

# SLUB内存分配器

注: 在本章节中，除了标注取材网络的几张图片，其余图片均为组员绘制。

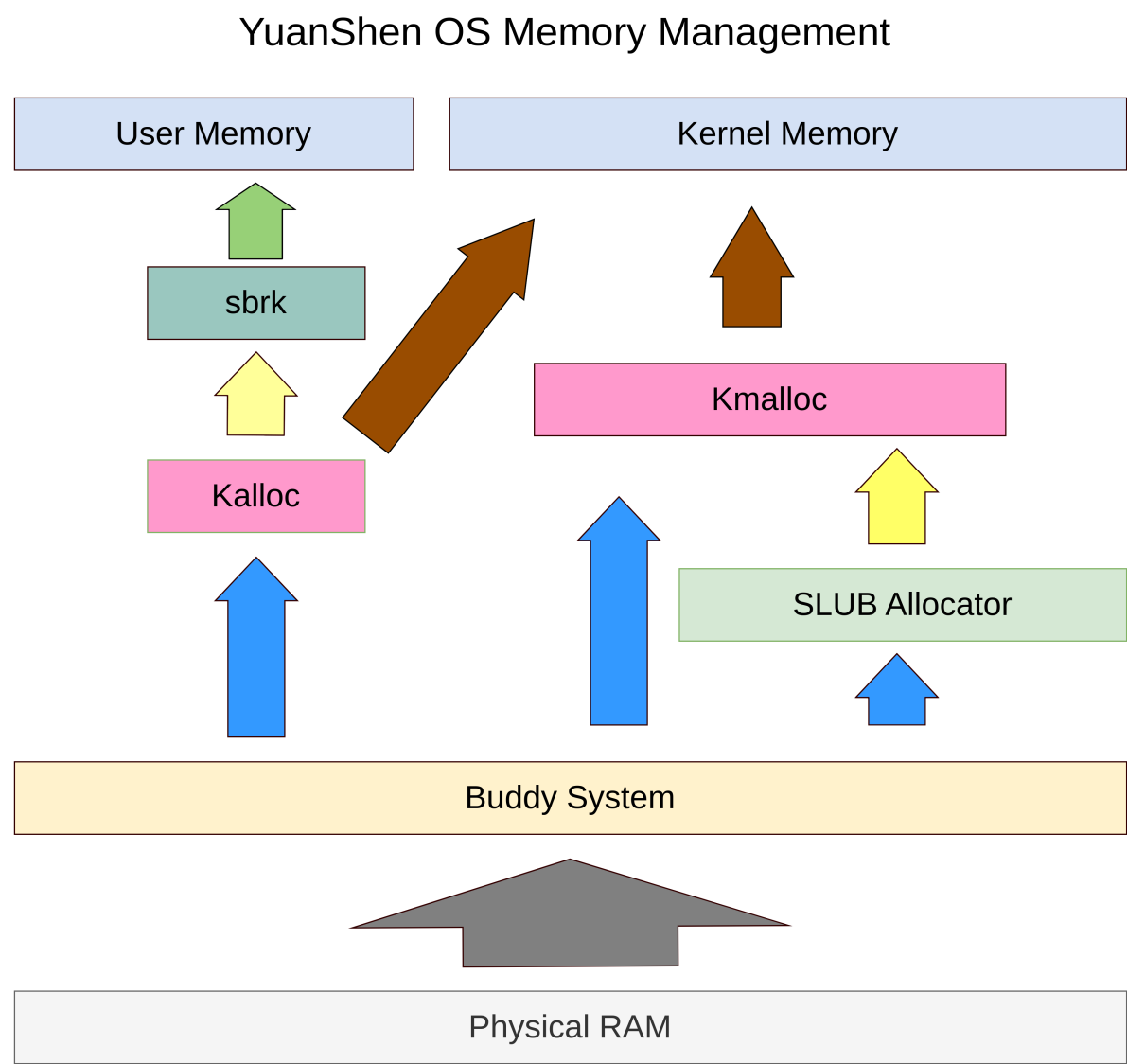


Fig1. YuanShen OS的内存管理系统

伙伴系统提供Linux最为底层的内存管理接口，然而它的最低分配单位为物理页。在很多场景下，内存需要分配的大小是以字节为单位的，达不到一个物理页的大小。如果继续使用伙伴系统进行内存分配，那么就会出现严重的内部碎片问题。因此我们需要更为细粒度的内存分配器。

