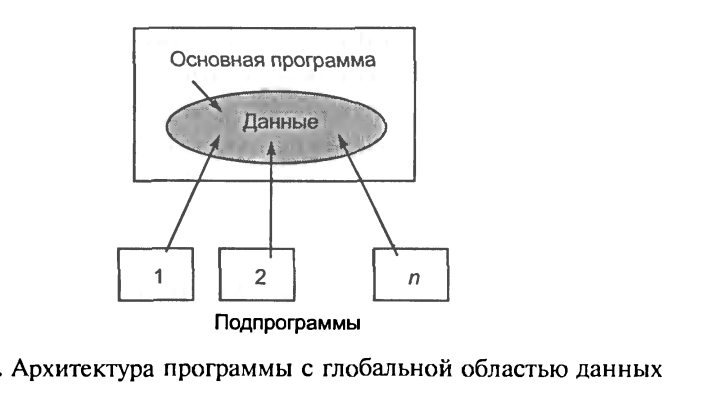
**Тема 1. Эволюция языков программирования. Основные принципы технологии структурного и объективно-ориентированного программирования**

Чтобы разобраться в существующих технологиях программирования и определить основные тенденции их развития, целесообразно рассматривать эти технологии в историческом контексте, выделяя основные этапы развития программирования как науки.

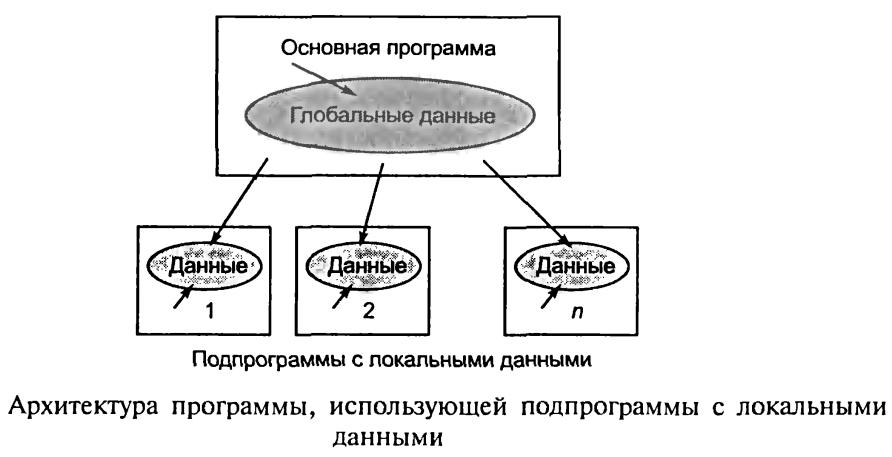
**Первый этап** — «стихийное» программирование. Этот этап охватывает период от момента появления первых вычислительных машин до середины 60-х годов XX в. В этот период практически отсутствовали сформулированные технологии и программирование фактически было искусством. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных (рис. 1.1).

Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно, отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

Появление ассемблеров позволило вместо двоичных или 16-ричных кодов использовать символические имена данных и мнемоники кодов операций. В результате программы стали более «читаемыми». Создание языков программирования высокого уровня, таких как FORTRAN и ALGOL, существенно упростило программирование вычислений, снизив уровень детализации операций. Это, в свою очередь, позволило увеличить сложность программ. Революционным было появление в языках средств, позволяющих оперировать подпрограммами. (Идея написания подпрограмм появилась гораздо раньше, но отсутствие средств поддержки в первых языковых средствах существенно снижало эффективность их применения.) Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы. Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части .



Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок в подпрограммах, было предложено размещать локальные данные .

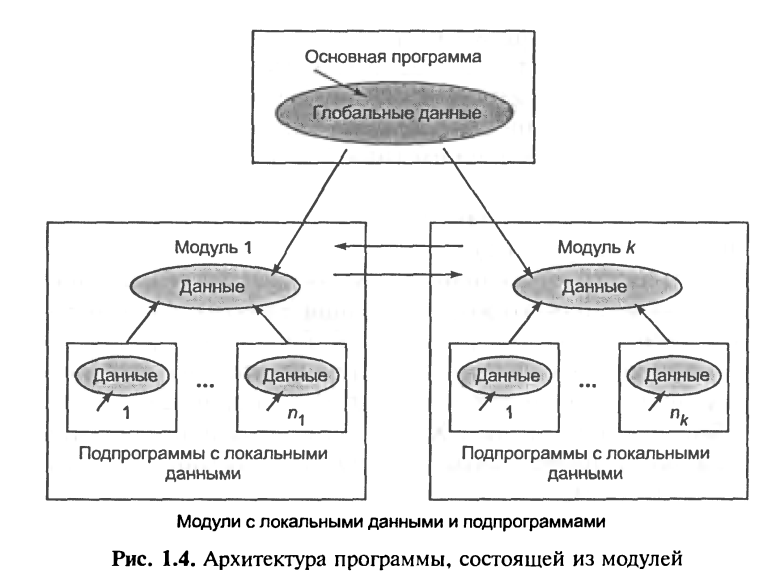


Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне. Однако появление средств поддержки подпрограмм позволило осуществлять разработку программного обеспечения нескольким программистам параллельно. В начале 60-х годов XX в. разразился «кризис программирования». Он выражался в том, что фирмы, взявшиеся за разработку сложного программного обеспечения, такого как операционные системы, срывали все сроки завершения проектов [8].

Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены. Объективно все это было вызвано несовершенством технологии программирования. Прежде всего стихийно использовалась разработка «снизу—вверх» — подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых затем пытались построить сложную программу.

В отсутствие четких моделей описания подпрограмм и методов их проектирования создание каждой подпрограммы превращалось в непростую задачу, интерфейсы подпрограмм получались сложными, и при сборке программного продукта выявлялось большое количество ошибок согласования. Исправление таких ошибок, как правило, требовало серьезного изменения уже разработанных подпрограмм, что еще более осложняло ситуацию, так как при этом в программу часто вносились новые ошибки, которые также необходимо было исправлять... В конечном итоге процесс тестирования и отладки программ занимал более 80 % времени разработки, если вообще когда-нибудь заканчивался. На повестке дня самым серьезным образом стоял вопрос разработки технологии создания сложных программных продуктов, снижающей вероятность ошибок проектирования. Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван «структурным» [19, 23].

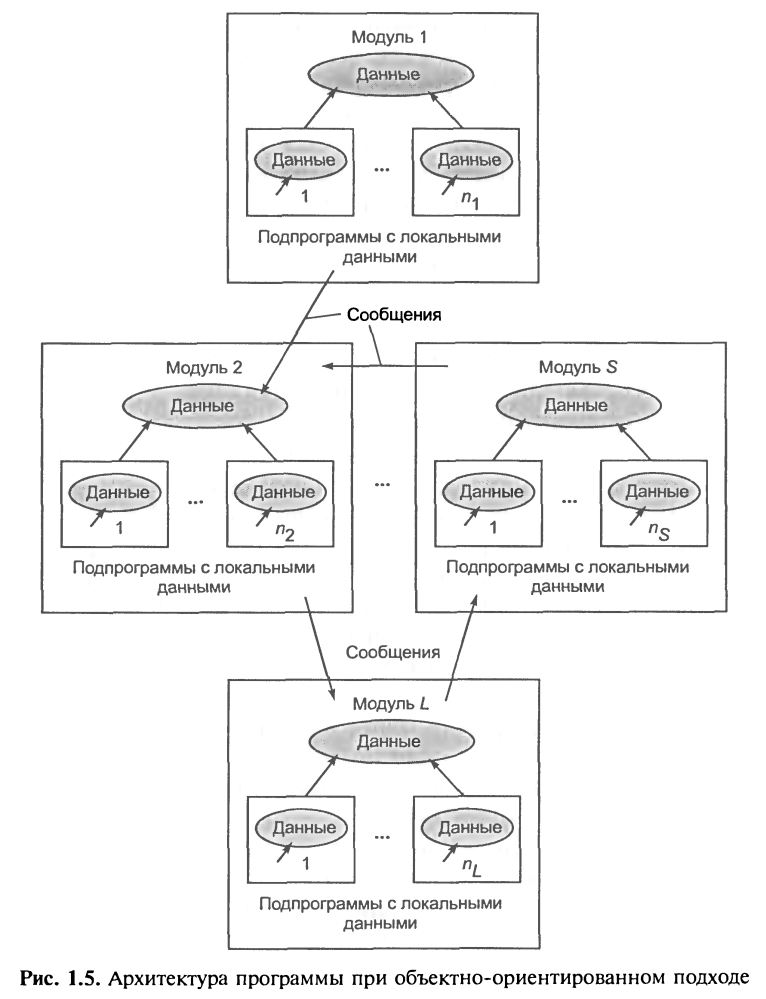
**Второй этап** — структурный подход к программированию (60—70-е годы XX в.). Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40—50 операторов) подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название процедурной декомпозиции. В отличие от используемого ранее процедурного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху—вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов — метод пошаговой детализации. Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования. Как правило, они включали основные «структурные» операторы передачи управления, поддерживали вложение подпрограмм, локализацию и ограничение области «видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, С.

Одновременно со структурным программированием появилось огромное количество языков, базирующихся на других концепциях, но большинство из них не выдержало конкуренции. Какие-то языки были просто забыты, идеи других были в дальнейшем использованы в следующих версиях развиваемых языков.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных. Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые модули (библиотеки подпрограмм), например модуль графических ресурсов, модуль подпрограмм вывода на принтер (рис. 1.4). Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и С (C++), языки Ада и Modula.

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов. Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, размер которых не превышает 100 000 операторов [10]. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема было предложено использовать объектный подход.

**Третий этап** — объектный подход к программированию (с середины 80-х до конца 90-х годов XX в.). Объектно-ориентированное программирование определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа {класса), а классы образуют иерархию с наследованием свойств [10, 24, 29]. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений (рис. 1.5).

Объектная структура программы впервые была использована в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х годах XX в. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования языке Smalltalk (70-е годы XX в.), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, C++, Modula, Java.

Основным достоинством объектно-ориентированного

программирования по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы. Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

Бурное развитие технологий программирования, основанных на объектном подходе, позволило решить многие проблемы. Так, были созданы среды, поддерживающие визуальное программирование, например Delphi, C++ Builder, Visual C++ и т. д. При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов.

Результатом визуального проектирования является заготовка будущей программы, в которую уже внесены соответствующие коды. Использование объектного подхода имеет много преимуществ, однако его конкретная реализация в объектно-ориентированных языках программирования, таких как Pascal и C++, имеет существенные недостатки:

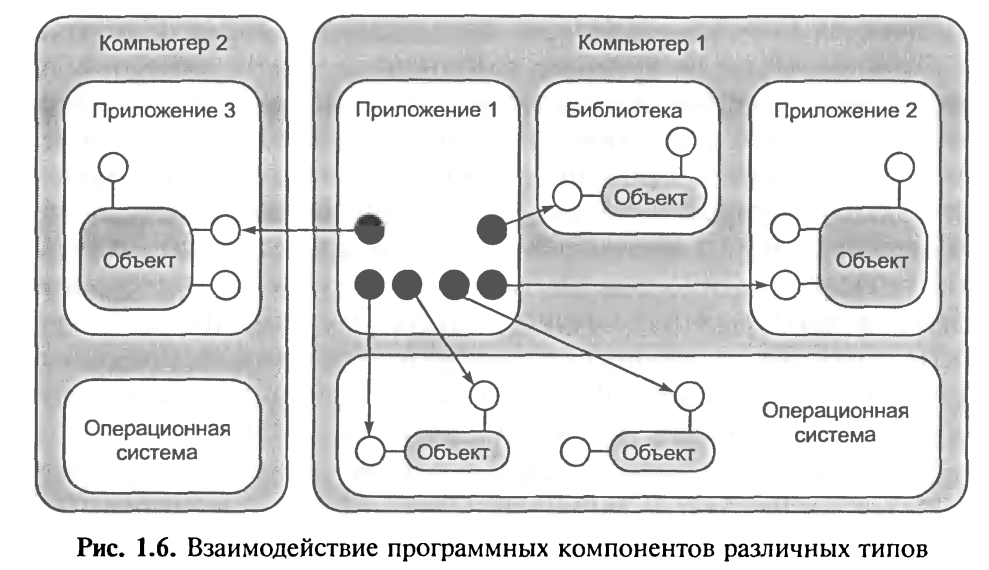
1) фактически отсутствуют стандарты компоновки двоичных результатов компиляции объектов в единое целое даже в пределах одного языка программирования. Компоновка объектов, полученных разными компиляторами C++, в лучшем случае проблематична, что приводит к необходимости разработки программного обеспечения с использованием средств и возможностей одного языка программирования высокого уровня и одного компилятора, а значит, требует наличия исходных кодов используемых библиотек классов;

2) изменение реализации одного из программных объектов, как минимум, связано с перекомпиляцией соответствующего модуля и перекомпоновкой всего программного обеспечения, использующего данный объект.

Таким образом, при использовании этих языков программирования сохраняется зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь к ресурсам друг друга. Связи модулей нельзя разорвать, но можно попробовать стандартизировать их взаимодействие, на чем и основан компонентный подход к программированию.

**Четвертый этап** — компонентный подход и CASE-технологии (с середины 90-х годов XX в. до нашего времени). Компонентный подход предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. На сегодня рынок объектов стал реальностью, так, в Интернете существуют узлы, предоставляющие большое количество компонентов, рекламой компонентов забиты журналы. Это позволяет программистам создавать продукты, хотя бы частично состоящие из повторно использованных частей, т. е. использовать технологию, хорошо зарекомендовавшую себя в области проектирования аппаратуры. Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model — компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture — общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология СОМ фирмы Microsoft является развитием технологии OLE I (Object Linking and Embedding — связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология СОМ определяет общую парадигму взаимодействия программ любых типов: библиотек, приложений, операционной системы, т. е. позволяет одной части программного обеспечения использовать функции (службы), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рис. 1.6). Модификация СОМ, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM — распределенная СОМ).

По технологии СОМ приложение предоставляет свои службы, используя специальные объекты — объекты СОМ, которые являются экземплярами классов СОМ. Объект СОМ так же, как обычный объект, включает поля и методы, но в отличие от обычных объектов каждый объект СОМ может реализовывать несколько интерфейсов, обеспечивающих доступ к его полям и функциям. Это достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса (по типу таблиц виртуальных методов). При этом интерфейс обычно объединяет несколько однотипных функций. Кроме того, классы СОМ поддерживают наследование интерфейсов, но не поддерживают наследования реализации, т. е. не наследуют код методов, хотя при необходимости объект класса-потомка может вызвать метод родителя.

Каждый интерфейс имеет имя, начинающееся с символа «I», и глобальный уникальный идентификатор IID (Interface IDentifier). Любой объект СОМ обязательно реализует интерфейс I Unknown (на схемах этот интерфейс всегда располагают сверху).

Использование этого интерфейса позволяет получить доступ к остальным интерфейсам объекта. Объект всегда функционирует в составе сервера — динамической библиотеки или исполняемого файла, которые обеспечивают функционирование объекта. Различают три типа серверов:

1) внутренний сервер; реализуется динамическими библиотеками, которые подключаются к приложению-клиенту и работают в одном с ними адресном пространстве, наиболее эффективный сервер, кроме того, он не требует специальных средств;

2) локальный сервер; создается отдельным процессом (модулем, ехе), который работает на одном компьютере с клиентом;

3) удаленный сервер; создается процессом, который работает на другом компьютере.

Например, Microsoft Word является локальным сервером. Он включает множество объектов, которые могут использоваться другими приложениями. Для обращения к службам клиент должен получить указатель на соответствующий интерфейс. Перед первым обращением к объекту клиент посылает запрос к библиотеке СОМ, хранящей информацию обо всех зарегистрированных в системе классах СОМ объектов, и передает ей имя класса, идентификатор интерфейса и тип сервера. Библиотека запускает необходимый сервер, создает требуемые объекты и возвращает указатели на объекты и интерфейсы. Получив указатели, клиент может вызывать необходимые функции объекта. Взаимодействие клиента и сервера обеспечивается базовыми механизмами СОМ или DCOM, поэтому клиенту безразлично местонахождение объекта. При использовании локальных и удаленных серверов в адресном пространстве клиента создается proxy-объект — заместитель объекта СОМ, а в адресном пространстве сервера СОМ — заглушка, соответствующая клиенту. Получив задание от клиента, заместитель упаковывает его параметры и, используя службы операционной системы, передает вызов заглушке. Заглушка распаковывает задание и передает его объекту СОМ. Результат возвращается клиенту в обратном порядке.

На базе технологии СОМ и ее распределенной версии DCOM были разработаны компонентные технологии, решающие различные задачи разработки программного обеспечения. OLE-automation или просто Automation (автоматизация) — технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений. Вводит понятие диспинтерфейса (dispinterface) — специального интерфейса, облегчающего вызов функций объекта. Эту технологию поддерживает, например, Microsoft Excel, предоставляя другим приложениям свои службы. ActiveX — технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения как сосредоточенного на одном компьютере, так и распределенного в сети. Предполагает использование визуального программирования для создания компонентов — элементов управления ActiveX. Полученные таким образом элементы управления можно устанавливать на компьютер дистанционно с удаленного сервера, причем устанавливаемый код зависит от используемой операционной системы. Это позволяет применять элементы управления ActiveX в клиентских частях приложений Интернета.

Основными преимуществами технологии ActiveX, обеспечивающими ей широкое распространение, являются:

1) быстрое написание программного кода, поскольку все действия, связанные с организацией взаимодействия сервера и клиента, берет на программное обеспечение СОМ, программирование сетевых приложений становится похожим на программирование для отдельного компьютера;

2) открытость и мобильность — спецификации технологии недавно были переданы в Open Group как основа открытого стандарта;

3) возможность написания приложений с использованием знакомых средств разработки, например Visual Basic, Visual C++, Borland Delphi, Borland C++ и любых средств разработки на Java;

4) большое количество уже существующих бесплатных программных элементов ActiveX (к тому же практически любой программный компонент OLE совместим с технологиями ActiveX и может применяться без модификаций в сетевых приложениях);

5) стандартность — технология ActiveX основана на широко используемых стандартах Интернет (TCP/IP, HTML, Java), с одной стороны, и стандартах, введенных в свое время Microsoft и необходимых для сохранения совместимости (COM, OLE). MTS (Microsoft Transaction Server — сервер управления транзакциями) — технология, обеспечивающая безопасность и стабильную работу распределенных приложений при больших объемах передаваемых данных. MIDAS (Multitier Distributed Application Server — сервер многозвенных распределенных приложений) — технология, организующая доступ к данным разных компьютеров с учетом балансировки нагрузки сети.

Все указанные технологии реализуют компонентный подход, заложенный в СОМ. Так, с точки зрения СОМ элемент управления ActiveX — внутренний сервер, поддерживающий технологию OLE-automation. Для программиста же элемент ActiveX — «черный ящик», обладающий свойствами, методами и событиями, который можно использовать как строительный блок при создании приложений. Технология CORBA, разработанная группой компаний ОМС (Object Management Group — группа внедрения объектной технологии программирования), реализует подход, аналогичный СОМ, на базе объектов и интерфейсов CORBA. Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ, и потому эту технологию можно использовать для создания распределенного программного обеспечения в гетерогенной (разнородной) вычислительной среде. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью специального посредника, названного VisiBroker, и другого специализированного программного обеспечения. Отличительной особенностью современного этапа развития технологии программирования, кроме изменения подхода, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые были названы CASE-технологиями (Computer-Aided Software/ System Engineering — разработка программного обеспечения программных систем с использованием компьютерной поддержки). Без средств автоматизации разработка достаточно сложного программного обеспечения на настоящий момент становится трудно осуществимой: память человека уже не в состоянии фиксировать все детали, которые необходимо учитывать при разработке программного обеспечения. На сегодня существуют CASE-техноло- гии, поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию. Появление нового подхода не означает, что отныне все программное обеспечение будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного программного обеспечения показывает, что он будет применяться достаточно широко.

|  |  |
| --- | --- |
| 1. Ассемблер | 1. SQL |
| 1. Паскаль | 1. Object PAL |
| 1. Си | 1. dBase |
| 1. С++ | 1. Java |
| 1. Фортран | 1. Пролог |
| 1. Ада | 1. Delphi |
| 1. Кобол | 1. Visual Basic |
| 1. ПЛ/1 | 1. Borland C++ |
| 1. Модула | 1. Visual C++ |
| 1. Бейсик | 1. PHP |
| 1. Лисп | 1. Python |

**Системное программное обеспечение**

В составе системного ПО можно выделить две составляющие:

1) базовое ПО;

2) сервисное ПО.

