****

****

操作系统实验报告

学号：161640201

姓名：徐瑞涓

时间：2019.6.29

# **实验一：文件读写编程题目**

## myecho.c

【实验题目】

* myecho.c的功能与系统echo程序相同
* 接受命令行参数，并将参数打印出来，例子如下：

$ ./myecho x

x

$ ./myecho a b c

a b c

【实验思路】

1、通过main函数的参数获取命令行参数。

2、main函数参数为(int argc,char \*argv[]) ，其中：

1）argc代表参数的个数。

2）所有的参数存储在argv字符指针数组中。

3）可以通过argv[i]访问第i个命令行参数。

4）由于第0个代表可执行程序本身，所以从第1个开始打印。

3、使用while循环逐个打印命令行参数。从而实现echo的效果。

【实验代码】

#include<unistd.h>  
#include<stdio.h>  
int main(int argc,char \*argv[])  
{  
 int i=1;  
 while(i<argc)  
 {  
 printf("%s ",argv[i]);  
 i++;  
 }  
 printf("\n");  
 return 0;  
}

## mycat.c

【实验题目】

* mycat.c的功能与系统cat程序相同
* mycat将指定的文件内容输出到屏幕，例子如下：
* 要求使用系统调用open/read/write/close实现

$ cat /etc/passwd

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin

...

$ ./mycat /etc/passwd

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin

...

【实验思路】

1. 使用系统调用open/read/close实现
2. 使用main函数的参数依次获取文件名。
3. 对于每一个文件执行如下操作

1）定义文件描述符fd。

2）使用open系统调用打开文件。同时判断文件是否存在。

3）若文件存在，使用read系统调用读取文件内容。

4）同时考虑文件大小的不确定性，使用大小为256字节的数组存储读取的内容。然后循环读取文件内容。

5）直到文件结束。

6）此时使用close系统调用关闭文件描述符。

【实验代码】  
#include<stdio.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<unistd.h>  
  
int main(int argc,char \*argv[])  
{  
 int i=1;  
 while(i<argc)  
 {  
 int fd;  
 fd = open(argv[i],O\_RDWR);  
 if(fd==-1)  
 printf("%s Not Find\n",argv[i]);  
 else//opening is ok.  
 {  
 char buf[257];  
 int count = read(fd, buf, 256);  
 while(count>0)  
 {  
 printf("%s",buf);  
 int j=0;  
 while(j<257) buf[j++] = '\0';  
 count = read(fd, buf, 256);  
 }  
 close(fd);  
  
 }  
  
 i++;  
  
 }  
 return 0;  
}

## mycp.c

【实验题目】

* mycp.c的功能与系统cp程序相同
* 将源文件复制到目标文件，例子如下：
* 要求使用系统调用open/read/write/close实现

$ cat /etc/passwd

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin

...

$ ./mycp /etc/passwd passwd.bak

$ cat passwd.bak

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin

...

【实验思路】

1. 使用系统调用open/read/write/close实现
2. 由于cp命令后跟的是源文件和目标文件，所以需要使用argc判断命令行参数的个数。只有数量是3时，才会继续执行。
3. 定义两个文件描述符，使用open使它们分别指向源文件和目标文件。
   * 目标文件不存在需要创建。
4. 判断这两个文件是否打开成功。
5. 如果打开成功：

1）定义一个257字节大小的数组。

2）循环从源文件中使用read读取256个字节，然后使用write系统调用写入到目标文件中

3）直到源文件结束。

1. 使用close关闭上述文件

【实验代码】

#include<stdio.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<unistd.h>  
  
int main(int argc,char \*argv[])  
{  
 if(3 == argc)  
 {  
 int fd1, fd2;  
 mode\_t mode = 0777;  
 fd1 = open(argv[1],O\_RDWR, mode);  
 fd2 = open(argv[2],O\_RDWR | O\_CREAT, mode);  
   
 if(fd1==-1)  
 printf("%s Not Find\n",argv[1]);  
 else if(fd2==-1)  
 printf("Error in create %s\n",argv[2]);  
 else//opening is ok.  
 {  
 char buf[257];  
 int count = read(fd1, buf, 256);  
 while(count>0)  
 {  
 write(fd2, buf, 256);  
 int j=0;  
 while(j<257) buf[j++] = '\0';  
 count = read(fd1, buf, 256);  
 }  
 close(fd1);  
 close(fd2);  
 }  
 }  
 else  
 printf("please input two filename\n");  
  
 return 0;  
}

# 实验二：多进程题目

## mysys.c

## 【实验题目】

## 实现函数mysys，用于执行一个系统命令，要求如下：

* mysys的功能与系统函数system相同，要求用进程管理相关系统调用自己实现一遍
* 使用fork/exec/wait系统调用实现mysys
* 不能通过调用系统函数system实现mysys
* 测试程序

