2018

操作系统实验报告

姓名：陶略

学号：031510319

班号：1615102

目录

文件读写编程题目

题一：myecho.c

题目要求：

1. myecho.c 的功能与系统 echo 程序相同
2. 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
```

实现思路：

①使用 main 函数里的 argc 和 argv 传参数，命令行输入的字符串会自动被分割成字符串数组，argc 记录字符串个数，argv 指向字符串数组。只需用一个循环输出以空格为间隔输出所有字符串即可，最后输出一个回车。

代码：

```
1. int main(int argc, char** argv)
2. {
3.     for(int i = 1; i < argc; i++)
4.         printf("%s ", argv[i]);
5.     printf("\n");
6.     return 0;
7. }
```

实验截图：

```
root@Lue:~/os_experiment/program_task/file_read_write# cc myecho.c -o myecho
root@Lue:~/os_experiment/program_task/file_read_write# ./myecho x
x
root@Lue:~/os_experiment/program_task/file_read_write# ./myecho a b c
a b c
```

题二：mycat.c

题目要求：

1. mycat.c 的功能与系统 cat 程序相同
2. mycat 将指定的文件内容输出到屏幕，例子如下：
3. 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
...
```

实现思路:

①首先实现了一个函数如下。该函数接受任意两个文件描述符，将 infile 文件的内容全部送入 outfile 文件。文件描述符 infile 可以是用 open 函数打开的文件，也可以时标准输入；文件描述符 outfile 可以是用 open 函数打开的文件，也可以是标准输出。

```
1.  int copy_file(int infile, int outfile)
2.  {
3.      int num;
4.      char buf[1];
5.      do{
6.          num = read(infile, buf, 1);
7.          write(outfile, buf, num);
8.      }while(num == 1);
9.      return num;
10. }
```

② cat 接受的参数个数可以是任意自然数，即包括 0。

当参数个数为 0 时，只需把标准输入的字符串送入标准输出即可。

```
1.  if(argc == 1)
2.  {
3.      copy_file(0, 1);
4.  }
```

当参数个数大于 0 时，对参数个数做一个循环，并判断文件打开是否成功，成功则调用 copy_file(infile, 1)将文件内容送入标准输出，失败则提示失败并跳过该文件。

```
1.  char *filename;
2.  int infile;
3.  for(int i = 1; i < argc; i++)
4.  {
5.      filename = argv[i];
6.      if((infile = open(argv[1], O_RDONLY)) == -1)
7.      {
8.          printf("mycat: %s: No such file or directory\n", filename
    );
```

```
9.          continue;
10.     }
11.     copy_file(infile, 1);
12.     close(infile);
13. }
```

实验截图：

```
root@Lue:~/os_experiment/program_task/file_read_write# cc mycat2.c -o mycat
root@Lue:~/os_experiment/program_task/file_read_write# ./mycat
hello world
hello world
^C
root@Lue:~/os_experiment/program_task/file_read_write# echo hello OS > input
root@Lue:~/os_experiment/program_task/file_read_write# ./mycat input
hello OS
root@Lue:~/os_experiment/program_task/file_read_write# ./mycat file_not_exist.txt input
mycat: file_not_exist.txt: No such file or directory
hello OS
```

题三：mycp.c

题目要求：

1. mycp.c 的功能与系统 cp 程序相同
2. 将源文件复制到目标文件，例子如下：
3. 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
...
```

实现思路：

①如题二，实现了同一个函数 copy_file(int, int)。该函数接受任意两个文件描述符，将 infile 文件的内容全部送入 outfile 文件。文件描述符 infile 可以是用 open 函数打开的文件，也可以时标准输入；文件描述符 outfile 可以是用 open 函数打开的文件，也可以是标准输出。

```
1.  int copy_file(int infile, int outfile)
```

②cp 接受的参数个数为 2 个。

①当参数个数为 0 时，报错。

```
5.  if(argc != 3)
6.  {
7.      printf("The format must be:cp file_src file_des");
8.      exit(0);
9.  }
```

②分别打开 argv[1]和 argv[2]，打开失败，则报错。

```
1.  if((infile = open(argv[1], O_RDONLY)) == -1)
2.  {
3.      printf("mycp: %s: No such file or directory\n", argv[1]);
4.      exit(0);
5.  }
6.  if((outfile = open(argv[2], O_CREAT | O_EXCL | O_WRONLY, 0644)) =
    = -1)
7.  {
8.      printf("mycp: %s: Can't create such file\n", argv[2]);
9.      exit(0);
10. }
```

③将打开成功的两个文件描述符传入 copy_file 函数，而后关闭文件。

```
1.  copy_file(infile, outfile);
2.
3.  close(infile);
4.  close(outfile);
```

实验截图：

```
root@Lue:~/os_experiment/program_task/file_read_write# cc mycp2.c -o mycp
root@Lue:~/os_experiment/program_task/file_read_write# ./mycp
The format must be:cp file_src file_des
root@Lue:~/os_experiment/program_task/file_read_write# ls
a.out  input  mycat  mycat2.c  mycat.c  mycp  mycp2.c  mycp.c  myecho  myecho.c
root@Lue:~/os_experiment/program_task/file_read_write# ./mycp file_not_exist.txt new.txt
mycp: file_not_exist.txt: No such file or directory
root@Lue:~/os_experiment/program_task/file_read_write# ./mycp input newfile
root@Lue:~/os_experiment/program_task/file_read_write# cat newfile
hello OS
```

多进程题目

题一：mysys.c

题目要求：

1. 实现函数 mysys，用于执行一个系统命令，要求如下
2. 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

3. 使用 fork/exec/wait 系统调用实现 mysys
4. 不能通过调用系统函数 system 实现 mysys

实现思路：本题有两种实现方式

①直接解析出命令和参数，在子进程中使用 execvp 装入并执行。

解析命令字符串：以空格为分隔符，把空格字符替换成'\0'字符，将字符串 code 解析成字符串数组 argv.

