Linux Dev-tools Documentation

The kernel development community

Jan 15, 2023

CONTENTS

This document is a collection of documents about development tools that can be used to work on the kernel. For now, the documents have been pulled together without any significant effort to integrate them into a coherent whole; patches welcome!

A brief overview of testing-specific tools can be found in *Kernel Testing Guide*

Table of contents

CHAPTER

ONE

KERNEL TESTING GUIDE

There are a number of different tools for testing the Linux kernel, so knowing when to use each of them can be a challenge. This document provides a rough overview of their differences, and how they fit together.

1.1 Writing and Running Tests

The bulk of kernel tests are written using either the kselftest or KUnit frameworks. These both provide infrastructure to help make running tests and groups of tests easier, as well as providing helpers to aid in writing new tests.

If you're looking to verify the behaviour of the Kernel — particularly specific parts of the kernel — then you'll want to use KUnit or kselftest.

1.1.1 The Difference Between KUnit and kselftest

KUnit (*KUnit - Linux Kernel Unit Testing*) is an entirely in-kernel system for "white box" testing: because test code is part of the kernel, it can access internal structures and functions which aren't exposed to userspace.

KUnit tests therefore are best written against small, self-contained parts of the kernel, which can be tested in isolation. This aligns well with the concept of 'unit' testing.

For example, a KUnit test might test an individual kernel function (or even a single codepath through a function, such as an error handling case), rather than a feature as a whole.

This also makes KUnit tests very fast to build and run, allowing them to be run frequently as part of the development process.

There is a KUnit test style guide which may give further pointers in *Test Style and Nomenclature*

kselftest (*Linux Kernel Selftests*), on the other hand, is largely implemented in userspace, and tests are normal userspace scripts or programs.

This makes it easier to write more complicated tests, or tests which need to manipulate the overall system state more (e.g., spawning processes, etc.). However, it's not possible to call kernel functions directly from kselftest. This means that only kernel functionality which is exposed to userspace somehow (e.g. by a syscall, device, filesystem, etc.) can be tested with kselftest. To work around this, some tests include a companion kernel module which exposes more information or functionality. If a test runs mostly or entirely within the kernel, however, KUnit may be the more appropriate tool.

kselftest is therefore suited well to tests of whole features, as these will expose an interface to userspace, which can be tested, but not implementation details. This aligns well with 'system' or 'end-to-end' testing.

For example, all new system calls should be accompanied by kselftest tests.

1.2 Code Coverage Tools

The Linux Kernel supports two different code coverage measurement tools. These can be used to verify that a test is executing particular functions or lines of code. This is useful for determining how much of the kernel is being tested, and for finding corner-cases which are not covered by the appropriate test.

Using gcov with the Linux kernel is GCC's coverage testing tool, which can be used with the kernel to get global or per-module coverage. Unlike KCOV, it does not record per-task coverage. Coverage data can be read from debugfs, and interpreted using the usual gcov tooling.

kcov: code coverage for fuzzing is a feature which can be built in to the kernel to allow capturing coverage on a per-task level. It's therefore useful for fuzzing and other situations where information about code executed during, for example, a single syscall is useful.

1.3 Dynamic Analysis Tools

The kernel also supports a number of dynamic analysis tools, which attempt to detect classes of issues when they occur in a running kernel. These typically each look for a different class of bugs, such as invalid memory accesses, concurrency issues such as data races, or other undefined behaviour like integer overflows.

Some of these tools are listed below:

- kmemleak detects possible memory leaks. See *Kernel Memory Leak Detector*
- KASAN detects invalid memory accesses such as out-of-bounds and use-after-free errors. See *The Kernel Address Sanitizer (KASAN)*
- UBSAN detects behaviour that is undefined by the C standard, like integer overflows. See *The Undefined Behavior Sanitizer - UBSAN*
- KCSAN detects data races. See *The Kernel Concurrency Sanitizer (KCSAN)*
- KFENCE is a low-overhead detector of memory issues, which is much faster than KASAN and can be used in production. See *Kernel Electric-Fence (KFENCE)*
- lockdep is a locking correctness validator. See Documentation/locking/lockdep-design.rst
- There are several other pieces of debug instrumentation in the kernel, many of which can be found in lib/Kconfig.debug

These tools tend to test the kernel as a whole, and do not "pass" like kselftest or KUnit tests. They can be combined with KUnit or kselftest by running tests on a kernel with these tools enabled: you can then be sure that none of these errors are occurring during the test.

Some of these tools integrate with KUnit or kselftest and will automatically fail tests if an issue is detected.

1.4 Static Analysis Tools

In addition to testing a running kernel, one can also analyze kernel source code directly (**at compile time**) using **static analysis** tools. The tools commonly used in the kernel allow one to inspect the whole source tree or just specific files within it. They make it easier to detect and fix problems during the development process.

Sparse can help test the kernel by performing type-checking, lock checking, value range checking, in addition to reporting various errors and warnings while examining the code. See the *Sparse* documentation page for details on how to use it.

Smatch extends Sparse and provides additional checks for programming logic mistakes such as missing breaks in switch statements, unused return values on error checking, forgetting to set an error code in the return of an error path, etc. Smatch also has tests against more serious issues such as integer overflows, null pointer dereferences, and memory leaks. See the project page at http://smatch.sourceforge.net/.

Coccinelle is another static analyzer at our disposal. Coccinelle is often used to aid refactoring and collateral evolution of source code, but it can also help to avoid certain bugs that occur in common code patterns. The types of tests available include API tests, tests for correct usage of kernel iterators, checks for the soundness of free operations, analysis of locking behavior, and further tests known to help keep consistent kernel usage. See the *Coccinelle* documentation page for details.

Beware, though, that static analysis tools suffer from **false positives**. Errors and warns need to be evaluated carefully before attempting to fix them.

1.4.1 When to use Sparse and Smatch

Sparse does type checking, such as verifying that annotated variables do not cause endianness bugs, detecting places that use `__user` pointers improperly, and analyzing the compatibility of symbol initializers.

Smatch does flow analysis and, if allowed to build the function database, it also does cross function analysis. Smatch tries to answer questions like where is this buffer allocated? How big is it? Can this index be controlled by the user? Is this variable larger than that variable?

It's generally easier to write checks in Smatch than it is to write checks in Sparse. Nevertheless, there are some overlaps between Sparse and Smatch checks.

1.4.2 Strong points of Smatch and Coccinelle

Coccinelle is probably the easiest for writing checks. It works before the pre-processor so it's easier to check for bugs in macros using Coccinelle. Coccinelle also creates patches for you, which no other tool does.

For example, with Coccinelle you can do a mass conversion from `kmalloc(x * size, GFP_KERNEL)` to `kmalloc_array(x, size, GFP_KERNEL)`, and that's really useful. If you just created a Smatch warning and try to push the work of converting on to the maintainers they would be annoyed. You'd have to argue about each warning if can really overflow or not.

Coccinelle does no analysis of variable values, which is the strong point of Smatch. On the other hand, Coccinelle allows you to do simple things in a simple way.

CHAPTER

TWO

CHECKPATCH

Checkpatch (scripts/checkpatch.pl) is a perl script which checks for trivial style violations in patches and optionally corrects them. Checkpatch can also be run on file contexts and without the kernel tree.

Checkpatch is not always right. Your judgement takes precedence over checkpatch messages. If your code looks better with the violations, then its probably best left alone.

2.1 Options

This section will describe the options checkpatch can be run with.

Usage:

```
./scripts/checkpatch.pl [OPTION]... [FILE]...
```

Available options:

- -q, –quiet

  Enable quiet mode.

- -v, –verbose Enable verbose mode. Additional verbose test descriptions are output so as to provide information on why that particular message is shown.

- –no-tree

  Run checkpatch without the kernel tree.

- –no-signoff

  Disable the ‘Signed-off-by’ line check. The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch.

  Example:

  ```
  Signed-off-by: Random J Developer <random@developer.example.org>
  ```

  Setting this flag effectively stops a message for a missing signed-off-by line in a patch context.

- –patch

  Treat FILE as a patch. This is the default option and need not be explicitly specified.

- –emacs

  Set output to emacs compile window format. This allows emacs users to jump from the error in the compile window directly to the offending line in the patch.

- –terse

  Output only one line per report.

- –showfile

  Show the diffed file position instead of the input file position.

- -g, –git

  Treat FILE as a single commit or a git revision range.

  Single commit with:

  - <rev>
  - <rev>^
  - <rev>~n

  Multiple commits with:

  - <rev1>..<rev2>
  - <rev1>…<rev2>
  - <rev>-<count>

- -f, –file

  Treat FILE as a regular source file. This option must be used when running checkpatch on source files in the kernel.

- –subjective, –strict

  Enable stricter tests in checkpatch. By default the tests emitted as CHECK do not activate by default. Use this flag to activate the CHECK tests.

- –list-types

  Every message emitted by checkpatch has an associated TYPE. Add this flag to display all the types in checkpatch.

  Note that when this flag is active, checkpatch does not read the input FILE, and no message is emitted. Only a list of types in checkpatch is output.

- –types TYPE(,TYPE2…)

  Only display messages with the given types.

  Example:

  ```
  ./scripts/checkpatch.pl mypatch.patch --types EMAIL_SUBJECT,BRACES
  ```

- –ignore TYPE(,TYPE2…)

  Checkpatch will not emit messages for the specified types.

  Example:

```
./scripts/checkpatch.pl mypatch.patch --ignore EMAIL_SUBJECT,BRACES
```

- –show-types

  By default checkpatch doesn’t display the type associated with the messages. Set this flag to show the message type in the output.

- –max-line-length=n

  Set the max line length (default 100). If a line exceeds the specified length, a LONG_LINE message is emitted.

  The message level is different for patch and file contexts. For patches, a WARNING is emitted. While a milder CHECK is emitted for files. So for file contexts, the –strict flag must also be enabled.

- –min-conf-desc-length=n

  Set the Kconfig entry minimum description length, if shorter, warn.

- –tab-size=n

  Set the number of spaces for tab (default 8).

- –root=PATH

  PATH to the kernel tree root.

  This option must be specified when invoking checkpatch from outside the kernel root.

- –no-summary

  Suppress the per file summary.

- –mailback

  Only produce a report in case of Warnings or Errors. Milder Checks are excluded from this.

- –summary-file

  Include the filename in summary.

- –debug KEY=[0|1]

  Turn on/off debugging of KEY, where KEY is one of ‘values’, ‘possible’, ‘type’, and ‘attr’ (default is all off).

- –fix

  This is an EXPERIMENTAL feature. If correctable errors exists, a file <inputfile>.EXPERIMENTAL-checkpatch-fixes is created which has the automatically fixable errors corrected.

- –fix-inplace

  EXPERIMENTAL - Similar to –fix but input file is overwritten with fixes.

  DO NOT USE this flag unless you are absolutely sure and you have a backup in place.

- –ignore-perl-version

  Override checking of perl version. Runtime errors maybe encountered after enabling this flag if the perl version does not meet the minimum specified.

- –codespell

  Use the codespell dictionary for checking spelling errors.
- –codespellfile

  Use the specified codespell file. Default is '/usr/share/codespell/dictionary.txt'.
- –typedefsfile

  Read additional types from this file.
- –color[=WHEN]

  Use colors 'always', 'never', or only when output is a terminal ('auto'). Default is 'auto'.
- –kconfig-prefix=WORD

  Use WORD as a prefix for Kconfig symbols (default is *CONFIG_*).
- -h, –help, –version

  Display the help text.

2.2 Message Levels

Messages in checkpatch are divided into three levels. The levels of messages in checkpatch denote the severity of the error. They are:

- ERROR

  This is the most strict level. Messages of type ERROR must be taken seriously as they denote things that are very likely to be wrong.
- WARNING

  This is the next stricter level. Messages of type WARNING requires a more careful review. But it is milder than an ERROR.
- CHECK

  This is the mildest level. These are things which may require some thought.

2.3 Type Descriptions

This section contains a description of all the message types in checkpatch.

2.3.1 Allocation style

ALLOC_ARRAY_ARGS The first argument for kcalloc or kmalloc_array should be the number of elements. sizeof() as the first argument is generally wrong.

See: https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html

ALLOC_SIZEOF_STRUCT The allocation style is bad. In general for family of allocation functions using sizeof() to get memory size, constructs like:

```
p = alloc(sizeof(struct foo), ...)
```

should be:

```
p = alloc(sizeof(*p), ...)
```

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#allocating-memory

ALLOC_WITH_MULTIPLY Prefer kmalloc_array/kcalloc over kmalloc/kzalloc with a sizeof multiply.

See: https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html

2.3.2 API usage

ARCH_DEFINES Architecture specific defines should be avoided wherever possible.

ARCH_INCLUDE_LINUX Whenever asm/file.h is included and linux/file.h exists, a conversion can be made when linux/file.h includes asm/file.h. However this is not always the case (See signal.h). This message type is emitted only for includes from arch/.

AVOID_BUG BUG() or BUG_ON() should be avoided totally. Use WARN() and WARN_ON() instead, and handle the "impossible" error condition as gracefully as possible.

See: https://www.kernel.org/doc/html/latest/process/deprecated.html#bug-and-bug-on

CONSIDER_KSTRTO The simple_strtol(), simple_strtoll(), simple_strtoul(), and simple_strtoull() functions explicitly ignore overflows, which may lead to unexpected results in callers. The respective kstrtol(), kstrtoll(), kstrtoul(), and kstrtoull() functions tend to be the correct replacements.

See: https://www.kernel.org/doc/html/latest/process/deprecated.html#simple-strtol-simple-strtoll-simple-strtoul-simple-strtoull

CONSTANT_CONVERSION Use of __constant_<foo> form is discouraged for the following functions:

```
__constant_cpu_to_be[x]
__constant_cpu_to_le[x]
__constant_be[x]_to_cpu
__constant_le[x]_to_cpu
```

```
__constant_htons
__constant_ntohs
```

Using any of these outside of include/uapi/ is not preferred as using the function without __constant_ is identical when the argument is a constant.

In big endian systems, the macros like __constant_cpu_to_be32(x) and cpu_to_be32(x) expand to the same expression:

```
#define __constant_cpu_to_be32(x) ((__force __be32)(__u32)(x))
#define __cpu_to_be32(x)          ((__force __be32)(__u32)(x))
```

In little endian systems, the macros __constant_cpu_to_be32(x) and cpu_to_be32(x) expand to __constant_swab32 and __swab32. __swab32 has a __builtin_constant_p check:

```
#define __swab32(x)                               \
  (__builtin_constant_p((__u32)(x)) ?     \
  ___constant_swab32(x) :                 \
  __fswab32(x))
```

So ultimately they have a special case for constants. Similar is the case with all of the macros in the list. Thus using the __constant_... forms are unnecessarily verbose and not preferred outside of include/uapi.

See: https://lore.kernel.org/lkml/1400106425.12666.6.camel@joe-AO725/

DEPRECATED_API Usage of a deprecated RCU API is detected. It is recommended to replace old flavourful RCU APIs by their new vanilla-RCU counterparts.

The full list of available RCU APIs can be viewed from the kernel docs.

See: https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html#full-list-of-rcu-apis

DEPRECATED_VARIABLE EXTRA_{A,C,CPP,LD}FLAGS are deprecated and should be replaced by the new flags added via commit f77bf01425b1 ("kbuild: introduce ccflags-y, asflags-y and ldflags-y").

The following conversion scheme maybe used:

```
EXTRA_AFLAGS    ->  asflags-y
EXTRA_CFLAGS    ->  ccflags-y
EXTRA_CPPFLAGS  ->  cppflags-y
EXTRA_LDFLAGS   ->  ldflags-y
```

See:

1. https://lore.kernel.org/lkml/20070930191054.GA15876@uranus.ravnborg.org/
2. https://lore.kernel.org/lkml/1313384834-24433-12-git-send-email-lacombar@gmail.com/
3. https://www.kernel.org/doc/html/latest/kbuild/makefiles.html#compilation-flags

DEVICE_ATTR_FUNCTIONS The function names used in DEVICE_ATTR is unusual. Typically, the store and show functions are used with <attr>_store and <attr>_show, where <attr> is a named attribute variable of the device.

Consider the following examples:

```
static DEVICE_ATTR(type, 0444, type_show, NULL);
static DEVICE_ATTR(power, 0644, power_show, power_store);
```

The function names should preferably follow the above pattern.

See: https://www.kernel.org/doc/html/latest/driver-api/driver-model/device.html#attributes

DEVICE_ATTR_RO The DEVICE_ATTR_RO(name) helper macro can be used instead of DEVICE_ATTR(name, 0444, name_show, NULL);

Note that the macro automatically appends _show to the named attribute variable of the device for the show method.

See: https://www.kernel.org/doc/html/latest/driver-api/driver-model/device.html#attributes

DEVICE_ATTR_RW The DEVICE_ATTR_RW(name) helper macro can be used instead of DEVICE_ATTR(name, 0644, name_show, name_store);

Note that the macro automatically appends _show and _store to the named attribute variable of the device for the show and store methods.

See: https://www.kernel.org/doc/html/latest/driver-api/driver-model/device.html#attributes

DEVICE_ATTR_WO The DEVICE_AATR_WO(name) helper macro can be used instead of DEVICE_ATTR(name, 0200, NULL, name_store);

Note that the macro automatically appends _store to the named attribute variable of the device for the store method.

See: https://www.kernel.org/doc/html/latest/driver-api/driver-model/device.html#attributes

DUPLICATED_SYSCTL_CONST Commit d91bff3011cf ("proc/sysctl: add shared variables for range check") added some shared const variables to be used instead of a local copy in each source file.

Consider replacing the sysctl range checking value with the shared one in include/linux/sysctl.h. The following conversion scheme may be used:

```
&zero     ->  SYSCTL_ZERO
&one      ->  SYSCTL_ONE
&int_max  ->  SYSCTL_INT_MAX
```

See:

1. https://lore.kernel.org/lkml/20190430180111.10688-1-mcroce@redhat.com/
2. https://lore.kernel.org/lkml/20190531131422.14970-1-mcroce@redhat.com/

ENOSYS ENOSYS means that a nonexistent system call was called. Earlier, it was wrongly used for things like invalid operations on otherwise valid syscalls. This should be avoided in new code.

See: https://lore.kernel.org/lkml/5eb299021dec23c1a48fa7d9f2c8b794e967766d.1408730669.git.luto@amacapital.net/

ENOTSUPP ENOTSUPP is not a standard error code and should be avoided in new patches. EOPNOTSUPP should be used instead.

See: https://lore.kernel.org/netdev/20200510182252.GA411829@lunn.ch/

EXPORT_SYMBOL EXPORT_SYMBOL should immediately follow the symbol to be exported.

IN_ATOMIC in_atomic() is not for driver use so any such use is reported as an ERROR. Also in_atomic() is often used to determine if sleeping is permitted, but it is not reliable in this use model. Therefore its use is strongly discouraged.

However, in_atomic() is ok for core kernel use.

See: https://lore.kernel.org/lkml/20080320201723.b87b3732.akpm@linux-foundation.org/

LOCKDEP The lockdep_no_validate class was added as a temporary measure to prevent warnings on conversion of device->sem to device->mutex. It should not be used for any other purpose.

See: https://lore.kernel.org/lkml/1268959062.9440.467.camel@laptop/

MALFORMED_INCLUDE The #include statement has a malformed path. This has happened because the author has included a double slash "//" in the pathname accidentally.

USE_LOCKDEP lockdep_assert_held() annotations should be preferred over assertions based on spin_is_locked()

See: https://www.kernel.org/doc/html/latest/locking/lockdep-design.html#annotations

UAPI_INCLUDE No #include statements in include/uapi should use a uapi/ path.

USLEEP_RANGE usleep_range() should be preferred over udelay(). The proper way of using usleep_range() is mentioned in the kernel docs.

See: https://www.kernel.org/doc/html/latest/timers/timers-howto.html#delays-information-on-the-various-kernel-delay-sleep-mechanisms

2.3.3 Comments

BLOCK_COMMENT_STYLE The comment style is incorrect. The preferred style for multi- line comments is:

```
/*
* This is the preferred style
* for multi line comments.
*/
```

The networking comment style is a bit different, with the first line not empty like the former:

```
/* This is the preferred comment style
* for files in net/ and drivers/net/
*/
```

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#commenting

C99_COMMENTS C99 style single line comments (//) should not be used. Prefer the block comment style instead.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#commenting

DATA_RACE Applications of data_race() should have a comment so as to document the reasoning behind why it was deemed safe.

See: https://lore.kernel.org/lkml/20200401101714.44781-1-elver@google.com/

FSF_MAILING_ADDRESS Kernel maintainers reject new instances of the GPL boilerplate paragraph directing people to write to the FSF for a copy of the GPL, since the FSF has moved in the past and may do so again. So do not write paragraphs about writing to the Free Software Foundation's mailing address.

See: https://lore.kernel.org/lkml/20131006222342.GT19510@leaf/

2.3.4 Commit message

BAD_SIGN_OFF The signed-off-by line does not fall in line with the standards specified by the community.

See: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#developer-s-certificate-of-origin-1-1

BAD_STABLE_ADDRESS_STYLE The email format for stable is incorrect. Some valid options for stable address are:

```
1. stable@vger.kernel.org
2. stable@kernel.org
```

For adding version info, the following comment style should be used:

```
stable@vger.kernel.org # version info
```

COMMIT_COMMENT_SYMBOL Commit log lines starting with a '#' are ignored by git as comments. To solve this problem addition of a single space infront of the log line is enough.

COMMIT_MESSAGE The patch is missing a commit description. A brief description of the changes made by the patch should be added.

See: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#describe-your-changes

EMAIL_SUBJECT Naming the tool that found the issue is not very useful in the subject line. A good subject line summarizes the change that the patch brings.

See: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#describe-your-changes

FROM_SIGN_OFF_MISMATCH The author's email does not match with that in the Signed-off-by: line(s). This can be sometimes caused due to an improperly configured email client.

This message is emitted due to any of the following reasons:

```
- The email names do not match.
- The email addresses do not match.
- The email subaddresses do not match.
- The email comments do not match.
```

MISSING_SIGN_OFF The patch is missing a Signed-off-by line. A signed-off-by line should be added according to Developer's certificate of Origin.

See: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#sign-your-work-the-developer-s-certificate-of-origin

NO_AUTHOR_SIGN_OFF The author of the patch has not signed off the patch. It is required that a simple sign off line should be present at the end of explanation of the patch to denote that the author has written it or otherwise has the rights to pass it on as an open source patch.

See: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#sign-your-work-the-developer-s-certificate-of-origin

DIFF_IN_COMMIT_MSG Avoid having diff content in commit message. This causes problems when one tries to apply a file containing both the changelog and the diff because patch(1) tries to apply the diff which it found in the changelog.

See: https://lore.kernel.org/lkml/20150611134006.9df79a893e3636019ad2759e@linux-foundation.org/

GERRIT_CHANGE_ID To be picked up by gerrit, the footer of the commit message might have a Change-Id like:

```
Change-Id: Ic8aaa0728a43936cd4c6e1ed590e01ba8f0fbf5b
Signed-off-by: A. U. Thor <author@example.com>
```

The Change-Id line must be removed before submitting.

GIT_COMMIT_ID The proper way to reference a commit id is: commit <12+ chars of sha1> ("<title line>")

An example may be:

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary
platform_set_drvdata()") removed the unnecessary
platform_set_drvdata(), but left the variable "dev" unused,
delete it.
```

See: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#describe-your-changes

2.3.5 Comparison style

ASSIGN_IN_IF Do not use assignments in if condition. Example:

```
if ((foo = bar(...)) < BAZ) {
```

should be written as:

```
foo = bar(...);
if (foo < BAZ) {
```

BOOL_COMPARISON Comparisons of A to true and false are better written as A and !A.

See: https://lore.kernel.org/lkml/1365563834.27174.12.camel@joe-AO722/

COMPARISON_TO_NULL Comparisons to NULL in the form (foo == NULL) or (foo != NULL) are better written as (!foo) and (foo).

CONSTANT_COMPARISON Comparisons with a constant or upper case identifier on the left side of the test should be avoided.

2.3.6 Indentation and Line Breaks

CODE_INDENT Code indent should use tabs instead of spaces. Outside of comments, documentation and Kconfig, spaces are never used for indentation.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#indentation

DEEP_INDENTATION Indentation with 6 or more tabs usually indicate overly indented code.

It is suggested to refactor excessive indentation of if/else/for/do/while/switch statements.

See: https://lore.kernel.org/lkml/1328311239.21255.24.camel@joe2Laptop/

SWITCH_CASE_INDENT_LEVEL switch should be at the same indent as case. Example:

```
switch (suffix) {
case 'G':
case 'g':
        mem <<= 30;
        break;
case 'M':
case 'm':
        mem <<= 20;
        break;
case 'K':
case 'k':
        mem <<= 10;
        fallthrough;
default:
        break;
}
```

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#indentation

LONG_LINE The line has exceeded the specified maximum length. To use a different maximum line length, the –max-line-length=n option may be added while invoking checkpatch.

Earlier, the default line length was 80 columns. Commit bdc48fa11e46 ("checkpatch/coding-style: deprecate 80-column warning") increased the limit to 100 columns. This is not a hard limit either and it's preferable to stay within 80 columns whenever possible.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#breaking-long-lines-and-strings

LONG_LINE_STRING A string starts before but extends beyond the maximum line length. To use a different maximum line length, the –max-line-length=n option may be added while invoking checkpatch.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#breaking-long-lines-and-strings

LONG_LINE_COMMENT A comment starts before but extends beyond the maximum line length. To use a different maximum line length, the –max-line-length=n option may be added while invoking checkpatch.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#breaking-long-lines-and-strings

SPLIT_STRING Quoted strings that appear as messages in userspace and can be grepped, should not be split across multiple lines.

See: https://lore.kernel.org/lkml/20120203052727.GA15035@leaf/

MULTILINE_DEREFERENCE A single dereferencing identifier spanned on multiple lines like:

```
struct_identifier->member[index].
member = <foo>;
```

is generally hard to follow. It can easily lead to typos and so makes the code vulnerable to bugs.

If fixing the multiple line dereferencing leads to an 80 column violation, then either rewrite the code in a more simple way or if the starting part of the dereferencing identifier is the same and used at multiple places then store it in a temporary variable, and use that temporary variable only at all the places. For example, if there are two dereferencing identifiers:

```
member1->member2->member3.foo1;
member1->member2->member3.foo2;
```

then store the member1->member2->member3 part in a temporary variable. It not only helps to avoid the 80 column violation but also reduces the program size by removing the unnecessary dereferences.

But if none of the above methods work then ignore the 80 column violation because it is much easier to read a dereferencing identifier on a single line.

TRAILING_STATEMENTS Trailing statements (for example after any conditional) should be on the next line. Statements, such as:

```
if (x == y) break;
```

should be:

```
if (x == y)
        break;
```

2.3.7 Macros, Attributes and Symbols

ARRAY_SIZE The ARRAY_SIZE(foo) macro should be preferred over sizeof(foo)/sizeof(foo[0]) for finding number of elements in an array.

The macro is defined in include/linux/kernel.h:

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

AVOID_EXTERNS Function prototypes don't need to be declared extern in .h files. It's assumed by the compiler and is unnecessary.

AVOID_L_PREFIX Local symbol names that are prefixed with *.L* should be avoided, as this has special meaning for the assembler; a symbol entry will not be emitted into the symbol table. This can prevent *objtool* from generating correct unwind info.

Symbols with STB_LOCAL binding may still be used, and *.L* prefixed local symbol names are still generally usable within a function, but *.L* prefixed local symbol names should not be used to denote the beginning or end of code regions via *SYM_CODE_START_LOCAL/SYM_CODE_END*

BIT_MACRO Defines like: 1 << <digit> could be BIT(digit). The BIT() macro is defined via include/linux/bits.h:

```
#define BIT(nr)         (1UL << (nr))
```

CONST_READ_MOSTLY When a variable is tagged with the __read_mostly annotation, it is a signal to the compiler that accesses to the variable will be mostly reads and rarely(but NOT never) a write.

const __read_mostly does not make any sense as const data is already read-only. The __read_mostly annotation thus should be removed.

DATE_TIME It is generally desirable that building the same source code with the same set of tools is reproducible, i.e. the output is always exactly the same.

The kernel does *not* use the `__DATE__` and `__TIME__` macros, and enables warnings if they are used as they can lead to non-deterministic builds.

See: https://www.kernel.org/doc/html/latest/kbuild/reproducible-builds.html#timestamps

DEFINE_ARCH_HAS The ARCH_HAS_xyz and ARCH_HAVE_xyz patterns are wrong.

For big conceptual features use Kconfig symbols instead. And for smaller things where we have compatibility fallback functions but want architectures able to override them with optimized ones, we should either use weak functions (appropriate for some cases), or the symbol that protects them should be the same symbol we use.

See: https://lore.kernel.org/lkml/CA+55aFycQ9XJvEOsiM3txHL5bjUc8CeKWJNR_H+MiicaddB42Q@mail.gmail.com/

DO_WHILE_MACRO_WITH_TRAILING_SEMICOLON do {} while(0) macros should not have a trailing semicolon.

INIT_ATTRIBUTE Const init definitions should use __initconst instead of __initdata.

Similarly init definitions without const require a separate use of const.

INLINE_LOCATION The inline keyword should sit between storage class and type.

For example, the following segment:

```
inline static int example_function(void)
{
        ...
}
```

should be:

```
static inline int example_function(void)
{
        ...
}
```

MISPLACED_INIT It is possible to use section markers on variables in a way which gcc doesn’t understand (or at least not the way the developer intended):

```
static struct __initdata samsung_pll_clock exynos4_plls[nr_plls] =
↪{
```

does not put exynos4_plls in the .initdata section. The __initdata marker can be virtually anywhere on the line, except right after “struct”. The preferred location is before the “=” sign if there is one, or before the trailing “;” otherwise.

See: https://lore.kernel.org/lkml/1377655732.3619.19.camel@joe-AO722/

MULTISTATEMENT_MACRO_USE_DO_WHILE Macros with multiple statements should be enclosed in a do - while block. Same should also be the case for macros starting with *if* to avoid logic defects:

```
#define macrofun(a, b, c)                 \
  do {                                    \
          if (a == 5)                     \
                do_this(b, c);          \
} while (0)
```

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#macros-enums-and-rtl

PREFER_FALLTHROUGH Use the *fallthrough;* pseudo keyword instead of /* *fallthrough* */ like comments.

TRAILING_SEMICOLON Macro definition should not end with a semicolon. The macro invocation style should be consistent with function calls. This can prevent any unexpected code paths:

```
#define MAC do_something;
```

If this macro is used within a if else statement, like:

```
if (some_condition)
        MAC;

else
        do_something;
```

Then there would be a compilation error, because when the macro is expanded there are two trailing semicolons, so the else branch gets orphaned.

See: https://lore.kernel.org/lkml/1399671106.2912.21.camel@joe-AO725/

SINGLE_STATEMENT_DO_WHILE_MACRO For the multi-statement macros, it is necessary to use the do-while loop to avoid unpredictable code paths. The do-while loop helps to group the multiple statements into a single one so that a function-like macro can be used as a function only.

But for the single statement macros, it is unnecessary to use the do-while loop. Although the code is syntactically correct but using the do-while loop is redundant. So remove the do-while loop for single statement macros.

WEAK_DECLARATION Using weak declarations like __attribute__((weak)) or __weak can have unintended link defects. Avoid using them.

2.3.8 Functions and Variables

CAMELCASE Avoid CamelCase Identifiers.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#naming

CONST_CONST Using *const <type> const* * is generally meant to be written *const <type>* * *const*.

CONST_STRUCT Using const is generally a good idea. Checkpatch reads a list of frequently used structs that are always or almost always constant.

The existing structs list can be viewed from *scripts/const_structs.checkpatch*.

See: https://lore.kernel.org/lkml/alpine.DEB.2.10.1608281509480.3321@hadrien/

EMBEDDED_FUNCTION_NAME Embedded function names are less appropriate to use as refactoring can cause function renaming. Prefer the use of "%s", __func__ to embedded function names.

Note that this does not work with -f (–file) checkpatch option as it depends on patch context providing the function name.

FUNCTION_ARGUMENTS This warning is emitted due to any of the following reasons:

1. Arguments for the function declaration do not follow the identifier name. Example:

   ```
   void foo
   (int bar, int baz)
   ```

   This should be corrected to:

   ```
   void foo(int bar, int baz)
   ```

2. Some arguments for the function definition do not have an identifier name. Example:

   ```
   void foo(int)
   ```

   All arguments should have identifier names.

FUNCTION_WITHOUT_ARGS Function declarations without arguments like:

```
int foo()
```

should be:

```
int foo(void)
```

GLOBAL_INITIALISERS Global variables should not be initialized explicitly to 0 (or NULL, false, etc.). Your compiler (or rather your loader, which is responsible for zeroing out the relevant sections) automatically does it for you.

INITIALISED_STATIC Static variables should not be initialized explicitly to zero. Your compiler (or rather your loader) automatically does it for you.

MULTIPLE_ASSIGNMENTS Multiple assignments on a single line makes the code unnecessarily complicated. So on a single line assign value to a single variable only, this makes the code more readable and helps avoid typos.

RETURN_PARENTHESES return is not a function and as such doesn't need parentheses:

```
return (bar);
```

can simply be:

```
return bar;
```

2.3.9 Permissions

DEVICE_ATTR_PERMS The permissions used in DEVICE_ATTR are unusual. Typically only three permissions are used - 0644 (RW), 0444 (RO) and 0200 (WO).

See: https://www.kernel.org/doc/html/latest/filesystems/sysfs.html#attributes

EXECUTE_PERMISSIONS There is no reason for source files to be executable. The executable bit can be removed safely.

EXPORTED_WORLD_WRITABLE Exporting world writable sysfs/debugfs files is usually a bad thing. When done arbitrarily they can introduce serious security bugs. In the past, some of the debugfs vulnerabilities would seemingly allow any local user to write arbitrary values into device registers - a situation from which little good can be expected to emerge.

See: https://lore.kernel.org/linux-arm-kernel/cover.1296818921.git.segoon@openwall.com/

NON_OCTAL_PERMISSIONS Permission bits should use 4 digit octal permissions (like 0700 or 0444). Avoid using any other base like decimal.

SYMBOLIC_PERMS Permission bits in the octal form are more readable and easier to understand than their symbolic counterparts because many command-line tools use this notation. Experienced kernel developers have been using these traditional Unix permission bits for decades and so they find it easier to understand the octal notation than the symbolic macros. For example, it is harder to read S_IWUSR|S_IRUGO than 0644, which obscures the developer's intent rather than clarifying it.

See: https://lore.kernel.org/lkml/CA+55aFw5v23T-zvDZp-MmD_EYxF8WbafwwB59934FV7g21uMGQ@mail.gmail.com/

2.3.10 Spacing and Brackets

ASSIGNMENT_CONTINUATIONS Assignment operators should not be written at the start of a line but should follow the operand at the previous line.

BRACES The placement of braces is stylistically incorrect. The preferred way is to put the opening brace last on the line, and put the closing brace first:

```
if (x is true) {
        we do y
}
```

This applies for all non-functional blocks. However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
        body of function
}
```

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#placing-braces-and-spaces

BRACKET_SPACE Whitespace before opening bracket ‘[‘ is prohibited. There are some exceptions:

1. With a type on the left:

```
int [] a;
```

2. At the beginning of a line for slice initialisers:

```
[0...10] = 5,
```

3. Inside a curly brace:

```
= { [0...10] = 5 }
```

CONCATENATED_STRING Concatenated elements should have a space in between. Example:

```
printk(KERN_INFO"bar");
```

should be:

```
printk(KERN_INFO "bar");
```

ELSE_AFTER_BRACE *else {* should follow the closing block *}* on the same line.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#placing-braces-and-spaces

LINE_SPACING Vertical space is wasted given the limited number of lines an editor window can display when multiple blank lines are used.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#spaces

OPEN_BRACE The opening brace should be following the function definitions on the next line. For any non-functional block it should be on the same line as the last construct.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#placing-braces-and-spaces

POINTER_LOCATION When using pointer data or a function that returns a pointer type, the preferred use of * is adjacent to the data name or function name and not adjacent to the type name. Examples:

```
char *linux_banner;
unsigned long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#spaces

SPACING Whitespace style used in the kernel sources is described in kernel docs.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#spaces

TRAILING_WHITESPACE Trailing whitespace should always be removed. Some editors highlight the trailing whitespace and cause visual distractions when editing files.

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#spaces

UNNECESSARY_PARENTHESES Parentheses are not required in the following cases:

1. Function pointer uses:

   ```
   (foo->bar)();
   ```

   could be:

   ```
   foo->bar();
   ```

2. Comparisons in if:

   ```
   if ((foo->bar) && (foo->baz))
   if ((foo == bar))
   ```

   could be:

   ```
   if (foo->bar && foo->baz)
   if (foo == bar)
   ```

3. addressof/dereference single Lvalues:

   ```
   &(foo->bar)
   *(foo->bar)
   ```

   could be:

   ```
   &foo->bar
   *foo->bar
   ```

WHILE_AFTER_BRACE while should follow the closing bracket on the same line:

```
do {
        ...
} while(something);
```

See: https://www.kernel.org/doc/html/latest/process/coding-style.html#placing-braces-and-spaces

2.3.11 Others

CONFIG_DESCRIPTION Kconfig symbols should have a help text which fully describes it.

CORRUPTED_PATCH The patch seems to be corrupted or lines are wrapped. Please regenerate the patch file before sending it to the maintainer.

CVS_KEYWORD Since linux moved to git, the CVS markers are no longer used. So, CVS style keywords (Id, $Revision$, Log) should not be added.

DEFAULT_NO_BREAK switch default case is sometimes written as "default:;". This can cause new cases added below default to be defective.

A "break;" should be added after empty default statement to avoid unwanted fallthrough.

DOS_LINE_ENDINGS For DOS-formatted patches, there are extra ^M symbols at the end of the line. These should be removed.

DT_SCHEMA_BINDING_PATCH DT bindings moved to a json-schema based format instead of freeform text.

See: https://www.kernel.org/doc/html/latest/devicetree/bindings/writing-schema.html

DT_SPLIT_BINDING_PATCH Devicetree bindings should be their own patch. This is because bindings are logically independent from a driver implementation, they have a different maintainer (even though they often are applied via the same tree), and it makes for a cleaner history in the DT only tree created with git-filter-branch.

See: https://www.kernel.org/doc/html/latest/devicetree/bindings/submitting-patches.html#i-for-patch-submitters

EMBEDDED_FILENAME Embedding the complete filename path inside the file isn't particularly useful as often the path is moved around and becomes incorrect.

FILE_PATH_CHANGES Whenever files are added, moved, or deleted, the MAINTAINERS file patterns can be out of sync or outdated.

So MAINTAINERS might need updating in these cases.

MEMSET The memset use appears to be incorrect. This may be caused due to badly ordered parameters. Please recheck the usage.

NOT_UNIFIED_DIFF The patch file does not appear to be in unified-diff format. Please regenerate the patch file before sending it to the maintainer.

PRINTF_0XDECIMAL Prefixing 0x with decimal output is defective and should be corrected.

SPDX_LICENSE_TAG The source file is missing or has an improper SPDX identifier tag. The Linux kernel requires the precise SPDX identifier in all source files, and it is thoroughly documented in the kernel docs.

See: https://www.kernel.org/doc/html/latest/process/license-rules.html

TYPO_SPELLING Some words may have been misspelled. Consider reviewing them.

CHAPTER

THREE

COCCINELLE

Coccinelle is a tool for pattern matching and text transformation that has many uses in kernel development, including the application of complex, tree-wide patches and detection of problematic programming patterns.

3.1 Getting Coccinelle

The semantic patches included in the kernel use features and options which are provided by Coccinelle version 1.0.0-rc11 and above. Using earlier versions will fail as the option names used by the Coccinelle files and coccicheck have been updated.

Coccinelle is available through the package manager of many distributions, e.g. :

- Debian
- Fedora
- Ubuntu
- OpenSUSE
- Arch Linux
- NetBSD
- FreeBSD

Some distribution packages are obsolete and it is recommended to use the latest version released from the Coccinelle homepage at http://coccinelle.lip6.fr/

Or from Github at:

https://github.com/coccinelle/coccinelle

Once you have it, run the following commands:

