**СОДЕРЖАНИЕ**

[Введение 2](#_Toc131708312)

[Задание №1 3](#_Toc131708313)

[Условие задания 3](#_Toc131708314)

[Решение 4](#_Toc131708315)

[Задание №2 12](#_Toc131708316)

[Условие задания 12](#_Toc131708317)

[Решение 12](#_Toc131708318)

[Задание №3 18](#_Toc131708319)

[Условие задания 18](#_Toc131708320)

[Решение 18](#_Toc131708321)

[Задание №4 23](#_Toc131708322)

[Условие задания 23](#_Toc131708323)

[Решение 23](#_Toc131708324)

[Заключение 27](#_Toc131708325)

[Список литературы 28](#_Toc131708326)

[Приложение А 29](#_Toc131708327)

[Приложение Б 30](#_Toc131708328)

# ВВЕДЕНИЕ

Существуют мультипрограммные и мультипроцессорные системы. В данных системах возникает потребность в синхронизации процессов и потоков. Синхронизация требуется для исключения эффектов «гонок» и «тупиков». Средство синхронизации относится к сервису ОС, IPC.

В общем случае процессы и потоки выполняются синхронно и независимо друг от друга. В прикладных задачах часто возникает необходимость согласования скоростей выполнения процессов и потоков, т.е. иногда нужно их приостанавливать до наступления определенного события и в дальнейшем активизировать их.

Ситуация когда два или более процессов или потоков образуют разделяемые данные и конечный результат зависит от соотношения скоростей потоков называется «гонкой».

В мультипрограммных системах «тупик» возникает из-за неправильного обращения к данным.

Рассмотрим несколько задач по тупикам, взаимным блокировкам.

Целью данной курсовой работы является изучение основных принципов работы ОС, ознакомление с алгоритмами работы планировщиков, построение схем арбитража, а также изучение основ синхронизации процессов.

Курсовая работа состоит из приведенных ниже 4-х заданий, варианты которых индивидуальны для каждого из студентов.

# Задание №1

## Условие задания

**а) Исходные данные.** Вычислительная система выполняет два процесса: опрос и обработку информации с датчика и опрос и обработку информации с датчика . Вычислительные процессы и периодические, и их периоды (периоды опроса датчиков) равны и соответственно. Времена обработки информации с датчиков А и В равны соответственно и . Планировщик процессов принимает решения с периодом .

**Задание**

1. Рассчитать требуемое число процессоров для выполнения процессов и в реальном масштабе времени.
2. Составить таблицу профиля выполнения процессов и .
3. Построить и описать временные диаграммы выполнения процессов и для следующих режимов планирования:
   1. с квантованием времени;
   2. с квантованием времени и вытеснением, если приоритет потока выше приоритета потока ;
   3. с квантованием времени и вытеснением, если приоритет потока выше приоритета потока ;
   4. с приоритетом процесса с наиболее ранним предельным сроком завершения задачи.
   5. с частотно-монотонным планированием.
4. Определить возможность выполнения процессов в реальном масштабе времени.
5. Рассмотреть перечень средств обеспечения выполнения процессов в реальном масштабе времени.

**б) Исходные данные.** Вычислительная система выполняет четыре непериодические процесса , , , , для которых в Таблица 1 заданы время поступления, время выполнения и предельные сроки начала работы.

**Задание.**

Построить и описать временные диаграммы выполнения процессов для следующих режимов планирования: наиболее ранний предельный срок, наиболее ранний срок со свободным временем простоя, «первым поступил - первым обслужен».

Таблица 1

Исходные данные

| Процесс | Время поступления | Время выполнения | Предельное время начала работы |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Решение

**Задание 1а**

1. Для расчета необходимого количества процессоров для выполнения процессов и может быть использовано выражение (1):

|  |  |
| --- | --- |
|  | (1) |

В данном случае:

Следовательно, на одном процессоре в реальном масштабе времени отсутствует возможность выполнения заданий.

1. Таблица профиля выполнения процессов и показана в Таблица 2.

Исходные данные

Таблица 2

| Процесс | Время поступления | Время выполнения | Предельное время окончания |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Компьютер (планировщик) принимает решение, связанное с планированием заданий, каждые ms.

1. Временные диаграммы выполнения процессов и для различных режимов планирования (Рисунок 1).



Рисунок 1 – Общая характеристика процессов

* 1. Планирование с квантованием времени (Рисунок 2).



Рисунок 2 – Планирование с квантованием времени

* 1. С квантованием времени и вытеснением, если приоритет потока выше приоритета потока (Рисунок 3).



Рисунок 3 – Планирование фиксированными приоритетами ()

Очевидно, что в соответствии с данным алгоритмом задание выполняется с опозданием. Чтобы его устранить, необходимо увеличить производительность процессора на следующую величину: кв.

Таким образом:

;

.

Измененная временная диаграмма представлена на Рисунок 4.



Рисунок 4 – Планирование фиксированными приоритетами (), результаты повышения производительности процессора

Опоздание устранено, однако возник простой системы. Его величина составляет:

* 1. С квантованием времени и вытеснением, если приоритет потока выше приоритета потока (Рисунок 5).



Рисунок 5 – Планирование фиксированными приоритетами ()

Данный алгоритм приводит к выполнению задания с опозданием. Для его устранения целесообразно увеличить производительность процессора на определенную величину:

;

;

;

.

Следовательно:

;

.

Измененная временная диаграмма представлена на Рисунок 6.



Рисунок 6 – Планирование фиксированными приоритетами (), результаты повышения производительности процессора

Опоздание в итоге устранено, однако уменьшилась загрузка процессора.

* 1. Планирование с приоритетом процесса с наиболее ранним предельным сроком завершения задачи (Рисунок 7).



Рисунок 7 – Планирование с наиболее ранним предельным сроком завершения

* 1. Частотно-монотонное планирование (Рисунок 8).



Рисунок 8 – Частотно-монотонное планирование

Данное планирование абсолютно аналогично планированию, представленному в пункте 3.1. Соответственно, и опоздания убираются аналогичным образом.

1. Необходимо оценить возможность выполнения в реальном времени двух периодических заданий, обладающих следующими параметрами:

задание : ; ;

задание : ; ;

Предполагается, что существует заданий, каждое из которых обладает своим фиксированным временем выполнения и периодом. Известно, что для процессов необходимое условие выполнения процессами предельных сроков задается неравенством (1).

В Таблица 3 представлены некоторые значения верхней границы загруженности процессора для метода для различных значений . С увеличением количества заданий верхняя граница стремится к значению .

