valgrind的用法-nlchjian-ChinaUnix博客

valgrind的用法

－－

源自：http://hi.baidu.com/franco%5Fyin/blog/item/9568bcece59cd3d52f2e2183.html

Valgrind查找内存泄露利器

Valgrind是一个GPL的软件，用于Linux（For x86, amd64 and ppc32）程序的内存调试和代码剖析。你可以在它的环境中运行你的程序来监视内存的使用情况，比如C 语言中的malloc和free或者 C++中的new和 delete。使用Valgrind的工具包，你可以自动的检测许多内存管理和线程的bug，避免花费太多的时间在bug寻找上，使得你的程序更加稳固。

Valgrind的主要功能
Valgrind工具包包含多个工具，如Memcheck,Cachegrind,Helgrind, Callgrind，Massif。下面分别介绍个工具的作用：

Memcheck 工具主要检查下面的程序错误：

使用未初始化的内存 (Use of uninitialised memory)
使用已经释放了的内存 (Reading/writing memory after it has been free'd)
使用超过 malloc分配的内存空间(Reading/writing off the end of malloc'd blocks)
对堆栈的非法访问 (Reading/writing inappropriate areas on the stack)
申请的空间是否有释放 (Memory leaks – where pointers to malloc'd blocks are lost forever)
malloc/free/new/delete申请和释放内存的匹配(Mismatched use of malloc/new/new [] vs free/delete/delete [])
src和dst的重叠(Overlapping src and dst pointers in memcpy() and related functions)
Callgrind

Callgrind收集程序运行时的一些数据，函数调用关系等信息，还可以有选择地进行cache 模拟。在运行结束时，它会把分析数据写入一个文件。callgrind_annotate可以把这个文件的内容转化成可读的形式。

Cachegrind
它模拟 CPU中的一级缓存I1,D1和L2二级缓存，能够精确地指出程序中 cache的丢失和命中。如果需要，它还能够为我们提供cache丢失次数，内存引用次数，以及每行代码，每个函数，每个模块，整个程序产生的指令数。这对优化程序有很大的帮助。

Helgrind
它主要用来检查多线程程序中出现的竞争问题。Helgrind 寻找内存中被多个线程访问，而又没有一贯加锁的区域，这些区域往往是线程之间失去同步的地方，而且会导致难以发掘的错误。Helgrind实现了名为” Eraser” 的竞争检测算法，并做了进一步改进，减少了报告错误的次数。

Massif
堆栈分析器，它能测量程序在堆栈中使用了多少内存，告诉我们堆块，堆管理块和栈的大小。Massif能帮助我们减少内存的使用，在带有虚拟内存的现代系统中，它还能够加速我们程序的运行，减少程序停留在交换区中的几率。

Valgrind 安装
1、 到www.valgrind.org下载最新版valgrind-3.2.3.tar.bz2
2、 解压安装包：tar –jxvf valgrind-3.2.3.tar.bz2
3、 解压后生成目录valgrind-3.2.3
4、 cd valgrind-3.2.3
5、 ./configure6、 Make;make install

1.检查内存错误：
例如我们原来有一个程序sec_infod，这是一个用gcc –g参数编译的程序，运行它需要：
#./a.out
如果我们想用valgrind的内存检测工具，我们就要用如下方法调用：
#valgrind --leak-check=full --show-reachable=yes --trace-children= yes ./a.out (2>logfile加上会好些，程序在执行期间stderr会有一些输出。提示比较多)

其中--leak-check=full 指的是完全检查内存泄漏，--show-reachable=yes是显示内存泄漏的地点，--trace-children=yes是跟入子进程。

如果您的程序是会正常退出的程序，那么当程序退出的时候valgrind自然会输出内存泄漏的信息。如果您的程序是个守护进程，那么也不要紧，我们只要在别的终端下杀死memcheck进程（因为valgrind默认使用memcheck工具，就是默认参数—tools=memcheck）：
#killall memcheck
这样我们的程序（./a.out）就被kill了

