**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：** | 2018级 5 班 |
| **姓 名：** | 刘羽翯 | **学号：** | 201801120916 |

2020**年**12**月**31**日**

**山东科技大学教务处制**

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** | 刘羽翯 | **同组实验者** |  |
| **实验项目名称** | 添加简单的Linux内核模块 | | | **实验日期** | 10月26日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2020年 月 日 | | |
| **一．实验目的**  1. 掌握基本的Linux内核模块开发框架和编译方法  2. 掌握Linux内核模块添加流程  3. 理解Linux内核模块代码中的一些常见宏和参数  4. 掌握Linux内核模块程序和应用程序的差异  **二．实验内容**  1. 添加最简单的内核模块kello  2. 完成kello内核模块的添加和撤销实验  3. 阅读和理解内核模块kello源代码及其Makefile  4. 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  **三. 实验步骤**  1. kello.c的编译，进入源代码文件的路径，然后运行make指令：  make -j 4  2. LKM的添加和删除，添加和撤销LKM模块都需要root权限，因此，在运行时需要通过sudo命令来执行，添加模块的命令为insmod，撤销模块的命令为rmmod，可以使用dmesg来查看内核模块的运行记录。  sudo insmod kello.ko  dmesg  sudo rmmod kello.ko  核心代码如下：  #include <linux/module.h>  int hello\_init(void)  {      printk("\n Hello, students from SDUST! This is in kernel space! \n");      return 0;  }  void hello\_exit(void)  {      printk("\n Goodbye now, students from SDUST! \n");  }  MODULE\_AUTHOR("lyh@sdust");  MODULE\_LICENSE("GPL");  module\_init(*hello\_init*);  module\_exit(*hello\_exit*);  **四. 实验结果**  实验结果如下图所示：        **五. 分析**  通过上述实验结果，可以观察到内核模块开发和应用程序开发的差异，应用程序的入口为main，而Linux内核模块的入口由module\_init()指定，出口由module\_exit()指定，并且Linux内核模块的链接和运行需要用到insmod命令，在内核模块中不允许链接函数，它只允许使用内核定义的函数。 | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** | 刘羽翯 | **同组实验者** |  |
| **实验项目名称** | 进程间通信 | | | **实验日期** | 11月2,9日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2020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**  使用任意一种进程间的通信方式（无名管道、命名管道、套接字通信等）完成下面的任务  实现两个进程间的通信，一个进程从键盘输入一个结构类数据，包括：学号（数字型），姓名（字符型），个人简历（20字符以内）等类型成员。发送给另一个进程。接收进程将该结构数据输出。 | | | | | |
| **三. 实验步骤**  1. 定义结构体，学号（int），姓名（char）,个人简历（char \*）  typedef struct Student {      int no;      char name;      char introduction[N];  }Student;  2. 创建无名管道  3. 使用fork创建父子进程  4. 父进程关闭无名管道的读端，从键盘获取用户的输入，将用户输入写入无名管道，并且等待子进程结束  5. 子进程关闭无名管道的写端，从无名管道中读取数据，将数据打印到控制台显示  6. 使用gcc编译源代码并运行  7. 输入测试样例，验证实验结果是否正确  核心代码：  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/wait.h>  #define READ\_END 0  #define WRITE\_END 1  #define N 21  typedef struct Student {      int no;      char name;      char introduction[N];  }Student;  *// 使用无名管道实现两个进程之间的通信*  int main() {      Student stu1, stu2;      int length = sizeof(Student);      pid\_t pid;      int fd[2];  *// 创建管道*      if (pipe(fd) == -1) {          fprintf(stderr, "Pipe failed!\n");          return 1;      }      pid = fork();      if (pid < 0) {          fprintf(stderr, "Fork failed!