Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Сибирский государственный индустриальный университет»

Кафедра систем информатики и управления

УПРАВЛЕНИЕ РЕСУРСАМИ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ

Лабораторный практикум
по дисциплине «Операционные системы».
Направления подготовки:
230100 – Информатика и вычислительная техника;
230400 – Информационные системы и технологии;
230700 – Прикладная информатика

Новокузнецк
2013

УДК 004.42(07)
У739

Рецензент
Кандидат технических наук,
доцент кафедры автоматизации и информационных систем
Андрианов О.Н.

У739 Управление ресурсами вычислительной машины: Лабораторный практикум. / Сост.: М.В. Ляховец. СибГИУ. – Новокузнецк, 2013. – 88 с., ил.

Приводится описание основных подходов к управлению ресурсами локальной вычислительной машины, а также сетевых распределенных систем. Рассматриваются алгоритмы управления процессами, организации работы с памятью и внешними устройствами. Приведены способы синхронизации процессов и передачи сообщений по сети для условий операционной системы Windows с примерами на языке программирования Delphi.

Лабораторный практикум по дисциплине «Операционные системы» предназначено для студентов всех форм обучения направлениям подготовки 230100 «Информатика и вычислительная техника», 230400 «Информационные системы и технологии», 230700 «Прикладная информатика».

ОГЛАВЛЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА №1 «МОДЕЛИРОВАНИЕ АЛГОРИТМОВ УПРАВЛЕНИЯ ПРОЦЕССАМИ В ОПЕРАЦИОННЫХ СИСТЕМАХ»

Цель работы: изучить основные алгоритмы управления процессами в операционных системах, разработать программную модель управления процессами в соответствии с вариантом задания.

1.1 Управление процессами

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. Процесс – абстракция, описывающая выполняющуюся программу. Для операционной системы (ОС) процесс представляет собой заявку на потребление системных ресурсов. Основными функциями подсистемы управления процессами является:

- распределение процессорного времени между несколькими одновременно существующими в системе процессами;
- создание и уничтожение процессов;
- обеспечение процессов необходимыми системными ресурсами;
- поддержка взаимодействий между процессами.

1.1.1 Состояние процессов

В многозадачной системе процесс может находиться в одном из трёх основных состояний:

ВЫПОЛНЕНИЕ – активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

ОЖИДАНИЕ – пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, он ждёт осуществления некоторого события;

ГОТОВНОСТЬ – пассивное состояние процесса, процесс заблокирован в связи с внешними по отношению к нему обстоятельствами.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в операционной системе.

В состоянии ВЫПОЛНЕНИЕ в однопроцессорной системе может находиться только один процесс, а в каждом из состояний ОЖИДАНИЕ и ГОТОВНОСТЬ – несколько процессов, эти процессы образуют очереди, соответственно, ожидающих и готовых процессов. Жизненный цикл процесса начинается с состояния ГОТОВНОСТЬ, когда процесс готов к выполнению и ждёт своей очереди. При активизации процесс переходит в состояние ВЫПОЛНЕНИЕ и находится в нём до тех пор, пока либо он сам освободит процессор, перейдя в состояние ОЖИДАНИЯ какого-нибудь события, либо будет насильно «вытеснен» из процессора. В последнем случае процесс возвращается в состояние

ГОТОВНОСТЬ. В это же состояние процесс переходит из состояния ОЖИДАНИЕ, после того, как ожидаемое событие произойдет.

Таким образом, на протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды (состояние операционной среды отображается состоянием регистров, режимом работы процессора, указателями на открытые файлы и т.д.). Эта информация называется контекстом процесса.

Кроме этого, операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса и другая информация. Информацию такого рода, используемую операционной системой для планирования процессов, называют дескриптором процесса.

Дескриптор процесса по сравнению с контекстом содержит более оперативную информацию, которая должна быть легко доступна подсистеме планирования процессов. Контекст процесса содержит менее актуальную информацию и используется операционной системой только после того, как принято решение о возобновлении прерванного процесса.

1.1.2 Алгоритмы планирования процессов

Планирование процессов включает в себя решение следующих задач:

- определение момента времени для смены выполняемого процесса;
- выбор процесса на выполнение из очереди готовых процессов;
- переключение контекстов "старого" и "нового" процессов.

Существует множество различных алгоритмов планирования процессов, по разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Среди этого множества алгоритмов можно выделить две группы наиболее часто встречающихся алгоритмов: алгоритмы, основанные на квантовании (рисунок 1), и алгоритмы, основанные на приоритетах (см. рис 2).

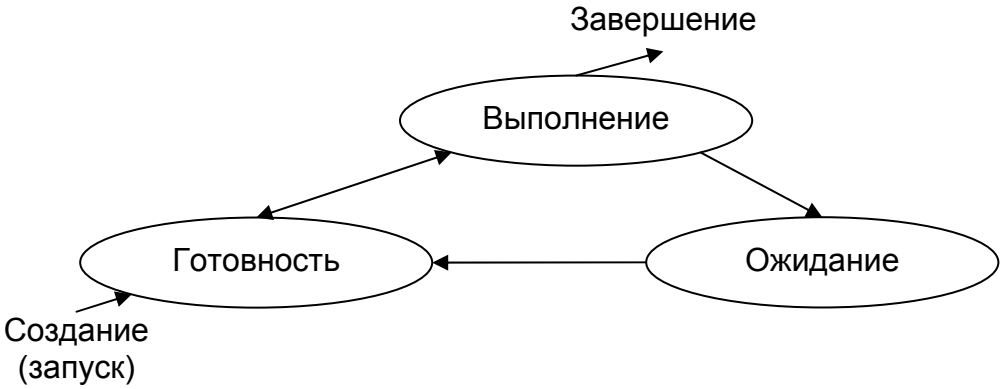


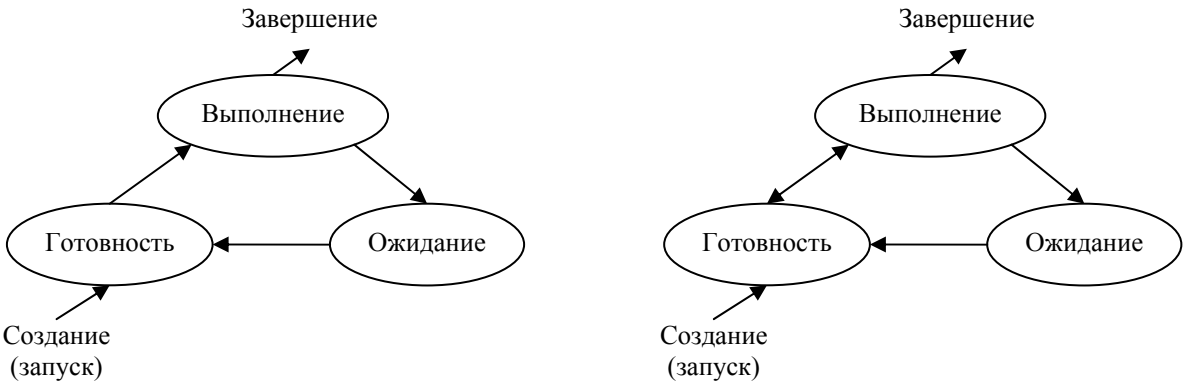
Рисунок 1 – Граф состояний процесса в многозадачной среде, основанной на квантовании

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешёл в состояние ОЖИДАНИЕ,
- исчерпан квант процессорного времени, отведённый данному процессу.

Процесс, который исчерпал свой квант, переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени. Очередь готовых процессов может быть организована: циклически, по правилу "первый пришёл – первый обслужился" (FIFO) или по правилу "последний пришёл – первый обслужился" (LIFO).

Вторая группа алгоритмов использует понятие «приоритет» процесса – некоторое число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины. Приоритет может назначаться директивно администратором системы в зависимости от важности работы, либо вычисляться самой системой по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты. В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ. В системах с абсолютными приоритетами выполнение активного процесса может быть прервано, если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние ГОТОВНОСТИ.



а) б)

Рисунок 2 - Граф состояний процесса в многозадачной среде
(а) – с относительными приоритетами; (б) – с абсолютными приоритетами.

Существуют комбинированные алгоритмы планирования процессов, например, в основе планирования лежит квантование, но величина кванта и/или

порядок выбора процесса из очереди готовых определяется приоритетами процессов.

1.1.3 Проблема синхронизации процессов

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов.

Сложность проблемы синхронизации состоит в нерегулярности возникающих ситуаций, а отсутствие механизмов синхронизации может привести к неправильной работе процессов или даже к краху системы. Примерами таких ситуаций являются гонки и тупики.

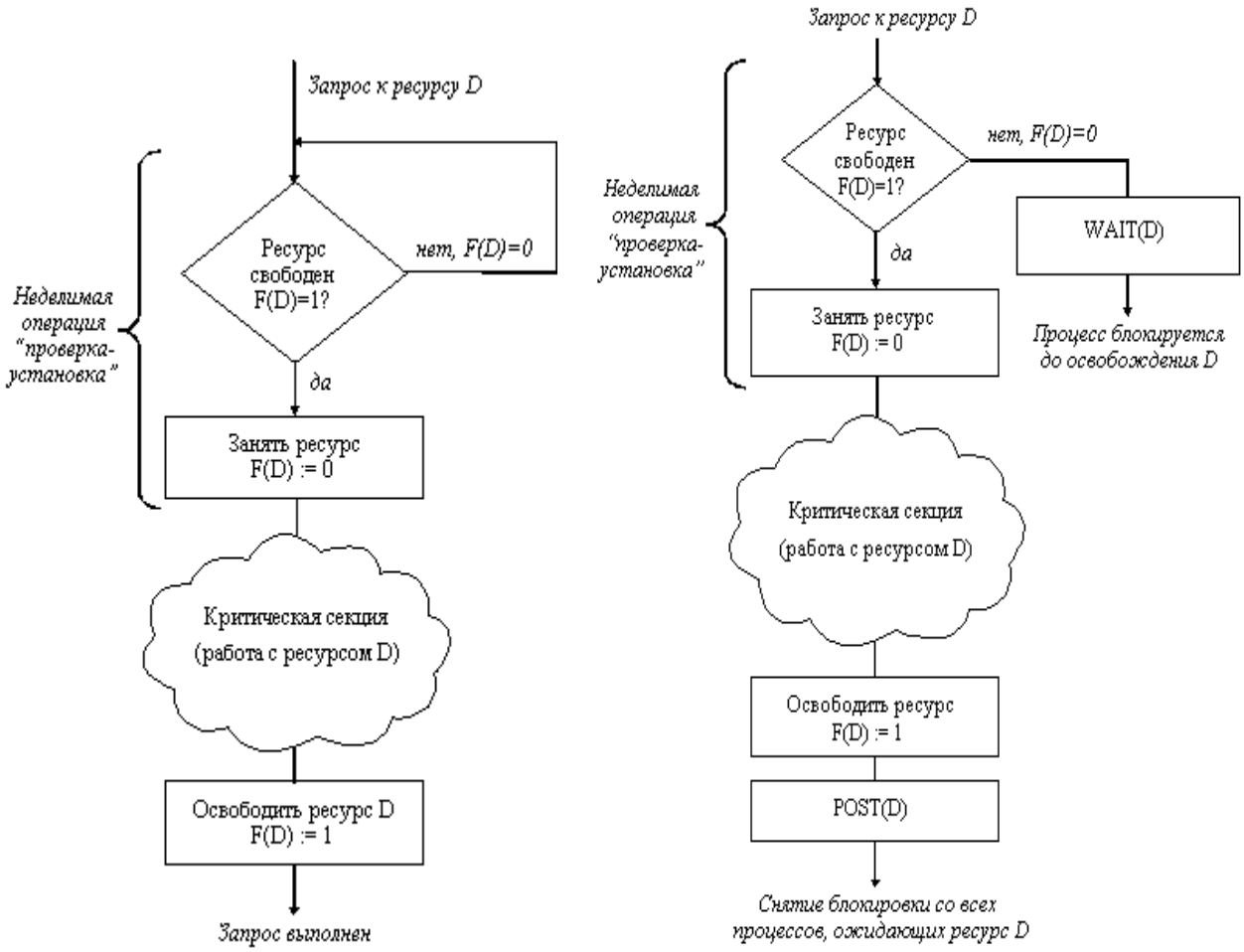
Гонками называются ситуации, когда в отсутствие синхронизации два (или более) процесса обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей процессов.

Тупики – это взаимные блокировки процессов, могущие возникать вследствие недостаточно корректного решения задачи синхронизации и состоящие в том, что ряд процессов удерживает ресурсы, запрашиваемые другими процессами, и в то же время запрашивает ресурсы, удерживаемые другими.

Важным понятием синхронизации процессов является понятие «критическая секция» программы. Критическая секция – это часть программы, в которой осуществляется доступ к разделяемым данным. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один процесс. Этот приём называют взаимным исключением.

Одним из способов взаимного исключения является использование блокирующих переменных. В этом случае с каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение 1, если ресурс свободен, и значение 0, если ресурс занят. На рисунке 3,а показан фрагмент алгоритма процесса, использующего для реализации взаимного исключения доступа к разделяемому ресурсу D блокирующую переменную F(D). Перед входом в критическую секцию процесс проверяет, свободен ли ресурс D. Если он занят, то проверка циклически повторяется, если свободен, то значение переменной F(D) устанавливается в 0, и процесс входит в критическую секцию. После того, как процесс выполнит все действия с разделяемым ресурсом D, значение переменной F(D) снова устанавливается равным 1. Следует помнить, что операция проверки и установки блокирующей переменной должна быть неделимой – атомарной.

Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется.



а) б)

Рисунок 3 – Реализация критических секций
(а) – с использованием блокирующих переменных;
(б) – с использованием аппарата событий

Реализация критических секций с использованием блокирующих переменных имеет существенный недостаток: в течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, бесполезно тратя процессорное время. Для устранения таких ситуаций может быть использован так называемый аппарат событий. В разных операционных системах аппарат событий реализуется по своему, но в любом случае используются системные функции аналогичного назначения, которые условно могут быть названы WAIT(x) и POST(x), где x – идентификатор некоторого события. На рисунке 3,б показан фрагмент алгоритма процесса, использующего эти функции. Если ресурс занят, то процесс вызывает системную функцию WAIT(D), здесь D обозначает событие, заключающееся в освобождении ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе о том, что процесс ожидает события D. Процесс, который в это время использует ресурс D, после выхода из критической секции выполняет системную функцию POST(D), в результате чего опера-

ционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D, в состояние ГОТОВНОСТЬ.

1.2 Задание на лабораторную работу

1.2.1 Задание

Разработать программу, моделирующую один из алгоритмов управления процессами в соответствии с вариантом задания. При моделировании считать что:

- однопроцессорная вычислительная система разделения времени;
- общий объём памяти вычислительной системы составляет 64К;
- ввод-вывод является разделяемым ресурсом, не допускающий одновременного использования несколькими процессами и требующий решения проблемы синхронизации процессов;
- поступаемые на выполнение задачи содержатся в файлах, моделирующих процессы (формат приведён ниже);
- вытеснение процесса из процессора приводит к сохранению информации о его состоянии в контексте процесса, который восстанавливается при последующем поступлении процесса в процессор;
- вытеснение процесса из процессора также приводит к освобождению занимаемой им памяти;
- пользователь может «загружать» в моделируемую систему новые задачи, порождающие процессы;

Предусмотреть возможность интерактивного слежения за состоянием вычислительной системы в процессе выполнения задач (состояние очередей «ОЖИДАНИЕ» и «ГОТОВНОСТЬ», приоритеты процессов, требования к ресурсам, состояние ПРОЦЕССОРА, состояние ВВОДА\ВЫВОДА, состояние ПАМЯТИ и т.д.).

Формат файла, моделирующего задачу:

ПАМЯТЬ-1000
ПРОЦЕССОР-10
ВВОД\ВЫВОД-20
ПРОЦЕССОР-12
...
ПРОЦЕССОР-5
ВВОД\ВЫВОД-25
КОНЕЦ

Где первая строка «ПАМЯТЬ» – определяет требования процесса на объём (в байтах) доступной памяти (если такового в системе на данный момент нет, то процесс поступает в очередь готовых). Остальные строки представляют собой поэтапное требование процесса в процессорном времени «ПРОЦЕССОР» (в секундах или квантах процессорного времени, в зависимости от реализуемого варианта задания) и во времени ввода-вывода «ВВОД-ВЫВОД» (в секундах). Командой окончания выполнения программы является команда «КОНЕЦ».

Варианты заданий представлены в таблице 1.

Таблица 1 – Варианты заданий лабораторной работы №1

Вариант	**Алгоритм управления процессами**
1	Управление процессами на основе бесприоритетного квантования. Очередь готовых процессов – FIFO. Синхронизация процессов с использованием аппарата событий.
2	Управление процессами на основе квантования. Очередь готовых процессов – с абсолютными приоритетами (приоритеты назначаются пользователем при загрузке соответствующей задачи). Синхронизация процессов с использованием блокирующих переменных.
3	Управление процессами на основе квантования. Очередь готовых процессов – с относительными приоритетами (приоритеты назначаются пользователем при загрузке соответствующей задачи). Синхронизация процессов с использованием аппарата событий.
4	Управление процессами на основе квантования. Очередь готовых процессов – LIFO. Квант времени, отводимый процессу, выбирается в соответствии с приоритетом процесса (приоритеты назначаются пользователем при загрузке соответствующей задачи). Синхронизация процессов с использованием блокирующих переменных.
5	Управление процессами на основе квантования. Очередь готовых процессов – с относительными приоритетами (приоритеты назначаются пользователем при загрузке соответствующей задачи). Квант времени, отводимый процессу, выбирается случайным образом в заданном диапазоне. Синхронизация процессов с использованием аппарата событий.
6	Управление процессами на основе безприоритетного квантования. Очередь готовых процессов – LIFO. Квант времени, отводимый процессу, равен среднему значению требований процессорного времени «ПРОЦЕССОР» соответствующей задачи. Синхронизация процессов с использованием блокирующих переменных.
7	Управление процессами на основе квантования. Очередь готовых процессов – с абсолютными приоритетами. Приоритеты назначаются системой: процесс, предъявляющий минимальные требования к вводу-выводу, получает наивысший приоритет Синхронизация процессов с использованием аппарата событий.
8	Управление процессами на основе квантования. Очередь готовых процессов – с относительными приоритетами (приоритеты назначаются системой случайным образом при загрузке соответствующей задачи). Синхронизация процессов с использованием блокирующих переменных.

1.2.2 Пример выполнения программных модулей

Рассмотрим вариант реализации модели алгоритма управления процессами, основанный на квантовании. Внешний вид программы представлен на рисунке 4.

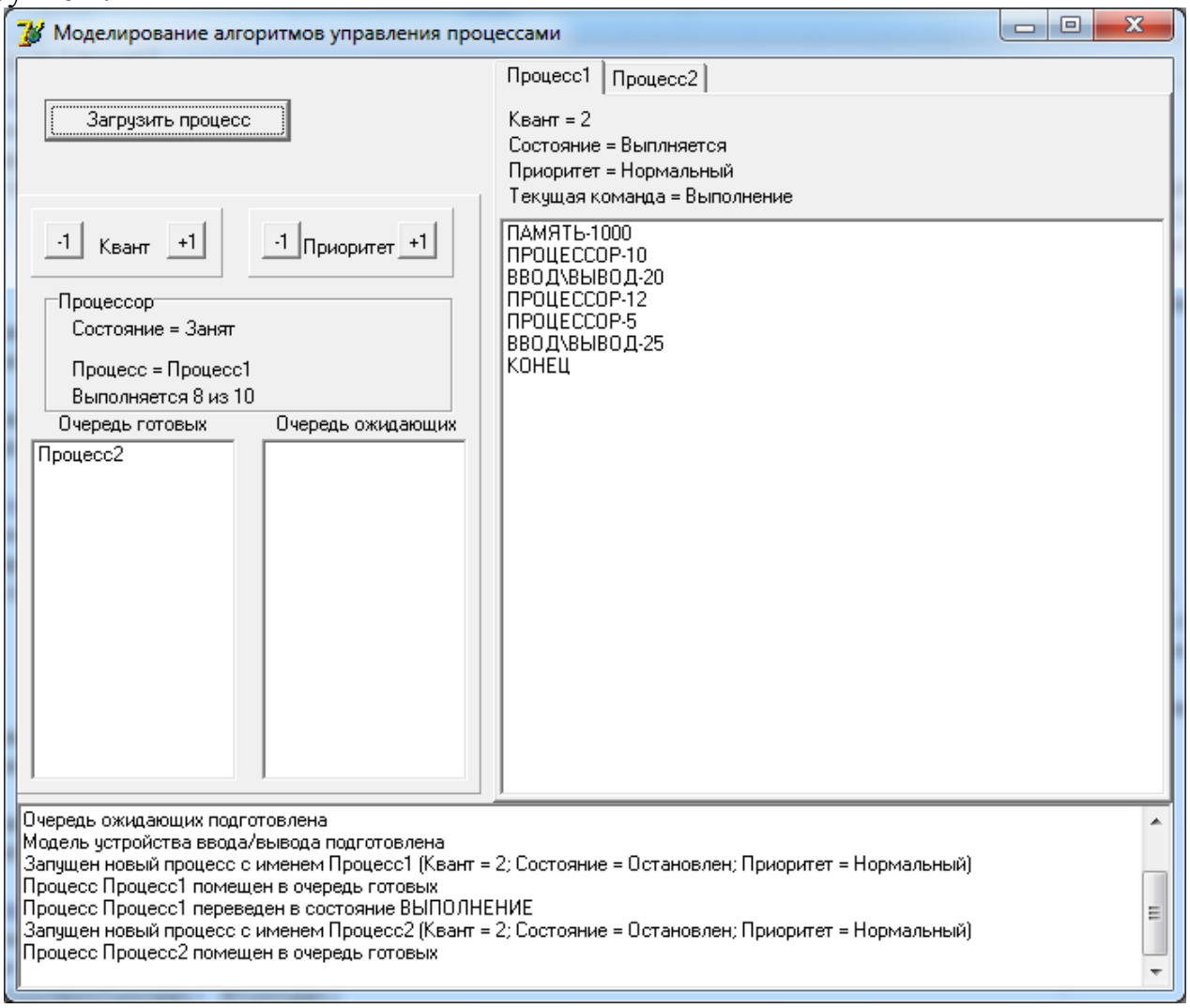


Рисунок 4 – Внешний вид моделирующей программы

Код программы представлен в приложении А и состоит из трёх программных модулей:

- Processes.dpr – основной модуль программы;
- Unit1.pas – модуль имитации процессора и устройства ввода-вывода;
- Unit2.pas – модуль объявления типов и служебных функций.

В примере реализован простейший алгоритм планирования процессов, основанный на квантовании, без учета приоритетов и поддержки средств синхронизации выполнения процессов. Очереди процессов реализованы согласно стековому алгоритму.

ЛАБОРАТОРНАЯ РАБОТА №2 «МЕТОДЫ РАСПРЕДЕЛЕНИЯ ПАМЯТИ. ИССЛЕДОВАНИЕ СПОСОБОВ ДЕФРАГМЕНТАЦИИ»

Цель работы: изучить основные алгоритмы управления памятью в операционных системах, разработать программную модель управления памятью в соответствии с вариантом задания.

2.1 Управление памятью

Все методы управления памятью могут быть разделены на два класса: методы, которые используют перемещение процессов между оперативной памятью и диском, и методы, которые не делают этого. Ниже рассмотрены основные алгоритмы управления памятью.

2.1.1 Методы распределения памяти без использования дискового пространства

Распределение памяти фиксированными разделами

Самым простым способом управления оперативной памятью является разделение её на несколько разделов фиксированной величины. Это может быть выполнено вручную оператором во время старта системы или во время её генерации. Очередная задача, поступившая на выполнение, помещается либо в общую очередь, либо в очередь к некоторому разделу. Подсистема управления памятью в этом случае выполняет следующие задачи: во-первых, сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел и, во-вторых, осуществляет загрузку программы и настройку адресов.

Достоинство метода – простота реализации.

Недостатки метода:

- уровень мультипрограммирования системы ограничен числом разделов, так как в каждом разделе может выполняться только одна программа;
- даже если программа имеет небольшой объём, она будет занимать весь раздел;
- если объём оперативной памяти машины достаточен для выполнения некоторой программы, но размер каждого отдельного раздела мал, то программа не будет загружена в память и, соответственно, не сможет быть выполнена.

Распределение памяти разделами переменной величины

В этом случае память машины не делится заранее на разделы. Сначала вся память свободна. Каждой вновь поступающей задаче выделяется необходимая ей память. Если достаточный объём памяти отсутствует, то задача не принимается на выполнение и стоит в очереди. После завершения задачи память освобождается, и на это место может быть загружена другая задача. Таким образом, в произвольный момент времени оперативная память представляет собой слу-

чайную последовательность занятых и свободных участков (разделов) произвольного размера.

Задачами ОС в этом случае является:

- ведение таблиц свободных и занятых областей;
- при поступлении новой задачи, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившей задачи;
- загрузка задачи в выделенный раздел и корректировка таблиц свободных и занятых областей;
- после завершения задачи корректировка таблиц свободных и занятых областей.

Недостаток метода: фрагментация памяти – наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов), таких что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объём фрагментов может составить величину, превышающую требуемый объем памяти.

Перемещаемые разделы

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область. В дополнение к функциям, которые выполняет операционная система при распределении памяти переменными разделами, в данном случае она должна ещё время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется "сжатием". Сжатие может выполняться либо при каждом завершении задачи, либо только тогда, когда для вновь поступившей задачи нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц, а во втором – реже выполняется процедура сжатия.

Достоинство метода – более эффективное использование памяти.

Недостаток метода – процедура сжатия может требовать значительного времени.

2.1.2 Методы распределения памяти с использованием дискового пространства

Страничное распределение

На рисунке 2.1 показана схема страничного распределения памяти. Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами. В общем случае размер виртуального адресного пространства не является кратным размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками).