```
char code[MAX_BUFFLEN];
char *argv[MAX_NUM];    // no more than 100 arguments
int count = 0;      // N.O. of arguments
char *next = NULL;
char *rest = code;
argv[count++] = code;

while(next = strchr(rest, ' '))
{
    next[0] = '\0';
    rest = next + 1;
    if(rest[0] != '\0' && rest[0] != ' ')
        argv[count++] = rest;
    if(count + 2 > MAX_NUM)
        return 127;
}
argv[count++] = NULL;
```

在子进程中用 execvp 装入命令和 argv。

```
int pid;
pid = fork();
if(pid == 0)
{
    int error = execvp(code, argv);
    if(error < 0)
    {
        perror("execvp");
        return 127;
    }
    else
        return 0;
}
```

②无需解析出命令和参数，直接用 execl 调用/bin/sh 执行命令。

```
if (pid == 0)
```

```
2.  {
3.      execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
4.      exit(127);
5.  }
```

测试程序:

```
1.  int main()
2.  {
3.      mysys("pwd");
4.      mysys("echo ,HELLO  WORLD ,  sdfa sdfadf        ss    ");
5.      mysys("echo /G");
6.      mysys("echo ,,");
7.      mysys("echo");
8.      mysys("asdfasdf");
9.      printf("--------------------------------------------------------\n");
10.     mysys("echo HELLO WORLD");
11.     printf("--------------------------------------------------------\n");
12.     mysys("ls /");
13.     printf("--------------------------------------------------------\n");
14.
15.     return 0;
16. }
```

实验结果:

```
root@Lue:~/os_experiment/program_task/multi_process# cc mysys.c -o mysys
root@Lue:~/os_experiment/program_task/multi_process# ./mysys
/root/os_experiment/program_task/multi_process
,HELLO WORLD , sdfa sdfadf ss
/G
,,

execvp: No such file or directory
------------------------------------------------------------
HELLO WORLD
------------------------------------------------------------
bin   etc         initrd.img.old  lib64       mnt   root  srv  usr      vmlinuz.old
boot  home        lib             lost+found  opt   run   sys  var
dev   initrd.img  lib32           media       proc  sbin  tmp  vmlinuz
------------------------------------------------------------
------------------------------------------------------------
HELLO WORLD
------------------------------------------------------------
bin   etc         initrd.img.old  lib64       mnt   root  srv  usr      vmlinuz.old
boot  home        lib             lost+found  opt   run   sys  var
dev   initrd.img  lib32           media       proc  sbin  tmp  vmlinuz
------------------------------------------------------------
```

题二：sh1.c

题目要求:

1. 实现 shell 程序，要求具备如下功能

2. 支持命令参数

```
$ echo arg1 arg2 arg3
$ ls /bin /usr/bin /home
```

3. 实现内置命令 cd、pwd、exit

```
$ cd /bin
$ pwd
/bin
```

实现思路:

①使用一个循环来产生一个可交互的终端，对输入的命令解释执行并输出相应结果。

```
1.  char buff[MAX_BUFFLEN];
2. printf("[%s]$ ", dir);
3. while(gets(buff))
4. {
5.     int res = judge_buff(buff);
6.     if(res == 0)
7.         mysys(buff);
8.     else if(res == 1)
9.         cd(buff);
10.    else if(res == 2)
11.        return 0;
12.    printf("[%s]$ ", dir);
13. }
```

②使用 judeg_buff(char *)函数来判断输入的命令是否是内置命令。若是 cd 则返回 1，若是 exit 则返回 2，否则返回 0。

```
1. int judge_buff(char *buff)
2. {
3.     char code[MAX_BUFFLEN];
4.     strcpy(code, buff);
5.     char *next = strchr(code, ' ');
6.     if(next != NULL)
7.         next[0] = '\0';
8.     if(strcmp(code, "cd") == 0)
9.         return 1;
10.    else if(strcmp(code, "exit") == 0)
11.        return 2;
12.    else
13.        return 0;
14. }
```

③若不是内置命令则将命令送入 mysys 函数执行，若是 cd 则送入 cd 函数执行，若是 exit 则程序退出。函数 cd(char *)判断参数，若存在参数，则调用 chdir 函数进入路径，

若不存在参数（参数为空格），则调用 chdir 进入 home 目录。

