**内存泄露与valgrind**

**qq: 2391387647**

**date: 2020-7-2**

**author: 邹卓恩**

**目录**

[**第一 章 内存 3**](#_Toc44517323)

[**1.1 内存的相关概念介绍 3**](#_Toc44517324)

[**1.2 内存分配 6**](#_Toc44517325)

[**1.3 内存释放 7**](#_Toc44517326)

[**1.4 内存使用情况分析 7**](#_Toc44517327)

[**1.5 进程内存分析 10**](#_Toc44517328)

[**1.6 内存泄露类型 10**](#_Toc44517329)

[**1.7 常见的内存错误 11**](#_Toc44517330)

[**1.8 常见的内存工具 11**](#_Toc44517331)

[**第二章 valgrind 13**](#_Toc44517332)

[**2.1 valgrind概述 13**](#_Toc44517333)

[**2.2 memcheck 内存错误分析 14**](#_Toc44517334)

[**2.3 massif 堆内存分析 23**](#_Toc44517335)

[**2.4 helgrind 线程分析 25**](#_Toc44517336)

[**2.5 cachegrind cache分析 26**](#_Toc44517337)

# 第一 章 内存

## 内存的相关概念介绍

并不是所有的虚拟内存都会分配物理内存，只有那些实际使用的虚拟内存才分配物理内存，并且分配后的物理内存，是通过内存映射来管理的。

1. MMU

内存管理单元，完成虚拟地址与物理地址之间的映射。

1. TLB

CPU访问内存页表还是不够快，加了TLB，用来缓存页表，提高物理内存访问效率。

1. 页表

记录虚拟地址与物理地址的映射关系。

1. 页

内存映射的最小单位，也就是页，通常是 4 KB 大小。这样，每一次内存映射，都需要关联 4 KB 或者 4KB 整数倍的内存空间。

1. 多级页表

页的大小只有 4 KB ，导致的另一个问题就是，整个页表会变得非常大。比方说，仅 32 位系统就需要 100 多万个页表项（4GB/4KB），才可以实现整个地址空间的映射。

多级页表就是把内存分成区块来管理，将原来的映射关系改成区块索引和区块内的偏移。由于虚拟内存空间通常只用了很少一部分，那么，多级页表就只保存这些使用中的区块，这样就可以大大地减少页表的项数。

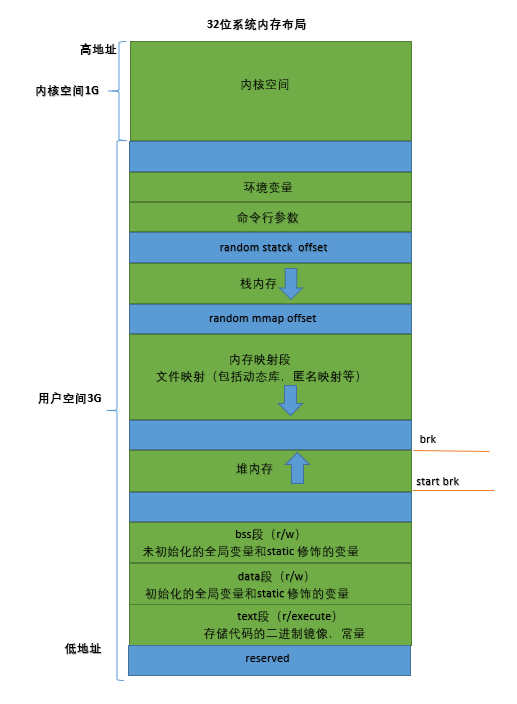
1. 缺页异常

当进程访问的虚拟地址在页表中查不到时，系统会产生一个缺页异常，进入内核空间分配物理内存、更新进程页表，最后再返回用户空间，恢复进程的运行。

1. 大页

比普通页更大的内存块，常见的大小有 2MB 和 1GB。大页通常用在使用大量内存的进程上，比如 DPDK 等。

1. 虚拟内存空间分布



* **栈：**

包括局部变量和函数调用的上下文等。栈的大小是固定的，一般是 8 MB,地址是从高地址开始增长的。

* **文件映射段：**

包括动态库、共享内存等，从高地址开始向下增长。

* **堆：**

包括动态分配的内存，从低地址开始向上增长。

* **数据段：**

初始化的全局变量和static修饰的变量。

* **只读段:**

包括代码和常量等。

## 内存分配

* 小块内存（小于 128K）

C 标准库使用 brk() 来分配（malloc的底层是brk），也就是通过移动堆顶的位置来分配内存。这些内存释放后并不会立刻归还系统，而是被缓存起来，这样就可以重复使用。

* 大块内存（大于 128K）

内存映射 mmap() 来分配，也就是在文件映射段找一块空闲内存分配出去。

mmap() 方式分配的内存，会在释放时直接归还系统，所以每次 mmap 都会发生缺页异常。在内存工作繁忙时，频繁的内存分配会导致大量的缺页异常，使内核的管理负担增大。这也是 malloc 只对大块内存使用 mmap 的原因。

## 内存释放

对应的API为free,unmap。

* 回收缓存

比如使用 LRU（Least Recently Used）算法，回收最近使用最少的内存页面；

* 回收不常访问的内存

把不常用的内存通过交换分区直接写到磁盘中，换入与换出。

Swap 其实就是把一块磁盘空间当成内存来用。它可以把进程暂时不用的数据存储到磁盘中（这个过程称为换出），当进程访问这些内存时，再从磁盘读取这些数据到内存中（这个过程称为换入）。

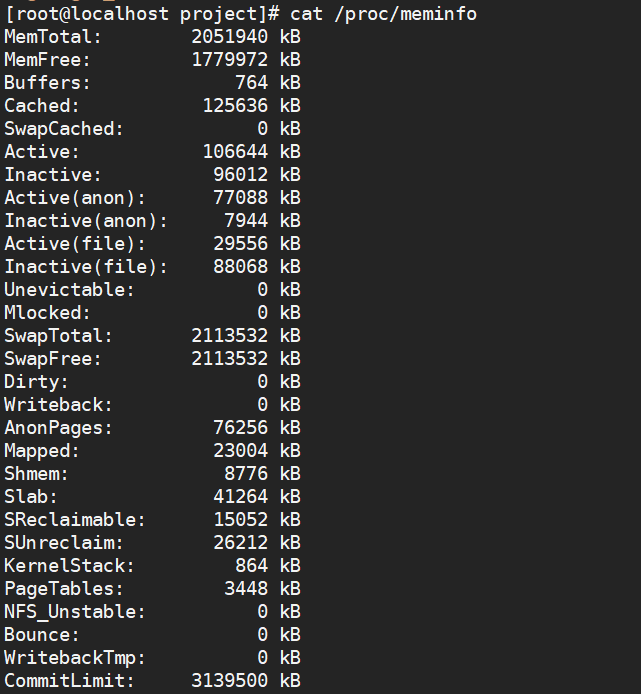
Swap 把系统的可用内存变大了。不过要注意，通常只在内存不足时，才会发生 Swap 交换。并且由于磁盘读写的速度远比内存慢，Swap 会导致严重的内存性能问题。

