avito.tech

C++ Russia Piter 2020

v.1.0

ПРО ПАМЯТЬ

Андрей Аксёнов

LE PLAN PIÈGE

01.

про
CPU

02.

про
OS

03.

про
***alloc**

04.

про
софт

[disclaimer] PhotoShop ;(

а вот, про

CPU

сто ~~оттенков серого~~ слоёв абстракции
от DIMM’а до регистра

ХАХА, СРАЗУ ОБМАН!

s/CPU/DIMM

- **Что мы* хотим знать про DIMM?**
  - Чем меньше, тем лучше, black box FTW!

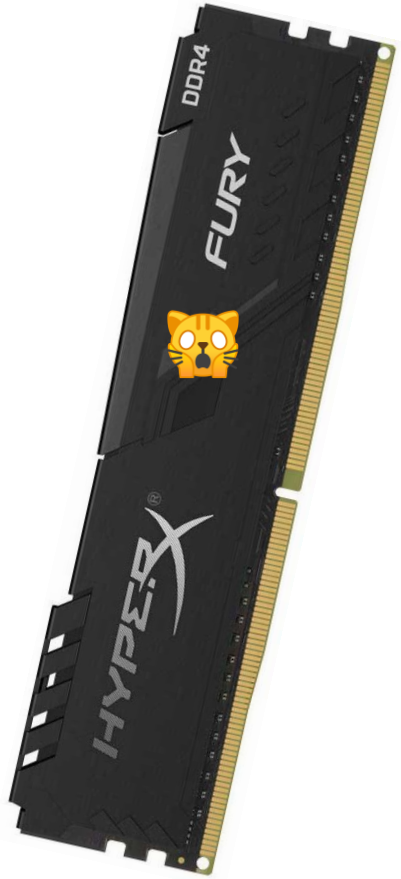
- Что для нас* линейная адресация, хотя нет
- Что **DDR4-2400** это **2400 Mtps** (T for Xfer)
- Что 2400/2 = **1200 Mcps** (C for Command, DDR)
- Что **CL17** = 17/1.200 = **14.166 ns**
  - И это 1st word latency, а не 8th word, ну лан

ПАМЯТЬ НЕ МГНОВЕННА

s/CPU/DIMM

- Так вот, CL17 = 17/1200M = 14.166 ns
- 1200M/17 ~= **71 Mrps**... а обещали **2400 Mtps**!?
- **Transfer != Request**

- 2400 Mtps * 8 bytes/t = **19200 MB/sec bandwidth**
- 1200 Mcps / 17 c/req ~= **71 Mreq/sec latency**
- 1 req = 4 bytes = **284 MB/sec**, например
- 1 req = 64 bytes = **4544 MB/sec**, например



ВНЕЗАПНО, ДИСК

Память, 2400CL17, per-channel (!)

- **19200 MB/s** по теории “линейного” чтения, seq
- **70.6M iops** по теории “случайного” чтения, rand
  - 1 io = **4...64** bytes

SSD диск, Samsung 970 Pro *

- **3500 MB/s** спека, **~3013 MB/s** бенчи
- **0.50M iops** спека, **~0.015M qd1**, **~0.418M qd64** бенчи
  - 1 io = **4096** bytes

