# 为什么需要内存池

课前预习资料：链接：https://pan.baidu.com/s/1fYjRjcQZGtqT6f838EUXdQ

提取码：217m

martin老师讲的，高性能服务器内存剖析。

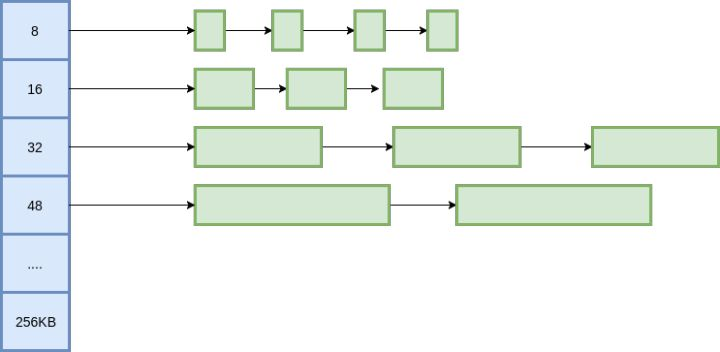
1）内存分配的性能

brk --> glibc --> tcmalloc / jemalloc --> 内存池（用户态）

2）工程的角度，可以防止程序因为内存错误（不当使用）而造成的崩溃或者其它各种各样的问题。

# 怎么设计内存池？

1. glibc一次性问内核申请一大块内存，然后划分成很多的小块，这样用户需要使用内存的时候，就直接找到合适的内存给用户，如果释放内存的时候，也是直接由glibc去回收，这样就避免频繁得调用系统调用问内核要内存。
2. tcmalloc，它按照线程划分内存块，解决多线程之间对内存申请和释放的竞争问题，提高了效率。



1. tamalloc为了要满足各种尺寸申请和释放，所以算法复杂，所以这里就有了内存池技术的必要性。

OK，内存池的设计要考虑哪些方面：

1. 尺寸
2. 线程安全
3. 内存空间的大小是否可以调整

## ringbuffer的设计：

1. 类似于内核的kfifo的设计，

struct ringbuffer {

void \*data; // 理所当然的是内存区

unsigned int size; // 内存区的尺寸

unsigned int read\_pos; // 从内存区域开始读的位置

unsigned int write\_pos; // 往内存区域开始写的位置

};

ssize\_t ringbuffer\_from\_dev(int fd, struct ringbuffer \*ring\_buf, unsigned int len)

char buffer[1024];

epoll\_wait(…)

for (;;)

{

read(fd, buffer, 8);

ringbuffer\_put(ringbuffer, buffer, 8); // 内存copy，

ringbuffer\_from\_dev（fd, ringbuffer, 8）; // zero copy ==》 用户态协议栈

}

UDP分片：

使用的场景：

单生产者单消费者的情况下使用。lock-free。UDP组包，分包，文件传输、网络消息

接收文件：一个线程读取网络上的文件内容，一个线程保存网络文件内容到文件

## multiple-ringbuffer

typedef struct RingBuffer16\_

{

unsigned short write;

unsigned short read;

spinlock\_t spin; // 自旋锁

void \*array[RING\_BUFFER\_16\_SIZE]; // 指针数组，65536个节点，每个节点指向一块内存区域，

} RingBuffer16;

typedef volatile int spinlock\_t; // volatile修饰了spinlock\_t这个变量，CPU频繁访问的变量，他也许会保存到CPU的缓存里，volatile的变量就一定需要到内存里去读。

#define INIT\_SPINLOCK(lock) lock = 0

#define spinlock\_lock(lock) \

do { \

while (!\_\_sync\_bool\_compare\_and\_swap(lock, 0, 1)) \

sched\_yield(); \

} while(0)

#define spinlock\_unlock(lock) \

do { \

\*lock = 0; \

} while(0)

* 在任务密集下的情况，使用自旋能提高效率。

M: multiple, Mr 多个读线程， Mw 多个写线程。

S: single, Sr 单个读线程 Sw 单个写线程。

* MrSw Get

前置条件：要有内容可读，

while (rb->write == rb->read) {

RingBufferWait(rb);

}

循环结束就是有内容可读了。

如何解决多个线程去读的情况，用原子

原子操作：cas(rb->read, readp), 如果read还等于readp,那说明没有人移动过read（没有人读过内容）， rb->read = rb->read + 1，否则就是有人读过内容，cas会返回失败。

