操作系统第五次课设报告

2021.12.23

题目一：tcmalloc就是一个内存分配器，管理堆内存，主要影响malloc和free，用于降低频繁分配、释放内存造成的性能损耗，并且有效地控制内存碎片。glibc中的内存分配器是ptmalloc2，tcmalloc号称要比它快。一次malloc和free操作，ptmalloc需要300ns，而tcmalloc只要50ns。同时tcmalloc也优化了小对象的存储，需要更少的空间。tcmalloc特别对多线程做了优化，对于小对象的分配基本上是不存在锁竞争，而大对象使用了细粒度、高效的自旋锁（spinlock）。分配给线程的本地缓存，在长时间空闲的情况下会被回收，供其他线程使用，这样提高了在多线程情况下的内存利用率，不会浪费内存，而这一点ptmalloc2是做不到的。

tcmalloc区别的对待大、小对象。它为每个线程分配了一个线程局部的cache，线程需要的小对象都是在其cache中分配的，由于是thread local的，所以基本上是无锁操作（在cache不够，需要增加内存时，会加锁）。同时，tcmalloc维护了进程级别的cache，所有的大对象都在这个cache中分配，由于多个线程的大对象的分配都从这个cache进行，所以必须加锁访问。在实际的程序中，小对象分配的频率要远远高于大对象，通过这种方式（小对象无锁分配，大对象加锁分配）可以提升整体性能。

Tcmalloc组织管理结构:

对于小块内存分配，其内部维护了**60个不同大小的分配器**（实际源码中看到的是86个），和ptmalloc不同的是，它的每个分配器的大小差是不同的，依此按8字节、16字节、32字节等间隔开。在内存分配的时候，会找到最小复合条件的，比如833字节到1024字节的内存分配请求都会分配一个1024大小的内存块。如果这些分配器的剩余内存不够了，会向中央堆申请一些内存，打碎以后填入对应分配器中。同样，如果中央堆也没内存了，就向中央内存分配器申请内存。

在线程缓存内的60个分配器分别维护了一个大小固定的自由空间链表，直接由这些链表分配内存的时候是不加锁的。但是中央堆是所有线程共享的，在由其分配内存的时候会加自旋锁(spin lock)。

在线程内存池每次从中央堆申请内存的时候，分配多少内存也直接影响分配性能。申请地太少会导致频繁访问中央堆，也就会频繁加锁，而申请地太多会导致内存浪费。在tcmalloc里，这个**每次申请的内存量是动态调整的**，调整方式使用了类似把tcp窗口反过来用的**慢启动（slow start）**算法调整max\_length， 每次申请内存是申请max\_length和每个分配器对应的num\_objects\_to\_move中取小的值的个数的内存块。

**num\_objects\_to\_move**取值比较简单，是**以64K为基准**，并且**最小不低于2，最大不高于32的值**。也就是说，对于大于等于32K的分配器这个值为2，对于小于2K的分配器，统一为32。其他的会把数据调整到64K / size 的个数。

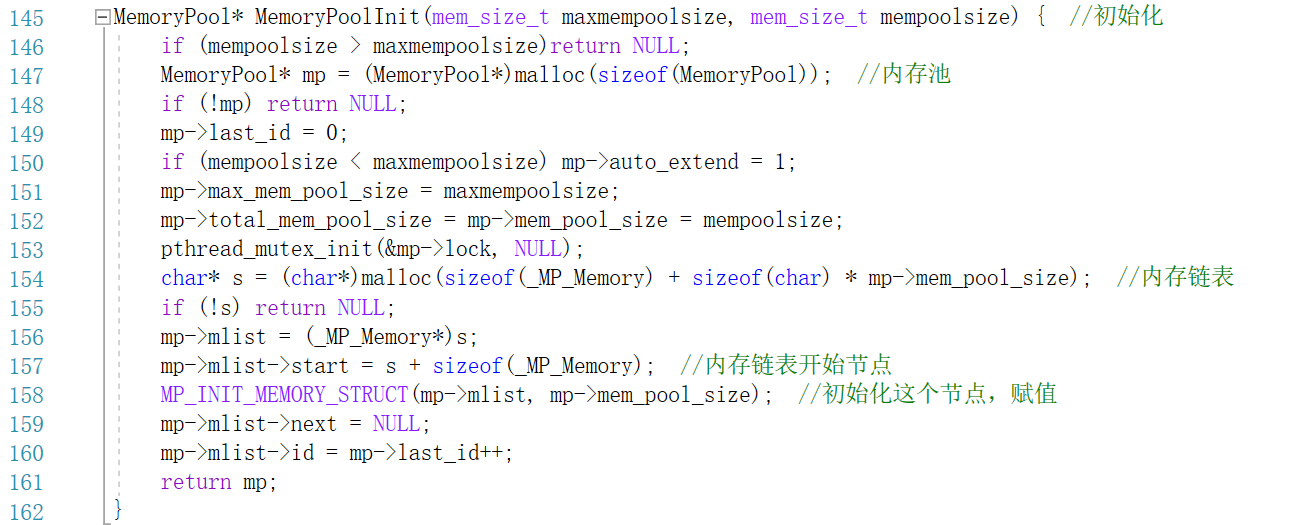
对于max\_length就比较复杂了，而且其更多是用于释放内存。max\_length由1开始，在其小于num\_objects\_to\_move的时候每次累加1，大于等于的时候累加num\_objects\_to\_move。释放内存的时候，首先max\_length会对齐到num\_objects\_to\_move，然后在大于num\_objects\_to\_move的释放次数超过一定阀值，则会按num\_objects\_to\_move缩减大小。

