|  |  |  |
| --- | --- | --- |
| **Цилюрик О.И. Модули ядра Linux** | | |
| [Назад](http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-04.html) | **Внутренние механизмы ядра** | [Вперед](http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-06.html) |

**Слабовый распределитель**[5](http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-05.html" \l "sdfootnote5sym)

Текущее состояние слабового распределителя можем рассмотреть в файловой системе /proc (что даёт достаточно много для понимания самого принципа слабового распределения):

**$ cat /proc/slabinfo**

slabinfo - version: 2.1

# name <active\_objs> <num\_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount>

<sharedfactor> : slabdata <active\_slabs> <num\_slabs> <sharedavail>

...

kmalloc-8192 28 32 8192 4 8 : tunables 0 0 0 : slabdata 8 8 0

kmalloc-4096 589 648 4096 8 8 : tunables 0 0 0 : slabdata 81 81 0

kmalloc-2048 609 672 2048 16 8 : tunables 0 0 0 : slabdata 42 42 0

kmalloc-1024 489 512 1024 16 4 : tunables 0 0 0 : slabdata 32 32 0

kmalloc-512 3548 3648 512 16 2 : tunables 0 0 0 : slabdata 228 228 0

kmalloc-256 524 656 256 16 1 : tunables 0 0 0 : slabdata 41 41 0

kmalloc-128 13802 14304 128 32 1 : tunables 0 0 0 : slabdata 447 447 0

kmalloc-64 12460 13120 64 64 1 : tunables 0 0 0 : slabdata 205 205 0

kmalloc-32 12239 12800 32 128 1 : tunables 0 0 0 : slabdata 100 100 0

kmalloc-16 25638 25856 16 256 1 : tunables 0 0 0 : slabdata 101 101 0

kmalloc-8 11662 11776 8 512 1 : tunables 0 0 0 : slabdata 23 23 0

...

Сам принцип прост: сам слаб должен быть создан (зарегистрирован) вызовом kmem\_cache\_create(), а потом из него можно «черпать» элементы фиксированного размера (под который и был создан слаб) вызовами kmem\_cache\_alloc() (это и есть тот вызов, в который, в конечном итоге, с наибольшей вероятностью ретранслируется ваш kmalloc()). Все сопутствующие описания ищите в <linux/slab.h>. Так это выглядит на качественном уровне. А вот при переходе к деталям начинается цирк, который состоит в том, что прототип функции kmem\_cache\_create() меняется от версии к версии.

В версии 2.6.18 и **практически во всей литературе** этот вызов описан так:

kmem\_cache\_t \*kmem\_cache\_create( const char \*name, size\_t size,

size\_t offset, unsigned long flags,

void (\*ctor)( void\*, kmem\_cache\_t\*, unsigned long flags ),

void (\*dtor)( void\*, kmem\_cache\_t\*, unsigned long flags ) );

name — строка имени кэша;

size — размер элементов кэша (единый и общий для всех элементов);

offset — смещение первого элемента от начала кэша (для обеспечения соответствующего выравнивания по границам страниц, достаточно указать 0, что означает выравнивание по умолчанию);

flags — опциональные параметры (может быть 0);

ctor, dtor — **конструктор** и **деструктор**, соответственно, вызываются при размещении-освобождении каждого элемента, но с некоторыми ограничениями ... например, деструктор будет вызываться (финализация), но не гарантируется, что это будет поисходить сразу непосредственно после удаления объекта.

К версии 2.6.24 [5, 6] он становится другим (деструктор исчезает из описания):

struct kmem\_cache \*kmem\_cache\_create( const char \*name, size\_t size,

size\_t offset, unsigned long flags,

void (\*ctor)( void\*, kmem\_cache\_t\*, unsigned long flags ) );

Наконец, в 2.6.32, 2.6.35 и 2.6.35 можем наблюдать следующую фазу изменений (меняется прототип конструктора):

struct kmem\_cache \*kmem\_cache\_create( const char \*name, size\_t size,

size\_t offset, unsigned long flags,

void (\*ctor)( void\* ) );

Это значит, что то, что компилировалось для одного ядра, перестанет компилироваться для следующего. Вообще то, это достаточно обычная практика для ядра, но к этому нужно быть готовым, а при использовании таких достаточно глубинных механизмов, руководствоваться не навыками, а изучением заголовочных файлов текущего ядра.

