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, 2023

2023.04.19

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.

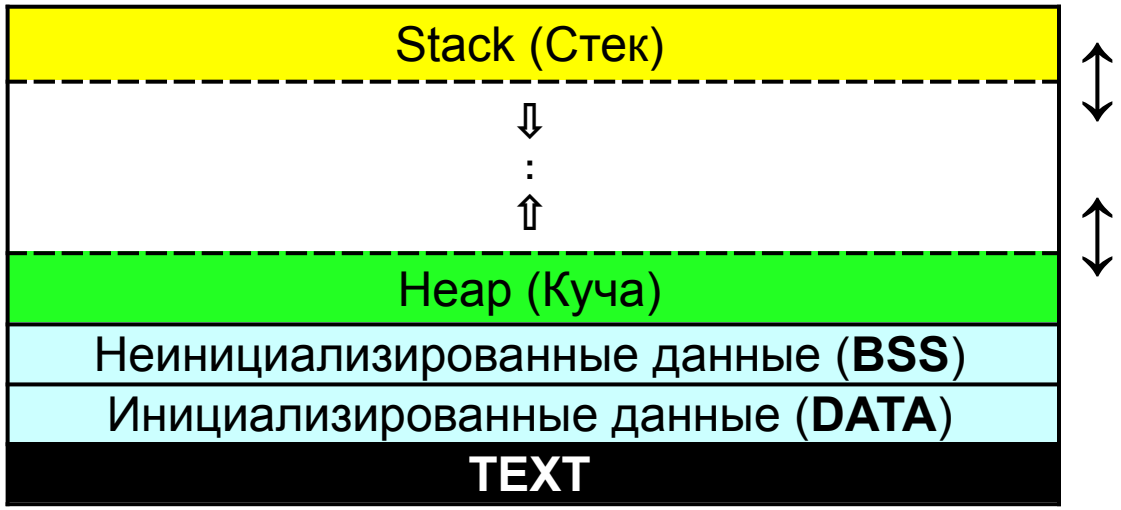
Память компьютерной программы

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)

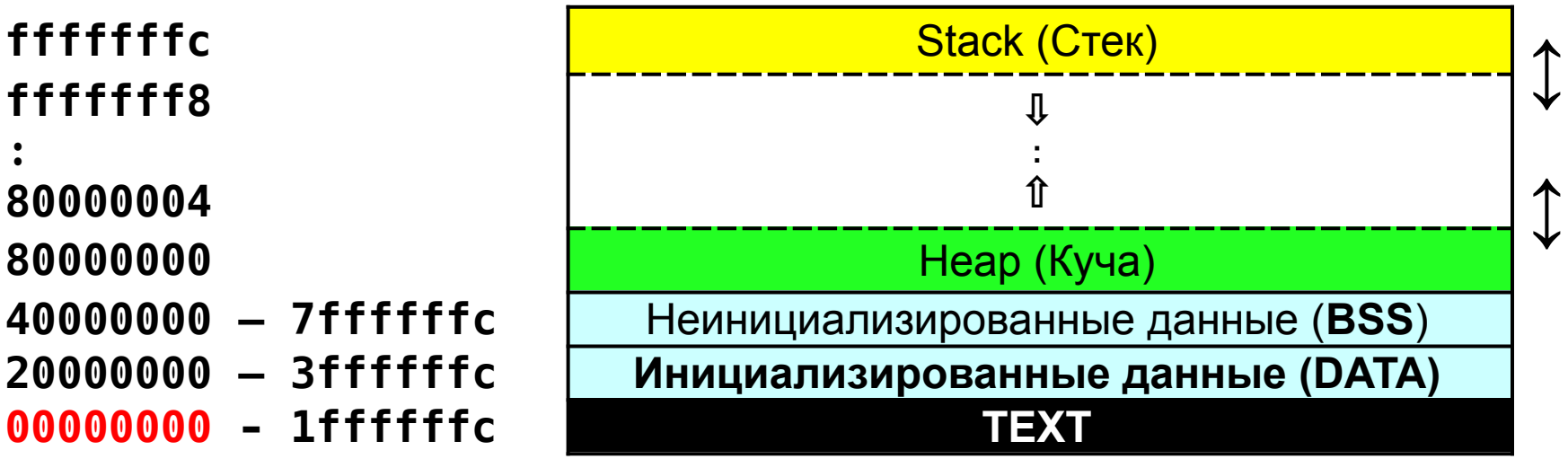
Сегмент **.data** содержит любые глобальные или статические переменные, которые имеют предопределенное значение и могут изменяться.

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)

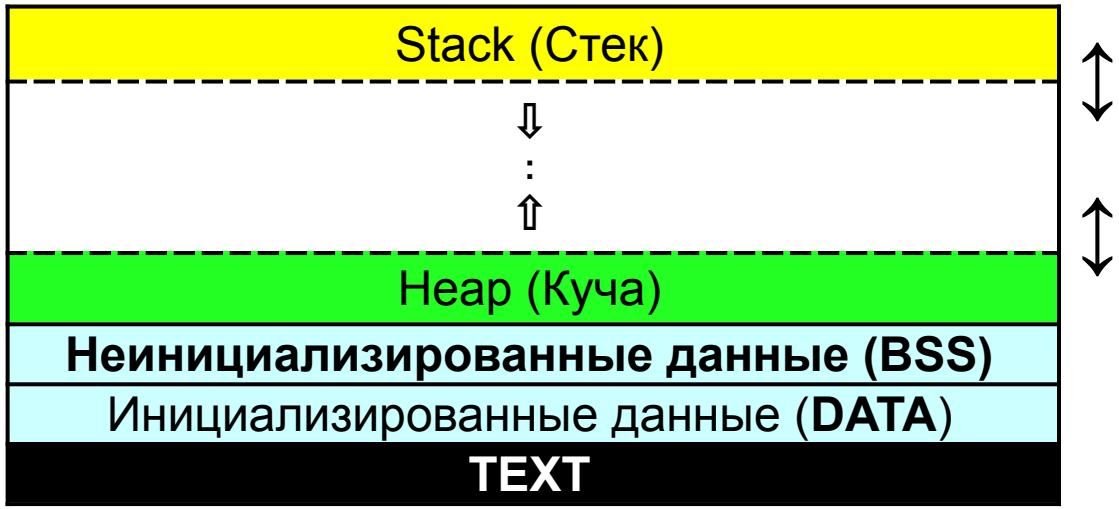
Сегмент BSS, известный как неинициализированные данные, содержит все глобальные переменные и статические переменные, которые инициализируются нулем или не имеют явной инициализации в исходном коде. Обычно примыкает к сегменту данных.

