**Linux内核slab内存分配器对象化实现 Project123 vvv小队**

**【项目描述】**

为了解决小块内存的分配，Linux内核基于Solaris 2.4中的slab分配算法实现了自己的slab分配器。除此之外，slab分配器另一个主要功能是作为一个高速缓存，它用来存储内核中那些经常分配并释放的对象。C++语言提供了面向对象的语言层面支持，使对象的管理更为清晰。本项目拟利用C++语言对Linux内核slab分配算法进行重新改写，实现面向对象的块内存分配与管理。

# 【设计思路】

Linux内核中基于伙伴算法实现的分区页框分配器适合大块内存的请求，它所分配的内存区是以页框为基本单位的。有时候，我们不需要一下子分配很大的内存，对于内核中小块连续内存的请求，例如分配一个 task\_struct 结构，只需要分配小块的内存，去存储这个进程描述结构的对象。为了满足对这种小内存块的需要，Linux 系统采用了一种被称为 slab 分配器的技术，用于分配称为 slab 的一小块内存。它的基本原理是从内存管理模块申请一整块页，然后划分成多个小块的存储池，用复杂的队列来维护这些小块的状态（状态包括：被分配了 / 被放回池子 / 应该被回收）

### **1.slab分配器的基本原理**

为每种对象类型创建一个内存缓存，每个内存缓存由多个大块组成，一个大块是一个或多个连续的物理页，每个大块包含多个对象。slab采用面向对象的思想，基于对象类型管理内存，每种对象被划分为一个类，比如进程描述符（task\_struct）是一个类，每个进程描述符实现是一个对象。

slab 分配器由kmem\_cache --> kmem\_cache\_node --> object三个结构相结合地方式来描述，其中cache由 struct kmem\_cache来描述，struct kmem\_cache.中的成员struct kmem\_cache\_node，lists数组包含三个slab list:full,part,free, 这三个list是有struct slab结构组成的链表。其中struct slab中的s\_mem是第一个object的起始地址，因此就可以通过kmem\_cache --> kmem\_cache\_node --> object找到空闲的object。其中slab中的page 的prev指向cache，next指向slab。

而slab还分为on-slab 和off-slab，是指当object的size特别大的时候，slab中新分配的page就全部放object，而struct slab结构单独从通用slab中进行分配，如果object的size比较小，则可以将struct slab放置在新分配的page开头，然后再放置object。

所有 CPU 共享的空闲对象链表

在 struct kmem\_cache\_node结构体中同样存在一个 array\_cache的成员 *struct array\_cache*shared;\*，其构成一个所有 CPU 共享的缓存。

在此基础上，一个常规的对象申请流程是这样的：

内核首先会从本地 CPU 空闲对象链表中尝试获取一个对象用于分配：如果失败，则检查所有CPU共享的空闲对象链表链表中是否存在，并且空闲链表中是否存在空闲对象，若有就转移 batchcount 个空闲对象到本地 CPU空闲对象链表中；

如果第 1 步失败，就尝试从 SLAB中分配；这时如果还失败，kmem\_cache会尝试从页框分配器中获取一组连续的页框建立一个新的SLAB，然后从新的SLAB中获取一个对象。

对象释放流程如下： - 首先会先将对象释放到本地CPU空闲对象链表中，如果本地CPU空闲对象链表中对象过多，kmem\_cache 会将本地CPU空闲对象链表中的batchcount个对象移动到所有CPU共享的空闲对象链表链表中， - 如果所有CPU共享的空闲对象链表链表的对象也太多了，kmem\_cache也会把所有CPU共享的空闲对象链表链表中batchcount个数的对象移回它们自己所属的SLAB中， - 这时如果SLAB中空闲对象太多，kmem\_cache会整理出一些空闲的SLAB，将这些SLAB所占用的页框释放回页框分配器中。

