**Виртуальная память** — метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём **автоматического перемещения частей программы между основной памятью и вторичным хранилищем** (например, жёстким диском).

**Страничный файл (Page File):** Это файл на жестком диске, который операционная система использует как расширение оперативной памяти (RAM). Когда оперативной памяти не хватает для всех запущенных программ и процессов, система перемещает менее активные данные из оперативной памяти в страничный файл. Этот процесс называется подкачкой или свопингом.

В основе виртуальной памяти лежит идея, что у каждого процесса **имеется собственное адресное пространство**, которое разбивается на участки, называемые **страницами** (обычно размером 4 КБ). **Каждая страница представляет собой непрерывный диапазон адресов**. Эти страницы **отображаются на физическую память**.

1. **Виртуальный адрес:** Адрес, который используется инструкциями программы и хранится в указателях, называется виртуальным адресом.
2. **Физический адрес:** Адрес, который непосредственно используется аппаратурой для доступа к данным в ОЗУ, называется физическим адресом. Процессор выставляет его на шину памяти.

Дальше проецируемые файлы и отображенные файлы – одно и то же.

**Отображение (проецирование) файла в память (на память)** — это способ работы с файлами, при котором всему файлу или некоторой непрерывной его части ставится в **соответствие определённый участок виртуальной памяти** (диапазон виртуальных адресов). При этом чтение данных из этих адресов фактически приводит к чтению данных из отображённого файла, а запись данных по этим адресам приводит к записи этих данных в файл. Физическая память (оперативная память, RAM) выделяется операционной системой по мере необходимости, когда происходит фактическое обращение к данным в виртуальной памяти.

**Виртуальная память:**

* Когда операционная система использует виртуальную память, она может нуждаться в дополнительной памяти, если у компьютера заканчивается физическая RAM.
* В этом случае операционная система возьмет наименее используемые страницы памяти из RAM и перенесет их в файл подкачки (page file).
* Физическая память (RAM), которая была занята этими страницами, освобождается, и её можно использовать для других процессов или данных. *Данные, которые находились в RAM, сохраняются в файл подкачки*.
* Когда программе снова понадобятся эти страницы, операционная система извлечет их из файла подкачки обратно в RAM (возможно, выгрузив другие страницы в файл подкачки, чтобы освободить место).

Файлы отображенные в память:

* Файл отображается в память, чтобы программа могла обращаться к нему так, как если бы это была часть оперативной памяти.
* Если операционной системе необходимо освободить RAM, которая используется для хранения страниц отображенного файла, она может "выгрузить" эти страницы, но *не обязательно в файл подкачки*.
* Вместо этого, поскольку страницы отображенного файла уже соответствуют данным, хранящимся непосредственно в *исходном файле на диске*, операционная система может просто *отбросить* эти страницы из RAM.
* Когда программе снова потребуется доступ к этим страницам, операционная система *просто повторно прочитает их из исходного файла на диске* (а не из файла подкачки).

Как и виртуальная память, проецируемые файлы **позволяют резервировать регион адресного пространства и передавать ему физическую память**

* **Не использует файл подкачки (страничный файл).**
* Данные загружаются *напрямую из файла на диске* в ОЗУ, когда программе они нужны. Загружаются только те части файла, к которым происходит обращение.
* Неиспользуемые данные выгружаются из ОЗУ.

После проецирования файла в память с ним можно работать так, **будто он уже целиком загружен в оперативную память**.

Проецируемые файлы применяются для:

* загрузки и выполнения исполняемых файлов и библиотек. Это позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению
* доступа к файлу данных, размещенному на диске. Обходимся без операций файлового ввода-вывода. Файл как бы становится частью оперативной памяти. Программе не нужно каждый раз обращаться к диску — данные читаются напрямую из памяти.
* разделения данных между несколькими процессами, выполняемыми на одной машине. (Например, в Windows есть и другие методы для совместного доступа разных процессов к одним данным – но все они так или иначе реализованы на основе проецируемых в память файлов.)

Примитивы, на основе которых диспетчер памяти реализует общую память, называются **объектами секций**; в Windows API они представлены **объектами отображенных файлов (объект «проекция файла»)**

Этот важнейший примитив диспетчера памяти используется для отображения виртуальных адресов в основную память, в страничный файл или в любой другой файл, к содержимому которого приложение захочет обращаться так, как если бы оно находилось в памяти

Соответственно в Windows выделяют два вида объектов секций:

* **Секции на базе файлов на диске** – файл называется **спроецированным** **(отображенным)**
* **Секции на базе страничных файлов –** объект секции в этом случае связан с подтвержденной памятью (для реализации общей памяти). Подтвержденная память, это виртуальная память, которая связана с физической. То есть в подтвержденной память процесс уже может хранить данные, а в зарезервированной не может. Зарезервированную память еще нужно подтвердить (связать её с физической). Страницы записываются в страничный файл, если возникнет необходимость в физической памяти. Общие подтвержденные страницы всегда заполняются нулями при первом обращении; таким образом предотвращается возможная утечка конфиденциальных данных!

Объект секции может ссылаться на файлы гораздо большего размера, чем может поместиться в адресном пространстве процесса. (Если объект секции существует на базе страничного файла, то в страничном файле и/или ОЗУ должно существовать достаточно пространства для его создания)

* **Адресное пространство процесса** — это диапазон виртуальных адресов, которые процесс может использовать.
* **Файл может быть огромным** (например, терабайты данных). Но процесс не может сразу отобразить весь файл в своё адресное пространство, потому что оно ограничено.
* **Решение**: Windows позволяет отображать только ту часть файла, которая нужна процессу в данный момент. Эта часть называется **представлением секции** (view of a section).

