МИНИСТЕРСТВО ТРАНСПОРТА РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ЖЕЛЕЗНОДОРОЖНОГО ТРАНСПОРТА
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ПУТЕЙ СООБЩЕНИЯ

ЭЛЕКТРОТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра «Цифровых технологий»

Курсовая работа по дисциплине «Архитектура параллельных вычислительных систем»

Выполнил:
студент 1 курса, группа ИВТм-21
Залесов Никита Алексеевич
Дата сдачи «__»_____2023 г.
Оценка_____

Руководитель:
к.т.н., доцент
Засов Валерий Анатольевич

Самара, 2023

Содержание

ВВЕДЕНИЕ

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия — Inter Process Communications (IPC), что отражает историческую первичность понятия «процесс» по отношению к понятию «поток». Обычно к средствам IPC относят не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

Любое взаимодействие процессов или потоков связано с их синхронизацией, которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными. Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он должен быть приостановлен.

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

Переход от одного процесса (потока) к другому осуществляется в результате планирования и диспетчеризации. Работа по определению того, в

какой момент необходимо прервать выполнение текущего активного потока и какому потоку предоставить возможность выполняться, называется планированием. Планирование реализуется компонентой ОС, называемой планировщиком (scheduler). Планирование процессов, по существу, включает в себя решение двух задач:

1) определение момента времени для смены текущего активного потока;

2) выбор для выполнения потока из очереди готовых потоков.

Существует множество различных алгоритмов планирования потоков, по-своему решающих каждую из приведенных выше задач.

В большинстве операционных систем универсального назначения планирование осуществляется динамически (on-line), то есть решения принимаются во время работы системы на основе анализа текущей ситуации. ОС работает в условиях неопределенности – потоки и процессы появляются в случайные моменты времени и так же непредсказуемо завершаются. Динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации и не используют никаких предположений о мультипрограммной смеси.

Другой тип планирования – статический (off-line) – может быть использован в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее, например, в системах реального времени.

Диспетчеризация заключается в реализации найденного в результате планирования (динамического или статического) решения, то есть в переключении процессора с одного потока на другой. Процесс диспетчеризации осуществляется диспетчером (dispatcher) ОС.

Диспетчеризация сводится к следующему:

1) сохранению контекста текущего потока, который требуется сменить;

2) загрузке контекста нового потока, выбранного в результате планирования;

3) запуску нового потока на выполнение.

С самых общих позиций все множество алгоритмов планирования можно разделить на два класса: вытесняющие и не вытесняющие алгоритмы планирования.

Не вытесняющие (non-preemptive) алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению поток.

Вытесняющие (preemptive) алгоритмы – это такие способы планирования потоков, в которых решение о переключении процессора с выполнения одного потока на выполнение другого потока принимается ОС, а не активной задачей.

В большинстве современных операционных систем (ОС), например Windows или Linux, используются вытесняющие алгоритмы планирования.

Далее среди множества алгоритмов планирования выделяются два больших класса – бесприоритетные и приоритетные алгоритмы.

При бесприоритетном планировании выбор процессов или потоков производится в соответствии с некоторым заранее установленным порядком без учета их относительной важности и времени обслуживания.

При реализации приоритетных дисциплин обслуживания отдельным процессам и потокам предоставляется преимущественное право перейти в состояние выполнения в соответствии с установленными для них приоритетами. Приоритеты могут быть фиксированными (постоянными) и динамическими (изменяемыми в ходе вычислительного процесса), что, конечно, требует дополнительных ресурсов ВС на вычисление приоритетов и усложняет ОС.

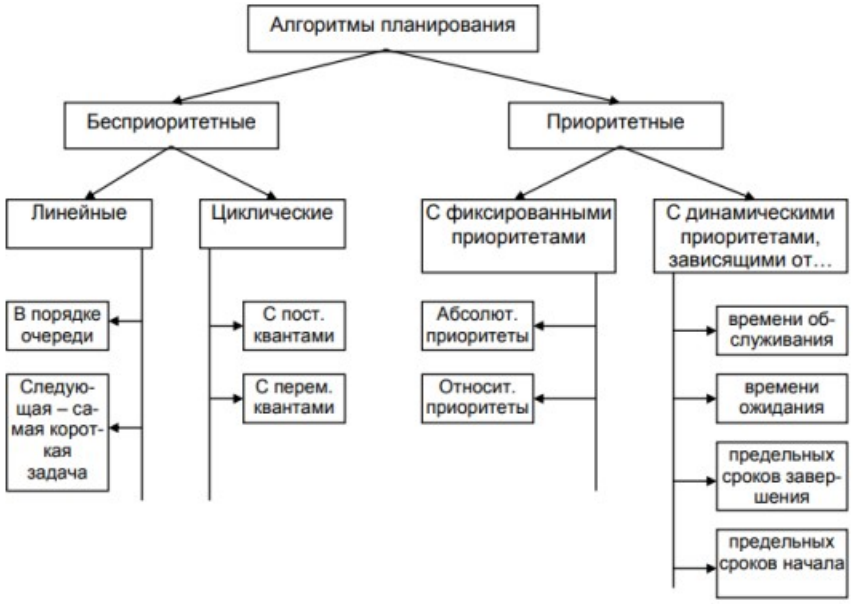

Рисунок 1 – Классификации алгоритмов планирования

Рассмотрим кратко некоторые наиболее распространенные алгоритмы планирования, временные диаграммы работы которых необходимо построить в первом задании курсовой работы.

Линейные алгоритмы планирования реализуют выполнение процессов и потоков в порядке очереди, которая организуется согласно тем или иным соглашениям.

Самой простой в реализации является дисциплина обслуживания, при которой процессы и потоки выполняются в порядке их появления в ВС. Этот алгоритм называется «первым пришел – первым обслужен», или FCFS (First come – First Served). Те процессы и потоки, которые были заблокированы в процессе выполнения, после перехода в состояние готовности вновь ставятся в очередь готовности. При этом возможны два варианта.

В первом наиболее простом варианте разблокированный процесс или поток ставится в конец очереди готовых процессов и потоков.

Во втором варианте планировщик помещает разблокированный процесс или поток перед теми процессами или потоками, которые еще не выполнялись. Таким образом, образуются две очереди: одна очередь образуется из новых процессов или потоков, а вторая очередь - из ранее выполнявшихся, но попавших в состояние ожидания. Такой подход позволяет реализовать стратегию обслуживания «заканчивать выполнение процессов

или потоков в порядке их появления». Рассмотренный алгоритм относится к невытесняющим.

К основным преимуществам этого алгоритма можно отнести его простую реализацию, справедливость и малые затраты системных ресурсов на формирование очереди задач

Существенным недостатком алгоритма является рост среднего времени ожидания обслуживания при увеличении загрузки ВС, причем короткие процессы, требующие небольших затрат машинного времени, вынуждены ожидать столько же, сколько и трудоемкие.

Избежать указанного недостатка позволяет невытесняющий алгоритм, при котором планировщик выбирает первым для выполнения самый короткий процесс (задачу). Этот алгоритм называется «кратчайший процесс (задача) – первый», или SJN (Shortest Job Next). Алгоритм позволяет уменьшить оборотное время – среднее время от момента поступления процесса (задачи) на выполнение до завершения выполнения.

В вытесняющем алгоритме планирования, основанном на квантовании, каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени – квант (time slice). Этот алгоритм получил название циклического (карусельного), или RR (Round Robin).

Смена активного потока происходит, если:

− поток завершился и покинул систему;

− произошла ошибка;

− поток перешел в состояние ожидания;

− исчерпан квант процессорного времени, отведенный данному потоку.

В основе вытесняющих алгоритмов планирования, использующих приоритеты процессов и потоков, лежит концепция приоритетного обслуживания. Приоритетное обслуживание предполагает наличие у процессов (потоков) некоторой изначально известной характеристики – приоритета, на основании которой определяется порядок их выполнения.

Приоритет – это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в частности процессорного времени: чем выше приоритет, тем выше привилегии, тем меньше времени будет проводить поток в очередях.

Смешанные алгоритмы планирования построены с использованием как концепции квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора потока из очереди готовых определяется приоритетами потоков.

В таких случаях удобно сгруппировать процессы в классы по приоритетам и использовать приоритетное планирование среди классов, но циклическое планирование внутри каждого класса.

Алгоритмы планирования с предельными сроками основаны на дополнительной информации о процессах, которая может включать следующее.

Время готовности – время, когда задание становится доступным для выполнения.

В повторяющемся или периодическом задании время готовности представляет собой последовательность заранее известных времен.

В непериодическом задании это время может быть известно заранее, но может быть и так, что ОС узнает его только в тот момент, когда задание станет действительно готовым к выполнению.

Предельное время начала выполнения – время, когда должно начаться выполнение задания.

Одним из эффективных методов планирования для периодических задач является частотно-монотонное планирование (rate monotonic scheduling — RMS).

Система RMS назначает приоритеты заданиям на основе их периодов.

В частотно-монотонном планировании заданием с наивысшим приоритетом является задание с наименьшим периодом; вторым по приоритетности является задание со вторым по величине периодом и т.д.

Соответственно, в случае готовности для выполнения нескольких заданий первым обслуживается задание с наименьшим периодом.

1.Задание на курсовую работу

Задание на курсовую работу по дисциплине
«Архитектура параллельных вычислительных систем», в № 7

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

Задание №1

а)Исходные данные. Вычислительная система выполняет два процесса: опрос и обработку информации с датчика А и опрос и обработку информации с датчика В. Вычислительные процессы А и В периодические, и их периоды (периоды опроса датчиков) равны T_A=140 и T_B= 350 соответственно. Времена обработки информации с датчиков А и В равны соответственно C_A=70 и C_B=175 Планировщик процессов принимает решения с периодом П= 70 .

Задание.

1. Рассчитать требуемое число процессоров для выполнения процессов А и В в реальном масштабе времени.
2. Составить таблицу профиля выполнения процессов А и В.
3. Построить и описать временные диаграммы выполнения процессов А и В для следующих режимов планирования:
3.0. с квантованием времени;
3.1. с квантованием времени и вытеснением, если приоритет потока А выше приоритета потока В;
3.2. с квантованием времени и вытеснением, если приоритет потока В выше приоритета потока А;
3.3.с приоритетом процесса с наиболее ранним предельным сроком завершения задачи.
3.4.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 А, В, С, Д, Е для которых в таблице 1.1 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 1.1

Процесс	Время поступления	Время выполнения	Предельное время начала работы
А	80	160	880
В	160	160	160
С	320	160	400
Д	400	160	720
Е	480	160	560

Задание №2

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.

Задание №3

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.

Задание №4

Исходные данные. Дана симметричная мультипроцессорная система. Все N=3 процессоров системы независимы, однотипны и функционально эквивалентны. Предельная скорость обмена по шине равна V= 160 , причем каждый процессор при решении задачи требует скорости обмена P= 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.

Задание №1

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 A и B равны соответственно и . Планировщик процессов принимает решения с периодом .

Задание.

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

Решение

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

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\ldots+\frac{C_n}{T_n}\le n\cdot\left(2^{\frac{1}{n}}-1\right) \quad (1)$$

В данном случае:

$$\frac{70}{140}+\frac{175}{350}=1>2\cdot\left(2^{\frac{1}{2}}-1\right)=0{,}828$$

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

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

Таблица 1

Процесс	Время поступления	Время выполнения	Предельное время окончания
$A(1)$	0	70	140
$A(2)$	140	70	280
$A(3)$	280	70	420
$A(4)$	420	70	560
$A(5)$	560	70	700
$B(1)$	0	175	350
$B(2)$	350	175	700

Компьютер может принимать решение о планировании каждые 70 ms.

3. Временные диаграммы выполнения процессов A и B для различных режимов планирования (Рисунок 1: Временная диаграмма

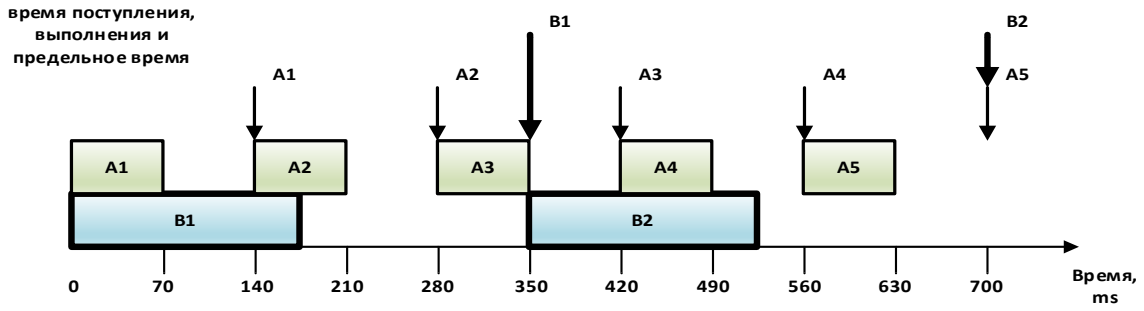

Рисунок 1: Временная диаграмма выполнения процессов A и B

выполнения процессов A и B).

3.0 Планирование с квантованием времени (Рисунок 2: Планирование процессов без применения алгоритма RR).

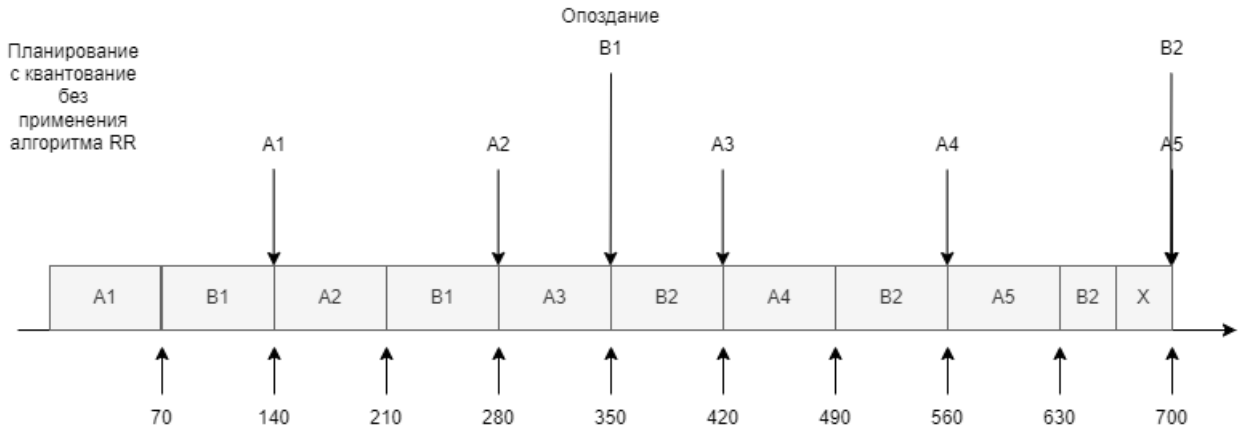

Рисунок 2: Планирование процессов без применения алгоритма RR

Очевидно, что в соответствии с данным алгоритмом задание B выполняется с опозданием. Чтобы его устранить, необходимо увеличить производительность процессора на следующую величину: $70/2=35$ кв.

