



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

Исследование многопоточности в задаче «Читатели-писатели»

Студентка ИУ7-63Б(В)
(Группа)

(Подпись, дата) **О.О. Ишкова-Запольская**
(И.О.Фамилия)

Руководитель курсовой работы

(Подпись, дата) **Н.Ю. Рязанова**
(И.О.Фамилия)

2022 г.

**Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)**

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
(Индекс)
И.В.Рудаков
(И.О.Фамилия)
« ____ » _____ 20 ____ г.

**З А Д А Н И Е
на выполнение курсового проекта**

по дисциплине Операционные системы

Студентка группы Ишкова-Запольская О.О., ИУ7-63Б(В)
(Фамилия, имя, отчество)

Тема курсового проекта «Исследование многопоточности в задаче
«Читатели-писатели» »

Направленность КП (учебный, исследовательский, практический, производный. др.)
Учебная

Источник тематики (кафедра, предприятие, НИР) Кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Задание: Реализовать решение задачи «Читатели-писатели» с помощью монитора Хоара.

Оформление курсового проекта:

Расчетно-пояснительная записка на 20-25 листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.).

На защиту проекта должна быть представлена презентация, состоящая из 10-15 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчётные соотношения, структура комплекса программ, интерфейс, характеристики разработанного ПО.

Дата выдачи задания « ____ » _____ 20__ г.

Руководитель курсового проекта

Н.Ю. Рязанова
(Подпись, дата) (И.О.Фамилия)

Студентка

О.О. Ишкова-Запольская
(Подпись, дата) (И.О.Фамилия)

Оглавление

Введение.....	4
1. Аналитический раздел.....	5-13
1.1 Постановка задача.....	5
1.2 Анализ задачи «Читатели-писатели».....	5
1.3 Мониторы	6
1.4 Выбор средств взаимoisключения.....	8
1.4.1 Семафоры.....	8
1.4.2 Мьютекс (Mutex).....	9
1.4.3 Событие (Event).....,,,	11
2. Конструкторский раздел.....	14-16
2.1 Структура ПО.....	14
2.2 Схема алгоритма «читателя».....	15
2.3 Схема алгоритма «писатель».....	16
3. Технологический раздел.....	17-20
3.1 Выбор языка и среды программирования.....	17
3.2 Код программы.....	18
4. Исследовательский раздел.....	21-22
4.1 Анализ разработанного ПО.....	21
Заключение.....	23
Список использованных источников.....	24

Введение

Целью данной курсовой работы является рассмотрение задачи «Читатели-писатели» и её решение предложенное Хоаром.

Данная задача представляет собой одну из широко известных и часто встречающихся классических задач взаимодействия параллельных процессов.

На основе задачи «Читатели-писатели» реализованы системы бронирования и продажа электронных билетов на самолёты и поезда, в кино и в театры.

1. Аналитический раздел

1.1 Постановка задачи

В соответствии с задачей на курсовой проект необходимо исследовать проблемы многопоточности на примере построения на основе решения Хоара задачи «Читатели-писатели».

Для решения поставленной задачи необходимо:

В соответствии с заданием на курсовую работу необходимо:

1. Провести анализ особенностей на основе монитора Хоара;
2. Исследовать возникающие проблемы многопоточности при увеличении числа потоков-писателей и потоков-читателей;
3. Разработать программное обеспечение;
4. Протестировать работу разработанного ПО.

1.2 Анализ задачи «Читатели-писатели»

Для задачи «Читатели-писатели» характерно наличие двух типов процессов: «читатели», которые только читают данные, и «писатели», которые модифицируют данные.

Потоки-читатели не меняют содержимого базы данных, поэтому несколько таких потоков могут обращаться к такой базе данных одновременно. Поток-писатель может изменять данные в базе, и поэтому он должен обладать к ней исключительным, монопольным доступом. Пока с базой данных работает писатель – никакие другие потоки (и писатели и читатели) работать с базой данных не должны. Такой режим монопольного доступа организуется для отдельных записей баз данных, а не для базы данных целиком. Таким образом, должны выполняться следующие правила:

- 1) когда один поток пишет в область общих данных – другие потоки не могут ни считывать, ни записывать в эту область;

2) когда один поток читает из области общих данных – другие потоки не могут туда ничего записывать, но могут читать данные.

1.3 Мониторы

Монитор (monitor) – это программное средство, разработанное Хоаром и дающее возможность управляемого совместного использования ресурсов среди асинхронных процессов, включая возможность управляемого обмена параметрами между процессами. Идея монитора заключается в создании механизма, который унифицирует взаимодействие параллельных процессов по разделяемым данным и процедурам, которые обрабатывают эти данные.

Монитор – это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации динамического распределения конкретного ресурса. Синтаксически монитор начинается ключевым словом `monitor`. Монитор защищает свои переменные. Доступ к переменным монитора можно получить, только используя процедуры монитора. Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, вызывая процедуру монитора. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Если вызов был успешным, то процесс считается, находящимся в мониторе. Вход в монитор находится под жестким контролем — здесь осуществляется взаимоисключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессы, которые хотят войти в монитор, когда он уже занят, ставятся в очередь к монитору, причем режимом ожидания автоматически управляет сам монитор. Монитор сам является ресурсом.

