Five100Miles

，从不 ~

博 % - 56 - 0 - 8 - 69316

: Five100Miles
: 3 11个%
丝: 32
关 : 0
+加关

mimalloc源码笔记

A

mimalloc 去 6 (了半 写) 内 分 , 其 Daan Leijen 发, 为Koka与Lean runtime system 供内 . , mimalloc 各 benchmark上 优于其 主 allocator(分别 tcmalloc 与jemalloc 7%与14%)且 内 .
mimalloc 前 为1.2.2, 可以 取 与 .

1. 编译与使用

mimalloc使 cmake作为 具, 以下 令即可 .
功 下会 (libmimalloc) 动 / 以及 .. 具.

```
mkdir -p [dir to build]
cd [dir to build]
cmake [source path]
make
```

可以 件中包 mimalloc.h 加-lmimalloc 使 mimalloc.
mimalloc可以与其 allocator共 , 于cmake 可以使 以下 令 :
target_link_libraries([program] PUBLIC mimalloc)
不 也可以 变 :
env LD_PRELOAD=[path/to/libmimalloc.so] [program]

T 印libmalloc .. 信 , 可以 变 (T debug):
env MIMALLOC_VERBOSE=1 [program]
env MIMALLOC_SHOW_STATS=1 [program]

其 :
印 / 信 : MIMALLOC_SHOW_ERRORS=1
使 huge page : MIMALLOC_LARGE_OS_PAGES=1

.. 个 .. 例:

```
[23:18:20] hansy@hansy:~$ cat 1.c
#include <sys/time.h>
int main() {
  struct timeval last;
  struct timeval next;
  gettimeofday(&last, 0);
  for (int i = 0; i < 10000000; ++i) {
    int *p = malloc(i * sizeof(int));
    free(p);
  }
  gettimeofday(&next, 0);
  printf("%llu.%06llu\n",
    (next.tv_usec > last.tv_usec ? next.tv_sec - last.tv_sec : next.tv_sec - 1
    (next.tv_usec > last.tv_usec ? next.tv_usec - last.tv_usec : 1000000 + next
  return 0;
}
[23:18:25] hansy@hansy:~$ gcc 1.c -w && ./a.out
8.606776
[23:18:42] hansy@hansy:~$ env LD_PRELOAD=./mimalloc/build_release/libmimalloc.s
1.314598
[23:18:57] hansy@hansy:~$ env MIMALLOC_VERBOSE=1 LD_PRELOAD=./mimalloc/build_re
mimalloc: process init: 0x7f4b0d558740
mimalloc: option 'large_os_pages': 0
mimalloc: option 'secure': 0
mimalloc: option 'page_reset': 0
mimalloc: option 'cache_reset': 0
```

2020 5 (6)
2020 4 (1)
2020 3 (1)
2020 1 (1)
2019 12 (2)
2019 7 (1)
2019 6 (2)
2019 4 (2)
2018 10 (2)
2018 6 (2)
2018 5 (2)
2018 4 (4)
2018 3 (3)
2018 2 (14)

1. LLVM (3) - PASS(4)
2. LLVM (10) - 令 (二) lowering (1)
3. LLVM (9) - 令 ( ) (1)
4. LLVM (6) - CompilerRT之safestack(1)
5. linux-glibc内 1(ptmalloc 分)(1)

1. LLVM (13) - 令 (五) select(3)
2. LLVM (11) - 令 (三) combine (2)
3. LLVM (9) - 令 ( ) (2)
4. mimalloc (2)
5. LLVM (5) - SMS(2)

1. Re:LLVM (10) - 令 (二) lowering

```
1.323777
mimalloc: option 'show_stats': 0
heap stats:     peak      total      freed       unit      count
   elapsed:     1.324 s
   process: user: 1.299 s, system: 0.012 s, faults: 0, reclaims: 479, rss: 2.6
mimalloc: process done: 0x7f4b0d558740
```

2. 设计思想

先 technical report, 代 可以 . 原 .
以下 T :
mimalloc
代allocator T 各 , 包 , 全, 化以及其 . 发 个
于Koka与Lean runtime system 到两 况:
short lived 分 , 个 制 allocator 优于jemalloc 主
allocator.
二 些runtime system 使 RC 制 内 . 为减 pause T
减 . 为 取 佳 们 T allocator 协助: 临内 压力 之前
decrement.

