

1. Here is the experimental prototype of the Van unit test system.

⟨Header files 4⟩
 ⟨The main program 16⟩

2. License

The GNU General Public License (GPL) is designed for engineers shipping products with GPL-licensed software:

SPDX-License-Identifier: GPL-2.0

Van unit test system.

Copyright (C) 2022 Gerald Schueller <gerald.schueller@web.de>

3. Header file list

<code>os.h</code>	Van Operating system: <code>printf()</code> .
<code>stddef.h</code>	Standard type definitions: <code>offsetof()</code> .
<code>stdbool.h</code>	C99 allows bool. true is 1 and false is 0.
<code>errno.h</code>	OS interface errors like <code>ENOMEM=12</code> or <code>EFAULT=14</code> .
<code>limits.h</code>	Sizes of integral or integer types: <code>INT_MIN</code> .

4. We include declarations from Van OS - `os.h` - and Unix. Above there is some evidence, why we need these interfaces with contents hints.

⟨Header files 4⟩ ≡
`#include "os.h"`
`#include <stddef.h>`
`#include <stdbool.h>`
`#include <errno.h>`
`#include <limits.h>`

This code is used in section 1.

5. This constant defines the van unit prompt.

`#define P "U>"`

6. Architectures might want to move the poison pointer offset into some well-recognized area such as `0xdead000000000000`, that is also not mappable by user-space exploits.

`#define POISON_POINTER_DELTA 0`

7. These are non-NULL pointers that will result in page faults under normal circumstances, used to verify that nobody uses non-initialized list entries.

`#define LIST_POISON1 ((void *) #100 + POISON_POINTER_DELTA)`
`#define LIST_POISON2 ((void *) #200 + POISON_POINTER_DELTA)`

8. The `noreturn` function specifier is used to tell to the compiler that the function will not return anything. If the program uses some return statement inside it, the compiler will generate compile time error.

`#define __noreturn __attribute__((__noreturn__))`

9. The compiler supports the ability to diagnose when the results of a function call expression are discarded under suspicious circumstances. A diagnostic is generated when a function or its return type is marked with the macro below.

`#define __must_check __attribute__((__warn_unused_result__))`

10. The counter saturates at `REFCOUNT_SATURATED` and will not move once there. This avoids wrapping the counter and causing ‘spurious’ use-after-free bugs.

```
#define REFCOUNT_SATURATED (INT_MIN/2)
```

11. This constant defines the size of a test log line in *struct kunit* { ...log; ... } .

```
#define KUNIT_LOG_SIZE 512
```

12. This constant defines the size of the kunit status_comment element in *struct kunit* { ...status_comment; ... } .

```
#define KUNIT_STATUS_COMMENT_SIZE 256
```

13. This constant defines the size of the test parameter buffer in *kunit_run_tests()*, to store them in *struct kunit* { ...log; ... } .

```
#define KUNIT_PARAM_DESC_SIZE 128
```

14. TAP - [Test Anything Protocol](#) - specifies subtest stream indentation of 4 spaces, 8 spaces for a sub-subtest. See the “Subtests” section at the inet page [TAP](#)

```
#define KUNIT_SUBTEST_INDENT "    "
```

```
#define KUNIT_SUBSUBTEST_INDENT "        "
```

15. This constant saves the source location of a line of code, see *struct kunit_loc*.

```
#define KUNIT_CURRENT_LOC { .file = __FILE__ , .line = __LINE__ }
```

16. Now we come to the layout of the *main* function.

⟨The main program 16⟩ ≡

```
void main(void)
{
    printf("Hello.\n");
}
```

This code is used in section 1.

17. printk infrastructure

Some subsystems have their own custom printk that applies a va_format to a generic format, for example, to include a device number or other metadata alongside the format supplied by the caller.

In order to store these in the way they would be emitted by the printk infrastructure, the subsystem provides us with the start, fixed string, and any subsequent text in the format string.

