СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящий лабораторный практикум имеет целью систематизировать практическую часть (лабораторные занятия) по дисциплине «Системное программирование». Набор заданий охватывает достаточно большое число тем, большинство из которых можно отнести к трём основным направлениям:

– элементы программирования в однозадачной среде (файловая система, ввод-вывод, обработка прерываний);

– основы программирования приложений Win32 (событийное управление, ввод-вывод, доступ к ресурсам);

– многозадачное и многопоточное программирование, взаимодействие процессов в среде Win 32.

В данный практикум сознательно не были включены некоторые крупные тематические блоки, традиционно относимые к системному ПО, такие как теоретические основы операционных систем и теория трансляторов. Также не рассматриваются иные операционные системы (в первую очередь семейство Unix-систем) и аспекты и применительно к платформе windows – технологии .NET. Это объясняется наличием в учебной программе соответствующих специализированных курсов, что позволило сосредоточиться на базовом уровне освоения наиболее распространенной платформы и универсальных для большинства сред задачах системного уровня. Внимание же, уделяемое низкоуровневому программированию и, в частности, ОС MS-DOS, обосновано тем, что в результате обнаруживается наиболее простой и экономичный путь практического изучения ряда задач – как актуальных для специальных применений, так и свойственных более сложным платформам.

Описания лабораторных работ строятся, насколько возможно, по единообразной схеме: цель работы, краткие вводные теоретические сведения, приблизительный перечень вопросов для контроля усвоения темы, варианты заданий к лабораторной работе. Следует отметить, что информация в теоретических блоках описаний лабораторных работ не является исчерпывающей и не может заменить справочные пособия и другие источники.

Жестких ограничений на используемые языки и среды программирования не накладывается. Для заданий первой группы ожидается использование Ассемблера, однако частично они могут быть выполнены и с помощью языков высокого уровня. Задания, рассчитанные на среду Windows, могут выполняться любыми подходящими средствами при условии демонстрации взаимодействия с соответствующими системными интерфейсами.

ЛАБОРАТОРНАЯ РАБОТА №1
РАБОТА С ФАЙЛАМИ

Цели работы:

1) изучение функций доступа к файлам и управления файлами в среде MS-DOS;

2) реализация алгоритмов поиска файлов;

3) закрепление практических навыков программирования средствами Ассемблера с использованием функций MS-DOS.

1.1. Краткие теоретические сведения

Данные на внешних носителях принято организовывать в виде файлов.

Файл — упорядоченный набор данных, пригодный для использования прикладными программами. Удобно представлять файл как совокупность *данных* (используются прикладными программами) и *метаданных* (данные о размещении данных, используются системными программами).

Для организации хранения данных, предоставления доступа к ним и обеспечения другого сервиса служит *файловая система* (ФС). В это понятие входят как структуры для собственно хранения данных и метаданных, так и включенные в ПО средства работы с ними.

Типичной для файловых систем является иерархическое (древовидное) построение: файлы могут объединяются в *каталоги* (*директорий – directory, папка – folder*), причем сами каталоги также могут быть вложенными. Таким образом, каталог также является файлом, но специального вида – предназначенным для хранения информации о включенных в него файлах и других каталогах.

Помимо каталогов, предусматриваются и другие виды специальных файлов: файлы логических устройств, файлы-ссылки, файлы – метки тома и т.д. В отличие от специальных типов объектов файловой системы «обычные» файлы программ или данных называют *регулярными*.

Файлы принято идентифицировать по их *именам*. Учитывая наличие каталогов и подкаталогов, для однозначной локализации файла требуется также и *путь* к нему (*path*) – перечисление всей цепочки каталогов.

Для ФС Microsoft характерна отдельная иерархия каталогов для каждого *логического* диска. Логический диск может соответствовать физическому накопителю или одному из его *разделов*, реже – виртуальному устройству. С точки зрения системы каждый логический диск рассматривается как самостоятельное устройство и идентифицируется буквой: A, B, C и т.д.

Таким образом, полный (*абсолютный*) путь к файлу будет начинаться от «буквы» логического диска, например: A:\directory1\directory2\...\file.ext. Относительный путь начинается от текущего каталога в файловой системе.

Традиционными для MS-DOS и ранних версий Windows были файловые системы FAT (более подробно см. лабораторную работу №2). Первоначально

имена ограничивались восемью символами собственно «имени» и тремя символами «*расширения*» (*extention*), причем набор символов ограничивался латинскими буквами, цифрами и некоторыми дополнительными символами: +, –, _ и т.д. Позже в именах были допущены символы национальных алфавитов, а затем введены и «*длинные*» имена, причем уже с использованием символов Unicode. Пределом длины «длинного» имени считается 255 символов, а имени вместе с путём – 260 символов.

Традиционные имена «8.3» содержат буквы только в верхнем регистре, и приведение их выполняется автоматически, поэтому для пользователя имена не чувствительны к регистру букв. «Длинные» имена хранятся с учётом регистра букв, однако их сравнения всё равно регистронечувствительны. Поэтому имена будут соответствовать своим файлам независимо от регистра букв в них, а в одном каталоге не могут быть два файла, имена которых различаются только регистром букв.

Для доступа к файлу он должен быть предварительно открыт, все последующие действия будут выполняться над системным объектом «открытый файл». Первоначально в MS-DOS применялись *FCB* (File Control Block) – структуры, содержащие информацию о файле и выполняемой над ним операции; в ту же структуру могли заноситься и результаты операции. Позже был введен более удобный подход, основанный на использовании *файловых дескрипторов* (*file descriptor*). Дескриптор представляет собой целое число, получаемое после открытия или создания файла и однозначно идентифицирующее его. Полученный дескриптор действует до закрытия файла. Вся служебная информация об объекте остается скрытой от прикладной программы, что упрощает работу. Сейчас FCB-ориентированные функции считаются устаревшими и поддерживаются лишь для совместимости с ранее написанным ПО, а основным методом доступа в современных ОС является дескрипторный.

Сервис MS-DOS для работы с файлами представлен рядом функций прерывания 21h. Ниже перечислены некоторые из них (дескрипторный доступ):

AH=3Ch – создание файла с усечением содержимого;
AH=3Dh –открытие существующего файла;
AH=5Bh – создание файла (если не существует) или открытие;
AX=6C00h – создание или открытие файла с выбором поведения функции с посредством дополнительных флагов;
AX=716Ch – то же с поддержкой длинных имён;
AH=42h – позиционирование в файле;
AH=3Fh – чтение из файла в буфер в памяти;
AH=40h – запись данных из буфера в файл;
AH=3Eh – закрытие файла.

Имеются также функции, работающие с объектами файловой системы без открытия файлов. Некоторые из них перечислены ниже:

AH=43h – получение или установка атрибутов файла (имеет подфункции);
AH=56h – переименование файла;
AX=7156h – переименование с поддержкой длинных имён;

AH=57h – получение или установка даты и времени для файла;
AH=41h – удаление файла;
AH=39h и 3Ah – создание и удаление каталога;
AX=7139h и 713Ah – то же с поддержкой длинных имён;
AH=47h и 3Bh – получение и установка текущего каталога;
AX=7147h и 713Bh – то же с поддержкой длинных имён;
AH=4Eh и 4Fh – поиск (начало и продолжение поиска) файла по шаблону, результат возвращается в области *DTA*;
AX=714Eh и 714Fh – поиск с поддержкой длинных имён.

1.2. Контрольные вопросы

1. Хранение файлов и организация доступа к ним.
2. Способы доступа к файлам.
3. Атрибуты файлов.
4. Функции MS-DOS для работы с файлами.
5. Открытие и закрытие файлов.
6. Чтение данных из файла.
7. Запись данных в файл.
8. Поиск файла.
9. Создание файлов.
10. Переименование файлов.
11. Удаление файлов.

1.3. Варианты заданий

1.3.1. С клавиатуры вводятся имя исходного файла и имя файла для хранения результата. Необходимо при помощи функции поиска файлов убедиться в существовании исходного файла и отсутствии результирующего. Если исходный файл отсутствует, выдается предупреждение, и выполнение останавливается. Если результирующий файл уже присутствует, выдается предупреждение, и прежнее содержимое файла перекрывается.

После проверки необходимо открыть исходный файл, прочитать его блоками по 256 байт, зашифровать при помощи своей фамилии и зашифрованные данные сохранить в результирующем файле. Для шифрования использовать функцию XOR для соответствующих байтов фамилии и данных

1.3.2. С клавиатуры вводятся первые символы имени файла (например myfile) и имя файла результата. Необходимо найти все файлы, начинающиеся с данных символов (myfile.txt, myfile1.doc, myfile005.txt, …), по очереди открыть эти файлы и склеить их содержимое в результирующем файле, читая блоками по 256 байт.

1.3.3. С клавиатуры вводятся расширение имени для поиска файлов (например, txt) и имя файла результата. Необходимо найти все файлы с этим расширением и сохранить их имена в результирующем файле. В случае если

файл для хранения результата уже существует, его старое содержимое должно быть сохранено, а новые записи надо добавлять в конец файла.

1.3.4. Репозиторий имен файлов и директориев (повышенной сложности).

Репозиторий представляет собой динамическую структуру в памяти (предположительно дерево или иерархический список), содержащую информацию об именах файлов и каталогах, принадлежности файлов каталогам, вложенности каталогов. Поддерживаются «длинные» имена.

Для репозитория должны быть реализованы функции: включение файла (каталога) в репозиторий; исключение файла (каталога); форматированный вывод содержимого репозитория; сравнение содержимого репозитория и файловой системы.

В качестве дополнительной функции можно предусмотреть частичную выгрузку содержимого репозитория в файл.

Данное задание требует знакомства со средствами управления памятью.

ЛАБОРАТОРНАЯ РАБОТА №2
ФАЙЛОВАЯ СИСТЕМА FAT

Цели работы:

1) изучить структуру файловых систем FAT;
2) научиться работать с файловой системой FAT;
3) ознакомиться с типичными задачами, требующими прямого доступа к файловой системе.

2.1. Краткие теоретические сведения

Файловая система (ФС) FAT отличается простотой, что позволяет относительно несложно запрограммировать низкоуровневый доступ к ней. По этой причине она до сих пор применяется для сменных носителей (флэш-накопителей, флэш-карт) и специализированных устройств с ограниченными ресурсами, а ее поддержка присутствует практически во всех современных ОС.

Файлы в ФС FAT хранятся в виде связных списков блоков – *кластеров*. Кластер представляет собой объединение от 1 до 64 целых секторов и, следовательно, имеют размер от 512 байт до 32 Кбайт. Таким образом, каждый файл, в зависимости от размера, представляется одним или несколькими кластерами, образующими цепочку (*chain*).

Ссылки, обеспечивающие связность списка, хранятся отдельно от данных, в специальной структуре – таблице размещения файлов (*File Allocation Table – FAT*), размещение которой на диске фиксировано и заранее известно. Каждая ячейка таблицы позиционно соответствует одному кластеру и содержит номер следующего кластера в цепочке. Кроме того, следующие значения ячеек зарезервированы для специальных случаев: 0 – пустой кластер, FFFh – последний кластер в цепочке, FF7h – кластер помечен как дефектный и не исполь-

зуется (здесь приведены значения для FAT12, для прочих они аналогичны). Первые две ячейки таблицы объединены и содержат код типа носителя.

Наибольшее количество кластеров, представимых таблицей, зависит от разрядности ее ячеек. На практике встречаются 12-, 16- и 32-разрядные версии FAT – соответственно FAT12, FAT16 и FAT32 (в FAT32 реально используются лишь 28-разрядные значения, но хранятся они в 4-байтовых ячейках). Малое число кластеров на разделах большого объема приводит к слишком большому размеру каждого кластера и, как следствие, непроизводительному расходу дискового пространства. Для сравнения: у файловых систем, свободных от ограничений FAT, характерный размер распределяемых блоков составляет несколько килобайт.

Для обеспечения надежности на диске присутствует обычно две копии FAT. Функции ОС, работающие с файловой системой, модифицируют обе копии согласованно, а также распознают повреждение таблиц, но автоматическое восстановление не предусмотрено. В своих программах при непосредственных обращениях к FAT также необходимо поддерживать когерентность её копий.

Сканирование цепочки кластеров требует знания ее начала – номера первого кластера. Он содержится в записи директория, описывающего файл. Записи директориев – 32-байтовые, содержат имя файла, его атрибуты и некоторую служебную информацию, в том числе и номер первого кластера файла (часть записей может быть занято «длинными» именами файлов – они хранятся разбитыми на фрагменты в нескольких соседних записях). Поэтому первоначально нужно считать содержимое директория, найти в нем нужное имя файла и взять из этой записи номер первого кластера.