* 杀死进程

内存紧张时系统还会通过 OOM（Out of Memory），直接杀掉占用大量内存的进程。

## 内存使用情况分析

cat /proc/meminfo



系统上内存使用情况的统计数据。其中常见项数据含义如下表：

|  |  |
| --- | --- |
| 选项 | 含义 |
| MemTotal | 总内存大小 |
| MemFree | 未使用的内存（MemTotal- MemFree就是已经使用的内存） |
| Buffers | Buffer 既可以用作“将要写入磁盘数据的缓存”，也可以用作“从磁盘读取数据的缓存”。 |
| Cached | Cache 既可以用作“从文件读取数据的页缓存”，也可以用作“写文件的页缓存”。 |
| SwapCached | 交换缓存 ，包括换入和换出 |
| Active(anon) | 活跃的匿名内存 |
| Inactive(anon) | 非活跃的匿名内存 |
| Active(file) | 活跃的文件页缓存 |
| Inactive(file) | 非活跃的文件页缓存 |
| Mlocked | mlock()系统调用锁定的内存大小 |
| Dirty | Dirty + NFS\_Unstable + Writeback |
| Writeback | 正准备回写硬盘的缓存页 |
| AnonPages | 匿名页 |
| Mapped | shared libraries、可执行程序的文件、mmap的文件等 |
| Shmem | shared memory |
| KernelStack | 内核栈，给kernel code使用的 |
| PageTables | Page Table用于将内存的虚拟地址翻译成物理地址，随着内存地址分配得越来越多，Page Table会增大，/proc/meminfo中的PageTables统计了Page Table所占用的内存大小 |
| NFS\_Unstable | 发给NFS server但尚未写入硬盘的缓存页 |
| HugePages\_Total | 总共的大页内存大小 |
| HugePages\_Free | 可用的大页内存大小 |
| Hugepagesize | 大页的大小 |
| DirectMap4k | 表示映射为4kB的内存数量 |
| DirectMap2M | 映射为2MB的内存数量 |
| DirectMap1G | 表示映射为1GB的内存数量 |

可用的物理内存=MemFree+Buffers+Cached

echo 1 > /proc/sys/vm/drop\_caches # free pagecache

echo 2> /proc/sys/vm/drop\_caches # free dentries and inodes

echo 3> /proc/sys/vm/drop\_caches #free pagecache, dentries and inodes

cat /proc/sys/vm/min\_free\_kbytes #查看系统保留内存大小

## 进程内存分析

cat /proc/{pid}/maps，通过分析maps文件，分析进程当前运行时的stack,heap,mmap映射内存，是否持续增大，从而大概知道哪个进程发生了内存泄露。

## 内存泄露类型

1. 常发性内存泄漏，泄露部分代码每执行一次，就泄露一块内存。
2. 偶发性内存泄露，偶然发生内存泄露。
3. 一次性内存泄露，内存泄露的代码只会被执行一次，发生一次泄露。
4. 隐式内存泄露，分配了大量的内存，由于程序一直在运行，内存就一直占用，得不到释放。

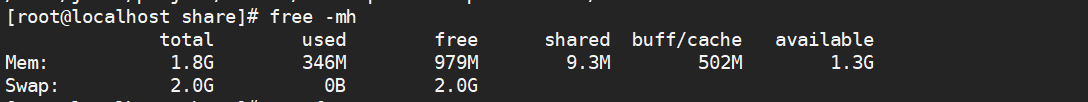
## 常见的内存错误

1. 使用未初始化的内存，局部变量和动态分配的内存，没有初始化就拿来使用。
2. 内存读写越界，数组越界操作。
3. 内存覆盖，strcpy拷贝字符串。
4. 动态内存管理错误，c++ malloc的内存，使用delete释放，释放了两次，如果一块内存释放了，操作系统会回收，被其它地方申请使用，如果这边再次释放，会导致其它地方产生不可预知的错误。

