



一次 Java 进程 OOM 的排查分析 (glibc 篇) 原创

挖坑的张师傅

2年前

© 15047

11

30

♂

33.4W

阅读

机械工业出版社《深入作者，掘金小册作者》

推荐阅读

调优 mybatis save

【全网首发】解Bu

一文读懂并发包中

图解并发包中锁的

Java限流及常用解

一文带你彻底理解

背景

前段时间有同学反馈一个 java RPC 项目在容器中启动完没多久就因为容器内存超过配额 1500M 被杀，我帮忙一起看了一下。

The screenshot shows a container management interface. A pod named 'rpc:stable-bd705854-07231134' is listed as '失败' (Failed). The '之前状态' (Previous State) shows '容器 ExitCode : 137, 原因: OOMKilled'. The '当前状态' (Current State) shows '运行中, 开始时间: 今天 [07-25] 21:35:02'. There are tabs for '特权终端', '容器日志', and '实例'.

在本地 Linux 环境中跑了一下，JVM 启动完通过 top 看到的 RES 内存就已经超过了 1.5G，如下图所示。

The screenshot shows the output of the 'top' command on a Linux system. The 'RES' column for the 'java' process (PID 5533) is highlighted and shows a value of '1.520g', which is significantly higher than the other processes listed. Other processes shown include 'root' and 'ya' with lower memory usage.

首先想到查看内存的分布情况，使用 arthas 是一个不错的选择，输入 dashboard 查看当前的内存使用情况，如下所示。

Memory	used	total	max	usage
heap	336M	965M	965M	34.88%
ps Eden Space	37M	237M	243M	15.59%
ps Survivor Space	45M	45M	45M	99.97%
ps Old Gen	253M	683M	683M	37.09%
Nonheap	154M	158M	-1	97.59%
Code Cache	34M	34M	240M	14.17%
Metaspace	106M	109M	-1	97.84%

专题推荐

15 篇文章

- 入门篇一：写给小白的 Java 课
- 入门篇二：Java 课
- 入门篇三：Java 虚

查看



15 篇文章

ptmalloc2 与 arena

Linux 中 malloc 的早期版本是由 Doug Lea 实现的，它有一个严重问题是内存分配只有一个分配区（arena），每次分配内存都要对分配区加锁，分配完释放锁，导致多线程下并发申请释放内存锁的竞争激烈。arena 单词的字面意思是「舞台；竞技场」，可能就是内存分配库表演的主战场的意思吧。

于是修修补补又一个版本，你不是多线程锁竞争厉害吗，那我多开几个 arena，锁竞争的情况自然会好转。

Wolfram Gloger 在 Doug Lea 的基础上改进使得 Glibc 的 malloc 可以支持多线程，这就是 ptmalloc2。在只有一个分配区的基础上，增加了非主分配区(non main arena)，主分配区只有一个，非主分配可以有很多个，具体个数后面会说。

当调用 malloc 分配内存的时候，会先查看当前线程私有变量中是否已经存在一个分配区 arena。如果存在，则尝试会对这个 arena 加锁。

- 如果加锁成功，则会使用这个分配区分配内存
- 如果加锁失败，说明有其它线程正在使用，则遍历 arena 列表寻找没有加锁的 arena 区域，如果找到则用这个 arena 区域分配内存。

主分配区可以使用 brk 和 mmap 两种方式申请虚拟内存，非主分配区只能 mmap。glibc 每次申请的虚拟内存区块大小是 **64MB**，glibc 再根据应用需要切割为小块零售。

这就是 linux 进程内存分布中典型的 64M 问题，那有多少个这样的区域呢？在 64 位系统下，这个值等于 **8 * number of cores**，如果是 4 核，则最多有 32 个 64M 大小的内存区域。

难道是因为 arena 数量太多了导致的？

设置 MALLOC_ARENA_MAX=1 有用吗？

加上这个环境变量启动 java 进程，确实 64M 的内存区域就不见了，但是集中到了一个大的接近 700M 的内存区域中，如下图所示。



Address	Kbytes	RSS	Dirty	Mode	Mapping
00000000c0000000	1061800	740308	739276	rw---	[anon]
0000000100cea000	1035352	0	0	----	[anon]
000055555554000	4	0	0	r-x--	java
0000555555754000	4	4	4	r----	java
0000555555755000	4	0	0	rw---	java
000055555756000	1960956	694068	692608	rw---	[anon]
00007ffffc9e6d000	4	0	0	----	[anon]
00007ffffc9e6d000	8192	16	16	-----	[anon]

