

操作系统原理与设计

实验手册

V2021



中国科学院软件研究所
Institute of Software, Chinese Academy of Sciences

湖南大学

华为技术有限公司

中科院软件所

目录

前言.....	1
1 实验一 Shell 的简单实现.....	3
1.1 实验目的.....	3
1.2 实验类型.....	3
1.3 预习要求.....	3
1.4 实验设备与环境.....	3
1.5 实验原理.....	3
1.6 实验任务.....	4
1.7 实验步骤和方法.....	4
2 实验二 鲲鹏云 ECS 的构建及内核编译.....	8
2.1 实验介绍.....	8
2.1.1 任务描述.....	8
2.2 实验目的.....	8
2.3 构建云实验环境.....	8
2.3.1 创建 VPC.....	8
2.3.2 购买 ECS.....	11
2.3.3 通过 ssh 登录系统.....	13
2.4 实验任务.....	14
2.4.1 openEuler 内核编译与安装.....	14
2.4.2 Hello, world!.....	17
2.5 云环境资源清理.....	18
2.5.1 ECS 关机.....	18
2.5.2 删除 ECS.....	19
3 实验三 进程管理.....	22
3.1 实验介绍.....	22
3.1.1 任务描述.....	22
3.2 实验目的.....	22
3.3 实验任务.....	22
3.3.1 创建内核进程.....	22
3.3.2 打印输出当前系统 CPU 负载情况.....	25
3.3.3 打印输出当前处于运行状态的进程的 PID 和名字.....	27



3.3.4 使用 cgroup 实现限制 CPU 核数.....	28
4 实验四 生产者-消费者问题.....	34
4.1 实验目的.....	34
4.2 实验类型.....	34
4.3 预习要求.....	34
4.4 实验设备与环境.....	34
4.5 实验原理.....	34
4.6 实验任务.....	35
4.7 实验步骤和方法.....	35

前言

鲲鹏云是“华为云鲲鹏云”的简称，本实验手册是基于鲲鹏云弹性云服务器（ECS）的 openEuler 操作系统内核编程实验手册，包括了内核编译、内存管理、进程管理、中断异常管理、内核时间管理、设备管理、文件管理以及网络管理等内核相关实验。

本实验手册不仅包含实验步骤，还在每章简要介绍了实验相关的原理和背景知识，以便读者能更好地理解操作系统内核的原理并进行实验。

一、实验网络环境介绍

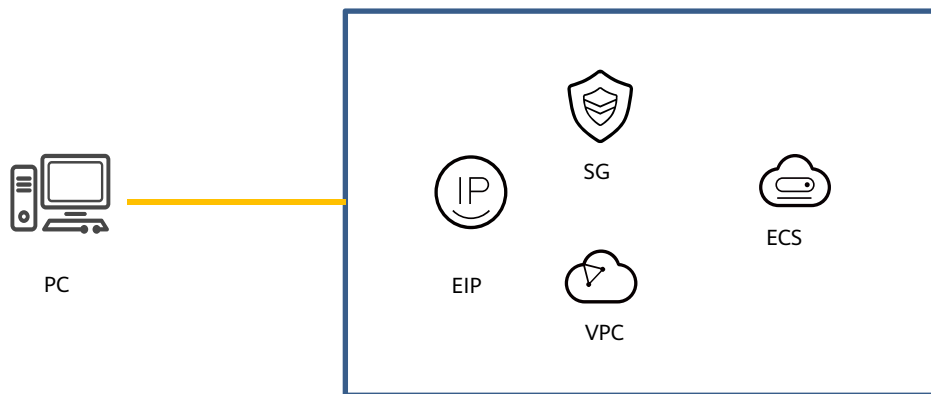


图 1-1 openEuler 操作系统实验的云环境

如上图所示，本实验的云环境主要是一台基于鲲鹏架构的 ECS，附加部件有弹性公网 IP（EIP）、虚拟私有云（VPC）和安全组（SG）。学生端的 PC 机使用 ssh 终端登录 ECS。

二、实验设备介绍

关键配置如下表所示：

表 1-1 关键配置

云资源	规格
ECS 鲲鹏计算	4vCPUs 8GB 40GB
EIP 带宽	按流量计费

软件方面，本实验需要一台终端电脑与弹性云服务器(ECS)链接以输入操作命令或/和传输文件。对于 Windows 10 / macOS / Linux，我们可以用命令行工具 ssh 和 scp 完成这个过程。

如果在有些 Windows 系统下不能运行 ssh 工具，也可以使用 Putty 和 WinSCP 工具软件。其中 Putty 工具的推荐下载地址：

<https://hcia.obs.cn-north-4.myhuaweicloud.com/v1.5/putty.exe>

WinSCP 的推荐下载地址:

<https://winscp.net/eng/index.php>

下文若无特殊说明，均以命令行工具 ssh 和 scp 为例进行讲解。

1

实验一 Shell 的简单实现

1.1 实验目的

学习如何编写一个 Unix Shell 程序，使得有机会了解如何创建子进程来执行一项专门的工作以及父进程如何继续进行子进程的工作。熟悉进程概念，了解 fork, execve, wait 等系统调用。本实验的目的主要在于学会如何在 Unix 系统下创建进程和管理进程。

1.2 实验类型

综合型实验

1.3 预习要求

完成第三章进程的学习，了解进程的基本概念及进程的创建与管理。

1.4 实验设备与环境

PII 以上电脑一台，已安装 Linux 操作系统，VC++、GCC 或其他 C 语言编译环境

1.5 实验原理

操作系统控制整个硬件与管理系统的活动监测，它不能被用户随意操作，若用户使用不当，可能会造成整个系统崩溃。但我们总是需要让用户操作系统的，这样就有了在操作系统上发展应用程序。用户可以通过应用程序来指挥内核，让内核来完成任务。在整个操作系统中，应用程序在最外层，就如同鸡蛋的外壳一样，这就是 shell 的由来。

在实现过程中，首先解析用户提交的命令行，通过 `fork()` 系统调用产生一进程，调用 `execvp()` 函数来完成命令所要求的操作。使用信号 (signals) 来通知进程事件的发生，并调用信号处理函数来完成处理工作。