**Базовое ПО** — минимальный набор программных средств, обеспечивающих работу компьютера.

**Сервисное ПО** — программы и программные комплексы, которые расширяют возможности базового ПО и организуют более удобную среду работы пользователя.

В базовое ПО входят:

1) операционная система;

2) операционные оболочки (текстовые, графические);

3) сетевая операционная система.

Компьютер выполняет действия в соответствии с предписаниями программы, созданной на одном из языков программирования. При работе пользователя на компьютере часто возникает необходимость выполнять операции с прикладной программой в целом, организовать работу внешних устройств, проверить работу различных блоков, скопировать информацию и т. д. Эти операции используются для работы с любой программой. Поэтому целесообразно из всего многообразия операций, выполняемых компьютером, выделить типовые и реализовать их с помощью специализированных программ, которые следует принять в качестве стандартных средств, поставляемых вместе с аппаратной частью. Программы, организующие работу устройств и не связанные со спецификой решаемой задачи, вошли в состав комплекса программ, названного операционной системой.

**Операционная система** — совокупность программных средств, обеспечивающих управление аппаратной частью компьютера и прикладными программами, а также их взаимодействием между собой и пользователем.

Операционная система образует автономную среду, не связанную ни с одним из языков программирования. Любая прикладная программа связана с ОС и может эксплуатироваться только на компьютерах, где имеется аналогичная системная среда. Программа, созданная в среде одной ОС, не функционирует в среде другой ОС, если в ней не обеспечена возможность конвертации (преобразования) программ.

Для работы с ОС необходимо овладеть языком этой среды — совокупностью команд, структура которых определяется синтаксисом этого языка.

ОС выполняет следующие функции:

• управление работой каждого блока ПО и их взаимодействием;

• управление выполнением программ;

• организацию хранения информации во внешней памяти;

• взаимодействие пользователя с компьютером (поддержку

интерфейса пользователя).

Обычно ОС хранится на жестком диске, а при его отсутствии выделяется специальный гибкий диск, который называется системным диском. При включении компьютера ОС автоматически загружается с диска в оперативную память и занимает в ней определенное место. ОС создается не для отдельной модели компьютера, а для серии компьютеров, в структуру которых заложена и развивается во всех последующих моделях определенная концепция.

Операционные системы для ПК делятся:

• на одно- и многозадачные (в зависимости от числа параллельного выполнения прикладных процессов);

• одно- и многопользовательские (в зависимости от числа пользователей, одновременно работающих с ОС);

• непереносимые и переносимые на другие типы компьютеров;

• несетевые и сетевые, обеспечивающие работу в ВС ЭВМ.

Операционные системы, как правило, сравнивают по следующим критериям:

• управлению памятью (максимальный объем адресуемого пространства, технические показатели использования памяти);

• функциональным возможностям вспомогательных программ (утилит) в составе ОС;

• наличию компрессии магнитных дисков;

• возможности архивирования файлов;

• поддержке многозадачного режима работы;

• наличию качественной документации;

• условию и сложности процесса инсталляции.

Наиболее распространенными ОС для персональных компьютеров в настоящее время являются:

• Windows 2000 (Microsoft);

• Windows XP (Microsoft);

• UNIX;

• OS/2 (IBM).

ОС OS/2 разработана IBM для ПК на основе системной прикладной архитектуры, ранее используемой для больших ЭВМ. Это многозадачная, многопользовательская ОС, обеспечивающая как текстовый, так и графический интерфейс пользователя. OS/2 обеспечивает:

• поддержку графического интерфейса пользователя;

• одновременную обработку нескольких приложений;

• многопоточную обработку нескольких задач одного

приложения;

• 32-разрядную обработку данных;

• сжатие данных при записи на магнитный диск;

• защиту памяти.

Важными особенностями OS/2 является высокопроизводительная файловая система HPFS (High Performance File System), имеющая преимущества для серверов БД (в отличие от MS DOS поддерживаются длинные имена файлов), а также поддержка мультипроцессорной обработки — до 16 процессоров типа Intel и Power PC. OS/2 Warp имеет встроенный доступ в Интернет, систему распознавания речи VoiceType, интегрированную версию Lotus Notes Mail для передачи электронной почты. В OS/2 могут выполняться прикладные программы Windows 3.x, но не могут выполняться приложения, работающие в среде Windows NT.

ОС UNIX, созданная корпорацией Bell Laboratory, реализует принцип открытых систем и широкие возможности по объединению в составе одной вычислительной системы разнородных технических и программных средств. UNIX обладает следующими свойствами:

• переносимость прикладных программ с одного компьютера

на другой;

• поддержка распределенной обработки данных в сети ЭВМ;

• сочетаемость с процессорами RISC. 24

UNIX получила распространение для суперкомпьютеров, работающих станций и профессиональных ПК, имеет большое число версий, разработанных различными производителями ПО.

**Сетевая ОС** — это комплекс программ, обеспечивающий обработку, передачу и хранение данных в сети. Сетевая ОС предоставляет пользователям различные виды сетевых служб (управление файлами, электронная почта, процессы управления сетью и т. д.), поддерживает работу в абонентских системах. Сетевые ОС используют архитектуру «клиент — сервер» или одноранговую архитектуру. Вначале эти ОС поддерживали только ЛВС, сейчас они распространяются и на объединения ЛВС. Наибольшее распространение имеют Windows 95, LAN Server, NetWare.

**Windows NT** является многозадачной, предназначенной для архитектуры «клиент — сервер» и использования различных протоколов транспортного уровня сетевой ОС (IPX/SPX, TCP/IP, Net Bios), имеет 32-разрядную архитектуру и обеспечивает следующие функции локальной сети:

• возможность каждой абонентской системы в сети быть сервером или клиентом;

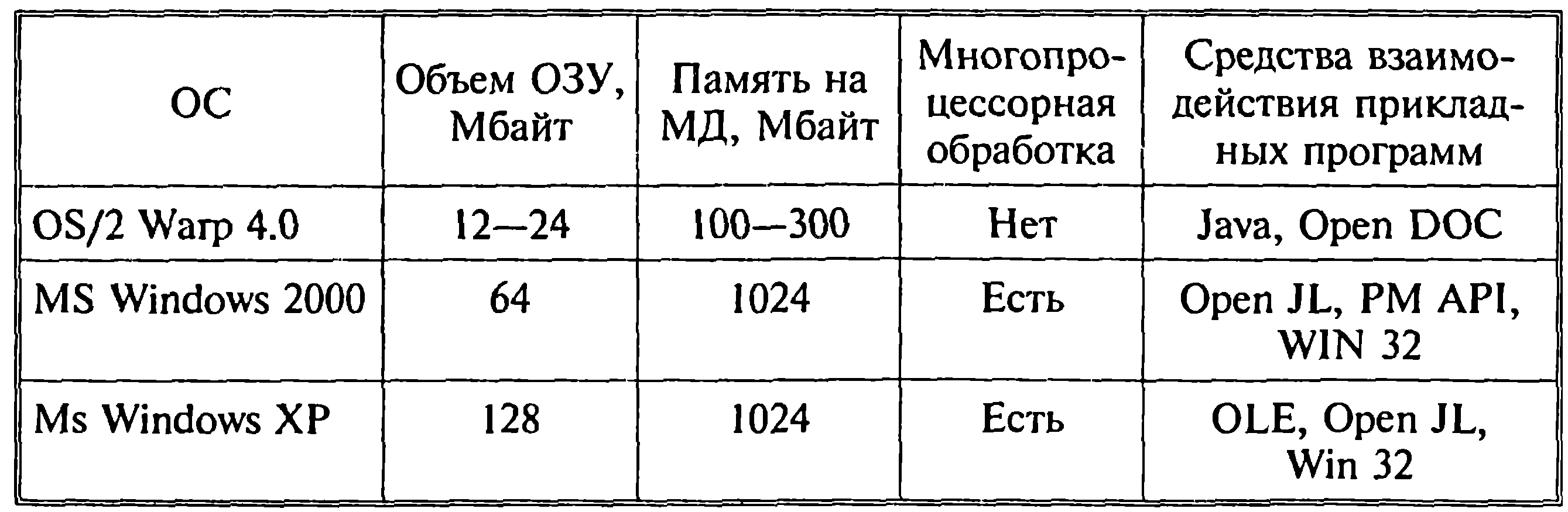
• совместную работу группы пользователей;

• адресацию оперативной и внешней памяти большого размера;

• многозадачность и многопоточность обработки данных;

• поддержку мультипроцессорной обработки и др.

В табл. приведены характеристики некоторых ОС.



**Операционные оболочки** — специальные программы, предназначенные для облегчения общения пользователя с командами ОС. Операционные оболочки имеют текстовый и графический варианты интерфейса конечного пользователя. Наиболее популярны такие текстовые оболочки MS DOS, как Norton Commander, DOS Navigator, Volkov Commander и др. Эти программы существенно упрощают задание управляющей информации для выполнения команд ОС, уменьшают сложность работы конечного пользователя.

Среди графических оболочек MS DOS можно выделить Windows 3.1, Windows 3.11 for Work Group, которые расширяют набор основных (диспетчер файлов, графический редактор, текстовый редактор и т. д.) и сервисных функций, обеспечивающих создание пользователю систем обработки информации в составе одноранговых локальных сетей.

Расширением базового ПО компьютера является набор сервисных, дополнительно устанавливаемых программ, которые можно классифицировать по функциональному признаку

следующим образом:

• программы диагностики работоспособности компьютера;

• антивирусные программы, обеспечивающие защиту компьютера, обнаружение и восстановление зараженных файлов;

• программы обслуживания дисков, обеспечивающие проверку качества поверхности магнитного диска, контроль сохранности файловой системы на логическом и физическом уровнях, сжатие дисков, резервное копирование данных на внешние носители и др.;

• программы архивирования данных, которые обеспечивают процесс сжатия информации с целью уменьшения объема памяти для ее хранения;

• программы обслуживания сети.

Эти программы часто называют утилитами.

**Утилиты** — программы, служащие для выполнения вспомогательных операций обработки данных или обслуживания компьютеров.

Наибольшее распространение имеют следующие утилиты:

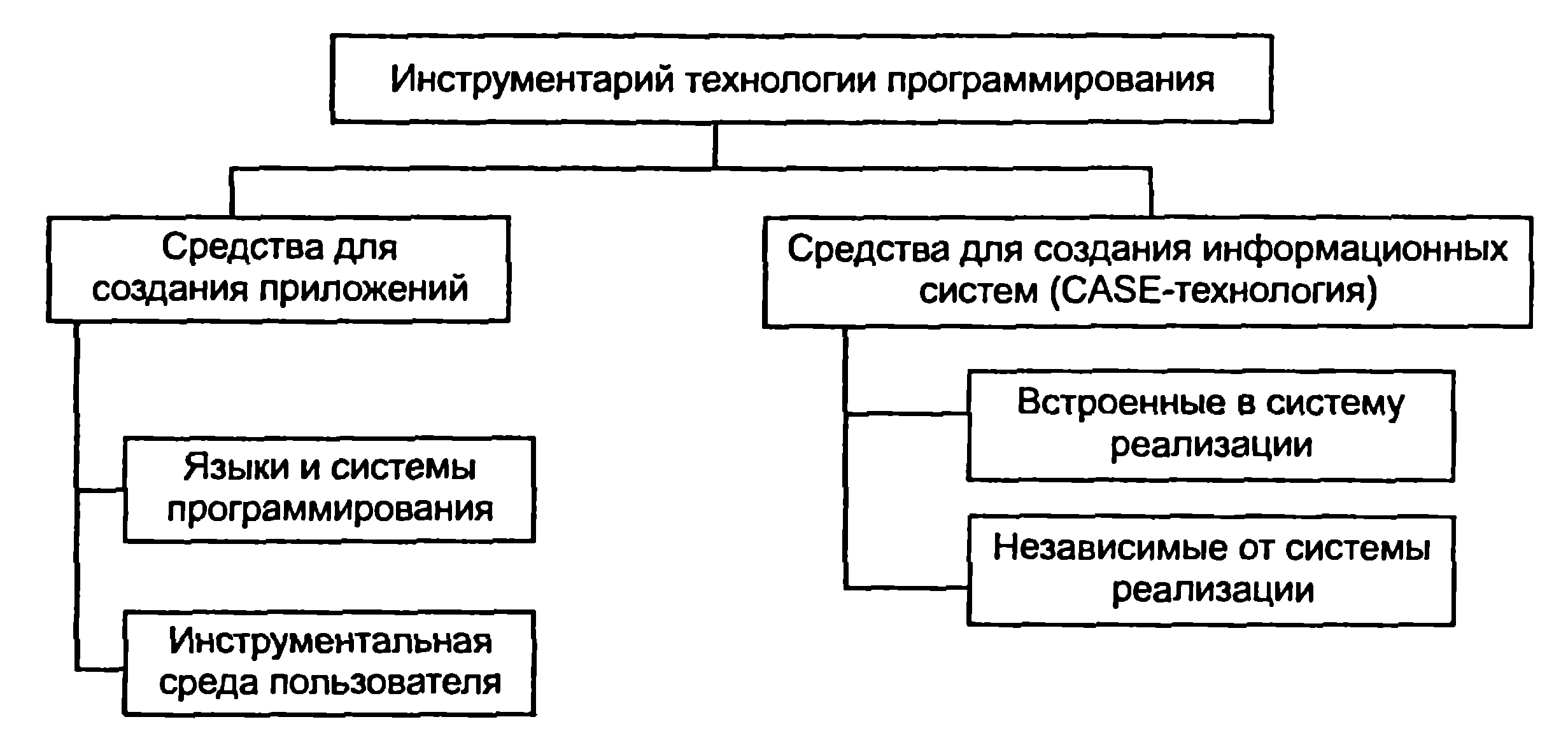
• Norton Utilities;

• программа резервного копирования HP Colorado BackUp for Windows 95.

Антивирусные программы (AVP Toolkit У Касперского; Norton Antivirus for Windows 95 (автоматическое создание резервных копий данных boot — секторов и критических файлов); Anti-Virus (Central Point Soft Ware Inc.) — обнаружение зараженных файлов без исправления как в DOS, так и в Windows и др.).

**Инструментарий технологии программирования**

Инструментарий технологии программирования — это программные продукты, предназначенные для поддержки технологии программирования.



**Средства для создания приложений** — совокупность языков и систем программирования, инструментальные среды пользователя, а также различные программные компоненты для отладки и поддержки создаваемых программ.

**Язык программирования** — это формализованный язык для описания алгоритма решения задач на компьютере. Языки программирования можно условно разделить на следующие классы:

• машинные языки — это языки, воспринимаемые аппаратной частью компьютера (машинные коды);

• машинно-ориентированные языки, отражающие структуру конкретного типа компьютера (ассемблер);

• процедурно-ориентированные языки — это языки, в которых имеется возможность описания программы как совокупности процедур, или подпрограмм (Си, Паскаль и др.);

• проблемно-ориентированные языки, предназначенные для решения задач определенного класса (ЛИСП, ПРОЛОГ).

Другой классификацией языков является их деление на языки, ориентированные на реализацию основ структурного программирования, основанного на модульной структуре программного продукта и типовых управляющих структурах алгоритмов обработки данных различных программных модулей, и объектно-ориентированные языки, поддерживающие понятие объектов, их свойств и методов обработки.

**Системы программирования** включают:

• компилятор (транслятор);

• интегрированную среду разработки программ (не всегда);

• отладчик;

• средства оптимизации кода программ;

• набор библиотек;

• редактор связей;

• сервисные средства (утилиты) (для работы с библиотеками, текстовыми и двоичными файлами);

• справочные системы;

• систему поддержки и управления продуктами программного комплекса.

Компилятор транслирует всю программу без ее выполнения. Трансляторы (интерпретаторы) выполняют пооперационную обработку и выполнение программы.

Отладчики (debugger) — специальные программы, предназначенные для трассировки и анализа выполнения других программ. Трассировка — это обеспечение выполнения в пооператорном варианте.

**Инструментальная среда пользователя** — это специальные средства, встроенные в пакеты прикладных программ, такие как:

• библиотека функций, процедур, объектов и методов обработки;

• макрокоманды;

• клавишные макросы;

• языковые макросы;

• конструкторы экранных форм и объектов;

• генераторы приложений;

• языки запросов высокого уровня;

• конструкторы меню и др.

**Интегрированные среды разработки программ** объединяют набор средств для их комплексного применения на технологических этапах создания программы.

Средства для создания информационных систем (ИС) и технологий поддерживают полный цикл проектирования сложной информационной системы или технологии от исследования объекта автоматизации до оформления проектной и прочей документации на информационную систему или технологию. Они позволяют вести коллективную работу над проектом за счет возможности работы в локальной сети, экспорта-импорта любых фрагментов проекта, организации управления проектом.

Одним из современных средств разработки ИС является **CASE-технология** (CASE — Computer-Aided System Engineering) — программный комплекс, автоматизирующий весь технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем.

Средства CASE-технологий делятся:

• на встроенные в систему реализации — все решения по проектированию и реализации привязки к выбранной СУБД;

• независимые от системы реализации — все решения по проектированию ориентированы на унификацию (определение) начальных этапов жизненного цикла программы и средств их документирования, обеспечивают большую гибкость в выборе средств реализации.

Основное достоинство CASE-технологии — это поддержка коллективной работы над проектом за счет возможности работы в локальной сети разработчиков, экспорта (импорта) любых фрагментов проекта, организованного управления проектами.

В некоторых CASE-системах поддерживается кодогенерация программ — создание каркаса программ и создание полного продукта.

Примеры программных продуктов для создания приложений: Visual C++, Delphi, Visual Basic и т. д.

**Тема 2. Системное программирование в ОС Windows**

**Задачи системного программирования**

Программная система это набор функций, при помощи которых можно решить любую задачу из некоторой предметной области.

Первыми серьезными программными системами были именно операционные системы, поэтому и основные концепции системного программирования отрабатывались при разработке и реализации операционных систем. Затем эти технологии использовались при разработке других программных систем, таких как, например, системы управления базами данных (СУБД). Классическое системное программирование рассматривает круг вопросов, связанных с синхронизацией и диспетчеризацией потоков и процессов, обменом данными между процессами, управлением устройствами компьютера и файлами. В последнее время большое внимание при проектировании систем также уделяется и обеспечению безопасности данных, что вызвано возросшими угрозами несанкционированного доступа к данным.

**Назначение и типы операционных систем**

Физическими или аппаратными ресурсами компьютера называются физические устройства, из которых состоит компьютер. К таким устройствам относятся центральный процессор, оперативная память, внешняя память, шины передачи данных и различные устройства ввода-вывода информации.

Логическими или информационными ресурсами компьютера называются данные и программы, которые хранятся в памяти компьютера. Когда говорят обо всех ресурсах компьютера, включая как физические, так и логические ресурсы, то обычно используют термины ресурсы компьютера или системные ресурсы.