### **2.slab分配器的作用**

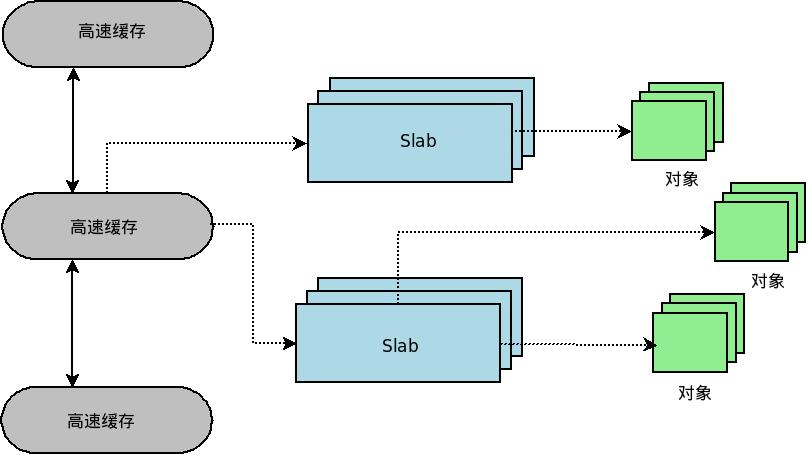
a.能够分配更小块的内存，可以帮助消除伙伴分配器原本会造成的内部碎片问题

b.缓存常用的object，因此内核不会在分配, 初始化和销毁object上浪费时间。

c.作为一个高速缓存，它用来存储内核中那些经常分配并释放的对象。通过着色技术调整对象以更好的使用硬件高速缓存

### **3.slab分配器的结构**

slab分配器为每种对象分配一个高速缓存，这个缓存可以看做是同类型对象的一种储备。每个高速缓存所占的内存区又被划分多个slab，每个 slab是由一个或多个连续的页框组成。每个页框中包含若干个对象，既有已经分配的对象，也包含空闲的对象。slab分配器的大致组成图如下：



# 【实现描述】

**malloc.cpp**

#include "slab\_allocator.h"

const size\_t PAGE\_SIZE = 0x1000;

static Slab<0x010, PAGE\_SIZE> slab\_0x10;

static Slab<0x020, PAGE\_SIZE> slab\_0x20;

static Slab<0x040, PAGE\_SIZE> slab\_0x40;

static Slab<0x060, PAGE\_SIZE> slab\_0x60;

static Slab<0x100, PAGE\_SIZE> slab\_0x100;

static Slab<0x200, PAGE\_SIZE> slab\_0x200;

static Slab<0x300, PAGE\_SIZE> slab\_0x300;

void init() {

slab\_0x10.init();

slab\_0x20.init();

slab\_0x40.init();

slab\_0x60.init();

slab\_0x100.init();

slab\_0x200.init();

slab\_0x300.init();

}

void\* custom\_malloc(size\_t size) {

if (size < 0x10) {

return slab\_0x10.alloc();

} else if (size < 0x20) {

return slab\_0x10.alloc();

} else if (size < 0x40) {

return slab\_0x40.alloc();

} else if (size < 0x60) {

return slab\_0x60.alloc();

} else if (size < 0x100) {

return slab\_0x100.alloc();

} else if (size < 0x200) {

return slab\_0x200.alloc();

} else if (size < 0x500) {

return slab\_0x300.alloc();

} else {

return nullptr;

}

}

void custom\_free(void\* address) {

slab\_0x10.free(address);

slab\_0x20.free(address);

slab\_0x40.free(address);

slab\_0x60.free(address);

slab\_0x100.free(address);

slab\_0x200.free(address);

slab\_0x300.free(address);

}

**slab\_allocator.h:**

template<size\_t slab\_size, size\_t memory\_size> class Slab;

template<size\_t slab\_size, size\_t memory\_size, size\_t max\_blocks = memory\_size / slab\_size> struct SlabHeader {

Slab<slab\_size, memory\_size>\* prev, \* next;

Bitmap<max\_blocks> mem\_map;

size\_t free\_blocks;

size\_t next\_fit\_block;

