Архитектура памяти в Win32 API

Организация «статической» виртуальной памяти

Работа приложений с виртуальной памятью

- Резервирование и выделение памяти производится блоками. Начальный адрес блока должен быть выровнен на границу 64K (округляется вниз), а размер кратен размеру страницы (округляется вверх). При выделении память обнуляется.
- Блок адресов в адресном пространстве процесса может находиться в одном из трех состояний:
  - Выделен (committed) – блоку адресов назначена физическая память либо часть файла подкачки.
  - Зарезервирован (reserved) – блок адресов помечен как занятый, но физическая память не распределена.
  - Свободен (free) – блок адресов не выделен и не зарезервирован.

Функции API для работы виртуальной памятью

- ***VirtualAlloc***
- ***VirtualAllocEx***
- ***VirtualFree***
- ***VirtualFreeEx***
- ***VirtualLock***
- ***VirtualUnlock***
- ***VirtualProtect***
- ***VirtualProtectEx***

Функции API для работы виртуальной памятью

- Для резервирования региона памяти в адресном пространстве процесса или выделения ее используется функция *VirtualAlloc*, а для освобождения – функция *VirtualFree*. Для работы в адресном пространстве произвольного процесса необходимо использовать функции *VirtualAllocEx* и *VirtualFreeEx.*
- Выделенные страницы можно заблокировать в памяти, т.е. запретить их вытеснение в файл подкачки. Для этих целей служит пара функций *VirtualLock* и *VirtualUnlock*. Процессу не разрешается блокировать более 30 страниц.
- Для изменения атрибутов защиты регионов используются функции *VirtualProtect* и *VirtualProtectEx*. Причем, первая позволяет изменять атрибуты защиты в адресном пространстве текущего процесса, а вторая – произвольного.

Функции API для работы с ВП: *VirtualAlloc*

```
LPVOID VirtualAlloc (
    // адрес, по которому надо зарезервировать
    // или выделить память
    LPVOID lpAddress,
    // размер выделяемого региона
    DWORD dwSize,
    // тип распределения памяти
    DWORD flAllocationType,
    // тип защиты доступа
    DWORD flProtect
);
```

Функции API для работы с ВП: *VirtualAllocEx*

```
LPVOID VirtualAllocEx (
    // дескриптор процесса
    HANDLE hProcess,
    // адрес, по которому надо зарезервировать
    // или выделить память
    LPVOID lpAddress,
    // размер выделяемого региона
    DWORD dwSize,
    // тип распределения памяти
    DWORD flAllocationType,
    // тип защиты доступа
    DWORD flProtect
);
```

Функции API для работы с ВП: *VirtualAlloc*

- Параметр **flAllocationType** может принимать следующие значения:
  - **MEM_RESERVE** - резервирует блок адресов без выделения памяти;
  - **MEM_COMMIT** - отображает ранее зарезервированный блок адресов на физическую память или файл подкачки, выделяя при этом память. Может комбинироваться с флагом MEM_RESERVE для одновременного резервирования и выделения;
  - **MEM_TOP_DOWN** - выделяет память по наибольшему возможному адресу. Имеет смысл только при lpAddress = NULL. В Windows 95 игнорируется.
  - **MEM_DECOMMIT** - освободить выделенную память;
  - **MEM_RELEASE** - освободить зарезервированный регион. При использовании этого флага параметр dwSize должен быть равен нулю.

Функции API для работы с ВП: *VirtualAlloc*

- Параметр **flProtect** - тип защиты доступа выделяемого региона:
  - **PAGE_READONLY** - допускается только чтение;
  - **PAGE_READWRITE** - допускается чтение и запись;
  - **PAGE_EXECUTE** - допускается только выполнение;
  - **PAGE_EXECUTE_READ** - допускается исполнение и чтение;
  - **PAGE_EXECUTE_READWRITE** - допускается выполнение, чтение и запись;
  - **PAGE_GUARD** - дополнительный флаг защиты, который комбинируется с другими флагами. При первом обращении к странице этот флаг сбрасывается и возникает исключение STATUS_GUARD_PAGE. Этот флаг используется для контроля размеров стека с возможностью его динамического расширения;
  - **PAGE_NOCACHE** - запрещает кэширование страниц. Может быть полезен при разработке драйверов устройств (например, данные в видеобуфер должны переписываться сразу, без кэширования).

