计算机操作系统

实验四

姓名：楚逸飞

学号：2020302878

班级：14012003

# 实验问题

**问题1：**

计算机程序（Computer Program）是指一组指示计算机或其他具有信息处理能力装置每一步动作的指令，通常用某种程序设计语言编写，运行于某种目标体系结构上。一个计算机程序是一系列指令的集合。

通常，计算机程序要经过编译和链接而成为一种人们不易看清而计算机可解读的格式，然后运行。未经编译就可运行的程序，通常称之为脚本程序（script）

在计算机科学中，库是用于开发软件的子程序集合。库和可执行文件的区别是，库不是独立程序，他们是向其他程序提供服务的代码。库链接是指把一个或多个库包括到程序中，有两种链接形式：静态链接和动态链接，相应的，前者链接的库叫做静态库，后者的叫做动态库，也叫共享库。

静态库指将所有相关的目标文件打包成为一个单独的文件-即静态库文件。以.a结尾。静态库可作为链接器的输入，链接器会将程序中使用的到函数的代码从库文件中拷贝到应用程序中。一旦链接完成，静态库就在程序里了。

共享库是一个目标模块。Linux系统上以.so后缀表示，Windows以.dll后缀。在运行时，可以加载到任意的存储器地址，并和一个在存储器中的程序链接起来，这个过程称为动态链接，是由一个叫做动态链接器的程序来执行的。应用程序在运行的时候需要共享库的支持。

系统调用是在内核中实现的，通过一定的方式把系统调用给用户，系统调用是用户程序和内核交互的接口

Linux下对文件操作有两种方式：系统调用(system call)和库函数调用(Library functions)。系统调用实际上就是指最底层的一个调用，在linux程序设计里面就是底层调用的意思。面向的是硬件。而库函数调用则面向的是应用开发的，相当于应用程序的api。库函数调用通常用于应用程序中对一般文件的访问。库函数调用是系统无关的，因此可移植性好。由于库函数调用是基于C库的，因此也就不可能用于内核空间的驱动程序中对设备的操作。

函数库调用在所有的ANSIC编译器版本中，C库函数是相同的，而系统调用在各个操作系统中的系统调用是不同的。函数库调用是调用函数库中的一段程序，而系统调用调用的是系统内核的服务。函数库调用它是与用户程序相联系，在用户地址空间执行，它的运行时间属于“用户时间”，而系统调用，它是操作系统的一个入口点，它是在内核空间执行，它的运行时间属于“系统时间”。另外从开销上来讲，系统调用由于是需要在用户空间和内核上下文环境间切换，开销较大，而函数库调用开销是比较小的。我们用一句话总结下就是：函数库调用是语言或应用程序的一部分，而系统调用是操作系统的一部分。

**问题2：**

共享内存是[进程间通信](http://baike.baidu.com/view/1492468.htm)中最简单的方式之一。共享内存允许两个或更多进程访问同一块内存，就如同 malloc() 函数向不同进程返回了指向同一个[物理内存](http://baike.baidu.com/view/138684.htm)区域的[指针](http://baike.baidu.com/view/159417.htm)。当一个进程改变了这块地址中的内容的时候，其它进程都会察觉到这个更改。因为所有[进程](http://baike.baidu.com/view/19746.htm)共享同一块内存，共享内存在各种进程间通信方式中具有最高的效率。访问共享内存区域和访问进程独有的内存区域一样快，并不需要通过[系统调用](http://baike.baidu.com/view/47173.htm)或者其它需要切入[内核](http://baike.baidu.com/view/1366.htm)的过程来完成。同时它也避免了对数据的各种不必要的复制。要使用一块共享内存，进程必须首先分配它。随后需要访问这个共享内存块的每一个进程都必须将这个共享内存绑定到自己的[地址空间](http://baike.baidu.com/view/1507129.htm)中。当完成通信之后，所有进程都将脱离共享内存，并且由一个进程释放该共享内存块。

线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程ID，当前指令指针(PC），寄存器集合和堆栈组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有就绪、阻塞和运行三种基本状态。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。线程是程序中一个单一的顺序控制流程。在单个程序中同时运行多个线程完成不同的工作，称为多线程。

# 实验主要内容

**问题1要求：**

（1）查阅相关资料，了解静态库和动态库的生成、开发、使用。并完成以下实验

**制作静态库：**

1. 分别编写add.h add.c main.c
2. add.h包含加法函数的定义 int add(int p1, intp2)
3. add.c 包含加法函数的实现
4. 对add.c进行编译生成目标文件add.o
5. 执行ar命令，生成libadd.a

