Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина операционные среды и системное программирование

ОТЧЁТ

к лабораторной работе №4

на тему

**Взаимодействие процессов (потоков): взаимное**

**исключение и синхронизация**

Выполнил: студент гр.253504 Божко Я.Д

Проверил: ассистент кафедры информатики

Гриценко Н.Ю

Минск 2024

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc21088)

[1 Теоретические сведения 4](#_Toc29053)

[1.1 Взаимодействие процессов 4](#_Toc10824)

[1.2 Изоляция процессов 5](#_Toc6214)

[1.3 Решения для межпроцессного взаимодействия (IPC) 6](#_Toc25052)

[1.4 ISO 12](#_Toc17217)

[2 Инструментальная языковая среда 15](#_Toc11381)

[3 Работа приложения 16](#_Toc27628)

[3.1 Структура приложения 16](#_Toc19395)

[3.2 Функции приложения 16](#_Toc10803)

[3.3 Анализ данных 16](#_Toc19191)

[Список использованных источников 18](#_Toc21158)

[Приложение А (обязательное) Реализация программы на языке С++ 19](#_Toc25367)

**ВВЕДЕНИЕ**

Цель работы: Приложение, демонстрирующее параллельную согласованную работу процессов (потоков) и их взаимодействие. Анализ корректности (отсутствия коллизий). Оценка эффективности механизмов синхронизации (ISO).

Реализация модели взаимодействия конкурирующих параллельных процессов (потоков) «обедающие философы» с возможностью ее параметризации и наглядного (не обязательно графического) представления результатов.

Проверка и демонстрация различных подходов к решению задачи обеспечения корректного взаимодействия. Изменяемые параметры модели: количество «философов»; выбор логики разрешения конфликта; интенсивность обращений к критическому ресурсу, длительность использования ресурса и т.п. (характеристики случайных величин при моделировании); величина тайм-аутов и др.

Взаимодействие заключается в передаче данных между процессами или совместном использовании некоторых ресурсов и обычно реализуется с помощью таких механизмов как транспортеры, очереди, сигналы, семафоры.

Модель OSI (Open Systems Interconnection) – взаимодействие открытых систем – семиуровневая модель протоколов передачи данных, разработанная Международной организацией по стандартизации (см . – “ISO ”) и CCITT (Consultative Committee for International Telephony and Telegraphy ) для сопряжения различных видов вычислительного и коммуникационного оборудования различных производителей.

Основная цель внедрения стандартов ISO заключается в повышении качества и безопасности бизнеса и технологий, что в свою очередь влечёт за собой рост потребительских характеристик выпускаемой продукции. Одновременно улучшается взаимодействие между покупателем и поставщиком, производителем и посредником, придавая рыночным отношениям цивилизованный взаимовыгодный характер.

**1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**1.1 Взаимодействие процессов**

Взаимодействие заключается в передаче данных между процессами или совместном использовании некоторых ресурсов и обычно реализуется с помощью таких механизмов как транспортеры, очереди, сигналы, семафоры.

**1.1.1** Транспортеры (каналы)

Являются средством взаимодействия родственных процессов, представляют собой область памяти, имеющую файловую организацию, для которой обеспечивается запись и считывание данных в транспортере. Реализуется очередь обслуживания. Порядок записи данных на транспортер неизменен, не допускается повторное считывание данных. Обмен данными происходит не непосредственно а через транспортер. Из вызвавшего процесса задается размер транспортера. Дочерние процессы могут использовать родительский транспортер.

**1.1.2** Очереди

Эти механизмы могут обеспечить передачу или использование общих данных без перемещения данных, а с передачей элемента очереди, содержащего указатель данных, и объем массива данных. Очередь используется вместе с механизмом общей памяти. Элемент очереди может быть считан с уничтожением или без уничтожения этого элемента. Чтение элемента может осуществляться в соответствии с механизмом очереди или стека. Чтение элементов очереди осуществляет только создающий очереди процесс, все другие процессы могут только записать элементы в очередь. Создающий процесс может выполнить следующие действия над очередью: создание просмотр, чтение, закрытие. Записывающий процесс может осуществлять действия: открыть, записать, закрыть.

Имя очереди, которое присваивается создающим процессом, имеет вид полной спецификации файла. Ожидание элементов в очереди организуется с помощью семафора, сигнализирующего о записи элемента в очередь. Для работы с очередью определены такие дополнительные функции:

1. Определение количества элементов в очереди в текущий момент.
2. Очистка очереди создавшим его процессом.

**1.1.3** Сигналы

Сигнал является механизмом передачи требования от одного процесса к другому на немедленное выполнение действия. Обработчик сигнала создается процессом и помещается в начале первого потока процесса. Является аналогом обработки прерывания. При передачи управления обработчику передается адрес возврата и тип принятого сигнала. Процесс посылающий сигнал типа «флаг», может передать дополнительную информацию обработчику сигнала. Характер выполняемых действий при возникновении сигнала: обработка системной ошибки при появлении сигнала, блокирование сигнала, передача управления подпрограмме.

