Debugging: Love It, Hate It Or Reverse It?

Debugging: Love It, Hate It Or Reverse It?.

Julian Smith, co-founder and CTO, Undo.
jsmith@undo.io
http://undo.io/

Overview

- Testing.
- Debugging:
  - Debugging with gdb.
  - Strace.
  - Valgrind.
  - Recording execution.

(Linux-specific.)

Testing.

Testing has changed:

- Continuous integration.
- Test-driven development.
- Cloud testing.

Resulting in:

- 1,000s of tests per hour.
- Many intermittent test failures.
- Very difficult to fix them all.

Testing.

- Security breaches.
- Production outages.
- Unhappy users.

Testing.

Fixing test failures is hard.

- Recreate complex setups:
  - Multi-application.
  - Networking.
  - Multi-machine.
- Re-run flakey tests many times to reproduce failure
- Recompile/link with changes when investigating.
  - Changes behaviour.
  - Slow.
  - Requires a developer machine.

Testing

Fixing test failures is slow.

- Reproducing slow failures is… slow.
- Reproducing intermittent failures is also slow.
  - Requires repeatedly running a test many times in order to catch the failure.

Critical bugs:

- Can occur one in a thousand runs.
- Each run can take hours.

Testing.

Tools to fix test failures

- Debuggers.
- Logging.
- System logging.
- Memory checkers.
- Recording execution.

Debugging.

GDB

- Better than you may remember.
- Ctrl-X Ctrl-A shows source code within terminal window.
- GDB-7 has python extension.
  - Scripted debugging, e.g. to reproduce intermittent failures.

Debugging.

GDB scripting.

repeat_until_non_zero_exit.py

```
'''
Repeatedly run debuggee until it fails.
'''
import gdb

while 1:
    print '-' * 40
    gdb.execute( 'run')
    e = gdb.parse_and_eval( '$_exitcode')
    print( '$_exitcode is: %s' % e)
    if e != 0:
        break
```

```
(gdb) source repeat_until_non_zero_exit.py
```

Debugging.

GUIs for gdb are getting better:

- Eclipse.
- CLion.
- Qt Creator.
- KDbg.
- Emacs.

Logging.

- Can sometimes work well.
- Need to control what to log.
  - Define areas of functionality and assign different debug levels.
  - E.g. *parser, lexer, network*.
  - More detailed: *memory allocator, socket, serialiser*.
- We can define debug levels for different categories to match the bug we are investigating.

This can get complicated.

```
logcategory_t*  io_category = ...;
logcategory_t*  serialisation_category = ...;
...
logf( io_category, "have read %zi bytes from socket fd=%i", n, fd);
...
logf( serialisation_category, "serialised %p to %zi bytes", foo, actualsize);
...
```

Logging.

Problems with logging categories.

How many categories - how detailed should we go?

- Depends on the bug we are investigating.
- May need to recompile with new categories.

What category do we use for code that writes serialised data to a file - `io_category` or `serialisation_category` ?

Logging.

Use programme structure for categories.

We already have *areas of functionality*:

- Source code directories.
- Source files.
- Functions.

We can use these as implicit categories:

- No need to define our own categories.
- We get different levels of categories for free.
- We get nested categories for free.

Logging.

Controlling verbosity programmatically:

```
debug_add( "network/socket", NULL, 1);
// Extra verbose for all diagnosics in network/socket*.*.
```

```
debug_add( "network/", NULL, 1);
debug_add( "network/socket", NULL, 1);
// Extra verbose for all diagnostics in network/*.*.
// Even more verbose in network/socket*.*.
```

```
debug_add( "heap/alloc.c", "", 1);
debug_add( "network/socket.c", Send, 2);
debug_add( "parser/", "", -1);
// Verbose for heap operations.
// Very verbose for all diagnostics in network/socket.c:Send().
// Less verbose in parser/.
```

Logging.

Control verbosity with environmental variables:

- QA-friendly.
- No need to recompile/link/build.
- Activate logging in different parts of the programme depending on the bug which is being investigated.

Example:

```
DEBUG_LEVELS=heap/alloc.c=1,parser/=-1,network/socket.c:Send=2 myprog ...
```

Strace.

