Лекция 8
Управление памятью

Файлы, отображаемые в память (memory mapped file)

- Файл или его часть отображаются непосредственно в адресное пространство процесса
- Содержимое файла можно читать просто обращаясь в оперативную память
- При изменении данных в памяти они могут быть сохранены в файле
- Момент сохранения в файле выбирается ядром, но можно им управлять msync

Системный вызов mmap

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- start – желаемый адрес подключения к адресному пространству
- length – размер подключаемого блока памяти
- prot – флаги: PROT_EXEC, PROT_READ, PROT_WRITE
- fd – файловый дескриптор (-1 в некоторых случаях)
- offset – смещение в файле

Системный вызов munmap

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

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

- Отключает отображение с адреса addr размера length

Системный вызов mmap

- flags: MAP_SHARED — разделяемое отображение, изменения в памяти отображаются обратно в файл
- MAP_PRIVATE — неразделяемое отображение, copy-on-write
- MAP_ANONYMOUS — анонимное отображение (не соответствует никакому файлу)
- MAP_FIXED — не пытаться размещать отображение по адресу, отличному от start
- MAP_NORESERVE — не резервировать область подкачки (для отображений, допускающих запись)

Особенности mmap

- Гранулярность работы — одна страница памяти (x86 — 4KiB):
  - Размер length должен быть кратен размеру страницы
  - Смещение в файле offset должно быть кратно одной странице
  - Файл не должен быть пустым
- Хвост файла (< размера страницы) отображается на целую страницу, но размер не меняется
  - Чтение данных после конца файла вернет 0
  - Запись данных после конца файла не попадет в файл

Типичное использование

- MAP_SHARED — если несколько процессов отобразят файл, они будут видеть изменения друг друга, измененное содержимое будет сохранено в файле — реализация общей памяти (shared memory) процессов
- MAP_PRIVATE — содержимое файла доступно для чтения, при модификации содержимого другие процессы не увидят изменений, они не будут сохранены в файле — отображение исполняемых файлов в память

Типичное использование

- MAP_SHARED | MAP_ANONYMOUS — отображенная память доступна самому процессу и порожденным им процессам (они видят изменения) — реализация общей памяти для родственных процессов
- MAP_PRIVATE | MAP_ANONYMOUS — содержимое памяти видимо только для одного процесса — дополнительная память в адресном пространстве процесса

Реализация

- Mmap модифицирует таблицы отображения виртуальной памяти, но не загружает содержимое в ОЗУ (VSZ — меняется, RSS — нет)
- Содержимое файла загружается при обращении к отображенным адресам памяти (on demand loading)
- Если MAP_SHARED — измененные страницы помечаются 'dirty', и сбрасываются на диск (независимо от процесса) демоном очистки буферного кеша pdflush
- Если MAP_PRIVATE — измененные страницы клонируются для каждого процесса (copy-on-write) и переотображаются на swap-файл

Файл страничной подкачки (swap)

- Ядру может потребоваться освободить страницы физической памяти, занимаемые процессом, для использования в других целях
  - Немодифицированные страницы просто освобождаются
  - «Грязные» страницы MAP_SHARED сбрасываются на диск
  - Страницы MAP_ANONYMOUS, MAP_PRIVATE, страницы стека и кучи сохраняются в специальной области диска — файле подкачки (swap file)
- На Linux — это отдельный раздел диска

Резервирование swap

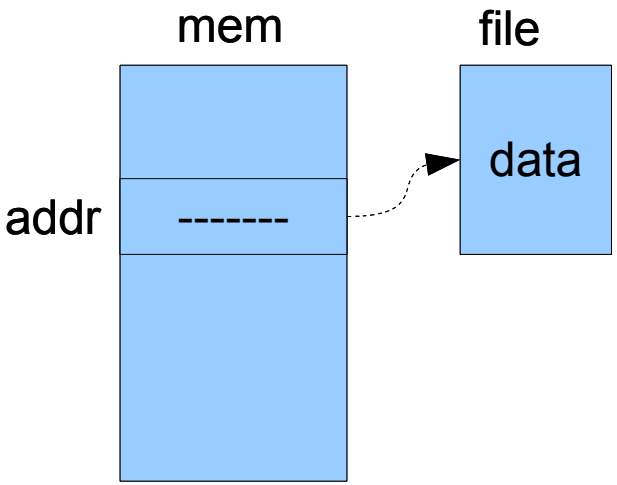
- Место в файле подкачки может быть зарезервировано при создании страницы, которую **может быть** потребуется сохранить в swap
  - Стек, куча
  - Все файлы, отображаемые в память с MAP_PRIVATE (т. е. исполняемые файлы и библиотеки) для каждого процесса
- В Linux место в файле подкачки выделяется при сохранении страницы в файле подкачки
- Возможны ситуации overcommit memory

