Project 1

秦啸涵　**521021910604**

前言：hello.c与simple.c的编译

笔者使用`ubuntu20.04`虚拟机，切换到对应目录下执行make，编译产生simple.ko，加载模块后使用`dmesg`查看内核日志缓冲区，成功看到模块已加载

```
[ 4947.940161] Loading Module
```

之后在`Makefile`中更改要编译的模块为hello.o，再次编译并加载模块，使用`cat /proc/hello`访问文件，成功看到缓冲区的输出

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ sudo insmod hello.ko
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ cat /proc/hello
Hello World
```

第一部分：jiffies内核模块设计

全局变量jiffies用于记录自系统启动后产生timer中断的总数。一秒钟产生时钟中断次数Hz，jiffies值增加Hz。新建文件jiffies.c，其内容与提供的hello.c基本一致，在头文件区增加一行`#include <linux/jiffies.h>`(但笔者在后续实践中发现似乎不需要include这个头文件也可以，可能在其他的内核模块中已定义)，将`PROC_NAME`改为`jiffies`，并在`proc_read`函数中输出当前jiffies:

```
ssize_t proc_read(struct file *file, char __user *usr_buf,
size_t count, loff_t *pos)
{
        int rv = 0;
        char buffer[BUFFER_SIZE];
        static int completed = 0;

        if (completed) {
                completed = 0;
                return 0;
        }

        completed = 1;
```

```
13
14          rv = sprintf(buffer, "The count of jiffies is %lu\n",
    jiffies);
15
16          // copies the contents of buffer to userspace usr_buf
17          copy_to_user(usr_buf, buffer, rv);
18
19          return rv;
20  }
```

执行make并加载模块，使用 `cat /proc/jiffies` 查看输出：

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ sudo insmod jiffies.ko
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ cat /proc/jiffies
The count of jiffies is 4296335403
```

成功打印出当前的jiffies.

第二部分：seconds模块设计

为获得内核模块加载后经过的秒数，我们需要知道内核加载时刻对应的 `jiffies0` 和文件读取时刻对应的 `jiffies1`，以及时钟频率 `HZ`，则最终结果为：

$$seconds = \frac{jiffies1 - jiffies0}{HZ}$$

具体而言，重用jiffies.c的代码，并进行如下改动：

- 定义 `begin` 为起始时刻 `jiffies`， `seconds` 为最终结果

```
9   unsigned long begin, seconds;
10  #define PROC_NAME "seconds"
```

- 在 `proc_init` 函数中(即模块加载时调用的函数)为 `begin` 赋值： `begin = jiffies;`
- 在 `proc_read` 函数中(即读取文件时调用的函数)计算 `seconds` 并输出

```
seconds = (jiffies - begin) / HZ;
rv = sprintf(buffer, "%lu seconds have passed since the module was loaded.\n", seconds);
```

执行make并加载模块，使用 `cat /proc/seconds` 查看输出：

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ sudo rmmod seconds.ko
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ sudo insmod seconds.ko
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ cat /proc/seconds
19 seconds have passed since the module was loaded.
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ cat /proc/seconds
22 seconds have passed since the module was loaded.
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ cat /proc/seconds
24 seconds have passed since the module was loaded.
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ cat /proc/seconds
25 seconds have passed since the module was loaded.
jianke@ubuntu:~/Desktop/final-src-osc10e/ch2$ cat /proc/seconds
26 seconds have passed since the module was loaded.
```

成功打印出经过的秒数

Bonus: copy_to_user()与memcpy()的差异

`copy_to_user` 和 `memcpy` 都是用于将数据从一个内存区域复制到另一个内存区域的函数，但它们的用途和实现方式有所不同。

`copy_to_user` 是Linux内核提供的函数之一，它用于将内核空间的数据复制到用户空间。这个函数的原型如下：

```
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long n);
```

其中，`to` 是指向用户空间的指针，`from` 是指向内核空间的指针，`n` 是要复制的字节数。`copy_to_user` 函数将尝试将 `from` 指向的数据复制到 `to` 指向的地址中，返回值为未能复制的字节数。

在执行 `copy_to_user` 函数时，内核会检查用户空间地址是否有效，是否有足够的空间存放要复制的数据，并且还会处理信号中断的情况，以确保数据的完整性和正确性。

相比之下，`memcpy` 是一个标准C库函数，用于将一段内存区域的数据复制到另一段内存区域。这个函数的原型如下：

```
void *memcpy(void *dest, const void *src, size_t n);
```

其中，`dest` 是指向目标内存区域的指针，`src` 是指向源内存区域的指针，`n` 是要复制的字节数。`memcpy` 函数会将 `src` 指向的数据复制到 `dest` 指向的地址中，并返回 `dest` 的指针。

与 `copy_to_user` 不同， `memcpy` 函数并不检查目标内存区域是否有效，也不会处理信号中断的情况。因此，如果要将内核空间的数据复制到用户空间，应该使用 `copy_to_user` 函数，以确保数据的正确性和安全性。