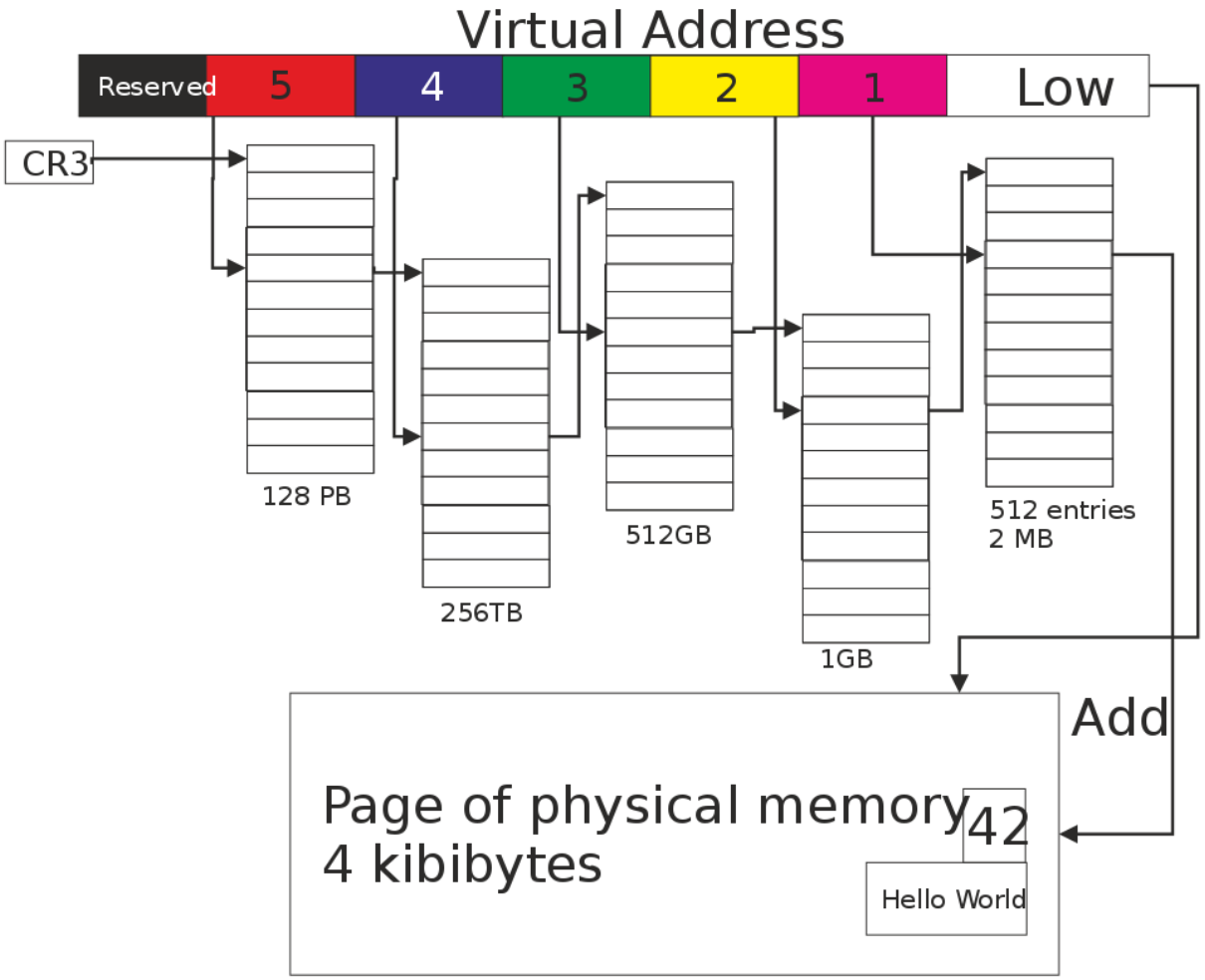
НАКОНЕЦ, CPU (НУ, MMU)

VM, MMU, L[123] cache, и все-все-все

- **Что мы* хотим знать про x64 MMU?** (Чем меньше...)

- А вот “4K странички”, “page table”, и чо?
- 2^64 / 4096 = $очень$много, как быть?
- Хоп, 4-слойное **дерево в указателе**, 16:9:9:9:9:12

63...48	47...39	38...29	29...21	20...12	11...0
Unused	**L4_idx**	**L3_idx**	**L2_idx**	**L1_idx**	**page_offset**


Virtual Address
Reserved
5
4
3
2
1
Low
CR3
128 PB
256TB
512GB
1GB
512 entries
2 MB
Add
Page of physical memory
4 kibibytes
42
Hello World

ВНЕЗАПНО, TLB

Translation **L**ookaside **B**uffer

- **4 (!)** проезда по дереву всегда – больно жирно
- Поэтому – кеширование, т.н. **TLB**, откуда **TLB misses**

- В порядке fyi...
- А там свои несколько уровней, L1 TLB != L1 cache
- А там бывают отдельные ITLB и DTLB
- А там еще superpages, hugepages, OS агрегация...
  - ...отдельный мир, нам туда (сегодня) не надо ;))

...НАМ НАДО В КЕШ

- **L1** / **L2** / **L3** cache
- Наконец-то “понятные” слова в части CPU!