Функции API для работы ВП: *VirtualFree*

```
BOOL VirtualFree (
    // адрес региона, который надо освободить
    LPVOID lpAddress,
    // размер освобождаемого региона
    DWORD dwSize,
    // тип освобождения
    DWORD dwFreeType
);
```

Функции API для работы ВП: *VirtualFree*

- **dwSize** – размер, если мы будем использовать тип освобождения, как **MEM_RELEASE**, то размер должен быть установлен в 0.
- **dwFreeType** – будет определять какая операция произойдет с памятью:
  - **MEM_RELEASE** – освобождена;
  - **MEM_DECOMMIT** – зарезервирована, но не используется.
- При успешном выполнении функция вернет TRUE в случае успеха и FALSE - в случае неудачи.

Функции API для работы ВП: *VirtualFreeEx*

```
BOOL VirtualFreeEx (
    // дескриптор процесса
    HANDLE hProcess,
    // адрес региона, который надо освободить
    LPVOID lpAddress,
    // размер освобождаемого региона
    DWORD dwSize,
    // тип освобождения
    DWORD dwFreeType
);
```

Функции API для работы ВП: *VirtualProtect* и *VirtualProtectEx*

- Для изменения атрибутов защиты регионов используются функции *VirtualProtect* и *VirtualProtectEx*.
- *VirtualProtect* позволяет изменять атрибуты защиты в адресном пространстве текущего процесса.
- *VirtualProtectEx* позволяет изменять атрибуты защиты в адресном пространстве произвольного процесса.

Функции API для работы ВП: *VirtualProtect* и *VirtualProtectEx*

```
BOOL VirtualProtect (
    // адрес региона для установки флага
    LPVOID lpAddress,
    // размер региона
    DWORD dwSize,
    // флаг
    DWORD flNewProtect,
    // адрес для сохранения старых флагов
    PDWORD lpflOldProtect
);
```

Функции API для работы ВП: *VirtualLock* и *VirtualUnlock*

- Функция ***VirtualLock()*** позволят предотвратить запись памяти на диск.
  - BOOL VirtualLock (LPVOID lpAddress, *// адрес начала памяти* SIZE_T dwSize *// количество байтов*);
- Если фиксация больше не нужна, то ее можно убрать функцией ***VirtualUnlock()***.
  - BOOL VirtualUnlock (LPVOID lpAddress, *// адрес начала памяти* SIZE_T dwSize *// количество байтов*);
- При успешном выполнении обе функции возвращают ненулевое значение.

Функции API для работы ВП: *VirtualQuery* и *VirtualQueryEx*

- Функции *VirtualQuery* и *VirtualQueryEx* позволяют определить статус указанного региона адресов.

Функции API для работы виртуальной памятью

- Процессу не разрешается блокировать более 30 страниц.
- Для настройки рабочего множества процесса может использоваться и функция *SetProcessWorkingSetSize*, которая может снять это ограничение.
- *SetProcessWorkingSetSize* позволяет задать для процесса минимальный и максимальный размер рабочего множества процесса.

Функции API для работы с ВП: SetProcessWorkingSetSize

```
BOOL SetProcessWorkingSetSize (
    // дескриптор процесса
    HANDLE hProcess,
    // мин. размер рабочего мн-ва процесса, в байтах
    DWORD dwMinimumWorkingSetSize,
    // макс. размер рабочего мн-ва процесса, в байтах
    DWORD dwMaximumWorkingSetSize
);
```

Функции API для работы с ВП: SetProcessWorkingSetSize

- Если и *dwMinimumWorkingSetSize* и *dwMaximumWorkingSetSize* имеют значение **–** (минус) 1, функция временно урезает рабочее множество процесса до нуля. Это, по существу, выносит процесс за пределы физической оперативной памяти.
- Аналогичный результат достигается с помощью функции ***EmptyWorkingSet***.

Функции API для работы с ВП: SetProcessWorkingSetSize

- Дескриптор процесса должен иметь права доступа **PROCESS_SET_QUOTA**.
- Если значения *dwMinimumWorkingSetSize* или *dwMaximumWorkingSetSize* больше, чем текущий размер рабочего множества памяти процесса, данный процесс должен иметь привилегию **SE_INC_BASE_PRIORITY_NAME**.

