Аллокации в linux: от new до физической памяти

Created by: Пресняков Арсений и Бережной Аким

2023-12-28

Содержание

Глава 1. Введение

Целью данной работы является разбор дерева вызовов, которые получит пользователь при запросе на выделении памяти. Задача дойти от вызова new до основ ядра операционной система. Мы рассмотрим как это происходит в C/C++. **Аллокатор** - функция, динамически распределяющая память для пользователя, которая выделяет или освобождает память при необходимости. При вызове new происходит вызов функции **malloc(size_t)** из glibc.h, которая является одним из представителей «семейства аллокаторов» и относится к general-purpose аллокатором(аллокатор общего предназначения, который подходит для решения любых задач). Существует огромное количество алгоритмов распределения памяти, преследующих различные цели. Они выбирают компромиссы между основными требованиями к аллокатору такими как:

Максимальная мобильность

Использование как можно меньшего количества системно-зависимых функций (таких как системные вызовы), при этом обеспечивая дополнительную поддержку других полезных функций, доступных только в некоторых системах; соответствие всем известным системным ограничениям на правила выравнивания и адресации.

Минимизация пространства

Аллокатор не должен тратить пространство впустую: он должен получать как можно меньше памяти от системы и поддерживать память таким образом, чтобы минимизировать фрагментацию — «дыры» в смежных участках памяти, которые не используются программой.

Минимизация времени

Подпрограммы **malloc**(), **free**() и **realloc**() в среднем должны работать как можно быстрее.

Максимизация возможностей настройки

Дополнительные функции и поведение должны контролироваться пользователями либо статически (с помощью макросов и т.п.), либо динамически (с помощью команд управления, таких как mallopt).

Максимизация локальности

Выделение фрагментов памяти, которые обычно используются вместе, рядом друг с другом. Это помогает свести к минимуму промахи страниц и кэша во время выполнения программы.

Максимизация обнаружения ошибок

Аллокторы должны предоставлять средства для обнаружения повреждений из-за перезаписи памяти, многократного освобождения и т. д.

Минимизация аномалий

Аллокатор, использующий настройки по умолчанию, должен хорошо работать в широком диапазоне реальных нагрузок, которые сильно зависят от динамического распределения — приложения с графическим интерфейсом, компиляторы, интерпретаторы, инструменты разработки, программы с интенсивным использованием сети (пакетов), пакеты с интенсивным использованием графики, веб-браузеры, приложения для обработки строк и т. д.

Глава 2. malloc()

2.1. Основные структуры

Спускаемся ниже. Теперь рассмотрим реализацию **malloc**() в glibc.h. Рассмотрим используемый здесь алгоритм. При небольшом количестве потоков каждый поток получает свой **struct malloc_state**. Сделано это для скорости работы, поскольку тогда нет необходимости в примитивах синхронизации. В случае очень большого количества потоков к одному **malloc_state** будут иметь доступ несколько потоков, поэтому внутри ему необходим mutex. Далее для простоты будем рассматривать только однопоточные программы. В них будет только один **malloc_state**, который называется **main_arena**. Рассмотрим **malloc_state**:

```
/* malloc.c */
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Set if the fastbin chunks contain recently inserted free blocks.  */
  /* Note this is a bool but not all targets support atomics on booleans.  */
  int have_fastchunks;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;

  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;

  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* Linked list */
  struct malloc_state *next;

  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;

  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */
  INTERNAL_SIZE_T attached_threads;

  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

В нем много непонятных нам полей. Рассмотрим основные. **mfastbinptr** и **mchunkptr** - разные typdef одного struct'а **malloc_chunk**:

```c
/* malloc.c */
typedef struct malloc_chunk* mchunkptr;
```

```c
/* malloc.c */
typedef struct malloc_chunk *mfastbinptr;
```

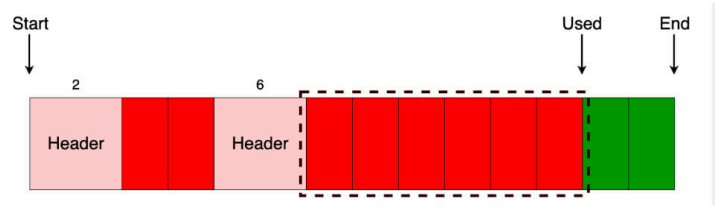
```c
/* malloc.c */
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

Это основообразующая структура всего аллокатора. Так называемый «Header» перед contigious блоком памяти. Это выглядит примерно вот так:



Чтобы разобраться в его полях давайте введем два понятия: свободный блок и занятый блок.

Свободный блок - contigious блок памяти, которым владеет **malloc**(он уже есть в виртуальном адресном пространстве программы), но который еще не выделен для пользователя, или блок, к которому был применен **free**().

Занятый блок - contigious блок памяти, который был передан пользователю с помощью **malloc**.

Структура **malloc_chunk** хранит размер предыдущего блока(который лежит contigious перед ним) - **mchunk_prev_size**, размер текущего блока - **mchunk_size**. Размер каждого чанка кратен 8 байтам, следовательно последние 3 бита поля **mchunk_size** не важны для определения размера и их можно использовать под флажки:

- A (0x04)

Allocated Arena - **main_arena** использует heap программы. Другие **malloc_state** аллоцируют память под их собственную кучу с помощью системного вызова **mmap()**. Если значения бита 0, этот чанк относится к **main_arena**, иначе этот блок из куска памяти, выделенного **mmap()**'ом для другого **malloc_state**.

