基于 Linux 内核的操作系统实验

- 概述：

编译linux-3.9.4内核代码，并通过打补丁的方式，基于linux-3.9.4源代码运行我们自己简单的、裁剪的操作系统内核。

- Simulator： qemu
- Linux version: linux-3.9.4

配置步骤：

安装**qemu**模拟器：

- sudo apt-get install qemu

创建**qemu**运行的链接

- sudo ln -s /usr/bin/qemu-system-i386 /usr/bin/qemu

下载 **linux-3.9.4**内核源代码（有心的同学熟悉下**wget**的命令～经常使用的）

- wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.4.tar.xz

下载补丁文件，该补丁文件修改了内核文件的部分代码，以运行我们自己设计的操作系统内核

- wget https://raw.github.com/mengning/mykernel/master/mykernel_for_linux3.9.4sc.patch

解压 **linux-3.9.4** 内核源代码

- xz -d linux.3.9.4.tar.xz

- tar -xvf linux-3.9.4.tar

进入**linux-3.9.4** 文件夹

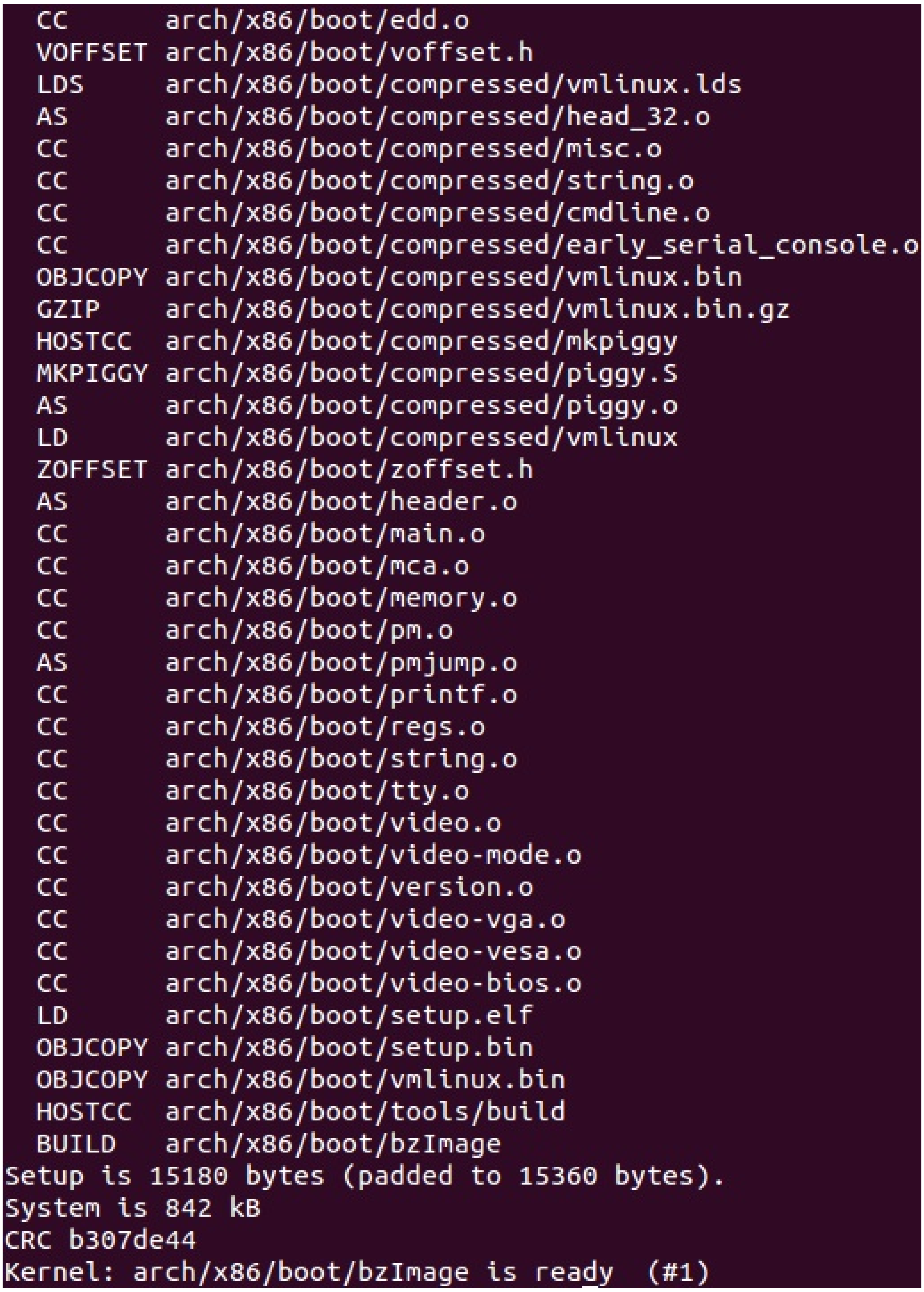
- cd linux-3.9.4

安装**mykernel**补丁

- patch -p1 < ../mykernel_for_linux3.9.4sc.patch

编译**linux-3.9.4**内核源码

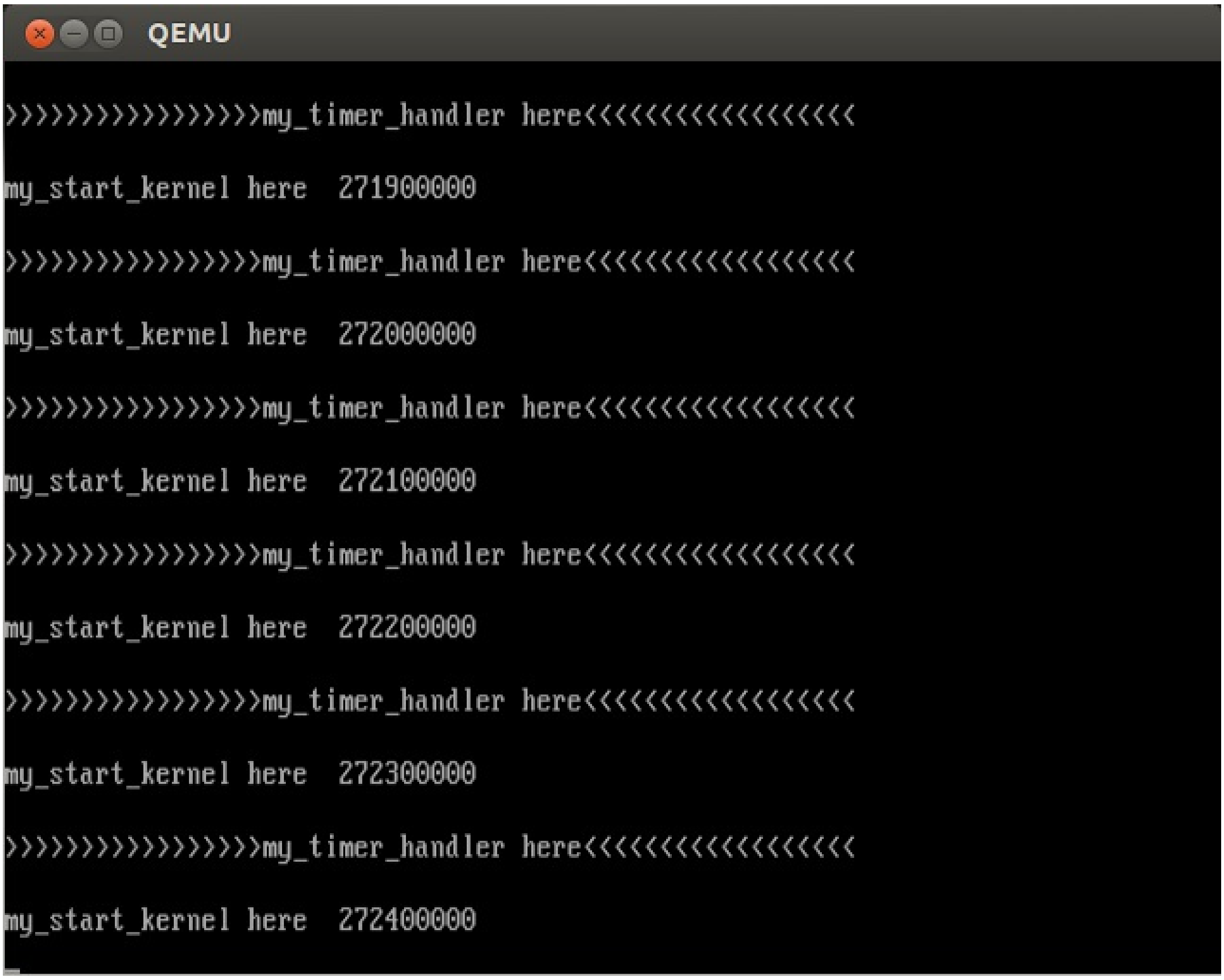
- make allnoconfig
- make



```
  CC      arch/x86/boot/edd.o
  VOFFSET arch/x86/boot/voffset.h
  LDS     arch/x86/boot/compressed/vmlinux.lds
  AS      arch/x86/boot/compressed/head_32.o
  CC      arch/x86/boot/compressed/misc.o
  CC      arch/x86/boot/compressed/string.o
  CC      arch/x86/boot/compressed/cmdline.o
  CC      arch/x86/boot/compressed/early_serial_console.o
  OBJCOPY arch/x86/boot/compressed/vmlinux.bin
  GZIP    arch/x86/boot/compressed/vmlinux.bin.gz
  HOSTCC  arch/x86/boot/compressed/mkpiggy
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
  LD      arch/x86/boot/compressed/vmlinux
  ZOFFSET arch/x86/boot/zoffset.h
  AS      arch/x86/boot/header.o