```
// count==2 意味着命令本身+NULL，若有参数存在，则 count 应大于 2
if(count == 2)
{
    chdir(home);
    dir = getcwd(NULL, 0);
}
else
{
    int res = chdir(argv[count - 2]);
    dir = getcwd(NULL, 0);
    if(res == -1)
    {
        printf("cd: No such path %s\n", argv[count - 2]);
        return -1;
    }
}
```

实验结果：

```
root@Lue:~/os_experiment/program_task/multi_process# ./sh1
[/root/os_experiment/program_task/multi_process]$ echo arg1 arg2 arg3
arg1 arg2 arg3
[/root/os_experiment/program_task/multi_process]$ ls /root
163music  163music2  animating-pikachu  os_experiment  package-lock.json  signUpAndLogIn  test  todo
[/root/os_experiment/program_task/multi_process]$ cd /bin
[/bin]$ pwd
/bin
[/bin]$ exit
```

题三：sh2.c

题目要求：

1. 实现 shell 程序，要求在第 1 版的基础上，添加如下功能
2. 实现文件重定向

```
$ echo hello >log
$ cat log
hello
```

实现思路：

①同样使用 while 循环来实现可交互的终端，print_prefix 函数用于打印前缀。函数 go_dup 用于解释执行命令，支持文件重定向。

```
//in main
    print_prefix();
    while(gets(buff))
    {   go_dup(buff);
        print_prefix();  }
```

```
6.
7.  void print_prefix()
8.  {
9.      if(strcmp(home, dir) == 0)
10.         printf("[~]$ ");
11.     else
12.         printf("[%s]$ ", dir);
13. }
```

②go_dup(char *buff)函数中，检查命令中是否含有字符'<'和'>'，若二者都有，则将标准输入和标准输出都重定向到输入文件和输出文件；若只有'<'，则将标准输入重定向到输入文件；若只有'>'，则将标准输出重定向到输出文件。而后调用 mysys 函数执行代码。若二者都无，则不进行重定向，并调用 go 函数执行代码，go 函数里进行了 sh1.c 中内置命令的集成。

```
1.  if(a != NULL && b != NULL)
2.  {
3.      char *in = a + 1 - buff + code;
4.      char *out = b + 1 - buff + code;
5.      strip_char(in, ' ');
6.      strip_char(out, ' ');
7.      int fdin, fdout;
8.      fdin = open(in, O_RDWR, 0666);
9.      fdout = open(out, O_CREAT|O_RDWR, 0666);
10.     if(fdin == -1)
11.     {
12.         printf("File %s open faild\n", in);
13.         return -1;
14.     }
15.     if(fdout == -1)
16.     {
17.         printf("File %s open faild\n", out);
18.         return -1;
19.     }
20.     dup2(fdin, 0);
21.     dup2(fdout, 1);
22.     close(fdin);
23.     close(fdout);
24.     return mysys(code);
25. }
26. else if(a != NULL)
27. {
28.     char *in = a + 1 - buff + code;
29.     strip_char(in, ' ');
30.     int fdin;
```

```
31.     fdin = open(in, O_RDWR, 0666);
32.     dup2(fdin, 0);
33.     close(fdin);
34.     return mysys(code);
35. }
36. else if(b != NULL)
37. {
38.     char *out = b + 1 - buff + code;
39.     strip_char(out, ' ');
40.     int fdout;
41.     fdout = open(out, O_CREAT|O_RDWR, 0666);
42.     dup2(fdout, 1);
43.     close(fdout);
44.     return mysys(code);
45. }
46. else
47. {
48.     return go(buff);
49. }
```

实验结果:

```
root@Lue:~/os_experiment/program_task/multi_process# ./sh2
[/root/os_experiment/program_task/multi_process]$ echo hello > log
[/root/os_experiment/program_task/multi_process]$ cat log
hello
[/root/os_experiment/program_task/multi_process]$ cat <log > newlog
[/root/os_experiment/program_task/multi_process]$ cat newlog
hello
[/root/os_experiment/program_task/multi_process]$ exit
root@Lue:~/os_experiment/program_task/multi_process#
```

题四：sh3.c

题目要求:

1. 实现 shell 程序，要求在第 2 版的基础上，添加如下功能
2. 实现管道

```
$ echo arg1 arg2 arg3
$ ls /bin /usr/bin /home
```

3. 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
```

实现思路:

①同样使用 while 循环来实现可交互的终端，print_prefix 函数用于打印前缀。函数 go_pipe 用于解释执行命令，支持管道和文件重定向。由于管道是半双工的，定义了两个管道，用于子进程中新老管道的交替。每执行一行命令更新一次管道。

```
print_prefix();
while(fgets(buff, sizeof(buff), stdin))
{
    strip(buff);
    go_pipe(buff);
    pipe(fd);
    pipe(fd_tmp);
    print_prefix();
}
```

②函数 go_pipe 中判断字符'|'的个数，若不存在'|',则调用 sh2.c 中的 go_dup 函数。

```
if(count == 1)  // count == 1 表示只存在一个命令，即不存在管道
{
    // recover_in 和 recover_out 用于恢复标准输入和标准输出
    fdin = recover_in;
    fdout = recover_out;
    return go_dup(buff);
}
```