Например, переменная, определенная как

```
static int i;
```

будет содержаться в сегменте BSS.

```
ffffffc
fffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc
```



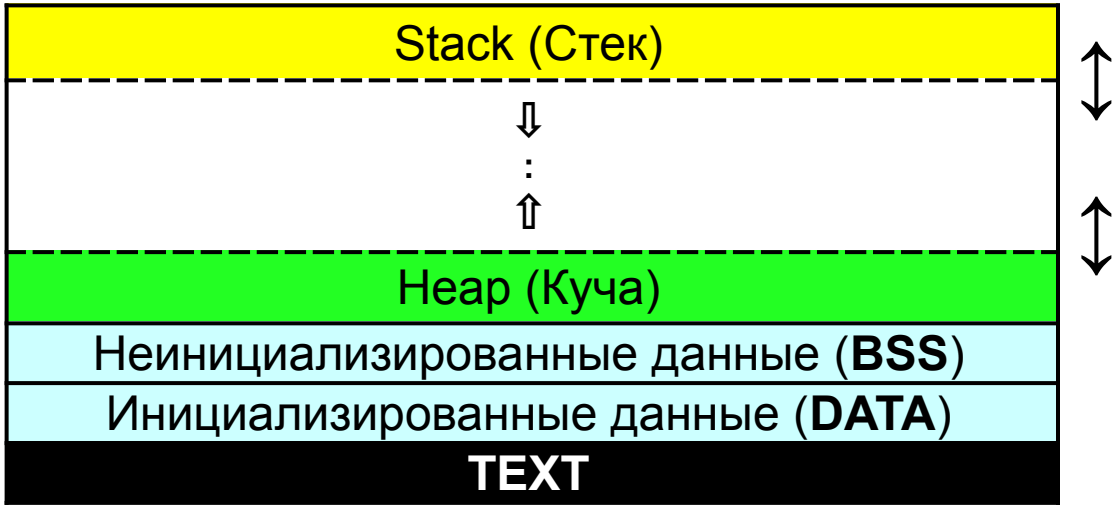
Куча (Heap)

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.

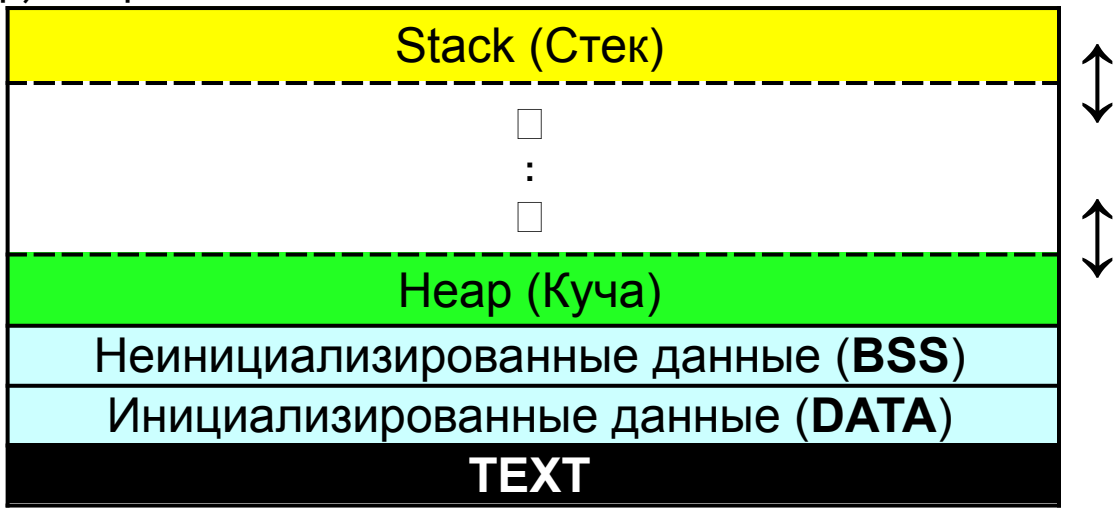
Программа для выполнения считывается в память, где и остается вплоть до своего завершения. Поэтому утверждение о том, что размер вашего двоичного файла не влияет на использование памяти, НЕВЕРНО.

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:

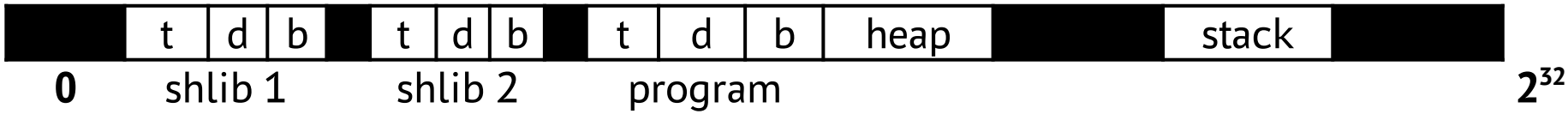
- либо он запускается с выделением для него столько оперативной памяти, сколько он может иметь (традиционный подход);

- 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.

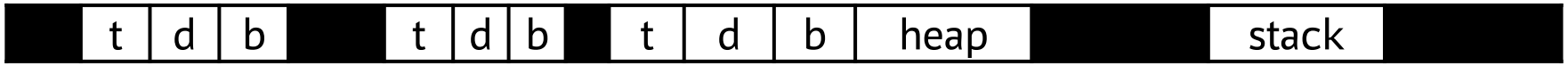
Адресное пространство в этом случае выглядит немного сложнее, но суть распределения памяти остается такой же.



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.



