# 操作系统实验四

> openEuler 实验二 进程管理实验
> 邓语苏 2021级计科 2024/05/22

# 任务一： 创建并运行内核线程

## 前置：理解kthread.c

```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("kthread myThread is running.");
        msleep(2000);
    }
    return 0;
}
static int __init kthread_init(void) {
    printk("Create kthread myThread.\n");
    myThread = kthread_run(print, NULL, "myThread");
    return 0;
}
static void __exit kthread_exit(void) {
    printk("Kill kthread myThread.\n");
    if (myThread) kthread_stop(myThread);
}
module_init(kthread_init);
module_exit(kthread_exit)
```

首先当通过insmod命令加载内核模块时，模块的加载函数kthread_init会自动被内核执行，首先执行printk()打印出`"Create kernel thread!\n"`，接着调用kthread_run()，这个函数的功能是创建并启动一个线程，它的三个参数分别为要执行的线程函数，线程函数的参数，线程的名字。要执行的线程函数为print(),线程名字为mythread,线程函数的参数为NULL。最后，若初始化成功，则返回0
代码如下:

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

于是在未卸载内核模块时，这个线程函数print()会一直运行直到接收到终止信号。因此函数中需要有判断是否收到信号的语句。kthread_should_stop()用于接收kthread_stop传递的结束线程函数。此外要主要到在 线程函数中需要在每一轮迭代后休眠一定时间，让出CPU给其他的任务，否则创建的这个线程会一直占用CPU，使得其他任务军瘫痪。更严重的是，使线程终止的命令也无法执行，导致这种状况一直执行下去。
代码为:

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

接着若运行当使用rmmod命令来卸载内核模块时，模块的卸载函数kthread_exit()会自动被内核执行。此时执行printk(),输出`"Kill new kthread.\n"`，接着执行kthread_stop(mythread)函数,作用是在模块卸载时，发送信息给mythread指向的线程，使之退出。注意到因为在调用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。所以要加个if(mythread)保证kthread_stop()在线程运行时执行。

此时模块已经卸载，线程退出。

## 1.1 验证结果

- 编写makefile文件

```shell
# Build module hello_magic_student
ifneq ($(KERNELRELEASE),)
    obj-m := kthread_stu_id.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
```

- 源码编译
  将makefile文件与kthread_stu_id.c放在一个文件夹下。进入该目录，输入`make`
- 内核模块安装
  `sudo insmod kthread_stu_id.ko id="22920212204066"`
- 确认安装成功
  `lsmod | grep kthread_stu_id`

```
[root@openEuler exp2]# lsmod | grep kthread_stu_id
kthread_stu_id         16384  0
```

- 查看内核日志
  `dmesg`
- 卸载内核
  `rmmod kthread_stu_id`
- 查看内核日志
  `dmesg`

若及时卸载内核，不会输出All digits of student ID have been printed

```
[ 4580.807262] Creating kthread stuidThread.
[ 4580.809773] Index 0 of student ID : 2.
[ 4583.947595] Index 1 of student ID : 2.
[ 4587.019665] Index 2 of student ID : 9.
[ 4590.091736] Index 3 of student ID : 2.
[ 4593.163803] Index 4 of student ID : 0.
[ 4596.235882] Index 5 of student ID : 2.
[ 4599.307943] Index 6 of student ID : 1.
[ 4602.380031] Index 7 of student ID : 2.
[ 4605.452116] Index 8 of student ID : 2.
[ 4608.524185] Index 9 of student ID : 0.
[ 4611.596256] Index 10 of student ID : 4.
[ 4614.668342] Index 11 of student ID : 0.
[ 4617.740416] Index 12 of student ID : 6.
[ 4620.812506] Index 13 of student ID : 6.
[ 4622.357963] Stopping kthread stuidThread.
```

