**实 验 报 告**

|  |  |  |  |
| --- | --- | --- | --- |
| **课程名称：** | 操作系统实验 | | |
| **学 院：** | 计算机科学与工程学院 | | |
| **专 业：** | 计算机科学与技术 | **班级：** | 2019级4班 |
| **姓 名：** |  | **学号：** | 20190106 |

2021**年**12**月**31**日**

**山东科技大学教务处制**

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** |  | **同组实验者** |  |
| **实验项目名称** | 添加最简单的内核模块 kello | | | **实验日期** | 10月14日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2021年 月 日 | | |
| **一．实验目的**  1. 完成kello内核模块的添加和撤销实验；  2. 阅读和理解内核源代码及其Makefile；  3. 对比并理解、掌握应用程序和内核模块代码的差异、编译和运行的差异。  **二．实验内容**  **1. 使用vim创建编辑kello.c和Makefile**  代码内容如下：  hachi@hachi-Inspiron-7590:~/Desktop/作业/操作系统/10.14$ cat kello.c  // kello.c  #include <linux/module.h>  int hello\_init( void )  {  printk("\n Hello kernel space \n");  return 0;  }  void hello\_exit( void )  {  printk("\n Goodbye\n");  }  MODULE\_AUTHOR("weilinfox<inuyasha@sdust.edu.cn>");  MODULE\_LICENSE("GPL");  module\_init(hello\_init);  module\_exit(hello\_exit);  hachi@hachi-Inspiron-7590:~/Desktop/作业/操作系统/10.14$ cat Makefile  ifneq ($(KERNELRELEASE),)  obj-m := kello.o  else  KVERSION := $(shell uname -r)  KDIR := /lib/modules/$(KVERSION)/build  PWD := $(shell pwd)  default:  $(MAKE) -C $(KDIR) M=$(PWD) modules  rm -r -f .tmp\_versions \*.mod.c \*.cmd \*.o \*.symvers  endif  **2. 使用make命令编译运行**  输出如下：  hachi@hachi-Inspiron-7590:~/Desktop/作业/操作系统/10.14$ make  make -C /lib/modules/5.11.0-37-generic/build M=/home/hachi/Desktop/作业/操作系统/10.14 modules  make[1]: Entering directory '/usr/src/linux-headers-5.11.0-37-generic'  CC [M] /home/hachi/Desktop/作业/操作系统/10.14/kello.o  MODPOST /home/hachi/Desktop/作业/操作系统/10.14/Module.symvers  CC [M] /home/hachi/Desktop/作业/操作系统/10.14/kello.mod.o  LD [M] /home/hachi/Desktop/作业/操作系统/10.14/kello.ko  BTF [M] /home/hachi/Desktop/作业/操作系统/10.14/kello.ko  Skipping BTF generation for /home/hachi/Desktop/作业/操作系统/10.14/kello.ko due to unavailability of vmlinux  make[1]: Leaving directory '/usr/src/linux-headers-5.11.0-37-generic'  rm -r -f .tmp\_versions \*.mod.c \*.cmd \*.o \*.symvers  **3. 使用insmod和rmmod命令添加和删除内核模块**  需要sudo权限，命令如下：  hachi@hachi-Inspiron-7590:~/Desktop/作业/操作系统/10.14$ sudo insmod kello.ko  hachi@hachi-Inspiron-7590:~/Desktop/作业/操作系统/10.14$ sudo rmmod kello  **4. 使用dmesg查看printk信息**  输出如下：  hachi@hachi-Inspiron-7590:~/Desktop/作业/操作系统/10.14$ sudo cat /proc/modules | grep kello  kello 16384 0 - Live 0xffffffffc1514000 (OE)  hachi@hachi-Inspiron-7590:~/Desktop/作业/操作系统/10.14$ sudo dmesg  （部分输出省略）  [ 2049.835010] ACPI Error: Aborting method \\_SB.PCI0.LPCB.ECDV.\_Q66 due to previous error (AE\_NOT\_FOUND) (20201113/psparse-529)  [ 3044.577244]  Hello kernel space  [ 3066.646037]  Goodbye  **三．遇到的问题**  Makefile的编写必须完全按照kbuild语法，刚开始编写的时候出现make: \*\*\* No targets. Stop.的错误，原因是第一行的括号中出现了多余的空格，删去以后即可。  **四．实验结果**  结果截图附在后面 | | | | | |
| **五．实验感悟**  本次实验实现了一个自定义内核模块的载入和运行。不同于普通 C 程序直接从 main() 函数开始运行，内核模块通过 module\_init 和module\_exit 指定内核模块开始运行和结束运行的函数。内核模块在内核态运行，通过普通的手段并不容易查看到它的运行状态，而通过 printk() 可以在日志中打印运行状态，该日志可以通过 dmesg 命令查看。内核模块可以解决很多普通运行在用户态的 C 程序不能解决的问题，非常值得学习。 | | | | | |