Каждая секция памяти имеет одно или несколько соответствующих представлений

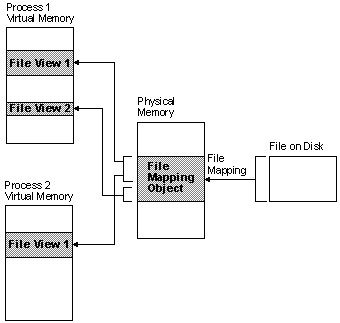
**Представление секции (section view)** – это часть секции, которая фактически видна процессу. Процесс создания представления для секции называется **отображением (проецированием) представления секции**

Каждый процесс, который манипулирует содержимым секции, имеет свое собственное представление; процесс также может иметь несколько представлений.

Например, можно спроецировать в одно представление первые 10 Кб файла, а затем – первые 4 Кб того же файла в другое представление. Пока Вы проецируете один и тот же объект, система гарантирует **согласованность** **(coherence)** отображаемых данных

Скажем, если программа изменяет содержимое файла в одном представлении, это приводит к обновлению данных и в другом. Так происходит потому, что система, несмотря на многократную проекцию страницы на виртуальное адресное пространство процесса, хранит данные на единственной странице оперативной памяти.

**!!! Windows позволяет создавать несколько объектов «проекция файла», связанных с одним и тем же файлом данных. Но тогда у Вас не будет гарантий, что содержимое представлений этих объектов согласованно. Такую гарантию Windows дает только для нескольких представлений одного объекта «проекция файла»**

****

Четыре возможных метода реализации программы, меняющей порядок следования всех байтов в файле на обратный

* **один файл, один буфер**

Простейший метод.

1. Программа выделяет блок оперативной памяти (буфер), который по размеру равен файлу
2. Открываем файл, считываем его содержимое в буфер, закрываем
3. Располагая в памяти содержимым файла, можно поменять первый байт с последним, второй – с предпоследним и т. д. Этот процесс будет продолжаться, пока мы не поменяем местами два смежных байта, находящихся в середине файла
4. Закончив эту операцию, вновь открываем файл и перезаписываем его содержимое

Проблемы:

a) придется выделить блок памяти такого же размера, что и файл. Если файл большой – проблема

b) если перезапись вдруг прервется, содержимое файла будет испорчено

* **два файла, один буфер**

1. Программа открывает исходный файл (который нужно обработать) и создаёт новый файл (куда будет записан результат).
2. Программа выделяет небольшой буфер в оперативной памяти (например, 8 КБ).
3. Указатель файла устанавливается на 8 КБ от конца исходного файла.
4. Программа считывает последние 8 КБ исходного файла в буфер.
5. В буфере меняется порядок байтов на обратный.
6. Изменённые данные записываются в новый файл.
7. Указатель файла сдвигается на 8 КБ ближе к началу, и процесс повторяется, пока не будет обработан весь файл.
8. После обработки исходный файл удаляется, а новый файл становится результатом.

Проблемы:

**a)**Программа должна постоянно перемещать указатель файла и считывать данные по частям. Это требует больше времени, чем работа с файлом целиком в оперативной памяти.

**b)**Пока программа работает, на диске хранятся оба файла: исходный и новый. Если исходный файл большой, это проблема.

* **один файл, два буфера**

Программа инициализирует два раздельных буфера, допустим, по 8 Кб и считывает первые 8 Кб файла в один буфер, а последние 8 Кб – в другой. Далее содержимое обоих буферов обменивается в обратном порядке и первый буфер записывается в конец, а второй – в начало того же файла

На каждой итерации программа перемещает восьмикилобайтовые блоки из одной половины файла в другую. Разумеется, нужно предусмотреть какую-то обработку на случай, если длина файла не кратна 16 Кб, и эта обработка будет куда сложнее, чем в предыдущем методе

Недостатки:

а) Сложен в реализации

б) Может испортить файл данных, если процесс вдруг прервется

* **один файл и никаких буферов**

1. Сначала мы открываем файл. Вместо того чтобы читать его содержимое в буфер, мы используем механизм отображения файла в память.
2. Когда файл открыт, мы просим операционную систему зарезервировать блок виртуального адресного пространства. Это необходимо для того, чтобы система могла "связать" этот регион с файлом. Говоря проще, система выделяет область памяти, в которой файл будет доступен как обычный массив данных.
3. Далее мы сообщаем системе, чтобы она "проецировала" (связала) первый байт файла с первым байтом этого зарезервированного региона памяти. После этого мы можем работать с этим регионом памяти как с обычным массивом данных, хотя это фактически все еще будет содержимое файла.
4. Изменение порядка байтов: После того как файл проецирован в память, мы начинаем изменять порядок байтов. Чтобы это сделать, мы можем использовать два указателя: Один указатель будет указывать на первый байт файла (в памяти). Другой указатель будет указывать на последний байт файла.
5. Мы начинаем с обмена этих байтов, затем сдвигаем указатели к центру, продолжая обмен до тех пор, пока не дойдем до середины файла.
6. После того как мы изменили порядок байтов, операционная система сама позаботится о том, чтобы изменения были записаны обратно в файл. Когда программа завершит работу с файлом, изменения будут автоматически сохранены.

Потенциальный риск потери данных: Хотя это решение эффективно с точки зрения производительности (поскольку данные не нужно вручную загружать и сохранять), оно имеет свои риски. Если в процессе работы программы произойдет сбой (например, из-за отключения питания), изменения могут не успеть быть сохранены в файл, так как они все еще находятся в памяти.