否则会一直输出All digits of student ID have been printed，直到卸载内核

```
[ 4723.221165] Creating kthread stuidThread.
[ 4723.223540] Index 0 of student ID : 2.
[ 4726.286983] Index 1 of student ID : 3.
[ 4729.359059] Index 2 of student ID : 0.
[ 4732.431120] Index 3 of student ID : 2.
[ 4735.503186] Index 4 of student ID : 0.
[ 4738.575255] Index 5 of student ID : 1.
[ 4741.647314] Index 6 of student ID : 9.
[ 4744.719375] Index 7 of student ID : 1.
[ 4747.791452] Index 8 of student ID : 1.
[ 4750.863503] All digits of student ID have been printed
[ 4755.983625] All digits of student ID have been printed
[ 4761.103730] All digits of student ID have been printed
[ 4766.223841] All digits of student ID have been printed
[ 4771.343950] All digits of student ID have been printed
[ 4776.464061] All digits of student ID have been printed
[ 4781.584160] All digits of student ID have been printed
[ 4786.704274] All digits of student ID have been printed
[ 4791.824380] All digits of student ID have been printed
[ 4796.944486] All digits of student ID have been printed
[ 4802.064595] All digits of student ID have been printed
[ 4807.184711] All digits of student ID have been printed
[ 4812.304802] All digits of student ID have been printed
[ 4817.424898] All digits of student ID have been printed
[ 4822.545008] All digits of student ID have been printed
[ 4827.665108] All digits of student ID have been printed
[ 4832.785216] All digits of student ID have been printed
[ 4837.909362] All digits of student ID have been printed
[ 4843.025466] All digits of student ID have been printed
[ 4848.145546] All digits of student ID have been printed
[ 4853.265621] All digits of student ID have been printed
[ 4858.385771] All digits of student ID have been printed
[ 4862.579350] Stopping kthread stuidThread.
```

## 1.2 对kthread_create()、wake_up_process()、kthread_run()的理解

1. **kthread_create()**：
   - 创建线程但不立即启动，需要调用 `wake_up_process()` 来唤醒线程。
   - 原型：

```c
struct task_struct *kthread_create(int (*threadfn)(void *data),
void *data, const char *namefmt, ...);
```

   - 参数：
     - `threadfn`：线程函数指针，指向一个函数，该函数将作为新线程的执行体。
     - `data`：传递给线程函数的参数。
     - `namefmt`：线程的名字格式化字符串。
   - 返回值：返回一个指向新创建线程的指针，如果创建失败则返回 NULL。
2. **kthread_create()**：
   - 唤醒一个休眠状态的内核线程，使其开始执行。
   - 原型：

```c
void wake_up_process(struct task_struct *tsk);
```

- 参数：
  - `tsk`：指向要唤醒的内核线程的 `task_struct` 结构体的指针。
- 返回值：无。

3. **kthread_create()**：
   - 是一个快捷函数，结合了创建和启动线程的功能。
   - 原型：

```c
struct task_struct *kthread_run(int (*threadfn)(void *data),
void *data, const char namefmt[], ...);
```

   - 参数：
     - `threadfn`：线程函数指针，指向一个函数，该函数将作为新线程的执行体。
     - `data`：传递给线程函数的参数。
     - `namefmt`：线程的名字格式化字符串。
   - 返回值：返回一个指向新创建线程的指针，如果创建失败则返回一个错误代码。

# 任务二： 绑定内核线程到指定 CPU

## 2.1

- **MyPrintk中current全局变量的含义**
  `current` 是一个指针，指向当前CPU上正在执行的进程或线程的 `task_struct` 结构体。
- 判断将线程绑定到指定 **CPU** 核心时，线程应当处于什么状态?

```
[ 1468.452159] State before bind: 2
[ 1468.452256] State after bind: 2
[ 1468.452311] State after wake: 0
```

