

操作系统实验报告

1. 文件读写编程题目

1.1 myecho.c

- 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：

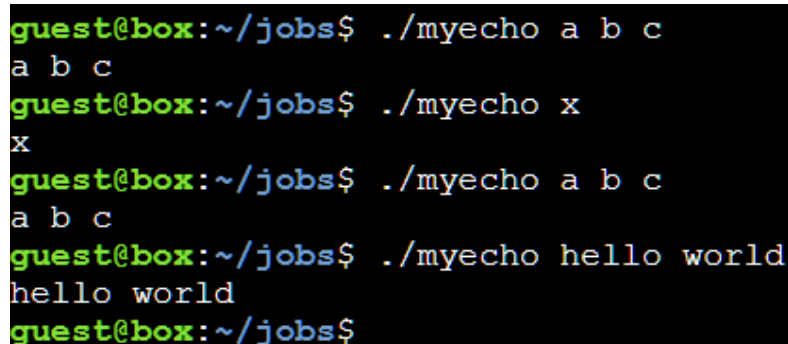
```
$ ./myecho x
x
$ ./myecho a b c
a b c
```

1.1.1 myecho 实验代码

```
#include<stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;
    for(i = 1; i < argc; i++)
        printf("%s ",argv[i]);
    printf("\n");
    return 0;
}
```

1.1.2 myecho 实验结果

A terminal window with a black background and green text. It shows the execution of the myecho program with various arguments. The prompt is 'guest@box:~/jobs\$'. The first command is './myecho a b c' and the output is 'a b c'. The second command is './myecho x' and the output is 'x'. The third command is './myecho a b c' and the output is 'a b c'. The fourth command is './myecho hello world' and the output is 'hello world'. The prompt returns after each command.

```
guest@box:~/jobs$ ./myecho a b c
a b c
guest@box:~/jobs$ ./myecho x
x
guest@box:~/jobs$ ./myecho a b c
a b c
guest@box:~/jobs$ ./myecho hello world
hello world
guest@box:~/jobs$
```

1.1.3 myecho 实验思路

1.实验预备知识

(1) main 函数的参数有 argc 和 argv 两个参数

(2) int argc(arguments count)

argc 表示运行程序传送给 main 函数的命令行参数总个数，包括可执行程序名，其中当 argc=1 时表示只有一个程序名

称，此时存储在 `argv[0]` 中

(3) `char *argv[]`(arguments value/vecotr)

`argv` 是一个字符串数组，用来存放指向字符串参数的指针数组，每个元素只想一个参数，空格分割参数，其长度为 `argc`。数组下标从 0 开始，`argv[argc] = NULL`。

2.实验思路 使用 `main` 函数的 `argv` 与 `argc` 传递参数，命令行输入的字符串会被分割为字符串数组。用 `argc` 作为循环变量，输出 `argv` 字符串数组中的值，注意用空格分隔即可。需要注意的是：`argv[0]`中存储的是程序名称，这里不应该被输出，所以下标应从 1 开始。

1.2 mycat.c

- 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
...
```

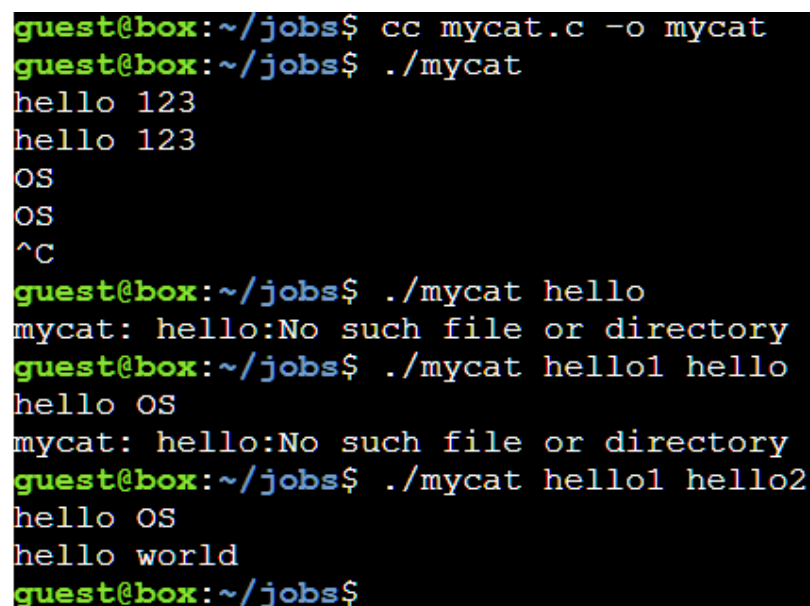
1.2.1 mycat 实验代码

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
    int fd;
    char c[1];
    if(argc == 1)
    {
        while(read(0,c,1))
            write(1,c,1);
    }
    else
    {
        int i = 0;
        for(i = 1;i < argc;i++)
        {
            fd = open(argv[i], O_RDONLY);
            if(fd == -1) //file open error, output error message
            {
```

```
                printf("mycat: %s:No such file or directory\n",
argv[i]);
                continue;
            }
            while(read(fd,c,1))
                write(1,c,1);
            close(fd);
        }
    }
    return 0;
}
```

1.2.2 mycat 实验结果



```
guest@box:~/jobs$ cc mycat.c -o mycat
guest@box:~/jobs$ ./mycat
hello 123
hello 123
OS
OS
^C
guest@box:~/jobs$ ./mycat hello
mycat: hello:No such file or directory
guest@box:~/jobs$ ./mycat hello1 hello
hello OS
mycat: hello:No such file or directory
guest@box:~/jobs$ ./mycat hello1 hello2
hello OS
hello world
guest@box:~/jobs$
```

1.2.3 mycat 实验思路

1.cat 命令后不可接参数

基于此想法，当命令后参数数量为 0(即 `argc` 为 1 时)，我们将接收到的字符串进行原样不动的输出。

`read(0,c,1)`表示从标准输入中读取字符数组

`write(1,c,1)`表示将读取的字符数组写到标准输出中去

2.cat 命令后可接多个参数

基于此想法，当命令后参数数量不为 0 是(即 `argc>1` 时)，我们对 `argv` 中的文件名进行逐个访问并打开，最后调用 `read` 与 `write` 函数将当前文件的字符写到标准输出中去。值得注意的是，如果文件夹名字不合法(即 `open` 函数返回为-1)，则应输出报错信息。

