**内存问题常用调试工具及使用方法**

**一、内存问题概述**

应用程序的编写及调试过程中，会遇到各种问题。在Linux或其他系统平台上，C/C++语言给予我们方便地分配和使用动态内存的特性，这种便利可能会使我们在内存的使用上和管理上的存在不善，而导致各种内存问题。

内存问题主要包含以下几种：

* 写内存越界 (memory overrun)
* 同一块内存释放两次 (double free)
* 内存释放后使用 (use after free)
* 释放内存的参数为非法值 (wild free)
* 访问未初始化内存 (access uninitialized memory)
* 读取非法内存，本质上也属于内存越界 (read invalid memory)
* 内存泄露 (memory leak)
* caller访问一个指针，该指针指向callee的栈内内存 (use after return)
* 栈溢出 (stack overflow)

其中，内存泄漏(memory leak)是指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况，在大型的、复杂的应用程序中，**内存泄漏是常见的内存问题**。当以前分配的一片内存不再需要使用或无法访问时，但是却并没有释放它，这时就出现了内存泄漏。尽管优秀的编程实践可以确保最少的泄漏，但是根据经验，当使用大量的函数对相同的内存块进行处理时，很可能会出现内存泄漏。

内存泄露可以分为以下几类：

1． 常发性内存泄漏。发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。

2． 偶发性内存泄漏。发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

3． 一次性内存泄漏。发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块且仅一块内存发生泄漏。比如，在一个Singletest类的构造函数中分配内存，在析构函数中却没有释放该内存。而Singletest类只存在一个实例，所以内存泄漏只会发生一次。

4． 隐式内存泄漏。程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

总之，内存泄露问题十分令人头疼，尤其在代码量大、模块多且复杂的程序当中，但是，防止内存泄露最基本的还是要从良好的编程习惯做起。

**二、常用工具介绍**

为了检测内存问题，尤其是内存泄露问题，应运而生了许多针对内存问题的检测工具，并且针对不同的系统平台或语言开发环境，工具和功能也各不相同。下面针对Linux平台，主要介绍以下几个运行时内存检查工具：mtrace、memwatch和valgrind。

三种工具的简要对比如表1所示。

表1 三种内存检测工具对比

|  |  |  |  |
| --- | --- | --- | --- |
| 工具名称 | mtrace | memwatch | valgrind |
| 编程语言 | C | C | C/C++ |
| 使用方法 | 包含头文件mcheck.h，并定义环境变量MALLOC\_TRACE为输出文件名，程序开始时调用mtrace()运行程序即可生成日志 | 添加头文件memwatch.h，编译时加上-DMEMWATCH -DMW\_STDIO选项及memwatch.c源码的编译 | 直接使用valgrind工具，基本使用方式(检查内存泄露)：valgrind --error-limit=no --leak-check=full --tool=memcheck <应用程序> |
| 调试输出 | 运行程序后使用  命令：mtrace <可执行程序名> <日志文件> | 生成日志文件memwatch.log，在程序执行期间，错误提示显示在stdout上 | 在调试时添加选项: —log-file=log.txt”表示将结果保存在log.txt文件中 |
| 设计原理 | 为malloc,realloc,free函数添加钩子函数,记录每一对malloc-free的执行 | 将malloc/realloc/calloc/strdup/free等重定义为mwMalloc(sz, \_\_FILE\_\_, \_\_LINE\_\_)等，内部维护一个操作链表 | 根据软件的内存操作维护一个有效地址空间表和无效地址空间表（进程的地址空间） |
| 优缺点 | 只能检查使用malloc/realloc/free造成的的内存泄露 | 能检测双重释放（double-free）、错误释放（erroneous free）、内存泄漏（unfreed memory）、溢出(Overflow)、下溢(Underflow)等等。非线程安全，效率上，大块分配不受影响，小块分配会受影响，因此它没法使用原分配函数中的**memory pool**。最坏情况下会有3-5x的慢速。它可以比较方便地模拟内存受限情况。 | 能够检测：使用未初始化的内存、使用已经释放了的内存、使用超过 malloc分配的内存空间、对堆栈的非法访问、申请的空间是否有释放等。缺点是，其基本原理是让程序跑在一个虚拟机上，因此速度会慢几十倍 |
| 下载地址 | glibc自带,可直接使用，<http://ftp.gnu.org/gnu/glibc/> | <http://www.linkdata.se/sourcecode/memwatch/> (最新版本为2.71) | <http://valgrind.org/> |