其中`bind_kthread->state`的值和含义如下：

> - TASK_RUNNING (0)：线程就绪。
> - TASK_TASK_TASK_INTERRUPTIBLE (2)：线程处于可中断睡眠状态。
> - TASK_TASK_KILLABLE (512)：任务是可以被致命信号唤醒的不可中断等待状态。

因此可知绑定前后线程都处于不可中断睡眠状态。在使用`wake_up_process(bind_kthread)`唤醒后才转为可运行态。

- 唤醒线程后能否通过 **kthread_bind()**切换线程所在 **CPU**?
  使用dmesg查看日志，发现产生warning

```
[ 3229.421989] State before bind: 2
[ 3229.423026] State after bind: 2
[ 3229.423985] State after wake: 512
[ 3229.424014] kthread bind_kthread is running on cpu 1
[ 3229.424944] kthread bind_kthread start on CPU 1
[ 3231.453134] ------------[ cut here ]------------
[ 3231.454249] WARNING: CPU: 1 PID: 2284 at kernel/kthread.c:459 __kthread_bind_mask+0
x2c/0x8c
```

  查阅资料可知：警告是由 `__kthread_bind_mask` 函数在尝试将线程绑定到 CPU 时触发的。这可能是因为目标 CPU 无效或者线程的状态不允许绑定。

在MyPrintk函数中，每次尝试更换CPU都失败。打印当前CPUID，始终为1。因此可知唤醒线程后不可切换线程所在CPU。

- 通过命令查看当前机器的 **CPU** 核数
  通过`nproc`指令可知当前CPU核数为4

```
[root@openEuler exp4]# nproc
4
```

- 若在绑定时设定的 **CPU** 核心 **ID** 超过机器本身的 **CPU** 核数，会产生什么结果?

```
[ 4496.313540] State before bind: 2
[ 4496.316203] State after bind: 2
[ 4496.318689] State after wake: 512
[ 4496.318739] kthread bind_kthread is running on cpu 0
[ 4496.321213] kthread bind_kthread start on CPU 0
[ 4498.343831] kthread bind_kthread is running on cpu 0
[ 4500.359861] kthread bind_kthread is running on cpu 0
[ 4502.375897] kthread bind_kthread is running on cpu 0
[ 4504.391937] kthread bind_kthread is running on cpu 0
[ 4506.407978] kthread bind_kthread is running on cpu 0
[ 4508.424016] kthread bind_kthread is running on cpu 0
[ 4510.440071] kthread bind_kthread is running on cpu 0
[ 4512.456088] kthread bind_kthread is running on cpu 0
[ 4514.472119] kthread bind_kthread is running on cpu 0
[ 4516.488157] kthread bind_kthread is running on cpu 0
[ 4518.504202] kthread bind_kthread is running on cpu 0
[ 4520.520230] kthread bind_kthread is running on cpu 0
[ 4522.536265] kthread bind_kthread is running on cpu 0
[ 4524.552311] kthread bind_kthread is running on cpu 0
[ 4526.568342] kthread bind_kthread is running on cpu 0
[ 4528.584388] kthread bind_kthread is running on cpu 0
[ 4530.600430] kthread bind_kthread is running on cpu 0
[ 4532.616466] kthread bind_kthread is running on cpu 0
[ 4534.632497] kthread bind_kthread is running on cpu 0
[ 4536.648544] kthread bind_kthread is running on cpu 0
[ 4538.664572] kthread bind_kthread is running on cpu 0
[ 4540.680605] kthread bind_kthread is running on cpu 0
[ 4542.696653] kthread bind_kthread is running on cpu 0
[ 4544.712713] kthread bind_kthread is running on cpu 1
[ 4546.728720] kthread bind_kthread is running on cpu 1
[ 4548.744767] kthread bind_kthread is running on cpu 1
[ 4550.760786] kthread bind_kthread is running on cpu 1
[ 4552.776821] kthread bind_kthread is running on cpu 1
[ 4554.792871] kthread bind_kthread is running on cpu 1
[ 4556.808892] kthread bind_kthread is running on cpu 1
```