#include <stdio.h>

void mysys(char \*command)

{

实现该函数，该函数执行一条命令，并等待该命令执行结束

}

int main()

{

printf("--------------------------------------------------\n")；

mysys("echo HELLO WORLD");

printf("--------------------------------------------------\n");

mysys("ls /");

printf("--------------------------------------------------\n");

return 0;

}

测试程序的输出结果

--------------------------------------------------

HELLO WORLD

--------------------------------------------------

bin core home lib mnt root snap tmp vmlinuz

boot dev initrd.img lost+found opt run srv usr vmlinuz.old

cdrom etc initrd.img.old media proc sbin sys var

--------------------------------------------------

【实验思路】

1. 使用fork/exec/wait系统调用实现mysys
2. 定义mysys函数。函数实现细节如下：
   1. 首先判断当前的命令是否为空，若非空则继续。
   2. 使用fork创建一个子进程。
   3. 在子进程中执行当前命令。
   4. 使用wait等待子进程结束。
3. 使用两个测试命令测试mysys的执行情况。

【实验代码】  
#include<stdio.h>  
#include<unistd.h>  
  
void mysys(char \*cmd)  
{  
 if(cmd != NULL)  
 {  
 pid\_t pid;  
 pid = fork();  
 if(pid==0)  
 execl("/bin/sh","sh", "-c", cmd, NULL);  
 wait();  
 }  
}  
int main(int argc, char\* argv[])  
{  
 printf("------------------------------------------------------------\n");  
 mysys("echo HELLO WORLD");  
 printf("------------------------------------------------------------\n");  
 mysys("ls /");  
 printf("------------------------------------------------------------\n");  
 return 0;  
}

## sh1.c

## 【实验题目】

## 该程序读取用户输入的命令，调用函数mysys(上一个作业)执行用户的命令，示例如下

# 编译sh1.c

$ cc -o sh1 sh1.c

# 执行sh1

$ ./sh

# sh1打印提示符>，同时读取用户输入的命令echo，并执行输出结果

> echo a b c

a b c

# sh1打印提示符>，同时读取用户输入的命令cat，并执行输出结果

> cat /etc/passwd

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin

* 请考虑如何实现内置命令cd、pwd、exit

【实验思路】

1. 使用while循环打印提示符>，并使用fgets接受用户输入的命令。
2. 使用mysys函数创建子进程执行命令。
3. mysys函数实现细节如下：
   1. 首先判断当前的命令是否为空，若非空则继续。
   2. 使用fork创建一个子进程。
   3. 在子进程中执行当前命令。
   4. 使用wait等待子进程结束。

【实验代码】  
#include<stdio.h>  
#include<unistd.h>  
  
void mysys(char \*cmd)  
{  
 if(cmd != NULL)  
 {  
 pid\_t pid;  
 pid = fork();  
 if(pid==0)  
 execl("/bin/sh","sh", "-c", cmd, NULL);  
 wait();  
 }  
}  
int main(int argc, char\* argv[])  
{  
 char cmd[100];  
 do{  
 printf("> ");  
 fgets(cmd, 99, stdin);  
 mysys(cmd);  
 }while(1);  
  
 return 0;  
}

## sh2.c

【实验题目】

## 实现shell程序，要求在第1版的基础上，添加如下功能

* 实现文件重定向

# 执行sh2

$ ./sh2

# 执行命令echo，并将输出保存到文件log中

> echo hello >log

# 打印cat命令的输出结果

> cat log

hello

【实验思路】

1. 使用while循环打印提示符>，并使用fgets接受用户输入的命令。
2. 使用mysys函数创建子进程执行命令。
3. mysys函数实现细节如下：

1）首先判断当前的命令是否为空，若非空则继续。

2）使用fork创建一个子进程。

3）使用while遍历命令，查看是否使用输出重定向符号>

4）若存在文件重定向，则记录出现的位置。

5）因为重定向符号后是输出文件的文件名，所以定义一个文件描述符指向该输出文件。并使用dup2将该描述符重定向到标准输出。这样的效果就是如果有数据要显示到标准输出，那么它就会写入该文件。

