实验三 进程同步控制实验（读者-写者问题）

一，实验目的：

本实验旨在动手设计一个进程同步控制实验，更深刻的理解进程之间的协作机制。

二，实验内容

2.1 实验内容

- 利用信号量机制，提供读者-写者问题的实现方案，并分别实现读者优先与写者优先。
- 读者-写者问题的读写操作限制：
  1. 写-写互斥：不能有两个写者同时进行写操作
  2. 读-写互斥：不能同时有一个线程在读，一个线程在写
  3. 读-读允许：允许多个读者同时执行读操作

  读者优先：在实现上述限制的同时，要求读者的操作优先级高于写者，要求没有读者保持等待除非已有一个写者已经被允许使用共享数据。

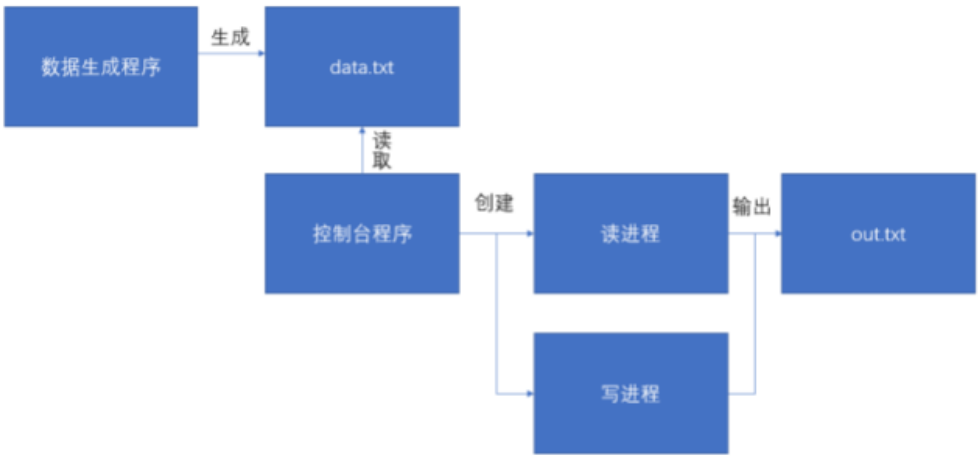
  写者优先：在实现上述限制的同时，要求写者的操作权限高于写者，要求一旦写者就绪，那么将不会有新的读者开始读操作。

2.2 实验要求

- 实验环境：在 OpenEuler/Linux 环境下，使用 C/C++开发环境。
- 程序要求：
  1. 创建一个包含 n 个线程的控制台程序，并用这 n 个线程表示 n 个读者或写者。
  2. 利用信号量机制，分别实现满足读者优先与写者优先的读者-写者问题。
  3. 输入要求：要求使用文件输入相应命令，并根据这些命令创建相应的读写进程
  4. 输出要求：要求运行结果在控制台输出并保存到相应文件中，输出内容包括线程创建提示，线程进入共享缓冲区提示，线程离开缓冲区提示。

三，实验原理图：

3.1 程序流程图



3.2 源代码分析

1. 数据结构设计
   - 包含数据的队列，写者向队尾写入数据，读者向从队首读取数据

     int que[1000];
   - 从主线程向新创建的线程传入的数据

```
struct thread_arg{
int number;//表示新创建的线程的序号
double create;//表示线程被创建的时间
int last;//表示此线程需要持续的时间
};
```

2. 各函数设计

- 持续时间函数设计

```
void mysleep(int second)
{
    clock_t start;
    start=clock();
    while(((clock()-start)/CLOCKS_PER_SEC)<second);
}
```

每次循环判断当前时间与起始时间的差是否已经大于所需持续的时间，如果大于则退出循环，否则继续循环占用 CPU

- 读者优先下的读进程与写进程

数据结构设计：

```
Int read_count=0;//读者数量
sem_t mutex;//互斥变量，用于控制对缓冲区的访问，初始哈为 1
sem_t RP_Write;//互斥变量，用于控制 read_count 的互斥访问，初始化为 1
```