1.6 实验任务

编写一个 C 语言程序作为 Linux 内核的 Shell 命令行解释程序，实现以下功能：

- (1) 解析用户提交的命令行；按照环境变量搜索目录系统；执行命令。
- (2) 提供 `ls`、`mkdir`、`rmdir`、`pwd`、`ps` 等内部命令。
- (3) 提供历史查询功能。如用户按下 `Ctrl+C`，信号处理器将输出最近的 10 个命令列表。

1.7 实验步骤和方法

(1) `setup()` 函数读取用户的下一条命令（最多 80 个字符），然后将之分析为独立的标记，这些标记被用来填充命令的参数向量（如果将要在后台运行命令，它将以 “&” 结尾，`setup()` 将会更新参数 `background`，以使 `main()` 函数相应地执行）。当用户按快捷键 `Ctrl+D` 后，`setup()` 调用 `exit()`，此程序将被终止。

`main()` 函数打印提示符 `COMMAND->`，然后调用 `setup()`，它等待用户输入命令。用户输入命令的内容被装入一个 `args` 数组。例如，如果用户在 `COMMAND->` 提示符处输入 `ls -l`，`args[0]` 等同于字符串 `ls` 和 `args[1]` 被设置为字符串 `-l`（这里的字符串指的是以 0 结束的 C 字符串变量）。

```
#include <stdio.h>

#include <unistd.h>

#define MAX LINE 80

void setup(char inputBuffer[], char *args[],int *background)
{
    //用于解析命令行的
}
```

```
int main(void)
{
    char  inputBuffer[MAXLINE]; /* buffer to hold command entered */
    int   background;           /* equals 1 if a command is followed by '&' */
    char *args[MAXLINE/2 + 1]; /* command line arguments */

    while(1)
    {
        background = 0;
        printf(" COMMAND->");
        /* setup() calls exit() when Control-D is entered */
        setup(inputBuffer,, args, &background);

        /* the steps are:
        (1) fork a child process using fork()
        (2) the child process will invoke execvp()
        (3) if background = 1, the parent will wait,
        otherwise it will invoke the setup() function again. */
    }
}
```

(2) 创建子进程

修改main()函数，以使从setup()返回时，创建一个子进程，并执行用户的命令。如前面所指出，setup()函数用用户指定命令装载args 数组的内容，args 数组将被传递给execvp()函数，该函数具有如下接口：

```
execvp(char *command, char *params[]);
```


其中command表示要执行的命令，params保存命令的参数。对于该项目，execvp()函数应作为execvp(args[0], args)来调用；需要保证检测background的值，以决定父进程是否需要等待子进程退出。

(3) 创建历史特性

信号处理函数应在main()之前声明，并且由于控制可在任意点传递给该函数，没有参数可以传递给它。因此，在程序中，它所访问的任意数据必须定义为全局，即在源文件的顶部、函数声明之前。在从信号处理函数返回之前，它应重发指令提示。

如果用户按下快捷键Ctrl+C，信号处理器将输出最近的10个命令列表。根据该列表，用户通过输入"rx"可以运行之前10个命令中的任何一个，其中"x"为该命令的第一个字母。如果有命令以"x"开头，则执行最近的一个。同样，用户可以通过仅输入"r"来再次运行最近的命令。可以假定只有一个空格来将"r"和第一个字母分开，并且该字母后面跟着"\n"。而且，如果希望执行最近的命令，单独的"r"将紧跟\n。

```
#include <signal.h>

#include <unistd.h>

#include <stdio.h>

#define BUFFER_SIZE 50

char buffer[BUFFER_SIZE];

/*信号处理函数*/

void handle_SIGINTO

{ //信号处理函数

    write(STDOUT_FILENO, buffer, strlen(buffer));

    exit (0);

}

int main(int argc, char *argv[])

{
```

```
/*创建信号处理器*/

struct sigaction handler;

    handler.sa_handler = handle_SIGINT;

handler.sa_handler = handle_SIGINT;

sigaction(SIGINT, &handler, NULL);

/*生成输出消息*/

strcpy(buffer, "Caught Control C\n");

/*循环运行，直至接收到<Ctrl+C>*/

while(1)

    ;

    return 0; ;

}
```

2

实验二 鲲鹏云 ECS 的构建及内核编译

2.1 实验介绍

本实验通过构建鲲鹏云 ECS、编译安装 openEuler 操作系统新内核以及简单的内核模块编程任务操作带领大家了解操作系统以及内核编程。

2.1.1 任务描述

- 构建鲲鹏云 ECS
- 编译安装 openEuler 操作系统新内核
- 简单的内核模块编程实验，在内核模块中打印 “Hello, world!”

2.2 实验目的

- 学习掌握如何安装构建 ECS
- 学习掌握如何编译操作系统内核
- 了解内核模块编程。

2.3 构建云实验环境

2.3.1 创建 VPC

步骤 1 在浏览器地址栏输入华为云控制台网址 console.huaweicloud.com 并按回车键，这时页面将跳转至登录页。




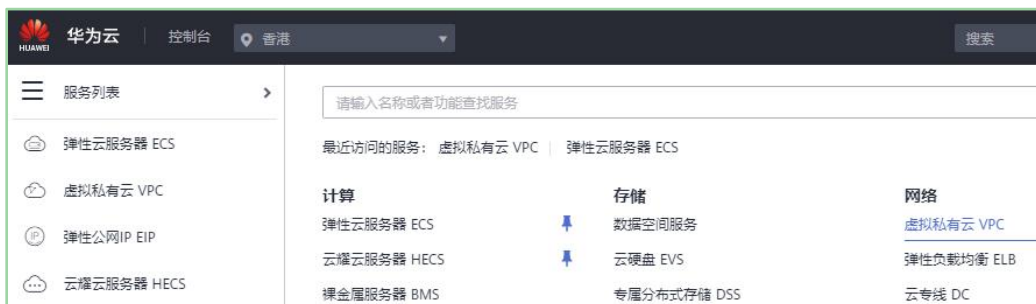
步骤 2 按要求输入账号密码，进行登录。

注意：在此之前您需要在华为云主页注册华为云账号。

步骤 3 登录成功后会自动进入控制台页面，这时将区域选在“华北-北京四”。



步骤 4 将鼠标悬停于左侧导航栏  图标处展开服务列表，然后在服务列表中点击“虚拟私有云 VPC”项。



步骤 5 点击“虚拟私有云”控制台页面右上角的“创建虚拟私有云”按钮。



步骤 6 在创建虚拟私有云的页面中按照下表内容配置虚拟私有云参数。

参数	配置
区域	华北-北京四
名称	vpc-test
网段	192.168.1.0/24
企业项目	default
默认子网可用区	可用区1
默认子网名称	subnet-test
子网网段	如192.168.1.0/24

