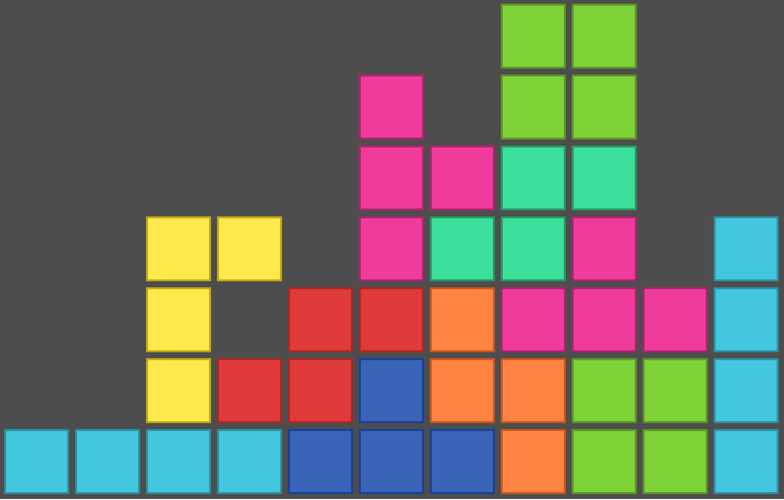
Unit testing and mocking with cmocka

devconf.cz 2020

Andreas Schneider
Principal Software Engineer



About me

Source Code Artist working on:

- Samba - The domain controller and file server
- libssh - The SSH Library
- cmocka - a unit testing framework for C
- cwrap - Client/Server testing made easy
- darktable - image raw developer
- LineageOS - Android with Privacy Features

The talk will cover:

- What is cmocka?
- What features does cmocka provide?
- What is mocking?
- How to write a mocking test?

1

What is cmocka?

cmocka ...

- is an elegant unit testing framework for C
- it only requires the standard C library
- **offers support for mock objects**.

cmocka ...

works on a range of computing platforms (including embedded) and works with different compilers.

Linux/BSD/Windows - GCC/Clang/MSVC

Mission Statement

The goal of this project is to provide a powerful testing framework for C, on different platforms and operating systems, which only requires the standard C library.

It has a website

cmocka.org

2

Features of cmocka

Test fixtures and groups

Test fixtures are setup and teardown functions that can be shared across multiple test cases to provide common functions that prepare the test environment and destroy it afterwards. This is also supported for groups.

Exception handling

- cmocka is able to recover the test state if there are exceptions like a segfault.
- Handling for `SIGSEGV`, `SIGILL`, etc.
- An attached debugger will stop when the segfault occurs

Exception handling

cmocka doesn't use `fork()` for exception handling in test cases!

- `fork()` is not available on all platforms
- `fork()` is implemented diffrently on some OSes (Linux vs. MacOSX)

Output formats

cmocka has it's own console output format, but supports additional message formats like:

- Test Anything Protocol
- Subunit (used by Samba)
- xUnit XML (parsed by Jenkins)

A cmocka test

```
#include <stdarg.h>
#include <stddef.h>
#include <sdtint.h>
#include <setjmp.h>
#include <cmocka.h>
```

A cmocka test

```c
/* A test case that does nothing and succeeds. */
static void null_test_success(void **state) {
    (void) state; /* unused */
}
```

A cmocka test

```
int main(void) {
    const struct CMUnitTest tests[] = {
        cmocka_unit_test(null_test_success),
    };

    return cmocka_run_group_tests(tests, NULL, NULL);
}
```

Assert functions

We have a lot of assert functions for …

Booleans

```
assert_true(x)
assert_false(x)
```

Integers

```
assert_int_equal(a, b)
assert_int_not_equal(a, b)
```

Integer ranges

```
assert_in_range(value, minimum, maximum)
assert_not_in_range(value, minimum, maximum)
```

Floats

```
assert_float_equal(a, b)
assert_float_not_equal(a, b)
```

Pointers

```
assert_non_null(x)
assert_null(x)
```

Return codes

```
assert_return_code(rc, errno)
```

Strings

```
assert_string_equal(a, b)
assert_string_not_equal(a, b)
```

Memory comparison

```
assert_memory_equal(a, b)
assert_memory_not_equal(a, b)
```

.... and a lot more

API Documentation

api.cmocka.org

A cmocka test with an assert

```c
/* A test case that compare intetgers and will fail. *
static void integer_failure(void **state) {
    int i = 4;

    assert_int_equal(i, 5);
}
```

Extending assert functions

You can also easily extend cmocka by writing special assert functions for your project.