内核线程会自动绑定到当前空闲的CPU上。

## 2.2

> 假设当前服务器 CPU 的核数为 N， 请你编写 kthread_bind_cores.c，实现创建 N 个线程，每个线程与一个 CPU 核心绑定，并在各个线程运行时每隔 2 秒打印一次当前线程名和占用的 CPU ID， 要求每个线程使用同一个 MyPrintk()打印函数。

将内核线程0与CPU0绑定;
将内核线程1与CPU1绑定;
将内核线程2与CPU2绑定;
将内核线程3与CPU3绑定;

```
[ 5276.710167] Created kthread bind_kthread_0 and bound to CPU 0
[ 5276.711343] kthread bind_kthread_0 is running on cpu 0
[ 5276.712392] Created kthread bind_kthread_1 and bound to CPU 1
[ 5276.712411] kthread bind_kthread_1 is running on cpu 1
[ 5276.713650] Created kthread bind_kthread_2 and bound to CPU 2
[ 5276.713673] kthread bind_kthread_2 is running on cpu 2
[ 5276.717324] Created kthread bind_kthread_3 and bound to CPU 3
[ 5276.717341] kthread bind_kthread_3 is running on cpu 3
[ 5278.738395] kthread bind_kthread_3 is running on cpu 3
[ 5278.738404] kthread bind_kthread_2 is running on cpu 2
[ 5278.738412] kthread bind_kthread_0 is running on cpu 0
[ 5278.738433] kthread bind_kthread_1 is running on cpu 1
[ 5280.754415] kthread bind_kthread_2 is running on cpu 2
[ 5280.754424] kthread bind_kthread_1 is running on cpu 1
[ 5280.754434] kthread bind_kthread_0 is running on cpu 0
[ 5280.754441] kthread bind_kthread_3 is running on cpu 3
```

## 2.3

自行编写 Makefile，完成源码的编译、 内核模块安装和卸载的过程，查看内核日志，验证结果的正确性。
在2.1、2.2已附图解释说明

# 任务三：内核线程的睡眠和唤醒

## 3.1

阅读程序打印日志， 内核初始化模块中，

- **schedule_timeout_uninterruptible ()**方法将哪个线程（给出线程名称 **comm**） 进入了睡眠状态?
  将wake_up_thread(current)线程进入睡眠状态。
- 日志中线程状态是以 **long** 类型输出的， 你能给出各个 **long** 类型状态数值代表的含义吗**(**如运行状态、结束状态、 睡眠状态等**)**?

```
[  527.754325] <<--- kthread new_thread PID is: 707
[  527.754351] kthread new_thread state is: 0
[  527.754370] wake_up_thread insmod PID is :706
[  527.754388] wake_up_thread insmod state is :2 --->>
[  529.770270] <<--- kthread new_thread PID is: 707
[  529.770314] kthread new_thread state is: 0
[  529.770334] wake_up_thread insmod PID is :706
[  529.770353] wake_up_thread insmod state is :128 --->>
```

> - TASK_RUNNING (0)：线程就绪。
> - TASK_INTERRUPTIBLE (2)：线程处于可中断睡眠状态。
> - TASK_UNINTERRUPTIBLE(128)：线程是不可中断的等待状态。
> - TASK_KILLABLE (512)：线程是可以被致命信号唤醒的不可中断等待状态。

## 3.2

**执行线程睡眠方法前后以及内核模块卸载前后，线程 new_thread 和 wake_up_thread 的 PID 和状态是否发生变化？ 这种变化是必然发生的吗？ 如有变化， 请你结合代码和线程的实际运行情况， 分析 PID 或状态变化的原因。 提示：可以从线程状态转换图、 Linux 中 task_struct 结构体复用等角度进行分析。**