\n");          return 1;      }  *// 父进程*      if (pid > 0) {  *// 关闭不使用的读端*          close(fd[READ\_END]);          printf("父进程（pid=%d）：请输入数据\n", getpid());  *// 从键盘读入数据*          scanf("%d %c", &stu1.no, &stu1.name);          getchar();          fgets(stu1.introduction, N, stdin);  *// 向管道中写入数据*          write(fd[WRITE\_END], &stu1, length);  *// 关闭管道的写端*          close(fd[WRITE\_END]);  *// 等待子进程*          wait(NULL);      } else {  *// 子进程*  *// 关闭不使用的写端*          close(fd[WRITE\_END]);  *// 从管道中读取数据*          read(fd[READ\_END], &stu2, length);          printf("子进程（pid=%d）：接收到数据\n", getpid());          printf("学号：%d\n", stu2.no);          printf("姓名：%c\n", stu2.name);          printf("个人简历：%s\n", stu2.introduction);  *// printf("%d %c %s\n", stu2.no, stu2.name, stu2.introduction);*  *// 关闭管道的读端*          close(fd[READ\_END]);      }      return 0;  }  **四. 实验结果**  测试样例如下  110 a hello world!  运行程序并输入测试样例，程序的运行结果如下图所示    **五. 分析**  管道是Linux进程间通信的主要手段之一，从本质上说，管道也是一种文件，但它又和一般文件有所不同。Linux中的管道有两大类，即无名管道和命名管道。这里使用无名管道来实现两个进程间的通信，管道两端的进程均将该管道看做是一个文件，一个进程负责往管道中写数据，而另一个进程从管道中读取数据。另外，管道的读、写行为和一般文件也不相同。当一个进程使用read读取一个空管道时，read会堵塞，直到管道中有数据写入，当一个进程试图向一个满的管道写入数据时，write将会被阻塞，直到足够多的数据被从管道中读取，write才可以将数据全部写入管道中。 | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** | 刘羽翯 | **同组实验者** |  |
| **实验项目名称** | 进程同步：生产者\消费者问题 | | | **实验日期** | 11月16,23日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2020年 月 日 | | |
| **一．实验目的**  1. 理解POSIX线程机制  2. 理解POSIX多线程的创建方法  3. 掌握POSIX多线程的同步机制  **二．实验内容**  在系统中有个缓冲区（每个可以放1个产品）构成的仓库。有个生产者，每个生产者可以生产个产品。有个消费者，每个消费者消费若干个产品，直到消费完所有的产品为止，即，其中是消费者实际消费的产品个数。  **输入**：生产者个数、消费者个数、缓冲区个数、每个生产者生产产品的个数等。  **输出**：生产者-消费者并发执行的过程、每个进程的状态变化。如：  （1）生产者将第X个产品放入仓库；  （2）消费者从仓库中消费第Y个产品；  （3）生产者阻塞；  （4）消费者阻塞；  （5）生产者被唤醒；  （6）消费者被唤醒。  **三. 实验步骤**  1. 问题分析  该问题中有多个生产者和多个消费者，缓冲区大小为b，生产者和消费者会竞争缓冲区资源，因此，应该保证生产者和消费者对缓冲区的互斥访问，使用一个互斥锁即可，同时，只有当生产者生产了产品后消费者才能消费，并且只有消费者消费了产品后，生产者才能继续生产，因此，存在两个同步结点，这里使用两个条件变量来保证线程之间的同步。  2. 定义数据结构、互斥锁、条件变量等  缓冲区使用循环队列来实现，定义互斥锁lock，两个条件变量not\_full和not\_empty  3. 初始化互斥锁、条件变量、数组等  4. 创建线程，主线程等待子线程结束  5. 释放资源  6. 用gcc编译程序并运行  gcc -o main main.c -lpthread  核心代码：  #include <stdio.h>  #include <pthread.h>  #include <stdlib.h>  *// 缓冲区*  typedef struct Buffer{      int \*buf;      int read\_ptr;      int write\_ptr;  }Buffer;  Buffer buffer;  *// 生产者n，消费者m，缓冲区大小b*  int n, m, b;  *// 产品编号*  int product\_id = 1;  int \*p\_array, \*c\_array;  int \*thread\_procuder\_id;  int \*thread\_consumer\_id;  *// 互斥锁*  pthread\_mutex\_t lock;  *// 条件变量*  pthread\_cond\_t not\_full, not\_empty;  *// 生产者*  void \*p(void \**args*) {      int idx = \*(int\*)*args*;      int num = p\_array[idx];      int i;      for (i = 0; i < num; i++) {          pthread\_mutex\_lock(&lock);          while ((buffer.write\_ptr + 1) % b == buffer.read\_ptr) {              printf("生产者p%d阻塞\n", idx+1);              pthread\_cond\_wait(&not\_full, &lock);              printf("生产者p%d被唤醒\n", idx+1);          }          printf("生产者p%d将第%d个产品放入仓库\n", idx+1, product\_id);          buffer.buf[buffer.write\_ptr] = product\_id++;          buffer.write\_ptr++;          if (buffer.write\_ptr >= b) {              buffer.write\_ptr = 0;          }          pthread\_cond\_signal(&not\_empty);          pthread\_mutex\_unlock(&lock);      }  }  *// 消费者*  void \*c(void \**args*) {      int idx = \*(int\*)*args*;      int num = c\_array[idx];      int i;      for (i = 0; i < num; i++) {          pthread\_mutex\_lock(&lock);          while (buffer.write\_ptr == buffer.read\_ptr) {              printf("消费者c%d阻塞\n", idx+1);              pthread\_cond\_wait(&not\_empty, &lock);              printf("消费者c%d被唤醒\n", idx+1);          }          printf("消费者c%d从仓库消费第%d个产品\n", idx+1, buffer.buf[buffer.read\_ptr]);          buffer.read\_ptr++;          if (buffer.read\_ptr >= b) {              buffer.read\_ptr = 0;          }          pthread\_cond\_signal(&not\_full);          pthread\_mutex\_unlock(&lock);      }  }  int main() {  *// 初始化*      buffer.read\_ptr =  0;      buffer.write\_ptr = 0;      pthread\_mutex\_init(&lock, NULL);      pthread\_cond\_init(&not\_empty, NULL);      pthread\_cond\_init(&not\_full, NULL);      scanf("%d%d%d", &n, &m, &b);      b += 1;      buffer.buf = (int\*)malloc(sizeof(int) \* b);      p\_array = (int\*)malloc(sizeof(int) \* n);      c\_array = (int\*)malloc(sizeof(int) \* m);      thread\_procuder\_id = (int\*)malloc(sizeof(int)\*n);      thread\_consumer\_id = (int\*)malloc(sizeof(int)\*m);      int i;      for (i = 0; i < n; i++) {          thread\_procuder\_id[i] = i;      }      for (i = 0; i < m; i++) {          thread\_consumer\_id[i] = i;      }      for (i = 0; i < n; i++) {          scanf("%d", p\_array+i);      }      for (i = 0; i < m; i++) {          scanf("%d", c\_array+i);      }      pthread\_t \*thread\_producer = (pthread\_t\*)malloc(sizeof(pthread\_t)\*n);      pthread\_t \*thread\_consumer = (pthread\_t\*)malloc(sizeof(pthread\_t)\*m);  *// 创建生产者和消费者线程*      for (i = 0; i < n; i++) {          pthread\_create(thread\_producer+i, NULL, p, (void\*)(thread\_procuder\_id + i));      }      for (i = 0; i < m; i++) {          pthread\_create(thread\_consumer+i, NULL, c, (void\*)(thread\_consumer\_id + i));      }  *// 主线程等待子线程执行完*      for (i = 0; i < n; i++) {          pthread\_join(thread\_producer[i], NULL);      }      for (i = 0; i < m; i++) {          pthread\_join(thread\_consumer[i], NULL);      }  *// 释放资源*      free(thread\_consumer\_id);      free(thread\_procuder\_id);      free(thread\_consumer);      free(thread\_producer);      free(p\_array);      free(c\_array);      free(buffer.buf);      pthread\_cond\_destroy(&not\_full);      pthread\_cond\_destroy(&not\_empty);      pthread\_mutex\_destroy(&lock);      return 0;  }  **四. 