Получение справочной информации по ВП процесса

- *GetProcessWorkingSetSize* – получение текущих значений минимального и максимального размера рабочего множества процесса.
- *GetProcessMemoryInfo* – получение расширенной статистики по использованию ВП процесса, например:
  - количество страничных прерываний;
  - текущий и пиковый размер рабочего множества процесса;
  - текущее и пиковое использование файла подкачки.

Архитектура памяти в Win32 API

Организация «динамической» виртуальной памяти

«Кучи» (heaps)

- Кучи (heaps) – это динамически распределяемые области данных.
- При порождении процесса ему предоставляется куча размером 1 Мбайт по умолчанию. Ее размер может изменяться параметром /HEAP при построении исполняемого модуля.
- Функции библиотеки времени исполнения компилятора CRT (malloc(), free() и т. д.) используют возможности куч.

Функции создания и использования «куч»

- HANDLE GetProcessHeap (VOID) – для получения дескриптора кучи по умолчанию;
- LPVOID HeapAlloc (HANDLE hHeap, DWORD dwFlags, DWORD dwSize) – для выделения из кучи блока памяти заданного размера и возвращения указателя;
- LPVOID HeapReAlloc (HANDLE hHeap, DWORD dwFlags, LPVOID lpOldBlock, DWORD dwSize) – для изменения размера выделенного блока памяти с возможностью перемещения блока при необходимости;
- BOOL HeapFree (HANDLE hHeap, DWORD dwFlags, LPVOID lpMem) – для освобождения выделенного блока памяти кучи.

Создание дополнительных «куч»

- для повышения эффективности управления памятью;
- для уменьшения рабочего множества процесса ;
- для повышения эффективности работы многонитевых приложений;
- для “защиты” друг от друга различных структур данных;
- для быстрого освобождение всей памяти в куче.

Повышение эффективности управления памятью

- В системах со страничной организацией отсутствует проблема фрагментации физической памяти. Однако существует проблема фрагментации адресного пространства. В 4Gb адресном пространстве эта проблема не актуальна, но она имеет значение в 1Mb куче.
- Если элементы какой-либо структуры имеют один размер, а элементы другой структуры - другой размер, то полезно размещать эти структуры в разных кучах.

Уменьшение рабочего множества процесса

- В соответствии с принципом локальности, работа с разными структурами, чаще всего, происходит не одновременно. Границы элементов разных структур не выровнены на границу страницы.
- Обращение к элементам одной структуры вызывает подкачку всей страницы, а, значит и элементов другой структуры. Это увеличивает рабочее множество процесса.

Создание и уничтожение «кучи»

```
HANDLE HeapCreate (
    DWORD dwFlags,
    DWORD dwInitialSize,
    DWORD dwMaximumSize
);

BOOL HeapDestroy ( HANDLE hHeap);
```

Создание «кучи» - *dwFlags*

- **HEAP_GENERATE_EXCEPTIONS** – указывает системе на то, что в случае возникновения ошибки необходимо генерировать исключительную ситуацию. Это будет происходить во всех случаях, когда функция должна была бы возвратить значение **null**.
- **HEAP_NO_SERIALIZE** – указывает, что пока выполняется текущий вызов HeapAlloc, к куче не будут происходить обращения из других потоков (т.е. программист сам берет на себя исключение ситуаций одновременных обращений).
- **HEAP_ZERO_MEMORY** – указывает, что выделяемая память должна инициализироваться нулями. В противном случае память не обязательно инициализируется нулями.

Дополнительные возможности по управлению «кучами»

- UINT HeapCompact (HANDLE hHeap, DWORD fdwFlags);
- BOOL HeapLock (HANDLE hHeap);
- BOOL HeapUnlock (HANDLE hHeap);
- BOOL HeapWalk (HANDLE hHeap, PPROCESS_HEAP_ENTRY pHeapEntry);

Архитектура памяти в Win32 API

Файлы, проецируемые в память

Проецируемые файлы

“Как и виртуальная память, проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память. Различие между этими механизмами состоит в том, что в последнем случае физическая память не выделяется из системного страничного файла, а берется из файла, уже находящегося на диске. Как только файл спроецирован в память, к нему можно обращаться так, как будто он в нее целиком загружен.”

