linux环境内存分配原理 mallocinfo

转载 于 2020-07-24 22:41:23 发布 · 2.7k 阅读 · 3 · 14 · CC 4.0 BY-SA版权

文章标签： #malloc #动态内存 #sbrk #mmap

目录

Linux的虚拟内存管理有几个关键概念:

Linux 虚拟地址空间如何分布？malloc和free是如何分配和释放内存？如何查看堆内内存的碎片情况？既然堆内内存brk和sbrk不能直接释放，为什么不全用 mmap 来分配，munmap直接释放呢 ？

Linux 的虚拟内存管理有几个关键概念：
1、每个进程都有独立的虚拟地址空间，进程访问的虚拟地址并不是真正的物理地址；
2、虚拟地址可通过每个进程上的页表(在每个进程的内核虚拟地址空间)与物理地址进行映射，获得真正物理地址；
3、如果虚拟地址对应物理地址不在物理内存中，则产生缺页中断，真正分配物理地址，同时更新进程的页表；如果此时物理内存已耗尽，则根据内存替换淘汰部分页面至物理磁盘中。

1、Linux 虚拟地址空间如何分布?

Linux 使用虚拟地址空间，大大增加了进程的寻址空间，**由低地址到高地址分别为**:
1、只读段：该部分空间只能读，不可写；(包括：代码段、rodata 段(C常量字符串和#define定义的常量))
2、数据段：保存全局变量、静态变量的空间;
3、堆 ：就是平时所说的动态内存， malloc /new 大部分都来源于此。其中堆顶的位置可通过函数 brk 和 sbrk 进行动态调整。
4、文件映射区域：如动态库、共享内存等映射物理空间的内存，一般是 mmap 函数所分配的虚拟地址空间。
5、栈：用于维护函数调用的上下文空间，一般为 8M ， 可通过 ulimit –s 查看。
6、内核虚拟空间：用户代码不可见的内存区域，由内核管理(页表就存放在内核虚拟空间)。
下图是 32 位系统典型的虚拟地址空间分布(来自《深入理解计算机系统》)。

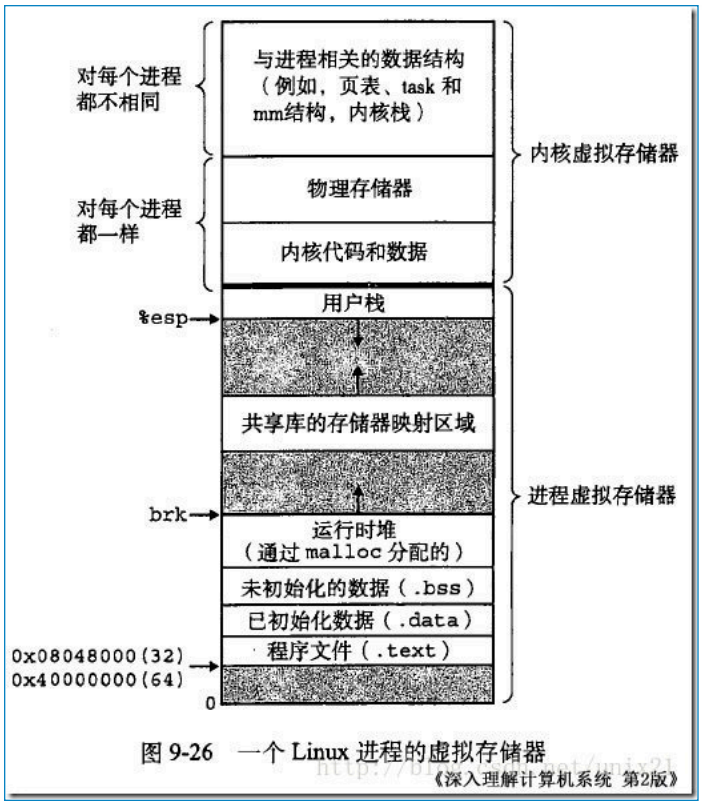


图 9-26 一个 Linux 进程的虚拟存储器

《深入理解计算机系统 第2版》

32 位系统有4G 的地址空间::

其中 0x08048000~0xbfffffff 是用户空间，0xc0000000~0xffffffff 是内核空间，包括内核代码和数据、与进程相关的数据结构（如页表、内核栈）等。另外，**执行栈顶，往低地址方向变化；brk/sbrk 函数控制堆顶_edata往高地址方向变化。**

64位系统结果怎样呢？ 64 位系统是否拥有 2^64 的地址空间吗?

