操作系统实践实验报告

051720205 岳雨涵 1617101班

1 文件读写编程题目

1.1 myecho.c

功能

接受命令行参数，并将参数打印出来。

实现

在主函数进行参数传递，argc为输入的命令条数，argv为字符串。

当终端运行myecho.o时，就会把从命令行键入的参数传递到argc和argv[]，因此在主函数中直接输出参数即可。

```
int main(int argc, char *argv[])
{
  int i;
  for(i = 1; i < argc; i++)
    printf( "%s ", argv[i]);//输出键入的参数即可
  printf("\n");
  return 0;
}
```

运行结果

```
(base) yuhan@yyhdeMacBook-Pro 文件读写编程题目 % ./myecho.o a bc d
a bc d
```

1.2 mycat.c

功能

mycat将指定的文件内容输出到屏幕。

实现

同样在主函数中实现参数传递，argc为输入的命令个数，argv为要打开的文件名。

因为cat的命令行应形如 **$ cat /etc/passwd**，因此argc至少为2；若小于2则输入错误，给出提示并退出程序。

```
    int main(){
      if(argc < 2){
        printf("INPUT ERROR!\n");
        exit(-1);
       }
      ...
    }
```

使用open函数打开文件，函数原型定义为

```
int open(const char *pathname, int flags);
```

flags指定打开参数，可用位或的方式进行组合。

在本题中的权限应为只读O_RDONLY，要注意加上0666代表打开权限。

```
int main(){
    ...
    int fp;
    fp = open(argv[1], O_RDONLY|O_CREAT);//返回的是文件描述符
    if (fp == -1) {
        printf("FILE OPEN ERROR!\n");
        exit(-1);
    }
    ...
}
```

定义缓冲区，使用read函数循环读取文件内容至缓冲区中。

```
int main(){
    ...
    char bf[BUFFERSIZE];
    int i =0;
    while((i = read(fp,bf,BUFFERSIZE))>0){
        if(write(STDOUT_FILENO, bf, i) != i)//判断
            printf("PRINT ERROR!\n");
    }
    ...
}
```

最后关闭文件。

运行结果

```
(base) yuhan@yyhdeMacBook-Pro 文件读写编程题目 % ./mycat.o test.txt
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 mycp.c

功能

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

实现

创建内存缓冲区和文件描述符fpR（源文件）,fpD（目标文件）。

```
    int fpR=0;
    int fpD=0;
    char bf[BUFFERSIZE];//BUFFERSIZE定为4096
```

使用open函数打开源文件，使用open函数创建目的文件。

```
int main(){
  ...
  if((fpR = open(argv[1], O_RDONLY, 0666)) == -1){//源文件以只读方式打开
      printf("CANNOT OPEN\n");
      exit(1);
  }

  if((fpD = open(argv[2],O_RDWR | O_CREAT, 0666)) == -1){
      //目的文件以读写方式打开，若输入的文件名不存在则创建新文件
      printf("CANNOT CREATE\n");
      exit(1);
  }
  ...
}
```

循环读取源文件中的数据到内存缓冲区中。read函数返回的是读取字节数，返回0则代表读到末尾；再用write函数写入到目的文件中。

```
    while((i = read(fpR, bf,BUFFERSIZE)) > 0){
        if(write(fpD, bf, i) != i){
            printf("WRITE ERROR\n");
            exit(1);
        }
    }
```

最后要注意关闭文件！

```
    close(fpR);
    close(fpD);
```

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

功能

- 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
- 执行测试程序

实现

在mysys函数中通过`fork()`创建子进程，在`mysys()`函数中创建一个新进程，调用execl函数执行命令,该函数会将子进程的空间全部替换为参数指定的路径的程序，并执行参数对应的命令。

父进程等待`wait()`等待子进程结束。

```
void mysys(char *command){
    pid_t pid;//进程参数
    if((pid = fork()) == 0){//创建一个子进程并判断当前是父进程还是子进程
        execl("/bin/sh","sh","-c",command,NULL);
    }
    //等待子进程结束
    wait(NULL);
}
```

测试程序：

```
int main()
{
    printf("----------------------------------------------------\n");
    mysys("echo HELLO WORLD");
    printf("----------------------------------------------------\n");
    mysys("ls /");
    printf("----------------------------------------------------\n");
    return 0;
}
```

运行结果