- M (0x02)

MMap'd chunk - этот блок был выделен для с помощью системного вызова **mmap()** и не является частью кучи. При его освобождении вызывается парный вызов - **munmap()**.

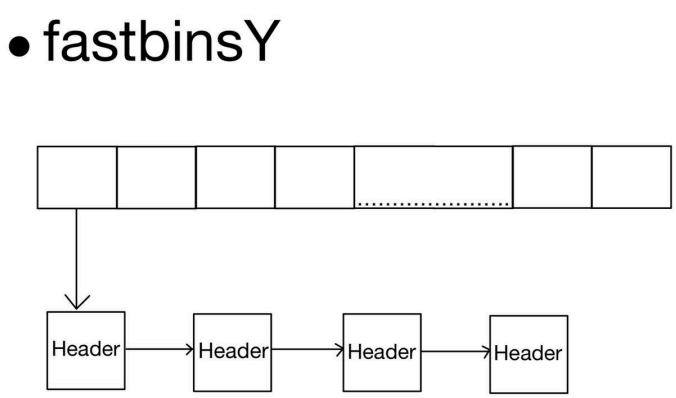
- P (0x01)

Previous chunk is in use - предыдущий блок занят. Этот флаг нам нужен для объединения двух соседних свободных чанков в один.

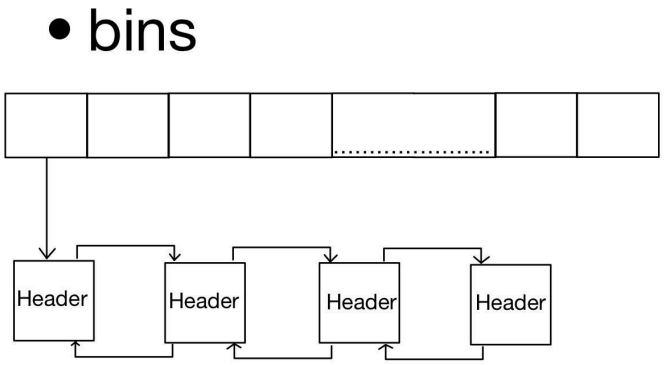
Далее поделим блоки на быстрые, маленькие и большие. Их границы зависят от **sizeof(size_t)** (будем считать его равным 8). Быстрые ($\leq$ 160 байт) – кэшируются.

Вернемся к **malloc_state** и рассмотрим **fastbinsY**. Это статический массив поинтеров на **malloc_chunk**, которые указывают на головы односвязных списков, хранящих равные по размеру быстрые **malloc_chunk**. Эти спис-

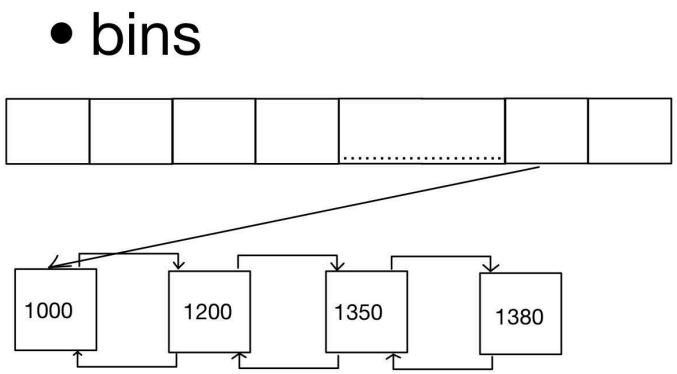
ки работают по принципу LIFO(то есть последний попавший в него блок будет лежать в голове списка), что повышает вероятность нахождения адреса в TLB-кэше при запросе на выделение из этого списка. Так же есть специальный макрос, позволяющий моментально найти список с подходящим размером блоков, что экономит значительное время, которое было бы потрачено на поиск.



Теперь рассмотрим как хранятся блоки маленького размера($\leq$ 512 байт). Свободные чанки одинакового размера хранятся в двусвязном списке. В **malloc_chunk** поле **fd** - указатель на следующий элемент, а поле **bk** на предыдущий. Массив указателей на головы этих списков также хранится в **malloc_state**, а именно в поле **bins**. Под хранение маленьких блоков выделены первые 64 ячейки данного массива. Блоки выделяются по FIFO. Здесь мы все еще имеем быстрый доступ к спискам с помощью #define, однако чанки требуют уже куда более долгой и тяжелой обработки.



В больших блоках(все остальные) мы начинаем использовать последние два поля **malloc_chunk** - **fd_nextsize** и **bk_nextsize**. Оставшиеся 64 ячейки массива **bins** выделяются под указатели со списками на большие чанки. Мы отходим от концепции хранения списков с одинаковым размером блоков. Теперь мы храним двусвязные списки с заранее известным диапазоном значений, а блоки в них отсортированы - **fd_nextsize** ссылается на минимальный блок имеющий больший или равный нам размер, а **bk_nextsize**, соответственно, на максимальный блок с меньшим или равным размером.



