Sysunit

Unit Testing for the FreeBSD Kernel

Ryan Stone

BSDCan 2022

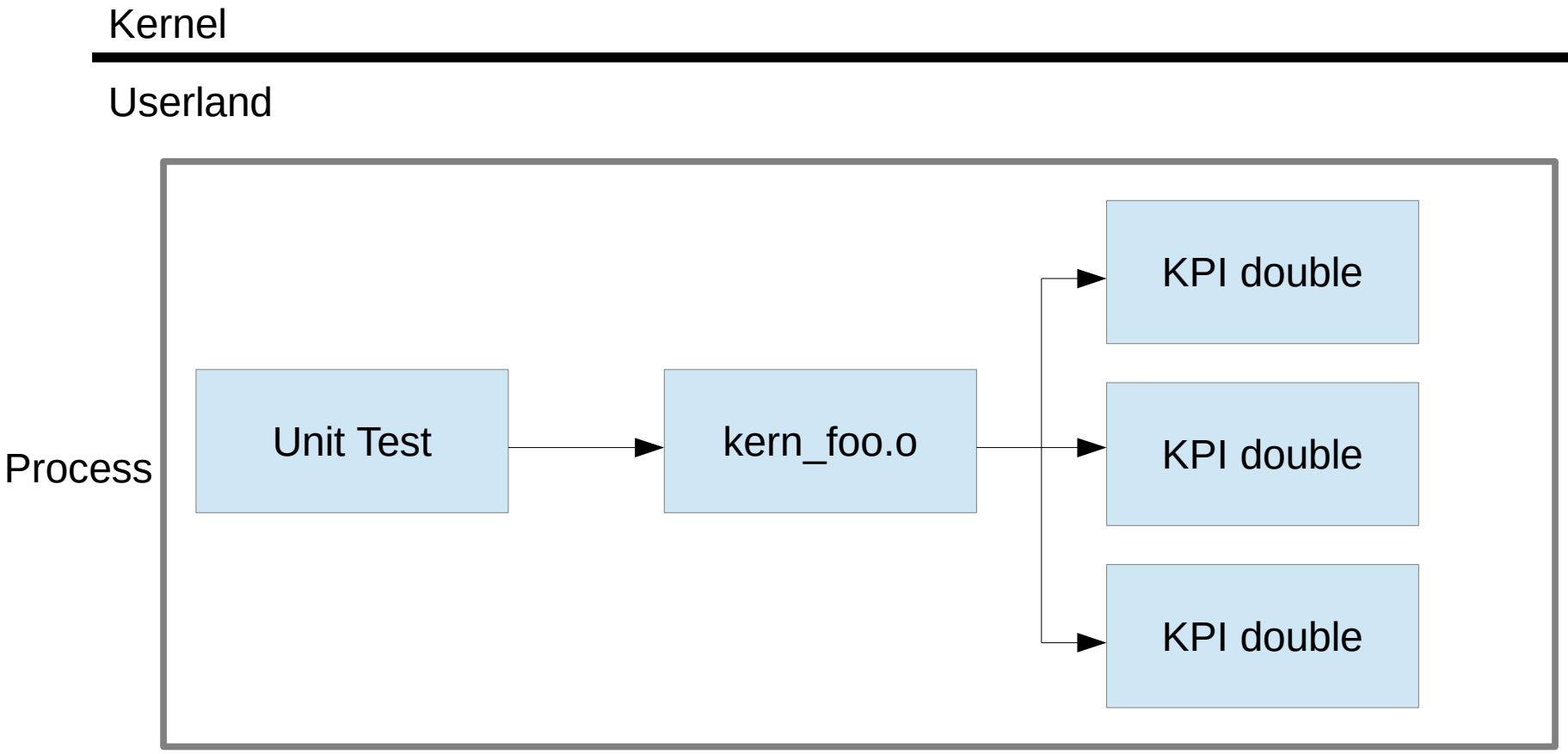
Topics

- What unit testing is and why we want it
- Walkthrough of sample unit tests
- Introduction to Test Doubles and how to implement one in Sysunit

