高并发内行池设计 (C++)

实现高并发内存池 (C++)*哔哩哔哩*bilibili

或高并发实现高效内存管理

或高效内存池设计与实现

- 背景
  - 实现了高效的多线程内存管理，用于替代系统的内存 分配相关的函数（malloc 、 free ）
  - C/C++ 、数据结构（链表、哈希桶）、操作系统内存管理、单例模式、多线程、互斥锁
- 我的工作
  - 模拟实现出一个自己的高并发内存池
  - **a.** 性能问题

    b. 多线程环境下，锁竞争问题

    c. 内存碎片问题
- 实现结果

  - 增加动态申请的效率
  - 减少陷入内核的次数
  - 减少系统内存碎片
  - 提升内存使用率
  - 尽量减少锁竞争
  - 应用于多核多线程场景

目前许多开发场景都是多核多线程，在申请内存的场景下，存在激烈的所竞争问题，tcmalloc在多线程高并发场景下更好用，所以实现的内存池需要考虑以下方面：
1.性能问题
2.多线程环境下，锁的竞争问题
3.内存碎片问题；

高并发内存池设计主要由3各部分组成：

1. 为每个线程创建一个固定大小的独享线程缓存thread cache，线程从这里申请内存不需要加锁，以实现并发线程池的高效性能。
2. 实现一个中心缓存central cache， 其存储结构设计为哈希桶，为所有线程锁共享。Thread cache是按需访问central cache中大小不同的桶。采用单例模式并实现对桶加锁，以避免多线程访问下的锁的竞争问题。
3. 实现一会页缓存page cache，以实现对central cache的资源补充、向操作系统申请资源和资源回收。采用基数树实现空间回收。回收span对象时合并相邻的页，以组成更大的页，缓解内存碎片的问题

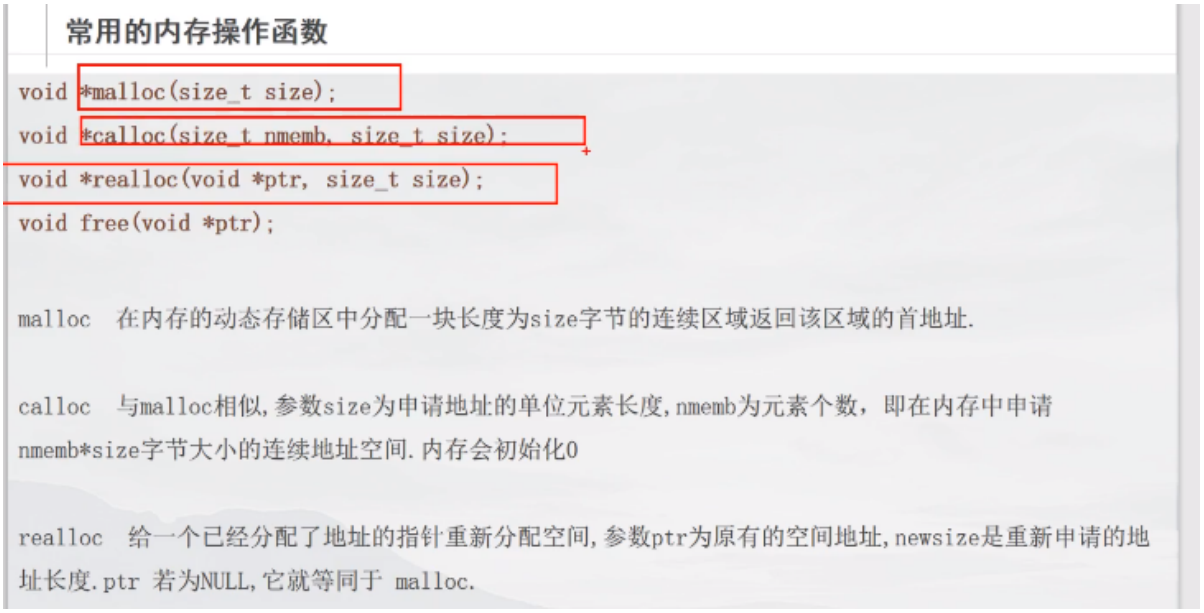
4. 向central cache分配时，实现对span定长大小的小块内存切割，

   central合适的试剂回收thread cache中的对象，避免一个线程占用了太多的内存，而其他线程的内存不够用，达到内存分配在多线程中更均衡的按需调度的目的。central cache 是存在竞争的所以从这里取内存对象是需要加锁的。首先者利用的是桶锁，其次只有thread cache的没有内存对象时才会找central cache，所以这里竞争不会很激烈。