- “Все знают”, что это “дешево”, например **@ 3.0 Ghz**
- full miss = 17 / 1.200 ns * 3.0 = **42 cycles @ DIMM... + X!**
- **L1** latency = **4-5** cycles, например (Intel 9th gen)
  - **mov eax, [eax]** ; load-to-use loop
- **L2** latency = **12** cycles
- **L3** latency = **42** cycles (совпадение) – это наш X

УПРАЖНЕНИЕ: ПОМЕРИТЬ САМИМ!

- Бенчмарки/спеки в инете ***еще (чуть) страшнее***
- kw = { “anandtech”, “LMBench”, “TinyMemBench” }

- L2 = **12...22** такта (от архитектуры), ок
- L3 = **38...56** тактов, ок
- 32 mb set = ~25-40 ns, **~120** тактов
- 512 mb set = ~90-100 ns, **~300** тактов 😿

- + пишут про 42 такта + **~50 ns** ram latency...

...И ОБРАТНО СТРАНИЧКИ

- 16 : 9 : 9 : 9 : 9 : 12

63...48	47...39	38...29	29...21	20...12	11...0
Unused	**L4_idx**	**L3_idx**	**L2_idx**	**L1_idx**	**page_offset**

- 2^12 = **4 kb** = **4096**
- 2^21 = **2 mb** = **min HugePageSize**
- 2^30 = **1 gb** = **max HugePageSize**

- Упражнение, **linear virtual != linear physical**
- “~~Папа~~ OS может”, правда?

Память – сложная машина.
На которой едет байт.

Он доедет даже с диска.
Только вот в каком году.

а вот, про

OS

самая короткая полусекция

ВНЕЗАПНО...

- **Что мы* хотим знать про OS?**

- Кроме того, что это Linux, “как у всех”?

OS vs CPU

- OS “прячет” нас от ужасов MMU и делает скучное
  - “Для CPU” нужно поддерживать page table
  - “Для программы” (linear) virtual address space
  - Page cache и прочий swap file опять же
- API “про память” считай ***один***
  - Актуален только **mmap()**
  - Неспехом вымирает sbrk()
- Но! Вызов OS *всегда* небыстрый, syscall

OS vs LIBC

- Поэтому **malloc()** это никогда не syscall
- Всегда нужен userland менеджер

- Тупо **libc**
- Или **jemalloc**
- Или **tcmalloc**
- Или ~~**ptmalloc2**~~ **glibc**
- Или...
- ...Или руками?!

а вот, про

ALLOC()

как же всё-таки устроен “аллокатор”?

АЛЛОКАТОР ЭТО ПРОСТО

- OS => libc это ж 1 функция
  - **mmap()**

- libc => программа ж это 2 функции
  - **malloc()**
  - **free()**

V1. ЧО ДУМАТЬ, НАДО КОДИТЬ

```
void M_init() {
    g_arena = mmap(... LOTS_OF_RAM...);
    g_arena_end = arena + LOTS_OF_RAM;
}
void * M_alloc(size_t size) {
    if (g_arena + size > g_arena_end)
        return NULL;
    g_arena += size;
    return g_arena - size;
}
```

МИНА ВЕЗДЕ!

- Память в систему – не отдаем
- Память сами себе – не отдаем
- Но и... неважно пока

- Потоков всего много
- **Data races** в обоих строчках
- Легко выправить atomic-ом =>
- **Работающий (!!!)** arena allocator, MT даже!

ХОП, НЕ *СОВСЕМ* ИГРУШКА

Что хорошо

- *зверски* быстр; инлайнится; оверхеды CPU нулевые
- *предельно* эффективный; оверхеды памяти нулевые
- можно* умудриться применить! 

Что плохо

- free() не работает, “instant leak” (не совсем)
- malloc() быстро сломается => нужно ещё арен!
- выравнивание* пока так себе (важно ли?)
- ооочень неудобно думать про “общий” lifetime

НАДО ОТДАВАТЬ!