大内存分配：对于大内存分配(大于8个分页, 即32K)，tcmalloc直接在中央堆里分配。**中央堆的内存管理是以分页为单位的**，同样按大小维护了256个空闲空间链表，前255个分别是1个分页、2个分页到255个分页的空闲空间，最后一个是更多分页的小的空间。这里的空间如果不够用，就会直接从系统申请了。

题目二：根据上面对Nginx中内存池的描述，实现与此描述类似的内存池（可以参考Nginx相关源代码）。注意此内存池要支持内存对齐和多线程。

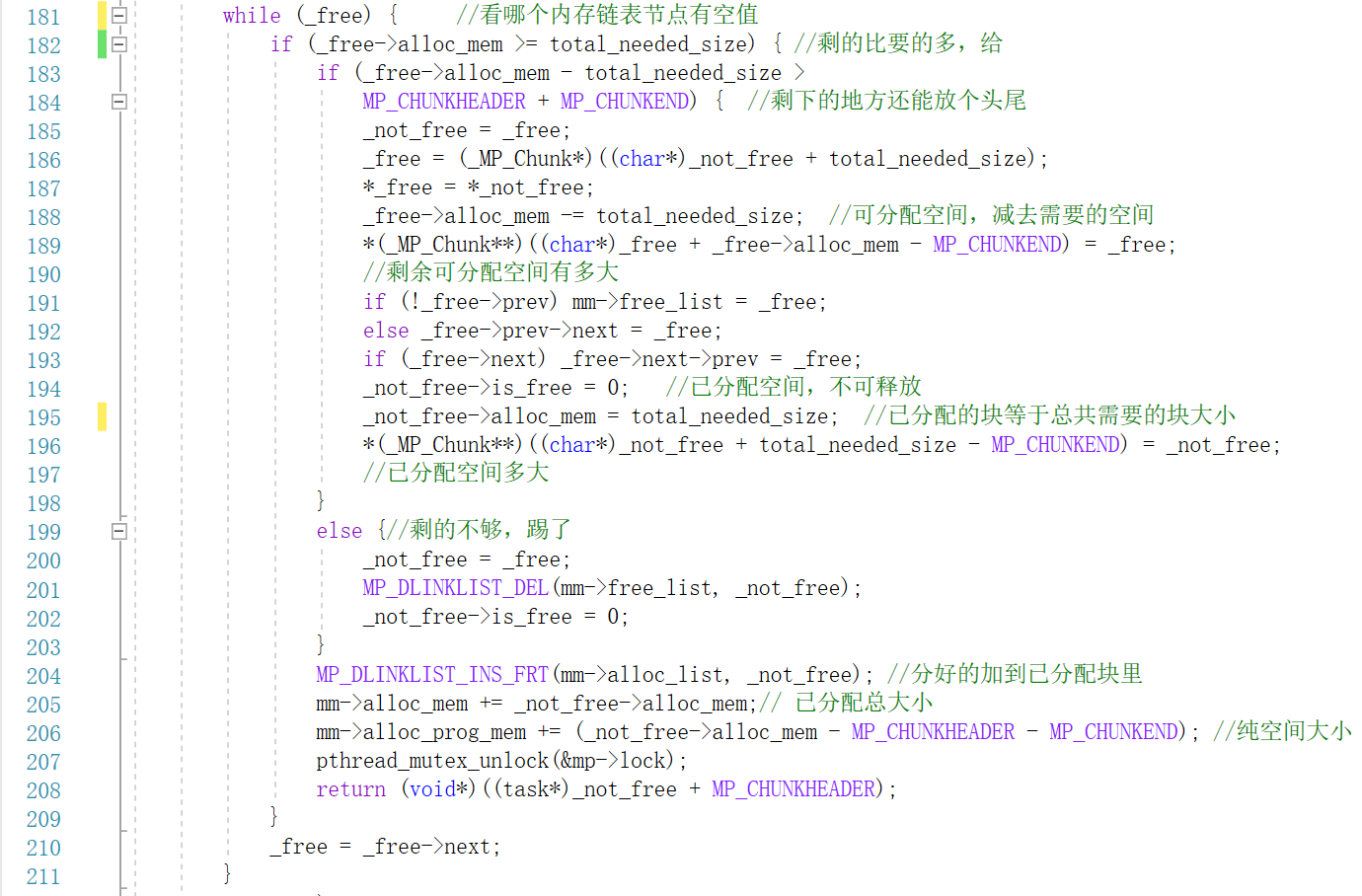


以上是结构体描述，主要分为三个，内存块，内存链表和内存池，分别对应nginx内存池的三个结构。

这是内存池初始化的函数，147行创建内存池，之后规定相关参数的大小，参数在内存池结构体里。



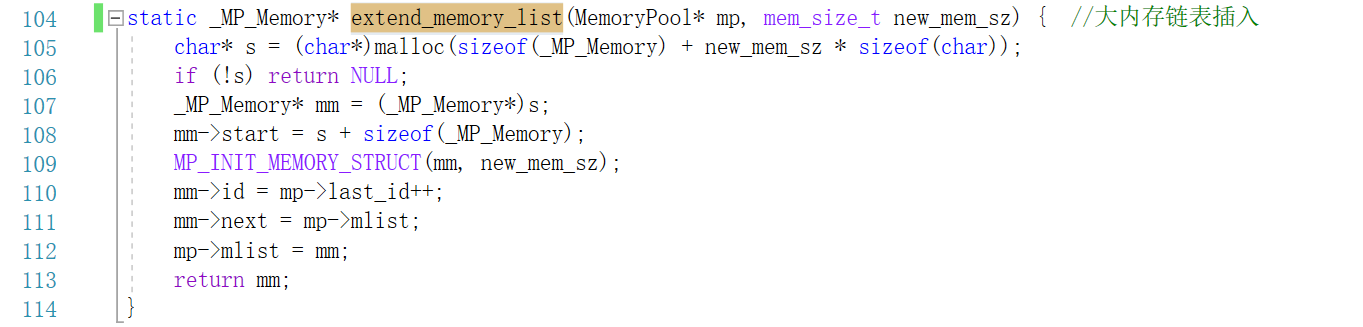
这是实现分配内存的方法，传入初始化的内存池和需要大小，先在小内存链表里找一个节点，这个节点的可分配空间（free 175行）要大于total\_needed\_size，则可操作，将该内存结点的free区域头指针给\_free。

然后进入该节点的free块，如果该区域大小（free->alloc\_mem）大于需要的大小，且能放下块的头大小，则用其来分配内存，具体分配的方法是将\_free的块变成\_not\_free（在187行），意思是这个块变成了属于used里面的。191-195行是把原来的\_free块插入used区域里，通过双向链表的插法。这是在\_free有足够空间下的内存节点操作。

