Управление памятью, отображение файлов и библиотеки DLL

Управление динамической памятью в той или иной форме требуется в большинстве программ. Необходимость в этом возникает всякий раз, когда требуется создавать структуры данных, размер которых не может быть определен заранее на стадии создания программы. Типичными примерами динамических структур данных могут служить деревья поиска, таблицы имен и связанные списки.

В Windows предусмотрены гибкие механизмы управления динамической памятью программы. Кроме того, Windows предоставляет средства отображения файлов, которые позволяют ассоциировать файл непосредственно с виртуальным адресным пространством процесса, благодаря чему ОС может управлять любыми перемещениями данных между файлом и памятью, так что программисту вообще не приходится иметь дело с функциями ReadFile, WriteFile, SetFilePointer и другими функциями ввода/вывода.

В случае использования отображения файлов программе удобно сохранять внутренние динамические структуры данных в виде постоянно существующих файлов, а все алгоритмы обработки применять к создаваемой в памяти копии файла. Более того, отображение файлов может значительно ускорить последовательную обработку файлов и предоставляет механизм, обеспечивающий совместное использование областей памяти одновременно несколькими процессами.

Важным специальным случаем отображения файлов и разделения памяти являются динамически компонуемые библиотеки (dynamic linked libraries, DLL), обеспечивающие возможность отображения файлов (обычно, когда они используются только для чтения) на адресное пространство процесса для их выполнения.

В этой главе описывается система управления памятью и функции отображения файлов Windows, что иллюстрируется целым рядом примеров их использования, а также обсуждаются явно и неявно связанные библиотеки DLL.

Архитектура системы управления памятью в Win32 и Win64

Win32 (в данном случае различия между Win32 и Win64 становятся существенными) — это API 32-разрядных ОС семейства

Windows. "32-разрядность" проявляет себя при адресации памяти тем, что указатели (LPSTR, LPDWORD и так далее) являются 4-байтовыми (32-битовыми) объектами. Win64 API предоставляет виртуальное адресное пространство гораздо большего объема, и 64-битовые указатели являются естественным результатом эволюции Win32. Тем не менее, о переносимости приложений на платформу Win64 необходимо заботиться отдельно. Настоящее обсуждение будет относиться только к Win32; вопросы миграции приложений на платформу Win64 обсуждаются в главе 16, где также приводятся ссылки на соответствующие источники информации.

Далее, в рамках Win32 у каждого процесса имеется собственное виртуальное адресное пространство объемом 4 Гбайт (*232* байт). Разумеется, объем виртуального адресного пространства в Win64 гораздо больше. По крайней мере, половину указанного пространства (2-3 Гбайт; расширение до 3 Гбайт должно производиться во время загрузки) Win32 делает доступной для процесса. Оставшаяся часть виртуального адресного пространства выделяется для совместно используемых данных и кода, системного кода, драйверов и так далее.

Хотя детали описанного распределения памяти и заслуживают интерес, здесь они обсуждаться не будут; прикладные программы используют абстрактные модели памяти, предоставляемые API. С точки зрения программиста ОС просто предоставляет адресное пространство большого объема для размещения кода, данных и других ресурсов. В этой главе мы сосредоточим свое внимание на использовании средств управления памятью в Windows, не заботясь о том, как все это реализуется в ОС. Тем не менее, ниже приводится соответствующий краткий обзор.

Обзор методов управления памятью

Обо всех деталях отображения виртуальных адресов на физические адреса (virtual to physical memory mapping), механизмах страничной подкачки (page swapping) и замещения страниц по запросу (demand paging) и прочих моментах заботится ОС. Эти вопросы подробно обсуждаются в документации по ОС, а также в книге Соломона (Solomon) и Руссиновича (Russinovich) *Inside Windows2000*. Краткое изложение наиболее существенных сведений приводится ниже:

• Система может располагать сравнительно небольшим объемом физической памяти; на практике для всех систем, кроме Windows XP, необходимый минимум составляет 128 Мбайт, однако в типичных случаях доступные объемы физической памяти оказываются намного большими.[21]

• Каждый отдельный процесс — а таких процессов, как пользовательских, так и системных, может выполняться одновременно несколько — имеет собственное виртуальное адресное пространство, объем которого может значительно превосходить объем доступного физического адресного пространства. Например, емкость виртуального адресного пространства объемом 1 Гбайт, относящегося к одному процессу, в восемь раз превышает емкость физической памяти объемом 128 Мбайт, и таких процессов может быть множество.

• ОС преобразует виртуальные адреса в физические адреса.

• Для большинства виртуальных страниц в физической памяти места не хватит, поэтому ОС имеет возможность реагировать на страничные ошибки (page faults), возникающие при попытках обращения к страницам, которые отсутствуют в памяти, и загружать данные с жесткого диска — из системного файла подкачки (swap file) или из обычного файла. Будучи прозрачными для программиста, страничные ошибки отрицательно влияют на производительность, поэтому программы должны проектироваться таким образом, чтобы вероятность возникновения подобных ошибок была сведена к минимуму. Более подробное освещение этой темы, рассмотрение которой выходит за рамки данной книги, вы найдете в справочной документации по ОС.

На рис. 5.1 проиллюстрировано расположение уровней API управления памятью Windows поверх диспетчера виртуальной памяти (Virtual Memory Manager, VMM). API виртуальной памяти Windows (VirtualAlloc, VirtualFree, Virtual-Lock, VirtualUnlock и так далее) работает с целыми страницами. API кучи Windows управляет блоками памяти, размер которых определяется пользователем.

Мы не будем останавливаться на топологии адресного пространства виртуальной памяти, поскольку она не имеет непосредственного отношения к API, различна в Windows 9x и Windows NT и в будущем может измениться. Соответствующая информация содержится в документации Microsoft.

Тем не менее, многим программистам хотелось бы знать больше о своей среде разработки. Начните исследование структуры памяти в вашей системе с вызова следующей функции:

VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo)

Параметром этой функции служит адрес структуры PSYSTEM_INFO, в которой содержится информация относительно размера системной страницы, а также адресах физической памяти, доступных для приложений.

Куча

Windows поддерживает пулы памяти, называемые *кучами* (heaps). Процесс может иметь несколько куч, которые используются для распределения памяти.

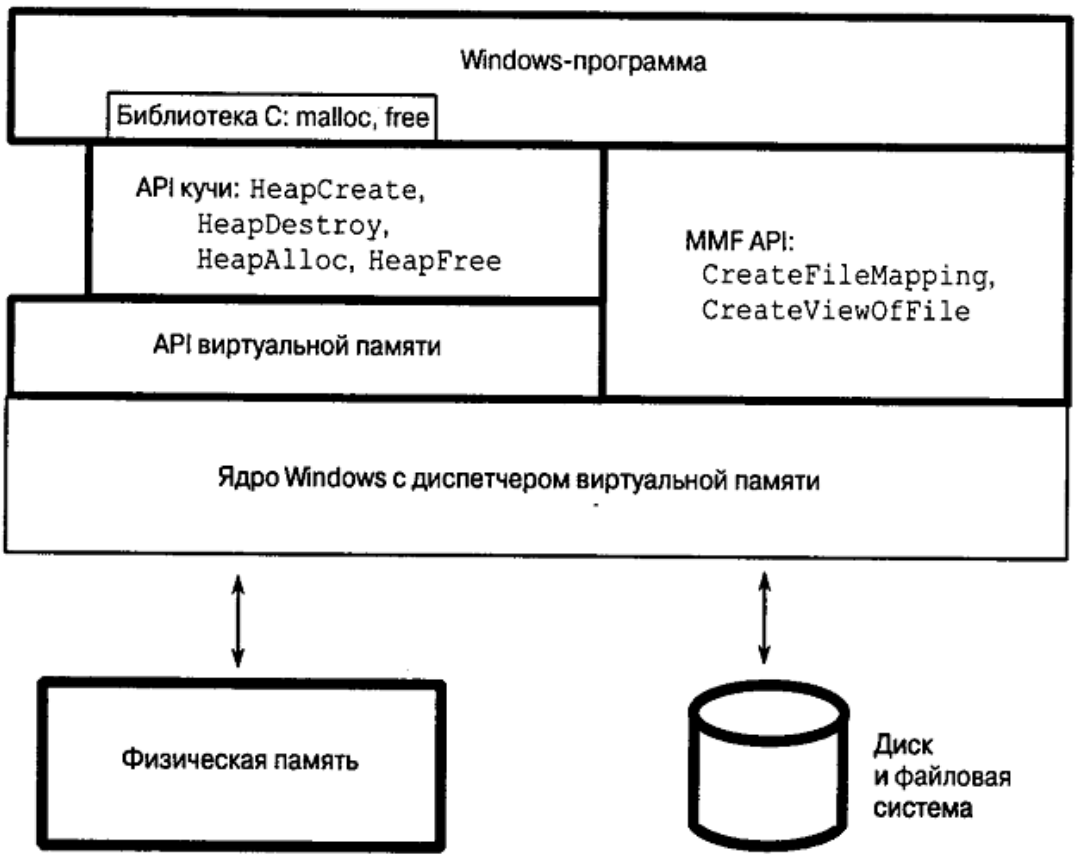

Рис. 5.1. Архитектура системы управления памятью Windows

Во многих случаях одной кучи вполне достаточно, но в силу ряда причин, о которых будет сказано ниже, иногда целесообразно иметь в своем распоряжении несколько куч. Если одной кучи вам хватает, можно обойтись использованием функций управления

памятью, предоставляемых библиотекой C (malloc, free, calloc, realloc).

Кучи являются объектами Windows и, следовательно, имеют дескрипторы. Дескриптор кучи используется при распределении памяти. У каждого процесса имеется куча, заданная по умолчанию, которую использует функция malloc и для получения дескриптора которой используется следующая функция:

HANDLE GetProcessHeap(VOID)

Возвращаемое значение: дескриптор кучи процесса; в случае неуспешного завершения — NULL.

Заметьте, что для индикации неудачного завершения функции используется возвращаемое значение NULL, а не INVALID_HANDLE_VALUE, как в случае функции CreateFile.

Программа также может создать несколько различных куч. Иногда для размещения в памяти отдельных структур данных оказывается удобным, чтобы для каждой из них была предусмотрена своя куча. Использование независимых куч обеспечивает ряд преимуществ.

• **Отсутствие взаимной дискриминации между потоками.** Ни один из потоков не сможет получить больше памяти, чем распределено для ее кучи. В частности, так называемая утечка памяти (memory leak), возникающая в тех случаях, когда программа "забывает" своевременно освободить память, занятую элементами данных, необходимости в которых больше нет, будет влиять лишь на один поток процесса.[22]

• **Повышение производительности.** Предоставление собственной кучи каждого потока уменьшает состязательность между ними, в результате чего общая производительность программы может значительно повыситься. См. главу 9.

• **Эффективность размещения данных.** Размещение элементов данных фиксированного размера в небольшой куче может оказаться гораздо более эффективным, чем размещение множества элементов самых различных размеров в одной большой куче. При этом также уменьшается фрагментация памяти. Кроме того, предоставление каждого потока собственной кучи существенно упрощает синхронизацию потоков, что приводит к дополнительному повышению производительности.

• **Эффективность освобождения памяти.** Области памяти, распределенные для кучи в целом и всех структур данных, которые она содержит, могут быть освобождены с помощью единственного вызова функции. Этот вызов также устранит отрицательные последствия утечки памяти, связанной с данной кучей.

• **Эффективность локализации обращений к памяти.** Сохранение структуры данных в небольшой куче гарантирует, что для всех элементов данных потребуется сравнительно небольшое количество страниц, а это может уменьшить вероятность возникновения ошибок страниц в процессе обработки элементов структур данных.

Ценность указанных преимуществ может варьироваться в зависимости от приложения, и многие программисты ограничиваются использованием только кучи процесса, для управления которой используют функции библиотеки С. Однако такой выбор лишает программу возможности воспользоваться способностью функций управления памятью Windows генерировать исключения (обсуждается при рассмотрении функций). В любом случае для создания и уничтожения куч применяются две функции, описания которых приводятся ниже.[23]

Начальный размер кучи, устанавливаемый параметром dwInitialSize (который может быть нулевым), всегда округляется до величины, кратной размеру страницы, и определяет объем физической памяти (в *файле подкачки),* который *передается* (commit) в распоряжение кучи (для последующего распределения памяти по запросам) первоначально, а не в ответ на запросы распределения (allocation) памяти из кучи. Когда программа исчерпывает первоначальный размер кучи, куче автоматически передаются дополнительные страницы памяти вплоть до тех пор, пока она не достигнет установленного для нее максимального размера. Поскольку файл подкачки является ограниченным ресурсом, рекомендуется откладывать передачу памяти куче на более поздний срок, если только заранее неизвестно, какой размер кучи потребуется. Максимально допустимый размер кучи при ее увеличении в результате динамического расширения определяется значением параметра dwMaximumSize (если оно ненулевое). Рост куч процессов, заданных по умолчанию, также осуществляется динамическим путем.

HANDLE HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize)

Возвращаемое значение: дескриптор кучи; в случае неудачного завершения — NULL.

