

18 | 进程的创建：如何发起一个新项目？

2019-05-08 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 10:38 大小 9.75M



前面我们学习了如何使用 fork 创建进程，也学习了进程管理和调度的相关数据结构。这一节，我们就来看一看，创建进程这个动作在内核里都做了什么事情。

fork 是一个系统调用，根据咱们讲过的系统调用的流程，流程的最后会在 sys_call_table 中找到相应的系统调用 sys_fork。


sys_fork 是如何定义的呢？根据 SYSCALL_DEFINE0 这个宏的定义，下面这段代码就定义了 sys_fork。

复制代码

```
1 SYSCALL_DEFINE0(fork)
2 {
3     .....
```

```
4         return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
5     }
```

`sys_fork` 会调用 `_do_fork`。

 复制代码

```
1 long _do_fork(unsigned long clone_flags,
2               unsigned long stack_start,
3               unsigned long stack_size,
4               int __user *parent_tidptr,
5               int __user *child_tidptr,
6               unsigned long tls)
7 {
8     struct task_struct *p;
9     int trace = 0;
10    long nr;
11
12
13    .....
14    p = copy_process(clone_flags, stack_start, stack_size,
15                    child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
16    .....
17    if (!IS_ERR(p)) {
18        struct pid *pid;
19        pid = get_task_pid(p, PIDTYPE_PID);
20        nr = pid_vnr(pid);
21
22
23        if (clone_flags & CLONE_PARENT_SETTID)
24            put_user(nr, parent_tidptr);
25
26
27    .....
28        wake_up_new_task(p);
29    .....
30        put_pid(pid);
31    }
32    .....
```

fork 的第一件大事：复制结构

`_do_fork` 里面做的第一件大事就是 `copy_process`，咱们前面讲过这个思想。如果所有数据结构都从头创建一份太麻烦了，还不如使用惯用“伎俩”，`Ctrl C + Ctrl V`。

这里我们再把 task_struct 的结构图拿出来，对比着看如何一个个复制。



复制代码

```
1 static __latent_entropy struct task_struct *copy_process(  
2     unsigned long clone_flags,  
3     unsigned long stack_start,  
4     unsigned long stack_size,  
5     int __user *child_tidptr,  
6     struct pid *pid,  
7     int trace,  
8     unsigned long tls,
```

```
9                                     int node)
10 {
11     int retval;
12     struct task_struct *p;
13     .....
14     p = dup_task_struct(current, node);
```

dup_task_struct 主要做了下面几件事情：

调用 alloc_task_struct_node 分配一个 task_struct 结构；


调用 alloc_thread_stack_node 来创建内核栈，这里面调用 __vmalloc_node_range 分配一个连续的 THREAD_SIZE 的内存空间，赋值给 task_struct 的 void *stack 成员变量；

调用 arch_dup_task_struct(struct task_struct *dst, struct task_struct *src)，将 task_struct 进行复制，其实就是调用 memcpy；

调用 setup_thread_stack 设置 thread_info。

到这里，整个 task_struct 复制了一份，而且内核栈也创建好了。

我们再接着看 copy_process。

 复制代码

```
1  retval = copy_creds(p, clone_flags);
```

轮到权限相关了，copy_creds 主要做了下面几件事情：

调用 prepare_creds，准备一个新的 struct cred *new。如何准备呢？其实还是从内存中分配一个新的 struct cred 结构，然后调用 memcpy 复制一份父进程的 cred；

接着 p->cred = p->real_cred = get_cred(new)，将新进程的“我能操作谁”和“谁能操作我”两个权限都指向新的 cred。

接下来，copy_process 重新设置进程运行的统计量。

```
1 p->utime = p->stime = p->gtime = 0;
2 p->start_time = ktime_get_ns();
3 p->real_start_time = ktime_get_boot_ns();
```

接下来，`copy_process` 开始设置调度相关的变量。

```
1 retval = sched_fork(clone_flags, p);
```

`sched_fork` 主要做了下面几件事情：

调用 `__sched_fork`，在这里面将 `on_rq` 设为 0，初始化 `sched_entity`，将里面的 `exec_start`、`sum_exec_runtime`、`prev_sum_exec_runtime`、`vruntime` 都设为 0。你还记得吗，这几个变量涉及进程的实际运行时间和虚拟运行时间。是否到时间应该被调度了，就靠它们几个；

