华为实验——进程管理

实验介绍

通过在内核态创建进程，读取系统CPU负载，打印系统当前运行进程PID并且使用cgroup限制CPU核数等操作了解进程管理。

任务描述

1. 编写内核模块，创建一个内核线程，并在模块退出时杀死该线程。
2. 编写内核模块，实现读取系统一分钟内的CPU负载。
3. 编写内核模块，打印当前系统处于运行状态的进程的PID和名字。
4. 使用cgroup实现CPU核数的限制。

实验目的

1. 掌握正确编写满足功能的源文件，正确编译。
2. 掌握正常加载、卸载内核模块，且内核模块功能满足任务所述。
3. 李艾哦姐操作系统的进程管理

相关知识

1. linux相关指令：
   ls：（英文全拼：list files）命令用于显示指定工作目录下之内容（列出目前工作目录所含之文件及子目录)。
   cd：（英文全拼：change directory）命令用于切换当前工作目录。
   mkdir：（英文全拼：make directory）命令用于创建目录。
   insmod：（英文全拼：install module）命令用于载入模块。
   dmesg：dmesg命令用于显示开机信息。
   rmmod：（英文全拼：remove module）命令用于删除模块。
2. gcc相关指令：
   编译：g++ test.cpp -o test

实验任务

创建内核进程

1. 实验步骤

   步骤1：正确编写满足功能的源文件，包括.c源文件和Makefile文件。

   步骤2：编译源文件。

   步骤3：加载编译完成的内核模块，并查看加载结果。

   步骤4：卸载内核模块，并查看结果。

2. 代码实现

   2.1 .c源文件

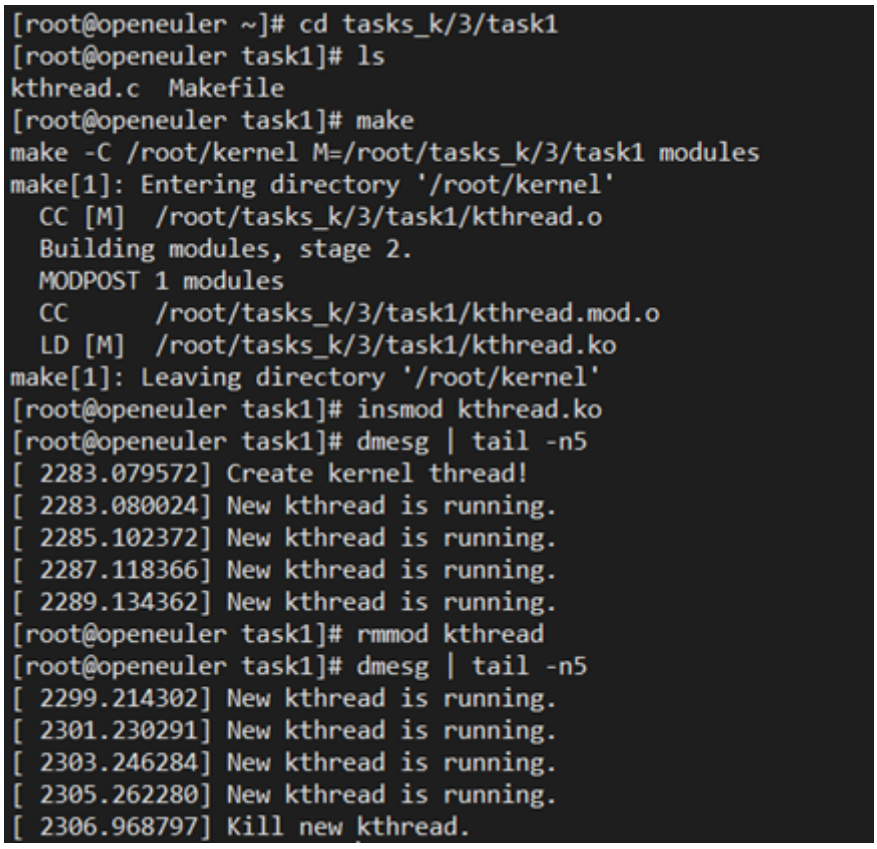
```c
#include <linux/kthread.h>
#include <linux/module.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");

#define BUF_SIZE 20

static struct task_struct *myThread = NULL;

static int print(void *data)
{
    while(!kthread_should_stop()){
        printk("New kthread is running.");
        msleep(2000);
    }
    return 0;
}

static int __init kthread_init(void)
{
    printk("Create kernel thread!\n");
    myThread = kthread_run(print, NULL, "new_kthread");
    return 0;
}

static void __exit kthread_exit(void)
{
    printk("Kill new kthread.\n");
    if(myThread)
```

```
        kthread_stop(myThread);
}

module_init(kthread_init);
module_exit(kthread_exit);
```

