

进程管理实验报告

课程：操作系统

指导老师：张惠娟

姓名：李航宇

学号：1951593

说明：

本次实验报告为进程管理实验报告，但需要提前搭建好实验环境。实验一为鲲鹏云ECS的构建及内核编译，其中介绍了实验环境搭建的步骤与内核模块的基本使用例子。因此本次实验报告也包括了部分实验一的内容。此外，实验参考文档中的任务描述、实验目的、相关知识介绍等在实验报告中不再重复列出。

进程管理实验报告

1.实验一 鲲鹏云ECS的构建及内核编译

1.1构建云实验环境

1.2 内核模块编程实验

2.实验三 进程管理

2.1 创建内核线程

2.1.1 实验步骤

2.1.2 分析与小结

2.2 打印输出当前系统CPU负载情况

2.2.1 实验过程

2.2.2 分析与小结

2.3 打印输出当前处于运行状态的进程的PID和名字

2.3.1 实验步骤

2.3.2 分析与小结

2.4 使用cgroup实现限制CPU核数

2.4.1 实验步骤

2.4.2 分析与小结

3 总结

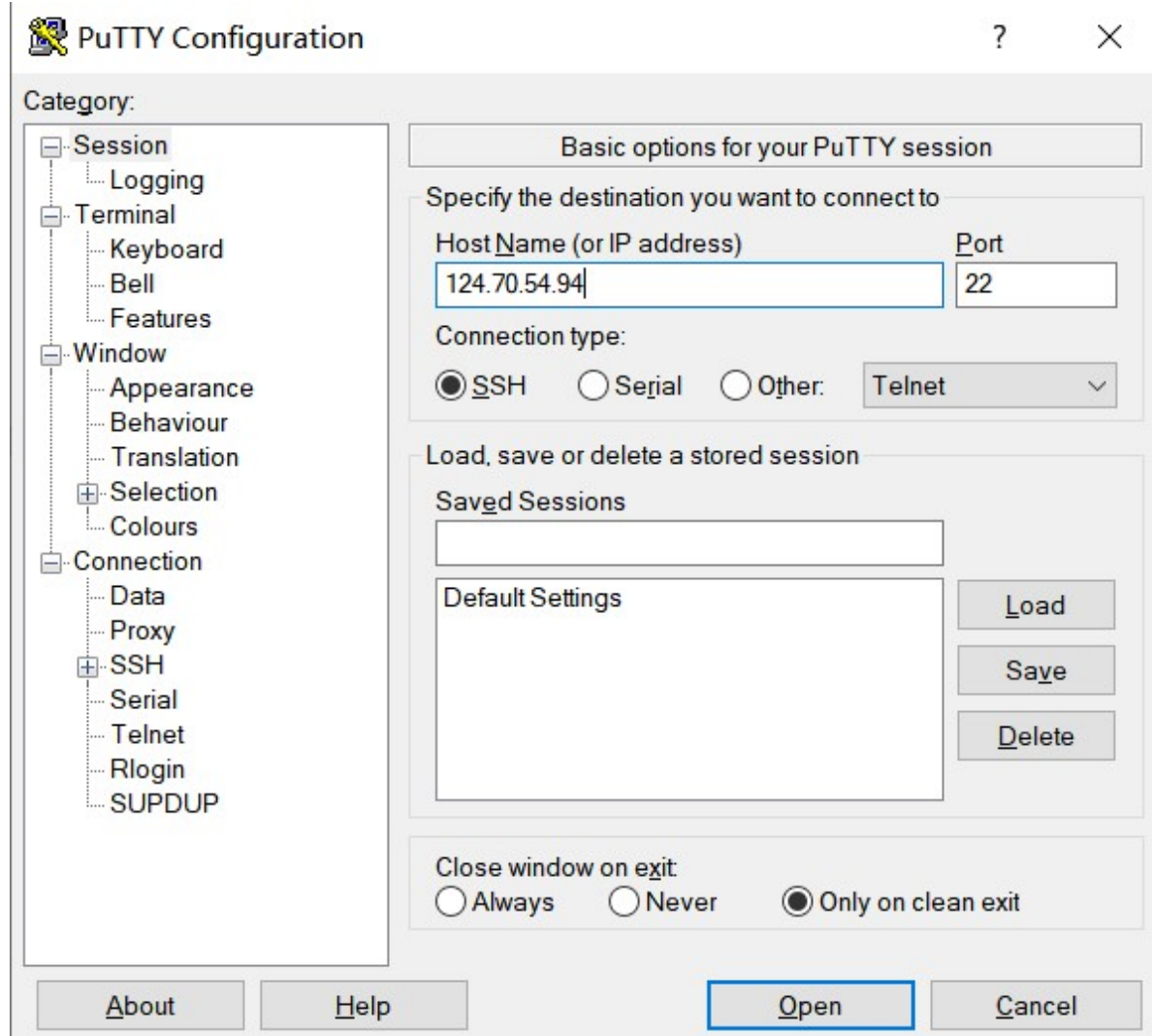
参考文献

1.实验一 鲲鹏云ECS的构建及内核编译

1.1构建云实验环境

实验一的步骤包括配置与购买云服务器，通过SSH登录系统，与进行实验任务。配置购买云服务器的部分完全参考实验文档，不再赘述。