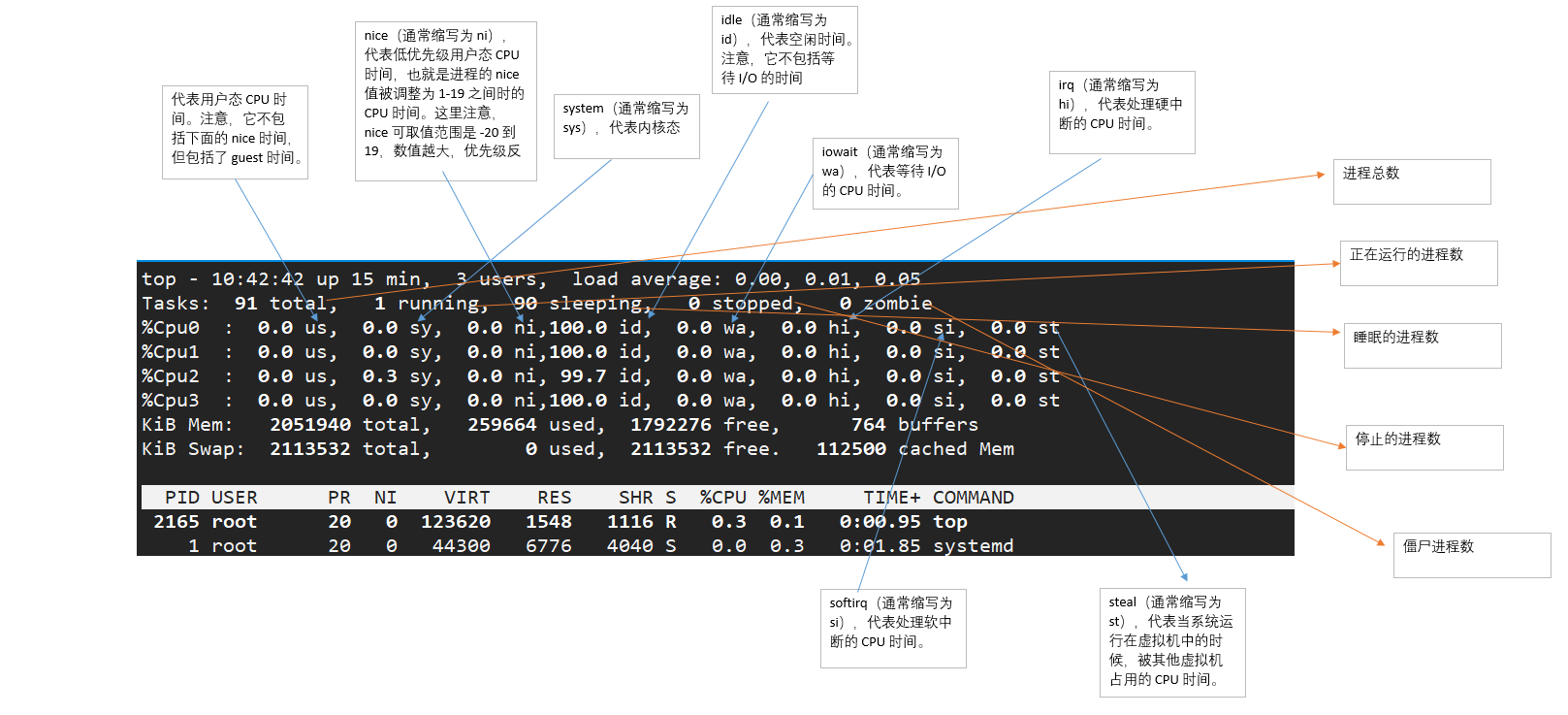
## 常见的内存工具

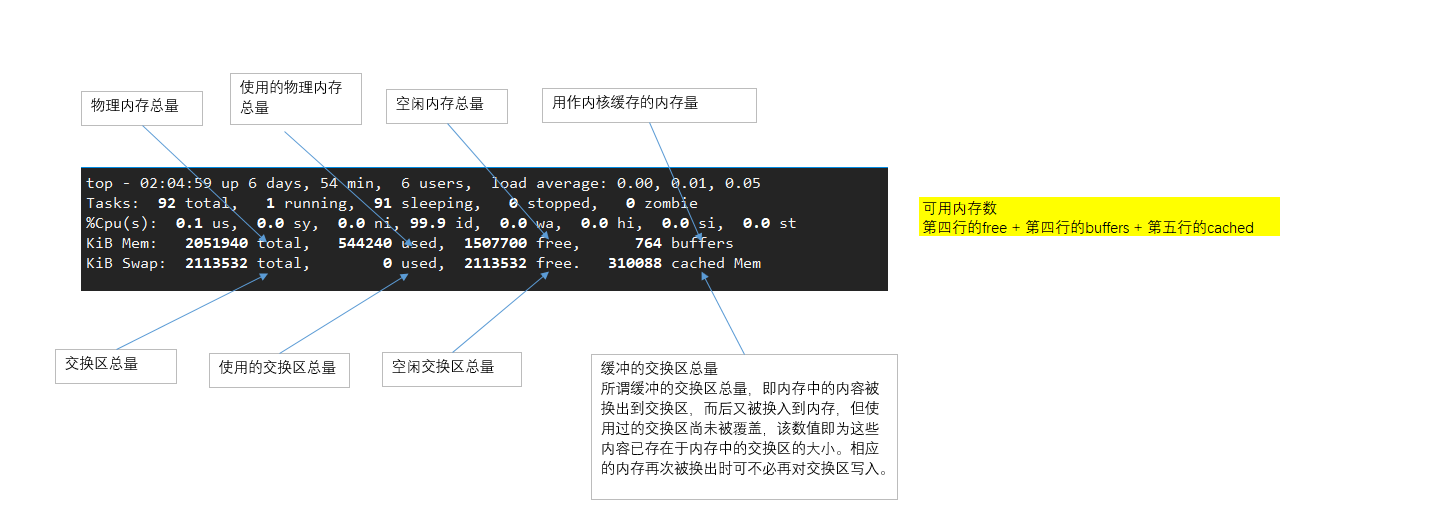
常用的工具有free/top/htop/sar/vmstat等

free #参数来源于/proc/meminfo，以下各参数解析参考1.4节内存使用情况分析



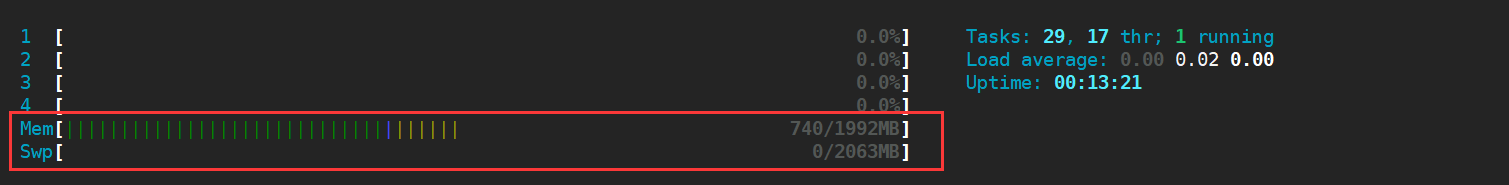
top



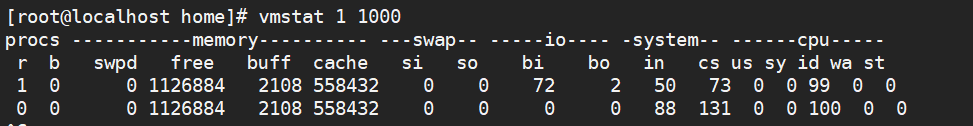




htop



vmstat 1 1000 #每1s运行一次，总共运行1000次



# 第二章 valgrind

## 2.1 valgrind概述

valgrind是一个运行在linux下的仿真工具集，其中包括以下常用工具。

* memcheck

内存检查器，能发生大部分的内存错误，比如未初始以化的内存，使用释放的内存，内存越界等。

* massif

堆分析器，提供了非常详细的堆内存分配信息。

* helgrind

检测多线程的错误问题，包括，线程API的使用错误，数据同步（访问共享资源没有加锁保护）等。

