kthreadd是kernel thread之母！

walterzh@walterzh-Precision-T1650:~$ ps aux | egrep "\[kthreadd\]"

root 2 0.0 0.0 0 0 ? S 08:27 0:00 [kthreadd]

in init/main.c

```
1.  static noinline void __init_refok rest_init(void)
2.  {
3.          int pid;
4.
5.          rcu_scheduler_starting();
6.          /*
7.           * We need to spawn init first so that it obtains pid 1, however
8.           * the init task will end up wanting to create kthreads, which, if
9.           * we schedule it before we create kthreadd, will OOPS.
10.          */
11.         kernel_thread(kernel_init, NULL, CLONE_FS);
12.         numa_default_policy();
13.         pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);        ①
14.         rcu_read_lock();
15.         kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
16.         rcu_read_unlock();
17.         complete(&kthreadd_done);
18.
19.         /*
20.          * The boot idle thread must execute schedule()
21.          * at least once to get things moving:
22.          */
23.         init_idle_bootup_task(current);
24.         schedule_preempt_disabled();
25.         /* Call into cpu_idle with preempt disabled */
26.         cpu_startup_entry(CPUHP_ONLINE);
27. }
```

①

create kthreadd thread，而该kernel thread的task就是create kernel thread!有点搞！

create kernel thread的标准function是kthread_create()

in include/linux/kthread.h

```
#define kthread_create(threadfn, data, namefmt, arg...) \
        kthread_create_on_node(threadfn, data, -1, namefmt, ##arg)
```

kthread_create_on_node()也并不是真正create thread，它只不过把传入的参数打包成一个kthread_create_info structure，然后把该structure添加到kthread_create_list这个global list上去。而真正的create thread是由"kthreadd" thread来完成的。"kthreadd" thread的任务就是从kthread_create_list上取下一个节点，也就是一个kthread_create_info structure,然后依据该structure中的参数来真正create thread。

"kthreadd" thread是一个创建kernel thread的daemon thread。它自己的创建是在系统初始化的后期完成，即①。

"kthreadd" thread运行的代码如下：

in kernel/kthread.c

```
1.  int kthreadd(void *unused)
2.  {
3.          struct task_struct *tsk = current;
4.
5.          /* Setup a clean context for our children to inherit. */
6.          set_task_comm(tsk, "kthreadd");
                                          ②
7.          ignore_signals(tsk);
8.          set_cpus_allowed_ptr(tsk, cpu_all_mask);
9.          set_mems_allowed(node_states[N_MEMORY]);
10.
11.         current->flags |= PF_NOFREEZE;
12.
13.         for (;;) {
14.                 set_current_state(TASK_INTERRUPTIBLE);
15.                 if (list_empty(&kthread_create_list))
                            ③
16.                         schedule();
17.                 __set_current_state(TASK_RUNNING);
18.
19.                 spin_lock(&kthread_create_lock);
20.                 while (!list_empty(&kthread_create_list)) {
                     ④
21.                         struct kthread_create_info *create;
22.
23.                         create = list_entry(kthread_create_list.next,
        ⑤
24.                                             struct kthread_create_info, list);
25.                         list_del_init(&create->list);
26.                         spin_unlock(&kthread_create_lock);
27.
28.                         create_kthread(create);
                                                        ⑥
29.
30.                         spin_lock(&kthread_create_lock);
31.                 }
32.                 spin_unlock(&kthread_create_lock);
33.         }
34.
35.         return 0;
36. }
```

②

这是创建kernel thread的thread的名字。

③

kthread_create_list要是空，没有kernel thread要创建，自然让出CPU

④

如果kthread_create_list上有节点，即有其他code调用了kthread_create()要求创建kernel thread。

⑤

从kthread_create_list上取下一个node

⑥

真正的去创建kernel thread。

```
1.  static void create_kthread(struct kthread_create_info *create)
2.  {
3.          int pid;
4.
5.  #ifdef CONFIG_NUMA
6.          current->pref_node_fork = create->node;
7.  #endif
8.          /* We want our own signal handler (we take no signals by default). */
9.          pid = kernel_thread(kthread, create, CLONE_FS | CLONE_FILES | SIGCHLD);
10.         if (pid < 0) {
11.                 /* If user was SIGKILLed, I release the structure. */
12.                 struct completion *done = xchg(&create->done, NULL);
13.
14.                 if (!done) {
15.                         kfree(create);
16.                         return;
17.                 }
18.                 create->result = ERR_PTR(pid);
                                ⑦
19.                 complete(done);
                                                                    ⑧
20.         }
21. }
```

in kernel/fork.c

```
/*
 * Create a kernel thread.
 */
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
        return do_fork(flags|CLONE_VM|CLONE_UNTRACED, (unsigned long)fn,
                (unsigned long)arg, NULL, NULL);
}
```

