openEuler内核编程技术

实训指导书

第四章 第1讲

《创建并运行内核线程》

软件所制

**术语表**

|  |  |  |  |
| --- | --- | --- | --- |
| 英文 | 中文 | 含义 | 别名 |
|  |  |  |  |
|  |  |  |  |

第四章 第1讲 创建并运行内核线程

1. **任务1：创建并运行内核线程（60min）**

## 相关知识

**一、内核模块介绍**

内核模块是Linux内核向外部提供的一个插口，其全称为动态可加载内核模块(Loadable Kernel Module)，简称模块，模块通常由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序或其他内核上层的功能，模块可以被编译到内核中，或者作为独立的模块被动态加载。

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不同的。模块通常由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序或其他内核上层的功能。

**二、 内核模块加载的方式**

1. 动态加载

编写内核模块源文件，用makefile来编译，最后生成xxx.ko文件，用insmod指令加载模块，rmmod指令来卸载模块

2. 静态加载

系统启动时自动加载，内核初始化的最后一步，会启动init进程，init进程会依次执行/etc/rc.d中的启动脚本，其中实现了内核模块的加载、服务的启动等功能

**三、 内核模块的优缺点**

* 优点

·将内核映象的尺寸保持在最小，并具有最大的灵活性；

·便于检验新的内核代码，而不需重新编译内核并重新引导。

* 缺点

·对系统性能和内存利用有负面影响；

·装入的内核模块和其他内核部分一样，具有相同的访问权限，因此，差的内核模块会导致系统崩溃；

·为了使内核模块访问所有内核资源，内核必须维护符号表，并在装入和卸载模块时修改这些符号表；

**四、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，内核线程是指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kthreadd交互，由kthreadd衍生出其它的内核线程。

Linux操作系统启动以后，你可以用”ps -ef”命令查看系统中的进程，这时会发现很多以”d”结尾的进程名，确切说名称显示里面加 "[]"的，这些进程就是内核线程。

**二、相关接口函数**

1．printk()：内核态信息打印函数，功能和C 库的 printf 类似。

头文件：< linux/kernel.h>

int printk(const char \*fmt, ...);

消息打印级别：

#define KERN\_EMERG "<0>" /\* 紧急事件消息，系统崩溃之前提示，表示系统不可用 \*/

#define KERN\_ALERT "<1>" /\* 报告消息，表示必须立即采取措施 \*/

#define KERN\_CRIT "<2>" /\* 临界条件，通常涉及严重的硬件或软件操作失败 \*/

#define KERN\_ERR "<3>" /\* 错误条件，驱动程序常用 KERN\_ERR 来报告硬件的错误 \*/

#define KERN\_WARNING "<4>" /\* 警告条件，对可能出现问题的情况进行警告 \*/

#define KERN\_NOTICE "<5>" /\* 正常但又重要的条件，用于提醒。常用于与安全相关的消息 \*/

#define KERN\_INFO "<6>" /\* 提示信息，如驱动程序启动时，打印硬件信息 \*/

#define KERN\_DEBUG "<7>" /\* 调试级别的消息 \*/

说明：不同级别使用不同的字符串表示，数字越小，级别越高。

printk() 函数是直接使用了向终端写函数 tty\_write() 。而 printf() 函数是调用 write() 系统调用函数向标准输出设备写。所以在用户态（如进程 0 ）不能够直接使用 printk() 函数，而在内核态由于它已是特权级，所以无需系统调用来改变特权级，因而能够直接使用 printk() 函数。 printk是内核输出，在终端是看不见的。但是我们可以使用命令： cat /var/log/messages ，或者使用 dmesg 命令看一下输出的信息。

2. kthread\_create()，创建内核线程

头文件：< linux/kthread.h>

#define kthread\_create(threadfn, data, namefmt, arg...) \

        kthread\_create\_on\_node(threadfn, data, -1, namefmt, ##arg)