⑥ 编写main.c文件，通过include add.h头文件和gcc编译实现对libadd.a中的加法函数的调用

**制作动态库：**

1. 分别编写add.h add.c main.c
2. add.h包含加法函数的定义 int add(int p1, intp2)
3. add.c 包含加法函数的实现
4. 对add.c进行编译生成共享库libadd.so
5. 编写main.c文件，通过include add.h头文件和gcc编译实现对libadd.so中的加法函数的调用

（2）查阅相关资料，了解Linux系统调用函数，并完成以下实验

检测文件当前的读写权限，如果文件具有读权限，则打印可读信息，如果有可写权限，则打印可写信息。否则返回错误信息。

如果打开文件为只读文件，则输出”read only”；

如果文件是只写文件，则输出”write only”；

如果文件可读可写，则输出”read write”；否则输出”unknown mode”

**问题2要求：**

（1）查阅相关资料，了解Linux进程间通信。并完成以下实验

编写程序，实现父子进程通过共享内存进行数据通信。父子进程通过竞争方式来创建一个共享内存单元，然后子进程接受用户输入的信息（通过argv[1]输入），并将其写入到共享内存单元；父进程则从共享内存单元将该信息读出，并显示信息的个数。具体步骤为：

1. 创建子进程；
2. 将运行参数argv[1]字符串写入共享内存；
3. 并打印写入字符串;
4. 在父进程中读出所写入的字符串并打印。
5. 最后练习memcpy和memmove内存段数据处理函数的使用

（2）查阅相关资料，了解Linux多线程编程，并完成以下实验

1、编写程序实现如下功能：编写程序，在主线程中创建一个新线程。要求在新线程中输出运行信息，在结束时返回主线程。使用函数pthread\_exit（0）退出线程。再试试exit(0)退出线程，看看是什么结果。比较一下二者的不同。

2、在两个线程之间实现交替执行输出。要求：例如主线程先输出“this is a thread0”,然后新线程输出“this is thread 2”一直交替直到结束。同时访问全局变量，修改变量的值，并打印看看输出结果。

3、编写程序实现使用信号量进行同步。编码实现输入字符串，统计每行的字符个数，以“end”结束输入。

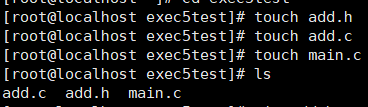
# 实验过程

**问题1:**

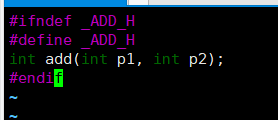
（1）查阅相关资料，了解静态库和动态库的生成、开发、使用。并完成以下实验

制作静态库：

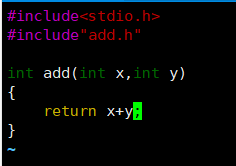
① 分别编写add.h add.c main.c



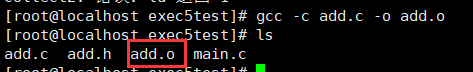
② add.h包含加法函数的定义 int add(int p1, intp2)



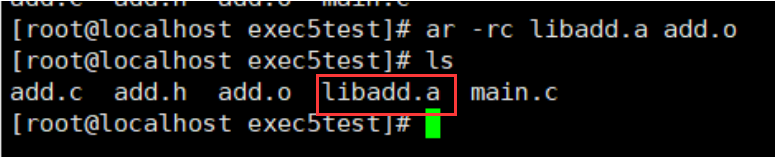
③ add.c 包含加法函数的实现



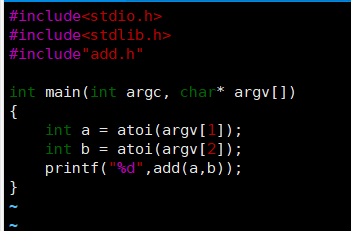
④ 对add.c进行编译生成目标文件add.o

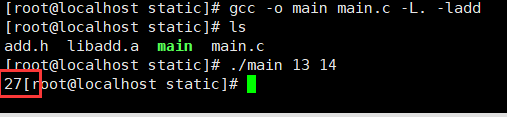


⑤ 执行ar命令，生成libadd.a



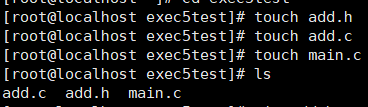
⑥ 编写main.c文件，通过include add.h头文件和gcc编译实现对libadd.a中的加法函数的调用