③若字符'|'的个数不为 0，则在子进程之间使用管道。在计算管道个数时，利用 loc 数组存下每个管道前后命令的位置，以方便为每个命令调用一个子进程。用一个 for 循环来创建每一个子进程。标志 flag 初始值为-1，用于告诉当前子进程，前后管道的数量。处于中间的子进程前面有管道，后面也有管道，令 flag=2；第一个子进程只有后面有管道，flag=0，最后一个子进程只有前面有管道，falg=1。

在进入第二个子进程之前，关闭管道的入口（写端）。

若前一个子进程用了两个管道，则在进入下一个子进程之前，抛弃老管道 fd，将临时管道 fd_tmp 设置成当前管道 fd，并关闭当前管道的入口（写端）。

调用 pipe_sys 函数执行子进程。

```
for(int i = 0; i < count; i++)
{
```

```
3.      //printf("[debug] %d pipe: %s\n", i, code+loc[i]);
4.      if(flag == 2)
5.      {
6.          dup2(fd_tmp[0], fd[0]);
7.          dup2(fd_tmp[1], fd[1]);
8.          close(fd_tmp[0]);
9.          close(fd_tmp[1]);
10.         pipe(fd_tmp);
11.         close(fd[1]);
12.     }
13.     if(flag == 0)
14.     {
15.         close(fd[1]);
16.     }
17.     if(i == 0)
18.     {
19.         flag = 0;
20.     }
21.     else if(i == count - 1)
22.     {
23.         flag = 1;
24.     }
25.     else
26.     {
27.         flag = 2;
28.     }
29.     res = pipe_sys(code + loc[i]);
30. }
```

④子进程中根据 flag 判断当前命令前后的管道数量，若前面无管道，则将标准输出重定向到管道入口；若后面无管道，则将标准输入重定向到管道出口；若前后均有管道，则将标准输入重定向到当前管道，将标准输出重定向到临时短道。

```
1.  int pipe_sys(const char *cmdstring)
2.  {
3.      pid_t pid;
4.
5.      pid = fork();
6.      if (pid == 0)
7.      {
8.          if(flag == 0)
9.          {
10.             dup2(fd[1], 1);
11.             close(fd[0]);
12.             close(fd[1]);
```

```
            execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
            exit(127);
        }
        else if(flag == 1)
        {
            dup2(fd[0], 0);
            close(fd[0]);
            close(fd[1]);
            execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
            exit(127);
        }
        else if(flag == 2)
        {
            dup2(fd[0], 0);
            close(fd[0]);
            close(fd[1]);
            // 输出进入临时管道
            dup2(fd_tmp[1], 1);
            close(fd_tmp[0]);
            close(fd_tmp[1]);
            execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
            exit(127);
        }
    }
    wait(NULL);
    return 0;
}
```

实验结果:

```
root@Lue:~/os experiment/program task/multi process# cc sh3.c -o sh3
root@Lue:~/os experiment/program task/multi process# ./sh3
> multi_process echo This is my great shell.
This is my great shell.
> multi_process ls
a.out  input.txt  log  mysys  mysys.c  mysys2.c  newlog  output.txt  sh1  sh1.c  sh2  sh2.c  sh3  sh3.c
> multi_process cat input.txt | wc -l
6
> multi_process rm output.txt
> multi_process cat <input.txt | sort | uniq | cat >output.txt
> multi_process cat output.txt
1
2
3
> multi_process echo Let\'s go home.
Let's go home.
> multi_process cd
> ~ ls
163music  163music2  animating-pikachu  os experiment  package-lock.json  signUpAndLogIn  test  todo
> ~ echo This is my home.
This is my home.
> ~ cd /bin
> bin pwd
/bin
> bin echo Goodbye!
Goodbye!
> bin exit
```

多线程题目

题一：pi1.c

题目要求：

1. 使用 2 个线程根据莱布尼兹级数计算 PI
2. 莱布尼兹级数公式：1 - 1/3 + 1/5 - 1/7 + 1/9 - ... = PI/4
3. 主线程创建 1 个辅助线程
4. 主线程计算级数的前半部分
5. 辅助线程计算级数的后半部分
6. 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

实现思路：

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。

```
1.  int main()
2.  {
3.      pthread_t worker_tid;
4.      float total;
5.
6.      pthread_create(&worker_tid, NULL, worker, NULL);
7.      master();
8.      pthread_join(worker_tid, NULL);
9.      total = worker_output + master_output;
10.     printf("PI = %.10f\n", total * 4);
11.     return 0;
12. }
```

②worker 函数和 master 函数分别计算级数的后半段和前半段。

```
1.   void *worker(void *arg)
2.  {
3.      int i;
4.      for(i = N / 2; i < N; i++)
5.          worker_output += (float)sign(i) / (2*i + 1);
6.
7.      printf("worker_output = %.10f\n", worker_output);
8.      return NULL;
9.  }
10.
11. void master()
```

```
{
    for(int i = 0; i < N / 2; i++)
        master_output += (float)sign(i) / (2*i + 1);

    printf("master_output = %.10f\n", master_output);
    return;
}
```

实验结果：

```
➜  multi_thread git:(master) X cc pi1.c -lpthread  -o pi1
➜  multi_thread git:(master) X ./pi1
master_output = 0.7853984833
worker_output = 0.0000002499
PI = 3.1415948868
```