Сведения о размещении таблицы FAT, ее размере, размере кластера и другая служебная информация о разделе могут быть получены из *блока параметров BIOS* (*BIOS Parameters Block – BPB*). Он содержится вместе с кодом загрузчика раздела (*Boot Record*) в первом его *логическом* секторе. Если носитель не предполагает наличия разделов (например дискета), то этот же сектор является и первым *физическим*, и других загрузчиков носитель не имеет.

При наличии разделов (логических дисков) потребуется также обрабатывать *таблицы разделов – Partition Tables*, описывающие их размещение, тип и размеры. Первая таблица разделов содержится в «главной» загрузочной записи (*MBR – Master Boot Record*), в первом физическом секторе диска. Если среди разделов есть *расширенные*, то они сами включают вложенные разделы и поэтому в первом своем секторе также содержат таблицу разделов.

Работа с FAT требует организовать низкоуровневый доступ к диску, который осуществляется на уровне *секторов*. Сектор имеет фиксированный в пределах одного диска размер, стандартно 512 байт, и является минимальной одновременно передаваемой порцией данных при обмене с блочным устройством. *Логическими* или *относительными* называют секторы в пределах одного логического диска (раздела), они адресуются линейно, т.е. фактически порядковыми номерами или индексами. Нумерация начинается с 1 (в случае MS-DOS). *Физические* или *абсолютные* секторы отсчитываются в рамках всего

диска, и для них действуют два основных подхода к адресации. Традиционная адресация *CHS* (*Cylinder-Head-Sector*) предполагает три компонента идентификации сектора: цилиндр (номер дорожки), головка (поверхность) и номер сектора в дорожке. Для современных дисков компоненты CHS-адреса уже не соответствуют их реальной физической конфигурации и автоматически транслируются контроллером, поэтому их правильнее воспринимать как «логические» цилиндры и поверхности. Поддержка больших объемов дисков для CHS-адресации ограничена разрядностью компонентов адреса: 1024 «цилиндра», 256 «поверхностей», 63 «сектора». *Линейная* адресация (*LBA – Linear Block Addressing*) гибче, проще в использовании и более масштабируема в сторону увеличения числа секторов.

Логические секторы в MS-DOS читаются и пишутся прерываниями int 25h и 26h соответственно, при этом передаются параметры: AL – номер устройства; DX – номер первого обрабатываемого сектора; CX – количество секторов; ES:BX – адрес буфера в памяти. Признаком ошибки служит ненулевое значение флага CF. Кроме того, после возврата из прерываний стек всегда остается невыровненным: на его верхушке находится слово с кодом результата, которое необходимо извлечь.

Так как доступ по абсолютным адресам потребуется, скорее всего, только для секторов в начале диска, то достаточно будет традиционных функций BIOS. Сервис BIOS доступа к дискам представлен прерыванием int 13h, имеющим ряд функций, в том числе чтение и запись группы секторов. Код функции всегда передается в AH (02h – чтение, 03h – запись); DL – номер устройства; CH – младшие 8 бит номера цилиндра; DH – номер поверхности; CL – номер начального сектора (биты 5..0) и два старших бита номера цилиндра (биты 7..6); AL – счетчик секторов (не более чем на одной дорожке); ES:BX – адрес буфера в памяти. Признаком ошибки выполнения служит единичное значение флага CF, в этом случае код ошибки возвращается в AH.

Операции над внутренними структурами файловой системы потенциально опасны, поэтому рекомендуется ограничиваться, хотя бы на этапе отладки, съемными носителями, не содержащими уникальных данных. При выполнении большинства заданий для данной лабораторной работы потребуется дискета, отформатированная под FAT. Все задания, если не оговорено иное, выполняются для корневого каталога этой дискеты. Вместо реальной (физической) дискеты можно использовать виртуальный диск A:, созданный, например, программой vfd. Также может использоваться и флэш-накопителях, отформатированный в FAT.

2.2. Контрольные вопросы

1. Понятие файловой системы.
2. Структура файловой системы FAT.
3. Разновидности FAT.
4. Загрузочная запись.
5. Организация каталога.

6. Организация FAT-таблицы.
7. Организация покластерного доступа к файлу.

2.3. Варианты заданий

2.3.1. С клавиатуры вводится имя файла. Необходимо найти запись для данного файла в корневом каталоге; составить список кластеров, в которых записано содержимое этого файла; покластерно прочитать его. Содержимое файла вывести на экран или (по выбору) сохранить в файл result.txt при помощи функций MS-DOS из первой лабораторной работы.

2.3.2. При помощи анализа цепочек выяснить, какой из файлов корневого каталога занимает наибольшее число кластеров. Имя этого файла показать на экране.

2.3.3. С клавиатуры вводится номер кластера. Необходимо выяснить, какому из файлов выделен данный кластер, и вывести имя этого файла.

2.3.4. Восстановление удаленного файла (повышенной сложности).

Файл при удалении его средствами MS-DOS не уничтожается полностью. Соответствующая ему запись в каталоге помечается как «удаленная» (символ E5h в начале имени файла), и все принадлежавшие файлу кластеры помечаются в таблице FAT как свободные. Ссылка на первый кластер и содержимое этих кластеров не затрагивается, поэтому сохраняется теоретическая возможность восстановления файла. Однако успешность восстановления не гарантируется, так как информация о порядке следования кластеров теряется, а сами они могут быть присоединены к другим файлам.

ЛАБОРАТОРНАЯ РАБОТА №3
РАБОТА С ПАМЯТЬЮ

Цели работы:

1) изучить организацию памяти и механизмы ее распределения в DOS;
2) научиться работать с динамической памятью в DOS;
3) научиться создавать сложные динамические структуры.

3.1. Краткие теоретические сведения

В MS-DOS реализовано управление памятью без организации виртуального адресного пространства, но с распределением блоков переменной длины. Базовые механизмы ОС опираются только на средства *реального* режима процессоров x86, и лишь для работы с «верхней» памятью используются особенности адресации 286+ и переключение в защищенный/виртуальный режим.

Память в MS-DOS представляется как непрерывный массив, распределяемый отдельными блоками. Блоки могут начинаться только на границе пара-

графа, поэтому для задания блока достаточно его сегментного адреса. Блоки следуют непосредственно друг за другом и составляют связный список. Вся доступная для распределения память должна быть включена в эти блоки, в том числе и свободная. Последний блок в списке соответствует нераспределенному нефрагментированному остатку памяти.

Каждый блок предваряется *управляющим блоком* (*MCB — Memory Control Block*), который занимает ровно один параграф и содержит поля:

байт 0 – опознавательный маркер блока: 5Ah ('Z') – для последнего блока списка, 4Dh ('M') – для всех остальных;

байты 1..2 – сегментный адрес PSP программы – владельца блока (в MS-DOS адрес PSP играет роль *PID* – идентификатора загруженной программы);

байты 3..4 – размер блока в параграфах, при отсутствии «зазоров» между блоками в списке это значение является «относительной» ссылкой на следующий блок (размер самого MCB в этом поле не учитывается!);

байты 8..15 – имя программы – владельца блока.

Поля PID и имени владельца для пустых блоков ожидаются пустыми. Для используемых блоков корректность их значений не гарантируется: обычная их интерпретация справедлива в случаях, если блок был выделен стандартным способом обычной прикладной программе. Блоки, выделенные DOS для своих нужд, имеют PID = 0008h и имя «владельца», начинающееся с комбинаций "SD" (data) или "SC" (code). Внутри «системных» блоков памяти могут организовываться вложенные списки с аналогичными структурами суб-блоков и суб-MCB. Маркеры суб-MCB могут содержать значения: 'B' – дисковые буферы, 'D' – драйвер устройства, 'F' – системная таблица файлов, 'L' – данные логического диска, 'S' – внутренние стеки DOS; 'X' – кэш FCB и так далее (эта информация считается недокументированной).

Зная адрес одного из блоков, можно просканировать все последующие до конца списка. DOS хранит адрес первого блока в специальной структуре или *DIB – DOS Info Block* (она же List of Lists). Адрес DIB возвращает в регистрах ES:BX недокументированная функция int 21h AH=52h. Слово по адресу ES:BX-2 содержит сегментный адрес первого в списке MCB.

Помимо этого, в качестве стартовой точки можно использовать первый не принадлежащий системе блок памяти, выделенный прикладной программе, которая имеет PSP. Для этого выполняется сканирование параграфов до обнаружения первого содержащего действительный MCB. Признаки такого MCB:

– наличие маркера MCB;

– владельцем блока является он сам, то есть PID владельца указывает на следующий после MCB параграф;

– блок содержит PSP – начинается с команды int 20h.

Так как первой «нормально» загружаемой программой является обычно командный интерпретатор (как правило, COMMAND.COM), то можно также искать в MCB его имя.

Других структур для описания распределяемой памяти не предусмотрено, вся необходимая информация получается, в том числе и системой, путем сканирования списка MCB, поэтому любые изменения в ней отражаются на работе всей системы. Так, уменьшение значения в поле размера последнего MCB приводит к изменению объема памяти, контролируемой DOS, и появлению «невидимой» области.

В общем случае, ОС должна обеспечить: выделение блока памяти по запросу прикладной программы, освобождение блока, перераспределение ранее выделенного блока (изменение размера), а также учёт свободной и занятой памяти, дефрагментацию блоков и другие сервисные функции.

В MS-DOS предусмотрены три основные функции прерывания int 21h:

AH = 48h – выделение блока памяти. На входе: BX – размер блока в параграфах. На выходе: AX – сегментный адрес выделенного блока, BX – максимальный доступный размер этого блока.

AH = 49h – освобождение блока. На входе: ES – сегментный адрес блока.

AH = 4Ah – изменение размера ранее выделенного блока. На входе: ES – сегментный адрес блока, BX – новый размер блока в параграфах. На выходе: BX – максимальный доступный размер этого блока.

Все функции при возникновении ошибки устанавливают флаг CF и возвращают ее код в AX.

DOS выделяет только непрерывные блоки памяти, перемещение выделенных фрагментов, а также дефрагментация свободной памяти не предусмотрены, за исключением объединения пустых блоков в конце списка. После стандартного завершения программы выделенные ей блоки освобождаются автоматически, но только при условии, что их владелец в MCB был указан корректно. Дополнительные функции позволяют управлять стратегиями распределения памяти.

3.2. Контрольные вопросы

1. Организация памяти в MS-DOS.
2. Структура MCB блока.
4. Функции для создания, удаления и изменения размера блока.
5. Организация сложных динамических структур данных.

3.3. Варианты заданий

3.3.1. Создать двунаправленный список для хранения строк. Каждый элемент списка должен хранить указатель на блок с предыдущим элементом, указатель на блок со следующим элементом, указатель на блок, хранящий строку. Строки необходимо хранить в отдельных блоках памяти. Пользователю должны быть доступны следующие функции: добавить строку в начало списка, добавить строку в конец списка, удалить элемент списка с указанным номером, очистить список, показать содержимое списка (хранимые строки) на экране.

3.3.2. Создать динамический массив для хранения строк. Массив должен представлять собой указатели на блоки памяти, в которых хранятся строки. При добавлении элемента необходимо проверить, есть ли для него место. Если весь массив занят, его необходимо выделить заново, увеличив размер вдвое, и скопировать туда старое содержимое. Пользователю должны быть доступны следующие функции: добавить строку в конец массива, установить новую строку в определенный элемент массива (с указанным индексом), удалить строку с указанным индексом, очистить массив, показать содержимое массива (хранимые строки) на экране.

3.3.3. Создать двоичное дерево, хранящее числа. Каждый элемент дерева должен хранить число и «левую» и «правую» ссылки на нижестоящие элементы. Необходимо реализовать алгоритмы добавления числа в двоичное дерево и удаления числа из него. Пользователю должны быть доступны следующие функции: добавить число, удалить число, очистить дерево, показать содержимое дерева на экране. При выводе на экран показывать дерево в виде строки вида: содержимое родителя (содержимое левой ветви, содержимое правой ветви); для каждой из ветвей функция вывода должна быть вызвана рекурсивно.

3.3.4. Сформировать путем анализа списка MCB и вывести карту памяти, включая размер блоков и их владельцев.

3.3.5. Реализовать выделение и освобождение блоков памяти без обращений к функциям DOS.

3.3.6. Реализовать перераспределение блока памяти; в случае невозможности увеличить размер блока попытаться переместить его на новое место (текущее содержимое блока копируется).

3.3.7. Объединяет функции вариантов 4..6, дополняется интерактивным интерфейсом, позволяющим выбирать функции (повышенной сложности).

ЛАБОРАТОРНАЯ РАБОТА №4
ОБРАБОТЧИКИ ПРЕРЫВАНИЙ

Цели работы:

1) изучить поддержку прерываний и их использование;
2) научиться создавать и использовать собственные обработчики прерываний;
3) научиться создавать и отлаживать резидентные программы.

4.1. Краткие теоретические сведения

Прерывания (*Interrupt*) – предусмотренный в системе механизм изменения естественного порядка выполнения программ с целью обработки возникающих событий и для организации взаимодействия программ. В однозадачных системах прерывания можно рассматривать как элемент многозадачности.