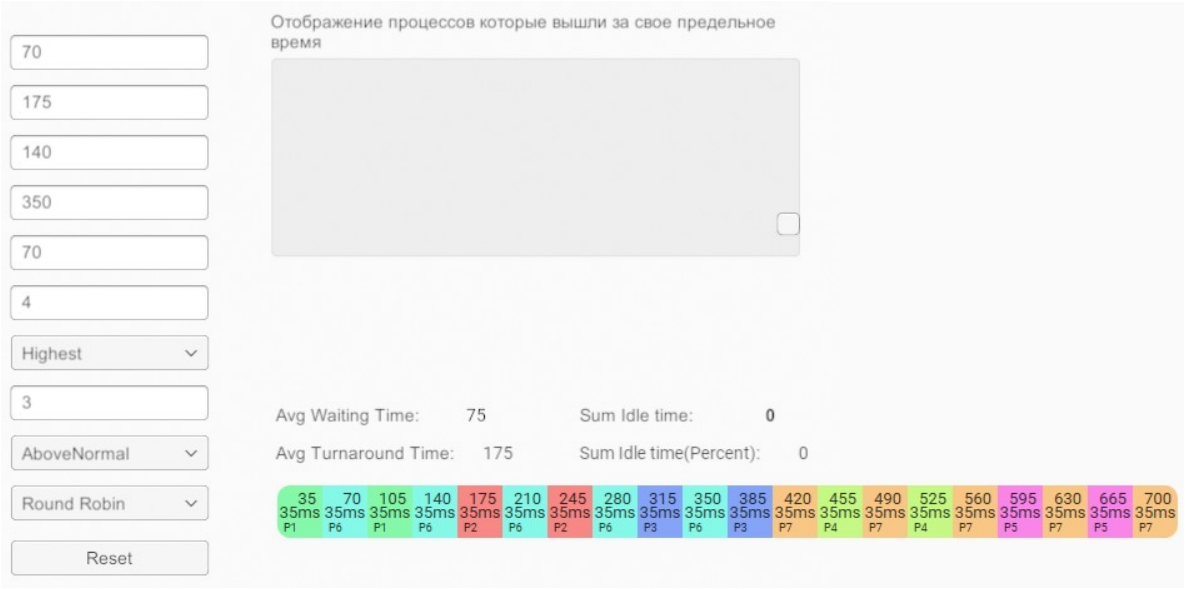

Рисунок 3: Планирование с квантованием с применением алгоритма RR

3.1 С квантованием времени и вытеснением, если приоритет потока A выше приоритета потока B (Рисунок 4).

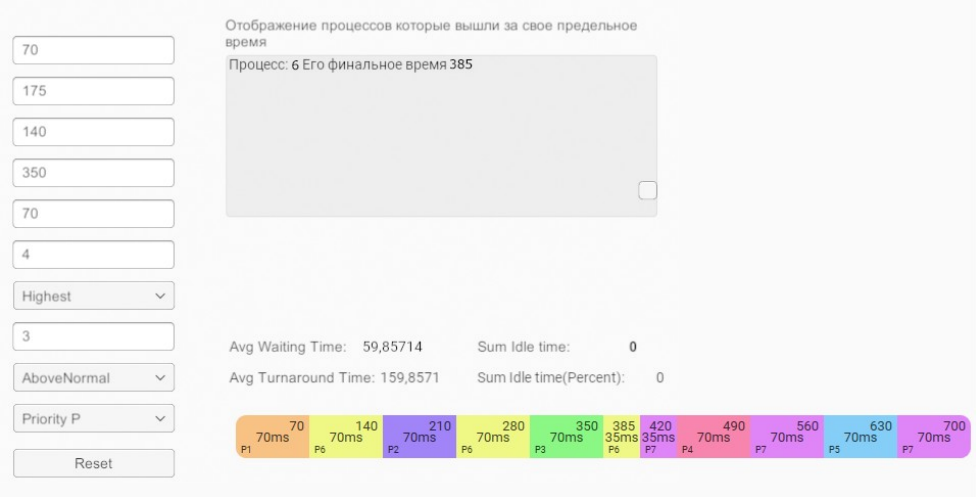

Рисунок 4: Планирование фиксированными приоритетами ($A>B$)

При использовании данного алгоритма планирования процесс B выполняется с опозданием. B1 обрабатывается не полностью. Для устранения опоздания уменьшим время выполнения Cb. Получим тогда следующую производительность:

$$P=\frac{175-140}{175}\cdot 100\%=20\%$$

Производительность процессора увеличилась на 20%.

$$C_A^{'}=0{,}8\cdot C_A=0{,}8\cdot 70=56$$

$$C_B^{'}=0{,}8\cdot C_B=0{,}8\cdot 175=140$$

Измененная временная диаграмма представлена на Рисунок 5

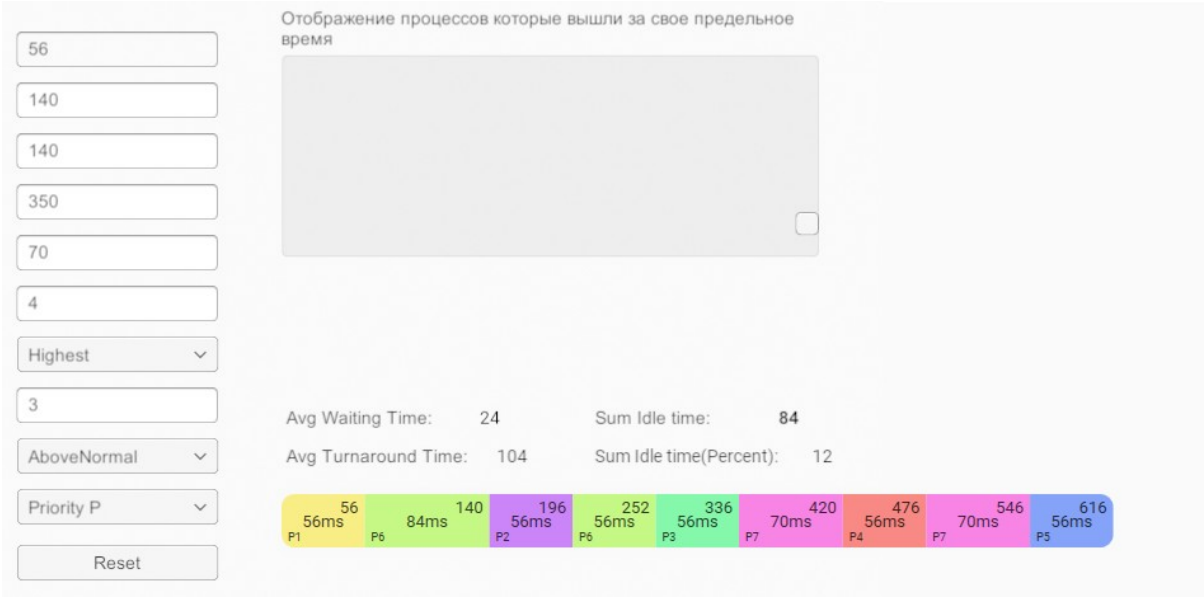

Рисунок 5: Планирование фиксированными приоритетами ($A > B$), результаты повышения производительности процессора

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

$$T_{прост} = \frac{84}{700} \cdot 100\,\% = 12\,\%$$

3.2 С квантованием времени и вытеснением, если приоритет потока B выше приоритета потока A (Рисунок 6).

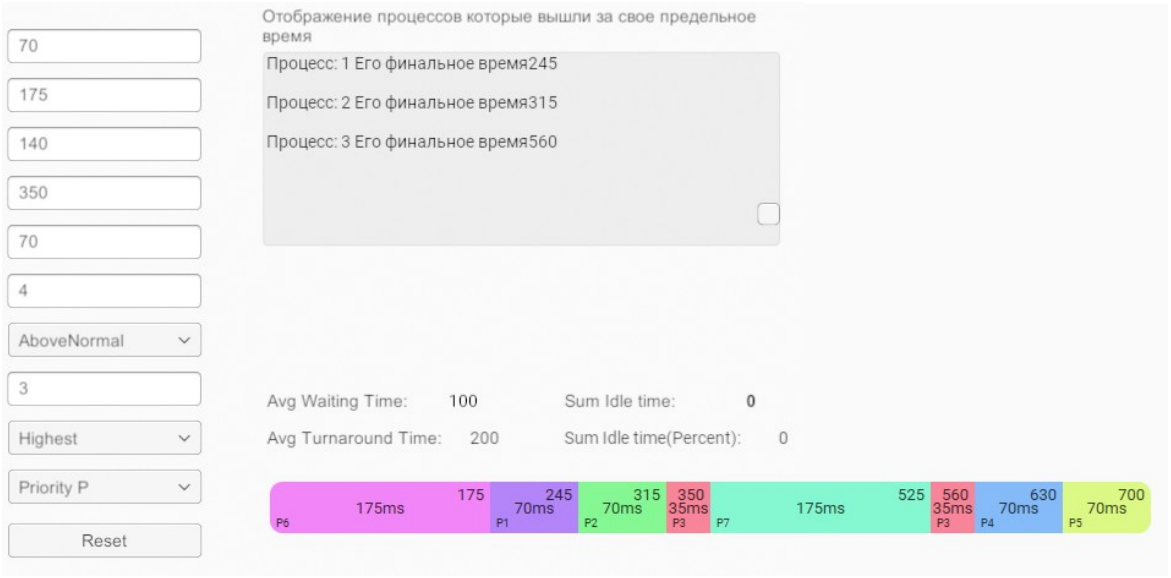

Рисунок 6: Планирование фиксированными приоритетами ($B > A$)

Данный алгоритм приводит к выполнению задания A с опозданием. Для устранения опозданий в режиме планирования квантование времени и вытеснение, если приоритет процесса B выше приоритета процесса A, нужно увеличить производительность работы процессора.

$$C'_{A+B} = 140;$$

$$\frac{175}{x} + \frac{70}{x} = 140 \quad 175 + 70 = 140 \cdot x;$$

$$245 = 140 \cdot x;$$

$$x = 1{,}75.$$

Следовательно:

$$C'_A = \frac{70}{1{,}75} = 40;$$

$$C'_B = \frac{175}{1{,}75} = 100.$$

Измененная временная диаграмма представлена на Рисунок 7

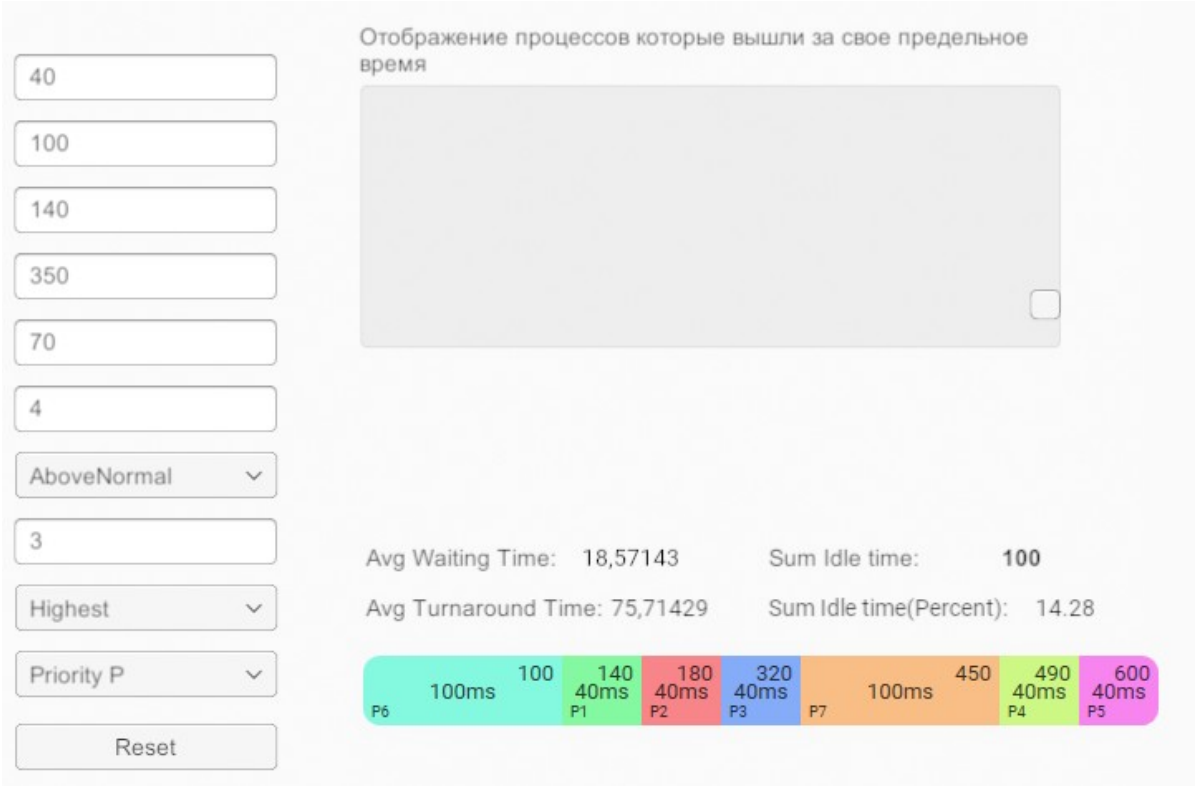

Рисунок 7: Планирование фиксированными приоритетами ($B > A$), результаты повышения производительности процессора

Опоздание A в итоге устранено, однако уменьшилась загрузка процессора.

$$T_{прост} = \frac{100}{700} \cdot 100\,\% \approx 14.28\,\%$$

3.3 Планирование с приоритетом процесса с наиболее ранним предельным сроком завершения задачи (Рисунок 8).

Работа данного метода заключается в том, что если единовременно поступают два процесса A и B, то первым начинает работу тот процесс, у которого предельный срок завершения задачи раньше.

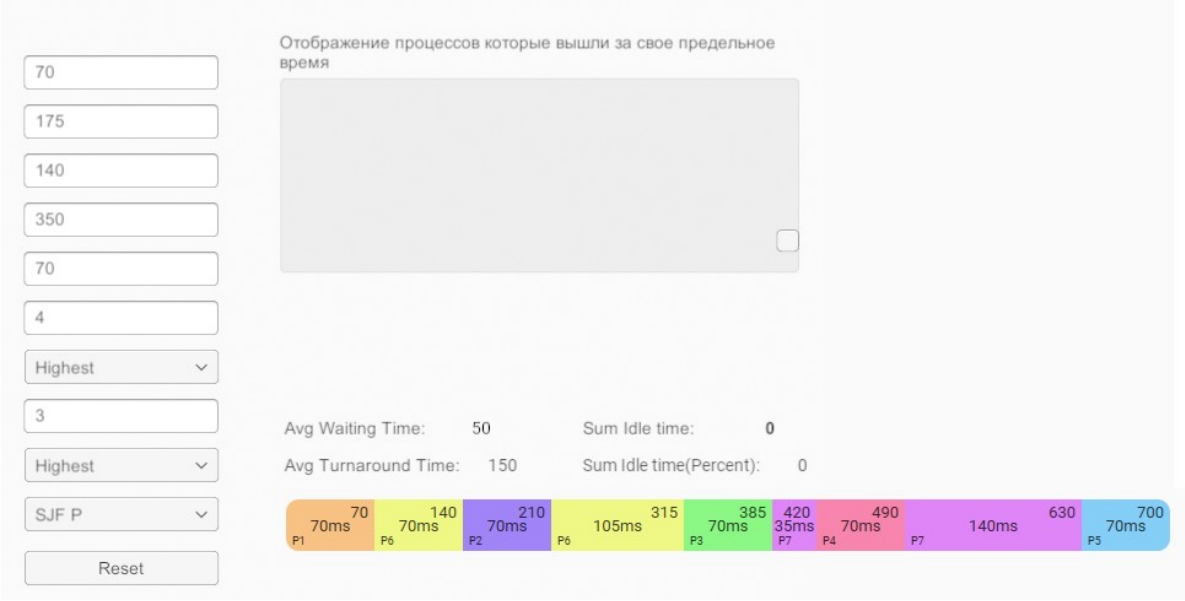

Рисунок 8: Планирование с наиболее ранним предельным сроком завершения

3.4 Частотно-монотонное планирование ().

