7 Управление памятью

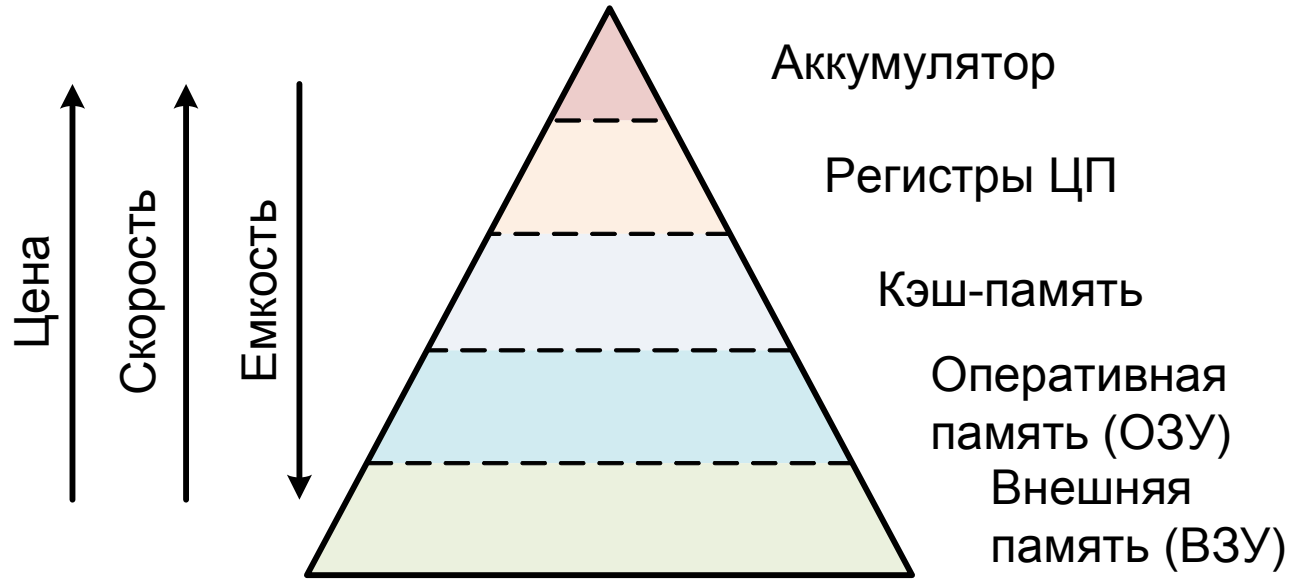
7.1 Подсистема памяти

Память – важнейший (после процессорного времени) ресурс: обеспечивает вычислительной системе хранение программ и данных.

В традиционных архитектурах основная память – адресная, представляет собой массив адресуемых ячеек (***слов***). Разрядность ячеек и их адресов (потенциальный объем адресуемой памяти) – важнейшие количественные характеристики вычислительной системы.

Адресное пространство – все множество доступных адресов (не обязательно реально заполненных) ячеек.

Организация подсистемы памяти – иерархическая: сочетание различных видов физической памяти с различными (даже противоположными) характеристиками):