2.2 makefile文件

```
ifneq ($(KERNELRELEASE),)
    obj-m := kthread.o
else
    KERNELDIR ?= /root/kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 实验内容



4. 实验结果

   加载内核模块后程序运行,创建新线程并输出"Create kernel thread!"
   当线程正在运行的时候每2000ms输出"New kthread is running.
   在模块被卸载后,线程被停止,输出"New kthread is running."以及"Kill new kthread."显示之前创建的线程已经被结束 。

打印输出当前系统CPU负载情况

1. 实验步骤

   步骤1：正确编写满足功能的源文件，包括.c源文件和Makefile文件。
   步骤2：编译源文件。
   步骤3：加载编译完成的内核模块，并查看加载结果。
   步骤4：卸载内核模块，并查看结果。

2. 代码实现

   2.1 .c源文件

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

 MODULE_LICENSE("GPL");

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

 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);
```

2.2 makefile文件

```
ifneq ($(KERNELRELEASE),)
    obj-m := cpu_loadavg.o
else
    KERNELDIR ?= /root/kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 实验内容

```
[root@openeuler ~]# cd tasks_k/3/task2
[root@openeuler task2]# ls
cpu_loadavg.c  Makefile
[root@openeuler task2]# make
make -C /root/kernel M=/root/tasks_k/3/task2 modules
make[1]: Entering directory '/root/kernel'
  CC [M]  /root/tasks_k/3/task2/cpu_loadavg.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/tasks_k/3/task2/cpu_loadavg.mod.o
  LD [M]  /root/tasks_k/3/task2/cpu_loadavg.ko
make[1]: Leaving directory '/root/kernel'
[root@openeuler task2]# ls
cpu_loadavg.c   cpu_loadavg.mod.c  cpu_loadavg.o  modules.order
cpu_loadavg.ko  cpu_loadavg.mod.o  Makefile       Module.symvers
[root@openeuler task2]# insmod cpu_loadavg.ko
[root@openeuler task2]# dmesg | tail -n2
[ 2601.627156] Start cpu_loadavg!
[ 2601.627829] The cpu loadavg in one minute is: 0.00
[root@openeuler task2]# rmmod cpu_loadavg
[root@openeuler task2]# dmesg | tail -n3
[ 2601.627156] Start cpu_loadavg!
[ 2601.627829] The cpu loadavg in one minute is: 0.00
[ 2618.192618] Exit cpu_loadavg!
```

4. 实验结果

   平均负载(Load Average)是一段时间内系统的平均负载,本程序中取的时间为1min

   输出了"Start cpu_loadavg!"标识程序开始运行的时间

   用"The cpu loadavg in one minute is: 0.00"输出了CPU的负载情况

   以及用"Exit cpu_loadavg!"标识程序结束运行的时间

打印输出当前运行状态的进程PID和名字

1. 实验步骤

   步骤1：正确编写满足功能的源文件，包括.c源文件和Makefile文件。
   步骤2：编译源文件。
   步骤3：加载编译完成的内核模块，并查看加载结果。
   步骤4：卸载内核模块，并查看结果。

2. 代码实现

   2.1 .c源文件

```c
#include <linux/module.h>
#include <linux/sched/signal.h>
#include <linux/sched.h>

MODULE_LICENSE("GPL");

struct task_struct *p;

static int __init process_info_init(void)
```

```
{
    printk("Start process_info!\n");
    for_each_process(p){
        if(p->state == 0)
            printk("1)name:%s  2)pid:%d  3)state:%ld\n", p-
>comm, p->pid, p->state);
    }
    return 0;
}

static void __exit process_info_exit(void)
{
    printk("Exit process_info!\n");
}

module_init(process_info_init);
module_exit(process_info_exit);
```

2.2 makefile文件