  CC      arch/x86/boot/main.o
  CC      arch/x86/boot/mca.o
  CC      arch/x86/boot/memory.o
  CC      arch/x86/boot/pm.o
  AS      arch/x86/boot/pmjump.o
  CC      arch/x86/boot/printf.o
  CC      arch/x86/boot/regs.o
  CC      arch/x86/boot/string.o
  CC      arch/x86/boot/tty.o
  CC      arch/x86/boot/video.o
  CC      arch/x86/boot/video-mode.o
  CC      arch/x86/boot/version.o
  CC      arch/x86/boot/video-vga.o
  CC      arch/x86/boot/video-vesa.o
  CC      arch/x86/boot/video-bios.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  OBJCOPY arch/x86/boot/vmlinux.bin
  HOSTCC  arch/x86/boot/tools/build
  BUILD   arch/x86/boot/bzImage
Setup is 15180 bytes (padded to 15360 bytes).
System is 842 kB
CRC b307de44
Kernel: arch/x86/boot/bzImage is ready  (#1)
```

使用**qemu**运行**kernel**， 可以看到 **my_start_kernel**在执行，同时 **my_timer_handler**时钟中断处理程序周期性执行

- qemu -kernel arch/x86/boot/bzImage

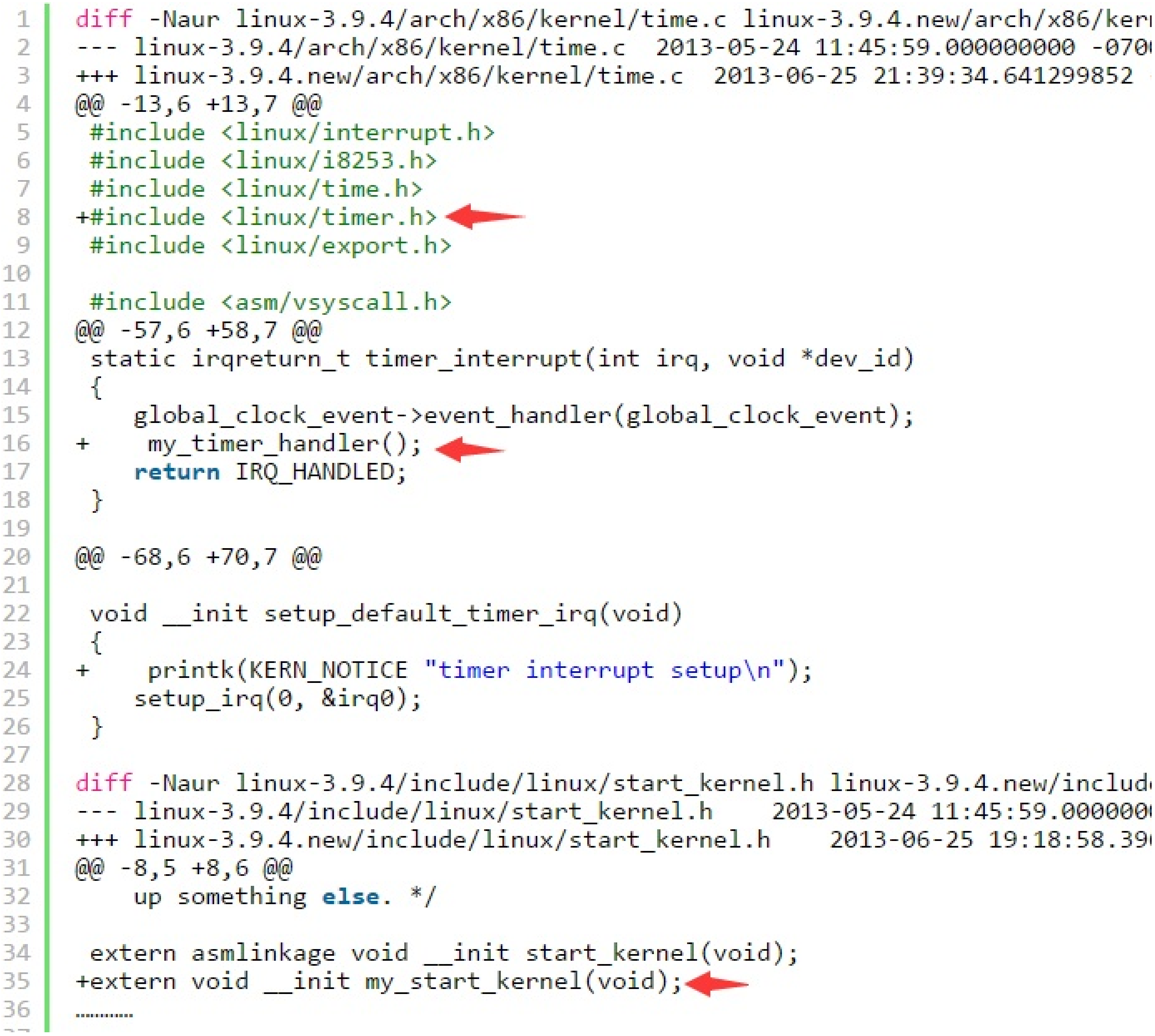


```
>>>>>>>>>>>>>>>>>my_timer_handler here<<<<<<<<<<<<<<<<<<

my_start_kernel here  271900000

>>>>>>>>>>>>>>>>>my_timer_handler here<<<<<<<<<<<<<<<<<<

my_start_kernel here  272000000

>>>>>>>>>>>>>>>>>my_timer_handler here<<<<<<<<<<<<<<<<<<

my_start_kernel here  272100000

>>>>>>>>>>>>>>>>>my_timer_handler here<<<<<<<<<<<<<<<<<<

my_start_kernel here  272200000

>>>>>>>>>>>>>>>>>my_timer_handler here<<<<<<<<<<<<<<<<<<

my_start_kernel here  272300000

>>>>>>>>>>>>>>>>>my_timer_handler here<<<<<<<<<<<<<<<<<<

my_start_kernel here  272400000
_
```