为 决以上 , 出了free list sharding(分) .
传 allocator 内 分 (size-class) free list, 个 . 其 分
内 可以 到O(1), , 内 个 上.
为 mimalloc修 了 , 先 分为 列(于分 不 内)
(mimalloc page, 前 64K), 个free list . 么 内 分 口可以
下:

```
struct block_t { struct block_t *next; }
void *malloc_by_size(page_t *page, int size) {
  if (block_t *block = page->free) {
    page->free = block->next;
    page->used++;
    return block;
  }
  return malloc_generic(size); // slow path
}
```

于swift与python 使 RC 制 会 i free, .
可以 制free 剩余 加入deferred decrement list. 于何
? 像kernel , 只 临内 压力 再 优 , T allocator .
malloc_generic(slow path)中mimalloc会 义 deferred_free .
假 内 分 内 i slow path 不 么办? mimalloc再 为
free list分 : 加 个local free list. 其 入local free list, 保 free list 会
变 . 分 口中再 local free list 值 free list 使 .
mimalloc中page 于 , 仅从 中分 内 , 但其 可以 .
为 免 入 , mimalloc再为 加 个thread free list 于 其 ( )
内 , 发 使 atomic_push(&page->thread_free, p) 其 入thread free list中.

```
void atomic_push(block_t **list, block_t *block) {
  do {
    block->next = *list;
  } while (!atomic_compare_and_swap(list, block, block->next));
}
```

thread free list 不光减 了 争, 减 了 不 争. 似于local
free list, thread free list 也 分 口中.
优 :
分 , 减 停
, 保 RC
升thread free list i
lock free
传 allocator 于减 内 使 , 加 分 / , mimalloc 了 升 内
可以 升allocator .

3. 源码分析

分 入, 博 , 也 了

--jahentao

2. Re:LLVM (9) - 令 ( )

写 , 为了 你 册 号

--Sm0ckingBird

3. Re:LLVM (3) - PASS

@yy172 你 了, 不 么 . 上ISelLowering 令 代 , 只T 你使 O2 优化 会 . 了 RISC V lowering ...

--Five100Miles

4. Re:LLVM (3) - PASS

@Five100Miles , 几个 下, llvm/lib/target/riscv下 RISCVISelLowering.cpp下发 了 全 变 优化, llc ...

--yy172

5. Re:LLVM (3) - PASS

@yy172 中 优化只会做 关 优化, 关 优化 T , 做 .具体 , 个 (lib/Target/[Arch]/) 下 个 [arch]Ta...

--Five100Miles

mimalloc (仅3.5k loc), .
先 下内 , 似于ptmalloc中malloc_chunk -> bin -> malloc_state, mimalloc也 三 mi_block_t -> mi_page_t -> mi_heap_s.

```
typedef uintptr_t mi_encoded_t;
typedef struct mi_block_s {
  mi_encoded_t next;
} mi_block_t;
```

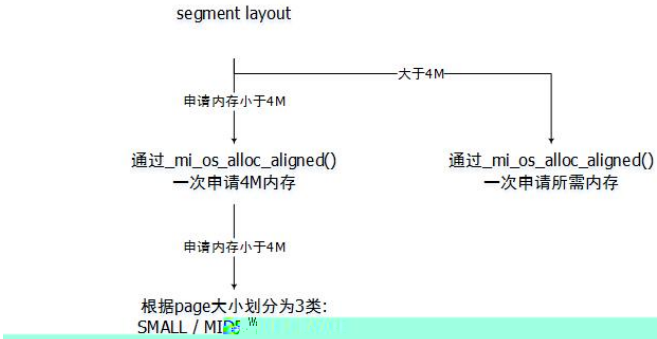
mimalloc中内 单位 mi_block_t, 区别于ptmalloc中malloc_chunk , mi_block_t只 个 ( )下 内 .
为 mimalloc中 内 size classed page中分 , 不 T 内 做migrate, 不 保 , ( ) 及 信 .