Помимо этого, есть еще один двусвязный список, хранящий блоки маленького и большого размера в не отсортированном порядке. Из него достаются блоки по принципу FIFO и добавляются в **bins**.

2.2. int_malloc()

Теперь, когда мы знакомы с тем, как хранятся **malloc_chunk**, мы можем перейти к разбору алгоритма **int_malloc()**, внутренней функции, которая вызывается в **malloc()**. Мы не будем вдаваться во все подробности, опустив некоторые детали(например, оптимизацию TCACHE в glibc). Наша задача – понять основные принципы работы алгоритма:

- При запросе на аллокацию менее 160 байт мы находим индекс в массиве **fastbinY** и выделяем голову этого односвязного списка, новая голова - следующий за ним блок.
- При запросе на аллокацию среднего блока (160 - 512 байт) мы получаем в **bins** индекс двусвязного списка с блоком нужного нам размера. Если такой список пуст, то обращаемся к списку, хранящему не отсортированные блоки, и пробуем среди них найти подходящий, параллельно добавляя блоки в соответствующие списки **bins**. Если мы нашли подходящий блок то его мы и вернем, иначе возьмем больший блок и разделим его на два: один, который мы займем и остаток. Остаток добавим в список свободных блоков.
- При аллокации более 512 байт мы очищаем **fastbinY**, объединяя свободные соседние блоки в один. Это необходимо для более эффективного использования памяти и борьбы с фрагментацией. Затем мы добавляем один чанк из списка неотсортированных блоков в **bins** и пробуем достать подходящий блок из соответсвующего размеру запроса списка из **bins**. Если он пуст, то пробуем найти подходящий чанк в неотсортированном списке.
- Запрос не полуичилось удовлетворить. В таком случае вызывается вспомогательная функция **sysmalloc()**.

Снова вернемся к структуре **malloc_state** и посмотрим на поле **top**. Оно хранит последний доступный блок памяти(если представить память contigious, то самый правый блок, указатель на конец нашего heap), он не лежит в **bins**. Это некий указатель на конец доступной нам памяти, и когда мы не можем удовлетворить запрос пользователя необходимо обратиться к операционной системе за новой порцией. В этом нам поможет функция **sysmalloc()** Рассмотрим алгоритм ее работы более подробно в следующем пункте.

2.3. sysmalloc()

Как было оговорено выше, **sysmalloc()** обрабатывает те случаи, когда от системы требуется выделить несколько больше памяти, чем предусмотрено в обработке случаев с небольшими аллокациями. Рассмотрим алгоритм работы **sysmalloc()** на примере его исходного кода:

```
/* malloc.c */
  if (av == NULL ||
      ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_threshold)
       && (mp_.n_mmaps < mp_.n_mmaps_max)))
    {
      char *mm;
      if (mp_.hp_pagesize > 0 && nb >= mp_.hp_pagesize)
        {
          /* There is no need to issue the THP madvise call if Huge Pages are
             used directly.  */
          mm = sysmalloc_mmap (nb, mp_.hp_pagesize, mp_.hp_flags, av);
          if (mm != MAP_FAILED)
            return mm;
        }
      mm = sysmalloc_mmap (nb, pagesize, 0, av);
      if (mm != MAP_FAILED)
        return mm;
      tried_mmap = true;
    }
```

Сначала смотрим на трешхолд (константу, которая ограничивает то минимальное количество байт, когда мы делаем **sysmalloc_mmap()** – обёрточную функцию, вызывающую **mmap()**) и, если он достигнут, пользуемся **sysmalloc_mmap()**. Здесь стоит сделать оговорку, что код написан с учётом так называемых **Transparent Huge Pages (THP)** – альтернативой **HugeTLB** (в современных версиях OS, они включены по умолчанию). Здесь используются два подхода: **system-wide** и **per-process**. При **system-wide** подходе, система автоматически пытается назначить большие страницы любому процессу, когда есть на то возможность, и процесс использует большие непрерывные участки виртуальной памяти. В случае с **per-process**, ядро отдает большие страницы отдель-

ным процессам в те участки памяти, которые специфицирует системный вызов **madvise()**. Пользователь может управлять этим поведением используя `/sys/kernel/mm/transparent_hugepage/enabled`. В этот файл могут быть записаны такие опции, как:

- `always` - для **system-wide** распределения;
- `madvise` - для **per-process** распределения;
- `never` - для отключения **THP**.

Далее, в случае, когда доступных арен нет и **sysmalloc_mmap()** провалился, возвращаем пустой указатель:

```
/* malloc.c */
/* There are no usable arenas and mmap also failed.  */
if (av == NULL)
  return 0;
```

Далее рассчитываем количество памяти, которое нам может потребоваться, учитывая выравнивание и минимальный порог, записывая его в **size**. Мы также заранее надеемся на то, что память, запрашиваемая нами сейчас попадет в contigious кусок сразу после прошлого запроса на **sysmalloc()**, если таковой был. Поэтому, мы вычитаем тот размер, который мы могли уже выделить на куче до текущего вызова:

```
/* malloc.c */
size = nb + mp_.top_pad + MINSIZE;

if (contiguous (av))
  size -= old_size;
```

Здесь **old_size** – это округленный **top** из арены, которая подалась аргументом в данный системный вызов.