Таблица 3

Значения верхней границы загруженности процессора для метода

|  | Значение |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Загруженность процессора каждым из двух заданий составляет . Следовательно, общая загруженность процессора двумя заданиями равна .

Верхняя граница загруженности этих задач в соответствии с методом RMS равна .

Так как общая загруженность процессора обработкой двух представленных задач выше верхней границы по методу (), то можно говорить о невозможности успешного выполнения всех заданий в соответствии с -планированием. Необходимо повышение производительности вычислительной системы, благодаря которому появится возможность выполнения этих заданий в режиме реального времени.

1. Перечень возможных средств обеспечения выполнения процессов в реальном масштабе времени:

* выбор соответствующего планировщика заданий;
* установка нескольких процессоров;
* установка одного более высокопроизводительного процессора.

**Задание 1б**

В соответствии с условием вычислительная система осуществляет выполнение пяти непериодических процессов , для которых в Таблица 1 заданы, соответственно следующие значения: время поступления, время выполнения и предельные сроки начала работы (Рисунок 9).

Далее приводятся временные диаграммы для следующих режимов планирования: наиболее ранний предельный срок (Рисунок 10), наиболее ранний срок со свободным временем простоя (Рисунок 11), ситуация «первым поступил – первым обслужен» (Рисунок 12).



Рисунок 9 – Временная диаграмма требований к выполнению процессов

Простейшей схемой планирования является запуск задания с наиболее ранним предельным сроком и выполнение его дол полного завершения.

Такой подход не позволяет выполнить задание , хотя оно и требует немедленного выполнения. Также отклоняется и задание . В этом и состоит ключевой риск работы с непериодическими заданиями, особенно если они имеют предельное время начала выполнения.



Рисунок 10 – Временная диаграмма выполнения процессов для планирования с наиболее ранним предельным сроком выполнения

Если предельное время выполнения заданий является известным до готовности заданий к выполнению, система планирования может быть усовершенствована путем использования системы, которая называется планированием с наиболее ранним предельным сроком начала со свободным временем простоя.

Принцип ее работы заключается в запуске планировщиком подходящего задания, обладающего наиболее ранним предельным сроком начала. Это задание выполняется до полного завершения. Безусловно, подходящее задание может быть не готовым, в результате чего, несмотря на существование готовых к выполнению заданий, процессор будет простаивать. В данном случае, система воздерживается от выполнения задания , хотя оно является единственным готовым к выполнению. В результате, хотя эффективность использования процессора в таком случае далека от максимальной, все требования к предельным срокам начала выполнения заданий в данном случае удовлетворены.



Рисунок 11 – Временная диаграмма выполнения процессов для планирования с наиболее ранним предельным сроком выполнения со свободным временем простоя



Рисунок 12 – Временная диаграмма выполнения процессов для планирования

«первым поступил – первым обслужен»

Следует отметить, что в схеме планировщика со свободным временем простоя (Рисунок 11) отсутствуют опоздания заданий, однако выполнение заданий осуществляется в порядке отличном от порядка их поступления в вычислительную систему.

# Задание №2

## Условие задания

Для заданной группы вычислительных процессов организовать доступ к критической секции с использованием (по указанию преподавателя): **блокирующей *переменной (Д)****, семафора, мьютекса, монитор и передачи сообщений.*

Объяснить достоинства и недостатки каждого из методов взаимного исключения или организации доступа к разделяемым ресурсам. Привести примеры использования объектов синхронизации в Windows XX.

## Решение

Синхронизация потоков, принадлежащих одному процессу, может быть осуществлена с помощью глобальных блокирующих переменных. С этими переменными, к которым для всех потоков процесса открыт прямой доступ, можно работать без обращения к системным вызовам операционной системы.

Каждой совокупности данных критического характера ставится в соответствие двоичная переменная. Ей потоком присваивается значение («ложь») при вхождении в критическую секцию, и значение («истина») при покидании ее. На Рисунок 13 представлен фрагмент алгоритма потока, который использует для взаимного исключения доступа к критическим данным блокирующую переменную . Перед входом в критическую секцию осуществляется проверка потоком, выполняется ли уже работа какого-либо потока с данными . Если значение переменной , то данные заняты и производится циклический повтор проверки. Если данные являются свободными (), то переменной присваивается значение , после чего выполняется вход потока в критическую секцию. После выполнения потоком всех манипуляций с данными значение переменной снова устанавливается в .



Рисунок 13 – Обработка критических секций с помощью блокирующих переменных

Использование блокирующих переменных оправдано не только при доступе к разделяемым данным, но и в случае необходимости доступа к разделяемым ресурсам любого вида.

Если при описании всех потоков выполнены вышеприведенные соглашения, то можно утверждать о гарантии взаимного исключения. При этом прерывание потоков может быть осуществлено операционной системой в любой момент и в любом месте, в частности, в критической секции.

Вместе с тем, одно ограничение на прерывания присутствует в данном случае. Отсутствует возможность прерывания потока между выполнением операций проверки и установки блокирующей переменной. Целесообразно остановиться на данном моменте подробнее. Например, в результате проверки переменной определено потоком, что ресурс не занят., однако сразу после этого, не успев установить переменную в , произошло прерывание потока. В течение интервала его приостановки другим потоком было выполнено занятие ресурса, вход в критическую секцию. Однако и он был прерван до окончания работы с разделяемым ресурсом. После возврата управления первому потоку он, посчитав ресурс свободным, осуществил установку признака занятости и приступил к выполнению своей критической секции. Следовательно, произошло нарушение принципа взаимного исключения, что в перспективе может повлечь за собой весьма негативные последствия. Чтобы предотвратить подобные ситуации система команд многих компьютеров содержит единую неделимую команду анализа и присвоения значения логической переменной (например, команды , , процессора ). Если такая программа в процессоре отсутствует, то подобные действия приходится осуществлять с помощью специальных системных примитивов, выполняющих запрет прерываний в течение всей операции проверки и установки.

Выполнение взаимного исключения с помощью описанного выше метода обладает значительным недостатком. Он заключается в том, что на протяжении временного промежутка, когда один поток находится в критической секции, другим потоком, в случае необходимости в том же ресурсе, при получении доступа к процессору, будет осуществляться непрерывный опрос блокирующей переменной. Другими словами, будет бесполезно расходоваться процессорное время, выделяемое ему, без возможности его расходования на выполнение другого потока. Чтобы устранить этот недостаток во многих операционных системах предусмотрены специальные системные вызовы для взаимодействия с критическими секциями.

