Лабораторная работа № 9
Управление виртуальной памятью
(2 часа)

Содержание: Архитектура памяти в Windows. Атрибуты защиты. Использование виртуальной памяти. Работа с кучей.

Цель: изучить архитектуру подсистемы управления памятью в Windows и её ограничения; ознакомиться с использованием атрибутов защиты страниц памяти; получить навыки использования API работы с памятью (виртуальная память, MMF, API кучи) в прикладных программах.

Как Вам будет известно из курса операционных систем ^_^, операционная система Windows использует при работе с памятью технологию виртуальной памяти со страничной организацией.

Каждому процессу доступно виртуальное адресное пространство, представляющее собой линейный массив байтов с последовательной нумерацией. Общий объем виртуального адресного пространства зависит от разрядности операционной системы и, например, для 32-битной ОС определяется как 2^32 = 4 Гб. Для удобства управления оно разбито на относительно небольшие блоки равного размера – страницы.

С каждой страницей памяти Windows связывает ряд атрибутов, которые характеризуют её расположение (в физической памяти или нет) и уровень доступа к ней (`PAGE_READONLY`, `PAGE_READWRITE`, `PAGE_NOACCESS`).

С точки зрения процесса, любая из страниц виртуального адресного пространства с которыми он может работать, находится в одном из 3 состояний:

- В свободном (для использования) и не задействованном процессом (free);
- Распределённом процессу для использования (committed);
- Зарезервированном процессом для использования, но пока не используемом (реальная память под такие страницы пока не выделена) (reserved).

Для того чтобы использовать какую-либо часть виртуального адресного пространства памяти, первоначально находящуюся в свободном состоянии, процесс должен вызвать функцию `VirtualAlloc`:

```
LPVOID VirtualAlloc (
     LPVOID lpAddress,          // адрес блока памяти
     SIZE_T   dwSize,           // размер блока памяти
     DWORD flAllocationType,    // тип распределения блока памяти
     DWORD flProtect            // тип защиты блока памяти
);
```

В случае успешного выделения блока памяти функция возвращает его адрес, а в случае ошибки – NULL.

Тип распределения блока памяти, как правило, равен MEM_COMMIT или MEM_RESERVE. Первое значение позволяет распределить и передать память для использования процессу, в то время как второе – только зарезервировать блок памяти для последующего распределения.

Тип защиты устанавливает для страниц блока памяти атрибуты, которые позволяют выполнять с этими страницами только указанные операции:

PAGE_READONLY – только чтение из страниц блока;

PAGE_READWRITE – чтение и запись в страницы блока;

PAGE_EXECUTE – исполнение кода в страницах блока;

PAGE_NOACCESS – запрещает доступ к страницам блока;

PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE – комбинацию соответствующих режимов доступа к страницам.

Попытка применения операции, неразрешенной для страницы виртуальной памяти приведёт к возникновению исключительной ситуации.

После окончания использования блока виртуальной памяти его необходимо освободить, вызвав функцию VirtualFree:

```
BOOL VirtualFree (
     LPVOID lpAddress,     // адрес блока памяти
     SIZE_T   dwSize,      // размер блока памяти
     DWORD dwFreeType      // тип операции освобождения
);
```

Функция возвращает ненулевое значение или FALSE в зависимости от того не было ошибки или была соответственно.

Параметр lpAddress задаёт адрес блока для которого выполняется операция освобождения.

Параметр dwSize задаёт размер в байтах области памяти, распределение которой отменяется, или 0 – если память переводится в состояние свободной (free).

Тип операции освобождения может принимать значения из следующего списка или их комбинацию:

MEM_DECOMMIT – отменяет распределение памяти и переводит её в состояние reserved;

MEM_RELEASE – освобождает виртуальную память (переводит её в состояние free).

В некотором смысле использование операции VirtualAlloc и VirtualFree подобно использованию операторов new и delete из C++. Однако, API работы с виртуальной памятью позволяет более тонко настраивать работу с выделяемыми процессу блоками виртуального адресного пространства. Это касается не только задания атрибутов защиты страниц блока, но и описанных ниже функции. Например, если наличие

какого-либо блока виртуальной памяти в реальной памяти критично для быстродействия Вашего приложения, вы можете воспользоваться функцией VirtualLock:

```
BOOL VirtualLock ( LPVOID lpAddress, SIZE_T dwSize);
```

которая закрепляет указанный блок в реальной памяти и запрещает его вытеснять во внешнюю память.