После того, как мы посчитали размер, учитываем **THP** оптимизацию и выравниваем память по размеру большой страницы. Иначе - по размеру обычной:

```
/* malloc.c */
#ifdef MADV_HUGEPAGE
      /* Defined in brk.c.  */
      extern void *__curbrk;
      if (__glibc_unlikely (mp_.thp_pagesize != 0))
      {
        uintptr_t top = ALIGN_UP ((uintptr_t) __curbrk + size,
                mp_.thp_pagesize);
        size = top - (uintptr_t) __curbrk;
      }
      else
#endif
        size = ALIGN_UP (size, GLRO(dl_pagesize));
```

В случае, когда размер посчитался корректно, вызываем **sbrk()**. На самом деле, в дальнейшем коде вызывается функция **MORECORE()**, но вы можете убедиться, что это тот же **sbrk()**, просто обёрнутый в другую функцию. Название изменено семантически, так как **MORECORE()** используется при добавление микро-кусочков памяти на куче при устранении проблем с выравниванием и сторонними вызовами **sbrk()**:

```
/* morecore.c */
void * __default_morecore (ptrdiff_t increment)
{
  void *result = (void *) __sbrk (increment);
  if (result == (void *) -1)
    return NULL;

  return result;
}
```

Если **sbrk()** провалился, нам не удалось выделить contigious кусок и мы пытаемся использовать **sysmalloc_mmap_fallback()** (функцию, похожую на **sysmalloc_mmap()**, но предназначенную для случаев, когда **sbrk()** не смог выделить память), чтобы выделить размер большой страницы, если это возможно или, если и такой вызов провалится, хотя бы кусок требуемого размера:

```
/* malloc.c */
if (size > 0)
{
  brk = (char *) (MORECORE (size));
  if (brk != (char *)(MORECORE_FAILURE))
    madvise_thp (brk, size);
  LIBC_PROBE (memory_sbrk_more, 2, brk, size);
}

if (brk == (char *) (MORECORE_FAILURE))
{
  char *mbrk = MAP_FAILED;
  if (mp_.hp_pagesize > 0)
    mbrk = sysmalloc_mmap_fallback (&size, nb, old_size, mp_.hp_pagesize,
                                    mp_.hp_pagesize, mp_.hp_flags, av);
  if (mbrk == MAP_FAILED)
    mbrk = sysmalloc_mmap_fallback (&size, nb, old_size,
                                    MMAP_AS_MORECORE_SIZE, pagesize, 0, av);
  if (mbrk != MAP_FAILED)
  {
    /* We do not need, and cannot use, another sbrk call to find end */
    brk = mbrk;
    snd_brk = brk + size;
  }
}
```

Если же **sbrk()** не провалился, то всё гораздо лучше: мы можем расширить кучу и получить кусок памяти, который лежит непрерывно с тем куском, что мы могли аллоцировать ранее (кэшу теперь будет приятно). При этом, не забываем передвинуть указатель на голову кучи подальше:

```
/* malloc.c */
if (mp_.sbrk_base == 0)
  mp_.sbrk_base = brk;
av->system_mem += size;
if (brk == old_end && snd_brk == (char *) (MORECORE_FAILURE))
  set_head (old_top, (size + old_size) | PREV_INUSE);
```

Если же **brk** стал меньше чем **old_end**, то произошло что-то странное: где-то произошла ошибка и мы ходим выйти как можно скорее. Выводим это в поток ошибок:

```
/* malloc.c */
else if (contiguous (av) && old_size && brk < old_end)
  /* Oops!  Someone else killed our space..  Can't touch anything.  */
  malloc_printerr ("break adjusted to free malloc space");
```

Остался третий случай: когда все указатели в корректном состоянии, но у нас остались проблемы с выравниванием. Далее в исходном коде идет большой пласт ифоф, которые отвечают только за то, чтобы поправить выравнивание в куче различных случаев. Для того, чтобы не захламлять общее понимание работы **sysmalloc()**, эту деталь мы опустим.

Итак, осталось только проверить, что соблюлись все инварианты и отдать указатель на память:

```
/* malloc.c */
if ((unsigned long) av->system_mem > (unsigned long) (av->max_system_mem))
```

```
  av->max_system_mem = av->system_mem;
check_malloc_state (av);

/* finally, do the allocation */
p = av->top;
size = chunksize (p);

/* check that one of the above allocation paths succeeded */
if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE))
  {
    remainder_size = size - nb;
    remainder = chunk_at_offset (p, nb);
    av->top = remainder;
    set_head (p, nb | PREV_INUSE | (av != &main_arena ? NON_MAIN_ARENA : 0));
    set_head (remainder, remainder_size | PREV_INUSE);
    check_malloced_chunk (av, p, nb);
    return chunk2mem (p);
  }

/* catch all failure paths */
__set_errno (ENOMEM);
return 0;
```

Подводя итог, разребем работу **sysmalloc()** поэтапно:

1. Смотрим на трэшхолд:
   - если перевалили его, вызываем **sysmalloc_mmap()** и при успешном завершении → см. пункт 3
   - если нет, пробуем вызвать **sbrk()**, чтобы выделить contigious кусок памяти
2. Как отработал **sbrk()**?
   - провалился → вызываем **sysmalloc_mmap_fallback()** и при успешном завершении → см. пункт 3
   - отработал без ошибок → учитываем выравнивания и идем дальше
3. Удостоверяемся, что память выделяется успешно: проверяем выравнивания, количество получившейся памяти, инварианты указателей и **malloc_state**:
   - если всё ок → отдаем указатель на выделенную память
   - если где-то ошибка, возвращаем пустой указатель и ставим флажок `ENOMEM`, если память кончилась.

