实验五

题目5-1 内核模块编程：打印当前CPU负载

实验目的

1. 了解linux 的proc文件系统功能。
2. 掌握内核模块编程的基本能力。
3. 掌握在内核中读写文件数据的方法（数据结构和函数接口等）。

实验内容

使用内核模块编程的方法从系统中获取1min内当前cpu的负载，并将其打印出来。

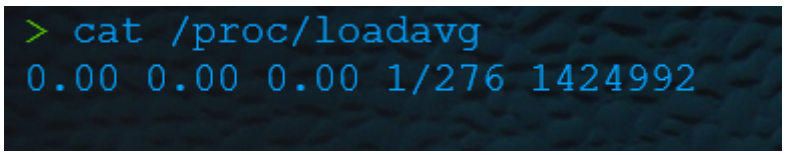
实验设计原理和步骤

原理方法

路径 /proc/下的系统文件loadavg中存有近期的cpu平均负载。

如图，前面三个数字分别是cpu在1分钟、5分钟、15分钟内的平均负载。

因此在代码中读取该文件内容从而获取平均负载即可。



代码步骤

代码的核心部分包括：

1. 将文件/proc/loadavg打开（get_loadavg函数完成）；
2. 读取文件最开头的数据，即1min内平均负载（get_loadavg函数完成）；
3. 将数据保存后关闭文件（get_loadavg函数完成）；
4. 通过printk函数打印（cpu_loadavg_init函数完成）；

   然后cpu_loadavg_init函数作为模块的init函数，在模块被加载时调用。

   详细代码见后文程序代码部分。

实验结果及分析

编译完成

```
lab2.c  Makefile
> make
make -C /usr/src/kernels/4.18.0-305.12.1.el8_4.x86_64 M=/root/osLab2 modules
make -C /usr/src/kernels/4.18.0-305.12.1.el8_4.x86_64 M=/root/osLab2 modules
make[1]: 进入目录“/usr/src/kernels/4.18.0-305.12.1.el8_4.x86_64”
  CC [M]  /root/osLab2/lab2.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/osLab2/lab2.mod.o
  LD [M]  /root/osLab2/lab2.ko
make[1]: 离开目录“/usr/src/kernels/4.18.0-305.12.1.el8_4.x86_64”
```

从加载模块到卸载模块

```
> insmod lab2.ko
> dmesg|tail -n 2
[1754571.824747] Start cpu_loadavg!
[1754571.825605] The cpu loadavg in one minute is: 0.08
> lsmod|grep lab2
libkmod: kmod_module_get_holders: could not open '/sys/module/lab1/holders': No such file or directory
lab2                   16384  0
> rmmod lab2
> dmesg|tail -n 3
[1754571.824747] Start cpu_loadavg!
[1754571.825605] The cpu loadavg in one minute is: 0.08
[1754639.177608] Exit cpu_loadavg!
```

单看dmesg输出

```
[1754571.824747] Start cpu_loadavg!
[1754571.825605] The cpu loadavg in one minute is: 0.08
[1754639.177608] Exit cpu_loadavg!
```

程序代码

模块源文件

lab2.c

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

MODULE_LICENSE("GPL");

char tmp_cpu_load[5] = {'\0'};
```

```c
static int get_loadavg(void)
{
    struct file *fp_cpu;
    loff_t pos = 0;
    char buf_cpu[10];
    fp_cpu = filp_open("/proc/loadavg", O_RDONLY, 0);
    if (IS_ERR(fp_cpu)){
        printk("Failed to open loadavg file!\n");
        return -1;
    }
    kernel_read(fp_cpu, buf_cpu, sizeof(buf_cpu), &pos);
    strncpy(tmp_cpu_load, buf_cpu, 4);
    filp_close(fp_cpu, NULL);
    return 0;
}

static int __init cpu_loadavg_init(void)
{
    printk("Start cpu_loadavg!\n");
    if(0 != get_loadavg()){
        printk("Failed to read loadarvg file!\n");
        return -1;
    }
    printk("The cpu loadavg in one minute is: %s\n", tmp_cpu_load);
    return 0;
}

static void __exit cpu_loadavg_exit(void)
{
    printk("Exit cpu_loadavg!\n");
}

module_init(cpu_loadavg_init);
module_exit(cpu_loadavg_exit);
```

Makefile文件

Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := lab2.o
else
    PWD := $(shell pwd)
    KVER ?=$(shell uname -r)
    KERNELDIR :=/usr/src/kernels/$(KVER)
default:
    @echo $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

题目5-2 使用cgroup限制程序使用的CPU核数

实验目的

1. 了解**cgroup 虚拟文件系统**的功能。
2. 掌握使用**cgroup 虚拟文件系统**进行管理的方法。

实验内容

创建临时文件系统格式（tmpfs）的cgroup虚拟文件系统，在其中挂载的cpuset管理子系统中限制CPU使用的核数。

然后编写程序并运行，对比普通运行核在指定cpuset管理子系统设置下运行时使用的CPU核数。

实验步骤

1. 使用mount命令创建tmpfs临时文件系统格式的cgroup虚拟文件系统并创建cpuset子系统文件夹。