之后把分配的插入已分配块里，并增加总大小，就可以返回分配的空间了。

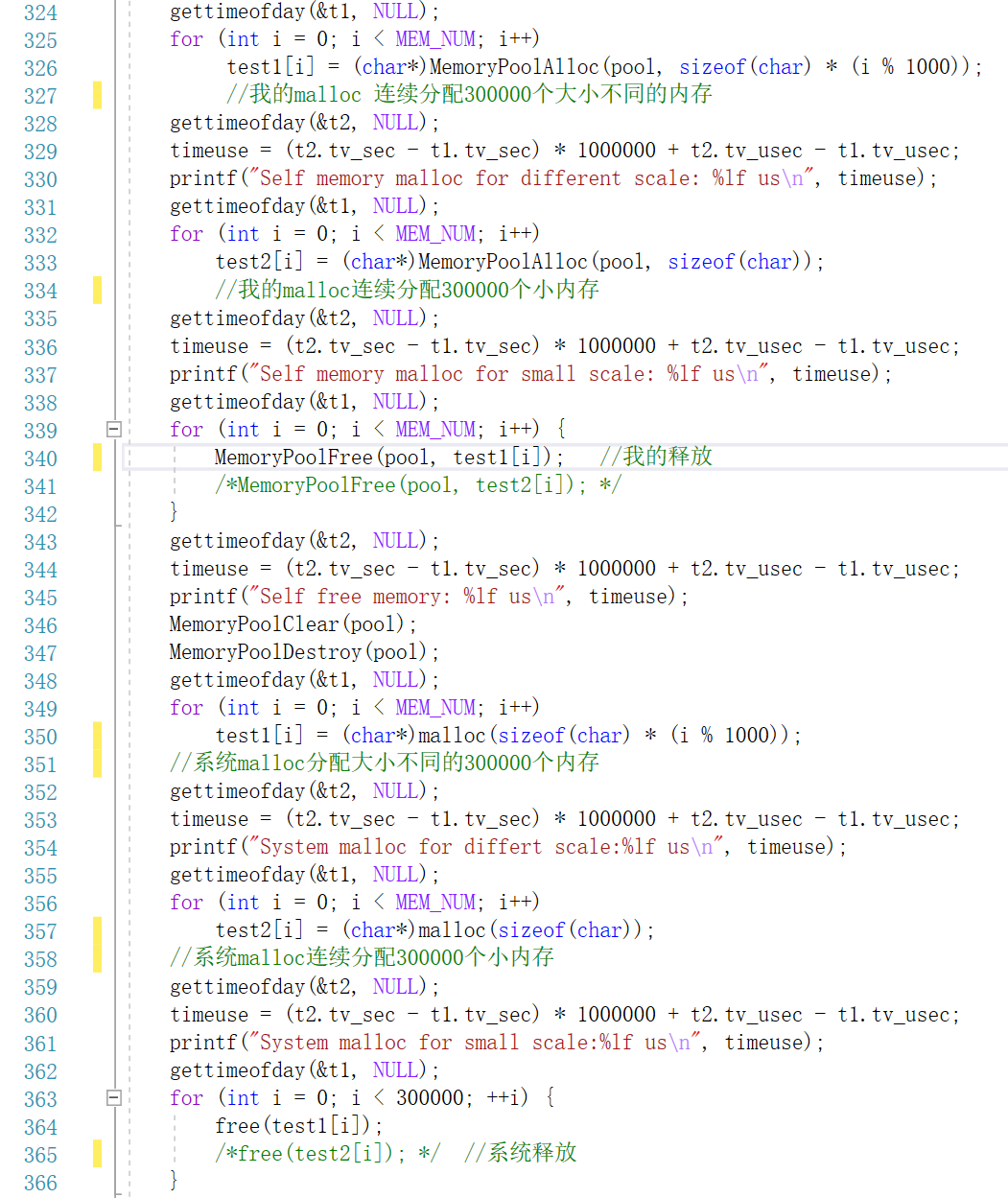


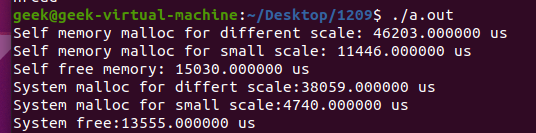
上图是大内存分配的方法，思路是先计算附加的大小，然后调用extend\_memory\_list()生成ngx\_pool\_large\_t结构，并加入到大内存链表中。并增加内存池大小，然后再次给到上面的while分配。



**题目3.** 设计测试代码，在不同内存申请和释放情况下，对比malloc/free和内存池的内存申请和释放效率。