В архитектуре ПЭВМ на основе процессоров x86 (реальный режим) выделяются внутренние прерывания процессора – *исключения* (*exception*), возникающие при исполнении инструкций, *маскируемые* и *немаскируемые* внешние прерывания. Среди исключений выделяются вызванные специальными инструкциями генерации прерываний int; их принято выделять как *программные* прерывания. Всего определено 256 прерываний, назначение части из которых документировано и закреплено за определенными источниками или программами-обработчиками, но большинство считаются зарезервированными для использования ОС и прикладными программами.

Для любых прерываний обработка их непосредственно процессором заканчивается выборкой адреса обработчика этого прерывания и передачей ему управления. Адреса обработчиков принято назвать *векторами* прерываний, они хранятся в *таблице векторов*, которая занимает первый физический килобайт адресного пространства x86.

Обработчик прерывания – часть кода, подпрограмма, ассоциированная с данным прерыванием и получающая управление при его возникновении. С одним прерыванием может ассоциироваться более одного обработчика, которые в таком случае образуют каскад (цепочку).

Вызов обработчика отличается от обращения к обычным подпрограммам тем, что в стеке помимо адреса возврата сохраняется также и регистр флагов. При выходе из обработчика его надо или восстановить из стека (инструкция процессора iret или retf), или отбросить (инструкция возврата с очисткой стека ret 2). Вызов обработчика и возврат из него всегда *дальние* (*far*) – адресá обработчика и точки возврата включают и смещение, и сегмент.

Резидентная программа, или *TSR* (*Terminate-and-Stay-Resident*) – в отличие от обычной, называемой *транзитной*, целиком или частично остается в памяти после того, как возвращает управление системе. Очевидно, выделение резидентных программ имеет смысл только в однозадачных системах.

Как правило, для резидентных программ установка обработчиков и перехват прерываний с целью их обработки и/или модификации – основной способ взаимодействия с другими программами.

В общем случае обработчик должен:

– сохранить текущий контекст выполнения, так как его вызов может произойти в любой момент времени и на фоне любой другой программы (если иметь в виду обработчик в резидентной программе);

– подготовить контекст для своего выполнения, в том числе проверить условия активизации;

– выполнить специфические функции, связанные с назначением обработчика;

– возможно, обратиться к сохраненному вектору прерывания;

– восстановить контекст, включая приведение в требуемое состояние аппаратных средств и контроллера прерываний, и вернуть управление.

Можно выделить несколько основных схем включения обработчика прерывания в каскад (цепочку).

1. Замещение – новый обработчик полностью перекрывает старый.

```
New_Handler PROC FAR
    push …    ;сохранение контекста
    …         ;функции обработчика
    pop …     ;восстановление контекста
    iret      ;возврат из обработчика
New_Handler ENDP
```

2. Предобработка – новый обработчик выполняет свои функции (например проверяет условия и модифицирует параметры) и затем передает управление старому.

```
New_Handler PROC FAR
    push …    ;сохранение контекста
    …         ;функции обработчика
    pop …     ;восстановление контекста
    jmp dword ptr cs:old_handler_off      ;передача управления старому обработчику
New_Handler ENDP
```

3. Постобработка – новый обработчик обращается к старому, вновь получает управление после его завершения и затем выполняет свои функции (например, модифицирует результат)

```
New_Handler PROC FAR
    push …    ;сохранение контекста
    …         ;функции обработчика
    pop …     ;восстановление контекста
    pushf     ;эмуляция «обычного» вызова обработчика
    call dword ptr cs:old_handler_off      ; обращение к старому обработчику
    iret      ;возврат из обработчика
New_Handler ENDP
```

4. Комбинированный – например, проверка условий, в зависимости от успешности которой выполняются подготовка параметров, вызов старого обра–ботчика и модификация результатов.

```
New_Handler PROC FAR
    push …    ;сохранение контекста
    …         ;функции обработчика
    pop …     ;восстановление контекста
    pushf     ;эмуляция «обычного» вызова обработчика
    call dword ptr cs:old_handler_off      ;обращение к старому обработчику
    push …
    …         ;продолжение функций обработчика
    pop …
    iret      ;возврат из обработчика
New_Handler ENDP
```

Кроме показанных, есть более изощренные способы организации пере–ходов и вызовов процедур. Например, можно разместить ячейки, хранящие адрес, непосредственно в коде процедуры-обработчика, предварив их кодом инструкции call far или jmp far с прямой адресацией. Бóльший практический интерес представляет следующая конструкция:

```
pushf
push <segment_to_jmp>
push <offset_to_jmp>
retf
```

Её результат эквивалентен «обычному» jmp far, но в отличие от косвенной адресации точки перехода не требуется обязательное наличие переменной, содержащей готовый адрес. Загружаемые в стек компоненты адреса могут быть константами, вычисляемыми значениями, находиться в регистрах и так далее в любых сочетаниях. Такой прием полезен при ограничениях на использование переменных: например, код может находиться в ПЗУ и не иметь сегмента данных, стек же, очевидно, всегда будет только в ОЗУ. Аналогичным образом получается и эквивалент инструкции call far.

Если новый обработчик был установлен на внешнее прерывание и нет передачи управления старому обработчику, то перед выходом необходимо разрешить работу внешнего контроллера прерываний. В противном случае новые прерывания останутся заблокированы.

```
mov al, 20h
out 20h, al
```

Однако если обращение к старому обработчику есть, то он уже выполняет эту операцию, и дублировать ее в своем коде не следует.

С осторожностью следует относиться и к разрешению или запрету прерываний посредством флага IF. Стандартно он сбрасывается перед передачей управления обработчику, а после завершения восстанавливается предыдущее значение всего регистра флагов. Аккуратность требуется, если сохраненные флаги игнорируются (возврат командой ret 2), а новое содержимое регистра FLAGS формируется самим обработчиком.

MS-DOS обеспечивает следующие основные функции прерывания int 21h для работы с прерываниями, их обработчиками и резидентными программами.

AH = 25h – установка вектора прерывания. Вход: AL – номер вектора, DS:DX – адрес нового устанавливаемого обработчика.

AH = 35h – получение вектора прерывания. Вход: AL – номер вектора. Выход: ES:BX – адрес текущего обработчика.

AH = 31h – завершение программы с сохранением ее резидентной. Вход: AL – код возврата, DX – размер оставляемой в памяти части программы (в параграфах).

Таким образом, получаем следующий «обобщенный» каркас резидентной программы, содержащей обработчик прерывания.

```
; начало резидентной части
Start:
    jmp Init
; область переменных
old_handler_off  DW ?
old_handler_seg DW ?
; код обработчиков
New_Handler PROC FAR
    …
    call dword ptr cs:old_handler_off
    …
    iret
New_Handler ENDP
; конец резидентной части
; транзитная часть – секция инициализации
Init:
    … ;другие подготовительные операции
; получение и сохранение адреса старого обработчика
    mov ah, 35h
    mov al, <номер_вектора>
    int 21h
    mov cs:old_handler_off, bx
    mov cs:old_handler_seg, es
; установка нового обработчика
    mov ah, 25h
    mov al, <номер_вектора>
    mov dx, seg New_Handler
    mov ds, dx
    lea dx, New_Handler
    int 21h
    … ;другие завершающие операции
; вычисление размера резидентной части и завершение программы
; с сохранением резидентной части в памяти (TSR)
    mov ax, 3100h
    mov dx, offset Init
    add dx, 0Fh
    shr dx, 1
    shr dx, 1
    shr dx, 1
    shr dx, 1
    int 21h
END Init
```

Метка Start: и инструкция jmp в начале программы избыточны – при явном указании метки при директиве END выполнение программы будет начинаться именно с нее. Однако этот элемент оформления может быть легко забыт, тогда стартовой точкой программы становится первая объявленная метка. Кроме того, в формате исполняемого файла COM (модель памяти TINY) возможности управлять начальной меткой нет, управление всегда передается на начало программы. Одновременно для COM-формата потребуется директива ORG 100h для пропуска сегмента PSP.

Драйверы – особый тип резидентных программ. Первоначально предназначались для управления внешними устройствами, в настоящее время обеспечивают функции ввода-вывода и работу *логических* устройств, как являющихся отображением реальных, так и виртуальных. В различных ОС для драйверов существуют специфические технологии и требования. Драйверы MS-DOS достаточно просты и представляют собой резидентные программы, загружаемые ранее любых прикладных программ (включая командный интерпретатор), имеющие определенную внутреннюю структуру и поддерживающие определенный интерфейс.

Интерфейс с драйвером опосредован системой: прикладная программа обращается к устройству, система транслирует это обращение в запрос к драйверу, который исполняет его и возвращает результаты. Для прикладной программы все преобразования прозрачны. Драйвер включает в себя *заголовок*, *очередь запросов, программу стратегии* и *программу прерывания*. Заголовок – структура с фиксированным набором полей, расположенная в начале кода драйвера и исчерпывающим образом описывающая его. Передаваемый драйверу запрос в виде структуры временно сохраняется стратегией в одноместной очереди, затем управление передается программе прерывания, которая и обеспечивает его выполнение. Перечень функций драйвера фиксирован: инициализация, чтение, запись и так далее, но поддерживаться могут не все из них. Адреса соответствующих процедур сводятся в таблицу в заголовке драйвера.

Кроме интерфейса запросов, драйвер при инициализации, подобно обычным резидентным программам, может устанавливать обработчики прерываний.

В MS-DOS *блочными* считаются устройства внешней памяти, на которых может быть размещена файловая система. Прочие устройства относятся к *символьным* и выглядят как файлы, не связанные ни с каким логическим диском. Блочные драйверы существенно сложнее символьных.

4.2. Контрольные вопросы

1. Понятия прерывания и его обработчика.
2. Таблица векторов прерываний.
3. Особенности обработки аппаратных и программных прерываний.
4. Основные прерывания BIOS и операционной системы DOS.
5. Способы перехвата и обработки (перекрытия) прерываний.
6. Каркас обработчиков прерываний.
7. Резидентные программы.
8. Каркас резидентной программы.
9. Обработка прерывания клавиатуры, таймера.
10. Обработка программных прерываний.

4.3. Варианты заданий

4.3.1. Перекрыть прерывание клавиатуры и сделать так, чтобы одна из букв (например “a”) подменялась другой (например “b”).

4.3.2. Перекрыть прерывание клавиатуры и сделать так, чтобы все согласные буквы игнорировались.

4.3.3. Перекрыть прерывание клавиатуры и сделать так, чтобы все гласные буквы заменялись на следующие по алфавиту.

4.3.4. Перекрыть прерывание клавиатуры и сделать так, чтобы вместо каждой введенной цифры вводилось две. Например, вместо “1” получалось “11”.

4.3.5. Перекрыть прерывание клавиатуры и сделать так, чтобы все введенные цифры заменялись на следующие по порядку.

4.3.6. Перекрыть прерывание клавиатуры и сделать так, чтобы регистр вводимых букв менялся с нижнего на верхний (или наоборот).

4.3.7. Перекрыть прерывание клавиатуры и сделать так, чтобы все вводимые цифры суммировались в переменной summa.

4.3.8. Перекрыть прерывание клавиатуры и сделать так, чтобы пробелы игнорировались.

4.3.9. Перекрыть прерывание клавиатуры и сделать так, чтобы вместо пробела вводился символ ввода.

4.3.10. Перекрыть прерывание таймера и сделать так, чтобы с заданным периодом (например, каждые пять секунд) на экран выводился заданный текст (например “go”).

4.3.11. Модифицировать прерывание 21h так, чтобы при выводе строки на экран функцией 09h регистр букв в строке менялся с верхнего на нижний (или наоборот).

4.3.12. Модифицировать прерывание 21h так, чтобы при выводе строки на экран функцией 09h в строке подсчитывалась сумма всех цифр, и эта сумма выводилась на экране вместо строки.

4.3.13. Модифицировать прерывание 21h так, чтобы при выводе строки на экран функцией 09h вместо строки показывалось число слов в ней.

4.3.14. Модифицировать прерывание 21h так, чтобы при вводе строки функцией 0Ah во введенной строке удалялись все пробелы.

ЛАБОРАТОРНАЯ РАБОТА №5
СЛОЖНЫЕ ОБРАБОТЧИКИ
И ВЗАИМОДЕЙСТВИЕ РЕЗИДЕНТНЫХ ПРОГРАММ

Цели работы:

1) ознакомиться со сложными и нестандартными ситуациями, возникающими при разработке и функционировании обработчиков;

2) научиться решать разрешать коллизии при перекрытии прерываний и взаимодействии обработчиков;

3) научиться организовывать интерфейс с резидентными программами.

5.1. Краткие теоретические сведения

Помимо типовой установки обработчиков, при создании резидентных программ приходится решать ряд задач, связанных с их спецификой и практически не встречающихся у обычных программ.

