
北京郵電大學

Beijing University of Posts and Telecommunications

# 操作系统
# 实验报告

[进程同步实验]

学院：计算机学院
2020211376 马天成
2022 年 11 月 13 日

# 北京邮电大学《操作系统》课程实验报告

<table>
<tr><td>实验<br>名称</td><td colspan="2">process</td><td>学<br>院</td><td>计算机</td><td>指导教师</td><td>李文生</td></tr>
<tr><td>班 级</td><td>班内序<br>号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2020211305</td><td>2</td><td colspan="2">2020211376</td><td>马天成</td><td colspan="2"></td></tr>
<tr><td>实<br>验<br>内<br>容</td><td colspan="6">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。<br><br>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。<br>读者-写者问题的读写操作限制：<br>写-写互斥：不能有两个写者同时进行写操作。<br>读-写互斥：不能同时有一个线程在读，一个进程在写。<br>读-读允许：允许多个读者同时执行读操作。<br>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。<br>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。</td></tr>
<tr><td>学生实验报告</td><td colspan="6">(详见“实验报告和源程序”册)</td></tr>
<tr><td>课<br>程<br>设<br>计<br>成<br>绩<br>评<br>定</td><td colspan="6">评语:<br><br>成绩:<br><br>指导教师签名:<br><br>年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

# 目录

1. 实验目的和要求.....4
2. 实验内容.....4
    2.1 准备工作.....4
    2.3 process.....5
3. 实验总结.....12

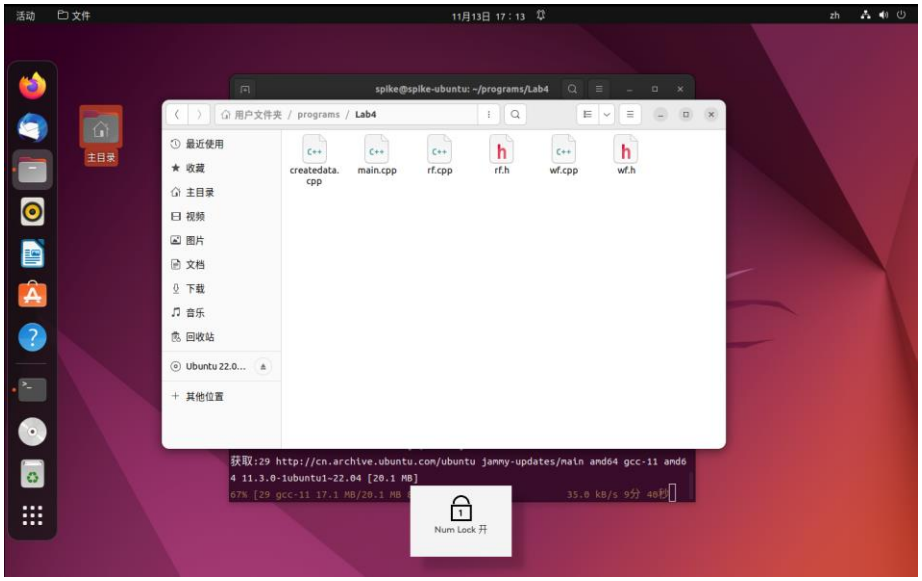
# 1. 实验目的和要求

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。

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：
写-写互斥：不能有两个写者同时进行写操作。
读-写互斥：不能同时有一个线程在读，一个进程在写。
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. 实验内容

## 2.1 准备工作




操作系统：Ubuntu22.04.1
编译环境：g++ 11

首先准备好 linux 平台。上述已经准备好了。
此外，也要准备好 g++环境。

代码编写的思路也很简单。照着 PDF 上的思路复现一遍基本上就可以了。

## 2.2 createdata.cpp

生成 data.txt 文件

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
int main()
{
    int dataNum = 20;//数据条数
    int timeMax=5;//读写最大时长，以 100 毫秒为单位
    char RW[2]={'R','W'};//命令类型，分为读、写命令
    ofstream  data("data.txt");//以覆盖方式打开文件
    if(data)//打开文件成功
    {
        srand((unsigned)time(NULL));
        for(int i=0;i<dataNum;i++){
            int index=rand()%2;//决定读写方式
            int spendTime=(rand()%timeMax+1);//读写花费时间
            data<<RW[index]<<""<<spendTime;//输出到文件保存
            if(i<dataNum-1)
                data<<"\r\n";
        }
        data.close();
    }
}
```

执行结果：

