# 广州大学学生实验报告

**开课学院及实验室：**计算机科学与教育软件学院计算机软件实验室 2020**年**3**月**26**日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | 计算机科学与教育软件学院 | **年级/专业/班** | 软件171 | **姓名** | 谢金宏 | **学号** | 1706300001 |
| **实验课**  **程名称** | Linux系统实验 | | | | | **成绩** |  |
| **实验项**  **目名称** | 进程通信 | | | | | **指导**  **老师** | 张艳玲 |

## 实验目的

1. 了解什么是管道；
2. 熟悉UNIX/LINUX支持的管道通信方式；
3. 理解内核模块的编写和装载方法。

## 实验内容

1. **编写程序实现进程的管道通信。用系统调用pipe()建立一管道，二个子进程P1和P2分别向管道各写一句话：**
   1. Child 1 is sending a message!
   2. Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示（要求先接收P1，后P2）。

1. **编写一个HelloWorld内核模块，并进行装载和卸载操作。**

## 实验涉及的系统调用以及内核函数

1. **pipe()** 建立一无名管道。

系统调用格式 *pipe(filedes)*

参数定义

*int pipe(filedes);*

*int filedes[2];*

其中，filedes[1]是写入端，filedes[0]是读出端。

该函数使用头文件如下：

*#include <unistd.h>*

*#include <signal.h>*

*#include <stdio.h>*

1. **read()**

系统调用格式

*read(fd,buf,nbyte)*

功能：从fd所指示的文件中读出nbyte个字节的数据，并将它们送至由指针buf所指示的缓冲区中。如该文件被加锁，等待，直到锁打开为止。

参数定义

*int read(fd,buf,nbyte);*

*int fd;*

*char \*buf;*

*unsigned nbyte;*

1. **write()**

系统调用格式

*write (fd,buf,nbyte)*

功能：把nbyte 个字节的数据，从buf所指向的缓冲区写到由fd所指向的文件中。如文件加锁，暂停写入，直至开锁。

参数定义同read()。

1. **printk()函数**

该函数是由内核定义的，功能和C库中的printf()类似，它把要打印的日志输出到终端或系统日志。

1. **module\_init(func):**

模块加载函数，func作为模块的入口函数，当模块被插入到内核时调用它。

1. **module\_exit(func)：**

模块卸载函数，func作为模块的出口函数，当模块从内核移走时调用它。

1. **\_\_init：**

\_\_init 宏告诉告诉编译程序相关的函数和变量仅用于初始化模块，如果这个模块被编译到内核，编译程序则把将标有\_\_init的所有代码存储到特殊的内存段（.init.text）段中。

1. **\_\_exit：**

\_\_exit宏释放被\_\_init 宏存储的代码所占的这段内存。

## 实验原理与重点

所谓管道，是指能够连接一个写进程和一个读进程的、并允许它们以生产者—消费者方式进行通信的一个共享文件，又称为pipe文件。由写进程从管道的写入端（句柄1）将数据写入管道，而读进程则从管道的读出端（句柄0）读出数据。

1. **有名管道**

一个可以在文件系统中长期存在的、具有路径名的文件。用系统调用mknod()建立。它克服无名管道使用上的局限性，可让更多的进程也能利用管道进行通信。因而其它进程可以知道它的存在，并能利用路径名来访问该文件。对有名管道的访问方式与访问其他文件一样，需先用open()打开。

1. **无名管道**

一个临时文件。利用pipe()建立起来的无名文件（无路径名）。只用该系统调用所返回的文件描述符来标识该文件，故只有调用pipe()的进程及其子孙进程才能识别此文件描述符，才能利用该文件（管道）进行通信。当这些进程不再使用此管道时，核心收回其索引结点。

