_rq中的节点数。

注1: 假如 cf\_rq 中有1个调度组A和2个调度实体, 其中调度组A包含了3个调度实体, 则nr\_running = 3, h\_nr\_running = 5

注2: 可以根据nr\_running 与 h\_nr\_running 是否相等来判断 cfs\_rq 中是否有调度组

注3: cfs\_rq中的调度组是通过task\_group结构表示的, 调度组的实质是: 调度时候不再以进程作为调度实体, 而是以进程组作为调度实体。虽然task group作为一个调度实体来竞争CPU资源, 但是task group是一组task se或者group se (task group可以嵌套)的集合, 不可能在一个CPU上进行调度, 因此task group的调度实际上在各个CPU上都会发生,

- exec\_clock: CFS就绪队列的总运行时间
- min\_vruntime: CFS就绪队列的红黑树中最左边节点的 vruntime

- task\_timeline: 红黑树，用于链接调度实体，以调度实体的vruntime 作为关键字排序
- \*curr: 指向当前正在运行的调度实体的指针
- \*next: 指向下一个要运行的调度实体的指针
- \*last: 指向上一个运行过的调度实体的指针
- \*skip: 指向跳过的调度实体的指针

### 3、struct sched\_entity

#### 3.1、思维导图

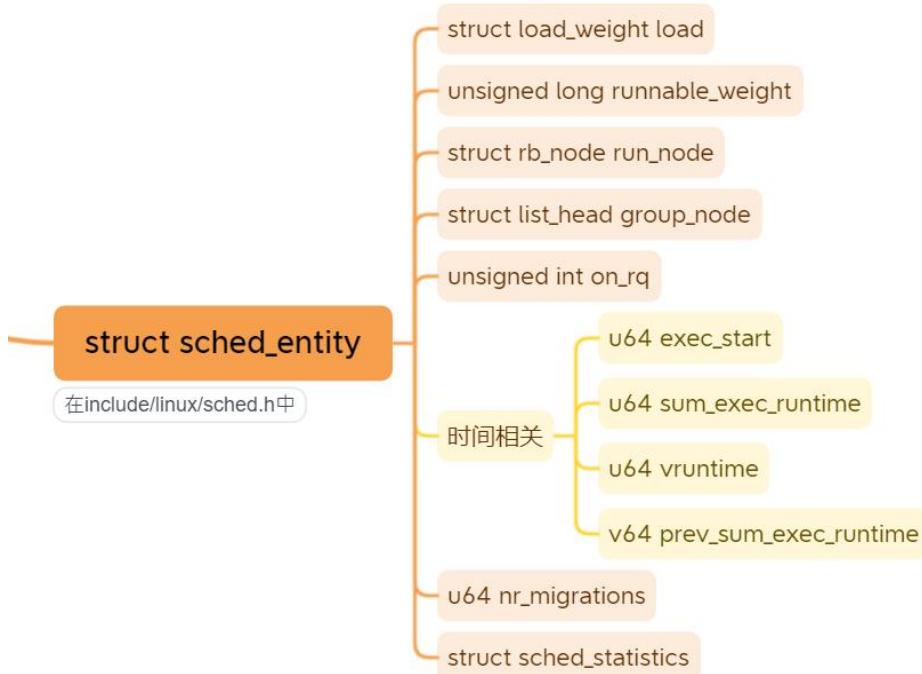


图4-3 struct sched\_entity 思维导图

#### 3.2、源代码

```

struct sched_entity {
    /* For load-balancing: */
    struct load_weight           load;
    unsigned long                runnable_weight;
    struct rb_node               run_node;
    struct list_head              group_node;
    unsigned int                 on_rq;

    u64                          exec_start;
    u64                          sum_exec_runtime;
    u64                          vruntime;
    v64                          prev_sum_exec_runtime;

    u64                          nr_migrations;

    struct sched_statistics       statistics;
}
  
```

图4-4 cfs\_rq 源代码

#### 3.3、详细介绍

- load: 调度实体的负载权重
- runnable\_weight: 调度实体的运行权重
- run\_node: 红黑树的数据节点，使用该rb\_node将当前节点挂到红黑树上面，红黑树用于存储就绪队列中的调度实体，按照vruntime排序。
- group\_node: 链表节点，被链接到percpu的rq->cfs\_tasks上，在做CPU之间的负载均衡时，就会从该链表上选出group\_node节点作为迁移进程（进一步了解请查阅2.3中的链接）