事实上， 64 位系统的虚拟地址空间划分发生了改变：

1、地址空间大小不是2^32，也不是2^64，而一般是2^48。

因为并不需要 2^64 这么大的寻址空间，过大空间只会导致资源的浪费。64位Linux一般使用48位来表示虚拟地址空间，40位表示物理地址，

这可通过#cat /proc/cpuinfo 来查看：

```
[root@localhost ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 58
model name      : Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz
stepping        : 9
cpu MHz         : 2366.032
cache size      : 6144 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 cl
flush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good pni ssse3 lahf_lm
bogomips        : 4732.06
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```


2、其中，0x0000000000000000~0x00007fffffffffff 表示用户空间， 0xFFFF800000000000~ 0xFFFFFFFFFFFFFFFF 表示内核空间，共提供 256TB(2^48 址空间。

这两个区间的特点是，第 47 位与 48~63 位相同，若这些位为 0 表示用户空间，否则表示内核空间。

3、用户空间由低地址到高地址仍然是**只读段、数据段、堆、文件映射区域和栈**；

2、malloc和free是如何分配和释放内存?

如何查看进程发生缺页中断的次数?

用# ps -o majflt,minflt -C program 命令查看

majflt代表major fault，中文名叫大错误，minflt代表minor fault，中文名叫小错误。

这两个数值表示一个进程自启动以来所发生的缺页中断的次数。

可以用命令ps -o majflt minflt -C program来查看进程的majflt, minflt的值，这两个值都是累加值，从进程启动开始累加。在对高性能要求的程序做压力测试候，我们可以多关注一下这两个值。
如果一个进程使用了mmap将很大的数据文件映射到进程的虚拟地址空间，我们需要重点关注majflt的值，因为相比minflt，majflt对于性能的损害是致命的，读一次磁盘的耗时数量级在几个毫秒，而minflt只有在大量的时候才会对性能产生影响。

发成缺页中断后，执行了那些操作?

当一个进程发生缺页中断的时候，进程会陷入内核态，执行以下操作:
1、检查要访问的虚拟地址是否合法