Основная особенность резидентной программы – событийный характер ее выполнения: вместо предсказуемого регулярного выполнения алгоритма имеют место асинхронные, происходящие в непредсказуемые моменты времени вызовы обработчиков прерываний. При этом их активизация может происходить на фоне любых других программ, находящихся в этот момент в заранее не известном состоянии. Состояние отдельных обработчиков самого резидента также может быть различным. Эти проблемы в целом характерны для параллельно выполняющихся и взаимодействующих программ, а обработчики прерываний и использующие их резидентные программы представляют собой элементы параллелизма в изначально однозадачной среде.

Кроме того, взаимодействие с резидентной программой является нетривиальной задачей ввиду отсутствия явно поддерживаемых MS-DOS интерфейсов для такого взаимодействия.

Повторное вхождение в программу (подпрограмму) может возникать либо в результате явной или неявной рекурсии либо, в случае обработчика прерывания, из-за повторного возникновения нового прерывания до завершения обработки предыдущего. Если рекурсивные вызовы по крайней мере прогнозируемы при анализе исходного текста, то повторная активизация обработчиков зависит от стечения «внешних» обстоятельств. Обработчики, допускающие повторный вызов («вхождение»), называют *реентерабельными*.

Обеспечение реентерабельности является сложной задачей. В первую очередь проблему создают глобальные переменные и обращения к глобальным ресурсам вообще. Обращение к такой переменной (или набору переменных) может быть прервано повторным вызовом того же обработчика, по завершении которого их значения могут оказаться изменены, что приведет к некорректной дальнейшей работе прерванного кода. В более общем виде это представляет собой проблему «*критического ресурса*», характерную для многозадачных систем.

Реентерабельность (и пригодность для рекурсий) обычных подпрограмм обеспечивается в большинстве языков использованием «автоматических» переменных (класс хранения *auto* в терминологии C/C++): они выделяются в стеке заново для каждой «копии» вызова подпрограммы. Однако полностью исключить использование глобальных переменных не всегда возможно (как минимум, регистры процессора являются своего рода «переменными», глобальными для всех программ), а доступное пространство в стеке для резидентных программ всегда ограничено, так как по умолчанию они вынуждены использовать стек той программы, на фоне которой был вызван обработчик. Для резидента можно зарезервировать собственный стек и переключаться на него при каждом вызове обработчика, но тогда сама эта область вместе с обслуживающими ее счетчиками окажется разделяемой между «копиями» вызовов, а динамическое выделение и управление несколькими стеками будет иметь очень сложную реализацию, особенно учитывая общие ограничения на доступную память.

В результате на практике обычно удается сделать реентерабельными лишь относительно простые обработчики. Для более сложных приходится ограничиваться «частичной» реентерабельностью – повторный вызов безопасен, но функции при этом не выполняются или выполняются частично. Типична реализация такого подхода с помощью флага активности (простейшего семафора):

```
is_active DB 0 ;переменная-семафор
Int_Handler PROC FAR
    cmp cs:is_active, 0 ;проверка семафора
    jz work
    iret ;выход – обработчик уже активен
work:
    inc cs:is_active ;закрыть семафор
    … ;функции обработчика
    inc cs:is_active ;открыть семафор
Int_Handler ENDP
```

Важно обеспечить *атомарность* (непрерывность) проверки значения семафора и его изменения, иначе сохраняется вероятность повторного вызова между этими инструкциями. В данном случае непрерывность обеспечивается предварительным запретом прерываний перед передачей управления обработчику, но на это можно рассчитывать не всегда. Система команд **x86** содержит инструкцию xchg – элементарный, гарантированно непрерываемый обмен двух значений, но воспользоваться им не всегда удобно. Заметим, что проблему представляют также и «вложенные» запреты и разрешения прерываний флагом IF, когда приходится корректно восстанавливать его значение на каждом «уровне».

Также возможно решить проблему повторного вхождения организацией «отложенных» вызовов. Для этого функционал делится между двумя обработчиками. Первый («инициирующий») реентерабелен, он лишь проверяет условия выполнения основных функций, и если это в данный момент невозможно, формирует соответствующий запрос и ставит его в очередь. Второй обработчик

(«исполнительный») активизируется периодически, например, по таймеру, и при наличии запроса начинает его выполнять; если не завершена обработка предыдущего запроса, он пропускает очередной цикла до следующей активизации. Роль простейшей «очереди запросов» может играть флаг или счетчик, указывающий на необходимость выполнения обработчика.

В любом случае необходимо минимизировать длительность как блокировки прерываний, так и выполнения своих обработчиков в целом.

Аналогичные проблемы свойственны не только пользовательским программам: реентерабельность стандартных «системных» обработчиков в общем случае также не гарантируется. Практически прерывания BIOS реентерабельны или имеют собственную блокировку критических участков, поэтому их вызов предположительно безопасен в любое время (документация этого не гарантирует). Прерывания DOS заведомо нереентерабельны.

Наиболее часто требуются обращения к прерыванию DOS int 21h. Так как полностью отказаться от него не всегда возможно, необходимо выбирать моменты для безопасного вызова. Имеются следующие основные возможности.

Контроль флагов InDOS (нахождение в обработчике функций DOS) и CriticalError (нахождение в обработчике критической ошибки). Адрес байта InDOS возвращается функцией int 21h AH=34h (необходимо вызывать заранее в секции инициализации так как нереентерабельность распространяется и на нее), CriticalError расположен в предыдущем байте памяти либо может быть получен недокументированным вызовом int 21h AX=6D06h. Начальные значения флагов – нулевые. Обработчики функций DOS инкрементируют InDOS при входе и декрементрируют при выходе. Обработчик критической ошибки устанавливает CriticalError и сбрасывает InDOS.

Обработка прерывания «холостого хода» int 28h позволяет определить период, когда DOS находится в состоянии ожидания ввода-вывода. Внутри обработчика int 28h можно безопасно обращаться к функциям DOS с номерами выше 0Ch независимо от состояния флагов.

В обоих случаях удобной будет описанная выше схема «отложенного вызова»: «исполнительный» обработчик устанавливается на прерывание холостого хода или проверяет возможность безопасного вызова DOS по флагам.

Кроме того, можно установить собственный монитор прерываний DOS и самостоятельно контролировать обращения к ее функциям. Так, безопасным считается совмещение консольного и файлового ввода-вывода. Но этот способ трудоемкий и не вполне надежный.

Кроме того, обработчики многих функций DOS предполагают, что вызывающая их в данный момент программа является также и текущей с точки зрения системы, однако в случае вызова из резидента это правило нарушается. Так как идентификатором программы служит адрес её PSP, необходимо, дождавшись возможности безопасного обращения к DOS, в первую очередь сохранить текущий PSP (функция AH=62h) и зарегистрировать в качестве текущего свой (функция AH=50h), а по окончании работы – восстановить сохраненный.

Для взаимодействия с резидентной программой нужно в общем случае обнаружить ее, а затем получить доступ к ее данным и процедурам. Возможны следующие варианты:

– прямое обращение к переменным и процедурам резидентной программы – необходимо знание ее внутренней структуры, то есть требуется или документирование, или наличие отдельной программы, обладающей этим «знанием» (например, транзитный запуск той же программы, из которой устанавливается резидент, с соответствующими ключами);

– захват и использование в качестве программного интерфейса определенных прерываний и/или их функций – универсально и эффективно, но может быть недостаточно гибко и создавать конфликты с другими программами;

– поддержка резидентом «горячих» клавиш – только интерактивное управление, но не интерфейс между программами.

В свою очередь обнаружение резидента также может быть решено несколькими способами:

– поиск сигнатуры (характерного достаточно длинного значения), содержащейся в определенном месте кода программы, путем сканирования памяти;

– обнаружение характерных для этого резидента изменений программной среды, в первую очередь прерываний и/или их функций;

– выделение специальных «диагностических» прерываний и/или их функций.

Наиболее эффективным, но достаточно сложным и трудоёмким решением для взаимодействия с резидентами является унифицированный интерфейс, основанный на использовании специально выделенного для него *мультиплексного* (или *мультиплексорного*) прерывания int 2Fh.

Основная идея состоит в наличии у резидента обработчика int 2Fh. Все эти обработчики каскадируются. В ходе установки каждая новая программа получает свободный идентификатор, который запоминается для сравнения. В дальнейшем этот идентификатор передается при обращениях к функциям int 2Fh: опознав его, программа выполняет функцию, иначе передает управление дальше по цепочке обработчиков. Кроме того, программа должна содержать по фиксированному адресу унифицированный блок параметров, которые описывают её и могут быть использованы для управления извне.

Отдельной, достаточно часто встречающейся задачей является деинсталляция резидентных программ. Она распадается на две подзадачи: деактивация (отключение) обработчиков и удаление резидентной части из памяти. Готового решения для каждой из них MS-DOS не предоставляет.

При деактивации обработчиков основную трудность представляет восстановление существовавшей ранее цепочки обработчиков. Как правило, это не удается сделать, если отключаемый обработчик был перекрыт следующим, установленным на то же прерывание, что можно определить, например, сравнивая реальный адрес обработчика со значением в таблице векторов прерываний. Тогда обработчик обычно просто деактивируется без удаления из цепочки – запрещается его выполнение, например с помощью флага:

```
old_handler_off  DW ?
old_handler_segDW ?
is_enabled       DB ?
Int_Handler PROC FAR
    cmp cs:is_enabled, 0
    jnz work
    iret
work:
    … ;функции обработчика
Int_Handler ENDP
```

Для отключения обработчика достаточно обнулить флаг разрешения, для включения – записать туда ненулевое значение.

Более сложный, но и более интересный способ – внутренняя «таблица переходов» (на примере единственного обработчика):

```
act_handler_off  DW ?
act_handler_segDW ?
old_handler_off  DW ?
old_handler_segDW ?
; общая часть обработчика – переход по таблице
Int_Handler PROC FAR
    jmp dword ptr cs:act_handler_off
Int_Handler ENDP
; рабочая часть обработчика
Handler_Work PROC FAR
    …
    call dword ptr cs:old_handler_off
    …
Handler_Work ENDP
```

Для включения и выключения такого обработчика в ячейки act_handler записывается точка входа в «рабочую» часть Handler_Work или сохраненный адрес старого обработчика. Этот подход хорош тем, что «таблицу переходов» и устанавливаемые в таблицу векторов обработчики могут быть компактно сгруппированы в начале программы, тогда при выгрузке резидента а памяти остаются только они, а «рабочие» функции могут быть отброшены.

Выгрузка резидента из памяти включает в себя освобождение занимаемого им блока памяти и всех выделенных ему ресурсов, включая открытые файлы. Хороший способ – завершение резидента стандартной функцией int 21h AH=4Ch. Однако DOS всегда предполагает завершение текущей программы, поэтому предварительно надо временно переключить адрес текущего PSP на PSP резидента. Альтернатива – выполнять освобождение «вручную».

В любом случае до выгрузки резидентной программы должны быть отключены все ее обработчики. Если обработчики не исключаются из цепочек полностью, а замещаются заглушками, полная выгрузка резидента невозможна: как минимум код заглушек должен оставаться в памяти.

Описанные способы подразумевают, что резидентная программа сама обеспечивает свое отключение и выгрузку. Если все это требуется сделать из внешней программы, не имеющей сведений о структуре резидента, то задача

существенно усложняется, и возможным становится, как правило, только достаточно грубое «общее» решение: периодические «моментальные снимки» среды, включающие таблицу векторов и карту распределения памяти. На основании этой информации выполняется откат системы к одному из предыдущих состояний. Может использоваться также и постоянный мониторинг функций распределения памяти и управления векторами прерываний.

5.2. Контрольные вопросы

1) Каскадные обработчики прерываний.
2) Проблемы идентификации обработчиков прерываний.
3) Мультиплексное прерывание.
4) Проблемы при удалении обработчиков прерываний.
5) Проблема реентерабельности обработчиков прерываний.
6) Повторное вхождение в прерывания DOS.

5.3. Задание

Запрограммировать клавиатурный шпион. Данная программа должна накапливать у себя в памяти вводимые пользователем символы и с заданной периодичностью (например 20 с.) сохранять их в файл spy.txt. Перехватываемые прерывания – клавиатура и таймер.

Должна быть обязательная проверка занятости DOS и данные должны сохраняться только когда система свободна. Должны быть решены проблемы повторного вхождения в прерывания DOS. Должна быть реализована возможность корректного удаления обработчика (допустимо не удалять резидент полностью, а заменять обработчик заглушкой). Должна также выполняться проверка повторной установки (предлагается использовать при помощи мультиплексного прерывания).

ЛАБОРАТОРНАЯ РАБОТА №6
ПРИЛОЖЕНИЯ WINDOWS С ИСПОЛЬЗОВАНИЕМ WIN 32 API

Цели работы:

1) изучить особенности приложений Windows и их структуру;
2) изучить функции Win 32 API и Window Messages для создания, выполнения, завершения приложений;
3) научиться создавать оконные приложения;
4) ознакомиться со средой программирования.