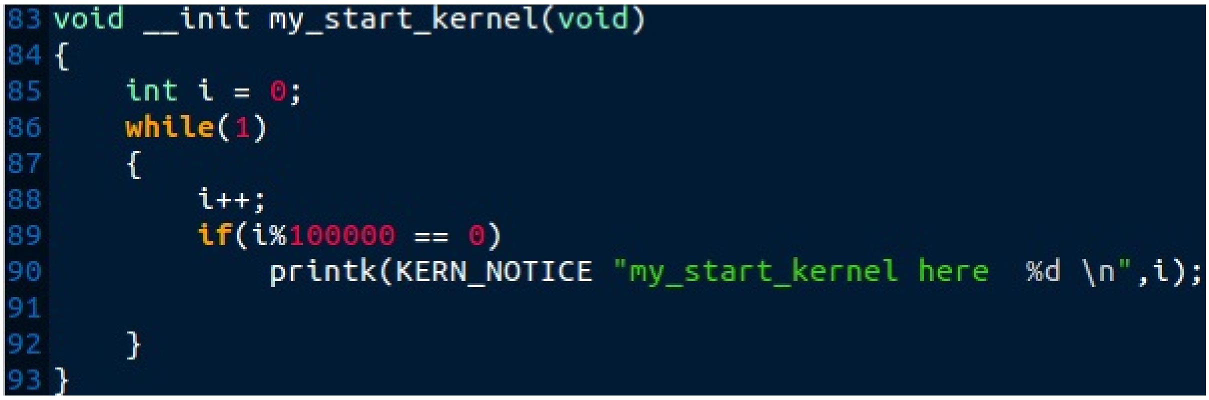
这是因为补丁文件修改了内核代码，加入运行我们自定义的mykernel内核的代码。

通过查看补丁文件，我们大概了解其增加了头文件 timer.h,添加时间处理函数 my_timer_handler()和内核启动函数 my_start_kernel()以及修改 makefile 令其多编译 mykernel 文件。 linux-3.9.4 内核的入口函数是linux-3.9.4 >> init >> main.c >> start_kernel(),在其最后添加外部函数 my_start_kernel(),使得能够在内核启动阶段运行我们自己编写的内核代码。

此时,可以进入 mykernel 文件夹查看 mymain.c 和 myinterrupt.c,我们的 mykernel 内核目前只实现了一个进程(每隔 100000 时间单位打印输出一条语句)和一个基于时间的中断。

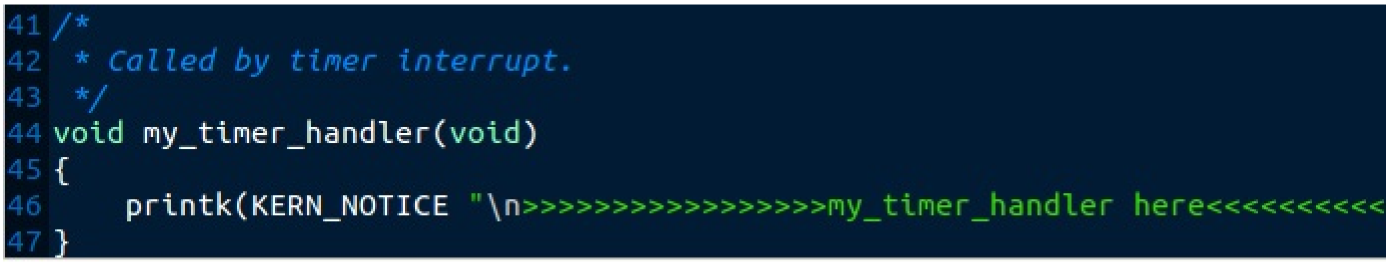


```
diff -Naur linux-3.9.4/arch/x86/kernel/time.c linux-3.9.4.new/arch/x86/kern
--- linux-3.9.4/arch/x86/kernel/time.c  2013-05-24 11:45:59.000000000 -070
+++ linux-3.9.4.new/arch/x86/kernel/time.c  2013-06-25 21:39:34.641299852
@@ -13,6 +13,7 @@
 #include <linux/interrupt.h>
 #include <linux/i8253.h>
 #include <linux/time.h>
+#include <linux/timer.h>
 #include <linux/export.h>

 #include <asm/vsyscall.h>
@@ -57,6 +58,7 @@
 static irqreturn_t timer_interrupt(int irq, void *dev_id)
 {
    global_clock_event->event_handler(global_clock_event);
+    my_timer_handler();
    return IRQ_HANDLED;
 }

@@ -68,6 +70,7 @@

 void __init setup_default_timer_irq(void)
 {
+    printk(KERN_NOTICE "timer interrupt setup\n");
    setup_irq(0, &irq0);
 }

diff -Naur linux-3.9.4/include/linux/start_kernel.h linux-3.9.4.new/includ
--- linux-3.9.4/include/linux/start_kernel.h    2013-05-24 11:45:59.000000
+++ linux-3.9.4.new/include/linux/start_kernel.h    2013-06-25 19:18:58.39
@@ -8,5 +8,6 @@
    up something else. */

 extern asmlinkage void __init start_kernel(void);
+extern void __init my_start_kernel(void);
............
```



```
void __init my_start_kernel(void)
{
    int i = 0;
    while(1)
    {
        i++;
        if(i%100000 == 0)
            printk(KERN_NOTICE "my_start_kernel here  %d \n",i);

    }
}
```