0 shlib 1 shlib 2 program $\mathbf{2^{32}}$

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.

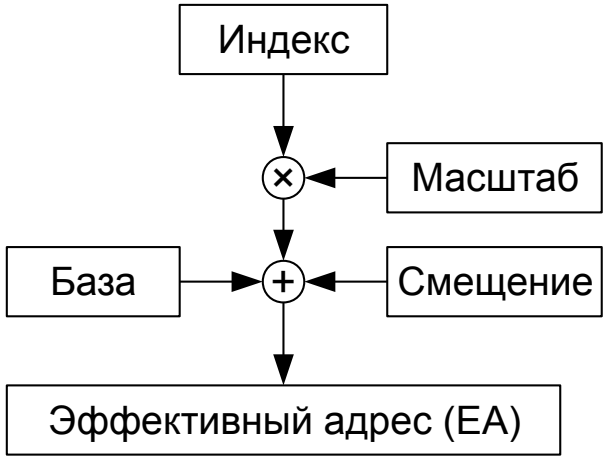
Куча не обязательно будет смежной с сегментом данных исполняемой программы, по крайней мере Linux для 64-разрядных программ обычно этого не делает.

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;

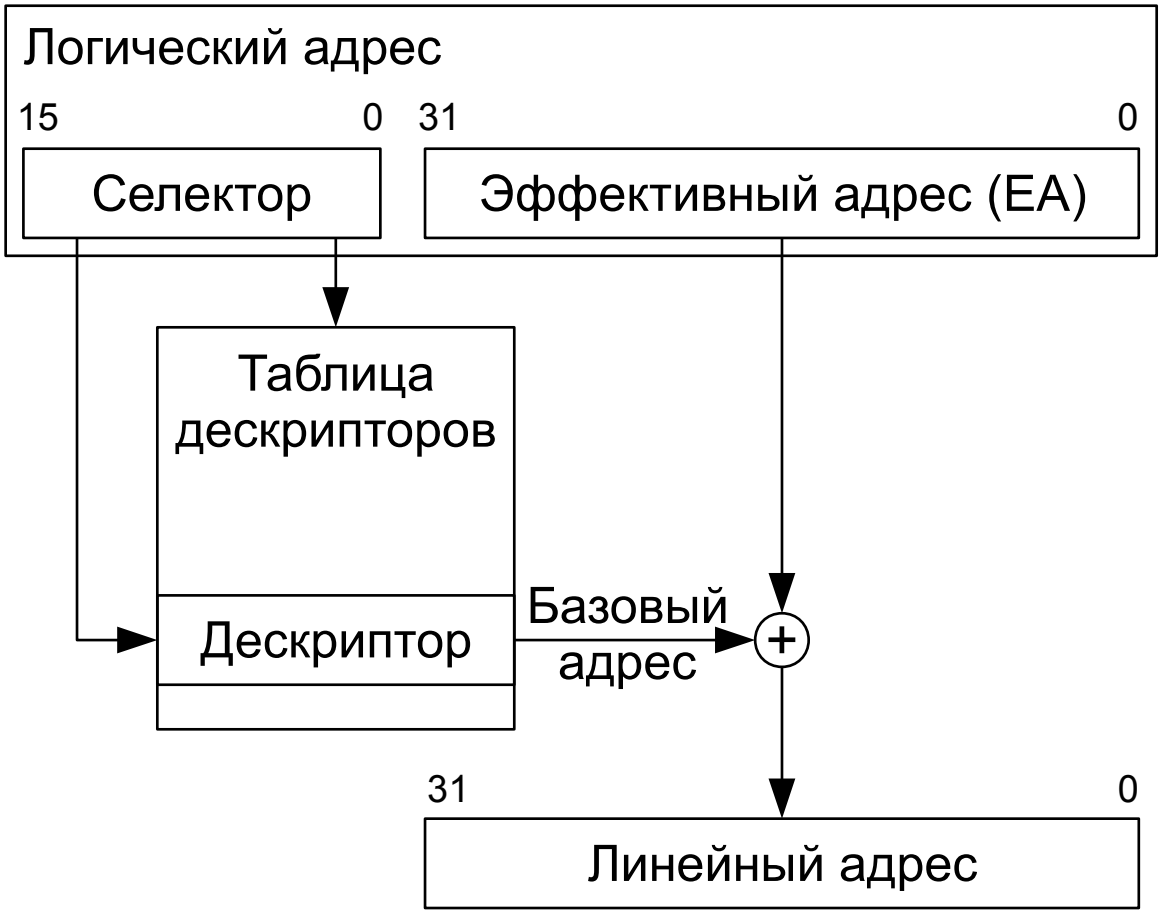
Формирование эффективного адреса



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).