При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждет. Проверка условия выполняется самим монитором, который и разблокирует ожидающий процесс. Поскольку

механизм монитора гарантирует взаимоисключение процессов, отсутствуют серьезные проблемы, связанные с организацией параллельных взаимодействующих процессов. При первом обращении монитор присваивает своим переменным начальные значения. При каждом последующем обращении используются те значения переменных, которые сохранились от предыдущего обращения. Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдает команду ожидания с указанием условия ожидания. Процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится. Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса или, если уже имеются процессы, ожидающие освобождения данного ресурса, выполнить команду извещения (сигнализации), чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.

Монитор осуществляет доступ к разделяемым ресурсам посредством использования переменных типа условие (conditional). Буде такие переменные называть просто «условие». Для каждой отдельно взятой причины, по которой процесс переводится в состояние ожидания (блокировки), назначается свое условие. На переменных «условие» определены два типа операций: `wait()` и `signal()`, которые по сути являются макрокомандами. Команда `wait()` блокирует процесс, если ресурс занят, и открывает доступ к монитору, если ресурс свободен. Выполнение заблокированного процесса активизируется командой `signal()`, которую вызывает другой процесс. Оператор `signal()` выполняется следующим образом: если очередь к переменной «условие» не пуста, то из очереди выбирается один из процессов и активизируется, иначе если очередь пуста, то `signal()` действий не выполняет.

Использование мониторов имеет ряд преимуществ по сравнению с низкоуровневыми средствами:

- локализация разделяемых переменных внутри тела монитора позволяет избавиться от малопонятных программных конструкций в синхронизируемых процессах;
- мониторы дают возможность процессам совместно использовать программные модули, представляющие критические секции (если несколько процессов совместно и одинаково используют некоторый разделяемый ресурс, то в составе монитора достаточно иметь одну копию соответствующей процедуры работы с этим ресурсом).

1.4 Выбор средств взаимного исключения

1.4.1 Семафоры

Эдсгер Дейкстра в своих работах по взаимодействию параллельных процессов, первая из которых была опубликована в 1965 году, суммировал опыт предыдущих разработок и поднятые в них проблемы и предложил механизм, названный семафором, в качестве средства реализации взаимного исключения.

Семафор – это неотрицательная, защищенная переменная, на которой определены две неделимые операции $P(S)$ и $V(S)$:

- операция $P(S)$ выполняет уменьшение значения семафора на 1, то есть $S = S - 1$, если $S > 0$;
уменьшение невозможно, если $S = 0$, и процесс блокируется в очереди на S до тех пор, пока уменьшение станет возможным;
- операция $V(S)$ выполняет увеличение значения семафора, т.е. $S = S + 1$, и тем самым разблокирует процесс, стоящий первым в очереди к данному семафору в ожидании его освобождения.

Если семафор принимает только два значения, то он называется бинарным или двоичным. Семафор, принимающий значения от 0 до n , называется считающим.

Обычно семафоры реализуются аппаратно и являются объектами ядра системы, но являются объектами высокого уровня, основанными на командах более низкого уровня таких, как `test-and-set`. Блокировка процесса на семафоре и постановка его в очередь выполняется функциями ядра. При этом сами семафоры являются разделяемыми ресурсами. Семафор, как разделяемая переменная, может находиться только в области данных ядра ОС, так как адресные пространства процессов являются защищенными, т.е. к ним закрыт доступ других процессов.

Семафор можно представить как систему вращающихся дверей. «Двери» устроены так, что повернуть их может только выходящий из второй двери. Промежуток между дверями – критическая секция. Когда процесс находится «между дверями», другой процесс попасть туда не может, так как входная дверь заблокирована. Выходя из критической секции, процесс как бы поворачивает обе двери и следующий процесс попадает в свою критическую секцию. Таким образом, процессы стоят в очереди к семафору пассивно – они заблокированы и, следовательно не занимают циклы процессорного времени. Семафоры исключают активное ожидание на процессоре, так как заблокированный процесс активизируется другим процессом, выполнившим операцию освобождения семафора («поворот двери»).

Семафоры используются в задачах «Производство-потребитель» и «Обедающих философов».

Основными проблемами при использовании семафоров являются «гонки» и взаимоблокировки. Использование одиночных семафоров может привести к взаимоблокировке в том случае, когда процесс использует несколько семафоров.

1.4.2 Мьютекс (Mutex)

Для решения проблемы связанной с взаимным исключением между параллельными потоками, выполняющимися в контексте разных процессов, в операционных системах Windows используется объект ядра мьютекс. Объект взаимного исключения (mutual exception), или мьютекс (mutex), обеспечивает более универсальную функциональность по сравнению с объектом CRITICAL_SECTION. Поскольку мьютексы могут иметь имена и дескрипторы, их можно использовать также для синхронизации потоков, принадлежащих различным процессам. Так, два процесса, разделяющие общую память посредством отображения файлов, могут использовать мьютексы для синхронизации доступа к разделяемым областям памяти.

Поток приобретает права владения мьютексом (или блокирует (block) мьютекс) путем вызова функции ожидания (WaitForSingleObject или WaitForMultipleObjects) по отношению к дескриптору мьютекса и уступает эти права посредством вызова функции ReleaseMutex. Необходимо тщательно следить за тем, чтобы потоки своевременно освобождали ресурсы, в которых они больше не нуждаются. Поток может завладевать одним и тем же ресурсом несколько раз, и при этом не будет блокироваться даже в тех случаях, когда уже владеет данным ресурсом.