是谁在分配释放内存

接下来，写一个自定义的 malloc 函数 hook。hook 实际上就是利用 LD_PRELOAD 环境变量替换 glibc 中的函数实现，在 malloc、free、realloc、calloc 这几个函数调用前先打印日志然后再调用实际的方法。以 malloc 函数的 hook 为例，部分代码如下所示。

```
// 获取线程 id 而不是 pid
static pid_t gettid() {
    return syscall(__NR_gettid);
}
static void *(*real_realloc)(void *ptr, size_t size) = 0;

void *malloc(size_t size) {
    void *p;
    if (!real_malloc) {
        real_malloc = dlsym(RTLD_NEXT, "malloc");
        if (!real_malloc) return NULL;
    }
    p = real_malloc(size);
    printLog("[0x%08x] malloc(%u)\t= 0x%08x ", GETRET(), size, p);
    return p;
}
```

设置 LD_PRELOAD 启动 JVM

```
LD_PRELOAD=/app/my_malloc.so java -Xms -Xmx -jar ....
```

在 JVM 启动的过程中同时开启 jstack 打印线程堆栈，当 jvm 进程完全启动以后，查看 malloc 的输出日志和 jstack 的日志。

这里输出了一个几十 M 的 malloc 日志，内容如下所示。日志的第一列是线程 id。

```

16342 00:00:00.000430 [0x00007fff] realloc(0xf0005810, 128) = 0xf0005a50
16342 00:00:00.000431 [0x00000000] malloc(20)    = 0xf0005ae0
16342 00:00:00.000432 [0x00007fff] realloc(0xf0005a50, 136) = 0xf0005a50
16342 00:00:00.000433 [0x00000000] malloc(37)    = 0xf0005b00
16342 00:00:00.000434 [0x00000000] malloc(29)    = 0xf0005b30
16342 00:00:00.000435 [0x00007fff] realloc(0xf0005a50, 144) = 0xf0005b60
16342 00:00:00.000436 [0x00000000] malloc(33)    = 0xf0005a50
16342 00:00:00.000437 [0x00007fff] realloc(0xf0005b60, 152) = 0xf0005b60
16342 00:00:00.000438 [0x00000000] malloc(33)    = 0xf0005a80
16342 00:00:00.000439 [0x00007fff] realloc(0xf0005b60, 160) = 0xf0005b60
16342 00:00:00.000440 [0x00000000] malloc(23)    = 0xf0005ab0
16342 00:00:00.000441 [0x00007fff] realloc(0xf0005b60, 168) = 0xf0005b60
16342 00:00:00.000442 [0x00000000] malloc(19)    = 0xf0005c10
16342 00:00:00.000443 [0x00007fff] realloc(0xf0005b60, 176) = 0xf0005c30
16342 00:00:00.000444 [0x00000000] malloc(22)    = 0xf0005b60
16342 00:00:00.000445 [0x00007fff] malloc(30)   = 0xf0005b80
16342 00:00:00.000446 [0x00007fff] malloc(22)   = 0xf0005bb0
16342 00:00:00.000447 [0x00007fff] malloc(48)   = 0xf0005bd0
16342 00:00:00.000448 [0x00007fff] malloc(30)   = 0xf0005cf0
16342 00:00:00.000449 [0x00007fff] malloc(22)   = 0xf0005d20

```

使用 awk 处理上的日志，统计线程处理的次数。

```
cat malloc.log | awk '{print $1}' | less | sort | uniq -c | sort -rn | less
```

```

284881 16342
 135 16341
 57 16349
 16 16346
 10 16345
  9 16351
  9 16350
  6 16343
  5 16348
  4 16347
  1 16352
  1 16344

