МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ»

Г.Э. Вощинская, М.А. Артемов

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 1

Учебно-методическое пособие для вузов

Издательско-полиграфический центр
Воронежского государственного университета
2012

Утверждено научно-методическим советом факультета прикладной математики, информатики и механики от 29 мая 2012 г., протокол № 10

Рецензент зав. кафедрой ПиИТ ф-та ФКН ВГУ Н.А. Тюкачев

Учебно-методическое пособие подготовлено на кафедре программного обеспечения и администрирования информационных систем факультета прикладной математики, информатики и механики Воронежского государственного университета.

Рекомендуется для студентов 3-го курса дневного отделения.

Для специальности 010503 – Математическое обеспечение и администрирование информационных систем.

Содержание

Введение

Определение операционной системы

Операционная система в наибольшей степени определяет облик всей вычислительной системы в целом. Несмотря на это, пользователи, активно использующие вычислительную технику, зачастую испытывают затруднения при попытке дать определение операционной системе. Частично это связано с тем, что ОС выполняет две по существу мало связанные между собой функции:

- обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины;
- повышение эффективности использования компьютера путем рационального управления его ресурсами.

Операционная система (ОС) – есть *организованная совокупность процессов*, которая действует как интерфейс между аппаратурой компьютера и пользователями. Она обеспечивает пользователей набором средств для обеспечения проектирования, программирования, отладки и сопровождения программ; и в то же время управляет распределением ресурсов для обеспечения эффективной работы.

ОС как расширенная машина

Использование большинства компьютеров на уровне машинного языка затруднительно, особенно это касается ввода-вывода. Например, для организации чтения блока данных с гибкого диска программист может использовать 16 различных команд, каждая из которых требует 13 параметров, таких как номер блока на диске, номер сектора на дорожке и т. п. Когда выполнение операции с диском завершается, контроллер возвращает 23 значения, отражающих наличие и типы ошибок, которые, очевидно, надо анализировать. Даже если не входить в курс реальных проблем программирования ввода-вывода, ясно, что среди программистов нашлось бы немного желающих непосредственно заниматься программированием этих операций. При работе с диском программисту-пользователю достаточно представлять его в виде некоторого набора файлов, каждый из которых имеет имя. Работа с файлом заключается в его открытии, выполнении чтения или записи, а затем в закрытии файла. Вопросы подобные таким, как следует ли при записи использовать усовершенствованную частотную модуляцию или в каком состоянии сейчас находится двигатель механизма перемещения считывающих головок, не должны волновать пользователя. Программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного просмотра указанных файлов, чтения или записи – это, конечно, операционная система. Точно так же, как ОС ограждает программистов от аппаратуры дискового накопителя и предоставляет ему простой

файловый интерфейс, операционная система берет на себя все малоприятные дела, связанные с обработкой прерываний, управлением таймерами и оперативной памятью, а также другие низкоуровневые проблемы. В каждом случае та абстрактная, воображаемая машина, с которой, благодаря операционной системе, теперь может иметь дело пользователь, гораздо проще и удобнее в обращении, чем реальная аппаратура, лежащая в основе этой абстрактной машины.

С этой точки зрения функция ОС – предоставление пользователю некоторой расширенной или виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину.

ОС как система управления ресурсами

Представление ОС как системы, обеспечивающей удобный интерфейс пользователям, соответствует рассмотрению сверху вниз. Другой взгляд, снизу вверх, дает представление об ОС как о некотором механизме, управляющем всеми частями сложной системы. Современные вычислительные системы состоят из процессоров, памяти, таймеров, дисков, накопителей на магнитных лентах, сетевой коммуникационной аппаратуры, принтеров и других устройств. В соответствии со вторым подходом функцией ОС является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. ОС должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования. Критерием эффективности может быть, например, пропускная способность или реактивность системы. Управление ресурсами включает решение двух общих, не зависящих от типа ресурса задач:

- планирование ресурса – то есть определение, кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс;
- отслеживание состояния ресурса – то есть поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов – какое количество ресурса уже распределено, а какое свободно.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, что, в конечном счете, и определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Так, например, алгоритм управления процессором в значительной степени определяет, является ли ОС системой разделения времени, системой пакетной обработки или системой реального времени.

Эволюция ОС

Первый период (1945–1955)

Известно, что компьютер был изобретен английским математиком Чарльзом Бэббиджем в конце восемнадцатого века. Его «аналитическая машина» так и не смогла по-настоящему заработать, потому что технологии того времени не удовлетворяли требованиям по изготовлению деталей точной механики, которые были необходимы для вычислительной техники. Известно также, что этот компьютер не имел операционной системы.

Некоторый прогресс в создании цифровых вычислительных машин произошел после Второй мировой войны. В середине 40-х годов XX в. были созданы первые ламповые вычислительные устройства. В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. Не было никакого другого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм.

Второй период (1955–1965)

С середины 50-х годов XX в. начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Компьютеры второго поколения стали более надежными, теперь они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно важных практических задач. Именно в этот период произошло разделение персонала на программистов и операторов, эксплуатационников и разработчиков вычислительных машин.

В эти годы появились первые алгоритмические языки, а, следовательно, и первые системные программы – компиляторы. Стоимость процессорного времени возросла, что потребовало уменьшения непроизводительных затрат времени между запусками программ. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом. В ходе реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине.

Совокупность нескольких заданий, как правило, в виде колоды перфокарт, получила название пакета заданий.

Третий период (1965–1980)

Следующий важный период развития вычислительных машин относится к 1965–1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что дало большие возможности новому, третьему поколению компьютеров.

Для этого периода характерно также создание семейств программно-совместимых машин. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию цена/производи-тельность. Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными «монстрами». Они состояли из многих миллионов ассемблерных строк, написанных тысячами программистов, и содержали тысячи ошибок, вызывающих нескончаемый поток исправлений. В каждой новой версии операционной системы исправлялись одни ошибки и вносились другие.

Однако, несмотря на необозримые размеры и множество проблем, OS/360 и другие ей подобные операционные системы машин третьего поколения действительно удовлетворяли большинству требований потребителей. Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования. ***Мультипрограммирование*** – это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении программ (однопрограммный режим), а выполняет другую программу (многопрограммный режим). При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом.

Другое нововведение – спулинг (spooling). Спулинг в то время определялся как способ организации вычислительного процесса, в соответствии с которым задания считывались с перфокарт на диск в том темпе, в котором они появлялись в помещении вычислительного центра, а затем, когда очередное задание завершалось, новое задание с диска загружалось в освободившийся раздел.

Наряду с мультипрограммной реализацией систем пакетной обработки появился новый тип ОС – системы разделения времени. Вариант мультипрограммирования, применяемый в системах разделения времени, нацелен на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины.

Четвертый период (1980–1990)

Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку, и наступила эра персональных компьютеров. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11, но вот цена у них существенно отличалась. Если миникомпьютер дал возможность иметь собственную вычислительную машину отделу предприятия или университету, то персональный компьютер сделал это возможным для отдельного человека.

Компьютеры стали широко использоваться неспециалистами, что потребовало разработки «дружественного» программного обеспечения, это положило конец кастовости программистов.

На рынке операционных систем доминировали две системы: MS-DOS и UNIX. Однопрограммная однопользовательская ОС MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская ОС UNIX доминировала в среде «неинтеловских» компьютеров, особенно построенных на базе высокопроизводительных RISC-процессоров.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных ОС.

В сетевых ОС пользователи должны быть осведомлены о наличии других компьютеров и должны делать логический вход в другой компьютер, чтобы воспользоваться его ресурсами, преимущественно файлами. Каждая машина в сети выполняет свою собственную локальную операционную систему, отличающуюся от ОС автономного компьютера наличием дополнительных средств, позволяющих компьютеру работать в сети. Сетевая ОС не имеет фундаментальных отличий от ОС однопроцессорного компьютера. Она обязательно содержит программную поддержку для сетевых интерфейсных устройств (драйвер сетевого адаптера), а также средства для удаленного входа в другие компьютеры сети и средства доступа к удаленным файлам, однако эти дополнения существенно не меняют структуру самой операционной системы.

Пятый период (1990–настоящее время)

Мировая гонка за создание компьютера пятого поколения началась еще в 1981 году.

Основные требования к компьютерам 5-го поколения:

• создание развитого человеко-машинного интерфейса (распознавание речи, образов);

• развитие логического программирования для создания баз знаний и систем искусственного интеллекта;

• создание новых технологий в производстве вычислительной техники;

• создание новых архитектур компьютеров и вычислительных комплексов.

K computer (Fujitsu, RIKEN, Advanced Institute for Computational Science, Kobe) создан на базе 8-ядерных SPARC64 VIIIfx процессоров , признан самым мощным в мире суперкомпьютером согласно версии списка TOP500 на ноябрь 2011 года. Его характеристики:

• производительность – 8,162 петафлопс (PFlops = 10^{15} Flops);

• общее количество процессорных ядер 548 352 в 68 544 процессорах;

• 1032 терабайта оперативной памяти (Тбайт = 10^{12} байт);

• соединение сети – шестимерный тор, называемый *тофу*;

• проект по созданию суперкомпьютера оценивается в € 975 млн.

Классификация ОС

Операционные системы могут различаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами.

Ниже приведена классификация ОС по нескольким основным признакам.

Особенности алгоритмов управления ресурсами

От эффективности алгоритмов управления локальными ресурсами компьютера во многом зависит эффективность всей сетевой ОС в целом. Поэтому, характеризуя сетевую ОС, часто приводят важнейшие особенности реализации функций ОС по управлению процессорами, памятью, внешними устройствами автономного компьютера. Так, например, в зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на многозадачные и однозадачные, многопользовательские и однопользовательские, на системы, поддерживающие многонитевую обработку и не поддерживающие ее, на многопроцессорные и однопроцессорные системы.

Поддержка многозадачности. По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса:

• однозадачные (например, MS-DOS, MSX) и

• многозадачные (ОС ЕС, OS/2, UNIX, Windows 95).

Однозадачные ОС в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным про-

цесс взаимодействия пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Многозадачные ОС, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

Поддержка многопользовательского режима. По числу одновременно работающих пользователей ОС делятся на:

• однопользовательские (MS-DOS, Windows 3.x, ранние версии OS/2);

• многопользовательские (UNIX, Windows NT).

Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

Вытесняющая и невытесняющая многозадачность. Важнейшим разделяемым ресурсом является процессорное время. Способ распределения процессорного времени между несколькими одновременно существующими в системе процессами (или нитями) во многом определяет специфику ОС. Среди множества существующих вариантов реализации многозадачности можно выделить две группы алгоритмов:

• невытесняющая многозадачность (NetWare, Windows 3.x);

• вытесняющая многозадачность (Windows NT, OS/2, UNIX).

Основным различием между вытесняющим и невытесняющим вариантами многозадачности является степень централизации механизма планирования процессов. В первом случае механизм планирования процессов целиком сосредоточен в операционной системе, а во втором – распределен между системой и прикладными программами. При невытесняющей многозадачности активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению процесс. При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается операционной системой, а не самим активным процессом.

Поддержка многонитевости. Важным свойством операционных систем является возможность распараллеливания вычислений в рамках одной задачи. Многонитевая ОС разделяет процессорное время не между задачами, а между их отдельными ветвями (нитями).

Многопроцессорная обработка. Другим важным свойством ОС является отсутствие или наличие в ней средств поддержки многопроцессорной обработки – мультипроцессирование. Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами.

В наши дни становится общепринятым введение в ОС функций поддержки многопроцессорной обработки данных. Такие функции имеются в операционных системах Solaris 2.x фирмы Sun, Open Server 3.x компании Santa Crus Operations, OS/2 фирмы IBM, Windows NT фирмы Microsoft и NetWare 4.1 фирмы Novell.

Многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой: асимметричные ОС и симметричные ОС. Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

Выше были рассмотрены характеристики ОС, связанные с управлением только одним типом ресурсов – процессором. Важное влияние на облик операционной системы в целом, на возможности ее использования в той или иной области оказывают особенности и других подсистем управления локальными ресурсами – подсистем управления памятью, файлами, устройствами ввода-вывода.

Специфика ОС проявляется и в том, каким образом она реализует сетевые функции: распознавание и перенаправление в сеть запросов к удаленным ресурсам, передача сообщений по сети, выполнение удаленных запросов. При реализации сетевых функций возникает комплекс задач, связанных с распределенным характером хранения и обработки данных в сети: ведение справочной информации обо всех доступных в сети ресурсах и серверах, адресация взаимодействующих процессов, обеспечение прозрачности доступа, тиражирование данных, согласование копий, поддержка безопасности данных.

Особенности аппаратных платформ

На свойства операционной системы непосредственное влияние оказывают аппаратные средства, на которые она ориентирована. По типу аппаратуры различают операционные системы персональных компьютеров, миникомпьютеров, мейнфреймов, кластеров и сетей ЭВМ.

Среди перечисленных типов компьютеров могут встречаться как однопроцессорные варианты, так и многопроцессорные. В любом случае специфика аппаратных средств, как правило, отражается на специфике операционных систем.

Очевидно, что ОС большой машины является более сложной и функциональной, чем ОС персонального компьютера. Так, в ОС больших машин функции по планированию потока выполняемых задач, очевидно, реализуются путем использования сложных приоритетных дисциплин и требуют большей вычислительной мощности, чем в ОС персональных компьютеров. Аналогично обстоит дело и с другими функциями.

Сетевая ОС имеет в своем составе средства передачи сообщений между компьютерами по линиям связи, которые совершенно не нужны в автономной ОС. На основе этих сообщений сетевая ОС поддерживает разделение ресурсов компьютера между удаленными пользователями, подключенными к сети. Для поддержания функций передачи сообщений сетевые ОС содержат специальные программные компоненты, реализующие популярные коммуникационные протоколы, такие как IP, IPX, Ethernet и другие.

Многопроцессорные системы требуют от операционной системы особой организации, с помощью которой сама операционная система, а также поддерживаемые ею приложения могли бы выполняться параллельно отдельными процессорами системы. Параллельная работа отдельных частей ОС создает дополнительные проблемы для разработчиков ОС, так как в этом случае гораздо сложнее обеспечить согласованный доступ отдельных процессов к общим системным таблицам, исключить эффект гонок и прочие нежелательные последствия асинхронного выполнения работ.

Другие требования предъявляются к операционным системам кластеров. Кластер – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой. Наряду со специальной аппаратурой для функционирования кластерных систем необходима и программная поддержка со стороны операционной системы, которая сводится в основном к синхронизации доступа к разделяемым ресурсам, обнаружению отказов и динамической реконфигурации системы. Одной из первых разработок в области кластерных технологий были решения компании Digital Equipment на базе компьютеров VAX. Этой компанией заключено соглашение с корпорацией Microsoft о разработке кластерной технологии, использующей Windows NT. Несколько компаний предлагают кластеры на основе UNIX-машин.

Наряду с ОС, ориентированными на совершенно определенный тип аппаратной платформы, существуют операционные системы, специально разработанные таким образом, чтобы они могли быть легко перенесены с компьютера одного типа на компьютер другого типа, так называемые мобильные ОС. Наиболее ярким примером такой ОС является популярная система UNIX. В этих системах аппаратно-зависимые места тщательно локализованы, так что при переносе системы на новую платформу переписываются только они. Средством, облегчающем перенос остальной части ОС, является написание ее на машинно-независимом языке, например, на С, который и был разработан для программирования операционных систем.

Особенности областей использования

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

- системы пакетной обработки (например, ОС ЕС),
- системы разделения времени (UNIX, VMS),
- системы реального времени (QNX, RT/11).

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели в системах пакетной обработки используются следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины; так, например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается «выгодное» задание. Следовательно, в таких ОС невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит только в случае, если активная задача сама отказывается от процессора, например, из-за необходимости выполнить операцию ввода-вывода. Поэтому одна задача может надолго занять процессор, что делает невозможным выполнение интерактивных задач. Таким образом, взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок снижает эффективность работы пользователя.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки – изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю системы разделения времени предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что

каждый из них единолично использует машину. Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая "выгодна" системе, и, кроме того, имеются накладные расходы вычислительной мощности на более частое переключение процессора с задачи на задачу. Критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя.

Системы реального времени применяются для управления различными техническими объектами, такими, например, как станок, спутник, научная экспериментальная установка или технологическими процессами, такими, как гальваническая линия, доменный процесс и т.п. Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа, управляющая объектом, в противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности для систем реального времени является их способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия). Это время называется временем реакции системы, а соответствующее свойство системы – реактивностью. Для этих систем мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется исходя из текущего состояния объекта или в соответствии с расписанием плановых работ.

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть задач может выполняться в режиме пакетной обработки, а часть – в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

Особенности методов построения

При описании операционной системы часто указываются особенности ее структурной организации и основные концепции, положенные в ее основу.