提示：

1、**memory pool**中文意思为[内存池](http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E6%B1%A0" \t "_blank)，也叫[动态内存](http://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E5%86%85%E5%AD%98)分配，内存池(Memory Pool)是一种内存分配方式。 通常我们习惯直接使用new、[malloc](http://baike.baidu.com/item/malloc" \t "_blank)等API申请分配内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的[内存碎片](http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E7%A2%8E%E7%89%87" \t "_blank)并进而降低性能。 [内存池](http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E6%B1%A0)则是在真正使用内存之前，先申请分配一定数量的、大小相等（一般情况下）的内存块留作备用。当有新的内存需求时，就从[内存池](http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E6%B1%A0" \t "_blank)中分出一部分内存块，若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了[内存碎片](http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E7%A2%8E%E7%89%87)，使得[内存分配](http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E5%88%86%E9%85%8D)效率得到提升。

三、工具详细使用

**1、Mtrace**

mtrace（memory trace），是GNU Glibc自带的内存问题检测工具，它可以用来协助定位内存泄露问题。它的实现源码在glibc源码的malloc目录下，其基本设计原理为设计一个函数void mtrace ()，函数对libc库中的malloc/free等函数的调用进行追踪，由此来检测内存是否存在泄漏的情况。

首先，我们来看下程序源码tst-mtrace.c：

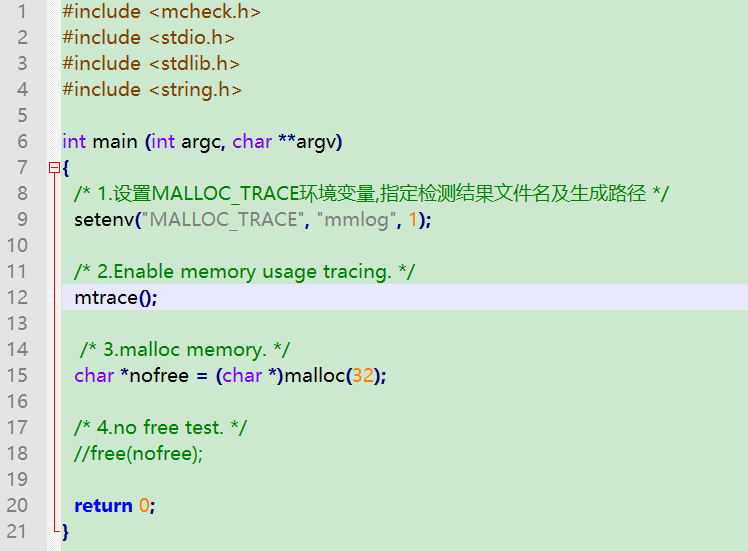


图1 mtrace的使用

图1中的源码，我们很容易可以看到该程序运行后存在内存泄露，因为第20行没有释放第16行malloc申请的内存。当然，以上代码量很少，一眼就可以看出内存泄露。但实际项目，代码量变大后，显然仅凭肉眼的方法检测内存泄露不太可取。于是，我们的mtrace登场了。

**1.1 Mtrace具体使用方法介绍：**

**（1）**、源码

首先，在代码中添加头文件#include <mcheck.h>,

然后，在代码中使用setenv()函数设置MALLOC\_TRACE环境变量，用于指明mtrace生成的log的路径，然后调用glibc库函数mtrace()、muntrace()。

如图1，第9行代码设置mtrace环境变量MALLOC\_TRACE的值为字符串“mmlog”，该字符串表示一个文件名称即mtrace生成的log名，如果当前目录下没有该文件则创建，有则覆盖。