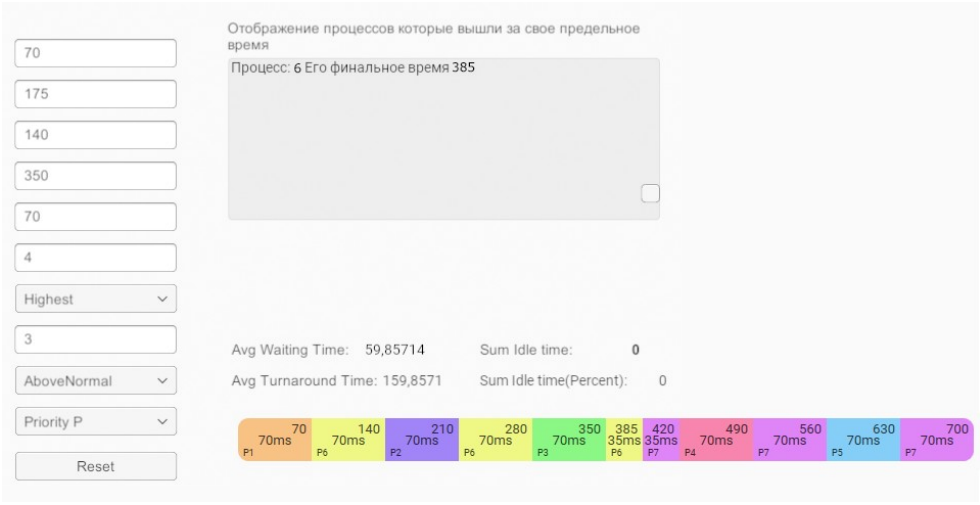

Рисунок 9: Частотно-монотонное планирование

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

4. Необходимо оценить возможность выполнения в реальном времени двух периодических заданий, обладающих следующими параметрами:
задание P_1: $C_1=70$; $T_1=140$;
задание P_2: $C_2=175$; $T_2=350$;
Предполагается, что существует n заданий, каждое из которых обладает своим фиксированным временем выполнения и периодом. Известно, что для n процессов необходимое условие выполнения процессами предельных сроков задается неравенством (1).
В Таблица 2 представлены некоторые значения верхней границы загруженности процессора для метода RMS для различных значений n. С увеличением количества заданий верхняя граница стремится к значению $ln(2)=0{,}693$.

Таблица 2

Значения верхней границы загруженности процессора для метода RMS

n	Значение $n\cdot\left(2^{\frac{1}{n}}-1\right)$
1	1,000
2	0,828
3	0,779
4	0,756
5	0,743
6	0,734

Загруженность процессора каждым из двух заданий составляет $U_1=U_2=0{,}5$.
Следовательно, общая загруженность процессора двумя заданиями равна $U=U_1+U_2=0{,}5+0{,}5=1$.
Верхняя граница загруженности этих задач в соответствии с методом RMS равна $0{,}828$.
Так как общая загруженность процессора обработкой двух представленных задач выше верхней границы по методу RMS ($1>0{,}828$), то можно говорить о невозможности успешного выполнения всех заданий в соответствии с RMS-планированием. Необходимо повышение производительности вычислительной системы, благодаря которому появится возможность выполнения этих заданий в режиме реального времени.

5. 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 A, B, C, D, E, для которых в Таблица 3 заданы, соответственно следующие значения: время поступления, время выполнения и предельные сроки начала работы (Рисунок 10). Далее приводятся временные диаграммы для следующих режимов планирования: наиболее ранний предельный срок (Рисунок 11), наиболее ранний срок со свободным временем простоя (Рисунок 12), ситуация «первым поступил – первым обслужен» (Рисунок 13).

Таблица 3

	Исходные данные		
Процесс	Время поступления	Время выполнения	Предельное время начала работы
A	80	160	880
B	160	160	160
C	320	160	400
D	400	160	720
E	480	160	560

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

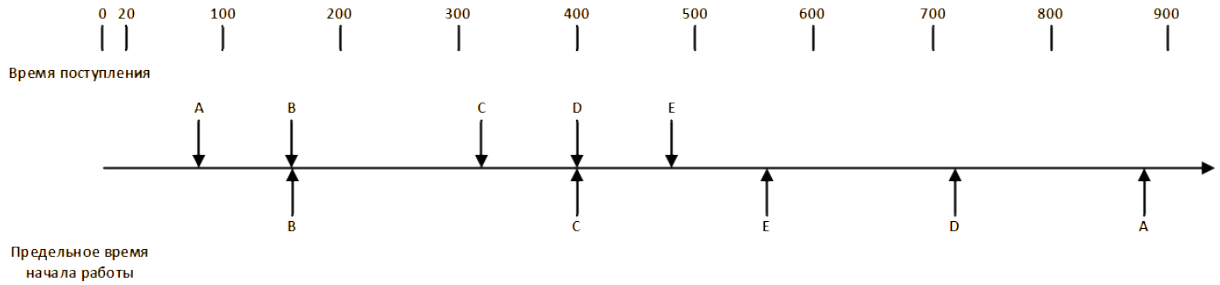

Рисунок 10: Временная диаграмма требований к выполнению процессов

Простейшей схемой планирования является запуск задания с наиболее ранним предельным сроком и выполнение его дол полного завершения.Такой подход не позволяет выполнить задание B, хотя оно и требует немедленного выполнения. Также отклоняется и задание E. В этом и состоит ключевой риск работы с непериодическими заданиями, особенно если они имеют предельное время начала выполнения.

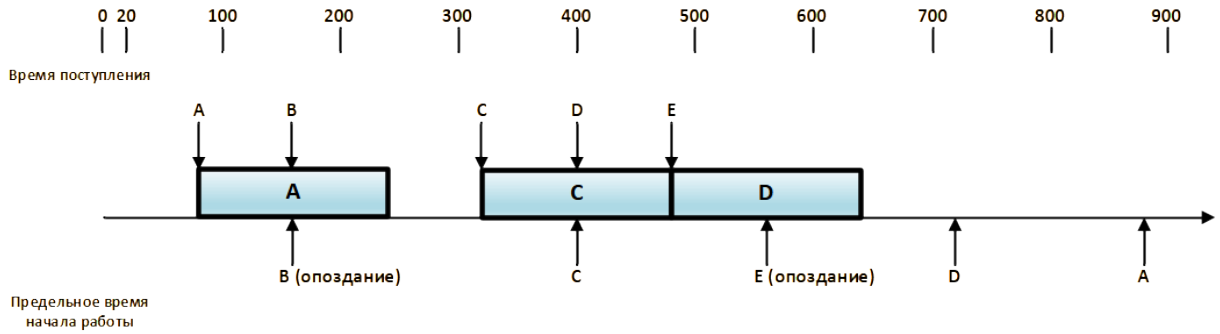

Рисунок 11: Временная диаграмма выполнения процессов для планирования с наиболее ранним предельным сроком выполнения

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 A, 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.

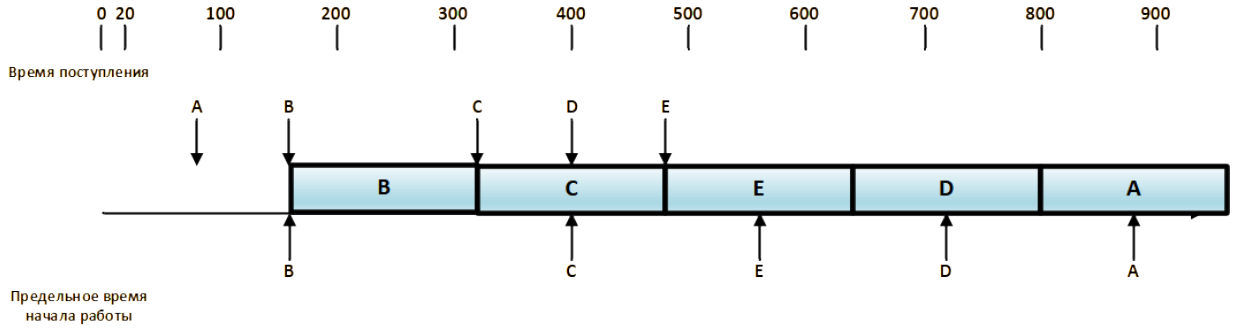

Рисунок 12: Временная диаграмма выполнения процессов для планирования с наиболее ранним предельным сроком выполнения со свободным временем простоя

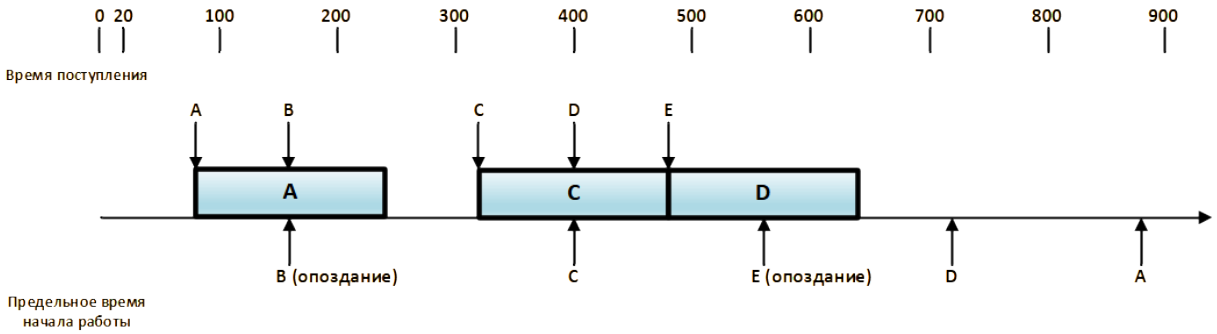

Рисунок 13: Временная диаграмма выполнения процессов для планирования«первым поступил – первым обслужен»

Следует отметить, что в схеме планировщика со свободным временем простоя (Рисунок 12) 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 0 («ложь») при вхождении в критическую секцию, и значение 1 («истина») при покидании ее. На Рисунок 14 представлен фрагмент алгоритма потока, который использует для взаимного исключения доступа к критическим данным блокирующую переменную $F(D)$. Перед входом в критическую секцию осуществляется проверка потоком, выполняется ли уже работа какого-либо потока с данными D. Если значение переменной $F(D)=0$, то данные заняты и производится циклический повтор проверки. Если данные являются свободными ($F(D)=1$), то переменной $F(D)$ присваивается значение 0, после чего выполняется вход потока в критическую секцию. После выполнения потоком всех манипуляций с данными D значение переменной $F(D)$ снова устанавливается в 1.

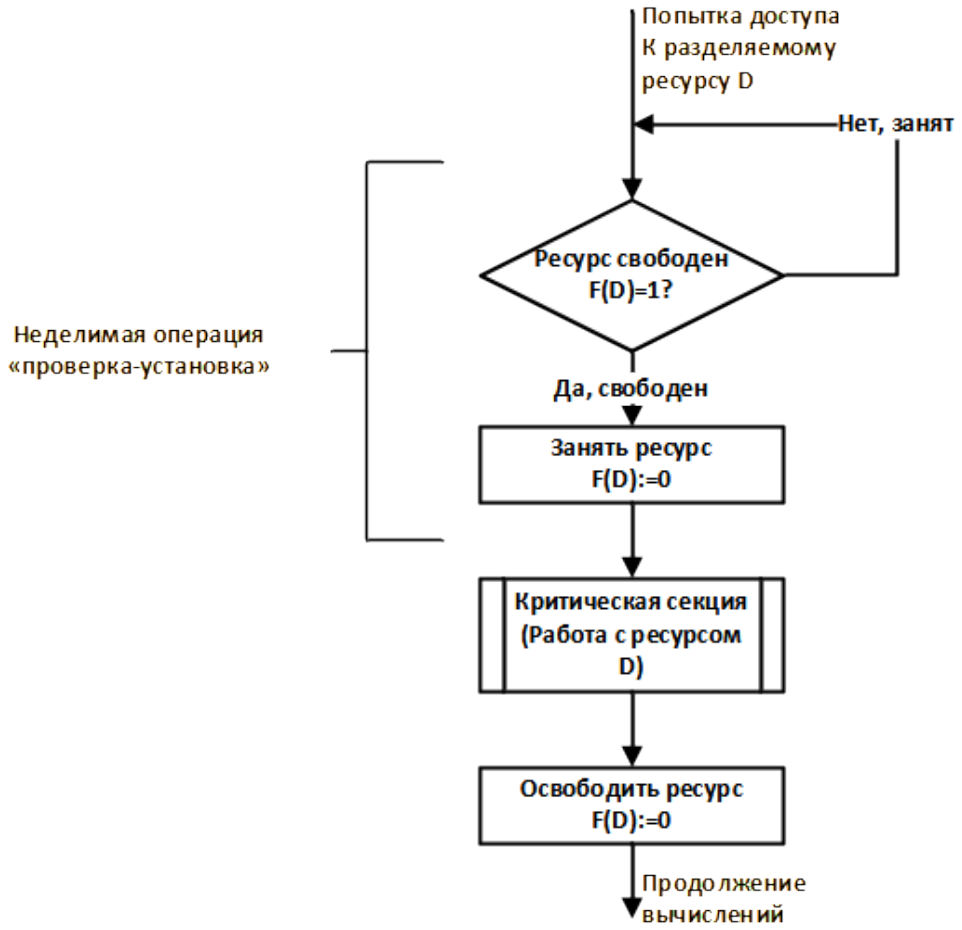

Рисунок 14: Обработка критических секций с помощью блокирующих переменных

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 0, 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 *BTC*, *BTR*, *BTS* процессора *Pentium*). 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 15 показано применение данных функций для реализации взаимного исключения в операционной системе *WIndows*. Перед началом изменения критических данных поток осуществляет выполнение системного вызова *EnterCriticalSection*(). 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 ($F(D)=0$), он, в отличие от предыдущего случая, не перейдет к циклическому опросу, а осуществит перевод потока в состояние ожидания (D) 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 *LeaveCriticalSection*(). Результатом этого будет присваивание блокирующей переменной значения, соответствующего свободному состоянию ресурса ($F(D)=1$). Кроме этого, операционная система осуществляет просмотр очереди ожидающих данный ресурс потоков с последующим переводом первого потока из очереди в состояние готовности.

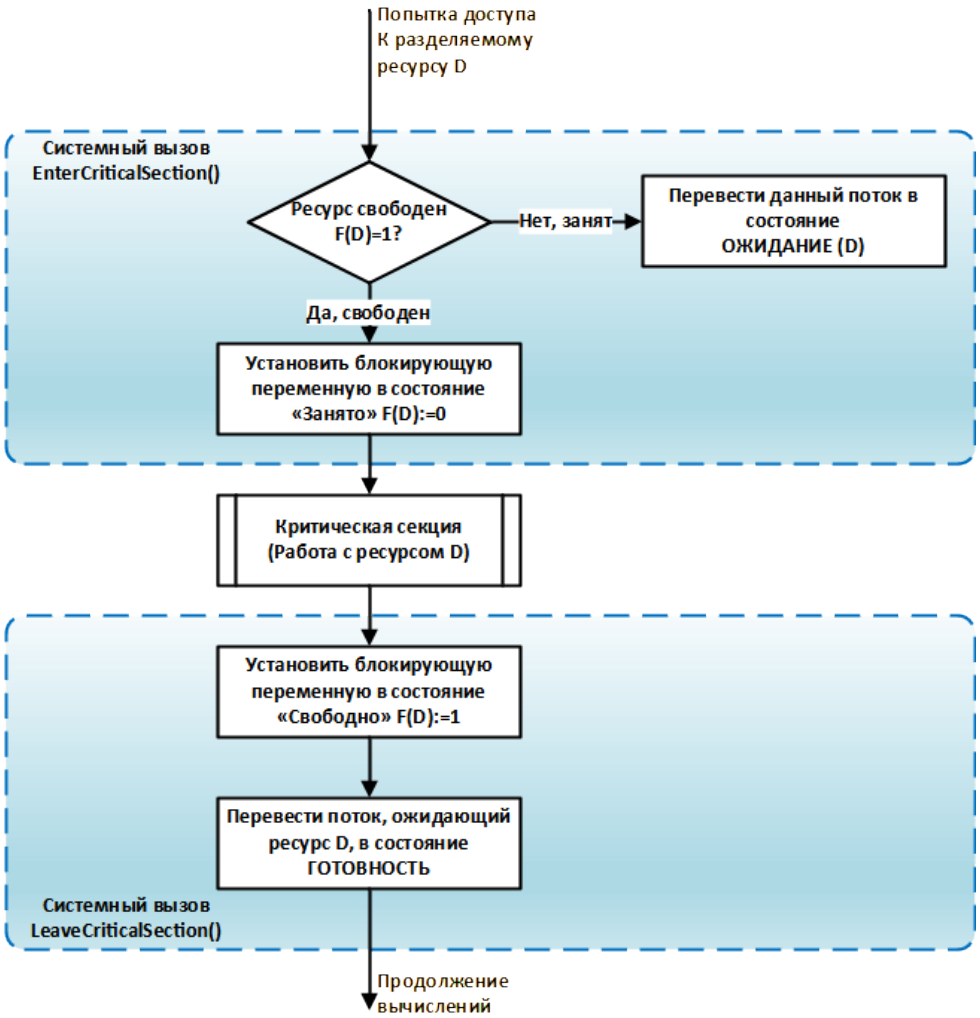

Рисунок 15: Обработка критических секций с помощью блокирующих переменных

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 «*DinnerPhilosopher*» рассматривающая практическую задачу «проблема обедающих философов». Суть задачи в том ,что 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 *Unity* на языке программирования *C*. Программный код приложения представлен в приложении Б.