Типом данных обоих упомянутых полей, связанных с размерами кучи, является не DWORD, а SIZE_T. Тип данных SIZE_T определяется как 32– или 64-битовое целое число без знака, в зависимости от флагов компилятора (_WIN32 или _WIN64). Этот тип данных был введен специально для того, чтобы обеспечить возможность миграции приложений Win64 (см. главу 16), и охватывает весь диапазон 32– и 64-битовых указателей. Вариантом этого типа данных для чисел со знаком является тип SSIZE_T).

flOptions — этот параметр может объединять следующие два флага:

• HEAP_GENERATE_EXCEPTIONS: в случае ошибки при распределении памяти вместо возврата значения NULL генерируется исключение, которое должно быть обработано средствами SEH (см. главу 4). Если установлен этот флаг, то такие исключения при сбоях будет возбуждаться не самой функцией HeapCreate, а такими функциями, как HeapAlloc, к рассмотрению которых мы вскоре перейдем.

• HEAP_NO SERIALIZE: при определенных обстоятельствах, о которых сказано ниже, установка этого флага может привести к незначительному повышению производительности.

Существуют другие важные моменты, связанные с параметром dwMaximumSize.

• Если параметр dwMaximumSize имеет ненулевое значение, то виртуальное адресное пространство резервируется в соответствии с этим значением, даже если первоначально не все оно передается в распоряжение кучи. Это значение определяет максимальный размер кучи, о котором в этом случае говорят как о *нерастущем* (nongrowable). Данный параметр ограничивает размер кучи, чтобы, например, обеспечить отсутствие дискриминации между потоками, о чем говорилось выше.

• Если же значение dwMaximumSize равно 0, то куча может *расти* (grow), превышая предел, установленный начальным размером, и в этом случае максимальный размер кучи ограничивается лишь объемом доступного виртуального адресного пространства, не

распределенного в данный момент для других куч и файла подкачки.

Заметьте, что кучи не имеют атрибутов защиты, поскольку доступ к ним извне процесса невозможен. В то же время, для объектов отображения файлов, описанных далее в этой главе, защита предусмотрена (глава 15), так как они могут совместно использоваться несколькими процессами.

Для уничтожения объекта кучи используется функция HeapDestroy. Она также может служить примером исключения из общих правил, в данном случае — правила, согласно которому для удаления ненужных дескрипторов любого типа используется функция CloseHandle.

BOOL HeapDestroy(HANDLE hHeap)

Параметр **hHeap** должен указывать на кучу, созданную посредством вызова функции HeapCreate. Будьте внимательны и следите за тем, чтобы случайно не уничтожить кучу процесса, заданную по умолчанию (дескриптор которой получают с помощью функции GetProcessHeap). В результате уничтожения кучи освобождается область виртуального адресного пространства и физическая область сохранения файла подкачки. Разумеется, правильно спроектированная программа должна уничтожать кучи, необходимости в которых больше нет.

Помимо всего прочего, уничтожение кучи позволяет быстро освободить память, занимаемую структурами данных, избавляя вас от необходимости отдельного уничтожения каждой из структур, однако экземпляры объектов C++ уничтожены не будут, поскольку их деструкторы при этом не вызываются. Применение операции уничтожения кучи имеет следующие положительные стороны:

1. Отпадает необходимость в написании программного кода, обеспечивающего обход структур данных.
2. Отпадает необходимость в освобождении памяти, занимаемой каждым из элементов, по отдельности.
3. Система не затрачивает время на обслуживание кучи, поскольку отмена распределения памяти для всех элементов структуры данных осуществляется посредством единственного вызова функции.

Функции библиотеки C используют только одну кучу. В силу этого иметь дело с чем-либо, напоминающим дескрипторы куч Windows, в данном случае не приходится.

В UNIX адресное пространство процесса может быть увеличено с помощью функции sbrk, однако эта функция не является диспетчером памяти общего назначения.

При неудачных попытках распределения памяти в UNIX сигналы не генерируются, поэтому в программах должна быть предусмотрена явная проверка значений возвращаемых указателей.

Управление памятью кучи

Для получения блока памяти из кучи следует указать дескриптор области памяти кучи, размер блока и некоторые флаги.

LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes)

Возвращаемое значение: в случае успешного выполнения — указатель на распределенный блок памяти, иначе — NULL (если только не была указана генерация исключения).

Параметры

hHeap — дескриптор кучи, из которой должен быть распределен блок памяти. Этот дескриптор должен быть предоставлен либо функцией GetProcessHeap, либо функцией HeapCreate.

dwFlags — может объединять следующие флаги:

• HEAP_GENERATE_EXCEPTIONS и HEAP_NO_SERIALIZE: эти флаги имеют тот же смысл, что и в случае функции HeapCreate. Первый флаг игнорируется, если он был установлен функцией кучи HeapCreate, но активизирует исключения для каждого отдельного вызова функции HeapAlloc, даже если функцией HeapCreate флаг HEAP_GENERATE_EXCEPTIONS и не был задан. При распределении памяти из кучи процесса второй флаг использовать не следует.

• HEAP_ZERO_MEMORY: этот флаг указывает, что распределенная память будет инициализирована значениями 0;

если этот флаг не установлен, содержимое памяти является неопределенным.

dwBytes — размер блока памяти, который должен быть распределен. Для нерастущих куч значение этого параметра не должно превышать 0x7FFF8 (приблизительно 0,5 Мбайт).

Примечание

Как только функция HeapAlloc вернула указатель, вы можете использовать его самым обычным способом; ссылаться после этого на его кучу нет никакой необходимости. Заметьте, что тип данных LPVOID может представлять либо 32-битовый, либо 64-битовый указатель.

Для освобождения блока памяти, распределенного из кучи достаточно вызвать следующую функцию:

BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem)

dwFlags — значениями этого параметра должны быть 0 или HEAP_NO_SERIALIZE. Значением параметра **lpMem** должно быть значение, возвращенное функциями HeapAlloc или HeapReAlloc (описана ниже), а дескриптор **hHeap** должен быть дескриптором кучи, которой принадлежит освобождаемый блок памяти, указываемый lpMem.

Для повторного распределения блоков памяти с целью изменения их размера используется следующая функция:

LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, SIZE_T dwBytes)

Возвращаемое значение: в случае успешного выполнения — указатель на перераспределенный блок памяти; в противном случае функция возвращает NULL или вызывает исключение.

Параметры

dwFlags -

• HEAP_GENERATE_EXCEPTIONS и HEAP_NO_SERIALIZE: это те же флаги, которые были описаны при рассмотрении функции HeapAlloc.

• HEAP_ZERO_MEMORY: нулями инициализируется лишь вновь распределенная память (когда значение параметра dwBytes превышает первоначальный размер блока). Содержимое исходного блока не изменяется.

• HEAP_REALLOC_IN_PLACE_ONLY: установка этого флага запрещает перемещение блока при перераспределении памяти. Если вы увеличиваете размер блока, адреса добавляемой памяти будут располагаться непосредственно вслед за адресами памяти, занимаемой существующим блоком.

lpMem — указывает на блок памяти, перераспределяемый из кучи hHeap.

dwBytes — размер нового блока памяти, который может быть как меньше, так и больше размера существующего блока.

Обычно возвращенный указатель имеет то же значение, что и указатель lpMem. В то же время, если блок перемещается (чтобы такое перемещение было разрешено, следует при вызове функции опустить флаг HEAP_REALLOC_IN_PLACE_ONLY), то возвращенное значение будет другим. Следите за своевременным изменением любых ссылок на блок. Независимо от того, перемещается блок или не перемещается, содержащиеся в нем данные остаются неизменными; в то же время, при уменьшении блока часть данных может теряться.

Размер распределенного блока памяти можно определить, вызвав функцию HeapSize (эту функцию следовало бы назвать BlockSize, поскольку о размере кучи она ничего не сообщает), используя в качестве параметров дескриптор кучи и указатель на блок.

DWORD HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem)

Возвращаемое значение: в случае успешного выполнения — размер блока; иначе — ноль.

Флаг HEAP_NO_SERIALIZE

При вызове функций HeapCreate, HeapAlloc и HeapReAlloc можно указывать флаг HEAP_NO_SERIALIZE. Использование этого флага иногда обеспечивает незначительный выигрыш в производительности, поскольку во время обращения функции к куче взаимоисключающая блокировка к потокам в этом случае применяться не будет. Результаты простых тестов, в которых не делалось ничего, кроме распределения блоков памяти, показали повышение производительности примерно на 16 процентов. Этот флаг без какого бы то ни было риска можно использовать в следующих ситуациях:

• Программа не использует потоки, или, точнее, процесс имеет только один поток. В данной главе этот флаг используется во всех примерах.

• Каждый поток имеет собственную кучу или набор куч, и никакой другой поток не имеет доступа к этой куче.

• Программа располагает собственным механизмом взаимоисключающей блокировки, который предотвращает одновременный доступ к куче сразу нескольких потоков, использующих функции HeapAlloc и HeapReAlloc. Для этой цели также могут применяться функции HeapLock и HeapUnlock.

Флаг HEAP_GENERATE_EXCEPTIONS

Разрешение исключений вместо возврата значений NULL в случае сбоев при распределении памяти позволяет избавиться от утомительной необходимости тестирования результатов каждой попытки такого распределения. К тому же, обработчики исключений или завершения могут производить очистку памяти, которая к этому моменту была частично распределена. Эта методика применена в нескольких примерах.

Возможны два кода исключения:

1. STATUS_NO_MEMORY: это значение указывает на то, что системе не удалось создать блок запрошенного объема. Причинами этого могут быть фрагментация памяти, достижение нерастущей кучей максимально допустимого размера или исчерпание всей доступной памяти растущими кучами.

2. STATUS_ACCESS_VIOLATION: это значение указывает на повреждение кучи.

Одной из возможных причин этого может быть выполнение программой записи в память с выходом за границы распределенного блока.

Другие функции кучи

Функция **HeapCompact** пытается уплотнить, или *дефрагментировать,* смежные блоки в куче. Функция **HeapValidate** пытается обнаруживать повреждения кучи. Функция **HeapWalk** перечисляет блоки в куче, а функция **GetProcessHeaps** получает все действительные дескрипторы куч.

Функции HeapLock и HeapUnlock позволяют потокам сериализовать доступ к куче, о чем говорится в главе 8.

Имейте в виду, что эти функции не работают под управлением Windows 9x или Windows CE. Кроме того, имеются некоторые вышедшие из употребления функции, которые использовались ранее для совместимости с 16-битовыми системами. Мы упомянули об этих функциях лишь для того, чтобы лишний раз подчеркнуть тот факт, что многие функции продолжают поддерживаться, хотя никакой необходимости в них больше нет.

Резюме: управление кучами

Обычная процедура использования куч не представляет никаких сложностей:

1. Получите дескриптор кучи, воспользовавшись одной из функций CreateHeap или GetProcessHeap.
2. Распределите блоки из кучи, используя функцию HeapAlloc.
3. В случае необходимости освободите все или только некоторые блоки при помощи функции HeapFree.
4. Уничтожьте кучу и закройте ее дескриптор при помощи функции HeapDestroy.

Этот процесс иллюстрируют рис. 5.2 и программа 5.2.

В отсутствие необходимости создания отдельных куч или генерации исключений программисты, которые привыкли использовать функции управления памятью из библиотеки C, могут использовать их и далее. При этом, если речь идет о куче процесса, функция malloc эквивалентна функции HeapAlloc, функция realloc — функции HeapReAlloc, а функция free — функции HeapFree. Функция calloc распределяет память и инициализирует объекты, и ее поведение легко эмулируется функцией HeapAlloc. Эквивалент функции HeapSize в библиотеке C отсутствует.

Пример: сортировка файлов с использованием бинарного дерева поиска

Распространенными динамическими структурами данных, требующими управления памятью, являются деревья поиска. Деревья поиска предоставляют удобный способ сопровождения коллекций записей, дополнительным преимуществом которого является возможность применения чрезвычайно эффективных алгоритмов обхода узлов.

Программа sortBT (программа 5.1) реализует ограниченную версию UNIX-команды sort за счет создания бинарного дерева поиска с использованием двух куч. Ключи размещаются в *куче узлов* (node heap), представляющей дерево поиска. Каждый узел содержит левый и правый указатели, ключ и указатель на запись в *куче данных* (data heap). Заметьте, что куча узлов состоит из блоков фиксированного размера, тогда как куча данных содержит строки переменной длины. Наконец, отсортированный файл выводится путем обхода дерева.

В данном примере для использования в качестве ключа произвольно выбраны первые 8 байтов строки, а не целая строка. В двух других вариантах реализации сортировки, приведенных в настоящей главе (программы 5.4 и 5.5), выполняется сортировка индексированных файлов, а показатели производительности всех трех программ сравниваются в приложении B.

Последовательность операций по созданию куч и размещению блоков в памяти представлена на рис. 5.2. Программный код, приведенный справа, является *псевдокодом,* который отражает лишь наиболее существенные вызовы функций и аргументы. В виртуальном адресном пространстве, схематически изображенном слева, выделена память для трех куч, в каждой из которых имеются распределенные блоки. Программа 5.1 незначительно отличается от рисунка в том, что на рисунке, в отличие от программы, корень дерева размещен в куче процесса.

Примечание

Фактическое расположение куч и блоков в пределах куч зависит от варианта реализации Windows, а также от предыстории использования памяти процессом, включая рост кучи сверх ее начального размера. Кроме того, после увеличения размера растущей кучи с выходом за границы начальной области она может уже не занимать непрерывное адресное пространство. Наиболее оптимальная практика программирования состоит в том, чтобы не делать относительно фактической топологии распределения памяти никаких предположений; просто используйте функции управления памятью так, как это определяют правила работы с ними.

