1 Valgrind

1.1. valgrind 相关基础知识

1.1.1. 什么是 valgrind

Valgrind 是一个开源的内存调试和性能分析工具，用于帮助开发者找出程序中的内存错误，如内存泄漏、使用未初始化的内存、非法内存访问等问题。它在 Linux 平台上广泛使用，并且支持多种处理器架构。

1.1.2. 为什么要学会 Valgrind

学习使用 Valgrind 是因为它是一个强大而实用的工具，可以帮助你在开发过程中发现和解决程序中的内存错误和性能问题。

- **发现内存错误：** Valgrind 的 Memcheck 工具可以帮助你检测内存错误，如使用未初始化的内存、内存泄漏、读写越界等。这些问题在程序运行时可能不会立即引发崩溃，但会导致程序的不稳定性和不可预测的行为。通过学习使用 Valgrind，你可以在开发过程中及时发现这些问题，从而提高代码质量。
- **提高程序稳定性：** 内存错误是导致许多程序崩溃和异常的主要原因之一。通过 Valgrind 的检测，你可以避免因为内存问题而导致程序崩溃，提高程序的稳定性和可靠性。
- **优化性能：** Valgrind 不仅可以检测内存错误，还包含其他工具用于性能分析，如 Cachegrind 和 Callgrind。通过这些工具，你可以了解程序的性能瓶颈和缓存使用情况，有助于优化代码，提高程序的执行效率。
- **提前发现问题：** 使用 Valgrind 可以帮助你在开发过程中尽早发现潜在问题，避免问题在生产环境中扩大化。这样可以减少调试和修复问题的时间，加快开发进度。
- **学习调试技巧：** Valgrind 输出的错误信息和报告可以帮助你学习调试技巧，了解如何分析和定位问题。这对于成为一名优秀的开发者是非常有益的。
- **开发最佳实践：** 学习使用 Valgrind 会使你养成良好的编码习惯，比如及时释放不再需要的内存、正确处理指针等。这些开发最佳实践有助于你编写更安全、更高效的代码。

1.1.3. 什么是内存泄露

在 Linux 应用程序中，内存泄漏通常发生在动态分配内存（例如使用 malloc、calloc、new 等）后忘记释放的情况。以下是一个简单的 C 语言代码示例，演示了在 Linux 应用中发生的内存泄漏：

```
#include <stdio.h>
#include <stdlib.h>

void memoryLeakExample() {
    // 在堆上分配一个整型数组
```

```
    int* myArray = (int*)malloc(100 * sizeof(int));

    // 忘记在不再需要时释放内存
    // free(myArray);
}

int main() {
    memoryLeakExample();

    // 在这里应该添加释放 myArray 的代码
    // free(myArray);

    return 0;
}
```

在这个例子中，memoryLeakExample() 函数分配了一个包含 100 个整型元素的数组，并且没有在函数结束后调用 free() 来释放该数组占用的内存。这就造成了内存泄漏，因为每次调用 memoryLeakExample() 都会分配一块新的内存，但没有释放，导致程序运行时占用的内存会逐渐增加，直到程序结束才会释放这部分内存。

在实际的应用程序中，内存泄漏可能涉及更复杂的数据结构和代码逻辑。定位和解决内存泄漏需要使用工具如 Valgrind 来分析应用程序的内存使用情况，从而找到哪里发生了内存泄漏并进行修复。通过检测和修复内存泄漏问题，可以确保应用程序在长时间运行后仍然能够保持稳定和可靠。

1.2. Valgrind 的移植

wget https://sourceware.org/pub/valgrind/valgrind-3.18.1.tar.bz2

```
book@100ask:~$ wget https://sourceware.org/pub/valgrind/valgrind-3.18.1.tar.bz2
--2023-07-19 22:17:49--  https://sourceware.org/pub/valgrind/valgrind-3.18.1.tar.bz2
Resolving sourceware.org (sourceware.org)... 8.43.85.97, 2620:52:3:1:0:246e:9693:128c
Connecting to sourceware.org (sourceware.org)|8.43.85.97|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16149159 (15M) [application/x-bzip2]
Saving to: 'valgrind-3.18.1.tar.bz2'

valgrind-3.18.1.tar.bz2                 100%[=======================================================================================>]  15.40M   175KB/s    in 2m 31s

2023-07-19 22:20:22 (104 KB/s) - 'valgrind-3.18.1.tar.bz2' saved [16149159/16149159]
```