1.3 mycp.c

- 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
...
```

1.3.1 mycp 实验代码

```
#include<sys/types.h>
#include<sys/stat.h>
#include<string.h>
#include<stdlib.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
    if(argc != 3)
    {
        printf("please check your format, the right format:cp file_src
file_target\n");
        exit(0);
    }

    char *sourcePath = argv[1];
    char *targetPath = argv[2];

    int fd1 = open(sourcePath, O_RDONLY);
    int fd2 = open(targetPath, O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if(fd1 == -1)
    {
        printf("open %s error!",sourcePath);
        exit(0);
    }
    if(fd2 == -1)
    {
        printf("open %s error!",targetPath);
        exit(0);
    }
    char buf[1];
    int count;

    while (read(fd1, buf,1))
    {
        write(fd2, buf, 1);
    }
}
```

```
    }  
    close(fd1);  
    close(fd2);  
    return 0;  
}
```

1.3.2 mycp 实验结果

```
guest@box:~/jobs$ cc mycp.c -o mycp  
guest@box:~/jobs$ ./mycp  
please check your format, the right format:cp file_src file_target  
guest@box:~/jobs$ ./mycp hello1 hello2 hello3  
please check your format, the right format:cp file_src file_target  
guest@box:~/jobs$ cat hello3  
cat: hello3: No such file or directory  
guest@box:~/jobs$ cat hello1  
hello OS  
guest@box:~/jobs$ ./mycp hello1 hello3  
guest@box:~/jobs$ cat hello3  
hello OS  
guest@box:~/jobs$
```

1.3.3 mycp 实验思路

1.首先对命令格式与文件是否正常打开进行判断

当命令后跟的参数数量少于 3 个时候，提醒用户输入格式错误，并退出当前程序。当文件打开错误时，也提醒用户打开文件失败，并退出当前程序。这里需要注意的点是，在打开目标文件时，若目标文件不存在应该创建一个新文件 (O_CREAT)，并且为了能够后续操作，将其权限赋为默认权限：664 权限。

2.进行复制

当文件都正常打开时，按照以往的方法进行复制。调用 read 函数对源文件逐个读取，当其返回的字符个数为 1 时，调用 write 函数写入目标文件中。

2. 多进程题目

2.1 mysys.c: 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
-----
```

2.1.1 mysys 实验代码

```
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int mysys(char *command)
{
    if(command[0] == '\0')
    {
        printf("command not found!\n");
        return 127; // "command not found!"
    }
    int pid;
    pid = fork();
    if(pid == 0)
    {
        char *argv[100];
        char *token;
        char cmd[sizeof(command) + 1];
        strcpy(cmd, command);

        //get first substr
        token = strtok(cmd, " ");
        int count = 0;
        while(token != NULL)
        {
            argv[count++] = token;
            token = strtok(NULL, " ");
        }
        argv[count] = 0;
    }
```

```

        if(execvp(argv[0],argv) == -1)
            printf("exec failed: %d\n",errno);
    }
    else
        wait(NULL);
}

int main()
{
    mysys("");
    printf("-----\n");
    mysys("pwd");

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

    printf("-----\n");
    mysys("echo HELLO WORLD");

    printf("-----\n");
    mysys("ls /");

    printf("-----\n");
    return 0;
}

```

2.1.2 mysys 实验结果

```

guest@box:~/jobs$ cc mysys.c -o mysys
guest@box:~/jobs$ ./mysys
command not found!
-----
/home/guest/jobs
-----
hello1  mycat.c  myecho    mysys.c   pc2       pi1.c     ring      sh1.c     sh3.c
hello2  mycp       myecho.c  pc1       pc2.c     pi2       ring.c    sh2.c     sort
mycat   mycp.c     mysys     pc1.c     pi1       pi2.c     sh1       sh3       sort.c
-----
HELLO WORLD
-----
bin    dev  home  lib32  libx32  mnt  proc  run  srv  tmp  var
boot  etc  lib   lib64  media  opt  root  sbin sys  usr
-----
guest@box:~/jobs$

```

2.1.3 mysys 实验思路

1.先验知识

(1) execvp 函数

```
int execvp(const char _file, char _ const argv []);
```

execvp() 会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名, 找到后便执行该文件, 然后将第二个参数 argv 传给该欲执行的文件

(2) strtok 函数

```
char *strtok(char *str, const char *delim)
```

str -- 要被分解成一组小字符串的字符串

delim -- 包含分隔符的 C 字符串。

该函数返回被分解的第一个子字符串，如果没有可检索的字符串，则返回一个空指针。

(3) fork 函数

```
pid_t fork(void);
```

pid 是进程 ID 的缩写，pid_t 是使用 typedef 定义的进程 ID 类型

父进程从 fork 返回处继续执行，在父进程中，fork 返回子进程 PID

子进程从 fork 返回处开始执行，在子进程中，fork 返回 0

2.实验思路

(1) 首先判断命令是否合法，经过对传入的命令字符串数组的首个字符串进行判断，若不存在则打印错误信息并 return 127(返回 127 是指 command not found!)

(2) 然后进行 fork 产生子进程，在子进程中完成对 execvp 函数的调用，其中若(pid==0)表达式为真，即当前进程为子进程。

(3) 在子进程中对传入的字符串进行分割，这里用到了 strtok 函数对空格进行分割，将其分割后的子字符串存入 argv 字符串数组中，然后调用 execvp 函数，传入命令及 argv 字符串进行系统调用。

(4) 父进程中等待子进程完成后退出 mysys 函数。

2.2 sh1.c

- 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

2.2.1 sh1 实验代码


```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>
char *home;
char *dir;
int mysys(char *command)
{
    if(command[0] == '\0')
    {
        printf("command not found!\n");
        return 127; //"command not found!"
    }
    int pid;
    pid = fork();
    if(pid == 0)
    {
        char *argv[100];
        char *token;
        char cmd[sizeof(command) + 1];
        strcpy(cmd, command);

        //get first substr
        token = strtok(cmd, " ");
        int count = 0;
        while(token != NULL)
        {
            argv[count++] = token;
            token = strtok(NULL, " ");
        }
        argv[count] = 0;
        if(execvp(argv[0],argv) == -1)
            printf("exec failed: %d\n",errno);
    }
    else
        wait(NULL);
}

int choose_fun(char *cmd)
{
    char argv[100];
    strcpy(argv,cmd);

    if(argv[0] == '\0')
        return 0;
    char *token = strtok(argv, " ");

    if(strcmp(token,"cd") == 0)
        return 1;
    else if(strcmp(token,"exit") == 0)
        return 2;
    else
```

```
        return 0;
    }

    int main()
    {
        home = getenv("HOME");
        char buff[100];
        while(1)
        {
            dir = getcwd(NULL,0);
            printf("[%s]> ",dir);
            gets(buff);

            int cmdStatus = choose_fun(buff);

            if(cmdStatus == 0)
                mysys(buff);
            else if(cmdStatus == 1)
            {
                char targetdir[256];
                sscanf(buff,"cd %s",targetdir);
                chdir(targetdir);
            }
            else if(cmdStatus == 2)
                exit(0);
        }
    }
}
```