```



可以看到线程 16342 分配释放内存最为凶残，那这个线程在做什么呢？在 jstack 的输出日志中搜索 16342 (0x3fd6) 线程，可以看到很多次都在处理 jar 包的解压。

```

"main" #1 prio=5 os_prio=0 tid=0x00007ffff0054000 nid=0x3fd6 runnable [0x00007ffff7fe6000
  java.lang.Thread.State: RUNNABLE
    at java.util.zip.ZipFile.read(Native Method)
    at java.util.zip.ZipFile.access$1400(ZipFile.java:60)
    at java.util.zip.ZipFile$ZipFileInputStream.read(ZipFile.java:734)
    - locked <0x00000000c009f6e0> (a java.util.jar.JarFile)
    at java.util.zip.ZipFile$ZipFileInflaterInputStream.fill(ZipFile.java:434)
    at java.util.zip.InflaterInputStream.read(InflaterInputStream.java:158)
    at sun.misc.Resource.getBytes(Resource.java:124)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:463)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:74)

-->
"main" #1 prio=5 os_prio=0 tid=0x00007ffff0054000 nid=0x3fd6 runnable [0x00007ffff7fe4000
  java.lang.Thread.State: RUNNABLE
    at java.util.zip.ZipFile.read(Native Method)
    at java.util.zip.ZipFile.access$1400(ZipFile.java:60)
    at java.util.zip.ZipFile$ZipFileInputStream.read(ZipFile.java:734)
    - locked <0x00000000c009f6e0> (a java.util.jar.JarFile)
    at java.util.zip.ZipFile$ZipFileInflaterInputStream.fill(ZipFile.java:434)
    at java.util.zip.InflaterInputStream.read(InflaterInputStream.java:158)
    at sun.misc.Resource.getBytes(Resource.java:124)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:463)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:74)

```

java 处理 zip 使用的是 `java.util.zip.Inflater` 类，调用它的 `end` 方法会释放 native 的内存。看到这里我以为是 `end` 方法没有调用导致的，这种的确是有可能的，`java.util.zip.InflaterInputStream` 类的 `close` 方法在一些场景下是不会调用 `Inflater.end` 方法，如下所示。

```

4     *}
5     public void close() throws IOException {
6         if (!closed) { java.util.zip.InflaterInputStream.closed: false
7             if (usesDefaultInflater) usesDefaultInflater: false
8                 inf.end();
9                 in.close();
10                closed = true;
11            }
12        }

```

高兴的有点早了。实际上并非如此，就算上层调用没有调用 Inflater.end，Inflater 类的 finalize 方法也调用了 end 方法，我强行 GC 试一下。

```
jcmd `pidof java` GC.run
```

通过 GC 日志确认确实触发了 FullGC，但内存并没有降下来。通过 valgrind 等工具查看内存泄露，也没有什么发现。如果说 JVM 本身的实现没有内存泄露，那就是 glibc 自己的问题了，调用 free 把内存还给了 glibc，glibc 并没有最终释放，这个内存二道贩子自己把内存截胡了。

glibc 的内存分配原理

这是一个很复杂的话题，如果这一块完全不熟悉，建议你先看看下面这几个资料。

- [Understanding glibc malloc](#)
- [淘宝华庭大师的《Glibc 内存管理 - Ptmalloc2 源代码分析》](#)

总体来看，需要理解下面这几个概念：

- 内存分配区 Arena
- 内存 chunk
- 空闲 chunk 的回收站（bins）

内存分配区 Arena

内存分配区 Arena 的概念在前面介绍过，也比较简单。为了更直观的了解 heap 的内部结构，可以使用 gdb 的 heap 扩展包，比较常见的有：

- [libheap](#)
- [Pwngdb](#)
- [pwndbg](#)

这些也是打 CTF 堆相关的题目可以使用的工具，接下来使用的是 Pwngdb 工具来介绍。输入 arenainfo 可以查看 Arena 的列表，如下所示。



```

gdb-peda$ arenasinfo
===== Main Arena =====
(0x20)    fastbin[0]: 0x0
(0x30)    fastbin[1]: 0x555555765540 --> 0x55555575f270 --> 0x5555
(0x40)    fastbin[2]: 0x555555762840 --> 0x0
(0x50)    fastbin[3]: 0x0
(0x60)    fastbin[4]: 0x0
(0x70)    fastbin[5]: 0x0
(0x80)    fastbin[6]: 0x0
(0x90)    fastbin[7]: 0x0
(0xa0)    fastbin[8]: 0x0
(0xb0)    fastbin[9]: 0x0
                           top: 0x555555766240 (size : 0x10dc0)
    last_remainder: 0x555555762880 (size : 0x40)
    unsortbin: 0x5555557657f0 (size : 0x950) <--> 0x5555557
(0x020)  smallbin[ 0]: 0x55555575a220 <--> 0x55555575f2a0 <--> 0x
(0x120)  smallbin[16]: 0x555555762ae0
===== Arena 1 =====
(0x20)    fastbin[0]: 0x0
(0x30)    fastbin[1]: 0x7fff980008b0 --> 0x0
(0x40)    fastbin[2]: 0x7fff98001560 --> 0x7fff98001520 --> 0x7fff
(0x50)    fastbin[3]: 0x7fff98001910 --> 0x7fff98001820 --> 0x0
(0x60)    fastbin[4]: 0x0
(0x70)    fastbin[5]: 0x7fff98001be0 --> 0x7fff98001b70 --> 0x0
(0x80)    fastbin[6]: 0x0
(0x90)    fastbin[7]: 0x0
(0xa0)    fastbin[8]: 0x0
(0xb0)    fastbin[9]: 0x0
                           top: 0x7fff98003140 (size : 0x1dec0)

