МИНОБРНАУКИ РОССИИ

федеральное государственное бюджетное образовательное учреждение   
высшего образования   
«Санкт-Петербургский государственный технологический институт  
(технический университет)»  
СПбГТИ(ТУ)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| УГНС | | 09.00.00 | Информатика и вычислительная техника | | |
| Направление подготовки | | 09.04.01 | Информатика и вычислительная техника | | |
| Направленность (профиль) | |  | Системы автоматизированного проектирования | | |
| Форма обучения | |  | очная | | |
|  | |  |  | | |
| Факультет | |  | Информационных технологий и управления | | |
| Кафедра | |  | Систем автоматизированного  проектирования и управления | | |
| Учебная дисциплина | |  | Основы разработки автоматизированных информационных систем | | |
| Курс | 1 | | | Группа | 499м |

Лабораторная работа 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тема: | Лабораторная работа 2 по курсу «Технологии разработки ПО» | | | | |
| Студент | |  |  |  | Плеханов А.А. |
| Преподаватели | |  |  |  | Корниенко И.Г., Федин К.А. |

Санкт-Петербург

2019

**1 Цель работы**

В данной лабораторной работе рассматриваются базовые возможности операционной системы, позволяющие повлиять на производительность разрабатываемого программного обеспечения.

**2 Постановка задачи**

1. Ознакомиться с возможностями операционной системы по созданию многопоточных приложений.
2. Краткое обоснование выбора СУБД для реализации даталогической модели, разработанной в предыдущей работе.
3. Разработка базы данных в выбранной СУБД, наполнение данными.
4. Пояснить полученные результаты.

**3 Разработка многопоточных приложений и их проблемы**

Поток – это главный интерфейс в многопоточном программировании, это последовательность инструкций, выполняемых в пределах контекста процесса.

Многопоточное программирование позволяет улучшить чувствительность приложения, улучшить структуру программы, использовать меньше ресурсов системы.

**3.1 Пример программы с созданными потоками**

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

// Метод Coundown считает от 1000 до 1

public static void Coundown()

{

for (int counter = 1000; counter > 0; counter--)

Console.Write(counter.ToString() + “ “);

}

static void Main()

{

// Создание второго потока

Thread t2 = new Thread(new ThreadStart(Coundown));

// Запуск потока

t2.Start();

// В это же время метод Coundown вызывается в основном потоке

Coundown();

Console.ReadLine();

}

}

}

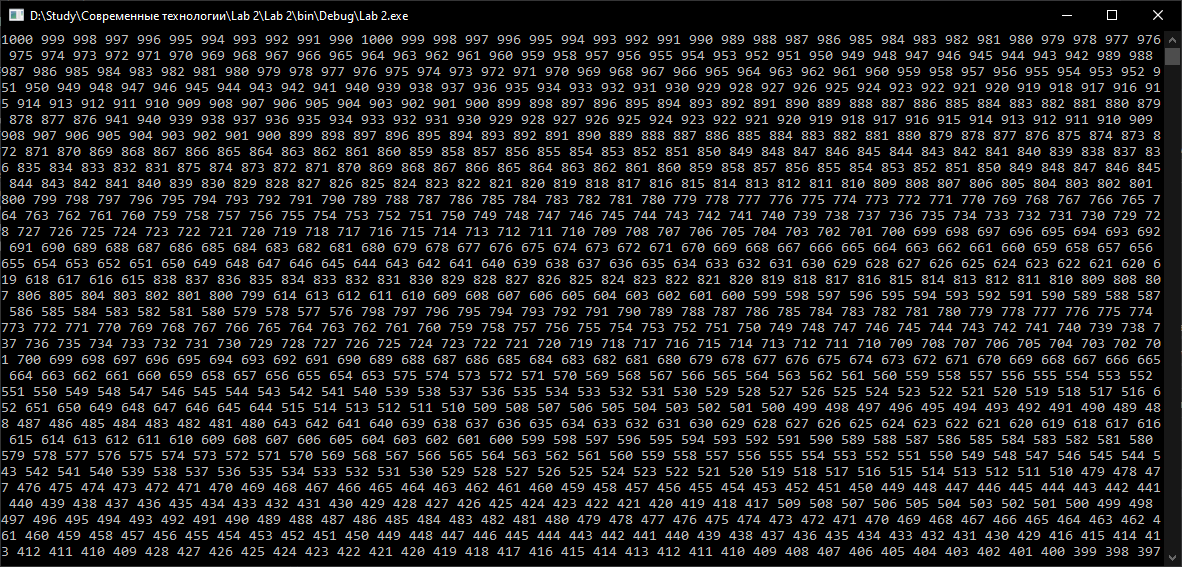


Рисунок 1 – Результат выполнения программы

Информация, выводимая первым и вторым потоками, перемешена, так как хоть потоки выполняются одновременно, но результаты выводятся на один дисплей компьютера.