2.2.2 sh1 实验结果

```

guest@box:~/jobs$ ./sh1
[/home/guest/jobs]> ls
123      hello2    mycp      myecho.c  pcl       pc2.c     pi2       ring.c    sh2.c     sort
core     mycat      mycp.c    mysys     pcl.c     pil       pi2.c     sh1       sh3       sort.c
hello1   mycat.c    myecho    mysys.c   pc2       pil.c     ring      sh1.c     sh3.c
[/home/guest/jobs]> cd /home
[/home]> ls
guest  linuxmooc
[/home]> cd guest/jobs
[/home/guest/jobs]> pwd
/home/guest/jobs
[/home/guest/jobs]> echo Hello OS
Hello OS
[/home/guest/jobs]> ls /
bin    dev    home   lib32   libx32  mnt     proc    run     srv     tmp     var
boot   etc    lib     lib64  media   opt     root    sbin    sys     usr
[/home/guest/jobs]> 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
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
libuid:x:100:101::/var/lib/libuid:
syslog:x:101:104::/home/syslog:/bin/false
guest:x:1000:1000:,,,:/home/guest:/bin/bash
linuxmooc:x:1001:1001:,,,:/home/linuxmooc:/bin/bash
[/home/guest/jobs]> exit
guest@box:~/jobs$

```

如上图所示，我实现了 ls 等基础命令的同时，也实现了 cd、pwd、exit 等内置命令。

2.2.3 sh1 实验思路

1. 先验知识

sscanf()

函数原型：int sscanf(const char **_restrict** _Src, const char **_restrict** _Format, ...)

函数功能：从一个字符串中读进与指定格式相符的数据的函数。sscanf 与 scanf 类似，都是用于输入的，只是后者以屏幕(stdin)为输入源，前者以固定字符串为输入源。

函数示例：sscanf(buff,"cd %s",targetdir);

2. 实验思路

(1) 先判断其是否为内置指令 `cd` 以及 `exit`。调用 `strtok` 函数获得指令的一个命令字，若是 `cd` 命令则返回 1，若是 `exit` 指令则返回 2，若都不是则返回 3

```
int choose_fun(char *cmd)
{
    char argv[100];
    strcpy(argv,cmd);

    if(argv[0] == '\0')
        return 0;
    char *token = strtok(argv, " ");

    if(strcmp(token,"cd") == 0)
        return 1;
    else if(strcmp(token,"exit") == 0)
        return 2;
    else
        return 0;
}
```

(2) `main` 函数中对 `choose_fun` 返回的状态进行判断，若是普通指令则调用 `mysys` 函数执行，若是 `cd` 指令调用 `sscanf` 指令对指令字符串进行解析得到 `targetdir` 然后改变路径至 `targetdir`

```
int main()
{
    home = getenv("HOME");
    char buff[100];
    while(1)
    {
        dir = getcwd(NULL,0);
        printf("[%s]> ",dir);
        gets(buff);

        int cmdStatus = choose_fun(buff);

        if(cmdStatus == 0)
            mysys(buff);
        else if(cmdStatus == 1)
        {
            char targetdir[256];
            sscanf(buff,"cd %s",targetdir);
            chdir(targetdir);
        }
        else if(cmdStatus == 2)
            exit(0);
    }
}
```

2.3 sh2.c: 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
```

2.3.1 sh2 实验代码

2.3.2 sh2 实验结果

2.3.3 sh2 实验思路

1.先验知识

(1) strchr()

函数原型: `extern char *strchr(const char *s,char c)`

函数功能: 可以查找字符串 `s` 中首次出现字符 `c` 的位置

函数示例:

(2) dup2()

函数原型: `int dup2(int oldfd, int newfd);`

函数功能: `dup2` 可以用 `newfd` 参数指定新描述符的数值, 如果 `newfd` 已经打开, 则先将其关闭。如果 `newfd` 等于 `oldfd`, 则 `dup2` 返回 `newfd`, 而不关闭它。`dup2` 函数返回的新文件描述符同样与参数 `oldfd` 共享同一文件表项。

函数示例:

2.4 sh3.c: 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
```

2.4.1 sh3 实验代码

2.4.2 sh3 实验结果

2.4.3 sh3 实验思路

3. 多线程题目

3.1 pi1.c: 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: $1 - 1/3 + 1/5 - 1/7 + 1/9 - \dots = \pi/4$
- 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结束后,将前半部分和后半部分相加

3.1.1 pi1 实验代码

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>

#define NUMBER 100000

double PI;
double worker_output;
double master_output;

void *worker(void *arg){
    int i;
    worker_output = 0;
    for(i = 1; i <= NUMBER;i++){
        if(i % 2 == 0)
            worker_output -= 1/(2*(double)i - 1);
        else
            worker_output += 1/(2*(double)i - 1);
    }
}

void master(){
    int i;
    master_output = 0;
    for(i = NUMBER + 1;i <= NUMBER*2;i++){
        if(i % 2 == 0)
```

```
        master_output -= 1 / (2 * (double)i - 1);
    else
        master_output += 1 / (2 * (double)i - 1);
    }
}

int main()
{
    pthread_t worker_tid;
    pthread_create(&worker_tid, NULL, &worker, NULL);
    master();
    pthread_join(worker_tid, NULL);
    PI = (worker_output + master_output) * 4;
    printf("PI:%lf\n", PI);
    return 0;
}
```

3.1.2 pi1 实验结果

```
guest@box:~/jobs$ cc pi1.c -o pi1 -lpthread
guest@box:~/jobs$ ./pi1
master_output = 0.7853981384
worker_output = 0.0000000125
PI:3.141593
guest@box:~/jobs$
```

3.1.3 pi1 实验思路

(1)使用两个线程计算 PI，主线程计算前半部分，辅助线程计算后半部分。将最后的计算结果相加后乘以 4 得到 PI 的估计值。采用 `pthread_create` 函数创建辅助线程，使用 `pthread_join` 函数等待辅助线程结束。

(2)worker 和 master 函数分别计算级数的后半段和前半段，迭代次数 NUMBER 越大，数字越精确。

3.2 pi2.c: 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

3.2.1 pi2 实验代码

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

#define NUMBER 100000
#define N 100

double PI;
struct param{
```

```
    int start;
    int end;
};

struct result{
    double worker_output;
};

void *worker(void *arg){
    int i;
    struct param *param;
    struct result *result;
    double worker_output = 0;
    param = (struct param *) arg;

    for(i = param->start; i <= param->end;i++){
        if(i % 2 == 0)
            worker_output -= 1/(2*(double)i - 1);
        else
            worker_output += 1/(2*(double)i - 1);
    }
    result = malloc(sizeof(struct result));
    result->worker_output = worker_output;
    printf("worker %d = %.10lf\n",param->start / NUMBER, worker_output);
    return result;
}

int main()
{
    int i;
    pthread_t worker_tids[N];
    struct param params[N];
    PI = 0.0;

    for(i = 0; i < N;i++){
        struct param *param;
        param = &params[i];
        param->start =i * NUMBER + 1;
        param->end = (i+1) * NUMBER;
        pthread_create(&worker_tids[i], NULL, worker, param);
    }

    for(i = 0;i < N;i++){
        struct result *result;
        pthread_join(worker_tids[i],(void **)&result);
        PI += result->worker_output;
        free(result);
    }
    PI = PI * 4;
    printf("PI:%.10lf\n",PI);
    return 0;
}
```