```

在这个例子中，有 1 个主分配区 Arena 和 15 个非主分配区 Arena。

内存 chunk 的结构

chunk 的概念也比较好理解，chunk 的字面意思是「大块」，是面向用户而言的，用户申请分配的内存用 chunk 表示。

可能这样说还是不好理解，下面一个实际的例子来说明。

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    void *p;

    p = malloc(1024);
    printf("%p\n", p);

    p = malloc(1024);
    printf("%p\n", p);

    p = malloc(1024);
    printf("%p\n", p);

    getchar();
    return (EXIT_SUCCESS);
}

```

这段代码分配了三次 1k 大小的内存，内存地址是：

```

./malloc_test
0x602010

```

0x602420
0x602830

pmap 输出的结果如下所示。

Address	Kbytes	RSS	Dirty	Mode	Mapping
0000000000400000	4	4	0	r-x--	malloc_test
0000000000600000	4	4	4	r----	malloc_test
0000000000601000	4	4	4	rw---	malloc_test
0000000000602000	132	4	4	rw---	[anon]
00007ffff7a0d000	1804	288	0	r-x--	libc-2.17.s
00007ffff7bd0000	2048	0	0	-----	libc-2.17.s
00007ffff7dd0000	16	16	16	r----	libc-2.17.s
00007ffff7dd4000	8	8	8	rw---	libc-2.17.s
00007ffff7dd6000	20	12	12	rw---	[anon]
00007ffff7ddb000	136	108	0	r-x--	ld-2.17.so
00007ffff7fed000	12	12	12	rw---	[anon]
00007ffff7ff7000	12	8	8	rw---	[anon]

可以看到第一次分配的内存区域地址 0x602010 在这块内存区域的基址(0x602000)偏移量 16(0x10)的地方。

再来看第二次分配的内存区域地址 0x602420 与 0x602010 的差值是 $1,040 = 1024 + 16(0x10)$

第三次分配的内存以此类推是一样的，每次都空了 0x10 个字节。这中间空出来的 0x10 是什么呢？

使用 gdb 查看一下就很清楚了，查看这三个内存地址往前 0x10 字节开始的 32 字节区域。

```

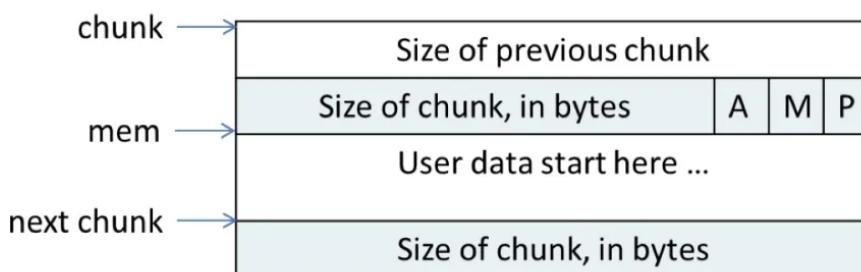
gdb-peda$ x/32bx 0x602010- 0x10
0x602000: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602008: 0x11 0x04 0x00 0x00 0x00 0x00 0x00 0x00
0x602010: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602018: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
gdb-peda$ x/32bx 0x602420- 0x10
0x602410: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602418: 0x11 0x04 0x00 0x00 0x00 0x00 0x00 0x00
0x602420: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602428: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
gdb-peda$ x/32bx 0x602830- 0x10
0x602820: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602828: 0x11 0x04 0x00 0x00 0x00 0x00 0x00 0x00
0x602830: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602838: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

```

可以看到实际上存储的是 0x0411，

$$0x0411 = 1024(0x0400) + 0x10(\text{block size}) + 0x01$$

其中 1024 很明显，是用户申请的内存区域大小，0x11 是什么？因为内存分配都会对齐，实际上最低 3 位对内存大小没有什么意义，最低 3 位被借用来表示特殊含义。一个使用中的 chunk 结构如下图所示。



最低三位的含义如下：

- A: 表示该 chunk 属于主分配区或者非主分配区，如果属于非主分配区，将该位置为 1，否则置为 0
- M: 表示当前 chunk 是从哪个内存区域获得的内存。M 为 1 表示该 chunk 是从 mmap 映射区域分配的，否则是从 heap 区域分配的
- P: 表示前一个块是否在使用中，P 为 0 则表示前一个 chunk 为空闲，这时 chunk 的第一个域 prev_size 才有效

这个例子中最低三位是 b001，A = 0 表示这个 chunk 不属于主分配区，M = 0，表示是从 heap 区域分配的，P = 1 表示前一个 chunk 在使用中。

从 glibc 源码中可以看的更清楚一些。