(Джеффри Рихтер. Windows для профессионалов.)

Проецируемые файлы

- Файлы, проецируемые (отображаемые) в память, - это один из самых замечательных сервисов, которые Win32 предоставляет программисту. Его существование стирает для программиста грань между оперативной и дисковой памятью. Действительно, с точки зрения классической теории кэш, оперативная память и дисковое пространство - это три вида памяти, отличающиеся скоростью доступа и размером. Но если заботу о перемещении данных между кэшем и ОП берут на себя процессор и ОС, то перемещение данных между ОП и диском обычно выполняет прикладной процесс с использованием функций read() и write().
- Win32 действует иначе: ОС берет на себя заботу о перемещении страниц адресного пространства процесса, находящихся в файле подкачки, причем в качестве файла подкачки может быть использован любой файл. Иначе говоря, страницы ВП любого процесса могут быть помечены как выгруженные, а в качестве места, куда они выгружены, может быть указан файл. Теперь при обращении к такой странице VMM произведет ее загрузку, используя стандартный механизм свопинга. Это позволяет работать с произвольным файлом как с регионом памяти.

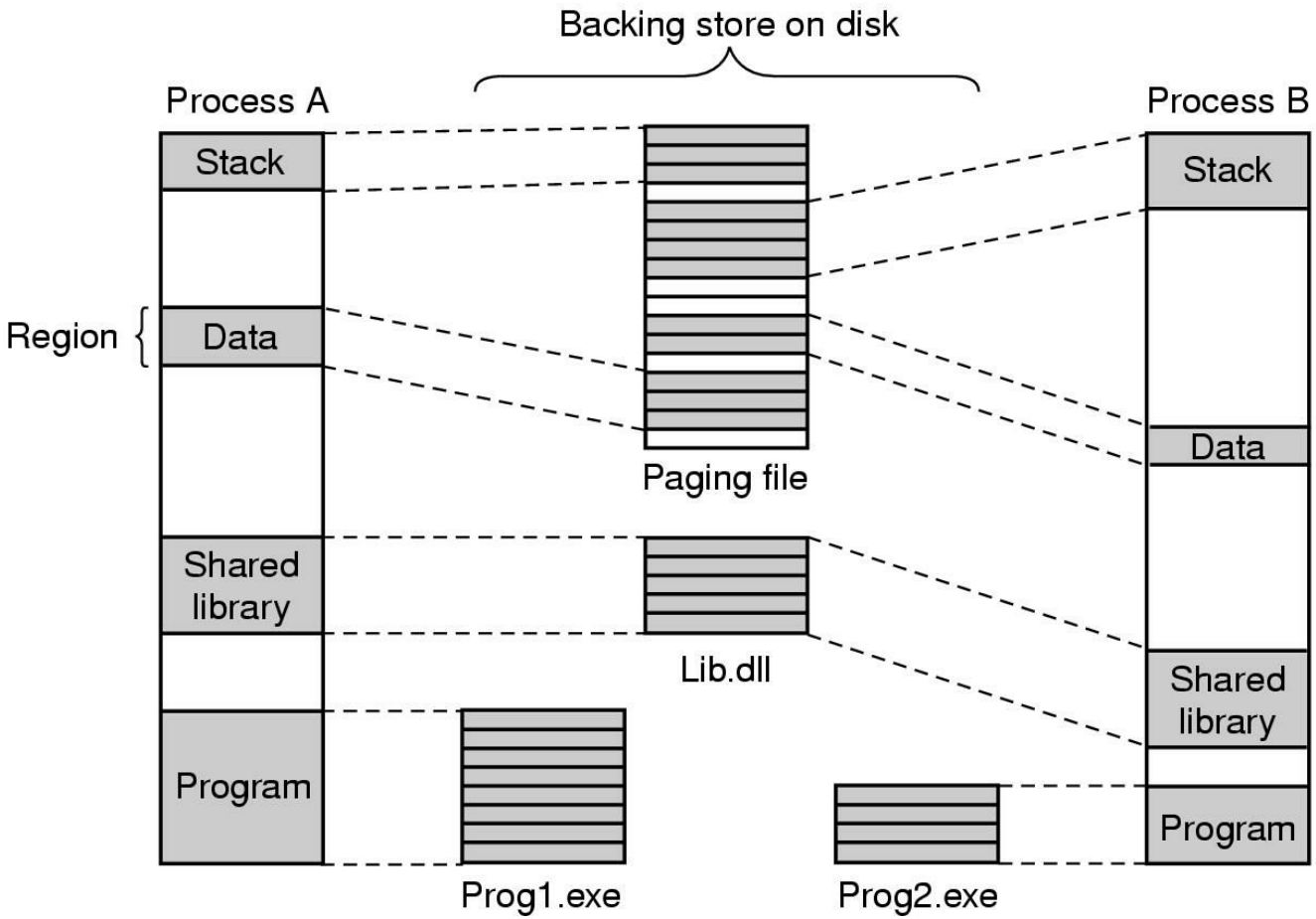
Применение проецируемых файлов

- Для запуска исполняемых файлов (EXE) и динамически связываемых библиотек (DLL).
- Для работы с файлами.
- Для одновременного использования одной области данных двумя процессами.