Для выполнения на компьютере какой-либо программы необходимо, чтобы она имела доступ к ресурсам компьютера. Этот доступ обеспечивает операционная система. Можно сказать, что операционная система — это комплекс программ, который обеспечивает доступ к ресурсам компьютера и управляет ими. Назначение операционной системы состоит в обеспечении пользователя программными средствами для использования ресурсов компьютера и эффективном разделении этих ресурсов между пользователями. Главными функциями операционной системы являются управление ресурсами компьютера и диспетчеризация или планирование этих ресурсов.

Все программы, которые работают на компьютере под управлением операционной системы, называются пользовательскими программами. Совокупность пользовательских программ, которая предназначена для решения определенной задачи, называется приложением. Если операционная система одновременно может исполнять только одну пользовательскую программу, то она называется однопрогроммной или однопользовательской. Если же под управлением операционной системы могут одновременно выполняться несколько пользовательских программ, то такая операционная система называется мультипрограммной или многопользовательской.

В зависимости от назначения операционной системы и аппаратуры компьютера, на котором она работает, можно определить несколько типов операционных систем. Если операционная система может работать только на компьютере с одним процессором, то такая операционная система называется однопроцессорной. Если же операционная система может работать также и на компьютере, который содержит несколько процессоров, то такая операционная система называется мультипроцессорной.

Следует делать различие между операционными системами, которые предназначены для обработки информации под управлением пользователя, и операционными системами, которые предназначены для управления объектами при помощи компьютера в реальном времени без участия пользователя. Такими объектами могут быть, например, робот или самолет. Операционная система, предназначенная для работы в режиме реального времени, называется операционной системой реального времени. Главное отличие операционных систем реального времени заключается в их быстром реагировании на внешние события и надежности функционирования. Если пользователь, сидя у компьютера, будет только раздражен медленной или ненадежной работой операционной системы, то медленная или ненадежная работа операционной системы реального времени может вызвать поломку оборудования и аварию.

В дальнейшем будут рассматриваться только операционные системы фирмы Microsoft – Windows. Они используют один и тот же интерфейс для программирования приложений — Win32 API.

**Интерфейс программирования приложений Win32 API**

Интерфейс программирования приложений Win32 API представляет собой набор функций и классов, которые используются для программирования приложений, работающих под управлением операционных систем фирмы Microsoft.

Функционально Win32 API подразделяется на следующие категории:

□ Base Services (базовые сервисы). Функции базовых сервисов обеспечивают приложениям доступ к ресурсам компьютера.

□ Common Control Library (библиотека общих элементов управления). Содержит классы окон, которые часто используются в приложениях;

□ Graphics Device Interface (интерфейс графических устройств). Обеспечивает функции для вывода графики на дисплей, принтер и другие графические устройства;

□ Network Services (сетевые сервисы) Используются при работе компьютеров в компьютерных сетях;

□ User Interface (интерфейс пользователя). Обеспечивает функции для взаимодействия пользователя с приложением, используя окна для ввода-вывода информации;

□ Windows NT Access Control (управление доступом для Windows NT). Содержит функции, которые используются для защиты информации путем контроля и ограничения доступа к защищаемым объектам;

□ Windows Shell (оболочка Windows);

□ Windows System Information (информация о системе Windows). Категории Windows Shell и Windows System Information содержат соответственно функции для работы с оболочкой и конфигурацией операционной системы Windows.

*В курсе системного программирования главным образом изучается назначение и использование функций из категорий Base Services и Windows NT Access Control. Функции из категорий Common Control Libraiy, Graphics Device Interface и User Interface используются для разработки интерфейса приложений, а курс, который изучает назначение и использование этих функций, как правило, называется "Программирование пользовательских интерфейсов в Windows".*

*В связи с тем, что программирование графических пользовательских интерфейсов в Windows само по себе является довольно трудоемким занятием, мы будем изучать функции ядра Windows, работая только с консольными приложениями. Это упростит изложение предмета и избавит нас от большого количества кода, не относящегося к существу рассматриваемых вопросов.*

**Типы данных в Win32 API**

Прежде всего заметим, что интерфейс программирования приложений Win32 API ориентирован на язык программирования С или, в более широком смысле, на процедурные языки программирования. Поэтому в этом интерфейсе, не используются такие возможности языка программирования C++, как классы, ссылки и механизм обработки исключений.

Чтобы сделать интерфейс Win32 API более независимым от конкретного языка программирования разработчики этого интерфейса определили новые простые типы данных. Эти типы данных используются в прототипах функций интерфейса Win32 API.

Новые простые типы данных определены как синонимы простых типов данных языка программирования С. Чтобы отличать эти типы от других типов, их имена определены прописными буквами. Общее количество простых типов данных, определенных в интерфейсе Win32 API, довольно велико. Поэтому ниже приведены определения только тех простых типов данных из этого интерфейса, которые очевидным образом переименовывают простые типы данных языка программирования С.

typedef char CHAR;

typedef unsigned char UCHAR;

typedef UCHAR \*PUCHAR;

typedef unsigned char BYTE;

typedef BYTE \*PBYTE;

typedef BYTE \*LPBYTE;

typedef short SHORT;

typedef unsigned short USHORT;

typedef USHORT \*PUSHORT;

typedef unsigned short WORD;

typedef WORD \*PWORD;

typedef WORD \*LPWORD;

typedef int INT;

typedef int \*PINT;

typedef int \*LPINT;

typedef int BOOL;

typedef BOOL \*PBOOL;

typedef BOOL \*LPBOOL

typedef unsigned int UINT;

typedef unsigned int \*PUINT;

typedef long LONG;

typedef long \*LPLONG;

typedef unsigned long ULONG;

typedef ULONG \*PULONG;

typedef unsigned long DWORD;

typedef DWORD \*PDWORD;

typedef DWORD \*LPDWORD;

typedef float FLOAT;

typedef FLOAT \*PFLOAT;

typedef void \*LPVOID;

typedef CONST void \*LPCVOID;

*Остальные простые типы данных, определенные в интерфейсе Win32 API, имеют, как правило, специфическое назначение и поэтому они будут описаны при их использовании.*

Кроме того, в интерфейсе Win32 API определены символические константы false и true для обозначения соответственно ложного и истинного логических значений.

#ifndef FALSE

#define FALSE 0

#endif

#ifndef TRUE

#define TRUE 1

#endif

**Объекты и их дескрипторы в Windows**

Объектом в Windows называется структура данных, которая представляет системный ресурс. Таким ресурсом может быть, например, файл, канал, графический рисунок. Операционные системы Windows предоставляют приложению объекты трех категорий:

□ User. Включает объекты, которые используются приложением для интерфейса с пользователем. Например, окна и курсоры;

□ Graphics Device Interface. Включает объекты, которые используются для вывода информации на графические устройства. Например, кисти и перья;

□ Kernel. Включает объекты ядра операционной системы Windows. Например, файлы и каналы.

При изучении системного программирования подробно рассматриваются только объекты категории Kernel. *Объекты двух оставшихся категорий рассматриваются при изучении программирования графических интерфейсов.*

Под доступом к объектам понимается возможность приложения выполнять над объектом некоторые функции. Приложение не имеет прямого доступа к объектам, а обращается к ним косвенно. Для этого в операционных системах Windows каждому объекту ставится в соответствие дескриптор (handle). В Win32 API дескриптор имеет тип handle. Дескриптор объекта представляет собой запись в таблице, которая поддерживается системой и содержит адрес объекта и средства для идентификации типа объекта. Дескрипторы объектов создаются операционной системой и возвращаются функциями Win32 API, которые создают объекты. За редким исключением, эти функции имеют вид createobject, где слово object заменяется именем конкретного объекта. Например, процесс создается при помощи вызова функции createProcess. Такие функции возвращают дескриптор созданного объекта. Если это значение не равно null (или отрицательному значению), то объект создан успешно.

После завершения работы с объектом его дескриптор нужно закрыть, используя функцию closeHandle, которая имеет следующий прототип:

BOOL CloseHandle(

HANDLE hObject // дескриптор объекта

);

При успешном завершении функция closeHandle возвращает ненулевое значение, в противном случае — false. Функция closeHandle удаляет дескриптор объекта, но сам объект удаляется не всегда. Дело в том, что в Windows на один и тот же объект могут ссылаться несколько дескрипторов, которые создаются другими функциями для доступа к уже созданному ранее объекту. Функция closeHandle уничтожает объект только в том случае, если на него больше не ссылается ни один дескриптор.

**Тема 3. Потоки и процессы**

**Определение потока**

Исполняя программу, процессор последовательно выполняет инструкции программы, иногда осуществляя переходы в зависимости от некоторых условий. Такая последовательность выполнения инструкций программы называется потоком управления внутри программы. Поток управления зависит от начального состояния переменных, которые используются в программе. В общем случае различные исходные данные порождают различные потоки управления. Поток управления можно представить как нить в программе, на которую нанизаны инструкции, выполняемые микропроцессором. Поэтому часто поток управления также называется нитью (thread). В русскоязычной литературе за потоком управления закрепилось название поток. Рассмотрим следующую программу, которая выводит минимальное число из двух целых чисел или сообщение о том, что числа равны.

#include <iostream.h>

int main()

{

int a, ' b;

cout « "Input two integers: ";

cin » a » b;

if (a == b)

{

cout « "There is no min." « endl;

return 0;}

if (а < Ь)

cout « "min = " « а « endl;

else

cout « "min = " « b « endl;

return 0;

}

Предположим, что перегруженные операторы ввода-вывода не образуют новых потоков. Тогда в зависимости от входных данных эта программа образует один из трех возможных потоков управления. А именно, если выполняется условие (а == Ь), то образуется поток:

cout « "Input two integers: ";

cin » a » b;

if (a == b)

{

cout « "There is no min." « endl;

return 0;

}

Если выполняется условие (а < b), то образуется поток:

cout « "Input two integers: ";

cin ». a » b;

if (a == b)

if (a < b)

cout « "min = " « a « endl;

return 0;

Если же выполняется условие (а > b), то образуется поток

cout « "Input two integers: ";

cin » a » b;

if (a == b)

if (a < b)

cout « "min = " « b « endl;

return 0;

Говорят, что программа является многопоточной, если в ней может одновременно существовать несколько потоков. Сами потоки в этом случае называются параллельными. Если в программе одновременно может существовать только один поток, то такая программа называется однопоточной. Например, следующая программа, которая просто вычисляет сумму двух чисел, является однопоточной:

#include <iostream.h>

int sum(int a, int b)

{

return a + b;

}

int main()

{

int a, b;

int с = 0;

cout « "Input two integers: ";

cin » 'a » b;

с = sum (a, b) ;

cout « "Sum = " « с « endl;

return 0;

}

Теперь предположим, что после вызова функции sum функция main не ждет возвращения значения из функции sum, а продолжает выполняться. В этом случае получим программу, состоящую из двух потоков, один из которых определяется функцией main, а второй — функцией sum. Причем эти потоки независимы, т. к. они не имеют доступа к общим или, другими словами, разделяемым переменным. Правда в этом случае не гарантируется, что поток main выведет сумму чисел а и b, т. к. инструкция вывода значения суммы может отработать раньше, чем поток sum вычислит эту сумму.

Из этих рассуждений видно, что для того чтобы отметить функцию, которая порождает новый поток в программе, должна использоваться специальная нотация. В операционных системах Windows для обозначения того, что функция образует поток, используются специальные спецификаторы функции. Такая функция обычно также называется потоком.

**Контекст потока**

В общем случае содержимое памяти, к которой поток имеет доступ во время своего исполнения, называется контекстом потока. Определим, каким ограничениям на доступ к памяти должны удовлетворять функции, чтобы их можно было безопасно вызывать в параллельных потоках. Для этого рассмотрим следующую функцию:

int f(int n)

if (n > 0)

-- n;

if (n < 0)

++ n;

return n;

} Сколько бы раз эта функция не вызывалась параллельно работающими потоками, она будет корректно изменять значение переменной n, т. к. эта переменная является локальной в функции f. To есть для каждого нового вызова функции f будет создан новый локальный экземпляр переменной n. Такая функция f называется безопасной для потоков. Теперь введем глобальную переменную n и изменим нашу функцию следующим образом:

int n ;

void g()

{

if (n > 0)

--n;

if (n < 0)

++n;

}

В этом случае параллельный вызов функции g несколькими потоками может дать некорректное изменение значения переменной n, т. к. значение этой переменной будет изменяться одновременно несколькими функциями g. В этом случае функция g не является безопасной для потоков.

Та же проблема встречается и в случае, когда функция использует статические переменные. Для разбора этого случая рассмотрим функцию

int count()

{

static int n = 0;

++n;

return n;

}

которая возвращает количество своих вызовов. Если эта функция будет вызвана несколькими параллельно исполняемыми потоками, то нельзя точно определить значение переменной n, которое вернет эта функция, т. к. это значение изменяется всеми потоками параллельно.

В общем случае функция называется повторно входимой или реентерабельной (reentrant или reenterable), если она удовлетворяет следующим требованиям:

* не использует глобальные переменные, значения которых изменяются параллельно исполняемыми потоками;
* не использует статические переменные, определенные внутри функции;
* не возвращает указатель на статические данные, определенные внутри функции.

В системном программировании часто также рассматриваются программы в кодах микропроцессора, выполнение которых может прерываться и возобновляться в любой момент времени. Причем одна и та же программа может запускаться прежде, чем завершилось исполнение предыдущего экземпляра этой программы. В этом случае также необходимо, чтобы программный код допускал корректное параллельное выполнение нескольких экземпляров программы. Это условие обеспечивается в том случае, если программа не изменяет свой код во время исполнения. Здесь под кодом подразумеваются как команды, так и данные, принадлежащие программе. Программа в кодах микропроцессора, которая не изменяет свой код, также называется реентерабельной.

В дополнение к реентерабельным функциям определяют также функции, безопасные для вызова параллельно исполняемыми потоками. Функция называется безопасной для потоков, если она обеспечивает блокировку доступа к ресурсам, которые она использует. В этом случае решается задача взаимного исключения доступа к разделяемым ресурсам, используя примитивы синхронизации.

Очевидно, что если функция не является реентерабельной, то она также не является и безопасной для потоков, т. к. в этом случае несколько потоков разделяют общую память, не блокируя доступ к ней, а память, как уже говорилось, также является системным ресурсом.

**Состояние потока**

Поток описывает динамическое поведение всей программы или какой-либо функции в программе. Для удобства обозначений предположим, что программа является однопоточной. Тогда поток можно рассматривать как пару:

**поток = (процессор, программа).**

Программа может исполняться процессором только в том случае, если она готова к исполнению. То есть все системные ресурсы, которые необходимы для исполнения этой программы, свободны для использования. Для исполнения программы необходимо, чтобы и сам процессор был свободен и готов к исполнению этой программы. Для более формального описания этих ситуаций вводятся понятия "состояние процессора" и "состояние программы". При этом предполагают, что процессор и программа могут находиться в следующих состояниях.

* Состояния процессора:

• процессор не выделен для исполнения программы;

• процессор выделен для исполнения программы.

* Состояния программы:

• программа не готова к исполнению процессором;

• программа готова к исполнению процессором.

Для краткости записи введем для этих состояний следующие названия:

* Состояния процессора:

• "не выделен";

• "выделен".

* Состояния программы:

• "не готова";

• "готова".

Тогда можем определить состояние потока как пару состояний:

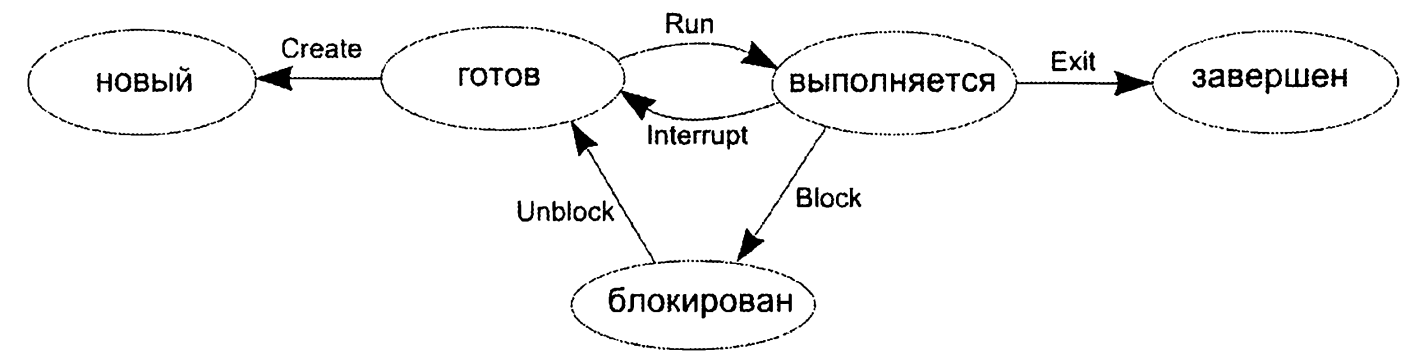
**состояние потока = (состояние процессора, состояние программы).**

Перечислив различные комбинации состояний процессора и программы, можно описать все возможные состояния потока. Введем для состояний потока следующие названия:

* поток блокирован = ("не выделен", "не готова");
* поток готов к выполнению = ("не выделен", "готова");
* поток выполняется = ("выделен", "готова");

Будем считать, что состояние ("выделен", "не готова") является недостижимым для потока. То есть программе, не готовой к исполнению, процессор не выделяется. Более кратко эти состояния потока будем просто обозначать словами: "блокирован", "готов" и "выполняется". Для полноты картины нужно ввести для потоков еще два состояния: "новый" и "завершен", которые описывают соответственно поток, еще не начавший свою работу, и поток, завершивший свою работу. Тогда диаграмма возможных переходов потока из состояния в состояние может быть изображена, как это показано на рис.

Модель пяти состояний потока



В результате мы получили простейшую диаграмму переходов потока из состояния в состояние. Сами переходы потока из состояния в состояние, которые на диаграмме обозначаются дугами, описывают некоторые операции над потоком. Названия этих операций указаны рядом со стрелками. Опишем эти операции:

* Операция **Create** выполняется потоком, который создает новый поток из функции. Эта операция переводит поток из состояния "новый" в состояние "готов".
* Операция **Exit** выполняется самим исполняемым потоком в случае его завершения. Эта операция переводит поток из состояния "выполняется" в состояние "завершен".