Глава 3. Системные вызовы brk() и mmap().

Теперь рассмотрим как работают **sbrk()** и **mmap()** в linux. Для начала дадаим определение:

Регион памяти - contigious кусок памяти лежащий не на куче и выделяемый с помощью системного вызова **mmap()**.

Познакомимся с двумя структурами:

vm_area_struct(vma) - структура, хранящая информацию о регионе памяти. Ее основные поля:
- vm_start, vm_end - начало и конец зоны памяти, соответственно
- vm_next, vm_prev - vm_area_struct каждого процесса завязаны в двусвязный список

mm_struct - структура, включающая в себя все vm_area_struct, относящиеся к данному процессу. Ее основные поля:
- mmap - указатель на голову двусвязного списка **vm_area_struct**
- mmap_cache - lru **vm_area_strcut**
- mmlist - список всех **mm_struct**
- start_brk, brk - начало и конец кучи, соответственно
- start_stack - начало стека

Ниже приведен код со всеми полями данной структуры(это не настоящий код ядра linux, здесь добавлены комментарии и вырезаны некоторые детали, усложняющие читаемость кода):

```
/* linux/mm_types.h */
struct mm_struct {
        struct vm_area_struct  *mmap;              /* list of memory areas */
        struct rb_root         mm_rb;              /* red-black tree of VMAs */
        struct vm_area_struct  *mmap_cache;        /* last used memory area */
        unsigned long          free_area_cache;    /* 1st address space hole */
        pgd_t                  *pgd;               /* page global directory */
        atomic_t               mm_users;           /* address space users */
        atomic_t               mm_count;           /* primary usage counter */
        int                    map_count;          /* number of memory areas */
        struct rw_semaphore    mmap_sem;           /* memory area semaphore */
        spinlock_t             page_table_lock;    /* page table lock */
        struct list_head       mmlist;             /* list of all mm_structs */
        unsigned long          start_code;         /* start address of code */
        unsigned long          end_code;           /* final address of code */
        unsigned long          start_data;         /* start address of data */
        unsigned long          end_data;           /* final address of data */
        unsigned long          start_brk;          /* start address of heap */
        unsigned long          brk;                /* final address of heap */
        unsigned long          start_stack;        /* start address of stack */
        unsigned long          arg_start;          /* start of arguments */
        unsigned long          arg_end;            /* end of arguments */
        unsigned long          env_start;          /* start of environment */
        unsigned long          env_end;            /* end of environment */
        unsigned long          rss;                /* pages allocated */
        unsigned long          total_vm;           /* total number of pages */
        unsigned long          locked_vm;          /* number of locked pages */
        unsigned long          def_flags;          /* default access flags */
        unsigned long          cpu_vm_mask;        /* lazy TLB switch mask */
        unsigned long          swap_address;       /* last scanned address */
        unsigned               dumpable:1;         /* can this mm core dump? */
        int                    used_hugetlb;       /* used hugetlb pages? */
        mm_context_t           context;            /* arch-specific data */
        int                    core_waiters;       /* thread core dump waiters */
        struct completion      *core_startup_done; /* core start completion */
        struct completion      core_done;          /* core end completion */
        rwlock_t               ioctx_list_lock;    /* AIO I/O list lock */
        struct kioctx          *ioctx_list;        /* AIO I/O list */
```

```
39          struct kioctx           default_kioctx;      /* AIO default I/O context */
40  };
```

3.1. sbrk()

Сейчас, когда мы познакомились с основными структурами управления виртуальной памятью в ядре, мы можем перейти к разбору кода функции **sbrk()** в linux.

```
1  /* linux/tools/include/nolibc/sys.h */
2  void *sys_brk(void *addr)
3  {
4    return (void *)my_syscall1(__NR_brk, addr);
5  }
6
7  void *sbrk(intptr_t inc)
8  {
9    /* first call to find current end */
10   void *ret = sys_brk(0);
11
12   if (ret && sys_brk(ret + inc) == ret + inc)
13     return ret + inc;
14
15   SET_ERRNO(ENOMEM);
16   return (void *)-1;
17 }
```

sbrk() дважды вызывает **sys_brk()** - обертку над системным вызовом **brk()**, который пробует заменить конец кучи процесса(поле **brk** структуры **mm_struct**) на переданный ему указатель. Первым вызовом **sys_brk(0)** мы получаем текущий **brk**, вторым вызовом **sys_brk(ret + inc)** мы пробуем увеличить **brk** на **inc** и, в случае неудачи, возвращаем ошибку. Теперь рассмотрим реализацию системного вызова **brk()**:

```
1  /* linux/mm/nommu.c */
2  SYSCALL_DEFINE1(brk, unsigned long, brk)
3  {
4    struct mm_struct *mm = current->mm;
5
6    if (brk < mm->start_brk || brk > mm->context.end_brk)
7      return mm->brk;
8
9    if (mm->brk == brk)
10     return mm->brk;
11
12   /*
13    * Always allow shrinking brk
14    */
15   if (brk <= mm->brk) {
16     mm->brk = brk;
17     return brk;
18   }
19
20   /*
21    * Ok, looks good - let it rip.
22    */
23   flush_icache_user_range(mm->brk, brk);
24   return mm->brk = brk;
25 }
```