6.1. Краткие теоретические сведения

Окно (*window*) – один из наиболее важных объектов в ОС Windows. Окно представляет собой прямоугольную область, в которой приложение отображает выводимую информацию и из которой получает вводимую. Кроме того, окно участвует и в программных интерфейсах – оно является получателем *со-*

общений. Существует много разновидностей окон: диалоговые, окна многодокументных интерфейсов (MDI), окна сообщений, окна элементов управления. Присутствующие в системе окна образуют иерархию, в которой они могут находиться в отношениях владелец – подчиненный и предок – потомок. Окна имеют заголовки (*Title, Caption*), но в большинстве случаев обращение к ним происходит по описателям – *Handle* (системный тип HWND)

Класс окна (*Window Class*) определяет общие свойства и особенности поведения группы окон. Классы делятся на системные глобальные (к ним относятся, например, стандартные элементы управления), прикладные глобальные (реализуются в DLL, требуют регистрации, после чего могут использоваться любыми приложениями) и прикладные локальные (регистрируются как доступные единственному приложению). Именно класс связывает окно с его оконной процедурой. Классы идентифицируются, как правило, по их именам.

Для создания и регистрации классов служат функции RegisterClass() и RegisterClassEx(), для создания окон любых видов – функции CreateWindow() и CreateWindowEx().

Оконная процедура (*Window Procedure* или *WndProc*) является получателем всех сообщений, адресованных окну. Обычно представляет собой конструкцию вида switch – case, распознающую поступившие сообщения и выполняющую их обработку:

```
LRESULT CALLBACK
MainWndProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case M_CLOSE: //команда на закрытие окна
            DestroyWindow( hWnd);
            break;
        case WM_DESTROY: //подтверждение о готовности деинициализации
            PostQuitMessage( 0);
            break;
        case …
            break;
        default: //для всех остальных сообщений
            return DefWindowProc( hWnd, uMsg, wParam, lParam);
    }
    return 0;
}
```

Примечание. В данном примере правильнее было бы проверять адресата сообщений и завершать приложение, только если это его главное окно.

Сообщения (*Window Message*) лежат в основе механизмов событийного управления приложениями. Они представляют собой структуры, передаваемые между приложениями и несущими информацию о событиях, команды, данные и так далее. Для представления сообщений служит структура MSG, поля которой содержат: тип сообщения, пару параметров wParam, lParam, адресата

сообщения, время и экранные координаты его возникновения (имеет смысл для некоторых сообщений).

Сообщения передаются в приложение либо напрямую, либо через очередь. Прямая передача сообщения оконной процедуре является фактически косвенным вызовом этой процедуры, в том числе и из другого приложения. Для этого служит системная функция SendMessage(). Важно, что выполнение отправителя сообщения блокируется до завершения оконной процедуры.

Многие сообщения передаются не напрямую, а через очередь, что избавляет программы от взаимной зависимости и блокировок. Для этого служит функция PostMessage(). Кроме того, есть некоторые другие функции, сочетающие свойства синхронной и асинхронной отсылки, например SendNotifyMessage(), которая передает сообщение минуя очередь, но и не ожидает окончания его обработки, поэтому блокирующей не является.

Цикл обработки сообщений выбирает сообщения из очереди и перенаправляет их в соответствующие оконные процедуры. В упрощенном представлении, цикл образуют три основные функции:

```
MSG msg;
while (GetMessage(&msg,NULL,0,0)) //выборка сообщений из очереди
{
    TranslateMessage(&msg); //доп. обработка (трансляция) сообщения
    DispatchMessage(&msg); //передача сообщения в оконную процедуру
}
```

Здесь выбираются все сообщения для всех окон приложения, однако параметры GetMessage() позволяют накладывать на них фильтры.

Функция GetMessage() является блокирующей – при отсутствии сообщений в очереди она переводит приложение в состояние ожидания. При наличии подходящего по фильтрам сообщения оно удаляется из очереди, и его содержимое переносится в структуру MSG. Функция возвращает нулевое значение при получении единственного сообщения WM_QUIT, для всех остальных – ненулевое. Как следствие цикл прекращается после приема этого сообщения.

Типичное приложение Windows строится по следующей общей схеме:

1) WinMain() — головной модуль программы;

2) инициализация внутренних переменных и прочие подготовительные операции;

3) RegisterClass() – регистрация локального класса;

4) CreateWindow() – создание главного окна приложения (если требуется, могут быть и другие классы и окна;

5) ShowWindow(), UpdateWindow() – отображение главного окна;

6) цикл обработки сообщений: GetMessage() – TranslateMessage() – DispatchMessage().

7) освобождение выделенных ресурсов, деинициализация внутренних переменных;

8) оконная процедура, прямых вызовов нет, так как предполагается «обратный вызов» (callback) со стороны системы.

6.2. Контрольные вопросы

1. Стандартная структура оконного приложения, его основные функции.
2. Окно. Функция создания нового окна и ее параметры.
3. Что такое оконный класс и каким образом он может быть создан.
4. Каким образом для окна можно указать собственные курсор и иконку.
5. За что отвечают функции UpdateWindow() и ShowWindow().
6. Сообщение Windows (Win Message), его основные параметры. Очередь сообщений.
7. Способы отправки сообщений другому окну.
8. Цикл обработки сообщений, функция GetMessage(), завершение цикла.
9. Функции TranslateMessage(), DispatchMessage().
10. Назначение, логика работы и параметры оконной процедуры.
11. Сообщение WM_PAINT. Функция вывода текста в окне, ее параметры.
12. Сообщение WM_COMMAND. Данные, передаваемые в wParam и lParam в данном сообщении.
11. Создание собственного пункта меню и обработка нажатия на него.
12. Функции GetWindowRect() и InvalidateRect().
13. Таймер. Создание и удаление таймера. Обработка сообщений таймера.

6.3. Задание

Написать программу, которая создает окно с собственными курсором и иконкой. В главное меню добавить подменю с двумя пунктами Start и Stop. При нажатии на пункт меню Start в окне должна появиться движущаяся по середине окна надпись. После этого выбор пункта меню Stop должно приостанавливать движение надписи, а пункта Start – возобновлять. Для реализации движения использовать обработчик сообщения WM_PAINT и таймер.

ЛАБОРАТОРНАЯ РАБОТА №7
СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

Цели работы:

1) изучить основной набор элементов управления;
2) научиться создавать элементы управления;
3) научиться обрабатывать события элементов управления.

7.1. Краткие теоретические сведения

Все элементы управления (*Control*) являются окнами, подчиненными и дочерними (*child*) по отношению к окну, на котором они располагаются. Система предоставляет готовые классы для ряда элементов управления, таких как кнопки (Button), текстовые поля (Static), поля ввода (Edit) и так далее. Для

этих стандартных элементов определены специфичные сообщения и присутствуют оконные процедуры, обрабатывающие эти сообщения. Программы пользователя могут создавать собственные элементы, в этом случае необходимо описать их поведение и, возможно, зарегистрировать соответствующие «пользовательские» сообщения.

Элементы управления должны скрывать от прикладной программы все подробности возникающих с ними элементарных событий (например, движение «мыши» и нажатия ее кнопок). Оконная процедура элемента должна преобразовать их в событие более высокого уровня (например, нажатие кнопки). Естественно, это событие также представляется соответствующим сообщением.

Для унификации и упорядочивания работы с элементами управления каждому из них присваивается собственный идентификатор (он же является и идентификатором дочернего окна). Идентификаторы назначаются произвольно, но должны быть уникальны для каждого элемента. Они передаются окну-родителю как один из параметров сообщений.

Так, элемент «Кнопка» в результате завершенного нажатия генерирует сообщение WM_COMMAND, в параметре wParam которого присутствует идентификатор этого элемента.

Приведенный ниже вызов создает стандартный элемент управления «Кнопка» с заданными именем, координатами, размером и стилем.

```
hButton = CreateWindow(
        "BUTTON", //имя глобального системного класса «Кнопка»
        "Btn_Action", //имя кнопки
        WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON, //стиль окна
        x, y, width, height, //положение и размер кнопки
        hMainWnd, //главное окно
        (HMENU)id_Btn_Action, //ID элемента управления
        hMainInst, //приложение
        NULL
);
```

В оконной процедуре будет присутствовать проверка поступающих сообщений WM_COMMAND на это значение идентификатора элемента:

```
switch (uMsg) {
    …
    case WM_COMMAND:
        switch( LOWORD( wParam)) {
            …
            case id_Btn_Action:
                … //обработка нажатия кнопки Btn_Action
                break;
            …
        }
    break;
    …
}
```

Непосредственная работа с идентификаторами и сообщениями достаточно трудоёмка, поэтому при программировании сложных интерфейсов целесообразно создавать некоторые средства автоматизации назначения и анализа идентификаторов. Среды программирования обычно предлагают собственные решения для этого, например *карты сообщений* в Visual Studio.

Для большинства операций, выполняемых с элементами управления (получение и установка параметров, перемещение и так далее) используются специализированные сообщения, поверх которых могут быть предусмотрены «оберточные» функции для более удобного обращения к ним. В зависимости от их типа сообщения могут требовать передачи либо через очередь, либо непосредственно в оконную процедуру соответствующего элемента. Список сообщений достаточно велик и различается для каждого вида элементов управления.

Например, следующий вызов заставляет кнопку изменить свое состояние на «нажатое»:

```
SendMessage( hButton, BM_SETSTATE, (WPARAM)TRUE, 0);
```

Однако это приводит только к перерисовке самой кнопки, но не к появлению события ее нажатия, которое приходится генерировать отдельно:

```
PostMessage( hButton, BM_CLICK, 0, 0);
```

В итоге получаем «программно нажимаемую» кнопку.

7.2. Контрольные вопросы

1. Создание на окне элементов управления: Edit, Button, ListBox, ComboBox и т.д.
2. Получение доступа текст, введенный в Edit, и установка его нового значения.
3. Обработка нажатий на кнопки (элемент Button).
4. Формирование списка строк в элементе ListBox, получение доступа к выделенной в нем строке.
5. Формирование списка строк в элементе ComboBox, получение доступа к выбранной в нем строке.

7.3. Задание

Написать программу, которая создает окно с двумя элементами управления ListBox, одним Edit и четырьмя Button («Add», «Clear», «ToRight» и «Delete»). При нажатии на кнопку «Add» текст из Edit должен добавляться в первый ListBox, если такого текста там еще нет (необходимо выполнить проверку). Нажатие кнопки «Clear» очищает оба ListBox-а. Нажатие кнопки «ToRight» копирует выделенную строку из первого ListBox во второй (если там еще нет такой строки). Нажатие кнопки «Delete» удаляет выделенные строки в каждом из ListBox-ов.

ЛАБОРАТОРНАЯ РАБОТА №8
ГРАФИЧЕСКИЙ ВВОД-ВЫВОД В ОКОННОМ ПРИЛОЖЕНИИ

Цели работы:

1) изучить графическую подсистему GDI.
2) научиться использовать графику GDI.

8.1. Краткие теоретические сведения

GDI (*Graphics Device Interface*) представляет собой единый унифицированный интерфейс устройств (средств) отображения графической информации в Windows. Работа GDI базируется на понятии *контекста* устройства (*device context – DC*), который абстрагирует свойства реальных устройств: экран (окно на экране), принтер, битовый образ в памяти и так далее. Контекст идентифицируется его описателем (тип handle DC – HDC).

Получив контекст, программа может обращаться к нему с единым набором функций, причем поведение контекста (изображения контекста) должно быть одинаковым независимо от того, с каким устройством он связан.

Для получения контекста служат функции: GetDC(), GetWindowDC(), GetDCEx(). Они применимы для оконных (экранных) контекстов. Функции CreateDC() и CreateCompatibleDC() создают контексты, связанные с иными (не окна) объектами либо «контексты в памяти», не связанные с реальным устройством. Освобождение выполняется функциями ReleaseDC() для оконных контекстов и DeleteDC() – для остальных.

Для формирования изображения в контексте служат функции графических примитивов (например, Ellipse(), DrawText() и так далее) и графические инструменты (объекты). Основными инструментами являются: «перо» (Pen), «кисть» (Brush) и «шрифт» (Font). Логика GDI такова, что система при отображении примитива сама выбирает соответствующий инструмент: например, при изображении закрашенного полигона текущее «перо» будет использовано для прорисовки его контура, а текущая «кисть» – для заполнения внутренней области.

Объекты типа Bitmap или Metafile сами способны хранить изображение, поэтому могут служить «рабочей поверхностью» контекста. Объекты «палитра», «область отсечения» и некоторые другие дополнительно влияют на формирование изображения.

Для создания инструментов служат соответствующие функции GDI API, например CreatePen(), CreateBrush() и так далее. Для сложных объектов может быть определено несколько функций, различающихся параметрами и получаемым эффектом.