两种管道的读写方式是相同的，本实验只用无名管道。

1. **pipe文件的建立**

分配磁盘和内存索引结点、为读进程分配文件表项、为写进程分配文件表项、分配用户文件描述符。

1. **读/写进程互斥**

内核为地址设置一个读指针和一个写指针，按先进先出顺序读、写。

为使读、写进程互斥地访问pipe文件，需使各进程互斥地访问pipe文件索引结点中的直接地址项。因此，每次进程在访问pipe文件前，都需检查该索引文件是否已被上锁。若是，进程便睡眠等待，否则，将其上锁，进行读/写。操作结束后解锁，并唤醒因该索引结点上锁而睡眠的进程。

1. **编写简单的内核模块**

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核，作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不同的。模块和内核都在内核空间运行，模块编程在一定意义上说就是内核编程。

## 实验原始数据、结果与分析

1. **编写程序实现进程的管道通信。用系统调用pipe()建立一管道，二个子进程P1和P2分别向管道各写一句话：**
   1. Child 1 is sending a message!
   2. Child 2 is sending a message!

**父进程从管道中读出二个来自子进程的信息并显示（要求先接收P1，后P2）。**

*// task1\_pipes.c*

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#define MESSAGE\_SIZE 50

int main()

{

    int pipefd[2];

    pipe(pipefd);

    int pid1 = 0, pid2 = 0;

    pid1 = fork();

    if (pid1 == 0)

    {   *// 子进程1*

        sleep(3);

        lockf(pipefd[1], F\_LOCK, 0);

        const char \*message = "Child 1 is sending a message!";

        write(pipefd[1], message, MESSAGE\_SIZE);

        sleep(3);

        lockf(pipefd[1], F\_ULOCK, 0);

        exit(0);

    }

    else

    {   *// 主进程*

        pid2 = fork();

        if (pid2 == 0)

        {   *// 子进程2*

            sleep(1);

            lockf(pipefd[1], F\_LOCK, 0);

            const char \*message = "Child 2 is sending a message!";

            write(pipefd[1], message, MESSAGE\_SIZE);

            sleep(3);

            lockf(pipefd[1], F\_ULOCK, 0);

            exit(0);

        }

        else

        {   *// 主进程*

            char buffer[MESSAGE\_SIZE];

            wait(0);

            read(pipefd[0], buffer, MESSAGE\_SIZE);

            printf("%s\n", buffer);

            wait(0);

            read(pipefd[0], buffer, MESSAGE\_SIZE);

            printf("%s\n", buffer);

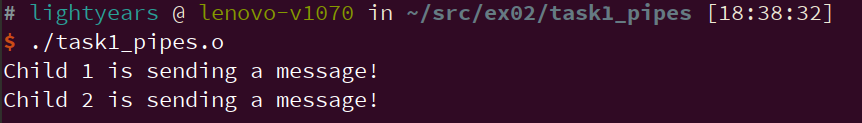
            exit(0);

        }

    }

}

在上面的代码中，主进程使用pipe()建立一个无名管道，随后创建两个子进程。两个子进程先后向管道写入信息。子进程写入信息前，需要先使用lockf(pipefd[1], F\_LOCK, 0)先锁定管道的写入端，随后使用write(pipefd[1], message, MESSAGE\_SIZE)向管道写入MESSAGE\_SIZE字节的字符串message。写入完毕后使用lockf(pipefd[1], F\_ULOCK, 0)进行对管道的写入端解锁。主进程使用wait()，等待子进程向管道写入信息并结束后，使用read(pipefd[0], buffer, MESSAGE\_SIZE)从管道中读出MESSAGE\_SIZE字节的消息装入缓存buffer，并将buffer的内容打印出来。



1. **编写一个HelloWorld内核模块，并进行装载和卸载操作。**

#include <linux/kconfig.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#define PRINTK\_TAG "task2\_kmodule: "

static int \_\_init kmodule\_init(void)

{

    printk(KERN\_INFO PRINTK\_TAG "Hello World!\n");

    return 0;

}

static void \_\_exit kmodule\_exit(void)

{

    printk(KERN\_INFO PRINTK\_TAG "Goodbye!\n");

}

module\_init(kmodule\_init);

module\_exit(kmodule\_exit);

MODULE\_LICENSE("GPL");