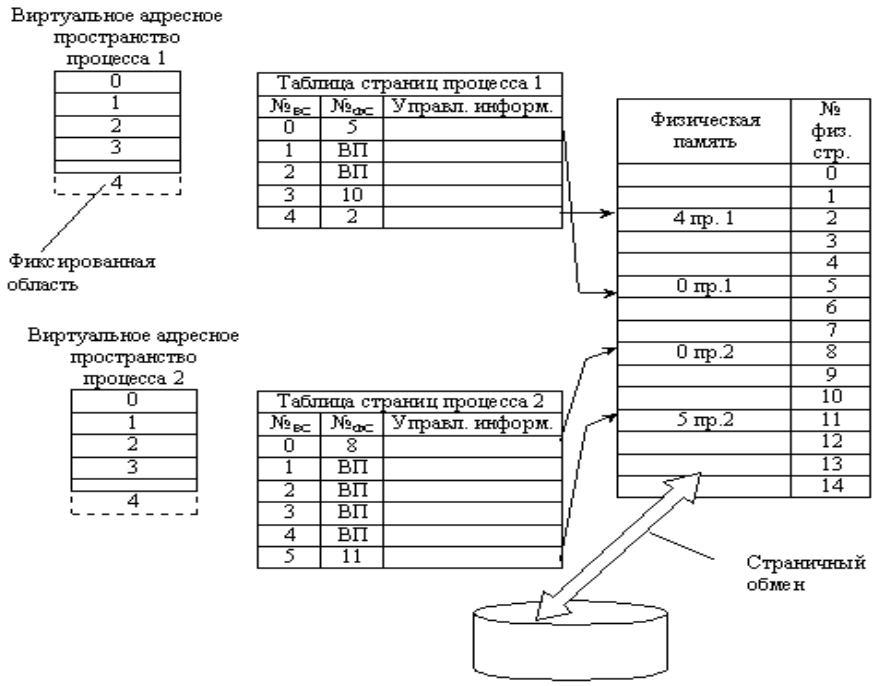


Рисунок 2.1 - Страничное распределение памяти

При загрузке процесса часть его виртуальных страниц помещается в оперативную память, а остальные - на диск. При загрузке операционная система создает для каждого процесса информационную структуру - таблицу страниц, в которой устанавливается соответствие между номерами виртуальных и физических страниц для страниц, загруженных в оперативную память, или делается отметка о том, что виртуальная страница выгружена на диск. Кроме того, в таблице страниц содержится управляющая информация, такая как признак модификации страницы, признак невыгружаемости (выгрузка некоторых страниц может быть запрещена), признак обращения к странице (используется для подсчета числа обращений за определенный период времени) и другие данные, формируемые и используемые механизмом виртуальной памяти.

При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое страничное прерывание. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти.

В данной ситуации может быть использовано много разных критериев выбора, наиболее популярные из них следующие: дольше всего не использовавшаяся страница; первая попавшаяся страница; страница, к которой в последнее время было меньше всего обращений.

Страничное распределение памяти может быть реализовано в упрощенном варианте, без выгрузки страниц на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страничной организации хотя и не предоставляет пользователю виртуальной памяти, но почти исключает фрагментацию за счет того, что программа может загружаться в несмежные области, а также того, что при загрузке виртуальных страниц никогда не образуется остатков.

Сегментное распределение

При страничной организации виртуальное адресное пространство процесса делится механически на равные части. Это не позволяет дифференцировать способы доступа к разным частям программы (сегментам), а это свойство часто бывает очень полезным. Кроме того, разбиение программы на "осмысленные" части делает принципиально возможным разделение одного сегмента несколькими процессами.

При сегментном распределении памяти (рисунок 2.2) виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Иногда сегментация программы выполняется по умолчанию компилятором.

При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

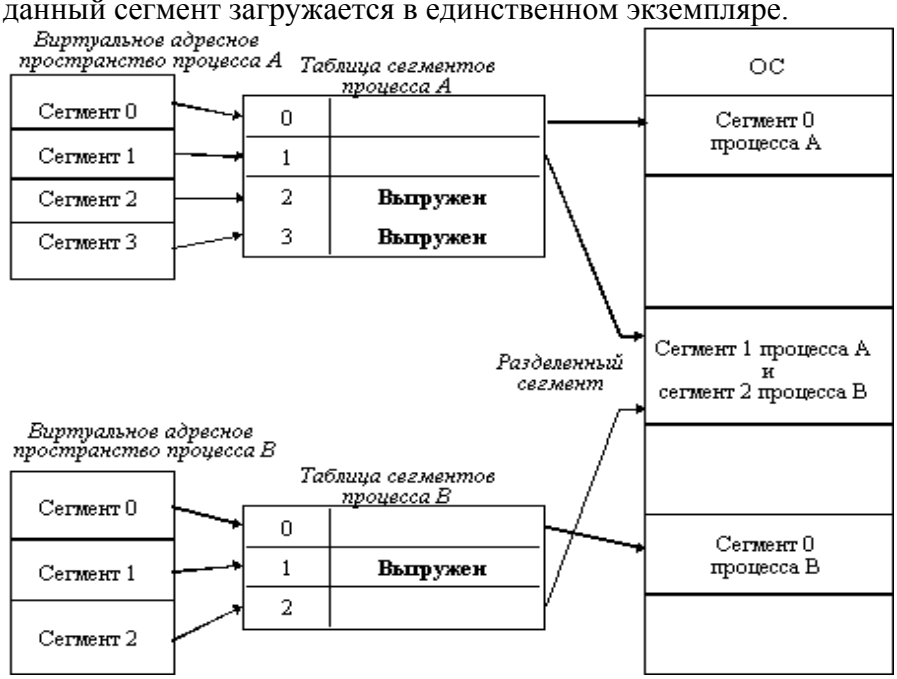


Рисунок 2.2 - Распределение памяти сегментами

Система с сегментной организацией функционирует аналогично системе со страничной организацией: время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются. Кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Недостатком данного метода распределения памяти является фрагментация на уровне сегментов и более медленное по сравнению со страничной организацией преобразование адреса.

Странично-сегментное распределение

Данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс.

Свопинг

Разновидностью виртуальной памяти является свопинг.

Анализ загрузки процессора в зависимости от числа одновременно выполняемых процессов показал, что для загрузки процессора на 90% достаточно всего трех счетных задач. Однако для того, чтобы обеспечить такую же загрузку интерактивными задачами, выполняющими интенсивный ввод-вывод, потребуются десятки таких задач. В этих условиях был предложен метод организации вычислительного процесса, называемый свопингом. В соответствии с этим методом некоторые процессы (обычно находящиеся в состоянии ожидания) временно выгружаются на диск. Планировщик операционной системы не исключает их из своего рассмотрения, и при наступлении условий активизации некоторого процесса, находящегося в области свопинга на диске, этот процесс перемещается в оперативную память. Если свободного места в оперативной памяти не хватает, то выгружается другой процесс.

При свопинге, в отличие от рассмотренных ранее методов реализации виртуальной памяти, процесс перемещается между памятью и диском целиком.

1.1 Задание на лабораторную работу

Модернизировать программу, полученную в ходе решения лабораторной работы №1 и моделирующую один из алгоритмов управления памятью в соответствии с вариантом задания. При моделировании считать что:

- объем моделируемой «памяти» составляет 64К;
- поступаемые на выполнение задачи содержатся в файлах, которые пользователь может «загружать» в моделируемую «память» и выгружать из нее (файл моделирует лишь размер задачи);
- размер задачи в диапазоне от 0 до 65535 байт.

Программа должна иметь возможность просмотра состояния моделируемой «памяти». Требование процесса на объем доступной памяти (в байтах) указывается в скриптовом файле после команды «ПАМЯТЬ».

Варианты заданий

Вариант	**Алгоритм управления памятью**
1	Распределение памяти фиксированными разделами
2	Распределение памяти перемещаемыми разделами
3	Распределение памяти разделами переменной величины
4	Страничное распределение памяти
5	Свопинг

2 ЛАБОРАТОРНАЯ РАБОТА №3 «СИНХРОННЫЕ И АСИНХРОННЫЕ ПЕРЕДАЧИ В ОПЕРАЦИЯХ ВВОДА-ВЫВОДА»

Цель работы: изучение основных способов управления устройствами ввода/вывода при осуществлении синхронных и асинхронных передач в операциях ввода/вывода, разработка программной модели управления внешним устройством в соответствии с вариантом задания.

2.1 Теоретические сведения

Физическая организация устройств ввода-вывода

Устройства ввода-вывода делятся на два типа:

- Блок-ориентированные устройства хранят информацию в блоках фиксированного размера, каждый из которых имеет свой собственный адрес (например, диск).
- Байт-ориентированные устройства не адресуемы и не позволяют производить операцию поиска, они генерируют или потребляют последовательность байтов (например, строчные принтеры, сетевые адаптеры).

Однако существуют устройства, не относящиеся ни к одному классу. (например, часы, которые, с одной стороны, не адресуемы, а с другой стороны, не порождают потока байтов).

Внешнее устройство обычно состоит из механического и электронного компонента. Электронный компонент называется контроллером устройства или адаптером. Механический компонент представляет собственно устройство. Некоторые контроллеры могут управлять несколькими устройствами.

ОС обычно имеет дело не с устройством, а с контроллером. Каждый контроллер имеет несколько регистров, которые используются для взаимодействия с центральным процессором. В некоторых компьютерах эти регистры являются частью физического адресного пространства. В таких компьютерах нет специальных операций ввода-вывода. В других компьютерах адреса регистров ввода-вывода, называемых часто портами, образуют собственное адресное пространство за счет введения специальных операций ввода-вывода.

ОС выполняет ввод-вывод, записывая команды в регистры контроллера. Когда команда принята, процессор оставляет контроллер и занимается другой работой. При завершении команды контроллер организует прерывание для того, чтобы передать управление процессором ОС, которая должна проверить результаты операции. Процессор получает результаты и статус устройства, читая информацию из регистров контроллера.

Организация программного обеспечения ввода-вывода

Основная идея организации программного обеспечения (ПО) ввода-вывода состоит в разбиении его на несколько уровней, причем нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних, а те, в свою очередь, обеспечивают удобный интерфейс для пользователей. Есть несколько принципов организации ПО:

- Независимость от устройств.
- Обработка ошибок.
- Использование блокирующих (синхронных) и неблокирующих (асинхронных) передач. Большинство операций физического ввода-вывода выполняется асинхронно - процессор начинает передачу и переходит на другую работу, пока не наступает прерывание. Пользовательские программы намного легче писать, если операции ввода-вывода блокирующие - после команды чтения программа автоматически приостанавливается до тех пор, пока данные не попадут в буфер программы. ОС выполняет операции ввода-вывода асинхронно, но представляет их для пользовательских программ в синхронной форме.

- Разделяемость устройств. Одни устройства являются разделяемыми (возможен одновременный доступ нескольких пользователей к устройству), а другие – выделенными (нежелателен одновременный доступ нескольких пользователей к устройству).

Чаще всего программное обеспечение ввода-вывода делят на четыре слоя:

- обработка прерываний,
- драйверы устройств,
- независимый от устройств слой операционной системы,
- пользовательский слой программного обеспечения.

Обработка прерываний

Процесс, инициировавший операцию ввода-вывода, блокирует себя до завершения операции и наступления прерывания. При наступлении прерывания процедура обработки прерывания выполняет разблокирование процесса, инициировавшего операцию ввода-вывода.

Драйверы устройств

Весь зависимый от устройства код помещается в драйвер устройства. Каждый драйвер управляет устройствами одного типа или одного класса. В операционных системах только драйвер устройства знает о конкретных особенностях какого-либо устройства.

Драйвер устройства принимает запрос от устройств программного слоя и решает, как его выполнить. Если драйвер был свободен во время поступления запроса, то он начинает выполнять запрос немедленно. Если же он был занят обслуживанием другого запроса, то вновь поступивший запрос присоединяется к очереди уже имеющихся запросов, и он будет выполнен, когда наступит его очередь.

Драйвер решает, какие операции контроллера нужно выполнить и в какой последовательности.

После передачи команды контроллеру драйвер решает, блокировать ли себя до окончания заданной операции или нет. Если операция занимает значительное время, то драйвер блокируется до тех пор, пока операция не завершится, и обработчик прерывания не разблокирует его. Если команда ввода-вывода выполняется быстро, то драйвер ожидает ее завершения без блокирования.

Независимый от устройств слой операционной системы

Типичными функциями для независимого от устройств слоя являются:

- обеспечение общего интерфейса к драйверам устройств,
- именование устройств,
- защита устройств,
- обеспечение независимого размера блока,
- буферизация,
- распределение памяти на блок-ориентированных устройствах,
- распределение и освобождение выделенных устройств,
- уведомление об ошибках.

Пользовательский слой программного обеспечения

Хотя большая часть программного обеспечения ввода-вывода находится внутри операционной системы, некоторая его часть содержится в библиотеках, связываемых с пользовательскими программами. Системные вызовы, включающие вызовы ввода-вывода, обычно делаются библиотечными процедурами. Стандартная библиотека ввода-вывода любого языка программирования содержит большое число процедур, которые выполняют ввод-вывод и работают как часть пользовательской программы.

Спулинг

Другой категорией программного обеспечения ввода-вывода является подсистема спулинга (spooling). Спулинг - это способ работы с выделенными устройствами в мультипрограммной системе. Рассмотрим типичное устройство, требующее спулинга - строчный принтер. Хотя технически легко позволить каждому пользовательскому процессу открыть специальный файл, связанный с принтером, такой способ опасен из-за того, что пользовательский процесс может монополизировать принтер на произвольное время. Вместо этого создается специальный процесс - монитор, который получает исключительные права на использование этого устройства. Также создается специальный каталог, называемый каталогом спулинга. Для того, чтобы напечатать файл, пользовательский процесс помещает выводимую информацию в этот файл и помещает его в каталог спулинга. Процесс-монитор по очереди распечатывает все файлы, содержащиеся в каталоге спулинга.

2.2 Задание на лабораторную работу

Модернизировать программу, полученную в ходе решения лабораторной работы №2 и моделирующую операции ввода/вывода в соответствии с вариантом задания. При моделировании считать, что:

- размер блока в блок-ориентированных устройствах – 1 Кбайт, время обработки которого принять равным 1 сек.
- время обработки одного байта команды ввода/вывода в байт-ориентированных устройствах принять равным 1 сек.
- предусмотреть обработку ошибок, возникающих при осуществлении операции ввода-вывода (вероятность возникновения ошибки 5%).

Программа должна иметь возможность просмотра состояния моделируемого «внешнего устройства» и этапов обработки операции ввода-ввода. Объем данных, передаваемых устройству при операции ввода/вывода, указывается в скриптовом файле после команды «ВВОД/ВЫВОД» в Кбайтах для блок-ориентированных устройств и в байтах для байт-ориентированных.

Варианты заданий

Вариант	**Задание**
1	Блок-ориентированное устройство. Синхронная передача операций ввода/вывода
2	Блок-ориентированное устройство. Асинхронная передача операций ввода/вывода
3	Байт-ориентированное устройство. Синхронная передача операций ввода/вывода
4	Байт-ориентированное устройство. Асинхронная передача операций ввода/вывода
5	Байт-ориентированное устройство. Синхронная передача операций ввода/вывода. Спулинг
6	Байт-ориентированное устройство. Асинхронная передача операций ввода/вывода. Спулинг
7	Блок-ориентированное устройство. Синхронная передача операций ввода/вывода. Спулинг
8	Блок-ориентированное устройство. Асинхронная передача операций ввода/вывода. Спулинг

3 ЛАБОРАТОРНАЯ РАБОТА №4 «ОРГАНИЗАЦИЯ ДОСТУПА К ФИЗИЧЕСКИМ ЗАПИСЯМ В ФАЙЛОВЫХ СИСТЕМАХ»

Цель работы: изучить основные методы построения файловых систем, разработать программную модель файловой системы.

3.1 Теоретические сведения

Файлы и каталоги

Файл - это именованная область на носителе внешней памяти. Файлы идентифицируются именами. Пользователи дают файлам символьные имена, при этом учитываются определенные ограничения ОС. Разные файлы могут иметь одинаковые символьные имена, в этом случае файл однозначно идентифицируется так называемым составным именем (путем), представляющем собой последовательность символьных имен каталогов.

Каталог – это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений, а с другой стороны - это файл, содержащий системную информацию о группе файлов, его составляющих. В каталоге содержится список файлов, входящих в него, и устанавливается соответствие между файлами и их характеристиками (атрибутами).

В разных файловых системах могут использоваться в качестве атрибутов разные характеристики, например: информация о разрешенном доступе; пароль для доступа к файлу; владелец файла; создатель файла; времена создания; текущий размер файла; признак "только для чтения"; признак "скрытый файл"; признак "системный файл"; признак "архивный файл", и т.д.

Каталоги могут непосредственно содержать значения характеристик файлов или ссылаться на таблицы, содержащие эти характеристики. Каталоги могут образовывать иерархическую (древовидную или сетевую) структуру за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня. Каталоги образуют дерево, если файлу разрешено входить только в один каталог, и сеть - если файл может входить сразу в несколько каталогов. Как и любой другой файл, каталог имеет символьное имя и однозначно идентифицируется составным именем, содержащим цепочку символьных имен всех каталогов, через которые проходит путь от корня до данного каталога.

Общая модель файловой системы

Функционирование любой файловой системы можно представить многоуровневой моделью (рис. 4.1), в которой каждый уровень предоставляет некоторый интерфейс (набор функций) вышележащему уровню, а сам, в свою очередь, для выполнения своей работы использует интерфейс (обращается с набором запросов) нижележащего уровня.

Физическая организация файла на устройстве внешней памяти

Файл состоит из физических записей - блоков. Блок - наименьшая единица данных, которой внешнее устройство обменивается с оперативной памятью. Существует четыре основных варианта размещения файла, который описаны ниже.

Непрерывное размещение – простейший вариант физической организации (рис. 4.2.а), при котором файлу предоставляется последовательность блоков диска, образующих единый сплошной участок дисковой памяти. Для задания адреса файла в этом случае достаточно указать только номер начального блока.

Размещение в виде связанного списка блоков дисковой памяти (рис. 4.2.б) – следующий способ физической организации. При таком способе в начале каждого блока содержится указатель на следующий блок. В этом случае адрес файла также может быть задан одним числом – номером первого блока.

Рисунок 4.1. Модель функционирования файловой системы

Использование связанного списка индексов (рис. 4.2.в). С каждым блоком связывается некоторый элемент - индекс. Индексы располагаются в отдельной области диска. Если некоторый блок распределен некоторому файлу, то индекс этого блока содержит номер следующего блока данного файла.

Перечисление в дескрипторе данного файла номеров блоков, занимаемых файлом (рис. 4.2. г). Для хранения адреса файла выделено 13 полей. Если размер файла меньше или равен 10 блокам, то номера этих блоков непосредственно перечислены в первых десяти полях адреса. Если размер файла больше 10 блоков, то следующее 11-е поле содержит адрес блока, в котором могут быть расположены еще 128 номеров следующих блоков файла. Если файл больше, чем 10+128 блоков, то используется 12-е поле, в котором находится номер блока, содержащего 128 номеров блоков, которые содержат по 128 номеров блоков данного файла. И, наконец, если файл больше 10+128+128*128, то используется последнее 13-е поле для тройной косвенной адресации, что позволяет задать адрес файла, имеющего размер максимум 10+ 128 + 128*128 + 128*128*128.

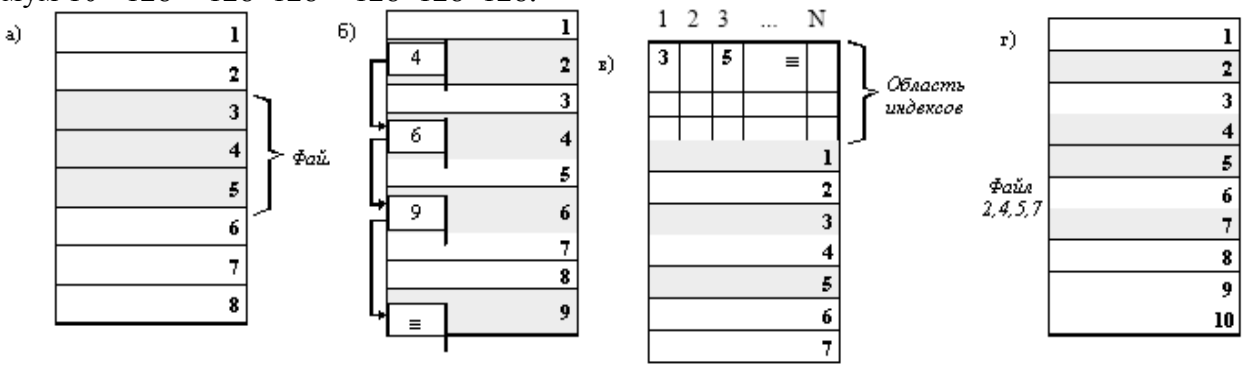


Рисунок 4.2 - Физическая организация файла:
а - непрерывное размещение; б - связанный список блоков; в - связанный список индексов; г - перечень номеров блоков

Права доступа к файлу и каталогу

Определить права доступа к файлу или каталогу – значит определить для каждого пользователя набор операций, которые он может применить к данному файлу. В разных файловых системах может быть определен свой список операций доступа: создание файла, уни-

чтожение файла, открытие файла, закрытие файла, чтение файла, запись в файл, установление новых значений атрибутов и т.д.

В самом общем случае права доступа могут быть описаны матрицей прав доступа, в которой столбцы соответствуют всем файлам системы, строки - всем пользователям, а на пересечении строк и столбцов указываются разрешенные операции. В некоторых системах пользователи могут быть разделены на отдельные категории. Для всех пользователей одной категории определяются единые права доступа.

Различают два основных подхода к определению прав доступа:

- избирательный доступ, когда для каждого файла и каждого пользователя сам владелец может определить допустимые операции;
- мандатный подход, когда система наделяет пользователя определенными правами по отношению к каждому разделяемому файлу в зависимости от того, к какой группе пользователь отнесен.

Файловые системы

Файловая система FAT

FAT применяется для дисков малой емкости. К основным недостаткам FAT могут быть отнесены следующие:

- ограничения, налагаемые на размер файлов и дискового пространства;
- ограничение длины имени файла;
- фрагментация файлов, приводящая к снижению быстродействия системы и износу оборудования;
- непроизводительные затраты памяти, вызванные большими размерами кластеров;
- подверженность потерям данных.

После форматирования диска в FAT на него наносится следующая служебная информация:

- ЗАПИСЬ СТАРТА СИСТЕМЫ (начиная с первого сектора нулевого цилиндра, состоит из таблицы, содержащей информацию о диске; машинного кода, загружающего файлы IO.SYS и MSDOS.SYS);
- ТАБЛИЦА РАЗМЕЩЕНИЯ ФАЙЛОВ И КОРНЕВОЙ КАТАЛОГ.

Корневой каталог - это таблица-описание содержимого диска. Каждому файлу в таблице соответствует одна запись. Запись занимает 32 байта, разбитых на 8 полей (см. табл. 4.1).

Таблица 4.1 - Структура дискриптора

Название поля	**Короткий адрес**
Имя файла	0-7
Расширение	8-10
Атрибуты	11
Для нужд системы	12-21
Метка времени	22-23
Метка даты	24-25
Начальный кластер	26-27
Размер файла	28-31

Корневой каталог занимает непрерывную область фиксированного размера. Размер корневого каталога задается при форматировании и определяет максимальное количество файлов и каталогов, которые могут быть описаны в корневом каталоге.

Таблица размещения файлов FAT представляет собой карту с адресами файлов на диске. Каждому файлу в таблице соответствует группа записей, описывающих его физическое расположение на диске. При форматировании диск разбивается на сектора, сектора объеди-

няются в кластеры. Кластерам присваиваются номера в соответствии с их физическим расположением на диске. Первый кластер расположен непосредственно за корневым каталогом, второй - непосредственно за первым кластером и т.д. Для каждого кластера FAT имеет свою индивидуальную ячейку, в которой хранится информация об использовании данного кластера.

При размещении нового файла или каталога (каталог – это файл специального вида, содержащий записи аналогичные корневому каталогу) ему отводятся последовательно или непоследовательно размещенные кластеры и производится модифицирование FAT. При этом в дескрипторе данного файла или каталога указывается номер первого занятого кластера, а в соответствующей ячейке FAT номер следующего кластера и т.д.

Эти ячейки могут содержать приведенные в таблице значения (см. табл. 4.2).

Таблица 4.2 - Значения в ячейках FAT

0000H	Свободный кластер
FFF0H - FFF6H	Зарезервированный кластер
FFF7H	Плохой кластер
FFF8H - FFFFH	Последний кластер в списке (признак конца файла)
0002H - FFEFH	Номер следующего кластера в списке

При этом для чтения файла с логического диска необходимо произвести следующие действия:

1. чтение дескриптора файла и определение номера первого кластера;
2. чтение кластера;
3. использование номера прочитанного кластера в качестве индекса в FAT для извлечения номера следующего кластера;
4. чтение кластера;
5. повторение действий, начиная с п.3 до тех пор, пока извлеченное из FAT значение не будет соответствовать концу файла.

В зависимости от размера ячейки в FAT различают FAT12, FAT16 и FAT32, что определяет максимальный размер логического диска отформатированного в данной файловой системе. При использовании FAT16 максимальный размер составляет около 2ГБ. При использовании FAT32 можно контролировать более 4 млрд. кластеров, что при использовании кластеров размер в 32 Кб, позволяет использовать диски размером около 4 млн. Гбайт.

Файловая система NTFS

Эта файловая система поддерживает объектно-ориентированные приложения, обрабатывая все файлы как объекты, которые имеют определяемые пользователем и системой атрибуты. NTFS позволяет задавать права доступа к отдельному файлу.

В отличие от разделов FAT все пространство тома NTFS представляет собой либо файл, либо часть файла. Основой структуры тома NTFS является Главная таблица файлов (Master File Table, MFT), которая содержит, по крайней мере, одну запись для каждого файла тома, включая одну запись для самой себя. Каждая запись имеет длину 2К.

Все файлы на томе NTFS идентифицируются номером файла, который определяется позицией файла в MFT. Каждый файл и каталог на томе NTFS состоит из набора атрибутов.

Базовая единица распределения дискового пространства для файловой системы NTFS – кластер. Размер кластера выражается в байтах и всегда равен целому количеству физических секторов. В качестве адреса файла NTFS использует номер кластера, а не физическое смещение в секторах или байтах.

Загрузочный сектор тома NTFS располагается в начале тома, а его копия – в середине тома. Загрузочный сектор состоит из стандартного блока параметров BIOS, количества секторов в томе, а также начального логического номера кластера основной копии MFT и зеркальной копии MFT.

Файлы NTFS состоят по крайней мере из следующих атрибутов:

- заголовок (H – header)
- стандартная информация (SI – standard information)
- имя файла (FN – file name)
- данные (Data)
- дескриптор безопасности (SD – security descriptor)

Небольшие файлы. Если файл имеет небольшой размер, то он может целиком располагаться внутри одной записи MFT размером 2К (рис. 4.3). Из-за того, что файл может иметь переменное количество атрибутов, а также из-за переменного размера атрибутов нельзя наверняка утверждать, что файл уместится внутри записи. Однако, обычно файлы размером менее 1500 байт помещаются внутри записи MFT.