Для отмены блокировки страниц в физической памяти следует вызвать:

```
BOOL VirtualUnlock ( LPVOID lpAddress, SIZE_T dwSize);
```

Также во время выполнения программы можно многократно изменять атрибуты доступа к нужным страницам виртуальной памяти при помощи функции VirtualProtect:

```
BOOL VirtualProtect (
     LPVOID lpAddress,     // адрес блока памяти
     SIZE_T   dwSize,      // размер блока памяти
     DWORD flNewProtect,   // устанавливаемые флаги доступа
     PDWORD lpflOldProtect // адрес переменной для старых
);
```

Значения первых трёх параметров совпадают со значениями параметров функции VirtualAlloc, а параметр lpflOldProtect задаёт адрес переменной типа DWORD в которую будут сохранены атрибуты доступа, которые были установлены до вызова функции.

API виртуальной памяти также предоставляет функции, которые удобно использовать даже без применения VirtualAlloc и VirtualFree. Например, они могут использоваться для заполнения или копирования структур данных или других составных переменных. К таким функциям можно отнести функцию для заполнения блока памяти определенным значением Fill:

```
void FillMemory (
     PVOID Destination,    // адрес блока памяти
     SIZE_T Length,        // размер блока памяти
     BYTE   Fill           // значение, которым заполняется блок
);
```

Для заполнения блока памяти нулями существует отдельная функция:

```
void ZeroMemory (
     PVOID   Destination, // адрес блока памяти
     SIZE_T   Length      // размер блока памяти
);
```

Кроме того, существуют отдельные функции для копирования содержимого некоторого блока памяти:

```
void CopyMemory (
     PVOID      Destination,    // адрес блока назначения
     const VOID      *Source,   // адрес исходного блока
     SIZE_T          Length     // размер копируемого блока
);
```

и перемещения содержимого некоторого блока по новому адресу:

```
void MoveMemory (
     PVOID      Destination,    // адрес блока назначения
```

```
    const VOID      *Source,   // адрес исходного блока
    SIZE_T          Length     // размер перемещаемого блока
);
```

Разница между функциями среди прочего заключается в том, что `MoveMemory` корректно обрабатывает ситуацию, когда исходный блок и результирующий перекрываются, а `CopyMemory` – нет. Кроме того, размер блока назначения должен быть не меньше, чем размер исходного блока.

Главный недостаток API виртуальной памяти – это то, что память выделяется и освобождается достаточно крупными блоками (размером в страницу) и кроме того адрес начала каждого такого блока выравнивается на границу, кратную 64Кб. Для устранения этого недостатка используется API кучи (heap). Этот API позволяет работать с блоками памяти меньшими страницы. Для каждого процесса при запуске выделяется одна куча размером в 1Мб. С этой кучей работают операторы `new/delete` языка C++ и `*alloc/free` функции языка C. Кроме этой кучи процесс может создавать дополнительные кучи. Подробную информацию о возможностях этого API и функциям, входящим в него можно найти в MSDN по адресу:
`http://msdn.microsoft.com/ru-ru/library/windows/desktop/aa366711%28v=vs.85%29.aspx`

Еще один API для работы с памятью в ОС Windows – MMF (Memory Mapped Files) – файлы, отображаемые в виртуальное адресное пространство процесса.

Рассмотрим алгоритм работы с отображаемым в память файлом:

1. Вначале нужно открыть файл, который будет отображаться в память, и получить его дескриптор с нужными правами доступа при помощи функции `CreateFile`.
2. Затем нужно создать объект ядра «секция» с помощью `CreateFileMapping`, также указав требуемый режим доступа.
3. Затем следует отобразить файл (или его часть, если он не помещается в память целиком) в адресное пространство процесса при помощи функции `MapViewOfFile`.
4. После этого можно выполнять любую обработку содержимого файла через обращение к памяти, в которую он отображён. Чаще всего в этом случае содержимое этого блока памяти рассматривается как массив переменных нужного типа или как структура нужного типа.
5. По окончании работы с содержимым проекции файла следует отменить отображение функцией `UnmapViewOfFile`. Если файл был обработан не полностью, то после этого можно перейти к пункту 3 и отобразить следующую часть файла.
6. После того, как работа с файлом завершена, следует закрыть дескрипторы объектов ядра, которые были созданы в порядке,

обратном их созданию, то есть вначале для объекта «секция», а затем для объекта «файл» при помощи функции CloseHandle.