**1.1.4** Семафоры

Являются механизмами передачи сообщений от одного потока к другому о наступлении некоторого события. Различают семафоры системные и оперативной памяти. Семафоры оперативной памяти – двойное слово в памяти системы, его описатель – адрес места в памяти. Такие семафоры не создаются и не открываются, а устанавливаются в определенное состояние. Процессы, использующие семафоры оперативной памяти, должны иметь доступ к соответствующему сегменту памяти. ОС такие семафоры не обслуживает и не сообщает об их освобождении или захвате. ОС контролирует завершение каждого процесса, владеющего системным семафором, и освобождает его для процессов.

Управление семафором реализуется с помощью функция:

1. Установка семафора с целью сигнализации.
2. Ожидание вызывающим потоком, пока семафор не будет выключен.
3. Ожидания потоком выключения одного из нескольких семафоров.[1]

Если семафор освобождается всеми использующими его процессами то он удаляется из системы.

**1.2 Изоляция процессов**

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является изоляция одного процесса от другого. Для этого операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

В операционной системе наряду с процессами нужен другой механизм распараллеливания вычислений, который учитывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС предлагают механизм многопоточной обработки (multithreading). При этом вводится новая единица работы – поток выполнения, а понятие «процесс» в значительной степени меняет смысл.

ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

В отличие от процессов, которые принадлежат разным конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, нежели процессы в традиционной мультипрограммной системе. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Потоки разных процессов хорошо защищены друг от друга.

Мультипрограммирование более эффективно на уровне потоков, а не процессов. Каждый поток имеет собственный счетчик команд и стек. Задача, оформленная в виде нескольких потоков в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей.

Вычислительные процессы (задачи) по наличию и активности взаимодействия между ними:

1. не связанны – выполняются независимо, возможны лишь неявные коллизии при использовании общесистемных ресурсов; удобны для распараллеливания;
2. слабо связанные – к общим ресурсам обращаются сравнительно редко; достаточно удобны для распараллеливания, в т.ч. на многомашинных конфигурациях при условии наличия быстрых интерфейсов обмена данными;
3. сильно связанные – интенсивно обращаются к общим ресурсам, что приводит к частым коллизиям; распараллеливание возможно на многопроцессорных (многоядерных) конфигурациях, но проблематично;
4. не поддающиеся распараллеливанию.

Проблема обмена данными вытекает из концепции изолированных адресных пространств – процессы не имеют возможности напрямую обращаться «внутрь» друг друга. Потоки одного процесса действуют в едином адресном пространстве, но в общем случае во взаимодействии могут участвовать потоки разных процессов.

Решением всех задач становится использование специальных системных объектов – объектов IPC (Inter-Process Communication) и, как частный случай, ISO (Inter-process Synchronization Objects).[2]

**1.3 Решения для межпроцессного взаимодействия (IPC)**

Операционная система Windows предоставляет механизмы для упрощения обмена данными и обмена данными между приложениями. В совокупности действия, включенные этими механизмами, называются межпроцессными коммуникациями (IPC). Некоторые формы IPC упрощают разделение труда между несколькими специализированными процессами. Другие формы IPC упрощают разделение труда между компьютерами в сети.

Как правило, приложения могут использовать IPC, классифицированные как клиенты или серверы. Клиент – это приложение или процесс, который запрашивает службу от другого приложения или процесса. Сервер – это приложение или процесс, который отвечает на запрос клиента. Многие приложения действуют как клиент, так и сервер в зависимости от ситуации. Например, приложение обработки слов может выступать в качестве клиента при запросе сводной таблицы производственных затрат от приложения электронной таблицы, выступающего в качестве сервера. В свою очередь, приложение электронной таблицы может выступать в качестве клиента при запросе последних уровней инвентаризации из приложения автоматического управления инвентаризацией.

**1.3.1** Использование буфера обмена для IPC

Буфер обмена выступает в качестве центрального депозита для обмена данными между приложениями. Когда пользователь выполняет операцию выреза или копирования в приложении, приложение помещает выбранные данные в буфер обмена в один или несколько стандартных или определяемых приложением форматов. Затем любое другое приложение может получить данные из буфера обмена, выбрав из доступных форматов, которые он понимает. Буфер обмена – это очень слабо связанная среда обмена, где приложениям нужно только согласиться с форматом данных. Приложения могут находиться на одном компьютере или на разных компьютерах в сети.

