Глава 14

Получение системной информации

В этой главе...

В этой главе вы узнаете, как создать полнофункциональную утилиту SysInfo, предназначенную для просмотра жизненно важных параметров системы. В процессе разработки этого приложения вы познакомитесь с менее известными функциями API, позволяющими получить доступ к низкоуровневой общесистемной информации о процессах, потоках, модулях, кучах, драйверах и страницах. В этой главе также рассматриваются различные способы получения этой информации в Windows 95/98 и Windows NT. С помощью SysInfo вы сможете получить информацию о свободных ресурсах памяти, данные о версии Windows, значения переменных окружения и список загруженных модулей. При этом вы не только вникнете в мельчайшие детали использования функций API, но и научитесь интегрировать полученную от системы информацию в функциональный и эстетически привлекательный интерфейс пользователя. Кроме того, вы узнаете, какие функции Win32 разработаны для замены функций API Windows 3.x.

Для чего же может понадобиться информация от Windows? Во-первых, для удовлетворения собственного любопытства (ведь все мы в душе немножечко хакеры, и одно только ощущение больших возможностей делает нас еще сильнее). Во-вторых (спустимся на землю), вам, возможно, придется написать программу, в которой потребуется получить доступ к переменным окружения, чтобы найти определенные файлы. В-третьих, у вас может возникнуть необходимость в информации обо всех загруженных модулях, чтобы вручную удалить их из памяти. В-четвертых, в-пятых... — словом, всех причин не перечесть.

Получение общей информации о системе

Для начала покажем, как получить системную информацию с помощью функции API, которая остается постоянной при смене версий Win32. Код этого приложения приобретет больше смысла, если сначала познакомиться с его пользовательским интерфейсом. Причем на сей раз сделаем это “с черного хода”, т.е. начнем с дочерней формы приложения. Эта форма (рис. 14.1) называется InfoForm и используется для отображения различных параметров системы и процессов, а именно: информации о памяти и аппаратных средствах, версии операционной системы (ОС), некоторых данных о каталогах и переменных окружения.

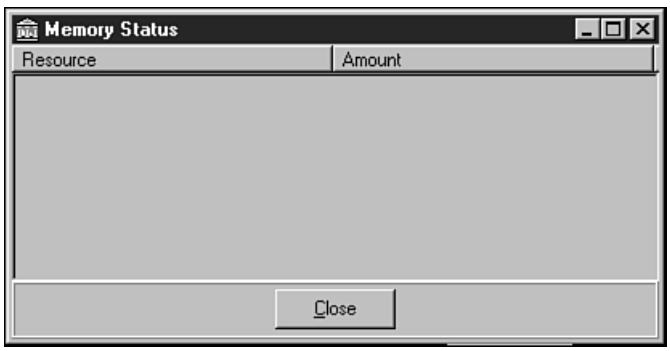

Рис. 14.1. Дочерняя форма InfoForm

Содержимое формы очень простое. В ее состав входит пользовательский компонент THeaderListbox (см. главу 21, “Создание пользовательских компонентов в Delphi”) и кнопка (компонент TButton). Напомню, что элемент управления THeaderListbox представляет собой сочетание элементов управления THeader (заголовок) и TListbox (список). При изменении размера разделов заголовка размер окна списка также меняется соответствующим образом. Элемент управления THeaderListbox (именуемый InfoLB) предназначен для отображения перечисленных выше видов информации.

Форматирование строк

В этом приложении широко используется функция Format() для форматирования встроенных строк с данными, полученными от ОС во время выполнения приложения. Эти строки определяются в разделе const главного модуля следующим образом:

```
const
  {  Строки состояния памяти }
  SMemUse    = 'Memory in useq%d%%';
        // Используемая память в байтах
```

```
STotMem   = 'Total physical memoryq$%.8x bytes';
      // Общая физическая память в байтах
SFreeMem  = 'Free physical memoryq$%.8x bytes';
      // Свободная физическая память в байтах
STotPage  = 'Total page file memoryq$%.8x bytes';
      // Общая память файлов подкачки в байтах
SFreePage = 'Free page file memoryq$%.8x bytes';
      // Свободная память файлов подкачки в байтах
STotVirt  = 'Total virtual memoryq$%.8x bytes';
      // Общая виртуальная память в байтах
SFreeVirt = 'Free virtual memoryq$%.8x bytes';
      // Свободная виртуальная память в байтах
{  Строки информации о версии ОС }
SOSVer    = 'OS Versionq%d.%d';
      // Версия операционной системы
SBuildNo  = 'Build Numberq%d';
      // Номер версии
SOSPlat   = 'Platformq%s';
      // Платформа
SOSWin32s = 'Windows 3.1x running Win32s';
SOSWin95  = 'Windows 95';
SOSWinNT  = 'Windows NT';
{  Строки системной информации }
SProc     = 'Processor Arhitectureq%s';
      // Архитектура процессора
SPIntel   = 'Intel';
SPageSize = 'Page Sizeq$%.8x bytes';
      // Размер страницы
SMinAddr  = 'Minimum Application Addressq$%p';
      // Минимальный адрес приложения
SMaxAddr  = 'Maximum Application Addressq$%p';
      // Максимальный адрес приложения
SNumProcs = 'Number of Processorsq%d';
      // Число процессоров
SAllocGra = 'Allocation Granularityq$%.8x bytes';
      // Степень разбиения при выделении ресурсов
SProcLevl = 'Processor Levelq%s';
      // Уровень процессора
SIntel3   = '80386';
SIntel4   = '80486';
SIntel5   = 'Pentium';
SIntel6   = 'Pentium Pro';
SProcRev  = 'Processor Revisionq%.4x';
      // Модификация процессора
{  Строки информации о каталогах }
SWinDir   = 'Windows directoryq%s';
      // Каталог Windows
SSysDir   = 'Windows system directoryq%s';
      // Системный каталог Windows
SCurDir   = 'Current directoryq%s';
      // Текущий каталог
```

Возможно, вас удивило присутствие буквы `q` в середине каждой строки. При отображении этих строк свойство `DelimChar` элемента управления `InfoLB` установлено равным `q`, а это значит, что символ `q` используется в окне списка в качестве разделителя между столбцами.

Существует три основных причины использования функции `Format()` с встроенными строками вместо отдельного форматирования строковых литералов.

- *Лаконичность*. Поскольку функция `Format()` принимает в качестве параметров различные типы данных, вам не нужно “утяжелять” программу вызовами таких функций, как `IntToStr()` или `IntToHex()`, которые форматируют различные типы параметров для отображения на экране.
- *Гибкость*. Функция `Format()` без труда справляется с несколькими типами данных. В этом случае используются строки формата `%s` и `%d`, чтобы отформатировать строковые и числовые данные, что является более гибким вариантом обработки данных.
- *Простота поддержки*. Хранение строк в отдельном программном блоке позволяет быстрее и проще найти их, дополнить или отредактировать в случае необходимости.

На заметку

Для отображения одиночного символа процента в форматированной строке используйте двойной знак процента (%%).

Получение информации о состоянии памяти

Первый бит системной информации, который можно получить для заполнения элемента управления InfoLB, относится к определению состояния памяти. Причем наше любопытство в этом отношении удовлетворяется путем вызова функции API GlobalMemoryStatus(), которая справляется со своей задачей благодаря доступу к var-параметру типа TMemoryStatus. Запись TMemoryStatus определяется следующим образом:

```
type
  TMemoryStatus = record
    dwLength: DWORD;
    dwMemoryLoad: DWORD;
    dwTotalPhys: DWORD;
    dwAvailPhys: DWORD;
    dwTotalPageFile: DWORD;
    dwAvailPageFile: DWORD;
    dwTotalVirtual: DWORD;
    dwAvailVirtual: DWORD;
  end;
```

- Первое поле в этой записи, dwLength, описывает длину записи TMemoryStatus. Необходимо инициализировать это поле значением SizeOf(TMemoryStatus) до вызова функции GlobalMemoryStatus(). Это позволит Windows изменять размер записи в будущих версиях, поскольку она сможет различать версии на основе значения первого поля.
- В поле dwMemoryLoad содержится число от 0 до 100, на основании которого вы получите общее представление об использовании памяти: 0 означает, что память вообще не используется, а 100 говорит о занятости всей памяти.
- Поле dwTotalPhys показывает общее число байтов физической памяти (объем памяти ОЗУ, установленного в компьютере), а поле dwAvailPhys — объем свободной в данный момент физической памяти.
- Поле dwTotalPageFile показывает общее число байтов, которые может сохранить на жестком диске файл (файлы) подкачки. Это число не совпадает с размером файла подкачки на диске. Поле dwAvailPageFile определяет объем еще доступной памяти из этого общего значения.
- Поле dwTotalVirtual показывает общее число байтов виртуальной памяти, используемой в вызывающем процессе, а поле dwAvailVirtual — объем этой памяти, доступной для вызывающего процесса.

С помощью следующего кода можно получить информацию о состоянии памяти, а также заполнить список этой информацией:

```
procedure TInfoForm.ShowMemStatus;
var
  MS: TMemoryStatus;
begin
  InfoLB.DelimChar := 'q';
  MS.dwLength := SizeOf(MS);
  GlobalMemoryStatus(MS);
  with InfoLB.Items, MS do
  begin
    Clear;
    Add(Format(SMemUse, [dwMemoryLoad]));
    Add(Format(STotMem, [dwTotalPhys]));
    Add(Format(SFreeMem, [dwAvailPhys]));
    Add(Format(STotPage, [dwTotalPageFile]));
    Add(Format(SFreePage, [dwAvailPageFile]));
    Add(Format(STotVirt, [dwTotalVirtual]));
    Add(Format(SFreeVirt, [dwAvailVirtual]));
  end;
  InfoLB.Sections[0].Text := 'Resource'; // Ресурс
  InfoLB.Sections[1].Text := 'Amount'; // Объем
```

```
  Caption:= 'Memory Status'; // Состояние памяти
end;
```

Внимание Не забудьте перед вызовом функции `GlobalMemoryStatus()` инициализировать поле `dwLength`.

На рис. 14.2 показана форма `InfoForm`, отображающая информацию о состоянии памяти.

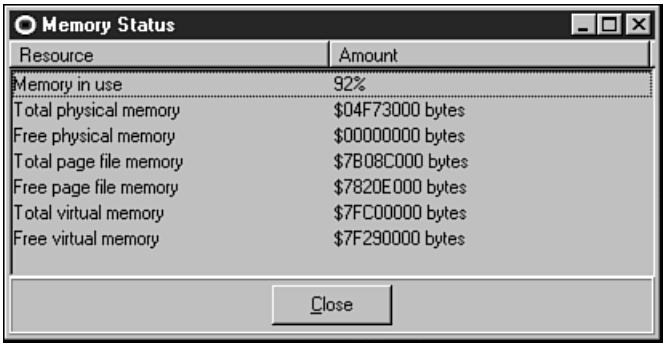

Рис. 14.2. Отображение информации о состоянии памяти

Получение информации о версии операционной системы

Вызвав функцию API `GetVersionEx()`, вы получите информацию о версии Windows и Win32, установленной на вашем компьютере. Эта функция принимает по ссылке только один параметр — запись `TOSVersionInfo`, которая определяется следующим образом:

```
type
  TOSVersionInfo = record
    dwOSVersionInfoSize: DWORD;
    dwMajorVersion: DWORD;
    dwMinorVersion: DWORD;
    dwBuildNumber: DWORD;
    dwPlatformId: DWORD;
    szCSDVersion: array[0..126] of AnsiChar; {Строка поддержки для использования PSS}
  end;
```

- Поле `dwOSVersionInfoSize` должно быть инициализировано значением `SizeOf(TOSVersionInfo)` до вызова функции `GetVersionEx()`.
- Поле `dwMajorVersion` содержит старший номер версии ОС. Другими словами, если версия ОС имеет номер 4.0, то в этом поле будет записано число `4`.
- Поле `dwMinorVersion` содержит младший номер версии ОС. Другими словами, если версия ОС имеет номер 4.0, то в этом поле будет записано число `0`.
- В поле `dwBuildNumber` содержится номер ОС, который хранится в ее самом младшем слове.
- Поле `dwPlatformId` описывает текущую платформу Win32. Этот параметр может иметь любое из перечисленных ниже значений.

Значение	*Платформа*
`VER_PLATFORM_WIN32s`	Win32s в Windows 3.1
`VER_PLATFORM_WIN32_WINDOWS`	Win32 в Windows 95
`VER_PLATFORM_WIN32_NT`	Windows NT

- Поле `szCSDVersion` содержит дополнительную информацию об ОС. Это поле часто хранит пустую строку.

Следующая процедура заполняет элемент управления `InfoLB` информацией о версии ОС:

```
procedure TInfoForm.GetOSVerInfo;
var
```

```
  VI: TOSVersionInfo;
begin
  VI.dwOSVersionInfoSize := SizeOf(VI);
  GetVersionEx(VI);
  with InfoLB.Items, VI do
  begin
    Clear;
    Add(Format(SOSVer, [dwMajorVersion, dwMinorVersion]));
    Add(Format(SBuildNo, [LoWord(dwBuildNumber)]));
    case dwPlatformID of
      VER_PLATFORM_WIN32S:        Add(Format(SOSPlat, [SOSWin32s]));
      VER_PLATFORM_WIN32_WINDOWS: Add(Format(SOSPlat, [SOSWin95]));
      VER_PLATFORM_WIN32_NT:      Add(Format(SOSPlat, [SOSWinNT]));
    end;
  end;
end;
```

На заметку

В Windows 3.x функция `GetVersion()` получала аналогичную информацию о версии ОС. Но поскольку вы сейчас работаете под управлением Win32, используйте функцию `GetVersionEx()`, которая предоставляет более детальную информацию, чем функция `GetVersion()`.

Получение информации о каталогах

Операционная система активно использует каталоги `Windows` и `System` для хранения DLL, драйверов, приложений и `INI`-файлов. Кроме того, Win32 поддерживает текущий каталог для каждого процесса. Вполне вероятно, что в период создания приложений Win32 вы попадете в ситуацию, когда вам понадобится получить точную информацию о расположении этих каталогов. И тогда вы скажете спасибо Win32 API за три функции, с помощью которых вы легко сможете получить эти данные.

О функциях `GetWindowsDirectory()`, `GetSystemDirectory()` и `GetCurrentDirectory()` можно сказать, что они очень просты в применении. Каждая из них принимает в качестве первого параметра указатель на буфер, в который копируется строка результата, а в качестве второго параметра — размер этого буфера. Результат, копируемый в буфер, представляет собой строку, содержащую путь, причем эта строка завершается ограничивающим нуль-символом. Надеемся, что по имени функции вы сможете легко определить, информацию о каком каталоге возвращает каждая функция. Если это не так, остается надеяться лишь на то, что вы зарабатываете себе на жизнь отнюдь не программированием.

