**DLmalloc**

1. 边界标识法：

1. Small chunks:

struct malloc\_chunk {

size\_t prev\_foot; /\* Size of previous chunk (if free). \*/

size\_t head; /\* Size and inuse bits. \*/

struct malloc\_chunk\* fd; /\* double links -- used only if free. \*/

struct malloc\_chunk\* bk;

};

malloc\_chunk结构可以表示的最小内存块是16字节，最大内存块是248字节，因此malloc\_chunk可以表示16、24、32、40、......、248共30种长度的内存块。dlmalloc定义了30条链表，相同长度的空闲内存块保存在一个链表中。

1. Larger chunks:

struct malloc\_tree\_chunk {

/\* The first four fields must be compatible with malloc\_chunk \*/

size\_t prev\_foot;

size\_t head;

struct malloc\_tree\_chunk\* fd;

struct malloc\_tree\_chunk\* bk;

struct malloc\_tree\_chunk\* child[2];

struct malloc\_tree\_chunk\* parent;

bindex\_t index;

};

dlmalloc定义了32棵树存储大内存块，每棵树中存储若干种长度的内存块，这棵树中可能保存了多个相同长度的内存块，这些相同长度的内存块构成了一棵链表。

1. Segments

struct malloc\_segment {

char\* base; /\* base address \*/

size\_t size; /\* allocated size \*/

struct malloc\_segment\* next; /\* ptr to next segment \*/

flag\_t sflags; /\* mmap and extern flag \*/

};

5.malloc\_state:

struct malloc\_state {

binmap\_t smallmap;

binmap\_t treemap;

size\_t dvsize;

size\_t topsize;

char\* least\_addr;

mchunkptr dv;

mchunkptr top;

size\_t trim\_check;

size\_t release\_checks;

size\_t magic;

mchunkptr smallbins[(NSMALLBINS+1)\*2];

tbinptr treebins[NTREEBINS];

size\_t footprint;

size\_t max\_footprint;

size\_t footprint\_limit; /\* zero means no limit \*/

flag\_t mflags;

#if USE\_LOCKS

MLOCK\_T mutex; /\* locate lock among fields that rarely change \*/

#endif /\* USE\_LOCKS \*/

msegment seg;

void\* extp; /\* Unused but available for extensions \*/

size\_t exts;

};

6.malloc\_params:

struct malloc\_params {

size\_t magic;

size\_t page\_size;

size\_t granularity;

size\_t mmap\_threshold;

size\_t trim\_threshold;

flag\_t default\_mflags;

};

64位虚拟内存使用情况：

**@1：没进行堆分配：**

**//text segment**

00400000-00401000 r-xp 00000000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00600000-00601000 r--p 00000000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00601000-00602000 rw-p 00001000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

**//内存 map segment**

7f685ff24000-7f68600de000 r-xp 00000000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68600de000-7f68602de000 ---p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68602de000-7f68602e2000 r--p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68602e2000-7f68602e4000 rw-p 001be000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68602e4000-7f68602e9000 rw-p 00000000 00:00 0

7f68602e9000-7f6860302000 r-xp 00000000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860302000-7f6860501000 ---p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860501000-7f6860502000 r--p 00018000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860502000-7f6860503000 rw-p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860503000-7f6860507000 rw-p 00000000 00:00 0

7f6860507000-7f686052a000 r-xp 00000000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6860703000-7f6860706000 rw-p 00000000 00:00 0

7f6860725000-7f6860729000 rw-p 00000000 00:00 0

7f6860729000-7f686072a000 r--p 00022000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f686072a000-7f686072b000 rw-p 00023000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f686072b000-7f686072c000 rw-p 00000000 00:00 0

7fff845f8000-7fff8461a000 rw-p 00000000 00:00 0 [stack]

7fff846b6000-7fff846b8000 r-xp 00000000 00:00 0 [vdso]

7fff846b8000-7fff846ba000 r--p 00000000 00:00 0 [vvar]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

