目录

一．实验目的

二．实验内容

三．实验原理

3.1 信号量机制

1962 年，荷兰学者 Dijksrta 在参与 X8 计算机的开发中设计并实现了具有多道程序运行能力的操作系统——THE Multiprogramming System。为了解决这个操作系统中进程（线程）的同步与互斥问题，他巧妙地利用火车运行控制系统中的“信号灯”（semaphore，或叫“信号量”）概念加以解决。信号量的值大于 0 时，

基本原理：两个或多个进程通过简单的信号进行合作，一个进程被迫在某一位置停止，直到它接收到一个特定的信号。为了发信号，需要使用一个称作信号量的特殊变量。

由于信号量只能进行两种操作等待和发送信号，即 P(sv)和 V(sv), 在荷兰文中，通过叫 passeren，释放叫 vrijgeven，PV 操作因此得名。

他们的行为是这样的：

P(sv)：如果 sv 的值大于零，就给它减 1；如果它的值为零，就挂起该进程的执行。

V(sv)：如果有其他进程因等待 sv 而被挂起，就让它恢复运行，如果没有进程因等待 sv 而挂起，就给它加 1。

3.2 读者写者问题

在现代操作系统中，一般都提供与信号量相关的 API，只是实现形式可能会有不同。但信号量属于操作系统内核对象，由内核负责管理。提供给用户的操作原语仅有初始化，申请，释放几种，用户通过对信号量的巧妙使用，可以解决各种复杂的 IPC 问题。

读者写者问题就是这样一个经典的 IPC 问题，问题的描述如下：

(1) 允许多个读者可以同时对文件执行读操作；

(2) 只允许一个写者往文件中写信息；

(3) 任一写者在完成写操作之前不允许其他读者或写者工作；

(4) 写者执行写操作前，应让已有的读者和写者全部退出。

也就是说，读进程不排斥其他读进程，而写进程需要排斥其他所有进程，包括读进程和写进程。

四．实验过程

4.1 实验环境

CPU	Intel core i5 8250U
内存	8GB
Windows	Windows 10
QT	QT 4.3.1

4.2 程序框架

我们程序的主要目标是，将之前隐藏于操作系统内核中的信号量调度过程直观地展示给用户，所以我们的程序需要在前端显示程序运行的状态，并在后台真正的执行它。程序采用界面与业务逻辑分离的前端后端模式，前端后端单独开发，前端调用后端提供的接口执行功能，接受后端返回的信息，根据信息来进行操作。