实验结果**  运行程序，使用如下的测试样例来验证程序的正确性  测试样例一  2 3 2  2 4  1 3 2    测试样例二  1 1 2  3  3    **五. 分析**  pthread是POSIX提供的重要线程工具包，pthread创建的线程对Linux来说是核外线程，也就是通常所说的用户线程，其调度是由pthread管理包完成的  pthread\_mutex\_t定义的mutex属于sleep-waiting类型的锁，也就是阻塞类型的锁，pthread\_mutex\_lock()操作如果没有加锁成功的话就会调用system\_wait()的系统调用，并将调用它的线程加入到这个mutex的等待队列中。  当多个线程之间只是资源竞争关系时，可以使用pthread工具包中的互斥锁来解决资源竞争问题。但在使用时需要注意互斥锁的属性，对普通互斥锁同一个线程不能二次加锁。当多个线程之间存在合作关系时，可以使用pthread工具包中的条件变量和互斥锁配合来解决该问题，但是在使用条件等待函数时需要注意防止虚假唤醒。 | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** | 刘羽翯 | **同组实验者** |  |
| **实验项目名称** | Linux内存管理 | | | **实验日期** | 11月30日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2020年 月 日 | | |
| **一．实验目的**  1. 了解Linux物理内存管理方法  2. 了解IA32架构中的逻辑地址映射机制，包括分段和分页  3. 了解Linux的分段机制及其特色  4. 理解并掌握Linux从线性地址到物理地址的映射机制  5. 理解Linux进程虚拟地址空间机制  6. 理解Linux虚拟内存区域vma的管理方法  **二．实验内容**  计算vma每个vma区域的大小，以及代码段和数据段的大小，并通过测试程序显示。 | | | | | |
| **三. 实验步骤**  1. 编译vma内核模块  make -j 4  2. 添加vma内核模块  sudo insmod vma.ko  3. 编译测试程序  gcc test.c -o test  4. 运行测试程序，读取vma信息  核心代码：  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 1024  int main() {      FILE \*fp;      char buf[MAX\_SIZE];      if (!(fp = fopen("/proc/vma", "r"))) {          printf("Open File Error!\n");          exit(-1);      }      while (fgets(buf, MAX\_SIZE, fp) != NULL) {          printf("%s", buf);      }      return 0;  }  **四. 实验结果**  程序运行结果如下图所示：      **五. 分析**  Linux在管理进程内存上主要是从虚拟内存的角度出发，而在实际分配内存时则是从物理内存角度出发。首先，一个进程的vma区域的大小都是页面大小的整数倍，也就是说，vma都是按4KB边界对齐的。  由于一个执行镜像可能因为属性不同会分裂为不同的vma，通过获取end\_code和start\_code来计算代码段的大小，同样也可以计算数据段的大小。其次，可以看到测试程序原文件本身很小，但是生成的进程后整个进程的vma区域的总大小却超过了2000KB，其中只有少量字节是测试程序的镜像文件，大部分都是由于进程执行时需要的链接库文件、匿名映射以及堆、栈产生的。 | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** | 刘羽翯 | **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** | 12月14,21日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2020年 月 日 | | |
| **一．实验目的**  1. 理解proc伪文件系统的基本概念和功能，掌握常见的操作命令  2. 了解Linux进程控制块task\_struct，并理解其重要成员变量的含义  3. 理解基于seq\_file机制的proc伪文件操作机制  4. 熟练掌握生成proc伪文件的Linux内核模块代码实现方法  **二．实验内容**  自己完成一个类似cat的读取程序，可以命名为mycat.c，用于显示/proc/tasklist中的进程信息（提示：proc文件既然是文件，就可以按照普通文件的方式对其进行读写）。 | | | | | |