**Как же использовать проецируемые в память файлы?**

Для этого нужно выполнить три операции:

1. Создать или открыть **объект ядра «файл»**, идентифицирующий дисковый файл, который Вы хотите использовать как проецируемый в память. **!!! Стоит отметить что для использования секций на базе страничных файлов первый пункт опускается**
2. Создать **объект ядра «проекция файла»**, чтобы сообщить системе размер файла и способ доступа к нему
3. Указать системе, **как отобразить в адресное пространство Вашего процесса объект «проекция файла»** – целиком или частично

Напомню, что проекция файла = объект секции.

Закончив работу с файлом отображенным в память, следует выполнить тоже три операции:

1. Сообщить системе об отмене отображения на адресное пространство процесса объекта ядра «проекция файла»
2. Закрыть этот объект
3. Закрыть объект ядра «файл» **!!! Стоит отметить что для использования секций на базе страничных файлов третий пункт опускается**

* **Этап 1: создание или открытие объекта ядра «файл»**

Для этого Вы должны применять только функцию [**CreateFile**](https://learn.microsoft.com/en-us/windows/win32/fileio/creating-and-opening-files)

**HANDLE CreateFileA(**

**[in] LPCSTR pszFileName,**

**[in] DWORD dwDesiredAccess,**

**[in] DWORD dwShareMode,**

**[in, optional] LPSECURITY\_ATTRIBUTES lpSecurityAttributes, // null**

**[in] DWORD dwCreationDisposition,**

**[in] DWORD dwFlagsAndAttributes, // null**

**[in, optional] HANDLE hTemplateFile // null**

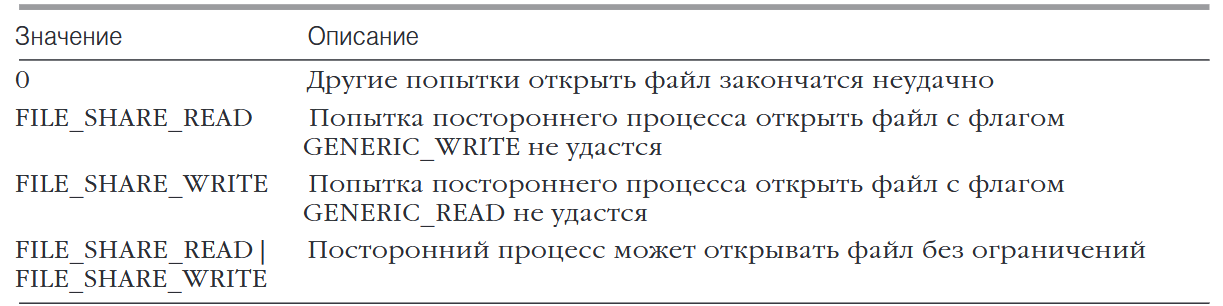
**);**

***pszFileName***, идентифицирует имя создаваемого или открываемого файла (при необходимости вместе с путем)

***dwDesiredAccess***, указывает способ доступа к содержимому файла

* *GENERIC\_READ* (только для чтения)
* комбинированный флаг *GENERIC\_READ* | *GENERIC\_WRITE*
* *только для записи бернацкий не рассматривал*

***dwShareMode***, указывает тип совместного доступа к данному файлу



***lpSecurityAttributes*** Указатель на структуру [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/ru-ru/windows/win32/api/wtypesbase/ns-wtypesbase-security_attributes) = null

***dwCreationDisposition:***

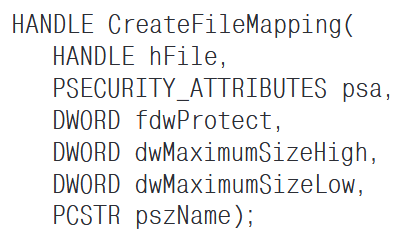
|  |  |
| --- | --- |
| **Ценность** | **Значение** |
| **CREATE\_ALWAYS** | Всегда создает новый файл.  Если указанный файл существует и доступен для записи, функция усечена файла, функция завершается успешно.  Если указанный файл не существует и является допустимым путем, создается новый файл, функция завершается успешно. |
| **CREATE\_NEW** | Создает новый файл, только если он еще не существует.  Если указанный файл существует, функция завершается ошибкой. |
| **OPEN\_ALWAYS** | Открывает файл всегда.  Если указанный файл существует, функция завершается успешно.  Если указанный файл не существует и является допустимым путем к записываемому расположению, функция создает файл. |
| **OPEN\_EXISTING** | Открывает файл или устройство, только если он существует.  Если указанный файл или устройство не существует, функция завершается ошибкой. |
| **TRUNCATE\_EXISTING** | Открывает файл и усечение его таким образом, чтобы его размер был равен нулю байтам, только если он существует.  Если указанный файл не существует, функция завершается ошибкой. |

* **Этап 2: создание объекта ядра «проекция файла»**

**При создании объекта «проекция файла» система не резервирует регион адресного пространства и не связывает его с физической памятью из файла. Это происходит позже, на этапе отображения файла в память**

Вызвав CreateFile, Вы указали операционной системе, где находится физическая память для проекции файла: на жестком диске, в сети, на CD-ROM или в другом месте

Теперь сообщите системе, какой объем физической памяти нужен проекции файла. Для этого вызовите функцию [**CreateFileMapping**](https://learn.microsoft.com/en-us/windows/win32/memory/creating-a-file-mapping-object)



