实验八

题目8-1：中断和异常管理——用工作队列实现周期打印helloworld

实验目的

学习使用Linux内核提供的工作队列的创建、调度使用及释放。

实验内容

1. 编写一个内核模块程序，用工作队列实现周期打印helloworld。
2. 加载、卸载模块并查看模块打印信息。

实验设计原理

一、内核工作队列的运行机制

工作队列是从Linux 内核2.5版本开始提供的实现延迟的新机制。
工作队列（workqueue）是另外一种将工作延迟执行的形式。工作队列可以把工作延迟，交由一个内核线程去执行，也就是说，这个下半部分可以在进程上下文中执行。这样，通过工作队列执行的代码能占尽进程上下文的所有优势，且工作队列实现了内核线程的封装，不易出错。最重要的就是工作队列允许被重新调度甚至是睡眠，允许内核代码来请求在将来某个时间调用一个函数，用来处理不是很紧急事件的回调方式处理方法。

二、工作队列的数据结构与编程接口API

1. 表示工作的数据结构（定义在内核源码：include/linux/workqueue.h）

    1. 正常的工作用 <linux/workqueue.h> 中定义的work_struct结构表示：

```
struct work_ struct {
    atomic_ Long_ t data;
    struct tist head entry;
    work_ func_ t func ;
#ifdef CONFIG L OCKDEP
    struct Lockdep map Lockdep map;
#endif
    KABI RESERVE(1)
    KABI RESERVE(2)
    KABI RESERVE (3 )
    KABI RESERVE( 4)
};
```

> 这些结构被连接成链表。当一个工作者线程被唤醒时，它会执行它的链表上的所有工作。工作被执行完毕，它就将相应的work_struct对象从链表上移去。当链表上不再有对象的时候，它就会继续休眠。

    2. 延迟的工作用delayed_work数据结构，可直接使用delay_work将任务推迟执行。

```
struct delayed work {
    struct work_ struct work;
    struct timer List timer :
    /* target workqueue and CPU >timer uses to queue >work */
    struct workqueue_ struct *wq;
    int cpu;
    /* delayed work private data, only used in pc iehp now */
    unsigned long data ;
    KABI RESERVE( 1)
    KABI RESERVE(2)
    KABI RESERVE(3)
    KABI RESERVE(4 )
};
```

2. 工作队列中待执行的函数（定义在内核源码：include/linux/workqueue.h）

    work_struct结构中包含工作队列待执行的函数定义 work_func_t func;
    该工作队列待执行的函数原型是： typedef void (*work_func_t)(struct work_struct *work)
    这个函数会由一个工作者线程执行，因此，函数会运行在进程上下文中。默认情况下，允许响应中断，并且不持有任何锁。如果需要，函数可以睡眠。需要注意的是，尽管该函数运行在进程上下文

中，但它不能访问用户空间，因为内核线程在用户空间没有相关的内存映射。通常在系统调用发生时，内核会代表用户空间的进程运行，此时它才能访问用户空间，也只有在此时它才会映射用户空间的内存。
例如：

```
void work_handle(struct work_struct *work)
{
        printk(KERN_ALERT "Hello World!/n");
}
```

三、工作队列的使用

1. 工作队列的创建
   要使用工作队列，需要先创建工作项，有以下两种方式：
   1. 静态创建

      DECLARE_WORK(n, f);　　定义正常执行的工作项
      DECLARE_DELAYED_WORK(n, f);　定义延后执行的工作项
      其中，n表示工作项的名字，f表示工作项执行的函数。
      这样就会静态地创建一个名为n，待执行函数为f的work_struct结构。
   2. 动态创建、运行时创建：
      通常在内核模块函数中执行以下函数：
      INIT_WORK(*work, _func);*　　初始化正常执行的工作项
      *INIT_DELAYED_WORK(*work, _func);　初始化延后执行的工作项

      其中，*work*表示*work_struct*的任务对象；func表示工作项执行的函数。
      这会动态地初始化一个由work指向的工作。
2. 工作项与工作队列的调度运行
   1. 工作项的调度运行

      工作成功创建后，我们可以调度它了。想要把给定工作的待处理函数提交给缺省的events工作线程，只需调用schedule_work(&work)；work马上就会被调度，一旦其所在的处理器上的工作者线程被唤醒，它就会被执行。有时候并不希望工作马上就被执行，而是希望它经过一段延迟以后再执行。在这种情况下，可以调度它在指定的时间执行：
      schedule_delayed_work(&work,delay);
      这时，&work指向的work_struct直到delay指定的时钟节拍用完以后才会执行。
   2. 工作队列的调度运行
      对于工作队列的调度，则使用以下两个函数：
      □ bool queue_work(struct workqueue_struct *wq, struct work_struct *work)
      调度执行一个指定workqueue中的任务。

      示例：queue_work(queue, &work)
      □ bool queue_delayed_work(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)延迟调度执行一个指定workqueue中的任务，功能与queue_work类似，输入参数多了一个delay。

```
    示例：queue_ delayed_work(queue, &work, 0)
```

