Linux Dev-tools Documentation

The kernel development community

Jul 14, 2020

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!

Table of contents

CHAPTER

ONE

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.

1.1 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

1.2 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

1.3 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.

1.3.1 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 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
```

1.4 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, the return value of the `make coccicheck` captures this return value.

1.5 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
```

1.6 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 following command may be used:

```
make C=1 CHECK="scripts/coccicheck"
```

To check only newly edited code, use the value 2 for the C flag, i.e.:

```
make C=2 CHECK="scripts/coccicheck"
```

In these modes, which works 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.

1.7 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 instance 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/arizona-irq.c
```

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

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

1.8 .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 defaults options for Linux with our own Linux .cocciconfig. This hints to coccinelle 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.

1.9 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.

1.10 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
```

1.11 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 at least a version of Coccinelle, this can be specified as follows, as an example if requiring at least Coccinelle >= 1.0.5:

```
// Requires: 1.0.5
```

1.12 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/`.

1.13 Detailed description of the `report` mode

`report` generates a list in the following format:

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

1.13.1 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
```

1.14 Detailed description of the `patch` mode

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

1.14.1 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;
```

1.15 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.

1.15.1 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;
```

1.16 Detailed description of the `org` mode

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

1.16.1 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@
```

(continues on next page)

(continued from previous page)

```
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

TWO

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.

2.1 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.

2.2 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.

2.3 Getting sparse

You can get latest released versions from the Sparse homepage at https://sparse.wiki.kernel.org/index.php/Main_Page

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.

2.4 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.

CHAPTER

THREE

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).

3.1 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
```

3.2 Coverage collection

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

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
```

(continues on next page)

(continued from previous page)

```
#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>

#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))
```

(continues on next page)

(continued from previous page)

```
            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).

3.3 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;
```

(continues on next page)

(continued from previous page)

```
    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++) {
            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.

3.4 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 proccess 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.

```
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)
```

(continues on next page)

(continued from previous page)

```
#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.
     */
```

(continues on next page)

(continued from previous page)

```
    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

FOUR

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?)

4.1 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
```

4.2 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.

4.3 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`.

4.4 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.

4.5 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
```

4.6 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.

4.7 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.

4.8 Appendix A: gather_on_build.sh

Sample script to gather coverage meta files on the build machine (see 6a):

```
#!/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
```

4.9 Appendix B: gather_on_test.sh

Sample script to gather coverage data files on the test machine (see 6b):

```
#!/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

FIVE

THE KERNEL ADDRESS SANITIZER (KASAN)

5.1 Overview

KernelAddressSANitizer (KASAN) is a dynamic memory error detector designed to find out-of-bound and use-after-free bugs. KASAN has two modes: generic KASAN (similar to userspace ASan) and software tag-based KASAN (similar to userspace HWASan).

KASAN uses compile-time instrumentation to insert validity checks before every memory access, and therefore requires a compiler version that supports that.

Generic KASAN is supported in both GCC and Clang. With GCC it requires version 4.9.2 or later for basic support and version 5.0 or later for detection of out-of-bounds accesses for stack and global variables and for inline instrumentation mode (see the Usage section). With Clang it requires version 7.0.0 or later and it doesn't support detection of out-of-bounds accesses for global variables yet.

Tag-based KASAN is only supported in Clang and requires version 7.0.0 or later.

Currently generic KASAN is supported for the x86_64, arm64, xtensa, s390 and riscv architectures, and tag-based KASAN is supported only for arm64.

5.2 Usage

To enable KASAN configure kernel with:

```
CONFIG_KASAN = y
```

and choose between CONFIG_KASAN_GENERIC (to enable generic KASAN) and CONFIG_KASAN_SW_TAGS (to enable software tag-based KASAN).

You also need to choose between CONFIG_KASAN_OUTLINE and CONFIG_KASAN_INLINE. Outline and inline are compiler instrumentation types. The former produces smaller binary while the latter is 1.1 - 2 times faster.

Both KASAN modes work with both SLUB and SLAB memory allocators. For better bug detection and nicer reporting, enable CONFIG_STACKTRACE.

To augment reports with last allocation and freeing stack of the physical page, it is recommended to enable also CONFIG_PAGE_OWNER and boot with page_owner=on.

To disable 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):

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

- For all files in one directory:

```
KASAN_SANITIZE := n
```

5.2.1 Error reports

A typical out-of-bounds access generic 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
```

(continues on next page)

(continued from previous page)

```
 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 header of the report provides a short summary of what kind of bug happened and what kind of access caused it. It's followed by a stack trace of the bad access, a stack trace of where the accessed memory was allocated (in case bad access happens on a slab object), 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 information about the accessed memory page.

In the last section the report shows memory state around the accessed address. Reading this part requires some understanding of how KASAN works.

The state of each 8 aligned bytes of memory is encoded in one shadow byte. Those 8 bytes can be accessible, partially accessible, freed or be a redzone. We use the following encoding for each shadow byte: 0 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. We use 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 arrows point to the shadow byte 03, which means that the accessed address is partially accessible.

