动态内存管理调研

张昱 (yuzhang@ustc.edu.cn)

University of Science and Technology of China

November 20, 2018

目录

Part I

内存统计分析工具

1 内存统计分析工具概述

1.1 进程的 smaps 文件

smaps 文件描述了进程在运行时，各个部分映射到物理内存的实际情况解释一下 smaps 中 8 个字段的含义分别如下：

- Size：表示该映射区域在虚拟内存空间中的大小。
- Rss：表示该映射区域当前在物理内存中占用了多少空间
- Shared_Clean：和其他进程共享的未被改写的 page 的大小
- Shared_Dirty：和其他进程共享的被改写的 page 的大小
- Private_Clean：未被改写的私有页面的大小。
- Private_Dirty：已被改写的私有页面的大小。
- Swap：表示非 mmap 内存（也叫 anonymous memory，比如 malloc 动态分配出来的内存）由于物理内存不足被 swap 到交换空间的大小。
- Pss：该虚拟内存区域平摊计算后使用的物理内存大小 (有些内存会和其他进程共享，例如 mmap 进来的)。比如该区域所映射的物理内存部分同时也被另一个进程映射了，且该部 > 分物理内存的大小为 1000KB，那么该进程分摊其中一半的内存，即 Pss=500KB。

RSS：The resident set size，进程运行时占用物理内存的大小，但是不包含共享页的信息。PSS：proportional set size，进程在运行时所占内存的大小，相对与 RSS，PSS 考虑了共享内存。

PSS 中对于共享内存的处理：假设一个进程自己有 1000 私有页，和另外一个进程共享 1000 页 >，那么得到的 PSS 会是 1500 页。USS：进程私有内存大小。（The "proportional set size" (PSS) of a process is the count of pages it has in memory, where each page is divided by the number of processes sharing it. So if a process has 1000 pages all to itself, and 1000 shared with one other process, its PSS will be 1500. The unique set size (USS), instead, is a simple count of unshared pages. It is, for all practical purposes, the number of pages which will be returned to the system if the process is killed.) 这部分详细的介绍可参考ELC: How much memory are applications really using? 在论文 [2] 也是用了 PSS 作为内存消耗的数值。

1.2 工具

- smem：是对内存整体使用情况检测工具。可以显示当前系统各个进程占用内存情况，各个进程的百分比，给出的表格和饼图比较直观。可参考smem memory reporting tool.
- memstats：从/proc 中收集数据。memstats 主页
- memrank：也是从 proc/目录下得到个进程的 PSS 值，进行排序。源码
- malloc_count (源码)：提供一组测量程序运行时所分配的内存数量的源代码工具，包括测量堆内存分配的当前值和峰值、写可用 GNUPlot 绘图的内存 profile、独立的用于测量栈使用的 stack_count。
- Massif：是 Valgrind 提供的一个堆 Profiler。这个工具很慢。

1.3 GNUPlot 绘图工具

gnuplot(www.gnuplot.info) 是可移植的命令行驱动的绘图工具，支持 Linux、OS/2、Windows、OSX、VMS 等许多平台。

`sudo apt-get install gnuplot`

在 Ubuntu12.04 下，执行 `sudo apt-get install plotutils` 得到的 plot 版本是 2.6。

2 malloc_count 及其变种

malloc_count 提供一组测量程序运行时所分配的内存数量的源代码工具，包括测量堆内存分配的当前值和峰值、写用于绘图的内存 profile、独立的用于测量栈使用的 stack count。

参见：http://panthema.net/2013/malloc_count/。源码可从下面获取：

`git clone` https://github.com/bingmann/malloc_count

[1] 的作者对 malloc_count 进行了扩展，使得它能测量串行、并行和事务区间的内存分配，详见 § 2.2。

2.1 malloc_count

功能简介　该工具通过拦截标准堆分配函数 malloc, free, realloc, calloc 等，对每个调用增加简单的计数统计，然后调用标准的堆分配函数进行分配或回收。该工具可以用于 C/C++ 程序，甚至可以用于像 Python 和 Perl 这样的脚本语言，因为 new 操作和大多数脚本解释器的分配都是基于 malloc 的。工具可以用于 Linux，也可能用于 Cygwin 和 MinGW，因为它们也支持标准 Linux 的动态链接加载机制。