Definition

- Unit testing: testing a small unit of code in an isolated environment
- In Sysunit, this means:
  - Compiling your kernel code for userland and linking it into a userland executable that tests it
  - KPIs are replaced by test doubles - versions written for testing
  - No dependencies on hardware or special kernel facilities

Kernel Code in Userland



Benefits

- Extremely fast test cycle: seconds from :w to testing your code
- No dependencies on separate HW or VMs
- Easier to deterministically inject failures and test all code paths
- Userland debug tools (e.g. gdb) available
- Easier to locate and root-cause regressions

Do Unit Tests Replace System Tests?

- Unit tests *complement* system tests by offering additional mechanisms for finding bugs
- However there is no substitute for booting a kernel and testing it
- Unit tests are poor at finding many types of bugs
  - e.g. Race conditions, API misunderstandings

Writing a Sysunit Test

- Place tests under `tests/sys/sysunit`
- Tests are written in C++ with Google Test

Writing a Sysunit Test

- Place tests under `tests/sys/sysunit`
- Tests are written in C++ with Google Test

Why C++?!

- gtest is *the* industry standard for C/C++ UT
- gtest is already in the src tree
- gtest isn’t ATF
- Out-of-the-box integration Google Mock
- Significantly simpler and more intuitive syntax for writing tests compared to a pure C test API

A Simple Test

```
int
sysctl_ctx_init(struct sysctl_ctx_list *c)
{

	if (c == NULL) {
		return (EINVAL);
	}

	TAILQ_INIT(c);
	return (0);
}
```

A Simple Test

```
int
sysctl_ctx_init(struct sysctl_ctx_list *c)
{

	if (c == NULL) {
		return (EINVAL);
	}

	TAILQ_INIT(c);
	return (0);
}
```

```
#include <sys/cdefs.h>

__BEGIN_DECLS
#include <sys/types.h>
#include <sys/sysctl.h>
__END_DECLS

#include <gtest/gtest.h>

#include "sysunit/TestSuite.h"

class SysctlHandlerTestSuite : public SysUnit::TestSuite
{
}

TEST_F(SysctlHandlerTestSuite, TestContextInit)
{
	struct sysctl_ctx_list ctx;
	int error;

	error = sysctl_ctx_init(&ctx);
	EXPECT_EQ(error, 0);
	EXPECT_TRUE(TAILQ_EMPTY(&ctx));

	error = sysctl_ctx_init(NULL);
	EXPECT_EQ(error, EINVAL);
}
```

Is This a Good Test?

- Do consumers care that it's a TAILQ?
- Do consumers care that the list is init'ed to be empty?
- Is EINVAL a critical part of the API?

```
#include <sys/cdefs.h>

__BEGIN_DECLS
#include <sys/types.h>
#include <sys/sysctl.h>
__END_DECLS

#include <gtest/gtest.h>

#include "sysunit/TestSuite.h"

class SysctlHandlerTestSuite : public SysUnit::TestSuite
{
}

TEST_F(SysctlHandlerTestSuite, TestContextInit)
{
      struct sysctl_ctx_list ctx;
      int error;

      error = sysctl_ctx_init(&ctx);
      EXPECT_EQ(error, 0);
      EXPECT_TRUE(TAILQ_EMPTY(&ctx));

      error = sysctl_ctx_init(NULL);
      EXPECT_EQ(error, EINVAL);
}
```

make

```
.include <src.opts.mk>
.include <bsd.sysunit.pre.mk>

TESTSDIR=  ${TESTSBASE}/sys/sysunit

SYSUNIT_TESTS=  \
     sysctl \

CXXSTD=c++17

SRCS.sysctl= \
     sysctl.cc \

KSRCS.sysctl= \
     kern_sysctl.c \

.PATH: $(SRCTOP)/sys/kern

.include <bsd.sysunit.mk>
.include <bsd.test.mk>
```

```
#include <sys/cdefs.h>

__BEGIN_DECLS
#include <sys/types.h>
#include <sys/sysctl.h>
__END_DECLS

#include <gtest/gtest.h>

#include "sysunit/TestSuite.h"

class SysctlHandlerTestSuite : public SysUnit::TestSuite
{
}

TEST_F(SysctlHandlerTestSuite, TestContextInit)
{
     struct sysctl_ctx_list ctx;
     int error;

     error = sysctl_ctx_init(&ctx);
     EXPECT_EQ(error, 0);
     EXPECT_TRUE(TAILQ_EMPTY(&ctx));

     error = sysctl_ctx_init(NULL);
     EXPECT_EQ(error, EINVAL);
}
```