Оставшиеся четыре операции выполняются операционной системой:

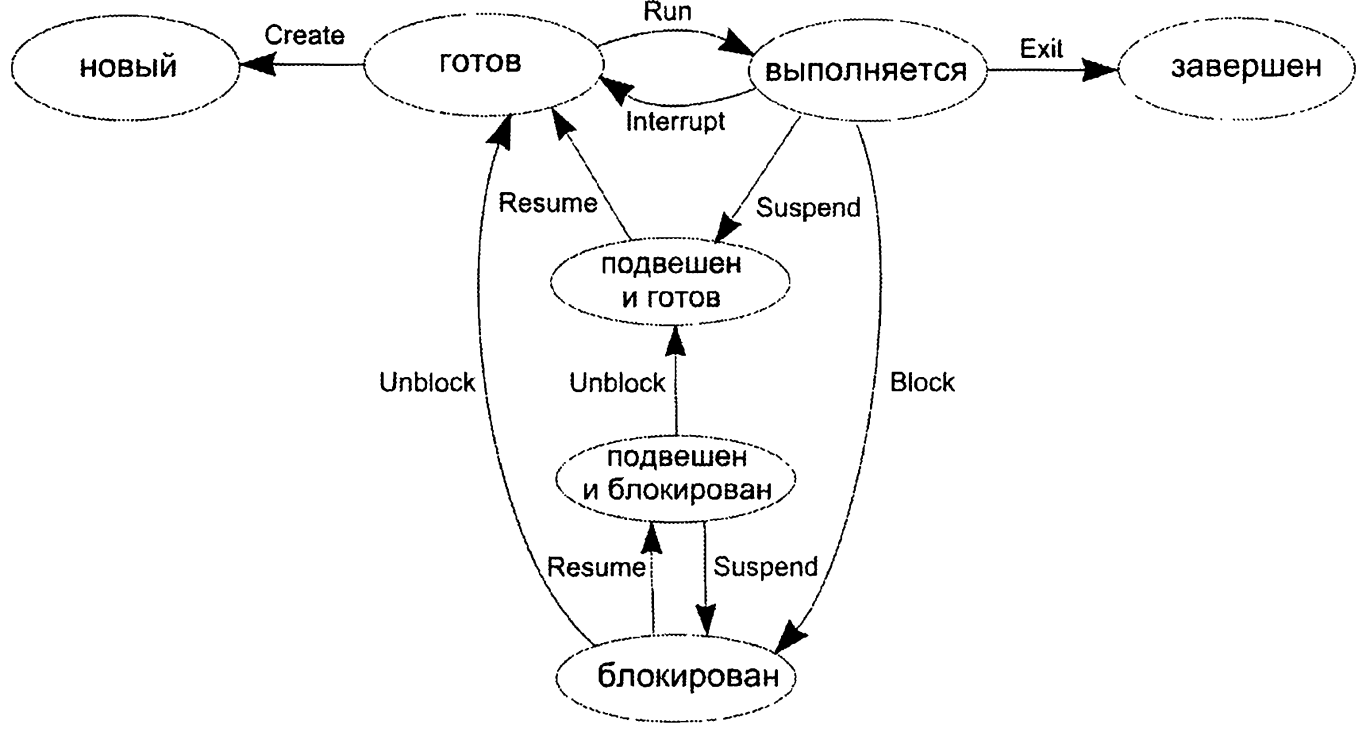
* Операция Run запускает готовый поток на выполнение, т. е. выделяет ему процессорное время. Эта операция переводит поток из состояния "готов" в состояние "выполняется". Поток получает процессорное время в том случае, если подошла его очередь к процессору на обслуживание.
* Операция Interrupt задерживает исполнение потока и переводит его из состояния "выполняется" в состояние "готов". Эта операция выполняется над потоком в том случае, если истекло процессорное время, выделенное потоку на исполнение, или исполнение потока прервано по каким-либо другим причинам.
* Операция Block блокирует исполнение потока, т. е. переводит его из состояния "выполняется" в состояние "блокирован". Эта операция выполняется над потоком в том случае, если он ждет наступления некоторого события, например, завершения операции ввода-вывода или освобождения ресурса.
* Операция Unblock разблокирует поток, т. е. переводит его из состояния "блокирован" в состояние "готов". Эта операция выполняется над потоком в том случае, если событие, ожидаемое потоком, наступило.

Разрешим потокам выполнять операции друг над другом. Введем операции Suspend И Resume.

* Операция suspend приостанавливает исполнение потока.
* Операция Resume возобновляет исполнение потока.

Используя эти операции, один поток может соответственно приостановить или возобновить исполнение другого потока независимо от того, в каком состоянии этот последний поток находится. Впрочем, заметим, что поток может приостановить и свое исполнение. Если над потоком выполнена операция suspend, то будем говорить, что поток находится в приостановленном или подвешенном состоянии (кратко поток "подвешен"). Дополним диаграмму состояний потока этими новыми операциями и состояниями. Получим более полную диаграмму состояний потока.

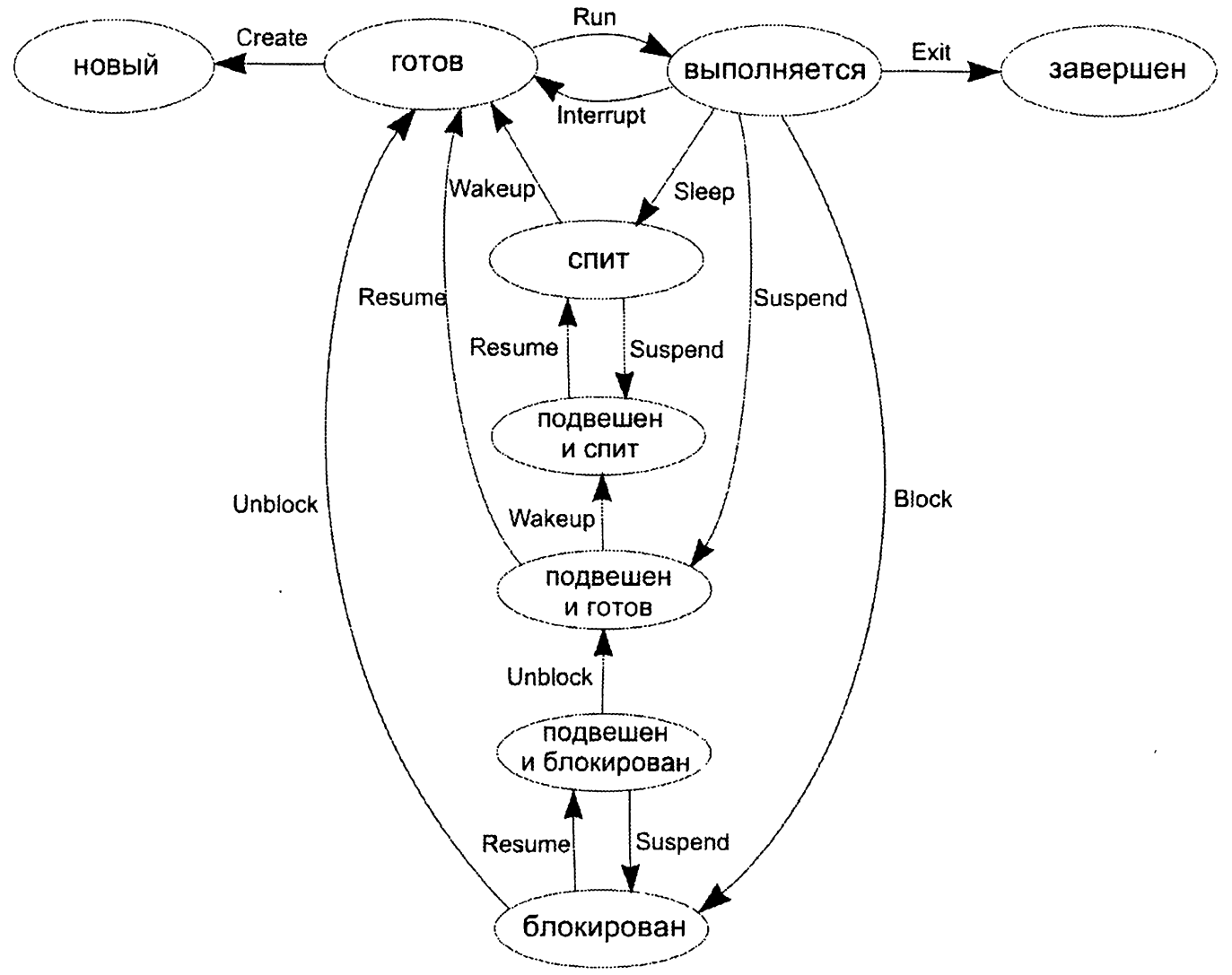
Модель семи состояний потока



Разрешим потоку выполнять операции над самим собой. Для этого введем операцию sleep.

* Операция sleep позволяет потоку приостановить свое исполнение на некоторый интервал времени или, другими словами, заснуть. Разбудить поток должна операционная система по истечении заданного интервала времени, используя операцию wakeup. Если поток выполнил операцию sleep, то будем говорить, что он перешел в сонное состояние или "спит".
* Операция wakeup позволяет операционной системе разбудить поток. В результате можно построить полную диаграмму состояний потока, которая и приведена на рисунке.

Модель девяти состояний потока



**Диспетчеризация и планирование потоков**

В однопрограммной операционной системе одновременно может выполняться только один поток, которому доступны все ресурсы компьютера. Поэтому блокировка потока может происходить только в случаях ожидания этим потоком события, отмечающего завершение операций ввода-вывода. Недостатком однопрограммных операционных систем является их низкая производительность, т. к. процессор простаивает, если поток блокирован.

В мультипрограммных операционных системах одновременно могут существовать несколько потоков, что повышает производительность компьютера. Однако в этом случае требуется некоторая дисциплина обслуживания этих потоков, смысл которой заключается в порядке выделения конкурирующим потокам ресурсов компьютера.

Для простоты дальнейшего изложения будем считать, что компьютер имеет только один процессор. Тогда общий подход к обслуживанию потоков в мультипрограммных операционных системах состоит в следующем. Время работы процессора делится на кванты (интервалы), которые выделяются потокам для работы. По истечении кванта времени исполнение потока прерывается и процессор назначается другому потоку. Распределением квантов времени между потоками занимается специальная программа, которая называется *менеджер потоков*.

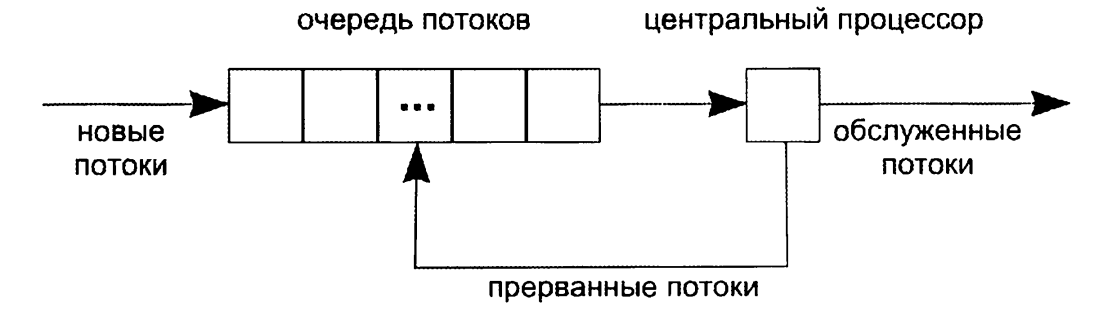
Когда менеджер потоков переключает процессор на исполнение другого потока, он должен выполнить следующие действия:

* сохранить контекст прерываемого потока;
* восстановить контекст запускаемого потока на момент его прерывания;
* передать управление запускаемому потоку.

Контекст потока это содержимое памяти, с которой работает поток. Поэтому в каждый момент времени работы потока, его контекст полностью определяется содержимым регистров микропроцессора в этот момент времени. Отсюда следует, что для сохранения контекста потока необходимо сохранить содержимое регистров микропроцессора на момент прерывания потока, а при восстановлении контекста потока необходимо восстановить содержимое этих регистров.

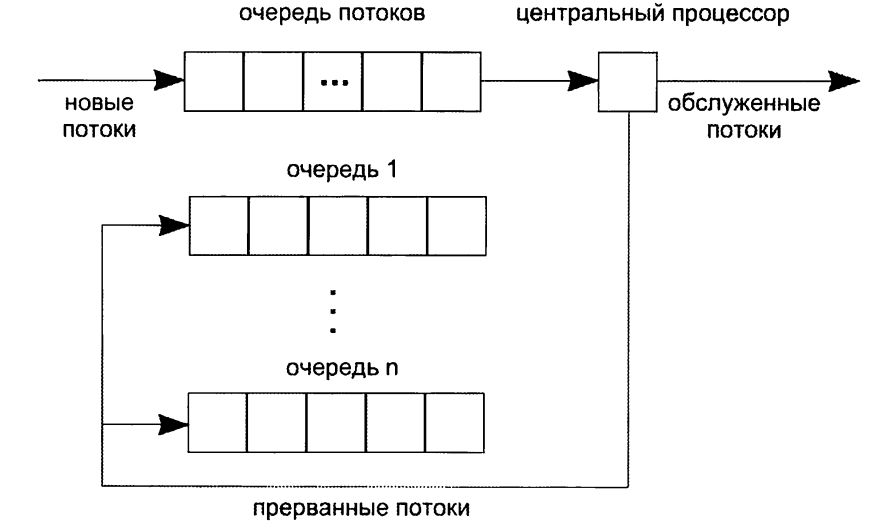
Предположим, что все потоки имеют одинаковый приоритет. Тогда они выстраиваются в одну очередь на обслуживание к процессору. Процессор обслуживает потоки в порядке FIFO (first in — first out), т. е. первым пришел — первым вышел, и прерванные потоки становятся в конец очереди. Такая дисциплина обслуживания называется циклическим обслуживанием. Так как незавершившиеся потоки блокируются до следующего обслуживания, а не уходят не обслуженными, то циклическое обслуживание также называется FCFS (first come — first served), т. е. первым пришел — первым обслужен.

Циклическое обслуживание потоков



Если потоки имеют разные приоритеты, то для управления ими используются более сложные дисциплины обслуживания с несколькими очередями. В этом случае каждая очередь включает потоки, которые имеют одинаковый приоритет.

Дисциплины обслуживания с несколькими очередями



Простейший алгоритм обслуживания нескольких очередей заключается в следующем: **первыми обслуживаются потоки, которые имеют наивысший приоритет.**

В общем случае управление потоками разделяется на планирование и диспетчеризацию. Под планированием потоков понимается алгоритм, используемый для постановки прерванных потоков в очереди. Менеджер потоков (диспетчер) может изменять приоритет прерванного потока, что изменяет очередь, в которую этот поток будет поставлен. Алгоритмы планирования изучаются математической дисциплиной, которая называется *теория расписаний*. Под диспетчеризацией потоков понимается алгоритм, устанавливающий порядок, в котором процессор обслуживает очереди. Алгоритмы диспетчеризации изучаются математической дисциплиной, которая называется *теория массового обслуживания*.

Алгоритмы управления потоками разрабатывают таким образом, чтобы оптимизировать следующие параметры системы:

* время загрузки микропроцессора работой должно быть максимальным;
* пропускная способность системы должна быть максимальной;
* время нахождения потока в системе должно быть минимальным;
* время ожидания потока в очереди должно быть минимальным;
* время реакции системы на обслуживание заявки должно быть минимальным.

При этом для каждой системы должен быть выбран оптимальный интервал обслуживания потоков, который снижает затраты на переключение контекстов потоков. В общем случае разделение времени работы процессора между потоками позволяет быстрее выполнять потоки, которые требуют немного времени на свое исполнение, но замедляет исполнение трудоемких потоков.

**Определение процесса**

Процессом или задачей называется исполняемое на компьютере приложение вместе со всеми ресурсами, которые требуются для его исполнения. Все ресурсы, необходимые для исполнения процесса, также называются контекстом процесса. Процессу обязательно принадлежат следующие ресурсы:

* адресное пространство процесса;
* потоки, исполняемые в контексте процесса.

Адресное пространство — это виртуальная память, выделенная процессу для запуска программ. Адресные пространства разных процессов не пересекаются. Более того, процесс не имеет непосредственного доступа в адресное пространство другого процесса. Это позволяет избежать влияния ошибок, произошедших в каком-либо процессе, на исполнение других процессов, что повышает надежность системы в целом. Потоки, исполняемые в контексте процесса, запускаются в одном адресном пространстве, которое принадлежит этому процессу. Основной причиной, вызвавшей введение в системное программирование понятия потока, и было разделение адресных пространств процессов. В этом случае взаимодействие между параллельными процессами требует больших затрат на пересылку данных, что заметно замедляет работу приложений. Потоки же выполняются в адресном пространстве одного процесса и, следовательно, могут обращаться к общим адресам памяти, что упрощает их взаимодействие.

**Тема 4. Потоки в Windows**

**Определение потока**

Потоком в Windows называется объект ядра, которому операционная система выделяет процессорное время для выполнения приложения. Каждому потоку принадлежат следующие ресурсы:

* код исполняемой функции;
* набор регистров процессора;
* стек для работы приложения;
* стек для работы операционной системы;
* маркер доступа, который содержит информацию для системы безопасности.

Все эти ресурсы образуют контекст потока в Windows. Кроме дескриптора каждый поток в Windows также имеет свой идентификатор, который уникален для потоков, выполняющихся в системе. Идентификаторы потоков используются служебными программами, которые позволяют пользователям системы отслеживать работу потоков.

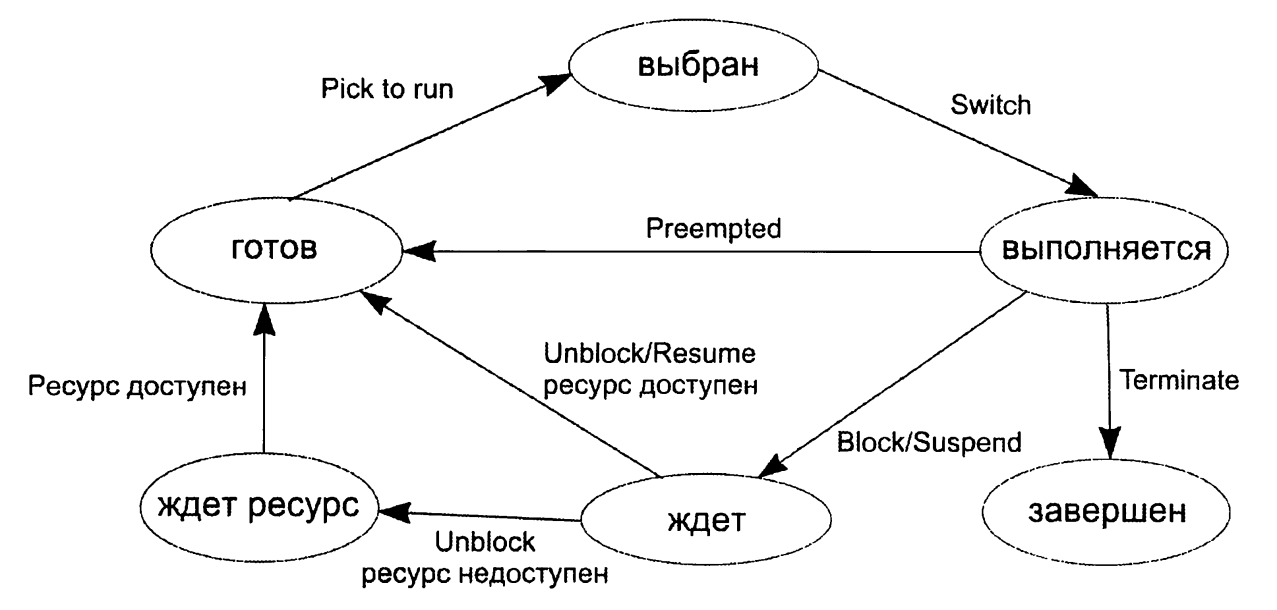
В операционных системах Windows различаются потоки двух типов:

* системные потоки;
* пользовательские потоки.

Системные потоки выполняют различные сервисы операционной системы и запускаются ядром операционной системы.

Пользовательские потоки служат для решения задач пользователя и запускаются приложением.

Диаграмма состояний потока в Windows



В работающем приложении различаются потоки двух типов:

* рабочие потоки (working threads);
* потоки интерфейса пользователя (user interface threads).

Рабочие потоки выполняют различные фоновые задачи в приложении. Потоки интерфейса пользователя связаны с окнами и выполняют обработку сообщений, поступающих этим окнам. Каждое приложение имеет, по крайней мере, один поток, который называется первичным (primary) или главным (main) потоком. В консольных приложениях это поток, который исполняет функцию main. В приложениях с графическим интерфейсом это поток, который исполняет функцию WinMain.

**Создание потоков**

Создается поток функцией CreateThread, которая имеет следующий прототип:

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes, // атрибуты защиты

DWORD dwStackSize, // размер стека потока в байтах

LPTHREAD\_START\_ROUTINE IpStartAddress, // адрес функции

LPVOID lpParameter, // адрес параметра

DWORD dwCreationFlags, // флаги создания потока

LPDWORD lpThreadld // идентификатор потока

);

При успешном завершении функция CreateThread возвращает дескриптор созданного потока и его идентификатор, который является уникальным для всей системы. В противном случае эта функция возвращает значение NULL.