Иерархия видов памяти

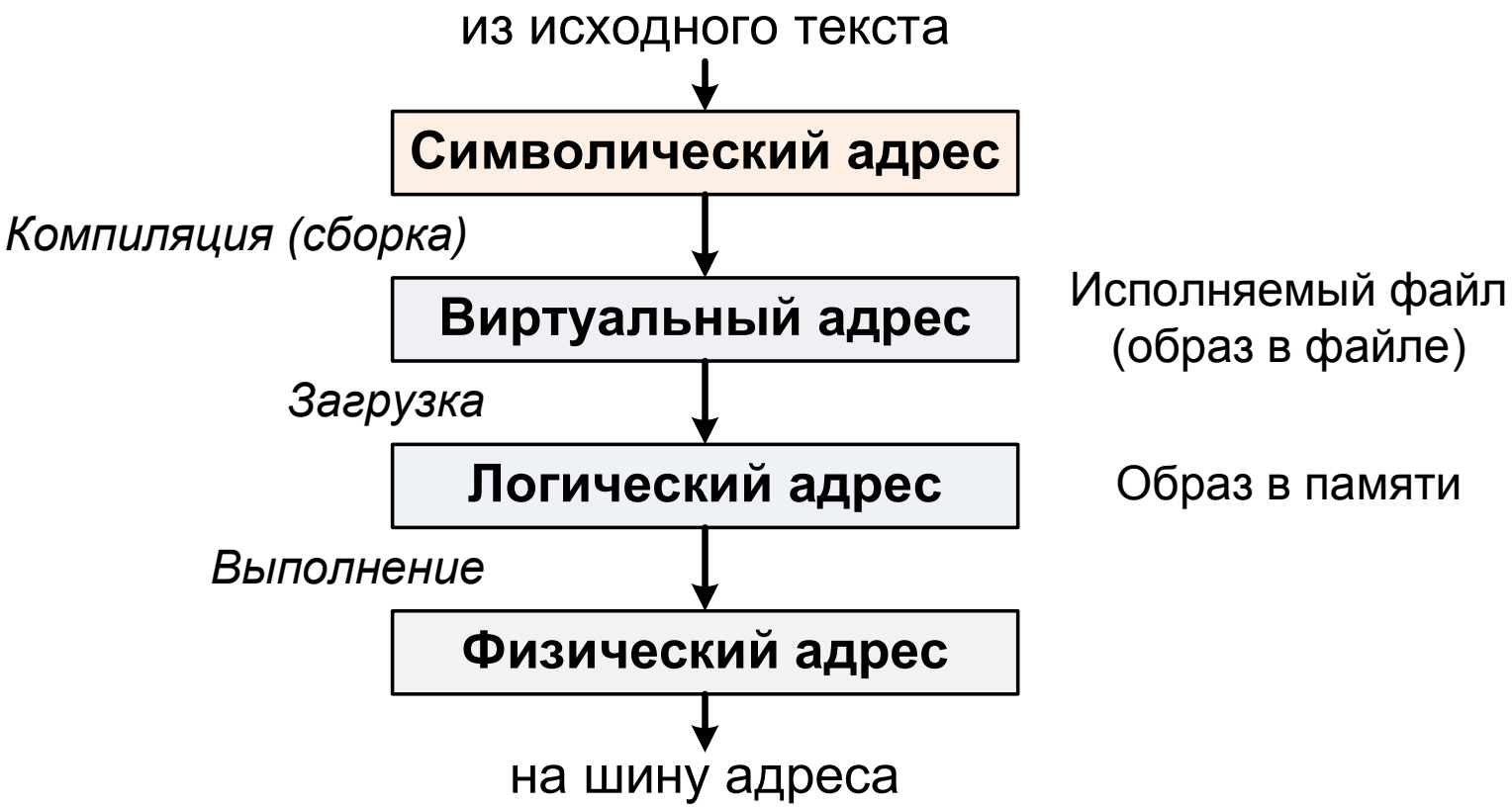
Уровни (этапы жизненного цикла) адресов:
– ***символические*** – имена, присвоенные в программе адресуемым объектам

– ***виртуальные*** – соответствуют по формату логическим/физическим, но отсчитываются в абстрактном виртуальном адресном пространстве

– ***логические*** – соответствующие программной модели процессора

– ***физические*** – реально выставляются на шину данных

Преобразование виртуальных адресов в физические – может быть ***ранним*** (при загрузке образа программы из файла в память) или ***поздним*** (в процессе работы программы, при обращениях к адресам.