На Рисунок 14 показано применение данных функций для реализации взаимного исключения в операционной системе . Перед началом изменения критических данных поток осуществляет выполнение системного вызова . В процессе данного вызова сначала происходит выполнение, как и в предыдущем случае, проверки блокирующей переменной, характеризующей состояние критического ресурса. Если системным вызовом определено, что ресурс занят (), он, в отличие от предыдущего случая, не перейдет к циклическому опросу, а осуществит перевод потока в состояние ожидания () с одновременной пометкой о том, что активизация данного потока должна произойти при освобождении соответствующего ресурса. Поток, который в данный временной период использует этот ресурс, после выхода из критической секции должен выполнить системную функцию . Результатом этого будет присваивание блокирующей переменной значения, соответствующего свободному состоянию ресурса (). Кроме этого, операционная система осуществляет просмотр очереди ожидающих данный ресурс потоков с последующим переводом первого потока из очереди в состояние готовности.



Рисунок 14 – Обработка критических секций с помощью блокирующих

Следовательно, можно говорить об исключении непроизводительной потери процессорного времени на циклическую проверку освобождения занятого ресурса. Вместе с тем, в тех случаях, когда объем операций в критической секции невелик и вероятность в скором доступе к разделяемому ресурсу высока, применение блокирующих переменных представляется более эффективным решением. Действительно, величина расходов операционной системы в этом случае на реализацию функций входа в критическую ситуацию т выхода из нее может оказаться больше, чем значение полученной экономии.

Для иллюстрации приведенных рассуждений была разработана программа «» рассматривающая практическую задачу «проблема обедающих философов». Пять философов обедают вместе за одним столом. У каждого философа есть свое место за столом. Между каждой тарелкой лежит вилка. Подаваемое блюдо представляет собой разновидность спагетти, которые нужно есть двумя вилками. Каждый философ может только попеременно думать и есть. Более того, философ может есть свои спагетти только тогда, когда у него есть и левая, и правая вилка. Таким образом, две вилки будут доступны только тогда, когда два их ближайших соседа думают, а не едят. После того, как отдельный философ закончит есть, он отложит обе вилки. Проблема заключается в том, как разработать режим (параллельный алгоритм) таким образом, чтобы ни один философ не голодал; т.е. каждый может вечно продолжать чередовать прием пищи и размышление, предполагая, что ни один философ не может знать, когда другие могут захотеть есть или думать (проблема неполной информации). ПО разработана в среде на языке программирования . Программный код приложения представлен в приложении A.

Суть программы – запуск пяти потоков(философов), вызывающих метод , и осуществляющих взаимодействие с критическими переменными и . Предполагается, что данный метод осуществит взятием философом обьекта. Но в ходе работы на практике осуществляется переключение между потоками, в результате чего значения переменных и становится невозможно предсказать. Например, результат может быть таким, как показано на Рисунок 15.

Рисунок 15 – Результаты работы без синхронизации

Решение данной задачи, очевидно, заключается в синхронизации потоков и ограничении доступа к разделяемым ресурсам на время их использования определенным потоком. Для этого применяется блокирующая переменная и конструкция . В операторе определяется часть кода, в пределах которой осуществляется блокировка всего кода, в результате чего весь код становится недоступным для других потоков до завершения выполнения текущего потока. Другие потоки переводятся в очередь ожидания до момента освобождения данного блока кода текущим потоком. Таким образом, оператор дает возможность осуществления синхронизации потоков.

Для блокировки оператором применяется так называемый объект-заглушка (в приводимом примере – это переменные и ). Чаще всего данная блокирующая переменная обладает типом . При перемещении программного кода на оператор происходит блокировка объекта, в результате которой на весь период его блокировки доступом к критической секции обладает только один поток. По завершении работы блока кода происходит освобождение объектов и они становится доступным для других потоков.

Результаты работы программы с синхронизацией потоков с помощью блокирующих переменных представлены на Рисунок 16 (упорядоченный вывод).



Рисунок 16 – Результаты работы с блокирующими переменными

# Задание №3

## Условие задания

**а)** **Разработать программу обнаружения взаимных блокировок процессов в вычислительной системе при наличии одного ресурса каждого типа. Распределение ресурсов в вычислительной системе задается графом распределения ресурсов.**

б)Разработать программу обнаружения взаимных блокировок процессов в вычислительной системе при наличии нескольких ресурсов каждого типа. Распределение ресурсов в вычислительной системе задается векторами существующих и доступных ресурсов.

**в) Разработать программу предотвращения взаимных блокировок процессов в вычислительной системе при наличии одного ресурса каждого типа.**

г)Разработать программу предотвращения взаимных блокировок процессов в вычислительной системе при наличии нескольких ресурсов каждого типа.

Программы, разработанные для задания №3 курсовой работы, должны быть отлажены, и их работоспособность должна быть продемонстрирована преподавателю.

## Решение

Обнаружение блокировок в вычислительной системе при наличии одного экземпляра ресурсов каждого типа производится на основе анализа построенного графа ресурсов и процессов. Наличие циклов в графе указывает на взаимную блокировку в вычислительной системе.

В графе процессы обозначаются как круг с именем процесса, а ресурсы обозначаются как квадрат с именем ресурса. Исходящее ребро от процесса к ресурсу означает, что процесс требует данный ресурс. Входящее ребро от ресурса к процессу означает, что процесс занял данный ресурс.

Один из возможных алгоритмов поиска циклов в графе следующий. Для каждого из узлов в графе выполняется пять шагов:

1. Задаются начальные условия: -пустой список, все ребра немаркированы.
2. Текущий узел добавляем в конец списка и проверяем количество появления узла в списке. Если он встречается два раза, значит цикл и взаимоблокировка.
3. Для заданного узла смотрим, выходит ли из него хотя бы одно немаркированное ребро. Если да, то переходим к шагу 4, если нет, то переходим к шагу 5.
4. Выбираем новое немаркированное исходящее ребро и маркируем его. И переходим по нему к новому узлу и возвращаемся к шагу 3.
5. Зашли в тупик. Удаляем последний узел из списка и возвращаемся к предыдущему узлу. Возвращаемся к шагу 3. Если это первоначальный узел, значит, циклов нет, и алгоритм завершается.

Для реализации работы алгоритма была разработана программа (Рисунок 18, Рисунок 20), протестированная на стандартных примерах.

В данную программу для быстрого вызова внесены примеры из методического указания, а также реализована проверка на корректность введенных данных (Рисунок 17, Рисунок 19).