```
(base) yuhan@yyhdeMacBook-Pro multi-proceess % ./mysys.o
----------------------------------------------------
HELLO WORLD
----------------------------------------------------
Applications  Volumes   etc   sbin
Library   bin   home    tmp
System    cores   opt   usr
Users   dev   private   var
----------------------------------------------------
```

注：在完成这个简单的mysys时，没有考虑到键入命令后字符串分割等问题，将在sh3中具体阐述。

2.2 sh3.c: 实现shell程序

功能

- 支持命令参数
- 实现内置命令cd、pwd、exit
- 实现文件重定向

- 实现管道
- 只要求连接两个命令，不要求连接多个命令（选做）
- 不要求同时处理管道和重定向（选做）

实现

首先定义所有用到的函数：

```
// 读入一行进入缓冲区
int readLine(char *, int);

// 实现system函数
int mysys(char *);

// 判断是否要重定向
int relocCheck(char *);

// 执行指令
void runPipe(char *);//管道指令
void deal(char *);//分情况对不同的命令进行处理

// 处理字符串中的输出重定向
char *dealReOutStr(char *);

// 处理字符串中的输入重定向
char *dealReInStr(char *);

// 删除字符串的一部分
void deleteFromStr(char *, int, int);
// 将字符串分成两半
char *split2(char *, int);
```

总体框架：

1. 在主循环中循环读取命令，使用`readLine()`函数将字符串逐个字符读取到缓冲区中，在缓冲区最后加入`'\0'`表示字符串结束，避免内存泄漏问题。
2. 接着`deal()`函数对buffer数组中存放的命令进行解析。
3. `deal()`函数对命令进行分类和集中处理，首先判断指令类型，若无重定向、无管道命令（即无|和<>符号），则当作普通的命令调用`mysys()`处理或直接处理cd、pwd和exit命令；若为管道命令，则调用`runPipe()`函数；若为重定向命令，则先获取文件名称，再进行操作。（实现过程后面详细阐述）

主函数框架

```
int main(int argc, char *argv[]) {
    char buffer[BUFFER_LEN];

    // 开始主循环
    while (1) {
        // 输出shell标识符
        printf("$");

        if (readLine(buffer, BUFFER_LEN)) {
            // 执行指令
            deal(buffer);
        } else {
            printf("ERROR! \n");
        }
    }

    return 0;
}
```

系统命令实现

系统命令调用`mysys()`函数，在子进程中直接用execl调用/bin/sh执行命令，再回到父进程。

```
int mysys(char *arg) {
    // fork一个子进程
    pid_t fpid = fork();
    if (fpid < 0) {
        // 如果获取子进程失败
        return -1;
    } else if (fpid == 0) {
        // 如果是子进程
        if (execl("/bin/sh", "sh", "-c", arg, (char *) 0) < 0) {
            return 127;
        }
    } else {
        // 等待子进程结束
        waitpid(fpid, NULL, 0);
        return 0;
    }
    return 0;
}
```

内置命令cd、pwd、exit实现

cd：chdir函数改变工作目录

```
if (!strcmp(p, "cd")) {//这里p为strtok分割出的第一个单词
    p = strtok(NULL, "");
    if (chdir(p) < 0)//chdir用于改变当前工作目录，其参数为Path目标目录
          printf("no such directory\n");
}
```

pwd：getcwd()

```
else if (!strcmp(p, "pwd")) {
  getcwd(path, BUFFER_LEN);//getcwd()会将当前工作目录的绝对路径复制到参数buffer所指的内存空间中
  printf("%s\n", path);
}
```

exit：直接退出程序即可