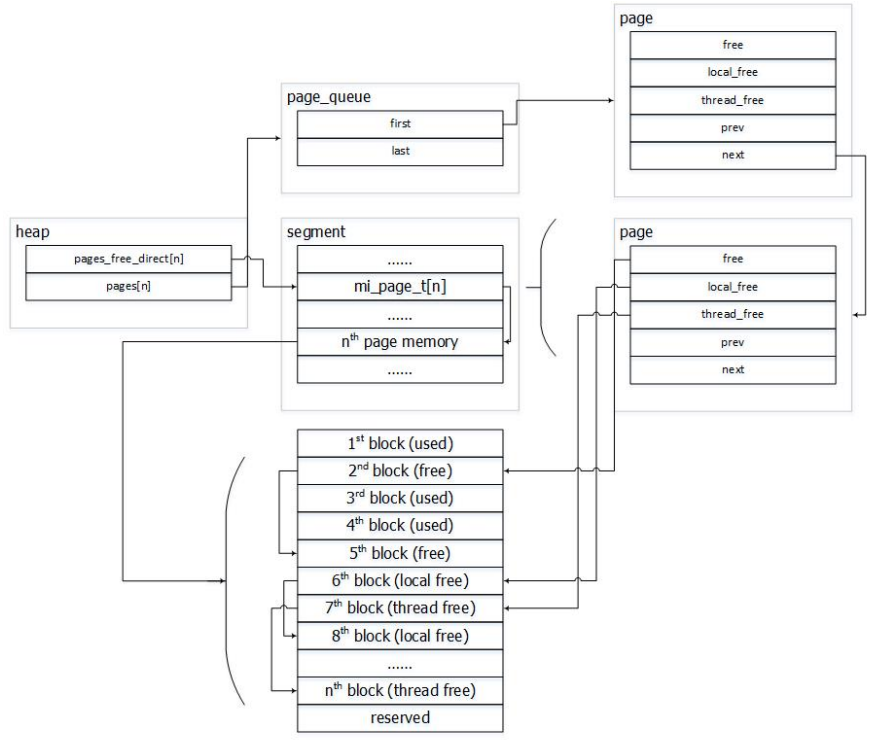
```
typedef union mi_page_flags_u {
  uint16_t value;
  struct {
    bool has_aligned;
    bool in_full;
  };
} mi_page_flags_t;
typedef struct mi_page_s {
  //        (segment)中      , page = &segment->pages[page->segment_idx]
  //     分   初  化(mi_segment_alloc),  于          内      (segment + idx
  uint8_t segment_idx;
  //            使 ( 分 内 )
  bool segment_in_use:1;
  //  位   (什么 候 位?)
  bool is_reset:1;

  //
  mi_page_flags_t flags;
  //      前内
  //      几个   :
  //         -          (SMALL / MEDIUM / LARGE)决
  //       分 内       - 即reserved,    (      /          )决
  //      前   内       - 即capacity,  于        T  值mi_block_t  i RSS   ,
  //      前分 内       - 即used
  //         动内       - used - thread_free - local_free
  uint16_t capacity;
  // 保        内    分   内    个 ,  mi_page_init中初 化, 为page_size / mi_p
  uint16_t reserved;

  // free   ,     前   内   , malloc   从      取   内
  mi_block_t *free;
  //    cookie,  于 全
  uintptr_t cookie;
  //       ,            locl_free与thread_free包   内
  size_t used;
  //    内         内     入     , 只   free       其 值 free
  mi_block_t *local_free;
  volatile uintptr_t thread_freed;
  //  似于local_free, 但 只 其       内     入     , 使 cas 决 争
  volatile mi_thread_free_t thread_free;

  //         内        ,  个 只          内
  size_t block_size;
  //        个
  mi_heap_t *heap;
  // 双        (   于    )          内
  struct mi_page_s *next;
  struct mi_page_s *prev;
} mi_page_t;
```

mimalloc 二 mi_page_t前 了, 其 .

```
typedef struct mi_tld_s mi_tld_t;
typedef struct mi_page_queue_s {
  mi_page_t *first;
  mi_page_t *last;
  size_t block_size;
```

