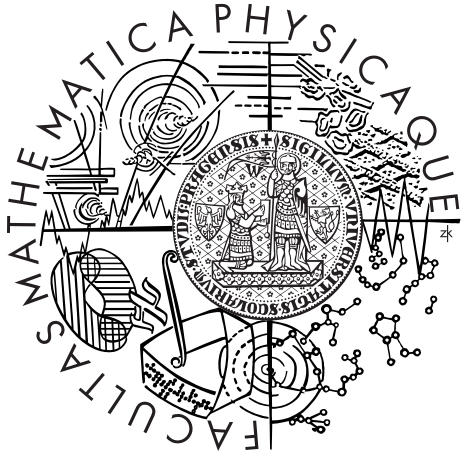
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Lukáš Lipavský

Linux kernel userspace modules

Department of Software Engineering
Supervisor: Mgr. Martin Děcký
Study program: Computer Science, Software Systems

I would like to thank my supervisor, Mgr. Martin Děcký, for his valuable feedback and many improvement proposals.

I would also like to thank my girlfriend and my parents for their support and unlimited patience.

I hereby certify that I wrote the thesis myself, using only the referenced sources. I give consent with lending the thesis.

Prague, August 7, 2008 Lukáš Lipavský

Contents

List of Figures

Název práce: Linux kernel userspace modules
Autor: Lukáš Lipavský
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Martin Děcký
e-mail vedoucího: Martin.Decky@mff.cuni.cz

Abstrakt: Tato práce navrhuje způsob implementace Linuxových ovladačů jakožto aplikací v uživatelském prostoru. Definice pevného rozhraní umožňujícího přístup k funkcím a strukturám jádra operačního systému byla v úvodní analýze zamítnuta z důvodu malé flexibility. Pokud aplikace vyžadovala funkcionalitu, která nebyla rozhraním poskytována, aplikace byla nucena implementovat vlastní modul jádra operačního systému pro její zpřístupnění. Navrhované řešení umožňuje dynamickou tvorbu rozhraní dle požadavků aplikace. Definice rozhraní je uložena v binární podobě (kódu), která je dynamicky zavedena do jádra operačního sytému. V kódu je také uložena definice funkcí, které se při zavedení začlení do jádra. Tyto funkce pak mohou být volány z ostatních funkcí jádra a realizovat například obsluhu přerušení. Formát kódu je nezávislý na architektuře počítače i na konfiguraci jádra operačního systému a je snadno přenositelný.

Navržené řešení umožňuje tvorbu ovladačů operačního systému Linux, které jsou implementovány jako uživatelské aplikace, aniž by ke své funkci vyžadovaly vlastní modul Linuxového jádra.
Klíčová slova: Linux, operační systém, jádro, ovladač

Title: Linux kernel userspace modules
Author: Lukáš Lipavský
Department: Department of Software Engineering
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: Martin.Decky@mff.cuni.cz

Abstract: This thesis proposes and implements a new method of implementing Linux kernel drivers in userspace applications – userspace drivers. Instead of proposing fixed interface for accessing kernel functionality from userspace application and using dedicated kernel module to implement functionality not accessible via the interface, proposed method allows applications to define custom interfaces to the kernel. The interface is defined in special bytecode that is loaded into the kernel. The bytecode also provides functions that can be called from the kernel and that work even in atomic context (interrupt handlers, etc.). The bytecode is architecture and kernel configuration independent.

The proposed method makes it possible to develop kernel drivers implemented in userspace applications without the need of dedicated kernel module.

Keywords: Linux, operating system, kernel, driver

Chapter 1

Introduction

The Linux kernel has a monolithic design. Although it has many advantages, it also implies several disadvantages. One of the disadvantages is that a device driver runs in the kernel and therefore has a limited access to some resources (libraries, etc.) and is potentially a threat to the system (a bug in the driver can panic the whole kernel). Differences between kernel modules and applications are described in [2].