步骤 7 配置完成后，点击“立即创建”，创建完成后会自动回到 VPC 控制台。

步骤 8 点击 VPC 控制台左侧导航栏的“访问控制”→“安全组”，进入安全组控制台。



步骤 9 点击右上角的“创建安全组”。



步骤 10 在弹出的对话框中按“通用 Web 服务器”配置安全组参数，然后点击“确定”。

创建安全组

名称

sg-test

模板

通用Web服务器

描述

通用Web服务器，默认放通22、3389、80、443端口和ICMP协议。适用于需要远程登录、公网ping及用于网站服务的云服务器场景。


0/255

查看模板规则

确定

取消

2.3.2 购买 ECS

步骤 1 将鼠标悬停于左侧导航栏图标处展开服务列表。然后在服务列表中点击“弹性云服务器 ECS”项。



步骤 2 点击弹性云服务器 ECS 控制台页面右上角的“购买弹性云服务器 ECS”按钮进入购买页面。



步骤 3 按照下表内容配置弹性云服务器 ECS 的参数。

参数	配置
计费模式	按需计费
区域	华北-北京四
可用区	可用区1
CPU架构	鲲鹏计算
规格	鲲鹏通用计算增强型 kc1.xlarge.2 4vCPUs 8GB
镜像	公共镜像 openEuler openEuler 20.03 64bit with ARM(40GB)
系统盘	通用型SSD 40GB

注意：这里“区域”的配置是和 VPC 的区域配置保持一致的。

步骤 4 配置完成后点击“下一步：网络配置”，进入网络配置，按下表配置网络参数。

参数	配置
网络	vpc-test subnet-test 自动分配IP地址
安全组	sg-test
弹性公网IP	现在购买
线路	全动态BGP
公网带宽	按流量计费
带宽大小	5Mbit/s

步骤 5 配置完成后，点击“下一步：高级配置”，按下表配置 ECS 高级配置参数。

参数	配置
云服务器名称	openEuler（输入符合规则名称）
登录凭证	密码
密码	请输入8位以上包含大小写字母、数字和特殊字符的密码，如 openEuler@123
确认密码	请再次输入密码
云备份	暂不购买

云服务器组	不配置
高级选项	不勾选

步骤 6 配置完成后点击右下角“下一步：确认配置”。勾选同意协议，然后点击：立即购买。

步骤 7 在提交任务成功后，点击“返回云服务器列表”，返回 ECS 控制台。

2.3.3 通过 ssh 登录系统

步骤 1 在 ECS 控制台查看 ECS 弹性公网 IP 地址。

<input type="checkbox"/>	名称/ID	监控	可用区	状态	规格/镜像	IP地址	计费...
<input type="checkbox"/>	openEuler 5e4b28a...		可用区1	 运行中	4vCPUs 8GB kc1.xlarge.2 openEuler 20.03 64bit with ARM	121.36... 192.168...	按需计费 2020/11/15 ...

步骤 2 在客户端机器操作系统里的 Console 控制台或 Terminal 终端里运行 ssh 命令：

```
$ ssh root@121.36.45.64
```

（注意：此处的 IP 地址 121.36.45.64 即是刚刚购买的弹性公网 IP 地址。）

在客户端（本地 PC）第一次登录时会有安全性验证的提示：

```
The authenticity of host '119.8.238.181 (119.8.238.181)' can't be established.
ECDSA key fingerprint is SHA256:RVxC1cSuMmqLtWdMw4n6f/VPsfWLkT/zDMT2q4qWxc0.
Are you sure you want to continue connecting (yes/no)? yes
```

在这里输入 yes 并按回车键继续：

```
Warning: Permanently added '119.8.238.181' (ECDSA) to the list of known hosts.
```

```
Authorized users only. All activities may be monitored and reported.
root@119.8.238.181's password:
```

输入密码(注意这里不会有任何回显)并回车，登录后的界面如下所示：

```
Welcome to Huawei Cloud Service
```

```
Last login: Mon May 18 15:35:37 2020
```

```
Welcome to Huawei Cloud Service
```

```
Last login: Mon May 18 15:35:37 2020
```


Welcome to 4.19.90-2003.4.0.0036.oe1.aarch64

System information as of time: Sun Nov 15 14:41:58 CST 2020

System load: 0.15
Processes: 131
Memory used: 5.0%
Swap used: 0.0%
Usage On: 9%
IP address: 192.168.1.5
Users online: 1

[root@openeuler ~]#

步骤 3 修改主机名

ECS 创建时被命名为“openEuler”，所以系统默认 hostname 为“openeuler”，为了和本实验手册另外两个版本保持行文上的一致，我们可以将主机名改为“openEuler”或“localhost”（一般在虚拟机中，主机名被默认为 localhost，而 ECS 也是虚拟机。本文的上下文环境中可能同时用到这三种名称，请鉴别）：

[root@openeuler ~]# vi /etc/hostname

[root@openeuler ~]# cat /etc/hostname

openEuler

[root@openeuler ~]# reboot

修改完成后重启系统并重新登录。

2.4 实验任务

2.4.1 openEuler 内核编译与安装

步骤 1 安装工具，构建开发环境：

```
[root@openEuler ~]# yum group install -y "Development Tools"
```

```
[root@openEuler ~]# yum install -y bc
```

```
[root@openEuler ~]# yum install -y openssl-devel
```