20.01.18 两





1. mi_heap_t中保存两个关于页信息的成员, pages_free_direct[]指向包含空闲小内存的各类页, pages[]指向所有类型内存的页的队列
2. page中包含三个链表, free, local_free, thread_free, 分别指向本页内的空闲内存, 本线程释放的内存, 其它线程释放的内存
3. 页内的内存块每次申请时按系统页(4K)大小初始化为空闲内存块, 其余内存为reserved状态, 原因是减少RSS大小

4. 接口实现

alloc-override*.c中 义 何 准malloc/free及c++ new/delete 作 alias为mi_* 口, malloc alias为mi_malloc:

```
#define MI_FORWARD(fun) __attribute__((alias(#fun), used, visibility("default")
void *malloc(size_t size) mi_attr_noexcept MI_FORWARD1(mi_malloc, size);
```

先 下mi_malloc , 内 fast path slow path.

```
void *mi_heap_malloc(mi_heap_t *heap, size_t size) {
  // MI_SMALL_SIZE_MAX = 128 * sizeof(void *)
  if (size <= MI_SMALL_SIZE_MAX) {
    return mi_heap_malloc_small(heap, size);
```

```
  }
  return _mi_malloc_generic(heap, size);
}
void *mi_malloc(size_t size) {
  return mi_heap_malloc(mi_get_default_heap(), size);
}
```

于 于1M 内 分 会 fast path. mi_heap_malloc_small会 .. 从 前 free 中 取 , 不 则 slow path. _mi_wsize_from_size会 传入 到sizeof(void *) (machine word size) 取index, pages_free_direct 了包 page.

```
void *mi_heap_malloc_small(mi_heap_t *heap, size_t size) {
  mi_page_t *page = _mi_heap_get_free_small_page(heap, size);
  return _mi_page_malloc(heap, page, size);
}
size_t _mi_wsize_from_size(size_t size) {
  return (size + sizeof(uintptr_t) - 1) / sizeof(uintptr_t);
}
mi_page_t *_mi_heap_get_free_small_page(mi_heap_t *heap, size_t size) {
  return heap->pages_free_direct[_mi_wsize_from_size(size)];
}
void *_mi_page_malloc(mi_heap_t *heap, mi_page_t *page, size_t size) {
  mi_block_t *block = page->free;
  if (block == NULL) {
    return _mi_malloc_generic(heap, size); // slow path
  }
  page->free = mi_block_next(page, block);
  page->used++;
  block->next = 0;
  return block;
}
```

内 于1M 不 则 _mi_malloc_generic. 前 , slow path 先会 .. 之前 内 .
可以 册deferred_free 于 之前 , 口会 _mi_deferred_free中 . 另 也会 其 内 (T MI_USE_DELAYED_FREE). 内 再 从候 列 分 内 , 到候 fast path分 (可 再 分 入slow path ?).

```
void *_mi_malloc_generic(mi_heap_t *heap, size_t size) {
  if (!mi_heap_is_initialized(heap)) {
    mi_thread_init();
    heap = mi_get_default_heap();
  }

  _mi_deferred_free(heap, false);

  _mi_heap_delayed_free(heap);

  mi_page_t *page;
  // MI_LARGE_SIZE_MAX = 512K, 32bit platform减半
  if (size > MI_LARGE_SIZE_MAX) {
    if (size >= (SIZE_MAX - MI_MAX_ALIGN_SIZE)) {
      page = NULL;
    }
    else {
      page = mi_huge_page_alloc(heap, size);
    }
  }
  else {
    page = mi_find_free_page(heap, size);
  }
  if (page == NULL) return NULL;

  return _mi_page_malloc(heap, page, size);
}
static mi_deferred_free_fun *deferred_free = NULL;
void _mi_deferred_free(mi_heap_t *heap, bool force) {
  heap->tld->heartbeat++;
  if (deferred_free != NULL) {
    deferred_free(force, heap->tld->heartbeat);
```

```
  }
}
void _mi_heap_delayed_free(mi_heap_t *heap) {
  mi_block_t *block;
  do {
    block = (mi_block_t*)heap->thread_delayed_free;
  } while (block != NULL && !mi_atomic_compare_exchange_ptr((volatile void**)&h

  while(block != NULL) {
    mi_block_t *next = mi_block_nextx(heap->cookie, block);
    if (!_mi_free_delayed_block(block)) {
      mi_block_t *dfree;
      do {
        dfree = (mi_block_t *)heap->thread_delayed_free;
        mi_block_set_nextx(heap->cookie, block, dfree);
      } while (!mi_atomic_compare_exchange_ptr((volatile void**)&heap->thread_d
    }
    block = next;
  }
}
```

