ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 10 — Сегменты общей памяти

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.04.11

Оглавление

Механизмы межпроцессного взаимодействия

ОС UNIX поддерживает три типа средств межпроцессной связи (IPC):
- очереди сообщений;
- наборы семафоров;
- совместно используемые сегменты памяти.

Существует две версии общей памяти:
- System V;
- POSIX.

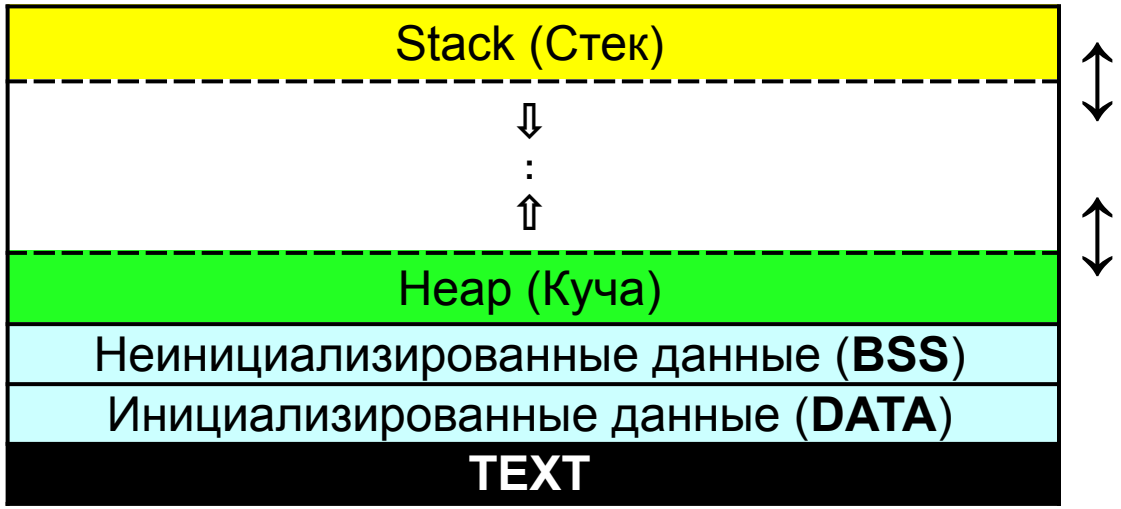
Память компьютерной программы

Память компьютерной программы может быть разделена в широком смысле на две части:
- только чтение (RO);
- чтение-запись (RW).

Ранние ЭВМ держали свою основную программу в постоянной памяти, такой как ROM (ПЗУ), PROM (ППЗУ), EPROM (ППППЗУ).

По мере усложнения систем и загрузки программ из других носителей в ОЗУ вместо выполнения их из ПЗУ, сохранялась идея о том, что **некоторые части памяти программы не должны изменяться**. Эти части стали программными сегментами (секциями) **`.text`** и **`.rodata`**, а остальная часть, которая может перезаписываться, разделилась на несколько других сегментов в зависимости от конкретной задачи.

```
ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc
```



Инициализированные данные (Data)

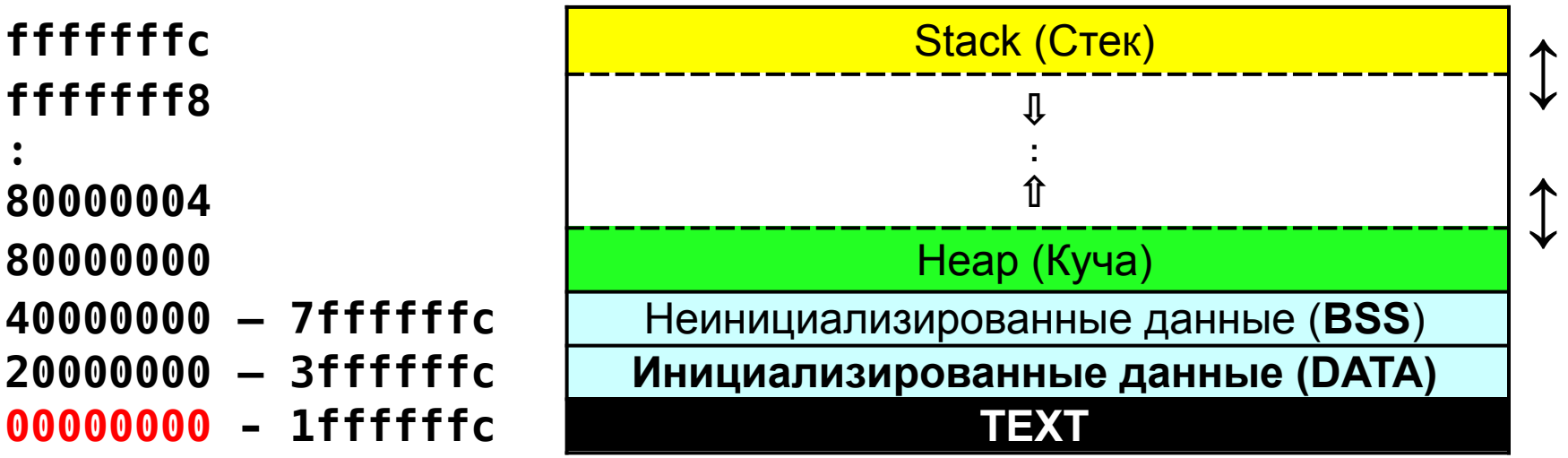
Сегмент **.data** содержит любые ***глобальные*** или ***статические*** переменные, которые имеют предопределенное значение и могут изменяться.

Это любые переменные, которые не определены внутри функций (и, следовательно могут быть доступны из любого места программы) или определены в функции, но определены как статические и сохраняют свое расположение (адрес) при последующих вызовах.

Пример на языке С:

```
int  val = 3;
char string[] = "Hello World";
```

Значения этих переменных первоначально сохраняются в постоянной памяти (обычно в **.text**) и копируются в сегмент **.data** во время процедуры запуска программы.



