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

2024.04.18

Оглавление

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), позволяющий 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) — Устанавливает 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 (должен) 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 типа **tmpfs(5)**, которая обычно монтируется в **/dev/shm**.

```
ls -ld /dev/shm
drwxrwxrwt. 2 root root 100 апр 11 11:18 /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 **'/'**).

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 у новых объектов биты, установленные маске режима создания файла (см. **umask(2)**), очищаются.

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

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 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 используется 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)[1].

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

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

---

1) 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>     // bits/posix_opt.h: #define _POSIX_MAPPED_FILES 200809L
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES
void *mmap(void   *addr,    // адрес начала отображения (м/б NULL)
           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,     // удаляет отображние начиная с addr
           size_t length);  // и размером 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 равную значению, заданному в **/proc/sys/vm/mmap_min_addr**) и попытается создать отображение.

```
$ cat /proc/sys/vm/mmap_min_addr
65536
```

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()**.

offset – должен быть равен нулю.

После возврата из вызова mmap(2) 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.

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

MAP_SHARED — Сделать отображение общим.

Изменения отображения видимы всем процессам, отображающим ту же область.

MAP_PRIVATE — Создать закрытое отображение с механизмом копирования при записи (COW).

Изменения отображения невидимы другим процессам.

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.

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/дескриптору.

Его содержимое инициализируется нулями, а аргумент **fd** игнорируется. Используется для создания общей памяти, которая используется только родственными процессами.

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

Механизм копирования при записи (Copy-On-Write, COW)

Для оптимизации многих процессов, происходящих в операционной системе, таких как, например, работа с оперативной памятью или файлами на диске, используется «Механизм копирования при записи (Copy-On-Write, COW)».

Главная идея **COW** — при копировании областей данных создавать реальную копию только когда ОС обращается к этим данным с целью записи.

Например, при работе UNIX-функции **fork()** вместо копирования выполняется отображение образа памяти материнского процесса в адресное пространство дочернего процесса, после чего ОС запрещает обоим процессам запись в эту память.

Попытка записи в отображённые страницы вызывает исключение (exception), после обработки которого часть данных будет скопирована в новую область.

Всегда необходимо задавать либо **MAP_SHARED**, либо **MAP_PRIVATE**.

Если в качестве его значения выбрано **MAP_SHARED**, то полученное отображение файла впоследствии будет использоваться и другими процессами, вызвавшими **mmap()** для этого файла с аналогичными значениями параметров, а все изменения, сделанные в отображенном файле, будут сохранены во вторичной памяти.

Если в качестве значения параметра **flags** указано **MAP_PRIVATE**, то процесс получает отображение файла в свое монопольное распоряжение, но все изменения в нем не могут быть занесены во вторичную память (т.е., проще говоря, не сохранятся).

Вышеуказанные три флага описаны в POSIX.1b (бывшем POSIX.4) and SUSv2.

Пример создания сегмента ОП для процесса и его потомков

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

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {

  // Можно читать и писать
  int protection = PROT_READ | PROT_WRITE; // will be readable and writable

  // Сегмент будет общим (то есть другие процессы смогут получить к нему доступ),
  // но анонимным (то есть сторонние процессы не смогут получить для него адрес),
  // поэтому его использовать смогут только этот процесс и его дочерние элементы:
  int visibility = MAP_SHARED | MAP_ANONYMOUS;

  // Остальные параметры `mmap()` для этого варианта использования не существенны
  return mmap(NULL,       // ОС сама определяет куда отобразить
              size,       // размер
              protection, // желаемый режим защиты памяти
              visibility, // флаги с типом отражаемого объекта и опции
              -1,         // файловый дескриптор (игнорир. для MAP_ANONYMOUS)
              0);         // смещение
}
```

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 помещается 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 (200809L).

