1.2016年OS考试题

题目1

1. 主进程创建1个子进程
   - 进程通过管道与子进程连接
   - 子进程的标准输出连接到管道的写端
2. 主进程的标准输入连接到管道的读端
3. 在子进程中调用exec(“echo”, “echo”, “hello world”, NULL)
4. 在父进程中调用read(0, buf, sizeof(buf))，从标准输入中获取子进程发送的字符串，并打印出来

实验代码

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

int main()
{
    int fd[2];
    char buff[32];
    pipe(fd);


    pid_t tid;
    tid = fork();

    if(tid == 0)//children
    {
        dup2(fd[1],1);
        close(fd[0]);
        close(fd[1]);
        execlp("echo","echo","hello wolrd",NULL);
        exit(0);
    }
    else//parent
    {

        dup2(fd[0],0);
        close(fd[0]);
        close(fd[1]);
        int readsize = read(0,buff,sizeof(buff));
        write(1,buff,readsize);
    }
    return 0;
}
```

题目2

1. 主进程创建2个子进程，主进程通过两个管道分别与两个子进程连接
2. 第一个子进程计算从1加到50的和，并将结果通过管道送给父进程
3. 第一个子进程计算从50加到100的和，并将结果通过管道送给父进程
4. 父进程读取两个子进程的结果，将他们相加，打印出来，结果为5050

实验代码

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

int computeResult(int start,int end)
{
    int result = 0;
    for(int i = start;i <= end;i++)
        result += i;
    return result;
}

int main()
{
    pid_t pid1,pid2;
    pid1 = -1;
    pid2 = -1;
    int fd1[2],fd2[2];
    pipe(fd1);
    pipe(fd2);


    pid1 = fork();
    if(pid1 > 0)
    {
        pid2 = fork();
    }
    if(pid1 == 0) //child1
    {
        int result1 =  computeResult(1,50);
        printf("result1:%d\n",result1);
        write(fd1[1],&result1,sizeof(result1));
        exit(1);
    }
    if(pid2 == 0)//child2
    {
        int result2 = computeResult(51,100);
        printf("result23:%d\n",result2);
        write(fd2[1],&result2,sizeof(result2));
        exit(1);
    }
    if(pid1 > 0)
    {
```

```
        int result1,result2;
        read(fd1[0],&result1,sizeof(result1));
        read(fd2[0],&result2,sizeof(result2));

        int result = result1 + result2;
        printf("result:%d\n",result);
    }
    return 0;
}
```

题目3

1. 主线程创建10个子线程 - 第0个子线程计算从01加到10的和 - 第1个子线程计算从11加到20的和 - 第2个子线程计算从21加到30的和 - ... - 第9个子线程计算从91加到100的和
2. 主线程归并10个子线程的计算结果，最终结果为5050
3. 本题必须使用线程参数来完成

实验代码

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#define N 10
struct param{
    int start;
    int end;
};
struct result{
    int sum;
};
void *add(void *arg)
{
    struct param *param = (struct param*) arg;
    struct result *result = malloc(sizeof(struct result));
    result->sum = 0;

    for(int i = param->start;i <= param->end;i++)
        result->sum += i;
    return result;
}
int main()
{
    int i = 0;
    struct param params[N];
    struct result result[N];
    pthread_t tids[N];
    int finalresult = 0;
    for(i = 0;i < N;i++)
    {
```

```
        struct param *param = &params[i];
        struct result *result = malloc(sizeof(struct result));
        param->start = i * 10 + 1;
        param->end = (i + 1) * 10;
        pthread_create(&tids[i],NULL,add,(void *)param);
        pthread_join(tids[i],(void **)&result);
        finalresult += result->sum;
        free(result);
    }
    printf("%d",finalresult);
    //printf("result: %d",result);
    return 0;
}
```

题目4

1. 主线程创建4个子线程T1、T2、T3、T4，主线程在4个子线程退出后，才退出
2. 线程T1、T2、T3、T4的运行时代码如下：

```
#include <unistd.h> // sleep函数声明在该头文件中

void *T1_entry(void *arg)
{
    sleep(2);  // 睡眠2秒，不准删除此条语句，否则答题无效
    puts(“T1”);
}

void *T2_entry(void *arg)
{
    sleep(1);  // 睡眠1秒，不准删除此条语句，否则答题无效
    puts(“T2”);
}

void *T3_entry(void *arg)
{
    sleep(1);  // 睡眠1秒，不准删除此条语句，否则答题无效
    puts(“T3”);
}

void *T4_entry(void *arg)
{
    puts(“T4”);
}
```

3. 使用信号量或者条件变量机制(而不是使用sleep函数)，使得这四个线程满足如下制约关系：
   - T1的print语句执行后，T2和T3才可以执行print语句
   - T2和T3的print语句执行后，T4才可以执行print语句
4. 程序输出结果为