* cachegrind

模拟程序如何与机器的缓存层次结构和(可选的)分支交互预测。

## 2.2 memcheck 内存错误分析

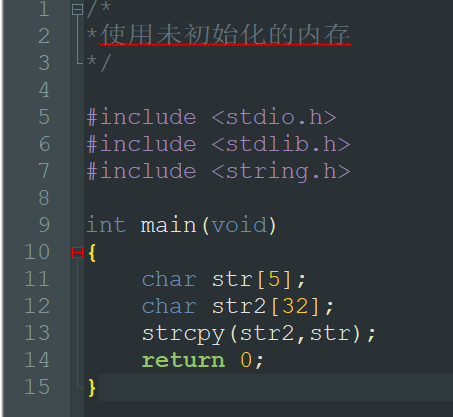
memcheck可以检测出，使用未初始化的内存，double free(双重释放)，内存越界，内存泄露等，下面结合这些内存错误来分析memcheck的用法。

1. 使用未初始化的内存

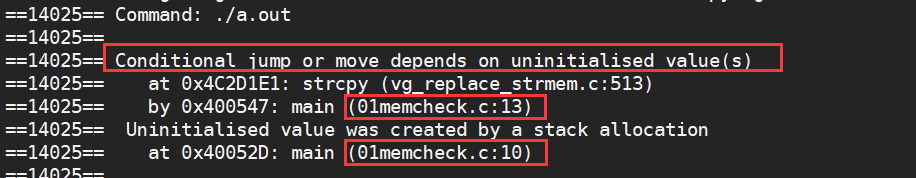
代码：01memcheck.c

编程程序并运行

gcc 01memcheck.c -g -O0



valgrind --tool=memcheck --track-origins=yes ./a.out

--track-origins # show origins of undefined values 

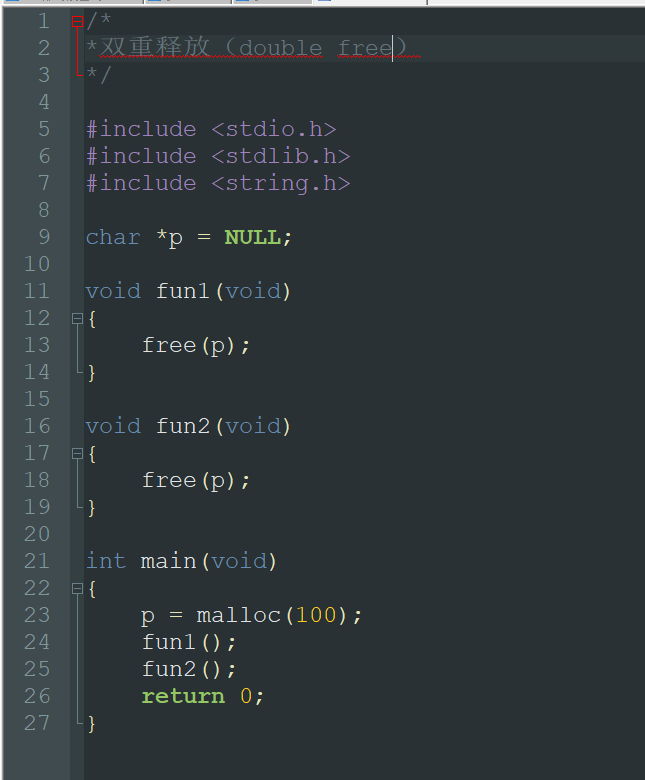
查看valgrind memcheck检测报告，可以看出，程序运行到每13行时，使用到了第10行代码的未初始化栈变量。

1. double free(双重释放堆内存)