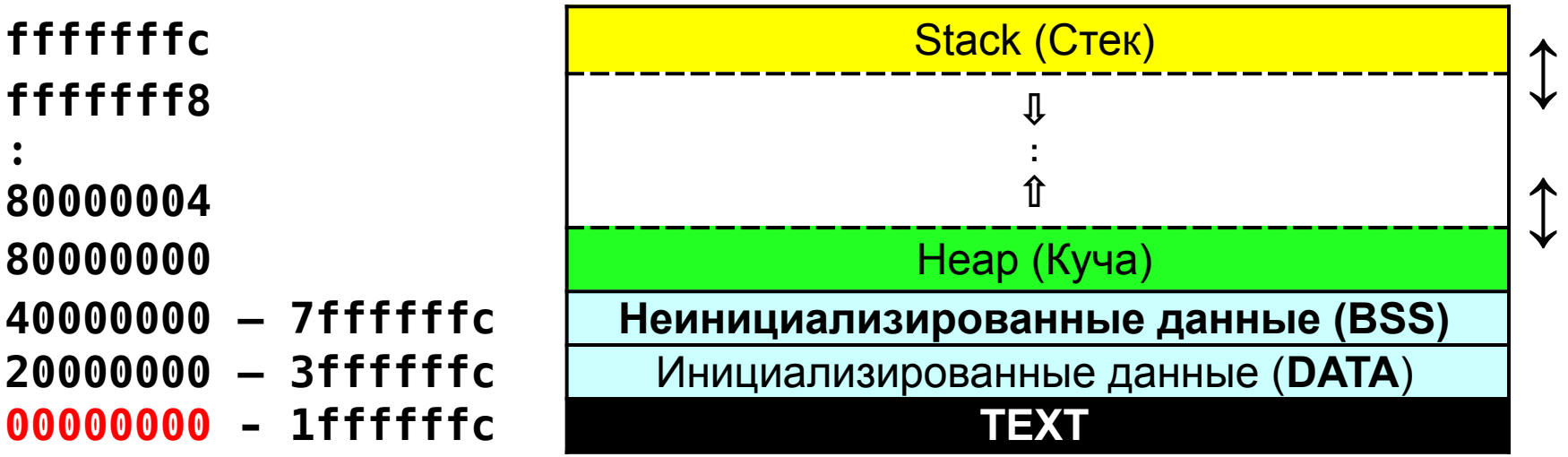
Неинициализированные данные (BSS — Block Started by Symbol)

Сегмент BSS, известный как неинициализированные данные, содержит все ***глобальные*** переменные и ***статические*** переменные, которые инициализируются нулем или не имеют явной инициализации в исходном коде. Обычно примыкает к сегменту данных.

Например, переменная, определенная как

```
static int i;
```

будет содержаться в сегменте BSS.



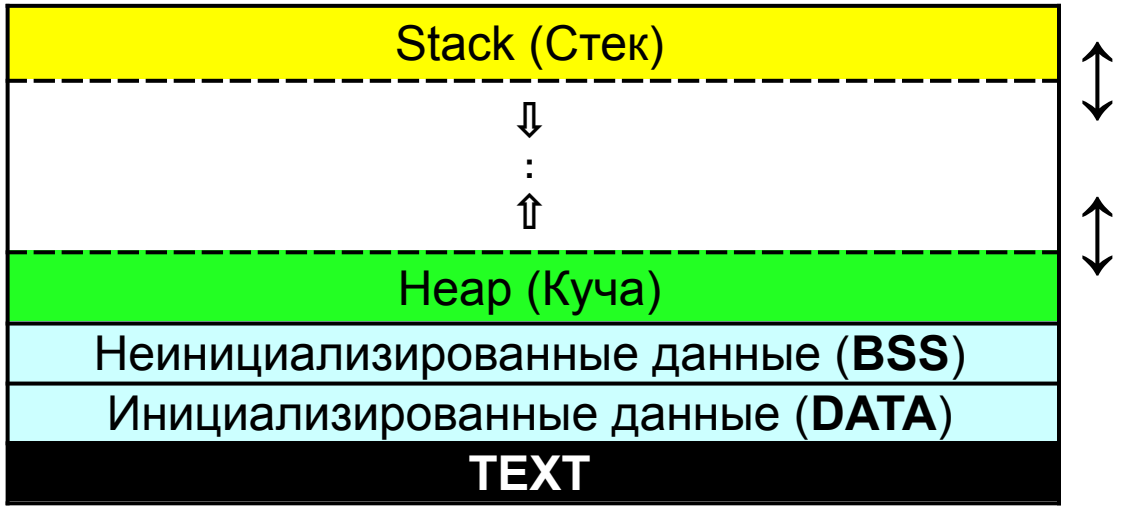
Куча (Heap)

Область кучи обычно начинается в конце сегментов **.bss** и **.data** и растет в сторону увеличения адресов. Область кучи из языка С управляется с помощью функций стандартной библиотеки

```
malloc();
calloc();
realloc();
free().
```

которые используют ***системные вызовы*** для настройки своего размера.

ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc



Стек (Stack)

Область стека содержит стек программы — структуру LIFO, обычно расположенную в верхних частях памяти.

Вершину стека отслеживает регистр «указателя стека» . Она корректируется каждый раз, когда значение «заталкивается» в стек.

Набор значений, выделяемых в стеке для одного вызова функции, называется «стековым фреймом/кадром».

Кадр стека состоит как минимум из адреса возврата.

В стеке также выделяются автоматические переменные

Область стека всегда традиционно примыкала к области кучи, и они росли навстречу друг другу. Когда указатель стека встречал указатель кучи, свободная память исчерпывалась.

При наличии большого адресного пространства и виртуальной памяти они, как правило, размещаются более свободно, но по-прежнему обычно растут в сходящемся направлении. На стандартной архитектуре x86 стек растет вниз (по направлению к нулевому адресу), что означает, что более свежие элементы, более глубокие в цепочке вызовов, находятся в более нижних адресах и ближе к куче.

На некоторых других архитектурах стек растет в обратном направлении.

Традиционная организация памяти компьютерной программы

Программный код загружается, начиная с адреса, немного превышающего 0, поскольку указатель со значением **NULL** никуда не указывает. Секция данных начинается сразу за секцией кода и включает все секции и подсекции с данными — DATA, BSS, и прочие, если присутствуют.

Программа для выполнения считывается в память, где и остается вплоть до своего завершения. Поэтому утверждение о том, что размер вашего двоичного файла не влияет на использование памяти, НЕВЕРНО.