We take a variable argument list as pr_fmt/dev_fmt/etc are sometimes passed as multiple arguments (eg: “%s: ”, “blah”), and we must only take the first one.

subsys_fmt_prefix must be known at compile time, or compilation will fail (since this is a mistake). If fmt or level is not known at compile time, no index entry will be made (since this can legitimately happen).

```
#define __printk_index_emit(...) do { } while (0)
#define printk_index_wrap(_p_func, _fmt, ...) (
{
    __printk_index_emit(_fmt, 1, 1);
    _p_func(_fmt, ##__VA_ARGS__);
}
)
```

18. printk()

`printk()` - print a kernel message.

fmt: format string.

...: arguments for the format string.

This is `printk()`. It can be called from any context. We want it to work.

If `printk` indexing is enabled, `_printk()` is called from `printk_index_wrap`. Otherwise, `printk` is simply #defined to `_printk`.

We try to grab the `console.lock`. If we succeed, it's easy - we log the output and call the console drivers. If we fail to get the semaphore, we place the output into the log buffer and return. The current holder of the `console_sem` will notice the new output in `console_unlock()`; and will send it to the consoles before releasing the lock.

One effect of this deferred printing is that code which calls `printk()` and then changes `console_loglevel` may break. This is because `console_loglevel` is inspected when the actual printing occurs.

See also:

`printf(3)`

See the `vsnprintf()` documentation for format string extensions over C99.

```
#define printk(fmt, ...) printk_index_wrap(_printk, fmt, ##__VA_ARGS__)
#define pr_fmt(fmt) "VUnit:_" fmt
#define pr_info(fmt, ...) printk(pr_fmt(fmt), ##__VA_ARGS__)
```

19. `WARN()`, `WARN_ON()` and so on can be used to report significant kernel issues that need prompt attention if they should ever appear at runtime.

Do not use these macros when checking for invalid external inputs.

```
#define __WARN_printf(arg ...) do
{
    fprintf(stderr, arg);
}
while (0)
#define WARN(condition, format ...) (
{
    int __ret_warn_on = !!(condition);
    if (unlikely(__ret_warn_on))
        printf(format);
    unlikely(__ret_warn_on);
}
)
#define WARN_ONCE(condition, format ...) WARN(condition, format)
#define WARN_ON(condition) (
{
    int __ret_warn_on = !!(condition);
    if (unlikely(__ret_warn_on))
        __WARN_printf("assertion failed at %s:%d\n",
            __FILE__, __LINE__);
    unlikely(__ret_warn_on);
}
)
```

20. Don't use `BUG()` or `BUG_ON()` unless there's really no way out; one example might be detecting data structure corruption in the middle of an operation that can't be backed out of. If the (sub)system can somehow continue operating, perhaps with reduced functionality, it's probably not BUG-worthy.

If you're tempted to `BUG()`, think again: is completely giving up really the **only** solution? There are usually better options, where users don't need to reboot ASAP and can mostly shut down cleanly.

```
#define BUG_ON(cond) OS_TRAP_IF(¬(cond))
#define BUG() BUG_ON(1)
```

21. Comparing types.

Are two types/vars the same type (ignoring qualifiers)?

```
#define __same_type(a, b) __builtin_types_compatible_p(typeof(a), typeof(b))
```

22. Multiplication with overflow.

```
#define check_mul_overflow(a, b, d) __must_check_overflow((
    {
        typeof(a) __a = (a);
        typeof(b) __b = (b);
        typeof(d) __d = (d);
        (void)(&__a ≡ &__b);
        (void)(&__a ≡ __d);
        __builtin_mul_overflow(__a, __b, __d);
    }
))
```

23. The `volatile` is due to gcc bugs: see [Extended Asm](#)

With extended asm you can read and write C variables from assembler and perform jumps from assembler code to C labels. Extended asm syntax uses colons (:) to delimit the operand parameters after the assembler template. See [Assembly with C](#)

In it find recommendations: [Barrier Sync](#)