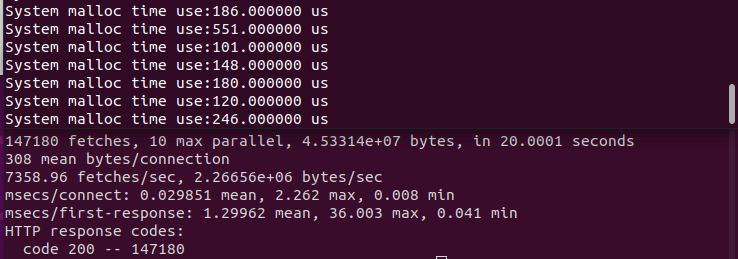
测试代码如下:

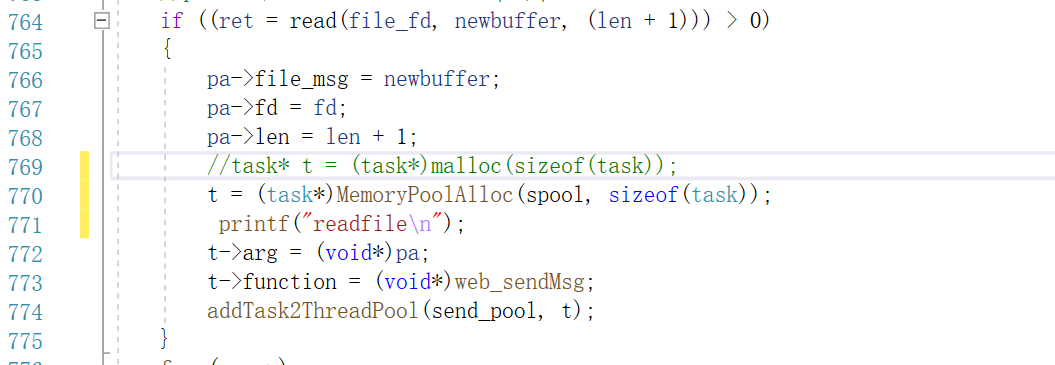




由上图可看出，在分配不同大小的内存时，我的malloc方法时间比系统malloc慢了8000多us，相对慢了18%；但分配大小相同的内存我的就比系统方法慢了两倍多了。释放内存的时间差不多。但我实际测试结果还发现，我的malloc运行时间很不稳定，我分配不同大小内存时间大约在30000us——70000us之间，相同大小的时间大约在8000us——20000us之间。