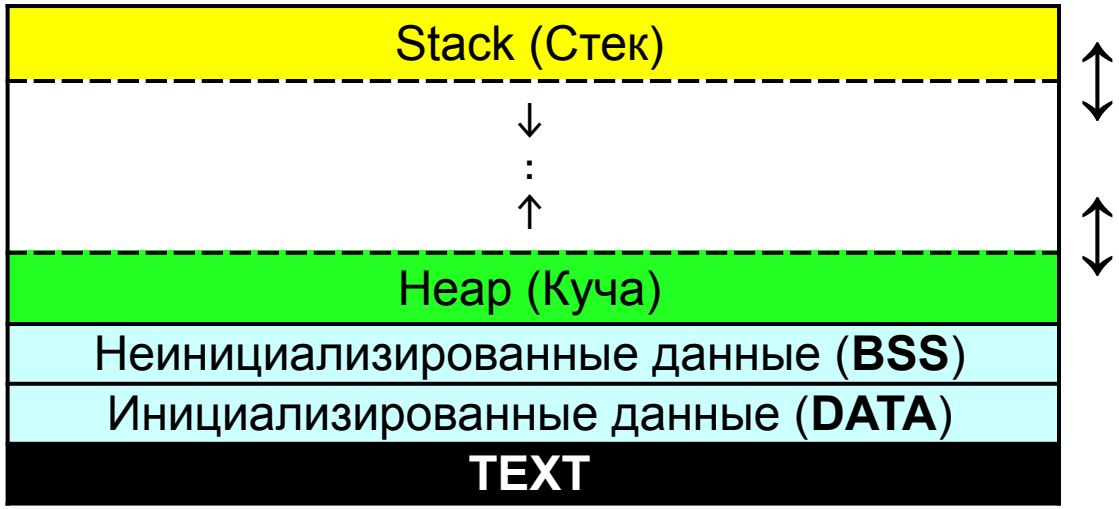
Куча располагается сразу за секцией данных.

При загрузке программы размер кучи нулевой

Стек располагается по верхним адресам.

Размер стека устанавливается на этапе загрузки. Стек растет вниз, куча — вверх. Первая же операция с кучей **malloc()** вызывают движение границы секции данных (пунктирный маркер) вверх.

```
ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc
```



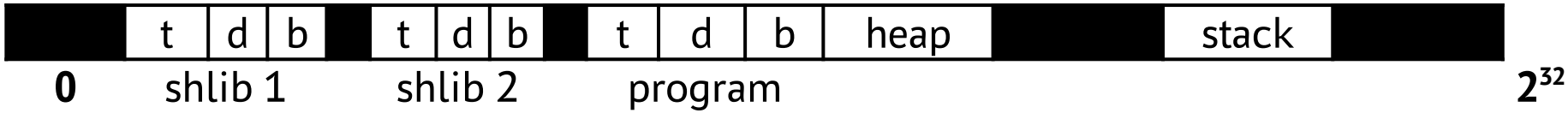
Стек не нуждается в явных системных вызовах для увеличения:

- либо он запускается с выделением для него столько оперативной памяти, сколько он может иметь (традиционный подход);
- либо существует область зарезервированных адресов ниже стека, для которой ядро автоматически выделяет ОЗУ, когда замечает туда попытку записи (это современный подход).

В любом случае, в нижней части адресного пространства, которую можно использовать для стека, может существовать или существовать «защитная» область (**guard area**).

Если такая область существует (все современные системы это делают), она никогда не отображается на физическую память и если либо стек, либо куча пытаются в него «врасти», немедленно возникает исключение ошибки сегментации.

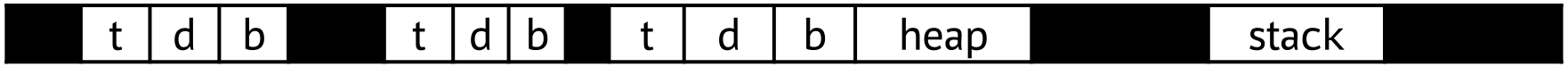
Адресное пространство в этом случае выглядит немного сложнее, но суть распределения памяти остается такой же.



Традиционное ядро не устанавливает границ — стек может дорасти до кучи или куча может дорасти до стека, они будут портить данные друг друга, и программа аварийно завершит работу.

Если очень повезет, это случится сразу после запуска.

Эту диаграмму *не следует* интерпретировать всеобъемлющим образом — тем, что в точности делает любая конкретная ОС, в том числе и Linux — Linux помещает исполняемый файл гораздо ближе к нулевому адресу, чем совместно используемые библиотеки.



Черные области на этой диаграмме не отображаются на физическую память и любая попытка к ним доступа вызывает немедленную ошибку сегментирования.

Размер таких областей для 64-разрядных программ обычно намного превышает размер отображаемой памяти.

Светлые области — это программа и ее совместно используемые библиотеки (отображаемых в пространство программы таких библиотек могут быть десятки).

У каждой совместно используемой библиотеки есть свои собственные сегменты кода, данных и bss.

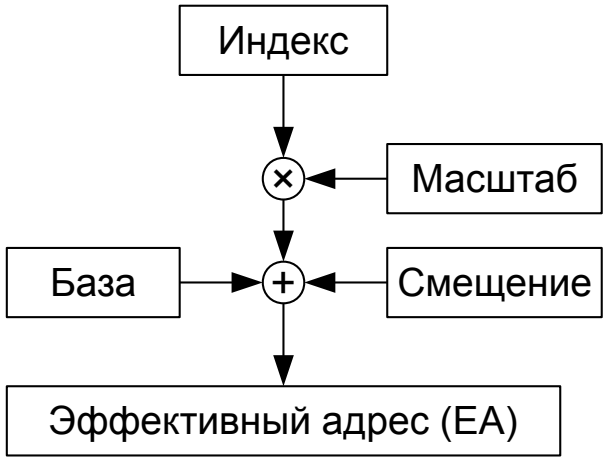
Куча не обязательно будет смежной с сегментом данных исполняемой программы, по крайней мере Linux для 64-разрядных программ обычно этого не делает.

Стек не привязан к вершине виртуального адресного пространства, а расстояние между кучей и стеком для 64-разрядных программ настолько велико, что обычно не нужно беспокоиться о его пересечении.