步骤 2 备份 boot 目录以防后续步骤更新内核失败

```
[root@openEuler ~]# tar czvf boot.origin.tgz /boot/
```

保存当前内核版本信息

```
[root@openEuler ~]# uname -r > uname_r.log
```

步骤 3 获取内核源代码并解压

```
[root@openEuler ~]# wget https://gitee.com/openeuler/kernel/repository/archive/kernel-4.19.zip
[root@openEuler ~]# unzip kernel-4.19.zip
```

步骤 4 编译内核

```
[root@openEuler ~]# cd kernel
[root@openEuler kernel]# make openeuler_defconfig
```

在这里,我们按源代码文件 kernel/arch/arm64/configs/openeuler_defconfig 的配置配置内核。

```
[root@openEuler kernel]# make help | grep Image
* Image.gz      - Compressed kernel image (arch/arm64/boot/Image.gz)
  Image         - Uncompressed kernel image (arch/arm64/boot/Image)
```

这一步查看了可编译的 Image。

```
[root@openEuler kernel]# make -j4 Image modules dtbs
```

这一步是编译内核的 Image、modules 和 dtbs。

步骤 5 安装内核

```
[root@openEuler kernel]# make modules_install
.....
INSTALL sound/soundcore.ko
DEPMOD  4.19.154
[root@openEuler kernel]# make install
/bin/sh ./arch/arm64/boot/install.sh 4.19.154 \
arch/arm64/boot/Image System.map "/boot"
dracut-install: Failed to find module 'xen-blkfront'
dracut: FAILED:  /usr/lib/dracut/dracut-install -D /var/tmp/dracut.tlIdPu/initramfs --kernel-dir
/lib/modules/4.19.154/ -m virtio_gpu xen-blkfront xen-netfront virtio_blk virtio_scsi virtio_net virtio_pci
virtio_ring virtio
```

注意：在最后一步“make install”时出现的错误在这里可以忽略。

步骤 6 以 VNC 登录 ECS

<input type="checkbox"/>	名称/ID	监控	可用区	状态	规格/镜像
<input type="checkbox"/>	openEuler 5e4b28a...				openEuler 5e4b28a8-e74c-4e62-8df5-f911704b7df1 20.03 64bit with ARM

在控制台“弹性云服务器 ECS”的页面中点击刚刚创建的虚拟机“openEuler”的名字超链接,在新打开的页面中点击“远程登录”按钮:



然后以控制台提供的 VNC 方式登录:



与以 ssh 登录一样，以 root 身份登录：

```
Authorized users only. All activities may be monitored and reported.
openEuler login: [ 429.483128] systemd-rc-local-generator[2892]: /etc/rc.d/rc.local is not marked executable, skipping.
openEuler login: root
Password:
Last login: Sun Nov 15 14:51:43 from 119.3.119.18

      Welcome to Huawei Cloud Service

Welcome to 4.19.90-2003.4.0.0036.oe1.aarch64

System information as of time:  Sun Nov 15 15:51:58 CST 2020

System load:   0.00
Processes:    122
Memory used:   5.2%
Swap used:     0.0%
Usage On:      38%
IP address:    192.168.1.5
Users online:  2

[root@openEuler ~]# _
```

大部分的时间，我们仅将此作为一个监视器使用。

步骤 7 重启系统

在 ssh 终端重启操作系统：

```
[root@openEuler kernel]# reboot
```

步骤 8 登录并验证

在 VNC 窗口中选择以新编译出来的内核启动系统：

```
openEuler (4.19.154) 20.03 (LTS)
openEuler (4.19.90-2003.4.0.0036.oe1.aarch64) 20.03 (LTS)
openEuler (0-rescue-95148c976dae4cd0bfb15a0242465b8a) 20.03 (LTS)
System setup

Use the ▲ and ▼ keys to change the selection.
Press 'e' to edit the selected item, or 'c' for a command prompt.
```

在这里新编译出来的内核版本为 4.19.154。您的子版本号可能与此不一样。

步骤 9 登录系统并查看版本

请以 VNC 和 ssh 终端登录系统，并在其中之一查看内核版本：

```
[root@openEuler ~]# uname -r
4.19.154
```

可以看出内核版本已更新。

2.4.2 Hello, world!

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。在这里我们的示例源文件存放在 tasks_k/1/task3 目录下。

```
[root@openEuler ~]# cd tasks_k/1/task3
[root@openEuler task1]# ls
helloworld.c Makefile
```



tasks_k.zip

实验中的源文件可以参考以上压缩包中内容（您可以用 scp 命令将其上传到 ECS）。

步骤 2 编译源文件

```
[root@openEuler task3]# make
make -C /root/kernel M=/root/tasks_k/1/task3 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/1/task3/helloworld.o
Building modules, stage 2.
```

```
MODPOST 1 modules
CC      /root/tasks_k/1/task3/helloworld.mod.o
LD [M]  /root/tasks_k/1/task3/helloworld.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

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

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task3]# rmmod helloworld
[root@openEuler task3]# dmesg | tail -n5
[ 708.247970] helloworld: loading out-of-tree module taints kernel.
[ 708.248513] helloworld: module verification failed: signature and/or required key missing - tainting kernel
[ 708.249859] hello_init
[ 708.250043] Hello, world!
[ 747.518247] hello_exit
```

您在 VNC 窗口中也会看到同样的结果。

（在这里，请忽略掉最开始安装模块时出现的两行错误提示信息。）

步骤 5 在虚拟机和 VNC 窗口中退出登录

```
[root@openEuler ~]# exit
```

注意：一般情况下不要 shutdown 虚拟机，若 shutdown 了虚拟机，需要联系实验管理员重启该虚拟机。

步骤 6 关闭 VNC 客户端页面

2.5 云环境资源清理

2.5.1 ECS 关机

当所实验完成后，应该对 ECS 进行关机以节约经费（ECS 关机后仍有少量扣费）。

步骤 1 回到 ECS 控制台，勾选 openEuler 虚拟机，进行关机。



在弹出的对话框中点击“是”按钮：



点击“是”按钮即可进行关机。

2.5.2 删除 ECS

可以等到所有的内核实验完成后再删除 ECS，否则每次都得重新编译内核。这里给出删除 ECS 的方法。

步骤 1 待关机完成后点击“更多”→“删除”