Рисунок 17 – Граф процессов и ресурсов к примеру на Рисунок 18

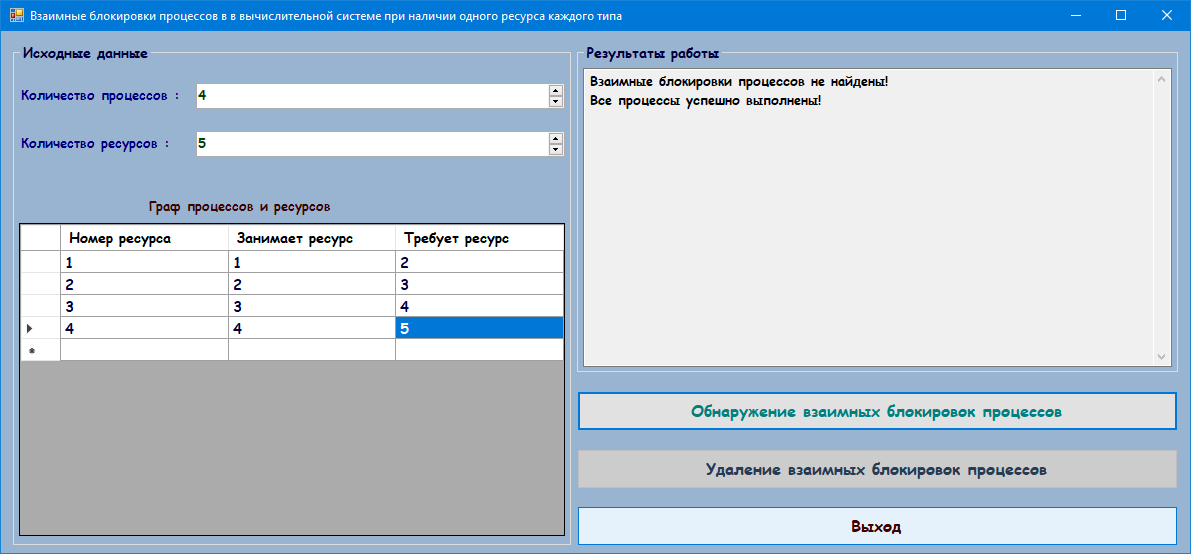


Рисунок 18 – Демонстрация работы тестового примера



Рисунок 19 – Граф процессов и ресурсов к примеру на Рисунок 20

В данном случае обнаружены взаимные блокировки процессов.

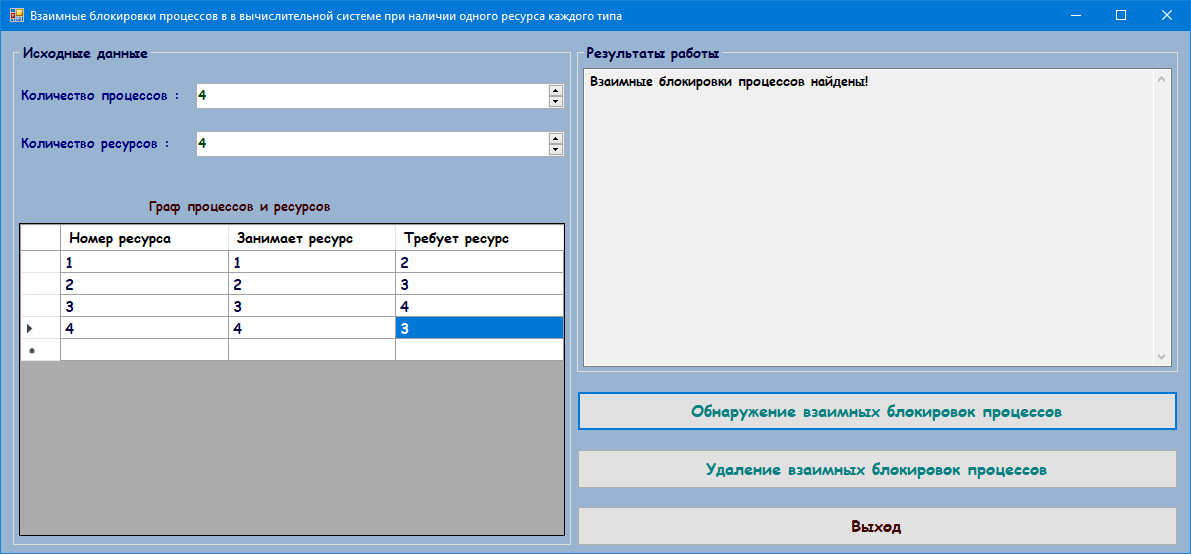


Рисунок 20 – Демонстрация работы тестового примера

Для удаления (выхода из) взаимных блокировок существует несколько методов:

1. Восстановление при помощи принудительной выгрузки ресурса. Как правило, требует ручного вмешательства (например: принтер).
2. Восстановление через откат. Состояние процессов записывается в контрольных точках, и в случае тупика можно сделать откат процесса на более раннее состояние, после чего он продолжит работу снова с этой точки.
3. Восстановление путем уничтожения процесса. Самый простой способ.

В данном случае использован третий метод (Рисунок 21).

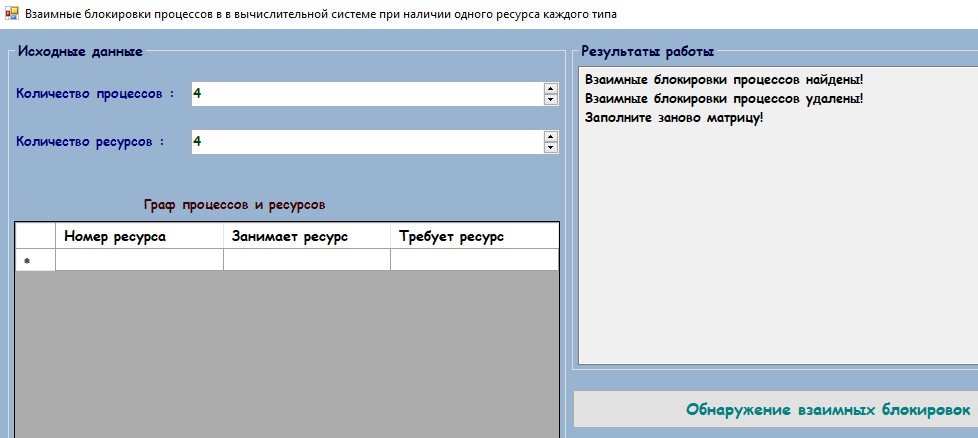


Рисунок 21 – Выход из взаимной блокировки

Полный код разработанного приложения «» приведен в приложении Б.