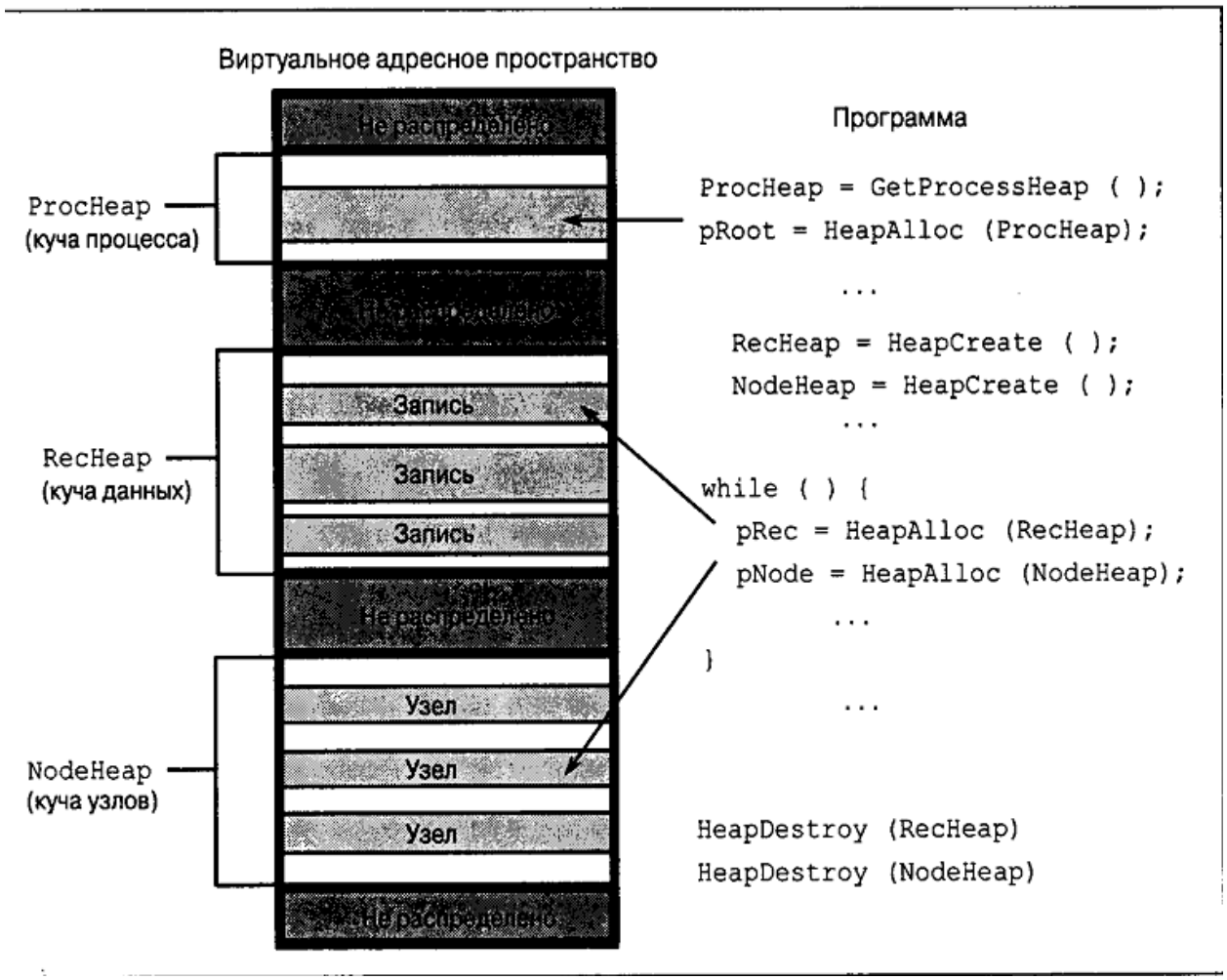

Рис. 5.2. Управление памятью при наличии нескольких куч

Программа 5.1 иллюстрирует некоторые методики, которые упрощают программу, но были бы невозможны при использовании одной только библиотеки C или же только кучи процесса.

• Элементы узлов имеют фиксированный размер и размещаются в собственной куче, тогда как элементы данных переменной длины размещаются в отдельной куче.

• Готовясь к сортировке очередного файла, программа уничтожает две кучи, а не освобождает память, занимаемую отдельными элементами.

• Ошибки при распределении памяти обрабатываются как исключения, вследствие чего отпадает необходимость в тестировании возвращаемых значений функциями для отслеживания нулевых указателей.

Если используется Windows, то сфера применимости таких программ, как программа 5.1, ограничивается файлами небольшого размера, поскольку в виртуальной памяти должны находиться

целиком весь файл и копии ключей. Абсолютный верхний предел размера файла определяется объемом доступного виртуального адресного пространства (максимум 3 Гбайт); фактически достижимый предел оказывается еще меньшим. В случае Win64 ограничения подобного рода практически отсутствуют.

В программе 5.1 вызываются некоторые функции управления деревом: FillTree, InsertTree, Scan и TreeCompare. Все они представлены в программе 5.2.

В этой программе используются исключения кучи. Можно было бы поступить иначе, отказавшись от использования флага HEAP_GENERATE_EXCEPTIONS и отслеживая ошибки, возникающие при распределении памяти, явным образом.

Программа 5.1. sortBT: сортировка с использованием бинарного дерева поиска

```
/* Глава 5. Команда sortBT. Версия, использующая бинарное дерево поиска.*/
#include "EvryThng.h"
#define KEY_SIZE 8

typedef struct _TreeNode {/* Описание структуры узла. */
 struct _TreeNode *Left, *Right;
 TCHAR Key[KEY_SIZE];
 LPTSTR pData;
} TREENODE, *LPTNODE, **LPPTNODE;
#define NODE_SIZE sizeof(TREENODE)
#define NODE_HEAP_ISIZE 0x8000
#define DATA_HEAP_ISIZE 0x8000
#define MAX_DATA_LEN 0x1000
#define TKEY_SIZE KEY_SIZE * sizeof(TCHAR)

LPTNODE FillTree(HANDLE, HANDLE, HANDLE);
BOOL Scan(LPTNODE);
int KeyCompare (LPCTSTR, LPCTSTR); iFile;

BOOL InsertTree (LPPTNODE, LPTNODE);

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hIn, hNode = NULL, hData = NULL;
```

```
   LPTNODE pRoot;
   CHAR ErrorMessage[256];
   int iFirstFile = Options(argc, argv, _T("n"), &NoPrint, NULL);
   /* Обработать все файлы, указанные в командной строке. */
   for (iFile = iFirstFile; iFile < argc; iFile++) __try {
    /* Открыть входной файл. */
   hIn = CreateFile(argv[iFile], GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL);
   if (hIn == INVALID_HANDLE_VALUE) RaiseException(0, 0, 0,
NULL);
    __try { /* Распределить две кучи. */
       hNode = HeapCreate(HEAP_GENERATE_EXCEPTIONS |
HEAP_NO_SERIALIZE, NODE_HEAP_ISIZE, 0);
    hData = HeapCreate(HEAP_GENERATE_EXCEPTIONS |
HEAP_NO_SERIALIZE, DATA_HEAP_ISIZE, 0);
     /* Обработать входной файл, создавая дерево. */
     pRoot = FillTree(hIn, hNode, hData);
     /* Отобразить дерево в порядке следования ключей. */
     _tprintf(_T("Сортируемый файл: %s\n"), argv [iFile]);
     Scan(pRoot);
    } _ finally { /* Кучи и дескрипторы файлов всегда закрываются.
     /* Уничтожить обе кучи и структуры данных. */
     if (hNode !=NULL) HeapDestroy (hNode);
     if (hNode != NULL) HeapDestroy (hData);
     hNode = NULL;
     hData = NULL;
     if (hIn != INVALID_HANDLE_VALUE) CloseHandle (hIn);
    }
   } /* Конец основного цикла обработки файлов и try-блока. */
   __except(EXCEPTION_EXECUTE_HANDLER) {
     _stprintf(ErrorMessage, _T("\n%s %s"), _T("sortBT, ошибка при
обработке файла:"), argv [iFile]);
    ReportError(ErrorMessage, 0, TRUE);
   }
   return 0;
  }
```

В программе 5.2 представлены функции, которые фактически реализуют алгоритмы поиска с использованием бинарного дерева. Первая из этих функций, FillTree, распределяет память в обеих

кучах. Вторая функция, KeyCompare, используется также в нескольких других программах в данной главе. Заметьте, что функции FillTree и KeyCompare используют обработчики завершения и исключений программы 5.1, которая вызывает эти функции. Таким образом, ошибки распределения памяти будут обрабатываться основной программой, которая после этого продолжит свое выполнение, переходя к обработке следующего файла.

Программа 5.2. FillTree и другие функции управления деревом поиска

```
LPTNODE FillTree(HANDLE hIn, HANDLE hNode, HANDLE hData)
/* Заполнение дерева записями из входного файла. Используется обработчик исключений вызывающей программы. */
{
 LPTNODE pRoot = NULL, pNode;
 DWORD nRead, i;
 BOOL AtCR;
 TCHAR DataHold [MAX_DATA_LEN] ;
 LPTSTR pString;
 while (TRUE) {
  /* Разместить и инициализировать новый узел дерева. */
  pNode = HeapAlloc(hNode, HEAP_ZERO_MEMORY, NODE_SIZE);
  /* Считать ключ из следующей записи файла. */
  if (!ReadFile(hIn, pNode->Key, TKEY_SIZE, &nRead, NULL) || nRead != TKEY_SIZE) return pRoot;
  AtCR = FALSE; /* Считать данные до конца строки. */
  for (i = 0; i < MAX_DATA_LEN; i++) {
   ReadFile(hIn, &DataHold [i], TSIZE, &nRead, NULL);
   if (AtCR && DataHold [i] == LF) break;
   AtCR = (DataHold [i] == CR);
  }
  DataHold[i – 1] = '\0';
  /* Объединить ключ и данные — вставить в дерево. */
   pString = HeapAlloc(hData, HEAP_ZERO_MEMORY, (SIZE_T)(KEY_SIZE + _tcslen (DataHold) + 1) * TSIZE);
  memcpy(pString, pNode->Key, TKEY_SIZE);
```

```
    pString [KEY_SIZE] = '\0';
    _tcscat (pString, DataHold);
    pNode->pData = pString;
    InsertTree(&pRoot, pNode);
   } /* Конец цикла while (TRUE). */
   return NULL; /* Ошибка */
  }

  BOOL InsertTree(LPPTNODE ppRoot, LPTNODE pNode)
  /* Добавить в дерево одиночный узел, содержащий данные. */
  {
   if (*ppRoot == NULL) {
    *ppRoot = pNode;
    return TRUE;
   }
   /* Обратите внимание на рекурсивные вызовы InsertTree. */
   if (KeyCompare(pNode->Key, (*ppRoot)->Key) < 0)
InsertTree(&((*ppRoot)->Left), pNode);
   else InsertTree(&((*ppRoot)->Right), pNode);
  }

  static int KeyCompare(LPCTSTR pKey1, LPCTSTR pKey2)
  /* Сравнить две записи, состоящие из обобщенных символов. */
  {
   return _tcsncmp(pKey1, pKey2, KEY_SIZE);
  }

  static BOOL Scan(LPTNODE pNode)
  /* Рекурсивный просмотр и отображение содержимого бинарного
дерева. */
  {
   if (pNode == NULL) return TRUE;
   Scan(pNode->Left);
   _tprintf(_T ("%s\n"), pNode->pData);
   Scan(pNode->Right);
   return TRUE;
  }
```

Примечание

Очевидно, что данную реализацию дерева поиска нельзя назвать самой эффективной, поскольку дереву поиска ничто не мешает стать несбалансированным. Разумеется, о балансировке дерева поиска следовало бы позаботиться, однако на организацию управления памятью в программе это никак не повлияет.

Отображение файлов

Динамическая память, распределенная в кучах, должна физически размещаться в файле подкачки. Управление перемещением страниц между физической памятью и файлом подкачки, а также отображением файла подкачки на виртуальное адресное пространство процесса осуществляется средствами ОС, ответственными за управление памятью. По завершении выполнения процесса физическое пространство в этом файле освобождается.

Те же функциональные возможности Windows, которые обеспечивают отображение файла подкачки, позволяют отображать и обычные файлы. Отображение файлов дает следующие преимущества:

- Отпадает необходимость в выполнении операций непосредственного файлового ввода/вывода (чтения и записи).
- Структуры данных, созданные в памяти, будут сохраняться в файле для последующего использования этой же или другими программами. Необходимо тщательно следить за правильностью использования указателей, что иллюстрируется в программе 5.5.
- Становится возможным применение удобных и эффективных алгоритмов, ориентированных на работу с файлами "в памяти" (in-memory files) (сортировка, деревья поиска, обработка строк и тому подобное), которые позволяют обрабатывать хранящиеся в файлах данные даже в тех случаях, когда размеры файлов значительно превышают доступный объем физической памяти. При больших размерах файлов особенности организации страничного обмена могут оказывать заметное влияние на производительность.
- В некоторых случаях значительно повышается эффективность обработки файлов.
- Исчезает необходимость в управлении буферами и манипулировании содержащимися в них данными файлов. Всю эту тяжелую работу выполняет ОС, причем делает она это в высшей степени эффективно и надежно.

• Обеспечивается возможность разделения памяти несколькими параллельно выполняющимися процессами (глава 6) за счет отображения на их виртуальные адресные пространства одного и того же обычного файла или файла подкачки (разделение памяти несколькими процессами является одной из основных причин использования объекта отображения файла подкачки).