```
./autogen
./configure
make
```

as a regular user, and install it with:

```
sudo make install
```

More detailed installation instructions to build from source can be found at:

https://github.com/coccinelle/coccinelle/blob/master/install.txt

3.2 Supplemental documentation

For supplemental documentation refer to the wiki:

https://bottest.wiki.kernel.org/coccicheck

The wiki documentation always refers to the linux-next version of the script.

For Semantic Patch Language(SmPL) grammar documentation refer to:

http://coccinelle.lip6.fr/documentation.php

3.3 Using Coccinelle on the Linux kernel

A Coccinelle-specific target is defined in the top level Makefile. This target is named `coccicheck` and calls the `coccicheck` front-end in the `scripts` directory.

Four basic modes are defined: `patch`, `report`, `context`, and `org`. The mode to use is specified by setting the MODE variable with `MODE=<mode>`.

- `patch` proposes a fix, when possible.
- `report` generates a list in the following format: file:line:column-column: message
- `context` highlights lines of interest and their context in a diff-like style. Lines of interest are indicated with `-`.
- `org` generates a report in the Org mode format of Emacs.

Note that not all semantic patches implement all modes. For easy use of Coccinelle, the default mode is "report".

Two other modes provide some common combinations of these modes.

- `chain` tries the previous modes in the order above until one succeeds.
- `rep+ctxt` runs successively the report mode and the context mode. It should be used with the C option (described later) which checks the code on a file basis.

3.3.1 Examples

To make a report for every semantic patch, run the following command:

```
make coccicheck MODE=report
```

To produce patches, run:

```
make coccicheck MODE=patch
```

The coccicheck target applies every semantic patch available in the sub-directories of `scripts/coccinelle` to the entire Linux kernel.

For each semantic patch, a commit message is proposed. It gives a description of the problem being checked by the semantic patch, and includes a reference to Coccinelle.

As with any static code analyzer, Coccinelle produces false positives. Thus, reports must be carefully checked, and patches reviewed.

To enable verbose messages set the V= variable, for example:

```
make coccicheck MODE=report V=1
```

3.4 Coccinelle parallelization

By default, coccicheck tries to run as parallel as possible. To change the parallelism, set the J= variable. For example, to run across 4 CPUs:

```
make coccicheck MODE=report J=4
```

As of Coccinelle 1.0.2 Coccinelle uses Ocaml parmap for parallelization; if support for this is detected you will benefit from parmap parallelization.

When parmap is enabled coccicheck will enable dynamic load balancing by using `--chunksize 1` argument. This ensures we keep feeding threads with work one by one, so that we avoid the situation where most work gets done by only a few threads. With dynamic load balancing, if a thread finishes early we keep feeding it more work.

When parmap is enabled, if an error occurs in Coccinelle, this error value is propagated back, and the return value of the `make coccicheck` command captures this return value.

3.5 Using Coccinelle with a single semantic patch

The optional make variable COCCI can be used to check a single semantic patch. In that case, the variable must be initialized with the name of the semantic patch to apply.

For instance:

```
make coccicheck COCCI=<my_SP.cocci> MODE=patch
```

or:

```
make coccicheck COCCI=<my_SP.cocci> MODE=report
```

3.6 Controlling Which Files are Processed by Coccinelle

By default the entire kernel source tree is checked.

To apply Coccinelle to a specific directory, M= can be used. For example, to check drivers/net/wireless/ one may write:

```
make coccicheck M=drivers/net/wireless/
```

To apply Coccinelle on a file basis, instead of a directory basis, the C variable is used by the makefile to select which files to work with. This variable can be used to run scripts for the entire kernel, a specific directory, or for a single file.

For example, to check drivers/bluetooth/bfusb.c, the value 1 is passed to the C variable to check files that make considers need to be compiled.:

```
make C=1 CHECK=scripts/coccicheck drivers/bluetooth/bfusb.o
```

The value 2 is passed to the C variable to check files regardless of whether they need to be compiled or not.:

```
make C=2 CHECK=scripts/coccicheck drivers/bluetooth/bfusb.o
```

In these modes, which work on a file basis, there is no information about semantic patches displayed, and no commit message proposed.

This runs every semantic patch in scripts/coccinelle by default. The COCCI variable may additionally be used to only apply a single semantic patch as shown in the previous section.

The “report” mode is the default. You can select another one with the MODE variable explained above.

3.7 Debugging Coccinelle SmPL patches

Using coccicheck is best as it provides in the spatch command line include options matching the options used when we compile the kernel. You can learn what these options are by using V=1; you could then manually run Coccinelle with debug options added.

Alternatively you can debug running Coccinelle against SmPL patches by asking for stderr to be redirected to stderr. By default stderr is redirected to /dev/null; if you’d like to capture stderr you can specify the `DEBUG_FILE="file.txt"` option to coccicheck. For instance:

```
rm -f cocci.err
make coccicheck COCCI=scripts/coccinelle/free/kfree.cocci MODE=report DEBUG_
↪FILE=cocci.err
cat cocci.err
```

You can use SPFLAGS to add debugging flags; for instance you may want to add both –profile –show-trying to SPFLAGS when debugging. For example you may want to use:

```
rm -f err.log
export COCCI=scripts/coccinelle/misc/irqf_oneshot.cocci
make coccicheck DEBUG_FILE="err.log" MODE=report SPFLAGS="--profile --show-
↪trying" M=./drivers/mfd
```

err.log will now have the profiling information, while stdout will provide some progress information as Coccinelle moves forward with work.

NOTE:

DEBUG_FILE support is only supported when using coccinelle >= 1.0.2.

Currently, DEBUG_FILE support is only available to check folders, and not single files. This is because checking a single file requires spatch to be called twice leading to DEBUG_FILE being set both times to the same value, giving rise to an error.

3.8 .cocciconfig support

Coccinelle supports reading .cocciconfig for default Coccinelle options that should be used every time spatch is spawned. The order of precedence for variables for .cocciconfig is as follows:

- Your current user's home directory is processed first
- Your directory from which spatch is called is processed next
- The directory provided with the –dir option is processed last, if used

Since coccicheck runs through make, it naturally runs from the kernel proper dir; as such the second rule above would be implied for picking up a .cocciconfig when using `make coccicheck`.

`make coccicheck` also supports using M= targets. If you do not supply any M= target, it is assumed you want to target the entire kernel. The kernel coccicheck script has:

```
if [ "$KBUILD_EXTMOD" = "" ] ; then
    OPTIONS="--dir $srctree $COCCIINCLUDE"
else
    OPTIONS="--dir $KBUILD_EXTMOD $COCCIINCLUDE"
fi
```

KBUILD_EXTMOD is set when an explicit target with M= is used. For both cases the spatch –dir argument is used, as such third rule applies when whether M= is used or not, and when M= is used the target directory can have its own .cocciconfig file. When M= is not passed as an argument to coccicheck the target directory is the same as the directory from where spatch was called.

If not using the kernel's coccicheck target, keep the above precedence order logic of .cocciconfig reading. If using the kernel's coccicheck target, override any of the kernel's .coccicheck's settings using SPFLAGS.

We help Coccinelle when used against Linux with a set of sensible default options for Linux with our own Linux .cocciconfig. This hints to coccinelle that git can be used for `git grep` queries over coccigrep. A timeout of 200 seconds should suffice for now.

The options picked up by coccinelle when reading a .cocciconfig do not appear as arguments to spatch processes running on your system. To confirm what options will be used by Coccinelle run:

```
spatch --print-options-only
```

You can override with your own preferred index option by using SPFLAGS. Take note that when there are conflicting options Coccinelle takes precedence for the last options passed. Using .cocciconfig is possible to use idutils, however given the order of precedence followed by Coccinelle, since the kernel now carries its own .cocciconfig, you will need to use SPFLAGS to use idutils if desired. See below section "Additional flags" for more details on how to use idutils.

3.9 Additional flags

Additional flags can be passed to spatch through the SPFLAGS variable. This works as Coccinelle respects the last flags given to it when options are in conflict.

```
make SPFLAGS=--use-glimpse coccicheck
```

Coccinelle supports idutils as well but requires coccinelle >= 1.0.6. When no ID file is specified coccinelle assumes your ID database file is in the file .id-utils.index on the top level of the kernel. Coccinelle carries a script scripts/idutils_index.sh which creates the database with:

```
mkid -i C --output .id-utils.index
```

If you have another database filename you can also just symlink with this name.

```
make SPFLAGS=--use-idutils coccicheck
```

Alternatively you can specify the database filename explicitly, for instance:

```
make SPFLAGS="--use-idutils /full-path/to/ID" coccicheck
```

See `spatch --help` to learn more about spatch options.

Note that the `--use-glimpse` and `--use-idutils` options require external tools for indexing the code. None of them is thus active by default. However, by indexing the code with one of these tools, and according to the cocci file used, spatch could proceed the entire code base more quickly.

3.10 SmPL patch specific options

SmPL patches can have their own requirements for options passed to Coccinelle. SmPL patch-specific options can be provided by providing them at the top of the SmPL patch, for instance:

```
// Options: --no-includes --include-headers
```

3.11 SmPL patch Coccinelle requirements

As Coccinelle features get added some more advanced SmPL patches may require newer versions of Coccinelle. If an SmPL patch requires a minimum version of Coccinelle, this can be specified as follows, as an example if requiring at least Coccinelle >= 1.0.5:

```
// Requires: 1.0.5
```

3.12 Proposing new semantic patches

New semantic patches can be proposed and submitted by kernel developers. For sake of clarity, they should be organized in the sub-directories of `scripts/coccinelle/`.

3.13 Detailed description of the `report` mode

`report` generates a list in the following format:

```
file:line:column-column: message
```

3.13.1 Example

Running:

```
make coccicheck MODE=report COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on report@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
coccilib.report.print_report(p[0], msg)
</smpl>
```

This SmPL excerpt generates entries on the standard output, as illustrated below:

```
/home/user/linux/crypto/ctr.c:188:9-16: ERR_CAST can be used with alg
/home/user/linux/crypto/authenc.c:619:9-16: ERR_CAST can be used with auth
/home/user/linux/crypto/xts.c:227:9-16: ERR_CAST can be used with alg
```

3.14 Detailed description of the `patch` mode

When the `patch` mode is available, it proposes a fix for each problem identified.

3.14.1 Example

Running:

```
make coccicheck MODE=patch COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on !context && patch && !org && !report @
expression x;
@@

- ERR_PTR(PTR_ERR(x))
+ ERR_CAST(x)
</smpl>
```

This SmPL excerpt generates patch hunks on the standard output, as illustrated below:

```
diff -u -p a/crypto/ctr.c b/crypto/ctr.c
--- a/crypto/ctr.c 2010-05-26 10:49:38.000000000 +0200
+++ b/crypto/ctr.c 2010-06-03 23:44:49.000000000 +0200
@@ -185,7 +185,7 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                              CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));
+           return ERR_CAST(alg);

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

3.15 Detailed description of the context mode

context highlights lines of interest and their context in a diff-like style.

> **NOTE**: The diff-like output generated is NOT an applicable patch. The intent of the context mode is to highlight the important lines (annotated with minus, -) and gives some surrounding context lines around. This output can be used with the diff mode of Emacs to review the code.

3.15.1 Example

Running:

```
make coccicheck MODE=context COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on context && !patch && !org && !report@
expression x;
@@

* ERR_PTR(PTR_ERR(x))
</smpl>
```

This SmPL excerpt generates diff hunks on the standard output, as illustrated below:

```
diff -u -p /home/user/linux/crypto/ctr.c /tmp/nothing
--- /home/user/linux/crypto/ctr.c   2010-05-26 10:49:38.000000000 +0200
+++ /tmp/nothing
@@ -185,7 +185,6 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                              CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

3.16 Detailed description of the org mode

org generates a report in the Org mode format of Emacs.

3.16.1 Example

Running:

```
make coccicheck MODE=org COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on org@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
msg_safe=msg.replace("[","@(").replace("]",")")
coccilib.org.print_todo(p[0], msg_safe)
</smpl>
```

This SmPL excerpt generates Org entries on the standard output, as illustrated below:

```
* TODO [[view:/home/user/linux/crypto/ctr.c::face=ovl-
 ↪face1::linb=188::colb=9::cole=16][ERR_CAST can be used with alg]]
* TODO [[view:/home/user/linux/crypto/authenc.c::face=ovl-
 ↪face1::linb=619::colb=9::cole=16][ERR_CAST can be used with auth]]
* TODO [[view:/home/user/linux/crypto/xts.c::face=ovl-
 ↪face1::linb=227::colb=9::cole=16][ERR_CAST can be used with alg]]
```

CHAPTER

FOUR

SPARSE

Sparse is a semantic checker for C programs; it can be used to find a number of potential problems with kernel code. See https://lwn.net/Articles/689907/ for an overview of sparse; this document contains some kernel-specific sparse information. More information on sparse, mainly about its internals, can be found in its official pages at https://sparse.docs.kernel.org.

4.1 Using sparse for typechecking

“__bitwise” is a type attribute, so you have to do something like this:

```
typedef int __bitwise pm_request_t;

enum pm_request {
        PM_SUSPEND = (__force pm_request_t) 1,
        PM_RESUME = (__force pm_request_t) 2
};
```

which makes PM_SUSPEND and PM_RESUME “bitwise” integers (the “__force” is there because sparse will complain about casting to/from a bitwise type, but in this case we really _do_ want to force the conversion). And because the enum values are all the same type, now “enum pm_request” will be that type too.

And with gcc, all the “__bitwise”/”__force stuff” goes away, and it all ends up looking just like integers to gcc.

Quite frankly, you don’t need the enum there. The above all really just boils down to one special “int __bitwise” type.

So the simpler way is to just do:

```
typedef int __bitwise pm_request_t;

#define PM_SUSPEND ((__force pm_request_t) 1)
#define PM_RESUME ((__force pm_request_t) 2)
```

and you now have all the infrastructure needed for strict typechecking.

One small note: the constant integer “0” is special. You can use a constant zero as a bitwise integer type without sparse ever complaining. This is because “bitwise” (as the name implies) was designed for making sure that bitwise types don’t get mixed up (little-endian vs big-endian vs cpu-endian vs whatever), and there the constant “0” really _is_ special.

4.2 Using sparse for lock checking

The following macros are undefined for gcc and defined during a sparse run to use the "context" tracking feature of sparse, applied to locking. These annotations tell sparse when a lock is held, with regard to the annotated function's entry and exit.

__must_hold - The specified lock is held on function entry and exit.

__acquires - The specified lock is held on function exit, but not entry.

__releases - The specified lock is held on function entry, but not exit.

If the function enters and exits without the lock held, acquiring and releasing the lock inside the function in a balanced way, no annotation is needed. The three annotations above are for cases where sparse would otherwise report a context imbalance.

4.3 Getting sparse

You can get tarballs of the latest released versions from: https://www.kernel.org/pub/software/devel/sparse/dist/

Alternatively, you can get snapshots of the latest development version of sparse using git to clone:

```
git://git.kernel.org/pub/scm/devel/sparse/sparse.git
```

Once you have it, just do:

```
make
make install
```

as a regular user, and it will install sparse in your ~/bin directory.

4.4 Using sparse

Do a kernel make with "make C=1" to run sparse on all the C files that get recompiled, or use "make C=2" to run sparse on the files whether they need to be recompiled or not. The latter is a fast way to check the whole tree if you have already built it.

The optional make variable CF can be used to pass arguments to sparse. The build system passes -Wbitwise to sparse automatically.

Note that sparse defines the __CHECKER__ preprocessor symbol.

CHAPTER

FIVE

KCOV: CODE COVERAGE FOR FUZZING

kcov exposes kernel code coverage information in a form suitable for coverage- guided fuzzing (randomized testing). Coverage data of a running kernel is exported via the “kcov” debugfs file. Coverage collection is enabled on a task basis, and thus it can capture precise coverage of a single system call.

Note that kcov does not aim to collect as much coverage as possible. It aims to collect more or less stable coverage that is function of syscall inputs. To achieve this goal it does not collect coverage in soft/hard interrupts and instrumentation of some inherently non-deterministic parts of kernel is disabled (e.g. scheduler, locking).

kcov is also able to collect comparison operands from the instrumented code (this feature currently requires that the kernel is compiled with clang).

5.1 Prerequisites

Configure the kernel with:

```
CONFIG_KCOV=y
```

CONFIG_KCOV requires gcc 6.1.0 or later.

If the comparison operands need to be collected, set:

```
CONFIG_KCOV_ENABLE_COMPARISONS=y
```

Profiling data will only become accessible once debugfs has been mounted:

```
mount -t debugfs none /sys/kernel/debug
```

5.2 Coverage collection

The following program demonstrates coverage collection from within a test program using kcov:

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/types.h>

#define KCOV_INIT_TRACE                     _IOR('c', 1, unsigned long)
#define KCOV_ENABLE                 _IO('c', 100)
#define KCOV_DISABLE                        _IO('c', 101)
#define COVER_SIZE                  (64<<10)

#define KCOV_TRACE_PC  0
#define KCOV_TRACE_CMP 1

int main(int argc, char **argv)
{
    int fd;
    unsigned long *cover, n, i;

    /* A single fd descriptor allows coverage collection on a single
     * thread.
     */
    fd = open("/sys/kernel/debug/kcov", O_RDWR);
    if (fd == -1)
            perror("open"), exit(1);
    /* Setup trace mode and trace size. */
    if (ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE))
            perror("ioctl"), exit(1);
    /* Mmap buffer shared between kernel- and user-space. */
    cover = (unsigned long*)mmap(NULL, COVER_SIZE * sizeof(unsigned long),
                                 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((void*)cover == MAP_FAILED)
            perror("mmap"), exit(1);
    /* Enable coverage collection on the current thread. */
    if (ioctl(fd, KCOV_ENABLE, KCOV_TRACE_PC))
            perror("ioctl"), exit(1);
    /* Reset coverage from the tail of the ioctl() call. */
    __atomic_store_n(&cover[0], 0, __ATOMIC_RELAXED);
    /* That's the target syscal call. */
    read(-1, NULL, 0);
    /* Read number of PCs collected. */
    n = __atomic_load_n(&cover[0], __ATOMIC_RELAXED);
    for (i = 0; i < n; i++)
            printf("0x%lx\n", cover[i + 1]);
    /* Disable coverage collection for the current thread. After this call
     * coverage can be enabled for a different thread.
     */
    if (ioctl(fd, KCOV_DISABLE, 0))
            perror("ioctl"), exit(1);
    /* Free resources. */
    if (munmap(cover, COVER_SIZE * sizeof(unsigned long)))
            perror("munmap"), exit(1);
    if (close(fd))
            perror("close"), exit(1);
    return 0;
}
```

After piping through addr2line output of the program looks as follows:

```
SyS_read
fs/read_write.c:562
__fdget_pos
fs/file.c:774
__fget_light
fs/file.c:746
__fget_light
fs/file.c:750
__fget_light
fs/file.c:760
__fdget_pos
fs/file.c:784
SyS_read
fs/read_write.c:562
```

If a program needs to collect coverage from several threads (independently), it needs to open /sys/kernel/debug/kcov in each thread separately.

The interface is fine-grained to allow efficient forking of test processes. That is, a parent process opens /sys/kernel/debug/kcov, enables trace mode, mmaps coverage buffer and then forks child processes in a loop. Child processes only need to enable coverage (disable happens automatically on thread end).

5.3 Comparison operands collection

Comparison operands collection is similar to coverage collection:

```
/* Same includes and defines as above. */

/* Number of 64-bit words per record. */
#define KCOV_WORDS_PER_CMP 4

/*
 * The format for the types of collected comparisons.
 *
 * Bit 0 shows whether one of the arguments is a compile-time constant.
 * Bits 1 & 2 contain log2 of the argument size, up to 8 bytes.
 */

#define KCOV_CMP_CONST          (1 << 0)
```

```
#define KCOV_CMP_SIZE(n)        ((n) << 1)
#define KCOV_CMP_MASK           KCOV_CMP_SIZE(3)

int main(int argc, char **argv)
{
    int fd;
    uint64_t *cover, type, arg1, arg2, is_const, size;
    unsigned long n, i;

    fd = open("/sys/kernel/debug/kcov", O_RDWR);
    if (fd == -1)
            perror("open"), exit(1);
    if (ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE))
            perror("ioctl"), exit(1);
    /*
    * Note that the buffer pointer is of type uint64_t*, because all
    * the comparison operands are promoted to uint64_t.
    */
    cover = (uint64_t *)mmap(NULL, COVER_SIZE * sizeof(unsigned long),
                                 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((void*)cover == MAP_FAILED)
            perror("mmap"), exit(1);
    /* Note KCOV_TRACE_CMP instead of KCOV_TRACE_PC. */
    if (ioctl(fd, KCOV_ENABLE, KCOV_TRACE_CMP))
            perror("ioctl"), exit(1);
    __atomic_store_n(&cover[0], 0, __ATOMIC_RELAXED);
    read(-1, NULL, 0);
    /* Read number of comparisons collected. */
    n = __atomic_load_n(&cover[0], __ATOMIC_RELAXED);
    for (i = 0; i < n; i++) {
            uint64_t ip;

            type = cover[i * KCOV_WORDS_PER_CMP + 1];
            /* arg1 and arg2 - operands of the comparison. */
            arg1 = cover[i * KCOV_WORDS_PER_CMP + 2];
            arg2 = cover[i * KCOV_WORDS_PER_CMP + 3];
            /* ip - caller address. */
            ip = cover[i * KCOV_WORDS_PER_CMP + 4];
            /* size of the operands. */
            size = 1 << ((type & KCOV_CMP_MASK) >> 1);
            /* is_const - true if either operand is a compile-time constant.*/
            is_const = type & KCOV_CMP_CONST;
            printf("ip: 0x%lx type: 0x%lx, arg1: 0x%lx, arg2: 0x%lx, "
                    "size: %lu, %s\n",
                    ip, type, arg1, arg2, size,
            is_const ? "const" : "non-const");
    }
    if (ioctl(fd, KCOV_DISABLE, 0))
            perror("ioctl"), exit(1);
    /* Free resources. */
    if (munmap(cover, COVER_SIZE * sizeof(unsigned long)))
            perror("munmap"), exit(1);
    if (close(fd))
            perror("close"), exit(1);
    return 0;
}
```

Note that the kcov modes (coverage collection or comparison operands) are mutually exclusive.

5.4 Remote coverage collection

With KCOV_ENABLE coverage is collected only for syscalls that are issued from the current process. With KCOV_REMOTE_ENABLE it's possible to collect coverage for arbitrary parts of the kernel code, provided that those parts are annotated with kcov_remote_start()/kcov_remote_stop().

This allows to collect coverage from two types of kernel background threads: the global ones, that are spawned during kernel boot in a limited number of instances (e.g. one USB hub_event() worker thread is spawned per USB HCD); and the local ones, that are spawned when a user interacts with some kernel interface (e.g. vhost workers); as well as from soft interrupts.

To enable collecting coverage from a global background thread or from a softirq, a unique global handle must be assigned and passed to the corresponding kcov_remote_start() call. Then a userspace process can pass a list of such handles to the KCOV_REMOTE_ENABLE ioctl in the handles array field of the kcov_remote_arg struct. This will attach the used kcov device to the code sections, that are referenced by those handles.

Since there might be many local background threads spawned from different userspace processes, we can't use a single global handle per annotation. Instead, the userspace process passes a non-zero handle through the common_handle field of the kcov_remote_arg struct. This common handle gets saved to the kcov_handle field in the current task_struct and needs to be passed to the newly spawned threads via custom annotations. Those threads should in turn be annotated with kcov_remote_start()/kcov_remote_stop().

Internally kcov stores handles as u64 integers. The top byte of a handle is used to denote the id of a subsystem that this handle belongs to, and the lower 4 bytes are used to denote the id of a thread instance within that subsystem. A reserved value 0 is used as a subsystem id for common handles as they don't belong to a particular subsystem. The bytes 4-7 are currently reserved and must be zero. In the future the number of bytes used for the subsystem or handle ids might be increased.

When a particular userspace process collects coverage via a common handle, kcov will collect coverage for each code section that is annotated to use the common handle obtained as kcov_handle from the current task_struct. However non common handles allow to collect coverage selectively from different subsystems.

```
/* Same includes and defines as above. */

struct kcov_remote_arg {
    __u32           trace_mode;
    __u32           area_size;
    __u32           num_handles;
    __aligned_u64   common_handle;
    __aligned_u64   handles[0];
};

#define KCOV_INIT_TRACE                     _IOR('c', 1, unsigned long)
#define KCOV_DISABLE                        _IO('c', 101)
#define KCOV_REMOTE_ENABLE          _IOW('c', 102, struct kcov_remote_arg)

#define COVER_SIZE  (64 << 10)

#define KCOV_TRACE_PC       0

#define KCOV_SUBSYSTEM_COMMON       (0x00ull << 56)
#define KCOV_SUBSYSTEM_USB  (0x01ull << 56)

#define KCOV_SUBSYSTEM_MASK (0xffull << 56)
#define KCOV_INSTANCE_MASK  (0xffffffffull)

static inline __u64 kcov_remote_handle(__u64 subsys, __u64 inst)
{
    if (subsys & ~KCOV_SUBSYSTEM_MASK || inst & ~KCOV_INSTANCE_MASK)
            return 0;
    return subsys | inst;
}

#define KCOV_COMMON_ID      0x42
#define KCOV_USB_BUS_NUM    1

int main(int argc, char **argv)
{
    int fd;
    unsigned long *cover, n, i;
    struct kcov_remote_arg *arg;

    fd = open("/sys/kernel/debug/kcov", O_RDWR);
    if (fd == -1)
            perror("open"), exit(1);
    if (ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE))
            perror("ioctl"), exit(1);
    cover = (unsigned long*)mmap(NULL, COVER_SIZE * sizeof(unsigned long),
                                 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((void*)cover == MAP_FAILED)
            perror("mmap"), exit(1);

    /* Enable coverage collection via common handle and from USB bus #1. */
    arg = calloc(1, sizeof(*arg) + sizeof(uint64_t));
    if (!arg)
            perror("calloc"), exit(1);
    arg->trace_mode = KCOV_TRACE_PC;
    arg->area_size = COVER_SIZE;
    arg->num_handles = 1;
    arg->common_handle = kcov_remote_handle(KCOV_SUBSYSTEM_COMMON,
                                                    KCOV_COMMON_ID);
    arg->handles[0] = kcov_remote_handle(KCOV_SUBSYSTEM_USB,
                                            KCOV_USB_BUS_NUM);
    if (ioctl(fd, KCOV_REMOTE_ENABLE, arg))
            perror("ioctl"), free(arg), exit(1);
    free(arg);

    /*
     * Here the user needs to trigger execution of a kernel code section
     * that is either annotated with the common handle, or to trigger some
     * activity on USB bus #1.
     */
    sleep(2);

    n = __atomic_load_n(&cover[0], __ATOMIC_RELAXED);
    for (i = 0; i < n; i++)
            printf("0x%lx\n", cover[i + 1]);
    if (ioctl(fd, KCOV_DISABLE, 0))
            perror("ioctl"), exit(1);
    if (munmap(cover, COVER_SIZE * sizeof(unsigned long)))
            perror("munmap"), exit(1);
    if (close(fd))
            perror("close"), exit(1);
    return 0;
}
```

CHAPTER

SIX

USING GCOV WITH THE LINUX KERNEL

gcov profiling kernel support enables the use of GCC's coverage testing tool gcov with the Linux kernel. Coverage data of a running kernel is exported in gcov-compatible format via the "gcov" debugfs directory. To get coverage data for a specific file, change to the kernel build directory and use gcov with the `-o` option as follows (requires root):

```
# cd /tmp/linux-out
# gcov -o /sys/kernel/debug/gcov/tmp/linux-out/kernel spinlock.c
```

This will create source code files annotated with execution counts in the current directory. In addition, graphical gcov front-ends such as lcov can be used to automate the process of collecting data for the entire kernel and provide coverage overviews in HTML format.

Possible uses:

- debugging (has this line been reached at all?)
- test improvement (how do I change my test to cover these lines?)
- minimizing kernel configurations (do I need this option if the associated code is never run?)

6.1 Preparation

Configure the kernel with:

```
CONFIG_DEBUG_FS=y
CONFIG_GCOV_KERNEL=y
```

and to get coverage data for the entire kernel:

```
CONFIG_GCOV_PROFILE_ALL=y
```

Note that kernels compiled with profiling flags will be significantly larger and run slower. Also CONFIG_GCOV_PROFILE_ALL may not be supported on all architectures.

Profiling data will only become accessible once debugfs has been mounted:

```
mount -t debugfs none /sys/kernel/debug
```

6.2 Customization

To enable profiling for specific files or directories, add a line similar to the following to the respective kernel Makefile:

- For a single file (e.g. main.o):

  ```
  GCOV_PROFILE_main.o := y
  ```

- For all files in one directory:

  ```
  GCOV_PROFILE := y
  ```

To exclude files from being profiled even when CONFIG_GCOV_PROFILE_ALL is specified, use:

```
GCOV_PROFILE_main.o := n
```

and:

```
GCOV_PROFILE := n
```

Only files which are linked to the main kernel image or are compiled as kernel modules are supported by this mechanism.

6.3 Files

The gcov kernel support creates the following files in debugfs:

`/sys/kernel/debug/gcov` Parent directory for all gcov-related files.

`/sys/kernel/debug/gcov/reset` Global reset file: resets all coverage data to zero when written to.

`/sys/kernel/debug/gcov/path/to/compile/dir/file.gcda` The actual gcov data file as understood by the gcov tool. Resets file coverage data to zero when written to.

`/sys/kernel/debug/gcov/path/to/compile/dir/file.gcno` Symbolic link to a static data file required by the gcov tool. This file is generated by gcc when compiling with option `-ftest-coverage`.

6.4 Modules

Kernel modules may contain cleanup code which is only run during module unload time. The gcov mechanism provides a means to collect coverage data for such code by keeping a copy of the data associated with the unloaded module. This data remains available through debugfs. Once the module is loaded again, the associated coverage counters are initialized with the data from its previous instantiation.

This behavior can be deactivated by specifying the gcov_persist kernel parameter:

```
gcov_persist=0
```

At run-time, a user can also choose to discard data for an unloaded module by writing to its data file or the global reset file.

6.5 Separated build and test machines

The gcov kernel profiling infrastructure is designed to work out-of-the box for setups where kernels are built and run on the same machine. In cases where the kernel runs on a separate machine, special preparations must be made, depending on where the gcov tool is used:

a) gcov is run on the TEST machine

   The gcov tool version on the test machine must be compatible with the gcc version used for kernel build. Also the following files need to be copied from build to test machine:

   from the source tree:

   - all C source files + headers

   from the build tree:

   - all C source files + headers
   - all .gcda and .gcno files
   - all links to directories

   It is important to note that these files need to be placed into the exact same file system location on the test machine as on the build machine. If any of the path components is symbolic link, the actual directory needs to be used instead (due to make's CURDIR handling).

b) gcov is run on the BUILD machine

   The following files need to be copied after each test case from test to build machine:

   from the gcov directory in sysfs:

   - all .gcda files
   - all links to .gcno files

   These files can be copied to any location on the build machine. gcov must then be called with the -o option pointing to that directory.

   Example directory setup on the build machine:

```
/tmp/linux:    kernel source tree
/tmp/out:      kernel build directory as specified by make O=
/tmp/coverage: location of the files copied from the test machine

[user@build] cd /tmp/out
[user@build] gcov -o /tmp/coverage/tmp/out/init main.c
```

6.6 Note on compilers

GCC and LLVM gcov tools are not necessarily compatible. Use gcov to work with GCC-generated .gcno and .gcda files, and use llvm-cov for Clang.

Build differences between GCC and Clang gcov are handled by Kconfig. It automatically selects the appropriate gcov format depending on the detected toolchain.

6.7 Troubleshooting

Problem Compilation aborts during linker step.

Cause Profiling flags are specified for source files which are not linked to the main kernel or which are linked by a custom linker procedure.

Solution Exclude affected source files from profiling by specifying `GCOV_PROFILE := n` or `GCOV_PROFILE_basename.o := n` in the corresponding Makefile.

Problem Files copied from sysfs appear empty or incomplete.

Cause Due to the way seq_file works, some tools such as cp or tar may not correctly copy files from sysfs.

Solution Use `cat` to read `.gcda` files and `cp -d` to copy links. Alternatively use the mechanism shown in Appendix B.

6.8 Appendix A: gather_on_build.sh

Sample script to gather coverage meta files on the build machine (see *Separated build and test machines a.*):

```
#!/bin/bash

KSRC=$1
KOBJ=$2
DEST=$3

if [ -z "$KSRC" ] || [ -z "$KOBJ" ] || [ -z "$DEST" ]; then
  echo "Usage: $0 <ksrc directory> <kobj directory> <output.tar.gz>" >&2
  exit 1
fi

KSRC=$(cd $KSRC; printf "all:\n\t@echo \${CURDIR}\n" | make -f -)
KOBJ=$(cd $KOBJ; printf "all:\n\t@echo \${CURDIR}\n" | make -f -)

find $KSRC $KOBJ \( -name '*.gcno' -o -name '*.[ch]' -o -type l \) -a \
                 -perm /u+r,g+r | tar cfz $DEST -P -T -

if [ $? -eq 0 ] ; then
  echo "$DEST successfully created, copy to test system and unpack with:"
  echo "  tar xfz $DEST -P"
```

```
else
  echo "Could not create file $DEST"
fi
```

6.9 Appendix B: gather_on_test.sh

Sample script to gather coverage data files on the test machine (see *Separated build and test machines b.*):

```
#!/bin/bash -e

DEST=$1
GCDA=/sys/kernel/debug/gcov

if [ -z "$DEST" ] ; then
  echo "Usage: $0 <output.tar.gz>" >&2
  exit 1
fi

TEMPDIR=$(mktemp -d)
echo Collecting data..
find $GCDA -type d -exec mkdir -p $TEMPDIR/\{\} \;
find $GCDA -name '*.gcda' -exec sh -c 'cat < $0 > '$TEMPDIR'/$0' {} \;
find $GCDA -name '*.gcno' -exec sh -c 'cp -d $0 '$TEMPDIR'/$0' {} \;
tar czf $DEST -C $TEMPDIR sys
rm -rf $TEMPDIR

echo "$DEST successfully created, copy to build system and unpack with:"
echo "  tar xfz $DEST"
```

CHAPTER

SEVEN

THE KERNEL ADDRESS SANITIZER (KASAN)

7.1 Overview

Kernel Address Sanitizer (KASAN) is a dynamic memory safety error detector designed to find out-of-bounds and use-after-free bugs.

KASAN has three modes:

1. Generic KASAN
2. Software Tag-Based KASAN
3. Hardware Tag-Based KASAN

Generic KASAN, enabled with CONFIG_KASAN_GENERIC, is the mode intended for debugging, similar to userspace ASan. This mode is supported on many CPU architectures, but it has significant performance and memory overheads.

Software Tag-Based KASAN or SW_TAGS KASAN, enabled with CONFIG_KASAN_SW_TAGS, can be used for both debugging and dogfood testing, similar to userspace HWASan. This mode is only supported for arm64, but its moderate memory overhead allows using it for testing on memory-restricted devices with real workloads.

Hardware Tag-Based KASAN or HW_TAGS KASAN, enabled with CONFIG_KASAN_HW_TAGS, is the mode intended to be used as an in-field memory bug detector or as a security mitigation. This mode only works on arm64 CPUs that support MTE (Memory Tagging Extension), but it has low memory and performance overheads and thus can be used in production.

For details about the memory and performance impact of each KASAN mode, see the descriptions of the corresponding Kconfig options.

The Generic and the Software Tag-Based modes are commonly referred to as the software modes. The Software Tag-Based and the Hardware Tag-Based modes are referred to as the tag-based modes.

7.2 Support

7.2.1 Architectures

Generic KASAN is supported on x86_64, arm, arm64, powerpc, riscv, s390, and xtensa, and the tag-based KASAN modes are supported only on arm64.

7.2.2 Compilers

Software KASAN modes use compile-time instrumentation to insert validity checks before every memory access and thus require a compiler version that provides support for that. The Hardware Tag-Based mode relies on hardware to perform these checks but still requires a compiler version that supports the memory tagging instructions.

Generic KASAN requires GCC version 8.3.0 or later or any Clang version supported by the kernel.

Software Tag-Based KASAN requires GCC 11+ or any Clang version supported by the kernel.

Hardware Tag-Based KASAN requires GCC 10+ or Clang 12+.

7.2.3 Memory types

Generic KASAN supports finding bugs in all of slab, page_alloc, vmap, vmalloc, stack, and global memory.

Software Tag-Based KASAN supports slab, page_alloc, vmalloc, and stack memory.

Hardware Tag-Based KASAN supports slab, page_alloc, and non-executable vmalloc memory.

For slab, both software KASAN modes support SLUB and SLAB allocators, while Hardware Tag-Based KASAN only supports SLUB.

7.3 Usage

To enable KASAN, configure the kernel with:

```
CONFIG_KASAN=y
```

and choose between `CONFIG_KASAN_GENERIC` (to enable Generic KASAN), `CONFIG_KASAN_SW_TAGS` (to enable Software Tag-Based KASAN), and `CONFIG_KASAN_HW_TAGS` (to enable Hardware Tag-Based KASAN).

For the software modes, also choose between `CONFIG_KASAN_OUTLINE` and `CONFIG_KASAN_INLINE`. Outline and inline are compiler instrumentation types. The former produces a smaller binary while the latter is up to 2 times faster.

To include alloc and free stack traces of affected slab objects into reports, enable `CONFIG_STACKTRACE`. To include alloc and free stack traces of affected physical pages, enable `CONFIG_PAGE_OWNER` and boot with `page_owner=on`.

7.3.1 Boot parameters

KASAN is affected by the generic panic_on_warn command line parameter. When it is enabled, KASAN panics the kernel after printing a bug report.

By default, KASAN prints a bug report only for the first invalid memory access. With kasan_multi_shot, KASAN prints a report on every invalid access. This effectively disables panic_on_warn for KASAN reports.

Alternatively, independent of panic_on_warn, the kasan.fault= boot parameter can be used to control panic and reporting behaviour:

- kasan.fault=report or =panic controls whether to only print a KASAN report or also panic the kernel (default: report). The panic happens even if kasan_multi_shot is enabled.