6）使用fork创建一个子进程。

7）在子进程中执行重定向符号之前的命令。该命令产生的输出都会写入文件中。

8）主进程使用wait等待子进程结束。

9）若不存在文件重定向，则直接执行该指令。

【实验代码】  
#include<stdio.h>  
#include<unistd.h>  
  
void mysys(char \*cmd)  
{  
 if(cmd != NULL)  
 {  
 pid\_t pid;  
 pid = fork();  
 if(pid==0){  
 int len = strlen(cmd);  
 int xiaoyu=0, shuxian=0;  
 while(xiaoyu<len)  
 if(cmd[xiaoyu++]=='>')  
 break;  
  
 if(xiaoyu!=len){  
 cmd[xiaoyu-1] = 0;  
 cmd[len-1] = 0;  
 int fd = open(&cmd[xiaoyu], O\_CREAT|O\_WRONLY, 0666);  
 dup2(fd, 1);  
 pid = fork();  
 if(pid==0){  
 mysys(cmd);  
 exit(0);  
 }  
 wait(NULL);  
 }  
 else{  
 execlp("sh", "sh", "-c", cmd, NULL);  
 }  
 exit(0);  
  
 }  
 wait(NULL);  
 }  
}  
  
  
int main(int argc, char\* argv[])  
{  
 char cmd[100];  
 do{  
 printf("> ");  
 fgets(cmd, 99, stdin);  
 mysys(cmd);  
 }while(1);  
  
 return 0;  
}

## sh3.c

【实验题目】

实现shell程序，要求在第2版的基础上，添加如下功能

* 实现管道

# 执行sh3

$ ./sh3

# 执行命令cat和wc，使用管道连接cat和wc

> cat /etc/passwd | wc -l

* 考虑如何实现管道和文件重定向

$ cat input.txt

3

2

1

3

2

1

$ cat <input.txt | sort | uniq | cat >output.txt

$ cat output.txt

1

2

3

【实验思路】

1. 使用while循环打印提示符>，并使用fgets接受用户输入的命令。
2. 使用mysys函数创建子进程执行命令。
3. mysys函数实现细节如下：
   1. 首先判断当前的命令是否为空，若非空则继续。
   2. 使用fork创建一个子进程。
   3. 使用while遍历命令，查看是否使用输出重定向符号&gt或管道符号|。
   4. 若存在管道符号，则记录出现的位置。
   5. 定义一个pipe并初始化。
   6. 使用fork创建一个子进程。
   7. 在子进程中重定向管道的写端到标准输出。代表该命令产生的输出都会通过管道写出去。
   8. 子进程重定向标准输入到管道的读端。
   9. 主进程使用wait等待子进程结束。
   10. 若存在文件重定向，则记录出现的位置。
   11. 因为重定向符号后是输出文件的文件名，所以定义一个文件描述符指向该输出文件。并使用dup2将该描述符重定向到标准输出。这样的效果就是如果有数据要显示到标准输出，那么它就会写入该文件。
   12. 使用fork创建一个子进程。
   13. 在子进程中执行重定向符号之前的命令。该命令产生的输出都会写入文件中。
   14. 主进程使用wait等待子进程结束。
   15. 若不存在文件重定向和管道，则直接执行该指令。

【实验代码】  
#include<stdio.h>  
#include<unistd.h>  
#include<string.h>  
#include<sys/wait.h>  
#include<fcntl.h>  
#include<stdlib.h>  
void mysys(char \*cmd)  
{  
 if(cmd != NULL)  
 {  
 pid\_t pid;  
 pid = fork();  
 if(pid==0){  
 int len = strlen(cmd);  
 int xiaoyu=0, shuxian=0;  
 while(xiaoyu<len)  
 if(cmd[xiaoyu++]=='>')  
 break;  
 while(shuxian<len)  
 if(cmd[shuxian++]=='|')  
 break;  
  
 if(xiaoyu!=len){  
 cmd[xiaoyu-1] = 0;  
 cmd[len-1] = 0;  
 int fd = open(&cmd[xiaoyu], O\_CREAT|O\_WRONLY, 0666);  
 dup2(fd, 1);  
 pid = fork();  
 if(pid==0){  
 mysys(cmd);  
 exit(0);  
 }  
 wait(NULL);  
 }  
 else if(shuxian!=len){  
 cmd[shuxian-1] = 0;  
 cmd[len-1] = 0;  
  
 int fd[2];  
 pipe(fd);  
  
 pid = fork();  
 if(pid==0){  
 dup2(fd[1],1);  
 close(fd[0]);  
 close(fd[1]);  
 execlp("sh", "sh", "-c", cmd, NULL);  
 exit(0);  
 }  
 dup2(fd[0], 0);  
 close(fd[0]);  
 close(fd[1]);  
 execlp("sh", "sh", "-c", &cmd[shuxian], NULL);  
 wait(NULL);  
 }  
 else{  
 execlp("sh", "sh", "-c", cmd, NULL);  
 }  
 exit(0);  
  