- Чуть обдумаем сложности

- Размеры в malloc() любые
- Размеры в free() любые
- **Спецпамяти под internals *нету***
- **Аллокатора ещё нету, мы – он!**

- Всё “легко” решаемо

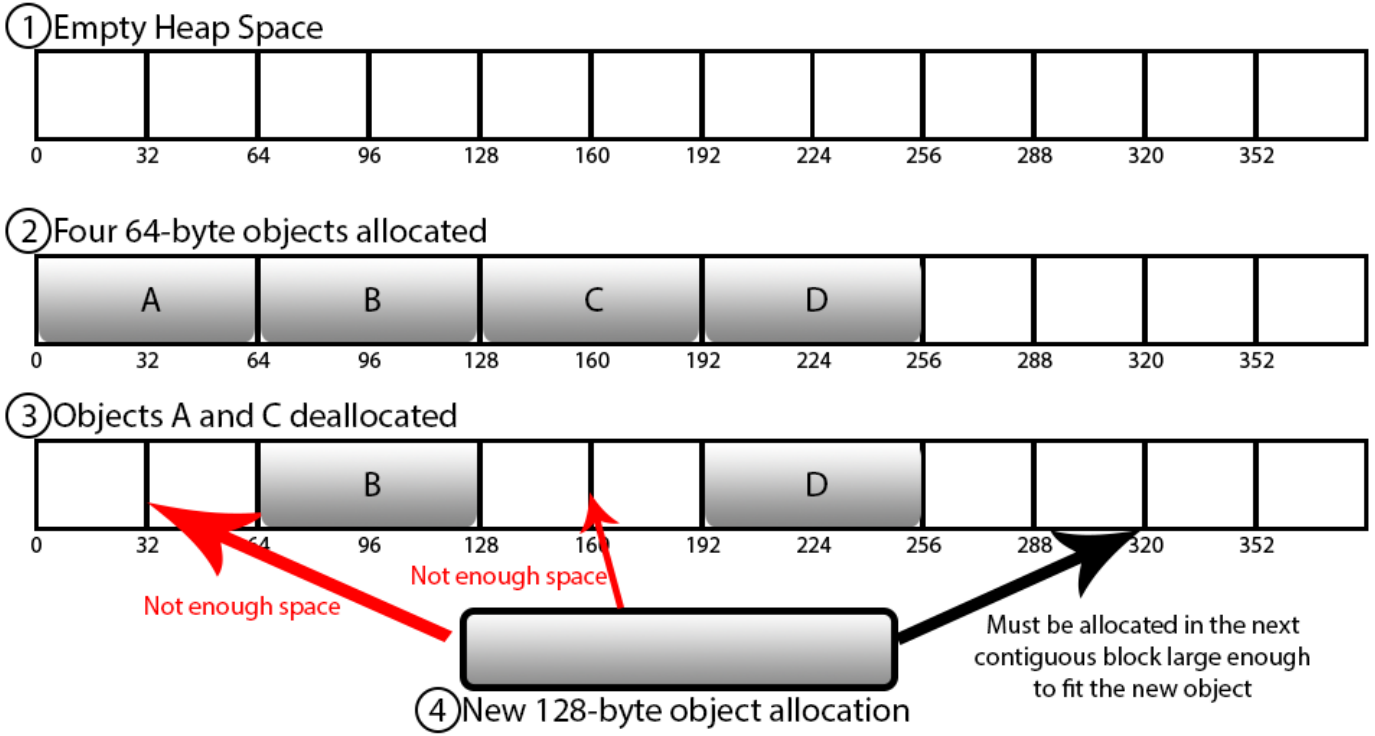
V2. СДЕЛАЕМ... КАК-ТО

```
struct Header {
    size_t size;
    Header *prev, *next;
    char data[8];
};
struct Arena {
    char *begin, *end, *cur;
    Header *used_head, *free_head;
    Arena *next;
} *g_root_arena;
```

***Хорошо* не стало**

- NB: мету кладем в саму арену!
- lifetime стал прозрачен
- free() работает и быстро, O(1)
- malloc() поначалу ок, но...
- malloc() vs free_root, O(n)
- +24 байта на *каждый* аллок
- кстати, где mutex?

“ВНУТРЕННЯЯ” ФРАГМЕНТАЦИЯ



ПОБОРЕМСЯ ЕЩЁ

- Как избежать +20 байт на *мелкие* аллоки?
- И фрагментации заодно тоже?!
- Табличка, aka **pool**

- Как избежать обхода free-списка?
- Не обходить (ненужный) список!

- Как избежать mutex-ов?
- Никак (но можно редуцировать)

V3. ПУЛЫ НАШЕ ВСЁ

```
template <int N>
struct Pool {
    bool is_full; // non-full first
    void *next;
    bool is_used[K]; // or bitmap
    char data[K][N];
};
Pool<8> * g_root8;
Pool<13> * g_root13; // yes, 13
Pool<32> * g_root32;
```

Становится интереснее!