Hardware Tag-Based KASAN mode (see the section about various modes below) is intended for use in production as a security mitigation. Therefore, it supports additional boot parameters that allow disabling KASAN or controlling features:

- kasan=off or =on controls whether KASAN is enabled (default: on).
- kasan.mode=sync, =async or =asymm controls whether KASAN is configured in synchronous, asynchronous or asymmetric mode of execution (default: sync). Synchronous mode: a bad access is detected immediately when a tag check fault occurs. Asynchronous mode: a bad access detection is delayed. When a tag check fault occurs, the information is stored in hardware (in the TFSR_EL1 register for arm64). The kernel periodically checks the hardware and only reports tag faults during these checks. Asymmetric mode: a bad access is detected synchronously on reads and asynchronously on writes.
- kasan.vmalloc=off or =on disables or enables tagging of vmalloc allocations (default: on).
- kasan.stacktrace=off or =on disables or enables alloc and free stack traces collection (default: on).

7.3.2 Error reports

A typical KASAN report looks like this:

```
==================================================================
BUG: KASAN: slab-out-of-bounds in kmalloc_oob_right+0xa8/0xbc [test_kasan]
Write of size 1 at addr ffff8801f44ec37b by task insmod/2760

CPU: 1 PID: 2760 Comm: insmod Not tainted 4.19.0-rc3+ #698
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1 04/01/2014
Call Trace:
 dump_stack+0x94/0xd8
 print_address_description+0x73/0x280
 kasan_report+0x144/0x187
 __asan_report_store1_noabort+0x17/0x20
 kmalloc_oob_right+0xa8/0xbc [test_kasan]
 kmalloc_tests_init+0x16/0x700 [test_kasan]
 do_one_initcall+0xa5/0x3ae
 do_init_module+0x1b6/0x547
 load_module+0x75df/0x8070
 __do_sys_init_module+0x1c6/0x200
 __x64_sys_init_module+0x6e/0xb0
 do_syscall_64+0x9f/0x2c0
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP: 0033:0x7f96443109da
RSP: 002b:00007ffcf0b51b08 EFLAGS: 00000202 ORIG_RAX: 00000000000000af
RAX: ffffffffffffffda RBX: 000055dc3ee521a0 RCX: 00007f96443109da
RDX: 00007f96445cff88 RSI: 0000000000057a50 RDI: 00007f9644992000
RBP: 000055dc3ee510b0 R08: 0000000000000003 R09: 0000000000000000
R10: 00007f964430cd0a R11: 0000000000000202 R12: 00007f96445cff88
R13: 000055dc3ee51090 R14: 0000000000000000 R15: 0000000000000000

Allocated by task 2760:
 save_stack+0x43/0xd0
 kasan_kmalloc+0xa7/0xd0
 kmem_cache_alloc_trace+0xe1/0x1b0
 kmalloc_oob_right+0x56/0xbc [test_kasan]
 kmalloc_tests_init+0x16/0x700 [test_kasan]
 do_one_initcall+0xa5/0x3ae
 do_init_module+0x1b6/0x547
 load_module+0x75df/0x8070
 __do_sys_init_module+0x1c6/0x200
 __x64_sys_init_module+0x6e/0xb0
 do_syscall_64+0x9f/0x2c0
 entry_SYSCALL_64_after_hwframe+0x44/0xa9

Freed by task 815:
 save_stack+0x43/0xd0
 __kasan_slab_free+0x135/0x190
 kasan_slab_free+0xe/0x10
 kfree+0x93/0x1a0
 umh_complete+0x6a/0xa0
 call_usermodehelper_exec_async+0x4c3/0x640
 ret_from_fork+0x35/0x40

The buggy address belongs to the object at ffff8801f44ec300
 which belongs to the cache kmalloc-128 of size 128
The buggy address is located 123 bytes inside of
 128-byte region [ffff8801f44ec300, ffff8801f44ec380)
The buggy address belongs to the page:
page:ffffea0007d13b00 count:1 mapcount:0 mapping:ffff8801f7001640 index:0x0
flags: 0x200000000000100(slab)
raw: 0200000000000100 ffffea0007d11dc0 0000001a0000001a ffff8801f7001640
raw: 0000000000000000 0000000080150015 00000001ffffffff 0000000000000000
page dumped because: kasan: bad access detected

Memory state around the buggy address:
 ffff8801f44ec200: fc fc fc fc fc fc fc fc fb fb fb fb fb fb fb fb
 ffff8801f44ec280: fb fb fb fb fb fb fb fb fc fc fc fc fc fc fc fc
>ffff8801f44ec300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
```

```
                                                                ^
 ffff8801f44ec380: fc fc fc fc fc fc fc fc fb fb fb fb fb fb fb fb
 ffff8801f44ec400: fb fb fb fb fb fb fb fb fc fc fc fc fc fc fc fc
==================================================================
```

The report header summarizes what kind of bug happened and what kind of access caused it. It is followed by a stack trace of the bad access, a stack trace of where the accessed memory was allocated (in case a slab object was accessed), and a stack trace of where the object was freed (in case of a use-after-free bug report). Next comes a description of the accessed slab object and the information about the accessed memory page.

In the end, the report shows the memory state around the accessed address. Internally, KASAN tracks memory state separately for each memory granule, which is either 8 or 16 aligned bytes depending on KASAN mode. Each number in the memory state section of the report shows the state of one of the memory granules that surround the accessed address.

For Generic KASAN, the size of each memory granule is 8. The state of each granule is encoded in one shadow byte. Those 8 bytes can be accessible, partially accessible, freed, or be a part of a redzone. KASAN uses the following encoding for each shadow byte: 00 means that all 8 bytes of the corresponding memory region are accessible; number N (1 <= N <= 7) means that the first N bytes are accessible, and other (8 - N) bytes are not; any negative value indicates that the entire 8-byte word is inaccessible. KASAN uses different negative values to distinguish between different kinds of inaccessible memory like redzones or freed memory (see mm/kasan/kasan.h).

In the report above, the arrow points to the shadow byte `03`, which means that the accessed address is partially accessible.

For tag-based KASAN modes, this last report section shows the memory tags around the accessed address (see the *Implementation details* section).

Note that KASAN bug titles (like `slab-out-of-bounds` or `use-after-free`) are best-effort: KASAN prints the most probable bug type based on the limited information it has. The actual type of the bug might be different.

Generic KASAN also reports up to two auxiliary call stack traces. These stack traces point to places in code that interacted with the object but that are not directly present in the bad access stack trace. Currently, this includes call_rcu() and workqueue queuing.

7.4 Implementation details

7.4.1 Generic KASAN

Software KASAN modes use shadow memory to record whether each byte of memory is safe to access and use compile-time instrumentation to insert shadow memory checks before each memory access.

Generic KASAN dedicates 1/8th of kernel memory to its shadow memory (16TB to cover 128TB on x86_64) and uses direct mapping with a scale and offset to translate a memory address to its corresponding shadow address.

Here is the function which translates an address to its corresponding shadow address:

```
static inline void *kasan_mem_to_shadow(const void *addr)
{
    return (void *)((unsigned long)addr >> KASAN_SHADOW_SCALE_SHIFT)
            + KASAN_SHADOW_OFFSET;
}
```

where `KASAN_SHADOW_SCALE_SHIFT = 3`.

Compile-time instrumentation is used to insert memory access checks. Compiler inserts function calls (`__asan_load*(addr)`, `__asan_store*(addr)`) before each memory access of size 1, 2, 4, 8, or 16. These functions check whether memory accesses are valid or not by checking corresponding shadow memory.

With inline instrumentation, instead of making function calls, the compiler directly inserts the code to check shadow memory. This option significantly enlarges the kernel, but it gives an x1.1-x2 performance boost over the outline-instrumented kernel.

Generic KASAN is the only mode that delays the reuse of freed objects via quarantine (see mm/kasan/quarantine.c for implementation).

7.4.2 Software Tag-Based KASAN

Software Tag-Based KASAN uses a software memory tagging approach to checking access validity. It is currently only implemented for the arm64 architecture.

Software Tag-Based KASAN uses the Top Byte Ignore (TBI) feature of arm64 CPUs to store a pointer tag in the top byte of kernel pointers. It uses shadow memory to store memory tags associated with each 16-byte memory cell (therefore, it dedicates 1/16th of the kernel memory for shadow memory).

On each memory allocation, Software Tag-Based KASAN generates a random tag, tags the allocated memory with this tag, and embeds the same tag into the returned pointer.

Software Tag-Based KASAN uses compile-time instrumentation to insert checks before each memory access. These checks make sure that the tag of the memory that is being accessed is equal to the tag of the pointer that is used to access this memory. In case of a tag mismatch, Software Tag-Based KASAN prints a bug report.

Software Tag-Based KASAN also has two instrumentation modes (outline, which emits callbacks to check memory accesses; and inline, which performs the shadow memory checks inline). With outline instrumentation mode, a bug report is printed from the function that performs the access check. With inline instrumentation, a `brk` instruction is emitted by the compiler, and a dedicated `brk` handler is used to print bug reports.

Software Tag-Based KASAN uses 0xFF as a match-all pointer tag (accesses through pointers with the 0xFF pointer tag are not checked). The value 0xFE is currently reserved to tag freed memory regions.

7.4.3 Hardware Tag-Based KASAN

Hardware Tag-Based KASAN is similar to the software mode in concept but uses hardware memory tagging support instead of compiler instrumentation and shadow memory.

Hardware Tag-Based KASAN is currently only implemented for arm64 architecture and based on both arm64 Memory Tagging Extension (MTE) introduced in ARMv8.5 Instruction Set Architecture and Top Byte Ignore (TBI).

Special arm64 instructions are used to assign memory tags for each allocation. Same tags are assigned to pointers to those allocations. On every memory access, hardware makes sure that the tag of the memory that is being accessed is equal to the tag of the pointer that is used to access this memory. In case of a tag mismatch, a fault is generated, and a report is printed.

Hardware Tag-Based KASAN uses 0xFF as a match-all pointer tag (accesses through pointers with the 0xFF pointer tag are not checked). The value 0xFE is currently reserved to tag freed memory regions.

If the hardware does not support MTE (pre ARMv8.5), Hardware Tag-Based KASAN will not be enabled. In this case, all KASAN boot parameters are ignored.

Note that enabling CONFIG_KASAN_HW_TAGS always results in in-kernel TBI being enabled. Even when `kasan.mode=off` is provided or when the hardware does not support MTE (but supports TBI).

Hardware Tag-Based KASAN only reports the first found bug. After that, MTE tag checking gets disabled.

7.5 Shadow memory

The contents of this section are only applicable to software KASAN modes.

The kernel maps memory in several different parts of the address space. The range of kernel virtual addresses is large: there is not enough real memory to support a real shadow region for every address that could be accessed by the kernel. Therefore, KASAN only maps real shadow for certain parts of the address space.

7.5.1 Default behaviour

By default, architectures only map real memory over the shadow region for the linear mapping (and potentially other small areas). For all other areas - such as vmalloc and vmemmap space - a single read-only page is mapped over the shadow area. This read-only shadow page declares all memory accesses as permitted.

This presents a problem for modules: they do not live in the linear mapping but in a dedicated module space. By hooking into the module allocator, KASAN temporarily maps real shadow memory to cover them. This allows detection of invalid accesses to module globals, for example.

This also creates an incompatibility with `VMAP_STACK`: if the stack lives in vmalloc space, it will be shadowed by the read-only page, and the kernel will fault when trying to set up the shadow data for stack variables.

7.5.2 CONFIG_KASAN_VMALLOC

With `CONFIG_KASAN_VMALLOC`, KASAN can cover vmalloc space at the cost of greater memory usage. Currently, this is supported on x86, arm64, riscv, s390, and powerpc.

This works by hooking into vmalloc and vmap and dynamically allocating real shadow memory to back the mappings.

Most mappings in vmalloc space are small, requiring less than a full page of shadow space. Allocating a full shadow page per mapping would therefore be wasteful. Furthermore, to ensure that different mappings use different shadow pages, mappings would have to be aligned to `KASAN_GRANULE_SIZE * PAGE_SIZE`.

Instead, KASAN shares backing space across multiple mappings. It allocates a backing page when a mapping in vmalloc space uses a particular page of the shadow region. This page can be shared by other vmalloc mappings later on.

KASAN hooks into the vmap infrastructure to lazily clean up unused shadow memory.

To avoid the difficulties around swapping mappings around, KASAN expects that the part of the shadow region that covers the vmalloc space will not be covered by the early shadow page but will be left unmapped. This will require changes in arch-specific code.

This allows `VMAP_STACK` support on x86 and can simplify support of architectures that do not have a fixed module region.

7.6 For developers

7.6.1 Ignoring accesses

Software KASAN modes use compiler instrumentation to insert validity checks. Such instrumentation might be incompatible with some parts of the kernel, and therefore needs to be disabled.

Other parts of the kernel might access metadata for allocated objects. Normally, KASAN detects and reports such accesses, but in some cases (e.g., in memory allocators), these accesses are valid.

For software KASAN modes, to disable instrumentation for a specific file or directory, add a `KASAN_SANITIZE` annotation to the respective kernel Makefile:

- For a single file (e.g., main.o):

  ```
  KASAN_SANITIZE_main.o := n
  ```

- For all files in one directory:

  ```
  KASAN_SANITIZE := n
  ```

For software KASAN modes, to disable instrumentation on a per-function basis, use the KASAN-specific `__no_sanitize_address` function attribute or the generic `noinstr` one.

Note that disabling compiler instrumentation (either on a per-file or a per-function basis) makes KASAN ignore the accesses that happen directly in that code for software KASAN modes. It does not help when the accesses happen indirectly (through calls to instrumented functions) or with Hardware Tag-Based KASAN, which does not use compiler instrumentation.

For software KASAN modes, to disable KASAN reports in a part of the kernel code for the current task, annotate this part of the code with a `kasan_disable_current()/kasan_enable_current()` section. This also disables the reports for indirect accesses that happen through function calls.

For tag-based KASAN modes, to disable access checking, use `kasan_reset_tag()` or `page_kasan_tag_reset()`. Note that temporarily disabling access checking via `page_kasan_tag_reset()` requires saving and restoring the per-page KASAN tag via `page_kasan_tag/page_kasan_tag_set`.

7.6.2 Tests

There are KASAN tests that allow verifying that KASAN works and can detect certain types of memory corruptions. The tests consist of two parts:

1. Tests that are integrated with the KUnit Test Framework. Enabled with `CONFIG_KASAN_KUNIT_TEST`. These tests can be run and partially verified automatically in a few different ways; see the instructions below.

2. Tests that are currently incompatible with KUnit. Enabled with `CONFIG_KASAN_MODULE_TEST` and can only be run as a module. These tests can only be verified manually by loading the kernel module and inspecting the kernel log for KASAN reports.

Each KUnit-compatible KASAN test prints one of multiple KASAN reports if an error is detected. Then the test prints its number and status.

When a test passes:

```
ok 28 - kmalloc_double_kzfree
```

When a test fails due to a failed `kmalloc`:

```
# kmalloc_large_oob_right: ASSERTION FAILED at lib/test_kasan.c:163
Expected ptr is not null, but is
not ok 4 - kmalloc_large_oob_right
```

When a test fails due to a missing KASAN report:

```
# kmalloc_double_kzfree: EXPECTATION FAILED at lib/test_kasan.c:974
KASAN failure expected in "kfree_sensitive(ptr)", but none occurred
not ok 44 - kmalloc_double_kzfree
```

At the end the cumulative status of all KASAN tests is printed. On success:

```
ok 1 - kasan
```

Or, if one of the tests failed:

```
not ok 1 - kasan
```

There are a few ways to run KUnit-compatible KASAN tests.

1. Loadable module

   With `CONFIG_KUNIT` enabled, KASAN-KUnit tests can be built as a loadable module and run by loading `test_kasan.ko` with `insmod` or `modprobe`.

2. Built-In

   With `CONFIG_KUNIT` built-in, KASAN-KUnit tests can be built-in as well. In this case, the tests will run at boot as a late-init call.

3. Using kunit_tool

   With `CONFIG_KUNIT` and `CONFIG_KASAN_KUNIT_TEST` built-in, it is also possible to use `kunit_tool` to see the results of KUnit tests in a more readable way. This will not print the KASAN reports of the tests that passed. See KUnit documentation for more up-to-date information on `kunit_tool`.

CHAPTER

EIGHT

THE UNDEFINED BEHAVIOR SANITIZER - UBSAN

UBSAN is a runtime undefined behaviour checker.

UBSAN uses compile-time instrumentation to catch undefined behavior (UB). Compiler inserts code that perform certain kinds of checks before operations that may cause UB. If check fails (i.e. UB detected) __ubsan_handle_* function called to print error message.

GCC has that feature since 4.9.x [1] (see `-fsanitize=undefined` option and its suboptions). GCC 5.x has more checkers implemented [2].

8.1 Report example

```
================================================================================
UBSAN: Undefined behaviour in ../include/linux/bitops.h:110:33
shift exponent 32 is to large for 32-bit type 'unsigned int'
CPU: 0 PID: 0 Comm: swapper Not tainted 4.4.0-rc1+ #26
 0000000000000000 ffffffff82403cc8 ffffffff815e6cd6 0000000000000001
 ffffffff82403cf8 ffffffff82403ce0 ffffffff8163a5ed 0000000000000020
 ffffffff82403d78 ffffffff8163ac2b ffffffff815f0001 0000000000000002
Call Trace:
 [<ffffffff815e6cd6>] dump_stack+0x45/0x5f
 [<ffffffff8163a5ed>] ubsan_epilogue+0xd/0x40
 [<ffffffff8163ac2b>] __ubsan_handle_shift_out_of_bounds+0xeb/0x130
 [<ffffffff815f0001>] ? radix_tree_gang_lookup_slot+0x51/0x150
 [<ffffffff8173c586>] _mix_pool_bytes+0x1e6/0x480
 [<ffffffff83105653>] ? dmi_walk_early+0x48/0x5c
 [<ffffffff8173c881>] add_device_randomness+0x61/0x130
 [<ffffffff83105b35>] ? dmi_save_one_device+0xaa/0xaa
 [<ffffffff83105653>] dmi_walk_early+0x48/0x5c
 [<ffffffff831066ae>] dmi_scan_machine+0x278/0x4b4
 [<ffffffff8111d58a>] ? vprintk_default+0x1a/0x20
 [<ffffffff830ad120>] ? early_idt_handler_array+0x120/0x120
 [<ffffffff830b2240>] setup_arch+0x405/0xc2c
 [<ffffffff830ad120>] ? early_idt_handler_array+0x120/0x120
 [<ffffffff830ae053>] start_kernel+0x83/0x49a
 [<ffffffff830ad120>] ? early_idt_handler_array+0x120/0x120
 [<ffffffff830ad386>] x86_64_start_reservations+0x2a/0x2c
 [<ffffffff830ad4f3>] x86_64_start_kernel+0x16b/0x17a
================================================================================
```

8.2 Usage

To enable UBSAN configure kernel with:

```
CONFIG_UBSAN=y
```

and to check the entire kernel:

```
CONFIG_UBSAN_SANITIZE_ALL=y
```

To enable instrumentation for specific files or directories, add a line similar to the following to the respective kernel Makefile:

- For a single file (e.g. main.o):

  ```
  UBSAN_SANITIZE_main.o := y
  ```

- For all files in one directory:

  ```
  UBSAN_SANITIZE := y
  ```

To exclude files from being instrumented even if `CONFIG_UBSAN_SANITIZE_ALL=y`, use:

```
UBSAN_SANITIZE_main.o := n
```

and:

```
UBSAN_SANITIZE := n
```

Detection of unaligned accesses controlled through the separate option - CONFIG_UBSAN_ALIGNMENT. It's off by default on architectures that support unaligned accesses (CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS=y). One could still enable it in config, just note that it will produce a lot of UBSAN reports.

8.3 References

CHAPTER

NINE

KERNEL MEMORY LEAK DETECTOR

Kmemleak provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector, with the difference that the orphan objects are not freed but only reported via /sys/kernel/debug/kmemleak. A similar method is used by the Valgrind tool (`memcheck --leak-check`) to detect the memory leaks in user-space applications.

9.1 Usage

CONFIG_DEBUG_KMEMLEAK in "Kernel hacking" has to be enabled. A kernel thread scans the memory every 10 minutes (by default) and prints the number of new unreferenced objects found. If the `debugfs` isn't already mounted, mount with:

```
# mount -t debugfs nodev /sys/kernel/debug/
```

To display the details of all the possible scanned memory leaks:

```
# cat /sys/kernel/debug/kmemleak
```

To trigger an intermediate memory scan:

```
# echo scan > /sys/kernel/debug/kmemleak
```

To clear the list of all current possible memory leaks:

```
# echo clear > /sys/kernel/debug/kmemleak
```

New leaks will then come up upon reading `/sys/kernel/debug/kmemleak` again.

Note that the orphan objects are listed in the order they were allocated and one object at the beginning of the list may cause other subsequent objects to be reported as orphan.

Memory scanning parameters can be modified at run-time by writing to the `/sys/kernel/debug/kmemleak` file. The following parameters are supported:

- **off** disable kmemleak (irreversible)
- **stack=on** enable the task stacks scanning (default)
- **stack=off** disable the tasks stacks scanning
- **scan=on** start the automatic memory scanning thread (default)
- **scan=off** stop the automatic memory scanning thread

- **scan=<secs>** set the automatic memory scanning period in seconds (default 600, 0 to stop the automatic scanning)
- **scan** trigger a memory scan
- **clear** clear list of current memory leak suspects, done by marking all current reported unreferenced objects grey, or free all kmemleak objects if kmemleak has been disabled.
- **dump=<addr>** dump information about the object found at <addr>

Kmemleak can also be disabled at boot-time by passing `kmemleak=off` on the kernel command line.

Memory may be allocated or freed before kmemleak is initialised and these actions are stored in an early log buffer. The size of this buffer is configured via the CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE option.

If CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF are enabled, the kmemleak is disabled by default. Passing `kmemleak=on` on the kernel command line enables the function.

If you are getting errors like "Error while writing to stdout" or "write_loop: Invalid argument", make sure kmemleak is properly enabled.

9.2 Basic Algorithm

The memory allocations via `kmalloc()`, `vmalloc()`, `kmem_cache_alloc()` and friends are traced and the pointers, together with additional information like size and stack trace, are stored in a rbtree. The corresponding freeing function calls are tracked and the pointers removed from the kmemleak data structures.

An allocated block of memory is considered orphan if no pointer to its start address or to any location inside the block can be found by scanning the memory (including saved registers). This means that there might be no way for the kernel to pass the address of the allocated block to a freeing function and therefore the block is considered a memory leak.

The scanning algorithm steps:

1. mark all objects as white (remaining white objects will later be considered orphan)
2. scan the memory starting with the data section and stacks, checking the values against the addresses stored in the rbtree. If a pointer to a white object is found, the object is added to the gray list
3. scan the gray objects for matching addresses (some white objects can become gray and added at the end of the gray list) until the gray set is finished
4. the remaining white objects are considered orphan and reported via /sys/kernel/debug/kmemleak

Some allocated memory blocks have pointers stored in the kernel's internal data structures and they cannot be detected as orphans. To avoid this, kmemleak can also store the number of values pointing to an address inside the block address range that need to be found so that the block is not considered a leak. One example is __vmalloc().

9.3 Testing specific sections with kmemleak

Upon initial bootup your /sys/kernel/debug/kmemleak output page may be quite extensive. This can also be the case if you have very buggy code when doing development. To work around these situations you can use the 'clear' command to clear all reported unreferenced objects from the /sys/kernel/debug/kmemleak output. By issuing a 'scan' after a 'clear' you can find new unreferenced objects; this should help with testing specific sections of code.

To test a critical section on demand with a clean kmemleak do:

```
# echo clear > /sys/kernel/debug/kmemleak
... test your kernel or modules ...
# echo scan > /sys/kernel/debug/kmemleak
```

Then as usual to get your report with:

```
# cat /sys/kernel/debug/kmemleak
```

9.4 Freeing kmemleak internal objects

To allow access to previously found memory leaks after kmemleak has been disabled by the user or due to an fatal error, internal kmemleak objects won't be freed when kmemleak is disabled, and those objects may occupy a large part of physical memory.

In this situation, you may reclaim memory with:

```
# echo clear > /sys/kernel/debug/kmemleak
```

9.5 Kmemleak API

See the include/linux/kmemleak.h header for the functions prototype.

- `kmemleak_init` - initialize kmemleak
- `kmemleak_alloc` - notify of a memory block allocation
- `kmemleak_alloc_percpu` - notify of a percpu memory block allocation
- `kmemleak_vmalloc` - notify of a vmalloc() memory allocation
- `kmemleak_free` - notify of a memory block freeing
- `kmemleak_free_part` - notify of a partial memory block freeing
- `kmemleak_free_percpu` - notify of a percpu memory block freeing
- `kmemleak_update_trace` - update object allocation stack trace
- `kmemleak_not_leak` - mark an object as not a leak
- `kmemleak_ignore` - do not scan or report an object as leak
- `kmemleak_scan_area` - add scan areas inside a memory block
- `kmemleak_no_scan` - do not scan a memory block

- `kmemleak_erase` - erase an old value in a pointer variable
- `kmemleak_alloc_recursive` - as kmemleak_alloc but checks the recursiveness
- `kmemleak_free_recursive` - as kmemleak_free but checks the recursiveness

The following functions take a physical address as the object pointer and only perform the corresponding action if the address has a lowmem mapping:

- `kmemleak_alloc_phys`
- `kmemleak_free_part_phys`
- `kmemleak_not_leak_phys`
- `kmemleak_ignore_phys`

9.6 Dealing with false positives/negatives

The false negatives are real memory leaks (orphan objects) but not reported by kmemleak because values found during the memory scanning point to such objects. To reduce the number of false negatives, kmemleak provides the kmemleak_ignore, kmemleak_scan_area, kmemleak_no_scan and kmemleak_erase functions (see above). The task stacks also increase the amount of false negatives and their scanning is not enabled by default.

The false positives are objects wrongly reported as being memory leaks (orphan). For objects known not to be leaks, kmemleak provides the kmemleak_not_leak function. The kmemleak_ignore could also be used if the memory block is known not to contain other pointers and it will no longer be scanned.

Some of the reported leaks are only transient, especially on SMP systems, because of pointers temporarily stored in CPU registers or stacks. Kmemleak defines MSECS_MIN_AGE (defaulting to 1000) representing the minimum age of an object to be reported as a memory leak.

9.7 Limitations and Drawbacks

The main drawback is the reduced performance of memory allocation and freeing. To avoid other penalties, the memory scanning is only performed when the /sys/kernel/debug/kmemleak file is read. Anyway, this tool is intended for debugging purposes where the performance might not be the most important requirement.

To keep the algorithm simple, kmemleak scans for values pointing to any address inside a block's address range. This may lead to an increased number of false negatives. However, it is likely that a real memory leak will eventually become visible.

Another source of false negatives is the data stored in non-pointer values. In a future version, kmemleak could only scan the pointer members in the allocated structures. This feature would solve many of the false negative cases described above.

The tool can report false positives. These are cases where an allocated block doesn't need to be freed (some cases in the init_call functions), the pointer is calculated by other methods than the usual container_of macro or the pointer is stored in a location not scanned by kmemleak.

Page allocations and ioremap are not tracked.

9.8 Testing with kmemleak-test

To check if you have all set up to use kmemleak, you can use the kmemleak-test module, a module that deliberately leaks memory. Set CONFIG_DEBUG_KMEMLEAK_TEST as module (it can't be used as built-in) and boot the kernel with kmemleak enabled. Load the module and perform a scan with:

```
# modprobe kmemleak-test
# echo scan > /sys/kernel/debug/kmemleak
```

Note that the you may not get results instantly or on the first scanning. When kmemleak gets results, it'll log `kmemleak: <count of leaks> new suspected memory leaks`. Then read the file to see then:

```
# cat /sys/kernel/debug/kmemleak
unreferenced object 0xffff89862ca702e8 (size 32):
  comm "modprobe", pid 2088, jiffies 4294680594 (age 375.486s)
  hex dump (first 32 bytes):
    6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
    6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5  kkkkkkkkkkkkkkk.
  backtrace:
    [<00000000e0a73ec7>] 0xffffffffc01d2036
    [<000000000c5d2a46>] do_one_initcall+0x41/0x1df
    [<0000000046db7e0a>] do_init_module+0x55/0x200
    [<00000000542b9814>] load_module+0x203c/0x2480
    [<00000000c2850256>] __do_sys_finit_module+0xba/0xe0
    [<000000006564e7ef>] do_syscall_64+0x43/0x110
    [<000000007c873fa6>] entry_SYSCALL_64_after_hwframe+0x44/0xa9
...
```

Removing the module with `rmmod kmemleak_test` should also trigger some kmemleak results.

CHAPTER

TEN

THE KERNEL CONCURRENCY SANITIZER (KCSAN)

The Kernel Concurrency Sanitizer (KCSAN) is a dynamic race detector, which relies on compile-time instrumentation, and uses a watchpoint-based sampling approach to detect races. KCSAN's primary purpose is to detect *data races*.

10.1 Usage

KCSAN is supported by both GCC and Clang. With GCC we require version 11 or later, and with Clang also require version 11 or later.

To enable KCSAN configure the kernel with:

```
CONFIG_KCSAN = y
```

KCSAN provides several other configuration options to customize behaviour (see the respective help text in `lib/Kconfig.kcsan` for more info).

10.1.1 Error reports

A typical data race report looks like this:

```
==================================================================
BUG: KCSAN: data-race in test_kernel_read / test_kernel_write

write to 0xffffffffc009a628 of 8 bytes by task 487 on cpu 0:
 test_kernel_write+0x1d/0x30
 access_thread+0x89/0xd0
 kthread+0x23e/0x260
 ret_from_fork+0x22/0x30

read to 0xffffffffc009a628 of 8 bytes by task 488 on cpu 6:
 test_kernel_read+0x10/0x20
 access_thread+0x89/0xd0
 kthread+0x23e/0x260
 ret_from_fork+0x22/0x30

value changed: 0x00000000000009a6 -> 0x00000000000009b2

Reported by Kernel Concurrency Sanitizer on:
```

```
CPU: 6 PID: 488 Comm: access_thread Not tainted 5.12.0-rc2+ #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

The header of the report provides a short summary of the functions involved in the race. It is followed by the access types and stack traces of the 2 threads involved in the data race. If KCSAN also observed a value change, the observed old value and new value are shown on the "value changed" line respectively.

The other less common type of data race report looks like this:

```
==================================================================
BUG: KCSAN: data-race in test_kernel_rmw_array+0x71/0xd0

race at unknown origin, with read to 0xffffffffc009bdb0 of 8 bytes by task 515␣
↪on cpu 2:
 test_kernel_rmw_array+0x71/0xd0
 access_thread+0x89/0xd0
 kthread+0x23e/0x260
 ret_from_fork+0x22/0x30

value changed: 0x0000000000002328 -> 0x0000000000002329

Reported by Kernel Concurrency Sanitizer on:
CPU: 2 PID: 515 Comm: access_thread Not tainted 5.12.0-rc2+ #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

This report is generated where it was not possible to determine the other racing thread, but a race was inferred due to the data value of the watched memory location having changed. These reports always show a "value changed" line. A common reason for reports of this type are missing instrumentation in the racing thread, but could also occur due to e.g. DMA accesses. Such reports are shown only if `CONFIG_KCSAN_REPORT_RACE_UNKNOWN_ORIGIN=y`, which is enabled by default.

10.1.2 Selective analysis

It may be desirable to disable data race detection for specific accesses, functions, compilation units, or entire subsystems. For static blacklisting, the below options are available:

- KCSAN understands the `data_race(expr)` annotation, which tells KCSAN that any data races due to accesses in `expr` should be ignored and resulting behaviour when encountering a data race is deemed safe. Please see "Marking Shared-Memory Accesses" in the LKMM for more information.
- Disabling data race detection for entire functions can be accomplished by using the function attribute `__no_kcsan`:

  ```
  __no_kcsan
  void foo(void) {
      ...
  ```

To dynamically limit for which functions to generate reports, see the *DebugFS interface* blacklist/whitelist feature.

- To disable data race detection for a particular compilation unit, add to the `Makefile`:

```
KCSAN_SANITIZE_file.o := n
```

- To disable data race detection for all compilation units listed in a `Makefile`, add to the respective `Makefile`:

```
KCSAN_SANITIZE := n
```

Furthermore, it is possible to tell KCSAN to show or hide entire classes of data races, depending on preferences. These can be changed via the following Kconfig options:

- `CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY`: If enabled and a conflicting write is observed via a watchpoint, but the data value of the memory location was observed to remain unchanged, do not report the data race.
- `CONFIG_KCSAN_ASSUME_PLAIN_WRITES_ATOMIC`: Assume that plain aligned writes up to word size are atomic by default. Assumes that such writes are not subject to unsafe compiler optimizations resulting in data races. The option causes KCSAN to not report data races due to conflicts where the only plain accesses are aligned writes up to word size.
- `CONFIG_KCSAN_PERMISSIVE`: Enable additional permissive rules to ignore certain classes of common data races. Unlike the above, the rules are more complex involving value-change patterns, access type, and address. This option depends on `CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY=y`. For details please see the `kernel/kcsan/permissive.h`. Testers and maintainers that only focus on reports from specific subsystems and not the whole kernel are recommended to disable this option.

To use the strictest possible rules, select `CONFIG_KCSAN_STRICT=y`, which configures KCSAN to follow the Linux-kernel memory consistency model (LKMM) as closely as possible.

10.1.3 DebugFS interface

The file `/sys/kernel/debug/kcsan` provides the following interface:

- Reading `/sys/kernel/debug/kcsan` returns various runtime statistics.
- Writing `on` or `off` to `/sys/kernel/debug/kcsan` allows turning KCSAN on or off, respectively.
- Writing `!some_func_name` to `/sys/kernel/debug/kcsan` adds `some_func_name` to the report filter list, which (by default) blacklists reporting data races where either one of the top stackframes are a function in the list.
- Writing either `blacklist` or `whitelist` to `/sys/kernel/debug/kcsan` changes the report filtering behaviour. For example, the blacklist feature can be used to silence frequently occurring data races; the whitelist feature can help with reproduction and testing of fixes.

10.1.4 Tuning performance

Core parameters that affect KCSAN's overall performance and bug detection ability are exposed as kernel command-line arguments whose defaults can also be changed via the corresponding Kconfig options.

- `kcsan.skip_watch` (`CONFIG_KCSAN_SKIP_WATCH`): Number of per-CPU memory operations to skip, before another watchpoint is set up. Setting up watchpoints more frequently will result in the likelihood of races to be observed to increase. This parameter has the most significant impact on overall system performance and race detection ability.
- `kcsan.udelay_task` (`CONFIG_KCSAN_UDELAY_TASK`): For tasks, the microsecond delay to stall execution after a watchpoint has been set up. Larger values result in the window in which we may observe a race to increase.
- `kcsan.udelay_interrupt` (`CONFIG_KCSAN_UDELAY_INTERRUPT`): For interrupts, the microsecond delay to stall execution after a watchpoint has been set up. Interrupts have tighter latency requirements, and their delay should generally be smaller than the one chosen for tasks.

They may be tweaked at runtime via `/sys/module/kcsan/parameters/`.

10.2 Data Races

In an execution, two memory accesses form a *data race* if they *conflict*, they happen concurrently in different threads, and at least one of them is a *plain access*; they *conflict* if both access the same memory location, and at least one is a write. For a more thorough discussion and definition, see "Plain Accesses and Data Races" in the LKMM.

10.2.1 Relationship with the Linux-Kernel Memory Consistency Model (LKMM)

The LKMM defines the propagation and ordering rules of various memory operations, which gives developers the ability to reason about concurrent code. Ultimately this allows to determine the possible executions of concurrent code, and if that code is free from data races.

KCSAN is aware of *marked atomic operations* (`READ_ONCE`, `WRITE_ONCE`, `atomic_*`, etc.), and a subset of ordering guarantees implied by memory barriers. With `CONFIG_KCSAN_WEAK_MEMORY=y`, KCSAN models load or store buffering, and can detect missing `smp_mb()`, `smp_wmb()`, `smp_rmb()`, `smp_store_release()`, and all `atomic_*` operations with equivalent implied barriers.

Note, KCSAN will not report all data races due to missing memory ordering, specifically where a memory barrier would be required to prohibit subsequent memory operation from reordering before the barrier. Developers should therefore carefully consider the required memory ordering requirements that remain unchecked.

10.3 Race Detection Beyond Data Races

For code with complex concurrency design, race-condition bugs may not always manifest as data races. Race conditions occur if concurrently executing operations result in unexpected system behaviour. On the other hand, data races are defined at the C-language level. The following macros can be used to check properties of concurrent code where bugs would not manifest as data races.

ASSERT_EXCLUSIVE_WRITER

`ASSERT_EXCLUSIVE_WRITER (var)`

assert no concurrent writes to **var**

Parameters

`var` variable to assert on

Description

Assert that there are no concurrent writes to **var**; other readers are allowed. This assertion can be used to specify properties of concurrent code, where violation cannot be detected as a normal data race.

For example, if we only have a single writer, but multiple concurrent readers, to avoid data races, all these accesses must be marked; even concurrent marked writes racing with the single writer are bugs. Unfortunately, due to being marked, they are no longer data races. For cases like these, we can use the macro as follows:

```
void writer(void) {
        spin_lock(&update_foo_lock);
        ASSERT_EXCLUSIVE_WRITER(shared_foo);
        WRITE_ONCE(shared_foo, ...);
        spin_unlock(&update_foo_lock);
}
void reader(void) {
        // update_foo_lock does not need to be held!
        ... = READ_ONCE(shared_foo);
}
```

Note

`ASSERT_EXCLUSIVE_WRITER_SCOPED()`, if applicable, performs more thorough checking if a clear scope where no concurrent writes are expected exists.

ASSERT_EXCLUSIVE_WRITER_SCOPED

`ASSERT_EXCLUSIVE_WRITER_SCOPED (var)`

assert no concurrent writes to **var** in scope

Parameters

`var` variable to assert on

Description

Scoped variant of *`ASSERT_EXCLUSIVE_WRITER()`*.

Assert that there are no concurrent writes to **var** for the duration of the scope in which it is introduced. This provides a better way to fully cover the enclosing scope, compared to multiple *ASSERT_EXCLUSIVE_WRITER()*, and increases the likelihood for KCSAN to detect racing accesses.

For example, it allows finding race-condition bugs that only occur due to state changes within the scope itself:

```
void writer(void) {
        spin_lock(&update_foo_lock);
        {
                ASSERT_EXCLUSIVE_WRITER_SCOPED(shared_foo);
                WRITE_ONCE(shared_foo, 42);
                ...
                // shared_foo should still be 42 here!
        }
        spin_unlock(&update_foo_lock);
}
void buggy(void) {
        if (READ_ONCE(shared_foo) == 42)
                WRITE_ONCE(shared_foo, 1); // bug!
}
```

ASSERT_EXCLUSIVE_ACCESS

`ASSERT_EXCLUSIVE_ACCESS (var)`

assert no concurrent accesses to **var**

Parameters

var variable to assert on

Description

Assert that there are no concurrent accesses to **var** (no readers nor writers). This assertion can be used to specify properties of concurrent code, where violation cannot be detected as a normal data race.