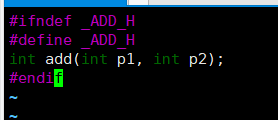


制作动态库：

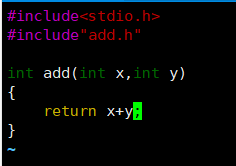
① 分别编写add.h add.c main.c



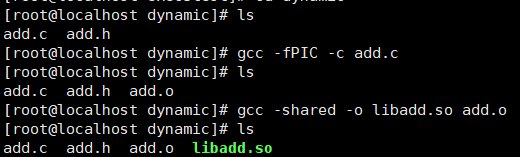
② add.h包含加法函数的定义 int add(int p1, intp2)



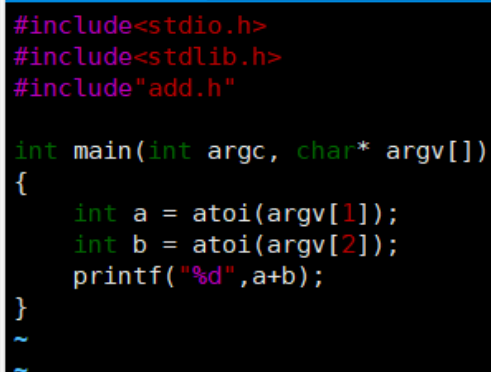
③ add.c 包含加法函数的实现

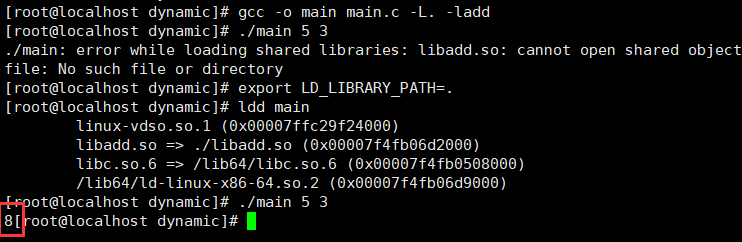


④ 对add.c进行编译生成共享库libadd.so



⑤ 编写main.c文件，通过include add.h头文件和gcc编译实现对libadd.so中的加法函数的调用





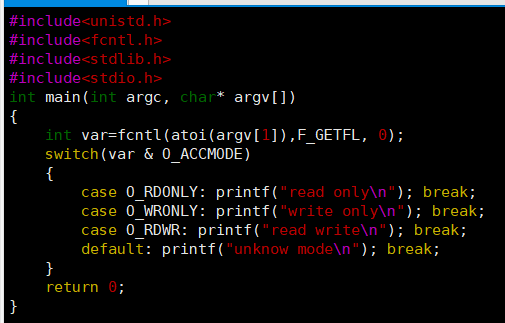
（2）查阅相关资料，了解Linux系统调用函数，并完成以下实验

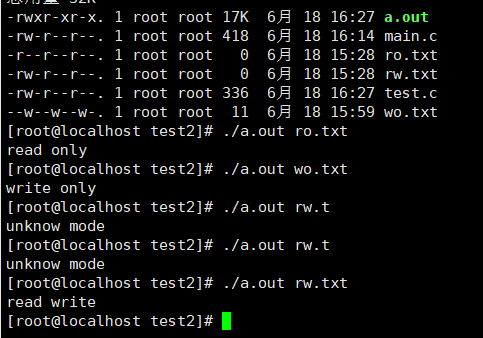
检测文件当前的读写权限，如果文件具有读权限，则打印可读信息，如果有可写权限，则打印可写信息。否则返回错误信息。

如果打开文件为只读文件，则输出”read only”；

如果文件是只写文件，则输出”write only”；

如果文件可读可写，则输出”read write”；否则输出”unknown mode”





**问题2:**

（1）查阅相关资料，了解Linux进程间通信。并完成以下实验

编写程序，实现父子进程通过共享内存进行数据通信。父子进程通过竞争方式来创建一个共享内存单元，然后子进程接受用户输入的信息（通过argv[1]输入），并将其写入到共享内存单元；父进程则从共享内存单元将该信息读出，并显示信息的个数。具体步骤为：