```
#define PREV_INUSE 0x1
/* extract inuse bit of previous chunk */
#define prev_inuse(p) ((p)->size & PREV_INUSE)

#define IS_MAPPED 0x2
/* check for mmap()'ed chunk */
#define chunk_is_mapped(p) ((p)->size & IS_MAPPED)

#define NON_MAIN_ARENA 0x4
/* check for chunk from non-main arena */
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)

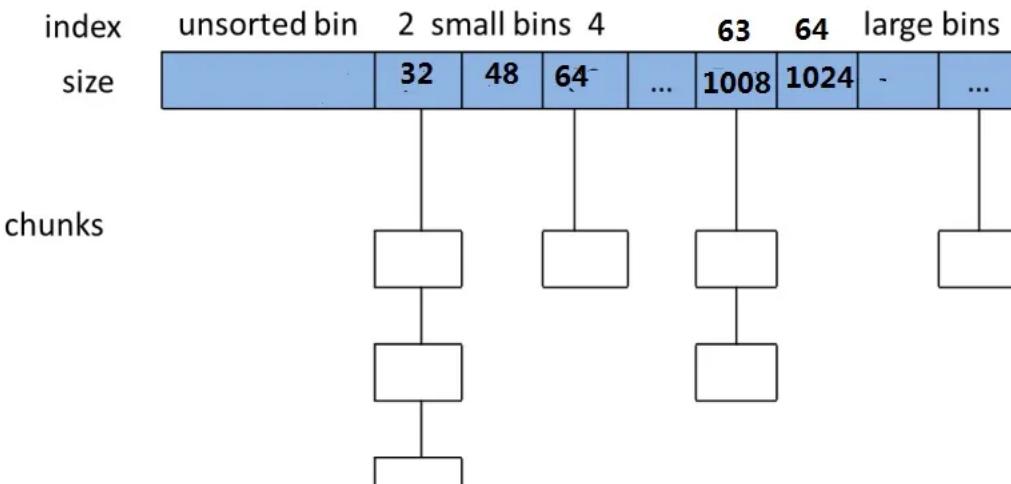
#define SIZE_BITS (PREV_INUSE|IS_MAPPED|NON_MAIN_ARENA)
/* Get size, ignoring use bits */
#define chunksize(p) ((p)->size & ~SIZE_BITS)
```

前面介绍的是 allocatd chunk 的结构，被 free 以后的空闲 chunk 的结构不太一样，还有一个称为 top chunk 的结构，这里不再展开。

chunk 的回收站 bins

bin 的字面意思是「垃圾箱」。内存应用调用 free 释放以后 chunk 不一定会立刻归还给系统，而是就被 glibc 这个二道贩子截胡。这也是为了效率的考量，当用户下次请求分配内存时，ptmalloc2 会先尝试从空闲 chunk 内存池中找到一个合适的内存区域返回给应用，这样就避免了频繁的 brk、mmap 系统调用。

为了更高效的管理内存分配和回收，ptmalloc2 中使用了一个数组，维护了 128 个 bins。



这些 bin 的介绍如下。

- bin0 目前没有使用
- bin1 是 unsorted bin，主要用于存放刚刚释放的 chunk 堆块以及大堆块分配后剩余的堆块，大小没有限制
- bin2~bin63 是 small bins，用于维护 < 1024B 的 chunk 内存块，同一条 small bin 链中的 chunk 具有相同的大 小，都为 $\text{index} * 16$ ，比如 bin2 对应的链表的 chunk 大小都是 32(0x20)，bin3 对应的链表的 chunk 大小为 48(0x30)。备注：淘宝的 pdf 图中有点问题，pdf 中的 size * 8，看源码，应该是 *16 才对
- bin64~bin126 是 large bins，用于维护 > 1024B 的堆块，同一条链表中的堆块大小不一定相同，具体的规则不 是本篇介绍的重点，不再展开。

具体到本例中，在 Pwngdb 中可以查看每个 arena 的 bins 信息。如下图所示。