• Отпадает необходимость в расходовании излишнего пространства файла подкачки.

ОС сама использует методы отображения файлов для реализации DLL, а также для загрузки и выполнения исполняемых (.EXE) файлов. Библиотеки DLL описаны в конце настоящей главы.

Объекты отображения файлов

Сначала необходимо создать для открытого файла *объект отображения файла* (file mapping object), у которого имеется дескриптор, а затем отобразить этот файл или только некоторую его часть на виртуальное адресное пространство процесса. Объектам отображения можно присваивать имена, по которым к ним смогут обращаться другие процессы, разделяющие память совместно с данным процессом. Кроме того, объекты отображения файлов имеют параметры размера и атрибуты защиты.

HANDLE CreateFileMapping(HANDLE hFile, LPSECURITY_ATTRIBUTES lpsa, DWORD dwProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR lpMapName)

Возвращаемое значение: в случае успешного выполнения — дескриптор объекта отображения файла, иначе — NULL.

Параметры

hFile — дескриптор открытого файла, атрибуты защиты которого совместимы с флагами защиты, указанными параметром dwProtect. Значение этого дескриптора (тип данных HANDLE), равное 0xFFFFFFFF (его эквивалент — символическая константа INVALID_HANDLE_VALUE), соответствует системному файлу подкачки, и его можно использовать для организации разделения памяти несколькими процессами без создания отдельного файла.

LPSECURITY_ATTRIBUTES — позволяет указать атрибуты защиты объекта отображения.

dwProtect — с помощью флагов, которые приводятся ниже, определяет возможности доступа к представлению файла при его отображении. Помимо упомянутых флагов предусмотрены дополнительные флаги, имеющие специальное назначение. Так, флаг SEC_IMAGE указывает на то, что открытый файл, на основе которого создается объект отображения, является исполняемым загрузочным модулем; для получения более подробной информации обратитесь к оперативной справочной документации.

• PAGE_READONLY: страницы в указанной области отображения доступны программе только для чтения; программа не может осуществлять в них запись или запускать на выполнение. Файл с дескриптором hFile должен быть открыт с правами доступа GENERIC_READ.

• PAGE_READWRITE: предоставляет полный доступ к объекту, если файл с дескриптором hFile был открыт с правами доступа GENERIC_READ и GENERIC_WRITE.

• PAGE_WRITECOPY: при изменении отображения файла его приватная (для данного процесса) копия записывается в файл подкачки, а не в исходный файл. Отладчики могут использовать этот флаг для установки точек прерывания в разделяемом коде.

dwMaximumSizeHigh и **dwMaximumSizeLow** — соответственно, старшая и младшая 32-битовые части значения максимального размера объекта отображения файла. Если оба эти параметры равны 0, используется текущий размер файла; в случае работы с файлом подкачки указание размера является обязательным. Если предполагается, что впоследствии файл может увеличиться, укажите его предполагаемый конечный размер, и, если это необходимо, этот размер будет сразу же установлен для файла. Не пытайтесь отображать область файла, лежащую за пределами указанного размера, поскольку размер объекта отображения расти не может.

lpMapName — указатель на строку, содержащую имя объекта отображения, которое другие процессы могут использовать для разделения объекта; имя объекта чувствительно к регистру. Если не предполагается разделение памяти, используйте для этого параметра значение NULL.

На возникновение ошибок указывает возвращение функцией значения NULL (а не INVALID_HANDLE_VALUE).

Дескриптор объекта отображения файла можно получить, указав имя существующего объекта отображения. Это имя должно совпадать с тем, которое было задано во время создания открываемого объекта отображения с помощью функции CreateFileMapping. Два процесса могут разделять память, разделяя отображение файла. При этом первый процесс создает именованный объект отображения, а второй открывает этот объект, используя его имя. Если объекта отображения с указанным именем не существует, попытка его открытия будет неудачной.

HANDLE OpenFileMapping(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpMapName)

Возвращаемое значение: в случае успешного выполнения — дескриптор объекта отображения файла, иначе — NULL.

Параметр **dwDesiredAccess** использует тот же набор флагов, что и параметр dwProtect в функции CreateFileMapping. Указатель **lpMapName** должен указывать на строку с именем, совпадающим с тем, которое было задано при вызове функции CreateFileMapping. Дескриптор наследования (bInheritTable) рассматривается в главе 6.

Как несложно догадаться, для закрытия дескрипторов объектов отображения используется функция **CloseHandle**.

Отображение файла на адресное пространство процесса

Следующим шагом является распределение виртуального адресного пространства и отображение на него файла с использованием объекта отображения. С точки зрения программиста этот процесс распределения памяти аналогичен тому, который обсуждался при рассмотрении функции HeapAlloc, хотя и делает это намного грубее, оперируя более крупными блоками. В результате этого распределения возвращается указатель на распределенный блок, или представление файла (file view); различие состоит в том, что этот распределенный блок является отображением пользовательского файла, а не файла подкачки. Объект отображения файла играет ту же роль, что и куча в случае использования функции HeapAlloc.

LPVOID MapViewOfFile(HANDLE hMapObject, DWORD dwAccess, DWORD dwOffsetHigh, DWORD dwOffsetLow, SIZE_T cbMap)

Возвращаемое значение: В случае успешного выполнения — начальный адрес блока (представления файла), иначе — NULL.

Параметры

hMapObject — дескриптор объекта отображения файла, возвращенный функцией CreateFileMapping или OpenFileMapping.

dwAccess — этот параметр должен быть совместимым с разрешенными типами доступа к объекту отображения. Тремя возможными флаговыми значениями являются FILE_MAP_WRITE, FILE_MAP_READ и FILE_MAP_ALL_ACCESS. (Последний флаг является результатом применения поразрядной операции "или" к двум предыдущим флагам).

dwOffsetHigh и **dwOffsetLow** — соответственно, старшая и младшая 32-битовые части смещения начала отображаемого участка в файле. Значение этого начального адреса должно быть кратным 64 Кбайт. Чтобы начало отображаемого участка совпадало с началом файла, оба параметра следует задать равными 0.

cbMap — размер отображаемого участка файла в байтах. Если значение этого параметра установлено равным 0, то отображаться будет весь файл, существующий в момент вызова функции MapViewOfFile.

Функция **MapViewOfFileEx** аналогична функции **MapViewOfFile**, но дополнительно позволяет указать при вызове начальный адрес памяти для отображенного представления. Например, в качестве такого адреса может быть указан адрес массива в пространстве данных программы. В Windows, если затребованная область памяти уже используется для отображения, выполнение этой функции завершится с ошибкой.

Точно так же как память, распределенная из кучи, должна освобождаться при помощи функции HeapFree, необходимо отменять и отображение представления файла, которое больше не используется.

BOOL UnmapViewOfFile(LPVOID lpBaseAddress)

Взаимосвязь между адресным пространством процесса и отображаемым файлом проиллюстрирована на рис. 5.3.

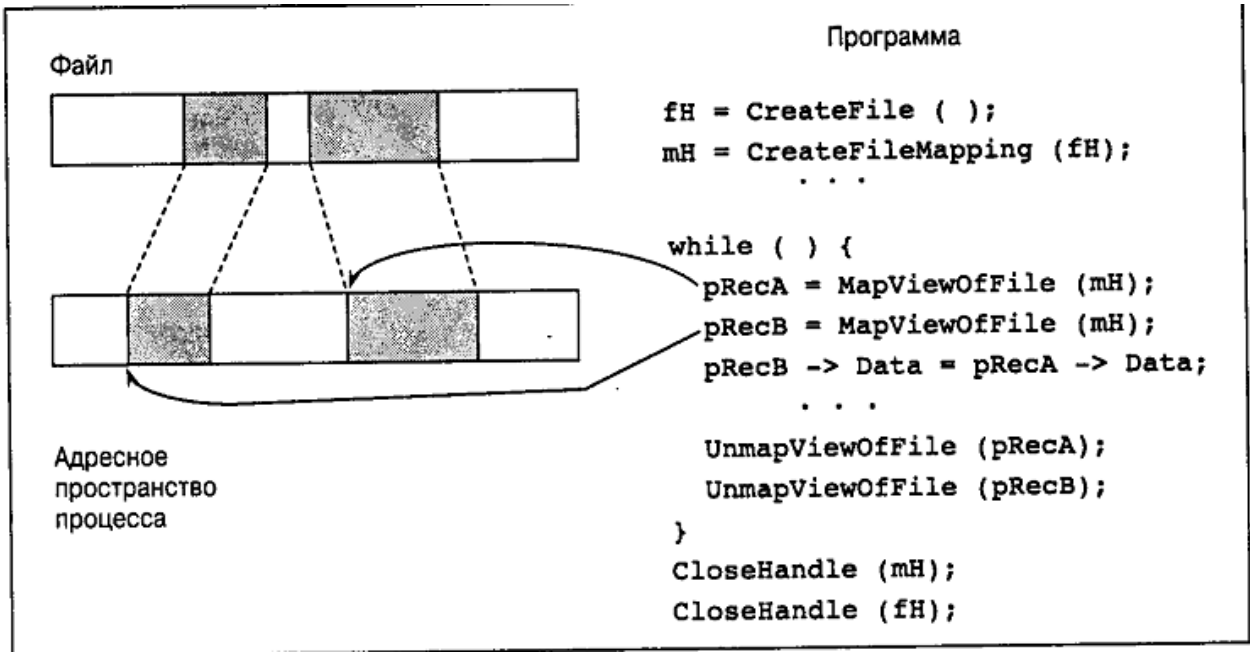

Рис. 5.3. Отображение представления файла на адресное пространство процесса

Вызов функции **FlushViewOfFile** вынуждает систему записать измененные страницы на диск. Как правило, процесс, получающий доступ к файлу через его отображение в памяти, и процесс, получающий доступ к файлу посредством обычных файловых операций ввода/вывода, будут "видеть" разные представления файла. Не решает эту проблему и выполнение файловых операций ввода/вывода без буферизации, так как представление отображаемого файла в памяти не записывается немедленно на диск.

В силу этого идея получения доступа к отображаемому файлу с помощью функций ReadFile и WriteFile не сулит ничего хорошего, поскольку согласованность данных при этом не гарантируется. С другой стороны, представления файла для процессов, получающих совместный доступ к нему через разделяемую память, будут согласованными. Если один процесс изменяет какой-либо участок памяти в области отображения файла, то другой процесс при получении доступа к соответствующему участку в своей области отображения файла получит измененные данные. Этот механизм проиллюстрирован на рис. 5.4, из которого следует, что согласованность отображенных представлений файла в двух процессах (PA и PB) действительно обеспечивается, поскольку виртуальным адресам данных в обоих процессах, несмотря на то, что эти адреса различны, соответствуют одни и те же участки

физической памяти; Естественным образом связанная с этим тема синхронизации процессов обсуждается в главах 8—10.[24]

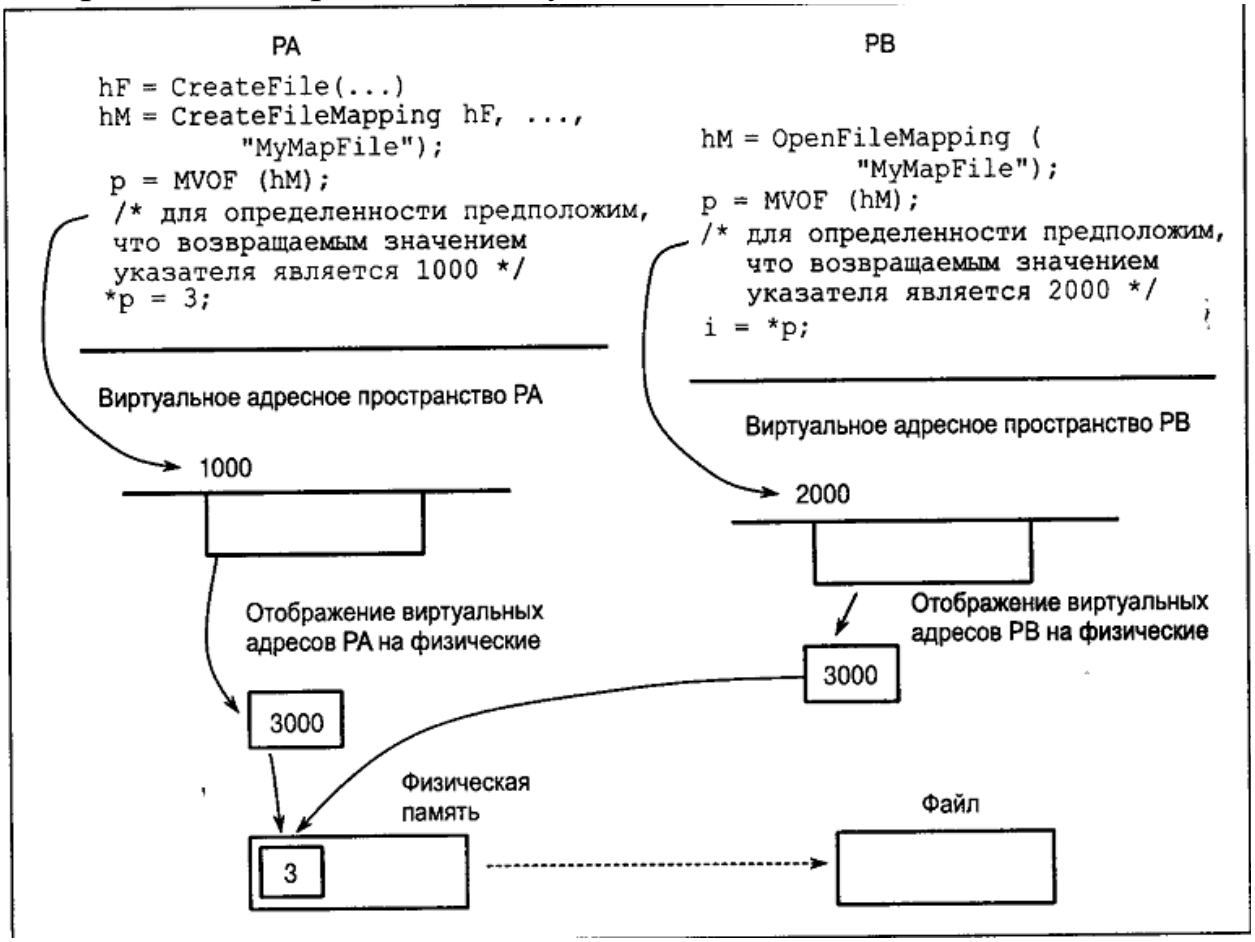

Рис. 5.4. Разделяемая память

В UNIX (выпуски SVR4 и 4.3+BSD) поддерживается функция mmap, аналогичная функции MapViewOfFile. В ее параметрах указывается та же информация, за исключением того, что объект отображения отсутствует.