tar xvf valgrind-3.18.1.tar.bz2

cd valgrind-3.18.1/

```
book@100ask:~/valgrind-3.18.1$ ls
aclocal.m4      COPYING             depcomp                                glibc-2.5.supp              Makefile.in             README                          solaris11.supp
AUTHORS         COPYING.DOCS        dhat                                   glibc-2.6.supp              Makefile.tool.am        README.aarch64                  solaris12.supp
autogen.sh      coregrind           docs                                   glibc-2.7.supp              Makefile.tool-tests.am  README.android                  tests
auxprogs        darwin10-drd.supp   drd                                    glibc-2.X-drd.supp          Makefile.vex.am         README.android_emulator         valgrind.pc.in
bionic.supp     darwin10.supp       exp-bbv                                glibc-2.X-drd.supp.in       Makefile.vex.in         README_DEVELOPERS               valgrind.spec
cachegrind      darwin11.supp       FAQ.txt                                glibc-2.X-helgrind.supp     massif                  README_DEVELOPERS_processes     valgrind.spec.in
callgrind       darwin12.supp       freebsd-drd.supp                       glibc-2.X-helgrind.supp.in  memcheck                README.freebsd                  VEX
compile         darwin13.supp       freebsd-helgrind.supp                  glibc-2.X.supp.in           missing                 README.mips                     vg-in-place
config.guess    darwin14.supp       freebsd.supp                           helgrind                    mpi                     README_MISSING_SYSCALL_OR_IOCTL xfree-3.supp
config.h        darwin15.supp       gdbserver_tests                        include                     musl.supp               README_PACKAGERS                xfree-4.supp
config.h.in     darwin16.supp       glibc-2.2-LinuxThreads-helgrind.supp   install-sh                  NEWS                    README.s390
config.sub      darwin17.supp       glibc-2.2.supp                         lackey                      NEWS.old                README.solaris
configure       darwin9-drd.supp    glibc-2.3.supp                         Makefile.all.am             none                    shared
configure.ac    darwin9.supp        glibc-2.4.supp                         Makefile.am                 perf                    solaris
```

./autogen.sh

```
book@100ask:~/valgrind-3.18.1$ ./autogen.sh
running: aclocal
running: autoheader
running: automake -a
running: autoconf
```

```
5992         { $as_echo "$as_me:${as_lineno-$LINENO}: result: ok (${host_cpu})" >&5
5993 $as_echo "ok (${host_cpu})" >&6; }
5994         ARCH_MAX="s390x"
5995         ;;
5996
5997       armv7* | arm)
```

./configure --host=arm-buildroot-linux-gnueabihf --target=arm-buildroot-linux-gnueabihf CC=arm-buildroot-linux-gnueabihf-gcc CPP=arm-buildroot-linux-gnueabihf-cpp CXX=arm-buildroot-linux-gnueabihf-c++ --prefix=/home/book/valgrind

配置成功：

```
config.status: executing depfiles commands

        Maximum build arch: arm
        Primary build arch: arm
      Secondary build arch:
                  Build OS: linux
    Link Time Optimisation: no
      Primary build target: ARM_LINUX
    Secondary build target:
          Platform variant: vanilla
     Primary -DVGPV string: -DVGPV_arm_linux_vanilla=1
        Default supp files: xfree-3.supp xfree-4.supp glibc-2.X-drd.supp glibc-2.X-helgrind.supp glibc-2.X.supp

book@100ask:~/valgrind-3.18.1$
```

make

```
mkdir -p ../.in_place; \
for f in ; do \
  rm -f ../.in_place/$f.dSYM; \
  ln -f -s ../mpi/$f.dSYM ../.in_place; \
done
make[2]: Leaving directory '/home/book/valgrind-3.18.1/mpi'
Making all in solaris
make[2]: Entering directory '/home/book/valgrind-3.18.1/solaris'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/book/valgrind-3.18.1/solaris'
Making all in docs
make[2]: Entering directory '/home/book/valgrind-3.18.1/docs'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/book/valgrind-3.18.1/docs'
make[1]: Leaving directory '/home/book/valgrind-3.18.1'
book@100ask:~/valgrind-3.18.1$
```