The goal of the thesis is to design an interface or framework which would simplify the communication between userspace and kernel and therefore decrease the complexity of modules that use advantages of userspace applications. Part of the thesis should be (simple) proof-of-concept implementation of the solution.

1.1 Definition of userspace module

Userspace module is a functional unit which can be dynamically loaded into the kernel in a way similar to standard kernel module. The main difference between them is that userspace module has most of its functionality implemented in userspace.

The advantage of userspace module is that userspace module can use all libraries and development tools available to userspace applications, thus the implementation of the required functionality is simpler and less error-prone.

1.2 Goals and requirements

The functional limitations of userspace modules comparing to standard kernel modules should be minimal. There already exist other projects targeted to implement functionality in userspace, but their scope is limited to strictly defined areas (e.g. FUSE [1] for filesystem implementations or UIO [6] for simple device drivers). In contrary, userspace modules should be as general as possible.

Userspace modules should be easy to use in common Linux distribution. Since most of the functionality is implemented in userspace, it should be possible to use

the same userspace module on different kernels without recompilation[1].

Since userspace module is closer to userspace application than to kernel module, it should not be necessary to have complete kernel build environment to build userspace module. Therefore, it should be simple to build packages for various distributions without the usage of distribution-specific kernel module packages, which are often complicated. It should be even possible to compile userspace module with statically linked libraries, which will run on all Linux distributions with compatible kernel[2].

[1] It is not necessary to recompile `bash` for new kernel either.

[2] Functions and types used in the userspace modules exist in the kernel and have same syntax. For example if function `f(int)` is used in userspace module, it must exist in the kernel.

Chapter 2

Architecture of userspace module

2.1 Analysis of possible solutions

There are several possible ways (models) of implementing userspace modules. In this chapter, possible models are described and their advantages and disadvantages are discussed.

2.1.1 Generic interface model

The most obvious model is generic interface model. Model creates interface which provides access to picked kernel functions to userspace applications. The interface should be generic enough to provide support for generic userspace module. Since Linux kernel consists of many subsystems, interface generic enough to provide access to all of them would consist of hundreds of functions. Because every subsystem is different, generic interface for all of them would be quite inconsistent and complicated.

Due to this fact, it would be really hard (in reality almost impossible) to maintain the interface. The interface would probably also provide only limited generic functionality making development of more complicated userspace modules impossible.

Since whole userspace module implementation is in userspace process, it is not possible to do any out-of-process-context task such as interrupt handling.

This model is nearly impossible to design and even if implemented, it would have no remarkable advantages. It is mentioned for completeness only.

Advantages of the model:

- *(none)*

Disadvantages of the model:

- Unmaintainable.
- Inconsistent due to the inconsistency of different kernel subsystems.