Linux并没有真正的"thread"，所谓thread也是task(task_struc)。fork用于创建task(process)，只不过VM是共享的(CLONE_VM)。kernel thread自然没有user space。

这里的⑦，当真正创建kernel thread失败，必须把出错码告诉申请创建方。

申请创建方与真正创建kernel thread的"kthreadd"通过create->done则个struct completion来同步。因为申请服务者与提供服务者运行在不同thread中，自然要同步。

⑧

当申请创建方提出申请后会wait在对应的struct completion上，直到这里的complete(done)来释放它。让它从kthread_create_on_node()返回。

kernel thread创建的workhorse如下：

```
/**
 * kthread_create_on_node - create a kthread.
 * @threadfn: the function to run until signal_pending(current).
 * @data: data ptr for @threadfn.
 * @node: memory node number.
 * @namefmt: printf-style name for the thread.
 *
 * Description: This helper function creates and names a kernel
 * thread.  The thread will be stopped: use wake_up_process() to start
 * it.  See also kthread_run().
 *
 * If thread is going to be bound on a particular cpu, give its node
 * in @node, to get NUMA affinity for kthread stack, or else give -1.
 * When woken, the thread will run @threadfn() with @data as its
 * argument. @threadfn() can either call do_exit() directly if it is a
 * standalone thread for which no one will call kthread_stop(), or
 * return when 'kthread_should_stop()' is true (which means
 * kthread_stop() has been called).  The return value should be zero
 * or a negative error number; it will be passed to kthread_stop().
 *
 * Returns a task_struct or ERR_PTR(-ENOMEM) or ERR_PTR(-EINTR).
 */
struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
                                           void *data, int node,
                                           const char namefmt[],
                                           ...)
{
        DECLARE_COMPLETION_ONSTACK(done);
                                                  ①
        struct task_struct *task;
        struct kthread_create_info *create = kmalloc(sizeof(*create),
                                                     GFP_KERNEL);

        if (!create)
                return ERR_PTR(-ENOMEM);
        create->threadfn = threadfn;
                                                                  ②
        create->data = data;
        create->node = node;
        create->done = &done;

        spin_lock(&kthread_create_lock);
        list_add_tail(&create->list, &kthread_create_list);
                   ③
        spin_unlock(&kthread_create_lock);

        wake_up_process(kthreadd_task);
                                                          ④
        /*
         * Wait for completion in killable state, for I might be chosen by
         * the OOM killer while kthreadd is trying to allocate memory for
         * new kernel thread.
         */
```

```
50.         if (unlikely(wait_for_completion_killable(&done))) {
            ⑤
51.                 /*
52.                  * If I was SIGKILLed before kthreadd (or new kernel thread)
53.                  * calls complete(), leave the cleanup of this structure to
54.                  * that thread.
55.                  */
56.                 if (xchg(&create->done, NULL))
57.                         return ERR_PTR(-EINTR);
58.                 /*
59.                  * kthreadd (or new kernel thread) will call complete()
60.                  * shortly.
61.                  */
62.                 wait_for_completion(&done);

63.         }
64.         task = create->result;
                                                                    ⑥
65.         if (!IS_ERR(task)) {
66.                 static const struct sched_param param = { .sched_priority = 0 };
67.                 va_list args;
68.
69.                 va_start(args, namefmt);
70.                 vsnprintf(task->comm, sizeof(task->comm), namefmt, args);
71.                 va_end(args);
72.                 /*
73.                  * root may have changed our (kthreadd's) priority or CPU mask.
74.                  * The kernel thread should not inherit these properties.
75.                  */
76.                 sched_setscheduler_nocheck(task, SCHED_NORMAL, ¶m);
77.                 set_cpus_allowed_ptr(task, cpu_all_mask);
                                    ⑦
78.         }
79.         kfree(create);
80.         return task;
81. }
```

①

在stack上定义一个struct completion，kernel thread的申请创建方就会wait在它上面。

②

把创建kernel thread所需的参数打包进kthread_create_info structure。

③

把kthread_create_info structure放入kthread_create_list。

④

kthread_create_list有节点了，自然要通知一下"kthreadd"(如果原来kthread_create_list为空，它可能睡着呢)

⑤

wait_for_completion_killable()返回0表示completion成功返回，也就是在create_kthread()中调用了complete(done)。thread创建成功与否要看create->result，但最起码创建过程没有被打断(比如申请方自己被kill了)。而如果wait_for_completion_killable()返回-ERESTARTSYS，则表示被kill。

⑥

取得"kthreadd"创建kernel thread的结果，成功还是失败。

⑦

成功的情况下，要设置新创建的kernel thread对CPU的"亲和力"。这里设置cpu_all_mask，表示该thread可以运行在所有CPU core上。