2、查找/分配一个物理页
3、填充物理页内容（读取磁盘，或者直接置0，或者啥也不干）
4、建立映射关系（虚拟地址到物理地址）
5、重新执行发生缺页中断的那条指令

关于第1步，如果虚拟地址不合法，内核直接抛出11号信号量（segment fault），进程收到后11号信号量后会异常退出
关于第2步，没有足够的物理内存页能，根据OMM默认的策略此时会杀掉系统中其他的进程以获得物理页，分配给本进程。
关于第3步，如果读取磁盘，那么这次缺页中断就是 majflt，否则就是 minflt，但是通常高性能的环境中会将swap分区关闭，因为磁盘读写实在是太慢了，
关于第5步，缺页中断是在执行一条指令时产生的中断，并立即转去处理；一般的中断则是在一条指令执行完后，发现有中断请求时才去响应处理。因此，中断处理完成后，仍然返回到原指令去重新执行，因为那条指令并未执行。而一般中断则是返回到下一条执行去执行，因为这条指令已经执行完毕了。

内存分配的原理

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：**brk和mmap（不考虑共享内存）。**

1、brk是将数据段(.data)的最高地址指针 _edata 往高地址推;

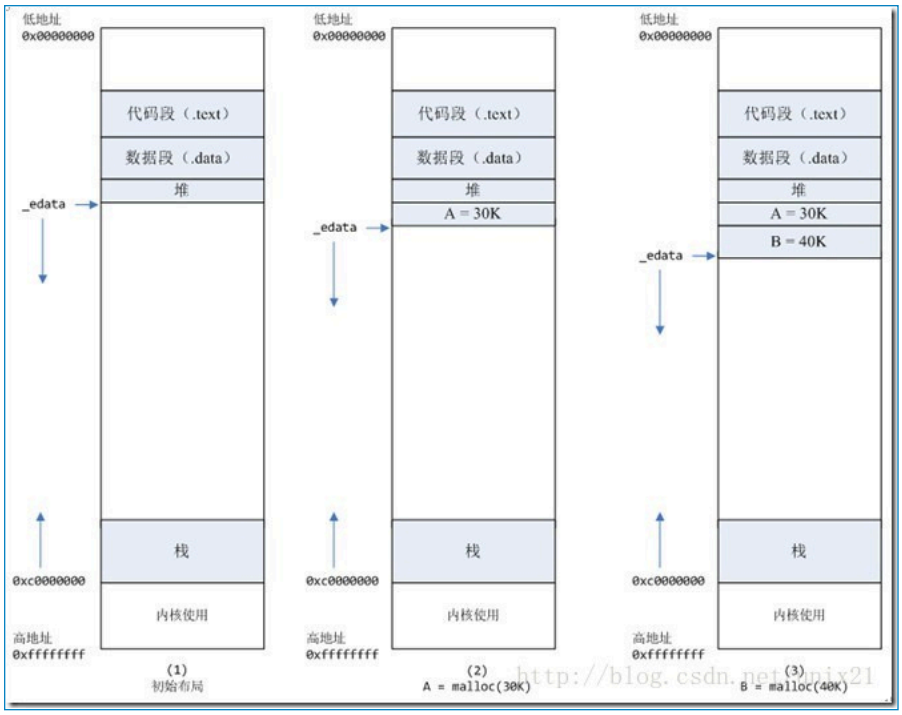
2、mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立内存和物理内存之间的映射关系。

在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk，mmap，munmap这些系统调用实现的。

下面以一个例子来说明内存分配的原理:

情况一、malloc小于128k的内存，使用brk分配内存，将_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)，如下图:



1、进程启动的时候，其（虚拟）内存空间的初始布局如图1所示。

其中，**mmap内存映射文件是在堆和栈的中间**（例如libc-2.2.93.so，其它数据文件等），为了简单起见，省略了内存映射文件。

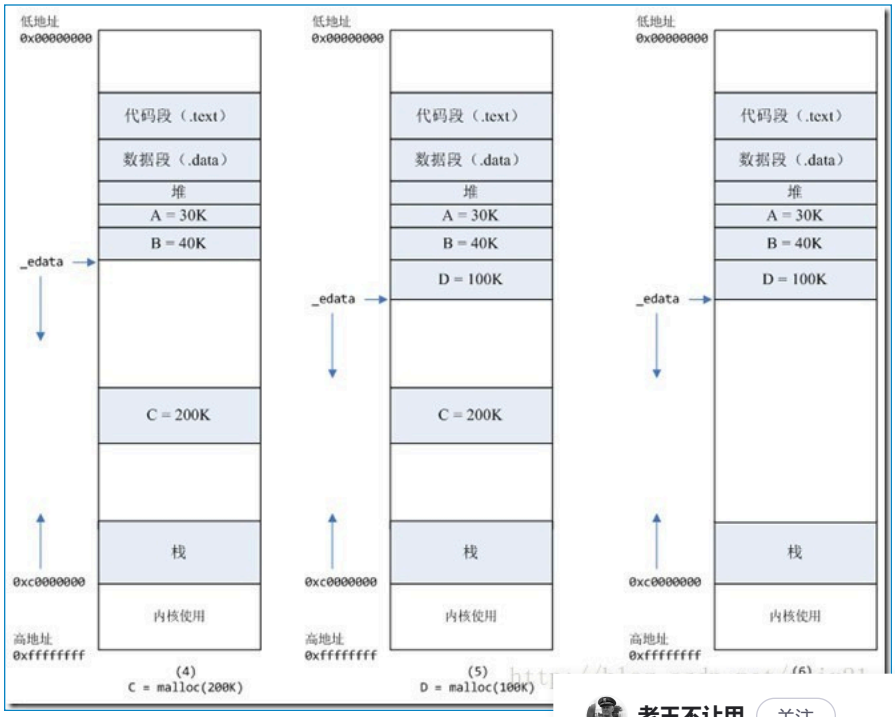
_edata指针（glibc里面定义）指向数据段的最高地址。
2、进程调用A=malloc(30K)以后，内存空间如图2：

malloc函数会调用brk系统调用，将_edata指针往高地址推30K，就完成虚拟内存　分配。

你可能会问：只要把_edata+30K就完成内存分配了？

事实是这样的，_edata+30K只是完成虚拟地址的分配，A这块内存现在还是没有物理页与之对应的，等到进程第一次读写A这块内存的时候，发生缺页中断这个时候，内核才分配A这块内存对应的物理页。**也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。**