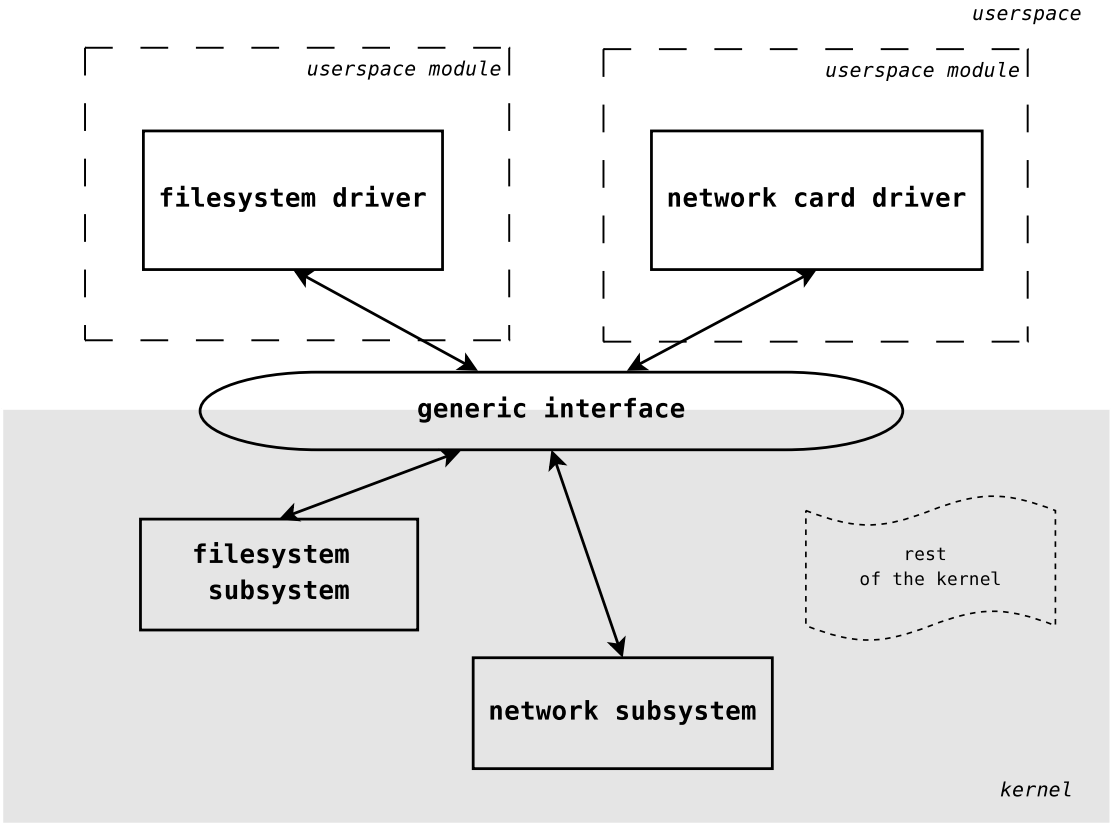


Figure 2.1: Generic interface model.

- Limited functionality – only generic functionality, impossible to run in atomic context, no access to kernel internals, etc.
- *(Unimplementable)*

2.1.2 Purpose-specific interface model

Generic interface model, described in chapter 2.1.1, is extremely complicated, inconsistent and unmaintainable. These problems can be solved by splitting the generic interface into several purpose-specific (subsystem-specific or functionality-specific) interfaces. Since purpose-specific interface provides access to the limited functionality area of the kernel, its functions can be designed to support advanced features.

Purpose-specific interface can be easily designed to be consistent since it covers limited (and usually logically related) functionality and multiple interfaces can overlap.

Though purpose-specific interface model solves all disadvantages which make generic interface model unusable, one disadvantage is preserved. Purpose-specific interface model does not provide a way to execute some parts in atomic or no-process context, access kernel internals, etc. However, many areas where this