3.2.2 pi2 实验结果


```
guest@box:~/jobs$ cc pi2.c -o pi2 -lpthread
guest@box:~/jobs$ ./pi2
worker 6 = 0.0000000595
worker 5 = 0.0000000833
worker 7 = 0.0000000446
worker 8 = 0.0000000347
worker 10 = 0.0000000227
worker 3 = 0.0000002083
worker 49 = 0.0000000010
worker 14 = 0.0000000119
worker 16 = 0.0000000092
worker 17 = 0.0000000082
worker 30 = 0.0000000027
worker 32 = 0.0000000024
worker 22 = 0.0000000049
worker 34 = 0.0000000021
worker 35 = 0.0000000020
worker 20 = 0.0000000060
worker 23 = 0.0000000045
worker 24 = 0.0000000042
worker 78 = 0.0000000004
worker 82 = 0.0000000004
worker 63 = 0.0000000006
worker 66 = 0.0000000006
worker 68 = 0.0000000005
worker 72 = 0.0000000005
worker 74 = 0.0000000005
worker 84 = 0.0000000004
worker 92 = 0.0000000003
worker 93 = 0.0000000003
worker 98 = 0.0000000003
worker 91 = 0.0000000003
worker 71 = 0.0000000005
worker 65 = 0.0000000006
worker 73 = 0.0000000005
worker 90 = 0.0000000003
worker 75 = 0.0000000004
worker 76 = 0.0000000004
worker 97 = 0.0000000003
worker 87 = 0.0000000003
worker 80 = 0.0000000004
worker 62 = 0.0000000006
worker 86 = 0.0000000003
PI:3.1415925536
guest@box:~/jobs$
```

3.2.3 pi2 实验思路

(1) 主线程采用 for 循环产生 100 个线程来计算 PI。每个线程计算 1/100 的部分，计算起止点作为 pthread_create 函数的参数 param 传入辅助线程的线程入口函数。在 Worker 函数中，通过 param = (struct param *) arg 来接收传过来的 param 结构体。

(2) 主线程采用 for 循环，将每个辅助线程的计算结果利用 pthread_join 函数接受线程入口函数的返回值 result, 然后获得每一个 worker 的 worker_output 在进行相加得到 PI 的值。

3.3 sort.c: 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结束后,使用归并排序算法归并数组的前半部分和后半部分

3.3.1 sort 实验代码

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

#define MAX_ARRAY 100
#define MAX_NUM 10000

void *selectSort(void *argc){
    int *argv = (int *)argc;
    int i,j,min,record = -1,temp;
    for(i = 0;i < MAX_ARRAY/2; ++i){
        min = argv[i];
        for(j = i;j < MAX_ARRAY / 2;++j){
            if(argv[j] < min){
                record = j;
                min = argv[j];
            }
        }
        temp = argv[i];
        argv[i] = argv[record];
        argv[record] = temp;
    }
    return NULL;
}

void Merge(int *arg1, int*arg2){
    int i = 0;
    int j = 0;
    int k = 0;
    for(j = MAX_ARRAY/2;i < MAX_ARRAY/2 && j < MAX_ARRAY;++k){
        if(arg1[i] < arg1[j])
            arg2[k] = arg1[i++];
        else
            arg2[k] = arg1[j++];
    }
    while(i < MAX_ARRAY/2)
        arg2[k++] = arg1[i++];
    while(j < MAX_ARRAY)
        arg2[k++] = arg1[j++];
}

void printArray(int *array){
    int i = 0;
    for(;i<MAX_ARRAY;++i)
    {
```

```
        printf("%5d ",array[i]);
        if((i+1) % 10 == 0 && i!=1)
            printf("\n");
    }
    printf("\n");
}

int main(){
    int array[MAX_ARRAY],result[MAX_ARRAY];
    int i;
    for(i = 0;i < MAX_ARRAY; ++i)
        array[i] = (rand() % MAX_NUM);

    printf("[UnSort ARRAY]!\n");
    printArray(array);
    pthread_t tid;
    int *arg = &array[MAX_ARRAY/2];
    selectSort(array);
    pthread_create(&tid, NULL, selectSort, (void *)arg);
    pthread_join(tid,NULL);

    printf("[SELECTSORT ARRAY]!\n");
    printArray(array);

    Merge(array,result);
    printf("[Sorted RESULT]!\n");
    printArray(result);
    return 0;
}
```

3.3.2 sort 实验结果

```

guest@box:~/jobs$ cc sort.c -o sort -lpthread
guest@box:~/jobs$ ./sort
[UnSort ARRAY] !
 9383   886   2777   6915   7793   8335   5386   492   6649   1421
 2362    27   8690    59   7763   3926    540   3426   9172   5736
 5211   5368   2567   6429   5782   1530   2862   5123   4067   3135
 3929   9802   4022   3058   3069   8167   1393   8456   5011   8042
 6229   7373   4421   4919   3784   8537   5198   4324   8315   4370
 6413   3526   6091   8980   9956   1873   6862   9170   6996   7281
 2305    925   7084   6327    336   6505    846   1729   1313   5857
 6124   3895   9582    545   8814   3367   5434    364   4043   3750
 1087   6808   7276   7178   5788   3584   5403   2651   2754   2399
 9932   5060   9676   3368   7739    12   6226   8586   8094   7539

[SELECTSORT ARRAY] !
  27    59    492    540    886   1393   1421   1530   2362   2567
 2777   2862   3058   3069   3135   3426   3784   6915   3926   3929
 4022   4067   4324   4370   4421   4919   5011   9383   5123   5198
 5211   5368   5386   5736   5782   6229   6429   6649   7373   7763
 7793   8042   9802   8167   8315   8335   8537   8456   8690   9172
  12    336    364    545    846    925   1087   1313   1729   1873
 7281   2305   2399   2651   2754   3367   3368   3526   3584   3750
 3895   4043   5060   5403   5434   5788   5857   8814   6091   6124
 6226   6327   6413   6505   6808   6862   6996   7084   7178   7276
 7539   7739   8094   8586   8980   9170   9582   9676   9932   9956

