

12 | 进程数据结构（上）：项目多了就需要项目管理系统

2019-04-22 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 12:08 大小 11.13M



前面两节，我们讲了如何使用系统调用，创建进程和线程。你是不是觉得进程和线程管理，还挺复杂的呢？如此复杂的体系，在内核里面应该如何管理呢？

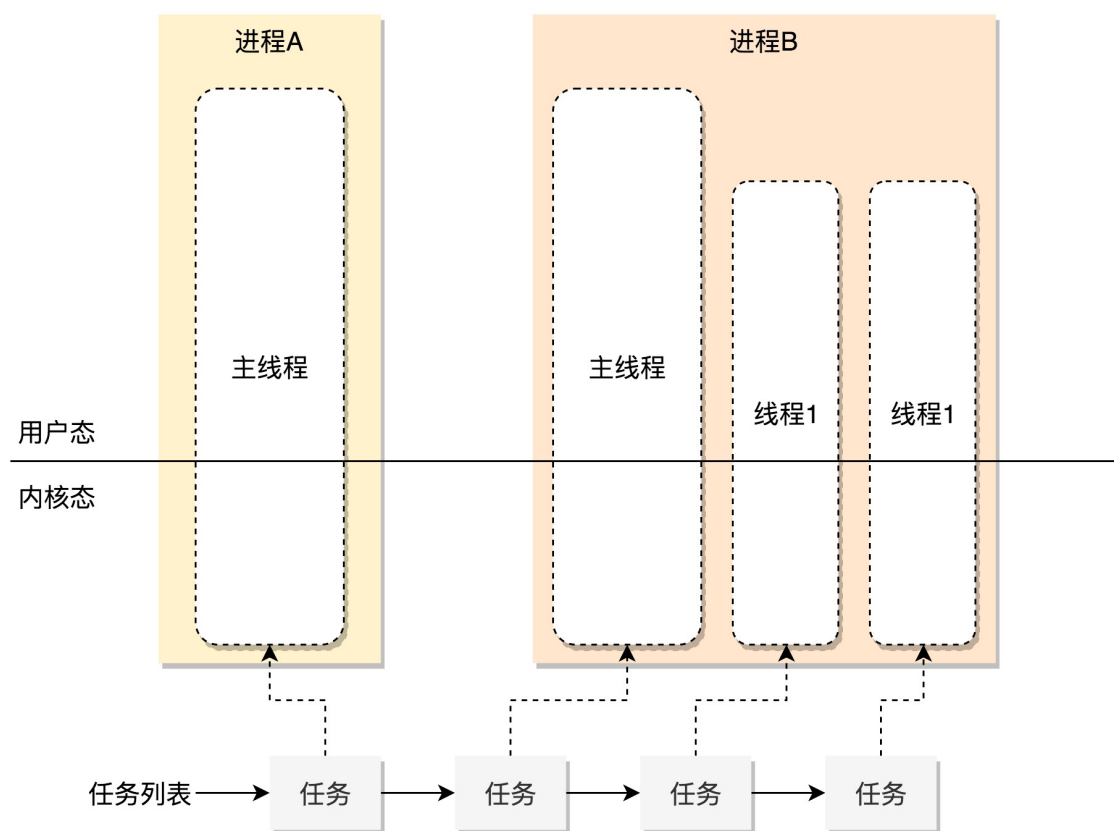
有的进程只有一个线程，有的进程有多个线程，它们都需要由内核分配 CPU 来干活。可是 CPU 总共就这么几个，应该怎么管理，怎么调度呢？你是老板，这个事儿得你来操心。

首先，我们得明确，公司的项目售前售后人员，接来了这么多的项目，这是个好事儿。这些项目都通过办事大厅立了项的，有的需要整个项目组一起开发，有的是一个项目组分成多个小组并行开发。无论哪种模式，到你这个老板这里，都需要有一个项目管理体系，进行统一排期、统一管理和统一协调。这样，你才能对公司的业务了如指掌。