**题目4**. 通过指定数量的线程或任务共享一个内存池的方式，来修改前面Web服务器中申请和释放内存的代码。每个任务都从指定的内存池申请内存。当使用一个内存池的任务全部完成后，才释放这个内存池。





（修改如图）

上面两张图分别是我修改第三次课设的业务分割代码的所有生成任务的malloc。计算时间在thread\_do函数里，因为readfile和readmsg函数都从这里调用。上图可见系统malloc的时间。用http\_load测试系统malloc在20秒内完成了147180次成功访问。而我写的memorypoolalloc没能运行成功，系统一旦运行我的malloc就会报错，具体原因未知。但我猜测可能是我写的只是模拟系统实现内存分配的方式，并不是真正动用内存分配，所以本质上我的memorypoolalloc并没有给变量分到内存。

附 源代码：

#ifndef \_Z\_MEMORYPOOL\_H\_

#define \_Z\_MEMORYPOOL\_H\_

#define \_Z\_MEMORYPOOL\_THREAD\_

#ifdef \_Z\_MEMORYPOOL\_THREAD\_

#include <pthread.h>

#endif

#include <stdlib.h>

#include <string.h>

#define mem\_size\_t unsigned long long

#define KB (mem\_size\_t)(1 << 10)

#define MB (mem\_size\_t)(1 << 20)

#define GB (mem\_size\_t)(1 << 30)

typedef struct \_mp\_chunk {

//内存块(已用used和没用的free会在代码里显示出来）

mem\_size\_t alloc\_mem; //每个块大小 块可以在used里也可以在free里

struct \_mp\_chunk\* prev, \* next; //块的前后节点

int is\_free; //是否被用

} \_MP\_Chunk;

typedef struct \_mp\_mem\_pool\_list { //内存链表

char\* start;

unsigned int id;

mem\_size\_t mem\_pool\_size;

mem\_size\_t alloc\_mem; //用的或无用的空间大小

mem\_size\_t alloc\_prog\_mem; //排除头尾节点的块大小

\_MP\_Chunk\* free\_list, \* alloc\_list; //分配起始终止节点

struct \_mp\_mem\_pool\_list\* next;

} \_MP\_Memory;

typedef struct \_mp\_mem\_pool {//内存池

unsigned int last\_id;

int auto\_extend;

mem\_size\_t mem\_pool\_size, total\_mem\_pool\_size, max\_mem\_pool\_size;

//内存池大小 总内存池大小 最大大小

struct \_mp\_mem\_pool\_list\* mlist;

#ifdef \_Z\_MEMORYPOOL\_THREAD\_

pthread\_mutex\_t lock;

#endif

} MemoryPool;

/\*

\* 内存池API

\*/

MemoryPool\* MemoryPoolInit(mem\_size\_t maxmempoolsize, mem\_size\_t mempoolsize); //初始化

void\* MemoryPoolAlloc(MemoryPool\* mp, mem\_size\_t want\_size); //分配

int MemoryPoolFree(MemoryPool\* mp, void\* p);

MemoryPool\* MemoryPoolClear(MemoryPool\* mp);

int MemoryPoolDestroy(MemoryPool\* mp);

int MemoryPoolSetThreadSafe(MemoryPool\* mp, int thread\_safe);

/\*

\* 内存池信息API

\*/

mem\_size\_t GetTotalMemory(MemoryPool\* mp);

mem\_size\_t GetUsedMemory(MemoryPool\* mp);

float MemoryPoolGetUsage(MemoryPool\* mp);

mem\_size\_t GetProgMemory(MemoryPool\* mp);

float MemoryPoolGetProgUsage(MemoryPool\* mp);

#endif // !\_Z\_MEMORYPOOL\_H\_

#define MP\_CHUNKHEADER sizeof(struct \_mp\_chunk)

#define MP\_CHUNKEND sizeof(struct \_mp\_chunk\*)

#define MP\_ALIGN\_SIZE(\_n) (\_n + sizeof(long) - ((sizeof(long) - 1) & \_n))

#define MP\_INIT\_MEMORY\_STRUCT(mm, mem\_sz) \ //初始化内存链表

do { \

mm->mem\_pool\_size = mem\_sz; \

mm->alloc\_mem = 0; \

mm->alloc\_prog\_mem = 0; \

mm->free\_list = (\_MP\_Chunk\*) mm->start; \

mm->free\_list->is\_free = 1; \

mm->free\_list->alloc\_mem = mem\_sz; \

mm->free\_list->prev = NULL; \

mm->free\_list->next = NULL; \

mm->alloc\_list = NULL; \

} while (0)