2,检查代码覆盖和性能瓶颈：
我们调用valgrind的工具执行程序：
#valgrind --tool=callgrind ./sec_infod

会在当前路径下生成callgrind.out.pid（当前生产的是callgrind.out.19689），如果我们想结束程序，可以：
#killall callgrind
然后我们看一下结果：
#callgrind_annotate --auto=yes callgrind.out.19689 >log
#vim log

3.Valgrind使用参数

--log-fd=N 默认情况下，输出信息是到标准错误stderr，也可以通过—log-fd=8，输出到描述符为8的文件

--log-file=filename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。可以通过--log-file exactly=filename指定就输出到filename文件。

--log-file-qualifier=,*取得环境变量的值来做为输出信息的文件名。如—log-file-qualifier=$FILENAME。*

--log-socket=IP:PORT 也可以把输出信息发送到网络中指定的IP:PORT去

--error-limit=no 对错误报告的个数据进行限制，默认情况不做限制

--tool= [default: memcheck]

--tool=memcheck：要求用memcheck这个工具对程序进行分析

--leak-ckeck=yes 要求对leak给出详细信息

--trace-children= [default: no]跟踪到子进程里去，默认请况不跟踪

--xml= [default: no]将信息以xml格式输出，只有memcheck可用
--gen-suppressions= [default: no]如果为yes，valgrind会在每发现一个错误便停下让用户做选择是继续还是退出

更多选项请参看： http://www.valgrind.org/docs/manual/manual-core.html可以把一些默认选项编辑在 ~/.valgrindrc文件里。

这里使用valgrind的memcheck和callgrind两个工具的用法，其实valgrind还有几个工具：“cachegrind”,用于检查缓存使用的；“helgrind”用于检测多线程竞争资源的，等等。

－－－

源自：http://hi.baidu.com/franco_yin/blog/item/7c5be1973d4bd86954fb9675.html

Valgrind手册翻译（上）

名字：
valgrind是一个调试和剖析的程序工具集。

概要用法：
valgrind [[valgrind] [options]] [your-program] [[your-program-options]]

概述：
Valgrind是一个Linux下灵活的调试和剖析可执行工具。它由在软件层提供综合的CPU内核,和一系列调试、剖析的工具组成。架构是模块化的，所以可以在不破坏现有的结构的基础上很容易的创建出新的工具来。
这本手册包括了基本的用法和选项。更多帮助理解的信息，请查看您系统的HTML文档：
/usr/share/doc/valgrind/html/index.html
或者在线文档：
http://www.valgrind.org/docs/manual/index.html.

用法：

一般像下面这样调用Valgrind:
valgrind program args

这样将在Valgrind使用Memcheck运行程序program(带有参数args)。内存检查
执行一系列的内存检查功能，包括检测访问未初始化的内存，已经分配内存的错误
使用(两次释放，释放后再访问，等等)并检查内存泄漏。

可用--tool指定使用其它工具：
valgrind --tool=toolname program args

可使用的工具如下：
o cachegrind是一个缓冲模拟器。它可以用来标出你的程序每一行执行的指令
数和导致的缓冲不命中数。

o callgrind在cachegrind基础上添加调用追踪。它可以用来得到调用的次数
以及每次函数调用的开销。作为对cachegrind的补充，callgrind可以分别
标注各个线程，以及程序反汇编输出的每条指令的执行次数以及缓存未命中
数。

o helgrind能够发现程序中潜在的条件竞争。

o lackey是一个示例程序，以其为模版可以创建你自己的工具。在程序结束后，
它打印出一些基本的关于程序执行统计数据。

o massif是一个堆剖析器，它测量你的程序使用了多少堆内存。

o memcheck是一个细粒度的的内存检查器。

o none没有任何功能。它它一般用于Valgrind的调试和基准测试。

基本选项：

这些选项对所有工具都有效。

-h --help
显示所有选项的帮助，包括内核和选定的工具两者。

--help-debug
和--help相同，并且还能显示通常只有Valgrind的开发人员使用的调试选项。

--version
显示Valgrind内核的版本号。工具可以有他们自己的版本号。这是一种保证工具只在它们可以运行的内核上工作的一种设置。这样可以减少在工具和内核之间版本兼容性导致奇怪问题的概率。