К таким базовым концепциям относятся:

- Способы построения ядра системы – монолитное ядро или микроядерный подход. Большинство ОС использует монолитное ядро, которое компонуется как один процесс, работающий в привилегированном режиме и использующий быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский и наоборот. Альтернативой является построение ОС на базе микроядра, работающего так-

же в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС – серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским, зато система получается более гибкой – ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

• Построение ОС на базе объектно-ориентированного подхода дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри операционной системы, а именно: аккумуляцию удачных решений в форме стандартных объектов, возможность создания новых объектов на базе имеющихся с помощью механизма наследования, хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне, структурированность системы, состоящей из набора хорошо определенных объектов.

• Наличие нескольких прикладных сред дает возможность в рамках одной ОС одновременно выполнять приложения, разработанные для нескольких ОС. Многие современные операционные системы поддерживают одновременно прикладные среды MS-DOS, Windows, UNIX (POSIX), OS/2 или хотя бы некоторого подмножества из этого популярного набора. Концепция множественных прикладных сред наиболее просто реализуется в ОС на базе микроядра, над которым работают различные серверы, некоторые из которых реализуют прикладную среду той или иной операционной системы.

• Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах. В распределенной ОС реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера. Характерными признаками распределенной организации ОС являются: наличие единой справочной службы разделяемых ресурсов, единой службы времени, использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам, многонитевой обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети, а также наличие других распределенных служб.

Структура сетевой операционной системы

Сетевая операционная система составляет основу любой вычислительной сети. Каждый компьютер в сети в значительной степени автономен, по-

этому под сетевой операционной системой в широком смысле понимается совокупность операционных систем отдельных компьютеров, взаимодействующих с целью обмена сообщениями и разделения ресурсов по единым правилам – протоколам. В узком смысле сетевая ОС – это операционная система отдельного компьютера, обеспечивающая ему возможность работать в сети.

В сетевой операционной системе отдельной машины можно выделить несколько частей.

• Средства управления локальными ресурсами компьютера: функции распределения оперативной памяти между процессами планирования и диспетчеризации процессов, управления процессорами в мультипроцессорных машинах, управления периферийными устройствами и другие функции управления ресурсами локальных ОС.

• Средства предоставления собственных ресурсов и услуг в общее пользование – серверная часть ОС (сервер). Эти средства обеспечивают, например, блокировку файлов и записей, что необходимо для их совместного использования; ведение справочников имен сетевых ресурсов; обработку запросов удаленного доступа к собственной файловой системе и базе данных; управление очередями запросов удаленных пользователей к своим периферийным устройствам.

• Средства запроса доступа к удаленным ресурсам и услугам и их использования – клиентская часть ОС (редиректор). Эта часть выполняет распознавание и перенаправление в сеть запросов к удаленным ресурсам от приложений и пользователей, при этом запрос поступает от приложения в локальной форме, а передается в сеть в другой форме, соответствующей требованиям сервера. Клиентская часть также осуществляет прием ответов от серверов и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо.

• Коммуникационные средства ОС, с помощью которых происходит обмен сообщениями в сети. Эта часть обеспечивает адресацию и буферизацию сообщений, выбор маршрута передачи сообщения по сети, надежность передачи и т.п., то есть является средством транспортировки сообщений.

В зависимости от функций, возлагаемых на конкретный компьютер, в его операционной системе может отсутствовать либо клиентская, либо серверная части.

На практике сложилось несколько подходов к построению сетевых операционных систем

Первые сетевые ОС представляли собой совокупность существующей локальной ОС и надстроенной над ней сетевой оболочки. При этом в локальную ОС встраивался минимум сетевых функций, необходимых для работы сетевой оболочки, которая выполняла основные сетевые функции. Примером такого подхода является использование на каждой машине се-

ти операционной системы MS DOS (у которой, начиная с ее третьей версии, появились такие встроенные функции, как блокировка файлов и записей, необходимые для совместного доступа к файлам). Принцип построения сетевых ОС в виде сетевой оболочки над локальной ОС используется и в современных ОС, таких, например, как LANtastic или Personal Ware.

Однако более эффективным представляется путь разработки операционных систем, изначально предназначенных для работы в сети. Сетевые функции у ОС такого типа глубоко встроены в основные модули системы, что обеспечивает их логическую стройность, простоту эксплуатации и модификации, а также высокую производительность. Примером такой ОС является система Windows NT фирмы Microsoft, которая за счет встроенности сетевых средств обеспечивает более высокие показатели производительности и защищенности информации по сравнению с сетевой ОС LAN Manager той же фирмы (совместная разработка с IBM), являющейся надстройкой над локальной операционной системой OS/2.

Одноранговые сетевые ОС и ОС с выделенными серверами

В зависимости от того, как распределены функции между компьютерами сети, сетевые операционные системы, а следовательно, и сети делятся на два класса: одноранговые и двухранговые. Последние чаще называют сетями с выделенными серверами.

Если компьютер предоставляет свои ресурсы другим пользователям сети, то он играет роль сервера. При этом компьютер, обращающийся к ресурсам другой машины, является клиентом. Как уже было сказано, компьютер, работающий в сети, может выполнять функции либо клиента, либо сервера, либо совмещать обе эти функции.

Если выполнение каких-либо серверных функций является основным назначением компьютера (например, предоставление файлов в общее пользование всем остальным пользователям сети или организация совместного использования факса, или предоставление всем пользователям сети возможности запуска на данном компьютере своих приложений), то такой компьютер называется выделенным сервером. В зависимости от того, какой ресурс сервера является разделяемым, он называется файл-сервером, факс-сервером, принт-сервером, сервером приложений и т.д.

Очевидно, что на выделенных серверах желательно устанавливать ОС, специально оптимизированные для выполнения тех или иных серверных функций. Поэтому в сетях с выделенными серверами чаще всего используются сетевые операционные системы, в состав которых входит нескольких вариантов ОС, отличающихся возможностями серверных частей. Например, сетевая ОС Novell NetWare имеет серверный вариант, оптимизированный для работы в качестве файл-сервера, а также варианты оболочек для рабо-

чих станций с различными локальными ОС, причем эти оболочки выполняют исключительно функции клиента. Другим примером ОС, ориентированной на построение сети с выделенным сервером, является операционная система Windows NT. В отличие от NetWare, оба варианта данной сетевой ОС – Windows NT Server (для выделенного сервера) и Windows NT Workstation (для рабочей станции) – могут поддерживать функции и клиента и сервера. Но серверный вариант Windows NT имеет больше возможностей для предоставления ресурсов своего компьютера другим пользователям сети, так как может выполнять более широкий набор функций, поддерживает большее количество одновременных соединений с клиентами, реализует централизованное управление сетью, имеет более развитые средства защиты.

Выделенный сервер не принято использовать в качестве компьютера для выполнения текущих задач, не связанных с его основным назначением, так как это может уменьшить производительность его работы как сервера. В связи с такими соображениями в ОС Novell NetWare на серверной части возможность выполнения обычных прикладных программ вообще не предусмотрена, то есть сервер не содержит клиентской части, а на рабочих станциях отсутствуют серверные компоненты. Однако в других сетевых ОС функционирование на выделенном сервере клиентской части вполне возможно. Например, под управлением Windows NT Server могут запускаться обычные программы локального пользователя, которые могут потребовать выполнения клиентских функций ОС при появлении запросов к ресурсам других компьютеров сети. При этом рабочие станции, на которых установлена ОС Windows NT Workstation, могут выполнять функции невыделенного сервера.

Важно понять, что несмотря на то, что в сети с выделенным сервером все компьютеры в общем случае могут выполнять одновременно роли и сервера, и клиента, эта сеть функционально не симметрична: аппаратно и программно в ней реализованы два типа компьютеров – одни, в большей степени ориентированные на выполнение серверных функций и работающие под управлением специализированных серверных ОС, а другие – в основном выполняющие клиентские функции и работающие под управлением соответствующего этому назначению варианта ОС. Функциональная несимметричность, как правило, вызывает и несимметричность аппаратуры – для выделенных серверов используются более мощные компьютеры с большими объемами оперативной и внешней памяти. Таким образом, функциональная несимметричность в сетях с выделенным сервером сопровождается несимметричностью операционных систем (специализация ОС) и аппаратной несимметричностью (специализация компьютеров).

В одноранговых сетях все компьютеры равны в правах доступа к ресурсам друг друга. Каждый пользователь может по своему желанию объя-

вить какой-либо ресурс своего компьютера разделяемым, после чего другие пользователи могут его эксплуатировать. В таких сетях на всех компьютерах устанавливается одна и та же ОС, которая предоставляет всем компьютерам в сети потенциально равные возможности. Одноранговые сети могут быть построены, например, на базе ОС LANtastic, Personal Ware, Windows for Workgroup, Windows NT Workstation.

В одноранговых сетях также может возникнуть функциональная несимметричность: одни пользователи не желают разделять свои ресурсы с другими, и в таком случае их компьютеры исполняют роль клиента; за другими компьютерами администратор закрепил только функции по организации совместного использования ресурсов, а значит, они являются серверами; в третьем случае, когда локальный пользователь не возражает против использования его ресурсов и сам не исключает возможности обращения к другим компьютерам, ОС, устанавливаемая на его компьютере, должна включать и серверную, и клиентскую части. В отличие от сетей с выделенными серверами, в одноранговых сетях отсутствует специализация ОС в зависимости от преобладающей функциональной направленности – клиента или сервера. Все вариации реализуются средствами конфигурирования одного и того же варианта ОС.

Одноранговые сети проще в организации и эксплуатации, однако, они применяются в основном для объединения небольших групп пользователей, не предъявляющих больших требований к объемам хранимой информации, ее защищенности от несанкционированного доступа и к скорости доступа. При повышенных требованиях к этим характеристикам более подходящими являются двухранговые сети, где сервер лучше решает задачу обслуживания пользователей своими ресурсами, так как его аппаратура и сетевая операционная система специально спроектированы для этой цели.

ОС для рабочих групп и ОС для сетей масштаба предприятия

Сетевые операционные системы имеют разные свойства в зависимости от того, предназначены они для сетей масштаба рабочей группы (отдела), для сетей масштаба кампуса или для сетей масштаба предприятия.

• *Сети отделов* – используются небольшой группой сотрудников, решающих общие задачи. Главной целью сети отдела является разделение локальных ресурсов, таких как приложения, данные, лазерные принтеры и модемы. Сети отделов обычно не разделяются на подсети.

• *Сети кампусов* – соединяют несколько сетей отделов внутри отдельного здания или внутри одной территории предприятия. Эти сети являются все еще локальными сетями, хотя и могут покрывать территорию в несколько квадратных километров. Сервисы такой сети включают взаимодействие между сетями отделов, доступ к базам данных предприятия, доступ к факс-серверам, высокоскоростным модемам и высокоскоростным принтерам.

• *Сети предприятия* (корпоративные сети) – объединяют все компьютеры всех территорий отдельного предприятия. Они могут покрывать город, регион или даже континент. В таких сетях пользователям предоставляется доступ к информации и приложениям, находящимся в других рабочих группах, других отделах, подразделениях и штаб-квартирах корпорации.

Главной задачей операционной системы, используемой в сети масштаба отдела, является организация разделения ресурсов, таких как приложения, данные, лазерные принтеры и, возможно, низкоскоростные модемы. Обычно сети отделов имеют один или два файловых сервера и не более чем 30 пользователей. Задачи управления на уровне отдела относительно просты. В задачи администратора входит добавление новых пользователей, устранение простых отказов, инсталляция новых узлов и установка новых версий программного обеспечения. Операционные системы сетей отделов хорошо отработаны и разнообразны, так же, как и сами сети отделов, уже давно применяющиеся и достаточно отлаженные. Такая сеть обычно использует одну или максимум две сетевые ОС. Чаще всего это сеть с выделенным сервером NetWare 3.x или Windows NT, или же одноранговая сеть, например, сеть Windows for Workgroups.

Пользователи и администраторы сетей отделов вскоре осознают, что они могут улучшить эффективность своей работы путем получения доступа к информации других отделов своего предприятия. Если сотрудник, занимающийся продажами, может получить доступ к характеристикам конкретного продукта и включить их в презентацию, то эта информация будет более свежей, и будет оказывать большее влияние на покупателей. Если отдел маркетинга может получить доступ к характеристикам продукта, который еще только разрабатывается инженерным отделом, то он может быстро подготовить маркетинговые материалы сразу же после окончания разработки.

Итак, следующим шагом в эволюции сетей является объединение локальных сетей нескольких отделов в единую сеть здания или группы зданий. Такие сети называют сетями кампусов. Сети кампусов могут простираться на несколько километров, но при этом глобальные соединения не требуются.

Операционная система, работающая в сети кампуса, должна обеспечивать для сотрудников одних отделов доступ к некоторым файлам и ресурсам сетей других отделов. Услуги, предоставляемые ОС сетей кампусов, не ограничиваются простым разделением файлов и принтеров, а часто предоставляют доступ и к серверам других типов, например, к факс-серверам и к серверам высокоскоростных модемов. Важным сервисом, предоставляемым операционными системами данного класса, является доступ к корпоративным базам данных, независимо от того, располагаются ли они на серверах баз данных или на миникомпьютерах.

Именно на уровне сети кампуса начинаются проблемы интеграции. В общем случае, отделы уже выбрали для себя типы компьютеров, сетевого оборудования и сетевых операционных систем. Например, инженерный отдел может использовать операционную систему UNIX и сетевое оборудование Ethernet, отдел продаж может использовать операционные среды DOS/Novell и оборудование Token Ring. Очень часто сеть кампуса соединяет разнородные компьютерные системы, в то время как сети отделов используют однотипные компьютеры.

Корпоративная сеть соединяет сети всех подразделений предприятия, в общем случае находящихся на значительных расстояниях. Корпоративные сети используют глобальные связи (WAN links) для соединения локальных сетей или отдельных компьютеров.

Пользователям корпоративных сетей требуются все те приложения и услуги, которые имеются в сетях отделов и кампусов, плюс некоторые дополнительные приложения и услуги, например, доступ к приложениям мейнфреймов и миникомпьютеров и к глобальным связям. Когда ОС разрабатывается для локальной сети или рабочей группы, то ее главной обязанностью является разделение файлов и других сетевых ресурсов (обычно принтеров) между локально подключенными пользователями. Такой подход не применим для уровня предприятия. Наряду с базовыми сервисами, связанными с разделением файлов и принтеров, сетевая ОС, которая разрабатывается для корпораций, должна поддерживать более широкий набор сервисов, в который обычно входят почтовая служба, средства коллективной работы, поддержка удаленных пользователей, факс-сервис, обработка голосовых сообщений, организация видеоконференций и др.

Кроме того, многие существующие методы и подходы к решению традиционных задач сетей меньших масштабов для корпоративной сети оказались непригодными. На первый план вышли такие задачи и проблемы, которые в сетях рабочих групп, отделов и даже кампусов либо имели второстепенное значение, либо вообще не проявлялись. Например, простейшая для небольшой сети задача ведения учетной информации о пользователях выросла в сложную проблему для сети масштаба предприятия. А использование глобальных связей требует от корпоративных ОС поддержки протоколов, хорошо работающих на низкоскоростных линиях, и отказа от некоторых традиционно используемых протоколов (например, тех, которые активно используют широковещательные сообщения). Особое значение приобрели задачи преодоления гетерогенности – в сети появились многочисленные шлюзы, обеспечивающие согласованную работу различных ОС и сетевых системных приложений.

К признакам корпоративных ОС могут быть отнесены также следующие особенности.

Поддержка приложений. В корпоративных сетях выполняются сложные приложения, требующие для выполнения большой вычислительной мощности. Такие приложения разделяются на несколько частей, например, на одном компьютере выполняется часть приложения, связанная с выполнением запросов к базе данных, на другом – запросов к файловому сервису, а на клиентских машинах – часть, реализующая логику обработки данных приложения и организующая интерфейс с пользователем. Вычислительная часть общих для корпорации программных систем может быть слишком объемной и неподъемной для рабочих станций клиентов, поэтому приложения будут выполняться более эффективно, если их наиболее сложные в вычислительном отношении части перенести на специально предназначенный для этого мощный компьютер – сервер приложений.

Сервер приложений должен базироваться на мощной аппаратной платформе (мультипроцессорные системы, часто на базе RISC-процессоров, специализированные кластерные архитектуры). ОС сервера приложений должна обеспечивать высокую производительность вычислений, а значит, поддерживать многонитевую обработку, вытесняющую многозадачность, мультипроцессирование, виртуальную память и наиболее популярные прикладные среды (UNIX, Windows, MS-DOS, OS/2). В этом отношении сетевую ОС NetWare трудно отнести к корпоративным продуктам, так как в ней отсутствуют почти все требования, предъявляемые к серверу приложений. В то же время хорошая поддержка универсальных приложений в Windows NT собственно и позволяет ей претендовать на место в мире корпоративных продуктов.

Справочная служба. Корпоративная ОС должна обладать способностью хранить информацию обо всех пользователях и ресурсах таким образом, чтобы обеспечивалось управление ею из одной центральной точки. Подобно большой организации, корпоративная сеть нуждается в централизованном хранении как можно более полной справочной информации о самой себе (начиная с данных о пользователях, серверах, рабочих станциях и кончая данными о кабельной системе). Естественно организовать эту информацию в виде базы данных. Данные из этой базы могут быть востребованы многими сетевыми системными приложениями, в первую очередь системами управления и администрирования. Кроме этого, такая база полезна при организации электронной почты, систем коллективной работы, службы безопасности, службы инвентаризации программного и аппаратного обеспечения сети, да и для практически любого крупного бизнес-приложения.