Уровни (стадии жизненного цикла) адресов

7.2 Функции подсистемы памяти

– Выделение потребителям блоков памяти

– Освобождение неиспользуемой памяти

– Учет выделенных и свободных блоков памяти

– Организация виртуального адресного пространства и преобразование адресов

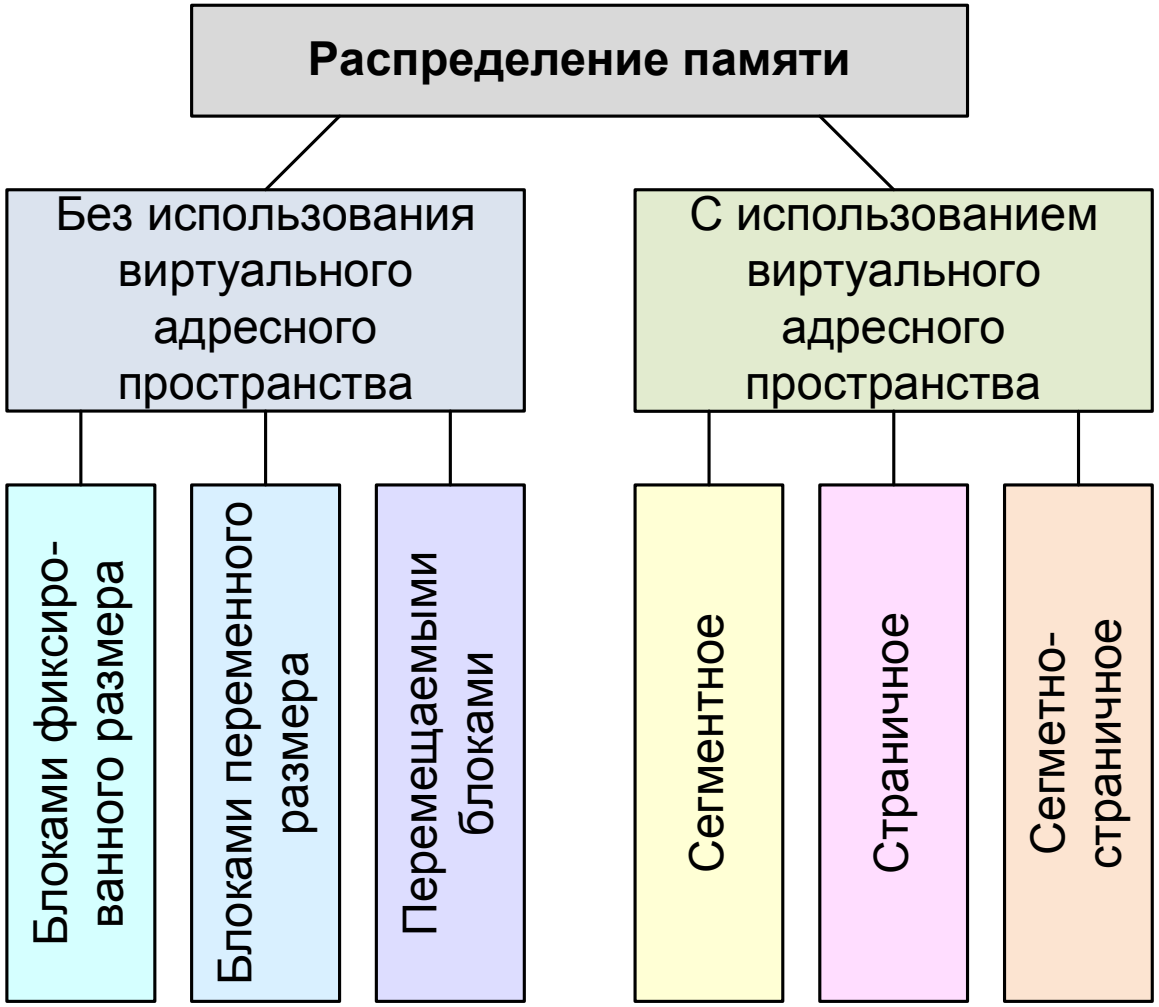
– Контроль прав доступа к областям памяти

– Оптимизация выделения памяти, в т.ч. перемещение блоков

Компонент ОС, отвечающий за распределение памяти – ***диспетчер*** или ***менеджер*** памяти. Поскольку современные ОС используют концепцию ***виртуальной*** памяти, это отражается и в названии ответственного за управление ею компонента. Например, в Windows – ***VMM***, Virtual Memory Manager.

Виртуальная память – абстракция, эмуляция наличия адресуемой памяти заданного объема на имеющихся аппаратных средствах. Как правило, подразумевает перемещение данных между оперативной памятью и внешними устройствами хранения.

7.3 Методы распределения памяти



Группы методов распределения памяти

7.3.1 Методы без использования виртуального адресного пространства

Простая логика, не требуется специальная аппаратная поддержка трансляции адресов.

1) Блоки (разделы) ***фиксированного*** размера.
Для описания блока достаточно только его начального адреса и даже порядкового номера, что предельно упрощает задачу менеджера памяти.

Проблема – неэффективное использование памяти, отчасти решается допущением нескольких фиксированных размеров (например «большие» и «малые» блоки).

2) Блоки ***переменного*** размера, обычно в виде списка.

Описание блока должно включать начальный адрес и размер блока (или начало следующего блока).

Проблема – фрагментация свободного пространства, невозможность объединить свободные фрагменты.

Пример реализации – MS DOS, MCB-блоки.

3) ***Перемещаемые*** блоки.

Позволяют перемещение в физической памяти уже выделенных блоков, дефрагментация свободного пространства.

Проблема – непредсказуемые изменения фактических адресов уже выделенных блоков, требующая повторной настройки.
Решение – жесткие ограничения на использование адресов либо переход к виртуальному адресному пространству.