};

template<size\_t slab\_size, size\_t memory\_size> class Slab {

private:

const static size\_t MAX\_HEADER\_SIZE = sizeof(SlabHeader<slab\_size, memory\_size>);

const static size\_t MAX\_BLOCKS = (memory\_size - MAX\_HEADER\_SIZE) / slab\_size;

static\_assert(memory\_size > MAX\_HEADER\_SIZE);

static\_assert((slab\_size + MAX\_HEADER\_SIZE) <= memory\_size);

SlabHeader<slab\_size, memory\_size, MAX\_BLOCKS> header;

char blocks[MAX\_BLOCKS][slab\_size];

bool is\_address\_in\_slab(void\* address);

void\* alloc\_in\_current\_slab(size\_t block\_index);

void\* alloc\_in\_new\_slab();

void free\_from\_current\_slab(size\_t block\_index);

void free\_from\_next\_slab(void\* address);

void\* request\_memory\_from\_os(size\_t size);

void free\_memory\_to\_os(void\* addrss, size\_t size);

public:

void init(Slab\* prev = nullptr);

void\* alloc();

void free(void\* address);

};

template<size\_t slab\_size, size\_t memory\_size>

void Slab<slab\_size, memory\_size>::init(Slab\* prev) {

header.prev = prev;

header.next = nullptr;

header.free\_blocks = MAX\_BLOCKS;

header.next\_fit\_block = 0;

header.mem\_map.init();

}

template<size\_t slab\_size, size\_t memory\_size>

void\* Slab<slab\_size, memory\_size>::alloc() {

size\_t block\_index = -1;

if (header.free\_blocks &&

((block\_index = header.mem\_map.find\_unused(header.next\_fit\_block)) != BITMAP\_NO\_BITS\_LEFT)) {

return alloc\_in\_current\_slab(block\_index);

} else {

return alloc\_in\_new\_slab();

}

}

template<size\_t slab\_size, size\_t memory\_size>

void Slab<slab\_size, memory\_size>::free(void\* address) {

if (is\_address\_in\_slab(address) == false) {

return free\_from\_next\_slab(address);

}

size\_t block\_index = (uintptr\_t(address) - uintptr\_t(blocks)) / slab\_size;

assert(header.mem\_map.check\_used(block\_index));

free\_from\_current\_slab(block\_index);

}

template<size\_t slab\_size, size\_t memory\_size>

bool Slab<slab\_size, memory\_size>::is\_address\_in\_slab(void\* address) {

if ((address >= blocks) && (address <= &blocks[MAX\_BLOCKS - 1][slab\_size - 1])) {

return true;

} else {

return false;

}

}

template<size\_t slab\_size, size\_t memory\_size>

void\* Slab<slab\_size, memory\_size>::alloc\_in\_new\_slab() {

Slab\* new\_slab = static\_cast<Slab\*>(request\_memory\_from\_os(sizeof(Slab)));

if (!new\_slab) {

return nullptr;

}

new\_slab->init(this);

header.next = new\_slab;

return new\_slab->alloc();

}

template<size\_t slab\_size, size\_t memory\_size>

void\* Slab<slab\_size, memory\_size>::alloc\_in\_current\_slab(size\_t block\_index) {

header.mem\_map.set\_used(block\_index);

header.next\_fit\_block = (block\_index + 1) % MAX\_BLOCKS;

header.free\_blocks--;

return static\_cast<void\*>(blocks[block\_index]);

}

template<size\_t slab\_size, size\_t memory\_size>

void Slab<slab\_size, memory\_size>::free\_from\_current\_slab(size\_t block\_index) {

header.mem\_map.set\_unused(block\_index);

header.next\_fit\_block = block\_index;

header.free\_blocks++;

if ((header.free\_blocks == 0) && (header.prev)) {

//slab is empty, and it's not the first;

header.prev->header.next = nullptr;

free\_memory\_to\_os(this, sizeof(Slab));

//The slab committed suicide, don't ever use it again!