读进程：

```
void RP_ReaderThread(void * arg)
{
    sem_wait(&mutex);//互斥访问 read_count
    read_count++;
    if(read_count==1)//如果是第一个读进程，申请获取访问权限
        sem_wait(&RP_Write);
    sem_post(&mutex);
//进入临界区
    int add;
    struct thread_arg *threadarg;
    threadarg=(struct thread_arg *)arg;//接收来自主进程的参数
    clock_t enter;
    clock_t leave;
    enter=clock();//记录进程进入临界区的时间
    mysleep(threadarg->last);//通过 mysleep 函数来表示线程持续时间
    printf("\nRead Thread %d is created:\n",threadarg->number);
    printf("Create Time=%f\n",threadarg->create);//输出线程创建时间
    printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
    add=rand()%100;
    if(pointer==0) //如果当前队列为空
        printf("The queue is empty\n");
    else//如果队列非空则从队首读出一个数据
    {
```

```
            add=que[header];
            printf("Pop %d from queue\n",add);
            header++;
        }
        leave=clock();//记录线程离开临界区的时间
        printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
    //离开临界区
        sem_wait(&mutex);
        read_count--;
        if(read_count==0)//如果是最后一个读进程，释放访问权限
            sem_post(&RP_Write);
        sem_post(&mutex);
}
```

写进程：

```
    void RP_WriterThread(void * arg)
    {

        sem_wait(&RP_Write);//等待访问权限
    //进入临界区
        int add;
        struct thread_arg *threadarg;
        threadarg=(struct thread_arg *)arg;
        clock_t enter;
        clock_t leave;
        enter=clock();//记录进入临界区的时间
        mysleep(threadarg->last);
        printf("\nWrite Thread %d is created:\n",threadarg->number);
        printf("Create Time=%f\n",threadarg->create);
        printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
        add=rand()%100;//随机生成要向缓冲区写入的数据
        que[pointer]=add;
        pointer++;
        printf("Add %d to queue\n",add);
        leave=clock();//记录离开临界区的时间
        printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
    //离开临界区
        sem_post(&RP_Write);//释放访问权限
}
```

- 写者优先下的读进程与写进程

  数据结构设计：

  ```
  int read_count=0;//读者数量
  int write_count=0;//写者数量
  sem_t mutex1;//互斥变量，控制 write_count 的互斥访问，初始化为 1
  ```

```
sem_t mutex2;//互斥变量，控制 read_count 的互斥访问，初始化为 1
sem_t cs_Read;//互斥变量，表示读者排队信号，初始化为 1
sem_t cs_Write;//互斥变量，控制对缓冲区的访问，初始化为 1
```

读进程：

```
    void RP_ReaderThread(void *arg)
    {
        sem_wait(&cs_Read);//申请排队权限
        sem_wait(&mutex2);//互斥访问 read_count
        read_count++;
        if(read_count==1)//如果是第一个读者，申请访问权限
            sem_wait(&cs_Write);
        sem_post(&mutex2);
        sem_post(&cs_Read);//释放排队权限
    //进入临界区
        int add;
        struct thread_arg *threadarg;
        threadarg=(struct thread_arg *)arg;
        clock_t enter;
        clock_t leave;
        enter=clock();
        mysleep(threadarg->last);
        printf("\nRead Thread %d is created:\n",threadarg->number);
        printf("Create Time=%f\n",threadarg->create);
        printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
        add=rand()%100;
        if(pointer==0)
            printf("The queue is empty\n");
        else
        {
            add=que[header];
            printf("Pop %d from queue\n",add);
            header++;
        }
        leave=clock();
        printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
    //离开临界区
        sem_wait(&mutex2);
        read_count--;
        if(read_count==0)//如果是最后一个读者，释放缓冲区访问权限
            sem_post(&cs_Write);
        sem_post(&mutex2);
}
```

写进程：

```
void RP_WriterThread(void *arg)
{

    sem_wait(&mutex1);//互斥访问 write_count
    write_count++;
    if(write_count==1)//如果是第一个写进程，申请获取读进程排队权限
        sem_wait(&cs_Read);
    sem_post(&mutex1);
    sem_wait(&cs_Write);//申请缓冲区访问权限
//进入临界区
    int add;
    struct thread_arg *threadarg;
    threadarg=(struct thread_arg *)arg;
    clock_t enter;
    clock_t leave;
    enter=clock();
    mysleep(threadarg->last);
    printf("\nWrite Thread %d is created:\n",threadarg->number);
    printf("Create Time=%f\n",threadarg->create);
    printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
    add=rand()%100;
    que[pointer]=add;
    pointer++;
    printf("Add %d to queue\n",add);
    leave=clock();
    printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
//离开临界区
    sem_post(&cs_Write);
    sem_wait(&mutex1);
    write_count--;
    if(write_count==0)
        sem_post(&cs_Read);//如果是最后一个写进程，释放读进程排队
权限，允许其排队访问
    sem_post(&mutex1);
}
```

