Лабораторная работа №8

Многозадачность. Многопоточность. Взаимодействие процессов

Работа выполняется в системе программирования Borland Delphi 2.0 в среде Windows 95. Использование средств визуального программирования и классов VCL не допускается, то есть Delphi используется просто как 32-разрядный компилятор с языка Паскаль для операционной системы Windows.

Теоретические положения

Windows 95 является 32-разрядной многозадачной операционной системой. Для 32-разрядных приложений Windows обеспечивает:

- плоскую (не сегментированную) модель виртуальной памяти;
- организацию для каждого процесса изолированного адресного пространства с возможностью адресации 4 Гб виртуальной памяти;
- возможность выполнения нескольких потоков в рамках каждого процесса;
- выполнение всех потоков всех 32-разрядных процессов в режиме вытесняющей многозадачности;
- поддержку средств синхронизации потоков – критических секций, семафоров, мутексов, событий;
- обмен данными между процессами.

Виртуальная память

Механизм виртуальной памяти позволяет решить проблему нехватки оперативной памяти в вычислительной системе. Он позволяет "расширить" объем оперативной памяти ВС за счет других видов памяти, имеющихся в системе, например – за счет дисковой памяти. При этом часть адресов (виртуальных адресов) соответствует адресам блоков информации, находящихся действительно в оперативной памяти, другие же адреса соответствуют информации, физически находящейся на диске.

Операционная система перехватывает обращение программы к виртуальным адресам и заботится о том, чтобы требуемый блок информации физически оказывался в оперативной памяти, при этом другие блоки перемещаются из памяти на диск. Кроме того, виртуальный адрес заменяется физическим адресом, соответствующим реальному положению адресуемого блока информации в оперативной памяти. Часть этих функций реализуется аппаратно процессором, часть – программно кодом операционной системы.

Плоская модель памяти Windows

ОС Windows 95 предназначена для выполнения на процессорах, совместимых с Intel 80386, в защищенном режиме. При этом поддерживаемый процессором механизм сегментации не используется, т.е. для обращения к любым данным или коду программа использует только 32-разрядное смещение. Сегментные регистры процессора при этом содержат нулевые значения и не могут быть изменены (попытка изменить сегментный регистр пользовательским процессом вызывает аппаратное прерывание и аварийное завершение программы операционной системой как выполнившей недопустимую команду).

32-разрядный адрес, используемый в любой машинной команде программы, не является физическим адресом, то есть процессор не выставляет его на шине адреса (разрядность которой – тоже 32 бита) для обращения к памяти. Этот адрес подвергается расшифровке в соот-

ветствии с аппаратно поддерживаемым процессором механизмом страничного преобразования.

Для каждого процесса операционной системой организуется собственное виртуальное адресное пространство. Это означает, в частности, что один и тот же адрес в контексте разных процессов как правило ссылается на различные физические адреса в памяти. За счет этого один пользовательский процесс физически не может испортить данные, с которыми работает другой процесс, так как их адресные пространства не пересекаются, и процесс просто не может задать такой адрес, чтобы обращение шло к данным другого процесса. С другой стороны, изолированность виртуальных адресных пространств процессов порождает проблемы, связанные с обменом данными между процессами – один процесс не может просто подготовить блок данных и передать в другой процесс указатель на этот блок, как это делалось в предыдущих версиях Windows (и возможно до сих пор, но только для 16-разрядных процессов). Windows API предоставляет специальные средства для обмена данными между процессами.

Адреса в виртуальном адресном пространстве пользовательских (не системных) процессов в Windows 95 распределяются следующим образом (речь идет не о выделении самой памяти, а о возможных значениях АДРЕСОВ):

$0..$3FFFFF – не используются 32-разрядными приложениями, могут использоваться для совместимости с программами DOS и Win16 (младшие 4Мбайта);

$400000..$7FFFFFFF – используются процессом для загрузки кодов и размещения данных (примерно 2 Гбайта);

$80000000..$BFFFFFFF – системная память, разделяемая всеми 32-разрядными приложениями (1 Гбайт);