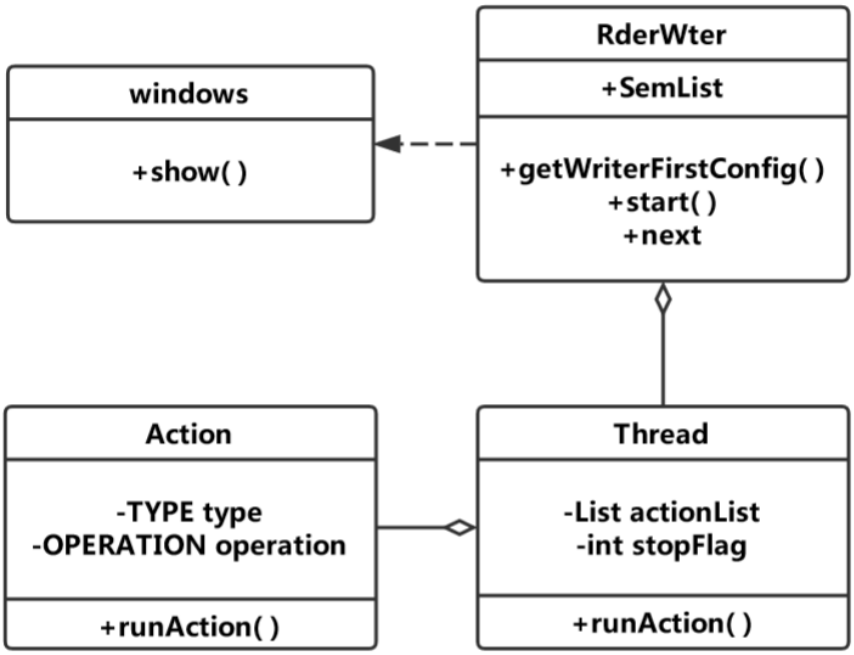

图 1 主要类的类图

程序前端包括 main.cpp 和 mainWindows.cpp 两个文件，其中业务逻辑主要在 mainWindows.cpp 中实现。后端部分包括 ReaderWriter.cpp ， Action.cpp ActionList.cpp ， Thread.cpp ， Semaphore.cpp 和相应的头文件。其中，ReaderWriter.cpp 主要作为供前端调用的接口，每一个读者写者对应一个 Thread 对象，该对象记录了读者写者线程需要执行的操作以及执行到了哪一步，每一步操作都是通过一个 Action 来记录的，Action 对象中最重要的成员是 type 和 operation 用来标识出具体的操作，每一个 Action 对象都要一个 runAction 方法，用来执行具体的操作。

类之间的调用和聚合关系如图 1 所示。

当程序启动时，main()函数创建一个主线程，当用户点击 start 按钮时，调用 start()函数读取事先准备好的命令文件，并为每一个读者写者创建一个 Thread 对象，这个对象接受刚刚读取的命令文件并使用这些命令来构造一个 ActionList 对象，这是许多 Action 对象的集合。同时，Thread 中还有一个成员来记录运行到了哪一步。这个功能有点像程序调试中的单步执行，当用户点击 next 按钮时，Thread 对象根据当前执行的位置，从列表中读出当前执行到的 Action，用它所包含的命令作为内容，创建一个线程并执行。执行完成后随即结束该线程，并将执行结果返回给用户。

时序图如图 2 所示。

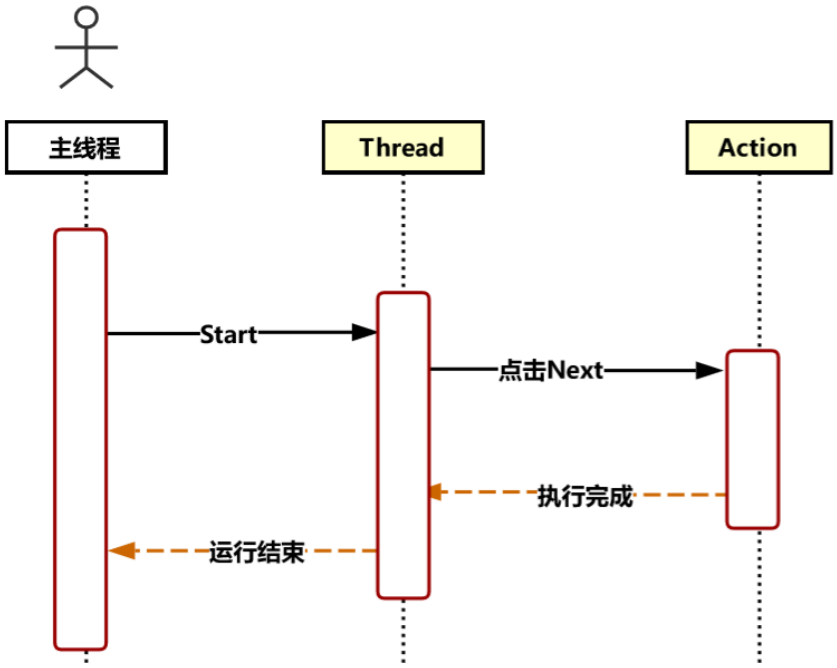

图 2　主要类的时序图

该程序中，每个读者写者都被抽象成一系列指令流组成的模型，这些指令流会在界面中显示给用户，同时，指令也是程序运行的依据。有 3 种不同类型的读者写者问题，我们就用 3 个 txt 文件保存初始的指令流。当用户点击 start 按钮时，指令流被以字符串形式读入，经过一系列变换之后转化为一个 ActionList 保存。同时，用户可以在层边实时观察每个信号量的状态，信号量属于内核对象，并不能直接读取，为了做到这一点，我们将原生的信号量封装成一个对象，并用其中的成员记录信号量的值，读取信号量时，实际上读取的是这个成员。

关于前端，后端，各个类的具体实现，在接下来的章节中有阐述。

4.3 读者写者问题的实现

4.4 后端具体实现

4.4.1 Thread 对象

Thread 对象保存一个读者写者线程的命令流和断点信息，在界面中有 6 个打开的读者写者框，所以，开始时一次创建 6 个 Thread 对象。Thread 对象中的成员包括 actionList 和 stopFlag，actionList 是一个 Action 的集合，一个 Action 对应一步操作，而 stopFlag 对应着这个集合中的下标。