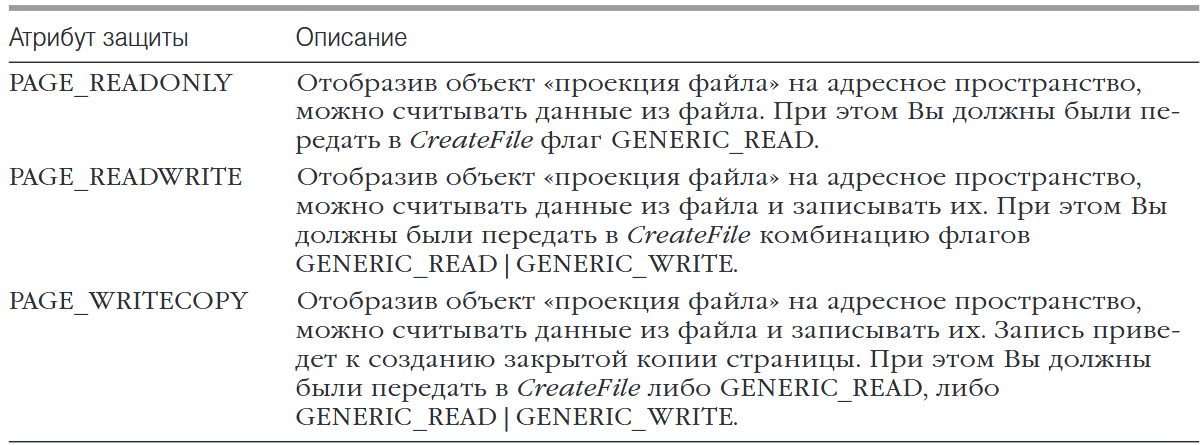
Первый параметр, ***hFile***, идентифицирует описатель файла, проецируемого на адресное пространство процесса. Этот описатель Вы получили после вызова CreateFile

При использовании **секций на базе страничных файлов** первый параметр должен быть установлен в **INVALID\_HANDLE\_VALUE**

Параметр ***psa*** – указатель на структуру SECURITY\_ATTRIBUTES, которая относится к объекту ядра «проекция файла»; для установки защиты по умолчанию ему присваивается NULL

Но, как только дело дойдет до отображения физической памяти на адресное пространство процесса, системе понадобится точно знать атрибут защиты, присваиваемый страницам физической памяти. Поэтому в ***fdwProtect*** надо указать желательные атрибуты защиты (Этот параметр отвечает за **защиту памяти** для отображённой секции)

**Самое главное правило в данном случае, что режимы доступа к объекту «проекция файла» должны соответствовать режимам доступа к файлу, для которого создается этот объект отображения**



Следующие два параметра этой функции (***dwMaximumSizeHigh*** и ***dwMaximumSizeLow***) самые важные. Основное назначение CreateFileMapping **– гарантировать, что объекту «проекция файла» доступен нужный объем физической памяти**. Через эти параметры мы сообщаем системе **максимальный размер файла в байтах**.

Для создания объекта «проекция файла» таким, чтобы он отражал **текущий размер файла**, передайте **в обоих параметрах нули**. Так же следует поступить, если Вы собираетесь ограничиться считыванием или как-то обработать файл, не меняя его размер.

Для дозаписи данных в файл выбирайте его размер максимальным, чтобы оставить пространство «для маневра».

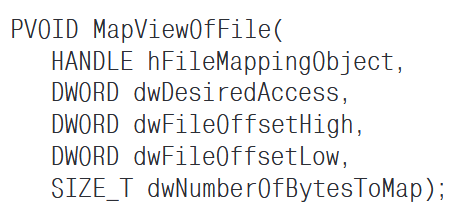
Вызов **CreateFileMapping** с флагом PAGE\_READWRITE заставляет систему проверять, чтобы размер соответствующего файла данных на диске был не меньше, чем указано в параметрах ***dwMaximumSizeHigh*** и ***dwMaximumSizeLow***. Если файл окажется меньше заданного, **CreateFileMapping** **увеличит его размер** до указанной величины.

Последний параметр функции **CreateFileMapping** – ***pszName*** – строка с нулевым байтом в конце; в ней указывается имя объекта «проекция файла», которое используется **для доступа к данному объекту из другого процесса**. Но обычно совместное использование проецируемого в память файла не требуется, и поэтому в данном параметре передают NULL.

* **Этап 3: проецирование файловых данных на адресное пространство процесса**

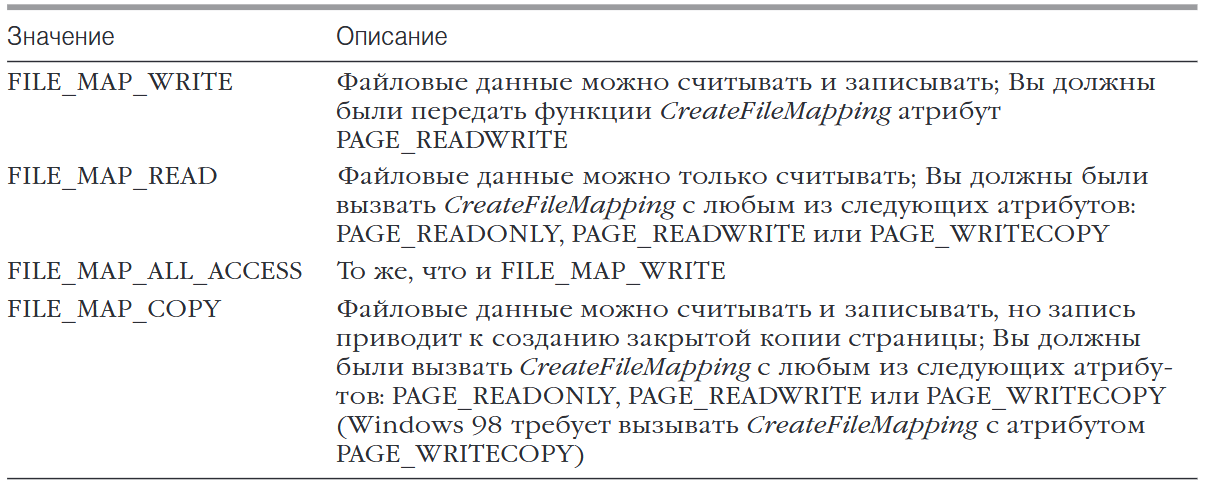
Функция MapViewOfFile отображает представление проецируемого файла в адресное пространство вызывающего процесса.