5. page cache：页缓存是在central cache缓存上面的一层缓存，存储的内存是以页单位存储及分配的，central cache没有内存对象时，从page cache分配处一定数量的page，并切割成定长大小的小块内存，分配给central cache。当一个span的几个跨度页的对象都回收以后，page cache会回收central cache满足条件的span对象，并且合并相邻的页，组成更大的页，缓解内存碎片的问题。

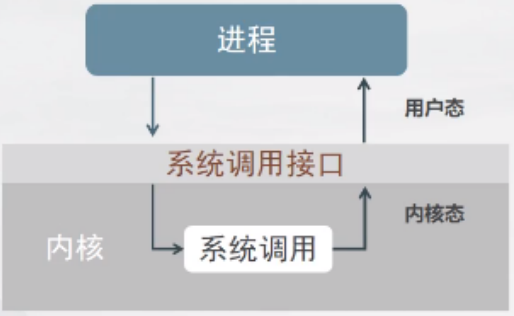
管理页数的span

模拟实现出一个自己的高并发内存池，在多线程环境下缓解了锁竞争问题，相比于malloc/free效率提高了25%左右，将内存碎片保持在10%左右。



弊端一

高并发时较小内存块使用导致系统调用频繁，降低了系统的执行效率



弊端二

频繁使用时增加了系统内存的碎片，降低内存使用效率

内部碎片 — 已经被分配出去（能明确指出属于哪个进程）却不能被利用的内存空间；

产生根源：1.内存分配必须起始于可被 4、8 或 16 整除（视处理器体系结构而定）的地

2.MMU的分页机制的限制

处理器	页大小	分页的级别	虚拟地址分级
x86	4KB	2	10+10+12
x86(extended)	4KB	1	10+22
x86(PAE)	4KB	3	2+9+9+12
x86-64	4KB	4	9+9+9+9+12

弊端三

没有垃圾回收机制，容易造成内存泄漏，导致内存枯竭

情形一：

```
void log_error(char *reason) {
    char *p1;
    p1 = malloc(100);
    sprintf(p1,"The f1 error occurred because of '%s'.", reason);
    log(p1);
    }
```

情形二：

```
int getkey(char *filename) {
    FILE *fp;
    int key;
    fp = fopen(filename, "r");
    fscanf(fp, "%d", &key);
    //fclose(fp);
    return key;
}
```

弊端四

内存分配与释放的逻辑在程序中相隔较远时，降低程序的稳定性

```
ret get_stu_info( Student * _stu ) {
    char * name= NULL;
    name = getName(_stu->no);
    //处理逻辑
    if(name) {
        free(name);
        name = NULL;
    }
}
```

```
char stu_name[MAX];

char * getName(int stu_no){

    //查找相应的学号并赋值给 stu_name
    snprintf(stu_name,MAX,"%s",name);
    return stu_name;
}
```

	PtMalloc (glibc 自带)	TcMalloc	JeMalloc
概念	Glibc 自带	Google 开源	Jason Evans （FreeBSD著名开发人员）
性能 (一次malloc/free 操作)	300ns	50ns	<=50ns
弊端	锁机制降低性能，容易导致内存碎片	1%左右的额外内存开销	2%左右的额外内存开销
优点	传统，稳定	线程本地缓存，多线程分配效率高	线程本地缓存，多核多线程分配效率相当高
使用方式	Glibc 编译	动态链接库	动态链接库
谁在用	较普遍	safari、chrome等	facebook、firefox 等
适用场景	除特别追求高效内存分配以外的	多线程下高效内存分配	多线程下高效内存分配