```
else if (!strcmp(p, "exit")) {
    exit(0);
}
```

实现输入、输出重定向

在relocCheck中可以得到当前命令属于以下四种情况：

1.输入重定向 `reloc_out`

2.输出重定向 `reloc_in`

3.输入+输出重定向 `reloc_both`

4.无重定向 `reloc_none`

在`deal()`中使用switch-case语句对以上情况分别进行处理。

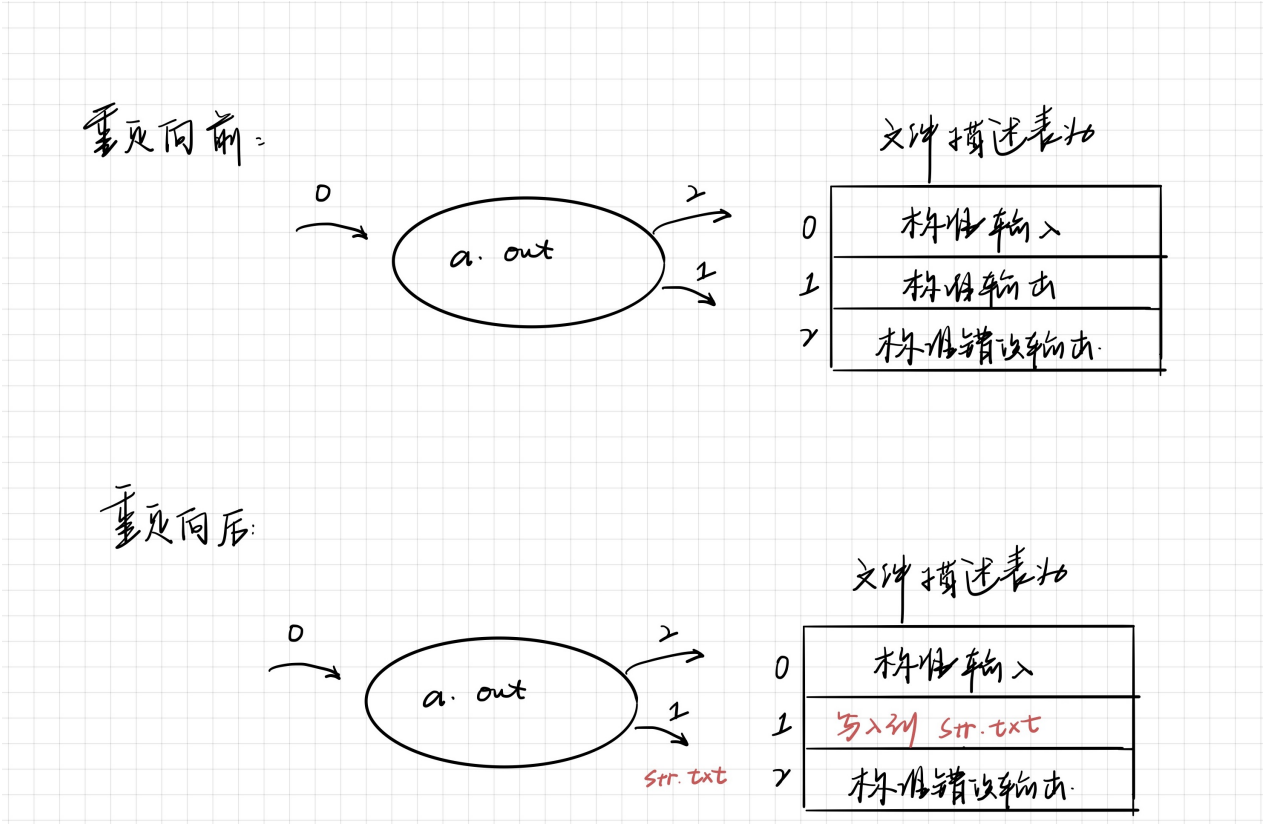
采用的解决方案是在父进程中置位重定向标志，在子进程中执行命令。

输出重定向为例，所谓的I/O重定向也就是让已创建的FD指向其他文件。因此先获取文件名(>后的第一个单词)，再将标准输出/输入重定向指向这个文件。

```
void deal(char *str){
    ...
     switch (type) {
           case reloc_none:
              ...
               break;
```