Адресация памяти в x86 (IA32)

Существует четыре вида адресов:
1. эффективный (адрес, формируемый процессором из инструкции[1]).
2. логический (сегмент и эффективный адрес);
3. линейный (или виртуальный);
4. физический — в системной памяти;

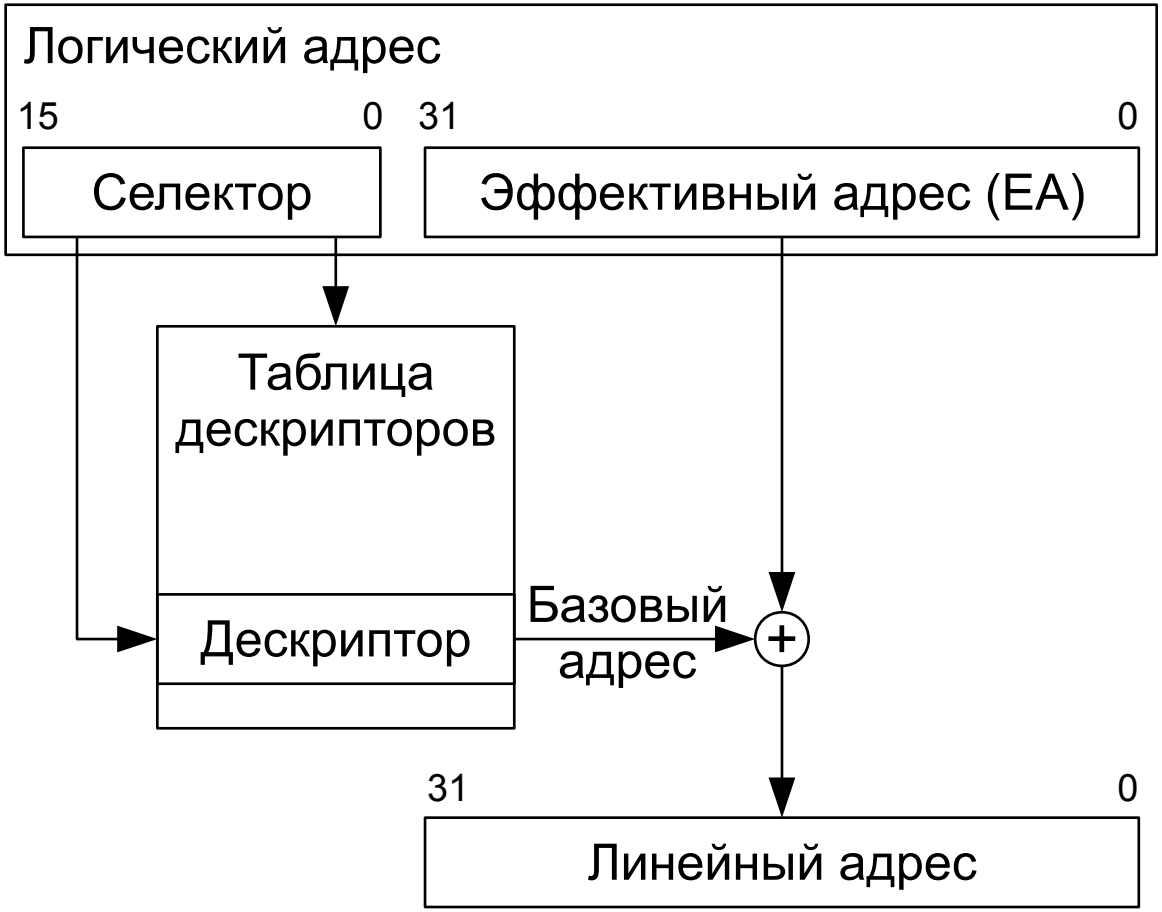
Формирование эффективного адреса



База — может содержаться в любом из РОН;
Индекс — может содержаться в любом из регистров, за исключением ESP;
Смещение — содержится в коде команды;
Масштаб — содержится в коде команды (1, 2, 4, 8).