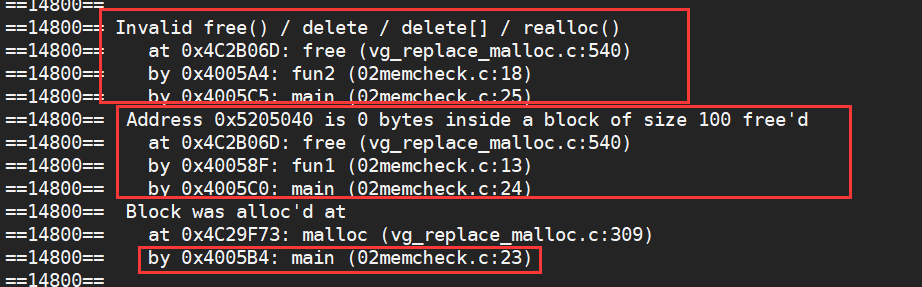
代码：02memcheck.c

编程程序并运行

gcc 02memcheck.c -g -O0



valgrind --tool=memcheck ./a.out



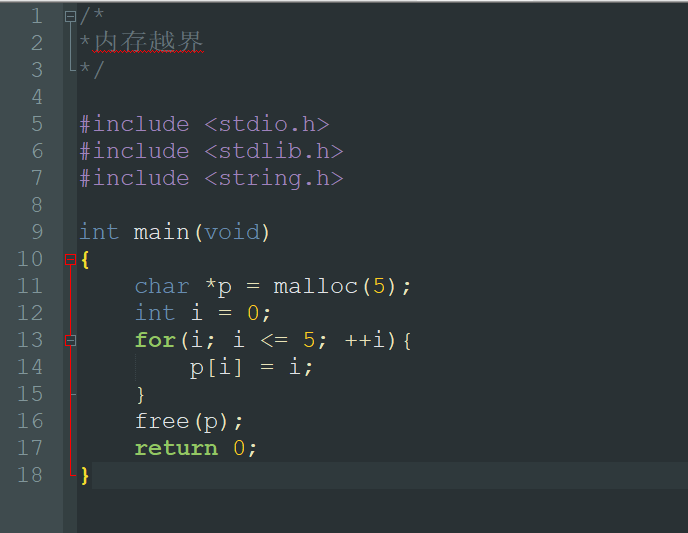
从报告可以看出，代码23行分析的堆内存，被释放了两次

1. 内存越界

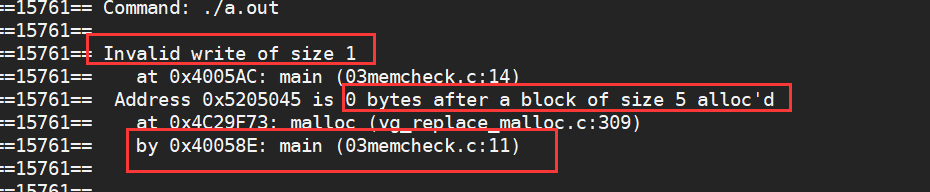
代码：03memcheck.c

编程程序并运行

gcc 03memcheck.c -g -O0



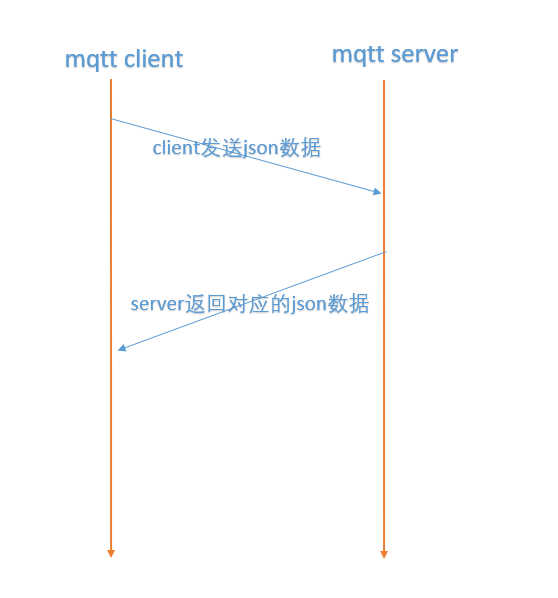
valgrind --tool=memcheck --track-origins=yes ./a.out



从程序运行报告得知，实际上代码11行只分配了5字节，但是在代码14行，有一字节的内存是无效访问。

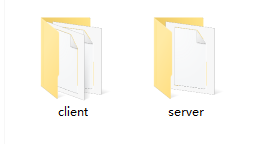
1. 内存泄露

上面的例子有些简单，而且问题很容易看出来，下面分析一下复杂的例子，检测内存泄露问题。

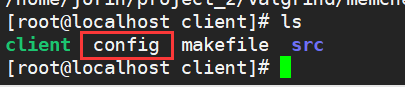


如图所示，client向server发送请求，server回复json数据，client解析json数据。

代码清单：

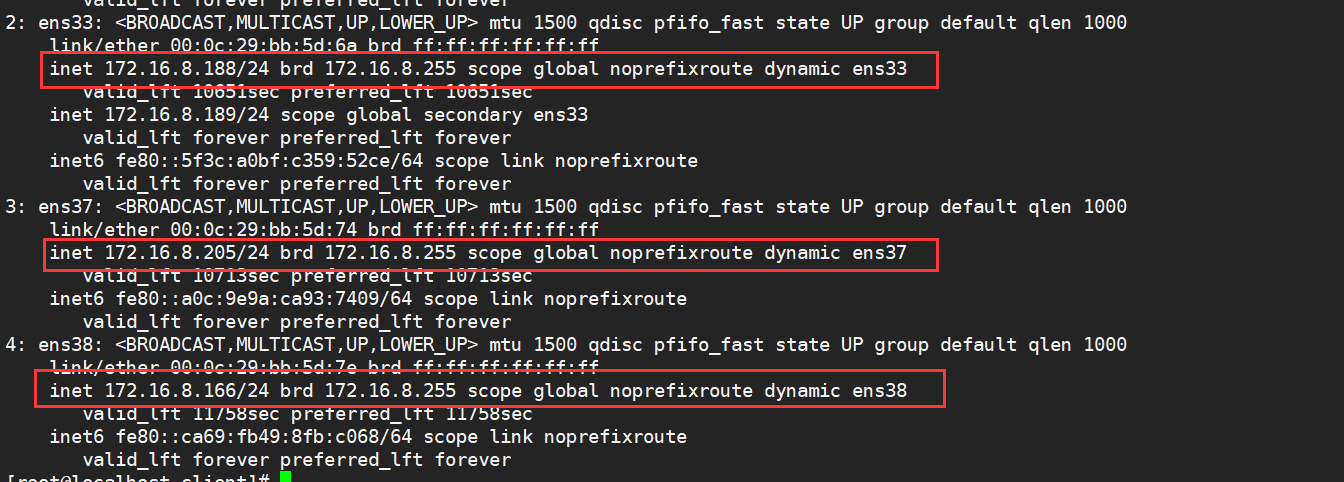


为了避免先入为主，先不要看代码，分别编译client和server并运行起来，其中client在运行之前，先在config中配置server的ip

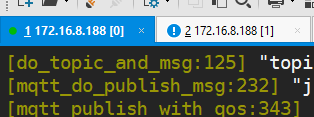




可配置成本地网卡任意一个ip

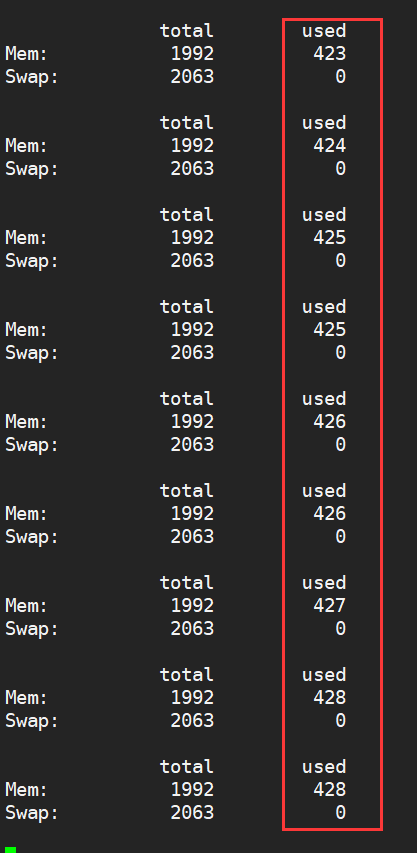


分别在xshell的两个窗口开两个窗口，分别启动server和client



执行free -m -c 10000 1 执行free命令，查看内存，总共运行10000次，每1s查看一次内存使用情况。

从图中可以看出，系统的使用内存正有规律地减少。



这个时候，valgrind memcheck用起来。

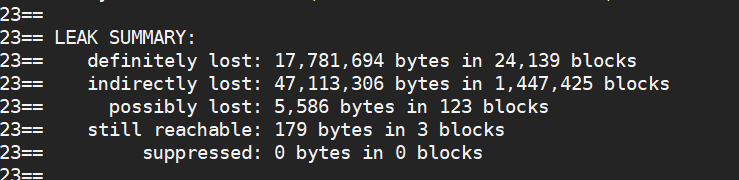
使用valgrind启动client。

--trace-children=yes #跟踪所有的父进程及其相关子进程

--show-leak-kinds=all --leak-check=full #检测所有泄露类型。

--log-file=/tmp/client.md #把检测报告记录到文件/tmp/client.md

等待几min后，停止client，查看/tmp/client.md文件。



先来分析一下，这个名词是什么意思。

definitely lost (17M左右)