多年以来，Linux 内核使用一种称为SLUB的内核对象缓冲区分配器。SLUB分配器包含若干个内核对象缓存 `kmem_cache`，这些内核对象就是对应的 `kmem_cache` 的分配单位。

`kmem_cache`有通用和特用之分，通用的对象缓存的功能是提供固定大小的内存块(大小在8到2048字节之间)，而特用的对象缓存则具有对象构造的能力，在分配内存块前会调用构造函数初始化该内存块。特用的 `kmem_cache`可以被专门用于定制化的内核数据结构的构造，例如文件索引结点，网络数据包的缓冲区分配以及各式各样的内核链表。下面的代码是YuanShen OS的SLUB分配器在实现VMA时的应用，详见关于虚拟内存的文档:

```
uint64 do_mmap(struct mm_struct *mm, uint64 addr, uint64 len, uint64
vm_pgoff, uint64 vm_filesz, int flags, struct file *f, int prot){
    //...检查入参
    struct vm_area_struct *ptr = (struct vm_area_struct*)
kmalloc(sizeof(struct vm_area_struct));
    if(ptr == 0)
        return -1;
    //...设置vma
    return ptr->vm_start;
}
```

YuanShen OS借鉴了Linux的内存管理系统架构，也配置了简化版的SLUB分配器，出于时间考量，去除了per cpu的kmem\_cache\_cpu的设计，让所有处理器共享一个分配器接口。也去除了kmem\_cache\_node的二级缓存设计，只采用一级缓存。

最后，我们利用SLUB分配器和伙伴系统实现了kmalloc和kfree，当需要的空间大小大于一页时使用伙伴系统分配器，否则使用SLUB分配器。

## 1.1 页面元信息

我们使用pages数组统一管理所有的物理页。第一项refcount被用于COW fork，与本节无关。page结构体的定义如下列代码所示：

```
//slub.h
enum { PAGE_FREE, PAGE_BUDDY, PAGE_SLUB, PAGE_RESERVE }; //页面状态

struct page {
    //COW:
    int refcount;
    //SLUB ALLOCATOR:
    uint32 active_obj_count; //未回收的对象数量
    uint64 flags;
    void *freelist; //由header管理,其它的页不设置该字段,其指向本slab页面内部维护的
freelist的第一个对象节点
    struct kmem_cache *slab;
    struct page *header; //the beginning page of a slab
    struct page *next; //指向下一个slab(的header页),该字段只由header管理
};
extern struct page pages[PGREF_MAX_ENTRIES];
```

其中，为了区分SLUB分配器和伙伴系统分配的页面，我们引入了flags的字段，当页面被分配的时候这个字段会被设置，而回收的时候，kfree会查询地址对应的页面对应的flags，若为PAGE\_SLUB,则将空闲区间归还SLUB分配器，反之如果为PAGE\_BUDDY,则使用buddy\_free\_将区间归还伙伴系统。

先前提到的对象缓存kmem\_cache每次在空闲页用完后都会向伙伴系统申请**连续**的若干物理页，称作一个slab，也就是块的意思。然后对象缓存会在一个slab对应的物理页内建立free list的组织结构。我们可以把一个slab看作是一张表，那么表头就保存在那个slab对应的页面的元信息的字段header对应的page结构体内。

这句话说了很绕，实际上就是我们会把一个slab对应的几个连续的page结构体都标记为PAGE\_SLUB,并把这个slab的元信息全部放进表头的那个结构体。YuanShen OS的SLUB分配器的内部结构如下图所示：