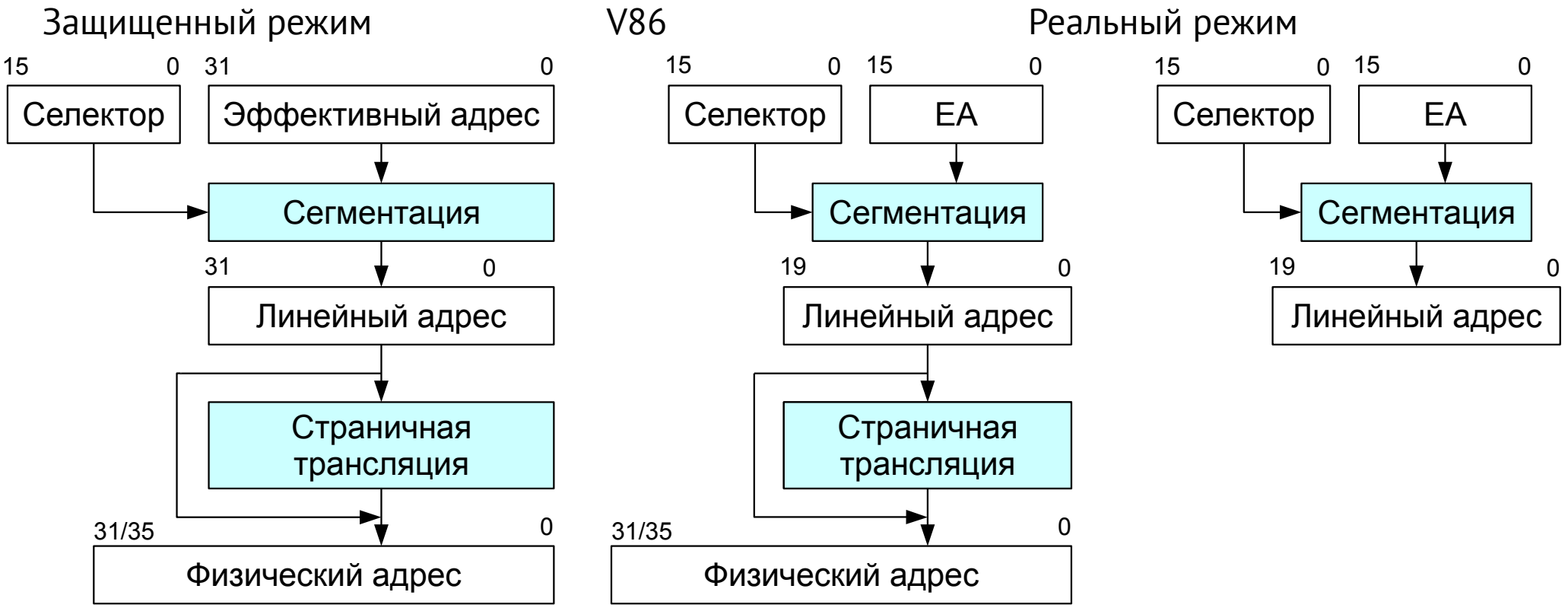
1) относительно начала сегмента

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

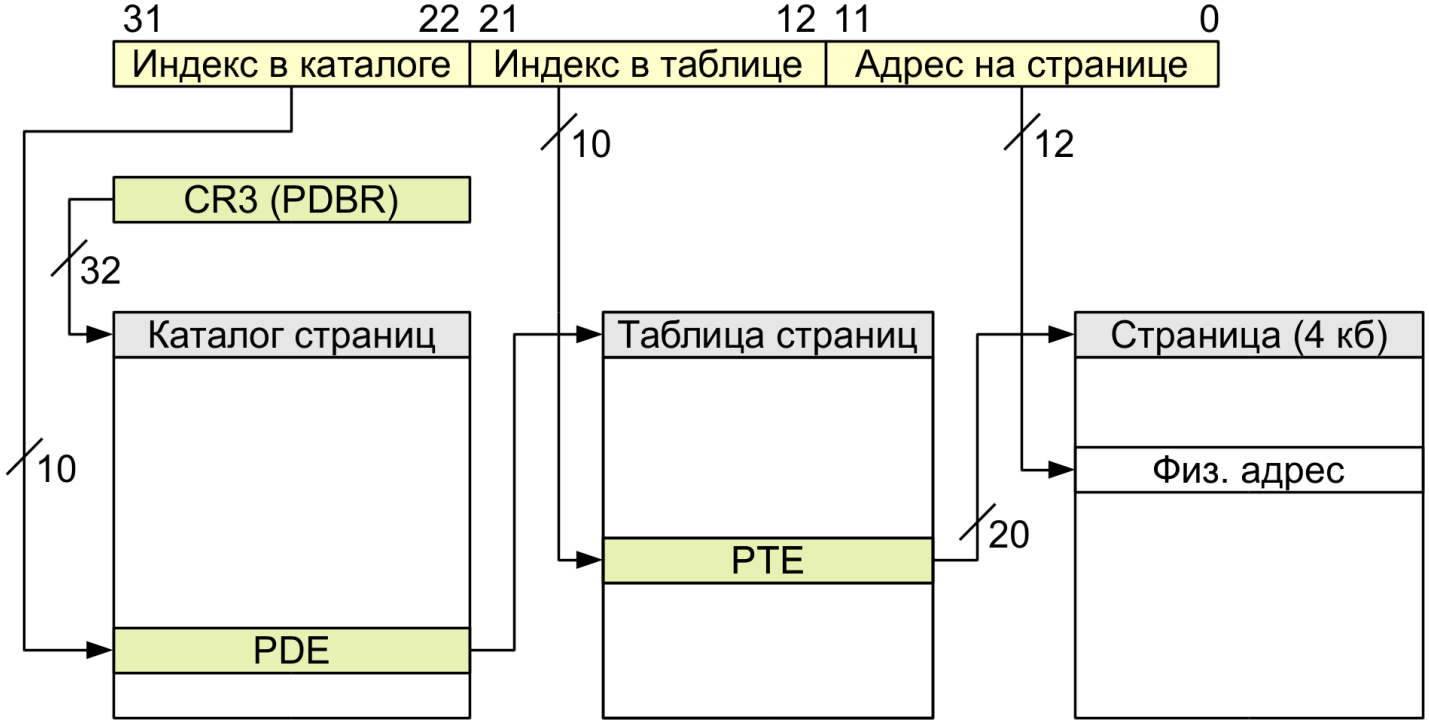


В случае страничной трансляции «линейный адрес» называется «виртуальный адрес».

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.

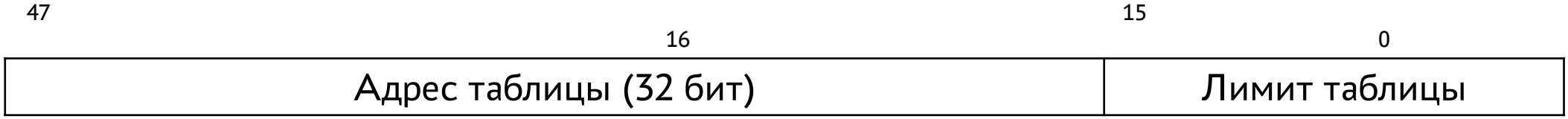
(¤) Регистр CR4

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

Чтобы затем получить доступ к совместно используемому сегменту, его нужно присоединить посредством системного вызова **`shmat(2)`**, который разместит сегмент в виртуальном пространстве процесса. После присоединения, в соответствии с правами доступа, процессы могут читать данные из сегмента и записывать их (быть может, синхронизируя свои действия с помощью семафоров).