В этом методе используется временный массив элементов типа `char`, в которые заносится информация о соответствующем каталоге. Затем информация из массива добавляется в элемент управления `InfoLB`, что видно из следующего кода:

```
procedure TInfoForm.GetDirInfo;
var
  S: array[0..MAX_PATH] of char;
begin
  {  Получаем каталог Windows }
  GetWindowsDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SWinDir, [S]));
  { Получаем системный каталог Windows }
  GetSystemDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SSysDir, [S]));
  { Получаем текущий каталог процесса}
  GetCurrentDirectory(SizeOf(S), S);
  InfoLB.Items.Add(Format(SCurDir, [S]));
end;
```

На заметку

Функции `GetWindowsDir()` и `GetSystemDir()` из Windows 3.x API под управлением Win32 недоступны.

Получение системной информации

В интерфейсе Win32 API предусмотрена процедура `GetSystemInfo()`, которая, в свою очередь, считывает некоторые "интимные" подробности о работе операционной системы. Эта процедура принимает по ссылке один параметр типа `TSystemInfo` и заполняет эту запись соответствующими значениями. Запись `TSystemInfo` определяется следующим образом:

```
type
  PSystemInfo = ^TSystemInfo;
  TSystemInfo = record
    case Integer of
      0: (
        dwOemId: DWORD);
      1: (
        wProcessorArchitecture: Word;
        wReserved: Word;
        dwPageSize: DWORD;
        lpMinimumApplicationAddress: Pointer;
        lpMaximumApplicationAddress: Pointer;
        dwActiveProcessorMask: DWORD;
        dwNumberOfProcessors: DWORD;
        dwProcessorType: DWORD;
        dwAllocationGranularity: DWORD;
        wProcessorLevel: Word;
        wProcessorRevision: Word);
  end;
```

- Поле `dwOemId` используется для Windows 95. Это поле всегда устанавливается равным `0` или значению `PROCESSOR_ARCHITECTURE_INTEL`.
- Под управлением Windows NT используется поле `wProcessorArchitecture`, которое описывает тип архитектуры процессора, при котором вы работаете в данный момент. Но, поскольку Delphi предназначена только для процессоров Intel, только этот тип и может составлять содержимое этого поля. Если же не ориентироваться лишь на Delphi (и ради полноты информации), то в этом поле может храниться одно из следующих значений:

```
PROCESSOR_ARCHITECTURE_INTEL
PROCESSOR_ARCHITECTURE_MIPS
PROCESSOR_ARCHITECTURE_ALPHA
PROCESSOR_ARCHITECTURE_PPC
```

- Поле `wReserved` пока не используется.
- Поле `dwPageSize` содержит размер страницы в килобайтах и определяет степень разбиения при защите и фиксации страниц. Например, на компьютерах Intel x86 это значение равно 4 Кбайт.
- В поле `lpMinimumApplicationAddress` хранится самый младший адрес памяти, доступный для приложений и DLL. Попытка получить доступ к адресу памяти ниже этого значения приведет, вероятнее всего, к нарушению прав доступа. Поле `lpMaximumApplicationAddress` содержит самый старший адрес памяти, доступный для приложений и DLL. Попытка получить доступ к адресу памяти выше этого значения приведет, скорее всего, к нарушению прав доступа.
- Поле `dwActiveProcessorMask` возвращает маску, представляющую набор процессоров, сконфигурированных в системе. Разряд 0 представляет первый процессор, а разряд 31 — 32-й. Было бы совсем неплохо иметь 32 процессора, правда? Но поскольку Windows 95/98 поддерживает только один процессор, то в данной реализации Win32 будет установлен лишь разряд 0.
- Поле `dwNumberOfProcessors` также возвращает количество процессоров в системе. Трудно сказать, зачем Microsoft понадобилось вносить эти два поля в запись `TSystemInfo`, но, как известно, "жираф большой, ему видней".
- Поле `dwProcessorType` больше неактуально. Оно оставлено для обратной совместимости. Это поле может иметь одно из следующих значений:

```
PROCESSOR_INTEL_386
PROCESSOR_INTEL_486
PROCESSOR_INTEL_PENTIUM
PROCESSOR_MIPS_R4000
PROCESSOR_ALPHA_21064
```

Конечно же, под управлением Windows 95 возможно только значение PROCESSOR_INTEL_x, в то время как под управлением Windows NT допустимы все значения.

- Поле dwAllocationGranularity возвращает степень разбиения, которая будет учитываться при распределении памяти. В предыдущих реализациях Win32 это значение было жестко закодировано в виде 64 Кбайт. Но вполне возможно, что другие типы архитектуры аппаратных средств могут потребовать других значений.
- Поле wProcessorLevel определяет уровень процессора, зависящий от архитектуры системы. Это поле может содержать различные значения для разных процессоров. Для процессоров ряда Intel этот параметр может принимать любое из перечисленных ниже значений.

Значение	*Описание*
3	Процессор 80386
4	Процессор 80486
5	Процессор Pentium

- Поле wProcessorRevision определяет модификацию процессора, зависящую от архитектуры системы. Подобно полю wProcessorLevel, оно может содержать различные значения для разных процессоров. Для архитектур Intel в этом поле будет записано число в формате *xxyy*. Для процессоров 386 и 486 ряда Intel *xx* + $0A означает уровень модификации, а *yy* — саму модификацию (например, 0300 означает микросхему D0). Для микросхем Intel Pentium или Cyrix/NextGen 486 *xx* означает номер модели, а *yy* — модификацию (например, 0201 означает модель 2, модификацию 1).

Следующая процедура используется для получения и добавления форматированных строк системной информации в элемент управления InfoLB (обратите внимание, что этот код нацелен на отображение информации только об архитектуре Intel):

```
procedure TInfoForm.GetSysInfo;
var
  SI: TSystemInfo;
begin
  GetSystemInfo(SI);
  with InfoLB.Items, SI do
  begin
    Add(Format(SProc, [SPIntel]));
    Add(Format(SPageSize, [dwPageSize]));
    Add(Format(SMinAddr, [lpMinimumApplicationAddress]));
    Add(Format(SMaxAddr, [lpMaximumApplicationAddress]));
    Add(Format(SNumProcs, [dwNumberOfProcessors]));
    Add(Format(SAllocGra, [dwAllocationGranularity]));
    case wProcessorLevel of
      3: Add(Format(SProcLevl, [SIntel3]));
      4: Add(Format(SProcLevl, [SIntel4]));
      5: Add(Format(SProcLevl, [SIntel5]));
      6: Add(Format(SProcLevl, [SIntel6]));
    else Add(Format(SProcLevl, [IntToStr(wProcessorLevel)]));
    end;
  end;
end;
```

На заметку Функция GetSystemInfo() эффективно заменяет функцию GetWinFlags() из Windows 3.x API.

На рис. 14.3 показана форма InfoForm, отображающая системную информацию, включая информацию о версии и каталогах.

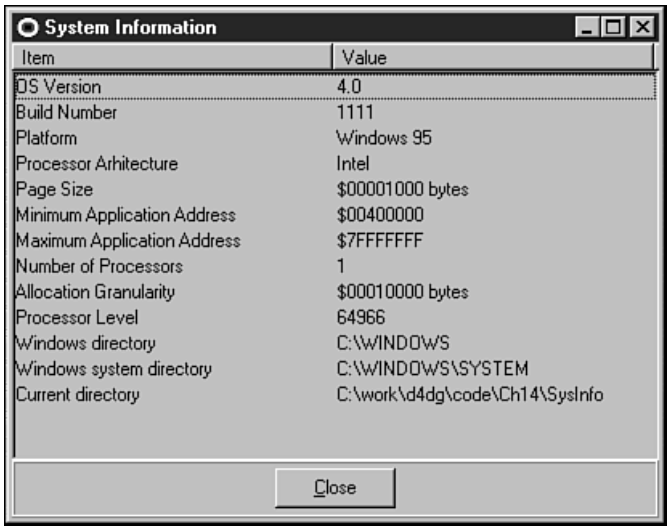

Рис. 14.3. Отображение системной информации

Получение информации о среде

Благодаря функции API `GetEnvironmentStrings()` получение списка переменных окружения — установок (`set`), пути и формы приглашения — стало для текущего процесса довольно простой задачей. Эта функция не имеет параметров и возвращает список строк окружения, разделенных нулями. Формат этого списка таков: строка, за которой следует нуль, за ним снова строка, снова нуль и так до тех пор, пока последняя строка не завершится двойным нулем (`#0#0`). В приложении `SysInfo` используется следующая функция, которая считывает результат работы функции `GetEnvironmentStrings()` и размещает его в элементе управления `InfoLB`:

```
procedure TInfoForm.ShowEnvironment;
var
  EnvPtr, SavePtr: PChar;
begin
  InfoLB.DelimChar := '=';
  EnvPtr := GetEnvironmentStrings;
  SavePtr := EnvPtr;
  InfoLB.Items.Clear;
  repeat
    InfoLB.Items.Add(StrPas(EnvPtr));
    inc(EnvPtr, StrLen(EnvPtr) + 1);
  until EnvPtr^ = #0;
  FreeEnvironmentStrings(SavePtr);
  InfoLB.Sections[0].Text := 'Environment Variable'; //  Переменная окружения
  InfoLB.Sections[1].Text := 'Value'; // Значение
  Caption:= 'Current Environment'; //  Текущее окружение
end;
```

На заметку

Метод `ShowEnvironment()` использует способность Object Pascal выполнять арифметические операции с указателями на строках типа `PChar`. Обратите внимание, что для обработки целого списка строк окружения требуется всего несколько строк программного кода.

Несколько комментариев к последнему методу. Во-первых, обратите внимание на то, что свойство `DelimChar` элемента управления `InfoLB` вначале устанавливается равным `'='`. А поскольку каждая пара, состоящая из переменной окружения и ее значения, уже разделена этим символом, то очень легко отобразить ее в элементе управления `InfoLB`. Кроме того, завершив обработку строк окружения, вы должны вызвать функцию `FreeEnvironmentStrings()`, чтобы освободить выделенный блок памяти.

Совет

С помощью функции `GetEnvironmentStrings()` нельзя получить или установить отдельные переменные окружения. Для этого нужно использовать функции `GetEnvironmentVariable()` и `SetEnvironmentVariable()` (см. справку Win32 API).

На рис. 14.4 показаны строки, определяющие состояние среды в том виде, как они отображаются на форме `InfoForm`.

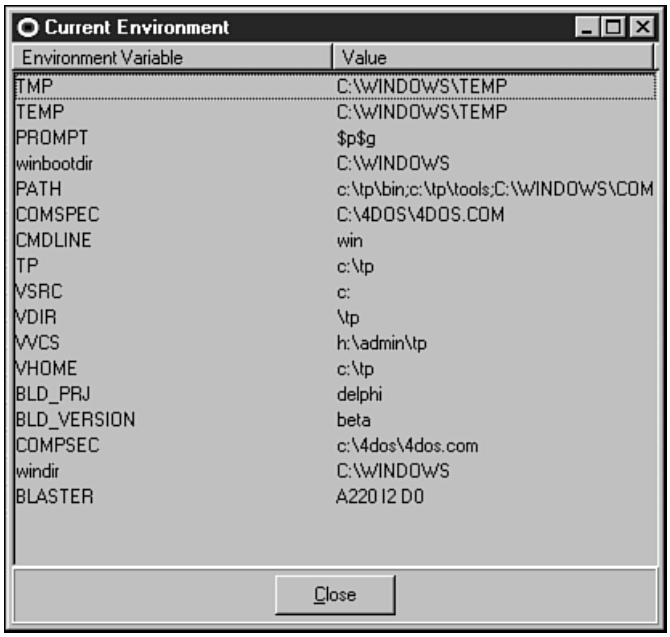

Рис. 14.4. Отображение строк системного окружения

В листинге 14.1 полностью представлен исходный код модуля `InfoU.pas`.