Контекст может иметь только по одному активному объекту каждого вида. Для их переключения служит функция SelectObject(), которая принимает экземпляр объекта (инструмента) и делает его текущим в заданном контексте; тип объекта определяется автоматически, и предыдущий объект этого типа выталкивается из контекста.

Общая схема отрисовки следующая:

– получение контекста;

– установка набора инструментов;

– формирование изображения из примитивов, в том числе со сменой используемых инструментов;

– освобождение контекста.

Для оконных контекстов, которые после окончания их использования не удаляются, а лишь освобождаются, перед освобождением следует восстановить те же инструменты, которые были активными при получении контекста.

В типичном случае перерисовка содержимого окна инициируется сообщением WM_PAINT – оно сигнализирует, что текущее видимое содержимое в окне было утрачено, и система уже выполнила автоматическую перерисовку самого окна и его фона.

8.2. Контрольные вопросы

1. Контексты графических устройств.
2. Последовательность формирования изображения в окне.
3. Активные инструменты Pen и Brush, их создание и выбор. Параметры функций создания Pen и Brush.
4. Как могут быть нарисованы линия, эллипс, дуга, прямоугольник. Параметры функций, рисующих данные графические примитивы.
5. Каким образом может быть нарисован Polyline.
6. Самоотрисовывающаяся кнопка (OWNERDRAW). Как она может быть создана, как на ней рисовать собственный рисунок и как обрабатывать нажатия на такие кнопки.

8.3. Задание

Написать программу, которая создает окно с двумя элементами управления типа самоотрисовывающейся кнопки (OWNERDRAW), для каждой из которых должен быть задан какой-нибудь рисунок. При нажатии на первую из них в окне выводится рисунок, состоящие не менее чем из двадцати графических примитивов (линия, дуга, полигон, прямоугольник, эллипс – каждый из перечисленных примитивов должен быть использован хотя бы два раза). Вторая из кнопок должна стирать данный рисунок.

ЛАБОРАТОРНАЯ РАБОТА №9
ДИНАМИЧЕСОЕ ОТОБРАЖЕНИЕ ДАННЫХ НА ОКНЕ

Цели работы:

1) изучить возможности графической подсистемы.
2) научиться использовать расширенные возможности графики GDI.

9.1. Краткие теоретические сведения

Помимо описанных выше графических примитивов GDI работает также и с растровыми изображениями, то есть целыми массивами (матрицами) пикселей. В простейшем случае изображение может формироваться попиксельно с помощью функции SetPixel(), но обычно этот путь слишком медленный и неудобный. (Имеется также функция GetPixel() – доступ к отдельным пикселям изображения.)

GDI предоставляет ряд функций для манипулирования целыми фрагментами изображений:

BitBlt() – копирование прямоугольной области из одного контекста в другой без каких-либо преобразований;

StretchBlt() – копирование области с масштабированием по обеим осям (обычно с уменьшением, так как при растяжении изображение заметно портится);

MaskBlt() – перенос с маскированием части изображения;

PlgBlt() – перенос прямоугольного фрагмента в непрямоугольную область с соответствующим геометрическим искажением.

В ряде случаев формирование всего изображения заново может быть слишком длительным – например, различного рода анимация или изображения из очень большого числа элементов. В этом случае эффективно использование «теневых» контекстов или «контекстов в памяти». Такие контексты не имеют связи с реальным устройством, вместо этого рабочую поверхность для них образует битовая карта (bitmap) или метафайл. (Под метафайлом здесь понимается объект Metafile, фактически хранящий последовательность применявшихся к нему графических примитивов и способный их воспроизвести.)

«Теневой» контекст создается совместимым с текущим экранным режимом, и с ним ассоциируется «битовая карта»:

```
hShadowDC = CreateCompatibleDC( NULL);
hBitmap = CreateCompatibleBitmap( hPrimDC, width, height);
SelectObject( hShadowDC, hBitmap);
```

Здесь hPrimeDC – «реальный» контекст, в котором необходимо формировать изображение.

После создания «теневого» контекста в нем формируется необходимое изображение. Далее в процессе работы программы это изображение или его фрагменты переносятся на «основной» (видимый) контекст.

9.1. Контрольные вопросы

1. Использование таймера.
2. Теневые графические контексты.
3. Формат BMP файла.
4. Отображение BMP картинки.
5. Анимация движения.

9.2. Задание

В графическом редакторе сделать несколько (например четыре) изображений движущегося объекта небольшого размера (например 32×32 пикселя). Реализовать поочередное отображение их в заданной позиции в окне. При щелчке «мышью» начинается движение объекта из текущей позиции в позицию указателя «мыши» с поочередной сменой изображений для имитации фаз движения. При отрисовке используются «теневые» контексты.

ЛАБОРАТОРНАЯ РАБОТА №10
ОБМЕН СООБЩЕНИЯМИ МЕЖДУ ОКНАМИ, ОБРАБОТКА СОБЫТИЙ ВВОДА-ВЫВОДА

Цели работы:

1) изучить некоторые основные события, группы сообщений, системные вызовы, используемые при взаимодействии окон;
2) изучить основные события ввода-вывода, соответствующие им сообщения, порядок их обработки;
3) научиться передавать информацию между окнами.

10.1. Краткие теоретические сведения

При организации взаимодействия в программе может возникнуть необходимость использовать собственные сообщения, не относящиеся ни к одному из «системных» типов. Для этого предусмотрены два диапазона «пользовательских» типов сообщений — от значения WM_USER до 0x7FFF и от WM_APP до 0xBFFF. Сообщения из этих диапазонов могут использоваться прикладными программами для собственных целей.

В пределах одного приложения (одного локального оконного класса) сообщения диапазона WM_USER не требуют регистрации в системе, и приложение может произвольно выбирать и использовать их. При взаимодействии между приложениями требуется предварительная регистрация сообщения в системе функцией RegisterWindowMessage(). В качестве аргумента она принимает текстовую строку, идентифицирующую сообщение (она предполагается известной обоим взаимодействующим программам), и возвращает числовой идентификатор зарегистрированного сообщения.

Кроме того, полезным может быть сообщение WM_COPYDATA – передача блока данных в памяти. Передаваемые данные упаковываются в структуру COPYDATASTRUCT.

Остальные вопросы, связанные с данной лабораторной работой, относятся к созданию и функционированию отдельных элементов управления, что было рассмотрено ранее.

10.2. Контрольные вопросы

1. Сообщение WM_USER.

2. Регистрация собственных (пользовательских) сообщений.

3. Передача собственных (пользовательских) сообщений другому окну.

4. Обработка собственных (пользовательских) сообщений.

5. Обработка сообщения о нажатии, передвижении и отпускании кнопок «мыши». Параметры данных сообщений.

6. Создание элементов RadioButton и CheckBox. Обработка сообщений от данных элементов управления и изменение их состояния.

7. Создание группы RadioButton.

10.3. Задание

Написать две программы, каждая из которых создает окно. На первом из них должны быть созданы две группы RadioButton. В первой из них имеется выбор из трех цветов: красный, синий, зеленый. Во второй – из четырех типов примитивов: ромб, квадрат, круг, звезда. Также на первом окне должен быть создан Checkbox с надписью «Draw». Информация об изменениях состояния данных Checkbox и RadioButtons должна передаваться во второе окно. При щелчке мышкой по второму окну проверяется переданная информация о состоянии CheckBox. Если он не выбран, ничего не происходит; если он выбран, то в точке щелчка мышки рисуется выбранный во второй группе RadioButtons примитив цветом, выбранным в первой из групп.

ЛАБОРАТОРНАЯ РАБОТА №11
ИСПОЛЬЗОВАНИЕ ПОТОКОВ

Цели работы:

1) изучить средства обеспечения многозадачности Windows.
2) изучить средства управления потоками.
3) научиться работать с потоками и организовывать взаимодействие между ними.

11.1. Краткие теоретические сведения

Поток (*Thread*) – системный объект, соответствующий последовательности («потоку») команд, выполняемой независимо и асинхронно по отношению к другим подобным последовательностям. Поток – базовый объект, для которого планировщик задач ОС распределяет время центрального процессора. Каждый процесс имеет как минимум один главный (*первичный – primary*) поток. Главный поток может порождать подчиненные (*вторичные*) потоки, которые будут выполняться одновременно с ним, равно как и с потоками прочих процессов.

Многопоточность, свойственная Win 32 и ряду других современных ОС, удачно дополняет их многозадачность и существенно повышает гибкость системы. Процесс в Win 32 играет роль владельца ресурсов и «контейнера потоков» планирования ресурсов. Ресурсы процесса доступны всем его потокам, и все потоки одного процесса совместно используют его виртуальное

адресное пространство (но каждый поток имеет собственный стек). В то же время процессорное время распределяется именно между потоками, а не между процессами.

Поток может находиться в нескольких состояниях, начиная от его инициализации до завершения, но наибольший интерес представляют следующие:

– *выполнение* (*running*) или *активность* (*active*) – поток обладает всеми необходимыми ресурсами, включая процессорное время, и выполняется;

– *готовность* (*ready*) – поток обладает ресурсами для выполнения, но время ему не выделено, и он ожидает активизации в очереди планировщика;

– *ожидание* (*wait*), или *заблокированное* (*blocked*), или «спящее» (*sleep*) – поток не обладает требуемыми ресурсами и не может выполняться, он ожидает выполнения необходимых условий или наступления каких-либо событий.

Например, активный поток переходит в состояние ожидания после обращения к *блокирующему* системному вызову до его завершения, затем перемещается в очередь планировщика, откуда выбирается для следующей активизации. В простейшем случае переход в состояние ожидания произойдет при выполнении вызова Sleep().

Планировщик задач ОС в рамках принятой в Win 32 модели многозадачности выделяет процессорное время отдельным потокам *квантами*. Переключение потоков и изменение их состояний происходит по мере израсходования выделенных квантов и в соответствии с приоритетами потоков.

Потоки идентифицируются их *описателями* (*handle*), уникальными в рамках одного процесса, и идентификаторами (уникальны в системе). Однако большинство функций работают именно с описателями. Поток всегда сам является обладателем специального *локального* описателя, поэтому закрытие «внешнего» его описателя не приводит к удалению объекта. Одновременно поток как объект системы не может быть удален даже после прекращения его выполнения, если «внешний» описатель не был закрыт. Неверная работа с описателями может приводить при интенсивном создании и разрушении потоков к заметному накоплению «мусора» и дальнейшим ошибкам.

Новые потоки порождаются из *функций потока* с помощью вызова CreateThread(). Функция потока – одна из функций головной программы, соответствующая требованиям по типу, параметрам и формату вызова:

```
DWORD WINAPI ThreadProc( LPVOID pParam)
{
    … //действия, выполняемые в потоке
    return result;
}
```

Интерпретации как параметра, так и возвращаемого значения произвольны и зависят от конкретной программы. Например, параметр может быть адресом структуры или просто целым числом (используя приведение типа).

Все операции, выполняемые этой функцией, включая вызовы других подпрограмм, будут выполняться внутри созданного потока. Поток заверша-

ется после завершения его функции. Возвращаемое ею значение будет доступно после завершения потока до его окончательного разрушения.

Последним параметром CreateThread() передается указатель на ячейку типа DWORD, куда записывается возвращаемый идентификатор созданного потока. В Window 9x этот указатель не должен быть нулевым. В Windows NT такого ограничения нет.

Уже существующий поток может быть открыт вызовом OpenThread(), для этого необходимо знать его идентификатор.

Для корректного завершения потока служит функция ExitThread(), вызываемая «изнутри» потока, её аргументом является код возврата потока. Фактически результат ничем не отличается от стандартного завершения функции потока. Принудительное завершение потока, в том числе и «извне», возможно с помощью функции TerminateThread(), но при этом глобальные данные и другие ресурсы, с которыми он в этот момент работал, могут остаться в некорректном состоянии. Поэтому предпочтительнее организовать взаимодействие так, чтобы завершение потоков согласовывалось с их текущим состоянием, оптимально – обеспечивалось самим же потоком.

Кроме завершения, поток может быть приостановлен и вновь запущен функциями SuspendThread() и ResumeThread() соответственно. В действительности для потока ведется счетчик, при нулевом значении которого он может выполняться, при ненулевом (положительном) – останавливается. SuspendThread() увеличивает «счетчик остановок», ResumeThread() уменьшает его, но не ниже нулевого значения. Это позволяет делать «вложенные» приостановки потока без риска ошибочно разблокировать его, но требует следить за сбалансированностью приостановок.

Остановленный поток не участвует в планировании выполнения, и его разблокирование зависит только от описанного счетчика.

11.2. Контрольные вопросы

1. Понятие потока.
2. Создание потока, параметры вызова.
3. Что такое функция потока, ее параметры и их использование.
4. Функции приостановки и возобновления потока.
5. Функции Sleep(), SleepEx(),

11.3. Варианты заданий

В каждом из заданий на главном окне должны быть созданы две кнопки Start и Stop. Нажатие кнопки Start запускает указанные в задании потоки. Нажатие кнопки Stop приостанавливает их. Выполнение потоков можно возобновить повторным нажатием кнопки Start. При выполнении задания по возможности использовать одну поточную функцию, передавая ей необходимые данные как параметр.