Когда разделяемый сегмент становится ненужным, его следует отсоединить, воспользовавшись системным вызовом **`shmdt(2)`**.

Для выполнения управляющих действий над разделяемыми сегментами памяти служит системный вызов **`shmctl(2)`**.

В число управляющих действий входит предписание удерживать сегмент в оперативной памяти и обратное предписание о снятии удержания. После того, как последний процесс отсоединил разделяемый сегмент, 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 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**). Так же новый сегмент общей памяти 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 ключем 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 разделяемой 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 процесса 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 разделяемой памяти.

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

Если выполнен вызов **exit()**, все сегменты разделяемой 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>

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

int shmdt(              // операция отсоединения сегментов (detach)
  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 флаг не включен, **shmaddr** должен быть 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>

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

Функция **shmdt()** отсоединяет сегмент общей памяти, находящийся по адресу **shmaddr**, от адресного пространства вызвающего 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.

Обзор общей памяти POSIX

Общая память System V (**shmget(2)**, **shmop(2)** и так далее) является старым API.

POSIX предоставляет более простой и лучше спроектированный интерфейс.

API общей памяти POSIX позволяет процессам обмениваться информацией через общую область памяти.

Процессы должны синхронизировать свой доступ к объекту общей памяти, например, с использованием семафоров POSIX.

Доступные интерфейсы:

shm_open(3) — создаёт и открывает новый объект, или открывает существующий объект. Аналог **open(2)**. Вызов возвращает файловый дескриптор, которые используется другими интерфейсами общей памяти POSIX. Размер созданного объекта общей памяти равен нулю.

shm_unlink(3) — Удаляет объект общей памяти с заданным именем.

close(2) — Закрывает файловый дескриптор (выделенный **shm_open(3)**), когда он больше не требуется.

ftruncate(2) — Назначает размер общего объекта памяти.

mmap(2) — Отображает объект общей памяти в виртуальное адресное пространство вызвавшего процесса.

munmap(2) — Удаляет отображение объекта общей памяти из виртуального адресного пространства вызвавшего процесса.

fstat(2) — Возвращает структуру **stat**, в которой описан объект общей памяти. Информация, возвращаемой этим вызовом: размер объекта (**st_size**), права (**st_mode**), владелец (**st_uid**) и группа (**st_gid**).

fchown(2) — Изменяет владельца объекта общей памяти.

fchmod(2) — Изменяет права на объект общей памяти.

Устойчивость

Объекты общей памяти POSIX являются устойчивыми на уровне ядра – объект будет существовать до самого отключения системы или до тех пор, пока все процессы не разорвут связь с объектом, после чего он может быть удален с помощью **shm_unlink(3)**.

Доступ к объектам общей памяти через файловую систему

В Linux объекты общей памяти создаются в виртуальной файловой системе (**tmpfs(5)**), которая обычно монтируется в каталог **/dev/shm**. Начиная с ядра версии 2.6.19, в Linux поддерживается использование списков контроля доступа (ACL) для управления доступа к объектам в виртуальной файловой системе.

shm_open(), shm_unlink() — создаёт и открывает или удаляет объекты общей памяти

```
#include <sys/mman.h>
#include <sys/stat.h> // константы для mode
#include <fcntl.h>    // константы O_*

int shm_open(const char *name, // имя объекта общей памяти
             int oflag,        // флаги доступа
             mode_t mode);     // права на объект при его создании

int shm_unlink(const char *name);
```

Компонуется с **librt** при указании параметра **-lrt**.

Функция **shm_open()** создаёт и открывает новый или открывает уже существующий объект общей памяти POSIX.

Объект общей памяти POSIX — это дескриптор, используемый несвязанными процессами для выполнения системного вызова **mmap(2)** для одной области общей памяти.

Реализация объектов общей памяти POSIX в Linux использует выделенную файловую систему **tmpfs(5)**, которая обычно монтируется в **/dev/shm**.

Функция **shm_unlink()** выполняет обратную операцию, удаляя объект, созданный ранее с помощью **shm_open()**.

```
int shm_open(const char *name, // имя объекта общей памяти
             int oflag,        // флаги доступа
             mode_t mode);     // права на объект при его создании
```

Действие **shm_open()** аналогично действию **open(2)**.

name – определяет создаваемый или открываемый объект общей памяти. Для использования в переносимых программах объект общей памяти должен опознаваться по имени в виде **/какое_то_имя**, то есть строкой, оканчивающейся **null** и длиной до **NAME_MAX** (т.е., 255) символов, **состоящей из начальной косой черты** и одного или более символов (любых, кроме косой черты).

oflag – содержит маску битов, созданную логическим сложением **OR** *одного из* флагов **O_RDONLY** или **O_RWDR** и любых других флагов, перечисленных далее.

O_RDONLY — открытый таким образом объект общей памяти можно указывать в **mmap(2)** только для чтения (**PROT_READ**).

O_RDWR — открыть объект для чтения и записи.

O_CREAT — создать объект общей памяти, если он не существует.

Владелец и группа объекта устанавливаются из соответствующих эффективных ID вызвавшего процесса, а биты прав на объект устанавливаются в соответствии с младшими 9 битами **mode**, за исключением того, что биты, установленные маске режима создания файла (см. umask(2)), очищаются у новых объектов. Набор макросов-констант, используемых для определения **mode**, описан в **open(2)**. Символические определения этих констант можно получить включением заголовка **<sys/stat.h>**.

Новый объект общей памяти изначально имеет нулевую длину, а для установки размера объекта можно использовать **ftruncate(2)**. Объект общей памяти автоматически заполняется 0.

O_EXCL — Если также был указан **O_CREAT** и объект общей памяти с заданным **name** уже существует, то возвращается ошибка.

Проверка существования объекта и его создание, если он не существует, выполняется атомарно.

O_TRUNC — Если объект общей памяти уже существует, то обрезать его до 0 байтов.