3. kthread\_run()，创建并运行内核线程

头文件：< linux/kthread.h>

#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; \

})

与kthread\_create()相比多了个wake\_up\_process()操作。

4.kthread\_stop()，停止指定内核线程

头文件：< linux/kthread.h>

int kthread\_stop(struct task\_struct \*k);

线程一旦启动起来之后，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。当然，如果线程函数永远不返回，并且不检查信号，它将永远不会停止。因此线程函数信号检查语句以及返回值非常重要。

注意，在调用kthread\_stop函数时，线程不能已经结束运行，否则，kthread\_stop函数会一直等待。

5.kthread\_should\_stop()，判断线程是否应该停止

头文件：< linux/kthread.h>

bool kthread\_should\_stop(void);

6.kthread\_bind()，绑定创建好的线程在执行CPU核心上运行

头文件：< linux/kthread.h>

void kthread\_bind(struct task\_struct \*k, unsigned int cpu);

也可在创建的时候调用如下函数在创建的同时一起绑定：

struct task\_struct \*kthread\_create\_on\_cpu(int (\*threadfn)(void \*data),

                   void \*data,

                   unsigned int cpu,

                   const char \*namefmt);

两者在使用后需要调用kthread\_run()里用到的wake\_up\_process()才能进入运行队列。

7、线程函数

用户在线程函数中指定要让该线程完成的任务。该函数会一直运行，直到接收到结束信号。因此函数中需要有判断是否收到信号的语句。

static int func(void \*data){

while(!kthread\_should\_stop()){

。。。。。。一些工作

msleep(2000);

}

return 0;

}

注意在线程函数中需要在每一轮迭代之后休眠一定时间，让出CPU 给其他的任务，否则创建的这个线程会一直占用CPU，使得其他任务均瘫痪。更严重的是，使线程终止的命令也无法执行，导致这种状态一直持续下去。

8. wake\_up\_process()：唤醒处于睡眠状态的进程，使进程由睡眠状态变为RUNNING状态，从而能够被CPU重新调度执行。

头文件：< linux/sched.h>

int wake\_up\_process(struct task\_struct\*tsk)

返回参数说明：此函数的返回结果是int型的变量，代表唤醒进程的情况，可能的取值是0或1。返回1代表当前进程不是处于RUNNING状态，唤醒进程成功；返回0代表当前进程处于RUNNING状态或唤醒进程失败。

9. msleep()：使当前进程被调度并让出cpu一段时间。

头文件：<linux/delay.h>

函数原型：void msleep(unsigned int msecs)

**三、task\_struct结构体**

每个进程在内核中都有一个进程控制块(PCB)来维护进程相关的信息,Linux内核的进程控制块是task\_struct结构体。task\_struct是Linux内核的一种数据结构，它会被装载到RAM中并且包含着进程的信息。每个进程都把它的信息放在 task\_struct 这个数据结构体，task\_struct 包含了这些内容：

1. **标示符**： 描述本进程的唯一标识符，用来区别其他进程。
2. **状态** ：任务状态，退出代码，退出信号等。
3. **优先级** ：相对于其他进程的优先级。
4. **程序计数器**：程序中即将被执行的下一条指令的地址。
5. **内存指针**：包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针。
6. **上下文数据**：进程执行时处理器的寄存器中的数据。
7. **I/O状态信息**：包括显示的I/O请求,分配给进程的I/O设备和被进程使用的文件列表。
8. **记账信息**：可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。

有关进程信息还有以下三点需要了解：

* 保存进程信息的数据结构叫做 task\_struct，可以在 include/linux/sched.h 中找到它；
* 所有运行在系统中的进程都以 task\_struct 链表的形式存在内核中；
* 进程的信息可以通过 /proc 系统文件夹查看。