3.2 Приоритеты потоков

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

// Метод Coundown считает от 1000 до 1

public static void Coundown()

{

for (int counter = 1000; counter > 0; counter--)

Console.Write(counter.ToString() + " ");

}

static void Main()

{

// Создание второго потока

Thread t2 = new Thread(new ThreadStart(Coundown));

// Приоритет потока

t2.Priority = ThreadPriority.Highest;

// Приоритет текущего потока - низший

Thread.CurrentThread.Priority = ThreadPriority.Lowest;

// Запуск потока

t2.Start();

// В это же время метод Coundown вызывается в основном потоке

for (int counter = 1000; counter > 0; counter--)

Console.Write(counter.ToString() + "a");

Console.ReadLine();

}

}

}

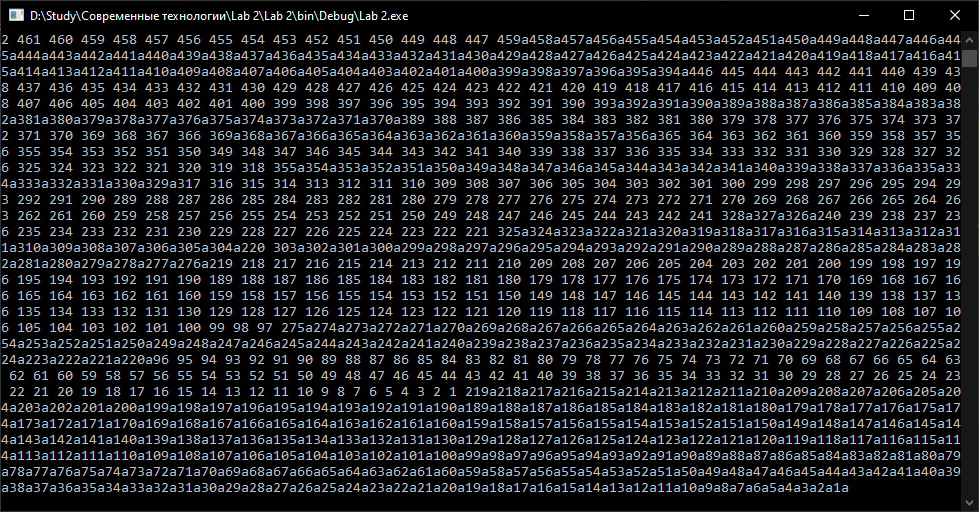


Рисунок 2 – Результаты программы при разных приоритетах потоков

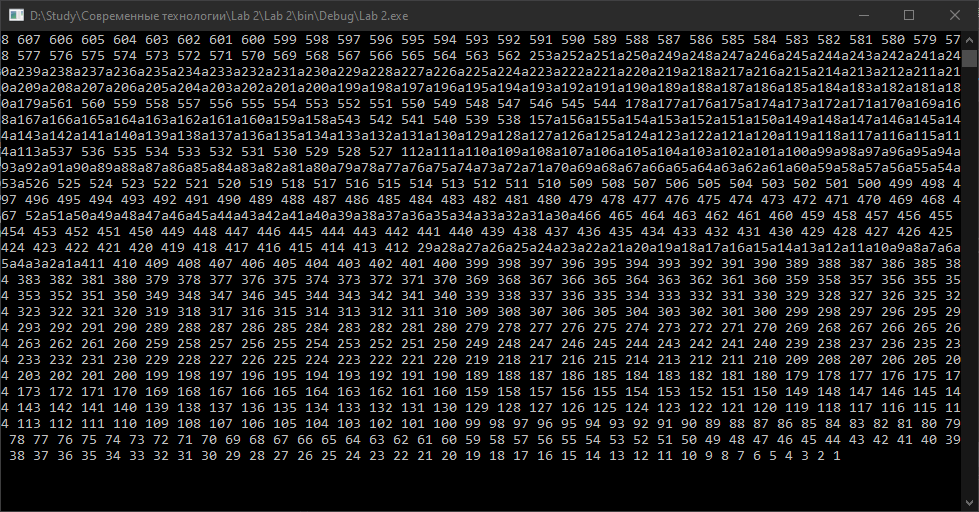


Рисунок 3 – Одинаковые приоритеты потоков

На рисунке 2 видно, что поток с высшим приоритетом закончился раньше, чем поток с низшим. На рисунке 3 показан результат при выполнении программы с одинаковыми приоритетами потоков, результат похож на результаты на рисунке 1.

3.3 Анализ состояния потока

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

// Метод Countdown считает от 10 до 1

public static void Countdown()

{

for (int counter = 10; counter > 0; counter--)

Console.Write(counter.ToString() + " ");

Console.WriteLine();

}

// Метод DumpThreadState выводит текущее состояние потока

// ThreadState - битовая маска состояния потока

public static void DumpThreadState(Thread t)

{

Console.Write("Текущее состояние: ");

if ((t.ThreadState & ThreadState.Aborted) == ThreadState.Aborted)

Console.Write("Отменен ");

if ((t.ThreadState & ThreadState.AbortRequested) == ThreadState.AbortRequested)

Console.Write("Запрос на отмену ");

if ((t.ThreadState & ThreadState.Background) == ThreadState.Background)

Console.Write("Выполняется в фоновом режиме ");

if ((t.ThreadState & (ThreadState.Stopped | ThreadState.Unstarted | ThreadState.Aborted)) == 0)

Console.Write("Выполняется ");

if ((t.ThreadState & ThreadState.Stopped) == ThreadState.Stopped)

Console.Write("Остановлен ");

if ((t.ThreadState & ThreadState.StopRequested) == ThreadState.StopRequested)

Console.Write("Запрос на остановку ");

if ((t.ThreadState & ThreadState.Suspended) == ThreadState.Suspended)

Console.Write("Приостановлен ");

if ((t.ThreadState & ThreadState.SuspendRequested) == ThreadState.SuspendRequested)

Console.Write("апрос на приостановку ");

if ((t.ThreadState & ThreadState.Unstarted) == ThreadState.Unstarted)

Console.Write("Не запущен ");

if ((t.ThreadState & ThreadState.WaitSleepJoin) == ThreadState.WaitSleepJoin)

Console.Write("Ожидает освобождения ресурсов ");

Console.WriteLine();