**1.3.2** Использование COM для IPC

Приложения, использующие OLE, управляют составными документами, т. е. документами, состоящими из различных приложений. OLE предоставляет службы, которые упрощают вызов приложений к другим приложениям для редактирования данных. Например, текстовый процессор, использующий OLE, может внедрить граф из электронной таблицы. Пользователь может автоматически запустить электронную таблицу из текстового процессора, выбрав внедренную диаграмму для редактирования. OLE заботится о запуске электронной таблицы и представлении графа для редактирования. Когда пользователь покидает электронную таблицу, граф будет обновлен в исходном документе с текстовым процессором. Электронная таблица, как представляется, является расширением обработчика слов.

Основой OLE является объектная модель компонента (COM). Программный компонент, использующий COM, может взаимодействовать с различными другими компонентами, даже теми, которые еще не написаны. Компоненты взаимодействуют как объекты и клиенты. Распределенная COM расширяет модель программирования COM, чтобы она работала в сети.

**1.3.3** Использование копирования данных для IPC

Копирование данных позволяет приложению отправлять сведения в другое [приложение с помощью сообщения WM\_COPYDATA](https://learn.microsoft.com/ru-ru/windows/win32/dataxchg/wm-copydata) . Этот метод требует сотрудничества между отправляющим приложением и получающим приложением. Принимающее приложение должно знать формат информации и иметь возможность идентифицировать отправителя. Отправляющее приложение не может изменить память, на которую ссылается любой указатель.

**1.3.4** Использование DDE для IPC

DDE – это протокол, позволяющий приложениям обмениваться данными в различных форматах. Приложения могут использовать DDE для однократных обменов данными или для текущих обменов, в которых приложения обновляют друг друга по мере того, как новые данные становятся доступными.

Форматы данных, используемые DDE, совпадают с форматами, используемыми буфером обмена. DDE можно рассматривать как расширение механизма буфера обмена. Буфер обмена почти всегда используется для однократного ответа на команду пользователя, например выбор команды "Вставить" в меню. DDE также обычно инициируется командой пользователя, но часто она продолжает функционировать без дальнейшего взаимодействия с пользователем. Кроме того, можно определить настраиваемые форматы данных DDE для IPC специального назначения между приложениями с более тесно связанных требований к обмену данными.

Обмены DDE могут происходить между приложениями, работающими на одном компьютере или на разных компьютерах в сети.

**1.3.5** Использование сопоставления файлов для IPC

Сопоставление файлов позволяет процессу обрабатывать содержимое файла, как если бы они были блоком памяти в адресном пространстве процесса. Процесс может использовать простые операции указателя для проверки и изменения содержимого файла. Если два или более процессов получают доступ к одному сопоставлению файлов, каждый процесс получает указатель на память в собственном адресном пространстве, который он может использовать для чтения или изменения содержимого файла. Процессы должны использовать объект синхронизации, например семафор, чтобы предотвратить повреждение данных в многозадачности.

Для предоставления именованной общей памяти между процессами можно использовать особый случай сопоставления файлов. Если при создании объекта сопоставления файлов указать файл переключения системы, объект сопоставления файлов рассматривается как общий блок памяти. Другие процессы могут получить доступ к тому же блоку памяти, открыв тот же объект сопоставления файлов.

Сопоставление файлов довольно эффективно, а также предоставляет поддерживаемые операционной системой атрибуты безопасности, которые могут помочь предотвратить несанкционированное повреждение данных. Сопоставление файлов можно использовать только между процессами на локальном компьютере; его нельзя использовать через сеть.

**1.3.6** Использование Mailslot для IPC

Mailslots обеспечивает односторонняя связь. Любой процесс, создающий mailslot, является сервером mailslot. Другие процессы, называемые клиентами mailslot, отправляют сообщения на сервер mailslot, записывая сообщение в mailslot. Входящие сообщения всегда добавляются в mailslot. Mailslot сохраняет сообщения до тех пор, пока сервер mailslot не считывает их. Процесс может быть как сервером mailslot, так и клиентом mailslot, поэтому двустороннее взаимодействие возможно с помощью нескольких почтовых слоток.

Клиент mailslot может отправлять сообщение в mailslot на локальном компьютере, в mailslot на другом компьютере или во все почтовые слоты с одинаковым именем на всех компьютерах в указанном сетевом домене. Сообщения, передаваемые во все почтовые ящики в домене, не могут превышать 400 байт, в то время как сообщения, отправленные в один почтовый слоток, ограничены только максимальным размером сообщения, указанным сервером mailslot при создании почтового слота.