|  | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** |  | **同组实验者** |  |
| **实验项目名称** | 进程间通信 | | | **实验日期** | 10月21日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2021年 月 日 | | |
| **一．实验目的**  1. 理解进程间通信的机制；  2. 了解进程间通信的各种方式；  3. 掌握管道通信和套接字通信的实现方法  **二．实验内容**  **1. 无名管道的使用方法1**  编写代码如下：  /\* pipe\_test01.c \*/  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <string.h>  #define BUFFER\_SIZE 25  #define READ\_END 0  #define WRITE\_END 1  int main(int argc, char \*argv[])  {  char write\_msg[BUFFER\_SIZE] = "Hello pipe";  char read\_msg[BUFFER\_SIZE];  pid\_t pid;  int fd[2];  if (pipe(fd) == -1) { /\* 创建管道 \*/  fprintf(stderr, "Pipe failed!");  return 1;  }  pid = fork(); /\* 创建子进程 \*/  if (pid < 0) {  fprintf(stderr, "Fork failed");  return 1;  }  if (pid > 0) { /\* 父进程 \*/  close(fd[READ\_END]); /\* 关闭不使用的终端 \*/  /\* 向管道中写入数据 \*/  write(fd[WRITE\_END], write\_msg, strlen(write\_msg)+1);  close(fd[WRITE\_END]);  wait(NULL);  } else { /\* 子进程 \*/  close(fd[WRITE\_END]);  /\* 从管道中读取数据 \*/  read(fd[READ\_END], read\_msg, BUFFER\_SIZE);  printf("Child read: %s\n", read\_msg);  close(fd[READ\_END]); /\* 关闭管道的读端 \*/  }  return 0;  }  **2. 无名管道的使用方法2**  编写代码如下：  /\* pipe\_test02.c \*/  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <string.h>  #define BUFFER\_SIZE 25  #define READ\_END 0  #define WRITE\_END 1  int main(int argc, char \*argv[])  {  char write\_msg[BUFFER\_SIZE] = "Hello pipe";  char read\_msg[BUFFER\_SIZE];  pid\_t pid;  int fd[2];  if (pipe(fd) == -1) { /\* 创建管道 \*/  fprintf(stderr, "Pipe failed!");  return 1;  }  pid = fork(); /\* 创建子进程 \*/  if (pid < 0) {  fprintf(stderr, "Fork failed");  return 1;  }  if (pid > 0) { /\* 父进程 \*/  close(fd[READ\_END]); /\* 关闭不使用的终端 \*/  /\* 向管道中写入数据 \*/  write(fd[WRITE\_END], write\_msg, strlen(write\_msg)+1);  close(fd[WRITE\_END]);  wait(NULL);  } else { /\* 子进程 \*/  close(0); /\* 关闭标准输入 \*/  close(fd[WRITE\_END]); /\* 关闭管道的写端 \*/  dup(fd[READ\_END]); /\* 将标准输入作为管道的读端 \*/  close(fd[READ\_END]); /\* 关闭旧的读端 \*/  execlp("/usr/bin/od", "od", "-c", NULL);  }  return 0;  }  **3. 通过无名管道向命令传输数据**  编写代码如下：  /\* pipe\_test03.c \*/  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <string.h>  #include <errno.h>  #define BUFFER\_SIZE 25  #define READ\_END 0  #define WRITE\_END 1  int main(int argc, char \*argv[])  {  char write\_msg[BUFFER\_SIZE];  char read\_msg[BUFFER\_SIZE];  pid\_t pid;  int fd[2];  if (pipe(fd) == -1) { /\* 创建管道 \*/  fprintf(stderr, "Pipe failed!");  return 1;  }  pid = fork(); /\* 创建子进程 \*/  if (pid < 0) {  fprintf(stderr, "Fork failed");  return 1;  }  if (pid > 0) { /\* 父进程 \*/  int i;  close(fd[READ\_END]); /\* 关闭不使用的终端 \*/  /\* 向管道中写入数据 \*/  for (i = 0; i < 5; i++) {  snprintf(write\_msg, BUFFER\_SIZE, "Hello %d", i);  write(fd[WRITE\_END], write\_msg, strlen(write\_msg)+1);  sleep(1);  }  close(fd[WRITE\_END]);  wait(NULL);  } else { /\* 子进程 \*/  int nstop = 1;  int nread;  close(fd[WRITE\_END]); /\* 关闭管道的写端 \*/  while (nstop) {  nread = read(fd[READ\_END], read\_msg, BUFFER\_SIZE);  switch (nread) {  case -1:  if (errno == EAGAIN) {  fprintf(stderr, "Pipe empty.