mymain.c

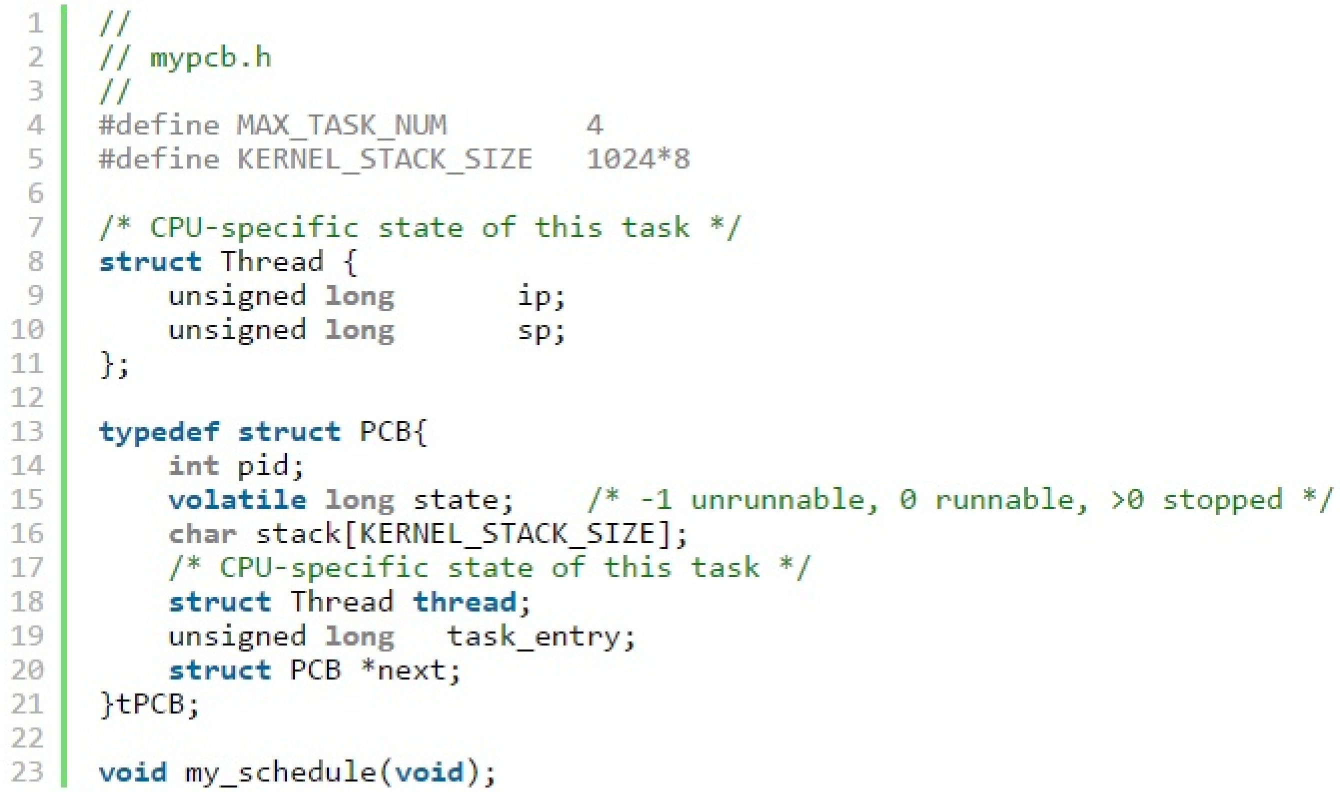


```c
/*
 * Called by timer interrupt.
 */
void my_timer_handler(void)
{
    printk(KERN_NOTICE "\n>>>>>>>>>>>>>>>>>my_timer_handler here<<<<<<<<<<
}
```

myinterrupt.c

这里，我们需要实现一个简单的时间片轮转多道程序内核代码（Round-Robin Scheduling)，与pintos的对应类别，比pintos简单很多。

主要修改linux-3.9.4/mukernel下的3个文件，mypcb.h,mymain.c,myinterrupt.c,这里直接使用github上已经实现好的代码。

