Linux进程是如何创建出来的？

Original 张彦飞allen 开发内功修炼 2022-09-28 18:12 Posted on 北京

收录于合集
#开发内功修炼之CPU篇

14个

大家好，我是飞哥！

在 Linux 中，进程是我们非常熟悉的东东了，哪怕是只写过一天代码的人也都用过它。但是你确定它不是你最熟悉的陌生人？我们今天通过深度剖析进程的创建过程，帮助你提高对进程的理解深度。

在这篇文章中，我会用 Nginx 创建 worker 进程的例子作为引入，然后带大家了解一些进程的数据结构 task_struct，最后再带大家看一下 fork 执行的过程。

学习完本文，你将深度理解进程中的那些关键要素，诸如进程地址空间、当前目录、父子进程关系、进程打开的文件 fd 表、进程命名空间等。也能学习到内核在保存已经使用的 pid 号时是如何优化内存占用的。我们展开今天的拆解！

一、Nginx 之 fork 创建 worker

在 Linux 进程的创建中，最核心的就是 fork 系统调用。不过我们先不着急介绍它，先拿多进程服务中的一个经典例子 - Nginx，来看看他是如何使用 fork 来创建 worker 的。

Nginx 服务采用的是多进程方式来工作的，它启动的时候会创建若干个 worker 进程出来，来响应和处理用户请求。创建 worker 子进程的源码位于 nginx 源码的 src/os/unix/ngx_process_cycle.c 文件中。通过循环调用 ngx_spawn_process 来创建 n 个 worker 出来。

```
//file:src/os/unix/ngx_process_cycle.c
static void ngx_start_worker_processes(...)
{
 ...
 for (i = 0; i < n; i++) {
  ngx_spawn_process(cycle, ngx_worker_process_cycle,
```

```
        (void *) (intptr_t) i, "worker process", type);
    ...
  }
}
```

我们在来看下负责具体进程创建的 ngx_spawn_process 函数。

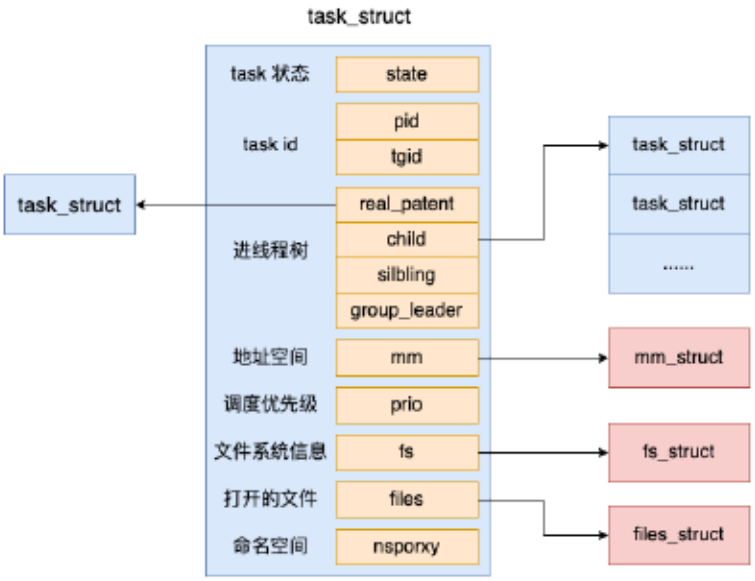
```
//file: src/os/unix/ngx_process.c
ngx_pid_t ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc,...)
{
 pid = fork();
 switch (pid) {
  case -1: //出错了
   ...
  case 0: //子进程创建成功
   ...
   proc(cycle, data);
   break;
 }
 ...
}
```

在 ngx_spawn_process 中调用 fork 来创建进程，创建成功后 Worker 进程就将进入自己的入口函数中开始工作了。

二、Linux 中对进程的表示

在深入理解进程创建之前，我们先来看一下进程的数据结构。

在 Linux 中，是用一个 task_struct 来实现 Linux 进程的（其实 Linux 线程也同样是用 task_struct 来表示的，这个我们以后文章单独再说）。



我们来看看 task_struct 具体的定义，它位于 include/linux/sched.h

```
//file:include/linux/sched.h
struct task_struct {
 //2.1 进程状态
 volatile long state;

 //2.2 进程线程的pid
 pid_t pid;
 pid_t tgid;

 //2.3 进程树关系：父进程、子进程、兄弟进程
 struct task_struct __rcu *parent;
 struct list_head children;
 struct list_head sibling;
 struct task_struct *group_leader;

 //2.4 进程调度优先级
 int prio, static_prio, normal_prio;
 unsigned int rt_priority;

 //2.5 进程地址空间
 struct mm_struct *mm, *active_mm;

 //2.6 进程文件系统信息（当前目录等）
 struct fs_struct *fs;
```

```
    //2.7 进程打开的文件信息
    struct files_struct *files;

    //2.8 namespaces
    struct nsproxy *nsproxy;
}
```