**1.3.7** Использование каналов для IPC

Существует два типа каналов для двусторонней связи: анонимные каналы и именованные каналы. Анонимные каналы позволяют связанным процессам передавать информацию друг другу. Как правило, анонимный канал используется для перенаправления стандартных входных или выходных данных дочернего процесса, чтобы он смог обмениваться данными с родительским процессом. Для обмена данными в обоих направлениях (дуплексная операция) необходимо создать два анонимных канала. Родительский процесс записывает данные в один канал с помощью дескриптора записи, а дочерний процесс считывает данные из этого канала с помощью дескриптора чтения. Аналогичным образом дочерний процесс записывает данные в другой канал, а родительский процесс считывает из него. Анонимные каналы не могут использоваться по сети, а также не могут использоваться между несвязанными процессами.

Именованные каналы используются для передачи данных между процессами, которые не связаны процессами и между процессами на разных компьютерах. Как правило, процесс сервера именованного канала создает именованный канал с известным именем или именем, который должен быть передан своим клиентам. Клиент именованного канала, который знает имя канала, может открыть свой другой конец, при условии ограничений доступа, указанных процессом сервера именованного канала. После подключения сервера и клиента к каналу они могут обмениваться данными, выполняя операции чтения и записи на канале.[3]

**1.3.8** Критическая секция

Объекты критической секции (critical section) обеспечивают синхронизацию подобно той, как это делается при помощи объектов типа мьютекс, за исключением того, что объекты критической секции могут использоваться только потоками единственного процесса. Объекты событие, мьютекс и семафор могут также использоваться и в однопроцессном приложении, но объекты критической секции предоставляют немного более быстрый, более эффективный механизм для синхронизации взаимоблокировки (специальная для процессора проверка и набор команд). Подобно объекту типа мьютекс, объект критической секции может принадлежать одновременно только одному потоку, что делает его полезным чтобы предохранить совместно используемый ресурс от одновременного доступа. Нет какого-либо гарантированного порядка в котором потоки получают монопольное использование критической секции, однако, система должен быть справедливой ко всем потокам. В отличие от объекта типа мьютекс, там не способа сообщить, была ли критическая секция оставлена.

Процесс несет ответственность распределение памяти, используемой критической секцией. Как правило, это происходит просто при помощи объявления переменной типа CRITICAL\_SECTION. Прежде, чем потоки процесса смогут использовать ее, надо инициализировать критическую секцию, используя функцию InitializeCritical Section или InitializeCritical SectionAndSpinCount.

Поток использует функцию EnterCritical Section или TryEnterCritical Section, чтобы запросить монопольное использование критической секции. Он использует функцию LeaveCriticalSection, чтобы освободить монопольное использование критической секции. Если объект критической секции в текущий момент принадлежит другому потоку, функция EnterCritical Section ждет неограниченно монопольное использование. Напротив, когда объект типа мьютекс используется для взаимоблокировки, функции ожидания принимают заданный интервал времени простоя. Функция TryEnterCriticalSection пытается входить в критическую секцию, не блокируя вызывающий поток.

Как только поток овладеет критической секцией, он может сделать дополнительные вызовы функции EnterCriticalSection или TryEnterCritical Section, не блокируя исполнение своего кода. Это предохраняет поток от  самоблокировки в то время, когда ожидает критическую секцию, которой он уже владеет. Чтобы освободить себя от  монопольного использования, поток должен вызвать LeaveCriticalSection по разу  для каждого случая, когда он входил в критическую секцию.

Поток использует функцию InitializeCriticalSectionAndSpin Count или SetCriticalSectionSpinCount, чтобы установить счет состояния занятости для объекта критической секции. На одно-процессорных системах, счет состояния занятости игнорируется, а счет состояния занятости критической секции устанавливается в 0. На многопроцессорных системах, если критическая секция недоступна, вызывающий поток будет тянуть dwSpinCount время перед выполнением операции ожидания на семафоре, связанном с критической секцией. Если критическая секция становится свободной, в ходе операции протягивания времени, вызывающий поток отменяет операцию ожидания.

Любой поток процесса может использовать функцию DeleteCritical Section, чтобы освободить системные ресурсы, которые были распределены, когда объект критической секции был инициализирован. После того, как эта функция вызвалась, объект критической секции больше не может использоваться для синхронизации.

Когда объект критической секции находится в монопольном использовании, затрагиваются только другие потоки - это те, которые ожидают монопольного использования при вызове функции EnterCritical Section. Потоки, которые не ожидают, особождаются, чтобы продолжить выполнение программы. [4]

**1.3.9** Критические ресурсы

Под критическим разумеется некоторый программный либо аппаратный ресурс, который в каждый момент времени может использоваться одним и только одним процессом, потоком или прерыванием. Правильное обращение с критическими ресурсами является основой грамотного, добротного и надежного программного обеспечения. Поэтому все программные компоненты, вызываемые несколькими различными потоками, должны рассматриваться с точки зрения отнесения к критическим ресурсам. Для этого будем использовать характеристику, обратную критичности, которая называется безопасностью по отношению к тем или иным событиям в программной системе. Для управляющих программ обычно требуется максимальный уровень безопасности компонент – по отношению к сигналам, то есть любым асинхронно возникающим в системе событиям: аппаратным прерываниям, переключениям контекста выполнения (потока) и другим.