Эквивалентом функции UnMapViewOfFile является функция munmap.

Для функций CreateFileMapping и OpenFileMapping эквиваленты отсутствуют. Любой обычный файл может непосредственно отображаться. В UNIX отображение файлов для разделения памяти не используется, и для этих целей предусмотрены специальные функции API, а именно, shmctl, shmat и shmdt.

Ограничения метода отображения файлов

Как уже отмечалось ранее, отображение файлов является весьма мощным и полезным средством. Существующая в Windows

диспропорция между 64-битовой файловой системой и 32-битовой адресацией снижает ценность обеспечиваемых этим средством преимуществ; Win64 свободен от этих ограничений.

Основная проблема заключается в том, что при больших размерах файлов (в данном случае, свыше 2—3 Гбайт) отображение всего файла на пространство виртуальной памяти становится невозможным. Более того, не будут доступны даже все 3 Гбайт, поскольку часть виртуального адресного пространства распределяется для других целей, а суммарный объем доступных смежных блоков будет гораздо меньше теоретического максимума. Win64 в значительной степени снимает это ограничение.

При работе с большими файлами, для которых объект отображения не может быть создан целиком, необходимо предусматривать отдельный программный код, осуществляющий отображение и отмену отображения соответствующих участков файла по мере необходимости. По сложности реализации такая методика сопоставима с организацией управления буферами в памяти, хотя необходимость в выполнении явных операций чтения и записи в данном случае отсутствует.

Двумя другими существенными недостатками метода отображения файлов являются следующие:

• Размер объекта отображения не может увеличиваться. Создавая объект отображения, вы должны заранее определить, какой максимальный размер вам может понадобиться, но во многих случаях сделать это трудно или вообще невозможно.

• Невозможно распределить память в пределах области, занимаемой представлением объекта отображения, не создав для этого собственные функции управления памятью. Было бы очень удобно, если бы существовал способ задавать объект отображения файла и указатель, возвращаемый функцией MapViewOfFile, с последующим получением дескриптора кучи.

Резюме: отображение файлов

Ниже приведена стандартная последовательность действий, которые необходимо выполнять, если используется отображение файлов:

1. Откройте файл. Убедитесь в том, что он имеет права доступа GENERIC_READ.

2. В случае создания нового файла укажите его размер, используя для этого либо функцию CreateFileMapping (шаг 3), либо функцию SetFilePointer с последующим вызовом функции SetEndOfFile.
3. Отобразите файл при помощи функций CreateFileMapping или OpenFileMapping.
4. Создайте одно или несколько представлений объекта отображения файла с помощью функции MapViewOfFile.
5. Осуществляйте доступ к файлу через обращения к памяти. Для перехода к другим участкам отображаемого файла используйте функции UnmapViewOfFile и MapViewOfFile.
6. Завершив работу, вызовите последовательно функции UnmapViewOfFile, CloseHandle для закрытия дескриптора объекта отображения и CloseHandle для закрытия дескриптора файла.

Пример: последовательная обработка файлов с использованием метода отображения

Программа atou (программа 2.4) иллюстрирует последовательную обработку файлов на примере преобразования ASCII-файлов к кодировке Unicode, приводящего к удвоению размера файла. Этот случай является идеальным для применения отображения файлов, поскольку наиболее естественным способом указанного преобразования является посимвольная обработка данных без обращения к операциям файлового ввода/вывода. Программа 5.3 сначала просто отображает входной и выходной файлы в память, предварительно вычисляя размер выходного файла путем удвоения размера входного файла, а затем осуществляет требуемое посимвольное преобразование.

Этот пример отчетливо демонстрирует, как сложность процесса отображения файлов, выполнение которого необходимо для инициализации программы, компенсируется простотой результирующей обработки. Принимая во внимание, насколько просто выполняются обычные операции файлового ввода/вывода, применение более сложного метода могло бы показаться излишним, однако это с лихвой окупается выигрышем в производительности. В приложении В показано, что версия, использующая отображение файлов, в файловых системах NTFS работает значительно быстрее по сравнению с версиями, использующими обычные способы доступа к файлам, так что некоторое усложнение программы себя полностью оправдывает.

Дополнительные результаты анализа производительности приведены на Web-сайте книги, поэтому ниже мы ограничимся лишь краткими выводами.

- Повышение производительности программ за счет использования отображения файлов наблюдается только в случае Windows NT и файловых систем NTFS.
- По сравнению с наилучшими из методик последовательной обработки файлов обеспечивается, по крайней мере, трехкратное повышение производительности .
- При работе с файлами большого размера преимущества в отношении производительности теряются. В нашем примере обычный последовательный просмотр файлов оказывается более предпочтительным, так как размер входного файла составляет около одной трети объема физической памяти. Снижение производительности метода отображения файлов в данном случае объясняется тем, что для входного файла требуется одна треть памяти, а для выходного файла, размер которого в два раза больше, — оставшиеся две трети, что заставляет нас сбрасывать отдельные части выходного файла на диск. Таким образом, в системе с объемом оперативной памяти 192 Мбайт ухудшение производительности метода отображения файлов будет наступать после достижения входными файлами размера 60 Мбайт. В большинстве случаев приходится иметь дело с файлами меньшего размера, в результате чего применение метода отображения файлов становится целесообразным.

В программе 5.3 представлена лишь функция Asc2UnMM. Основная программа совпадает с той, которая приведена в программе 2.4.

Программа 5.3. Asc2UnMM: преобразование файла с использованием метода отображения файлов

```
/* Глава 5. Asc2UnMM.c: Реализация, использующая отображение файлов. */
#include "EvryThng.h"

BOOL Asc2Un(LPCTSTR fin, LPCTSTR fOut, BOOL bFaillfExists)
{
  HANDLE hIn, hOut, hInMap, hOutMap;
  LPSTR pIn, pInFile;
```

```
  LPWSTR pOut, pOutFile;
  DWORD FsLow, dwOut;
  /* Открыть и отобразить входной и выходной файлы. */
  hIn = CreateFile(fIn, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  hInMap = CreateFileMapping(hIn, NULL, PAGE_READONLY, 0, 0,
NULL);
  pInFile = MapViewOfFile(hInMap, FILE_MAP_READ, 0, 0, 0);
  dwOut = bFailIfExists ? CREATE NEW : CREATE ALWAYS;
  hOut = CreateFile(fOut, GENERIC_READ | GENERIC_WRITE, 0,
NULL, dwOut, FILE_ATTRIBUTE_NORMAL, NULL);
  FsLow = GetFileSize (hIn, NULL); /* Установить размер
отображения. */
  hOutMap = CreateFileMapping(hOut, NULL, PAGE_READWRITE,
0, 2* FsLow, NULL);
  pOutFile = MapViewOfFile(hOutMap, FILE_MAP_WRITE, 0, 0,
(SIZE_T)(2 * FsLow));
  /* Преобразовать данные отображенного файла из ASCII в
Unicode. */
  pIn = pInFile;
  pOut = pOutFile;
  while (pIn < pInFile + FsLow) {
   *pOut = (WCHAR) *pIn;
   pIn++;
   pOut++;
  }
  UnmapViewOfFile(pOutFile);
  UnmapViewOfFile(pInFile);
  CloseHandle(hOutMap);
  CloseHandle(hInMap);
  CloseHandle(hIn);
  CloseHandle(hOut);
  return TRUE;
 }
```

Пример: сортировка отображенных файлов

Дополнительным преимуществом метода отображения файлов является то, что он допускает применение обычных алгоритмов обработки файлов в памяти компьютера. Так, сортировку данных в

памяти осуществить гораздо легче, чем сортировку записей в файле.

Программа 5.4 предназначена для сортировки файлов с записями фиксированной длины. Данная программа, sortFL, аналогична программе 5.1 в том отношении, что предполагает наличие 8-байтового ключа сортировки в начале записи, но ограничивается записями фиксированной длины. В программе 5.5 этот недостаток будет устранен за счет некоторого усложнения программы.

Сортировку выполняет описанная в файле <stdlib.h> функция qsort, входящая в состав библиотеки C. Заметьте, что эта функции требует от программиста предоставления функции, осуществляющей сравнение записей, в качестве которой нами будет использована функция KeyCompare из программы 5.2.

Структура программы достаточно проста. Сначала на основе временной копии входного файла создается объект отображения файла, затем создается единое представление объекта отображения файла в памяти, и, наконец, вызывается функция qsort. При этом какие-либо операции файлового ввода/вывода отсутствуют. Отсортированный файл направляется далее на стандартный вывод, причем в конце отображения файла добавляется нулевой символ.

Программа 5.4. sortFL: сортировка файла с использованием его отображения в памяти

```
/* Глава 5. sortFL. Сортировка файлов. Записи имеют фиксированную длину.*/
/* Использование: sortFL файл */
#include "EvryThng.h"

typedef struct _RECORD {
 TCHAR Key[KEY_SIZE];
 TCHAR Data[DATALEN];
} RECORD;

#define RECSIZE sizeof(RECORD)

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hFile = INVALID_HANDLE_VALUE, hMap = NULL;
 LPVOID pFile = NULL;
 DWORD FsLow, Result = 2;
```

```
  TCHAR TempFile[MAX_PATH];
  LPTSTR pTFile;
  /* Создать имя временного файла, предназначенного для
хранения копии сортируемого файла, которая и подвергается
сортировке. */
  /* Можно действовать и по-другому, оставив файл в качестве
постоянно хранимой сортируемой версии. */
  _stprintf(TempFile, _T("%s%s"), argv[1], _T(".tmp"));
  CopyFile(argv[1], TempFile, TRUE);
  Result = 1; /* Временный файл является вновь созданным и
должен быть удален. */
  /* Отобразить временный файл и выполнить его сортировку в
памяти. */
  hFile = CreateFile(TempFile, GENERIC_READ |
GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
  FsLow = GetFileSize(hFile, NULL);
  hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0,
FsLow + TSIZE, NULL);
  pFile = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0 /*
FsLow + TSIZE */, 0);
  qsort(pFile, FsLow / RECSIZE, RECSIZE, KeyCompare);
  /* KeyCompare – как в программе 5.2. */
  /* Отобразить отсортированный файл. */
  pTFile = (LPTSTR)pFile;
  pTFile[FsLow/TSIZE] = '\0';
  _tprintf(_T("%s"), pFile);
  UnmapViewOfFile(pFile);
  CloseHandle(hMap);
  CloseHandle(hFile);
  DeleteFile(TempFile);
  return 0;
 }
```

Описанный вариант реализации довольно прост, однако возможен и другой вариант, не требующий использования отображения файлов. Для этого достаточно распределить память, считать весь файл, выполнить его сортировку в памяти и записать на диск. По своей эффективности это решение, которое приведено на Web-сайте книги, не уступает программе 5.4, а нередко и превосходит ее, как показано в приложении B.

Базовые указатели

Как показали предыдущие примеры, во многих случаях метод отображения файлов является весьма удобным. Однако предположим, что в программе создается структура данных с указателями, ссылающимися на область отображения файла, и ожидается, что впоследствии к этому файлу будет производиться обращение. В этом случае указатели оказываются установленными относительно виртуального адреса, возвращенного функцией MapViewOfFile, и не будут иметь смысла при использовании представления объекта отображения в следующий раз. Решение состоит в том, чтобы использовать базовые указатели (based pointers), являющиеся фактически смещениями относительно другого указателя. Соответствующий синтаксис Microsoft C, доступный в Visual C++ и некоторых других системах, выглядит следующим образом:

тип _based (база) *объявление*

Ниже показаны два примера таких указателей.

```
LPTSTR pInFile = NULL;
DWORD _based (pInFile) *pSize;
TCHAR _based (pInFile) *pIn;
```

Обратите внимание на тот факт, что синтаксис требует использования символа *, хотя такая практика противоречит соглашениям Windows.

Пример: использование базовых указателей

Рассмотренные выше примеры относились к сортировке файлов в различных ситуациях. Вместе с тем, должно быть очевидным, что наша цель состояла не в обсуждении методик сортировки, а в демонстрации применения различных методов управления памятью. В программе 5.1 используется бинарное дерево поиска, которое уничтожается при переходе к сортировке очередного файла, тогда как в программе 5.4 сортируется массив фиксированных записей, отображенный в памяти компьютера. В приложении В представлены показатели производительности для различных вариантов реализации, включая и тот, который реализует программа 5.5.