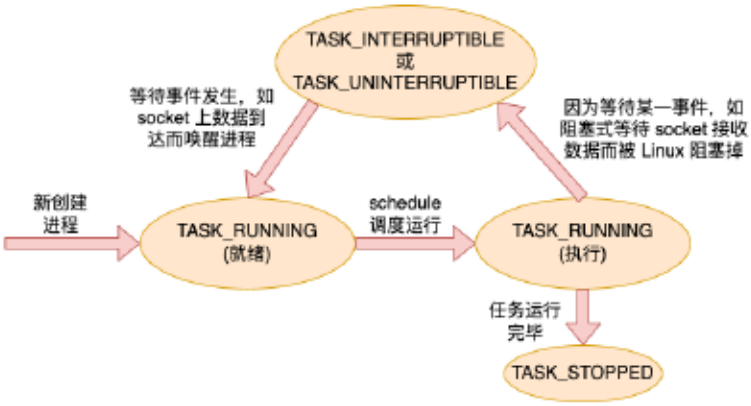
2.1 进程线程状态

进程线程都是有状态的，它的状态就保存在 state 字段中。常见的状态中 TASK_RUNNING 表示进程线程处于就绪状态或者是正在执行。TASK_INTERRUPTIBLE 表示进程线程进入了阻塞状态。

一个任务（进程或线程）刚创建出来的时候是 TASK_RUNNING 就绪状态，等待调度器的调度。调度器执行 schedule 后，任务获得 CPU 后进入 TASK_INTERRUPTIBLE 执行状态进行运行。当需要等待某个事件的时候，例如阻塞式 read 某个 socket 上的数据，但是数据还没有到达的时候，任务进入 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 状态，任务被阻塞掉。

当等待的事件到达以后，例如 socket 上的数据到达了。内核在收到数据后会查看 socket 上阻塞的等待任务队列，然后将之唤醒，使得任务重新进入 TASK_RUNNING 就绪状态。任务如此往复地在各个状态之间循环，直到退出。

一个任务（进程或线程）的大概状态流转图如下。



全部的状态值在 include/linux/sched.h 中进行了定义。

```
//file:include/linux/sched.h
#define TASK_RUNNING            0
```

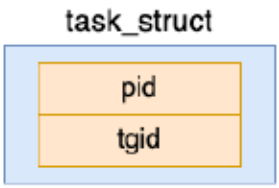
```
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED  4
#define __TASK_TRACED  8
...

#define TASK_DEAD  64
#define TASK_WAKEKILL  128
#define TASK_WAKING  256
#define TASK_PARKED  512
#define TASK_STATE_MAX  1024
......
```

2.2 进程 ID

我们知道，每一个进程都有一个进程 id 的概念。在 task_struct 中有两个相关的字段，分别是 pid 和 tgid。

```
//file:include/linux/sched.h
struct task_struct {
 ......
 pid_t pid;
 pid_t tgid;
}
```

其中 pid 是 Linux 为了标识每一个进程而分配给它们的唯一号码，称做进程 ID 号，简称 PID。对于没有创建线程的进程(只包含一个主线程)来说，这个 pid 就是进程的 PID，tgid 和 pid 是相同的。

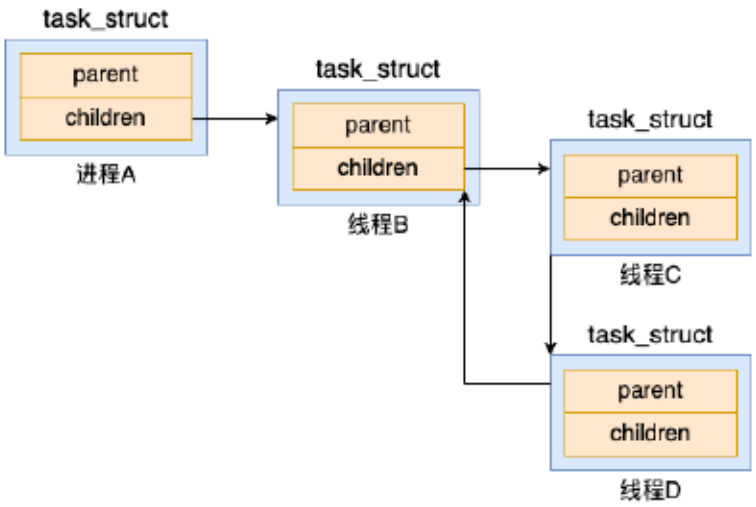


2.3 进程树关系

在 Linux 下所有的进程都是通过一棵树来管理的。在操作系统启动的时候，会创建 init 进程，接下来所有的进程都是由这个进程直接或者间接创建的的。通过 pstree 命令可以查看你当前服务器上的进程树信息。

```
init-+-atd
  |-cron
  |-db2fmcd
  |-db2syscr-+-db2fmp---4*[{db2fmp}]
  |          |-db2fmp---3*[{db2fmp}]
  |          |-db2sysc---13*[{db2sysc}]
  |          |-3*[db2syscr]
  |          |-db2vend
  |          `-{db2syscr}
  |-dbus-daemon
```