- 执行线程睡眠方法前后
  - 线程 new_thread 和 wake_up_thread 的 PID没有发生变化
  - 执行状态发生变化

```
[ 1654.944789] <<--- kthread new_thread PID is: 830
[ 1654.947078] kthread new_thread state is: 0
[ 1654.949357] wake_up_thread insmod PID is :829
[ 1654.951613] wake_up_thread insmod state is :2 --->>
[ 1655.744882] make current thread sleep for some time.
[ 1656.960776] <<--- kthread new_thread PID is: 830
[ 1656.965343] kthread new_thread state is: 0
[ 1656.967599] wake_up_thread insmod PID is :829
[ 1656.969926] wake_up_thread insmod state is :128 --->>
```

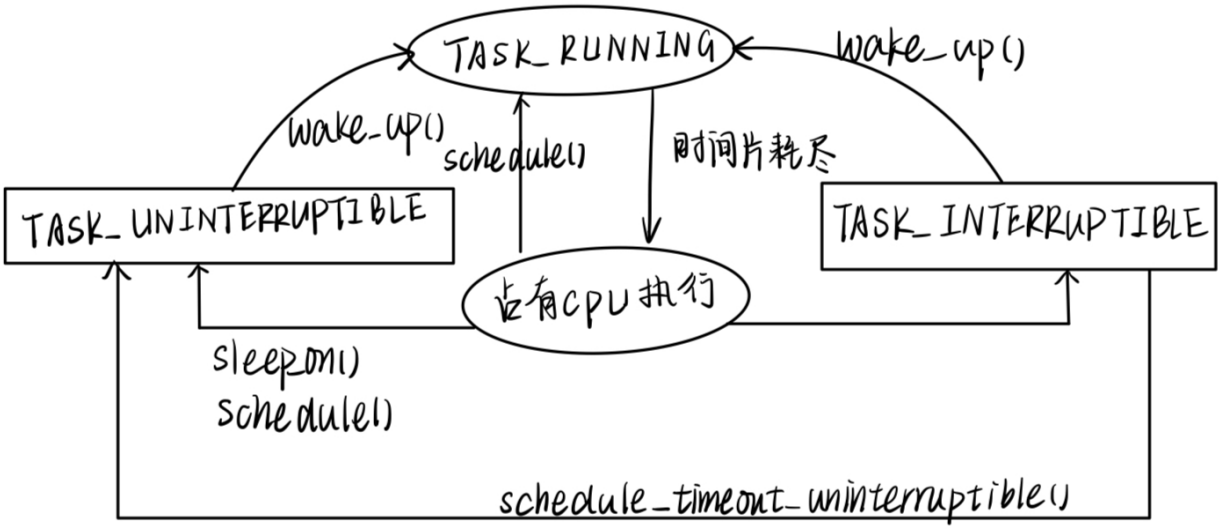
    wake_up_thread由可中断睡眠状态变为不可中断的睡眠状态。
    变化的原因：在 `schedule_timeout_uninterruptible` 被调用时，`wake_up_thread` 的状态会从 `TASK_INTERRUPTIBLE(2)` 变为 `TASK_UNINTERRUPTIBLE(128)`，这意味着线程在睡眠期间不会被信号唤醒。

- 内核模块卸载前后

```
[ 1359.877927] <<--- kthread new_thread PID is: 488
[ 1359.880284] kthread new_thread state is: 0
[ 1359.883210] wake_up_thread insmod PID is :487
[ 1359.885503] wake_up_thread insmod state is :128 --->>
[ 1361.893878] <<--- kthread new_thread PID is: 488
[ 1361.896228] kthread new_thread state is: 0
[ 1361.898569] wake_up_thread systemd-cgroups PID is :524
[ 1361.900878] wake_up_thread systemd-cgroups state is :128 --->>
[ 1363.909879] kill kthread new_thread
```

wake_up_thread的PID发生变化、执行状态未发生变化。
变化的原因：内核模块卸载后，如果 `wake_up_thread` 退出，其 结构体可能会被内核复用。这解释了为什么模块卸载后 `wake_up_thread` 的 PID 发生变化，而执行状态未发生变化。