上面给出了一个简单的Linux内核模块。

module\_init(kmodule\_init)语句指明该内核模块在insmod进入内核后，要执行kmodule\_init函数。module\_exit(kmodule\_exit)语句指明该内核模块在rmmod即将退出内核前，要执行kmodule\_exit函数。kmodule\_init和kmodule\_exit只包含一个printk语句，用于将特定的消息打印到内核日志。KERN\_INFO宏表明日志的级别为INFO。

编写Makefile，然后使用make编译内核模块。

obj-m := task2\_kmodule.o

CURRENT\_PATH := $(shell pwd)

LINUX\_KERNEL := $(shell uname -r)

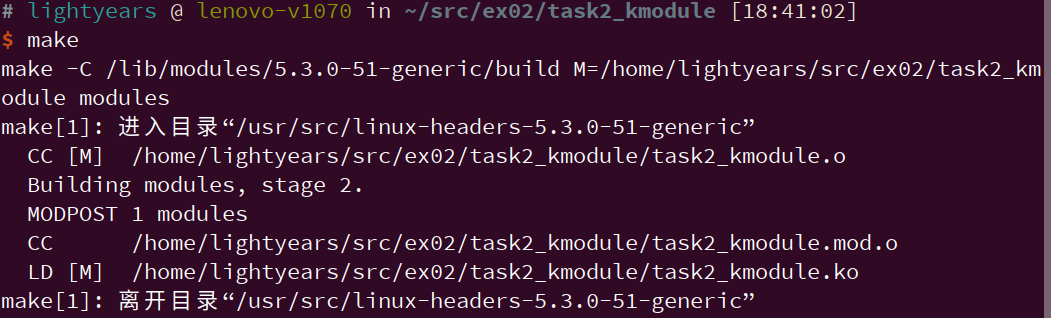
LINUX\_KERNEL\_PATH := /lib/modules/$(LINUX\_KERNEL)/build

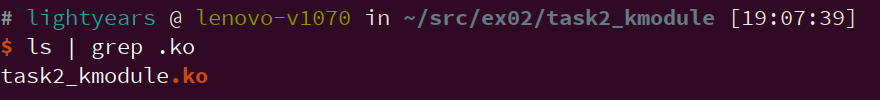
all:

    make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

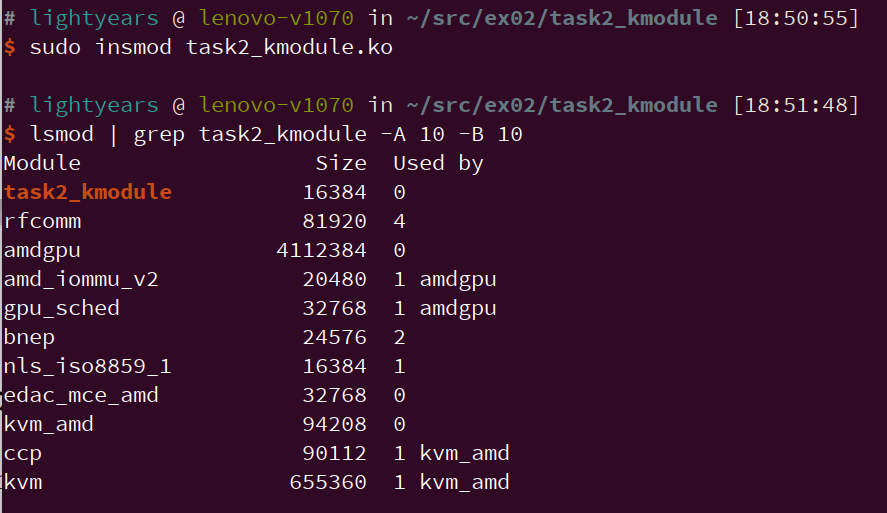
clean:

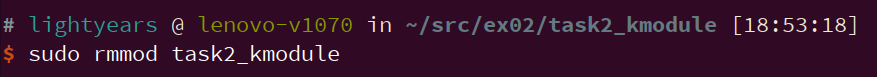
    make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) clean