那么，这棵进程树就是由 task_struct 下的 parent、children、sibling 等字段来表示的。这几个字段将系统中的所有 task 串成了一棵树。



2.4 进程调度优先级

在 task_struct 中有几个字段是表示进程优先级的，在进程调度的时候会根据这几个字段来决定优先让哪个任务（进程或线程）开始执行。

- static_prio: 用来保存静态优先级，可以调用 nice 系统直接来修改取值范围为 100~139
- rt_priority: 用来保存实时优先级,取值范围为 0~99
- prio: 用来保存动态优先级
- normal_prio: 它的值取决于静态优先级和调度策略

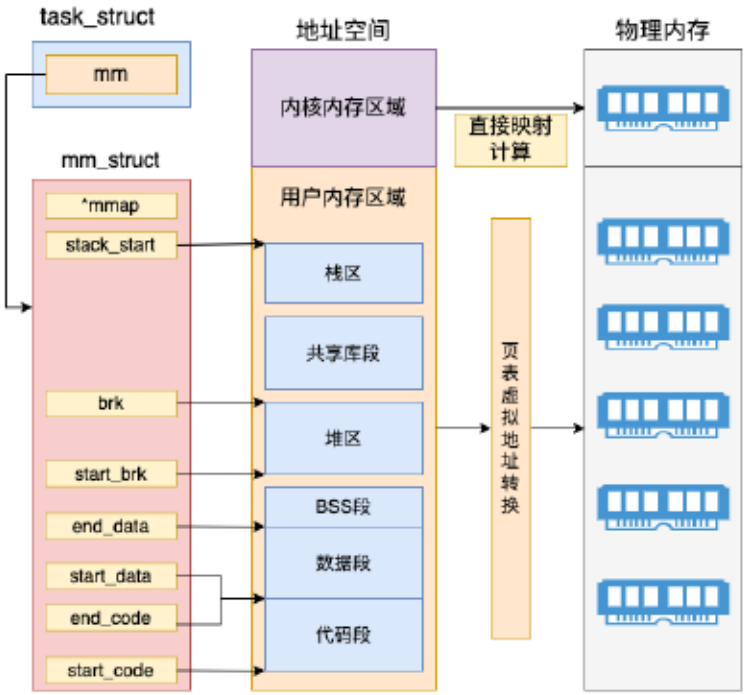
2.5 进程地址空间

对于用户进程来讲，内存描述符 mm_struct(mm 代表的是 memory descriptor)是非常核心的数据结构。整个进程的虚拟地址空间部分都是由它来表示的。

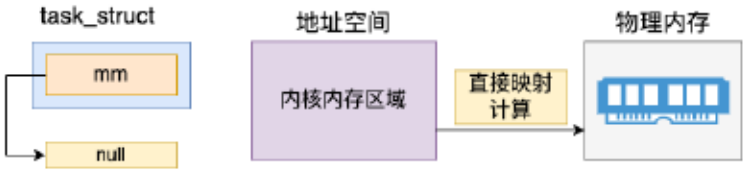
进程在运行的时候，在用户态其所需要的代码，全局变量数据，以及 mmap 内存映射等全部都是通过 mm_struct 来进行内存查找和寻址的。这个数据结构的定义位于 include/linux/mm_types.h 文件下。

```
//file:include/linux/mm_types.h
struct mm_struct {
 struct vm_area_struct * mmap;  /* list of VMAs */
 struct rb_root mm_rb;

 unsigned long mmap_base;  /* base of mmap area */
 unsigned long task_size;  /* size of task vm space */
 unsigned long start_code, end_code, start_data, end_data;
 unsigned long start_brk, brk, start_stack;
 unsigned long arg_start, arg_end, env_start, env_end;
}
```

其中 start_code、end_code 分别指向代码段的开始与结尾、start_data 和 end_data 共同决定数据段的区域、start_brk 和 brk 中间是堆内存的位置、start_stack 是用户态堆栈的起始地址。整个 mm_struct 和地址空间、页表、物理内存的关系如下图。