База данных, хранящая справочную информацию, предоставляет все то же многообразие возможностей и порождает все то же множество проблем, что и любая другая крупная база данных. Она позволяет осуществлять различные операции поиска, сортировки, модификации и т.п., что очень силь-

но облегчает жизнь как администраторам, так и пользователям. Но за эти удобства приходится расплачиваться решением проблем распределенности, репликации и синхронизации.

В идеале сетевая справочная информация должна быть реализована в виде единой базы данных, а не представлять собой набор баз данных, специализирующихся на хранении информации того или иного вида, как это часто бывает в реальных операционных системах. Например, в Windows NT имеется, по крайней мере, пять различных типов справочных баз данных. Главный справочник домена (NT Domain Directory Service) хранит информацию о пользователях, которая используется при организации их логического входа в сеть. Данные о тех же пользователях могут содержаться и в другом справочнике, используемом электронной почтой Microsoft Mail. Еще три базы данных поддерживают разрешение низкоуровневых адресов: WINS – устанавливает соответствие Netbios-имен IP-адресам, справочник DNS – сервер имен домена – оказывается полезным при подключении NT-сети к Internet, и наконец, справочник протокола DHCP используется для автоматического назначения IP-адресов компьютерам сети. Ближе к идеалу находятся справочные службы, поставляемые фирмой Banyan (продукт Streettalk III) и фирмой Novell (NetWare Directory Services), предлагающие единый справочник для всех сетевых приложений. Наличие единой справочной службы для сетевой операционной системы – один из важнейших признаков ее корпоративности.

Безопасность. Особую важность для ОС корпоративной сети приобретают вопросы безопасности данных. С одной стороны, в крупномасштабной сети объективно существует больше возможностей для несанкционированного доступа – из-за децентрализации данных и большой распределенности «законных» точек доступа, из-за большого числа пользователей, благонадежность которых трудно установить, а также из-за большого числа возможных точек несанкционированного подключения к сети. С другой стороны, корпоративные бизнес-приложения работают с данными, которые имеют жизненно важное значение для успешной работы корпорации в целом. И для защиты таких данных в корпоративных сетях наряду с различными аппаратными средствами используется весь спектр средств защиты, предоставляемый операционной системой: избирательные или мандатные права доступа, сложные процедуры аутентификации пользователей, программная шифрация.

Назначение ОС

- Обеспечение максимальной эффективности системы.
  - Загрузка устройств (распараллеливание CPU и ввода/вывода).
  - Обеспечение высокой пропускной способности.
  - Минимизация времени ответа.

- Обеспечение максимальных удобств пользователю.
  - Системы программирования.
  - Ввод/вывод –> файлы –> базы данных.
  - Сервисные программы.
  - Работа с библиотеками.
  - Средства секционирования и объединения программ.
  - Диагностика.
- Обеспечение максимальной надёжности.
  - Организация защиты.
  - Обработка сбоев.
  - Средства восстановления при отказах.
  - Контрольные точки.

Требования к аппаратуре

- Способы адресации.
- Аппаратная организация защиты (интервалы адресов, ключи).
- Наличие привилегированных команд и состояний супервизор/задача.
- Наличие аппаратных прерываний.

Понятия программы процесса

Всем пользователям компьютеров знакомо понятие «программа». Во время выполнения программы различают три объекта:

1) процедура – последовательность команд, которая определяет программу;

2) процессор – агент, выполняющий процедуру;

3) среда – та часть окружающего мира, которую процессор может воспринимать и изменять.

Свойства программы:

1) операции выполняются последовательно;

2) среда полностью находится под управлением программы и изменяется только в результате шагов, выполняемых программой;

3) между шагами программа может быть прервана.

Из данного определения следует, что *программа – замкнутая система. Такие программы не могут быть операционными системами,* так как:

- предполагается, что ОС эффективно использует все компоненты компьютера, то есть операции должны совмещаться;
- система должна отвечать на запросы за определенный промежуток времени, но моменты поступления запросов непредсказуемы и не управляются системой.

Процесс во многом подобен однократному выполнению программы. Он состоит из серии несовмещенных шагов, каждый из которых воспринимает и изменяет среду определенным образом. Обычно он управляется про-

цедурой, которая сама является частью среды (поскольку хранится в ней). Процесс может быть безопасно прерван между шагами и во время его выполнения может произойти смена процессора. Существенная разница между процессом и программой – в том, что *процесс не является замкнутой системой*. Он может взаимодействовать с другими процессами либо явно, либо неявно, изменяя и воспринимая часть среды, которую он разделяет с другими процессами.

Операционная система представляет собой *совокупность многочисленных взаимодействующих процессов*. В вычислительных системах одновременно действуют многочисленные процессы.

Управление локальными ресурсами

Важнейшей функцией операционной системы является организация рационального использования всех аппаратных и программных ресурсов системы. К основным ресурсам могут быть отнесены: процессоры, память, внешние устройства, данные и программы. Располагающая одними и теми же ресурсами, но управляемая различными ОС, вычислительная система может работать с разной степенью эффективности. Поэтому знание внутренних механизмов операционной системы позволяет косвенно судить о её эксплуатационных возможностях и характеристиках.

Управление процессами

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами. Программный процесс однозначно характеризуется информационной структурой, называемой *дескриптором* процесса или вектором состояния, содержащим идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другую информацию. В дескрипторе может быть предусмотрено место для организации общения с другими процессами.

Состояние процессов

В многозадачной (многопроцессной) системе процесс может находиться в одном из трех основных состояний:

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

•ОЖИДАНИЕ – пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса;

•ГОТОВНОСТЬ – также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке 1.

В состоянии ВЫПОЛНЕНИЕ в однопроцессорной системе может находиться только один процесс, а все остальные – в одном из состояний ОЖИДАНИЕ и ГОТОВНОСТЬ. Жизненный цикл процесса начинается с состояния ГОТОВНОСТЬ, когда процесс готов к выполнению и ждет своей очереди получения процессорного времени. При активизации процесс переходит в состояние ВЫПОЛНЕНИЕ и находится в нем до тех пор, пока он либо завершится, либо перейдет в состояние ОЖИДАНИЕ какого-нибудь события, либо опять в состояние ГОТОВНОСТЬ, если процесс исчерпал отведенное ему процессорное время.

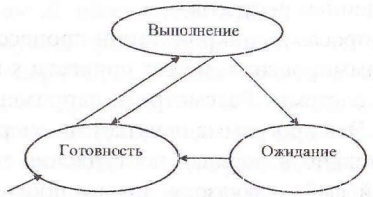


Рис. 1. Граф состояний процесса в многозадачной среде

Контекст и дескриптор процесса

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды отображается состоянием регистров и программного счетчика, режимом работы процессора, указателями на от-

крытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т.д. Эта информация называется *контекстом процесса*.

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

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Программный код только тогда начнет выполняться, когда для него операционной системой будет создан процесс. Создать процесс – это значит:

1. Создать информационные структуры, описывающие данный процесс, то есть его дескриптор и контекст.
2. Включить дескриптор нового процесса в очередь готовых процессов.
3. Загрузить кодовый сегмент процесса в оперативную память или в область свопинга.

Средства синхронизации и взаимодействия процессов

Проблема синхронизации

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

Пренебрежение вопросами синхронизации процессов, выполняющихся в режиме мультипрограммирования, может привести к их неправильной работе или даже к краху системы. Рассмотрим, например, программу печати файлов (принт-сервер). Эта программа печатает по очереди все файлы, имена которых последовательно в порядке поступления записывают в специальный общедоступный файл «заказов» другие программы. Особая переменная NEXT, также доступная всем процессам-клиентам, содержит номер первой свободной для записи имени файла позиции файла «заказов». Процессы-клиенты читают эту переменную, записывают в соответствующую позицию файла «заказов» имя своего файла и наращивают значение NEXT на единицу. Предположим, что в некоторый момент процесс R решил распечатать свой файл, для этого он прочитал значение переменной NEXT, значение которой для определенности предположим равным 4. Процесс запомнил это значение, но поместить имя файла не успел, так как его выполнение было прервано (например, вследствие исчерпания кванта). Очередной

процесс S, желающий распечатать файл, прочитал то же самое значение переменной NEXT, поместил в четвертую позицию имя своего файла и нарастил значение переменной на единицу. Когда в очередной раз управление будет передано процессу R, то он, продолжая свое выполнение, в полном соответствии со значением текущей свободной позиции, полученным во время предыдущей итерации, запишет имя файла также в позицию 4, поверх имени файла процесса S.

Таким образом, процесс S никогда не увидит свой файл распечатанным. Сложность проблемы синхронизации состоит в нерегулярности возникающих ситуаций: в предыдущем примере можно представить и другое развитие событий: были потеряны файлы нескольких процессов или, напротив, не был потерян ни один файл. В данном случае все определяется взаимными скоростями процессов и моментами их прерывания. Поэтому отладка взаимодействующих процессов является сложной задачей. Ситуации подобные той, когда два или более процессов обрабатывают разделяемые данные, и конечный результат зависит от соотношения скоростей процессов, называются **гонками** («Race conditions»).

Критическая секция

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

Простейший способ обеспечить взаимное исключение – позволить процессу, находящемуся в критической секции, запрещать все прерывания. Однако этот способ непригоден, так как опасно доверять управление системой пользовательскому процессу; он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому, что прерывания никогда не будут разрешены.

Другим способом является использование блокирующих операций. Примером такой операции является операция, условно названная «проверка и установка». Она использует два параметра: *локальный* и *общий* и выполняется следующим образом. Значение блокирующей переменной *общий* равно 1, когда критический ресурс свободен. Следует заметить, что «операция проверки и установки» блокирующей переменной должна быть неделимой. Поясним это. Пусть в результате проверки переменной процесс определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой процесс занял ресурс, вошел в свою критическую секцию, но также был прерван, не

завершив работы с разделяемым ресурсом. Когда управление было возвращено первому процессу, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен принцип взаимного исключения, что потенциально может привести к нежелаемым последствиям. Во избежание таких ситуаций в системе команд машины желательно иметь единую команду «проверка-установка», или же реализовывать системными средствами соответствующие программные примитивы, которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Особенность операции «проверка и установка» в том, что оба действия выполняются как один шаг, и между ними процесс не может быть прерван. Следующий пример показывает, синхронизацию двух процессов с использованием операции «проверка и установка».

```
program Example_Test_Set;
var
comm: integer;
loc1,loc2:integer;
begin
   comm:=0;
 parbegin
   process1:
      while(true) do
        begin
           repeat
             Test_Set(loc1,comm);
           until loc1=0;
           Critical_section1;
           comm:=0;
        end;
   process2:
      while(true) do
        begin
           repeat
             Test_Set(loc2,comm);
           until loc2=0;
           Critical_section2;
           comm:=0;
        end;
 parend;
end.
```

Здесь для записи параллельного алгоритма (предполагается, что процессы выполняются параллельно) используется паскаль, дополненный операторными скобками **parbegin parend**, обозначающими, что блоки внутри скобок выполняются параллельно (в данном примере process1 и process2). Операция «проверка и установка» названа Test_Set. Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется.

Реализация критических секций с использованием блокирующих переменных имеет существенный недостаток: в течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, бесполезно тратя процессорное время. Для устранения таких ситуаций может быть использован так называемый аппарат событий. С помощью этого средства могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов. В разных операционных системах аппарат событий реализуется по-своему, но в любом случае используются системные функции аналогичного назначения, которые условно назовем WAIT(x) и POST(x), где x – идентификатор некоторого события. Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию WAIT(D), здесь D обозначает событие, заключающееся в освобождении ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе о том, что процесс ожидает события D. Процесс, который в это время использует ресурс D, после выхода из критической секции выполняет системную функцию POST(D), в результате чего операционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D, в состояние ГОТОВНОСТЬ.

Обобщающее средство синхронизации процессов предложил Дейкстра, который ввел два новых примитива. В абстрактной форме эти примитивы, обозначаемые P и V, оперируют над целыми неотрицательными переменными, называемыми ***семафорами***. Пусть S такой семафор. Операции определяются следующим образом:

- V(S): переменная S увеличивается на 1 одним неделимым действием; выборка, инкремент и запоминание не могут быть прерваны, и к S нет доступа другим процессам во время выполнения этой операции;
- P(S): уменьшение S на 1, если это возможно. Если S = 0, то невозможно уменьшить S и остаться в области целых неотрицательных значений, в этом случае процесс, вызывающий P-операцию, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также является неделимой операцией.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную. Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания, в то время как V-операция может при некото-

рых обстоятельствах активизировать другой процесс, приостановленный операцией P (сравните эти операции с системными функциями WAIT и POST).

Рассмотрим использование семафоров на классическом примере взаимодействия двух процессов, выполняющихся в режиме мультипрограммирования. Каждый из процессов перед входом в критическую секцию выполняет операцию P(S), а при завершении – операцию V(S).

```
program Example_Semaphore;
var
    S:integer;
begin
  S:=1;
  parbegin
    process1:
    while(true) do
      begin
        P(S);
        Critical_section1;
        V(S);
      end;
    process2:
      while(true) do
        begin
          P(S);
          Critical_section2;
          V(S);
        end;
  parend;
end.
```

Следующий пример – задача «поставщик – потребитель». Поставщик помещает данные в буферный пул, а потребитель считывает их из буферного пула. Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. При реализации этой задачи необходимо учитывать следующие проблемы. Процессы «поставщик» и «потребитель» развиваются независимо друг от друга, каждый со своей скоростью. Процесс «поставщик» не может положить сообщение, если нет свободных мест и должен быть приостановлен до тех пор, пока не освободится хотя бы один буфер. Напротив, процесс «потребитель» приостанавливается, когда буферный пул не содержит ни одного сообщения, и активизируется при появлении хотя бы одной записи. Кроме того, необходимо обеспечить взаимоисключение при обращении к буферному пулу.

Введем два семафора: *presence* – число пустых буферов и *filled* – число заполненных буферов. Предположим, что запись в буфер и считывание из буфера являются критическими секциями (как в примере с принт-сервером в начале данного раздела). Введем также двоичный семафор *excep*, используемый для обеспечения взаимного исключения. Тогда процессы могут быть описаны следующим образом:

```
program Supplier_Consumer;
var
 presence:integer;// количество свободных буферов
 filled:integer;  //количество заполненных буферов
 excep:integer;   //семафор взаимоисключения
begin
   presence:=quantity_of_buffers;// сначала свободны все буфера
   filled:=0;//заполненных пока нет
   excep:=1;
 parbegin
    Supplier://поставщик
            while(true) do
              begin
                Prepare_mes;//подготовить сообщение
                P(presence);//проверить наличие свободных мест в
                //буфере
                    P(excep);   //захватить буферный пул
                      Send_mes; //положить сообщение
                V(filled); //увеличить количество занятых буферов
                    V(excep);   //освободить буферный пул
              end;
    Consumer://потребитель
            while(true) do
              begin
                P(filled);//проверить наличие заполненных буферов
                  P(excep); //захватить буферный пул
                    Receive_mes;//получить сообщение
                V(presence);// увеличить количество свободных
                буферов
                  V(excep); //освободить буферный пул
                Treat_mes;  //обработать сообщение
              end;
 parend;
end.
```

Реализация семафоров и мьютексов в Windows

В операционной системе Windows семафор создается функцией

```
function CreateSemaphore(Attr:Pointer;
    //указатель на атрибуты безопасности InitialCount:longInt;
    //начальное значение MaxCount:LongInt;
    //максимальное значениеlpName:PChar
    //указатель на имя семафора):THandel;
    //дескриптор семафора
```

Частный случай семафора – двоичный семафор – мьютекс (MUTual EXclusions – взаимоисключение) создается функцией

```
function CreateMutex(lpMutexAttributes:PSecurityAttributes;
//указатель на атрибуты безопасностиbInitialOwner:Boolean;
//true - поток, создающий мьютекс является его владельцем;
//false - мьютекс не имеет владельца lpName:PChar
//указатель на имя мьютекса; nil - у мьютекса нет имени;
//если значение отлично от nil, то в системе будет выполнен поиск
//мьютекса с таким именем; при успешном завершении поиска
//функция вернёт дескриптор найденного мьютекса, иначе
//дескриптор нового мьютекса
):THandle;stdcall; // дескриптор мьютекса
```

Для доступа к объекту (семафору или мьютексу) используется функция

```
function WaitForSignaleObject(hHandle:THandle;
dwMilliseconds:DWORD):DWORD;stdcall;
```

Текущий поток переводится в состояние ожидания пока объект, заданный параметром *hHandle*, не станет доступным, при этом ожидание может продлиться до истечения интервала, заданного в миллисекундах параметром *dwMilliseconds*:

если *dwMilliseconds*=0, то проверка состояния и немедленный возврат;

если *dwMilliseconds*=*INFINITE*, то ожидание продолжается до тех пор, пока объект не станет доступным.

