Мои заметки про zram и память в ядре

Литература

- Linux. Системное программирование (Роберт Лав), глава 12, 14, 15, 16 – с этого начинать, книга в целом лёгкая на подъем (но нужны какие-то базовые знания по ядру). Указаные главы конкретно про память, лучше читать с начала.
- Драйверы Устройств Linux, 3-я редакция – сложно, но надо
- Ядро Linux (Бовет Д., Чезати М.) – пока не читал. Говорят немного устаревшее, но раскрываются основные идеи памяти в ядре
- In-kernel memory compression – https://lwn.net/Articles/545244/
- zram docs – https://www.kernel.org/doc/Documentation/blockdev/zram.txt

Kernel memory

Всякое разное про пейджы

struct page

Физическая страница памяти описывается структурой `struct page`. В ядре есть большой линейный массив, который содержит все структуры. То есть, существует некоторые массив хранящий **описание** (это важно) всех физических страниц. И к этим **описаниям** можно обратиться. Но, не каждый struct page можно замапить на виртуальную память - то есть описание существует страницы существует всегда, но вот обратиться к этой странице можно не всегда. Это зависит от состояния системы и от всяких других тонких моментов.

huge pages

huge pages - это пейджы, которые больше 4кб. Аллоцировать их можно только на boottime, например передав загрузчику параметры для ядра. Они непрерывные. Например, если аллоцировать страницу на 1 гигабайт - будет залочен 1 гигабайт физический памяти под эту страницу. Описывается так же с помощью struct page.

Поддерживает это не каждый процессор, и при этом должна быть включён специальный параметр в ядре. `/sys/kernel/mm/hugepages/` - всякая инфа про них в системе

idle pages

У `struct page` есть поле `_count` - счётчик ссылок на страницу. Если на страницу никто не ссылается, то счётчик равен 0 или отрицательный (неожиданно), поэтому чтобы проверить, используется ли сейчас страница, например, каким-нибудь процессом, нужно вызывать `page_count()` - если 0, то страница такая страница называется idle (никем не используется), если != 0 то значит где-то используется.

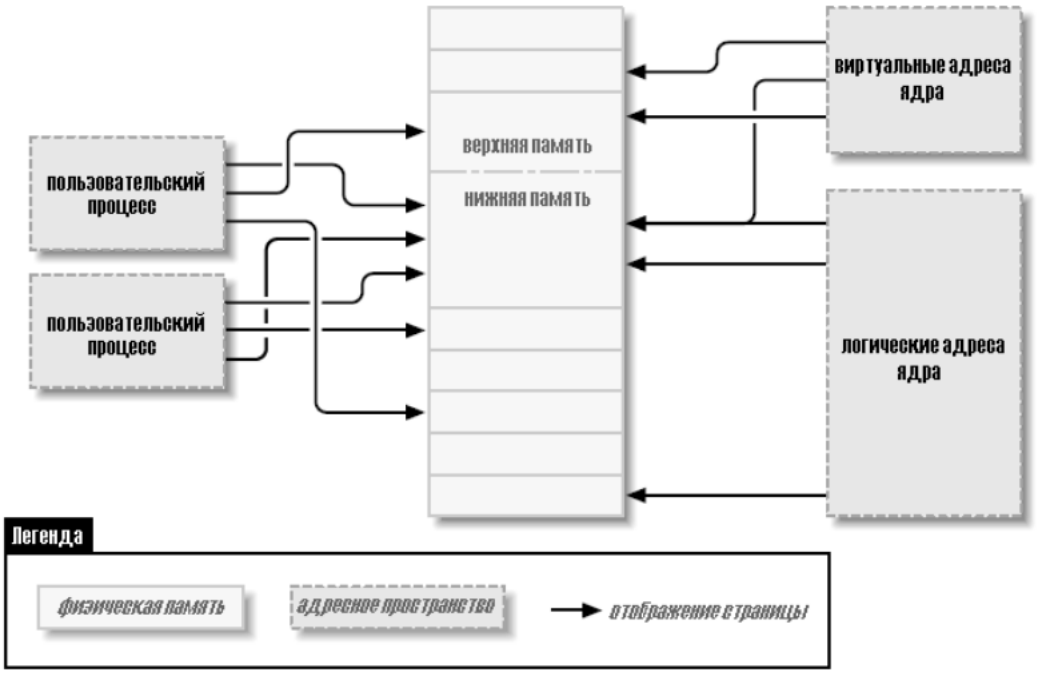
Адресация, mmap и прочее

mmap

mmap – системный вызов с помощью которого мапится память устройства в пользовательскую память.