При работе с мьютексами используются функции CreateMutex, ReleaseMutex и OpenMutex:

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpsa, BOOL  
bInitialOwner, LPCTSTR lpMutexName)
```

```
BOOL ReleaseMutex(HANDLE hMutex)
```

bInitialOwner – если значение этого флага установлено равным True, вызывающий поток немедленно приобретает права владения новым мьютексом. Эта атомарная операция позволяет предотвратить приобретение прав владения мьютексом другими потоками, прежде чем это сделает поток, создающий мьютекс. Как следует из самого его названия (initial owner – исходный владелец), этот флаг не оказывает никакого действия, если мьютекс уже существует.

lpMutexName – указатель на строку, содержащую имя мьютекса; в отличие от файлов имена мьютексов чувствительны к регистру. Если этот параметр равен NULL, то мьютекс создается без имени. События, мьютексы, семафоры, отображения файлов и другие объекты ядра – все они используют одно и то же пространство имен, отличное от пространства имен файловой системы. Поэтому имена всех объектов синхронизации должны быть различными. Длина указанных имен не может превышать 260 символов.

Возвращаемое значение имеет тип HANDLE. Значение NULL указывает на неудачное завершение функции. Функция OpenMutex открывает существующий именованный мьютекс. Эта функция дает возможность потокам, принадлежащим различным процессам, синхронизироваться так, как если бы они принадлежали одному и тому же процессу. Вызову функции OpenMutex в одном процессе должен предшествовать вызов функции CreateMutex в другом процессе. Для семафоров и событий, как и для отображенных файлов, также имеются соответствующие функции Create и Open. При вызове этих функций всегда предполагается, что сначала один процесс, например, сервер, вызывает функцию Create для создания именованного объекта, а затем другие процессы вызывают функцию Open, которая завершается неудачей, если именованный объект к этому моменту еще не был создан. Возможен и такой вариант, когда все процессы самостоятельно используют вызов функции Create с одним и тем же именем, если порядок создания объектов не имеет значения. Функция ReleaseMutex освобождает мьютекс, которым владеет вызывающий поток. Если мьютекс не принадлежит потоку, функция завершается с ошибкой.

1.4.3 Событие (Event)

Событием называется оповещение о некотором выполненном действии. Объекты события используются для того, чтобы сигнализировать другим

потокам о наступлении какого-либо события, например, о появлении нового сообщения.

Важной дополнительной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько потоков. Объекты события делятся на сбрасываемые вручную и автоматически сбрасываемые, и это их свойство устанавливается при вызове функции `CreateEvent`.

Сбрасываемые вручную события (`manual-reset events`) могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в несигнальное состояние программно.

Автоматически сбрасываемые события (`auto-reset event`) сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в сигнальное состояние.

События используют пять новых функций: `CreateEvent`, `OpenEvent`, `SetEvent`, `ResetEvent` и `CreateEvent`.

`HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpSa, BOOL bManualReset, BOOL bInitialState, LPTCSTR lpEventName)`

Чтобы создать событие, сбрасываемое вручную, необходимо установить значение параметра `bManualReset` равным `True`. Точно так же, чтобы сделать начальное состояние события сигнальным, установите равным `True` значение параметра `bInitialState`. Для открытия именованного объекта события используется функция `OpenEvent`, причем это может сделать и другой процесс.

Для управления объектами событий используются следующие три функции:

`BOOL SetEvent(HANDLE hEvent)` – тип события. Определяет, будет ли `Event` переключаемым вручную (`TRUE`) или автоматически (`FALSE`).

`BOOL ResetEvent(HANDLE hEvent)` – задаёт начальное состояние. Если `TRUE`, то объект изначально находится в сигнальном состоянии.

`BOOL PulseEvent(HANDLE hEvent)` – имя события (или `NULL`, если имя не требуется).

Поток может установить событие в сигнальное состояние, используя функцию `SetEvent`. Если событие является автоматически сбрасываемым, то оно автоматически возвращается в несигнальное состояние уже после освобождения только одного из ожидающих потоков. В отсутствие потоков, ожидающих наступления этого события, оно остается в сигнальном состоянии до тех пор, пока такой поток не появится, после чего этот поток сразу же освобождается.

С другой стороны, если событие является сбрасываемым вручную, то оно остается в сигнальном состоянии до тех пор, пока какой-либо поток не вызовет функцию `ResetEvent`, указав дескриптор этого события в качестве аргумента. В это время все ожидающие потоки освобождаются, но до выполнения такого сброса события другие потоки могут как переходить в состояние его ожидания, так и освобождаться.

Функция `PulseEvent` освобождает все потоки, ожидающие наступления сбрасываемого вручную события, но после этого событие сразу же автоматически сбрасывается. В случае же использования автоматически сбрасываемого события функция `PulseEvent` освобождает только один ожидающий поток, если таковые имеются.

Выводы. Таким образом, в поставленной задаче существуют два вида потоков: «читателями», которые могут только читать данные, и «писатели», которые могут модифицировать данные. Читатели могут работать параллельно, поскольку они друг другу не мешают, а писатели могут работать только в режиме монопольного доступа: только один писатель может получить доступ к разделяемой переменной, причем, когда работает писатель, то другие писатели и читатели не могут получить доступ к этой переменной.

2. Конструкторский раздел

2.1 Структура ПО

Структура монитора представляет собой:

```
monitor readers_writers
int readers = 0;
bool writerlock = false;
Condition can_write_event, can_read_event;

void startReading()
{
    if(writerlock || queue_canwrite)
        can_read_event.wait;
    readers = readers + 1;
    can_read_event.signal;
}

void startWriting()
{
    if(readers!= 0 || writerlock)
        can_write_event.wait;
    writerlock = true;
}

void stopReading()
{
    readers = readers -1;
    if(readers == 0)
        can_write_event.signal;
}

void stopWriting()
{
    writerlock = false;
    if(queue_canread)
        can_read_event.signal;
    else
        can_write_event.signal;
}
```

2.2 Схема алгоритма «читателя»

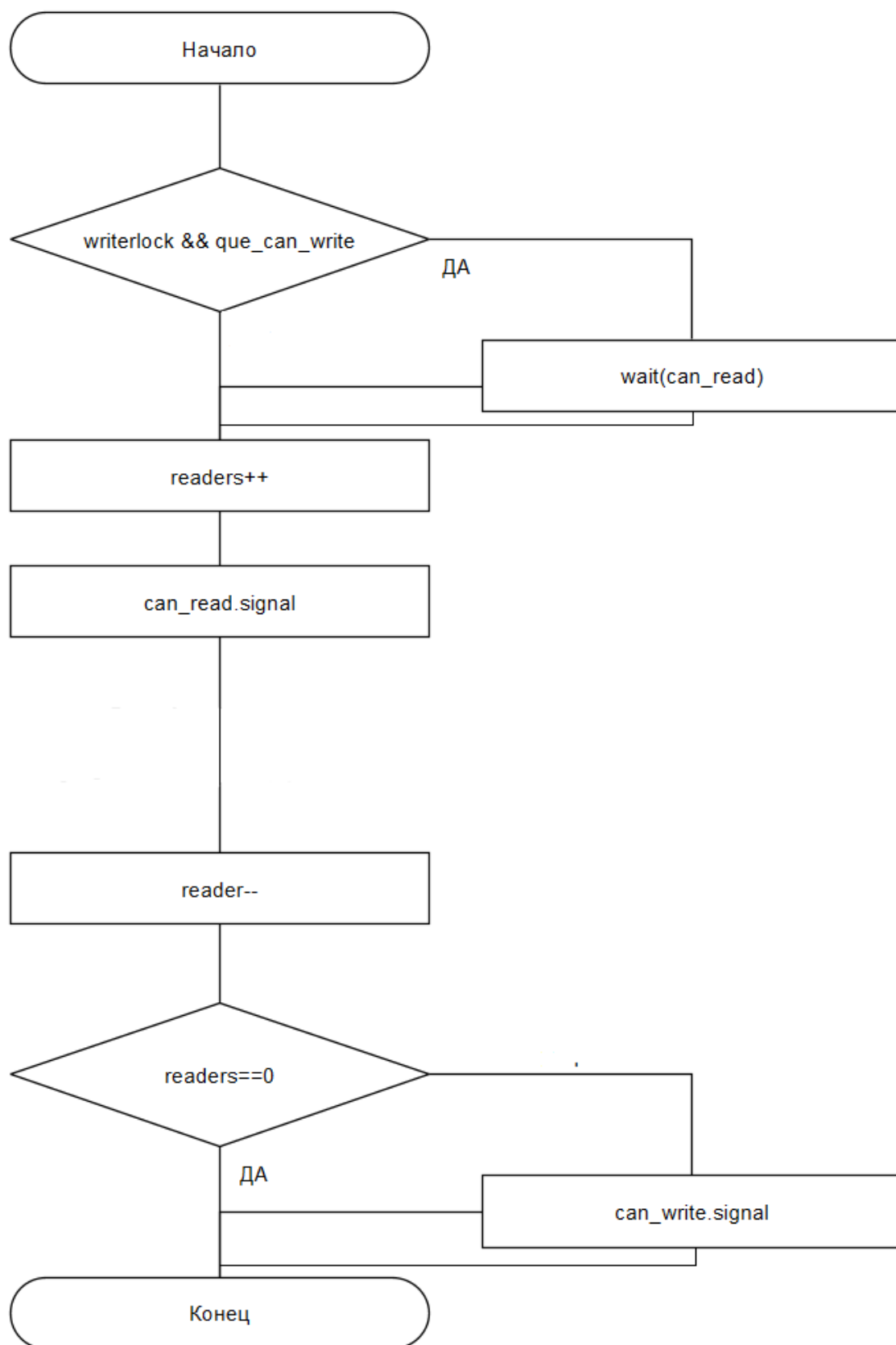


Рис.1 – Схема алгоритма «читателя»

2.3 Схема алгоритма «писателя»

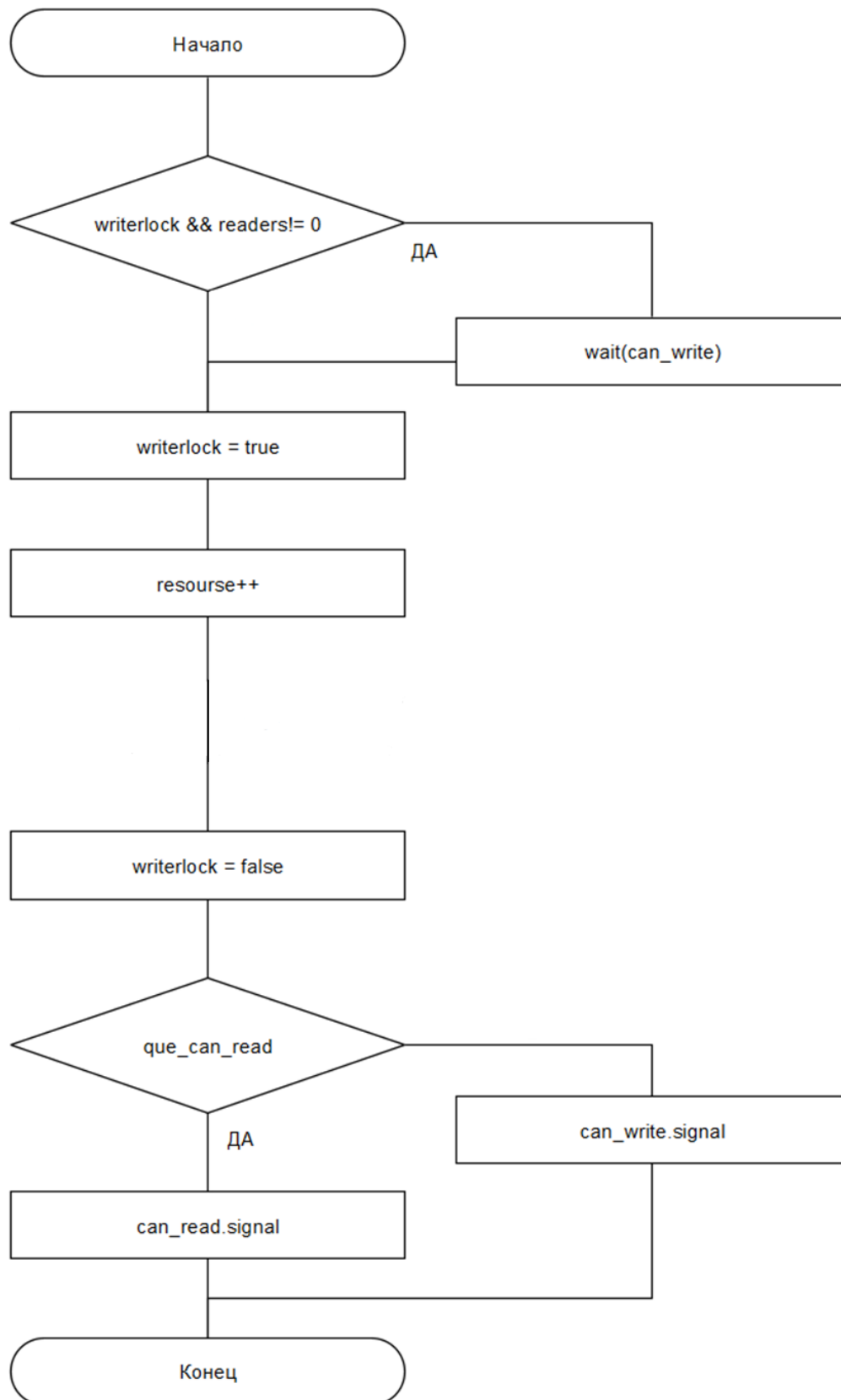


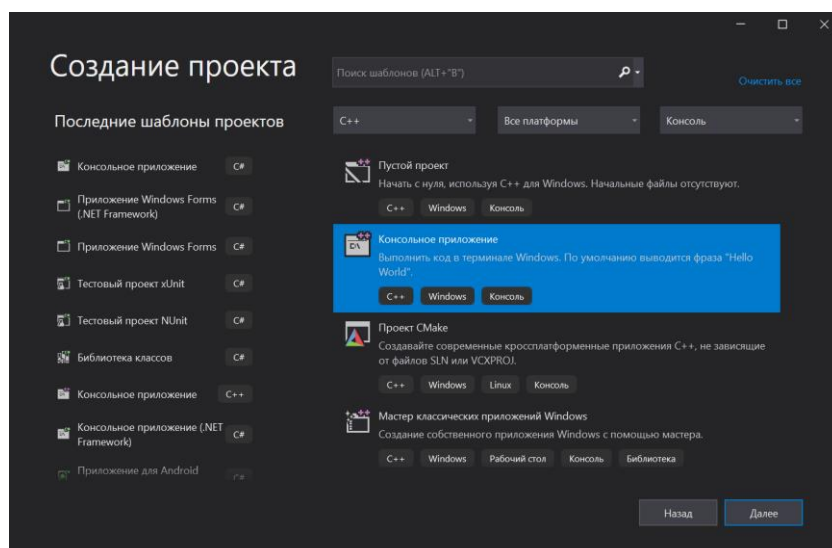
Рис.2 – Схема алгоритма «писателя»

3. Технологический раздел

3.1 Выбор языка и среды программирования

В качестве языка программирования был выбран C++. C++ – компилируемый, статически типизированный язык программирования общего назначения. Поддерживает разные парадигмы программирования: процедурную, обобщённую, функциональную; наибольшее внимание уделено поддержке объектно-ориентированного программирования. C++ широко используется для разработки программного обеспечения, являясь одним из самых популярных языков программирования. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также развлекательных приложений.

В качестве среды разработки была выбрана Visual Studio 2019. Интегрированная среда разработки Microsoft Visual Studio – это стартовая площадка для написания, отладки и сборки кода, а также последующей публикации приложений. Помимо стандартного редактора и отладчика, которые есть в большинстве сред IDE, Visual Studio включает в себя компиляторы, средства автозавершения кода, графические конструкторы и многие другие функции для улучшения процесса разработки.



3.2 Код программы

Для реализации алгоритма «Читатели-писатели» была написана программа. Программа представляет собой чтение данных из переменной и запись данных в переменную.

Листинг программы с комментариями кода на языке C++:

```
#include "stdio.h"
#include <conio.h>
#include <tchar.h>
#include <Windows.h>
#include <process.h>
#include <locale.h>

using namespace std;

const int readers_number = 7;
const int writers_number = 10;
const int waiting_time_writers = 20;
const int waiting_time_readers = 20;
const int max_resource = 500;
bool writerlock = 0;
int queue_canwrite = 0;
int queue_canread = 0;
int readers = 0; // переменная-счётчик количества читателей, защищаемая
мьютексом
int resource = 0; // разделяемая переменная-ресурс, защищаемая event с
автосбросом
HANDLE can_read_event; // событие, пропускающее читателей (со сбросом
вручную)
HANDLE can_write_event; // событие, пропускающее писателей (с
автосбросом)
HANDLE rw_mutex; // мьютекс, защищающий счётчик читателей

// открытие работы читателей
void startReading()
{
    InterlockedIncrement((LONG*)&queue_canread);
    if (writerlock || queue_canwrite)
        WaitForSingleObject(can_read_event, INFINITE);

    WaitForSingleObject(rw_mutex, INFINITE);
    InterlockedDecrement((LONG*)&queue_canread);
    InterlockedIncrement((LONG*)&readers);
    ReleaseMutex(rw_mutex);
}
```

```

// закрытие работы читателей
void stopReading()
{
    InterlockedDecrement((LONG*)&readers);
    if (!queue_canread)
    {
        ResetEvent(can_read_event);
        SetEvent(can_write_event);
    }
}

// открытие работы писателей
void startWriting()
{
    InterlockedIncrement((LONG*)&queue_canwrite);
    if (readers || writerlock)
        WaitForSingleObject(can_write_event, INFINITE);

    InterlockedDecrement((LONG*)&queue_canwrite);
    InterlockedIncrement((LONG*)&writerlock);
}

// закрытие работы писателей
void stopWriting()
{
    InterlockedDecrement((LONG*)&writerlock);
    if (queue_canread)
        SetEvent(can_read_event);
    else
        SetEvent(can_write_event);
}

void read(void* parametr)
{
    int* number = (int*)parametr;
    while (resource < max_resource)
    {
        Sleep(rand() % waiting_time_readers);
        startReading();

        setlocale(LC_ALL, "Russian");
        printf("Читатель номер %d; Ресурс %d\n", *number, resource);
        stopReading();
    }
}

void write(void* parametr)
{
    int* number = (int*)parametr;
    while (resource < max_resource)

```

```

    {
        Sleep(rand() % waiting_time_writers);
        startWriting();
        resource++;

        setlocale(LC_ALL, "Russian");
        printf("Писатель номер: %d; Ресурс %d\n", *number, resource);
        stopWriting();
    }
}

int main()
{
    int rw_count = readers_number + writers_number; // нумерация
    читателей и писателей
    int* rw_num = new int[readers_number + writers_number];
    for (int i = 0; i < rw_count; i++)
    {
        rw_num[i] = i;
    }

    //создание мьютекса и события
    rw_mutex = CreateMutex(NULL, FALSE, NULL); // создание мьютекс
    can_read_event = CreateEvent(NULL, TRUE, TRUE, NULL); // сброс
    вручную
    can_write_event = CreateEvent(NULL, FALSE, TRUE, NULL); // с
    автосбросом

    HANDLE hThread[readers_number + writers_number];
    for (int i = 0; i < rw_count; i++) //запуск читателей и писателей
    {
        void* parametr = &rw_num[i];
        if (i < readers_number)
            hThread[i] = (HANDLE)_beginthread(read, 0, parametr);
        else hThread[i] = (HANDLE)_beginthread(write, 0, parametr);
    }