以下是对task\_struct的定义及注释：

struct task\_struct {

volatile long state; //说明了该进程是否可以执行,还是可中断等信息

unsigned long flags; //Flage 是进程号,在调用fork()时给出

int sigpending; //进程上是否有待处理的信号

mm\_segment\_t addr\_limit; //进程地址空间,区分内核进程与普通进程在内存存放的位置不同

//0-0xBFFFFFFF for user-thead

//0-0xFFFFFFFF for kernel-thread

//调度标志,表示该进程是否需要重新调度,若非0,则当从内核态返回到用户态,会发生调度

volatile long need\_resched;

int lock\_depth; //锁深度

long nice; //进程的基本时间片

//进程的调度策略,有三种,实时进程:SCHED\_FIFO,SCHED\_RR, 分时进程:SCHED\_OTHER

unsigned long policy;

struct mm\_struct \*mm; //进程内存管理信息

int processor;

//若进程不在任何CPU上运行, cpus\_runnable 的值是0，否则是1 这个值在运行队列被锁时更新

unsigned long cpus\_runnable, cpus\_allowed;

struct list\_head run\_list; //指向运行队列的指针

unsigned long sleep\_time; //进程的睡眠时间

//用于将系统中所有的进程连成一个双向循环链表, 其根是init\_task

struct task\_struct \*next\_task, \*prev\_task;

struct mm\_struct \*active\_mm;

struct list\_head local\_pages; //指向本地页面

unsigned int allocation\_order, nr\_local\_pages;

struct linux\_binfmt \*binfmt; //进程所运行的可执行文件的格式

int exit\_code, exit\_signal;

int pdeath\_signal; //父进程终止时向子进程发送的信号

unsigned long personality;

//Linux可以运行由其他UNIX操作系统生成的符合iBCS2标准的程序

int did\_exec:1;

pid\_t pid; //进程标识符,用来代表一个进程

pid\_t pgrp; //进程组标识,表示进程所属的进程组

pid\_t tty\_old\_pgrp; //进程控制终端所在的组标识

pid\_t session; //进程的会话标识

pid\_t tgid;

int leader; //表示进程是否为会话主管

struct task\_struct \*p\_opptr,\*p\_pptr,\*p\_cptr,\*p\_ysptr,\*p\_osptr;

struct list\_head thread\_group; //线程链表

struct task\_struct \*pidhash\_next; //用于将进程链入HASH表

struct task\_struct \*\*pidhash\_pprev;

wait\_queue\_head\_t wait\_chldexit; //供wait4()使用

struct completion \*vfork\_done; //供vfork() 使用

unsigned long rt\_priority; //实时优先级，用它计算实时进程调度时的weight值

//it\_real\_value，it\_real\_incr用于REAL定时器，单位为jiffies, 系统根据it\_real\_value

//设置定时器的第一个终止时间. 在定时器到期时，向进程发送SIGALRM信号，同时根据

//it\_real\_incr重置终止时间，it\_prof\_value，it\_prof\_incr用于Profile定时器，单位为jiffies。

//当进程运行时，不管在何种状态下，每个tick都使it\_prof\_value值减一，当减到0时，向进程发送

//信号SIGPROF，并根据it\_prof\_incr重置时间.

//it\_virt\_value，it\_virt\_value用于Virtual定时器，单位为jiffies。当进程运行时，不管在何种

//状态下，每个tick都使it\_virt\_value值减一当减到0时，向进程发送信号SIGVTALRM，根据

//it\_virt\_incr重置初值。

unsigned long it\_real\_value, it\_prof\_value, it\_virt\_value;

unsigned long it\_real\_incr, it\_prof\_incr, it\_virt\_value;

struct timer\_list real\_timer; //指向实时定时器的指针

struct tms times; //记录进程消耗的时间

unsigned long start\_time; //进程创建的时间

//记录进程在每个CPU上所消耗的用户态时间和核心态时间

long per\_cpu\_utime[NR\_CPUS], per\_cpu\_stime[NR\_CPUS];

//内存缺页和交换信息:

