第 0006 讲 5 进程调度 API 系统调用案例分析

一、进程调度常用 API 函数：kthread_create_on_node(...)函数功能：指定存储节点创建新内核线程。具体 Linux 内核源码函数设计如下：

```
kernel > C kthread.c > kthread_create_on_node(int(* )(void *data), void *, int, const char [], ...)
380
381 struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
382                                             void *data, int node,
383                                             const char namefmt[],
384                                             ...)
385 {
386     struct task_struct *task;
387     va_list args;
388
389     va_start(args, namefmt);
390     task = __kthread_create_on_node(threadfn, data, node, namefmt, args);
391     va_end(args);
392
393     return task;
394 }
395 EXPORT_SYMBOL(kthread_create_on_node);
396
```

【运行结果】

```
root@ubuntu: /home/vico/Desktop/knode
File  Edit  View  Search  Terminal  Help
root@ubuntu:/home/vico/Desktop/knode# ls -l
total 36
-rw------- 1 vico vico 1154 7月   28 16:40 ktconode.c
-rw-r--r-- 1 root root 5968 7月   28 16:41 ktconode.ko
-rw-r--r-- 1 root root   37 7月   28 16:41 ktconode.mod
-rw-r--r-- 1 root root  852 7月   28 16:41 ktconode.mod.c
-rw-r--r-- 1 root root 3320 7月   28 16:41 ktconode.mod.o
-rw-r--r-- 1 root root 3512 7月   28 16:41 ktconode.o
-rw------- 1 vico vico  283 7月   28 16:36 Makefile
-rw-r--r-- 1 root root   37 7月   28 16:41 modules.order
-rw-r--r-- 1 root root    0 7月   28 16:37 Module.symvers
root@ubuntu:/home/vico/Desktop/knode# insmod ktconode.ko
root@ubuntu:/home/vico/Desktop/knode# dmesg -c
[ 1762.038727] 正常退出内核:kthread_create_on_node(...)函数.
[ 1844.839363] 调用KthreadCreateOnNode_Func(...)函数.
[ 1844.840858] 打印新线程的PID的值为：5062
[ 1844.840860] 打印当前进程的PID的值为：5061
[ 1844.840860] 退出KthreadCreateOnNode_Func(...)函数.
[ 1844.846682] 调用线程函数MyFunc_ThreadFunc(...)
[ 1844.846685] 打印输出当前进程PID的值为：5062
[ 1844.846685] 退出线程函数MyFunc_ThreadFunc(...)
root@ubuntu:/home/vico/Desktop/knode# rmmod ktconode.ko
root@ubuntu:/home/vico/Desktop/knode# dmesg -c
[ 1898.421839] 正常退出内核:kthread_create_on_node(...)函数.
root@ubuntu:/home/vico/Desktop/knode#
```

【程序源码】

```
#include <linux/module.h>

#include <linux/pid.h>

#include <linux/sched.h>

#include <linux/wait.h>

#include <linux/kthread.h>
```

```
int MyFunc_ThreadFunc(void *argc)
{
	printk("调用线程函数 MyFunc_ThreadFunc(...)\n");

	printk("打印输出当前进程 PID 的值为：%d\n",current->pid);

	printk("退出线程函数 MyFunc_ThreadFunc(...)\n");

	return 0;
}

static int __init KthreadCreateOnNode_Func(void)
{
	struct task_struct *pts=NULL;

	printk("调用 KthreadCreateOnNode_Func(...)函数.\n");

	pts=kthread_create_on_node(MyFunc_ThreadFunc,NULL,-1,"ktconode.c"); //ktconode

	printk("打印新线程的 PID 的值为：%d\n",pts->pid);

	wake_up_process(pts); //  唤醒刚才新创建的内核线程

	printk("打印当前进程的 PID 的值为：%d\n",current->pid);

	printk("退出 KthreadCreateOnNode_Func(...)函数.\n");

	return 0;
}
```

```
static void __exit KthreadCreateOnNode_Exit(void)
{
    printk("正常退出内核:kthread_create_on_node(...)函数.\n");
}

MODULE_LICENSE("GPL");

module_init(KthreadCreateOnNode_Func);
module_exit(KthreadCreateOnNode_Exit);
```

二、进程调度常用 API 函数：wake_up_process(...)函数功能：使唤醒处于睡眠状态的进程状态转为 RUNNING 状态，让 CPU 重新调度处理。具体 Linux 内核源码函数设计如下：

```
kernel > sched > C core.c > wake_up_state(task_struct *, unsigned int)
2657
2658  int wake_up_process(struct task_struct *p) // 此参数指针：进程的描述符信息，保存进程信息
2659  {
2660      return try_to_wake_up(p, TASK_NORMAL, 0);
2661  }
2662  EXPORT_SYMBOL(wake_up_process);
2663  // 返回：0（相当于当前进程处于RUNNING状态或唤醒进程已经失败）
2664  // 返回：1（相当于当前进程不是处于RUNNING状态，唤醒进程已经成功）
2665
```