```
ifneq ($(KERNELRELEASE),)
    obj-m := process_info.o
else
    KERNELDIR ?= /root/kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 实验内容

```
[root@openeuler task2]# cd ..
[root@openeuler 3]# cd task3
[root@openeuler task3]# ls
Makefile  process_info.c
[root@openeuler task3]# make
make -C /root/kernel M=/root/tasks_k/3/task3 modules
make[1]: Entering directory '/root/kernel'
  CC [M]  /root/tasks_k/3/task3/process_info.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/tasks_k/3/task3/process_info.mod.o
  LD [M]  /root/tasks_k/3/task3/process_info.ko
make[1]: Leaving directory '/root/kernel'
[root@openeuler task3]# ls
Makefile      Module.symvers  process_info.ko     process_info.mod.o
modules.order  process_info.c  process_info.mod.c  process_info.o
[root@openeuler task3]# insmod process_info.ko
[root@openeuler task3]# dmesg | tail -n3
[ 2717.160163] Start process_info!
[ 2717.160901] 1)name:kworker/2:3  2)pid:285  3)state:0
[ 2717.161581] 1)name:insmod  2)pid:5555  3)state:0
[root@openeuler task3]# rmmod process_info
[root@openeuler task3]# dmesg | tail -n4
[ 2717.160163] Start process_info!
[ 2717.160901] 1)name:kworker/2:3  2)pid:285  3)state:0
[ 2717.161581] 1)name:insmod  2)pid:5555  3)state:0
[ 2730.768628] Exit process_info!
```

4. 实验结果

   输出了当前处于运行状态的进程的pid和名字,其中,pid是 Linux 中在其命名空间中唯一标识进程而分配给它的一个号码，称做进程ID号，简称PID。在使用 fork 或 clone 系统调用时产生的进程均会由内核分配一个新的唯一的PID值。

使用cgroup实现限制CPU核数

1. 实验步骤

   步骤1：安装libcgroup：dnf install libcgroup -y。

   步骤2：挂载tmpfs格式的cgroup文件夹。

```
# mkdir /cgroup
# mount -t tmpfs tmpfs /cgroup
# cd /cgroup
```

   步骤3：挂载cpuset管理子系统。

```
# mkdir cpuset
# mount -t cgroup -o cpuset cpuset /cgroup/cpuset    #挂载
cpuset子系统
# cd cpuset
# mkdir mycpuset     #创建一个控制组，删除用 rmdir 命令
# cd mycpuset
```

步骤4：设置cpu核数。

```
# echo 0 > cpuset.mems                    #设置0号内存结点。mems默认为空，因此需要填入值。
# echo 0-2 > cpuset.cpus                  #这里的0-2指的是使用cpu的0、1、2三个核。实现了只是用这三个核。
# cat cpuset.mems
0
# cat cpuset.cpus
0-2
```