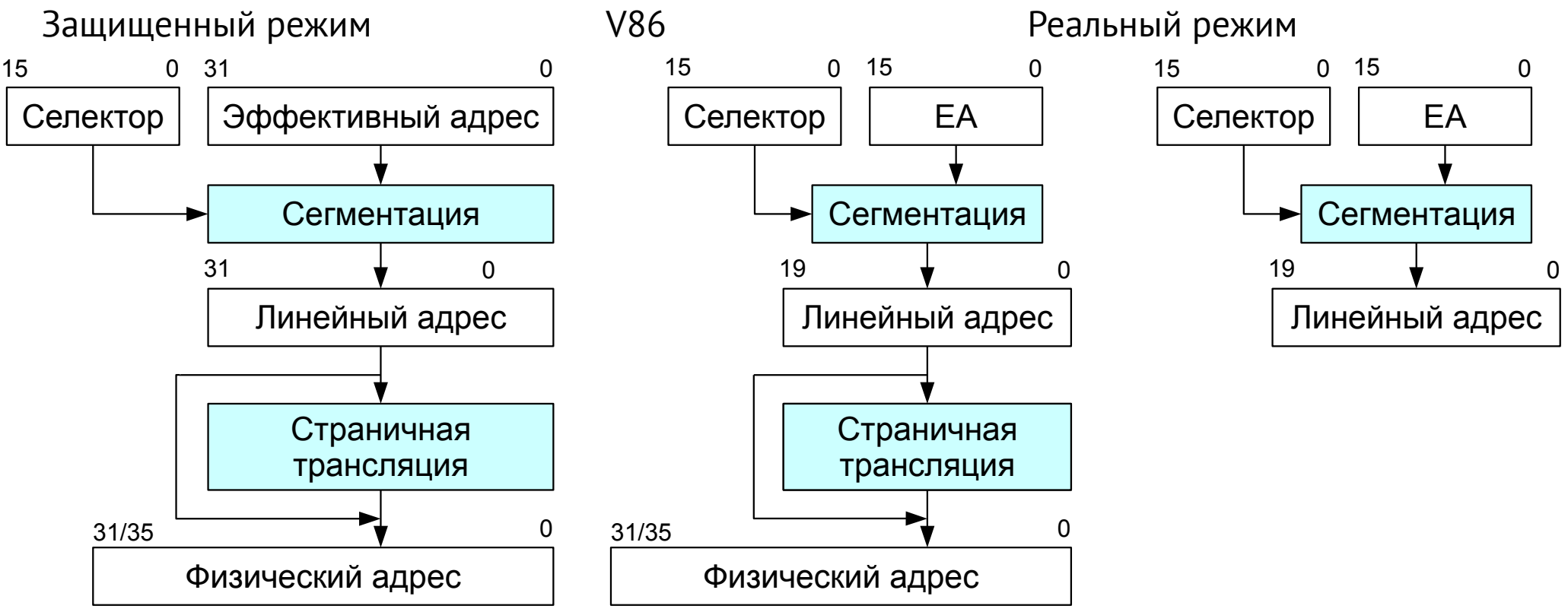
1) относительно начала сегмента

Формирование линейного адреса (сегментация) в IA32



1. из селектора извлекается поле индекса;
2. по индексу находится соответствующий дескриптор;
3. из дескриптора извлекается поле адреса базы.
4. К адресу базы добавляется смещение.

Формирование физического адреса

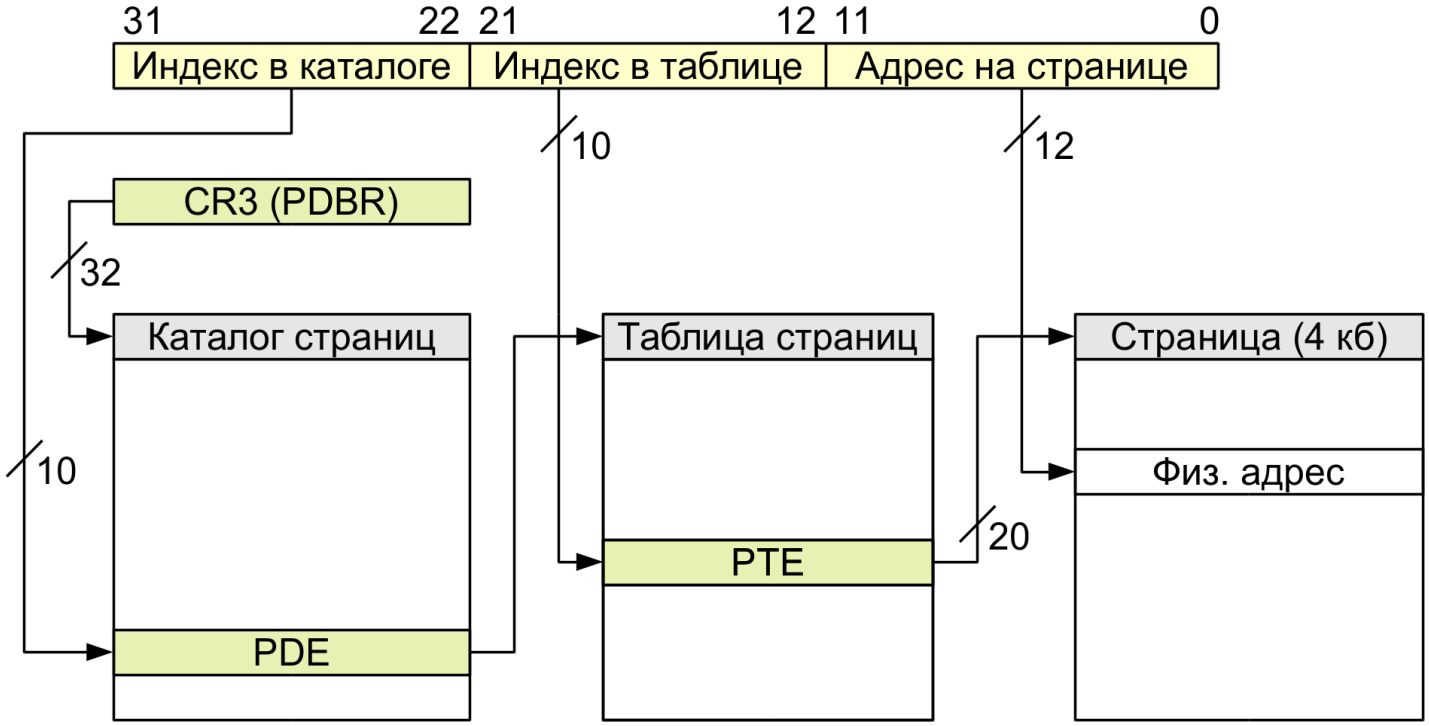


В случае страничной трансляции «линейный адрес» называется «виртуальный адрес».

Иные архитектуры могут иметь иные механизмы преобразования EA в физический адрес, тем не менее, понятие виртуального адреса остается.

(¤) Трансляция виртуального адреса в физический в обычном режиме IA32

При каждом обращении к памяти виртуальный адрес делится на три части:



Первая часть — индекс (PDE — Page Directory Entry) элемента в каталоге страниц (Page Directory). Из этого элемента извлекается физический адрес таблицы страниц (Page Table).

Вторая часть — индекс (PTE — Page Table Entry) в таблице страниц. Из этого элемента извлекается физический адрес страницы.

Третья часть интерпретируется как смещение в этой странице.

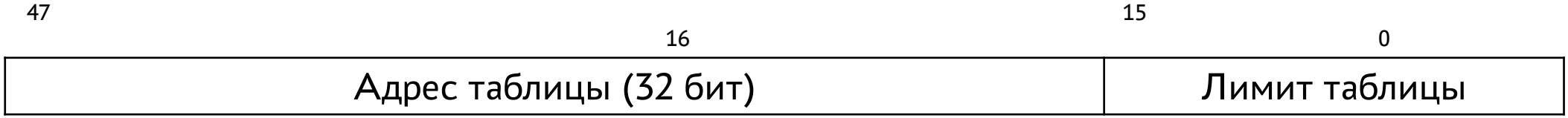
(¤) Регистр CR4

Регистр CR4 управляет дополнительными возможностями процессора, а также возможностями, которыми не управляет регистр CR0.

(¤) Регистры GDTR и IDTR

Регистры GDTR и IDTR служат для указания параметров глобальных таблиц: глобальной дескрипторной таблицы (GDT) и глобальной таблицы прерываний (IDT).

Формат регистров GDTR и IDTR



Адрес таблицы является абсолютным виртуальным адресом, он не зависит от содержимого каких-либо сегментных регистров.