那具体应该怎么做呢？还记得咱们平时开发的时候，用的项目管理软件 Jira 吧？它的办法对我们来讲，就很有参考意义。


我们这么来看，其实，无论是一个大的项目组一起完成一个大的功能（单体应用模式），还是把一个大的功能拆成小的功能并行开发（微服务模式），这些都是开发组根据客户的需求来定的，项目经理没办法决定，但是从项目经理的角度来看，这些都是任务，需要同样关注进度、协调资源等等。

同样在 Linux 里面，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。这个结构非常复杂，但你也不用怕，我们慢慢来解析。



接下来，我们沿着建立项目管理体系的思路，设想一下，Linux 的任务管理都应该干些啥？

首先，所有执行的项目应该有个项目列表吧，所以 Linux 内核也应该先弄一个**链表**，将所有的 task_struct 串起来。

 复制代码


```
1 struct list_head          tasks;
```

接下来，我们来看每一个任务都应该包含哪些字段。

任务 ID

每一个任务都应该有一个 ID，作为这个任务的唯一标识。到时候排期啊、下发任务啊等等，都按 ID 来，就不会产生歧义。

task_struct 里面涉及任务 ID 的，有下面几个：

 复制代码

```
1 pid_t pid;
2 pid_t tgid;
3 struct task_struct *group_leader;
```

你可能觉得奇怪，既然是 ID，有一个就足以做唯一标识了，这个怎么看起来这么麻烦？这是因为，上面的进程和线程到了内核这里，统一变成了任务，这就带来两个问题。

第一个问题是，**任务展示**。

啥是任务展示呢？这么说吧，你作为老板，想了解的肯定是，公司都接了哪些项目，每个项目多少营收。什么项目执行是不是分了小组，每个小组是啥情况，这些细节，项目经理没必要全都展示给你看。

前面我们学习命令行的时候，知道 ps 命令可以展示出所有的进程。但是如果你是这个命令的实现者，到了内核，按照上面的任务列表把这些命令都显示出来，把所有的线程全都平摊开来显示给用户。用户肯定觉得既复杂又困惑。复杂在于，列表这么长；困惑在于，里面出现了很多并不是自己创建的线程。

第二个问题是，**给任务下发指令**。

如果客户突然给项目组提个新的需求，比如说，有的客户觉得项目已经完成，可以终止；再比如说，有的客户觉得项目做到一半没必要再进行下去了，可以中止，这时候应该给谁发指令？当然应该给整个项目组，而不是某个小组。我们不能让客户看到，不同的小组口径不一致。这就好比说，中止项目的指令到达一个小组，这个小组很开心就去休息了，同一个项目组的其他小组还干的热火朝天的。

Linux 也一样，前面我们学习命令行的时候，知道可以通过 kill 来给进程发信号，通知进程退出。如果发给了其中一个线程，我们就不能只退出这个线程，而是应该退出整个进程。当然，有时候，我们希望只给某个线程发信号。

所以在内核中，它们虽然都是任务，但是应该加以区分。其中，pid 是 process id，tgid 是 thread group ID。


任何一个进程，如果只有主线程，那 pid 是自己，tgid 是自己，group_leader 指向的还是自己。

但是，如果一个进程创建其他线程，那就会有所变化了。线程有自己的 pid，tgid 就是进程的主线程的 pid，group_leader 指向的就是进程的主线程。

好了，有了 tgid，我们就知道 task_struct 代表的是一个进程还是代表一个线程了。

信号处理

这里既然提到了下发指令的问题，我就顺便提一下 task_struct 里面关于信号处理的字段。

 复制代码

```
1 /* Signal handlers: */
2 struct signal_struct      *signal;
3 struct sighand_struct     *sighand;
4 sigset_t                  blocked;
5 sigset_t                  real_blocked;
6 sigset_t                  saved_sigmask;
7 struct sigpending         pending;
8 unsigned long             sas_ss_sp;
9 size_t                    sas_ss_size;
10 unsigned int              sas_ss_flags;
```