Example: socket_wrapper tests offer:

```
assert_sockaddr_equal(ss, a)
assert_sockaddr_port_equal(ss, a, prt)
```

Testing assert() of libc

If you use libc **assert()** in your code, you can redefine **assert()** and test it!

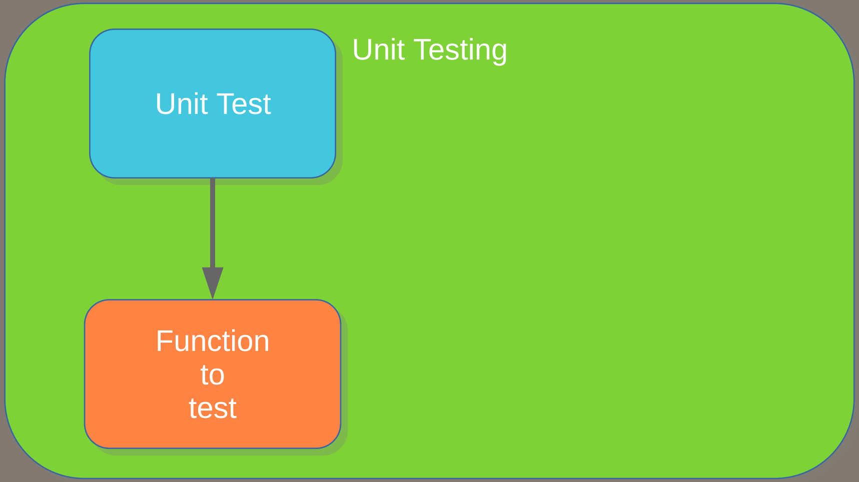
```
#define assert mock_assert
void showmessage(const char *message) {
  assert(message);
}

int main(void) {
  expect_assert_failure(show_message(NULL));
  printf("succeeded\n");
  return 0;
}
```

3

Mocking in unit tests

Standard unit test


Unit Testing
Unit Test
Function to test

An example

Lets write a test for 'uptime'

```
./example/uptime/uptime
up 3 days, 24 minutes
```

Source code be found here.

Uptime

consists of two functions

- `calculate_uptime()`
- `read_proc_uptime()`

Uptime

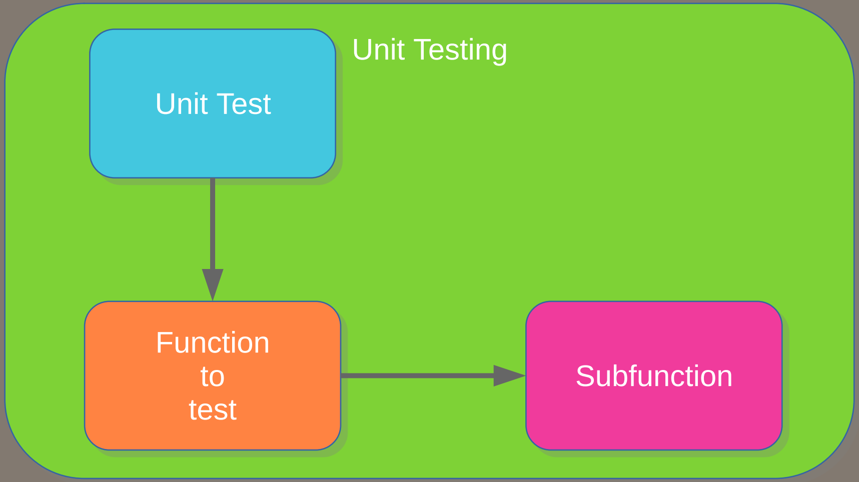
`calculate_uptime()` calls `read_proc_uptime()`