Возвращаемый функцией *WaitForSignaleObject* результат – константа типа DWORD, имеющая одно из следующих значений:

WAIT_ABANDONED – заданный объект является объектом мьютекса, но поток, владевший им, был завершен до его освобождения, мьютекс считается покинутым и право собственности на него передается вызывающему потоку, мьютекс определяется как недоступный.

WAIT_OBJECT_0 – состояние заданного объекта определяется как доступное.

WAIT_TIMEOUT – установленный интервал времени истек, но состояние объекта определяется как недоступное.

Для доступа к нескольким объектам используется функция

```
function WaitForMultipleObjects(cCount:Cardinal; // количество
объектов
    lpHandles:PWOHandelArray; // массив дескрипторов объектов
    bWaitAll:LongBool; //true, если требуется освобождение всех
объектов, //false, если достаточно освобождения хотя бы //одного
объекта
    dwMilliseconds:DWORD // время ожидания
    ):DWORD; stdcall;
```

После завершения работы с совместно используемым ресурсом мьютекс освобождается

```
function ReleaseMutex(hHandle:THandle ):Boolean;
```

hHandle – дескриптор мьютекса.

Право собственности на мьютекс у потока отбирается. Мьютекс становится доступным.

После завершения работы с совместно используемым ресурсом семафор освобождается с помощью функции

```
function ReleaseSemaphore (hHandle:THandle  // дескриптор семафора
    lReleaseCount:LongInt;//приращение счётчика
    lpPreviousCount:Pointer//если неравно nil, то старое
//значение будет сохранено в lpPreviousCount.
    ):Boolean;
```

При завершении работы с семафором или мьютексом необходимо выполнить

```
function CloseHandle(hHandle:THandle):Boolean;
```

Тупики

Неаккуратное использование семафоров может привести к взаимной блокировке, называемой также дедлоком (deadlock), клинчем (clinch) или тупиком. Эту проблему синхронизации можно проиллюстрировать на примере уже описанной выше задачи «поставщик – потребитель». Переставим местами операции P(presence) и P(excep) в процессе «поставщик» и предположим, что буферный пул заполнен, а «поставщик» пытается положить следующее сообщение. Выполнив P(excep), он «захватывает» критический ре-

сурс, но поскольку свободных мест нет, то при выполнении P(presence) «поставщик» будет заблокирован до тех пор, пока «потребитель» не заберет хотя бы одно сообщение. Однако, «потребитель» не может этого сделать, так как для этого необходимо войти в критическую секцию, вход в которую заблокирован процессом «поставщик».

Рассмотрим еще один пример тупика. Пусть двум процессам, выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например, принтер и диск. И пусть после того, как процесс А занял принтер (установил блокирующую переменную), он был прерван. Управление получил процесс В, который сначала занял диск, но при выполнении следующей команды был заблокирован, так как принтер оказался уже занятым процессом А. Управление снова получил процесс А, который в соответствии со своей программой сделал попытку занять диск и был заблокирован: диск уже распределен процессу В. В таком положении процессы А и В могут находиться сколь угодно долго.

В зависимости от соотношения скоростей процессов, они могут либо совершенно независимо использовать разделяемые ресурсы, либо образовывать очереди к разделяемым ресурсам, либо взаимно блокировать друг друга. Тупиковые ситуации надо отличать от простых очередей, хотя и те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: процесс приостанавливается и ждет освобождения ресурса. Однако очередь – это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Она возникает тогда, когда ресурс недоступен в данный момент, но через некоторое время он освобождается, и процесс продолжает свое выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией.

В рассмотренных примерах тупик был образован двумя процессами, но взаимно блокировать друг друга могут и большее число процессов.

Проблема тупиков включает в себя следующие задачи:

- предотвращение тупиков,
- распознавание тупиков,
- восстановление системы после тупиков.

Тупики могут быть предотвращены на стадии написания программ, то есть программы должны быть написаны таким образом, чтобы тупик не мог возникнуть ни при каком соотношении взаимных скоростей процессов. Так, если бы в предыдущем примере процесс А и процесс В запрашивали ресурсы в одинаковой последовательности, то тупик был бы в принципе невозможен. Второй подход к предотвращению тупиков называется динамическим и заключается в использовании определенных правил при назначении ресурсов процессам, например, ресурсы могут выделяться в определенной последовательности, общей для всех процессов.

В некоторых случаях, когда тупиковая ситуация образована многими процессами, использующими много ресурсов, распознавание тупика является нетривиальной задачей. Существуют формальные, программно-реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки.

Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные процессы. Можно снять только часть из них, при этом освобождаются ресурсы, ожидаемые остальными процессами, можно вернуть некоторые процессы в область свопинга, можно совершить «откат» некоторых процессов до так называемой контрольной точки, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в местах, после которых возможно возникновение тупика.

Из всего вышесказанного ясно, что использовать семафоры нужно очень осторожно, так как одна незначительная ошибка может привести к остановке системы. Для того чтобы облегчить написание корректных программ, было предложено высокоуровневое средство синхронизации, называемое монитором. **Монитор** – это набор процедур, переменных и структур данных. Процессы могут вызывать процедуры монитора, но не имеют доступа к внутренним данным монитора. Мониторы имеют важное свойство, которое делает их полезными для достижения взаимного исключения: ***только один процесс может быть активным по отношению к монитору***. Компилятор обрабатывает вызовы процедур монитора особым образом. Обычно, когда процесс вызывает процедуру монитора, то первые несколько инструкций этой процедуры проверяют, не активен ли какой-либо другой процесс по отношению к этому монитору. Если да, то вызывающий процесс приостанавливается, пока другой процесс не освободит монитор. Таким образом, исключение входа нескольких процессов в монитор реализуется не программистом, а компилятором, что делает ошибки менее вероятными. В виде монитора можно реализовать описанные выше операции P(S) и V(S).

```
monitor Bin_Semaph;
  var
    integer S;
    condition Semaphore_Positive;//тип, определяющий
//условие для продолжения работы
  procedure P;
  begin
   if S<1 then Semaphore_Positive.Wait;//Wait –
//специальная операция в мониторе для
                    //блокировки процесса.
```

```
S:=S-1;
  end;
  procedure V;
  begin
 S:=S+1;
if S=1 then Semaphore_Positive.Signal;//Signal -
//специальная операция в мониторе для разблокировки процесса.
  end;
begin
  S:=1;// инициализация семафора
end.
```

Взаимное исключение критических участков при помощи монитора

```
parbegin
  P1:while true do
 begin
  Bin_Semaph.P;
  //критический участок 1
  Bin_Semaph.V;
end;
  P2:while true do
begin
  Bin_Semaph.P;
  //критический участок 2
  Bin_Semaph.V;
end;
parend
```

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, семафоры и мониторы оказываются непригодными. В таких системах синхронизация может быть реализована только с помощью обмена сообщениями.

Нити

Многозадачность является важнейшим свойством ОС. Для поддержки этого свойства ОС определяет и оформляет для себя те внутренние единицы работы, между которыми и будет разделяться процессор и другие ресурсы компьютера. Эти внутренние единицы работы в разных ОС носят разные названия – задача, задание, процесс, нить. В некоторых случаях сущности, обозначаемые этими понятиями, принципиально отличаются друг от друга.

Говоря о процессах, мы отмечали, что операционная система поддерживает их обособленность: у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы – файлы, окна, семафоры и т.д. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы машины, конкурируют друг с другом. В общем случае процессы принадлежат разным пользователям, разделяющим один компьютер, и ОС берет на себя роль арбитра в спорах процессов за ресурсы.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса). Однако задача, решаемая в рамках одного процесса, может обладать внутренним параллелизмом, который в принципе позволяет ускорить ее решение. Например, в ходе выполнения задачи происходит обращение к внешнему устройству, и на время этой операции можно не блокировать полностью выполнение процесса, а продолжить вычисления по другой «ветви» процесса.

Для этих целей современные ОС предлагают использовать сравнительно новый механизм многонитевой обработки (multithreading). При этом вводится новое понятие «нить» (thread), а понятие «процесс» в значительной степени меняет смысл.

Мультипрограммирование теперь реализуется на уровне нитей, и задача, оформленная в виде нескольких нитей в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей. Например, если электронная таблица была разработана с учетом возможностей многонитевой обработки, то пользователь может запросить пересчет своего рабочего листа и одновременно продолжать заполнять таблицу. Особенно эффективно можно использовать многонитевость для выполнения распределенных приложений, например, многонитевый сервер может параллельно выполнять запросы сразу нескольких клиентов.

Нити, относящиеся к одному процессу, не настолько изолированы друг от друга, как процессы в традиционной многозадачной системе, между ними легко организовать тесное взаимодействие. Действительно, в отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все нити одного процесса всегда принадлежат одному приложению, поэтому программист, пишущий это приложение, может заранее продумать работу множества нитей процесса таким образом, чтобы они могли взаимодействовать, а не бороться за ресурсы.

В традиционных ОС понятие «нить» тождественно понятию «процесс». В действительности часто бывает желательно иметь несколько нитей, разделяющих единое адресное пространство, но выполняющихся квазипараллельно, благодаря чему нити становятся подобными процессам (за исключением разделяемого адресного пространства).

Нити иногда называют облегченными процессами или мини-процессами. Действительно, нити во многих отношениях подобны процессам. Каждая нить выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Нити, как и процессы, могут, например, порождать нити-потомки, могут переходить из состояния в состояние. Подобно традиционным процессам (то есть процессам, состоящим из одной нити), нити могут находиться в одном из следующих состояний: ВЫПОЛНЕНИЕ, ОЖИДАНИЕ и ГОТОВНОСТЬ. Пока одна нить заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так, как это делают процессы, в соответствии с различными вариантами планирования.

Однако различные нити в рамках одного процесса не настолько независимы, как отдельные процессы. Все такие нити имеют одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждая нить может иметь доступ к каждому виртуальному адресу, одна нить может использовать стек другой нити. Между нитями нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Все нити одного процесса всегда решают общую задачу одного пользователя, и аппарат нитей используется здесь для более быстрого решения задачи путем ее распараллеливания. При этом программисту очень важно получить в свое распоряжение удобные средства организации взаимодействия частей одной задачи. Кроме разделения адресного пространства, все нити разделяют также набор открытых файлов, таймеров, сигналов и т.п.

Итак, нити имеют собственные:

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

Нити разделяют:

- адресное пространство,
- глобальные переменные,
- открытые файлы,
- таймеры,
- семафоры,

- статистическую информацию.

Многонитевая обработка повышает эффективность работы системы по сравнению с многозадачной обработкой. Например, в многозадачной среде Windows можно одновременно работать с электронной таблицей и текстовым редактором. Однако если пользователь запрашивает пересчет своего рабочего листа, электронная таблица блокируется до тех пор, пока эта операция не завершится, что может потребовать значительного времени. В многонитевой среде в случае, если электронная таблица была разработана с учетом возможностей многонитевой обработки, предоставляемых программисту, этой проблемы не возникает, и пользователь всегда имеет доступ к электронной таблице.

Широкое применение находит многонитевая обработка в распределенных системах.

Существуют две модели применения потоков:

асимметричная – потоки решают разные задачи и, как правило, не разделяют ресурсы;

симметричная – потоки выполняют одну и ту же задачу, разделяют ресурсы и используют один и тот же код.

Некоторые прикладные задачи легче программировать, используя параллелизм, например задачи типа «писатель-читатель», в которых одна нить выполняет запись в буфер, а другая считывает записи из него. Поскольку они разделяют общий буфер, не стоит их делать отдельными процессами. Другой пример использования нитей – это управление сигналами, такими как прерывание с клавиатуры (del или break). Вместо обработки сигнала прерывания, одна нить назначается для постоянного ожидания поступления сигналов. Таким образом, использование нитей может сократить необходимость в прерываниях пользовательского уровня. В этих примерах не столь важно параллельное выполнение, сколь важна ясность программы.

Наконец, в мультипроцессорных системах для нитей из одного адресного пространства имеется возможность выполняться параллельно на разных процессорах. Это действительно один из главных путей реализации разделения ресурсов в таких системах. С другой стороны, правильно сконструированные программы, которые используют нити, должны работать одинаково хорошо как на однопроцессорной машине в режиме разделения времени между нитями, так и на настоящем мультипроцессоре.

Реализация потоков в Windows

Использование библиотеки API Win32 (Application Program Interface) – наиболее мощный и универсальный способ работы с потоками.

Поток создаётся с помощью функции API

```
function CreateThread (Attr: Pointer;//адрес атрибутов
//безопасности
```

```
Stack:Dword;//размер стека для потока
Start:Pointer;//начальный адрес потока
par:Pointer;//аргументы потока
flag:Dword;// флаг создания
var ID:Dword //идентификатор потока
):THandle;//дескриптор потока
```

Attr = nil, т.е. атрибуты безопасности по умолчанию.

Stack – задаёт размер стека для потока. Если *Stack*=0, то размер совпадает с размером стека основного приложения.

Start – основной. Он задаёт адрес функции, вызываемой при запуске потока. Она обязана возвращать результат типа Longint, иметь один параметр типа Pointer. При её описании указывается атрибут stdcall.

Par – указатель на структуру, содержащую аргументы. Если Par=nil, то аргументов нет.

Flag=0 – поток сразу начнёт работу, *Flag*=CREATE_SUSPEND – поток начнёт работу после вызова функции ResumeThread.

Другие функции доя работы с потоками:

`function ResumeThread (hThread:THandle):Dword;` – возобновить работу потока после приостановки.

hThread – дескриптор потока, полученный при создании.

`function SuspendThread (hThread:THandle):Dword;` – приостановить поток.

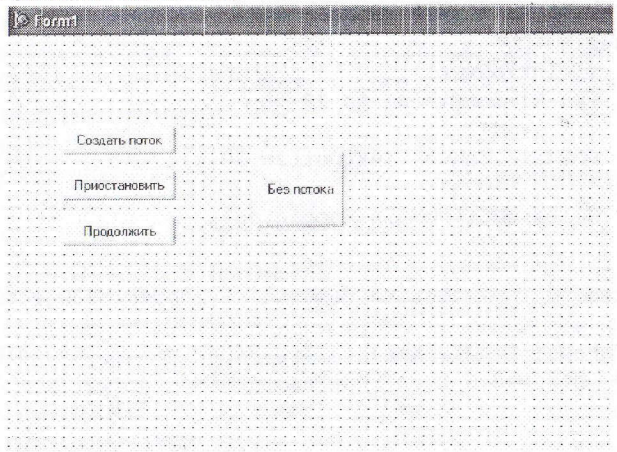
`function CloseHandle (hThread:THandle):Dword;` – срочно завершает поток после приостановки.

Примеры создания потоков в среде Delphi

Любой процесс, запускаемый в Windows, состоит, как минимум, из одного главного потока. Этот поток выполняет и требуемую обработку информации, и осуществляет реакцию на внешние события, например, нажатие кнопок клавиатуры или мыши. Если собственная работа приложения оказывается долгой, приложение не отвечает на запросы, как говорят «виснет». Чтобы этого не происходило, желательно длительные вычисления запускать в отдельном потоке. В этом случае время, отводимое процессу, разделяется между его потоками. Главный поток отвечает на внешние события, а потоки выполняют соответствующую работу.

В приведенном примере функция func выполняет длительное вычисление и запускается либо в отдельном потоке – кнопкой «Создать поток», либо в главном потоке – кнопкой «Без потока». В первом случае можно приостановить вычисления, нажав кнопку «Приостановить» или их возобно-

вить кнопкой «Продолжить», можно переместить или закрыть форму. Во втором случае до завершения вычислений все попытки выполнения подобных действий остаются без ответа.




```
unit UMain;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls;
type
  TFrmMain = class(TForm)
    BtnCreate: TButton;
    BtnSusp: TButton;
    BtnRes: TButton;
    BtnWithout: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  end;
var
  FrmMain: TFrmMain;
implementation
 var hThread:THandle;//дескриптор потока
     ThreadID:DWord; //идентификатор потока
{$R *.dfm}
function func (p:pointer):LongInt;stdcall;
var
    i,k:integer;
    dc:hDc;//дескриптор контекста графического устройства
```

```
    s:string[10];
begin
   dc:=GetDc(FrmMain.Handle);//получаем контекст формы
   for i:=0 to 1000000 do
   begin
     s:=IntToStr(i);
     for k:=length(s) to 10 do
         s:=s+' ';
     FrmMain.Canvas.TextOut(20,20,s);
   end;
     ReleaseDC(FrmMain.Handle,dc);
end;

procedure TFrmMain.BtnCreateClick(Sender: TObject);
begin
  hThread:=CreateThread(nil,0,@func,nil,0,ThreadID);
  if  hThread=0 then ShowMessage('No Thread');
end;

procedure TFrmMain.BtnSuspClick(Sender: TObject);
begin
  SuspendThread(hThread);
end;

procedure TFrmMain.BtnResClick(Sender: TObject);
begin
  ResumeThread(hThread);
end;
procedure TFrmMain.BtnWithoutClick(Sender: TObject);
begin
   Func(nil);
end;
end.
```

Рассмотрим основные методы и свойства этого класса.

constructor Create(CreateSuspend:boolean); создает поток.

Если аргумент = False, то созданный поток сразу начинает выполнение – управление передается методу Execute.