上图程序没有加muntrace，也可以在程序退出前加上，关键取决于你要调试的代码段。

注：MALLOC\_TRACE环境变量也可以在shell环境下设置，如export MALLOC\_TRACE=mmlog

**（2）**、编译

gcc tst-mtrace.c –o tst-mtrace -g

注意：编译时需要加-g调试选项，同时生成可执行后不能进行strip去掉符号表的操作。

**（3）**、运行

直接运行程序

**./tst-mtrace**

会在程序执行目录生成log文件，如图2所示：

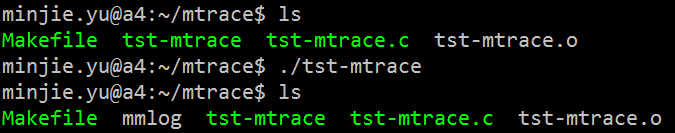


图2 mtrace生成log

**（4）**、分析

我们来看下生成的日志mmlog文件里的内容:

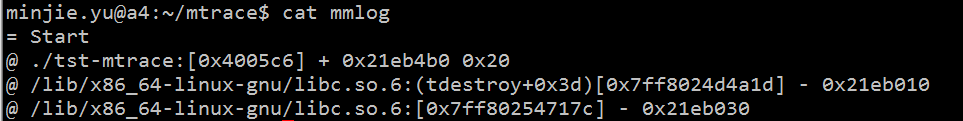


图3 mtrace日志文件内容

tst-mtrace运行生成的log文件，记录了的程序运行时内存的分配信息，包括每次malloc，calloc，free的大小，文件名， 地址等。该文件有时会很大，并且不容易懂。其中，“+”号是malloc，“-”号是free。如图3，程序中仅有一次malloc。

显然不能直接看出日志里那里出现了内存泄露，我们还需要用到mtrace工具（一个perl脚本，它对符号地址进行了转换，转换为代码文本，因此可以直观的看到内存分配信息，它需要编译或交叉编译安装glibc）来将生成可读的内存分配信息。

使用帮助：

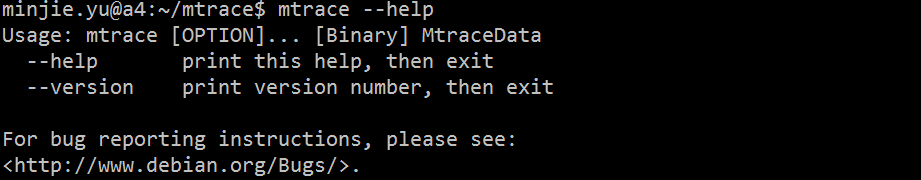


图4 mtrace工具使用帮助

如输入命令：

**mtrace tst-mtrace mmlog**

打印出可读的内存分配信息，如图5所示：

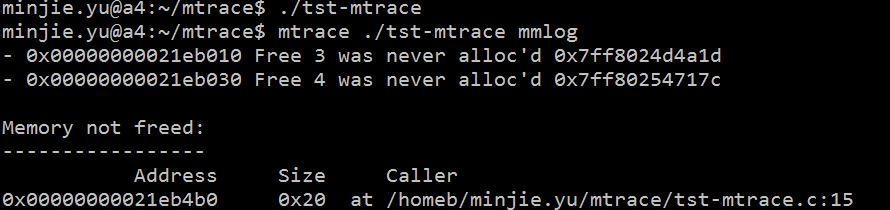


图5 mtrace日志分析

 上图中，可以看到tst-mtrace.c第15行申请的内存没有释放，申请的大小为32个字节。

注：分配信息结果较多时，可以重定向到文件，再利用其他脚本工具awk等再详细分析；

**1.2 Mtrace使用总结:**

1）. 在需要内存泄漏检查的代码的开始调用void mtrace(void) (在mcheck.h中有声明)。mtrace为malloc等函数安装hook, 用于记录内存分配信息. 在需要内存泄漏检查的代码的结束调用void muntrace(void).

注意: 一般情况下不需要调用muntrace, 而让程序自然结束. 因为可能有些释放内存代码要到muntrace之后才运行.

2）. 用debug模式编译被检查代码(-g)，同时不将程序进行strip处理；

3）. 设置环境变量MALLOC\_TRACE为一文件名, 该文件在程序运行后，会保存内存分配信息.；

4）. 运行被检查程序, 直至结束或muntrace被调用；

5）. 用mtrace命令解析内存分配信息log文件($MALLOC\_TRACE) , (mtrace foo $MALLOC\_TRACE, where foo is the executible name) ,如果有内存泄漏, mtrace会输出分配泄漏内存的代码位置,以及分配数量.