Определим два уровня собственной сигналобезопасности программ:

1. Безусловная сигналобезопасность. Обеспечивается повторно-входимыми программными компонентами, обращение к которым может происходить в любой момент времени без каких-либо конфликтов. Такие программы должны выполняться независимо от любой другой программы, включая самою себя. Это достигается несколькими способами:
   1. Программа оперирует исключительно локальными данными, которые размещаются в стеке. Такой метод используется при реализации функций, допускающих рекурсивный вызов. Следует отметить, что стандарт языка программирования С поддерживает рекурсивное обращение к любым функциям.
   2. Программа блокирует все события (прерывания) в системе на время своего выполнения. Этот метод делает сигналобезопасной любую программу, однако должен использоваться лишь для минимальных сегментов кода. Блокировка всех событий может оказаться полезной для гарантированного обеспечения атомарности [семафорных](http://vserv.sinp.msu.ru/invisiblebook/Chapters/PMC_semaphore.html" \l "%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80%D1%8B.|outline) операций.
   3. Программа в состоянии восстановить в начальное состояние свои разделяемые ресурсы по завершении повторного вызова. Технология восстановления – на любителя, но весьма удобна для практического использования в частности, в управляющих программах, выполняемых без поддержки операционной системы и диспетчера потоков.
2. Транзакционная сигналобезопасность. Программные компоненты с транзакционной сигналобезопасностью могут вызываться в любой момент времени, но без повторной входимости. То есть выполнение программы должно полностью завершаться до ее очередного вызова. Транзакционно безопасная программа зависит только от самой себя, но независима от любой другой программы. Например, операции чтения и записи кольцевого буфера являются безусловно сигналобезопасными друг относительно друга, поскольку манипуляции головой и хвостом буфера независимы. Чтение буфера может быть прервано операцией записи и наоборот. Однако, ни чтение, ни запись сами по себе не являются повторно входимыми и должны рассматриваться как критические секции. См. [Кольцевой буфер (FIFO).](http://vserv.sinp.msu.ru/invisiblebook/Chapters/PMC_mantissa.html)

Если программные компоненты зависят как от самих себя, так и от других программ, они не являются сигналобезопасными. Например, операции чтения и записи разделяемой памяти не являются сигналобезопасными: их наложение и на самих себя и друг на друга может привести к искажениям данных. Такие программы следует рассматривать как критические разделяемые ресурсы и применять внешние методы их защиты: [семафоры](http://vserv.sinp.msu.ru/invisiblebook/Chapters/PMC_semaphore.html" \l "%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80%D1%8B.|outline) или взаимные исключения (mutexes). [5]

**1.4 ISO**

Независимая оценка качества и безопасности производства, товаров и услуг – это ключевая задача для участников развитого цивилизованного рынка, ориентированного на интересы конечного потребителя. Именно поэтому в 1946 году инженеры и эксперты ряда развитых стран образовали международный комитет стандартизации, который стал транснациональной платформой для создания системы универсальных и специфических стандартов ISO (International Organization for Standardization).

**1.4.1** Цель стандартизации

В настоящее время в мире действует около 2000 различных стандартов, выпущенных Международной организацией стандартизации, которые применяются как в узкоспециализированных областях, так и в межнациональном масштабе. В частности, вышеупомянутая система ISO 9000 в настоящее время используется в качестве базовой платформы стандартизации менеджмента многими странами, в том числе и Россией.

Основная цель внедрения стандартов ISO заключается в повышении качества и безопасности бизнеса и технологий, что в свою очередь влечёт за собой рост потребительских характеристик выпускаемой продукции. Одновременно улучшается взаимодействие между покупателем и поставщиком, производителем и посредником, придавая рыночным отношениям цивилизованный взаимовыгодный характер.

**1.4.2** Модель OSI

Модель OSI (Open Systems Interconnection) – взаимодействие открытых систем – семиуровневая модель протоколов передачи данных, разработанная Международной организацией по стандартизации (см . – “ISO ”) и CCITT (Consultative Committee for International Telephony and Telegraphy ) для сопряжения различных видов вычислительного и коммуникационного оборудования различных производителей.