(¤) Регистры LDTR и TR

Регистры LDTR и TR имеют размер 16 бит и хранят селекторы локальной дескрипторной таблицы (LDT) и текущей выполняющейся задачи (TSS) в GDT.

Также в этих регистрах есть теневые части, которые содержат сами дескрипторы в целях минимизации обращений к глобальной дескрипторной таблице (GDT).

Общая память

Потоки внутри процесса совместно используют одни и те же статические данные — глобальные переменные и данные со статическим классом размещения внутри функций (совместно используют пользовательский контекст).

Процессы работают в изолированных друг от друга адресных пространствах.

Например, потомок, запущенный вызовом **`fork()`**, получает всего лишь ***копию данных*** своего предка.

Общая память представляет собой область, которая может совместно использоваться несколькими процессами.

Как и в случае с потоками, чтобы упорядочить обмен данными через совместно используемую память, процессы должны пользоваться мьютексами или семафорами.

В обеих версиях процесс должен «открыть» сегмент общей памяти и получить указатель на нее, после чего этот указатель может свободно использоваться обычными операторами языков программирования C и C++, не прибегая к системным вызовам.

Как правило, процессы получают разные значения указателей, имеющие смысл только в пределах конкретного процесса, но они ссылаются на одну и ту же область физической памяти.

Общая память System V

Для работы с системными вызовами System V нужен ключ, который можно получить, например, с помощью **`ftok()`**. Работа с совместно используемой памятью выглядит следующим образом:

1) процесс при помощи системного вызова **`shmget(2)`** создает совместно используемый (общий) сегмент, указывая первоначальные права доступа к этому сегменту (чтение и/или запись), а также его размер в байтах.

2) чтобы затем получить доступ к совместно используемому сегменту, его нужно присоединить посредством системного вызова **`shmat(2)`**, который разместит сегмент в виртуальном пространстве процесса.

3) после присоединения, в соответствии с правами доступа, процессы могут читать данные из сегмента и записывать их (быть может, синхронизируя свои действия с помощью семафоров).

4) когда общий сегмент становится ненужным, его следует отсоединить, воспользовавшись системным вызовом **`shmdt(2)`**.

5) управление – для выполнения управляющих действий над общими сегментами памяти служит системный вызов **`shmctl(2)`**.

В число управляющих действий входит предписание удерживать сегмент в оперативной памяти и обратное предписание о снятии удержания.

6) после того, как последний процесс отсоединил сегмент общей памяти, следует выполнить управляющее действие по удалению сегмента из системы.

Сегмент общей памяти определяется структурой **shmid_ds**:

```
struct shmid_ds {
    struct ipc_perm  shm_perm;   // права на операции
    int              shm_segsz;  // размер сегмента (в байтах)
    time_t           shm_atime;  // время последнего подключения
    time_t           shm_dtime;  // время последнего отключения
    time_t           shm_ctime;  // время последнего изменения
    unsigned short   shm_cpid;   // ID процесса создателя
    unsigned short   shm_lpid;   // ID последнего пользователя
    short            shm_nattch; // количество подключений
};
```

Для каждого ресурса система использует общую структуру типа **struct ipc_perm**, хранящую необходимую информацию о правах для проведения IPC-операции.

Структура **ipc_perm** включает следующие поля:

```
struct ipc_perm {
    key_t   __key;
    ushort  uid;   // ID владельца euid и egid
    ushort  gid;   // ID группы владельца egid
    ushort  cuid;  // ID создателя euid
    ushort  cgid;  // ID группы создателя egid
    ushort  mode;  // младшие 9 битов shmflg -- права для операций чтения/записи
    ushort  seq;   // номер последовательности
};
```

shmget (2) — возвращает идентификатор сегмента общей памяти

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, // ключ
           int size,  // размер сегмента
           int shmflg // флаги создания
);
```

shmget() возвращает идентификатор сегмента общей памяти, соответствующий значению аргумента **key**.

Его можно использовать для получения идентификатора ранее созданного общего сегмента памяти если **shmflg** равно нулю и **key** не содержит значения **IPC_PRIVATE**, а также для создания нового.

Если значение **key** равно **IPC_PRIVATE**, создается новый сегмент общей памяти размером **size** (округленным до размера, кратного **PAGE_SIZE**).

Новый сегмент общей памяти также создается если значение **key != IPC_PRIVATE**, **но** если нет идентификатора, соответствующего **key**, причем **shmflg** должен содержать флаг **IPC_CREAT**.

Аргумент **size** имеет смысл только в том случае, если создается новый сегмент.

Вновь созданный сегмент памяти заполняется нулями.

Чтобы получить возможность пользоваться сегментом памяти, нужно обратиться к вызову **shmat()**, который вернет указатель.

IPC_PRIVATE является не полем, а типом **key_t**. – в этом случае системный вызов игнорирует все, кроме 9-и младших битов **shmflg**, и создает новый сегмент совместно используемой памяти.

Поле **shmflg** может содержать:

IPC_CREAT — служит для создания нового сегмента. Если этого флага нет, то функция **shmget()** будет искать сегмент, соответствующий ключу **key** и затем проверит, имеет ли пользователь права на доступ к сегменту.

IPC_EXCL — этот флаг используется совместно с **IPC_CREAT** для того, чтобы вызов создал новый сегмент. Если сегмент уже существует, то вызов завершается с ошибкой.

```
/* Mode bits for `msgget', `semget', and `shmget'.  */
#define IPC_CREAT  01000      // Создаем сегимент key если такого нет
#define IPC_EXCL   02000      // Неуспех если shm с key уже есть
#define IPC_NOWAIT 04000      // Return error on wait
```

mode_flags (младшие 9 битов) указывают на права создателя, владельца, группы и др.

Если создается новый сегмент, то права доступа копируются из **shmflg** в **shm_perm**, являющийся членом структуры **shmid_ds**, которая определяет сегмент.