//min\_flt, maj\_flt累计进程的次缺页数（Copy on　Write页和匿名页）和主缺页数（从映射文件或交换

//设备读入的页面数）； nswap记录进程累计换出的页面数，即写到交换设备上的页面数。

//cmin\_flt, cmaj\_flt, cnswap记录本进程为祖先的所有子孙进程的累计次缺页数，主缺页数和换出页面数。

//在父进程回收终止的子进程时，父进程会将子进程的这些信息累计到自己结构的这些域中

unsigned long min\_flt, maj\_flt, nswap, cmin\_flt, cmaj\_flt, cnswap;

int swappable:1; //表示进程的虚拟地址空间是否允许换出

//进程认证信息

//uid,gid为运行该进程的用户的用户标识符和组标识符，通常是进程创建者的uid，gid

//euid，egid为有效uid,gid

//fsuid，fsgid为文件系统uid,gid，这两个ID号通常与有效uid,gid相等，在检查对于文件

//系统的访问权限时使用他们。

//suid，sgid为备份uid,gid

uid\_t uid,euid,suid,fsuid;

gid\_t gid,egid,sgid,fsgid;

int ngroups; //记录进程在多少个用户组中

gid\_t groups[NGROUPS]; //记录进程所在的组

//进程的权能，分别是有效位集合，继承位集合，允许位集合

kernel\_cap\_t cap\_effective, cap\_inheritable, cap\_permitted;

int keep\_capabilities:1;

struct user\_struct \*user;

struct rlimit rlim[RLIM\_NLIMITS]; //与进程相关的资源限制信息

unsigned short used\_math; //是否使用FPU

char comm[16]; //进程正在运行的可执行文件名

//文件系统信息

int link\_count, total\_link\_count;

//NULL if no tty 进程所在的控制终端，如果不需要控制终端，则该指针为空

struct tty\_struct \*tty;

unsigned int locks;

//进程间通信信息

struct sem\_undo \*semundo; //进程在信号灯上的所有undo操作

struct sem\_queue \*semsleeping; //当进程因为信号灯操作而挂起时，他在该队列中记录等待的操作

//进程的CPU状态，切换时，要保存到停止进程的task\_struct中

struct thread\_struct thread;

//文件系统信息

struct fs\_struct \*fs;

//打开文件信息

struct files\_struct \*files;

//信号处理函数

spinlock\_t sigmask\_lock;

struct signal\_struct \*sig; //信号处理函数

sigset\_t blocked; //进程当前要阻塞的信号，每个信号对应一位

struct sigpending pending; //进程上是否有待处理的信号

unsigned long sas\_ss\_sp;

size\_t sas\_ss\_size;

int (\*notifier)(void \*priv);

void \*notifier\_data;

sigset\_t \*notifier\_mask;

u32 parent\_exec\_id;

u32 self\_exec\_id;

spinlock\_t alloc\_lock;

void \*journal\_info;

};