步骤5：简单的死循环C源文件while_long.c。

步骤6：测试验证。

2. 代码实现

2.1 .c源文件

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

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

3. 实验内容

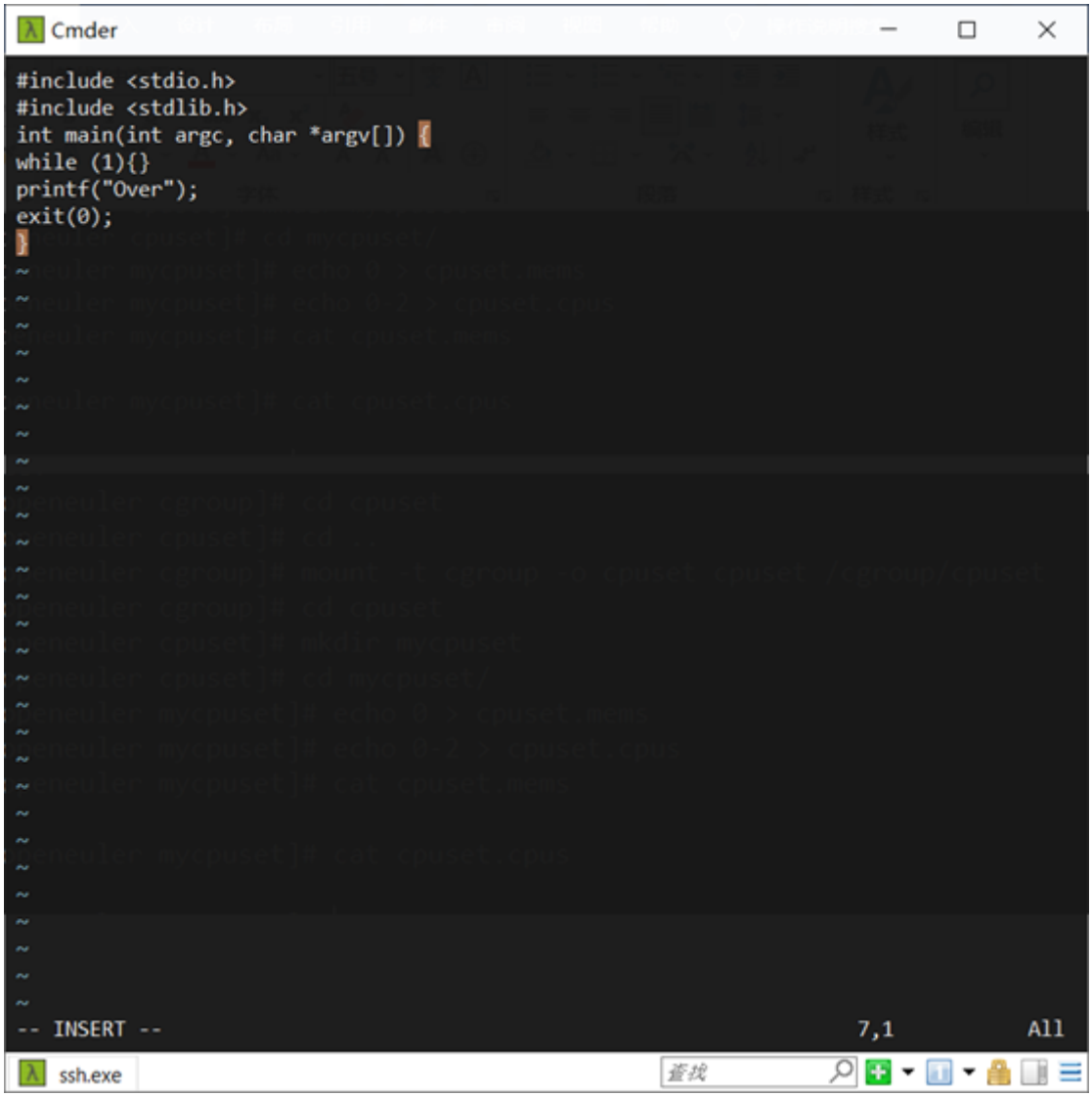
3.1 安装libcgroup

```
[root@openeuler ~]# dnf install libcgroup -y
Last metadata expiration check: 2:30:53 ago on Wed 19 May 2021 03:54:27 PM CST.
Dependencies resolved.
================================================================================
 Package              Architecture       Version                  Repository     Size
================================================================================
Installing:
 libcgroup            aarch64            0.41-23.oe1              OS             94 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 94 k
Installed size: 953 k
Downloading Packages:
libcgroup-0.41-23.oe1.aarch64.rpm                  2.7 MB/s |  94 kB     00:00
--------------------------------------------------------------------------------
Total                                              2.6 MB/s |  94 kB     00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                        1/1
  Running scriptlet: libcgroup-0.41-23.oe1.aarch64                          1/1
  Installing       : libcgroup-0.41-23.oe1.aarch64                          1/1
  Running scriptlet: libcgroup-0.41-23.oe1.aarch64                          1/1
  Verifying        : libcgroup-0.41-23.oe1.aarch64                          1/1

Installed:
  libcgroup-0.41-23.oe1.aarch64

Complete!
```

3.2 挂载tmpfs格式的cgroup文件夹及设置CPU核数

```
[root@openeuler ~]# mkdir /cgroup
mkdir: cannot create directory ‘/cgroup’: File exists
[root@openeuler ~]# mount -t tmpfs tmpfs /cgroup
[root@openeuler ~]# cd /cgroup
[root@openeuler cgroup]# mkdir cpuset
[root@openeuler cgroup]# mount -t cgroup -o cpuset cpuset /cgroup/cpuse
mount: /cgroup/cpuse: mount point does not exist.
[root@openeuler cgroup]# ls
cpuset
[root@openeuler cgroup]# dnf install libcgroup -y
Last metadata expiration check: 1:27:15 ago on Wed 19 May 2021 07:48:25 PM CST.
Package libcgroup-0.41-23.oe1.aarch64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[root@openeuler cgroup]# cd cpuset
[root@openeuler cpuset]# cd ..
[root@openeuler cgroup]# mount -t cgroup -o cpuset cpuset /cgroup/cpuset
[root@openeuler cgroup]# cd cpuset
[root@openeuler cpuset]# mkdir mycpuset
[root@openeuler cpuset]# cd mycpuset/
[root@openeuler mycpuset]# echo 0 > cpuset.mems
[root@openeuler mycpuset]# echo 0-2 > cpuset.cpus
[root@openeuler mycpuset]# cat cpuset.mems
0
[root@openeuler mycpuset]# cat cpuset.cpus
0-2
```

3.3 建立简单的死循环C源文件while_long.c




```
[root@openeuler ~]# ls
boot.origin.tgz  cpuset  kernel  kernel-4.19.zip  tasks_k  uname_r.log  while_long
[root@openeuler ~]#
```

```
[root@openeuler ~]# ls
boot.origin.tgz  cpuset  kernel  kernel-4.19.zip  tasks_k  uname_r.log  while_long
[root@openeuler ~]# cat while_long
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
while (1){}
printf("Over");
exit(0);
}
```

```
[root@openeuler ~]# mv while_long while_long.c
```