这里定义了哪些信号被阻塞暂不处理（blocked），哪些信号尚等待处理（pending），哪些信号正在通过信号处理函数进行处理（sighand）。处理的结果可以是忽略，可以是结束进程等等。

信号处理函数默认使用用户态的函数栈，当然也可以开辟新的栈专门用于信号处理，这就是 `sas_ss_xxx` 这三个变量的作用。

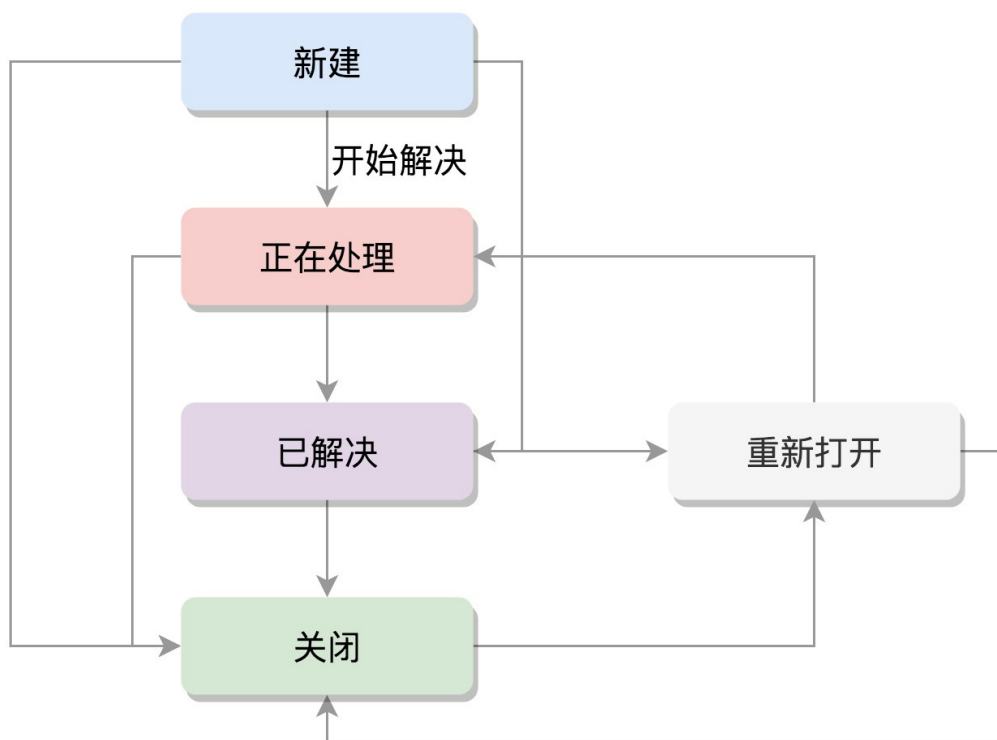
上面我说了下发信号的时候，需要区分进程和线程。从这里我们其实也能看出一些端倪。

`task_struct` 里面有一个 `struct sigpending pending`。如果我们进入 `struct signal_struct *signal` 去看的话，还有一个 `struct sigpending shared_pending`。它们一个是本任务的，一个是线程组共享的。