Листинг 14.1. Исходный код модуля InfoU.pas

```
unit InfoU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, HeadList, StdCtrls, ExtCtrls, SysMain;

type
  TInfoVariety = (ivMemory, ivSystem, ivEnvironment);

  TInfoForm = class(TForm)
    InfoLB: THeaderListbox;
    Panel1: TPanel;
    OkBtn: TButton;
  private
    procedure GetOSVerInfo;
    procedure GetSysInfo;
    procedure GetDirInfo;
  public
    procedure ShowMemStatus;
    procedure ShowSysInfo;
    procedure ShowEnvironment;
  end;

procedure ShowInformation(Variety: TInfoVariety);

implementation

{ $R *.DFM}

procedure ShowInformation(Variety: TInfoVariety);
begin
  with TInfoForm.Create(Application) do
    try
      Font := MainForm.Font;
      case Variety of
        ivMemory: ShowMemStatus;
        ivSystem: ShowSysInfo;
```

```
        ivEnvironment: ShowEnvironment;
      end;
      ShowModal;
    finally
      Free;
    end;
end;
const
  {  Строки состояния памяти }
  SMemUse   = 'Memory in useq%d%%';
  STotMem   = 'Total physical memoryq$%.8x bytes';
  SFreeMem  = 'Free physical memoryq$%.8x bytes';
  STotPage  = 'Total page file memoryq$%.8x bytes';
  SFreePage = 'Free page file memoryq$%.8x bytes';
  STotVirt  = 'Total virtual memoryq$%.8x bytes';
  SFreeVirt = 'Free virtual memoryq$%.8x bytes';

  {  Строки информации о версии ОС }
  SOSVer    = 'OS Versionq%d.%d';
  SBuildNo  = 'Build Numberq%d';
  SOSPlat   = 'Platformq%s';
  SOSWin32s = 'Windows 3.1x running Win32s';
  SOSWin95  = 'Windows 95';
  SOSWinNT  = 'Windows NT';

  {  Строки информации о системе }
  SProc     = 'Processor Arhitectureq%s';
  SPIntel   = 'Intel';
  SPageSize = 'Page Sizeq$%.8x bytes';
  SMinAddr  = 'Minimum Application Addressq$%p';
  SMaxAddr  = 'Maximum Application Addressq$%p';
  SNumProcs = 'Number of Processorsq%d';
  SAllocGra = 'Allocation Granularityq$%.8x bytes';
  SProcLevl = 'Processor Levelq%s';
  SIntel3   = '80386';
  SIntel4   = '80486';
  SIntel5   = 'Pentium';
  SIntel6   = 'Pentium Pro';
  SProcRev  = 'Processor Revisionq%.4x';

  {  Строки информации о каталогах }
  SWinDir   = 'Windows directoryq%s';
  SSysDir   = 'Windows system directoryq%s';
  SCurDir   = 'Current directoryq%s';

procedure TInfoForm.ShowMemStatus;
var
  MS: TMemoryStatus;
begin
  InfoLB.DelimChar := 'q';
  MS.dwLength := SizeOf(MS);
  GlobalMemoryStatus(MS);
  with InfoLB.Items, MS do
  begin
    Clear;
    Add(Format(SMemUse, [dwMemoryLoad]));
    Add(Format(STotMem, [dwTotalPhys]));
    Add(Format(SFreeMem, [dwAvailPhys]));
    Add(Format(STotPage, [dwTotalPageFile]));
    Add(Format(SFreePage, [dwAvailPageFile]));
    Add(Format(STotVirt, [dwTotalVirtual]));
    Add(Format(SFreeVirt, [dwAvailVirtual]));
  end;
  InfoLB.Sections[0].Text := 'Resource';
  InfoLB.Sections[1].Text := 'Amount';
  Caption:= 'Memory Status';
end;

procedure TInfoForm.GetOSVerInfo;
var
  VI: TOSVersionInfo;
begin
```

```
  VI.dwOSVersionInfoSize := SizeOf(VI);
  GetVersionEx(VI);
  with InfoLB.Items, VI do
  begin
    Clear;
    Add(Format(SOSVer, [dwMajorVersion, dwMinorVersion]));
    Add(Format(SBuildNo, [LoWord(dwBuildNumber)]));
    case dwPlatformID of
      VER_PLATFORM_WIN32S: Add(Format(SOSPlat, [SOSWin32s]));
      VER_PLATFORM_WIN32_WINDOWS: Add(Format(SOSPlat, [SOSWin95]));
      VER_PLATFORM_WIN32_NT: Add(Format(SOSPlat, [SOSWinNT]));
    end;
  end;
end;

procedure TInfoForm.GetSysInfo;
var
  SI: TSystemInfo;
begin
  GetSystemInfo(SI);
  with InfoLB.Items, SI do
  begin
    Add(Format(SProc, [SPIntel]));
    Add(Format(SPageSize, [dwPageSize]));
    Add(Format(SMinAddr, [lpMinimumApplicationAddress]));
    Add(Format(SMaxAddr, [lpMaximumApplicationAddress]));
    Add(Format(SNumProcs, [dwNumberOfProcessors]));
    Add(Format(SAllocGra, [dwAllocationGranularity]));
    case wProcessorLevel of
      3: Add(Format(SProcLevl, [SIntel3]));
      4: Add(Format(SProcLevl, [SIntel4]));
      5: Add(Format(SProcLevl, [SIntel5]));
      6: Add(Format(SProcLevl, [SIntel6]));
    else Add(Format(SProcLevl, [IntToStr(wProcessorLevel)]));
    end;
  end;
end;

procedure TInfoForm.GetDirInfo;
var
  S: array[0..MAX_PATH] of char;
begin
  {  Получение каталога Windows }
  GetWindowsDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SWinDir, [S]));
  {  Получение системного каталога Windows }
  GetSystemDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SSysDir, [S]));
  {  Получение текущего каталога процесса }
  GetCurrentDirectory(SizeOf(S), S);
  InfoLB.Items.Add(Format(SCurDir, [S]));
end;

procedure TInfoForm.ShowSysInfo;
begin
  InfoLB.DelimChar := 'q';
  GetOSVerInfo;
  GetSysInfo;
  GetDirInfo;
  InfoLB.Sections[0].Text := 'Item';
  InfoLB.Sections[1].Text := 'Value';
  Caption:= 'System Information';
end;

procedure TInfoForm.ShowEnvironment;
var
  EnvPtr, SavePtr: PChar;
begin
  InfoLB.DelimChar := '=';
  EnvPtr := GetEnvironmentStrings;
  SavePtr := EnvPtr;
  InfoLB.Items.Clear;
```

```
  repeat
    InfoLB.Items.Add(StrPas(EnvPtr));
    inc(EnvPtr, StrLen(EnvPtr) + 1);
  until EnvPtr^ = #0;
  FreeEnvironmentStrings(SavePtr);
  InfoLB.Sections[0].Text := 'Environment Variable';
  InfoLB.Sections[1].Text := 'Value';
  Caption:= 'Current Environment';
end;

end.
```

Обеспечение независимости от платформы

Приложение `SysInfo` разработано для функционирования под управлением как Windows 95/98, так и Windows NT, невзирая на то что разные версии Win32 имеют различные способы доступа к низкоуровневой информации (о процессорах и памяти). Подход, который взят нами на вооружение для обеспечения независимости от платформы, состоит в определении интерфейса, содержащего методы получения системной информации. Затем этот интерфейс реализуется для двух различных операционных систем. Он называется `IWin32Info` и имеет довольно простую организацию, которая выглядит следующим образом:

```
type
  IWin32Info = interface
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;
```

- Процедура `FillProcessInfoList()` предназначена для заполнения элементов управления `TListView` и `TImageList` списком выполняющихся процессов и связанных с ними пиктограмм, если таковые имеются.
- Процедура `ShowProcessProperties()` вызывается для получения дополнительной информации об отдельном процессе, выбранном в списке (элементе управления `TListView`).

В проекте `SysInfo` есть модуль `W95Info`, содержащий класс `TWin95Info`, который реализует интерфейс `IWin32Info` для Windows 95 с помощью ToolHelp32 API. И точно так же в этом проекте содержится модуль `WNTInfo` с классом `TWinNTInfo`, который использует преимущества PSAPI для реализации интерфейса `IWin32Info`. В следующем фрагменте кода (`SysMain`), который был взят из главного модуля проекта, показано создание соответствующего класса в зависимости от операционной системы:

```
if Win32Platform = VER_PLATFORM_WIN32_WINDOWS then
    FWinInfo := TWin95Info.Create
else if Win32Platform = VER_PLATFORM_WIN32_NT then
    FWinInfo := TWinNTInfo.Create
else
    raise Exception.Create('This application must be run on Win32');
    { Это приложение должно запускаться под управлением Win32 }
```

Windows 95/98: использование ToolHelp32

ToolHelp32 — это семейство функций и процедур, составляющих подмножество Win32 API, которое позволяет быть в курсе некоторых низкоуровневых операций ОС. В частности, речь идет о функциях, с помощью которых можно получить информацию обо всех процессах, выполняющихся в системе в данный момент, а также потоках, модулях и кучах, принадлежащих каждому процессу. Нетрудно догадаться, что большинство данных, получаемых из ToolHelp32, используется главным образом приложениями, которые должны заглядывать “внутрь” ОС (например, отладчики), хотя необходимо признать, что с помощью этих функций даже средний разработчик получит более точное представление о работе Win32.

На заметку

Семейство процедур и функций ToolHelp32 API доступно только в варианте реализации Win32 для Windows 95/98. Их работа приведет к нарушению системы защиты и безопасности NT-процессов. Поэтому приложения, которые используют функции ToolHelp32, работоспособны только под управлением Windows 95, но не Windows NT.

Мы используем название *ToolHelp32*, чтобы отличить эту подсистему от 16-разрядной версии ToolHelp, которая была включена в Windows 3.1*x*. Большинство функций предыдущей версии не применимы к Win32 и, следовательно, больше не поддерживаются. Кроме того, под управлением Windows 3.1*x* функции ToolHelp были физически расположены в DLL, именуемой `TOOLHELP.DLL`, в то время как функции ToolHelp32 расположены в ядре Win32.

Типы и определения функций ToolHelp32 размещаются в модуле `TlHelp32`, поэтому при работе с этими функциями не забудьте включить его имя в список инструкции `uses`. Для закрепления полученных знаний в приложение, создание которого описано в этой главе, включены все функции, определенные в модуле `TlHelp32`.

На рис. 14.5 показана главная форма проекта `SysInfo`. Пользовательский интерфейс состоит главным образом из пользовательского элемента управления `TheaderListbox` (см. главу 11, "Создание многопоточных приложений"). Дважды щелкнув на любом процессе в списке, вы получите о нем более подробную информацию, которая отображается в дочерней форме, похожей на главную.

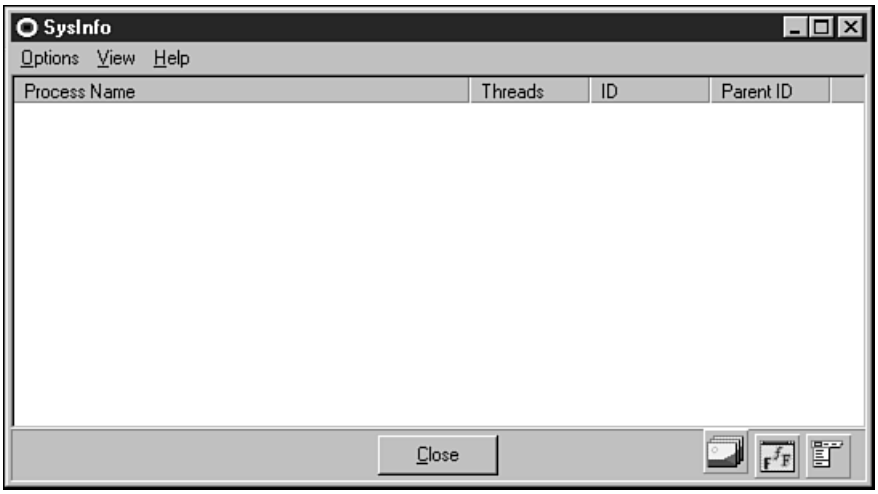

Рис. 14.5. Главная форма `TMainForm` проекта `SysInfo`

Моментальные снимки

Благодаря многозадачной природе среды Win32 такие объекты, как процессы, потоки, модули и т.п., постоянно создаются, разрушаются и модифицируются. И поскольку состояние компьютера непрерывно изменяется, системная информация, которая, возможно, будет иметь значение в данный момент, через секунду уже никого не заинтересует. Например, предположим, что вы хотите написать программу для регистрации всех модулей, загруженных в системе. Поскольку операционная система в любое время может прервать выполнение потока, отрабатывающего вашу программу, чтобы предоставить какие-то кванты времени другому потоку в системе, модули теоретически могут создаваться и разрушаться даже в момент съема информации о них.

В этой динамической среде имело бы смысл на мгновение заморозить систему, чтобы получить такую системную информацию. В ToolHelp32 не предусмотрено средств по замораживанию системы, но есть функция, с помощью которой можно сделать "снимок" системы в заданный момент времени. Эта функция называется `CreateToolhelp32Snapshot()`, и ее объявление выглядит следующим образом:

```
function CreateToolhelp32Snapshot(dwFlags, th32ProcessID: DWORD): THandle; stdcall;
```

- Параметр `dwFlags` означает тип информации, подлежащий включению в моментальный снимок. Этот параметр может иметь одно из перечисленных ниже значений.

Значение	*Описание*
TH32CS_INHERIT	Означает, что дескриптор снимка будет наследуемым
TH32CS_SNAPALL	Эквивалентно заданию значений TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS и TH32CS_SNAPTHREAD
TH32CS_SNAPHEAPLIST	Включает в снимок список куч заданного процесса Win32
TH32CS_SNAPMODULE	Включает в снимок список модулей заданного процесса Win32
TH32CS_SNAPPROCESS	Включает в снимок список процессов Win32
TH32CS_SNAPTHREAD	Включает в снимок список потоков Win32

- Параметр th32ProcessID идентифицирует процесс, для которого вы хотите получить информацию. Для индикации текущего процесса передайте в качестве этого параметра нулевое значение. Данный параметр влияет только на список модулей и куч, поскольку они определяются конкретным процессом. При этом списки процессов и потоков, предоставляемые ToolHelp32, являются общесистемными.
- Функция CreateToolhelp32Snapshot() возвращает дескриптор снимка или -1 в случае ошибки. Возвращаемый дескриптор работает подобно другим дескрипторам Win32 относительно процессов и потоков, для которых он действителен.

Следующий код создает дескриптор снимка, который содержит информацию обо всех процессах, загруженных в настоящий момент (EToolHelpError — это исключительная ситуация, определенная программистом):

```
var
  Snap: THandle;
begin
  Snap := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if Snap = -1 then
    raise EToolHelpError.Create('CreateToolHelp32Snapshot failed');
end;
```

На заметку

По завершении работы с дескриптором используйте функцию CloseHandle() для освобождения ресурсов, связанных с дескриптором, который был создан функцией CreateToolhelp32Snapshot().

Прогулка по процессам

Имея дескриптор снимка, содержащий информацию о процессах, можно воспользоваться двумя функциями ToolHelp32, которые позволяют совершить “прогулку” по процессам в системе. Функции Process32First() и Process32Next() определены следующим образом:

```
function Process32First(hSnapshot: THandle; var lppe: TProcessEntry32): BOOL; stdcall;
function Process32Next(hSnapshot: THandle; var lppe: TProcessEntry32): BOOL; stdcall;
```

Первый параметр у обеих функций является дескриптором снимка, возвращаемым функцией CreateToolhelp32Snapshot().

Второй параметр, lppe, представляет собой запись TProcessEntry32, которая передается по ссылке. По мере прохождения по элементам перечисления функции будут заполнять эту запись информацией о следующем процессе. Запись TProcessEntry32 определяется так:

```
type
  TProcessEntry32 = record
    dwSize: DWORD;
    cntUsage: DWORD;
    th32ProcessID: DWORD;
    th32DefaultHeapID: DWORD;
```

```
    th32ModuleID: DWORD;
    cntThreads: DWORD;
    th32ParentProcessID: DWORD;
    pcPriClassBase: Longint;
    dwFlags: DWORD;
    szExeFile: array[0..MAX_PATH - 1] of Char;
  end;
```

- Поле `dwSize` содержит размер записи `TProcessEntry32`. До использования этой записи поле `dwSize` должно быть инициализировано значением `SizeOf(TProcessEntry32)`.
- В поле `cntUsage` хранится значение счетчика ссылок процесса. Когда это значение станет равным нулю, операционная система выгрузит процесс.
- Поле `th32ProcessID` содержит идентификационный номер процесса.
- Поле `th32DefaultHeapID` предназначено для хранения идентификатора (ID) для кучи процесса, действующей по умолчанию. Этот ID имеет значение только для функций ToolHelp32, и его нельзя использовать с другими функциями Win32.
- Поле `thModuleID` идентифицирует модуль, связанный с процессом. Это поле имеет значение только для функций ToolHelp32.
- По значению поля `cntThreads` можно судить о том, сколько потоков начало выполняться в данном процессе.
- Поле `th32ParentProcessID` идентифицирует родительский процесс для данного процесса.
- В поле `pcPriClassBase` хранится базовый приоритет процесса. Операционная система использует это значение для управления работой потоков.
- Поле `dwFlags` зарезервировано.
- Поле `szExeFile` содержит строку с ограничивающим нуль-символом, которая представляет собой путь и имя файла EXE-программы или драйвера, связанного с данным процессом.