При отображении в память существующего файла он открывается при помощи функции CreateFile. При этом результат, возвращенный функцией CreateFile обязательно нужно проверить на значение INVALIDE_HANDLE_VALUE. Особенность функции CreateFileMapping в том, что в случае такого значения она отработает без ошибок. Причина такого поведения объясняется тем, что это документированный способ организации обмена между программами через файл страничной подкачки.

Пример:

```
HANDLE hFile;
char name[MAX_PATH] = "file.txt";
hFile = CreateFile(name, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (hFile == INVALID_HANDLE_VALUE) {
    cerr << "Ошибка открытия файла" << endl;
    system("pause");
    return 1;
}
```

Для открытого файла создается объект отображения в память. Функция CreateFileMapping, выполняющая это, определена так:

```
HANDLE CreateFileMapping(
HANDLE hFile,
LPSECURITY_ATTRIBUTES lpAttributes,
DWORD flProtect,
DWORD dwMaximumSizeHigh,
DWORD dwMaximumSizeLow,
LPCTSTR lpName);
```

Здесь hFile – дескриптор открытого файла, который будет отображаться в память;

lpAttributes – атрибуты защиты объекта отображения; при значении NULL объект отображения будет доступен создавшему его пользователю и администраторам;

flProtect – флаги, определяющие права доступа к отображению в памяти:

- PAGE_READONLY – данные из файла можно будет только читать;
- PAGE_READWRITE – данные можно будет и читать, и писать;
- PAGE_WRITECOPY – данные можно будет читать, а при записи будет создаваться новая копия файла.

Кроме того в этом параметре могут быть заданы значения, использующиеся системой для проецирования в память исполнимых файлов (например, PAGE_EXECUTE_READ).

Режим доступа к объекту должен соответствовать режиму доступа к файлу, установленному при его открытии.

dwMaximumSizeHigh, dwMaximumSizeLow – старшее и младшее двойное слово, задающее размер объекта отображения. Если параметр dwMaximumSizeHigh равен 0, то размер объекта отображения будет равен текущему значению размера файла.

lpName – строка, задающая имя объекта отображения. Если задаётся имя уже существующего объекта отображения, то функция попытается получить к нему доступ с указанными правами. Допустимо значение NULL, если объект создается безымянным.

В случае успешного завершения функция создает объект отображения файла и возвращает его дескриптор, иначе – NULL.

Пример:

```
HANDLE hFileMapping;
hFileMapping = CreateFileMapping(hFile, NULL,
PAGE_READONLY, 0, 0, NULL);
if(!hFileMapping) {
    cerr << "Ошибка открытия файла" << endl;
    system("pause");
    return 1;
}
```

После создания объекта отображения, файл или часть файла отображается в память процесса функцией MapViewOfFile:

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

Здесь hMapObject – дескриптор объекта отображения файла, возвращенный CreateFileMapping или OpenFileMapping;

dwAccess – защита страниц файла в памяти. Может принимать такие значения:

- FILE_MAP_WRITE – чтение и запись данных;
- FILE_MAP_READ – только чтение данных;
- FILE_MAP_ALL_ACCESS – аналогично комбинации FILE_MAP_READ | FILE_MAP_WRITE;

– FILE_MAP_COPY – при записи создается копия данных файла в pagefile, которая удаляется при отключении файла от адресного пространства процесса.

Режим должен соответствовать режиму, в котором был создан объект отображения файла в память.

dwOffsetHigh, dwOffsetLow – старшая и младшая части смещения начала отображаемого участка от начала файла;

cbMap – размер отображаемого участка файла в байтах. Если этот параметр задать равным 0, то в память будет отображён весь файл.

При успешном завершении функция возвращает указатель на распределенный блок памяти или NULL в случае ошибки.