The first barriers does nothing at runtime. It's called a SW barrier. The second barrier would translate into a HW barrier, probably a fence (mfence/sfence) operations if you're on x86. This instruction tells the micro code of a CPU network to make sure that loads or stores can't pass this point and must be observed in the correct side of the sync point.

```
#define barrier() __asm__ __volatile__ ( "" : : "memory" )
```

24. Prevent the compiler from merging or refetching reads or writes. The compiler is also forbidden from reordering successive instances of `READ_ONCE` and `WRITE_ONCE`, but only when the compiler is aware of some particular ordering. One way to make the compiler aware of ordering is to put the two invocations of `READ_ONCE` or `WRITE_ONCE` in different C statements.

These two macros will also work on aggregate data types like structs or unions. If the size of the accessed data type exceeds the word size of the machine (e.g., 32 bits or 64 bits) `READ_ONCE()` and `WRITE_ONCE()` will fall back to `memcpy` and print a compile-time warning.

Their two major use cases are: (1) Mediating communication between process-level code and irq/NMI handlers, all running on the same CPU, and (2) Ensuring that the compiler does not fold, spindle, or otherwise mutilate accesses that either do not require ordering or that interact with an explicit memory barrier or atomic instruction that provides the required ordering.

```
#define READ_ONCE(x) ( {
    union {
        typeof(x) __val;
        char __c[1];
    } __u =
    { . __c = {0} } ;
    __read_once_size(&(x), __u.__c, sizeof (x));
    __u.__val;
} )

#define WRITE_ONCE(x, val) ( {
    union {
        typeof(x) __val;
        char __c[1];
    } __u =
    { . __val = (val) } ;
    __write_once_size(&(x), __u.__c, sizeof (x));
    __u.__val;
} )
```

25. Print a refcount warning like "underflow; use-after-free".

```
#define REFCOUNT_WARN(str) WARN_ONCE(1, "refcount_t:␣" str ".\n")
```

26. Built-in Function: *long __builtin_expect (long exp, long c)*See [Builtin_Expect](#)

You may use `__builtin_expect` to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this (`-fprofile-arcs`), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

3The return value is the value of `exp`, which should be an integral expression. The semantics of the built-in are that it is expected that `exp == c`. For example:

```
if (__builtin_expect(x, 0))
    foo()
```

indicates that we do not expect to call `foo`, since we expect `x` to be zero. Since you are limited to integral expressions for `exp`, you should use constructions such as

```
if (__builtin_expect(ptr != NULL, 1))
    foo(*ptr)
```

when testing pointer or floating-point values.

For the purposes of branch prediction optimizations, the probability that a `__builtin_expect` expression is true is controlled by GCC's `builtin-expected-probability` parameter, which defaults to 90

You can also use `__builtin_expect_with_probability` to explicitly assign a probability value to individual expressions. If the built-in is used in a loop construct, the provided probability will influence the expected number of iterations made by loop optimizations.

27. The attributes `likely` and `unlikely` may be applied to labels and statements (other than declaration-statements). They may not be simultaneously applied to the same label or statement.

- 1 Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are more likely than any alternative path of execution that does not include such a statement.
- 2 Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are less likely than any alternative path of execution that does not include such a statement.

```
#define unlikely(x) __builtin_expect(¬¬(x), 0)
```

28. `static_assert` - check integer constant expression at build time

`static_assert()` is a wrapper for the C11 `_Static_assert`, with a little macro magic to make the message optional (defaulting to the stringification of the tested expression).

Contrary to `BUILD_BUG_ON()`, `static_assert()` can be used at global scope, but requires the expression to be an integer constant expression (i.e., it is not enough that `__builtin_constant_p()` is true for `expr`).

Also note that `BUILD_BUG_ON()` fails the build if the condition is true, while `static_assert()` fails the build if the expression is false.

```
#define static_assert(expr, ...) __static_assert(expr, ##__VA_ARGS__, #expr)
#define __static_assert(expr, msg, ...) _Static_assert(expr, msg)
```

29. `container_of` - cast a member of a structure out to the containing structure.