提示：

1）. 可以将mtrace, muntrace放入信号处理函数(USR1, USR2), 以动态地进行内存泄漏检查控制；

2）. 尽量不要用muntrace()；

3）.更加详尽的使用，请参考glibc相关文档；

**2、Memwatch**

memwatch(memory watch)由Johan Lindh编写,是一个开放源代码C语言内存错误检测工具。memwatch支持 ANSI C，它提供结果日志纪录，能检测双重释放（double-free）、错误释放（erroneous free）、内存泄漏（unfreed memory）、溢出(Overflow)、下溢(Underflow)等等。

**2.1 Memwatch的内存管理：**

memwatch将全部分配的内存用0xFE填充，所以，假设你看到错误的数据是用0xFE填充的，那就是你没有初始化数据。例外是calloc()，它会直接把分配的内存用0填充。

memwatch将全部已释放的内存用0xFD填充(zapped with 0xFD).如果你发现使用的数据是用0xFD填充的，那你就使用的是已释放的内存。

为了帮助跟踪野指针的写情况，memwatch能提供no-mans-land（NML）内存填充。no-mans-land将使用0xFC填充.当no-mans-land开启时，memwatch转变释放的内存为NML填充状态。

**2.2 Memwatch具体使用方法介绍：**

在程序中，为了使用memwatch，不需要额外的安装，只需要使用其提供的函数，并引入头文件memwatch.h，同时将memwatch.c和程序源码一起编译即可。使用过程中，涉及到的常用函数：

**mwInit() mwTerm() mwAbort()**

mwInit()负责初始化memwatch，而mwTerm()终止它。如果你没有自己调用mwInit()的话，memwatch也可以自动初始化。这样的话，memwatch就要使用atexit()注册mwTerm()到atexit队列中。

如果你自己使用了atexit()去作清除工作，自动初始化技术会有一个警告。memwatch可能会在程序完成前已经终止。处于安全的考虑，请使用mwInit()和mwTerm()。

在一个程序终止可控的方式下，你可能要使用mwAbort()来替代mwTerm()。mwAbort()将会停止memwatch，即使还有未处理的mwTerm()。

推荐使用的方式是，mwInit()和mwTerm()配合一起来使用。

提示：

1)、引用memwatch.h，会将malloc/realloc/calloc/strdup/free等ANSI C函数重定义为mwMalloc(n,\_\_FILE\_\_,\_\_LINE\_\_)、mwFree(p,\_\_FILE\_\_,\_\_LINE\_\_)等函数，memwatch.c的实现则对这些mwMalloc()等函数，维护一个操作链表，因此可以记录到内存的使用情况。

