# 概述

# Valgrind

参考：

<https://w.cnblogs.com/AndyStudy/p/6409287.html>

## 原理

## 使用

工具使用命令：

valgrind --tool=memcheck --leak-check=full --log-file=filename execute\_bin

Ctrl+C结束检测，统计泄露

valgrind支持很多工具：memecheck、addrcheck、cachegrind、massif、helpgrind和callgrind等。

possibly lost是指仍然存在某个指针能够访问某块内存，但该指针指向的已经不是该内存首地址。

definitely lost是指已经不能够访问这块内存，分为两种：直接的（direct）和间接的（indirect）。直接和间接的区别就是，直接是没有任何指针指向该内存，间接是指指向该内存的指针都位于内存泄漏处。

memcheck探测程序中内存管理存在的问题：

1. 使用未初始化的内存unitialised
2. 在内存被释放后进行读/写invalid read
3. 从已分配内存块的尾部进行读/写invalid read
4. 内存泄漏1 byte in 1block are definitely lost in loss record 1 of 1 //definite明确的
5. 不匹配地使用malloc/new/new[]和free/delete/delete[]

# ASAN

参考：

<https://blog.csdn.net/GreatSQL2021/article/details/124704679>

<https://www.modb.pro/db/111652>

## 简介

首先，先介绍一下 Sanitizer 项目，该项目是谷歌出品的一个开源项目，该项目包含了ASAN、LSAN、MSAN、TSAN等内存、线程错误的检测工具，这里简单介绍一下这几个工具的作用：

ASAN: 内存错误检测工具，在编译命令中添加-fsanitize=address启用

LSAN: 内存泄漏检测工具，已经集成到ASAN中，可以通过设置环境变量ASAN\_OPTIONS=detect\_leaks=0来关闭ASAN上的LSAN，也可以使用-fsanitize=leak编译选项代替-fsanitize=address来关闭ASAN的内存错误检测，只开启内存泄漏检查。

MSAN: 对程序中未初始化内存读取的检测工具，可以在编译命令中添加-fsanitize=memory -fPIE -pie启用，还可以添加-fsanitize-memory-track-origins选项来追溯到创建内存的位置

TSAN: 对线程间数据竞争的检测工具，在编译命令中添加-fsanitize=thread启用 其中ASAN就是我们今天要介绍的重头戏。

ASAN，全称AddressSanitizer，可以用来检测内存问题，例如缓冲区溢出或对悬空指针的非法访问等。

根据谷歌的工程师介绍ASAN已经在chromium项目上检测出了300多个潜在的未知bug，而且在使用ASAN作为内存错误检测工具对程序性能损耗也是及其可观的。

根据检测结果显示可能导致性能降低2倍左右，比Valgrind（官方给的数据大概是降低10-50倍）快了一个数量级。

而且相比于Valgrind只能检查到堆内存的越界访问和悬空指针的访问，ASAN 不仅可以检测到堆内存的越界和悬空指针的访问，还能检测到栈和全局对象的越界访问。

这也是ASAN在众多内存检测工具的比较上出类拔萃的重要原因，基本上现在C/C++项目都会使用ASAN来保证产品质量，尤其是大项目中更为需要。

## 原理

ASAN的内存检测方法与Valgrind的AddrCheck工具很像，都是使用shadow内存来记录应用程序的每个字节是否可以被安全的访问，在访问内存时都对其映射的shadow内存进行检查。

但是，ASAN使用一个更具效率的shadow内存映射机制和更加紧凑的内存编码来实现，并且除了堆内存外还能检测栈和全局对象中的错误访问，且比AddrCheck快一个数量级。

ASAN由两部分组成：代码插桩模块和运行时库。

代码插桩模块会修改代码使其在访问内存时检查每块内存访问状态，称为shadow 状态，以及在内存两侧创建redzone的内存区域。

运行时库则提供一组接口用来替代malloc和free以及相关的函数，使得在分配堆空间时在其周围创建redzone，并在内存出错时报告错误。

首先，我们先介绍一下什么是shadow内存和redzone。

### shadow内存

在ASAN中malloc函数返回的内存地址通常至少是8个字节对齐，比如malloc(15)将分配得到2块大小为8字节的内存，在这个场景中，第二块8字节内存的前5个字节是可以访问，但剩下的3个字节是不能访问的。