```
===== Arena 15 =====
(0x20)    fastbin[0]: 0x0
(0x30)    fastbin[1]: 0x0
(0x40)    fastbin[2]: 0x7fff647c8510 --> 0x7fff6a8f6ce0 --> 0x7fff6ac89040 --> 0x7ffff0e5
9a890 --> 0x7fff6a19a8d0 --> 0x7fff6a87a150 --> 0x7fff6a87c720 --> 0x7fff6a87c760 --> 0x7f
ffff6a87a0d0 --> 0x7fff6a87a110 --> 0x7ffff3f25250 --> 0x7ffff3f25290 --> 0x7fff6a8f9150 -->
--> 0x7ffff3f70590 --> 0x7fff648e35d0 --> 0x7ffff1587b40 --> 0x7fff96a6d020 --> 0x7ffff1014
b100 --> 0x7ffff1259be0 --> 0x7ffff0e75860 --> 0x7ffff136a270 --> 0x7ffff136a2b0 --> 0x7ff
(0x50)    fastbin[9]: 0x0
    top: 0x7fff2830eac0 (size : 0x11540)
    last_remainder: 0x7fff6a9cef40 (size : 0x8810)
    unsortbin: 0x7fff6a8faed0 (size : 0x8000) <--> 0x7fff6ac4fe90 (size : 0x130) <
    0x7ffff414540 (size : 0x2020) <--> 0x7ffff414540 (size : 0x2020) <--> 0x7ffff414540
(0x20)    smallbin[ 0]: 0x7fff6a19fae0 <--> 0x7fff6a9c3690 <--> 0x7fff6a9d8e00 <--> 0x7fff6a9cbdd0

(0x80)    smallbin[ 6]: 0x7fff6a9cdac0
    largebin[62]: 0x7fff50000020 (size : 0x3fffc0) <--> 0x7fff8c000020 (size : 0x3ffd60) <->
    ffd50 <--> 0x7fff34000020 (size : 0x3fffc0) <--> 0x7fff90000020 (size : 0x3fffc0) <--> 0x7fff8
-> 0x7fff5c000020 (size : 0x3fffc0) <--> 0x7fff54000020 (size : 0x3fffc0) <--> 0x7fff3c000020 (s
40000020 (size : 0x3fffc70) <--> 0x7fff2c000020 (size : 0x3ffbfc0) <--> 0x7fff40000020 (size : 0x3f
ize : 0x3ffbfc0) <--> 0x7fff70000020 (size : 0x3ffbfc0) <--> 0x7fff58000020 (size : 0x3ffbfc0) <-->
fbfc0) <--> 0x7fff38000020 (size : 0x3ffbfc0) <--> 0x7fff48000020 (size : 0x3ffbfc90) <--> 0x7fff60
> 0x7fff84000020 (size : 0x3ffbfc60) <--> 0x7fff6c000020 (size : 0x3ffbfc60) <--> 0x7fff64a17420 (si
000020 (size : 0x3532ae0) <--> 0x7fff44d12d20 (size : 0x32e9260) <--> 0x7fff30000020 (size : 0x2ae
```

fastbin

一般情况下，程序在运行过程中会频繁分配一些小的内存，如果这些小内存被频繁的合并和切割，效率会比较低下，因此 ptmalloc 在除了上面的 bin 组成部分，还有一个非常重要的结构 fastbin，专门用来管理小的内存堆块。

64 位系统中，不大于 128 字节的内存堆块被释放以后，首先会被放到 fastbin 中，fastbin 中的 chunk 的 P 标记始终为 1，fastbin 的堆块会被当做使用中，因此不会被合并。

在分配小于 128 字节的内存时，ptmalloc 会首先在 fastbin 中查找对应的空闲块，如果没有才去其它 bins 中查找。

换个角度来看，fastbin 可以看做是 smallbin 的一道缓存。

内存碎片与回收

接下来我们来做一个实验，看看内存碎片如何影响 glibc 的内存回收，代码如下所示。

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define K (1024)
#define MAXNUM 500000

int main() {

    char *ptrs[MAXNUM];
    int i;
    // malloc large block memory
    for (i = 0; i < MAXNUM; ++i) {
        ptrs[i] = (char *)malloc(1 * K);
        memset(ptrs[i], 0, 1 * K);
    }
    //never free, only 1B memory leak, what it will impact to the system?
    char *tmp1 = (char *)malloc(1);
    memset(tmp1, 0, 1);

    printf("%s\n", "malloc done");
    getchar();

    printf("%s\n", "start free memory");
    for(i = 0; i < MAXNUM; ++i) {
        free(ptrs[i]);
    }
    printf("%s\n", "free done");

    getchar();

    return 0;
}
```




```

static int mtrim(mstate av, size_t pad)
{
    /* Ensure initialization/consolidation */
    malloc_consolidate(av);

    const size_t ps = GLRO(dl_pagesize);
    int psindex = bin_index(ps);
    const size_t psm1 = ps - 1;

    int result = 0;
    for (int i = 1; i < NBINS; ++i)
        if (i == 1 || i >= psindex)
    {
        mbinptr bin = bin_at(av, i);

        for (mchunkptr p = last(bin); p != bin; p = p->bk)
        {
            INTERNAL_SIZE_T size = chunkszie(p);

            if (size > psm1 + sizeof(struct malloc_chunk))
            {
                /* See whether the chunk contains at least one unused page. */
                char *paligned_mem = (char *) (((uintptr_t) p
                    + sizeof(struct malloc_chunk)
                    + psm1) & ~psm1);

                assert ((char *) chunk2mem(p) + 4 * SIZE_SZ <= paligned_mem);
                assert ((char *) p + size > paligned_mem);

                /* This is the size we could potentially free. */
                size -= paligned_mem - (char *) p;

                if (size > psm1)
                {
#define MALLOC_DEBUG
                    /* When debugging we simulate destroying the memory
                     * content. */
                    memset(paligned_mem, 0x89, size & ~psm1);
#endif
                    __madvise(paligned_mem, size & ~psm1, MADV_DONTNEED);
                }
            }
        }
    }
}