make install

```
make[4]: Leaving directory '/home/book/valgrind-3.18.1/docs'
make[3]: Leaving directory '/home/book/valgrind-3.18.1/docs'
make[2]: Leaving directory '/home/book/valgrind-3.18.1/docs'
make[1]: Leaving directory '/home/book/valgrind-3.18.1'
book@100ask:~/valgrind-3.18.1$ a
```

最后生成的文件位于：/home/book/valgrind 目录下。

```
book@100ask:~/valgrind$ ls
bin  include  lib  libexec  share
book@100ask:~/valgrind$ pwd
/home/book/valgrind
```

我们需要将 bin 文件中的 valgrind 通过网口上传到/usr/bin 目录下

```
[root@imx6ull:~]# ls  /usr/bin/valgrind
/usr/bin/valgrind
```

我们还需要将相关库文件放到/home/book/valgrind 目录下：

```
[root@imx6ull:/home/book/valgrind/lib/valgrind]# pwd
/home/book/valgrind/lib/valgrind
```

我们还需要将/home/book/valgrind/libexec/valgrind 中的文件也复制到
/home/book/valgrind/lib/valgrind 中。
此时我们运行 valgrind，若出现如下错误：

```
[root@imx6ull:/usr/bin]# ./valgrind
valgrind: failed to start tool 'memcheck' for platform 'arm-linux': No such file or directory
```

那就就需要指定 LIB 库的环境变量：

```
export VALGRIND_LIB=/home/book/valgrind/lib/valgrind
```

最后运行效果如下：

```
[root@imx6ull:~]# valgrind  -h
usage: valgrind [options] prog-and-args

  tool-selection option, with default in [ ]:
    --tool=<name>             use the Valgrind tool named <name> [memcheck]

  basic user options for all Valgrind tools, with defaults in [ ]:
    -h --help                 show this message
    --help-debug              show this message, plus debugging options
    --help-dyn-options        show the dynamically changeable options
    --version                 show version
    -q --quiet                run silently; only print error msgs
    -v --verbose              be more verbose -- show misc extra info
    --trace-children=no|yes   Valgrind-ise child processes (follow execve)? [no]
    --trace-children-skip=patt1,patt2,...    specifies a list of executables
                              that --trace-children=yes should not trace into
    --trace-children-skip-by-arg=patt1,patt2,...   same as --trace-children-skip=
                             but check the argv[] entries for children, rather
                              than the exe name, to make a follow/no-follow decision
    --child-silent-after-fork=no|yes omit child output between fork & exec? [no]
    --vgdb=no|yes|full        activate gdbserver? [yes]
                              full is slower but provides precise watchpoint/step
    --vgdb-error=<number>     invoke gdbserver after <number> errors [999999999]
                              to get started quickly, use --vgdb-error=0
                              and follow the on-screen directions
    --vgdb-stop-at=event1,event2,... invoke gdbserver for given events [none]
         where event is one of:
           startup exit valgrindabexit all none
    --track-fds=no|yes|all    track open file descriptors? [no]
                              all includes reporting stdin, stdout and stderr
    --time-stamp=no|yes       add timestamps to log messages? [no]
    --log-fd=<number>         log messages to file descriptor [2=stderr]
    --log-file=<file>         log messages to <file>
    --log-socket=ipaddr:port  log messages to socket ipaddr:port
```