3、工作队列的释放

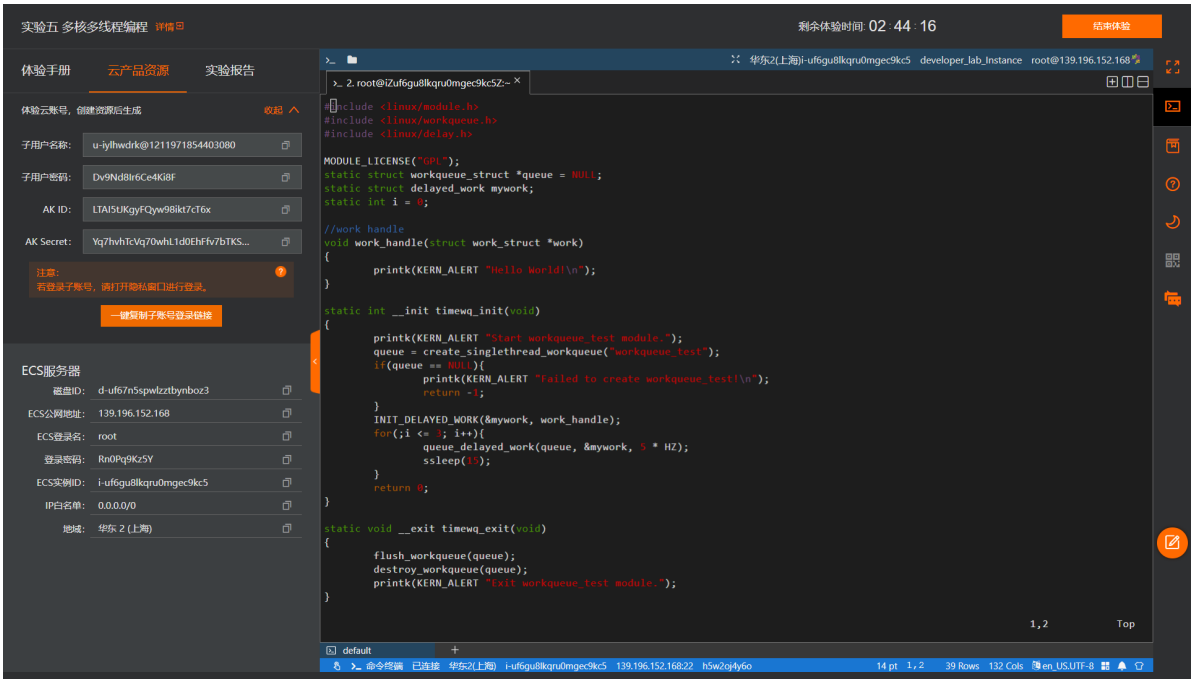
void flush_workqueue(struct workqueue_struct *wq);　//刷新工作队列，等待指定列队中的任务全部执行完毕。

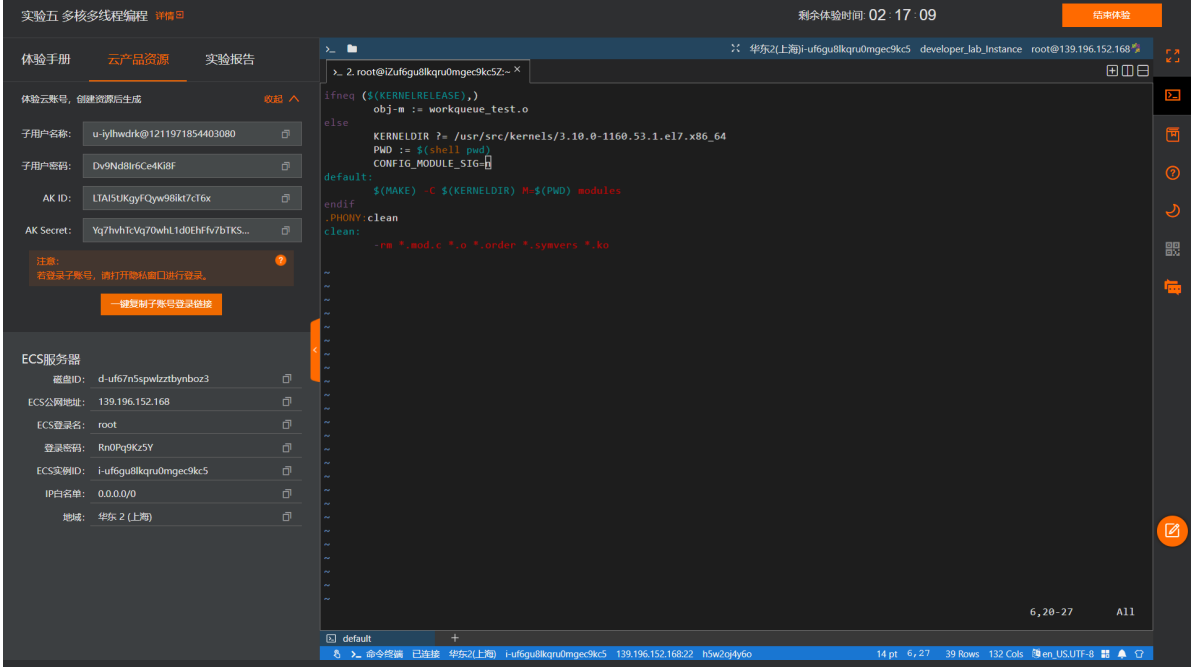
void destroy_workqueue(struct workqueue_struct *wq);　//释放工作队列所占的资源