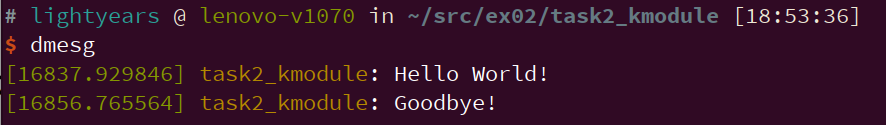




使用insmod插入内核模块，使用rmmod移除内核模块，最后使用dmesg查看内核日志。



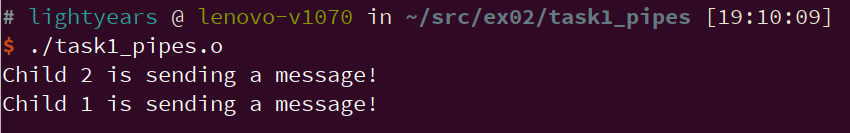




## 实验思考题

1. **程序中的sleep起什么作用？**

由于管道采用先进先出的机制，为了使父进程先读取子进程P1的消息，后读取子进程P2的消息，P1的消息必须先于P2写入管道。因此，在P2被fork之后，需要先sleep让出CPU，以确保P1先获得管道的写入权。（同理，如果令P1通过sleep让出CPU，可以使P2先获得管道写入权，导致父进程先读出P2的消息，如下图所示。）



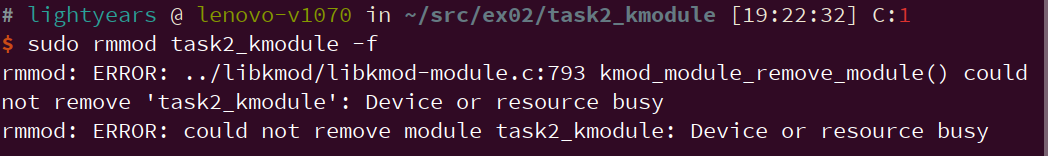
在子进程的管道加锁lockf(pipefd[1], F\_LOCK, 0)与管道解锁lockf(pipefd[1], F\_ULOCK, 0)之间进行sleep，只是为了让程序执行的时间得长一些。

1. **子进程1和2为什么也能对管道进行操作？**

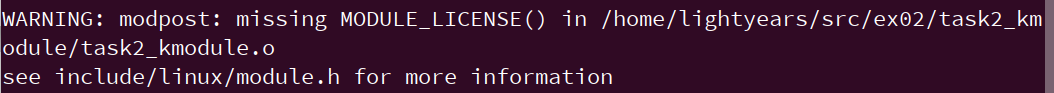
管道资源是在父进程fork出子进程之前建立的，而管道资源是通过文件描述符来标识的。因为fork，两个子进程都获得了管道文件描述符的一个拷贝，拥有这个文件描述符即可对管道进行操作。

1. **一个内核模块必须具备哪些内容？**

要编写一个内核模块，必须包含相应的内核头文件，如“linux/kernel.h”、“linux/init.h”等，并使用module\_init和module\_exit宏指定模块的初始化函数和清理函数。每个内核模块都需要一个module\_init。内核模块可以没有module\_exit，但缺少module\_exit会导致模块无法正常使用rmmod移除，如下图所示。



MODULE\_LICENSE宏用于声明内核模块的许可证，它不是必须的，但缺少它会使编译器在编译器出现警告（如下图所示）。在将缺少MODUEL\_LICENSE声明的模块插入内核时，也可能出现内核被污染的警告。



1. **在Linux内核源程序中，经常利用内核模块实现的功能有哪些？**

通过lsmod可列出实验所用的机器上面安装的内核模块。通过分析这个列表，可得知经常用内核模块实现的功能有：特定硬件的驱动程序（如amdgpu、psmouse模块）、文件系统（如autofs4模块等）和一些需要较高权限的服务（如加密服务cryptod模块和aes\_x86\_64模块以及基于内核的虚拟机kvm模块等）。