-q --quiet
安静的运行，只打印错误信息。在进行回归测试或者有其它的自动化测试机制时会非常有用。

-v --verbose
显示详细信息。在各个方面显示你的程序的额外信息，例如：共享对象加载，使用的重置，执行引擎和工具的进程，异常行为的警告信息。重复这个标记可以增加详细的级别。

-d 调试Valgrind自身发出的信息。通常只有Valgrind开发人员对此感兴趣。重复这个标记可以产生更详细的输出。如果你希望发送一个bug报告，通过-v -v -d -d生成的输出会使你的报告更加有效。

--tool= [default: memcheck]
运行toolname指定的Valgrind，例如，Memcheck, Addrcheck, Cachegrind, 等等。

--trace-children= [default: no]

当这个选项打开时，Valgrind会跟踪到子进程中。这经常会导致困惑，而且通常不是你所期望的，所以默认这个选项是关闭的。

--track-fds= [default: no]
当这个选项打开时，Valgrind会在退出时打印一个打开文件描述符的列表。每个文件描述符都会打印出一个文件是在哪里打开的栈回溯，和任何与此文件描述符相关的详细信息比如文件名或socket信息。

--time-stamp= [default: no]
当这个选项打开时，每条信息之前都有一个从程序开始消逝的时间，用天，小时，分钟，秒和毫秒表示。

--log-fd= [default: 2, stderr]
指定Valgrind把它所有的消息都输出到一个指定的文件描述符中去。默认值2， 是标准错误输出(stderr)。注意这可能会干扰到客户端自身对stderr的使用, Valgrind的输出与客户程序的输出将穿插在一起输出到stderr。

--log-file=
指定Valgrind把它所有的信息输出到指定的文件中。实际上，被创建文件的文件名是由filename、'.'和进程号连接起来的（即.），
从而每个进程创建不同的文件。

--log-file-exactly=
类似于--log-file，但是后缀".pid"不会被添加。如果设置了这个选项，
使用Valgrind跟踪多个进程，可能会得到一个乱七八糟的文件。

--log-file-qualifier=
当和--log-file一起使用时，日志文件名将通过环境变量$VAR来筛选。这对于MPI程序是有益的。更多的细节，查看手册2.3节 "注解"。

--log-socket=
指定Valgrind输出所有的消息到指定的IP，指定的端口。当使用1500端口时，端口有可能被忽略。如果不能建立一个到指定端口的连接，Valgrind将输出写到标准错误(stderr)。这个选项经常和一个Valgrind监听程序一起使用。更多的细节，查看手册2.3节 "注解"。

错误相关选项：
这些选项适用于所有产生错误的工具，比如Memcheck, 但是Cachegrind不行。

--xml= [default: no]
当这个选项打开时，输出将是XML格式。这是为了使用Valgrind的输出做为输入的工具，例如GUI前端更加容易些。目前这个选项只在Memcheck时生效。

--xml-user-comment=
在XML开头 附加用户注释，仅在指定了--xml=yes时生效，否则忽略。

--demangle= [default: yes]
打开/关闭C++的名字自动解码。默认打开。当打开时，Valgrind将尝试着把编码过的C++名字自动转回初始状态。这个解码器可以处理g++版本为2.X, 3.X或4.X生成的符号。

一个关于名字编码解码重要的事实是，禁止文件中的解码函数名仍然使用他们未解码的形式。Valgrind在搜寻可用的禁止条目时不对函数名解码，因为这将使禁止文件内容依赖于Valgrind的名字解码机制状态， 会使速度变慢，且无意义。

--num-callers= [default: 12]
默认情况下，Valgrind显示12层函数调用的函数名有助于确定程序的位置。可以通过这个选项来改变这个数字。这样有助在嵌套调用的层次很深时确定程序的位置。注意错误信息通常只回溯到最顶上的4个函数。(当前函数，和