- K подгоняем “под 4096* байт”
- Pool::alloc() довольно быстр
- Pool::free() очень быстр, O(1)
- malloc() поначалу ок, но...
- free() стоит O(**sum**(num_pools))
- т.к. *теперь* надо addr => pool
- кстати, все ещё mutex!

ДУМАЕМ ВСЛУХ

- **Большое?**
- Сразу в mmap()! glibc от 128 kb, btw

- **Среднее?**
- Списки? Пулы? mmap()? Неясно...

- **Мелкое?**
- Сразу в пулы! Но есть 100M free() problem

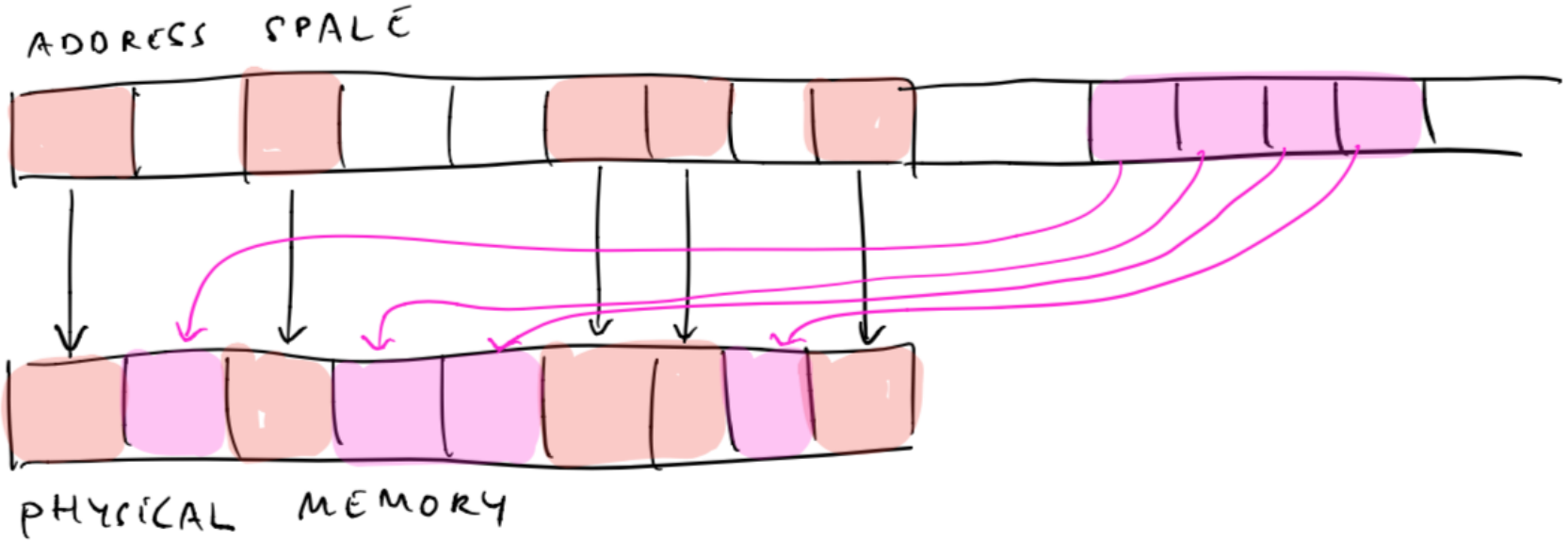
ДУМАЕМ ЕЩЁ ВСЛУХ

- **Обсудим мелкое**

- Пулы побольше, eg. 64 kb min
- Пулы подращивать, eg. 64...2048 kb
- И/или адреса хэшировать

- 1x 2048 kb pool ~= 128k obj * 16b
- 100M objs ~= 763 pools => влезет в небольшой хэш

“ВНЕШНЯЯ”... ДЕФРАГМЕНТАЦИЯ



- проблемы древнего 32 битного мира, в общем

ПОРА!

- **Пора сделать заход на V4!**

а вот, про

JEMALLOC

как устроено у взрослых?!