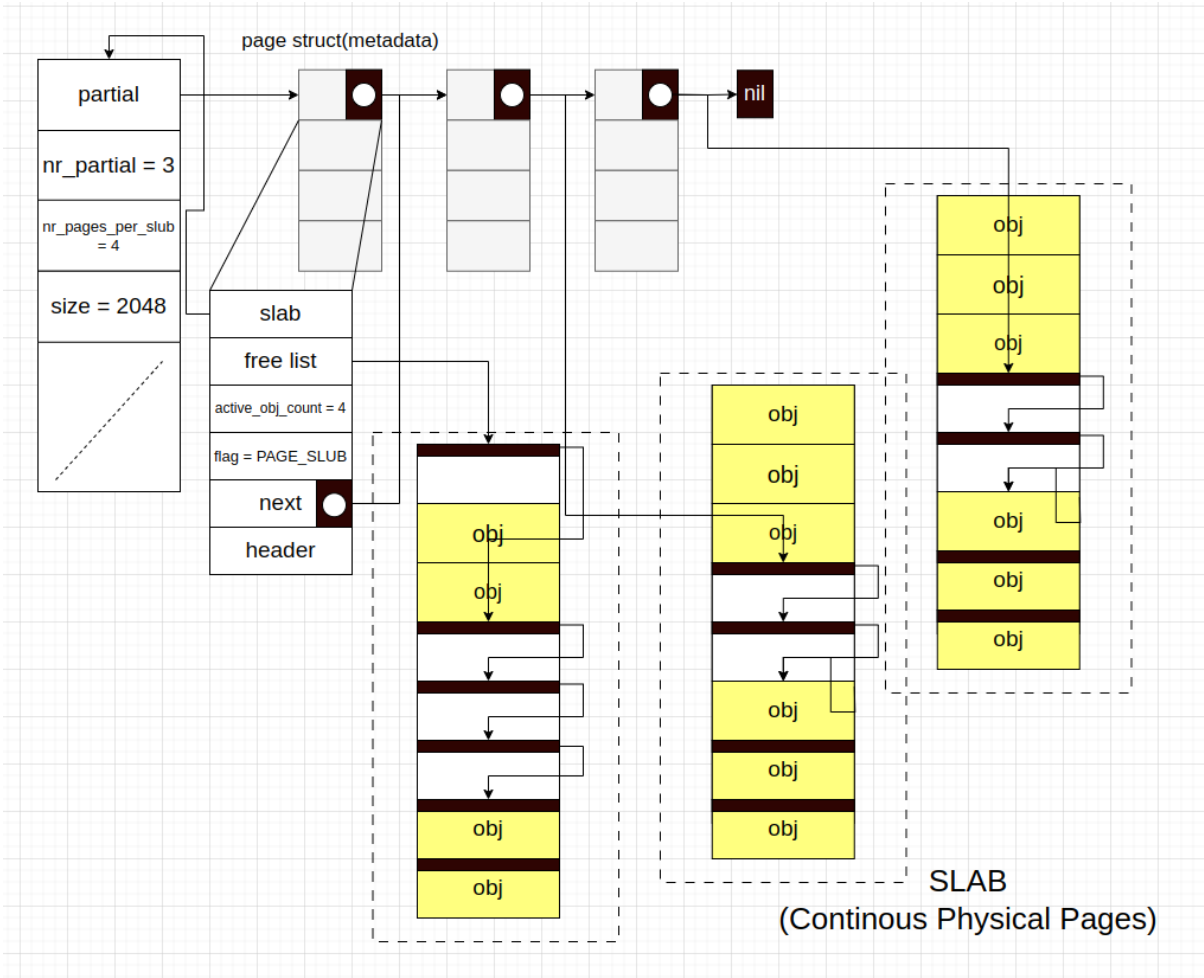


Fig2. YuanShen OS的SLUB分配器原理图

上图的链表由page结构体组成的。若干个连续的page结构体，在图中是连续四个(标记为灰色),表示一个slab。它们维护了这个slab的内存配置的元信息，这些元信息存储在第一个page结构体，也就是表头内，内容有：

- active\_obj\_count，意指这个slab已经分配出去的对象数量。
- freelist，指向这个slab内的第一个空闲对象。
- next，这个字段只在表头结构体内才有，它指向下一个slab的表头，因为kmem\_cache的内存池里可以有不止一个slab。
- slab，这个字段表示它是从哪个kmem\_cache分配出去的。它在对象回收的时候很有用。

1.2 SLAB页的内部结构与kmem\_cache

我们不妨先看看kmem\_cache和slab究竟长什么样。如图，一个slab包含连续的若干物理页，不同的slab由链表串联。在YuanShen OS里，这个链表在刚才提到的page结构体中维护。每个slab对应的表头页结构体中的next会指向下一个slab对应的表头页结构体，而每个slab内部对象的free list也是维护在表头的。