设置进程的状态 `p->state = TASK_NEW`；

初始化优先级 `prio`、`normal_prio`、`static_prio`；

设置调度类，如果是普通进程，就设置为 `p->sched_class = &fair_sched_class`；

调用调度类的 `task_fork` 函数，对于 CFS 来讲，就是调用 `task_fork_fair`。在这个函数里，先调用 `update_curr`，对于当前的进程进行统计量更新，然后把子进程和父进程的 `vruntime` 设成一样，最后调用 `place_entity`，初始化 `sched_entity`。这里有一个变量 `sysctl_sched_child_runs_first`，可以设置父进程和子进程谁先运行。如果设置了子进程先运行，即便两个进程的 `vruntime` 一样，也要把子进程的 `sched_entity` 放在前面，然后调用 `resched_curr`，标记当前运行的进程 `TIF_NEED_RESCHED`，也就是说，把父进程设置为应该被调度，这样下次调度的时候，父进程会被子进程抢占。


接下来，`copy_process` 开始初始化与文件和文件系统相关的变量。

```
1 retval = copy_files(clone_flags, p);
2 retval = copy_fs(clone_flags, p);
```

`copy_files` 主要用于复制一个进程打开的文件信息。这些信息用一个结构 `files_struct` 来维护，每个打开的文件都有一个文件描述符。在 `copy_files` 函数里面调用 `dup_fd`，在这里面会创建一个新的 `files_struct`，然后将所有的文件描述符数组 `fdtable` 拷贝一份。

`copy_fs` 主要用于复制一个进程的目录信息。这些信息用一个结构 `fs_struct` 来维护。一个进程有自己的根目录和根文件系统 `root`，也有当前目录 `pwd` 和当前目录的文件系统，都在 `fs_struct` 里面维护。`copy_fs` 函数里面调用 `copy_fs_struct`，创建一个新的 `fs_struct`，并复制原来进程的 `fs_struct`。

接下来，`copy_process` 开始初始化与信号相关的变量。


 复制代码

```
1 init_sigpending(&p->pending);
2 retval = copy_sighand(clone_flags, p);
3 retval = copy_signal(clone_flags, p);
```

`copy_sighand` 会分配一个新的 `sighand_struct`。这里最主要的是维护信号处理函数，在 `copy_sighand` 里面会调用 `memcpy`，将信号处理函数 `sighand->action` 从父进程复制到子进程。

`init_sigpending` 和 `copy_signal` 用于初始化，并且复制用于维护发给这个进程的信号的数据结构。`copy_signal` 函数会分配一个新的 `signal_struct`，并进行初始化。

接下来，`copy_process` 开始复制进程内存空间。


 复制代码

```
1 retval = copy_mm(clone_flags, p);
```

进程都有自己的内存空间，用 `mm_struct` 结构来表示。`copy_mm` 函数中调用 `dup_mm`，分配一个新的 `mm_struct` 结构，调用 `memcpy` 复制这个结构。`dup_mmap` 用于复制内存空间中内存映射的部分。前面讲系统调用的时候，我们说过，`mmap` 可以分配大块的内

存，其实 `mmap` 也可以将一个文件映射到内存中，方便可以像读写内存一样读写文件，这个在内存管理那节我们讲。

接下来，`copy_process` 开始分配 `pid`，设置 `tid`，`group_leader`，并且建立进程之间的亲缘关系。


 复制代码

```
1      INIT_LIST_HEAD(&p->children);
2      INIT_LIST_HEAD(&p->sibling);
3      .....
4      p->pid = pid_nr(pid);
5      if (clone_flags & CLONE_THREAD) {
6          p->exit_signal = -1;
7          p->group_leader = current->group_leader;
8          p->tgid = current->tgid;
9      } else {
10         if (clone_flags & CLONE_PARENT)
11             p->exit_signal = current->group_leader->exit_signal;
12         else
13             p->exit_signal = (clone_flags & CSIGNAL);
14         p->group_leader = p;
15         p->tgid = p->pid;
16     }
17     .....
18     if (clone_flags & (CLONE_PARENT|CLONE_THREAD)) {
19         p->real_parent = current->real_parent;
20         p->parent_exec_id = current->parent_exec_id;
21     } else {
22         p->real_parent = current;
23         p->parent_exec_id = current->self_exec_id;
24     }
```

