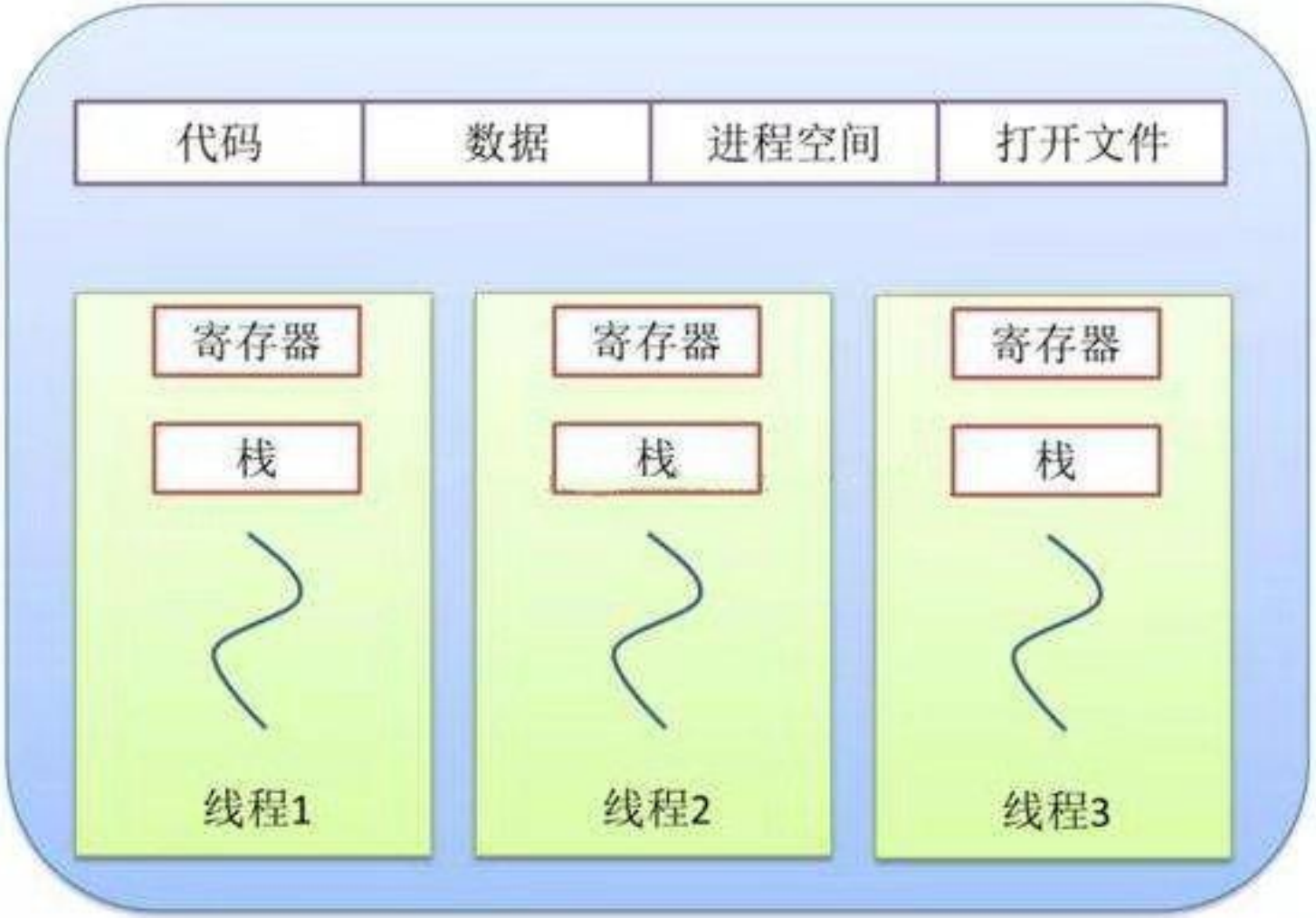
3.2 进程创建

西安邮电大学

进程和线程

- 目前在用户态程序开发中，不仅仅涉及到进程，还涉及到线程和协程，他们到底是如何创建的，为什么创建了一个进程或者线程后觉得自己对其没有控制权，这是因为创建这件事完全由操作系统操控，你只是发出一个创建的请求，然后，整个生孩子这件事就交给操作系统了，如果你对这个过程不了解，那么，一旦程序在运行的过程中出问题，你就可能就束手无策。但这个过程实际上非常复杂，那么，如何入手，本讲将给予简要介绍。