在内核内存区域，可以通过直接计算得出物理内存地址，并不需要复杂的页表计算。而且最重要的是所有内核进程、以及用户进程的内核态，这部分内存都是共享的。



另外要注意的是，mm（mm_struct）表示的是虚拟地址空间。而对于内核线程来说，是没有用户态的虚拟地址空间的。所以内核线程的 mm 的值是 null。

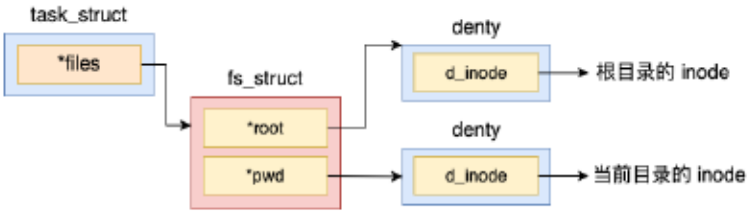
2.6 进程文件系统信息（当前目录等）

进程的文件位置等信息是由 fs_struct 来描述的，它的定义位于 include/linux/fs_struct.h 文件中。

```
//file:include/linux/fs_struct.h
struct fs_struct {
 ...
 struct path root, pwd;
};

//file:include/linux/path.h
struct path {
 struct vfsmount *mnt;
```

```
 struct dentry *dentry;
};
```

通过以上代码可以看出，在 fs_struct 中包含了两个 path 对象，而每个 path 中都指向了一个 struct dentry。在 Linux 内核中，denty 结构是对一个目录项的描述。



拿 pwd 来举例，该指针指向的是进程当前目录所处的 denty 目录项。假如我们在 shell 进程中执行 pwd，或者用户进程查找当前目录下的配置文件的时候，都是通过访问 pwd 这个对象，进而找到当前目录的 denty 的。

2.7 进程打开的文件信息

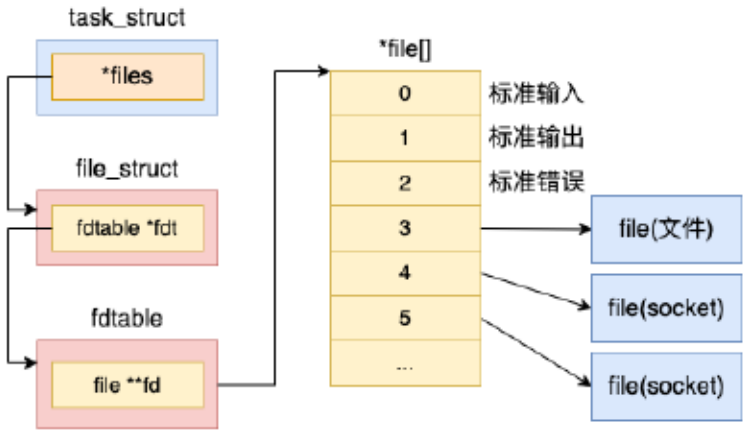
每个进程用一个 files_struct 结构来记录文件描述符的使用情况， 这个 files_struct 结构称为用户打开文件表。它的定义位于 include/linux/fdtable.h。

> 注意：飞哥用的内核源码一直是 3.10.0，所以本文也不例外。不同版本的源码这里稍微可能有些出入。

```
//file:include/linux/fdtable.h
struct files_struct {
 ......
 //下一个要分配的文件句柄号
 int next_fd;

 //fdtable
 struct fdtable __rcu *fdt;
}

struct fdtable {
 //当前的文件数组
 struct file   rcu **fd;
```

```
    ......
};
```

在 files_struct 中，最重要的是在 fdtable 中包含的 file **fd 这个数组。这个数组的下标就是文件描述符，其中 0、1、2 三个描述符总是默认分配给标准输入、标准输出和标准错误。这就是你在 shell 命令中经常看到的 2>&1 的由来。这几个字符的含义就是把标准错误也一并打到标准输出中来。



