**概述**

在整个编译期都需要涉及到大量数据结构的分配和回收，这需要定制一个较为优秀的内部内存管理机制，而不是使用默认的malloc。

这里我使用了一个宏定义DEFAULT\_MEM来决定使用哪个alloc机制。

另外，我必须指出的是，如果你想更好的进行debug，强烈推荐开启这个宏定义，使用系统的alloc机制，毕竟，如果访问了不该访问的内存，会有一个中断的，但是，如果使用了内存池，那么很可能导致不正确的内存分配和释放问题，那就很要命了，因为根本不会有中断发生。当然，如果一切没问题，还是用我设计的内存池吧（或者你可以自己做一个。。。我没意见，接口都给了）

至于接下来的设计，其实也是数易其稿，毕竟，数量规模发生变动之后，一切都需要重新设计。

* **结论：自己设计的内存池，还是挂了——反正因为内存对齐算法太复杂，我放弃了。**

**改天整个编译器过了测试，再考虑吧，真的改的我自己都烦了。最主要这个内存对齐太麻烦了。**

* **内存池数据结构设计**

主要包括三层数据结构设计。

最底下一层item，用于管理单块内存——该内存块大小都是2的倍数，如果需要分配不是2的倍数的内存，就向上取整——这会造成一定的浪费，但是并不夸张。

第二层，是一个block，每个block由若干个内存块大小相同的item聚合而成，block之间使用环装链表进行链接——请注意，尽管我们定义了vector，但是可惜的是vector的底层内存分配是用的m\_alloc函数，这导致了我们无法使用vector，只能另做一个使用系统malloc的链表了。当然，更不可能让vector这样的东西去做成用系统malloc作为底层的，那简直舍本逐末。

在每个block链内部，同样维护着一个空闲块和使用块的链表

第三层，对于不同内存块大小的block链进行汇总。

* **内存池分配和释放算法**

当给出某个需要分配的大小n的时候，向上取整得到内存块的大小。随后找到对应的block链，遍历整个链的每个block，查看其空闲块链表，看是否存在空闲块，如果有，那么摘下这个块，并使用，如果每个块都没有，那么向系统申请一个block，初始化之后，从头结点摘下一个块，并返回一个块，如果分配的内存太大，则直接向系统申请就好了。

当free的时候，和分配类似，但是无需遍历，只需要释放，插入到空闲链表尾部，并返回即可。

对于向系统申请的块，直接free。

——但是如何分辨是系统分配的呢？我们知道每个block的头指针，以及block对应的链，还有每个block的大小，所以一个办法就是，遍历block链的所有block，直到找到对应的block并修改。这看上去并不优雅，并且，当整个空间非常大的时候，这种遍历意味着更高的成本。另一个方法是，记录下每次直接通过malloc向系统申请的指针。

这两种方法所对应的内存申请情况是不同的，前者对于小内存的大量分配非常不友好，但是大量大块内存是可以接受的，因为小内存大量分配意味着需要遍历的block非常多，这会产生非常多的block，遍历的成本非常高，后者则相反，对于大量大块内存的分配不友好，因为需要大量的记录并且遍历——当然使用哈希等等方法其实可以减轻压力，但是哈希表也存在一个问题，那就是查找链条的中断问题，显然，内存池在free之后不应当保留该块属于系统内存的指针，但是如果置空，那么哈希表的查找链条又会中断，总之，最后我还是选择了单链表。

考虑到实际情况，我选择后一种，我做了一个简单的单链表，向系统申请的内存插入到单链表的头部，free的时候遍历链表，如果在链表中就是向系统申请的，摘下那个节点就好了。

（事实上，只考虑free的话，尾部应该更好一些，毕竟通常来说先分配的一般先释放，但是，当同时考虑alloc和free，应该放在头部，因为如果放在尾部，那么alloc的时候必然遍历一遍，而free优先释放先分配的节点的概率没有那么高，相反，放在头部，alloc必然只有O（1），而free也未必会遍历一整遍，所以头部需要遍历的总体次数应该更小（抠抠搜搜。。。））