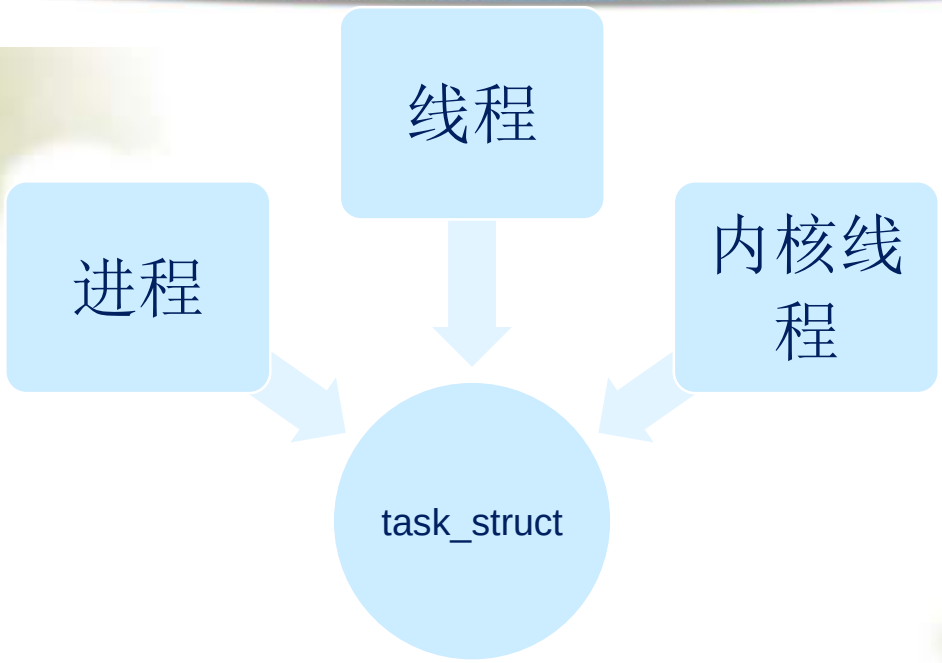
进程和线程



进程和线程

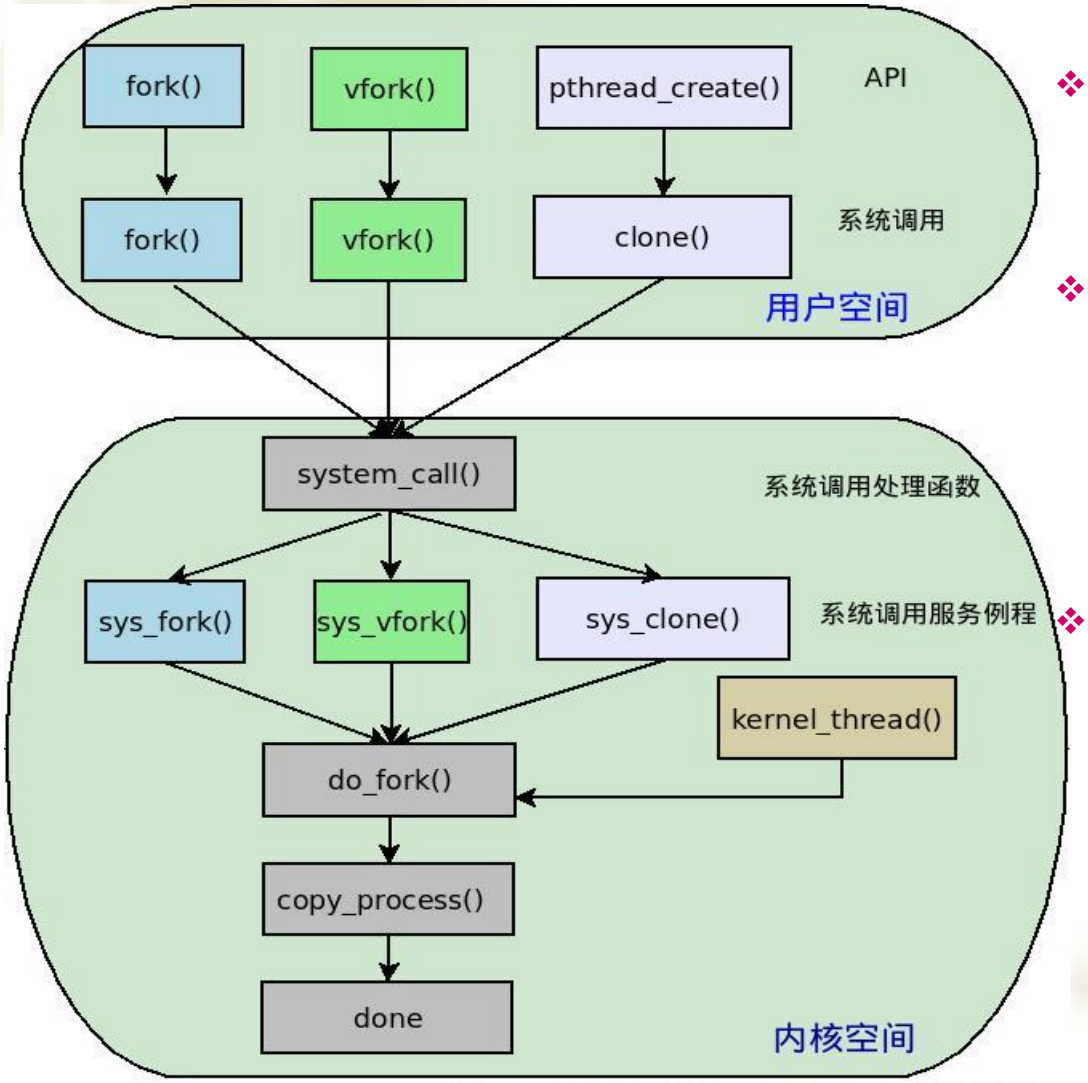
- 我们知道进程是系统资源分配的基本单位，线程是独立运行的基本单位。
- 但这种说法过于笼统，进程的资源到底有哪些，如何体现，线程为什么是轻量级的运行单位，如何体现？
- 从图中可以看出，进程和线程几乎共享所有的资源，包括代码，数据，进程空间，打开的文件等，线程只拥有自己的寄存器和栈。这些概念上的代码段，数据段，进程空间，打开的文件，在内核中是如何表示的？

task_struct结构的统一性与多样性



- 内核如何对待进程，线程和内核线程呢？Linux内核坚持平等的原则，对它们一视同仁，即内核使用唯一的数据结task_struct来分别表示他们；也使用相同的调度算法对这三者进行调度；尽管表面看起它们很不一样，但是在内核中最终都通过do_fork()分别创建这样处理对内核来说简单方便，在统一的基础上又保持各自的特性，这是如何做到的呢?

进程的API实现



- 我们站在用户态函数库看过去，创建进程和创建线程调用了不同的函数，
- 分别为fork()和pthread_create()而对应的系统调用为fork()和clone（），vfork与fork类似，后面会讲到二者的差异。
- 所有的系统调用进入内核只有一个入口，进去以后，又似乎分道扬镳了，各自有自己的服务例程，但分手只是暂时的，归到一处是最终的选择，因此，do_fork就成为它们的聚合点。

do_fork()

- do_fork()在内核中是怎样的原型

```
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             unsigned long stack_size,
              int __user *parent_tidptr,
             int __user *child_tidptr)
```

看这一堆参数，是否有眩晕的感觉，在用户态调用fork()时，轻松自由，一个参数也不需要给，为什么进入内核后这么麻烦，在这里，我们就知道你的风花雪月到底是谁帮你当担的。

三个系统调用如何调用do_fork

```
int sys_fork(struct pt_regs *regs)
{
        return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}

int sys_vfork(struct pt_regs *regs)
{
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->sp, regs, 0,
                       NULL, NULL);
}

long
sys_clone(unsigned long clone_flags, unsigned long newsp,
          void __user *parent_tid, void __user *child_tid, struct pt_regs *regs)
{
        if (!newsp)
                newsp = regs->sp;
        return do_fork(clone_flags, newsp, regs, 0, parent_tid, child_tid);
}
```

fork的实现

```
int sys_fork(struct pt_regs *regs)
{
    return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}
```