Когда объект «проекция файла» создан, нужно, чтобы система, зарезервировав регион адресного пространства под данные файла, передала их как физическую память, отображенную на регион. Это делает функция [**MapViewOfFile**](https://learn.microsoft.com/en-us/windows/win32/memory/creating-a-file-view):



Параметр ***hFileMappingObject*** идентифицирует описатель объекта «проекция файла», возвращаемый предшествующим вызовом либо **CreateFileMapping**, либо **OpenFileMapping**.

Параметр ***dwDesiredAccess*** идентифицирует **вид доступа к данным**. Все правильно: придется опять указывать, как именно мы хотим обращаться к файловым данным. Можно задать одно из четырех значений, описанных в следующей таблице



Проецируя на адресное пространство процесса представление файла, нужно сделать две вещи. Во-первых, сообщить системе, какой байт файла данных считать в представлении первым. Для этого предназначены параметры ***dwFileOffsetHigh*** и ***dwFileOffsetLow***

Во-вторых, от Вас потребуется указать размер представления, т. е. сколько байтов файла данных должно быть спроецировано на адресное пространство в параметре ***dwNumberOfBytesToMap***

**Заметьте, что смещение в файле должно быть кратно гранулярности выделения памяти в данной системе (64 КБ)**

* **Гранулярность — это "шаг", с которым система выделяет память.**

**Функция MapViewOfFile возвращает указатель на начало отображённого региона виртуальной памяти.**

* **Этап 4: отключение файла данных от адресного пространства процесса**

Когда необходимость в данных файла (спроецированного на регион адресного пространства процесса) отпадет, освободите регион вызовом [**UnmapViewOfFile**](https://learn.microsoft.com/en-us/windows/win32/memory/closing-a-file-mapping-object):



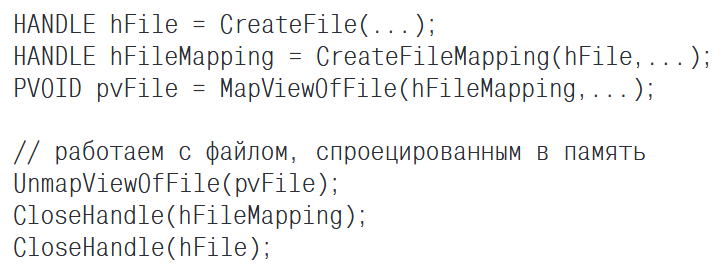
Ее единственный параметр, ***pvBaseAddress***, указывает базовый адрес возвращаемого системе региона. Он должен совпадать со значением, полученным после вызова **MapViewOfFile**. Вы обязаны вызывать функцию **UnmapViewOfFile**. Если Вы не сделаете этого, регион не освободится до завершения Вашего процесса

И еще: повторный вызов **MapViewOfFile** приводит к резервированию нового региона в пределах адресного пространства процесса, **но ранее выделенные регионы не освобождаются**

* **Этапы 5 и 6: закрытие объектов «проекция файла» и «файл»**

Закончив работу с любым открытым Вами объектом ядра, Вы должны его закрыть, иначе в процессе начнется утечка ресурсов. Конечно, по завершении процесса система автоматически закроет объекты, оставленные открытыми. Но, если процесс поработает еще какое-то время, может накопиться слишком много незакрытых описателей.

Для закрытия объектов «проекция файла» и «файл» дважды вызовите **функцию CloseHandle**



В Windows каждый объект (например, файл или проекция файла) имеет **счётчик ссылок** (reference count), который отслеживает, сколько раз этот объект используется. Когда счётчик ссылок достигает нуля, объект автоматически удаляется системой.

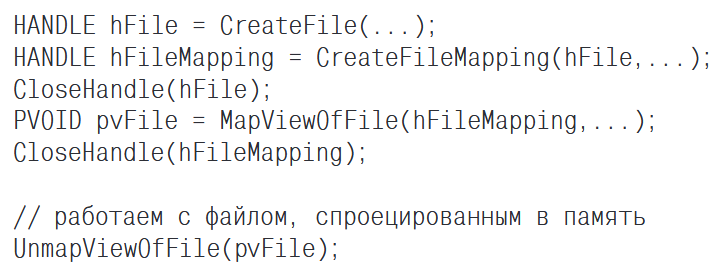
* **Счётчик ссылок увеличивается**, когда:
  + Объект создаётся (например, с помощью CreateFile или CreateFileMapping).
  + Объект используется (например, с помощью MapViewOfFile).
* **Счётчик ссылок уменьшается**, когда:
  + Объект закрывается (например, с помощью CloseHandle).

Когда ты вызываешь MapViewOfFile, система:

1. Увеличивает счётчик ссылок для объекта **"файл"** (созданного с помощью CreateFile).
2. Увеличивает счётчик ссылок для объекта **"проекция файла"** (созданного с помощью CreateFileMapping).

Это означает, что даже если ты закроешь дескрипторы файла и проекции файла с помощью CloseHandle, объекты не будут удалены, пока счётчики ссылок не станут равны нулю.

Этот механизм позволяет **закрыть дескрипторы файла и проекции файла сразу после отображения файла в память**, не дожидаясь завершения работы с отображённой памятью. Это помогает избежать утечек ресурсов.