Суть программы – запуск пяти потоков(философов), вызывающих метод *startAction*(), и осуществляющих взаимодействие с критическими переменными *leftResource* и *RightResource*. В зависимости от состояния объекту присваивается цвет материала. Так же есть ползунки которые задают в *ms* время состояния.

Таблица 3 Состояния Объекта

Состояние объекта	Цвет состояния
Ожидание(Idle)	Белый
Взятие объекта в руку(Pick object)	Зеленый
Думает(Thinking)	Оранжевый
Кушает(Eating)	Красный

Предполагается, что данный метод осуществит взятием философом обьекта. Но в ходе работы на практике осуществляется переключение между потоками, в результате чего значения переменных *leftResource* и *RightResource* становится невозможно предсказать. Например, результат может быть таким, как показано на Рисунок 16.

Рисунок 16: 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 *useLockSync* и конструкция *lock*. В операторе *lock* 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 *lock* дает возможность осуществления синхронизации потоков.

Для блокировки оператором *lock* применяется так называемый объект-заглушка (в приводимом примере – это переменные *leftResource* и *RightResource*). Чаще всего данная блокирующая переменная обладает типом *object*. При перемещении программного кода на оператор *lock* 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 *leftResource* и *RightResource* они становится доступным для других потоков.

Результаты работы программы с синхронизацией потоков с помощью блокирующих переменных представлены на Рисунок 17 (Кубики становятся красными только при условии что обе руки заняты).

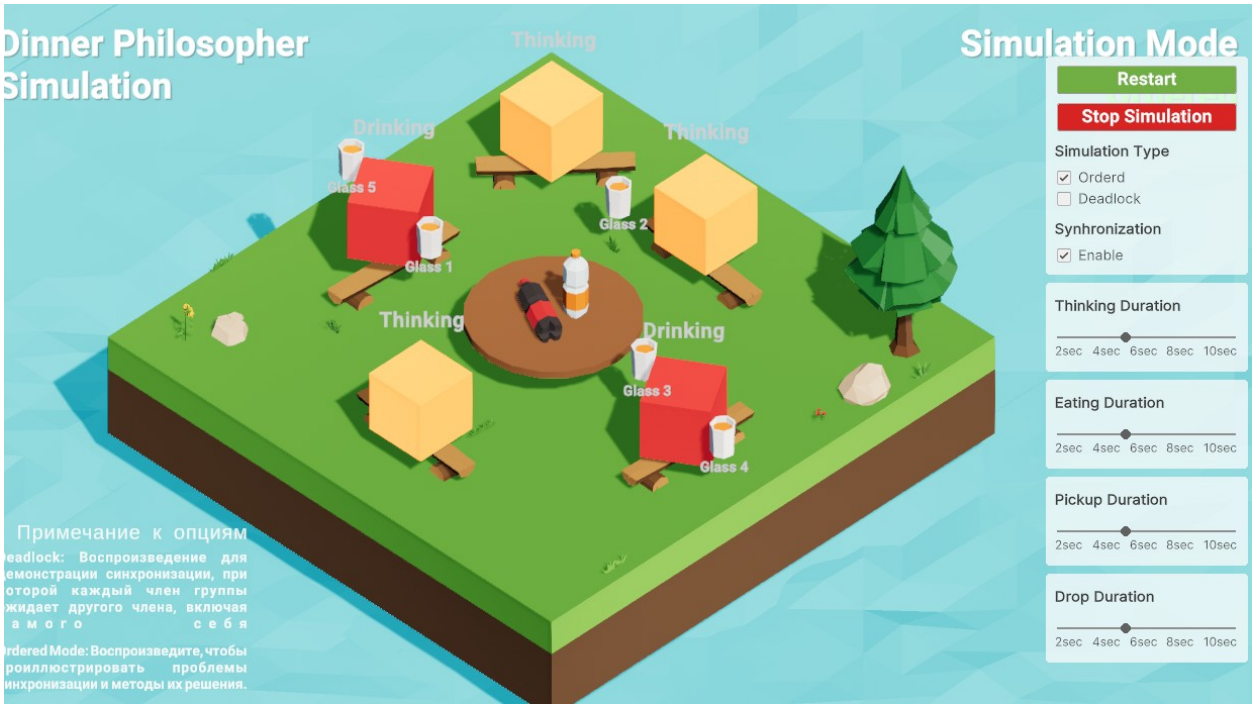

Рисунок 17: Результаты работы с синхронизацией

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 N узлов в графе выполняется пять шагов:

1. Задаются начальные условия: L-пустой список, все ребра немаркированы.
2. Текущий узел добавляем в конец списка L 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 19, Рисунок 21), протестированная на стандартных примерах.
В данную программу для быстрого вызова внесены примеры из методического указания, а также реализована проверка на корректность введенных данных (Рисунок 18, Рисунок 20).

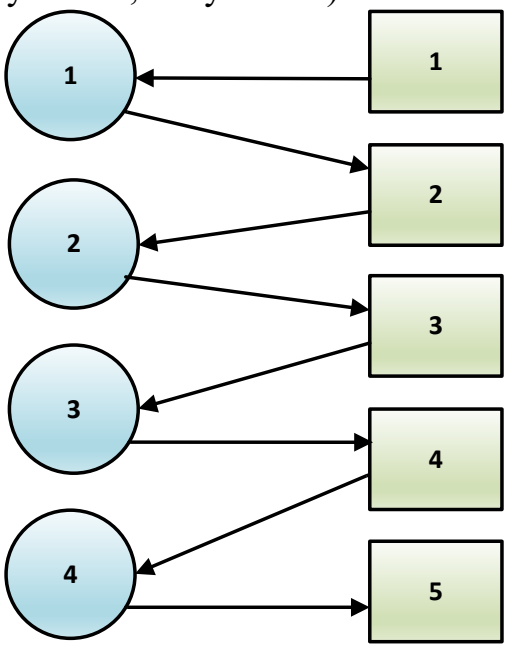

Рисунок 18 – Граф процессов и ресурсов к примеру на Рисунок 19

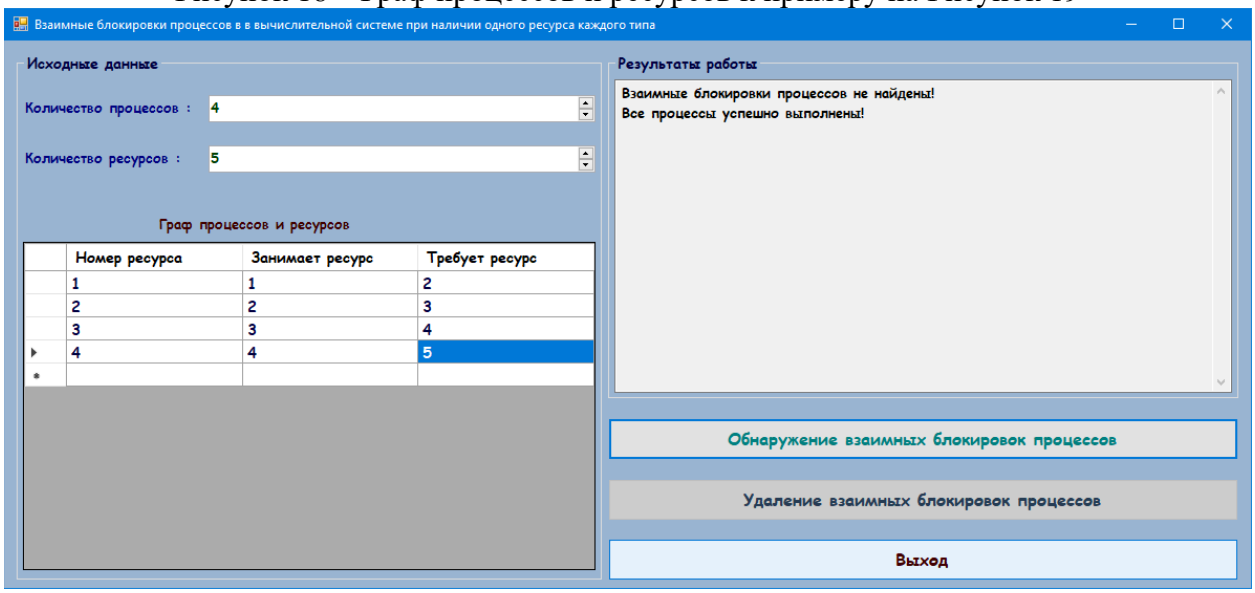

Рисунок 19 – Демонстрация работы тестового примера

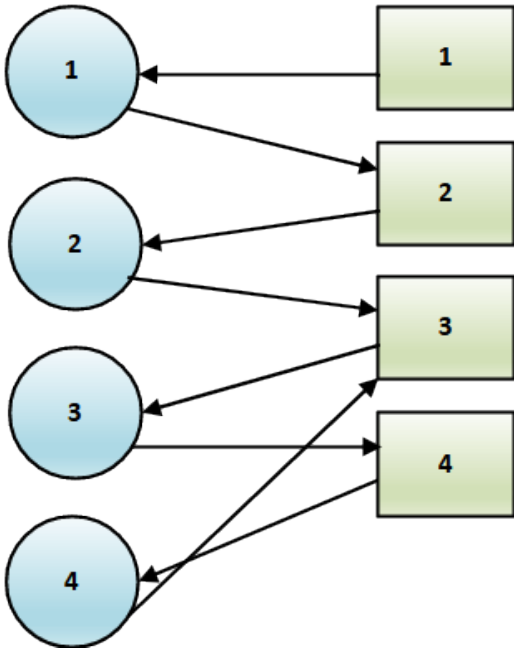

Рисунок 20 – Граф процессов и ресурсов к примеру на Рисунок 21

В данном случае обнаружены взаимные блокировки процессов.

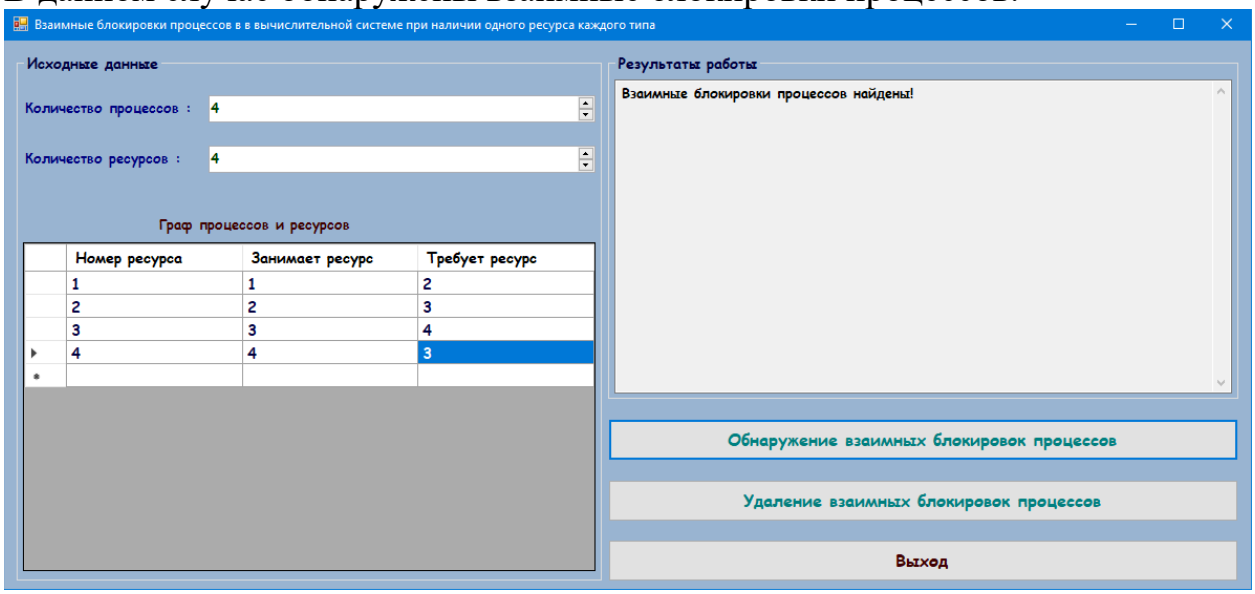

Рисунок 21 – Демонстрация работы тестового примера

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 22).

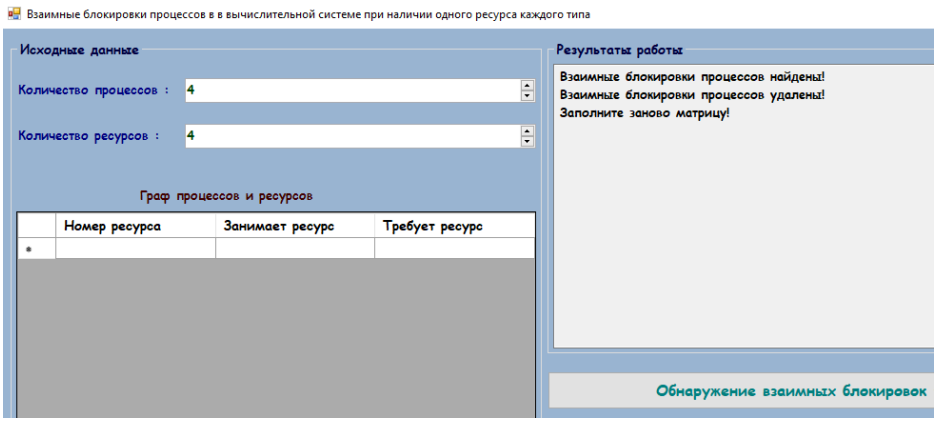

Рисунок 22 – Выход из взаимной блокировки

Полный код разработанного приложения «$DeadLocks_{App.sln}$» приведен в приложении В.

Задание №4

Условие задания

Исходные данные. Дана симметричная мультипроцессорная система. Все $N=3$ процессоров системы независимы, однотипны и функционально эквивалентны. Предельная скорость обмена по шине равна $V=160$, причем каждый процессор при решении задачи требует скорости обмена P= 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 . 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.

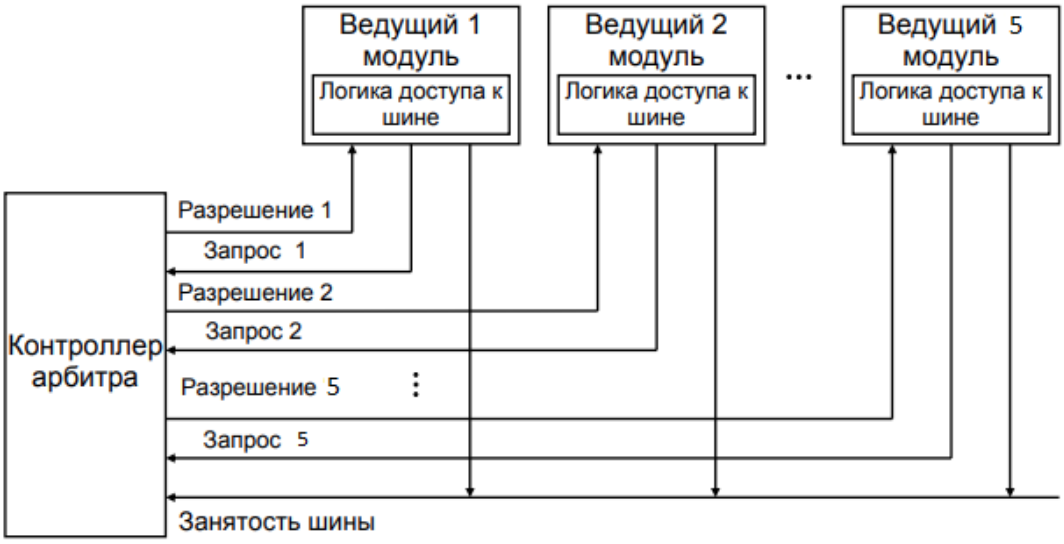

Рисунок 23: Структурная схема централизованного арбитража независимых запросов

Алгоритм централизованного параллельного арбитража выглядит следующим образом. На вход центрального арбитра осуществляется поступление сигналов ЗШ (запрос шины) по индивидуальным линиям. Ведущему модулю с номером i, 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 i, сигнала ШЗ (занятие шины).