```
            case reloc_both:
            case reloc_out:
                // 获取重定向文件名
                name = dealReOutStr(str2);
                if (name) {
                    pid = fork();
                    if (pid == 0) {
                        // 执行重定向
                        fd = open(name, O_CREAT | O_RDWR, 0666);
                        dup2(fd, 1);//重定向，文件描述符1为标准输出
                        close(fd);//不再使用fd
                        // 递归
                        deal(str2);
                        exit(0);
                    }
                  else waitpid(pid, NULL, 0);
                }
                break;
            case reloc_in:
                ...
                break;
}
```

重定向的原理如上图，通过dup2()将标准输出重定向到由dealReOutStr()获得的重定向目的文件名字。



管道实现

判断输入的命令中有'|'符号后，则需要调用runPipe()函数.

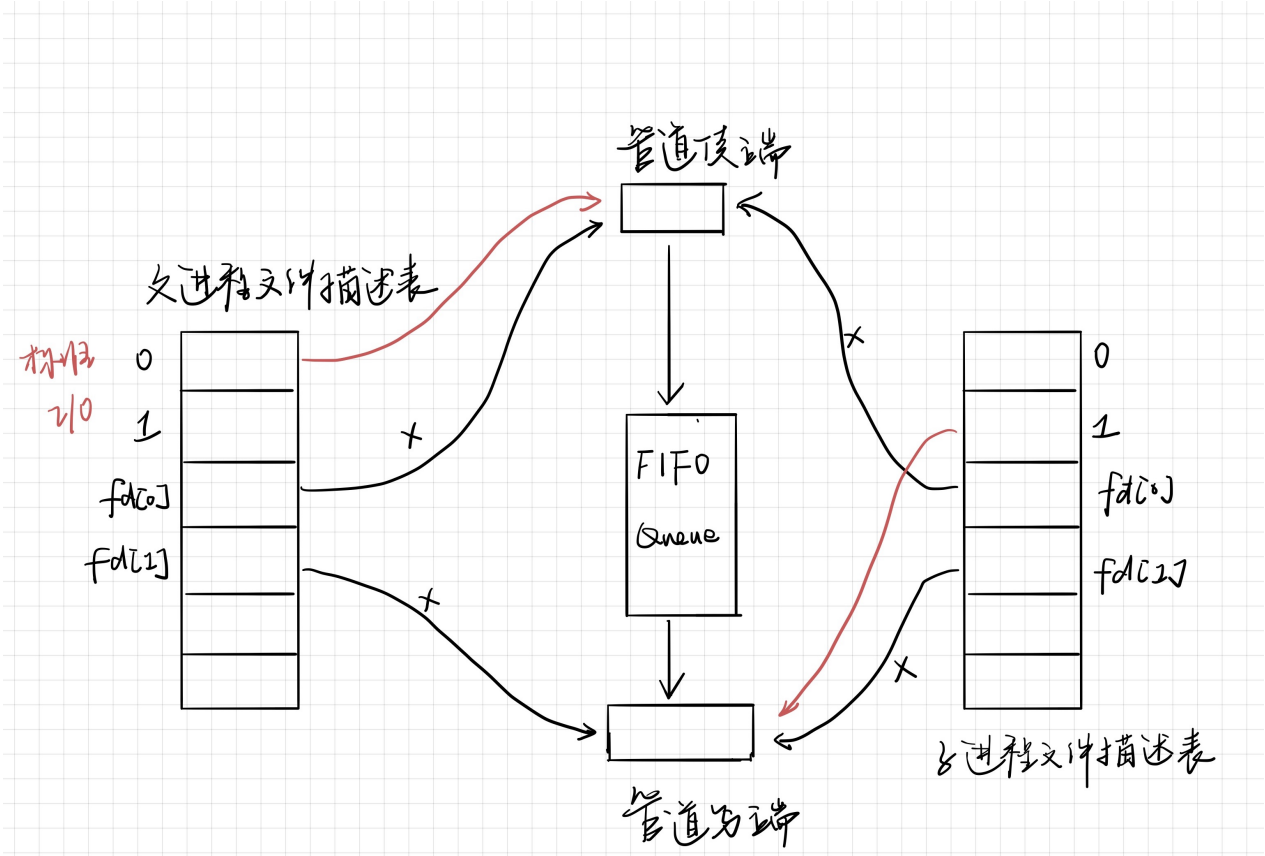
先创建管道，再创建子进程；子进程将继承文件描述符`fd[0]`和`fd[1]`；在子进程中调用`dup2(fd[1], 1)`，用于将标准输出定向到管道写端；子进程用标准输出流将数据发送到父进程。调用deal函数以执行命令。

父进程中将标准输入定向到管道的读端`fd[0]`,关闭`fd[0]`和`fd[1]`，不再使用。父子进程通过管道连接，子进程的标准输出连接到了父进程的标准输入。

连接多个命令：递归，将命令以' | '为界限分为两部分，并对右半边递归执行。

```
void runpipe(){
  pipe(fd);
        pid = fork();
        if (pid == 0) {
            dup2(fd[1], 1);
            close(fd[0]);
            close(fd[1]);
            // 执行指令
            deal(left);
            exit(0);
        }
        else {
            dup2(fd[0], 0);
            close(fd[0]);
            close(fd[1]);
            waitpid(pid, NULL, 0);
            // 递归执行右半边
            runPipe(right);
        }
}
```

管道示意图



为了生成linux中的管道，首先使用pipe()函数得到一对文件描述符，它们是只读文件描述符和只写文件描述符。fork()函数执行之后，子进程会将父进程的数据拷贝一份，同样，子进程也会拥有父进程所有文件描述符的副本。这时在父进程中关闭读文件描述符，只留下写文件描述符；而在子进程则关闭写文件描述符，只留下读文件描述符。当父进程进行写操作而子进程进行读操作时，就相当于两个进程在通信。

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

功能

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

实现

//参考例子1

定义全局变量worker_output和master_output，第一项用于存放worker子线程的计算结果，第二项用于存放主线程的计算结果。

首先，子函数sign用于判断每一项的符号为正还是负：

```
int sign(int n)//注意从0开始算第一项，计算范围为0～N-1
{
    if(n % 2 == 0)//偶数项为正
        return 1;
    else
        return -1;//奇数项为负
}
```

master函数用于创建主线程计算前半部分级数：

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

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

Worker用于创建辅助线程计算后半部分级数：

```
void *worker(void *arg)
{
    int i;
    for(i = N / 2; i < N; i++)
        worker_output += (float)sign(i) / (2*i + 1);