3、进程调用B=malloc(40K)以后，内存空间如图3。

情况二、malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)，如下图：



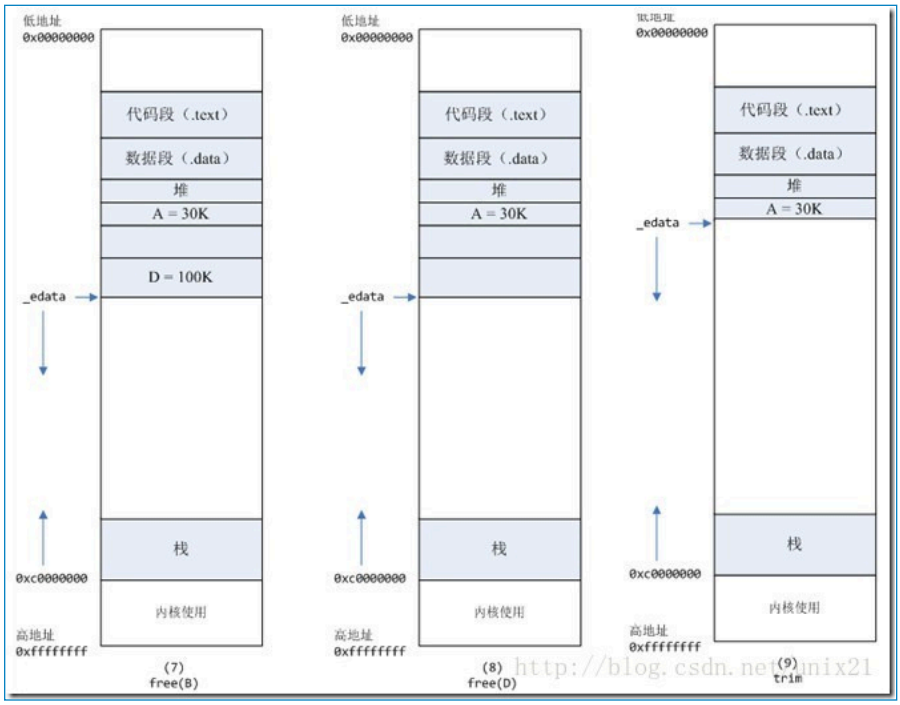
4、进程调用C=malloc(200K)以后，内存空间如图4：

默认情况下，**malloc函数分配内存，如果请求内存大于128K（可由M_MMAP_THRESHOLD选项调节），那就不是去推_edata指针了，而是利用mmap**
用，从堆和栈的中间分配一块虚拟内存。

这样子做主要是因为::

brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候紧缩看下面），
mmap分配的内存可以单独释放。

当然，还有其它的好处，也有坏处，再具体下去，有兴趣的同学可以去看glibc里面malloc的代码了。
5、进程调用D=malloc(100K)以后，内存空间如图5；
6、进程调用free(C)以后，C对应的虚拟内存和物理内存一起释放。



7、进程调用free(B)以后，如图7所示：

B对应的虚拟内存和物理内存都没有释放，因为只有一个_edata指针，如果往回推，那么D这块内存怎么办呢?

当然，B这块内存，是可以重用的，如果这个时候再来一个40K的请求，那么malloc很可能就把B这块内存返回回去了。
8、进程调用free(D)以后，如图8所示：

B和D连接起来，变成一块140K的空闲内存。

9、默认情况下：

当最高地址空间的空闲内存超过128K（可由M_TRIM_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址
内存超过128K，于是内存紧缩，变成图9所示。

真相大白：
说完内存分配的原理，那么被测模块在内核态cpu　消耗高的原因就很清楚了：每次请求来都malloc一块2M的内存，默认情况下，malloc调用 mmap分配
请求结束的时候，调用munmap释放内存。假设每个请求需要6个物理页，那么每个请求就会产生6个缺页中断，在2000的压力下，每 秒就产生了10000多
中断，这些缺页中断不需要读取磁盘解决，所以叫做minflt；缺页中断在内核态执行，因此进程的内核态cpu消耗很大。缺 页中断分散在整个请求的处理过
所以表现为分配语句耗时（10us）相对于整条请求的处理时间（1000us）比重很小。
解决办法：
将动态内存改为静态分配，或者启动的时候，用malloc为每个线程分配，然后保存在threaddata里面。但是，由于这个模块的特殊性，静态分配，或者启动
分配都不可行。另外，Linux下默认栈的大小限制是10M，如果在栈上分配几M的内存，有风险。
禁止malloc调用mmap分配内存，禁止内存紧缩。
在进程启动时候，加入以下两行代码：

cpp　　AI写代码　　复制