1. 创建子进程；

2. 将运行参数argv[1]字符串写入共享内存；

3. 并打印写入字符串;

4. 在父进程中读出所写入的字符串并打印。

5. 最后练习memcpy和memmove内存段数据处理函数的使用

**代码：**

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/shm.h>

#include <string.h>

int main(int argc, const char \*argv[])

{

key\_t key\_n ;

int shm\_id;

pid\_t pid;

int running = 1; //决定是否继续运行的变量

void \*shm = NULL; //共享内存的首地址

char buf[1024+1];

if((key\_n = ftok("/",'s')<0)) //生成共享内存的key值

{

perror("Fail to ftok");

exit(EXIT\_SUCCESS);

}

printf("key num:%d\n",key\_n);

if((shm\_id = shmget(key\_n,1024,0666|IPC\_CREAT))==-1) //创建共享内存

{

perror("Fail to shmget");

exit(EXIT\_SUCCESS);

}

printf("share memory id:%d\n",shm\_id);

if((pid = fork())==-1)

{

perror("Fail to fork");

exit(EXIT\_SUCCESS);

}

if(pid!=0) //子进程，用来写

{

printf("I is father process:%d\n",pid);

shm = shmat(shm\_id,0,0);//将共享内存连接到当前进程的地址空间

if(shm==(void \*)-1)

{

perror("Fail to shmat");

exit(EXIT\_SUCCESS);

}

printf("father Memory attached at %x\n",(int)shm);

shared = shm;//设置共享内存

shared->is\_write = 0;

while(running) //向共享内存中写数据

{

while(shared->is\_write==1)

{

printf("wait...\n");

sleep(1);

}

//向共享内存中写入数据

strncpy(shared->data,argv[1],1024);

shared->is\_write = 1;//写完数据后，置1使可读

if(strncmp(shared->data,"end",3)==0)

{

running = 0;

}

}

if(shmdt(shm)==-1)//把共享内存从这块进程分离

{

perror("Fail to shmdt");

exit(EXIT\_SUCCESS);

}

sleep(2);

exit(EXIT\_SUCCESS);

}

if(pid==0)//父进程，用来读

{

printf("I is son process!\n");

shm = shmat(shm\_id,0,0);//将共享内存连接到当前进程的地址空间

if(shm==(void \*)-1)

{

perror("Fail to shmat");

exit(EXIT\_SUCCESS);

}

printf(" son Memory attached at %x\n",(int)shm);

shared = (struct shareMemory \*)shm;//设置共享内存

shared->is\_write = 0;

while(running) //读取共享内存中的数据

{

if(shared->is\_write!=0)

{

printf("YOu worte:%s\n",shared->data);

sleep(rand()%3);

shared->is\_write = 0;

if(strncmp(shared->data,"end",3)==0)//输入end退出循环

{

running = 0;

}

else//有其他进程在写数据，不能读

sleep(1);

}

}

if(shmdt(shm)==-1)//把共享内存从当前进程分离

{

perror("Fail to shmdt");

exit(EXIT\_SUCCESS);

}

if(shmctl(shm\_id,IPC\_RMID,0)==-1)//删除这块共享内存

{

perror("Fail to shmctl(IPC\_RMID)");

exit(EXIT\_SUCCESS);

}

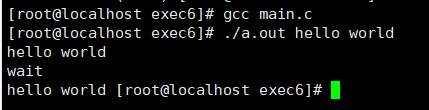
exit(EXIT\_SUCCESS);

}

return 0;

}

**运行测试结果：**



**memcpy和memmove内存段数据处理函数学习：**

memcpy和memmove（）都是C语言中的库函数，在头文件string.h中，作用是拷贝一定长度的内存的内容，原型分别如下：

void \*memcpy(void \*dst, const void \*src, size\_t count);

void \*memmove(void \*dst, const void \*src, size\_t count);

他们的作用是一样的，唯一的区别是，当内存发生局部重叠的时候，memmove保证拷贝的结果是正确的，memcpy不保证拷贝的结果的正确。

第一种情况下，拷贝重叠的区域不会出现问题，内容均可以正确的被拷贝。

第二种情况下，问题出现在 右边的两个字节，这两个字节的原来的内容首先就被覆盖了 ， 而且没有保存。所以接下来拷贝的时候，拷贝的是已经被覆盖的内容 ，显然这是有问题的。