}

}

template<size\_t slab\_size, size\_t memory\_size>

void Slab<slab\_size, memory\_size>::free\_from\_next\_slab(void\* address) {

if (header.next) {//if there is another slab in the list check on it too.

header.next->free(address);

return;

} else {

//address doesn't belong any slab.

return;

}

}

template<size\_t slab\_size, size\_t memory\_size>

void\* Slab<slab\_size, memory\_size>::request\_memory\_from\_os(size\_t size) {

//system dependent function, returns aligned memory region.

return VirtualAlloc(0, size, MEM\_COMMIT, PAGE\_READWRITE);

}

template<size\_t slab\_size, size\_t memory\_size>

void Slab<slab\_size, memory\_size>::free\_memory\_to\_os(void\* addrss, size\_t size) {

//system dependent function, returns aligned memory region.

VirtualFree(addrss, size, MEM\_FREE);

}

**Bitmap.h**

#pragma once

#include <cstdint>

#include <assert.h>

#include <cstring>

#define CHECK\_BIT(value, bit) ((value >> bit) & 1)

#define BITMAP\_NO\_BITS\_LEFT 0xFFFFFFFF

template <size\_t SIZE> class Bitmap {

private:

uint8\_t m\_bitmap\_data[SIZE];

public:

void init();

void set\_used(unsigned position);

void set\_unused(unsigned position);

unsigned find\_unused(unsigned search\_start = 0);

unsigned find\_used(unsigned search\_start = 0);

bool check\_used(unsigned position);

bool check\_unused(unsigned position);

};

template <size\_t SIZE> void Bitmap<SIZE>::init() {

memset(m\_bitmap\_data, 0, sizeof(m\_bitmap\_data));

}

template <size\_t SIZE> void Bitmap<SIZE>::set\_used(unsigned position) {

assert(position < SIZE);

m\_bitmap\_data[position / 8] |= (1 << (position % 8));

}

template <size\_t SIZE> void Bitmap<SIZE>::set\_unused(unsigned position) {

assert(position < SIZE);

m\_bitmap\_data[position / 8] &= ~(1 << (position % 8));

}

template <size\_t SIZE> unsigned Bitmap<SIZE>::find\_unused(unsigned search\_start) {

assert(search\_start < SIZE);

size\_t bit\_index = search\_start;

while (bit\_index < SIZE) {

if (m\_bitmap\_data[bit\_index / 8] == 0xFF) {

bit\_index += 8;

continue;

}

if (!CHECK\_BIT(m\_bitmap\_data[bit\_index / 8], bit\_index % 8))

return bit\_index;

bit\_index++;

}

return BITMAP\_NO\_BITS\_LEFT;

}

template <size\_t SIZE> unsigned Bitmap<SIZE>::find\_used(unsigned search\_start) {

assert(search\_start < SIZE);

size\_t bit\_index = search\_start;

while (bit\_index < SIZE) {

if (m\_bitmap\_data[bit\_index / 8] == 0) {

bit\_index += 8;

continue;

}

if (CHECK\_BIT(m\_bitmap\_data[bit\_index / 8], bit\_index % 8))

return bit\_index;

bit\_index++;

}

return BITMAP\_NO\_BITS\_LEFT;

}

template <size\_t SIZE> bool Bitmap<SIZE>::check\_used(unsigned position) {

return CHECK\_BIT(m\_bitmap\_data[position / 8], position % 8);

}

template <size\_t SIZE> bool Bitmap<SIZE>::check\_unused(unsigned position) {

return !CHECK\_BIT(m\_bitmap\_data[position / 8], position % 8);

}

# 【开发遇到的问题和解决方法】

为了理解slab算法的概念及其基本应用，我组查找了相应的文献和教学资源理解了slab算法的实现过程和原理

在Slab缓存分配过程中，对于如何查找可用的内存时，出现了分歧，将slab kmem\_cache\_zalloc()函数中的调用过程不足够理解，导致后续的问题，无法运行。在经过查错和网上对比资料将其修正。

kmem\_cache\_zalloc(include**/**linux**/**slab.h)

kmem\_cache\_alloc(mm**/**slab.c)

slab\_alloc(mm**/**slab.c)

\_\_do\_cache\_alloc(mm**/**slab.c)

\_\_\_\_cache\_alloc(mm**/**slab.c)

cpu\_cache\_get(mm**/**slab.c)

cache\_alloc\_refill(mm**/**slab.c)