四，实验环境

- 操作系统：Ubuntu 18.04
- 编译环境：gcc 编译器

五，实验步骤

data1.txt:

```
1 R 5
2 R 4
3 R 3
4 R 2
5 R 1
```

data2.txt:

```
1 W 5
2 R 4
3 W 2
4 R 3
```

读者优先：

```
andypja@ubuntu:~/app/lab-3$ gcc read_first.c -o read -lpthread
andypja@ubuntu:~/app/lab-3$ ./read
data1.txt 5

Read Thread 4 is created:
Create Time=0.004153
Enter critical section=0.005158
The queue is empty
Leave critical section=1.010205

Read Thread 3 is created:
Create Time=0.003739
Enter critical section=0.004604
The queue is empty
Leave critical section=2.009107

Read Thread 2 is created:
Create Time=0.003528
Enter critical section=0.042134
The queue is empty
Leave critical section=3.042691

Read Thread 1 is created:
Create Time=0.003114
Enter critical section=0.004011
The queue is empty
Leave critical section=4.004263

Read Thread 0 is created:
Create Time=0.002852
Enter critical section=0.003438
The queue is empty
Leave critical section=5.003557
```

写者优先：

```
andypja@ubuntu:~/app/lab-3$ gcc write_first.c -o write -lpthread
andypja@ubuntu:~/app/lab-3$ ./write
data2.txt 4
```

Write Thread 0 is created:
Create Time=0.003596
Enter critical section=0.004260
Add 83 to queue
Leave critical section=5.004467

Write Thread 2 is created:
Create Time=0.003943
Enter critical section=5.004605
Add 86 to queue
Leave critical section=7.004704

Read Thread 3 is created:
Create Time=0.004402
Enter critical section=7.005192
Pop 0 from queue
Leave critical section=10.005693

Read Thread 1 is created:
Create Time=0.003815
Enter critical section=7.005012
Pop 0 from queue
Leave critical section=11.005107

六，源代码：
读者优先：

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<time.h>
int read_count=0;
sem_t mutex,RP_Write;
int que[1000];
int pointer=0;
int header=0;

struct thread_arg{
int number;
double create;
int last;
};
```

```
void mysleep(int second)
{
	clock_t start;
	start=clock();
	while(((clock()-start)/CLOCKS_PER_SEC)<second);
}

void RP_ReaderThread(void * arg)
{
	sem_wait(&mutex);
	read_count++;
	if(read_count==1)
		sem_wait(&RP_Write);
	sem_post(&mutex);
	int add;
	struct thread_arg *threadarg;
	threadarg=(struct thread_arg *)arg;
	clock_t enter;
	clock_t leave;
	enter=clock();
	mysleep(threadarg->last);
	printf("\nRead Thread %d is created:\n",threadarg->number);
	printf("Create Time=%f\n",threadarg->create);
	printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
	add=rand()%100;
	if(pointer==0)
		printf("The queue is empty\n");
	else
	{
		add=que[header];
		printf("Pop %d from queue\n",add);
		header++;
	}
	leave=clock();
	printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
	sem_wait(&mutex);
	read_count--;
	if(read_count==0)
		sem_post(&RP_Write);
	sem_post(&mutex);
}

void RP_WriterThread(void * arg)
{
```

```
        sem_wait(&RP_Write);
        int add;
        struct thread_arg *threadarg;
        threadarg=(struct thread_arg *)arg;
        clock_t enter;
        clock_t leave;
        enter=clock();
        mysleep(threadarg->last);
        printf("\nWrite Thread %d is created:\n",threadarg->number);
        printf("Create Time=%f\n",threadarg->create);
        printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
        add=rand()%100;
        que[pointer]=add;
        pointer++;
        printf("Add %d to queue\n",add);
        leave=clock();
        printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
        sem_post(&RP_Write);
}

int main()
{
        pthread_t thread[100];
        char name[20],mode,useless;
        int row;
        int i;
        int last;
        clock_t create;
        struct thread_arg arg[100];
        clock_t time;
        scanf("%s %d",name,&row);
        FILE *fp;
        fp=fopen(name,"r");
        sem_init(&mutex,0,1);
        sem_init(&RP_Write,0,1);
        for(i=0;i<row;i++)
        {
                arg[i].number=i;
                fscanf(fp,"%c %d%c",&mode,&last,&useless);
                arg[i].last=last;
                if(mode=='R')
                {
                        create=clock();
```