题二：pi2.c

题目要求：

1. 使用 N 个线程根据莱布尼兹级数计算 PI
2. 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
3. 主线程创建 N 个辅助线程
4. 每个辅助线程计算一部分任务，并将结果返回
5. 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

实现思路：

①主线程采用 for 循环产生 N 个线程来计算 PI。每个线程计算 1/N 的部分，计算起止点作为参数传入辅助线程的线程入口函数。

```
// 参数结构体的定义
typedef struct param {
    int start;
    int end;
}Param;
//返回值结构体的定义
typedef struct result {
    float sum;
}Result;
```

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。

```
int main()
{
    pthread_t workers[NR_CPU];
    Param params[NR_CPU];
```

```
5.      float total = 0;
6.
7.      for(int i = 0; i < NR_CPU; i++)
8.      {
9.          Param *param;
10.         param = ¶ms[i];
11.         param->start = i * NR_CHILD;
12.         param->end = (i + 1) * NR_CHILD;
13.         pthread_create(&workers[i], NULL, compute, param);
14.     }
15.
16.     for(int i = 0; i < NR_CPU; i++)
17.     {
18.         Result *result;
19.         pthread_join(workers[i], (void **)&result);
20.         total += result->sum;
21.         free(result);
22.     }
23.
24.     printf("PI = %.10f\n", total * 4);
25.     return 0;
26. }
```

实验结果:

```
➜  multi_thread git:(master) ✗ cc pi2.c -lpthread -o pi2
➜  multi_thread git:(master) ✗ ./pi2
worker 6 = 0.0000000476
worker 7 = 0.0000000357
worker 5 = 0.0000000667
worker 4 = 0.0000000999
worker 3 = 0.0000001667
worker 2 = 0.0000003332
worker 1 = 0.0000010000
worker 0 = 0.7853971124
PI = 3.1415958405
```

题三: sort.c

题目要求:

1. 多线程排序
2. 主线程创建一个辅助线程
3. 主线程使用选择排序算法对数组的前半部分排序
4. 辅助线程使用选择排序算法对数组的后半部分排序
5. 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

实现思路:

①主线程采用 pthread_create 函数创建辅助线程，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。线程入口函数采用冒泡排序。

```
int main()
{
    generate_nums();
    show_nums(nums, "unsort");

    pthread_t worker_tid;
    Param params[2];
    params[0].start = 0;
    params[0].end = NUMMAX / 2;
    params[1].start = NUMMAX / 2;
    params[1].end = NUMMAX;

    pthread_create(&worker_tid, NULL, sort, &params[1]);
    sort(&params [0]);

    pthread_join(worker_tid, NULL);
    merge(0, NUMMAX / 2 - 1, NUMMAX - 1);

    show_nums(nums, "sorted");
    return 0;
}
```

②使用 pthread_join 函数等待辅助线程结束。使用 merge 函数归并数组的前半部分和后半部分。

```
void merge(const int left, const int mid, const int right)
{
    int temp[NUMMAX];
    memcpy(temp, nums, NUMMAX * sizeof(int));
    int s1 = left;
    int s2 = mid + 1;
    int t = left;
    while(s1 <= mid && s2 <= right)
    {
        if(temp[s1] < temp[s2])
            nums[t++] = temp[s1++];
        else
            nums[t++] = temp[s2++];
    }
    while(s1 <= mid)
        nums[t++] = temp[s1++];
```

```
    while(s2 <= right)
        nums[t++] = temp[s2++];
}
```

实验结果：

```
➜  multi_thread git:(master) ✗ cc sort.c -lpthread -o sort
➜  multi_thread git:(master) ✗ ./sort
[unsort]
          4992  8256  7412  1893  8682  3681  2862   577  4677  2829
          2429  1691  7323  1090  1421   401  5446  3821  9713  1882
          2354   399  6087  2737  3509  4770  3134  3818  2462  8662
          8439  3807  6918  5851  5700  1952  5884  4915  8882   562
          7744  1311  2253  1419  8753    26  1820   551  3848  7886
          2433  2554  4637  4873  5292  8146  9643  4778  8317  8458
          9792  6756  2265  6710  2608  4317  5014  8492  9232   248
          5406  6977  7912  7659  4748  6665  4038  6569  7217  7886
           807  6002  6792  1796   875  2084  6295   519  3214  4612
          8977  3006  1368  7594  9716  3976  1911  1083  8821  7496
[sorted]
            26   248   399   401   519   551   562   577   807   875
          1083  1090  1311  1368  1419  1421  1691  1796  1820  1882
          1893  1911  1952  2084  2253  2265  2354  2429  2433  2462
          2554  2608  2737  2829  2862  3006  3134  3214  3509  3681
          3807  3818  3821  3848  3976  4038  4317  4612  4637  4677
          4748  4770  4778  4873  4915  4992  5014  5292  5406  5446
          5700  5851  5884  6002  6087  6295  6569  6665  6710  6756
          6792  6918  6977  7217  7323  7412  7496  7594  7659  7744
          7886  7886  7912  8146  8256  8317  8439  8458  8492  8662
          8682  8753  8821  8882  8977  9232  9643  9713  9716  9792
```