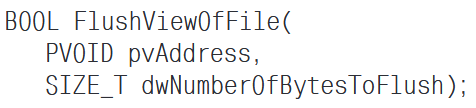


Когда файл отображается в память (с помощью CreateFileMapping и MapViewOfFile), Windows использует **буферизацию** для повышения производительности. Вот как это работает:

* **Данные файла загружаются в оперативную память (ОЗУ)** по мере обращения к ним. Это называется **ленивой загрузкой** (lazy loading).
* Если программа изменяет данные в отображённой памяти, эти изменения **сначала сохраняются в оперативной памяти**, а не сразу записываются на диск.
* Это позволяет системе работать быстрее, так как операции с памятью выполняются намного быстрее, чем операции с диском.

Изменения в отображённой памяти могут быть записаны на диск в следующих случаях:

1. **Автоматически**. Когда система решает, что пора освободить память (например, если памяти не хватает).
2. **Вручную**: Когда программа вызывает функцию **FlushViewOfFile**, чтобы принудительно записать изменения на диск.



Ее первый параметр принимает адрес байта, который содержится в границах представления файла, проецируемого в память. Переданный адрес округляется до значения, кратного размеру страниц

Второй параметр определяет количество байтов, которые надо записать в дисковый образ файла

**Но! Данные в файле так или иначе будут обновлены и без вызова данной функции. Это происходит во время свопинга страниц между памятью и файлом**

Для ОС семейства Linux не присуща дополнительная сложность в виде объектов секций и прочего, **объект с диска отображается в память ядром напрямую**

Следующий момент (схожий с Windows) заключается в том, что отображения могут производиться для двух типов объектов:

* **Обычный файл в файловой системе Linux**

секция файла (или сегмент файла) — это часть файла, которая отображается в память процесса с помощью механизма memory-mapped files

Секция файла разделена на фрагменты размером со страницу, каждый из которых содержит исходное содержимое виртуальной страницы. Из-за **подкачки по требованию** ни одна из этих виртуальных страниц фактически не загружается в физическую память до тех пор, пока центральный процессор сначала не затронет страницу (т.е. не обратиться по виртуальному адресу, который попадает в область адресного пространства этой страницы)

**Если область памяти больше, чем секция файла, то часть которая выходит за рамки размеров файла заполняется нулями (файл при этом системой не расширяется)**

* **Анонимный файл**

**Анонимный файл** — это специальная область памяти, созданная ядром, которая не связана с файлом на диске. Она используется для выделения памяти процессу.

* Когда процесс запрашивает анонимную память, ядро создаёт виртуальные страницы, которые заполнены нулями. Эти страницы называются **demand-zero pages** (страницы с нулями по требованию). Изначально никакая физическая память не выделяется. Виртуальные страницы просто зарезервированы в адресном пространстве процесса.
* Когда процесс впервые обращается к странице (например, записывает данные), происходит следующее:
  1. Ядро находит свободную страницу физической памяти (или выгружает на диск страницу-жертву, если память переполнена).
* Если подробнее, то она может быть clean – данные на диске и в памяти совпадают – тогда она просто удаляется. Иначе он dirty и нужно её выгрузить на диск
  1. Страница-жертва заполняется нулями.
  2. Таблица страниц обновляется, чтобы указать, что страница теперь резидентная (находится в физической памяти).

**Обратите внимание, что на самом деле данные не передаются между диском и памятью**

**Таблица страниц (page table)** — это структура данных, которая используется операционной системой для управления виртуальной памятью. Она связывает виртуальные адреса (которые использует процесс) с физическими адресами (реальными адресами в ОЗУ).

Новые процессы создаются вызовом int pid=**fork()**, который создаёт точную копию вызвавшего его процесса. Пара процессов называются "родительский" и "дочерний" и отличаются друг от друга тремя значениями:

* уникальный идентификатор процесса *PID*
* идентификатор родительского процесса *PPID*
* значение, возвращаемое вызовом fork(). В родительском это *PID* дочернего процесса или ошибка (-1), в дочернем fork() всегда возвращает 0.

Участок памяти в отображении одного процесса можно разделять с отображением другого процесса (**то есть записи таблицы со страницами каждого из процессов будут указывать на одни и те же страницы физической памяти**). В Linux этого можно достичь двумя способами:

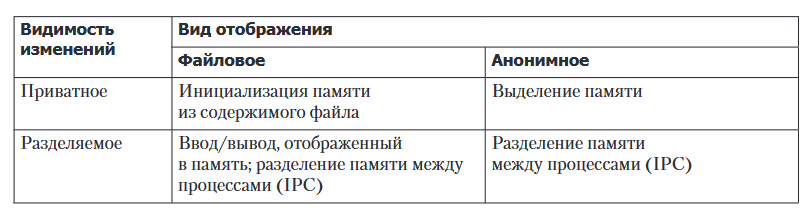
* Когда два процесса **отображают один и тот же участок файла**, они получают доступ к **общим страницам физической памяти**
* Дочерний процесс, созданный с помощью **вызова fork**, наследует копии родительских отображений, указывающих на те же страницы физической памяти, что и отображения родителя

**Разделяемое отображение (MAP\_SHARED): Изменения видны всем процессам.**

* Разделяемое отображение позволяет процессам видеть изменения друг друга.
* Если отображение связано с файлом, изменения также записываются в исходный файл.

**Приватное отображение (MAP\_PRIVATE): Изменения видны только тому процессу, который их внёс.**

* Приватное отображение использует механизм копирования при записи (Copy-On-Write, COW му-му).
* Изначально процессы разделяют одни и те же страницы памяти.
* Когда один из процессов пытается изменить страницу, ядро создаёт копию этой страницы для этого процесса.
* Изменения в копии страницы не видны другим процессам и не записываются в исходный файл.