Из флагов создания, поскольку они также находятся в постоянном изменении, и большая часть из них относится к отладочным опциям, стоит назвать:

SLAB\_HWCACHE\_ALIGN — расположение каждого элемента в слабе должно выравниваться по строкам процессорного кэша, это может существенно поднять производительность, но непродуктивно расходуется память;

SLAB\_POISON — начально заполняет слаб предопределённым значением (A5A5A5A5) для обнаружения выборки неинициализированных значений;

Если не нужны какие-то особые изыски, то нулевое значение будет вполне уместно для параметра flags.

Как для любой операции выделения, ей сопутствует обратная операция по уничтожению слаба:

int kmem\_cache\_destroy( kmem\_cache\_t \*cache );

Операция уничтожения может быть успешна (здесь достаточно редкий случай, когда функция уничтожения возвращает значение результата), только если уже **все** объекты, полученные из кэша, были возвращены в него. Таким образом, модуль должен проверить статус, возвращённый kmem\_cache\_destroy(); ошибка указывает на какой-то вид утечки памяти в модуле (так как некоторые объекты не были возвращены).

После того, как кэш объектов создан, вы можете выделять объекты из него, вызывая:

void \*kmem\_cache\_alloc( kmem\_cache\_t \*cache, int flags );

Здесь flags - те же, что передаются kmalloc().

Полученный объект должен быть возвращён когда в нём отпадёт необходимость :

void kmem\_cache\_free( kmem\_cache\_t \*cache, const void \*obj );

Несмотря на изменчивость API слаб алокатора, вы можете охватить даже диапазон версий ядра, пользуясь директивами условной трансляции препроцессора; модуль использующий такой алокатор может выглядеть подобно следующему (архив slab.tgz):

***slab.c :***

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/version.h>

MODULE\_LICENSE( "GPL" );

MODULE\_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );

MODULE\_VERSION( "5.2" );

static int size = 7; // для наглядности - простые числа

module\_param( size, int, 0 );

static int number = 31;

module\_param( number, int, 0 );

static void\* \*line = NULL;

static int sco = 0;

static

#if LINUX\_VERSION\_CODE > KERNEL\_VERSION(2,6,31)

void co( void\* p ) {

#else

void co( void\* p, kmem\_cache\_t\* c, unsigned long f ) {

#endif

\*(int\*)p = (int)p;

sco++;

}

#define SLABNAME "my\_cache"

struct kmem\_cache \*cache = NULL;

static int \_\_init init( void ) {

int i;

if( size < sizeof( void\* ) ) {

printk( KERN\_ERR "invalid argument\n" );

return -EINVAL;

}

line = kmalloc( sizeof(void\*) \* number, GFP\_KERNEL );

if( !line ) {

printk( KERN\_ERR "kmalloc error\n" );

goto mout;

}

for( i = 0; i < number; i++ )

line[ i ] = NULL;

#if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2,6,32)

cache = kmem\_cache\_create( SLABNAME, size, 0, SLAB\_HWCACHE\_ALIGN, co, NULL );

#else

cache = kmem\_cache\_create( SLABNAME, size, 0, SLAB\_HWCACHE\_ALIGN, co );

#endif

if( !cache ) {

printk( KERN\_ERR "kmem\_cache\_create error\n" );

goto cout;

}

for( i = 0; i < number; i++ )

if( NULL == ( line[ i ] = kmem\_cache\_alloc( cache, GFP\_KERNEL ) ) ) {

printk( KERN\_ERR "kmem\_cache\_alloc error\n" );

goto oout;

}

printk( KERN\_INFO "allocate %d objects into slab: %s\n", number, SLABNAME );

printk( KERN\_INFO "object size %d bytes, full size %ld bytes\n", size, (long)size \* number );

printk( KERN\_INFO "constructor called %d times\n", sco );

return 0;

oout:

for( i = 0; i < number; i++ )

kmem\_cache\_free( cache, line[ i ] );

cout:

kmem\_cache\_destroy( cache );

mout:

kfree( line );

return -ENOMEM;