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

5.3 Implementation details

5.3.1 Generic KASAN

From a high level, our approach to memory error detection is similar to that of kmemcheck: use shadow memory to record whether each byte of memory is safe to access, and use compile-time instrumentation to insert checks of shadow memory on each memory access.

Generic KASAN dedicates 1/8th of kernel memory to its shadow memory (e.g. 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 ((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 access is valid or not by checking corresponding shadow memory.

GCC 5.0 has possibility to perform inline instrumentation. Instead of making function calls GCC directly inserts the code to check the shadow memory. This option significantly enlarges kernel but it gives x1.1-x2 performance boost over outline instrumented kernel.

5.3.2 Software tag-based KASAN

Tag-based KASAN uses the Top Byte Ignore (TBI) feature of modern arm64 CPUs to store a pointer tag in the top byte of kernel pointers. Like generic KASAN 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 tag-based KASAN generates a random tag, tags the allocated memory with this tag, and embeds this 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 tag of the memory that is being accessed is equal to tag of the pointer that is used to access this memory. In case of a tag mismatch tag-based KASAN prints a bug report.

Software tag-based KASAN also has two instrumentation modes (outline, that emits callbacks to check memory accesses; and inline, that performs the shadow memory checks inline). With outline instrumentation mode, a bug report is simply 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.

A potential expansion of this mode is a hardware tag-based mode, which would use hardware memory tagging support instead of compiler instrumentation and manual shadow memory manipulation.

5.4 What memory accesses are sanitised by KASAN?

The kernel maps memory in a number of different parts of the address space. This poses something of a problem for KASAN, which requires that all addresses accessed by instrumented code have a valid shadow region.

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.

5.4.1 By default

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 in to the module allocator, KASAN can temporarily map 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.

5.4.2 CONFIG_KASAN_VMALLOC

With `CONFIG_KASAN_VMALLOC`, KASAN can cover vmalloc space at the cost of greater memory usage. Currently this is only supported on x86.

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_SHADOW_SCALE_SIZE * PAGE_SIZE`.

Instead, we share backing space across multiple mappings. We allocate 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.

We hook in to the vmap infrastructure to lazily clean up unused shadow memory.

To avoid the difficulties around swapping mappings around, we expect 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.

CHAPTER

SIX

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].

6.1 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
================================================================================
```

6.2 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.

6.3 References

CHAPTER

SEVEN

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. Kmemleak is supported on x86, arm, arm64, powerpc, sparc, sh, microblaze, mips, s390, nds32, arc and xtensa.

7.1 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.

7.2 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().

7.3 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
```

7.4 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
```

7.5 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`

7.6 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.

7.7 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.

7.8 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 bult-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
```

(continues on next page)

(continued from previous page)

```
    [<000000007c873fa6>] entry_SYSCALL_64_after_hwframe+0x44/0xa9
...
```

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

CHAPTER

EIGHT

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.

8.1 Usage

KCSAN requires Clang 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).

8.1.1 Error reports

A typical data race report looks like this:

```
==================================================================
BUG: KCSAN: data-race in generic_permission / kernfs_refresh_inode

write to 0xffff8fee4c40700c of 4 bytes by task 175 on cpu 4:
 kernfs_refresh_inode+0x70/0x170
 kernfs_iop_permission+0x4f/0x90
 inode_permission+0x190/0x200
 link_path_walk.part.0+0x503/0x8e0
 path_lookupat.isra.0+0x69/0x4d0
 filename_lookup+0x136/0x280
 user_path_at_empty+0x47/0x60
 vfs_statx+0x9b/0x130
 __do_sys_newlstat+0x50/0xb0
 __x64_sys_newlstat+0x37/0x50
 do_syscall_64+0x85/0x260
 entry_SYSCALL_64_after_hwframe+0x44/0xa9

read to 0xffff8fee4c40700c of 4 bytes by task 166 on cpu 6:
 generic_permission+0x5b/0x2a0
 kernfs_iop_permission+0x66/0x90
 inode_permission+0x190/0x200
```

(continues on next page)

(continued from previous page)

```
 link_path_walk.part.0+0x503/0x8e0
 path_lookupat.isra.0+0x69/0x4d0
 filename_lookup+0x136/0x280
 user_path_at_empty+0x47/0x60
 do_faccessat+0x11a/0x390
 __x64_sys_access+0x3c/0x50
 do_syscall_64+0x85/0x260
 entry_SYSCALL_64_after_hwframe+0x44/0xa9

Reported by Kernel Concurrency Sanitizer on:
CPU: 6 PID: 166 Comm: systemd-journal Not tainted 5.3.0-rc7+ #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.12.0-1 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.

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