在数组元素中记录了当前进程打开的每一个文件的指针。这个文件是 Linux 中抽象的文件，可能是真的磁盘上的文件，也可能是一个 socket。

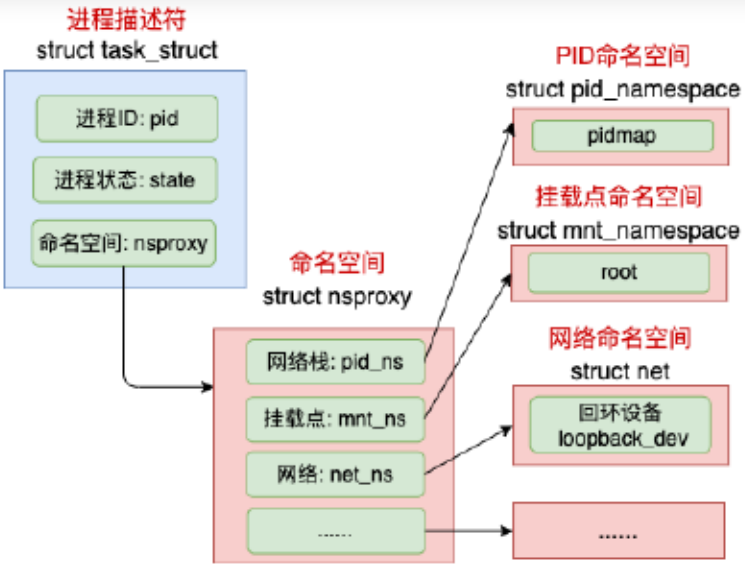
2.8 namespaces

在 Linux 中，namespace 是用来隔离内核资源的方式。通过 namespace 可以让一些进程只能看到与自己相关的一部分资源，而另外一些进程也只能看到与它们自己相关的资源，这两拨进程根本就感觉不到对方的存在。

具体的实现方式是把一个或多个进程的相关资源指定在同一个 namespace 中，而进程究竟是属于哪个 namespace，都是在 task_struct 中由 *nsproxy 指针表明了这个归属关系。

```
//file:include/linux/nsproxy.h
struct nsproxy {
 atomic_t count;
 struct uts_namespace *uts_ns;
 struct ipc_namespace *ipc_ns;
 struct mnt_namespace *mnt_ns;
 struct pid_namespace *pid_ns;
```

```
    struct net      *net_ns;
};
```



命名空间包括PID命名空间、挂载点命名空间、网络命名空间等多个。飞哥在咱们「开发内功修炼」前面的一篇文章《动手实验+源码分析，彻底弄懂Linux网络命名空间》这一文中详细介绍过网络命名空间，感兴趣的同学可以详细阅读。

三、解密 fork 系统调用

前面我们看了 Nginx 使用 fork 来创建 worker 进程，也了解了进程的数据结构 task_struct ，我们再来看看 fork 系统调用的内部逻辑。

这个 fork 在内核中是以一个系统调用来实现的，它的内核入口是在 kernel/fork.c 下。

```
//file:kernel/fork.c
SYSCALL_DEFINE0(fork)
{
 return do_fork(SIGCHLD, 0, 0, NULL, NULL);
}
```

这里注意下调用 do_fork 时传入的第一个参数，这个参数是一个 flag 选项。它可以传入的值包括 CLONE_VM、CLONE_FS 和 CLONE_FILES 等等很多，但是这里只传了一个 SIGCHLD（子进程在终止后发送 SIGCHLD 信号通知父进程），并没有传 CLONE_FS 等其它 flag。

```
//file:include/uapi/linux/sched.h
//cloning flags:
...
#define CLONE_VM 0x00000100
#define CLONE_FS 0x00000200
#define CLONE_FILES 0x00000400
...
```

在 do_fork 的实现中，核心是一个 copy_process 函数，它以拷贝父进程的方式来生成一个新的 task_struct 出来。

```
//file:kernel/fork.c
long do_fork(unsigned long clone_flags,
    unsigned long stack_start,
    unsigned long stack_size,
    int __user *parent_tidptr,
    int __user *child_tidptr)
{
 //复制一个 task_struct 出来
 struct task_struct *p;
 p = copy_process(clone_flags, stack_start, stack_size,
    child_tidptr, NULL, trace);

 //子任务加入到就绪队列中去，等待调度器调度
 wake_up_new_task(p);
 ...
}
```

在创建完毕后，调用 wake_up_new_task 将新创建的任务添加到就绪队列中，等待调度器调度执行。

copy_process 的代码很长，我对其进行了一定程度的精简，参加下面的代码。

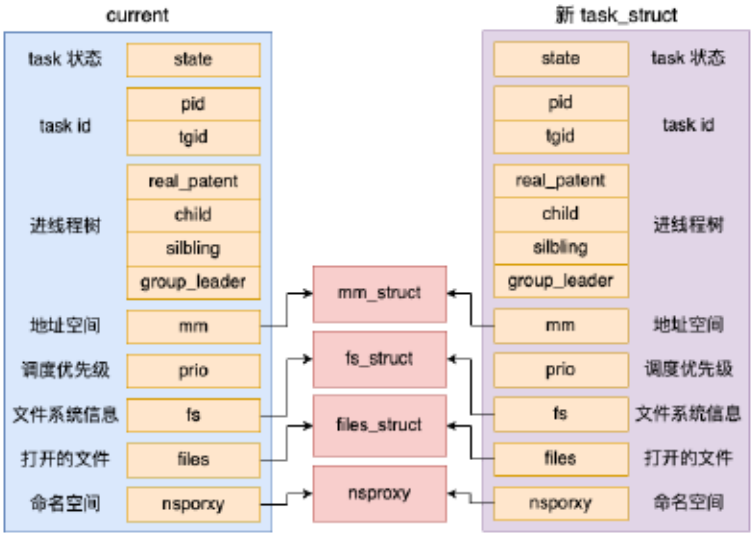
```
//file:kernel/fork.c
static struct task_struct *copy_process(...)
```

```
{
 //3.1 复制进程 task_struct 结构体
 struct task_struct *p;
 p = dup_task_struct(current);
 ...

 //3.2 拷贝 files_struct
 retval = copy_files(clone_flags, p);

 //3.3 拷贝 fs_struct
 retval = copy_fs(clone_flags, p);

 //3.4 拷贝 mm_struct
 retval = copy_mm(clone_flags, p);

 //3.5 拷贝进程的命名空间 nsproxy
 retval = copy_namespaces(clone_flags, p);

 //3.6 申请 pid && 设置进程号
 pid = alloc_pid(p->nsproxy->pid_ns);
 p->pid = pid_nr(pid);
 p->tgid = p->pid;
 if (clone_flags & CLONE_THREAD)
  p->tgid = current->tgid;

 ......
}
```

