进程同步实验-readers-writers problem

目录

读者-写者问题

1. 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。

2. 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：
  - 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。

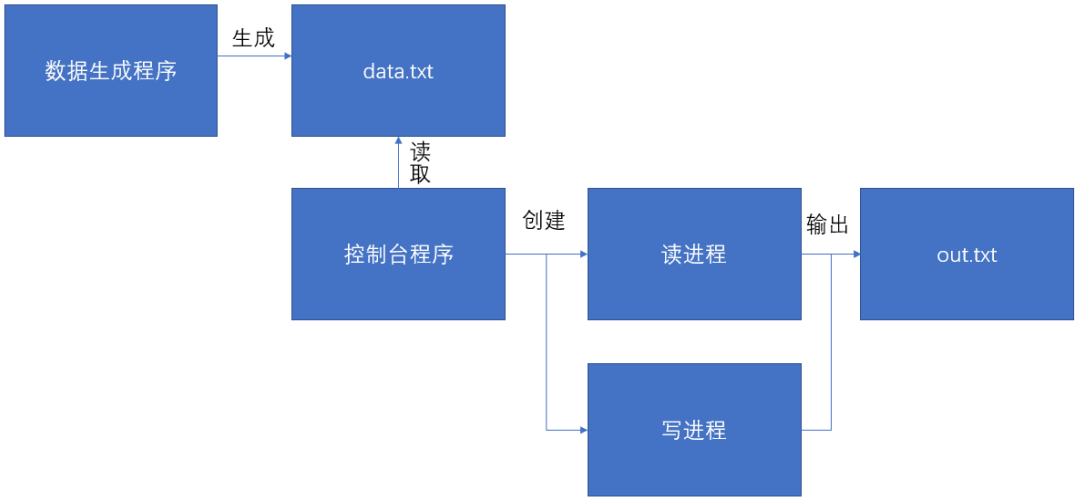
2.2 实验要求

- 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在相应文件中。输出内容包括线程创建提示、线程进入临界区提示、线程操作执行提示、线程离开临界区提示。

3. 实验原理

3.1 程序流程图



3.2 数据生成逻辑

1. 数据结构设计

```
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
```

2. 算法逻辑

```
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();
}
```

3.3 程序输入逻辑

1. 数据结构设计

- 数据类型 `order`，用于表示从 `data.txt` 读入的一条命令。

```
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;          //文件名称
int   orderNum;           //命令数量
```

2. 算法实现

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

3.4 读写进程创建与关闭

1. 数据结构设计

```
pthread_t *p;     //指向 pthread_t 类型的指针，根据命令数动态创建进程数组。
int *p_id;        //指向 int 类型的指针，用于存储线程编号，根据命令数量动态创建。
```

2. 算法实现

```
pthread_t *p = (pthread_t *)malloc(orderNum * sizeof(pthread_t));
int *p_id = (int *)malloc(orderNum * sizeof(int));

//读写进程创建
for(int i = 0; i < orderNum; i++)
{
    if(orders[i].rw == 'R') //创建读进程
    {
        p_id[i] = i + 1;
        pthread_create(&p[i], NULL, reader, &p_id[i]);
    }
    else                //创建写进程
    {
        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);
}
```

3.4 读者优先逻辑

1. 数据结构设计

```
int   shared_data;      //共享数据
int   read_count;       //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
```

2．算法实现

写线程逻辑

```
void *writer (void *param)    {
 sem_wait(&rp_wrt); //等待访问权限
 /* 打印输出‘W’、自己的 id，并将 id 写入 shared_data ; */
 sem_post(&rp_wrt); //释放访问权限
 }
```

读线程逻辑

```
void *reader(void *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);
 sem_post(&mutex);

 /* 打印输出‘R’、自己的 id，读取 shared_data，并打印输出 */

 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_wrt);
 sem_post(&mutex);
}
```

3.5 写者优先逻辑

1. 数据结构设计

```
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
```

2. 算法实现

写进程逻辑

```
void *writer(void *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);
 sem_post(&mutex_w);

 sem_wait(&wp_wrt);     //申请缓冲区访问权限
/* 打印输出‘W’、自己的 id，并将 id 写入 shared_data */
```

```
 sem_post(&wp_wrt);

 sem_wait(&mutex_w);
 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(&mutex_w);
}
```

读进程逻辑

```
void *reader(void *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);
 sem_post(&mutex_r);
 sem_post(&cs_read);        //释放排队权限

 /* 打印输出‘R’、自己的 id，读取 shared_data，并打印输出 */

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
sem_post(&wp_wrt);
sem_post(&mutex_r);

}
```

4. 实验环境

- 操作系统：Ubuntu18.04
- 编译环境：g++编译器

5. 实验步骤

5.1 运行 createdata.cpp 文件，生成 data.txt 文件

- 设置命令数 dataNum 与读写进程执行的最长时间

例：
```
int dataNum = 20;  //生成 20 条命令
int timeMax = 5;   //读写最大时长为 500ms
```

- 编译命令：g++ createdata.cpp -o createdata
- 运行命令：./createdata
- 运行结果：

5.2 执行主进程（读者优先）

- 编译命令：g++ main.cpp rf.cpp rf.h -o rf -lpthread
- 运行命令：./rf
- 运行结果：
- 查看 out.txt 文件：cat output.txt

5.3 执行主进程（写者优先）

- 编译命令：g++ main.cpp wf.cpp wf.h -o wf -lpthread
- 运行命令：./wf
- 运行结果：
- 查看 output.txt 文件：cat output.txt