Назначение параметров функции CreateThread:

1. Параметр lpThreadAttributes устанавливает атрибуты защиты создаваемого потока. Пока будем устанавливать значения этого параметра в null при вызове почти всех функций ядра Windows. Это означает, что операционная система сама установит атрибуты защиты потока, используя настройки по умолчанию.
2. Параметр dwstacksize определяет размер стека, который выделяется потоку при запуске. Если этот параметр равен нулю, то потоку выделяется стек, размер которого по умолчанию равен 1 Мбайт. Это наименьший размер стека, который может быть выделен потоку. Если величина параметра dwstacksize меньше значения, заданного по умолчанию, то все равно потоку выделяется стек размером в 1 Мбайт. Операционная система Windows округляет размер стека до одной страницы памяти, который обычно равен 4 Кбайт.
3. Параметр lpstartAddress указывает на исполняемую потоком функцию. Эта функция должна иметь следующий прототип:

DWORD WINAPI имя\_функции\_потока(LPVOID lpParameters);

1. Функции потока может быть передан единственный параметр lpParameter, который является указателем на пустой тип. Это ограничение следует из того, что функция потока вызывается операционной системой, а не прикладной программой. Программы операционной системы являются исполняемыми модулями и поэтому они должны вызывать только функции, сигнатура которых заранее определена. Поэтому для потоков определили самый простой список параметров, который содержит только указатель. Так как функции потоков вызываются операционной системой, то они также получили название функции обратного вызова.
2. Параметр dwCreationFiags определяет, в каком состоянии будет создан поток. Если значение этого параметра равно 0, то функция потока начинает выполняться сразу после создания потока. Если же значение этого параметра равно create\_suspended, то поток создается в подвешенном состоянии. В дальнейшем этот поток можно запустить вызовом функции ResumeThread.
3. Параметр lpThreadid является выходным, т. е. его значение устанавливает Windows. Этот параметр должен указывать на переменную, в которую Windows поместит идентификатор потока. Этот идентификатор уникален для всей системы и может в дальнейшем использоваться для ссылок на поток. Идентификатор потока главным образом используется системными функциями и редко функциями приложения. Действителен идентификатор потока только на время существования потока. После завершения потока тот же идентификатор может быть присвоен другому потоку. Если значение = null, тогда операционная система не возвратит идентификатор потока.

Пример программы, которая использует функцию CreateThread для создания потока и демонстрирует способ передачи параметров исполняемой потоком функции.

#include <conio.h>

#include <windows.h>

#include <iostream.h>

#pragma hdrstop

#pragma argsused

volatile int n;

DWORD WINAPI Add(LPVOID iNum)

{

cout<<"Thread is started."<< endl;

n+= (int)iNum;

cout<<"Thread is finished."<<endl;

return 0;}

int main(int argc, char\* argv[])

{

int inc = 10;

HANDLE hThread;

DWORD IDThread;

cout<<"n = "<<n<<endl;

// запускаем поток Add

hThread = CreateThread (NULL, 0, Add, (void\*) inc, 0, & IDThread) ;

if (hThread == NULL)

return GetLastError();

// ждем, пока поток Add закончит работу-

WaitForSingleObject(hThread, INFINITE);

// закрываем дескриптор потока Add

CloseHandle(hThread);

cout<<"n = "<<n<<endl;

getch();

return 0; }

Программа использует функцию WaitForSingleObject, которая ждет завершения потока Add.

**Консольное приложение**

Меню File выбрать команду New | Other и на вкладке New появившегося диалогового окна New Items щелкнуть на значке Console Wizard. Появится окно Console Wizard. В этом окне можно выбрать язык программирования и указать, будет ли использоваться та или иная библиотека.

Начинается консольное приложение директивой #pragma hdrstop, которая запрещает выполнение предварительной компиляции подключаемых файлов. После этой директивы надо вставить директивы #include, обеспечивающие подключение необходимых библиотек. Директива #pragma argsused отключает предупреждение компилятора о том, что аргументы, указанные в заголовке функции, не используются.

**Завершение потоков**

Поток завершается вызовом функции ExitThread, которая имеет следующий прототип:

VOID ExitThread(

DWORD dwExitCode // код завершения потока

);

Эта функция может вызываться как явно, так и неявно при возврате значения из функции потока. При выполнении этой функции система посылает динамическим библиотекам, которые загружены процессом, сообщение DLL\_THREAD\_DETACH, которое говорит о том, что поток завершает свою работу.

Один поток может завершить другой поток, вызвав функцию TerminateThread, которая имеет следующий прототип:

BOOL TerminateThread(

HANDLE hThread, // дескриптор потока

DWORD dwExitThread // код завершения потока

);

В случае успешного завершения функция TerminateThread возвращает ненулевое значение, в противном случае — FALSE. Функция TerminateThread завершает поток, но не освобождает все ресурсы, принадлежащие этому потоку. Это происходит потому, что при выполнении этой функции система не посылает динамическим библиотекам, загруженным процессом, сообщение о том, что поток завершает свою работу. В результате динамическая библиотека не освобождает ресурсы, которые были захвачены для работы с этим потоком. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании потока.

Ниже приведена программа, которая демонстрирует работу функции TerminateThread. В этой программе следует обратить внимание на квалификатор типа volatile, который указывает компилятору, что значение переменной count должно храниться в памяти, т. к. к этой переменной имеют доступ параллельные потоки. Дело в том, что сам компилятор языка программирования С или C++ не знает, что такое поток. Для него это просто функция. А в языках программирования С и C++ любая функция вызывается только синхронно, т. е. функция, вызвавшая другую функцию, ждет завершения этой функции. Если не использовать квалификатор volatile, то компилятор может оптимизировать код и в одном потоке хранить значение переменной в регистре, а в другом потоке — в оперативной памяти. В результате параллельно работающие потоки будут обращаться к разным переменным.

#include <vcl.h>

#include <windows.h>

#include <iostream.h>

#include <conio.h>

#pragma hdrstop

#pragma argsused

volatile UINT co;

void thread()

{

for (;;)

{

++co;

Sleep(100); // немного отдохнем

}

}

int main()

{

HANDLE hThread;

DWORD IDThread;

char c;

hThread = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE) thread, NULL,

0, &IDThread);

if (hThread == NULL)

return GetLastError();

for (;;)

{

cout << "Input 'y' to display the count or any char to finish: ";

cin>>c;

if (c == 'y')

cout << "count = " << co << endl;

else

break;

}

// прерываем выполнение потока thread

TerminateThread(hThread, 0);

// закрываем дескриптор потока

CloseHandle(hThread);

getch();

return 0;

}

**Приостановка и возобновление потоков**

Каждый созданный поток имеет счетчик приостановок, максимальное значение которого равно maximum\_suspend\_count. Счетчик приостановок показывает, сколько раз исполнение потока было приостановлено. Поток может исполняться только при условии, что значение счетчика приостановок равно нулю. В противном случае поток не исполняется или, как говорят, находится в подвешенном состоянии. Исполнение каждого потока может быть приостановлено вызовом функции suspendThread, которая имеет следующий прототип:

DWORD SuspendThread( HANDLE hThread // дескриптор потока

);

Эта функция увеличивает значение счетчика приостановок на 1 и, при успешном завершении, возвращает текущее значение этого счетчика. В случае неудачи функция SuspendThread возвращает значение, равное -1.

Отметим, что поток может приостановить также и сам себя. Для этого он должен передать функции SuspendThread свой псевдодескриптор, который можно получить при помощи функции GetcurrentThread.

Для возобновления исполнения потока используется функция ResumeThread, которая имеет следующий прототип:

DWORD ResumeThread(

HANDLE hThread // дескриптор потока

);

Функция ResumeThread уменьшает значение счетчика приостановок на 1 при условии, что это значение было больше нуля. Если полученное значение счетчика приостановок равно 0, то исполнение потока возобновляется, в противном случае поток остается в подвешенном состоянии. Если при вызове функции ResumeThread значение счетчика приостановок было равным 0, то это значит, что поток не находится в подвешенном состоянии. В этом случае функция не выполняет никаких действий. При успешном завершении функция ResumeThread возвращает текущее значение счетчика приостановок, в противном случае — значение -1.

Поток может задержать свое исполнение вызовом функции sleep, которая

имеет следующий прототип:

VOID Sleep(

DWORD dwMiHiseconds // миллисекунды

);

Единственный параметр функции Sleep определяет количество миллисекунд, на которые поток, вызвавший эту функцию, приостанавливает свое исполнение. Если значение этого параметра равно 0, то выполнение потока просто прерывается, а затем возобновляется при условии, что нет других потоков, ждущих выделения процессорного времени. Если же значение этого параметра равно INFINITE, то поток приостанавливает свое исполнение навсегда, что приводит к блокированию работы приложения.

Приведем программу, которая демонстрирует работу функций SuspendThread, ResumeThread и Sleep.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

# include <windows.h>

#include <iostream.h>

volatile UINT nCount;

volatile DWORD dwCount;

void thread()

{

for (;;)

{

nCount++;

// приостанавливает поток на 100 миллисекунд

Sleep(100);

}

}

int main ()

{

HANDLE hThread;

DWORD IDThread;

char c;

hThread = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE) thread, NULL,0, &IDThread);

if (hThread == NULL)

return GetLastError();

for (;;)

{

cout << "Input :" << endl;

cout<< "\t'n' to exit" << endl;

cout << "\t'y' to display the count"<< endl;

cout << "\t's' to suspend thread"<< endl;

cout << "\t'r' to resume thread" << endl;

cin >> c;

if (c == 'n')

break;

switch (c)

{

case 'y':

cout << "count = " << nCount << endl;

break;

case 's':

// приостанавливает поток thread

dwCount = SuspendThread(hThread);

cout << "Thread suspend count = " << dwCount << endl;

break;

case 'r':

// возобновляем поток thread

dwCount = ResumeThread(hThread);

cout << "Thread suspend count = " << dwCount << endl;

break;

}

}

// прерываем выполнение потока thread

TerminateThread(hThread, 0);

// закрываем дескриптор потока thread

CloseHandle(hThread);

return 0;

}

**Псевдодескрипторы потоков**

Иногда потоку требуется знать свой дескриптор, чтобы изменить какие-то свои характеристики. Например, поток может изменить свой приоритет. Для этих целей в Win32 API существует функция GetCurrentThread, которая имеет следующий прототип:

HANLDE GetCurrentThread(VOID);

и возвращает псевдодескриптор текущего потока. Псевдодескриптор текущего потока отличается от настоящего дескриптора потока тем, что он может использоваться только самим текущим потоком и, следовательно, может наследоваться другими процессами. Псевдодескриптор потока не нужно закрывать после его использования. Из псевдодескриптора потока можно получить настоящий дескриптор потока, для этого псевдодескриптор нужно продублировать, вызвав функцию DuplicateHandle.

Пример программы, которая вызывает функцию GetCurrentThread, а затем выводит на консоль полученный псевдодескриптор.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

# include <windows.h>

#include <iostream.h>

int main ()

{

HANDLE hThread;

// получаем псевдодескриптор текущего потока

hThread=GetCurrentThread();

// выводим псевдодескриптор на консоль

cout << hThread <<endl;

cin.get();

return 0;

}

**Тема 5. Процессы в Windows**

**Определение процесса**

В Windows под процессом понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением. Поэтому в Windows процессом является исполняемое приложение. Выполнение каждого процесса начинается с первичного потока. Во время своего исполнения процесс может создавать другие потоки. Исполнение процесса заканчивается при завершении работы всех его потоков.

Каждый процесс в операционной системе Windows владеет следующими ресурсами:

* виртуальным адресным пространством;
* рабочим множеством страниц в реальной памяти;
* маркером доступа, содержащим информацию для системы безопасности;
* таблицей для хранения дескрипторов объектов ядра.

Кроме дескриптора, каждый процесс в Windows имеет свой идентификатор, который является уникальным для процессов, выполняющихся в системе. Идентификаторы процессов используются, главным образом, служебными программами, которые позволяют пользователям системы отслеживать работу процессов.

**Создание процессов**

Новый процесс в Windows создается вызовом функции CreateProcess, которая имеет следующий прототип:

BOOL CreateProcess(

LPCTSTR lpApplicationName, // имя исполняемого модуля

LPTSTR lpCoramandLine, // командная строка

LPSECURITY\_ATTRIBUTES lpProcessAttributes, // защита процесса

LPSECURITY\_ATTRIBUTES lpThreadAttributes, // защита потока

BOOL blnheritHandles, // признак наследования дескриптора

DWORD dwCreationFlags, // флаги создания процесса

LPVOID IpEnvironment, // блок новой среды окружения

LPCTSTR lpCurrentDirectory, // текущий каталог

LPSTARTUPINFO IpStartUpInfo, // вид главного окна

LPPROCESS\_INFORMATION lpProcessInformation // информация о процессе

);

Функция CreateProcess возвращает ненулевое значение, если процесс был создан успешно. В противном случае эта функция возвращает значение false. Процесс, который создает новый процесс, называется родительским процессом (parent process) по отношению к создаваемому процессу. Новый же процесс, который создается другим процессом, называется дочерним процессом (child process) по отношению к процессу-родителю.

Первый параметр lpAppiicationName определяет строку с именем исполняемого файла, который имеет тип ехе и будет запускаться при создании нового процесса. Эта строка должна заканчиваться нулем и содержать полный путь к исполняемому файлу.

Рассмотрим программу, которая выводит на консоль свое имя и параметры.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

#include <conio.h>

int main(int argc, char \*argv[])

{

int i;

cputs("I am created.");

cputs("\nMy name is: ");

cputs(argv [0]);

for (i = 1; i <argc; ++i)

cprintf ("\n My %d parameter = %s", i, argv[i]);

cputs("\nPress any key to finish.\n");

getch();

return 0;

}

Скомпилируем эту программу. Полученный ехе-файл сохраним на диске С: и назовем ConsoleProcess.exe. Наша задача состоит в запуске этого файла как нового процесса.

Программа, запускающая созданный ехе-файл как консольный процесс с новой консолью.

#include <vcl.h>

#include <stdio.h>

#include <windows.h>

#include <Winbase.h>

#pragma hdrstop

#pragma argsused

#include <conio.h>

int main()

{

char IpszAppName[] = "С:\\ConsoleProcess.exe";

STARTUPINFO si;

PROCESS\_INFORMATION piApp;

ZeroMemory(&si,sizeof(STARTUPINFO));

si.cb=sizeof(STARTUPINFO);

// создаем новый консольный процесс

if (!CreateProcess(IpszAppName,NULL,NULL,NULL,FALSE,  
CREATE\_NEW\_CONSOLE,NULL,NULL,&si,&piApp))

{

cputs("The new process is not created.\n");

cputs("Check a name of the process.\n");

cputs("Press any key to finish.\n");

getch();

return 0;

}

cputs("The new process is created.\n");

// ждем завершение созданного процесса

WaitForSingleObject(piApp.hProcess, INFINITE);

// закрываем дескрипторы этого процесса в текущем процессе

CloseHandle(piApp.hThread);

CloseHandle(piApp.hProcess);

return 0;

}

Перед запуском консольного процесса ConsoleProcess.exe все поля структуры si типа startupinfo должны заполняться нулями. Это делается при помощи вызова функции ZeroMemory, которая имеет следующий прототип: VOID ZeroMemory ( PVOID Destination, // адрес блока памяти

SIZE\_T Length ); // длина блока памяти

Если мы передаем системе имя нового процесса и его параметры через командную строку, то имя нового процесса может и несодержать полный путь к ехе-файлу, а только имя самого ехе-файла.

В качестве примера запустим приложение Notepad.exe (Блокнот), используя командную строку.

#include <windows.h>

#include <iostream.h>

#include <vcl.h>

#pragma hdrstop

#pragma argsused

int main()

{

STARTUPINFO si;

PROCESS\_INFORMATION pi;

// заполняем значения структуры STARTUPINFO по умолчанию

ZeroMemory(&si, sizeof(STARTUPINFO));

si.cb = sizeof(STARTUPINFO);

// запускаем процесс Notepad

if (!CreateProcess(

NULL, // имя не задаем

"Notepad.exe", // имя программы

NULL, // атрибуты защиты процесса устанавливаем по умолчанию

NULL, // атрибуты защиты первичного потока по умолчанию

FALSE, // дескрипторы текущего процесса не наследуются

0, //по умолчанию NORMAL\_PRIORITY\_CLASS

NULL, // используем среду окружения вызывающего процесса

NULL, // текущий диск и каталог, как и в вызывающем процессе

&si, // вид главного окна - по умолчанию

&pi // информация о новом процессе

)

)

{

cout<< "The mew process is not created." << endl

<<"Check a name of the process." <<endl;

return 0;

}

Sleep(1000); // немного подождем и закончим свою работу

// закроем дескрипторы запущенного процесса в текущем процессе

CloseHandle(pi.hThread) ;

CloseHandle(pi.hProcess);

return 0;

}

**Завершение процессов**

Процесс может завершить свою работу вызовом функции ExitProcess, которая имеет следующий прототип:

VOID ExitProcess(

UINT uExitCode // код возврата из процесса

);

При вызове функции ExitProcess завершаются все потоки процесса с кодом возврата, который является параметром этой функции. При выполнении этой функции система посылает динамическим библиотекам, которые загружены процессом, сообщение DLL\_PROCESS\_DETACH, которое говорит

о том, что динамическую библиотеку необходимо отсоединить от процесса.

Пример программы, которая завершает свою работу вызовом функции ExitProcess.

#include <windows.h>

#include <iostream.h>

#include <vcl.h>

#include <conio.h>

#pragma hdrstop

#pragma argsused

volatile UINT co;

void thread()

{

for (;;)

{

co++;

Sleep(100);

}

}

int main()

{

char c;

HANDLE hThread;

DWORD IDThread;

hThread = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)thread, NULL, 0,&IDThread);

if (hThread == NULL)

return GetLastError();

for (;;)

{

cout<< "Input 'y' to display the count or any char to exit: ";

cin>> (char)c;

if (c == 'y')

cout << "count = " <<co <<endl;

else

ExitProcess(1);

getch();

}

}

**Консольное приложение**

К**онсольное приложение** – это приложение, которое для взаимодействия с пользователем не использует графический интерфейс. Устройством, обеспечивающим взаимодействие с пользователем, является консоль – клавиатура и монитор, работающий в режиме отображения символьной информации (буквы, цифры и специальные знаки).

В ОС консольное приложение работает в окне командной строки.

Универсальными функциями, обеспечивающими вывод и ввод информации в консольных приложениях, являются функции **printf** и **scanf**.

Для того чтобы программа могла их использовать, в начало программы надо включить директиву **#include <stdio.h>**.

В общем виде инструкция вызова функции **printf** выглядит так:

printf(УправляющаяСтрока, СписокПеременных)

Параметр **УправляющаяСтрока** задает способ отображения (формат) значений переменных, имена которых задает параметр список переменных. Также может содержать символы и управляющие последовательности.

Параметр **Списокпеременных** не является обязательным и представляет собой последовательность разделенных запятыми имен переменных, значения которых должны быть выведены.

Спецификатор формата задает вид вывода. Например, значение переменной типа **float** можно вывести как десятичное число с точкой (%f) или как число в формате с плавающей точкой (%е). В спецификаторе формата можно задать размер поля вывода (количество позиций экрана), а для формата f – размер поля для вывода целой и дробной частей числа. Если во время работы программы окажется, что выводимое значение не умещается в поле, указанном в спецификации, то для его вывода будет использовано столько позиций, сколько необходимо.