Missing KPIs at Link Time

```
ld: error: undefined symbol: sbuf_printf_drain
>>> referenced by kern_sysctl.c
>>>               kern_sysctl.o:(sysctl_warn_reuse)

ld: error: undefined symbol: sbuf_set_drain
>>> referenced by kern_sysctl.c
>>>               kern_sysctl.o:(sysctl_warn_reuse)
>>> referenced by kern_sysctl.c
>>>               kern_sysctl.o:(sbuf_new_for_sysctl)

ld: error: undefined symbol: copyin
>>> referenced by kern_sysctl.c
>>>               kern_sysctl.o:(sys___sysctl)

ld: error: undefined symbol: getenv_array
>>> referenced by kern_sysctl.c
>>>               kern_sysctl.o:(sysctl_register_oid)
```

Test Doubles

- Kernel code depends on a *lot* of KPIs
- A test double is a replacement for an API that can’t be used directly in a unit test
- Three types:
  - Stub
    - A double whose behaviour never changes
  - Fake
    - A double whose behaviour changes based on function params
  - Mock
    - A double whose behaviour is programmable by the unit test

Stubs

- Stub libraries go in src/lib/sysunit/stub
- Stubs should generally be limited to link dependencies that the test never exercises

```
int
copyin(const void * udaddr __unused,
    void *kaddr __unused, size_t len __unused)
{
    abort();
}
```

A Second Test

```
int
sysctl_handle_int(SYSCTL_HANDLER_ARGS)
{
	int tmpout, error = 0;

	if (arg1)
		tmpout = *(int *)arg1;
	else
		tmpout = arg2;
	error = SYSCTL_OUT(req, &tmpout, sizeof(int));

	if (error || !req->newptr)
		return (error);

	if (!arg1)
		error = EPERM;
	else
		error = SYSCTL_IN(req, arg1,sizeof(int));
	return (error);
}
```

```
#define SYSCTL_IN(r, p, l) (r->newfunc)(r, p, l)
#define SYSCTL_OUT(r, p, l) (r->oldfunc)(r, p, l)
```

A Second Test

```
int
sysctl_handle_int(SYSCTL_HANDLER_ARGS)
{
	int tmpout, error = 0;

	if (arg1)
		tmpout = *(int *)arg1;
	else
		tmpout = arg2;
	error = SYSCTL_OUT(req, &tmpout, sizeof(int));

	if (error || !req->newptr)
		return (error);

	if (!arg1)
		error = EPERM;
	else
		error = SYSCTL_IN(req, arg1,sizeof(int));
	return (error);
}
```

```
#define SYSCTL_IN(r, p, l) (r->newfunc)(r, p, l)
#define SYSCTL_OUT(r, p, l) (r->oldfunc)(r, p, l)
```

```
TEST_F(SysctlHandlerTestSuite, TestHandleIntGet)
{
	struct sysctl_oid oid;
	struct sysctl_req req;
	int source, error;

	source = 1;
	error = sysctl_handle_int(&oid, &source, 0, &req);
	EXPECT_EQ(error, 0);
	EXPECT_EQ(???, source);
}
```

Fakes

- Fake libraries go in src/lib/sysunit/fake
- Prefer a fake in cases where you just need a working implementation of a KPI for your test
- Use the actual KPI implementation if it’s viable
  - Beware bringing in too many extra dependencies!

