Proper way of handling threads in kernel?

Ask Question

I've seen bits of scattered information all around, but I can't seem to get to one final answer. How do you clean up a zombie thread in kernel?

Just to make sure, and produce a final correct way of handling threads in kernel, I would like to ask this question more broadly. **How do you create, terminate and clean up a thread in the Linux kernel?**

What I have so far is this:

```
thread_func:
    exited = 0;
    while (!must_exit)
        do stuff
    exited = 1;
    do_exit(0)

init_module:
    must_exit = 0;
    exited = 1;
    kthread_run(thread_func, ...)    /* creates and runs the thread */

cleanup_module:
    must_exit = 1;
    while (!exited)
        set_current_state(TASK_INTERRUPTIBLE);
        msleep(1);
    /* How do I cleanup? */
```

The closest thing I have found to the cleanup solution is release_task, but I didn't find anywhere talking about it. I imagined since the thread functions are `kthread_create`, `kthread_run` etc, there should be a `kthread_join` or `kthread_wait`, but there wasn't. `do_wait` also seemed likely, but it doesn't take a `struct task_struct *`.

Furthermore, I am not sure if `do_exit` is a good idea, or if at all necessary either. Can someone please come up with the minimum sketch of how a kthread should be created, terminated and cleaned up?

asked Apr 16 '12 at 16:03

Shahbaz

33.1k1080137

1 Answer 1

up vote 11 down vote accepted
One of the "right" ways to do this is to have your thread function check if it `kthread_should_stop`, and simply return if it does need to stop.

You don't need to call `do_exit`, and if you intend to `kthread_stop` it from the module exit function, you probably shouldn't.

You can see this by looking at the documentation for `kthread_create_on_node` in `kernel/kthread.c` (extract from Linux kernel 3.3.1):

> /**
> * 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).
> */

A "matching" comment is present for `kthread_stop`:

> If threadfn() may call do_exit() itself, **the caller must ensure task_struct can't go away**.

(And I'm not sure how you do that - probably holding on to the `struct_task` with a `get_task_struct`.)

If you walk the path of a thread creation you'll get something like:

```
kthread_create                                          // macro in kthread.h
  -> kthread_create_on_node                             // in kthead.c
    -> adds your thread request to kthread_create_list
    -> wakes up the kthreadd_task
```

`kthreadd_task` is set up in `init/main.c` in `reset_init`. It runs the `kthreadd` function (from `kthread.c`)

```
kthreadd                                                    // all in kthread.c
  -> create_kthread
    -> kernel_thread(kthread, your_kthread_create_info, ...)
```

And the `kthread` function itself does:

```
kthread
  -> initialization stuff
  -> schedule() // allows you to cancel the thread before it's actually started
  -> if (!should_stop)
    -> ret = your_thread_function()
  -> do_exit(ret)
```

... So if `your_thread_function` simply returns, `do_exit` will be called with its return value. No need to do it yourself.

Your Answer

Sign up or log in

Sign up using Google
Sign up using Facebook
Sign up using Email and Password

Post as a guest

Not the answer you're looking for? Browse other questions tagged c multithreading linux-kernel zombie-process or ask your own question.

lang-c