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

主函数中先调用辅助线程，主线程等待其结束后再运行：

```
int main(){
    ...
    pthread_create(&son_tid, NULL, worker, NULL);
    master();
    pthread_join(son_tid, NULL);//等待辅助线程运算完毕
    ...
}
```

使用全局变量存放两个线程的运算结果，并在主函数里相加。

```
total = master_output + worker_output;
printf("PI = %.10f\n", total * 4);
```

关于本题：刚开始和我例子一样使用了数组：先初始化数组元素，再使用两个线程对数组元素进行计算；后来发现不使用数组更为方便，不需要开辟额外空间，因此改为了只判断项的符号，直接进行计算。

3. pi2.c 使用N个线程根据莱布尼兹级数计算PI

功能

- 与上一题类似，但本题更加通用化，能适应N个核心
- 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
- 本题要求 1: 使用线程参数，消除程序中的代码重复
- 本题要求 2: 不能使用全局变量存储线程返回值

实现

//例子2

和pi1不一样的是这里使用局部变量保存线程的参数和计算结果。

首先定义结构体，用struct param描述线程参数，start记录计算范围的起始位置，end记录结束位置；struct result描述计算结果。

```
typedef struct param {//分段计算
    int start;//计算起点
    int end;//计算重点
}Param;

typedef struct result {
    float sum;
}Result;//存储结果
```

利用宏定义1. NR_CPU: 计算辅助线程的个数，假定为2；

2.NR_CHILD: 每个线程需要计算的项的个数；

3.NR_TOTAL: 莱布尼兹级数的精度

```
#define NR_TOTAL 100000
#define NR_CPU    2
#define NR_CHILD (NR_TOTAL/NR_CPU)
```

在主线程中，创建NR_CPU个辅助线程，workers数组保存了每一个工作线程的tid，pramas数组存放每个线程的计算范围：

```
 pthread_t workers[NR_CPU];//workers数组保存了每一个工作线程的tid
 Param params[NR_CPU];
```

对于每一个子线程，给出计算范围，并启动线程：

```
int main(){
    ...
    for(i = 0; i < NR_CPU; i++){
       Param *param;
       param = &params[i];
       param->start = i * NR_CHILD;
       param->end = (i + 1) * NR_CHILD;
       pthread_create(&workers[i], NULL, compute, param);
    }
    ...
}
```

在compute函数中，利用params数组记录的起始和结束位置计算每个辅助线程的计算结果，并存放入result中（给result分配空间），返回result：

```
void *compute(void *arg){
    ...
    for(i = param->start; i < param->end; i++)
        sum += (float)sign(i) / (2 * i + 1);

    //printf("worker %d = %.10f\n", param->start / NR_CHILD, sum);
    result =(Result *)malloc(sizeof(Result));
    result->sum = sum;
    return result;
}
```

再回到主线程，等待所有辅助线程运算结束，将返回的result累加到sum ，并释放sum的空间：

```
for(i = 0; i < NR_CPU; i++)
    {
        Result *result;
        pthread_join(workers[i], (void **)&result);
        sum += result->sum;
        free(result);
    }
```

此时得到的sum的4倍即是PI的值

```
printf("PI = %.10f\n", sum * 4);
```

3.3 sort.c: 多线程排序

功能

- 主线程创建两个辅助线程
- 辅助线程1使用选择排序算法对数组的前半部分排序
- 辅助线程2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子线程的计算结果
- 本题要求 1: 使用线程参数，消除程序中的代码重复

实现

同样用struct param描述线程参数，指定计算范围和数组指针：

```
typedef struct param {//分段计算
    int start;//计算起点
    int end;//计算重点
    int *nums;
}Param;
```

Init_nums函数通过随机生成N个数来初始化数组：

```
void init_nums(){
  for(int i = 0; i < N; i++)
        nums[i] = rand() % 10000;
  ...
}
```

主函数中，先执行init_nums()函数来生成随机数组，接着初始化线程参数，给定两次选择排序的起点和终点：

```
 int main(){
    init_nums();
    pthread_t child[2];
    Param params[2];
    params[0].start = 0;
    params[0].end = N / 2;
    params[1].start = N / 2;
    params[1].end = N;
 }
```

执行副主线程并等待结束：

```
int main(){
  ...
  for(int i=0; i<2; i++){//创建线程辅助运算
     pthread_create(&child[i], NULL, thread_func, &params[i]);
    }

for(int i=0; i<2; i++){ // 等待线程执行完毕
     pthread_join(child[i], NULL);
    }
}
```

thread_func里对传入的params[i]（0或1，因为只有两个线程）进行排序；params结构体给定了每个辅助线程的计算范围，将此计算范围的起点和长度传递给选择排序函数：

```
void *thread_func(void *args) {
  ...
  Param *param = (Param *)args;
    int left = param->start;
    int right = param->end;
     if(left >= right)
         return NULL;
     selectSort(&nums[param->start], N / 2);//选择排序
}
```