```
struct shmid_ds {
   struct ipc_perm shm_perm;   // права операции
   int             shm_segsz;  // размер сегмента (в байтах)
   time_t          shm_atime;  // время последнего подключения
   time_t          shm_dtime;  // время последнего отключения
   time_t          shm_ctime;  // время последнего изменения
   unsigned short  shm_cpid;   // ID процесса создателя
   unsigned short  shm_lpid;   // ID последнего пользователя
   short           shm_nattch; // количество подключений
};
```

```
struct ipc_perm {
    key_t          __key;    // ключ, передаваемый в shmget(2)
    uid_t          uid;      // эффективный UID владельца
    gid_t          gid;      // эффективный GID владельца
    uid_t          cuid;     // эффективный UID создателя
    gid_t          cgid;     // эффективный GID создателя
    unsigned short mode;     // права + флаги SHM_DEST и
                                SHM_LOCKED
    unsigned short __seq;    // порядковый номер
};
```

При создании нового сегмента совместно используемой (общей) памяти системный вызов инициализирует структуру данных **shmid_ds** следующим образом:

- устанавливаемые значения **shm_perm.cuid** и **shm_perm.uid** становятся равными значению идентификатора эффективного пользователя вызывающего процесса.

- **shm_perm.cgid** и **shm_perm.gid** устанавливаются равными идентификатору эффективной группы пользователей вызывающего процесса.

- младшим 9-и битам **shm_perm.mode** присваивается значение младших 9-и битов **shmflg**.

- **shm_segsz** присваивается значение **size**.

- устанавливаемое значение **shm_lpid**, **shm_nattch**, **shm_atime** и **shm_dtime** становится равным нулю.

- **shm_ctime** устанавливается на текущее время.

Если сегмент уже существует, то права доступа подтверждаются, а проверка производится для того, чтобы убедиться, что сегмент не помечен на удаление.

Возвращаемое значение

При удачном завершении вызова возвращается идентификатор сегмента **shmid**, и **-1** при ошибке и **errno** устанавливается в:

EINVAL — если создается новый сегмент, а **size < SHMMIN** или **size > SHMMAX**, либо новый сегмент не был создан.

EINVAL — сегмент с данным ключом существует, но **size** больше чем размер этого сегмента.

EEXIST — если значение **IPC_CREAT | IPC_EXCL** было указано, а сегмент уже существует.

ENOSPC — если все возможные идентификаторы сегментов уже распределены (**SHMMNI**) или если размер выделяемого сегмента превысит системные лимиты (**SHMALL**).

ENOENT — если не существует сегмента для ключа **key**, а значение **IPC_CREAT** не указано.

EACCES — если у пользователя нет прав доступа к сегменту общей памяти.

ENOMEM — если в памяти нет свободного для сегмента пространства.

Ограничения для сегментов общей памяти

Ниже приведены ограничения для сегментов общей памяти, которые могут отразиться на вызове **`shmget()`**.

SHMALL – Максимальное количество страниц общей памяти зависит от настроек системы.

SHMMAX – Максимальный размер сегмента в байтах зависит от системных настроек (обычно это 4M).

SHMMIN – Минимальный размер сегмента в байтах зависит от системных настроек (обычно он равен одному байту, поэтому **PAGE_SIZE** является минимальным эффективным размером).

SHMMNI – Максимальное количество сегментов общей памяти в системе.

SHMSEG – Максимальное количество сегментов общей памяти на процесс.

```
#define SHMMIN 1
#define SHMMNI 4096
#define SHMMAX (ULONG_MAX - (1UL << 24))
#define SHMALL (ULONG_MAX - (1UL << 24))
#define SHMSEG SHMMNI
```

??? Выбор названия **`IPC_PRIVATE`** неудачен, более подошло бы по смыслу **`IPC_NEW`** ???

Состояния сегмента общей памяти

Бит удержания	Бит подкачки	Бит размещения	Состояние
0	**0**	**0**	**Неразмещенный сегмент**
0	**0**	**1**	**В памяти**
0	1	0	Не используется
0	**1**	**1**	**На диске**
1	0	0	Не используется
1	**0**	**1**	**Удерживается в памяти**
1	1	0	Не используется
1	1	1	Не используется

Неразмещенный сегмент — сегмент общей памяти, ассоциированный с данным идентификатором, но для использования не размещен.

В памяти — сегмент размещен для использования. Это означает, что сегмент существует и в данный момент находится в оперативной памяти.

На диске — сегмент в данный момент вытолкнут на устройство подкачки.

Удерживается в памяти — сегмент удерживается в оперативной памяти и не будет рассматриваться в качестве кандидата на выталкивание, пока не будет снято удержание.

Удерживать и освобождать общие сегменты может только суперпользователь.

Не используется — состояние в настоящий момент не используется и при работе обычного пользователя с общими сегментами памяти возникнуть не может.

После того, как создан уникальный идентификатор общего сегмента памяти и ассоциированная с ним структура данных, можно использовать системные вызовы семейства **`shmop()`** — операции над сегментами общей памяти и **`shmctl()`** — управление сегментами общей памяти.

shmctl() — управление сегментами общей памяти

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid,              // ID общего сегмента памяти, полученный от shmget().
           int cmd,                // команда
           struct shmid_ds *buf); // предоставляемая пользователем структура данных
```

shmctl() позволяет пользователю:
- получать информацию о общих сегментах памяти;
- устанавливать владельца, группу общего сегмента, права на него;
- удалить сегмент.

shmid – идентификатор общего сегмента памяти, полученный при помощи **shmget()**.
Информация о сегменте **shmid**, возвращается в структуре **shmid_ds**.
В **shm_perm** могут устанавливаться следующие поля:

```
struct ipc_perm {
  key_t  key;
  ushort uid;  // действующие ID владельца и группы euid и egid
  ushort gid;
  ushort cuid; // действующие ID создателя euid и egid
  ushort cgid;
  ushort mode; // младшие 9 битов shmflg
  ushort seq;  // номер последовательности
};
```

Значения аргумента **cmd** могут быть следующими:

IPC_STAT – используется для копирования информации о сегменте в буфер **buf**.

Пользователь должен иметь права на чтение сегмента.

IPC_SET – используется для применения пользовательских изменений к содержимому полей **uid**, **gid** или **mode** в структуре **shm_perms**.

Используются только младшие 9 битов **mode**.

Поле **shmid_ds.shm_ctime** при этом обновляется.

Пользователь должен быть владельцем, создателем общего сегмента памяти или суперпользователем.

IPC_RMID – используется для пометки сегмента как удаленного.

Сегмент будет удален после отключения (например, когда поле **shm_nattch** ассоциированной структуры **shmid_ds** станет равным нулю).