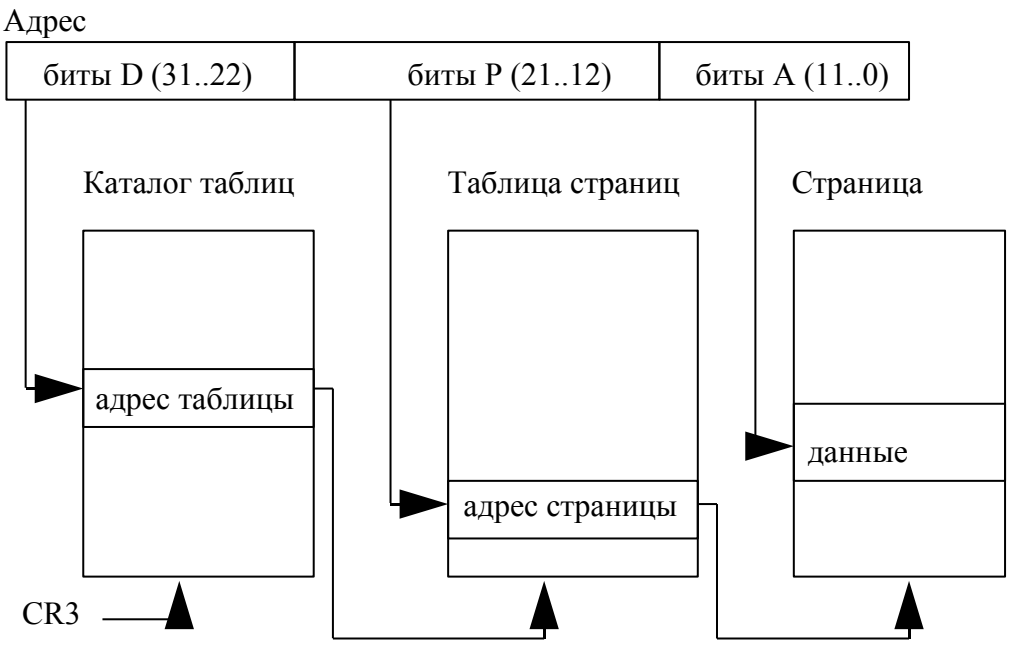
$C0000000..$FFFFFFFF – частная память процесса, используется ОС для обслуживания приложения (1 Гбайт).

Таким образом, приложения могут использовать для размещения данных и кода максимум два гигабайта адресного пространства из четырех возможных. Кроме того, приложение может иметь доступ к данным ОС через адреса в третьем гигабайте адресного пространства, что чревато нарушением работы ОС неверно работающим приложением. Модель памяти Windows NT не подразумевает такого разделения системной памяти приложениями, вследствие чего эта ОС значительно более устойчива к сбоям в приложениях.

Страничная организация виртуальной памяти

При страничной организации виртуальной памяти память выделяется процессам одинаковыми небольшими блоками. В процессорах Intel размер страниц составляет 4 кбайта. Каждая страница может находиться либо в оперативной памяти, либо на диске.

Для страничной организации виртуальной памяти операционная система поддерживает в памяти особые управляющие структуры, называемые таблицами страниц и каталогами страниц. Эти структуры аппаратно используются процессором для преобразования виртуальных адресов в физические. При этом биты 32-разрядного линейного адреса разделяются на три группы и используются следующим образом:



Каждая страница имеет размер 4 килобайта, и поэтому начинается с адреса, в котором младшие 12 бит нулевые (12 бит адресуют 4 кбайта).

ОС поддерживает таблицы страниц, в которых содержатся 32-разрядные дескрипторы страниц. Каждая таблица страниц сама по себе является страницей и занимает 4 кбайта. Старшие 20 бит дескриптора содержат старшие 20 бит физического адреса, с которого начинается страница (младшие 12 бит этого адреса нулевые). В оставшихся 12 битах дескриптора кодируются различные признаки, в частности – находится ли страница в памяти или на диске, можно ли производить запись в нее, была ли такая запись и т.д.

Кроме того, для каждого процесса ОС создает каталог таблиц страниц (тоже 4 кбайт, т.е. одна страница), с помощью которого процесс обращается к выделенным ему страницам памяти. Физический адрес этого каталога содержится в спецрегистре процессора CR3, и изменять его может только код, исполняемый в режиме ядра. Каждый элемент каталога таблиц содержит 32-разрядный дескриптор таблицы, в котором содержатся 20 старших бит ее адреса.

Когда в какой-либо машинной команде производится обращение по 32-разрядному адресу, процессор берет 20 разрядов адреса каталога из CR3, приписывает к ним десять битов поля D и два нуля и по полученному адресу извлекает дескриптор таблицы страниц. Затем берется 20 старших битов дескриптора, к ним приписываются 10 битов поля P, два нуля, и по полученному физическому адресу из таблицы страниц извлекается дескриптор страницы.

Если биты признаков дескриптора страницы содержат информацию о том, что страница находится в памяти, то происходит обращение к странице по физическому адресу, получаемому комбинацией 20 старших битов дескриптора и 12 битов поля A адреса. Если же страница отсутствует в памяти (находится на диске), то происходит аппаратное прерывание, обслуживая которое операционная система должна обеспечить подкачку нужной страницы с диска в память.