**1.4.3** Уровни OSI

Уровни OSI [OSI layers ] – группы протоколов передачи данных, связанные между собой иерархическими отношениями (см . ” Иерархическая структура ” ). Каждый уровень обслуживает вышестоящий уровень и, в свою очередь, пользуется услугами нижестоящего. Наименование уровней OSI (от нижнего к верхнему):

1. Физический уровень [physical layer ] – описывает механические, электрические и функциональные характеристики среды передачи данных, а также средства, предназначенные для установления, поддержания и разъединения связи (“соединений”). При необходимости обеспечивает также кодирование и модуляцию сигнала, передаваемого в сети.
2. Канальный уровень [data link layer ] – отвечает за надежность передачи данных по определенному каналу между двумя соседними узлами, а также за установление, поддержание и разрыв соединений. Блок данных, передаваемых на канальном уровне, называется кадром. Процедуры канального уровня добавляют в передаваемые кадры соответствующие адреса, контролируют ошибки и при необходимости осуществляют повторную передачу кадров. Реализует методы доступа к среде передачи, основанные на передаче маркера (token passing ) или на соперничестве ( см . “Contention”).
3. Сетевой уровень [network layer ] – обеспечивает маршрутизацию пакетов (то есть передачу через несколько каналов по одной или нескольким сетям), что обычно требует включения в пакет сетевого адреса получателя. Отвечает также за обработку ошибок, мультиплексирование пакетов и управление протоколами данных. Самые известные протоколы этого уровня: X.25 (в сетях с коммутацией пакетов), IP (в сетях TCP/IP ), и IPX (в сетях NetWare ). Кроме того, к сетевому уровню относятся протоколы построения маршрутных таблиц для маршрутизаторов, например, OSPF, RIP, ES-IS и IS-IS.
4. Транспортный уровень [transport layer ] – обеспечивает предоставление услуг по надежной передаче данных между оконечными узлами сети, в том числе взаимодействующими через несколько промежуточных узлов коммутации или даже транзитных сетей. Служит границей, ниже которой единицей передаваемой информации являются пакеты, а выше – сообщения. В рамках транспортного протокола модели OSI предусмотрены пять классов сервиса передачи сообщений (0–4).
5. Сеансовый уровень [session layer ] – обеспечивает предоставление услуг, связанных с организацией и синхронизацией обмена данными между процессами на уровне представления.
6. Уровень представления данных [presentation layer ] – включает служебные операции, к которым обращается прикладной уровень ( см. далее ) для интерпретации и преобразования передаваемых и принимаемых данных. Обеспечивает установление общих правил взаимодействия двух ЭВМ различных типов.
7. Прикладной уровень [application layer ] – отвечает за взаимодействие прикладных программ и интерфейс пользователя. Предоставляемые им услуги: электронная почта, идентификация пользователей, передача файлов и т. п.

**1.4.4** Подуровни модели OSI

Подуровни семиуровневой модели OSI :

1. MAC (Media Access Control) – управление доступом к среде.
2. Подуровень канального уровня. Определяет методы доступа к среде передачи данных, формат кадров и адресацию.

Общий термин для описания метода доступа сетевых устройств к среде передачи данных (преимущественно используется применительно к ЛВС ).

LLC (Logical Linc Control) – управление логическим каналом.

1. Подуровень канального уровня, ориентированный на поддержку функций, не зависящих от среды передачи данных. Использует сервис подуровня MAC для предоставления услуг сетевому уровню.
2. Протокол канального уровня, разработанный Комитетом IEEE 802 для локальных вычислительных сетей (см . “ ЛВС”). Является общим для всех стандартных технологий ЛВС. В стандарте IEEE 802.2 определены три класса протоколов управления логическим каналом:
3. LLC1 – без установления соединения, подтверждений, исправления ошибок и управления потоком;
4. LLC2 – с установлением соединения;
5. LLC3 – без установления соединения, но с подтверждениями.

PMD (Physical layer Medium Dependent ) – подуровень физического уровня, зависящий от среды передачи. Является составной частью стандарта FDDI , регламентирующего характеристики волоконно-оптического кабеля для передачи данных, типы коннекторов (соединительных устройств), мощность передатчиков и др. [6]

**2 ИНСТРУМЕНТАЛЬНАЯ ЯЗЫКОВАЯ СРЕДА**

Для выполнения лабораторной работы был выбран язык разработки С++.

Язык программирования С++ представляет высокоуровневый компилируемый язык программирования общего назначения со статической типизацией, который подходит для создания самых различных приложений. На сегодняшний день С++ является одним из самых популярных и распространенных языков.

В качестве среды разработки Visual Studio.

Win32 API (также называемый Windows API) – это исходная платформа для собственных Windows-приложений на языке C/C++, которым требуется прямой доступ к Windows и оборудованию. Он предоставляет интерфейс разработки первого класса без зависимости от управляемой среды выполнения, такой как .NET и WinRT (для приложений UWP для Windows 10). Благодаря этому API Win32 стает оптимальной платформой для приложений, которым требуется самый высокий уровень производительности и прямой доступ к системному оборудованию.