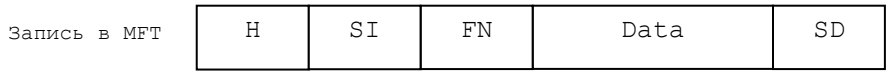


Рисунок 4.3 - Небольшие файлы

Большие файлы. Если файл не вмещается в одну запись MFT, то атрибут «данные» содержит виртуальный номер кластера для первого кластера каждого фрагмента данных (data run), а также количество непрерывных кластеров в каждом фрагменте (рис. 4.4).

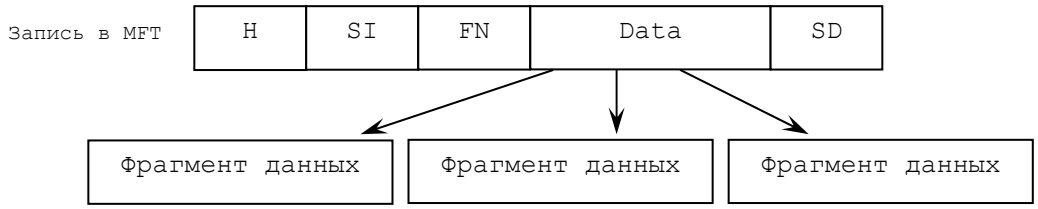


Рисунок 4.4 - Большие файлы

Очень большие файлы. Если файл настолько велик, что его атрибут данных не помещается в одной записи, то этот атрибут становится нерезидентным, то есть он находится в другой записи таблицы MFT, ссылка на которую помещена в исходной записи о файле (рис. 4.5). Эта ссылка называется внешним атрибутом (external attribute – EA). Нерезидентный атрибут содержит указатели на фрагменты данных.

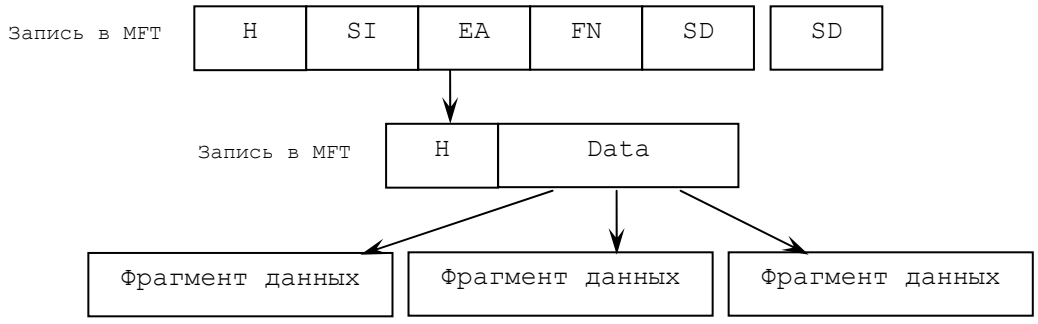


Рисунок 4.5 - Очень большие файлы

Сверхбольшие файлы. Для сверхбольших файлов внешний атрибут может указывать на несколько нерезидентных атрибутов (рис. 4.6). Кроме того, внешний атрибут, как и любой другой атрибут может храниться в нерезидентной форме, поэтому в NTFS не может быть атрибутов слишком большой длины, которые система не может обработать.

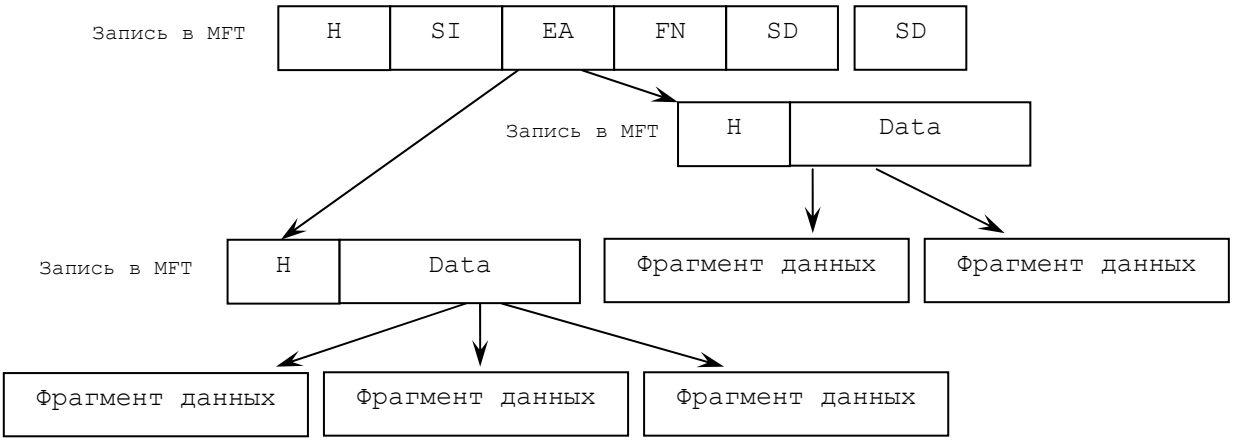


Рисунок 4.6 - Сверхбольшие файлы

Каждый каталог NTFS представляет собой один вход в таблицу MFT, который содержит список файлов специальной формы, называемый индексом (index). Индексы позволяют сортировать файлы для ускорения поиска, основанного на значении определенного атрибута. Обычно в файловых системах FAT используется сортировка файлов по имени. NTFS позволяет использовать для сортировки любой атрибут, если он хранится в резидентной форме.

Имеется две формы списка файлов.

Небольшие списки файлов. Если количество файлов в каталоге невелико, то список файлов может быть резидентным в записи в MFT, являющейся каталогом (рис. 4.7). В этом случае он называется небольшим каталогом. Небольшой список файлов содержит значения атрибутов файла. По умолчанию это имя файла, а также номер записи MTF, содержащей начальную запись файла.

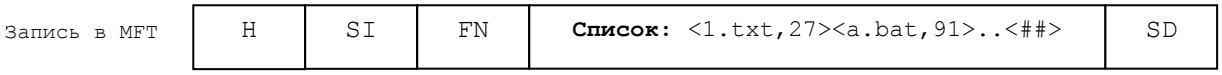


Рисунок 4.7 - Небольшие списки файлов (<##> – признак конца списка файлов)

Большие списки файлов. По мере того, как каталог растет, список файлов может потребовать нерезидентной формы хранения. Однако начальная часть списка всегда остается резидентной в корневой записи каталога в таблице MFT (рис. 4.8). Остальные части списка файлов размещаются вне MFT. Для их поиска используется специальный атрибут "размещение списка" (Index Allocation - IA), представляющий собой набор номеров кластеров, которые указывают на остальные части списка. Одни части списков являются листьями дерева, а другие являются промежуточными узлами, то есть содержат наряду с именами файлов атрибут Index Allocation, указывающий на списки файлов более низких уровней.

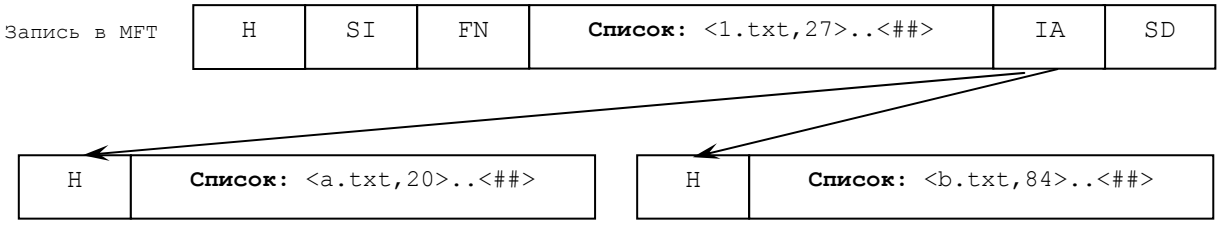


Рисунок 4.8 - Большие списки файлов (<##> – признак конца списка файлов)

Каждый атрибут файла или каталога NTFS состоит из полей: тип атрибута, длина атрибута, значение атрибута и, возможно, имя атрибута.

Имеется системные набор атрибутов (с фиксированными именами и кодами их типа) и атрибуты, определяемые пользователями. Существует два способа хранения атрибутов файла – резидентное хранение в записях таблицы MFT и нерезидентное хранение вне ее. Сортировка может осуществляться только по резидентным атрибутам.

Ниже приведены некоторые атрибуты.

- *Attribute List* - определяет список атрибутов, которые являются допустимыми для данного конкретного файла;
- *File Name* - содержит длинное имя файла, а также номер входа в MFT для родительского каталога; если этот файл содержится в нескольких каталогах, то у него будет несколько атрибутов типа "File Name"; этот атрибут всегда должен быть резидентным;
- *MS-DOS Name* - содержит имя файла в формате 8.3;
- *Version* - содержит номер последней версии файла;
- *Security Descriptor* - содержит информацию о защите файла: список прав доступа ACL и поле аудита (определяет регистрируемые операции);
- *Data* - содержит обычные данные файла;
- *Standard Information* - хранит всю остальную стандартную информацию о файле, например, время создания файла, время обновления и т.д.
- и др. атрибуты

Файловая система EXT2

Как и во многих файловых системах в Ext2 существует загрузочная область. На первичном разделе она содержит загрузочную запись – фрагмент кода, который инициирует процесс загрузки операционной системы при запуске. Всё остальное пространство раздела делится на блоки стандартного размера. Блок может иметь размер 1, 2 или 4 Кбайт. Блок является минимальной логической единицей дискового пространства (в других операционных системах такой блок называют кластером). Выделение места файлам осуществляется целыми блоками.

Блоки, в свою очередь, объединяются в группы блоков. Каждая группа блоков имеет одинаковое строение. Структура приведена в таблице 4.3.

Таблица 4.3 - Структура группы блоков

Суперблок (Superblock)
Описание группы блоков (Group Descriptors)
Битовая карта блока (Block Bitmap)
Битовая карта индексного дескриптора (Inode Bitmap)
Таблица индексных дескрипторов (Inode Table)
Блоки данных

Суперблок одинаков для всех групп, все же остальные поля индивидуальны для каждой группы. Суперблок хранится в первом блоке каждой группы блоков, является начальной точкой файловой системы, имеет размер 1024 байта и располагается по смещению 1024 байта от начала файловой системы. Копии Суперблока используются при восстановлении файловой системы после сбоев.

Информация в суперблоке служит для доступа к остальным данным на диске. В суперблоке определяется размер файловой системы, максимальное число файлов в разделе, объем свободного пространства и т.п. При старте операционной системы суперблок считывается в память, и все изменения файловой системы сначала записываются в копию суперблока, находящуюся в оперативной памяти, и только затем сохраняются на диске.

После суперблока следует являющееся массивом описание группы блоков (Group Descriptors).

Битовая карта блоков (Block Bitmap) – это структура, каждый бит которой показывает, отведен ли соответствующий ему блок какому-либо файлу. Если бит равен 1, то блок занят. Эта карта служит для поиска свободных блоков в тех случаях, когда надо выделить место под файл.

Битовая карта индексных дескрипторов (Inode Bitmap) выполняет аналогичную функцию по отношению к таблице индексных дескрипторов – показывает, какие дескрипторы заняты.

Блоки данных – в этой части файловой системы хранятся реальные данные файлов. Все блоки данных одного файла пытаются разместить в одной группе блоков.

Таблица индексных дескрипторов (Inode Table) содержит индексные дескрипторы файлов (i-узлы). Индексные дескрипторы файлов содержат информацию о файлах группы блоков. Каждому файлу на диске соответствует один и только один индексный дескриптор файла, который идентифицируется своим порядковым номером – индексом файла. Отсюда следует, что число файлов, которые могут быть созданы в файловой системе, ограничено числом индексных дескрипторов. В i-узле хранится информация, описывающая файл: режимы доступа к файлу, время создания и последней модификации, идентификатор пользователя и идентификатор группы создателя файла, описание блочной структуры файла и т.д.

Каталог, по сути, является специальным файлом, содержимое которого состоит из записей определенной структуры.

Система адресации данных позволяет находить нужный файл среди блоков на диске. В Ext2 система адресации реализуется полем `i_block` индексного дескриптора файла.

Поле `i_block` в индексном дескрипторе файла представляет собой массив из 15 адресов блоков. Первые 12 адресов в этом массиве представляют собой прямые ссылки на номера блоков, в которых хранятся данные из файла. Следующий адрес в этом массиве является косвенной ссылкой (адресом блока), в котором хранится список адресов следующих блоков с данными из этого файла. Следующий адрес в поле `i_block` индексного дескриптора указывает на блок двойной косвенной адресации (double indirect block). Этот блок содержит список адресов блоков, которые, в свою очередь, содержат списки адресов следующих блоков данных того файла, который задается индексным дескриптором.

Последний адрес в поле `i_block` индексного дескриптора задает адрес блока тройной косвенной адресации, то есть блока со списком адресов блоков, которые являются блоками двойной косвенной адресации.

Журналируемые файловые системы

Основная цель, которая преследуется при создании журналируемых файловых систем, состоит в том, чтобы обеспечить как можно большую вероятность быстрого восстановления системы после сбоев (например, после потери питания). После сбоя специализированная утилита операционной системы должна проверить все диски, которые не были корректно демонтированы, с целью восстановления потерянной информации. При современных объемах жестких дисков, исчисляемых десятками гигабайт, на проверку двух-трех таких дисков может уйти слишком много времени. Кроме того, нет гарантии, что все данные удастся восстановить.

В журналируемых файловых системах для решения этой проблемы применяют транзакции. Идея транзакции достаточно проста – существует набор связанных операций, называемых транзакцией, и эта группа операций является атомарной (неделимой). Таким образом, транзакция является успешной (завершенной) в том случае, если все операции, составляющие транзакцию, завершились успешно. Кроме того, система ведет журнал, в котором отражаются все действия с данными, и все изменения данных протоколируются. В случае сбоя на основании журнала можно вернуть систему в безошибочное состояние. В журнале сохраняются только мета-данные: индексные дескрипторы изменяемых файлов, битовые карты распределения свободных блоков и свободных индексных дескрипторов.

К журналируемым файловым системам относятся ReiserFS, XFS, JFS, GFS, ext3.

Ограничения файловых систем

В настоящее время существует почти две сотни разных файловых систем, каждые из которых имеют различные ограничения на использование. В таблице 4.4 для популярных файловых систем приведены некоторые ограничения.

3.2 Задание на лабораторную работу

Модернизировать программу, полученную в ходе решения лабораторной работы №3 и моделирующую файловую системы в соответствии с вариантом задания. При моделировании считать, что:

- объем внешней памяти, используемой для хранения файловой системы – не менее 64К;
- размер кластера или блока – 512 байт.
- модель файловая система должна выполнять операции создания, удаления, копирования, перемещения и переименования файлов.
- размер таблицы размещения файлов (FAT), размер суперблока, размер таблицы i-узлов и т.п. должен быть обосновано выбран в соответствии с заданными размерами кластера или блока и объемом внешней памяти.

Предусмотреть возможность визуального контроля за распределением пространства внешней памяти. Программа должна уметь загружать и хранить в соответствии с моделью заданной файловой системы набор скриптовых файлов, моделирующих процессы в рамках выполнения предыдущих лабораторных работ.

Варианты заданий

Вариант	**Файловая система**
1	Файловая система, использующая связанный список блоков внешней памяти. Предусмотреть журнализацию.
2	Файловая система FAT. Предусмотреть журнализацию.
3	Файловая система NTFS. Пердусмотреть возможность разграничения прав доступа пользователей.
4	Файловая система NTFS с журназизацией. Исключить возможность использования сверхбольших файлов. Предусмотреть кроме основных операций возможность создания жестких ссылок. Пердусмотреть возможность разграничения прав доступа пользователей.
5	Файловая система EXT2. Предусмотреть кроме основных операций возможность создания жестких ссылок. Пердусмотреть возможность разграничения прав доступа пользователей.
6	Файловая система EXT2 с журназизацией. Исключить тройную косвенную адресацию. Предусмотреть кроме основных операций возможность создания жестких ссылок. Пердусмотреть возможность разграничения прав доступа пользователей.
7	Файловая система FAT с журнализацией.Пердусмотреть возможность разграничения прав доступа пользователей (с этой целью модифицировать стандартный дискриптор файла).

Таблица 4.4 – Ограничения ФС
Примечание: LFN – драйвер поддержки длинных имен

Файловая система		Максимальная длина имени файла	Допустимые символы	Метаданные	Максимальная длина пути	Максимальный размер файла	Максимальный объем раздела	Максимальное количество файлов в разделе	Журналируемость
ISO 9660	**Level 1**	12 символов в формате 8.3	Буквы A-Z, цифры, подчеркивание (с помощью LFN - кириллица)	Нет	64 (не более 8 подкаталогов)	-	-	-	-
	Level 2	31 символ. Можно использовать больше одной точки в имени			255	-	-	-	-
	Joilet	64 символа. Можно использовать больше одной точки в имени	Все символы юникод, кроме NUL			2 Гбайт	-	-	-
	1999	207 символов. Можно использовать больше одной точки в имени					-	-	-
	Rock Ridge	255 символов		Да	-	-	-	-	-
UDF		254 символа			1023	16 Эбайт	16 Эбайт	-	-
FAT12		12 символов в формате 8.3. С помощью LFN – до 254 символов	Буквы A-Z, цифры, подчеркивание (с помощью LFN - кириллица)	Нет	Ограничена на уровне WinAPI до 260 символов. Собственно ФС не имеет архитектурных ограничений	16 Мбайт	16 Мбайт	4096 (2^{12})	Нет
FAT16		12 символов в формате 8.3. С помощью LFN – до 255 символов				4 Гбайт; 2 Гбайт в DOS, Win95/98/Me	4 Гбайт блоками по 64 Кбайт; 2 Гбайт в DOS, Win95/98/Me	65536 (2^{16}); 512 файлов или каталогов в корне раздела	
FAT32		255 символов				4 Гбайт минус 2 байта	2 Тбайт (или 32 Гбайт при создании встроенными	4194304 (2^{22}); 65534 файлов или подкаталогов в	

Файловая система	Максимальная длина имени файла	Допустимые символы	Метаданные	Максимальная длина пути	Максимальный размер файла	Максимальный объем раздела	Максимальное количество файлов в разделе	Журналируемость
						средствами WinXP)	каталоге	
NTFS		Все символы юникод, кроме < > / \ * ? :	Да	До 32767 символов. Имя каждого каталога и самого файла не более 255 символов	2 Тбайт для 32-битных ОС и 16 Тбайт для 64-битных	2 Тбайт для 32-битных ОС и 16 Тбайт для 64-битных	2 Тбайт для раздела основного типа и 256 Тбайт для динамического	Да
ext3		Все символы юникод, кроме NUL		Ограничения задаются ядром ОС, архитектурные отсутствуют	2 Тбайт	2 Тбайт	2 Тбайт для 32-битных ОС и 16 Тбайт для 64-битных	
ReiserFS					8 Тбайт для 32-битных ОС	8 Тбайт для 32-битных ОС	16 Тбайт	
HPFS					2 Гбайт / 4 Гбайт	2 Гбайт / 4 Гбайт	2 Тбайт; 64 Тбайт в OS/2	Нет
JFS (64 бита)					Для 32-битных ОС: 16 Тбайт при размере блока 4 Кбайт	Для 32-битных ОС: 16 Тбайт при размере блока 4 Кбайт	В ОС AIX – 4 Пбайт при размере блока 4 Кбайт. В Linux – 32 Пбайт	Да
XFS (64 бита)					Для 32-битных ОС: 64 Тбайт при размере блока 16 Кбайт	Для 32-битных ОС: 64 Тбайт при размере блока 16 Кбайт	Теоретический предел – 8,38 Пбайт. Практический лимит зависит от выбранной ОС	Да (для метаданных)

4 ЛАБОРАТОРНАЯ РАБОТА №5 «СРЕДСТВА СИНХРОНИЗАЦИИ ОПЕРАЦИОННЫХ СИСТЕМ ПРИ УПРАВЛЕНИИ РАСПРЕДЕЛЁННЫМИ РЕСУРСАМИ»

Цель работы: изучение основных методов организации синхронизации выполнения процессов в операционной системе при управлении распределёнными ресурсами, разработка программной модели, использующей средства синхронизации на примере средств операционной системы семейства Windows, в соответствии с вариантом задания.

4.1 Теоретические сведения

Изучение основных методов организации синхронизации выполнения процессов в операционной системе при управлении распределёнными ресурсами произведем на примере использования средств синхронизации выполнения многопоточных приложений в операционной среде Windows.

Синхронизация в распределенных системах

Алгоритмы взаимного исключения

Централизованный алгоритм

Наиболее очевидный и простой путь реализации взаимного исключения в распределенных системах - это применение тех же методов, которые используются в однопроцессорных системах. Один из процессов выбирается в качестве координатора (например, процесс, выполняющийся на машине, имеющей наибольшее значение сетевого адреса). Когда какой-либо процесс хочет войти в критическую секцию, он посылает сообщение с запросом к координатору, оповещая его о том, в какую критическую секцию он хочет войти, и ждет от координатора разрешение. Если в этот момент ни один из процессов не находится в критической секции, то координатор посылает ответ с разрешением. Если же некоторый процесс уже выполняет критическую секцию, связанную с данным ресурсом, то никакой ответ не посылается; запрашивавший процесс ставится в очередь, и после освобождения критической секции ему отправляется ответ-разрешение. Этот алгоритм гарантирует взаимное исключение, но вследствие своей централизованной природы обладает низкой отказоустойчивостью.

Распределенный алгоритм

Когда процесс хочет войти в критическую секцию, он формирует сообщение, содержащее имя нужной ему критической секции, номер процесса и текущее значение времени. Затем он посылает это сообщение всем другим процессам. Предполагается, что передача сообщения надежна, то есть получение каждого сообщения сопровождается подтверждением. Когда процесс получает сообщение такого рода, его действия зависят от того, в каком состоянии по отношению к указанной в сообщении критической секции он находится. Имеют место три ситуации:

- Если получатель не находится и не собирается входить в критическую секцию в данный момент, то он отсылает назад процессу-отправителю сообщение с разрешением.
- Если получатель уже находится в критической секции, то он не отправляет никакого ответа, а ставит запрос в очередь.
- Если получатель хочет войти в критическую секцию, но еще не сделал этого, то он сравнивает временную отметку поступившего сообщения со значением времени, которое содержится в его собственном сообщении, разосланном всем другим процессам. Если время в поступившем к нему сообщении меньше, то есть его собственный запрос возник позже, то он посылает сообщение-разрешение, в обратном случае он не посылает ничего и ставит поступившее сообщение-запрос в очередь.

Процесс может войти в критическую секцию только в том случае, если он получил ответные сообщения-разрешения от всех остальных процессов. Когда процесс покидает критическую секцию, он посылает разрешение всем процессам из своей очереди и исключает их из очереди.

Алгоритм Token Ring

Все процессы системы образуют логическое кольцо, т.е. каждый процесс знает номер своей позиции в кольце, а также номер ближайшего к нему следующего процесса. Когда кольцо инициализируется, процессу 0 передается так называемый токен. Токен циркулирует по кольцу. Он переходит от процесса n к процессу n+1 путем передачи сообщения по типу "точка-точка". Когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему самому войти в критическую секцию. Если да, то процесс входит в критическую секцию. После того, как процесс выйдет из критической секции, он передает токен дальше по кольцу. Если же процесс, принявший токен от своего соседа, не заинтересован во вхождении в критическую секцию, то он сразу отправляет токен в кольцо. Следовательно, если ни один из процессов не желает входить в критическую секцию, то в этом случае токен просто циркулирует по кольцу с высокой скоростью.

К сожалению все эти три алгоритма плохо защищены от отказов. В первом случае к краху приводит отказ координатора, во втором - отказ любого процесса (парадоксально, но распределенный алгоритм оказывается менее отказоустойчивым, чем централизованный), а в третьем - потеря токена или отказ процесса.

Поддержка многонитиевости (многопоточности)

Многозадачность является важнейшим свойством ОС. Для поддержки этого свойства ОС определяет и оформляет для себя те внутренние единицы работы, между которыми и будет разделяться процессор и другие ресурсы компьютера.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме. Однако задача, решаемая в рамках одного процесса, может обладать внутренним параллелизмом, который в принципе позволяет ускорить ее решение.

Для этих целей современные ОС предлагают использовать механизм многонитевой обработки (многопоточность). Мультипрограммирование реализуется на уровне нитей, и задача, оформленная в виде нескольких нитей в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей.

Многопоточность имеет большие преимущества:

- Увеличение скорости.
- Использование общих ресурсов - потоки одного процесса используют общую память и файлы. Нити, относящиеся к одному процессу, не настолько изолированы друг от друга, как процессы в традиционной многозадачной системе, между ними легко организовать тесное взаимодействие
- Экономия памяти (общие ресурсы) и времени (переключение осуществляется быстрее).
- Использование мультипроцессорных архитектур позволяет осуществлять параллельное выполнение.

Нити иногда называют облегченными процессами или мини-процессами. Процессы, в свою очередь, часто называют тяжеловесными. Действительно, нити во многих отношениях подобны процессам. Они имеют собственные:

- программный счетчик,
- стек,
- регистры,
- нити-потомки,

- состояние.

Однако различные нити в рамках одного процесса не настолько независимы, как отдельные процессы. Они разделяют:

- адресное пространство,
- глобальные переменные,
- открытые файлы,
- таймеры,
- семафоры,
- статистическую информацию.

Многонитевая обработка повышает эффективность работы системы по сравнению с многозадачной обработкой. Широкое применение находит многонитевая обработка в распределенных системах.

Существуют различные модели многопоточности – способы отображения пользовательских потоков[1] в потоки ядра[2]. Существуют следующие модели многопоточности:

- Модель много/один – отображение нескольких пользовательских потоков в один и тот же поток ядра. Используется в операционных системах, не поддерживающих множественные системные потоки.
- Модель один/один – взаимно-однозначное отображение каждого пользовательского потока в определенный поток ядра (например, Windows 95/98/NT/2000/XP/2003/2008/7; OS/2).
- Модель много/много – модель, допускающая отображение нескольких пользовательских потоков в несколько системных потоков. Такая модель позволяет ОС создавать большое число системных потоков.

