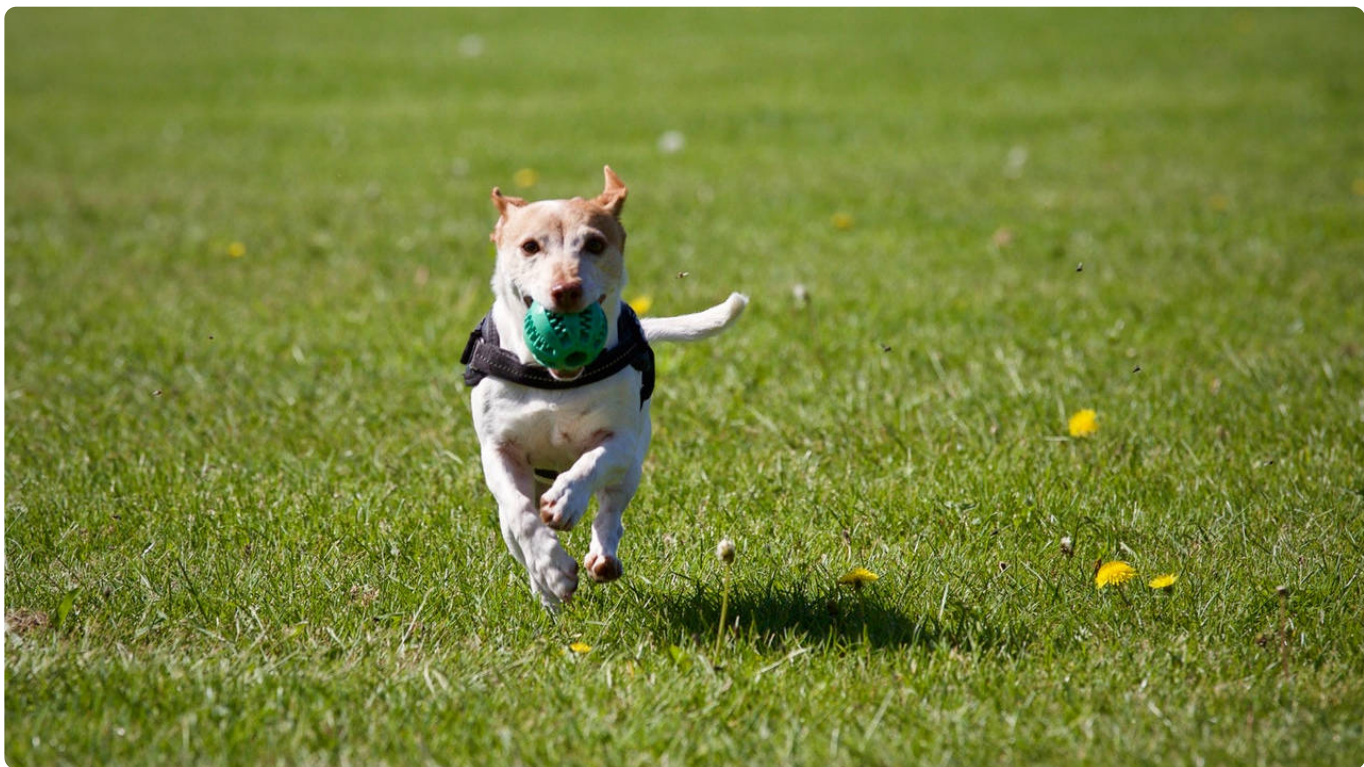


19 | 线程的创建：如何执行一个新子项目？

2019-05-10 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 13:06 大小 12.00M



上一节，我们了解了进程创建的整个过程，今天我们来看线程创建的过程。


我们前面已经写过多线程编程的程序了，你应该都知道创建一个线程调用的是 `pthread_create`，可以你知道它背后的机制吗？

用户态创建线程

你可能会问，咱们之前不是讲过了吗？无论是进程还是线程，在内核里面都是任务，管起来不是都一样吗？但是问题来了，如果两个完全一样，那为什么咱们前两节写的程序差别那么大？如果不一样，那怎么在内核里面加以区分呢？

其实，线程不是一个完全由内核实现的机制，它是由内核态和用户态合作完成的。
pthread_create 不是一个系统调用，是 Glibc 库的一个函数，所以我们还要去 Glibc 里面去找线索。