## Linux 内核线程的状态转换图




# 任务四：利用/proc 文件系统实时获取系统状态信息

创建线程 cycle_print_kthread，要求实时显示系统当前运行时长和内存占用率，其中内存占用率需要包含三项信息：内存总空间、空闲内存空间、已用内存空间，且单位为 MB，保留整数部分即可。

以显示内存占用率为例，首先使用 `cat /proc/meminfo` 命令查看文件内容：

```
[root@openEuler exp1]# cat /proc/meminfo
MemTotal:        8000012 kB
MemFree:         7797680 kB
MemAvailable:    7825036 kB
Buffers:           12936 kB
Cached:           113320 kB
SwapCached:            0 kB
Active:            69708 kB
Inactive:          77440 kB
Active(anon):        496 kB
Inactive(anon):    37660 kB
```

可知，我们需要获取第一行和第二行的数据，而已用内存空间则由两者相减得到。
主要改动集中在`load_kernel_info` 函数，相关步骤如下：

1. 读取文件，设置缓冲区大小为128（足够覆盖一二行）

```c
pos = 0;
fp_meminfo = filp_open("/proc/meminfo", O_RDONLY, 0);
if (IS_ERR(fp_meminfo))
{
    printk("open proc file error\n");
    return -1;
}
bytes = kernel_read(fp_meminfo, buf_meminfo, sizeof(buf_meminfo),
&pos);
buf_meminfo[bytes] = '\0';
```

2. 取数。这是这个任务的难点。用 `sscanf` 可以轻松实现。

```c
sscanf(buf_meminfo, "MemTotal: %d kB\nMemFree: %d kB",
&total_memory, &free_memory);
```

3. 关闭文件。

```c
filp_close(fp_meminfo, NULL);
```

4. occupy_memory 计算及单位转换（kB → MB）

```c
occupy_memory = total_memory - free_memory;
total_memory /= 1024;
free_memory /= 1024;
occupy_memory /= 1024;
```

输出结果如下：

```
[ 2739.980212] cycle_print_kthread started
[ 2739.980261] 22920212204181
[ 2739.980348] current uptime: 2739.94s
[ 2739.980375] total memory: 7812 MB
[ 2739.980394] free memory: 7586 MB
[ 2739.980412] occupy memory: 225 MB
[ 2742.995920] current uptime: 2742.96s
[ 2742.995960] total memory: 7812 MB
[ 2742.995979] free memory: 7586 MB
[ 2742.995997] occupy memory: 225 MB
[ 2746.015920] cycle_print_kthread stopped
```

# 任务五：使用 cgroup 限制 CPU 核数

1. 安装 libcgroup 软件包并重启系统

```
Installed:
  libcgroup-2.0.3-2.oe2203sp3.aarch64

Complete!
[root@openEuler exp1]# reboot
[root@openEuler exp1]#
```

2. 挂载 tmpfs 格式的 cgroup 文件夹和 cpuset 管理子系统

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

命令说明：tmpfs 直接建立在 VM 之上，通过 mount 命令即可快速创建 tmpfs 文件系统。tmpfs 文件系统的 特点是速度快，并且可以动态分配文件系统大小。

```
[root@openEuler cgroup]# mkdir cpuset
[root@openEuler cgroup]# mount -t cgroup -o cpuset cpuset /cgroup/cpuset
```

命令说明：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；对应的删除命令为 rmdir 或 cgdelete。

3. 创建 mycpuset 控制组

```
[root@openEuler cgroup]# cd cpuset/
[root@openEuler cpuset]# mkdir mycpuset
[root@openEuler cpuset]# cd mycpuset/
```