```
spike@spike-ubuntu:~/programs/Lab4$ g++ createdata.cpp -o createdata
spike@spike-ubuntu:~/programs/Lab4$ ./createdata
spike@spike-ubuntu:~/programs/Lab4$ ls
createdata  createdata.cpp  data.txt  main.cpp  rf.cpp  rf.h  wf.cpp  wf.h
```

## 2.3 main.cpp

鉴于其他四个文件都是给好的，我们也就是在老师代码的基础上加了输出锁，所以只展示 main 文件。

```
#include <iostream>
#include <iostream>
#include <cstdio>
#include <vector>
#include <string>
#include <fstream>
#include <pthread.h>
#include <cstdlib>
#include "rf.h"
//#include "wf.h"          // 两个功能分别注释一个

using namespace std;

typedef struct order
{
    char rw;          //读写进程标志
    int spendtime;        //读写时间
}order;

vector<order>orders;      //命令集合
string fileName = "data.txt";   //文件名称
int orderNum = 20;            //命令数量

string output_name = "output.txt";
ofstream outFile;

string output_str = "";

int main()
{
    // solve input
    ifstream file(fileName);
    order t;
    for (int i = 0; i < orderNum; i++) {
        file >> t.rw >> t.spendtime;
        orders.push_back(t);
    }

    // solve output
```

```
    outFile.open(output_name);

    pthread_t * p = (pthread_t*)malloc(orderNum * sizeof(pthread_t));
    int* p_id = (int*)malloc(orderNum * sizeof(int));//读写进程创建

    initing();  //互斥量初始化
    for(int i=0;i<orderNum;i++)
    {
        if(orders[i].rw=='R')//创建读进程
        {
            output_str += to_string(i+1) + " process for Reader created.\n";
            p_id[i]=i+1;
            pthread_create(&p[i],NULL,reader,&p_id[i]);

        }
        else//创建写进程
        {
            output_str += to_string(i+1) + " process for Writer created.\n";
            p_id[i]=i+1;
            pthread_create(&p[i], NULL, writer, &p_id[i]);
        }
    }

    //等待线程全部结束后主线程结束
    for(int i=0;i<orderNum;i++)
        pthread_join(p[i],NULL);

    // output
    cout << output_str;
    outFile << output_str;
    outFile.close();

    return 0;
}
```

主要就是执行代码部分。他根据输入文件来调用线程创建。并使用输出锁来进行输出限制（因为在实验过程中会出现很多乱序的现象。这个现象是由输出不具备线程稳定性所导致的）。

所以我们需要读或者写的时间。这里规定为 10ms。这样就可以很明确的看到读写进程的优先级问题了。

## 2.4 读者优先

这里显然用到的是老师给的 rf：rd.cpp 和 rf.h：

1. cpp 中含有详细的线程逻辑，以及各种互斥设置（只展示 cpp 文件）

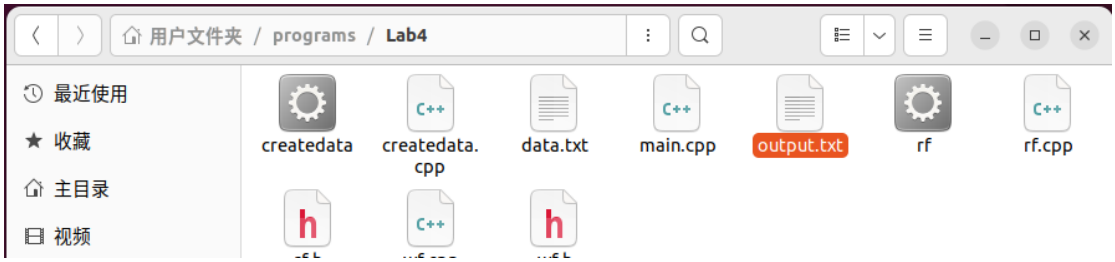
2. h：包含 extern 的输出 string 变量，使得可以进行输出往字符串里加。