2)、编译时，需要添加**-DMEMWATCH -DMW\_STDIO**

**2.3 Memwatch具体使用方法介绍：**

程序源码如下：

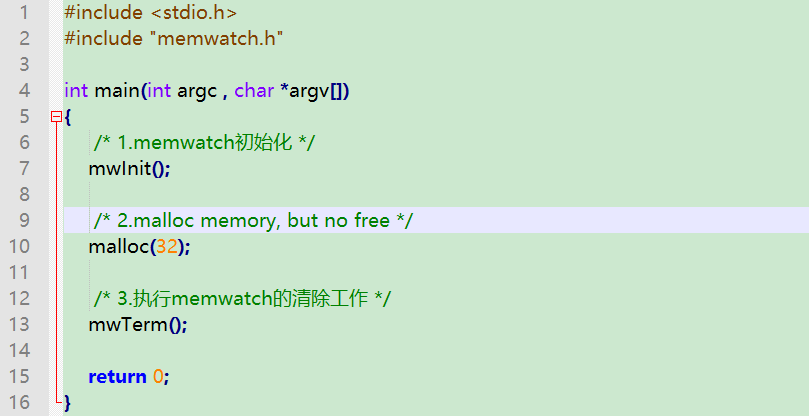


图6 tst-memwatch.c源码

**1）**、源码

程序源码中，第2行引入了头文件#include “memwatch.h”,分别在程序开始和结尾处，用到了mwInit()和mwTerm()函数。

源码中，第10行有内存的申请操作，但整个程序没有对这块申请的内存的释放处理。

**2）**、编译

编译该源程序：

gcc -DMEMWATCH -DMW\_STDIO memwatch.c tst-memwatch.c -o tst-memwatch

**3)**、运行

执行./tst-memwatch,运行结果如下：

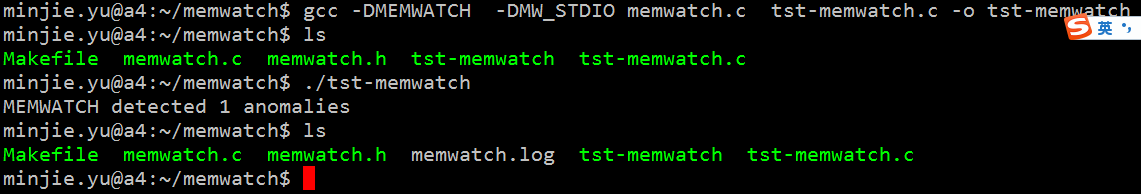


图7 tst-memwatch运行结果

运行结果中打印出：MEMWATCH detected 1 anomalies，表示检测到了内存异常。同时，生成了日志文件memwatch.log，文件全部内容如图8所示:

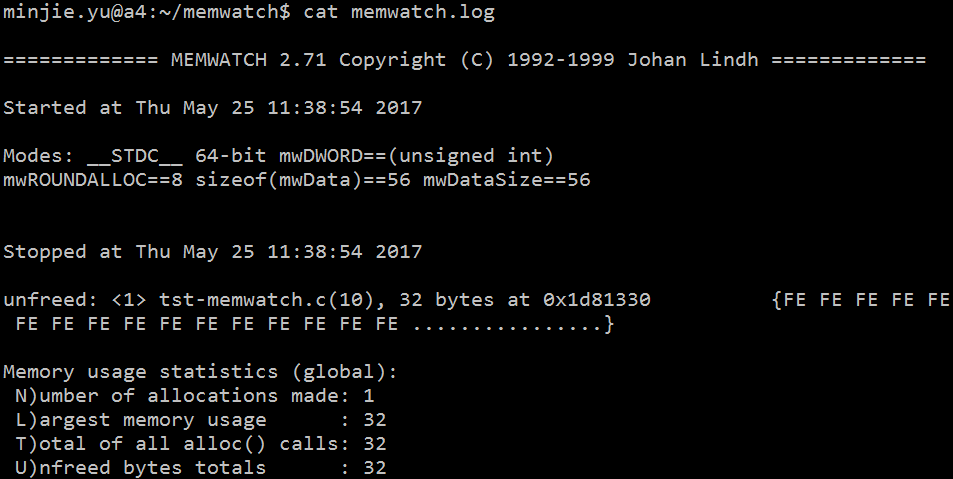


图8 memwatch日志

日志文件中的以下内容，描述了程序出现未释放内存的问题，出现了一次，且出现在tst-memwatch.c的第10行，大小为32个字节，其中并打印出了未释放的内存的数据全为FE（memwatch分配的内存用0xFE填充）。同时，在日志文件的末尾，有针对该程序的详细统计信息，由此我们便可定位，程序中的内存问题。

unfreed: <1> tst-memwatch.c(10), 32 bytes at 0x1d81330 {FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE ................}

Memory usage statistics (global):

N)umber of allocations made: 1

L)argest memory usage : 32

T)otal of all alloc() calls: 32

U)nfreed bytes totals : 32

**2.3 Memwatch的使用注意事项**

memwatch的优点是无需特別配置，不需安装便能使用，但缺点是它会拖慢程序的运行速度，尤其是释放内存时它会作大量检查。但它比mtrace和dmalloc(另一内存检测工具)多了一项功能，就是能模拟系统内存不足的情況，使用者只需用mwLimit(long num\_of\_byte)函数来限制程序的heap memory大小(以byte单位)。

最详细的使用说明(包括优点缺点，运行原理等)已在源码包(<http://www.linkdata.se/downloads/sourcecode/memwatch/memwatch-2.71.tar.gz>)里的USING中列出，强烈建议使用前参考该文件。