Спецификаторы формата

| **Спецификатор** | **Тип переменной** | **Форма вывода** |
| --- | --- | --- |
| %nd | int | Десятичное со знаком |
| %n.mf | float или double | Дробное с десятичной точкой |
| %ne | float или double | Дробное в виде числа с плавающей точкой |
| %nc | char | Символ |
| %ns |  | Строка |

При выводе одной инструкцией значений нескольких переменных значение первой переменной выводится в соответствии с первым по порядку спецификатором формата из управляющей строки, второй со вторым и т. д.

|  |  |
| --- | --- |
| **Спецификаторы преобразования для функции printf()** | |
| ***Код*** | ***Формат*** |
| %a | Шестнадцатеричное в виде *0xh.hhhhp+d* (только С99) |
| %A | Шестнадцатеричное в виде *0Xh.hhhhP+d* (только С99) |
| %c | Символ |
| %d | Десятичное целое со знаком |
| %i | Десятичное целое со знаком |
| %e | Экспоненциальное представление ('е' на нижнем регистре) |
| %E | Экспоненциальное представление ('Е' на верхнем регистре) |
| %f | Десятичное с плавающей точкой |
| %g | В зависимости от того, какой вывод будет короче, используется %е или %f |
| %G | В зависимости от того, какой вывод будет короче, используется %Е или %F |
| %o | Восьмеричное без знака |
| %s | Строка символов |
| %u | Десятичное целое без знака |
| %x | Шестнадцатеричное без знака (буквы на нижнем регистре) |
| %X | Шестнадцатеричное без знака (буквы на верхнем регистре) |
| %p | Выводит указатель |
| %n | Аргумент, соответствующий этому спецификатору, должен быть указателем на целочисленную переменную. Спецификатор позволяет сохранить в этой переменной количество записанных символов (записанных до того места, в котором находится код %n) |
| %% | Выводит знак % |

Применение спецификатора преобразования %g показано в следующей программе:

#include <stdio.h>

int main(void)

{

double f;

for(f=1.0; f<1.0e+10; f=f\*10)

printf("%g ", f);

return 0;

}

В результате выполнения получится следующее:

1 10 100 1000 10000 100000 1e+06 1e+07 1e+08 1e+09

Если надо вывести символ, который не может быть помещен в строку вывода обычным образом путем набора на клавиатуре, – например, символ новой строки или двойная кавычка, которая в языке C/C++ используется для ограничения в тексте программы строк, – то вместо этого символа применяется специальная последовательность символов. Специальная (управляющая) последовательность начинается символом обратной наклонной черты

Управляющие последовательности

| **Последовательность** | **Действие** |
| --- | --- |
| **\n** | Переводит курсор в начало следующей строки |
| **\r** | Переводит курсор на следующую строку текущей колонки |
| **\t** | Переводит курсор в следующую позицию табуляции |
| **\** | Выводит двойную кавычку |
| **\\** | Выводит обратную наклонную черту |
| **\0x** ШестнадцатеричноеЧисло | Выводит символ, код которого указан |

Для вывода на экран сообщений часто используют функцию puts, которая, в отличие от printf, после вывода автоматически переводит курсор в начало следующей строки. У функции puts один параметр – сообщение. В простейшем случае в качестве параметра функции puts используется строковая константа. Например, функция:

puts("У лукоморья дуб зеленый,/nЗлатая цепь на дубе том.")

Выводит две строчки стихотворения, каждую на отдельной строке, и переводит курсор в начало следующей строки.

Чтобы вывести цветной текст, надо использовать функции cprintf и cputs. Они ничем не отличаются от рассмотренных ранее printf и puts, за исключением того, что цвет символов, выводимых этими функциями, можно задать, вызвав функцию textcolor, а цвет фона – textbackground.

В общем виде инструкции вызова указанных выше функций выглядят так:

textcolor(Цвет);

textbackground(Цвет);

Параметр цвет – параметр целого типа, в качестве которого обычно используют одну из именованных констант.

**Таблица 7.3**. Константы, в качестве параметра цвет.

| **Цвет** | **Константа** | **Значение константы** |
| --- | --- | --- |
| Черный | BLACK | 0 |
| Синий | BLUE | 1 |
| Зеленый | GREEN | 2 |
| Бирюзовый | CYAN | 3 |
| Красный | RED | 4 |
| Сиреневый | MAGENTA | 5 |
| Коричневый | BROWN | 6 |
| Светло-серый | LIGHTGRAY | 7 |
| Серый | DARKGRAY | 8 |
| Голубой | LIGHTBLUE | 9 |
| Светло-зеленый | LIGHTGREEN | 10 |
| Светло-бирюзовый | LIGHTCYAN | 11 |
| Алый | LIGHTRED | 12 |
| Светло-сиреневый | LIGHTMAGENTA | 13 |
| Желтый | YELLOW | 14 |
| Белый (яркий) | WHITE | 15 |

В качестве параметра функции textcolor можно использовать символьные константы со значением от 0 до 15, а в качестве параметра функции textbackground – только от 0 до 7.

При выводе на экран весьма полезна функция clrscr, которая очищает экран, закрашивая его цветом фона, установленным функцией textbackground.

Функции textcolor, textbackground, clrscr и приведенные выше константы объявлены в файле conio.h, поэтому, чтобы они были доступны, в текст. Необходимо включить директиву #include <conio.h>.

**Функция scanf**

Наиболее универсальной функцией, которая позволяет ввести данные с клавиатуры, является функция **scanf**. В общем виде инструкция вызова функции **scanf** для ввода значения одной переменной выглядит так:

scanf(Формат,& Переменная);

Где:

* **Формат** – это строка, которая содержит спецификатор формата, определяющий то, как должна интерпретироваться строка, введенная с клавиатуры. Наиболее часто используемыми спецификаторами являются: %i – для ввода целых, %f – для ввода дробных, %s – для ввода строк;
* **&переменная** – это адрес переменной, значение которой вводится. Например, инструкция:

scanf("%i", &Skol); ­ Вводит целое число, а инструкция:

scanf("%i%f", &kol, &cena); – Вводит целое и дробное.

При вызове функции **scanf** происходит следующее. Программа приостанавливает работу и ждет, пока пользователь наберет на клавиатуре строку символов и нажмет клавишу **Enter**. После нажатия клавиши **Enter** функция **scanf** преобразует введенную строку в данные и записывает их в переменную, адрес которой указан. Преобразование выполняется в соответствии со спецификатором формата. Например, в результате выполнения инструкции **scanf ("%f", &cena)** и набора на клавиатуре строки 25.99 значение переменной **сеnа** будет равно 25.99.

Следует обратить внимание, что при использовании функции **scanf** наиболее частой ошибкой, причем не обнаруживаемой компилятором, является отсутствие символа **&** перед именем переменной.

Если введенная пользователем строка не соответствует типу ожидаемых данных – например, программа ждет ввода целого числа, а пользователь ввел дробное, – то функция **scanf** обрабатывает только ту часть введенной строки, которая может быть преобразована в требуемые данные. Например, в программе для ввода данных о стоимости покупки используется инструкция **scanf("%i%f", &kol, &cena)**, которая предполагает, что пользователь введет в одной строке сначала количество предметов, а затем цену предмета. Если во время работы программы вместо строки 3 24.99 (три предмета по 24.99) ввести строку 24.99 3, то значение переменной **kol** будет равно 24, а переменной **сеnа** – 99.

Следует обратить внимание на то, что консольное приложение разрабатывается в Windows, а выполняется как программа DOS. В DOS и Windows буквы русского алфавита имеют разные коды (в DOS используется кодировка ASCII, а в Windows – ANSI). Это приводит к тому, что консольное приложение вместо сообщений на русском языке выводит "абракадабру".

Проблему вывода сообщений на русском языке консольными приложениями можно решить, разработав функцию перекодировки ANSI-строки в строку ASCII. Если эту функцию назвать rus, то инструкция вывода сообщения может выглядеть, например, так:

printf(rus("Скорость: %3.2f км/час"), v);

Пример консольного приложения программа "Угадай число", которая для вывода сообщений использует функцию **rus**. Значение функции **rus** – строка символов в кодировке ASCII, соответствующая строке, полученной в качестве параметра.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

#include <stdio.h>

#include <conio.h> // для доступа к getch()

#include <stdlib.h> // для доступа к srandf), rand()

#include <time.h> // для доступа к time\_t и time 0

char\* rus(char\* st);

// преобразует ANSI-строку в строку ASCII

int main(int argc, char\* argv[])

{

int comp, // число, "задуманное" компьютером

igrok, // вариант игрока

n=0; // число попыток

// ГСЧ — генератор случайных чисел

time\_t t; // текущее время (для инициализации ГСЧ)

srand( (unsigned) time (&t) ) ; // инициализация ГСЧ

comp = rand () % 10 + 1;

puts ( rus ( " \nКомпьютер \"задумал\" число от 1 до 10."));

puts ( rus ( "Вы должны его угадать за три попытки."));

do

{

printf ("->");

scanf("%i", &igrok) ;n++;

}

while ( igrok != comp && n < 3) ;

if (igrok == comp)

printf ( rus ( "ВЫ ВЫИГРАЛИ ! " ) ) ;

else {

puts ( rus ( "Вы проиграли . " ) ) ;

printf ( rus ( "Компьютер \"задумал\" число %d") , comp) ;

}

printf ( rus("\nДля завершения нажмите любую клавишу. .."));

getch();

return 0;}

/\* Функция rus преобразует ANSI -строку в строку ASCII

и может использоваться для вывода сообщений

на русском языке в консольных программах.

Пример использования:

printff rus ("Скорость: %3.2f км/час"), v) ;

print f ( rus ("У лукоморья дуб зеленый \ п ") ) ;

\*/

char\* rus (char\* st)

{

unsigned char\* p = st;

/\* при объявлении символов как char русские буквы

кодируются отрицательными числами \*/

while ( \*p)

{

if (\*p >= 192) // здесь русская буква

if (\*p <= 239) // А, Б, ... Я, а, б, ... п

\*p -=64;

else // p ... я

\*p -= 16;

p++;

}

return st;}

Получить доступ к модулю консольного приложения (тексту программы) для того, чтобы внести изменения в программу можно, выбрав в меню **File** команду **Open Project**, нужно открыть файл проекта. Затем надо открыть окно **Project Manager** (команда **View › Project Manager**), раскрыть список файлов, выбрать срр-файл и из контекстного меню выбрать команду **Open** (или сделать двойной щелчок на имени срр-файла).

**Указатели**

Указатель – это адрес поля памяти, занимаемого программным объектом.

Пусть в программе определены три переменные разных типов:

int a=5;

char с='G';

float r=1.2E8;

Эти величины разместились в памяти компьютера следующим образом:



**Операция & — адрес**. Применение этой операции к имени переменной дает в результате ее адрес в памяти. Для переменных из данного выше примера: &а равно FFC0, &с - FFC2, &r - FFC3.

**Описание указателей.** Для хранения адресов используются переменные типа «указатель». Формат описания таких переменных следующий: **тип \*имя\_переменной**

Примеры описания указателей:

int \*pti; char \*ptc; float \*ptf; После такого описания переменная pti может принимать значение указателя на величину целого типа; переменная ptc предназначена для хранения указателя на величину типа char; переменная ptf — на величину типа float.

Указателям могут присваиваться значения адресов объектов только того типа, с которым они описаны. В нашем примере допустимы операторы

pti=&a; ptc=&c; ptf=&r;

В результате указатели примут следующие значения: pti - FFCO, ptc - FFC2, ptf - FFC3.

Как и для других типов данных, значения указателей могут инициализироваться при описании. Например:

int a=5; int \*pti=&a;

char c='G'; char \*ptc=&c;

float r=1.2E8; float \*ptf=&r;

**Использование строк в командной строке**

Предположим, что мы откомпилировали файл **a.cpp** и получили файл **a.exe**.

Строка вида **C:\>a** называется ***командной строкой***, но она может быть записана и так:

C:\>a Ученье свет

Ранее мы не указывали аргументы функции **main()**, в дальнейшем нам будет необходимо указывать два аргумента функции **main()**.

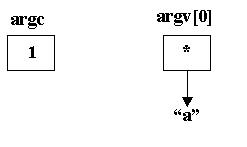
Записывать ее мы будем так:

main (int argc,char \*argv[])

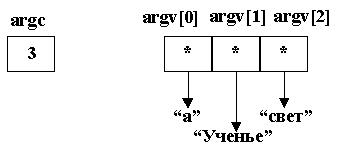
где:

* **argc** - количество параметров в командной строке;
* **argv** - массив указателей на строки.

Для командной строки: C:\>a

  
Рис.1. Значение аргументов функции main()

а для строки: C:\>a Ученье свет

  
Рис.2. Значение аргументов функции main()

Теперь параметры командной строки мы можем использовать в программе для ввода исходной информации.

**Пример.** Для запоминания числа   иногда используют "магические" фразы, например: "Это я знаю и помню прекрасно Пи многие знаки мне лишни напрасны" или "Кто и шутя и скоро пожелаетъ Пи узнать число ужъ знаетъ". Число букв в каждом слове любой из данных фраз представляет собой некоторую цифру числа : "это"-3, "я"-1, "знаю"-4 и т.д. Попробуем вывести на печать число , используя любую из указанных фраз.

    Фраза записывается в строке с именем программы, перед запуском.

#include<iostream.h>

#include<string.h>

void main (int nargs,char \*args[])

{

int n;

if (nargs<2)

cout << "Строка не введена\n";

else

for (int i=1;i<nargs;i++)

{

n=strlen(args[i]);

if (n>9)

cout << "\nОшибка!!! Слово длиннее 9 букв!";

else

if (i==1)

cout << n <<".";

else

cout << n;

} cout << endl;}

    Результат работы программы:

Это я знаю и помню прекрасно Пи многие знаки мне лишни напрасны

3.14159265358

apro ldv orp keng o

4.3341

Кто и шутя и скоро пожелаетъ Пи узнать число ужъ знаетъ

3.1415926536

**Работа с Файлами**

Прежде, чем читать или записывать информацию в файл, его нужно открыть с помощью стандартной библиотечной функции fopen(), которая работает с физическим именем файла и возвращает значение логического имени (указатель на файл), которое и используется системой для чтения и записи в данный файл. Общий вид этой функции следующий:

<указатель на файл> = fopen (<имя файла>,<режим доступа>); .

В этой функции указатель на файл должен быть описан так:

FILE \*fp;

Этот тип данных определяет поток и содержит информацию, необходимую для управления потоком, в том числе указатель на буфер  потока, и его показатели состояния. (stdio.h)

|  |  |
| --- | --- |
| **Режимы доступа к файлам для функции fopen()** | |
| **Строка режима** | **Описание** |
| **"r"** | Открывает файл только для чтения. Модификация файла не разрешена. |
| **"w"** | Создает новый файл только для записи. Перезаписывает любой существующий файл с тем же именем. Чтение информации из файла не разрешено. |
| **"a"** | Открывает файл в режиме "только для записи" с добавлением новой информации в конец файла. Если файл не существует, он создается, а любой существующий файл с таким же именем перезаписывается. Чтение информации из файла не разрешено. |
| **"r+"** | Открывает существующий файл для чтения и записи. |
| **"w+"** | Создает новый файл для чтения и записи. Перезаписывает любой существующий файл с тем же именем. |
| **"a+"** | Открывает файл в режиме чтения и записи для добавления новой информации в конец файла. Если файл не существует, он создается, и любой существующий файл с тем же именем перезаписывается. |

Выше указанные спецификаторы режима доступа к файлу используются только в текстовых файлах. Для того, чтобы открыть двоичный файл, символ b должен быть включен в режим доступа. Этот дополнительный символ b может быть добавлен в конец строки, что даёт следующие режимы доступа к бинарным файлам: rb, wb, ab, r+b, w+b, a+b или может быть вставлен между буквой и знаком +, в случае со смешанными режимами: rb+, wb+, ab+.

При невозможности открыть файл с заданным именем функция fopen() возвращает нулевое (NULL) значение указателя файла, что сразу сигнализирует об ошибке.

    Функция

fclose (<указатель на файл>);

"разрывает" связь между указателем на файл и физическим именем, установленную функцией fopen(), и освобождает указатель для другого файла.

Для чтения информации из уже открытого файла или записи в него существует несколько возможностей. Рассмотрим пока самые простые - функции getc() и putc(). Функция

getc (<указатель на файл>);

возвращает очередной символ из файла с заданным указателем. Если уже достигнут конец файла, то функция возвращает значение EOF. Функция

putc (<переменная>, <указатель на файл>);

помещает значение переменной в файл с заданным указателем.

С началом работы любой программы автоматически открываются три файла:

стандартный вход,

стандартный выход и

стандартный выход для ошибок.

Соответствующие ссылки на файлы называются stdin, stdout и stderr. По умолчанию все они связаны с терминалом, однако stdin и stdout можно связать с файлами и межпрограммными каналами. Ссылки на stdin, stdout и stderr имеются в файле стандартных заголовков stdio.h.

**Пример 1**. Напишите программу очистки содержимого данного файла (имя файла задается в командной строке).

#include <iostream.h>

#include <stdio.h>

void main (int argc,char \*argv[])

{

FILE \*in;

/\* ------ \*/

if (argc < 2)

cout << "Отсутствует имя файла в командной строке.";

else

{

in = fopen (argv[1],"w"); fclose (in);

}

}

**Пример 2**. Заданный файл копировать в новый файл, который имеет расширение **red** и содержит только те символы исходного файла, номера позиций которых в исходном файле кратны трем (0,3,6,9,...).

#include <iostream.h>

#include <stdio.h>

#include <string.h>

void main (int argc,char \*argv[])

{

FILE \*in,\*out;

char ch;

static char name[20];

int count=0;

/\* -------------------- \*/

if (argc < 2)

/\* Проверяем, введено ли имя входного файла. \*/

cout << "Отсутствует имя файла \n";

else

{

if ((in=fopen(argv[1],"r")) != NULL)

{

/\* Копируем имя файла в строку. \*/

strcpy (name,argv[1]);

/\* Проверяем, есть ли расширение. \*/

char \*p = strchr (name,'.');

if (p)

\*p=NULL; //Удалить расширение.

/\* Добавляем .red к имени файла. \*/

out = fopen (strcat (name,".red"),"w");

while ( (ch=getc (in)) != EOF)

/\* Печатаем каждый третий символ файла. \*/

if (count++%3==0)

putc (ch,out);

fclose (in); fclose (out);

}

else cout << "Файл не существует!";

}

}

Теперь если мы поместим эту программу в файл **c70.cpp** и откомпилируем ее, а в файл **edd.txt** поместим ***Даже Эдди нас опередил с детским хором***, то после выполнения командной строки:

C:\>c70.exe edd.txt

получим в файле **edd.red**: ***Дед спел тихо***.

**Файловые системы Windows**

Windows поддерживает на непосредственно подключенных устройствах файловые системы четырех типов:

1. Файловая система NT (NTFS) — современная файловая система, которая поддерживает длинные имена файлов, а также безопасность, устойчивость к сбоям, шифрование, сжатие, расширенные атрибуты, и позволяет работать с очень большими файлами и объемами данных. Заметьте, что на гибких дисках система NTFS использоваться не может; не поддерживается она и системами Windows 9x.(рекомендуется компанией Microsoft для использования в качестве основной)
2. Файловые системы FAT и FAT32 (от File Allocation Table — таблица размещения файлов) происходят от 16-разрядной файловой системы (FAT16), первоначально использовавшейся в MS-DOS и Windows 3.1. FAT32 впервые была введена в Windows 98 для поддержки жестких дисков большого объема и других усовершенствованных возможностей. FAT является единственно доступной файловой системой для дисков (но не компакт-дисков), работающих под управлением Windows 9x, а также гибких дисков. Разновидностью FAT является TFAT — ориентированная на поддержку механизма транзакций версия, используемая в Windows СЕ.
3. Файловая система компакт-дисков (CDFS), предназначена для доступа к информации, записанной на компакт-дисках. CDFS удовлетворяет требованиям стандарта ISO 9660.
4. Универсальный дисковый формат (Universal Disk Format, UDF) поддерживает диски DVD. Поддержка UDF в Windows XP поддерживает как чтение, так и запись файлов, тогда как в Windows 2000 для UDF обеспечивается только запись.

Windows поддерживает такие распределенные файловые системы, как Networked File System (Сетевая файловая система), или NFS, и Common Internet File System (Общая межсетевая файловая система), или CIFS; на серверах обычно используют NTFS.