```cpp
mallopt(M_MMAP_MAX, 0); // 禁止malloc调用mmap分配
```

```
mallopt(M_TRIM_THRESHOLD, -1); // 禁止内存紧缩
```

效果：加入这两行代码以后，用ps命令观察，压力稳定以后，majlt和minflt都为0。进程的系统态cpu从20降到10。

3、如何查看堆内内存的碎片情况？

glibc 提供了以下结构和接口来查看堆内内存和 mmap 的使用情况。

cpp AI写代码 复制

```cpp
struct mallinfo {
  int arena;            /* non-mmapped space allocated from system */
  int ordblks;          /* number of free chunks */
  int smblks;           /* number of fastbin blocks */
  int hblks;            /* number of mmapped regions */
  int hblkhd;           /* space in mmapped regions */
  int usmblks;          /* maximum total allocated space */
  int fsmblks;          /* space available in freed fastbin blocks */
  int uordblks;         /* total allocated space */
  int fordblks;         /* total free space */
  int keepcost;         /* top-most, releasable (via malloc_trim) space */
};
```

收起 ^

/*返回heap(main_arena)的内存使用情况，以 mallinfo 结构返回 */
struct mallinfo mallinfo();

/* 将heap和mmap的使用情况输出到stderr*/
void malloc_stats();**可通过以下例子来验证mallinfo和malloc_stats输出结果。**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <malloc.h>

size_t  heap_malloc_total, heap_free_total,mmap_total, mmap_count;

void print_info()
{
    struct mallinfo mi = mallinfo();
printf("count by itself:\n");
    printf("\theap_malloc_total=%lu heap_free_total=%lu heap_in_use=%lu\n\tmmap_total=%lu mmap_count=%lu\n",
              heap_malloc_total*1024, heap_free_total*1024, heap_malloc_total*1024-heap_free_total*1024,
              mmap_total*1024, mmap_count);
printf("count by mallinfo:\n");
printf("\theap_malloc_total=%lu heap_free_total=%lu heap_in_use=%lu\n\tmmap_total=%lu mmap_count=%lu\n",
             mi.arena, mi.fordblks, mi.uordblks,
             mi.hblkhd, mi.hblks);
printf("from malloc_stats:\n");
malloc_stats();
}

#define ARRAY_SIZE 200
int main(int argc, char** argv)
{
    char** ptr_arr[ARRAY_SIZE];
    int i;
    for( i = 0; i < ARRAY_SIZE; i++)
    {
            ptr_arr[i] = malloc(i * 1024);
```

```
            if ( i < 128)                                   //glibc默认128k以上使用mmap
                    heap_malloc_total += i;
            }
            else
            {
                    mmap_total += i;
                   mmap_count++;
            }
    }
    print_info();

    for( i = 0; i < ARRAY_SIZE; i++)
    {
           if ( i % 2 == 0)
                continue;
           free(ptr_arr[i]);

           if ( i < 128)
           {
                   heap_free_total += i;
           }
           else
           {
                  mmap_total -= i;
                  mmap_count--;
           }
    }
    printf("\nafter free\n");
    print_info();

    return 1;
}
```

收起 ^

该例子第一个循环为指针数组每个成员分配索引位置 (KB) 大小的内存块，并通过 128 为分界分别对 heap 和 mmap 内存分配情况进行计数；
第二个循环是 free 索引下标为奇数的项，同时更新计数情况。通过程序的计数与mallinfo/malloc_stats 接口得到结果进行对比，并通过 print_info 打印到终

下面是一个执行结果：

```
count by itself:
    heap_malloc_total=8323072 heap_free_total=0 heap_in_use=8323072
    mmap_total=12054528 mmap_count=72
count by mallinfo:
    heap_malloc_total=8327168 heap_free_total=2032 heap_in_use=8325136
    mmap_total=12238848 mmap_count=72

from malloc_stats:
Arena 0:
system bytes     =    8327168
in use bytes     =    8325136
Total (incl. mmap):
system bytes     =   20566016
in use bytes     =   20563984
max mmap regions =         72
max mmap bytes   =   12238848

after free
count by itself:
    heap_malloc_total=8323072 heap_free_total=4194304 heap_in_use=4128768
    mmap_total=6008832 mmap_count=36

count by mallinfo:
    heap_malloc_total=8327168 heap_free_total=419736
    mmap_total=6119424 mmap_count=36
```

```
from malloc_stats:
Arena 0:
system bytes     =    8327168
in use bytes     =    4129808
Total (incl. mmap):
system bytes     =   14446592
in use bytes     =   10249232
max mmap regions =         72
max mmap bytes   =   12238848
```