}

module\_init( init );

static void \_\_exit exit( void ) {

int i;

for( i = 0; i < number; i++ )

kmem\_cache\_free( cache, line[ i ] );

kmem\_cache\_destroy( cache );

kfree( line );

}

module\_exit( exit );

А вот как выглядит выполнение этого размещения (картина весьма поучительная, поэтому остановимся на ней подробнее):

**$ sudo insmod ./slab.ko**

**$ dmesg | tail -n300 | grep -v audit**

allocate 31 objects into slab: my\_cache

object size 7 bytes, full size 217 bytes

constructor called 257 times

**$ cat /proc/slabinfo | grep my\_**

# name <active\_objs> <num\_objs> <objsize> ...

my\_cache 256 256 16 256 1 : tunables 0 0 0 : slabdata 1 1 0

**$ sudo rmmod slab**

Итого: объекты размером 7 байт благополучно разместились в новом слабе с именем my\_cache, отображаемом в /proc/slabinfo, организованным с размером элементов 16 байт (эффект выравнивания?), конструктор при размещении 31 таких объектов вызывался 257 раз. Обратим внимание на чрезвычайно важное обстоятельство: при создании слаба никаким образом не указывается реальный или максимальный объём памяти, находящейся под управлением этого слаба: это динамическая структура, «добирающая» столько страниц памяти, сколько нужно для поддержания размещения требуемого числа элементов данных (с учётом их размера). Увеличенное число вызовов конструктора можно отнести: а). на необходимость переразмещения существующих элементов при последующих запросах, б). эффекты SMP (2 ядра) и перераспределения данных между процессорами. Проверим тот же тест на однопроцессорном Celeron и более старой версии ядра:

**$ uname -r**

2.6.18-92.el5

**$ sudo /sbin/insmod ./slab.ko**

**$ /sbin/lsmod | grep slab**

slab 7052 0

**$ dmesg | tail -n3**

allocate 31 objects into slab: my\_cache

object size 7 bytes, full size 217 bytes

constructor called 339 times

**$ cat /proc/slabinfo | grep my\_**

# name <active\_objs> <num\_objs> <objsize> ...

my\_cache 31 339 8 339 1 : tunables 120 60 8 : slabdata 1 1 0

**$ sudo /sbin/rmmod slab**

Число вызовов конструктора не уменьшилось, а даже возросло, а вот размер объектов, под который создан слаб, изменился с 16 на 8.

***Примечание:****Если рассмотреть 3 первых поля вывода /proc/slabinfo, то и в первом и во втором случае видно, что под слаб размечено некоторое фиксированное количество фиксированных объекто-мест (339 в последнем примере), которые укладываются в некоторый начальный объём слаба меньше или порядка 1-й страницы физической памяти.*

А вот тот же тест при больших размерах объектов и их числе:

**$ sudo insmod ./slab.ko size=1111 number=300**

**$ dmesg | tail -n3**

allocate 300 objects into slab: my\_cache

object size 1111 bytes, full size 333300 bytes

constructor called 330 times

**$ sudo rmmod slab**

**$ sudo insmod ./slab.ko size=1111 number=3000**

**$ dmesg | tail -n3**

allocate 3000 objects into slab: my\_cache

object size 1111 bytes, full size 3333000 bytes

constructor called 3225 times

**$ sudo rmmod slab**

***Примечание:****Последний рассматриваемый пример любопытен в своём поведении. Вообще то «завалить» операционную систему Linux — ничего не стоит, когда вы пишете модули ядра. В противовес тому, что за несколько лет плотной (почти ежедневной) работы с микроядерной операционной системой QNX мне так и не удалось её «завалить» ни разу (хотя попытки и предпринимались). Это, попутно, к цитировавшемуся ранее эпиграфом высказыванию Линуса Торвальдса относительно его оценок микроядерности. Но сейчас мы не о том... Если погонять показанный тест с весьма большим размером блока и числом блоков для размещения (заметно больше показанных выше значений), то можно наблюдать прелюбопытную ситуацию: нет, система не виснет, но распределитель памяти настолько активно отбирает память у системы, что постепенно угасают все графические приложения, потом и вся подсистема X11 ... но остаются в живых чёрные текстовые консоли, в которых даже живут мыши. Интереснейший получается эффект**[6](http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-05.html" \l "sdfootnote6sym).*