所谓的shadow 内存就是在应用程序的虚拟地址空间中预留一段地址空间，用来存储映射应用程序访问的内存块中哪些字节可以被使用的信息，这些信息就是shadow 状态。其中每1个字节的shadow 内存，映射到8个字节的应用程序内存，因此，shadow状态可能有3种：

0: 表示映射的8个字节均可以使用

k(1<=k<=7): 表示表示映射的8个字节中只有前k个字节可以使用

负值: 表示映射的8个字节均不可使用，且不同的值表示所映射不同的内存类型（堆、栈、全局对象或已释放内存）

ASAN使用带有比例和偏移量的直接映射将应用程序地址转换为其对应的shadow内存地址：

shadow\_address = (addr >> 3) + offset

假设max - 1是虚拟地址空间中的最大有效地址，则offset的值应选择为在启动时不被占用的从offset到offset+Max/8的区域。

在32位linux系统中，虚拟地址空间为：

0x00000000-0xffffffff，offset = 0x20000000(2^29)。

在64位系统中，ofsset = 0x0000100000000000(2^44)。

在某些情况下（例如，在Linux上使用-fPIE/-pie 编译器标志）可以使用零偏移来进一步简化检测。

以下是32位linux系统中的地址空间分布

0x1 0000 0000 ---------------

| HIGH |

| MEMORY |

0x4000 0000 ---------------

| HIGH SHADOW |

0x2800 0000 ---------------

| BAD REGION |

0x2400 0000 ---------------

| LOW SHADOW |

0x2000 0000 ---------------

| LOW MEMORY |

0x0000 0000 ---------------

虚拟地址空间被划分为高低两部分，每个部分的内存地址映射到相应的shadow 内存。注意：将shadow 内存中的地址进行映射会得到Bad 区域中的地址，Bad 区域是被页面保护标记为不可访问的地址空间。

shadow映射方式可以推导为(addr >> scale) + offset的形式，其中scale是的取值范围是1～7，当 scale=N时，shadow 内存占用虚拟地址空间的1/2^N, red-zone的最小大小为2^N字节（保证malloc()的对齐要求）。shadow 内存中的每个字节描述了2^N个内存字节的状态并有2^N + 1个不同的值。

### redzone

ASAN会在应用程序使用的堆、栈、全局对象的内存周围分配额外内存，这个额外的内存叫做redzone，redzone会被shadow内存标记为不可使用状态，当应用程序访问redzone内存时说明已经溢出访问了，此时，ASAN检测redzone的shadow 状态后就会报告相应错误。readzone越大，检测内存下溢和上溢的范围越大。具体的分配策略将在下面涉及。

### 代码插桩

ASAN 会在应用程序访问内存的位置进行插桩，对于访问完整8字节内存的位置，插入以下代码检查内存对应的 shadow 内存，以此判断是否访问异常：

ShadowAddr = (Addr >> 3) + Offset;

if (\*ShadowAddr != 0)

ReportAndCrash(Addr);

由于应用程序访问8字节的内存，因此，其映射的shadow 内存的存储值必须是0，表示该8字节内存完全可用，否则，报错。

应用程序对 1、2、或者 4 字节内存的访问要复杂一些，如果访问的内存块对应的shadow 内存的存储值如果不是负数，且不为0，或者将要访问内存块超过了shadow 内存表示的可用范围，意味着本次将访问到不可使用的内存：

ShadowAddr = (Addr >> 3) + Offset;

k = \*ShadowAddr;

if (k != 0 && ((Addr & 7) + AccessSize > k))

ReportAndCrash(Addr);

需要注意的是，ASAN对源代码的插桩时机是在LLVM对代码编译优化之后，也就意味着ASAN只能检测 LLVM 优化后幸存下来的内存访问，例如：被 LLVM 优化掉的对栈对象进行访问的代码将不会被ASAN所识别。