state的可能取值为：

#define TASK\_RUNNING 0//进程要么正在执行，要么准备执行

#define TASK\_INTERRUPTIBLE 1 //可中断的睡眠，可以通过一个信号唤醒

#define TASK\_UNINTERRUPTIBLE 2 //不可中断睡眠，不可以通过信号进行唤醒

#define \_\_TASK\_STOPPED 4 //进程停止执行

#define \_\_TASK\_TRACED 8 //进程被追踪

/\* in tsk->exit\_state \*/

#define EXIT\_ZOMBIE 16 //僵尸状态的进程，表示进程被终止，但是父进程还没有获取它的终止信息，比如进程有没有执行完等信息。

#define EXIT\_DEAD 32 //进程的最终状态，进程死亡

/\* in tsk->state again \*/

#define TASK\_DEAD 64 //死亡

#define TASK\_WAKEKILL 128 //唤醒并杀死的进程

#define TASK\_WAKING 256 //唤醒进程

**四、内核模块编程**

源码编写—— .c源文件

Makefile文件编写

编译模块——make

模块加载进内核——insmod

查看加载的内容——dmesg

查看内核模块——lsmod

卸载内核模块——rmmod

## 任务描述

1. 编写内核模块，创建一个内核线程；并在模块退出时杀死该线程。
2. 加载、卸载模块并查看模块打印信息。

## 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

## 参考答案

一、参考答案源码

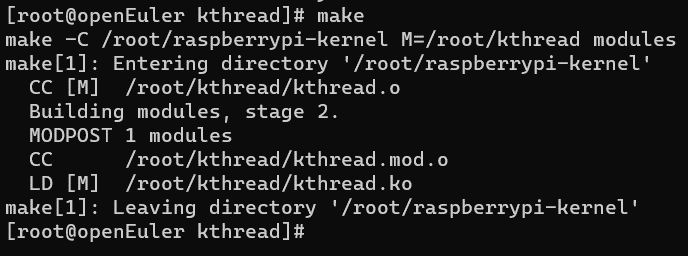
1、kthread.c

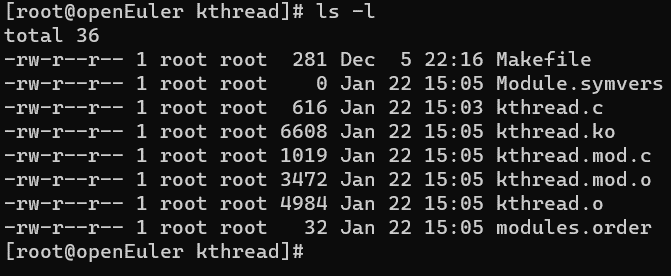
|  |
| --- |
| #include <linux/kthread.h>  #include <linux/module.h>  #include <linux/delay.h>  MODULE\_LICENSE("GPL");  #define BUF\_SIZE 20  static struct task\_struct \*myThread = NULL;  static int print(void \*data)  {  while(!kthread\_should\_stop()){  printk("New kthread is running.");  msleep(2000);  }  return 0;  }  static int \_\_init kthread\_init(void)  {  printk("Create kernel thread!\n");  myThread = kthread\_run(print, NULL, "new\_kthread");  return 0;  }  static void \_\_exit kthread\_exit(void)  {  printk("Kill new kthread.\n");  if(myThread)  kthread\_stop(myThread);  }  module\_init(kthread\_init);  module\_exit(kthread\_exit); |

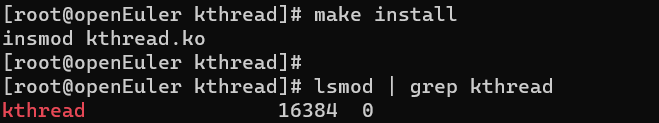
2、Makefile

|  |
| --- |
| ifneq ($(KERNELRELEASE),)  obj-m := kthread.o  else  KERNELDIR ?=/root/raspberrypi-kernel  PWD := $(shell pwd)  default:  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  endif  .PHONY:clean  install:  insmod kthread.ko  uninstall:  rmmod kthread  clean:  -rm \*.mod.c \*.o \*.order \*.symvers \*.ko |