我用的是putty来连接服务器，如下图所示，选择连接类型为ssh,输入云服务器的公网IP（云服务器还有一个私网IP，不能用其连接），点击Open进行连接即可，进入后，以root用户登录，并输入登录密码（在配置云服务器时已设置过）。



按照实验文档，连接和云服务器后，可以修改hostname为openEuler（创建云服务器时将其命名openEuler，系统默认hostname为openeuler）。这时可输入命令vi /etc/hostname用vi进行修改（此时是在root目录下进行这个命令）。刚启动vi时，进图的是命令模式，需要先按i键切换到输入模式，进行修改名称。再按ESC键回到命令模式，输入英文冒号进入底线命令模式，再输入wq进行保存文件并退出程序。

此时再输入cat /etc/hostname命令可查看修改后的主机名。修改名称后需要用reboot命令进行重启系统并重新登录。

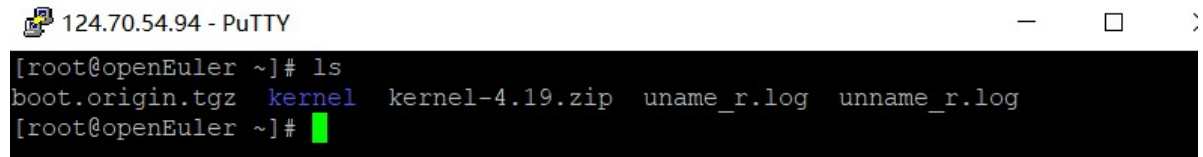
注：执行reboot命令后putty会显示断开连接，需要重新登录。


```
[root@openEuler ~]# uname -r
4.19.190
[root@openEuler ~]# _
```

编译安装好新的 openEuler 内核后，再执行示例源文件。

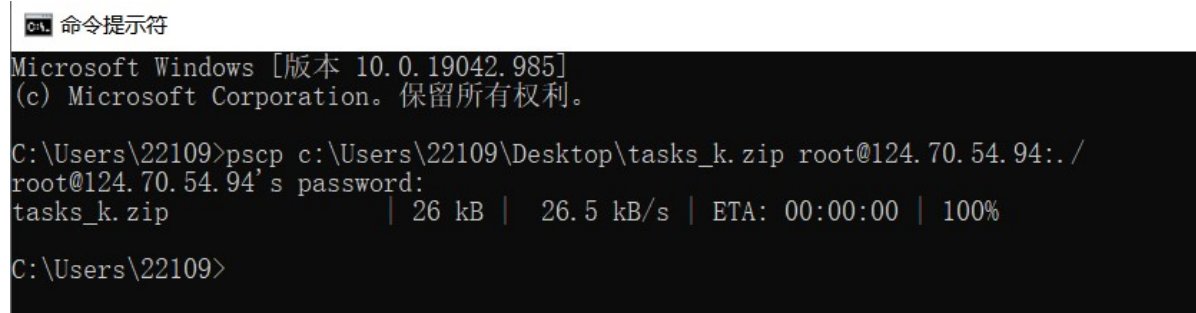
首先需要将示例源文件的压缩包 tasks_k.zip 传至 root 目录下。这里先将 tasks_k.zip 下载至本地，再通过 putty 自带的 PSCP 文件传输工具完成上传。