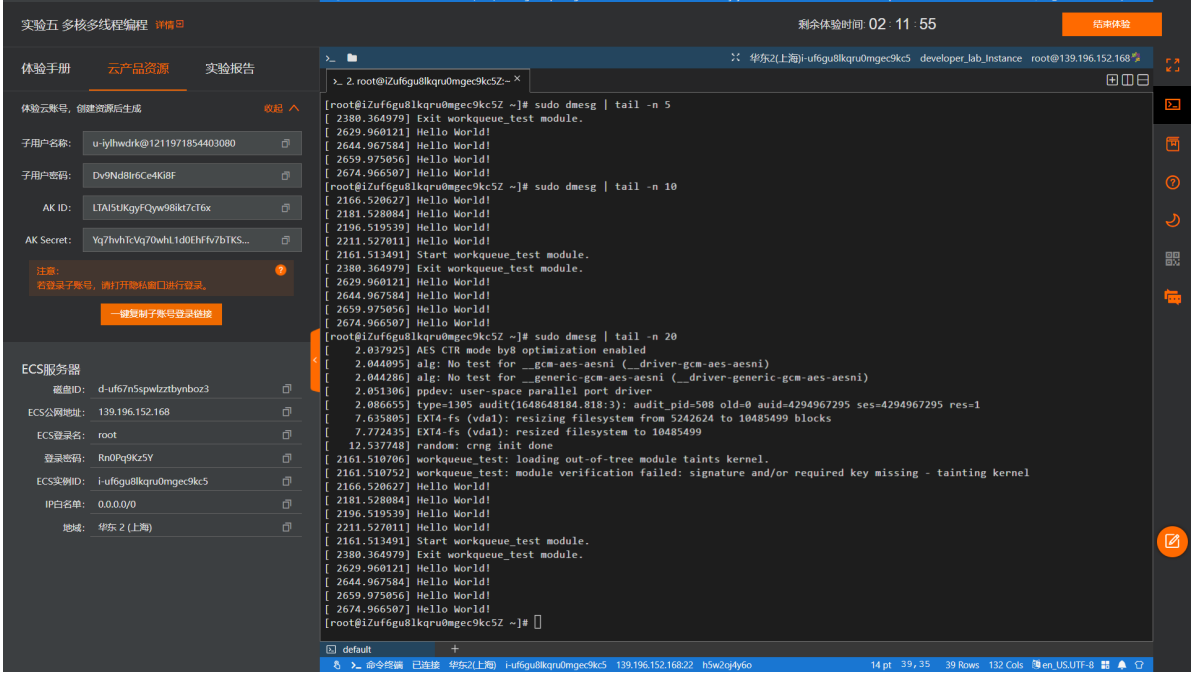
实验步骤

编写C程序代码。首先调用模块初始化函数创建一个名为workqueue_test工作队列，并将“Hello World”的输出工作加载进入队列中。之后4次循环使输出工作以5Hz的延迟进行，并在每次工作后加入15Hz的睡眠时间。最后调用出口函数将工作队列释放。
编写Makefile代码。在基本格式中额外加入CONFIG_MODULE_SIG=n参数来确保加载模块的时候不检查签名，以免在检测未通过后加载失败。

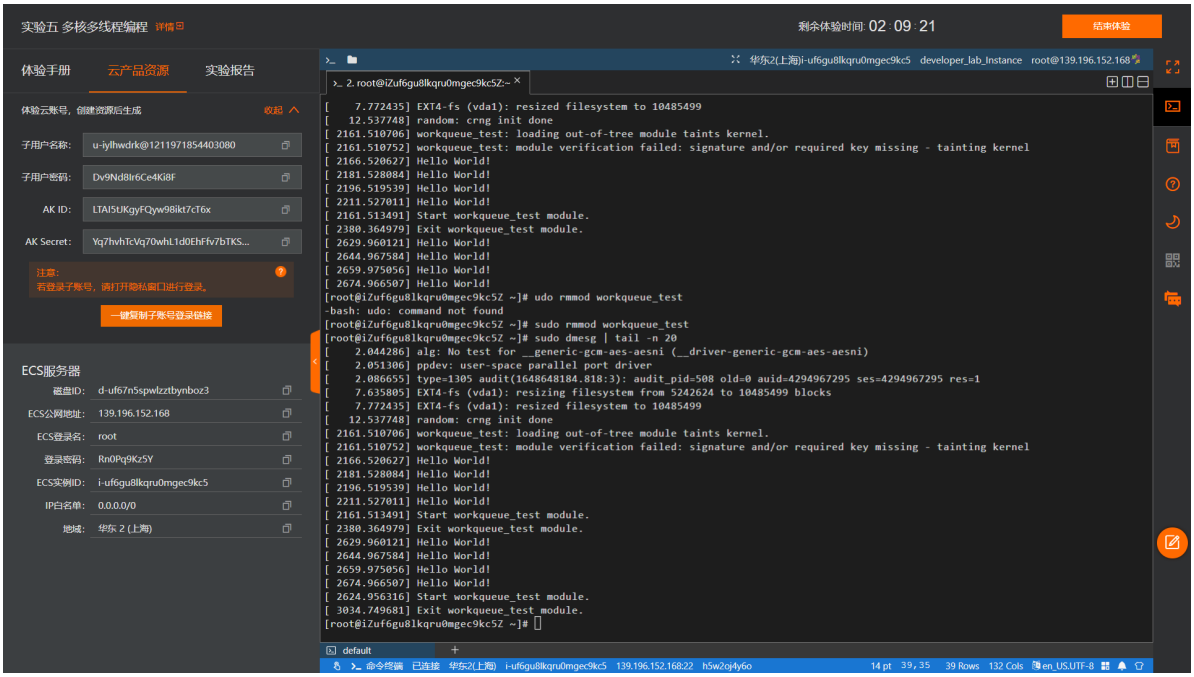
实验结果及分析







实验完成后一共加载两次模块，前一次由于签名未通过在Makefile文件中加入了上述步骤的关闭签名检测的代码。



通过实验最终输出，可以确认模块在载入的5秒后开始执行输出工作，并在之后每隔15秒输出一次，最终输出4行Hello World文本，与代码显示工作队列的执行流程相符。

程序代码

workqueue_test.c

```c
#include <linux/module.h>
#include <linux/workqueue.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");
static struct workqueue_struct *queue = NULL;
static struct delayed_work mywork;
static int i = 0;

//work handle
void work_handle(struct work_struct *work)
{
    printk(KERN_ALERT "Hello World!\n");
}

static int __init timewq_init(void)
{
    printk(KERN_ALERT "Start workqueue_test module.");
    queue = create_singlethread_workqueue("workqueue_test");
    if(queue == NULL){
        printk(KERN_ALERT "Failed to create workqueue_test!\n");
        return -1;
    }
    INIT_DELAYED_WORK(&mywork, work_handle);
    for(;i <= 3; i++){
        queue_delayed_work(queue, &mywork, 5 * HZ);
        ssleep(15);
    }
    return 0;
}
```

```
static void __exit timewq_exit(void)
{
    flush_workqueue(queue);
    destroy_workqueue(queue);
    printk(KERN_ALERT "Exit workqueue_test module.");
}

module_init(timewq_init);
module_exit(timewq_exit);
```

Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := workqueue_test.o
else
    KERNELDIR ?= /usr/src/kernels/3.10.0-1160.53.1.el7.x86_64
PWD := $(shell pwd)
    CONFIG_MODULE_SIG=n
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