Определения значений этих флагов можно получить включением **<fcntl.h>**.

При успешном выполнении **shm_open()** возвращает новый файловый дескриптор, ссылающийся на объект общей памяти. Этот файловый дескриптор гарантированно будет дескриптором файла с самым маленьким номером среди ещё не открытых процессом.

У дескриптора файла устанавливается флаг **FD_CLOEXEC** (см. **fcntl(2)**).

Дескриптор файла обычно используется в последующих вызовах **ftruncate(2)** (для новых объектов) и **mmap(2)**. После вызова **mmap(2)** дескриптор файла может быть закрыт без влияния на отображение памяти.

Действие **shm_unlink()** аналогично **unlink(2)** — оно удаляет имя объекта общей памяти и, как только все процессы завершили работу с объектом и отменили его отображение, очищает пространство и уничтожает связанную с ним область памяти.

После успешного выполнения **shm_unlink()** попытка выполнить **shm_open()** для объекта с тем же именем **name** завершается ошибкой (если не был указан **O_CREAT**, в этом случае создаётся новый, уже другой объект).

Возвращаемое значение

При успешном выполнении **shm_open()** возвращает неотрицательный дескриптор файла.

При ошибках **shm_open()** возвращает -1.

При успешном выполнении **shm_unlink()** возвращает 0 и -1 при ошибке.

Ошибки

При ошибках в **errno** записываются причины ошибки. Значения **errno** могут быть такими:

EACCES — Отказ в доступе для **shm_unlink()** для объекта общей памяти.

EACCES — Отказ в доступе для **shm_open()** с заданным **name** и режимом **mode**, или был указан **O_TRUNC**, а вызывающий не имеет прав на запись для объекта.

EEXIST — В **shm_open()** указаны **O_CREAT** и **O_EXCL**, но объект общей памяти **name** уже существует.

EINVAL — Аргумент **name** для **shm_open()** некорректен.

EMFILE — Было достигнуто ограничение по количеству открытых файловых дескрипторов на процесс.

ENAMETOOLONG — Длина name превышает **PATH_MAX**.

ENFILE — Достигнуто максимальное количество открытых файлов в системе.

ENOENT — Была сделана попытка выполнить **shm_open()** для несуществующего **name** и при этом не был указан **O_CREAT**.

ENOENT — Была сделана попытка выполнить **shm_unlink()** для несуществующего **name**.

Замечания

POSIX оставляет неопределённым поведение при комбинации **O_RDONLY** и **O_TRUNC**.

В Linux это приводит к успешному обрезанию существующего объекта общей памяти, но в других системах UNIX может быть по-другому.

truncate(), ftruncate() — обрезает файл до заданного размера

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

int ftruncate(int fd,           // файловый дескриптор
              off_t length);    //

int truncate(const char *path, // имя файла
             off_t length);    //
```

Функции **truncate()** и **ftruncate()** обрезают обычный файл, указанный по имени **path** или ссылке **fd**, до размера, указанного в **length** (в байтах).

Вызов **ftruncate()** также может использоваться для установки размера объекта общей памяти POSIX, полученной с помощью **shm_open(3)**. Если до этого объект был больше указанного размера, все лишние данные будут утеряны. Если объект был меньше, он будет увеличен, а дополнительная часть будет заполнена нулевыми байтами ('**\0**'). **Смещение в файле не изменяется.**

Если размер изменился, поля **st_ctime** и **st_mtime** (время последнего изменения состояния и время последнего изменения, соответственно) объекта будут обновлены, а биты режимов **set-user-ID** и **set-group-ID** могут быть сброшены.

Для **ftruncate()** объект должен быть ***открыт на запись***.

Для **truncate()** файл должен быть ***доступен на запись***.

Возвращаемое значение

При успешном выполнении возвращается 0. В случае ошибки возвращается -1, а **errno** устанавливается в соответствующее значение.

Ошибки

Для **truncate()**:

EACCES — В одном из каталогов префикса не разрешен поиск, либо указанный файл не доступен на запись для пользователя.

EFAULT — Значение **path** указывает за пределы адресного пространства, выделенного процессу.

EFBIG — Аргумент **length** больше максимально допустимого размера файла/объекта.

EINTR — При блокирующем ожидании завершения вызов был прерван обработчиком сигналов.

EINVAL — Аргумент **length** является отрицательным или больше максимально допустимого размера объекта.

EIO — Во время обновления индексного дескриптора (inode) возникла ошибка ввода/вывода.

EISDIR — Указанный файл является каталогом.

ELOOP — Во время определения **pathname** встретилось слишком много символьных ссылок.

ENAMETOOLONG — Компонент имени пути содержит более 255 символов, или весь путь содержит более 1023 символов.

ENOENT — Указанный файл не существует.

ENOTDIR — Компонент в префиксе пути не является каталогом.

EPERM — Используемая файловая система не поддерживает расширение файла больше его текущего размера.

EPERM — Выполнение операции предотвращено опечатыванием (наложением ограничений)[2].

EROFS — Указанный файл находится на файловой системе, смонтированной только для чтения.

ETXTBSY — Файл является исполняемым файлом, который в данный момент исполняется.

---

2) file sealing (man fcntl(2))

Для **ftruncate()** действуют те же ошибки, за исключением того, что вместо ошибок, связанных с неправильным **path**, появляются ошибки, связанные с файловым дескриптором **fd**:

EBADF — Значение **fd** не является правильным файловым дескриптором.

EBADF или **EINVAL** — Дескриптор **fd** не открыт для записи.

EINVAL — Дескриптор **fd** не указывает на обычный файл или объект общей памяти POSIX.

EINVAL или **EBADF** — Файловый дескриптор **fd** не открыт на запись. В POSIX это допускается и переносимые приложения должны обрабатывать любую ошибку для этого случая (Linux возвращает **EINVAL**).

Замечания

На некоторых 32-битных архитектурах интерфейс этих системных вызовов отличается от описанного выше по причинам, указанным в **syscall(2)**.

mmap(), munmap() — отображение в памяти, или удаление отображения

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES
void *mmap(void   *addr,    // адрес начала отображения
           size_t  length,  // количество отображаемых байтов
           int     prot,    // желаемый режим защиты памяти
           int     flags,   // тип отражаемого объекта и опции
           int     fd,      // дескриптор открытого файла/объекта (shm_open)
           off_t   offset); // смещение в файле

int munmap(void *addr,
           size_t length);
#endif
```

Вызов **mmap()** создаёт новое отображение в виртуальном адресном пространстве вызывающего процесса.

addr – адрес начала нового отображения.

length – задаётся длина отображения (должна быть больше 0).