Copy-on-write

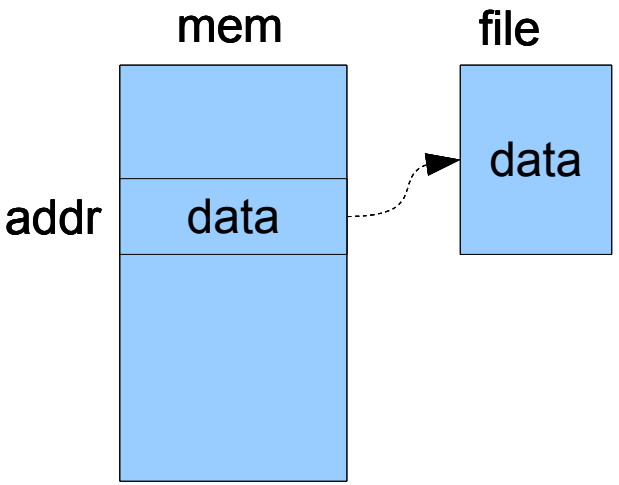
- Механизм оптимизации копирования страниц
- При обычном механизме копия страницы с выделением места в области подкачки создается немедленно
- При механизме copy-on-write создание копии страницы откладывается до первой записи в страницу

Copy-on-write

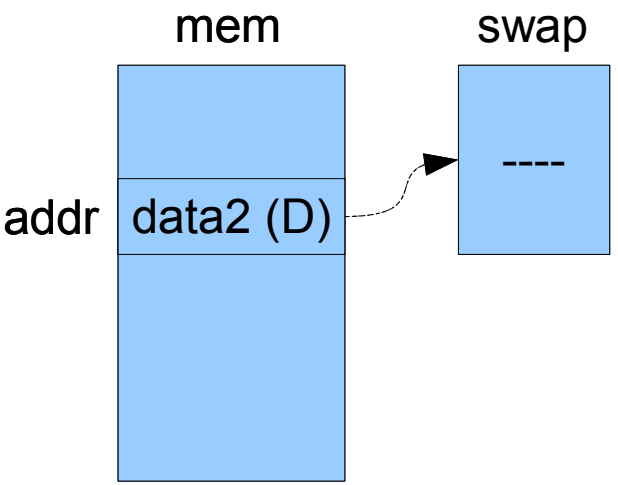
```
fd = open("file", O_RDWR, 0);
addr = mmap(0, 4096, PROT_READ|PROT_WRITE,MAP_PRIVATE,fd, 0);
```



При создании отображения страница в памяти помечена как отсутствующая, но отображенная на соответствующий файл