在弹出的对话框中勾选“释放云服务器绑定的弹性公网 IP 地址”和“删除云服务器挂载的数据盘”，然后点击“是”，删除 ECS。



步骤 2 您可以在控制台点击“更多 | 资源 | 我的资源”菜单项，检查资源是否全部删除



注意：(1) 虚拟私有云 VPC 和安全组可以不删除，以留下次使用。(2) 若在除“华北-北京四”之外区域购买了 ECS 和 EIP，请切换到那个区域查看。

3

实验三 进程管理

3.1 实验介绍

本实验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 以及使用 cgroup 限制 CPU 核数等任务操作，让学生们了解并掌握操作系统中的进程管理。

3.1.1 任务描述

- 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。
- 编写一个内核模块，实现读取系统一分钟内的 CPU 负载。
- 编写一个内核模块，打印当前系统处于运行状态的进程的 PID 和名字。
- 使用 cgroup 实现限制 CPU 核数。

3.2 实验目的

- 掌握正确编写满足功能的源文件，正确编译。
- 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。
- 了解操作系统的进程管理。

3.3 实验任务

3.3.1 创建内核进程

3.3.1.1 相关知识

一、内核线程介绍

内核经常需要在后台执行一些操作，这种任务就可以通过内核线程 (kernel thread) 完成，内核线程是指独立运行在内核空间的标准进程。内核线程和普通的进程间的区别在于：内核线程没有独立的地址空间，mm 指针被设置为 NULL；它只在内核空间运行，从来不切换到用户空间去；并且和普通进程一样，可以被调度，也可以被抢占。

内核线程只能由其它的内核线程创建，Linux 内核通过给出的函数接口与系统中的初始内核线程 kthreadd 交互，由 kthreadd 衍生出其它的内核线程。

二、相关接口函数

1、kthread_create():

函数返回一个 task_struct 指针，指向代表新建内核线程的 task_struct 结构体。注意：使用该函数创建的内核线程处于不可运行状态，需要将 kthread_create 返回的 task_struct 传递给 wake_up_process 函数，通过此函数唤醒新建的内核线程。

2、kthread_run()

头文件：<linux/kthread.h>

函数原型：struct task_struct *kthread_run(int (*threadfn)(void *data), void *data, const char *namefmt, ...);

功能：创建并启动一个线程。

参数：int (*threadfn)(void *data)----->线程函数，指定该线程要完成的任务。这个函数会一直运行直到接收到终止信号。

void *data----->线程函数的参数。

const char *namefmt----->线程名字。

3、线程函数

用户在线程函数中指定要让该线程完成的任务。该函数会一直运行，直到接收到结束信号。因此函数中需要有判断是否收到信号的语句。

```
static int func(void *data){
    while(!kthread_should_stop()){
        . . . . . 一些工作
        msleep(2000);
    }
    return 0;
}
```

注意在线程函数中需要在每一轮迭代之后休眠一定时间，让出 CPU 给其他的任务，否则创建的这个线程会一直占用 CPU，使得其他任务均瘫痪。更严重的是，使线程终止的命令也无法执行，导致这种状态一直持续下去。

4、kthread_stop():

头文件：<linux/kthread.h>

函数原型：int kthread_stop(struct task_struct *k);

功能：在模块卸载时，发送信号给 k 指向的线程，使之退出。

线程一旦启动起来之后，会一直运行，除非该线程主动调用 do_exit 函数，或者其他的进程调用 kthread_stop 函数，结束线程的运行。当然，如果线程函数永远不返回，并且不检查信号，它将永远不会停止。因此线程函数信号检查语句以及返回值非常重要。

注意，在调用 kthread_stop 函数时，线程不能已经结束运行，否则，kthread_stop 函数会一直等待。

5、kthread_should_stop():

头文件: <linux/kthread.h>

函数原型: bool kthread_should_stop(void);

功能: 该函数位于内核线程函数体内, 与 kthread_stop 配合使用, 用于接收 kthread_stop 传递的结束线程信号, 如果内核线程中未用此函数, 则 kthread_stop 使其结束。

3.3.1.2 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。在这里我们的示例源文件存放在 tasks_k/3/task1 目录下。

```
[root@openEuler ~]# cd tasks_k/3/task1
[root@openEuler task1]# ls
kthread.c  Makefile
```

步骤 2 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'
[root@openEuler task1]# ls
kthread.c  kthread.ko  kthread.mod.c  kthread.mod.o  kthread.o  Makefile  modules.order
Module.symvers
```

步骤 3 加载编译完成的内核模块, 并查看加载结果。

```
[root@openEuler task1]# insmod kthread.ko
[root@openEuler task1]# dmesg | tail -n5
[23146.801386] Create kernel thread!
[23146.801978] New kthread is running.
[23148.811025] New kthread is running.
[23150.826971] New kthread is running.
[23152.842938] New kthread is running.
```

步骤 4 卸载内核模块, 并查看结果。