7.3.2 Методы с использованием виртуального адресного пространства

Сегмент – непрерывная область памяти, характеризующаяся единым владельцем, способом использования, правами доступа. Описывается начальным адресом и размером.

Страница – фрагмент памяти фиксированного размера, минимальная порция, которой оперируют аппаратные механизмы процессора (но не минимальная адресуемая ячейка!). Исключение размера из описания страницы упрощает составление каталога страниц, но хранение других характеристик сильно увеличивает объем служебных данных.

Размер страниц – аппаратно-зависимый, одна из характеристик аппаратной платформы (обычно в пределах единиц килобайт), для архитектуры x86 это 4 кбайта (стандартно, не учитывая расширений).

В любом случае, при использовании механизмов виртуальной памяти актуально также текущее состояние: присутствие в оперативной памяти или выгрузка во внешнюю.

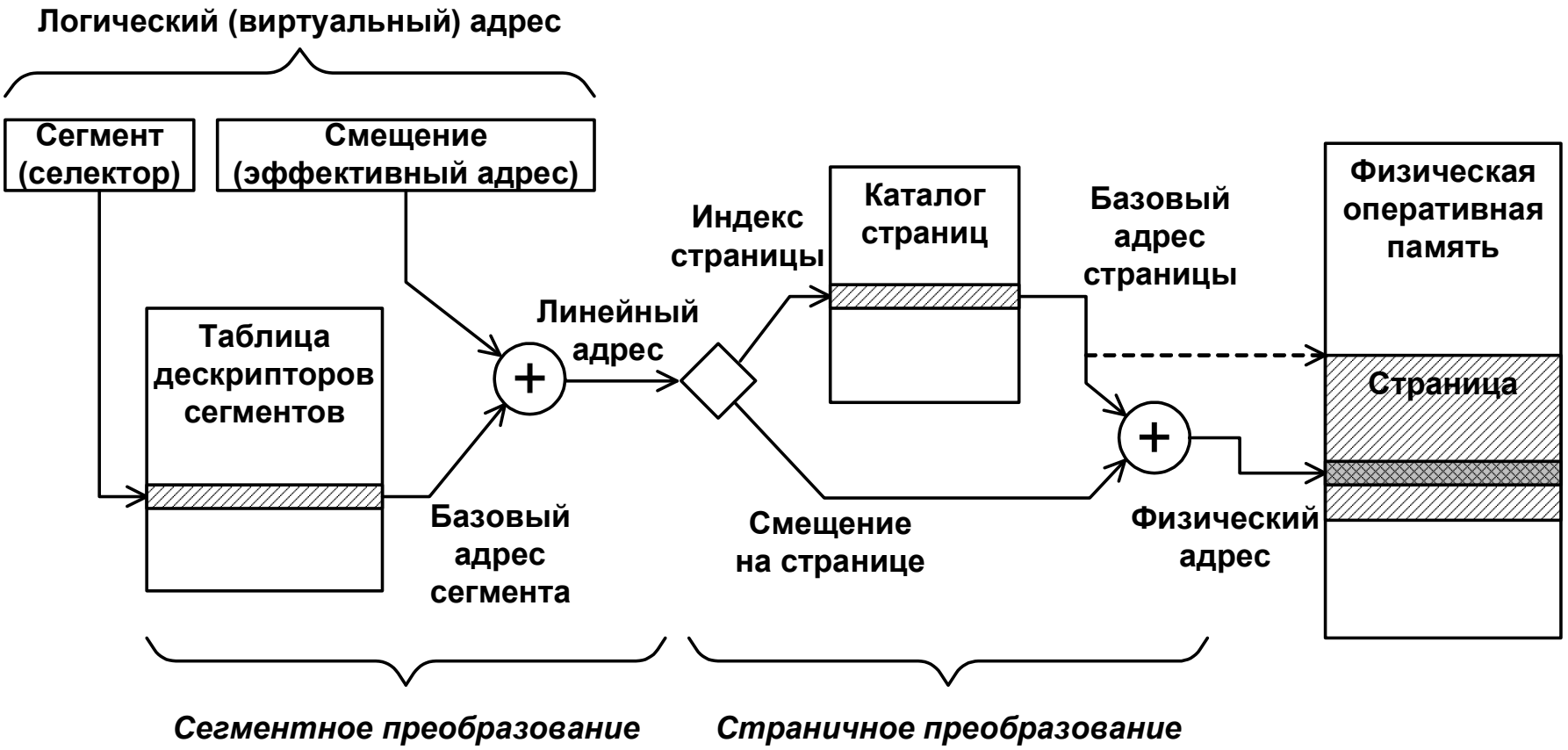
При ***сегментно-страничном*** распределении происходит разделение «обязанностей»:

– сегменты – общий размер, принадлежность, параметры использования;

– страницы – текущее состояние, участие в перемещении.