 }  
 wait(NULL);  
 }  
}  
  
int main(int argc, char\* argv[])  
{  
 char cmd[100];  
 do{  
 printf("> ");  
 fgets(cmd, 99, stdin);  
 mysys(cmd);  
 }while(1);  
  
 return 0;  
}

# 实验三：多线程题目

## pi1.c

## 【实验题目】

## 使用2个线程根据莱布尼兹级数计算PI

* 莱布尼兹级数公式: 1 - 1/3 + 1/5 - 1/7 + 1/9 - ... = PI/4
* 主线程创建1个辅助线程
* 主线程计算级数的前半部分
* 辅助线程计算级数的后半部分
* 主线程等待辅助线程运行結束后,将前半部分和后半部分相加

【实验思路】

1. 在main函数主线程使用pthread\_create创建1个辅助线程
2. 并传入该线程执行的函数。
3. 该函数通过循环计算级数的后半部分
4. 并通过return返回结果
5. 主线程计算级数的前半部分
6. 并使用pthread\_join等待辅助线程运行結束,使用sum2获取线程的结果。
7. 最后在主线程中将前半部分和后半部分相加,输出结果

【实验代码】  
#include<stdio.h>  
#include<pthread.h>  
#include<stdlib.h>  
  
#define SIZE 10000 //the size of all numbers  
  
void \*another(void \*arg){  
 double \*sum = (double\*)malloc(sizeof(double));  
 \*sum = 0;  
 short sign = ((SIZE/2)%2==0)?1:-1;  
 int i=0;  
 while(i<SIZE/2){  
 \*sum += sign\*(double)1/(2\*i+1+SIZE);  
 sign = - sign;  
 i++;  
 }  
  
 return (void \*)sum;  
}  
  
int main(){  
  
 pthread\_t pid;  
 pthread\_create(&pid, NULL, &another, NULL);  
  
 double sum1=0;  
 short sign = 1;  
 int i=0;  
 while(i<SIZE/2){  
 sum1 += sign\*(double)1/(2\*i+1);  
 sign = - sign;  
 i++;  
 }  
 double \*sum2 = (double \*)malloc(sizeof(double));  
 pthread\_join(pid, (void \*\*)&sum2);  
  
 printf("sum is %f.\n", sum1+\*sum2);  
  
 return 0;  
}

## pi2.c

## 【实验题目】

## 使用N个线程根据莱布尼兹级数计算PI

* 与上一题类似，但本题更加通用化，能适应N个核心，需要使用线程参数来实现
* 主线程创建N个辅助线程
* 每个辅助线程计算一部分任务，并将结果返回
* 主线程等待N个辅助线程运行结束，将所有辅助线程的结果累加

【实验思路】

子线程：

1. 通过类型转化，获取函数参数中所对应的索引信息。
2. 通过该信息得知从哪个数进行计算。
3. 然后循环计算累加值，最后返回。

主线程：

1. 创建多个线程，并通过参数传入各个进程执行的函数需要使用的位置索引。
2. 等待所有进程结束，并通过pthread\_join获取各个线程的返回值。
3. 累加所有的返回值，完成计算。

【实验代码】  
#include<stdio.h>  
#include<pthread.h>  
#include<stdlib.h>  
  
#define SIZE 10000 //the size of all numbers  
#define PTHREAD\_SIZE 100 //the size of all pthreads  
  
void \*another(void \*arg){  
 double \*sum = (double\*)malloc(sizeof(double));  
 int start = \*(int \*)arg;  
 \*sum = 0;  
 short sign = ((start/2)%2==0)?1:-1;  
 int i=start;  
 while(i<(start + SIZE/PTHREAD\_SIZE)){  
 \*sum += sign\*(double)1/(2\*i+1);  
 sign = - sign;  
 i++;  
 }  
  
 return (void \*)sum;  
}  
  
int main(){  
  
 pthread\_t pidArr[PTHREAD\_SIZE];  
 int start[PTHREAD\_SIZE];  
  