Предположим, что необходимо обеспечить сопровождение постоянно существующего индексного файла, предоставляющего

отсортированный ключ исходного файла. Могло бы показаться, что очевидным решением является отображение в памяти файла, содержащего постоянно хранимые индексы в виде дерева поиска, или его формы с отсортированным ключом. Однако это решение страдает серьезным недостатком. Все указатели дерева, сохраняемые в файле, являются заданными относительно адреса, возвращенного функцией MapViewOfFile. Когда программа будет запущена в следующий раз и создаст отображение файла, эти указатели окажутся бесполезными.

Программа 5.5, которая должна применяться совместно с программой 5.6, решает эту проблему, которая проявляется всякий раз, когда отображаются структуры данных, использующие указатели. В предлагаемом решении используется ключевое слово _based, предоставляемое Microsoft C. Альтернативным вариантом было бы отображение файла в массив и обеспечение доступа к записям в представлении объекта отображения файла с помощью индекса.

Программа написана в виде еще одной версии команды sort, которой в данном случае присвоено имя sortMM. Данная версия программы отличается следующими особенностями, заслуживающими внимания:

• Записи могут иметь переменную длину.

• Программа использует первое поле в качестве ключа, но определяет его длину.

• Строятся два представления файла. Одно из них представляет исходный файл, а второе — файл, содержащий отсортированные ключи. Второй файл является *индексным файлом* (index file), каждая из записей которого содержит ключ и указатель (базовый адрес), относящийся к исходному файлу. Для сортировки индексного файла, во многом по аналогии с программой 5.4, применяется функция qsort.

• Индексный файл сохраняется и впоследствии может быть использован, причем предусмотрена возможность (параметр командной строки –I) отказаться от сортировки и использовать существующий индексный файл. Кроме того, индексный файл может быть использован для быстрого поиска ключей путем проведения бинарного поиска (возможно, с использованием входящей в библиотеку C функции bsearch) в индексном файле.

Взаимосвязь между индексным файлом и сортируемым файлом иллюстрирует рис. 5.5. Главной программой является программа 5.5, которая обеспечивает создание представлений файлов в памяти компьютера, осуществляет сортировку индексного файла и отображает результаты. Эта программа вызывает функцию CreateIndexFile, представленную программой 5.6.

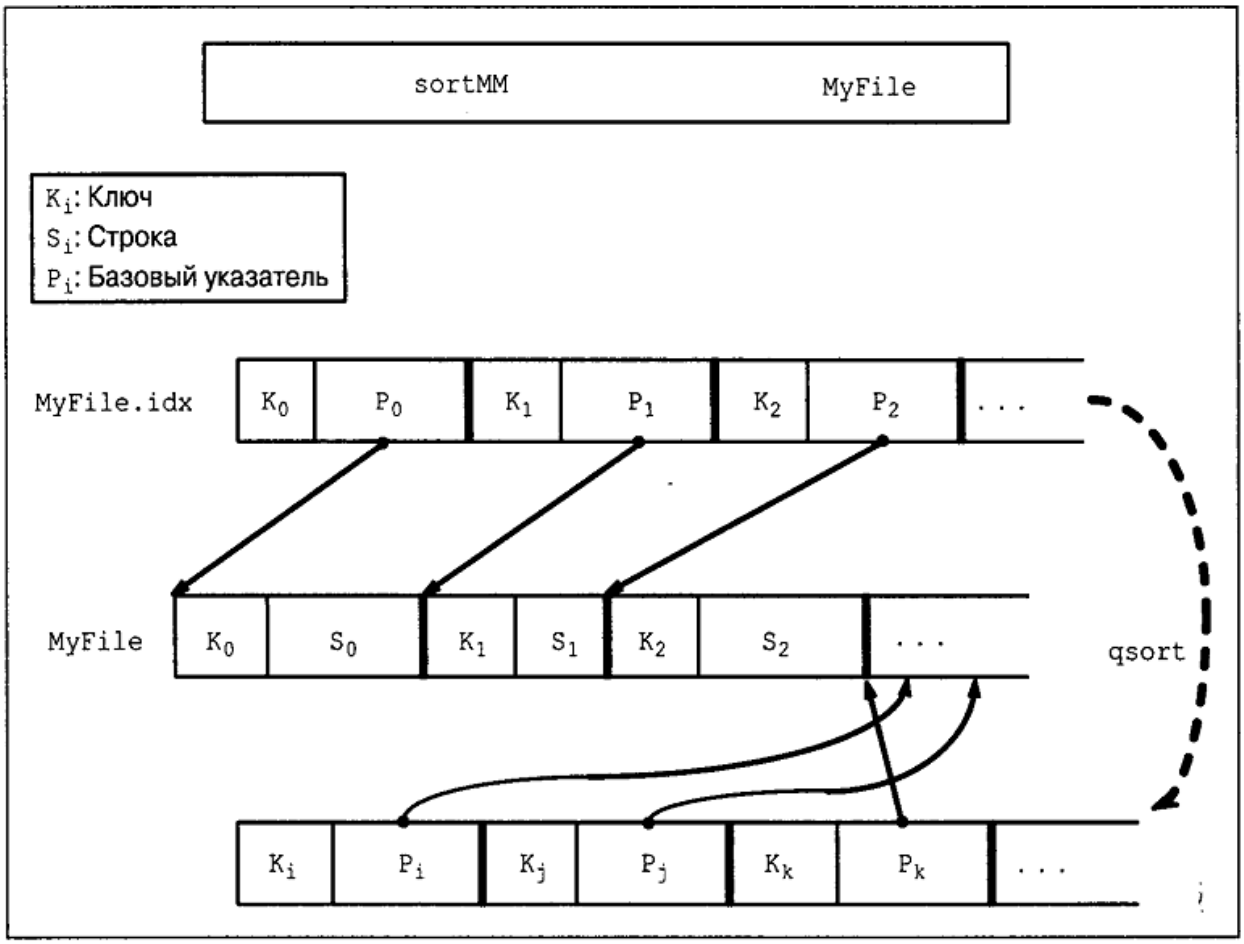

Рис. 5.5. Сортировка с использованием отображения индексного файла

Программа 5.5. sortMM: использование базовых указателей в индексном файле

```
/* Глава 5. Команда sortMM.
   Сортировка отображенного в памяти файла – только один файл. Опции:
   -r Сортировать в обратном порядке.
   -I Использовать индексный файл для получения отсортированного файла. */
   #include "EvryThng.h"
```

```
  int KeyCompare(LPCTSTR , LPCTSTR);
  DWORD CreateIndexFile (DWORD, LPCTSTR, LPTSTR);
  DWORD KStart, KSize; /* Начальная позиция и размер ключа (TCHAR) . */
  BOOL Revrs;

  int _tmain(int argc, LPTSTR argv []) {
   HANDLE hInFile, hInMap; /* Дескрипторы входного файла. */
   HANDLE hXFile, hXMap; /* Дескрипторы индексного файла. */
   HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
   BOOL IdxExists;
   DWORD FsIn, FsX, RSize, iKey, nWrite, *pSizes;
   LPTSTR pInFile = NULL;
   LPBYTE pXFile = NULL, pX;
   TCHAR _based(pInFile) *pIn;
   TCHAR IdxFlNam [MAX_PATH], ChNewLine = TNEWLINE;
   int FlIdx = Options(argc, argv, _T("rI"), &Revrs, &IdxExists, NULL);
   /* Шаг 1: открыть и отобразить входной файл. */
   hInFile = CreateFile(argv [FlIdx], GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
   hInMap = CreateFileMapping(hInFile, NULL, PAGE_READWRITE, 0, 0, NULL);
   pInFile = MapViewOfFile(hInMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
   FsIn = GetFileSize(hInFile, NULL);
   /* Шаги 2 и 3: создать имя индексного файла. */
   _stprintf(IdxFlNam, _T("%s%s"), argv[FlIdx], _T(".idx"));
   if (!IdxExists) RSize = CreateIndexFile(FsIn, IdxFlNam, pInFile);
   /* Шаг 4: отобразить индексный файл. */
   hXFile = CreateFile(IdxFlNam, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
   hXMap = CreateFileMapping(hXFile, NULL, PAGE_READWRITE, 0, 0, NULL);
   pXFile = MapViewOfFile(hXMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
   FsX = GetFileSize(hXFile, NULL);
   pSizes = (LPDWORD)pXFile; /* Поля размера в .idx-файле. */
```

```
  KSize = *pSizes;  /* Размер ключа */
  KStart = *(pSizes + 1); /* Начальная позиция ключа в записи.  */
  FsX –= 2 * sizeof(DWORD);
  /* Шаг 5: сортировать индексный файл при помощи qsort. */
  if (!IdxExists) qsort(pXFile + 2 * sizeof(DWORD), FsX / RSize,
RSize, KeyCompare);
  /* Шаг 6: отобразить входной файл в отсортированном виде. */
  pX = pXFile + 2 * sizeof(DWORD) + RSize – sizeof(LPTSTR);
  for (iKey = 0; iKey < FsX / RSize; iKey++) {
   WriteFile(hStdOut, &ChNewLine, TSIZE, &nWrite, NULL);
   /* Приведение типа pX, если это необходимо! */
   pIn = (TCHAR _based (pInFile)*) *(LPDWORD)pX;
   while ((*pIn != CR || * (pIn + 1) != LF) && (DWORD) pIn < FsIn) {
    WriteFile(hStdOut, pIn, TSIZE, &nWrite, NULL); pIn++;
   }
   pX += RSize;
  }
  UnmapViewOfFile(pInFile);
  CloseHandle(hInMap);
  CloseHandle(hInFile);
  UnmapViewOfFile(pXFile);
  CloseHandle(hXMap);
  CloseHandle(hXFile);
  return 0;
 }
```

Программа 5.6 представляет собой функцию CreateIndexFile, с помощью которой создается индексный файл. Сначала она просматривает входной файл для определения размера ключа по первой записи. После этого она должна просматривать входной файл для нахождения границ каждой из записей переменной длины для организации структуры, представленной на рис. 5.5.

Программа 5.6. sortMM: создание индексного файла

```
 DWORD CreateIndexFile(DWORD FsIn, LPCTSTR IdxFlNam,
LPTSTR pInFile) {
  HANDLE hXFile;
  TCHAR _based (pInFile) *pInScan = 0;
  DWORD nWrite;
```

```
   /* Шаг 2a: создать индексный файл. Не отображать его на данной стадии. */
   hXFile = CreateFile(IdxFlNam, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS, 0, NULL);
   /* Шаг 2b: получить первый ключ и определить его размер и начальную позицию. Пропустить пробел и получить длину ключа. */
   KStart = (DWORD) pInScan;
   while (*pInScan != TSPACE && *pInScan != TAB) pInScan++; /* Найти поле первого ключа. */
   KSize = ((DWORD)pInScan – KStart) / TSIZE;
   /* Шаг 3: просмотреть весь файл, записывая ключи и указатели записей в индексный файл. */
   WriteFile(hXFile, &KSize, sizeof(DWORD) , &nWrite, NULL);
   WriteFile(hXFile, &KStart, sizeof(DWORD), &nWrite, NULL);
   pInScan = 0;
   while ((DWORD)pInScan < FsIn) {
     WriteFile(hXFile, pInScan + KStart, KSize * TSIZE, &nWrite, NULL);
    WriteFile(hXFile, &pInScan, sizeof(LPTSTR), &nWrite, NULL);
    while ((DWORD)pInScan < FsIn && ((*pInScan != CR) || (*(pInScan + 1) != LF))) {
     pInScan++; /* Пропустить до конца строки. */
    }
    pInScan += 2; /* Пропустить CR, LF. */
   }
   CloseHandle(hXFile);
   /* Размер отдельной записи. */
   return KSize * TSIZE + sizeof(LPTSTR);
  }
```

Динамически компонуемые библиотеки

Как вы имели возможность убедиться, средства управления памятью и отображения файлов оказываются важными и полезными для широкого класса программ. Системы управления памятью используются также самими ОС, и наиболее важной и заслуживающей внимания сферой применения отображения файлов являются библиотеки DLL. DLL широко используются

приложениями Windows, являясь существенным элементом таких высокоуровневых технологий, как COM, а многие компоненты программного обеспечения поставляются в виде DLL.

Нашим первым шагом будет рассмотрение различных методов построения библиотек наиболее часто используемых функций.

Статические и динамические библиотеки

Самый непосредственный способ построения любой программы — это объединение исходных кодов всех функций, их компиляция и компоновка всех необходимых элементов в один исполняемый модуль. Чтобы упростить процесс сборки, такие функции общего назначения, как ReportError, можно поместить в библиотеку. Этот подход применялся во всех представленных до сих пор примерах программ, хотя и касался всего лишь нескольких функций, большинство из которых предназначались для вывода сообщений об ошибках.

Эта монолитная, одномодульная модель отличается простотой, однако обладает и рядом недостатков.

• Исполняемый модуль может разрастаться до больших размеров, занимая большие объемы дискового пространства и физической памяти во время выполнения и требуя дополнительных усилий для организации управления модулем и передачи его пользователям.

• При каждом обновлении потребуется повторная сборка всей программы, даже если необходимые изменения незначительны или носят локальный характер.