题四：pc1.c

题目要求：

1. 使用条件变量解决生产者、计算者、消费者问题
2. 系统中有 3 个线程：生产者、计算者、消费者
3. 系统中有 2 个容量为 4 的缓冲区：buffer1、buffer2
4. 生产者生产'a'、'b'、'c'、'd'、'e'、'f'、'g'、'h'八个字符，放入到 buffer1
5. 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
6. 消费者从 buffer2 取出字符，将其打印到屏幕上

实现思路：

①这道题可以被分解为两个生产者-消费者问题，计算者同时承担两个角色。计算者先作为 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)
{
    char item;
    for(int 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);
        pthread_cond_signal(&wait_empty_buffer1);
        pthread_mutex_unlock(&mutex1);

        item += 'A' - 'a';

        pthread_mutex_lock(&mutex2);
        while(buffer_is_full(2))
            pthread_cond_wait(&wait_empty_buffer2, &mutex2);
        put_item(item, 2);
        printf("\033[33m compute item: %c\n\033[0m", item);     //黄色为计算者

        pthread_cond_signal(&wait_full_buffer2);
        pthread_mutex_unlock(&mutex2);

    }
    return NULL;
}
```

生产者作为 buffer1 的生产者，给 buffer1 加锁并存数。

```
void *produce(void *arg)
{
    char item;
    for(int 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("\033[31m produce item: %c\n\033[0m", item);//红色为生产者

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

消费者作为 buffer2 的消费者，给 buffer2 加锁并取数。

```
void *consume(void *arg)
{
    int item;
    for(int 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("\033[34m consume item: %c\n\033[0m", item);     //蓝色为消费者

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

②设置两个互斥量、两个空信号和两个满信号：

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

③创建三个线程分别用于承担生产者、计算者和消费者：

```
int main()
{
    pthread_t producer_tid, computer_tid, consumer_tid;

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

    pthread_create(&producer_tid, NULL, produce, NULL);
    pthread_create(&computer_tid, NULL, compute, NULL);
    pthread_create(&consumer_tid, NULL, consume, NULL);

```

```
    pthread_join(producer_tid, NULL);
    pthread_join(computer_tid, NULL);
    pthread_join(consumer_tid, NULL);

    return 0;
}
```

实验结果:
红色为生产者，黄色为计算者，蓝色为消费者。

```
➜  multi_thread git:(master) X cc pc1.c -lpthread -o pc1
➜  multi_thread git:(master) X ./pc1
produce item: a
produce item: b
produce item: c
compute item: A
compute item: B
compute item: C
consume item: A
consume item: B
consume item: C
produce item: d
produce item: e
produce item: f
compute item: D
compute item: E
compute item: F
consume item: D
consume item: E
consume item: F
produce item: g
produce item: h
compute item: G
compute item: H
consume item: G
consume item: H
```

题五：pc2.c

题目要求:

1. 使用信号量解决生产者、计算者、消费者问题
2. 功能和前面的实验相同，使用信号量解决

实现思路:

①这道题与上一题思路相同，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;
```

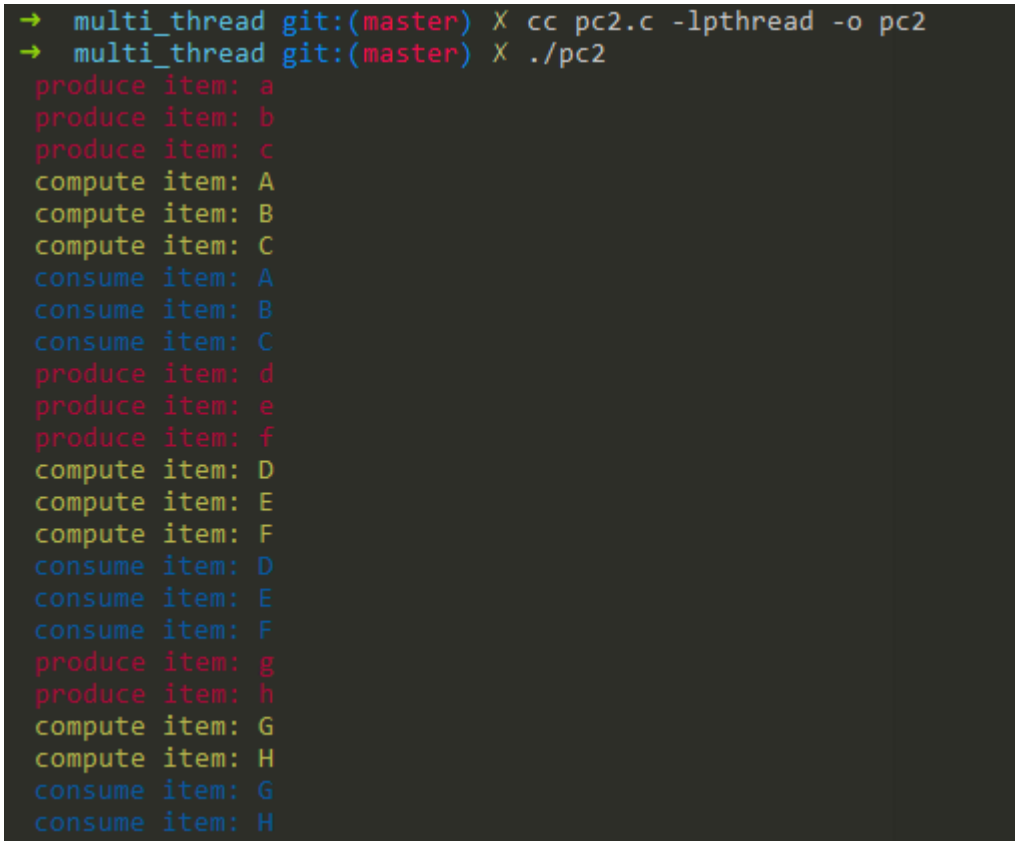
```
    pthread_mutex_t mutex;
    pthread_cond_t cond;
}sema_t;

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);
    int i = 1;
    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 += 1;
    pthread_cond_signal(&sema->cond);
    pthread_mutex_unlock(&sema->mutex);
}
```