**3、Vargind**

Valgrind是一套Linux下，开放源代码的动态调试工具集合，能够检测内存管理错误、线程BUG等，目前最新版本为3.12.0（<http://valgrind.org/downloads/valgrind-3.12.0.tar.bz2>）。Valgrind由内核（core）以及基于内核的其他调试工具组成。内核类似于一个框架（framework），它模拟了一个CPU环境，并提供服务给其他工具；而其他工具则类似于插件 (plug-in)，利用内核提供的服务完成各种特定的内存调试任务。

注：valgrind支持的所有平台列表：

X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x and later), ARM64/Android, X86/Android (4.0 and later), MIPS32/Android, X86/Darwin and AMD64/Darwin (Mac OS X 10.10, with initial support for 10.11)

Valgrind包括的工具如下：

1、Memcheck，这是valgrind应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。这也是本文将重点介绍的部分。

2、Callgrind，主要用来检查程序中函数调用过程中出现的问题。

3、Cachegrind，主要用来检查程序中缓存使用出现的问题。

4、Helgrind，它主要用来检查多线程程序中出现的竞争问题。

5、Massif，它主要用来检查程序中堆栈使用中出现的问题。

6、Extension，可以利用core提供的功能，自己编写特定的内存调试工具

* 1. **Valgrind在ARM平台的交叉编译**

交叉编译过程如下：

# wget http://valgrind.org/downloads/valgrind-3.12.0.tar.bz2

# tar -jxvf valgrind-3.12.0.tar.bz2

# cd valgrind-3.12.0

# ./autogen.sh

执行完autogen.sh后，修改configure脚本 ，将armv7\*) 改为 armv7\* | arm )，然后再执行以下configure脚本

# ./configure --host=arm-hisiv200-linux CC=arm-hisiv200-linux-gcc CPP=arm-hisiv200-linux-cpp CXX=arm-hisiv200-linux-g++ --prefix=**/mnt/valgrind**

# make;make install

在ARM平台上，用上述configure脚本，生成的可执行程序路径在/mnt/valgrind下，注意实际使用时，也应将生成的程序放到/mnt目录下。

* 1. **Valgrind使用常用选项简介**

用法: valgrind [options] prog-and-args [options]:

--tool=<name> 最常用的选项。运行 valgrind中名为toolname的工具。默认memcheck。

-h --help 显示帮助信息。

--version 显示valgrind内核的版本，每个工具都有各自的版本。

-q --quiet 安静地运行，只打印错误信息。

-v --verbose 更详细的信息, 增加错误数统计。

--trace-children=no|yes 跟踪子线程? [no]

--track-fds=no|yes 跟踪打开的文件描述？[no]

--time-stamp=no|yes 增加时间戳到LOG信息? [no]

--log-fd=<number> 输出LOG到描述符文件 [2=stderr]

--log-file=<file> 将输出的信息写入到filename.PID的文件里，PID是运行程序的进行ID

--log-file-exactly=<file> 输出LOG信息到 file

--log-file-qualifier=<VAR> 取得环境变量的值来做为输出信息的文件名。 [none]

--log-socket=ipaddr:port 输出LOG到socket ，ipaddr:port

LOG信息输出：

--xml=yes 将信息以xml格式输出，只有memcheck可用

--num-callers=<number> show <number> callers in stack traces [12]

--error-limit=no|yes 如果太多错误，则停止显示新错误? [yes]

--error-exitcode=<number> 如果发现错误则返回错误代码 [0=disable]

--db-attach=no|yes 当出现错误，valgrind会自动启动调试器gdb。[no]

--db-command=<command> 启动调试器的命令行选项[gdb -nw %f %p]

Memcheck工具的相关选项：

--leak-check=no|summary|full 要求对leak给出详细信息? [summary]

--leak-resolution=low|med|high how much bt merging in leak check [low]

--show-reachable=no|yes show reachable blocks in leak check? [no]

例如检查内存泄露使用：

/mnt/valgrind/bin/valgrind --error-limit=no --leak-check=full --tool=memcheck ./sample