# Задание №4

## Условие задания

**Исходные данные.** Дана симметричная мультипроцессорная система. Все процессоров системы независимы, однотипны и функционально эквивалентны. Предельная скорость обмена по шине равна , причем каждый процессор при решении задачи требует скорости обмена Р= 60.

**Задание**

1. Разработать структурную и функциональную схемы арбитража со сменой приоритетов для мультипроцессорной системы и описать алгоритм ее работы. Типы арбитража (по указанию преподавателя): приоритетная цепочка, поллинг, **независимые запросы,** децентрализованный.
2. Определить максимальное число процессорных модулей, подключаемых к шине без достижения шиной насыщения.
3. Предложить для заданной схемы методы преодоления эффекта насыщения в шине.

## Решение

1. Арбитраж со сменой приоритетов для мультипроцессорной системы по типу независимых запросов часто называют централизованным параллельным арбитражем.

Действительно, в случае централизованного арбитража система включает в себя специальное устройство, называемое центральным арбитром. Оно осуществляет предоставление доступа к шине только по одному из запросивших ведущих. Данное устройство, которое иногда называют центральным контроллером шины, может представлять собой как самостоятельный модуль, так и часть центрального процессора. Присутствие на шине только одного арбитра свидетельствует о наличии в централизованной схеме только одной точки отказа. В соответствии со схемой подключения ведущих устройств к центральному арбитру выделяются параллельные и последовательные схемы централизованного арбитража.

В случае параллельного варианта для связи центрального арбитра с каждым из потенциальным ведущим используются индивидуальные двухпроводные тракты. Так как запросы к центральному арбитру осуществляются в независимом и параллельном режиме, то данный арбитраж и называют централизованным параллельным или централизованным арбитражем независимых запросов.

Структурная схема данного вида арбитража показана на Рисунок 22. Центральный арбитр (ЦА) осуществляет предоставление доступа к шине только для одного из запросивших ведущих. В данном случае, ЗШ – сигнал запроса шины, ПШ – сигнал предоставления шины, а ШЗ – сигнал занятия шины.

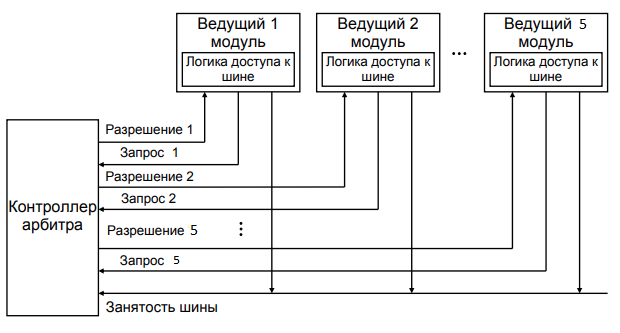


Рисунок 22 – Структурная схема централизованного арбитража независимых запросов

Алгоритм централизованного параллельного арбитража выглядит следующим образом. На вход центрального арбитра осуществляется поступление сигналов ЗШ (запрос шины) по индивидуальным линиям. Ведущему модулю с номером , который был выбран арбитром, происходит возврат сигнала ПШ (предоставление шины). Он передается также по индивидуальной линии. Следует отметить, что занятие шины новым ведущим модулем выполняется не ранее момента снятия предыдущим ведущим, номер которого , сигнала ШЗ (занятие шины).

После получения запроса от ведущего модуля, обладающего более высоким приоритетом, чем текущий ведущий модуль, происходит снятие арбитром сигнала на входе текущего ведущего модуля с одновременной выдачей сигнала предоставления шины ПШ запросившему ведущему модулю. Соответственно, текущий ведущий модуль в ответ на снятие арбитром сигнала осуществляет снятие собственных сигналов ШЗ и . Далее осуществляется переход управления шиной к запросившему ведущему модулю. Следует отметить, что при осуществлении передачи информации в момент пропадания сигнала ПШ на шине текущим ведущим модулем сначала будет завершена передача, после чего выполняется снятие собственных сигналов.

Возможная реализация описанного арбитража представлена на Рисунок 23.

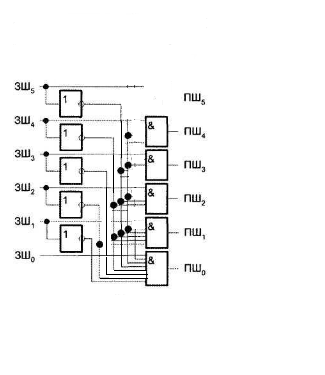


Рисунок 23 – Функциональная схема централизованного арбитража независимых запросов глобальный арбитр

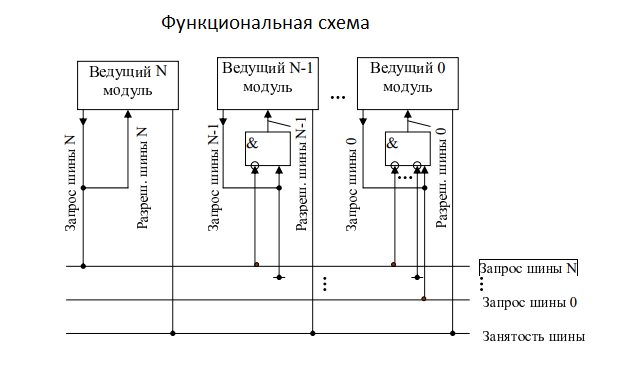


Рисунок 24 – Функциональная схема централизованного арбитража независимых запросов локальные арбитры

Достоинства такого варианта реализации арбитража очевидны. Схема централизованного параллельного арбитража обладает большой гибкостью. Например, статические приоритеты могут заменяться любыми вариантами смены приоритетов в динамическом режиме. Кроме этого наличие прямых связей между центральным арбитром и ведущими модулями существенно повышает быстродействие. Вместе с тем, именно эти непосредственные связи повышают стоимость реализации. Недостатком также является наличие сложностей при подключении дополнительных устройств. Чаще всего количество ведущих модулей при арбитраже подобного типа не превышает восьми. Данной схеме присущ еще один значительный недостаток. Наличие сигналов запроса и подтверждения исключительно только на индивидуальных линиях и их отсутствие на общих линиях шины обуславливает существенные сложности при проведении диагностики.