Если аргумент = True, то поток ожидает вызова метода Resume.

destructor Destroy:override; – завершает поток и освобождает все ресурсы, им занятые. Вызывается автоматически при завершении метода Execute.

procedure Resume; возобновляет поток после приостановки.

property Suspended:boolean; – при записи True/False приостанавливает/возобновляет поток. При чтении показывает, приостановлен ли поток.

function Terminate:integer; – окончательно завершает поток без последующего запуска. Он автоматически вызывается из деструктора.

function WaitFor:integer; – предназначена для синхронизации и позволяет одному потоку дождаться, когда завершится другой поток.

Если внутри потока с именем FirstThread записана команда Code:=SecondThread.WaitFor, то поток FirstThread останавливается до завершения потока SecondThread.

property Handle:THandle; – дескриптор потока.

property ThreadID:THandle read FThreadID; – идентификатор потока.

Свойства Handle и ThreadID дают возможность непосредственного доступа к потоку средствами Win32 API.

property Priority:TThreadPriority; – приоритет потока:

tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical.

procedure Synchronize(Method:TThreadMethod); метод относится к секции protected, т.е. может быть вызван только из потомков TThread. Используется для безопасного вызова методов VCL внутри потоков. Метод не должен иметь параметров и не должен возвращать никакого значения.

procedure Execute;virtual;abstract; главный метод класса. Обязательно переопределяется. Метод является абстрактным, поэтому нельзя создать экземпляр TThread, необходимо создать потомка класса TThread и перекрыть в нем метод Execute.

property ReturnValue:integer; код завершения потока. По умолчанию ноль. Другие значения могут быть присвоены внутри потока.

property OnTerminate:TNotifyEvent; событие, происходящее после завершения метода Execute, но перед Destroy.

Рассмотрим простой пример создания потока с помощью класса TThread.

Приложение будет содержать два потока основной и дополнительный. Дополнительный поток осуществляет вычисления, не мешая работе главного потока.

Создадим новое приложение. Его форма содержит компонент TEdit, в котором будут отражаться результаты вычислений, и две кнопки – «Выполнить» для запуска потока и «Остановить» для его остановки. Для создания класса типа TTread можно воспользоваться шаблоном, предоставляемым Delphi в меню File->New->Other. Выберем значок Thread Object (рис. 3). Будет создан модуль, содержащий шаблон класса Thread. Как уже было сказано выше, в нем обязательно должен быть переопределен метод Execute. Ниже приведены тексты обоих модулей.

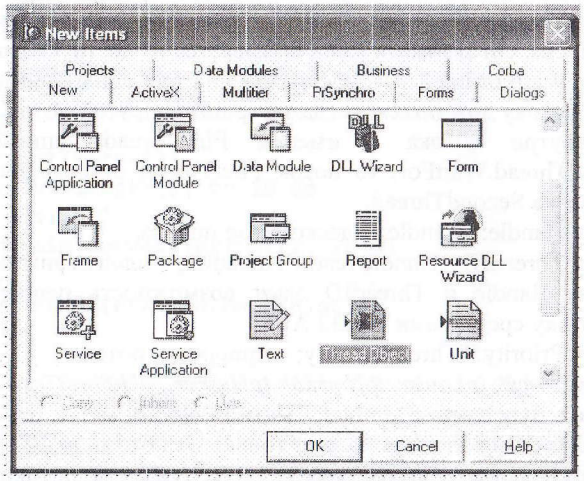


Рис. 3. Выбор меню для создания потока

```
unit USynchro;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms,   Dialogs, StdCtrls,UThr;
type
  TFrmSynch = class(TForm)
    EdtSynch: TEdit;
    BtnRun: TButton;
    BtnStop: TButton;
    procedure BtnRunClick(Sender: TObject);
    procedure BtnStopClick(Sender: TObject);
  end;
var
  FrmSynch: TFrmSynch;
  Thread:TSimpleThread;
implementation

{$R *.dfm}
procedure TFrmSynch.BtnRunClick(Sender: TObject);
begin
   Thread:=TSimpleThread.Create(false);
   Thread.Count:=0
end;
procedure TFrmSynch.BtnStopClick(Sender: TObject);
begin
```



```
    Thread.Terminate;
end;
end.
unit UThr;
interface
uses Classes;
type
  TSimpleThread = class(TThread)
  protected
    procedure Execute; override;
  public
     Count:integer;
     procedure OutMessage;
  end;
implementation
uses USynchro,SysUtils;
procedure TSimpleThread.Execute;
begin
  while not Terminated do
  // пока не нажата кнопка остановить
  begin
     Count:=Count+1;
     Sleep(1000);
//вывод на форму из потока осуществляется процедурой
//OutMessage через метод Synchronize, т.к. работа
//визуальных компонентов в дополнительных потоках неустойчива.
     Synchronize(OutMessage);
  end;
end;
procedure TSimpleThread.OutMessage;
begin
  FrmSynch.EdtSynch.Text:=IntToStr(Count);
end;
end.
```

Второй пример многопоточного приложения содержит три потока: главный и два дополнительных. Дополнительные потоки будут выполнять некоторые простые вычисления, а главная программа – отображать, сколько вычислений в секунду выполнено в каждом потоке. Форма приложения содержит два регулятора для установки приоритетов потоков и две строки редактирования – для наблюдения за их изменением. Ниже приведены тексты обоих модулей.

```
unit UMultyThread;
interface
```

```
  uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, ComCtrls, ExtCtrls, StdCtrls,UThr;
  type
    TFrmMain = class(TForm)
      Edit1: TEdit;
      Edit2: TEdit;
      Label1: TLabel;
      Label2: TLabel;
      Timer: TTimer;
      TrackBar1: TTrackBar;
      TrackBar2: TTrackBar;
      procedure FormCreate(Sender: TObject);
      procedure Timer1Timer(Sender: TObject);
      procedure TrackBar1Change(Sender: TObject);
    public
      Thread1,Thread2:TSimpleThread;
    end;
  var
    FrmMain: TFrmMain;
  implementation
  {$R *.dfm}
  procedure TFrmMain.FormCreate(Sender: TObject);
  begin
      Thread1:=TSimpleThread.Create(False);
      Thread1.Priority:=tpLowest;
      Thread2:=TSimpleThread.Create(False);
      Thread2.Priority:=tpLowest;
  end;
  procedure TFrmMain.Timer1Timer(Sender: TObject);
  begin
      Edit1.Text:=IntToStr(Thread1.Count);
      Edit2.Text:=IntToStr(Thread2.Count);
      Thread1.Count:=0;
      Thread2.Count:=0;
  end;
  procedure TFrmMain.TrackBar1Change(Sender: TObject);
  var
     I:integer;
     Priority:TThreadPriority;
  begin
     Priority:=tpLowest;
     for I:=0 to (Sender as tTrackBar).Position-1 do
        inc(Priority);
     If Sender=TrackBar1
```



```
     then
       Thread1.Priority:=Priority
     else
       Thread2.Priority:=Priority;
  end;
  end.
  unit UThr;
  interface
  uses
   Classes;
  type
    TSimpleThread = class(TThread)
    protected
      procedure Execute; override;
    public
      Count:integer;
    end;
  implementation
  { TSimpleThread }
  procedure TSimpleThread.Execute;
  //подсчитать среднее значение 10 случайных чисел и
  //увеличитьCount на 1.
  var
     I,Total,Avg:integer;
  begin
    while True do
    begin
      Total:=0;
      for I:=1 to 10 do
         inc(Total,Random(MaxInt));
      Avg:=Total div 10;
      inc(Count);
    end;
  end;
  end.
```

В Win32 API синхронизация критических секций может осуществляться с помощью глобальной записи специального вида типа `TRTLCriticalSection`. Ее инициализация выполняется процедурой `InitializeCriticalSection(var lpCriticalSection: TRTLCriticalSection);`

Перед входом в критическую секцию требуется вызвать процедуру `EnterCriticalSection(var lpCriticalSection: TRTLCriticalSection);`

При выходе –

```
LeaveCriticalSection(var lpCriticalSection:
TRTLCriticalSection);
```

По окончании работы –

```
DeleteCriticalSection(var lpCriticalSection:
TRTLCriticalSection);
```

Реализация потоков в среде Visual Studio .NET

Домен приложения

Выполнение приложений .NET начинается с запуска .NET Framework. Это процесс со своими потоками, специальными атрибутами и правилами взаимодействия с другими процессами. Приложения .NET выполняются в ОДНОМ процессе. Для этих приложений процесс .NET Framework играет роль, аналогичную роли операционной системы при обеспечении выполнения процессов. Выполняемые в процессе. NET Framework .NET-приложения также изолируются друг от друга. Средством изоляции .NET-приложений являются домены приложений. Домен приложения изолирует (на логическом уровне) выполняемые приложения и используемые в его рамках ресурсы. В процессе .NET Framework может выполняться множество доменов приложений. При этом в рамках одного домена может выполняться множество потоков. На рис. 6 представлена структура процесса .NET Framework, где TLS – локальная память потока.

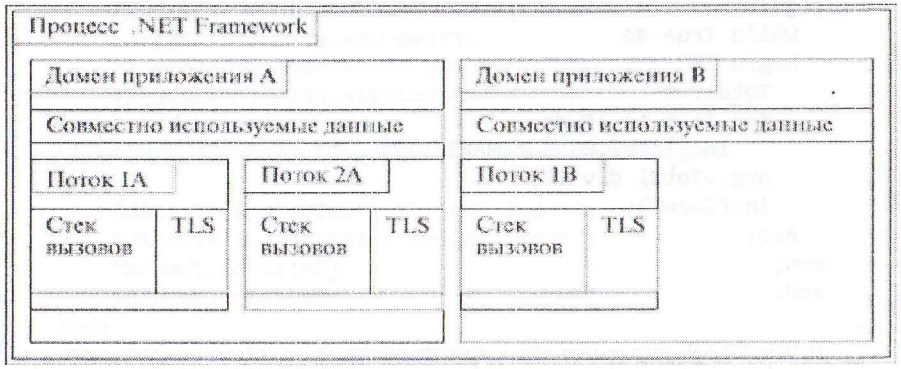


Рис.6. Процесс .NET Framework

Обзор пространства имен System.Threading

В этом пространстве объявляются типы, которые используются для создания многопоточных приложений: работа с потоком, средства синхронизации доступа к общим данным, примитивный вариант класса Timer.

Тип	Назначение
Interlocked	Синхронизация доступа к общим данным
Monitor	Синхронизация потоковых объектов при помощи блокировок и управления ожиданием
Mutex	Синхронизация ПРОЦЕССОВ
Thread	Собственно класс потока, работающего в среде выполнения .NET. В текущем домене приложения с помощью этого класса создаются новые потоки
ThreadPool	Класс, предоставляющий средства управления набором взаимосвязанных потоков
ThreadStart	Класс-делегат для метода, который должен быть выполнен перед запуском потока
Timer	Вариант класса-делегата, который обеспечивает передачу управления некоторой функции-члену (неважно какого класса!) в указанное время. Сама процедура ожидания выполняется потоком в пуле потоков
TimerCallback	Класс-делегат для объектов класса Timer.
WaitHandle	Объекты – представители этого класса являются объектами синхронизации (обеспечивают многократное ожидание).
WaitCallback	Делегат, представляющий методы для рабочих элементов (объектов) класса ThreadPool

Класс Thread. Общая характеристика

Thread-класс представляет управляемые потоки: создает потоки и управляет ими — устанавливает приоритет и статус потоков. Это объектная оболочка вокруг определенного этапа выполнения программы внутри домена приложения.

Статические члены класса Thread	Назначение
CurrentThread	Свойство. Только для чтения. Возвращает ссылку на поток, выполняемый в настоящее время
GetData() SetData()	Обслуживание слота текущего потока
GetDomain() GetDomainID()	Получение ссылки на домен приложения (на ID домена), в рамках которого работает указанный поток
Sleep()	Блокировка выполнения потока на определенное время
Нестатические члены	Назначение
IsAlive	Свойство. Если поток запущен, то true
IsBackground	Свойство. Работа в фоновом режиме. GC работает как фоновый поток

Name	Свойство. Дружественное текстовое имя потока. Если поток никак не назван – значение свойства установлено в null. Поток может быть поименован единожды. Попытка переименования потока возбуждает исключение
Priority	Свойство. Значение приоритета потока. Область значений – значения перечисления ThreadPriority
ThreadState	Свойство. Состояние потока. Область значений – значения перечисления ThreadState
Interrupt()	Прерывание работы текущего потока
Join()	Ожидание появления другого потока (или определенного промежутка времени) с последующим завершением
Resume()	Возобновление выполнения потока после приостановки
Start()	Начало выполнения ранее созданного потока, представленного делегатом класса ThreadStart
Suspend()	Приостановка выполнения потока
Abort()	Завершение выполнения потока посредством генерации исключения TreadAbortException в останавливаемом потоке. Это исключение следует перехватывать для продолжения выполнения оставшихся потоков приложения. Перегруженный вариант метода содержит параметр типа object, который может включать дополнительную специфичную для данного приложения информацию.

Запустить поток можно единственным способом: указав точку входа потока, – метод, к выполнению операторов которого должен приступить запускаемый поток.

Точкой входа ПЕРВИЧНОГО потока являются СТАТИЧЕСКИЕ функции Main или WinMain. Точнее, первый оператор метода.

Точка входа ВТОРИЧНОГО потока назначается при создании потока.

Создание потока предполагает три этапа:

- определение метода, который будет играть роль точки входа в поток;
- создание объекта – представителя специального класса-делегата (**ThreadStart class**), который настраивается на точку входа в поток;
- создание объекта – представителя класса потока. При создании объекта потока конструктору потока передается в качестве параметра ссылка на делегата, настроенного на точку входа.

Точкой входа в поток не может быть конструктор, поскольку не существует делегатов, которые могли бы настраиваться на конструкторы.

Сигнатура точки входа в поток определяется характеристиками класса-делегата:

```
public delegate void ThreadStart();
```


Запуск вторичных потоков

В следующем примере из главного потока запускаются вторичные потоки последовательно. А уже последовательность выполнения этих потоков определяется планировщиком.

```
using System;
using System.Threading;

namespace ThreadApp_1
{
      // Рабочий класс.
      class Worker
      {
        int allTimes;
        int n;
        // Конструктор умолчания...
        public Worker()
        {
          n = 0;
          allTimes = 0;
           }
// Конструктор с параметрами...
public Worker(int nKey, int tKey)
{
n = nKey;
allTimes = tKey;
}

// Тело рабочей функции...
public void DoItEasy()
{//===================================
      int i;
      for (i = 0; i < allTimes; i++)
      {
      if (n == 0)
      Console.Write("{0,25}\r",i);
      else
      Console.Write("{0,10}\r",i);
      }
      Console.WriteLine("\nWorker was here!");
      }//===================================
}
class StartClass
      {
   static void Main(string[] args)
```

```
    {
      Worker w0 = new Worker(0,100000);
      Worker w1 = new Worker(1,100000);
      ThreadStart t0, t1;
      t0 = new ThreadStart(w0.DoItEasy);
      t1 = new ThreadStart(w1.DoItEasy);
      Thread th0, th1;
      th0 = new Thread(t0);
      th1 = new Thread(t1);
      // При создании потока не обязательно использовать делегат.
      // Возможен и такой вариант. Главное – это сигнатура функции.
      // th1 = new Thread(w1.DoItEasy);
      th0.Start();
      th1.Start();
         }
      }
}
```

Первичный поток ничем не лучше любых других потоков приложения. Он может завершиться раньше всех им же порожденных потоков! Приложение же завершается после выполнения ПОСЛЕДНЕЙ команды в ПОСЛЕДНЕМ выполняемом потоке. Неважно, в каком.

Приостановка выполнения потока

Обеспечивается статическим методом Sleep(). Метод статический – это значит, что всегда производится не «усыпление», а «САМОусыпление» выполняемого в данный момент потока. Выполнение методов текущего потока блокируется на определенные интервалы времени. Все зависит от выбора перегруженного варианта метода. Планировщик потоков смотрит на поток и принимает решение относительно того, можно ли продолжить выполнение усыпленного потока.

В самом простом случае целочисленный параметр определяет временной интервал блокировки потока в миллисекундах.

Если значение параметра установлено в 0, поток будет остановлен до того момента, пока не будет предоставлен очередной интервал для выполнения операторов потока.