关于信号，你暂时了解到这里就够用了，后面我们会有单独的章节进行解读。

任务状态

作为一个项目经理，另外一个需要关注的是项目当前的状态。例如，在 Jira 里面，任务的运行就可以分成下面的状态。




在 task_struct 里面，涉及任务状态的是下面这几个变量：

 复制代码

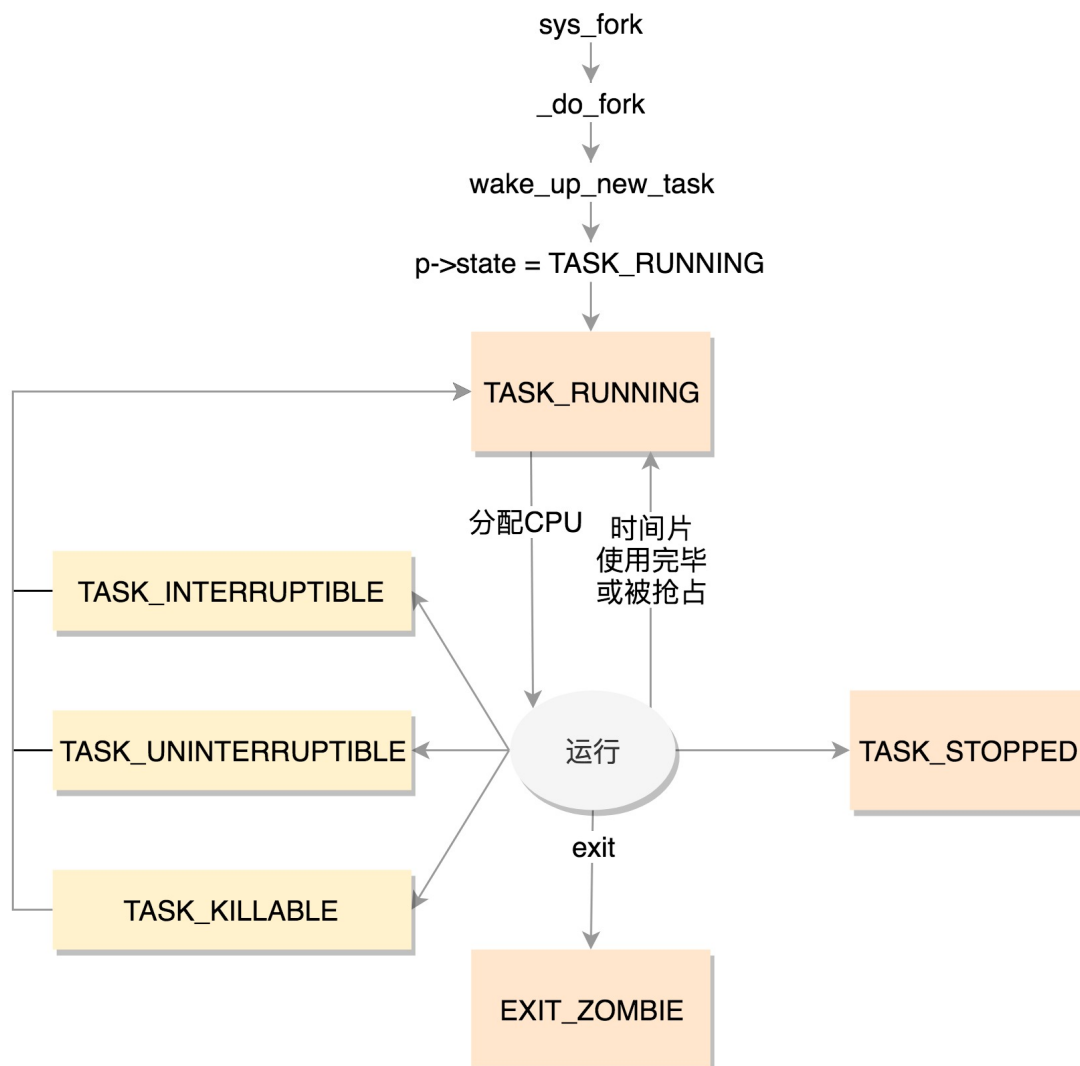
```
1 volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
2 int exit_state;
3 unsigned int flags;
```

state（状态）可以取的值定义在 include/linux/sched.h 头文件中。

 复制代码

```
1 /* Used in tsk->state: */
2 #define TASK_RUNNING          0
3 #define TASK_INTERRUPTIBLE    1
4 #define TASK_UNINTERRUPTIBLE  2
5 #define __TASK_STOPPED        4
6 #define __TASK_TRACED         8
7 /* Used in tsk->exit_state: */
8 #define EXIT_DEAD             16
9 #define EXIT_ZOMBIE           32
10 #define EXIT_TRACE             (EXIT_ZOMBIE | EXIT_DEAD)
11 /* Used in tsk->state again: */
12 #define TASK_DEAD              64
13 #define TASK_WAKEKILL          128
14 #define TASK_WAKING            256
15 #define TASK_PARKED            512
16 #define TASK_NOLOAD            1024
17 #define TASK_NEW               2048
18 #define TASK_STATE_MAX        4096
```

