似水流年

暂无签名

renyi_UNIX

博客访问： 7936
博文数量： 9
博客积分： 0
博客等级： 民兵
技术积分： 145
用 户 组： 普通用户
注册时间： 2013-09-26 14:47

加关注 短消息

Linux内核线程

2014-03-07 21:54:01

分类： LINUX

1.内核线程介绍：

内核经常需要在后台执行一些操作，这种任务就可以通过内核线程（kernle thread）完成——独立运行在内核空间的标准进程。内核线程和普通的进程间的区别在于内核线程没有独立的地址空间，mm指针被设置为NULL；它只在内核空间运行，从来不切换到用户空间去；并且和普通进程一样，可以被调度，也可以被抢占。

内核线程只能由其它的内核线程创建，Linux内核通过给出的函数接口与系统中的初始内核线程kthrreadd交互，由kthreadd衍生出其它的内核线程。

2.相关接口函数：

1.kthread_create：函数返回一个task_struct指针，指向代表新建内核线程的task_struct结构体。注意：使用该函数创建的内核线程处于不可运行状态，需要将kthread_create返回的task_struct传递给wake_up_process函数，通过此函数唤醒新建的内核线程。

2.kthread_run：该函数新建并运行新创建的线程。该函数定义如下：

```
#define kthread_run(threadfn, data, namefmt, ...)              \
({                                                             \
    struct task_struct *__k                                    \
        = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__); \
    if (!IS_ERR(__k))                                          \
```

```
        wake_up_process(__k);                    \
    __k;                                         \
})
```

3.kthread_stop：线程一旦启动起来后，会一直运行，除非该线程主动调用do_exit函数，或者其他的进程调用kthread_stop函数，结束线程的运行。

4.kthread_should_stop：该函数位于内核线程函数体内，用于接收kthread_stop传递的结束线程信号，如果内核线程中未用此函数，则kthread_stop使其结束

3.实现举例

3.1头文件：

#include <linux/sched.h> //wake_up_process()
#include <linux/kthread.h> //kthread_create()、kthread_run()
#include <err.h> //IS_ERR()、PTR_ERR()

3.2实现

1.创建线程

在内核模块初始化时，可以进行线程的创建。使用kthread_create+wake_up_process或者kthread_run函数。如：

```
static struct task_struct *test_task;
static int test_init_module(void)
{
    int err;
    test_task = kthread_create(test_thread, NULL, "test_task");
    if(IS_ERR(test_task)){
      printk("Unable to start kernel thread.\n");
      err = PTR_ERR(test_task);
      test_task = NULL;
      return err;
    }
    wake_up_process(test_task);
    return 0;
}
module_init(test_init_module);
```

2.线程函数

在线程函数里，完成所需的业务逻辑工作。主要框架如下所示：

```
int threadfunc(void *data){
        …
        while(1){
                set_current_state(TASK_UNINTERRUPTIBLE);
                if(kthread_should_stop()) break;
                if(){//条件为真
```

```
                        //进行业务处理
                }
                else{//条件为假
                        //让出CPU运行其他线程，并在指定的时间内重新被调度
                        schedule_timeout(HZ);
                }
        }
        …
        return 0;
}
```

3.结束线程

在模块卸载时，可以使用kthread_stop结束线程的运行。

函数原型：int kthread_stop(struct task_struct *k);

```
static void test_cleanup_module(void)
{
            if(test_task){
                kthread_stop(test_task);
                test_task = NULL;
            }
}
module_exit(test_cleanup_module);
```

3.3注意事项

（1）在调用kthread_stop函数时，线程函数不能已经运行结束。否则，kthread_stop函数会一直进行等待。

（2）线程函数必须能让出CPU，以便能运行其他线程。同时线程函数也必须能重新被调度运行。在例子程序中，这是通过schedule_timeout()函数完成的，也可自己调用schedule函数等方式。

4.代码示例

以下代码，摘抄自网络：

```
 #include <linux/kthread.h>
#include <linux/module.h>
#ifndef SLEEP_MILLI_SEC
#define SLEEP_MILLI_SEC(nMilliSec)\
do { \
long timeout = (nMilliSec) * HZ / 1000; \
while(timeout > 0) \
{ \
timeout = schedule_timeout(timeout); \
} \
```

```
}while(0);
#endif
static struct task_struct * MyThread = NULL;
static int MyPrintk(void *data)
{
char *mydata = kmalloc(strlen(data)+1,GFP_KERNEL);
memset(mydata,'\0',strlen(data)+1);
strncpy(mydata,data,strlen(data));
while(!kthread_should_stop())
{
SLEEP_MILLI_SEC(1000);
printk("%s\n",mydata);
}
kfree(mydata);
return 0;
}
static int __init init_kthread(void)
{
MyThread = kthread_run(MyPrintk,"hello world","mythread");
return 0;
}
static void __exit exit_kthread(void)
{
if(MyThread)
{
printk("stop MyThread\n");
kthread_stop(MyThread);
}
}
module_init(init_kthread);
module_exit(exit_kthread);
```