1.3. Valgrind 的使用

直接运行 valgrind -h，查看到它的用法：

```
[root@imx6ull:~]# valgrind  -h
usage: valgrind [options] prog-and-args

  tool-selection option, with default in [ ]:
    --tool=<name>             use the Valgrind tool named <name> [memcheck]

  basic user options for all Valgrind tools, with defaults in [ ]:
    -h --help                 show this message
    --help-debug              show this message, plus debugging options
    --help-dyn-options        show the dynamically changeable options
    --version                 show version
    -q --quiet                run silently; only print error msgs
    -v --verbose              be more verbose -- show misc extra info
    --trace-children=no|yes   Valgrind-ise child processes (follow execve)? [no]
    --trace-children-skip=patt1,patt2,...    specifies a list of executables
                              that --trace-children=yes should not trace into
    --trace-children-skip-by-arg=patt1,patt2,...   same as --trace-children-skip=
                              but check the argv[] entries for children, rather
                              than the exe name, to make a follow/no-follow decision
    --child-silent-after-fork=no|yes omit child output between fork & exec? [no]
    --vgdb=no|yes|full        activate gdbserver? [yes]
                              full is slower but provides precise watchpoint/step
    --vgdb-error=<number>     invoke gdbserver after <number> errors [999999999]
                              to get started quickly, use --vgdb-error=0
                              and follow the on-screen directions
    --vgdb-stop-at=event1,event2,... invoke gdbserver for given events [none]
         where event is one of:
           startup exit valgrindabexit all none
    --track-fds=no|yes|all    track open file descriptors? [no]
                              all includes reporting stdin, stdout and stderr
    --time-stamp=no|yes       add timestamps to log messages? [no]
    --log-fd=<number>         log messages to file descriptor [2=stderr]
    --log-file=<file>         log messages to <file>
    --log-socket=ipaddr:port  log messages to socket ipaddr:port
```

```
user options for Valgrind tools that report errors:
  --xml=yes                 emit error output in XML (some tools only)
  --xml-fd=<number>         XML output to file descriptor
  --xml-file=<file>         XML output to <file>
  --xml-socket=ipaddr:port  XML output to socket ipaddr:port
  --xml-user-comment=STR    copy STR verbatim into XML output
  --demangle=no|yes         automatically demangle C++ names? [yes]
  --num-callers=<number>    show <number> callers in stack traces [12]
  --error-limit=no|yes      stop showing new errors if too many? [yes]
  --exit-on-first-error=no|yes exit code on the first error found? [no]
  --error-exitcode=<number> exit code to return if errors found [0=disable]
  --error-markers=<begin>,<end> add lines with begin/end markers before/after
                            each error output in plain text mode [none]
  --show-error-list=no|yes  show detected errors list and
                            suppression counts at exit [no]
  -s                        same as --show-error-list=yes
  --keep-debuginfo=no|yes   Keep symbols etc for unloaded code [no]
                            This allows saved stack traces (e.g. memory leaks)
                            to include file/line info for code that has been
                            dlclose'd (or similar)
  --show-below-main=no|yes  continue stack traces below main() [no]
  --default-suppressions=yes|no
                            load default suppressions [yes]
  --suppressions=<filename> suppress errors described in <filename>
  --gen-suppressions=no|yes|all    print suppressions for errors? [no]
  --input-fd=<number>       file descriptor for input [0=stdin]
  --dsymutil=no|yes         run dsymutil on Mac OS X when helpful? [yes]
  --max-stackframe=<number> assume stack switch for SP changes larger
                            than <number> bytes [2000000]
  --main-stacksize=<number> set size of main thread's stack (in bytes)
                            [min(max(current 'ulimit' value,1MB),16MB)]
```

1.3.1. Valgrind 相关工具

主要的 Valgrind 工具包括：

- **Memcheck**: Memcheck 是 Valgrind 中最常用的工具，用于检测内存错误。它可以检测到许多问题，例如使用未初始化的内存、内存泄漏、读写越界等。

- **Cachegrind**: Cachegrind 是用于缓存分析和性能优化的工具。它可以模拟处理器的缓存行访问，帮助开发者发现程序中缓存不友好的代码。
- **Callgrind**: Callgrind 是用于分析程序函数调用关系和性能的工具。它可以生成函数调用图，帮助开发者了解程序中各个函数之间的调用关系，从而进行性能优化。
- **Helgrind**: Helgrind 用于检测多线程程序中的并发错误，如竞争条件和死锁。
- **Massif**: Massif 用于检测程序的堆内存使用情况，帮助开发者发现内存泄漏和内存占用高的部分。