注意：删除该控制组使用 rmdir 命令。

4. 设置 CPU 核数

```
[root@openEuler mycpuset]# sudo echo 0 > cpuset.mems #设置 0 号内存结点。mems 默认为空，因此需要填入值
[root@openEuler mycpuset]#  sudo echo 0-1 > cpuset.cpus #这里的 0-1 指的是使用 cpu0 和 cpu1 两个核
```

思考：查看当前机器的 CPU 核数，若写入 0-4 到 cpuset.cpus，会得到什么结果?

```
[root@openEuler mycpuset]# sudo echo 0-4 > cpuset.cpus
echo: write error: Invalid argument
```

会报错：参数无效

因为通过命令`cat /proc/cpuinfo`可以知道当前机器只有 0~3 四个CPU核数。

```
[root@openEuler mycpuset]# sudo cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 108.00
Features        : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 1
BogoMIPS        : 108.00
Features        : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 2
BogoMIPS        : 108.00
Features        : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 3
BogoMIPS        : 108.00
Features        : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

Hardware        : BCM2835
Revision        : d03115
Serial          : 100000007c8e9d07
```

5. 编写测试代码

```
[root@openEuler mycpuset]# mkdir -p /scripts && cd /scripts
[root@openEuler scripts]# vim while_long.c
```

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    while (1){}
    printf("Over");
    exit(0);
}
```

6. 测试验证

```
* 在当前 shell 终端执行以下命令：
![image-20240523172428159](image-20240523172428159.png)
* 保持当前终端，并打开新的 shell 终端，执行以下命令

        a. 通过 top 命令查看 while_long 进程的 PID，这里为 433.
        ![image-20240523172644574](image-20240523172644574.png)
        b. 查看 while_long 进程的处理器亲和性
        ![image-20240523172814356](image-20240523172814356.png)
        结果解释：通过 taskset 命令可以看出 while_long 的处理器亲和性掩码是 3，即二进制 11，表示本机一共有 2 个 CPU 核心，该进程可以在任何一个 CPU 核心上(0 和 1)运行。
```

实验理解

- **cgroup** 的作用：cgroup 用于限制、记录和隔离进程组的资源使用。通过 cgroup，系统管理员可以对进程的资源使用进行细粒度控制。
- **cpuset** 的作用：cpuset 是 cgroup 的一个子系统，专门用于分配和限制进程可以使用的 CPU 核心和内存节点。这对确保关键任务的资源可用性以及防止非关键任务过度使用资源非常有用。

为什么这样做：

- 资源管理：就像餐厅需要管理顾客以确保服务质量一样，操作系统需要管理进程以确保系统性能。
- 性能优化：通过限制某些进程只能使用特定的 CPU 核心，可以防止它们消耗过多资源，影响其他进程。
- 隔离和安全：确保关键任务（如餐厅中的 VIP 顾客）总能获得必要的资源，同时隔离不重要的任务，防止它们干扰关键操作。
- 公平性：保证所有进程公平地共享 CPU 资源，就像确保所有顾客都能在餐厅得到服务一样。

# 任务六：使用 cgroup 限制 CPU 利用率

**cgroup 限制 CPU 利用率的过程**

1. 挂载 **CPU** 管理子系统： 创建并进入 `/sys/fs/cgroup/cpu` 目录，然后创建一个新的控制组 `mycpu`。

2. 编写测试代码： 编写一个执行死循环操作的 C 程序，用于模拟高 CPU 使用率的场景。

3. 编译并运行测试程序： 在 `/scripts` 目录下编译并执行测试程序 `cgroup_cpu`。
   测试验证是否实现了对 CPU 利用率的限制

4. 在当前终端执行以下命令：