```
T1
T2
T3
T4
```

或者

```
T1
T3
T2
T4
```

实验代码 - 1(调用系统的信号量和条件变量) by LogicJake

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
int t1_gone = 0,t2_gone = 0,t3_gone = 0,t4_gone = 0;
pthread_mutex_t mutex1,mutex2;
pthread_cond_t signal1,signal2;
void *T1_entry(void *arg)
{
    pthread_mutex_lock(&mutex1);

    sleep(2);
    puts("T1");

    t1_gone = 1;
    pthread_cond_broadcast(&signal1);
    pthread_mutex_unlock(&mutex1);

}
void *T2_entry(void *arg)
{
    pthread_mutex_lock(&mutex1);
    while(!t1_gone)
        pthread_cond_wait(&signal1,&mutex1);
    pthread_mutex_unlock(&mutex1);

    sleep(1);
    puts("T2");

    pthread_mutex_lock(&mutex2);
    t2_gone = 1;
    pthread_cond_signal(&signal2);
    pthread_mutex_unlock(&mutex2);
```

```
}
void *T3_entry(void *arg)
{
    pthread_mutex_lock(&mutex1);
    while(!t1_gone)
        pthread_cond_wait(&signal1,&mutex1);
    pthread_mutex_unlock(&mutex1);

    sleep(1);
    puts("T3");

    pthread_mutex_lock(&mutex2);
    t3_gone = 1;
    pthread_cond_signal(&signal2);
    pthread_mutex_unlock(&mutex2);
}
void *T4_entry(void *arg)
{
    pthread_mutex_lock(&mutex2);
    while(!t2_gone || !t3_gone)
        pthread_cond_wait(&signal2,&mutex2);

    puts("T4");

    pthread_mutex_unlock(&mutex2);
}
int main()
{
    pthread_t tids[4];
    pthread_create(&tids[0],0,T1_entry,NULL);
    pthread_create(&tids[1],0,T2_entry,NULL);
    pthread_create(&tids[2],0,T3_entry,NULL);
    pthread_create(&tids[3],0,T4_entry,NULL);

    pthread_join(tids[3],NULL);
}
```

实验代码 - 2(自己实现信号量) - by LogicJake

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

typedef struct {
        int value;
        pthread_mutex_t mutex;
        pthread_cond_t cond;
}sema_t;
```

```
sema_t t1_2_ready;
sema_t t1_3_ready;
sema_t t2_ready;
sema_t t3_ready;


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);
}

void *T1_entry(void *arg)
{
        sleep(2);  // 睡眠2秒，不准删除此条语句，否则答题无效
        puts("T1");
        sema_signal(&t1_2_ready);
        sema_signal(&t1_3_ready);

}

void *T2_entry(void *arg)
{
        sema_wait(&t1_2_ready);
        sleep(1);  // 睡眠1秒，不准删除此条语句，否则答题无效
        puts("T2");
        sema_signal(&t2_ready);
}

void *T3_entry(void *arg)
{
        sema_wait(&t1_3_ready);
        sleep(1);  // 睡眠1秒，不准删除此条语句，否则答题无效
        puts("T3");
        sema_signal(&t3_ready);
```

```
}

void *T4_entry(void *arg)
{
        sema_wait(&t2_ready);
        sema_wait(&t3_ready);
        puts("T4");
}

int main()
{
        sema_init(&t1_2_ready,0);
        sema_init(&t1_3_ready,0);
        sema_init(&t2_ready,0);
        sema_init(&t3_ready,0);

        pthread_t T1,T2,T3,T4;
        pthread_create(&T1, NULL, T1_entry, NULL);
        pthread_create(&T2, NULL, T2_entry, NULL);
        pthread_create(&T3, NULL, T3_entry, NULL);
        pthread_create(&T4, NULL, T4_entry, NULL);
        pthread_join(T1,NULL);
        pthread_join(T2,NULL);
        pthread_join(T3,NULL);
        pthread_join(T4,NULL);
        return 0;
}
```

一些自己写的代码

1. sh1 -- 进程 + 一些字符串操作

sh1 实验思路

- 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

sh1 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) 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);
        }
}
```