После создания снимка, содержащего информацию о процессах, для опроса данных по каждому процессу достаточно вызвать сначала функцию `Process32First()`, а затем вызывать функцию `Process32Next()` до тех пор, пока она не возвратит значение `False`.

Код опроса процессов инкапсулирован в классе `TWin95Info`, который реализует интерфейс `IWin32Info`. В следующем листинге представлен код метода `Refresh()` класса `TWin95Info`, который предназначен для опроса системных процессов и для добавления каждого из них в список:

```
procedure TWin95Info.Refresh;
var
  PE: TProcessEntry32;
  PPE: PProcessEntry32;
begin
  FProcList.Clear;
  if FSnap > 0 then CloseHandle(FSnap);
  FSnap := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if FSnap = -1 then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  PE.dwSize := SizeOf(PE);
  if Process32First(FSnap, PE) then      // Получаем процесс
    repeat
      New(PPE);                          // Создаем новый PPE
      PPE^ := PE;                        // Заполняем его
      FProcList.Add(PPE);                // Добавляем его в список
    until not Process32Next(FSnap, PE);  // Получаем следующий процесс
end;
```

Метод `Refresh()` вызывается методом `FillProcessInfoList()`. Как пояснялось выше, этот метод заполняет элементы управления `TListView` и `TImageList` информацией обо всех выполняемых процессах. В этом вы можете убедиться сами, рассмотрев внимательно следующий листинг:

```
procedure TWin95Info.FillProcessInfoList(ListView: TListView; ImageList: TImageList);
var
  I: Integer;
  ExeFile: string;
  PE: TProcessEntry32;
  HAppIcon: HIcon;
begin
```

```
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  for I := 0 to FProcList.Count - 1 do
  begin
    PE := PProcessEntry32(FProcList.Items[I])^;
    HAppIcon := ExtractIcon(HInstance, PE.szExeFile, 0);
    try
      if HAppIcon = 0 then HAppIcon := FWinIcon;
      ExeFile := PE.szExeFile;
      if ListView.ViewStyle = vsList then
        ExeFile := ExtractFileName(ExeFile);
        { Вставляем новый элемент, устанавливаем значение
          его заголовка, добавляем подэлементы. }
      with ListView.Items.Add, SubItems do
      begin
        Caption := ExeFile;
        Data := FProcList.Items[I];
        Add(IntToStr(PE.cntThreads));
        Add(IntToHex(PE.th32ProcessID, 8));
        Add(IntToHex(PE.th32ParentProcessID, 8));
        if ImageList <> nil then
          ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
      end;
    finally
      if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
    end;
  end;
end;
```

На рис. 14.6 показаны результаты выполнения этого кода, отображающие информацию о процессах, запущенных под управлением Windows 95.

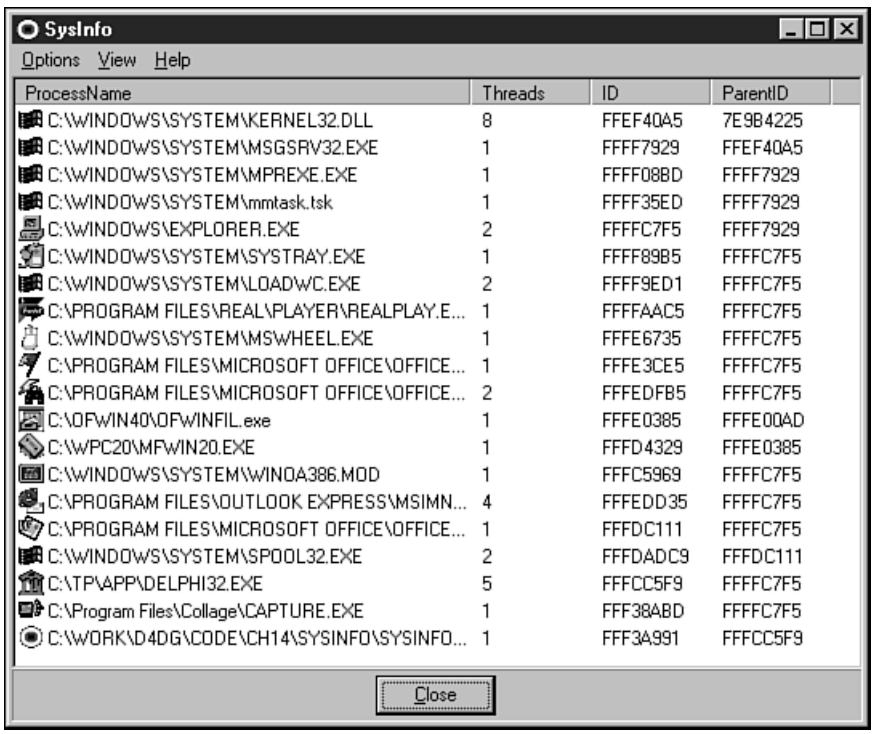

Рис. 14.6. Просмотр процессов под управлением Windows 95

Не оставьте без внимания код, который обеспечивает каждый процесс соответствующим значком (пиктограммой). Отображение значка вместе с именем приложения придаст программе более

профессиональный вид и создаст ощущение “родного” продукта Windows. Функция API ExtractIcon() из модуля ShellAPI делает попытку выделить значок из файла приложения. Если работа функции ExtractIcon() завершится неудачей, для отображения на форме используется стандартная пиктограмма Windows HWinIcon, которая заранее загружается в обработчике события OnCreate для этой формы с помощью функции API LoadImage():

```
FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE, LR_DEFAULTSIZE,
LR_DEFAULTSIZE or LR_DEFAULTCOLOR or LR_SHARED);
```

При двойном щелчке пользователя на одном из процессов в главной форме (см. рис. 14.6) вызывается метод ShowProcessProperties() интерфейса IWin32Info, а код реализации этого метода передает соответствующий параметр методу ShowProcessDetails() из модуля Detail95:

```
procedure TWin95Info.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(PProcessEntry32(Cookie));
end;
```

Чтобы получить снимок информации для выбранного процесса, метод ShowProcessDetails() должен сделать другой снимок с помощью функции CreateToolHelp32Snapshot(). Это реализуется путем передачи параметра Cookie, который идентифицирует процесс (с помощью ID в данном случае), в качестве поля th32ProcessID для функции CreateToolHelp32Snapshot(). Чтобы поместить всю информацию в снимок, в качестве параметра dwFlags передается признак TH32CS_SNAPALL, как показано в следующем фрагменте:

```
{  Создаем снимок для текущего процесса }
FCurSnap := CreateToolhelp32Snapshot(TH32CS_SNAPALL, P^.th32ProcessID);
if FCurSnap = -1 then raise EToolHelpError.Create('CreateToolHelp32Snapshot failed');
```

Объект TDetailForm отображает одновременно только один список. При этом тип списка определяется с помощью переменной перечислимого типа:

```
type
  TListType = (ltThread, ltModule, ltHeap);
```

Кроме того, объект TDetailForm управляет тремя отдельными списками TStringList для потоков, модулей и куч. Эти списки определяются как часть массива DetailLists:

```
DetailLists: array[TListType] of TStringList;
```

Прогулка по потокам

Для составления списка потоков некоторого процесса в ToolHelp32 предусмотрены две функции, которые аналогичны функциям, предназначенным для регистрации процессов: Thread32First() и Thread32Next(). Эти функции объявляются следующим образом:

```
function Thread32First(hSnapshot: THandle; var lpte: TThreadEntry32): BOOL; stdcall;

function Thread32Next(hSnapshot: THandle; var lpte: TThreadENtry32): BOOL; stdcall;
```

Помимо обычного параметра hSnapshot, эти функции также принимают по ссылке параметр типа TThreadEntry32. Как и в случае функций, работающих с процессами, каждая из них заполняет запись TThreadEntry32, объявление которой имеет вид

```
type
  TThreadEntry32 = record
    dwSize: DWORD;
    cntUsage: DWORD;
    th32ThreadID: DWORD;
    th32OwnerProcessID: DWORD;
    tpBasePri: Longint;
    tpDeltaPri: Longint;
    dwFlags: DWORD;
  end;
```

- Поле dwSize определяет размер записи и поэтому должно быть инициализировано значением SizeOf(TThreadEntry32) до использования этой записи.
- Поле cntUsage содержит счетчик ссылок данного потока. При обнулении этого счетчика поток выгружается операционной системой.

- Поле `th32ThreadID` представляет собой идентификационный номер потока, который имеет значение только для функций ToolHelp32.
- Поле `th32OwnerProcessID` содержит идентификатор (ID) процесса, которому принадлежит данный поток. Этот ID можно использовать с другими функциями Win32.
- Поле `tpBasePri` представляет собой базовый класс приоритета потока. Это значение одинаково для всех потоков данного процесса. Возможные значения этого поля обычно лежат в диапазоне от 4 до 24. Ниже приведены описания этих значений.

Значение	*Описание*
4	Ожидающий
8	Нормальный
13	Высокий
24	Реальное время

- Поле `tpDeltaPri` представляет собой *дельта-приоритет* (разницу), определяющий величину отличия от значения `tpBasePri`. Это число со знаком, которое в сочетании с базовым классом приоритета показывает общий приоритет потока. Ниже перечислены константы, определенные для всех возможных значений дельта-приоритета.

Константа	*Значение*
`THREAD_PRIORITY_IDLE`	`-15`
`THREAD_PRIORITY_LOWEST`	`-2`
`THREAD_PRIORITY_BELOW_NORMAL`	`-1`
`THREAD_PRIORITY_NORMAL`	`0`
`THREAD_PRIORITY_ABOVE_NORMAL`	`1`
`THREAD_PRIORITY_HIGHEST`	`2`
`THREAD_PRIORITY_TIME_CRITICAL`	`15`

- Поле `dwFlags` в данный момент зарезервировано и не должно использоваться.

Метод `WalkThreads()` объекта `TDetailForm` используется для составления списка потоков. Это делается путем добавления значимой информации о каждом потоке в "потоковый" элемент массива `DetailLists`. Вот как выглядит код этого метода:

```
procedure TWin95DetailForm.WalkThreads;
{  Для составления списка потоков используются функции ToolHelp32 }
var
  T: TThreadEntry32;
begin
  DetailLists[ltThread].Clear;
  T.dwSize := SizeOf(T);
  if Thread32First(FCurSnap, T) then
    repeat
      {  Обязательно убеждаемся, что исследуемый поток принадлежит текущему процессу }
      if T.th32OwnerProcessID = FCurProc.th32ProcessID then
        DetailLists[ltThread].Add(Format(SThreadStr, [T.th32ThreadID,
                                  GetClassPriorityString(T.tpBasePri),
                                  GetThreadPriorityString(T.tpDeltaPri),
                                  T.cntUsage]));
    until not Thread32Next(FCurSnap, T);
end;
```

На заметку

Следующая строка кода в методе `WalkThreads()` имеет важное значение, поскольку списки потоков, "составляемые" с помощью функций ToolHelp32, не связываются с определенным потоком:

```
if T.th32OwnerProcessID = FCurProc.th32ProcessID then
```

Иными словами, при сканировании потоков нужно обязательно вручную проверять результат приведенного выше сравнения, чтобы выделить потоки, связанные с интересующим вас процессом.

На рис. 14.7 показана вспомогательная форма, на которой отображен список потоков.

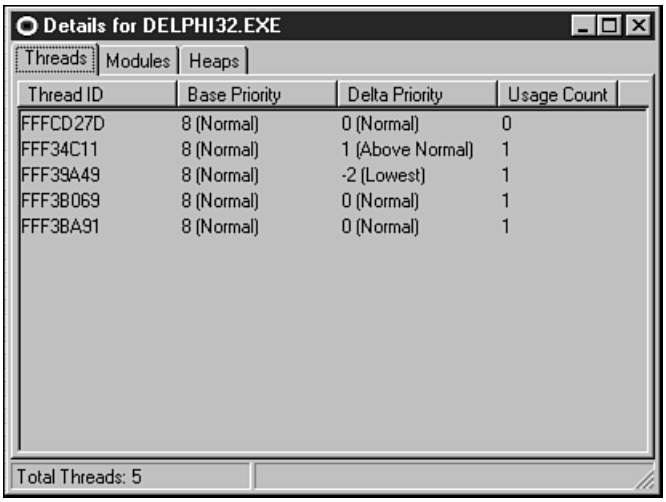

Рис. 14.7. Отображение потоков Windows 95 на вспомогательной форме

Опрос модулей

Опрос модулей выполняется практически так же, как опрос процессов или потоков. Для этого в ToolHelp32 предусмотрены две функции: `Module32First()` и `Module32Next()`, которые определяются следующим образом:

```
function Module32First(hSnapshot: THandle; var lpme: TModuleEntry32): BOOL; stdcall;

function Module32Next(hSnapshot: THandle; var lpme: TModuleEntry32): BOOL; stdcall;
```

Первым параметром в обеих функциях является дескриптор снимка, а вторым (на этот раз `var`-параметром) — запись `TModuleEntry32`. Ее определение имеет следующий вид:

```
type
  TModuleEntry32 = record
    dwSize: DWORD;
    th32ModuleID: DWORD;
    th32ProcessID: DWORD;
    GlblcntUsage: DWORD;
    ProccntUsage: DWORD;
    modBaseAddr: PBYTE;
    modBaseSize: DWORD;
    hModule: HMODULE;
    szModule: array[0..MAX_MODULE_NAME32 + 1] of Char;
    szExePath: array[0..MAX_PATH - 1] of Char;
  end;
```

- Поле `dwSize` определяет размер записи и поэтому должно быть инициализировано значением `SizeOf(TModuleEntry32)` до использования этой записи.
- Поле `th32ModuleID` представляет собой идентификатор модуля, который имеет значение только для функций ToolHelp32.
- Поле `th32ProcessID` содержит идентификатор (ID) опрашиваемого процесса. Этот ID можно использовать с другими функциями Win32.
- Поле `GlblcntUsage` содержит глобальный счетчик ссылок данного модуля.
- Поле `ProccntUsage` содержит счетчик ссылок модуля в контексте процесса-владельца.
- Поле `modBaseAddr` представляет собой базовый адрес модуля в памяти. Это значение действительно только в контексте идентификатора процесса `th32ProcessID`.
- Поле `modBaseSize` определяет размер (в байтах) модуля в памяти.

- Поле `hModule` содержит дескриптор модуля. Это значение действительно только в контексте идентификатора процесса `th32ProcessID`.
- Поле `szModule` содержит строку с именем модуля, завершающуюся нуль-символом.
- Поле `szExepath` предназначено для хранения строки с ограничивающим нуль-символом, содержащей полный путь модуля.