Linux/unix-specific.

- Get a detailed log of all syscalls.

```
> strace date
execve("/bin/date", ["date"], [/* 34 vars */]) = 0
brk(0)                                  = 0xd50000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7602059000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3</etc/ld.so.cache>
fstat(3</etc/ld.so.cache>, {st_mode=S_IFREG|0644, st_size=144491, ...}) = 0
mmap(NULL, 144491, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f7602035000
close(3</etc/ld.so.cache>)              = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3</lib/x86_64-linux-gnu/libc-2.19.so>
read(3</lib/x86_64-linux-gnu/libc-2.19.so>, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\34\2\0\0\0\0\0"...,
832) = 832
fstat(3</lib/x86_64-linux-gnu/libc-2.19.so>, {st_mode=S_IFREG|0755, st_size=1738176, ...}) = 0
mmap(NULL, 3844640, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-linux-gnu/libc-2.19.so>, 0) =
0x7f7601a90000
mprotect(0x7f7601c32000, 2093056, PROT_NONE) = 0
mmap(0x7f7601e31000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3</lib/x86_64-linux-
gnu/libc-2.19.so>, 0x1a1000) = 0x7f7601e31000
mmap(0x7f7601e37000, 14880, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7601e37000
close(3</lib/x86_64-linux-gnu/libc-2.19.so>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7602034000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7602033000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7602032000
arch_prctl(ARCH_SET_FS, 0x7f7602033700) = 0
mprotect(0x7f7601e31000, 16384, PROT_READ) = 0
mprotect(0x60e000, 4096, PROT_READ)     = 0
mprotect(0x7f760205b000, 4096, PROT_READ) = 0
munmap(0x7f7602035000, 144491)          = 0
brk(0)                                  = 0xd50000
brk(0xd71000)                           = 0xd71000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3</usr/lib/locale/locale-archive>
fstat(3</usr/lib/locale/locale-archive>, {st_mode=S_IFREG|0644, st_size=1607760, ...}) = 0
mmap(NULL, 1607760, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) = 0x7f7601ea9000
close(3</usr/lib/locale/locale-archive>) = 0
open("/etc/localtime", O_RDONLY|O_CLOEXEC) = 3</etc/localtime>
fstat(3</etc/localtime>, {st_mode=S_IFREG|0644, st_size=3661, ...}) = 0
fstat(3</etc/localtime>, {st_mode=S_IFREG|0644, st_size=3661, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7602058000
read(3</etc/localtime>, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\7\0\0\0\7\0\0\0\0"..., 4096) = 3661
lseek(3</etc/localtime>, -2338, SEEK_CUR) = 1323
read(3</etc/localtime>, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\10\0\0\0\10\0\0\0\0"..., 4096) = 2338
close(3</etc/localtime>)                = 0
munmap(0x7f7602058000, 4096)            = 0
fstat(1</dev/pts/50>, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 50), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7602058000
write(1</dev/pts/50>, "Mon 26 Sep 12:27:50 BST 2016\n", 29Mon 26 Sep 12:27:50 BST 2016
) = 29
close(1</dev/pts/50>)                   = 0
munmap(0x7f7602058000, 4096)            = 0
close(2</dev/pts/50>)                   = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

- Subset of syscalls - file operations:

```
> strace -y -e trace=file date
execve("/bin/date", ["date"], [/* 34 vars */]) = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3</etc/ld.so.cache>
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3</lib/x86_64-linux-gnu/libc-2.19.so>
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3</usr/lib/locale/locale-archive>
open("/etc/localtime", O_RDONLY|O_CLOEXEC) = 3</etc/localtime>
Mon 26 Sep 12:29:01 BST 2016
+++ exited with 0 +++
```

- Subset of syscalls - memory operations:

```
> strace -y -e trace=memory date
brk(0)                                  = 0x25b8000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f14cc871000
mmap(NULL, 144491, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f14cc84d000
mmap(NULL, 3844640, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-linux-gnu/libc-2.19.so>, 0) =
```