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**.

mremap() — изменяет отражение адреса виртуальной памяти

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

void *mremap(void *old_address,    // старый адрес виртуальной памяти
             size_t old_size,      // старый размер блока виртуальной памяти
             size_t new_size,      // требуемый размер блока виртуальной памяти
             unsigned long flags); //
```

mremap() увеличивает или уменьшает размер текущего отражения памяти, одновременно перемещая его при необходимости, что контролируется параметром **flags** и доступным виртуальным адресным пространством.

Специфична для Linux и не должна использоваться в переносимых программах.

При удачном выполнении **mremap()** возвращает указатель на новую область виртуальной памяти. При ошибке возвращается **-1**, а переменная **errno** устанавливается в соответствующее ошибке значение.

Память Linux делится на страницы. Пользовательскому процессу выделяется один или несколько неразрывных сегментов виртуальной памяти. Каждый из этих сегментов имеет одно или несколько отображений в реальной памяти посредством таблиц страниц.

У каждого сегмента есть своя защита, или свои права доступа. При сегментировании может случиться ошибка, если производится попытка неразрешенного доступа, например, запись информации в сегмент, режим которого «только для чтения». Доступ к виртуальной памяти за пределами сегментов также приведет к ошибке сегментирования.

old_address — старый адрес виртуальной памяти, которую необходимо изменить. **old_address** должен быть выровнен по границе страницы.

old_size — старый размер блока виртуальной памяти.

new_size — требуемый размер блока виртуальной памяти.

flags — параметр flags состоит из побитно и логически сложенных флагов.

mremap() использует таблицы страниц Linux и изменяет соответствие виртуальных адресов страницам памяти.

MREMAP_MAYMOVE — указывает, вернет ли функция ошибку или изменит виртуальный адрес, если невозможно изменить размер сегмента данного виртуального адреса.

mlock() — запрещает страничный обмен в некоторых областях памяти

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

int mlock(const void *addr, // начало области
          size_t      len); // размер области
```

mlock() запрещает страничный обмен памяти в области, начинающейся с адреса **addr** длиной **len** байтов. Все страницы памяти, включающие в себя часть заданной области памяти, будут помещены в ОЗУ, если системный вызов **mlock()** проделан успешно, и они останутся в памяти до тех пор, пока не произойдет одно из:

- страницы не будут освобождены функциями **munlock()** или **munlockall()**;
- страницы не будут высвобождены при помощи **munmap()**;
- процесс не завершит работу;
- процесс не запустит другую программу при помощи **exec()**.

Блокировка страниц не наследуется дочерними процессами, созданными с помощью fork().

Блокировка памяти используется, в основном, в двух случаях:

- в алгоритмах реального времени;
- в работе с защищенными данными.

Программам реального времени необходимы предсказуемые задержки в работе, а страничный обмен (наряду с системой переключения процессов) может привести к неожиданным задержкам в работе.

Режим таких приложений часто переключается на режим реального времени при помощи функции **sched_setscheduler()**.

Криптографические системы защиты данных очень часто содержат важные данные, например, пароли или секретные ключи, в структурах данных.

В результате страничного обмена эти данные могут попасть в область подкачки, находящуюся на устройстве длительного хранения (таком, как жесткий диск), где к этим данным после того, как они пропадут из памяти, может получить доступ практически кто угодно.

ВНИМАНИЕ!!! — в режиме «засыпания» на лаптопах и некоторых компьютерах копия памяти ОЗУ системы сохраняется на жесткий диск независимо от блокировок памяти.

Блокировка памяти не попадает в стек, т.е., страницы, блокированные несколько раз при помощи функций **mlock()** или **mlockall()**, будут разблокированы одним вызовом **munlock()** с соответствующими параметрами или **munlockall()**.

Страницы, в которых размещены несколько областей памяти или принадлежащие нескольким процессам, будут заблокированы в памяти до тех пор, пока они заблокированы хотя бы в одной из областей памяти или хотя бы одним процессом.

В POSIX-системах, в которых доступны **mlock()** и **munlock()**, в **<unistd.h>** задана константа **_POSIX_MEMLOCK_RANGE**, а в <limits.h> значение **PAGESIZE**, задающее количество байтов на странице.

В Linux **addr** автоматически округляется вниз до ближайшей границы страницы. Однако, согласно POSIX 1003.1-2001, возможны реализации этой функции, требующие, чтобы **addr** изначально был выравнен по границе страниц, поэтому переносимые приложения должны гарантировать выравнивание.

При удачном завершении вызова возвращается **0**.