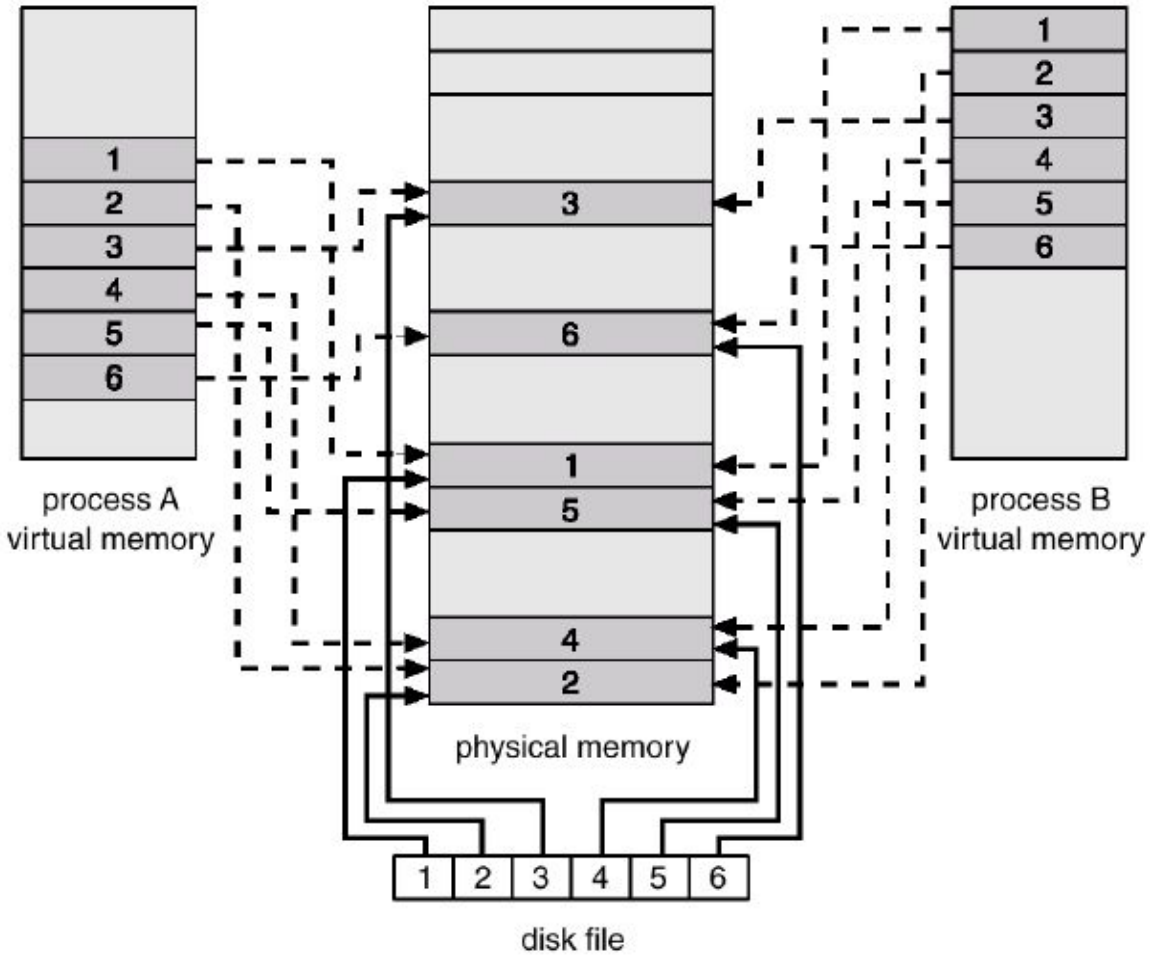
Запуск процесса

1. Создать виртуальное адресное пространство процесса (размером 4Gb).
2. Резервировать в ВАП регион размером, достаточным для размещения исполняемого файла. Начальный адрес региона определяется в заголовке EXE-модуля. Обычно он равен 0x00400000.
3. Отобразить исполняемый файл на зарезервированное адресное пространство.
4. Таким же образом отобразить на ВАП процесса необходимые ему динамически связываемые библиотеки. Информация о необходимых библиотеках находится в заголовке EXE-модуля. Желательное расположение региона адресов описано внутри библиотеки.

Запуск EXE-файлов и DLL-библиотек



Одновременное использование одной области данных двумя процессами



Файлы данных, проецируемые в память

Проецирование файла данных в память:

- Создается объект ядра “файл”. Для создания объекта “файл” используется функция ***CreateFile***.
- С помощью функции ***CreateFileMapping*** создается объект ядра “проецируемый файл”. При этом используется дескриптор файла, возвращенный функцией CreateFile.
- Производится отображение объекта “проецируемый файл” или его части на адресное пространство процесса. Для этого применяется функция ***MapViewOfFile***.

Завершение проецирования файла данных:

- Выполняется откреплeние файла от адресного пространства процесса с помощью функции ***UnmapViewOfFile***.
- Выполняется уничтожение объектов “файл” и “проецируемый файл” с помощью функции ***CloseHandle***.

Обеспечение когерентности

- Если один процесс меняет разделяемую область данных, то она меняется и для другого процесса.