**Правила именования файлов**

Windows поддерживает обычную иерархическую систему имен файлов, основанную на следующих правилах:

* Полное имя файла на диске, содержащее путь доступа к нему, начинается с указания буквенного имени диска,
* Существует и другой возможный вариант задания полного пути доступа — использование универсальной кодировки имен (Universal Naming Code, UNC), в соответствии с которой указание пути начинается с глобального корневого каталога, обозначаемого двумя символами обратной косой черты (\\), с последующим указанием имени сервера и *имени разделяемого ресурса* (share name) для определения местоположения ресурса на файловом сервере сети. Таким образом, первая часть полного пути доступа в данном случае будет иметь вид: [\\servername\sharename](file:////servername/sharename).
* При указании полного пути доступа в качестве *разделителя* обычно используется символ обратной косой черты (\), но в параметрах API для этой цели можно воспользоваться также символом прямой косой черты (/), как это принято в С.
* В именах каталогов и файлов не должны встречаться символы ASCII, численные значения которых попадают в интервал 1-31, а также любой из перечисленных ниже символов:

< > : " I ? \* \ / В именах разрешается использовать пробелы. В то же время, если имена файлов, содержащие пробелы, указываются в командной строке, то каждое такое имя следует заключать в кавычки, чтобы его нельзя было интерпре­тировать как два разных имени, относящихся к двум отдельным файлам.

* Строчные и прописные буквы в именах каталогов и файлов не различают­ся, то есть имена не чувствительны к регистру (case-insensitive), но в то же время они запоминают регистр (case-retaining); другими словами, если файл был создан с именем MyFile, то это же имя будет использоваться и при его отображении, хотя, например, для доступа к файлу может быть использо­вано также имя myFILE.
* Длина имени каталога и файла не должна превышать 255 символов, а длина полного пути доступа ограничивается значением параметра МАХ\_РАТН (те­кущим значением которого является 256).
* Для отделения имени файла от расширения используется символ точки (.), причем расширения имен (как правило, два или три символа, находящиеся справа от самой последней точки, входящей в имя файла) обозначают предположительные типы файлов в соответствии с определенными согла­шениями. Так, можно ожидать, что файл atou.EXE — это исполняемый файл, а файл atou.С — файл с исходным текстом программы на языке С. Допускается использование в именах файлов нескольких символов точки.
* Одиночный символ точки (.) и два символа точки (. .), используемые в качестве имен каталогов, обозначают, соответственно, текущий каталог и его родительский каталог.

**Создание и открытие файла**

Общий вид функции CreateFile:

HANDLE CreateFile (

LPCTSTR lpName,

DWORD dwAccess

DWORD dwShareMode,

LPSECURITY ATTRIBUTES IpSecurityAttributes,

DWORD dwCreate,

DWORD dwAttrsAndFlags,

HANDLE hTemplateFile)

Возвращаемое значение: в случае успешного выполнения – дескриптор открытого файла (типа HANDLE),иначе —INVALID\_HANDLE\_VALUE.

Параметры:

Имена параметров иллюстрируют некоторые соглашения Windows. Префикс dw используется в именах параметров типа DWORD (32-битовые целые без знака), в которых могут храниться флаги или числовые значения, например счетчики, тогда как префикс lpsz (длинный указатель на строку, завершающуюся нулем), или в упрощенной форме — lр, используется для строк, содержащих пути доступа, либо иных строковых значений.

**lpName** — указатель на строку с завершающим нулевым символом, содержащую имя файла, канала или любого другого именованного объекта, который необходимо открыть или создать. Допустимое количество символов при указании путей доступа обычно ограничивается значением МАХ\_РАТН (260), однако в Windows NT это ограничение можно обойти, поместив перед именем префикс \\?\, что обеспечивает возможность использования очень длинных имен. Сам префикс в имя не входит. Тип данных LPCTSTR относится к строковым данным.

**dwAccess** — определяет тип доступа к файлу — чтение или запись, что соответственно указывается флагами GENERIC\_READ и GENERIC\_WRITE. Ввиду отсутствия флаговых значений READ и WRITE использование префикса GENERIC\_ может показаться излишним, однако он необходим для совместимости с именами макросов, определенных в заголовочном файле Windows WINNT.H.

Указанные значения можно объединять операцией поразрядного "или" (I), и тогда для получения доступа к файлу как по чтению, так и по записи, следует воспользоваться таким выражением:

GENERIC\_READ | GENERIC\_WRITE

**dwShareMode** — может объединять с помощью операции поразрядного "или" следующие значения:

* 0 — запрещает разделение (совместное использование) файла. Более того, открытие второго дескриптора для данного файла запрещено даже в рамках одного и того же вызывающего процесса.
* FILE\_SHARE\_READ — другим процессам, включая и тот, который осуществил данный вызов функции, разрешается открывать этот файл для параллельного доступа по чтению.
* FILE\_SHARE\_WRITE — разрешает параллельную запись в файл.

Используя блокирование файла или иные механизмы, программист должен самостоятельно позаботиться об обработке ситуаций, в которых осуществляются одновременно несколько попыток записи в одно и то же место в файле.

**lpSecurityAttributes** — указывает на структуру SECURITY\_ATTRIBUTES. На первых порах при вызовах функции CreateFile и всех остальных функций вам будет достаточно использовать значение NULL.

**dwCreate** — конкретизирует запрашиваемую операцию: создать новый файл, перезаписать существующий файл и тому подобное. Может принимать одно из приведенных ниже значений, которые могут объединяться при помощи операции поразрядного "или" языка С.

* CREATE\_NEW — создать новый файл; если указанный файл уже существует, выполнение функции завершается неудачей.
* CREATE\_ALWAYS — создать новый файл; если указанный файл уже существует, функция перезапишет его.
* OPENE.EXISTING — открыть файл; если указанный файл не существует, вы­полнение функции завершается неудачей.
* OPEN\_ALWAYS — открыть файл; если указанный файл не существует, функция создаст его.
* TRUNCATE\_EXISTING — открыть файл; размер файла будет установлен равным нулю. Уровень доступа к файлу, установленный параметром dwAccess, должен быть не ниже GENERIC\_WRITE. Если указанный файл существует, его содержимое будет уничтожено. В отличие от случая CREATE NEW выполне­ние функции будет успешным даже в тех случаях, когда указанный файл не существует.

**dwAttrsAndFlags** — позволяет указать атрибуты файла и флаги. Всего имеется 16 флагов и атрибутов. Атрибуты являются характеристиками файла, а не открытого дескриптора, и игнорируются, если открывается существующий файл. Наиболее важных флаговые значения:

* FILE\_ATTRIBUTE\_NORMAL — этот атрибут можно использовать лишь при ус­ловии, что одновременно с ним не устанавливаются никакие другие атри­буты (тогда как для всех остальных флагов одновременная установка до­пускается).
* FILE\_ATTRIBUTE\_READONLY — этот атрибут запрещает приложениям осуще­ствлять запись в данный файл или удалять его.
* FILE\_FLAG\_DELETE\_ON\_CLOSE — этот флаг полезно применять в случае вре­менных файлов. Файл будет удален сразу же после закрытия последнего из его открытых дескрипторов.
* FILE\_FLAG\_OVERLAPPED — этот флаг играет важную роль при выполнении операций асинхронного ввода/вывода.

• FILE\_FLAG\_WRITE\_THROUGH — устанавливает режим сквозной записи промежуточных данных непосредственно в файл на диске, минуя кэш.

• FILE\_FLAG\_NO\_BUFFERING — устанавливает режим отсутствия промежуточной буферизации или кэширования, при котором обмен данными происходит непосредственно с буферами данных программы, указанными при вызове функций ReadFile или WriteFile. Соответственно требуется, чтобы начала программных буферов совпадали с границами секторов, а их размеры были кратными размеру сектора тома. Чтобы определить размер сектора при указании этого флага, вы можете воспользоваться функцией GetDiskFreeSpace.

* FILE\_FLAG\_RANDOM\_ACCESS — предполагается открытие файла для произвольного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа.
* FILE\_FLAG\_SEQUENTIAL\_SCAN — предполагается открытие файла для последовательного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа. Оба последних режима реализуются системой лишь по мере возможностей.

**hTemplateFile** — дескриптор с правами доступа GENERIC\_READ к шаблону файла, предоставляющему расширенные атрибуты, которые будут применены к создаваемому файлу вместо атрибутов, указанных в параметре dwAttrsAndFlags. Обычно значение этого параметра устанавливается равным NULL. При открытии существующего файла параметр hTemplateFile игнорируется. Этот параметр используется в тех случаях, когда требуется, чтобы атрибуты вновь создаваемого файла совпадали с атрибутами уже существующего файла.

**Закрытие файла**

Для закрытия объектов любого типа, объявления недействительными их дескрипторов и освобождения системных ресурсов почти во всех случаях используется одна и та же универсальная функция. Закрытие дескриптора сопровождается уменьшением на единицу счетчика ссылок на объект, что делает возможным удаление таких не хранимых постоянно (nonpersistent) объектов, как временные файлы или события. При выходе из программы система автоматически закрывает все открытые дескрипторы, однако лучше, чтобы программа самостоятельно закрывала свои дескрипторы перед тем, как завершить работу.

Попытки закрытия недействительных дескрипторов или повторного закрытия одного и того же дескриптора приводят к исключениям.

BOOL CloseHandle (HANDLE hObject)

Возвращаемое значение: в случае успешного выполнения функции – TRUE, иначе – FALSE.

Функции UNIX, сопоставимые с рассмотренными выше, отличаются от них в нескольких отношениях. Функция (системный вызов) UNIX **open** возвращает целочисленный дескриптор (descriptor) файла, а не дескриптор типа HANDLE, причем для указания всех параметров доступа, разделения и создания файлов, а также атрибутов и флагов используется единственный целочисленный параметр oflag. Возможные варианты выбора, доступные в обеих системах, перекрываются, однако набор опций, предлагаемый Windows, отличается большим разнообразием.

В UNIX отсутствует параметр, эквивалентный параметру dwShareMode. Файлы UNIX всегда являются разделяемыми.

В обеих системах при создании файла используется информация, касающаяся его защиты. В UNIX для задания хорошо известных разрешений на доступ к файлу для владельца, членов группы и прочих пользователей используется аргумент mode.

Функция **close** отличается от CloseHandle меньшей универсальностью.

Функции библиотеки С, описанные в заголовочном файле <stdio.h>, используют объекты FILE, которые можно поставить в соответствие дескрипторам, связанным с потоками. Имена большинства функций стандартной библиотеки С, предназначенных для работы с объектами FILE, снабжены префиксом "f".

**Чтение файла**

BOOL ReadFile (

HANDLE hFile,

LPVOID lpBuffer,

DWORD nNumberOfBytesToRead,

LPDWORD lpNumberOfBytesRead,

LPOVERLAPPED lpOverlapped)

**Возвращаемое значение:** в случае успешного выполнения (которое считается таковым, даже если не был считан ни один байт из-за попытки чтения с выходом за пределы файла) – TRUE, иначе – FALSE.

Будем предполагать, что дескрипторы файлов создаются без указания флага перекрывающегося ввода/вывода FILE\_FLAG\_OVERLAPPED в параметре dwAttrsAndFlags. В этом случае функция ReadFile начинает чтение с текущей позиции указателя файла, и указатель файла сдвигается на число считанных байтов.

Если значения дескриптора файла или иных параметров, используемых при вызове функции, оказались недействительными, возникает ошибка, и функция возвращает значение FALSE. Попытка выполнения чтения в ситуациях, когда указатель файла позиционирован в конце файла, не приводит к ошибке; вместо этого количество считанных байтов (\*lpNumberOfBytesRead) устанавливается равным 0.

***Параметр:***

**hFile** — дескриптор считываемого файла, который должен быть создан с пра­вами доступа GENERIC\_READ.

**lpBuffer** является указателем на буфер в памяти, куда помещаются считываемые данные.

**nNumberOfBytesToRead** — количество байт, которые должны быть считаны из файла.

**lpNumberOfBytesRead** — указатель на переменную, предназначенную для хранения числа байт, которые были фактически считаны в результате вызова функции ReadFile. Этот параметр может принимать нулевое значение, если перед выполнением чтения указатель файла был позиционирован в конце файла или если во время чтения возникли ошибки, а также после чтения из именованного канала, работающего в режиме обмена сообщениями, если переданное сообщение имеет нулевую длину.

**lpOverlapped** — указатель на структуру OVERLAPPED. На данном этапе просто устанавливайте значение этого параметра равным NULL.

**Запись в файл**

BOOL WriteFile (

HANDLE hFile,

LPCVOID lpBuffer,

DWORD nNumberOfBytesToWrite,

LPDWORD lpNumberOfBytesWritten,

LPOVERLAPPED lpOverlapped)

**Возвращаемое значение:** в случае успешного выполнения - - TRUE, иначе-FALSE.

Заметим, что успешное выполнение записи еще не говорит о том, что данные действительно оказались записанными на диск, если только при создании файла с помощью функции СгеateFile не был использован флаг FILE\_FLAG\_WRITE\_THROUGH. Если во время вызова функции указатель файла был позиционирован в конце файла, Windows увеличит длину существующего файла.

Функции ReadFileGather и WriteFileGather позволяют выполнять операции чтения и записи с использованием набора буферов различного размера.

Сопоставимыми функциями UNIX являются функции read и write, которым программист в качестве параметров должен предоставлять дескриптор файла, буфер и счетчик байтов. Возвращаемые значения этих функций указывают на количество фактически переданных байтов. Возврат функцией read значения 0 означает чтение конца файла, а значения -1 — возникновение ошибки. В противоположность этому в Windows для подсчета количества переданных байтов используется отдельный счетчик, а на успех или неудачу выполнения функции указывает возвращаемое ею булевское значение.

В обеих системах функции имеют сходное назначение и могут выполнять соответствующие операции с использованием файлов, терминалов, ленточных устройств, каналов и так далее.

Входящие в состав стандартной библиотеки С функции read и fwrite, выполняющие операции ввода/вывода в двоичном режиме, вместо счетчика одиночных байтов, как в UNIX и Windows, используют размер объекта и счетчик объектов. Преждевременное прекращение передачи данных может быть вызвано как достижением конца файла, так и возникновением ошибки; точная причина устанавливается с использованием функций feof или ferror. Библиотека предоставляет полный набор функций, ориентированных на работу с текстовыми файлами, таких как fgets или fputs, для которых в каждой из рассматриваемых ОС аналоги вне библиотеки С отсутствуют.

**Пример** создания файла и запись в него данных:

#include <vcl.h>

#pragma hdrstop

#pragma argsused

#include <windows.h>

#include <iostream.h>

int main ()

{

HANDLE hFile;

// создаем файл для записи данных

hFile = CreateFile(

"D:\\demo\_file.txt", // имя файла

GENERIC\_WRITE, // запись в файл

0, // монопольный доступ к файлу

NULL, // защиты нет

CREATE\_NEW, // создаем новый файл

FILE\_ATTRIBUTE\_NORMAL, // обычный файл

NULL // шаблона нет

);

// проверяем на успешное создание

if (hFile == INVALID\_HANDLE\_VALUE)

{

cerr<< "Create file failed." << endl

<<"The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

// пишем данные в файл

char a='k';

for (int i = 1; i <= 10; i++)

{

DWORD dwBytesWrite;

if (!WriteFile(

hFile, // дескриптор файла

&a, // адрес буфера, откуда идет запись

sizeof(a), // количество записываемых байтов

&dwBytesWrite, // количество записанных байтов

(LPOVERLAPPED)NULL // запись синхронная

))

{

cerr<< "Write file failed.." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hFile);

cout <<"Press any key to finish.";

cin.get() ;

return 0;

}

}

// закрьгоаем дескриптор файла

CloseHandle(hFile);

cout<< "The file is created and written." << endl;

return 0; }

**Удаление файлов**

Для физического удаления файла с диска используется функция DeleteFile, которая имеет следующий прототип:

BOOL DeleteFile(

LPCTSTR lpFileName // имя файла

);

Единственный параметр lpFileName является указателем на строку, которая указывает полный путь к файлу. При успешном завершении функция возвращает ненулевое значение, а при неудаче — false.

Программа, которая удаляет файл с именем demo\_file.txt, который расположен в корневом каталоге на диске С:

#include <windows.h>

#include <iostream.h>

int main()

{

if(!DeleteFile("C:\\demo\_file.txt"))

{

cerr << "Delete file failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout <<"Press any key to finish.";

cin.get();

return 0;

}

cout << "The file is deleted." << endl;

return 0;

}

**Копирование файла**

Для копирования файлов используется функция CopyFile, которая имеет следующий прототип:

BOOL CopyFile(

LPCTSTR IpExistingFileName, // имя существующего файла

LPCTSTR lpNewFileName, // имя нового файла

BOOL bFallifExists // действия в случае существования файла

);

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение false. При этом отметим, что функция CopyFile копирует для нового файла также и атрибуты доступа старого файла, но атрибуты безопасности не копируются.

Параметры этой функции имеют следующее назначение:

* Параметр pExistingFileName должен указывать на строку, содержащую имя копируемого файла.
* Параметр lpNewFileName должен указывать на строку с именем файла, в который будет копироваться существующий файл. При этом отметим, что новый файл создается самой функцией CopyFile.
* Параметр bFallifExists определяет действия, которые нужно осуществить в случае, если файл, в который выполняется копирование, уже существует.

Если значение этого параметра равно false, тo функция перезаписывает существующий файл. Если же значение этого параметра равно true, тo выполнение функции в этом случае заканчивается неудачей.

**Пример** программы, которая выполняет копирование файла, используя функцию CopyFile.

#pragma hdrstop

#pragma argsused

# include <windows.h>

#include <iostream.h>

int main()

{

if(!CopyFile("D:\\demo\_file.txt","D:\\new\_file.txt", FALSE))

{

cerr << "Copy file failed." <<endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get() ;

return 0;

}

cout << "The file is copied." << endl;

return 0;}

**Перемещение файла**

Функция MoveFile, служит для перемещения файлов. Перемещение файла отличается от копирования файла только тем, что старый файл после его перемещения удаляется.

Функция MoveFile имеет следующий прототип:

BOOL MoveFile(

LPCTSTR IpExistingFiieName, // имя существующего файла

LPCTSTR lpNewFileName // имя нового файла

);

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение false. Отметим, что функция MoveFile сохраняет все атрибуты перемещаемого файла. Параметры этой функции имеют следующее назначение:

* Параметр IpExistingFiieName должен указывать на строку, содержащую имя перемещаемого файла.
* Параметр lpNewFileName должен указывать на строку с именем файла, в который будет перемещаться существующий файл. При этом отметим, что новый файл Создается самой функцией MoveFile.

**Пример** программы, которая выполняет перемещение файла, используя функцию MoveFile.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

# include <windows.h>

#include <iostream.h>

int main ()

{

if(!MoveFile("D:\\demo\_file.txt", "D:\\new\_file.txt"))

{

cerr <<"Move file failed." <<endl

<<"The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

cout << "The file is moved." << endl;

return 0;

}

**Замещение файла**

В Windows XP определена функция ReplaceFile, которая предназначена для замещения файлов. Существенное отличие этой функции от функций CopyFile и MoveFile состоит в том, что она копирует в замещаемый файл не только атрибуты доступа, но также и атрибуты безопасности файла-заместителя. Отметим также, что функция ReplaceFile работает только с файлами, которые находятся на одном томе (логическом диске).