**在64位系统中使用前32位作为text和heap segment的地址。**

**使用前48位作为map和stack segment的地址。**

**@2第一次主线程分配堆以后：**

00400000-00401000 r-xp 00000000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00600000-00601000 r--p 00000000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00601000-00602000 rw-p 00001000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

**//从01792000开始分配132kb的虚拟内存**

**//132KB的堆空间叫做arena**

**//该处为主线程的arena 使用brk进行分配**

01792000-017b3000 rw-p 00000000 00:00 0 [heap]

7f685ff24000-7f68600de000 r-xp 00000000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68600de000-7f68602de000 ---p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68602de000-7f68602e2000 r--p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68602e2000-7f68602e4000 rw-p 001be000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f68602e4000-7f68602e9000 rw-p 00000000 00:00 0

7f68602e9000-7f6860302000 r-xp 00000000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860302000-7f6860501000 ---p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860501000-7f6860502000 r--p 00018000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860502000-7f6860503000 rw-p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6860503000-7f6860507000 rw-p 00000000 00:00 0

7f6860507000-7f686052a000 r-xp 00000000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6860703000-7f6860706000 rw-p 00000000 00:00 0

7f6860725000-7f6860729000 rw-p 00000000 00:00 0

7f6860729000-7f686072a000 r--p 00022000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f686072a000-7f686072b000 rw-p 00023000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f686072b000-7f686072c000 rw-p 00000000 00:00 0

**//主线程栈空间大小为132KB**

7fff845f8000-7fff8461a000 rw-p 00000000 00:00 0 [stack]

7fff846b6000-7fff846b8000 r-xp 00000000 00:00 0 [vdso]

7fff846b8000-7fff846ba000 r--p 00000000 00:00 0 [vvar]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

**@3//创建第一个子线程：**

00400000-00401000 r-xp 00000000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00600000-00601000 r--p 00000000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00601000-00602000 rw-p 00001000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

0214a000-0216b000 rw-p 00000000 00:00 0 [heap]

**//堆和子线程的栈空间相邻**

**//堆由低地址向高地址进行扩展**

**//栈有高地址向低地址进行扩展**

**//默认子线程的栈大小为8M**

**//4KB？？？？**

7f6d9d034000-7f6d9d035000 ---p 00000000 00:00 0

7f6d9d035000-7f6d9d835000 rw-p 00000000 00:00 0 [stack:15336]

7f6d9d835000-7f6d9d9ef000 r-xp 00000000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9d9ef000-7f6d9dbef000 ---p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9dbef000-7f6d9dbf3000 r--p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9dbf3000-7f6d9dbf5000 rw-p 001be000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9dbf5000-7f6d9dbfa000 rw-p 00000000 00:00 0

7f6d9dbfa000-7f6d9dc13000 r-xp 00000000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9dc13000-7f6d9de12000 ---p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9de12000-7f6d9de13000 r--p 00018000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9de13000-7f6d9de14000 rw-p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9de14000-7f6d9de18000 rw-p 00000000 00:00 0

7f6d9de18000-7f6d9de3b000 r-xp 00000000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6d9e014000-7f6d9e017000 rw-p 00000000 00:00 0

7f6d9e036000-7f6d9e03a000 rw-p 00000000 00:00 0

7f6d9e03a000-7f6d9e03b000 r--p 00022000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6d9e03b000-7f6d9e03c000 rw-p 00023000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6d9e03c000-7f6d9e03d000 rw-p 00000000 00:00 0

7fffeb80c000-7fffeb82e000 rw-p 00000000 00:00 0 [stack]

7fffeb9c9000-7fffeb9cb000 r-xp 00000000 00:00 0 [vdso]

7fffeb9cb000-7fffeb9cd000 r--p 00000000 00:00 0 [vvar]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

**@4子线程分配堆：**

00400000-00401000 r-xp 00000000 08:07 19136607 /work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00600000-00601000 r--p 00000000 08:07 19136607 work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

00601000-00602000 rw-p 00001000 08:07 19136607 work/DOC/all-doc/linux/dlmalloc堆分配器/test/dlmalloc