Если значение **addr** равно **NULL**, то ядро само выбирает адрес (выровненный по странице), по которому создаётся отображение.

Это наиболее переносимый метод создания нового отображения. Настоящее местоположение возвращается ОС и никогда не бывает равным **NULL**.

Если значение **addr** не равно **NULL**, то ядро *учитывает это* при размещении отображения.

В Linux ядро выберет ближайшую к границе страницу (но всегда выше или равною значению, заданному в **/proc/sys/vm/mmap_min_addr**) и попытается создать отображение.

Если по этому адресу уже есть отображение, то ядро выберет новый адрес, который может и не зависеть от подсказки. Адрес нового отображения возвращается как результат вызова.

fd — корректный файловый дескриптор для объекта, который мы хотим отобразить в адресное пространство, в частности, это значение, которое вернул системный вызов **shm_open()**.

Содержимое *файлового отображения* (в отличие от анонимного отображения **MAP_ANONYMOUS**) инициализируется данными из файла (или объекта), на который указывает файловый дескриптор **fd**, длиной **length** байт, начиная со смещения **offset**.

offset – должно быть кратно размеру страницы[3].

После возврата из вызова **mmap()** файловый дескриптор **fd** может быть закрыт, при этом отображение остается действительным.

prot – указывается желаемая защита памяти отображения (не должна конфликтовать с режимом открытого объекта/файла). Значением может быть **PROT_NONE** или побитово сложенные (OR) следующие флаги:

PROT_EXEC — Страницы доступны для исполнения.
PROT_READ — Страницы доступны для чтения.
PROT_WRITE — Страницы доступны для записи.
PROT_NONE — Страницы недоступны.

---

3) возвращается **sysconf(_SC_PAGE_SIZE)**

flags — задаётся будут ли изменения отображения видимы другим процессам, отображающим ту же область, и будут ли изменения перенесены в отображённый файл. Данное поведение определяется в **flags** одним из следующих значений:

MAP_SHARED — Сделать отображение общим. Изменения отображения видимы всем процессам, отображающим ту же область и (если отображение выполняется из файла) изменения заносятся в отображённый файл (для более точного контроля над изменениями файла нужно использовать **msync(2)**).

MAP_PRIVATE — Создать закрытое отображение с механизмом копирования при записи (COW). Изменения отображения невидимы другим процессам, отображающим тот же файл, и сам файл не изменяется. Будут ли видимы в отображённой области изменения в файле, сделанные после вызова **mmap()**, не определено,

MAP_NORESERVE — Не резервировать страницы пространства подкачки для этого отображения. Если пространство подкачки резервируется, то для отображения гарантируется возможность изменения. Если пространство подкачки не резервируется, то можно получить сигнал **SIGSEGV** при записи, если физическая память будет недоступна.

Память, отображённая с помощью **mmap()**, сохраняется при **fork(2)** с теми же атрибутами.

Файл отображается по кратному размеру страницы. Для файла, который не кратен размеру страницы, оставшаяся память при отображении заполняется нулями, и запись в эту область не приводит к изменению файла. Действия при изменении размера отображаемого файла на страницы, которые соответствуют добавленным или удалённым областям файла, не определены.

MAP_FIXED – не учитывать **addr** как подсказку, размещать отображение точно по этому адресу.

Значение **addr** должно быть выровнено соответствующим образом – на большинстве архитектур оно должно быть кратно размеру страницы.

Некоторые архитектуры могут накладывать дополнительные ограничения.

Если область памяти, задаваемая **addr** и **len**, перекрывается со страницами существующих отображений, то перекрывающаяся часть существующих отображений будет отброшена.

Если заданный адрес не может быть использован, то вызов **mmap()** завершается ошибкой.

В переносимом ПО флаг **MAP_FIXED** нужно использовать осторожно, так как точная раскладка процесса в памяти, доступная для изменения, может значительно отличаться в разных версиях ядер, библиотеки C и выпусках операционной системы.

MAP_ANONYMOUS (или **MAP_ANON**) – отображение не привязанное к файлу.

Его содержимое инициализируется нулями, а аргумент **fd** игнорируется.

В некоторых реализациях при указании **MAP_ANONYMOUS** (или **MAP_ANON**) требуется указывать **fd** равным -1, и так всегда нужно поступать для переносимости приложений.

offset – должен быть равен нулю.

Использование **MAP_ANONYMOUS** вместе с **MAP_SHARED** поддерживается в Linux только начиная с ядра версии 2.4.

munmap()