1.3.2. Valgrind 相关参数

Valgrind 是一款非常强大的开源工具，用于检测和调试 C、C++等编程语言中的内存泄漏、内存错误和线程错误。它可以帮助开发人员发现和修复潜在的内存问题，提高程序的稳定性和性能。以下是一些常用的 Valgrind 参数：

1. --tool=<toolname>: 指定要使用的 Valgrind 工具，常见的工具有：
- memcheck: 用于检查内存错误和内存泄漏，是最常用的工具。
- helgrind: 用于检查多线程程序中的竞争条件和同步问题。
- cachegrind: 用于缓存命中率分析，可帮助优化程序的缓存性能。
- massif: 用于检查堆内存的使用情况，帮助找出内存占用的高峰。
- callgrind: 用于生成程序调用图和函数调用关系，帮助分析函数调用开销。

2. --leak-check=<option>: 指定内存泄漏检查的级别，常见选项有：
- no: 关闭内存泄漏检查。
- summary: 在程序结束时显示简要的内存泄漏报告。
- full: 在程序结束时显示详细的内存泄漏报告，包含每个泄漏的内存块信息。
- yes: --leak-check=full 的简写形式。

3. --show-leak-kinds=<kind>: 指定显示哪些类型的内存泄漏，常见选项有：
- definite: 只显示明确的内存泄漏。
- indirect: 显示间接内存泄漏。
- all: 显示所有内存泄漏。

4. --track-origins=<yes|no>: 是否跟踪未初始化内存的来源。当开启此选项时，Valgrind 会报告未初始化内存的位置。

5. --tool=helgrind --ignore-race=<addr>: 在使用 helgrind 工具检测多线程程序时，可以忽略某些地址的竞争条件报告。

6. --suppressions=<file>: 指定一个包含 Valgrind 抑制规则的文件，用于抑制一些已知的、无害的错误报告。

7. --num-callers=<number>: 设置在报告函数调用堆栈时显示的调用者数量。

这些参数只是 Valgrind 中的一部分，Valgrind 还有其他一些参数和功能。使用时可以通过命令行传递这些参数，例如：

```
valgrind --tool=memcheck --leak-check=yes ./your_program
```

请注意，Valgrind 的使用可能会对程序执行速度产生影响，尤其是在使用 memcheck

工具时，因为它会对程序进行详细的内存访问跟踪。但它是一款非常有用的工具，能够帮助开发者发现隐藏的内存问题，因此在调试和优化阶段是非常有价值的。

1.3.3. 输出格式

Valgrind 运行中会输出一些文本信息，包括错误报告和其他重要事件。所有行均采用以下格式：

```
==12345== some-message-from-Valgrind
```

其中 12345 是 PID。采用这种格式可以把程序输出与 Valgrind 输出很好的区分开。默认情况下，Valgrind 只输出重要信息，如果需要输出更多细节，可以增加-v 选项。

1.3.4. 使用未初始化的内存案例

故障代码如下：

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *p;

    char c = *p;

    printf("\n [%c]\n",c);

    return 0;
}
```

在上面的代码中，我们尝试使用未初始化的指针 p。
我们对上面代码进行编译，将编译结果 a.out 上传至开发板。

```
book@100ask:~/demo$ arm-buildroot-linux-gnueabihf-gcc test1.c
```

让我们运行 Memcheck 来看下结果。

```
[root@imx6ull:/home/book]# valgrind --tool=memcheck ./a.out
==481== Memcheck, a memory error detector
==481== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==481== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==481== Command: ./a.out
==481==
==481== Use of uninitialised value of size 4
==481==    at 0x10420: main (in /home/book/a.out)
==481==
==481== Invalid read of size 1
==481==    at 0x10420: main (in /home/book/a.out)
==481==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==481==
```

从上面的输出可以看到,Valgrind 检测到了未初始化的变量

1.3.5. 内存泄露

故障代码如下：

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *p = malloc(1);
    *p = 'a';

    char c = *p;

    printf("\n [%c]\n",c);

    return 0;
}
```

我们对上面代码进行编译，将编译结果 a.out 上传至开发板。

```
book@100ask:~/demo$ arm-buildroot-linux-gnueabihf-gcc test2.c
```