`ptr`: the pointer to the member.
`type`: the type of the container struct this is embedded in.
`member`: the name of the member within the struct.

```
#define container_of(ptr, type, member) ( {
    void *__mptr = (void *) (ptr);
    static_assert ( __same_type (*(ptr), ( ( type * ) 0 ) - member ) ∨
        __same_type (*(ptr), void),
        "pointer_type_mismatch_in_container_of()" );
    ( ( type * ) (__mptr - offsetof(type, member)) ); }
```

30. `list_entry` - get the struct for this entry.

`ptr`: the `&struct list_head` pointer.
`type`: the type of the struct this is embedded in.
`member`: the name of the `list_head` within the struct.

```
#define list_entry(ptr, type, member)
    container_of(ptr, type, member)
```

31. `list_prev_entry` - get the prev element in list.

`pos`: the type `*` to cursor.
`member`: the name of the `list_head` within the struct.

```
#define list_prev_entry(pos, member)
    list_entry((pos)→member.prev, typeof(*(pos)), member)
```

32. `list_next_entry` - get the next element in list.

`pos`: the type `*` to cursor.
`member`: the name of the `list_head` within the struct.

```
#define list_next_entry(pos, member)
    list_entry((pos)→member.next, typeof(*(pos)), member)
```

33. `list_last_entry` - get the last element from a list.

`ptr`: the list head to take the element from.
`type`: the type of the struct this is embedded in.
`member`: the name of the `list_head` within the struct.

```
#define list_last_entry(ptr, type, member)
    list_entry((ptr)→prev, type, member)
```

34. `list_first_entry` - get the first element from a list.

`ptr`: the list head to take the element from.
`type`: the type of the struct this is embedded in.
`member`: the name of the `list_head` within the struct.

```
#define list_first_entry(ptr, type, member)
    list_entry((ptr)→next, type, member)
```

35. `list_entry_is_head` - test if the entry points to the head of the list.

`pos`: the type `*` to cursor.
`head`: the head for your list.
`member`: the name of the `list_head` within the struct.

```
#define list_entry_is_head(pos, head, member)
    (&pos→member ≡ (head))
```

36. `list_for_each_entry` - iterate over list of given type.

`pos`: the type `*` to cursor.
`head`: the head for your list.
`member`: the name of the `list_head` within the struct.

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
        ¬list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

37. `list_for_each_entry_reverse` - iterate backwards over list of given type.

`pos:` the type * to cursor.
`head:` the head for your list.
`member:` the name of the list_head within the struct.

```
#define list_for_each_entry_reverse(pos, head, member)
    for (pos = list_last_entry(head, typeof(*pos), member);
        ¬list_entry_is_head(pos, head, member);
        pos = list_prev_entry(pos, member))
```

38. `list_for_each_entry_safe` - iterate over list of given type safe against removal of list entry.

`pos:` the type * to use as a loop cursor.
`n:` another type * to use as temporary storage.
`head:` the head for your list.
`member:` the name of the list_head within the struct.

```
#define list_for_each_entry_safe(pos, n, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member),
        n = list_next_entry(pos, member);
        ¬list_entry_is_head(pos, head, member);
        pos = n, n = list_next_entry(n, member))
```