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES
int munmap(void *addr, size_t length);
#endif
```

Системный вызов **munmap()** удаляет отображение для указанного адресного диапазона и это приводит к тому, что дальнейшее обращение по адресам внутри диапазона приводит к генерации неправильных ссылок на память. Также для диапазона отображение автоматически удаляется при завершении работы процесса.

Закрытие файлового дескриптора не приводит к удалению отображения диапазона.

addr должен быть кратен размеру страницы (но значения **length** это не касается).

Все страницы, содержащие хотя бы часть указанного диапазона, удаляются из отображения и последующие ссылки на эти страницы приводят к генерации сигнала **SIGSEGV**.

Если указанный диапазон не содержит каких-либо отображённых страниц, это не ошибка.

Возвращаемое значение

При успешном выполнении **mmap()** возвращается указатель на отображённую область.

При ошибке возвращается значение **MAP_FAILED** (а именно, **(void *)-1**) и **errno** устанавливается в соответствующее значение.

При успешном выполнении **munmap()** возвращает 0.

При сбое возвращается -1, и код ошибки кладётся в **errno** (скорее всего **EINVAL**).

Ошибки

EACCES — Файловый дескриптор указывает на не обычный файл.

EACCES — Было запрошено отображение файла (mapping), но **fd** не открыт на чтение.

EACCES — Был указан флаг **MAP_SHARED** и установлен бит **PROT_WRITE**, но **fd** не открыт в режиме чтения/записи (**O_RDWR**).

EACCES — Был указан флаг **PROT_WRITE**, но файл доступен только для дополнения.

EAGAIN — Файл заблокирован, или блокируется слишком много памяти (смотрите setrlimit(2)).

EBADF — Значение **fd** не является правильным файловым дескриптором.

EINVAL — Неправильное значение addr, length или offset (например, оно либо слишком велико, либо не выровнено по границе страницы).

EINVAL — (начиная с Linux 2.6.12) Значение length равно 0.

ENFILE — Достигнуто максимальное количество открытых файлов в системе.

ENODEV — Используемая файловая система для указанного файла не поддерживает отображение памяти.

ENOMEM — Больше нет доступной памяти.

ENOMEM — Процесс превысил бы ограничение на максимальное количество отображений. Эта ошибка также может возникнуть в **munmap()** при удалении отображения области в середине существующего отображения, так как при этом выполняется удаление отображения двух отображений меньшего размера на любом конце области.

EOVERFLOW — На 32-битной архитектуре вместе с расширением для больших файлов (т.е., используется 64-битный **off_t**): количество страниц, используемых для **length** плюс количество страниц, используемых для offset приводит к переполнению **unsigned long** (32 бита).

EPERM Аргументом prot запрашивается **PROT_EXEC**, но отображённая область принадлежит файлу на файловой системе, которая смонтирована с флагом **no-exec**.

EPERM — Выполнение операции предотвращено «опечатыванием».

Сигналы

При использовании отображаемой области памяти могут возникать следующие сигналы:

SIGSEGV — Попытка записи в область, отображённую только для чтения.

SIGBUS — Попытка доступа к части буфера, которая не совпадает файлом (например, она может находиться за пределами файла. Подобной является ситуация, когда другой процесс уменьшил длину файла).

Доступность

В системах POSIX, в которых есть вызовы **mmap()**, **msync(2)** и **munmap()**, значение **_POSIX_MAPPED_FILES**, определённое в **<unistd.h>**, больше 0.

Замечания

На некоторых архитектурах (i386), флаг **PROT_WRITE** подразумевает флаг **PROT_READ**. Также от архитектуры зависит подразумевает ли **PROT_READ** флаг **PROT_EXEC** или нет.

Переносимые программы должны всегда устанавливать **PROT_EXEC**, если они собираются выполнять код, находящийся в отображении.

Переносимый способ создания отображения

Состоит в том, чтобы указать в **addr** значение 0 (**NULL**) и не использовать **MAP_FIXED** в **flags**.

В этом случае, система сама выберет адрес для отображения. Адрес, выбранный таким образом, не будет конфликтовать с существующими отображениями и не будет равен **NULL**.

Если указан флаг **MAP_FIXED** и значение **addr** равно 0 (**NULL**), то адрес отображения будет равен 0 (**NULL**).

Приложение может определить какие страницы отображены в данный момент в буфере/страничном кэше с помощью вызова **mincore(2)**.

Отличия между библиотекой C и ядром

`mmap()` является обёрточной функцией из библиотеки **`glibc`**. Раньше, эта функция обращалась к системному вызову с тем же именем.

Начиная с ядра 2.4, данный системный вызов был заменён на **`mmap2(2)`**.

В настоящее время обёрточная функция **`mmap()`**, вызывает **`mmap2(2)`** с подходящим подкорректированным значением **`offset`**.

fstat() — считывает статус файлового дескриптора

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int fstat(int fd,             //
          struct stat *buf); // буфер, предоставляемый пользователем
```

fstat() возвращает информацию о **fd**, возвращаемый **shm_open(2)** и заполняет буфер **buf** структуры **stat**:

```
struct stat {
   dev_t     st_dev;     // устройство
   ino_t     st_ino;     // inode
   mode_t    st_mode;    // режим доступа
   nlink_t   st_nlink;   // количество жестких ссылок
   uid_t     st_uid;     // идентификатор пользователя-владельца
   gid_t     st_gid;     // идентификатор группы-владельца
   dev_t     st_rdev;    // тип устройства если это устройство
   off_t     st_size;    // общий размер в байтах
   blksize_t st_blksize; // размер блока ввода-вывода в ФС
   blkcnt_t  st_blocks;  // количество выделенных блоков
   time_t    st_atime;   // время последнего доступа
   time_t    st_mtime;   // время последней модификации
   time_t    st_ctime;   // время последнего изменения
};
```

fchown(2) — изменяет владельца объекта общей памяти

```
#include <unistd.h>

int fchown(int fd, uid_t owner, gid_t group);
```

Только привилегированный процесс может сменить владельца файла.

Владелец файла может сменить группу на любую группу, в которой он числится.

Привилегированный процесс может задавать произвольную группу.

Если параметр **owner** или **group** равен **-1**, то соответствующий **ID** не изменяется.

Когда владелец или группа исполняемого файла изменяется непривилегированным пользователем, то биты режима **S_ISUID** и **S_ISGID** сбрасываются.

В POSIX не указано, должно ли это происходить если **fchown()** выполняется суперпользователем

Возвращает

При успешном выполнении возвращается 0.

В случае ошибки возвращается **-1**, а **errno** устанавливается в соответствующее значение.

fchmod(2) — Изменяет права на объект общей памяти

```
#include <sys/stat.h>

int fchmod(int fd, mode_t mode);
```

Изменяют биты режима дескриптора (режим дескриптора в общем случае состоит из бит прав доступа к объекту плюс биты **set-user-ID**, **set-group-ID** и бит закрепления).

S_I<op><who>

<op> — **R**, **W**

<who> — **USR**, **GRP**, **OTH**

Возвращает

При успешном выполнении возвращается 0.

В случае ошибки возвращается **-1**, а **errno** устанавливается в соответствующее значение.