[Sorted RESULT] !
  12    27    59    336    364    492    540    545    846    886
  925   1087   1313   1393   1421   1530   1729   1873   2362   2567
 2777   2862   3058   3069   3135   3426   3784   6915   3926   3929
 4022   4067   4324   4370   4421   4919   5011   7281   2305   2399
 2651   2754   3367   3368   3526   3584   3750   3895   4043   5060
 5403   5434   5788   5857   8814   6091   6124   6226   6327   6413
 6505   6808   6862   6996   7084   7178   7276   7539   7739   8094
 8586   8980   9170   9383   5123   5198   5211   5368   5386   5736
 5782   6229   6429   6649   7373   7763   7793   8042   9582   9676
 9802   8167   8315   8335   8537   8456   8690   9172   9932   9956

guest@box:~/jobs$

```

3.3.3 sort 实验思路

(1)首先随机生成了100个数，将其打印。主线程调用selectSort函数进行前半部分的排序，采用pthread_create函数创建辅助线程，辅助线程使用选择排序算法对数组的后半部分排序。将辅助线程的排序任务起止点利用线程入口函数的参数传入。线程入口函数采用选择排序。使用pthread_join函数等待辅助线程结束。

```

void *selectSort(void *argc){
    int *argv = (int *)argc;
    int i,j,min,record = -1,temp;

```

```

        for(i = 0; i < MAX_ARRAY/2; ++i){
            min = argv[i];
            for(j = i; j < MAX_ARRAY / 2; ++j){
                if(argv[j] < min){
                    record = j;
                    min = argv[j];
                }
            }
            temp = argv[i];
            argv[i] = argv[record];
            argv[record] = temp;
        }
        return NULL;
    }

    int array[MAX_ARRAY], result[MAX_ARRAY];
    int i;
    for(i = 0; i < MAX_ARRAY; ++i)
        array[i] = (rand() % MAX_NUM);

    printf("[UnSort ARRAY]!\n");
    printArray(array);
    pthread_t tid;
    int *arg = &array[MAX_ARRAY/2];
    selectSort(array);
    pthread_create(&tid, NULL, selectSort, (void *)arg);
    pthread_join(tid, NULL);

    printf("[SELECTSORT ARRAY]!\n");
    printArray(array);

```

(2)使用merge函数归并数组的前半部分和后半部分,直接在主线程里调用自定义的Merge函数即可将两个选择排序的结果归并排序完成。

```

void Merge(int *arg1, int*arg2){
    int i = 0;
    int j = 0;
    int k = 0;
    for(j = MAX_ARRAY/2; i < MAX_ARRAY/2 && j < MAX_ARRAY; ++k){
        if(arg1[i] < arg1[j])
            arg2[k] = arg1[i++];
        else
            arg2[k] = arg1[j++];
    }
    while(i < MAX_ARRAY/2)
        arg2[k++] = arg1[i++];
    while(j < MAX_ARRAY)
        arg2[k++] = arg1[j++];
}

Merge(array, result);
printf("[Sorted RESULT]!\n");
printArray(result);

```

3.4 pc1.c: 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

3.4.1 pc1 实验代码

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#define CAPACITY 4

char buffer1[CAPACITY];
char buffer2[CAPACITY];
int in1,out1;
int in2,out2;

int buffer_is_empty(int index){
    if(index == 1)
        return in1 == out1;
    if(index == 2)
        return in2 == out2;
    else
        printf("Don`t exist this buffer!,Empty");
}

int buffer_is_full(int index){
    if(index == 1)
        return (in1 + 1) % CAPACITY == out1;
    if(index == 2)
        return (in2 + 1) % CAPACITY == out2;
    else
        printf("Don`t exist this buffer!,Full");
}

char get_item(int index){
    char item;
    if(index == 1){
        item = buffer1[out1];
        out1 = (out1 + 1) % CAPACITY;
    }
    if(index == 2){
        item = buffer2[out2];
        out2 = (out2 + 1) % CAPACITY;
    }
    //else
    // printf("Don`t exist this buffer!,Get%d\n",index);
    return item;
}

void put_item(char item, int index){
```

```

    if(index == 1){
        buffer1[in1] = item;
        in1 = (in1 + 1) % CAPACITY;
    }
    if(index == 2){
        buffer2[in2] = item;
        in2 = (in2 + 1) % CAPACITY;
    }
    //else
    //    printf("Don't exist this buffer!Put%c %d\n",item,index);
}

pthread_mutex_t mutex1,mutex2;
pthread_cond_t wait_empty_buffer1;
pthread_cond_t wait_full_buffer1;
pthread_cond_t wait_empty_buffer2;
pthread_cond_t wait_full_buffer2;

volatile int global = 0;

#define ITEM_COUNT 8

void *produce(void *arg){
    int i;
    char item;

    for(i = 0;i < ITEM_COUNT;i++){
        pthread_mutex_lock(&mutex1);
        while(buffer_is_full(1))
            pthread_cond_wait(&wait_empty_buffer1, &mutex1);
        item = 'a' + i;
        put_item(item,1);
        printf("produce item:%c\n",item);

        pthread_cond_signal(&wait_full_buffer1);
        pthread_mutex_unlock(&mutex1);
    }
    return NULL;
}

void *compute(void *arg){
    int i;
    char item;
    for(i = 0;i < ITEM_COUNT;i++){
        pthread_mutex_lock(&mutex1);
        while(buffer_is_empty(1))
            pthread_cond_wait(&wait_full_buffer1, &mutex1);
        item = get_item(1);
        //printf("    compute get item:%c\n",item);
        pthread_cond_signal(&wait_empty_buffer1);
        pthread_mutex_unlock(&mutex1);

        item -= 32;

        pthread_mutex_lock(&mutex2);
        while(buffer_is_full(2))
            pthread_cond_wait(&wait_empty_buffer2, &mutex2);
    }
}

```

```

        put_item(item,2);
        printf("    compute put item:%c\n", item);
        pthread_cond_signal(&wait_full_buffer2);
        pthread_mutex_unlock(&mutex2);
    }
    return NULL;
}

void *consume(void *arg){
    int i;
    char item;
    for(i = 0;i < ITEM_COUNT;i++){
        pthread_mutex_lock(&mutex2);
        while(buffer_is_empty(2))
            pthread_cond_wait(&wait_full_buffer2, &mutex2);
        item = get_item(2);
        printf("    consume item:%c\n", item);

        pthread_cond_signal(&wait_empty_buffer2);
        pthread_mutex_unlock(&mutex2);
    }
    return NULL;
}

int main(){
    int i;
    in1 = 0;
    in2 = 0;
    out1 = 0;
    out2 = 0;
    pthread_t tids[3];
    pthread_create(&tids[0],NULL,produce,NULL);
    pthread_create(&tids[1],NULL,compute,NULL);
    pthread_create(&tids[2],NULL,consume,NULL);

    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);
    pthread_cond_init(&wait_empty_buffer1, NULL);
    pthread_cond_init(&wait_full_buffer1, NULL);
    pthread_cond_init(&wait_empty_buffer2, NULL);
    pthread_cond_init(&wait_full_buffer2, NULL);

    for(i = 0;i < 3;i++)
        pthread_join(tids[i],NULL);
    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}