На самом деле процесс формирования физического адреса происходит без многократного обращения к таблицам дескрипторов в памяти, так как в процессоре есть средства для кэширования этой информации (хранения ее в сверхоперативной памяти внутри процессора).

Нетрудно видеть, что такой механизм управления памятью исключает проблему фрагментации физической памяти: страницы, образующие непрерывный блок в виртуальном адресном пространстве, могут быть как угодно разбросаны в физической памяти и загружаться при подкачке с диска в любое место. Вместе с тем, проблема фрагментации виртуальной памяти страничным механизмом не устраняется.

Сообщение WM_COPYDATA

Это сообщение позволяет передать блок информации от одного процесса другому. При этом в процессе обработки этого сообщения, и только в это время, передаваемый блок данных становится "видимым" из адресного пространства процесса-получателя этого сообщения и доступным для чтения. В документации к Windows сказано, что процесс-получатель не имеет права модифицировать информацию внутри блока, и он должен скопировать нужную информацию в самостоятельно выделенную область памяти, если эта информация нужна вне обработчика сообщения WM_COPYDATA.

Сообщение WM_COPYDATA всегда должно посылаться небуферизованным способом с помощью функции SendMessage. Отправлять его с помощью PostMessage недопустимо. Чтобы воспользоваться этим сообщением, нужно знать хэндл окна-получателя, относящегося к другому процессу, поэтому процессы предварительно должны "установить соединение" – например, с помощью широковещательной рассылки сообщений.

wParam - хэндл окна, отправившего данные; в принципе здесь может фигурировать хэндл любого окна процесса-отправителя.

lParam - указатель на структуру следующего вида:

```
TCopyDataStruct = packed record
  dwData: integer;
  cbData: dword;
  lpData: pointer;
end;
```

dwData – произвольное 32-разрядное значение, lpData – указатель на передаваемый блок данных, cbData – размер блока, адресуемого lpData.

Процесс-отправитель должен заполнить поля этой структуры, установив в lpData адрес блока данных, которые необходимо передать в другой процесс. Windows перед вызовом обработчика этого сообщения в другом процессе произведет необходимые действия по переносу данных в адресное пространство получателя, и в обработчике поле lpData окажется указывающим на блок передаваемых данных уже в адресном пространстве получателя. Этот указатель действителен только во время обработки сообщения, т.е. пока не завершилась функция SendMessage в процессе-отправителе.

С помощью этого сообщения можно передавать данные между любыми 16- и 32-разрядными приложениями.

Отображение файлов на память (File Mapping)

Само по себе отображение файлов на память работает следующим образом. Пользовательская программа просит ОС отобразить определенный дисковый файл в некоторый диапазон адресов своего виртуального адресного пространства, при этом создается особый управляющий объект, называемый отображением файла (file mapping). После этого при чтении и записи по адресам памяти, на которые отображен файл, программа читает и записывает содержимое соответствующего дискового файла. Это достигается за счет того, что страни-

цы памяти, соответствующие адресам отображения, выгружаются ОС не в системный файл подкачки, а в указанный файл; программа на самом деле читает и пишет в страницы памяти, в которые было считано содержимое файла при создании отображения и откуда информация будет возвращена в файл после закрытия отображения.

Сам по себе такой способ работы с файлом может иногда оказаться полезен, однако основное применение возможность отображения файлов в память находит не по прямому назначению (для работы с файлами), а для обмена информацией между приложениями. Это связано с тем, что один и тот же файл могут отобразить в память несколько приложений одновременно. При этом операционная система обеспечивает, чтобы изменения, сделанные в отображенном файле одним процессом, были отражены в образе файла, которую "видят" другие процессы. Фактически это достигается за счет того, что для отображения одного и того же файла в разных процессах используется одна и та же разделяемая физическая область памяти. При этом виртуальные адреса отображений в разных процессах могут не совпадать.

Для работы с отображением файла необходимы два объекта ОС: отображающий объект (file mapping) и образ файла (file view). Можно рассматривать образ файла как диапазон адресов виртуального адресного пространства процесса, используемых для работы с файлом, а отображающий объект – как страницы физической памяти, связанные с файлом. В связи с этим отображающий объект создается один раз в одном из взаимодействующих процессов, а образы файла – в каждом процессе с использованием существующего отображающего объекта.