由上可知，程序统计和mallinfo 得到的信息基本吻合，其中 heap_free_total 表示堆内已释放的内存碎片总和。
如果想知道堆内究竟有多少碎片，可通过 mallinfo 结构中的 fsmblks 、smblks 、ordblks 值得到，这些值表示不同大小区间的碎片总个数，这些区间分 0~80 字节，80~512 字节，512~128k。如果 fsmblks 、 smblks 的值过大，那碎片问题可能比较严重了。不过， mallinfo 结构有一个很致命的问题，就是定义全部都是 int ，在 64 位环境中，其结构中的 uordblks/fordblks/arena/usmblks 很容易就会导致溢出，应该是历史遗留问题，使用时要注意！！！

4、既然堆内内存brk和sbrk不能直接释放，为什么不全部使用 mmap 来分配，munmap直接释放呢?

既然堆内碎片不能直接释放，导致疑似“内存泄露”问题，为什么 malloc 不全部使用 mmap 来实现呢(mmap分配的内存可以会通过 munmap 进行 free ，实释放)？而是仅仅对于大于 128k 的大块内存才使用 mmap ？

其实，进程向 OS 申请和释放地址空间的接口 sbrk/mmap/munmap 都是系统调用，频繁调用系统调用都比较消耗系统资源的。并且， mmap 申请的内存 munmap 后，重新申请会产生更多的缺页中断。例如使用 mmap 分配 1M 空间，第一次调用产生了大量缺页中断 (1M/4K 次) ，当munmap 后再次分配 1 间，会再次产生大量缺页中断。**缺页中断是内核行为，会导致内核态CPU消耗较大。**另外，如果使用 mmap 分配小内存，会导致地址空间的分片更多，内 理负担更大。
同时堆是一个连续空间，并且堆内碎片由于没有归还 OS ，如果可重用碎片，再次访问该内存很可能不需产生任何系统调用和缺页中断，这将大大降低 C 消耗。 因此， glibc 的 malloc 实现中，充分考虑了 sbrk 和 mmap 行为上的差异及优缺点，默认分配大块内存 (128k) 才使用 mmap 获得地址空间，也可 mallopt(M_MMAP_THRESHOLD, <SIZE>) 来修改这个临界值。

5、如何查看进程的缺页中断信息?

可通过以下命令查看缺页中断信息
ps -o majflt,minflt -C <program_name>
ps -o majflt,minflt -p <pid>
其中:: majflt 代表 major fault ， 指大错误;

minflt 代表 minor fault ， 指小错误。

这两个数值表示一个进程自启动以来所发生的缺页中断的次数。
其中 majflt 与 minflt 的不同是::

majflt 表示需要读写磁盘，可能是内存对应页面在磁盘中需要load 到物理内存中，也可能是此时物理内存不足，需要淘汰部分物理页面至磁盘中。

参看:: http://blog.163.com/xychenbaihu@yeah/blog/static/132229655201210975312473/

6、除了 glibc 的 malloc/free ， 还有其他第三方实现吗?

其实，很多人开始诟病 glibc 内存管理的实现，特别是高并发性能低下和内存碎片化问题都比较严重，因此，陆续出现一些第三方工具来替换 glibc 的实现 名的当属 google 的tcmalloc和facebook 的jemalloc 。
网上有很多资源，可以自己查(只用使用第三方库，代码不用修改，就可以使用第三方库中的malloc)。

参考资料:
《深入理解计算机系统》第 10 章
http://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