При ошибке возвращается **-1**, а переменной **errno** присваивается номер ошибки, и ни с одной из блокировок памяти ничего не происходит.

mlockall() запрет страничного обмена для всех страниц в области памяти

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

int mlockall(int flags);
```

Запрещает страничный обмен для всех страниц в области памяти вызывающего процесса.
Запрет страничного обмена касается:
1) всех страниц сегментов кода, данных и стека;
2) совместно используемых библиотек;
3) пользовательских данных ядра;
4) совместно используемой процессом памяти;
5) отраженных в память файлов.

Все эти страницы будут помещены в ОЗУ, если вызов **mlockall()** был выполнен успешно, и останутся там до тех пор, пока не произойдет одно из:
1) страницы не будут освобождены функциями **munlock()** или **munlockall()**;
2) процесс не завершит работу;
3) процесс не запустит другую программу при помощи **exec()**.

Блокировка страниц не наследуется дочерними процессами, созданными с помощью fork().

Блокировка памяти используется, в основном, в двух случаях:
1) в алгоритмах реального времени;
2) в работе с защищенными данными.

Параметр **flags** формируется побитовым сложением следующих констант:

MCL_CURRENT — заблокировать все страницы, находящиеся в адресном пространстве процесса на текущий момент.

MCL_FUTURE — заблокировать все страницы, которые будут переданы процессу в будущем. Это могут быть страницы растущей кучи или стека, а также отраженные в память файлы и общие области памяти.

Если была задана константа **MCL_FUTURE**, и после этого количество заблокированных процессом страниц превысит лимит, то системный вызов, потребовавший новые страницы, их не получит и пошлет сообщение об ошибке **ENOMEM**.

Если новые страницы будут затребованы растущим стеком, то ядро не разрешит увеличение стека и пошлет процессу сигнал **SIGSEGV**.

Процессы, выполняющиеся в реальном времени, должны резервировать для себя достаточное количество страниц в стеке до входа в процедуры, критические по времени, чтобы системные вызовы не привели к сбою работы процесса.

Этого можно достичь путем вызова функции, которая содержит достаточно большой массив. В этот массив функция записывает данные, чтобы задействовать страницы памяти. Таким образом, стеку будет выделено достаточное количество страниц, и они будут заблокированы в ОЗУ. Запись в эти страницы предотвращает возможные ошибки типа **copy-on-write**, которые могут возникнуть при выполнении критичной по времени части программы.

Страницы, блокированные несколько раз при помощи функций **mlockall()** или **mlock()**, будут разблокированы одним вызовом **munlockall()**. Страницы, помещенные в несколько областей памяти или принадлежащие нескольким процессам, будут заблокированы в памяти до тех пор, пока они заблокированы хотя бы в одной из областей памяти или, по меньшей мере, одним процессом.

При удачном завершении вызова возвращается **0**.

При ошибке возвращается **-1**, а переменная errno устанавливается соответствующим образом.

ENOMEM — процесс попытался превысить максимальное заданное для него количество блокированных страниц. Обычным процессам (не-root) разрешено блокировать до их текущего **RLIMIT_MEMLOCK** ограничения ресурсов.

EPERM — вызывающий процесс не имеет соответствующих прав и привилегий. Процессам разрешено блокировать страницы, если они обладают возможностью **CAP_IPC_LOCK** (обычно она действительна только для root) или если их текущее ограничение ресурсов **RLIMIT_MEMLOCK** не равно нулю.

EINVAL — было задано неверное значение поля flags.

munlock() разрешает страничный обмен в областях памяти

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

int munlock(const void *addr, // начало области
            size_t      len); // размер области
```

munlock() разрешает страничный обмен в областях памяти, указание на которую начинается с адреса **addr** длиной **len** байтов.

Все страницы, содержащие часть заданной области памяти, могут быть помещены ядром во внешнюю область подкачки с помощью вызова **munlock()**.

Блокировка памяти не попадает в стек, т.е., страницы, блокированные несколько раз при помощи функций **mlock()** или **mlockall()**, будут разблокированы одним вызовом **munlock()** (с соответствующими параметрами) или **munlockall()**.

Страницы, помещенные в несколько областей памяти или принадлежащие нескольким процессам, будут заблокированы в памяти до тех пор, пока они заблокированы хотя бы в одной из областей памяти или одним процессом.

При удачном завершении вызова возвращаемое значение равно **0**.