#define MP\_DLINKLIST\_INS\_FRT(head, x) \ //内存链表头插

do { \

x->prev = NULL; \

x->next = head; \

if (head) head->prev = x; \

head = x; \

} while (0)

#define MP\_DLINKLIST\_DEL(head, x) \ //删除x

do { \

if (!x->prev) { \

head = x->next; \

if (x->next) x->next->prev = NULL; \

} else { \

x->prev->next = x->next; \

if (x->next) x->next->prev = x->prev; \

} \

} while (0)

static \_MP\_Memory\* extend\_memory\_list(MemoryPool\* mp, mem\_size\_t new\_mem\_sz) { //大内存链表插入

char\* s = (char\*)malloc(sizeof(\_MP\_Memory) + new\_mem\_sz \* sizeof(char));

if (!s) return NULL;

\_MP\_Memory\* mm = (\_MP\_Memory\*)s;

mm->start = s + sizeof(\_MP\_Memory);

MP\_INIT\_MEMORY\_STRUCT(mm, new\_mem\_sz);

mm->id = mp->last\_id++;

mm->next = mp->mlist;

mp->mlist = mm;

return mm;

}

static \_MP\_Memory\* find\_memory\_list(MemoryPool\* mp, void\* p) {

\_MP\_Memory\* tmp = mp->mlist;

while (tmp) {

if (tmp->start <= (char\*)p &&

tmp->start + mp->mem\_pool\_size > (char\*)p)

break;

tmp = tmp->next;

}

return tmp;

}

static int merge\_free\_chunk(MemoryPool\* mp, \_MP\_Memory\* mm, \_MP\_Chunk\* c) { //合并未分配的块

\_MP\_Chunk\* p0 = c, \* p1 = c;

while (p0->is\_free) {

p1 = p0;

if ((char\*)p0 - MP\_CHUNKEND - MP\_CHUNKHEADER <= mm->start) break;

p0 = \*(\_MP\_Chunk\*\*)((char\*)p0 - MP\_CHUNKEND);

}

p0 = (\_MP\_Chunk\*)((char\*)p1 + p1->alloc\_mem);

while ((char\*)p0 < mm->start + mp->mem\_pool\_size && p0->is\_free) { //p0块没有分配，且p0在used某特定位置前面

MP\_DLINKLIST\_DEL(mm->free\_list, p0);

p1->alloc\_mem += p0->alloc\_mem;

p0 = (\_MP\_Chunk\*)((char\*)p0 + p0->alloc\_mem);

}

\*(\_MP\_Chunk\*\*)((char\*)p1 + p1->alloc\_mem - MP\_CHUNKEND) = p1;

pthread\_mutex\_unlock(&mp->lock);

return 0;

}

MemoryPool\* MemoryPoolInit(mem\_size\_t maxmempoolsize, mem\_size\_t mempoolsize) { //初始化

if (mempoolsize > maxmempoolsize)return NULL;

MemoryPool\* mp = (MemoryPool\*)malloc(sizeof(MemoryPool)); //内存池

if (!mp) return NULL;

mp->last\_id = 0;

if (mempoolsize < maxmempoolsize) mp->auto\_extend = 1;

mp->max\_mem\_pool\_size = maxmempoolsize;

mp->total\_mem\_pool\_size = mp->mem\_pool\_size = mempoolsize;

pthread\_mutex\_init(&mp->lock, NULL);

char\* s = (char\*)malloc(sizeof(\_MP\_Memory) + sizeof(char) \* mp->mem\_pool\_size); //内存链表

if (!s) return NULL;

mp->mlist = (\_MP\_Memory\*)s;

mp->mlist->start = s + sizeof(\_MP\_Memory); //内存链表开始节点

MP\_INIT\_MEMORY\_STRUCT(mp->mlist, mp->mem\_pool\_size); //初始化这个节点，赋值

mp->mlist->next = NULL;

mp->mlist->id = mp->last\_id++;

return mp;

}

void\* MemoryPoolAlloc(MemoryPool\* mp, mem\_size\_t wantsize) { //nginx实现内存池

if (wantsize <= 0) return NULL;

mem\_size\_t total\_needed\_size = MP\_ALIGN\_SIZE(wantsize + MP\_CHUNKHEADER + MP\_CHUNKEND);

//总共需要的，是分配大小，加上块的头尾

if (total\_needed\_size > mp->mem\_pool\_size) return NULL;

\_MP\_Memory\* mm = NULL, \* mm1 = NULL;

\_MP\_Chunk\* \_free = NULL, \* \_not\_free = NULL;

pthread\_mutex\_lock(&mp->lock);

FIND\_FREE\_CHUNK:

mm = mp->mlist;

while (mm) { //找到有空间的内存链表节点

if (mp->mem\_pool\_size - mm->alloc\_mem < total\_needed\_size) {

//剩余空间小于需要空间，不给，下一个

mm = mm->next;

continue;