}

static void Main()

{

// Создание еще одного потока

Thread t2 = new Thread(new ThreadStart(Countdown));

DumpThreadState(t2);

// Запуск нового потока

t2.Start();

DumpThreadState(t2);

// В это же время метод Countdown вызывается в

// основном потоке

Countdown();

// Отмена второго потока

t2.Abort();

DumpThreadState(t2);

Console.ReadLine();

}

}

}

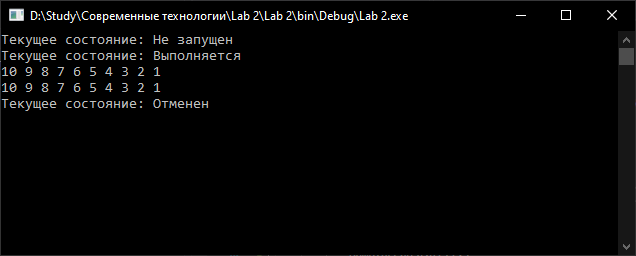


Рисунок 4 – Результаты анализа состояния потока

На рисунке 4 видно, что после создания потока, он находится в состоянии не запущен. После вызова метода старта потока, он переходит в состояние выполняется. После вызова метода отмены потока, проверяется его состояние, выводится: отменен.

3.4 Порядок выполнения потоков

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

public static void CheckTime(Object state)

{

Console.WriteLine(DateTime.Now);

}

static void Main()

{

// Создание делегата, который будет вызываться объектом Timer

TimerCallback tc = new TimerCallback(CheckTime);

// Создание таймера, срабатывающего дважды в секунду

// Первый запуск произойдет через одну секунду

Timer t = new Timer(tc, null, 1000, 500);

// Ожидание ввода пользователя

Console.WriteLine("Нажмите Enter для выхода");

Console.Read();

// Освобождение ресурсов

t.Dispose();

t = null;

}

}

}



Рисунок 5 – Результат использования класса Timer

На рисунке 5 метод CheckTime вызывается спустя 1 секунду с периодом 0,5 секунды.

3.5 Последовательное выполнение потоков

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

// Метод Countdown считает от 100 до 1

public static void Countdown()

{

for (int counter = 100; counter > 0; counter--)

Console.Write(counter.ToString() + " ");

}

// Метод Countdown\_l считает от 100 до 1 с буквой А

public static void Countdown\_1()

{

for (int counter = 100; counter > 0; counter--)

Console.Write(counter.ToString() + "A");

}

static void Main()

{

// Создание второго потока

Thread t2 = new Thread(new ThreadStart(Countdown));

// Запуск второго потока

t2.Start();

// Блокировка первого потока до завершения

//второго

t2.Join();

// Вызов метода Countdown\_l из первого потока

Countdown\_1();

Console.Read();

}

}

}

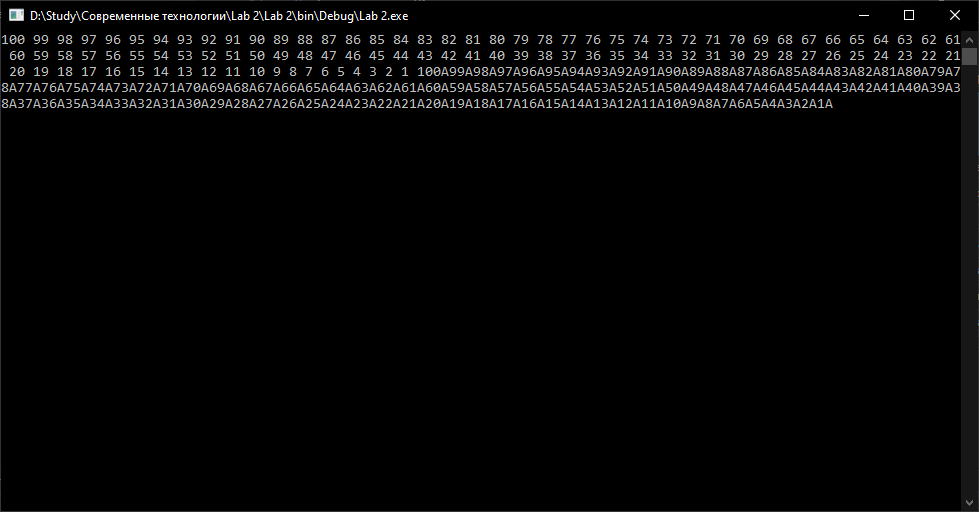


Рисунок 6 – Результат использования метода Join

На рисунке 6 видно, что потоки не пересекаются, так как был вызван метод Join, который приостановил основной поток, до завершения второго.

3.6 Критические секции

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

// Общий счетчик

int Runs = 0;

// Метод CountUp увеличивает значение общего счетчика

public void CountUp()

{

while (Runs <= 10)

{

int Temp = Runs;

Temp++;

Console.WriteLine(Thread.CurrentThread.Name + " " +Temp);

Thread.Sleep(1000);

Runs = Temp;

}

}

public void RunThreads()

{

// Создание и запуск двух потоков

Thread t2 = new Thread(new ThreadStart(CountUp));

t2.Name = "t2";

Thread t3 = new Thread(new ThreadStart(CountUp));

t3.Name = "t3";

t2.Start();

t3.Start();

}

static void Main()

{

// Создание экземпляра текущего класса

Program Section = new Program();

// Выполнение тестов

Section.RunThreads(); Console.Read();

}

}

}



Рисунок 7 – Результат многопоточной работы с проблемами критической секции

* Поток t2 присваивает переменной Temp значение переменной Runs и увеличивает Temp на единицу.
* Поток t2 приостанавливает свое выполнение на одну секунду.
* Планировщик передает управление потоку t3.
* Поток t3 присваивает переменной Temp значение переменной Runs и увеличивает Temp на единицу. Здесь используются те же значения, что ив потоке t2, т.к. поток t2 еще не успел изменить значение общего счетчика (он «заснул» на 1000 миллисекунд).
* Поток t3 приостанавливает свое выполнение на одну секунду.
* Поток t2 возобновляет выполнение и присваивает новое значение переменной Runs.
* Поток t3 возобновляет выполнение и присваивает новое значение переменной Runs, изменяя значение, присвоенное потоком t2.
* Цикл повторяется.

