Лекция 5  
Технологии программирования.

Библиотеки динамической компоновки

* Динамическую библиотеку можно загружать в адресное пространство при необходимости, что позволяет использовать подгружаемый код.
* Не накрадывается ограничений на язык описания исходного кода.
* Более простое управление проектами, если ведется командная разработка проекта. При использовании динамических библиотек достаточно обговорить лишь интерфейсы
* Экономия памяти, так как для нескольких процессов может быть загружена единственная копия библиотеки
* Возможность разделения ресурсов между всеми приложениями
* Упрощение локализации
* Расширение специфических возможностей использования COM объектов, и тд

Для выполнения любого приложения необходимо чтобы была создана исполняемая программа, которую называют загрузочным модулем или исполняемым файлом. Исполняемый файл создается на основе исходных текстов программы, которая подвергается трансляции. Процесс трансляции состоит из следующих этапов:

* Препроцессирование
* Лексический анализ, который предполагает выделение слов, входящие в состав языка
* Синтаксический анализ
* Семантический анализ, который предполагает оценивание правильности действия программы
* Генерация и оптимизация кода

Создается объектный модуль. Из набора объектных модулей, а так же модулей сторонних разработчиков компоновщик создает исполняемый файл программы, содержащий все необходимые связи для загрузки запуска и работы приложения

На этапе компоновки выделяют 2 варианта объединения исходных модулей в исполняемый файл:

* Статическая
* Динамическая

При статической компоновке функции созданные программистом и оформленные в виде отдельной библиотеки в виде копий встраиваются в исполняемый модуль создаваемый программой, при этом каждый исполняемый модуль, содержащий обращение к функции имеет свою собственную копию этой функции внутри программы.

Для многозадачных, многопользовательских ОС типичной является ситуации, когда несколько программ обращаются к одним и тем же функциям. Для экономии памяти и более эффективного использования процессорного времени в таких системах используют библиотеки динамической компоновки, тогда библиотека загружается в память однократно, а каждая программа на этапе компоновки в точке вызова функции получает ссылку на вызываемую функцию внешней библиотеки. Библиотечный файл загружается в память только в том случае, если какая либо из загруженных в систему программ выполняет обращение к ее функции. При выполнения обращения для того чтобы приложение могло вызвать функцию из библиотеки образ библиотечного файла должен быть спроецирован на адресное пространство вызывающего его процесса. Это может быть выполнено при помощи неявного связывания, либо с помощью явного связывания во время выполнения приложения. Как только библиотека спроецирована на адресное пространство вызвавшего его процесса, функции библиотеки становятся доступными всем потокам данного процесса. Для потоков код и данные библиотеки являются дополнительными кодом и данными. В случае если поток вызывает функцию из библиотеки, то происходит считывание параметров в стек потока и работа с ними как с локальными переменными. Таким образом говорят, что функция работает в контексте вызвавшего ее процесса.

Каждая библиотека использует специальный счетчик, содержимое которого увеличивается при каждом отображении библиотеки пространства процесса и уменьшается при выполнении приложением операции освобождения библиотеки. Реальная выгрузка библиотеки из глобального пула памяти осуществляется лишь в том случае, если счетчик использования будет иметь 0 значение.

Формат dll библиотеки идентичен формату загружаемого модуля, однако библиотека не может быть запущена. Это связано с отличием внутренней структуры процесса и модуля. Каждое приложение запускаемое в системе использует загруженный модуль приложений. Модуль включает в свой состав код приложения и используемые ресурсы. Если в системе запускается несколько копий одного приложения то они имеют общую ссылку на модуль приложения. При создании процесса, каждая копия приложений создает собственную область локальных данных, собственный стек, очередь сообщений, и виртуальный процессор состоящий из набора регистров процессора. По сути копия приложения является контекстом в котором выполняется модуль приложения. Идентификатор копии HInstance используется при выполнении программного модуля для связи с требуемым контекстом.

ОС в регистрационной БД хранит информацию о ссылках на экспортируемые функции модулей, расположение кода в модуле и ресурсе. Библиотека является модулем находится в памяти так же в единственном экземпляре, но в дополнение имеет один сегмент локальных данных, необходимых для хранения констант и внутренних переменных библиотеки. Библиотека не создает контекста при запуске, данных, стека и очереди сообщений. Библиотека использует все это у приложения, в контексте которого она запускается.