Файлы заголовков для API Windows позволяют создавать 32-разрядные и 64-разрядные приложения. Они включают объявления для версий API Юникода и ANSI.

Microsoft Visual C++ включает копии файлов заголовков Windows, которые были текущими в момент выпуска Visual C++. Таким образом, если вы устанавливаете обновленные файлы заголовков из пакета SDK, на компьютере может возникнуть несколько версий файлов заголовков Windows. Если вы не уверены, что используете последнюю версию файлов заголовков пакета SDK, при компиляции кода, использующего функции, представленные после выпуска Visual C++, будет получен следующий код ошибки: ошибка C2065: необъявленный идентификатор. [5]

**3 РАБОТА ПРИЛОЖЕНИЯ**

**3.1 Структура приложения**

Код организован в классы и функции:

1. Fork: Взаимное исключение (мьютекс), представляющее вилку;
2. Philosopher: Класс, представляющий философа. Управляет вилками и имеет поток;
3. AtomicLoggerOstream: Класс, который защищает поток с помощью мьютекса для атомарной записи;
4. main: Функция, которая создает и запускает философов.

**3.2 Функции приложения**

Код позволяет параметризировать следующие аспекты модели:

1. количество философов (N);
2. имена философов;
3. минимальное и максимальное время ожидания.

Код использует AtomicLoggerOstream для записи действий философов в поток в атомарной манере. Это обеспечивает четкий и последовательный вывод, где действия каждого философа отслеживаются отдельно.

Код демонстрирует один подход к решению задачи обеспечения корректного взаимодействия: использование мьютексов для взаимного исключения доступа к вилкам.

Изменяемые параметры:

Код позволяет изменять следующие параметры модели:

1. количество философов;
2. логика разрешения конфликта (в данном случае используется захват обеих вилок одновременно);
3. интенсивность обращений к критическому ресурсу (регулируется минимальным и максимальным временем ожидания);
4. длительность использования ресурса (подразумевается, что философы едят в течение неопределенного времени);
5. величина тайм-аутов (не используется в данной реализации).

**3.3 Анализ данных**

Код собирает следующие данные:

Количество успешных обращений к вилкам для каждого философа. Это число увеличивается каждый раз, когда философ успешно захватывает обе вилки и начинает есть.

Количество неуспешных обращений к вилкам для каждого философа. Это число увеличивается каждый раз, когда философ пытается захватить вилку, но она уже занята другим философом.

Время ожидания для каждого философа. Это общее время, которое философ проводит в состоянии ожидания, пока он пытается захватить вилки.

Данные собираются в членах класса Philosopher:

1. m\_successful\_forks;
2. m\_unsuccessful\_forks;
3. m\_waiting\_time.

Эти члены обновляются в функции потока thread\_func каждый раз, когда философ выполняет одно из следующих действий:

1. успешно захватывает обе вилки и начинает есть;
2. не может захватить вилку, потому что она уже занята;
3. освобождает вилки после еды.

В функции main эти данные суммируются для всех философов и выводятся вместе с общей статистикой.

Пример вывода представлен на рисунке 3.1.

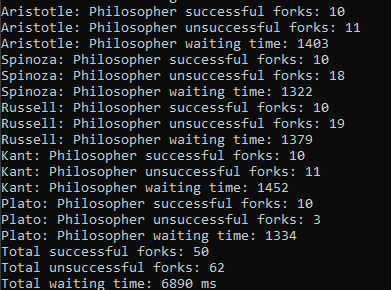


Рисунок 3.1 Вывод данных

Собранные данные можно использовать для анализа и настройки модели. Например, можно использовать для определения того, какие философы чаще всего блокируются, и для настройки параметров модели (например, времени ожидания) для улучшения общей производительности.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Взаимодействие процессов. – [Электронный ресурс]. – Режим доступа: <https://studfile.net/preview/7273593/>.

[2] Многозадачность в ОС и методы оптимизации параллельной работы. – [Электронный ресурс]. – Режим доступа: [https://www.polessu.by/ sites/default/files/sites/default/files/02per/34.pdf](https://www.polessu.by/sites/default/files/sites/default/files/02per/34.pdf).

[3] Взаимодействие между процессами. – [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/>windows/win32/ipc/interprocess  
-communications.