- 先看fork调用的do_fork，除了SIGCHLD参数外，有三个参数就是空手而来，有两个参数似乎也没有明确的目标。但作为子进程，它完全有自己的个性的，根本不想共享父进程的任何资源，而是让父亲把他所有资源给自己复制一份，父亲真的就给他复制一份吗，老爸没有那么傻，而是假装复制了一下，也就是用一个指针指过去而已，等真正需要的时候，比如，要写一个页面，这时，写时复制技术就登场了，只有父子进程中不管谁想写一个页面时，这个页面才被复制一份。

vfork的实现

```
int sys_vfork(struct pt_regs *regs)
{
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->sp, regs, 0,
                       NULL, NULL);
}
```

- vfork()，比fork还狡猾，直接传递了两个标志过去，
- 第一个标志（CLONE_VFORK），儿子优先，老爸等着。于是父进程就去睡觉，等子进程结束才能醒来。
- 第二个标志（CLONE_VM）儿子干脆与父亲待在一个进程的地址空间中，对，就是共享父进程的内存地址空间（父进程的页表项除外）。
- vfork()看起来很聪明，但从聪明反被聪明误，因为写实复制技术的招数更高，也就是更高效，因此，它没有了生存空间，直接被取代了。

clone的实现

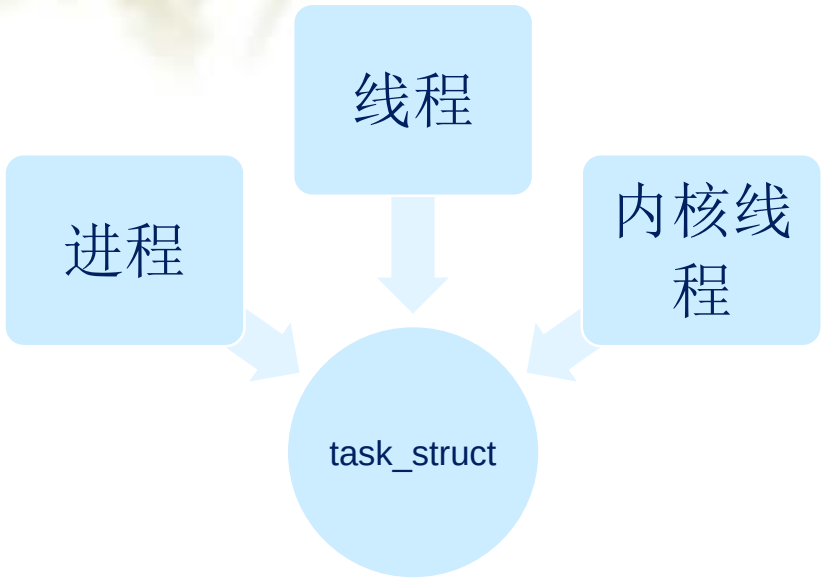
```
long
sys_clone(unsigned long clone_flags, unsigned long newsp,
          void __user *parent_tid, void __user *child_tid, struct pt_regs *regs)
{
        if (!newsp)
                newsp = regs->sp;
        return do_fork(clone_flags, newsp, regs, 0, parent_tid, child_tid);
}
```

- clone()：克隆？对，就是克隆技术，线程就是这么诞生的。
- 怎么克隆，听起来很神秘，实际上，很简单，无非就是传了一堆参数，告诉老爸，你这我要共享，你那我也都要共享，于是老爸的地址空间，文件系统，打开的文件，信号处理函数等就都被儿子一句话说过来了，看过来就是这四个参数：
- CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND