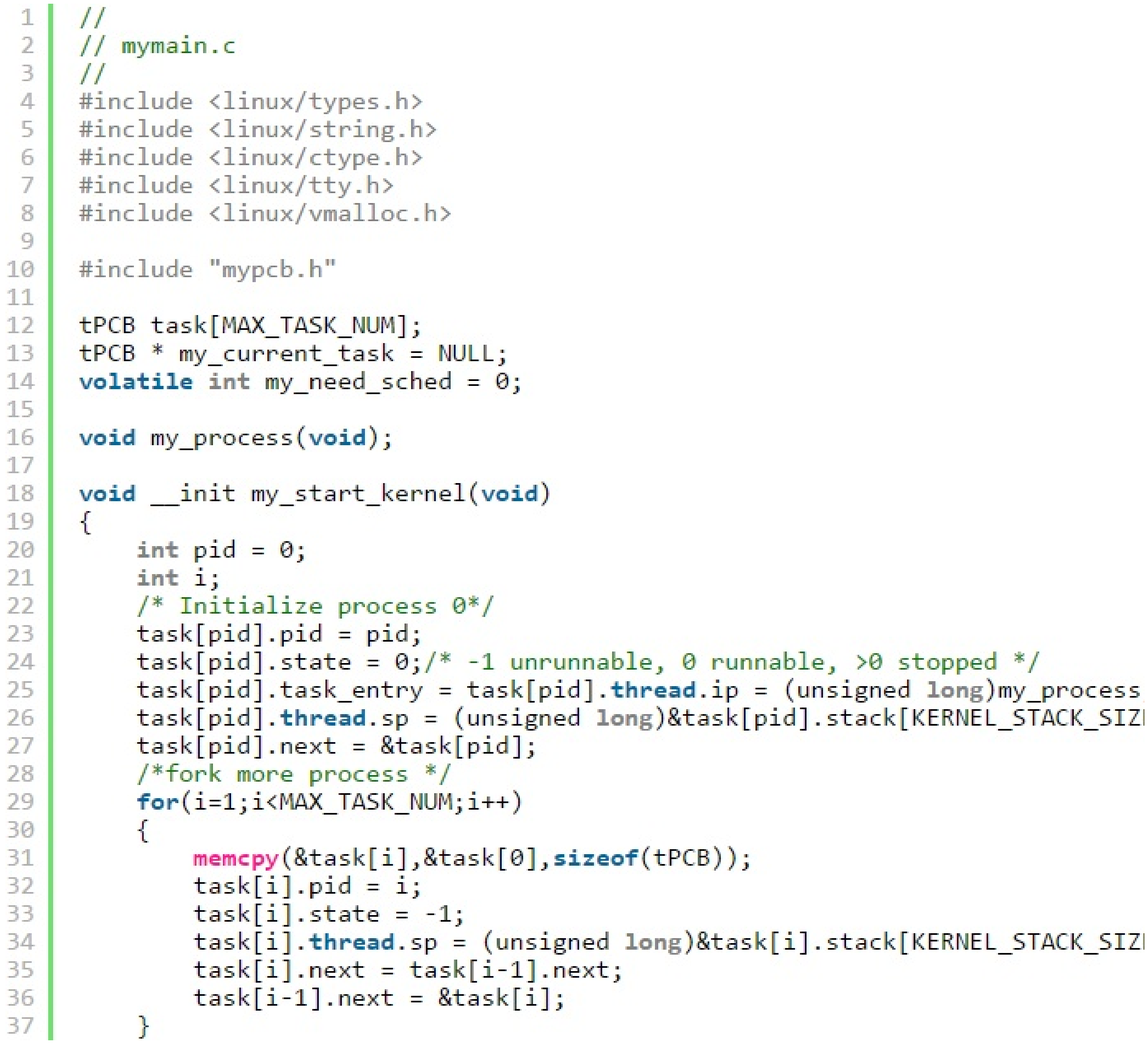


```c
//
// mypcb.h
//
#define MAX_TASK_NUM        4
#define KERNEL_STACK_SIZE   1024*8

/* CPU-specific state of this task */
struct Thread {
    unsigned long       ip;
    unsigned long       sp;
};

typedef struct PCB{
    int pid;
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    char stack[KERNEL_STACK_SIZE];
    /* CPU-specific state of this task */
    struct Thread thread;
    unsigned long   task_entry;
    struct PCB *next;
}tPCB;

void my_schedule(void);
```