В таком случае сегмент представляет собой список (каталог) страниц, входящих в него.
Адресное пространство процесса – таблица сегментов, каталог (или каталоги) страниц, принадлежащих этому процессу.

В упрощенном виде:



Упрощенная схема формирования адреса
при сегментно-страничном распределении памяти

Чаще реализуется «плоская» модель памяти – наложение всех сегментов, манипулирование только смещением в сегменте.

Трансляция адреса – использование аппаратной поддержки ЦП (например, механизмы защищенного режима ЦП Intel i286/386+)

Подкачка страниц – ***своппинг*** (***swapping***). Страничный файл (page-file) или файл подкачки (swap-file).

Отображение (***mapping***) файлов в память (см. ниже) – фактически продолжение механизма подкачки, но вместо общего страничного файла используется произвольный явно заданный файл.

Менеджер виртуальной памяти должен принимать решения о выборе перемещаемых (вытесняемых из физической оперативной памяти во внешнюю) тех или иных страниц. ***Стратегия*** выбора может основываться на различных характеристиках страниц, например на «возрасте» страницы (времени последнего обращения к ней), частоте обращений и т.д. Неоптимальный выбор приводит к снижению эффективности подсистемы памяти, например к «пробуксовке» при своппинге – неоправданному вытеснению страниц, которые спустя короткое время приходится загружать заново.

Кроме того, по-разному могут быть выбраны и адреса для выделения новых областей памяти, например:

– наименьший из подходящих по размеру – минимизация неиспользуемого «остатка»

– наибольший из доступных – больший остаток более полезен в дальнейшем

Для систем ***реального времени*** с жесткими ограничениями на время реакции использование своппинга может оказаться неприемлемым из-за непредсказуемых задержек на перемещение и подгрузку страниц.

7.4 «Кучи» (Heap)

«Куча» (Heap) – зарезервированная область памяти в адресном пространстве пользователя. Как правило, работа с «кучей» идет посредством функций, реализованных в библиотеках уровня пользователя, что ускоряет обращения а ним.

Heap для прикладного процесса и для VMM

Полезный эффект от применения «кучи»:

– Упорядочивание выделения памяти, возможность не смешивать в одной структуре выделяемые объекты, резко различающиеся по прочим признакам

– Предотвращение утечек памяти: «кучу» можно освободить целиком, например при завершении использовавшего ее потока

– Оптимизация алгоритмов распределения под типичные характеристики выделяемых блоков