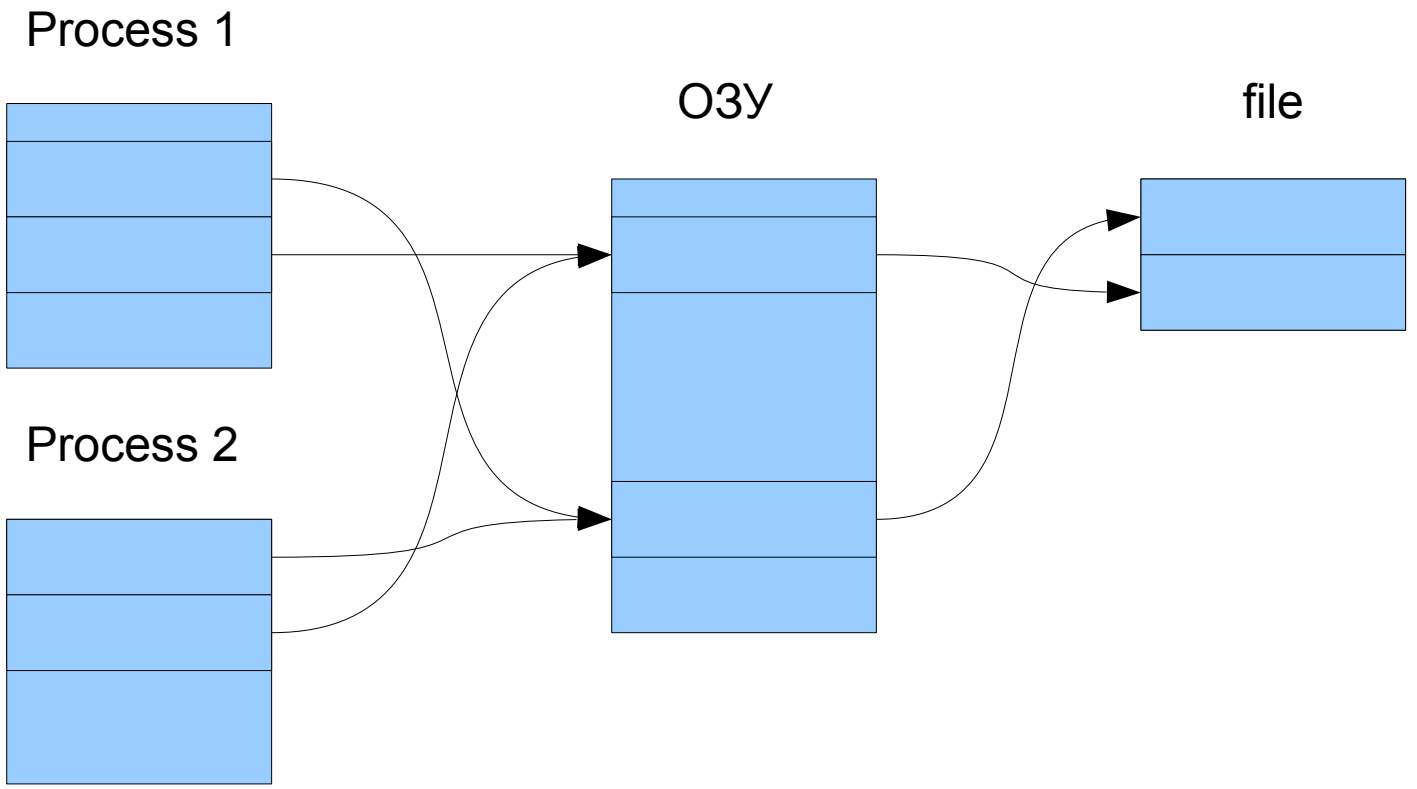
При чтении содержимое страницы подгружается из файла, страница помечается как «только для чтения»



При записи в страницу выделяется место в области подкачки, при необходимости создается копия страницы в ОЗУ, отображение переключается на swap

Разделение страниц между процессами

- Процессы, выполняющие отображение одного и того же файла, разделяют физические страницы ОЗУ



Необеспеченная память (memory overcommit)

- Стратегия выделения copy-on-write приводит к тому, что операция копирования страницы может быть выполнена в некоторый момент в будущем, причем прозрачным для выполняющегося процесса образом
- Когда потребуется создать копию страницы может оказаться, что память исчерпана и страница создана быть не может
- Необходимо снять с выполнения какой-нибудь процесс и таким образом освободить память (OOM killer)

OOM Killer

- Задача: выбрать минимальное число процессов, чтобы освободить максимальный объем памяти, но нанести минимальный ущерб системе
- Для каждого процесса вычисляется oom_score (/proc/${PID}/oom_score)
  - Чем больше RSS и Swap usage — тем хуже
  - Привилегированные процессы лучше обычных
  - Пользователь может задать поправку: /proc/${PID}/oom_score_adj

Загрузка файла на выполнение

- ELF-файл имеет структуру, оптимизированную для отображения файла mmap
- Секция кода (.text) отображается PROT_READ | PROT_EXECUTE, MAP_PRIVATE
- Константные данные (.rodata): PROT_READ, MAP_PRIVATE
- Данные (.data): PROT_READ | PROT_WRITE, MAP_PRIVATE
- Секции .text и .rodata у всех процессов, запущенных из одного файла, будут использовать одни и те же физические страницы памяти

Разделяемые библиотеки

- Позволяют избежать дублирования кода в процессах (например, все процессы имеют общую реализацию printf)
- Делает возможным разделять код библиотек между процессами разных исполняемых файлов (при статической компоновке реализация printf может располагаться по разным адресам, что делает невозможным разделение)
- Облегчают обновление ПО