②同样创建三个线程分别用于承担生产者、计算者和消费者，empty_buffer_sema1 和 empty_buffer_sema2 信号量初始值给为 buffer 容量-1 的大小，表示初始时 buffer1 和 buffer2 为空；full_buffer_sema1 和 full_buffer_sema2 信号量初始值设为 0，表示 buffer1 和 buffer2 初始时都没有元素供取出。

```
int main()
{
    pthread_t producer_tid, computer_tid, consumer_tid;

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

```
    sema_init(&full_buffer_sema2, 0);

    pthread_create(&producer_tid, NULL, produce, NULL);
    pthread_create(&computer_tid, NULL, compute, NULL);
    pthread_create(&consumer_tid, NULL, consume, NULL);

    pthread_join(producer_tid, NULL);
    pthread_join(computer_tid, NULL);
    pthread_join(consumer_tid, NULL);

    return 0;
}
```

实验结果：
红色为生产者，黄色为计算者，蓝色为消费者。



题六：ring.c

题目要求：

1. 创建 N 个线程，它们构成一个环
2. 创建 N 个线程：T1、T2、T3、… TN
3. T1 向 T2 发送整数 1

4. T2 收到后将整数加 1
5. T2 向 T3 发送整数 2
6. T3 收到后将整数加 1
7. T3 向 T4 发送整数 3
8. …
9. TN 收到后将整数加 1
10. TN 向 T1 发送整数 N

实现思路：

①创建 N 个缓冲区 B，分别对应着 N 个线程的接受信号口。T1 给 B2 存入整数 1，并释放 T2 已满的信号，T2 发现 B2 满了或者收到 B2 已满的信号便从 T2 取数，T3 如上。T100 取数后，将数加一送入 B1，此时等待已久的 T1 收到 T100 释放出的 B1 已满的信号，便从 B1 中取数。

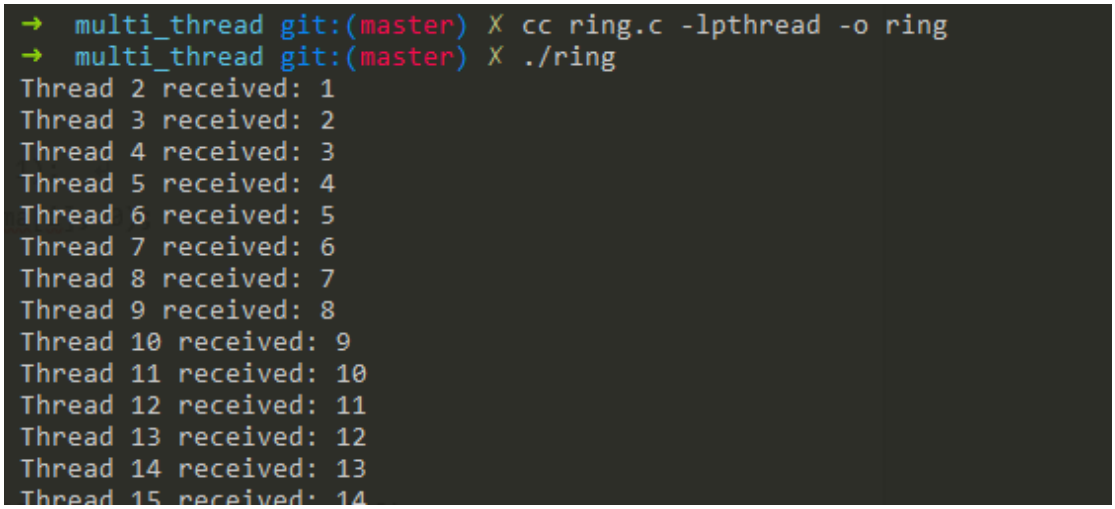
```
sema_t mutex_sema[N];
sema_t full_buffer_sema[N];

void *add(void *arg)
{
    int receive;
    Param *param = (Param *)arg;
    int order = param->order;
    if(order == 0)
    {
        sema_wait(&mutex_sema[order + 1]);
        buff[order + 1] = 1;
        sema_signal(&mutex_sema[order + 1]);
        sema_signal(&full_buffer_sema[order + 1]);