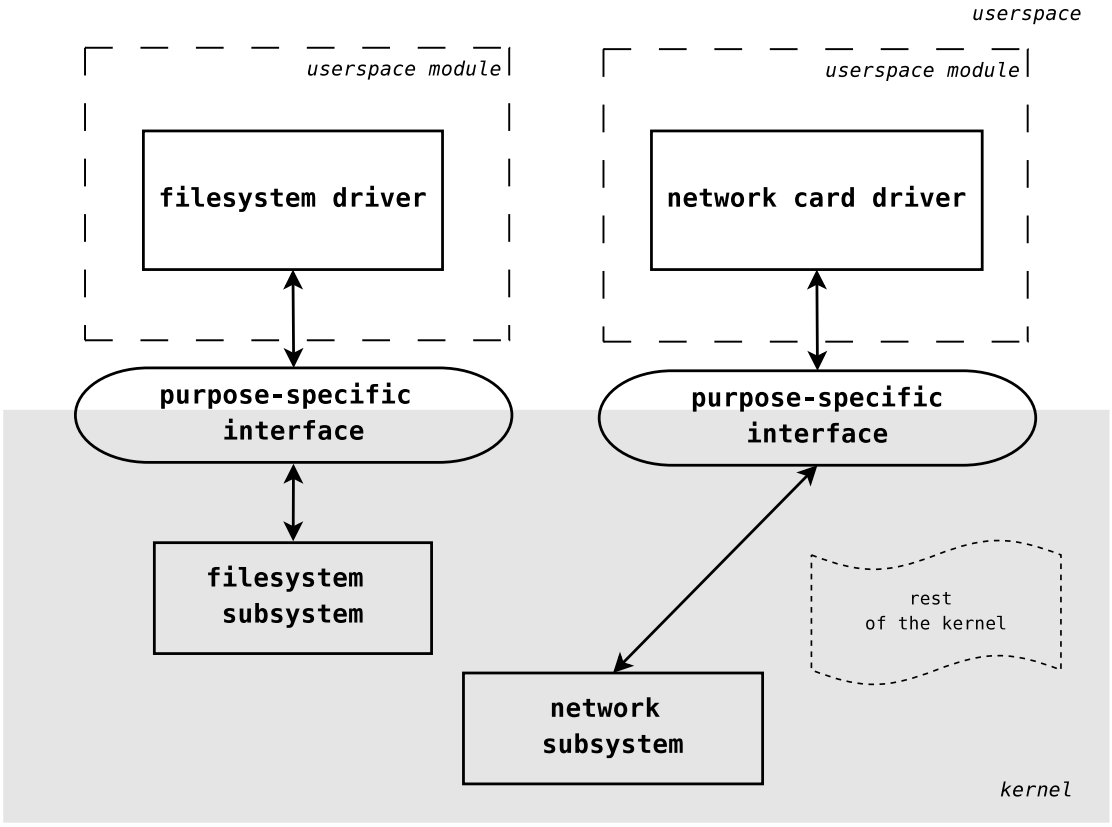


Figure 2.2: Purpose-specific interface model.

is not a problem exist. The functional limitation, though here mentioned as disadvantage, can be also considered to be an advantage from both security and reliability point of view.

If the functional limitation is not a problem in particular area, this model is the best possible solution since it is easy to use, reliable, secure and requires no new code in the kernel. One real example of purpose-specific interface is FUSE project [1], which provides interface for implementation of filesystem drivers in userspace.

Advantages of the model:

- Simple to use.
- Probably the best model when limited functionality is sufficient.

Disadvantages of the model:

- Limited functionality – impossible to run in atomic context, no access to kernel internals, etc.
- Limited by interface to specific area – not generic.