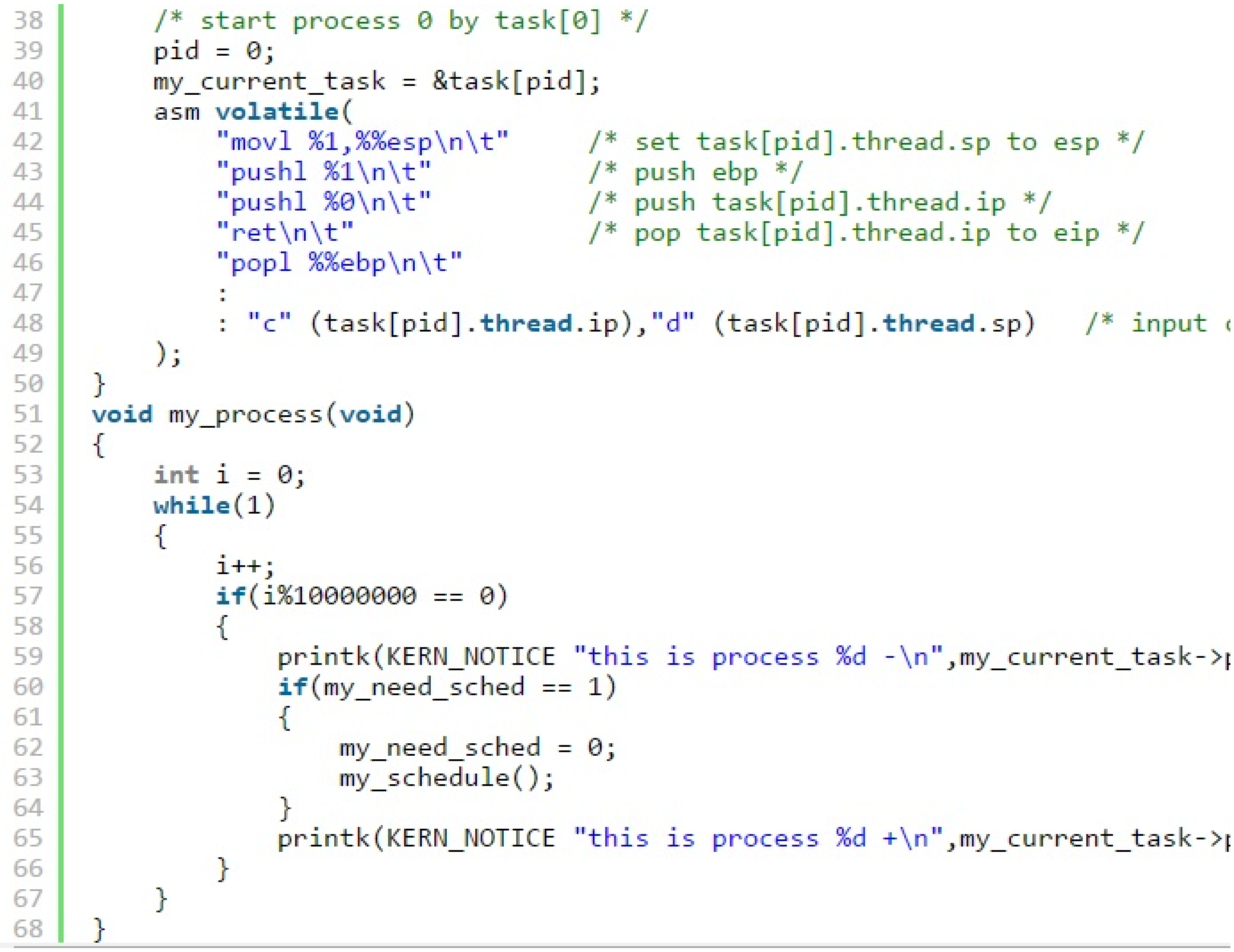
mypcb.h

mypcb.h 定义了thread结构体和PCB进程控制块。

- 注：进程状态：就绪（-1）、运行（0），阻塞（>0）;
- 了解关键字 volatile；
- ip(指令指针寄存器)
- sp（堆栈寄存器）

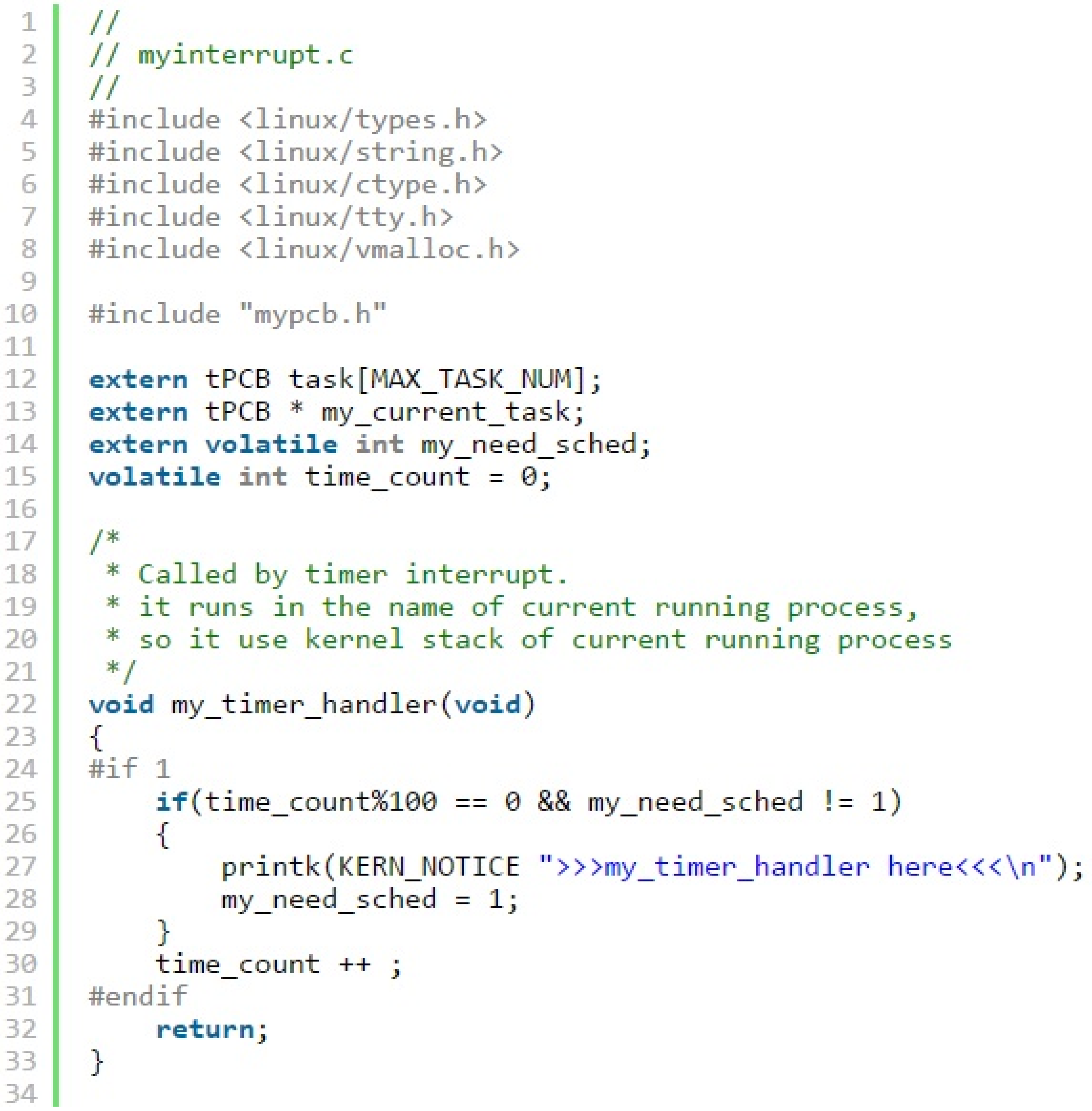


```c
//
// mymain.c
//
#include <linux/types.h>
#include <linux/string.h>
#include <linux/ctype.h>
#include <linux/tty.h>
#include <linux/vmalloc.h>

#include "mypcb.h"

tPCB task[MAX_TASK_NUM];
tPCB * my_current_task = NULL;
volatile int my_need_sched = 0;

void my_process(void);

void __init my_start_kernel(void)
{
    int pid = 0;
    int i;
    /* Initialize process 0*/
    task[pid].pid = pid;
    task[pid].state = 0;/* -1 unrunnable, 0 runnable, >0 stopped */
    task[pid].task_entry = task[pid].thread.ip = (unsigned long)my_process
    task[pid].thread.sp = (unsigned long)&task[pid].stack[KERNEL_STACK_SIZ
    task[pid].next = &task[pid];
    /*fork more process */
    for(i=1;i<MAX_TASK_NUM;i++)
    {
        memcpy(&task[i],&task[0],sizeof(tPCB));
        task[i].pid = i;
        task[i].state = -1;
        task[i].thread.sp = (unsigned long)&task[i].stack[KERNEL_STACK_SIZ
        task[i].next = task[i-1].next;
        task[i-1].next = &task[i];
    }
```