Если значение интервала задано с помощью объекта класса *TimeSpan*, то момент, когда может быть возобновлено выполнение потока, определяется с учетом закодированной в этом объекте информации:

```
// Поток заснул на 1 час, 2 минуты, 3 секунды:
Thread.Sleep(new TimeSpan(1,2,3));
:::::
// Поток заснул на 1 день, 2 часа, 3 минуты, 4 секунды,
//5 миллисекунд:
```



```
Thread.Sleep(new TimeSpan(1,2,3,4,5));
```

Значение параметра, представленное выражением *System.Threading.Timeout.Infinite* позволяет усыпить поток на неопределенное время. А разбудить поток при этом можно с помощью метода *Interrupt()*, который в этом случае вызывается из другого потока:

```
using System;
using System.Threading;

class Worker
 {
      int allTimes;
      // Конструктор с параметрами...
      public Worker(int tKey)
      {
      allTimes = tKey;
      }

      // Тело рабочей функции...
     public void DoItEasy()
      {//====================================
          int i;
          for (i = 0; i < allTimes; i++)
          {
             Console.Write("{0}\r",i);
             if (i == 5000)
             {
             try
             {
            Console.WriteLine("\nThread go to sleep!");
            Thread.Sleep(System.Threading.Timeout.Infinite);
             }
             catch (ThreadInterruptedException e)
             {
             Console.WriteLine("{0}, {1}",e,e.Message);
             }
        }
 }
Console.WriteLine("\nWorker was here!");
 }//====================================
 }
class StartClass
 {
```

```
        static void Main(string[] args)
        {
            Worker w0 = new Worker(10000);
            ThreadStart t0;
            t0 = new ThreadStart(w0.DoItEasy);
            Thread th0;
            th0 = new Thread(t0);
            th0.Start();
            Thread.Sleep(10000);
            if (th0.ThreadState.Equals(ThreadState.WaitSleepJoin))
            th0.Interrupt();
              Console.WriteLine("MainThread was here...");
        }
    }
```

Отстранение потока от выполнения

Обеспечивается нестатическим методом Suspend(). Возобновление выполнения потока должно быть выполнено другим потоком с помощью метода Resume(). Если все не отстраненные от выполнения потоки оказались завершены и некому запустить отстраненный поток, приложение в буквальном смысле «зависает». Пример «зависающего» приложения.

```
using System;
using System.Threading;
public class ThreadWork
 {
    public static void DoWork()
    {
      for(int i=0; i<10; i++)
      {
         Console.WriteLine("Thread - working.");
         Thread.Sleep(25);
          }
    Console.WriteLine("Thread - still alive and working.");
    Console.WriteLine("Thread - finished working.");
 }
 }

class ThreadAbortTest
    {
        public static void Main()
        {
        ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
        Thread myThread = new Thread(myThreadDelegate);
         myThread.Start();
```



```
        Thread.Sleep(10);
        Console.WriteLine("Main - aborting my thread.");
        myThread.Suspend();
        Console.WriteLine("Main ending.");
         }
    }
```

Следует иметь в виду, что метод Suspend() уже устарел... «System.Threading.Thread.Suspend() is obsolete: Thread.Suspend has been deprecated. Please use other classes in System.Threading such as Monitor, Mutex, Event, and Semaphore to synchronize thread or project resources...», такое сообщение выдается компилятором при использовании метода Suspend().

Завершение потоков

•Первый вариант остановки потока тривиален. Поток завершается после выполнения ПОСЛЕДНЕГО оператора выполняемой цепочки операторов. Допустим, в ходе выполнения условного оператора значение некоторой переменной сравнивается с фиксированным значением и в случае совпадения значений управление передается оператору return:

```
for (x=0;;x++)
{
    if (x==max)
    return; // Все. Этот оператор оказался последним.
    else
    {
    ::::::::::
    }
}
```

•Поток может быть остановлен в результате выполнения метода Abort(). Эта остановка является достаточно сложным делом.

1. При выполнении этого метода происходит активация исключения ThreadAbortException. Естественно, это исключение может быть перехвачено в соответствующем блоке catch. Во время обработки исключения допустимо выполнение самых разных действий, которые осуществляются в «остановленном» потоке. В том числе возможна и реанимация остановленного потока путем вызова метода ResetAbort().

2. При перехвате исключения CLR обеспечивает выполнение операторов блоков **finally**, которые выполняются все в том же потоке.

Таким образом, остановка потока путем вызова метода Abort не может рассматриваться как НЕМЕДЛЕННАЯ остановка выполнения потока:

```
using System;
using System.Threading;
public class ThreadWork
 {
```

```
 public static void DoWork()
 {
 int i;
try
 {
  for(i=0; i<100; i++)
  {
    Console.WriteLine("Thread - working {0}.", i);
    Thread.Sleep(10);
  }
 }
 catch(ThreadAbortException e)
 {
Console.WriteLine("Thread - caught ThreadAbortException -
resetting.");
Console.WriteLine("Exception message: {0}", e.Message);
//Thread.ResetAbort();//возможно возобновление потока
 }
 finally
 {
        Console.WriteLine("Thread - in finally statement.");
 }
 Console.WriteLine("Thread - still alive and working.");
Console.WriteLine("Thread - finished working.");
 }
 }
class ThreadAbortTest
 {
 public static void Main()
      {
    ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
    Thread myThread = new Thread(myThreadDelegate);
    myThread.Start();// Вторичный поток стартовал!
    Thread.Sleep(50); // первичный поток - самоусыпился!
    Console.WriteLine("Main - aborting my thread.");
    myThread.Abort();
    Console.WriteLine("Main ending.");
      }
}
```

Метод Join()

Несколько потоков выполняются «параллельно» в соответствии с предпочтениями планировщика потоков. Нестатический метод *Join()* позволяет изменить последовательность выполнения потоков многопоточного приложения. Метод *Join()* выполняется в одном из потоков по отношению к другому потоку.

В результате выполнения этого метода данный текущий поток немедленно блокируется до тех пор, пока не завершит свое выполнение поток, по отношению к которому был вызван метод *Join*.

Перегруженный вариант метода имеет целочисленный аргумент, который воспринимается как временной интервал. В этом случае выполнение текущего потока может быть возобновлено по истечении этого периода времени до завершения этого потока:

```
using System;
using System.Threading;
public class ThreadWork
 {
     public static void DoWork()
 {
     for(int i=0; i<10; i++)
 {
     Console.WriteLine("Thread - working.");
     Thread.Sleep(10);
 }

     Console.WriteLine("Thread - finished working.");
       }
 }

class ThreadTest
 {
 public static void Main()
 {
    ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
    Thread myThread = new Thread(myThreadDelegate);
    myThread.Start();
    Thread.Sleep(100);
    myThread.Join(); // Закомментировать вызов метода и осознать разницу.
Console.WriteLine("Main ending.");
      }
 }
```

Состояния потока (перечисление ThreadState)

Класс *ThreadState* определяет набор всех возможных состояний выполнения для потока. После создания потока и до завершения он находится, по крайней мере, в одном из состояний. Потоки, созданные в общеязыковой среде выполнения, изначально находятся в состоянии *Unstarted*, в то время как внешние потоки, приходящие в среду выполнения, находятся уже в состоянии *Running*. Потоки с состоянием *Unstarted* переходят в состояние *Running* при вызове метода *Start*. Не все комбинации значений *ThreadState*

являются допустимыми. В следующей таблице перечислены действия, вызывающие смену состояния.

Действие	Состояние потока
Поток создается в среде CLR	Unstarted
Поток вызывает метод Start	Running
Поток начинает выполнение	Running
Поток вызывает метод Sleep	WaitSleepJoin
Поток вызывает метод Wait для другого объекта	WaitSleepJoin
Поток вызывает метод Join для другого потока	WaitSleepJoin
Другой поток вызывает метод Interrupt	Running
Другой поток вызывает метод Suspend	SuspendRequested
Поток отвечает на запрос метода Suspend	Suspended
Другой поток вызывает метод Resume	Running
Другой поток вызывает метод Abort	AbortRequested
Поток отвечает на запрос метода Abort	Stopped
Поток завершен	Stopped

Начальное состояние потока (если это не главный поток), в котором он оказывается непосредственно после его создания, – *Unstarted*. В этом состоянии он пребывает до тех пор, пока вызовом метода *Start()* не будет переведен в состояние *Running*.

В дополнение к вышеперечисленным состояниям существует также *Background* – состояние, которое указывает, выполняется ли поток на фоне или на переднем плане.

Свойство *Thread.ThreadState* потока содержит текущее состояние потока. Для определения текущего состояния потока в приложении можно использовать битовые маски. Пример условного выражения:

```
if((myThread.ThreadState & (ThreadState.Stopped | ThreadState. Unstarted))==0) {...}
```

Члены перечисления:

Имя члена	Описание	Значение
Running	Поток был запущен, он не заблокирован, и нет задерживающегося объекта ThreadAbortException	0
StopRequested	Поток запрашивается на остановку. Это только для внутреннего использования	1
SuspendRequested	Запрашивается приостановка работы потока	2
Background	Поток выполняется как фоновый, что является противоположным к приоритетному потоку. Это состояние контролируется заданием свойства Thread.IsBackground	4
Unstarted	Метод Thread.Start не был вызван для потока	8

Stopped	Поток остановлен	16
WaitSleepJoin	Поток заблокирован в результате вызова к методам Wait, Sleep или Join	32
Suspended	Работа потока была приостановлена	64
AbortRequested	Метод Thread.Abort был вызван для потока, но поток еще не получил задерживающийся объект System.Threading.ThreadAbortException, который будет пытаться завершить поток	128
Aborted	Поток находится в Stopped-состоянии	256

Одновременное пребывание потока в различных состояниях

В условиях многопоточного приложения разные потоки могут переводить друг друга в разные состояния. Таким образом, поток может находиться одновременно БОЛЕЕ ЧЕМ В ОДНОМ состоянии.

Например, если поток блокирован в результате вызова метода *Wait*, а другой поток вызвал по отношению к блокированному потоку метод *Abort*, то блокированный поток окажется в одно и то же время в состояниях *WaitSleepJoin* и *AbortRequested*.

В этом случае, как только поток выйдет из состояния *WaitSleepJoin* (в котором он оказался в результате выполнения метода *Wait*), ему будет предъявлено исключение *ThreadAbortException*, связанное с началом процедуры *aborting*.

С другой стороны, не все сочетания значений *ThreadState* допустимы. Например, поток не может одновременно находиться в состояниях *Aborted* и *Unstarted*. Перевод потока из одного состояния в другое, несовместимое с ним состояние, а также повторная попытка перевода потока в одно и то же состояние (пара потоков один за другим применяют метод *Resume()* к одному и тому же потоку) может привести к генерации исключения. Поэтому операторы, связанные с управлением потоками, следует размещать в блоках `try`.

	AR	Ab	Back	U	S	R	W	St	SusR	StopR
Abort Requested	—	N	Y	Y	Y	N	Y	N	Y	N
Aborted	N	—	Y	N	N	N	N	N	N	N
Background	Y	Y	—	Y	Y	N	Y	Y	Y	N
Unstarted	Y	N	Y	—	N	N	N	N	N	N
Suspended	Y	N	Y	N	—	N	Y	N	N	N
Running WaitSleep	N	N	N	N	N	—	N	N	N	N
Join	Y	N	Y	N	Y	N	—	N	Y	N
Stopped	N	N	Y	N	N	N	N	—	Y	N
Suspend Requested	Y	N	Y	N	N	N	Y	N	—	N
Stop Requested	N	N	N	N	N	N	N	N	N	—

Информация о возможности одновременного пребывания потока в нескольких состояниях представлена в таблице допустимых состояний:

Фоновый поток

Потоки выполняются:

- в обычном режиме (Foreground threads) и
- в фоновом режиме (Background threads).

Состояние *Background*-state распознается по значению свойства *IsBackground,* которое указывает на режим выполнения потока: *background* или *foreground*.

Любой *Foreground*-поток можно перевести в фоновый режим, установив значение свойства *IsBackground* в `true`.

Завершение Background-потока не влияет на завершение приложения в целом.

Завершение последнего *Foreground*-потока приводит к завершению приложения, независимо от состояния потоков, выполняемых в фоновом режиме.

Ниже в примере один из потоков переводится в фоновый режим. Изменяя значения переменных, определяющих характеристики циклов, можно проследить за поведением потоков:

```
using System;
using System.Threading;

namespace ThreadApp_1
{

class Worker
 {
 int allTimes;
 public Worker(int tKey)
 {
     allTimes = tKey;
 }
 // Тело рабочей функции...
public void DoItEasy()
 {
     int i;
     for (i = 0; i < allTimes; i++)
 {
 Console.WriteLine("Back thread >>>> {0}\r",i);
 }
Console.WriteLine("\nBackground thread was here!");
 }
 }

class StartClass
```



```
{
   static void Main(string[] args)
{
    long i;
    Worker w0 = new Worker(100000);
    ThreadStart t0;
    t0 = new ThreadStart(w0.DoItEasy);
    Thread th0;
    th0 = new Thread(t0);
    th0.IsBackground = true;
    th0.Start();
    for (i = 0; i < 100 ; i++)
    {
    Console.WriteLine("Fore thread: {0}", i);
    }
    Console.WriteLine("Foreground thread ended");
    }
  }
}
```

Приоритет потока

Задается значениями перечисления *ThreadPriority*. Эти значения используются при планировке очередности выполнения потоков в ПРОЦЕССЕ.

Приоритет потока определяет относительный приоритет потоков.

Каждый поток имеет собственный приоритет. Изначально он задается как *Normal priority*.

Алгоритм планирования выполнения потока позволяет системе определить последовательность выполнения потоков. Операционная система может также корректировать приоритет потока динамически, переводя поток из состояния *foreground* в *background*.

Значение приоритета не влияет на состояние потока. Система планирует последовательность выполнения потоков на основе информации о состоянии потока. Потоки с низшим уровнем приоритета выполняются лишь после того, как в процессе будет завершено выполнение потоков с более высоким приоритетом.

Приоритет процесса может принимать следующие значения

- *Highest*
- *AboveNormal*
- *Normal*
- *BelowNormal*
- *Lowest*

Приоритет потока – это относительная величина. Прежде всего, система планирует очередность выполнения ПРОЦЕССА. В рамках выполняемого процесса определяется последовательность выполнения потоков.

Передача данных во вторичный поток

Делегат – представитель класса-делегата ThreadStart обеспечивает запуск вторичных потоков. Это элемент СТАНДАРТНОГО механизма поддержки вторичных потоков. Именно этим и объясняется главная особенность этого делегата: настраиваемые с его помощью стартовые функции потоков НЕ имеют параметров и не возвращают значений. Это означает, что невозможно осуществить запуск потока с помощью метода, имеющего параметры, а также получить какое-либо значение при завершении стартовой функции потока.

Вторичный поток строится на основе методов конкретного класса. Это означает, что сначала создается объект – представитель класса, затем объект потока с настроенным на стартовую функцию делегатом, после чего поток стандартным образом запускается. При этих условиях задача передачи данных потоку может быть возложена на конструкторы класса. На списки их параметров никаких особых ограничений не накладывается. В конструкторе могут быть реализованы самые сложные алгоритмы подготовки данных.

```
using System;
using System.Threading;
 // Класс WorkThread содержит всю необходимую для выполнения
 // данной задачи информацию, а также и соответствующий метод.
 public class WorkThread
 {
     // State information used in the task.
     private string entryInformation;
     private int value;
     // Конструктор получает всю необходимую//информацию через параметры.
     public WorkThread(string text, int number)
     {
        entryInformation = text;
        value = number;
      }
//Рабочий метод потока непосредственно после своего запуска
//получает доступ ко всей необходимой ранее//подготовленной информации.
 public void ThreadProc()
 {
    Console.WriteLine(entryInformation, value);
 }
 }
 // Точка входа приложения.
 //
```



```
    public class Example
 {
    public static void Main()
 {
 //Подготовка к запуску вторичного потока предполагает создание
 //объекта класса потока.
 //В этот момент вся необходимая для работы
 //потока информация передается через параметры конструктора.
 //Здесь переданы необходимые детали, которые будут составлены
 //стандартным образом в строку методом WriteLine.
 WorkThread tws =new WorkThread("This report displays the num-
ber {0}.", 125);
  //Создали объект потока, затем его запустили.
  Thread t = new Thread(new ThreadStart(tws.ThreadProc));
  t.Start();
  Console.WriteLine("Первичный поток поработал. Теперь ждет.");
  t.Join();
  Console.WriteLine
  ("Сообщение из первичного потока: Вторичный поток отработал.");
  Console.WriteLine
     ("Сообщение из первичного потока: Главный поток остановлен. ");
   }
 }
```

Контроль вторичных потоков. Callback-методы

Первичный поток создал и запустил вторичный поток для решения определенной задачи. Вторичный поток выполнил поставленную задачу и завершил свою работу. Возможно, что от результатов работы вторичного потока зависит дальнейшая работа приложения. Возможно, что до завершения выполнения вторичного потока первичному потоку вообще нечего делать, и он приостановлен в результате выполнения метода Join.

Проблема заключается в том, КОГДА и КАКИМ ОБРАЗОМ о проделанной работе станет известно в первичном потоке.

Для анализа результата выполнения вторичного потока можно использовать метод класса, который обеспечивает запуск вторичного потока. Соответствующим образом настроенный делегат также может быть передан в качестве параметра конструктору вторичного потока. Вызывать метод класса, запустившего вторичный поток, можно будет по выполнении работ во ВТОРИЧНОМ потоке.