【运行结果】

```
root@ubuntu: /home/vico/Desktop/wakeuptest
File  Edit  View  Search  Terminal  Help
-rw-r--r-- 1 root root    0 7月   28 17:16 Module.symvers
-rw------- 1 vico vico 2687 7月   28 17:16 wakeup.c
-rw-r--r-- 1 root root 7784 7月   28 17:16 wakeup.ko
-rw-r--r-- 1 root root   40 7月   28 17:16 wakeup.mod
-rw-r--r-- 1 root root 1061 7月   28 17:16 wakeup.mod.c
-rw-r--r-- 1 root root 3640 7月   28 17:16 wakeup.mod.o
-rw-r--r-- 1 root root 4896 7月   28 17:16 wakeup.o
root@ubuntu:/home/vico/Desktop/wakeuptest# insmod wakeup.ko
root@ubuntu:/home/vico/Desktop/wakeuptest# dmesg -c
[ 3976.746918] wakeup: module license 'unspecified' taints kernel.
[ 3976.746919] Disabling lock debugging due to kernel taint
[ 3976.747104] 调用wakeupprocess_functest_init(...)函数.
[ 3976.747446] 打印新的内核线程的PID值为：7505
[ 3976.747446] 打印当前进程的PID值为：7504
[ 3976.747447] 调用wake_up_process(...)函数，唤醒新线程之后的结果为：1
[ 3976.747449] 调用内核线程函数：MyFunc_ThreadFunc(...).
[ 3976.747449] 打印当前进程的pid的值为：7505
[ 3976.747450] 初始化函数状态为:2
[ 3976.747451] 调用wake_up_process之后的函数状态为：0
[ 3976.747451] 调用wake_up_process函数返回结果为:1
[ 3976.747451] 退出内核线程函数：MyFunc_ThreadFunc(...).
[ 3976.747453] 唤醒当前进程为RUNNING状态之后线程结果为：0
[ 3976.747454] 调用sched_timeout_uninterruptible(...)函数，返回的值为：20000
[ 3976.747454] 退出wakeupprocess_functest_init(...)函数.
root@ubuntu:/home/vico/Desktop/wakeuptest# rmmod wakeup.ko
root@ubuntu:/home/vico/Desktop/wakeuptest# dmesg -c
[ 4017.936340] 正常退出内核：wake_up_process(...)函数.
root@ubuntu:/home/vico/Desktop/wakeuptest#
```

【程序源码】

```
#include <linux/kthread.h>
#include <linux/wait.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/pid.h>
#include <linux/delay.h>
#include <linux/list.h>

// 声明全局变量专用于保存：进程描述符的信息
struct task_struct *pts_thread=NULL;

int MyFunc_ThreadFunc(void *argc)
{
    int iData=-1;

    printk("调用内核线程函数：MyFunc_ThreadFunc(...).\n");
    printk("打印当前进程的 pid 的值为：%d\n",current->pid);

    // 显示父进程的状态
    printk("初始化函数状态为:%ld\n",pts_thread->state);
    iData=wake_up_process(pts_thread);

    // 打印输出函数调用之后的父进程状态
    printk("调用 wake_up_process 之后的函数状态为：%ld\n",pts_thread->state);
    printk("调用 wake_up_process 函数返回结果为:%d\n",iData);

    printk("退出内核线程函数：MyFunc_ThreadFunc(...).\n");

    return 0;
```

```
}

static int __init wakeupprocess_functest_init(void)
{
    int result_data=-1; // 保存 wake_up_process 结果
    char cName[]="wakeup.c%s";
    struct task_struct *pResult=NULL;
    long time_out;

    // 等待队列元素
    wait_queue_head_t head;
    wait_queue_entry_t data;

    printk("调用 wakeupprocess_functest_init(...)函数.\n");
    // 指定存储节点，创建一个新的内核线程
    pResult=kthread_create_on_node(MyFunc_ThreadFunc,NULL,-1,cName);
    printk("打印新的内核线程的 PID 值为：%d\n",pResult->pid);
    printk("打印当前进程的 PID 值为：%d\n",current->pid);

    init_waitqueue_head(&head); // 初始化等待队列的头元素
    init_waitqueue_entry(&data,current);
    add_wait_queue(&head,&data);
    pts_thread=current; // 保存当前进程的数据信息

    result_data=wake_up_process(pResult); // 唤醒刚创建新线程
    printk("调用 wake_up_process(...)函数，唤醒新线程之后的结果为：%d\n",result_data);

    // 让当前进程进入睡眠状态
    time_out=schedule_timeout_uninterruptible(2000*10);
```

```
    // 唤醒当前进程，让当前进程处于 RUNNING 状态
    result_data=wake_up_process(current);
    printk("唤醒当前进程为 RUNNING 状态之后线程结果为：%d\n",result_data);

    // 输出 time_out 的值
    printk("调用 sched_timeout_uninterruptible(...)函数，返回的值为：%ld\n",time_out);

    printk("退出 wakeupprocess_functest_init(...)函数.\n");

    return 0;
}

static void __exit wakeupprocess_functest_exit(void)
{
    printk("正常退出内核：wake_up_process(...)函数.\n");
}

module_init(wakeupprocess_functest_init);
module_exit(wakeupprocess_functest_exit);
```