- `on_rq`: 标志位，代表当前调度实体是否在就绪队列上
- `exec_start`: 当前实体上次被调度执行的时间
- `sum_exec_runtime`: 当前实体总执行时间
- `vruntime`: 调度实体的虚拟时间
- `prev_sum_exec_runtime`: 截止到上次统计，进程执行的时间，通常，通过`sum_exec_runtime - prev_sum_exec_runtime`来统计进程本次在CPU上执行了多长时间，以执行某些时间相关的操作

## 4. struct sched\_class

### 4.1、思维导图

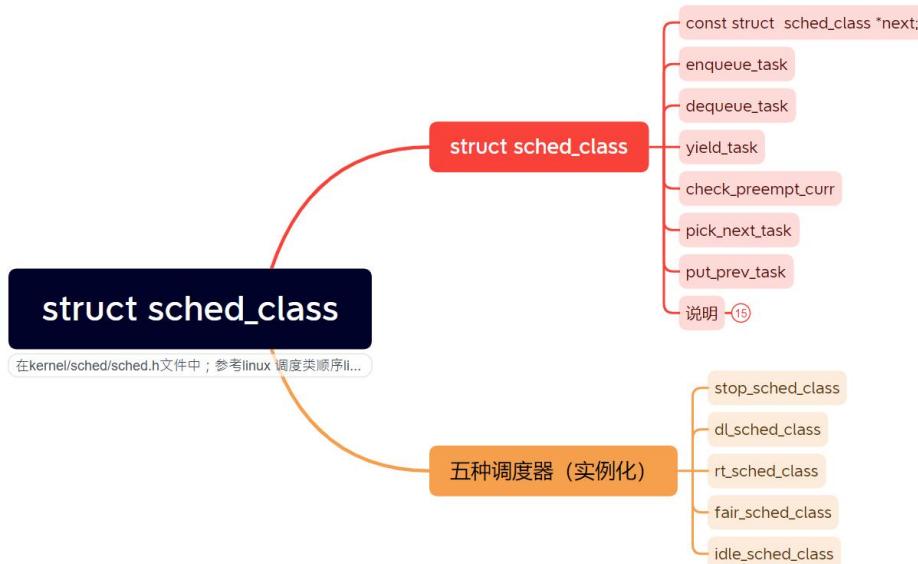


图4-5 struct sched\_entity 思维导图

### 4.2、源代码

```

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 (*yield_task) (struct rq *rq);
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p, bool preempt);

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);

    /*
     * It is the responsibility of the pick_next_task() method that will
     * return the next task to call put_prev_task() on the @prev task or
     * something equivalent.
     *
     * May return RETRY_TASK when it finds a higher prio class has runnable
     * tasks.
     */
    struct task_struct * (*pick_next_task)(struct rq *rq,
                                         struct task_struct *prev,
                                         struct rq_flags *rf);
    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
}
  
```

图4-6 struct sched\_entity 源代码

### 4.3、详细介绍

- `const struct sched_class *next`: 指向下一个比其等级低的class。其顺序依次为stop, deadline, real time, fair, idle。
- `enqueue_task`: 将一个task插入到相应的runqueue里面
- `dequeue_task`: 将一个task从runqueue里面删除
- `yield_task`: 放弃CPU执行权限，将任务从执行队列中出队，然后再放入到队列末尾。
- `check_preempt_curr`: 用于检查当前进程是否可以被新的进程抢占
- `pick_next_task`: 选择下一个运行的进程
- `put_prev_task`: 将进程放回到运行队列当中

注：sched\_class是Linux内核为不同调度策略定义的调度类，是对调度器公共部分的抽象。它是一个可扩展的调度模块层次结构，可用于实现不同的调度策略。

## 5、struct sched\_class fair\_sched\_class

### 5.1、实例化源代码

```
'  
const struct sched_class fair_sched_class = {  
    .next              = &idle_sched_class,  
    .enqueue_task      = enqueue_task_fair,  
    .dequeue_task      = dequeue_task_fair,  
    .yield_task        = yield_task_fair,  
    .yield_to_task     = yield_to_task_fair,  
  
    .check_preempt_curr = check_preempt_wakeup,  
  
    .pick_next_task    = pick_next_task_fair,  
    .put_prev_task     = put_prev_task_fair,
```

图4-7 struct fair\_sched\_class 源代码

fair\_sched\_class 是 sched\_class 类其中一个实例化，整个fair.c文件都是关于 fair\_sched\_class 类的函数的定义。

关于 enqueue\_task、dequeue\_task、yield\_task、check\_preempt\_curr、put\_prev\_task 这五个函数不作进一步介绍，可以参考其字面意思来理解。