`read_proc_uptime()` reads to doubles from `/proc/uptime`

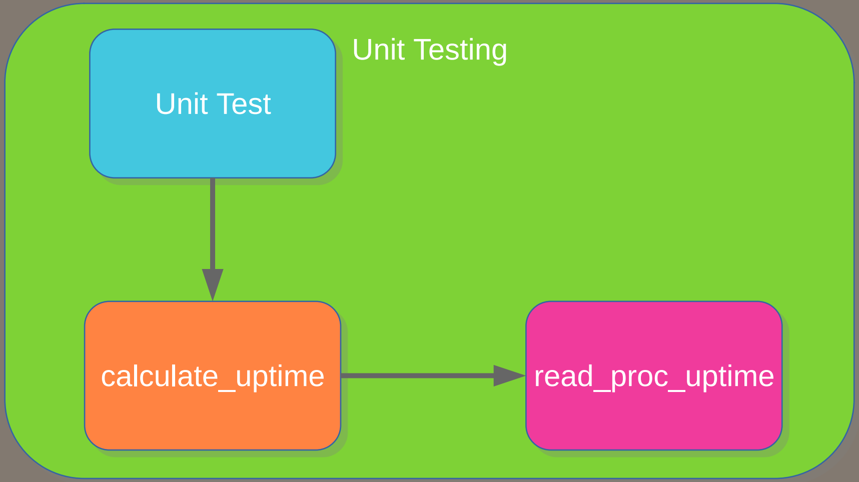
```
krikkit:~ # cat /proc/uptime
436821.10 1066410.33
```

`calculate_uptime()` poduces a human readable form out of those two doubles.

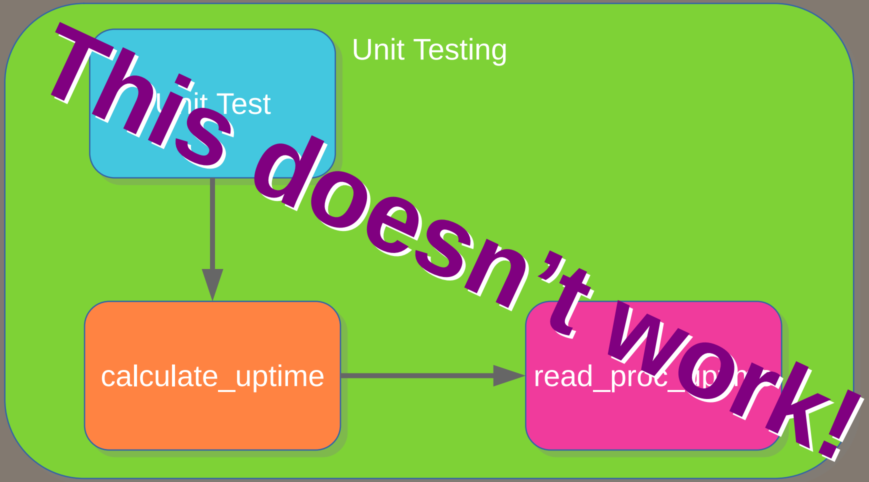
Unit test with a subfunction


Unit Testing
Unit Test
Function to test
Subfunction

Uptime example


Unit Testing
Unit Test
calculate_uptime
read_proc_uptime

Uptime example


Unit Testing
Unit Test
calculate_uptime
read_proc_uptime
This doesn't work!

Why?

- `/proc/uptime` constantly ticks!

Solution: We need mocking!

What is mocking?

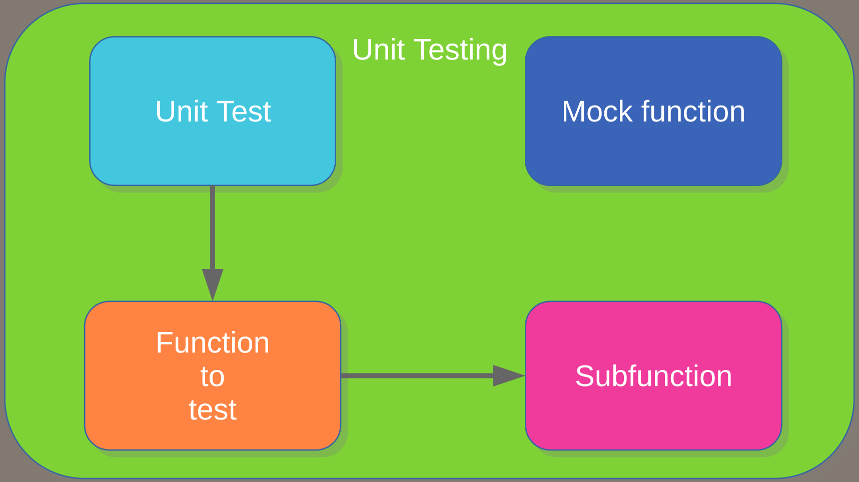
Mocking is a way to create instrumented objects that simulate the behavior of real objects.