Пример:

```
char* file;
file = (char*)MapViewOfFile(hFileMapping, FILE_MAP_READ, 0, 0, 0);
```

После отображения в память, адрес по которому расположен отображенный файл может использоваться для обработки содержимого отображения. Например так:

```
for ( int i = 0; i < viewsize; i++)
    if ( file[i] == 'a') count_a++;
```

После обработки файловых данных отображение файла или его части на адресное пространство процесса отменяется функцией:

```
BOOL UnmapViewOfFile(LPVOID lpBaseAddress);
```

Здесь lpBaseAddress – начальный адрес распределенного блока памяти, который должен быть равен адресу, полученному от MapViewOfFile().

Пример:

```
cout << "В файле используется " << count_a << " букв \"a\"" << endl;
UnmapViewOfFile((LPVOID)file);
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Механизм отображения файлов в память используется операционной системой при запуске исполнимых файлов, а также для загрузки DLL в адресное пространство процесса.

Задание. Дано два файла (прилагаются к работе), полученных путём «нарезки» исходного файла на фрагменты по 32 байта и записи чётных фрагментов в один файл, а нечётных фрагментов в другой файл. Разработайте приложение для сборки исходного файла из фрагментов в адресном пространстве процесса, записи его на диск и отображения

пользователю. Для чтения и сборки файлов используйте API согласно Вашему варианту. **Готовые проекты следует загрузить на sdo!**

Варианты:

1) Чтение входных файловых данных в адресное пространство процесса выполняется путём отображения файлов в память. Размещение и сборка результата в памяти выполняется на основе функций API виртуальной памяти. Запись ответа в файл выполняется при помощи функций работы с файлами в синхронном режиме.
2) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в синхронном режиме. Размещение и сборка результата в памяти выполняется на основе функций API работы с кучей (не в куче по умолчанию). Запись ответа в файл выполняется при помощи функций работы с файлами в асинхронном режиме.
3) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в асинхронном режиме. Размещение и сборка результата в памяти выполняется на основе операторов/функции языка программирования для работы с памятью (куча процесса по умолчанию). Запись ответа в файл выполняется путём применения отображения файла в память.
4) Чтение входных файловых данных в адресное пространство процесса выполняется путём отображения файлов в память. Размещение и сборка результата в памяти выполняется на основе функции API кучи (не в куче процесса по умолчанию). Запись ответа в файл выполняется при помощи функций работы с файлами в синхронном режиме.
5) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в синхронном режиме. Размещение и сборка результата в памяти выполняется на основе функции API виртуальной памяти. Запись ответа в файл выполняется при помощи API отображения файла в память.
6) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в асинхронном режиме. Размещение и сборка результата в памяти выполняется на основе функции API кучи (не в куче по умолчанию). Запись ответа в файл выполняется при помощи функций работы с файлами в асинхронном режиме.
7) Чтение входных файловых данных в адресное пространство процесса выполняется путём отображения файлов в память. Размещение и сборка результата в памяти выполняется на основе операторов/функций работы с памятью языка программирования (с использованием кучи процесса по умолчанию). Запись ответа в файл выполняется при помощи механизма отображения файла на адресное пространство процесса.

8) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в синхронном режиме. Размещение и сборка результата в памяти выполняется на основе функций API работы с виртуальной памятью. Запись ответа в файл выполняется при помощи функций работы с файлами в асинхронном режиме.
9) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в асинхронном режиме. Размещение и сборка результата в памяти выполняется на основе функций API работы с виртуальной памятью. Запись ответа в файл выполняется при помощи функций работы с файлами в синхронном режиме.
10) Чтение входных файловых данных в адресное пространство процесса выполняется путём отображения файлов в память. Размещение и сборка результата в памяти выполняется на основе операторов/функций распределения памяти языка программирования (используя кучу процесса по умолчанию). Запись ответа в файл выполняется при помощи функций работы с файлами в асинхронном режиме.
11) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в синхронном режиме. Размещение и сборка результата в памяти выполняется на основе функций API виртуальной памяти. Запись ответа в файл выполняется при помощи функций работы с файлами в синхронном режиме.
12) Чтение входных файловых данных в адресное пространство процесса выполняется при помощи функций работы с файлами в асинхронном режиме. Размещение и сборка результата в памяти выполняется на основе функций API кучи (не в куче процесса по умолчанию). Запись ответа в файл выполняется при помощи проецирования файла на адресное пространство процесса.