重点介绍一下 pick\_next\_entity 函数，因为该函数是调度函数 schedule 的重要组成部分。

pick\_next\_entity 被 pick\_next\_task\_fair 函数封装

### 5.2、pick\_next\_task\_fair 源代码

```
1 static struct task_struct *  
2 pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)  
3 {  
4     struct cfs_rq *cfs_rq = &rq->cfs;  
5     struct sched_entity *se;  
6     struct task_struct *p;  
7     int new_tasks;  
8     unsigned long time;  
9  
10    again:  
11        if (!cfs_rq->nr_running)  
12            goto idle;  
13        put_prev_task(rq, prev);  
14        do {  
15            se = pick_next_entity(cfs_rq, NULL);  
16            set_next_entity(cfs_rq, se);  
17            cfs_rq = group_cfs_rq(se);  
18        } while (cfs_rq);  
19        p = task_of(se);  
20
```

```
21    done: __maybe_unused;  
22    if (hrtick_enabled(rq))  
23        hrtick_start_fair(rq, p);  
24    return p;  
25  
26    idle:  
27        time = schedstat_start_time();  
28        rq_idle_stamp_update(rq);  
29        new_tasks = idle_balance(rq, rf);  
30        if (new_tasks == 0)  
31            new_tasks = try_steal(rq, rf);  
32        schedstat_end_time(rq, time);  
33        if (new_tasks)  
34            rq_idle_stamp_clear(rq);  
35        if (new_tasks < 0)  
36            return RETRY_TASK;  
37        if (new_tasks > 0)  
38            goto again;  
39        return NULL;  
40 }
```

图4-8 pick\_next\_task\_fair 源代码

### 5.3、pick\_next\_task\_fair代码逻辑

- (1) 如果就绪队列为空，就跳转到idle标签。
- (2) 否则，把之前运行的任务放回就绪队列。
- (3) 然后，用真正的选择函数 pick\_next\_entity () 从就绪队列中选择最优先的调度实体 (sched\_entity)，可能是一个单个任务或者一个任务组。
- (4) 如果选择的调度实体是一个任务组，就递归地进入它对应的子就绪队列，直到找到一个单个任务。
- (5) 最后，返回找到的任务。

### 5.3、pick\_next\_entity源代码以及注解

只需要理解第一部分即可，后两部分可以忽略。

```
1 static struct sched_entity *
2 pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
3 {
4     // 找到运行队列中最左边的调度实体，赋值给left变量
5     struct sched_entity *left = __pick_first_entity(cfs_rq);
6     struct sched_entity *se;
7
8     // 如果curr不为空，也就是说有一个正在运行的调度实体，
9     // 那么检查它是否比left还要早，如果是，就把left更新为curr。
10    if (!left || (curr && entity_before(curr, left)))
11        left = curr;
12
13    // 把left赋值给se变量，表示理想情况下要运行的调度实体
14    se = left;
```

图4-9 pick\_next\_entity 源代码（一）

```
16 // 如果se是运行队列中被跳过执行的调度实体，则需要进行以下过程
17 // 下面代码的目的是避免运行跳过伙伴,
18 // 如果可以在不太不公平的情况下运行其他调度实体
19 if (cfs_rq->skip == se) {
20     struct sched_entity *second;
21     // 如果se等于curr，也就是说跳过伙伴是当前正在运行的调度实体,
22     // 那么把运行队列中最左边的调度实体赋值给second
23     if (se == curr) {
24         second = __pick_first_entity(cfs_rq);
25     } else {
26         // 如果se不等于curr，也就是说跳过伙伴是运行队列中的一个调度实体,
27         // 那么把se的下一个调度实体赋值给second。
28         second = __pick_next_entity(se);
29         // 如果second不存在，或者curr存在并且比second还要早,
30         // 那么把curr赋值给second
31         if (!second || (curr && entity_before(curr, second)))
32             second = curr;
33     }
34
35     // 最后，如果second存在,
36     // 并且它和left之间的不公平程度小于1（也就是说它没有被太多地延迟）,
37     // 那么把se更新为second
38     if (second && wakeup_preempt_entity(second, left) < 1)
39         se = second;
40 }
```

图4-10 pick\_next\_entity 源代码（二）

```

42 // 下面的代码是为了优先运行最后伙伴和下一个伙伴,
43 // 尝试返回CPU给一个被抢占的任务或者一个被唤醒的任务。
44
45 // 检查运行队列中是否有一个最后伙伴,
46 // 如果有, 并且它和left之间的不公平程度小于1
47 // (也就是说它没有被太多地延迟), 那么把se更新为最后伙伴
48 if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1)
49     se = cfs_rq->last;
50
51 // 检查运行队列中是否有一个下一个伙伴,
52 // 如果有, 并且它和left之间的不公平程度小于1,
53 // 那么把se更新为下一个伙伴
54 if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1)
55     se = cfs_rq->next;
56
57 // 清除运行队列中的所有伙伴标记
58 clear_buddies(cfs_rq, se);
59
60 return se;
61 }