可见，copy_process 先是复制了一个新的 task_struct 出来，然后调用 copy_xxx 系列的函数对 task_struct 中的各种核心对象进行拷贝处理，还申请了 pid。接下来我们分小节来查看该函数的每一个细节。

3.1 复制进程 task_struct 结构体

注意一下，上面调用 dup_task_struct 时传入的参数是 current，它表示的是当前进程。在 dup_task_struct 里，会申请一个新的 task_struct 内核对象，然后将当前进程复制给它。需要注意的是，这次拷贝只会拷贝 task_struct 结构体本身，它内部包含的 mm_struct 等成员只是复制了指针，仍然指向和 current 相同的对象。



我们来简单看下具体的代码。

```
//file:kernel/fork.c
static struct task_struct *dup_task_struct(struct task_struct *orig)
{
 //申请 task_struct 内核对象
 tsk = alloc_task_struct_node(node);

 //复制 task_struct
 err = arch_dup_task_struct(tsk, orig);
 ...
}
```

其中 alloc_task_struct_node 用于在 slab 内核内存管理区中申请一块内存出来。关于 slab 机制请参考- **内核内存管理**

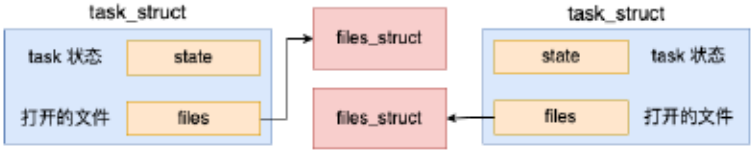
```
//file:kernel/fork.c
static struct kmem_cache *task_struct_cachep;
static inline struct task_struct *alloc_task_struct_node(int node)
{
 return kmem_cache_alloc_node(task_struct_cachep, GFP_KERNEL, node);
}
```

申请完内存后，调用 arch_dup_task_struct 进行内存拷贝。

```
//file:kernel/fork.c
int arch_dup_task_struct(struct task_struct *dst,
         struct task_struct *src)
{
 *dst = *src;
 return 0;
}
```

3.2 拷贝 files_struct

由于进程之间都是独立的，所以创建出来的新进程需要拷贝一份独立的 files 成员出来。



我们看 copy_files 是如何申请和拷贝 files 成员的。

```
//file:kernel/fork.c
static int copy_files(unsigned long clone_flags, struct task_struct *tsk)
{
 struct files_struct *oldf, *newf;
 oldf = current->files;

 if (clone_flags & CLONE_FILES) {
  atomic_inc(&oldf->count);
  goto out;
 }
 newf = dup_fd(oldf, &error);
 tsk->files = newf;
 ...
}
```

看上面代码中判断了是否有 CLONE_FILES 标记，如果有的话就不执行 dup_fd 函数了，增加个引用计数就返回了。前面我们说了，do_fork 被调用时并没有传这个标记。所以还是会执行到 dup_fd 函数：

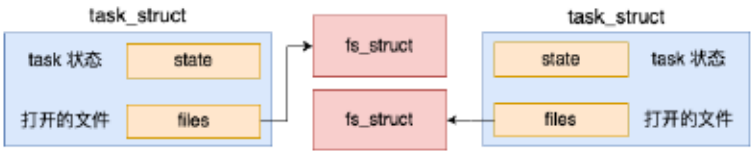
```
//file:fs/file.c
```

```
struct files_struct *dup_fd(struct files_struct *oldf, ...)
{
 //为新 files_struct 申请内存
 struct files_struct *newf;
 newf = kmem_cache_alloc(files_cachep, GFP_KERNEL);

 //初始化 & 拷贝
 new_fdt->max_fds = NR_OPEN_DEFAULT;
 ...
}
```

这个函数就是到内核中申请一块内存出来，保存 files_struct 使用。然后对新的 files_struct 进行各种初始化和拷贝。至此，新进程有了自己独立的 files 成员了。