果然，我们在 nptl/pthread_create.c 里面找到了这个函数。这里的参数我们应该比较熟悉了。

 复制代码

```
1 int __pthread_create_2_1 (pthread_t *newthread, const pthread_attr_t *attr, void *(*sta
2 {
3     .....
4 }
5 versioned_symbol (libpthread, __pthread_create_2_1, pthread_create, GLIBC_2_1);
```


下面我们依次来看这个函数做了些啥。

首先处理的是线程的属性参数。例如前面写程序的时候，我们设置的线程栈大小。如果没有传入线程属性，就取默认值。

 复制代码


```
1 const struct pthread_attr *iattr = (struct pthread_attr *) attr;
2 struct pthread_attr default_attr;
3 if (iattr == NULL)
4 {
5     .....
6     iattr = &default_attr;
7 }
```

接下来，就像在内核里一样，每一个进程或者线程都有一个 task_struct 结构，在用户态也有一个用于维护线程的结构，就是这个 pthread 结构。

 复制代码


```
1 struct pthread *pd = NULL;
```

凡是涉及函数的调用，都要使用到栈。每个线程也有自己的栈。那接下来就是创建线程栈了。

 复制代码

```
1 int err = ALLOCATE_STACK (iattr, &pd);
```

ALLOCATE_STACK 是一个宏，我们找到它的定义之后，发现它其实就是一个函数。只是，这个函数有些复杂，所以我这里把主要的代码列一下。

 复制代码

```
1 # define ALLOCATE_STACK(attr, pd) allocate_stack (attr, pd, &stackaddr)
2
3
4 static int
5 allocate_stack (const struct pthread_attr *attr, struct pthread **pdp,
6                 ALLOCATE_STACK_PARMS)
7 {
8     struct pthread *pd;
9     size_t size;
10    size_t pagesize_m1 = __getpagesize () - 1;
11    .....
12    size = attr->stacksize;
13    .....
14    /* Allocate some anonymous memory.  If possible use the cache.  */
15    size_t guardsize;
16    void *mem;
17    const int prot = (PROT_READ | PROT_WRITE
18                     | ((GL(dl_stack_flags) & PF_X) ? PROT_EXEC : 0));
19    /* Adjust the stack size for alignment.  */
20    size &= ~__static_tls_align_m1;
21    /* Make sure the size of the stack is enough for the guard and
22       eventually the thread descriptor.  */
23    guardsize = (attr->guardsize + pagesize_m1) & ~pagesize_m1;
24    size += guardsize;
25    pd = get_cached_stack (&size, &mem);
26    if (pd == NULL)
27    {
28        /* If a guard page is required, avoid committing memory by first
29           allocate with PROT_NONE and then reserve with required permission
30           excluding the guard page.  */
31        mem = __mmap (NULL, size, (guardsize == 0) ? prot : PROT_NONE,
32                     MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
33        /* Place the thread descriptor at the end of the stack.  */
34    #if TLS_TCB_AT_TP
35        pd = (struct pthread *) ((char *) mem + size) - 1;
```

```

36 #elif TLS_DTV_AT_TP
37     pd = (struct pthread *) (((uintptr_t) mem + size - __static_tls_size) & ~__static_1
38 #endif
39     /* Now mprotect the required region excluding the guard area. */
40     char *guard = guard_position (mem, size, guardsize, pd, pagesize_m1);
41     setup_stack_prot (mem, size, guard, guardsize, prot);
42     pd->stackblock = mem;
43     pd->stackblock_size = size;
44     pd->guardsize = guardsize;
45     pd->specific[0] = pd->specific_1stblock;
46     /* And add to the list of stacks in use. */
47     stack_list_add (&pd->list, &stack_used);
48 }
49
50 *pdp = pd;
51 void *stacktop;
52 # if TLS_TCB_AT_TP
53     /* The stack begins before the TCB and the static TLS block. */
54     stacktop = ((char *) (pd + 1) - __static_tls_size);
55 # elif TLS_DTV_AT_TP
56     stacktop = (char *) (pd - 1);
57 # endif
58 *stack = stacktop;
59 .....
60 }

```

我们来看一下，`allocate_stack` 主要做了以下这些事情：

如果你在线程属性里面设置过栈的大小，需要你把设置的值拿出来；

为了防止栈的访问越界，在栈的末尾会有一块空间 `guardsize`，一旦访问到这里就错误了；

其实线程栈是在进程的堆里面创建的。如果一个进程不断地创建和删除线程，我们不可能不断地去申请和清除线程栈使用的内存块，这样就需要有一个缓存。`get_cached_stack` 就是根据计算出来的 `size` 大小，看一看已经有的缓存中，有没有已经能够满足条件的；