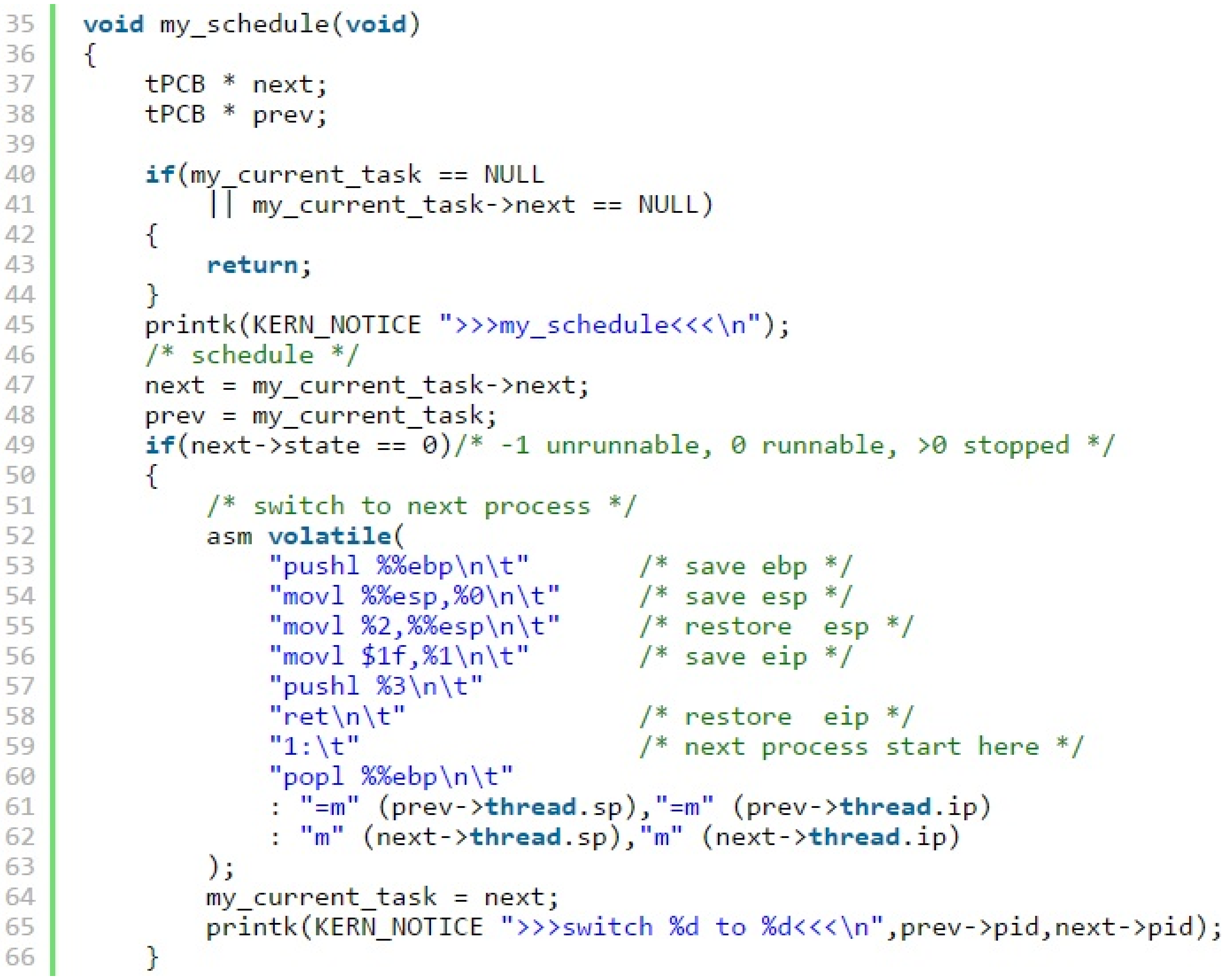


```c
    /* start process 0 by task[0] */
    pid = 0;
    my_current_task = &task[pid];
    asm volatile(
        "movl %1,%%esp\n\t"     /* set task[pid].thread.sp to esp */
        "pushl %1\n\t"          /* push ebp */
        "pushl %0\n\t"          /* push task[pid].thread.ip */
        "ret\n\t"               /* pop task[pid].thread.ip to eip */
        "popl %%ebp\n\t"
        :
        : "c" (task[pid].thread.ip),"d" (task[pid].thread.sp)   /* input 
    );
}
void my_process(void)
{
    int i = 0;
    while(1)
    {
        i++;
        if(i%10000000 == 0)
        {
            printk(KERN_NOTICE "this is process %d -\n",my_current_task->
            if(my_need_sched == 1)
            {
                my_need_sched = 0;
                my_schedule();
            }
            printk(KERN_NOTICE "this is process %d +\n",my_current_task->
        }
    }
}
```

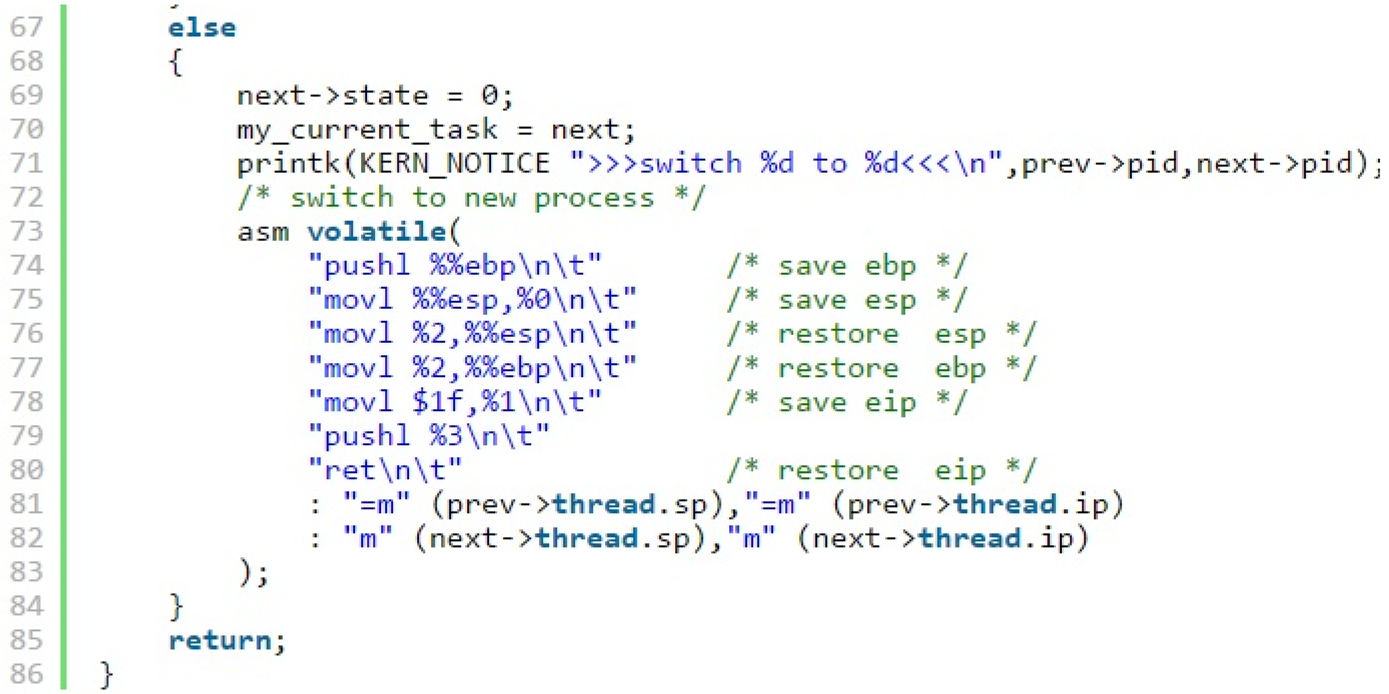
mymain.c



```c
//
// myinterrupt.c
//
#include <linux/types.h>
#include <linux/string.h>
#include <linux/ctype.h>
#include <linux/tty.h>
#include <linux/vmalloc.h>

#include "mypcb.h"

extern tPCB task[MAX_TASK_NUM];
extern tPCB * my_current_task;
extern volatile int my_need_sched;
volatile int time_count = 0;

/*
 * Called by timer interrupt.
 * it runs in the name of current running process,
 * so it use kernel stack of current running process
 */
void my_timer_handler(void)
{
#if 1
    if(time_count%100 == 0 && my_need_sched != 1)
    {
        printk(KERN_NOTICE ">>>my_timer_handler here<<<\n");
        my_need_sched = 1;
    }
    time_count ++ ;
#endif
    return;
}