注：./sample为可执行程序名

* 1. **Valgrind的使用介绍**

**1、内存泄露**

源码如下：

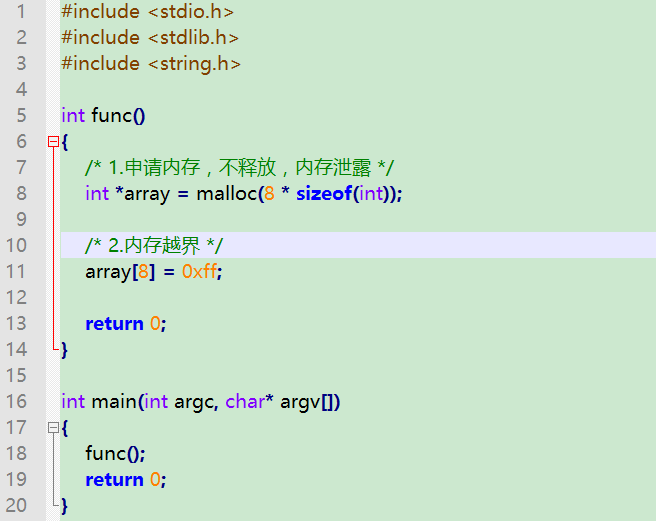


图9 tst-memcheck.c源码

源码中，存在两个缺陷：其一，第8行代码，申请内存后，在程序运行完成后没有释放；其二，第11行存在越界访问内存的问题。

**Valgrind使用：**

1. **在海思Linux平台的运行,**

运行工具时，可能会出现提示ld-linux.so.3的问题，解决方案如下：

1）、需要将ld-linux.so.3 拷贝到机顶盒名字为/lib/ld-2.11.1.so

2）、针对可写的文件系统，可以重新采用链接so的方式：ln -sf /mnt/valgrind/ld-linux.so.3 /lib/ld-linux.so.3

注：

1. ld-linux.so.3是glibc编译出的debug版本的库，针对arm-hisi平台，可以更新需要自己进行交叉编译glibc-2.11.1(注意：海思高清平台glibc的版本为glibc-2.11.1)，生成的ld-linux.so.3和libc.so.6替换到盒子中；
2. 如果采用软连接方式，还原软连接处理为：

ln -sf /lib/ld-2.11.1.so ld-linux.so.3

1. **运行监测命令如下：**

./valgrind/bin/valgrind --error-limit=no --leak-check=full --tool=memcheck ./tst-memcheck