```
[root@openEuler scripts]# gcc cgroup_cpu.c -o cgroup_cpu
[root@openEuler scripts]# ./cgroup_cpu
```

5. 保持当前终端，并打开新的 shell 终端，执行以下命令：

   a. 执行 top 查看 while_long 进程的 PID 和 CPU 利用率

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    532 root      20   0    2044    700    620 R 100.0   0.0   1:29.23 cgroup_cpu
    534 root      20   0   26728   4940   2900 R   0.7   0.1   0:00.12 top
      1 root      20   0  166508  10376   7916 S   0.0   0.1   0:02.69 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 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
      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 rcu_tasks_kthre
     10 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
     11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
     12 root      20   0       0      0      0 S   0.0   0.0   0:00.04 ksoftirqd/0
```

   从图中可以看出，当前 while_long 进程的 PID 为 532，CPU 利用率为 100%。

   b. 设置 CPU 利用限制 保持在最近打开的 shell 终端，退出 top 命令，执行以下命令：

```
[root@openEuler scripts]# echo 20000 > /sys/fs/cgroup/cpu/mycpu/cpu.cfs_quota_us
[root@openEuler scripts]#  cat /sys/fs/cgroup/cpu/mycpu/cpu.cfs_quota_us
20000
[root@openEuler scripts]# echo 532 > /sys/fs/cgroup/cpu/mycpu/tasks
```

   c. 再次执行 top 命令查看 while_long 进程的 CPU 利用率

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    532 root      20   0    2044    700    620 R  25.0   0.0   3:02.01 cgroup_cpu
    536 root      20   0   26728   4996   2956 R   8.3   0.1   0:00.04 top
      1 root      20   0  166508  10376   7916 S   0.0   0.1   0:02.69 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 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
      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 rcu_tasks_kthre
     10 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
     11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
     12 root      20   0       0      0      0 S   0.0   0.0   0:00.04 ksoftirqd/0
     13 root      20   0       0      0      0 I   0.0   0.0   0:00.09 rcu_preempt
     14 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 migration/0
```

   从图中可以看出，经过步骤 b 后，进程 while_long 的 CPU 利用率下降至 25%。

**实验任务**：已知 **CPU** 利用率主要由 **cpu.cfs_quota_us** 和 **cpu.cfs_period_us** 两个参数决定，请你搜索相关资料，针对本任务，在实验报告中描述这两个参数的含义，并尝试调整二者的值使**cgroup_cpu** 的利用率维持在 **40%**。

**cpu.cfs_quota_us**：代表 cgroup 中进程组的 CPU 使用时间配额，单位是微秒（us）。如果设置为 `-1`（默认值），则表示没有限制，进程可以使用所有可用的 CPU 时间。如果设置了一个正值，它将限制进程组在每个 `cpu.cfs_period_us` 时间周期内可以使用的 CPU 时间。

**cpu.cfs_period_us**：定义了 CPU 使用时间配额的周期，单位也是微秒（us）。它是 `cpu.cfs_quota_us` 的时间窗口。默认情况下，这个值设置为 100000 微秒（即 100 毫秒）。

要将 cgroup_cpu 的利用率维持在 40%，只需要设置 `cpu.cfs_period_us` 为一个固定值，然后将 `cpu.cfs_quota_us` 设置为这个周期的 40%：

```
[root@openEuler scripts]# echo 10000 > /sys/fs/cgroup/cpu/mycpu/cpu.cfs_period_us
[root@openEuler scripts]# echo 4000 > /sys/fs/cgroup/cpu/mycpu/cpu.cfs_quota_us
```

结果如下图所示，成功使 cgroup_cpu 的利用率维持在 40%：

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    532 root      20   0    2044    700    620 R  40.1   0.0   5:36.17 cgroup_cpu
    548 root      20   0   26728   5088   3048 R   0.7   0.1   0:00.15 top
      1 root      20   0  166508  10376   7916 S   0.0   0.1   0:02.69 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 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
      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 rcu_tasks_kthre
     10 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
     11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
     12 root      20   0       0      0      0 S   0.0   0.0   0:00.05 ksoftirqd/0
     13 root      20   0       0      0      0 I   0.0   0.0   0:00.11 rcu_preempt
     14 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 migration/0
```