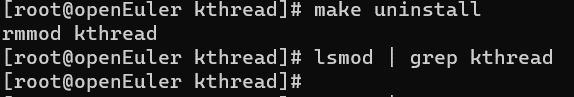
二、内核模块加载结果







****





# 任务2：绑定内核线程到指定的cpu（60min）

## 相关知识

同上

## 任务描述

1. 编写内核模块，创建一个内核线程后，满足以下功能需求：  
（1）将该内核线程绑定到指定的cpu中；（2）在模块退出时销毁该线程。

2. 加载、卸载模块并查看模块打印信息。

## 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

## 参考答案

一、参考答案源码

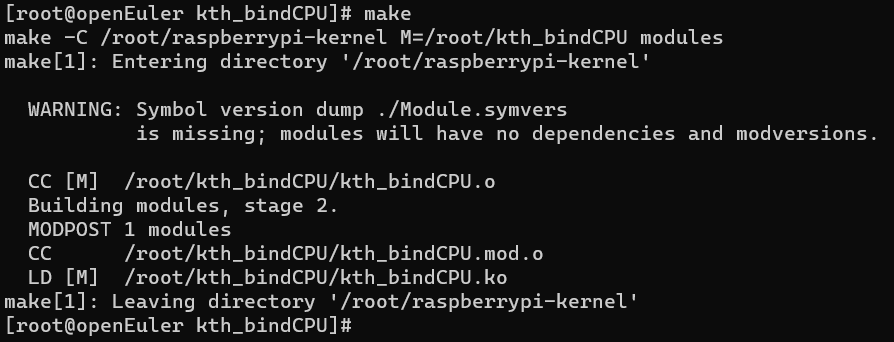
1、kth\_bindCPU.c

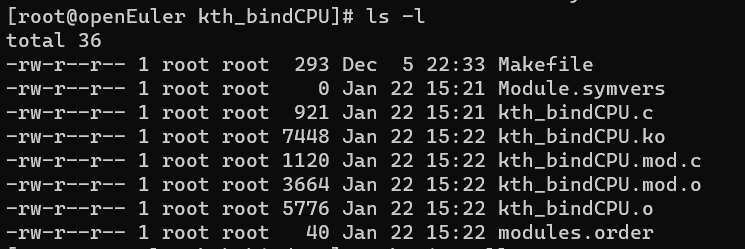
|  |
| --- |
| #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/init.h>  #include <linux/kthread.h>  #include <linux/sched.h>  #include <linux/delay.h>  static struct task\_struct \*kthread\_test = NULL;  static int MyPrintk(void \*data)  {  while (!kthread\_should\_stop())  {  int cpu = get\_cpu();  put\_cpu();  printk("hello kernel thread %s, cpu = %d\n", kthread\_test->comm, cpu);  msleep(2000);  }  return 0;  }  static int \_\_init init\_kthread(void)  {  printk("Kernel thread start");  kthread\_test = kthread\_create(MyPrintk, NULL, "kthreadtest");  kthread\_bind(kthread\_test, 1);  wake\_up\_process(kthread\_test);  return 0;  }  static void \_\_exit exit\_kthread(void)  {  printk("Kill new kthread.\n");  if(kthread\_test)  {  kthread\_stop(kthread\_test);  }  }  module\_init(init\_kthread);  module\_exit(exit\_kthread);  MODULE\_LICENSE("GPL"); |

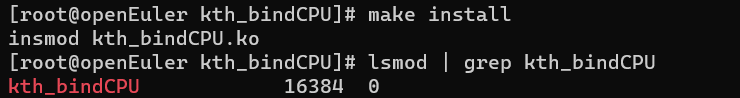
2、Makefile

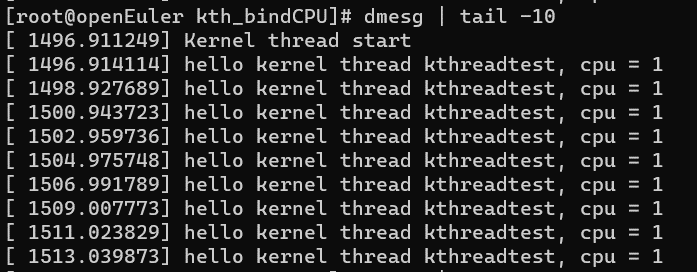
|  |
| --- |
| ifneq ($(KERNELRELEASE),)  obj-m := kth\_bindCPU.o  else  KERNELDIR ?=/root/raspberrypi-kernel  PWD := $(shell pwd)  default:  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  endif  .PHONY:clean  install:  insmod kth\_bindCPU.ko  uninstall:  rmmod kth\_bindCPU  clean:  -rm \*.mod.c \*.o \*.order \*.symvers \*.ko |