Мы получаем указатель на **mm_struct** текущего процесса. Далее, если принятый указатель меньше начала кучи(**start_brk**), больше максимально возможного конца кучи(**context.end_brk**) или равен текущему **brk** мы не меняем **brk** и возвращаем его. Таким образом, **sys_brk(0)** возвращает текущий **brk**. Данный системный вызов

допускает уменьшение размера кучи, поэтому, если принятый указатель больше **start_brk** и меньше **brk** мы уменьшим текущий **brk**. В единственном оставшемся случае наш указатель увлечит **brk**, расширив объем кучи, и вернет его. Это **brk()** делает, вызывая функцию **flush_icache_user_range()**, которая вызывает аллокатор физических старниц памяти, о котором мы расскажем в следующей главе. Так, **sbrk()** понимает, что системный вызов **brk()** сработал ожидаемым образом, увеличив кучу.

3.2. mmap()

```
/* mm/mmap.c */
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Обратите внимание, что здесь и далее слово "регион" и словосочетание "участок памяти" будут пониматься как полные синонимы.

Системный вызов **mmap()** создаёт новый регион памяти внутри адресного пространства процесса. Конкретно на архитектуре x86, происходит вызов цепочки функций **sys_mmap2()** → **do_mmap2()** → **do_mmap_pageoff()**, последняя из которых и делает всю работу, при этом являясь общей для всех архитектур.

Сначала выполняются проверки входных параметров: мы проверяем корректность используемой файловой системы, убеждаемся, что размер запрашиваемой страницы выровнен и мы не пытаемся выделить страницы в адресном пространстве ядра. Помимо этого, мы проверяем, не слишком ли много на данный момент у процесса уже выделенных страниц.

После вышеперечисленных этапов подготовки происходят следующие шаги:

1. Поиск свободного куска адресного пространства, достаточно большого, чтобы быть использованным в рамках данной аллокации
2. Проверки флагов доступа выбранного региона
3. Аллокация **vm_area_struct** (структура описана выше) с помощью slab-аллокатора.
4. Провязывание страницы в **VMA**

Далее будут рассмотрены несколько процессов, необходимые для полного понимания работы **mmap**.

Первый – нахождение региона памяти, уже выделенного **mmap()**:

Для того, чтобы найти **VMA**, к которому принадлежит определенный адрес, например, во время таких операций, как **page_fault**, обычно используется функция **find_vma()**.

```
/* mm/mmap.c */
struct vm_area_struct *
find_vma(struct mm_struct * mm, unsigned long addr)
{
  struct vm_area_struct *vma = NULL;

  if (mm) {
    /* Check the cache first. */
    /* (Cache hit rate is typically around 35%.) */
    vma = mm->mmap_cache;
    if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
      rb_node_t * rb_node;

      rb_node = mm->mm_rb.rb_node;
      vma = NULL;

      while (rb_node) {
        struct vm_area_struct * vma_tmp;

        vma_tmp = rb_entry(rb_node,
        struct vm_area_struct, vm_rb);

```

```
        if (vma_tmp->vm_end > addr) {
          vma = vma_tmp;
          if (vma_tmp->vm_start <= addr)
            break;
          rb_node = rb_node->rb_left;
        } else
          rb_node = rb_node->rb_right;
      }
      if (vma)
        mm->mmap_cache = vma;
    }
  }
  return vma;
}
```

Сначала она проверяет поле кэша **mmap**, которое запоминает результат последнего вызова **find_vma()**, поскольку вполне вероятно, что один и тот же регион может потребоваться несколько раз подряд. Если это не желаемый регион, то совершается поиск по красно-черному дереву (в поле **mm_rb**). Если желаемый адрес не содержится ни в одном **VMA**, функция аллоцирует с помощью физического аллокатора (см. Главу 4) и возвращает вернет **VMA**, ближайший к запрошенному адресу, поэтому вызывающей стороне важно перепроверить, что возвращаемый **VMA** содержит переданный адрес. Также используется другая функция – **find_vma_prev()**,

```
/* mm/mmap.c */
struct vm_area_struct *
find_vma_prev(struct mm_struct * mm, unsigned long addr, struct vm_area_struct **pprev)
{
  if (mm) {
    /* Обход красно-черного дерева */
    struct vm_area_struct * vma;
    rb_node_t *rb_node, *rb_last_right, *rb_prev;

    rb_node = mm->mm_rb.rb_node;
    rb_last_right = rb_prev = NULL;
    vma = NULL;

    while (rb_node) {
      struct vm_area_struct * vma_tmp;

      vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);

      if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        rb_prev = rb_last_right;
        if (vma_tmp->vm_start <= addr)
          break;
        rb_node = rb_node->rb_left;
      } else {
        rb_last_right = rb_node;
        rb_node = rb_node->rb_right;
      }
    }
    if (vma) {
      if (vma->vm_rb.rb_left) {
        rb_prev = vma->vm_rb.rb_left;
        while (rb_prev->rb_right)
          rb_prev = rb_prev->rb_right;
      }
      *pprev = NULL;
      if (rb_prev)
        *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
      if ((rb_prev ? (*pprev)->vm_next : mm->mmap) != vma)
```