```



```
void my_schedule(void)
{
    tPCB * next;
    tPCB * prev;

    if(my_current_task == NULL
        || my_current_task->next == NULL)
    {
        return;
    }
    printk(KERN_NOTICE ">>>my_schedule<<<\n");
    /* schedule */
    next = my_current_task->next;
    prev = my_current_task;
    if(next->state == 0)/* -1 unrunnable, 0 runnable, >0 stopped */
    {
        /* switch to next process */
        asm volatile(
            "pushl %%ebp\n\t"       /* save ebp */
            "movl %%esp,%0\n\t"     /* save esp */
            "movl %2,%%esp\n\t"     /* restore  esp */
            "movl $1f,%1\n\t"       /* save eip */
            "pushl %3\n\t"
            "ret\n\t"               /* restore  eip */
            "1:\t"                  /* next process start here */
            "popl %%ebp\n\t"
            : "=m" (prev->thread.sp),"=m" (prev->thread.ip)
            : "m" (next->thread.sp),"m" (next->thread.ip)
        );
        my_current_task = next;
        printk(KERN_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);
    }
```

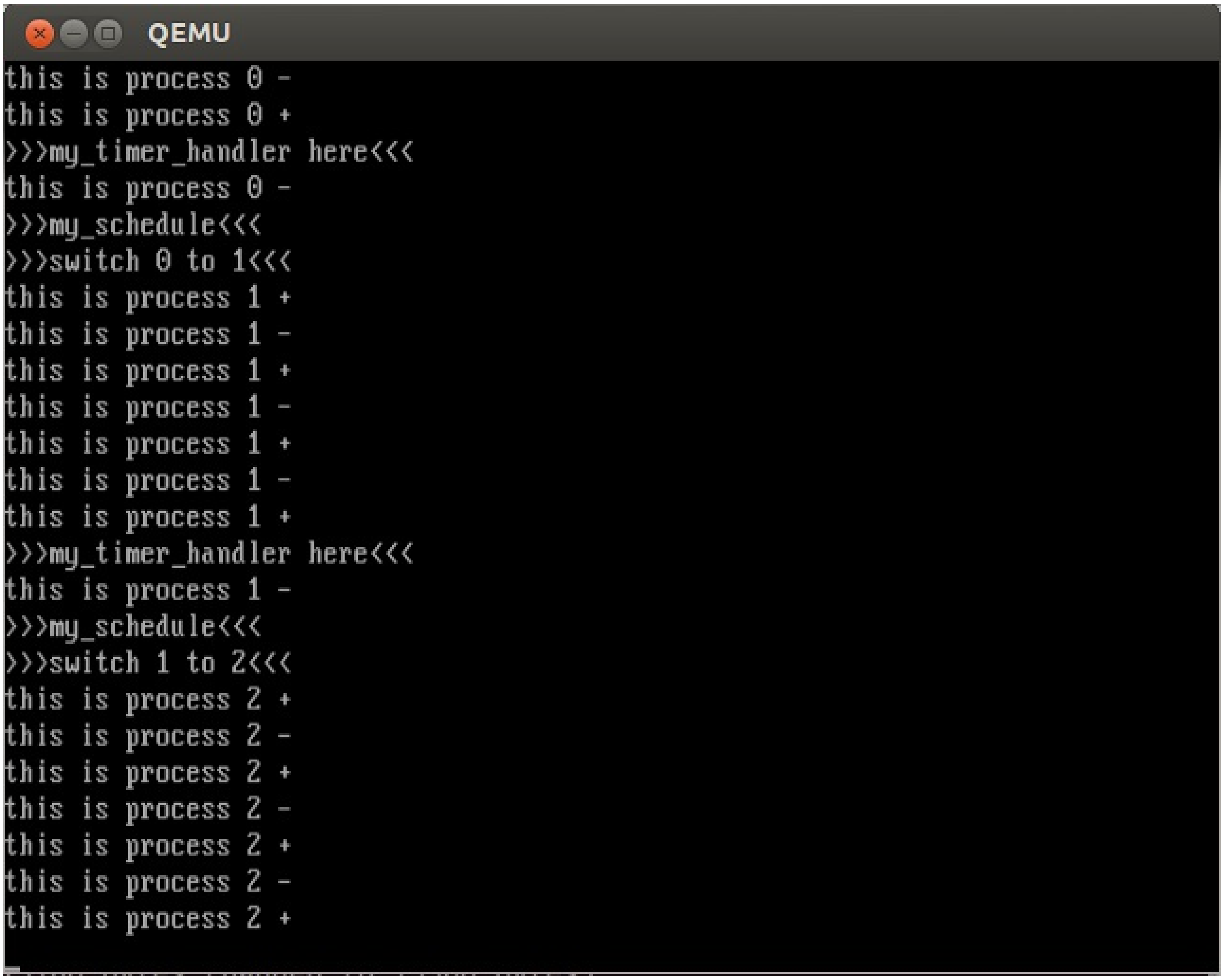


```
    else
    {
        next->state = 0;
        my_current_task = next;
        printk(KERN_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);
        /* switch to new process */
        asm volatile(
            "pushl %%ebp\n\t"       /* save ebp */
            "movl %%esp,%0\n\t"     /* save esp */
            "movl %2,%%esp\n\t"     /* restore  esp */
            "movl %2,%%ebp\n\t"     /* restore  ebp */
            "movl $1f,%1\n\t"       /* save eip */
            "pushl %3\n\t"
            "ret\n\t"               /* restore  eip */
            : "=m" (prev->thread.sp),"=m" (prev->thread.ip)
            : "m" (next->thread.sp),"m" (next->thread.ip)
        );
    }
    return;
}
```

myinterrupt.c

完成3个文件的代码编写后，在linux-3.9.4目录下重新编译内核，并使用qemu运行

- make
- qemu -kernel arch/x86/boot/bzImage



```
this is process 0 -
this is process 0 +
>>>my_timer_handler here<<<
this is process 0 -
>>>my_schedule<<<
>>>switch 0 to 1<<<
this is process 1 +
this is process 1 -
this is process 1 +
this is process 1 -
this is process 1 +
this is process 1 -
this is process 1 +
>>>my_timer_handler here<<<
this is process 1 -
>>>my_schedule<<<
>>>switch 1 to 2<<<
this is process 2 +
this is process 2 -
this is process 2 +
this is process 2 -
this is process 2 +
this is process 2 -
this is process 2 +
```

作业要求

按照配置教程完成实验

截取最后运行成功的图片提交与assignment_9一起提交到week_9。

命名方式：学号_PinYinXingming_linux_kernel.png