最后在主函数中对nums数组的前半段和后半段进行归并排序即可，归并排序的代码这里不再赘述。

运行结果

```
(base) yuhan@yyhdeMacBook-Pro Multi-pthread % ./sort.o
unsorted:  6807 5249 73 3658 8930 1272 7544 878 7923 7709 4440 8165 4492 3042
7987 2503 2327 1729 8840 2612 //未排序
result: 73 878 1272 1729 2327 2503 2612 3042 3658 4440 4492 5249 6807 7544
7709 7923 7987 8165 8840 8930 //排序后
```

3.4 pc1.c: 使用条件变量解决生产者、计算者、消费者问题

功能

- 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

实现

本题分析：

本题和普通生产者-消费者问题的不同之处在于增加了计算者线程，同时缓冲区增加为两个。其实本质上是两个生产者-消费者问题：计算者先作为消费者从buffer1中取出数据，进行计算，再作为生产者向buffer2中存入数据。

因此，需要定义两个缓冲区，两套读写指针，缓冲区使用环形队列实现：

```
#define CAPACITY 4       //缓冲区最大容量
char buffer1[CAPACITY]; //缓冲区数组
char buffer2[CAPACITY];
int in1, in2;            //缓冲区写指针
int out1, out2;          //缓冲区读指针
int size1, size2;        //缓冲区中数据个数
```

缓冲区应该具有的功能有如下四个：判空，此时消费者不能取数；判满，此时生产者不能存入数；以及取数据存数据。

```
//缓冲区操作

int buffer_is_empty(int n);     //缓冲区判空
int buffer_is_full(int n);      //缓冲区判满
char buffer_get(int n);         //从缓冲区中取出一个数据
void buffer_put(char c, int n);//向缓冲区中追加一个数据

int buffer_is_empty(int n)
{
    if(n == 1)
        return size1 == 0;
    else if(n == 2)
        return size2 == 0;
    else
        exit(-1);
}


int buffer_is_full(int n)
{
    if( n == 1)
        return size1 == CAPACITY;
    else if(n == 2)
        return size2 == CAPACITY;
    else
        exit(-1);
}


char buffer_get(int n)
{
    char item;
    if(n == 1)
    {
        item = buffer1[out1];
        out1 = (out1 + 1) % CAPACITY;
        size1--;
```

```
    }
    else if(n == 2)
    {
        item = buffer2[out2];
        out2 = (out2 + 1) % CAPACITY;
        size2--;
    }
    else
        exit(-1);
    return item;
}


void buffer_put(char item, int n)
{
    if(n == 1)
    {
        buffer1[in1] = item;
        in1 = (in1 + 1) % CAPACITY;
        size1++;
    }
    else if(n == 2)
    {
        buffer2[in2] = item;
        in2 = (in2 + 1) % CAPACITY;
        size2++;
    }
    else
        exit(-1);
}
```

生产者-消费者问题分析：

1. **同步关系：**生产者和消费者之间的同步关系即当缓冲区满时，生产者不能进行存放，需等待消费者取出数据；缓冲区空时，消费者要等待生产者放入数据。

   同步关系使用`pthread_cond_wait`和`pthread_cond_signal`解决。当出现缓冲区满或空时的情况，就需要阻塞线程，等待缓冲区的状态发生变化，再唤醒阻塞在条件变量上的其它的生产者或消费者线程。

2. **互斥关系：**当某个生产者执行buffer_is_full、buffer_put时，访问了变量in1、out1和size1，只能允许该生产者独占访问这三个变量，禁止其他生产者和消费者访问这些共享变量。

   当某个消费者执行buffer_is_empty、buffer_get时，访问了变量in2、out2和size2，只能允许该消费者独占访问这三个变量，禁止其他生产者和消费者访问这些共享变量。

   当计算者执行buffer_is_empty和buffer_get时，访问了in1/in2/out1/out2/size1/size2，禁止其他消费者和生产者访问这些共享变量。

互斥关系使用`pthread_mutex_lock`和`pthread_mutex_unlock`解决，访问前先对共享变量上锁，访问结束后解锁。

因此，我们先定义条件变量和互斥量：

```
//互斥量：mutex1用于给buffer1的in1、out1、size1上锁，mutex1用于给buffer2上锁
pthread_mutex_t mutex1, mutex2;

//条件变量：同样分别对应buffer1和2
pthread_cond_t wait_empty_buffer1, wait_empty_buffer2;
pthread_cond_t wait_full_buffer1, wait_full_buffer2;
```

