

Как Linux работает с памятью.

Исследование реализации виртуальной памяти в Linux

Stanislav levlev, Вторник, 20 Июль 2004, 00:20

Как Linux работает с памятью.

Случилось мне однажды поинтересоваться как же ядро работает с самым дорогим что у него есть, с памятью. Первые попытки разобраться с налету что и как ни к чему не привели. Не все так просто как хотелось бы. Отовсюду торчат концы, вроде все ясно, но как связать их воедино...

Возникла мысль обратиться к прошлому, чтобы по крайней мере разобраться как все это развивалось (версия 0.1). Это помогло понять и современное ядро. В дальнейшем речь пойдет о ядрах серии 2.2 об изменениях в 2.4 будет сообщено особо.

Не буду углубляться в тонкости функционирования защищенного режима процессора об этом написаны целые фолианты. Посмотрим только самую суть.

Итак, в основе всего лежат страницы памяти. В ядре они описываются структурой **mem_map_t**.

```
typedef struct page {
    /* these must be first (free area handling) */
    struct page *next;
    struct page *prev;
    struct inode *inode;
    unsigned long offset;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags; /* atomic flags, some possibly updated asynchronously */
    struct wait_queue *wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
} mem_map_t;
```

Уже тут наблюдается навороченность. Множество всяких ссылок. Все они используются. Одна страница может находиться в разных списках, например и в списке страниц в страничном кеше и в списке страниц относящихся к отображенному в память файлу (inode). В структуре описывающей последний можно найти и обратную ссылку, что очень удобно.

Все страницы адресуются глобальным указателем **mem_map**

`mem_map_t * mem_map`

Адресация страниц происходит очень хитро. Если раньше в структуре `page` было отдельное поле указывающее на физический адрес (`map_nr`), то теперь он вычисляется

```
static inline unsigned long page_address(struct page * page)
{
    return PAGE_OFFSET + PAGE_SIZE * (page - mem_map);
}
```

Свободные страницы хранятся в особой структуре **free_area**

```
static struct free_area_struct free_area[NR_MEM_TYPES][NR_MEM_LISTS];
```

, где первое поле отвечает за тип области : Ядра, Пользователя, DMA и.т.д. И обрабатываются по очень хитрому алгоритму.

Страницы делятся на свободные непрерывные области размера 2^x умноженной на размер страницы ($(2^x) * \text{PAGE_SIZE}$). Области одного размера лежат в одной области массива.

```
....
|-----
|Свободные Страницы размера PAGE_SIZE*4 ---> список свободных областей
|-----
|Свободные Страницы размера PAGE_SIZE*2 ---> список свободных областей
|-----
|Свободные Страницы размера PAGE_SIZE ---> список свободных областей
|-----
```

Выделяет страницу функция **get_free_pages(order)**. Она выделяет страницы составляющие область размера $\text{PAGE_SIZE} * (2^{\text{order}})$. Ищется область соответствующего размера или больше. Если есть только область большего размера то она делится на несколько маленьких и берется нужный кусок. Если свободных страниц недостаточно, то некоторые будут сброшены в область подкачки и процесс выделения начнется снова.

Возвращает страницу функция **free_pages(struct page, order)**. Высвобождает страницы начинающиеся с `page` размера $\text{PAGE_SIZE} * (2^{\text{order}})$. Область возвращается в массив свободных областей в соответствующую позицию и после этого происходит попытка объединить несколько областей для создания одной большего размера.

Отсутствие страницы в памяти обрабатывается ядром особо. Страница может или вообще отсутствовать или находиться в области подкачки.

Вот собственно и вся базовая работа с реальными страницами. Самое время вспомнить, что процесс работает все-таки с виртуальными адресами, а не с физическими. Преобразование происходит посредством вычислений, используя таблицы дескрипторов, и каталоги таблиц. Linux поддерживает 3 уровня таблиц: каталог таблиц первого уровня (PGD - Page Table Directory), каталог таблиц второго уровня (PMD - Medium Page Table Directory), и наконец таблица дескрипторов (PTE - Page Table Entry). Реально конкретным процессором могут поддерживаться не все уровни, но запас позволяет поддерживать больше возможных архитектур (Intel имеет 2 уровня таблиц, а Alpha - целых 3). Преобразование виртуального адреса в физический происходит соответственно в 3 этапа. Берется указатель PGD, имеющийся в структуре описывающий каждый процесс, преобразуется в указатель записи PMD, а последний преобразуется в указатель в таблице дескрипторов PTE. И наконец к реальному адресу указывающему на начало страницы прибавляют смещение от ее начала.

```
page_dir = pgd_offset(vma->vm_mm, address);
if (pgd_none(*page_dir))
    return;
if (pgd_bad(*page_dir)) {
    printk("bad page table directory entry %p: [%lx]\n", page_dir, pgd_val(*page_dir));
    pgd_clear(page_dir);
    return;
}
page_middle = pmd_offset(page_dir, address);
if (pmd_none(*page_middle))
    return;
if (pmd_bad(*page_middle)) {
    printk("bad page table directory entry %p: [%lx]\n", page_dir, pgd_val(*page_dir));
    pmd_clear(page_middle);
    return;
}
page_table = pte_offset(page_middle, address);
```

Вообще-то все данные об используемой процессом памяти помещаются в структуре **mm_struct**

```
struct mm_struct {
    struct vm_area_struct *mmap;          /* Список отображенных областей */
    struct vm_area_struct *mmap_avl;      /* Те же области но уже в виде дерева для более быс
того поиска*/
    struct vm_area_struct *mmap_cache;    /* Последняя найденная область*/
    pgd_t * pgd;                          /*Каталог таблиц*/
    atomic_t count;
    int map_count;                         /* Количество областей*/
    struct semaphore mmap_sem;
    unsigned long context;
```

Как Linux работает с памятью.