`stack_count` 提供两个简单的函数，能测量程序中两点之间的最大栈使用。使用 memprofile.h，可以产生统计文件，可直接用GNUPlot绘图。

`malloc_alloc.c` 中的函数不是线程安全的。其中不安全的部分是对计数器的增值和减值`inc_count, dec_cou`
`malloc_alloc.c` 代码中若包含`#define THREAD_SAFE_GCC_INTRINSICS`，则可以使用 GCC 对原子计数操作的内部函数。如果使用 gcc，定义该宏可以使`malloc_count` 是线程安全的。`memprofile.h` 中的函数也不是线程安全的。

使用方法　编译`malloc_count.c` 并将它与你的程序链接，必须将`-ldl`（实现支持在运行时链接的 4 个函数 dlopen, dlclose, dlsym, dlerror）加入到要链接的库列表中。

内部实现技术　为了跟踪每个已分配内存区域的大小，它为每个分配指针预置两个簿记变量：分配的大小 size 和标记 (sentinel) 值。从而，当分配 n 字节时，实际请求 libc 的 malloc() 分配 n+c 字节以保存大小，c 缺省为 16，可以被调整以解决对齐问题。标记只用于检查你的程序没有覆盖 size 信息。

2.1.1 malloc_count 的使用和实现机制

```
#include "malloc_count.h"
...
int main() {
  ...
  ...malloc(...);
  malloc_count_print_status();
  ...
}
```

实现机制

1. 在 malloc_count.c 中定义了声明有`__attribute__((constructor))` 属性的函数 init()，该函数将在执行进入 main() 时被调用。该函数三次调用`dlsym(RTLD_NEXT, A)` （A 依次为"malloc", "realloc", "free"），动态查找并获取给定函数名称的函数地址，依次将这些地址赋值给全局变量`real_malloc, real_realloc, real_free`。
2. 当应用程序调用 malloc 时，执行 malloc_count.c 中的 malloc(size)。其中若 size 不等于 0 且`real_malloc` 不为空时，则：
   - 调用`ret=(*real_malloc)(size+alignment)` 分配空间，全局常量 alignment 设置为 16(字节)；
   - 调用`inc_count(size)` 对分配的大小进行累计；
   - 在 ret 指向的内存块的头部依次保存 size 和标记 sentinel（为 0xDEADC0DEu），即：

```
1 *(size_t*)ret = size;
2 *(size_t*)((char*)ret + alignment - sizeof(size_t)) = sentinel;
```

   - 返回 (char *)ret+alignment 给应用程序中的 malloc 调用者。
3. 当应用程序调用 free 时，执行 malloc_count.c 中的 free(ptr)。其中若 ptr 不为 NULL，则：
   - 若 ptr 在 [init_heap, init_heap_use] 中，则打印所释放的指针在 init heap 中，并返回；
   - 若 real_free 为空，则报错，并返回；
   - ptr = (char *)ptr - alignment;
   - 检查 sentinel 标记是否存在，若不存在说明内存被破坏
   - 取得所释放的单元的`size = *(siz_t *)ptr;`
   - 调用`dec_count(size)` 从累计值中减去释放的大小；
   - 调用`(*real_free)(ptr);` 释放内存块；
4. `inc_count(),dec_count()` 中维护的统计量：
   - curr(多线程下的 mycurr)：当前使用的动态内存总量
   - peak：历史使用的动态内存峰值
   - total：累计的动态内存总分配量
   - num_allocs：返回总的分配次数

几个可以被应用程序调用的函数.

