乌有理工大学

《操作系统》课程实验报告

实验题目	**实验四：中断和异常管理及系统调用**		
姓　　名	**Nemo**	**学　　号**	**2019114514**
组　　别	–	**合 作 者**	–
班　　级	**计科 NaN 班**	**指导教师**	**Outis**

1　实验概述

1.1　实验介绍

通过在使用软中断延迟机制 tasklet 实现打印 helloworld、用工作队列实现周期打印 helloworld 、在用户态下捕获终端按键信号等任务操作、为 openEuler 编写一个系统调用，让学生们了解并掌握操作系统中的中断和异常管理机制，掌握系统调用的基本原理。

1.2　任务描述

- 编写内核模块，使用 tasklet 实现打印 helloworld。
- 编写内核模块，用工作队列实现周期打印 helloworld。
- 在用户态编写一个信号捕获程序，捕获终端按键信号（中止进程、中止进程并保存内存信息、停止进程）。
- 为 openEuler 增加一个系统调用，当用户程序调用这个系统调用时，返回给用户程序当前进程、当前进程的父进程以及当前进程的所有子进程信息。

1.3　实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块，且内核模块功能满足任务所述。
- 了解操作系统的中断与异常管理。
- 掌握系统调用的基本原理。

2　实验内容

2.1　编写内核模块

用 `make` 命令编译 tasklet_intertupt.c 和 workqueue_test.c，用 `insmod` 命令加载编译完成的内核模块，用 `rmmod` 命令卸载内核模块，用 `dmesg` 命令查看内核模块在内核态的运行结果。

实验结果如下所示（openEuler 20.03）：

```
[root@ostest lab4]# cd task1
[root@ostest task1]# make
make -C /root/kernel M=/root/lab4/task1 modules
make[1]: Entering directory 'black'''black'/'black'root'black'/'black'kernel'black''
  CC [M]  /root/lab4/task1/tasklet_intertupt.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/lab4/task1/tasklet_intertupt.mod.o
  LD [M]  /root/lab4/task1/tasklet_intertupt.ko
make[1]: Leaving directory 'black'''black'/'black'root'black'/'black'kernel'black''
[root@ostest task1]# insmod tasklet_intertupt.ko
[root@ostest task1]# rmmod tasklet_intertupt.ko
[root@ostest task1]# dmesg | tail -n3
[  133.356790] Start tasklet module...
[  133.357045] Hello World! tasklet is working...
[  154.154861] Exit tasklet module...
[root@ostest task1]# cd ../task2
[root@ostest task2]# make
make -C /root/kernel M=/root/lab4/task2 modules
make[1]: Entering directory 'black'''black'/'black'root'black'/'black'kernel'black''
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory 'black'''black'/'black'root'black'/'black'kernel'black''
[root@ostest task2]# insmod workqueue_test.ko
[root@ostest task2]# rmmod workqueue_test.ko
[root@ostest task2]# dmesg | tail -n6
[  196.104525] Start workqueue_test module.
[  201.217760] Hello World!
[  216.321620] Hello World!
[  231.425472] Hello World!
[  246.529340] Hello World!
[  322.408367] Exit workqueue_test module.
```

2.2 编写信号捕获程序

Linux 提供的信号机制是一种进程间异步的通信机制，每个进程在运行时，都要通过信号机制来检查是否有信号到达，若有，便中断正在执行的程序，转向与该信号相对应的处理程序，以完成对该事件的处理；处理结束后再返回到原来的断点继续执行。实质上，信号机制是对中断机制的一种模拟，在实现上是一种软中断。

本程序中需要用到的信号有：

信号	值	动作	说明
SIGINT	2	中止进程（Term）	用户发送 CTRL+C 触发
SIGQUIT	3	中止进程并保存内存信息（Core）	用户发送 CTRL+\触发
SIGTSTP	20	停止进程（Stop）	用户发送 CTRL+Z 触发

编写代码如下所示：

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

void signal_handler(int sig) {
  switch (sig) {
    case SIGINT:
      printf("\nGet a signal: SIGINT. You pressed CTRL+C.\n");
      break;
    case SIGQUIT:
      printf("\nGet a signal: SIGQUIT. You pressed CTRL+\\.\n");
      break;
    case SIGTSTP:
      printf("\nGet a signal: SIGTSTP. You pressed CTRL+Z.\n");
      break;
  }
  exit(0);
}

int main() {
  printf("Current process ID is %d\n", getpid());
  signal(SIGINT, signal_handler);
  signal(SIGQUIT, signal_handler);
  signal(SIGTSTP, signal_handler);
  for (;;) {
  }
}
```

实验结果如下所示（openEuler 20.03）：