```

3.4.2 pc2 实验结果


```
produce item:a
produce item:b
produce item:c
    compute put item:A
    compute put item:B
    compute put item:C
        comsume item:A
        comsume item:B
        comsume item:C
produce item:d
produce item:e
produce item:f
    compute put item:D
    compute put item:E
    compute put item:F
        comsume item:D
        comsume item:E
        comsume item:F
produce item:g
produce item:h
    compute put item:G
    compute put item:H
        comsume item:G
        comsume item:H
guest@box:~/jobs$
```

3.4.3 pc2 实验思路

1.先验知识

(1) pthread_mutex_t 与 pthread_cond_t

类型名	类型功能	声明原型	声明示例
pthread_mutex_t	声明一个互斥锁(用于线程互斥)	typedef void *pthread_mutex_t	pthread_mutex_t mutex1,mutex2;
pthread_cond_t	声明一个条件变量(用于线程同步)	typedef void *pthread_mutex_t	pthread_cond_t wait_empty_buffer1;

(2) pthread_mutex_lock() 和 pthread_mutex_unlock()

函数名	函数功能	函数原型	函数示例

函数名	函数功能	函数原型	函数示例
pthread_mutex_lock()	对互斥锁加锁	int pthread_mutex_lock(pthread_mutex_t *m)	pthread_mutex_lock(&mutex1);
pthread_mutex_unlock()	对互斥锁解锁	int pthread_mutex_unlock(pthread_mutex_t *m)	pthread_mutex_unlock(&mutex1);

(3) pthread_cond_wait() 和 pthread_cond_signal()

函数名	函数功能	函数原型	函数示例
pthread_cond_wait()	无条件等待	int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *external_mutex)	pthread_cond_wait(&wait_empty_buffer1, &mutex1);
pthread_cond_signal()	激活等待的线程	int pthread_cond_signal(pthread_cond_t *cv, pthread_mutex_t *external_mutex)	pthread_cond_signal(&wait_full_buffer1);

(4) pthread_mutex_init()、pthread_mutex_destroy() 和 pthread_cond_init()

函数名	函数功能	函数原型	函数示例
pthread_mutex_init()	互斥锁的初始化	int pthread_mutex_init(pthread_mutex_t *_mutex, const pthread_mutexattr_t _attr)	pthread_mutex_init(&mutex1, NULL);

函数名	函数功能	函数原型	函数示例
pthread_mutex_destory()	互斥锁的释放	int pthread_mutex_destory(pthread_mutex_t *m)	pthread_mutex_destory(&mutex1);
pthread_cond_init()	条件变量的初始化	int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *a)	thread_cond_init(&wait_empty_buffer1, NULL);

(5) pthread_create() 和 pthread_join()

函数名	函数功能	函数原型	函数示例
pthread_create()	线程的创建	int pthread_create(pthread_t *th, const pthread_attr_t *attr, void *(*func)(void *), void *arg)	pthread_create(&tids,NULL,produce,NULL);
pthread_join()	用于等待一个线程的结束,线程间同步的操作	int pthread_join(pthread_t t, void **res)	pthread_join(tids,NULL);

2.实验思路

(1) 定义两个容量为 4 的 buffer：buffer1 与 buffer2。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。定义互斥信号量用于进程间互斥，定义条件变量用于进程间同步

```
#define CAPACITY 4 //缓冲区的大小
#define ITEM_COUNT 8 //字符的数量
char buffer1[CAPACITY];
char buffer2[CAPACITY];
int in1,out1; //定义当前buffer1的读指针和写指针
int in2,out2; //定义当前buffer2的读指针和写指针
pthread_mutex_t mutex1,mutex2; //定义互斥信号量
pthread_cond_t wait_empty_buffer1;
pthread_cond_t wait_full_buffer1; //定义条件变量用于produce与compute之间的同步
pthread_cond_t wait_empty_buffer2;
pthread_cond_t wait_full_buffer2; //定义条件变量用于compute与consume之间的同步
```

(2) produce 程序作为 buffer1 的生产者，在操作之前给 buffer1 加锁并将数据存入。

```
void *produce(void *arg){
    int i;
    char item;

    for(i = 0;i < ITEM_COUNT;i++){
        pthread_mutex_lock(&mutex);                //对互斥锁进行加锁
        while(buffer_is_full(1))
            pthread_cond_wait(&wait_empty_buffer1, &mutex); //P操作：若buffer1满了就等待
其为空
        item = 'a' + i;
        put_item(item,1);
        printf("produce item:%c\n",item);

        pthread_cond_signal(&wait_full_buffer1);    //V操作：将buffer1的数据缓冲
区数目(wait_full_buffer1) + 1
        pthread_mutex_unlock(&mutex);              //释放信号量
    }
    return NULL;
}
```

(3) compute 程序先作为 buffer1 的消费者，给 buffer1 加锁并取数；计算者将小写字母变成大写字母；计最后再作为 buffer2 的生产者，给 buffer2 加锁并存数。

```
void *compute(void *arg){
    int i;
    char item;

    for(i = 0;i < ITEM_COUNT;i++){
        pthread_mutex_lock(&mutex1);                //对信号量1加锁
        while(buffer_is_empty(1))
            pthread_cond_wait(&wait_full_buffer1, &mutex1); //P操作：若buffer1为空则持续
等待
        item = get_item(1);
        //printf("    compute get item:%c\n",item);
        pthread_cond_signal(&wait_empty_buffer1);    //V操作：将buffer1的数据缓冲
区数目(wait_empty_buffer1)-1
        pthread_mutex_unlock(&mutex1);              //释放信号量1

        item -= 32;

        pthread_mutex_lock(&mutex2);                //对信号量2加锁
        while(buffer_is_full(2))
            pthread_cond_wait(&wait_empty_buffer2, &mutex2); //P操作：若buffer2满了则持续
等待
        put_item(item,2);
        printf("    compute put item:%c\n", item);
        pthread_cond_signal(&wait_full_buffer2);    //V操作：将buffer2的数据缓冲
区数目(wait_full_buffer2)+1
        pthread_mutex_unlock(&mutex2);              //释放信号量2
    }
}
```

```
    return NULL;
}
```

(4)消费者作为 `buffer2` 的消费者，给 `buffer2` 加锁并取数字。

```
void *consume(void *arg){
    int i;
    char item;
    for(i = 0; i < ITEM_COUNT; i++){
        pthread_mutex_lock(&mutex2);                //对信号量2加锁
        while(buffer_is_empty(2))
            pthread_cond_wait(&wait_full_buffer2, &mutex2); //P操作：若buffer2为空则持续
等待
        item = get_item(2);
        printf("        consume item:%c\n", item);
        pthread_cond_signal(&wait_empty_buffer2);    //V操作：将buffer2的数据缓冲
区数目(wait_empty_buffer2)-1
        pthread_mutex_unlock(&mutex2);                //释放信号量2
    }
    return NULL;
}
```