它的3个调用者的位置)。所以这并不影响报告的错误总数。

这个值的最大值是50。注意高的设置会使Valgrind运行得慢，并且使用更多的内存,但是在嵌套调用层次比较高的程序中非常实用。

--error-limit= [default: yes]
当这个选项打开时，在总量达到10,000,000，或者1,000个不同的错误，Valgrind停止报告错误。这是为了避免错误跟踪机制在错误很多的程序下变成一个巨大的性能负担。

--error-exitcode= [default: 0]
指定如果Valgrind在运行过程中报告任何错误时的退出返回值，有两种情况；当设置为默认值(零)时，Valgrind返回的值将是它模拟运行的程序的返回值。当设置为非零值时，如果Valgrind发现任何错误时则返回这个值。在Valgrind做为一个测试工具套件的部分使用时这将非常有用，因为使测试工具套件只检查Valgrind返回值就可以知道哪些测试用例Valgrind报告了错误。

--show-below-main= [default: no]
默认地，错误时的栈回溯不显示main()之下的任何函数(或者类似的函数像glibc的__libc_start_main()，如果main()没有出现在栈回溯中)；这些大部分都是令人厌倦的C库函数。如果打开这个选项，在main()之下的函数也将会显示。

--suppressions= [default: $PREFIX/lib/valgrind/default.supp]
指定一个额外的文件读取不需要理会的错误；你可以根据需要使用任意多的额外文件。

--gen-suppressions= [default: no]
当设置为yes时，Valgrind将会在每个错误显示之后自动暂停并且打印下面这一行：

---- Print suppression ? --- [Return/N/n/Y/y/C/c] ----

这个提示的行为和--db-attach选项(见下面)相同。

如果选择是，Valgrind会打印出一个错误的禁止条目，你可以把它剪切然后粘帖到一个文件，如果不希望在将来再看到这个错误信息。

当设置为all时，Valgrind会对每一个错误打印一条禁止条目，而不向用户询问。

这个选项对C++程序非常有用，它打印出编译器调整过的名字。

注意打印出来的禁止条目是尽可能的特定的。如果需要把类似的条目归纳起来，比如在函数名中添加通配符。并且，有些时候两个不同的错误也会产生同样的禁止条目，这时Valgrind就会输出禁止条目不止一次，但是在禁止条目的文件中只需要一份拷贝(但是如果多于一份也不会引起什么问题)。并且，禁止条目的名字像<在这儿输入一个禁止条目的名字>;名字并不是很重要，它只是和-v选项一起使用打印出所有使用的禁止条目记录。

--db-attach= [default: no]

当这个选项打开时，Valgrind将会在每次打印错误时暂停并打出如下一行：

---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ----

按下回车,或者N、回车，n、回车，Valgrind不会对这个错误启动调试器。

按下Y、回车，或者y、回车，Valgrind会启动调试器并设定在程序运行的这个点。当调试结束时，退出，程序会继续运行。在调试器内部尝试继续运行程序，将不会生效。

按下C、回车，或者c、回车，Valgrind不会启动一个调试器，并且不会再次询问。

注意：--db-attach=yes与--trace-children=yes有冲突。你不能同时使用它们。Valgrind在这种情况下不能启动。

2002.05: 这是一个历史的遗留物，如果这个问题影响到你，请发送邮件并投诉这个问题。

2002.11:如果你发送输出到日志文件或者到网络端口，我猜这不会让你有任何感觉。不须理会。

--db-command= [default: gdb -nw %f %p]

通过--db-attach指定如何使用调试器。默认的调试器是gdb.默认的选项是一个运行时扩展Valgrind的模板。 %f会用可执行文件的文件名替换，%p会被可执行文件的进程ID替换。

这指定了Valgrind将怎样调用调试器。默认选项不会因为在构造时是否检测到了GDB而改变,通常是/usr/bin/gdb.使用这个命令，你可以指定一些调用其它的调试器来替换。

给出的这个命令字串可以包括一个或多个%p %f扩展。每一个%p实例都被解释成将调试的进程的PID，每一个%f实例都被解释成要调试的进程的可执行文件路径。

--input-fd= [default: 0, stdin]