从定义的数值很容易看出来，flags 是通过 bitset 的方式设置的也就是说，当前是什么状态，哪一位就置一。



`TASK_RUNNING` 并不是说进程正在运行，而是表示进程在时刻准备运行的状态。当处于这个状态的进程获得时间片的时候，就是在运行中；如果没有获得时间片，就说明它被其他进程抢占了，在等待再次分配时间片。

在运行中的进程，一旦要进行一些 I/O 操作，需要等待 I/O 完毕，这个时候会释放 CPU，进入睡眠状态。

在 Linux 中，有两种睡眠状态。

一种是 **`TASK_INTERRUPTIBLE`**，**可中断的睡眠状态**。这是一种浅睡眠的状态，也就是说，虽然在睡眠，等待 I/O 完成，但是这个时候一个信号来的时候，进程还是要被唤醒。


只不过唤醒后，不是继续刚才的操作，而是进行信号处理。当然程序员可以根据自己的意愿，来写信号处理函数，例如收到某些信号，就放弃等待这个 I/O 操作完成，直接退出，也可也收到某些信息，继续等待。

另一种睡眠是 **TASK_UNINTERRUPTIBLE**，**不可中断的睡眠状态**。这是一种深度睡眠状态，不可被信号唤醒，只能死等 I/O 操作完成。一旦 I/O 操作因为特殊原因不能完成，这个时候，谁也叫不醒这个进程了。你可能会说，我 kill 它呢？别忘了，kill 本身也是一个信号，既然这个状态不可被信号唤醒，kill 信号也被忽略了。除非重启电脑，没有其他办法。

因此，这其实是一个比较危险的事情，除非程序员极其有把握，不然还是不要设置成 TASK_UNINTERRUPTIBLE。

于是，我们就有了一种新的进程睡眠状态，**TASK_KILLABLE**，**可以终止的新睡眠状态**。进程处于这种状态中，它的运行原理类似 TASK_UNINTERRUPTIBLE，只不过可以响应致命信号。

从定义可以看出，TASK_WAKEKILL 用于在接收到致命信号时唤醒进程，而 TASK_KILLABLE 相当于这两位都设置了。

 复制代码

```
1 #define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
```

TASK_STOPPED 是在进程接收到 SIGSTOP、SIGTTIN、SIGTSTP 或者 SIGTTOU 信号之后进入该状态。


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程进入该状态。

一旦一个进程要结束，先进入的是 EXIT_ZOMBIE 状态，但是这个时候它的父进程还没有使用 wait() 等系统调用来获知它的终止信息，此时进程就成了僵尸进程。

EXIT_DEAD 是进程的最终状态。

EXIT_ZOMBIE 和 EXIT_DEAD 也可以用于 exit_state。

上面的进程状态和进程的运行、调度有关系，还有其他的一些状态，我们称为**标志**。放在 flags 字段中，这些字段都被定义称为**宏**，以 PF 开头。我这里举几个例子。

 复制代码

```
1 #define PF_EXITING          0x00000004
2 #define PF_VCPU             0x00000010
3 #define PF_FORKNOEXEC       0x00000040
```


PF_EXITING表示正在退出。当有这个 flag 的时候，在函数 find_alive_thread 中，找活着的线程，遇到有这个 flag 的，就直接跳过。

PF_VCPU表示进程运行在虚拟 CPU 上。在函数 account_system_time 中，统计进程的系统运行时间，如果有这个 flag，就调用 account_guest_time，按照客户机的时间进行统计。