确定的内存泄露

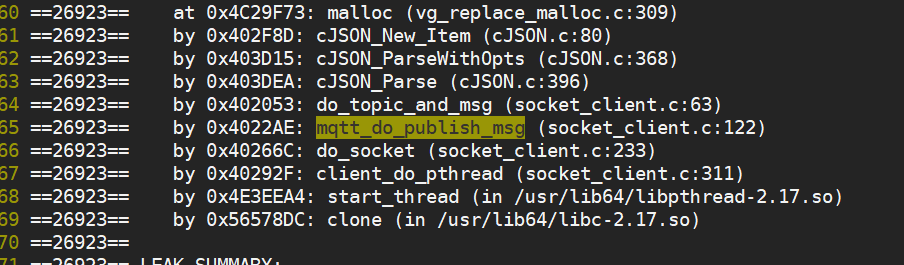
indirectly lost （47M左右）

间接的，没有任何指针指向该内存

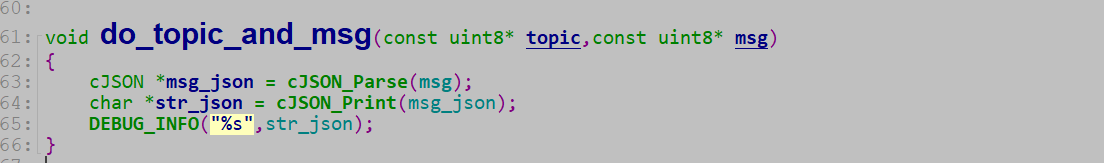
possibly lost （5KB左右）

可能的内存泄露，指某个指针访问某块内存，但这块内存已经不是这块首地址了。

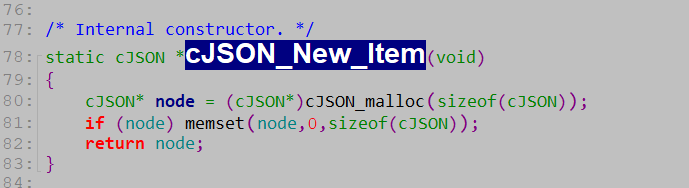
查看内存分配调用栈，可以大概知道分析堆内存的函数调用栈。

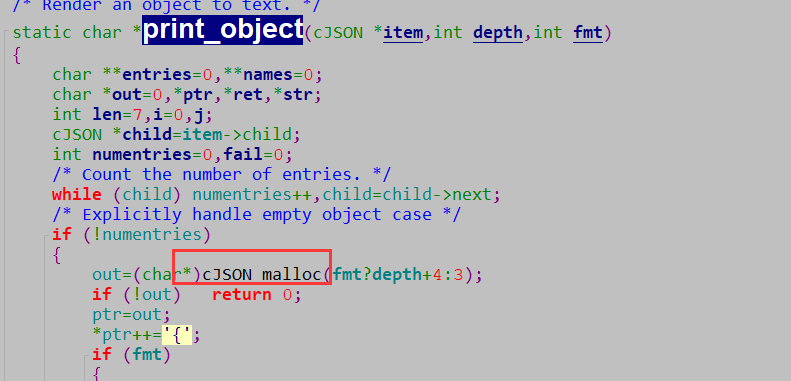


这两行代码负责json数据的解析与打印。

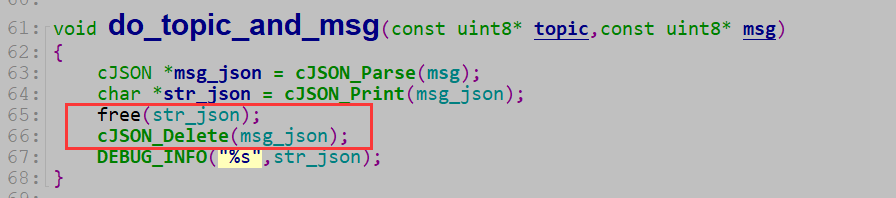


占进去查看,发现cJSON\_Parse和 cJSON\_Print都内部都调用了malloc分配了堆内存，所以也要相应的释放。

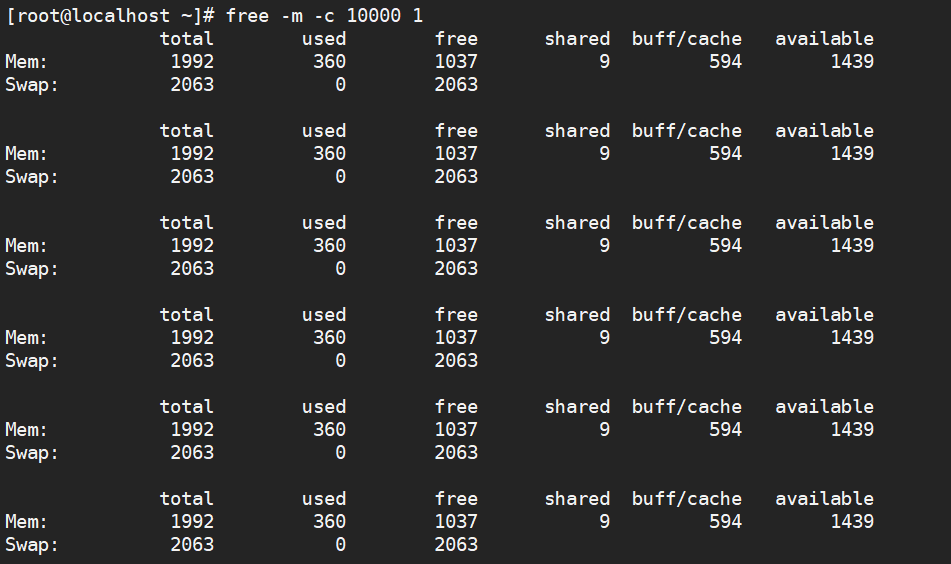




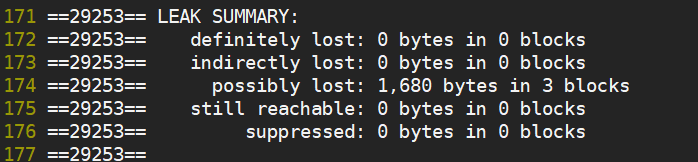
处理方法，释放内存：



执行free -m查看，内存基本没有再减少了。



再次查看修改后的valgrind memleak分析报告：

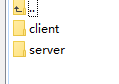


几处内存泄露的点，值都为0。