1. Максимальное количество процессорных модулей, которые можно подключить к шине без достижения эффекта насыщения, можно получить в соответствии с выражением (2) :

|  |  |
| --- | --- |
|  | (2) |

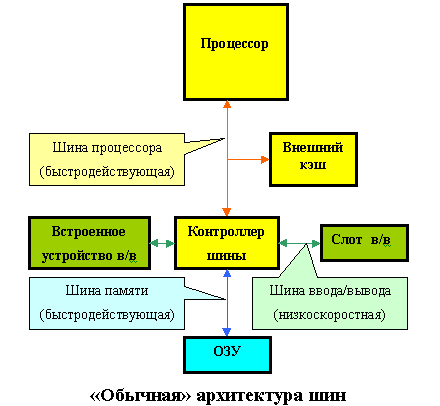
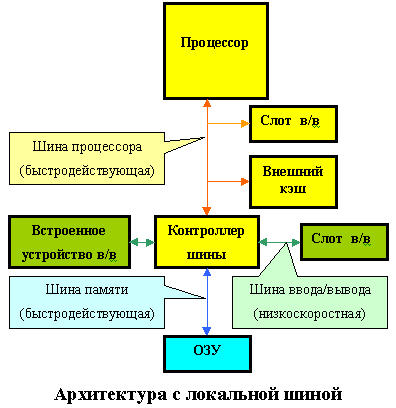
Следовательно, .

Таким образом, наибольшее количество процессорных модулей, которые можно подключить к шине без достижения эффекта насыщения, составляет модуля.

1. Эффект насыщения в шине для заданной схемы может быть предотвращен путем выбора более высокопроизводительной шины. Например, может быть использована быстрая шина Мб/с

Тогда:

Очевидным выходом из создавшегося положения является следующий: осуществлять часть операций обмена данными, требующих высоких скоростей, не через шину ввода/вывода, а через шину процессора, примерно так же, как подключается внешний кэш. Такая конструкция получила название локальной шины (Local Bus). Рисунки наглядно демонстрируют различие между обычной архитектурой и архитектурой с локальной шиной.

Локальная шина не заменяла собой прежние стандарты, а дополняла их. Основными шинами в компьютере по-прежнему оставались ISA или EISA, но к ним добавлялись один или несколько слотов локальной шины. Первоначально эти слоты использовались почти исключительно для установки видеоадаптеров, при этом к 1992 году было разработано несколько несовместимых между собой вариантов локальных шин, исключительные права на которые принадлежали фирмам-изготовителям. VESA (Video Electronic Standard Association) — ассоциация, представляющая более 100 компаний — предложила в августе 1992 года свою спецификацию локальной шины.

# ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы были изучены основные принципы работы операционных систем, организован доступ к критической секции с использованием передачи сообщений, разработаны программы обнаружения взаимных блокировок процессов в вычислительной системе при наличии одного и нескольких ресурсов каждого типа, а также изучены основные объекты синхронизации процессов.

# СПИСОК ЛИТЕРАТУРЫ

1. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. - СПб.: Питер, 2001. - 544 с.
2. Иртегов Д. В. Введение в операционные системы. - СПб.: БХВ-Петербург, 2002. - 624 с.
3. Таненбаум Э. Современные операционные системы. - СПб.: Питер, 2002. - 1040 с.
4. Таненбаум Э., Вудхал А. Операционные системы: Разработка и реализация. Классика CS. - СПб.: Питер, 2006. - 576 с.
5. Столлингс В. Операционные системы: Внутреннее устройство и принципы проектирования. - М.: Изд. дом «Вильямс», 2002. - 848 с.
6. Операционные системы. Параллельные и распределенные системы / Д. Бэкон, Г. Харрис. - СПб.: Питер, 2004. - 800 с.
7. Цилькер Б.Я., Орлов С.А. Организация ЭВМ и систем. - СПб.: Питер, 2004. - 668 с.
8. Засов В.А. Операционные системы. - Самара, 2006 - 44 с.

# ПРИЛОЖЕНИЕ А

Программный код приложения «»

Код модуля «Philosopher.cs»

using System.Collections;

using System.Collections.Generic;

using System.Threading;

using UnityEngine;

using TMPro;

public class Philosopher : MonoBehaviour

{

    public UIController uicontroller;

    public int averageDuration;

    public int PickDuration;

    public int DropDuration;

    public string philosopher\_Name;

    public int grabResourcesSpeed = 30;

    public Transform leftTarget;

    public Transform rightTarget;

    public Transform oldleftTarget;

    public Transform oldrightTarget;

    public bool useLockSync;

    public Resource leftResource;

    public Resource RightResource;

    private int random\_thinking\_time;

    private int random\_eating\_duration;

    private int random\_drop\_resource\_duration;

    private int random\_pick\_resource\_duration;

    private Renderer current;

    public CubeAimation CubeAimation;

    public GameManager gameManager;

    private MeshRenderer meshRenderer;

    public bool handleLoop;

    public TextMeshProUGUI textMesh;

    public string actionlog = "idle";

    public Floater floating;

    public Material idleColor, thinkingColor, eatingColor,holdingColor;

    public Quaternion defaultRotation;

    /\*Philosophers(object leftResource, object RightResource)

    {

        this.leftResource = leftResource;

        this.RightResource = RightResource;

    }\*/

    private void Start()

    {

        uicontroller = GameObject.Find("UIDocument").GetComponent<UIController>();

        //Load Material Resource

        // idleColor =  Resources.Load<Material>("59 EMISSION-ORANGE");

        // eatingColor = Resources.Load<Material>("60 EMMISION-RED");

        // holdingColor = Resources.Load<Material>("61 EMISSION-ORANGE");

        // thinkingColor = Resources.Load<Material>("63 EMISSION-GREEN");

        // defaultRotation = transform.rotation;

        meshRenderer = GetComponent<MeshRenderer>();

        floating = GetComponent<Floater>();

        current = gameObject.GetComponent<Renderer>();

        CubeAimation = GetComponent<CubeAimation>();

        gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();

        floating.frequency = Random.Range(0.25f, 0.5f);

    }

    private void DoAction(string action)

    {

        actionlog = action;

        Thread.Sleep(random\_thinking\_time);

    }

    private void EatingAction()

    {

        actionlog = "Drinking" ;

        Thread.Sleep(random\_eating\_duration);

    }

    private void DropResouce(string resouceName)

    {

        actionlog = resouceName;

        // Debug.Log(actionlog);

        // Thread.Sleep(100);

        Thread.Sleep(random\_drop\_resource\_duration);

    }

    private void PickUpResouce(string resouceName,Resource resource)

    {

        actionlog = resouceName;

        // Debug.Log(resource.Name);

        Thread.Sleep(random\_pick\_resource\_duration);

    }

    private void restartAction()

    {

        actionlog = "Restart Action";

        Thread.Sleep(100);

    }

    private void Awake()