主函数中对条件变量和互斥量初始化：

```
    //初始化互斥量
    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);

    //初始化条件变量
    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);
```

生产者线程：

```
//生产者执行的线程
void *produce(void *arg)
{
    char item;

    for(int i = 0; i < CAPACITY*2; i++)//生产 CAPACITY*2 个数据
    {
        pthread_mutex_lock(&mutex1);

        // 当缓冲区为满时，生产者需要等待
        while(buffer_is_full(1))
            // 当前线程已经持有了mutex，首先释放mutex，然后阻塞，醒来后再次获取mutex
            pthread_cond_wait(&wait_empty_buffer1, &mutex1);

        //此时缓冲区不满
        item = 'a' + i;// a ~ h
```

```
        buffer_put(item, 1);

        // 改变条件变量并且解锁
        pthread_cond_signal(&wait_full_buffer1);
        pthread_mutex_unlock(&mutex1);

        printf("produce item: %c\n", item);


    }
    return NULL;
}
```

消费者线程：

```
//消费者执行的线程
void *consume(void *arg)
{
    int item;
    for(int i = 0; i < CAPACITY*2; i++)//消费 CAPACITY*2 个数据
    {
        pthread_mutex_lock(&mutex2);
        // 当缓冲区为空时，消费者需要等待
        while(buffer_is_empty(2))
            //当前线程已经持有了mutex，首先释放mutex，然后阻塞，醒来后再次获取mutex
            pthread_cond_wait(&wait_full_buffer2, &mutex2);

        // 此时缓冲区非空
        item = buffer_get(2);

        pthread_mutex_unlock(&mutex2);

        // 缓冲区的状态发生了变化，唤醒其它的生产者或消费者
        pthread_cond_signal(&wait_empty_buffer2);

       printf("                consume item: %c\n", item);
    }
    return NULL;
}
```

计算者线程：

```
//计算者执行的线程
void *compute(void *arg)
{
    char item;
    for(int i = 0; i < CAPACITY*2; i++)//计算CAPACITY*2个数据
    {
      //当buffer1为空时，计算者不能取数据
        pthread_mutex_lock(&mutex1);
        while(buffer_is_empty(1))
            pthread_cond_wait(&wait_full_buffer1, &mutex1);
      //从buffer1中取出要计算的数据
        item = buffer_get(1);
      //解锁，唤醒其他进程可以访问共享变量
        pthread_cond_signal(&wait_empty_buffer1);
        pthread_mutex_unlock(&mutex1);

      //进行计算
        item += 'A' - 'a';

      //当buffer2为满时，计算者不能存数据
        pthread_mutex_lock(&mutex2);
        while(buffer_is_full(2))
            pthread_cond_wait(&wait_empty_buffer2, &mutex2);
      //存入计算完毕的数据
        buffer_put(item, 2);
        printf("        compute item: %c\n", item);
      //唤醒，解锁
        pthread_cond_signal(&wait_full_buffer2);
        pthread_mutex_unlock(&mutex2);

    }
    return NULL;
}
```

主函数中分别创建进程和等待进程完成：

```
   int main(){
   ...
   //创建三个线程
    pthread_create(&producer_tid, NULL, produce, NULL);
    pthread_create(&computer_tid, NULL, compute, NULL);
    pthread_create(&consumer_tid, NULL, consume, NULL);

    //等待线程完成
    pthread_join(producer_tid, NULL);
    pthread_join(computer_tid, NULL);
    pthread_join(consumer_tid, NULL);
```

```
    return 0;
}
```

运行结果

```
(base) yuhan@yyhdeMacBook-Pro Multi-pthread % ./pc1.o
produce item: a
produce item: b
produce item: c
produce item: d
produce item: e
        compute item: A
        compute item: B
        compute item: C
        compute item: D
produce item: f
produce item: g
produce item: h
                consume item: A
                consume item: B
        compute item: E
        compute item: F
                consume item: C
                consume item: D
                consume item: E
                consume item: F
        compute item: G
        compute item: H
                consume item: G
                consume item: H
```

3.5 pc2.c: 使用信号量解决生产者、计算者、消费者问题

功能

- 功能和前面的实验相同，使用信号量解决

实现

一个信号量除了初始化外只能通过两个标准原子操作：`wait ()` 和 `signal()` 来访问.