Ещё одна вариация на тему распределителя памяти, в том числе и слаб-алокатора — механизм пула памяти:

#include <linux/mempool.h>

mempool\_t \*mempool\_create( int min\_nr,

mempool\_alloc\_t \*alloc\_fn,

mempool\_free\_t \*free\_fn,

void \*pool\_data );

Пул памяти сам по себе вообще не является алокатором, а всего лишь является **интерфейсом** к алокатору (к тому же кэшу, например). Само наименование «пул» (имеющее схожий смысл в разных контекстах и разных операционных системах) предполагает, что такой механизм будет всегда поддерживать «в горячем резерве» некоторое количество объектов для распределения. Аргумент вызова min\_nr является тем минимальным числом выделенных объектов, которые пул должен всегда поддерживать в наличии. Фактическое выделение и освобождение объектов по запросам обслуживают alloc\_fn() и free\_fn(), которые предлагается написать пользователю, и которые имеют такие прототипы:

typedef void\* (\*mempool\_alloc\_t)( int gfp\_mask, void \*pool\_data );

typedef void (\*mempool\_free\_t)( void \*element, void \*pool\_data );

Последний параметр mempool\_create() - pool\_data передаётся последним параметром в вызовы alloc\_fn() и free\_fn().

Но обычно просто дают обработчику-распределителю ядра выполнить за нас задачу — объявлено (<linux/mempool.h>) несколько групп API для разных распределителей памяти. Так, например, существуют две функции, например, (mempool\_alloc\_slab() и mempool\_free\_slab()), ориентированный на рассмотренный уже слаб алокатор, которые выполняют соответствующие согласования между прототипами выделения пула памяти и kmem\_cache\_alloc() и kmem\_cache\_free(). Таким образом, код, который инициализирует пул памяти, который будет использовать слаб алокатор для управления памятью, часто выглядит следующим образом:

// создание нового слаба

kmem\_cache\_t \*cache = kmem\_cache\_create( ... );

// создание пула, который будет распределять память из этого слаба

mempool\_t \*pool = mempool\_create( MY\_POOL\_MINIMUM, mempool\_alloc\_slab, mempool\_free\_slab, cache );

После того, как пул был создан, объекты могут быть выделены и освобождены с помощью:

void \*mempool\_alloc\_slab( gfp\_t gfp\_mask, void \*pool\_data );

void mempool\_free\_slab( void \*element, void \*pool\_data );

После создания пула памяти функция выделения будет вызвана достаточное число раз для создания пула предопределённых объектов. После этого вызовы mempool\_alloc\_slab() пытаются получить новые объекты от функции выделения - возвращается один из предопределённых объектов (если таковые сохранились). Когда объект освобождён mempool\_free\_slab(), он сохраняется в пуле если количество предопределённых объектов в настоящее время ниже минимального, в противном случае он будет возвращён в систему.

***Примечание:****Такие же группы API есть для использования в качестве распределителя памяти для пула kmalloc() (mempool\_kmalloc()) и страничного распределителя памяти (mempool\_alloc\_pages()).*

Размер пула памяти может быть динамически изменён:

int mempool\_resize( mempool\_t \*pool, int new\_min\_nr, int gfp\_mask );

- в случае успеха этот вызов изменяет размеры пула так, чтобы иметь по крайней мере new\_min\_nr объектов.

Когда пул памяти больше не нужен он возвращается системе:

void mempool\_destroy( mempool\_t \*pool );

[5](http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-05.html" \l "sdfootnote5anc)) В литературе (публикациях) мне встречалось русскоязычное наименование такого распределителя как: «слабовый», «слябовый», «слэбовый»... Поскольку термин нужно как-то именовать, а ни одна из транскрипций не лучше других, то я буду пользоваться именно первым произношением из перечисленных.

[6](http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-05.html" \l "sdfootnote6anc)) Что напомнило высказывание классика отечественного юмора М. Жванецкого: «А вы не пробовали слабительное со снотворным? Удивительный получается эффект!».