Процессы и нити в распределенных системах

Способы организации вычислительного процесса с использованием нитей

Выделим три способа организации вычислительного процесса:

- *Нить-диспетчер* читает приходящие запросы на работу из почтового ящика системы. После проверки запроса диспетчер выбирает простаивающую (то есть блокированную) рабочую нить, передает ей запрос и активизирует ее, устанавливая, например, семафор, который она ожидает. Когда рабочая нить активизируется, она проверяет, может ли быть выполнен запрос с данными разделяемого блока кэша, к которому имеют отношение все нити. Если нет, она посылает сообщение к диску, чтобы получить нужный блок, и переходит в состояние блокировки, ожидая завершения дисковой операции. В этот момент происходит обращение к планировщику, в результате работы которого активизируется другая нить, возможно, нить-диспетчер или некоторая рабочая нить, готовая к выполнению.
- *Модель "команда"*. В этом случае все нити эквивалентны, каждая получает и обрабатывает свои собственные запросы. Иногда работы приходят, а нужная нить занята, особенно, если каждая нить специализируется на выполнении особого вида работ. В этом случае может создаваться очередь незавершенных работ. При такой организации нити должны вначале просматривать очередь работ, а затем почтовый ящик.
- *Конвейер*. Нити организованы в виде конвейера. В этом случае первая нить порождает некоторые данные и передает их для обработки следующей нити и т.д. Хотя эта организация и не подходит для файл-сервера, для других задач, например, задач типа "производитель-потребитель", это хорошее решение.

[1] С точки зрения пользователя и его программ, управление потоками реализовано через библиотеку потоков пользовательского уровня.

[2] Низкоуровневые потоки, в которые отображаются пользовательские потоки, называются потоками ядра. Они поддержаны и используются на уровне ядра ОС.

Вопросы реализации нитей

Существует два подхода к управлению нитями: статический и динамический. При статическом подходе вопрос, сколько будет нитей, решается уже на стадии написания программы или на стадии компиляции. Каждой нити назначается фиксированный стек. Этот подход простой, но негибкий. Более общим является динамический подход, который позволяет создавать и удалять нити оперативно по ходу выполнения.

Завершаться нити могут одним из двух способов: по своей инициативе, когда завершается работа, и извне.

Поскольку нити разделяют общую память, они могут (и, как правило, делают это) использовать ее для сохранения данных, которые совместно используются множеством нитей, таких, например, как буфер в системе "производитель-потребитель". Доступ к разделяемым данным обычно программируется с использованием критических секций, предотвращающих попытки сразу нескольких нитей обратиться к одним и тем же данным в одно и то же время. Критическая секция наиболее легко реализуется с использованием семафоров, мониторов и аналогичных конструкций.

Нити могут быть реализованы как в пользовательском пространстве (работают на базе прикладной системы, управляющей всеми операциями с нитями), так и в пространстве ядра.

4.2 Использование потоков в приложениях Windows

В настоящее время все современные операционные системы являются многозадачными, то есть имеют возможность выполнять несколько процессов (задач) одновременно (параллельно или псевдопараллельно), иными словами ОС предоставляет приложению некоторый интервал времени работы центрального процессора и в момент, когда приложение переходит к ожиданию сообщений или освобождает процессор, ОС передает управление другой задаче. В современных ОС семейства Windows кроме многозадачности реализована и многопоточность, при которой выполняемый процесс делится на несколько потоков (или нитей). Обычно многопоточность используется разработчиками программного обеспечения для:

- обхода медленных процессов (выполнение каких-либо действий одновременно с «медленными» процессами);
- организации поведения приложения (наделять каждую задачу, выполняемую одним приложением разными приоритетами выполнения);
- поддержки мультипроцессорной обработки (выполнение одного приложения на нескольких процессорах одного компьютера одновременно).

Общий обзор потоков

Поток (Thread) — это объект операционной системы, заключенный в процесс и реализующий какую-либо задачу. Каждое приложение (процесс) Win32 имеет по крайней мере один поток, который называется главным (основным, стандартным). Каждый процесс может содержать несколько потоков. Необходимо заметить, что процессорное время выделяется потокам, а не процессам. Время процессора выделяется квантами[3] (около 19 мс).

Существует два главных типа потоков:

- Асимметричные потоки – это потоки, решающие различные потоки и, как правило, не разделяющие совместные ресурсы.
- Симметричные потоки – это потоки, выполняющие одну и ту же работу, разделяют одни ресурсы и исполняют один код.

[3] Квант времени – это интервал, имеющийся в распоряжении потока до тех пор, пока время не будет передано в распоряжение другого потока.

Любой поток — это объект, получающий определенное процессорное время. Всякое приложение Windows является процессом ОС. Каждый процесс состоит хотя бы из одного потока, который называется главным. Вообще, Windows не ограничивает число потоков для каждого процесса.

Исключительные ситуации

Потоки одного приложения не являются настолько изолированными друг от друга как процессы в операционной системе: потоки, как правило, имеют минимум ресурсов, а родительский процесс распоряжается виртуальной памятью, кодом, данными, файлами и другими ресурсами ОС. Таким образом, при работе с глобальными переменными или другими разделяемыми ресурсами между потоками могут возникать исключительные ситуации – это гонки и тупики.

Ситуация гонок возникает, когда два или более потока пытаются получить доступ к общему ресурсу и изменить его состояние. Причем порядок исполнения потоков практически не зависит от самого программиста: планировщик ОС может запускать и останавливать их в любое время.

Ситуация тупиков имеет место, когда поток ожидает ресурс, который в данный момент принадлежит другому потоку, который, в свою очередь, ожидает освобождение ресурса, занимаемого первым потоком.

Таким образом, для избежания возникновения ситуаций гонок и тупиков необходимо использовать средства синхронизации потоков, которые в полной мере представлены в инструментарии Delphi.

Теоретические основы синхронизации выполнения процессов были подробно рассмотрены в рамках выполнения лабораторной работы №1.

Приоритеты потоков

Программист может управлять распределением процессорного времени между потоками, для чего существует понятие приоритета. Приоритет каждого потока складывается из двух составляющих:

- класса приоритета — приоритета процесса, породившего поток – может быть:
  - Класс реального времени - наивысший уровень приоритета (RealTime). Можно использовать только для обработки высокоскоростных потоков данных или на очень короткое время, так как компьютер не будет откликаться на события пользователя.
  - Класс с высоким приоритетом – приоритет высокого уровня (Hight). Также применяется достаточно редко.
  - Класс с нормальным приоритетом – стандартный приоритет, который имеют большинство приложений Windows (Normal).
  - Класс с фоновым приоритетом – низший приоритет для выполнения фоновых задач (Idle). Процессы с таким приоритетом запускаются только в том случае, если очереди Диспетчера задач нет других процессов.
- относительного приоритета — приоритета самого потока – может быть задан одним из семи значений:
  - Фоновый (TpIdle) – данный поток выполняется, когда система не занята и не выполняются никакие другие потоки. Windows не будет прекращать работу других потоков для выполнения потока, имеющего этот приоритет.
  - Низший (TpLowest) – данный поток занимает минимум процессорного времени.
  - Низкий (TpLower) – данный поток занимает немного больше процессорного времени, чем предыдущий приоритет.
  - Нормальный (TpNormal) – все потоки по умолчанию имеют данный приоритет.
  - Высокий (TpHigher) – данный поток имеет приоритет выше нормального.
  - Высший (TpHighest) – данный поток имеет приоритет выше, чем предыдущий.

- Наивысший (TpTimeCritical) - поток с данным приоритетом занимает максимум процессорного времени.

Общий приоритет потока получается сложением класса приоритета и относительного приоритета потока и в числовом эквиваленте представлен в таблице 5.1.

Таблица 5.1 – Числовой эквивалент приоритета потока

	TpIdle	TpLowest	TpLower	TpNormal	TpHigher	TpHighest	TpTimeCritical
Idle	-9	2	3	4	5	6	19
Normal	-7	6	7	8	9	10	23
Hight	-2	11	12	13	14	15	28
RealTime	9	22	23	24	25	26	39

Как видно из приведенной таблицы, есть широкий диапазон возможностей по управлению скоростью выполнения пока. Однако необходимо помнить, что лучше избегать использования потоков с большим приоритетом.

Вообще при использовании потоков необходимо остерегаться создавать слишком много потоков — это может привести к большой загруженности операционной системы и процессора (рекомендуемое ограничение числа активных потоков в одном процессе — 16); использовать синхронизацию в случае, когда несколько потоков пытаются получить доступ к одному ресурсу; методы, которые обращаются к объектам VCL и изменяют содержимое формы, должны вызываться из главного VCL-потока или использовать метод синхронизации потоков.

Использование потоков средствами Delphi

Класс TThread

В большинстве случаев для реализации многопотокового приложения достаточно использовать специализированный класс TThread, являющийся конструкцией Delphi и соответствующий потоку ОС. Данный класс является абстрактным, поэтому экземпляр данного класса создать нельзя, а можно сначала создать потомка класса TThread, а только после этого создать экземпляр класса-потомка TThread.

Одним из самых простых способов создания потомка класса TThread является использование диалогового окна New Items (вызывается через меню: File|New|Other). В появившемся диалоговом окне на закладке New надо выбрать пункт Thread Object (Объект потока). В открывшемся диалоговом окне New Thread Object напишите имя нового объекта потока (например TSimpleThread). После всего этого Delphi создаст новый модуль и в окне редактора кода появится новая вкладка. В новом модуле, сгенерированном Delphi, вы можете увидеть код - заготовку для нового объекта потока:

```
unit Unit2;
interface
uses Classes;
type
     TSimpleThread = class(TThread)
     private
     protected
          procedure Execute; override;
     end;
implementation
procedure TSimpleThread.Execute;
begin
     { Place thread code here }
```

```
end;
end.
```

В автоматически сгенерированном файле модуля можно инициализировать поток и заполнить метод Execute объекта потока, разместив там функции и процедуры, которые и будут выполняться во вновь созданном потоке. Таким образом, основным методом, в котором выполняются действия в потоке, является метод Execute. Для примера предположим, что если необходимо выполнить сложные вычисления, то реализовать их надо внутри этого метода:

```
procedure TSimpleThread.Execute;
var
  i: Integer;
begin
  FreeOnTerminate := True;
  Resultat := 0;
  for i := 1 to 20000000 do begin
    if Terminated then Break;
    Resultat := Resultat + Round(Abs(Sin(Sqrt(i))));
  end;
end;
```

В принципе, указанные вычисления не имеют никакого смысла – просто очень много вычислений. Однако в методе Execute присутствует строка

FreeOnTerminate := True;

которая устанавливает значение FreeOnTerminate свойства в True. Данное свойство управляет завершением работы потока. Если свойство FreeOnTerminate равно True, то поток освобождает занимаемые ресурсы (память) по завершению своей работы, если же свойство FreeOnTerminate равно False, то программист сам явно должен завершить работу потока. Имеется возможность прекращения выполнения одного потока из другого потока подачей команды о прекращении с помощью вызова метода Terminate. Данный метод переводит свойство потока Terminated в True, а в теле потока можно явно проверить значение этого свойства и выполнить соответствующие действия (например, как в рассмотренном ранее примере, осуществить прерывание работы цикла, что приведет к прекращению выполнения потока).

Существует еще один способ аварийного завершения работы потока с помощью функции Win32API

```
function TerminateThread(hTread: THandle; dwExitCode: DWORD);
```

например,

```
TerminateThread(SimpleThread.Handle, 0);
```

но использовать данную функцию следует с большой осторожностью и в крайних случаях. Кстати, переменная Resultat, использованная в примере, - это поле класса TSimpleThread типа Integer.

Однако приложение еще не готово. Необходимо перед запуском потока создать его, вызвав конструктор Create, например, в обработчике события щелчка по кнопке, расположенной на форме.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SimpleThread: TSimpleThread;
begin
  SimpleThread := TSimpleThread.Create(False);
end;
```

В качестве аргумента конструктор получает параметр CreateSuspended: Boolean. Если его значение равно False (как в примере), то вновь созданный поток начинает выполняться сразу же, а конструктор завершается. Если же его значение равно True, то вновь

созданный поток не начинает выполняться до тех пор, пока не будет сделан вызов метода Resume этого же экземпляра потока, например:

```
SimpleThread.Resume;
```

Данный метод служит для возобновления выполнения потока после остановки. Приостановить выполнения какого-либо потока с возможностью повторного запуска в последствии, в свою очередь, можно с помощью метода Suspend:

```
SimpleThread.Suspend;
```

Синхронизация с подпрограммами библиотеки VCL

Метод Synchronize

В рассмотренном примере с помощью потока происходят какие-то вычисления, однако вывести какие-либо промежуточные результаты нельзя, потому что работа с визуальными компонентами, с помощью которых производится вывод информации на форме, производится только из главного потока приложения. Это связано с тем, чтобы избежать ситуации гонок при обращении потоков к визуальным компонентам (визуальные компоненты могут работать только при условии доступа к ним лишь одного потока в каждый момент времени). Для исключения возникновения ситуации гонок при обращении к визуальным компонентам применяется метод Synchronize экземпляра потока, который дает гарантию, что к каждому визуальному компоненту одновременно имеет доступ только один поток. В качестве аргумента, передаваемого в метод Synchronize, выступает имя метода класса-потока, который производит обращение к визуальным компонентам. Этот метод не должен иметь никаких параметров и не должен возвращать никаких значений. Например, для условий нашего примера напишем процедуру Show, которая будет отображать результат вычислений на форме в Label, а процедуру вычислений в ProgressBar. Вид приложения на рисунке 5.1.

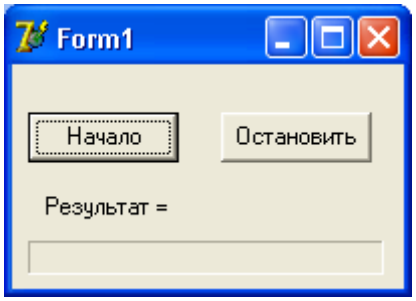


Рисунок 5.1 – Интерфейс приложения

Исходный текст модуля с потоком:

```
unit Unit2;
interface
uses  Classes;

const
  max=200000;

type
  TSimpleThread = class(TThread)
    Resultat: Integer;
    ii: Integer;
    private
    protected
    procedure Show;
    procedure Execute; override;
  end;
```

```
implementation
uses Unit1, SysUtils;

procedure TSimpleThread.Execute;
var
  i: Integer;
begin
  FreeOnTerminate := True;
  Resultat := 0;
  ii := 0;
  for i := 1 to max do begin
    if Terminated then Break;
    inc(ii);
    Resultat := Resultat + Round(Abs(Sin(Sqrt(i))));
    Synchronize(Show);
  end;
end;

procedure TSimpleThread.Show;
begin
  Form1.Label1.Caption := 'Результат = ' + IntToStr(Resultat);
  Form1.ProgressBar1.Position := Round(ii/max*100);
end;
end.
```

При запуске приложения и нажатии на кнопку «Начало» будет производиться вычисления, а результаты непрерывно выводиться на форму.

Однако данный метод не может использоваться в консольных приложениях. В этом случае надо использовать другие методы для исключения ситуации гонок, о которых будет рассказано далее.

Средства синхронизации потоков

Проиллюстрируем проблему синхронизации на следующем примере, в котором появляется ситуация гонок. Допустим, есть приложение, которое рассчитывает период математического маятника в зависимости от начального угла наклона нити маятника. Причем при расчете производится интегрирование, с увеличением точности расчетов растет и время самих расчетов, поэтому каждый расчет (для каждого начального угла) производится в отдельном потоке, результаты вычислений отображаются в Memo, расположенного в основном модуле:

```
unit Unit2;
interface
uses Classes;

type
  TCalcThread = class(TThread)
    Value, Res: Real;
    Count: Integer;
    constructor Create(v: Real; C: Integer);
  private
  protected
    procedure Show;
    procedure Execute; override;
  end;
```

```
  function Period(value: real; count: Integer): real; stdcall;

implementation
uses Unit1, SysUtils;

function Period(value: real; count: Integer): real; external
'ND.dll' name 'Period';

constructor TCalcThread.Create(v: Real; C: Integer);
begin
  Value := v;
  Count := c;
  inherited Create(False);
end;

procedure TCalcThread.Execute;
begin
  FreeOnTerminate := True;
  Res := Period(Value, Count);
  Synchronize(Show);
end;

procedure TCalcThread.Show;
begin
  Form1.Memo1.Lines.Add(FloatToStr(Value) + ';' + IntTo-
Str(Count) + ' -> ' + FloatToStr(Res));
end;
end.
```

В представленном примере необходимо обратить внимание на тот факт, что программист может переопределять конструктор потока и производить там некие действия.

В данном случае расчеты вынесены в динамическую библиотеку, так как для примера они не важны:

```
library ND;
uses SysUtils, Classes;

function f(fi, fi0: real): real;
begin
  result := 1 / sqrt(cos(fi) - cos(fi0));
end;

function Period(value: real; count: Integer): real; stdcall;
var
  fi0, h, i, sum: real;
begin
  fi0 := value / 180 * pi;
  h := fi0 / count;
  i := h/2;
  sum := 0;
  while i < fi0 - h do begin
    sum := sum + (f(i, fi0) * h);
    i := i + h;
  end;
```

```
  Result := (4/sqrt(2*(9.81)))*sum;
end;

exports Period;
begin
end.
```

В основном модуле создается заданное количество потоков, результаты выводятся по мере окончания расчетов в потоках:

```
unit Unit1;
interface
uses  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, Unit2, Spin, Math, ComCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Memo1: TMemo;
    SpinEdit1: TSpinEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    SpinEdit2: TSpinEdit;
    Label4: TLabel;
    SpinEdit3: TSpinEdit;
    procedure Button1Click(Sender: TObject);
  private
  public
  end;

var
  Form1: TForm1;
  CalcThread: TCalcThread;

implementation

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Randomize;
  Memo1.Clear;
  Memo1.Lines.Add('Угол;Порядок -> Период');
  for i := 1 to SpinEdit3.Value do begin
    CalcThread   :=   TCalcThread.Create(SpinEdit2.Value   *   i,
Round((Random(10) + 1) * power(10, SpinEdit1.Value)));
  end;
end;

end.
```

В результате работы приложения (рисунок 5.2) видим, что задано изменение угла наклона от 10 до 100 град. с шагом 10 град., а вывод результатов производится не по порядку.

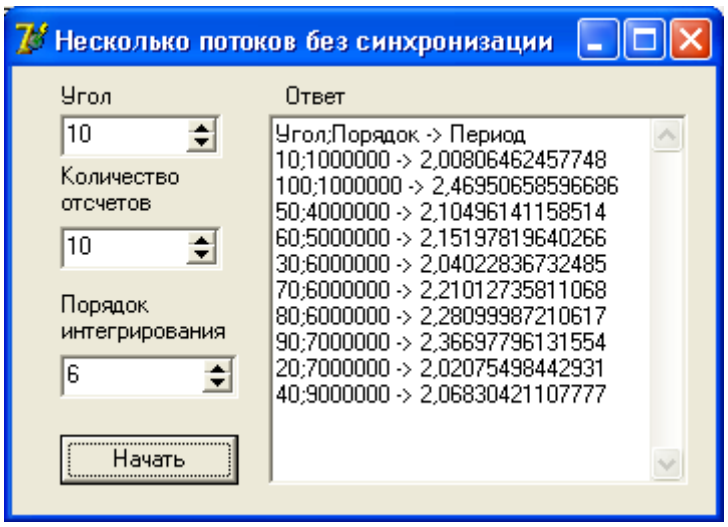


Рисунок 5.2 – Результат работы программы

Это происходит в связи с тем, что работа потоков производится псевдопараллельно и не синхронно, работа некоторых потоков прерывается по окончанию кванта процессорного времени. Решение этой проблемы лежит в синхронизации работы потоков.

Как класс TThread, так и программный интерфейс Win32 (API), предоставляют богатый набор инструментов, которые могут понабиться для организации совместной работы потоков. Главные понятия для понимания механизмов синхронизации – функции ожидания и объекты ожидания. В API предусмотрен ряд функций, позволяющих приостановить выполнение вызвавшего его функцию потока вплоть до того момента, как будет изменено состояние какого-то объекта, называемого объектом ожидания. К возможным вариантам относится четыре объекта ОС, разработанных специально для синхронизации: событие, взаимное исключение, семафор и таймер. К ним может быть добавлена и критическая секция.

Простейший способ

Если создаваемый поток не взаимодействует с ресурсами других потоков и не обращается к визуальным компонентам, а производит, например, просто много сложных вычислений, то для организации синхронизации можно воспользоваться методом WaitFor, инкапсулированном в класс TThread. Данный метод позволяет одному потоку дождаться момента, когда завершится другой поток. Например, если внутри потока FirstThread написать код:

```
Code := SecondThread.WaitFor;
```

То поток FirstThread останавливается до момента завершения потока SecondThread.

Или:

```
//Создаем поток
CalcThread := TCalcThread.Create(False);
//пока поток считает, выполняем другую работу
DoSomeWork;
//Ждем завершение потока
CalcThread.WaitFor;
```

Однако использовать такую схему нельзя в случае, если во время своей работы поток CalcThread обращается к визуальным компонентам посредством метода Syncronize. В этом случае поток ждет главный поток для обращения к визуальным компонентам, а тот, в свою очередь, его – получается тупиковая ситуация.

В нашем примере можно добавить вызов соответствующего метода после создания потока в обработчике события щелчка по кнопке:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
```

```
begin
  Randomize;
  Memo1.Clear;
  Memo1.Lines.Add(Угол;Порядок -> Период);
  for i := 1 to SpinEdit3.Value do begin
    CalcThread   :=   TCalcThread.Create(SpinEdit2.Value   *   i,
Round((Random(10) + 1) * power(10, SpinEdit1.Value)));
    CalcThread.WaitFor;
  end;
end;
```

Однако большого смысла от этого нет, потому что хоть и будут выводиться результаты последовательно, но и считаться они будут тоже последовательно, таким образом, потоки будут создаваться последовательно и не будет никакого выигрыша в скорости вычисления, кроме возможности формы реагировать на действия пользователя (нет эффекта зависания). Еще раз напомню, что использование этого метода синхронизации может привести к тупиковой ситуации.

Использование события

Объект типа событие является простейшим примитивом для решения задачи синхронизации. Объект типа событие реализован с помощью класса TEvent, описанный в модуле SYNCOBJS.PAS, может принимать одно из двух состояний: активное или пассивное.

Когда объект ожидания находится в активном состоянии, его видят многие потоки одновременно. В результате такой объект можно использовать для управления работой сразу многих потоков.

В классе TEvent для перевода объекта в активное состояние используется метод SetEvent, а для перевода в пассивное состояние – ResetEvent.

Для ожидания перевода события в активное состояние используется метод WaitFor:

```
type TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrEr-
ror);
function WaitFor(Timeout: DWORD): TWaitResult;
```

Данный метод имеет один параметр Timeout, задающий количество миллисекунд, в течение которых объект ожидает активизацию события. Этот метод возвращает значение wrSignaled в том случае, если активизация события произошла, и wrTimeout — в противном случае. Если в качестве параметра Timeout передать значение infinite, то ожидание активизации события будет бесконечно долгим.

Перед использованием объекта, его необходимо создать, воспользовавшись функцией CreateEvent:

```
CreateEvent (lpEventAttributes, bManualReset, blnitialState, ip-
Name);
```

где:

- lpEventAttributes —дескриптор безопасности для нового события. В Windows NT, если данный параметр равен nil, то событие получает дескриптор безопасности, установленный по умолчанию. В Windows 9x данный параметр игнорируется;
- bManualReset — определяет ручной или автоматический сброс состояния события. Если значение данного параметра true, то необходимо использовать функцию ResetEvent для ручного перевода события в пассивное состояние. Если значение false, то Windows автоматически осуществит такой перевод;
- bInitialState —начальное состояние события (активное или пассивное). Если значение параметра true, то состояние активное, иначе — пассивное;
- LpName —имя события для обращения к нему из других потоков. Имя может содержать любые символы, кроме обратной косой черты (\). Имя чувствительно к регистру букв. В случае, когда имя повторяет имя созданного события, происходит обращение

к уже существующему событию. Если значение имени равно nil, то событие создается без имени. Если событие имеет такое же имя, как семафор, мьютекс или другой объект, генерируется ошибка error_invalid_handle.

В нашем примере, используя события, можно было бы произвести синхронизацию следующим образом:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, Unit2, Spin, Math, ComCtrls,
SyncObjs;

type
  TForm1 = class(TForm)
    ...
  end;
  TCalcThread = class(TThread)
    Value, Res: Real;
    Count: Integer;
    constructor Create(v: Real; C: Integer);
  private
  protected
    procedure Show;
    procedure Execute; override;
  end;
  function Period(value: real; count: Integer): real; stdcall;

var
  Form1: TForm1;
  CalcThread: TCalcThread;
  Event: TEvent;
  res: TWaitResult;

implementation

function Period(value: real; count: Integer): real; external
'ND.dll' name 'Period';

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Randomize;
  Memo1.Clear;
  Memo1.Lines.Add('Угол;Порядок -> Период');
  Event := TEvent.Create(nil, True, False, 'event');
  for i := 1 to SpinEdit3.Value do begin
    Event.ReSetEvent;
    CalcThread := TCalcThread.Create(SpinEdit2.Value * i,
Round((Random(10) + 1) * power(10, SpinEdit1.Value)));
    res := Event.WaitFor(10000);
  end;
  Event.Free;
```

```
end;

constructor TCalcThread.Create(v: Real; C: Integer);
begin
  Value := v;
  Count := c;
  inherited Create(False);
end;

procedure TCalcThread.Execute;
begin
  FreeOnTerminate := True;
  Res := Period(Value, Count);
  Synchronize(Show);
  Event.SetEvent;
end;

procedure TCalcThread.Show;
begin
  Form1.Memo1.Lines.Add(FloatToStr(Value)   +   ';'   +   IntTo-
Str(Count) + ' -> ' + FloatToStr(Res));
end;
end.
```

Однако на практике такой способ чаще всего не работает, потому что возникает тупиковая ситуация, так как главный поток ждет, пока не произойдет событие, а пользовательский поток не может работать. Для иллюстрации применения событий рассмотрим другой пример:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, SyncObjs;

type
  TForm1 = class(TForm)
    ...
  end;
  TSimpleThread = class(TThread)
  private
  protected
    procedure Execute; override;
    procedure Show;
  end;

var
  Form1: TForm1;
  SimpleThread: TSimpleThread;
  e: TEvent;
  res: TWaitResult;

implementation

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  Memo1.Clear;
  SimpleThread := TSimpleThread.Create(false);
end;

procedure TSimpleThread.Execute;
begin
  e := TEvent.Create(nil, true, false, 'test');
  repeat
    e.ResetEvent;
    res := e.WaitFor(INFINITE);
    Synchronize(Show);
  until Terminated;
  e.Free;
end;

procedure TSimpleThread.Show;
begin
    case res of
    wrSignaled: Form1.Memo1.Lines.Add('Событие произошло');
    wrTimeout: Form1.Memo1.Lines.Add('Событие не произошло');
    wrAbandoned: Form1.Memo1.Lines.Add('Событие Abandoned');
    wrError: Form1.Memo1.Lines.Add('Ошибка');
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  e.SetEvent;
end;

end.
```

В данном примере только в случае нажатия пользователем второй кнопки происходит событие и появится надпись в Memo1, иначе поток приостановит свою работу и будет ждать события.