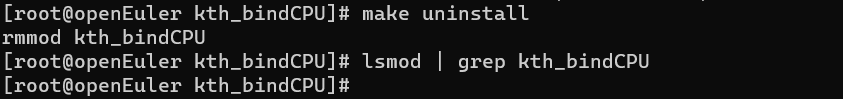
二、内核模块加载结果

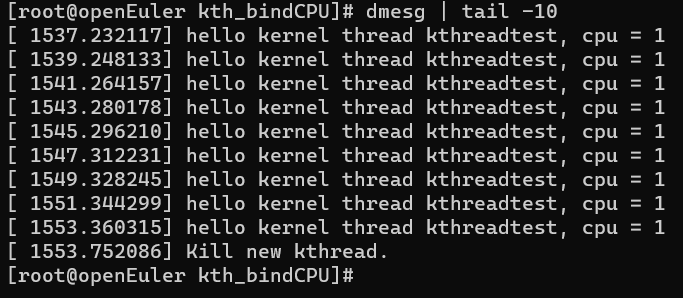












# 任务3：内核线程的睡眠与唤醒（60min）

## 相关知识

同上

## 任务描述

1. 编写内核模块，创建一个内核线程后，满足以下功能需求：

（1）设置在一定时间内使该内核线程进入睡眠；

（2）唤醒睡眠的线程；

（3）在模块退出时销毁该线程。

2. 加载、卸载模块并查看模块打印信息。

## 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

## 参考答案

一、参考答案源码

1、kthread\_sleep.c

|  |
| --- |
| #include <linux/module.h>  #include <linux/sched.h>  #include <linux/pid.h>  #include <linux/wait.h>  #include <linux/list.h>  #include <linux/kthread.h>  #include <linux/delay.h>  MODULE\_LICENSE("GPL");  struct task\_struct \* old\_thread;  int my\_function(void \* argc)  {  int data=-1;  printk("the sub\_thrread pid is:%d\n",current->pid);  printk("the state of the init function is :%ld\n",old\_thread->state);  data=wake\_up\_process(old\_thread);  printk("the state of the init function after wake\_up\_process is :%ld\n",old\_thread->state);  printk("the result of the wake\_up\_process is:%d\n",data);  return 0;  }  static int \_\_init wake\_up\_process\_init(void)  {  char namefrm[]="wake\_up\_process";  int result\_data=-1;  long time\_out;  struct task\_struct \* result;  result=kthread\_create\_on\_node(my\_function,NULL,-1,namefrm);  printk("the pid of the sub\_thread is:%d\n",result->pid);  printk("the current pid is:%d\n",current->pid);  old\_thread=current;  result\_data=wake\_up\_process(result);  printk("the result of wake\_up\_process for new thread is: %d\n",result\_data);  time\_out=schedule\_timeout\_uninterruptible(1000\*10); // 让当前进程进入睡眠状态  result\_data=wake\_up\_process(current);  printk("the result of wake\_up\_process for init thread is: %d\n",result\_data);  return 0;  }  static void \_\_exit wake\_up\_process\_exit(void)  {  printk("Goodbye wake\_up\_process\n");  }  module\_init(wake\_up\_process\_init);  module\_exit(wake\_up\_process\_exit); |

2、Makefile

|  |
| --- |
| ifneq ($(KERNELRELEASE),)  obj-m := kthread\_sleep.o  else  KERNELDIR ?=/root/raspberrypi-kernel  PWD := $(shell pwd)  default:  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  endif  .PHONY:clean  install:  insmod kthread\_sleep.ko  uninstall:  rmmod kthread\_sleep  clean:  -rm \*.mod.c \*.o \*.order \*.symvers \*.ko |

二、内核模块加载结果