实际上， memcpy只是memmove的一个子集 。

以上为学习部分资料，实践过程已经进行了相关函数使用。

（2）查阅相关资料，了解Linux多线程编程，并完成以下实验

1、编写程序实现如下功能：编写程序，在主线程中创建一个新线程。要求在新线程中输出运行信息，在结束时返回主线程。使用函数pthread\_exit（0）退出线程。再试试exit(0)退出线程，看看是什么结果。比较一下二者的不同。

**使用pthread\_exit(0):**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**#include<pthread.h>**

**void \*thread\_a(void \*args)**

**{**

**for(int i=0;i<3;i++)**

**{**

**printf("thread:running\n");**

**}**

**pthread\_exit(0);**

**}**

**int main(){**

**pthread\_t t1;**

**pthread\_create(&t1,NULL,thread\_a,NULL);**

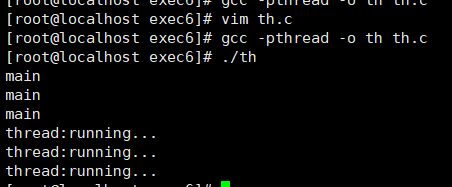
**for(int i=0;i<3;i++)**

**printf("main");**

**pthread\_join(t1,NULL);**

**return 0;**

**}**



**使用exit(0):**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**#include<pthread.h>**

**void \*thread\_a(void \*args)**

**{**

**for(int i=0;i<3;i++)**

**{**

**printf("thread:running\n");**

**}**

**exit(0);**

**}**

**int main(){**

**pthread\_t t1;**

**pthread\_create(&t1,NULL,thread\_a,NULL);**

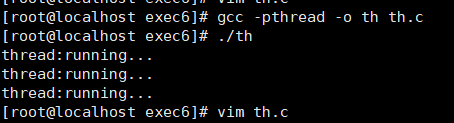
**for(int i=0;i<3;i++)**

**printf("main");**

**pthread\_join(t1,NULL);**

**return 0;**

**}**

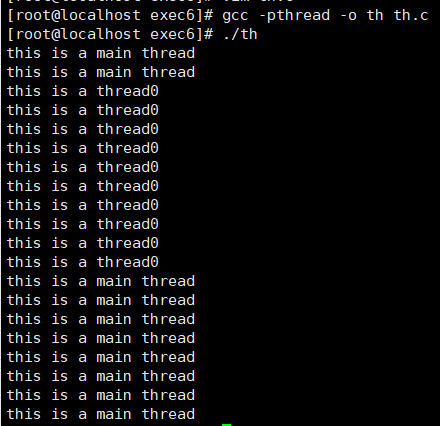


**pthread\_exit()用于线程退出，可以指定返回值，以便其他线程通过**

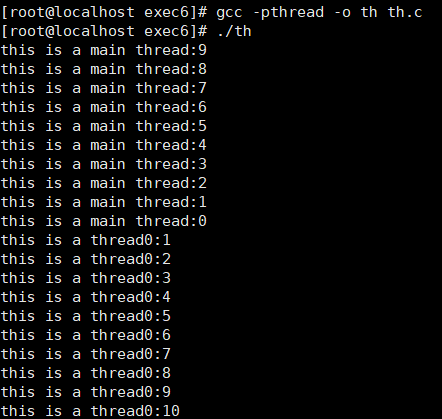
**exit（）是进程退出，如果在线程函数中调用exit，那该线程的进程也就挂了,会导致该线程所在进程的其他线程也挂掉，比较严重**

2、在两个线程之间实现交替执行输出。要求：例如主线程先输出“this is a thread0”,然后新线程输出“this is thread 2”一直交替直到结束。同时访问全局变量，修改变量的值，并打印看看输出结果。

**①仅输出字符串：**



**②修改全局变量：**



**③代码：**

**#include<stdlib.h>**

**#include<unistd.h>**

**#include<pthread.h>**

**int share = 10;**

**void \*thread\_a(void \*args)**

**{**

**for(int i=0;i<10;i++)**

**{**

**share++;**

**printf("this is a thread0:%d\n",share);**

**}**

**}**

**int main(){**

**pthread\_t t1;**

**pthread\_create(&t1,NULL,thread\_a,NULL);**

**for(int i=0;i<10;i++)**

**{**

**share--;**

**printf("this is a main thread:%d\n",share);**

**}**

**pthread\_join(t1,NULL);**

**return 0;**

**}**

3、编写程序实现使用信号量进行同步。编码实现输入字符串，统计每行的字符个数，以“end”结束输入。

**信号量学习资料：**

信号量Semaphore

定义：

有时被称为信号灯，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。

目的：

类似计数器，常用在多线程同步任务上，信号量可以在当前线程某个任务完成后，通知别的线程，再进行别的任务。

分类:

二值信号量：信号量的值只有0和1，这和互斥量很类似，若资源被锁住，信号量的值为0，若资源可用，则信号量的值为1；

计数信号量：信号量的值在0到一个大于1的限制值之间，该计数表示可用的资源的个数。

信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访问该信号量保护的共享资源，初始值为1就变成互斥锁Mutex，即同时只能有一个任务可以访问信号量保护的共享资源

函数使用：

首先需要include <semaphore.h>。

sem\_init

简述：创建信号量

第一个参数：指向的信号对象

第二个参数：控制信号量的类型，如果其值为0，就表示信号量是当前进程的局部信号量，否则信号量就可以在多个进程间共享

第三个参数：信号量sem的初始值

返回值：success为0，failure为-1

sem\_post

简述：信号量的值加1

第一个参数：信号量对象

返回值：success为0，failure为-1

sem\_wait

简述：信号量的值加-1

第一个参数：信号量对象

返回值：success为0，failure为-1

sem\_destroy

简述：用完记得销毁

第一个参数：信号量对象

返回值：success为0，failure为-1

**代码：**

**#include <pthread.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <semaphore.h>**

**#define SIZE 1024**

**char buffer[SIZE];**

**void \*thread\_function(void \*arg);**

**pthread\_mutex\_t mutex;**

**int main()**

**{**

**int res;**

**pthread\_t a\_thread;**

**void \*thread\_result;**

**res = pthread\_mutex\_init(&mutex, NULL);**

**if (res != 0)**

**{**

**perror("Mutex init failed!");**

**exit(EXIT\_FAILURE);**

**}**

**res = pthread\_create(&a\_thread, NULL, thread\_function, NULL);**

**if (res != 0)**

**{**

**perror("Thread create failed!");**

**exit(EXIT\_FAILURE);**

**}**

**printf("Input some sentences. Enter 'end' to finish\n");**

**while (1)**

**{**

**pthread\_mutex\_lock(&mutex);**

**scanf("%s", buffer);**

**pthread\_mutex\_unlock(&mutex);**

**if (strncmp("end", buffer, 3) == 0)**

**break;**

**sleep(1);**

**}**

**res = pthread\_join(a\_thread, &thread\_result);**

**if (res != 0)**

**{**

**perror("Thread join failed!");**

**exit(EXIT\_FAILURE);**

**}**

**printf("Thread joined\n");**

**pthread\_mutex\_destroy(&mutex);**

**exit(EXIT\_SUCCESS);**

**}**

**void \*thread\_function(void \*arg)**

**{**

**sleep(1);**

**while (1)**

**{**

**pthread\_mutex\_lock(&mutex);**

**printf("You input %d characters\n", strlen(buffer));**

**pthread\_mutex\_unlock(&mutex);**

**if (strncmp("end", buffer, 3) == 0)**

**break;**

**sleep(1);**

**}**

**}**

# 分析结果

实验四的第一部分中的静态库与动态库的实践以及系统调用函数的使用对我来说是比较陌生的知识领域，无法像之前的实践一样很快上手，而需要进行较长时间的学习和理解。本次实践过程中很快的完成了静态库与动态库相关内容。在解决系统调用函数的问题时，由于不怎么使用这些知识，对于这些函数较为陌生，尤其花了很多时间去学习理解fcntl函数。

实验四的第二部分主要研究了使用共享内存进行进程间通信；在进程中创建新线程、在一个进程中同步线程之间的数据访问、修改线程属性、在同一个进程中，从一个线程中控制另一个线程。这些方面对我来说同样也是比较陌生的，花了很多时间来学习有关共享内存以及线程的知识，庆幸的是，本次实验内容并不是很多，查阅相关资料并与同学们进行了讨论后，很快完成了这些比较简单的程序。

通过这四次的操作系统实验，我熟悉了Linux的一些基本操作，了解了操作系统中进程以及线程的相关知识，同时也对openEuler有了一些接触。实验课程是短暂的，但未来研究操作系统的路还很长，我在今后的学习中也会继续努力，把在操作系统课程中学到的知识运用到实践之中。