https://www.ibm.com/developerworks/cn/linux/l-lvm64/

http://www.kerneltravel.net/journal/v/mem.htm

http://blog.csdn.net/baiduforum/article/details/6126337

http://www.nosqlnotes.net/archives/105

http://www.man7.org/linux/man-pages/man3/mallinfo.3.html

原文地址：http://blog.163.com/xychenbaihu@yeah/blog/st

测试程序代码:

```
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>

static void
display_mallinfo(void)
{
    struct mallinfo mi;

   mi = mallinfo();

    printf("Total non-mmapped bytes (arena):       %d\n", mi.arena);
    printf("# of free chunks (ordblks):            %d\n", mi.ordblks);
    printf("# of free fastbin blocks (smblks):     %d\n", mi.smblks);
    printf("# of mapped regions (hblks):           %d\n", mi.hblks);
    printf("Bytes in mapped regions (hblkhd):      %d\n", mi.hblkhd);
    printf("Max. total allocated space (usmblks):  %d\n", mi.usmblks);
    printf("Free bytes held in fastbins (fsmblks): %d\n", mi.fsmblks);
    printf("Total allocated space (uordblks):      %d\n", mi.uordblks);
    printf("Total free space (fordblks):           %d\n", mi.fordblks);
    printf("Topmost releasable block (keepcost):   %d\n", mi.keepcost);
}

int
main(int argc, char *argv[])
{
#define MAX_ALLOCS 2000000
    char *alloc[MAX_ALLOCS];
    int numBlocks, j, freeBegin, freeEnd, freeStep;
    size_t blockSize;

   if (argc < 3 || strcmp(argv[1], "--help") == 0)
   {
        printf("%s num-blocks block-size [free-step [start-free "
                "[end-free]]]\n", argv[0]);
        return 0;
   }
    numBlocks = atoi(argv[1]);
    blockSize = atoi(argv[2]);
    freeStep = (argc > 3) ? atoi(argv[3]) : 1;
    freeBegin = (argc > 4) ? atoi(argv[4]) : 0;
    freeEnd = (argc > 5) ? atoi(argv[5]) : numBlocks;

   printf("============== Before allocating blocks ==============\n");
    display_mallinfo();

   for (j = 0; j < numBlocks; j++) {
        if (numBlocks >= MAX_ALLOCS)
            std::cout<<"Too many allocations"<<std::endl;

       alloc[j] = (char *)malloc(blockSize);
        if (alloc[j] == NULL)
            std::cout<<"malloc"<<std::endl;
    }

   printf("\n============== After allocating blocks ==============\n");
    display_mallinfo();

   for (j = freeBegin; j < freeEnd; j += freeStep)
   {
        free(alloc[j]);
   }
```

```
64      printf("\n============== After freeing blocks ==============\n");
65      display_mallinfo();
66
67      exit(EXIT_SUCCESS);
68  }
```

收起 ^

深入理解 malloc：ptmalloc 机制、堆布局与内核映射 love131452098的博客

本文系统介绍了glibc中ptmalloc内存管理机制的工作原理与内核协作过程。主要内容包括：malloc通过brk和mmap系统调用获取内存，采用arena、bins和chunk结构管理堆空间

mallinfo, 打印堆栈, malloc钩子, mtrace()

mallinfo, 打印堆栈, malloc钩子, mtrace()一：获得即时内存状态：void getMemStatus(){ struct mallinfo info = mallinfo (); printf("arena = %d/n", info.arena); printf("ordblks

24、【OS】【Nuttx】【OSTest】内存监控:用户堆成员_mallinfo...

当已使用内存里,有内存被释放时,此时产生内存碎片 fordblks',布局如下: 获取mallinfo 接口如下(这个返回类型和函数名用同一个名字也是容易看错),返回当前用户堆信息的一个 n

使用meminfo分析Android单个进程内存信息_unity3d dumpsys...

Native Heap Free: 从mallinfo fordblks获得,代表总共剩余空间 Native Heap Size 约等于Native Heap Alloc + Native Heap Free mallinfo是一个C库, mallinfo 函数提供了各种各样

linux环境内存分配原理--虚拟内存 mallocinfo cp3alai的专栏

Linux的虚拟内存管理有几个关键概念： Linux 虚拟地址空间如何分布？ malloc和free是如何分配和释放内存？ 如何查看堆内内存的碎片情况？ 既然堆内内存brk和sbrk不能直接释

IOS 崩溃使用 info malloc-history 来看堆栈 zltianhen的专栏

IOS 崩溃使用 info malloc-history 来看堆栈

glibc函数详解与手册

mallinfo和malloc_info函数用于内存分配统计信息。 mallinfo函数返回当前内存分配的状态信息。其原型为: c struct mallinfo mallinfo(void); 返回一个mallinfo结构体,包含了当前分

mallinfo函数解析

文章浏览阅读705次。pagckages/isoinfra/v3_0/include/stdlib.h中的mallinfo函数_ecos系统有没有查看内存的命令

malloc打印mallinfo信息定位内存增长的问题 onlyForCloud的专栏

背景 malloc 的基本原理，之前的几篇文章已经介绍过了。这里主要想说明一下根据mallinfo的信息来分析内存实际的问题。 malloc后立即free void test2() { print_info(); cout <<

Linux内存分配与回收

利用malloc和 calloc函数实现动态内存的...利用链表实现动态内存分配。 1、 了解静态内存与动态内存的区别； 2、 理解动态内存的分配和释放原理； 3、 掌握如何调整动态内