39. Index. Here is the list of the identifiers used.

[__a:](#) [22](#).
[__asm__:](#) [23](#).
[__attribute__:](#) [8](#), [9](#).
[__b:](#) [22](#).
[__builtin_expect:](#) [27](#).
[__builtin_mul_overflow:](#) [22](#).
[__builtin_types_compatible_p:](#) [21](#).
[__c:](#) [24](#).
[__d:](#) [22](#).
[__FILE__:](#) [15](#), [19](#).
[__LINE__:](#) [15](#), [19](#).
[__mptr:](#) [29](#).
[__must_check:](#) [9](#).
[__must_check_overflow:](#) [22](#).
[__noreturn:](#) [8](#).
[__noreturn__:](#) [8](#).
[__printk_index_emit:](#) [17](#).
[__read_once_size:](#) [24](#).
[__ret_warn_on:](#) [19](#).
[__same_type:](#) [21](#), [29](#).
[__static_assert:](#) [28](#).
[__u:](#) [24](#).
[__VA_ARGS__:](#) [17](#), [18](#), [28](#).
[__val:](#) [24](#).
[__volatile__:](#) [23](#).
[__WARN_printk:](#) [19](#).
[__warn_unused_result__:](#) [9](#).
[__write_once_size:](#) [24](#).
[_fmt:](#) [17](#).
[_p_func:](#) [17](#).
[_printk:](#) [18](#).
[_Static_assert:](#) [28](#).
[arg:](#) [19](#).
[barrier:](#) [23](#).
[BUG:](#) [20](#).
[BUG_ON:](#) [20](#).
[check_mul_overflow:](#) [22](#).
[cond:](#) [20](#).
[condition:](#) [19](#).
[container_of:](#) [29](#), [30](#).
[expr:](#) [28](#).
[file:](#) [15](#).
[fmt:](#) [18](#).
[format:](#) [19](#).
[fprintf:](#) [19](#).
[head:](#) [35](#), [36](#), [37](#), [38](#).
[INT_MIN:](#) [10](#).
[KUNIT_CURRENT_LOC:](#) [15](#).
[KUNIT_LOG_SIZE:](#) [11](#).
[KUNIT_PARAM_DESC_SIZE:](#) [13](#).
[KUNIT_STATUS_COMMENT_SIZE:](#) [12](#).
[KUNIT_SUBSUBTEST_INDENT:](#) [14](#).
[KUNIT_SUBTEST_INDENT:](#) [14](#).
[list_entry:](#) [30](#), [31](#), [32](#), [33](#), [34](#).
[list_entry_is_head:](#) [35](#), [36](#), [37](#), [38](#).
[list_first_entry:](#) [34](#), [36](#), [38](#).
[list_for_each_entry:](#) [36](#).
[list_for_each_entry_reverse:](#) [37](#).
[list_for_each_entry_safe:](#) [38](#).
[list_last_entry:](#) [33](#), [37](#).
[list_next_entry:](#) [32](#), [36](#), [38](#).
[LIST_POISON1:](#) [7](#).
[LIST_POISON2:](#) [7](#).
[list_prev_entry:](#) [31](#), [37](#).
[main:](#) [16](#).
[member:](#) [29](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#).
[msg:](#) [28](#).
[next:](#) [32](#), [34](#).
[OS_TRAP_IF:](#) [20](#).
[P:](#) [5](#).
[POISON_POINTER_DELTA:](#) [6](#), [7](#).
[pos:](#) [31](#), [32](#), [35](#), [36](#), [37](#), [38](#).
[pr_fmt:](#) [18](#).
[pr_info:](#) [18](#).
[prev:](#) [31](#), [33](#).
[printf:](#) [16](#), [19](#).
[printk:](#) [18](#).
[printk_index_wrap:](#) [17](#), [18](#).
[ptr:](#) [29](#), [30](#), [33](#), [34](#).
[READ_ONCE:](#) [24](#).
[REFCOUNT_SATURATED:](#) [10](#).
[REFCOUNT_WARN:](#) [25](#).
[static_assert:](#) [28](#).
[stderr:](#) [19](#).
[str:](#) [25](#).
[type:](#) [29](#), [30](#), [33](#), [34](#).
[typeof:](#) [21](#), [22](#), [24](#), [31](#), [32](#), [36](#), [37](#), [38](#).
[unlikely:](#) [19](#), [27](#).
[val:](#) [24](#).
[WARN:](#) [19](#).
[WARN_ON:](#) [19](#).
[WARN_ONCE:](#) [19](#), [25](#).
[WRITE_ONCE:](#) [24](#).

⟨ Header files [4](#) ⟩ Used in section [1](#).

⟨ The main program [16](#) ⟩ Used in section [1](#).

VU-MAIN

	Section	Page
Index	39	9