```
==================================================================
BUG: KCSAN: data-race in e1000_clean_rx_irq+0x551/0xb10

race at unknown origin, with read to 0xffff933db8a2ae6c of 1 bytes by␣
↪interrupt on cpu 0:
 e1000_clean_rx_irq+0x551/0xb10
 e1000_clean+0x533/0xda0
 net_rx_action+0x329/0x900
 __do_softirq+0xdb/0x2db
 irq_exit+0x9b/0xa0
 do_IRQ+0x9c/0xf0
 ret_from_intr+0x0/0x18
 default_idle+0x3f/0x220
 arch_cpu_idle+0x21/0x30
 do_idle+0x1df/0x230
 cpu_startup_entry+0x14/0x20
 rest_init+0xc5/0xcb
 arch_call_rest_init+0x13/0x2b
 start_kernel+0x6db/0x700

Reported by Kernel Concurrency Sanitizer on:
CPU: 0 PID: 0 Comm: swapper/0 Not tainted 5.3.0-rc7+ #2
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.12.0-1 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 can occur either due to missing instrumentation or e.g. DMA accesses. These reports will only be generated if `CONFIG_KCSAN_REPORT_RACE_UNKNOWN_ORIGIN=y` (selected by default).

8.1.2 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.
- 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.

8.1.3 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 re-

porting 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.

8.1.4 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/`.

8.2 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.

8.2.1 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.), but is oblivious of any ordering guarantees and simply assumes that memory barriers are placed correctly. In other words, KCSAN assumes that as long as a plain access is not observed to race with another conflicting access, memory operations are correctly ordered.

This means that KCSAN will not report potential data races due to missing memory ordering. Developers should therefore carefully consider the required memory ordering requirements that remain unchecked. If, however, missing memory ordering (that is observable with a particular compiler and architecture) leads to an observable data race (e.g. entering a critical section erroneously), KCSAN would report the resulting data race.

8.3 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(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(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(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);
}
```

Note

ASSERT_EXCLUSIVE_ACCESS_SCOPED(), if applicable, performs more thorough checking if a clear scope where no concurrent accesses are expected exists.

For cases where the object is freed, KASAN is a better fit to detect use-after-free bugs.

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(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:

8.4 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.

8.4.1 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 not explicitly aware of the LKMM's 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).

8.5 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

NINE

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.

9.1 Requirements

- gdb 7.2+ (recommended: 7.4+) with python support enabled (typically true for distributions)

9.2 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
```

9.3 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
```

(continues on next page)

(continued from previous page)

```
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:

```
(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>,
```

(continues on next page)

(continued from previous page)

```
  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"
}
```

9.4 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

TEN

USING KGDB, KDB AND THE KERNEL DEBUGGER INTERNALS

Author Jason Wessel

10.1 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.

10.2 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.

10.2.1 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 to 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
```

10.2.2 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
```

10.3 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.

10.3.1 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.

10.3.2 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
```

10.3.3 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.

10.3.4 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.

10.3.5 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.

10.3.6 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.

10.4 Using kdb

10.4.1 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.

10.4.2 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.

10.5 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.

10.5.1 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 remotebaud 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.

10.6 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.

10.6.1 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.

10.6.2 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)
```

10.7 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.

10.8 Kernel Debugger Internals

10.8.1 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 `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 initalization.

Parameters

void no arguments

Description

This function will handle the initalization 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 initalization.

Parameters

void no arguments

Description

This function will handle the late initalization 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.

10.8.2 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 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,
[...]
};
```

10.9 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

ELEVEN

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.

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.

11.1 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
```

11.2 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.

11.3 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.

11.4 Install selftests

You can use the kselftest_install.sh tool to install selftests in the default location, which is tools/testing/selftests/kselftest, or in a user specified location.

To install selftests in default location:

```
$ cd tools/testing/selftests
$ ./kselftest_install.sh
```

To install selftests in a user specified location:

```
$ cd tools/testing/selftests
$ ./kselftest_install.sh install_dir
```

11.5 Running installed selftests

Kselftest install as well as the Kselftest tarball provide 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
$ ./run_kselftest.sh
```

11.6 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 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
```

11.7 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.

11.8 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

11.9 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/kselftest/kselftest_module.h`
- `tools/testing/kselftest/kselftest/module.sh`

11.9.1 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
```

11.9.2 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();

/*
```

(continues on next page)

(continued from previous page)

```
 * 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");
```

11.9.3 Example test script

```
#!/bin/bash
# SPDX-License-Identifier: GPL-2.0+
$(dirname $0)/../kselftest/module.sh "foo" test_foo
```

11.10 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.