| **三. 实验步骤**  1. 编译tasklist模块  make -j 4  2. 添加模块  sudo insmod tasklist.ko  3. 编译my\_cat.c  gcc -o my\_cat my\_cat.c  4. 运行my\_cat，查看结果  核心代码：  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 1024  int main() {      FILE \*fp;      char buf[MAX\_SIZE];      if (!(fp = fopen("/proc/tasklist", "r"))) {          printf("Open File Error!\n");          exit(-1);      }      while (fgets(buf, MAX\_SIZE, fp) != NULL) {          printf("%s", buf);      }      return 0;  }  **四. 实验结果**          **五. 分析**  proc伪文件系统是Linux提供的内核空间与用户空间之间进行数据交换的重要接口。之所以称其为伪文件系统，是因为它是在系统启动时动态生成的，只存在于内存中，当用户访问proc文件时，系统根据相应的内核模块动态地从系统中提取信息返回给用户。proc文件既然是一个文件，那我们就可以用读文件的方式来查看它，my\_cat程序就是利用这个原理来查看/proc/tasklist中进程信息的。 | | | | | |

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **组 别** |  | **姓 名** | 刘羽翯 | **同组实验者** |  |
| **实验项目名称** | Linux驱动程序 | | | **实验日期** | 12月 28 日 |
| **教 师 评 语** |  | | | | |
| **实验成绩：** | | | **指导教师（签名）：**  2020年 月 日 | | |
| **一．实验目的**  1. 掌握Linux设备驱动程序的基本原理  2. 掌握Linux字符设备驱动的创建流程  3. 掌握Linux字符设备驱动的常见函数  4. 理解Linux设备驱动的内存映射机制  **二．实验内容**  mmapdriver内核模块测试时，只显示了内存映射区域的信息。如果想把当前进程在内存映射后vma区域的变化显示出来，应该如何测试？ | | | | | |
| **三. 实验步骤**  1. 编译mmapdriver模块  make -j 4  2. 添加miscdev模块  sudo insmod miscdev\_map.ko  3. 给文件增加读写权限  sudo chmod a+rw /dev/mymap  4. 编译vma模块  make -j 4  5. 添加vma模块  sudo insmod vma.ko  6. 编译test程序  gcc -o test test.c  7. 运行test程序，查看结果  核心代码：  #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <fcntl.h>  #include <linux/fb.h>  #include <sys/mman.h>  #include <sys/ioctl.h>    #define PAGE\_SIZE 4096  #define MAX\_SIZE 1024  void show\_vma() {      FILE \*fp;      char buf[MAX\_SIZE];      if (!(fp = fopen("/proc/vma", "r"))) {          printf("Open File Error!\n");          exit(-1);      }      while (fgets(buf, MAX\_SIZE, fp) != NULL) {          printf("%s", buf);      }  }  int main(int *argc* , char \**argv*[])  {      int fd1;      unsigned char \*p\_map;  *//打开设备*      fd1 = open("/dev/mymap",O\_RDWR);      if(fd1 < 0)      {          printf("open fail\n");          exit(1);      }      show\_vma();      printf("\n\n----------\n\n");  *//内存映射*      p\_map = (unsigned char \*)mmap(0, PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED,fd1, 0);      if(p\_map == MAP\_FAILED)      {          printf("mmap fail\n");          goto out;      }    *//打印映射后的内存中的内容*      printf("%s\n",p\_map);      printf("\n----------\n\n");      show\_vma();    out:      munmap(p\_map, PAGE\_SIZE);      return 0;  }  **四. 实验结果**  程序的运行结果如下图所示：              **五. 分析**  Linux设备管理的核心思想是以文件方式实现对设备的管理，即设备文件，设备文件在建立过程中，需要设备号，需要与设备文件相关联的文件操作函数集合，从本质上看，这些与设备文件相关联的文件操作函数实际构成了设备驱动程序的实现。  在本次实验中，以读写的方式打开设备/dev/mymap，然后调用mmap函数对打开的文件描述符fd进行内存映射，因为要测试该进程在内存映射前后vma区域的变化，所以，我们需要在进行内存映射前和进行内存映射后，分别读取vma区域的信息，我们可以通过vma内核模块，来读取到vma区域的信息。  内存映射前的vma区域信息：    内存映射后的vma区域信息：    通过实验结果，我们可以发现，内存映射前后map\_count的值发生了变化，vma区域增加了。 | | | | | |