Таким образом, функция, контролирующая завершение работы вторичного потока, сама будет выполняться во ВТОРИЧНОМ потоке. Если при этом для дальнейшей работы первичного потока (который в этот момент, возможно, находится в состоянии ожидания) необходима информация о ре-

зультатах проделанной вторичным потоком работы, контролирующая функция справится с этой задачей за счет того, что она имеет доступ ко всем данным и методам своего класса. И неважно, в каком потоке она при этом выполнялась. Пример использования Callback-метода представлен ниже.

```
using System;
using System.Threading;
 // Класс WorkThread включает необходимую информацию, ме-
тод и делегат
 //для вызова метода, который запускается после выполнения
задачи.
 public class WorkThread
 {
 // Входная информация.
 private string entryInformation;
 private int value;
 //Ссылка на объект – представитель класса-делегата,
 //с помощью/которого вызывается метод обратного вызова.
 //Сам класс-делегат
 //объявляется позже.
 private CallbackMethod callback;
 // Конструктор получает входную информацию и настраивает
 // callback delegate.
 public WorkThread(string text, int number,
CallbackMethod callbackDelegate)
 {
        entryInformation = text;
        value = number;
        callback = callbackDelegate;
 }
 // Метод, обеспечивающий выполнение поставленной задачи:
 // составляет строку и после дополнительной
 //проверки настройки
 // callback-делегата обеспечивает вызов метода.
 public void ThreadProc()
 {
 Console.WriteLine(entryInformation, value);
 if (callback != null)
 callback(1); // вызвал ЧУЖОЙ МЕТОД
//в СВОЕМ потоке.
 }
 }
 // Класс-делегат задает сигнатуру callback-методу.
```



```
 //
 public delegate void CallbackMethod(int lineCount);
 // Entry Point for the example.
 //
 public class Example
 {
    public static void Main()
 {
 // Supply the state information required by the task.
      WorkThread tws = new WorkThread("This report displays the
      number {0}.",125, new CallbackMethod(ResultCallback));
Thread t = new Thread(new (tws.ThreadProc));
 t.Start();
   Console.WriteLine("Первичный поток поработал. Теперь ждет.");
 t.Join();
 Console.WriteLine("Вторичный поток отработал. Главный поток остановлен.");
 }
// Callback-метод соответствует сигнатуре callback класса
//делегата.
 public static void ResultCallback(int lineCount)
 {
 Console.WriteLine("Вторичный поток обработал {0} строк.",
 lineCount);
 }
 }
```

Callback-метод – метод – член класса, запустившего вторичный поток. Этот метод запускается «в качестве уведомления» о том, что вторичный поток «завершил выполнение своей миссии». Особенность Callback-метода заключается в том, что он выполняется в «чужом» потоке.

Организация взаимодействия потоков. Блокировки и тупики

Пример взаимодействующих потоков рассматривается далее:

```
// Взаимодействующие потоки разделяют общие ресурсы – пару
очередей.
// Каждый поток должен последовательно получить доступ к каждой из
очередей.
// Из одной очереди взять, в другую положить. Поток оказыва-
ется
// заблокирован, когда одна из очередей оказывается занятой другим по-
током.
using System;
using System.Threading;
```

```
using System.Collections;
namespace CommunicatingThreadsQueue
{
 // Модифицированный вариант очереди - очередь с флажком.
 // Захвативший эту очередь поток объявляет очередь "закры-
той".
public class myQueue: Queue
 {
private bool isFree;
public bool IsFree
{
 get
 {
    return isFree;
 }
 set
 {
    isFree = value;
 }
 }
public object myDequeue()
 {
     if (IsFree)
    {
       IsFree = false;
       return base.Dequeue();
    }
     else return null;
 }

public bool myEnqueue(object obj)
 {
if (IsFree == true) {base.Enqueue(obj); return true;}
else return false;
 }
 }
public delegate void CallBackFromStartClass (string param);
// Данные. Предмет и основа взаимодействия двух потоков.
class CommonData
{
  private int iVal;
  public int iValProp
{
get
{
```



```
  +
  +return iVal;
}
set
{
iVal = value;
}
}
public CommonData(int key)
{
  iVal = key;
}
}
   // Классы Receiver и Sender: основа взаимодействующих потоков.
class Receiver
{
myQueue cdQueue0;
myQueue cdQueue1;
CallBackFromStartClass callBack;
int threadIndex;
// Конструктор...
public Receiver(ref myQueue queueKey0,ref myQueue queueKey1,
CallBackFromStartClass cbKey,int iKey)
{
  threadIndex = iKey;
  if (threadIndex == 0)
{
    cdQueue0 = queueKey0;
    cdQueue1 = queueKey1;
}
else
{
      cdQueue1 = queueKey0;
      cdQueue0 = queueKey1;
}
callBack = cbKey;
}
public void startReceiver()
{
       DoIt();
}
// Тело рабочей функции...
public void DoIt()
{
   CommonData cd = null;
```

```
    while (true)
    {
    if (cdQueue0.Count > 0)
    {
        while (true)
        {
        cd = (CommonData)cdQueue0.myDequeue();
        if (cd != null) break;
        Console.WriteLine(">> Receiver{0} is blocked.", threadIndex);
}
// Задержка "на обработку" полученного блока информации влияет
// на частоту и продолжительность блокировок.
Thread.Sleep(cd.iValProp*100);
// И это не ВЗАИМНАЯ блокировка потоков.
// "Обработали" блок - открыли очередь.
// И только потом предпринимается попытка
// обращения к очереди оппонента.
cdQueue0.IsFree = true;
//Записали результат во вторую очередь.
while (cdQueue1.myEnqueue(cd) == false)
{
Console.WriteLine("<< Receiver{0} is blocked.", threadIndex);
}
// А вот если пытаться освободить захваченную потоком очередь
// в этом месте - взаимной блокировки потоков не избежать!
// cdQueue0.IsFree = true;

// Сообщили о состоянии очередей.
Console.WriteLine("Receiver{0}{1}>{2}",threadIndex.ToString(),
cdQueue0.Count,cdQueue1.Count);
}
else
{
   cdQueue0.IsFree = true;
   callBack(string.Format("Receiver{0}",threadIndex.ToString()));
}
}
}
 }
class Sender
 {
   Random rnd;
   int stopVal;
myQueue cdQueue0;
myQueue cdQueue1;
```



```
CallBackFromStartClass callBack;
// Конструктор...
public Sender(ref myQueue queueKey0,ref myQueue
queueKey1,int key,
CallBackFromStartClass cbKey)
{
   rnd = new Random(key);
   stopVal = key;
   cdQueue0 = queueKey0;
   cdQueue1 = queueKey1;
   callBack = cbKey;
}
public void startSender()
{
   sendIt();
}
// Тело рабочей функции...
public void sendIt()
{//==================================
cdQueue0.IsFree = false;
 cdQueue1.IsFree = false;
while (true)
{
if (stopVal > 0)
{
      // Размещение в очереди нового члена.
      cdQueue0.Enqueue(new CommonData(rnd.Next(0,stopVal)));
      cdQueue1.Enqueue(new CommonData(rnd.Next(0,stopVal)));
      stopVal--;
    }
    else
    {
        cdQueue0.IsFree = true;
        cdQueue1.IsFree = true;
        callBack("Sender");
    }
    Console.WriteLine
    ("Sender. The rest of notes: {0}, notes
    in queue:{1},{2}.", stopVal, cdQueue0.Count, cdQueue1.Count);
     }
     }//==================================
 }
class StartClass
{
static Thread th0, th1, th2;
static myQueue NotificationQueue0;
static myQueue NotificationQueue1;
```

```
static string[] report = new string[3];
static void Main(string[] args)
{
StartClass.NotificationQueue0 = new myQueue();
StartClass.NotificationQueue1 = new myQueue();
// Конструкторы классов Receiver и Sender несут дополнительную
// нагрузку. Они обеспечивают необходимыми значениями методы,
// выполняемые во вторичных потоках.
Sender sender;
// По окончании работы отправитель вызывает
//функцию-терминатор.
// Для этого используется специальный делегат.
sender = new Sender(ref NotificationQueue0, ref NotificationQueue1,
10, new CallBackFromStartClass(StartClass.StopMain));
Receiver receiver0;
// Выбрав всю очередь, получатель вызывает
//функцию-терминатор.
receiver0 = new Receiver(ref NotificationQueue0,
//ref NotificationQueue1,
 new CallBackFromStartClass(StartClass.StopMain),0);
Receiver receiver1;
// Выбрав всю очередь, получатель вызывает
//функцию-терминатор.
receiver1 = new Receiver(ref NotificationQueue0,
ref NotificationQueue1,
new CallBackFromStartClass(StartClass.StopMain),1);
// Стартовые функции потоков должны соответствовать сигнатуре
// класса делегата ThreadStart.
//Поэтому они не имеют параметров.
ThreadStart t0, t1, t2;
t0 = new ThreadStart(sender.startSender);
t1 = new ThreadStart(receiver0.startReceiver);
t2 = new ThreadStart(receiver1.startReceiver);
// Созданы вторичные потоки.
StartClass.th0 = new Thread(t0);
StartClass.th1 = new Thread(t1);
StartClass.th2 = new Thread(t2);
// Запущены вторичные потоки.
StartClass.th0.Start();
// Еще раз о методе Join():
// Выполнение главного потока приостановлено до завершения
// выполнения вторичного потока загрузки очередей.
// Потоки получателей пока отдыхают.
StartClass.th0.Join();
// Отработал поток загрузчика.
```



```
// Очередь получателей.
StartClass.th1.Start();
StartClass.th2.Start();
// Метод Join():
// Выполнение главного потока опять приостановлено
// до завершения выполнения вторичных потоков.
StartClass.th1.Join();
StartClass.th2.Join();
// Последнее слово остается за главным потоком приложения.
// Но только после того, как отработают терминаторы.
Console.WriteLine
("Main():"+report[0]+". "+report[1]+". "+report[2]+". Bye.");
}
// Функция - член класса StartClass выполняется во ВТОРИЧНОМ потоке!
public static void StopMain(string param)
{
Console.WriteLine("StopMain: " + param);
// Остановка рабочих потоков. Ее выполняет функция - член
// класса StartClass. Этой функции в силу своего определения
// известно ВСЕ о вторичных потоках. Но выполняется она
// в ЧУЖИХ (вторичных) потоках.
if (param.Equals("Sender"))
{
    report[0] = "Sender all did.";
    StartClass.th0.Abort();
}
if (param.Equals("Receiver0"))
{
    report[1] = "Receiver0 all did.";
    StartClass.th1.Abort();
}
if (param.Equals("Receiver1"))
{
    report[2] = "Receiver1 all did.";
    StartClass.th2.Abort();
}
// Этот оператор не выполняется! Поток, в котором выполняется
// метод - член класса StartClass StopMain(), остановлен.
Console.WriteLine("StopMain(): bye.");
}
}
}
```

Безопасность данных и критические секции кода

Типичными средствами синхронизации потоков являются:

- критические секции,
- мониторы,
- мьютексы.

Специальные возможности мониторов

Класс *Monitor* управляет доступом к коду с использованием объекта синхронизации. Объект синхронизации предоставляет возможности для ограничения доступа к блоку кода, в общем случае обозначаемого как критическая секция.

Поток выполняет операторы. Выполнение оператора, обеспечивающего захват с помощью монитора объекта синхронизации, закрывает критическую секцию кода.

Другие потоки, выполняющие данную последовательность операторов, не могут продвинуться дальше оператора захвата монитором объекта синхронизации и переходят в состояние ожидания до тех пор, пока поток, захвативший с помощью монитора критическую секцию кода, не освободит ее.

Таким образом, монитор гарантирует, что никакой другой поток не сможет обратиться к коду, выполняемому потоком, который захватил с помощью монитора и данного объекта синхронизации критическую секцию кода, пока она не будет освобождена, если только потоки не выполняют данную секцию кода с использованием другого объекта синхронизации.

Следующая таблица описывает действия, которые могут быть предприняты потоками при взаимодействии с монитором:

Действие	Описание
Enter, TryEnter	Закрытие секции с помощью объекта синхронизации. Это действие также обозначает начало критической секции. Никакие другие потоки не могут войти в заблокированную критическую секцию, если только они не используют другой объект синхронизации
Exit	Освобождает блокировку критической секции кода. Также обозначает конец критической секции, связанной с данным объектом синхронизации
Wait	Поток переходит в состояние ожидания, предоставляя тем самым другим потокам возможность выполнения других критических секций кода, связанных с данным объектом синхронизации. В состоянии ожидания поток остается до тех пор, пока на выходе из другой секции, связанной с данным объектом синхронизации, другой поток не выполнит на мониторе действия Pulse (PulseAll), которые означают изменение состояния объекта синхронизации и обеспечивают выход потока из состояния ожидания на входе в критическую секцию

Pulse (signal), PulseAll	Посылает сигнал ожидающим потокам. Сигнал служит уведомлением ожидающему потоку, что состояние объекта синхронизации изменилось, и что владелец объекта готов его освободить. Находящийся в состоянии ожидания поток фактически находится в очереди для получения доступа к объекту синхронизации

Enter- и Exit-методы используются для обозначения начала или конца критической секции. Если критическая секция представляет собой «непрерывное» множество инструкций, закрытие кода посредством метода Enter гарантирует, что только один поток сможет выполнять код, закрытый объектом синхронизации.

Рекомендуется размещать эти инструкции в try блок и помещать Exit в finally-блоке.

Все эти возможности обычно используются для синхронизации доступа к статическим и нестатическим методам класса. Нестатический метод блокируется посредством объекта синхронизации. Статический объект блокируется непосредственно в классе, членом которого он является.

```
 // Синхронизация потоков с использованием класса монитора.
 // Монитор защищает очередь от параллельного вторжения со
 //стороны
 // взаимодействующих потоков из разных фрагментов кода.
 // Однако монитор не может защитить потоки от взаимной
 //блокировки.
 // Поток просыпается, делает свою работу, будит конкурента,
 //засыпает сам.
 // К тому моменту, как поток будит конкурента, конкурент
 //должен спать.
 // Активизация незаснувшего потока не имеет никаких
 //последствий.
 // Если работающий поток разбудит не успевший заснуть
 //поток – возникает
 // тупиковая ситуация. Оба потока оказываются
 //погруженными в сон.
 // В этом случае имеет смысл использовать перегруженный
 //вариант метода
 // Wait – с указанием временного интервала.
using System;
 using System.Threading;
 using System.Collections;
namespace MonitorCS1
{
 class MonitorApplication
 {
```

```
 const int MAX_LOOP_TIME = 100;
 Queue xQueue;
public MonitorApplication()
 {
   xQueue = new Queue();
 }
public void FirstThread()
 {
   int counter = 0;
   while(counter < MAX_LOOP_TIME)
   {
       Console.WriteLine("Thread_1___");
       counter++;
       Console.WriteLine("Thread_1...{0}", counter);
      try
       {
       //Push element.
       xQueue.Enqueue(counter);
      foreach(int ctr in xQueue)
       {
       Console.WriteLine(":::Thread_1:::{0}", ctr);
   }
   }
       catch (Exception ex)
       {
       Console.WriteLine(ex.ToString());
       }
       //Release the waiting thread. Применяется к конкурирующему потоку.
       lock(xQueue){Monitor.Pulse(xQueue);}
     Console.WriteLine(">1 Wait<");
       //Wait, if the queue is busy. Применяется к текущему потоку.
       // Собственное погружение в состояние ожидания.
       lock(xQueue){Monitor.Wait(xQueue,1000);}
       Console.WriteLine("!1 Work!");
   }
   Console.WriteLine("*****1 Finish*****");
   lock(xQueue) {Monitor.Pulse(xQueue);}
 }
public void SecondThread()
 {
    int counter = 0;
   while(counter < MAX_LOOP_TIME)
    {
    //Release the waiting thread. Применяется к конкурирующему потоку.
    lock(xQueue){Monitor.Pulse(xQueue);}
```



```
    Console.WriteLine(">2 Wait<");
    // Собственное погружение в состояние ожидания.
    lock(xQueue){Monitor.Wait(xQueue,1000);}
   Console.WriteLine("!2 Work!");
    Console.WriteLine("Thread_2___");
   try
    {
    foreach(int ctr in xQueue)
    {
        Console.WriteLine(":::Thread_2:::{0}", ctr);
    }
   //Pop element.
    counter = (int)xQueue.Dequeue();
    }
    catch (Exception ex)
    {
      counter = MAX_LOOP_TIME;
      Console.WriteLine(ex.ToString());
    }
   Console.WriteLine("Thread_2...{0}",counter);
    }
    Console.WriteLine("*****2 Finish*****");
   lock(xQueue) {Monitor.Pulse(xQueue);}
 }
static void Main(string[] args)
 {
 // Create the MonitorApplication object.
MonitorApplication test = new MonitorApplication();
Thread tFirst = new Thread(new ThreadStart(test.FirstThread));
 // Вторичные потоки созданы!
 Thread tSecond = new Thread(new ThreadStart(test.SecondThread));
 //Start threads.
 tFirst.Start();
 tSecond.Start();
 // Ждать завершения выполнения вторичных потоков.
 tFirst.Join();
 tSecond.Join();
 }
 }
 }
```

Первый способ синхронизации – воспользоваться ключевым словом **lock**. Это ключевое слово позволяет заблокировать блок кода таким образом, что в отдельный момент времени его сможет использовать только один

поток. Всем остальным потокам придется ждать, пока поток, занявший этот блок кода, завершит свою работу.

```
internal class WorkerClass
{
 public void DoSomeWork()
{
 //только один поток в конкретный момент времени сможет
// выполнять этот код
lock(this)
{
      for(int i=0; i<5;i++)
      {
           Console.WriteLine("Worker says: {0},",i);
}
      }
}
}
public class MainClass
{
 public static int Main(string[] args)
 {
      //Создаем единственный объект класса WorkerClass
      WorkerClass w = new WorkerClass();
// создаем три потока, каждый из которых производит вызов
//одного и того же объекта
Thread workerThreadA = new Thread(new ThreadStart(w.DoSameWork));
Thread workerThreadB = new Thread(new ThreadStart(w.DoSameWork));
Thread workerThreadC = new Thread(new ThreadStart(w.DoSameWork));
// Запускаем все три потока
workerThreadA.Start();
workerThreadB.Start();
workerThreadC.Start();
return 0;
}
}
```

Можно считать, что ключевое слово **lock** в C# – аналог примитива CRITICAL_SECTION и соответствующих вызовов API Win32.