好了，`copy_process` 要结束了，上面图中的组件也初始化的差不多了。

fork 的第二件大事：唤醒新进程

`_do_fork` 做的第二件大事是 `wake_up_new_task`。新任务刚刚建立，有没有机会抢占别人，获得 CPU 呢？

 复制代码

```
1 void wake_up_new_task(struct task_struct *p)
2 {
```


```

3      struct rq_flags rf;
4      struct rq *rq;
5      .....
6      p->state = TASK_RUNNING;
7      .....
8      activate_task(rq, p, ENQUEUE_NOCLOCK);
9      p->on_rq = TASK_ON_RQ_QUEUED;
10     trace_sched_wakeup_new(p);
11     check_preempt_curr(rq, p, WF_FORK);
12     .....
13 }

```

首先，我们需要将进程的状态设置为 TASK_RUNNING。

activate_task 函数中会调用 enqueue_task。


 复制代码

```

1 static inline void enqueue_task(struct rq *rq, struct task_struct *p, int flags)
2 {
3     .....
4     p->sched_class->enqueue_task(rq, p, flags);
5 }

```

如果是 CFS 的调度类，则执行相应的 enqueue_task_fair。

 复制代码

```

1 static void
2 enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
3 {
4     struct cfs_rq *cfs_rq;
5     struct sched_entity *se = &p->se;
6     .....
7     cfs_rq = cfs_rq_of(se);
8     enqueue_entity(cfs_rq, se, flags);
9     .....
10    cfs_rq->h_nr_running++;
11    .....
12 }


```


在 enqueue_task_fair 中取出的队列就是 cfs_rq，然后调用 enqueue_entity。

在 enqueue_entity 函数里面，会调用 update_curr，更新运行的统计量，然后调用 __enqueue_entity，将 sched_entity 加入到红黑树里面，然后将 se->on_rq = 1 设置在队列上。

回到 enqueue_task_fair 后，将这个队列上运行的进程数目加一。然后，wake_up_new_task 会调用 check_preempt_curr，看是否能够抢占当前进程。

在 check_preempt_curr 中，会调用相应的调度类的 rq->curr->sched_class->check_preempt_curr(rq, p, flags)。对于 CFS 调度类来讲，调用的是 check_preempt_wakeup。

 复制代码

```
1 static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
2 {
3     struct task_struct *curr = rq->curr;
4     struct sched_entity *se = &curr->se, *pse = &p->se;
5     struct cfs_rq *cfs_rq = task_cfs_rq(curr);
6     .....
7     if (test_tsk_need_resched(curr))
8         return;
9     .....
10    find_matching_se(&se, &pse);
11    update_curr(cfs_rq_of(se));
12    if (wakeup_preempt_entity(se, pse) == 1) {
13        goto preempt;
14    }
15    return;
16 preempt:
17    resched_curr(rq);
18    .....
19 }
```

在 check_preempt_wakeup 函数中，前面调用 task_fork_fair 的时候，设置 sysctl_sched_child_runs_first 了，已经将当前父进程的 TIF_NEED_RESCHED 设置了，则直接返回。

否则，`check_preempt_wakeup` 还是会调用 `update_curr` 更新一次统计量，然后 `wakeup_preempt_entity` 将父进程和子进程 PK 一次，看是不是要抢占，如果要则调用 `resched_curr` 标记父进程为 `TIF_NEED_RESCHED`。