如果缓存里面没有，就需要调用 `__mmap` 创建一块新的，系统调用那一节我们讲过，如果要在堆里面 `malloc` 一块内存，比较大的话，用 `__mmap`；

线程栈也是自顶向下生长的，还记得每个线程要有一个 `pthread` 结构，这个结构也是放在栈的空间里面的。在栈底的位置，其实是地址最高位；

计算出 `guard` 内存的位置，调用 `setup_stack_prot` 设置这块内存的是受保护的；


接下来，开始填充 pthread 这个结构里面的成员变量 stackblock、stackblock_size、guardsize、specific。这里的 specific 是用于存放 Thread Specific Data 的，也即属于线程的全局变量；

将这个线程栈放到 stack_used 链表中，其实管理线程栈总共有两个链表，一个是 stack_used，也就是这个栈正被使用；另一个是 stack_cache，就是上面说的，一旦线程结束，先缓存起来，不释放，等有其他的线程创建的时候，给其他的线程用。

搞定了用户态栈的问题，其实用户态的事情基本搞定了一半。

内核态创建任务

接下来，我们接着 pthread_create 看。其实有了用户态的栈，接着需要解决的就是用户态的程序从哪里开始运行的问题。


 复制代码

```
1 pd->start_routine = start_routine;
2 pd->arg = arg;
3 pd->schedpolicy = self->schedpolicy;
4 pd->schedparam = self->schedparam;
5 /* Pass the descriptor to the caller. */
6 *newthread = (pthread_t) pd;
7 atomic_increment (&__nptl_nthreads);
8 retval = create_thread (pd, iattr, &stopped_start, STACK_VARIABLES_ARGS, &thread_ran);
```

start_routine 就是咱们给线程的函数，start_routine，start_routine 的参数 arg，以及调度策略都要赋值给 pthread。

接下来 __nptl_nthreads 加一，说明有多了一个线程。

真正创建线程的是调用 create_thread 函数，这个函数定义如下：

 复制代码

```
1 static int
2 create_thread (struct pthread *pd, const struct pthread_attr *attr,
3 bool *stopped_start, STACK_VARIABLES_PARAMS, bool *thread_ran)
4 {
5     const int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM | CLONE_SIGHAND
6     ARCH_CLONE (&start_thread, STACK_VARIABLES_ARGS, clone_flags, pd, &pd->tid, tp, &pd->tid,
```

```


7  /* It's started now, so if we fail below, we'll have to cancel it
8  and let it clean itself up.  */
9  *thread_ran = true;
10 }

```



这里面有很长的 `clone_flags`，这些咱们原来一直没注意，不过接下来的过程，我们要特别的关注一下这些标志位。

然后就是 `ARCH_CLONE`，其实调用的是 `__clone`。看到这里，你应该就有感觉了，马上就要到系统调用了。

 复制代码

```

1  # define ARCH_CLONE __clone
2
3
4  /* The userland implementation is:
5     int clone (int (*fn)(void *arg), void *child_stack, int flags, void *arg),
6     the kernel entry is:
7     int clone (long flags, void *child_stack).
8
9
10     The parameters are passed in register and on the stack from userland:
11     rdi: fn
12     rsi: child_stack
13     rdx: flags
14     rcx: arg
15     r8d: TID field in parent
16     r9d: thread pointer
17 %esp+8: TID field in child
18
19
20     The kernel expects:
21     rax: system call number
22     rdi: flags
23     rsi: child_stack
24     rdx: TID field in parent
25     r10: TID field in child
26     r8:  thread pointer  */
27
28     .text
29 ENTRY (__clone)
30     movq    $-EINVAL,%rax
31     .....
32     /* Insert the argument onto the new stack.  */
33     subq    $16,%rsi

```



```

34      movq    %rcx,8(%rsi)
35
36
37      /* Save the function pointer.  It will be popped off in the
38         child in the ebx frobbing below.  */
39      movq    %rdi,0(%rsi)
40
41
42      /* Do the system call.  */
43      movq    %rdx, %rdi
44      movq    %r8, %rdx
45      movq    %r9, %r8
46      mov     8(%rsp), %R10_LP
47      movl    $SYS_ify(clone),%eax
48 .....
49      syscall
50 .....
51 PSEUDO_END (__clone)