```
[root@openEuler task1]# rmmod kthread
[root@openEuler task1]# dmesg | tail -n5
[23318.152447] New kthread is running.
[23320.168408] New kthread is running.
[23322.184379] New kthread is running.
[23324.200342] New kthread is running.
[23324.484171] Kill new kthread.
```

3.3.2 打印输出当前系统 CPU 负载情况

3.3.2.1 相关知识

一、proc 文件系统

1、proc 文件简介

proc 文件系统是 Linux 中的特殊文件系统，提供给用户一个可以了解内核内部工作过程的可读窗口，在运行时访问内核内部数据结构、改变内核设置的机制。

proc 文件系统能够保存系统当前工作的特殊数据，但并不存在于任何物理设备中，对其进行读写时，才根据系统中的相关信息即时生成。所有 proc 文件挂载在 /proc 目录下。

/proc 的文件可以用于访问有关内核的状态、计算机的属性、正在运行的进程的状态等信息。大部分 /proc 中的文件和目录提供系统物理环境最新的信息。它们实际上并不存在磁盘上，也不占用任何空间。（用 `ls -l` 可以显示它们的大小）当查看这些文件时，实际上是在访问存在内存中的信息，这些信息用于访问系统。

尽管 /proc 中的文件是虚拟的，但它们仍可以使用任何文件编辑器或像 'more'、'less' 或 'cat' 这样的程序来查看。当编辑程序试图打开一个虚拟文件时，这个文件就通过内核中的信息被凭空地创建了。

2、proc 文件组成

（1）有用内核信息

proc 文件系统可以被用于收集有用的关于系统和运行中的内核的信息。下面是一些重要的文件：

/proc/cpuinfo	-----	CPU 的信息（型号，家族，缓存大小等）
/proc/meminfo	-----	物理内存、交换空间等的信息
/proc/loadavg	-----	查看系统 1 分钟、5 分钟、15 分钟的平均负载情况
/proc/mounts	-----	已加载的文件系统的列表
/proc/devices	-----	可用设备的列表
/proc/filesystems	-----	被支持的文件系统
/proc/modules	-----	已加载的模块
/proc/version	-----	内核版本
/proc/cmdline	-----	系统启动时输入的内核命令行参数

proc 中的文件远不止上面列出的这么多。想要进一步了解，可以对 /proc 的每一个文件都 'more' 一下。

（2）进程相关信息

/proc 文件系统可以用于获取运行中的进程的信息。在 /proc 中有一些编号的子目录。每个编号的目录对应一个进程 id(PID)。这样，每一个运行中的进程 /proc 中都有一个用它的 PID 命名的目录。这些子目录中包含可以提供有关进程的状态和环境的重要细节信息的文件。

(3) 通过 proc 文件与内核交互

上面讨论的大部分/proc 的文件是只读的。而实际上/proc 文件系统通过/proc 中可读写的文件提供了对内核的交互机制。写这些文件可以改变内核的状态，因而要慎重改动这些文件。

/proc/sys -----目录存放所有可读写的文件的目录，可以被用于改变内核行为。

/proc/sys/kernel -----这个目录包含通用内核行为的信息。

/proc/sys/kernel/{domainname, hostname} 存放着机器/网络的域名和主机名。

二、内核中读写文件数据的方法

有时候需要在 Linux kernel 中读写文件数据，如调试程序的时候，或者内核与用户空间交换数据的时候。在 kernel 中操作文件没有标准库可用，需要利用 kernel 的一些函数，这些函数主要有：

filp_open() 在 kernel 中打开指定文件。

filp_close() kernel 中文件的读操作。

kernel_read() kernel 中文件的写操作。

kernel_write() 关闭指定文件。

这些函数在 <linux/fs.h> 头文件中声明。具体读写接口说明如下：

3.3.2.2 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。在这里我们的示例源文件存放在 tasks_k/3/task2 目录下。

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

步骤 2 编译源文件

```
[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.ko  cpu_loadavg.mod.c  cpu_loadavg.mod.o  cpu_loadavg.o  Makefile
modules.order  Module.symvers
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task2]# insmod cpu_loadavg.ko
[root@openEuler task2]# dmesg | tail -n2
```

```
[27644.911012] Start cpu_loadavg!  
[27644.911209] The cpu loadavg in one minute is: 0.01
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task2]# rmmod cpu_loadavg  
[root@openEuler task2]# dmesg | tail -n3  
[27644.911012] Start cpu_loadavg!  
[27644.911209] The cpu loadavg in one minute is: 0.01  
[27686.382949] Exit cpu_loadavg!
```

3.3.3 打印输出当前处于运行状态的进程的 PID 和名字

3.3.3.1 相关知识

当前进程在 `/proc` 文件系统也有保存，只不过需要遍历所有进程文件夹，从 `stat` 文件中读取状态，来判定是否为当前运行进程。而内核中可用进程遍历函数来遍历所有进程，且进程描述符 `task_struct` 结构里边有 `state` 状态，`state` 为 0 的进程就是当前进程。

1、进程描述符 `task_struct`

系统中存放进程的管理和控制信息的数据结构称为进程控制块 PCB (Process Control Block)，是进程管理和控制的最重要的数据结构。

每一个进程均有一个 PCB，在创建进程时，建立 PCB，伴随进程运行的全过程，直到进程撤消而撤消。

在 Linux 中，每一个进程都有一个进程描述符 `task_struct`，也就是 PCB；`task_struct` 结构体是 Linux 内核的一种数据结构，它会被装载到 RAM 里并包含每个进程所需的所有信息。是对进程控制的唯一手段也是最有效的手段。

`task_struct` 定义在 `<linux/sched.h>` 头文件中。

2、`for_each_process`

`for_each_process` 是一个宏，定义在 `<linux/sched/signal.h>` 文件中，提供了依次访问整个任务队列的能力。

3.3.3.2 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。在这里我们的示例源文件存放在 `tasks_k/3/task3` 目录下。

```
[root@openEuler ~]# cd tasks_k/3/task3  
[root@openEuler task3]# ls  
Makefile process_info.c
```

步骤 2 编译源文件