\n");  sleep(1);  } else {  fprintf(stderr, "Error no: %d", errno);  }  break;  case 0:  fprintf(stdout, "End of conversation.\n");  nstop = 0;  break;  default:  fprintf(stdout, "MSG: %s\n", read\_msg);  break;  }  }  close(fd[READ\_END]); /\* 关闭旧的读端 \*/  }  return 0;  }  **4. 命名管道实现进程间通信**  编写进程1代码：  /\* fifo\_client01.c \*/  #include "fifo\_type.h"  int main(int argc, char \*argv[])  {  int pipe\_fd, sour\_fd;  int count, bytes\_sent, bytes\_read;  char buffer[PIPE\_BUF+1];  if (access(NAMEDPIPE, F\_OK) == -1) { /\* 不存在则创建 \*/  mkfifo(NAMEDPIPE, 0777);  }  fprintf(stdout, "Process %d opening FIFO O\_WRONLY\n", getpid());  pipe\_fd = open(NAMEDPIPE, O\_WRONLY); /\* 只写方式 \*/  if (pipe\_fd == -1)  exit(EXIT\_FAILURE);  sour\_fd = open(SOURTEXT, O\_RDONLY); /\* 只读方式 \*/  if (sour\_fd == -1)  exit(EXIT\_FAILURE);  fprintf(stdout, "Process %d result %d and %d\n", getpid(), pipe\_fd, sour\_fd);  /\* 读取数据文件并发送 \*/  bytes\_read = read(sour\_fd, buffer, PIPE\_BUF);  buffer[bytes\_read] = '\0';  while (bytes\_read > 0) {  count = write(pipe\_fd, buffer, bytes\_read);  bytes\_sent += count;  bytes\_read = read(sour\_fd, buffer, PIPE\_BUF);  buffer[bytes\_read] = '\0';  }  close(pipe\_fd); close(sour\_fd);  fprintf(stdout, "Process %d finished, %d bytes sent\n", getpid(), bytes\_sent);  return 0;  } | | | | | |
| 编写进程2代码如下：  /\* fifo\_server01.c \*/  #include "fifo\_type.h"  int main(int argc, char \*argv[])  {  int pipe\_fd, dest\_fd;  int count, bytes\_read, bytes\_write;  char buffer[PIPE\_BUF+1];  memset(buffer, '\0', sizeof(buffer));    pipe\_fd = open(NAMEDPIPE, O\_RDONLY); /\* 只读阻塞方式 \*/  if (pipe\_fd == -1)  exit(EXIT\_FAILURE);  fprintf(stdout, "Process %d opening FIFO O\_RDONLY.\n", pipe\_fd);  dest\_fd = open(DESTTEXT, O\_WRONLY|O\_CREAT, 0644); /\* 只写方式 \*/  if (dest\_fd == -1)  exit(EXIT\_FAILURE);  fprintf(stdout, "Process %d result %d\n", getpid(), pipe\_fd);    bytes\_read = 0;  do {  count = read(pipe\_fd, buffer, PIPE\_BUF);  /\* 保存 FIFO 中读取的数据保存在 dest\_fd \*/  bytes\_write = write(dest\_fd, buffer, count);  bytes\_read += count;  } while (count > 0);  close(pipe\_fd); close(dest\_fd);  fprintf(stdout, "Process %d finished, %d bytes read.\n", getpid(), bytes\_read);  return 0;  }  运行时需要先运行进程2再运行进程1，否则会导致实验失败。  **5. 传输结构类型的数据求和**  编写两个进程间的通信的进程，实现：传输结构类型的数据（包含3个类型（其中两个以上为 int 型）及以上的成员）。接收者将其中的 int 型求和后，返回给发送者，由原发送者将结果输出。  编写代码如下：  /\* expr.c \*/  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <string.h>  #include <errno.h>  #define BUFFER\_SIZE 256  #define WRITE\_END 1  #define READ\_END 0  int main(int argc, char \*argv[])  {  char write\_msg[2][BUFFER\_SIZE];  char read\_msg[2][BUFFER\_SIZE];  pid\_t pid;  int fd[2][2];  int nums[] = {1, 2, 3, 4, 5};  int numlen;  if (pipe(fd[0]) == -1 || pipe(fd[1]) == -1) {  fprintf(stderr, "Pipe failed!\n");  return 1;  }  pid = fork();  if (pid < 0) {  fprintf(stderr, "Fork failed!