Метод `WalkModules()` объекта `TDetailForm` очень похож на метод `WalkThreads()`. Как показано в следующем коде, этот метод создает список модулей и добавляет его в соответствующую часть списка массива `DetailLists`:

```
procedure TWin95DetailForm.WalkModules;
{  Использует функции Uses ToolHelp32 для создания списка модулей }
var
  M: TModuleEntry32;
begin
  DetailLists[ltModule].Clear;
  M.dwSize := SizeOf(M);
  if Module32First(FCurSnap, M) then
    repeat
      DetailLists[ltModule].Add(Format(SModuleStr, [M.szModule, M.ModBaseAddr,
M.ModBaseSize, M.ProcCntUsage]));
    until not Module32Next(FCurSnap, M);
end;
```

На рис. 14.8 показана вспомогательная форма, на которой отображается список модулей.

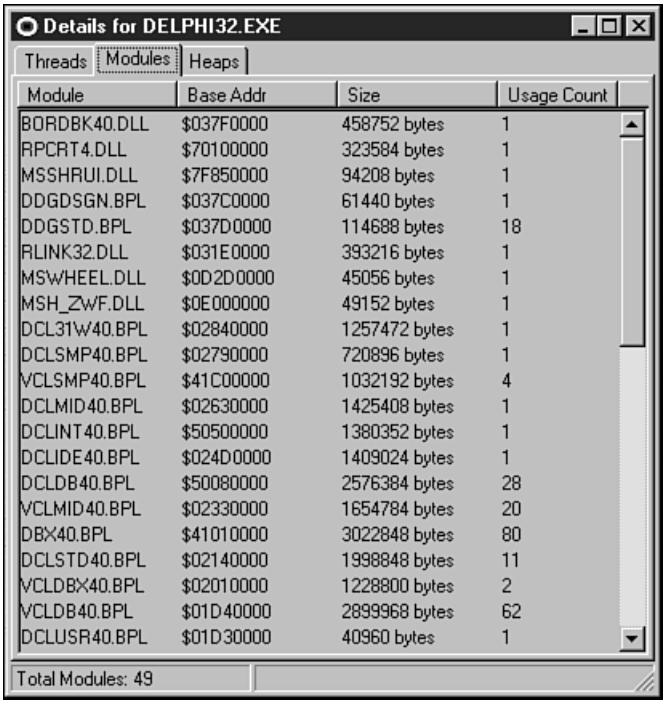

Рис. 14.8. Отображение модулей Windows 95 во вспомогательной форме

Опрос куч

Опрос куч несколько сложнее опроса других типов объектов, с которыми вы познакомились в этой главе. В ToolHelp32 предусмотрены четыре функции, с помощью которых можно получить информацию о кучах. Первые две (`Heap32ListFirst()` и `Heap32ListNext()`) позволяют выполнить проход по всем кучам процесса, а две другие (`Heap32First()` и `Heap32Next()`) используются для получения более подробной информации обо всех блоках внутри отдельной кучи.

Функции `Heap32ListFirst()` и `Heap32ListNext()` определяются следующим образом:

```
function Heap32ListFirst(hSnapshot: THandle; var lphl: THeapList32): BOOL; stdcall;

function Heap32ListNext(hSnapshot: THandle; var lphl: THeapList32): BOOL; stdcall;
```

И вновь первый параметр является дескриптором снимка, а второй (lphl) представляет запись типа THeapList32, передаваемую по ссылке. Определение этой записи имеет следующий вид:

```
type
  THeapList32 = record
    dwSize: DWORD;
    th32ProcessID: DWORD;
    th32HeapID: DWORD;
    dwFlags: DWORD;
  end;
```

- Поле dwSize определяет размер записи и поэтому должно быть инициализировано значением SizeOf(THeapList32) до использования этой записи.
- Поле th32ProcessID содержит идентификатор (ID) процесса-владельца.
- Поле th32HeapID содержит идентификатор (ID) кучи. Этот ID имеет значение только для заданного процесса, и его можно использовать только с функциями ToolHelp32.
- В поле dwFlags хранится признак, который определяет тип кучи. В качестве значения этого поля может использоваться либо константа HF32_DEFAULT (которая означает, что текущая куча является стандартной кучей процесса), либо константа HF32_SHARED (которая означает, что текущая куча является обычно разделяемой кучей).

Функции Heap32First() и Heap32Next() определяются следующим образом:

```
function Heap32First(var lphe: THeapEntry32; th32ProcessID, th32HeapID: DWORD): BOOL;
stdcall;

function Heap32Next(var lphe: THeapEntry32): BOOL; stdcall;
```

Обратите внимание на то, что списки параметров этих функций немного отличаются от соответствующих списков функций, связанных с перечислением процессов, потоков, модулей и куч, с которыми вы познакомились выше в этой главе. Эти функции предназначены для перечисления блоков данной кучи в данном процессе, а не некоторых свойств одного процесса. При вызове функции Heap32First() параметры, установленные в полях th32ProcessID и th32HeapID, должны быть равны значениям одноименных полей записи THeapList32, заполненной с помощью функций Heap32ListFirst() или Heap32ListNext(). Var-параметр lphe функций Heap32First() и Heap32Next() имеет тип THeapEntry32. Эта запись содержит дескриптивную информацию, относящуюся к блоку кучи, и ее определение имеет следующий вид:

```
type
  THeapEntry32 = record
    dwSize: DWORD;
    hHandle: THandle;      // Дескриптор этого блока кучи
    dwAddress: DWORD;      // Линейный адрес начала блока
    dwBlockSize: DWORD;    // Размер блока в байтах
    dwFlags: DWORD;
    dwLockCount: DWORD;
    dwResvd: DWORD;
    th32ProcessID: DWORD; // Процесс-владелец
    th32HeapID: DWORD;     // Идентификатор кучи в нем
  end;
```

- Поле dwSize определяет размер записи и поэтому должно быть инициализировано значением SizeOf(THeapEntry32) до использования этой записи.
- Поле hHandle содержит дескриптор блока кучи.
- Поле dwAddress представляет собой линейный адрес начала блока кучи.
- В поле dwBlockSize содержится размер в байтах этого блока кучи.
- В поле dwFlags хранится признак, который определяет тип блока кучи. Это поле может иметь одно из значений, приведенных в таблице.

Значение	*Описание*
LF32_FIXED	Блок памяти имеет фиксированное местонахождение
LF32_FREE	Блок памяти не используется

LF32_MOVEABLE	Блок памяти можно перемещать

- Поле dwLockCount представляет собой счетчик блокировок блока памяти. Это значение увеличивается на единицу при каждом вызове процессом одной из функций: GlobalLock() или LocalLock().
- Поле dwResvd зарезервировано в данный момент и не должно использоваться.
- Поле th32ProcessID содержит идентификатор процесса-владельца.
- Поле th32HeapID является идентификатором кучи, которой принадлежит блок.

Поскольку до составления списка блоков кучи вам придется сначала составить список куч, то код опроса блоков кучи немного сложнее того, что было продемонстрировано до сих пор. Как видно из приведенного ниже метода TDetailForm.WalkHeaps(), весь фокус состоит во вложении цикла Heap32First()/Heap32Next() внутрь цикла Heap32ListFirst()/Heap32ListNext(). В этом методе вводится дополнительный уровень сложности путем добавления указателя на объекты типа записи PHeapEntry32 в ту часть массива DetailLists, которая относится к списку куч. Причем это выполняется таким образом, чтобы информация о куче была доступна позже, при присмотре содержимого кучи.

```
procedure TWin95DetailForm.WalkHeaps;
{  Использует функции ToolHelp32 для составления списка куч }
var
  HL: THeapList32;
  HE: THeapEntry32;
  PHE: PHeapEntry32;
begin
  DetailLists[ltHeap].Clear;
  HL.dwSize := SizeOf(HL);
  HE.dwSize := SizeOf(HE);
  if Heap32ListFirst(FCurSnap, HL) then
    repeat
      if Heap32First(HE, HL.th32ProcessID, HL.th32HeapID) then
        repeat
          New(PHE);
          PHE^ := HE;
          DetailLists[ltHeap].AddObject(Format(SHeapStr, [HL.th32HeapID,
                                Pointer(HE.dwAddress), HE.dwBlockSize,
                                GetHeapFlagString(HE.dwFlags)]), TObject(PHE));
        until not Heap32Next(HE);
    until not Heap32ListNext(FCurSnap, HL);
  HeapListAlloc := True;
end;
```

На рис. 14.9 показана вспомогательная форма, отображающая список блоков кучи.

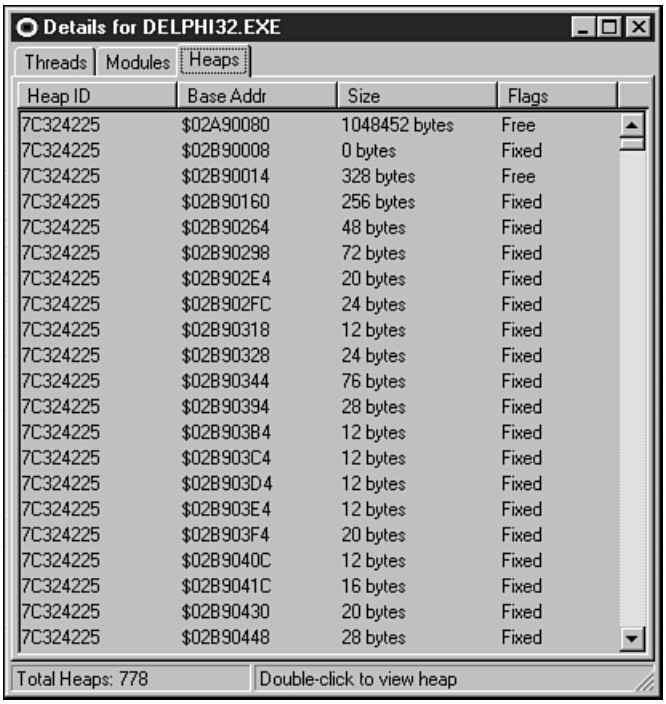

Рис. 14.9. Отображение блоков кучи Windows 95 на вспомогательной форме

Просмотр куч

К этому моменту вы уже узнали обо всех функциях ToolHelp32 API, за исключением одной: `ToolHelp32ReadProcessMemory()`. Желательно, чтобы чтение этой главы вызвало у вас чувство глубокого удовлетворения, а потому познакомьтесь и с этой функцией.

Функция `ToolHelp32ReadProcessMemory()` объявляется следующим образом:

```
function Toolhelp32ReadProcessMemory(th32ProcessID: DWORD; lpBaseAddress: Pointer; var
lpBuffer; cbRead: DWORD; var lpNumberOfBytesRead: DWORD): BOOL; stdcall;
```

Эта функция, возможно, самая могущественная и, бесспорно, самая увлекательная в ToolHelp32, поскольку действительно позволяет заглянуть в пространство памяти другого процесса. Рассмотрим ее параметры.

- Параметр `th32ProcessID` является идентификатором процесса, память которого вы хотите прочитать. Это значение можно получить с помощью одной из функций перечислений ToolHelp32. Для указания текущего процесса можно передать в качестве этого параметра нулевое значение.
- Параметр `lpBaseAddress` представляет собой линейный адрес первого байта памяти, которую вы хотите прочитать в процессе `th32ProcessID`. При этом нужно указывать правильный процесс и правильный адрес, поскольку любой линейный адрес имеет значение только для конкретного процесса.
- Параметр `lpBuffer` — это буфер, в который вы собираетесь скопировать память процесса `th32ProcessID`. Не забудьте позаботиться о выделении памяти для этого буфера.
- Параметр `cbRead` определяет количество байтов для чтения из процесса `th32ProcessID`, начиная с адреса `lpBaseAddress`.
- Параметр `lpNumberOfBytesRead` заполняется во время работы функции перед возвратом из нее. Он определяет количество байтов, действительно прочитанных из процесса `th32ProcessID`.

Поскольку с помощью этой функции память отдельного процесса копируется в локальный буфер, приложение `SysInfo` использует еще одну модальную форму — `HeapViewForm`, которая форматирует дамп памяти для просмотра. Для выполнения этого форматирования форма `HeapViewForm` использует пользовательский компонент `TddgMemView`. Поскольку описание работы этого элемента управления выходит за рамки данной главы (и поскольку он не слишком сложный для понимания), вы можете просмотреть исходный код для этого компонента, обратившись к компакт-диску, прилагаемому к данной книге. А здесь

приводится исходный код метода DetailLBDblClick() формы TDetailForm, который вызывается при двойном щелчке пользователя на любом элементе в DetailLB:

```
procedure TWin95DetailForm.DetailLBDblClick(Sender: TObject);
{  Эта процедура вызывается при двойном щелчке пользователя на любом
   элементе в списке DetailLB.  Если текущей вкладкой является вкладка
   куч, отображается форма просмотра куч. }
var
  NumRead: DWORD;
  HE: THeapEntry32;
  MemSize: integer;
begin
  inherited;
  if DetailTabs.TabIndex = 2 then
  begin
    HE := PHeapEntry32(DetailLB.Items.Objects[DetailLB.ItemIndex])^;
    MemSize := HE.dwBlockSize;          // Получаем размер кучи
    {  Если куча слишком велика, используем ProcMemMaxSize }
    if MemSize > ProcMemMaxSize then MemSize := ProcMemMaxSize;
    ProcMem := AllocMem(MemSize);      // Выделяем временный буфер
    Screen.Cursor := crHourGlass;
    try
      {  Копируем кучу во временный буфер }
      if Toolhelp32ReadProcessMemory(FCurProc.th32ProcessID,
        Pointer(HE.dwAddress), ProcMem^, MemSize, NumRead) then
        {  Указываем на временный буфер для элемента управления HeapView }
        ShowHeapView(ProcMem, MemSize)
      else
        MessageDlg(SHeapReadErr, mtInformation, [mbOk], 0);
    finally
      Screen.Cursor := crDefault;
      FreeMem(ProcMem, MemSize);
    end;
  end;
end;
```

В этом методе сначала проверяется, является ли вкладка списка куч текущей. При положительном результате выделяется временный буфер, который передается для заполнения функции ToolHelp32ReadProcessMemory(). После заполнения буфер отображается в элементе управления TddgMemView, и форма HeapViewForm отображается модально. При возвращении этой формы из вызова ShowModal() буфер освобождается. На рис. 14.10 показан пример просмотра куч.