题目8-2：中断和异常管理——捕获终端按键信号

实验目的

1. 了解Linux的信号处理机制；
2. 学习使用Linux内置信号处理函数。

实验内容

1. 在用户态编写一个信号捕获程序，捕获终端按键信号（包括ctrl+c、ctrl+z、ctrl+\）。
2. 编译上述程序后运行，在终端输入按键信号（ctrl+c、ctrl+z、ctrl+\），查看输出信息。

实验设计原理

一、Linux信号处理机制

1. 基本概念

   Linux提供的信号机制是一种进程间异步的通信机制，每个进程在运行时，都要通过信号机制来检查是否有信号到达，若有，便中断正在执行的程序，转向与该信号相对应的处理程序，以完成对该事件的处理；处理结束后再返回到原来的断点继续执行。实质上，信号机制是对中断机制的一种模拟，在实现上是一种软中断。

2. 信号的产生

   信号的生成来自内核，让内核生成信号的请求来自3个地方：

   - 用户：用户能够通过终端按键产生信号，例如；

ctrl+c ----> 2) SIGINT(终止、中断)

ctrl+\ ----> 3) SIGQUIT(退出)

ctrl+z ----> 20) SIGTSTP（暂时、停止）

或者是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号

- 内核：当进程执行出错时，内核会给进程发送一个信号，例如非法段存取(内存访问违规)、浮点数溢出等；
- 进程：一个进程可以通过系统调用kill给另一个进程发送信号，一个进程可以通过信号和另外一个进程进行通信。
  当信号发送到某个进程中时，操作系统会中断该进程的正常流程，并进入相应的信号处理函数执行操作，完成后再回到中断的地方继续执行。需要说明的是，信号只是用于通知进程发生了某个事件，除了信号本身的信息之外，并不具备传递用户数据的功能。

3. 信号的响应动作/处理
   每个信号都有自己的响应动作，当接收到信号时，进程会根据信号的响应动作执行相应的操作，信号的响应动作有以下几种：
   1. 中止进程(Term)
   2. 忽略信号(Ign)
   3. 中止进程并保存内存信息(Core)
   4. 停止进程(Stop)
   5. 继续运行进程(Cont)

   用户可以通过signal或sigaction函数修改信号的响应动作（也就是常说的“注册信号”）。另外，在多线程中，各线程的信号响应动作都是相同的，不能对某个线程设置独立的响应动作。
4. 信号类型
   Linux支持的信号类型可以参考kill -l显示的编号列表，其中1-31为常规信号，34-64为实时信号。

```
1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL    5) SIGTRAP
 6) SIGABRT   7) SIGBUS    8) SIGFPE    9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG  24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF 28) SIGWINCH    29) SIGIO   30) SIGPWR
31) SIGSYS  34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

二、信号处理函数signal()

函数原型：void (*signal (int signum ,void (*handler)(int))) (int) ;

功能：设置捕捉某一信号后，对应的处理函数。

头文件：#include <signal.h>

参数说明：
signum：指定的信号的编号（或捕捉的信号），可以使用头文件中规定的宏；
handle：函数指针，是信号到来时需要运行的处理函数，参数是signal()的第一个参数signum。
对于第二个参数，可以设置为SIG_IG，表示忽略第一个参数的信；可以设置为SIG_DFL，表示采用默认的方式处理信号；也可以指定一个函数地址，自己实现处理方式。

返回值：
运行成功，返回原信号处理函数的指针；失败则返回SIG_ERR。

说明：
　signal()用于注册一个信号捕捉函数，而捕捉信号的操作由内核进行。
捕获信号示例：

```
void sighandle(int sig)
{
     printf("捕获到信号%d", sig);
     exit(0);         // 释放资源
}