11.3.1. Должны быть реализованы три потока, каждый из которых осуществляет передвижение собственной надписи по главному окну. Все надписи должны быть различны и двигаться с разной скоростью.

11.3.2. Главное окно должно быть разделено на четыре части. Также должны быть созданы четыре потока, каждый из которых раз в секунду изменяет цвет фона в своей части окна.

11.3.3. На главном окне должны быть созданы три элемента управления Edit. Необходимо создать три потока, каждый из которых раз в две секунды установит случайное число в соответствующий Edit.

11.3.4. При помощи трех потоков необходимо реализовать движение трех квадратиков по главному окну в случайных направлениях. Каждый поток должен двигать собственный квадратик.

11.3.5. При помощи пяти потоков реализовать падение пяти букв с различной скоростью по главному окну.

11.3.6. При помощи двух потоков необходимо реализовать движение двух шариков друг за другом по синусоиде.

11.3.7. При помощи двух потоков реализовать вращение двух палочек по экрану с различной скоростью.

11.3.8. Главное окно должно быть разделено диагональю на две части. Необходимо создать два потока, каждый из которых раз в полсекунды закрашивает соответствующую часть главного окно в случайный цвет.

11.3.9. При помощи трех потоков реализовать механические часы: первый поток должен двигать часовую стрелку, второй минутную и третий секундную.

11.3.10. При помощи трех потоков реализовать электронные часы: первый поток должен выводить значение часа, второй – минуты, а третий – секунды.

11.3.11. Реализовать шесть потоков, каждый из которых двигает свой шарик по окружности.

11.3.12. Реализовать четыре потока, каждый из которых в случайном месте рисует прямоугольник, ожидает полсекунды, стирает прямоугольник, через полсекунды рисует его в новой позиции и так далее.

11.3.13. Главное окно делится на три части. Необходимо реализовать три потока, каждый из которых рисует постепенно опускающуюся красную полосу в соответствующей части экрана.

11.3.14. Реализовать два потока, каждый из которых отрисовывает ProgressBar (постепенно закрашивающийся прямоугольник с индикацией процента закраски).

11.3.15. Реализовать восемь потоков, каждый из которых рисует постепенно удлиняющийся луч. Все лучи должны исходить из одной точки и быть направлены под разными углами.

ЛАБОРАТОРНАЯ РАБОТА №12
СИНХРОНИЗАЦИЯ ДОСТУПА К РЕСУРСАМ

Цели работы:

1) изучить предусмотренные в Win32 средства синхронизации и соответствующие системные объекты;

2) научиться синхронизировать ресурсы при помощи различных объектов синхронизации.

12.1. Краткие теоретические сведения

В многозадачной (многопоточной) среде возникает проблема одновременного доступа к одним и тем же ресурсам (данным) со стороны нескольких потоков. В случае если конкретный ресурс не допускает такого использования, возникает конфликт, требующий разрешения. Это называют задачей *синхронизации* (*взаимного исключения*).

Критический ресурс – ресурс (объект, данные), который в силу своей физической природы либо логики использования не может быть доступен одновременно нескольким пользователям. В зависимости от контекста речь может идти об определенном сочетании обращений или любых обращениях вообще (например, одновременное чтение допустимо, но одновременная запись или чтение и запись – нет).

Примерами критических ресурсов могут служить устройство вывода (одновременно выводимые данные нескольких источников будут перемешаны), записываемый файл (чтение до окончания записи дает неверные результаты), счетчик цикла (постороннее изменение нарушает работу цикла) и так далее.

Критическая секция – участок кода, выполняющий обращение или логически связную последовательность обращений к критическому ресурсу. В зависимости от контекста может быть удобнее представлять, что критическая секция одна и определяется связью ее с критическим ресурсом, либо однотипные критические секции принадлежат различным пользователям, но связаны посредством единого критического ресурса.

В этих терминах задача исключения сводится к обеспечению единственности пользователя, находящегося внутри критической секции, связанной с данным критическим ресурсом.

Проблема усугубляется еще и тем, что проверка условий возможности доступа должна быть неотделима от самого доступа или хотя бы блокировки его для других пользователей, то есть *атомарность*. Так как прикладная программа обычно не может обеспечить это самостоятельно, многозадачные системы обязательно предоставляют механизмы синхронизации.

Простейшее средство исключения в Windows – объект CriticalSection. Он представляет собой обычную структуру и не имеет глобальной идентификации, поэтому может использоваться только потоками одного процесса. Использование CriticalSection выглядит в общем случае следующим образом:

```
CRITICAL_SECTION cs;
InitializeCriticalSection( &cs);
…
EnterCriticalSection( &cs);
… //защищенная критическая секция
LeaveCriticalSection( &cs);
…
DeleteCriticalSection( &cs);
```

Вызов EnterCriticalSection – попытка входа в критическую секцию. Если секция уже занята другим потоком, то поток блокируется до момента ее освобождения. Повторное вхождение в критическую секцию допускается только для одного и того же потока, что помогает избегать самоблокировок. Блокировка управляется счетчиком, каждое вхождение в секцию увеличивает счетчик, освобождение – уменьшает, поэтому их количество должно быть сбалансировано.

Для синхронизации потоков разных процессов предусмотрены специальные объекты: Event, Mutex, Semaphore, WaitableTimer. Все они являются именованными и допускают глобальную идентификацию по именам.

Сам перевод потока в состояние ожидания осуществляется функциями WaitForSingleObject(), WaitForMultipleObjects() и их разновидностями. В зависимости от параметров эти функции блокируют выполнение потока до обнаружения одного или нескольких переданных им объектов в состоянии «готовности» (*signaled*).

Объект «*событие*» (Event) – наиболее простая разновидность. Объекты создаются функцией CreateEvent() и бывают двух типов: «ручные» и «автоматические». Открытие существующего объекта «событие» по его имени – OpenEvent(). Установка события (перевод в состояние signaled) происходит всегда явно функцией SetEvent(), сброс – для «ручных» событий явно, функцией ResetEvent(), для «автоматических» – автоматически при успешном выполнении Wait-функции. Также имеется функция PulseEvent() – временная установка события, активизация всех ожидавших его потоков и автоматический сброс. Кроме того, объекты Event ассоциируются с файлами при организации асинхронного ввода-вывода.

Объект «*мьютекс*» (*Mutex*) – простейший двузначный семафор для организации критических секций. Принято считать, что он «захватывается» потоком, и если мьютекс в этот момент уже занят, очередной захватывающий его поток блокируется. Подобно CriticalSection, Mutex допускает повторный захват, но только одним и тем же потоком. Создание нового мьютекса – вызов CreateMutex(), открытие существующего по имени – OpenMutex(), освобождение – ReleaseMutex(), захват с возможной блокировкой – Wait-функции.

Объект «*семафор*» (*Semaphore*) – отличается от мьютекса тем, что является счетчиком и может принимать множество значений от нуля и выше. «Занятым» считается семафор с нулевым значением, с ненулевым – свободным. При попытке опустить значение ниже нуля происходит блокировка. Работа семафора не зависит от того, разные потоки обращаются к нему или один и тот же. Создание нового семафора – вызов CreateSemaphore(), открытие существующего по имени – OpenSemaphore(), «подъём» счетчика (разблокирование) – ReleaseSemaphore(), проверка и «опускание», в том числе блокирование – Wait-функции.

Кроме специализированных объектов синхронизации Wait-функции могут работать также и с другими объектами, например:

– процессы и потоки – ожидание завершения;

– файлы – ожидание окончания текущей операции, и так далее.

12.2. Контрольные вопросы

1. Что такое синхронизация доступа к ресурсам и зачем она нужна.
2. Объекты синхронизации в Win 32.
3. Объект синхронизации CriticalSection, его использование.
4. Функция WaitForSingleObject, ее параметры и возвращаемые значения. Использование данной функции.
5. Объект синхронизации Event, его создание, уничтожение и использование. Параметры данных функций.
6. Объекты Event с автоматическим сбросом.
7. Объект синхронизации Mutex, его создание, уничтожение и использование. Параметры данных функций.
8. Объект синхронизации Semaphore, его создание, уничтожение и использование. Параметры данных функций. Особенности данного объекта синхронизации.
9. Отличие объекта синхронизации CriticalSection от объекта синхронизации Event, Mutex, Semaphore.

12.3. Варианты заданий

В каждом из заданий необходимо создать несколько потоков и защищенный ресурс. Каждый из потоков должен делать следующее: проверить, свободен ли защищенный ресурс; если занят, то дождаться освобождения; если свободен, то занять его, выполнить какие-то действия (указанные в задании),

сделать паузу на одну секунду и освободить ресурс. Если в задании указано два объекта синхронизации, то необходимо выполнить отдельную программу для каждого из них.

12.3.1. Каждый из трех потоков должен пытаться закрасить главное окно в свой цвет: первый – в синий, второй – в красный и третий – в зеленый. В результате каждую секунду цвет фона главного окна будет изменяться. Реализовать синхронизацию доступа к ресурсам через Event, а затем через CriticalSection.

12.3.2. Каждый из трех потоков должен пытаться закрасить главное окно в свой цвет: первый – в желтый, второй – в голубой и третий – в черный. В результате каждую секунду цвет фона главного окна будет изменяться. Реализовать синхронизацию доступа к ресурсам через Mutex, а затем через Semaphore.

12.3.3. На главном окне необходимо создать Edit. Каждый из трех потоков должен пытаться установить в данный Edit соответствующий текст: First, Second или Third. Реализовать синхронизацию доступа к ресурсам через Event, а затем через CriticalSection.

4. На главном окне необходимо создать Edit. Каждый из трех потоков должен пытаться установить в данный Edit соответствующий текст: String1, String2, String3. Реализовать синхронизацию доступа к ресурсам через Mutex, а затем через Semaphore.

12.3.5. На главном окне необходимо нарисовать движущуюся справа налево фигуру (например квадрат). Также необходимо создать два потока: первый из них будет опускать фигуру вниз, а второй – поднимать вверх. Синхронизацию доступа к ресурсам реализовать через Event, а затем через CriticalSection.

12.3.6. На главном окне необходимо нарисовать движущуюся сверху вниз фигуру. Также необходимо создать два потока: первый из них будет смещать фигуру влево, а второй вправо. Реализовать синхронизацию доступа к ресурсам через Mutex, а затем через Semaphore.

12.3.7. Создать четыре потока, каждый из которых будет пытаться вывести в центре окна свой текст: AAAA, BBBB, CCCC, DDDD. Реализовать синхронизацию доступа к выводу на окно через Event, а затем через CriticalSection.

12.3.8. Создать четыре потока, каждый из которых будет пытаться вывести в центре окна свой текст: XXXX, ZZZZ, TTTT, YYYY. Реализовать синхронизацию доступа к выводу на окно через Mutex, а затем через Semaphore.

12.3.9. Создать три потока, каждый из которых будет пытаться вывести в центре окна свой рисунок: звездочку, квадратик, закрашенный эллипс. Реализовать синхронизацию доступа к выводу на окно через Event, а затем через CriticalSection.

12.3.10. Создать три потока, каждый из которых будет пытаться вывести в центре окна свой рисунок: домик, дерево, ромбик. Реализовать синхрониизацию доступа к выводу на окно через Mutex, а затем через Semaphore.

12.3.11. Создать на окне элемент управления ListBox. Также создать два потока, каждый из которых будет добавлять в данный ListBox свой текст: First или Second. Реализовать синхронизацию доступа к ListBox через Event, а затем через CriticalSection.

12.3.12. Создать на окне элемент управления ListBox. Также создать два потока, каждый из которых будет добавлять в данный ListBox свой текст: First или Second. Реализовать синхронизацию доступа к ListBox через Mutex, а затем через Semaphore.

12.3.13. Создать три потока, каждый из которых будет двигать по окну слева направо паровозик. В каждый момент доступ к выводу на окно должен иметь только один поток. Реализовать синхронизацию доступа к выводу на окно через Event, а затем через CriticalSection.

12.3.14. Создать пять потоков, каждый из которых будет двигать по окну слева направо паровозик. В каждый момент доступ к выводу на окно должны иметь два потока. Реализовать синхронизацию доступа к выводу на окно через Semaphore.

12.3.15. Реализовать восемь потоков, каждый из которых рисует постепенно удлиняющийся луч. Все лучи должны исходить из одной точки и быть направлены под разными углами. В каждый момент должны двигаться только три луча. Реализовать синхронизацию доступа к выводу на окно через Semaphore.