```

通过 Systemtap 脚本可以同步确认这一点。

```

probe begin {
    log("begin to probe\n")
}

probe kernel.function("SYSC_madvise") {
    if (ppid() == target()) {
        printf("\n%#s: %s\n", probefunc(), $$vars)
        print_backtrace();
    }
}

```

执行 `malloc_trim` 时，有大量的 madvise 系统调用，如下图所示。

```
in Sys_madvise: behavior=0x4 len_in=0x2ac000 start=0x7fff89760000
0xffffffff811ac384 : SyS_madvise+0x34/0x9c0 [kernel]
0xffffffff816b51d2 : tracesys+0xdd/0xe2 [kernel]

in Sys_madvise: behavior=0x4 len_in=0x325000 start=0x7fff8fa85000
0xffffffff811ac384 : SyS_madvise+0x34/0x9c0 [kernel]
0xffffffff816b51d2 : tracesys+0xdd/0xe2 [kernel]

in Sys_madvise: behavior=0x4 len_in=0x335000 start=0x7fff79264000
0xffffffff811ac384 : SyS_madvise+0x34/0x9c0 [kernel]
0xffffffff816b51d2 : tracesys+0xdd/0xe2 [kernel]

in Sys_madvise: behavior=0x4 len_in=0x36f000 start=0x7fff72f97000
0xffffffff811ac384 : SyS_madvise+0x34/0x9c0 [kernel]
0xffffffff816b51d2 : tracesys+0xdd/0xe2 [kernel]
```