11.10.1 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;
```

(continues on next page)

(continued from previous page)

```
  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
```

11.10.2 Helpers

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_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_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(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)
```

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_name)
Called once per fixture to setup the data and register

Parameters

fixture_name fixture name

Description

```
FIXTURE(datatype 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_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_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_name)
Optionally called once per fixture to declare fixture variant

Parameters

fixture_name fixture name

Description

```
FIXTURE_VARIANT(datatype name) {
  type property1;
  ...
};
```

Defines type of constant parameters provided to `FIXTURE_SETUP()` and `TEST_F()` as variant. Variants allow the same tests to be run with different arguments.

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_ADD(datatype 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(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.

Warning: use of ASSERT_* here will skip TEARDOWN.

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.

11.10.3 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(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_EQ(expected, measured): expected == measured

ASSERT_NE(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_NE(expected, measured): expected != measured

ASSERT_LT(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_LT(expected, measured): expected < measured

ASSERT_LE(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_LE(expected, measured): expected <= measured

ASSERT_GT(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_GT(expected, measured): expected > measured

ASSERT_GE(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_GE(expected, measured): expected >= measured

ASSERT_NULL(seen)

Parameters

seen measured value

Description

ASSERT_NULL(measured): NULL == measured

ASSERT_TRUE(seen)

Parameters

seen measured value

Description

ASSERT_TRUE(measured): measured != 0

ASSERT_FALSE(seen)

Parameters

seen measured value

Description

ASSERT_FALSE(measured): measured == 0

ASSERT_STREQ(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_STREQ(expected, measured): !strcmp(expected, measured)

ASSERT_STRNE(expected, seen)

Parameters

expected expected value

seen measured value

Description

ASSERT_STRNE(expected, measured): strcmp(expected, measured)

EXPECT_EQ(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_EQ(expected, measured): expected == measured

EXPECT_NE(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_NE(expected, measured): expected != measured

EXPECT_LT(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_LT(expected, measured): expected < measured

EXPECT_LE(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_LE(expected, measured): expected <= measured

EXPECT_GT(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_GT(expected, measured): expected > measured

EXPECT_GE(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_GE(expected, measured): expected >= measured

EXPECT_NULL(seen)

Parameters

seen measured value

Description

EXPECT_NULL(measured): NULL == measured

EXPECT_TRUE(seen)

Parameters

seen measured value

Description

EXPECT_TRUE(measured): 0 != measured

EXPECT_FALSE(seen)

Parameters

seen measured value

Description

EXPECT_FALSE(measured): 0 == measured

EXPECT_STREQ(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_STREQ(expected, measured): !strcmp(expected, measured)

EXPECT_STRNE(expected, seen)

Parameters

expected expected value

seen measured value

Description

EXPECT_STRNE(expected, measured): strcmp(expected, measured)

CHAPTER

TWELVE

KUNIT - UNIT TESTING FOR THE LINUX KERNEL

12.1 Getting Started

12.1.1 Installing dependencies

KUnit has the same dependencies as the Linux kernel. As long as you can build the kernel, you can run KUnit.

12.1.2 Running tests with the KUnit Wrapper

Included with KUnit is a simple Python wrapper which runs tests under User Mode Linux, and formats the test results.

The wrapper can be run with:

```
./tools/testing/kunit/kunit.py run --defconfig
```

For more information on this wrapper (also called kunit_tool) check out the kunit_tool How-To page.

Creating a .kunitconfig

If you want to run a specific set of tests (rather than those listed in the KUnit defconfig), you can provide Kconfig options in the `.kunitconfig` file. This file essentially contains the regular Kernel config, with the specific test targets as well. The `.kunitconfig` should also contain any other config options required by the tests.

A good starting point for a `.kunitconfig` is the KUnit defconfig:

```
cd $PATH_TO_LINUX_REPO
cp arch/um/configs/kunit_defconfig .kunitconfig
```

You can then add any other Kconfig options you wish, e.g.:

```
CONFIG_LIST_KUNIT_TEST=y
```

kunit_tool will ensure that all config options set in `.kunitconfig` are set in the kernel `.config` before running the tests. It'll warn you if you haven't included the dependencies of the options you're using.

Note: Note that removing something from the `.kunitconfig` will not trigger a rebuild of the `.config` file: the configuration is only updated if the `.kunitconfig` is not a subset of `.config`. This means that you can use other tools (such as make menuconfig) to adjust other config options.

Running the tests (KUnit Wrapper)

To make sure that everything is set up correctly, simply invoke the Python wrapper from your kernel repo:

```
./tools/testing/kunit/kunit.py run
```

Note: You may want to run `make mrproper` first.

If everything worked correctly, you should see the following:

```
Generating .config ...
Building KUnit Kernel ...
Starting KUnit Kernel ...
```

followed by a list of tests that are run. All of them should be passing.

Note: Because it is building a lot of sources for the first time, the `Building KUnit kernel` step may take a while.

12.1.3 Running tests without the KUnit Wrapper

If you'd rather not use the KUnit Wrapper (if, for example, you need to integrate with other systems, or use an architecture other than UML), KUnit can be included in any kernel, and the results read out and parsed manually.

Note: KUnit is not designed for use in a production system, and it's possible that tests may reduce the stability or security of the system.

Configuring the kernel

In order to enable KUnit itself, you simply need to enable the `CONFIG_KUNIT` Kconfig option (it's under Kernel Hacking/Kernel Testing and Coverage in menuconfig). From there, you can enable any KUnit tests you want: they usually have config options ending in `_KUNIT_TEST`.

KUnit and KUnit tests can be compiled as modules: in this case the tests in a module will be run when the module is loaded.

Running the tests (w/o KUnit Wrapper)

Build and run your kernel as usual. Test output will be written to the kernel log in TAP format.

Note: It's possible that there will be other lines and/or data interspersed in the TAP output.

12.1.4 Writing your first test

In your kernel repo let's add some code that we can test. Create a file `drivers/misc/example.h` with the contents:

```
int misc_example_add(int left, int right);
```

create a file `drivers/misc/example.c`:

```
#include <linux/errno.h>

#include "example.h"

int misc_example_add(int left, int right)
{
        return left + right;
}
```

Now add the following lines to `drivers/misc/Kconfig`:

```
config MISC_EXAMPLE
        bool "My example"
```

and the following lines to `drivers/misc/Makefile`:

```
obj-$(CONFIG_MISC_EXAMPLE) += example.o
```

Now we are ready to write the test. The test will be 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)
```

(continues on next page)

(continued from previous page)

```
{
        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);
```

Now add the following to `drivers/misc/Kconfig`:

```
config MISC_EXAMPLE_TEST
        bool "Test for my example"
        depends on MISC_EXAMPLE && KUNIT
```

and the following to `drivers/misc/Makefile`:

```
obj-$(CONFIG_MISC_EXAMPLE_TEST) += example-test.o
```

Now add it to your `.kunitconfig`:

```
CONFIG_MISC_EXAMPLE=y
CONFIG_MISC_EXAMPLE_TEST=y
```

Now you can 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!

12.1.5 Next Steps

- Check out the Using KUnit page for a more in-depth explanation of KUnit.

12.2 Using KUnit

The purpose of this document is to describe what KUnit is, how it works, how it is intended to be used, and all the concepts and terminology that are needed to understand it. This guide assumes a working knowledge of the Linux kernel and some basic knowledge of testing.

For a high level introduction to KUnit, including setting up KUnit for your project, see Getting Started.

12.2.1 Organization of this document

This document is organized into two main sections: Testing and Isolating Behavior. The first covers what unit tests are and how to use KUnit to write them. The second covers how to use KUnit to isolate code and make it possible to unit test code that was otherwise un-unit-testable.

12.2.2 Testing

What is KUnit?

"K" is short for "kernel" so "KUnit" is the "(Linux) Kernel Unit Testing Framework." KUnit is intended first and foremost for writing unit tests; it is general enough that it can be used to write integration tests; however, this is a secondary goal. KUnit has no ambition of being the only testing framework for the kernel; for example, it does not intend to be an end-to-end testing framework.

What is Unit Testing?

A unit test is a test that tests code at the smallest possible scope, a unit of code. In the C programming language that's a function.

Unit tests should be written for all the publicly exposed functions in a compilation unit; so that is all the functions that are exported in either a class (defined below) or all functions which are **not** static.

Writing Tests

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 a function to be tested 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, so 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 is a way to specify that you 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 of the 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 fail, the test case will fail.

It is important to understand that a test case fails when any expectation is violated; however, the test will continue running, potentially trying 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 expectations supported by KUnit, see Test API.

Note: A single test case should be pretty short, pretty easy to understand, focused on a single behavior.

For example, if we wanted to properly test the add function above, we would create additional tests cases which would each test a different property that an add function should have like this:

```
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));
}
```

Notice how it is immediately obvious what all the properties that we are testing for are.

Assertions

KUnit also has the concept of an assertion. An assertion is just like an expectation except the assertion immediately terminates the test case if it is not satisfied.

For example:

```
static void mock_test_do_expect_default_return(struct kunit *test)
{
        struct mock_test_context *ctx = test->priv;
        struct mock *mock = ctx->mock;
        int param0 = 5, param1 = -5;
        const char *two_param_types[] = {"int", "int"};
        const void *two_params[] = {&param0, &param1};
        const void *ret;

        ret = mock->do_expect(mock,
                              "test_printk", test_printk,
                              two_param_types, two_params,
                              ARRAY_SIZE(two_params));
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ret);
        KUNIT_EXPECT_EQ(test, -4, *((int *) ret));
}
```

In this example, the method under test should return a pointer to a value, so if the pointer returned by the method is null or an errno, we don't want to bother

continuing 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 have not been satisfied to complete the test.

Test Suites

Now obviously one unit test isn't very helpful; the power comes from having many test cases covering all of a unit's behaviors. Consequently 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 just a collection of test cases for a unit of code with a set up function that gets invoked before every test case and then a tear down function that gets invoked after every test case completes.

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`, would 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 would have `example_test_exit` called immediately after it. `kunit_test_suite(example_test_suite)` registers the test suite with the KUnit test framework.

Note: A test case will only be run if it is associated with a test suite.

For more information on these types of things see the Test API.