- `size_t malloc_count_current(void)`：返回当前的 curr 值
- `size_t malloc_count_peak(void)`：返回当前的 peak 值
- `void malloc_count_reset_peak(void)`：将 peak 置为当前的 curr 值
- `size_t malloc_count_num_allocs(void)`：返回当前的 num_allocs 值
- `void malloc_count_print_status(void)`：打印 curr 和 peak 的值
- `void malloc_count_set_callback(malloc_count_callback_type` cb, void* cookie)：可以设置用户自己的内存 profile 回调函数。回调函数的类型定义为：
  typedef void (*`malloc_count_callback_type`)(void* cookie, size_t current);

2.1.2 stack_count 的使用

```
1 #include "stack_count.h"
```

```
...
void* base = stack_count_clear();
...
printf("maximum stack usage: %lld\n",
       (long long)stack_count_usage(base));
```

base = stack_count_clear() 的实现机制. Linux 缺省的栈大小为 8MB，该函数调用时会在当前栈上分配 6MB 的局部数组，然后将该数组的每个 32 位元素初始化为 0xDEADC0DEu，同时返回数组的起始地址。简单来说，该函数将栈增长的空间初始化为特定值，以便日后知道栈的最大使用情况。

stack_count_usage(base) 的实现机制. base 是进行栈使用统计开始时的起址，从最初设置 0xDEADC0DEu 的栈的高端 (低地址端) 开始向栈底方向检测，直至遇到非 0xDEADC0DEu 的单元，base 与该单元地址的差值反映栈的最大使用值。

2.1.3 memprofile 的使用

```
#include "memprofile.h"
...
int main() {
    //参数: 要写入的日志文件, 时间分辨率, 尺寸分辨率
    MemProfile mp("memprofile.txt", 0.1, 1024);
    ...
}
```

实现机制. 在 memprofile.h 中定义了类 MemProfile，其中包含以下数据成员：

- `double m_time_resolution`: 时间分辨率，单位 s
- `size_t m_size_resolution`: 内存分辨率，单位字节
- `const char* m_funcname`: 针对多函数输出的函数名
  [YuZhang20150825: 尚未看到使用该域的实例]
- `FILE* m_file`: 输出文件
- `double m_base_ts`: 当前内存 profile 的起始时间戳
- `size_t m_base_mem`: 当前内存 profile 的起始内存使用量
- `char* m_stack_base`: 当前内存 profile 的起始栈地址
- `double m_prev_ts`: 上次日志输出的时间戳
- `size_t m_prev_mem`: 上次日志输出的内存使用量
- `size_t m_max`: 上次日志输出的最大内存使用量

在类的构造器中，调用 `malloc_count_set_callback`(MemProfile::static_callback, this) 设置 malloc 调用时进行内存 profile 的回调函数。

所定义的回调函数会将当前使用的堆内存大小和栈内存大小（相对于 profile 的起始大小）之和保存到局部变量 mem 中，并使成员变量 m_max 保存自本次 profile 开始以来的历史最大值。在本次 malloc 调用的时间戳距上次输出的时间戳超过时间分辨率，或者本次的内存使用量与上次输出的内存使用量的差值绝对值超过内存分辨率时，输出（时间戳, 最大内存用量）二元组到文件中。

```
    /// callback invoked by malloc_count when heap usage changes.
```

```
    inline void callback(size_t memcurr)
    {
        size_t mem = (memcurr > m_base_mem) ? (memcurr - m_base_mem) : 0;

        if ((char*)&mem < m_stack_base) // add stack usage
            mem += m_stack_base - (char*)&mem;

        double ts = timestamp();
        if (m_max < mem) m_max = mem; // keep max usage to last output

        // check to output a pair
        if (ts - m_prev_ts > m_time_resolution ||
            absdiff(mem, m_prev_mem) > m_size_resolution )
        {
            output(ts, m_max);
            m_max = 0;
            m_prev_ts = ts;
            m_prev_mem = mem;
        }
    }
```