 int cur\_start = 0;  
 int i=0;  
 while(i<PTHREAD\_SIZE){  
  
 start[i] = cur\_start;  
 void \*arg = (void \*)(start + i);  
 pthread\_create(&pidArr[i], NULL, &another, arg);  
 cur\_start += SIZE/PTHREAD\_SIZE;  
 i++;  
 }  
  
 double allSum = 0;  
 i=0;  
  
 while(i<PTHREAD\_SIZE){  
 double \*sum;  
 pthread\_join(pidArr[i], (void \*\*)&sum);  
 allSum += \*sum;  
 free(sum);  
 i++;  
 }  
  
 printf("sum is %f.\n", allSum);  
  
 return 0;  
}

## sort.c

## 【实验题目】

## 多线程排序

* 主线程创建一个辅助线程
* 主线程使用选择排序算法对数组的前半部分排序
* 辅助线程使用选择排序算法对数组的后半部分排序
* 主线程等待辅助线程运行結束后,使用归并排序算法归并数组的前半部分和后半部分

【实验思路】

子线程：

1. 通过函数参数得到数组的地址。
2. 然后计算数组的长度，从而确定自己需要排序的区间。
3. 使用排序算法进行排序。

主线程：

1. 随机产生一百个数
2. 创建一个子线程，并将数组的地址作为参数传过去。
3. 使用选择排序算法对数组的前半部分进行排序。
4. 使用pthread\_join等待进程结束。
5. 然后使用归并算法对数组的全部元素进行归并。
6. 完成排序。

【实验代码】  
#include<stdio.h>  
#include<pthread.h>  
#include<stdlib.h>  
  
#define SIZE 20 //the size of all numbers  
  
void \*another(void \*arg){  
 int \*arr = (int \*)arg;  
 int len = SIZE - SIZE/2;  
  
 int min, temp;  
 int i=0,j;  
 while(i<len){  
 min = i;  
 j = i+1;  
 while(j<len){  
 if(arr[j]<arr[min])  
 min = j;  
 j++;  
 }  
 temp = arr[i];  
 arr[i] = arr[min];  
 arr[min] = temp;  
 i++;  
 }  
  
 return NULL;  
}  
  
int main(){  
  
 int arr[SIZE];  
 int i=0,j;  
 while(i<SIZE){  
 arr[i++] = rand()%100;  
 }  
  
 void \*arg = (void \*)(arr + SIZE/2);  
  
 pthread\_t pid;  
 pthread\_create(&pid, NULL, &another, arg);  
  
 int min, temp;  
 i=0;  
 while(i<SIZE/2){  
 min = i;  
 j = i+1;  
 while(j<SIZE/2){  
 if(arr[j]<arr[min])  
 min = j;  
 j++;  
 }  
 temp = arr[i];  
 arr[i] = arr[min];  
 arr[min] = temp;  
 i++;  
 }  
  
 pthread\_join(pid, NULL);  
   
 i=0,j=SIZE/2;  
 while(i<SIZE/2&&j<SIZE){  
 if(arr[i]<arr[j]){  
 printf("%d ", arr[i]);  
 i++;  
 }  
 else{  
 printf("%d ", arr[j]);  
 j++;  
 }  
 }  
  
 while(i<SIZE/2){  
 printf("%d ", arr[i++]);  
 }  
 while(j<SIZE){  
 printf("%d ", arr[j++]);  
 }  
 printf("\n");  
  
  
 return 0;  
}

## pc1.c

## 【实验题目】

## 使用条件变量解决生产者、计算者、消费者问题

* 系统中有3个线程：生产者、计算者、消费者
* 系统中有2个容量为4的缓冲区：buffer1、buffer2
* 生产者生产'a'、'b'、'c'、‘d'、'e'、'f'、'g'、'h'八个字符，放入到buffer1
* 计算者从buffer1取出字符，将小写字符转换为大写字符，放入到buffer2
* 消费者从buffer2取出字符，将其打印到屏幕上

【实验思路】

1. 主线程作为生产者，主线程创建两个子线程分别作为计算者和消费者。
2. 定义四个条件变量。意义分别是：是否有空位用来生产，是否有空位用来存储计算结果，是否有生产过的用来计算，是否有计算好的用来消费。

生产者：

1. 判断是否有空位用来生产。
2. 如果没有，则阻塞。
3. 如果有，产生结果，并存储到缓冲区1。
4. 同时signal条件变量wait\_fullOfCal。