```
[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  modules.order  Module.symvers  process_info.c  process_info.ko  process_info.mod.c
process_info.mod.o  process_info.o
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task3]# insmod process_info.ko
[root@openEuler task3]# dmesg | tail -n3
[27874.701269] 1)name:insmod 2)pid:14142 3)state:0
[27874.701507] 1)name:systemd-udevd 2)pid:14143 3)state:0
[27874.701774] 1)name:(spawn) 2)pid:14144 3)state:0
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task3]# rmmod process_info
[root@openEuler task3]# dmesg | tail -n4
[27874.701269] 1)name:insmod 2)pid:14142 3)state:0
[27874.701507] 1)name:systemd-udevd 2)pid:14143 3)state:0
[27874.701774] 1)name:(spawn) 2)pid:14144 3)state:0
[27894.806272] Exit process_info!
```

3.3.4 使用 cgroup 实现限制 CPU 核数

3.3.4.1 相关知识

一、cgroup 介绍

cgroup (Control Groups) 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。I/O 或内存的分配控制等具体的资源管理功能是通过这个功能来实现的，这些具体的资源管理功能称为 cgroup 子系统或控制器。

cgroup 的机制是：它以分组的形式对进程使用系统资源的行为进行管理和控制。也就是说，用户通过 cgroup 对所有进程进行分组，再对该分组整体进行资源的分配和控制。

cgroup 中的每个分组称为进程组，它包含多个进程。最初情况下，系统内的所有进程形成一个进程组（根进程组），根据系统对资源的需求，这个根进程组将被进一步细分为子进程组，子进程组内的进程是根进程组内进程的子集。而这些子进程组很有可能继续被进一步细分，最终，系统内所有的进程组形成一颗具有层次等级（hierarchy）关系的进程组树。

如果某个进程组内的进程创建了子进程，那么该子进程默认与父进程处于同一进程组中。也就是说，cgroup 对该进程组的资源控制同样作用于子进程。比如，我们限制进程的 CPU 使用为 20%，我们就可以建一个 cpu 占用为 20% 的 cgroup，然后将进程添加到这个 cgroup 中。当然，一个 cgroup 可以有多个进程。

cgroup 提供了一个 cgroup 虚拟文件系统，作为进行分组管理和各子系统设置的用户接口。要使用 cgroup，必须挂载 cgroup 文件系统。这时通过挂载选项指定使用哪个子系统。

cgroup 为每种可以控制的资源定义了一个子系统。典型的子系统介绍如下：

- 1) cpu 子系统：该子系统为每个进程组设置一个使用 CPU 的权重值，以此来管理进程对 CPU 的访问，限制进程的 CPU 使用率。
 - 2) cpuset 子系统：对于多核 CPU，该子系统可以设置进程组只能在指定的核上运行，并且还可以设置进程组在指定的内存节点上申请内存。如果要使用 cpuset 控制器，需要同时配置 cpuset.cpus 和 cpuset.mems 两个文件(参数)。cpuset.mems 用来设置内存节点的；cpuset.cpus 用来限制进程可以使用的 CPU 核心；这两个参数中 CPU 核心、内存节点都用 id 表示，之间用 “,” 分隔，比如 0,1,2 ；也可以用 “-” 表示范围，如 0-3 ；两者可以结合起来用。如“0-2,6,7”。在添加进程前，cpuset.cpus、cpuset.mems 必须同时设置，而且必须是兼容的，否则会出错。
 - 3) cpuacct 子系统：该子系统只用于生成当前进程组内的进程对 CPU 的使用报告。
 - 4) memory 子系统：该子系统提供了以页面为单位对内存的访问，比如对进程组设置内存使用上限等，同时可以生成内存资源报告。
 - 5) blkio 子系统：该子系统用于限制每个块设备的输入输出（比如物理设备（磁盘，固态硬盘，USB 等等）。首先，与 CPU 子系统类似，该系统通过为每个进程组设置权重来控制块设备对其的 I/O 时间；其次，该子系统也可以限制进程组的 I/O 带宽以及 IOPS。
 - 6) devices 子系统：通过该子系统可以限制进程组对设备的访问，即该允许或禁止进程组对某设备的访问。
 - 7) freezer 子系统：该子系统可以使得进程组中的所有进程挂起或恢复。
 - 8) net_cls 子系统：该子系统使用等级识别符标记网络数据包，可允许 Linux 流量控制程序识别从具体 cgroup 中生成的数据包，提供对网络带宽的访问限制，比如对发送带宽和接收带宽进程限制。
 - 9) ns 子系统：名称空间子系统，可以使不同 cgroups 下面的进程使用不同的 namespace。
- 针对运行中的内核而言，可以使用的 cgroup 子系统由 /proc/cgroup 来确认。

```
[root@openEuler ~]# cat /proc/cgroups
#subsys_name hierarchy num_cgroups enabled
cpuset 12 1 1
cpu 6 1 1
cpuacct 6 1 1
blkio 13 1 1
memory 11 77 1
devices 9 35 1
freezer 7 1 1
net_cls 5 1 1
perf_event 3 1 1
net_prio 5 1 1
hugetlb 10 1 1
pids 4 41 1
rdma 2 1 1
files 8 1 1
[root@openEuler ~]#
```

cgroups 最初的目标是为资源管理提供的一个统一的框架，既整合现有的 cpuset 等子系统，也为未来开发新的子系统提供接口。现在的 cgroups 适用于多种应用场景，从单个进程的资源控制，到实现操作系统层次的虚拟化。

cgroups 提供了以下功能：

限制进程组可以使用的资源数量。比如：memory 子系统可以为进程组设定一个 memory 使用上限，一旦进程组使用的内存达到限额再申请内存，就会出发 OOM。

进程组的优先级控制。比如：可以使用 cpu 子系统为某个进程组分配特定 cpu share。

记录进程组使用的资源数量。比如：可以使用 cpuacct 子系统记录某个进程组使用的 cpu 时间。

进程组隔离。比如：使用 ns 子系统可以使不同的进程组使用不同的 namespace，以达到隔离的目的，不同的进程组有各自的进程、网络、文件系统挂载空间。

进程组控制。比如：使用 freezer 子系统可以将进程组挂起和恢复。

二、tmpfs 文件系统

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。

在 Linux 内核中，虚拟内存资源由物理内存（RAM）和交换分区 swap 组成，这些资源是由内核中的虚拟内存子系统来负责分配和管理。tmpfs 就是和虚拟内存子系统来“打交道”的，他向虚拟内存子系统请求页来存储文档，他同 Linux 的其他请求页的部分相同，不知道分配给自己的页是在内存中还是在交换分区中。tmpfs 同 ramfs 相同，其大小也不是固定的，而是随着所需要的空间而动态的增减。

所有在 tmpfs 上储存的资料在理论上都是临时存放的，也就是说，档案不会建立在硬盘上面。一旦重新开机，所有在 tmpfs 里面的资料都会消失不见。理论上，内存使用量会随着 tmpfs 的使用而时有增长或消减。tmpfs 将所有内容放入内核内部高速缓存中，并进行扩展和收缩以容纳其中包含的文件，并且能够将不需要的页面交换出来以交换空间。它具有最大大小限制，可以通过“mount -o remount”即时调整。

由于 tmpfs 完全存在于页面缓存和交换中，因此所有 tmpfs 页面将在 /proc/meminfo 中显示为“Shmem”，在 free 命令后中显示为“Shared”。请注意，这些计数器还包括共享内存（shmem）。获取计数的最可靠方法是使用 df 和 du。