15 ЛЕТ ПРОГРЕССА

Было в 2005 и v.4

- **Region**, собственно user data
- **Small** class, 8 b … 14 kb
- **Large** class, 16 kb … 1792 kb
- **Huge** class, 2 mb … inf

Стало в 2017 и v.5

- **Region**, собственно user data
- **Small** class, 8 b … 14 kb
- **Large** class, 16 kb …

… inf

15 ЛЕТ ПРОГРЕССА

Было в 2005 и v.4

- **Chunk**, логический, 1-2 mb
- **Arena**, список chunks/bins
- **Run**, “странички подряд”

- **Thread cache***

Стало в 2017 и v.5

- **Extent**, page aligned... и всё
- **Arena**, список extents/bins
- **Slab**, “small regions подряд”

- **Thread cache**

SMALL & LARGE

Spacing	Small sizes
-	8
16	**[16, 32, 48, 64, 80, 96, 112, 128]**
32	**[160, 192, 224, 256]**
64	[320, 384, 448, 512]
...	...
1 kb	[5 kb, 6 kb, 7 kb, 8 kb]
2 kb	[10 kb, 12 kb, 14 kb]

Spacing	Large sizes
2 kb	**[16 kb]**
4 kb	[20 kb, 24 kb, 28 kb, 32 kb]
8 kb	[40 kb, 48 kb, 56 kb, 64 kb]
16 kb	[160 kb, 192 kb, 224 kb, 256 kb]
...	...
1 mb	[5 mb, 6 mb, 7 mb, 8 mb]
...	...

ТЕРМИНОЛОГИЯ СВОЯ

- **Regions** = и есть аллокации
- **Slabs** = и есть пулы для мелочи
- **Extents** = ~2+ mb (?) “куски адресов”

- Оверхеды на **17 байт** классически **94%**
- Оверхеды на **33 байта** классически **46%**
- Оверхеды на **161+ байт** не более **20%**
- Начиная с **4 страничек** “большое” и там тоже **20%**

НО ВЫГЛЯДИТ УЖЕ ЗНАКОМО

- **Small** objects => slabs => несколько на extent
- **Large** objects “each have their own extents backing them”
- Large это лишь 16 kb

- На каждый *размер* помельче свой extent?
- Или для 16 kb свой extent, для 32 kb свой?
- Todo: проверить, я небось не успею!
- Rant: а все потому, что доки не пишут!

ЕЩЁ ИНТЕРЕСНОЕ

- А вот обратно ptr2meta для free(); какая СД?
- NB: надо всегда; несмотря на pools/slabs

- **Freelist?** Тупой нет; но можно “по размерам”; но можно pow2; но можно inplace; etc
- **Hash?** Ну такое, гранулярность

- В целом… “аллоки” собираются ж в **дерево**
- jemalloc = **красно-черные деревья** :)

...КТО ТУТ КНОПКА?

а вот, про

SMP

lockfree это важно

ЯДЕР НЫНЧЕ МНОГО

- **И они друг другу мешают! 15 сек об этом**

- *Основная* ачивка je/tc/...? Вся эта сложность?
- Раз, **multiple arenas** – локи есть, contention меньше
- Два, **thread cache** – локов, иногда, вообще* нет

- Есть и *непобедимые* моменты
- Cache line contention на запись, vs multi-arena...