下图为未上传压缩包时的 root 目录内容。



```
124.70.54.94 - PuTTY
[root@openEuler ~]# ls
boot.origin.tgz  kernel  kernel-4.19.zip  uname_r.log  unname_r.log
[root@openEuler ~]#
```

下图为在 cmd 窗口中将本地的 tasks_k.zip 文件上传至服务器的命令。本地的 tasks_k.zip 位于桌面，上传至服务器的 ./ 目录，也就是 root 目录下。

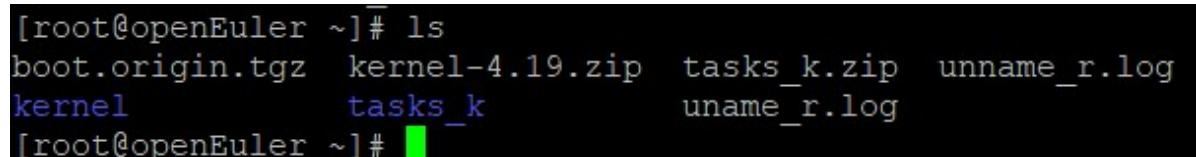


```
命令提示符
Microsoft Windows [版本 10.0.19042.985]
(c) Microsoft Corporation。保留所有权利。

C:\Users\22109>pscp c:\Users\22109\Desktop\tasks_k.zip root@124.70.54.94:./
root@124.70.54.94's password:
tasks_k.zip          | 26 kB | 26.5 kB/s | ETA: 00:00:00 | 100%

C:\Users\22109>
```

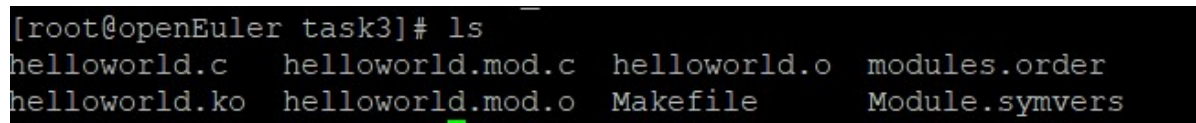
下图为上传 tasks_k.zip 文件后，通过 unzip 命令解压后 root 目录中的文件。可以看出，多了 task_k.zip 文件与 task_k 文件夹。



```
[root@openEuler ~]# ls
boot.origin.tgz  kernel-4.19.zip  tasks_k.zip  unname_r.log
kernel          tasks_k          uname_r.log
[root@openEuler ~]#
```

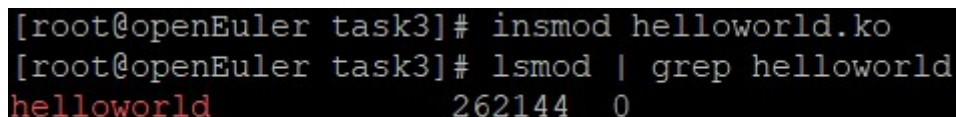
再进入 tasks_k/1/task3 目录。此当前目录中有 helloworld.c 源文件与 Makefile 文件。

执行 make 命令编译源文件。此时当前目录中有如下文件



```
[root@openEuler task3]# ls
helloworld.c  helloworld.mod.c  helloworld.o  modules.order
helloworld.ko  helloworld.mod.o  Makefile      Module.symvers
```