2.1.4 STX B+ 树中测量四种 C++ maps 的内存使用

该例子参见http://panthema.net/2013/0505-STX-B+Tree-Memory-Usage/。源代码可以通过git clone https://github.com/bingmann/stx-btree获取。

在 stx-btree/memprofile/main.cc 中的 main()6 次调用 `write_memprofile<TestClass>(filename)` 函数。该模板函数定义如下

```
template <typename TestClass>
void write_memprofile(const char* filename)
{
    // the memory profile test seriously messes up malloc(), so that massive
    // house-keeping is necessary after the structure is freed. Use just fork
    // instead.
    pid_t pid = fork();

    if (pid == 0)
    { // 子进程
        std::cout << "Writing␣memory␣profile␣" << filename << std::endl;
        {
            MemProfile mp(filename, 0.1, 16 * 1024);
            TestClass test(insertnum);  // initialize test structures

            double ts1 = timestamp();
            test.run(insertnum);        // run timed test procedure
            double ts2 = timestamp();
            std::cout << "done,␣time=" << (ts2 - ts1) << std::endl;
        }
        exit(0);
    }

    int status;
    wait(&status);
```

```
}
```

stx-btree/memprofile/memprofile.h 与 malloc_count 中的相应文件的内容基本相同。从该例中仍未看到通过 memprof.h 中 m_funcname 来支持多输出的使用案例。

2.2 malloc_profile

Baldassin 等针对malloc_count 进行扩展以支持对串行、并行、事务不同区域的内存分配测量。相关资源见:

http://lampiao.lsc.ic.unicamp.br/ baldas/artifact/ppopp15-artifact.html

https://github.com/baldas/tm-study-malloc

2.2.1 malloc_profile.[ch]

在 malloc_profile.h 中扩展了以下内容:

```
// 修改了回调函数的类型
typedef void (*malloc_count_callback_type)(void* cookie, size_t current, size_t mallocs, size_t
        frees, size_t mallocs_tx, size_t frees_tx);
// 增加了区域类型
typedef enum {SEQ_REGION, PAR_REGION, TX_REGION} region_type;
// 增加了区域的进入和退出
extern int malloc_enter_region(region_type reg);
extern void malloc_exit_current_region();
```

2.2.2 memprofile.[ch]

在 memprofile.h 中含有以下声明:

```
void memprofile_start(const char* filepath, double time_resolution,
                size_t size_resolution, const char* funcname);
void memprofile_insertnull();
void memprofile_end();
```

memprofile.c 是在 malloc_count 的 memprofile.h(§2.1.3) 的基础上修改，将 MemProfile 类的包装去除，即类中的数据和函数成员分别变成全局变量和函数。然后增加了上面三个函数的实现。

2.2.3 stamp 测试程序中对 memprofile 的使用

在 stamp/trunk/common/seq/tm.h 中

```
// 在MALLOC_COUNT定义时，以下代码起作用
# include "malloc_profile.h"
# define MAIN(argc, argv)              int main (int argc, char** argv) { memprofile_start("
        memprofile.dat", 100000.0, 4096, NULL);
# define MAIN_RETURN(val)              memprofile_end();  return val; }
# define GOTO_SIM()                    memprofile_insertnull(); malloc_enter_region(PAR_REGION)
# define GOTO_REAL()                   memprofile_insertnull(); malloc_enter_region(SEQ_REGION)

# define TM_MALLOC(size)               malloc_enter_region(TX_REGION) ? (void *)malloc(size):(
        void *)0; malloc_exit_current_region()
```

```
9  # define TM_FREE(ptr)                  malloc_enter_region(TX_REGION); free(ptr);
          malloc_exit_current_region()
10 # define TM_FREE2(ptr, size)            malloc_enter_region(TX_REGION); free(ptr);
          malloc_exit_current_region()
```

参考

[1] Alexandro Baldassin, Edson Borin, and Guido Araujo. Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 87–96. ACM, 2015.

[2] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. PREDATOR: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 3–14, New York, NY, USA, 2014. ACM.