0214a000-0216b000 rw-p 00000000 00:00 0 [heap]

**//132KB？？？？（可读可写属性）**

7f6d98000000-7f6d98021000 rw-p 00000000 00:00 0

**//63M**

7f6d98021000-7f6d9c000000 ---p 00000000 00:00 0

7f6d9d034000-7f6d9d035000 ---p 00000000 00:00 0

7f6d9d035000-7f6d9d835000 rw-p 00000000 00:00 0 [stack:15336]

7f6d9d835000-7f6d9d9ef000 r-xp 00000000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9d9ef000-7f6d9dbef000 ---p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9dbef000-7f6d9dbf3000 r--p 001ba000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9dbf3000-7f6d9dbf5000 rw-p 001be000 08:06 135423 /lib/x86\_64-linux-gnu/libc-2.19.so

7f6d9dbf5000-7f6d9dbfa000 rw-p 00000000 00:00 0

7f6d9dbfa000-7f6d9dc13000 r-xp 00000000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9dc13000-7f6d9de12000 ---p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9de12000-7f6d9de13000 r--p 00018000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9de13000-7f6d9de14000 rw-p 00019000 08:06 132087 /lib/x86\_64-linux-gnu/libpthread-2.19.so

7f6d9de14000-7f6d9de18000 rw-p 00000000 00:00 0

7f6d9de18000-7f6d9de3b000 r-xp 00000000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6d9e014000-7f6d9e017000 rw-p 00000000 00:00 0

7f6d9e036000-7f6d9e03a000 rw-p 00000000 00:00 0

7f6d9e03a000-7f6d9e03b000 r--p 00022000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6d9e03b000-7f6d9e03c000 rw-p 00023000 08:06 135076 /lib/x86\_64-linux-gnu/ld-2.19.so

7f6d9e03c000-7f6d9e03d000 rw-p 00000000 00:00 0

7fffeb80c000-7fffeb82e000 rw-p 00000000 00:00 0 [stack]

7fffeb9c9000-7fffeb9cb000 r-xp 00000000 00:00 0 [vdso]

7fffeb9cb000-7fffeb9cd000 r--p 00000000 00:00 0 [vvar]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

**系统有几个core就会创建相应个数的arena，多个线程可以共享一个arena：**

假设有如下情境：一台只含有一个处理器核心的PC机安装有32位操作系统，其上运行了一个多线程应用程序，共含有4个线程——主线程和三个用户线程。显然线程个数大于系统能维护的最大arena个数（2\*核心数 + 1= 3），那么此时glibc malloc就需要确保这4个线程能够正确地共享这3个arena，那么它是如何实现的呢？

当主线程首次调用malloc的时候，glibc malloc会直接为它分配一个main arena，而不需要任何附加条件。

当用户线程1和用户线程2首次调用malloc的时候，glibc malloc会分别为每个用户线程创建一个新的thread arena。此时，各个线程与arena是一一对应的。但是，当用户线程3调用malloc的时候，就出现问题了。因为此时glibc malloc能维护的arena个数已经达到上限，无法再为线程3分配新的arena了，那么就需要重复使用已经分配好的3个arena中的一个(main arena, arena 1或者arena 2)。那么该选择哪个arena进行重复利用呢？

1)首先，glibc malloc循环遍历所有可用的arenas，在遍历的过程中，它会尝试lock该arena。如果成功lock(该arena当前对应的线程并未使用堆内存则表示可lock)，比如将main arena成功lock住，那么就将main arena返回给用户，即表示该arena被线程3共享使用。

2)而如果没能找到可用的arena，那么就将线程3的malloc操作阻塞，直到有可用的arena为止。

3)现在，如果线程3再次调用malloc的话，glibc malloc就会先尝试使用最近访问的arena(此时为main arena)。如果此时main arena可用的话，就直接使用，否则就将线程3阻塞，直到main arena再次可用为止。

这样线程3与主线程就共享main arena了。至于其他更复杂的情况，以此类推。