下 何 个 内 . 于分 内 mi_heap_t->pages[] ,
index 取mi_page_queue_t, 其 分 (LRU) . 可
, 以先T _mi_page_free_collect 做free 作.
作 仍 LRU 不 , T
mi_page_queue_find_free_ex 历 内 . 口会 历 列中 个 内
, 三 况: A
个 则 . T 发 全 则会 个
(作retire), 个 又会保 到 retire 之 i 内 分 .
发 个 但 capacity 于reserved .
mi_page_extend_free , 口可以发 不 制 ,
不 个 (OS page), 分 extend 原 RSS 义 (初 化free
会写 i 内 分).
发 个 则 其 入full (mi_page_to_full), 历 低 . mi_heap_t->pages[] 个index (以 , 什么 候 其取 到 ?).
历 仍 取 么 mi_page_fresh 取 个全 .

```
mi_page_t *mi_find_free_page(mi_heap_t *heap, size_t size) {
  mi_page_queue_t *pq = mi_page_queue(heap, size);
  mi_page_t *page = pq->first;
  if (page != NULL) {
    if (mi_option_get(mi_option_secure) >= 3 && page->capacity < page->reserved
      mi_page_extend_free(heap, page, &heap->tld->stats);
    }
    else {
      _mi_page_free_collect(page);
    }
    if (mi_page_immediate_available(page)) {
      return page;
    }
  }
  return mi_page_queue_find_free_ex(heap, pq);
}
mi_page_queue_t *mi_page_queue(const mi_heap_t *heap, size_t size) {
  return &((mi_heap_t*)heap)->pages[_mi_bin(size)];
}
bool mi_page_immediate_available(const mi_page_t *page) {
  return (page->free != NULL);
}
void _mi_page_free_collect(mi_page_t *page) {
  if (page->local_free != NULL) {
    if (page->free == NULL) {
      page->free = page->local_free;
    }
    else {
      mi_block_t *tail = page->free;
      mi_block_t *next;
      while ((next = mi_block_next(page, tail)) != NULL) {
        tail = next;
```

```
      }
      mi_block_set_next(page, tail, page->local_free);
    }
    page->local_free = NULL;
  }
  if (mi_tf_block(page->thread_free) != NULL) {
    mi_page_thread_free_collect(page);
  }
}
mi_page_t *mi_page_queue_find_free_ex(mi_heap_t *heap, mi_page_queue_t *pq) {
  mi_page_t *rpage = NULL;
  size_t count = 0;
  size_t page_free_count = 0;
  mi_page_t *page = pq->first;
  while( page != NULL)
  {
    mi_page_t *next = page->next;
    count++;

    _mi_page_free_collect(page);

    if (mi_page_immediate_available(page)) {
      if (page_free_count < 8 && mi_page_all_free(page)) {
        page_free_count++;
        if (rpage != NULL) _mi_page_free(rpage, pq, false);
        rpage = page;
        page = next;
        continue;
      }
      break;
    }

    if (page->capacity < page->reserved) {
      mi_page_extend_free(heap, page, &heap->tld->stats);
      break;
    }

    mi_page_to_full(page, pq);
    page = next;
  }

  mi_stat_counter_increase(heap->tld->stats.searches, count);

  if (page == NULL) {
    page = rpage;
    rpage = NULL;
  }
  if (rpage != NULL) {
    _mi_page_free(rpage, pq, false);
  }

  if (page == NULL) {
    page = mi_page_fresh(heap, pq);
  }
  return page;
}
mi_page_extend_free(mi_heap_t *heap, mi_page_t *page, mi_stats_t *stats) {
  if (page->free != NULL) return;
  if (page->capacity >= page->reserved) return;

  size_t page_size;
  _mi_page_start(_mi_page_segment(page), page, &page_size);
  if (page->is_reset) {
    page->is_reset = false;
    mi_stat_decrease(stats->reset, page_size);
  }

  mi_stat_increase( stats->pages_extended, 1);

  size_t extend = page->reserved - page->capacity;
  size_t max_extend = MI_MAX_EXTEND_SIZE / page->block_size;
```

```
  if (max_extend < MI_MIN_EXTEND) max_extend = MI_MIN_EXTEND;

  if (extend > max_extend) {
    extend = (max_extend==0 ? 1 : max_extend);
  }

  mi_page_free_list_extend(heap, page, extend, stats);
}
void mi_page_to_full(mi_page_t *page, mi_page_queue_t *pq) {
  _mi_page_use_delayed_free(page, MI_USE_DELAYED_FREE);
  if (page->flags.in_full) return;

  mi_page_queue_enqueue_from(&page->heap->pages[MI_BIN_FULL], pq, page);
  mi_page_thread_free_collect(page);
}
mi_page_t *mi_page_fresh(mi_heap_t *heap, mi_page_queue_t *pq) {
  mi_page_t *page = pq->first;
  if (!heap->no_reclaim &&
      _mi_segment_try_reclaim_abandoned(heap, false, &heap->tld->segments) &&
      page != pq->first)
  {
    page = pq->first;
    if (page->free != NULL) return page;
  }
  page = mi_page_fresh_alloc(heap, pq, pq->block_size);
  if (page==NULL) return NULL;
  return page;
}
```