    {

        Debug.Log("Sleep time " + random\_thinking\_time);

    }

    public void startAction()

    {

        while (handleLoop)

        {

            restartAction();

            DoAction("Thinking");

            if(useLockSync)

            {

                lock (leftResource)

                {

                    PickUpResouce("Pick left Glass",leftResource);

                    lock (RightResource)

                    {

                        PickUpResouce("Pick right Glass",RightResource);

                        EatingAction();

                        DropResouce("Drop right Glass");

                    }

                    DropResouce("Drop left Glass");

                }

            }

            else

            {

                PickUpResouce("Pick left Glass",leftResource);

                PickUpResouce("Pick right Glass",RightResource);

                EatingAction();

                DropResouce("Drop left Glass");

                DropResouce("Drop right Glass");

            }

        }

    }

    void Update()

    {

        averageDuration = (int)uicontroller.ThinkingSlider.value;

        useLockSync = uicontroller.useSynhToggle.value;

        if (handleLoop & !uicontroller.DeadLockToggle.value)

        {

            random\_thinking\_time = Random.Range(0, averageDuration);

            random\_eating\_duration = Random.Range(0, averageDuration);

            random\_pick\_resource\_duration = PickDuration;

            random\_drop\_resource\_duration = DropDuration;

        }

        else

        {

            random\_thinking\_time = (int)uicontroller.ThinkingSlider.value;

            random\_eating\_duration = (int)uicontroller.EatingSlider.value;

            random\_pick\_resource\_duration = (int)uicontroller.PickUpSlider.value;

            random\_drop\_resource\_duration = (int)uicontroller.DropSlider.value;

        }

        textMesh.text = actionlog;

        if (actionlog == "Restart Action")

        {

            IdleState();

        }

        if  (actionlog == "Thinking")

        {

            ThinkingState();

        }

        else if (actionlog == "Pick left Glass" || actionlog == "Pick right Glass" )

        {

            MoveTowardsTO(leftResource,leftTarget);

            PickState();

            // GrabResources(leftResource);

        }

        else if (actionlog == "Drinking")

        {

            MoveTowardsTO(leftResource,leftTarget);

            MoveTowardsTO(RightResource,rightTarget);

            EatingState();

        }

        else if (actionlog == "Drop left Glass" || actionlog == "Drop right Glass")

        {

            MoveTowardsTO(leftResource,oldleftTarget);

            MoveTowardsTO(RightResource, oldrightTarget);

            IdleState();

        }

        else

        {

            IdleState();

        }

        // if (actionlog == "Pick right Glass")

        // {

        //     GrabResources(RightResource);

        // }

        // if (actionlog == "Pick left Glass")

        // {

        //     GrabResources(leftResource);

        // }

    }

    void MoveTowardsTO(Resource resource,Transform target)

    {

        resource.gameObject.transform.position =

            Vector3.MoveTowards(resource.gameObject.transform.position, target.position, grabResourcesSpeed \* Time.deltaTime);

        // resource.transform.parent = gameObject.transform;

        // resource.gameObject.transform.position = new Vector3(0.7f, 0, 0.7f);

    }

    // void DropResouce(Resource resource,Transform target)

    // {

    //     resource.gameObject.transform.position =

    //         Vector3.MoveTowards(resource.gameObject.transform.position, target.position, 20 \* Time.deltaTime);

    // }

    void EatingState()

    {

        meshRenderer.material = eatingColor;

        floating.degreesPerSecond = Random.Range(30, 60);

    }

    void PickState()

    {

        meshRenderer.material = holdingColor;

    }

    void ThinkingState()

    {

        meshRenderer.material = thinkingColor;

        floating.degreesPerSecond = 0;

        transform.eulerAngles = new Vector3(0f, 0f, 0f);

        // DropResouce(leftResource);

        // DropResouce(RightResource);

    }

    void IdleState()

    {

        meshRenderer.material = idleColor;

        floating.degreesPerSecond = 0;

        transform.eulerAngles = new Vector3(0f, 0f, 0f);

        MoveTowardsTO(leftResource,oldleftTarget);

        MoveTowardsTO(RightResource, oldrightTarget);

        // DropResouce(leftResource);

        // DropResouce(RightResource);

    }

}

Код модуля «Resource.cs»

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Resource : MonoBehaviour

{

    public string Name;

    public Vector3 oldPosition;

    // Start is called before the first frame update

    void Start()

    {

        oldPosition = transform.position;

    }

    // Update is called once per frame

    void Update()

    {

    }

}

# ПРИЛОЖЕНИЕ Б

Программный код приложения «»

Код модуля «MyOrientedGraph.cs»

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DeadLocks\_App

{

//===Класс, описывающий ориентированный граф

class MyOrientedGraph

{

private readonly int Vertex; //количество вершин

private readonly List<List<int>> adjacency\_list; //список смежности

//===Конструктор класса

public MyOrientedGraph(int Vertex)

{

this.Vertex = Vertex;

adjacency\_list = new List<List<int>>(Vertex);

for (int i = 0; i < Vertex; i++)

adjacency\_list.Add(new List<int>());

}

//===Метод проверки существования циклов в графе

public bool IsCyclExist(int i, bool[] visited,

bool[] recStack)

{

// пометка текущего узла как посещенного

// часть рекурсии

if (recStack[i])

return true;

if (visited[i])

return false;

visited[i] = true;

recStack[i] = true;

List<int> children = adjacency\_list[i];

foreach (int c in children) if (

IsCyclExist(c, visited, recStack)) return true;

recStack[i] = false;

return false;

}

//===Метод добавления очередного ребра в графе

public void addEdge(int sou, int dest)

{

adjacency\_list[sou].Add(dest);

}

//===Результирующий метод определения циклов в графе

// Возвращает true если в графе есть циклы

// или false в обратном случае

public bool isCyclicResult()

{

// пометка всех вершин, как непосещенных

// не является частью рекурсивного вызова

bool[] visited = new bool[Vertex];

bool[] recStack = new bool[Vertex];

// Call the recursive helper function to

// detect cycle in different DFS trees

for (int i = 0; i < Vertex; i++)

if (IsCyclExist(i, visited, recStack))

return true;

return false;

}

}

}

Код модуля «Form\_Main.cs»

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace DeadLocks\_App

{

    public partial class Form\_Main : Form

    {

        public Form\_Main()

        {

            InitializeComponent();

        }

        //===Константы программы

        const int PR\_Max = 100;                                          //максимальная размерность матрицы графа