5.内核线程创建原理详解

kernel_thread为真实的用于创建内核线程的函数，其最终会调用do_fork函数进行内核线程的创建，内核提供的内核线程创建函数如kthread_create等，最终都会间接调用到kernel_thread函数进行内核线程的创建。

```
/*函数定义于: (linux)/include/linux/sched.h
 *函数实现于: (linux)/kernel/fork.c
```

```
*/
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);
}
```

内核线程之父kthreadd的创建

在内核启动的后续阶段，start_kernel最后会调用rest_init函数进行init函数以及kthreadd内核线程的初始化，函数源码如下所示。kernel_init内核线程负责创建init进程，创建成功后，该内核线程运行结束；接下来创建kthreadd内核线程，该内核线程将常驻，并根据需求与相关函数调用衍生出新的内核线程，kthreadd将作为其它内核线程的模板。

```
//(linux)//init/main.c
static noinline void __init_refok rest_init(void)
{
    ……
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ……
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns); //全局变量kthreadd_task为指向kthreadd的task结构体
    rcu_read_unlock();
    ……
}
```

kthreadd工作原理如下：

kthreadd内核线程的主体是内核中的kthreadd这个函数，其源码如下所示。其中for循环负责根据kthread_create_list中的信息，创建新的内核线程。

```
int kthreadd(void *unused)
{
    struct task_struct *tsk = current;

    /* Setup a clean context for our children to inherit. */
    set_task_comm(tsk, "kthreadd");
    ignore_signals(tsk);
    set_cpus_allowed_ptr(tsk, cpu_all_mask);
    set_mems_allowed(node_states[N_MEMORY]);

    current->flags |= PF_NOFREEZE;
```

```
    for (;;) {
        set_current_state(TASK_INTERRUPTIBLE);
        if (list_empty(&kthread_create_list))
            schedule();   //如果链表上没有新的内核线程创建请求，则调度其它进程
        //当调用kthread_create函数时，其会通过wak_up_process唤醒此线程，并从此处开始运行
        __set_current_state(TASK_RUNNING);

        spin_lock(&kthread_create_lock);
        while (!list_empty(&kthread_create_list)) {
            struct kthread_create_info *create;

            create = list_entry(kthread_create_list.next,
                        struct kthread_create_info, list);
            list_del_init(&create->list);
            spin_unlock(&kthread_create_lock);

            create_kthread(create);   //函数中，会调用kernel_threadh函数，创建由create所指定信息的内核线程

            spin_lock(&kthread_create_lock);
        }
        spin_unlock(&kthread_create_lock);
    }

    return 0;
}
```

kthread_create是一个宏，对应的函数为kthread_create_on_node:

```
//(linux)/include/linux/kthread.h
#define kthread_create(threadfn, data, namefmt, arg...) \
    kthread_create_on_node(threadfn, data, -1, namefmt, ##arg)
```

kthread_create_on_node定义如下，其原理为:
1）向***kthread_create_list***添加新的内核线程创建需求;
2）通过**wake_up_process**唤醒kthreadd内核线程，接下来kthreadd内核线程会取***kthread_create_list***新的请求，然后调用create_kthread进行内核线程创建，create_thread最终会调用kernel_create函数

```
struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
                        void *data, int node,
                        const char namefmt[],
                        ...)
```

```
{
    ……
    spin_lock(&kthread_create_lock);
    list_add_tail(&create.list, &kthread_create_list); 将内核线程创建的请求添加到kthread_create_list上
    spin_unlock(&kthread_create_lock);

    wake_up_process(kthreadd_task); //唤醒kthreadd，使其创建新的内核线程
    wait_for_completion(&create.done);
    ……
}
EXPORT_SYMBOL(kthread_create_on_node);
```


上一篇：Linux日志系统——rsyslog
下一篇：桥接XEN虚拟机到物理网络

给主人留下些什么吧！~~

评论热议

请登录后评论。

登录 注册