При ошибке возвращается **-1**, переменной **errno** присваивается номер ошибки, и ни с одной из блокировок памяти ничего не произойдет.

munlockall() — разрешает обмен всех страниц памяти

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

int munlockall(void);
```

munlockall() разрешает обмен всех страниц памяти, находящихся в адресном пространстве вызывающего процесса.

Блокировка памяти не попадает в стек, т.е., страницы, блокированные несколько раз при помощи функций **mlock()** или **mlockall()**, будут разблокированы одним вызовом **munlock()** (с соответствующими параметрами) или **munlockall()**.

Страницы, помещенные в несколько областей памяти или принадлежащие нескольким процессам, будут заблокированы в памяти до тех пор, пока они заблокированы хотя бы в одной из областей памяти или одним процессом.

В POSIX-системах, в которых доступны **mlock()** и **munlock()**, в файле **<unistd.h>** задана константа **_POSIX_MEMLOCK_RANGE**.

При удачном завершении вызова возвращаемое значение равно **0**.

При ошибке оно равно **-1**, а переменной **errno** присваивается номер ошибки.

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.
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.

Файлы, отображаемые в памяти

С помощью системного вызова **open( )** операционная система отображает файл из **пространства имен** в **дисковое пространство** файловой системы, подготавливая почву для осуществления других операций.

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

ssize_t read(int fd, void  *buf, size_t count);  // fread( )
ssize_t write(int fd, void  *buf, size_t count); // fwrite( )
off_t   lseek(int fd, off_t offset, int whence); // fseek( ), ftell( ), fgetpos( )..
```

С появлением концепции виртуальной памяти, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии вычислительных систем, стало возможным **отображать файлы непосредственно в адресное пространство процессов**.

Иными словами, появилась возможность работать с файлами как с обычной памятью, заменив выполнение базовых операций над ними с помощью системных вызовов на использование операций обычных языков программирования.

Файлы, чье содержимое отображается непосредственно в адресное пространство процессов, получили название файлов, отображаемых в память, или, по-английски, **memory mapped** файлов.

Такое отображение может быть осуществлено не только для всего файла в целом, но и для его части.

С точки зрения программиста работа с такими файлами выглядит следующим образом:

1) Отображение файла из пространства имен в адресное пространство процесса происходит в два этапа — сначала выполняется отображение в дисковое пространство, и только затем возможно его отображение из дискового пространства в адресное.

Поэтому вначале файл необходимо открыть, используя обычный системный вызов **`open()`**.

2) Вторым этапом является отображение файла целиком или частично из дискового пространства в адресное пространство процесса.

Для этого используется системный вызов `mmap()`.

Файл после этого можно и закрыть, выполнив системный вызов **`close()`**, так как необходимая информация о расположении файла на диске сохранилась в других структурах данных при вызове **`mmap()`**.

3. После этого с содержимым файла можно работать, как с содержимым обычной области памяти.

4. По окончании работы с содержимым файла, необходимо освободить дополнительно выделенную процессу область памяти, предварительно **синхронизировав**, если это необходимо, содержимое файла на диске с содержимым этой области.

Эти действия выполняет системный вызов `munmap()`.

mmap()/munmap()

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

#ifdef _POSIX_MAPPED_FILES

void *mmap(void   *start,  // 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
           off_t   offset  // смещение в файле
);

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

Системный вызов **mmap( )** служит для отображения предварительно открытого файла, например, с помощью системного вызова **open( )**, в адресное пространство вычислительной системы.

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

После его выполнения файл может быть закрыт, например, системным вызовом close( ), что никак не повлияет на дальнейшую работу с отображенным файлом.

fd — корректный файловый дескриптор для файла, который мы хотим отобразить в адресное пространство, т.е. значением, которое вернул системный вызов **open()**.

start — адрес, с которого будет отображаться файл. Обычно используется **NULL**, при этом начало области отображения выбирает ОС и оно никогда не бывает равным **NULL**.

offset — смещение от начала файла в байтах. Должно быть кратно размеру страницы, получаемому при помощи **getpagesize()**[2].

length — размер отображаемой части файла в байтах. В память будет отображаться часть файла, начиная с позиции, заданной значением параметра **offset** и длиной **length**.

Значение параметра **length** можно указать и существенно большим, чем реальная длина от позиции **offset** до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал **SIGBUS**. Реакция на этот сигнал по умолчанию — прекращение процесса с образованием core dump файла.

flags — задает тип отражаемого объекта, способ отображения файла в адресное пространство (опции отражения) и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие, а также будут ли изменения перенесены в отображённый файл.