        sema_wait(&full_buffer_sema[order]);
        sema_wait(&mutex_sema[order]);
        receive = buff[order];
        printf("Thread %d received: %d\n", order + 1, receive);
        sema_signal(&mutex_sema[order]);
    }
    else if(order == N - 1)
    {
        sema_wait(&full_buffer_sema[order]);
        sema_wait(&mutex_sema[order]);
        receive = buff[order];
        printf("Thread %d received: %d\n", order + 1, receive);
        sema_signal(&mutex_sema[order]);

        sema_wait(&mutex_sema[0]);
        buff[0] = receive + 1;
        sema_signal(&mutex_sema[0]);
```

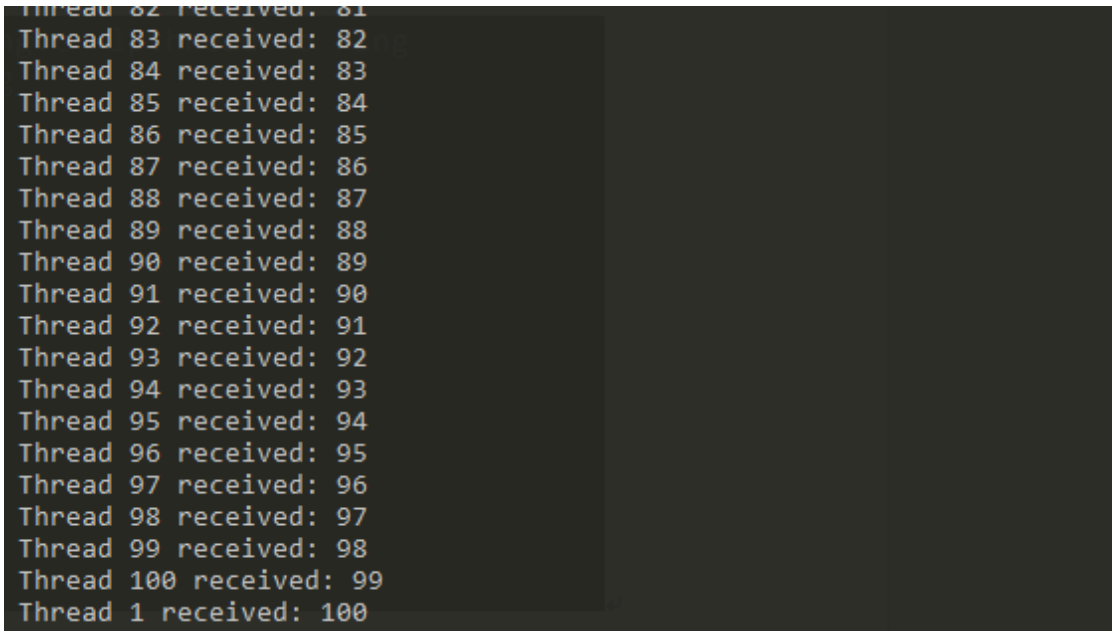
```
        sema_signal(&full_buffer_sema[0]);
    }
    else
    {
        sema_wait(&full_buffer_sema[order]);
        sema_wait(&mutex_sema[order]);
        receive = buff[order];
        printf("Thread %d received: %d\n", order + 1, receive);
        sema_signal(&mutex_sema[order]);

        sema_wait(&mutex_sema[order + 1]);
        buff[order + 1] = receive + 1;
        sema_signal(&mutex_sema[order + 1]);
        sema_signal(&full_buffer_sema[order + 1]);
    }
}
```

②主线程初始化 N 个互斥量和 N 个 full_buffer_sema 信号量，创建 N 个线程，并等待 N 个线程结束。线程初始化时，传递参数给线程入口函数，告诉新创建的线程它是第几个线程。

```
int main()
{
    pthread_t ring_tid[N];
    Param params[N];
    for(int i = 0; i < N; i++)
    {
        sema_init(&mutex_sema[i], 1);
        sema_init(&full_buffer_sema[i], 0);
    }

    for(int i = 0; i < N; i++)
    {
        params[i].order = i;
        pthread_create(&ring_tid[i], NULL, add, &params[i]);
    }

    for(int i = 0; i < N; i++)
        pthread_join(ring_tid[i], NULL);

    return 0;
}
```

实验结果:
此处 N 设置为 100.


→ multi_thread git:(master) X cc ring.c -lpthread -o ring
→ multi_thread git:(master) X ./ring
Thread 2 received: 1
Thread 3 received: 2
Thread 4 received: 3
Thread 5 received: 4
Thread 6 received: 5
Thread 7 received: 6
Thread 8 received: 7
Thread 9 received: 8
Thread 10 received: 9
Thread 11 received: 10
Thread 12 received: 11
Thread 13 received: 12
Thread 14 received: 13


Thread 83 received: 82
Thread 84 received: 83
Thread 85 received: 84
Thread 86 received: 85
Thread 87 received: 86
Thread 88 received: 87
Thread 89 received: 88
Thread 90 received: 89
Thread 91 received: 90
Thread 92 received: 91
Thread 93 received: 92
Thread 94 received: 93
Thread 95 received: 94
Thread 96 received: 95
Thread 97 received: 96
Thread 98 received: 97
Thread 99 received: 98
Thread 100 received: 99
Thread 1 received: 100