## 2.3 massif 堆内存分析

可以分析堆内存的分配详细信息。

编译运行以下程序，其中client按照下面的方式运行



valgrind --tool=massif --trace-children=yes --heap=yes --massif-out-file=massif.out.%p ./client

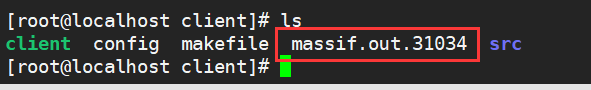
各参数含义：

--trace-children=yes #同时跟踪子线程

--heap=yes #分析堆信息使能

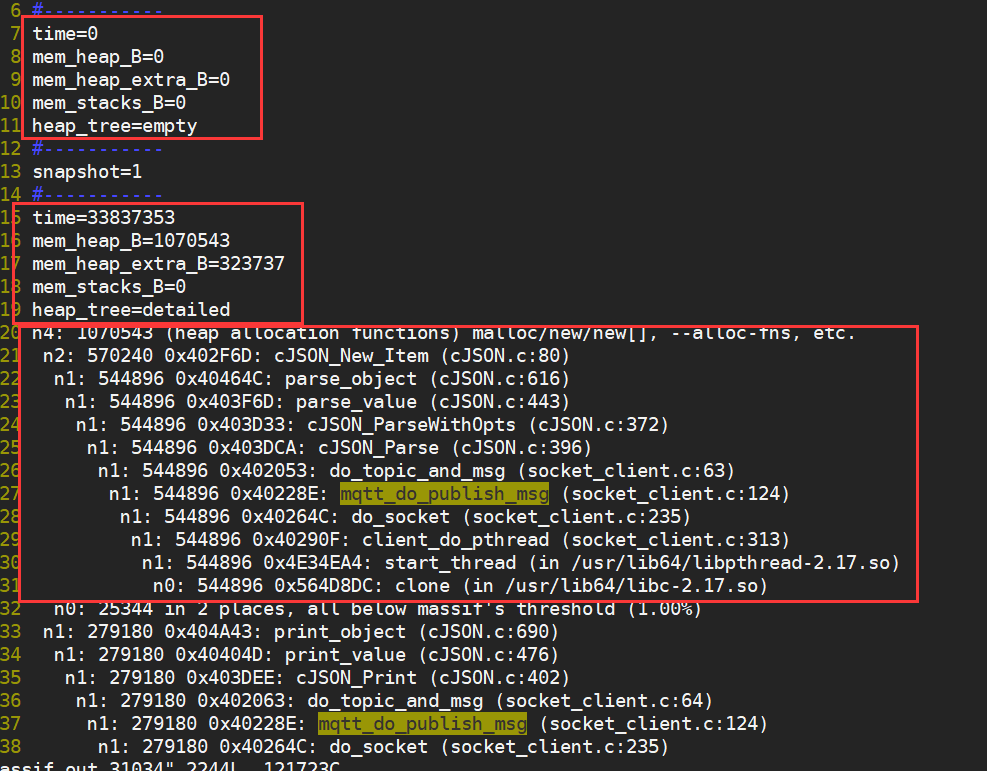
--massif-out-file=massif.out.%p #分析文件按照这种格式输出

运行一段时间后，得到以下文件



打开查看

可以得出堆内存分配情况，及函数调用堆栈。



## 2.4 helgrind 线程分析

检测多线程的错误问题，包括，线程API的使用错误，数据同步（访问共享资源没有加锁保护）等。

下面以访问共享资源没有加锁保护为案例，分析一下helgrind的使用方法。

编译：

gcc mutex.c -g -O0 -lpthread

运行：

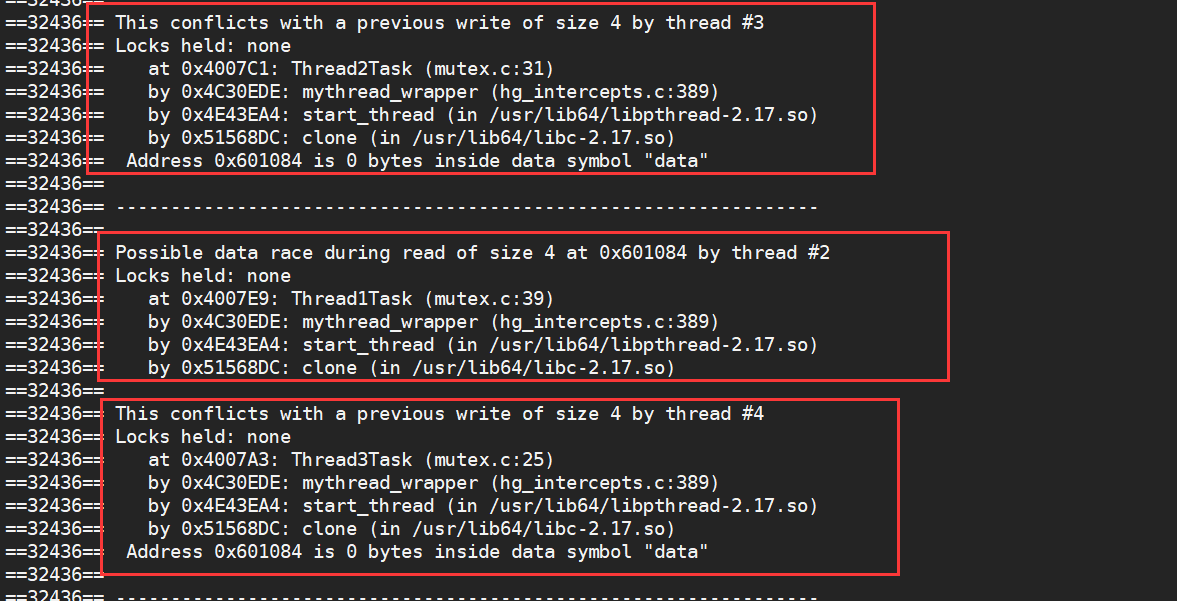
valgrind --tool=helgrind --track-lockorders=yes --trace-children=yes ./a.out