```

如果对于汇编不太熟悉也没关系，你可以重点看上面的注释。

我们能看到最后调用了 `syscall`，这一点 `clone` 和我们原来熟悉的其他系统调用几乎是一致的。但是，也有少许不一样的地方。

如果在进程的主线程里面调用其他系统调用，当前用户态的栈是指向整个进程的栈，栈顶指针也是指向进程的栈，指令指针也是指向进程的主线程的代码。此时此刻执行到这里，调用 `clone` 的时候，用户态的栈、栈顶指针、指令指针和其他系统调用一样，都是指向主线程的。

但是对于线程来说，这些都要变。因为我们希望当 `clone` 这个系统调用成功的时候，除了内核里面有这个线程对应的 `task_struct`，当系统调用返回到用户态的时候，用户态的栈应该是线程的栈，栈顶指针应该指向线程的栈，指令指针应该指向线程将要执行的那个函数。

所以这些都需要我们自己做，将线程要执行的函数的参数和指令的位置都压到栈里面，当从内核返回，从栈里弹出来的时候，就从这个函数开始，带着这些参数执行下去。

接下来我们就要进入内核了。内核里面对于 `clone` 系统调用的定义是这样的：

```


1 SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
2                 int __user *, parent_tidptr,
3                 int __user *, child_tidptr,
4                 unsigned long, tls)
5 {
6     return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr, tls);
7 }

```

看到这里，发现了熟悉的面孔 `_do_fork`，是不是轻松了一些？上一节我们已经沿着它的逻辑过了一遍了。这里我们重点关注几个区别。

第一个是上面**复杂的标志位设定**，我们来看都影响了什么。

对于 `copy_files`，原来是调用 `dup_fd` 复制一个 `files_struct` 的，现在因为 `CLONE_FILES` 标识位变成将原来的 `files_struct` 引用计数加一。


 复制代码

```

1 static int copy_files(unsigned long clone_flags, struct task_struct *tsk)
2 {
3     struct files_struct *oldf, *newf;
4     oldf = current->files;
5     if (clone_flags & CLONE_FILES) {
6         atomic_inc(&oldf->count);
7         goto out;
8     }
9     newf = dup_fd(oldf, &error);
10    tsk->files = newf;
11 out:
12    return error;
13 }

```

对于 `copy_fs`，原来是调用 `copy_fs_struct` 复制一个 `fs_struct`，现在因为 `CLONE_FS` 标识位变成将原来的 `fs_struct` 的用户数加一。

 复制代码

```

1 static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
2 {
3     struct fs_struct *fs = current->fs;
4     if (clone_flags & CLONE_FS) {
5         fs->users++;

```



```

6         return 0;
7     }
8     tsk->fs = copy_fs_struct(fs);
9     return 0;
10 }

```

对于 `copy_sighand`，原来是创建一个新的 `sighand_struct`，现在因为 `CLONE_SIGHAND` 标识位变成将原来的 `sighand_struct` 引用计数加一。

 复制代码

```

1 static int copy_sighand(unsigned long clone_flags, struct task_struct *tsk)
2 {
3     struct sighand_struct *sig;
4
5
6     if (clone_flags & CLONE_SIGHAND) {
7         atomic_inc(&current->sighand->count);
8         return 0;
9     }
10    sig = kmem_cache_alloc(sighand_cache, GFP_KERNEL);
11    atomic_set(&sig->count, 1);
12    memcpy(sig->action, current->sighand->action, sizeof(sig->action));
13    return 0;
14 }

```

对于 `copy_signal`，原来是创建一个新的 `signal_struct`，现在因为 `CLONE_THREAD` 直接返回了。


 复制代码

```

1 static int copy_signal(unsigned long clone_flags, struct task_struct *tsk)
2 {
3     struct signal_struct *sig;
4     if (clone_flags & CLONE_THREAD)
5         return 0;
6     sig = kmem_cache_zalloc(signal_cache, GFP_KERNEL);
7     tsk->signal = sig;
8     init_sigpending(&sig->shared_pending);
9     .....
10 }