再用 insmod helloworld.ko 加载编译完成的内核模块，并查看加载结果，如下图所示。



```
[root@openEuler task3]# insmod helloworld.ko
[root@openEuler task3]# lsmod | grep helloworld
helloworld      262144  0
```

再按照文档，卸载内核模块，并查看结果。

注：dmseg | tail -n5 输出了最后5条日志，实际上有用的只是后面的3条，因为前面有两条报错信息（按文档提示可忽略）。因此至少需要输出最后5条日志才可看到 hello exit。

```
[root@openEuler task3]# rmmod helloworld
[root@openEuler task3]# dmesg | tail -n5
[ 170.171242] helloworld: loading out-of-tree module taints kernel.
[ 170.171763] helloworld: module verification failed: signature and/or required
key missing - tainting kernel
[ 170.172858] hello_init
[ 170.173026] Hello, world!
[ 387.106339] hello exit
```

注：每次将云服务器关机后，重新开机后第一次进入系统时默认会选择老的内核版本，这会导致后续加载编译完成的内核模块时出现如下报错。因此每次将云服务器关机后，重新开机后应先重启并切换到正确的内核版本。下面两张图分别为重新开机后系统默认选择的内核版本与报错提示。

```
[root@openEuler ~]# uname -r
4.19.90-2003.4.0.0036.oe1.aarch64
[root@openEuler ~]# _
```

```
[root@openEuler ~]# cd tasks_k/1/task3
[root@openEuler task3]# insmod helloworld.ko
insmod: ERROR: could not insert module helloworld.ko: Invalid module format
[root@openEuler task3]#
```

2.实验三 进程管理

2.1 创建内核线程

2.1.1 实验步骤

首先进入到 `tasks_k/3/task1` 目录，用 `ls` 命令查看当前目录内容。

```
[root@openEuler task1]# ls
kthread.c  Makefile
```

执行 `make` 命令编译源文件

```
[root@openEuler task1]# make
make -C /root/kernel M=/root/tasks_k/3/task1 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/3/task1/kthread.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/3/task1/kthread.mod.o
LD [M] /root/tasks_k/3/task1/kthread.ko
make[1]: Leaving directory '/root/kernel'
```

通过 `insmod kthread.ko` 加载编译完成的内核模块，用过 `dmesg | tail -n5` 查看最近五条日志结果。可以看到最近日志都是提示新进程正在运行，与实验帮助文档中不一致，下面的分析与总结中会提到这一点。

```
[root@openEuler task1]# dmesg | tail -n5
[ 1569.120324] New kthread is running.
[ 1571.136304] New kthread is running.
[ 1573.152289] New kthread is running.
[ 1575.168264] New kthread is running.
[ 1577.184242] New kthread is running.
```


通过 `rmmod kthread` 卸载内核模块,用 `dmesg | tail -n5` 查看最近五条日志结果。可以看到最后一条记录了线程被杀死。

```
[root@openEuler task1]# dmesg | tail -n5
[ 1653.791475] New kthread is running.
[ 1655.807463] New kthread is running.
[ 1657.823435] New kthread is running.
[ 1659.839419] New kthread is running.
[ 1660.312084] Kill new kthread.
```

2.1.2 分析与小结

```
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);
}
```

`kthread.c` 的主要代码如下。可以看出在创建线程与结束线程时都会打印相应的提示。此外,在线程被创建后, `print` 函数每隔2秒 (2000ms) 就会打印一条信息。

但在实验中,执行 `dmesg | tail -n5` 时打印出的信息没有创建信息的提示,与实验帮助文档中不一致,这是因为执行命令时可能已超过 $5 \times 2 = 10$ 秒,最新的五条信息都是提示新线程正在运行的信息,故没有显示创建线程的提示。退出时,则正常显示了进程被杀死的信息。

2.2 打印输出当前系统CPU负载情况

2.2.1 实验过程

首先切换到 `tasks_k/3/task2` 目录,并查看当前目录内容。

```
124.70.54.94 - PuTTY
[root@openEuler ~]# cd tasks_k/3/task2
[root@openEuler task2]# ls
cpu_loadavg.c  Makefile
```