Adding a Fake

```
int
sysctl_handle_int(SYSCTL_HANDLER_ARGS)
{
	int tmpout, error = 0;

	if (arg1)
		tmpout = *(int *)arg1;
	else
		tmpout = arg2;
	error = SYSCTL_OUT(req, &tmpout, sizeof(int));

	if (error || !req->newptr)
		return (error);

	if (!arg1)
		error = EPERM;
	else
		error = SYSCTL_IN(req, arg1,sizeof(int));
	return (error);
}
```

```
#define SYSCTL_IN(r, p, l) (r->newfunc)(r, p, l)
#define SYSCTL_OUT(r, p, l) (r->oldfunc)(r, p, l)
```

```
static int
FakeOldFunc(struct sysctl_req *req, const void *src,
    size_t len)
{
	memcpy(req->oldptr, src, len);
	return (0);
}

TEST_F(SysctlHandlerTestSuite, TestHandleIntGet)
{
	struct sysctl_oid oid;
	struct sysctl_req req;
	int fetched, source, error;

	req.oldfunc = FakeOldFunc;
	req.oldptr = &fetched;
	req.oldlen = sizeof(fetched);

	source = 1;
	error = sysctl_handle_int(&oid, &source, 0, &req);
	EXPECT_EQ(error, 0);
	EXPECT_EQ(fetched, source);

	source = INT_MAX;
	error = sysctl_handle_int(&oid, &source, 0, &req);
	EXPECT_EQ(error, 0);
	EXPECT_EQ(fetched, source);
}
```

Testing Error Paths

```
int
sysctl_handle_int(SYSCTL_HANDLER_ARGS)
{
	int tmpout, error = 0;

	if (arg1)
		tmpout = *(int *)arg1;
	else
		tmpout = arg2;
	error = SYSCTL_OUT(req, &tmpout, sizeof(int));

	if (error || !req->newptr)
		return (error);

	if (!arg1)
		error = EPERM;
	else
		error = SYSCTL_IN(req, arg1,sizeof(int));
	return (error);
}
```

```
#define SYSCTL_IN(r, p, l) (r->newfunc)(r, p, l)
#define SYSCTL_OUT(r, p, l) (r->oldfunc)(r, p, l)
```

```
static int fake_error;

static int
FakeOldFunc(struct sysctl_req *req, const void *src,
    size_t len)
{
	if (fake_error != 0)
		return (fake_error);

	memcpy(req->oldptr, src, len);
	return (0);
}
```

Mocks

- A mock is a test double with programmable behaviour
- A test case can configure a mock to validate function parameters, enforce ordering between API calls, or return canned values
- Google Mock is available in the base system
- Mock libraries go in src/lib/syslib/mock

Writing a Mock

```
#include "sysunit/GlobalMock.h"

using SysUnit::GlobalMock;

class MockSysctl : public GlobalMock<MockSysctl>
{
public:
     MOCK_METHOD3(old_func, int(struct sysctl_req *req,
         const void *src, size_t len));
};

template <>
typename GlobalMock<MockSysctl>::Initializer
GlobalMock<MockSysctl>::initializer(0);

static int
MockOldFunc(struct sysctl_req *req, const void *src,
    size_t len)
{
     return MockSysctl::MockObj().old_func(req, src, len);
}
```

```
using testing::_;

TEST_F(SysctlHandlerTestSuite,
    TestHandleIntGetError)
{
     int fetched, source, error;

     req.oldfunc = MockOldFunc;
     req.oldptr = &fetched;
     req.oldlen = sizeof(fetched);

     EXPECT_CALL(MockSysctl::MockObj(),
         old_func(&req, _, sizeof(int)))
         .Times(1)
         .WillOnce(testing::Return(EFAULT))
         .RetiresOnSaturation();

     source = 1;
     error = sysctl_handle_int(&oid, &source, 0,
         &req);
     EXPECT_EQ(error, EFAULT);
}
```

Project Status

- First set of reviews are open
- One suite of sample tests written for TCP LRO
- Need to identify components amenable to unit testing and write lots more tests!
- https://github.com/rysto32/freebsd/tree/sysunit