Критическая секция

Критические секции представляют собой один из самых простых способов синхронизации потоков. Критическая секция — это область глобальной памяти (некоторый участок памяти), которую необходимо защищать от одновременного использования её разными потоками.

Для использования критической секции в приложении необходимо создать глобальный экземпляр класса TCriticaiSection, реализация которого описана в модуле SYNCOBJS.PAS. Данный класс имеет два метода для работы с критической секцией: Acquire (или Enter) и Release (или Leave).

Метод Acquire (Enter) связывает критическую секцию с потоком, вызвавшим данный метод, блокируя доступ к этой секции всем остальным потокам.

Метод Release (Leave) снимает блокировку и позволяет другим потокам обращаться к критической секции.

Для примера рассмотрим небольшую программу, которая содержит глобальную переменную критической секции Section. Данная переменная блокирует доступ к глобальным переменным x и y.

```
var
```

```
     Section: TCriticalSection;
...
Section := TCriticalSection.Create;
...
Section.Acquire;
try
     Y := sin(X);
finally
     Section.Release;
end;
...
Section.Free;
```

Когда другие потоки доходят до оператора захвата секции и обнаруживают, что она уже захвачена, они приостанавливаются вплоть до освобождения секции. Высвобождение секции лучше реализовать в конструкции try ... finally для обеспечения бóльшей надежности.

Мьютексы (взаимные исключения)

Данный тип объекта ожидания разрешает доступ только одному потоку в определенный момент времени. Мьютекс сигнализирует, когда он не принадлежит потоку и прекращает сигнализировать, когда он принадлежит потоку. После того как поток перестает обращаться к мьютексу, любой другой поток получает возможность доступа к нему. Данный объект полезен в случае обращения к памяти с целью ее изменения различными потоками.

Для того чтобы воспользоваться мьютексами можно либо напрямую вызывать WinAPI-функции, либо воспользоваться специальным классом TMutex, который описан в файле IPCTHRD.PAS, распространяемого вместе с примером в каталоге IPCDEMOS (C:\Program Files\Borland\Delphi7\Demos). В принципе, указанный класс примера инкапсулирует все методы и свойства и использует в своей работе WinAPI-функции.

Мьютекс в WinAPI

Для использования мьютекса необходимо сначала объявить его дескриптор в секции объявления глобальных переменных:

```
var
  Form1: TForm1;
  hMutex: THandle;
```

Затем перед созданием потоков, которые должны будут ориентироваться на мьютекс, необходимо создать его, вызвав функцию:

```
  hMutex := CreateMutex(nil, False, nil);
```

где

- первый параметр (lpMutexAttributes) – определяет, может ли обработчик наследоваться потоками-потомками. В Windows NT, при равным nil, мьютекс получает дескриптор безопасности, установленный по умолчанию. В Windows 9x данный параметр функции игнорируется;
- второй параметр (bInitialOwner) – определяет начального владельца мьютекса. Если данный параметр равен true, то владельцем мьютекса становится вызвавший его поток. В противном случае, владелец мьютекса не определен;
- третий параметр (lpName) — определяет имя мьютекса. Имя предназначено для обращения к мьютексу из других процессов. Имя может содержать любые символы, кроме обратной косой черты (\). Имя чувствительно к регистру букв. В случае, когда имя повторяет имя уже созданного мьютекса, происходит обращение к уже существующему мьютексу. Если значение имени равно nil, то мьютекс создается без имени. Имя не должно совпадать с именами события, семафора или другого объекта.

Для управления входом потоков в синхронизируемый блок используется функция WaitForSingleObject с помощью, например, такой структуры:

```
if WaitForSingleObject(hMutex, INFINITE) = WAIT_OBJECT_0 then
begin
    ...
end;
```

Эта функция предназначена для перевода текущего потока в состояние ожидания, пока объект API, заданный первым параметром, не станет доступным. При этом состояние ожидания может продлиться вплоть до истечения интервала времени, заданного в миллисекундах вторым параметром (в нашем случае значение INFINITE указывает, что ждать следует до тех пор, пока объект не станет доступным). Данная функция может возвращать одно из трех значений:

- WAIT_ABANDONED – мьютекс считается покинутым (поток завершился, не освободив его), мьютекс определяется как недоступный;
- WAIT_OBJECT_0 – состояние объекта определяется как доступное;
- WAIT_TIMEOUT – установленный интервал времени истек, состояние объекта определяется как недоступное.

Для того чтобы сделать мьютекс доступным для других потоков (освободить его при окончании условия взаимного исключения), необходимо вызвать функцию:

```
ReleaseMutex(hMutex);
```

где в качестве параметра передать дескриптор мьютекса.

В конце работы приложения необходимо высвободить объект API, для чего можно воспользоваться функцией:

```
CloseHandle(hMutex);
```

Класс TMutex

Для использования мьютекса необходимо сначала объявить его в секции объявления глобальных переменных:

```
var
  Form1: TForm1;
  hMutex: TMutex;
```

Затем перед созданием потоков, которые должны будут ориентироваться на мьютекс, необходимо создать его, вызвав функцию:

```
  hMutex := TMutex.Create('test');
```

где в качестве параметра передается имя мьютекса. Имя предназначено для обращения к мьютексу из других процессов. Имя может содержать любые символы, кроме обратной косой черты (\). Имя чувствительно к регистру букв. В случае, когда имя повторяет имя уже созданного мьютекса, происходит обращение к уже существующему мьютексу. Если значение имени равно nil, то мьютекс создается без имени. Имя не должно совпадать с именами события, семафора или другого объекта.

Для управления входом потоков в синхронизируемый блок используется функция Get с помощью, например, такой структуры:

```
if hMutex.Get(INFINITE) then begin
    ...
end;
```

Эта функция предназначена для перевода текущего потока в состояние ожидания, пока мьютекс не станет доступным. При этом состояние ожидания может продлиться вплоть до истечения интервала времени, заданного в миллисекундах параметром (в нашем случае значение INFINITE указывает, что ждать следует до тех пор, пока объект не станет доступным). Данная функция может возвращать логическое значение: true – мьютекс доступен, иначе – не доступен.

Для того чтобы сделать мьютекс доступным для других потоков (освободить его при окончании условия взаимного исключения), необходимо вызвать метод:

```
hMutex.Release;
```

В конце работы приложения необходимо высвободить мьютекс, для чего можно воспользоваться деструктором:

```
hMutex.Free;
```

Семафоры

Объект синхронизации типа семафор подобен взаимному исключению, он, в отличие от последнего, может устанавливаться на доступ к нему определенного числа потоков. При этом все потоки, которые вновь обращаются к семафору, но, при этом, их количество уже превышает предельно допустимое количество, - приостанавливаются. При отключении присоединенных к семафору потоков "ждущие" потоки подключаются к нему. Семафоры полезны при контролировании доступа к разделяемым ресурсам, которые могут поддерживать ограниченное число пользователей.

Семафоры используются аналогично рассмотренных ранее мьютексов с помощью WinAPI-функций.

Для использования семафора необходимо сначала объявить его дескриптор в секции объявления глобальных переменных:

```
var
  Form1: TForm1;
  hSem: THandle;
```

Затем перед созданием потоков, которые должны будут ориентироваться на семафор, необходимо создать его, вызвав функцию:

```
  hSem := CreateSemaphore(nil, 10, 10, nil);
```

где

- первый параметр (lpSemaphoreAttributes) – определяет дескриптор безопасности для нового семафора, данный параметр игнорируется для Windows 9x. Если параметр равен nil, то семафор получает дескриптор безопасности по умолчанию;
- второй параметр (lInitialCount) — определяет начальное значение счетчика для семафора. Оно должно быть больше нуля и меньшим или равным значению третьего параметра lMaximumCount;
- третий параметр (lMaximumCount) — определяет максимальное значение счетчика для семафора (максимальное число потоков). Оно должно быть больше нуля;
- четвертый параметр (lpName) — определяет имя семафора. Требования к имени семафора аналогично рассмотренным ранее.

Для управления входом потоков в синхронизируемый блок используется функция WaitForSingleObject с помощью, например, такой структуры:

```
if WaitForSingleObject(hSem, INFINITE) = WAIT_OBJECT_0 then
begin
     ...
end;
```

Эта функция предназначена для перевода текущего потока в состояние ожидания, пока объект API, заданный первым параметром, не станет доступным. При этом состояние ожидания может продлиться вплоть до истечения интервала времени, заданного в миллисекундах вторым параметром (в нашем случае значение INFINITE указывает, что ждать следует до тех пор, пока объект не станет доступным). Данная функция может возвращать одно из трех значений:

- WAIT_ABANDONED – мьютекс считается покинутым (поток завершился, не освободив его), мьютекс определяется как недоступный;
- WAIT_OBJECT_0 – состояние объекта определяется как доступное;
- WAIT_TIMEOUT – установленный интервал времени истек, состояние объекта определяется как недоступное.

Для того, чтобы сделать семафор доступным для других потоков (освободить его при окончании условия взаимного исключения), необходимо вызвать функцию:
```
ReleaseSemaphore(hSem);
```
где в качестве параметра передать дескриптор семафора.

В конце работы приложения (или в случае, если семафор больше не нужен) необходимо высвободить объект API, для чего можно воспользоваться функцией:
```
CloseHandle(hSem);
```

Инструменты работы с потоками

Определение времени, занимаемого потоком

В многопоточной операционной системе узнать время, необходимое для выполнения каких-либо операций, например выполнения потоков, может использоваться функция GetThreadTimes, которая предоставляет информацию о времени работы потока:
```
function GetThreadTimes (hThread: THandle; var lpCreationTime, lpExitTime, lpKernelTime, lpUserTime: TFileTime): BOOL; stdcall;
```
где:

- hThread — дескриптор потока, для которого определяется время выполнения;
- lpCreationTime — время создания потока;
- lpExitTime — время завершения работы потока (определяется только при завершении работы потока);
- lpKernelTime — время, затраченное ОС для выполнения собственного кода;
- lpUserTime — время, затраченное на выполнение кода приложения.

Последние четыре параметра имеют тип TFileTime:
```
type
  TFiieTime = record
    dwLowDateTime: DWORD;
    dwHighDateTime: DWORD;
  end;
```
Указанные элементы записи в сумме образуют 64-разрядное значение. Это значение определяет количество интервалов (каждый интервал равен 100 наносекунд), которые прошли начиная с 1 января 1601 года.

Для работы с типом TFileTime удобно использовать функцию приведения данного типа к типу Int64, например: Int64(CreationTime). Но не забудьте о том, что время представлено в наносекундах и его надо перевести в секунды.

Настройка приоритета

Установка приоритета выполнения потока производится с помощью свойства Priority. Возможные значения данного свойства были рассмотрены ранее. Например,
```
ASomeThread.Priority := tpHighest;
```

Порождение дочернего процесса

Для создания дочернего процесса можно воспользоваться функцией
```
function CreateProcess(lpApplicationName: PChar; lpCommanLine: PChar; lpProcessAttributes, lpTreadAttributes: PSecurityAttributes; bInheritedHandles: BOOL; dwCreationFlags: DWORD; lpEnvironment: Pointer; lpCurrentDirectory: PChar; const lpStartupInfo: TStartupInfo; var lpProcessInformation: TProcessInformation): BOOL;
```
Первые два параметра – имя запускаемого приложения и передаваемые ему в командной строке параметры. Параметр dwCreationFlags содержит флаги, определяющие способ создания нового процесса и его будущий приоритет (CREATE_NEW_CONSOLE, NORMAL_PRIORITY_CLASS и другие, описаны в модуле Windows). Структура TStartupInfo содержит сведения о размере, цвете, положении окна создаваемого приложе-

ния (например, если поле wShowWindow установить равным SW_SHOWNORMAL, то окно будет визуализировано с нормальным размером). На выходе функции заполняется структура lpProcessInformation, в которой возвращаются дескрипторы и идентификаторы созданного процесса и его первичного потока (например, дескриптор процесса находится в поле lpProcessInformation.hProcess).

4.3 Задание на лабораторную работу

Реализовать программу, моделирующую управляющую систему в соответствии с вариантом задания. Предусмотреть возможность визуального контроля за работой системы.

Варианты заданий

Вариант	Управляющая система
1	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации взаимного исключения в распределенной системе использовать централизованный алгоритм. Для организации синхронизации использовать аппарат событий.
2	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации взаимного исключения в распределенной системе использовать централизованный алгоритм. Для организации синхронизации использовать аппарат критической секции.
3	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации взаимного исключения в распределенной системе использовать централизованный алгоритм. Для организации синхронизации использовать мьютексы WinAPI.
4	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации взаимного исключения в распределенной системе использовать централизованный алгоритм. Для организации синхронизации использовать мьютексы Delphi.
5	Управление передвижением автомобилей по мосту. Ограничение: грузоподъемность моста не более 100 тонн; вес автомобилей варьируется в пределах от 10 до 100 тонн. Для организации синхронизации использовать семафоры. Для организации взаимного исключения в распределенной системе использовать централизованный алгоритм.
6	Управление передвижением автомобилей по мосту. Ограничение: грузоподъемность моста не более 100 тонн; вес автомобилей варьируется в пределах от 10 до 100 тонн. Для организации взаимного исключения в распределенной системе использовать алгоритм Token Ring. Для организации синхронизации использовать семафоры.
7	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации взаимного исключения в распределенной системе использовать алгоритм Token Ring.

5 ЛАБОРАТОРНАЯ РАБОТА №6 «СЕТИ И СЕТЕВЫЕ СТРУКТУРЫ»

Цель работы: изучение основных методов организации передачи данных по сети и основных сетевых структур, используемых в современных операционных системах, разработка программной модели, использующей передачу данных по сети на примере средств операционной системы семейства Windows, в соответствии с вариантом задания.

5.1 Теоретические сведения

Топологии сетей

Основные топологии сетей (рисунок 6.1): полностью соединенная сеть (любая машина соединена с любой другой), частично соединенная сеть, сеть древовидной структуры, сеть типа звезда, сеть типа кольцо.

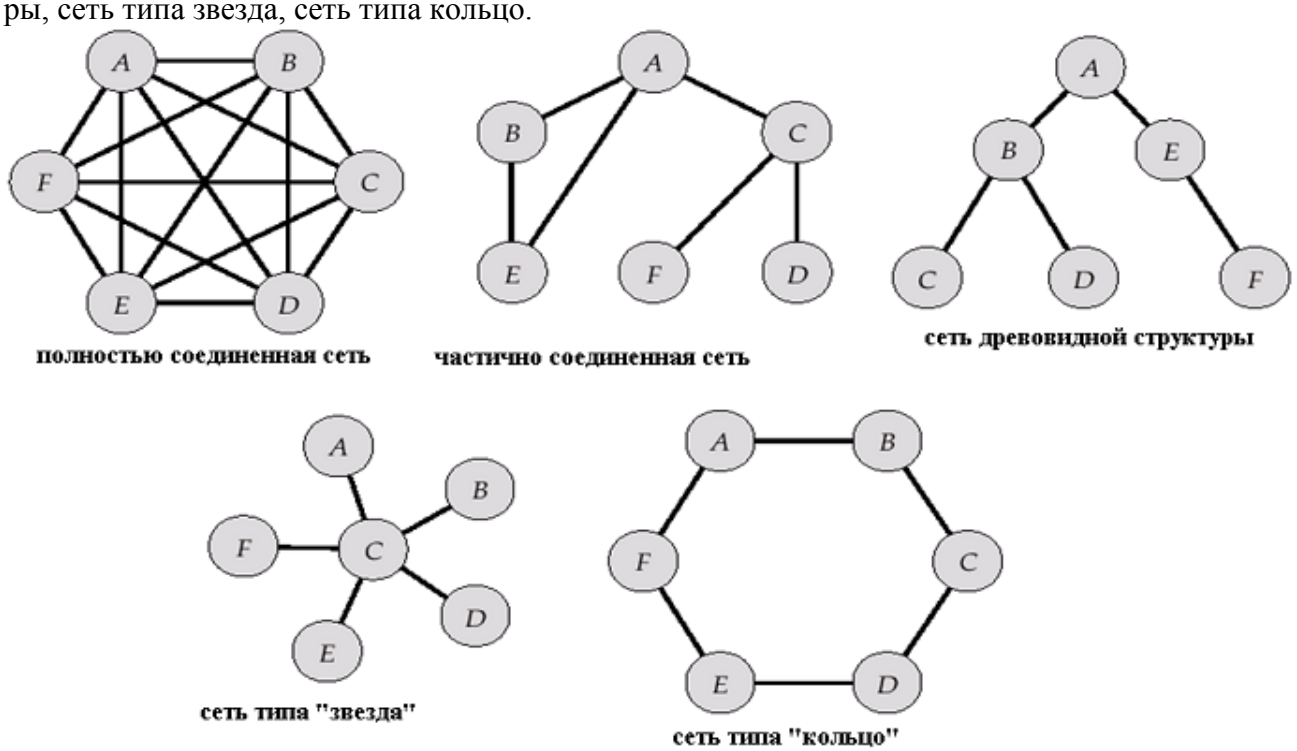


Рисунок 6.1 – Основные топологии сетей

На практике, практически любая проводная локальная сеть организована логически по принципу полностью соединенной сети, но физически сеть реализована следующим образом: каждая машина подсоединена к концентратору (hub) – устройству для установки коммуникаций между машинами в сети, а непосредственные соединения каждой машины с любой другой отсутствуют. В беспроводных сетях, аналогично, используются особые сетевые концентраторы для коммуникации машин друг с другом, так что можно также считать, что беспроводная локальная сеть – это полностью соединенная сеть.

В клиент-серверных региональных и глобальных сетях, разумеется, схема иная – компьютеры-клиенты соединяются только со своим сервером.

Типы сетей

По числу машин, размеру и протяженности сети подразделяются на локальные и глобальные.

Локальные сети.

Локальная сеть - Local-Area Network (LAN) – это сеть, расположенная на небольшой площади, например, в пределах здания или нескольких соседних зданий – офисов, либо даже в одной комнате (например, домашняя локальная сеть).

Локальная сеть организуется на основе многопользовательской шины, топологии кольца или звезды.

Скорость работы в локальной сети – порядка 10 мегабит в секунду при использовании кабеля типа витая пара (twisted pair) и сетевых концентраторов (hubs). Эта скорость невелика, особенно если машины расположены на расстоянии несколько сот метров друг от друга; при работе чувствуется замедление. Если используются волоконно-оптические кабели (fiber optic cables) и оптические переключатели (optical switches), то скорость работы сети возрастает до 100 мегабит в секунду, но такой способ соединения в сеть гораздо более дорогой. Современные сетевые адаптеры персональных компьютеров позволяют осуществлять соединение в сеть на скорости до 1 гигабита в секунду.

Узлами локальной сети являются, как правило, рабочие станции и (или) персональные компьютеры. В локальной сети могут быть также несколько (обычно одна или две) mainframe-машин или даже суперкомпьютеров или компьютерных кластеров (последнее характерно для университетов и крупных исследовательских центров)

В локальной сети доступны такие разделяемые сетевые ресурсы, как сетевые принтеры и другие устройства (например, сетевые диски – mass storage devices). Такие устройства могут иметь сетевые карты (адаптеры), свои сетевые адреса и свои имена в сети, так что они являются полноправными элементами сети, наряду с компьютерами.

Таким образом, тот минимум, который необходимо иметь для организации локальной сети, – это два или более компьютеров, сетевой концентратор (hub) и сетевые кабели типа витая пара с разъемами типа RJ45 для подсоединения к концентратору каждого компьютера локальной сети. В стандартной TCP/IP-сети каждый компьютер должен иметь свой IP-адрес и свое сетевое имя.

Глобальные и региональные сети

Глобальные сети - Wide-Area Network (WAN) – связывают географически удаленные машины. Используются соединения типа "точка-точка" (point-to-point) по линиям большой протяженности – выделенным линиям (dedicated lines). Более современные способы организации глобальных сетей – волоконно-оптические кабели и беспроводная связь типа Wi-Fi.

Взаимодействие в глобальной сети обычно требует нескольких сообщений.

Узлами глобальной сети могут быть как персональные компьютеры, так и mainframe-компьютеры, суперкомпьютеры, кластеры компьютеров.

Одним из современных видов глобальных и региональных сетей является WWAN (Wireless Wide Area Network) – беспроводная региональная сеть. Подобные сети реализуются фирмой Cingular (США). Распространены в США и Канаде. Основаны на протоколах мобильной связи GSM и CDMA. Скорость такой сети - порядка 100 МБит/с. В России аналогом являются беспроводные сети Wi-MAX, которые в настоящее время находят все более широкое распространение.

Проблема организации коммуникаций по сети.

Именование и разрешение имен в сети.

Системы (машины) в сети имеют имена. Сообщения идентифицируются номерами процессов (process ids). Процесс на удаленной системе идентифицируется парой <host-name, process-identifier>.

Для глобального разрешения сетевых имен используется Domain Name Service (DNS), обеспечивающий структуру именования машин, а также преобразование имени в

IP-адрес (в сети Интернет). Пример доменного имени (domain name) машины в сети Интернет: sibsiu.ru – доменное имя Сибирского государственного индустриального университета. С помощью протокола и сервисов DNS доменные имена преобразуются в числовые IP-адреса конкретных машин в сети, например, 62.231.181.225.

Стратегии маршрутизации

Маршрутизация (routing) – поиск маршрута для каждого сетевого пакета и направление его по найденному маршруту. Для маршрутизации в сетях используется специальное сетевое оборудование – маршрутизаторы (routers), которые обязательно должны использоваться в больших локальных сетях. Рассмотрим возможные стратегии маршрутизации:

- Фиксированная маршрутизация. Путь от A к B задан заранее; он изменяется, только если им невозможно воспользоваться из-за отказов аппаратуры. При этом:
  - Поскольку выбирается кратчайший путь, затраты на коммуникацию минимизированы.
  - Фиксированная маршрутизация не может быть адаптирована к изменению загрузки.
  - Обеспечивается получение сообщений в том же порядке, в каком они были посланы.
- Виртуальная цепочка. Путь от A к B фиксируется на время одного сеанса. Различные сеансы, включающие сообщения от A в B, могут иметь различную маршрутизацию. Особенности данного способа маршрутизации:
  - Это частичное средство адаптации к изменениям загрузки.
  - Обеспечивается получение сообщений в том же порядке, в каком они были посланы.
- Динамическая маршрутизация. Путь для отправки сообщения от A к B определяется только в момент отправки данного сообщения.
  - Обычно система посылает сообщение другой системе через соединение, наименее используемое в данный момент времени.
  - Метод адаптирован к изменениям загрузки.
  - Сообщения при данном методе могут приходить в другом порядке. Эта проблема может быть решена путем присваивания номера последовательности каждому сообщению (что и реализовано в сетях TCP/IP).

Стратегии соединения и разрешение коллизий

Для осуществления сетевой коммуникации необходимо установить сетевое соединение (connection). Различаются следующие виды сетевых соединений:

- Переключение схем. Устанавливается постоянное физическое соединение на все время коммуникации (например, по телефонной линии или по кабелю "витая пара").
- Переключение сообщений. Устанавливается временное соединение на период передачи сообщения (например, пересылка электронной почты).
- Переключение пакетов. Сообщения переменной длины делятся на пакеты фиксированной длины, которые и посылаются адресату. Пакеты могут передаваться по сети различными путями. Пакеты должны быть вновь собраны в сообщения по их прибытии. Пример – основной протокол Интернета TCP/IP.

Переключение схем требует времени для установки, но меньших накладных расходов на посылку каждого сообщения; при этом могут иметь место потери пропускной способности сети. Переключение сообщений и пакетов требует меньшего времени на установку, но накладные расходы на передачу сообщений больше, чем в первых двух методах.

Коллизии в сети возникают, если несколько систем одновременно обращаются к одному и тому же участку сети для передачи информации.

Для разрешения коллизий в сетях используется метод CSMA/CD (Carrier sense with multiple access; collision detection: носитель, чувствительный к одновременному доступу; обнаружение коллизий). Суть метода: Система определяет, не передается ли одновременно по данному участку сети сообщение другой системой. Если две или более систем начинают передачу сообщений в точности в одно и то же время, то фиксируется коллизия, и передача прекращается. Недостаток: если система сильно загружена, то может возникнуть множество коллизий, что приведет к падению производительности.

Метод CSMA/CD успешно используется в сетях типа Ethernet.

Известны следующие методы организации сетевого соединения и коммуникации в сетях с кольцевыми топологиями.

Передача маркера (token). Специальные сообщения, называемые маркерами, постоянно циркулируют в системе (обычно – при кольцевой топологии сети). Машина, которой требуется передать информацию, должна дождаться получения сообщения-маркера. Когда машина завершает свой раунд передачи сообщения, она передает по сети маркерное сообщение. Схема передачи маркерных сообщений используется в системах IBM и Apollo. Такая архитектура сети называется маркерным кольцом (token ring).

Слоты для сообщений. Несколько слотов для сообщений фиксированного размера постоянно циркулируют в системе (обычно – кольцевой структуры). Поскольку слот может вмещать только сообщения фиксированного размера, единое с логической точки зрения сообщение может быть разбито на несколько пакетов меньшей длины, каждый из которых пересылается в отдельном слоте. Такая схема была опробована в экспериментальной архитектуре сети Cambridge Digital Communication Ring (Кембриджское кольцо).

Классические и современные сетевые коммуникационные протоколы

Протоколы коммуникации

Модель OSI.

Сетевой протокол – это набор команд (операций) для взаимодействия узлов в сети. Ввиду сложности организации сетей, используемая модель сетевых протоколов содержит большое число уровней абстракции.