– Меньшие «накладные расходы» ввиду реализации функций на уровне пользователя

7.5 API подсистемы памяти Windows

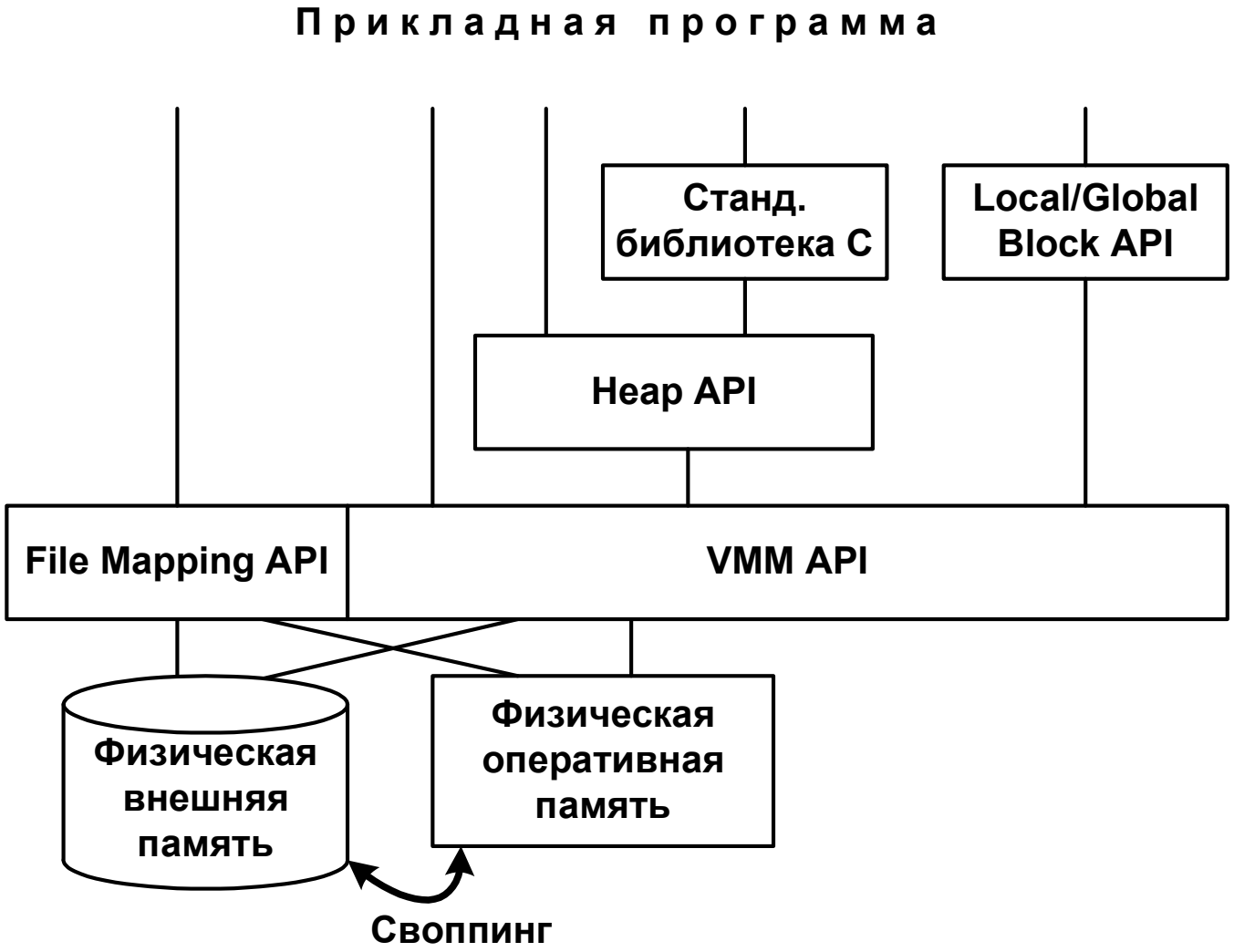
– ***VMM*** API – нижний уровень, взаимодействие непосредственно с Virtual Memory Manager

– ***Heap*** API – создание «куч» (как объектов) и выделение/освобождение фрагментов в ней

– ***File Mapping*** API – отображение файлов в память

– ***Global/Local Memory Block*** API – работа с объектами «блок памяти»; считается устаревшим и поддерживается по соображениям совместимости

– Стандартная библиотека ***C Runtime Library***



Иерархия API подсистемы памяти (Windows)

7.5.1 VMM API

Нижний базовый уровень, доступный для прикладных программ. Основная единица для функций этой группы – регион.

Регион – непрерывный отрезок адресов. Регион может находиться в состояниях:
– свободен – не заполнен ничем
– зарезервирован – содержимое пока не существует, но соответствующие адреса уже недоступны для иного выделения
– «подключен» (commited) – регион заполнен страницами.

Любая из страниц региона в состоянии commited может находиться в состояниях:
– загружена в оперативную память
– еще не выделена физически
– выгружена в файл подкачки.

Во всех случаях, когда страница в данный момент недоступна, происходит исключение, и система старается исправить проблему (например, подгрузить страницу из файла подкачки.