Существуют явная и неявная компоновка.

**Неявная компоновка** – динамическое связывание выполняемое во время загрузки программы, при этом код приложения ссылается на идентификаторы содержащиеся в dll. Тем самым заставляет загрузчик неявно загружать нужную библиотеку при запуске приложения.

Будем считать, что исполняемый exe модуль импортирует функции и переменные с dll, а библиотека в свою очередь экспортирует в исполняемом модуле. Собирая исполняемый модуль система осуществляет следующий порядок действий:

1. Формируется заголовочный файл с прототипами функций, структурами и идентификаторами экспортируемыми из dll. Этот файл включается в исходный код всех необходимых модулей dll. Он же требуется при сборке исполняемого модуля, использующего функцию dll.
2. Модули исходного кода с телами функции и определениями переменных, которые должны находится в dll оформляются в виде отдельных модулей. Данные модули в явном виде не требуются при выполнении сборки, они могут является коммерческой тайной разработчика.
3. Компилятор преобразует исходные модули библиотеки в объектные модули.
4. Компоновщик собирает все объектные модули в единый загрузочный dll модуль, в который помещается двоичный код и переменные относящиеся к dll. Полученный файл используется при сборке исполняемого модуля приложения
5. Если компоновщик обнаруживает, что библиотека экспортирует хотя бы одну функцию, то компоновщик дополнительно создает lib файл. Lib файл содержит список символьных имен экспортируемых библиотекой.
6. Во все модули исходного кода где есть ссылка на внешние функции и переменные включается заголовочный файл, предоставляемый разработчиком библиотеки.
7. В исходных модулях приложения организуется вызов внешних функций.
8. Компилятор формирует объектные модули из объявленных исходных модулей.
9. Компоновщик собирает объектные модули, загрузочный модуль, который так же помещает двоичный код и переменные относящиеся к запускаемому приложению. В создаваемом модуле определяется раздел импорта, где перечисляются все имена вызываемых внешней функции переменной необходимые dll. Для каждой dll в разделе импорта указывается на какие символьные имена ссылается двоичный код исполняемого файла. Эти сведения используются загрузчиком при работе приложения
10. Загрузчик создает виртуальное адресное пространство для нового процесса и проецируется для него исполняемый код.
11. Загрузчик анализирует раздел импорта, находит все необходимые dll модули и так же проецирует в адресное пространство процессов. Если проецируемые dll имеет собственный раздел импорта, то происходит каскадная загрузка.

Таким образом все необходимые связи производятся в момент загрузки приложения. Это хорошо тем, что все что надо уже загружено в память, следовательно максимально быстро происходит обращение. Долгое время запуска не всегда обосновано.

Заголовочный файл всегда должен поставляться разработчиком вместе с поставляемой библиотеке. Все функции описываются в заголовочном файле при помощи export. Extern C не позволяет компилятору искажать названия переменных и функции, которые будут доступны в исполняемом модуле, причем имя не будет изменено в любом случае не зависимо от языка программирования. Модификатор declspec обозначает, что описанная переменная, функция или класс экспортируется из динамической библиотеки, при этом компилятор С встраивает в конечный объектный файл дополнительную информацию. Доп информация используется компоновщиком при сборке dll из объектного файла. Обнаружив доп. Информацию компоновщик создает lib файл со списком экспортируемых идентификаторов. Конечный dll файл компоновщик встраивает в раздел экспорта. Раздел экспорта представляет собой таблицу, где все экспортируемые имена располагаются в алфавитном порядке их идентификаторов.

Таблица содержит:

* порядковый номер экспортируемого имени
* не символьный идентификатор hint, который определяет порядок экспортируемого имени внутри объектного модуля. Данный идентификатор может использоваться при организации обращения к экспортируемому имени из внешней программы. Присваивается в момент компиляции.
* Относительный виртуальный адрес RVA. Смещение в dll в котором можно найти экспортируемый идентификатор.
* Name. Символьное имя переменной\функции\класса.

Столбец ordinal можно использовать только для совместимости с 16 разрядными приложениями

При создании исходного ЕХЕ модуля необходимо определить внешнюю функцию.