int main()
{
    .....
    signal(SIGINT, sighandle);  //设置信号处理函数
    .....
}
```

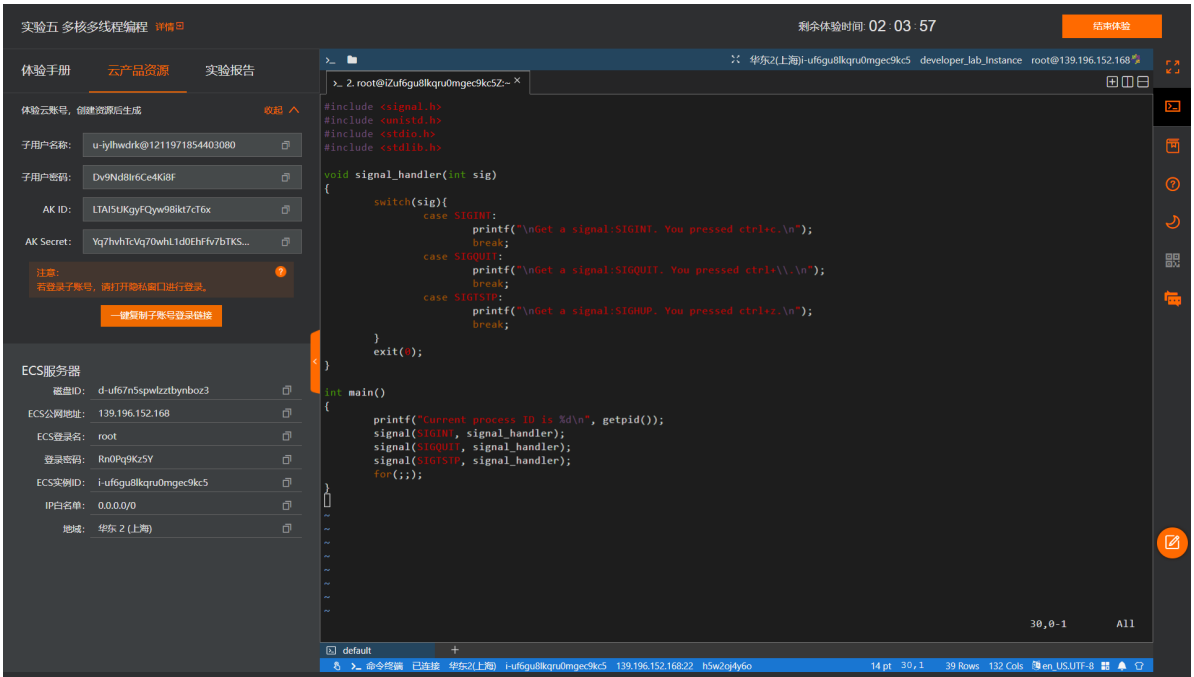
三、信号与中断的异同点

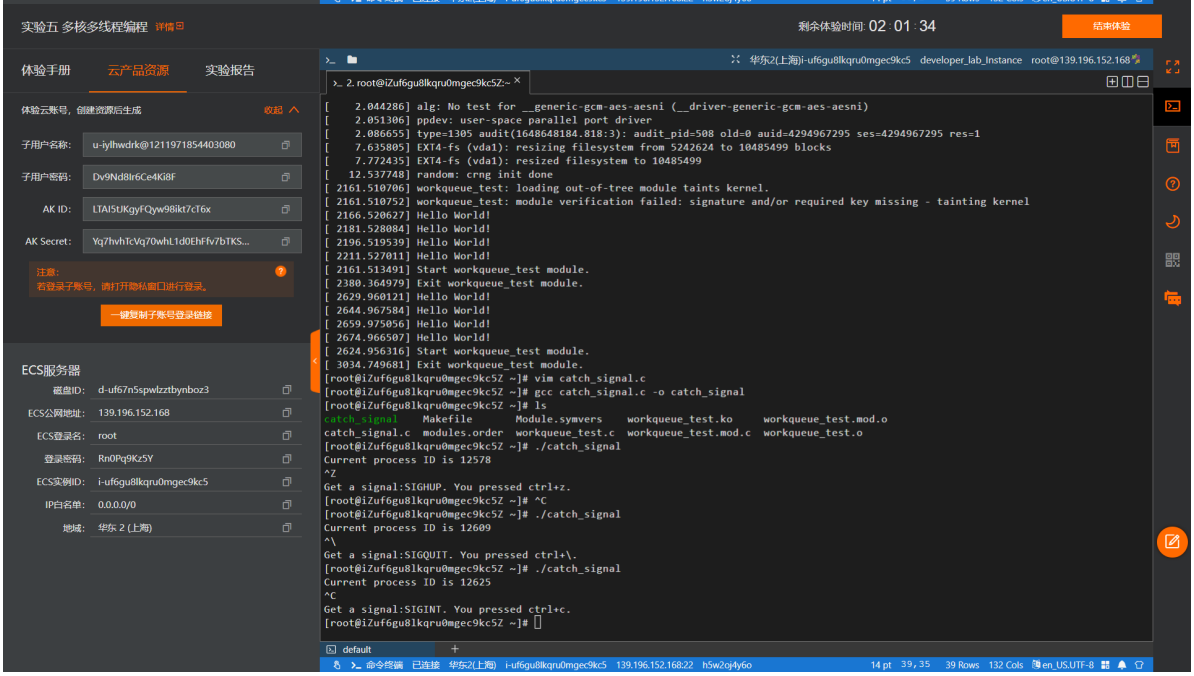
1. 信号与中断的相似点
   （1）采用了相同的异步通信方式；
   （2）当检测出有信号或中断请求是，都暂停正在执行的程序，转而去执行相应的处理程序；
   （3）都在处理完毕后返回到原来的断点；
   （4）对信号或中断都可进行评比。
2. 信号与中断的区别
   （1）中断有优先级，而信号没有优先级，所有信号都是平等的；
   （2）信号处理程序是在用户态下运行的；而中断处理程序是在内核态下运行的；
   （3）中断响应是及时的，而信号响应通常有较大的时间延迟。

实验步骤

编写C程序。利用switch根据Linux从键盘读入的不同指令输出不同文本，并将当前进程的PID打印出来。

实验结果及分析





当获取到用户输入ctrl+c，输出文本得到了“中断信号”；

当获取到用户输入ctrl+\，输出文本得到了“退出信号”；

当获取到用户输入ctrl+z，输出文本得到了“停止信号”。

需要额外注意的是：虽然实验文档供给的代码将ctrl+z标注为SIGHUP，但这是终端断连信号，与其本意SIGSTP有巨大的差异，后者才是正确的信号。

程序代码

catch_signal.c

```c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void signal_handler(int sig)
{
```

```
    switch(sig){
        case SIGINT:
            printf("\nGet a signal:SIGINT. You pressed ctrl+c.\n");
            break;
        case SIGQUIT:
            printf("\nGet a signal:SIGQUIT. You pressed ctrl+\\.\n");
            break;
        case SIGTSTP:
            printf("\nGet a signal:SIGSTP. You pressed ctrl+z.\n");
            break;
    }
    exit(0);
}

int main()
{
    printf("Current process ID is %d\n", getpid());
    signal(SIGINT, signal_handler);
    signal(SIGQUIT, signal_handler);
    signal(SIGTSTP, signal_handler);
    for(;;);
}
```

题目8-3：内核时间管理——调用内核时钟接口监控运行时间

实验目的

学习使用Linux内核时钟接口的调用。

实验内容

1. 调用内核时钟接口，编写内核模块，监控实现累加计算sum=1+2+3+...+100000 所花时间。
2. 加载、卸载模块并查看模块打印信息。

实验设计原理

1. timeval结构体

   头文件：<linux/time.h>

```
struct timeval {
        __kernel_time_t         tv_sec;         /* seconds */
        __kernel_suseconds_t    tv_usec;         /* microseconds */
 };
```

其中tv_sec是自1970年1月1日 00:00:00 起到现在的秒数。而tv_usec是当前秒数已经经过的微秒数。

2. do_gettimeofday()

   头文件：<linux/time.h>

   函数原型：void do_gettimeofday(struct timeval *tv);

功能：返回自1970-01-01 00:00:00到现在的秒数，及当前秒经过的毫秒数，保存在tv指向的timeval 结构体中。

3. rtc_time结构体

   头文件<linux/rtc.h>