运行后打印的结果如图10所示：

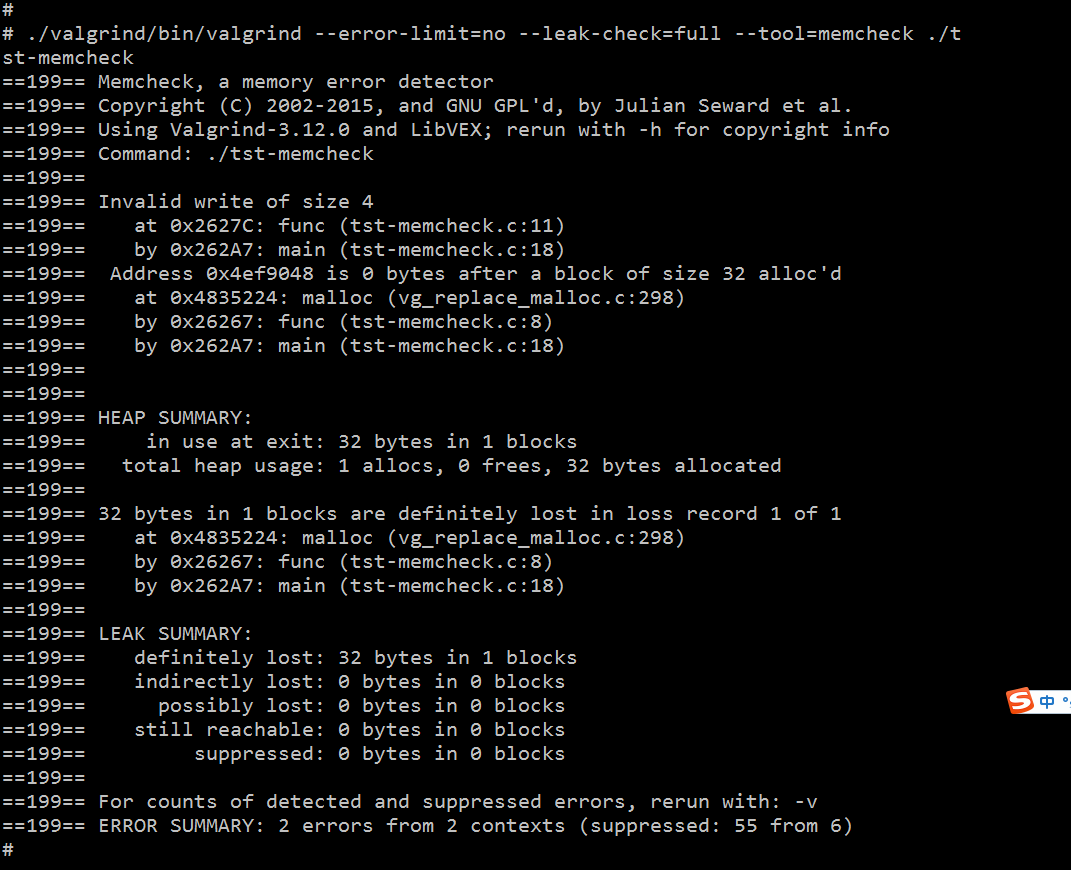


图10 valgrind运行内存泄露检测结果报告

结果报告中,最后一行显示存在两个错误，分别如下：

1）、

==198== Invalid write of size 4

==198== at 0x2627C: func (tst-memcheck.c:11)

==198== by 0x262A7: main (tst-memcheck.c:18)

==198== Address 0x4ef9048 is 0 bytes after a block of size 32 alloc'd

==198== at 0x4835224: malloc (vg\_replace\_malloc.c:298)

==198== by 0x26267: func (tst-memcheck.c:8)

==198== by 0x262A7: main (tst-memcheck.c:18)

显示tst-memcheck.c第11行， array[8]访问了无效的地址0x4ef9048.

2）、

==198== 32 bytes in 1 blocks are definitely lost in loss record 1 of 1

==198== at 0x4835224: malloc (vg\_replace\_malloc.c:298)

==198== by 0x26267: func (tst-memcheck.c:8)

==198== by 0x262A7: main (tst-memcheck.c:18)

==198==

==198== LEAK SUMMARY:

==198== definitely lost: 32 bytes in 1 blocks

==198== indirectly lost: 0 bytes in 0 blocks

==198== possibly lost: 0 bytes in 0 blocks

==198== still reachable: 0 bytes in 0 blocks

==198== suppressed: 0 bytes in 0 blocks

tst-memcheck.c第8行代码，确定的内存泄露(definitely lost)。

* 1. **Valgrind的使用总结**

1、valgrind输出结果，提示没有符合表显示"？？？",编译时不对程序或库进行strip处理，同时需要注意编译加-g选项；

2、一些常用选项的使用详解:

1)、leak-check

--leak-check=<no|summary|yes|full> [default: summary]

用于控制内存泄漏检测力度。

no，不检测内存泄漏；

summary，仅报告总共泄漏的数量，不报告具体泄漏位置；

yes/full，报告泄漏总数、泄漏的具体位置。

2)、show-reachable

--show-reachable=<yes|no> [default: no]

用于控制是否检测控制范围之外的泄漏，比如全局指针、static指针等。

#include <stdlib.h>

#include <stdio.h>

char \*gptr = NULL;

int main(void)

{

gptr = (char \*)malloc(10);

return 0;

}

对应以上代码，若--show-reachable为no，则valgrind不报告内存泄漏，否则会报告。

3)、undef-value-errors

--undef-value-errors=<yes|no> [default: yes]

用于控制是否检测代码中使用未初始化变量的情况。

对应以下代码：

int a;

printf("a = %d \n", a);

若 --undef-value-errors＝no，则valgrind不报告错误，否则报告“Use of uninitialised value ...”的错误。

4)、其他选项

**--log-file=filename 将结果输出到文件。**

--log-socket=192.168.213.32:8080 输出到网络。

--trace-children=<yes|no> [default: no]

--track-fds=<yes|no> [default: no]

--log-fd=<number> [default: 2, stderr]

--xml=<yes|no> [default: no]

--num-callers=<number> [default: 12]

--show-below-main=<yes|no> [default: no]

1. valgrind其他详细使用可参考教程：

<https://www.ibm.com/developerworks/cn/linux/l-cn-valgrind/>

1. 其他详细信息，可参考源码帮助文档；