3.3 拷贝 fs_struct

同样，新进程也需要一份独立的文件系统信息 - fs_struct 成员的。



我们来看 copy_fs 是如何申请和初始化 fs_struct 的。

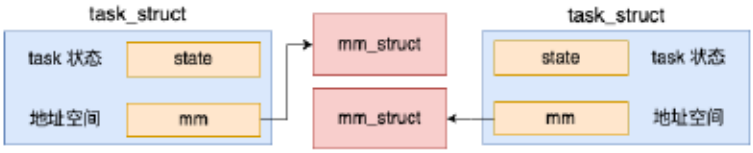
```
//file:kernel/fork.c
static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
{
 struct fs_struct *fs = current->fs;
 if (clone_flags & CLONE_FS) {
  fs->users++;
  return 0;
 }
 tsk->fs = copy_fs_struct(fs);
 return 0;
}
```

在创建进程的时候，没有传递 CLONE_FS 这个标志，所会进入到 copy_fs_struct 函数中申请新的 fs_struct 并进行赋值。

```
//file:fs/fs_struct.c
struct fs_struct *copy_fs_struct(struct fs_struct *old)
{
 //申请内存
 struct fs_struct *fs = kmem_cache_alloc(fs_cachep, GFP_KERNEL);

 //赋值
 fs->users = 1;
 fs->root = old->root;
 fs->pwd = old->pwd;
 ...
 return fs;
}
```

3.4 拷贝 mm_struct

前面我们说过，对于进程来讲，地址空间是一个非常重要的数据结构。而且进程之间地址空间也必须是要隔离的，所以还会新建一个地址空间。



创建地址空间的操作是在 copy_mm 中执行的。

```
//file:kernel/fork.c
static int copy_mm(unsigned long clone_flags, struct task_struct *tsk)
{
 struct mm_struct *mm, *oldmm;
 oldmm = current->mm;

 if (clone_flags & CLONE_VM) {
  atomic_inc(&oldmm->mm_users);
  mm = oldmm;
  goto good_mm;
 }
 mm = dup_mm(tsk);
good_mm:
 return 0;
}
```

do_fork 被调用时也没有传 CLONE_VM，所以会调用 dup_mm 申请一个新的地址空间出来。

```
//file:kernel/fork.c
struct mm_struct *dup_mm(struct task_struct *tsk)
{
 struct mm_struct *mm, *oldmm = current->mm;
 mm = allocate_mm();
 memcpy(mm, oldmm, sizeof(*mm));
 ...
}
```

在 dup_mm 中，通过 allocate_mm 申请了新的 mm_struct，而且还将当前进程地址空间 current->mm 拷贝到新的 mm_struct 对象里了。

地址空间是进程线程最核心的东西，每个进程都有独立的地址空间

3.5 拷贝进程的命名空间 nsproxy

在创建进程或线程的时候，还可以让内核帮我们创建独立的命名空间。在默认情况下，创建进程没有指定命名空间相关的标记，因此也不会创建。新旧进程仍然复用同一套命名空间对象。

3.6 申请pid

接下来 copy_process 还会进入 alloc_pid 来为当前任务申请 PID。

```
//file:kernel/fork.c
static struct task_struct *copy_process(...)
{
 ...
 //申请pid
```

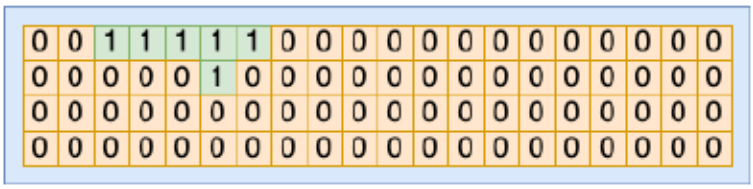
```
 pid = alloc_pid(p->nsproxy->pid_ns);

 //赋值
 p->pid = pid_nr(pid);
 p->tgid = p->pid;
 ...
}
```

注意下，在调用 alloc_pid 的时候，其参数传递的是新进程的 pid namespace。我们来深看一下 alloc_pid 的执行逻辑。

```
//file:kernel/pid.c
struct pid *alloc_pid(struct pid_namespace *ns)
{
 //申请 pid 内核对象
 pid = kmem_cache_alloc(ns->pid_cachep, GFP_KERNEL);
 if (!pid)
  goto out;

 //调用到alloc_pidmap来分配一个空闲的pid编号
 //注意，在每一个命令空间中都需要分配进程号
 tmp = ns;
 pid->level = ns->level;
 for (i = ns->level; i >= 0; i--) {
  nr = alloc_pidmap(tmp);
  pid->numbers[i].nr = nr;
  ...
 }
 ...
 return pid
}
```