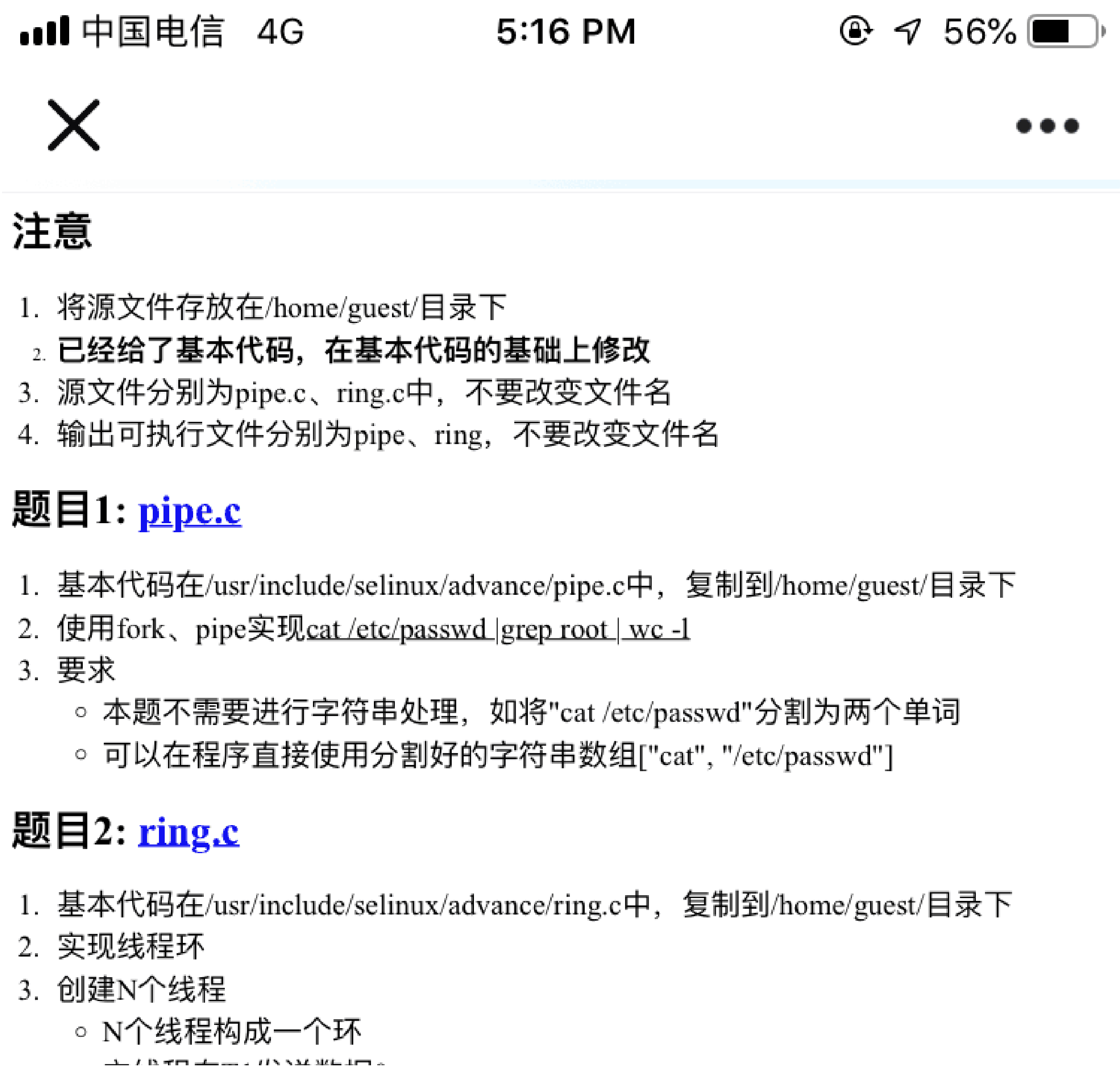
sh1 实验代码

```
#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. ring.c -- 多线程 + 参数传递



注意

1. 将源文件存放在/home/guest/目录下
2. **已经给了基本代码，在基本代码的基础上修改**
3. 源文件分别为pipe.c、ring.c中，不要改变文件名
4. 输出可执行文件分别为pipe、ring，不要改变文件名

题目1: pipe.c

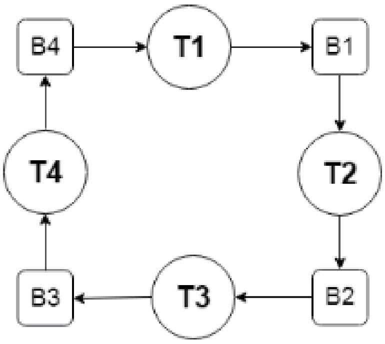
1. 基本代码在/usr/include/selinux/advance/pipe.c中，复制到/home/guest/目录下
2. 使用fork、pipe实现cat /etc/passwd |grep root | wc -l
3. 要求
   - 本题不需要进行字符串处理，如将"cat /etc/passwd"分割为两个单词
   - 可以在程序直接使用分割好的字符串数组["cat", "/etc/passwd"]

题目2: ring.c

1. 基本代码在/usr/include/selinux/advance/ring.c中，复制到/home/guest/目录下
2. 实现线程环
3. 创建N个线程
   - N个线程构成一个环

- 主线程向T1发送数据0
- T1收到数据后，将数据加1，向T2发送数据1
- T2收到数据后，将数据加1，向T3发送数据2
- T3收到数据后，将数据加1，向T4发送数据3
- ...

4. 创建N个缓冲区
5. 每个线程有一个输入缓冲和一个输出缓冲
6. 最终的系统结构如下



7. **本程序不能使用任何全局变量，如果使用了全局变量，本题没有得分**

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#define N 10
#define LOOPCOUNT 25

void *add(void *arg){
    int *num = (int *)arg;
    num[1] = num[0] + 1;
    int *result = num;
}
int main()
{
    int buff[N][2];
    int i = 0;
    for(i = 0;i < N;i++)
    {
        buff[i][0]=0;
        buff[i][1]=0;
    }
    pthread_t tids[N];
```

```
    i = 0;
    int count = 0;
    while(i < N)
    {
        count++;
        if(count == LOOPCOUNT)
            break;

        printf("from T[%d]",i+1);
        pthread_create(&tids[i],NULL,add,(void *)&buff[i]);
        pthread_join(tids[i],NULL);
        int result = buff[i][1];

        i = (i+1) % N;
        buff[i][0] = result;
        printf("to T[%d] send %d\n",i+1,result);
    }
    return 0;
}
```