<http://www.osrc.info/plugins/content/content.php?content.51>

```
unsigned long start_code, end_code, start_data, end_data;

unsigned long start_brk, brk, start_stack;

unsigned long arg_start, arg_end, env_start, env_end;

unsigned long rss, total_vm, locked_vm;

unsigned long def_flags;

unsigned long cpu_vm_mask;

unsigned long swap_cnt; /* number of pages to swap on next pass */

unsigned long swap_address;

/*
 * This is an architecture-specific pointer: the portable
 * part of Linux does not know about any segments.
 */

void * segments;
```

Сразу замечаем, что помимо вполне понятных указателей на начало данных (`start_code`, `end_code` ...) кода и стека есть указатели на данные отображенных файлов (`mmap`). Это надо сказать особенность Linux - тащить в себя все что только можно. Может быть это и хорошо, но с другой стороны так разбазариваться памятью (вспомним еще буфера ввода/вывода при файловой системе, которые тоже будут кушать все новую память пока она есть) Данный подход может негативно отразиться на стабильности системы, ведь для запуска какого-то жизненно необходимого процесса может потребоваться время на освобождение лишних кешей. Простенькая проверка на потерю свободной памяти: введите команду `"cat /dev/mem >/image "` и посмотрите сколько свободной памяти после этого осталось. Если вам это не нравится, то обратите взгляд на функцию **`invalidate_inode_pages(* struct_inode)`**, освобождающую страничный кэш для данного файла.

При любом открытии файла, он сразу же отображается в память и добавляется в страничный кэш. Реальный же запрос на отображение файла только возвращает адрес на уже скешированные страницы.

На уровне процесса работа может вестись как со страницами напрямую, так и через абстрактную структуру **vm area struct**

```
struct vm_area_struct {
    struct mm_struct * vm_mm;    /* VM area parameters */
    unsigned long vm_start;
    unsigned long vm_end;

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;
    unsigned short vm_flags;

    /* AVL tree of VM areas per task, sorted by address */

```



short vm_avl_height;

struct vm_area_struct * vm_avl_left;
struct vm_area_struct * vm_avl_right;

/* For areas with inode, the list inode->i_mmap, for shm areas,
* the list of attaches, otherwise unused.
*/

struct vm_area_struct *vm_next_share;
struct vm_area_struct **vm_pprev_share;

struct vm_operations_struct * vm_ops;
unsigned long vm_offset;
struct file * vm_file;
unsigned long vm_pte; /* shared mem */

};

```
struct vm_operations_struct {  
    void (*open)(struct vm_area_struct * area);  
    void (*close)(struct vm_area_struct * area);  
    void (*unmap)(struct vm_area_struct *area, unsigned long, size_t);  
    void (*protect)(struct vm_area_struct *area, unsigned long, size_t, unsigned int newprot);  
    int (*sync)(struct vm_area_struct *area, unsigned long, size_t, unsigned int flags);  
    void (*advise)(struct vm_area_struct *area, unsigned long, size_t, unsigned int advise);  
    unsigned long (*nopage)(struct vm_area_struct * area, unsigned long address, int write_access);  
  
    unsigned long (*wppage)(struct vm_area_struct * area, unsigned long address,  
        unsigned long page);  
    int (*swapout)(struct vm_area_struct *, struct page *);  
    pte_t (*swabin)(struct vm_area_struct *, unsigned long, unsigned long);  
};
```

Идея данной структуры возникла из идеи виртуальной файловой системы, поэтому все операции над виртуальными областями абстрактны и могут быть специфичными для разных типов памяти, например при отображении файлов операции чтения одни а при отображении памяти (через файл /dev/mem) совершенно другие. Первоначально vm_area_struct появилась для обеспечения нужд отображения, но постепенно распространяется и на другие области.

Что делать, когда требуется получить новую область памяти. Есть целых 3 способа.

1. Уже знакомый **get_free_page()**
2. **kmalloc** - Простенькая (по возможностям, но отнюдь не коду) процедура с большими ограничениями по выделению новых областей и по их размеру.
3. **vmalloc** - Мощная процедура, работающая с виртуальной памятью, может выделять большие объемы памяти.

С каждой из двух процедур в ядре связаны еще по списку свободных/занятых областей, что еще больше усложняет понимание работы с памятью. (vmlist для vmalloc, kmem_cache для

Что же в 2.4.

Добавлена поддержка новой архитектуры памяти NUMA. В противовес классической UMA память делится на зоны с разным временем доступа к каждой из них. Это очень полезно и для кластерных решений. В связи с этим появились новые обертки на функции и найти суть стало еще сложнее. Появилась также поддержка памяти до 64Гб.

Раньше для всех файловых систем был один `generic_file_read` и `generic_file_mmap` в связи с тотальным засасыванием всего подряд в память при чтении (различия делались уже только на уровне `inode->readpage`). Теперь появился и `generic_file_write`. В общем еще пара таких `generic` и прощай виртуальная файловая система.

Но посмотрим - увидим. Ведь Linux развивается очень быстро и не всегда предсказуемо.