- Операционная система обеспечивает ***когерентность*** разделяемой области данных для всех процессов. Но для обеспечения когерентности процессы должны работать с одним объектом “проецируемый файл”, а не с одним файлом.

Создание объекта «проецируемый файл»

```
HANDLE CreateFileMapping (
    // дескриптор файла
    HANDLE hFile,
    // атрибуты защиты объекта
    LPSECURITY_ATTRIBUTES lpAttributes,
    // атрибуты защиты
    DWORD flProtect,
     // старшее слово размера
    DWORD dwMaximumSizeHigh,
    // младшее слово размера
    DWORD dwMaximumSizeLow,
     // имя объекта
    LPCTSTR lpName
);
```

Открытие объекта «проецируемый файл»

```
HANDLE OpenFileMapping (
    // режим доступа
    DWORD dwDesiredAccess,
    // флажок наследования
    BOOL bInheritHandle,
    // имя объекта
    LPCTSTR lpName
);
```

Функция проецирования области

```
LPVOID MapViewOfFile (
    // дескриптор объекта проецируемый файл
    HANDLE hFileMappingObject,
     // режим доступа
    DWORD dwDesiredAccess,
    // старшее DWORD смещения
    DWORD dwFileOffsetHigh,
    // младшее DWORD смещения
    DWORD dwFileOffsetLow,
     // число отображаемых байтов
    SIZE_T dwNumberOfBytesToMap
);
```

Функция проецирования области по определенному адресу

```
LPVOID MapViewOfFileEx (
    // дескриптор отображаемого объекта
    HANDLE hFileMappingObject,
    // режим доступа
    DWORD dwDesiredAccess,
    // старшее DWORD смещения
    DWORD dwFileOffsetHigh,
    // младшее DWORD смещения
    DWORD dwFileOffsetLow,
    // число отображаемых байтов
    SIZE_T dwNumberOfBytesToMap,
    // начальный адрес
    LPVOID lpBaseAddress
);
```