在这次的代码中，我们申请了一个字节但是没有将它释放.现在让我们运行 Valgrind 看看会发生什么：

```
[root@imx6ull:/home/book]# valgrind --tool=memcheck --leak-check=full ./a.out
==495== Memcheck, a memory error detector
==495== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==495== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==495== Command: ./a.out
==495==

 [a]
==495==
==495== HEAP SUMMARY:
==495==     in use at exit: 1 bytes in 1 blocks
==495==   total heap usage: 2 allocs, 1 frees, 4,097 bytes allocated
==495==
==495== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==495==    at 0x483FE64: malloc (vg_replace_malloc.c:381)
==495==    by 0x10457: main (in /home/book/a.out)
==495==
==495== LEAK SUMMARY:
==495==    definitely lost: 1 bytes in 1 blocks
==495==    indirectly lost: 0 bytes in 0 blocks
==495==      possibly lost: 0 bytes in 0 blocks
==495==    still reachable: 0 bytes in 0 blocks
==495==         suppressed: 0 bytes in 0 blocks
==495==
==495== For lists of detected and suppressed errors, rerun with: -s
==495== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

1.3.6. 在内存被释放后进行读/写案例

故障代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    char *p = malloc(1);
    *p = 'a';
```

```
    char c = *p;

    printf("\n [%c]\n",c);

    free(p);
    c = *p;
    return 0;
}
```

我们对上面代码进行编译，将编译结果 a.out 上传至开发板。

```
book@100ask:~/demo$ arm-buildroot-linux-gnueabihf-gcc test3.c
```

上面的代码中，我们有一个释放了内存的指针‘p’然后我们又尝试利用指针获取值。让我们运行 memcheck 来看一下 Valgrind 对这种情况是如何反应的。

```
[root@imx6ull:~]# valgrind --tool=memcheck ./a.out
==438== Memcheck, a memory error detector
==438== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==438== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==438== Command: ./a.out
==438==

 [a]
==438== Invalid read of size 1
==438==    at 0x104CC: main (in /root/a.out)
==438==  Address 0x494c028 is 0 bytes inside a block of size 1 free'd
==438==    at 0x4843220: free (vg_replace_malloc.c:872)
==438==    by 0x104C7: main (in /root/a.out)
==438==  Block was alloc'd at
==438==    at 0x483FE64: malloc (vg_replace_malloc.c:381)
==438==    by 0x1048B: main (in /root/a.out)
==438==
==438==
==438== HEAP SUMMARY:
==438==     in use at exit: 0 bytes in 0 blocks
==438==   total heap usage: 2 allocs, 2 frees, 1,025 bytes allocated
==438==
==438== All heap blocks were freed -- no leaks are possible
==438==
==438== For lists of detected and suppressed errors, rerun with: -s
==438== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

从上面的输出内容可以看到，Valgrind 检测到了无效的读取操作然后输出了警告‘Invalid read of size 1’.

1.3.7. 从已分配内存块的尾部进行读/写案例

故障代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    char *p = malloc(1);
    *p = 'a';

    char c = *(p+1);

    printf("\n [%c]\n",c);

    free(p);
```

```
    return 0;
}
```

我们对上面代码进行编译，将编译结果 a.out 上传至开发板。

```
book@100ask:~/demo$ arm-buildroot-linux-gnueabihf-gcc test4.c
```

在上面的代码中，我们已经为‘p’分配了一个字节的内存,但我们在将值读取到 ‘c’中的时候使用的是地址 p+1.

现在我们使用 Valgrind 运行上面的代码 ：

```
[root@imx6ull:~]# valgrind --tool=memcheck ./a.out
==447== Memcheck, a memory error detector
==447== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==447== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==447== Command: ./a.out
==447==
==447== Invalid read of size 1
==447==    at 0x104A4: main (in /root/a.out)
==447==  Address 0x494c029 is 0 bytes after a block of size 1 alloc'd
==447==    at 0x483FE64: malloc (vg_replace_malloc.c:381)
==447==    by 0x1048B: main (in /root/a.out)
==447==

 []
==447==
==447== HEAP SUMMARY:
==447==     in use at exit: 0 bytes in 0 blocks
==447==   total heap usage: 2 allocs, 2 frees, 1,025 bytes allocated
==447==
==447== All heap blocks were freed -- no leaks are possible
==447==
==447== For lists of detected and suppressed errors, rerun with: -s
==447== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

同样，该工具在这种情况下也检测到了无效的读取操作.

1.3.8. 两次释放内存案例

故障代码如下：