```
0x7f14cc2a8000
mprotect(0x7f14cc44a000, 2093056, PROT_NONE) = 0
mmap(0x7f14cc649000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3</lib/x86_64-linux-
gnu/libc-2.19.so>, 0x1a1000) = 0x7f14cc649000
mmap(0x7f14cc64f000, 14880, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f14cc64f000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f14cc84c000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f14cc84b000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f14cc84a000
mprotect(0x7f14cc649000, 16384, PROT_READ) = 0
mprotect(0x60e000, 4096, PROT_READ)     = 0
mprotect(0x7f14cc873000, 4096, PROT_READ) = 0
munmap(0x7f14cc84d000, 144491)          = 0
brk(0)                                  = 0x25b8000
brk(0x25d9000)                          = 0x25d9000
mmap(NULL, 1607760, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) = 0x7f14cc6c1000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f14cc870000
munmap(0x7f14cc870000, 4096)            = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f14cc870000
Mon 26 Sep 12:29:40 BST 2016
munmap(0x7f14cc870000, 4096)            = 0
+++ exited with 0 +++
```

Strace.

Summary:

- Not perfect - only works on syscall level.
- But still useful for low-level investigations.
- No recompilation required.

Valgrind.

Overview:

- Linux, OS X, Solaris, Android.
- Very detailed checking of execution.
- Free.
- Similar to Purify etc.

Valgrind.

Memory checking:

- Illegal memory accesses:
  - Overrun/under run heap blocks.
  - Overrun stack.
  - Use-after-free.
- Double free.
- Memory leaks.

Thread checking.

- Inconsistent lock orderings.
- Data races (e.g. missing mutex).

Other:

- CPU cache behaviour.
- Heap profiler.

Valgrind.

Highly recommended!

Recording execution.

New debugging technology in last few years.

- Linux:
  - Undo Live Recorder.
  - RR.
- Windows:
  - Intellitrace (partial recording only).
  - TimeMachineFor.Net (partial recording only).
- Java:
  - Chronon.
  - Undo (soon).

Live Recorder.

- A library, for linking into an application.
- Allows the application to control the recording of its own execution.
- Provides a simple C API to start/save/stop recording.
- API is defined in `undolr.h` header file and implemented in `libundolr` library.

Live Recorder.

Live Recorder recordings:

- Are standard Undo Recording files.
- Contain everything need to replay execution:
  - Non-deterministic events (inputs to program).
  - Initial state (initial memory and registers).
- Also contain information needed for symbolic debugging:
  - Complete executable and `.so` files.
  - Debuginfo files.
- Allows debugging even when libraries and/or debug information is not available locally (e.g. load and replay on a different Linux distribution).
- Loaded into UndoDB as with Save-Load:
  - `undodb-gdb --undodb-load <filename>`
  - `(undodb-gdb) undodb-load <filename>`
- Full reversible debugging.

Live Recorder.

Library API (`undolr.h`):

```
int undolr_recording_start( undolr_error_t* o_error);

int undolr_recording_stop( void);

int undolr_recording_save( const char* filename);

int undolr_recording_stop_and_save( const char* filename);

int undolr_save_on_termination( const char* filename);

int undolr_save_on_termination_cancel( void);

int undolr_event_log_size_get( long* o_bytes);

int undolr_event_log_size_set( long bytes);

int undolr_include_symbol_files( int include);
```

Live Recorder.

Use Live Recorder in internal testing:

- Investigate test failures easily using reversible debugging.
- Avoid problems with differing environments.
- No need to reproduce complex multi-machine setups.
- Can be used in different ways:
  - Disabled by default, but re-run failing tests with Live Recorder activated.
  - Enabled by default, but tell Live Recorder to save recording only if test fails.
- Have multiple developers work on the same test failure.

Live Recorder.

Use Live Recorder in customer releases:

- No overhead if not used.
- You and your customer control when/if recording is enabled.
- Customer has control over pruning the recording to protect their IP.
- Debug an exact copy of a customer failure, without having to create a test-case.
- Have multiple developers work on the same customer bug.

Live Recorder: Demo.

Live Recorder.

Questions?

EOF.

http://undo.io/