У каждого устройства есть драйвер. Если разработчик устройства хочет, чтобы его память можно было замапить, он должен самостоятельно реализовать mmap для своего устройства и для этого есть всякое разное API.

Типы адресов



- Пользовательские виртуальные адреса - это понятно;
- физические адреса - это тоже;
- адреса шин - адреса, используемые между периферийными шинами и памятью. До конца не понятно, что это такое;

- логические адреса ядра - это сложно;
- виртуальные адреса ядра - и это тоже.

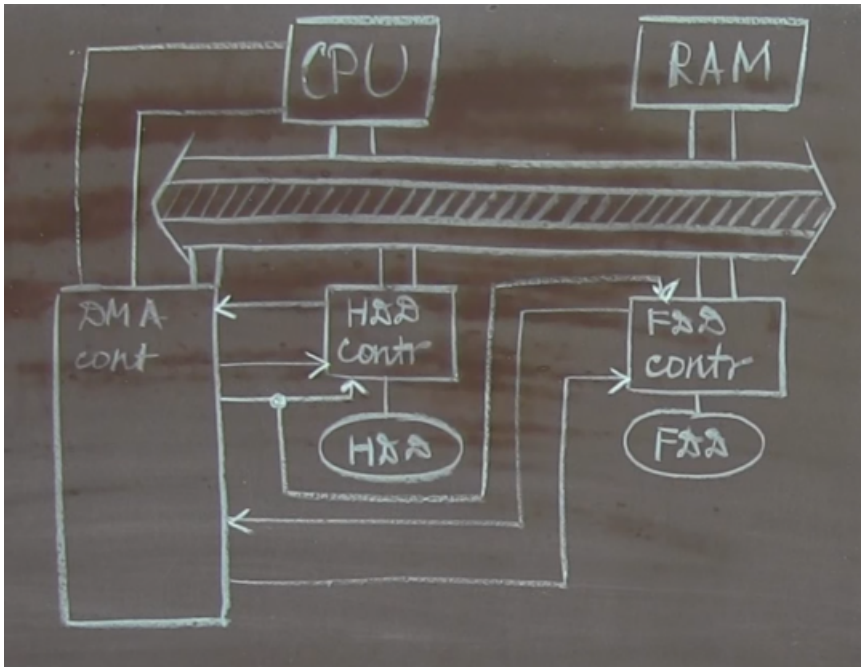
Суть в том что это всё абстракции над одним и тем же куском железа (памятью).

Нижняя и верхняя память

- Нижняя память – память, для которой существуют логические адреса в пространстве ядра;
- верхняя память – память для которой логические адреса не существуют, потому что она выходит за рамки диапазона адресов, отведённых для виртуальных адресов ядра.

DMA

Что это вообще такое?



Когда мы хотим прочитать что-то с устройства (речь о блочных устройствах):

- CPU обращается к контроллеру устройства;
- читает данные;
- кладёт данные на шину;
- по шине они идут в RAM.

Direct memory access (DMA) – грубо говоря, с помощью этой штуки можно обращаться к памяти без использования CPU. Существует DMA controller – некоторая физическая штука, которая ставится в устройстве вместе со всем остальным. Происходит это примерно так (на примере чтения данных, для записи тоже самое):

- DMA controller шлёт запрос к CPU о том что он хочет почитать устройство;
- CPU отсылает сигнал - что-то типа "окей, можешь читать";
- выставляются какие-то сигналы между контроллером DMA и контроллером устройства (это не так важно);
- DMA controller начинает читать данные и ставить их на шину;
- нужно отметить, что шина заблокирована для CPU, и он туда писать ничего не может;
- когда DMA controller поставил на шину весь блок данных, они уже идут в RAM;
- обмен сигналами между CPU и DMA controller что чтение успешно завершено (чтоб CPU уже знал, что ему можно если что вдруг лезть на шину памяти)

В ядре

todo

zRam

zram – модуль ядра, позволяющий жать страницы памяти прямо в оперативной памяти (в отличи от zswap, который жмёт те страницы, которые будут отправлены в swap устройство). Жмёт он только страницы размеров 4096, о чём говорит вот этот кусок кода (zram_drv.c):

```
src = zs_map_object(zram->mem_pool, handle, ZS_MM_RO);
if (size == PAGE_SIZE) {
  dst = kmap_atomic(page);
  memcpy(dst, src, PAGE_SIZE);
  kunmap_atomic(dst);
  ret = 0;
} else {
  dst = kmap_atomic(page);
  ret = zcomp_decompress(zstrm, src, size, dst);
  kunmap_atomic(dst);
  zcomp_stream_put(zram->comp);
}
```

Как он работает

zsmalloc

У zram есть свой аллокатор – zsmalloc. Хорошо работает в условиях нехватки памяти. Особенности:

- аллоцировать объекты он может только размера не больше PAGE_SIZE - если запросить больше, то `zs_malloc()` возвращает ошибку;
- `zs_malloc()` **не возвращет** указатель, который можно разыменовать. Он возвращает некоторый дескриптор (unsinged long), который описывает аллоцированный объект;
- чтобы сконвертить этот дескриптор в область памяти, в которую уже можно писать/читать, нужно вызывать `zs_map_object()`. Это связано с тем, что на 32-битных машинах virtual address area сильно ограничен (память короче экономят на древних машинах).