#Include “myDll.h”;

Extern “C” \_\_declspec(dllimport) MyFunc(char\* str)

Компоновщик при создании конечного ЕХЕ встраивает в него раздел импорта. Раздел импорта содержит все импортируемые идентификаторы с указанием библиотек, которые их предоставляет. Раздел импорта создается на основе lib файлов, которые поставляются разработчиком. Компоновщик должен удостовериться в существовании идентификатора и определить в какой библиотеке находится. Компоновка завершается успешно, если разрешены все внешние ссылки. Внутри исходного модуля программы, при неявном связывании происходит прямой вызов функции по ее имени, которое сопоставляется с именем в таблице импорта.

*Int iCode = myFunc(“Hello”)*

При запуске ЕХЕ модуля загрузчик создает ВИРТУАЛЬНОЕ АДРЕСНОЕ ПРОСТРАНСТВО… и проецирует на ВИРТУАЛЬНОЕ АДРЕСНОЕ ПРОСТРАНСТВО процесса все необходимые dll. Затем загрузчик настраивает ссылки на импортируемые идентификаторы, т. Е. связывает точку вызова с РВА. С этого времени ссылка доступна в коде, динамическая связь установлена, приложение может работать, иначе возбуждается исключительная ситуация.

**Явное связывание.** Производится в момент вызова приложения, при этом предполагается, что поток явно загружает dll в адресное пространство процесса, получает виртуальный адрес необходимого dll идентификатора и вызывает функцию переменную или класс по этому адресу.

Процесс явного связывания:

1. Разработчик создает заголовочный файл, в котором определяются экспортируемые структуры\переменные\классы и тд
2. Разработчик создает модули на выбранном ЯП, в котором все это реализуется
3. Исходные модули подвергаются компиляции, при этом для каждого исходного модуля создается объектный модуль
4. На основании имеющихся объектных модулей, компоновщик собирает dll библиотеку. Если dll имеет хотя бы 1 экспортируемое имя, то создается lib файл.
5. Заголовочный файл с импортируемыми функциями\структурами и классами включаются в исходный модуль приложения
6. Разрабатывается набор модулей на ЯП, в которых используются внешние функции, структуры и тд.
7. Компилятор создает объектный файл для каждого исходного модуля.
8. Компоновщик собирает ЕХЕ файл из объектных модулей, при этом не lib\dll не используются, так как никаких прямых ссылок на экспортируемые идентификаторы не существует, то раздел импорта не создается.
9. Загрузчик создает ВАП для исполняемого модуля.
10. Как только поток вызывает внешнюю функцию, то соответствующая dll загружается в адресное пространство процесса и все потоки данного процесса могут использовать функции библиотеки ссылаясь на экспортируемы идентификаторы. Если библиотеки по указанному адресу не существует, либо отсутствует символьное имя, то выбрасывается исключение.

LoadLibrary(string path); Загрузка либы.

LoadLibrary грузит библиотеку и возвращает ссылку на нее в качестве результата, если неудача, то возвращается 0. После загрузки библиотеки к каждому экспортируемому ресурсу можно обратиться путем получения адреса точки входа. Адрес точки входа формируется с помощью вызова функции getProcAddress(hDll, “MyFunc”). Если указанное символьное имя отсутствует, то возвращается NULL.

FreeLibrary(hDll) // Выгрузка библиотеки из памяти.

Основная функцию DLL

Boll APIENTRY DllMain()

HINSTANCE – индетефикатор dll

DWORD – код причины вызова функции:

* Запуск процесса.
* Создана новая задача.
* Задача завершает свою работу не аварийным способом
* Нормальное завершение процесса

LPVOID – зарезервированный параметр.

Идентификатор библиотеки используется для обращения к библиотечным ресурсам.

В некоторых случая создается специализированные файлы, позволяющие согласовать импортируемые и экспортируемые элементы, их называют файлами определения или DEF файлы. Они содержат описание экспортируемых функции в следующем формате. Для всех экспортируемых функций определяется идентификатор EXPORTS, далее указывается имя точки входа. Под ним понимается имя с которым экспортируемая функция или переменная будет доступна для вызова. Далее необязательные аргументы:

* Внутреннее имя.
* @Номер. Номер экспортируемой функции. Может быть задан для более быстрого обращения к внешнему имени. Определяет значение hinst в таблице.
* NONAME. Определяет, что порядковый номер может быть невидимым для внешнего использования.
* CONSTANT. Позволяет экспортировать из dll не только функции но и данные, тогда имя точки входа является именем глобальной переменной.