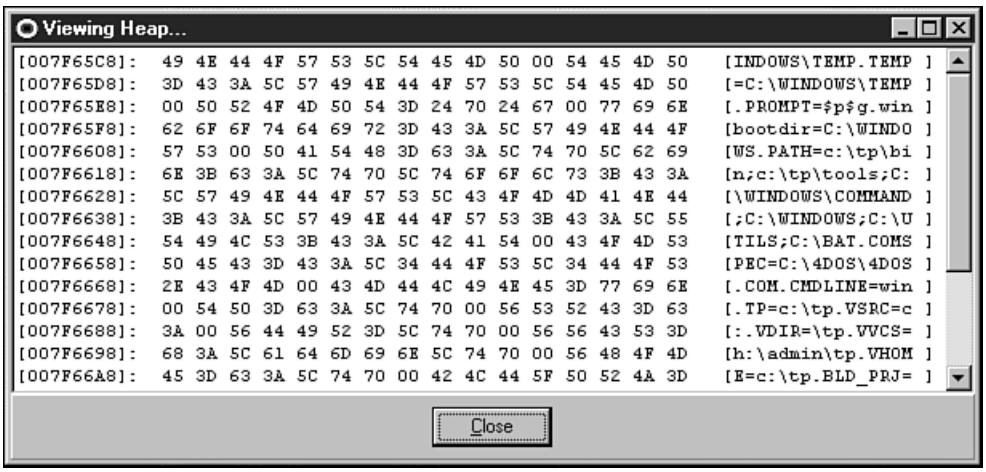

Рис. 14.10. Просмотр куч другого процесса Windows 95

Исходный код

В листингах 14.2 и 14.3 представлен исходный код модулей W95Info.pas и Detail95.pas соответственно.

Листинг 14.2. Модуль W95Info.pas

```
unit W95Info;

interface

uses Windows, InfoInt, Classes, TlHelp32, Controls, ComCtrls;

type
  TWin95Info = class(TInterfacedObject, IWin32Info)
  private
    FProcList: TList;
    FWinIcon: HICON;
    FSnap: THandle;
    procedure Refresh;
public
    constructor Create;
    destructor Destroy; override;
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;

implementation

uses ShellAPI, CommCtrl, SysUtils, Detail95;

const
  ProcessInfoCaptions: array[0..3] of string = ('ProcessName', 'Threads',
                                                'ID', 'ParentID');

{  TProcList }

type
  TProcList = class(TList)
    procedure Clear; override;
  end;

procedure TProcList.Clear;
var
  I: Integer;
begin
  for I := 0 to Count - 1 do Dispose(PProcessEntry32(Items[I]));
  inherited Clear;
end;

{  TWin95Info }

constructor TWin95Info.Create;
begin
  FProcList := TProcList.Create;
  FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
                        LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR
                        or LR_SHARED);
end;

destructor TWin95Info.Destroy;
begin
  DestroyIcon(FWinIcon);
  if FSnap > 0 then CloseHandle(FSnap);
  FProcList.Free;
  inherited Destroy;
end;

procedure TWin95Info.FillProcessInfoList(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  ExeFile: string;
  PE: TProcessEntry32;
  HAppIcon: HIcon;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
```

```
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  for I := 0 to FProcList.Count - 1 do
  begin
    PE := PProcessEntry32(FProcList.Items[I])^;
    HAppIcon := ExtractIcon(HInstance, PE.szExeFile, 0);
    try
      if HAppIcon = 0 then HAppIcon := FWinIcon;
      ExeFile := PE.szExeFile;
      if ListView.ViewStyle = vsList then
        ExeFile := ExtractFileName(ExeFile);
        { Вставляем новый элемент, устанавливаем его заголовок,
          добавляем подэлементы. }
      with ListView.Items.Add, SubItems do
      begin
        Caption := ExeFile;
        Data := FProcList.Items[I];
        Add(IntToStr(PE.cntThreads));
        Add(IntToHex(PE.th32ProcessID, 8));
        Add(IntToHex(PE.th32ParentProcessID, 8));
        if ImageList <> nil then
          ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
      end;
    finally
      if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
    end;
  end;
end;

procedure TWin95Info.Refresh;
var
  PE: TProcessEntry32;
  PPE: PProcessEntry32;
begin
  FProcList.Clear;
  if FSnap > 0 then CloseHandle(FSnap);
  FSnap := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if FSnap = -1 then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  PE.dwSize := SizeOf(PE);
  if Process32First(FSnap, PE) then     // Получаем процесс
    repeat
      New(PPE);                         // Создаем новый PPE
      PPE^ := PE;                       // Заполняем его
      FProcList.Add(PPE);               // Добавляем его в список
    until not Process32Next(FSnap, PE); // Получаем следующий процесс
end;

procedure TWin95Info.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(PProcessEntry32(Cookie));
end;

end.
```

Листинг 14.3. Модуль Detail95.pas

```
unit Detail95;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, HeadList, TlHelp32, Menus, SysMain, DetBase;

type
  TListType = (ltThread, ltModule, ltHeap);
```

```
  TWin95DetailForm = class(TBaseDetailForm)
    procedure DetailTabsChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DetailLBDblClick(Sender: TObject);
  private
    FCurSnap: THandle;
    FCurProc: TProcessEntry32;
    DetailLists: array[TListType] of TStringList;
    ProcMem: PByte;
    HeapListAlloc: Boolean;
    procedure FreeHeapList;
    procedure ShowList(ListType: TListType);
    procedure WalkThreads;
    procedure WalkHeaps;
    procedure WalkModules;
  public
    procedure NewProcess(P: PProcessEntry32);
  end;

procedure ShowProcessDetails(P: PProcessEntry32);

implementation

{ $R *.DFM}

uses ProcMem;

const
  {  Массив строк для заголовка каждого соответствующего списка. }
  HeaderStrs: array[TListType] of TDetailStrings = (
      ('Thread ID', 'Base Priority', 'Delta Priority', 'Usage Count'),
      ('Module', 'Base Addr', 'Size', 'Usage Count'),
      ('Heap ID', 'Base Addr', 'Size', 'Flags'));

  {  Массив строк для нижней части страницы каждого списка. }
  ACountStrs: array[TListType] of string[31] = (
      'Total Threads: %d', 'Total Modules: %d', 'Total Heaps: %d');

  TabStrs: array[TListType] of string[7] = ('Threads', 'Modules', 'Heaps');

  SCaptionStr  = 'Details for %s';       // Заголовок формы
  SThreadStr   = '%x'#1'%s'#1'%s'#1'%d';
    { ID, базовый приоритет, дельта-приоритет, использование }
  SModuleStr   = '%s'#1'$%p'#1'%d bytes'#1'%d';
    { Имя, адрес, размер, использование }
  SHeapStr     = '%x'#1'$%p'#1'%d bytes'#1'%s';
    { ID, адрес, размер, признаки }
  SHeapReadErr = 'This heap is not accessible for read access.';
    { Эта куча недоступна для чтения }

  ProcMemMaxSize = $7FFE;
    { Максимальный размер диапазона просмотра кучи }

procedure ShowProcessDetails(P: PProcessEntry32);
var
  I: TListType;
begin
  with TWin95DetailForm.Create(Application) do
    try
      for I := Low(TabStrs) to High(TabStrs) do
        DetailTabs.Tabs.Add(TabStrs[I]);
      NewProcess(P);
      Font := MainForm.Font;
      ShowModal;
    finally
      Free;
    end;
end;

function GetThreadPriorityString(Priority: DWORD): string;
{  Возвращает строки, описывающие приоритет потока }
```

```
begin
  case Priority of
    THREAD_PRIORITY_IDLE:          Result := '%d (Idle)';
    THREAD_PRIORITY_LOWEST:        Result := '%d (Lowest)';
    THREAD_PRIORITY_BELOW_NORMAL:  Result := '%d (Below Normal)';
    THREAD_PRIORITY_NORMAL:        Result := '%d (Normal)';
    THREAD_PRIORITY_ABOVE_NORMAL:  Result := '%d (Above Normal)';
    THREAD_PRIORITY_HIGHEST:       Result := '%d (Highest)';
    THREAD_PRIORITY_TIME_CRITICAL: Result := '%d (Time critical)';
  else
    Result := '%d (unknown)';
  end;
  Result := Format(Result, [Priority]);
end;

function GetClassPriorityString(Priority: DWORD): String;
{  Возвращает строки, описывающие класс приоритета процесса }
begin
  case Priority of
    4:   Result := '%d (Idle)';      // Ожидающий
    8:   Result := '%d (Normal)';    // Нормальный
    13:  Result := '%d (High)';      // Высокий
    24:  Result := '%d (Real time)'; // Реальное время
  else
    Result := '%d (non-standard)';   // Нестандартный
  end;
  Result := Format(Result, [Priority]);
end;

function GetHeapFlagString(Flag: DWORD): String;
{  Возвращает строку, описывающую признак кучи }
begin
  case Flag of
    LF32_FIXED:    Result := 'Fixed';    // Фиксированная
    LF32_FREE:     Result := 'Free';     // Свободная
    LF32_MOVEABLE: Result := 'Moveable'; // Перемещаемая
  end;
end;

procedure TWin95DetailForm.ShowList(ListType: TListType);
{  Отображает соответствующий список потоков, куч или модулей в DetailLB }
var
  i: Integer;
begin
  Screen.Cursor := crHourGlass;
  try
    with DetailLB do
    begin
      for i := 0 to 3 do
        Sections[i].Text := HeaderStrs[ListType, i];
      Items.Clear;
      Items.Assign(DetailLists[ListType]);
    end;
     DetailSB.Panels[0].Text := Format(ACountStrs[ListType],
                                       [DetailLists[ListType].Count]);
     if ListType = ltHeap then
       DetailSB.Panels[1].Text := 'Double-click to view heap'
        { Для просмотра кучи щелкните дважды }
     else
       DetailSB.Panels[1].Text := '';
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TWin95DetailForm.WalkThreads;
{  Использует функции ToolHelp32 для опроса списка потоков }
var
  T: TThreadEntry32;
begin
  DetailLists[ltThread].Clear;
  T.dwSize := SizeOf(T);
```

```
  if Thread32First(FCurSnap, T) then
    repeat
      {  Убедитесь, что поток принадлежит текущему процессу }
      if T.th32OwnerProcessID = FCurProc.th32ProcessID then
        DetailLists[ltThread].Add(Format(SThreadStr, [T.th32ThreadID,
                                  GetClassPriorityString(T.tpBasePri),
                                  GetThreadPriorityString(T.tpDeltaPri),
                                  T.cntUsage]));
    until not Thread32Next(FCurSnap, T);
end;

procedure TWin95DetailForm.WalkModules;
{  Использует функции ToolHelp32 для опроса списка модулей }
var
  M: TModuleEntry32;
begin
  DetailLists[ltModule].Clear;
  M.dwSize := SizeOf(M);
  if Module32First(FCurSnap, M) then
    repeat
      DetailLists[ltModule].Add(Format(SModuleStr, [M.szModule,
                                M.ModBaseAddr, M.ModBaseSize,
                                M.ProcCntUsage]));
    until not Module32Next(FCurSnap, M);
end;

procedure TWin95DetailForm.WalkHeaps;
{  Использует функции ToolHelp32 для опроса списка куч }
var
  HL: THeapList32;
  HE: THeapEntry32;
  PHE: PHeapEntry32;
begin
  DetailLists[ltHeap].Clear;
  HL.dwSize := SizeOf(HL);
  HE.dwSize := SizeOf(HE);
  if Heap32ListFirst(FCurSnap, HL) then
    repeat
      if Heap32First(HE, HL.th32ProcessID, HL.th32HeapID) then
        repeat
          New(PHE);
          PHE^ := HE;
          DetailLists[ltHeap].AddObject(Format(SHeapStr, [HL.th32HeapID,
                                        Pointer(HE.dwAddress), HE.dwBlockSize,
                                        GetHeapFlagString(HE.dwFlags)]),
                                        TObject(PHE));
        until not Heap32Next(HE);
    until not Heap32ListNext(FCurSnap, HL);
  HeapListAlloc := True;
end;

procedure TWin95DetailForm.FreeHeapList;
{  Поскольку в список добавляются специальные выделения объектов
   PHeapList32, они должны быть освобождены. }
var
  i: integer;
begin
  for i := 0 to DetailLists[ltHeap].Count - 1 do
    Dispose(PHeapEntry32(DetailLists[ltHeap].Objects[i]));
end;

procedure TWin95DetailForm.NewProcess(P: PProcessEntry32);
{  Эта процедура вызывается из главной формы, чтобы отобразить
   вспомогательную форму для отдельного процесса. }
begin
  {  Создаем снимок для текущего процесса }
  FCurSnap := CreateToolhelp32Snapshot(TH32CS_SNAPALL, P^.th32ProcessID);
  if FCurSnap = -1 then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  HeapListAlloc := False;
  Screen.Cursor := crHourGlass;
  try
```

```
    FCurProc := P^;
    {  Включаем имя модуля в заголовок вспомогательной формы }
    Caption := Format(SCaptionStr, [ExtractFileName(FCurProc.szExeFile)]);
    WalkThreads;                // Проходим по спискам ToolHelp32
    WalkModules;
    WalkHeaps;
    DetailTabs.TabIndex := 0;   // 0 = вкладка потоков
    ShowList(ltThread);         // Сначала отображаем вкладку потоков
  finally
    Screen.Cursor := crDefault;
    if HeapListAlloc then FreeHeapList;
    CloseHandle(FCurSnap);      // Закрываем дескриптор снимка
  end;
end;

procedure TWin95DetailForm.DetailTabsChange(Sender: TObject);
{  Обработчик события OnChange для установки вкладки.
   Устанавливает список, соответствующий вкладке. }
begin
  inherited;
  ShowList(TListType(DetailTabs.TabIndex));
end;

procedure TWin95DetailForm.FormCreate(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  {  Создаем списки }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT] := TStringList.Create;
end;

procedure TWin95DetailForm.FormDestroy(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  {  Освобождаем списки }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT].Free;
end;

procedure TWin95DetailForm.DetailLBDblClick(Sender: TObject);
{  Эта процедура вызывается при двойном щелчке пользователя на элементе
   в DetailLB. Если текущей является вкладка куч, отображается форма для
   просмотра куч. }
var
  NumRead: DWORD;
  HE: THeapEntry32;
  MemSize: integer;
begin
  inherited;
  if DetailTabs.TabIndex = 2 then
begin
    HE := PHeapEntry32(DetailLB.Items.Objects[DetailLB.ItemIndex])^;
    MemSize := HE.dwBlockSize;          // Получаем размер кучи
    {  Если куча слишком велика, используем значение ProcMemMaxSize }
    if MemSize > ProcMemMaxSize then MemSize := ProcMemMaxSize;
    ProcMem := AllocMem(MemSize);       // Выделяем временный буфер
    Screen.Cursor := crHourGlass;
    try
      {  Копируем кучу во временный буфер }
      if Toolhelp32ReadProcessMemory(FCurProc.th32ProcessID,
        Pointer(HE.dwAddress), ProcMem^, MemSize, NumRead) then
        ShowHeapView(ProcMem, MemSize)
      else
        MessageDlg(SHeapReadErr, mtInformation, [mbOk], 0);
    finally
      Screen.Cursor := crDefault;
      FreeMem(ProcMem, MemSize);
    end;
```

```
  end;
end;

end.
```

Windows NT: PSAPI

Как упоминалось выше, подмножество функций ToolHelp32 не существует в среде Windows NT. Однако в SDK (Software Development Kit — комплекс инструментальных средств разработки программного обеспечения) Windows NT предусмотрена DLL `PSAPI.DLL`, из которой можно получить ту же информацию, что и с помощью ToolHelp32:

- выполняющиеся процессы;
- модули, загружаемые для каждого процесса;
- загруженные драйверы устройств;
- информацию об использовании памяти процессов;
- файлы, отображенные в память.