Пользователь должен быть владельцем, создателем сегмента или суперпользователем.

Пользователь должен удостовериться, что сегмент удален, иначе страницы, которые не были удалены, останутся в памяти или в разделе подкачки.

Важно

После **fork()** дочерние процессы наследуют сегменты общей памяти.

После **exec()** все подключенные сегменты общей памяти отключаются (но не удаляются).

Если выполнен вызов **exit()**, все сегменты общей памяти отключаются (но не удаляются).

Возвращаемое значение

При удачном выполнении возвращается 0, а при ошибке -1.

В случае ошибки переменной **errno** присваиваются следующие значения:

EACCES — Она возникает, если запрашивается **IPC_STAT** , а **shm_perm.mode** не дает доступа для чтения.

EFAULT — Аргумент **cmd** равен **IPC_SET** или **IPC_STAT**, а адрес, указываемый **buf**, недоступен.

EINVAL — Эта ошибка происходит, если **shmid** является неверным идентификатором сегмента или **cmd** является неправильной командой.

EIDRM — Эта ошибка возвращается, если **shmid** указывает на удаленный идентификатор.

EPERM — Эта ошибка возвращается, если была произведена попытка выполнить **IPC_SET** или **IPC_RMID**, эффективный идентификатор вызывающего процессы не является идентификатором создателя (в соответствие с **shm_perm.cuid**), владельца (в соответствие с **shm_perm.uid**) или суперпользователя.

EOVERFLOW — возвращается если запрашивается **IPC_STAT**, а значения **gid** или **uid** слишком велики для помещения в структуру, на которую указывает **buf**.

Операции над сегментами общей памяти

```
#include <sys/types.h>
#include <sys/shm.h>

// операция присоединения сегментов (attach)
void *shmat(int         shmid,   // id сегмента общей памяти
            const void *shmaddr, // адрес присоединения
            int         shmflg)  // флаги (SHM_RND, SHM_RDONLY)

// операция отсоединения сегментов (detach)
int shmdt(const void *shmaddr)  // адрес присоединения
```

shmat() — присоединение сегментов

Функция **shmat()** подсоединяет сегмент общей памяти **shmid** к адресному пространству вызывающего процесса.

shmid — идентификатор сегмента общей памяти, предварительно полученный при помощи системного вызова **shmget()**.

shmaddr – задает адрес, по которому сегмент должен быть присоединен, то есть тот адрес в виртуальном пространстве пользователя, который получит начало сегмента.

Не всякий адрес является приемлемым.

shmaddr — адрес присоединенного сегмента определяется параметром согласно одного из перечисленных ниже критериев:

- если **shmaddr** равен **NULL**, то система выбирает для присоединяемого сегмента подходящий (неиспользованный) адрес в адресном пространстве процесса.

- если **shmaddr** не равен **NULL**, а в поле **shmflg** включен флаг **SHM_RND**, то присоединение производится по адресу **shmaddr**, округленному ***вниз*** до ближайшего кратного адресу границы сегмента (**SHMLBA**), если же флаг не включен, **shmaddr** будет округлен до размера страницы.

Рекомендуется использовать адреса вида

```
0x80000000
0x80040000
0x80080000
 . . .
```

Если значение **shmaddr** равно нулю, система выбирает адрес присоединения по своему усмотрению (это наиболее предпочтительный вариант).

shmflg — параметр используется для передачи системному вызову **shmat()** флагов:

- **SHM_RND** — адрес **shmaddr** следует округлить до некоторой системно-зависимой величины.

- **SHM_RDONLY** — присоединяемый сегмент будет доступен только для чтения, и вызывающий процесс должен иметь права на чтение этого сегмента. В противном случае сегмент будет доступен для чтения и записи, и у процесса должны быть соответствующие права. Сегментов «только-запись» не существует.

При завершении работы процесса (**exit( )**) сегмент будет отсоединен.

Один и тот же сегмент может быть присоединен в адресное пространство процесса несколько раз, как «только для чтения», так и в режиме «чтение-запись».

При удачном выполнении системный вызов **shmat( )** обновляет содержимое структуры **shmid_ds**, связанной с сегментом общей памяти, следующим образом:

shm_atime — устанавливается в текущее время.
shm_lpid — устанавливается в идентификатор вызывающего процесса.
shm_nattch — увеличивается на 1.

Присоединение производится и в том случае, если присоединяемый сегмент помечен на удаление.

При успешном завершении системного вызова **shmat( )** результат равен адресу, который получил присоединенный сегмент. В случае неудачи возвращается -1.

shmdt() — отсоединение сегментов

```
#include <sys/types.h>
#include <sys/shm.h>

// операция отсоединения сегментов (detach)
int shmdt(const void *shmaddr)  // адрес присоединения
```

Функция **shmdt()** отсоединяет сегмент общей памяти, находящийся по адресу **shmaddr**, от адресного пространства вызывающего процесса. Эта функция освобождает занятую ранее этим сегментом область памяти в адресном пространстве процесса.

Отсоединяемый сегмент должен быть среди присоединенных ранее функцией **shmat()**.

shmaddr – задает начальный адрес отсоединяемого сегмента общей памяти.

При удачном выполнении системный вызов **shmdt()** обновляет содержимое структуры **shmid_ds**, связанной с сегментом общей памяти, следующим образом:

shm_dtime – устанавливается в текущее время.

shm_lpid – устанавливается в идентификатор вызывающего процесса.

shm_nattch – уменьшается на 1.

Если значение **shm_nattch** становится равным 0, а сегмент помечен на удаление, то сегмент удаляется из памяти (но не удаляется из системы).

При успешном завершении системного вызова **shmdt()** результат равен нулю; в случае неудачи возвращается -1.

После того, как последний процесс отсоединил сегмент общей памяти, этот сегмент вместе с идентификатором и ассоциированной структурой данных следует удалить с помощью системного вызова **`shmctl()`**.

Недостатков общей памяти System V немного. Вызовы просты, эффективны и хорошо реализованы, однако:

Не следует полагать, что операции с указателем на общую память выполняются атомарно.

Поэтому всегда при совместном использовании памяти процессами следует предусматривать какой-либо механизм, управляющий доступом к ней, например семафоры или мьютексы, как это имеет место в случае с потоками.