```
struct rtc_time {
     int tm_sec;            // 表「秒」数，在[0,61]之间，多出来的两秒是用来处理跳秒问题用的。
     int tm_min;           // 表「分」数，在[0,59]之间。
     int tm_hour;          // 表「时」数，在[0,23]之间。
     int tm_mday;         // 表「本月第几日」，在[1,31]之间。
     int tm_mon;          // 表「本年第几月」，在[0,11]之间。
     int tm_year;          // 要加1900表示那一年。
     int tm_wday;         // 表「本周第几日」，在[0,6]之间。
     int tm_yday;         // 表「本年第几日」，在[0,365]之间，闰年有366日。
     int tm_isdst;         // 表是否为「日光节约时间」。
 };
```

年份加上1900，月份加上1，小时加上8。

4. rtc_time_to_tm()

   头文件：<linux/rtc.h>

   函数原型：void rtc_time_to_tm(unsigned long time, struct rtc_time *tm);

   功能：将time存储的秒数转换为年月日时分秒等信息保存在rtc_time结构体中。

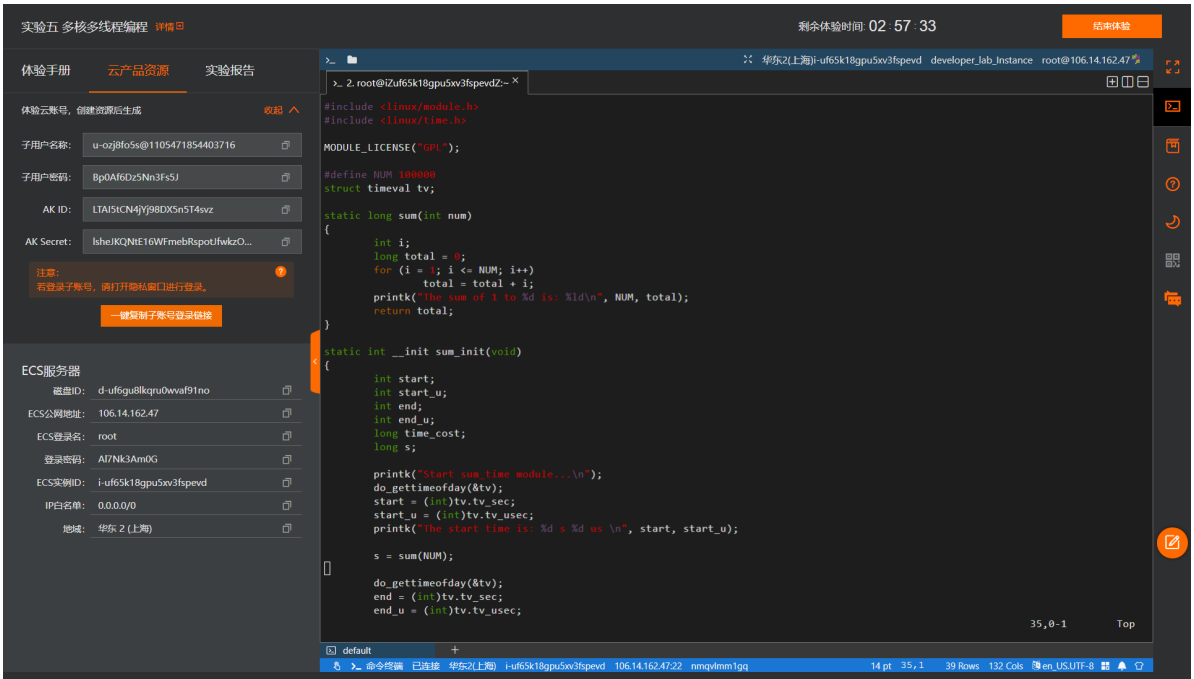
   参数：time为秒数，可以是do_gettimeofday()函数获取的秒数。tm是rtc_time结构体指针，结构体中存放了年月日时分秒等信息。

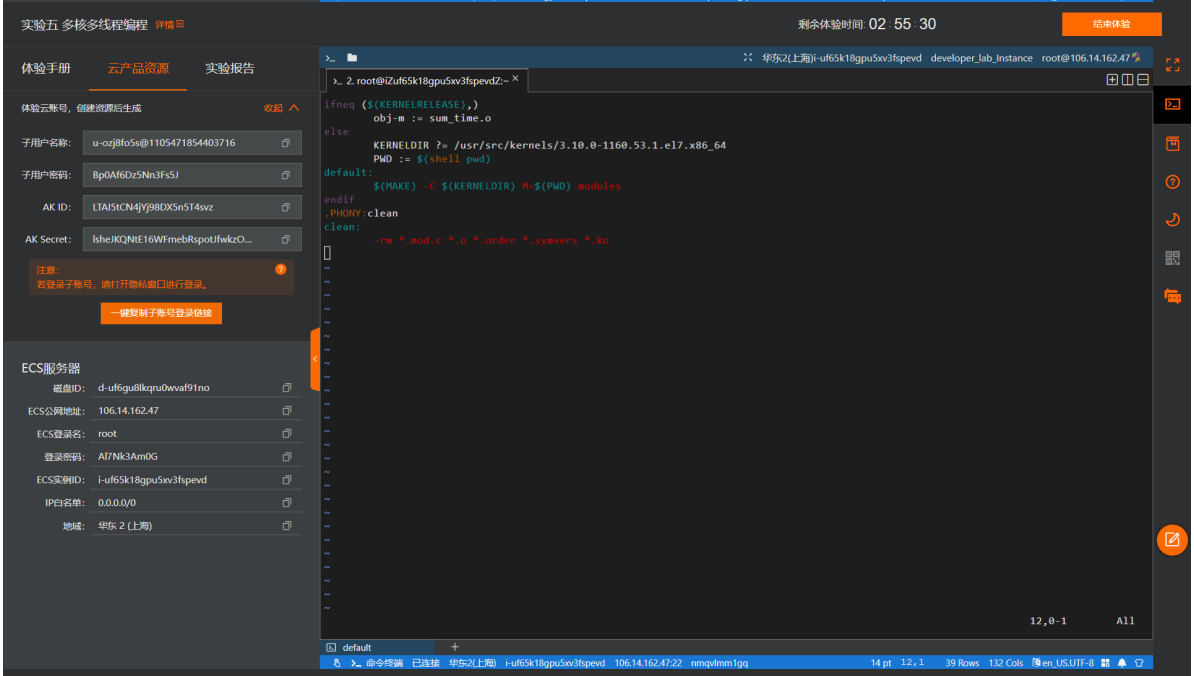
实验步骤

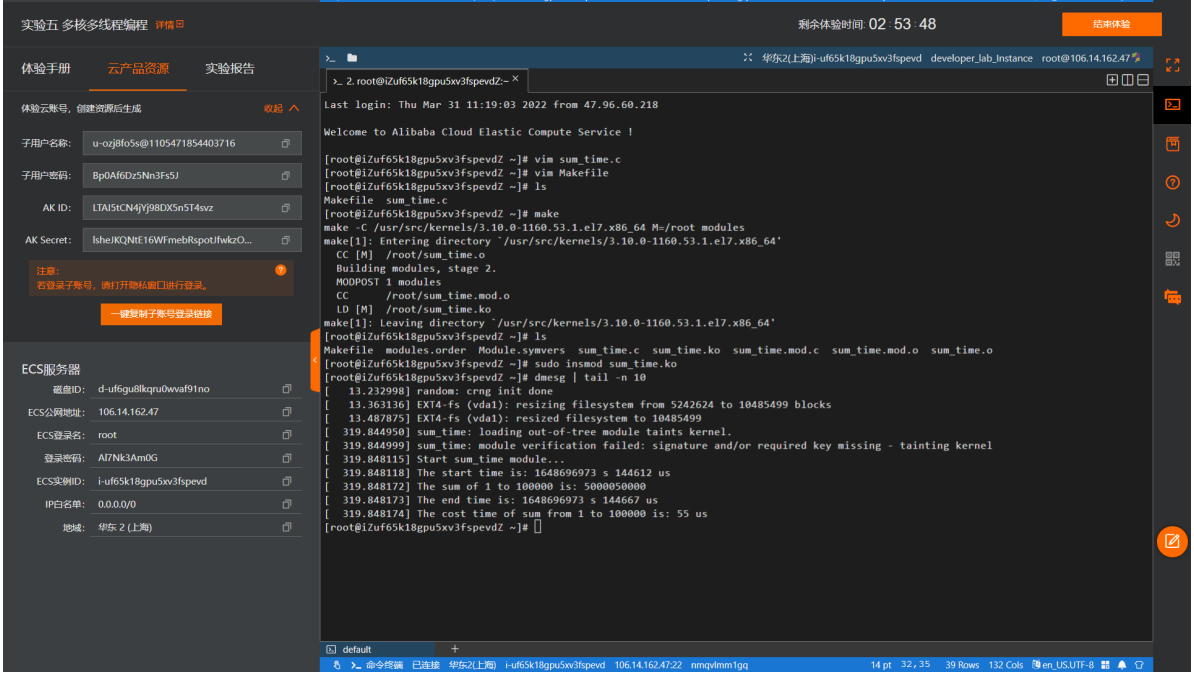
编写C程序，程序主体以及测试对象为一个从1到100000的累加for循环。在调用累加函数之前使用模块初始化函数包装它：首先使用gettimeofday以及timeval获取当前系统时间并存储其秒级及微秒级部分，作为开始时间打印到屏幕上；随后执行累加函数；最后再次调用gettimeofday、timeval存储终止时间并打印，最终得到与开始时间的差打印到屏幕上即为总运行时间。最后退出并释放该模块。

实验结果及分析

实验结果







最终结果为累加求和总运行时间为55μs。

另关于实验6-5-2的分析

在选择内核时间管理部分的实验内容时，初选择的题目为6-5-2编写timer定时器。但因为笔者最终发现在Linux内核3.x版本中的定时器timer数据结构存在一些无法轻易调优的缺陷，而且内核产生了无法定位问题原因的软死锁，故放弃了该实验的进行转而进行本文的6-5-3。

我们先假设系统的内核抢占已经被开启：