C++用valgrind排查内存泄露_valgrind检查c++内存泄漏

文章浏览阅读1k次。本文介绍了C/C++中内存泄漏的原因和危害,以及如何通过任务管理器、valgrind工具、mallinfo函数和系统调用strace来检测内存泄漏。valgrind是强大的内存

malloc 底层实现及Linux内存分配原理 weixin_43869778的博客

1） malloc 函数实在虚拟地址空间中划分一片区域，而没有与物理页对应。 1）当开辟的空间小于 128K 时，malloc 的底层实现是调用 brk()系统调用函数来在虚拟地址空间分配

Linux 多线程原理深剖 **热门推荐** qq_61500888的博客

带你一命通关 Linux 多线程原理 三十分钟手撕底层内涵

Linux Slub Allocator原理 **最新发布**

Linux Slub Allocator工作机制剖析#### 内存分配层次结构在进一步探讨Slub分配器的运作机制之前，我们首先需要简明回顾Linux内存管理中的内存分配层次结构。Linux内

【Linux】什么是虚拟内存? 2403_86785171的博客

Linux虚拟内存（1）什么是虚拟内存？（2）虚拟内存的工作原理（3）虚拟内存的优点（3）Linux中的虚拟内存管理工具总结虚拟内存是一种内存管理技术，它允许操作系统通

mallinfo - obtain memory allocation information qingsong1001的专栏

mallinfo - obtain memory allocation information SYNOPSIS #include <malloc.h> struct mallinfo mallinfo(void); DESCRIPTION The mallinfo() function returns a copy of ...

Linux虚拟内存介绍，以及malloc_stats和malloc_info 监控查看内存情况 zzhongcy的专栏

查找内存泄漏问题，可以使用valgrind、malloc_stats和malloc_info 监控查看内存情况。 1、 Linux内存介绍 1.1 Linux 的虚拟内存管理有几个关键概念： 1、每个进程都有独立的

32、【OS】【Nuttx】【OSTest】内存监控：mallinfo HIT_Weston的博客

已经分析了内存监控前，包括堆内存成员，已经堆内存申请与初始化，下面回到内存监控这个主题，继续看内存监控是如何获取内存信息的。

第7章 进程环境 cabbage2008的专栏

main函数 C程序总是从main函数开始执行。main函数的原型是： in

android 如何分析应用的内存（十）——malloc统计和libmemunreachable 安仔的博客
接下来介绍native heap内存的第四个板块————malloc统计和libmemunreachable。

Linux 分析进程动态内存需求的方法 maimang1001的专栏
1、使用 mallinfo 接口： static void display_mallinfo(void) { struct mallinfo mi; mi = mallinfo(); printf("Total non-mmapped bytes (arena): %d\n", mi.arena); printf("# of free chunk

valgrind排查内存泄露 qianfeng_dashuju的博客
前言 C/C++运行高效，不管是操作系统内核还是对性有要求的程序（比如游戏引擎）都要求使用C/C++来编写，其实C/C++强大的一点在于能够使用指针自由地控制内存的使用

内存泄露调试经验 whuzm08的专栏
使用mallinfo确定是否有内存泄露：static struct mallinfo mi1,mi2;static struct timeval oldTime, currentTime;#define MALLOC_STAT_TIME (60)void sk_factory_test_malloc_sta

老王不让用
博客等级 码龄11年

93	888	5002	673
原创	点赞	收藏	粉丝

关注 私信

热门文章

什么是.so文件 76648
交换机的工作原理 62108
iptables命令详解和举例（完整版） 50989
mockcpp使用方法详细介绍附难点代码演练 29131
traceroute 命令使用方法详解 21544

分类专栏

linux内核协议栈 付费 99篇
socket编程 付费 18篇
tcp协议 41篇
网卡驱动 11篇

展开全部

上一篇：0-1背包相关题集
下一篇：iptables基础知识详解

大家在看

工业AMR任务建模与可审计管理 470
加密货币崩盘对区块链开发的影响：软件测试从业者的专业视角
AI人才短缺下的测试转型之路 12
AI元人文的反思与实践 456

老王不让用 关注
3

最新文章

linux内核协议栈之网卡驱动载波检测 netif_carrier_ok/netif_carrier_on/netif_carrier_off

route -n命令——显示系统的网络路由表

linux 内核协议栈发包流程（dev_queue_xmit & qos & NET_TX）

2024年 2篇	2023年 11篇
2022年 30篇	2021年 50篇
2020年 190篇	2019年 80篇
2015年 4篇	

目录

收起