3. pipe.c -- 多进程 + 管道

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

int main()
{
    int std_in = dup(0);
    int std_out = dup(1);
    pid_t tid1 = -1,tid2 = -1;
    int fd1[2],fd2[2];
    pipe(fd1);
    pipe(fd2);

    tid1 = fork();
    if(tid1 > 0)
    {
        tid2 = fork();
    }
    if(tid1 == 0) //cat
    {
        dup2(fd1[1],1);
        close(fd1[0]);
        close(fd1[1]);
        execlp("cat","cat","/etc/passwd",NULL);
    }
    if(tid2 == 0) //grep
    {
```

```
        dup2(fd1[0],0);
        close(fd1[0]);
        close(fd1[1]);
        // char buff[1024];
        // int readsize = read(fd[0],buff,sizeof(buff));
        // write(fd2[1],buff,readsize);

        dup2(fd2[1],1);
        close(fd2[0]);
        close(fd2[1]);

        execlp("grep","grep","root",NULL);
    }
    if(tid1>0) //wc
    {
        dup2(fd2[0],0);
        close(fd2[0]);
        close(fd2[1]);

        dup2(std_out,1);
        execlp("wc","wc","-l",NULL);
    }
    return 0;
}
```

4. computePI.c -- 多线程 + 参数传递

- 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

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

#define NUMBER 1000000
#define WORKERNUM 100

struct param{
    int start;
    int end;
    double result;
};

void *worker(void *arg){
    int i;
```

```
    struct param *param =  (struct param *) arg;
    for (i = param->start; i <= param->end;i++){
        if(i%2 == 0)
            param->result -= 1/(2 * (double)i - 1);
        else
            param->result += 1/(2 * (double)i - 1);
    }

}



void master()
{
    int i;
    pthread_t worker_tids[WORKERNUM];
    struct param params[WORKERNUM];
    double  PI = 0.0;

    for(i = 0;i < WORKERNUM;i++)
    {
        struct param *param = &params[i];
        param->start = i * NUMBER + 1;
        param->end = (i+1) * NUMBER;
        param->result = 0;
        pthread_create(&worker_tids[i],NULL,worker,(void *)&params[i]);
        pthread_join(worker_tids[i],NULL);
        PI += param->result;
    }
    PI = PI * 4;
    printf("PI;%lf\n",PI);
}

int main()
{
    master();
    return 0;
}
```

5 pc1.c: 使用条件变量解决生产者、计算者、消费者问题 -- 多线程 + 信号量、条件变量

- 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("            comsume 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

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("            comsume 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;
}
```

6 pc2.c: 使用信号量解决生产者、计算者、消费者问题 -- 多进程 + 信号量、条件变量

- 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("            comsume 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;
}
```

3.5.3 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("            comsume 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;
}
```