```

图4-11 pick\_next\_entity 源代码 (三)

## 6、Chapter4 总结 & 完整思维导图

Chapter4主要介绍了什么是完全公平调度算法（策略）。CFS算法引入了虚拟时间vruntime的概念，每次调度都会选择vruntime值最小的调度实体。CPU内采用CFS调度策略的进程都位于一个cfs\_rq类型的队列中，该队列采用红黑树的方式保存了所有的调度实体，并以vruntime值的大小为键值。红黑树的最左端就是vruntime最小值对应的调度实体。红黑树的具体实现比较复杂，需要进一步了解的可以参考以下链接

参考1: <https://blog.csdn.net/xiaofeng10330111/article/details/106080394>

参考2: <https://blog.csdn.net/cy973071263/article/details/122543826>

现在，本文已经将进程调度的整体流程大体上讲解完毕，为了帮助理解，以下是进程调度的完整思维导图，帮助大家理解从 Chapter1 到 Chapter4 的所有知识点。

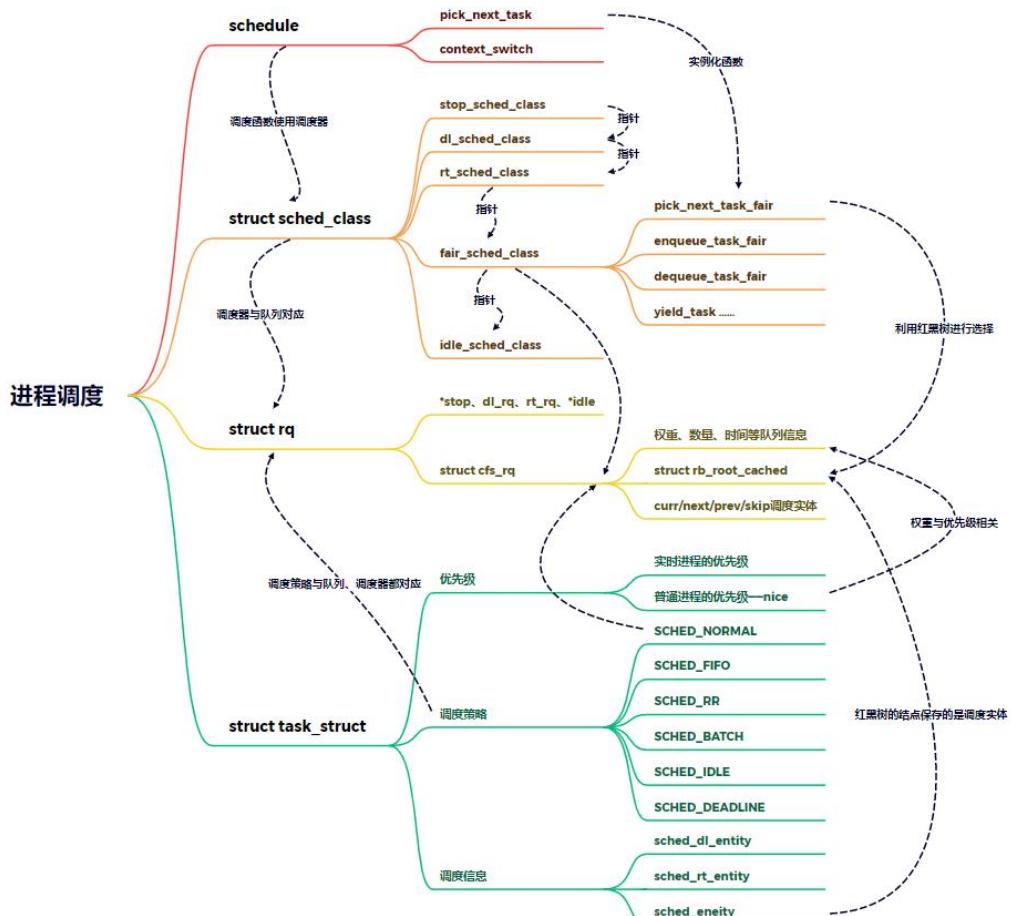


图4-12 完整思维导图

## Chapter5 选做部分

修改core.c中—\_\_schedule()函数，修改部分的代码如下图所示

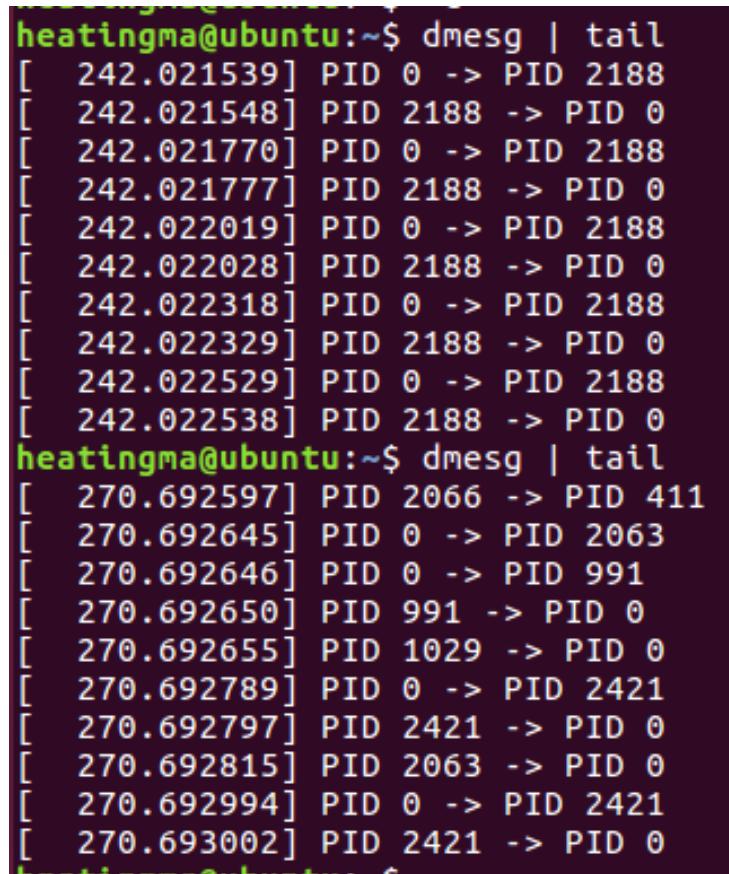
```
if (likely(prev != next)) {  
    next -> last_task_pid = prev -> pid;  
    printk(KERN_DEBUG "PID %d -> PID %d\n", prev->pid,next->pid);  
    rq->n_r_switches++;  
}
```

图5-1 修改的部分

printk()函数用于在终端输出进程切换前后的PID值，KERN\_DEBUG表示该printk语句优先级最低，因此不会在开机时进行输出，一般情况下也不会在终端输出，只有在终端输入dmesg | tail时才会呈现。

printk()函数用于在终端输出进程切换前后的PID值，KERN\_DEBUG表示该printk语句优先级最低，因此不会在开机时进行输出，一般情况下也不会在终端输出，只有在终端输入dmesg | tail时才会呈现。

