实验四报告要求

Linux 内核编程可用于扩展操作系统的功能，如 TCP/IP 协议栈中的拥塞控制算法可采用 Linux 内核编程的方式实现。在编译 Linux 内核时，需要安装必要的编译环境，具体命令如下：

$**sudo apt-get install build-essential linux-headers-$(uname -r)**

为了快速感受 Linux 内核编程的魅力，接下来给出一个学习编程语言时常用的 Hello World 程序，它在以内核模块的方式实现。具体的内核代码(**hello.c**)如下：

```
#include <linux/module.h>         /* Needed by all modules */
#include <linux/kernel.h>         /* Needed for KERN_INFO */
#include <linux/init.h>           /* Needed for the macros */

///< The license type -- this affects runtime behavior
MODULE_LICENSE("GPL");
///< The author -- visible when you use modinfo
MODULE_AUTHOR("Norbert");
///< The description -- see modinfo
MODULE_DESCRIPTION("A simple Hello world LKM!");
///< The version of the module
MODULE_VERSION("0.1");

static int __init hello_start(void)
{
    printk(KERN_INFO "Loading hello module...\n");
    printk(KERN_INFO "Hello world\n");
    return 0;
}

static void __exit hello_end(void)
{
    printk(KERN_INFO "Goodbye Mr.\n");
}

module_init(hello_start);
module_exit(hello_end);
```

为了编译源码方便，此处我们用 make 工具进行管理，需编写 **Makefile** 文件，具体 Makefile 内容如下：

```
obj-m = hello.o
all:
        make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

将上述两个文件放置在同一个目录下，然后执行：$**make**。如果正确编译，可以得到 **hello.ko** 文件，利用命令$**sudo insmod hello.ko** 将新模块插入内核中。如果想要移除插入的

模块，可执行命令：**$sudo rmmod hello.ko**。为了查看内核模块的输出，可以执行命令：**$dmesg**。

上述是一个内核版本的 Hello World 程序完整实现和使用，接下来，大家需要自己完成以下内容：

(1) 一个输出内核时间的内核模块；

(2) 一个发送速率稳定的拥塞控制模块；

实验提交内容：实验提交两个模块，并完成实验报告。

[**每位同学独立完成，遇到疑问可询问助教和教师**]

第 2 个模块的参考代码：

```
#include <linux/module.h>
#include <net/tcp.h>
#include <linux/random.h>
#include <linux/mm.h>
#include <linux/inet_diag.h>

#define CBR_RATE_MIN 1024u

struct cbr {
        /* rate control variables */
        s64 rate; /* current delivery rate */
};

static u32 cbr_get_rtt(struct tcp_sock *tp) //it's ok
{
        if (tp->srtt_us) {
                return max(tp->srtt_us >> 3, 1U);
        } else {
                return USEC_PER_MSEC;
        }
}

/* Initialize cwnd to support current pacing rate (more then 4 packets) */
static void cbr_set_cwnd(struct sock *sk) //it's ok
{
        struct tcp_sock *tp = tcp_sk(sk);
        u64 cwnd = sk->sk_pacing_rate;
        cwnd *= cbr_get_rtt(tcp_sk(sk));
        cwnd /= tp->mss_cache;
        cwnd /= USEC_PER_SEC;
        cwnd *= 2;
        cwnd = max(4ULL, cwnd);
        cwnd = min((u32)cwnd, tp->snd_cwnd_clamp); /* apply cap */
        tp->snd_cwnd = cwnd;
}

/* NetRate Main Function */
static void cbr_main_process(struct sock *sk, const struct rate_sample *rs)
{

}

static void cbr_init(struct sock *sk)
{
        struct cbr *cbrt = inet_csk_ca(sk);
        cbrt->rate = CBR_RATE_MIN*512; //512KBps or 4Mbps
        //cbrt->rate = CBR_RATE_MIN*6*1024;
        tcp_sk(sk)->snd_ssthresh = TCP_INFINITE_SSTHRESH;
        cmpxchg(&sk->sk_pacing_status, SK_PACING_NONE, SK_PACING_NEEDED);
        sk->sk_pacing_rate = cbrt->rate;
        cbr_set_cwnd(sk);
}

static void cbr_cong_avoid(struct sock *sk, u32 ack, u32 acked) //it's ok
{
}

static void cbr_pkts_acked(struct sock *sk, const struct ack_sample *acks) //it's ok
{
}

static void cbr_ack_event(struct sock *sk, u32 flags) //it's ok
{
}

static void cbr_cwnd_event(struct sock *sk, enum tcp_ca_event event) //it's ok
{
}

static void cbr_release(struct sock *sk) //it's ok
{
}

static u32 cbr_undo_cwnd(struct sock *sk) //it's ok
{
        return tcp_sk(sk)->snd_cwnd;
}
```

```
static u32 cbr_ssthresh(struct sock *sk) //it's ok
{
	return TCP_INFINITE_SSTHRESH; /* CBR does not use ssthresh */
}

static struct tcp_congestion_ops tcp_cbr_cong_ops __read_mostly = {
	.flags			= TCP_CONG_NON_RESTRICTED,
	.name			= "cbr",
	.owner			= THIS_MODULE,
	.init			= cbr_init,
	.cong_control	= cbr_main_process,
	//.set_state		= cbr_set_state,

	/* Keep the windows static */
	.undo_cwnd		= cbr_undo_cwnd,
	.release		= cbr_release,
	/* Slow start threshold will not exist */
	.ssthresh		= cbr_ssthresh,
	.cong_avoid		= cbr_cong_avoid,
	.pkts_acked		= cbr_pkts_acked,
	.in_ack_event	= cbr_ack_event,
	.cwnd_event	= cbr_cwnd_event,
};

/* Kernel module section */

static int __init cbr_register(void)
{
	BUILD_BUG_ON(sizeof(struct cbr) > ICSK_CA_PRIV_SIZE);
	printk(KERN_INFO "cbr init reg\n");
	return tcp_register_congestion_control(&tcp_cbr_cong_ops);
}

static void __exit cbr_unregister(void)
{
	tcp_unregister_congestion_control(&tcp_cbr_cong_ops);
}

module_init(cbr_register);
module_exit(cbr_unregister);

MODULE_AUTHOR("Xianliang Jiang <norbert.jiang@gmail.com>");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("TCP CBR");
```

```
ifneq ($(KERNELRELEASE),)

# kbuild part of makefile
obj-m := tcp_cbr.o

else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build
default:
	$(MAKE) -C $(KDIR) M=$$PWD
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
endif
```