计算者：

1. 判断是否有需要计算的资源。若没有，则阻塞。
2. 判断是否有存储计算结果的位置。若没有，则阻塞。
3. 如果上述条件满足，则计算。
4. 结束时signal条件变量wait\_fullOfCon和wait\_emptyOfPro

消费者：

1. 判断是否有可以消费的资源。若没有，则阻塞。
2. 如果上述条件满足，则打印（消费）。
3. 结束时signal条件变量wait\_emptyOfCal。

【实验代码】  
#include<stdio.h>  
#include<pthread.h>  
#include<stdlib.h>  
  
#define SIZE 4  
#define PRODUCE\_COUNT 8  
  
char buffer1[SIZE];  
char buffer2[SIZE];  
int in=0;  
int cal=0;  
int out=0;  
  
int emptyOfCon(){  
 return cal==out;  
}  
  
int emptyOfCal(){  
 return cal==in;  
}  
  
int fullOfCal(){  
 return (cal+1)%SIZE==out;  
}  
  
int fullOfPro(){  
 return (in+1)%SIZE==cal;  
}  
  
char get(){  
 char n = buffer2[out];  
 out = (out+1)%SIZE;  
 return n;  
}  
  
void put(char n){  
 buffer1[in] = n;  
 in = (in+1)%SIZE;  
}  
  
void calculate(){  
 buffer2[cal] = buffer1[cal] + 'A' - 'a';  
 cal = (cal+1)%SIZE;  
}  
  
pthread\_mutex\_t mutex;  
pthread\_cond\_t wait\_emptyOfPro;  
pthread\_cond\_t wait\_fullOfCal;  
pthread\_cond\_t wait\_emptyOfCal;  
pthread\_cond\_t wait\_fullOfCon;  
  
void \*Produce(void \*arg){  
 int i=0;  
 while(i<PRODUCE\_COUNT){  
 pthread\_mutex\_lock(&mutex);  
 while(fullOfPro())  
 pthread\_cond\_wait(&wait\_emptyOfPro, &mutex);  
  
 put(i+'a');  
 printf("produce=>%c\n", i+'a');  
  
 pthread\_cond\_signal(&wait\_fullOfCal);  
 pthread\_mutex\_unlock(&mutex);  
 i++;  
 }  
}  
  
void \*Calculate(void \*arg){  
 int i=0;  
 while(i<PRODUCE\_COUNT){  
 pthread\_mutex\_lock(&mutex);  
 while(emptyOfCal())  
 pthread\_cond\_wait(&wait\_fullOfCal, &mutex);  
 while(fullOfCal())  
 pthread\_cond\_wait(&wait\_emptyOfCal, &mutex);  
  
 calculate();  
 printf("\tcalculate<=\n");  
  
 pthread\_cond\_signal(&wait\_fullOfCon);  
 pthread\_cond\_signal(&wait\_emptyOfPro);  
 pthread\_mutex\_unlock(&mutex);  
 i++;  
 }  
}  
  
void \*Consume(void \*arg){  
 int i=0;  
 while(i<PRODUCE\_COUNT){  
 pthread\_mutex\_lock(&mutex);  
 while(emptyOfCon())  
 pthread\_cond\_wait(&wait\_fullOfCon, &mutex);  
  
 printf("\t\tconsume=>%c\n", get());  
  
 pthread\_cond\_signal(&wait\_emptyOfCal);  
 pthread\_mutex\_unlock(&mutex);  
 i++;  
 }  
}  
  
int main(){  
  
 pthread\_t consumePid;  
 pthread\_t calculatePid;  
  
 pthread\_mutex\_init(&mutex, NULL);  
 pthread\_cond\_init(&wait\_emptyOfPro, NULL);  
 pthread\_cond\_init(&wait\_emptyOfCal, NULL);  
 pthread\_cond\_init(&wait\_fullOfCal, NULL);  
 pthread\_cond\_init(&wait\_fullOfCon, NULL);  
  
 pthread\_create(&consumePid, NULL, &Consume, NULL);  
 pthread\_create(&calculatePid, NULL, &Calculate, NULL);  
  