```
CONFIG_PREEMPT=y     //即在内核配置文件中修改内核抢占为开启
```

实验中的timer初始化回调函数是这样定义的：

```
    static int __init timer_init(void)
    {
        printk("Start timer_example module...\n");
        timer.expires = jiffies + 10 * HZ;
        timer.function = print;
        add_timer(&timer);
        return 0;
    }
```

看着没有任何问题的代码，看起来和死锁丝毫没有关系。

当我们详细了解了Linux timer的执行原理后就会明白：

- 挂载在同一CPU上的所有过期timer是顺序遍历执行的。
- 一轮timer的顺序遍历执行是持有自旋锁的。

这意味着在执行一轮过期timer的过程中，watchdog实时线程将无法被调度处理死锁，这意味着：

- 同一CPU上的过期timer积累到一定量，其回调函数的延时之和大于20秒，将会产生soft lockup（软死锁）。

  > soft lockup是指CPU被内核代码占据，以至于无法执行其它进程的死锁情况。检测soft lockup的原理是给每个CPU分配一个定时执行的内核线程[watchdog/x]，如果该线程在设定的期限内没有得到执行的话就意味着发生了soft lockup，[watchdog/x]是SCHED_FIFO实时进程，优先级为最高的99，拥有优先运行的特权。

个人推测：实验中仅仅执行加载调度的print函数的timer，也有触发soft lockup的可能性。

得到这个推测前，我们需要先着手于Linux timer的工作机制来理清思路。

可以把timer的执行过程抽象成下面的逻辑：