Для создания отображающего объекта нужен хэндл открытого файла. Обычно файл специально создается для этой цели с помощью функции CreateFile, при этом желательно, чтобы создающий процесс создавал файл с эксклюзивным доступом (другие процессы не могли бы его открыть/испортить). У этой функции достаточно много параметров (см. Справку по Win32 API), возвращает она хэндл файла:

```
hFile := CreateFile(
      'myfile.map', // имя файла
      GENERIC_READ or GENERIC_WRITE, // доступ на чтение-запись
      0, // эксклюзивный доступ
      0, // атрибуты защиты, не поддерживается Windows 95, только NT
      CREATE_ALLWAYS, // создавать заново
      FILE_ATTRIBUTE_NORMAL);
```

Затем может быть создан отображающий объект:

```
hMapping := CreateFileMapping(
      hFile, // хэндл открытого файла
      0, // атрибуты защиты, не поддерживается Windows 95, только NT
      PAGE_READ_WRITE,
      0, 4096, // 4096 байт, если 0 - размер равен размеру файла
      'MyMapping' // имя-идентификатор объекта
      );
```

После этого может быть создан образ файла в адресном пространстве процесса, функция возвращает указатель на начало блока памяти размером, в нашем случае, 4096 байт:

```
pData := MapViewOfFile(
      hMapping, // хэндл отображающего объекта
      FILE_MAP_ALL_ACCESS, // доступ на чтение-запись
      0,0,0 // доступ ко всему отображению (можно задать часть)
      );
```

Далее можно производить в области, адресуемой указателем pData действия по чтению и записи информации. Например, можно переписать оттуда 256 байт в короткую строку с именем S:

Move(pData^, S, 256);

Другие процессы должны не создавать отображающий объект, а получать хэндл готового по имени:

```
hMapping := OpenFileMapping(0,false,'MyMapping');
```

Далее тоже вызывается MapViewOfFile.

При использовании данного механизма для обмена данными имеется возможность отказаться от использования собственного файла-основы и отображать системный файл подкачки. Для этого в функции CreateFileMapping в качестве первого параметра можно задать –1 (т.е. $FFFFFFFF). В этом случае обязательно необходимо задавать размер отображения.

Образ файла должен уничтожаться после использования функцией

```
UnmapViewOfFile(pData);
```

в качестве аргумента указывается указатель на образ файла, полученный с помощью MapViewOfFile.

Отображающий объект и сам файл (если он создавался) закрываются с помощью функции CloseHandle:

```
CloseHandle(hMapping);
CloseHandle(hFile);
```

Обычно разделение памяти осуществляется между тесно связанными программами, входящими в состав одного пакета. Например, некоторые сервисные функции пакета могут быть реализованы в отдельных программах-утилитах, работающих как самостоятельно, так и под управлением главной программы пакета. В этом случае проблемы "кому создавать отображающий объект" обычно не возникает – его создает главная программа, а вызываемая ей программа-утилита использует готовый объект через OpenFileMapping.

Если же разделение памяти должно осуществляться между равноправными приложениями (например – разными экземплярами одного и того же приложения), то можно использовать следующий подход:

1. Приложение пытается открыть объект с помощью OpenFileMapping.

2. Если возвращено нулевое значение, то объект не существует, поэтому его надо создать именно в этом экземпляре приложения, а затем удалить при помощи CloseHandle.

Обмен сообщениями между приложениями

Windows предоставляет возможность посылать сообщения между приложениями. При этом программист имеет возможность реализовать любой удобный для его программных продуктов протокол связи между процессами. С помощью посылки сообщений реализован также стандартный протокол обмена данными DDE (см. описание в отдельном файле). Обычно с помощью сообщений передают небольшие объемы данных, для кодирования которых достаточно двух 4-байтовых параметров сообщения.

При реализации собственного протокола обмена сообщениями необходимо учитывать, что Windows "согласится" доставлять только сообщения, коды которых лежат в диапазоне C000h .. FFFFh.

Хотя программист может "волевым решением" установить, что его программы будут пользоваться сообщениями, допустим, с кодом D000h и E000h, такой подход нельзя назвать удачным: никто не гарантирует, что какие-то программы параллельно не посылают друг другу сообщения с такими кодами. Поэтому Windows в виде функции RegisterWindowMessage

предоставляет возможность регистрировать сообщения по уникальному строковому идентификатору. Вероятность того, что различные группы приложений используют одни и те же строковые идентификаторы сообщений можно сделать достаточно низкой, если использовать в идентификаторах, например, название фирмы-производителя программного продукта.