```

对于 copy_mm, 原来是调用 dup_mm 复制一个 mm_struct, 现在因为 CLONE_VM 标识位而直接指向了原来的 mm_struct

 复制代码

```
1 static int copy_mm(unsigned long clone_flags, struct task_struct *tsk)
2 {
3     struct mm_struct *mm, *oldmm;
4     oldmm = current->mm;
5     if (clone_flags & CLONE_VM) {
6         mmget(oldmm);
7         mm = oldmm;
8         goto good_mm;
9     }
10    mm = dup_mm(tsk);
11 good_mm:
12    tsk->mm = mm;
13    tsk->active_mm = mm;
14    return 0;
15 }
```

第二个就是**对于亲缘关系的影响**, 毕竟我们要识别多个线程是不是属于一个进程。

 复制代码

```
1 p->pid = pid_nr(pid);
2 if (clone_flags & CLONE_THREAD) {
3     p->exit_signal = -1;
4     p->group_leader = current->group_leader;
5     p->tgid = current->tgid;
6 } else {
7     if (clone_flags & CLONE_PARENT)
8         p->exit_signal = current->group_leader->exit_signal;
9     else
10        p->exit_signal = (clone_flags & CSIGNAL);
11    p->group_leader = p;
12    p->tgid = p->pid;
13 }
14 /* CLONE_PARENT re-uses the old parent */
15 if (clone_flags & (CLONE_PARENT|CLONE_THREAD)) {
16     p->real_parent = current->real_parent;
17     p->parent_exec_id = current->parent_exec_id;
18 } else {
19     p->real_parent = current;
20     p->parent_exec_id = current->self_exec_id;
21 }
```


从上面的代码可以看出，使用了 CLONE_THREAD 标识位之后，使得亲缘关系有了一定的变化。

如果是新进程，那这个进程的 group_leader 就是他自己，tgid 是它自己的 pid，这就完全重打锣鼓另开张了，自己是线程组的头。如果是新线程，group_leader 是当前进程的，group_leader，tgid 是当前进程的 tgid，也就是当前进程的 pid，这个时候还是拜原来进程为老大。

如果是新进程，新进程的 real_parent 是当前的进程，在进程树里面又见一辈人；如果是新线程，线程的 real_parent 是当前的进程的 real_parent，其实是平辈的。


第三，**对于信号的处理**，如何保证发给进程的信号虽然可以被一个线程处理，但是影响范围应该是整个进程的。例如，kill 一个进程，则所有线程都要被干掉。如果一个信号是发给一个线程的 pthread_kill，则应该只有线程能够收到。

在 copy_process 的主流程里面，无论是创建进程还是线程，都会初始化 struct sigpending pending，也就是每个 task_struct，都会有这样一个成员变量。这就是一个信号列表。如果这个 task_struct 是一个线程，这里的信号就是发给这个线程的；如果这个 task_struct 是一个进程，这里的信号是发给主线程的。

 复制代码

```
1 init_sigpending(&p->pending);
```

另外，上面 copy_signal 的时候，我们可以看到，在创建进程的过程中，会初始化 signal_struct 里面的 struct sigpending shared_pending。但是，在创建线程的过程中，连 signal_struct 都共享了。也就是说，整个进程里的所有线程共享一个 shared_pending，这也是一个信号列表，是发给整个进程的，哪个线程处理都一样。


 复制代码

```
1 init_sigpending(&sig->shared_pending);
```

至此，clone 在内核的调用完毕，要返回系统调用，回到用户态。


用户态执行线程

根据 `__clone` 的第一个参数，回到用户态也不是直接运行我们指定的那个函数，而是一个通用的 `start_thread`，这是所有线程在用户态的统一入口。

 复制代码

```
1 #define START_THREAD_DEFN \  
2     static int __attribute__((noretun)) start_thread (void *arg)  
3  
4  
5 START_THREAD_DEFN  
6 {  
7     struct pthread *pd = START_THREAD_SELF;  
8     /* Run the code the user provided. */  
9     THREAD_SETMEM (pd, result, pd->start_routine (pd->arg));  
10    /* Call destructors for the thread_local TLS variables. */  
11    /* Run the destructor for the thread-local data. */  
12    __nptl_deallocate_tsd ();  
13    if (__glibc_unlikely (atomic_decrement_and_test (&__nptl_nthreads)))  
14        /* This was the last thread. */  
15        exit (0);  
16    __free_tcb (pd);  
17    __exit_thread ();  
18 }
```