```
[root@ostest task3]# gcc catch_signal.c -o catch_signal
[root@ostest task3]# ./catch_signal
Current process ID is 3798
^C
Get a signal: SIGINT. You pressed CTRL+C.
[root@ostest task3]# ./catch_signal
Current process ID is 3812
^Z
Get a signal: SIGTSTP. You pressed CTRL+Z.
[root@ostest task3]# ./catch_signal
Current process ID is 3825
^\
Get a signal: SIGQUIT. You pressed CTRL+\.
```

2.3 系统调用

2.3.1 实验过程

使用 `cp -r kernel kernel.org` 备份内核源代码目录,在 ~/kernel/include/uapi/asm-generic/unistd.h 中新增一个系统调用 294，将系统调用总数增加至 295，该系统调用扩展为 `sys_proc_relate`。

把头文件复制到 ~/kernel/kernel 目录中。考虑到 gcc 编译器对 `struct` 的优化导致的内存偏移量问题，头文件中的结构体全部加上 `__attribute__((packed))`，目的是告知编译器取消编译过程中的优化对齐，按照实际字节数进行对齐。

在 ~/kernel/kernel/sys.c 中增加如下片段:

```
SYSCALL_DEFINE1(proc_relate, char *, out) {
  struct buffer *buf = kmalloc(4096, GFP_KERNEL);
  struct proc_info *info = buf->procs;
  int cur_sessionid = current->sessionid;
  struct task_struct *p;
  int cnt = 0;
  for_each_process(p) {
    if (p->sessionid == cur_sessionid) {
      info->state = 0;
      info->pid = p->pid;
      info->tgid = p->tgid;
      strncpy(info->comm, p->comm, 16);
      info->prio = p->prio;
      info->static_prio = p->static_prio;
      info->mm = p->mm;
      info->active_mm = p->active_mm;
      info->sessionid = p->sessionid;
      info->real_parent = p->real_parent->pid;
      info->parent = p->parent->pid;
      info->group_leader = p->group_leader->pid;
      info++;
      cnt++;
      if (cnt > ((4096 - sizeof(int)) / sizeof(struct proc_info)) - 1) {
        break;
      }
    }
  }
  buf->count = cnt;
  copy_to_user(out, buf, 4096);
  return 4096;
}
```

编译安装内核，并以新内核启动系统。本设计先以把 test.c 中用于调试的 IS_KERNEL_MODULE 宏设为 0，编译运行。结果如下:

```
[root@ostest ~]# ./test.o
This is the first process, and my ID = 7325
This is the second process, and my ID = 7326
This is my thread, my process ID = 7323, thread ID = 281471867613664
Here is me, and my ID = 7323
count = 6
pid = 7148, comm = sshd
pid = 7160, comm = sshd
pid = 7161, comm = bash
pid = 7323, comm = test.o
pid = 7325, comm = test.o
pid = 7326, comm = test.o
read result = 4096
```

使用 `diff -urN kernel.org kernel >mypatch.patch` 制作内核补丁。mypatch.patch 作为实验结果。

2.3.2 思考题

思考 openEuler 为什么要将中断处理的过程分为上半部和下半部?

中断处理上半部完成尽可能少的比较紧急的功能，它往往只是简单的读取寄存器中的中断状态并清除中断标志后就进行登记中断，响应速度快。中断处理下半部处理相对来说不是非常紧急的事件。

思考 tasklet 和工作队列各用在什么场合?

如果推后执行的任务需要睡眠，那么只能选择工作队列。如果推后执行的任务需要延时指定的时间再触发，那么使用工作队列。如果推后执行的任务需要在一个 tick 之内处理，则使用软中断或 tasklet，因为其可以抢占普通进程和内核线程，同时不可睡眠。如果推后执行的任务对延迟的时间没有任何要求，则使用工作队列，此时通常为无关紧要的任务。

思考 openEuler 中的信号与信号量是同一件事情吗?

信号量（semaphore，旗标）是一个同步对象，用于保持在 0 至指定最大值之间的一个计数值。信号（signals）是 Unix 操作系统中进程间通讯的一种有限制的方式。

思考 对信号的处理与对中断的处理有什么异同?

信号与中断都采用了相同的异步通信方式；当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序；都在处理完毕后返回到原来的断点；对信号或中断都可进行屏蔽。

中断有优先级，而信号没有优先级，所有的信号都是平等的；信号处理程序是在用户态下运行的，而中断处理程序是在核心态下运行；中断响应是及时的，而信号响应通常都有较大的时间延迟。

3 小结

通过在使用软中断延迟机制 tasklet 实现打印 helloworld、用工作队列实现周期打印 helloworld 、在用户态下捕获终端按键信号等任务操作、为 openEuler 编写一个系统调用，了解了并掌握了操作系统中的中断和异常管理机制，掌握了系统调用的基本原理。

本实验在华为弹性云服务器 root@121.36.*.*（openEuler 20.03）完成。