 Produce(NULL);  
  
 pthread\_join(consumePid, NULL);  
 pthread\_join(calculatePid, NULL);  
  
 pthread\_mutex\_destroy(&mutex);  
  
 return 0;  
}

## 5、pc2.c

## 【实验题目】

## 使用信号量解决生产者、计算者、消费者问题

* 功能和前面的实验相同，使用信号量解决

【实验思路】

1. 定义五个信号量， mutex\_sema emptyOfPro\_sema fullOfCal\_sema emptyOfCal\_sema fullOfCon\_sema
2. 后四个的含义非别是可以生产的空位的个数，可以计算的个数，可以存储计算结果的个数，可以消费的资源个数。
3. 主线程作为生产者，主线程创建两个子线程分别作为计算者和消费者。

生产者：

1. 判断是否有空位用来生产。
2. 如果没有，则阻塞。
3. 如果有，emptyOfPro\_sema的值减一，产生结果，并存储到缓冲区1。
4. 同时释放信号量fullOfCal\_sema

计算者：

1. 判断是否有需要计算的资源。若没有，则阻塞。
2. 判断是否有存储计算结果的位置。若没有，则阻塞。
3. 如果上述条件满足，fullOfCal\_sema和emptyOfCal\_sema的值减一，进行计算。
4. 结束时释放信号量emptyOfPro\_sema和fullOfCon\_sema

消费者：

1. 判断是否有可以消费的资源。若没有，则阻塞。
2. 如果上述条件满足，fullOfCon\_sema的值减一，则打印（消费）。
3. 结束时释放信号量emptyOfCal\_sema

【实验代码】  
#include<stdio.h>  
#include<pthread.h>  
#include<stdlib.h>  
  
#define SIZE 4  
#define PRODUCE\_COUNT 8  
  
char buffer1[SIZE];  
char buffer2[SIZE];  
int in=0;  
int cal=0;  
int out=0;  
  
  
char get(){  
 char n = buffer2[out];  
 out = (out+1)%SIZE;  
 return n;  
}  
  
void put(char n){  
 buffer1[in] = n;  
 in = (in+1)%SIZE;  
}  
  
void calculate(){  
 buffer2[cal] = buffer1[cal] + 'A' - 'a';  
 cal = (cal+1)%SIZE;  
}  
  
typedef struct{  
 int value;  
 pthread\_mutex\_t mutex;  
 pthread\_cond\_t cond;  
}sema\_t;  
  
void sema\_init(sema\_t \*sema, int value){  
 sema->value = value;  
 pthread\_mutex\_init(&sema->mutex, NULL);  
 pthread\_cond\_init(&sema->cond, NULL);  
}  
  
void sema\_wait(sema\_t \*sema){  
 pthread\_mutex\_lock(&sema->mutex);  
 while(sema->value <= 0)  
 pthread\_cond\_wait(&sema->cond, &sema->mutex);  
  
 sema->value--;  
 pthread\_mutex\_unlock(&sema->mutex);  
}  
  
void sema\_signal(sema\_t \*sema){  
 pthread\_mutex\_lock(&sema->mutex);  
 ++sema->value;  
 pthread\_cond\_signal(&sema->cond);  
 pthread\_mutex\_unlock(&sema->mutex);  
}  
  
sema\_t mutex\_sema;  
sema\_t emptyOfPro\_sema;  
sema\_t fullOfCal\_sema;  
sema\_t emptyOfCal\_sema;  
sema\_t fullOfCon\_sema;  
  
void \*Produce(void \*arg){  
 int i=0;  
 while(i<PRODUCE\_COUNT){  
 sema\_wait(&emptyOfPro\_sema);  
 sema\_wait(&mutex\_sema);  
  
 put(i+'a');  
 printf("produce=>%c\n", i+'a');  
  
 sema\_signal(&mutex\_sema);  
 sema\_signal(&fullOfCal\_sema);  
 i++;  
 }  
}  
  
void \*Calculate(void \*arg){  
 int i=0;  
 while(i<PRODUCE\_COUNT){  
 sema\_wait(&fullOfCal\_sema);  
 sema\_wait(&emptyOfCal\_sema);  
 sema\_wait(&mutex\_sema);  
  
 calculate();  
 printf("\tcalculate<=\n");  
  
 sema\_signal(&mutex\_sema);  
 sema\_signal(&emptyOfPro\_sema);  
 sema\_signal(&fullOfCon\_sema);  
 i++;  
 }  
}  
  
void \*Consume(void \*arg){  
 int i=0;  
 while(i<PRODUCE\_COUNT){  
 sema\_wait(&fullOfCon\_sema);  
 sema\_wait(&mutex\_sema);  
  
 printf("\t\tconsume=>%c\n", get());  
  