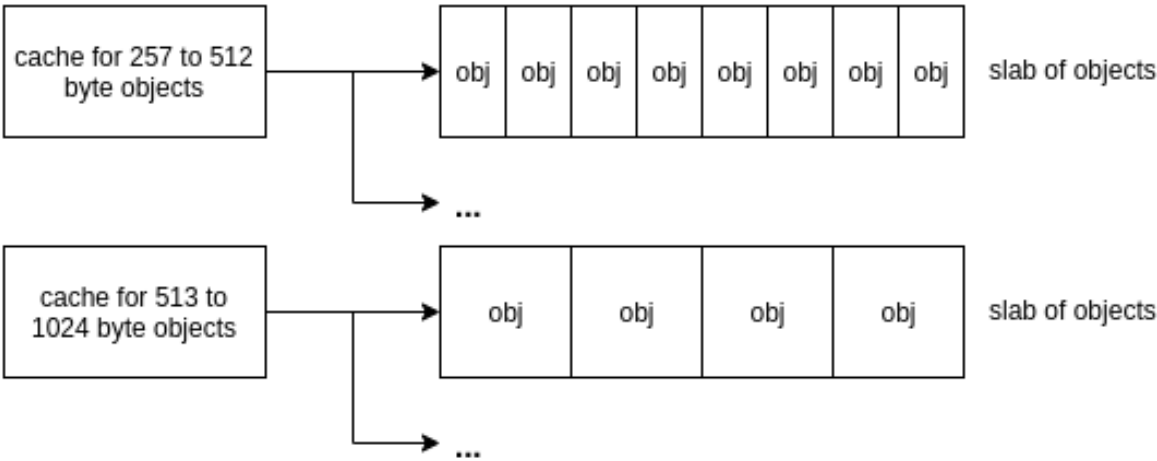


Fig3. kmem\_cache将若干连续的物理页划分为无数对象(网图)

kmem\_cache就是我们说的内核对象分配器，通过缓存机制一次性向伙伴系统申请一个大块空间，并将空间初始化为小型内核对象。最值得注意的是在组织对象在slab内的内部结构时我们会将对象做八字节对齐，因此不满8字节大小的对象也会在slab中占据8字节的区域。

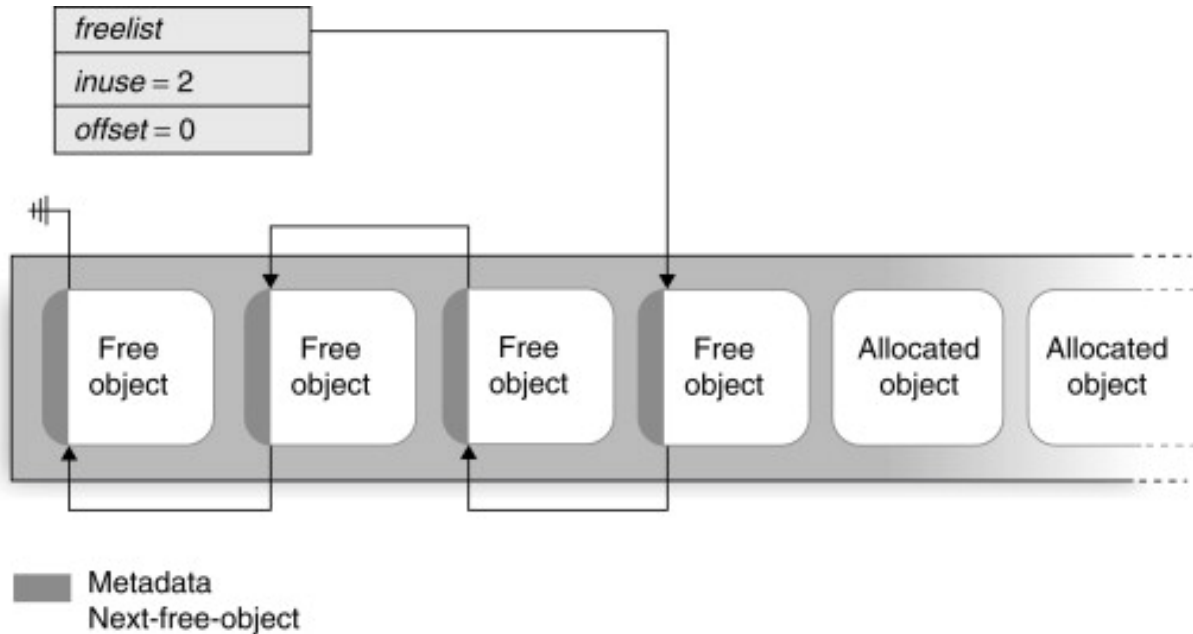


Fig4. 每个slab内部对象的自由链表的结构(网图)