Структуры данных

Драйвер описывается структурой `struct zram`:

```
1  struct zram {
2    struct zram_table_entry *table;
3    struct zs_pool *mem_pool;
4    struct zcomp *comp;
5    struct gendisk *disk;
6    /* Prevent concurrent execution of device init */
7    struct rw_semaphore init_lock;
8    /*
9    * the number of pages zram can consume for storing compressed data
10   */
11   unsigned long limit_pages;
12
13   struct zram_stats stats;
14   /*
15   * This is the limit on amount of *uncompressed* worth of data
16   * we can store in a disk.
17   */
18   u64 disksize; /* bytes */
19   char compressor[CRYPTO_MAX_ALG_NAME];
20   /*
21   * zram is claimed so open request will be failed
22   */
23   bool claim; /* Protected by disk->open_mutex */
24  #ifdef CONFIG_ZRAM_WRITEBACK
25   struct file *backing_dev;
26   spinlock_t wb_limit_lock;
27   bool wb_limit_enable;
28   u64 bd_wb_limit;
29   struct block_device *bdev;
30   unsigned long *bitmap;
31   unsigned long nr_pages;
32  #endif
33  #ifdef CONFIG_ZRAM_MEMORY_TRACKING
34   struct dentry *debugfs_dir;
35  #endif
36 };
```

```
struct zram_table_entry {
  union {
    unsigned long handle;
    unsigned long element;
  };
  unsigned long flags;
  #ifdef CONFIG_ZRAM_MEMORY_TRACKING
  ktime_t ac_time;
  #endif
};
```

- каждой странице памяти соответствует структура zram_table_entry, структура zram хранит массив этих структур – **table**. Сама структура zram_table_entry состоит из handle (дескриптор zs_malloc, см. выше) или element: кейс, когда страница состоит из одного и того же символа – в таком случае хранится просто этот элемент. Ну и различные флаги (flags)
- mem_pool –
- zcomp – dynamic per-device compression frontend. Штука через которую происходят все махинации с сжатием.
- disk –
- limit_pages –
- disksize – ???
- claim – ???

Функция `__zram_bvec_write()`

В этой функции происходит вся магия сжатия страницы и сохранения сжатого буфера. Алгоритм работы этой функции можно разделить на несколько кейсов:

1. Страница состоит из одного и того же элемента.

```
mem = kmap_atomic(page);
if (page_same_filled(mem, &element)) {
  kunmap_atomic(mem);
  /* Free memory associated with this sector now. */
  flags = ZRAM_SAME;
  atomic64_inc(&zram->stats.same_pages);
  goto out;
}
```

Функция `page_same_filled()` проверяет, состоит ли страница из одного и того же элемента. Если так, то:

```
zram_slot_lock(zram, index);
zram_free_page(zram, index);

zram_set_flag(zram, index, flags);
zram_set_element(zram, index, element);

zram_slot_unlock(zram, index);

/* Update stats */
atomic64_inc(&zram->stats.pages_stored);
```

- функция `zram_free_page()` обрабатывает всякие ситуации, когда не нужно хранить сжатую страницу. В нашем кейсе это из-за того что страница состоит из одного и того же элемента. Внутри функция (в нашем кейсе) ничего не делает.
- ставит флаг, что страничка состоит из одного и того же элемента
- выставляет этот элемент

NOTE: я убрал из оригинального кода строчки, которые не выполняются в этом кейсе.

2. Обычная страница

```
zstrm = zcomp_stream_get(zram->comp);
src = kmap_atomic(page);
ret = zcomp_compress(zstrm, src, &comp_len);
kunmap_atomic(src);
```

Тут просто жмут страницу.

```
if (comp_len >= huge_class_size)
  comp_len = PAGE_SIZE;
```

У `zs_malloc` есть huge class или non-huge class объекты. Если сжатый буффер больше чем размер huge class, значит буффер (сжатую страницу) нельзя сохранить в аллокаторе, поэтому тут делают пометку (2 строчка), что будем хранить просто не сжатую страницу (см. ниже).