Согласно стандарту коммуникационной модели OSI, коммуникационная сеть подразделяется на следующие основные уровни (layers):

1. Физический уровень (physical layer) – механические и электрические устройства для передачи сигналов. Самый нижний уровень сетевой коммуникации. Включает сетевое оборудование - сетевые кабели, разъемы, концентраторы и т.д.
2. Уровень (связывания) данных (Data link layer) – обрабатывает фреймы (frames), или части пакетов фиксированной длины, включая обнаружение ошибок и восстановление после ошибок на физическом уровне.
3. Сетевой уровень (network layer) – обеспечивает соединение и маршрутизацию пакетов в коммуникационной сети, включая обработку адресов исходящих пакетов, декодирование адресов входящих пакетов и поддержку информации для маршрутизации для соответствующего ответа для изменения уровней загрузки.
4. Транспортный уровень (transport layer) – отвечает за сетевой доступ нижнего уровня и за передачу сообщений между клиентами, включая разделение сообщений на пакеты, сопровождение порядка пакетов, поток управления и генерацию физических адресов.
5. Уровень сеанса (session layer) – реализует сеансы (sessions), или протоколы коммуникации между процессами.
6. Уровень презентаций (presentation layer) – инкапсулирует различие в форматах между различными системами в сети, включая преобразования символов и полудуплексную (дуплексную) связь (эхо-вывод).
7. Уровень приложений (application layer) – самый высокий уровень модели сетевых протоколов. Взаимодействует непосредственно с запросами на передачу файлов поль-

зовательского уровня, протоколами удаленных входов и передачи электронной почты, а также со схемами распределенных баз данных.

Сообщение, передаваемое по сети, имеет заголовки каждого уровня, начиная от уровня связывания данных, затем следует тело сообщения и признак конца сообщения (уровня связывания данных, отвечающего за целостность передачи сообщения). Таким образом, реализация каждого уровня абстракции обрабатывает сообщение, используя заголовок сообщения соответствующего уровня.

Сети Ethernet

Ethernet (стандарт IEEE 802.3) – наиболее распространенный метод организации сетей. Относится к физическому (physical Ethernet) уровню и уровню связывания данных, согласно 8-уровневой модели ISO.

Основоположник сетей Ethernet – Роберт Меткалф (R. Metcalfe 1973). Он же впоследствии основал фирму 3COM, одну из наиболее известных компаний в области сетевых технологий.

Основные идеи Ethernet - использование коаксиального кабеля (BNC) и 48-битового адреса (MAC-адреса), который присваивается каждому сетевому компьютеру и используется для идентификации источников и получателей пакетов в сетях

Ethernet-адрес каждого компьютера фиксирован, выдается при загрузке операционной системы и может быть также выведен на консоль специальными командами ОС.

Протокол TCP/IP

TCP/IP (Transmission Control Protocol / Internet Protocol) – наиболее распространенное семейство протоколов сетевого и транспортного уровня, используемых в Интернете. Основоположники TCP/IP – Роберт Кан (Robert Kahn) и Винтон Серф (Vinton Cerf) -1972 – 1974.

Протокол TCP/IP основан на использовании IP-адресов каждого хоста (компьютера), имеющих вид: a.b.c.d (все четыре числа – в диапазоне от 0 до 255) и обеспечивает пересылку по сети пакетов (packets) фиксированного размера, содержащих адрес получателя и номер пакета в сообщении. TCP/IP -протокол обеспечивает транспорт сетевых пакетов, деление сообщения на пакеты отправителем и сборку сообщения из пакетов получателем. IP-адрес может быть закреплен за компьютером постоянно Интернет-провайдером пользователя компьютера, либо присваивается компьютеру динамически (каждый раз – разный) при выходе в Интернет.

Более общее современное название TCP/IP - Internet Protocol Suite. Различаются более новая версия – IPv6 и более старая – IPv4.

Протокол TCP/IP – синхронный, т.е. получатель ожидает получения каждого пакета и посылает отправителю подтверждение об этом. Другой вариант Интернет-протокола - UDP/IP (Universal Datagram Protocol / Internet Protocol) – асинхронный транспортный протокол, обеспечивающий обмен датаграммами – байтовыми массивами переменной длины; он считается менее надежным, чем TCP/IP, но работает быстрее, поэтому часто для быстрого обмена сообщениями в сетях предпочитают именно его.

Скорость TCP/IP не всегда удовлетворительна, ввиду огромного числа IP-узлов в Интернете. Для оптимизации связи между узлами сети применяются Distributed Hash Tables (DHT) – распределенные хеш-таблицы. В них собственная система имен узлов сети и более быстрого их поиска, чем с использованием TCP/IP протоколов, работающая "поверх" TCP/IP.

На рисунке 6.2 изображены уровни протокола TCP/IP и перечислены основные протоколы прикладного уровня, работающие поверх TCP/IP и UDP/IP, - протокол передачи гипертекста (основа функционирования World Wide Web), протоколы передачи файлов, передачи электронной почты, взаимодействия с удаленным терминалом, управления сетью.

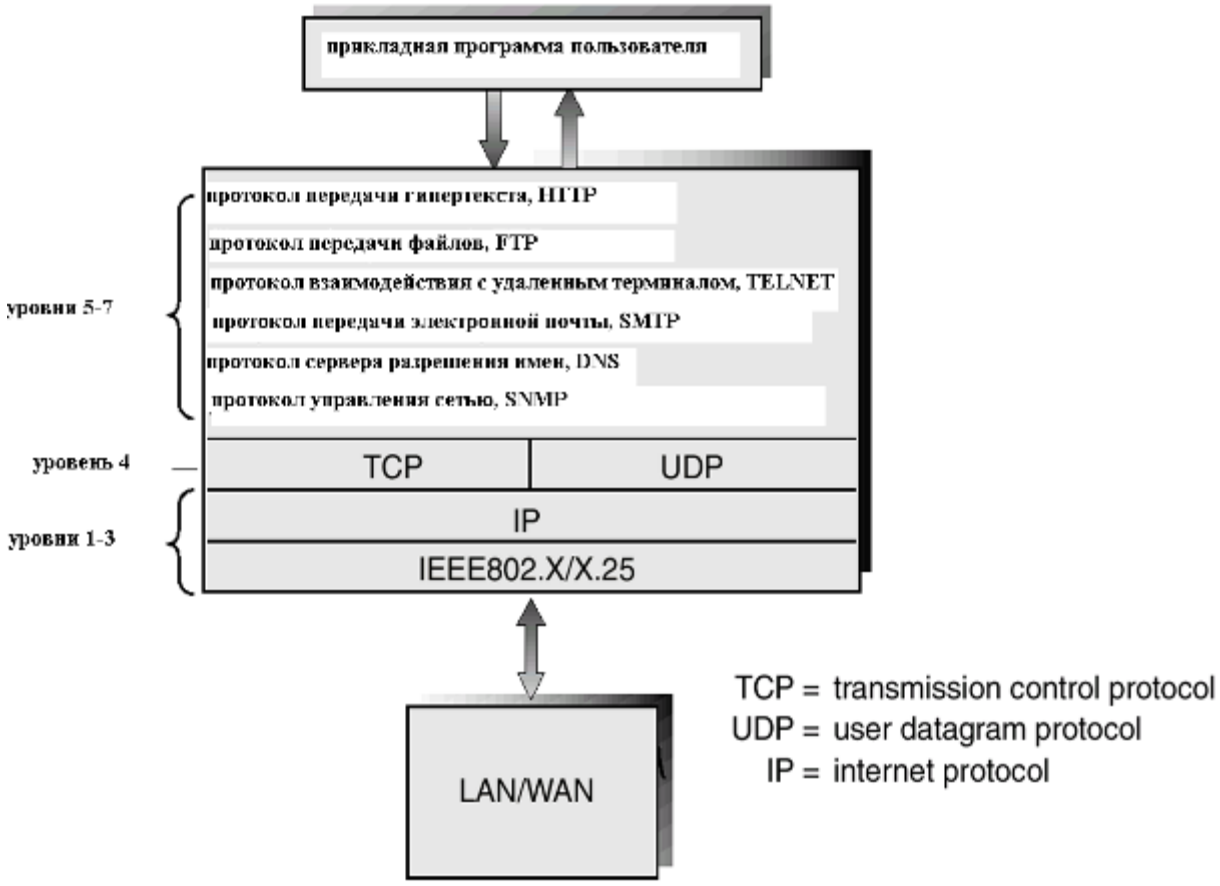


Рисунок 6.2 - Уровни протокола TCP/IP.

Современные сетевые протоколы

Протокол GPRS.

GPRS (General Packet Radio Service) - протокол беспроводной радиосвязи уровня связывания данных (уровня 2) по модели OSI, широко используемый в мобильной связи (GSM). Данный протокол "понимает" структуру IP-пакетов. Обеспечивается реальная скорость связи до 60 КБит/с, сравнимая со скоростью обычного модема и обмена через телефонную линию (dial-up).

Используется для реализации SMS (текстовых сообщений), MMS (мультимедийных сообщений), Instant messaging and presence (отправки и получения мгновенных сообщений), WAP (упрощенного доступа к Web для мобильных телефонов), мобильного Интернета.

При использовании TCP/IP GPRS-протокол присваивает каждому мобильному телефону один или несколько IP-адресов и обеспечивает надежную пересылку IP-пакетов. IP-адреса, как правило, присваиваются динамически.

Для маршрутизации пакетов используются точки доступа (access points) со своими именами Access Point Names (APNs). При настройке GPRS в мобильном телефоне необходимо указать APN, предоставляемое провайдером.

Семейство протоколов Wi-Fi (IEEE 802.11x)

Wi-Fi (IEEE 802.11x) – это семейство протоколов уровня связывания данных (уровня 2 по модели OSI) для беспроводной радиосвязи в локальных сетях (WLAN). Другое неофициальное (более старое) название того же семейства протоколов – RadioEthernet.

Используется для выхода в Интернет, передачи голосовых сообщений через TCP/IP (VoIP), связи с мультимедийными устройствами (цифровыми камерами, проекторами и т.п.)

Wi-Fi - связь доступна в радиусе действия точки доступа (access point) ~ 200-250 метров. Зона доступа Wi-Fi носит название hotspot.

Wi-MAX – более высокоскоростной вариант Wi-Fi (со скоростью передачи данных до 1 Гбит / с) с большим радиусом действия.

Обмен мгновенными сообщениями (Instant Messaging and Presence)

Instant Messaging and Presence (IMP) - семейство протоколов и технологий верхнего уровня (application layer) для обмена сообщениями между клиентами, использующими мобильные телефоны, коммуникаторы, лаптопы и перемещающимися из одной точки в другую.

IMP использует адреса, сходные с email-адресами, например: node@domain.work – XMPP-адрес.

Посылаемые сообщения – как правило, текстовые, но также возможно посылать и графические образы.

Основные протоколы IMP: SIMPLE / SIP; XMPP (Jabber); Wireless Village.

5.2 Практическая реализация в ОС Windows

Передача данных с помощью сокетов

Для передачи данных по сети чаще всего используют механизм сокетов. С точки зрения своей сущности сокет – это модель одного конца сетевого соединения, со своими свойствами и возможностью читать и записывать данные. С точки зрения содержания сокет – это прикладной программный интерфейс, входящий в состав разных операционных систем, в том числе Windows – начиная с версии 3.11. Таким образом, сокеты (sockets) представляют собой высокоуровневый унифицированный интерфейс взаимодействия с телекоммуникационными протоколами. В технической литературе встречаются различные переводы этого слова – их называют и гнездами, и соединителями, и патронами, и патрубками, и т. д.

Механизм соединения при помощи сокетов прост. На одной стороне создается клиентский сокет. Для инициализации связи ему нужно задать путь к серверному сокету, с которым предстоит установить соединение.

Путь в сети задается двумя параметрами: адресом или равноценным ему именем хоста, или просто хостом и номером порта. *Хост* — это система, в которой запущено приложение, содержащее сокет. У компьютера может быть несколько сетевых устройств и несколько адресов. Адрес в сетях TCP/IP задается четверкой чисел в диапазоне 0..255, например, так: 192.168.99.1. Имя хоста, как правило, — символьная строка, поставленная в соответствие адресу и записанная по правилам UNC, например http://www.microsoft.com. Взаимное соответствие между именами и адресами может осуществляться по-разному, в зависимости от масштаба сети и применяемой ОС. В Internet существует система имен доменов (DNS) — специальные серверы, хранящие и поддерживающие таблицы соответствия между символьным именем и адресом. Но в любом случае соединение по адресу быстрее, так как не нужно обращаться за дополнительной информацией.

Номер порта – это число, обычно зарезервированное для протоколов более высокого уровня. Если сокет сравнить с радиостанцией, то порт это частота, на которой она вступает в связь с другой радиостанцией.

С одной из двух вступающих в связь сторон запускается серверный сокет. Первоначально он находится в состоянии прослушивания (listening), то есть ожидания связи. После получения запроса от другой стороны – клиента – устанавливается связь. Но в то же время создается новый сокет для продолжения прослушивания.

В настоящее время в Delphi есть очень много компонентов, позволяющих создать приложение, передающее данные по сети с помощью механизмов сокетов. Здесь рассмотрим некоторые из них. Для более полного «погружения» в принципы функционирования сокетов начнем изучение механизма сокетов с использования API-функций Windows, ко-

торые собраны в библиотеке ws2_32.dll, находящейся в системной папке Windows. Данная библиотека предоставляет огромные возможности по использованию сокетов. Однако, рассмотрим только некоторые, чаще всего используемые при создании приложений.

Использование API-функций библиотеки WINSOCK

Обзор сокетов

Библиотека WINSOCK поддерживает два вида сокетов — *синхронные* (блокируемые) и *асинхронные* (неблокируемые). Синхронные сокеты задерживают управление на время выполнения операции, а асинхронные возвращают его немедленно, продолжая выполнение в фоновом режиме, и, закончив работу, уведомляют об этом вызывающий код.

Операционные системы семейства Windows 3.x поддерживает только асинхронные сокеты, поскольку в среде с корпоративной многозадачностью захват управления одной задачей "подвешивает" все остальные, включая и саму систему. Операционные системы семейства Windows 9x/NT поддерживают оба вида сокетов, однако, в силу того, что синхронные сокеты программируются более просто, чем асинхронные, последние не получили большого распространения.

Сокеты позволяют работать со множеством протоколов и являются удобным средством межпроцессорного взаимодействия, однако в настоящее время чаще всего на практике используются сокеты семейства протоколов TCP/IP, использующихся для обмена данными между узлами сети Интернет, поэтому сконцентрируем свое внимание именно на этом протоколе.

Независимо от вида, сокеты делятся на два типа — потоковые и дейтаграммные. Потоковые сокеты (протокол TCP) работают с установкой соединения, обеспечивая надежную идентификацию обеих сторон и гарантируют целостность и успешность доставки данных. Дейтаграммные сокеты (протокол UDP) работают без установки соединения и не обеспечивают ни идентификации отправителя, ни контроля успешности доставки данных, зато они заметно быстрее потоковых.

Выбор того или иного типа сокетов определяется транспортным протоколом, на котором работает сервер, — клиент не может по своему желанию установить с дейтаграммным сервером потоковое соединение.

Этапы создания программы, использующей синхронные сокеты

Как отмечалось ранее, для организации передачи данных по сети с помощью сокетов должно быть, по крайней мере, два приложения: сокетный сервер и клиентское приложение. В создании каждого из них есть много одинаковых этапов, поэтому рассмотрим эти этапы вместе для сервера и клиента в их эволюционном порядке.

Сервер и клиент. Шаг первый. Инициализация библиотеки

Для работы с библиотекой Winsock 2.x в исходный тест программы необходимо включить в секцию uses ссылку на модуль winsock2, который представляет собой заголовочные файлы библиотеки WinSock. Здесь необходимо отметить, что в поставке среды разработки программ Delphi нет соответствующего модуля, есть модуль winsock.pas, который содержит заголовочные файлы библиотеки Winsock первой версии. Таким образом, проще всего найти и скачать такие заголовочные файлы из Интернета. Для окончательного подключения заголовочных файлов необходимо также сделать еще одно действие из трех возможных:

- либо скопировать файлы winsock2.pas, ws2tcpip.inc, wsipx.inc, wsnwlink.inc и wsnetbs.inc в ту же директорию, в которой находится разрабатываемая программа;
- либо поместить эти файлы в папку Delphi\Lib;
- либо поместить указанные файлы в отдельную директорию, а затем подключить эту директорию к Delphi с помощью меню Tools пункт Environment options.

Далее перед началом использования функций библиотеки WINSOCK ее необходимо подготовит к работе вызовом функции

```
function WSAStartup(wVersionRequired: word; var WSData: TWSAData): Integer;
```

Первый параметр (wVersionRequired) – номер требуемой версии и подверсии WinSock; второй параметр (WSData) – тип-запись TWSAData, в которую при успешной инициализации будет занесена информация о производителе библиотеки. Поля этой структуры, которые могут быть использованы в программе, следующие:

wVersion: Word – номер версии

szDescription: Array[0..WSADESCRIPTION_LEN] of Char – описание библиотеки

szSystemStatus: Array[0..WSASYS_STATUS_LEN] of Char – статус библиотеки (запущена или нет).

Остальные поля не представляют какого-либо интереса для прикладного приложения.

Если инициализация библиотеки не осуществляется, функция возвращает ненулевое значение.

Для определения ошибки инициализации библиотеки можно использовать и специальную функцию WSAGetLastError, которая возвращает код возникшей ошибки при любом действии с сокетами (то есть ее можно использовать не только при инициализации библиотеки, но и при других действиях, связанных с сокетами). Коды ошибок можно обрабатывать и выдавать пользователю «осмысленный» ответ о причине возникновения ошибки. Все коды перечислены в файле WinSock2.pas. Например, ошибка 10093 (WSANOTINITIALISED) означает невозможность инициализации библиотеки. Например, предыдущий пример можно записать так:

```
var
  GInitData : TWSADATA;
begin
  WSAStartup($0202, GInitData);
  if WSAGetLastError = 0 then begin
    WriteLn('Library ', GInitData.szDescription, ' is ', GInitData.szSystemStatus);
  end
  else begin
    WriteLn('Error to initialized library: ', WSAGetLastError);
    ReadLn;
    exit;
  end;
```

Сервер и клиент. Шаг второй. Создание сокета

Необходимо создать объект "сокет". Это осуществляется функцией

```
function socket(const af, struct, protocol: Integer): TSocket;
```

У данной функции следующие параметры:

- Тип используемой адресации. Для интернет-приложений он должен иметь значение AF_INET. Также существуют и другие, они достаточно хорошо описаны в файле winsock2.pas.
- Тип создаваемого сокета – потоковый (SOCK_STREAM) или дейтаграммный (SOCK_DGRAM). В принципе, существует еще один тип – сырые сокеты, но они не поддерживаются средствами WinSock.
- Тип транспортного протокола. Нулевое значение (или IPPROTO_IP) соответствует выбору по умолчанию: TCP - для потоковых сокетов и UDP - для дейтаграммных. В большинстве случаев нет никакого смысла задавать протокол вручную и обычно полагаются на автоматический выбор по умолчанию. Однако можно задать и вручную, возможных значений очень много, например: IPPROTO_IP, IPPROTO_ICMP,

IPPROTO_IGMP, IPPROTO_GGP, IPPROTO_TCP, IPPROTO_PUP, IPPROTO_UDP, IPPROTO_IDP, IPPROTO_ND, IPPROTO_RAW, IPPROTO_MAX.

Если функция завершилась успешно, она возвращает дескриптор сокета, в противном случае - INVALID_SOCKET. Пример использования функции:

```
var
  GInitData : TWSADATA;
  ASocket: TSocket;
begin
  if WSAStartup($0202, GInitData) = 0 then begin
    WriteLn('Library ', GInitData.szDescription, ' is ',
            GInitData.szSystemStatus);
  end;
  ASocket := Socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
  if ASocket <> INVALID_SOCKET then begin
    WriteLn('Socket ', ASocket, ' is create');
  end;
```

Здесь и далее, если при инициализации библиотеки или создании сокета возникнет любая ошибка, то надо прервать дальнейшую работу программы, так как она [работа] абсолютно бессмысленна.

Дальнейшие шаги зависят от того, является приложение сервером или клиентом. Ниже эти два случая будут описаны раздельно.

Клиент. Шаг третий. Установка соединения

Для установки соединения с удаленным узлом потоковый сокет должен вызвать функцию

```
function connect(const s: TSocket; const name: PSockAddr;
namelen: Integer): Integer;
```

У данной функции следующие параметры:

- Дескриптор сокета, возращенный функцией socket.
- Указатель на структуру типа TSockAddr, содержащую в себе адрес и порт удаленного узла, с которым устанавливается соединение. Причем адрес необходимо перевести из имени или спецификации ip4 - XXX.XXX.XXX.XXX, использование их в «чистом» виде невозможно. Номер порта также нельзя использовать в традиционном виде, а необходимо использовать специальный сетевой порядок байтов.
- Размер структуры типа TSockAddr.

После вызова данной функции система предпринимает попытку установить соединение с указанным узлом. Если по каким-то причинам это сделать не удастся (адрес задан неправильно, узел не существует или "висит", компьютер находится не в сети), функция возвратит ненулевое значение.

По работе данной функции необходимо сделать ряд замечаний:

Во-первых, данная функция справедлива для работы потоковых сокетов, а дейтаграммные сокеты работают без установки соединения, поэтому обычно не обращаются к функции connect. Однако дейтаграммные сокеты могут использовать в своей работе вызов функции connect. При использовании данной функции дейтаграммные сокеты могут обмениваться данными с узлом не только функциями sendto, recvfrom, но и более удобными и компактными send и recv.

Во-вторых, в функции connect используется указатель на структуру типа TSockAddr, которая имеет достаточно сложную внутреннюю структуру, а при заполнении полей используется множество специфических полей. Кратко остановимся на структуре и функциях.

Работа с адресами

Для задания адреса и порта используется структура типа TSockAddr, которая определена следующим образом:

```
TSockAddrIn = packed record
  case Integer of
    0: (sin_family : Word;    // Семейство протоколов
                              // (как правило, AF_INET)
        sin_port   : Word;    // Порт
        sin_addr   : TInAddr; // IP-адрес
        sin_zero   : array[0..7] of Char);  // Хвост из
                   // восьми нулевых байт, который остался от
                   // четырнадцатисимвольного массива sa_data
    1: (sa_family  : Word; // Семейство протоколов
                           // (как правило, AF_INET)
        sa_data    : array[0..13] of Char)  // IP-адрес узла
                                            // и порт
end;
TSockAddr = TSockAddrIn;
PSockAddr = ^TSockAddr;
```

Используемая в качестве поля структура TInAddr описана следующим образом:

```
TInAddr = packed record
  case integer of
    0: (S_un_b: SunB);
    1: (S_un_w: SunW);
    2: (S_addr: DWORD);
end;
SunB = packed record
  s_b1, s_b2, s_b3, s_b4: Byte;
end;

SunW = packed record
  s_w1, s_w2: Word;
end;
```

Как видно, она состоит из одного IP-адреса, записанного в трех "ипостасях" — четырехбайтовой последовательности (S_un_b), пары двухбайтовых слов (S_un_w) и одного длинного целого (S_addr) — использовать можно любой из них.

Как отмечалось выше, нельзя использовать адрес в «пользовательском» виде. Для преобразования IP-адреса, записанного в виде символьной последовательности наподобие "127.0.0.1" в четырехбайтовую числовую последовательность, предназначена функция

```
function inet_addr(cp: PChar): DWORD;
```

Она принимает указатель на символьную строку и в случае успешной операции преобразует ее в четырехбайтовый IP-адрес или -1, если это невозможно.

Однако попытка передать функции inet_addr доменное имя узла приводит к ошибке. Узнать IP-адрес такого домена можно с помощью функции

```
function gethostbyname(name: PChar): PHostEnt;
```

функция обращается к DNS и возвращает свой ответ в структуре THostEnt или нуль, если DNS-сервер не смог определить IP-адрес данного домена.

Структура THostEnt выглядит следующим образом:

```
PHostEnt = ^THostEnt;
THostEnt = packed record
  h_name: PChar;                  // Официальное имя узла
  h_aliases: ^PChar;              // Альтернативные имена узла
                                  // (массив строк)
  h_addrtype: Smallint;           // Тип адреса
                                  // (как правило, AF_INET)
```

```
  h_length: Smallint;              // Длина адреса
  case Byte of
    0: (h_addr_list: ^PChar);      // Список указателей на
                                   // IP-адреса
    1: (h_addr: ^PChar);           // Список четырехбайтовой
                                   // последовательности,
                                   // являющейся первым адресом в
                                   // предыдущем списке адресов
end;
```

Таким образом, программист должен предоставить возможность пользователю ввести адрес либо в спецификации IP4, либо в виде доменного имени, а программа должна самостоятельно определить вид адреса и преобразовать его в сетевой адрес с помощью той или иной функции. Это можно сделать следующим образом: передать введенную пользователем строку функции inet_addr, если она возвратит ошибку, то вызвать функцию gethostbyname.

Для решения обратной задачи – определения доменного имени по IP-адресу предусмотрена функция

```
function gethostbyaddr(addr: Pointer; len, struct: Integer): PHostEnt;
```

которая во всем аналогична gethostbyname, за тем исключением, что ее аргументом является не указатель на строку, содержащую имя, а указатель на четырехбайтовый IP-адрес, рассмотренный нами ранее. Еще два аргумента задают его длину и тип (соответственно, 4 и AF_INET).

Для обратного по отношению к функции inet_addr преобразования используется функция

```
function inet_ntoa(inaddr: TInAddr): PChar;
```

которая в качестве аргумента принимает IP-адрес, записанный в сетевом формате, а возвращает IP-адрес в виде строки символов вида XXX.XXX.XXX.XXX.

Сетевой порядок байтов

Среди производителей процессоров нет единого мнения на счет порядка следования младших и старших байтов. Так, например, у микропроцессоров Intel младшие байты располагаются по меньшим адресам, а у микропроцессоров Motorola 68000 — наоборот. Естественно, это вызывает проблемы при межсетевом взаимодействии, поэтому, был введен специальный сетевой порядок байтов, предписывающий старший байт передавать первым (не так, как у Intel).

Для преобразований чисел из сетевого формата в формат локального хоста и наоборот предусмотрено четыре функции - первые две манипулируют короткими целыми (16-битовыми словами), а две последние - длинными (32-битовыми двойными словами):

```
function ntohs(netshort: Word): Word;
function htons(hostshort: Word): Word;
function ntohl(netlong: DWORD): DWORD;
function htonl(hostlong: DWORD): DWORD;
```

Все значения, возвращенные socket-функциями, уже находятся в сетевом формате и "вручную" их преобразовывать нельзя. Так как это преобразование исказит результат и приведет к неработоспособности.