参数：

--track-lockorders=yes #表示进行锁检测

查看检测报告:

从报告可以看出，线程Thread2Task，Thread1Task，Thread3Task有4字节数据存在竞争，没有加锁保护。



## 2.5 cachegrind cache分析

模拟程序如何与机器的缓存层次结构和(可选的)分支交互预测。

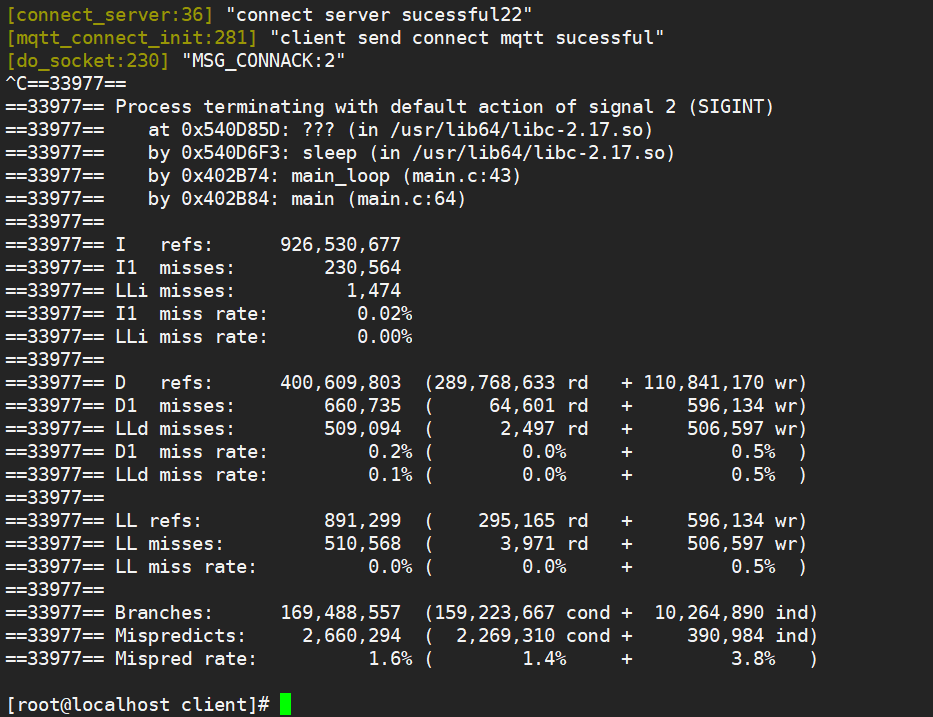
编译运行以下代码实例:



其中客户端以以下方式运行：

valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes --cachegrind-out-file=cache.%p ./client

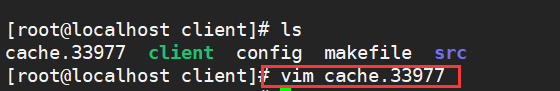
运行一段时间后，停止,查看停止后的报告分析：

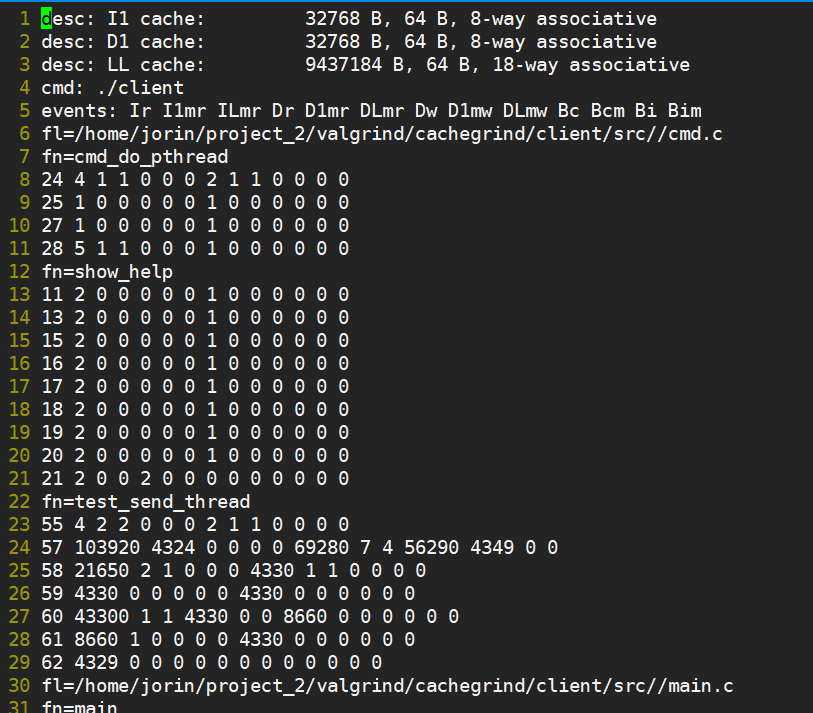


从表中，可以得到指令cache和数据cache的miss情况。

打开文件：

cache.33977



从报告中可以得出，I1，D1 cache的使用情况。

## 2.6 总结

valgrind只能分析随着进程一起启动的相关（内存，线程，cache）的信息，但是对于正在运行的程序就无能为力了，这时就要结合其它工具综合分析，如下：

* 系统已用，可用，剩余内存：free，vmstat，sar，proc/meminfo。
* 进程虚拟内存，常驻内存，共享内存：ps，top。
* 进程内存分布：/proc/[pid]/maps。
* 缓存/缓冲区用量：free，vmstat，sar，bcc工具集合中的(cachestat)。
* 进程换页情况:sar。
* 进程缺页情况:ps ，top。