    _getch();
    return 0;
}

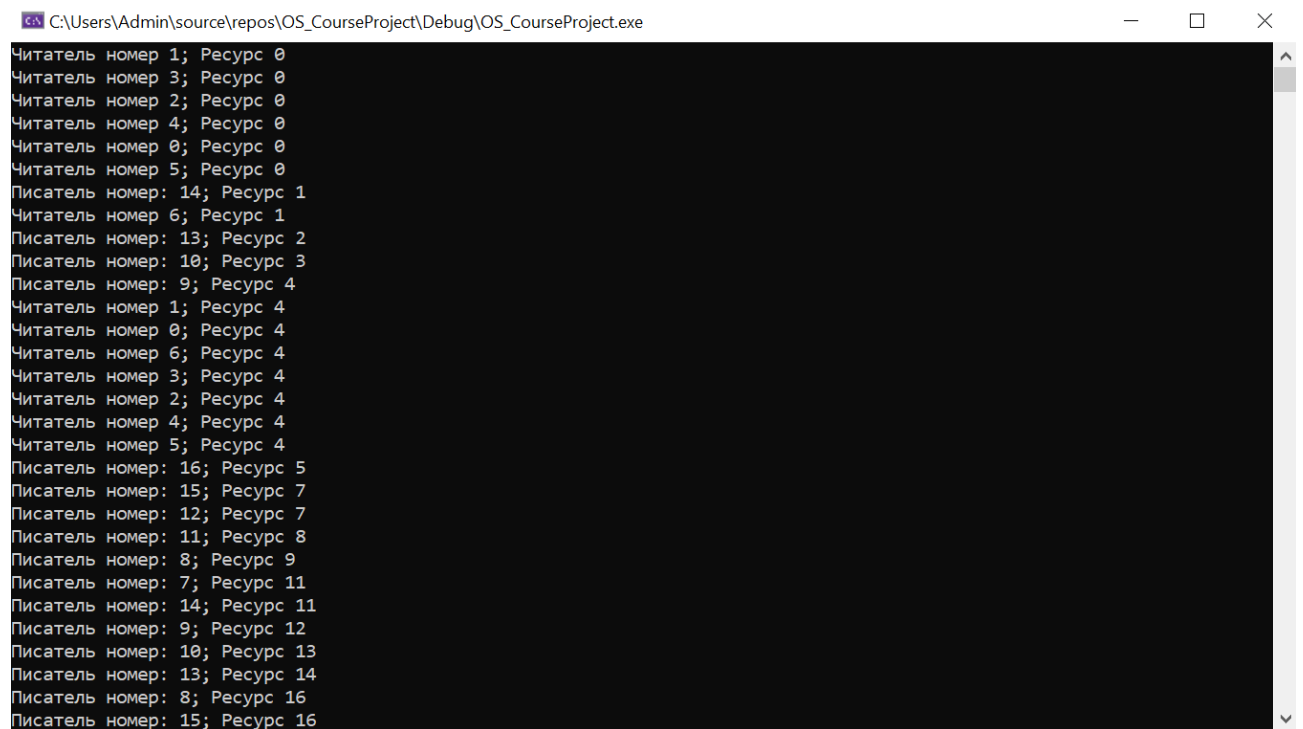
```

4. Исследовательский раздел

4.1 Анализ разработанного ПО

Для работы с монитором нам необходимо создать параллельные потоки, имитирующие работу писателей и читателей. Количество писателей определяется переменной `writers_number`, а количество читателей – переменной `readers_number` (в программе: 7 и 10 соответственно).

В качестве разделяемого ресурса выступает целочисленная переменная `resource`. При каждом обращении к ней читатели должны считывать ее значение, а писатели увеличивать её на единицу. Считывание проходит одновременно всеми читателями, а запись может выполняться только одним писателем в порядке очереди.



```
C:\Users\Admin\source\repos\OS_CourseProject\Debug\OS_CourseProject.exe
Читатель номер 1; Ресурс 0
Читатель номер 3; Ресурс 0
Читатель номер 2; Ресурс 0
Читатель номер 4; Ресурс 0
Читатель номер 0; Ресурс 0
Читатель номер 5; Ресурс 0
Писатель номер: 14; Ресурс 1
Читатель номер 6; Ресурс 1
Писатель номер: 13; Ресурс 2
Писатель номер: 10; Ресурс 3
Писатель номер: 9; Ресурс 4
Читатель номер 1; Ресурс 4
Читатель номер 0; Ресурс 4
Читатель номер 6; Ресурс 4
Читатель номер 3; Ресурс 4
Читатель номер 2; Ресурс 4
Читатель номер 4; Ресурс 4
Читатель номер 5; Ресурс 4
Писатель номер: 16; Ресурс 5
Писатель номер: 15; Ресурс 7
Писатель номер: 12; Ресурс 7
Писатель номер: 11; Ресурс 8
Писатель номер: 8; Ресурс 9
Писатель номер: 7; Ресурс 11
Писатель номер: 14; Ресурс 11
Писатель номер: 9; Ресурс 12
Писатель номер: 10; Ресурс 13
Писатель номер: 13; Ресурс 14
Писатель номер: 8; Ресурс 16
Писатель номер: 15; Ресурс 16
```

Рис. 4 – Начало работы программы

Для доступа к разделяемой переменной (ресурсу) в программе будут использованы события (Event). Для читателей это `can_read_event` (со сбросом

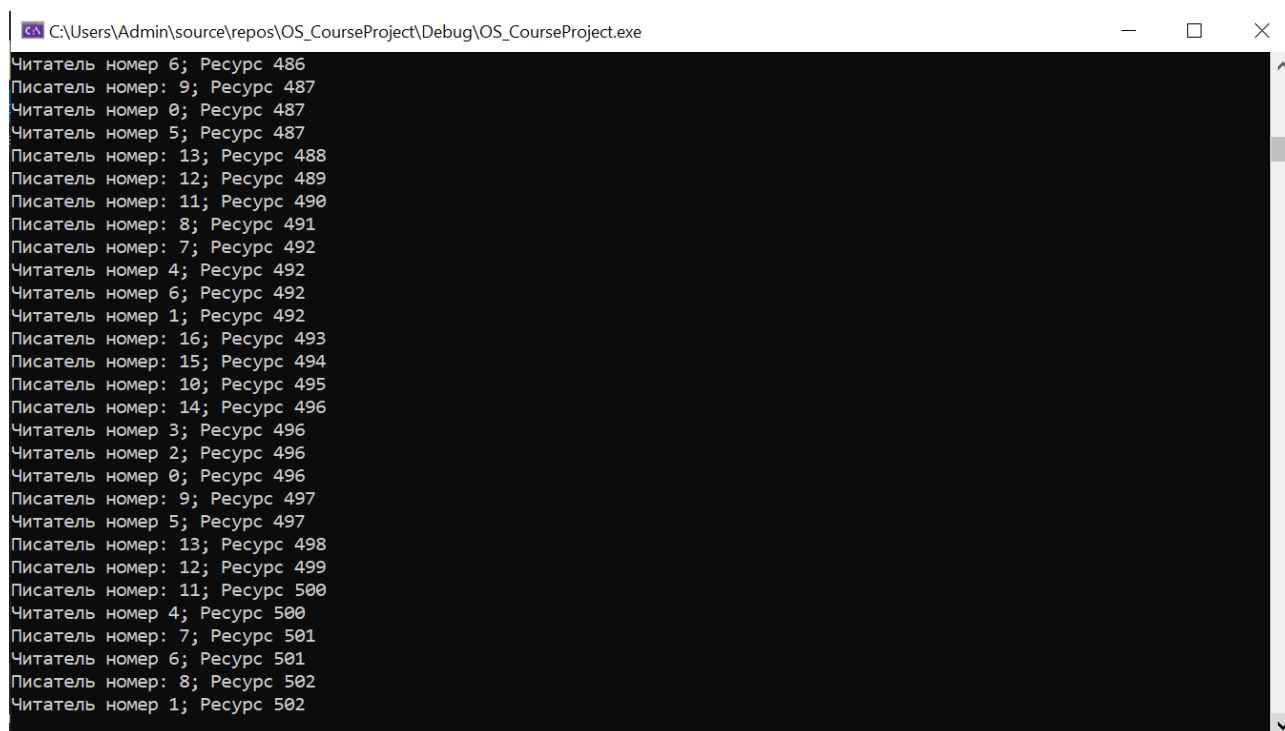
вручную), а для писателей – `can_write_event` (с автоматическим сбросом). Читатели будут ожидать завершения работы всех ждущих писателей (функция `startReading`), а писатели будут ожидать как завершения

работы читателей, так и завершения работы писателей, стоящих в очереди (функция `startWriting`).

Также в программе будет использован мьютекс `rw_mutex` для защиты разделяемого ресурса, чтобы гарантировать, что одновременно два писателя не получают к нему доступ.

Функции `stopReading` и `stopWriting` осуществляют проверки состояния монитора на момент завершения операций считывания или записи ресурсов и, иницииируют соответствующие события (функции `SetEvent`), которые запускают потоки, ожидающие в очереди.

Программа завершает свою работу при условии, что ресурс достиг максимальной отметки (значения) `max_resource` (рис. 5).