Планировщик может переключиться в любом месте цикла, перед тем как переменной Runs присвоено новое значение, если не использовать метод Sleep.

3.7 Блокировка

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

// Общий счетчик

int Runs = 0;

// Метод CountUp увеличивает значение общего счетчика

public void CountUp()

{

while (Runs < 10)

{

lock (this)

{

int Temp = Runs;

Temp++;

Console.WriteLine(Thread.CurrentThread.Name + " " +Temp);

Thread.Sleep(1000);

Runs = Temp;

}

}

}

public void RunThreads()

{

// Создание и запуск двух потоков

Thread t2 = new Thread(new ThreadStart(CountUp));

t2.Name = "t2";

Thread t3 = new Thread(new ThreadStart(CountUp));

t3.Name = "t3";

t2.Start();

t3.Start();

}

static void Main()

{

// Создание экземпляра текущего класса

Program Section = new Program();

// Выполнение тестов

Section.RunThreads();

Console.Read();

}

}

}

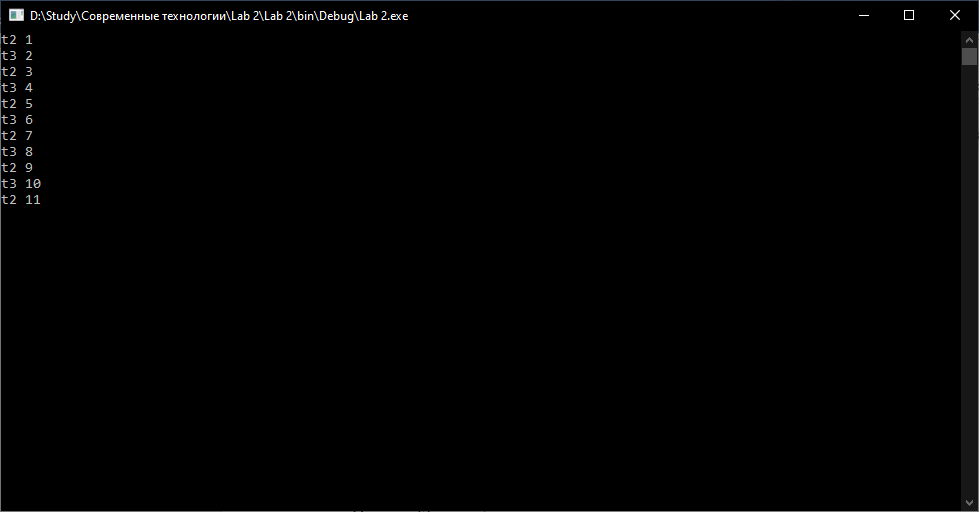


Рисунок 8 – Результат выделения критической секции с помощью Lock

По рисунку 8 видно, что проблема с критической секцией была решена, теперь переменная Runs изменяется правильно, без повторного изменения.

3.8 Монитор

using System;

using System.Threading;

namespace Lab\_2

{

class Program

{

// Общий счетчик

int Runs = 0;

// Метод CountUp увеличивает значение общего счетчика

public void CountUp()

{

while (Runs < 10)

{

Monitor.Enter(this);

int Temp = Runs;

Temp++;

Console.WriteLine(Thread.CurrentThread.Name + " " + Temp);

Thread.Sleep(1000);

Runs = Temp;

Monitor.Exit(this);

}

}

public void RunThreads()

{

// Создание и запуск двух потоков

Thread t2 = new Thread(new ThreadStart(CountUp));

t2.Name = "t2";

Thread t3 = new Thread(new ThreadStart(CountUp));

t3.Name = "t3";

t2.Start();

t3.Start();

}

static void Main()

{

// Создание экземпляра текущего класса

Program Section = new Program();

// Выполнение тестов

Section.RunThreads();

Console.Read();

}

}

}



Рисунок 9 – Результат использования класса монитор

Результат на рисунке 9 не отличается от результата на рисунке 8.

3.9 Семафоры

using System;

using System.Threading;

namespace ConsoleApplication9

{

class Semafor

{

// Общий счетчик

static int Runs = 0;

// Семафор

static Mutex mtx;

// Метод CountUp увеличивает значение общего счетчика

public static void CountUp()

{

while (Runs < 10)

{

// Получение семафора

mtx.WaitOne();

int Temp = Runs;

Temp++;

Console.WriteLine(Thread.CurrentThread.Name + " " +Temp);

Runs = Temp;

// Освобождение семафора

mtx.ReleaseMutex();

Thread.Sleep(1000);

}

}

static void Main()

{

// Создание семафора

mtx = new Mutex(false, "RunsMutex");

// Создание и запуск двух потоков

Thread t2 = new Thread(new ThreadStart(CountUp));

t2.Name = "t2";

Thread t3 = new Thread(new ThreadStart(CountUp));

t3.Name = "t3";

t3.Start();

t2.Start();

Console.Read();

}

}

}



Рисунок 10 – Результат использования семафора

На рисунке 10 видно, что семафор позволяет синхронизировать оба потока. Результат отличается от результата на рисунках 9 и 8, так как метод Sleep вызывается за блоком синхронизации, что позволяет задержать основной поток, пока второй не изменит значение Runs, после чего основной поток продолжает цикл, а второй приостановлен.