Например, приложение может выполнить команду:

idMSG1:= RegisterWindowMessage('Borland_Data_exchange_message');

При этом функция возвращает значение, однозначно соответствующее указанной строке, т.е. если другое приложение вызовет эту же функцию с такой же строкой в качестве параметра, то будет возвращено то же число, а для другой строки число гарантированно будет другим. Если по каким-то причинам зарегистрировать сообщение не удается (например, слишком длинная строка, хотя в документации такие ограничения не упомянуты), то возвращается 0, иначе – число в диапазоне C000h .. FFFFh. Зарегистрированные сообщения остаются зарегистрированными до завершения сеанса работы в Windows (выключения или перезагрузки компьютера).

Часто для начального установления связи одно из приложений широковещательно рассылает сообщение-приветствие, в котором указывается хэндл окна-отправителя. Широковещательная рассылка достигается при указании в первом параметре SendMessage константы HWND_BROADCAST вместо хэндла окна. При получении приветствия (такое сообщение должно быть зарегистрировано как инициатором связи, так и предполагаемым партнером) другое приложение отвечает сообщением-подтверждением, в котором также указывается хэндл окна, которому следует направлять дальнейшие сообщения в процессе связи.

Например, два приложения зарегистрировали одни и те же сообщения:

```
msg1:= RegisterWindowMessage('Запрос');
msg2:= RegisterWindowMessage('Ответ');
msg3:= RegisterWindowMessage('Данные');
```

Затем первое приложение рассылает сообщение-запрос:

```
SendMessage(HWND_BROADCAST, msg1, hClientWindow,0);
```

При получении такого сообщения (его получат все окна первого плана, т.е. не имеющие окон-собственников, всех прилохений) второе приложение запоминает хэндл запрашивавшего окна (в нашем случае он передавался в wParam) и отвечает, указывая хэндл своего окна, например так:

```
hOtherWindow:=wparam;
PostMessage(wparam, msg2, hServerWindow, 0);
```

Приняв такое сообщение, окно hClientWindow первого приложения готово обмениваться сообщениями с окном hServerWindow второго приложения, причем в каждом приложении известен хэндл окна-партнера из другого приложения.

```
hOtherWindow:=wparam;
```

И теперь в любом приложении может быть послано сообщение:

```
PostMessage(hOtherWindow,msg3, 10,20);
```

где hOtherWindow – хэндл окна-партнера, сохраненный при установлении связи.

При двустороннем обмене сообщениями всегда предпочтительно пользоваться функцией PostMessage (если нет особых причин использовать SendMessage), так как при этом исключается рекурсивный вызов оконной процедуры. Дело в том, что если в ответ на SendMessage одного приложения второе приложение пошлет ответ через SendMessage, то оконная процедура в первом приложении запустится еще раз (рекурсивно), хотя предыдущий код,

посылавший SendMessage, еще не завершен. Это вполне допустимо, однако при плохом понимании программистом особенностей рекурсии способствует появлению логических ошибок в программе.

Буфер обмена

Приложения в принципе могут обмениваться информацией, помещая ее в системный буфер обмена и считывая оттуда. Однако это некорректно по отношению к пользователю. Дело в том, что буфер обмена предназначен для того, чтобы пользователь компьютера по своей инициативе помещал туда те или иные данные или считывал их оттуда (точнее – давал соответствующие инструкции используемым приложениям). Если приложения начнут записывать данные по своему усмотрению, а не по желанию пользователя, то это будет мешать его (пользователя) удобной и нормальной работе.

Для работы с буфером обмена существуют функции (подробности Clipboard overview в Справке):

OpenClipboard, EmptyClipboard, SetClipboardData, GetClipboardData, CloseClipboard, GetClipboardOwner, RegisterClipboardFormat, IsClipboardFormatAvailable и др.

Организация нескольких потоков в рамках процесса

Windows позволяет исполнять в рамках одного процесса несколько участков кода (потоков, нитей – threads) в режиме вытесняющей многозадачности. При этом при создании процесса порождается единственный (главный) поток, в котором могут быть выполнены действия для порождения дополнительных потоков. Рекомендуется строить программу так, чтобы все операции диалога с пользователем были сосредоточены в одном (главном) потоке программы, т.е. только там рекомендуется порождать все окна. В дополнительные потоки рекомендуется выносить длительные операции, не требующие общения с пользователем, например – сложный итеративный расчет или долгую операцию, связанную с обменом большим объемом информации с медленным устройством вроде дисковода или модема.

Для каждого потока Windows создает очередь сообщений, как только в потоке будет произведен вызов функции модулей USER или GDI. Это означает, что если поток не создает окон, нигде не рисует и не вызывает других функций USER и GDI, то очереди сообщений у него нет.

Все окна, созданные во время работы потока, получают сообщения именно через его очередь сообщений. Это означает, что в каждом потоке, создающем окна, должен быть СВОЙ цикл обработки сообщений, так как через главный цикл обработки сообщений программы идут только сообщения, относящиеся к окнам, созданным в главном потоке.