Состоит из комбинации следующих битов:

MAP_FIXED — не использовать другой адрес, если адрес задан в параметрах функции. Если заданный адрес не может быть использован, то функция **mmap()** вернет сообщение об ошибке. Если используется **MAP_FIXED**, то **start** должен быть кратен размеру страницы.

MAP_SHARED — использовать это отражение совместно с другими процессами, отражающими тот же объект. При записи информации в эту область изменения заносятся в отображённый файл.

Файл при этом может реально не обновляться до вызова функций **msync(2)** или **munmap(2)**., поэтому для более точного контроля над изменениями файла нужно использовать **msync(2)**.

---

2) В переносимых программах вместо getpagesize(2) следует использовать sysconf(_SC_PAGESIZE).

MAP_PRIVATE — создать не используемое совместно отражение с механизмом копирования при записи (COW – copy-on-write). Запись в эту область памяти не влияет на файл. Будут ли видимы в отображённой области изменения в файле, сделанные после вызова **`mmap()`**, не определено.

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.

prot — описывает желаемый режим защиты памяти, при этом он не должен конфликтовать с режимом открытия файла. Режим защиты памяти является либо **PROT_NONE** либо побитовым **ИЛИ** одного или нескольких флагов **PROT_***.

PROT_EXEC — данные в страницах могут исполняться;
PROT_READ — данные можно читать;
PROT_WRITE — в эту область можно записывать информацию;
PROT_NONE — доступ к этой области памяти запрещен.

Параметр **prot** определяет разрешенные операции над областью памяти, в которую будет отображен файл. Необходимо отметить две существенные особенности системного вызова, связанные с этим параметром:

Значение параметра prot не может быть шире, чем операции над файлом, заявленные при его открытии в параметре flags системного вызова open().

Например, нельзя открыть файл только для чтения, а при его отображении в память использовать значение **`prot = PROT_READ | PROT_WRITE`**.

Будет содержаться PROT_EXEC в PROT_READ или нет — зависит от архитектуры.

Портируемые программы должны всегда устанавливать **`PROT_EXEC`** если они намерены исполнить код в новом распределении.

MAP_NORESERVE — используется вместе с **MAP_PRIVATE**. Не выделяет страницы пространства подкачки для этого отображения.

Если пространство подкачки выделяется, то гарантируется возможность изменения (COW) отображения. Если же пространство подкачки не выделено, то при записи и отсутствии доступной памяти можно получить **SIGSEGV**.

MAP_LOCKED — (Linux 2.5.37 и выше) — блокировать страницу или размеченную область в памяти аналогично **mlock()**.

MAP_GROWSDOWN — используется для стеков. Для VM системы ядра обозначает, что отображение должно расширяться вниз по памяти.

MAP_ANONYMOUS — отображение не резервируется ни в каком файле, при этом аргументы **fd** и **offset** игнорируются.

MAP_32BIT — поместить размещение в первые 2Гб адресного пространства процесса.

Игнорируется, если указано **MAP_FIXED**. Этот флаг поддерживается только на x86-64 для 64-битных программ.

Замечания

Поле **st_atime** (время последнего доступа) отображаемого файла может быть обновлено в любой момент между вызовом **mmap( )** и соответствующим снятием отображения — первое же обращение к отображенной странице обновит это поле, если оно до этого уже не было обновлено.

Поля **st_ctime** (время изменения характеристик) и **st_mtime** (время изменения содержимого) файла, отображенного по **PROT_WRITE** и **MAP_SHARED**, будут обновлены после записи в отображенный диапазон до вызова последующего **msync()** с флагом **MS_SYNC** или **MS_ASYNC**, если такой случится.

Все, что отображено вызовом mmap(), сохраняется с теми же атрибутами при вызове fork().

Возвращаемые значения

При удачном выполнении **mmap()** возвращает указатель на область с отраженными данными. При ошибке возвращается **MAP_FAILED (-1)**, а переменная **errno** устанавливается в соответствующее значение.

При удачном выполнении **munmap()** возвращаемое значение равно нулю. При ошибке возвращается **-1**, а переменная **errno** устанавливается в соответствующее значение. (Вероятнее всего, это будет **EINVAL**).

В результате ошибки в операционной системе Linux при работе на 486-х и 586-х процессорах попытка записать в отображение файла, открытое только для записи, более 32-х байт одновременно приводит к ошибке — возникает сигнал о нарушении защиты памяти.