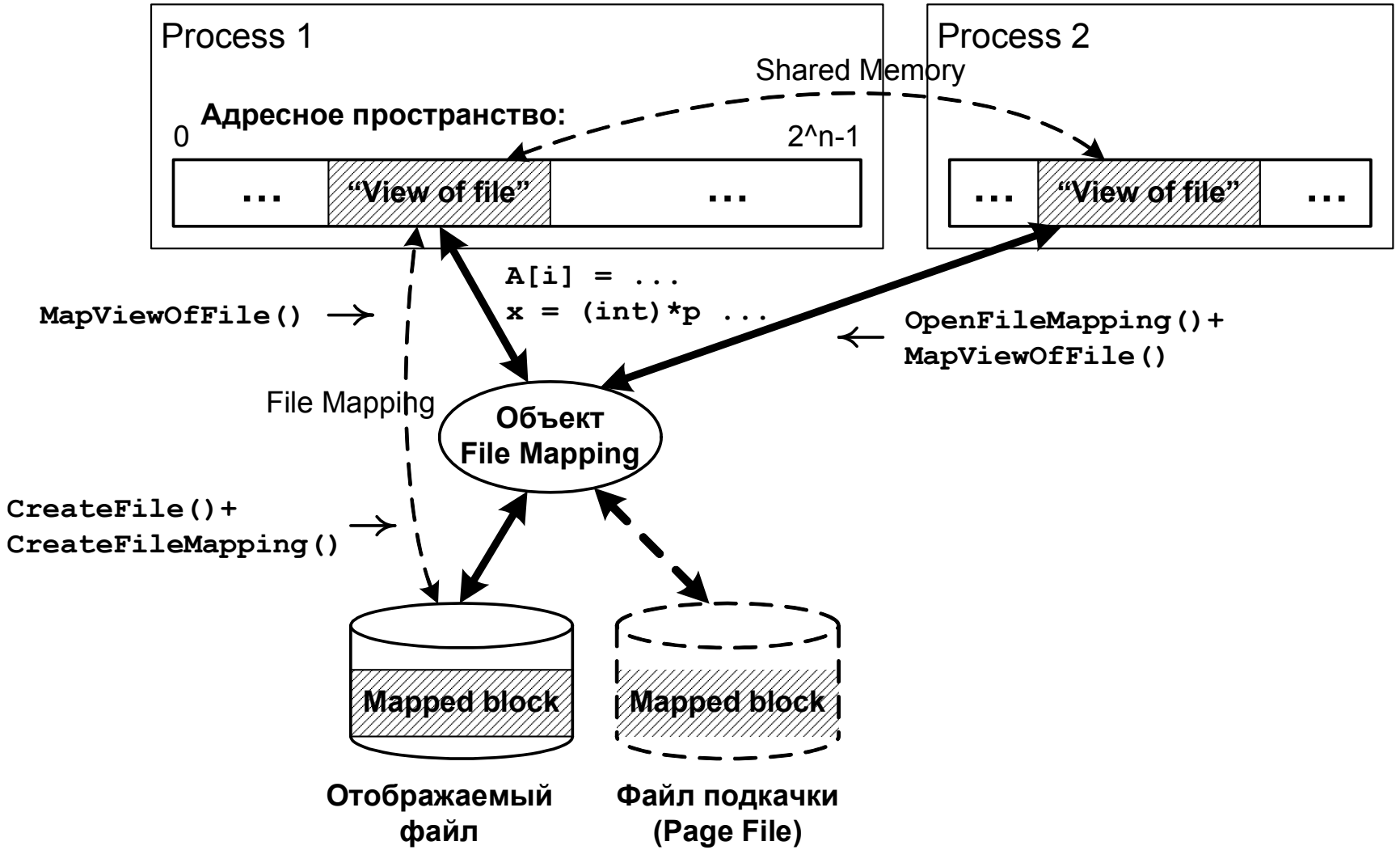
Регионы не являются объектами, не имеют «описателей» и идентифицируются адресом начала. Регионы дробятся и объединяются: например, сделав «подключенной» часть «зарезервированного» региона, в результате получим три региона. Для регионов существенно сказывается гранулярность памяти: регионы начинаются с «круглых» адресов, кратных 64 К, и их размеры кратны размеру страницы.

Функции:
`VirtualAlloc()`, **`VirtualFree()`**
`VirtualQuery()`, **`VirtualProtect()`**
и др.

Функции VMM API обеспечивают ряд полезных (и даже незаменимых) дополнительных возможностей, но при интенсивных обращениях неудобны и/или неэффективны.

7.5.2 File Mapping API

Выполняется в две стадии: создание объекта **`FileMapping`** и отображение (проецирование) его на адресное пространство процесса. После проецирования образуется регион адресов памяти, для которого страницы выделены, а их содержимое загружено из файла. Следовательно, чтение отображения равносильно чтению из файла, запись в отображение – записи в файл.



Отображение файла в память (File Mapping)

Создание объекта-отображения или получение доступа к уже существующему:

```
HANDLE CreateFileMapping()
HANDLE CreateFileMappingNuma()
HANDLE OpenFileMapping()
```

При создании объекта-отображения файл уже должен быть открыт (в функцию передается его Handle). Если он не указан, будет использоваться часть файла подкачки. Параметрами выбирается отображаемая часть файла. После выполнения функции файл можно закрыть.
Нежелательно продолжать использовать открытый файл для обычного ввода-вывода – при этом возможны конфликты, а когерентность считываемых и записываемых данных не гарантируется.