        //===Переменные программы

        public static int Processes\_Count;                               //кол-во процессов в ВС

        public static int Resources\_Count;                               //кол-во ресурсов в ВС

        public static int PR;                                            //реальная размерность матрицы смежности графа

        public static int[,] Adjacency\_Matrix = new int[PR\_Max, PR\_Max]; //матрица смежности графа

        //===Обработка события загрузки формы

        private void Form\_Main\_Load(object sender, EventArgs e)

        {

            Processes\_Count = Convert.ToInt32(Processes\_NumericUpDown.Value); //кол-во процессов в ВС

            Resources\_Count = Convert.ToInt32(Resources\_NumericUpDown.Value); //кол-во ресурсов в ВС

            PR = Processes\_Count + Resources\_Count;                      //размерность матрицы смежности

            Func\_MatrZeros();                                            //обнуление элементов матрицы

            Graph\_DataGridView.Rows.Clear();                             //очистка графа

            Graph\_DataGridView.RowCount = Processes\_Count;               //кол-во строк в графе

            for (int i = 0; i < Graph\_DataGridView.RowCount; i++)

            {

                Graph\_DataGridView.Rows[i].Cells[0].Value = i + 1;

            }

        }

        //===Функция обнуления элементов матрицы смежности

        public void Func\_MatrZeros()

        {

            for (int i = 0; i < PR; i++)

            {

                for (int j = 0; j < PR; j++)

                {

                    Adjacency\_Matrix[i, j] = 0;

                }

            }

        }

        //===Обработка события изменения количества процессов

        private void Processes\_NumericUpDown\_ValueChanged(object sender, EventArgs e)

        {

            Processes\_Count = Convert.ToInt32(Processes\_NumericUpDown.Value); //кол-во процессов в ВС

            Resources\_Count = Convert.ToInt32(Resources\_NumericUpDown.Value); //кол-во ресурсов в ВС

            PR = Processes\_Count + Resources\_Count;                      //размерность матрицы смежности

            Graph\_DataGridView.Rows.Clear();                             //очистка графа

            Graph\_DataGridView.RowCount = Processes\_Count;               //кол-во строк в графе

            for (int i = 0; i < Graph\_DataGridView.RowCount; i++)

            {

                Graph\_DataGridView.Rows[i].Cells[0].Value = i + 1;

            }

        }

        //===Обработка события изменения количества ресурсов

        private void Resources\_NumericUpDown\_ValueChanged(object sender, EventArgs e)

        {

            Processes\_Count = Convert.ToInt32(Processes\_NumericUpDown.Value); //кол-во процессов в ВС

            Resources\_Count = Convert.ToInt32(Resources\_NumericUpDown.Value); //кол-во ресурсов в ВС

            PR = Processes\_Count + Resources\_Count;                      //размерность матрицы смежности

        }

        //===Обработка нажатия кнопки "Обнаружение взаимных блокировок процессов"

        private void button1\_Click(object sender, EventArgs e)

        {

            Results\_TextBox.Clear();                                     //очистка окна результатов

            Func\_MatrZeros();                                            //обнуление элементов матрицы

            MyOrientedGraph my\_graph = new MyOrientedGraph(PR);          //добавление графа

            //Формирование графа (матрицы смежности)

            for (int i = 0; i < Graph\_DataGridView.RowCount - 1; i++)

            {

                if (Graph\_DataGridView.Rows[i].Cells[1].Value != "")  //заполнение занимаемых процессами ресурсов

                {

                    int temp\_From = Processes\_Count + Convert.ToInt32(Graph\_DataGridView.Rows[i].Cells[1].Value);

                    int temp\_To = Convert.ToInt32(Graph\_DataGridView.Rows[i].Cells[0].Value);

                    //Adjacency\_Matrix[temp\_From, temp\_To] = 1;

                    my\_graph.addEdge(temp\_From - 1, temp\_To - 1);

                }

                if (Graph\_DataGridView.Rows[i].Cells[2].Value != "")  //заполнение требуемых процессами ресурсов

                {

                    int temp\_To = Processes\_Count + Convert.ToInt32(Graph\_DataGridView.Rows[i].Cells[2].Value);

                    int temp\_From = Convert.ToInt32(Graph\_DataGridView.Rows[i].Cells[0].Value);

                    //Adjacency\_Matrix[temp\_From + 1, temp\_To + 2] = 1;

                    my\_graph.addEdge(temp\_From - 1, temp\_To - 1);

                }

            }

            // вызов метода определения циклов

            if (my\_graph.isCyclicResult())

            {

                Results\_TextBox.Text += "Взаимные блокировки процессов найдены!" + Environment.NewLine;

                Del\_Button.Enabled = true;

            }

            else

            {

                Results\_TextBox.Text += "Взаимные блокировки процессов не найдены!" + Environment.NewLine;

                Results\_TextBox.Text += "Все процессы успешно выполнены!" + Environment.NewLine;

                Del\_Button.Enabled = false;

            }

        }

        //===Функция поиска всех циклов в глубину

        IEnumerable<Stack<int>> Func\_FindAllCycles(int[,] edg\_arr, int curr\_v, HashSet<int> vis\_alr, Stack<int> curr\_p)

        {

            if (vis\_alr.Contains(curr\_v))

            {

                var res = new Stack<int>();

                res.Push(curr\_v);

                foreach (var vert in curr\_p)

                {

                    res.Push(vert);

                    // обнаружение пути только до начала цикла

                    if (vert == curr\_v) break;

                }

                yield return res;

            }

            else

            {

                vis\_alr.Add(curr\_v);

                curr\_p.Push(curr\_v);

                for (int i = 0; i < edg\_arr.GetLength(1); i++)

                    if (curr\_v != i && edg\_arr[curr\_v, i] == 1)

                        foreach (var cycle in Func\_FindAllCycles(edg\_arr, i, vis\_alr, curr\_p)) yield return cycle;

                vis\_alr.Remove(curr\_v);

                curr\_p.Pop();

            }

        }

        //===Обработка события нажатия кнопки "Удалить взаимные блокировки"

        private void Del\_Button\_Click(object sender, EventArgs e)

        {

            //TODO:Delete only row there get deadlocks

            Graph\_DataGridView.Rows.Clear();

            Results\_TextBox.Text += "Взаимные блокировки процессов удалены!" + Environment.NewLine;

            Results\_TextBox.Text += "Заполните заново матрицу!" + Environment.NewLine;

            Del\_Button.Enabled = false;

        }

        //===Обработка события нажатия кнопки "Выход"

        private void Exit\_Button\_Click(object sender, EventArgs e)

        {

            Application.Exit();

        }

    }

}