```
40          BUG();
41        return vma;
42      }
43    }
44    *pprev = NULL;
45    return NULL;
46  }
```

которая в общем то аналогична **find_vma()**, за исключением того, что она также возвращает указатель на VMA, предшествующий желаемому **VMA**. Это нужно сделать, потому что мы хотим провязывать **VMA** в односвязный список, чтобы при возможности, смёрджить несколько страниц в одну. Последней функцией для поиска **VMA** является **find_vma_intersection()**,

```
1  /* include/linux/mm.h */
2
3  static inline struct vm_area_struct *
4  find_vma_intersection(struct mm_struct * mm, unsigned long start_addr, unsigned long end_addr)
5  {
6    struct vm_area_struct * vma = find_vma(mm,start_addr);
7
8    if (vma && end_addr <= vma->vm_start)
9      vma = NULL;
10   return vma;
11 }
```

которая используется для поиска **VMA**, перекрывающей заданный диапазон адресов. Наиболее заметное использование этой функции - во время вызова функции **do_brk()**, когда регион увеличивается. Важно убедиться, что растущий регион не будет перекрывать старый регион.

Второй – нахождение еще не выделенного региона памяти:

Когда требуется выделить новую область в памяти, необходимо найти свободный регион, который достаточно велик, чтобы вместить запрашиваемое количество страниц. Функция, ответственная за поиск такой свободной области, называется **get_unmapped_area()**.