Загрузка разделяемых библиотек

- ELF-файл содержит секцию .interp. Эта секция содержит путь к «интерпретатору» - /lib/ld-linux.so.2 — загрузчик динамических библиотек
- Загрузчик проходит по списку зависимостей библиотек, находит их в файловой системе и загружает в память, рекурсивно, пока все зависимости не будут удовлетворены
- Загрузка каждой библиотеки аналогична загрузке исполняемого файла (mmap)
- Но! Одна и та же библиотека может быть загружена в разных процессах по разным адресам

Позиционно-зависимый код

- printf("Hello");
- «Обычный» код

```
8048461: 68 50 84 04 08   push $0x8048450
8048466: e8 a5 fe ff ff   call 8048310 <printf@plt>
```

- В инструкции call используется смещение относительно адреса самой инструкции call, поэтому код может загружаться в любое место
- Инструкция push использует адрес строки "Hello" в памяти (`0x8048450`), если изменится адрес загрузки кода в память, изменится и этот адрес
- Скомпилированный код не может быть загружен в память по произвольному адресу без модификации

Секции перемещения

- ELF-файл может содержать секции перемещения (relocations), содержащие записи <Тип,Адрес в коде>

<R_386_32, 0x8048462>

- К значению по адресу 0x8048462 прибавляется смещение, на которое перемещен адрес загрузки, код адаптируется на новое размещение
- Достоинство: простота
- Недостаток: при MAP_PRIVATE если секция кода модифицируется, создается локальная для процесса копия — **теряется свойство разделения страниц физической памяти при использовании библиотеки в разных процессах**!

Procedure linkage table (PLT)

```
080483f0 <dynlink>:
 80483f0:       ff 35 04 a0 04 08       pushl  0x804a004
 80483f6:       ff 25 08 a0 04 08       jmp    *0x804a008
 80483fc:       00 00                   add    %al,(%eax)

08048410 <printf@plt>:
 8048410:       ff 25 10 a0 04 08  jmp   *0x804a010
 8048416:       68 08 00 00 00     push  $0x8
 804841b:       e9 d0 ff ff ff     jmp   80483f0 <dynlink>

0804a000 <_GLOBAL_OFFSET_TABLE_>:
 804a000:       14 9f 04 08        .int  0x8049f14 <_DYNAMIC>
 804a004:       00 00 00 00        .int  0
 804a008:       00 00 00 00        .int  0
 804a00c:       06 84 04 08        .int  0x8048406
 804a010:       16 84 04 08        .int  0x8048416
```

Lazy binding

- При первом вызове <plt@printf> управление попадет в динамический загрузчик. В стеке будет передано смещение на дескриптор загружаемой функции
- Динамический загрузчик запишет в GOT адрес функции printf в загруженной динамической библиотеке
- Все последующие вызовы будут передавать управление сразу на printf в динамической библиотеке

Позиционно-независимый код

```
000003d0 <dynlnk>:
 3d0: ff b3 04 00 00 00     pushl  0x4(%ebx)
 3d6: ff a3 08 00 00 00     jmp    *0x8(%ebx)

000003f0 <puts@plt>:
 3f0: ff a3 10 00 00 00     jmp    *0x10(%ebx)
 3f6: 68 08 00 00 00        push   $0x8
 3fb: e9 d0 ff ff ff        jmp    3d0 <dynlnk>

00000410 <__x86.get_pc_thunk.bx>:
 410: 8b 1c 24              mov    (%esp),%ebx
 413: c3                    ret
```

Позиционно-независимый код

```
00000550 <func>:
 550: 53                    push   %ebx
 551: e8 ba fe ff ff        call   410 <__x86.get_pc_thunk.bx>
 556: 81 c3 aa 1a 00 00     add    $0x1aaa,%ebx
 55c: 83 ec 14              sub    $0x14,%esp
 55f: 8d 83 84 e5 ff ff     lea    -0x1a7c(%ebx),%eax
 565: 50                    push   %eax
 566: e8 85 fe ff ff        call   3f0 <puts@plt>
 56b: 83 c4 18              add    $0x18,%esp
 56e: 5b                    pop    %ebx
 56f: c3                    ret
```