ЛАБОРАТОРНАЯ РАБОТА №13
ПРИОРИТЕТЫ

Цели работы:

1) изучить систему приоритетов;

2) изучить средства управления приоритетами в ОС Windows;

3) практически изучить влияние приоритетов на выполнение приложений и способы управления выполнение приложений посредством приоритетов.

13.1. Краткие теоретические сведения

Процессорное время выделяется потокам в соответствии с их уровнем приоритета. Потоку с более низким приоритетом не выделяется время, если на него претендует поток с более высоким уровнем приоритета. Более того, процесс с более низким приоритетом прерывается до истечения кванта времени, если на процессор претендует более приоритетный поток.

Уровни приоритетов варьируются в диапазоне от 0 (низший) до 31 (высший).

Действующий приоритет каждого потока образуют три составляющие: *класс приоритета* процесса, *уровень приоритета* потока внутри класса приоритета процесса, *динамический* уровень приоритета.

Класс приоритета процесса и уровень приоритета потока внутри класса определяют базовый уровень приоритета потока. Привилегированными считаются приоритеты от 16 до 31, они резервируются за системными программами реального времени. Прикладная программа получить приоритет выше 15 не может.

Определены следующие классы приоритетов, которым соответствуют константы:

– *Idle* (простаивающий) – IDLE_PRIORITY_CLASS (4), процесс активизируется только при простое других процессов;

– *Normal* (нормальный) – NORMAL_PRIORITY_CLASS (7), большинство процессов в системе, в частности, все процессы пользователя; приоритет владеющего активным окном процесса повышается на 2 и составляет 9;

– *High* (высокий) – HIGH_PRIORITY_CLASS (13), системные процессы, реагирующие на соответствующие события;

– *Real time* (реального времени) – REALTIME_PRIORITY_CLASS (24), некоторые системные процессы в "особых случаях".

Внутри классов приоритетов процессов определены уровни приоритетов потоков:

– *низший* (THREAD_PRIORITY_LOWEST) – –2 от уровня класса;

– *пониженный* (THREAD_PRIORITY_BELOW_NORMAL) – –1 от уровня класса;

– *нормальный* (THREAD_PRIORITY_NORMAL) – равен уровню класса;

– *повышенный* (THREAD_PRIORITY_ABOVE_NORMAL) – +1 к уровню класса;

– *высший* (THREAD_PRIORITY_HIGHEST) – +2 к уровню класса;

– *простаивающий* (THREAD_PRIORITY_IDLE) – равен 16 для REALTIME_PRIORITY_CLASS и 1 для остальных классов;

– *«критический»* (THREAD_PRIORITY_TIME_CRITICAL) – равен 31 для REALTIME_PRIORITY_CLASS и 15 для остальных классов.

Для класса REALTIME_PRIORITY_CLASS может использоваться также расширенный диапазон значений – от –7 до +6. Начиная с Windows 2003 были добавлены два специальных значения: THREAD_MODE_BACKGROUND_BEGIN и THREAD_MODE_BACKGROUND_END, они связаны с дополнительными возможностями планировщика.

Динамический уровень приоритета образуется повышением базового уровня потока на две единицы при поступлении сообщений в его очередь; по истечении некоторого времени восстанавливается исходное значение. Временное повышение приоритета делается также и для долго не получавшего управление потока. Эти правила действуют только для потоков с уровнем приоритета не выше 15.

Для управления приоритетами выполнения процессов и потоков служат следующие функции.

GetPriorityClass() – получение текущего класса приоритета для процесса;

SetPriorityClass() – установка класса приоритета для процесса;

GetThreadPriority() – получение текущего приоритета выполнения потока;

SetThreadPriority() – установка приоритета выполнения потока.

В сбалансированной системе высокоприоритетные потоки выполняют, как правило, короткие операции, связанные с реагированием на события. Кроме того, само событийное управление процессами предполагает частое переключение в состояние ожидания. Это дает шанс на исполнения потокам с низким уровнем приоритета.

13.2. Контрольные вопросы

1. Понятия приоритета процесса и потока. Роль приоритетов в планировании выполнения задач.
2. Классы приоритетов. Краткая характеристика основных классов приоритетов.
3. Функция для получения всех выполняющихся в данный момент процессов. Ее параметры и использование.
4. Функция для получения всех модулей заданного процесса. Ее параметры и использование.
5. Основные функции для управления приоритетами.
6. Создание всплывающих меню.

13.3. Задание

Написать программу, на главном окне которой будет показан ListBox, в который должны быть занесены все доступные из выполняющихся в данный момент процессов и их приоритеты. При выборе какого-нибудь из них во втором ListBox-е должны быть показаны его модули. Также должно быть реализовано всплывающее при нажатии правой кнопки мышки меню, в которое должны быть занесены основные классы приоритетов. При выборе какого-либо

пункта данного меню приоритет выбранного в первом ListBox процесса должен измениться на заданный.

ЛАБОРАТОРНАЯ РАБОТА №14
РЕЕСТР WINDOWS

Цели работы:

1) изучение системного реестра Windows;
2) изучение API для работы с реестром;
3) практическое ознакомление с некоторыми задачи, связанными с работой с реестром.

14.1. Краткие теоретические сведения

Реестр (*Registry*) – специальная системная база данных, в которой приложения и операционная система могут сохранять информацию о конфигурации.

Системный реестр служит для осуществления ряда функций:

– хранение конфигураций оборудования и сведений об устройствах «Plug-and-Play»;

– хранение списка драйверов и их параметров;

– описания программных интерфейсов (например, интерфейсов COM-серверов);

– таблица ассоциаций файлов данных;

– хранение конфигурации и значений параметров программ;

– обслуживание различных административных программ, например, панели управления (Control Panel).

Реестр имеет иерархическую древовидную структуру. Узлы дерева называются *ключами* (*key*). Каждый ключ может содержать любое количество *подключей* (*sub-key*) и значений (*values*), причем и те, и другие организованы в виде неупорядоченных списков, элементам которых присвоены индексы, начиная с нуля. Различие между ключами и подключами в действительности условно, фактически все ключи являются подключами различного уровня нескольких предопределенных ключей. Значения ключей могут быть строковыми, двоичными и числовыми; также ключ может быть ссылкой на другой ключ. Каждый ключ идентифицируется его именем, уникальным относительно вышестоящего ключа; открытым ключам присваиваются описатели (handle) HKEY. Значения идентифицируются именами и индексами в списке.

Данные в системном реестре хранятся в двоичном виде. Для работы с ними приложения должны использовать специальные системные функции. Различают *hive* (букв. «улей») ключей, то есть их двоичный образ в структурах в памяти, называемых собственно реестром, и файлы данных реестра.

Для работы с ключом приложение должно открыть его. При открытии ключа необходимо указать открытый ранее ключ в качестве вышестоящего.

Система всегда предоставляет четыре предопределенных (*predefined*) ключа верхнего уровня, которые считаются открытыми всегда и могут использоваться как точки входа в реестр: HKEY_LOCAL_MACHINE – описание известных на этот момент конфигураций компьютера; HKEY_CLASSES_ROOT – описание текущей конфигурации машины, ссылка на одну из конфигураций HKEY_LOCAL_MACHINE); HKEY_USERS – описание всех имеющихся пользователей; HKEY_CURRENT_USER – описание текущего пользователя, ссылка на одного из пользователей HKEY_USERS; HKEY_CURRENT_CONFIG – текущий подключ Config (ссылка) ключа HKEY_LOCAL_MACHINE;

Система предоставляет ряд функций для доступа к реестру. Перечислим некоторые из них:

– RegCreateKey(), RegCreateKeyEx() – создание нового или открытие существующего ключа в реестре; создаваемый или открываемый ключ обязательно должен быть подключом уже открытого ключа;

– RegOpenKey(), RegOpenKeyEx() – только открытие существующего ключа, в остальном аналогичны предыдущим;

– RegCloseKey() – закрытие открытого ключа;

– RegDeleteKey() – удаление ключа;

– RegFlushKey() – выгрузка содержимого ключа в соответствующий файл реестра;

– RegSaveKey(), RegLoadKey() – выгрузка содержимого ключа в отдельный файл и загрузка из него;

– RegEnumKey(), RegEnumKeyEx() – получение списка подключей;

– RegEnumValue() – получение списка значений;

– RegQueryValue(), RegQueryValueEx() – чтение значения по его имени;

– RegQueryMultipleValues() – чтение нескольких значений.

Ввиду относительной сложности и громоздкости обращений к реестру можно реализовать собственные «оберточные» функции для доступа по именам и «путям» реестре, аналогично работе с файловой системой.

На пользовательском уровне для работы с реестром служит утилита regedit, поддерживающая как интерактивный, так и командный (выполнение пакетных файлов) режимы.

14.2. Контрольные вопросы

1) Назначение реестра
2) Структура реестра
3) Поддерживаемые типы данных в реестре
4) API для работы с реестром
5) Порядок и правила работы с реестром.

14.3. Варианты заданий

14.3.1. Поиск в реестре значений, в том числе по шаблону (для текстовых), и вывод в удобной форме результатов поиска.

14.3.2. Поиск в реестре неиспользуемых и/или недействительных ключей и значений, например ссылающихся на несуществующие файлы и пути.

14.3.3. Получение и модификация отдельных элементов реестра (например, чтение текущей конфигурации, изменение отдельных настроек, чистка «историй» программ и т.п.)

ЛИТЕРАТУРА

1. Бек, Л. Введение в системное программирование / Л. Бек ; пер. с англ. – М.: Мир, 1988. – 448 с.
2. Вильямс, А. Системное программирование в Windows 2000 для профессионалов / А. Вильямс ; пер. с англ. – СПб.: Питер, 2001. – 624 с.
3. Гордеев, А.В. Системное программное обеспечение / А.В. Гордеев, А.Ю. Молчанов. – СПб. : Питер, 2001. – 736 с.
4. Гук, М. Аппаратные средства IBM PC / М. Гук. – СПб : Питер, 1996. – 224 с.
5. Гук, М. Аппаратные интерфейсы ПК. Энциклопедия / М. Гук. – СПб. : Питер, 2002. – 528 с.
6. Джордейн, Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT / Р. Джордейн ; пер. с англ. – М. : Финансы и статистика, 1992. – 544 с.
7. Зубков, С.В. Assembler для DOS, Windows и UNIX. / С.В. Зубков. – 3-е изд. – М. : ДМК пресс, СПб. : Питер, 2006. – 608 с.
8. Касаткин, А.И. Профессиональное программирование на языке Си. Управление ресурсами : справ. пособие / А.И. Касаткин.– Минск. : Вышэйшая школа, 1992. – 432 с.
9. Кулаков, В. Программирование дисковых подсистем / В. Кулаков. – СПб. : Питер, 2002. – 768 с.
10. Кулаков, В. Программирование на аппаратном уровне : спец. справочник / В. Кулаков. – 2-е изд. – СПб. : Питер, 2003. – 848 с.
11. Мешков, А.В., Тихомиров Ю.В. Visual C++ и MFC / А.В. Мешков, Ю.В. Тихомиров ; пер. с англ. – 2-е изд., перераб. и доп. – СПб. : БХВ-Петербург, 2002. – 1040 с.
12. Просиз, Дж. Управление памятью в DOS 5 / Дж. Просиз ; пер. с англ. – М. : Мир, 1994. – 240 с.
13. Рихтер Дж. Windows для профессионалов / Дж. Рихтер ; пер. с англ. – СПб.: Питер, 2000. – 752 с.
14. Скляров, В.А. Программное и лингвистическое обеспечение персональных ЭВМ. Системы общего назначения : справ. пособие / В.А. Скляров. – Минск: Вышэйшая школа, 1992. – 462 с.

15. Скляров, В.А. Программное и лингвистическое обеспечение персональных ЭВМ. Новые системы : справ. пособие / В.А. Скляров. – Минск: Вышэйшая школа, 1992. – 334 с.: ил.

16. Соломон, Д. Внутреннее устройство Microsoft Windows / Д. Соломон, М.Е. Руссинович ; пер. с англ. – 4-е изд. – СПб. : Питер, Русская Редакция, 2005. – 992 с.

17. Сорокина, С.И. Программирование драйверов и систем безопасности : учеб. пособие / С.И. Сорокина, А.Ю. Тихонов, А.Ю. Щербаков. – СПб. : БХВ-Петербург, М.: издатель Молчанов С.В. – 2002. – 256 с.

18. Таненбаум, Э. Современные операционные системы / Э. Таненбаум ; пер. с англ. – 2-е изд. – СПб. : Питер, 2002. – 1040 с.

19. Таненбаум Э., Вудхалл А. Операционные системы. Разработка и реализация / Э. Таненбаум, А. Вудхалл ; пер. с англ. – 3-е изд. – СПб. : Питер, 2007. – 704 с.

20. Юров, В.И. Assembler : учебник для вузов / В.И. Юров. – 2-е изд. – СПб. : Питер, 2007. – 640 с.

21. Assembler : практикум / В.И. Юров. – СПб.: Питер, 2002. – 400 с.