}

\_free = mm->free\_list; //未分配块

while (\_free) { //看哪个内存链表节点有空值

if (\_free->alloc\_mem >= total\_needed\_size) { //剩的比要的多，给

if (\_free->alloc\_mem - total\_needed\_size >

MP\_CHUNKHEADER + MP\_CHUNKEND) { //剩下的地方还能放个头尾

\_not\_free = \_free;

\_free = (\_MP\_Chunk\*)((char\*)\_not\_free + total\_needed\_size);

\*\_free = \*\_not\_free;

\_free->alloc\_mem -= total\_needed\_size; //可分配空间，减去需要的空间

\*(\_MP\_Chunk\*\*)((char\*)\_free + \_free->alloc\_mem - MP\_CHUNKEND) = \_free;

//剩余可分配空间有多大

if (!\_free->prev) mm->free\_list = \_free;

else \_free->prev->next = \_free;

if (\_free->next) \_free->next->prev = \_free;

\_not\_free->is\_free = 0; //已分配空间，不可释放

\_not\_free->alloc\_mem = total\_needed\_size; //已分配的块等于总共需要的块大小

\*(\_MP\_Chunk\*\*)((char\*)\_not\_free + total\_needed\_size - MP\_CHUNKEND) = \_not\_free;

//已分配空间多大

}

else {//剩的不够，踢了

\_not\_free = \_free;

MP\_DLINKLIST\_DEL(mm->free\_list, \_not\_free);

\_not\_free->is\_free = 0;

}

MP\_DLINKLIST\_INS\_FRT(mm->alloc\_list, \_not\_free); //分好的加到已分配块里

mm->alloc\_mem += \_not\_free->alloc\_mem;// 已分配总大小

mm->alloc\_prog\_mem += (\_not\_free->alloc\_mem - MP\_CHUNKHEADER - MP\_CHUNKEND); //纯空间大小

pthread\_mutex\_unlock(&mp->lock);

return (void\*)((task\*)\_not\_free + MP\_CHUNKHEADER);

}

\_free = \_free->next;

}

mm = mm->next;

}

//下面是分大内存区域的

if (mp->auto\_extend) {

if (mp->total\_mem\_pool\_size >= mp->max\_mem\_pool\_size) { //总内存池大小大于上限

pthread\_mutex\_unlock(&mp->lock);

return NULL;

}

mem\_size\_t add\_mem\_sz = mp->max\_mem\_pool\_size - mp->mem\_pool\_size; //附加大小

add\_mem\_sz = add\_mem\_sz >= mp->mem\_pool\_size ? mp->mem\_pool\_size : add\_mem\_sz;

mm1 = extend\_memory\_list(mp, add\_mem\_sz); //大内存节点

if (!mm1) {

pthread\_mutex\_unlock(&mp->lock);

return NULL;

}

mp->total\_mem\_pool\_size += add\_mem\_sz; //内存池大小增加

goto FIND\_FREE\_CHUNK;

}

pthread\_mutex\_unlock(&mp->lock);

return NULL;

}

int MemoryPoolFree(MemoryPool\* mp, void\* p) {

if (p == NULL || mp == NULL) return 1;

pthread\_mutex\_lock(&mp->lock);

\_MP\_Memory\* mm = mp->mlist;

if (mp->auto\_extend) mm = find\_memory\_list(mp, p);

\_MP\_Chunk\* ck = (\_MP\_Chunk\*)((char\*)p - MP\_CHUNKHEADER);

MP\_DLINKLIST\_DEL(mm->alloc\_list, ck); //已分配的，释放

MP\_DLINKLIST\_INS\_FRT(mm->free\_list, ck); //未分配的，头插

ck->is\_free = 1;

mm->alloc\_mem -= ck->alloc\_mem;

mm->alloc\_prog\_mem -= (ck->alloc\_mem - MP\_CHUNKHEADER - MP\_CHUNKEND);

return merge\_free\_chunk(mp, mm, ck);

}

MemoryPool\* MemoryPoolClear(MemoryPool\* mp) {

if (!mp) return NULL;

pthread\_mutex\_lock(&mp->lock);

\_MP\_Memory\* mm = mp->mlist;

while (mm) {

MP\_INIT\_MEMORY\_STRUCT(mm, mm->mem\_pool\_size);

mm = mm->next;

}

pthread\_mutex\_unlock(&mp->lock);

return mp;

}

int MemoryPoolDestroy(MemoryPool\* mp) {

if (mp == NULL) return 1;

pthread\_mutex\_lock(&mp->lock);

\_MP\_Memory\* mm = mp->mlist, \* mm1 = NULL;

while (mm) {

mm1 = mm;

mm = mm->next;

free(mm1);