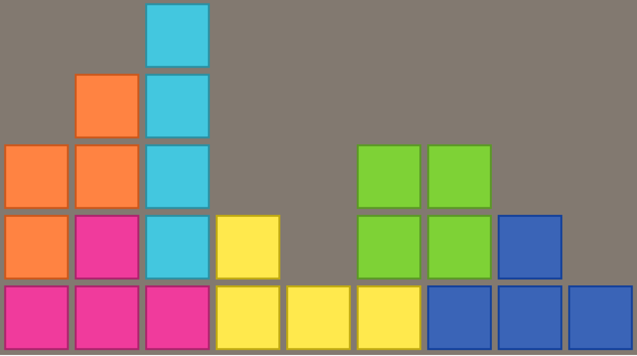
What is mocking?

to mock = to imitate something

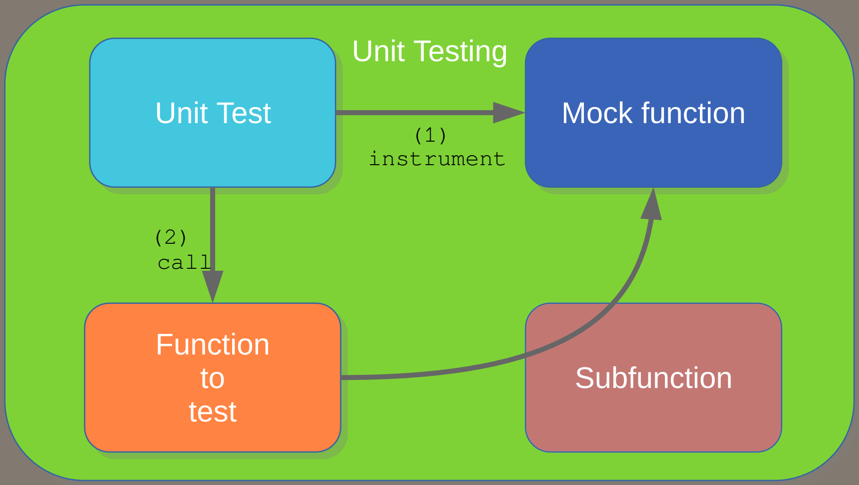
Mocking in unit testing is a way to isolate behaviour of complex algorithms. This is useful if some functions are impractical to incorporate into the unit test.

Mocking test


Unit Testing
Unit Test
Mock function
Function to test
Subfunction



Mocking test


Unit Testing
Unit Test
Mock function
(1) instrument
(2) call
Function to test
Subfunction

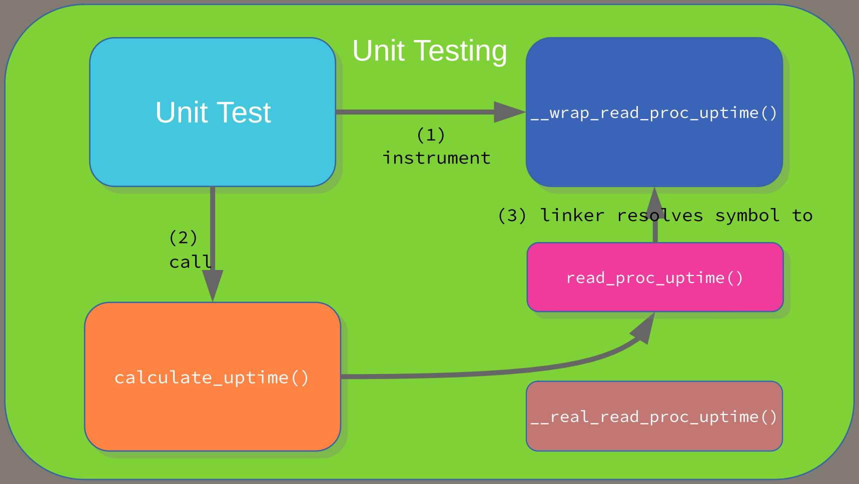
GNU linker magic

Use a wrapper function for a symbol.

```
ld --wrap=<symbol>
```

Supported by `ld.bfd`, `ld.gold` and `llvm-ld`

Mocking test


Unit Testing
Unit Test
(1) instrument
__wrap_read_proc_uptime()
(2) call
(3) linker resolves symbol to
read_proc_uptime()
calculate_uptime()
__real_read_proc_uptime()

Linker function wrapping

If the function prototype is:

```
int read_proc_uptime(double *uptime_secs, double *idle_
```

We implement in the the mock function called:

```
int __wrap_read_proc_uptime(double *uptime_secs, double
{
    ...
}
```

Linker function wrapping

Linker makes

```
read_proc_uptime()
```

available under the symbol

```
__real_read_proc_uptime()
```

Linker function wrapping

The symbol

```
read_proc_uptime()
```

will be resolved to

```
__wrap_read_proc_uptime
```

We still can call the original function in our mock function!

```
__real_read_proc_uptime()
```

Symbol binding order!