\n");  return 1;  }  numlen = sizeof(nums) / sizeof(int);  if (pid > 0) {  int i;  int ans;  close(fd[0][READ\_END]);  close(fd[1][WRITE\_END]);    for (i = 0; i < numlen; i++) {  snprintf(write\_msg[0], BUFFER\_SIZE-1, " %d", nums[i]);  fprintf(stdout, "Father write: %s\n", write\_msg[0]);  write(fd[0][WRITE\_END], write\_msg[0], strlen(write\_msg[0])+1);  read(fd[1][READ\_END], read\_msg[0], BUFFER\_SIZE-1);  }  //wait(NULL);  //sleep(1);  //fscanf(fd[FATHER\_END], "%d", &ans);  read(fd[1][READ\_END], read\_msg[0], BUFFER\_SIZE-1);  fprintf(stdout, "Father read: %s\n", read\_msg[0]);  sscanf(read\_msg[0], "%d", &ans);  fprintf(stdout, "Father get ans: %d\n", ans);  close(fd[0][READ\_END]);  close(fd[1][WRITE\_END]);  } else {  int sum = 0;  int t;  int i;  int offset;  char\* cp;  close(fd[0][WRITE\_END]);  close(fd[1][READ\_END]);  //sleep(1);    for (i = 0; i < numlen; i++) {  read(fd[0][READ\_END], read\_msg[1], BUFFER\_SIZE-1);  //cp = read\_msg[1];  fprintf(stdout, "Child read %s\n", read\_msg[1]);  sscanf(read\_msg[1], "%d", &t);  write(fd[1][WRITE\_END], "\n", 2);  sum += t;  }    sleep(1);  snprintf(write\_msg[1], BUFFER\_SIZE-1, " %d ", sum);  fprintf(stdout, "Child write: %s\n", write\_msg[1]);  write(fd[1][WRITE\_END], write\_msg[1], strlen(write\_msg[1])+1);  //fprintf(stdout, "%d\n", sum);    close(fd[1][WRITE\_END]);  close(fd[0][READ\_END]);  }  return 0;  }  程序首先开出一个子进程，并创建两个无名管道。由于管道是单向的，需要两个管道进行双向通信。主进程把五个需要求和的 int 类型依次传入管道，子进程则依次读取，并在读取的过程中求和。随后主进程读取第二个管道，等待子进程写入结果，而子进程则将结果写入第二个管道。主进程读取以后将结果打印在终端上。  **三．遇到的问题**  无名管道进行通信的时候，经常出现读取的过程中只读取一半的情况。但是如果预先写入所有需要一次读取的数据，然后再让子进程读取，就可以解决这个问题，即进程之间需要有一定的同步措施。由于实验中每个进程每次运行同一段代码的时间大致相同，所以通过在适当的地方延时就可以解决这个问题。 | | | | | |
| **四．实验结果**  编写两个进程间的通信的进程，实现：传输结构类型的数据（包含3个类型（其中两个以上为 int 型）及以上的成员）。接收者将其中的 int 型求和后，返回给发送者，由原发送者将结果输出。  实验结果如下：    可以看到，父进程首先将5个 int 型的数据传入，子进程则依次读取这5个数据，然后求和并写入，父进程成功读取并打印出来，实验成功。  **五．实验感悟**  多进程使单道批处理进化为多道批处理，但是多道程序之间互相并不知道对方在干什么。管道和套接字很好地解决了这个问题，而管道相对更简单实用。我们在进行 shell 编程时也经常使用管道运算符在各个命令之间便捷传输数据。通过管道，进程之间互相通信，高效传输数据，使程序变得更加灵活。 | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** |  | **同组实验者** | |  |
| **实验项目名称** | 进程同步与互斥  ——生产者/消费者问题 | | | **实验日期** | 月 日 | |
| **教 师 评 语** |  | | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2021年 月 日 | | | |
| **一．实验目的**  1.  2.  ……  **二．实验内容**  1.  2.  …… | | | | | | |
|  | | | | | | |
|  | | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** |  | **同组实验者** | |  |
| **实验项目名称** | Linux内存管理 | | | **实验日期** | 月 日 | |
| **教 师 评 语** |  | | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2021年 月 日 | | | |
| **一．实验目的**  1.  2.  ……  **二．实验内容**  1.  2.  …… | | | | | | |
|  | | | | | | |
|  | | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** |  | **同组实验者** | |  |
| **实验项目名称** | proc文件系统及查看进程信息 | | | **实验日期** | 月 日 | |
| **教 师 评 语** |  | | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2021年 月 日 | | | |
| **一．实验目的**  1.  2.  ……  **二．实验内容**  1.  2.  …… | | | | | | |
|  | | | | | | |
|  | | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** |  | **同组实验者** | |  |
| **实验项目名称** | Linux驱动程序 | | | **实验日期** | 月 日 | |
| **教 师 评 语** |  | | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2021年 月 日 | | | |
| **一．实验目的**  1.  2.  ……  **二．实验内容**  1.  2.  …… | | | | | | |
|  | | | | | | |
|  | | | | | | |