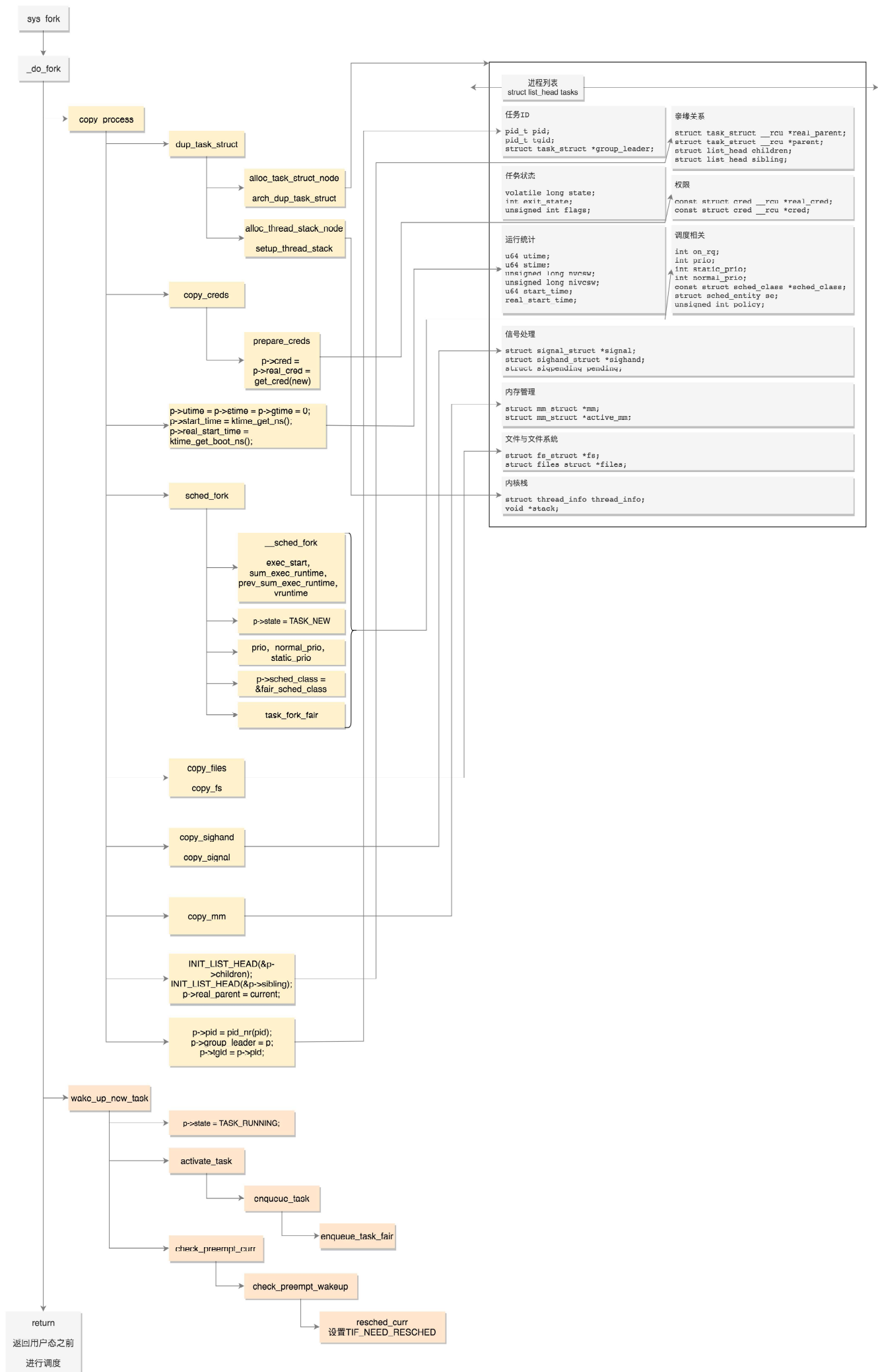
如果新创建的进程应该抢占父进程，在什么时间抢占呢？别忘了 `fork` 是一个系统调用，从系统调用返回的时候，是抢占的一个好时机，如果父进程判断自己已经被设置为 `TIF_NEED_RESCHED`，就让子进程先跑，抢占自己。