这里的 behavior=0x4 表示是 MADV_DONTNEED, len_in 表示长度, start 表示内存开始地址。

malloc_trim 对前一个节中的内存碎片实验同样是生效的。

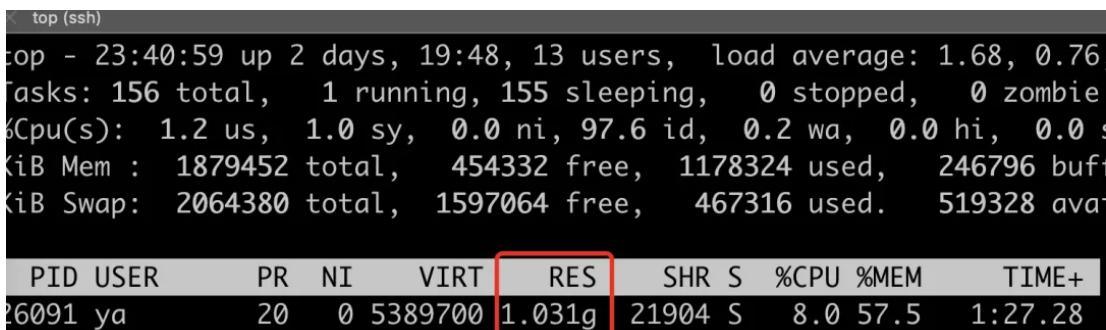
jemalloc 登场

既然是因为 glibc 的内存分配策略导致的碎片化内存回收问题, 导致看起来像是内存泄露, 那有没有更好一点的对碎片化内存的 malloc 库呢? 业界常见的有 google 家的 tcmalloc 和 facebook 家的 jemalloc。

这两个我都试过, jemalloc 的效果比较明显, 使用 LD_PRELOAD 挂载 jemalloc 库。

```
LD_PRELOAD=/usr/local/lib/libjemalloc.so
```

重新启动 Java 程序, 可以看到内存 RES 消耗降低到了 1G 左右



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+
26091	ya	20	0	5389700	1.031g	21904	S	8.0	57.5	1:27.28

使用 jemalloc 比 glibc 小了 500M 左右, 只比 malloc_trim 的 900 多 M 多了一点点。

至于为什么 jemalloc 在这个场景这么厉害, 又是一个复杂的话题, 这里先不展开, 有时间可以详细介绍一下 jemalloc 的实现原理。

经多次实验, malloc_trim 有概率会导致 JVM Crash, 使用的时候需要小心。

经过替换 ptmalloc2 为 jemalloc, 进程的内存 RES 占用显著下降, 至于性能、稳定性还需进一步观察。

番外篇

最近也在写一个简单的 malloc 库, 可能真正写过才知道 tcmalloc、jemalloc 到底想解决什么痛点, 复杂设计的背后权衡是如何做的。

小结

内存相关的问题是相对而言比较复杂的, 影响的因素很多, 如果排除是应用层自己的问题, 那是最简单的, 如果是 glibc 或者内核本身的问题, 那就只能通过大胆假设, 一点点验证了。关于内存的分配和管理是一个比较复杂的话题, 希望可以在后面的文章中再详细介绍介绍。

上面的内容可能都是错的, 看看思路就好。

点赞

收藏



挖坑的张师傅

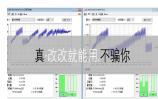
机械工业出版社《深入理解 JVM 字节码》作者，掘金小册作者《JVM 字节码从入门到精通》、《深入理解TCP协议》作者，Vim 死忠粉、Kotlin&Go 爱好者、能抓一手好包、喜欢底层技术和分享。微信公众号：张师傅的...

请先登录，查看11条评论吧

快去登录吧，你将获得

- 浏览更多精彩评论
- 和开发者讨论交流，共同进步

为你推荐



虽然是我遇到的一个棘手的生产问题，...

jvm java 多线程



刺激，线程池的一个BUG直接把CPU...

java



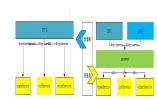
“堆外内存”这玩意是真不错，我要写进...

java 堆内存



单服务并发出票实践

java 多线程 性能优化



设计模式在我工作中的实践

java 设计模式



【全网首发】记一次SSL握手导致业务...

java 线程阻塞 网络

本月精选性能专题

- Linux性能优化—CPU性能篇
- Linux性能优化—内存性能篇
- Linux系统—内核源码分析
- 由浅入深了解GC原理
- 由浅入深理解Java线程池
- MQ DLedger多副本切换与主从切换
- 小白必看的JVM入门教程

本月精选线上案例

- 记一次服务器被入侵排查
- 记一次业务线程阻塞案例分析
- https证书链不完整导致请求失败
- FullGC没及时处理，差点造成PO事故
- 两行代码把CPU使用率干到了90%+
- 记一次数据库连接泄漏导致的响应迟缓
- 一次Nginx配置proxy_pass的404问题

本月精选原创好文

- OTSClient连接池连接无法复用
- JDK Bug导致线程阻塞案例分析
- 聊一聊Tomcat启动慢背后的真相
- 微服务10：系统服务熔断、限流
- 总结mysql所有buffer，一网打尽就这篇
- CPU 是如何与内存交互的？
- 幽灵攻击与编译器中的消减方法介绍