****

**Приватное файловое отображение.** Основное применение этого вида отображения заключается в инициализации участка памяти, используя содержимое файла. Например, можно инициализировать сегменты процесса, хранящие его код и данные, заполнив их соответствующими участками двоичного исполняемого файла или разделяемой библиотеки.

**Приватное анонимное отображение.** Основное применение приватного анонимного отображения заключается в выделении (и заполнении нулями) памяти для процесса (например, при выделении больших блоков памяти malloc использует вызов mmap)

**Разделяемое файловое отображение. Применение:**

1. Отображение ввода/вывода в память:
   * Файл загружается в память, и все изменения автоматически записываются в файл.
   * Это альтернатива вызовам read и write.
2. Межпроцессное взаимодействие:
   * Неродственные процессы могут использовать общую память для обмена данными.

**Разделяемое анонимное отображение.** Позволяют организовать межпроцессное взаимодействие по тому же принципу, что и в случае с сегментами разделяемой памяти, но только между родственными процессами

**Как же использовать проецируемые в память файлы?**

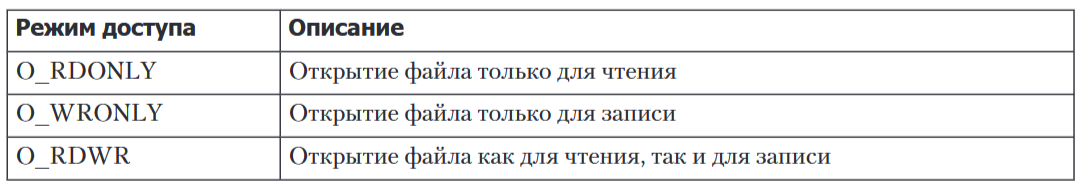
Для этого нужно выполнить две операции:

1. **Получить дескриптор файла (обычно с помощью вызова open)**

**int open(const char \****pathname***, int** *flags***);**

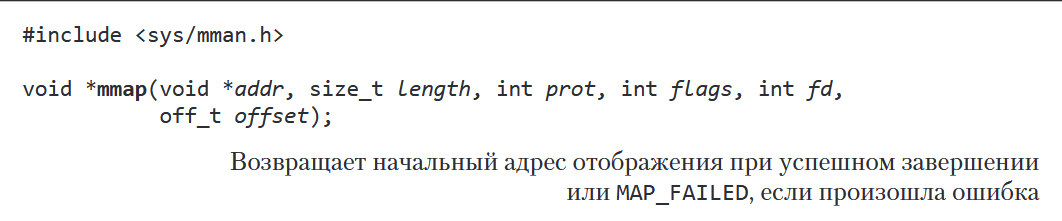
Чтобы файл открылся, он должен пройти идентификацию по аргументу *pathname*. Если в этом аргументе находится символьная ссылка, она разыменовывается. В случае успеха open возвращает дескриптор файла, который используется для ссылки на файл в последующих системных вызовах.

Аргумент ***flags*** является битовой маской, указывающей режим доступа к файлу с использованием одной из констант



1. **Передать этот файловый дескриптор вызову mmap в виде аргумента fd**

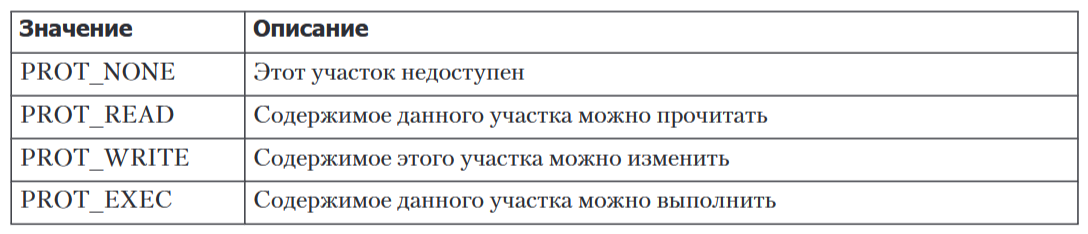
Системный вызов [mmap](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mmap.html) создает новое отображение в виртуальном адресном пространстве вызывающего процесса:

****

Аргумент ***addr*** обозначает виртуальный адрес, по которому будет находиться отображение. Если присвоить ему значение NULL, то ядро само выберет подходящий адрес. Это предпочтительный способ создания отображения.

Аргумент ***length*** обозначает размер отображения в байтах. И хотя он не должен быть кратным размеру страниц памяти в системе, при создании отображения ядро использует единицы измерения, благодаря которым *length* автоматически округляется до ближайшего следующего числа, кратного размеру страницы.

Аргумент ***prot*** представляет собой битовую маску, позволяющую задать защиту отображения. Он может быть равен либо значению PROT\_NONE, либо сочетанию любых других трех флагов (к которым применяется побитовое ИЛИ)

****

Битовая маска ***flags*** состоит из параметров, управляющих различными аспектами работы отображения. В их число обязательно должно входить одно из следующих значений:

* **MAP\_SHARED**
* Изменения в отображённой памяти видны всем процессам, которые разделяют это отображение.
* Если отображение связано с файлом, изменения записываются в файл.
* **MAP\_PRIVATE**
* Изменения в отображённой памяти не видны другим процессам.
* Если отображение связано с файлом, изменения не записываются в файл.