Thread 对象的定义如代码 1 所示。

```
1       class Thread
2       {
3       private:
4           string name;            //可识别出类型
5           ActionList actionList; //指令流
6           int stopFlag;           //断点位置
7                   ……
8       public:
9           Thread(string sName, int iNumber,
10              int iPeriod, bool bRandom);
11          void appendAction(Action aAction);
12          void insertAction(Action aAction);
13          void removeAction(int iIndex);
14          void runAction(int iIndex);
15          // 调用 actionList.runAction
16      };
```

代码 1　Thread 对象

name

Thread 对象的唯一标识。可以用来识别 Thread 对象是读者还是写者。

actionList

该读者写者线程需要指令流。同一种类型的 Thread 对象具有相同的指令流，指令流是对象创建时，由字符串读入然后转变而来的。整个程序中 ActionList 类只在这里使用。

stopFlag

记录断点位置。其具体含义是，ActionList 中的数组下标，每次用户点击 next 按钮时，都会根据 stopFlag 读出相应的 Action，并且执行 Action。

appendAction()

用来向 actionList 末尾中添加 Action 对象，供构造 actionList 对象时使用。

insertAction()

用来向 actionList 任意位置添加 Action 对象。

removeAction()

从 actionList 中删除一个 Action 对象。

runAction()

该函数没有具体实现。调用成员中的实现。

4.4.2 Action 对象

Action 对象是程序中执行单步操作的载体，它最为核心方法是 runAction()方法，静态的指令流通过这个方法得到具体的执行。读者和写者执行的指令流最初记录在一个文本文件中，经过一系列转化，调用 ReaderWriter::addString()方法，可以为 Action 中的成员依次赋值。多个 Action 最终被聚合成一个 ActionList，再由 ActionList 的所有者——Thread 对象调用。

Action 对象的定义如代码 2 所示。

```
1       class Action
2       {
3       private:
4           ACTION_TYPE type;   //操作类型
5           ACTION_OPERATION operation;
6           int semaphore;      //全局数组的下标
7       int counter;
8
9       public:
10          Action();
11          void setType(ACTION_TYPE aType);
12          ACTION_TYPE getType();
13                  ……
14          void runAction();   //执行操作
15      };
```

代码 2　Action 对象

type

操作的类型。可以为 SEMPHARE，COUNTER，IF，或者 DATA。

operation

具体的操作。根据 type 字段的不同而不同。如果是 SEMPHARE 操作，则可以取值 P，V。如果是 COUNTER 操作，则可以取值 INCREASE，DECREASE。如果是 IF 操作，则可以取值 EQUAL，GREATER，SMALLER。

semaphore

该操作需要改变的信号量。ReaderWriter::start()定义了一组信号量，由于信号量不为任何进程私有，所以给出信号量的名称就可以操作信号量，这里使用信号量数组的下标访问。

counter

供 IF 操作和 DATA 操作使用的计数变量。同样也是一个全局数组的下标。

runAction()

执行 Action 的具体指令。这是 Action 对象的核心方法，如果 type 是信号量，则改变信号量的值，如果是其他则读取或改变 counter 的值。

4.4.3 Semaphore 对象

Semaphore 对象实现了队操作系统原生信号量的封装，以及其 PV 操作的封装。在前端界面中需要实时地向用户显示当前各个信号量的值，然而，信号量属于内核对象，由内核负责维护调度，不能直接获取信号量的值。所以 Semaphore 对象的另外一个重要功能就是记录该信号量的值，并提供 getValue()方法读出。

Semaphore 对象的定义如代码 3 所示。

```
1       class Semaphore
2       {
3       private:
4           string name;
5           int value;
6           HANDLE hSem;
7       public:
8           Semaphore(string sName,
9               int iLow, int iHigh);
10              ……
11          int getValue();
12          void p();
13          void v();
14      };
```

代码 3　Semaphore 对象

name

信号量的名称。用于唯一标识信号量。

value

信号量的值。它与内核中信号量的值同步，是信号量真实值的一个副本，可供用户直接读出。

hSem

信号量的句柄。

getValue()

返回当前信号量的值。实际上读取的是 int value 字段的值。

p()

封装 P 操作，每次调用原生 P 操作前都要将 value 减 1。

v()