这里的 PID 并不是一个整数，而是一个结构体，所以先试用 kmem_cache_alloc 把它申请出来。接下来调用 alloc_pidmap 到 pid 命名空间中申请一个 pid 号出来，申请完后赋值记录。

回顾我们开篇提到的一个问题：操作系统是如何记录使用过的进程号的？在 Linux 内部，为了节约内存，进程号是通过 bitmap 来管理的。



在每一个 pid 命名空间内部，会有一个或者多个页面来作为 bitmap。其中每一个 bit 位（注意是 bit 位，不是字节）的 0 或者 1 的状态来表示当前序号的 pid 是否被占用。

```
//file:include/linux/pid_namespace.h
#define BITS_PER_PAGE  (PAGE_SIZE * 8)
#define PIDMAP_ENTRIES  ((PID_MAX_LIMIT+BITS_PER_PAGE-1)/BITS_PER_PAGE)
struct pid_namespace {
 struct pidmap pidmap[PIDMAP_ENTRIES];
 ...
}
```

在 alloc_pidmap 中就是以 bit 的方式来遍历整个 bitmap，找到合适的未使用的 bit，将其设置为已使用，然后返回。

```
//file:kernel/pid.c
static int alloc_pidmap(struct pid_namespace *pid_ns)
{
 ...
 map = &pid_ns->pidmap[pid/BITS_PER_PAGE];
}
```

在各种语言中，一般一个 int 都是 4 个字节，换算成 bit 就是 32 bit。而使用这种 bitmap 的思想的话，只需要一个 bit 就可以表示一个整数，相当的节约内存。所以，在很多超大规模数据处理中都会用到这种思想来进行优化内存占用的。

3.7 进入就绪队列

当 copy_process 执行完毕的时候，表示新进程的一个新的 task_struct 对象就创建出来了。接下来内核会调用 wake_up_new_task 将这个新创建出来的子进程添加到就绪队列中等待调度。

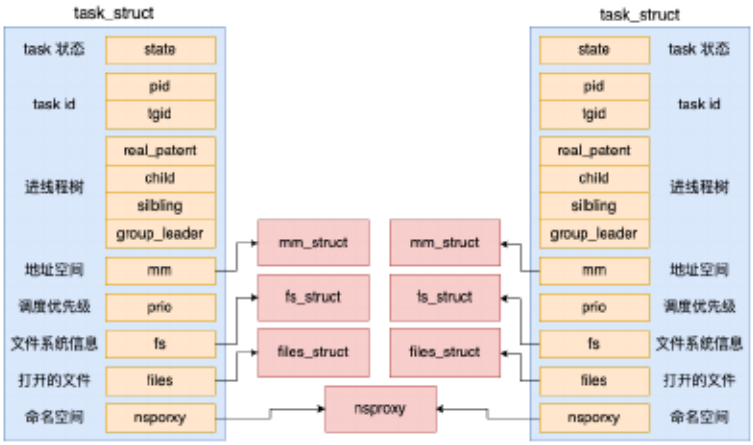
```
//file:kernel/fork.c
long do_fork(...)
{
 //复制一个 task_struct 出来
 struct task_struct *p;
 p = copy_process(clone_flags, stack_start, ...);

 //子任务加入到就绪队列中去，等待调度器调度
 wake_up_new_task(p);
 ...
}
```

等操作系统真正调度开始的时候，子进程中的代码就可以真正开始执行了。

四、总结

在这篇文章中，我用 Nginx 创建 worker 进程的例子作为引入，然后带大家了解一些进程的数据结构 task_struct，最后又带大家看一下 fork 执行的过程。

在 fork 创建进程的时候，地址空间 mm_struct、挂载点 fs_struct、打开文件列表 files_struct 都要是独立拥有的，所以都去申请内存并初始化了它们。但由于今天我们的例子父子进程是同一个命名空间，所以 nsproxy 还仍然是共用的。



其中 mm_struct 是一个非常核心的数据结构，用户进程的虚拟地址空间就是用它来表示的。对于内核线程来讲，不需要虚拟地址空间，所以 mm 成员的值为 null。

另外还学到了内核是用 bitmap 来管理使用和为使用的 pid 号的，这样做的好处是极大地节约了内存开销。而且由于数据存储的足够紧凑，遍历起来也是非常的快。一方面原因是数据

小，加载起来快。另外一方面是会加大提高 CPU 缓存的命中率，访问非常快。

今天的进程创建过程就学习完了。不过细心的同学可能发现了，我们这里只介绍了子进程的调用。但是对于 Nginx 主进程如何加载起来执行的还没有讲到。我们将来还会展开叙述，敬请期待！

最后，欢迎你将这个硬核的技术公众号分享给你关系最好的朋友，一定对他也会有很大帮助的！

开发内功修炼

长按二维码识别关注