Параметр *dwDesiredAccess*

- Параметр *dwDesiredAccess* определяет требуемый режим доступа для страниц ВП, используемых для отображения:
  - **FILE_MAP_WRITE** – доступ к операциям чтения-записи, проецируемый файл должен быть создан с защитой **PAGE_READWRITE**.
  - **FILE_MAP_READ** – доступ только для чтения, проецируемый файл должен быть создан с защитой **PAGE_READWRITE** или **PAGE_READONLY**.
  - **FILE_MAP_ALL_ACCESS** – то же самое, что и **FILE_MAP_WRITE** .
  - **FILE_MAP_COPY** – копирование при доступе для записи, проецируемый файл должен создаваться с флажком защиты **PAGE_WRITECOPY**.
  - **FILE_MAP_EXECUTE** – доступ к исполнению кода из отображаемой памяти , проецируемый файл должен быть создан с доступом **PAGE_EXECUTE_READWRITE** или **PAGE_EXECUTE_READ** .

Функция отмены проецирования области

```
BOOL UnmapViewOfFile (
    // начальный адрес
   LPCVOID lpBaseAddress
);
```

Создание и использование проецируемых файлов

- Общий механизм таков: один процесс создает объект “проецируемый файл” с помощью функции *CreateFileMapping, передавая* в параметре *lpName* имя объекта, которое является глобальным в системе.
- Другой процесс открывает уже созданный объект “проецируемый файл” по глобальному имени.
- Теперь два процесса могут совместно использовать объект “проецируемый файл”. При этом, при помощи функции *MapViewOfFile* каждый процесс проецирует этот объект на свое ВАП и используют эту часть адресного пространства как разделяемую область данных.

Взаимодействие процессов через общую область данных

Для обеспечения *когерентности* процессы должны работать с одним объектом “проецируемый файл”, а не с одним файлом.

Использование файла подкачки

- Общая область данных может быть создана не только путем проецирования файла, но и путем проецирования части файла подкачки.
- Для этого в функцию ***CreateFileMapping*** необходимо передать в качестве параметра не описатель ранее открытого файла, а -1. В этом случае необходимо задать размеры выделяемой области.

Пример работы с проецированным файлом

```
HANDLE hFile, hFileMapping;
PVOID pMassive;
hFile = CreateFile( “File Name”, ... );
hFileMapping = CreateFileMapping( hFile, ... );
CloseHandle( hFile ) ;
pMassive = MapViewOfFile( hFileMapping, ... );

//Здесь производится работа с массивом pMassive
UnmapViewOfFile( pMassive );
```

Архитектура памяти в Win32 API

Доступ к ВП другого процесса

Доступ к ВП другого процесса

- Мы говорили о том, что менеджер ВП изолирует ВАП процессов для защиты от несанкционированного доступа.
- Для решения задачи обмена информацией между процессами следует использовать проецируемые файлы или другие специальные технологии, которые будут рассмотрены в разделе «Межпроцессное взаимодействие».
- Однако есть еще один способ получить доступ к памяти другого процесса.

Функции *ReadProcessMemory* и *WriteProcessMemory*

- **ReadProcessMemory** – читает данные из области памяти в заданном процессе.
- **WriteProcessMemory** – пишет данные области памяти в заданном процессе.

- Пример для самостоятельной работы: http://faceh0r.narod.ru/doc/GameTrainer.html

Функции *ReadProcessMemory* и *WriteProcessMemory*

```
BOOL WriteProcessMemory (
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesWritten
);
```

Функции *ReadProcessMemory* и *WriteProcessMemory*

```
BOOL ReadProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesRead
);
```

Архитектура памяти в Win32 API

Локальная память потока (TLS)

Локальная память потока (TLS)

- Для решения ситуаций, когда есть данные, которые должны быть связаны индивидуально с каждым потоком, необходимо использовать механизм локальной памяти потока **TLS (Thread Local Storage)**.
- Поток имеет доступ лишь к своим TLS-переменным, и не может обратиться к TLS-переменным любого другого потока.

Назначение TLS

- Например, пусть процесс владеет некоторым массивом. Каждый элемент массива вместе с его содержимым соответствует отдельному потоку. Откуда поток узнает, какой индекс в глобальном массиве его? Да, можно передать функции потока **ThreadProc** параметр в виде индекса. Тогда индекс будет храниться в локальной переменной.
- Но представьте, что **ThreadProc** вызывает какую-то функцию потом еще одну, и так он может вызывать сотни функций с разными уровнями вложенности. Куда денется индекс, которым владеет поток?? Да, можно передавать индекс каждой функции параметром, но это очевидно будет сказываться на эффективности. Очевидным решением стало создание для потока локальной памяти– **TLS**.