}

pthread\_mutex\_unlock(&mp->lock);

pthread\_mutex\_destroy(&mp->lock);

free(mp);

return 0;

}

mem\_size\_t GetTotalMemory(MemoryPool\* mp) {

return mp->total\_mem\_pool\_size;

}

mem\_size\_t GetUsedMemory(MemoryPool\* mp) {

pthread\_mutex\_lock(&mp->lock);

mem\_size\_t total\_alloc = 0;

\_MP\_Memory\* mm = mp->mlist;

while (mm) {

total\_alloc += mm->alloc\_mem;

mm = mm->next;

}

pthread\_mutex\_unlock(&mp->lock);

return total\_alloc;

}

mem\_size\_t GetProgMemory(MemoryPool\* mp) {

pthread\_mutex\_lock(&mp->lock);

mem\_size\_t total\_alloc\_prog = 0;

\_MP\_Memory\* mm = mp->mlist;

while (mm) {

total\_alloc\_prog += mm->alloc\_prog\_mem;

mm = mm->next;

}

pthread\_mutex\_unlock(&mp->lock);

return total\_alloc\_prog;

}

float MemoryPoolGetUsage(MemoryPool\* mp) {

return (float)GetUsedMemory(mp) / GetTotalMemory(mp);

}

float MemoryPoolGetProgUsage(MemoryPool\* mp) {

return (float)GetProgMemory(mp) / GetTotalMemory(mp);

}

#undef MP\_CHUNKHEADER

#undef MP\_CHUNKEND

#undef MP\_ALIGN\_SIZE

#undef MP\_INIT\_MEMORY\_STRUCT

#undef MP\_DLINKLIST\_INS\_FRT

#undef MP\_DLINKLIST\_DEL

#define MEM\_NUM 300000

int main() {

char\* test1[MEM\_NUM],test2[MEM\_NUM];

MemoryPool\* pool = MemoryPoolInit(500 \* MB, 100 \* MB);

struct timeval t1, t2;

double timeuse;

gettimeofday(&t1, NULL);

for (int i = 0; i < MEM\_NUM; i++)

test1[i] = (char\*)MemoryPoolAlloc(pool, sizeof(char) \* (i % 1000));

//我的malloc 连续分配300000个大小不同的内存

gettimeofday(&t2, NULL);

timeuse = (t2.tv\_sec - t1.tv\_sec) \* 1000000 + t2.tv\_usec - t1.tv\_usec;

printf("Self memory malloc for different scale: %lf us\n", timeuse);

gettimeofday(&t1, NULL);

for (int i = 0; i < MEM\_NUM; i++)

test2[i] = (char\*)MemoryPoolAlloc(pool, sizeof(char));

//我的malloc连续分配300000个小内存

gettimeofday(&t2, NULL);

timeuse = (t2.tv\_sec - t1.tv\_sec) \* 1000000 + t2.tv\_usec - t1.tv\_usec;

printf("Self memory malloc for small scale: %lf us\n", timeuse);

gettimeofday(&t1, NULL);

for (int i = 0; i < MEM\_NUM; i++) {

MemoryPoolFree(pool, test1[i]); //我的释放

/\*MemoryPoolFree(pool, test2[i]); \*/

}

gettimeofday(&t2, NULL);

timeuse = (t2.tv\_sec - t1.tv\_sec) \* 1000000 + t2.tv\_usec - t1.tv\_usec;

printf("Self free memory: %lf us\n", timeuse);

MemoryPoolClear(pool);

MemoryPoolDestroy(pool);

gettimeofday(&t1, NULL);

for (int i = 0; i < MEM\_NUM; i++)

test1[i] = (char\*)malloc(sizeof(char) \* (i % 1000));

//系统malloc分配大小不同的300000个内存

gettimeofday(&t2, NULL);

timeuse = (t2.tv\_sec - t1.tv\_sec) \* 1000000 + t2.tv\_usec - t1.tv\_usec;

printf("System malloc for differt scale:%lf us\n", timeuse);

gettimeofday(&t1, NULL);

for (int i = 0; i < MEM\_NUM; i++)

test2[i] = (char\*)malloc(sizeof(char));

//系统malloc连续分配300000个小内存

gettimeofday(&t2, NULL);

timeuse = (t2.tv\_sec - t1.tv\_sec) \* 1000000 + t2.tv\_usec - t1.tv\_usec;

printf("System malloc for small scale:%lf us\n", timeuse);

gettimeofday(&t1, NULL);

for (int i = 0; i < 300000; ++i) {

free(test1[i]);

/\*free(test2[i]); \*/ //系统释放

}

gettimeofday(&t2, NULL);

timeuse = (t2.tv\_sec - t1.tv\_sec) \* 1000000 + t2.tv\_usec - t1.tv\_usec;

printf("System free:%lf us\n", timeuse);

return 0;

}