2.1.3 Purpose-specific interface model with kernel module

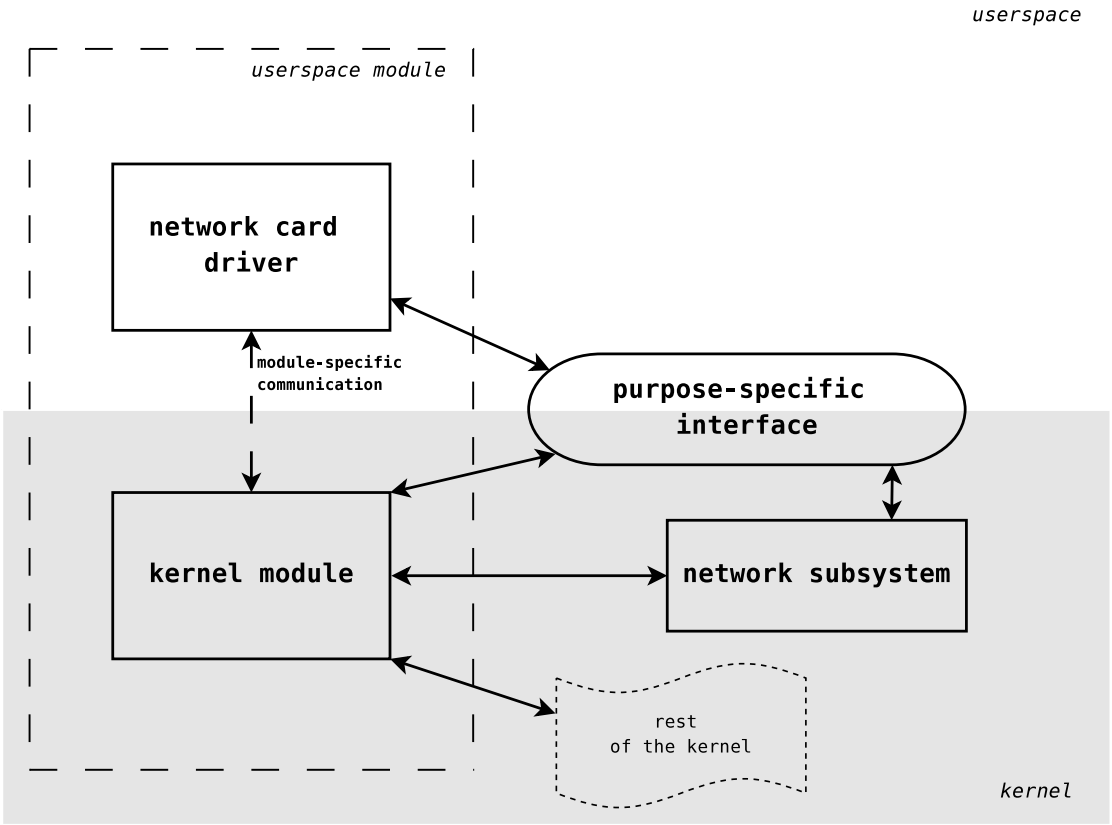


Figure 2.3: Purpose-specific interface model with kernel module.

It is often required to run some code in atomic context or to access some kernel internal structures which is not possible outside the kernel. Purpose-specific interface model can be extended to support this functionality by combining it with a kernel module. The module implements parts that cannot be done from userspace and the rest of the work is done in the same way as in purpose-specific interface model (in userspace). Kernel can for example handle interrupt and put data (received during the interrupt) to some memory area accessible through the interface to the userspace process.

Addition of kernel module makes the model able to do anything that usual kernel module can do. Disadvantage of this model is the need of a purpose-specific interface implementation. It is therefore not convenient to use this model for uncommon functionality[1].

This model is used by UIO [6], which provides interface for simple device drivers in userspace.

Advantages of the model:

[1] If no other implementation would use the interface.

- Simple to use.
- Kernel module runs in kernel space. It is therefore possible to run in atomic context and access kernel internals.

Disadvantages of the model:

- Scope is limited by interface or custom out-of-interface communication between module and userspace must be established.
- Kernel module is needed.[2].