```
#include "rf.h"
#include <iostream>
#include <fstream>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>



using namespace std;

int shared_data; //共享数据
int read_count = 0; //读者数量
sem_t rp_wrt; //互斥变量，用于控制对缓冲区的访问，初始化为 1
sem_t mutex; //互斥变量，用于控制 read_count 的互斥访问，初始化为 1



void initing()
{
    sem_init(&rp_wrt, 0, 1);
    sem_init(&mutex, 0, 1);
}



void* writer(void* param)
{
    int* id = (int*)param;
    sem_wait(&rp_wrt); //等待访问权限
    output_str += "-----Writer visit critical section.\n";
    output_str += "This is Writer, with id="+to_string(*id)+".\n";
    shared_data = *id;
    sleep(10);
    output_str += "-----Writer left critical section.\n";
    sem_post(&rp_wrt); //释放访问权限
}

void* reader(void* param)
{
    int* id = (int*)param;
    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_wrt);
        output_str += "-----Reader(the fitst) visit critical section.\n";
    }

    sem_post(&mutex);
    output_str += "This is Reader, with id="+to_string(*id)+",
shared_data="+to_string(shared_data)+".\n";
    sleep(10);
    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
    {
        output_str += "-----Reader(the last) left critical section.\n";
        sem_post(&rp_wrt);
    }
    sem_post(&mutex);
}
```

运行结果：

```
spike@spike-ubuntu:~/programs/Lab4$ g++ main.cpp rf.cpp rf.h -o rf -lpthread
rf.cpp: In function 'void* writer(void*)':
rf.cpp:34:1: warning: no return statement in function returning non-void [-Wreturn-type]
   34 | }
      | ^
rf.cpp: In function 'void* reader(void*)':
rf.cpp:59:1: warning: no return statement in function returning non-void [-Wreturn-type]
   59 | }
      | ^
spike@spike-ubuntu:~/programs/Lab4$ ./rf
1 process for Writer created.
2 process for Reader created.
3 process for Reader created.
4 process for Writer created.
-----Writer visit critical section.
This is Writer, with id=1.
17 process for Writer created.
18 process for Writer created.
19 process for Writer created.
20 process for Writer created.
-----Writer left critical section.
-----Reader(the fitst) visit critical section.
This is Reader, with id=3, shared_data=1.
This is Reader, with id=2, shared_data=1.
This is Reader, with id=8, shared_data=1.
This is Reader, with id=15, shared_data=1.
This is Reader, with id=9, shared_data=1.
This is Reader, with id=7, shared_data=1.
This is Reader, with id=16, shared_data=1.
This is Reader, with id=11, shared_data=1.
This is Reader, with id=13, shared_data=1.
-----Reader(the last) left critical section.
```



这时候观察 output.txt 可知，我们确实是对读者进行了优先处理。

在这个过程中，我们会发现一把情况下我们的输出时乱序的。因为 cout 和写入文件都是需要系统调用的，在这个过程中输出可能并不能显出来和线程执行的顺序一样。所以我们这里采取的策略是：用**字符串共享变量进行输出的收集**，在最后 main 函数结尾之前输出到 terminal 和文件中。

## 2.5 写者优先

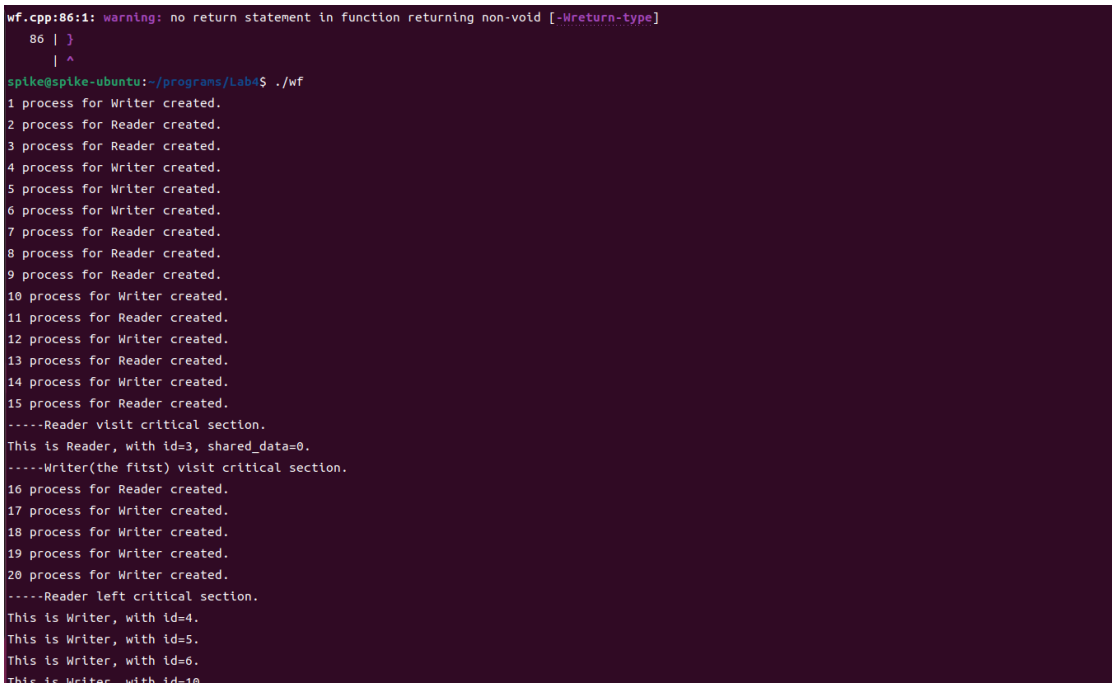
和读者优先类似，其实还是一个处理的问题。