一是因为对齐访问具有较高的性能，二是因为维护对象的freelist的metadata,也就是64位指针正好需要八字节。如上图所示，自由对象的区域的开头在未被分配的时候可以用于存放链表的指针。

```
/*出于简化，我们不考虑CPU和node区分的两种情况，只使用一条partial list。因此
也不会有per cpu的partial list*/
struct kmem_cache {
    /* kmem_cache_cpu */
    void **freelist; /* Pointer to next available object */
    uint64 tid; /* Globally unique transaction id */
    struct page *page; /* The slab from which we are allocating */

    /* Used for retrieving partial slabs, etc. */
    int refcount;
    uint64 min_partial;
```

```

uint32 size;          /* The size of an object including metadata */
uint32 object_size;   /* The size of an object without metadata */
uint32 offset; /* Free pointer offset */
uint64 nr_page_per_slub;

void (*ctor)(void *);
uint32 inuse;
uint32 align;
uint32 red_left_pad;
const char *name;
uint64 nr_partial;
struct page *partial_list; /* partial List of slabs */
};

```

在kmem\_cache里，freelist表示当前kmem\_cache正在存取的slab中的自由对象。如果它非空的话，就会调整freelist，取出头结点，并使用ctor这个函数指针作用在这个对象的尚未初始化的区域上，实现自动化对象构造。page则是指向这个slab对应的表头页结构体。

min\_partial字段则是为了维持缓存的性能。SLUB分配器规定，如果当前的partial链表，也就是所有slab组织成的链表的长度大于min\_partial，那么当某个slab的对象全部归还，也就是active\_obj\_count变回0的时候，这个slab就应该还给伙伴系统。

## 2. 核心函数

我们挑选一两个函数来展示YuanShen OS的SLUB分配器工作原理。

### 2.1 slab页初始化

这个函数用于初始化SLAB页，分配器会向伙伴系统申请连续的(必须连续，不然当前的表结构就无法维护)，长度为cache->nr\_page\_per\_slub的页面。显然它的工作就是在这个slab里初始化一堆指针，用于组织free list，实现逻辑如下列代码所示：

```

void *init_object_list(void *addr, uint32 objsize, uint32 size) {
    uint64 end = (uint64)addr + size;
    uint64 this_obj = (uint64)addr;
    uint64 next_obj = (uint64)addr + objsize;

    while (next_obj < end) {
        *(void **)(this_obj) = (void *)next_obj;
        this_obj += objsize;
        next_obj += objsize;
    }

    if (next_obj == end) {
        *(void **)(this_obj) = 0;
        return (void *)this_obj;
    } else {
        *(void **)(this_obj - objsize) = 0;
        return (void *)(this_obj - objsize);
    }
}

```

```
    }
}
```

## 2.2 对象归还

对象归还的核心思路就是先把对象插回对应的slab页，调整header维护的freelist，然后检查该slab内活跃对象的数量。如果所有的对象已经归还，那么就要检查slab的数量nr\_partial是否超过min\_partial。

如果是的话，就要把slab归还伙伴系统，这样就让partial链表不会太长，有助于分配器提升性能。

```
void kmem_cache_free(void *obj) {
    if(obj == 0)
        return;
    struct page *header = pages[PA2PGREF_ID((uint64)obj)].header;
    struct kmem_cache *cache = header->slab;

    *((void**)obj) = header->freelist;
    header->freelist = obj;
    if(header == cache->page)
        cache->freelist = obj;
    header->active_obj_count--;

    if(header->active_obj_count == 0 && cache->min_partial < cache-
>nr_partial && header != cache->page){
        //删除partial list中的header
        struct page *ptr = cache->partial_list;
        //由于不可能为空
        for(; ptr->next != header; ptr = ptr->next){
            if(ptr == 0)
                panic("kmem_cache_free");
        }
        ptr->next = header->next;

        .....//这里省略的逻辑是把slab还给伙伴系统，然后把pages数组维护的页面元信息重
置。

        cache->nr_partial--;
    }
}
```