再执行 `make` 命令编译源文件并查看当前目录内容。

```
[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]#
```

加载编译完成内核模块并查看加载结果，可以看到最近一分钟的平均CPU负载信息被打印出。

```
[root@openEuler task2]# insmod cpu_loadavg.ko
[root@openEuler task2]# dmesg | tail -n2
[ 3892.899026] Start cpu_loadavg!
[ 3892.899439] The cpu loadavg in one minute is: 0.00
```

卸载内核模块并查看结果。

```
[root@openEuler task2]# rmmod cpu_loadavg
[root@openEuler task2]# dmesg | tail -n3
[ 3892.899026] Start cpu_loadavg!
[ 3892.899439] The cpu loadavg in one minute is: 0.00
[ 3945.635291] Exit cpu_loadavg!
```

2.2.2 分析与小结

```
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 loadavg 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");
}
```

上面为 `cpu_loadavg.c` 的核心代码。创建线程与结束线程时都会打印相应的提示。可以看到，CPU的负载情况是通过读取 `/proc/loadavg` 文件进行的。由实验文档介绍，`/proc` 文件系统是Linux中的特殊文件系统，其中文件可以用于访问内核状态等信息。`/proc` 目录下的文件是虚拟的，但可以使用任何文件编辑器查看。

程序中，在 `get_loadavg` 函数内将 `/proc/loadavg` 先读取到了一个file结构体中，再读取到了字符数组 `buf_cpu` 中，最后将 `buf_cpu` 中数据复制到 `tmp_cpu_load` 中，并在 `cpu_loadavg_init` 函数中打印出进程数、PID与对应的CPU负载。

在Linux内核中进行文件读写操作时用到的相关函数及其主要功能如下图所示。

函数名称	功能说明
<code>filp_open()</code>	在kernel中打开指定文件。
<code>filp_close()</code>	kernel中文件的读操作。
<code>kernel_read()</code>	kernel中文件的写操作。
<code>kernel_write()</code>	关闭指定文件。

程序中，由于只在开始时打印了CPU负载情况，故提示CPU负载的信息只会被打印一次，这与实验中查看的日志结果一致。

2.3 打印输出当前处于运行状态的进程的PID和名字

2.3.1 实验步骤

首先切换到 `tasks_k/3/task3` 目录下，该目录内容如下

```
[root@openEuler task3]# ls
Makefile  process_info.c
```

编译源文件并查看编译后当前目录内容

```
[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
```

加载已编译好的内核模块并查看加载结果


```
[root@openEuler task3]# dmesg | tail -n3
[ 4355.764950] Start process_info!
[ 4355.765421] 1)name:kworker/1:3 2)pid:195 3)state:0
[ 4355.765919] 1)name:insmod 2)pid:4461 3)state:0
```

卸载内核模块并查看结果

```
[root@openEuler task3]# rmmod process_info
[root@openEuler task3]# dmesg | tail -n4
[ 4355.764950] Start process_info!
[ 4355.765421] 1)name:kworker/1:3 2)pid:195 3)state:0
[ 4355.765919] 1)name:insmod 2)pid:4461 3)state:0
[ 4411.231808] Exit process_info!
```

2.3.2 分析与小结

```
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");
}
```

上面是 `process_info.c` 的核心代码。`task_struct` 是Linux中的进程描述符结构体,包含了一个进程所需的所有信息,所以 `task_struct` 就是Linux系统中的PCB。在 `process_info_init` 函数中,通过 `for_each_process(p)` 访问量整个任务队列中的进程,并打印符合条件(处于运行状态)的进程的名称, `PID` 与状态。在 `sched.h` 文件中有如下的宏定义:

```
#define TASK_RUNNING      0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE       3
#define TASK_STOPPED      4
```

因此,在程序中打印state为0,即 `TASK_RUNNING` 的进程信息,也就是处于运行状态的进程。打印结果中处于 `TASK_RUNNING` 状态的进程只有 `kworker/1:3` 与 `insmod`,其中, `kworker` 是内核工作线程的占位符进程; `insmod` 是加载内核模块的进程。