Чаще всего к вызовам четырех рассматриваемых функций прибегают для преобразования номера порта согласно сетевому порядку.

Пример определения адреса с помощью функции gethostbyname и установки соединения с сокетом следующий:

```
var
  ASocket: TSocket;
```

```
  addr: TSockAddr;
begin
  addr.sin_family := AF_INET;
  addr.sin_addr := TInAddr(inet_addr(PChar('192.168.0.1')));
  addr.sin_port := htons(1025);
  connect(ASocket, @addr, sizeof(addr));
end;
```

Пример определения адреса с помощью функции inet_addr и установки соединения с сокетом следующий:

```
var
  ASocket: TSocket;
  addr: TSockAddr;
  HostEnt: PHostEnt;
begin
  addr.sin_family := AF_INET;
  HostEnt := gethostbyname(PChar('www.sibsiu.ru'));
  addr.sin_addr.S_un_b.s_b1 := Byte(HostEnt^.h_addr^[0]);
  addr.sin_addr.S_un_b.s_b2 := Byte(HostEnt^.h_addr^[1]);
  addr.sin_addr.S_un_b.s_b3 := Byte(HostEnt^.h_addr^[2]);
  addr.sin_addr.S_un_b.s_b4 := Byte(HostEnt^.h_addr^[3]);
  addr.sin_port := htons(1025);
  connect(ASocket, @addr, sizeof(addr));
end;
```

Также можно и даже нужно производить проверку результатов подключения, и при возникновении ошибки сообщать пользователю и корректно завершать программу.

Сервер. Шаг третий. Связывание сервера с адресом

Прежде чем сервер сможет использовать сокет, он должен связать его с локальным адресом. Локальный, как, впрочем, и любой другой адрес Интернета, состоит из IP-адреса узла и номера порта. Если сервер имеет несколько IP-адресов, то сокет может быть связан как со всеми ними сразу (для этого вместо IP-адреса следует указать константу INADDR_ANY, равную нулю), так и с каким-то конкретным одним. Связывание осуществляется вызовом функции

```
function bind(const s: TSocket; const addr: PSockAddr; const namelen: Integer): Integer;
```

Параметры функции следующие:

- Дескриптор сокета, возвращенный функцией socket.
- Указатель на структуру типа TSockAddr.
- Длина структуру типа TSockAddr.

Необходимо отметить, что физический сервер может иметь несколько IP-адресов, например, локальный адрес «127.0.0.1» и адрес в локальной сети «192.168.0.255», поэтому лучше при объявлении адреса указать

```
addr.sin_addr.S_addr := 0;
```

или

```
addr.sin_addr.S_addr := INADDR_ANY;
```

и далее указать порт:

```
addr.sin_port := htons(1025);
```

больше ничего не надо указывать в структуре типа TSockAddr.

Строго говоря, клиент также должен связывать сокет с локальным адресом перед его использованием, однако, за него это делает функция connect, ассоциируя сокет с одним из портов, наугад выбранных из диапазона 1024-5000. Сервер же должен "садиться" на заранее определенный порт, например, 21 для FTP, 23 для telnet, 25 для SMTP, 80 для Web и

т.д. Поэтому ему приходится осуществлять связывание "вручную". При успешном выполнении функция возвращает нулевое значение и ненулевое - в противном случае.

Сервер. Шаг четвертый. Прослушивание

Выполнив связывание, потоковый сервер переходит в режим ожидания подключений, вызывая функцию

```
function listen(s: TSocket; backlog: Integer): Integer;
```

Параметры функции следующие:

- Дескриптор сокета.
- Максимально допустимый размер очереди сообщений. Размер очереди ограничивает количество одновременно обрабатываемых соединений, поэтому к его выбору следует подходить внимательно. Если очередь полностью заполнена, очередной клиент при попытке установить соединение получит отказ (TCP-пакет с установленным флагом RST). В то же время максимально разумное количество подключений определяется производительностью сервера, объемом оперативной памяти и т. д. Если этот параметр равен SOMAXCONN, то система сама установит максимально возможное для него значение.

Дейтаграммные серверы не вызывают функцию listen, т. к. работают без установки соединения и сразу же после выполнения связывания могут вызывать recvfrom для чтения входящих сообщений, минуя четвертый и пятый шаги.

Сервер. Шаг четвертый. Соединение

Извлечение запросов на соединение из очереди осуществляется функцией

```
function accept(const s: TSocket; var addr: TSockAddr; var addrlen: Integer): TSocket;
```

Параметры функции следующие:

- Дескриптор серверного сокета.
- В параметр addr типа TSockAddr функция заносит сведения о подключившемся клиенте (IP-адрес и порт).
- Длина структуру типа TSockAddr.

Результатом работы функции является дескриптор сокета, автоматически созданного и связанного с IP-адресом, который занесен в параметр addr.

Если в момент вызова accept очередь пуста, функция не возвращает управление до тех пор, пока с сервером не будет установлено хотя бы одно соединение. В случае возникновения ошибки функция возвращает отрицательное значение.

Для параллельной работы с несколькими клиентами следует сразу же после извлечения запроса из очереди порождать новый поток (процесс), передавая ему дескриптор созданного функцией accept сокета, затем вновь извлекать из очереди очередной запрос и т. д. В противном случае, пока не завершит работу один клиент, север не сможет обслуживать всех остальных.

```
var
  ASocket: TSocket;
  Client_Socket: TSocket;
  Client_addr: TSockAddr;
  Client_addr_size: Integer;
  HostEntClient: PHostEnt;
  CountClients: Integer = 0;
begin
  While true do begin
    Client_addr_size := sizeof(Client_Addr);
    Client_Socket := accept(ServerSocket, Client_Addr,
                            Client_addr_size);
    HostEntClient := gethostbyaddr(
```

```
                    @Client_addr.sin_addr.S_addr, 4, AF_INET);
    inc(CountClients);
    WriteLn('Connect client "', HostEntClient.h_name,
            '" to Server. Count client: ', CountClients);
    TClientThread.Create(Client_Socket,
                         HostEntClient.h_name);
  end;
end;
```

В данном примере производится не только подключение клиентского сокета, но и определение имени подключившегося клиента. Также необходимо обратить внимание, что функция accept в качестве третьего параметра принимает именно переменную, которая будет содержать размер структуры TSockAddr, а не функцию sizeof(Client_Addr). Также в данном примере последней инструкцией создается поток, в котором производится дальнейшая работа с клиентским сокетом.

Сервер и клиент. Обмен данными

После того как соединение установлено, потоковые сокеты могут обмениваться с удаленным узлом данными, вызывая функции

```
function send(s: TSocket; var Buf; len, flags: Integer): Integer;
```

и

```
function recv(s: TSocket; var Buf; len, flags: Integer): Integer;
```

для посылки и приема данных соответственно.

Параметры у этих функций одинаковые:

1. Дескриптор сокета, с которым производится обмен.
2. Указатель на посылаемые/получаемые данные (можно использовать не указатель, а просто переменные).
3. Длина посылаемых/получаемых данных (Длина сообщения не должна превышать значения в SO_MAX_MSG_SIZE).
4. Параметр работы функций. Этот параметр может принимать следующие значения: MSG_PEEK, MSG_OOB и MSG_DONTROUTE:
   a. Параметр MSG_PEEK заставляет функцию recv просматривать данные вместо их чтения. Просмотр, в отличие от чтения, не уничтожает просматриваемые данные.
   b. Параметр MSG_OOB предназначен для передачи и приема срочных данных. Срочные данные не имеют преимущества перед другими при пересылке по сети, а всего лишь позволяют оторвать клиента от нормальной обработки потока обычных данных и сообщить ему "срочную" информацию. Если данные передавались функцией send с установленным параметром MSG_OOB, для их чтения параметр MSG_OOB функции recv также должен быть установлен. Настоятельно рекомендуется воздержаться от использования срочных данных в своих приложениях.
   c. Параметр и MSG_DONTROUTE предписывает передавать данные без маршрутизации

Функция send возвращает управление сразу же после ее выполнения, независимо от того, получила ли принимающая сторона наши данные или нет. При успешном завершении функция возвращает количество передаваемых, но не переданных, данных. Протокол TCP гарантирует успешную доставку данных получателю, но лишь при условии, что соединение не будет преждевременно разорвано. Если связь прервется до окончания пересылки, данные останутся не переданными, но вызывающий код не получит об этом никакого уведомления. А ошибка возвращается лишь в том случае, если соединение разорвано до вызова функции send.

Функция recv возвращает управление только после того, как получит хотя бы один байт. Точнее говоря, она ожидает прихода целой дейтаграммы. Дейтаграмма — это сово-

купность одного или нескольких IP-пакетов, посланных вызовом send. Упрощенно говоря, каждый вызов recv за один раз получает столько байтов, сколько их было послано функцией send. При этом подразумевается, что функции recv предоставлен буфер достаточных размеров, - в противном случае ее придется вызвать несколько раз. Однако при всех последующих обращениях данные будет браться из локального буфера, а не приниматься из сети, т. к. TCP-провайдер не может получить "кусочек" дейтаграммы, а только ею всю целиком.

Дейтаграммный сокет также может пользоваться функциями send и recv если предварительно вызовет connect, но у него есть и свои функции:

```
function sendto(s: TSocket; var Buf; len, flags: Integer; var
addrto: TSockAddr; tolen: Integer): Integer;
```

и

```
function recvfrom(s: TSocket; var Buf; len, flags: Integer; var
from: TSockAddr; var fromlen: Integer): Integer;
```

Они очень похожи на функции send и recv – разница лишь в том, что данные функции требуют явного указания адреса узла, принимаемого или передаваемого данные. Вызов recvfrom не требует предварительного задания адреса передающего узла – функция принимает все пакеты, приходящие на заданный UDP-порт со всех IP-адресов и портов. Напротив, отвечать отправителю следует на тот же самый порт, откуда пришло сообщение. Поскольку функция recvfrom заносит IP-адрес и номер порта клиента после получения от него сообщения, программисту фактически ничего не нужно делать – только передать функции sendto тот же самый указатель на структуру TSockAddr, который был ранее передан функции recvfrem, получившей сообщение от клиента.

Во всем остальном обе пары функций полностью идентичны и работают с теми же самыми флагами.

Все четыре функции при возникновении ошибки возвращают значение SOCKET_ERROR.

Пример отправки данных клиентскому сокету:

```
send(ClientSocket, ResultCheck, sizeof(Integer), 0);
```

Пример получения данных серверным сокетом:

```
Byte_recv := recv(ClientSocket, Answer, sizeof(Integer), 0);
if Byte_recv = SOCKET_ERROR then begin
  dec(CountClients);
  WriteLn('Disconnect client "', ClientName, '" to Server.
          Count client: ', CountClients);
  closesocket(ClientSocket);
  exit;
end;
```

Здесь не используются флаги (передаётся 0), потому что нам абсолютно все равно как отправятся данные, а при получении данных необходимо их прочитать и удалить из стека.

Сервер и клиент. Шаг последний. Завершение работы

При завершении работы с сокетами необходимо совершить несколько действий по освобождению занимаемой сокетами памяти.

Для закрытия соединения и уничтожения сокета предназначена функция

```
function closesocket(const s: TSocket): Integer;
```

которая в случае удачного завершения операции возвращает нулевое значение.

Перед выходом из программы необходимо вызвать функцию

```
function WSACleanup: Integer;
```

для деинициализации библиотеки WINSOCK и освобождения используемых этим приложением ресурсов.

Необходимо помнить, что созданные в рамках использования механизма сокетов объекты не являются объектами с управляемым временем жизни, и, таким образом, автоматически они никогда не будут уничтожены, а ресурсы не будут освобождены.

Наряду с указанными способами существует еще один способ корректного завершения передачи данных. Протокол TCP позволяет выборочно закрывать соединение любой из сторон, оставляя другую сторону активной. Для этого необходимо вызвать функцию

```
function shutdown(s: TSocket; how: Integer): Integer;
```

передав в аргументе how одно из следующих значений: SD_RECEIVE для закрытия канала "сервер->клиент", SD_SEND для закрытия канала "клиент<-сервер", и, наконец, SD_BOTH для закрытия обоих каналов. При этом, например, если клиенту не нужно передавать данные серверу, то он может закрыть соединение "клиент->сервер", однако готов продолжать принимать от сервера данные до тех пор, пока он будет их посылать, т. е. хочет оставить соединение "клиент<-сервер" открытым.

Вызов функции shutdown не освобождает от необходимости закрытия сокета функцией closesocket.

Асинхронные сокеты

В рассмотренных этапах создания и работы приложения передача данных производится с помощью синхронных сокетов. Однако при их использовании некоторые функции (например, recv) приостанавливают работу приложения до возникновения некоего события. Конечно, работу каждого клиента и каждого сервера можно осуществлять в отдельном потоке, но лучше воспользоваться асинхронными сокетами, которые не прерывают работу приложения. В библиотеке WinSock асинхронные сокеты можно реализовать несколькими способами, например, с помощью функции select и с помощью функции WSAAsyncSelect.

Реализация асинхронных сокетов с помощью функции select

Функция

```
function select(nfds: Integer; readfds, writefds, exceptfds: PFDSet; timeout: PTimeVal): Integer;
```

позволяет производить контроль состояния сокета на данный момент, с использованием макросов FD_XXXXX. Работа данной функции проста. Она ожидает возникновение некоего события, как то: чтения (readfds), записи (writefds), исключения (exceptfds) – определенное время (параметр timeout), за которое это событие должно произойти, если оно не произошло, то больше его не ждут, а если произошло, то можно производить обработку этого события. Таким образом, задержка на событие связана только с накладными расходами на сервере, а не с ожиданием события. Иными словами, задержка выполнения программы будет не большей, чем указанное время ожидания, а может и меньшей, если сервер даст ответ раньше, чем окончится время ожидания. Время ожидания задается указателем на переменную, имеющую тип TTimeVal, структура которого следующая:

```
PTimeVal = ^TTimeVal;
TTimeVal = packed record
  tv_sec: Longint;           // время в секундах
  tv_usec: Longint;
end;
```

Также при работе рассматриваемой функции используется структура TFDSet, которая представляет собой массив сокетов:

```
const
  FD_SETSIZE     =   64;
type
  PFDSet = ^TFDSet;
  TFDSet = packed record
    fd_count: Word;
```

```
    fd_array: array[0..FD_SETSIZE-1] of TSocket;
  end;
```

Механизм использования функции select при реализации асинхронных сокетов следующий: создается массив сокетов типа TFDSet, в который добавляются с помощью процедуры

```
procedure FD_SET(Socket: TSocket; var FDSet: TFDSet);
```

те сокеты, на события которых необходимо реагировать; затем устанавливается время отслеживания возникновения события; вызывается функция select, в параметрах которой указывается за какими сокетами и какими событиями следить; позже проверяем массив сокетов типа TFDSet с помощью функции

```
function FD_ISSET(Socket: TSocket; var FDSet: TFDSet): Boolean;
```

если интересующий сокет есть в массиве, то событие произошло и далее его обрабатываем.

Рассмотрим на примере:

```
var
  ASocket: TSocket;
  addr: TSockAddr;
  fset: TFDset;
  tv: TTimeVal;

begin
...
  connect(ASocket, @addr, sizeof(addr));
  if WSAGetLastError = 0 then begin
  {соединение с сервером произошло}
    {устанавливаем время ожидания}
    tv.tv_sec := 2;
    tv.tv_usec := 0;
    {обнуляем массив сокетов, подготавливая к дальнейшей работе}
    FD_Zero(fset);
  end;
  {бесконечный цикл обмена данными с сервером}
  while true do begin
    {добавляем в массив тот сокет, за которым надо следить}
    FD_SET(ASocket, fset);
    {подготавливаем и отправляем данные на сервер}
    i := Random(100);
    send(ASocket, i, sizeof(Integer), 0);
    {производим наблюдение за очередью сообщений, пока сервер не
напишет в ответ, а клиент, соответственно, не прочитает}
    select(0, @fset, nil, nil, @tv);
    {производится ожидание возникновения события или пока не за-
кончится время ожидания}
    if FD_ISSET(ASocket, fset) then begin
    {если сокет остался в массиве, то есть событие состоялось}
      {убираем данный сокет из массива, что бы повторно не обра-
ботать}
      FD_CLR(ASocket, fset);
      {получаем данные от сервера}
      Byte_recv := recv(ASocket, i, sizeof(Integer), 0);
      if Byte_recv = SOCKET_ERROR then begin
```

```
        {если возникла ошибка при передаче или сервер отключил-
ся}
        closesocket(ASocket);
        WSACleanup;
        exit;
      end;
      {выводим полученное сообщение}
      ShowMessage(IntToStr(i));
    end;
  end;
end;
```

Реализация асинхронных сокетов с помощью функции WSAAsyncSelect

В предыдущем случае все равно приложение клиента будет «подвисать» пока производится обработка на стороне сервера. Чтобы исключить полностью эффект зависания можно воспользоваться обработкой событий библиотеки WinSock, и возможностью передачи этих событий окну Windows для дальнейшей обработки. Это делается с помощью функции

```
function WSAAsyncSelect(s: TSocket;: HWND; wMsg: u_int; lEvent:
Longint): Integer;
```

которая перенаправляет указанные события (wMsg, lEvent), возникшие в сокете (s), окну ОС Windows, имеющему дескриптор (HWindow). События, которые может отслеживать библиотека WinSock: FD_READ (чтение сокетом принимаемых данных); FD_WRITE (отправка сокетом данных); FD_OOB (принятие срочных данных); FD_ACCEPT (установка соединения сервером); FD_CONNECT (установка соединения); FD_CLOSE (закрытие сокета); FD_QOS; FD_GROUP_QOS; FD_ALL_EVENTS (все события). Причем можно комбинировать эти события в любом порядке суммируя их. Также программисту необходимо объявить пользовательское событие, и передать его в параметре wMsg, а также написать обработчик пользовательского события, например, с помощью Application.OnMessage.

Пример использования рассматриваемой функции следующий:

```
const
{объявляем пользовательское событие}
  WM_SOCKET = WM_USER + 100;

type
  TForm1 = class(TForm)
...
{объявляем обработчик пользовательского события, например, в
форме}
    procedure AppMessage(var Msg:TMsg; var Handled: Boolean);
...
  end;

var
  Form1: TForm1;
  ASocket: TSocket;
  addr: TSockAddr;

implementation

procedure TForm1.AppMessage(var Msg: TMsg; var Handled: Boole-
an);
var
```

```
  i, Byte_recv: Integer;
begin
  if Msg.message = WM_SOCKET then begin
  {если произошло пользовательское событие}
    if Msg.lParam=FD_READ then begin
    {если произошло чтение клиентом данных от сервера}
      Handled := True;
      Byte_recv := recv(ASocket, i, sizeof(Integer), 0);
      if Byte_recv = SOCKET_ERROR then begin
       {если возникла ошибка при передаче или сервер отключился}
        closesocket(ASocket);
        WSACleanup;
        exit;
      end;
      {как-то обрабатываем полученные данные, но лучше не так…}
      ShowMessage(IntToStr(i));
    end;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  {настраиваем соединение с сервером}
...
  connect(ASocket, @addr, sizeof(addr));
  if WSAGetLastError = 0 then begin
  {если соединение с сервером прошло успешно}
    {назначаем за каким событием следить}
    WSAAsyncSelect(ASocket, Application.Handle, WM_SOCKET,
                   FD_READ);
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
begin
  {отправка данных серверу}
  i := Random(100);
  send(ASocket, i, sizeof(Integer), 0);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := AppMessage;
end;
```

В примере вызовом функции WSAAsyncSelect производится указание библиотеки WinSock посылать пользовательское сообщение WM_SOCKET указанному приложению Application.Handle при возникновении сокетного события FD_READ чтения полученных данных.

Достоинством данного подхода является то, что программа абсолютно не имеет эффекта «зависания». Обработка полученных данных производится только тогда, когда эти

данные поступают в стек клиента. Однако есть и недостаток. Клиенту надо принять столько данных, сколько ему послали. Так как принимает клиент их частями и в функции реакции на прием данных в рассмотренном выше примере их нельзя как-то «докачать» (после срабатывания FD_READ, recv для «докачки» не может получить из стека остальные данные потому, что еще не была вызвана функция получения сообщения от операционной системы и доступ к очереди сокета закрыт). Значит необходимо хранить где-то все сокетные соединения и часть уже пришедших на них данных, а при срабатывании события FD_READ, накапливать эти данные. При достижении необходимого размера или комбинации символов, соответствующих сигналу конца пакета, соответственно реагировать.

Дополнительные возможности

Для "тонкой" настройки сокетов предусмотрена функция

```
function setsockopt(s: TSocket; level, optname: Integer; optval: PChar; optlen: Integer): Integer;
```

аргументы которой следующие:

1. Дескриптор сокета, который собираются настраивать.
2. Уровень настройки. С каждым уровнем связан свой набор опций. Всего определено два уровня – SOL_SOCKET и IPPROTO_TCP.
3. Название опции, определенной в уровне настройке. Таких опций достаточно много. Некоторые из них:
   a. Уровень SOL_SOCKET: SO_SNDBUF (размер входного буфера для передачи данных); SO_RCVBUF (размер входного буфера для приема данных); SO_ERROR (получение статуса ошибки и очистка); SO_TYPE (тип сокета) и др.
   b. Уровень IPPROTO_TCP: TCP_NODELAY (выключает Алгоритм Нагла) и др.
4. Указатель на переменную, содержащую значение опции.
5. Размер переменной, содержащей значение опции.

Для получения текущих значений опций сокета предусмотрена функция

```
function getsockopt(const s: TSocket; const level, optname: Integer; optval: PChar; var optlen: Integer): Integer;
```

которая полностью аналогична предыдущей функции setsockopt за исключением того, что не устанавливает опции, а возвращает их значения.

5.3 Задание на лабораторную работу

Реализовать программу, моделирующую сетевую распределенную управляющую систему в соответствии с вариантом задания. При моделировании принять:

– для организации взаимного исключения в распределенной системе использовать централизованный алгоритм;
– для обмена сетевыми сообщениями использовать протокол TCP;
– в качестве сервера реализовать управляющую систему, разрешающую или запрещающую проезд в туннеле (по мосту). В качестве клиентов выступают «бортовые компьютеры», установленные в моделируемые транспортные средства и разрешающие или запрещающие проезд в зависимости от ответа управляющего сервера.
  Предусмотреть возможность визуального контроля за работой системы.

Варианты заданий

Вариант	Управляющая система
1	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать аппарат событий. Для организации передачи данных использовать синхронные сокеты.
2	Управление передвижением поездов в туннеле. Ограничение: в каждый

	момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать аппарат критической секции. Для организации передачи данных использовать синхронные сокеты.
3	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать мьютексы WinAPI. Для организации передачи данных использовать синхронные сокеты.
4	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать мьютексы Delphi. Для организации передачи данных использовать синхронные сокеты.
5	Управление передвижением автомобилей по мосту. Ограничение: грузоподъемность моста не более 100 тонн; вес автомобилей варьируется в пределах от 10 до 100 тонн. Для организации синхронизации использовать семафоры. Для организации передачи данных использовать асинхронные сокеты.
6	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать аппарат событий. Для организации передачи данных использовать асинхронные сокеты.
7	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать аппарат критической секции. Для организации передачи данных использовать асинхронные сокеты.
8	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать мьютексы WinAPI. Для организации передачи данных использовать асинхронные сокеты.
9	Управление передвижением поездов в туннеле. Ограничение: в каждый момент временив туннеле может находиться только один поезд. Для организации синхронизации использовать мьютексы Delphi. Для организации передачи данных использовать асинхронные сокеты.
10	Управление передвижением автомобилей по мосту. Ограничение: грузоподъемность моста не более 100 тонн; вес автомобилей варьируется в пределах от 10 до 100 тонн. Для организации синхронизации использовать семафоры. Для организации передачи данных использовать асинхронные сокеты.

6 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Таненбаум Э. Современные операционные системы. 4-е изд. – СПб: Питер, 2010. – 1120 с.: ил.
2. Столлингс В. Операционные системы. 4-е изд. – М: Изд. дом «Вильямс», 2004. – 848 с.: ил.
3. Олифер Н. А., Олифер В. Г. Сетевые операционные системы. - СПб: Питер, 2009 г. – 672 с., ил.
4. Васильков А. Фатальные проблемы // Компьютерра. №5, 2009 г. – с. 37-39.
5. Понамарев В.А. COM и ActiveX в Delphi. – Спб.: БХВ-Петербург, 2001. – 320 с.:ил.
6. Дарахвелидзе П.Г., Марков Е.П. Delphi 4. – Спб.: БХВ-Петербург, 1999. – 816 с.:ил.
7. Тейксейра С., Пачеко К. Delphi 5. Руководство разработчика, том 1. Основные методы и технологии программирования: Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 832 с.: ил. – Парал. тит. англ.
8. Сафонов В.О. Основы современных операционных систем. - Интернет-университет информационных технологий - ИНТУИТ.ру, БИНОМ. Лаборатория знаний, 2011 г., 584 с.
9. Касперски К. ПК: решение проблем. – СПб.: БХВ-Петербург, 2003. – 560 с.: ил.
10. Фленов М.Е. Программирование в Delphi глазами хакера. – СПб.: БХВ-Петербург, 2003. – 368 с.: ил.
11. Danil. Сокеты: электронный ресурс – http://www.danil.dp.ua.