• Каждый исполняемый модуль тех программ в системе, которые используют эти функции, будет иметь свои экземпляры функций, версии которых могут различаться. Подобная схема компоновки приводит к тому, что при одновременном выполнении нескольких таких программ будет напрасно расходоваться дисковое пространство и, что намного существеннее, физическая память.

• Для достижения наилучшей производительности в различных средах может потребоваться использование различных версий программы, в которых применяются различные методики. Так, функция Asc2Un в программе 2.4 (atou) и программе 5.3 (Asc2UnMM) реализована по-разному. Единственный способ выполнения программ, имеющих несколько различных реализаций, — это заранее принять решение относительно того, какую из двух версий запускать, исходя из свойств окружения.

Библиотеки DLL обеспечивают возможность элегантного решения этих и других проблем.

• Библиотечные функции не связываются во время компоновки. Вместо этого их связывание осуществляется во время загрузки программы (*неявное связывание*) или во время ее выполнения *(явное связывание).* Это позволяет существенно уменьшить размер модуля программы, поскольку библиотечные функции в него не включаются.

• DLL могут использоваться для создания *общих библиотек* (shared libraries). Одну и ту же библиотеку DLL могут совместно использовать несколько одновременно выполняющихся программ, но в память будет загружена только одна ее копия. Все программы отображают код DLL на адресные пространства своих процессов, хотя каждый поток будет иметь собственный экземпляр неразделяемого хранилища в стеке. Например, функция ReportError использовалась почти в каждом из приведенных ранее примеров программ, тогда как для всех программ было бы вполне достаточно ее единственной DLL-реализации.

• Новые версии программ или другие возможные варианты их реализации могут поддерживаться путем простого предоставления новой версии DLL, а все программы, использующие эту библиотеку, могут выполняться как новая версия без внесения каких бы то ни было дополнительных изменений.

• В случае явного связывания решение о том, какую версию библиотеки использовать, программа может принимать во время выполнения. Разные библиотеки могут быть альтернативными вариантами реализации одной и той же функции или решать совершенно иные задачи, как если бы они были независимыми программами. Библиотека выполняется в том же процессе и том же потоке, что и вызывающая программа.

Библиотеки DLL, иногда в ограниченном виде, используются практически в любой ОС. Так, в UNIX аналогичные библиотеки фигурируют под названием "разделяемых библиотек" (shared libraries). В Windows библиотеки DLL используются, помимо прочего, для создания интерфейсов ОС. Весь Windows API поддерживается одной DLL, которая для предоставления дополнительных услуг вызывает ядро Windows.

Один код DLL может совместно использоваться несколькими процессами Windows, но после его вызова он выполняется как часть вызывающего процесса или потока, Поэтому библиотека может использовать ресурсы вызывающего процесса, например дескрипторы файлов, и стек потока. Следовательно, DLL должны создаваться таким образом, чтобы обеспечивалась безопасная многопоточная поддержка (thread safety). (Более подробная информация относительно DLL и безопасной многопоточной поддержки содержится в главах 8, 9 и 10. Методы создания DLL, предоставляющих многопоточную поддержку, иллюстрируются программами 12.4 и 12.5.) Кроме того, DLL могут экспортировать переменные, а также точки входа функций.

Неявное связывание

Неявное связывание, или *связывание во время загрузки* (load-time linking) является простейшей из двух методик связывания. Порядок действий в случае использования Microsoft C++ следующий:

1. После того как собраны все необходимые для новой DLL функции, осуществляется сборка DLL, а не, например, консольного приложения.
2. В процессе сборки создается библиотечный .LIB-файл, играющий роль *заглушки* (stub) для фактического кода. Этот файл должен помещаться в каталог библиотек общего пользования, указанный в проекте.
3. В процессе сборки создается также .DLL-файл, содержащий исполняемый модуль. В типичных случаях этот файл размещается в том же каталоге, что и приложение, которое будет его использовать, и приложение загружает DLL в процессе своей инициализации. Вторым возможным местом расположения DLL является рабочий каталог, а далее ОС будет осуществлять поиск .DLL-файла в системном каталоге, каталоге Windows, а также в путях доступа, указанных в переменной окружения PATH.
4. В исходном коде DLL следует предусмотреть экспортирование интерфейсов функций, о чем рассказано ниже.

Экспортирование и импортирование интерфейсов

Самое значительное изменение, которое требуется внести в функцию, прежде чем ее можно будет поместить в DLL, — это

объявить ее экспортируемой (UNIX и некоторые другие системы не требуют явного выполнения этого шага). Это достигается либо за счет использования .DEF-файла, либо, что проще и возможно в Microsoft C, за счет использования в объявлениях модификатора _declspec (dllexport) следующим образом:

```
_declspec(dllexport) DWORD MyFunction (…);
```

Далее в процессе сборки создаются .DLL-файл и .LIB-файл. .LIB-файл — это библиотека-заглушка, которая должна быть скомпонована с вызывающей программой для разрешения внешних ссылок и создания актуальных связей с . DLL-файлом во время загрузки.

Вызывающая, или *клиентская,* программа должна объявить о том, что функцию следует импортировать, используя для этого модификатор _declspec (dllexport). Стандартный метод заключается в создании включаемого файла, использующего переменную препроцессора, имя которой формируется на основе имени проекта Microsoft Visual C++, набранного в верхнем регистре и дополненного суффиксом _EXPORTS.

Вам также может потребоваться еще одно объявление. Если вызывающая (клиентская) программа написана на C++, то для нее будет определена переменная препроцессора __cplusplus, и вы должны будете указать на необходимость использования системы соглашений о вызовах, принятой в C, с помощью следующего выражения:

```
extern "C"
```

Если, например, в качестве части сборки DLL в проекте MyLibrary определена функция MyLibrary, то содержимое заголовочного файла должно быть таким:

```
#if defined(MYLIBRARY_EXPORTS)
#define LIBSPEC _declspec(dllexport)
#elif defined(__cplusplus)
#define LIBSPEC extern "C" _declspec(dllimport)
#else
#define LIBSPEC _declspec(dllimport)
#endif
LIBSPEC DWORD MyFunction (…);
```

Visual C++ автоматически определяет MYLIBRARY_EXPORTS при вызове компилятора, если проект предназначен для создания DLL MyLibrary. Клиентский проект, который использует DLL,

переменную MYLIBRARYEXPORTS не определяет, и поэтому имя функции импортируется из библиотеки.

При построении вызывающей программы укажите соответствующий .DLL-файл. Когда будете запускать вызывающую программу на выполнение, убедитесь в наличии доступа к этому файлу; часто это обеспечивается размещением .DLL-файла в одном каталоге с исполняемым файлом. Как ранее уже отмечалось, существует ряд правил поиска DLL, определяющих последовательность просмотра каталогов, в которых Windows будет осуществлять поиск указанного .DLL-файла, *а также* других DLL и исполняемых файлов, необходимых указанному файлу, прекращая этот поиск, как только будет найден первый подходящий экземпляр. Ниже приведена *стандартная последовательность просмотра каталогов при поиске,* используемая как в случае явного, так и в случае неявного связывания.

- Каталог, в котором находится загружаемое приложение.
- Текущий каталог, если он отличен от каталога, содержащего исполняемый модуль.
- Системный каталог Windows. Вы можете определить этот путь, вызвав функцию GetSystemDirectory; таковым обычно является каталог c:\WINDOWS\SYSTEM32.
- Системный каталог 16-разрядной Windows, который отсутствует в системах Windows 9x. Функция, позволяющая получить путь доступа к этому каталогу, отсутствует, и для наших целей он оказывается ненужным.
- Каталог Windows (используйте функцию GetWindowsDirectory).
- Каталоги, перечисленные в переменной окружения PATH, которые будут просматриваться в той последовательности, в какой они указаны.

Заметьте, что этот стандартный порядок просмотра каталогов при поиске можно изменить, о чем говорится в разделе "Явное связывание". Для получения более подробной информации относительно стратегии поиска посетите Web-сайт по адресу http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/loadlibrary.asp, а также ознакомьтесь с описанием функции SetDllDirectory, введенной в Windows NT 5.1 (то есть Windows XP). Изменить стратегию поиска позволяет также функция LoadLibraryEx, описанная в следующем разделе.

Применение стандартной стратегии поиска иллюстрируется в проекте Utilities, доступном на Web-сайте книги, а такие вспомогательные функции, как ReportError, используются почти в каждом примере проектов.

Возможно также экспортирование и импортирование переменных, а также точек входа функций, хотя эти возможности в примерах не иллюстрируются.

Явное связывание

Явное связывание, или *связывание во время выполнения* (run-time linking), требует, чтобы в программе содержались конкретные указания относительно того, когда именно необходимо загрузить или освободить библиотеку DLL. Далее программа получает адрес запрошенной точки входа и использует этот адрес в качестве указателя при вызове функции. В вызывающей программе функция не объявляется; вместо этого в качестве указателя на функцию объявляется переменная. Поэтому во время компоновки программы присутствие библиотеки не является обязательным. Для выполнения необходимых операций требуются три функции: LoadLibrary (или LoadLibraryEx), GetProcAddress и FreeLibrary. На 16-битовое происхождение определений функций указывает присутствие в них дальних (far) указателей и дескрипторов различных типов.

Для загрузки библиотеки служат две функции: LoadLibrary и LoadLibraryEx.

HINSTANCE LoadLibrary(LPCTSTR lpLibFileName)

HINSTANCE LoadLibraryEx(LPCTSTR lpLibFileName, HANDLE hFile, DWORD dwFlags)

В обоих случаях значением возвращаемого дескриптора (типа HINSTANCE, а не HANDLE) в случае ошибки будет NULL. Суффикс .DLL в имени файла указывать не обязательно. С помощью функций LoadLibrary можно загружать также .EXE-файлы. При указании путей доступа должны использоваться символы обратной косой черты (\); символы прямой косой черты (/) в данном случае работать не будут.

Поскольку библиотеки DLL являются совместно используемым ресурсом, системой поддерживается счетчик ссылок на каждую DLL (который увеличивается на единицу при каждом вызове любой из указанных выше функций загрузки), так что повторное

отображение фактического файла библиотеки не требуется. Функция LoadLibrary завершится с ошибкой даже в случае успешного нахождения .DLL-файла, если данная библиотека DLL неявно связана с другой DLL, найти которую программе не удалось.

Функция LoadLibraryEx аналогична функции LoadLibrary, однако имеет несколько флагов, которые оказываются полезными для указания альтернативных путей поиска и загрузки библиотек в виде файла данных. Параметр hFile зарезервирован для использования в будущем. Параметр **dwFlags** позволяет определять различные варианты поведения системы путем указания одного из трех значений:

1. LOAD_WITH_ALTERED_SEARCH_PATH: отменяет ранее описанный стандартный порядок просмотра каталогов при поиске, изменяя лишь первый из шагов стратегии поиска. Вместо каталога, из которого загружалось приложение, используется путь поиска, указанный в имени lpLibFileName.
2. LOAD_LIBRARY_AS_DATAFILE: файл воспринимается как файл данных и не требует выполнения каких-либо действий по его подготовке к запуску, на пример вызова функции DllMain.
3. DONT_RESOLVE_DLL_REFERENCE: функция DllMain для инициализаций процессов и потоков не вызывается; загрузка дополнительных модулей, на которые имеются ссылки в указанной DLL, также не производится.

Закончив работать с экземпляром DLL — возможно, с намерением загрузить другую ее версию — вы должны освободить дескриптор библиотеки, тем самым освобождая ресурсы, в том числе распределенное для библиотеки виртуальное адресное пространство. Однако DLL продолжает оставаться загруженной, если счетчик ссылок указывает на то, что она все еще используется другими процессами.

BOOL FreeLibrary(HINSTANCE hLibModule)

После загрузки библиотеки, но до ее освобождения, вы можете получить адрес любой точки входа, используя функцию GetProcAddress.

FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName)

Параметр **hModule**, несмотря на другой тип имени (HINSTANCE определен как HMODULE), является экземпляром (instance)

библиотеки, получаемым посредством вызова функции LoadLibrary или GetModuleHandle. **lpProcName** — указатель на строку, содержащую имя точки входа; это имя не может задаваться в кодировке Unicode. В случае неуспешного выполнения функция возвращает значение NULL. Слово FARPROC, означающее "длинный указатель на процедуру", является анахронизмом.

Имя файла, связанного с дескриптором hHandle, можно получить с помощью функции GetModuleFileName. Возможно и обратное: для заданного имени файла (.DLL или .EXE) функция GetModuleHandle в случае успешного выполнения возвратит дескриптор, связанный с этим файлом, если текущий процесс загрузил его.

В следующем примере показано, как использовать адрес точки входа для вызова функции.

Пример: явное связывание функций и преобразования файлов

Программа 2.4, предназначенная для преобразования кодировки текстовых файлов из ASCII в Unicode, вызывает функцию Asc2Un (программа 2.5), выполняющую обработку файла с использованием операций файлового ввода/вывода. Программа 5.3 (Asc2UnMM) представляет альтернативную функцию, которая для выполнения той же операции использует отображение файлов. Обстоятельства, при которых функция Asc2UnMM обеспечивает выигрыш в скорости выполнения преобразования, ранее уже обсуждались; в основном они сводятся к тому, что файловой системой должна быть NTFS, а размер файла не должен быть слишком большим.