2.4 使用 cgroup 实现限制CPU核数

2.4.1 实验步骤

首先安装 `libcgroup`。

注：安装时未截图，下图为已安装后再次执行安装命令，提示已安装。

```
[root@openEuler task3]# dnf install libcgroup -y
Last metadata expiration check: 2:40:02 ago on Tue 18 May 2021 10:26:41 AM CST.
Package libcgroup-0.41-23.oe1.aarch64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
```

挂载 `tmpfs` 格式的 `cgroup` 文件夹

```
[root@openEuler /]# mkdir /cgroup
[root@openEuler /]# mount -t tmpfs tmpfs /cgroup
[root@openEuler /]# cd /cgroup
```

挂载 `cpuset` 管理子系统

```
[root@openEuler cgroup]# mkdir cpuset
[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]#
```

设置CPU核数

```
[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
```

进入 `tasks_k/3/task` 文件夹，通过 `ls` 命令可以看到当前目录下只有一个 `.c` 文件，因此不能通过 `make` 命令编译。

```
[root@openEuler task4]# ls
while_long.c
```

执行下图两行命令，前一行使用 `gcc` 编译 `.c` 文件，后一行指定在 `cpuset` 子系统的 `mycpuset` 控制组中运行

```
[root@openEuler task4]# gcc while_long.c -o while_long
[root@openEuler task4]# cgexec -g cpuset:mycpuset ./while_long
```

打开一个新的终端，执行 `top` 命令

```
124.70.54.94 - PuTTY
[root@openEuler ~]# top
top - 15:36:05 up 4:04, 3 users, load average: 0.90, 0.37, 0.14
Tasks: 127 total, 2 running, 125 sleeping, 0 stopped, 0 zombie
%Cpu(s): 25.0 us, 0.0 sy, 0.0 ni, 74.9 id, 0.0 wa, 0.1 hi, 0.0 si, 0.0 st
MiB Mem : 6811.7 total, 5869.2 free, 363.4 used, 579.1 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used. 6103.2 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 5597 root        20   0    2368    896    448  R 100.0   0.0    2:14.79 while_1+
   1 root        20   0 107584 16448   8832  S   0.0   0.2    0:02.18 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+
   6 root         0 -20      0      0      0  I   0.0   0.0    0:00.00 kworker+
   8 root         0 -20      0      0      0  I   0.0   0.0    0:00.00 mm_perc+
   9 root        20   0      0      0      0  S   0.0   0.0    0:00.00 ksoftir+
  10 root        20   0      0      0      0  I   0.0   0.0    0:00.24 rcu_sch+
  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    0:00.01 migrati+
  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    0:00.00 migrati+
  16 root        20   0      0      0      0  S   0.0   0.0    0:00.00 ksoftir+
  18 root         0 -20      0      0      0  I   0.0   0.0    0:00.00 kworker+
```

最后进行测试验证

```
[root@openEuler ~]# taskset -p 5597
pid 5597's current affinity mask: 7
[root@openEuler ~]# taskset -pc 5597
pid 5597's current affinity list: 0-2
```

2.4.2 分析与小结

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

上面为 while_long.c 的核心代码，这是一个死循环的程序，因此运行此程序后，其对应进程将一直存在，故可以方便查看此进程信息。

打开另一个终端后，执行 top 指令。首行各项对应的含义如下表所示

字段名称	含义
PID	进程ID
USER	进程所有者的实际用户名
PR	进程调度优先级，当PR状态为rt时，进程运行在real time（实时状态）模式
NI	进程优先级，越小值优先级越高,是用户层面概念，
VIRT	进程使用的虚拟内存总量
RES	驻留内存（进程使用的未被换出的物理内）大小
SHR	进程使用的共享内存大小
S	进程状态 R=running, S=interruptible sleeping, D=uninterruptible sleeping, T=stopped, Z=zombie。
%CPU	从上一次更新到现在任务所使用的CPU时间百分比
%MEM	进程使用的可用物理内存百分比
TIME+	任务启动后到现在所使用的全部CPU时间
COMMAND	进行进程所使用的命令