For example, where exclusive access is expected after determining no other users of an object are left, but the object is not actually freed. We can check that this property actually holds as follows:

```
if (refcount_dec_and_test(&obj->refcnt)) {
        ASSERT_EXCLUSIVE_ACCESS(*obj);
        do_some_cleanup(obj);
        release_for_reuse(obj);
}
```

1. *ASSERT_EXCLUSIVE_ACCESS_SCOPED()*, if applicable, performs more thorough checking if a clear scope where no concurrent accesses are expected exists.
2. For cases where the object is freed, KASAN is a better fit to detect use-after-free bugs.

Note

ASSERT_EXCLUSIVE_ACCESS_SCOPED

`ASSERT_EXCLUSIVE_ACCESS_SCOPED (var)`

assert no concurrent accesses to **var** in scope

Parameters

`var` variable to assert on

Description

Scoped variant of *`ASSERT_EXCLUSIVE_ACCESS()`*.

Assert that there are no concurrent accesses to **var** (no readers nor writers) for the entire duration of the scope in which it is introduced. This provides a better way to fully cover the enclosing scope, compared to multiple *`ASSERT_EXCLUSIVE_ACCESS()`*, and increases the likelihood for KCSAN to detect racing accesses.

`ASSERT_EXCLUSIVE_BITS`

`ASSERT_EXCLUSIVE_BITS (var, mask)`

assert no concurrent writes to subset of bits in **var**

Parameters

`var` variable to assert on

`mask` only check for modifications to bits set in **mask**

Description

Bit-granular variant of *`ASSERT_EXCLUSIVE_WRITER()`*.

Assert that there are no concurrent writes to a subset of bits in **var**; concurrent readers are permitted. This assertion captures more detailed bit-level properties, compared to the other (word granularity) assertions. Only the bits set in **mask** are checked for concurrent modifications, while ignoring the remaining bits, i.e. concurrent writes (or reads) to ~mask bits are ignored.

Use this for variables, where some bits must not be modified concurrently, yet other bits are expected to be modified concurrently.

For example, variables where, after initialization, some bits are read-only, but other bits may still be modified concurrently. A reader may wish to assert that this is true as follows:

```
ASSERT_EXCLUSIVE_BITS(flags, READ_ONLY_MASK);
foo = (READ_ONCE(flags) & READ_ONLY_MASK) >> READ_ONLY_SHIFT;
```

```
ASSERT_EXCLUSIVE_BITS(flags, READ_ONLY_MASK);
foo = (flags & READ_ONLY_MASK) >> READ_ONLY_SHIFT;
```

Another example, where this may be used, is when certain bits of **var** may only be modified when holding the appropriate lock, but other bits may still be modified concurrently. Writers, where other bits may change concurrently, could use the assertion as follows:

```
spin_lock(&foo_lock);
ASSERT_EXCLUSIVE_BITS(flags, FOO_MASK);
old_flags = flags;
new_flags = (old_flags & ~FOO_MASK) | (new_foo << FOO_SHIFT);
```

```
if (cmpxchg(&flags, old_flags, new_flags) != old_flags) { ... }
spin_unlock(&foo_lock);
```

Note

The access that immediately follows *ASSERT_EXCLUSIVE_BITS()* is assumed to access the masked bits only, and KCSAN optimistically assumes it is therefore safe, even in the presence of data races, and marking it with READ_ONCE() is optional from KCSAN's point-of-view. We caution, however, that it may still be advisable to do so, since we cannot reason about all compiler optimizations when it comes to bit manipulations (on the reader and writer side). If you are sure nothing can go wrong, we can write the above simply as:

10.4 Implementation Details

KCSAN relies on observing that two accesses happen concurrently. Crucially, we want to (a) increase the chances of observing races (especially for races that manifest rarely), and (b) be able to actually observe them. We can accomplish (a) by injecting various delays, and (b) by using address watchpoints (or breakpoints).

If we deliberately stall a memory access, while we have a watchpoint for its address set up, and then observe the watchpoint to fire, two accesses to the same address just raced. Using hardware watchpoints, this is the approach taken in DataCollider. Unlike DataCollider, KCSAN does not use hardware watchpoints, but instead relies on compiler instrumentation and "soft watchpoints".

In KCSAN, watchpoints are implemented using an efficient encoding that stores access type, size, and address in a long; the benefits of using "soft watchpoints" are portability and greater flexibility. KCSAN then relies on the compiler instrumenting plain accesses. For each instrumented plain access:

1. Check if a matching watchpoint exists; if yes, and at least one access is a write, then we encountered a racing access.
2. Periodically, if no matching watchpoint exists, set up a watchpoint and stall for a small randomized delay.
3. Also check the data value before the delay, and re-check the data value after delay; if the values mismatch, we infer a race of unknown origin.

To detect data races between plain and marked accesses, KCSAN also annotates marked accesses, but only to check if a watchpoint exists; i.e. KCSAN never sets up a watchpoint on marked accesses. By never setting up watchpoints for marked operations, if all accesses to a variable that is accessed concurrently are properly marked, KCSAN will never trigger a watchpoint and therefore never report the accesses.

10.4.1 Modeling Weak Memory

KCSAN's approach to detecting data races due to missing memory barriers is based on modeling access reordering (with `CONFIG_KCSAN_WEAK_MEMORY=y`). Each plain memory access for which a watchpoint is set up, is also selected for simulated reordering within the scope of its function (at most 1 in-flight access).

Once an access has been selected for reordering, it is checked along every other access until the end of the function scope. If an appropriate memory barrier is encountered, the access will no longer be considered for simulated reordering.

When the result of a memory operation should be ordered by a barrier, KCSAN can then detect data races where the conflict only occurs as a result of a missing barrier. Consider the example:

```
int x, flag;
void T1(void)
{
    x = 1;                  // data race!
    WRITE_ONCE(flag, 1);    // correct: smp_store_release(&flag, 1)
}
void T2(void)
{
    while (!READ_ONCE(flag));   // correct: smp_load_acquire(&flag)
    ... = x;                    // data race!
}
```

When weak memory modeling is enabled, KCSAN can consider `x` in `T1` for simulated reordering. After the write of `flag`, `x` is again checked for concurrent accesses: because `T2` is able to proceed after the write of `flag`, a data race is detected. With the correct barriers in place, `x` would not be considered for reordering after the proper release of `flag`, and no data race would be detected.

Deliberate trade-offs in complexity but also practical limitations mean only a subset of data races due to missing memory barriers can be detected. With currently available compiler support, the implementation is limited to modeling the effects of "buffering" (delaying accesses), since the runtime cannot "prefetch" accesses. Also recall that watchpoints are only set up for plain accesses, and the only access type for which KCSAN simulates reordering. This means reordering of marked accesses is not modeled.

A consequence of the above is that acquire operations do not require barrier instrumentation (no prefetching). Furthermore, marked accesses introducing address or control dependencies do not require special handling (the marked access cannot be reordered, later dependent accesses cannot be prefetched).

10.4.2 Key Properties

1. **Memory Overhead:** The overall memory overhead is only a few MiB depending on configuration. The current implementation uses a small array of longs to encode watchpoint information, which is negligible.
2. **Performance Overhead:** KCSAN's runtime aims to be minimal, using an efficient watchpoint encoding that does not require acquiring any shared locks in the fast-path. For kernel boot on a system with 8 CPUs:
   - 5.0x slow-down with the default KCSAN config;
   - 2.8x slow-down from runtime fast-path overhead only (set very large `KCSAN_SKIP_WATCH` and unset `KCSAN_SKIP_WATCH_RANDOMIZE`).
3. **Annotation Overheads:** Minimal annotations are required outside the KCSAN runtime. As a result, maintenance overheads are minimal as the kernel evolves.
4. **Detects Racy Writes from Devices:** Due to checking data values upon setting up watchpoints, racy writes from devices can also be detected.
5. **Memory Ordering:** KCSAN is aware of only a subset of LKMM ordering rules; this may result in missed data races (false negatives).
6. **Analysis Accuracy:** For observed executions, due to using a sampling strategy, the analysis is *unsound* (false negatives possible), but aims to be complete (no false positives).

10.5 Alternatives Considered

An alternative data race detection approach for the kernel can be found in the Kernel Thread Sanitizer (KTSAN). KTSAN is a happens-before data race detector, which explicitly establishes the happens-before order between memory operations, which can then be used to determine data races as defined in *Data Races*.

To build a correct happens-before relation, KTSAN must be aware of all ordering rules of the LKMM and synchronization primitives. Unfortunately, any omission leads to large numbers of false positives, which is especially detrimental in the context of the kernel which includes numerous custom synchronization mechanisms. To track the happens-before relation, KTSAN's implementation requires metadata for each memory location (shadow memory), which for each page corresponds to 4 pages of shadow memory, and can translate into overhead of tens of GiB on a large system.

CHAPTER

ELEVEN

KERNEL ELECTRIC-FENCE (KFENCE)

Kernel Electric-Fence (KFENCE) is a low-overhead sampling-based memory safety error detector. KFENCE detects heap out-of-bounds access, use-after-free, and invalid-free errors.

KFENCE is designed to be enabled in production kernels, and has near zero performance overhead. Compared to KASAN, KFENCE trades performance for precision. The main motivation behind KFENCE's design, is that with enough total uptime KFENCE will detect bugs in code paths not typically exercised by non-production test workloads. One way to quickly achieve a large enough total uptime is when the tool is deployed across a large fleet of machines.

11.1 Usage

To enable KFENCE, configure the kernel with:

```
CONFIG_KFENCE=y
```

To build a kernel with KFENCE support, but disabled by default (to enable, set `kfence.sample_interval` to non-zero value), configure the kernel with:

```
CONFIG_KFENCE=y
CONFIG_KFENCE_SAMPLE_INTERVAL=0
```

KFENCE provides several other configuration options to customize behaviour (see the respective help text in `lib/Kconfig.kfence` for more info).

11.1.1 Tuning performance

The most important parameter is KFENCE's sample interval, which can be set via the kernel boot parameter `kfence.sample_interval` in milliseconds. The sample interval determines the frequency with which heap allocations will be guarded by KFENCE. The default is configurable via the Kconfig option `CONFIG_KFENCE_SAMPLE_INTERVAL`. Setting `kfence.sample_interval=0` disables KFENCE.

The sample interval controls a timer that sets up KFENCE allocations. By default, to keep the real sample interval predictable, the normal timer also causes CPU wake-ups when the system is completely idle. This may be undesirable on power-constrained systems. The boot parameter `kfence.deferrable=1` instead switches to a “deferrable” timer which does not force CPU wake-ups on idle systems, at the risk of unpredictable sample intervals. The default is configurable via the Kconfig option `CONFIG_KFENCE_DEFERRABLE`.

Warning: The KUnit test suite is very likely to fail when using a deferrable timer since it currently causes very unpredictable sample intervals.

The KFENCE memory pool is of fixed size, and if the pool is exhausted, no further KFENCE allocations occur. With `CONFIG_KFENCE_NUM_OBJECTS` (default 255), the number of available guarded objects can be controlled. Each object requires 2 pages, one for the object itself and the other one used as a guard page; object pages are interleaved with guard pages, and every object page is therefore surrounded by two guard pages.

The total memory dedicated to the KFENCE memory pool can be computed as:

```
( #objects + 1 ) * 2 * PAGE_SIZE
```

Using the default config, and assuming a page size of 4 KiB, results in dedicating 2 MiB to the KFENCE memory pool.

Note: On architectures that support huge pages, KFENCE will ensure that the pool is using pages of size `PAGE_SIZE`. This will result in additional page tables being allocated.

11.1.2 Error reports

A typical out-of-bounds access looks like this:

```
==================================================================
BUG: KFENCE: out-of-bounds read in test_out_of_bounds_read+0xa6/0x234

Out-of-bounds read at 0xffff8c3f2e291fff (1B left of kfence-#72):
 test_out_of_bounds_read+0xa6/0x234
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#72: 0xffff8c3f2e292000-0xffff8c3f2e29201f, size=32, cache=kmalloc-32

allocated by task 484 on cpu 0 at 32.919330s:
 test_alloc+0xfe/0x738
 test_out_of_bounds_read+0x9b/0x234
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 0 PID: 484 Comm: kunit_try_catch Not tainted 5.13.0-rc3+ #7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

The header of the report provides a short summary of the function involved in the access. It is followed by more detailed information about the access and its origin. Note that, real kernel addresses are only shown when using the kernel command line option `no_hash_pointers`.

Use-after-free accesses are reported as:

```
===================================================================
BUG: KFENCE: use-after-free read in test_use_after_free_read+0xb3/0x143

Use-after-free read at 0xffff8c3f2e2a0000 (in kfence-#79):
 test_use_after_free_read+0xb3/0x143
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#79: 0xffff8c3f2e2a0000-0xffff8c3f2e2a001f, size=32, cache=kmalloc-32

allocated by task 488 on cpu 2 at 33.871326s:
 test_alloc+0xfe/0x738
 test_use_after_free_read+0x76/0x143
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

freed by task 488 on cpu 2 at 33.871358s:
 test_use_after_free_read+0xa8/0x143
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 2 PID: 488 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc3+
 ↪#7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
===================================================================
```

KFENCE also reports on invalid frees, such as double-frees:

```
===================================================================
BUG: KFENCE: invalid free in test_double_free+0xdc/0x171

Invalid free of 0xffff8c3f2e2a4000 (in kfence-#81):
 test_double_free+0xdc/0x171
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#81: 0xffff8c3f2e2a4000-0xffff8c3f2e2a401f, size=32, cache=kmalloc-32

allocated by task 490 on cpu 1 at 34.175321s:
 test_alloc+0xfe/0x738
 test_double_free+0x76/0x171
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

freed by task 490 on cpu 1 at 34.175348s:
 test_double_free+0xa8/0x171
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 1 PID: 490 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc3+
 ↪#7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

KFENCE also uses pattern-based redzones on the other side of an object's guard page, to detect out-of-bounds writes on the unprotected side of the object. These are reported on frees:

```
==================================================================
BUG: KFENCE: memory corruption in test_kmalloc_aligned_oob_write+0xef/0x184

Corrupted memory at 0xffff8c3f2e33aff9 [ 0xac . . . . . . ] (in kfence-#156):
 test_kmalloc_aligned_oob_write+0xef/0x184
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#156: 0xffff8c3f2e33afb0-0xffff8c3f2e33aff8, size=73, cache=kmalloc-96

allocated by task 502 on cpu 7 at 42.159302s:
 test_alloc+0xfe/0x738
 test_kmalloc_aligned_oob_write+0x57/0x184
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 7 PID: 502 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc3+
 ↪#7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

For such errors, the address where the corruption occurred as well as the invalidly written bytes (offset from the address) are shown; in this representation, '.' denote untouched bytes. In the example above `0xac` is the value written to the invalid address at offset 0, and the remaining '.' denote that no following bytes have been touched. Note that, real values are only shown if the kernel was booted with `no_hash_pointers`; to avoid information disclosure otherwise, '!' is used instead to denote invalidly written bytes.

And finally, KFENCE may also report on invalid accesses to any protected page where it was not possible to determine an associated object, e.g. if adjacent object pages had not yet been allocated:

```
==================================================================
BUG: KFENCE: invalid read in test_invalid_access+0x26/0xe0

Invalid read at 0xffffffffb670b00a:
 test_invalid_access+0x26/0xe0
 kunit_try_run_case+0x51/0x85
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x137/0x160
 ret_from_fork+0x22/0x30

CPU: 4 PID: 124 Comm: kunit_try_catch Tainted: G        W         5.8.0-rc6+ #7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.13.0-1 04/01/2014
==================================================================
```

11.1.3 DebugFS interface

Some debugging information is exposed via debugfs:

- The file `/sys/kernel/debug/kfence/stats` provides runtime statistics.
- The file `/sys/kernel/debug/kfence/objects` provides a list of objects allocated via KFENCE, including those already freed but protected.

11.2 Implementation Details

Guarded allocations are set up based on the sample interval. After expiration of the sample interval, the next allocation through the main allocator (SLAB or SLUB) returns a guarded allocation from the KFENCE object pool (allocation sizes up to PAGE_SIZE are supported). At this point, the timer is reset, and the next allocation is set up after the expiration of the interval.

When using `CONFIG_KFENCE_STATIC_KEYS=y`, KFENCE allocations are "gated" through the main allocator's fast-path by relying on static branches via the static keys infrastructure. The static branch is toggled to redirect the allocation to KFENCE. Depending on sample interval, target workloads, and system architecture, this may perform better than the simple dynamic branch. Careful benchmarking is recommended.

KFENCE objects each reside on a dedicated page, at either the left or right page boundaries selected at random. The pages to the left and right of the object page are "guard pages", whose attributes are changed to a protected state, and cause page faults on any attempted access. Such page faults are then intercepted by KFENCE, which handles the fault gracefully by reporting an out-of-bounds access, and marking the page as accessible so that the faulting code can (wrongly) continue executing (set `panic_on_warn` to panic instead).

To detect out-of-bounds writes to memory within the object's page itself, KFENCE also uses pattern-based redzones. For each object page, a redzone is set up for all non-object memory. For typical alignments, the redzone is only required on the unguarded side of an object. Because KFENCE must honor the cache's requested alignment, special alignments may result in unprotected gaps on either side of an object, all of which are redzoned.

The following figure illustrates the page layout:

```
---+-----------+-----------+-----------+-----------+-----------+---
   | xxxxxxxxx | O :       | xxxxxxxxx |       : O | xxxxxxxxx |
   | xxxxxxxxx | B :       | xxxxxxxxx |       : B | xxxxxxxxx |
   | x GUARD x | J : RED-  | x GUARD x | RED-  : J | x GUARD x |
   | xxxxxxxxx | E :  ZONE | xxxxxxxxx |  ZONE : E | xxxxxxxxx |
   | xxxxxxxxx | C :       | xxxxxxxxx |       : C | xxxxxxxxx |
   | xxxxxxxxx | T :       | xxxxxxxxx |       : T | xxxxxxxxx |
---+-----------+-----------+-----------+-----------+-----------+---
```

Upon deallocation of a KFENCE object, the object's page is again protected and the object is marked as freed. Any further access to the object causes a fault and KFENCE reports a use-after-free access. Freed objects are inserted at the tail of KFENCE's freelist, so that the least recently freed objects are reused first, and the chances of detecting use-after-frees of recently freed objects is increased.

If pool utilization reaches 75% (default) or above, to reduce the risk of the pool eventually being fully occupied by allocated objects yet ensure diverse coverage of allocations, KFENCE limits currently covered allocations of the same source from further filling up the pool. The "source" of an allocation is based on its partial allocation stack trace. A side-effect is that this also limits frequent long-lived allocations (e.g. pagecache) of the same source filling up the pool permanently, which is the most common risk for the pool becoming full and the sampled allocation rate dropping to zero. The threshold at which to start limiting currently covered allocations can be configured via the boot parameter `kfence.skip_covered_thresh` (pool usage%).

11.3 Interface

The following describes the functions which are used by allocators as well as page handling code to set up and deal with KFENCE allocations.

bool **is_kfence_address**(const void *addr)
    check if an address belongs to KFENCE pool

Parameters

const void *addr address to check

Return

true or false depending on whether the address is within the KFENCE object range.

Description

KFENCE objects live in a separate page range and are not to be intermixed with regular heap objects (e.g. KFENCE objects must never be added to the allocator freelists). Failing to do so may and will result in heap corruptions, therefore *is_kfence_address()* must be used to check whether an object requires specific handling.

Note

This function may be used in fast-paths, and is performance critical. Future changes should take this into account; for instance, we want to avoid introducing another load and therefore need to keep KFENCE_POOL_SIZE a constant (until immediate patching support is added to the kernel).

void **kfence_shutdown_cache**(struct kmem_cache *s)
handle shutdown_cache() for KFENCE objects

Parameters

struct kmem_cache *s cache being shut down

Description

Before shutting down a cache, one must ensure there are no remaining objects allocated from it. Because KFENCE objects are not referenced from the cache directly, we need to check them here.

Note that shutdown_cache() is internal to SL*B, and kmem_cache_destroy() does not return if allocated objects still exist: it prints an error message and simply aborts destruction of a cache, leaking memory.

If the only such objects are KFENCE objects, we will not leak the entire cache, but instead try to provide more useful debug info by making allocated objects "zombie allocations". Objects may then still be used or freed (which is handled gracefully), but usage will result in showing KFENCE error reports which include stack traces to the user of the object, the original allocation site, and caller to shutdown_cache().

void ***kfence_alloc**(struct kmem_cache *s, size_t size, gfp_t flags)
allocate a KFENCE object with a low probability

Parameters

struct kmem_cache *s struct kmem_cache with object requirements

size_t size exact size of the object to allocate (can be less than **s->size** e.g. for kmalloc caches)

gfp_t flags GFP flags

Return

- NULL - must proceed with allocating as usual,
- non-NULL - pointer to a KFENCE object.

Description

kfence_alloc() should be inserted into the heap allocation fast path, allowing it to transparently return KFENCE-allocated objects with a low probability using a static branch (the probability is controlled by the kfence.sample_interval boot parameter).

size_t **kfence_ksize**(const void *addr)
get actual amount of memory allocated for a KFENCE object

Parameters

const void *addr pointer to a heap object

Return

- 0 - not a KFENCE object, must call __ksize() instead,
- non-0 - this many bytes can be accessed without causing a memory error.

Description

kfence_ksize() returns the number of bytes requested for a KFENCE object at allocation time. This number may be less than the object size of the corresponding struct kmem_cache.

void ***kfence_object_start**(const void *addr)

find the beginning of a KFENCE object

Parameters

const void *addr address within a KFENCE-allocated object

Return

address of the beginning of the object.

Description

SL[AU]B-allocated objects are laid out within a page one by one, so it is easy to calculate the beginning of an object given a pointer inside it and the object size. The same is not true for KFENCE, which places a single object at either end of the page. This helper function is used to find the beginning of a KFENCE-allocated object.

void **__kfence_free**(void *addr)

release a KFENCE heap object to KFENCE pool

Parameters

void *addr object to be freed

Description

Requires: is_kfence_address(addr)

Release a KFENCE object and mark it as freed.

bool **kfence_free**(void *addr)

try to release an arbitrary heap object to KFENCE pool

Parameters

void *addr object to be freed

Return

- false - object doesn’t belong to KFENCE pool and was ignored,
- true - object was released to KFENCE pool.

Description

Release a KFENCE object and mark it as freed. May be called on any object, even non-KFENCE objects, to simplify integration of the hooks into the allocator’s free codepath. The allocator must check the return value to determine if it was a KFENCE object or not.

bool **kfence_handle_page_fault**(unsigned long addr, bool is_write, struct pt_regs *regs)

perform page fault handling for KFENCE pages

Parameters

unsigned long addr faulting address

bool is_write is access a write

struct pt_regs *regs current struct pt_regs (can be NULL, but shows full stack trace)

Return

- false - address outside KFENCE pool,
- true - page fault handled by KFENCE, no additional handling required.

Description

A page fault inside KFENCE pool indicates a memory error, such as an out-of-bounds access, a use-after-free or an invalid memory access. In these cases KFENCE prints an error message and marks the offending page as present, so that the kernel can proceed.

11.4 Related Tools

In userspace, a similar approach is taken by GWP-ASan. GWP-ASan also relies on guard pages and a sampling strategy to detect memory unsafety bugs at scale. KFENCE's design is directly influenced by GWP-ASan, and can be seen as its kernel sibling. Another similar but non-sampling approach, that also inspired the name "KFENCE", can be found in the userspace Electric Fence Malloc Debugger.

In the kernel, several tools exist to debug memory access errors, and in particular KASAN can detect all bug classes that KFENCE can detect. While KASAN is more precise, relying on compiler instrumentation, this comes at a performance cost.

It is worth highlighting that KASAN and KFENCE are complementary, with different target environments. For instance, KASAN is the better debugging-aid, where test cases or reproducers exists: due to the lower chance to detect the error, it would require more effort using KFENCE to debug. Deployments at scale that cannot afford to enable KASAN, however, would benefit from using KFENCE to discover bugs due to code paths not exercised by test cases or fuzzers.

CHAPTER

TWELVE

DEBUGGING KERNEL AND MODULES VIA GDB

The kernel debugger kgdb, hypervisors like QEMU or JTAG-based hardware interfaces allow to debug the Linux kernel and its modules during runtime using gdb. Gdb comes with a powerful scripting interface for python. The kernel provides a collection of helper scripts that can simplify typical kernel debugging steps. This is a short tutorial about how to enable and use them. It focuses on QEMU/KVM virtual machines as target, but the examples can be transferred to the other gdb stubs as well.

12.1 Requirements

- gdb 7.2+ (recommended: 7.4+) with python support enabled (typically true for distributions)

12.2 Setup

- Create a virtual Linux machine for QEMU/KVM (see www.linux-kvm.org and www.qemu.org for more details). For cross-development, https://landley.net/aboriginal/bin keeps a pool of machine images and toolchains that can be helpful to start from.
- Build the kernel with CONFIG_GDB_SCRIPTS enabled, but leave CONFIG_DEBUG_INFO_REDUCED off. If your architecture supports CONFIG_FRAME_POINTER, keep it enabled.
- Install that kernel on the guest, turn off KASLR if necessary by adding “nokaslr” to the kernel command line. Alternatively, QEMU allows to boot the kernel directly using -kernel, -append, -initrd command line switches. This is generally only useful if you do not depend on modules. See QEMU documentation for more details on this mode. In this case, you should build the kernel with CONFIG_RANDOMIZE_BASE disabled if the architecture supports KASLR.
- Enable the gdb stub of QEMU/KVM, either
  - at VM startup time by appending “-s” to the QEMU command line

  or

  - during runtime by issuing “gdbserver” from the QEMU monitor console
- cd /path/to/linux-build

- Start gdb: gdb vmlinux

  Note: Some distros may restrict auto-loading of gdb scripts to known safe directories. In case gdb reports to refuse loading vmlinux-gdb.py, add:

  ```
  add-auto-load-safe-path /path/to/linux-build
  ```

  to ~/.gdbinit. See gdb help for more details.

- Attach to the booted guest:

  ```
  (gdb) target remote :1234
  ```

12.3 Examples of using the Linux-provided gdb helpers

- Load module (and main kernel) symbols:

  ```
  (gdb) lx-symbols
  loading vmlinux
  scanning for modules in /home/user/linux/build
  loading @0xffffffffa0020000: /home/user/linux/build/net/netfilter/xt_
  ↪tcpudp.ko
  loading @0xffffffffa0016000: /home/user/linux/build/net/netfilter/xt_
  ↪pkttype.ko
  loading @0xffffffffa0002000: /home/user/linux/build/net/netfilter/xt_limit.
  ↪ko
  loading @0xffffffffa00ca000: /home/user/linux/build/net/packet/af_packet.ko
  loading @0xffffffffa003c000: /home/user/linux/build/fs/fuse/fuse.ko
  ...
  loading @0xffffffffa0000000: /home/user/linux/build/drivers/ata/ata_
  ↪generic.ko
  ```

- Set a breakpoint on some not yet loaded module function, e.g.:

  ```
  (gdb) b btrfs_init_sysfs
  Function "btrfs_init_sysfs" not defined.
  Make breakpoint pending on future shared library load? (y or [n]) y
  Breakpoint 1 (btrfs_init_sysfs) pending.
  ```

- Continue the target:

  ```
  (gdb) c
  ```

- Load the module on the target and watch the symbols being loaded as well as the breakpoint hit:

  ```
  loading @0xffffffffa0034000: /home/user/linux/build/lib/libcrc32c.ko
  loading @0xffffffffa0050000: /home/user/linux/build/lib/lzo/lzo_compress.ko
  loading @0xffffffffa006e000: /home/user/linux/build/lib/zlib_deflate/zlib_
  ↪deflate.ko
  loading @0xffffffffa01b1000: /home/user/linux/build/fs/btrfs/btrfs.ko

  ```

```
Breakpoint 1, btrfs_init_sysfs () at /home/user/linux/fs/btrfs/sysfs.c:36
36              btrfs_kset = kset_create_and_add("btrfs", NULL, fs_kobj);
```

- Dump the log buffer of the target kernel:

```
(gdb) lx-dmesg
[     0.000000] Initializing cgroup subsys cpuset
[     0.000000] Initializing cgroup subsys cpu
[     0.000000] Linux version 3.8.0-rc4-dbg+ (...
[     0.000000] Command line: root=/dev/sda2 resume=/dev/sda1 vga=0x314
[     0.000000] e820: BIOS-provided physical RAM map:
[     0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff]␣
↪usable
[     0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff]␣
↪reserved
....
```

- Examine fields of the current task struct(supported by x86 and arm64 only):

```
(gdb) p $lx_current().pid
$1 = 4998
(gdb) p $lx_current().comm
$2 = "modprobe\000\000\000\000\000\000\000"
```

- Make use of the per-cpu function for the current or a specified CPU:

```
(gdb) p $lx_per_cpu("runqueues").nr_running
$3 = 1
(gdb) p $lx_per_cpu("runqueues", 2).nr_running
$4 = 0
```

- Dig into hrtimers using the container_of helper:

```
(gdb) set $next = $lx_per_cpu("hrtimer_bases").clock_base[0].active.next
(gdb) p *$container_of($next, "struct hrtimer", "node")
$5 = {
  node = {
    node = {
      __rb_parent_color = 18446612133355256072,
      rb_right = 0x0 <irq_stack_union>,
      rb_left = 0x0 <irq_stack_union>
    },
    expires = {
      tv64 = 1835268000000
    }
  },
  _softexpires = {
    tv64 = 1835268000000
  },
  function = 0xffffffff81078232 <tick_sched_timer>,
  base = 0xffff88003fd0d6f0,
  state = 1,
  start_pid = 0,
  start_site = 0xffffffff81055c1f <hrtimer_start_range_ns+20>,
  start_comm = "swapper/2\000\000\000\000\000\000"
}
```

12.4 List of commands and functions

The number of commands and convenience functions may evolve over the time, this is just a snapshot of the initial version:

```
(gdb) apropos lx
function lx_current -- Return current task
function lx_module -- Find module by name and return the module variable
function lx_per_cpu -- Return per-cpu variable
function lx_task_by_pid -- Find Linux task by PID and return the task_struct␣
↪variable
function lx_thread_info -- Calculate Linux thread_info from task variable
lx-dmesg -- Print Linux kernel log buffer
lx-lsmod -- List currently loaded modules
lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded modules
```

Detailed help can be obtained via “help <command-name>” for commands and “help function <function-name>” for convenience functions.

CHAPTER

THIRTEEN

USING KGDB, KDB AND THE KERNEL DEBUGGER INTERNALS

Author Jason Wessel

13.1 Introduction

The kernel has two different debugger front ends (kdb and kgdb) which interface to the debug core. It is possible to use either of the debugger front ends and dynamically transition between them if you configure the kernel properly at compile and runtime.

Kdb is simplistic shell-style interface which you can use on a system console with a keyboard or serial console. You can use it to inspect memory, registers, process lists, dmesg, and even set breakpoints to stop in a certain location. Kdb is not a source level debugger, although you can set breakpoints and execute some basic kernel run control. Kdb is mainly aimed at doing some analysis to aid in development or diagnosing kernel problems. You can access some symbols by name in kernel built-ins or in kernel modules if the code was built with `CONFIG_KALLSYMS`.

Kgdb is intended to be used as a source level debugger for the Linux kernel. It is used along with gdb to debug a Linux kernel. The expectation is that gdb can be used to "break in" to the kernel to inspect memory, variables and look through call stack information similar to the way an application developer would use gdb to debug an application. It is possible to place breakpoints in kernel code and perform some limited execution stepping.

Two machines are required for using kgdb. One of these machines is a development machine and the other is the target machine. The kernel to be debugged runs on the target machine. The development machine runs an instance of gdb against the vmlinux file which contains the symbols (not a boot image such as bzImage, zImage, uImage…). In gdb the developer specifies the connection parameters and connects to kgdb. The type of connection a developer makes with gdb depends on the availability of kgdb I/O modules compiled as built-ins or loadable kernel modules in the test machine's kernel.

13.2 Compiling a kernel

- In order to enable compilation of kdb, you must first enable kgdb.
- The kgdb test compile options are described in the kgdb test suite chapter.

13.2.1 Kernel config options for kgdb

To enable `CONFIG_KGDB` you should look under *Kernel hacking* → *Kernel debugging* and select *KGDB: kernel debugger*.

While it is not a hard requirement that you have symbols in your vmlinux file, gdb tends not to be very useful without the symbolic data, so you will want to turn on `CONFIG_DEBUG_INFO` which is called *Compile the kernel with debug info* in the config menu.

It is advised, but not required, that you turn on the `CONFIG_FRAME_POINTER` kernel option which is called *Compile the kernel with frame pointers* in the config menu. This option inserts code into the compiled executable which saves the frame information in registers or on the stack at different points which allows a debugger such as gdb to more accurately construct stack back traces while debugging the kernel.

If the architecture that you are using supports the kernel option `CONFIG_STRICT_KERNEL_RWX`, you should consider turning it off. This option will prevent the use of software breakpoints because it marks certain regions of the kernel's memory space as read-only. If kgdb supports it for the architecture you are using, you can use hardware breakpoints if you desire to run with the `CONFIG_STRICT_KERNEL_RWX` option turned on, else you need to turn off this option.

Next you should choose one of more I/O drivers to interconnect debugging host and debugged target. Early boot debugging requires a KGDB I/O driver that supports early debugging and the driver must be built into the kernel directly. Kgdb I/O driver configuration takes place via kernel or module parameters which you can learn more about in the in the section that describes the parameter kgdboc.

Here is an example set of `.config` symbols to enable or disable for kgdb:

```
# CONFIG_STRICT_KERNEL_RWX is not set
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
```

13.2.2 Kernel config options for kdb

Kdb is quite a bit more complex than the simple gdbstub sitting on top of the kernel's debug core. Kdb must implement a shell, and also adds some helper functions in other parts of the kernel, responsible for printing out interesting data such as what you would see if you ran `lsmod`, or `ps`. In order to build kdb into the kernel you follow the same steps as you would for kgdb.

The main config option for kdb is `CONFIG_KGDB_KDB` which is called *KGDB_KDB: include kdb frontend for kgdb* in the config menu. In theory you would have already also selected an I/O driver such as the `CONFIG_KGDB_SERIAL_CONSOLE` interface if you plan on using kdb on a serial port, when you were configuring kgdb.

If you want to use a PS/2-style keyboard with kdb, you would select `CONFIG_KDB_KEYBOARD` which is called *KGDB_KDB: keyboard as input device* in the config menu. The `CONFIG_KDB_KEYBOARD` option is not used for anything in the gdb interface to kgdb. The `CONFIG_KDB_KEYBOARD` option only works with kdb.

Here is an example set of `.config` symbols to enable/disable kdb:

```
# CONFIG_STRICT_KERNEL_RWX is not set
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
CONFIG_KGDB_KDB=y
CONFIG_KDB_KEYBOARD=y
```

13.3 Kernel Debugger Boot Arguments

This section describes the various runtime kernel parameters that affect the configuration of the kernel debugger. The following chapter covers using kdb and kgdb as well as providing some examples of the configuration parameters.

13.3.1 Kernel parameter: kgdboc

The kgdboc driver was originally an abbreviation meant to stand for “kgdb over console”. Today it is the primary mechanism to configure how to communicate from gdb to kgdb as well as the devices you want to use to interact with the kdb shell.

For kgdb/gdb, kgdboc is designed to work with a single serial port. It is intended to cover the circumstance where you want to use a serial console as your primary console as well as using it to perform kernel debugging. It is also possible to use kgdb on a serial port which is not designated as a system console. Kgdboc may be configured as a kernel built-in or a kernel loadable module. You can only make use of `kgdbwait` and early debugging if you build kgdboc into the kernel as a built-in.

Optionally you can elect to activate kms (Kernel Mode Setting) integration. When you use kms with kgdboc and you have a video driver that has atomic mode setting hooks, it is possible to enter the debugger on the graphics console. When the kernel execution is resumed, the previous graphics mode will be restored. This integration can serve as a useful tool to aid in diagnosing crashes or doing analysis of memory with kdb while allowing the full graphics console applications to run.

kgdboc arguments

Usage:

```
kgdboc=[kms][[,]kbd][[,]serial_device][,baud]
```

The order listed above must be observed if you use any of the optional configurations together.

Abbreviations:

- kms = Kernel Mode Setting
- kbd = Keyboard

You can configure kgdboc to use the keyboard, and/or a serial device depending on if you are using kdb and/or kgdb, in one of the following scenarios. The order listed above must be observed if you use any of the optional configurations together. Using kms + only gdb is generally not a useful combination.

Using loadable module or built-in

1. As a kernel built-in:

   Use the kernel boot argument:

   ```
   kgdboc=<tty-device>,[baud]
   ```

2. As a kernel loadable module:

   Use the command:

   ```
   modprobe kgdboc kgdboc=<tty-device>,[baud]
   ```

   Here are two examples of how you might format the kgdboc string. The first is for an x86 target using the first serial port. The second example is for the ARM Versatile AB using the second serial port.

   1. `kgdboc=ttyS0,115200`
   2. `kgdboc=ttyAMA1,115200`

Configure kgdboc at runtime with sysfs

At run time you can enable or disable kgdboc by echoing a parameters into the sysfs. Here are two examples:

1. Enable kgdboc on ttyS0:

   ```
   echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
   ```

2. Disable kgdboc:

   ```
   echo "" > /sys/module/kgdboc/parameters/kgdboc
   ```

Note: You do not need to specify the baud if you are configuring the console on tty which is already configured or open.

More examples

You can configure kgdboc to use the keyboard, and/or a serial device depending on if you are using kdb and/or kgdb, in one of the following scenarios.

1. kdb and kgdb over only a serial port:

   ```
   kgdboc=<serial_device>[,baud]
   ```

   Example:

   ```
   kgdboc=ttyS0,115200
   ```

2. kdb and kgdb with keyboard and a serial port:

```
kgdboc=kbd,<serial_device>[,baud]
```

Example:

```
kgdboc=kbd,ttyS0,115200
```

3. kdb with a keyboard:

```
kgdboc=kbd
```

4. kdb with kernel mode setting:

```
kgdboc=kms,kbd
```

5. kdb with kernel mode setting and kgdb over a serial port:

```
kgdboc=kms,kbd,ttyS0,115200
```

Note: Kgdboc does not support interrupting the target via the gdb remote protocol. You must manually send a SysRq-G unless you have a proxy that splits console output to a terminal program. A console proxy has a separate TCP port for the debugger and a separate TCP port for the "human" console. The proxy can take care of sending the SysRq-G for you.

When using kgdboc with no debugger proxy, you can end up connecting the debugger at one of two entry points. If an exception occurs after you have loaded kgdboc, a message should print on the console stating it is waiting for the debugger. In this case you disconnect your terminal program and then connect the debugger in its place. If you want to interrupt the target system and forcibly enter a debug session you have to issue a Sysrq sequence and then type the letter g. Then you disconnect the terminal session and connect gdb. Your options if you don't like this are to hack gdb to send the SysRq-G for you as well as on the initial connect, or to use a debugger proxy that allows an unmodified gdb to do the debugging.

13.3.2 Kernel parameter: kgdboc_earlycon

If you specify the kernel parameter kgdboc_earlycon and your serial driver registers a boot console that supports polling (doesn't need interrupts and implements a nonblocking read() function) kgdb will attempt to work using the boot console until it can transition to the regular tty driver specified by the kgdboc parameter.

Normally there is only one boot console (especially that implements the read() function) so just adding kgdboc_earlycon on its own is sufficient to make this work. If you have more than one boot console you can add the boot console's name to differentiate. Note that names that are registered through the boot console layer and the tty layer are not the same for the same port.

For instance, on one board to be explicit you might do:

```
kgdboc_earlycon=qcom_geni kgdboc=ttyMSM0
```

If the only boot console on the device was "qcom_geni", you could simplify:

```
kgdboc_earlycon kgdboc=ttyMSM0
```

13.3.3 Kernel parameter: kgdbwait

The Kernel command line option kgdbwait makes kgdb wait for a debugger connection during booting of a kernel. You can only use this option if you compiled a kgdb I/O driver into the kernel and you specified the I/O driver configuration as a kernel command line option. The kgdbwait parameter should always follow the configuration parameter for the kgdb I/O driver in the kernel command line else the I/O driver will not be configured prior to asking the kernel to use it to wait.

The kernel will stop and wait as early as the I/O driver and architecture allows when you use this option. If you build the kgdb I/O driver as a loadable kernel module kgdbwait will not do anything.

13.3.4 Kernel parameter: kgdbcon

The kgdbcon feature allows you to see printk() messages inside gdb while gdb is connected to the kernel. Kdb does not make use of the kgdbcon feature.

Kgdb supports using the gdb serial protocol to send console messages to the debugger when the debugger is connected and running. There are two ways to activate this feature.

1. Activate with the kernel command line option:

   ```
   kgdbcon
   ```

2. Use sysfs before configuring an I/O driver:

   ```
   echo 1 > /sys/module/kgdb/parameters/kgdb_use_con
   ```

Note: If you do this after you configure the kgdb I/O driver, the setting will not take effect until the next point the I/O is reconfigured.

Important: You cannot use kgdboc + kgdbcon on a tty that is an active system console. An example of incorrect usage is:

```
console=ttyS0,115200 kgdboc=ttyS0 kgdbcon
```

It is possible to use this option with kgdboc on a tty that is not a system console.

13.3.5 Run time parameter: kgdbreboot

The kgdbreboot feature allows you to change how the debugger deals with the reboot notification. You have 3 choices for the behavior. The default behavior is always set to 0.

1	`echo -1 > /sys/module/debug_core/parameters/kgdbreboot`	Ignore the reboot notification entirely.
2	`echo 0 > /sys/module/debug_core/parameters/kgdbreboot`	Send the detach message to any attached debugger client.
3	`echo 1 > /sys/module/debug_core/parameters/kgdbreboot`	Enter the debugger on reboot notify.

13.3.6 Kernel parameter: nokaslr

If the architecture that you are using enable KASLR by default, you should consider turning it off. KASLR randomizes the virtual address where the kernel image is mapped and confuse gdb which resolve kernel symbol address from symbol table of vmlinux.

13.4 Using kdb

13.4.1 Quick start for kdb on a serial port

This is a quick example of how to use kdb.

1. Configure kgdboc at boot using kernel parameters:

   ```
   console=ttyS0,115200 kgdboc=ttyS0,115200 nokaslr
   ```

   OR

   Configure kgdboc after the kernel has booted; assuming you are using a serial port console:

   ```
   echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
   ```

2. Enter the kernel debugger manually or by waiting for an oops or fault. There are several ways you can enter the kernel debugger manually; all involve using the `SysRq-G`, which means you must have enabled `CONFIG_MAGIC_SYSRQ=y` in your kernel config.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using minicom 2.2

     Press: `CTRL-A f g`

   - When you have telneted to a terminal server that supports sending a remote break

     Press: `CTRL-]`

     Type in: `send break`

Press: `Enter g`

3. From the kdb prompt you can run the `help` command to see a complete list of the commands that are available.

   Some useful commands in kdb include:

`lsmod`	Shows where kernel modules are loaded
`ps`	Displays only the active processes
`ps A`	Shows all the processes
`summary`	Shows kernel version info and memory usage
`bt`	Get a backtrace of the current process using dump_stack()
`dmesg`	View the kernel syslog buffer
`go`	Continue the system

4. When you are done using kdb you need to consider rebooting the system or using the `go` command to resuming normal kernel execution. If you have paused the kernel for a lengthy period of time, applications that rely on timely networking or anything to do with real wall clock time could be adversely affected, so you should take this into consideration when using the kernel debugger.

13.4.2 Quick start for kdb using a keyboard connected console

This is a quick example of how to use kdb with a keyboard.

1. Configure kgdboc at boot using kernel parameters:

   ```
   kgdboc=kbd
   ```

   OR

   Configure kgdboc after the kernel has booted:

   ```
   echo kbd > /sys/module/kgdboc/parameters/kgdboc
   ```

2. Enter the kernel debugger manually or by waiting for an oops or fault. There are several ways you can enter the kernel debugger manually; all involve using the `SysRq-G`, which means you must have enabled `CONFIG_MAGIC_SYSRQ=y` in your kernel config.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using a laptop keyboard:

     Press and hold down: `Alt`

     Press and hold down: `Fn`

     Press and release the key with the label: `SysRq`

     Release: `Fn`

     Press and release: `g`

     Release: `Alt`

   - Example using a PS/2 101-key keyboard

     Press and hold down: `Alt`

     Press and release the key with the label: `SysRq`

     Press and release: `g`

     Release: `Alt`

3. Now type in a kdb command such as `help`, `dmesg`, `bt` or `go` to continue kernel execution.

13.5 Using kgdb / gdb

In order to use kgdb you must activate it by passing configuration information to one of the kgdb I/O drivers. If you do not pass any configuration information kgdb will not do anything at all. Kgdb will only actively hook up to the kernel trap hooks if a kgdb I/O driver is loaded and configured. If you unconfigure a kgdb I/O driver, kgdb will unregister all the kernel hook points.

All kgdb I/O drivers can be reconfigured at run time, if `CONFIG_SYSFS` and `CONFIG_MODULES` are enabled, by echo'ing a new config string to `/sys/module/<driver>/parameter/<option>`. The driver can be unconfigured by passing an empty string. You cannot change the configuration while the debugger is attached. Make sure to detach the debugger with the `detach` command prior to trying to unconfigure a kgdb I/O driver.

13.5.1 Connecting with gdb to a serial port

1. Configure kgdboc

   Configure kgdboc at boot using kernel parameters:

   ```
   kgdboc=ttyS0,115200
   ```

   OR

   Configure kgdboc after the kernel has booted:

   ```
   echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
   ```

2. Stop kernel execution (break into the debugger)

   In order to connect to gdb via kgdboc, the kernel must first be stopped. There are several ways to stop the kernel which include using kgdbwait as a boot argument, via a `SysRq-G`, or running the kernel until it takes an exception where it waits for the debugger to attach.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using minicom 2.2

     Press: `CTRL-A f g`

- When you have telneted to a terminal server that supports sending a remote break

  Press: `CTRL-]`

  Type in: `send break`

  Press: `Enter` `g`

3. Connect from gdb

   Example (using a directly connected port):

   ```
   % gdb ./vmlinux
   (gdb) set serial baud 115200
   (gdb) target remote /dev/ttyS0
   ```

   Example (kgdb to a terminal server on TCP port 2012):

   ```
   % gdb ./vmlinux
   (gdb) target remote 192.168.2.2:2012
   ```

   Once connected, you can debug a kernel the way you would debug an application program.

   If you are having problems connecting or something is going seriously wrong while debugging, it will most often be the case that you want to enable gdb to be verbose about its target communications. You do this prior to issuing the `target remote` command by typing in:

   ```
   set debug remote 1
   ```

Remember if you continue in gdb, and need to "break in" again, you need to issue an other `SysRq-G`. It is easy to create a simple entry point by putting a breakpoint at `sys_sync` and then you can run `sync` from a shell or script to break into the debugger.

13.6 kgdb and kdb interoperability

It is possible to transition between kdb and kgdb dynamically. The debug core will remember which you used the last time and automatically start in the same mode.

13.6.1 Switching between kdb and kgdb

Switching from kgdb to kdb

There are two ways to switch from kgdb to kdb: you can use gdb to issue a maintenance packet, or you can blindly type the command `$3#33`. Whenever the kernel debugger stops in kgdb mode it will print the message `KGDB or $3#33 for KDB`. It is important to note that you have to type the sequence correctly in one pass. You cannot type a backspace or delete because kgdb will interpret that as part of the debug stream.

1. Change from kgdb to kdb by blindly typing:

   ```
   $3#33
   ```

2. Change from kgdb to kdb with gdb:

```
maintenance packet 3
```

Note: Now you must kill gdb. Typically you press CTRL-Z and issue the command:

```
kill -9 %
```

Change from kdb to kgdb

There are two ways you can change from kdb to kgdb. You can manually enter kgdb mode by issuing the kgdb command from the kdb shell prompt, or you can connect gdb while the kdb shell prompt is active. The kdb shell looks for the typical first commands that gdb would issue with the gdb remote protocol and if it sees one of those commands it automatically changes into kgdb mode.

1. From kdb issue the command:

   ```
   kgdb
   ```

   Now disconnect your terminal program and connect gdb in its place

2. At the kdb prompt, disconnect the terminal program and connect gdb in its place.

13.6.2 Running kdb commands from gdb

It is possible to run a limited set of kdb commands from gdb, using the gdb monitor command. You don't want to execute any of the run control or breakpoint operations, because it can disrupt the state of the kernel debugger. You should be using gdb for breakpoints and run control operations if you have gdb connected. The more useful commands to run are things like lsmod, dmesg, ps or possibly some of the memory information commands. To see all the kdb commands you can run `monitor help`.

Example:

```
(gdb) monitor ps
1 idle process (state I) and
27 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Task Addr       Pid   Parent [*] cpu State Thread     Command

0xc78291d0        1        0  0    0   S  0xc7829404  init
0xc7954150      942        1  0    0   S  0xc7954384  dropbear
0xc78789c0      944        1  0    0   S  0xc7878bf4  sh
(gdb)
```

13.7 kgdb Test Suite

When kgdb is enabled in the kernel config you can also elect to enable the config parameter `KGDB_TESTS`. Turning this on will enable a special kgdb I/O module which is designed to test the kgdb internal functions.

The kgdb tests are mainly intended for developers to test the kgdb internals as well as a tool for developing a new kgdb architecture specific implementation. These tests are not really for end users of the Linux kernel. The primary source of documentation would be to look in the `drivers/misc/kgdbts.c` file.

The kgdb test suite can also be configured at compile time to run the core set of tests by setting the kernel config parameter `KGDB_TESTS_ON_BOOT`. This particular option is aimed at automated regression testing and does not require modifying the kernel boot config arguments. If this is turned on, the kgdb test suite can be disabled by specifying `kgdbts=` as a kernel boot argument.

13.8 Kernel Debugger Internals

13.8.1 Architecture Specifics

The kernel debugger is organized into a number of components:

1. The debug core

   The debug core is found in `kernel/debugger/debug_core.c`. It contains:

   - A generic OS exception handler which includes sync'ing the processors into a stopped state on an multi-CPU system.
   - The API to talk to the kgdb I/O drivers
   - The API to make calls to the arch-specific kgdb implementation
   - The logic to perform safe memory reads and writes to memory while using the debugger
   - A full implementation for software breakpoints unless overridden by the arch
   - The API to invoke either the kdb or kgdb frontend to the debug core.
   - The structures and callback API for atomic kernel mode setting.

     Note: kgdboc is where the kms callbacks are invoked.

2. kgdb arch-specific implementation

   This implementation is generally found in `arch/*/kernel/kgdb.c`. As an example, `arch/x86/kernel/kgdb.c` contains the specifics to implement HW breakpoint as well as the initialization to dynamically register and unregister for the trap handlers on this architecture. The arch-specific portion implements:

   - contains an arch-specific trap catcher which invokes kgdb_handle_exception() to start kgdb about doing its work
   - translation to and from gdb specific packet format to struct pt_regs

   - Registration and unregistration of architecture specific trap hooks
   - Any special exception handling and cleanup
   - NMI exception handling and cleanup
   - (optional) HW breakpoints

3. gdbstub frontend (aka kgdb)

   The gdbstub is located in `kernel/debug/gdbstub.c`. It contains:

   - All the logic to implement the gdb serial protocol

4. kdb frontend

   The kdb debugger shell is broken down into a number of components. The kdb core is located in kernel/debug/kdb. There are a number of helper functions in some of the other kernel components to make it possible for kdb to examine and report information about the kernel without taking locks that could cause a kernel deadlock. The kdb core contains implements the following functionality.

   - A simple shell
   - The kdb core command set
   - A registration API to register additional kdb shell commands.
     - A good example of a self-contained kdb module is the `ftdump` command for dumping the ftrace buffer. See: `kernel/trace/trace_kdb.c`
     - For an example of how to dynamically register a new kdb command you can build the kdb_hello.ko kernel module from `samples/kdb/kdb_hello.c`. To build this example you can set `CONFIG_SAMPLES=y` and `CONFIG_SAMPLE_KDB=m` in your kernel config. Later run `modprobe kdb_hello` and the next time you enter the kdb shell, you can run the `hello` command.
   - The implementation for kdb_printf() which emits messages directly to I/O drivers, bypassing the kernel log.
   - SW / HW breakpoint management for the kdb shell

5. kgdb I/O driver

   Each kgdb I/O driver has to provide an implementation for the following:

   - configuration via built-in or module
   - dynamic configuration and kgdb hook registration calls
   - read and write character interface
   - A cleanup handler for unconfiguring from the kgdb core
   - (optional) Early debug methodology

   Any given kgdb I/O driver has to operate very closely with the hardware and must do it in such a way that does not enable interrupts or change other parts of the system context without completely restoring them. The kgdb core will repeatedly "poll" a kgdb I/O driver for characters when it needs input. The I/O driver is expected to return immediately if there is no data available. Doing so allows for the future possibility to touch watchdog hardware in such a way as to have a target system not reset when these are enabled.

If you are intent on adding kgdb architecture specific support for a new architecture, the architecture should define `HAVE_ARCH_KGDB` in the architecture specific Kconfig file. This will enable kgdb for the architecture, and at that point you must create an architecture specific kgdb implementation.

There are a few flags which must be set on every architecture in their `asm/kgdb.h` file. These are:

- **`NUMREGBYTES`:** The size in bytes of all of the registers, so that we can ensure they will all fit into a packet.
- **`BUFMAX`:** The size in bytes of the buffer GDB will read into. This must be larger than NUMREGBYTES.
- **`CACHE_FLUSH_IS_SAFE`:** Set to 1 if it is always safe to call flush_cache_range or flush_icache_range. On some architectures, these functions may not be safe to call on SMP since we keep other CPUs in a holding pattern.

There are also the following functions for the common backend, found in `kernel/kgdb.c`, that must be supplied by the architecture-specific backend unless marked as (optional), in which case a default function maybe used if the architecture does not need to provide a specific implementation.

int **`kgdb_skipexception`**(int exception, struct pt_regs *regs)
    (optional) exit kgdb_handle_exception early

Parameters

`int exception` Exception vector number

`struct pt_regs *regs` Current `struct pt_regs`.

On some architectures it is required to skip a breakpoint exception when it occurs after a breakpoint has been removed. This can be implemented in the architecture specific portion of kgdb.

void **`kgdb_breakpoint`**(void)
    compiled in breakpoint

Parameters

`void` no arguments

Description

This will be implemented as a static inline per architecture. This function is called by the kgdb core to execute an architecture specific trap to cause kgdb to enter the exception processing.

int **`kgdb_arch_init`**(void)
    Perform any architecture specific initialization.

Parameters

`void` no arguments

Description

This function will handle the initialization of any architecture specific callbacks.

void **`kgdb_arch_exit`**(void)
    Perform any architecture specific uninitalization.

Parameters

void no arguments

Description

This function will handle the uninitalization of any architecture specific callbacks, for dynamic registration and unregistration.

void **pt_regs_to_gdb_regs**(unsigned long *gdb_regs, struct pt_regs *regs)
Convert ptrace regs to GDB regs

Parameters

unsigned long *gdb_regs A pointer to hold the registers in the order GDB wants.

struct pt_regs *regs The `struct pt_regs` of the current process.

Convert the pt_regs in **regs** into the format for registers that GDB expects, stored in **gdb_regs**.

void **sleeping_thread_to_gdb_regs**(unsigned long *gdb_regs, struct task_struct *p)
Convert ptrace regs to GDB regs

Parameters

unsigned long *gdb_regs A pointer to hold the registers in the order GDB wants.

struct task_struct *p The `struct task_struct` of the desired process.

Convert the register values of the sleeping process in **p** to the format that GDB expects. This function is called when kgdb does not have access to the `struct pt_regs` and therefore it should fill the gdb registers **gdb_regs** with what has been saved in `struct thread_struct` thread field during switch_to.

void **gdb_regs_to_pt_regs**(unsigned long *gdb_regs, struct pt_regs *regs)
Convert GDB regs to ptrace regs.

Parameters

unsigned long *gdb_regs A pointer to hold the registers we've received from GDB.

struct pt_regs *regs A pointer to a `struct pt_regs` to hold these values in.

Convert the GDB regs in **gdb_regs** into the pt_regs, and store them in **regs**.

int **kgdb_arch_handle_exception**(int vector, int signo, int err_code, char *remcom_in_buffer, char *remcom_out_buffer, struct pt_regs *regs)
Handle architecture specific GDB packets.

Parameters

int vector The error vector of the exception that happened.

int signo The signal number of the exception that happened.

int err_code The error code of the exception that happened.

char *remcom_in_buffer The buffer of the packet we have read.

char *remcom_out_buffer The buffer of `BUFMAX` bytes to write a packet into.

struct pt_regs *regs The `struct pt_regs` of the current process.

This function MUST handle the 'c' and 's' command packets, as well packets to set / remove a hardware breakpoint, if used. If there are additional packets which the hardware needs to handle, they are handled here. The code should return -1 if it wants to process more packets, and a `0` or `1` if it wants to exit from the kgdb callback.

void **`kgdb_arch_handle_qxfer_pkt`**(char *remcom_in_buffer, char *remcom_out_buffer)
Handle architecture specific GDB XML packets.

Parameters

`char *remcom_in_buffer` The buffer of the packet we have read.

`char *remcom_out_buffer` The buffer of `BUFMAX` bytes to write a packet into.

void **`kgdb_call_nmi_hook`**(void *ignored)
Call kgdb_nmicallback() on the current CPU

Parameters

`void *ignored` This parameter is only here to match the prototype.

If you're using the default implementation of *`kgdb_roundup_cpus()`* this function will be called per CPU. If you don't implement *`kgdb_call_nmi_hook()`* a default will be used.

void **`kgdb_roundup_cpus`**(void)
Get other CPUs into a holding pattern

Parameters

`void` no arguments

Description

On SMP systems, we need to get the attention of the other CPUs and get them into a known state. This should do what is needed to get the other CPUs to call kgdb_wait(). Note that on some arches, the NMI approach is not used for rounding up all the CPUs. Normally those architectures can just not implement this and get the default.

On non-SMP systems, this is not called.

void **`kgdb_arch_set_pc`**(struct pt_regs *regs, unsigned long pc)
Generic call back to the program counter

Parameters

`struct pt_regs *regs` Current `struct pt_regs`.

`unsigned long pc` The new value for the program counter

This function handles updating the program counter and requires an architecture specific implementation.

void **`kgdb_arch_late`**(void)
Perform any architecture specific initialization.

Parameters

`void` no arguments

Description

This function will handle the late initialization of any architecture specific callbacks. This is an optional function for handling things like late initialization of hw breakpoints. The default implementation does nothing.

struct **kgdb_arch**
Describe architecture specific values.

Definition

```
struct kgdb_arch {
  unsigned char           gdb_bpt_instr[BREAK_INSTR_SIZE];
  unsigned long           flags;
  int (*set_breakpoint)(unsigned long, char *);
  int (*remove_breakpoint)(unsigned long, char *);
  int (*set_hw_breakpoint)(unsigned long, int, enum kgdb_bptype);
  int (*remove_hw_breakpoint)(unsigned long, int, enum kgdb_bptype);
  void (*disable_hw_break)(struct pt_regs *regs);
  void (*remove_all_hw_break)(void);
  void (*correct_hw_break)(void);
  void (*enable_nmi)(bool on);
};
```

Members

gdb_bpt_instr The instruction to trigger a breakpoint.

flags Flags for the breakpoint, currently just KGDB_HW_BREAKPOINT.

set_breakpoint Allow an architecture to specify how to set a software breakpoint.

remove_breakpoint Allow an architecture to specify how to remove a software breakpoint.

set_hw_breakpoint Allow an architecture to specify how to set a hardware breakpoint.

remove_hw_breakpoint Allow an architecture to specify how to remove a hardware breakpoint.

disable_hw_break Allow an architecture to specify how to disable hardware breakpoints for a single cpu.

remove_all_hw_break Allow an architecture to specify how to remove all hardware breakpoints.

correct_hw_break Allow an architecture to specify how to correct the hardware debug registers.

enable_nmi Manage NMI-triggered entry to KGDB

struct **kgdb_io**
Describe the interface for an I/O driver to talk with KGDB.

Definition

```
struct kgdb_io {
  const char              *name;
  int (*read_char) (void);
  void (*write_char) (u8);
  void (*flush) (void);
  int (*init) (void);
  void (*deinit) (void);
  void (*pre_exception) (void);
  void (*post_exception) (void);
  struct console          *cons;
};
```

Members

name Name of the I/O driver.

read_char Pointer to a function that will return one char.

write_char Pointer to a function that will write one char.

flush Pointer to a function that will flush any pending writes.

init Pointer to a function that will initialize the device.

deinit Pointer to a function that will deinit the device. Implies that this I/O driver is temporary and expects to be replaced. Called when an I/O driver is replaced or explicitly unregistered.

pre_exception Pointer to a function that will do any prep work for the I/O driver.

post_exception Pointer to a function that will do any cleanup work for the I/O driver.

cons valid if the I/O device is a console; else NULL.

13.8.2 kgdboc internals

kgdboc and uarts

The kgdboc driver is actually a very thin driver that relies on the underlying low level to the hardware driver having "polling hooks" to which the tty driver is attached. In the initial implementation of kgdboc the serial_core was changed to expose a low level UART hook for doing polled mode reading and writing of a single character while in an atomic context. When kgdb makes an I/O request to the debugger, kgdboc invokes a callback in the serial core which in turn uses the callback in the UART driver.

When using kgdboc with a UART, the UART driver must implement two callbacks in the struct uart_ops. Example from `drivers/8250.c`:

```
#ifdef CONFIG_CONSOLE_POLL
    .poll_get_char = serial8250_get_poll_char,
    .poll_put_char = serial8250_put_poll_char,
#endif
```

Any implementation specifics around creating a polling driver use the `#ifdef CONFIG_CONSOLE_POLL`, as shown above. Keep in mind that polling hooks have to be implemented in such a way that they can be called from an atomic context and have to restore the state of the UART chip on return such that the system can return to normal when the debugger detaches. You need to be very careful with any kind of lock you consider, because failing here is most likely going to mean pressing the reset button.

kgdboc and keyboards

The kgdboc driver contains logic to configure communications with an attached keyboard. The keyboard infrastructure is only compiled into the kernel when `CONFIG_KDB_KEYBOARD=y` is set in the kernel configuration.

The core polled keyboard driver for PS/2 type keyboards is in `drivers/char/kdb_keyboard.c`. This driver is hooked into the debug core when kgdboc populates the callback in the array called `kdb_poll_funcs[]`. The kdb_get_kbd_char() is the top-level function which polls hardware for single character input.

kgdboc and kms

The kgdboc driver contains logic to request the graphics display to switch to a text context when you are using `kgdboc=kms,kbd`, provided that you have a video driver which has a frame buffer console and atomic kernel mode setting support.

Every time the kernel debugger is entered it calls kgdboc_pre_exp_handler() which in turn calls con_debug_enter() in the virtual console layer. On resuming kernel execution, the kernel debugger calls kgdboc_post_exp_handler() which in turn calls con_debug_leave().

Any video driver that wants to be compatible with the kernel debugger and the atomic kms callbacks must implement the `mode_set_base_atomic`, `fb_debug_enter` and `fb_debug_leave operations`. For the `fb_debug_enter` and `fb_debug_leave` the option exists to use the generic drm fb helper functions or implement something custom for the hardware. The following example shows the initialization of the .mode_set_base_atomic operation in drivers/gpu/drm/i915/intel_display.c:

```
static const struct drm_crtc_helper_funcs intel_helper_funcs = {
[...]
        .mode_set_base_atomic = intel_pipe_set_base_atomic,
[...]
};
```

Here is an example of how the i915 driver initializes the fb_debug_enter and fb_debug_leave functions to use the generic drm helpers in `drivers/gpu/drm/i915/intel_fb.c`:

```
static struct fb_ops intelfb_ops = {
[...]
       .fb_debug_enter = drm_fb_helper_debug_enter,
       .fb_debug_leave = drm_fb_helper_debug_leave,
[...]
};
```

13.9 Credits

The following people have contributed to this document:

1. Amit Kale <amitkale@linsyssoft.com>
2. Tom Rini <trini@kernel.crashing.org>

In March 2008 this document was completely rewritten by:

- Jason Wessel <jason.wessel@windriver.com>

In Jan 2010 this document was updated to include kdb.

- Jason Wessel <jason.wessel@windriver.com>

CHAPTER

FOURTEEN

LINUX KERNEL SELFTESTS

The kernel contains a set of “self tests” under the tools/testing/selftests/ directory. These are intended to be small tests to exercise individual code paths in the kernel. Tests are intended to be run after building, installing and booting a kernel.

Kselftest from mainline can be run on older stable kernels. Running tests from mainline offers the best coverage. Several test rings run mainline kselftest suite on stable releases. The reason is that when a new test gets added to test existing code to regression test a bug, we should be able to run that test on an older kernel. Hence, it is important to keep code that can still test an older kernel and make sure it skips the test gracefully on newer releases.

You can find additional information on Kselftest framework, how to write new tests using the framework on Kselftest wiki:

https://kselftest.wiki.kernel.org/

On some systems, hot-plug tests could hang forever waiting for cpu and memory to be ready to be offlined. A special hot-plug target is created to run the full range of hot-plug tests. In default mode, hot-plug tests run in safe mode with a limited scope. In limited mode, cpu-hotplug test is run on a single cpu as opposed to all hotplug capable cpus, and memory hotplug test is run on 2% of hotplug capable memory instead of 10%.

kselftest runs as a userspace process. Tests that can be written/run in userspace may wish to use the *Test Harness*. Tests that need to be run in kernel space may wish to use a *Test Module*.

14.1 Running the selftests (hotplug tests are run in limited mode)

To build the tests:

```
$ make -C tools/testing/selftests
```

To run the tests:

```
$ make -C tools/testing/selftests run_tests
```

To build and run the tests with a single command, use:

```
$ make kselftest
```

Note that some tests will require root privileges.

Kselftest supports saving output files in a separate directory and then running tests. To locate output files in a separate directory two syntaxes are supported. In both cases the working directory must be the root of the kernel src. This is applicable to "Running a subset of selftests" section below.

To build, save output files in a separate directory with O=

```
$ make O=/tmp/kselftest kselftest
```

To build, save output files in a separate directory with KBUILD_OUTPUT

```
$ export KBUILD_OUTPUT=/tmp/kselftest; make kselftest
```

The O= assignment takes precedence over the KBUILD_OUTPUT environment variable.

The above commands by default run the tests and print full pass/fail report. Kselftest supports "summary" option to make it easier to understand the test results. Please find the detailed individual test results for each test in /tmp/testname file(s) when summary option is specified. This is applicable to "Running a subset of selftests" section below.

To run kselftest with summary option enabled

```
$ make summary=1 kselftest
```

14.2 Running a subset of selftests

You can use the "TARGETS" variable on the make command line to specify single test to run, or a list of tests to run.

To run only tests targeted for a single subsystem:

```
$ make -C tools/testing/selftests TARGETS=ptrace run_tests
```

You can specify multiple tests to build and run:

```
$  make TARGETS="size timers" kselftest
```

To build, save output files in a separate directory with O=

```
$ make O=/tmp/kselftest TARGETS="size timers" kselftest
```

To build, save output files in a separate directory with KBUILD_OUTPUT

```
$ export KBUILD_OUTPUT=/tmp/kselftest; make TARGETS="size timers" kselftest
```

Additionally you can use the "SKIP_TARGETS" variable on the make command line to specify one or more targets to exclude from the TARGETS list.

To run all tests but a single subsystem:

```
$ make -C tools/testing/selftests SKIP_TARGETS=ptrace run_tests
```

You can specify multiple tests to skip:

```
$  make SKIP_TARGETS="size timers" kselftest
```

You can also specify a restricted list of tests to run together with a dedicated skiplist:

```
$  make TARGETS="bpf breakpoints size timers" SKIP_TARGETS=bpf kselftest
```

See the top-level tools/testing/selftests/Makefile for the list of all possible targets.

14.3 Running the full range hotplug selftests

To build the hotplug tests:

```
$ make -C tools/testing/selftests hotplug
```

To run the hotplug tests:

```
$ make -C tools/testing/selftests run_hotplug
```

Note that some tests will require root privileges.

14.4 Install selftests

You can use the "install" target of "make" (which calls the *kselftest_install.sh* tool) to install selftests in the default location (*tools/testing/selftests/kselftest_install*), or in a user specified location via the *INSTALL_PATH* "make" variable.

To install selftests in default location:

```
$ make -C tools/testing/selftests install
```

To install selftests in a user specified location:

```
$ make -C tools/testing/selftests install INSTALL_PATH=/some/other/path
```

14.5 Running installed selftests

Found in the install directory, as well as in the Kselftest tarball, is a script named *run_kselftest.sh* to run the tests.

You can simply do the following to run the installed Kselftests. Please note some tests will require root privileges:

```
$ cd kselftest_install
$ ./run_kselftest.sh
```

To see the list of available tests, the *-l* option can be used:

```
$ ./run_kselftest.sh -l
```

The *-c* option can be used to run all the tests from a test collection, or the *-t* option for specific single tests. Either can be used multiple times:

```
$ ./run_kselftest.sh -c bpf -c seccomp -t timers:posix_timers -t␣
↪timer:nanosleep
```

For other features see the script usage output, seen with the *-h* option.

14.6 Packaging selftests

In some cases packaging is desired, such as when tests need to run on a different system. To package selftests, run:

```
$ make -C tools/testing/selftests gen_tar
```

This generates a tarball in the *INSTALL_PATH/kselftest-packages* directory. By default, *.gz* format is used. The tar compression format can be overridden by specifying a *FORMAT* make variable. Any value recognized by tar's auto-compress option is supported, such as:

```
$ make -C tools/testing/selftests gen_tar FORMAT=.xz
```

make gen_tar invokes *make install* so you can use it to package a subset of tests by using variables specified in *Running a subset of selftests* section:

```
$ make -C tools/testing/selftests gen_tar TARGETS="bpf" FORMAT=.xz
```

14.7 Contributing new tests

In general, the rules for selftests are

- Do as much as you can if you're not root;
- Don't take too long;
- Don't break the build on any architecture, and
- Don't cause the top-level "make run_tests" to fail if your feature is unconfigured.

14.8 Contributing new tests (details)

- Use TEST_GEN_XXX if such binaries or files are generated during compiling.

  TEST_PROGS, TEST_GEN_PROGS mean it is the executable tested by default.

  TEST_CUSTOM_PROGS should be used by tests that require custom build rules and prevent common build rule use.

  TEST_PROGS are for test shell scripts. Please ensure shell script has its exec bit set. Otherwise, lib.mk run_tests will generate a warning.

  TEST_CUSTOM_PROGS and TEST_PROGS will be run by common run_tests.

TEST_PROGS_EXTENDED, TEST_GEN_PROGS_EXTENDED mean it is the executable which is not tested by default. TEST_FILES, TEST_GEN_FILES mean it is the file which is used by test.

- First use the headers inside the kernel source and/or git repo, and then the system headers. Headers for the kernel release as opposed to headers installed by the distro on the system should be the primary focus to be able to find regressions.
- If a test needs specific kernel config options enabled, add a config file in the test directory to enable them.

  e.g: tools/testing/selftests/android/config

14.9 Test Module

Kselftest tests the kernel from userspace. Sometimes things need testing from within the kernel, one method of doing this is to create a test module. We can tie the module into the kselftest framework by using a shell script test runner. `kselftest/module.sh` is designed to facilitate this process. There is also a header file provided to assist writing kernel modules that are for use with kselftest:

- `tools/testing/selftests/kselftest_module.h`
- `tools/testing/selftests/kselftest/module.sh`

14.9.1 How to use

Here we show the typical steps to create a test module and tie it into kselftest. We use kselftests for lib/ as an example.

1. Create the test module
2. Create the test script that will run (load/unload) the module e.g. `tools/testing/selftests/lib/printf.sh`
3. Add line to config file e.g. `tools/testing/selftests/lib/config`
4. Add test script to makefile e.g. `tools/testing/selftests/lib/Makefile`
5. Verify it works:

```
# Assumes you have booted a fresh build of this kernel tree
cd /path/to/linux/tree
make kselftest-merge
make modules
sudo make modules_install
make TARGETS=lib kselftest
```

14.9.2 Example Module

A bare bones test module might look like this:

```
// SPDX-License-Identifier: GPL-2.0+

#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#include "../tools/testing/selftests/kselftest/module.h"

KSTM_MODULE_GLOBALS();

/*
 * Kernel module for testing the foobinator
 */

static int __init test_function()
{
        ...
}

static void __init selftest(void)
{
        KSTM_CHECK_ZERO(do_test_case("", 0));
}

KSTM_MODULE_LOADERS(test_foo);
MODULE_AUTHOR("John Developer <jd@fooman.org>");
MODULE_LICENSE("GPL");
```

14.9.3 Example test script

```
#!/bin/bash
# SPDX-License-Identifier: GPL-2.0+
$(dirname $0)/../kselftest/module.sh "foo" test_foo
```

14.10 Test Harness

The kselftest_harness.h file contains useful helpers to build tests. The test harness is for userspace testing, for kernel space testing see *Test Module* above.

The tests from tools/testing/selftests/seccomp/seccomp_bpf.c can be used as example.

14.10.1 Example

```
#include "../kselftest_harness.h"

TEST(standalone_test) {
  do_some_stuff;
  EXPECT_GT(10, stuff) {
     stuff_state_t state;
     enumerate_stuff_state(&state);
     TH_LOG("expectation failed with state: %s", state.msg);
  }
  more_stuff;
  ASSERT_NE(some_stuff, NULL) TH_LOG("how did it happen?!");
  last_stuff;
  EXPECT_EQ(0, last_stuff);
}

FIXTURE(my_fixture) {
  mytype_t *data;
  int awesomeness_level;
};
FIXTURE_SETUP(my_fixture) {
  self->data = mytype_new();
  ASSERT_NE(NULL, self->data);
}
FIXTURE_TEARDOWN(my_fixture) {
  mytype_free(self->data);
}
TEST_F(my_fixture, data_is_good) {
  EXPECT_EQ(1, is_my_data_good(self->data));
}

TEST_HARNESS_MAIN
```

14.10.2 Helpers

TH_LOG

`TH_LOG (fmt, ...)`

Parameters

fmt format string

... optional arguments

Description

```
TH_LOG(format, ...)
```

Optional debug logging function available for use in tests. Logging may be enabled or disabled by defining TH_LOG_ENABLED. E.g., #define TH_LOG_ENABLED 1

If no definition is provided, logging is enabled by default.

If there is no way to print an error message for the process running the test (e.g. not allowed to write to stderr), it is still possible to get the ASSERT_* number for which the test failed. This behavior can be enabled by writing *_metadata->no_print = true;* before the check sequence that is unable to print. When an error occur, instead of printing an error message and calling *abort(3)*, the test process call *_exit(2)* with the assert number as argument, which is then printed by the parent process.

TEST

`TEST (test_name)`

Defines the test function and creates the registration stub

Parameters

`test_name` test name

Description

```
TEST(name) { implementation }
```

Defines a test by name. Names must be unique and tests must not be run in parallel. The implementation containing block is a function and scoping should be treated as such. Returning early may be performed with a bare "return;" statement.

EXPECT_* and ASSERT_* are valid in a *`TEST()`* { } context.

TEST_SIGNAL

`TEST_SIGNAL (test_name, signal)`

Parameters

`test_name` test name

`signal` signal number

Description

```
TEST_SIGNAL(name, signal) { implementation }
```

Defines a test by name and the expected term signal. Names must be unique and tests must not be run in parallel. The implementation containing block is a function and scoping should be treated as such. Returning early may be performed with a bare "return;" statement.

EXPECT_* and ASSERT_* are valid in a *`TEST()`* { } context.

FIXTURE_DATA

`FIXTURE_DATA (datatype_name)`

Wraps the struct name so we have one less argument to pass around

Parameters

`datatype_name` datatype name

Description

```
FIXTURE_DATA(datatype_name)
```

Almost always, you want just *FIXTURE()* instead (see below). This call may be used when the type of the fixture data is needed. In general, this should not be needed unless the *self* is being passed to a helper directly.

FIXTURE

FIXTURE (fixture_name)

Called once per fixture to setup the data and register

Parameters

fixture_name fixture name

Description

```
FIXTURE(fixture_name) {
  type property1;
  ...
};
```

Defines the data provided to *TEST_F()*-defined tests as *self*. It should be populated and cleaned up using *FIXTURE_SETUP()* and *FIXTURE_TEARDOWN()*.

FIXTURE_SETUP

FIXTURE_SETUP (fixture_name)

Prepares the setup function for the fixture. *_metadata* is included so that EXPECT_* and ASSERT_* work correctly.

Parameters

fixture_name fixture name

Description

```
FIXTURE_SETUP(fixture_name) { implementation }
```

Populates the required "setup" function for a fixture. An instance of the datatype defined with *FIXTURE_DATA()* will be exposed as *self* for the implementation.

ASSERT_* are valid for use in this context and will prempt the execution of any dependent fixture tests.

A bare "return;" statement may be used to return early.

FIXTURE_TEARDOWN

FIXTURE_TEARDOWN (fixture_name)

Parameters

fixture_name fixture name

Description

_metadata is included so that EXPECT_* and ASSERT_* work correctly.