После получения запроса от ведущего модуля, обладающего более высоким приоритетом, чем текущий ведущий модуль, происходит снятие арбитром сигнала $ПШ_i$ 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 $ПШ_i$ осуществляет снятие собственных сигналов ШЗ и $ЗШ_i$. 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 .

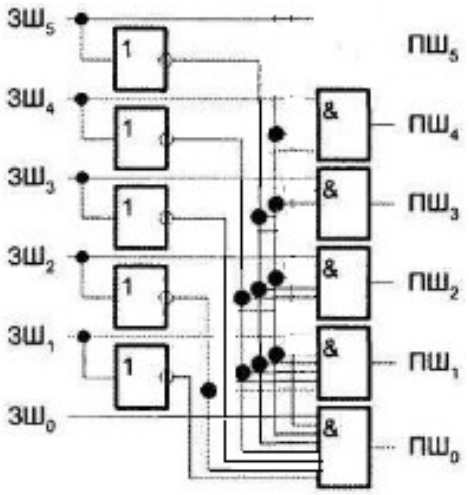

Рисунок 24: Функциональная схема централизованного арбитража независимых запросов глобальный арбитр

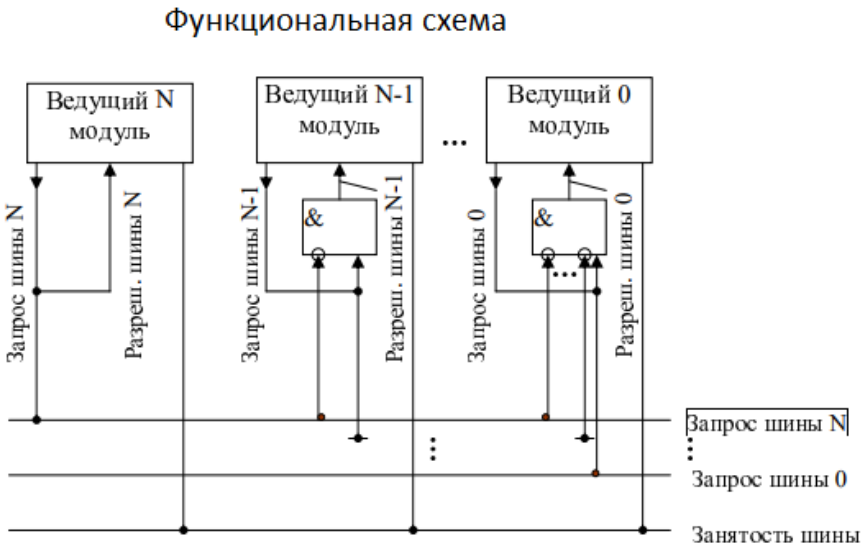

Рисунок 25: Функциональная схема централизованного арбитража независимых запросов локальные арбитры

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.

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

$$N = \frac{V}{P} \qquad (2)$$

Следовательно, $N = \frac{160}{60} \approx 2$.

Таким образом, наибольшее количество процессорных модулей, которые можно подключить к шине без достижения эффекта насыщения, составляет 2 модуля.

3. Эффект насыщения в шине для заданной схемы может быть предотвращен путем выбора более высокопроизводительной шины или использование локальной шины или коммутатором. Например, может быть использована быстрая шина $VME = 320$ Мб/с

Тогда:

$N = \frac{320}{60} \approx 5$

Эффект насыщения в шине для заданной схемы может быть предотвращен путем осуществление часть операций обмена данными, требующих высоких скоростей, не через шину ввода/вывода, а через шину процессора, примерно так же, как подключается внешний кэш. Рисунки наглядно демонстрируют различие между обычной архитектурой и архитектурой с локальной шиной.

Рисунок 26: Архитектура с локальной шиной

Рисунок 27: Обычная архитектура шин

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.

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

Листинг класса MainWindow.cs

```
using CPUScheduling_Sim.Source;
using MaterialDesignThemes.Wpf;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace CPUScheduling_Sim
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            // A process
            /*Scheduler.Processes.Add(new Process { PID = 1,
ArriveTime = TimeSpan.Zero, CPUTime = TimeSpan.FromMilliseconds(40),
Priority = 1,                   CriticalTime =
TimeSpan.FromMilliseconds(140) });
            Scheduler.Processes.Add(new Process { PID = 2, ArriveTime
= TimeSpan.FromMilliseconds(140), CPUTime =
TimeSpan.FromMilliseconds(40), Priority = 1,  CriticalTime =
TimeSpan.FromMilliseconds(280) });
            Scheduler.Processes.Add(new Process { PID = 3, ArriveTime
= TimeSpan.FromMilliseconds(280), CPUTime =
TimeSpan.FromMilliseconds(40), Priority = 1,  CriticalTime =
TimeSpan.FromMilliseconds(420) });
            Scheduler.Processes.Add(new Process { PID = 4, ArriveTime
= TimeSpan.FromMilliseconds(420), CPUTime =
TimeSpan.FromMilliseconds(40), Priority = 1,  CriticalTime =
TimeSpan.FromMilliseconds(560) });
```

```
            Scheduler.Processes.Add(new Process { PID = 5, ArriveTime
= TimeSpan.FromMilliseconds(560), CPUTime =
TimeSpan.FromMilliseconds(40), Priority = 1,  CriticalTime =
TimeSpan.FromMilliseconds(700) });

            // B process
            Scheduler.Processes.Add(new Process { PID = 6, ArriveTime
= TimeSpan.FromMilliseconds(0), CPUTime =
TimeSpan.FromMilliseconds(100), Priority = 0,   CriticalTime =
TimeSpan.FromMilliseconds(350) });
            Scheduler.Processes.Add(new Process { PID = 7, ArriveTime
= TimeSpan.FromMilliseconds(350), CPUTime =
TimeSpan.FromMilliseconds(100), Priority = 0, CriticalTime =
TimeSpan.FromMilliseconds(700) });*/
            Scheduler.Processes.Add(new Process { PID = 0, ArriveTime
= TimeSpan.FromMilliseconds(80), CPUTime =
TimeSpan.FromMilliseconds(160), Priority = 0, CriticalTime =
TimeSpan.FromMilliseconds(880) });
            Scheduler.Processes.Add(new Process { PID = 1, ArriveTime
= TimeSpan.FromMilliseconds(160), CPUTime =
TimeSpan.FromMilliseconds(160), Priority = 0, CriticalTime =
TimeSpan.FromMilliseconds(160) });
            Scheduler.Processes.Add(new Process { PID = 2, ArriveTime
= TimeSpan.FromMilliseconds(320), CPUTime =
TimeSpan.FromMilliseconds(160), Priority = 0, CriticalTime =
TimeSpan.FromMilliseconds(400) });
            Scheduler.Processes.Add(new Process { PID = 3, ArriveTime
= TimeSpan.FromMilliseconds(400), CPUTime =
TimeSpan.FromMilliseconds(160), Priority = 0, CriticalTime =
TimeSpan.FromMilliseconds(720) });
            Scheduler.Processes.Add(new Process { PID = 4, ArriveTime
= TimeSpan.FromMilliseconds(480), CPUTime =
TimeSpan.FromMilliseconds(160), Priority = 0, CriticalTime =
TimeSpan.FromMilliseconds(560) });

            DataContext = this;
            processTable.ItemsSource = Scheduler.Processes;
        }

        void Calculate()
        {
            if (algothimsCB.SelectedIndex == 0)
                return;

            if (Scheduler.Processes.Count == 0)
                return;

            Algorithm algorithm =
(Algorithm)algothimsCB.SelectedIndex;

            if(algorithm == Algorithm.ROUND_ROBIN)
```

```
            {
                timeQuantum.Visibility = Visibility.Visible;
                timeQuantum.Focus();

                int quantum;
                if(!int.TryParse(timeQuantum.Text, out quantum) ||
quantum == 0)
                {
                    return;
                }
                Scheduler.TimeQuantum =
TimeSpan.FromMilliseconds(quantum);
            }
            else
                timeQuantum.Visibility = Visibility.Hidden;

            try
            {
                var processes =
Scheduler.ScheduleProcesses(algorithm);

                if (processStack.Children.Count > 0)
                    processStack.Children.Clear();

                var chart = Chart.GenerateChart(processStack,
processes);
                foreach (var item in chart)
                    processStack.Children.Add(item);

                avgWaitingTime.Text = $"Average Waiting Time:
{processes.AverageWaitingTime.TotalMilliseconds}ms";
                avgTurnAroundTime.Text = $"Average Turn Around Time: {
processes.AverageTurnAroundTime.TotalMilliseconds}ms";
            }
            catch(Exception ex)
            {
                MessageBox.Show(ex.Message, "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
            }
        }

        private void addProcess_Click(object sender, RoutedEventArgs
e)
        {
            arriveTimeTB.Text = criticalTimeTB.Text = cpuTimeTB.Text =
priorityTB.Text = string.Empty;
            addDialogBox.IsOpen = true;
        }

        private void dialog_AddProcess_Click(object sender,
RoutedEventArgs e)
```

```
        {
            AddProcess();
        }
        void AddProcess()
        {
            float arriveTime, cpuTime, criticalTime;
            int priority;

            if (!float.TryParse(arriveTimeTB.Text, out arriveTime) ||
arriveTime < 0)
            {
                MessageBox.Show("Arrive Time is invalid");
                return;
            }
            if (!float.TryParse(cpuTimeTB.Text, out cpuTime) ||
cpuTime < 0)
            {
                MessageBox.Show("CPU Time is invalid");
                return;
            }
            if (!int.TryParse(priorityTB.Text, out priority) ||
priority < 0)
            {
                MessageBox.Show("Priority is invalid");
                return;
            }
            if (!float.TryParse(criticalTimeTB.Text, out criticalTime)
|| criticalTime < 0)
            {
                MessageBox.Show("Critical Time is invalid");
                return;
            }

            Process process = new Process
            {
                PID = Scheduler.Processes.Count > 0 ?
Scheduler.Processes[Scheduler.Processes.Count - 1].PID + 1 : 1,
                ArriveTime = TimeSpan.FromMilliseconds(arriveTime),
                CPUTime = TimeSpan.FromMilliseconds(cpuTime),
                Priority = priority,
                CriticalTime = TimeSpan.FromMilliseconds(criticalTime)
            };
            Scheduler.Processes.Add(process);

            processTable.Items.Refresh();
            Calculate();

            addDialogBox.IsOpen = false;

        }
```

```
        private void dialog_Cancel_Click(object sender,
RoutedEventArgs e)
        {
            addDialogBox.IsOpen = false;
        }

        private void algothimsCB_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            Calculate();
        }

        private void clearBtn_Click(object sender, RoutedEventArgs e)
        {
            Scheduler.Processes.Clear();

            processTable.Items.Refresh();

            if (processStack.Children.Count > 0)
                processStack.Children.Clear();

            avgWaitingTime.Text = "Average Waiting Time:";
            avgTurnAroundTime.Text = "Average Turn Around Time:";

        }

        private void processTable_PreviewKeyDown(object sender,
KeyEventArgs e)
        {
            var selectedIndex = processTable.SelectedIndex;
            if (e.Key != Key.Delete || selectedIndex < 0)
                return;

            Scheduler.Processes.RemoveAt(selectedIndex);

            processTable.Items.Refresh();
            Calculate();
        }

        private void addBtn_PreviewKeyDown(object sender, KeyEventArgs
e)
        {
            if(e.Key == Key.Enter)
                AddProcess();
        }

        private void timeQuantum_TextChanged(object sender,
TextChangedEventArgs e)
        {
            Calculate();
        }
```

```
    }
}
```