再 下 取, mi_huge_page_alloc也 mi_page_fresh_alloc 取 , 传入 决 何 , mi_segment_page_alloc.

```
mi_page_t *mi_huge_page_alloc(mi_heap_t *heap, size_t size) {
  size_t block_size = _mi_wsize_from_size(size)  *sizeof(uintptr_t);
  mi_page_queue_t *pq = mi_page_queue(heap,block_size);
  mi_page_t *page = mi_page_fresh_alloc(heap,pq,block_size);
  if (page != NULL) {
    mi_heap_stat_increase( heap, huge, block_size);
  }
  return page;
}
mi_page_t *mi_page_fresh_alloc(mi_heap_t *heap, mi_page_queue_t *pq, size_t blo
  mi_page_t *page = _mi_segment_page_alloc(block_size, &heap->tld->segments, &h
  if (page == NULL) return NULL;
  mi_page_init(heap, page, block_size, &heap->tld->stats);
  mi_heap_stat_increase( heap, pages, 1);
  mi_page_queue_push(heap, pq, page);
  return page;
}
mi_page_t *_mi_segment_page_alloc(size_t block_size, mi_segments_tld_t *tld, mi
  mi_page_t *page;
  if (block_size <= (MI_SMALL_PAGE_SIZE / 16) * 3)
    page = mi_segment_small_page_alloc(tld,os_tld);
  else if (block_size <= (MI_MEDIUM_PAGE_SIZE / 16) * 3)
    page = mi_segment_medium_page_alloc(tld, os_tld);
  else if (block_size < (MI_LARGE_SIZE_MAX - sizeof(mi_segment_t)))
    page = mi_segment_large_page_alloc(tld, os_tld);
  else
    page = mi_segment_huge_page_alloc(block_size,tld,os_tld);
  return page;
}
mi_page_t *mi_segment_page_alloc(mi_page_kind_t kind, size_t page_shift, mi_seg
  mi_segment_queue_t *free_queue = mi_segment_free_queue_of_kind(kind,tld);
  if (mi_segment_queue_is_empty(free_queue)) {
    mi_segment_t *segment = mi_segment_alloc(0,kind,page_shift,tld,os_tld);
    if (segment == NULL) return NULL;
    mi_segment_enqueue(free_queue, segment);
  }
  return mi_segment_page_alloc_in(free_queue->first,tld);
}
```

```
mi_segment_t *mi_segment_alloc(size_t required, mi_page_kind_t page_kind, size_
  size_t capacity;
  if (page_kind == MI_PAGE_HUGE) {
    capacity = 1;
  }
  else {
    size_t page_size = (size_t)1 << page_shift;
    capacity = MI_SEGMENT_SIZE / page_size;
  }
  size_t info_size;
  size_t pre_size;
  size_t segment_size = mi_segment_size(capacity, required, &pre_size, &info_si
  size_t page_size = (page_kind == MI_PAGE_HUGE ? segment_size : (size_t)1 << p

  mi_segment_t *segment = NULL;
  segment = mi_segment_cache_find(tld,segment_size);
  if (segment != NULL && mi_option_is_enabled(mi_option_secure) && (segment->pa
    _mi_os_unprotect(segment,segment->segment_size);
  }

  if (segment == NULL) {
    segment = (mi_segment_t*)_mi_os_alloc_aligned(segment_size, MI_SEGMENT_SIZ
    if (segment == NULL) return NULL;
    mi_segments_track_size((long)segment_size,tld);
  }

  memset(segment, 0, info_size);
  if (mi_option_is_enabled(mi_option_secure)) {
    _mi_os_protect((uint8_t*)segment + info_size, (pre_size - info_size) );
    size_t os_page_size = _mi_os_page_size();
    if (mi_option_get(mi_option_secure) <= 1) {
      _mi_os_protect((uint8_t*)segment + segment_size - os_page_size, os_page_s
    }
    else {
      for (size_t i = 0; i < capacity; i++) {
        _mi_os_protect((uint8_t*)segment + (i+1)*page_size - os_page_size, os_p
      }
    }
  }

  segment->page_kind  = page_kind;
  segment->capacity   = capacity;
  segment->page_shift = page_shift;
  segment->segment_size = segment_size;
  segment->segment_info_size = pre_size;
  segment->thread_id  = _mi_thread_id();
  segment->cookie = _mi_ptr_cookie(segment);
  for (uint8_t i = 0; i < segment->capacity; i++) {
    segment->pages[i].segment_idx = i;
  }
  mi_stat_increase(tld->stats->page_committed, segment->segment_info_size);
  return segment;
}
```