 sema\_signal(&mutex\_sema);  
 sema\_signal(&emptyOfCal\_sema);  
 i++;  
 }  
}  
  
int main(){  
  
 pthread\_t consumePid;  
 pthread\_t calculatePid;  
  
 sema\_init(&mutex\_sema, 1);  
 sema\_init(&emptyOfPro\_sema,SIZE-1);  
 sema\_init(&fullOfCal\_sema,0);  
 sema\_init(&emptyOfCal\_sema,SIZE-1);  
 sema\_init(&fullOfCon\_sema,0);  
  
 pthread\_create(&consumePid, NULL, &Consume, NULL);  
 pthread\_create(&calculatePid, NULL, &Calculate, NULL);  
  
 Produce(NULL);  
  
 pthread\_join(consumePid, NULL);  
 pthread\_join(calculatePid, NULL);  
  
 return 0;  
}

## ring.c

## 【实验题目】

## 创建N个线程，它们构成一个环

* 创建N个线程：T1、T2、T3、… TN
* T1向T2发送整数1
* T2收到后将整数加1
* T2向T3发送整数2
* T3收到后将整数加1
* T3向T4发送整数3
* …
* TN收到后将整数加1
* TN向T1发送整数N

【实验思路】

1. 主线程定义SIZE个信号量。
2. 初始第一个信号量为1，其余为0
3. 创建SIZE个线程，并通过函数参数告诉每个线程自己的身份ID

对于每个线程：

1. 通过函数参数获取自己的ID。
2. 执行时wait第ID个信号量，然后输出当前的值并累加。
3. 结束释放第ID个的后一个信号量。
4. 从而实现环形轮流并行传递足球。

【实验代码】  
#include<stdio.h>  
#include<pthread.h>  
#include<stdlib.h>  
  
#define SIZE 4  
#define PRODUCE\_COUNT 4  
int Number = 0;  
  
typedef struct{  
 int value;  
 pthread\_mutex\_t mutex;  
 pthread\_cond\_t cond;  
}sema\_t;  
  
void sema\_init(sema\_t \*sema, int value){  
 sema->value = value;  
 pthread\_mutex\_init(&sema->mutex, NULL);  
 pthread\_cond\_init(&sema->cond, NULL);  
}  
  
void sema\_wait(sema\_t \*sema){  
 pthread\_mutex\_lock(&sema->mutex);  
 while(sema->value <= 0)  
 pthread\_cond\_wait(&sema->cond, &sema->mutex);  
  
 sema->value--;  
 pthread\_mutex\_unlock(&sema->mutex);  
}  
  
void sema\_signal(sema\_t \*sema){  
 pthread\_mutex\_lock(&sema->mutex);  
 ++sema->value;  
 pthread\_cond\_signal(&sema->cond);  
 pthread\_mutex\_unlock(&sema->mutex);  
}  
  
sema\_t mutex\_sema;  
sema\_t semaArr[SIZE];  
  
void \*Produce(void \*arg){  
 int curID = \*(int \*)arg;  
 int i=0;  
  
 while(i<PRODUCE\_COUNT){  
 sema\_wait(&semaArr[curID]);  
 sema\_wait(&mutex\_sema);  
  
 printf("ring(%d) => %d\n", curID, Number++);  
  
 sema\_signal(&mutex\_sema);  
 sema\_signal(&semaArr[(curID+1)%SIZE]);  
 i++;  
 }  
}  
  
  
int main(){  
  
 pthread\_t pid[SIZE];  
  
 sema\_init(&mutex\_sema, 1);  
 sema\_init(&semaArr[0],1);  
 int i=1;  
 while(i<SIZE){  
 sema\_init(&semaArr[i], 0);  
 i++;  
 }  
  
 int ID[SIZE];  
 i=0;  
 while(i<SIZE){  
 ID[i] = i;  
 pthread\_create(&pid[i], NULL, &Produce, &ID[i]);  
 i++;  
 }  
  
 i=0;  
 while(i<SIZE){  
 pthread\_join(pid[i++], NULL);  
 }  
  
 return 0;  
}

四、 实验总结

在操作系统的上机实验中，我通过学习和使用很多函数，实现了一些简单的操作系统题目，在这个过程中，课堂理论知识和上机动手实践相结合，两者相辅相成，让我对操作系统有了更加清晰的认识，当然我们现在了解的只是基础，要想真正自己写出操作系统还需要长期的学习和钻研。真挚感谢老师本学期的指导与帮助！