PF_FORKNOEXEC表示 fork 完了，还没有 exec。在 _do_fork 函数里面调用 copy_process，这个时候把 flag 设置为 PF_FORKNOEXEC。当 exec 中调用了 load_elf_binary 的时候，又把这个 flag 去掉。

进程调度

进程的状态切换往往涉及调度，下面这些字段都是用于调度的。为了让你理解 task_struct 进程管理的全貌，我先在这里列一下，咱们后面会有单独的章节讲解，这里你只要大概看一下里面的注释就好了。

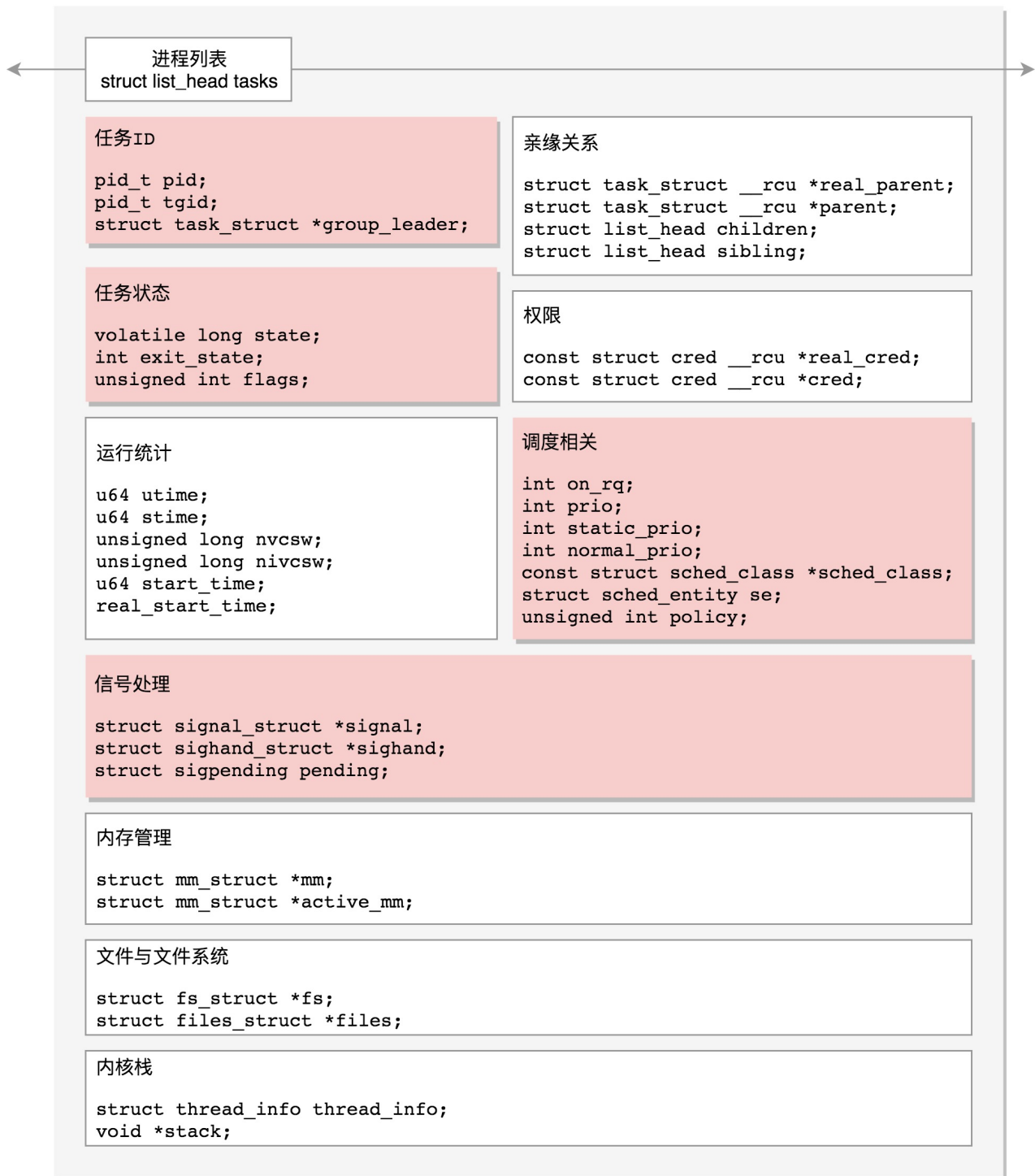
 复制代码

```
1 // 是否在运行队列上
2 int                                on_rq;
3 // 优先级
4 int                                prio;
5 int                                static_prio;
6 int                                normal_prio;
7 unsigned int                       rt_priority;
8 // 调度器类
9 const struct sched_class            *sched_class;
10 // 调度实体
11 struct sched_entity                se;
12 struct sched_rt_entity              rt;
13 struct sched_dl_entity              dl;
```

```
14 // 调度策略
15 unsigned int                policy;
16 // 可以使用哪些 CPU
17 int                        nr_cpus_allowed;
18 cpumask_t                  cpus_allowed;
19 struct sched_info          sched_info;
```