Функция ReplaceFile имеет следующий прототип:

BOOL RepliaceFile(

LPCTSTR lpReplacedFileName, // имя замещаемого файла

LPCTSTR lpReplacementFileName, // имя файла-заместителя

LPCTSTR lpBackupFileName, // имя резервной копии файла-заместителя

DWORD dwReplасеFlags, // опции замещения

LPVOID IpExclude, // не используется

LPVOID lpReserved //не используется

);

Функция замещения файлов в случае успешного завершения возвращает ненулевое значение, а в случае неудачи — false. Назначение параметров этой функции:

* Параметр lpReplacedFileName должен указывать на строку, которая содержит имя замещаемого файла. Функция ReplaceFile может как создавать новый файл, так и использовать уже существующий.
* Параметр lpReplacementFileName должен указывать на строку, которая содержит имя файла-заместителя. Этот файл заменит файл, на который указывает параметр lpReplacedFileName. При этом отметим, что сам файл-заместитель удаляется с диска.
* В параметр lpBackupFileName устанавливается адрес строки, которая содержит имя файла, который содержит резервную копию файла-заместителя. Если этот параметр установлен в null, тo резервная копия файла-заместителя не создается.
* В параметре dwRepiaceFiags устанавливаются флаги, которые указывают режимы замещения файла. Можно установить любую комбинацию следующих флагов:

□ replacefile\_write\_through — освободить буферы перед выходом из функции;

□ replacefile\_ignore\_merge\_errors — игнорировать ошибки при копировании данных.

* Параметры IpExclude И IpReserved нe используются. поэтому они должны быть установлены в NULL.

**Пример** программы, в которой использова функция ReplaceFile для замещения файла.

#include <windows.h>

#include <iostream.h>

int main()

{

// перемещаем файл

if(!ReplaceFile(

"D: \\new.txt", // имя замещаемого файла

"D:\\demo.txt", // имя файла-заместителя

"D:\\back.txt", // имя резервного файла

REPLACEFILE\_WRITE\_THROUGH, // освободить буферы

NULL, NULL //не используются

))

{

cerr << "Replace file failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get() ;

return 0;}

cout << "The file is replaced." << endl;

return 0;

}

**Работа с указателем позиции файла**

Указатель позиции файла состоит из двух значений типа DWORD, которые будут называться старшая и младшая часть указателя позиции файла соответственно. Следовательно, указатель позиции имеет длину в 64 бита. Если длина файла не превышает двух гигабайт без двух байт, т. е. 231-2 байта, то в указателе позиции используется только его младшая часть, которая рассматривается как целое число со знаком, но при этом старшая часть указателя позиции должна быть установлена в NULL.

Для работы с указателем позиции файла служит функция SetFilePointer, которая имеет следующий прототип:

DWORD SetFilePointer(

HANDLE hFile, // дескриптор файла

LONG lDistanceToMove, // младшая часть сдвига указателя в байтах

PLONG lpDistanceToMoveHigh, // указатель на старшую часть сдвига указателя в байтах

DWORD dwMoveMethod // начальная точка сдвига

);

В случае удачного завершения эта функция возвращает младшую часть новой позиции указателя файла, а по адресу, заданному параметром lpDistanceToMoveHigh, записывает старшую часть новой позиции указателя файла. Если функция устанавливает старшую часть указателя позиции в NULL, то младшая часть указателя позиции представлена положительным целым числом. В случае неудачного завершения функция SetFilePointer возвращает значение -1 и при этом устанавливает значение параметра lpDistanceToMoveHigh в NULL. Если же значение этого параметра не установлено в NULL, тo возвращаемое значение -1 может быть и действительной младшей частью указателя позиции. В этом случае нужно проверить код последней ошибки, который возвращает функция GetLastError. если этот код равен NO\_ERROR, то ошибки нет, а в противном случае выполнение функции SetFilePointer завершилось неудачей. Назначение параметров функции SetFilePointer.

* Параметр hFile должен содержать дескриптор файла, причем сам файл должен быть открыт в режиме чтения или записи
* Параметр lDistanceToMove должен содержать младшую часть сдвига для указателя позиции файла. ЕСЛИ значение параметра IpDistanceToMoveHigh установлено в NULL, тo значение этого параметра рассматривается как целое число со знаком. В случае положительного числа функция выполняет сдвиг указателя вперед на заданное количество байт, а в случае отрицательного числа выполняется сдвиг назад.
* Параметр lpDistanceToMoveHigh должен содержать адрес старшей части сдвига для указателя позиции файла. Старшая и младшая части указателя позиции рассматриваются как целое число со знаком. Если значение этого параметра равно NULL, то сдвиг задается только младшей частью.
* Параметр dwMoveMethod задает начальную точку, от которой выполняется сдвиг указателя позиции. Этот параметр может принимать только одно из следующих значений:

□ file\_begin — сдвиг от начала файла;

□ file\_current — сдвиг от текущей позиции файла;

□ file\_end — сдвиг от конца файла.

Пример программы, которая читает запись файла, предварительно установив на эту запись указатель. Такое чтение записей файла называется прямым доступом к файлу. В общем случае прямой доступ к файлу подразумевает чтение записи с заданным значением ключа, который определяется содержимым одного или нескольких полей записи. Правда, в этом случае нужно знать зависимость указателя позиции файла от значения ключа записи.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

#include <windows.h>

#include <iostream.h>

#include <conio.h>

int main()

{

HANDLE hFile; // дескриптор файла

long n; // для номера записи

long p; // для указателя позиции

DWORD dwBytesRead; // количество прочитанных байт

int m; // прочитанное число

// открываем файл для чтения

hFile = CreateFile(

"D:\\demo.dat", // имя файла

GENERIC\_READ, // чтение из файла

0, // монопольный доступ к файлу

NULL, // защиты нет

OPEN\_EXISTING, // открываем существующий файл

FILE\_ATTRIBUTE\_NORMAL, // обычный файл

NULL // шаблона нет

);

// проверяем на успешное открытие

if (hFile == INVALID\_HANDLE\_VALUE)

{

cerr << "Create file failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

// вводим номер нужной записи

cout << "Input a number from 0 to 9: ";

cin >> n;

// сдвигаем указатель позиции файла

p = SetFilePointer(hFile, n \* sizeof(int), NULL, FILE\_BEGIN);

if(p == -1)

{

cerr << "Set file pointer failed." <<endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hFile);

cout << "Press any key to finish.";

cin.get();

return 0;

}

// выводим на консоль значение указателя позиции файла

cout << "File pointer: " << p <<endl;

// читаем данные из файла

if (!ReadFile(

hFile, // дескриптор файла

&m, // адрес буфера, куда читаем данные

sizeof(m), //количество читаемых байтов

&dwBytesRead, // количество прочитанных байтов

(LPOVERLAPPED)NULL // чтение синхронное

))

{

cerr << "Read file failed." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hFile);

cout << "Press any key to finish.";

cin.get();

return 0;

}

// выводим прочитанное число на консоль

cout << "The read number: " << m << endl;

// закрываем дескриптор файла

CloseHandle(hFile);

getch();

return 0;

}

**Определение размеров файла**

Размер файла является целым числом и для его хранения требуются два значения типа DWORD или, другими словами, два двойных слова. Первое из этих двойных слов содержит старшую часть размера файла, а второе — младшую. Если размер файла входит только в младшую часть, то значение старшей устанавливается в NULL.

Определить размер файла можно при помощи функции GetFileSize, которая имеет следующий прототип:

DWORD GetFileSize(

HANDLE hFile, // дескриптор файла

LPDWORD lpFileSizeHigh // указатель на старшую часть размера файла

);

В случае успешного завершения эта функция возвращает младшую часть размера файла, а по адресу, указанному в параметре lpFileSizeHigh, записывает старшую часть размера файла. В случае неудачи функция GetFileSize возвращает значение -1, если значение адреса, заданного параметром lpFileSizeHigh, установлено в NULL. Если же значение этого адреса не равно NULL и функция закончилась неудачей, то она возвращает значение -1 и функция кода последней ошибки GetLastError возвратит значение, отличное от no\_error.

Отметим, что для правильной работы функции GetFilesize необходимо,

чтобы файл был открыт в режиме чтения или записи.

Пример программы, которая определяет размер файла:

#include <vcl.h>

#pragma hdrstop

#pragma argsused

#include <windows.h>

#include <iostream.h>

#include <conio.h>

int main()

{

HANDLE hFile;

DWORD dwFileSize;// младшая часть размера файла

// открываем файл для чтения

hFile = CreateFile(

"D:\\2.doc",// имя файла

0,// получение информации о файле

0,// монопольный доступ к файлу

NULL, // зашиты нет

OPEN\_EXISTING,// открываем существующий файл

FILE\_ATTRIBUTE\_NORMAL, // обычный файл

NULL // шаблона нет

);

// проверяем на успешное открытие

if (hFile == INVALID\_HANDLE\_VALUE)

{

cerr << "Create file failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

// определяем размер файла

dwFileSize = GetFileSize(hFile, NULL);

if (dwFileSize == -1)

{

cerr << "Get file size failed." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hFile);

cout << "Press any key to finish.";

cin.get();

return 0;

}

// выводим размер файла

cout << "File size: " << dwFileSize << " bait"<<endl;

// закрываем дескриптор файла

CloseHandle(hFile);

getch();

return 0;

}

**Получение информации о файле**

Чтобы получить информацию о файле, можно использовать функцию GetFileInformationByHandle, которая имеет следующий прототип:

BOOL GetFileInformationByHandle(

HANDLE hFile, // дескриптор файла

// указатель на информацию

LPBY\_HANDLE\_FILE\_INFORMATION lpFileInformation

);

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — false.

В параметре hFile этой функции должен быть установлен дескриптор файла, информацию о котором требуется получить. Отметим, что этот файл может быть открыт в любом режиме доступа.

Параметр lpFileInformation должен указывать на структуру типа

BY\_HANDLE\_FILE\_INFORMATION, в которую функция запишет информацию о файле. Эта структура имеет следующий формат:

typedef struct \_BY\_HANDLE\_FILE\_INFORMATION {

DWORD dwFileAttributes; // атрибуты файла

FILETIME ftCreationTime; // время создания файла

FILETIME ftLastAccessTime; // время последнего доступа к файлу

FILETIME ftLastWriteTime; // время последней записи в файл

DWORD dwVolumeSerialNumber; // серийный номер тома

DWORD nFileSizeHigh; // старшая часть размера файла

DWORD nFileSizeLow; // младшая часть размера файла

DWORD nNumberOfLinks; // количество ссылок на файл

DWORDbnFilelndexHigh; // старшая часть индекса файла

DWORD nFilelndexLow; // младшая часть индекса файла

} BY\_HANDLE\_FILE\_INFORMATION, \*LPBY\_HANDLE\_FILE\_INFORMATION;

Пример программы, которая получает информацию о файле и распечатывает ее.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

#include <windows.h>

#include <iostream.h>

#include <conio.h>

int main()

{

HANDLE hFile;

BY\_HANDLE\_FILE\_INFORMATION bhfi; // информация о файле

// открываем файл для чтения

hFile = CreateFile(

"D:\\demo.txt", // имя файла

0, // получение информации о файле

0, // монопольный доступ к файлу

NULL, // защиты нет

OPEN\_EXISTING, // открываем существующий файл

FILE\_ATTRIBUTE\_NORMAL, // обычный файл

NULL // шаблона нет

);

// проверяем на успешное открытие

if (hFile == INVALID\_HANDLE\_VALUE)

{

cerr << "Create file failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

// получаем информацию о файле

if (!GetFileInformationByHandle(hFile, &bhfi))

{

cerr << "Get file information by handle failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

// распечатываем информацию о файле

cout << "File attributes: " << bhfi.dwFileAttributes << endl

<< "Creation time: high date: "

<< bhfi.ftCreationTime.dwHighDateTime << endl

<<"Creation time: low date: "

<< bhfi.ftCreationTime.dwLowDateTime << endl

<< "Last access time: high date: "

<<bhfi. ftLastAccessTime.dwHighDateTime <<endl

<<"Last access time: low date: "

<< bhfi.ftLastAccessTime.dwLowDateTime <<endl

<<"Last write time: high date: "

<<bhfi.ftLastWriteTime.dwHighDateTime <<endl

<<"Last write time: low date: "

<<bhfi.ftLastWriteTime.dwLowDateTime <<endl

<<"Volume serial number: " <<bhfi.dwVolumeSerialNumber << endl

<<"File size high: " <<bhfi.nFileSizeHigh <<endl

<<"File size low: " <<bhfi.nFileSizeLow << endl

<<"Number of links: " << bhfi.nNumberOfLinks << endl

<< "File index high: " << bhfi.nFileIndexHigh << endl

<< "File index low: " << bhfi.nFileIndexLow << endl;

// закрываем дескриптор файла

CloseHandle(hFile);

getch();

return 0;

}

Структура типа BY\_HANDLE\_FILE\_INFORMATION содержит структуру типа FILETIME, которая служит для хранения времени. Эта структура имеет следующий формат:

typedef struct \_FILETIME {

DWORD dwLowDateTime; // младшая часть времени

DWORD dwHighDateTime; // старшая часть времени

} FILETIME, \*PFILETIME;

Само время задается в интервалах, каждый из которых равен 100 наносекунд. Естественно, что такое время неудобно просматривать пользователю. Поэтому для перевода времени в более удобную форму существует функция FileTimeToSystemTime, которая имеет следующий прототип:

BOOL FileTimeToSystemTime (

CONST FILETIME \*lpFileTime; // указатель на время в формате "файл"

LPSYSTEMTIME IpSystemTime // указатель на время в формате "система"

);

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — false.

Параметр lpFileTime этой функции должен указывать на структуру типа FILETIME, которая содержит время в формате, используемом для хранения в файловой системе.

Параметр lpSystemTime должен указывать на структуру типа SYSTEMTIME, которая имеет следующий формат:

typedef struct \_SYSTEMTIME {

WORD wYear; // год

WORD wMonth; // месяц

WORD wDayOfWeek; // день недели

WORD wDay; // день

WORD wHour; // час

WORD wMinute; // минута

WORD wSecond; // секунда

WORD wMilliseconds; // миллисекунда

} SYSTEMTIME, \*LPSYSTEMTIME;

Дополним программу:

SYSTEMTIME st; // системное время

if (!FileTimeToSystemTime(&(bhfi.ftCreationTime),&st))

{

cerr <<"File time to system time failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

// распечатываем системное время

cout << "File creation time in system format: " << endl

<<"\tYear: " << st.wYear << endl

<<"\tMonth: " << st.wMonth << endl

<<"\tDay of week: " << st.wDayOfWeek << endl

<<"\tDay: " << st.wDay << endl

<<"\tHour: " << st.wHour << endl

<<"\tMinute: " << st.wMinute << endl

<<"\tSecond: " << st.wSecond << endl

<<"\tMilliseconds: " << st.wMilliseconds << endl;

**Работа с каталогами (папками) в Windows**

Создание каталога

Для создания каталога используется функция CreateDirectory, которая имеет следующий прототип:

BOOL CreateDirectory(

LPCTSTR lpPathName, // имя каталога

LPSECUTITY\_ATTRIBUTES lpSecurutyAttributes // атрибуты защиты

);

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — false. В параметре lpPathName задается указатель на символьную строку, которая содержит имя создаваемого каталога, а параметр lpSecurutyAttributes задает атрибуты безопасности этого каталога.

Пример программы, которая создает каталог.

#include <vcl.h>

#pragma hdrstop

#pragma argsused

# include <windows.h>

#include <iostream.h>

#include <conio.h>

int main()

{

// создаем каталог

if (!CreateDirectory("D:\\Papka", NULL))

{

cerr << "Create directory failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

cout << "The directory is created." <<endl;

getch();

return 0;

}

После создания каталога можно получить его дескриптор, используя для этого функцию CreateFile с установленным флагом FILE\_FLAG\_BACKUP\_SEMANTICS.

Для создания подкаталогов можно использовать функцию CreateDirectoryEx, которая позволяет наследовать атрибуты другого каталога, который в этом случае называется шаблонным каталогом. В качестве шаблонного каталога можно указать родительский каталог, в котором создается текущий каталог. Функция CreateDirectoryEx имеет следующий прототип:

BOOL CreateDirectoryEx(

LPCTSTR lpTemplateDirectory, // имя шаблонного каталога

LPCTSTR lpNewDirectory, // имя нового каталога

LPSECUTITY\_ATTRIBUTES lpSecurutyAttributes // атрибуты зашиты

);

Эта функция отличается от функции CreateDirectory только наличием параметра lpTemplateDirectory, который содержит указатель на имя шаблонного каталога.

Например, программа создает подкаталог Papka1в каталоге Papka, созданном предыдущей программой.

int main ()

{

if (! CreateDirectoryEx ("D:\\Papka", "D:\\Papka\\Papka1", NULL))

{

cerr <<"Create directory failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

cout << "The subdirectory is created." <<endl;

return 0;

}

**Удаление каталога**

Для удаления пустого каталога предназначена функция RemoveDirectory, которая имеет следующий прототип:

BOOL RemoveDirectory (

LPCTSTR lpPathName // имя каталога

);

Единственный параметр этой функции должен указывать на символьную строку, содержащую имя удаляемого каталога. В случае успешного завершения функция RemoveDirectory возвращает ненулевое значение, а в случае неудачи — false.

На практике каталог редко бывает пустым, поэтому прежде чем удалить сам каталог, нужно удалить все находящиеся в нем подкаталоги и файлы. Приведем программу, которая удаляет из каталога файлы, а затем удаляет сам каталог.

#include <windows.h>

#include <iostream.h>

#include <stdio.h>

int main()

{

HANDLE hFindFile;

WIN32\_FIND\_DATA fd;

char szFullFileName[MAX\_PATH] ;

// находим первый файл

hFindFile = FindFirstFile("D:\\Papka\\\*", &fd) ;

if (hFindFile == INVALID\_HANDLE\_VALUE)

{

cerr <<"Find first file failed." <<endl

<< "The last error code: " <<GetLastError() << endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

// выводим на консоль имя первого файла

cout << "The first file name: " << fd.cFileName << endl;

// удаляем из каталога файлы

while (FindNextFile(hFindFile, &fd))

{

// если это не подкаталог, то удаляем его

if (!(fd.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY))

{

// формируем имя файла

sprintf(szFullFileName,"D:\\Papka\\%s",fd.cFileName);

// удаляем файл

if (!DeleteFile(szFullFileName))

{

cerr << "Delete file failed." <<endl

<< "The last error code: " << GetLastError() << endl;

cout<< "Press any key to finish.";

cin.get();

return 0;

}

else

cout << "The next file: " << fd.cFileName << " is deleted." <<endl;

}

else

cout <<"The next directory: " << fd.cFileName << " is not deleted."

<< endl;

}

// закрываем дескриптор поиска

if (!FindClose(hFindFile))

{

cout << "Find close failed." << endl;

return 0;

}

// удаляем каталог

if (!RemoveDirectory("D:\\Papka"))

{

cerr << "Remove directory failed." <<endl

<< "The last error code: " << GetLastError()<< endl;

cout << "Press any key to finish.";

cin.get();

return 0;

}

cout<< "The directory is removed." << endl;

// закрываем дескриптор поиска

FindClose(hFindFile);

return 0;

}

**Перемещение каталога**

Так же как и файлы, каталоги можно перемещать при помощи функции MoveFile. В этом случае на исполнение функции MoveFile накладывается одно ограничение, которое заключается в том, что можно перемещать только каталоги, которые находятся на одном томе.

Функция MoveFile имеет следующий прототип:

BOOL MoveFile(

LPCTSTR IpExistingFileName, // имя существующего файла

LPCTSTR lpNewFileName // имя нового файла (создается функцией)

);

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение false. В случае перемещения каталогов параметры lpExistingFileName и lpNewFileName должны указывать на строки, которые содержат имена перемещаемого и нового каталогов соответственно.

Приведем программа, которая выполняет перемещение каталога, используя функцию MoveFile.

#include <windows.h>

#include <iostream.h>

int main()

{

if(!MoveFile("D:\\Papka1", "D:\\Papka"))

{

cerr << "Move file failed." << endl

<< "The last error code: " << GetLastError() << endl; cout << "Press any key to finish.";

cin.get();

return 0;

}

cout << "The directory is moved." << endl; return 0; }