使用--db-attach=yes和--gen-suppressions=yes选项，在发现错误时，Valgrind会停下来去读取键盘输入。默认地，从标准输入读取，所以关闭了标准输入的程序会有问题。这个选项允许你指定一个文件描述符来替代标准输入读取。

--max-stackframe= [default: 2000000]

栈的最大值。如果栈指针的偏移超过这个数量，Valgrind则会认为程序是切换到了另外一个栈执行。

如果在程序中有大量的栈分配的数组，你可能需要使用这个选项。valgrind保持对程序栈指针的追踪。如果栈指针的偏移超过了这个数量，Valgrind假定你的程序切换到了另外一个栈，并且Memcheck行为与栈指针的偏移没有超出这个数量将会不同。通常这种机制运转得很好。然而，如果你的程序在栈上申请了大的结构，这种机制将会表现得愚蠢，并且Memcheck将会报告大量的非法栈内存访问。这个选项允许把这个阀值设置为其它值。

应该只在Valgrind的调试输出中显示需要这么做时才使用这个选项。在这种情况下，它会告诉你应该指定的新的阀值。

普遍地，在栈中分配大块的内存是一个坏的主意。因为这很容易用光你的栈空间，尤其是在内存受限的系统或者支持大量小堆栈的线程的系统上，因为Memcheck执行的错误检查，对于堆上的数据比对栈上的数据要高效很多。如果你使用这个选项，你可能希望考虑重写代码在堆上分配内存而不是在栈上分配。

－－

源自： http://bbs.chinaunix.net/viewthread.php?tid=407853&extra=&page=1

请教**LINUX**下的内存泄露和数组越界等测试工具请教LINUX下的内存泄露和数组越界等测试工具

sorry,刚才查了一下, lint在*nix下是自带的,名称也都是叫lint,比如aix,sunos,……

在linux上,lint工具有增强:

splint:A tool for statically checking C programs

weblint:A Syntax and Minimal Style Checker for HTML

lclint: An implementation of the lint program Libc6 Contribs for i386

jlint:A lint for Java. SourceForge

－－

源自： http://www.cnitblog.com/qiuyangzh/archive/2007/04/27/26292.html

这段时间在对一个产品进行稳定性测试，让程序在Purify下运行，结果Purify最后出现了错误。同事告诉我有一款工具——valgrind，可以定位程序内存方面的错误，于是我下载下来，运行测试，效果很好，这个工具运行稳定，给出的结果也很清晰。

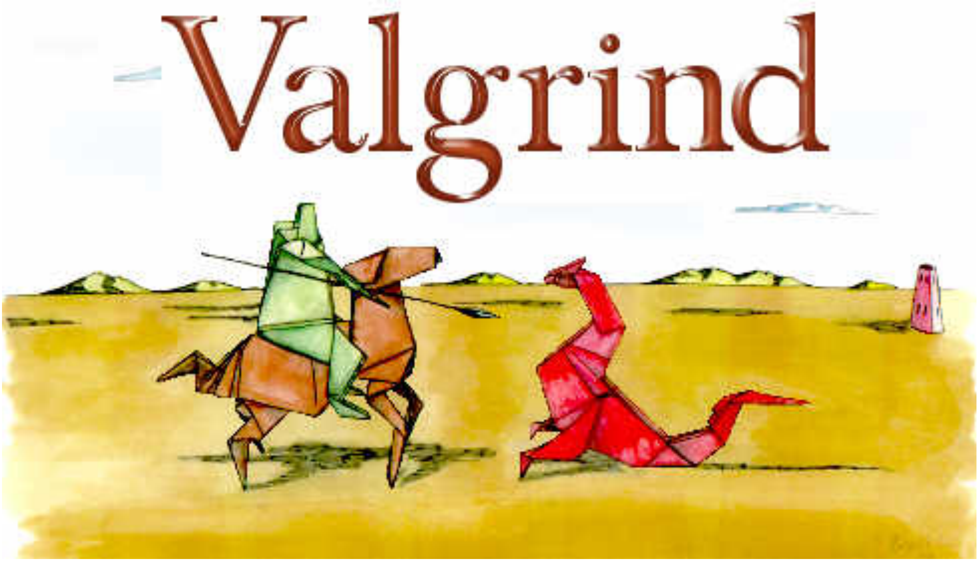
valgrind这款工具是运行在linux下的，可以用来定位c/c++程序中内存使用方面的错误。包括：内存泄漏、使用未初始化的内存、读/写已释放的内存、读/写内存越界、使用malloc/new/new[]和free/delete/delete[]不匹配，等等。