```
#include <stdio.h>#include <stdlib.h> int main(void){
    char *p = (char*)malloc(1);
    *p = 'a';

    char c = *p;
    printf("\n [%c]\n",c);
    free(p);
    free(p);
    return 0;}
```

我们对上面代码进行编译，将编译结果 a.out 上传至开发板。

```
book@100ask:~/demo$ arm-buildroot-linux-gnueabihf-gcc test5.c
```

在上面的代码中，我们两次释放了'p'指向的内存. 现在让我们运行 memcheck :

```
[root@imx6ull:~]# valgrind --tool=memcheck ./a.out
==461== Memcheck, a memory error detector
==461== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==461== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==461== Command: ./a.out
==461==

 [a]
==461== Invalid free() / delete / delete[] / realloc()
==461==    at 0x4843220: free (vg_replace_malloc.c:872)
==461==    by 0x104CF: main (in /root/a.out)
==461==  Address 0x494c028 is 0 bytes inside a block of size 1 free'd
==461==    at 0x4843220: free (vg_replace_malloc.c:872)
==461==    by 0x104C7: main (in /root/a.out)
==461==  Block was alloc'd at
==461==    at 0x483FE64: malloc (vg_replace_malloc.c:381)
==461==    by 0x1048B: main (in /root/a.out)
==461==
==461==
==461== HEAP SUMMARY:
==461==     in use at exit: 0 bytes in 0 blocks
==461==   total heap usage: 2 allocs, 3 frees, 1,025 bytes allocated
==461==
==461== All heap blocks were freed -- no leaks are possible
==461==
==461== For lists of detected and suppressed errors, rerun with: -s
==461== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

该功能检测到我们对同一个指针调用了两次释放内存操作。

1.4. Valgrind 的工作原理

Valgrind 是一个开源的内存调试、内存泄漏检测和性能分析工具。它的原理是通过创建一个虚拟的执行环境，将目标程序运行在这个虚拟环境中，并在运行过程中对程序的执行进行监控和干预。Valgrind 的原理可以简要概括为以下几点：

- 代码重写：Valgrind 利用"动态二进制重写"技术，将目标程序的机器代码进行重写，将其替换成自己的代码。这样 Valgrind 可以在目标程序的运行过程中进行监控和干预。
- 虚拟机：Valgrind 创建一个虚拟的执行环境，称为"Valgrind VM"。目标程序在这个虚拟环境中运行，Valgrind 在虚拟环境中模拟 CPU 的执行。
- 指令解释：Valgrind 对目标程序的每一条指令进行解释执行，而不是直接在物理 CPU 上执行。这样 Valgrind 能够捕捉每一条指令的执行过程，并进行监控。
- 内存跟踪：Valgrind 利用"影子内存"技术来跟踪目标程序的内存操作。它为目标程序的每一个字节都维护一个影子内存空间，记录每个字节的内存状态。
- 内存检查：在目标程序的执行过程中，Valgrind 会监控所有的内存读写操作。如果目标程序试图读取未初始化的内存、访问已经释放的内存或者越界访问内存，Valgrind 就会检测到并报告。
- 内存泄漏检测：Valgrind 还会跟踪目标程序中未释放的内存块。如果某个内存块在目标程序结束时没有被释放，Valgrind 将会检测到并生成泄漏报告。
- 错误报告：当 Valgrind 检测到内存问题时，它会输出相关的错误报告，告诉开发人员哪些地方出现了内存错误，以及如何定位问题。

通过这种方式，Valgrind 能够在非常细粒度的层面对目标程序进行监控，帮助开发人员找到内存错误和性能问题，并进行调试和优化。但由于 Valgrind 的原理是通过解释执行和虚拟化，它会对程序的执行速度产生一定的影响，因此在进行性能测试时，不宜使用 Valgrind，而是在调试和测试阶段使用。