实验结果如下：



The terminal window shows two sessions of dmesg | tail output. The first session shows PID 2188 switching between tasks with PID 0. The second session shows PID 411 switching between tasks with PIDs 0, 991, 1029, 2421, and 2063.

```
heatingma@ubuntu:~$ dmesg | tail  
[ 242.021539] PID 0 -> PID 2188  
[ 242.021548] PID 2188 -> PID 0  
[ 242.021770] PID 0 -> PID 2188  
[ 242.021777] PID 2188 -> PID 0  
[ 242.022019] PID 0 -> PID 2188  
[ 242.022028] PID 2188 -> PID 0  
[ 242.022318] PID 0 -> PID 2188  
[ 242.022329] PID 2188 -> PID 0  
[ 242.022529] PID 0 -> PID 2188  
[ 242.022538] PID 2188 -> PID 0  
  
heatingma@ubuntu:~$ dmesg | tail  
[ 270.692597] PID 2066 -> PID 411  
[ 270.692645] PID 0 -> PID 2063  
[ 270.692646] PID 0 -> PID 991  
[ 270.692650] PID 991 -> PID 0  
[ 270.692655] PID 1029 -> PID 0  
[ 270.692789] PID 0 -> PID 2421  
[ 270.692797] PID 2421 -> PID 0  
[ 270.692815] PID 2063 -> PID 0  
[ 270.692994] PID 0 -> PID 2421  
[ 270.693002] PID 2421 -> PID 0
```

图5-2 实验结果

注：这种方式对虚拟机内存要求较高，经测试，内存需要大于等于10GB（比较保险）