```
#include "wf.h"
#include <iostream>
#include <fstream>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>

using namespace std;

int shared_data; //共享数据
int read_count; //读者数量
int write_count; //写者数量
sem_t rp_wrt; //互斥变量,控制对缓冲区的访问，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 mutex_w; //互斥变量，控制 write_count 的互斥访问，初始化为 1
sem_t mutex_r; //互斥变量，控制 read_count 的互斥访问，初始化为 1
sem_t wp_wrt; //互斥变量，控制对缓冲区的访问，初始化为 1

string output_file = "output2.txt";

void initing()
{
    sem_init(&rp_wrt, 0, 1);
    sem_init(&wp_wrt, 0, 1);
    sem_init(&cs_read, 0, 1);
    sem_init(&mutex_w, 0, 1);
    sem_init(&mutex_r, 0, 1);
}

void* writer(void* param)
{
    int* id = (int*)param;

    sem_wait(&mutex_w); //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);
        output_str += "-----Writer(the fitst) visit critical section.\n";
    }
    sem_post(&mutex_w);
    sem_wait(&wp_wrt); //申请缓冲区访问权限

    output_str += "This is Writer, with id="+to_string(*id)+".\n";
    shared_data = *id;
    sleep(10);

    sem_post(&wp_wrt);
    sem_wait(&mutex_w);
    write_count--;
    if (write_count == 0)
    {
        output_str += "-----Writer(the last) left critical section.\n";
        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(&mutex_w);
}

void* reader(void* param)
{
    int* id = (int*)param;

    sem_wait(&cs_read); //申请排队权限
    sem_wait(&mutex_r); //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(&wp_wrt);
        output_str += "-----Reader visit critical section.\n";
    }
    sem_post(&mutex_r);
    sem_post(&cs_read); //释放排队权限

    output_str += "This is Reader, with id="+to_string(*id)+",
shared_data="+to_string(shared_data)+".\n";
    sleep(10);

    sem_wait(&mutex_r);
    read_count--;
```

```
    if (read_count == 0) //如果是最后一个读者，释放缓冲区访问权限
    {
        output_str += "-----Reader left critical section.\n";
        sem_post(&wp_wrt);
    }
    sem_post(&mutex_r);
}
```

运行结果：




# 3. 实验总结

- 互斥变量未初始化问题

最初运行时：程序会进入阻塞状态，因为所给程序中并没有为互斥变量初始化，导致所有线程都阻塞在开始访问互斥变量的阶段。

- 输出乱序问题

虽然我们的线程运行没有啥问题，但是我们的线程输出有问题。cout 和文件输出流都不是线程安全的，因为系统调用中会使得输出乱序。所以我们直接用字符串进行输出的收集，在最后进行输出。

通过本次实验，我深入理解了课本中所学的同步与互斥的知识。我们在试验中详细实践了互斥变量的使用，懂得了需要初始化以及配套使用互斥变量。在这个过程中，对我的编程能力也有了很大的提高。这是一次非常有意思的实验。