Более поздние версии Windows NT включают `PSAPI.DLL`, хотя вы можете и сами распространять этот файл со своими приложениями. В Delphi для этой DLL предусмотрен модуль интерфейса `PSAPI.pas`, который динамически загружает все ее функции. Следовательно, приложения, которые используют этот модуль, будут запускаться на компьютерах в любом случае: с или без `PSAPI.DLL` (но, конечно же, если `PSAPI.DLL` не установлена, то ее функции работать не будут, хотя само приложение будет запускаться).

Первый этап получения информации о процессах с помощью PSAPI состоит в вызове функции `EnumProcesses()`, которая определяется следующим образом:

```
function EnumProcesses(lpidProcess: LPDWORD; cb: DWORD; var cbNeeded: DWORD): BOOL;
```

- Параметр `lpidProcess` — это указатель на массив элементов типа `DWORD`, который в результате работы этой функции будет заполнен значениями ID процессов.
- Параметр `cb` содержит количество элементов типа `DWORD`, передаваемых в массиве с помощью параметра `lpidProcess`.
- По окончании работы функции параметр `cbNeeded` будет содержать количество байтов, скопированных в массив, на который указывает параметр `lpidProcess`. Выражение `cbNeeded div SizeOf(DWORD)` будет определять число элементов, скопированных в массив, т.е. число запущенных процессов.

После вызова этой функции массив, передаваемый с помощью параметра `lpidProcess`, будет содержать несколько значений ID процессов. Нужно отметить, что эти значения сами по себе не представляют особой ценности, но с их помощью (путем передачи ID процесса функции API `OpenProcess()`) можно получить дескриптор процесса, вооружившись которым вы сможете вызывать другие функции PSAPI или даже другие функции Win32 API, требующие дескриптор процесса.

В PSAPI для получения информации о загруженных драйверах устройств имеется функция `EnumDeviceDrivers()`; ее определение выглядит следующим образом:

```
function EnumDeviceDrivers(lpImageBase: PPointer; cb: DWORD; var lpcbNeeded: DWORD):
BOOL;
```

- Параметр `lpImageBase` — это указатель на массив указателей, т.е. элементов типа `Pointer`, который в результате работы этой функции будет заполнен базовыми адресами всех драйверов устройств.
- Параметр `cb` содержит количество элементов типа `Pointer`, передаваемых в массиве с помощью параметра `lpImageBase`.
- По окончании работы функции параметр `lpcbNeeded` будет содержать количество байтов, скопированных в массив, на который указывает параметр `lpImageBase`.

В проекте `SysInfo` модуль `WNTInfo.pas` содержит класс `TWinNTInfo`, который реализует интерфейс `IWin32Info`. Этот класс содержит закрытый метод `Refresh()`, который получает информацию о процессах и драйверах устройств:

```
procedure TWinNTInfo.Refresh;
var
  Count: DWORD;
  BigArray: array[0..$3FFF - 1] of DWORD;
begin
  // Получаем массив значений ID процессов
  if not EnumProcesses(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FProcList, Count div SizeOf(DWORD));
  Move(BigArray, FProcList[0], Count);
  // Получаем массив адресов драйверов устройств
  if not EnumDeviceDrivers(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FDrvList, Count div SizeOf(DWORD));
  Move(BigArray, FDrvList[0], Count);
end;
```

Этот метод сначала передает локальный массив `BigArray` функциям `EnumProcesses()` и `EnumDeviceDrivers()`, а затем перемещает данные из массива `BigArray` в динамические массивы `FProcList` и `FDrvList`. Такая неуклюжая реализация этих функций простительна, поскольку ни `EnumProcesses()`, ни `EnumDeviceDrivers()` не предоставляют средств для определения числа элементов, которые будут возвращены перед выделением памяти для массива. Поэтому мы передаем методам заведомо большой массив (полагаем, что достаточно большой) и копируем результаты в динамический массив соответствующего размера.

Метод `FillProcessInfoList()` класса `TWinNTInfo` вызывает два вспомогательных метода: `FillProcesses()` и `FillDrivers()`, которые предназначены для заполнения содержимого компонента `TListView` на главной форме. Метод `FillProcesses()` представлен в следующем листинге:

```
procedure TWinNTInfo.FillProcesses(ListView: TListView; ImageList: TImageList);
var
  I, Count: Integer;
  ProcHand: THandle;
  ModHand: HMODULE;
  HAppIcon: HICON;
  ModName: array[0..MAX_PATH] of char;
begin
  for I := Low(FProcList) to High(FProcList) do
  begin
    ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
  False, FProcList[I]);
    if ProcHand > 0 then
      try
        EnumProcessModules(Prochand, @ModHand, 1, Count);
        if GetModuleFileNameEx(Prochand, ModHand, ModName, SizeOf(ModName)) > 0 then
        begin
          HAppIcon := ExtractIcon(HInstance, ModName, 0);
          try
            if HAppIcon = 0 then HAppIcon := FWinIcon;
            with ListView.Items.Add, SubItems do
            begin
              Caption := ModName;               // Имя файла
              Data := Pointer(FProcList[I]);    // Сохраняем ID
              Add(SProcName);                   // Процесс
              Add(IntToStr(FProcList[I]));      // ID процесса
              Add('$' + IntToHex(ProcHand, 8)); // Дескриптор процесса
              // Класс приоритета
              Add(GetPriorityClassString(GetPriorityClass(ProcHand)));
              // Значок
              if ImageList <> nil then
                ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
            end;
          finally
            if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
          end;
        end;
      finally
        CloseHandle(ProcHand);
      end;
  end;
end;
```

В этом методе функция OpenProcess() используется для преобразования идентификатора каждого процесса в дескриптор процесса. В качестве первого параметра этому методу можно передать несколько признаков, но для запроса информации с помощью PSAPI лучше всего использовать тандем PROCESS_QUERY_INFORMATION и PROCESS_VM_READ. Имея на руках дескриптор процесса, метод FillProcesses() вызывает функцию EnumProcessModules() для получения имени файла, связанного с процессом. Эта функция определяется следующим образом:

```
function EnumProcessModules(hProcess: THandle; lphModule: LPDWORD; cb: DWORD; var
lpcbNeeded: DWORD): BOOL;
```

Этот метод работает аналогично другим функциям PSAPI: hProcess представляет собой дескриптор процесса, lphModule — это указатель на массив дескрипторов модулей, cb определяет число элементов в массиве, а последний параметр возвращает количество байтов, скопированных в массив, на который указывает параметр lphModule.

Поскольку в данный момент нас интересует только главный модуль этого процесса, мы передаем массив, состоящий только из одного элемента. Первый модуль, возвращаемый функцией EnumProcessModules(), и является главным модулем процесса. А затем вся информация о процессах добавляется в элемент управления TListView аналогично тому, как показано в методах класса TWin95Info.

Подобным образом действует и метод FillDrivers(), за исключением того, что он вызывает функцию GetDeviceDriverFileName(), которая определяется как

```
function GetDeviceDriverFileName(ImageBase: Pointer; lpFileName: PChar; nSize: DWORD):
DWORD;
```

Этот метод в качестве первого параметра принимает базовый адрес драйвера устройства, в качестве второго — указатель на строковый буфер, а в качестве последнего — размер буфера. При успешном выполнении функции параметр lpFileName будет содержать имя файла драйвера устройства. В следующем листинге показан наш вариант использования метода:

```
procedure TWinNTInfo.FillDrivers(ListView: TListView; ImageList: TImageList);
var
  I: Integer;
  DrvName: array[0..MAX_PATH] of char;
begin
  for I := Low(FDrvList) to High(FDrvList) do
    if GetDeviceDriverFileName(FDrvList[I], DrvName, SizeOf(DrvName)) > 0 then
      with ListView.Items.Add do
      begin
        Caption := DrvName;
        SubItems.Add(SDrvName);
        SubItems.Add('$' + IntToHex(Integer(FDrvList[I]), 8));
      end;
end;
```

На рис. 14.11 показан результат работы приложения SysInfo, запущенного на компьютере с установленной Windows NT 4.0.

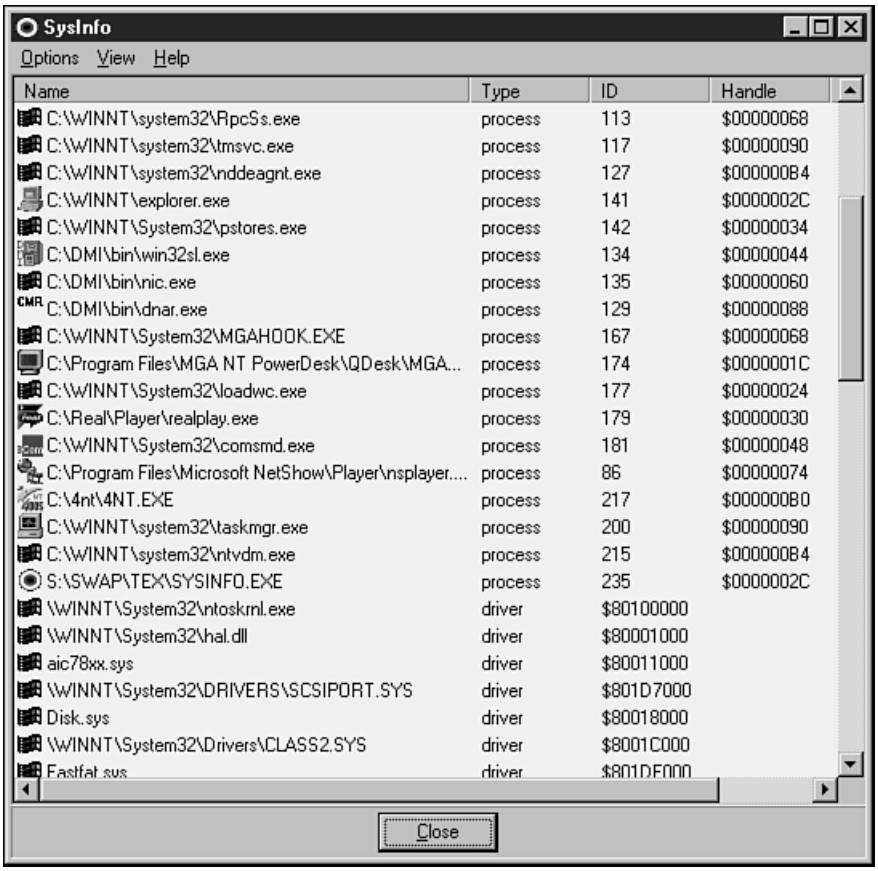

Рис. 14.11. Просмотр процессов и драйверов устройств Windows NT

Подобно тому как в классе `TWin95Info` реализован метод `ShowProcessProperties()`, в классе `TWinNTInfo` выполняется обращение к другому модулю, чтобы отобразить форму, содержащую более подробную информацию о процессах. В частности, дополнительная информация связана с модулями и памятью, используемой процессами. Метод, предназначенный для получения этой информации, принадлежит классу `TWinNTDetailForm`, определенному в модуле `DetailNT`, и включает следующий код:

```
procedure TWinNTDetailForm.NewProcess(ProcessID: DWORD);
const
  AddrMask = DWORD($FFFFF000);
var
  I, Count: Integer;
  ProcHand: THandle;
  WSPtr: Pointer;
  ModHandles: array[0..$3FFF - 1] of DWORD;
  WorkingSet: array[0..$3FFF - 1] of DWORD;
  ModInfo: TModuleInfo;
  ModName, MapFileName: array[0..MAX_PATH] of char;
begin
  ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
                          False, ProcessID);
  if ProcHand = 0 then
    raise Exception.Create('No information available for this process/driver');
  try
    EnumProcessModules(ProcHand, @ModHandles, SizeOf(ModHandles), Count);
    for I := 0 to (Count div SizeOf(DWORD)) - 1 do
      if (GetModuleFileNameEx(ProcHand, ModHandles[I],
        ModName, SizeOf(ModName)) > 0) and
        GetModuleInformation(ProcHand,
        ModHandles[I], @ModInfo, SizeOf(ModInfo)) then
        with ModInfo do
          DetailLists[ltModules].Add(Format(SModuleStr,
              [ModName, lpBaseOfDll, SizeOfImage, EntryPoint]));
    if QueryWorkingSet(ProcHand, @WorkingSet, SizeOf(WorkingSet)) then
      for I := 1 to WorkingSet[0] do
      begin
        WSPtr := Pointer(WorkingSet[I] and AddrMask);
```

```
        GetMappedFileName(ProcHand, WSPtr, MapFileName, SizeOf(MapFileName));
        DetailLists[ltMemory].Add(Format(SMemoryStr,
            [WSPtr, MemoryTypeToString(WorkingSet[I]),
            MapFileName]));
      end;
  finally
    CloseHandle(ProcHand);
  end;
end;
```

Как видите, в этом методе выполняются обращения к функциям `OpenProcess()` и `EnumProcessModules()`, с которыми вы уже знакомы. Этот метод также вызывает функцию PSAPI `QueryWorkingSet()`, но в данном случае это делается для получения информации о процессе. Упомянутая функция определяется следующим образом:

```
function QueryWorkingSet(hProcess: THandle; pv: Pointer; cb: DWORD): BOOL;
```

Параметр `hProcess` является дескриптором процесса, параметр `pv` — это указатель на массив элементов типа `DWORD`, а параметр `cb` содержит число элементов в массиве. По окончании работы этой функции параметр `pv` будет указывать на массив элементов типа `DWORD`, причем старшие 20 разрядов каждого элемента будут содержать базовый адрес страницы памяти, а младшие 12 разрядов — признаки, по которым можно определить, читается ли данная страница, записывается ли, выполняется ли и т.д.