```
                arg[i].create=(double)create/CLOCKS_PER_SEC;
                pthread_create(&thread[i],NULL,(void *)&RP_ReaderThread,(void *)&arg[i]);
            }
            else if(mode=='W')
            {
                create=clock();
                arg[i].create=(double)create/CLOCKS_PER_SEC;
                pthread_create(&thread[i],NULL,(void *)&RP_WriterThread,(void *)&arg[i]);
            }
        }
        for(i=0;i<row;i++)
        {
            pthread_join(thread[i],NULL);
        }
        return 0;
}
```

写者优先：

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<time.h>
int read_count=0,write_count=0;
sem_t mutex1,mutex2,cs_Read,cs_Write;
int que[1000];
int pointer=0;
int header=0;

struct thread_arg{
int number;
double create;
int last;
};

void mysleep(int second)
{
        clock_t start;
        start=clock();
        while(((clock()-start)/CLOCKS_PER_SEC)<second);
}

void RP_ReaderThread(void *arg)
```

```
{


    sem_wait(&cs_Read);
    sem_wait(&mutex2);
    read_count++;
    if(read_count==1)
        sem_wait(&cs_Write);
    sem_post(&mutex2);
    sem_post(&cs_Read);
    int add;
    struct thread_arg *threadarg;
    threadarg=(struct thread_arg *)arg;
    clock_t enter;
    clock_t leave;
    enter=clock();
    mysleep(threadarg->last);
    printf("\nRead Thread %d is created:\n",threadarg->number);
    printf("Create Time=%f\n",threadarg->create);
    printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
    add=rand()%100;
    if(pointer==0)
        printf("The queue is empty\n");
    else
    {

        add=que[pointer];
        printf("Pop %d from queue\n",add);
        header++;
    }
    leave=clock();
    printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
    sem_wait(&mutex2);
    read_count--;
    if(read_count==0)
        sem_post(&cs_Write);
    sem_post(&mutex2);
}

void RP_WriterThread(void *arg)
{

    sem_wait(&mutex1);
    write_count++;
```

```
        if(write_count==1)
            sem_wait(&cs_Read);
        sem_post(&mutex1);
        sem_wait(&cs_Write);
        int add;
        struct thread_arg *threadarg;
        threadarg=(struct thread_arg *)arg;
        clock_t enter;
        clock_t leave;
        enter=clock();
        mysleep(threadarg->last);
        printf("\nWrite Thread %d is created:\n",threadarg->number);
        printf("Create Time=%f\n",threadarg->create);
        printf("Enter critical section=%f\n",(double)enter/CLOCKS_PER_SEC);
        add=rand()%100;
        que[pointer]=add;
        pointer++;
        printf("Add %d to queue\n",add);
        leave=clock();
        printf("Leave critical section=%f\n",(double)leave/CLOCKS_PER_SEC);
        sem_post(&cs_Write);
        sem_wait(&mutex1);
        write_count--;
        if(write_count==0)
            sem_post(&cs_Read);
        sem_post(&mutex1);
}

int main()
{
        pthread_t thread[100];
        char name[20],mode,useless;
        int row;
        int i;
        int last;
        clock_t create;
        struct thread_arg arg[100];
        clock_t time;
        scanf("%s %d",name,&row);
        FILE *fp;
        fp=fopen(name,"r");
        sem_init(&mutex1,0,1);
        sem_init(&mutex2,0,1);
        sem_init(&cs_Read,0,1);
```

```
    sem_init(&cs_Write,0,1);
    for(i=0;i<row;i++)
    {
        arg[i].number=i;
        fscanf(fp,"%c %d%c",&mode,&last,&useless);
        arg[i].last=last;
        if(mode=='R')
        {
            create=clock();
            arg[i].create=(double)create/CLOCKS_PER_SEC;
            pthread_create(&thread[i],NULL,(void *)&RP_ReaderThread,(void *)&arg[i]);
        }
        else if(mode=='W')
        {
            create=clock();
            arg[i].create=(double)create/CLOCKS_PER_SEC;
            pthread_create(&thread[i],NULL,(void *)&RP_WriterThread,(void *)&arg[i]);
        }
    }
    for(i=0;i<row;i++)
    {
        pthread_join(thread[i],NULL);
    }
    return 0;
}
```

七，实验体会

这次实验差不多花了两个半天的时间去完成，主要的时间花在了查找信号量函数的资料和编写程序上。因为上课时指介绍了利用信号量解决进程同步问题的思想，并给出了伪代码，但是并没有给出可以在程序中使用的信号量函数。在查阅资料的过程中，发现有各种不同的信号量函数可以使用，经过比较最终选择了程序中使用的信号量函数版本。

之后的时间主要进行编程实现和调错以及设计输入的数据。经过这次实验我对进程同步的控制问题有个更清晰的理解，并且通过分析程序运行的结果，充分的意识到了进程同步控制的必要性，并且在编程的过程中熟悉了信号量及其函数的使用。