12.2.3 Isolating Behavior

The most important aspect of unit testing that other forms of testing do not provide is the ability to limit the amount of code under test to a single unit. In practice, this is only possible by being able to control what code gets run when the unit under test calls a function and this is usually accomplished through some sort of indirection 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 primarily comes from two constructs, classes, structs that contain function pointers that are 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, pretty much 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. In order for it to truly be a class, the function pointers must specify that a pointer to the class, known as a class handle, be one of the parameters; this makes it possible for the member functions (also known as methods) to have access to member variables (more commonly known as fields) allowing the same implementation to have multiple instances.

Typically a class can be overridden by child classes by embedding the parent class in the child class. Then when a method provided by the child class is called, the child implementation knows that the pointer passed to it is of a parent contained within the child; because of this, 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 shape, parent);

        return self->length * self->width;
};
```

(continues on next page)

(continued from previous page)

```
void rectangle_new(struct rectangle *self, int length, int width)
{
        self->parent.area = rectangle_area;
        self->length = length;
        self->width = width;
}
```

In this example (as in most kernel code) the operation of 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 just provides an implementation of a piece of code that is different than what runs in a production instance, but behaves identically from the standpoint of the callers; this is usually done to replace a dependency that is hard to deal with, or is slow.

A good example for this might be implementing a fake EEPROM that just stores the "contents" in an internal buffer. For example, let's 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 some 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 easily 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)
```

(continues on next page)

(continued from previous page)

```
{
        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);
}
```

(continues on next page)

(continued from previous page)

```
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;
```

(continues on next page)

(continued from previous page)

```
        return 0;
}

static void eeprom_buffer_test_exit(struct kunit *test)
{
        struct eeprom_buffer_test *ctx = test->priv;

        destroy_eeprom_buffer(ctx->eeprom_buffer);
}
```

12.2.4 KUnit on non-UML architectures

By default KUnit uses UML as a way to provide dependencies for code under test. Under most circumstances KUnit's usage of UML should be treated as an implementation detail of how KUnit works under the hood. Nevertheless, there are instances where being able to run architecture specific code or test against real hardware is desirable. For these reasons KUnit supports running on other architectures.

Running existing KUnit tests on non-UML architectures

There are some special considerations when running existing KUnit tests on non-UML architectures:

- Hardware may not be deterministic, so a test that always passes or fails when run under UML may not always do so on real hardware.
- Hardware and VM environments may not be hermetic. KUnit tries its best to provide a hermetic environment to run tests; however, it cannot manage state that it doesn't know about outside of the kernel. Consequently, tests that may be hermetic on UML may not be hermetic on other architectures.
- Some features and tooling may not be supported outside of UML.
- Hardware and VMs are slower than UML.

None of these are reasons not to run your KUnit tests on real hardware; they are only things to be aware of when doing so.

The biggest impediment will likely be that certain KUnit features and infrastructure may not support your target environment. For example, at this time the KUnit Wrapper (`tools/testing/kunit/kunit.py`) does not work outside of UML. Unfortunately, there is no way around this. Using UML (or even just a particular architecture) allows us to make a lot of assumptions that make it possible to do things which might otherwise be impossible.

Nevertheless, all core KUnit framework features are fully supported on all architectures, and using them is straightforward: all you need to do is to take your kunitconfig, your Kconfig options for the tests you would like to run, and merge them into whatever config your are using for your platform. That's it!

For example, let's say you have the following kunitconfig:

```
CONFIG_KUNIT=y
CONFIG_KUNIT_EXAMPLE_TEST=y
```

If you wanted to run this test on an x86 VM, you might add the following config options to your `.config`:

```
CONFIG_KUNIT=y
CONFIG_KUNIT_EXAMPLE_TEST=y
CONFIG_SERIAL_8250=y
CONFIG_SERIAL_8250_CONSOLE=y
```

All these new options do is enable support for a common serial console needed for logging.

Next, you could build a kernel with these tests as follows:

```
make ARCH=x86 olddefconfig
make ARCH=x86
```

Once you have built a kernel, you could run it on QEMU as follows:

```
qemu-system-x86_64 -enable-kvm \
                   -m 1024 \
                   -kernel arch/x86_64/boot/bzImage \
                   -append 'console=ttyS0' \
                   --nographic
```

Interspersed in the kernel logs you might see the following:

```
TAP version 14
        # Subtest: example
        1..1
        # example_simple_test: initializing
        ok 1 - example_simple_test
ok 1 - example
```

Congratulations, you just ran a KUnit test on the x86 architecture!

In a similar manner, kunit and kunit tests can also be built as modules, so if you wanted to run tests in this way you might add the following config options to your `.config`:

```
CONFIG_KUNIT=m
CONFIG_KUNIT_EXAMPLE_TEST=m
```

Once the kernel is built and installed, a simple

```
modprobe example-test
```

…will run the tests.

Writing new tests for other architectures

The first thing you must do is ask yourself whether it is necessary to write a KUnit test for a specific architecture, and then whether it is necessary to write that test for a particular piece of hardware. In general, writing a test that depends on having access to a particular piece of hardware or software (not included in the Linux source repo) should be avoided at all costs.