在 `start_thread` 入口函数中，才真正的调用用户提供的函数，在用户的函数执行完毕之后，会释放这个线程相关的数据。例如，线程本地数据 `thread_local variables`，线程数目也减一。如果这是最后一个线程了，就直接退出进程，另外 `__free_tcb` 用于释放 `pthread`。

 复制代码

```
1 void  
2 internal_function  
3 __free_tcb (struct pthread *pd)  
4 {  
5     .....  
6     __deallocate_stack (pd);  
7 }  
8  
9  
10 void  
11 internal_function  
12 __deallocate_stack (struct pthread *pd)  
13 {
```

```
14  /* Remove the thread from the list of threads with user defined
15     stacks.  */
16  stack_list_del (&pd->list);
17  /* Not much to do.  Just free the mmap()ed memory.  Note that we do
18     not reset the 'used' flag in the 'tid' field.  This is done by
19     the kernel.  If no thread has been created yet this field is
20     still zero.  */
21  if (__glibc_likely (! pd->user_stack))
22      (void) queue_stack (pd);
23 }
```

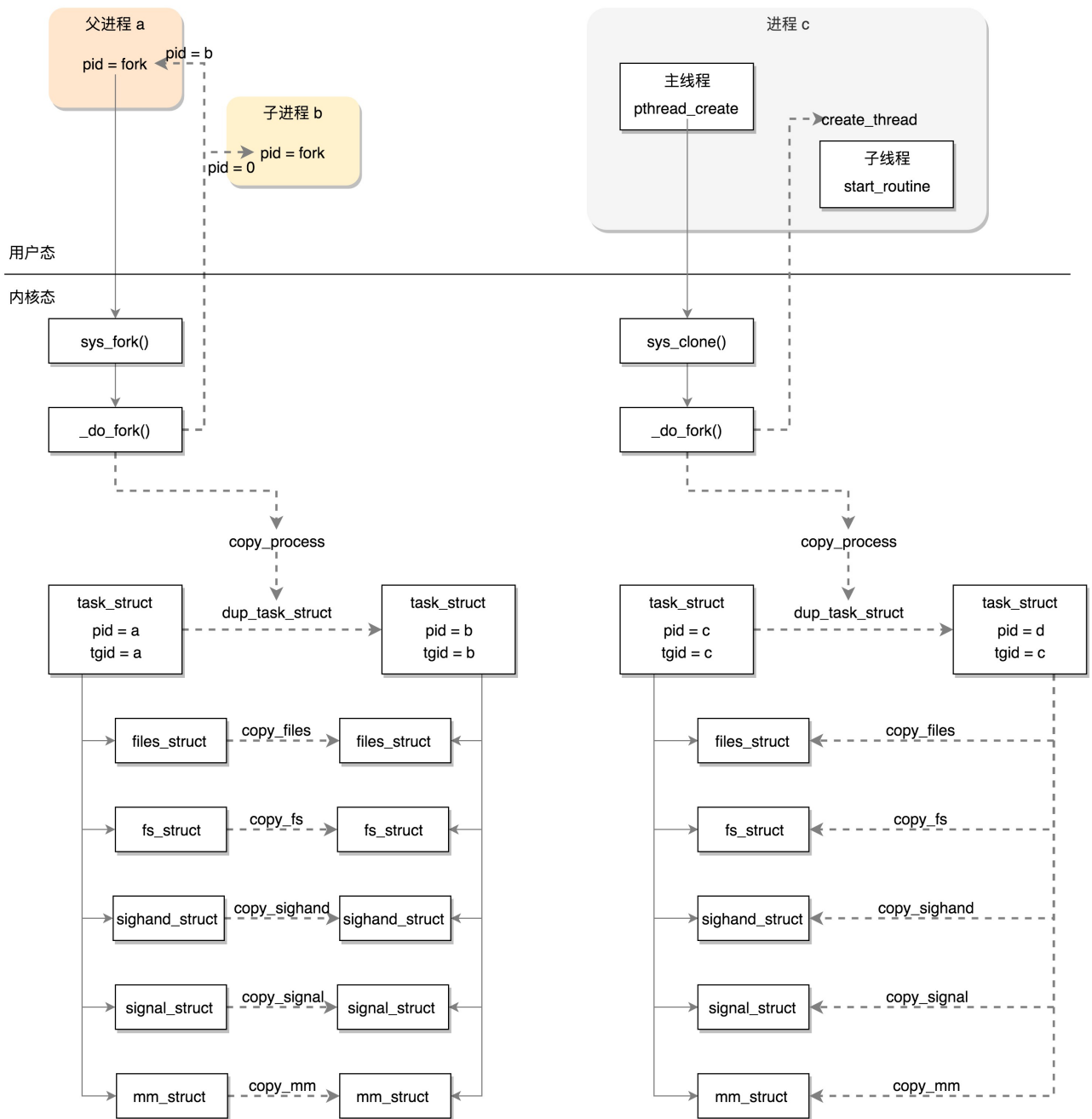
`__free_tcb` 会调用 `__deallocate_stack` 来释放整个线程栈，这个线程栈要从当前使用线程栈的列表 `stack_used` 中拿下来，放到缓存的线程栈列表 `stack_cache` 中。