以上设计的前提，我想都是存在大量的小内存需要分配，而真正向系统申请的次数并不多的前提条件，在我实际设计当中，也进行了测试，需要向系统分配的次数相比于整个分配的次数，应当来说，非常的低。

（具体我就不多说了，网上介绍的一堆，方法都差不多）

另外，对于内存溢出导致的错误如何处理。Emmm，没办法。诚然，因为向上取整了，所以肯定会在尾部有一定的内存富裕，但是，这不顶事儿

当然，在这个项目中，有一定办法可以尽量阻止这样做的风险，首先是分配多少用多少，而且，哪里分配就哪里使用，这在任何地方都是尽量好的方法，其次，尽量使用封装好的比如vector等等数据结构，而不是自己alloc，这可以很大程度上解决问题

——至于封装好的数据结构出问题，额，会测试的啦，真的要是碰上了，自认倒霉，慢慢修bug吧，我还真的遇到过一次，哈希表（vector不需要memset的）向系统申请内存之后没有用memset导致的错误，修了多久，我就不说了。

* **编译器的内存释放问题**

1、对于访问空指针的问题，我选择了一个办法，释放之后置空，我相信大家都是这么做的。

2、在我做内存释放的时候，遇到了大量的重复释放指针的问题。使用自己定义的分配机制当然能够一定程度上解决问题。但是造成这种情况的根源在于数据结构的复杂性导致多个数据结构都指向同一个变量。

使用自己做的内存池可以解决这个问题，不过，重复释放指针在某种程度上面意味着错误，使用内存池不等于杜绝这个逻辑上的错误。

我尽量使用默认molloc和free的时候过一遍，使得它不会出现严重的内核报错，但是这不代表这里面没有bug。

* **内存对齐问题**

（两个月后回来考古）

其实在最开始的设计中，并没有考虑这个问题，直到我开始在语义分析里面设计结构体的对齐问题。对于malloc而言，他分配给用户的东西，一定会是内存对齐的——当然，冗余肯定是免不了的。

对于内存池而言，分配给用户的内存可以不对齐，但是这意味着可能存在的对象的多次的内存访问。而这从某种角度而言是会降低效率的。

既然是考古，那先看一下上面的设计对内存对齐上面的问题吧。（太惨烈了，得重新设计了）。

首先是，我们要做什么，要做的就是m\_alloc返回的地址指针，是按照内存对齐大小要求来的，比如说要求对齐模数是8，那么指针要被8整除。然而按照原来版本的内存池进行设计的话，在代码生成阶段，如果使用m\_alloc去分配，会导致其首地址指针不对齐，就拿结构体来说，在其offset为0的时候，第一个指针就不是对齐的，所以访问每个成员都需要两次访问，这执行效率怕不是要笑死人（因为有常量表达式，所以有会在编译阶段直接计算的情况）。

办法呢，就是在MEM模块的时候就保证其返回的地址是对齐的。

考虑到原始版本的内存池，有两个办法，1、改造每个block的结构，使得每个item掐头之后的位置是内存对齐的。2，改造分配和释放算法，我不让每个item的首地址是对齐的，但是在这块内存开头，把开头不对齐的部分敲掉，返回剩下部分的首地址（但是这样会使得返回的可用大小小于该item的实际大小，所以需要改造分配算法）。等到释放的时候，我自然没法直接减掉点地址去寻找header的部分（因为那是被敲掉的部分），所以释放算法也需要改造。

显然，我认为前一种方式更好一些，后者会针对每个block的不同偏移的item单独去计算需要的空间是否满足，这会消耗很多的计算资源。

同时，必须要考虑到需要根据设定的align size自动改变的情况。

所以我觉得吧，仿照ptmalloc的设计其实也不错。

首先，block的data部分是使用系统的malloc来分配的。所以，这是可以保证block的首地址对齐是对齐的。那么gcc的分配器，在不同位数的系统下给默认malloc的分配的对齐系数是多少呢？我查到的资料说，32位系统下默认对齐模数是4字节，64是8字节。（但是我不确定）所以我在def中设计的对齐模数也是如此。