Программа 5.7 является модифицированным вариантом вызывающей программы, обеспечивающим возможность принятия решения относительно того, какой вариант реализации функции преобразования должен быть загружен, во время выполнения. Программа загружает DLL, получает адрес точки входа Asc2Un и вызывает функцию. В данном случае существует только одна точка входа, но реализовать вариант с несколькими точками входа не составляет особого труда. Основная программа является, по существу, той же, что и прежде, за исключением того, что библиотека DLL, которую необходимо использовать, указывается в виде параметра командной строки. В упражнении 5.9 вам предлагается написать вариант программы, в котором нужная DLL определяется на основе свойств системы и файла. Обратите

внимание на то, каким образом осуществляется приведение типа адреса FARPROC к типу соответствующей функции с использованием необходимого в этом случае, но довольно сложного, синтаксиса C.

Программа 5.7. atouEL: преобразование файлов с использованием явного связывания

```
/* Глава 5. Версия atou, использующая явное связывание. */
#include "EvryThng.h"

int _tmain(int argc, LPTSTR argv[]) {
 /* Объявить переменную Asc2Un как функцию. */
 BOOL (*Asc2Un)(LPCTSTR, LPCTSTR, BOOL);
 DWORD LocFileIn, LocFileOut, LocDLL, DashI;
 HINSTANCE hDLL;
 FARPROC pA2U;
 LocFileIn = Options(argc, argv, _T("i"), &DashI, NULL);
 LocFileOut = LocFileIn + 1;
 LocDLL = LocFileOut + 1;
 /* Проверить существование файла, а также опущен ли параметр DashI. */
 /* Загрузить функцию преобразования из ASCII в Unicode. */
 hDLL = LoadLibrary(argv[LocDLL]);
 if (hDLL == NULL) ReportError(_T("Не удается загрузить DLL."), 1, TRUE);
 /* Получить адрес точки входа. */
 pA2U = GetProcAddress(hDLL, "Asc2Un");
 if (pA2U == NULL) ReportError(_T("Не найдена точка входа."), 2, TRUE);
 /* Привести тип указателя. Здесь можно использовать typedef. */
 Asc2Un = (BOOL(*)(LPCTSTR, LPCTSTR, BOOL))pA2U;
 /* Вызвать функцию. */
 Asc2Un(argv[LocFileIn], argv[LocFileOut], FALSE);
 FreeLibrary(hDLL);
 return 0;
}
```

Создание библиотек DLL на основе функции Asc2Un

Программа тестировалась с двумя функциями преобразования файлов, которые должны были создаваться в виде библиотек DLL, имеющих различные имена, но идентичные точки входа. В данном случае существует только одна точка входа. Единственным существенным изменением в исходном коде является добавление модификатора _declspec(dllexport) для экспортирования функции.

Точки входа библиотеки DLL

Для каждой создаваемой DLL вы можете указать точку входа запуска библиотеки, которая обычно автоматически вызывается при каждом подключении или отключении процесса. В то же время, в функции LoadLibraryEx предусмотрена опция, позволяющая подавить вызов точки входа. В случае неявно связываемых (связываемых во время выполнения) библиотек DLL подключение и отключение процесса происходит, соответственно, при его запуске и завершении. В случае же явно связываемых DLL это осуществляется при вызове функций LoadLibrary, LoadLibraryEx и FreeLibrary.

Кроме того, точка входа вызывается всякий раз, когда процесс создает новый поток (глава 7) или прекращает его выполнение.

Точкой входа с именем DllMain, прототип которой приводится ниже, мы воспользуемся в полной мере только в главе 12 (программа 12.4), где она предоставит потокам удобный способ управления ресурсами и так называемыми локальными областями хранения потоков (Thread Local Storage, SLT) в DLL с многопоточной поддержкой.

BOOL DllMain(HINSTANCE hDll, DWORD Reason, LPVOID Reserved)

Параметр hDll является дескриптором экземпляра DLL, возвращенным функцией LoadLibrary. Значение NULL параметра Reserved указывает на то, что подключение процесса к библиотеке произошло в результате вызова функции Load-Library; иные значения этого параметра свидетельствуют о подключении к библиотеке в результате неявного связывания во время загрузки. Подобным образом, к значению NULL параметра Reserved приводит и отключение процесса от библиотеки в результате вызова функции FreeLibrary.

Параметр Reason может иметь одно из четырех значений: DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH,

DLL_THREAD_DETACH и DLL_PROCESS_DETACH. Функции точки входа DLL обычно используют операторы switch и в качестве индикатора успешного выполнения возвращают значение TRUE.

Система сериализует вызовы DllMain таким образом, что в каждый момент времени выполнять ее может только один поток (к подробному обсуждению потоков мы приступим в главе 7). Эта сериализация весьма существенна, поскольку операции инициализации, которые должна выполнять DllMain, не должны прерываться до их завершения. По этой же причине внутри точки входа не рекомендуется использовать блокирующие вызовы функций, например, функций ввода/вывода или функций ожидания (см. главу 8), поскольку они будут препятствовать запуску точки входа другими потоками. В частности, не следует вызывать внутри точки входа DLL функции LoadLibrary и LoadLibraryEx, поскольку это будет порождать дополнительные вызовы точек входа DLL.

Функция DisableThreadLibraryCalls отменяет отправку указанному экземпляру DLL уведомлений о подключении и отключении потоков. Запрет отправки уведомлений может пригодиться в тех случаях, когда потоки не нуждаются в каких-либо уникальных ресурсах во время инициализации.

Управление версиями DLL

При использовании DLL обычно проявляются трудности, обусловленные обновлением библиотек за счет введения новых символов и добавления новых средств. Основное преимущество DLL заключается в том, что несколько приложений могут совместно использовать одну и ту же библиотеку, находящуюся в памяти. Вместе с тем, это порождает целый ряд осложнений, связанных с совместимостью версий, что иллюстрируется приведенными ниже примерами.

- В результате добавления новых функций в случае неявного связывания могут стать недействительными смещения, определенные для приложений во время компоновки с .lib-файлами. От этой проблемы можно избавиться, применив явное связывание.
- Поведение новых версий функций может быть иным, в результате чего существующие приложения могут испытывать проблемы, если не будут своевременно обновлены.

• Для приложений, использующих обновленную функциональность DLL, возможны случая связывания с прежними версиями DLL.

Проблемы совместимости различных версий DLL, носящие жаргонное название "кошмара DLL", не являются столь острыми, если в одном каталоге поддерживать только одну версию DLL. Однако предоставить отдельный каталог для каждой из различных версий вовсе не так просто, как может показаться. Существует несколько других вариантов решения этой проблемы.

• Можно использовать номер версии DLL в именах .DLL– и .LIB-файлов, обычно в виде суффикса. Так, чтобы соответствовать номеру версии, используемой в данной книге, в примерах, приведенных на Web-сайте книги, и во всех проектах используются файлы Utility_3_0.LIB и Utility_3_0.DLL. Применяя явное или неявное связывание, приложения могут формулировать свои требования к версиям и получать доступ к файлам с различными именами. Такое решение характерно для UNIX-приложений.

• Компания Microsoft ввела понятие параллельных DLL (side-by-side DLL), или сборок (assemblies) и компонентов (components). При таком подходе в приложение необходимо включать объявление на языке XML, в котором определяются требования к DLL. Рассмотрение этой темы выходит за рамки данной книги, однако дополнительную информацию вы можете получить на Web-сайте компании Microsoft, в разделе, посвященном вопросам разработки приложений.

• Платформа .NET Framework предоставляет дополнительные средства поддержки выполнения приложений в условиях сосуществования различных версий DLL.

В примерах проектов, используемых в данной книге, используется первый из отмеченных подходов, предусматривающий включение номеров версий в имена файлов. С целью предоставления дополнительной поддержки, обеспечивающей возможность получения приложениями информации о DLL, во всех DLL реализована функция DllGetVersion. Кроме того, Microsoft предоставляет эту косвенно вызываемую функцию в качестве стандартного средства получения информации о версии в динамическом режиме. Указанная функция имеет следующий прототип:

HRESULT CALLBACK DllGetVersion(DLLVERSIONINFO *pdvi)

Информация о DLL возвращается в структуре DLLVERSIONINFO, в которой имеются поля типа DWORD для параметров cbSize (размер структуры), dwMajorVersion, dwMinorVersion, dwBuildNumber и dwPlatformID. В последнем поле, dwPlatformID, может быть установлено значение DLLVER_PLATFORM_NT, если библиотека не выполняется под управлением Windows 9x, или DLLVER_PLATFORM_WINDOWS, если это ограничение отсутствует. В поле cbSize должно находиться значение sizeof (DLLVERSIONINFO). В случае успешного выполнения функция возвращает значение NOERROR. Функция DllGetVersion реализована в проекте Utility_3_0.

Резюме

Система управления памятью Windows предоставляет следующие возможности:

- Использование средств Windows, осуществляющих управление кучей, а также обработчиков исключений для обнаружения и обработки ошибок, возникающих при распределении памяти, значительно упрощает логическую организацию.
- Использование нескольких независимых куч обладает рядом преимуществ по сравнению с распределением памяти из одной кучи.
- Методы отображения файлов, доступные в UNIX, но не предоставляемые библиотекой C, обеспечивают обработку файлов в памяти, что было проиллюстрировано несколькими примерами. Отображение файлов в памяти осуществляется независимо от управления кучей и упрощает решение многих задач программирования. Преимущества использования отображения файлов подтверждаются данными о достигаемом за счет этого повышении производительности, приведенными в приложении B.
- DLL являются важным специальным случаем отображения файлов и могут загружаться либо явным, либо неявным образом. DLL, предназначенные для использования многими приложениями, должны предоставлять информацию о версии библиотеки.

Использование явного связывания

DLL и явное связывание имеют фундаментальное значение для использования модели COM, которая широко применяется при разработке программного обеспечения Windows. Важность функций LoadLibrary и GetProcAddress продемонстрирована в главе 1 книги [3].

Упражнения

5.1. Спроектируйте и проведите эксперименты для оценки выигрыша в производительности, достигаемого за счет использования флага HEAP_NO_SERIALIZE при вызове функций HeapCreate и HeapAlloc. Как зависит этот показатель от размера кучи и размера блока? Заметна ли разница в результатах для различных версий Windows? На Web-сайте книги находится программа HeapNoSr.c, которая поможет вам приступить к выполнению этого и следующего упражнений.

5.2. Измените тестовую программу из предыдущего упражнения таким образом, чтобы она позволяла определить, генерирует ли функция malloc исключения или возвращает нулевой указатель в случае нехватки памяти. Является ли обнаруженное поведение функции корректным? Сравните также производительность, обеспечиваемую функцией malloc, с результатами предыдущего упражнения.

5.3. Доля накладных издержек при распределении памяти из кучи колеблется в зависимости от используемой версии Windows, что особенно заметно в случае выходящих из употребления версий Windows 9x. Спроектируйте и проведите эксперимент для определения количества блоков памяти фиксированного размера, которые каждая из систем предоставляет в одной куче. Используя SEH для определения того момента, когда распределенными оказываются все блоки, вы значительно упростите программу. Подобным образом ведет себя программа clear.c, находящаяся на Web-сайте книги, если игнорировать часть ее кода, ответственную за явное тестирование ОС. Между прочим, эта программа используется в некоторых тестах по измерению временных характеристик для гарантии того, что данные, полученные в процессе выполнении предыдущего теста, не остались в памяти.

5.4. Путем изменения программы sortFL (программа 5.4) создайте программу sortHP, распределяющую память для буфера, размер которого достаточно велик, чтобы в нем уместился весь файл, и

выполните считывание файла в этот буфер. Отображение файла использовать не следует. Сравните производительность обеих программ.

5.5. В программе 5.5 применены указатели типа _base, специфические для Microsoft C. Если ваш компилятор не поддерживает это средство (но в любом случае — просто в качестве упражнения) переделайте программу 5.5, используя для генерации значений базового указателя макрос, массив или иной механизм.

5.6. Напишите программу поиска записей по указанному ключу в файле, проиндексированном с применением программы 5.5. Для этой цели удобно воспользоваться функцией bsearch, входящей в состав библиотеки C.

5.7. Реализуйте программу tail из главы 3, используя отображение файлов.

5.8. Поместите вспомогательные функции ReportError, PrintStrings, PrintMsg и ConsolePrompt в DLL и перекомпонуйте некоторые из программ, с которыми мы работали раньше. Проделайте то же самое с функциями Options и GetArgs, предназначенными, соответственно, для обработки параметров командной строки и аргументов. Важно, чтобы как вспомогательная DLL, так и вызывающая программа использовали также и библиотеку C в виде DLL. Например, в Visual C++ и Visual Studio 6.0 выберите, начав со строки главного меню, следующие команды: Project (Проект), Settings (Параметры), вкладку C/C++, Category (Code Generation) (Категория (Генерация кода)), Use Run-Time Library (Multithreaded DLL) (Использовать библиотеку времени выполнения (многопоточная DLL)). Заметьте, что библиотеки DLL, вообще говоря, должны обеспечивать многопоточную поддержку, поскольку они будут использоваться потоками нескольких процессов. Пример возможного решения содержится в проекте Utilities_3_0, доступном на Web-сайте книги.

5.9. Измените программу 5.7 таким образом, чтобы решение относительно того, какую DLL следует использовать, базировалось на размере файла и конфигурации системы. .LIB-файл здесь не требуется, поэтому продумайте, как отменить его генерацию. Для определения типа файловой системы используйте функцию GetVolumeInformation.

5.10. Создайте дополнительные DLL для функции преобразования из предыдущего упражнения, каждая версия которых использует иную методику обработки файлов, и расширьте вызывающую программу таким образом, чтобы она сама решала, когда и какую версию использовать.