3.10 Взаимоблокировка

using System;

using System.Threading;

namespace ConsoleApplication9

{

class SectionDeadlock

{

// Объявление полей - общие строки

private static string Runs; // строка цифр от 0 до 9

private static string Runsl; // строка букв от а до к

// Объявление конструктора

SectionDeadlock(string A, string В)

{

Runs = A;

Runsl = В;

}

// Метод CountUp увеличивает значение в строке Runs от 0 до 10

public void CountUp()

{

lock (Runsl)

{

for (int i = 1; i <= 10; i++)

{

lock (Runs)

{

string Res = Runs.Substring(i - 1, 1);

Console.WriteLine(Thread.CurrentThread.Name + " " + Res); Thread.Sleep(1000);

}

}

}

}

// Метод Char выдает 10 букв в порядке, записанном в строке Runsl

public void Char()

{

lock (Runs)

{

for (int i = 1; i <= 10; i++)

{

lock (Runsl)

{

string Resl = Runsl.Substring(i - 1, 1);

Console.WriteLine(Thread.CurrentThread.Name + " " + Resl);

Thread.Sleep(1000);

}

}

}

}

static void Main()

{

// Создание экземпляра текущего класса

SectionDeadlock Section = new SectionDeadlock("0123456789", "абвгдежзик");

SectionDeadlock Sectionl = new SectionDeadlock("0123456789", "абвгдежзик");

// Создание и запуск двух потоков

Thread t2 = new Thread(new ThreadStart(Section.CountUp));

t2.Name = "t2";

Thread t3 = new Thread(new ThreadStart(Sectionl.Char));

t3.Name = "t3";

t2.Start();

t3.Start();

}

}

}



Рисунок 11 – Результат взаимоблокировки

В программе, создается два объекта Section и Section1 которые используют общие данные (Runs и Runsl) и два метода для работы с ними. Поток t2 использует метод Count Up, в котором предварительно блокируется Runsl, и производится работа с Runs, а поток t3 использует метод Char, в котором блокируется Runs и производится работа с Runsl. Первым получает управление поток t2. Он блокирует Runsl, печатает цифру 1 и засыпает. Управление получает процесс t3. Он блокирует Runs, но не печатает первую букву Runsl, поскольку поток t2 заблокировал Runsl. Управление переходит к потоку t2. Однако он не может выполняться, т.к. Runs заблокирован потоком t3. Налицо тупиковая ситуация.

**4 Выбор СУБД для реализации базы производственных данных для программного комплекса интеллектуального анализа данных**

Выбор системы управления базами данных (СУБД) представляет собой сложную задачу и является одним из важных этапов при разработке баз данных. Выбранная СУБД должна удовлетворять как текущим, так и будущим потребностям разрабатываемого решения, при этом стоит учитывать финансовые затраты на приобретения сервера баз данных, самой системы, разработку решения на основе выбранной СУБД и обучение персонала.

Далее будут рассмотрены основные модели СУБД.

4.1 Реляционные системы управления базами данных

На сегодняшний день являются наиболее распространенными. Представляют собой набор данных со связями между ними: в виде таблиц хранится информация об объектах, представленных в БД. В каждом столбце хранится определённый тип данных, в каждой ячейке – значение атрибута; строки – это набор связанных значений, относящихся к одному объекту/сущности. Каждая строка таблицы помечается уникальным идентификатором, который называют первичным ключом, а строки нескольких таблиц могут быть связаны первичным ключом. Подобное представление позволяет минимизировать избыточность хранимой информации. Недостатком РСУБД является медленный доступ к данным при их большом количестве, однако данную проблему можно решить с помощью добавления индексов и оптимизации запросов к БД. Наиболее распространенными представителями данной группы СУБД являются Oracle, Microsoft SQL Server и MySQL.

4.2 Технология NoSQL

NoSQL –термин, который используют для описания нереляционных баз данных. В NoSQL БД используются различные модели данных, в том числе: документарная, графовая, столбчатая, ключ-значение. Технология NoSQL получила широкое распространение в связи с возможностью масштабирования, высокой доступностью и гибкостью.

Основными недостатками нереляционных баз данных являются ограниченная ёмкость встроенного языка запросов (меньшая функциональность по сравнению с SQL), отсутствие большого числа специалистов по NoSQL, что затрудняет поддержку нереляционных БД в долгосрочной перспективе. Наиболее известными представителями NoSQL являются MongoDB и Neo4J.

Поскольку логические требования к хранимым данным определены заранее, требуется хорошо зарекомендовавшая себя технология с возможностью технической поддержки, решено использовать реляционные БД. В обзоре представлены следующие системы управления базами данных: Oracle, MS SQL MySQL, так как именно они являются самыми популярными и распространёнными СУБД среди РСУБД.

4.3 Oracle Database

Oracle Database — это полнофункциональное решение по управлению данными, ориентированное на средний бизнес. Серверная база данных позволяет для начала приобретать только самое необходимое, а затем масштабировать систему с помощью кластеров Oracle Real Application Clusters по мере роста задач организации. При функционировании на кластере рабочая нагрузка автоматически балансируется по доступным машинам в целях распределения задач между всем существующим оборудованием. Совместима с ОС семейства Windows и UNIX. К минусам можно отнести необходимость оплаты техподдержки отдельно от цены продукта.