Виды TLS

- динамическая TLS:
  - размер ячейки локальных данных – 4 байт
  - количество локальных данных – ограничено
  - требует использования функций API
- статическая TLS:
  - размер ячейки локальных данных – не ограничен
  - количество локальных данных – не ограничено
  - не требует использования функций API

Динамическая TLS

- Каждому потоку выделяется определенное количество ячеек размером 4 байта.
- Количество ячеек зависит от версии Windows, самое маленькое – это 64 ячейки для Windows 95, в ОС Windows 2000 и старше – 1088 ячеек.
- Для работы с динамической **TLS** поток может использовать четыре функции –
  - **TlsAlloc,**
  - **TlsGetValue,**
  - **TlsSetValue,**
  - **TlsFree**.

Функции для работы с динамической TLS: *TlsAlloc*

- Итак, чтобы получить 4-х байтную ячейку, мы вызываем функцию – **TlsAlloc**:

  DWORD TlsAlloc (VOID).

  Если ассоциированный массив ячеек полностью использован, возвращаемое значение будет равно TLS_OUT_OF_INDEXES, что сообщает об ошибке выделения ячейки.

- Данная функция резервирует ячейку в локальной памяти потока и возвращает индекс этого DWORD'а. Далее этот индекс передают в функции **TlsSetValue** и **TlsGetValue**.

Функции для работы с динамической TLS: *TlsSetValue* и *TlsGetValue*

```
BOOL TlsSetValue(
 // TLS index to set value for
 // value to be stored
 DWORD dwTlsIndex,
 LPVOID lpvTlsValue
);

LPVOID TlsGetValue(
 // TLS index to retrieve value
 DWORD dwTlsIndex
);
```

Функции для работы с динамической TLS: *TlsSetValue* и *TlsGetValue*

- Функция **TlsSetValue** устанавливает значение в ячейке с данным индексом. Она принимает индекс возвращенный функцией TlsAlloc, а также значение для сохранения в ячейке с данным индексом. Функция возвращает 1 в случае успеха и 0 в противном случае. Для получения дополнительной информации в случае ошибки как обычно вызывайте функцую GetLastError.
- Функция **TlsGetValue** соответственно возвращает значение указанное данным индексом. В случае ошибки возвращается 0. Чтобы различить нулевое значение в ячейке, с сигнализацией об ошибке вызывайте GetLastError. Если ошибки не было, то GetLastError вернет NO_ERROR.

Статическая TLS

- Статическая локальная память позволяет хранить данные любого фиксированного размера.
- Статическая локальная память потока опирается на механизмы загрузчика и свои собственные структуры.
- Статическая локальная память для потока не использует API функций.
- Компиляторы высокоуровневых языков предоставляют специальный синтаксис для работы со статической TLS.
- В программах на ассемблере статическую TLS придется реализовывать ее вручную.

Статическая TLS

- Так, компилятор Microsoft VC++ позволяет использовать следующий синтаксис для создания переменной специфичной для потока:

  Цитата:

  __declspec(thread) int tls_i = 1;

- Этим кодом создается переменная tls_i локальная для потока, которая инициализируется значением 1. Переменная может быть любого типа.

Объявление переменной в статической TLS

- Переменная, указываемая за **__declspec**(**thread**), должна быть либо глобальной, либо статической внутри (или вне) функции.
- Локальную переменную с модификатором **__declspec**(**thread**) объявить нельзя.

Ограничения на использование статической TLS

1. Спецификатор __declspec(thread) может быть использован только с данными.
2. Как было сказано выше, TLS можно применять только к статическим переменным – т.е. нелокальным.
3. Нельзя получить адрес переменной TLS, т.к. он не является константой.
4. Могут возникнуть проблемы с DLL, которую динамически загружают с помощью LoadLibrary. Для DLL, которые могут быть загружены с помощью LoadLibrary и которые используют TLS рекомендуется использовать динамическую TLS.