封装 V 操作，每次调用原生 V 操作前都要将 value 加 1。

4.5 后端流程

4.5.1 初始化

后端的操作主要可以分为两个阶段，初始化数据以及单步操作。当程序启动时，首先创建界面主线程，监听用户选择的模式，当用户选择了一种模式之后，如选择读者优先模式，则调用 ReaderWriter::getReaderFirstConfig()方法读取对应的指令流文件，并将其写入到一个字符串组成的数组——vReader 中，每个指令对应数组中的一项。

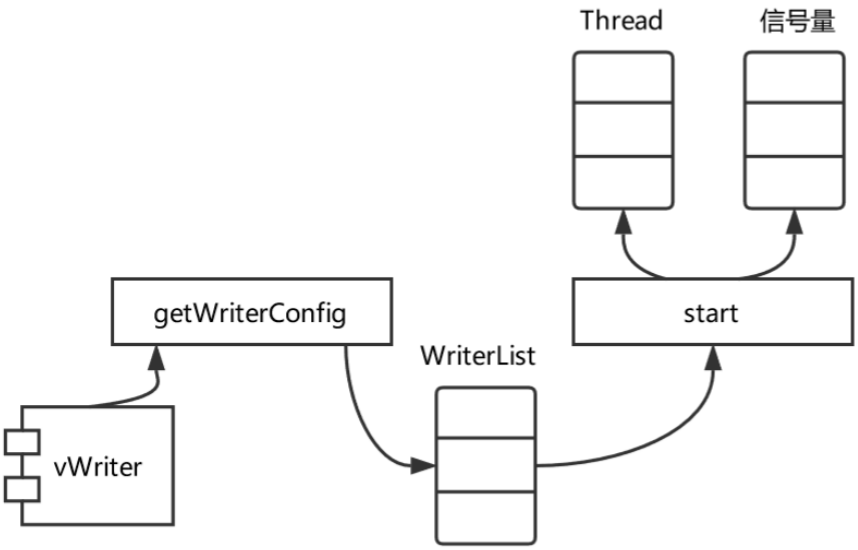

图 3　初始化流程

当用户点击 start 按钮时，调用 ReaderWriter::start()函数。该函数首先根据所选择的模式创建信号量数组并设置值，供全局使用，接着创建一个 count 全局数组并设置值供 Action 对象使用。最后，ReaderWriter::start()函数函数还负责为每一个读者写者创建一个对应的 Thread 对象。每个 Thread 对象都有一个 ActionList 成员，它有许多 Action，ReaderWriter::addString()函数根据 vReader 中的内容为每个 Action 赋初值。

初始化流程如图 3 所示。

4.5.2 单步操作

当用户在界面上点击 next 按键时，业务函数调用 ReaderWriter::next()方法，由于控制一个线程单步执行相当困难，所以对于每一个单步操作，我们都根据当前指令创建一个新线程，执行完这步操作后就结束这个线程。线程的入口函数为 run()全局函数，run()函数再根据操作的类型执行具体的操作。

若是 IF 或者 DATA 类型操作，则在 run()函数内部完成操作，如果是信号量操作，则需要调用该 Action 对象的 runAction()方法来执行操作。操作完成后即可以结束该线程，避免浪费资源。

单步操作的流程如图 4 所示。

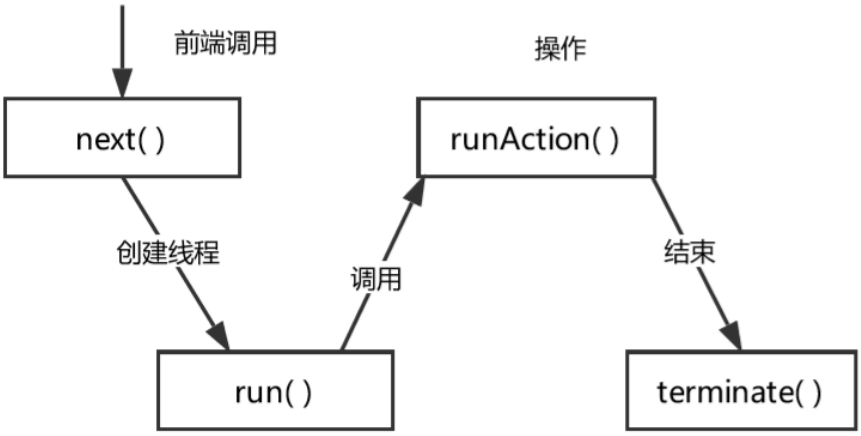

图 4　单步操作流程

4.6　接口

五．实验结果

六．心得体会