4.4 MS SQL

Microsoft SQL server — реляционная система управления базами данных, поддерживающая множество возможностей управления транзакциями и бизнес-аналитику в корпоративных сетях. Имеет свой собственный язык программирования Transact-SQL (реализация SQL от Microsoft, добавляющая дополнительные расширения к стандартному SQL). Имеет большое число решений, направленное на защиту хранимой информации. Позволяет вносить изменения в структуру данных, а также производить резервное копирование данных во время работы сервера. К недостаткам можно отнести отсутствие ранних версий MS SQL на платформах, отличных от Windows (поддержка Linux появилась только в версии 2017 года).

4.5 MySQL

MySQL — свободная РСУБД, имеющая популярность в мире база данных с открытым кодом.

По имеющейся функциональности MySQL приближается к ведущим коммерческим РСУБД. Программное обеспечение MySQL представляет собой многопоточный, многопользовательский SQL-сервер баз данных. Программное обеспечение MySQL имеет двойное лицензирование. Это означает, что пользователи могут выбирать, использовать ли ПО MySQL бесплатно по общедоступной лицензии GNU General Public License (GPL) или приобрести одну из стандартных коммерческих лицензий MySQL AB. В настоящее время основным разработчиком MySQL является компания Oracle. К недостаткам можно отнести сравнительно медленную разработку.

В таблице 1 приведены сравнительные характеристики СУБД.

Таблица 1 - Сравнительная таблица СУБД

|  |  |  |  |
| --- | --- | --- | --- |
| **Характеристика** | **Oracle database** | **MS SQL Server** | **MySQL** |
| Поддерживаемые ОС | Windows, UNIX-подобные | Windows, Linux | Windows, UNIX-подобные |
| Лицензия | Коммерческая | Коммерческая, имеется бесплатная версия | GPL, Коммерческая |
| Степень интеграции с существующим проектом | Низкая | Высокая | Средняя |

В связи с наличием бесплатной версии, а также разработкой под ОС Microsoft Windows Server и прочими критериями, приведенным выше, была выбрана СУБД MS SQL Server 2017.

**5 Реализация базы производственных данных**

Через программное обеспечение Microsoft SQL Server Management Studio 2017 можно работать с базой данных MS SQL Server 2017. Воспроизведем сущности и их связи, как было указано в даталогической модели в лабораторной работе 1. На рисунках 12 – 15 показаны процессы создания сущностей базы производственных данных.