内核线程的创建

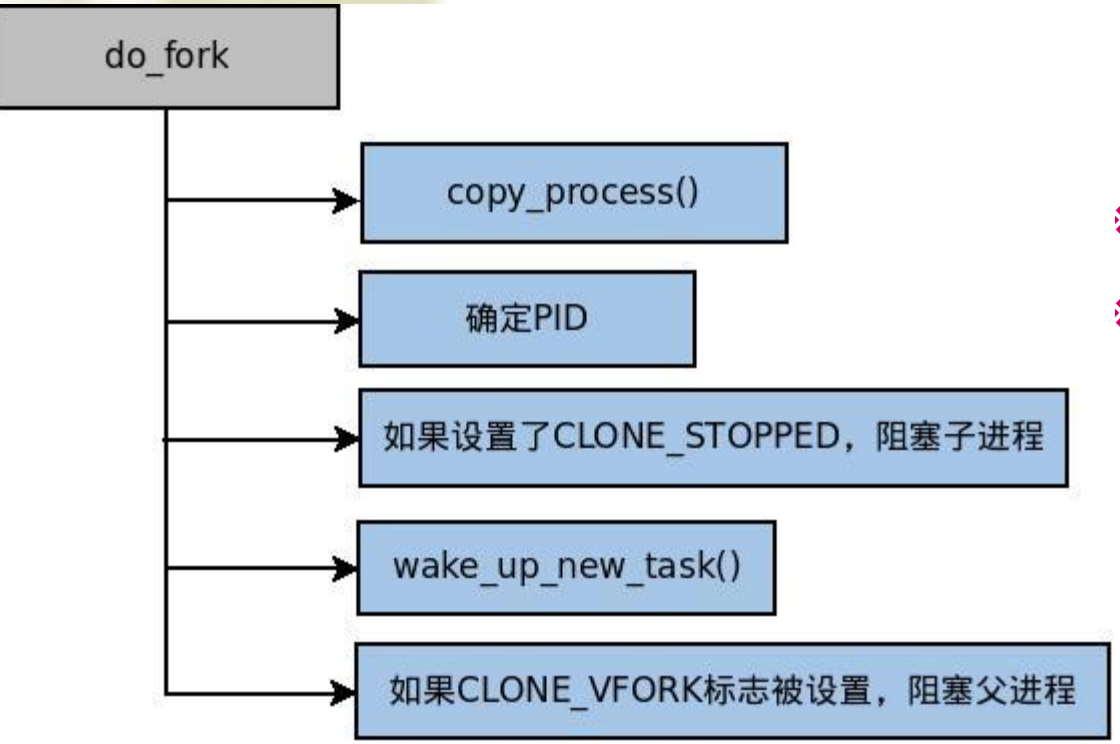
- 如果我是一个内核线程，我的出生是否更有优势呢？没错，因为用户空间对我来说根本就没有意义，我根本就不知道它的存在。
- 该调用哪个函数创建呢？
- 早期内核中创建内核线程是通过kernel_thread()而创建的，目前内核中调用 kthread_create()创建的，其本质也是向do_fork()提供特定的flags标志而创建的。

task_struct带来的统一性



- 到底是谁给我们乡（进程，线程和内核线程）的诞生带来了方便，说到底，还是因为我们站在同一个战壕中，对，就是那个task_struct结构。
- 由此，我们的生命历程具有了诸多相似，不管是被调度到CPU上去跑，还是分配各种资源，到最终的诞生都是调用了相同的数do_fork()。
- 能不能看看do_fork()的代码流程是什么样的？

do_fork()代码流程



- 1. 调用copy_process()复制父进程的进程控制块。
- 2. 获得子进程的pid。
- 3. 如果设置了暂停标志，则子进程的状态被设置为暂停。否则，通过唤醒函数将子进程的状态设置为就绪，并且将子进程加入就绪队列。
- 4. 如果使用vfork()创建进程，则阻塞父进程。

