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 №3

на тему

**ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ: ОБМЕН ДАННыМИ**

Выполнил: студент гр.253504 Божко Я.Д

Проверил: ассистент кафедры информатики

Гриценко Н.Ю

Минск 2024

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc24815)

[1 Теоретические сведения 4](#_Toc24046)

[1.1 Взаимодействие процессов 4](#_Toc5945)

[1.2 Изоляция процессов 5](#_Toc8038)

[1.3 Решения для межпроцессного взаимодействия (IPC) 6](#_Toc7741)

[2 Инструментальная языковая среда 10](#_Toc14524)

[3 Работа приложения 11](#_Toc10638)

[3.1 Структура приложения 11](#_Toc24650)

[3.2 Функции приложения 11](#_Toc26093)

[3.3 Анализ данных 12](#_Toc18220)

[Список использованных источников 13](#_Toc27136)

[Приложение А 14](#_Toc2963)

**ВВЕДЕНИЕ**

Цель работы: приложение, демонстрирующее работу многозадачного комплекса с обменом (передачей) или совместным использованием данных несколькими процессами. Анализ корректности. Оценка эффективности механизмов IPC.

Совместно выполняемые процессы могут быть либо независимыми либо взаимодействующими. Взаимодействие процессов часто понимается в смысле взаимного обмена данными через общий буфер данных. Взаимодействие удобно рассматривать в схеме производитель – потребитель. Для взаимодействия процесса - производителя и процесса - потребителя создается совместный буфер, заполняемый этими процессами.

Буфер имеет фиксированные размеры, и следовательно, процессы могут находиться в состоянии ожидания, когда:

1. Буфер заполнен – ожидает процесс – производитель.
2. Буфер пуст – ожидает процесс – потребитель.

Буфер может предоставляться и поддерживаться самой ОС, например с помощью средств межпроцессорной коммуникации, либо должен быть организован прикладным программистом. При этом оба процесса используют общий участок памяти.

Необходимость решать также проблемы согласованного доступа к данным: путем копирования данных при передаче между процессами либо синхронизации (взаимного исключения).

Процессы двух видов:

1. Процесс-диспетчер: создание рабочих процессов и объектов IPC (например, каналов) для взаимодействия с ними; выдача «заданий» рабочим процессам и сбор результатов.
2. Рабочий процесс: прием «задания», его выполнение (обработка данных), возврат результата.

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 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]

**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.

**3 РАБОТА ПРИЛОЖЕНИЯ**

**3.1 Структура приложения**

Работа кода:

1. Создается очередь задач (TaskQueue).
2. Создается список задач (tasks), каждая из которых имеет идентификатор и набор данных.
3. Запускается поток диспетчера (dispatcherThread), который распределяет задачи среди потоков-рабочих и добавляет задачу выхода в очередь.
4. Создаются потоки-рабочие (workerThreads), которые извлекают задачи из очереди и выполняют их.
5. Поток диспетчера завершается после распределения всех задач.
6. Потоки-рабочие продолжают выполнять задачи, пока не будет извлечена задача выхода.
7. Потоки-рабочие завершаются, и main ожидает их завершения.

**3.2 Функции приложения**

Приложение, демонстрирующее работу многозадачного комплекса с обменом (передачей) или совместным использованием данных несколькими процессами.

Очередь задач (TaskQueue) представляет собой защищенный мьютексом контейнер (std::vector<Task>), который позволяет добавлять и удалять задачи. Она также использует условие переменной (std::condition\_variable), чтобы уведомлять потоки-потребители о наличии новых задач.

Диспетчер (dispatcher) отвечает за распределение задач среди потоков-рабочих. Он делит все задачи на равные части (в данном случае по 2 задачи на поток) и добавляет их в очередь задач. Диспетчер также добавляет в очередь особую задачу выхода, которая сигнализирует потокам-рабочим о завершении работы.

Потоки-рабочие (worker) извлекают задачи из очереди и выполняют их. Они суммируют элементы данных задачи, умножая каждый элемент на идентификатор задачи.