(5)在主函数中创建三个线程分别用于承担生产者，计算者与消费者。对线程进行初始化，并且定义两个锁用于线程间互斥，再定义四个信号量用于线程间同步，再将三个进程都调用 `pthread_join()` 函数等待线程结束，最终对互斥锁进行注销。

```
int main(){
    int i;
    in1 = 0;
    in2 = 0;
    out1 = 0;
    out2 = 0;
    pthread_t tids[3];
    pthread_create(&tids[0], NULL, produce, NULL);
    pthread_create(&tids[1], NULL, compute, NULL);
    pthread_create(&tids[2], NULL, consume, NULL);

    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);
    pthread_cond_init(&wait_empty_buffer1, NULL);
    pthread_cond_init(&wait_full_buffer1, NULL);
    pthread_cond_init(&wait_empty_buffer2, NULL);
    pthread_cond_init(&wait_full_buffer2, NULL);

    for(i = 0; i < 3; i++)
        pthread_join(tids[i], NULL);
    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}
```

3.5 pc2.c: 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

3.5.1 pc2 实验代码

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#define CAPACITY 4

char buffer1[CAPACITY];
char buffer2[CAPACITY];
int in1,out1;
int in2,out2;

int buffer_is_empty(int index){
    if(index == 1)
        return in1 == out1;
    if(index == 2)
        return in2 == out2;
    else
        printf("Don`t exist this buffer!,Empty");
}

int buffer_is_full(int index){
    if(index == 1)
        return (in1 + 1) % CAPACITY == out1;
    if(index == 2)
        return (in2 + 1) % CAPACITY == out2;
    else
        printf("Don`t exist this buffer!,Full");
}

char get_item(int index){
    char item;
    if(index == 1){
        item = buffer1[out1];
        out1 = (out1 + 1) % CAPACITY;
    }
    if(index == 2){
        item = buffer2[out2];
        out2 = (out2 + 1) % CAPACITY;
    }
    //else
    // printf("Don`t exist this buffer!,Get%d\n",index);
    return item;
}

void put_item(char item, int index){
    if(index == 1){
        buffer1[in1] = item;
        in1 = (in1 + 1) % CAPACITY;
    }
    if(index == 2){
```

```

        buffer2[in2] = item;
        in2 = (in2 + 1) % CAPACITY;
    }
    //else
    //    printf("Don`t exist this buffer!Put%c  %d\n",item,index);
}

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_sema1,mutex_sema2;
sema_t empty_buffer_sema1;
sema_t full_buffer_sema1;
sema_t empty_buffer_sema2;
sema_t full_buffer_sema2;

volatile int global = 0;

#define ITEM_COUNT 8

void *produce(void *arg){
    int i;
    char item;

    for(i = 0;i < ITEM_COUNT;i++){
        sema_wait(&empty_buffer_sema1);
        sema_wait(&mutex_sema1);

        item = 'a' + i;
        put_item(item,1);
        printf("produce item:%c\n",item);

        sema_signal(&mutex_sema1);
    }
}

```

```
        sema_signal(&full_buffer_sema1);
    }
    return NULL;
}

void *compute(void *arg){
    int i;
    char item;
    for(i = 0; i < ITEM_COUNT; i++){
        sema_wait(&full_buffer_sema1);
        sema_wait(&mutex_sema1);

        item = get_item(1);
        // printf("    compute get item:%c\n", item);

        sema_signal(&mutex_sema1);
        sema_signal(&empty_buffer_sema1);

        item -= 32;

        sema_wait(&empty_buffer_sema2);
        sema_wait(&mutex_sema2);

        put_item(item, 2);
        printf("    compute put item:%c\n", item);

        sema_signal(&mutex_sema2);
        sema_signal(&full_buffer_sema2);
    }
    return NULL;
}

void *consume(void *arg){
    int i;
    char item;
    for(i = 0; i < ITEM_COUNT; i++){

        sema_wait(&full_buffer_sema2);
        sema_wait(&mutex_sema2);

        item = get_item(2);
        printf("        consume item:%c\n", item);

        sema_signal(&mutex_sema2);
        sema_signal(&empty_buffer_sema2);
    }
    return NULL;
}

int main(){
    int i;
    in1 = 0;
    in2 = 0;
    out1 = 0;
    out2 = 0;
    pthread_t tids[3];

    sema_init(&mutex_sema1, 1);
```



```

        sema_init(&mutex_sema2, 1);
        sema_init(&empty_buffer_sema1,CAPACITY - 1);
        sema_init(&full_buffer_sema1,0);
        sema_init(&empty_buffer_sema2,CAPACITY - 1);
        sema_init(&full_buffer_sema1,0);

        pthread_create(&tids[0],NULL,produce,NULL);
        pthread_create(&tids[1],NULL,compute,NULL);
        pthread_create(&tids[2],NULL,consume,NULL);

        for(i = 0;i < 3;i++)
            pthread_join(tids[i],NULL);

        return 0;
    }

```

3.5.2 pc2 实验结果

```

guest@box:~/jobs$ vim pc2.c
guest@box:~/jobs$ cc pc2.c -o pc2 -lpthread
guest@box:~/jobs$ ./pc2
produce item:a
produce item:b
produce item:c
    compute put item:A
    compute put item:B
    compute put item:C
        consume item:A
        consume item:B
        consume item:C
produce item:d
produce item:e
produce item:f
    compute put item:D
    compute put item:E
    compute put item:F
        consume item:D
        consume item:E
        consume item:F
produce item:g
produce item:h
    compute put item:G
    compute put item:H

```

3.5.3 pc2 实验思路

(1) 信号量的实现

此题与上题思路相同，区别在于实现的时候利用信号量。信号量的定义、初始化、wait 和 signal 定义如下，初始化时可以送入信号量的初始个数，wait 一次减少一次信号量个数，signal 一次则增加一次信号量个数。

```

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);
}

```

(2)定义信号量并使用

定义两个信号量 `mutex_sema1,mutex_sema2`，分别对(生产者-计算者)与(计算者-消费者)进行线程间互斥。此外也定义了四个信号量 对共享变量 `buffer1,buffer2` 进行线程间同步。

在生产者、计算者、消费者的函数中，先进行 P 操作等待互斥信号量(上锁)，再 P 操作获取同步信号量，对 `buffer` 中的数据进行操作后，V 操作释放互斥信号量及同步信号量(解锁)。这里值得注意的是，需要 P 操作需要先获取同步信号量再对互斥信号量进行上锁，不然可能造饥饿的现象。