```
C:\Users\Admin\source\repos\OS_CourseProject\Debug\OS_CourseProject.exe
Читатель номер 6; Ресурс 486
Писатель номер: 9; Ресурс 487
Читатель номер 0; Ресурс 487
Читатель номер 5; Ресурс 487
Писатель номер: 13; Ресурс 488
Писатель номер: 12; Ресурс 489
Писатель номер: 11; Ресурс 490
Писатель номер: 8; Ресурс 491
Писатель номер: 7; Ресурс 492
Читатель номер 4; Ресурс 492
Читатель номер 6; Ресурс 492
Читатель номер 1; Ресурс 492
Писатель номер: 16; Ресурс 493
Писатель номер: 15; Ресурс 494
Писатель номер: 10; Ресурс 495
Писатель номер: 14; Ресурс 496
Читатель номер 3; Ресурс 496
Читатель номер 2; Ресурс 496
Читатель номер 0; Ресурс 496
Писатель номер: 9; Ресурс 497
Читатель номер 5; Ресурс 497
Писатель номер: 13; Ресурс 498
Писатель номер: 12; Ресурс 499
Писатель номер: 11; Ресурс 500
Читатель номер 4; Ресурс 500
Писатель номер: 7; Ресурс 501
Читатель номер 6; Ресурс 501
Писатель номер: 8; Ресурс 502
Читатель номер 1; Ресурс 502
```

Рис. 5 – Завершение работы программы

Заключение

При выполнении данной курсовой работы были подробно изучена многозадачность операционной системы, позволяющий реализовать одновременное выполнение нескольких программ на одном процессоре.

Работа содержит в себе все необходимые компоненты, приведенные в списке задач. Содержит следующие разделы: аналитический, конструкторский, технологический и исследовательский.

Достигнута цель курсового проекта – приведена классическая задача «Читатели-писатели» и реализовано её решение с помощью монитора Хоара на языке программирования C++.

Список использованных источников информации

1. Операционные системы Т. 2 / Дейтел Х. М., Дейтел П. Дж., Чофнес Д. Р.; пер. с англ. Молявко С. М. – 3-е изд. – М.: Бином, 2009. – 704 с.
2. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64 разрядной версии Windows; пер. с англ. – 4-е изд. – СПб.: Питер; М.: Русская Редакция, 2008. – 752 с.
3. Рязанова Н.Ю. Лекции по операционным системам. МГТУ им. Баумана, 2017.
4. Таненбаум Э., Бос Х. Современные операционные системы; пер. с англ. – СПб.: Питер, 2015. – 1120 с.
5. Харт Дж.М. Системное программирование в среде Windows; пер. с англ. Гузикевич А.Г. – 3-е изд. – М.: Вильямс, 2005. – 592 с.