Если поток согласно логике работы программы ждет наступления какого-либо события, его следует переводить в состояние ожидания. Например, когда поток ждет прихода очередного сообщения, он переводится в состояние ожидания функцией GetMessage. Имеется возможность описывать в программе собственные события, управлять их состоянием (событие наступило или не наступило) и переводить потоки в состояние ожидания до тех пор, пока, например, другой поток не установит некоторое событие в состояние "наступило".

Поток создается при вызове функции

```
function CreateThread(
        lpThreadAttributes: Pointer;
        dwStackSize: DWORD;
```

lpStartAddress: Pointer;
lpParameter: Pointer;
dwCreationFlags: DWORD;
var lpThreadId: DWORD): THandle;

Назначение параметров:

lpThreadAttributes – атрибуты защиты, в Windows 95 не используется, передается nil.

dwStackSize – размер стека; если 0, то выделяется такой же стек, как у главного потока; при необходимости размер стека в дальнейшем автоматически увеличивается.

lpStartAddress – указатель на процедуру потока.

lpParameter – указатель (в принципе – любое 32-разрядное значение), который передается процедуре потока при запуске.

dwCreationFlags – равен 0 или CREATE_SUSPENDED, в последнем случае поток порождается в состоянии ожидания, из которого выводится функцией ResumeThread.

lpThreadId – указатель на 32-разрядную переменную, в которую будет записан идентификатор (не хэндл!) потока, нужный некоторым функциям управления потоками.

Возвращаемое значение – хэндл потока (или 0 при неудаче).

Функция потока объявляется как:

function MyThreadProc(data: pointer): dword; **stdcall**;

Поток завершается:

- когда функция потока завершается, при этом возвращаемое ею значение считается кодом возврата;

- когда внутри функции потока (или вызываемых ею функций) выполняется вызов функции ExitThread(code), где code – 32-разрядный код возврата;

- когда в любом потоке вызывается функция TerminateThread с указанием хэндля завершаемого потока и кода возврата; это является аварийным завершением, при котором не освобождаются загруженные завершаемым потоком DLL, занятый стек и т.д.

После завершения потока код возврата можно узнать с помощью функции

GetExitCodeThread(hThread, status);

где status – 32-разрядная переменная, в которую будет передан код возврата. Если поток еще исполняется (не завершен), то будет возвращено значение STILL_ACTIVE.

После завершения потока соответствующий системный объект продолжает существовать, и его хэндл можно удалить при помощи функции

CloseHandle(hThread);

Хэндл потока действителен только в рамках процесса, в котором он существует. При завершении процесса все потоки завершаются, их хэндлы освобождаются. В отличие от этого идентификатор потока действует глобально во всей системе.

Функция ExitProcess(exitCode) приводит к завершению всех потоков процесса и самого процесса.

Средства синхронизации потоков

Критические секции

Для защиты критических участков кода, выполняющихся в различных потоках ОДНОГО ПРОЦЕССА Windows поддерживает механизм критических секций. Так как критическая секция организуется с помощью особой структуры, располагаемой в изолированном адресном пространстве процесса, этот механизм непригоден для синхронизации параллельно выполняющихся процессов.

Чтобы воспользоваться критической секцией, программа должна создать экземпляр переменной типа TRTLCriticalSection (захватив под него память из кучи или просто создав глобальную переменную этого типа):

var section: TRTLCriticalSection;

Информация внутри структуры должна быть предварительно проинициализирована с помощью вызова функции

InitializeCriticalSection(section);

Далее потоки могут пытаться занять критическую секцию с помощью вызовов

EnterCriticalSection(section)

или

if TryEnterCriticalSection(section) then ...

Первая функция просто реализует захват критической секции вызывающим ее потоком, при занятости КС поток переводится в состояние ожидания и ждет освобождения КС.

Вторая функция вместо перехода в состояние ожидания возвращает логическое значение FALSE, если критическая секция занята (и входит в КС с возвратом TRUE, если КС свободна).

После завершения действий, выполняемых в КС, поток должен освободить КС, вызвав

LeaveCriticalSection(section);

Критических секций в программе может быть сколько угодно.

Если КС используется в программе только на одном из этапов ее работы, то занимаемые КС системные ресурсы можно освободить, вызвав функцию

DeleteCriticalSection(section);

Защищенные целые операции

Windows обеспечивает функции для производства простых операций с целыми переменными, в ходе выполнения которых гарантируется, что доступ к изменяемым переменным из других потоков производиться не будет. Если переменные находятся в разделяемой памяти, то их можно использовать для синхронизации процессов.