```
FIXTURE_TEARDOWN(fixture_name) { implementation }
```

Populates the required "teardown" function for a fixture. An instance of the datatype defined with *FIXTURE_DATA()* will be exposed as *self* for the implementation to clean up.

A bare "return;" statement may be used to return early.

FIXTURE_VARIANT

`FIXTURE_VARIANT (fixture_name)`

Optionally called once per fixture to declare fixture variant

Parameters

fixture_name fixture name

Description

```
FIXTURE_VARIANT(fixture_name) {
  type property1;
  ...
};
```

Defines type of constant parameters provided to *FIXTURE_SETUP()*, *TEST_F()* and FIXTURE_TEARDOWN as *variant*. Variants allow the same tests to be run with different arguments.

FIXTURE_VARIANT_ADD

`FIXTURE_VARIANT_ADD (fixture_name, variant_name)`

Called once per fixture variant to setup and register the data

Parameters

fixture_name fixture name

variant_name name of the parameter set

Description

```
FIXTURE_VARIANT_ADD(fixture_name, variant_name) {
  .property1 = val1,
  ...
};
```

Defines a variant of the test fixture, provided to *FIXTURE_SETUP()* and *TEST_F()* as *variant*. Tests of each fixture will be run once for each variant.

TEST_F

`TEST_F (fixture_name, test_name)`

Emits test registration and helpers for fixture-based test cases

Parameters

fixture_name fixture name

test_name test name

Description

```
TEST_F(fixture, name) { implementation }
```

Defines a test that depends on a fixture (e.g., is part of a test case). Very similar to *TEST()* except that *self* is the setup instance of fixture's datatype exposed for use by the implementation.

TEST_HARNESS_MAIN

`TEST_HARNESS_MAIN ()`

Simple wrapper to run the test harness

Parameters

Description

```
TEST_HARNESS_MAIN
```

Use once to append a main() to the test file.

14.10.3 Operators

Operators for use in *TEST()* and *TEST_F()*. ASSERT_* calls will stop test execution immediately. EXPECT_* calls will emit a failure warning, note it, and continue.

ASSERT_EQ

`ASSERT_EQ (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_EQ(expected, measured): expected == measured

ASSERT_NE

`ASSERT_NE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_NE(expected, measured): expected != measured

ASSERT_LT

`ASSERT_LT (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_LT(expected, measured): expected < measured

ASSERT_LE

`ASSERT_LE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_LE(expected, measured): expected <= measured

ASSERT_GT

`ASSERT_GT (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_GT(expected, measured): expected > measured

ASSERT_GE

`ASSERT_GE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_GE(expected, measured): expected >= measured

ASSERT_NULL

`ASSERT_NULL (seen)`

Parameters

seen measured value

Description

ASSERT_NULL(measured): NULL == measured

ASSERT_TRUE

`ASSERT_TRUE (seen)`

Parameters

seen measured value

Description

ASSERT_TRUE(measured): measured != 0

ASSERT_FALSE

`ASSERT_FALSE (seen)`

Parameters

seen measured value

Description

ASSERT_FALSE(measured): measured == 0

ASSERT_STREQ

`ASSERT_STREQ (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_STREQ(expected, measured): !strcmp(expected, measured)

ASSERT_STRNE

`ASSERT_STRNE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

ASSERT_STRNE(expected, measured): strcmp(expected, measured)

EXPECT_EQ

`EXPECT_EQ (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_EQ(expected, measured): expected == measured

EXPECT_NE

`EXPECT_NE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_NE(expected, measured): expected != measured

EXPECT_LT

`EXPECT_LT (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_LT(expected, measured): expected < measured

EXPECT_LE

`EXPECT_LE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_LE(expected, measured): expected <= measured

EXPECT_GT

`EXPECT_GT (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_GT(expected, measured): expected > measured

EXPECT_GE

`EXPECT_GE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_GE(expected, measured): expected >= measured

EXPECT_NULL

`EXPECT_NULL (seen)`

Parameters

seen measured value

Description

EXPECT_NULL(measured): NULL == measured

EXPECT_TRUE

`EXPECT_TRUE (seen)`

Parameters

seen measured value

Description

EXPECT_TRUE(measured): 0 != measured

EXPECT_FALSE

`EXPECT_FALSE (seen)`

Parameters

seen measured value

Description

EXPECT_FALSE(measured): 0 == measured

EXPECT_STREQ

`EXPECT_STREQ (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_STREQ(expected, measured): !strcmp(expected, measured)

EXPECT_STRNE

`EXPECT_STRNE (expected, seen)`

Parameters

expected expected value

seen measured value

Description

EXPECT_STRNE(expected, measured): strcmp(expected, measured)

CHAPTER

FIFTEEN

KUNIT - LINUX KERNEL UNIT TESTING

15.1 Getting Started

15.1.1 Installing Dependencies

KUnit has the same dependencies as the Linux kernel. As long as you can build the kernel, you can run KUnit.

15.1.2 Running tests with kunit_tool

kunit_tool is a Python script, which configures and builds a kernel, runs tests, and formats the test results. From the kernel repository, you can run kunit_tool:

```
./tools/testing/kunit/kunit.py run
```

For more information on this wrapper, see: *Run Tests with kunit_tool*.

Creating a `.kunitconfig`

By default, kunit_tool runs a selection of tests. However, you can specify which unit tests to run by creating a `.kunitconfig` file with kernel config options that enable only a specific set of tests and their dependencies. The `.kunitconfig` file contains a list of kconfig options which are required to run the desired targets. The `.kunitconfig` also contains any other test specific config options, such as test dependencies. For example: the `FAT_FS` tests - `FAT_KUNIT_TEST`, depends on `FAT_FS`. `FAT_FS` can be enabled by selecting either `MSDOS_FS` or `VFAT_FS`. To run `FAT_KUNIT_TEST`, the `.kunitconfig` has:

```
CONFIG_KUNIT=y
CONFIG_MSDOS_FS=y
CONFIG_FAT_KUNIT_TEST=y
```

1. A good starting point for the `.kunitconfig` is the KUnit default config. You can generate it by running:

```
cd $PATH_TO_LINUX_REPO
tools/testing/kunit/kunit.py config
cat .kunit/.kunitconfig
```

Note: `.kunitconfig` lives in the `--build_dir` used by kunit.py, which is `.kunit` by default.

Note: You may want to remove CONFIG_KUNIT_ALL_TESTS from the `.kunitconfig` as it will enable a number of additional tests that you may not want.

2. You can then add any other Kconfig options, for example:

```
CONFIG_LIST_KUNIT_TEST=y
```

Before running the tests, kunit_tool ensures that all config options set in `.kunitconfig` are set in the kernel `.config`. It will warn you if you have not included dependencies for the options used.

Note: If you change the `.kunitconfig`, kunit.py will trigger a rebuild of the `.config` file. But you can edit the `.config` file directly or with tools like `make menuconfig O=.kunit`. As long as its a superset of `.kunitconfig`, kunit.py won't overwrite your changes.

Running Tests (KUnit Wrapper)

1. To make sure that everything is set up correctly, invoke the Python wrapper from your kernel repository:

```
./tools/testing/kunit/kunit.py run
```

If everything worked correctly, you should see the following:

```
Generating .config ...
Building KUnit Kernel ...
Starting KUnit Kernel ...
```

The tests will pass or fail.

Note: Because it is building a lot of sources for the first time, the `Building KUnit kernel` may take a while.

15.1.3 Running Tests without the KUnit Wrapper

If you do not want to use the KUnit Wrapper (for example: you want code under test to integrate with other systems, or use a different/ unsupported architecture or configuration), KUnit can be included in any kernel, and the results are read out and parsed manually.

Note: `CONFIG_KUNIT` should not be enabled in a production environment. Enabling KUnit disables Kernel Address-Space Layout Randomization (KASLR), and tests may affect the state of the kernel in ways not suitable for production.

Configuring the Kernel

To enable KUnit itself, you need to enable the `CONFIG_KUNIT` Kconfig option (under Kernel Hacking/Kernel Testing and Coverage in `menuconfig`). From there, you can enable any KUnit tests. They usually have config options ending in `_KUNIT_TEST`.

KUnit and KUnit tests can be compiled as modules. The tests in a module will run when the module is loaded.

Running Tests (without KUnit Wrapper)

Build and run your kernel. In the kernel log, the test output is printed out in the TAP format. This will only happen by default if KUnit/tests are built-in. Otherwise the module will need to be loaded.

Note: Some lines and/or data may get interspersed in the TAP output.

15.1.4 Writing Your First Test

In your kernel repository, let's add some code that we can test.

1. Create a file `drivers/misc/example.h`, which includes:

```
int misc_example_add(int left, int right);
```

2. Create a file `drivers/misc/example.c`, which includes:

```
#include <linux/errno.h>

#include "example.h"

int misc_example_add(int left, int right)
{
        return left + right;
}
```

3. Add the following lines to `drivers/misc/Kconfig`:

```
config MISC_EXAMPLE
        bool "My example"
```

4. Add the following lines to `drivers/misc/Makefile`:

```
obj-$(CONFIG_MISC_EXAMPLE) += example.o
```

Now we are ready to write the test cases.

1. Add the below test case in drivers/misc/example_test.c:

```
#include <kunit/test.h>
#include "example.h"

/* Define the test cases. */

static void misc_example_add_test_basic(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 1, misc_example_add(1, 0));
        KUNIT_EXPECT_EQ(test, 2, misc_example_add(1, 1));
        KUNIT_EXPECT_EQ(test, 0, misc_example_add(-1, 1));
        KUNIT_EXPECT_EQ(test, INT_MAX, misc_example_add(0, INT_MAX));
        KUNIT_EXPECT_EQ(test, -1, misc_example_add(INT_MAX, INT_MIN));
}

static void misc_example_test_failure(struct kunit *test)
{
        KUNIT_FAIL(test, "This test never passes.");
}

static struct kunit_case misc_example_test_cases[] = {
        KUNIT_CASE(misc_example_add_test_basic),
        KUNIT_CASE(misc_example_test_failure),
        {}
};

static struct kunit_suite misc_example_test_suite = {
        .name = "misc-example",
        .test_cases = misc_example_test_cases,
};
kunit_test_suite(misc_example_test_suite);
```

2. Add the following lines to drivers/misc/Kconfig:

```
config MISC_EXAMPLE_TEST
        tristate "Test for my example" if !KUNIT_ALL_TESTS
        depends on MISC_EXAMPLE && KUNIT=y
        default KUNIT_ALL_TESTS
```

3. Add the following lines to drivers/misc/Makefile:

```
obj-$(CONFIG_MISC_EXAMPLE_TEST) += example_test.o
```

4. Add the following lines to .kunitconfig:

```
CONFIG_MISC_EXAMPLE=y
CONFIG_MISC_EXAMPLE_TEST=y
```

5. Run the test:

```
./tools/testing/kunit/kunit.py run
```

You should see the following failure:

```
...
[16:08:57] [PASSED] misc-example:misc_example_add_test_basic
[16:08:57] [FAILED] misc-example:misc_example_test_failure
[16:08:57] EXPECTATION FAILED at drivers/misc/example-test.c:17
[16:08:57]      This test never passes.
...
```

Congrats! You just wrote your first KUnit test.

15.1.5 Next Steps

- *KUnit Architecture* - KUnit architecture.
- *Run Tests with kunit_tool* - run kunit_tool.
- *Run Tests without kunit_tool* - run tests without kunit_tool.
- *Writing Tests* - write tests.
- *Tips For Writing KUnit Tests* - best practices with examples.
- *API Reference* - KUnit APIs used for testing.
- *kunit_tool How-To* - kunit_tool helper script.
- *Frequently Asked Questions* - KUnit common questions and answers.

15.2 KUnit Architecture

The KUnit architecture can be divided into two parts:

- Kernel testing library
- kunit_tool (Command line test harness)

15.2.1 In-Kernel Testing Framework

The kernel testing library supports KUnit tests written in C using KUnit. KUnit tests are kernel code. KUnit does several things:

- Organizes tests
- Reports test results
- Provides test utilities

Test Cases

The fundamental unit in KUnit is the test case. The KUnit test cases are grouped into KUnit suites. A KUnit test case is a function with type signature `void (*)(struct kunit *test)`. These test case functions are wrapped in a struct called *struct kunit_case*.

Each KUnit test case gets a `struct kunit` context object passed to it that tracks a running test. The KUnit assertion macros and other KUnit utilities use the `struct kunit` context object. As an exception, there are two fields:

- `->priv`: The setup functions can use it to store arbitrary test user data.
- `->param_value`: It contains the parameter value which can be retrieved in the parameterized tests.

Test Suites

A KUnit suite includes a collection of test cases. The KUnit suites are represented by the `struct kunit_suite`. For example:

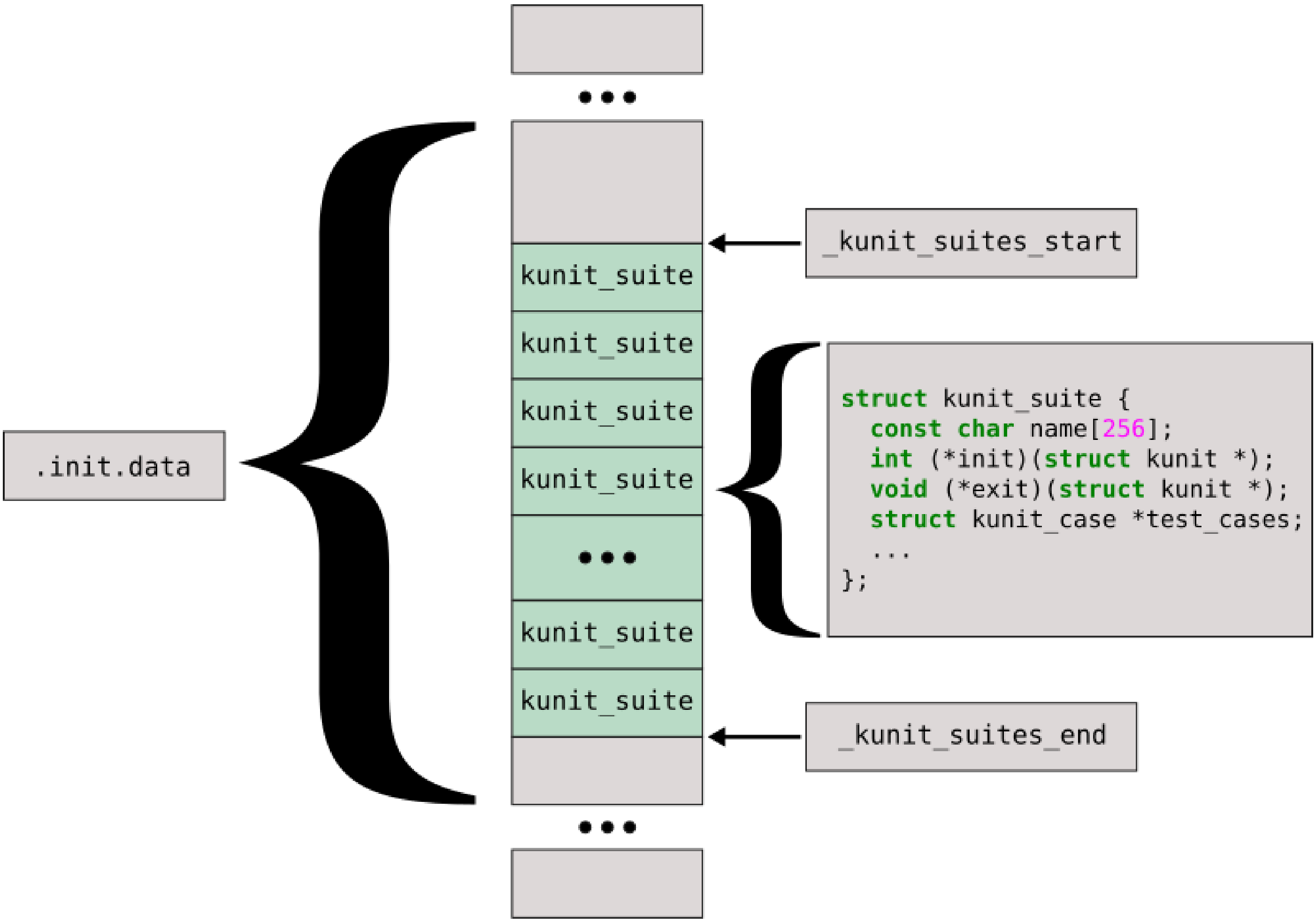
```
static struct kunit_case example_test_cases[] = {
        KUNIT_CASE(example_test_foo),
        KUNIT_CASE(example_test_bar),
        KUNIT_CASE(example_test_baz),
        {}
};

static struct kunit_suite example_test_suite = {
        .name = "example",
        .init = example_test_init,
        .exit = example_test_exit,
        .test_cases = example_test_cases,
};
kunit_test_suite(example_test_suite);
```

In the above example, the test suite `example_test_suite`, runs the test cases `example_test_foo`, `example_test_bar`, and `example_test_baz`. Before running the test, the `example_test_init` is called and after running the test, `example_test_exit` is called. The `kunit_test_suite(example_test_suite)` registers the test suite with the KUnit test framework.

Executor

The KUnit executor can list and run built-in KUnit tests on boot. The Test suites are stored in a linker section called `.kunit_test_suites`. For code, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/asm-generic/vmlinux.lds.h?h=v5.15#n945. The linker section consists of an array of pointers to `struct kunit_suite`, and is populated by the `kunit_test_suites()` macro. To run all tests compiled into the kernel, the KUnit executor iterates over the linker section array.

On the kernel boot, the KUnit executor uses the start and end addresses of this section to iterate over and run all tests. For code, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/



```
struct kunit_suite {
  const char name[256];
  int (*init)(struct kunit *);
  void (*exit)(struct kunit *);
  struct kunit_case *test_cases;
  ...
};
```

Fig. 1: KUnit Suite Memory Diagram

linux.git/tree/lib/kunit/executor.c

When built as a module, the `kunit_test_suites()` macro defines a `module_init()` function, which runs all the tests in the compilation unit instead of utilizing the executor.

In KUnit tests, some error classes do not affect other tests or parts of the kernel, each KUnit case executes in a separate thread context. For code, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/lib/kunit/try-catch.c?h=v5.15#n58

Assertion Macros

KUnit tests verify state using expectations/assertions. All expectations/assertions are formatted as: `KUNIT_{EXPECT|ASSERT}_<op>[_MSG](kunit, property[, message])`

- `{EXPECT|ASSERT}` determines whether the check is an assertion or an expectation.
  - For an expectation, if the check fails, marks the test as failed and logs the failure.
  - An assertion, on failure, causes the test case to terminate immediately.
    - Assertions call function: `void __noreturn kunit_abort(struct kunit *)`.
    - `kunit_abort` calls function: `void __noreturn kunit_try_catch_throw(struct kunit_try_catch *try_catch)`.
    - `kunit_try_catch_throw` calls function: `void kthread_complete_and_exit(struct completion *, long) __noreturn;` and terminates the special thread context.
- `<op>` denotes a check with options: `TRUE` (supplied property has the boolean value "true"), `EQ` (two supplied properties are equal), `NOT_ERR_OR_NULL` (supplied pointer is not null and does not contain an "err" value).
- `[_MSG]` prints a custom message on failure.

Test Result Reporting

KUnit prints test results in KTAP format. KTAP is based on TAP14, see: https://github.com/isaacs/testanything.github.io/blob/tap14/tap-version-14-specification.md. KTAP (yet to be standardized format) works with KUnit and Kselftest. The KUnit executor prints KTAP results to dmesg, and debugfs (if configured).

Parameterized Tests

Each KUnit parameterized test is associated with a collection of parameters. The test is invoked multiple times, once for each parameter value and the parameter is stored in the `param_value` field. The test case includes a *`KUNIT_CASE_PARAM()`* macro that accepts a generator function. The generator function is passed the previous parameter and returns the next parameter. It also provides a macro to generate common-case generators based on arrays.

15.2.2 kunit_tool (Command Line Test Harness)

kunit_tool is a Python script (`tools/testing/kunit/kunit.py`) that can be used to configure, build, exec, parse and run (runs other commands in order) test results. You can either run KUnit tests using kunit_tool or can include KUnit in kernel and parse manually.

- `configure` command generates the kernel `.config` from a `.kunitconfig` file (and any architecture-specific options). For some architectures, additional config options are specified in the `qemu_config` Python script (For example: `tools/testing/kunit/qemu_configs/powerpc.py`). It parses both the existing `.config` and the `.kunitconfig` files and ensures that `.config` is a superset of `.kunitconfig`. If this is not the case, it will combine the two and run `make olddefconfig` to regenerate the `.config` file. It then verifies that `.config` is now a superset. This checks if all Kconfig dependencies are correctly specified in `.kunitconfig`. `kunit_config.py` includes the parsing Kconfigs code. The code which runs `make olddefconfig` is a part of `kunit_kernel.py`. You can invoke this command via: `./tools/testing/kunit/kunit.py config` and generate a `.config` file.
- `build` runs `make` on the kernel tree with required options (depends on the architecture and some options, for example: build_dir) and reports any errors. To build a KUnit kernel from the current `.config`, you can use the `build` argument: `./tools/testing/kunit/kunit.py build`.
- `exec` command executes kernel results either directly (using User-mode Linux configuration), or via an emulator such as QEMU. It reads results from the log via standard output (stdout), and passes them to `parse` to be parsed. If you already have built a kernel with built-in KUnit tests, you can run the kernel and display the test results with the `exec` argument: `./tools/testing/kunit/kunit.py exec`.
- `parse` extracts the KTAP output from a kernel log, parses the test results, and prints a summary. For failed tests, any diagnostic output will be included.

15.3 Run Tests with kunit_tool

We can either run KUnit tests using kunit_tool or can run tests manually, and then use kunit_tool to parse the results. To run tests manually, see: *Run Tests without kunit_tool*. As long as we can build the kernel, we can run KUnit.

kunit_tool is a Python script which configures and builds a kernel, runs tests, and formats the test results.

Run command:

```
./tools/testing/kunit/kunit.py run
```

We should see the following:

```
Generating .config...
Building KUnit kernel...
Starting KUnit kernel...
```

We may want to use the following options:

```
./tools/testing/kunit/kunit.py run --timeout=30 --jobs=`nproc --all
```

- `--timeout` sets a maximum amount of time for tests to run.
- `--jobs` sets the number of threads to build the kernel.

kunit_tool will generate a `.kunitconfig` with a default configuration, if no other `.kunitconfig` file exists (in the build directory). In addition, it verifies that the generated `.config` file contains the `CONFIG` options in the `.kunitconfig`. It is also possible to pass a separate `.kunitconfig` fragment to kunit_tool. This is useful if we have several different groups of tests we want to run independently, or if we want to use pre-defined test configs for certain subsystems.

To use a different `.kunitconfig` file (such as one provided to test a particular subsystem), pass it as an option:

```
./tools/testing/kunit/kunit.py run --kunitconfig=fs/ext4/.kunitconfig
```

To view kunit_tool flags (optional command-line arguments), run:

```
./tools/testing/kunit/kunit.py run --help
```

15.3.1 Create a `.kunitconfig` File

If we want to run a specific set of tests (rather than those listed in the KUnit `defconfig`), we can provide Kconfig options in the `.kunitconfig` file. For default .kunitconfig, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/testing/kunit/configs/default.config. A `.kunitconfig` is a `minconfig` (a .config generated by running `make savedefconfig`), used for running a specific set of tests. This file contains the regular Kernel configs with specific test targets. The `.kunitconfig` also contains any other config options required by the tests (For example: dependencies for features under tests, configs that enable/disable certain code blocks, arch configs and so on).

To create a `.kunitconfig`, using the KUnit `defconfig`:

```
cd $PATH_TO_LINUX_REPO
cp tools/testing/kunit/configs/default.config .kunit/.kunitconfig
```

We can then add any other Kconfig options. For example:

```
CONFIG_LIST_KUNIT_TEST=y
```

kunit_tool ensures that all config options in `.kunitconfig` are set in the kernel `.config` before running the tests. It warns if we have not included the options dependencies.

Note: Removing something from the `.kunitconfig` will not rebuild the `.config file`. The configuration is only updated if the `.kunitconfig` is not a subset of `.config`. This means that we can use other tools (For example: `make menuconfig`) to adjust other config options. The build dir needs to be set for `make menuconfig` to work, therefore by default use `make O=.kunit menuconfig`.

15.3.2 Configure, Build, and Run Tests

If we want to make manual changes to the KUnit build process, we can run part of the KUnit build process independently. When running kunit_tool, from a `.kunitconfig`, we can generate a `.config` by using the `config` argument:

```
./tools/testing/kunit/kunit.py config
```

To build a KUnit kernel from the current `.config`, we can use the `build` argument:

```
./tools/testing/kunit/kunit.py build
```

If we already have built UML kernel with built-in KUnit tests, we can run the kernel, and display the test results with the `exec` argument:

```
./tools/testing/kunit/kunit.py exec
```

The `run` command discussed in section: **Run Tests with kunit_tool**, is equivalent to running the above three commands in sequence.

15.3.3 Parse Test Results

KUnit tests output displays results in TAP (Test Anything Protocol) format. When running tests, kunit_tool parses this output and prints a summary. To see the raw test results in TAP format, we can pass the `--raw_output` argument:

```
./tools/testing/kunit/kunit.py run --raw_output
```

If we have KUnit results in the raw TAP format, we can parse them and print the human-readable summary with the `parse` command for kunit_tool. This accepts a filename for an argument, or will read from standard input.

```
# Reading from a file
./tools/testing/kunit/kunit.py parse /var/log/dmesg
# Reading from stdin
dmesg | ./tools/testing/kunit/kunit.py parse
```

15.3.4 Run Selected Test Suites

By passing a bash style glob filter to the `exec` or `run` commands, we can run a subset of the tests built into a kernel . For example: if we only want to run KUnit resource tests, use:

```
./tools/testing/kunit/kunit.py run 'kunit-resource*'
```

This uses the standard glob format with wildcard characters.

15.3.5 Run Tests on qemu

kunit_tool supports running tests on qemu as well as via UML. To run tests on qemu, by default it requires two flags:

- `--arch`: Selects a configs collection (Kconfig, qemu config options and so on), that allow KUnit tests to be run on the specified architecture in a minimal way. The architecture argument is same as the option name passed to the `ARCH` variable used by Kbuild. Not all architectures currently support this flag, but we can use `--qemu_config` to handle it. If `um` is passed (or this flag is ignored), the tests will run via UML. Non-UML architectures, for example: i386, x86_64, arm and so on; run on qemu.
- `--cross_compile`: Specifies the Kbuild toolchain. It passes the same argument as passed to the `CROSS_COMPILE` variable used by Kbuild. As a reminder, this will be the prefix for the toolchain binaries such as GCC. For example:
  - `sparc64-linux-gnu` if we have the sparc toolchain installed on our system.
  - `$HOME/toolchains/microblaze/gcc-9.2.0-nolibc/microblaze-linux/bin/microblaze-linux` if we have downloaded the microblaze toolchain from the 0-day website to a directory in our home directory called toolchains.

If we want to run KUnit tests on an architecture not supported by the `--arch` flag, or want to run KUnit tests on qemu using a non-default configuration; then we can write our own``QemuConfig``. These `QemuConfigs` are written in Python. They have an import line `from..qemu_config import QemuArchParams` at the top of the file. The file must contain a variable called `QEMU_ARCH` that has an instance of `QemuArchParams` assigned to it. See example in: `tools/testing/kunit/qemu_configs/x86_64.py`.

Once we have a `QemuConfig`, we can pass it into kunit_tool, using the `--qemu_config` flag. When used, this flag replaces the `--arch` flag. For example: using `tools/testing/kunit/qemu_configs/x86_64.py`, the invocation appear as

```
./tools/testing/kunit/kunit.py run \
        --timeout=60 \
        --jobs=12 \
        --qemu_config=./tools/testing/kunit/qemu_configs/x86_64.py
```

To run existing KUnit tests on non-UML architectures, see: Documentation/dev-tools/kunit/non_uml.rst.

15.3.6 Command-Line Arguments

kunit_tool has a number of other command-line arguments which can be useful for our test environment. Below the most commonly used command line arguments:

- `--help`: Lists all available options. To list common options, place `--help` before the command. To list options specific to that command, place `--help` after the command.

  Note: Different commands (`config`, `build`, `run`, etc) have different supported options.

- `--build_dir`: Specifies kunit_tool build directory. It includes the `.kunitconfig`, `.config` files and compiled kernel.

- `--make_options`: Specifies additional options to pass to make, when compiling a kernel (using `build` or `run` commands). For example: to enable compiler warnings, we can pass `--make_options W=1`.
- `--alltests`: Builds a UML kernel with all config options enabled using `make allyesconfig`. This allows us to run as many tests as possible.

  Note: It is slow and prone to breakage as new options are added or modified. Instead, enable all tests which have satisfied dependencies by adding `CONFIG_KUNIT_ALL_TESTS=y` to your `.kunitconfig`.

15.4 Run Tests without kunit_tool

If we do not want to use kunit_tool (For example: we want to integrate with other systems, or run tests on real hardware), we can include KUnit in any kernel, read out results, and parse manually.

Note: KUnit is not designed for use in a production system. It is possible that tests may reduce the stability or security of the system.

15.4.1 Configure the Kernel

KUnit tests can run without kunit_tool. This can be useful, if:

- We have an existing kernel configuration to test.
- Need to run on real hardware (or using an emulator/VM kunit_tool does not support).
- Wish to integrate with some existing testing systems.

KUnit is configured with the `CONFIG_KUNIT` option, and individual tests can also be built by enabling their config options in our `.config`. KUnit tests usually (but don't always) have config options ending in `_KUNIT_TEST`. Most tests can either be built as a module, or be built into the kernel.

Note: We can enable the `KUNIT_ALL_TESTS` config option to automatically enable all tests with satisfied dependencies. This is a good way of quickly testing everything applicable to the current config.

Once we have built our kernel (and/or modules), it is simple to run the tests. If the tests are built-in, they will run automatically on the kernel boot. The results will be written to the kernel log (`dmesg`) in TAP format.

If the tests are built as modules, they will run when the module is loaded.

```
# modprobe example-test
```

The results will appear in TAP format in dmesg.

Note: If CONFIG_KUNIT_DEBUGFS is enabled, KUnit test results will be accessible from the debugfs filesystem (if mounted). They will be in /sys/kernel/debug/kunit/<test_suite>/results, in TAP format.

15.5 Writing Tests

15.5.1 Test Cases

The fundamental unit in KUnit is the test case. A test case is a function with the signature void (*)(struct kunit *test). It calls the function under test and then sets *expectations* for what should happen. For example:

```
void example_test_success(struct kunit *test)
{
}

void example_test_failure(struct kunit *test)
{
        KUNIT_FAIL(test, "This test never passes.");
}
```

In the above example, example_test_success always passes because it does nothing; no expectations are set, and therefore all expectations pass. On the other hand example_test_failure always fails because it calls KUNIT_FAIL, which is a special expectation that logs a message and causes the test case to fail.

Expectations

An *expectation* specifies that we expect a piece of code to do something in a test. An expectation is called like a function. A test is made by setting expectations about the behavior of a piece of code under test. When one or more expectations fail, the test case fails and information about the failure is logged. For example:

```
void add_test_basic(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 1, add(1, 0));
        KUNIT_EXPECT_EQ(test, 2, add(1, 1));
}
```

In the above example, add_test_basic makes a number of assertions about the behavior of a function called add. The first parameter is always of type struct kunit *, which contains information about the current test context. The second parameter, in this case, is what the value is expected to be. The last value is what the value actually is. If add passes all of these expectations, the test case, add_test_basic will pass; if any one of these expectations fails, the test case will fail.

A test case *fails* when any expectation is violated; however, the test will continue to run, and try other expectations until the test case ends or is otherwise terminated. This is as opposed to *assertions* which are discussed later.

To learn about more KUnit expectations, see *Test API*.

Note: A single test case should be short, easy to understand, and focused on a single behavior.

For example, if we want to rigorously test the add function above, create additional tests cases which would test each property that an add function should have as shown below:

```
void add_test_basic(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 1, add(1, 0));
        KUNIT_EXPECT_EQ(test, 2, add(1, 1));
}

void add_test_negative(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 0, add(-1, 1));
}

void add_test_max(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, INT_MAX, add(0, INT_MAX));
        KUNIT_EXPECT_EQ(test, -1, add(INT_MAX, INT_MIN));
}

void add_test_overflow(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, INT_MIN, add(INT_MAX, 1));
}
```

Assertions

An assertion is like an expectation, except that the assertion immediately terminates the test case if the condition is not satisfied. For example:

```
static void test_sort(struct kunit *test)
{
        int *a, i, r = 1;
        a = kunit_kmalloc_array(test, TEST_LEN, sizeof(*a), GFP_KERNEL);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, a);
        for (i = 0; i < TEST_LEN; i++) {
                r = (r * 725861) % 6599;
                a[i] = r;
        }
        sort(a, TEST_LEN, sizeof(*a), cmpint, NULL);
        for (i = 0; i < TEST_LEN-1; i++)
```

```
                KUNIT_EXPECT_LE(test, a[i], a[i + 1]);
}
```

In this example, the method under test should return pointer to a value. If the pointer returns null or an errno, we want to stop the test since the following expectation could crash the test case. *ASSERT_NOT_ERR_OR_NULL(…)* allows us to bail out of the test case if the appropriate conditions are not satisfied to complete the test.

Test Suites

We need many test cases covering all the unit's behaviors. It is common to have many similar tests. In order to reduce duplication in these closely related tests, most unit testing frameworks (including KUnit) provide the concept of a *test suite*. A test suite is a collection of test cases for a unit of code with optional setup and teardown functions that run before/after the whole suite and/or every test case. For example:

```
static struct kunit_case example_test_cases[] = {
        KUNIT_CASE(example_test_foo),
        KUNIT_CASE(example_test_bar),
        KUNIT_CASE(example_test_baz),
        {}
};

static struct kunit_suite example_test_suite = {
        .name = "example",
        .init = example_test_init,
        .exit = example_test_exit,
        .suite_init = example_suite_init,
        .suite_exit = example_suite_exit,
        .test_cases = example_test_cases,
};
kunit_test_suite(example_test_suite);
```

In the above example, the test suite `example_test_suite` would first run `example_suite_init`, then run the test cases `example_test_foo`, `example_test_bar`, and `example_test_baz`. Each would have `example_test_init` called immediately before it and `example_test_exit` called immediately after it. Finally, `example_suite_exit` would be called after everything else. `kunit_test_suite(example_test_suite)` registers the test suite with the KUnit test framework.

Note: A test case will only run if it is associated with a test suite.

`kunit_test_suite(...)` is a macro which tells the linker to put the specified test suite in a special linker section so that it can be run by KUnit either after `late_init`, or when the test module is loaded (if the test was built as a module).

For more information, see *Test API*.

15.5.2 Writing Tests For Other Architectures

It is better to write tests that run on UML to tests that only run under a particular architecture. It is better to write tests that run under QEMU or another easy to obtain (and monetarily free) software environment to a specific piece of hardware.

Nevertheless, there are still valid reasons to write a test that is architecture or hardware specific. For example, we might want to test code that really belongs in `arch/some-arch/*`. Even so, try to write the test so that it does not depend on physical hardware. Some of our test cases may not need hardware, only few tests actually require the hardware to test it. When hardware is not available, instead of disabling tests, we can skip them.

Now that we have narrowed down exactly what bits are hardware specific, the actual procedure for writing and running the tests is same as writing normal KUnit tests.

Important: We may have to reset hardware state. If this is not possible, we may only be able to run one test case per invocation.

15.6 Common Patterns

15.6.1 Isolating Behavior

Unit testing limits the amount of code under test to a single unit. It controls what code gets run when the unit under test calls a function. Where a function is exposed as part of an API such that the definition of that function can be changed without affecting the rest of the code base. In the kernel, this comes from two constructs: classes, which are structs that contain function pointers provided by the implementer, and architecture-specific functions, which have definitions selected at compile time.

Classes

Classes are not a construct that is built into the C programming language; however, it is an easily derived concept. Accordingly, in most cases, every project that does not use a standardized object oriented library (like GNOME's GObject) has their own slightly different way of doing object oriented programming; the Linux kernel is no exception.

The central concept in kernel object oriented programming is the class. In the kernel, a *class* is a struct that contains function pointers. This creates a contract between *implementers* and *users* since it forces them to use the same function signature without having to call the function directly. To be a class, the function pointers must specify that a pointer to the class, known as a *class handle*, be one of the parameters. Thus the member functions (also known as *methods*) have access to member variables (also known as *fields*) allowing the same implementation to have multiple *instances*.

A class can be *overridden* by *child classes* by embedding the *parent class* in the child class. Then when the child class *method* is called, the child implementation knows that the pointer passed to it is of a parent contained within the child. Thus, the child can compute the pointer to itself because the pointer to the parent is always a fixed offset from the pointer to the child. This offset is the offset of the parent contained in the child struct. For example:

```
struct shape {
        int (*area)(struct shape *this);
};

struct rectangle {
        struct shape parent;
        int length;
        int width;
};

int rectangle_area(struct shape *this)
{
        struct rectangle *self = container_of(this, struct rectangle, parent);

        return self->length * self->width;
};

void rectangle_new(struct rectangle *self, int length, int width)
{
        self->parent.area = rectangle_area;
        self->length = length;
        self->width = width;
}
```

In this example, computing the pointer to the child from the pointer to the parent is done by container_of.

Faking Classes

In order to unit test a piece of code that calls a method in a class, the behavior of the method must be controllable, otherwise the test ceases to be a unit test and becomes an integration test.

A fake class implements a piece of code that is different than what runs in a production instance, but behaves identical from the standpoint of the callers. This is done to replace a dependency that is hard to deal with, or is slow. For example, implementing a fake EEPROM that stores the "contents" in an internal buffer. Assume we have a class that represents an EEPROM:

```
struct eeprom {
        ssize_t (*read)(struct eeprom *this, size_t offset, char *buffer, size_
↪t count);
        ssize_t (*write)(struct eeprom *this, size_t offset, const char␣
↪*buffer, size_t count);
};
```

And we want to test code that buffers writes to the EEPROM:

```
struct eeprom_buffer {
        ssize_t (*write)(struct eeprom_buffer *this, const char *buffer, size_
↪t count);
        int flush(struct eeprom_buffer *this);
        size_t flush_count; /* Flushes when buffer exceeds flush_count. */
};

struct eeprom_buffer *new_eeprom_buffer(struct eeprom *eeprom);
void destroy_eeprom_buffer(struct eeprom *eeprom);
```

We can test this code by *faking out* the underlying EEPROM:

```
struct fake_eeprom {
        struct eeprom parent;
        char contents[FAKE_EEPROM_CONTENTS_SIZE];
};

ssize_t fake_eeprom_read(struct eeprom *parent, size_t offset, char *buffer,␣
↪size_t count)
{
        struct fake_eeprom *this = container_of(parent, struct fake_eeprom,␣
↪parent);

        count = min(count, FAKE_EEPROM_CONTENTS_SIZE - offset);
        memcpy(buffer, this->contents + offset, count);

        return count;
}

ssize_t fake_eeprom_write(struct eeprom *parent, size_t offset, const char␣
↪*buffer, size_t count)
{
        struct fake_eeprom *this = container_of(parent, struct fake_eeprom,␣
↪parent);

        count = min(count, FAKE_EEPROM_CONTENTS_SIZE - offset);
        memcpy(this->contents + offset, buffer, count);

        return count;
}

void fake_eeprom_init(struct fake_eeprom *this)
{
        this->parent.read = fake_eeprom_read;
        this->parent.write = fake_eeprom_write;
        memset(this->contents, 0, FAKE_EEPROM_CONTENTS_SIZE);
}
```

We can now use it to test `struct eeprom_buffer`:

```
struct eeprom_buffer_test {
        struct fake_eeprom *fake_eeprom;
        struct eeprom_buffer *eeprom_buffer;
};
```

```
static void eeprom_buffer_test_does_not_write_until_flush(struct kunit *test)
{
        struct eeprom_buffer_test *ctx = test->priv;
        struct eeprom_buffer *eeprom_buffer = ctx->eeprom_buffer;
        struct fake_eeprom *fake_eeprom = ctx->fake_eeprom;
        char buffer[] = {0xff};

        eeprom_buffer->flush_count = SIZE_MAX;

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0);

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0);

        eeprom_buffer->flush(eeprom_buffer);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0xff);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0xff);
}

static void eeprom_buffer_test_flushes_after_flush_count_met(struct kunit␣
 ↪*test)
{
        struct eeprom_buffer_test *ctx = test->priv;
        struct eeprom_buffer *eeprom_buffer = ctx->eeprom_buffer;
        struct fake_eeprom *fake_eeprom = ctx->fake_eeprom;
        char buffer[] = {0xff};

        eeprom_buffer->flush_count = 2;

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0);

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0xff);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0xff);
}

static void eeprom_buffer_test_flushes_increments_of_flush_count(struct kunit␣
 ↪*test)
{
        struct eeprom_buffer_test *ctx = test->priv;
        struct eeprom_buffer *eeprom_buffer = ctx->eeprom_buffer;
        struct fake_eeprom *fake_eeprom = ctx->fake_eeprom;
        char buffer[] = {0xff, 0xff};

        eeprom_buffer->flush_count = 2;

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0);
        eeprom_buffer->write(eeprom_buffer, buffer, 2);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0xff);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0xff);
        /* Should have only flushed the first two bytes. */
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[2], 0);
}

static int eeprom_buffer_test_init(struct kunit *test)
{
        struct eeprom_buffer_test *ctx;

        ctx = kunit_kzalloc(test, sizeof(*ctx), GFP_KERNEL);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ctx);

        ctx->fake_eeprom = kunit_kzalloc(test, sizeof(*ctx->fake_eeprom), GFP_
↪KERNEL);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ctx->fake_eeprom);
        fake_eeprom_init(ctx->fake_eeprom);

        ctx->eeprom_buffer = new_eeprom_buffer(&ctx->fake_eeprom->parent);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ctx->eeprom_buffer);

        test->priv = ctx;

        return 0;
}

static void eeprom_buffer_test_exit(struct kunit *test)
{
        struct eeprom_buffer_test *ctx = test->priv;

        destroy_eeprom_buffer(ctx->eeprom_buffer);
}
```

15.6.2 Testing Against Multiple Inputs

Testing just a few inputs is not enough to ensure that the code works correctly, for example: testing a hash function.

We can write a helper macro or function. The function is called for each input. For example, to test sha1sum(1), we can write:

```
#define TEST_SHA1(in, want) \
        sha1sum(in, out); \
        KUNIT_EXPECT_STREQ_MSG(test, out, want, "sha1sum(%s)", in);

char out[40];
TEST_SHA1("hello world",  "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed");
TEST_SHA1("hello world!", "430ce34d020724ed75a196dfc2ad67c77772d169");
```

Note the use of the _MSG version of KUNIT_EXPECT_STREQ to print a more detailed error and make the assertions clearer within the helper macros.

The _MSG variants are useful when the same expectation is called multiple times (in a loop or helper function) and thus the line number is not enough to identify what failed, as shown below.

In complicated cases, we recommend using a *table-driven test* compared to the helper macro variation, for example:

```
int i;
char out[40];

struct sha1_test_case {
        const char *str;
        const char *sha1;
};

struct sha1_test_case cases[] = {
        {
                .str = "hello world",
                .sha1 = "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed",
        },
        {
                .str = "hello world!",
                .sha1 = "430ce34d020724ed75a196dfc2ad67c77772d169",
        },
};
for (i = 0; i < ARRAY_SIZE(cases); ++i) {
        sha1sum(cases[i].str, out);
        KUNIT_EXPECT_STREQ_MSG(test, out, cases[i].sha1,
                              "sha1sum(%s)", cases[i].str);
}
```

There is more boilerplate code involved, but it can:

- be more readable when there are multiple inputs/outputs (due to field names).
  - For example, see fs/ext4/inode-test.c.
- reduce duplication if test cases are shared across multiple tests.
  - For example: if we want to test sha256sum, we could add a sha256 field and reuse cases.
- be converted to a “parameterized test”.

Parameterized Testing

The table-driven testing pattern is common enough that KUnit has special support for it.

By reusing the same cases array from above, we can write the test as a “parameterized test” with the following.