```
1  /* mm/mmap.c */
2  unsigned long get_unmapped_area(struct file *file, unsigned long addr, unsigned long len,
3                                  unsigned long pgoff, unsigned long flags)
4  {
5    if (flags & MAP_FIXED) {
6      if (addr > TASK_SIZE - len)
7        return -ENOMEM;
8      if (addr & ~PAGE_MASK)
9        return -EINVAL;
10     return addr;
11   }
12
13   if (file && file->f_op && file->f_op->get_unmapped_area)
14     return file->f_op->get_unmapped_area(file, addr, len, pgoff, flags);
15   return arch_get_unmapped_area(file, , len, pgoff, flags);
16 }
```

Функции передается ряд параметров, такие как:
- структура с метаданными по файлу, которому мы сейчас выделяем страницы
- адрес, с которого выделять страницы и длина
- смещение внутри файла
- restriction-флаги области памяти

Если выделение страниц происходит для внешнего устройства, например видеокарты, используется соответствующий ей **f_op→get unmapped area()**. Это связано с тем, что внешние устройства могут иметь дополни-

тельные требования к выделению страниц, о которых общий код ядра не может знать, например, выравнивание адреса по определенному виртуальному адресу. Если особых требований нет, вызывается специфичная для архитектуры функция **arch_get_unmapped_area()**. Не все архитектуры предоставляют свои собственные функции. Для тех, кто так не делает, в `mm/mmap.c` предоставляется универсальная версия.

Третий – вставка региона:

Основной функцией для вставки новой области памяти является **insert_vm_struct()**. На самом деле, это очень простая функция. Если разбирать поэтапно, сначала она вызывает **find_vma_prepare()**, чтобы найти подходящие **VMA**, между которыми должна быть вставлена новая область. Помимо этого, мы также спускаемся по вышеупомянутому красно-черному дерево и находим узел для вставки. Затем вызывается **vma_link()**, которая просто провязывает нас в найденные точки linked_list'а и красно-черного дерева.

Многие функции ядра не используют **insert_vm_struct()**, предпочитая вместо этого самим вызывать **find_vma_prepare()**, за которой сразу вызывается **vma_link()**, чтобы избежать многократного обхода дерева.

Связывание в **vma_link()** состоит из двух основных этапов, которые содержатся в трех отдельных функциях. Сначала **__vma_link_list()** вставляет **VMA** в вышеупомянутый односвязный список. Если это первое наименьший адрес в адресном пространстве (то есть мы произвели вставки в голову связного списка), оно помимо всего прочего станет корневым узлом красно-черного дерева. Второй этап - привязка узла к красно-черному дереву с помощью **__vma_link_rb()**.

Четвертый – слияние участков памяти:

В Linux раньше была функция под названием **merge_segments()**, которая отвечала за объединение соседних областей памяти вместе, если файл и разрешения совпадали. Цель состояла в том, чтобы уменьшить требуемое количество **VMA**, особенно потому, что многие операции приводили к выделению непомерно большого количества страниц. Это была очень дорогостоящая операция, поскольку она приводила к тому, мы много раз проходили почти по всем **VMA**. Именно поэтому, позже она была удалена, так как программы, которым результат её работы был критично важен, часто пытались её вызывать и проводили ну уж очень много времени в поисках страниц, которые можно смерджить. Взамен ей, сейчас мы имеет эквивалентную функцию **vma_merge()**, но она используется только в двух местах:

1. В **sys_mmap()**, если сопоставляется анонимный участок памяти (потому что анонимные регионы часто поддаются объединению).
2. Во время выполнения функции **brk()**, которая расширяет один участок памяти во вновь выделенный, где два участка должны быть объединены.

Вместо линейного прохода по всем регионам и поиска тем, что могут быть объедеенены, **vma_merge()** проверяет, может ли существующий регион быть расширен в соответствии с новым распределением, что сводит на нет необходимость создания нового региона. Следует также отметить, что расширение участка памяти может произойти, только если если разрешения для двух областей совпадают.

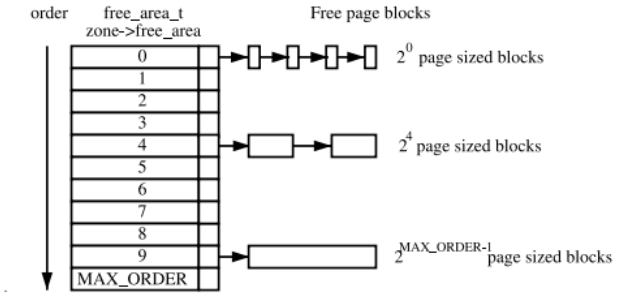
Глава 4. Аллокатор физических страниц

В этой глвае мы рассмотрим как работает аллокация физических страниц памяти. В linux за основу взят Binary Buddy Allocator, который выделяет 2^n физических страниц памяти. В нем довольно большая внутренняя фрагментация, однако он идеально подходит под эту задачу, посколько запрашиваются в основном большие блоки памяти, которые редко освобождаются.

Рассмотрим реализацию buddy аллокатора. Все выделяемые свободные блоки имеют размер в степень двойки: 4, 8, 16, 32, ... страниц физической памяти. Для каждой степени двойки есть специальная структура, хранящая его - **free_area_t**.

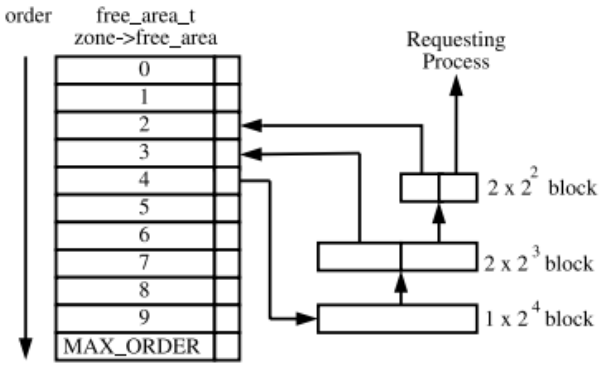
```
typedef struct free_area_struct {
struct list_head free_list;
unsigned long *map;
} free_area_t;
```

Хранится статический массив **free_area_t**, где i-ый элемент отвечает за свободные блоки размера 2^i. **free_list** - двусвязный список свободных блоков одного размера. Выглядит это так:



4.1. Аллокация

При запросе на аллокацию N страниц физической памяти мы находим минимальную степень 2 больше N. Далее проверяем наличие свободных страниц такого размера и отдаём если есть. В случае, если двусвязный список оказался пустым, то находим минимальный блок большего размера и рекурсивно разбиваем его пополам до нужного размера. Получившиеся половины пополняют списки свободных блоков нужного размера. Ниже приведен пример разбиений блока при запросе на аллокацию двух физических страниц:



Далее полученный физический адрес транслируется в виртуальный и возвращается пользователю.

4.2. Освобождение

При освобождении блока мы добавляем его в список свободных, а далее проверяем: если соседний блок так же свободен, то мы склеиваем эти два блока и добавляем в список свободных блоков следующего уровня. Эту процедуру мы продолжаем до тех пор, пока соседний блок пуст.

Глава 5. Итоги

Исходя из проведённого исследования, можно сказать, что, вопреки всеобщему мнению, прерывания при вызове **malloc()** вызываются не всегда, а довольно редко и в основном в случаях аллокаций довольно больших кусков памяти. В принципе этапы аллокации на других операционных системах схожи с тем, что описали мы. Отличаются отдельные детали реализации, но идеи и выбранные структуры данных идентичны.

В ходе исследования было также выяснено, что выгоднее не чередовать выделение маленьких кусков памяти с большими, поскольку большие аллокации затирают кэши.

Авторы считают, что данная работа имеет большую ценность и высокую образовательную составляющую. Она может помочь углубиться в понимание того, как работают аллокаторы в целом. Было разобрано более 4000 строк исходного кода и выделены ключевые моменты, позволяющие расширить понимание линуксовых аллокаций.

Список источников

1. Исходный код glibc [Электронный ресурс] : репозиторий. URL : https://sourceware.org/git/?p=glibc.git;a=tree;h=refs/heads/master;hb=refs/heads/master. (дата обращения: 20.12.23)
2. Исходный код malloc.c [Электронный ресурс] : отдельный файл репозитория в частности. URL : https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=78a531bc7a04bde7032886d08321d913a5541eb3;hb=refs/heads/master. (дата обращения: 20.12.23)
3. Исходный код linux [Электронный ресурс] : персональный репозиторий Линуса Торвальдса. URL : https://github.com/torvalds/linux. (дата обращения: 24.12.23)
4. «Understanding the Linux® Virtual Memory Manager» by Mel Gorman [Электронный ресурс] : техническая документация. URL : https://pdos.csail.mit.edu/~sbw/links/gorman_book.pdf. (дата обращения: 25.12.23)