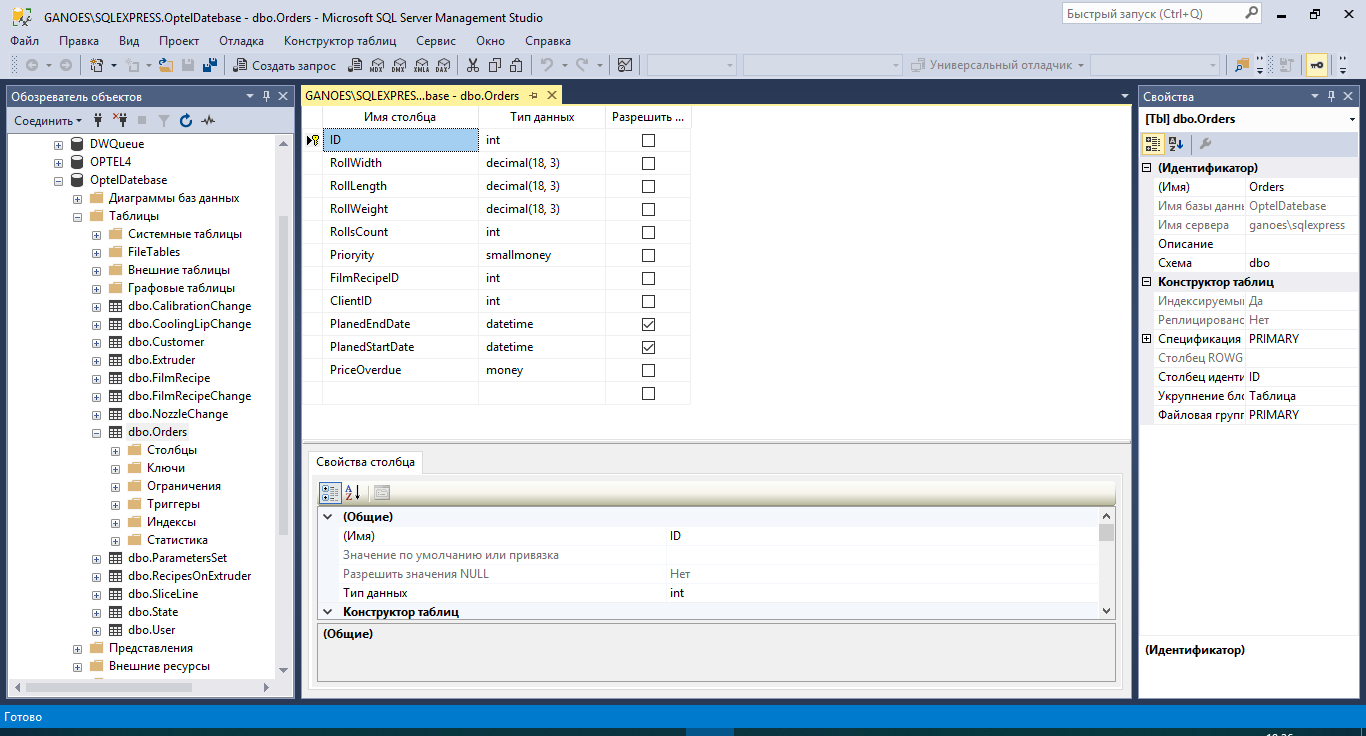


Рисунок 12 – Проектирование сущности заказов

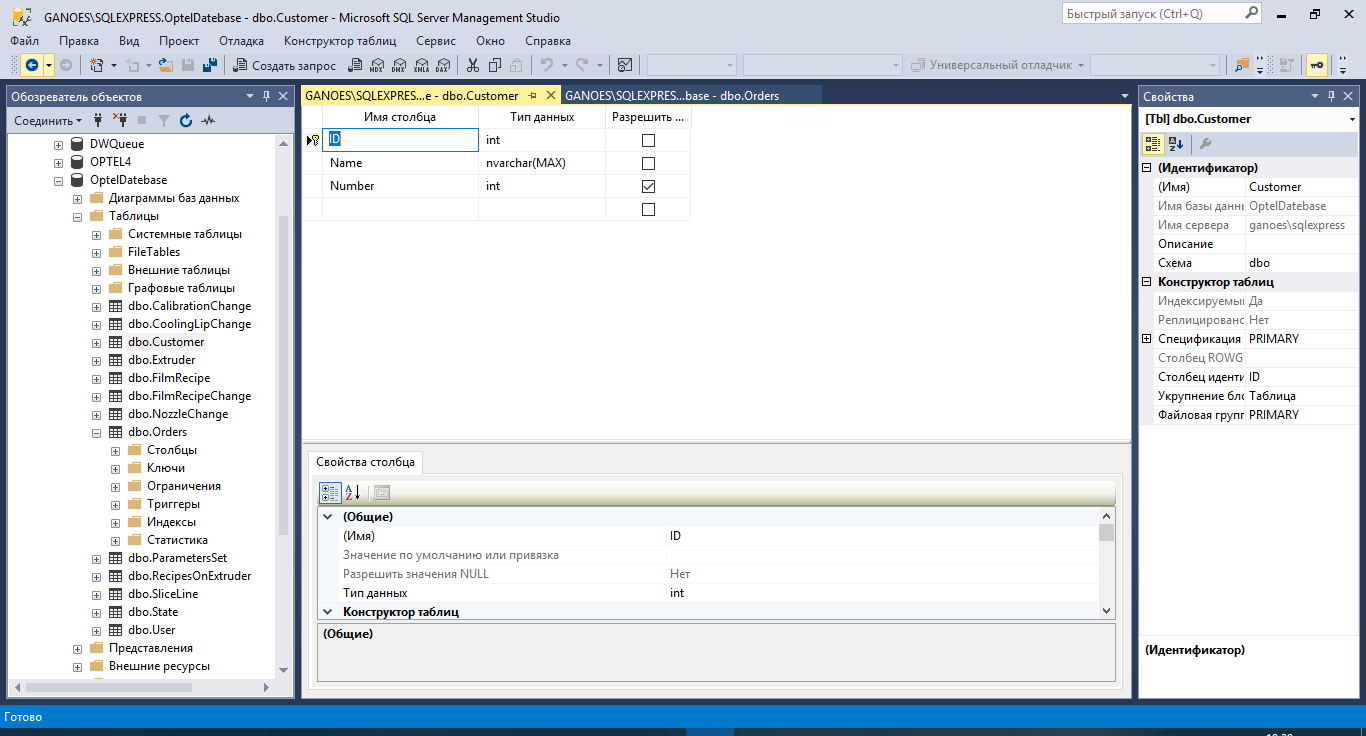


Рисунок 13 – Проектирование сущности клиентов

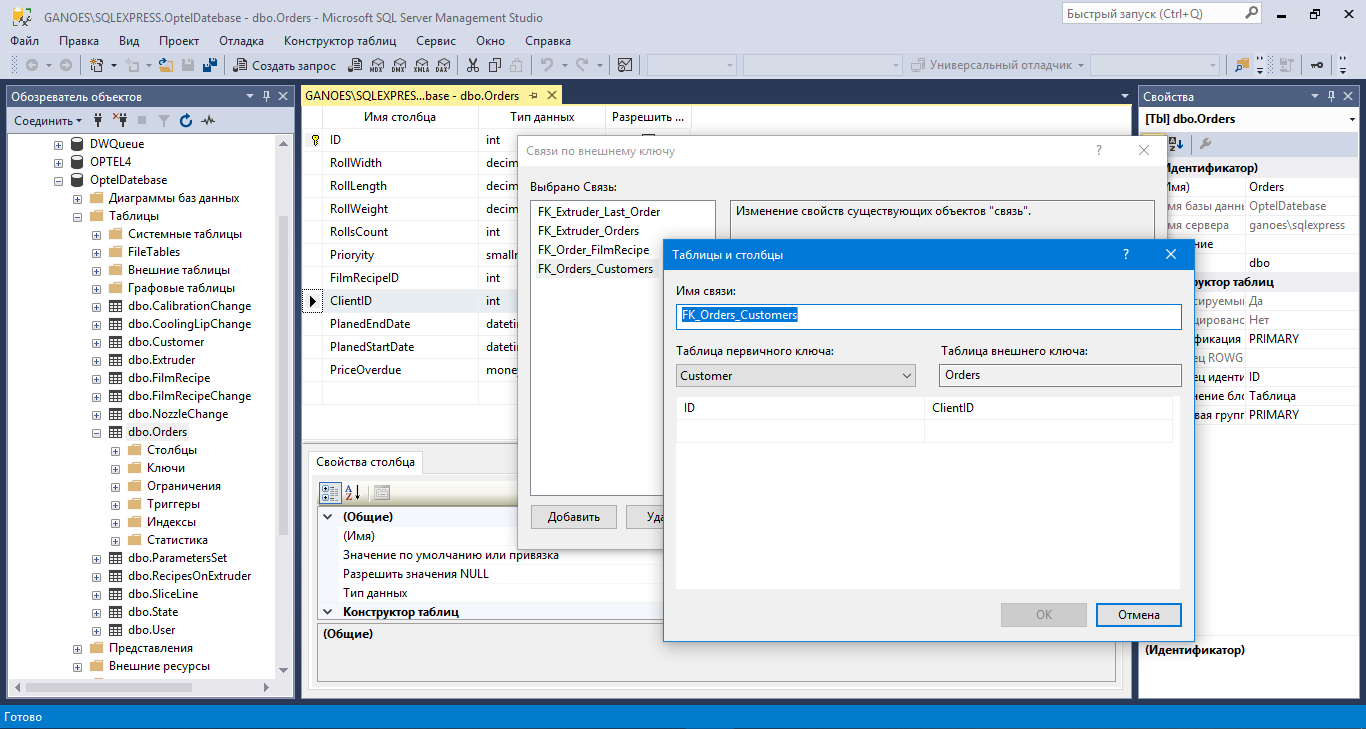


Рисунок 14 – Создание связи клиент – заказы

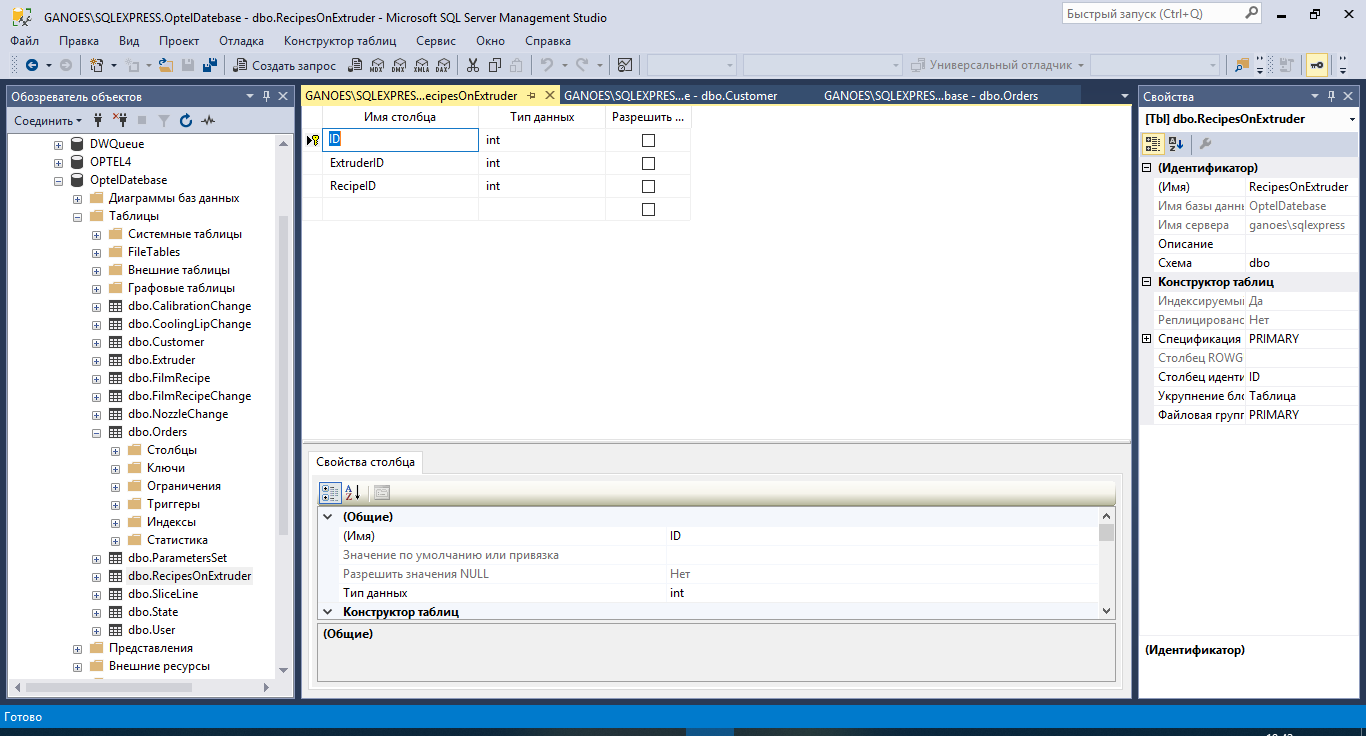


Рисунок 15 – Проектирование таблицы для связи многие ко многим между сущностями экструдеров и рецептов

**Выводы**

В ходе выполнения данной лабораторной работы был разобран механизм для создания многопотоковых приложений на языке C#, с помощью класса Thread, расставление приоритетов потокам, получение состояния потоков, с помощью ThreadState, использование класса Timer, для отложенного и периодического запуска, синхронизация потоков, используя метод Join, блокировки, монитор, семафоры, а также рассмотрена ситуация взаимоблокировки.

Во второй половине данной лабораторной работы была выбрана СУБД MS SQL Server 2017 и реализована база производственных данных для программного комплекса оптимизации многоассортиментного производства полимерных материалов.