```
    run_timers()
    {
      while (now > base.early_jiffies) {
        for_each_timer(timer, base.list) {
          detach_timer(timer)
          forward_early_jiffies(base)
          call_timer_fn(timer)
        }
      }
    }
```

内核把当前过期的timer轮流执行到结束，而在系统开启内核抢占时，我们假设run_timers是在时钟中断退出时的软中断上下文中执行的，此时它不能被watchdog抢占。

视角回到前文timer回调函数中执行修改延迟的操作：

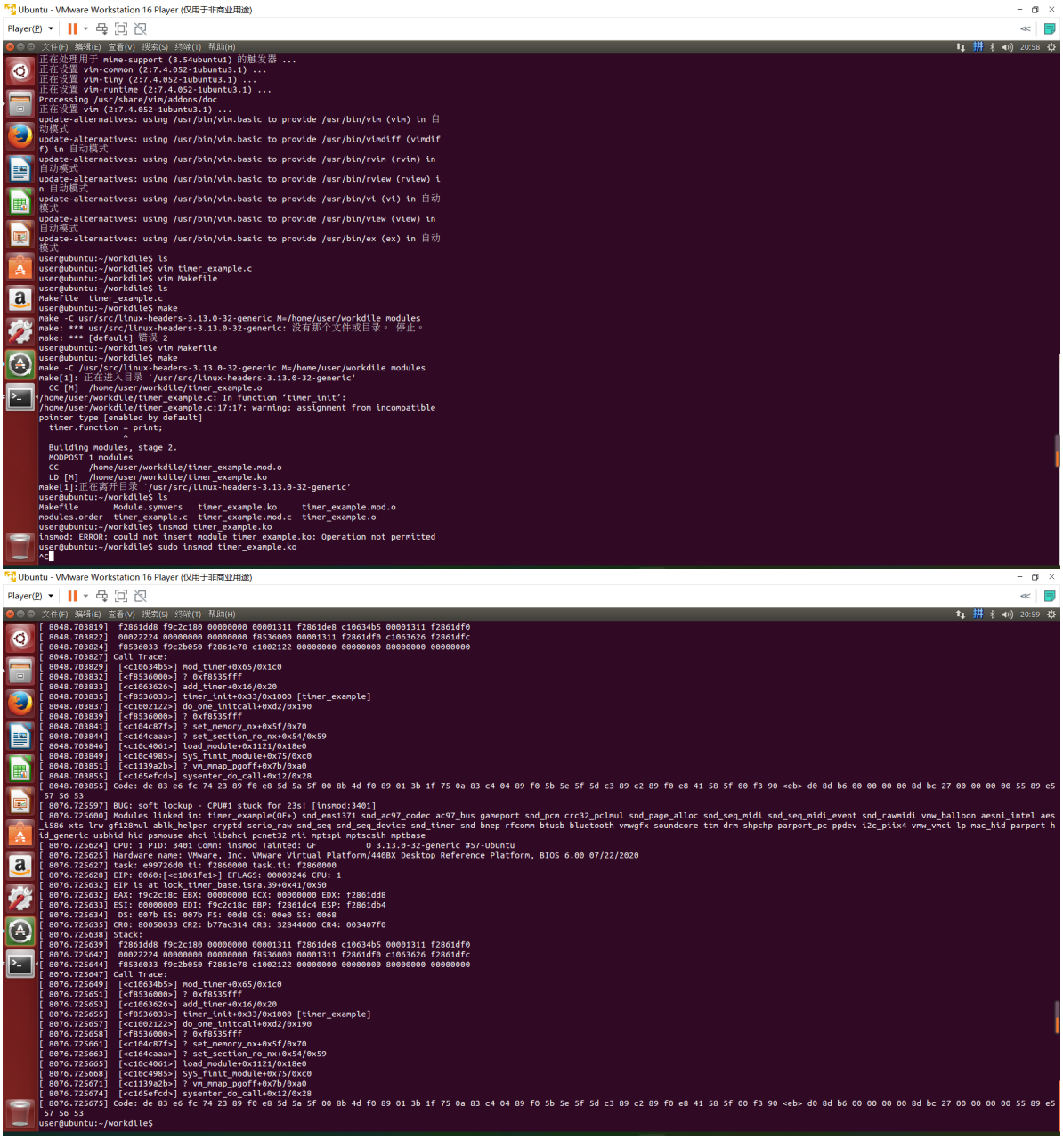
```
timer.expires = jiffies + 10 * HZ;
```

实际上它在逻辑处理中将timer又插回了list，如果我们把这个list看作是一条时间线的话，它事实上只是往后移了expires这么远的距离。

假设所有timer的expire都是固定的常量，若：

- 我们注册了足够多的timer，多到依据其expires重新填入队列时恰好能填补空隙。
- 我们的timer回调函数耗时恰好和timer的expires一致。

那么，一轮timer的执行将永远不会结束：



单核跑满，timer已经拼接成龙，23秒后，我们将看到soft lockup。

当然这只是不保证正确的一种猜想，作为这一大部分的小插曲记录在文档中。而通过后来的不断测试证明事实是只要在初始化函数中通过add_timer(&timer)注册timer就会无条件进入软死锁，原因未知。

程序代码

sum_time.c

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

MODULE_LICENSE("GPL");

#define NUM 100000
struct timeval tv;

static long sum(int num)
{
    int i;
    long total = 0;
    for (i = 1; i <= NUM; i++)
        total = total + i;
    printk("The sum of 1 to %d is: %ld\n", NUM, total);
    return total;
}

static int __init sum_init(void)
{
    int start;
    int start_u;
    int end;
    int end_u;
    long time_cost;
    long s;

    printk("Start sum_time module...\n");
    do_gettimeofday(&tv);
    start = (int)tv.tv_sec;
    start_u = (int)tv.tv_usec;
    printk("The start time is: %d s %d us \n", start, start_u);

    s = sum(NUM);

    do_gettimeofday(&tv);
    end = (int)tv.tv_sec;
    end_u = (int)tv.tv_usec;
    printk("The end time is: %d s %d us \n", end, end_u);
    time_cost = (end - start) * 1000000 + end_u - start_u;
    printk("The cost time of sum from 1 to %d is: %ld us \n", NUM, time_cost);
    return 0;
}

static void __exit sum_exit(void)
{
    printk("Exit sum_time module...\n");
}

module_init(sum_init);
module_exit(sum_exit);
```

Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := sum_time.o
else
    KERNELDIR ?= /usr/src/kernels/3.10.0-1160.53.1.el7.x86_64
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```