## 3.1 kmalloc实现

有了SLUB分配器之后，kmalloc的实现变得极为显然。在内核启动的过程中YuanShen OS会事先初始化若干个kmem\_cache，对应不同的内存块大小。如果某个kmem\_cache适合完成当前的分配任务，那么我们就使用slub分配器进行分配。

这个过程中对应的slab页面会被打上PAGE\_SLUB的flag，这个信息被用于kfree的回收。

由于slub分配器主要完成小规模分配任务，因此当request size为页级时，kmalloc会调用伙伴系统接口。同样，对应的伙伴区间的页面结构体会被标记为PAGE\_BUDDY。

kmalloc函数的实现逻辑如下代码所示:

```
void *kmalloc_(uint32 size) {
    int objindex;
    void *p = 0;
    if (size == 0) return 0;

    for (objindex = 0; objindex < SLUB_NUM; objindex++) {
        if (size <= kmem_cache_objsize[objindex]) {
            p = kmem_cache_alloc(slub_allocator[objindex]);
            return p;
        }
    }

    // size 若不在 kmem_cache_objsize 范围之内, 则使用 buddy system 来分配内存
    if (objindex >= SLUB_NUM) {
        uint32 s = 1024;
        while (s < size)
            s = s << 1;

        p = (void*)buddy_alloc_((size - 1) / PGSIZE + 1);
    }

    return p;
}
```

## 4. 测试

### 4.1 stress\_test 测试

我们采用的压力测试会尝试使用kmalloc配置从1到MAX\_ALLOC\_SIZE区间上所有尺寸对应的对象。由于无论是缓存还是伙伴系统，通常分配的内存块都是2的幂次的大小，因此已经覆盖了多次使用同一个模块的测试。

```
void stress_test() {
    printf("Running stress test with %d allocations...\n", TEST_ROUNDS);

    void *ptrs[TEST_ROUNDS];
    for (int i = 0; i < TEST_ROUNDS; i++) {
        int size = (i % MAX_ALLOC_SIZE) + 1; // Vary sizes from 1 to
        MAX_ALLOC_SIZE
        ptrs[i] = kmalloc(size);

        if (!ptrs[i]) {
            printf("Test failed: kmalloc failed on round %d for size
%d.\n", i, size);
            panic("test fail");
        }
    }
}
```

```
printf("Stress allocation successful. Freeing memory...\n");

for (int i = 0; i < TEST_ROUNDS; i++) {
    kfree(ptrs[i]);
}
for(int i = 0; i < SLUB_NUM; ++i) {
    test_kmem_cache_full(slub_allocator[i]);
    test_kmem_cache_nr_partial(slub_allocator[i]);
    test_kmem_cache_free_list(slub_allocator[i]);
}
printf("Stress test passed.\n");
}
```

这三个测试函数的强度逐渐提高，第一个验证kfree后对应页面的结构体维护的active\_obj\_count是否变回0,第二个通过链表遍历验证slab的数量是否的确为kmem\_cache维护的nr\_partial，第三个最为苛刻，会遍历每个slab页面的freelist，进行计数，最后与这个slab维护的理论对象数进行比较。

```
test_kmem_cache_full(slub_allocator[i]);
test_kmem_cache_nr_partial(slub_allocator[i]);
test_kmem_cache_free_list(slub_allocator[i]);
```

在main.c中打开宏test\_kmalloc即可进行功能测试，运行结果如下图所示：

```
xv6 kernel is booting

Running simple allocation test...
hart 1 starting
Simple allocation successful. Freeing memory...
hart 2 starting
Simple allocation test passed.
Running stress test with 300 allocations...
Stress allocation successful. Freeing memory...
Stress test passed.
```

成功通过测试。

## 参考材料

[1] smcdef. "Linux SLUB分配器详解." 蜗窝科技, 2018-02-22.  
[http://www.wowotech.net/memory\\_management/426.html](http://www.wowotech.net/memory_management/426.html).