Проецирование содержимого объекта-отображения на виртуальное адресное пространство и «отключение» такой проекции.

```
void* MapViewOfFile()  void* MapViewOfFileEx()
void* MapViewOfFileExNuma()
void UnmapViewOfFile()
```

После проецирования файла выделенная для него область памяти идентифицируется обычным адресом в виртуальном адресном пространстве процесса, и доступ к содержимому осуществляется по обычным указателям, как к массиву или к структурированным данным.
Реальная загрузка данных из файла в страницы памяти происходит по мере обращений к ячейкам. Выгрузка изменившегося содержимого страниц в файл выполняется автоматически (с некоторой задержкой).

Основные варианты использования файловых отображений:

– Работа с файлами данных прямого доступа

– Организация разделяемой памяти. Для этого необходимо отобразить один и тот же участок одного и того же файла в адресные пространства двух или более процессов. Важно, что это должно быть проецирование одного и того же объекта **`FileMapping`**, в этом случае процессы будут работать с физически одними и теми же страницами. При независимом проецировании файла через различные объекты **`FileMapping`** когерентность содержимого отображений не гарантируется: данные становятся видны партнеру только после прохождения их через файл.

– Исполнение программ, включая присоединение динамических библиотек (***DLL***): образ программы (библиотеки) из файла проецируется в память, после чего на него можно передавать управление. При этом достигается также ускорение запуска и оптимизация расхода физической памяти: начинать выполнять можно даже не полностью загруженный образ.

– Нетривиальные задачи, например эмуляция адресного пространства устройства, используя заготовленный файл.

7.5.3 Heap API

«Куча» (Heap) – системный объект, идентифицируется HANDLE. Процесс обязательно имеет «кучу по умолчанию» (основную), но дополнительно может создавать произвольное количество других «куч».

Создание новой «кучи» (Задаются размеры и некоторые другие характеристики):

```
HANDLE HeapCreate( dwOptions,
  dwInitialSize, dwMaximumSize
);
```

Использование нескольких «куч» бывает целесообразно для:

– более надежного разделения структур данных

– повышения эффективности управления памятью, сокращения потерь из-за фрагментации

– снижения вероятности «утечки» памяти (отдельные «кучи» для потоков)

Получение доступа к имеющимся у процесса «кучам»:

`HANDLE GetProcessHeap()` – *«куча» по умолчанию*

`DWORD GetProcessHeaps()` – *список всех «куч»*

Проверка состояния (целосности) «кучи» в целом или отдельного блока в ней:

`BOOL HeapValidate( hHeap, dwFlags, pMem)`

Удаление «кучи» (за исключением «кучи» по умолчанию!):

`BOOL HeapDestroy( hHeap)`

Выделение, освобождение, перераспределение блока памяти в «куче»:

```
void* HeapAlloc( hHeap, dwFlags, dwBytes)
void* HeapReAlloc( hHeap, dwFlags, pMem, dwBytes)
BOOL HeapFree( hHeap, dwFlags, pMem)
```

Размер блока в «куче»:

```
SIZE_T HeapSize( hHeap, dwFlags, pMem)
```

Обращения к «кучам» в общем случае не потокобезопасны. В многопоточном приложении потокам следует либо ограничиваться «кучей» по умолчанию, либо использовать локальную «кучу», созданную и контролируемую этим же потоком, либо строить сложную схему синхронизации доступа. В противном случае возможны коллизии.

7.5.4 Стандартная библиотека

Функции стандартной библиотеки C:

```
void* malloc()
void* calloc()
void* realloc()
void free()
```

Документация MSDN объявляет их «устаревшими», однако функции ценны своей переносимостью.

Функции используют общую «кучу» процесса.

7.5.5 Local/Global Memory Blocks

Объекты «блок памяти» продолжают поддерживаться по соображениям совместимости. На практике используются редко, одно из исключений – буфер обмена (clipboard).

Работа с блоками происходит в два этапа (отчасти похоже на отображение файла):

– создание объекта «блок памяти», включая выделение ему соответствующего ресурса

– «закрепление» объекта – привязка его к интервалу адресов в адресном пространстве

После привязки (фактически отображения на адресное пространство) к содержимому блока можно обращаться по указателям.

HGLOBAL GlobalAlloc(unsigned *uFlags*, SIZE_T *dwBytes*);

HGLOBAL GlobalFree(HGLOBAL *hMem*); – *при успешном выполнении возвращает NULL*

HGLOBAL GlobalReAlloc(HGLOBAL *hMem*, SIZE_T *dwBytes*, unsigned *uFlags*);

void* GlobalLock(HGLOBAL *hMem*);

BOOL GlobalUnlock(HGLOBAL *hMem*);