Листинг класса Chart.cs

```
using CPUScheduling_Sim.Source;
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;

namespace CPUScheduling_Sim
{
    internal class Chart
    {
        private static Dictionary<int, Color> chartColors = new
Dictionary<int, Color>();

        /// <summary>
        /// Generates gantt chart from processes
        /// </summary>
        /// <param name="panel">panel in UI</param>
        /// <param name="processes">processes to generate chart
from</param>
        /// <returns>List of grids representing the gantt
chart</returns>
        public static List<Grid> GenerateChart(StackPanel panel,
Processes processes)
        {
            List<Grid> grids = new List<Grid>(processes.Count);
            double totalTime = Scheduler.FindCompletionTime(processes,
processes.Count - 1).TotalMilliseconds;
            var offset = Scheduler.FindCompletionTime(processes, 0);

            if(processes[0].ArriveTime != TimeSpan.Zero)
            {
                var width = (processes[0].ArriveTime.TotalMilliseconds
/ totalTime) * (panel.Width - 5);
                var block = BlankBlock(panel.Height, width);
                foreach (var border in block.Children)
                {
                    ((Border)border).CornerRadius = new
CornerRadius(10, 0, 0, 10);
                }
                grids.Add(block);
            }

            for (int i = 0; i < processes.Count; i++)
            {
```

```
                Process? process = processes[i];
                Grid grid = new Grid();

                var thisOffset =
Scheduler.FindCompletionTime(processes, i);

                if (i > 0 && thisOffset - offset != process.CPUTime)
                {
                    var _width = ((thisOffset - offset -
process.CPUTime).TotalMilliseconds / totalTime) * (panel.Width - 5);
                    grids.Add(BlankBlock(panel.Height, _width));
                }

                Border block = new Border();

                block.Height = panel.Height;
                var width = (process.CPUTime.TotalMilliseconds /
totalTime) * (panel.Width - 5);
                offset = thisOffset;

                DoubleAnimation widthAnimation = new
DoubleAnimation();
                widthAnimation.From = 0;
                widthAnimation.To = width;
                widthAnimation.EasingFunction = new QuinticEase();
                widthAnimation.Duration = new
Duration(TimeSpan.FromSeconds(.5));

                block.BeginAnimation(Border.WidthProperty,
widthAnimation);

                GenerateColors();

                ColorAnimation chartColorAnimation = new
ColorAnimation();
                chartColorAnimation.From = Colors.Transparent;
                chartColorAnimation.To = chartColors[process.PID];
                chartColorAnimation.EasingFunction = new
QuinticEase();
                chartColorAnimation.Duration = new
Duration(TimeSpan.FromSeconds(.3));

                var brush = new SolidColorBrush(Colors.Transparent);
                brush.BeginAnimation(SolidColorBrush.ColorProperty,
chartColorAnimation);

                block.Background = brush;

                if (grids.Count == 0)
                    block.CornerRadius = new CornerRadius(10, 0, 0,
10);
```

```
                else if(i == processes.Count - 1)
                    block.CornerRadius = new CornerRadius(0, 10, 10,
0);

                TextBlock cpuTime = new TextBlock();

                cpuTime.Text =
$"{process.CPUTime.TotalMilliseconds}ms";
                cpuTime.HorizontalAlignment =
HorizontalAlignment.Center;
                cpuTime.VerticalAlignment = VerticalAlignment.Center;
                cpuTime.TextAlignment = TextAlignment.Center;

                TextBlock ct = new TextBlock();
                ct.HorizontalAlignment = HorizontalAlignment.Right;
                ct.VerticalAlignment = VerticalAlignment.Top;
                ct.Margin = new Thickness(0, 3, 5, 0);
                ct.Text = $"{thisOffset.Milliseconds}";

                TextBlock pid = new TextBlock();
                pid.HorizontalAlignment = HorizontalAlignment.Left;
                pid.VerticalAlignment = VerticalAlignment.Bottom;
                pid.Margin = new Thickness(5, 0, 0, 3);
                pid.Text = $"P{process.PID}";
                pid.FontSize = 9;

                grid.Children.Add(block);
                grid.Children.Add(cpuTime);
                grid.Children.Add(ct);
                grid.Children.Add(pid);

                grids.Add(grid);
            }
            return grids;
        }

        static double? currentHue = null;
        /// <summary>
        /// Generates random color palette using the golden ratio
scheme
        /// </summary>
        private static void GenerateColors()
        {
            var processes = Scheduler.Processes;
            var processCount = processes.Count;

            double goldenRatioConjugate = 0.618033988749895;
            currentHue ??= new Random().NextDouble();

            for (int i = 0; i < processCount; i++)
            {
```

```
                if (chartColors.ContainsKey(processes[i].PID))
                    continue;

                chartColors.Add(processes[i].PID,
HSLToRGB((double)currentHue, .9, .75));
                currentHue += goldenRatioConjugate;
                currentHue %= 1;
            }
        }

        private static Color HSLToRGB(double h, double s, double l)
        {
            double[] t = new double[] { 0, 0, 0 };
            byte r = 0;
            byte g = 0;
            byte b = 0;

            if (s == 0)
            {
                r = g = b = (byte)(l * 255);
                return Color.FromRgb(r, g, b);
            }

            double q, p;

            q = l < 0.5 ? l * (1 + s) : l + s - (l * s);
            p = 2 * l - q;

            t[0] = h + (1.0 / 3.0);
            t[1] = h;
            t[2] = h - (1.0 / 3.0);

            for (byte i = 0; i < 3; i++)
            {
                t[i] = t[i] < 0 ? t[i] + 1.0 : t[i] > 1 ? t[i] - 1.0 :
t[i];

                if (t[i] * 6.0 < 1.0)
                    t[i] = p + ((q - p) * 6 * t[i]);
                else if (t[i] * 2.0 < 1.0)
                    t[i] = q;
                else if (t[i] * 3.0 < 2.0)
                    t[i] = p + ((q - p) * 6 * ((2.0 / 3.0) - t[i]));
                else
                    t[i] = p;
            }
            return Color.FromRgb((byte)(t[0] * 255), (byte)(t[1] *
255), (byte)(t[2] * 255));
        }
        private static Grid BlankBlock(double height, double width)
        {
```

```
            Border[] rec = new Border[2];

            DoubleAnimation widthAnimation = new DoubleAnimation();
            widthAnimation.From = 0;
            widthAnimation.To = width;
            widthAnimation.EasingFunction = new QuinticEase();
            widthAnimation.Duration = new
Duration(TimeSpan.FromSeconds(.5));

            for (int j = 0; j < rec.Length; j++)
            {
                var r = new Border();
                r.Height = height;
                r.BeginAnimation(Border.WidthProperty,
widthAnimation);
                rec[j] = r;
            }
            rec[0].Background = PatternBrush();
            rec[1].Background = new SolidColorBrush(Color.FromArgb(20,
0, 0, 0));

            var blank = new Grid();
            //blank.Children.Add(rec[0]);

            //blank.Children.Add(rec[1]);

            return blank;
        }
        private static Brush PatternBrush()
        {
            VisualBrush vb = new VisualBrush();

            vb.TileMode = TileMode.Tile;
            vb.Viewport = new Rect(0, 0, 10, 10);
            vb.ViewportUnits = BrushMappingMode.Absolute;

            vb.Viewbox = new Rect(0, 0, 10, 10);
            vb.ViewboxUnits = BrushMappingMode.Absolute;

            Line line = new Line();
            line.Stroke = Brushes.Gray;
            var rotation = new RotateTransform();
            rotation.Angle = 45;
            line.RenderTransform = rotation;
            line.X1 = 0;
            line.Y1 = 0;
            line.X2 = 10;
            line.Y2 = 0;

            vb.Visual = line;
```

```
            return vb;
        }
    }
}
```

Листинг класса Process.cs

```
using System;
using System.Collections.Generic;

namespace CPUScheduling_Sim.Source
{
    public class Process : ICloneable
    {
        public int PID { get; set; }
        public TimeSpan ArriveTime { get; set; }
        public TimeSpan CPUTime { get; set; }
        public TimeSpan CriticalTime { get; set; }
        public int Priority { get; set; }

        public object Clone()
        {
            var process = (Process)MemberwiseClone();
            return process;
        }
    }

    public class Processes : List<Process>, ICloneable
    {
        public TimeSpan AverageWaitingTime { get; set; }
        public TimeSpan AverageTurnAroundTime { get; set; }

        public object Clone()
        {
            Processes processes = new Processes();
            foreach (var process in this)
            {
                processes.Add((Process)process.Clone());
            }
            return processes;
        }
    }
}
```

Листинг класса Scheduler.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace CPUScheduling_Sim.Source
{
    internal enum Algorithm
    {
```

```
        FCFS = 1,
        SJF_PREEMPTIVE,
        SJF_NONPREEMPTIVE,
        PRIORITY_PREEMPTIVE,
        PRIORITY_NONPREEMPTIVE,
        ROUND_ROBIN
    }
    internal static class Scheduler
    {
        /// <summary>
        /// The list of the processes.
        /// </summary>
        public static Processes Processes { get; set; } = new
Processes();
        public static TimeSpan TimeQuantum { get; set; }

        /// <summary>
        /// Schedules processes by the given algorithm
        /// </summary>
        /// <param name="algorithm">algorithm to be used to schedule
the processes</param>
        /// <returns>Scheduled processes based on the given
algorithm</returns>
        /// <exception cref="NotImplementedException">Thrown when the
given algorithm is not implemented</exception>
        public static Processes ScheduleProcesses(Algorithm algorithm)
=> algorithm switch
        {
            Algorithm.FCFS => FCFSSchedule(),
            Algorithm.SJF_PREEMPTIVE => SJFPreSchedule(),
            Algorithm.SJF_NONPREEMPTIVE => SJFNonPreSchedule(),
            Algorithm.PRIORITY_PREEMPTIVE => PriorityPreSchedule(),
            Algorithm.PRIORITY_NONPREEMPTIVE =>
PriorityNonPreSchedule(),
            Algorithm.ROUND_ROBIN => RoundRobinSchedule(),
            _ => throw new NotImplementedException("Please select a
valid algorithm")
        };

        /// <summary>
        /// Schedule processes based on the FCFS algorithm and
calculates the average waiting and turn around time
        /// </summary>
        /// <returns>Processes scheduled based on the FCFS
algorithm</returns>
        private static Processes FCFSSchedule()
        {
            Processes processes = new Processes();

            processes.AddRange(Processes.OrderBy(p => p.ArriveTime));
            //Unccoment if you want calculate with FCFS alg
```

```
            //Under section comment for output graphics without alpply
schedule alg.
            CalculateNonPreAverageTime(processes);
            CalculateNonPreCriticalTime(processes);

            return processes;
        }

        /// <summary>
        /// Schedule processes based on the Preemptive SJF algorithm
and calculates the average waiting and turn around time
        /// </summary>
        /// <returns>Processes scheduled based on the Preemptive SJF
algorithm</returns>
        private static Processes SJFPreSchedule()
        {
            Processes processes = new Processes();
            Processes final = new Processes();
            // do the sjf preemptive sort and calculate properties

processes.AddRange(((Processes)Processes.Clone()).OrderBy(p =>
p.ArriveTime).ThenBy(p => p.CPUTime));

            var completionTime = FindCompletionTime(processes,
processes.Count - 1);
            var timer = processes[0].ArriveTime;
            Process current = new Process { PID = processes[0].PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
            int i = 0;

            while (timer <= completionTime)
            {
                var ms = TimeSpan.FromMilliseconds(1);

                current.CPUTime += ms;
                processes[i].CPUTime -= ms;
                timer += ms;

                var next = processes.Find(p => p.CPUTime <
processes[i].CPUTime && timer >= p.ArriveTime);

                if (processes[i].CPUTime == TimeSpan.Zero)
                {
                    if (processes.Count > 1)
                        processes.Remove(processes[i]);

                    next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.CPUTime);
                    while (timer <= completionTime && next is null)
                    {
                        timer += ms;
```

```
                        next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.CPUTime);
                    }

                }

                if (next is not null)
                {
                    final.Add(current);

                    i = processes.IndexOf(next);
                    current = new Process { PID = processes[i].PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
                }
            }
            // Calculating Turn around time and waiting time
            CalculatePreAverageTime(final);
            CalculatePreCriticalTime(final);

            return final;
        }
        /// <summary>
        /// Schedule processes based on the Non-Preemptive SJF
algorithm and calculates the average waiting and turn around time
        /// </summary>
        /// <returns>Processes scheduled based on the Non-Preemptive
SJF algorithm</returns>
        private static Processes SJFNonPreSchedule()
        {
            Processes processes = new Processes();
            Processes final = new Processes();

processes.AddRange(((Processes)Processes.Clone()).OrderBy(p =>
p.ArriveTime).ThenBy(p => p.CPUTime));

            var completionTime = FindCompletionTime(processes,
processes.Count - 1);
            var timer = processes[0].ArriveTime;
            Process current = new Process { PID = processes[0].PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
            int i = 0;

            while (timer <= completionTime)
            {
                var ms = TimeSpan.FromMilliseconds(1);

                timer += ms;
                current.CPUTime += ms;
                processes[i].CPUTime -= ms;

                if (processes[i].CPUTime > TimeSpan.Zero)
```

```
                    continue;

                processes.Remove(processes[i]);

                if (processes.Count == 0)
                {
                    final.Add(current);
                    break;
                }
                var next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.CPUTime);
                while (timer <= completionTime && next is null)
                {
                    timer += ms;
                    next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.CPUTime);
                }

                if (next is not null)
                {
                    final.Add(current);

                    i = processes.IndexOf(next);
                    current = new Process { PID = processes[i].PID,
ArriveTime = processes[i].ArriveTime, CPUTime = TimeSpan.Zero };
                }
            }

            CalculateNonPreAverageTime(final);
            CalculateNonPreCriticalTime(final);

            return final;

        }

        /// <summary>
        /// Schedule processes based on the Preemptive Priority
algorithm and calculates the average waiting and turn around time
        /// </summary>
        /// <returns>Processes scheduled based on the Preemptive
Priority algorithm</returns>
        private static Processes PriorityPreSchedule()
        {
            Processes processes = new Processes();
            Processes final = new Processes();
            // do the priority sort and calculate properties

processes.AddRange(((Processes)Processes.Clone()).OrderBy(p =>
p.ArriveTime).ThenBy(p => p.Priority));
```

```
            var completionTime = FindCompletionTime(processes,
processes.Count - 1);
            var timer = processes[0].ArriveTime;
            Process current = new Process { PID = processes[0].PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
            int i = 0;

            while (timer <= completionTime)
            {
                var ms = TimeSpan.FromMilliseconds(1);

                current.CPUTime += ms;
                processes[i].CPUTime -= ms;
                timer += ms;

                var next = processes.Find(p => p.Priority <
processes[i].Priority && timer >= p.ArriveTime);

                if (processes[i].CPUTime == TimeSpan.Zero)
                {
                    if (processes.Count > 1)
                        processes.Remove(processes[i]);
                    next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.Priority);
                    while (timer <= completionTime && next is null)
                    {
                        timer += ms;
                        next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.Priority);
                    }
                }

                if (next is not null)
                {
                    final.Add(current);

                    i = processes.IndexOf(next);
                    current = new Process { PID = processes[i].PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
                }
            }

            CalculatePreAverageTime(final);
            CalculatePreCriticalTime(final);

            return final;
        }

        /// <summary>
        /// Schedule processes based on the Non-Preemptive Priority
algorithm and calculates the average waiting and turn around time
```

```
        /// </summary>
        /// <returns>Processes scheduled based on the Non-Preemptive
Priority algorithm</returns>
        private static Processes PriorityNonPreSchedule()
        {
            Processes processes = new Processes();
            Processes final = new Processes();
            // do the priority sort and calculate properties

processes.AddRange(((Processes)Processes.Clone()).OrderBy(p =>
p.ArriveTime).ThenBy(p => p.Priority));

            var completionTime = FindCompletionTime(processes,
processes.Count - 1);
            var timer = processes[0].ArriveTime;
            Process current = new Process { PID = processes[0].PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
            int i = 0;

            while (timer <= completionTime)
            {
                var ms = TimeSpan.FromMilliseconds(1);

                current.CPUTime += ms;
                processes[i].CPUTime -= ms;
                timer += ms;

                if (processes[i].CPUTime > TimeSpan.Zero)
                    continue;

                processes.Remove(processes[i]);

                if (processes.Count == 0)
                {
                    final.Add(current);
                    break;
                }
                var next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.Priority);
                while (timer <= completionTime && next is null)
                {
                    timer += ms;
                    next = processes.FindAll(p => timer >=
p.ArriveTime).MinBy(p => p.Priority);
                }

                if (next != null)
                {
                    final.Add(current);

                    i = processes.IndexOf(next);
```

```
                    current = new Process { PID = processes[i].PID,
ArriveTime = processes[i].ArriveTime, CPUTime = TimeSpan.Zero };
                }
            }

            CalculateNonPreAverageTime(final);
            CalculateNonPreCriticalTime(final);

            return final;

        }
        private static Processes RoundRobinSchedule()
        {
            Processes processes = new Processes();
            Processes final = new Processes();

processes.AddRange(((Processes)Processes.Clone()).OrderBy(p =>
p.ArriveTime));

            var readyQueue = new Queue<Process>();
            var quantum = TimeQuantum;
            var completionTime = FindCompletionTime(processes,
processes.Count - 1);
            var timer = processes[0].ArriveTime;

            Process current = new Process { PID = processes[0].PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };

            var next = processes.FindAll(p => timer == p.ArriveTime);
            foreach (var process in next)
            {
                readyQueue.Enqueue(process);
            }

            Process instance = readyQueue.Dequeue();

            while (timer <= completionTime)
            {

                var ms = TimeSpan.FromMilliseconds(1);

                quantum -= ms;

                current.CPUTime += ms;
                instance.CPUTime -= ms;
                timer += ms;

                next = processes.FindAll(p => timer == p.ArriveTime);
                foreach (var process in next)
                {
                    readyQueue.Enqueue(process);
```

```
                }

                if (quantum == TimeSpan.Zero)
                {
                    if (instance.CPUTime != TimeSpan.Zero)
                        readyQueue.Enqueue(instance);
                    else
                        processes.Remove(instance);

                    final.Add(current);

                    while (timer <= completionTime && readyQueue.Count
== 0 && processes.Count > 0)
                    {
                        timer += ms;
                        next = processes.FindAll(p => timer ==
p.ArriveTime);
                        foreach (var process in next)
                            readyQueue.Enqueue(process);
                    }
                    if (timer >= completionTime)
                        break;

                    instance = readyQueue.Dequeue();

                    current = new Process { PID = instance.PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
                }

                if (instance.CPUTime == TimeSpan.Zero)
                {
                    processes.Remove(instance);
                    final.Add(current);

                    while (timer <= completionTime && readyQueue.Count
== 0 && processes.Count > 0)
                    {
                        timer += ms;
                        next = processes.FindAll(p => timer ==
p.ArriveTime);
                        foreach (var process in next)
                            readyQueue.Enqueue(process);
                    }
                    if (timer >= completionTime)
                        break;

                    instance = readyQueue.Dequeue();
                    quantum = TimeQuantum;
                    current = new Process { PID = instance.PID,
ArriveTime = timer, CPUTime = TimeSpan.Zero };
```

```
                }

                if (quantum == TimeSpan.Zero)
                {
                    quantum = TimeQuantum;
                }