См. Справку по следующим функциям:

InterlockedIncrement, InterlockedDecrement, InterlockedExchange, InterlockedCompareExchange, InterlockedExchangeAdd.

Функции ожидания

Windows API содержит ряд функций, с помощью которых поток может быть переведен в состояние ожидания на определенное время или до наступления некоторых системных событий. Далее рассмотрено лишь ограниченное количество таких функций.

procedure Sleep(milliseconds:integer);

- Эта функция переводит вызывающий ее поток в состояние ожидания на указанное количество миллисекунд (32-разрядное целое). При указании в качестве параметра 0 происходит просто досрочная передача управления другим потокам, находящимся в состоянии готовности. Поток, переведенный в состояние ожидания с помощью этой функции, не может быть возобновлен до истечения указанного времени. Это означает, что если в этом потоке создавались окна и какому-либо окну посылаются сообщения, пусть даже с помощью SendMessage, эти сообщения не будут обработаны до завершения указанного времени.

function WaitForSingleObject(hObject:THandle; TimeOut:integer):dword;

- Функция ожидает перехода объекта синхронизации с хэндлом hObject (семафора, мутекса, события) во взведенное состояние. С помощью этой функции также можно ожидать завершения потока или процесса, передав соответствующий хэндл в качестве объекта ожидания. Выход из состояния ожидания производится также по истечении времени в миллисекундах, указываемого во втором параметре. Можно задать бесконечный тайм-аут при помощи константы INFINITE. При указании 0 в качестве тайм-аута сначала проверяется состояние объекта, и если он взведен, то это указывается в коде возврата. Функция возвращает значение, позволяющее судить о причинах выхода из состояния ожидания:

WAIT_OBJECT_0 – объект перешел во взведенное состояние (процесс или поток переходят в такое состояние после завершения);

WAIT_TIMEOUT – истек тайм-аут;

WAIT_ABANDONED – специфично для мутексов (см. ниже): поток, владевший мутексом, не освободил его в явном виде, но завершился, вследствие чего мутекс освободился и передан данному потоку.

function WaitForMultipleObjects(nCount: DWORD; lpHandles: pointer; bWaitAll: BOOL; dwMilliseconds: DWORD): DWORD;

Во втором параметре передается указатель на массив хэндлов объектов, в первом – количество хэндлов в этом массиве (не должно превышать MAXIMUM_WAIT_OBJECTS), третий параметр определяет, должны ли для окончания ожидания быть взведены все объекты массива (TRUE), или достаточно взведения лишь одного из них (FALSE). В четвертом параметре указывается тайм-аут ожидания. Возвращаемое значение:

WAIT_TIMEOUT – истек тайм-аут;

WAIT_OBJECT_0 .. WAIT_OBJECT_0 + (nCount–1) – объект с соответствующим номером перешел во взведенное состояние (процесс или поток переходят в такое состояние после завершения);

WAIT_ABANDONED_0 .. WAIT_ABANDONED_0 + (nCount–1) – специфично для мутексов, имело место для мутекса, хэндл которого указан в массиве в соответствующей позиции.

Семафоры

Windows поддерживает семафоры как объекты операционной системы. Семафоры могут использоваться для синхронизации между потоками. Семафор в Windows имеет целое значение от 0 до максимального, объявляемого при создании семафора. Семафор считается взведенным (не требует ожидания), когда его значение отлично от нуля. Если значение семафора равно нулю, то обращение к функции ожидания с указанием его хэндла приводит к переводу потока в состояние ожидания. При любом обращении к функции ожидания с указанием хэндла взведенного семафора его (семафора) значение автоматически уменьшается.

Пример создания семафора:

```
hSema:=CreateSemaphore(
          nil, // атрибуты защиты
          1, // начальное значение
          1, // максимальное значение
          'MySemaphore' // имя семафора
          );
```

Уменьшение значений семафора с возможным переходом в состояние ожидания осуществляется функциями ожидания, например

WaitForSingleObject(hSema,INFINITE);

Увеличение значения семафора производится функцией

ReleaseSemaphore(hSema,1,nil);

- увеличение на единицу, в последнем параметре можно передать указатель на 32-разрядную переменную, в которую будет помещено предыдущее значение семафора.

Получить хэндл уже созданного семафора по имени (например, в зависимом процессе) можно при помощи функции

hSema := OpenSemaphore(SEMAPHORE_ALL_ACCESS, true, 'MySemaphore');

Также можно вызвать CreateSemaphore, которая в случае существования семафора с тем же именем вернет хэндл уже существующего объекта, при этом начальное и максимальное значение будут проигнорированы.