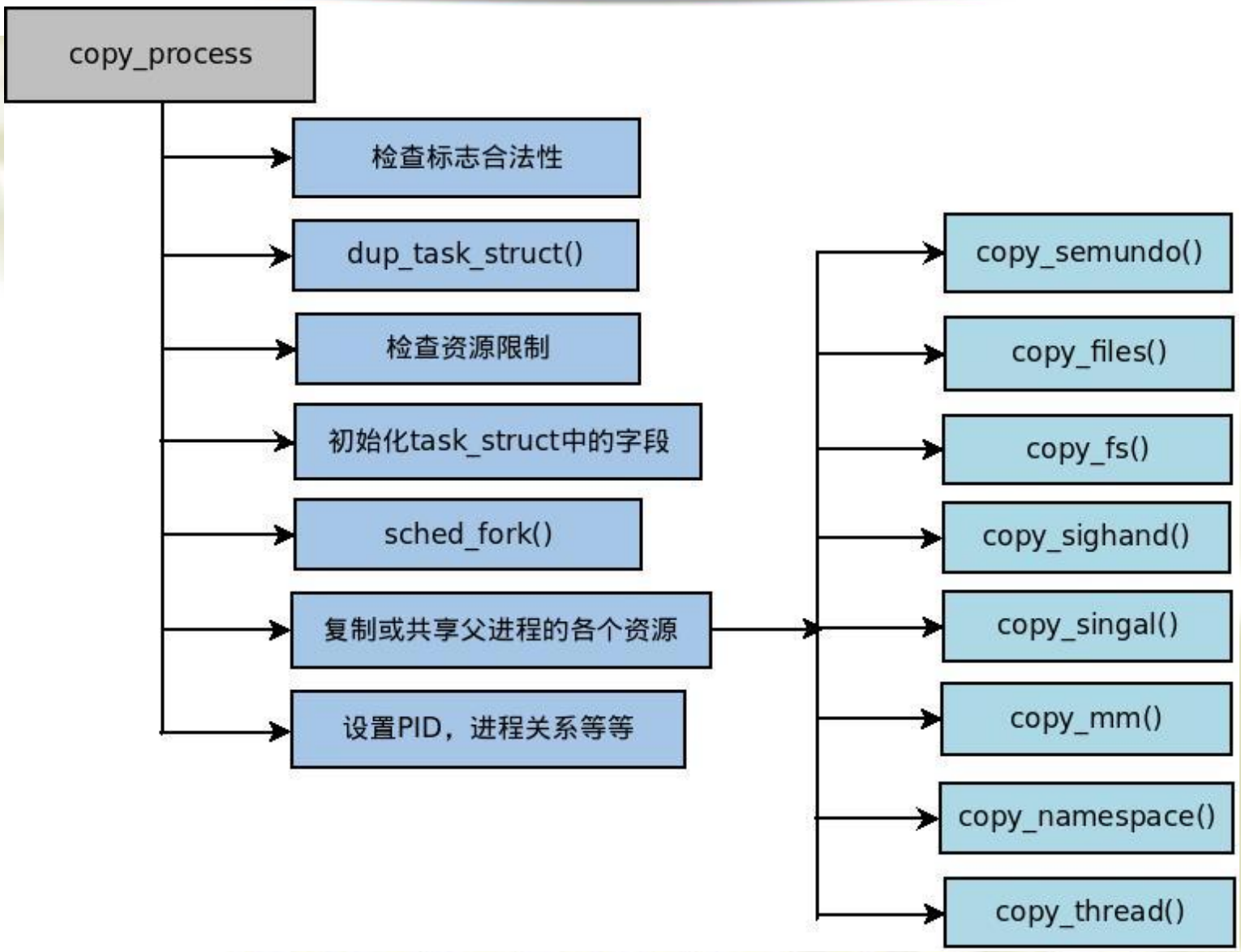
copy_process()

- copy_process()主要用于创建进程控制块以及子进程执行时所需要的其他数据结构。该函数的参数与do_fork()的参数大致相同，并添加了子进程的pid。

```
static struct task_struct *copy_process(unsigned long clone_flags,
                                        unsigned long stack_start,
                                        struct pt_regs *regs,
                                        unsigned long stack_size,
                                        int __user *child_tidptr,
                                        struct pid *pid,
                                        int trace)
```