Я думаю (но это надо проверить), что huge_class_size равен PAGE_SIZE.

```
/*
* handle allocation has 2 paths:
* a) fast path is executed with preemption disabled (for
*  per-cpu streams) and has __GFP_DIRECT_RECLAIM bit clear,
*  since we can't sleep;
* b) slow path enables preemption and attempts to allocate
*  the page with __GFP_DIRECT_RECLAIM bit set. we have to
*  put per-cpu compression stream and, thus, to re-do
*  the compression once handle is allocated.
```

```
10 *
11 * if we have a 'non-null' handle here then we are coming
12 * from the slow path and handle has already been allocated.
13 */
14
15 if (!handle)
16   handle = zs_malloc(zram->mem_pool, comp_len,
17     __GFP_KSWAPD_RECLAIM |
18     __GFP_NOWARN |
19     __GFP_HIGHMEM |
20     __GFP_MOVABLE);
21
22 if (!handle) {
23   zcomp_stream_put(zram->comp);
24   atomic64_inc(&zram->stats.writestall);
25   handle = zs_malloc(zram->mem_pool, comp_len,
26     GFP_NOIO | __GFP_HIGHMEM |
27     __GFP_MOVABLE);
28   if (handle)
29     goto compress_again;
30     return -ENOMEM;
31 }
```

Первый раз попытка аллоцировать объект с выключенным вытеснением: это должно работать быстрее, поэтому они и называют это `fast path`. Но, видимо, не всегда аллокатор может сработать с выключенным вытеснением (не могу пока понять, в каком случае так может быть). Если не получилось, идём в аллокатор еще раз, при этом вытеснения разрешаются. Если удалось, то, **не понятно почему, жмут страницу опять** (самый верхний листинг).

```
1 alloced_pages = zs_get_total_pages(zram->mem_pool);
2 update_used_max(zram, alloced_pages);
3
4 if (zram->limit_pages && alloced_pages > zram->limit_pages) {
5   zcomp_stream_put(zram->comp);
6   zs_free(zram->mem_pool, handle);
7   return -ENOMEM;
8 }
```

У zram есть параметр, отвечающий за то, сколько максимум может быть сжато страниц. Тут проверяется на то что, **на данный момент** это ограничение не достигнуто. Почему эта проверка находится не в начале функции? Потому что в начале они могут быть, а пока идет сжатие и прочие процессы, лимит уже может быть достигнут (из-за параллельности).

```
1 dst = zs_map_object(zram->mem_pool, handle, ZS_MM_WO);
2
3 src = zstrm->buffer;
4 if (comp_len == PAGE_SIZE)
5   src = kmap_atomic(page);
```

```
6  memcpy(dst, src, comp_len);
7  if (comp_len == PAGE_SIZE)
8    kunmap_atomic(src);
9
10 zcomp_stream_put(zram->comp);
11 zs_unmap_object(zram->mem_pool, handle);
12 atomic64_add(comp_len, &zram->stats.compr_data_size);
```

Дескриптор аллокатора мапится в память, и сжатый буффер (aka `zstrm->buffer`) копируется в аллоцированный объект. Если страница сжалась с коэффициентом <= 1 (то есть размер не изменился или увеличился), в аллоцированный объект записывают не сжатую страницу, а оригинальную: во-первых, если размер сжатой страницы больше, то записать ее в аллоцированный объект нельзя в принципе (т.к zsmalloc объект может быть максимум `PAGE_SIZE`). Во-вторых, если даже размер "сжатой"страницы **равен** `PAGE_SIZE`, то, как минимум, придётся вызывать decompress.

```
1  zram_slot_lock(zram, index);
2  zram_free_page(zram, index);
3
4  if (comp_len == PAGE_SIZE) {
5    zram_set_flag(zram, index, ZRAM_HUGE);
6    atomic64_inc(&zram->stats.huge_pages);
7    atomic64_inc(&zram->stats.huge_pages_since);
8  }
9
10 zram_set_handle(zram, index, handle);
11 zram_set_obj_size(zram, index, comp_len);
12 zram_slot_unlock(zram, index);
13
14 /* Update stats */
15 atomic64_inc(&zram->stats.pages_stored);
```

Кусок практически тот же, что и в кейсе один (см. выше). Единственное, что в случае если страница с размером PAGE_SIZE (не сжатая), выставляют всякую статистику и флаг для `zram_table_entry`.

Функция `__zram_bvec_read()`

TODO

Параметры при сборке

ZRAM_MEMORY_TRACKING

Добавляет дополнительную статистику по allocated blocks states в debug ноде: `/sys/kernel/debug/zram/zramX/block_state`

ZRAM_WRITEBACK

TODO