valgrind官方网站：http://valgrind.org/，是一款open source软件。我下载的时候最新版是3.2.3。安装过程很简单——configure，make，make install，安装后不需要任何设置，直接可以开始使用。

valgrind运行方法很简单:valgrind --leak-check=full --track-fds=yes 程序 参数

这样你的程序就会在valgrind监控下运行了，结果会直接输出到屏幕上。如果想把结果输出到日志文件（通常也需要这样），用如下的命令： valgrind --log-file=valgrind_log --leak-check=full --track-fds=yes 程序 参数

感谢我的同事肖锋，告诉我这款工具的信息。也感谢这款工具的开发者——Jonathan Riddell 。



*valgrind网站上的，骑士战恶龙

PS：说起Run-time错误定位工具，我最初使用的是Boundchecker，那还是在Visual c++ 6.0的时代，那款工具确实很好用，功能、性能都不错，但后来不知道Compuware公司出了什么问题，对这款工具的改进很少，到了MS .net出来以后，我试用过Boundchecker针对.net的版本，性能很差。后来是否再有更新我就不清楚了，因为再也没有使用过 Compuware的产品。Purify是现在的公司使用的Run-time错误定位工具，使用方法和Boundchecker非常类似（2002年的时候Compuware指控IBM错误分析器产品中使用自己拥有版权的代码，就是这款产品），也许是因为曾经使用过Boundchecker先入为主的原因，对这款工具始终感觉不是很好，在这次测试过程中，Purify自己竟然宕了。使用过valgrind后，我查了关于它的一些资料，这款工具已经有很多人在使用了，也得过不少殊荣，评价很好。遇到相同需求的朋友，倒是可以试一下valgrind这款工具！

－－

源自： http://www.xker.com/page/e2007/1025/36805.html

用C/C++开发其中最令人头疼的一个问题就是内存管理，有时候为了查找一个内存泄漏或者一个内存访问越界，需要要花上好几天时间，如果有一款工具能够帮助我们做这件事情就好了，valgrind正好就是这样的一款工具。

Valgrind是一款基于模拟linux下的程序调试器和剖析器的软件套件，可以运行于x86, amd64和ppc32架构上。valgrind包含一个核心，它提供一个虚拟的CPU运行程序，还有一系列的工具，它们完成调试，剖析和一些类似的任务。valgrind是高度模块化的，所以开发人员或者用户可以给它添加新的工具而不会损坏己有的结构。

valgrind的官方网址是：http://valgrind.org

你可以在它的网站上下载到最新的valgrind，它是开放源码和免费的。

一、介绍

valgrind包含几个标准的工具，它们是：

1、memcheck

memcheck探测程序中内存管理存在的问题。它检查所有对内存的读/写操作，并截取所有的malloc/new/free/delete调用。因此memcheck工具能够探测到以下问题：

1）使用未初始化的内存

2）读/写已经被释放的内存

3）读/写内存越界

4）读/写不恰当的内存栈空间

5）内存泄漏

6）使用malloc/new/new[]和free/delete/delete[]不匹配。

2、cachegrind

cachegrind是一个cache剖析器。它模拟执行CPU中的L1, D1和L2 cache，因此它能很精确的指出代码中的cache未命中。如果你需要，它可以打印出cache未命中的次数，内存引用和发生cache未命中的每一行代码，每一个函数，每一个模块和整个程序的摘要。如果你要求更细致的信息，它可以打印出每一行机器码的未命中次数。在x86和amd64上， cachegrind通过CPUID自动探测机器的cache配置，所以在多数情况下它不再需要更多的配置信息了。