Главная функция (main) создает задачи, запускает диспетчер и потоки-рабочие, а затем дожидается их завершения. Атомарная переменная tasksRemaining отслеживает количество оставшихся задач, а атомарная переменная shouldExit сигнализирует потокам-рабочим о необходимости завершения работы.

**3.3 Анализ данных**

Включает в себя анализ корректности работы с коллизиями, а также оценку эффективности IPC.

**3.3.1** Анализ корректности работы с коллизиями

Код использует мьютекс для защиты доступа к очереди задач. Это гарантирует, что только один поток может одновременно добавлять или удалять задачи из очереди, что предотвращает коллизии.

Кроме того, код использует условие переменной для уведомления потоков-потребителей о наличии новых задач. Это также помогает избежать коллизий, гарантируя, что потоки не будут ожидать наличия задач, которые еще не добавлены в очередь.

**3.3.2** Оценка эффективности IPC

Код использует потоки и мьютексы для реализации IPC (межпроцессного взаимодействия). Мьютексы являются эффективным способом защиты доступа к разделяемым ресурсам (в данном случае к очереди задач). Однако они могут привести к замедлению работы программы, если несколько потоков часто пытаются получить доступ к защищенному ресурсу.

В данном коде потокам-рабочим не нужно часто получать доступ к очереди задач, так как они получают задачи только тогда, когда очередь не пуста. Это означает, что использование мьютексов в коде не должно существенно влиять на производительность.

# 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>.

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

**ПРИЛОЖЕНИЕ** **А**

**(обязательное)**

**Реализация программы на языке С++**

#include <iostream>

#include <thread>

#include <vector>

#include <mutex>

#include <condition\_variable>

#include <atomic>

struct Task {

int id;

std::vector<int> data;

int result;

};

class TaskQueue {

public:

std::mutex mtx;

std::condition\_variable cv;

std::vector<Task> tasks;

void addTask(const Task& task) {

std::lock\_guard<std::mutex> lock(mtx);

tasks.push\_back(task);

cv.notify\_one();

}

Task getTask() {

std::unique\_lock<std::mutex> lock(mtx);

cv.wait(lock, [&] { return !tasks.empty(); });

Task task = tasks.front();

tasks.erase(tasks.begin());

return task;

}

};

std::atomic<int> tasksRemaining;

std::atomic<bool> shouldExit;

void dispatcher(TaskQueue& taskQueue, std::vector<Task>& tasks, int numWorkers) {

int tasksPerWorker = tasks.size() / numWorkers;

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

std::vector<Task> workerTasks(tasks.begin() + i \* tasksPerWorker, tasks.begin() + (i + 1) \* tasksPerWorker);

for (const Task& task : workerTasks) {

taskQueue.addTask(task);

tasksRemaining++;

}

}

for (int i = numWorkers \* tasksPerWorker; i < tasks.size(); i++) {

taskQueue.addTask(tasks[i]);

tasksRemaining++;

}

Task exitTask;

exitTask.id = -1;

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

taskQueue.addTask(exitTask);

}

}

void worker(TaskQueue& taskQueue) {

while (!shouldExit) {

Task task = taskQueue.getTask();

if (task.id == -1) {

break;

}

task.result = 0;

for (int x : task.data) {

task.result += x \* task.id;

}

std::cout << "Обработана задача " << task.id << " с результатом " << task.result << std::endl;

tasksRemaining--;

}

}

int main() {

setlocale(LC\_ALL, "Russian");

std::vector<Task> tasks;

for (int i = 0; i < 10; i++) {

Task task;

task.id = i;

task.data = { 1, 2, 3, 4, 5 };

tasks.push\_back(task);

}

TaskQueue taskQueue;

int numWorkers = 4;

tasksRemaining = tasks.size();

shouldExit = false;

std::thread dispatcherThread(dispatcher, std::ref(taskQueue), std::ref(tasks), numWorkers);

std::vector<std::thread> workerThreads;

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

workerThreads.push\_back(std::thread(worker, std::ref(taskQueue)));

}

dispatcherThread.join();

for (std::thread& t : workerThreads) {

t.join();

}

return 0;

}