ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 13 — Файлы, отображаемые в памяти

Преподаватель: Поденок Леонид Петрович, 505a-5

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

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

Кафедра ЭВМ, 2023

2023.05.10

Оглавление

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()`.

Системный вызов mmap()

```
#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. 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()**.

length — размер отбражаемой 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. 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. Запись информации в эту область памяти будет эквивалентна записи в файл.

Файл при этом может реально не обновляться до вызова функций **msync(2)** или **munmap(2)**.

MAP_PRIVATE — создать не используемое совместно отражение с механизмом **copy-on-write**. Запись в эту область памяти не влияет на файл. При этом не определено, являются или нет изменения в файле после вызова **mmap()** видимыми в отраженном диапазоне.

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.

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.

Нестандартные флаги

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 сейчас поддерживается только на x86-64 для 64-битных программ.

Адрес **start** должен быть кратен размеру страницы. Все страницы, содержащие часть указанного диапазона, не отображаются и последующие ссылки на эти страницы будут генерировать **SIGSEGV**. Это не является ошибкой, если указанный диапазон не содержит отображенных страниц.

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.

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** 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:

- всех страниц сегментов кода, данных и стека;
- совместно используемых библиотек;
- пользовательских данных ядра;
- совместно используемой процессом памяти;
- 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:

- страницы не будут освобождены функциями **munlock()** или **munlockall()**;
- 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.

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.