```

sema_t mutex_sema1,mutex_sema2;
sema_t empty_buffer_sema1;
sema_t full_buffer_sema1;
sema_t empty_buffer_sema2;
sema_t full_buffer_sema2;
void *produce(void *arg){
    int i;
    char item;

    for(i = 0;i < ITEM_COUNT;i++){
        sema_wait(&empty_buffer_sema1);
        sema_wait(&mutex_sema1);

        item = 'a' + i;
        put_item(item,1);
        printf("produce item:%c\n",item);

        sema_signal(&mutex_sema1);
        sema_signal(&full_buffer_sema1);
    }
    return NULL;
}

```

```

void *compute(void *arg){
    int i;
    char item;
    for(i = 0;i < ITEM_COUNT;i++){
        sema_wait(&full_buffer_sema1);
        sema_wait(&mutex_sema1);

        item = get_item(1);
        // printf("    compute get item:%c\n",item);

        sema_signal(&mutex_sema1);
        sema_signal(&empty_buffer_sema1);

        item -= 32;

        sema_wait(&empty_buffer_sema2);
        sema_wait(&mutex_sema2);

        put_item(item,2);
        printf("    compute put item:%c\n", item);

        sema_signal(&mutex_sema2);
        sema_signal(&full_buffer_sema2);
    }
    return NULL;
}

void *consume(void *arg){
    int i;
    char item;
    for(i = 0;i < ITEM_COUNT;i++){

        sema_wait(&full_buffer_sema2);
        sema_wait(&mutex_sema2);

        item = get_item(2);
        printf("        consume item:%c\n", item);

        sema_signal(&mutex_sema2);
        sema_signal(&empty_buffer_sema2);
    }
    return NULL;
}

```

(3) main 函数中开启三个线程分别对应生产者、计算者、消费者，再对两个互斥信号量以及四个同步信号量进行初始化，调用 pthread_join 函数等待三个进程的结束即可。

```

int main(){
    int i;
    in1 = 0;
    in2 = 0;
    out1 = 0;
    out2 = 0;
    pthread_t tids[3];

```

```

    sema_init(&mutex_sema1, 1);
    sema_init(&mutex_sema2, 1);
    sema_init(&empty_buffer_sema1, CAPACITY - 1);
    sema_init(&full_buffer_sema1, 0);
    sema_init(&empty_buffer_sema2, CAPACITY - 1);
    sema_init(&full_buffer_sema1, 0);

    pthread_create(&tids[0], NULL, produce, NULL);
    pthread_create(&tids[1], NULL, compute, NULL);
    pthread_create(&tids[2], NULL, consume, NULL);

    for(i = 0; i < 3; i++)
        pthread_join(tids[i], NULL);

    return 0;
}

```

3.6 ring.c: 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

3.6.1 ring 实验代码

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#define N 100
int buffer = 0;

void *add(void *arg){
    int *num = (int *)arg;
    num[0]++;
    int *result = num;
    return (void *)result;
}

void init(int a[N][2]){
    int i;
    for(i = 0; i < N; i++){
        a[i][0] = 0;
        a[i][1] = i;
    }
}

```

```
    }  
}  
  
int main(){  
    int i = 0;  
    int array[N][2];  
    init(array);  
    int *result;  
  
    pthread_t tids[N];  
    pthread_create(&tids[0],NULL,add,(void *)array[0]);  
    pthread_join(tids[0], (void *)&result);  
  
    while(i < N){  
        printf("from T[%d]", i+1);  
        i = (i+1) % N;  
        printf("to T[%d] send %d\n",i+1,result[0]);  
        pthread_create(&tids[i],NULL,add,result);  
        pthread_join(tids[i], (void *)&result);  
        if(i == 0)  
            break;  
        // sleep(1);  
    }  
  
    return 0;  
}
```

3.6.2 ring 实验结果

```
guest@box:~/jobs$ vim ring.c
guest@box:~/jobs$ cc ring.c -o ring -lpthread
guest@box:~/jobs$ ./ring
from T[1]to T[2] send 1
from T[2]to T[3] send 2
from T[3]to T[4] send 3
from T[4]to T[5] send 4
from T[5]to T[6] send 5
from T[6]to T[7] send 6
from T[7]to T[8] send 7
from T[8]to T[9] send 8
from T[9]to T[10] send 9
from T[10]to T[11] send 10
from T[11]to T[12] send 11
from T[12]to T[13] send 12
from T[13]to T[14] send 13
from T[14]to T[15] send 14
from T[15]to T[16] send 15
from T[16]to T[17] send 16
from T[17]to T[18] send 17
from T[18]to T[19] send 18
from T[19]to T[20] send 19

from T[77]to T[78] send 77
from T[78]to T[79] send 78
from T[79]to T[80] send 79
from T[80]to T[81] send 80
from T[81]to T[82] send 81
from T[82]to T[83] send 82
from T[83]to T[84] send 83
from T[84]to T[85] send 84
from T[85]to T[86] send 85
from T[86]to T[87] send 86
from T[87]to T[88] send 87
from T[88]to T[89] send 88
from T[89]to T[90] send 89
from T[90]to T[91] send 90
from T[91]to T[92] send 91
from T[92]to T[93] send 92
from T[93]to T[94] send 93
from T[94]to T[95] send 94
from T[95]to T[96] send 95
from T[96]to T[97] send 96
from T[97]to T[98] send 97
```

```
from T[98]to T[99] send 98
from T[99]to T[100] send 99
from T[100]to T[1] send 100
guest@box:~/jobs$
```

3.6.3 ring 实验思路

(1) 创建 N 个容量为 2 的缓冲区 `array[N][2]`, 初始化每个缓冲区的第一个数字都为 0, 第二个数字为当前缓冲区的编号。接着定义了一个数字数组 `result`. 创建 N 个线程, 对第一个线程进行初始化, 该进程运行 `add` 函数, 然后将 `array[0]` 作为参数传入 `add` 函数中。并调用 `pthread_join()` 函数等待线程结束。

```
void *add(void *arg){
    int *num = (int *)arg;
    num[0]++;
    int *result = num;
    return (void *)result;
}

void init(int a[N][2]){
    int i;
    for(i = 0; i < N; i++){
        a[i][0] = 0;
        a[i][1] = i;
    }
}

int i = 0;
int array[N][2];
init(array);
int *result;

pthread_t tids[N];
pthread_create(&tids[0], NULL, add, (void *)array[0]);
pthread_join(tids[0], (void *)&result);
```

(2) 在循环中, 首先打印发送方的信息, 然后 i 循环加 1, 然后打印接收方的信息并创建线程, 将收到的数作为参数传入线程中执行 `add` 函数, 并调用 `pthread_join()` 函数等待线程结束。当 i 循环加 1 到 0 到 i 为 0 时(又返回第一个缓冲区)时结束。

```
while(i < N){
    printf("from T[%d]", i+1);
    i = (i+1) % N;
    printf("to T[%d] send %d\n", i+1, result[0]);
    pthread_create(&tids[i], NULL, add, result);
    pthread_join(tids[i], (void *)&result);
    if(i == 0)
        break;
    // sleep(1);
}
```