Symbols are searched and bound by the linker in the follow order:

1. The executable itself
2. Preloaded libraries
3. Libraries in linking order

Check also `-wrap` resolving in '`man ld`'

Debug symbol binding

With GNU ld.so ..

```
LD_DEBUG=symbols ./examples/uptime/uptime
```

See '`man ld.so`'

Writing mocking functions

Features for mocking

- Parameter Checking
- Mocking
- Call ordering

Parameter checking in mocking functions

```
void mytest(void **state) {
    expect_string(__wrap_mock, food, "wurst");

    myfunction("wurstbrot");
}

int __wrap_mock(char *food) {
    check_expected(food);
}
```

api.cmocka.org -> Checking Parameters

Mocking

```
void mytest(void **state) {
    int rc;

    will_return(__wrap_mock, 0);

    rc = myfunction("wurstbrot");
    assert_return_code(rc, errno);
}

int __wrap_mock(char *name) {
    return mock_type(int);
}
```

api.cmocka.org -> Mock Objects

Call ordering

- Allows you to check that mock functions are called in the right order!

api.cmocka.org -> Call Ordering

4

How to write a mocking test?

This is an exercise for you!

Take a look at the cmocka source code:

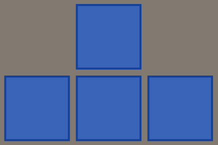
`example/mock/uptime/`

Another mocking example

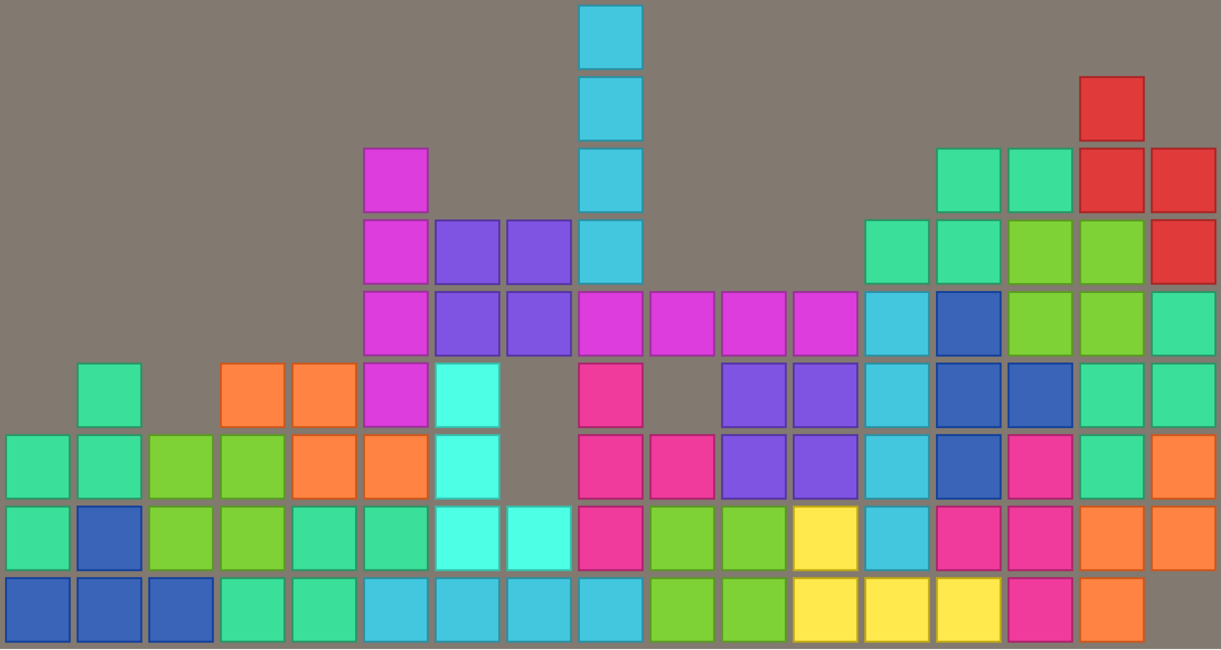
- Samba source code:

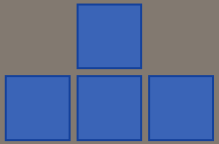
`lib/util/tests/test_talloc_keep_secret.c`

Test that verifies that memset is called when a talloc pointer is defined as a secret.



GAME OVER





- Twitter: @cryptomilk
- Blog: blog.cryptomilk.org