总结时刻

这一节，我们讲述了进程管理复杂的数据结构，我还是画一个图总结一下。这个图是进程管理 `task_struct` 的结构图。其中红色的部分是今天讲的部分，你可以对着这张图说出它们的含义。



课堂练习

这一节我们讲了任务的状态，你可以试着在代码里面搜索一下这些状态改变的地方是哪个函数，是什么时机，从而进一步理解任务的概念。

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 线程：如何让复杂的项目并行执行？

下一篇 13 | 进程数据结构（中）：项目多了就需要项目管理系统

精选留言 (24)

写留言



why

2019-04-22

14

- 内核中进程, 线程统一为任务, 由 `task_struct` 表示
- 通过链表串起 `task_struct`
- `task_struct` 中包含: 任务ID; 任务状态; 信号处理相关字段; 调度相关字段; 亲缘关系; 权限相关; 运行统计; 内存管理; 文件与文件系统; 内核栈;
- 任务 ID; 包含 `pid`, `tgid` 和 `*group_leader...`

展开



Egos

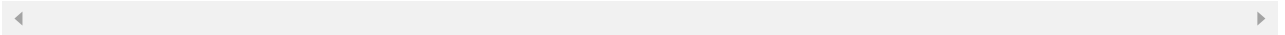
2019-04-22

4

看文章理解的`task_struct` 是Thread 的一个链表？

展开 ▾

作者回复: 进程和线程在一起的链表



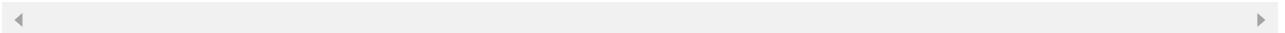
中翅Lzc

2019-04-22

👍 2

如果进程创建其他多个线程，那么tpid就是主线程id，pid就是其他线程id了，两者肯定不相等啊

作者回复: 对的



唐稳

2019-05-01

👍 1

介绍的很详细，赞一个。

有个问题一直纠结，信号处理函数到底是在哪个线程中运行的？



积微致知

2019-04-23

👍 1

老师好，有个疑惑所有的task_struct为什么用链表串联起来而不是用数组？
数组在物理空间上必须要连续，而链表物理空间上可以不连续。



Dracula

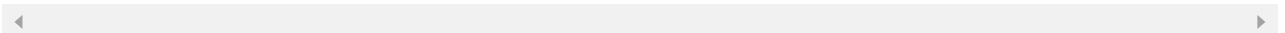
2019-04-22

👍 1

老师好，有个疑惑所有的task_struct为什么用链表串联起来而不是用数组

展开 ▾

作者回复: 很多插入和删除



第十人

2019-04-22

👍 1

tgid和threadleader都是进程的主线程，那这两个参数不就重复了么？有其他的含义么？

作者回复: 一个是id，一个是指针，只知道ID，不得一个个找么



勤劳的小胖...

