Presented by undo

Valgrind Quick Reference Guide

VALGRIND MEMCHECK

$ valgrind path/to/myprog myargs
Use Valgrind Memcheck to detect common memory errors in myprog.

$ valgrind --leak-check=yes path/to/myprog
Use Valgrind Memcheck to detect memory errors and memory leaks.

$ valgrind --leak-check=yes --track-origins=yes myprog
Take longer and trace the origin of uninitialized values.

$ valgrind --track-fds=yes myprog
Detect unclosed file descriptors.

$ valgrind --xtree-memory=full --leak-check=yes myprog
Produce a xtmemory.kcg file. Install KCachegrind to examine it. It shows a visual backtrace of places in the code that leaked memory.

$ valgrind --error-exitcode=1 myprog
Run silently. Return a failure exit code if errors found, rather than myprog's exit code. Useful in automated tests.

VALGRIND HELGRIND

$ valgrind --tool=helgrind --free-is-write=yes myprog
Use Valgrind Helgrind to detect common threading errors in myprog.

$ valgrind --tool=helgrind --track-lockorders=yes prog
Also detect potential deadlocks. Can be verbose for many programs.

GENERAL VALGRIND TIPS

- Output has no line numbers? Have build system compile with -g.
- Programs run 10-50x slower. Test with small workloads.
- Too many errors? Fix the first errors, that may remove further errors.
- Check out the **--vgdb-error** and **--vgdb-stop-at** options for using GDB to step through the program.

VALGRIND SUPPRESSION FILES

$ valgrind --suppressions=myerrors.supp [--tool=...]
Ignore errors of given types and locations in file mysuppressions.supp.

$ valgrind --gen-suppressions=yes [--tool=...]
Print a suppression for each error, for copying to a suppression file.

myerrors.supp

```
{
 __gconv_transform_ascii_internal/__mbrtowc/mbtowc
   Name identifying this entry.
 Memcheck:Value4
   Error given by Memcheck tool. Uninitialized value of width 4 bytes.
 fun:__gconv_transform_ascii_internal
 fun:__mbr*toc
 fun:mbtowc
   Call stack matches these three functions in order. Note the wildcard.
}
Example entry in a suppression file. It applies if all the conditions are met.
```

Valgrind also ignores known errors in system libraries on many systems. On others you may need a lot of suppression wildcards for library errors.

Valgrind Quick Start: https://valgrind.org/docs/manual/quick-start.html
Helgrind Manual: https://valgrind.org/docs/manual/hg-manual.html

Presented by undo

Valgrind Quick Reference Guide

MEMCHECK ERRORS

```
Conditional jump or move depends on uninitialised
value(s)
   at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
   by 0x402E8476: _IO_printf (printf.c:36)
   by 0x8048472: main (tests/manuel1.c:8)
```

Use of an uninitialized variable. Memcheck prints the backtrace where the value was used. **`--track-origins=yes`** *can find where it came from.*

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   ...
 Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

Read from memory which is not allocated. In this case in unused stack memory below the stack. Often Memcheck can say "in freed memory" etc.

Other common errors detected are: Invalid pointers in system calls. Double **`free`**s. Mixing **`new`**/**`free`**. Overlapping **`memcpy`**. **`realloc(0)`**.

TYPES OF MEMORY LEAK

- ***Reachable*** Not leaked, but not deallocated before exit
- ***Lost*** Pointer deallocated without deallocating memory
- ***Possibly Lost*** Some part can be reached but not the start of the memory

MEMCHECK QUICK TIPS

- Run until the program exits. Memcheck detects many leaks at the end.
- Test an optimised build first. An unoptimized build has different errors.
- Use custom memory management? Include **`valgrind/valgrind.h`** and use vanilla malloc and free if **`RUNNING_ON_VALGRIND`** macro is true.
- Use custom alloc functions? Tell valgrind about them with **`VALGRIND_MALLOCLIKE_BLOCK`** + **`VALGRIND_FREELIKE_BLOCK`**

HELGRIND ERRORS

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
   at 0x4C2408D: pthread_mutex_unlock
   by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
   by 0x40079B: main (tc09_bad_unlock.c:50)
  Lock at 0x7FEFFFA90 was first observed
   at 0x4C25D01: pthread_mutex_init
   by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
   by 0x40079B: main (tc09_bad_unlock.c:50)
```

Lock was unlocked without first being locked. Helgrind shows where it originated.

```
Thread #1 is the program's root thread

Thread #2 was created
...
Possible data race during read of size 4 at 0x601038 by
thread #1
Locks held: none
   at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread
#2
...
Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```

Possible race. Helgrind shows the backtrace for both threads' accesses

Helgrind also detects pthreads API misuses, as well as races and deadlocks.

HELGRIND QUICK TIPS

- Use **`pthread`**s best practices or get many errors reported.
- If you write your own thread functions, or alloc functions that reuse a pool of buffers, identify them with **`helgrind.h`** macros.