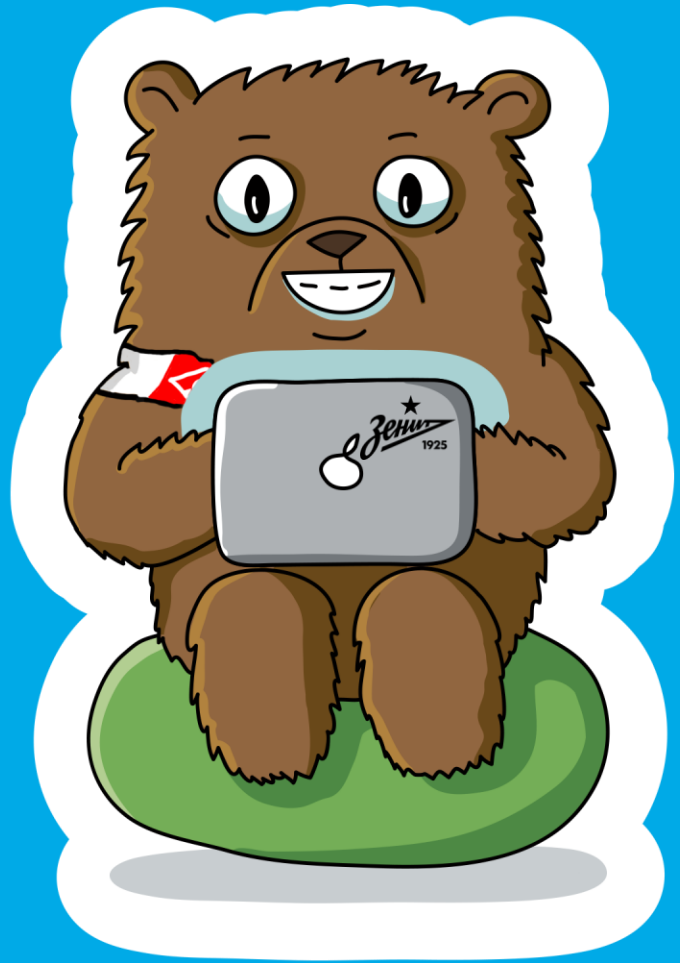
KNOW YOUR (SMP) TOOLS

- Вывод, **1x allocer Nx writers** = не надо так
- Вывод, **ephemeral + longterm** = не ~~надо~~ круто так
  - Внезапно, Aerospike
- Вывод, **даёшь jemalloc*** (плюс хинты бы ещё)
- Вывод, руками не надо?!

а вот, про

ПРОГРАММЫ

самый сильный аллокатор –
красно-синий* программист



ЧТО МЫ УЗНАЛИ СЕГОДНЯ?

- Много бесполезных фактов

ЧТО МЫ УЗНАЛИ СЕГОДНЯ?

- Что аллокаторы* так-то неглупы
- Что многопоточка сложно… и медленно
- **Что локальную мини-арену… не обогнать!**
- Что оверхеды “на мелочи” ажно до 50%
- Что в CPU/DRAM прячется Сатана

- …

КАК НАМ ИХ ПРИМЕНИТЬ?

- **Оптимально забыть навсегда**

Э! ОБМАН! FRAG! VSZ! RSS! OOM!!1

- Кстати, стало же понятнее?!
- Почему hugepage это минима 2 mb? ;)

- **VSZ** = “захваченная”; p = malloc(1*TB);
- **RSZ** = “реальная”; for (...) *p++ = 1;
- **SHR** = “общая”; >1 процесса

- А вот и **OOM**, посчитал бяку и...

#ЧОКАСАЕМО ФРАГМЕНТАЦИИ

- Аллокаторы умные, но мы вредные

- 1 аллок внутри *каждого* пула бдыщь
- Куча пулов бдыщь
- Фрагментация щька
- **OOM killer бжю! Пущь пущь пущь!!**

- iirc je умеет отчёты, but I’m already in my pajamas

КАК *ЕЩЁ* НАМ ИХ ПРИМЕНИТЬ?

- А вдруг...
- А вдруг таки 100M объектов по 17 байт?!
- А вдруг вместо vector<string> таки vector<char>?
- А вдруг “аллокации”, но в файле данных (хихи)?
- А вдруг лучше* поймем, что/как крутить в jemalloc?
- А вдруг “просто” ловчее аллоки? Или миниарены? Или...

- А вдруг хоть **перестанем бояться**? :)))
- Списки, пулы, хэши, деревья... пф-пф-пф

avito.tech

C++ Russia Piter 2020

Андрей Аксёнов

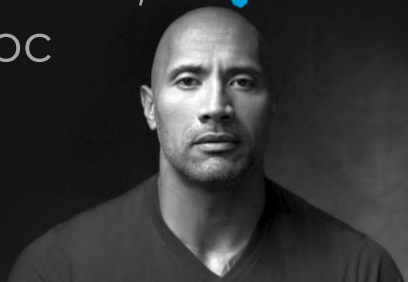
Search Infra Lead

shodan@shodan.ru

shodanium

«Товарищи призывники. Надо понимать всю глубину наших глубин.

– Джейсон Стэтхем, автор JSMalloc



BACKLOG

- Фрагментация int, ext
- OOM, где его открутить
- VIRT, RSS, SHR, чему верить
- Убертруд Дреппера
- Философия native vs managed
- Утечки, проезды, борьба (nonrel?)
- AVX trick, и кеши/префетчи
- VM trick, unused ptr as guid
- VM trick, endless “ring” buffer