Even if you only ever plan on running your KUnit test on your hardware configuration, other people may want to run your tests and may not have access to your hardware. If you write your test to run on UML, then anyone can run your tests without knowing anything about your particular setup, and you can still run your tests on your hardware setup just by compiling for your architecture.

Important: Always prefer tests that run on UML to tests that only run under a particular architecture, and always prefer tests that run under QEMU or another easy (and monetarily free) to obtain software environment to a specific piece of hardware.

Nevertheless, there are still valid reasons to write an architecture or hardware specific test: for example, you might want to test some code that really belongs in `arch/some-arch/*`. Even so, try your best to write the test so that it does not depend on physical hardware: if some of your test cases don't need the hardware, only require the hardware for tests that actually need it.

Now that you have narrowed down exactly what bits are hardware specific, the actual procedure for writing and running the tests is pretty much the same as writing normal KUnit tests. One special caveat is that you have to reset hardware state in between test cases; if this is not possible, you may only be able to run one test case per invocation.

12.2.5 KUnit debugfs representation

When kunit test suites are initialized, they create an associated directory in `/sys/kernel/debug/kunit/<test-suite>`. The directory contains one file

- results: "cat results" displays results of each test case and the results of the entire suite for the last test run.

The debugfs representation is primarily of use when kunit test suites are run in a native environment, either as modules or builtin. Having a way to display results like this is valuable as otherwise results can be intermixed with other events in dmesg output. The maximum size of each results file is KUNIT_LOG_SIZE bytes (defined in `include/kunit/test.h`).

12.3 kunit_tool How-To

12.3.1 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.

12.3.2 What is a kunitconfig?

It's just a defconfig that kunit_tool looks for in the base directory. 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.

12.3.3 How do I use 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.

If you just want to use the defconfig that ships with the kernel, you can append the `--defconfig` flag as well:

```
./tools/testing/kunit/kunit.py run --timeout=30 --jobs=`nproc --all` --
↪defconfig
```

Note: This command is particularly helpful for getting started because it just works. No kunitconfig needs to be present.

For a list of all the flags supported by kunit_tool, you can run:

```
./tools/testing/kunit/kunit.py run --help
```

12.4 API Reference

12.4.1 Test API

This file documents all of the standard testing API excluding mocking or mocking related features.

struct **kunit_resource**
represents a test managed resource

Definition

```
struct kunit_resource {
  void *allocation;
  kunit_resource_free_t free;
};
```

Members

allocation for the user to store arbitrary data.

free a user supplied function to free the resource. Populated by `kunit_alloc_resource()`.

Description

Represents a test managed resource, a resource which will automatically be cleaned up at the end of a test case.

```
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->allocation = kmalloc(params->size, params->gfp);

        if (!res->allocation)
                return -ENOMEM;

        return 0;
}

static void kunit_kmalloc_free(struct kunit_resource *res)
{
        kfree(res->allocation);
}

void *kunit_kmalloc(struct kunit *test, size_t size, gfp_t gfp)
{
        struct kunit_kmalloc_params params;
        struct kunit_resource *res;

        params.size = size;
        params.gfp = gfp;
```

(continues on next page)

(continued from previous page)

```
        res = kunit_alloc_resource(test, kunit_kmalloc_init,
                kunit_kmalloc_free, &params);
        if (res)
                return res->allocation;

        return NULL;
}
```

Example

struct **kunit_case**
    represents an individual test case.

Definition

```
struct kunit_case {
  void (*run_case)(struct kunit *test);
  const char *name;
};
```

Members

run_case the function representing the actual test case.

name the name of the test case.

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(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.

struct **kunit_suite**
describes a related collection of `struct kunit_case`

Definition

```
struct kunit_suite {
  const char name[256];
  int (*init)(struct kunit *test);
  void (*exit)(struct kunit *test);
  struct kunit_case *test_cases;
};
```

Members

name the name of the test. Purely informational.

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(suites_list)
used to register one or more `struct kunit_suite` with KUnit.

Parameters

suites_list a statically allocated list of `struct kunit_suite`.

Description

Registers **suites_list** with the test framework. See `struct kunit_suite` for more information.

When builtin, KUnit tests are all run as late_initcalls; this means that they cannot test anything where tests must run at a different init phase. One significant restriction resulting from this is that KUnit cannot reliably test anything that is initialize in the late_init phase; another is that KUnit is useless to test things that need to be run in an earlier init phase.

An alternative is to build the tests as a module. Because modules do not support multiple late_initcall()s, we need to initialize an array of suites for a module.

TODO(brendanhiggins**google.com**): Don't run all KUnit tests as late_initcalls. I have some future work planned to dispatch all KUnit tests from the same place, and at the very least to do so after everything else is definitely initialized.

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.

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 each 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, const void * res,
void * match_data)
Match a resource with the same instance.