            }
            // Calculating Turn around time and waiting time
            CalculatePreAverageTime(final);
            CalculatePreCriticalTime(final);

            return final;
        }

        /// <summary>
        /// Finds the completion time of a processes from a list of
processes
        /// </summary>
        /// <param name="processes">The list of processes</param>
        /// <param name="index">The index of the processes in the
list</param>
        /// <returns>The completion time of a processes</returns>
        public static TimeSpan FindCompletionTime(Processes processes,
int index)
        {
            TimeSpan completionTime = processes[0].ArriveTime;
            int i = 0;
            while (i <= index)
            {
                var arrive = processes[i].ArriveTime;
                if (arrive > completionTime)
                    completionTime += arrive - completionTime;
                else
                {
                    completionTime += processes[i].CPUTime;
                    i++;
                }
            }
            return completionTime;
        }

        /// <summary>
        /// Calculates the average waiting and turn around time for
non-preemtive algorithms
        /// </summary>
        /// <param name="processes">the scheduled processes</param>
        private static void CalculateNonPreAverageTime(Processes
processes)
        {
            var avgTrTime = TimeSpan.Zero;
```

```
            var avgWtTime = TimeSpan.Zero;
            for (int i = 0; i < processes.Count; i++)
            {
                var ct = FindCompletionTime(processes, i);
                var arrival = processes[i].ArriveTime;
                var cpuTime = processes[i].CPUTime;

                avgTrTime += ct - arrival;
                avgWtTime += ct - arrival - cpuTime;
            }

            avgTrTime /= processes.Count;
            avgWtTime /= processes.Count;

            processes.AverageTurnAroundTime = avgTrTime;
            processes.AverageWaitingTime = avgWtTime;
        }

        /// <summary>
        /// Check completion time rich critical time or not for non-
preemtive algorithms
        /// </summary>
        /// <param name="processes"></param>
        private static void CalculateNonPreCriticalTime(Processes
processes)
        {
            for (int i = 0; i < Processes.Count; i++)
            {
                var ct = FindCompletionTime(processes, i);
                if (ct > processes[i].CriticalTime)
                    Console.WriteLine("Process" + i);
            }
        }

        /// <summary>
        /// Calculates the average waiting and turn around time for
preemtive algorithms
        /// </summary>
        /// <param name="processes">the scheduled processes</param>
        private static void CalculatePreAverageTime(Processes
processes)
        {
            // Calculating Turn around time and waiting time
            var avgTrTime = TimeSpan.Zero;
            var avgWtTime = TimeSpan.Zero;
            for (int j = 0; j < Processes.Count; j++)
            {
                var lastExecution = processes.FindLast(p => p.PID ==
Processes[j].PID);
```

```
                var ct = lastExecution.ArriveTime +
lastExecution.CPUTime;
                var arrival = Processes[j].ArriveTime;
                var cpuTime = Processes[j].CPUTime;

                avgTrTime += ct - arrival;
                avgWtTime += ct - arrival - cpuTime;
            }

            avgTrTime /= Processes.Count;
            avgWtTime /= Processes.Count;

            processes.AverageTurnAroundTime = avgTrTime;
            processes.AverageWaitingTime = avgWtTime;
        }

        /// <summary>
        /// Check completion time rich critical time or not for
preemtive algorithms
        /// </summary>
        /// <param name="processes"></param>
        private static void CalculatePreCriticalTime(Processes
processes)
        {
            for (int i = 0; i < Processes.Count; i++)
            {
                var lastExecution = processes.FindLast(p => p.PID ==
Processes[i].PID);
                var ct = lastExecution.ArriveTime +
lastExecution.CPUTime;
                if (ct > processes[i].CriticalTime)
                    Console.WriteLine("Process" + i);
            }
        }
    }
}
```

Листинг переключения ProcessPriority.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace Scheduling
{
    [System.Serializable]
    public enum ProcessPriority
    {
        Highest,//0
        AboveNormal,//1
        Normal,//2
        BelowNormal,//3
```

```
        Lowest,//4
    }
}
```

Листинг класса который во втором приложении отображает опоздания процессов и свободное время ProcessControl.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System.Globalization;

public class ProcessControl : MonoBehaviour
{
    [Header("Input Main Data GUI")]
    public float Ca = 70.0f;
    public float Cb = 175.0f;
    public float Ta = 140.0f;
    public float Tb = 350.0f;
    public float P = 70.0f;

    public InputField CaInput;
    public InputField CbInput;
    public InputField TaInput;
    public InputField TbInput;
    public InputField PInput;

    public Scheduling.Algorithm algSelector =
Scheduling.Algorithm.ROUND_ROBIN;
    public Dropdown algo;

    [Header("Input Process Data GUI")]
    public int ACountProcess = 4;
    public InputField CountProcessAInput;
    public Dropdown aPriority;
    public Scheduling.ProcessPriority priorityAProcess =
Scheduling.ProcessPriority.AboveNormal;
    public int BCountProcess = 3;
    public InputField CountProcessBInput;
    public Dropdown bPriority;
    public Scheduling.ProcessPriority priorityBProcess =
Scheduling.ProcessPriority.Normal;

    [Header("Filled Process Data")]
    public List<Scheduling.Process> processData;

    [Header("")]
    public SchedulingAlgorithms alg;

    public Text avgWaitText;
    public Text avgTurnText;
```

```
public Text freeTimeText;
public Text freeTimesPercentText;

private int CountProcesss;
private float quant = 0.0f;

public void SetData()
{
    if (float.TryParse(CaInput.text, out Ca))
    {
        Debug.Log("Sucsess input: " + CaInput.text);
    }
    else
    {
        Debug.Log("No Sucsess input: " + CaInput.text);
    }

    if (float.TryParse(CbInput.text, out Cb))
    {
        Debug.Log("Sucsess input: " + CbInput.text);
    }
    else
    {
        Debug.Log("No Sucsess input: " + CbInput.text);
    }

    if (float.TryParse(TaInput.text, out Ta))
    {
        Debug.Log("Sucsess input: " + TaInput.text);
    }
    else
    {
        Debug.Log("No Sucsess input: " + TaInput.text);
    }

    if (float.TryParse(TbInput.text, out Tb))
    {
        Debug.Log("Sucsess input: " + TbInput.text);
    }
    else
    {
        Debug.Log("No Sucsess input: " + TbInput.text);
    }

    if (float.TryParse(PInput.text, out P))
    {
        Debug.Log("Sucsess input: " + PInput.text);
    }
    else
    {
        Debug.Log("No Sucsess input: " + PInput.text);
```

```
        }

        if (int.TryParse(CountProcessAInput.text, out ACountProcess))
        {
            Debug.Log("Sucsess input: " + CountProcessAInput.text);
        }
        else
        {
            Debug.Log("No Sucsess input: " + CountProcessAInput.text);
        }

        if (int.TryParse(CountProcessBInput.text, out BCountProcess))
        {
            Debug.Log("Sucsess input: " + CountProcessBInput.text);
        }
        else
        {
            Debug.Log("No Sucsess input: " + CountProcessBInput.text);
        }

        switch (aPriority.value)
        {
            case 0:
                priorityAProcess = Scheduling.ProcessPriority.Highest;
                break;
            case 1:
                priorityAProcess =
Scheduling.ProcessPriority.AboveNormal;
                break;
            case 2:
                priorityAProcess = Scheduling.ProcessPriority.Normal;
                break;
            case 3:
                priorityAProcess =
Scheduling.ProcessPriority.BelowNormal;
                break;
            case 4:
                priorityAProcess = Scheduling.ProcessPriority.Lowest;
                break;
        }

        switch (bPriority.value)
        {
            case 0:
                priorityBProcess = Scheduling.ProcessPriority.Highest;
                break;
            case 1:
                priorityBProcess =
Scheduling.ProcessPriority.AboveNormal;
                break;
            case 2:
```

```
                priorityBProcess = Scheduling.ProcessPriority.Normal;
                break;
            case 3:
                priorityBProcess =
Scheduling.ProcessPriority.BelowNormal;
                break;
            case 4:
                priorityBProcess = Scheduling.ProcessPriority.Lowest;
                break;
        }

        switch (algo.value)
        {
            case 0:
                algSelector = Scheduling.Algorithm.FCFS;
                break;
            case 1:
                algSelector = Scheduling.Algorithm.SJF_PREEMPTIVE;
                break;
            case 2:
                algSelector = Scheduling.Algorithm.SJF_NONPREEMPTIVE;
                break;
            case 3:
                algSelector =
Scheduling.Algorithm.PRIORITY_PREEMPTIVE;
                break;
            case 4:
                algSelector =
Scheduling.Algorithm.PRIORITY_NONPREEMPTIVE;
                break;
            case 5:
                algSelector = Scheduling.Algorithm.ROUND_ROBIN;
                break;
        }

        CountProcesss = ACountProcess + BCountProcess;
        if (algSelector == Scheduling.Algorithm.ROUND_ROBIN)
            quant = P / 2.0f; // quant for Round Robin alg

        var count = 0.0f;
        var count2 = 0.0f;
        for (int i = 0; i < CountProcesss; i++)
        {
            //Fill by A process
            if (i <= ACountProcess)
            {
                Scheduling.Process data = new Scheduling.Process();
                data.PID = i;
                data.ArriveTime = count;
                count += Ta;
                data.CPUTime = Ca;
```

```
                data.CriticalTime = count;
                data.Priority = (int)priorityAProcess;
                processData.Insert(i, data);
            }
            //Fill by B process
            else if (i > ACountProcess)
            {
                Scheduling.Process data = new Scheduling.Process();
                data.PID = i;
                data.ArriveTime = count2;
                count2 += Tb;
                data.CPUTime = Cb;
                data.CriticalTime = count2;
                data.Priority = (int)priorityBProcess;
                processData.Insert(i, data);
            }
        }

        switch (algSelector)
        {
            case Scheduling.Algorithm.FCFS:
                alg.Initialize();
                alg.ScheduleFirstComeFirstServed(processData);
                break;
            case Scheduling.Algorithm.SJF_PREEMPTIVE:
                alg.Initialize();
                alg.ScheduleShortestJobFirstPremprive(processData);
                break;
            case Scheduling.Algorithm.SJF_NONPREEMPTIVE:
                alg.Initialize();
                alg.ScheduleShortestJobFirstNonPremptive(processData);
                break;
            case Scheduling.Algorithm.PRIORITY_PREEMPTIVE:
                alg.Initialize();
                alg.SchedulePriorityPremprive(processData);
                break;
            case Scheduling.Algorithm.PRIORITY_NONPREEMPTIVE:
                alg.Initialize();
                alg.ScheduleShortestJobFirstNonPremptive(processData);
                break;
            case Scheduling.Algorithm.ROUND_ROBIN:
                alg.Initialize();
                alg.ScheduleRoundRobin(processData, quant);
                break;
        }
        avgWaitText.text = alg.avg_waiting.ToString();
        avgTurnText.text = alg.avg_turnaround.ToString();
        freeTimeText.text = alg.freeTimes.ToString();
        freeTimesPercentText.text = alg.freeTimesPercent.ToString();
    }
```