3、helgrind

helgrind查找多线程程序中的竞争数据。helgrind查找内存地址，那些被多于一条线程访问的内存地址，但是没有使用一致的锁就会被查出。这表示这些地址在多线程间访问的时候没有进行同步，很可能会引起很难查找的时序问题。

二、**valgrind**对你的程序都做了些什么

valgrind被设计成非侵入式的，它直接工作于可执行文件上，因此在检查前不需要重新编译、连接和修改你的程序。要检查一个程序很简单，只需要执行下面的命令就可以了

valgrind --tool=tool_name program_name

比如我们要对ls -l命令做内存检查，只需要执行下面的命令就可以了

valgrind --tool=memcheck ls -l

不管是使用哪个工具，valgrind在开始之前总会先取得对你的程序的控制权，从可执行关联库里读取调试信息。然后在valgrind核心提供的虚拟CPU上运行程序，valgrind会根据选择的工具来处理代码，该工具会向代码中加入检测代码，并把这些代码作为最终代码返回给valgrind核心，最后valgrind核心运行这些代码。

如果要检查内存泄漏，只需要增加--leak-check=yes就可以了，命令如下

valgrind --tool=memcheck --leak-check=yes ls -l

不同工具间加入的代码变化非常的大。在每个作用域的末尾，memcheck加入代码检查每一片内存的访问和进行值计算，代

码大小至少增加12倍，运行速度要比平时慢25到50倍。

valgrind模拟程序中的每一条指令执行，因此，检查工具和剖析工具不仅仅是对你的应用程序，还有对共享库，GNU C 库，X的客户端库都起作用。

三、现在开始

首先，在编译程序的时候打开调试模式（gcc编译器的-g选项）。如果没有调试信息，即使最好的valgrind工具也将中能够猜测特定的代码是属于哪一个函数。打开调试选项进行编译后再用valgrind检查，valgrind将会给你的个详细的报告，比如哪一行代码出现了内存泄漏。

当检查的是C++程序的时候，还应该考虑另一个选项 -fno-inline。它使得函数调用链很清晰，这样可以减少你在浏览大型C++程序时的混乱。比如在使用这个选项的时候，用memcheck检查 openoffice就很容易。当然，你可能不会做这项工作，但是使用这一选项使得valgrind生成更精确的错误报告和减少混乱。

一些编译优化选项(比如-O2或者更高的优化选项)，可能会使得memcheck提交错误的未初始化报告，因此，为了使得valgrind的报告更精确，在编译的时候最好不要使用优化选项。

如果程序是通过脚本启动的，可以修改脚本里启动程序的代码，或者使用--trace-children=yes选项来运行脚本。

下面是用memcheck检查ls -l命令的输出报告，在终端下执行下面的命令

valgrind --tool=memcheck ls -l

程序会打印出ls -l命令的结果，最后是valgrind的检查报告如下：

==4187==

==4187== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 19 from 2)

==4187== malloc/free: in use at exit: 15,154 bytes in 105 blocks.

==4187== malloc/free: 310 allocs, 205 frees, 60,093 bytes allocated.

==4187== For counts of detected errors, rerun with: -v

==4187== searching for pointers to 105 not-freed blocks.

==4187== checked 145,292 bytes.

==4187==

==4187== LEAK SUMMARY:

==4187== definitely lost: 0 bytes in 0 blocks.

==4187== possibly lost: 0 bytes in 0 blocks.

==4187== still reachable: 15,154 bytes in 105 blocks.

==4187== suppressed: 0 bytes in 0 blocks.

==4187== Reachable blocks (those to which a pointer was found) are not shown.