Семейство Interlocked-методов

Если несколько потоков обращается к общим данным, нужно обеспечить безопасность потоков. Самый быстрый способ – задействовать семейство Interlocked-методов. Эти методы работают очень быстро (по сравнению с другими конструкциями синхронизации потоков) и просты в применении. Недостаток один: очень ограниченные возможности. В классе *System.Threading. Interlocked*

определено несколько статических методов, которые могут автоматически изменять переменную безопасно для потоков. Методы, оперирующие переменными типа *Int32*, используются наиболее часто:

```
public static class Interlocked
{
//Автоматически выполняет (location ++)
public static Int32 Increment(ref Int32 location);
//Автоматически выполняет (location --)
public static Int32 Decrement(ref Int32 location);
//Автоматически выполняет (location +=value),
// value может быть отрицательным
public static Int32 Add(ref Int32 location1,Int32 value);
//Автоматически выполняет (location = value)
public static Int32 Exchange(ref Int32 location1, Int32 value);
//Автоматически выполняет следующее:
// if (location1 = =comparand) location1=value
public static Int32 CompareExchange
(ref Int32 location1,Int32 value, Int32 comparand);
…
}
```

Помимо этого, в классе *Interlocked* есть методы *Exchange* и *CompareExchange*, принимающие параметры *Object*, *IntPtr*, *Single*, *Double*.

Класс Monitor и блоки синхронизации

В Win32 API структура CRITICAL_SECTION и связанные с ней функции предлагают самый быстрый и эффективный способ синхронизации потоков для взаимоисключающего доступа к общему ресурсу, когда все потоки работают в общем потоке. Взаимоисключающий доступ к общему ресурсу нескольких потоков – это, наверное, самая распространенная форма синхронизации потоков. CLR не предоставляет структуру CRITICAL_SECTION, но предлагает похожий механизм, позволяющий организовать взаимоисключающий доступ к ресурсу в наборе потоков одного процесса. Используя этот механизм, задействуют класс System.Threading.Monitor и блоки синхронизации.

Семафоры

public sealed class Semaphore : WaitHandle

Используйте класс Semaphore для управления доступом к пулу ресурсов.

Потоки производят захват семафора, вызывая метод WaitOne, унаследованный от класса WaitHandle, и освобождают семафор вызовом метода Release.

Счетчик семафора уменьшается на единицу каждый раз, когда выполняется WaitOne, и увеличивается на единицу, когда поток освобождает семафор. Когда счетчик равен нулю, последующие запросы блокируются, по-

ка другие потоки не освободят семафор. Когда семафор освобожден всеми потоками, счетчик имеет максимальное значение, заданное при создании семафора.

Гарантированный порядок, в котором бы освобождались заблокированные семафором потоки, например FIFO или LIFO, отсутствует.

Поток может выполнять захват семафора несколько раз, вызывая многократно метод WaitOne. Чтобы освободить некоторые из этих входов, поток может вызвать перегруженный метод Release() без параметров несколько раз, или вызвать перегруженный метод Release(Int32), указывающий количество освобождаемых входов.

Класс Semaphore не обеспечивает потоковой идентификации для вызовов методов WaitOne или Release. Обеспечить корректное освобождение потоками семафоров — ответственность программиста. Предположим, например, что у семафора максимальное значение счетчика равно двум, и два потока, A и B захватывают семафор. Если ошибка программирования в потоке B заставляет его вызывать метод Release дважды, оба вызова оканчиваются успешно. Счетчик на семафоре достиг максимального значения, и если поток A вызовет метод Release, будет выдано исключение SemaphoreFullException.

Семафоры бывают двух типов: локальные семафоры и именованные системные семафоры. При создании объекта Semaphore с помощью конструктора, позволяющего передавать параметр с именем семафора, объект связывается с имеющим данное имя семафором операционной системы. Именованные системные семафоры доступны в пределах всей операционной системы и могут быть использованы для синхронизации действий процессов. Можно создать несколько объектов Semaphore, представляющих один и тот же именованный системный семафор, и использовать метод OpenExisting для открытия существующего именованного системного семафора.

Локальный семафор существует только внутри одного процесса. Он может использоваться любым потоком в процессе, имеющим ссылку на локальный объект Semaphore. Каждый объект Semaphore является отдельным локальным семафором.

В следующем примере кода создается семафор с максимальным значением счетчика равным 3 и исходным значением счетчика равным 0. В примере запускаются пять потоков, которые блокируют ожидание семафора. Главный поток использует перегруженный метод Release(Int32) для увеличения счетчика семафора до максимального значения, позволяя трем потокам захватить семафор. В каждом потоке используется метод Thread.Sleep для создания имитирующей работу односекундной задержки, а затем вызывается перегруженный метод Release() для освобождения семафора. Каждый раз при освобождении семафора отображается предыдущее значение

счетчика семафора. Выводимые в консоль сообщения позволяют отслеживать использование семафора.

```
using System;
using System.Threading;
public class Example
{
    // A semaphore that simulates a limited resource pool.
    //
    private static Semaphore _pool;
    // A padding interval to make the output more orderly.
    private static int _padding;

    public static void Main()
    {
        // Create a semaphore that can satisfy up to three
        // concurrent requests. Use an initial count of zero,
        // so that the entire semaphore count is initially
        // owned by the main program thread.
        //
        _pool = new Semaphore(0, 3);
        // Create and start five numbered threads.
        //
        for(int i = 1; i <= 5; i++)
        {
            Thread t = new Thread
            (new ParameterizedThreadStart(Worker));
            // Start the thread, passing the number.
            //
            t.Start(i);
        }
        // Wait for half a second, to allow all the
        // threads to start and to block on the semaphore.
        //
        Thread.Sleep(500);
        // The main thread starts out holding the entire
        // semaphore count. Calling Release(3) brings the
        // semaphore count back to its maximum value, and
        // allows the waiting threads to enter
        // the semaphore,
        // up to three at a time.
        //
        Console.WriteLine("Main thread calls Release(3).");
        _pool.Release(3);
        Console.WriteLine("Main thread exits.");
```

```
    }
    private static void Worker(object num)
    {
        // Each worker thread begins by requesting the
        // semaphore.
        Console.WriteLine("Thread {0} begins " +
        "and waits for the semaphore.", num); _pool.WaitOne();
        // A padding interval to make the output more orderly.
        int padding = Interlocked.Add(ref _padding, 100);
        Console.WriteLine("Thread {0} enters
        the semaphore.", num);
        // The thread's "work" consists of sleeping for
        // about a second. Each thread "works" a little
        // longer, just to make the output more orderly.
        //
        Thread.Sleep(1000 + padding);

        Console.WriteLine("Thread {0} releases
        the semaphore.", num);
      Console.WriteLine("Thread {0} previous semaphore count: {1}",
num, _pool.Release());
    }
}
```

Mutex

Когда двум или более потокам нужно произвести доступ к разделяемому ресурсу одновременно, системе необходим механизм синхронизации для того, чтобы гарантировать использование ресурса только одним процессом. Класс Mutex — это примитив синхронизации, который предоставляет эксклюзивный доступ к разделяемому ресурсу только для одного процесса. Если поток получает мьютекс, второй поток, желающий получить этот мьютекс, приостанавливается до тех пор, пока первый поток не освободит мьютекс.

Можно использовать метод WaitHandle.WaitOne для запроса на владение мьютексом. Поток, владеющий мьютексом, может запрашивать его в повторяющихся вызовах Wait без блокировки выполнения. Однако поток должен вызвать метод ReleaseMutex соответственное количество раз для того, чтобы прекратить владеть мьютексом. Если поток завершается нормально во время владения мьютексом, состояние мьютекса задается сигнальным, и следующий ожидающий поток становится владельцем мьютекса. Если нет потоков, владеющих мьютексом, его состояние является сигнальным.

Открытые конструкторы

Mutex - конструктор	Перегружен. Инициализирует новый экземпляр класса Mutex

Открытые свойства

Handle (унаследовано от WaitHandle)	Получает или задает собственный дескриптор операционной системы

Открытые методы

Close (унаследовано от WaitHandle)	При переопределении в производном классе освобождает все ресурсы, занимаемые текущим объектом WaitHandle
CreateObjRef (унаследовано от MarshalByRefObject)	Создает объект, который содержит всю необходимую информацию для разработки прокси-сервера, используемого для коммуникации с удаленными объектами
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object
GetHashCode (унаследовано от Object)	Служит хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-таблице
GetLifetimeService (унаследовано от MarshalByRefObject)	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра
InitializeLifetimeService (унаследовано от MarshalByRefObject)	Получает служебный объект срока действия для управления средствами срока действия данного экземпляра
ReleaseMutex	Освобождает объект Mutex один раз
ToString (унаследовано от Object)	Возвращает String, который представляет текущий Object
WaitOne (унаследовано от WaitHandle)	Перегружен. В случае переопределения в производном классе, блокирует текущий поток до получения сигнала текущим объектом WaitHandle

Защищенные методы

Dispose (унаследовано от WaitHandle)	При переопределении в производном классе отключает неуправляемые ресурсы, используемые WaitHandle, и по возможности освобождает управляемые ресурсы

Finalize (унаследовано от WaitHandle)	Переопределен. Освобождает ресурсы, удерживаемые текущим экземпляром. В языках C# и C++ для функций финализации используется синтаксис деструктора
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего объекта object

Многопоточное приложение. Способы синхронизации

Приводимый ниже пример является многопоточным приложением, пара дополнительных потоков которого получают доступ к одному и тому же объекту (объекту синхронизации). Результаты воздействия образующих потоки операторов наглядно проявляются на экране консольного приложения.

```
using System;
 using System.Threading;
namespace threads12
 {
 class TextPresentation
 {
 public Mutex mutex;
public TextPresentation()
 {
  mutex = new Mutex();
 }
public void showText(string text)
 {
 int i;
 // Объект синхронизации в данном конкретном случае –
 // представитель класса TextPresentation. Для его обозначения
 // используется первичное выражение this.
 //1. Блокировка кода монитором (начало)
 Monitor.Enter(this);
 //2. Критическая секция кода (начало)
 //lock(this) {
//3.Блокировка кода мьютексом (начало)
// mutex.WaitOne();
//
    Console.WriteLine("\n" + (char)31 + (char)31 + (char)31 + (char)31);
    for (i = 0; i < 250; i++)
     {
     Console.Write(text);
     }
   Console.WriteLine("\n" + (char)30 + (char)30 + (char)30 + (char)30);
 //mutex.ReleaseMutex();
```



```
 //3.Блокировка кода мьютексом (конец)//
 //2. Критическая секция кода (конец)
// }
 //1. Блокировка кода монитором (конец)
 Monitor.Exit(this);
 }
 }
class threadsRunners
 {
   public static TextPresentation tp = new TextPresentation();
   public static void Runner1()
{
Console.WriteLine("thread_1 run!");
Console.WriteLine("thread_1 - calling
TextPresentation.showText");
tp.showText("*");
Console.WriteLine("thread_1 stop!");
}
public static void Runner2()
{
Console.WriteLine("thread_2 run!");
Console.WriteLine("thread_2 - calling
TextPresentation.showText");
tp.showText("|");
Console.WriteLine("thread_2 stop!");
}
static void Main(string[] args)
 {
     ThreadStart runner1 = new ThreadStart(Runner1);
     ThreadStart runner2 = new ThreadStart(Runner2);
    Thread th1 = new Thread(runner1);
     Thread th2 = new Thread(runner2);
    th1.Start();
     th2.Start();
}
}
}
```

Рекомендации по недопущению блокировок потоков

- Соблюдать определенный порядок при выделении ресурсов.
- При освобождении выделенных ресурсов придерживаться обратного (reverse) порядка.
- Минимизировать время неопределенного ожидания выделяемого ресурса.

• Не захватывать ресурсы без необходимости и при первой возможности освобождать захваченные ресурсы.

• Захватывать ресурс только в случае крайней необходимости.

• В случае, если ресурс не удается захватить, повторную попытку его захвата производить только после освобождения ранее захваченных ресурсов.

• Максимально упрощать структуру задачи, решение которой требует захвата ресурсов. Чем проще задача – тем на меньший период захватывается ресурс.

Задания для самостоятельной работы

I. Реализовать задачу «Поставщик-потребитель». Поставщик генерирует данные и отправляет их в общий буфер. Размер буфера ограничен. Потребитель забирает данные из буфера. Поставщик не может положить данные, если в буфере нет свободных мест. Потребитель не может взять данные, если буфер пуст. Поставщик и потребитель не могут одновременно работать с буфером.

Варианты заданий по представлению буфера(a), по средствам синхронизации(b), по задаче(c).

a. ***Представление буфера.***

1) Стек.
2) Очередь.

b. ***Средства синхронизации.***

1) Критические секции (lock).
2) Семафоры и мьютексы.
3) Монитор(Enter-Exit).

c. ***Задача.***

1) Создать приложение с двумя дополнительными потоками писателей и двумя потоками читателей. Писатели в случайные моменты времени помещают записи, содержащие номер потока писателя и номер записи в буфер, читатели в случайные моменты времени удаляют записи, делая об этом сообщения.

2) Создать многопоточное приложение с одним потоком-читателем, удаляющим данные из буфера. Главный поток в случайные моменты времени порождает потоки-писатели, которые в случайные моменты времени помещают данные в буфер, если в структуре имеется свободное место, или самоуничтожаются с соответствующим сообщением.

3) Создать многопоточное приложение с одним потоком-писателем, который в случайные моменты времени помещает данные в буфер и сообщает об этом. Главный поток в случайные моменты времени порождает потоки - читатели, которые в случайные моменты времени удаляют данные из буфера с соответствующим сообщением. Каждый поток-читатель завершается после удаления заданного числа данных.

4) Создать многопоточное приложение, в котором главный поток в случайные моменты времени порождает либо поток - читатель, который в случайные моменты времени удаляет данные из буфера с соответствующим сообщением, либо поток - писатель, который в случайные моменты времени помещает данные в буфер и сообщает об этом. Каждый поток – читатель завершается после удаления заданного числа данных. Каждый поток – писатель завершается после занесения заданного числа данных.

II. Задача «обедающие философы»

Пять философов предаются размышлениям, и время от времени пополняют истраченные силы в столовой. В столовой – один стол, на нём большая миска спагетти и пять золотых вилок. Для того, чтобы поесть спагетти философу нужны две вилки одновременно. Проект должен обеспечивать возможность каждому философу поесть. Каждый философ – поток; Каждая вилка – мьютекс.

Список литературы

1. Олифер В. Г. Сетевые операционные системы : учебник для вузов / В. Г. Олифер, Н. А. Олифер. – 2-е изд. – СПб. [и др.] : Питер, 2008. – 668 с.

2. Таненбаум Э. Современные операционные системы / Э. Таненбаум. – 2-е изд. – СПб. : Питер, 2002. – 1037 с.

3. Системное и прикладное программное обеспечение : учебно-методическое пособие / сост. : М.А. Артемов [и др.] : Воронеж. гос. ун-т. – Воронеж : ЛОП ВГУ, 2006. – 74 с.

4. Гордеев А. В. Системное программное обеспечение : учебник для студ. вузов, обуч. по специальностям: «Вычисл. машины, комплексы, системы и сети» и «Автоматизир. системы обраб. информ. и упр.» направления подгот. дипломир. специалистов «Информ. и вычисл. Техника» / А. В. Гордеев, А. Ю. Молчанов. – СПб. и др. : Питер, 2002. – 734 с.

Учебное издание

Вощинская Гильда Эдгаровна,
Артемов Михаил Анатольевич

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 1

Учебно-методическое пособие для вузов

Редактор И.Г. Валынкина

Компьютерная верстка О.В. Шкуратько

Подп. в печ. 21.11.2012. Формат 60×84/16.
Усл. печ. л. 5,2. Тираж 50 экз. Заказ 728.

Издательско-полиграфический центр
Воронежского государственного университета.
394000, г. Воронеж, пл. им. Ленина, 10. Тел. (факс): +7 (473) 259-80-26
http://www.ppc.vsu.ru; e-mail: pp_center@ppc.vsu.ru

Отпечатано с готового оригинал-макета
в типографии Издательско-полиграфического центра
Воронежского государственного университета.
394000, г. Воронеж, ул. Пушкинская, 3. Тел. +7 (473) 220-41-33