同时，ASAN也不会对 LLVM 生成的内存访问代码进行插桩，例如：寄存器溢出检查等等。

另外，即使错误报告代码ReportAndCrash(Addr)只会被调用一次，但由于会在代码中的许多位置进行插入，因此，错误报告代码也必须相当紧凑。

目前 ASAN 使用了一个简单的函数调用来处理错误报告，当然还有另一个选择是插入一个硬件异常。

### 运行时库

在应用程序启动时，将映射整个shadow内存，因此程序的其他部分不能使用它。BAD区域也是受保护的，应用程序也不能访问。

在linux操作系统中，shadow内存区域不会被占用，因此映射总是成功的。但在MacOS中可能需要禁用地址空间布局（ASLR）。

另外，根据Google工程师介绍，shadow内存区域的布局也适用于windows操作系统。

启用ASAN时，源代码中的malloc和free函数将会被替换为运行时库中的malloc和free函数。

malloc分配的内存区域被组织为为一个与对象大小相对应的空闲列表数组。当对应于所请求内存大小的空闲列表为空时，从操作系统（例如，使用mmap）分配带有redzone的内存区域。n个内存块，将分配n+1个redzone：

| redzone-1 | memory-1 | redzone-2 | memory-2 | redzone-3 |

free函数会将整个内存区域置成不可使用并将其放入隔离区，这样该区域就不会马上被malloc分配给应用程序。

目前，隔离区是使用一个FIFO队列实现的，它在任何时候都拥有一定数量的内存。

默认情况下，malloc和free记录当前调用堆栈，以便提供更多信息的错误报告。malloc调用堆栈存储在左侧redzone中（redzone越大，可以存储的帧数越多），而free调用堆栈存储在内存区域本身的开头。

到这里你应该已经明白了对于动态分配的内存，ASAN是怎么实现检测的，但你可能会产生疑惑：动态分配是通过malloc函数分配redzone来支持错误检测，那栈对象和全局对象这类没有malloc分类内存的对象是怎么实现的呢？其实原理也很简单：

对于全局变量，redzone在编译时创建，redzone的地址在应用程序启动时传递给运行时库。运行时库函数会将redzone设置为不可使用并记录地址以供进一步错误报告。

对于栈对象，redzone是在运行时创建和置为不可使用。 目前，使用32字节的 redzone。例如以下代码片段：

void foo() {

char a[10];

<function body>

}

经ASAN处理后的代码大致如下：

void foo() {

char rz1[32]

char arr[10];

char rz2[32-10+32];

unsigned \* shadow = (unsigned\*)(((long)rz1>>8)+Offset);

// 将 redzone 设置为不可使用

shadow[0] = 0xffffffff; // rz1

shadow[1] = 0xffff0200; // arr and rz2

shadow[2] = 0xffffffff; // rz2

<function body>

// 将所有内存设置成可以使用

shadow[0] = shadow[1] = shadow[2] = 0;

}

### 总结

ASAN使用shadow内存和redzone来提供准确和即时的错误检测。

传统观点认为，shadow内存和redzone要么通过多级映射方案产生高开销，要么占用大量的程序内存。但，ASAN的使用的shadow映射机制和shadow状态编码减少了对内存空间占用。

最后，如果你觉得ASAN插桩代码和检测的对你某些的代码来说太慢了，那么可以使用编译器标志来禁用特定函数的，使ASAN跳过对代码中某个函数的插桩和检测，跳过分析函数的编译器指令是：

\_\_attribute\_\_((no\_sanitize\_address))

## 使用

现在只需要在编译命令中加上-fsanitize=address检测选项就可以让ASAN在你的项目中大展神通，接下来通过几个例子来看一下ASAN到底有哪些本领。

注意：

在测试中一般可以打开调试标志-g，这是当发现内存错误时调试符号可以帮助错误报告更准确的告知错误发生位置的堆栈信息，如果错误报告中的堆栈信息看起来不太正确，请尝试使用-fno-omit-frame-pointer来改善堆栈信息的生成情况。

如果构建代码时，编译和链接阶段分开执行，则必须在编译和链接阶段都添加-fsanitize=address选项。

# 对比