==4187== To see them, rerun with: --show-reachable=yes

这里的“4187”指的是执行ls -l的进程ID，这有利于区别不同进程的报告。memcheck会给出报告，分配置和释放了多少内存，有多少内存泄漏了，还有多少内存的访问是可达的，检查了多少字节的内存。

下面举两个用valgrind做内存检查的例子

例子一 (test.c)：

```
#include

int main(int argc, char *argv[])
```

```
{
    char *ptr;
    ptr = (char*) malloc(10);

    strcpy(ptr, "01234567890");
    return 0;

}
```

编译程序

gcc -g -o test test.c

用valgrind执行命令

valgrind --tool=memcheck --leak-check=yes ./test

报告如下

==4270== Memcheck, a memory error detector.

==4270== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.

==4270== Using LibVEX rev 1606, a library for dynamic binary translation.

==4270== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.

==4270== Using valgrind-3.2.0, a dynamic binary instrumentation framework.

```
==4270== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==4270== For more details, rerun with: -v
==4270==
==4270== Invalid write of size 1
==4270== at 0x4006190: strcpy (mc_replace_strmem.c:271)
==4270== by 0x80483DB: main (test.c:8)
==4270== Address 0x4023032 is 0 bytes after a block of size 10 alloc'd
==4270== at 0x40044F6: malloc (vg_replace_malloc.c:149)
==4270== by 0x80483C5: main (test.c:7)
==4270==
==4270== Invalid write of size 1
==4270== at 0x400619C: strcpy (mc_replace_strmem.c:271)
==4270== by 0x80483DB: main (test.c:8)
==4270== Address 0x4023033 is 1 bytes after a block of size 10 alloc'd
==4270== at 0x40044F6: malloc (vg_replace_malloc.c:149)
==4270== by 0x80483C5: main (test.c:7)
==4270==
==4270== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 12 from 1)
```

==4270== malloc/free: in use at exit: 10 bytes in 1 blocks.

==4270== malloc/free: 1 allocs, 0 frees, 10 bytes allocated.

==4270== For counts of detected errors, rerun with: -v

==4270== searching for pointers to 1 not-freed blocks.

==4270== checked 51,496 bytes.

==4270==

==4270==

==4270== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1

==4270== at 0x40044F6: malloc (vg_replace_malloc.c:149)

==4270== by 0x80483C5: main (test.c:7)

==4270==

==4270== LEAK SUMMARY:

==4270== definitely lost: 10 bytes in 1 blocks.

==4270== possibly lost: 0 bytes in 0 blocks.

==4270== still reachable: 0 bytes in 0 blocks.

==4270== suppressed: 0 bytes in 0 blocks.

==4270== Reachable blocks (those to which a pointer was found) are not shown.

==4270== To see them, rerun with: --show-reachable=yes

从这份报告可以看出，进程号是4270，test.c的第8行写内存越界了，引起写内存越界的是strcpy函数，

第7行泄漏了10个字节的内存，引起内存泄漏的是malloc函数。

例子二（test2.c)

```
#include

int foo(int x)


{
    if (x < 0) {
        printf("%d ", x);
    }

    return 0;


}

int main(int argc, char *argv[])


{
    int x;

       foo(x);

    return 0;
```

}

编译程序

gcc -g -o test2 test2.c

用valgrind做内存检查

valgrind --tool=memcheck ./test2

输出报告如下

==4285== Memcheck, a memory error detector.

==4285== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.

==4285== Using LibVEX rev 1606, a library for dynamic binary translation.

==4285== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.

==4285== Using valgrind-3.2.0, a dynamic binary instrumentation framework.

==4285== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.

==4285== For more details, rerun with: -v

==4285==

==4285== Conditional jump or move depends on uninitialised value(s)

==4285== at 0x8048372: foo (test2.c:5)

==4285== by 0x80483B4: main (test2.c:16)

==4285==p p

==4285== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 1)

==4285== malloc/free: in use at exit: 0 bytes in 0 blocks.

==4285== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.

==4285== For counts of detected errors, rerun with: -v

==4285== All heap blocks were freed -- no leaks are possible.

从这份报告可以看出进程PID是4285，test2.c文件的第16行调用了foo函数，在test2.c文件的第5行foo函数使用了一个未初始化的变量。

valgrind还有很多使用选项，具体可以查看valgrind的man手册页和valgrind官方网站的在线文档。

－－