Помимо **MAP\_PRIVATE** и **MAP\_SHARED**, аргумент flags может принимать и другие флаги, разделенные побитовым ИЛИ. Но для нас они пока не представляют интереса, за исключением разве что **MAP\_ANONYMOUS (Отображение не связано с файлом. Это анонимная память, которая заполняется нулями)**

Оставшиеся аргументы, *fd* и *offset*, используются **в сочетании с файловыми отображениями (и игнорируются анонимными)**

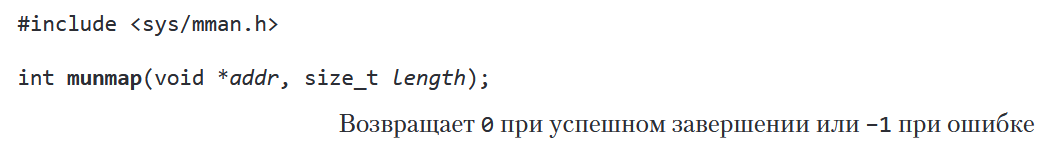
Аргумент ***fd*** – это файловый дескриптор файла, который нужно отобразить

Аргумент ***offset*** задает начальную позицию отображения в файле и должен быть кратным размеру страницы памяти в системе. Чтобы отобразить весь файл целиком, в качестве *offset* можно указать 0, а *length* передать размер файла

**Закончив работу с файлом отображенным в память, следует выполнить тоже две операции:**

1. **Отменить отображение файла**

Системный вызов [**munmap**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/munmap.html)выполняет действие, обратное **mmap**, – удаляет отображение из виртуального адресного пространства вызывающего процесса:



Аргумент ***addr*** обозначает начальный адрес участка памяти, с которого нужно удалить отображение. Он должен совпадать с началом страницы

Аргумент ***length*** представляет собой неотрицательное целое число, обозначающее размер участка, с которого будет удаляться отображение (в байтах). Диапазон адресов будет округлен до следующего значения, кратного размеру страницы в системе

Обычно отображение удаляется целиком. Следовательно, нужно указать адрес, возвращенный предыдущим вызовом **mmap**, и использовать то же значение ***length***, что и при создании отображения

Но можно также выполнить частичное удаление. В этом случае отображение либо урезается, либо делится на две части – в зависимости от места выполнения удаления

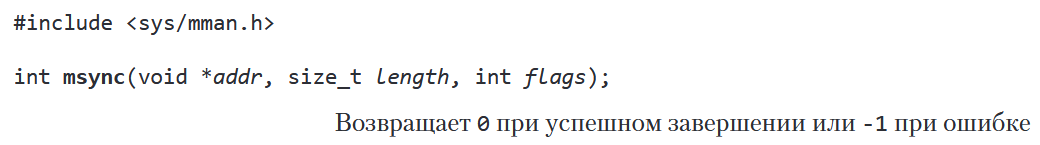
Кроме того, существует возможность указать диапазон адресов, охватывающий несколько отображений, в результате чего все они будут удалены

1. **Закрыть объект ядра «файл»**

Для закрытия объекта «файл» вызовите функцию [**close**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/close.html)

Отображения типа MAP\_SHARED обратно в исходный файл, но по умолчанию нет никаких гарантий относительно того, когда именно произойдет такая синхронизация. Системный вызов [**msync**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/msync.html) предоставляет приложению возможность самостоятельно **синхронизировать разделяемое отображение с отображенным файлом**. Выполняя данный вызов, мы делаем обновления изменяемого отображения видимыми для других процессов, которые считывают исходный файл

Аргументы ***addr*** и ***length*** вызова **msync** обозначают начальный адрес и размер участка памяти, который нужно синхронизировать. Адрес должен быть выровнен по странице, а длина округлена до следующего кратного размеру страницы в системе



Аргумент ***flags*** может принимать одно из следующих значений:

* MS\_SYNC – выполняет синхронную запись в файл
* MS\_ASYNC — выполняет асинхронную запись в файл

**Когда использовать File Mapping:**

* **Когда вы работаете с большими файлами**
* **Когда требуется произвольный доступ**
* **Когда производительность критична**
* **Когда память ограничена**

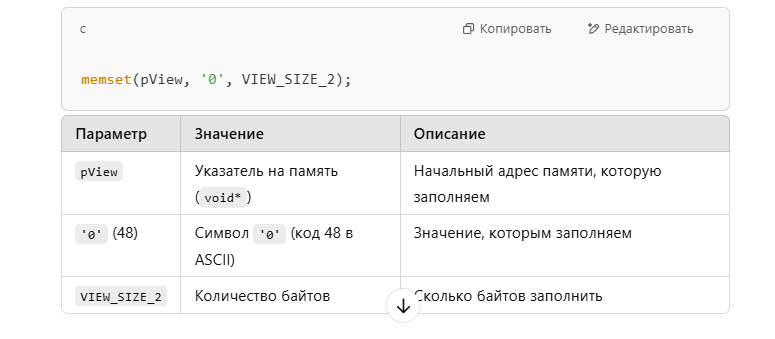
**Когда не использовать File Mapping:**

* **Когда вы работаете с маленькими файлами**
* **Когда требуется последовательный доступ**
* **Частые изменения файла**

**size\_t fwrite(const void \*ptr, size\_t size, size\_t count, FILE \*stream);**

* ptr: const void \*ptr - это адрес в памяти, с которого функция fwrite начнет читать данные для записи. Важно, что данные, на которые указывает этот указатель, не должны изменяться функцией fwrite, отсюда и const.
* size: size\_t size - размер в байтах каждого элемента данных, который нужно записать.
* count: size\_t count - количество элементов (размером size каждый), которые нужно записать.
* stream: FILE \*stream - указатель на поток вывода, в который нужно записать данные.
  + stdout: Стандартный поток вывода (обычно консоль).
  + stderr: Стандартный поток ошибок (обычно консоль).

**Функция memset() заполняет заданную область памяти определенным значением.**

****

**memcpy()** Это стандартная библиотечная функция (из <string.h> в C или <cstring> в C++), предназначенная для копирования *n* байт из области памяти, на которую указывает *источник*, в область памяти, на которую указывает *назначение*.