Parameters

struct kunit * test Test case to which the resource belongs.

const void * res The data stored in kunit_resource->allocation.

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**.

int **kunit_resource_destroy**(struct kunit * test, kunit_resource_match_t match, kunit_resource_free_t free, 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**.

kunit_resource_free_t free Must match free on the kunit_resource to free.

void * match_data Data passed into **match**.

Description

Free the latest kunit_resource of **test** for which **free** matches the kunit_resource_free_t associated with the resource and for which **match** returns true.

Return

0 if kunit_resource is found and freed, -ENOENT if not found.

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

Just like kmalloc(…), 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_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()` for more information.

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(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(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(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(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(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(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(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(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(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(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(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(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(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(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(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(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_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(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(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(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(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(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(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(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(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(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(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(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(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_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.

This section documents the KUnit kernel testing API. It is divided into the following sections:

Test API	documents all of the standard testing API excluding mocking or mocking related features.

12.5 Frequently Asked Questions

12.5.1 How is this different from Autotest, kselftest, etc?

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, hence the name. A unit test should be the finest granularity of testing and as such should allow all possible code paths to be tested in the code under test; this is only possible if the code under test is very 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 VM and all 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.

12.5.2 Does KUnit support running on architectures other than UML?

Yes, well, mostly.

For the most part, the KUnit core framework (what you 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 some infrastructure, like the KUnit Wrapper (`tools/testing/kunit/kunit.py`) that does not support other architectures.

In short, this means that, yes, you can run KUnit on other architectures, but it might require more work than using KUnit on UML.

For more information, see KUnit on non-UML architectures.

12.5.3 What is the difference between a unit test and these 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, hence the name. A unit test should be the finest granularity of testing and as such should allow all possible code paths to be tested in the code under test; this is only possible if the code under test is very 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.

12.5.4 KUnit isn't working, what should I do?

Unfortunately, there are a number of things which can break, but here are some things to try.

1. Try running `./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 (e.g., 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 run 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 Using KUnit.

If none of the above tricks help, you are always welcome to email any issues to kunit-dev@googlegroups.com.

12.6 What is KUnit?

KUnit is a lightweight unit testing and mocking framework for the Linux kernel.

KUnit is heavily inspired by JUnit, Python's unittest.mock, and Googletest/Googlemock for C++. KUnit provides facilities for defining unit test cases, grouping related test cases into test suites, providing common infrastructure for running tests, and much more.

KUnit consists of a kernel component, which provides a set of macros for easily writing unit tests. Tests written against KUnit will run on kernel boot if built-in, or when loaded if built as a module. These tests write out results to the kernel log in TAP format.

To make running these tests (and reading the results) easier, KUnit offers kunit_tool, which builds a User Mode Linux kernel, runs it, and parses the test results. This provides a quick way of running KUnit tests during development, without requiring a virtual machine or separate hardware.

Get started now: Getting Started

12.7 Why KUnit?

A unit test is supposed to test a single unit of code in isolation, hence the name. A unit test should be the finest granularity of testing and as such should allow all possible code paths to be tested in the code under test; this is only possible if the code under test is very small and does not have any external dependencies outside of the test's control like hardware.

KUnit provides a common framework for unit tests within the kernel.

KUnit tests can be run on most architectures, and most tests are architecture independent. All built-in KUnit tests run on kernel startup. Alternatively, KUnit

and KUnit tests can be built as modules and tests will run when the test module is loaded.

Note: KUnit can also run tests without needing a virtual machine or actual hardware under User Mode Linux. User Mode Linux is a Linux architecture, like ARM or x86, which compiles the kernel as a Linux executable. KUnit can be used with UML either by building with `ARCH=um` (like any other architecture), or by using kunit_tool.

KUnit is fast. Excluding build time, from invocation to completion KUnit can run several dozen tests in only 10 to 20 seconds; this might not sound like a big deal to some people, but having such fast and easy to run tests fundamentally changes the way you go about testing and even writing code in the first place. Linus himself said in his git talk at Google:

> "…a lot of people seem to think that performance is about doing the same thing, just doing it faster, and that is not true. That is not what performance is all about. If you can do something really fast, really well, people will start using it differently."

In this context Linus was talking about branching and merging, but this point also applies to testing. If your tests are slow, unreliable, are difficult to write, and require a special setup or special hardware to run, then you wait a lot longer to write tests, and you wait a lot longer to run tests; this means that tests are likely to break, unlikely to test a lot of things, and are unlikely to be rerun once they pass. If your tests are really fast, you run them all the time, every time you make a change, and every time someone sends you some code. Why trust that someone ran all their tests correctly on every change when you can just run them yourself in less time than it takes to read their test log?

12.8 How do I use it?

- Getting Started - for new users of KUnit
- Using KUnit - for a more detailed explanation of KUnit features
- API Reference - for the list of KUnit APIs used for testing
- kunit_tool How-To - for more information on the kunit_tool helper script
- Frequently Asked Questions - for answers to some common questions about KUnit