[4] Объекты критической секции. – [Электронный ресурс]. – Режим доступа: <https://vsokovikov.narod.ru/New_MSDN_API/Synchronization/critical_> section.htm

[5] Критические разделяемые ресурсы. – [Электронный ресурс]. – Режим доступа: <http://vserv.sinp.msu.ru/invisiblebook/Chapters/PMC_semaphore.> html

[6] Основные стандарты сетей передачи данных. – [Электронный ресурс]. – Режим доступа: https://www.gpntb.ru/win/book/5/Doc11.HTML

**ПРИЛОЖЕНИЕ** **А**

**(обязательное)**

**Реализация программы на языке С++**

#include <vector>

#include <string>

#include <iostream>

#include <boost/cstdint.hpp>

#include <boost/thread.hpp>

#include <boost/thread/locks.hpp>

#include <boost/format.hpp>

#include <boost/shared\_ptr.hpp>

typedef boost::mutex Fork;

typedef boost::shared\_ptr< Fork > ForkPtr;

typedef boost::lock\_guard< Fork > ForkLock;

#define MIN\_WAIT\_TIME 100

#define NUM\_MEALS 10

#define MAX\_JITTER 50

template< typename Stream >

class AtomicLogger {

public:

AtomicLogger(Stream& stream) :

m\_mutex(),

m\_stream(stream)

{

}

void log(const std::string& str) {

boost::mutex::scoped\_lock lock(m\_mutex);

m\_stream << str << std::endl;

}

private:

mutable boost::mutex m\_mutex;

Stream& m\_stream;

};

typedef AtomicLogger< std::ostream > AtomicLoggerOstream;

typedef boost::shared\_ptr< AtomicLoggerOstream > AtomicLoggerOstreamPtr;

class Philosopher {

public:

Philosopher(

const std::string& name,

ForkPtr fork\_left,

ForkPtr fork\_right,

AtomicLoggerOstreamPtr p\_logger) :

m\_name(name),

m\_continue(true),

mp\_fork\_left(fork\_left),

mp\_fork\_right(fork\_right),

m\_thread(boost::thread(boost::bind(&Philosopher::thread\_func,

this,

&m\_continue,

mp\_fork\_left,

mp\_fork\_right))),

m\_meals\_left(NUM\_MEALS),

mp\_logger(p\_logger)

{

}

~Philosopher() {

done\_dining();

wait\_for\_cmplt();

}

void done\_dining() { m\_continue = false; }

void wait\_for\_cmplt() { m\_thread.join(); }

private:

inline bool can\_grab\_fork(ForkPtr& p\_fork) { return p\_fork->try\_lock(); }

void thread\_func(volatile bool\* p\_continue, ForkPtr fork\_left, ForkPtr fork\_right) {

bool failed\_to\_grab\_fork = false;

while (p\_continue && m\_meals\_left) {

mp\_logger->log(boost::str(boost::format("%1% is thinking") % this->m\_name));

wait();

mp\_logger->log(boost::str(boost::format("%1% is hungry") % this->m\_name));

if (can\_grab\_fork(fork\_left)) {

ForkLock lock\_left(\*fork\_left, boost::adopt\_lock);

if (can\_grab\_fork(fork\_right)) {

ForkLock lock\_right(\*fork\_right, boost::adopt\_lock);

mp\_logger->log(boost::str(boost::format("%1% is eating (%2%)...") % m\_name % m\_meals\_left));

wait();

--m\_meals\_left;

}

else {

failed\_to\_grab\_fork = true;

}

}

else {

failed\_to\_grab\_fork = true;

}

if (failed\_to\_grab\_fork) {

mp\_logger->log(boost::str(boost::format("%1% couldn't get forks; waiting...") % m\_name));

failed\_to\_grab\_fork = false;

wait();

}

}

mp\_logger->log(boost::str(boost::format("%1% is done dining") % m\_name));

}

inline void wait() {

wait(MIN\_WAIT\_TIME + (std::rand() % MAX\_JITTER));

}

inline void wait(boost::uint32\_t time\_in\_ms) {

boost::this\_thread::sleep(boost::posix\_time::milliseconds(time\_in\_ms));

}

std::string m\_name;

volatile bool m\_continue;

ForkPtr mp\_fork\_left;

ForkPtr mp\_fork\_right;

boost::thread m\_thread;

boost::uint32\_t m\_meals\_left;

AtomicLoggerOstreamPtr mp\_logger;

};

typedef boost::shared\_ptr< Philosopher > PhilosopherPtr;

int main() {

const int N = 5;

std::string names[] = { "Aristotle", "Spinoza", "Russell", "Kant", "Plato" };

std::vector< PhilosopherPtr > philosophers;

philosophers.reserve(N);

AtomicLoggerOstreamPtr p\_logger(new AtomicLoggerOstream(std::cout));

std::vector< ForkPtr > forks;

forks.reserve(N);

for (int i = 0; i < N; ++i) {

forks.push\_back(ForkPtr(new Fork()));

}

for (int i = 0; i < N; ++i) {

philosophers.push\_back(PhilosopherPtr(

new Philosopher(names[i], forks[i], forks[(i + 1) % N], p\_logger)));

}

for (int i = 0; i < N; ++i) {

philosophers[i]->wait\_for\_cmplt();

}

p\_logger->log("Everyone is done dining.");

return 0;

}