Удаление семафора:

CloseHandle(hSema);

Мутексы

Мутексы (от mutual exclusion – взаимное исключение) представляют собой объекты, владеть которыми может только один поток. При задании хэндла мутекса в функции ожидания, поток либо переходит в состояние ожидания, если мутекс захвачен другим потоком, либо захватывает мутекс и продолжает свое выполнение. Действия с мутексами во многом аналогичны действиям с семафорами.

См. функции:

CreateMutex, OpenMutex, ReleaseMutex, CloseHandle.

События

События (events) представляют собой объекты, которые могут взводиться и сбрасываться произвольным образом по усмотрению программиста. При этом имеется возможность создавать события, которые автоматически сбрасываются при завершении функции ожидания, а также события, которые необходимо сбрасывать в явном виде. Действия с событиями аналогичны действиям с семафорами и мутексами, см. функции

CreateEvent, OpenEvent, SetEvent, ResetEvent, CloseHandle.

Задание

Выполняется при самостоятельной подготовке:

1. Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

2. По материалам Справки (Help) изучить описание функций и структур данных, на которые имеются ссылки в описании работы.

Выполняется в лаборатории:

1. Включить компьютер. Запустить систему Delphi 2.0.

2. Загрузить исходный текст примера

3. Откомпилировать и запустить пример. Изучить поведение созданных окон.

4. Написать и отладить программу в соответствии с индивидуальным заданием. Продемонстрировать результаты работы преподавателю.

5. Завершить работу с Delphi. Оставить компьютер включенным.

Варианты заданий:

1. "Телетайп". Два различных приложения. В одном имеется окно редактирования (edit), в котором пользователь набирает текст. Набираемые символы должны параллельно отображаться в другом окне. Для передачи символов и установления связи использовать межпроцессные сообщения (любые). При отсутствии запущенного приложения-сервера выдавать сообщение об ошибке.

2. Многопоточная программа, аналогичная лаб.2: главный поток обслуживает интерфейс пользователя, два дочерних потока рисуют в клиентской области окна случайные линии и окружности. Для запуска и остановки рисования предусмотреть кнопки.

3. "Массовая рассылка". Приложение построено следующим образом: имеется окно класса edit для ввода текста и кнопка "отправить". Может быть запущено произвольное количество экземпляров приложения. При нажатии кнопки "отправить" набранный текст должен быть скопирован в окна ввода всех запущенных экземпляров этого приложения. Размер текста ограничен 2000 символов. Использовать отображение файлов на память и широковещательную рассылку сообщения "обновить".

4. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения семафоров добиться корректной работы с массивами. Все потоки должны нормально завершаться при закрытии окна.

5. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и

обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения мутексов добиться корректной работы с массивами. Все потоки должны нормально завершаться при закрытии окна.

6. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения механизма событий. Все потоки должны нормально завершаться при закрытии окна.

7. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения механизма критических секций. Все потоки должны нормально завершаться при закрытии окна.

8. "Надзиратель". Взаимодействие программы-сервера и программы-клиента строится следующим образом: при запуске программы-клиента она посылает широковещательное сообщение с целью установить связь с сервером. Если сервер есть, он в ответ сообщает клиенту его номер, который клиент должен отобразить в рабочей области своего окна. Сервер отображает текущее количество клиентов. При завершении клиент посылает серверу уведомление. Считается, что сервер не может быть завершен раньше клиента (такую ситуацию обрабатывать не требуется).

9. "Надзиратель". Взаимодействие программы-сервера и программы-клиента строится следующим образом: при запуске программы-клиента она посылает широковещательное сообщение с целью установить связь с сервером. Если сервер есть, он в ответ сообщает клиенту его имя (строку длиной до 255 символов), которое клиент должен отобразить в заголовке своего окна. Сервер отображает текущее количество клиентов. При завершении клиент посылает серверу уведомление. Считается, что сервер не может быть завершен раньше клиента (такую ситуацию обрабатывать не требуется). Использовать для передачи строк отображение файла на память.

10. "Надзиратель". Взаимодействие программы-сервера и программы-клиента строится следующим образом: при запуске программы-клиента она посылает широковещательное сообщение с целью установить связь с сервером. Если сервер есть, он в ответ сообщает клиенту его имя (строку длиной до 255 символов), которое клиент должен отобразить в заголовке своего окна. Сервер отображает текущее количество клиентов. При завершении клиент посылает серверу уведомление. Считается, что сервер не может быть завершен раньше клиента (такую ситуацию обрабатывать не требуется). Использовать для передачи строк сообщение WM_COPYDATA.