信号量相关定义如下：

```
```

```
//信号量定义
typedef struct {//使用条件变量实现信号量sema_t
    int value;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
}sema_t;

sema_t mutex_sema1, mutex_sema2;                //用于互斥访问缓冲区in1/in2...
sema_t empty_buffer_sema1, empty_buffer_sema2;//用于线程同步
sema_t full_buffer_sema1, full_buffer_sema2;  //用于线程同步


//初始化信号量
void sema_init(sema_t *sema, int value)
{
    sema->value = value;//记录信号量的值
    pthread_mutex_init(&sema->mutex, NULL);
    pthread_cond_init(&sema->cond, NULL);
}
```

信号量的P操作：

```
//如果信号量的值<= 0，则等待条件变量
void sema_wait(sema_t *sema)
{
    pthread_mutex_lock(&sema->mutex);
    while(sema->value <= 0)
    {
        pthread_cond_wait(&sema->cond, &sema->mutex);
    }
    sema->value--;//wait一次信号量值-1
    pthread_mutex_unlock(&sema->mutex);
}
```

信号量的V操作：

```
void sema_signal(sema_t *sema)
{
    pthread_mutex_lock(&sema->mutex);
    sema->value += 1;//signal一次信号量值+1
    pthread_cond_signal(&sema->cond);//唤醒等待条件变量的线程
    pthread_mutex_unlock(&sema->mutex);
}
```

主函数中给定义的六个信号量赋初值：

```
//初始化信号量
    sema_init(&mutex_sema1, 1);
    sema_init(&mutex_sema2, 1);
    sema_init(&empty_buffer_sema1, CAPACITY - 1);//初始值位CAPACITY-1，表buffer1和2都为空
    sema_init(&empty_buffer_sema2, CAPACITY - 1);
    sema_init(&full_buffer_sema1, 0);//初始值设为0，表示没有元素供出
    sema_init(&full_buffer_sema2, 0);
```

生产者线程：

```
//生产者线程
void *produce(void *arg)
{
    char item;
    for(int i = 0; i < CAPACITY * 2; i++)
    {
        sema_wait(&empty_buffer_sema1);//生产者需要一个空的buffer，所以申请信号量empty_buffer_sema
        sema_wait(&mutex_sema1);

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

        sema_signal(&mutex_sema1);//产生一个新的满buffer
        sema_signal(&full_buffer_sema1);//释放
    }
    return NULL;
}
```

计算者线程：

```
//计算者线程
void *compute(void *arg)
{
    char item;
    for(int i = 0; i < CAPACITY * 2; i++)
    {
        //对于buffer1，计算者线程作为消费者
        sema_wait(&full_buffer_sema1);//获取一个buffer1的元素
```

```
        sema_wait(&mutex_sema1);
        item = buffer_get(1);
        sema_signal(&mutex_sema1);
        sema_signal(&empty_buffer_sema1);//释放一个buffer1的位置

        item += 'A' - 'a';

      //对于buffer2，计算者线程作为生产者
        sema_wait(&empty_buffer_sema2);///获取一个buffer2的空位置
        sema_wait(&mutex_sema2);
        buffer_put(item, 2);
        printf("        compute item: %c\n", item);
        sema_signal(&mutex_sema2);
        sema_signal(&full_buffer_sema2);//消耗一个空位置
    }
    return NULL;
}
```

消费者线程：

```
//消费者线程
void *consume(void *arg)
{
    int item;
    for(int i = 0; i < CAPACITY * 2; i++)
    {
        sema_wait(&full_buffer_sema2);//消费者需要一个满的buffer，申请信号量
full_buffer_sema
        sema_wait(&mutex_sema2);//上锁，解决互斥问题

        item = buffer_get(2);//取出数据
        printf("                consume item: %c\n", item);

        sema_signal(&mutex_sema2);//取走数据后，产生一个新的空buffer
        sema_signal(&empty_buffer_sema2);//释放信号量empty_buffer_sema
    }
    return NULL;
}
```

最后同样创建三个线程并等待执行。

```
   int main()
```

```
{
  ...
//创建三个线程
 pthread_create(&producer_tid, NULL, produce, NULL);
 pthread_create(&computer_tid, NULL, compute, NULL);
 pthread_create(&consumer_tid, NULL, consume, NULL);

 //等待线程执行
 pthread_join(producer_tid, NULL);
 pthread_join(computer_tid, NULL);
 pthread_join(consumer_tid, NULL);

 return 0;
 }
```

运行结果

```
(base) yuhan@yyhdeMacBook-Pro Multi-thread % ./pc2.o
produce item: a
produce item: b
produce item: c
        compute item: A
        compute item: B
        compute item: C
produce item: d
produce item: e
produce item: f
                consume item: A
                consume item: B
                consume item: C
produce item: g
        compute item: D
        compute item: E
        compute item: F
                consume item: D
                consume item: E
                consume item: F
produce item: h
        compute item: G
        compute item: H
                consume item: G
                consume item: H
```