```
[root@openEuler ~]# cat /proc/meminfo | grep 'Shmem'
Shmem:                13312 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
[root@openEuler ~]# free
              total        used         free      shared  buff/cache   available
Mem:      6975488      321408      6089088       13312      564992      6297728
Swap:            0              0              0
[root@openEuler ~]#
```

```
[root@openEuler ~]# df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        3.1G   0    3.1G   0% /dev
tmpfs           3.4G   0    3.4G   0% /dev/shm
tmpfs           3.4G  13M   3.4G   1% /run
tmpfs           3.4G   0    3.4G   0% /sys/fs/cgroup
/dev/vda2       39G   14G   23G  38% /
tmpfs           3.4G  64K   3.4G   1% /tmp
/dev/vda1      1022M  5.8M  1017M   1% /boot/efi
tmpfs           682M   0    682M   0% /run/user/0
[root@openEuler ~]#
```

使用 tmpfs，首先您编译内核时得选择“虚拟内存文档系统支持（Virtual memory filesystem support）”或设置 CONFIG_TMPFS=y，然后就能够加载 tmpfs 文档系统了。

详细的介绍，可参见内核源码中的官方文档：Documentation/filesystems/tmpfs.txt。

可挂载 tmpfs 格式的 cgroup 文件夹进行 cgroup 的相关操作。

三、相关命令

mount 命令：加载指定的文件系统。

echo 命令：显示文字。

taskset 命令：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）。

cgexec 命令：在指定的 cgroup 中运行任务。

3.3.4.2 实验步骤

步骤 1 安装 libcgroup: dnf install libcgroup -y

libcgroup 包含 cgroup 用户空间工具套件（如 lscgroup，lssubsys 等）以及静态或者动态库，以供其他程序调用，并且包含 debug 套件。

步骤 2 挂载 tmpfs 格式的 cgroup 文件夹

在 root 权限下执行以下命令：

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

挂载 tmpfs 文件类型：tmpfs 是直接建立在 VM 之上的，用一个简单的 mount 命令就可以创建 tmpfs 文件系统了。速度快，可以动态分配文件系统大小。

步骤 3 挂载 cpuset 管理子系统

挂载某一个 cgroups 子系统到挂载点之后，就可以通过在挂载点下面建立文件夹或者使用 cgcreate 命令的方法创建 cgroups 层级结构中的节点/控制组；对应的删除则使用 rmdir 删除文件夹，或使用 cgdelete 命令删除。

```
# 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

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

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

步骤 6 测试验证

(1) 打开一个终端，执行以下命令：

```
# gcc while_long.c -o while_long      # 编译上述 C 源文件
# ls
while_long  while_long.c
# cgexec -g cpuset:mycpuset ./while_long  # 指定在 cpuset 子系统的 mycpuset 控制组中运行
```

(2) 不要关闭上述终端，另打开一个终端，执行以下命令：

```
# top      #查看程序 while_long 的 PID，假设为 19518。输入 q 退出当前查看状态
```

执行如下：

```
top - 17:37:18 up 1:35, 2 users, load average: 1.00, 0.80, 0.41
Tasks: 122 total, 2 running, 120 sleeping, 0 stopped, 0 zombie
%Cpu(s): 25.0 us, 0.0 sy, 0.0 ni, 75.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 6812.0 total, 5852.4 free, 332.1 used, 627.4 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used. 6130.5 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
10127	root	20	0	2368	896	448	R	100.0	0.0	7:38.50	while_long
10201	root	20	0	218432	6144	3392	R	0.3	0.1	0:00.03	top
1	root	20	0	174144	16768	8832	S	0.0	0.2	0:02.23	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd

```
# taskset -p 10127      #显示的如果是 7 (111)，则测试限制 cpu 核数成功。
```

```
pid 10127's current affinity mask: 7
```

```
# taskset -pc 10127
```

```
pid 10127's current affinity list: 0-2
```



4

实验四 生产者-消费者问题

4.1 实验目的

掌握基本的同步互斥算法，理解生产者消费者模型；了解 windows 多线程的并发执行机制，线程间的同步与互斥；学习使用 windows 中基本的同步对象，掌握相应的 API。

4.2 实验类型

综合型实验

4.3 预习要求

已学习进程同步问题，对生产者-消费者问题有个基本的了解。

4.4 实验设备与环境

P4 以上电脑一台，已经安装 VC++、GCC 或其他 C 语言编译环境。

4.5 实验原理

在同一个进程地址空间内执行的两个线程。生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放出一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻塞，直到新的物品被生产出来。

4.6 实验任务

生产者/消费者模型为依据，在 Windows 环境下创建一个控制台进程，在该进程中创建 n 个线程模拟生产者和消费者，实现进程(线程)的同步与互斥。

4.7 实验步骤和方法

先初始化缓冲区长度为 5：

```
/*buffer.h*/
```

```
typedef int buffer_item;
```

```
#define BUFFER_SIZE 5
```

创建三个信号量：mutex 信号量，作为互斥信号量，用于互斥的访问缓冲区；

full 信号量，判断缓冲区是否有值，初值为 0；

empty 信号量，判断缓冲区是否有空缓冲区，初值为缓冲区数。

缓冲将会被用于两个函数：insert_item()和 remove_item()。

编写两个函数：DWORD WINAPI producer(void *param)和 DWORD WINAPI consumer(void *param)，随机函数 rand()产生随机数。

编写 main () 函数，主要功能是：

```
int main(int argc, char *argv[])
```

```
{
```

```
/*1. Get command line arguments argv[1], argv[2], argv[3]*/
```

```
/*2. Initialize buffer*/
```

```
/*3. Create producer threads(s)*/
```

```
/*4. Create consumer threads(s)*/
```

```
/*5. Sleep*/
```

```
/*6.Exit*/
```

```
}
```

打印出相应结果。