```
    public void ClearData()
    {
        Ca = 0.0f;
        Cb = 0.0f;
        Ta = 0.0f;
        Tb = 0.0f;
        P = 0.0f;
        algSelector = Scheduling.Algorithm.FCFS;
        ACountProcess = 0;
        BCountProcess = 0;
        priorityAProcess = Scheduling.ProcessPriority.Highest;
        priorityBProcess = Scheduling.ProcessPriority.Highest;
        processData.Clear();
        avgWaitText.text = 0.0f.ToString();
        alg.avg_waiting = 0.0f;
        alg.avg_turnaround = 0.0f;
        alg.freeTimes = 0.0f;
        alg.freeTimesPercent= 0.0f;
        alg.lastValid = 0;
        alg.debugIdlesTime = null;
        alg.debugIdlesTime2 = null;
        avgTurnText.text = 0.0f.ToString();
        freeTimeText.text = 0.0f.ToString();
        freeTimesPercentText.text = 0.0f.ToString();
    }
}
```

Листинг класса который во втором приложении обрабатывает диаграммы процессов SchedulingAlgorithms.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System.Linq;

public class SchedulingAlgorithms : MonoBehaviour
{
    //outputs
    public Text logerOverCriticalTime;
    public float[] waitingTime;
    public float[] turnaroundTime;
    public float[] finalTime;
    public float[] start;
    public float[] end;
    public string[] debugIdlesTime;
    public string[] debugIdlesTime2;
    public int[] proc;

    public float avg_waiting;
    public float avg_turnaround;
    public float freeTimes;
```

```
    public float freeTimesPercent;
    public int lastValid;

    public void Initialize()
    {
        waitingTime = new float[2000];
        turnaroundTime = new float[2000];
        start = new float[2000];
        finalTime = new float[2000];
        end = new float[2000];
        debugIdlesTime = new string[2000];
        debugIdlesTime2 = new string[2000];
        proc = new int[2000];
        avg_waiting = 0.0f;
        freeTimes = 0.0f;
        freeTimesPercent = 0.0f;
        avg_turnaround = 0.0f;
        lastValid = 0;

        for (int i = 0; i < 2000; i++)
        {
            waitingTime[i] = 0; turnaroundTime[i] = 0; start[i] = 0;
end[i] = 0; proc[i] = -1; finalTime[i] = 0; debugIdlesTime[i] = "";
debugIdlesTime2[i] = "";
        }
    }

    private void Awake()
    {
        Initialize();
    }

    private void SortArrivalsTime(List<Scheduling.Process> l)
    {
        for (int i = 0; i < l.Count; i++)
        {
            for (int j = 0; j < l.Count; j++)
            {
                if (l[i].ArriveTime < l[j].ArriveTime)
                {
                    Scheduling.Process temp = l[i];
                    l[i] = l[j];
                    l[j] = temp;
                }
            }
        }
    }

    private void SortPrioritys(List<Scheduling.Process> l)
    {
        for (int i = 0; i < l.Count; i++)
```

```
        {
            for (int j = 0; j < l.Count; j++)
            {
                if (l[i].Priority < l[j].Priority)
                {
                    Scheduling.Process temp = l[i];
                    l[i] = l[j];
                    l[j] = temp;
                }
            }
        }
    }
    private void SortBurstsTimes(List<Scheduling.Process> l)
    {
        for (int i = 0; i < l.Count; i++)
        {
            for (int j = 0; j < l.Count; j++)
            {
                if (l[i].CPUTime < l[j].CPUTime)
                {
                    Scheduling.Process temp = l[i];
                    l[i] = l[j];
                    l[j] = temp;
                }
            }
        }

    }

    private void SortCriticalsTimes(List<Scheduling.Process> l)
    {
        for (int i = 0; i < l.Count; i++)
        {
            for (int j = 0; j < l.Count; j++)
            {
                if (l[i].CriticalTime < l[j].CriticalTime)
                {
                    Scheduling.Process temp = l[i];
                    l[i] = l[j];
                    l[j] = temp;
                }
            }
        }

    }

    public void ScheduleFirstComeFirstServed(List<Scheduling.Process>
processData)
    {
        List<Scheduling.Process> p = new
List<Scheduling.Process>(processData.Count);
```

```
        for (int i = 0; i < processData.Count; i++)
        {
            p.Insert(i, processData[i]);
        }

        SortArrivalsTime(p);

        float cnt = 0; int idx = 0;
        for (int i = 0; i < processData.Count; i++)
        {
            if (p[i].ArriveTime <= cnt)
            {
                start[idx] = cnt;
                end[idx] = cnt + p[i].CPUTime;
                proc[idx] = p[i].PID;
                waitingTime[i] = start[idx] - p[i].ArriveTime;
                turnaroundTime[i] = end[idx] - p[i].ArriveTime;
                cnt += p[i].CPUTime;
                finalTime[i] = cnt;
                if (cnt > processData[i].CriticalTime)
                {
                    Debug.Log("Процесс: " + p[i].PID + " Его финальное
время" + cnt);
                    debugIdlesTime[i] = "Процесс: " + p[i].PID + " Его
финальное время" + cnt;
                }
                idx++;
            }
            else
            {
                start[idx] = cnt;
                end[idx] = p[i].ArriveTime;
                proc[idx] = -1;
                freeTimes += end[idx] - start[idx];
                cnt = p[i].ArriveTime;
                idx++;
                start[idx] = cnt;
                end[idx] = cnt + p[i].CPUTime;
                proc[idx] = p[i].PID;
                waitingTime[i] = start[idx] - p[i].ArriveTime;
                turnaroundTime[i] = end[idx] - p[i].ArriveTime;
                cnt += p[i].CPUTime;
                idx++;
            }
            freeTimesPercent = (freeTimes / finalTime[i]) * 100;
        }

        for (int i = 0; i < processData.Count; i++)
        {
            if (waitingTime[i] == 0) continue;
```

```
            else
            {
                avg_waiting += waitingTime[i];
            }
        }
        avg_waiting /= processData.Count;
        for (int i = 0; i < processData.Count; i++)
        {
            if (turnaroundTime[i] == 0) continue;
            else
            {
                avg_turnaround += turnaroundTime[i];
            }
        }
        avg_turnaround /= processData.Count;
        lastValid = idx;
    }

    public void ScheduleRoundRobin(List<Scheduling.Process>
processData, float mQuantum)
    {
        List<Scheduling.Process> p = new
List<Scheduling.Process>(processData.Count);

        for (int i = 0; i < processData.Count; i++)
        {
            p.Insert(i, processData[i]);
        }

        SortArrivalsTime(p);

        float cnt = 0; int idx = 0;
        while (p.Count != 0)
        {
            List<Scheduling.Process> ready = new
List<Scheduling.Process>(processData.Count);
            if (p.First().ArriveTime > cnt)
            {
                start[idx] = cnt;
                end[idx] = p.First().ArriveTime;
                //Free time output
                proc[idx] = -1;
                freeTimes += end[idx] - start[idx];
                idx++;
                cnt = p.First().ArriveTime;
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
```

```
                    p.RemoveAt(j);
                }
                else j++;

            }
            while (ready.Count != 0)
            {
                Scheduling.Process readyProcess = ready[0];
                ready.RemoveAt(0);
                if (mQuantum < readyProcess.CPUTime)
                {
                    start[idx] = cnt;
                    end[idx] = cnt + mQuantum;
                    proc[idx] = readyProcess.PID;
                    cnt += mQuantum;
                    finalTime[idx] = cnt;
                    readyProcess.CPUTime -= mQuantum;
                    ready.Add(readyProcess);
                    idx++;
                }
                else
                {
                    start[idx] = cnt;
                    end[idx] = cnt + readyProcess.CPUTime;
                    proc[idx] = readyProcess.PID;
                    cnt += readyProcess.CPUTime;
                    finalTime[idx] = cnt;

                    idx++;
                }
                for (int i = 0; i < processData.Count; i++)
                {
                    if (cnt > readyProcess.CriticalTime)
                    {
                        Debug.Log("Процесс: " + readyProcess.PID + "
Его финальное время" + cnt);
                        debugIdlesTime[i] = "Процесс: " +
readyProcess.PID + " Его финальное время" + cnt;
                    }
                }

                if (cnt > readyProcess.CriticalTime)
                {
                    Debug.Log("Процесс: " + readyProcess.PID + " Его
финальное время" + cnt);
                    debugIdlesTime[idx] = "Процесс: " +
readyProcess.PID + " Его финальное время" + cnt;
                    logerOverCriticalTime.text = debugIdlesTime[idx];
                }
                freeTimesPercent = (freeTimes / cnt) * 100;
```

```
                for (int i = 0, j = 0; j < p.Count; i++)
                {
                    if (p[j].ArriveTime <= cnt)
                    {
                        ready.Add(p[j]);
                        p.RemoveAt(j);
                    }
                    else j++;
                }
            }
        }

        for (int j = 0; j < processData.Count; j++)
        {
            bool k = false; float wait = 0;
            for (int i = 0; i <= idx; i++)
            {
                if (proc[i] == j)
                {
                    if (!k)
                    {
                        waitingTime[j] = start[i] -
processData[j].ArriveTime; k = true;
                        wait = end[i];

                    }
                    else if (k)
                    {
                        waitingTime[j] += start[i] - wait;
                        wait = end[i];
                    }
                    turnaroundTime[j] = wait -
processData[j].ArriveTime;
                }
            }
        }
        for (int i = 0; i < processData.Count; i++)
        {
            if (waitingTime[i] == 0) continue;
            else
            {
                avg_waiting += waitingTime[i];
            }
        }
        avg_waiting /= processData.Count;
        for (int i = 0; i < processData.Count; i++)
        {
            if (turnaroundTime[i] == 0) continue;
            else
            {
                avg_turnaround += turnaroundTime[i];
```

```
            }
        }
        avg_turnaround /= processData.Count;
        lastValid = idx;

    }

    public void SchedulePriorityNonPremptive(List<Scheduling.Process>
processData)
    {
        List<Scheduling.Process> p = new
List<Scheduling.Process>(processData.Count);

        for (int i = 0; i < processData.Count; i++)
        {
            p.Insert(i, processData[i]);
        }

        SortArrivalsTime(p);

        float cnt = 0; int idx = 0;

        while (p.Count != 0)
        {
            List<Scheduling.Process> ready = new
List<Scheduling.Process>(processData.Count);
            if (p.First().ArriveTime > cnt)
            {
                start[idx] = cnt;
                end[idx] = p.First().ArriveTime;
                //Free time output
                proc[idx] = -1;
                freeTimes += end[idx] - start[idx];
                idx++;
                cnt = p.First().ArriveTime;
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
            }
            while (ready.Count != 0)
            {
                SortPrioritys(ready);
                Scheduling.Process readyProcess = ready[0];
                ready.RemoveAt(0);
                start[idx] = cnt;
```

```
                end[idx] = cnt + readyProcess.CPUTime;

                proc[idx] = readyProcess.PID;
                waitingTime[idx] = start[idx] -
readyProcess.ArriveTime;
                turnaroundTime[idx] = end[idx] -
readyProcess.ArriveTime;
                idx++;
                cnt += readyProcess.CPUTime;
                finalTime[idx] = cnt;
                if (cnt > readyProcess.CriticalTime)
                {
                    Debug.Log("Процесс: " + readyProcess.PID + " Его
финальное время" + cnt);
                    debugIdlesTime[idx] = "Процесс: " +
readyProcess.PID + " Его финальное время" + cnt;
                    logerOverCriticalTime.text = debugIdlesTime[idx];
                }

                freeTimesPercent = (freeTimes / cnt) * 100;

                for (int i = 0, j = 0; i < p.Count; i++)
                {
                    if (p[j].ArriveTime <= cnt)
                    {
                        ready.Add(p[j]);
                        p.RemoveAt(j);
                    }
                    else j++;
                }
            }

        }
        for (int i = 0; i < processData.Count; i++)
        {
            if (waitingTime[i] == 0) continue;
            else
            {
                avg_waiting += waitingTime[i];
            }
        }
        avg_waiting /= processData.Count;
        for (int i = 0; i < processData.Count; i++)
        {
            if (turnaroundTime[i] == 0) continue;
            else
            {
                avg_turnaround += turnaroundTime[i];
            }
        }
        avg_turnaround /= processData.Count;
```

```
        lastValid = idx;
    }

    public void SchedulePriorityPremprive(List<Scheduling.Process>
processData)
    {
        List<Scheduling.Process> p = new
List<Scheduling.Process>(processData.Count);
        for (int i = 0; i < processData.Count; i++)
        {
            p.Insert(i, processData[i]);
        }
        SortArrivalsTime(p);

        float cnt = 0;
        int idx = -1;
        if (p[0].ArriveTime > cnt)
        {
            idx = 0;
        }
        while (p.Count != 0)
        {
            List<Scheduling.Process> ready = new
List<Scheduling.Process>(processData.Count);
            if (p.First().ArriveTime > cnt)
            {
                idx++;
                start[idx] = cnt;
                end[idx] = p.First().ArriveTime;
                proc[idx] = -1;
                freeTimes += end[idx] - start[idx];
                idx++;
                cnt = p.First().ArriveTime;
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
            }
            int cur = -2;
            while (ready.Count != 0)
            {
                SortPrioritys(ready);
                Scheduling.Process readyProcess = ready[0];
                ready.RemoveAt(0);
                if (cur != readyProcess.PID)
                {
```

```
            idx++;
            start[idx] = cnt;
            proc[idx] = readyProcess.PID;

            readyProcess.CPUTime--;
            cur = readyProcess.PID;
            cnt++;
            end[idx] = cnt;
            finalTime[idx] = cnt;

            if (readyProcess.CPUTime > 0)
            {
                ready.Add(readyProcess);
            }
            else
            {
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
            }
        }
        else
        {
            readyProcess.CPUTime--;
            cur = readyProcess.PID;
            cnt++;
            end[idx] = cnt;
            finalTime[idx] = cnt;

            if (readyProcess.CPUTime > 0)
            {
                ready.Add(readyProcess);
            }
            else
            {

            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
```

```
                    }
                }
                if (cnt > readyProcess.CriticalTime)
                {
                    Debug.Log("Процесс: " + readyProcess.PID + " Его
финальное время" + cnt);
                    debugIdlesTime[idx] = "Процесс: " +
readyProcess.PID + " Его финальное время" + cnt;
                    logerOverCriticalTime.text = debugIdlesTime[idx];
                }
                freeTimesPercent = (freeTimes / finalTime[idx]) * 100;
            }
        }
        for (int j = 0; j < processData.Count; j++)
        {
            bool k = false; float wait = 0;
            for (int i = 0; i <= idx; i++)
            {
                if (proc[i] == j)
                {
                    if (!k)
                    {
                        waitingTime[j] = start[i] -
processData[j].ArriveTime; k = true;
                        wait = end[i];
                    }
                    else if (k)
                    {
                        waitingTime[j] += start[i] - wait;
                        wait = end[i];
                    }
                    turnaroundTime[j] = wait -
processData[j].ArriveTime;
                }
            }
        }
        for (int i = 0; i < processData.Count; i++)
        {
            if (waitingTime[i] == 0) continue;
            else
            {
                avg_waiting += waitingTime[i];
            }
        }
        avg_waiting /= processData.Count;
        for (int i = 0; i < processData.Count; i++)
        {
            if (turnaroundTime[i] == 0) continue;
            else
            {
                avg_turnaround += turnaroundTime[i];
```

```
            }
        }
        avg_turnaround /= processData.Count;
        lastValid = idx + 1;
    }

    public void
ScheduleShortestJobFirstNonPremptive(List<Scheduling.Process>
processData)
    {
        List<Scheduling.Process> p = new
List<Scheduling.Process>(processData.Count);

        for (int i = 0; i < processData.Count; i++)
        {
            p.Insert(i, processData[i]);
        }

        SortArrivalsTime(p);

        float cnt = 0; int idx = 0;

        while (p.Count != 0)
        {
            List<Scheduling.Process> ready = new
List<Scheduling.Process>(processData.Count);
            if (p.First().ArriveTime > cnt)
            {
                start[idx] = cnt;
                end[idx] = p.First().ArriveTime;
                //Free time output
                proc[idx] = -1;
                freeTimes += end[idx] - start[idx];
                idx++;
                cnt = p.First().ArriveTime;
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
            }
            while (ready.Count != 0)
            {
                SortBurstsTimes(ready);
                Scheduling.Process readyProcess = ready[0];
                ready.RemoveAt(0);
                start[idx] = cnt;
```

```
                end[idx] = cnt + readyProcess.CPUTime;
                proc[idx] = readyProcess.PID;
                waitingTime[idx] = start[idx] -
readyProcess.ArriveTime;
                turnaroundTime[idx] = end[idx] -
readyProcess.ArriveTime;
                idx++;
                cnt += readyProcess.CPUTime;
                finalTime[idx] = cnt;
                if (cnt > readyProcess.CriticalTime)
                {
                    Debug.Log("Процесс: " + readyProcess.PID + " Его
финальное время" + cnt);
                    debugIdlesTime[idx] = "Процесс: " +
readyProcess.PID + " Его финальное время" + cnt;
                    logerOverCriticalTime.text = debugIdlesTime[idx];
                }
                freeTimesPercent = (freeTimes / cnt) * 100;

                for (int i = 0, j = 0; i < p.Count; i++)
                {
                    if (p[j].ArriveTime <= cnt)
                    {
                        ready.Add(p[j]);
                        p.RemoveAt(j);
                    }
                    else j++;
                }
            }

        }
        for (int i = 0; i < processData.Count; i++)
        {
            if (waitingTime[i] == 0) continue;
            else
            {
                avg_waiting += waitingTime[i];
            }
        }
        avg_waiting /= processData.Count;
        for (int i = 0; i < processData.Count; i++)
        {
            if (turnaroundTime[i] == 0) continue;
            else
            {
                avg_turnaround += turnaroundTime[i];
            }
        }
        avg_turnaround /= processData.Count;
        lastValid = idx;
    }
```

```
    public void
ScheduleShortestJobFirstPremprive(List<Scheduling.Process>
processData)
    {
        List<Scheduling.Process> p = new
List<Scheduling.Process>(processData.Count);

        for (int i = 0; i < processData.Count; i++)
        {
            p.Insert(i, processData[i]);
        }

        SortArrivalsTime(p);
        float cnt = 0;
        int idx = -1;
        if (p[0].ArriveTime > cnt)
        {
            idx = 0;
        }
        while (p.Count != 0)
        {
            List<Scheduling.Process> ready = new
List<Scheduling.Process>(processData.Count);
            if (p.First().ArriveTime > cnt)
            {
                start[idx] = cnt;
                end[idx] = p.First().ArriveTime;
                proc[idx] = -1;
                freeTimes += end[idx] - start[idx];
                idx++;
                cnt = p.First().ArriveTime;
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
            }
            int cur = -2;
            while (ready.Count != 0)
            {
                SortBurstsTimes(ready);
                Scheduling.Process readyProcess = ready[0];
                ready.RemoveAt(0);
                if (cur != readyProcess.PID)
                {
                    idx++;
```

```
            start[idx] = cnt;
            proc[idx] = readyProcess.PID;

            readyProcess.CPUTime--;
            cur = readyProcess.PID;
            cnt++;
            end[idx] = cnt;
            finalTime[idx] = cnt;

            if (readyProcess.CPUTime > 0)
            {
                ready.Add(readyProcess);
            }
            else
            {
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
            }
        }
        else
        {
            readyProcess.CPUTime--;
            cur = readyProcess.PID;
            cnt++;
            end[idx] = cnt;
            finalTime[idx] = cnt;

            if (readyProcess.CPUTime > 0)
            {
                ready.Add(readyProcess);
            }
            else
            {
            }
            for (int i = 0, j = 0; j < p.Count; i++)
            {
                if (p[j].ArriveTime <= cnt)
                {
                    ready.Add(p[j]);
                    p.RemoveAt(j);
                }
                else j++;
            }
        }
```

```
                if (cnt > readyProcess.CriticalTime)
                {
                    Debug.Log("Процесс: " + readyProcess.PID + " Его
финальное время" + cnt);
                    debugIdlesTime[idx] = "Процесс: " +
readyProcess.PID + " Его финальное время" + cnt;
                    logerOverCriticalTime.text = debugIdlesTime[idx];
                }
                freeTimesPercent = (freeTimes / cnt) * 100;
            }
        }
        for (int j = 0; j < processData.Count; j++)
        {
            bool k = false; float wait = 0;
            for (int i = 0; i <= idx; i++)
            {
                if (proc[i] == j)
                {
                    if (!k)
                    {
                        waitingTime[j] = start[i] -
processData[j].ArriveTime; k = true;
                        wait = end[i];

                    }
                    else if (k)
                    {
                        waitingTime[j] += start[i] - wait;
                        wait = end[i];
                    }
                    turnaroundTime[j] = wait -
processData[j].ArriveTime;
                }
            }
        }
        for (int i = 0; i < processData.Count; i++)
        {
            if (waitingTime[i] == 0) continue;
            else
            {
                avg_waiting += waitingTime[i];
            }
        }
        avg_waiting /= processData.Count;
        for (int i = 0; i < processData.Count; i++)
        {
            if (turnaroundTime[i] == 0) continue;
            else
            {
                avg_turnaround += turnaroundTime[i];
            }
```

```
        }
        avg_turnaround /= processData.Count;
        lastValid = idx + 1;
    }
}
```

ПРИЛОЖЕНИЕ Б

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 В

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();
        }
    }
}
```