- copy_process()所做的处理必须考虑到各种可能的情况，这些特殊情况即通过clone_flags来具体体现。下面忽略特殊情况，给出一般的执行过程。

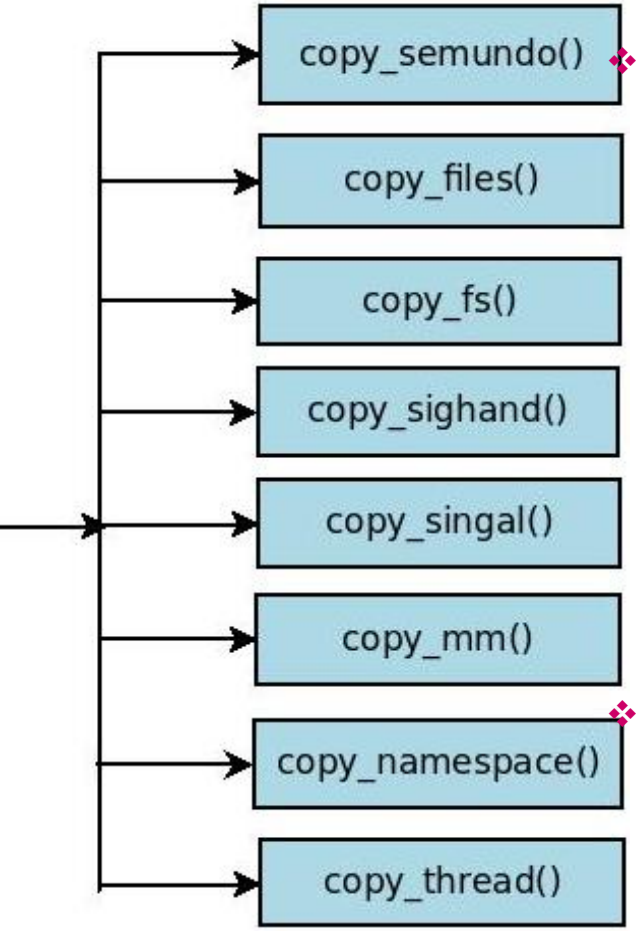
copy_process()代码流程图



copy_process()主要实现过程

- 这个函数主要是为子进程创建父进程PCB的副本，然后对子进程PCB中的各个字段进行初始化。同时，子进程对父进程的各种资源进行复制或共享，具体取决于clone_flags所设置的标志。每种资源的复制或共享都通过形如copy_XYZ()这样的函数完成，当然子进程获得了新的pid。

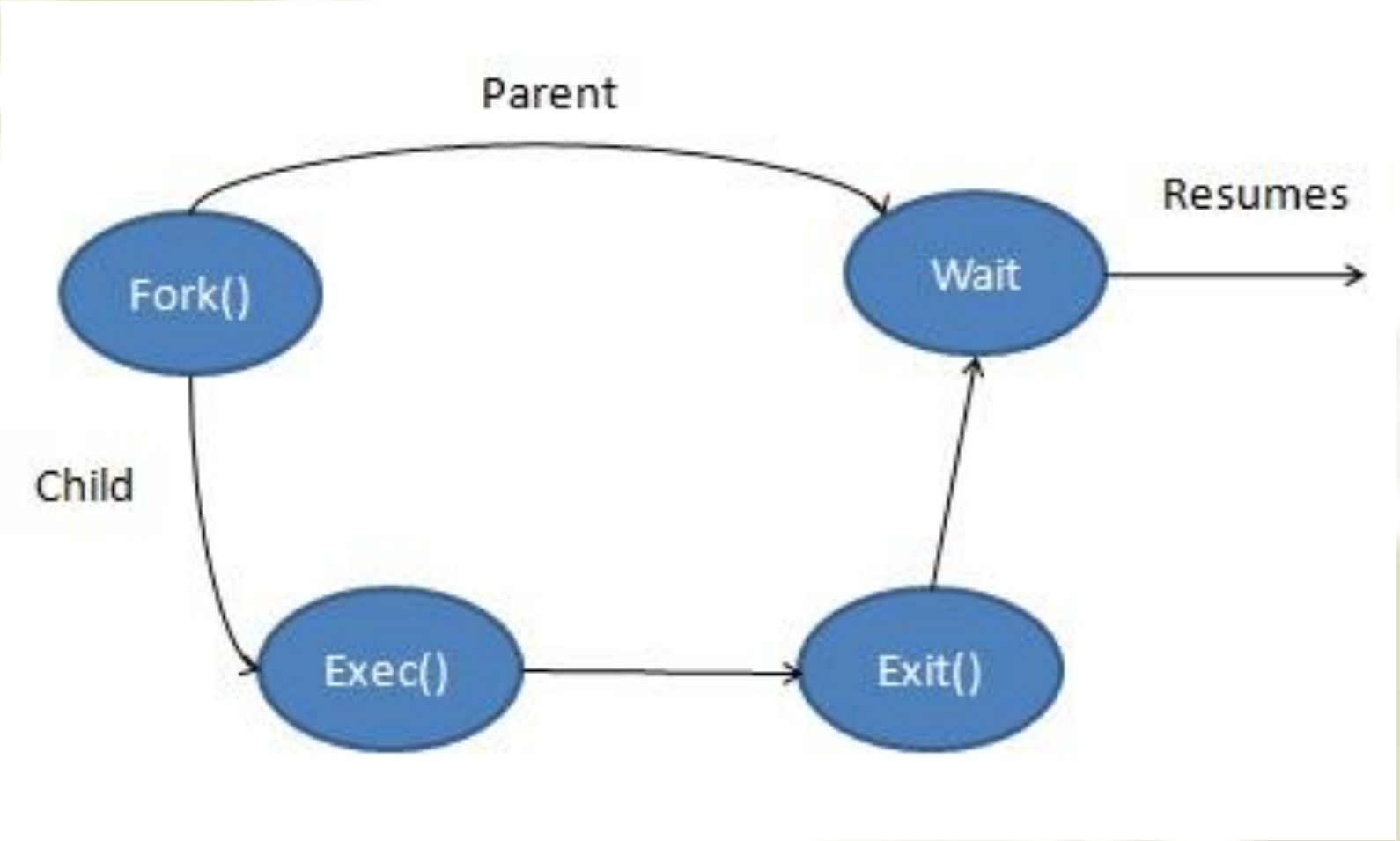
父子进程的资源共享-copy_xyz()