2019-05-11



应该是一个值，一个是指针用于快速访问吧。。

展开 ∨

作者回复: 对



勤劳的小胖...

2019-05-11



任何一个进程，如果只有主线程，那 pid 是自己，tgid 是自己，group_leader 指向的还是自己。

但是有多个线程就不一样了，pid是这个子线程，tgid和group_leader都是指向主线程。

...

展开 ∨

作者回复: 一个是id，一个是地址，有地址就能直接找到了



Milittle

2019-05-09



要是老师把对应源码位置给出就好了，有时候找不到，可以么

展开 ∨



fangxuan

2019-05-09



如果一个进程只有主线程，那么task_struct是一个还是两个？如果是一个还好，这个task_struct既代表进程也代表主线程；如果是两个，进程的pid,tgid都指向自己，那怎么知

道主线程是谁?

展开 ▾



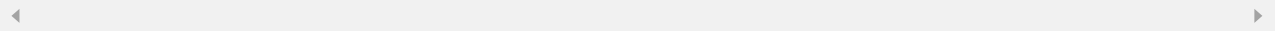
咖啡巧克力...

2019-05-05



task_struct是个描述每个任务的结构体，任务有个链表，结点就是每个任务的task_struct是这个意思吗

作者回复: 是的



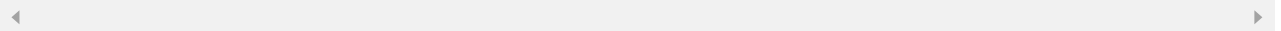
Leon

2019-05-02



老师，可中断睡眠是不是对应软中断，不可中断睡眠对应硬中断，这几个对应关系能详细解释下嘛

作者回复: 不是的，和软硬中断没有关系，是信号到来的时候的处理机制问题



TinnyFlam...

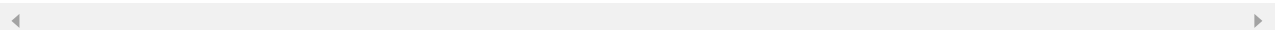
2019-04-25



数组和链表虽然都是线性表的实现，但是就这个应用场景来说链表明显不合适。首先数组的插入和删除是 $O(N)$ 级别的，对于进程管理来说，明显插入和删除操作是主要需求，设想有大量的进程和线程在一个数组里，这时候有调度需求给他们挪位置画面就太美好了.....而且对于很多调度算法来说，链表操作起来都非常方便快速。而数组的优点无非是随机访问和对CPU缓存机制更友好，但说实话我想不到这两个点在进程管理时有什么太...

展开 ▾

作者回复: 是链表啊



免费的人

2019-04-24



说用数组替换链表的人 你们问问题的时候思考过吗

展开 ▾

作者回复: 是的, 想想插入删除

◀ ▶



庄小P

2019-04-24



老师, 附上代码的能够能加上是哪个头文件那, 想具体看看里面的某些东西

展开 ▾

作者回复: 这个考虑了好久, 加不加从哪个文件到哪个文件, 后来想应该把代码当成原理和流程的佐证, 而不是一本逐行解析代码的书, 所以就没加, 后面可以考虑加上

◀ ▶



tux

2019-04-23



需要多下一番努力(文章末尾:反复研读)

展开 ▾



道觉迷遥

2019-04-23



请问老师, 这里底层用链表来连接所有task_struct是出于什么考量呢? 想了想没相处理由, 发现数组也能吧。

作者回复: 创建和删除进程, 再创建再删除

◀ ▶



道觉迷遥

2019-04-23



进程和线程在底层核心用统一的数据结构task_struct来表示, 其实可以根据一些字段来区别是哪一类型。

作者回复: pid和tgid

◀ ▶



一笔一画

2019-04-23



老师，请教一下，之前看书上说用户进程和内核线程是多对多的模型？这个怎么理解，我们常用的发行版又是怎样的模型？

作者回复: 操作系统的理论是有多种模型的，多对一，一对一，多对多，Linux是一对一。