```
root@ubuntu:~# mkdir cgroup
root@ubuntu:~# mount -t tmpfs tmpfs cgroup
root@ubuntu:~# ls
cgroup          Documents                           lab4          Public
cpuset.cpus~    Downloads                           lab4.tar.gz   Templates
cpuset.cpuz~    examples.desktop                    Music         Videos
Desktop         gcc-3.4-base_3.4.6-6ubuntu3_i386.deb  Pictures
root@ubuntu:~# cd cgroup
root@ubuntu:~/cgroup# mkdir cpuset
```

2. 进入cpuset文件夹挂载自己的cpuset子系统。

```
root@ubuntu:~/cgroup# mount -t cgroup -o cpuset cpuset cpuset
```

```
root@ubuntu:~/cgroup# cd cpuset
root@ubuntu:~/cgroup/cpuset# mkdir whx
root@ubuntu:~/cgroup/cpuset# ls
```

3. 设置限制的cpu核数。

   备注：这里应当注意，echo 0 >cpuset.mems不能省略。

   这里的cpu核数限制为最多只能使用核0、1和2

```
root@ubuntu:~/cgroup/cpuset/whx# echo 0-2 > cpuset.cpus
root@ubuntu:~/cgroup/cpuset/whx# echo 0 >cpuset.mems
root@ubuntu:~/cgroup/cpuset/whx# cat cpuset.cpus
0-2
root@ubuntu:~/cgroup/cpuset/whx# cat cpuset.mems
0
```

4. 编写程序运行，并查看其运行在哪些核上。

   运行：

```
root@ubuntu:~/Desktop# gcc ostest.c -o ostest
root@ubuntu:~/Desktop# ./ostest
^C
```

   查看：

```
root@ubuntu:~/cgroup/cpuset/whx# ps -a
  PID TTY          TIME CMD
 2074 pts/1    00:00:00 sudo
 2075 pts/1    00:00:00 bash
 3150 pts/17   00:00:00 sudo
 3151 pts/17   00:00:00 bash
 3978 pts/17   00:00:04 ostest
 3979 pts/1    00:00:00 ps
root@ubuntu:~/cgroup/cpuset/whx# taskset -p 3978
pid 3978's current affinity mask: f
root@ubuntu:~/cgroup/cpuset/whx# taskset -cp 3978
pid 3978's current affinity list: 0-3
```

5. 同样的程序运行在刚刚设置的cgroup上。

   运行：

```
root@ubuntu:~/Desktop# cgexec -g cpuset:whx ./ostest
```

   查看：

```
root@ubuntu:~/cgroup/cpuset/whx# ps -a
  PID TTY          TIME CMD
 2074 pts/1    00:00:00 sudo
 2075 pts/1    00:00:00 bash
 3150 pts/17   00:00:00 sudo
 3151 pts/17   00:00:00 bash
 3986 pts/17   00:00:55 ostest
 3993 pts/1    00:00:00 ps
root@ubuntu:~/cgroup/cpuset/whx# taskset -p 3986
pid 3986's current affinity mask: 7
root@ubuntu:~/cgroup/cpuset/whx# taskset -cp 3986
pid 3986's current affinity list: 0-2
root@ubuntu:~/cgroup/cpuset/whx#
```

备注：

需要注意的是，这里运行时不宜通过查看各cpu占有率的方式来判断死循环程序运行在哪些cpu核上，因为这个死循环程序并未设计为可以在多核上运行。

实验结果及分析

1. 查看本机内核：共有四个内核，说明不做限制时程序默认可以运行在所有核上。
2. 从上面的实验中我们通过`taskset`命令看到`cgroup`可以指定其运行在限定的内核上，例如这里的0、1、核2号内核。
3. 总结：实验证明，`cgroup`可以限制程序运行在哪些内核上。

程序代码

运行的程序（死循环）

ostest.c

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

int main(int argc, char *argv[])
{
    int i=0;
    while (1){i++;}
    printf("Over");
    exit(0);
}
```