На рис. 14.12 и 14.13 показаны результаты отображения информации о модулях и используемой памяти в среде Windows NT. В листингах 14.4 и 14.5 представлен исходный код модулей `WNTInfo.pas` и `DetailNT.pas` соответственно.

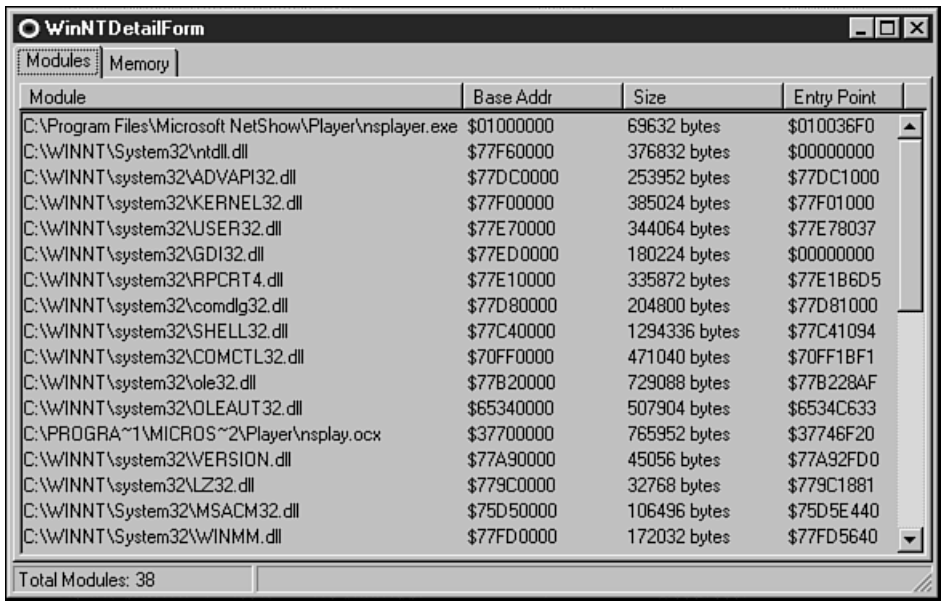

Рис. 14.12. Просмотр модулей процессов в Windows NT

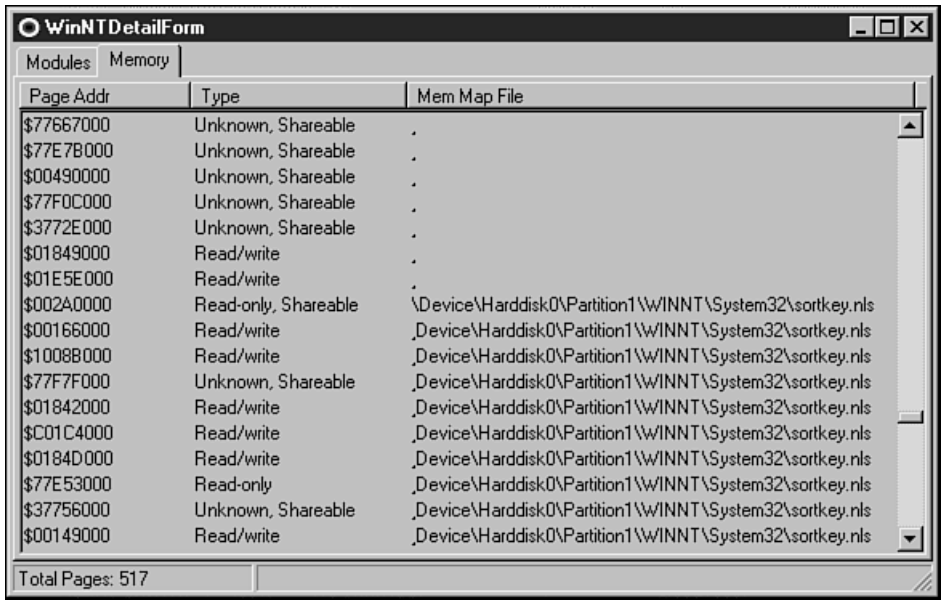

Рис. 14.13. Просмотр данных об использовании памяти процессами в Windows NT

Листинг 14.4. Модуль WNTInfo.pas

```
unit WNTInfo;

interface

uses InfoInt, Windows, Classes, ComCtrls, Controls;

type
  TWinNTInfo = class(TInterfacedObject, IWin32Info)
  private
    FProcList: array of DWORD;
    FDrvlist: array of Pointer;
    FWinIcon: HICON;
    procedure FillProcesses(ListView: TListView; ImageList: TImageList);
    procedure FillDrivers(ListView: TListView; ImageList: TImageList);
    procedure Refresh;
  public
    constructor Create;
    destructor Destroy; override;
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;

implementation

uses SysUtils, PSAPI, ShellAPI, CommCtrl, DetailNT;

const
  SFailMessage = 'Failed to enumerate processes or drivers.  Make sure ' +
                 'PSAPI.DLL is installed on your system.';
    { Не удалось опросить процессы или драйверы. Убедитесь в установке
      библиотеки PSAPI.DLL в вашей системе. }
  SDrvName = 'driver';  // Драйвер
  SProcname = 'process';  // Процесс
  ProcessInfoCaptions: array[0..4] of string = (
    'Name', 'Type', 'ID', 'Handle', 'Priority');
  // Имя, тип, идентификатор, дескриптор, приоритет
function GetPriorityClassString(PriorityClass: Integer): string;
begin
  case PriorityClass of
    HIGH_PRIORITY_CLASS: Result := 'High';          // Высокий
    IDLE_PRIORITY_CLASS: Result := 'Idle';          // Ждущий
    NORMAL_PRIORITY_CLASS: Result := 'Normal';      // Нормальный
    REALTIME_PRIORITY_CLASS: Result := 'Realtime'; // Реального времени
  else
    Result := Format('Unknown ($%x)', [PriorityClass]); // Неизвестный
```

```
  end;
end;

{  TWinNTInfo }

constructor TWinNTInfo.Create;
begin
  FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
                        LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR
                        or LR_SHARED);
end;

destructor TWinNTInfo.Destroy;
begin
  DestroyIcon(FWinIcon);
  inherited Destroy;
end;

procedure TWinNTInfo.FillDrivers(ListView: TListView; ImageList: TImageList);
var
  I: Integer;
  DrvName: array[0..MAX_PATH] of char;
begin
  for I := Low(FDrvList) to High(FDrvList) do
    if GetDeviceDriverFileName(FDrvList[I], DrvName, SizeOf(DrvName)) > 0 then
      with ListView.Items.Add do
      begin
        Caption := DrvName;
        SubItems.Add(SDrvName);
        SubItems.Add('$' + IntToHex(Integer(FDrvList[I]), 8));
      end;
end;

procedure TWinNTInfo.FillProcesses(ListView: TListView;
  ImageList: TImageList);
var
  I, Count: Integer;
  ProcHand: THandle;
  ModHand: HMODULE;
  HAppIcon: HICON;
  ModName: array[0..MAX_PATH] of char;
begin
  for I := Low(FProcList) to High(FProcList) do
  begin
    ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
                            False, FProcList[I]);
    if ProcHand > 0 then
      try
        EnumProcessModules(Prochand, @ModHand, 1, Count);
        if GetModuleFileNameEx(Prochand, ModHand, ModName, SizeOf(ModName)) > 0 then
        begin
          HAppIcon := ExtractIcon(HInstance, ModName, 0);
          try
            if HAppIcon = 0 then HAppIcon := FWinIcon;
            with ListView.Items.Add, SubItems do
            begin
              Caption := ModName;                  // Имя файла
              Data := Pointer(FProcList[I]);       // Сохраняем ID
              Add(SProcName);                      // "Процесс"
              Add(IntToStr(FProcList[I]));         // ID процесса
              Add('$' + IntToHex(ProcHand, 8));    // Дескриптор процесса
              // Класс приоритета
              Add(GetPriorityClassString(GetPriorityClass(ProcHand)));
              // Значок
              if ImageList <> nil then
                ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
            end;
          finally
            if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
          end;
        end;
      finally
```

```
        CloseHandle(ProcHand);
      end;
  end;
end;

procedure TWinNTInfo.FillProcessInfoList(ListView: TListView;
                                         ImageList: TImageList);
var
  I: Integer;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  FillProcesses(ListView, ImageList);
  { Добавляем процессы в список просмотра }
  FillDrivers(ListView, ImageList);
  { Добавляем драйверы устройств в список просмотра }
end;

procedure TWinNTInfo.Refresh;
var
  Count: DWORD;
  BigArray: array[0..$3FFF - 1] of DWORD;
begin
  // Получаем массив идентификаторов процессов
  if not EnumProcesses(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FProcList, Count div SizeOf(DWORD));
  Move(BigArray, FProcList[0], Count);
  // Получаем массив адресов драйверов
  if not EnumDeviceDrivers(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FDrvList, Count div SizeOf(DWORD));
  Move(BigArray, FDrvList[0], Count);
end;

procedure TWinNTInfo.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(DWORD(Cookie));
end;

end.
```

Листинг 14.5. Модуль DetailNT.pas

```
unit DetailNT;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DetBase, ComCtrls, HeadList;

type
  TListType = (ltModules, ltMemory);

  TWinNTDetailForm = class(TBaseDetailForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DetailTabsChange(Sender: TObject);
  private
    FProcHand: THandle;
    DetailLists: array[TListType] of TStringList;
    procedure ShowList(ListType: TListType);
  public
    procedure NewProcess(ProcessID: DWORD);
```

```
  end;

procedure ShowProcessDetails(ProcessID: DWORD);

implementation

uses PSAPI;

{ $R *.DFM}

const
  TabStrs: array[0..1] of string[7] = ('Modules', 'Memory');
        // Модули, память

  {  Массив строк для нижней части каждого списка. }
  ACountStrs: array[TListType] of string[31] =
    ( 'Total Modules: %d', 'Total Pages: %d');
    // Всего модулей, всего страниц

  {  Массив строк для заголовка каждого соответствующего списка. }
  HeaderStrs: array[TListType] of TDetailStrings = (
    ('Module', 'Base Addr', 'Size', 'Entry Point'),
    ('Page Addr', 'Type', 'Mem Map File', ''));
    { Модуль, базовый адрес, размер, точка входа, адрес страницы,
      тип, файл, отображенный в память. }

  SCaptionStr  = 'Details for %s';    // Заголовок формы
  SModuleStr   = '%s'#1'$%p'#1'%d bytes'#1'$%p';
    { Имя, адрес, size, точка входа }
  SMemoryStr   = '$%p'#1'%s'#1'%s';
    { Адрес, тип, файл, отображенный в память }

procedure ShowProcessDetails(ProcessID: DWORD);
var
  I: Integer;
begin
  with TWinNTDetailForm.Create(Application) do
    try
      for I := Low(TabStrs) to High(TabStrs) do
        DetailTabs.Tabs.Add(TabStrs[I]);
      NewProcess(ProcessID);
      ShowList(ltModules);
      ShowModal;
    finally
      Free;
    end;
end;

function MemoryTypeToString(Value: DWORD): string;
const
  TypeMask = DWORD($0000000F);
begin
  Result := '';
  case Value and TypeMask of
    1: Result := 'Read-only';
    2: Result := 'Executable';
    4: Result := 'Read/write';
    5: Result := 'Copy on write';
  else
    Result := 'Unknown';
  end;
  if Value and $100 <> 0 then
    Result := Result + ', Shareable';
end;

procedure TWinNTDetailForm.FormCreate(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  {  Создание списков }
for LT := Low(TListType) to High(TListType) do
```

```
    DetailLists[LT] := TStringList.Create;
end;

procedure TWinNTDetailForm.FormDestroy(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  {  Освобождение списков }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT].Free;
end;

procedure TWinNTDetailForm.NewProcess(ProcessID: DWORD);
const
  AddrMask = DWORD($FFFFF000);
var
  I, Count: Integer;
  ProcHand: THandle;
  WSPtr: Pointer;
  ModHandles: array[0..$3FFF - 1] of DWORD;
  WorkingSet: array[0..$3FFF - 1] of DWORD;
  ModInfo: TModuleInfo;
  ModName, MapFileName: array[0..MAX_PATH] of char;
begin
  ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ, False,
ProcessID);
  if ProcHand = 0 then
    raise Exception.Create('No information available for this process/driver');
    { Для данного процесса/драйвера нет доступной информации }
  try
    EnumProcessModules(ProcHand, @ModHandles, SizeOf(ModHandles), Count);
    for I := 0 to (Count div SizeOf(DWORD)) - 1 do
      if (GetModuleFileNameEx(ProcHand, ModHandles[I], ModName,
        SizeOf(ModName)) > 0) and GetModuleInformation(ProcHand,
        ModHandles[I], @ModInfo, SizeOf(ModInfo)) then
        with ModInfo do
          DetailLists[ltModules].Add(Format(SModuleStr, [ModName, lpBaseOfDll,
                                     SizeOfImage, EntryPoint]));
    if QueryWorkingSet(ProcHand, @WorkingSet, SizeOf(WorkingSet)) then
      for I := 1 to WorkingSet[0] do
      begin
        WSPtr := Pointer(WorkingSet[I] and AddrMask);
        GetMappedFileName(ProcHand, WSPtr, MapFileName, SizeOf(MapFileName));
        DetailLists[ltMemory].Add(Format(SMemoryStr, [WSPtr,
                                  MemoryTypeToString(WorkingSet[I]),
                                  MapFileName]));
      end;
  finally
    CloseHandle(ProcHand);
  end;
end;

procedure TWinNTDetailForm.ShowList(ListType: TListType);
var
  I: Integer;
begin
  Screen.Cursor := crHourGlass;
  try
    with DetailLB do
    begin
      for I := 0 to 3 do
        Sections[I].Text := HeaderStrs[ListType, i];
      Items.Clear;
      Items.Assign(DetailLists[ListType]);
    end;
     DetailSB.Panels[0].Text := Format(ACountStrs[ListType],
       [DetailLists[ListType].Count]);
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

```
procedure TWinNTDetailForm.DetailTabsChange(Sender: TObject);
begin
  inherited;
  ShowList(TListType(DetailTabs.TabIndex));
end;

end.
```

Резюме

В этой главе продемонстрированы методы доступа к системной информации из программ Delphi. Особое внимание уделялось корректному использованию функций ToolHelp32 в среде Windows 95/98 и функций PSAPI в среде Windows NT. Вы узнали, как использовать некоторые функции Win32 API для получения других типов системной информации, включая данные об использовании памяти, переменных окружения и о версии операционной системы. Кроме того, вы научились включать в свои приложения такие пользовательские компоненты, как `TListView`, `TImageList`, `THeaderListbox` и `TddgMemView`. Следующая глава посвящена переходу к Delphi 4 от предыдущих версий.