总结时刻

好了，`fork` 系统调用的过程咱们就解析完了。它包含两个重要的事件，一个是将 `task_struct` 结构复制一份并且初始化，另一个是试图唤醒新创建的子进程。

这个过程我画了一张图，你可以对照着这张图回顾进程创建的过程。

这个图的上半部分是复制 `task_struct` 结构，你可以对照着右面的 `task_struct` 结构图，看这里面的成员是如何一部分一部分的被复制的。图的下半部分是唤醒新创建的子进程，如果条件满足，就会将当前进程设置应该被调度的标识位，就等着当前进程执行 `__schedule` 了。



课堂练习

你可以试着设置 `sysctl_sched_child_runs_first` 参数，然后使用系统调用写程序创建进程，看看执行结果。

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。

 极客时间

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 17 | 调度（下）：抢占式调度是如何发生的？

下一篇 19 | 线程的创建：如何执行一个新子项目？

精选留言 (13)

写留言



why

2019-05-10

4

- fork -> sys_call_table 转换为 sys_fork()->`_do_fork`

- 创建进程做两件事: 复制初始化 task_struct; 唤醒新进程
- 复制并初始化 task_struct, copy_process()
 - dup_task_struct: 分配 task_struct 结构体; 创建内核栈, 赋给`* stack`; 复制 task_struct, 设置 thread info;...

展开 ∨



刘強

2019-05-08

👍 1

有个问题:

在数据库中, 有个事务的概念, 也就是保证一连串操作的原子性, 如果其中任何一步错误, 整个操作回滚, 回到原来的状态, 好像什么也没发生。但是在文章中我看到, 在创建进程的过程中, 步骤太多了。每一步都要申请空间, 复制数据。如果其中一步发生了错误, 怎么保证释放这些空间, 回到原来状态?

展开 ∨



刘強

2019-05-08

👍 1

文章中出现了SYSCALL_DEFINE0宏定义, 不明白, 就网上查了一下, 一看吓一跳, 宏定义里面又有一堆宏定义, 其实就是一个函数调用, 为什么弄得这么复杂呢? 原来是为了修复一个bug。这让我意识到linux内核代码的复杂性。linux是一个集大成者, 为了适应各种硬件架构平台, 修复各种意想不到的bug, 里面充斥着各种兼容性代码, 修复补丁等等。而且里面的代码也是世界各路大神, 黑客写出来的, 为了保证内核的安全性, 健壮性, 扩...

展开 ∨



一苇渡江

2019-05-08

👍 1

老师写的太棒了, 特别是这个图, 肯定是花了不少时间, 把这个图手抄了一遍, 时不时拿出来看看



小美

2019-05-24

👍

内核态的内核进程和用户态的用户进程创建过程有区别吗?

展开 ∨

作者回复: 有区别的



周平

2019-05-17



讲得好的细节，与前面的内容可以无缝连接，不至于管中窥豹，让学习者越学越乱，谢谢老师



尚墨

2019-05-11



反复研读都已经高亮了。我几乎每篇都要听，读三次以上，才能懵懵懂懂。

展开 ∨



Milittle

2019-05-10



老师，要是能把对应代码路径给出就好了，有时候自己找不见，谢谢老师~

展开 ∨



免费的人

2019-05-09



我是来收图的。

展开 ∨



青石

2019-05-09



如果是完全公平调度算法的话，`sched_fork`的时候，将子进程的vruntime修改为与父进程的vruntime一致，是为了将子进程的vruntime设置到与其他进程在同一个量级上，父进程的执行说明当前它处在红黑树的最左节点，将父进程的TIF_NEED_RESCHED标记为允许被抢占，当系统回调时调用`__schedule()`，更新父进程的vruntime后，子进程处在红黑树最左节点，此时运行子进程。

展开 ∨



chengzise

2019-05-08



解析的逻辑特别清晰，后面需要自己结合代码，加深理解。老师说的太好了

展开 ▾



安排

2019-05-08



调度类是全局的吗？ 还是每个cpu核有自己的调度类集合？

展开 ▾



blentle

2019-05-08



子进程是如何抢占父进程的呢？

展开 ▾