```
// This is copy-pasted from above.
struct sha1_test_case {
        const char *str;
        const char *sha1;
};
struct sha1_test_case cases[] = {
        {
                .str = "hello world",
                .sha1 = "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed",
        },
        {
                .str = "hello world!",
                .sha1 = "430ce34d020724ed75a196dfc2ad67c77772d169",
        },
};

// Need a helper function to generate a name for each test case.
static void case_to_desc(const struct sha1_test_case *t, char *desc)
{
        strcpy(desc, t->str);
}
// Creates `sha1_gen_params()` to iterate over `cases`.
KUNIT_ARRAY_PARAM(sha1, cases, case_to_desc);

// Looks no different from a normal test.
static void sha1_test(struct kunit *test)
{
        // This function can just contain the body of the for-loop.
        // The former `cases[i]` is accessible under test->param_value.
        char out[40];
        struct sha1_test_case *test_param = (struct sha1_test_case *)(test->
↪param_value);

        sha1sum(test_param->str, out);
        KUNIT_EXPECT_STREQ_MSG(test, out, test_param->sha1,
                              "sha1sum(%s)", test_param->str);
}

// Instead of KUNIT_CASE, we use KUNIT_CASE_PARAM and pass in the
// function declared by KUNIT_ARRAY_PARAM.
static struct kunit_case sha1_test_cases[] = {
        KUNIT_CASE_PARAM(sha1_test, sha1_gen_params),
        {}
};
```

15.6.3 Exiting Early on Failed Expectations

We can use KUNIT_EXPECT_EQ to mark the test as failed and continue execution. In some cases, it is unsafe to continue. We can use the KUNIT_ASSERT variant to exit on failure.

```
void example_test_user_alloc_function(struct kunit *test)
{
        void *object = alloc_some_object_for_me();

        /* Make sure we got a valid pointer back. */
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, object);
        do_something_with_object(object);
}
```

15.6.4 Allocating Memory

Where you might use kzalloc, you can instead use kunit_kzalloc as KUnit will then ensure that the memory is freed once the test completes.

This is useful because it lets us use the KUNIT_ASSERT_EQ macros to exit early from a test without having to worry about remembering to call kfree. For example:

```
void example_test_allocation(struct kunit *test)
{
        char *buffer = kunit_kzalloc(test, 16, GFP_KERNEL);
        /* Ensure allocation succeeded. */
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, buffer);

        KUNIT_ASSERT_STREQ(test, buffer, "");
}
```

15.6.5 Testing Static Functions

If we do not want to expose functions or variables for testing, one option is to conditionally #include the test file at the end of your .c file. For example:

```
/* In my_file.c */

static int do_interesting_thing();

#ifdef CONFIG_MY_KUNIT_TEST
#include "my_kunit_test.c"
#endif
```

15.6.6 Injecting Test-Only Code

Similar to as shown above, we can add test-specific logic. For example:

```
/* In my_file.h */

#ifdef CONFIG_MY_KUNIT_TEST
/* Defined in my_kunit_test.c */
void test_only_hook(void);
#else
void test_only_hook(void) { }
#endif
```

This test-only code can be made more useful by accessing the current `kunit_test` as shown in next section: *Accessing The Current Test*.

15.6.7 Accessing The Current Test

In some cases, we need to call test-only code from outside the test file. For example, see example in section *Injecting Test-Only Code* or if we are providing a fake implementation of an ops struct. Using `kunit_test` field in `task_struct`, we can access it via `current->kunit_test`.

The example below includes how to implement "mocking":

```
#include <linux/sched.h> /* for current */

struct test_data {
        int foo_result;
        int want_foo_called_with;
};

static int fake_foo(int arg)
{
        struct kunit *test = current->kunit_test;
        struct test_data *test_data = test->priv;

        KUNIT_EXPECT_EQ(test, test_data->want_foo_called_with, arg);
        return test_data->foo_result;
}

static void example_simple_test(struct kunit *test)
{
        /* Assume priv (private, a member used to pass test data from
         * the init function) is allocated in the suite's .init */
        struct test_data *test_data = test->priv;

        test_data->foo_result = 42;
        test_data->want_foo_called_with = 1;

        /* In a real test, we'd probably pass a pointer to fake_foo somewhere
         * like an ops struct, etc. instead of calling it directly. */
        KUNIT_EXPECT_EQ(test, fake_foo(1), 42);
}
```

In this example, we are using the priv member of struct kunit as a way of passing data to the test from the init function. In general priv is pointer that can be used for any user data. This is preferred over static variables, as it avoids concurrency issues.

Had we wanted something more flexible, we could have used a named kunit_resource. Each test can have multiple resources which have string names providing the same flexibility as a priv member, but also, for example, allowing helper functions to create resources without conflicting with each other. It is also possible to define a clean up function for each resource, making it easy to avoid resource leaks. For more information, see *Test API*.

15.6.8 Failing The Current Test

If we want to fail the current test, we can use kunit_fail_current_test(fmt, args...) which is defined in <kunit/test-bug.h> and does not require pulling in <kunit/test.h>. For example, we have an option to enable some extra debug checks on some data structures as shown below:

```
#include <kunit/test-bug.h>

#ifdef CONFIG_EXTRA_DEBUG_CHECKS
static void validate_my_data(struct data *data)
{
        if (is_valid(data))
                return;

        kunit_fail_current_test("data %p is invalid", data);

        /* Normal, non-KUnit, error reporting code here. */
}
#else
static void my_debug_function(void) { }
#endif
```

15.7 kunit_tool How-To

15.7.1 What is kunit_tool?

kunit_tool is a script (tools/testing/kunit/kunit.py) that aids in building the Linux kernel as UML (User Mode Linux), running KUnit tests, parsing the test results and displaying them in a user friendly manner.

kunit_tool addresses the problem of being able to run tests without needing a virtual machine or actual hardware with User Mode Linux. User Mode Linux is a Linux architecture, like ARM or x86; however, unlike other architectures it compiles the kernel as a standalone Linux executable that can be run like any other program directly inside of a host operating system. To be clear, it does not require any virtualization support: it is just a regular program.

15.7.2 What is a .kunitconfig?

It's just a defconfig that kunit_tool looks for in the build directory (`.kunit` by default). kunit_tool uses it to generate a .config as you might expect. In addition, it verifies that the generated .config contains the CONFIG options in the .kunitconfig; the reason it does this is so that it is easy to be sure that a CONFIG that enables a test actually ends up in the .config.

It's also possible to pass a separate .kunitconfig fragment to kunit_tool, which is useful if you have several different groups of tests you wish to run independently, or if you want to use pre-defined test configs for certain subsystems.

15.7.3 Getting Started with kunit_tool

If a kunitconfig is present at the root directory, all you have to do is:

```
./tools/testing/kunit/kunit.py run
```

However, you most likely want to use it with the following options:

```
./tools/testing/kunit/kunit.py run --timeout=30 --jobs=`nproc --all`
```

- `--timeout` sets a maximum amount of time to allow tests to run.
- `--jobs` sets the number of threads to use to build the kernel.

Note: This command will work even without a .kunitconfig file: if no .kunitconfig is present, a default one will be used instead.

If you wish to use a different .kunitconfig file (such as one provided for testing a particular subsystem), you can pass it as an option.

```
./tools/testing/kunit/kunit.py run --kunitconfig=fs/ext4/.kunitconfig
```

For a list of all the flags supported by kunit_tool, you can run:

```
./tools/testing/kunit/kunit.py run --help
```

15.7.4 Configuring, Building, and Running Tests

It's also possible to run just parts of the KUnit build process independently, which is useful if you want to make manual changes to part of the process.

A .config can be generated from a .kunitconfig by using the `config` argument when running kunit_tool:

```
./tools/testing/kunit/kunit.py config
```

Similarly, if you just want to build a KUnit kernel from the current .config, you can use the `build` argument:

```
./tools/testing/kunit/kunit.py build
```

And, if you already have a built UML kernel with built-in KUnit tests, you can run the kernel and display the test results with the `exec` argument:

```
./tools/testing/kunit/kunit.py exec
```

The `run` command which is discussed above is equivalent to running all three of these in sequence.

All of these commands accept a number of optional command-line arguments. The `--help` flag will give a complete list of these, or keep reading this page for a guide to some of the more useful ones.

15.7.5 Parsing Test Results

KUnit tests output their results in TAP (Test Anything Protocol) format. kunit_tool will, when running tests, parse this output and print a summary which is much more pleasant to read. If you wish to look at the raw test results in TAP format, you can pass the `--raw_output` argument.

```
./tools/testing/kunit/kunit.py run --raw_output
```

The raw output from test runs may contain other, non-KUnit kernel log lines. You can see just KUnit output with `--raw_output=kunit`:

```
./tools/testing/kunit/kunit.py run --raw_output=kunit
```

If you have KUnit results in their raw TAP format, you can parse them and print the human-readable summary with the `parse` command for kunit_tool. This accepts a filename for an argument, or will read from standard input.

```
# Reading from a file
./tools/testing/kunit/kunit.py parse /var/log/dmesg
# Reading from stdin
dmesg | ./tools/testing/kunit/kunit.py parse
```

This is very useful if you wish to run tests in a configuration not supported by kunit_tool (such as on real hardware, or an unsupported architecture).

15.7.6 Filtering Tests

It's possible to run only a subset of the tests built into a kernel by passing a filter to the `exec` or `run` commands. For example, if you only wanted to run KUnit resource tests, you could use:

```
./tools/testing/kunit/kunit.py run 'kunit-resource*'
```

This uses the standard glob format for wildcards.

15.7.7 Running Tests on QEMU

kunit_tool supports running tests on QEMU as well as via UML (as mentioned elsewhere). The default way of running tests on QEMU requires two flags:

`--arch` Selects a collection of configs (Kconfig as well as QEMU configs options, etc) that allow KUnit tests to be run on the specified architecture in a minimal way; this is usually not much slower than using UML. The architecture argument is the same as the name of the option passed to the `ARCH` variable used by Kbuild. Not all architectures are currently supported by this flag, but can be handled by the `--qemu_config` discussed later. If `um` is passed (or this this flag is ignored) the tests will run via UML. Non-UML architectures, e.g. i386, x86_64, arm, um, etc. Non-UML run on QEMU.

`--cross_compile` Specifies the use of a toolchain by Kbuild. The argument passed here is the same passed to the `CROSS_COMPILE` variable used by Kbuild. As a reminder this will be the prefix for the toolchain binaries such as gcc for example `sparc64-linux-gnu-` if you have the sparc toolchain installed on your system, or `$HOME/toolchains/microblaze/gcc-9.2.0-nolibc/microblaze-linux/bin/microblaze-linux-` if you have downloaded the microblaze toolchain from the 0-day website to a directory in your home directory called `toolchains`.

In many cases it is likely that you may want to run an architecture which is not supported by the `--arch` flag, or you may want to just run KUnit tests on QEMU using a non-default configuration. For this use case, you can write your own QemuConfig. These QemuConfigs are written in Python. They must have an import line `from ..qemu_config import QemuArchParams` at the top of the file and the file must contain a variable called `QEMU_ARCH` that has an instance of `QemuArchParams` assigned to it. An example can be seen in `tools/testing/kunit/qemu_configs/x86_64.py`.

Once you have a QemuConfig you can pass it into kunit_tool using the `--qemu_config` flag; when used this flag replaces the `--arch` flag. If we were to do this with the `x86_64.py` example from above, the invocation would look something like this:

```
./tools/testing/kunit/kunit.py run \
        --timeout=60 \
        --jobs=12 \
        --qemu_config=./tools/testing/kunit/qemu_configs/x86_64.py
```

15.7.8 Other Useful Options

kunit_tool has a number of other command-line arguments which can be useful when adapting it to fit your environment or needs.

Some of the more useful ones are:

`--help` Lists all of the available options. Note that different commands (`config`, `build`, `run`, etc) will have different supported options. Place `--help` before the command to list common options, and after the command for options specific to that command.

`--build_dir` Specifies the build directory that kunit_tool will use. This is where the .kunitconfig file is located, as well as where the .config and compiled kernel will be placed. Defaults to `.kunit`.

--make_options Specifies additional options to pass to make when compiling a kernel (with the build or run commands). For example, to enable compiler warnings, you can pass `--make_options W=1`.

--alltests Builds a UML kernel with all config options enabled using `make allyesconfig`. This allows you to run as many tests as is possible, but is very slow and prone to breakage as new options are added or modified. In most cases, enabling all tests which have satisfied dependencies by adding `CONFIG_KUNIT_ALL_TESTS=1` to your .kunitconfig is preferable.

There are several other options (and new ones are often added), so do check `--help` if you're looking for something not mentioned here.

15.8 API Reference

15.8.1 Test API

This file documents all of the standard testing API.

enum **kunit_status**
Type of result for a test or test suite

Constants

KUNIT_SUCCESS Denotes the test suite has not failed nor been skipped

KUNIT_FAILURE Denotes the test has failed.

KUNIT_SKIPPED Denotes the test has been skipped.

struct **kunit_case**
represents an individual test case.

Definition

```
struct kunit_case {
  void (*run_case)(struct kunit *test);
  const char *name;
  const void* (*generate_params)(const void *prev, char *desc);
};
```

Members

run_case the function representing the actual test case.

name the name of the test case.

generate_params the generator function for parameterized tests.

Description

A test case is a function with the signature, `void (*)(struct kunit *)` that makes expectations and assertions (see *KUNIT_EXPECT_TRUE()* and *KUNIT_ASSERT_TRUE()*) about code under test. Each test case is associated with a *struct kunit_suite* and will be run after the suite's init function and followed by the suite's exit function.

A test case should be static and should only be created with the *KUNIT_CASE()* macro; additionally, every array of test cases should be terminated with an empty test case.

```
void add_test_basic(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 1, add(1, 0));
        KUNIT_EXPECT_EQ(test, 2, add(1, 1));
        KUNIT_EXPECT_EQ(test, 0, add(-1, 1));
        KUNIT_EXPECT_EQ(test, INT_MAX, add(0, INT_MAX));
        KUNIT_EXPECT_EQ(test, -1, add(INT_MAX, INT_MIN));
}

static struct kunit_case example_test_cases[] = {
        KUNIT_CASE(add_test_basic),
        {}
};
```

Example

KUNIT_CASE

KUNIT_CASE (test_name)

A helper for creating a *struct kunit_case*

Parameters

test_name a reference to a test case function.

Description

Takes a symbol for a function representing a test case and creates a *struct kunit_case* object from it. See the documentation for *struct kunit_case* for an example on how to use it.

KUNIT_CASE_PARAM

KUNIT_CASE_PARAM (test_name, gen_params)

A helper for creation a parameterized *struct kunit_case*

Parameters

test_name a reference to a test case function.

gen_params a reference to a parameter generator function.

Description

The generator function:

```
const void* gen_params(const void *prev, char *desc)
```

is used to lazily generate a series of arbitrarily typed values that fit into a void*. The argument **prev** is the previously returned value, which should be used to derive the next value; **prev** is set to NULL on the initial generator call. When no more values are available, the generator must return NULL. Optionally write a string into **desc** (size of KUNIT_PARAM_DESC_SIZE) describing the parameter.

struct **kunit_suite**

describes a related collection of *struct kunit_case*

Definition

```
struct kunit_suite {
  const char name[256];
  int (*suite_init)(struct kunit_suite *suite);
  void (*suite_exit)(struct kunit_suite *suite);
  int (*init)(struct kunit *test);
  void (*exit)(struct kunit *test);
  struct kunit_case *test_cases;
};
```

Members

name the name of the test. Purely informational.

suite_init called once per test suite before the test cases.

suite_exit called once per test suite after all test cases.

init called before every test case.

exit called after every test case.

test_cases a null terminated array of test cases.

Description

A kunit_suite is a collection of related *struct kunit_case* s, such that **init** is called before every test case and **exit** is called after every test case, similar to the notion of a *test fixture* or a *test class* in other unit testing frameworks like JUnit or Googletest.

Every *struct kunit_case* must be associated with a kunit_suite for KUnit to run it.

struct **kunit**

represents a running instance of a test.

Definition

```
struct kunit {
  void *priv;
};
```

Members

priv for user to store arbitrary data. Commonly used to pass data created in the init function (see *struct kunit_suite*).

Description

Used to store information about the current context under which the test is running. Most of this data is private and should only be accessed indirectly via public functions; the one exception is **priv** which can be used by the test writer to store arbitrary data.

kunit_test_suites_for_module

`kunit_test_suites_for_module (__suites)`

used to register one or more *struct kunit_suite* with KUnit.

Parameters

__suites a statically allocated list of *struct kunit_suite*.

Description

Registers **__suites** with the test framework. See *struct kunit_suite* for more information.

If a test suite is built-in, module_init() gets translated into an initcall which we don't want as the idea is that for builtins the executor will manage execution. So ensure we do not define module_{init|exit} functions for the builtin case when registering suites via *kunit_test_suites()* below.

kunit_test_suites

`kunit_test_suites (__suites...)`

used to register one or more *struct kunit_suite* with KUnit.

Parameters

__suites... a statically allocated list of *struct kunit_suite*.

Description

Registers **suites** with the test framework. See *struct kunit_suite* for more information.

When builtin, KUnit tests are all run via executor; this is done by placing the array of *struct kunit_suite* * in the .kunit_test_suites ELF section.

An alternative is to build the tests as a module. Because modules do not support multiple initcall()s, we need to initialize an array of suites for a module.

kunit_test_init_section_suites

`kunit_test_init_section_suites (__suites...)`

used to register one or more *struct kunit_suite* containing init functions or init data.

Parameters

__suites... a statically allocated list of *struct kunit_suite*.

Description

This functions identically as :c:type:`kunit_test_suites`() except that it suppresses modpost warnings for referencing functions marked __init or data marked __initdata; this is OK because currently KUnit only runs tests upon boot during the init phase or upon loading a module during the init phase.

NOTE TO KUNIT DEVS: If we ever allow KUnit tests to be run after boot, these tests must be excluded.

The only thing this macro does that's different from kunit_test_suites is that it suffixes the array and suite declarations it makes with _probe; modpost suppresses warnings about referencing init data for symbols named in this manner.

void ***kunit_kmalloc_array**(struct *kunit* *test, size_t n, size_t size, gfp_t gfp)

Like kmalloc_array() except the allocation is *test managed*.

Parameters

struct kunit *test The test context object.

size_t n number of elements.

size_t size The size in bytes of the desired memory.

gfp_t gfp flags passed to underlying kmalloc().

Description

Just like *kmalloc_array(...)*, except the allocation is managed by the test case and is automatically cleaned up after the test case concludes. See *struct kunit_resource* for more information.

void ***kunit_kmalloc**(struct *kunit* *test, size_t size, gfp_t gfp)
Like kmalloc() except the allocation is *test managed*.

Parameters

struct kunit *test The test context object.

size_t size The size in bytes of the desired memory.

gfp_t gfp flags passed to underlying kmalloc().

Description

See kmalloc() and *kunit_kmalloc_array()* for more information.

void **kunit_kfree**(struct *kunit* *test, const void *ptr)
Like kfree except for allocations managed by KUnit.

Parameters

struct kunit *test The test case to which the resource belongs.

const void *ptr The memory allocation to free.

void ***kunit_kzalloc**(struct *kunit* *test, size_t size, gfp_t gfp)
Just like *kunit_kmalloc()*, but zeroes the allocation.

Parameters

struct kunit *test The test context object.

size_t size The size in bytes of the desired memory.

gfp_t gfp flags passed to underlying kmalloc().

Description

See kzalloc() and *kunit_kmalloc_array()* for more information.

void ***kunit_kcalloc**(struct *kunit* *test, size_t n, size_t size, gfp_t gfp)
Just like *kunit_kmalloc_array()*, but zeroes the allocation.

Parameters

struct kunit *test The test context object.

size_t n number of elements.

size_t size The size in bytes of the desired memory.

gfp_t gfp flags passed to underlying kmalloc().

Description

See kcalloc() and *kunit_kmalloc_array()* for more information.

kunit_mark_skipped

`kunit_mark_skipped (test_or_suite, fmt, ...)`

Marks **test_or_suite** as skipped

Parameters

`test_or_suite` The test context object.

`fmt` A printk() style format string.

`...` variable arguments

Description

Marks the test as skipped. **fmt** is given output as the test status comment, typically the reason the test was skipped.

Test execution continues after *`kunit_mark_skipped()`* is called.

`kunit_skip`

`kunit_skip (test_or_suite, fmt, ...)`

Marks **test_or_suite** as skipped

Parameters

`test_or_suite` The test context object.

`fmt` A printk() style format string.

`...` variable arguments

Description

Skips the test. **fmt** is given output as the test status comment, typically the reason the test was skipped.

Test execution is halted after *`kunit_skip()`* is called.

`kunit_info`

`kunit_info (test, fmt, ...)`

Prints an INFO level message associated with **test**.

Parameters

`test` The test context object.

`fmt` A printk() style format string.

`...` variable arguments

Description

Prints an info level message associated with the test suite being run. Takes a variable number of format parameters just like printk().

`kunit_warn`

`kunit_warn (test, fmt, ...)`

Prints a WARN level message associated with **test**.

Parameters

test The test context object.

fmt A printk() style format string.

... variable arguments

Description

Prints a warning level message.

kunit_err

`kunit_err (test, fmt, ...)`

Prints an ERROR level message associated with **test**.

Parameters

test The test context object.

fmt A printk() style format string.

... variable arguments

Description

Prints an error level message.

KUNIT_SUCCEED

`KUNIT_SUCCEED (test)`

A no-op expectation. Only exists for code clarity.

Parameters

test The test context object.

Description

The opposite of *KUNIT_FAIL()*, it is an expectation that cannot fail. In other words, it does nothing and only exists for code clarity. See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_FAIL

`KUNIT_FAIL (test, fmt, ...)`

Always causes a test to fail when evaluated.

Parameters

test The test context object.

fmt an informational message to be printed when the assertion is made.

... string format arguments.

Description

The opposite of *KUNIT_SUCCEED()*, it is an expectation that always fails. In other words, it always results in a failed expectation, and consequently always causes the test case to fail when evaluated. See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_TRUE

`KUNIT_EXPECT_TRUE (test, condition)`

Causes a test failure when the expression is not true.

Parameters

`test` The test context object.

`condition` an arbitrary boolean expression. The test fails when this does not evaluate to true.

Description

This and expectations of the form *KUNIT_EXPECT_** will cause the test case to fail when the specified condition is not met; however, it will not prevent the test case from continuing to run; this is otherwise known as an *expectation failure*.

`KUNIT_EXPECT_FALSE`

`KUNIT_EXPECT_FALSE (test, condition)`

Makes a test failure when the expression is not false.

Parameters

`test` The test context object.

`condition` an arbitrary boolean expression. The test fails when this does not evaluate to false.

Description

Sets an expectation that **condition** evaluates to false. See *`KUNIT_EXPECT_TRUE()`* for more information.

`KUNIT_EXPECT_EQ`

`KUNIT_EXPECT_EQ (test, left, right)`

Sets an expectation that **left** and **right** are equal.

Parameters

`test` The test context object.

`left` an arbitrary expression that evaluates to a primitive C type.

`right` an arbitrary expression that evaluates to a primitive C type.

Description

Sets an expectation that the values that **left** and **right** evaluate to are equal. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) == (**right**)). See *`KUNIT_EXPECT_TRUE()`* for more information.

`KUNIT_EXPECT_PTR_EQ`

`KUNIT_EXPECT_PTR_EQ (test, left, right)`

Expects that pointers **left** and **right** are equal.

Parameters

`test` The test context object.

`left` an arbitrary expression that evaluates to a pointer.

`right` an arbitrary expression that evaluates to a pointer.

Description

Sets an expectation that the values that **left** and **right** evaluate to are equal. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) == (**right**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_NE

`KUNIT_EXPECT_NE (test, left, right)`

An expectation that **left** and **right** are not equal.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an expectation that the values that **left** and **right** evaluate to are not equal. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) != (**right**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_PTR_NE

`KUNIT_EXPECT_PTR_NE (test, left, right)`

Expects that pointers **left** and **right** are not equal.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a pointer.

right an arbitrary expression that evaluates to a pointer.

Description

Sets an expectation that the values that **left** and **right** evaluate to are not equal. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) != (**right**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_LT

`KUNIT_EXPECT_LT (test, left, right)`

An expectation that **left** is less than **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an expectation that the value that **left** evaluates to is less than the value that **right** evaluates to. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) < (**right**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_LE

`KUNIT_EXPECT_LE (test, left, right)`

Expects that **left** is less than or equal to **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an expectation that the value that **left** evaluates to is less than or equal to the value that **right** evaluates to. Semantically this is equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) <= (**right**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_GT

`KUNIT_EXPECT_GT (test, left, right)`

An expectation that **left** is greater than **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an expectation that the value that **left** evaluates to is greater than the value that **right** evaluates to. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) > (**right**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_GE

`KUNIT_EXPECT_GE (test, left, right)`

Expects that **left** is greater than or equal to **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an expectation that the value that **left** evaluates to is greater than the value that **right** evaluates to. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, (**left**) >= (**right**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_STREQ

`KUNIT_EXPECT_STREQ (test, left, right)`

Expects that strings **left** and **right** are equal.

Parameters

`test` The test context object.

`left` an arbitrary expression that evaluates to a null terminated string.

`right` an arbitrary expression that evaluates to a null terminated string.

Description

Sets an expectation that the values that **left** and **right** evaluate to are equal. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, !strcmp((**left**), (**right**))). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_STRNEQ

`KUNIT_EXPECT_STRNEQ (test, left, right)`

Expects that strings **left** and **right** are not equal.

Parameters

`test` The test context object.

`left` an arbitrary expression that evaluates to a null terminated string.

`right` an arbitrary expression that evaluates to a null terminated string.

Description

Sets an expectation that the values that **left** and **right** evaluate to are not equal. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, strcmp((**left**), (**right**))). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_NULL

`KUNIT_EXPECT_NULL (test, ptr)`

Expects that **ptr** is null.

Parameters

`test` The test context object.

`ptr` an arbitrary pointer.

Description

Sets an expectation that the value that **ptr** evaluates to is null. This is semantically equivalent to KUNIT_EXPECT_PTR_EQ(**test**, ptr, NULL). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_NOT_NULL

`KUNIT_EXPECT_NOT_NULL (test, ptr)`

Expects that **ptr** is not null.

Parameters

`test` The test context object.

`ptr` an arbitrary pointer.

Description

Sets an expectation that the value that **ptr** evaluates to is not null. This is semantically equivalent to KUNIT_EXPECT_PTR_NE(**test**, ptr, NULL). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_EXPECT_NOT_ERR_OR_NULL

`KUNIT_EXPECT_NOT_ERR_OR_NULL (test, ptr)`

Expects that **ptr** is not null and not err.

Parameters

test The test context object.

ptr an arbitrary pointer.

Description

Sets an expectation that the value that **ptr** evaluates to is not null and not an errno stored in a pointer. This is semantically equivalent to KUNIT_EXPECT_TRUE(**test**, !IS_ERR_OR_NULL(**ptr**)). See *KUNIT_EXPECT_TRUE()* for more information.

KUNIT_ASSERT_TRUE

`KUNIT_ASSERT_TRUE (test, condition)`

Sets an assertion that **condition** is true.

Parameters

test The test context object.

condition an arbitrary boolean expression. The test fails and aborts when this does not evaluate to true.

Description

This and assertions of the form *KUNIT_ASSERT_** will cause the test case to fail *and immediately abort* when the specified condition is not met. Unlike an expectation failure, it will prevent the test case from continuing to run; this is otherwise known as an *assertion failure*.

KUNIT_ASSERT_FALSE

`KUNIT_ASSERT_FALSE (test, condition)`

Sets an assertion that **condition** is false.

Parameters

test The test context object.

condition an arbitrary boolean expression.

Description

Sets an assertion that the value that **condition** evaluates to is false. This is the same as *KUNIT_EXPECT_FALSE()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_EQ

`KUNIT_ASSERT_EQ (test, left, right)`

Sets an assertion that **left** and **right** are equal.

Parameters

`test` The test context object.

`left` an arbitrary expression that evaluates to a primitive C type.

`right` an arbitrary expression that evaluates to a primitive C type.

Description

Sets an assertion that the values that **left** and **right** evaluate to are equal. This is the same as *KUNIT_EXPECT_EQ()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_PTR_EQ

`KUNIT_ASSERT_PTR_EQ (test, left, right)`

Asserts that pointers **left** and **right** are equal.

Parameters

`test` The test context object.

`left` an arbitrary expression that evaluates to a pointer.

`right` an arbitrary expression that evaluates to a pointer.

Description

Sets an assertion that the values that **left** and **right** evaluate to are equal. This is the same as *KUNIT_EXPECT_EQ()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_NE

`KUNIT_ASSERT_NE (test, left, right)`

An assertion that **left** and **right** are not equal.

Parameters

`test` The test context object.

`left` an arbitrary expression that evaluates to a primitive C type.

`right` an arbitrary expression that evaluates to a primitive C type.

Description

Sets an assertion that the values that **left** and **right** evaluate to are not equal. This is the same as *KUNIT_EXPECT_NE()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_PTR_NE

`KUNIT_ASSERT_PTR_NE (test, left, right)`

Asserts that pointers **left** and **right** are not equal. *KUNIT_ASSERT_PTR_EQ()* - Asserts that pointers **left** and **right** are equal.

Parameters

`test` The test context object.

left an arbitrary expression that evaluates to a pointer.

right an arbitrary expression that evaluates to a pointer.

Description

Sets an assertion that the values that **left** and **right** evaluate to are not equal. This is the same as *KUNIT_EXPECT_NE()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_LT

`KUNIT_ASSERT_LT (test, left, right)`

An assertion that **left** is less than **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an assertion that the value that **left** evaluates to is less than the value that **right** evaluates to. This is the same as *KUNIT_EXPECT_LT()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_LE

`KUNIT_ASSERT_LE (test, left, right)`

An assertion that **left** is less than or equal to **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an assertion that the value that **left** evaluates to is less than or equal to the value that **right** evaluates to. This is the same as *KUNIT_EXPECT_LE()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_GT

`KUNIT_ASSERT_GT (test, left, right)`

An assertion that **left** is greater than **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an assertion that the value that **left** evaluates to is greater than the value that **right** evaluates to. This is the same as *KUNIT_EXPECT_GT()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_GE

`KUNIT_ASSERT_GE (test, left, right)`

Assertion that **left** is greater than or equal to **right**.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a primitive C type.

right an arbitrary expression that evaluates to a primitive C type.

Description

Sets an assertion that the value that **left** evaluates to is greater than the value that **right** evaluates to. This is the same as *KUNIT_EXPECT_GE()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_STREQ

`KUNIT_ASSERT_STREQ (test, left, right)`

An assertion that strings **left** and **right** are equal.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a null terminated string.

right an arbitrary expression that evaluates to a null terminated string.

Description

Sets an assertion that the values that **left** and **right** evaluate to are equal. This is the same as *KUNIT_EXPECT_STREQ()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_STRNEQ

`KUNIT_ASSERT_STRNEQ (test, left, right)`

Expects that strings **left** and **right** are not equal.

Parameters

test The test context object.

left an arbitrary expression that evaluates to a null terminated string.

right an arbitrary expression that evaluates to a null terminated string.

Description

Sets an expectation that the values that **left** and **right** evaluate to are not equal. This is semantically equivalent to KUNIT_ASSERT_TRUE(**test**, strcmp((**left**), (**right**))). See *KUNIT_ASSERT_TRUE()* for more information.

KUNIT_ASSERT_NULL

`KUNIT_ASSERT_NULL (test, ptr)`

Asserts that pointers **ptr** is null.

Parameters

test The test context object.

ptr an arbitrary pointer.

Description

Sets an assertion that the values that **ptr** evaluates to is null. This is the same as *KUNIT_EXPECT_NULL()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_NOT_NULL

`KUNIT_ASSERT_NOT_NULL (test, ptr)`

Asserts that pointers **ptr** is not null.

Parameters

test The test context object.

ptr an arbitrary pointer.

Description

Sets an assertion that the values that **ptr** evaluates to is not null. This is the same as *KUNIT_EXPECT_NOT_NULL()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ASSERT_NOT_ERR_OR_NULL

`KUNIT_ASSERT_NOT_ERR_OR_NULL (test, ptr)`

Assertion that **ptr** is not null and not err.

Parameters

test The test context object.

ptr an arbitrary pointer.

Description

Sets an assertion that the value that **ptr** evaluates to is not null and not an errno stored in a pointer. This is the same as *KUNIT_EXPECT_NOT_ERR_OR_NULL()*, except it causes an assertion failure (see *KUNIT_ASSERT_TRUE()*) when the assertion is not met.

KUNIT_ARRAY_PARAM

`KUNIT_ARRAY_PARAM (name, array, get_desc)`

Define test parameter generator from an array.

Parameters

name prefix for the test parameter generator function.

array array of test parameters.

get_desc function to convert param to description; NULL to use default

Description

Define function **name_gen_params** which uses **array** to generate parameters.

15.8.2 Resource API

This file documents the KUnit resource API.

Most users won't need to use this API directly, power users can use it to store state on a per-test basis, register custom cleanup actions, and more.

struct **kunit_resource**
represents a *test managed resource*

Definition

```
struct kunit_resource {
  void *data;
  const char *name;
  kunit_resource_free_t free;
};
```

Members

data for the user to store arbitrary data.

name optional name

free a user supplied function to free the resource.

Description

Represents a *test managed resource*, a resource which will automatically be cleaned up at the end of a test case. This cleanup is performed by the 'free' function. The *struct kunit_resource* itself is freed automatically with kfree() if it was allocated by KUnit (e.g., by *kunit_alloc_resource()*), but must be freed by the user otherwise.

Resources are reference counted so if a resource is retrieved via *kunit_alloc_and_get_resource()* or *kunit_find_resource()*, we need to call *kunit_put_resource()* to reduce the resource reference count when finished with it. Note that *kunit_alloc_resource()* does not require a kunit_resource_put() because it does not retrieve the resource itself.