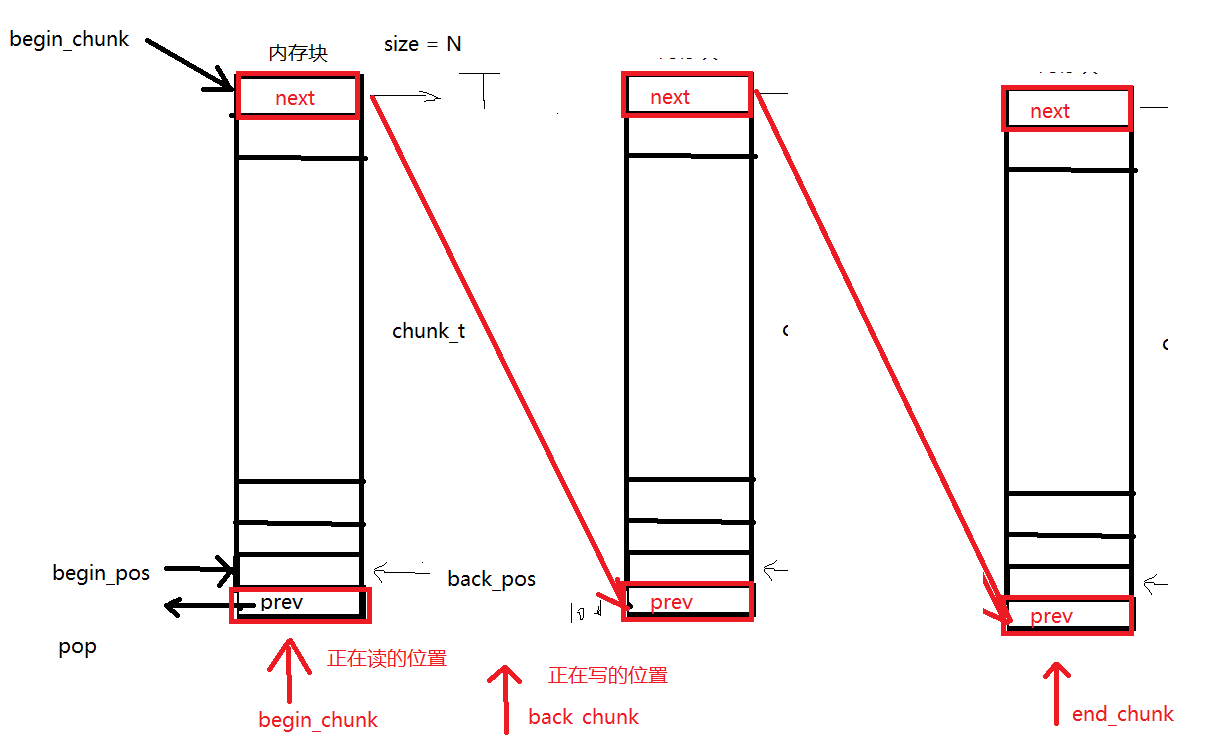
1. 返回readpos指向的指针: readp = rb->read
2. readpos++ //readpos 移动一格, cas(rb->read, readp)
3. 判断预置条件是否满足，满足继续step1, 否则就是等

作业：

1. 文件传输：支持各种各样的文件，传输要完整，协议不限（TCP/UDP/HTTP），你要加上完整性校验，断点传输，迅雷一样，区块链的方式去传输，

考察一个点：大家所有的传输的性能比拼。

## ZeroMQ的设计



inline T \*cas (T \*cmp\_, T \*val\_)

{

#if defined ZMQ\_ATOMIC\_PTR\_CXX11

ptr.compare\_exchange\_strong (cmp\_, val\_, std::memory\_order\_acq\_rel);

return cmp\_;

#else

return (T \*) atomic\_cas ((void \*\*) &ptr, cmp\_, val\_

#if defined ZMQ\_ATOMIC\_PTR\_MUTEX

#endif

}

ptr ： old\_value

comp\_\_: 一个待比较的值

val：要设置的值

cas: if (ptr == cmp\_\_)

{

ptr = val;

return ptr;

}

else

{

return ptr;

}

问题：这样的chunk，要初始化多少个呢？

1. spare\_chunk如果为NULL：
2. 分配一个新的chunk；
3. 把新的chunk加入到chunk链表里；

预置条件：就是在调用pop的时候，如果begin\_chunk所有的内存已经读完，那么就把spare\_chunk 指向了end\_chunk

1. 把end\_chunk指向spare\_chunk指向的内存。

spare\_chunk永远只保存最近一次待释放的内存块，

总结：

1. 把即将要释放的内存保存起来，以便下次重复利用，达到了内存高效实用；
2. 如果不需要那么多的内存（消费者的速率比生产者的速率要快），就是缩减内存空间；
3. 如果需要更多的内存块（生产者的速率比消费者的速率要快），就分配更多的内存块。

动态调整内存空间：

ZMQ可以把多条消息组包成一个大的消息去发送 ： 多次写，一次性预取的机制。保证把很多小的内存组成一个大包去发送。 把本地的socket的内存尽量去填满。最大利用了内核的空间。提高了网络发送的效率：

ZMQ 🡪 分布式集群下的开发：高并发

内存池的总结：

1. 集中化的存储了消息：Cmsg \* m = new CMSG;

对于网络开发来讲，能够提高应用处理网络IO的效率，缓解网络流量的洪峰。

1. 节省了很多malloc、new和free、delete这些频繁的对内存操作说造成的消耗。

ringbuffer zmq boost::pool nginx : ==》 多线程的支撑。