ПРИЛОЖЕНИЕ А

Программа Proccesses.dpr

```
program Proccesses;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas';

{$R *.res}

begin
  Application.Initialize;
  Application.Title := 'Proccesses';
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Модуль Unit1.pas

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, ExtCtrls, ComCtrls, Unit2, Buttons;

type
  TForm1 = class(TForm)
    Button1: TButton;
    PageControl1: TPageControl;
    OpenDialog1: TOpenDialog;
    Memo1: TMemo;
    Panel3: TPanel;
    Panel1: TPanel;
    Label1: TLabel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    Panel2: TPanel;
    Label2: TLabel;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    ListBox1: TListBox;
    Label3: TLabel;
    ListBox2: TListBox;
    Label4: TLabel;
    GroupBox1: TGroupBox;
    Label5: TLabel;
    Label6: TLabel;
    Timer1: TTimer;
    Label7: TLabel;
    Timer2: TTimer;
    procedure Button1Click(Sender: TObject);
    procedure FormActivate(Sender: TObject);
```

```
    procedure WriteToLog(Text: String);
    procedure SpeedButton1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure SpeedButton3Click(Sender: TObject);
    procedure SpeedButton4Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    function ContextProcessByID(ID: Integer): PContext;
    procedure Timer2Timer(Sender: TObject);
  private
  public
  end;

var
  Form1: TForm1;
  TabSheet : array [0..9] of TTabSheet;
  ListBox: array [0..9] of TListBox;
  ALabel: array [0..3, 0..9] of TLabel;
  Process: array [0..9] of TProcess;     //Массив процессов
  Processor: TProcessor;                 //Модель процессора
  IODevice: TProcessor;                  //Модель устройства вво-
да/вывода
  Readys: PDescriptorStack;              //Ссылка на очередь готовых
  Waits: PDescriptorStack;               //Ссылка на очередь ожидающих
  StateReadys, StateWaits: TStateStack;  //Состояние очередей

implementation

{$R *.dfm}

//Создание процесса:
//ID - идентификационный номер процесса
//Name - название процесса
//Memory - количество памяти/, необходимое процессу
function CreateProccess(ID: Integer; Name: String; Memory: Integer):
TProcess;
begin
  with Result.Descriptor do begin
    PID := ID;
    Kvant := 2;
    State := spNone;
    Prioritet := ppNormal;
  end;
  New(Result.Context);
  Result.Context^.PID := ID;
  Result.Context^.CommandLine := 0;
  Result.Context^.CountRun := 0;
  Result.Context^.Command := cMEMORY;
  Result.Context^.CurrentRun := 0;
  Result.Context^.Name := Name;
  Result.Context^.Memory := Memory;
end;

//Запись в лог
procedure TForm1.WriteToLog(Text: String);
begin
  Memo1.Lines.Add(Text);
end;
```

```
//Обработка кнопки запуска процесса
procedure TForm1.Button1Click(Sender: TObject);
begin
  if PageControl1.PageCount = 10 then begin
    ShowMessage('Больше процессов нельзя!');
    exit;
  end;
  if OpenDialog1.Execute then begin
    //Готовим внешний вид новой вкладки
    TabSheet[PageControl1.PageCount] := TTabSheet.Create(Self);
    TabSheet[PageControl1.PageCount].Caption := 'Процесс' + IntTo-
Str(PageControl1.PageCount + 1);
    TabSheet[PageControl1.PageCount].PageControl := PageControl1;
    ListBox[PageControl1.PageCount-1] := TListBox.Create(Self);
    ListBox[PageControl1.PageCount-1].Parent := Tab-
Sheet[PageControl1.PageCount-1];
    ALabel[0,PageControl1.PageCount-1] := TLabel.Create(Self);
    ALabel[0,PageControl1.PageCount-1].Caption := 'Квант = ';
    ALabel[0,PageControl1.PageCount-1].Top := 5;
    ALabel[0,PageControl1.PageCount-1].Left := 5;
    ALabel[0,PageControl1.PageCount-1].Parent := Tab-
Sheet[PageControl1.PageCount-1];
    ALabel[1,PageControl1.PageCount-1] := TLabel.Create(Self);
    ALabel[1,PageControl1.PageCount-1].Caption := 'Состояние = ';
    ALabel[1,PageControl1.PageCount-1].Top := 20;
    ALabel[1,PageControl1.PageCount-1].Left := 5;
    ALabel[1,PageControl1.PageCount-1].Parent := Tab-
Sheet[PageControl1.PageCount-1];
    ALabel[2,PageControl1.PageCount-1] := TLabel.Create(Self);
    ALabel[2,PageControl1.PageCount-1].Caption := 'Приоритет = ';
    ALabel[2,PageControl1.PageCount-1].Top := 35;
    ALabel[2,PageControl1.PageCount-1].Left := 5;
    ALabel[2,PageControl1.PageCount-1].Parent := Tab-
Sheet[PageControl1.PageCount-1];
    ALabel[3,PageControl1.PageCount-1] := TLabel.Create(Self);
    ALabel[3,PageControl1.PageCount-1].Caption := 'Текущая команда =
';
    ALabel[3,PageControl1.PageCount-1].Top := 50;
    ALabel[3,PageControl1.PageCount-1].Left := 5;
    ALabel[3,PageControl1.PageCount-1].Parent := Tab-
Sheet[PageControl1.PageCount-1];
    PageControl1.ActivePageIndex := PageControl1.PageCount-1;
    PageControl1.ActivePage.Align := alClient;
    ListBox[PageControl1.PageCount-1].Align  := alBottom;
    ListBox[PageControl1.PageCount-1].Height :=
trunc(PageControl1.ActivePage.Height-70);
    ListBox[PageControl1.PageCount-
1].Items.LoadFromFile(OpenDialog1.FileName);
    ListBox[PageControl1.PageCount-1].ItemIndex := -1;
    //Создание процесса
    Process[PageControl1.PageCount-1] := Create-
Proccess(PageControl1.PageCount-1,
TabSheet[PageControl1.PageCount-1].Caption,
ParseInt(ListBox[PageControl1.PageCount-1].Items[0]));
    ALabel[0,PageControl1.PageCount-1].Caption := 'Квант = ' + IntTo-
Str(Process[PageControl1.PageCount-1].Descriptor.Kvant);
```

```
    ALabel[1,PageControl1.PageCount-1].Caption := 'Состояние = ' +
StateToStr(Process[PageControl1.PageCount-1].Descriptor.State);
    ALabel[2,PageControl1.PageCount-1].Caption := 'Приоритет = ' +
PriorToStr(Process[PageControl1.PageCount-1].Descriptor.Prioritet);;
    ALabel[3,PageControl1.PageCount-1].Caption := 'Текущая команда = '
+ CommandToStr(Process[PageControl1.PageCount-1].Context^.Command);
    WriteToLog('Запущен новый процесс с именем ' + Pro-
cess[PageControl1.PageCount-1].Context^.Name +
               ' (Квант = ' + IntToStr(Process[PageControl1.PageCount-
1].Descriptor.Kvant) +
               '; Состояние = ' + State-
ToStr(Process[PageControl1.PageCount-1].Descriptor.State) +
               '; Приоритет = ' + PriorTo-
Str(Process[PageControl1.PageCount-1].Descriptor.Prioritet) + ')');
    Panel3.Enabled := True;
    //Помещаем процесс в стек готовых
    In_Stack(Readys,
@Process[PageControl1.ActivePageIndex].Descriptor);
    StateReadys := ssFully;

ListBox1.Items.Add(Process[PageControl1.ActivePageIndex].Context^.Name
);
    WriteToLog('Процесс ' + Pro-
cess[PageControl1.ActivePageIndex].Context^.Name + ' помещен в очередь
готовых');
    //Включаем моделирование процессора и устройств ввода/вывода
    if not Timer1.Enabled then Timer1.Enabled := True;
    if not Timer2.Enabled then Timer2.Enabled := True;
  end;
end;

//Действия при активации формы
procedure TForm1.FormActivate(Sender: TObject);
begin
  WriteToLog('Программа моделирования алгоритмов планирования процес-
сов запущена');
  GetMem(Processor.Run, SizeOf(TDescriptor));
  Processor.State := spEmpty;
  Label5.Caption := 'Состояние = ' + StateProcessorTo-
Str(Processor.State);
  Label6.Caption := 'Процесс = ';
  WriteToLog('Модель процессора подготовлена');
  GetMem(Readys, SizeOf(TDescriptorStack));
  StateReadys := ssEmpty;
  WriteToLog('Очередь готовых подготовлена');
  GetMem(Waits, SizeOf(TDescriptorStack));
  StateWaits := ssEmpty;
  WriteToLog('Очередь ожидающих подготовлена');
  GetMem(IODevice.Run, SizeOf(TDescriptor));
  IODevice.State := spEmpty;
  WriteToLog('Модель устройства ввода/вывода подготовлена');
end;

//Уменьшение кванта процессорного времени, если это возможно
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
```

```
  if CheckKvant-
Down(Process[PageControl1.ActivePageIndex].Descriptor.Kvant - 1) then
begin
    Process[PageControl1.ActivePageIndex].Descriptor.Kvant := Pro-
cess[PageControl1.ActivePageIndex].Descriptor.Kvant - 1;
    ALabel[0,PageControl1.ActivePageIndex].Caption := 'Квант = ' +
IntToStr(Process[PageControl1.ActivePageIndex].Descriptor.Kvant);
    WriteToLog('Процессу ' + Pro-
cess[PageControl1.ActivePageIndex].Context^.Name +
               ' уменьшен квант процессорного времени до ' + IntTo-
Str(Process[PageControl1.ActivePageIndex].Descriptor.Kvant));
  end
  else begin
    ShowMessage('Уменьшение кванта процессорного времени невозмож-
но!');
  end;
end;

//Увеличение кванта процессорного времени, если это возможно
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  if CheckKvan-
tUp(Process[PageControl1.ActivePageIndex].Descriptor.Kvant + 1) then
begin
    Process[PageControl1.ActivePageIndex].Descriptor.Kvant := Pro-
cess[PageControl1.ActivePageIndex].Descriptor.Kvant + 1;
    ALabel[0,PageControl1.ActivePageIndex].Caption := 'Квант = ' +
IntToStr(Process[PageControl1.ActivePageIndex].Descriptor.Kvant);
    WriteToLog('Процессу ' + Pro-
cess[PageControl1.ActivePageIndex].Context^.Name +
               ' увеличен квант процессорного времени до ' + IntTo-
Str(Process[PageControl1.ActivePageIndex].Descriptor.Kvant));
  end
  else begin
    ShowMessage('Увеличение кванта процессорного времени невозмож-
но!');
  end;
end;

//Уменьшение приоритета процесса, если это возможно
procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  if Check-
PriorDown(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet)
then begin
    Process[PageControl1.ActivePageIndex].Descriptor.Prioritet :=
TPrioritetPro-
cess(Byte(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet)
- 1);
    ALabel[2,PageControl1.ActivePageIndex].Caption := 'Приоритет = ' +
PriorTo-
Str(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet);
    WriteToLog('Процессу ' + Pro-
cess[PageControl1.ActivePageIndex].Context^.Name +
               ' уменьшен приоритет до ' + PriorTo-
Str(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet));
  end
```

```
  else begin
    ShowMessage('Уменьшение приоритета невозможно!');
  end;
end;

//Увеличение приоритета процесса, если это возможно
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  if CheckPrio-
rUp(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet) then
begin
    Process[PageControl1.ActivePageIndex].Descriptor.Prioritet :=
TPrioritetPro-
cess(Byte(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet)
+ 1);
    ALabel[2,PageControl1.ActivePageIndex].Caption := 'Приоритет = ' +
PriorTo-
Str(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet);
    WriteToLog('Процессу ' + Pro-
cess[PageControl1.ActivePageIndex].Context^.Name +
               ' увеличен приоритет до ' + PriorTo-
Str(Process[PageControl1.ActivePageIndex].Descriptor.Prioritet));
  end
  else begin
    ShowMessage('Увеличение приоритета невозможно!');
  end;
end;

//Моделирование диспетчера
procedure TForm1.Timer1Timer(Sender: TObject);
var
  CurrentContext: PContext;
begin
  //Если процессор свободен, занимаем его следующим из очереди готовых
  if Processor.State = spEmpty then begin
    if StateReadys = ssEmpty then exit;
    //Изымаем следующий процесс из очереди готовых
    Out_Stack(Readys, Processor.Run);
    Processor.State := spBusy;
    Label5.Caption := 'Состояние = ' + StateProcessorTo-
Str(Processor.State);
    CurrentContext := ContextProcessByID(Processor.Run^.PID);
    Label6.Caption := 'Процесс = ' + CurrentContext^.Name;
    ListBox1.Items.Delete(0);
    Processor.Run.State := spRun;
    ALabel[1,Processor.Run^.PID].Caption := 'Состояние = ' + State-
ToStr(Processor.Run^.State);
    ALabel[3,Processor.Run^.PID].Caption := 'Текущая команда = ' +
CommandToStr(CurrentContext^.Command);
    WriteToLog('Процесс ' + CurrentContext^.Name + ' переведен в со-
стояние ВЫПОЛНЕНИЕ');
    if ListBox1.Items.Count = 0 then StateReadys := ssEmpty;
    //Если предыдущая команда выполнена
    if CurrentContext^.CountRun = CurrentContext^.CurrentRun then
begin
      //Переходим на следующую команду
      CurrentContext^.CommandLine := CurrentContext^.CommandLine + 1;
```

```
      CurrentContext^.Command := Par-
seCommand(ListBox[Processor.Run^.PID].Items[CurrentContext^.CommandLin
e]);
      CurrentContext^.CountRun := Par-
seInt(ListBox[Processor.Run^.PID].Items[CurrentContext^.CommandLine]);
      CurrentContext^.CurrentRun := 0;
      Label7.Caption := 'Выполняется ' + IntTo-
Str(CurrentContext^.CurrentRun) + ' из ' + IntTo-
Str(CurrentContext^.CountRun);
      ALabel[1,Processor.Run^.PID].Caption := 'Состояние = ' + State-
ToStr(Processor.Run^.State);
      ALabel[3,Processor.Run^.PID].Caption := 'Текущая команда = ' +
CommandToStr(CurrentContext^.Command);
      //Если команда ПРОЦЕССОР, то продолжаем работать,
      if CurrentContext^.Command = cPROCESSOR then begin
      end;
      //Если команда ВВОД/ВЫВОД, то перемещаем процесс в очередь ожи-
дающих
      if CurrentContext^.Command = cIO then begin
        In_Stack(Waits, Processor.Run);
        StateWaits := ssFully;
        ListBox2.Items.Add(CurrentContext^.Name);
        Processor.Run^.State := spWait;
        WriteToLog('Процесс ' + CurrentContext^.Name + ' помещен в
очередь ожидающих');
        ALabel[1,Processor.Run^.PID].Caption := 'Состояние = ' +
StateToStr(Processor.Run^.State);
        ALabel[3,Processor.Run^.PID].Caption := 'Текущая команда = '
+ CommandToStr(CurrentContext^.Command);
        //Процессор свободен
        Processor.State := spEmpty;
        Label5.Caption := 'Состояние = ' + StateProcessorTo-
Str(Processor.State);
        Label6.Caption := 'Процесс = -';
        Label7.Caption := 'Выполняется -  из -';
      end;
      //Если команды нет, то заканчиваем процесс
      if CurrentContext^.Command = cNONE then begin
        Processor.Run^.State := spNone;
        WriteToLog('Процесс ' + CurrentContext^.Name + ' завершен');
        ALabel[1,Processor.Run^.PID].Caption := 'Состояние = ' +
StateToStr(Processor.Run^.State);
        ALabel[3,Processor.Run^.PID].Caption := 'Текущая команда = '
+ CommandToStr(CurrentContext^.Command);
        //Процессор свободен
        Processor.State := spEmpty;
        Label5.Caption := 'Состояние = ' + StateProcessorTo-
Str(Processor.State);
        Label6.Caption := 'Процесс = -';
        Label7.Caption := 'Выполняется -  из -';
      end;
    end;
  end
  else begin
    //Если процессор занять какой-то работой
    CurrentContext := ContextProcessByID(Processor.Run^.PID);
```

```
    //Если он занять нужной работой, то определяем не закончилось ли
время для выполнения команды
    if CurrentContext^.Command = cPROCESSOR then begin
      CurrentContext^.CurrentRun := CurrentContext^.CurrentRun + 1;
      //Если команда исчерпала время
      if CurrentContext^.CountRun = CurrentContext^.CurrentRun then
begin
        //То переводим процесс в очредь готовых
        In_Stack(Readys, Processor.Run);
        StateReadys := ssFully;
        ListBox1.Items.Add(CurrentContext^.Name);
        WriteToLog('Процесс ' + CurrentContext^.Name + ' помещен в
очередь готовых');
        Processor.Run^.State := spReady;
        ALabel[1,Processor.Run^.PID].Caption := 'Состояние = ' +
StateToStr(Processor.Run^.State);
        ALabel[3,Processor.Run^.PID].Caption := 'Текущая команда = -';
        //Процессор свободен
        //По правилам необходимо выбрать следующий процесс из очереди
готовых, но
        //для лучшей иллюстрации работы процессора, сделаем это на
следующем кванте
        Processor.State := spEmpty;
        Label5.Caption := 'Состояние = ' + StateProcessorTo-
Str(Processor.State);
        Label6.Caption := 'Процесс = -';
        Label7.Caption := 'Выполняется -  из -';
      end
      else begin
        //Если нет, то смотрим отведенный квант
        if CurrentContext^.CurrentRun mod Processor.Run^.Kvant = 0
then begin
          Label7.Caption := 'Выполняется ' + IntTo-
Str(CurrentContext^.CurrentRun) + ' из ' + IntTo-
Str(CurrentContext^.CountRun);
        end;
      end;
    end;
  end;
end;

//Определение контекста процесса по идентификатору этого процесса
function TForm1.ContextProcessByID(ID: Integer): PContext;
var
  i: Integer;
begin
   for i := 0 to PageControl1.PageCount-1 do begin
     if Process[i].Descriptor.PID = ID then begin
       Result := Process[i].Context;
       exit;
     end;
   end;
   Result := nil;
end;

//Моделирование устройств ввода/вывода
procedure TForm1.Timer2Timer(Sender: TObject);
```

```
var
  CurrentContext: PContext;
begin
  //Если устройство ничем не занято, то пытаемся его занять
  if IODevice.State = spEmpty then begin
    //Если список ожидающих операции ввода/вывода пуст, то ничего не
делаем
    if StateWaits = ssEmpty then exit;
    //Изымаем следующий процесс из очереди ожидающих
    Out_Stack(Waits, IODevice.Run);
    IODevice.State := spBusy;
    CurrentContext := ContextProcessByID(IODevice.Run^.PID);
    ListBox2.Items.Delete(0);
    IODevice.Run.State := spRun;
    ALabel[1,IODevice.Run^.PID].Caption := 'Состояние = ' + State-
ToStr(IODevice.Run^.State);
    ALabel[3,IODevice.Run^.PID].Caption := 'Текущая команда = ' + Com-
mandToStr(CurrentContext^.Command);
    WriteToLog('Процесс ' + CurrentContext^.Name + ' осуществляет опе-
рации ввода/вывода');
    if ListBox2.Items.Count = 0 then StateWaits := ssEmpty;
  end
  else begin
    //Если устройство занято какой-то работой
    CurrentContext := ContextProcessByID(IODevice.Run^.PID);
    //Если он занять нужной работой, то определяем не закончилось ли
время для выполнения команды
    if CurrentContext^.Command = cIO then begin
      CurrentContext^.CurrentRun := CurrentContext^.CurrentRun + 1;
      //Если команда исчерпала время
      if CurrentContext^.CountRun = CurrentContext^.CurrentRun then
begin
        //ТО переводим процесс в очередь готовых
        In_Stack(Readys, IODevice.Run);
        StateReadys := ssFully;
        ListBox1.Items.Add(CurrentContext^.Name);
        WriteToLog('Процесс ' + CurrentContext^.Name + ' помещен в
очередь готовых');
        IODevice.Run^.State := spReady;
        ALabel[1,IODevice.Run^.PID].Caption := 'Состояние = ' + State-
ToStr(IODevice.Run^.State);
        ALabel[3,IODevice.Run^.PID].Caption := 'Текущая команда = -';
        //Устройство свободно и можно получать следующее задание
        IODevice.State := spEmpty;
        WriteToLog('Устройство ввода/вывода свободно');
      end;
    end;
  end;
end;

end.
```

Модуль Unit2.pas

```
unit Unit2;

interface
uses   SysUtils;
```

```
type
  //Тип-перечисление: Тип команды
  TCommand = (cNONE,
              cMEMORY,       //Память
              cPROCESSOR,    //Процессор
              cIO);          //Ввод/вывод

  //Тип-перечисление: Состояние процессора
  TStateProcessor = (spEmpty,  //Свободен
                     spBusy);  //Занят

 //Тип-перечисление: Состояние стека
  TStateStack = (ssEmpty,  //Пустой
                 ssFully);  //Занят

  //Тип-перечисление: Состояние процесса
  TStateProcess = ( spNone,   //Нет состояния
                    spRun,    //Выполнение
                    spReady,  //Готовность
                    spWait);  //Ожидание

  //Тип-перечисление: Приоритет процесса
  TPrioritetProcess = (ppLowest,  //Очень низкий
                       ppLow,     //Низкий
                       ppNormal,  //Нормальный
                       ppHigh,    //Высокий
                       ppHighest);//Очень высокий

  //Тип-запись: Дескриптор процесса
  TDescriptor = record
    PID: Integer;                  //Идентификатор процесса
    Kvant: Integer;                //Квант процесса
    State: TStateProcess;          //Состояние процесса
    Prioritet: TPrioritetProcess; //Приоритет процесса
  end;

  PContext = ^TContext;
  //Тип-запись: Контекст процесса
  TContext = record
    PID: Integer;                  //Идентификатор процесса
    Name: String[15];              //Имя процесса
    CommandLine: Integer;          //Счётчик команд
    Command: TCommand;             //Какая команда
    CountRun: Integer;             //Сколько надо выполнять команду
    CurrentRun: Integer;           //Счётчик выполнения команды
    Memory: Integer;                //Требуемое для процесса количество
памяти
  end;

  //Тип-запись: Процесс
  TProcess = record
    Descriptor: TDescriptor;      //Дескриптор процесса
    Context: PContext;           //Ссылка на контекст процесса
  end;

  PDescriptor = ^TDescriptor;
```

```
  //Тип-стек: Очередь дескриптеров
  PDescriptorStack = ^TDescriptorStack;
  TDescriptorStack = record
    Descriptor: PDescriptor;        //Дескриптор процесса
    Next: PDescriptorStack;         //Следующий элемент стека
  end;

  //Тип-запись: Процессор
  TProcessor = record
    Run: PDescriptor;               //Ссылка на выполняющийся процесс
    State: TStateProcessor;         //Состояние процессора
  end;

  //Процедура помещения в стек
  procedure    In_Stack(var    First:    PDescriptorStack;    AValue:
PDescriptor);
  //Процедура извлечения из стека
  procedure   Out_Stack(var   First:   PDescriptorStack;   Var   AValue:
PDescriptor);
  //Преобразования состояния процесса в строку текста
  function StateToStr(AState: TStateProcess): String;
  //Преобразования состояния процессора в строку текста
  function StateProcessorToStr(AState: TStateProcessor): String;
  //Преобразования приоритета процесса в строку текста
  function PriorToStr(APrior: TPrioritetProcess): String;
  //Преобразования команды процесса в строку текста
  function CommandToStr(ACommand: TCommand): String;
  //Проверка возможности уменьшения кванта времени
  function CheckKvantDown(AKvant: Integer): Boolean;
  //Проверка возможности увеличения кванта времени
  function CheckKvantUp(AKvant: Integer): Boolean;
  //Проверка возможности уменьшения приоритета
  function CheckPriorDown(APrior: TPrioritetProcess): Boolean;
  //Проверка возможности увеличения приоритета
  function CheckPriorUp(APrior: TPrioritetProcess): Boolean;
  //Извлечение данных из строки текста
  function ParseInt(Text: String): Integer;
  //Извлечение команды из строки текста
  function ParseCommand(Text: String): TCommand;

implementation

  procedure    In_Stack(var    First:    PDescriptorStack;    AValue:
PDescriptor);
  var
    Temp: PDescriptorStack;
  begin
    New(Temp);
    Temp^.Descriptor := AValue;
    Temp^.Next := First;
    First := Temp;
  end;

  procedure   Out_Stack(var   First:   PDescriptorStack;   Var   AValue:
PDescriptor);
  var
    Temp: PDescriptorStack;
```

```
begin
  Temp := First;
  First := First^.Next;
  AValue := Temp^.Descriptor;
  Dispose(Temp);
end;

function ParseInt(Text: String): Integer;
var
  temp: Integer;
begin
  temp := pos('-', Text);
  if temp = 0 then begin
    Result := 0;
    exit;
  end;
  temp := StrToInt(copy(Text, temp + 1, Length(Text) - temp));
  Result := temp;
end;

function ParseCommand(Text: String): TCommand;
var
  temp: Integer;
  tempStr: String;
begin
  temp := pos('-', Text);
  if temp = 0 then begin
    Result := cNONE;
    exit;
  end;
  tempStr := copy(Text, 1, temp-1);
  Result := cNONE;
  if tempStr = 'ПАМЯТЬ' then Result := cMEMORY;
  if tempStr = 'ПРОЦЕССОР' then Result := cPROCESSOR;
  if tempStr = 'ВВОД\ВЫВОД' then Result := cIO;
end;

function CheckPriorDown(APrior: TPrioritetProcess): Boolean;
begin
  Result := Aprior <> ppLowest;
end;

function CheckPriorUp(APrior: TPrioritetProcess): Boolean;
begin
  Result := Aprior <> ppHighest;
end;

function CheckKvantDown(AKvant: Integer): Boolean;
begin
  Result := AKvant > 0;
end;

function CheckKvantUp(AKvant: Integer): Boolean;
begin
  Result := AKvant < 10;
end;
```

```
  function StateProcessorToStr(AState: TStateProcessor): String;
  begin
    case AState of
    spEmpty:  Result := 'Свободен';
    spBusy:   Result := 'Занят';
    end;
  end;

  function CommandToStr(ACommand: TCommand): String;
  begin
    case ACommand of
    cNONE:      Result := 'Остановлен';
    cMEMORY:    Result := 'Запуск';
    cPROCESSOR: Result := 'Выполнение';
    cIO:        Result := 'Ввод/вывод';
    end;
  end;

  function StateToStr(AState: TStateProcess): String;
  begin
    case AState of
    spNone:  Result := 'Остановлен';
    spRun:   Result := 'Выплняется';
    spReady: Result := 'Готов';
    spWait:  Result := 'Ожидает';
    end;
  end;

  function PriorToStr(APrior: TPrioritetProcess): String;
  begin
    case APrior of
    ppLowest:  Result := 'Очень низкий';
    ppLow:     Result := 'Низкий';
    ppNormal:  Result := 'Нормальный';
    ppHigh:    Result := 'Высокий';
    ppHighest: Result := 'Очень высокий';
    end;
  end;
end.
```

Составитель:
Ляховец Михаил Васильевич

УПРАВЛЕНИЕ РЕСУРСАМИ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ

Направления подготовки:
230100 – Информатика и вычислительная техника;
230400 – Информационные системы и технологии;
230700 – Прикладная информатика

Печатается в полном соответствии с авторским оригиналом

Подписано в печать «___»_____ 20___г.
Формат бумаги 60×84 1/16. Бумага писчая. Печать офсетная.
Усл. печ. л. . Уч.-изд. л. . Тираж ___ экз. Заказ _____.

Сибирский государственный индустриальный университет
654007, г. Новокузнецк, ул. Кирова, 42.
Типография СибГИУ