再 下 口mi_free, 先 到 ,再 到 . 内 分 则 入local free (则 会 ..), 则 入thread free .

```
void mi_free(void *p) {
  const mi_segment_t *const segment = _mi_ptr_segment(p);
  if (segment == NULL) return;
  bool local = (_mi_thread_id() == segment->thread_id);
  mi_page_t *page = _mi_segment_page_of(segment, p);
  if (page->flags.value==0) {
    mi_block_t *block = (mi_block_t*)p;
    if (local) {
      mi_block_set_next(page, block, page->local_free);
      page->local_free = block;
      page->used--;
      if (mi_page_all_free(page)) { _mi_page_retire(page); }
    }
```

```
    else {
      _mi_free_block_mt(page, block);
    }
  }
  else {
    mi_free_generic(segment, page, local, p);
  }
}
mi_segment_t *_mi_ptr_segment(const void *p) {
  return (mi_segment_t*)((uintptr_t)p & ~MI_SEGMENT_MASK);
}
mi_page_t *_mi_segment_page_of(const mi_segment_t *segment, const void *p) {
  ptrdiff_t diff = (uint8_t*)p - (uint8_t*)segment;
  uintptr_t idx = (uintptr_t)diff >> segment->page_shift;
  return &((mi_segment_t*)segment)->pages[idx];
}

bool mi_page_all_free(const mi_page_t *page) {
  return (page->used - page->thread_freed == 0);
}
```

5. 遗留问题与思考

下 , mimalloc也 到 别 . slab分 , 为什么 升 10% ?

受 ,

1. slab 切A 制 migrate内 , 减 化, 低了 体 .
2. 产 列区分 , local 代 lock free, 升 发 .
3. 制, 似于RCU , 低 争 .
4. 于 到 , 么 出 , .

:

1. bin与page 关 ?
2. OS 口 写.
3. , 内 写 乱了.

分 :

Five100Miles
关 - 0
丝 - 32
+加关

2 0

« 上 : LLVM (7) - 令 side effect
» 下 : LLVM (8) - tablegen介

posted @ 2020-01-09 02:23 Five100Miles (1667) (0) 举

刷 刷

发 , 即 博 %

云 2022 华: 云上 ,
发 VSCode 件 Cnblogs Client For VSCode
不务 业% 做华为云代 业务,
华为 发 专区, 与 发 万 互 世