3.4 编译文件

```
[root@openeuler ~]# gcc while_long.c -o while_long
gcc: error: while_long.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[root@openeuler ~]# gcc while_long -o while_long.c
while_long: file not recognized: file format not recognized
collect2: error: ld returned 1 exit status
[root@openeuler ~]# mv while_long while_long.c
[root@openeuler ~]# gcc while_long -o while_long.c
gcc: error: while_long: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[root@openeuler ~]# gcc while_long.c -o while_long
[root@openeuler ~]# ls
boot.origin.tgz  kernel           tasks_k      while_long
cpuset           kernel-4.19.zip  uname_r.log  while_long.c
```

3.5 运行该文件并查看while_long程序的PID

```
top - 21:51:18 up  4:13,  3 users,  load average: 0.49, 0.13, 0.04
Tasks: 127 total,   2 running, 125 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25.0 us,  0.1 sy,  0.0 ni, 74.8 id,  0.0 wa,  0.1 hi,  0.0 si,  0.0 st
MiB Mem :   6811.7 total,   5860.3 free,    354.7 used,    596.7 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   6110.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   6757 root      20   0    2368    960    448 R 100.0   0.0   0:42.48 while_long
     10 root      20   0       0      0      0 I   0.3   0.0   0:00.24 rcu_sched
      1 root      20   0  174208  16832   8832 S   0.0   0.2   0:02.31 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-+
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
      9 root      20   0       0      0      0 S   0.0   0.0   0:00.00 ksoftirqd/0
     11 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_bh
     12 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 migration/0
     13 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
     15 root      rt   0       0      0      0 S   0.0   0.0   0:00.01 migration/1
     16 root      20   0       0      0      0 S   0.0   0.0   0:00.00 ksoftirqd/1
     18 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/1:0H-+
     19 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/2
     20 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 migration/2
     21 root      20   0       0      0      0 S   0.0   0.0   0:00.00 ksoftirqd/2
     23 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/2:0H-+
     24 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/3
     25 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 migration/3
     26 root      20   0       0      0      0 S   0.0   0.0   0:00.00 ksoftirqd/3
     28 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/3:0H-+
     29 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kdevtmpfs
     30 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
     31 root      20   0       0      0      0 S   0.0   0.0   0:00.02 kauditd
     33 root      20   0       0      0      0 S   0.0   0.0   0:00.00 khungtaskd
     34 root      20   0       0      0      0 S   0.0   0.0   0:00.00 oom_reaper
```

3.6 测试限制CPU核数

```
[root@openeuler ~]# taskset -p 6757
pid 6757's current affinity mask: 7
[root@openeuler ~]#  taskset -pc 6757
pid 6757's current affinity list: 0-2
```

4. 实验结果

控制群组（control group）是linux kernel的一项功能：在一个系统中运行的层级制进程组，可对其进行资源分配。

tmpfs即临时文件系统，是一种基于内存的文件系统，也称之为虚拟内存文档系统。它不同于传统的用块设备形式来实现的ramdisk，也不同于针对物理内存的ramfs。tmpfs能够使用物理内存，也能够使用交换分区。

我们设置只能使用0、1、2三个核并设置0号内存结点进行运行。

最后测试时使用了taskset命令，即依据线程PID（TID）查询或设置线程的CPU亲和性（即与哪个CPU核心绑定）。最终显示的结果为7以及0-2，所以测试限制CPU核数成功。

实验小结

在进行本次实验的过程中，因为对linux系统的不熟悉，有很多地方会发生一些错误，比如vim的使用不到位、对gcc的编译理解有误等。虽然刚开始很多都改不出来，但通过查阅资料将这些问题一一解决了，逐步加深了对linux系统以及linux内核的理解。

在做实验的过程中，我仔细阅读了相关的实验资料，对如何编写内核程序有了基本的掌握，并且掌握了如何编写makefile文件。同时，根据实验以及其结果对linux内核的运行过程和进程管理有了更深一步的了解，同时掌握了有关命令行控制linux的相关操作以及对内核的一些基本操作，在进行控制群组的实验时，我还对内核源码的官方文档进行了阅读，以对其功能有清晰的认识。

本次实验让我收获良多，不仅学习了操作系统进程调度相关的知识，同时也对linux系统有了更加深刻的了解，对命令行的使用有了一定的掌握。