当然以防万一（毕竟使用的系统malloc库不是我写的，我也不能完全确定各种各样其他版本的gcc和其他编译器的情况，说不定有例外呢），因此我设计了一条规则：向系统请求的大小为需求的block data size加上对齐模数（作为冗余量），然后检测是否满足对齐，如果满足，直接用，如果不满足，则data的指针往后移动，直到满足为止（这个偏移是可以计算的，而且不超过一个对齐模数的大小）。

另外，所需求的block data size的大小，也必须按照对齐模数对齐，不够的则向上补足。

这需要加两个哦——冗余肯定可能有存在，但是呢，这点开销对于block一次分配那么多的情况，微不足道吧。

OK，block的好了，再来说接下来的内存池表项item

在header部分，就一个union，大小为原来的header和对齐模数的更大值，当然，原来的header是两个unsigned int，可以保证一定是2的整倍数（按理说加起来应该是8字节啊，各种环境下就不好说）对于对齐模数小于他的，因为我宏定义的时候限制了对齐模数，所以小于的情况，header一定是对齐模数的整倍数，大于的，则按照对齐模数的值来弄，同样也保证了后面data部分的对齐，实际测试了一下，大量的分配和释放次数，还是比较健壮的

* **速度改进方案**

（在上一次出现问题之后，继续来考古）

内存果然不愧是C程序员一生之敌。

大概的问题是这样的，由于每次都遍历block链表，太傻乎乎了，导致当大量内存需要分配的时候，这个性能慢的我想死，这当然有我其他地方算法设计的锅，但是，也不能说一点内存池设计的问题都没有吧。简直就是在刮痧。

这里首先优化掉需要遍历链表的情况。一个是分配的情况，需要遍历所有节点去找，一旦开头大量block的都被占用，那么每次都要不断的遍历开头已经被完全占用的block，这是一种浪费。

另一个是释放的情况。释放给系统的我不说，毕竟编译器的设计本身保证了，向系统直接申请大块内存的次数屈指可数。绝大多数都是小块内存需求。

在这里，仍然需要每次都遍历所有的block去找到到底是在哪块内存分配的，这个性能也一样拉胯。

总而言之，一句话，申请和释放两边都拉胯。

申请的优化是这样的，弄一个free block 链表，每次从头开始，保证该链表里面的有空间可分配。分配完了之后，如果没有空间，就从free链头摘下来。如果释放了之后，导致某个block有空间了，那么再挂上去。

如果这个链空了，就没有新的block可以用了，那么去向系统申请，挂上去，并且还挂到原来的block链上面去

释放的优化这样子。每个item都记住他所在的链号和在block中所在的块号，从而得出block的data的首地址即可。本来呢，直接用柔性数组成员，就可以直接得到目标block头的位置，可是不巧的是，这emmm怎么说呢，为了满足align的要求，这个data实际上存在偏移，这就麻了爪子。所以怎么办呢，在data最前面，再放一个数据，指向block块的位置。之所以放在前面，就是因为，放在后面的话，一旦最后一个溢出，直接gg。

至于完事儿之后的结果，反正我感觉不出来和系统malloc的性能差距，而且，申请的内存都是小内存且数量基本达到我的物理内存上限的情况下都基本感觉不到差距。

应该没啥问题的话，这是最终版本文档了——难不成还要弄多线程的嘛？

当然实际性能会有一定差距，目测实际使用的内存会比sys malloc要多50%左右。我个人推测，这应当是由于我在内部建立的数据结构，以及冗余的内存申请所导致的，尤其是其中有大量的小内存的申请。使得内存块的头的空间占比非常高，这应当是合理的。