```
struct kunit_kmalloc_params {
        size_t size;
        gfp_t gfp;
};

static int kunit_kmalloc_init(struct kunit_resource *res, void *context)
{
        struct kunit_kmalloc_params *params = context;
        res->data = kmalloc(params->size, params->gfp);

        if (!res->data)
```

```
                return -ENOMEM;

        return 0;
}

static void kunit_kmalloc_free(struct kunit_resource *res)
{
        kfree(res->data);
}

void *kunit_kmalloc(struct kunit *test, size_t size, gfp_t gfp)
{
        struct kunit_kmalloc_params params;

        params.size = size;
        params.gfp = gfp;

        return kunit_alloc_resource(test, kunit_kmalloc_init,
                kunit_kmalloc_free, &params);
}
```

Resources can also be named, with lookup/removal done on a name basis also. *kunit_add_named_resource()*, *kunit_find_named_resource()* and kunit_destroy_named_resource(). Resource names must be unique within the test instance.

Example

void **kunit_get_resource**(struct *kunit_resource* *res)

Hold resource for use. Should not need to be used by most users as we automatically get resources retrieved by kunit_find_resource*().

Parameters

struct kunit_resource *res resource

void **kunit_put_resource**(struct *kunit_resource* *res)

When caller is done with retrieved resource, *kunit_put_resource()* should be called to drop reference count. The resource list maintains a reference count on resources, so if no users are utilizing a resource and it is removed from the resource list, it will be freed via the associated free function (if any). Only needs to be used if we alloc_and_get() or find() resource.

Parameters

struct kunit_resource *res resource

int **__kunit_add_resource**(struct *kunit* *test, kunit_resource_init_t init, kunit_resource_free_t free, struct *kunit_resource* *res, void *data)

Internal helper to add a resource.

Parameters

struct kunit *test The test context object.

kunit_resource_init_t init a user-supplied function to initialize the result (if needed). If none is supplied, the resource data value is simply set to **data**. If an init function is supplied, **data** is passed to it instead.

kunit_resource_free_t free a user-supplied function to free the resource (if needed).

struct kunit_resource *res The resource.

void *data value to pass to init function or set in resource data field.

Description

res->should_kfree is not initialised.

int **kunit_add_resource**(struct *kunit* *test, kunit_resource_init_t init, kunit_resource_free_t free, struct *kunit_resource* *res, void *data)

Add a *test managed resource*.

Parameters

struct kunit *test The test context object.

kunit_resource_init_t init a user-supplied function to initialize the result (if needed). If none is supplied, the resource data value is simply set to **data**. If an init function is supplied, **data** is passed to it instead.

kunit_resource_free_t free a user-supplied function to free the resource (if needed).

struct kunit_resource *res The resource.

void *data value to pass to init function or set in resource data field.

int **kunit_add_named_resource**(struct *kunit* *test, kunit_resource_init_t init, kunit_resource_free_t free, struct *kunit_resource* *res, const char *name, void *data)

Add a named *test managed resource*.

Parameters

struct kunit *test The test context object.

kunit_resource_init_t init a user-supplied function to initialize the resource data, if needed.

kunit_resource_free_t free a user-supplied function to free the resource data, if needed.

struct kunit_resource *res The resource.

const char *name name to be set for resource.

void *data value to pass to init function or set in resource data field.

struct *kunit_resource* ***kunit_alloc_and_get_resource**(struct *kunit* *test, kunit_resource_init_t init, kunit_resource_free_t free, gfp_t internal_gfp, void *context)

Allocates and returns a *test managed resource*.

Parameters

struct kunit *test The test context object.

kunit_resource_init_t init a user supplied function to initialize the resource.

kunit_resource_free_t free a user supplied function to free the resource (if needed).

gfp_t internal_gfp gfp to use for internal allocations, if unsure, use GFP_KERNEL

void *context for the user to pass in arbitrary data to the init function.

Description

Allocates a *test managed resource*, a resource which will automatically be cleaned up at the end of a test case. See *struct kunit_resource* for an example.

This is effectively identical to kunit_alloc_resource, but returns the *struct kunit_resource* pointer, not just the 'data' pointer. It therefore also increments the resource's refcount, so *kunit_put_resource()* should be called when you've finished with it.

Note

KUnit needs to allocate memory for a kunit_resource object. You must specify an **internal_gfp** that is compatible with the use context of your resource.

void ***kunit_alloc_resource**(struct *kunit* *test, kunit_resource_init_t init, kunit_resource_free_t free, gfp_t internal_gfp, void *context)

Allocates a *test managed resource*.

Parameters

struct kunit *test The test context object.

kunit_resource_init_t init a user supplied function to initialize the resource.

kunit_resource_free_t free a user supplied function to free the resource (if needed).

gfp_t internal_gfp gfp to use for internal allocations, if unsure, use GFP_KERNEL

void *context for the user to pass in arbitrary data to the init function.

Description

Allocates a *test managed resource*, a resource which will automatically be cleaned up at the end of a test case. See *struct kunit_resource* for an example.

Note

KUnit needs to allocate memory for a kunit_resource object. You must specify an **internal_gfp** that is compatible with the use context of your resource.

bool **kunit_resource_instance_match**(struct *kunit* *test, struct *kunit_resource* *res, void *match_data)

Match a resource with the same instance.

Parameters

struct kunit *test Test case to which the resource belongs.

struct kunit_resource *res The resource.

void *match_data The resource pointer to match against.

Description

An instance of kunit_resource_match_t that matches a resource whose allocation matches **match_data**.

bool **kunit_resource_name_match**(struct *kunit* *test, struct *kunit_resource* *res, void *match_name)

Match a resource with the same name.

Parameters

struct kunit *test Test case to which the resource belongs.

struct kunit_resource *res The resource.

void *match_name The name to match against.

struct *kunit_resource* ***kunit_find_resource**(struct *kunit* *test, kunit_resource_match_t match, void *match_data)

Find a resource using match function/data.

Parameters

struct kunit *test Test case to which the resource belongs.

kunit_resource_match_t match match function to be applied to resources/match data.

void *match_data data to be used in matching.

struct *kunit_resource* ***kunit_find_named_resource**(struct *kunit* *test, const char *name)

Find a resource using match name.

Parameters

struct kunit *test Test case to which the resource belongs.

const char *name match name.

int **kunit_destroy_resource**(struct *kunit* *test, kunit_resource_match_t match, void *match_data)

Find a kunit_resource and destroy it.

Parameters

struct kunit *test Test case to which the resource belongs.

kunit_resource_match_t match Match function. Returns whether a given resource matches **match_data**.

void *match_data Data passed into **match**.

Return

0 if kunit_resource is found and freed, -ENOENT if not found.

void **kunit_remove_resource**(struct *kunit* *test, struct *kunit_resource* *res)

remove resource from resource list associated with test.

Parameters

struct kunit *test The test context object.

struct kunit_resource *res The resource to be removed.

Description

Note that the resource will not be immediately freed since it is likely the caller has a reference to it via alloc_and_get() or find(); in this case a final call to `kunit_put_resource()` is required.

This section documents the KUnit kernel testing API. It is divided into the following sections:

Test API

- documents all of the standard testing API

Resource API

- documents the KUnit resource API

15.9 Test Style and Nomenclature

To make finding, writing, and using KUnit tests as simple as possible, it is strongly encouraged that they are named and written according to the guidelines below. While it is possible to write KUnit tests which do not follow these rules, they may break some tooling, may conflict with other tests, and may not be run automatically by testing systems.

It is recommended that you only deviate from these guidelines when:

1. Porting tests to KUnit which are already known with an existing name.
2. Writing tests which would cause serious problems if automatically run. For example, non-deterministically producing false positives or negatives, or taking a long time to run.

15.9.1 Subsystems, Suites, and Tests

To make tests easy to find, they are grouped into suites and subsystems. A test suite is a group of tests which test a related area of the kernel. A subsystem is a set of test suites which test different parts of a kernel subsystem or a driver.

Subsystems

Every test suite must belong to a subsystem. A subsystem is a collection of one or more KUnit test suites which test the same driver or part of the kernel. A test subsystem should match a single kernel module. If the code being tested cannot be compiled as a module, in many cases the subsystem should correspond to a directory in the source tree or an entry in the `MAINTAINERS` file. If unsure, follow the conventions set by tests in similar areas.

Test subsystems should be named after the code being tested, either after the module (wherever possible), or after the directory or files being tested. Test subsystems should be named to avoid ambiguity where necessary.

If a test subsystem name has multiple components, they should be separated by underscores. *Do not* include "test" or "kunit" directly in the subsystem name unless we are actually testing other tests or the kunit framework itself. For example, subsystems could be called:

`ext4` Matches the module and filesystem name.

`apparmor` Matches the module name and LSM name.

`kasan` Common name for the tool, prominent part of the path `mm/kasan`

`snd_hda_codec_hdmi` Has several components (`snd`, `hda`, `codec`, `hdmi`) separated by underscores. Matches the module name.

Avoid names as shown in examples below:

`linear-ranges` Names should use underscores, not dashes, to separate words. Prefer `linear_ranges`.

qos-kunit-test This name should use underscores, and not have "kunit-test" as a suffix. qos is also ambiguous as a subsystem name, because several parts of the kernel have a qos subsystem. power_qos would be a better name.

pc_parallel_port The corresponding module name is parport_pc, so this subsystem should also be named parport_pc.

Note: The KUnit API and tools do not explicitly know about subsystems. They are a way of categorizing test suites and naming modules which provides a simple, consistent way for humans to find and run tests. This may change in the future.

Suites

KUnit tests are grouped into test suites, which cover a specific area of functionality being tested. Test suites can have shared initialization and shutdown code which is run for all tests in the suite. Not all subsystems need to be split into multiple test suites (for example, simple drivers).

Test suites are named after the subsystem they are part of. If a subsystem contains several suites, the specific area under test should be appended to the subsystem name, separated by an underscore.

In the event that there are multiple types of test using KUnit within a subsystem (for example, both unit tests and integration tests), they should be put into separate suites, with the type of test as the last element in the suite name. Unless these tests are actually present, avoid using _test, _unittest or similar in the suite name.

The full test suite name (including the subsystem name) should be specified as the .name member of the kunit_suite struct, and forms the base for the module name. For example, test suites could include:

ext4_inode Part of the ext4 subsystem, testing the inode area.

kunit_try_catch Part of the kunit implementation itself, testing the try_catch area.

apparmor_property_entry Part of the apparmor subsystem, testing the property_entry area.

kasan The kasan subsystem has only one suite, so the suite name is the same as the subsystem name.

Avoid names, for example:

ext4_ext4_inode There is no reason to state the subsystem twice.

property_entry The suite name is ambiguous without the subsystem name.

kasan_integration_test Because there is only one suite in the kasan subsystem, the suite should just be called as kasan. Do not redundantly add integration_test. It should be a separate test suite. For example, if the unit tests are added, then that suite could be named as kasan_unittest or similar.

Test Cases

Individual tests consist of a single function which tests a constrained codepath, property, or function. In the test output, an individual test's results will show up as subtests of the suite's results.

Tests should be named after what they are testing. This is often the name of the function being tested, with a description of the input or codepath being tested. As tests are C functions, they should be named and written in accordance with the kernel coding style.

Note: As tests are themselves functions, their names cannot conflict with other C identifiers in the kernel. This may require some creative naming. It is a good idea to make your test functions *static* to avoid polluting the global namespace.

Example test names include:

`unpack_u32_with_null_name` Tests the `unpack_u32` function when a NULL name is passed in.

`test_list_splice` Tests the `list_splice` macro. It has the prefix `test_` to avoid a name conflict with the macro itself.

Should it be necessary to refer to a test outside the context of its test suite, the *fully-qualified* name of a test should be the suite name followed by the test name, separated by a colon (i.e. `suite:test`).

15.9.2 Test Kconfig Entries

Every test suite should be tied to a Kconfig entry.

This Kconfig entry must:

- be named `CONFIG_<name>_KUNIT_TEST`: where <name> is the name of the test suite.
- be listed either alongside the config entries for the driver/subsystem being tested, or be under [Kernel Hacking]->[Kernel Testing and Coverage]
- depend on `CONFIG_KUNIT`.
- be visible only if `CONFIG_KUNIT_ALL_TESTS` is not enabled.
- have a default value of `CONFIG_KUNIT_ALL_TESTS`.
- have a brief description of KUnit in the help text.

If we are not able to meet above conditions (for example, the test is unable to be built as a module), Kconfig entries for tests should be tristate.

For example, a Kconfig entry might look like:

```
config FOO_KUNIT_TEST
        tristate "KUnit test for foo" if !KUNIT_ALL_TESTS
        depends on KUNIT
        default KUNIT_ALL_TESTS
        help
          This builds unit tests for foo.
          For more information on KUnit and unit tests in general,
          please refer to the KUnit documentation in Documentation/dev-tools/
↪kunit/.

          If unsure, say N.
```

15.9.3 Test File and Module Names

KUnit tests can often be compiled as a module. These modules should be named after the test suite, followed by `_test`. If this is likely to conflict with non-KUnit tests, the suffix `_kunit` can also be used.

The easiest way of achieving this is to name the file containing the test suite `<suite>_test.c` (or, as above, `<suite>_kunit.c`). This file should be placed next to the code under test.

If the suite name contains some or all of the name of the test's parent directory, it may make sense to modify the source filename to reduce redundancy. For example, a `foo_firmware` suite could be in the `foo/firmware_test.c` file.

15.10 Frequently Asked Questions

15.10.1 How is this different from Autotest, kselftest, and so on?

KUnit is a unit testing framework. Autotest, kselftest (and some others) are not.

A unit test is supposed to test a single unit of code in isolation and hence the name *unit test*. A unit test should be the finest granularity of testing and should allow all possible code paths to be tested in the code under test. This is only possible if the code under test is small and does not have any external dependencies outside of the test's control like hardware.

There are no testing frameworks currently available for the kernel that do not require installing the kernel on a test machine or in a virtual machine. All testing frameworks require tests to be written in userspace and run on the kernel under test. This is true for Autotest, kselftest, and some others, disqualifying any of them from being considered unit testing frameworks.

15.10.2 Does KUnit support running on architectures other than UML?

Yes, mostly.

For the most part, the KUnit core framework (what we use to write the tests) can compile to any architecture. It compiles like just another part of the kernel and runs when the kernel boots, or when built as a module, when the module is loaded. However, there is infrastructure, like the KUnit Wrapper (`tools/testing/kunit/kunit.py`) that does not support other architectures.

In short, yes, you can run KUnit on other architectures, but it might require more work than using KUnit on UML.

For more information, see *Exiting Early on Failed Expectations*.

15.10.3 What is the difference between a unit test and other kinds of tests?

Most existing tests for the Linux kernel would be categorized as an integration test, or an end-to-end test.

- A unit test is supposed to test a single unit of code in isolation. A unit test should be the finest granularity of testing and, as such, allows all possible code paths to be tested in the code under test. This is only possible if the code under test is small and does not have any external dependencies outside of the test's control like hardware.
- An integration test tests the interaction between a minimal set of components, usually just two or three. For example, someone might write an integration test to test the interaction between a driver and a piece of hardware, or to test the interaction between the userspace libraries the kernel provides and the kernel itself. However, one of these tests would probably not test the entire kernel along with hardware interactions and interactions with the userspace.
- An end-to-end test usually tests the entire system from the perspective of the code under test. For example, someone might write an end-to-end test for the kernel by installing a production configuration of the kernel on production hardware with a production userspace and then trying to exercise some behavior that depends on interactions between the hardware, the kernel, and userspace.

15.10.4 KUnit is not working, what should I do?

Unfortunately, there are a number of things which can break, but here are some things to try.

1. Run `./tools/testing/kunit/kunit.py run` with the `--raw_output` parameter. This might show details or error messages hidden by the kunit_tool parser.
2. Instead of running `kunit.py run`, try running `kunit.py config`, `kunit.py build`, and `kunit.py exec` independently. This can help track down where an issue is occurring. (If you think the parser is at fault, you can run it manually against `stdin` or a file with `kunit.py parse`.)
3. Running the UML kernel directly can often reveal issues or error messages, `kunit_tool` ignores. This should be as simple as running `./vmlinux` after building the UML kernel (for example, by using `kunit.py build`). Note that UML has some unusual requirements (such as the host having a tmpfs filesystem mounted), and has had issues in the past when built statically and the host has KASLR enabled. (On older host kernels, you may need to run `` setarch `uname -m` -R ./vmlinux `` to disable KASLR.)
4. Make sure the kernel .config has `CONFIG_KUNIT=y` and at least one test (e.g. `CONFIG_KUNIT_EXAMPLE_TEST=y`). kunit_tool will keep its .config around, so you can see what config was used after running `kunit.py run`. It also preserves any config changes you might make, so you can enable/disable things with `make ARCH=um menuconfig` or similar, and then re-run kunit_tool.
5. Try to run `make ARCH=um defconfig` before running `kunit.py run`. This may help clean up any residual config items which could be causing problems.
6. Finally, try running KUnit outside UML. KUnit and KUnit tests can be built into any kernel, or can be built as a module and loaded at runtime. Doing so should allow you to determine if UML is causing the issue you're seeing. When tests are built-in, they will execute when the kernel boots, and modules will automatically execute associated tests when loaded.

> Test results can be collected from /sys/kernel/debug/kunit/<test suite>/results, and can be parsed with kunit.py parse. For more details, see "KUnit on non-UML architectures" in *Writing Tests*.

If none of the above tricks help, you are always welcome to email any issues to kunit-dev@googlegroups.com.

15.11 Tips For Writing KUnit Tests

15.11.1 Exiting early on failed expectations

KUNIT_EXPECT_EQ and friends will mark the test as failed and continue execution. In some cases, it's unsafe to continue and you can use the KUNIT_ASSERT variant to exit on failure.

```
void example_test_user_alloc_function(struct kunit *test)
{
        void *object = alloc_some_object_for_me();

        /* Make sure we got a valid pointer back. */
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, object);
        do_something_with_object(object);
}
```

15.11.2 Allocating memory

Where you would use kzalloc, you should prefer kunit_kzalloc instead. KUnit will ensure the memory is freed once the test completes.

This is particularly useful since it lets you use the KUNIT_ASSERT_EQ macros to exit early from a test without having to worry about remembering to call kfree.

Example:

```
void example_test_allocation(struct kunit *test)
{
        char *buffer = kunit_kzalloc(test, 16, GFP_KERNEL);
        /* Ensure allocation succeeded. */
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, buffer);

        KUNIT_ASSERT_STREQ(test, buffer, "");
}
```

15.11.3 Testing static functions

If you don't want to expose functions or variables just for testing, one option is to conditionally #include the test file at the end of your .c file, e.g.

```
/* In my_file.c */

static int do_interesting_thing();

#ifdef CONFIG_MY_KUNIT_TEST
#include "my_kunit_test.c"
#endif
```

15.11.4 Injecting test-only code

Similarly to the above, it can be useful to add test-specific logic.

```
/* In my_file.h */

#ifdef CONFIG_MY_KUNIT_TEST
/* Defined in my_kunit_test.c */
void test_only_hook(void);
#else
void test_only_hook(void) { }
#endif
```

This test-only code can be made more useful by accessing the current kunit test, see below.

15.11.5 Accessing the current test

In some cases, you need to call test-only code from outside the test file, e.g. like in the example above or if you're providing a fake implementation of an ops struct. There is a `kunit_test` field in `task_struct`, so you can access it via `current->kunit_test`.

Here's a slightly in-depth example of how one could implement "mocking":

```
#include <linux/sched.h> /* for current */

struct test_data {
        int foo_result;
        int want_foo_called_with;
};

static int fake_foo(int arg)
{
        struct kunit *test = current->kunit_test;
        struct test_data *test_data = test->priv;

        KUNIT_EXPECT_EQ(test, test_data->want_foo_called_with, arg);
        return test_data->foo_result;
}

static void example_simple_test(struct kunit *test)
{
        /* Assume priv is allocated in the suite's .init */
        struct test_data *test_data = test->priv;

        test_data->foo_result = 42;
        test_data->want_foo_called_with = 1;

        /* In a real test, we'd probably pass a pointer to fake_foo somewhere
         * like an ops struct, etc. instead of calling it directly. */
        KUNIT_EXPECT_EQ(test, fake_foo(1), 42);
}
```

Note: here we're able to get away with using `test->priv`, but if you wanted something more flexible you could use a named `kunit_resource`, see *Test API*.

15.11.6 Failing the current test

But sometimes, you might just want to fail the current test. In that case, we have `kunit_fail_current_test(fmt, args...)` which is defined in `<kunit/test-bug.h>` and doesn't require pulling in `<kunit/test.h>`.

E.g. say we had an option to enable some extra debug checks on some data structure:

```
#include <kunit/test-bug.h>

#ifdef CONFIG_EXTRA_DEBUG_CHECKS
static void validate_my_data(struct data *data)
{
        if (is_valid(data))
                return;

        kunit_fail_current_test("data %p is invalid", data);

        /* Normal, non-KUnit, error reporting code here. */
}
#else
static void my_debug_function(void) { }
#endif
```

15.11.7 Customizing error messages

Each of the KUNIT_EXPECT and KUNIT_ASSERT macros have a _MSG variant. These take a format string and arguments to provide additional context to the automatically generated error messages.

```
char some_str[41];
generate_sha1_hex_string(some_str);

/* Before. Not easy to tell why the test failed. */
KUNIT_EXPECT_EQ(test, strlen(some_str), 40);

/* After. Now we see the offending string. */
KUNIT_EXPECT_EQ_MSG(test, strlen(some_str), 40, "some_str='%s'", some_str);
```

Alternatively, one can take full control over the error message by using KUNIT_FAIL(), e.g.

```
/* Before */
KUNIT_EXPECT_EQ(test, some_setup_function(), 0);

/* After: full control over the failure message. */
if (some_setup_function())
        KUNIT_FAIL(test, "Failed to setup thing for testing");
```

Next Steps

- Optional: see the *Writing Tests* page for a more in-depth explanation of KUnit.

15.12 Tips For Running KUnit Tests

15.12.1 Using kunit.py run ("kunit tool")

Running from any directory

It can be handy to create a bash function like:

```
function run_kunit() {
  ( cd "$(git rev-parse --show-toplevel)" && ./tools/testing/kunit/kunit.py␣
↪run $@ )
}
```

Note: Early versions of kunit.py (before 5.6) didn't work unless run from the kernel root, hence the use of a subshell and cd.

Running a subset of tests

`kunit.py run` accepts an optional glob argument to filter tests. The format is `"<suite_glob>[.test_glob]"`.

Say that we wanted to run the sysctl tests, we could do so via:

```
$ echo -e 'CONFIG_KUNIT=y\nCONFIG_KUNIT_ALL_TESTS=y' > .kunit/.kunitconfig
$ ./tools/testing/kunit/kunit.py run 'sysctl*'
```

We can filter down to just the "write" tests via:

```
$ echo -e 'CONFIG_KUNIT=y\nCONFIG_KUNIT_ALL_TESTS=y' > .kunit/.kunitconfig
$ ./tools/testing/kunit/kunit.py run 'sysctl*.*write*'
```

We're paying the cost of building more tests than we need this way, but it's easier than fiddling with `.kunitconfig` files or commenting out `kunit_suite`'s.

However, if we wanted to define a set of tests in a less ad hoc way, the next tip is useful.

Defining a set of tests

`kunit.py run` (along with `build`, and `config`) supports a `--kunitconfig` flag. So if you have a set of tests that you want to run on a regular basis (especially if they have other dependencies), you can create a specific `.kunitconfig` for them.

E.g. kunit has one for its tests:

```
$ ./tools/testing/kunit/kunit.py run --kunitconfig=lib/kunit/.kunitconfig
```

Alternatively, if you're following the convention of naming your file `.kunitconfig`, you can just pass in the dir, e.g.

```
$ ./tools/testing/kunit/kunit.py run --kunitconfig=lib/kunit
```

Note: This is a relatively new feature (5.12+) so we don't have any conventions yet about on what files should be checked in versus just kept around locally. It's up to you and your maintainer to decide if a config is useful enough to submit (and therefore have to maintain).

Note: Having `.kunitconfig` fragments in a parent and child directory is iffy. There's discussion about adding an "import" statement in these files to make it possible to have a top-level config run tests from all child directories. But that would mean `.kunitconfig` files are no longer just simple .config fragments.

One alternative would be to have kunit tool recursively combine configs automagically, but tests could theoretically depend on incompatible options, so handling that would be tricky.

Setting kernel commandline parameters

You can use `--kernel_args` to pass arbitrary kernel arguments, e.g.

```
$ ./tools/testing/kunit/kunit.py run --kernel_args=param=42 --kernel_
↪args=param2=false
```

Generating code coverage reports under UML

Note: TODO(brendanhiggins@google.com): There are various issues with UML and versions of gcc 7 and up. You're likely to run into missing `.gcda` files or compile errors.

This is different from the "normal" way of getting coverage information that is documented in *Using gcov with the Linux kernel*.

Instead of enabling `CONFIG_GCOV_KERNEL=y`, we can set these options:

```
CONFIG_DEBUG_KERNEL=y
CONFIG_DEBUG_INFO=y
CONFIG_DEBUG_INFO_DWARF_TOOLCHAIN_DEFAULT=y
CONFIG_GCOV=y
```

Putting it together into a copy-pastable sequence of commands:

```
# Append coverage options to the current config
$ echo -e "CONFIG_DEBUG_KERNEL=y\nCONFIG_DEBUG_INFO=y\nCONFIG_DEBUG_INFO_DWARF_
↪TOOLCHAIN_DEFAULT=y\nCONFIG_GCOV=y" >> .kunit/.kunitconfig
$ ./tools/testing/kunit/kunit.py run
# Extract the coverage information from the build dir (.kunit/)
$ lcov -t "my_kunit_tests" -o coverage.info -c -d .kunit/

# From here on, it's the same process as with CONFIG_GCOV_KERNEL=y
# E.g. can generate an HTML report in a tmp dir like so:
$ genhtml -o /tmp/coverage_html coverage.info
```

If your installed version of gcc doesn't work, you can tweak the steps:

```
$ ./tools/testing/kunit/kunit.py run --make_options=CC=/usr/bin/gcc-6
$ lcov -t "my_kunit_tests" -o coverage.info -c -d .kunit/ --gcov-tool=/usr/bin/
↪gcov-6
```

15.12.2 Running tests manually

Running tests without using `kunit.py run` is also an important use case. Currently it's your only option if you want to test on architectures other than UML.

As running the tests under UML is fairly straightforward (configure and compile the kernel, run the `./linux` binary), this section will focus on testing non-UML architectures.

Running built-in tests

When setting tests to `=y`, the tests will run as part of boot and print results to dmesg in TAP format. So you just need to add your tests to your `.config`, build and boot your kernel as normal.

So if we compiled our kernel with:

```
CONFIG_KUNIT=y
CONFIG_KUNIT_EXAMPLE_TEST=y
```

Then we'd see output like this in dmesg signaling the test ran and passed:

```
TAP version 14
1..1
    # Subtest: example
    1..1
    # example_simple_test: initializing
    ok 1 - example_simple_test
ok 1 - example
```

Running tests as modules

Depending on the tests, you can build them as loadable modules.

For example, we'd change the config options from before to

```
CONFIG_KUNIT=y
CONFIG_KUNIT_EXAMPLE_TEST=m
```

Then after booting into our kernel, we can run the test via

```
$ modprobe kunit-example-test
```

This will then cause it to print TAP output to stdout.

Note: The `modprobe` will *not* have a non-zero exit code if any test failed (as of 5.13). But `kunit.py parse` would, see below.

Note: You can set `CONFIG_KUNIT=m` as well, however, some features will not work and thus some tests might break. Ideally tests would specify they depend on `KUNIT=y` in their `Kconfig`'s, but this is an edge case most test authors won't think about. As of 5.13, the only difference is that current->kunit_test will not exist.

Pretty-printing results

You can use kunit.py parse to parse dmesg for test output and print out results in the same familiar format that kunit.py run does.

```
$ ./tools/testing/kunit/kunit.py parse /var/log/dmesg
```

Retrieving per suite results

Regardless of how you're running your tests, you can enable CONFIG_KUNIT_DEBUGFS to expose per-suite TAP-formatted results:

```
CONFIG_KUNIT=y
CONFIG_KUNIT_EXAMPLE_TEST=m
CONFIG_KUNIT_DEBUGFS=y
```

The results for each suite will be exposed under /sys/kernel/debug/kunit/<suite>/results. So using our example config:

```
$ modprobe kunit-example-test > /dev/null
$ cat /sys/kernel/debug/kunit/example/results
... <TAP output> ...

# After removing the module, the corresponding files will go away
$ modprobe -r kunit-example-test
$ cat /sys/kernel/debug/kunit/example/results
/sys/kernel/debug/kunit/example/results: No such file or directory
```

Generating code coverage reports

See *Using gcov with the Linux kernel* for details on how to do this.

The only vaguely KUnit-specific advice here is that you probably want to build your tests as modules. That way you can isolate the coverage from tests from other code executed during boot, e.g.

```
# Reset coverage counters before running the test.
$ echo 0 > /sys/kernel/debug/gcov/reset
$ modprobe kunit-example-test
```

This section details the kernel unit testing framework.

15.13 Introduction

KUnit (Kernel unit testing framework) provides a common framework for unit tests within the Linux kernel. Using KUnit, you can define groups of test cases called test suites. The tests either run on kernel boot if built-in, or load as a module. KUnit automatically flags and reports failed test cases in the kernel log. The test results appear in TAP (Test Anything Protocol) format. It is inspired by JUnit, Python's unittest.mock, and GoogleTest/GoogleMock (C++ unit testing framework).

KUnit tests are part of the kernel, written in the C (programming) language, and test parts of the Kernel implementation (example: a C language function). Excluding build time, from invocation to completion, KUnit can run around 100 tests in less than 10 seconds. KUnit can test any kernel component, for example: file system, system calls, memory management, device drivers and so on.

KUnit follows the white-box testing approach. The test has access to internal system functionality. KUnit runs in kernel space and is not restricted to things exposed to user-space.

In addition, KUnit has kunit_tool, a script (`tools/testing/kunit/kunit.py`) that configures the Linux kernel, runs KUnit tests under QEMU or UML (User Mode Linux), parses the test results and displays them in a user friendly manner.

15.13.1 Features

- Provides a framework for writing unit tests.
- Runs tests on any kernel architecture.
- Runs a test in milliseconds.

15.13.2 Prerequisites

- Any Linux kernel compatible hardware.
- For Kernel under test, Linux kernel version 5.5 or greater.

15.14 Unit Testing

A unit test tests a single unit of code in isolation. A unit test is the finest granularity of testing and allows all possible code paths to be tested in the code under test. This is possible if the code under test is small and does not have any external dependencies outside of the test's control like hardware.

15.14.1 Write Unit Tests

To write good unit tests, there is a simple but powerful pattern: Arrange-Act-Assert. This is a great way to structure test cases and defines an order of operations.

- Arrange inputs and targets: At the start of the test, arrange the data that allows a function to work. Example: initialize a statement or object.
- Act on the target behavior: Call your function/code under test.
- Assert expected outcome: Verify that the result (or resulting state) is as expected.

15.14.2 Unit Testing Advantages

- Increases testing speed and development in the long run.
- Detects bugs at initial stage and therefore decreases bug fix cost compared to acceptance testing.
- Improves code quality.
- Encourages writing testable code.

15.15 How do I use it?

- *Getting Started* - for KUnit new users.
- *KUnit Architecture* - KUnit architecture.
- *Run Tests with kunit_tool* - run kunit_tool.
- *Run Tests without kunit_tool* - run tests without kunit_tool.
- *Writing Tests* - write tests.
- *Tips For Writing KUnit Tests* - best practices with examples.
- *API Reference* - KUnit APIs used for testing.
- *kunit_tool How-To* - kunit_tool helper script.
- *Frequently Asked Questions* - KUnit common questions and answers.

CHAPTER

SIXTEEN

THE KERNEL TEST ANYTHING PROTOCOL (KTAP), VERSION 1

TAP, or the Test Anything Protocol is a format for specifying test results used by a number of projects. It's website and specification are found at this link. The Linux Kernel largely uses TAP output for test results. However, Kernel testing frameworks have special needs for test results which don't align with the original TAP specification. Thus, a "Kernel TAP" (KTAP) format is specified to extend and alter TAP to support these use-cases. This specification describes the generally accepted format of KTAP as it is currently used in the kernel.

KTAP test results describe a series of tests (which may be nested: i.e., test can have subtests), each of which can contain both diagnostic data – e.g., log lines – and a final result. The test structure and results are machine-readable, whereas the diagnostic data is unstructured and is there to aid human debugging.

KTAP output is built from four different types of lines: - Version lines - Plan lines - Test case result lines - Diagnostic lines

In general, valid KTAP output should also form valid TAP output, but some information, in particular nested test results, may be lost. Also note that there is a stagnant draft specification for TAP14, KTAP diverges from this in a couple of places (notably the "Subtest" header), which are described where relevant later in this document.

16.1 Version lines

All KTAP-formatted results begin with a "version line" which specifies which version of the (K)TAP standard the result is compliant with.

For example: - "KTAP version 1" - "TAP version 13" - "TAP version 14"

Note that, in KTAP, subtests also begin with a version line, which denotes the start of the nested test results. This differs from TAP14, which uses a separate "Subtest" line.

While, going forward, "KTAP version 1" should be used by compliant tests, it is expected that most parsers and other tooling will accept the other versions listed here for compatibility with existing tests and frameworks.

16.2 Plan lines

A test plan provides the number of tests (or subtests) in the KTAP output.

Plan lines must follow the format of "1..N" where N is the number of tests or subtests. Plan lines follow version lines to indicate the number of nested tests.

While there are cases where the number of tests is not known in advance – in which case the test plan may be omitted – it is strongly recommended one is present where possible.

16.3 Test case result lines

Test case result lines indicate the final status of a test. They are required and must have the format:

```
<result> <number> [<description>][ # [<directive>] [<diagnostic data>]]
```

The result can be either "ok", which indicates the test case passed, or "not ok", which indicates that the test case failed.

<number> represents the number of the test being performed. The first test must have the number 1 and the number then must increase by 1 for each additional subtest within the same test at the same nesting level.

The description is a description of the test, generally the name of the test, and can be any string of words (can't include #). The description is optional, but recommended.

The directive and any diagnostic data is optional. If either are present, they must follow a hash sign, "#".

A directive is a keyword that indicates a different outcome for a test other than passed and failed. The directive is optional, and consists of a single keyword preceding the diagnostic data. In the event that a parser encounters a directive it doesn't support, it should fall back to the "ok" / "not ok" result.

Currently accepted directives are:

- "SKIP", which indicates a test was skipped (note the result of the test case result line can be either "ok" or "not ok" if the SKIP directive is used)
- "TODO", which indicates that a test is not expected to pass at the moment, e.g. because the feature it is testing is known to be broken. While this directive is inherited from TAP, its use in the kernel is discouraged.
- "XFAIL", which indicates that a test is expected to fail. This is similar to "TODO", above, and is used by some kselftest tests.
- "TIMEOUT", which indicates a test has timed out (note the result of the test case result line should be "not ok" if the TIMEOUT directive is used)
- "ERROR", which indicates that the execution of a test has failed due to a specific error that is included in the diagnostic data. (note the result of the test case result line should be "not ok" if the ERROR directive is used)

The diagnostic data is a plain-text field which contains any additional details about why this result was produced. This is typically an error message for ERROR or failed tests, or a description of missing dependencies for a SKIP result.

The diagnostic data field is optional, and results which have neither a directive nor any diagnostic data do not need to include the "#" field separator.

Example result lines include:

```
ok 1 test_case_name
```

The test "test_case_name" passed.

```
not ok 1 test_case_name
```

The test "test_case_name" failed.

```
ok 1 test # SKIP necessary dependency unavailable
```

The test "test" was SKIPPED with the diagnostic message "necessary dependency unavailable".

```
not ok 1 test # TIMEOUT 30 seconds
```

The test "test" timed out, with diagnostic data "30 seconds".

```
ok 5 check return code # rcode=0
```

The test "check return code" passed, with additional diagnostic data "rcode=0"

16.4 Diagnostic lines

If tests wish to output any further information, they should do so using "diagnostic lines". Diagnostic lines are optional, freeform text, and are often used to describe what is being tested and any intermediate results in more detail than the final result and diagnostic data line provides.

Diagnostic lines are formatted as "# <diagnostic_description>", where the description can be any string. Diagnostic lines can be anywhere in the test output. As a rule, diagnostic lines regarding a test are directly before the test result line for that test.

Note that most tools will treat unknown lines (see below) as diagnostic lines, even if they do not start with a "#": this is to capture any other useful kernel output which may help debug the test. It is nevertheless recommended that tests always prefix any diagnostic output they have with a "#" character.

16.5 Unknown lines

There may be lines within KTAP output that do not follow the format of one of the four formats for lines described above. This is allowed, however, they will not influence the status of the tests.

This is an important difference from TAP. Kernel tests may print messages to the system console or a log file. Both of these destinations may contain messages either from unrelated kernel or userspace activity, or kernel messages from non-test code that is invoked by the test. The kernel code invoked by the test likely is not aware that a test is in progress and thus can not print the message as a diagnostic message.

16.6 Nested tests

In KTAP, tests can be nested. This is done by having a test include within its output an entire set of KTAP-formatted results. This can be used to categorize and group related tests, or to split out different results from the same test.

The “parent” test’s result should consist of all of its subtests’ results, starting with another KTAP version line and test plan, and end with the overall result. If one of the subtests fail, for example, the parent test should also fail.

Additionally, all lines in a subtest should be indented. One level of indentation is two spaces: ” “. The indentation should begin at the version line and should end before the parent test’s result line.

“Unknown lines” are not considered to be lines in a subtest and thus are allowed to be either indented or not indented.

An example of a test with two nested subtests:

```
KTAP version 1
1..1
  KTAP version 1
  1..2
  ok 1 test_1
  not ok 2 test_2
# example failed
not ok 1 example
```

An example format with multiple levels of nested testing:

```
KTAP version 1
1..2
  KTAP version 1
  1..2
    KTAP version 1
    1..2
    not ok 1 test_1
    ok 2 test_2
  not ok 1 test_3
  ok 2 test_4 # SKIP
```

```
not ok 1 example_test_1
ok 2 example_test_2
```

16.7 Major differences between TAP and KTAP

Feature	TAP	KTAP
yaml and json in diagnosic message	ok	not recommended
TODO directive	ok	not recognized
allows an arbitrary number of tests to be nested	no	yes
“Unknown lines” are in category of “Anything else”	yes	no
“Unknown lines” are	incorrect	allowed

The TAP14 specification does permit nested tests, but instead of using another nested version line, uses a line of the form “Subtest: <name>” where <name> is the name of the parent test.

16.8 Example KTAP output

```
KTAP version 1
1..1
  KTAP version 1
  1..3
    KTAP version 1
    1..1
    # test_1: initializing test_1
    ok 1 test_1
  ok 1 example_test_1
    KTAP version 1
    1..2
    ok 1 test_1 # SKIP test_1 skipped
    ok 2 test_2
  ok 2 example_test_2
    KTAP version 1
    1..3
    ok 1 test_1
    # test_2: FAIL
    not ok 2 test_2
    ok 3 test_3 # SKIP test_3 skipped
  not ok 3 example_test_3
not ok 1 main_test
```

This output defines the following hierarchy:

A single test called “main_test”, which fails, and has three subtests: - “example_test_1”, which passes, and has one subtest:

- “test_1”, which passes, and outputs the diagnostic message “test_1: initializing test_1”
- “example_test_2”, which passes, and has two subtests:

  - “test_1”, which is skipped, with the explanation “test_1 skipped”
  - “test_2”, which passes
- “example_test_3”, which fails, and has three subtests
  - “test_1”, which passes
  - “test_2”, which outputs the diagnostic line “test_2: FAIL”, and fails.
  - “test_3”, which is skipped with the explanation “test_3 skipped”

Note that the individual subtests with the same names do not conflict, as they are found in different parent tests. This output also exhibits some sensible rules for “bubbling up” test results: a test fails if any of its subtests fail. Skipped tests do not affect the result of the parent test (though it often makes sense for a test to be marked skipped if _all_ of its subtests have been skipped).

16.9 See also:

- The TAP specification: https://testanything.org/tap-version-13-specification.html
- The (stagnant) TAP version 14 specification: https://github.com/TestAnything/Specification/blob/tap-14-specification/specification.md
- The kselftest documentation: *Linux Kernel Selftests*
- The KUnit documentation: *KUnit - Linux Kernel Unit Testing*

INDEX