由此可以看出 PID 为5597的进程命令名为 `while_1+`,也就是刚刚的死循环程序对应的进程，此进程占用了一定的虚拟内存、驻留内存与共享内存，处于运行态，从上一次更新到现在此进程所使用的CPU时间百分比为100%(因为没有运行其他程序)。

Linux内核硬亲和性(affinity)指的是利用Linux内核提供给用户的API，强行将进程或者线程绑定到某一个指定的CPU核运行。通过执行 `taskset -p 5597` 命令，可以查看 PID 为5597的进程运行的CPU，显示结果 `affinity mask` 为7，对应的二进制数为111，即对应的是CPU的0,1,2三个核，，执行 `taskset -pc 5597` 命令，可以看到 `affinity list` 为0-2。这两个结果都与上面的 `echo 0-2 > cpuset.cpus` 命令对应，即使用CPU的0，1，2三个核。

3 总结

本次进行华为云的操作系统实验是在云服务器上进行的，`openEuler` 操作系统是Linux的一个发行版本。通过此次实验，我掌握了Linux的一些基本命令，适应了通过命令行与操作系统进行交互，理解了Linux中的内核模块编程，对于Linux的进程及如何查看进程信息也有了一定了解。

通过创建内核进程实验与阅读实验帮助文档，我了解到了内核线程的概念，内核线程是独立运行在内核空间的标准进程，它与普通的进程一个区别是内核线程有独立的地址空间。此外，内核线程只在内核空间运行。它们的共同点在于都可以被调度与抢占。线程函数需要在每一轮迭代后休眠一段时间以让出CPU给其他任务，否则此线程会一直占用CPU。

在打印输出当前系统CPU负载情况实验中，我对于 `proc` 文件系统有了一定了解。它能够保存系统当前工作的特殊数据，但实际上不存放在磁盘上，在执行查看命令时可以根据内核中信息创建。实验中的CPU负载信息就是创建的，尽管看起来就像是读取操作。

在打印输出当前处于运行状态的进程的PID和名字实验中，我了解到Linux中的PCB就是进程描述符 `task_struct`，通过查阅资料，对于Linux中的进程状态也有了一定认识，如0代表 `TASK_RUNNING`，1代表 `TASK_INTERRUPTIBLE` 等。

在使用cgroup实现限制CPU核数实验中，我了解了cgroup (Control Groups) 的相关知识，它是将任意进程进行分组化管理的Linux内核功能，提供将进程进行分组化管理的功能和接口的基础结构。在实验中，通过cgroup对于一个执行死循环程序的进程CPU核数进行了限制，并通过top指令查看其相关信息，最后进行了验证。

在实验期间也碰到了一些问题，如从本地windows系统上传文件至服务器Linux系统的方法、重启操作系统时需要重新选择较新的版本内核等。此外，对于一些文档中未详细提到的命令不够了解，如执行top命令后各第一行的各项所代表的具体含义，这些问题都已都通过网上查阅资料、与同学交流解决。

参考文献

- [1]<https://blog.csdn.net/sgmcumt/article/details/79135395> 使用pscp实现Windows 和 Linux服务器间远程传递文件
- [2]<https://blog.csdn.net/yanlinwang/article/details/8193709> Linux进程状态详解
- [3]<https://www.cnblogs.com/zhoug2020/p/6336453.html> Linux top命令的用法详细详解
- [4]http://blog.sina.com.cn/s/blog_6dab997f0101dhjf.html Linux top命令输出中PR值和NI值有什么不同
- [5]<https://blog.csdn.net/xluren/article/details/43202201> 如何指定进程运行的CPU(命令行 taskset)
- [6]<https://zhuanlan.zhihu.com/p/38541212> Linux中CPU亲和性 (affinity)