父子进程通过copy_xyz（）这样的函数共享各种资源，比如，打开的文件，所在的文件系统，进程的地址空间，信号，命名空间等等。如果进入这些函数去阅读，几乎延伸到内核的各个子系统，因此，当你还对这些内容还没有了解的时候，先粗线条的阅读，等你了解了各个子系统后，再回头阅读相关内容，会发现一个进程的创建牵一发而动全局，把相关的知识都可以穿起来了，那些零零散散的知识通过fork而聚集在一起，知识的活力也充分的体现出来了。

这里为什么不给出具体的源代码，因为内核版本不同，代码一直在变化，但是，当你对各个对象之间的关系理清楚以后去读阅码，这些源码就只是一种实现了。

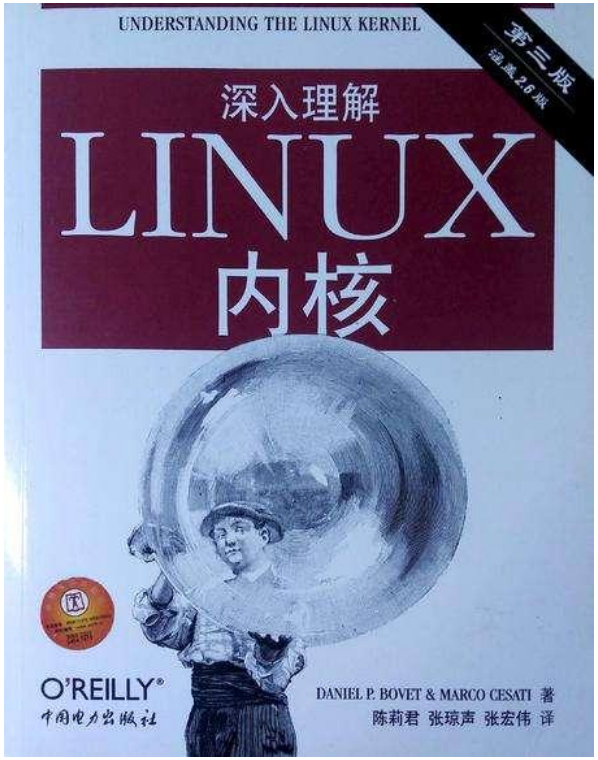
进程的生命周期



进程的生命周期

- 这里比较详细的介绍了fork的创建过程，与之相应的还有exec,wait,exit三个系统调用，就不一一详细介绍了，他们之间是如何配合的？
- 下面让我们用一些形象的比喻，来对进程短暂的一生作一个小小的总结：
- 随着一句fork，一个新进程呱呱落地，但这时它只是老进程的一个克隆。然后，随着exec，新进程脱胎换骨，离家独立，开始了独立工作的职业生涯。
- 人有生老病死，进程也一样，它可以是自然死亡，即运行到主（main）函数的最后一个"}"，从容地离我们而去；也可以是中途退场，退场有2种方式，一种是调用exit函数，一种是在主（main）函数内使用return，无论哪一种方式，它都可以留下留言，放在返回值里保留下来；甚至它还可能被谋杀，被其它进程通过另外一些方式结束它的生命。
- 进程死掉以后，会留下一个空壳，wait站好最后一班岗，打扫战场，使其最终归于无形。这就是进程完整的一生。

参考资料



深入理解Linux内核 第三版第三章

Linux内核设计与实现第三版第三章

谢谢大家！

THANK YOU