好了，整个线程的生命周期到这里就结束了。

总结时刻

线程的调用过程解析完毕了，我画了一个图总结一下。这个图对比了创建进程和创建线程在用户态和内核态的不同。

创建进程的话，调用的系统调用是 `fork`，在 `copy_process` 函数里面，会将五大结构 `files_struct`、`fs_struct`、`sighand_struct`、`signal_struct`、`mm_struct` 都复制一遍，从此父进程和子进程各用各的数据结构。而创建线程的话，调用的是系统调用 `clone`，在 `copy_process` 函数里面，五大结构仅仅是引用计数加一，也即线程共享进程的数据结构。



课堂练习

你知道如果查看一个进程的线程以及线程栈的使用情况吗？请找一下相关的命令和 API，尝试一下。

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 18 | 进程的创建：如何发起一个新项目？

下一篇 20 | 内存管理（上）：为客户保密，规划进程内存空间布局

精选留言 (11)

写留言



Milittle

2019-05-10

👍 6

刘老师，您好，您可以把文档中给出的代码文件定位给出来么，一般在对应看源码的时候，很难定位到老师给的代码点的对应源码文件，谢谢老师~老师讲的真的让我把多年的零散知识可以连贯起来，然后理解的更加透彻，但是也会有不太理解的地方，再次感谢。这个课很值得~。

总结以下进程和线程的异同点：...

展开 ▾



why

2019-05-14

👍 3

- 线程的创建

- 线程是由内核态和用户态合作完成的, pthread_create 是 Glibc 库的一个函数
- pthread_create 中
 1. 设置线程属性参数, 如线程栈大小
 2. 创建用户态维护线程的结构, pthread...

展开 ∨



徐凯

2019-05-10

👍 3

"将这个线程栈放到 stack_used 链表中, 其实管理线程栈总共有两个链表, 一个是 stack_used, 也就是这个栈正被使用; 另一个是 stack_cache, 就是上面说的, 一旦线程结束, 先缓存起来, 不释放, 等有其他的线程创建的时候, 给其他的线程用。" 这一段是线程池的意思么 如果是的话 既然内部已经有这个设计 我们有时候还要在程序中自己去设计一个呢?

展开 ∨



why

2019-05-14

👍 1

老师, 多线程的内核栈是共享的吗, 会不会出现问题?

展开 ∨

作者回复: 不共享, 进了内核都是单独的任务了



lfn

2019-05-10

👍 1

所以, 线程局部变量其实是存储在每个线程自己的用户栈里咯?

展开 ∨



humor

2019-05-30

👍

老师之前说过进程默认会有一个主线程, 意思是在创建进程的时候也会同时创建一个线程吗?



nora

👍



2019-05-18

之前总是认为线程和进程都占用了内核的taskstruct 认为实际上线程进程没啥区别，这篇文章真是醍醐灌顶啊，谢谢老师。



WL

2019-05-14



请问一下老师为什么栈的结构是栈底是高地址栈顶是低地址呢，为什么不是反过来的呢？

作者回复: 约定



六月星空20...

2019-05-13



这里应该说的是线程栈的内存空间不释放，后续创建线程的时候直接复用内存空间，而不是线程池的概念。



W.jyao

2019-05-11



问一下，如果一个信号是进程共享，比如说kill，那么是主线程会收到这个信号还是这个进程的任意一个线程都有可能收到？麻烦老师解答下



安排

2019-05-10



在proc目录下有每个进程的目录，进程的目录下又包含这个进程下每个线程的目录，进入线程目录中即可查看每个线程的详细信息，这是在命令行通过proc文件系统查看的情况。pthread库应该也提供了api，用来获取每个线程的信息，暂时还没有去查。