2.1.4 Custom interface model

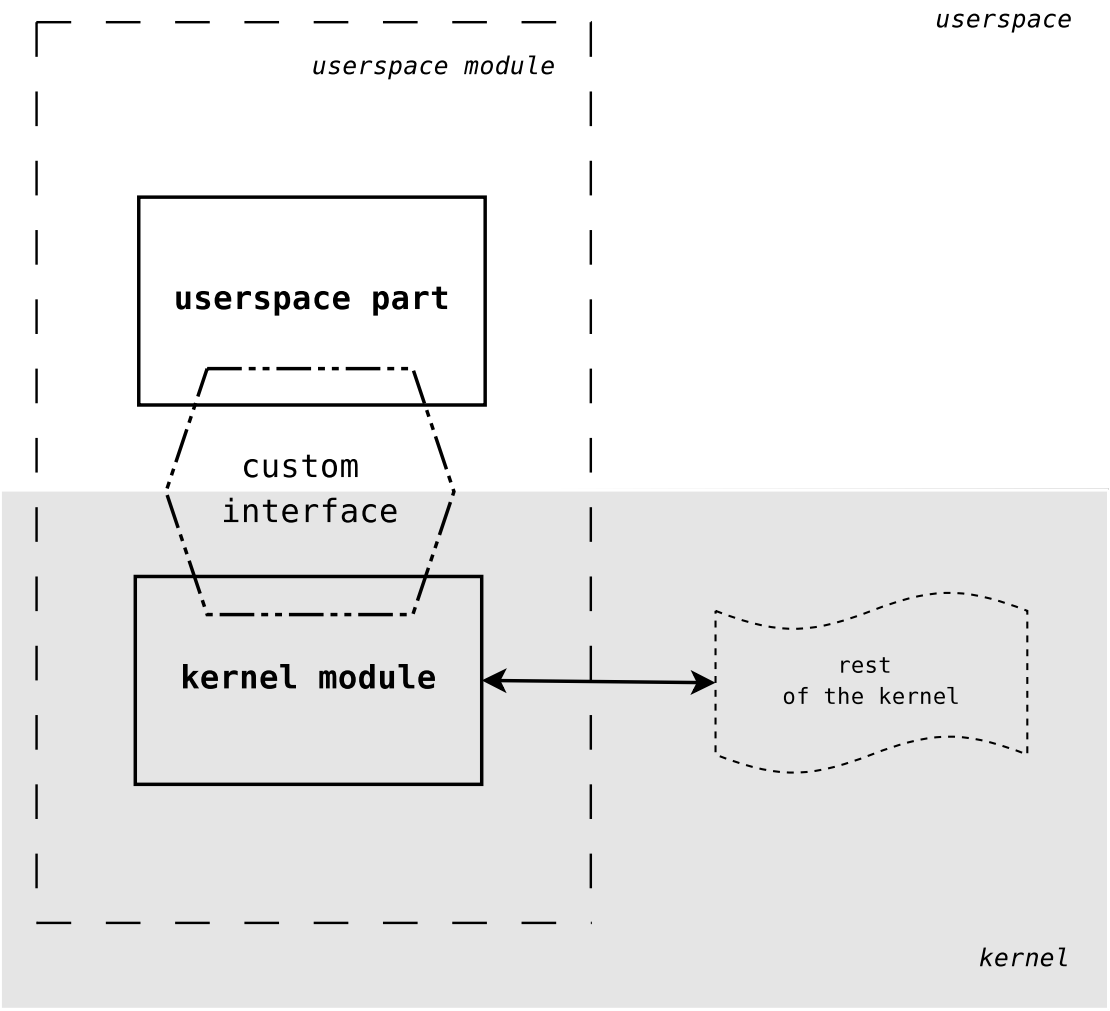


Figure 2.4: Custom interface model.

[2]Implies problems with compatibility across different distributions, kernel versions, etc.

The disadvantage of previous model is the fixed purpose-specific interface, which makes the model inconvenient for implementing modules relying on infrequently used features. Instead of relying on already implemented interfaces, the custom interfaces model makes definition of required (custom) interfaces between kernel module and userspace process possible.

There are several ways of implementing custom interface, but since kernel code is reactive by design (registered functions are called by the kernel as reaction to some event[3]), the interface should be designed to make calling of userspace functions from the kernel possible. Proposed realization of the interface is simplified RPC – userspace function call looks like standard function call from developer's perspective. The interface between kernel and userspace is implemented (or generated) in both kernel module and userspace process.

The model has full functionality of kernel modules (interrupt handling, accessing kernel structures, etc.) and it has simple access to userspace functions.

Advantages of the model:

- Full functionality of kernel module with additional userspace access.
- Simple to use.

Disadvantages of the model:

- Kernel module is needed[4].

2.1.5 Custom interface model with bytecode

The custom interface model can be modified by replacing kernel module by bytecode.

The disadvantage of using bytecode instead of kernel module is significant speed loss in bytecode part. However, if the bytecode part is minimal and the functionality is moved to the userspace, the impact is usually bearable[5].

The speed loss introduced by the addition of the bytecode is compensated by several advantages. The bytecode can be designed to be architecture, kernel version and kernel configuration independent. The only requirement is that actually used data and functions remain unchanged. Due to the bytecode independence, deployment of the userspace module is as simple as userspace application deployment. It is also possible to control the bytecode processing since various hooks can be integrated into bytecode interpreter.

Advantages of the model:

- Full functionality of kernel module with additional userspace access.
- Simple to use.

[3]For example when kernel need to fill page with data.

[4]Implies problems with compatibility across different distributions, kernel versions, etc.

[5]Except for time-critical modules.

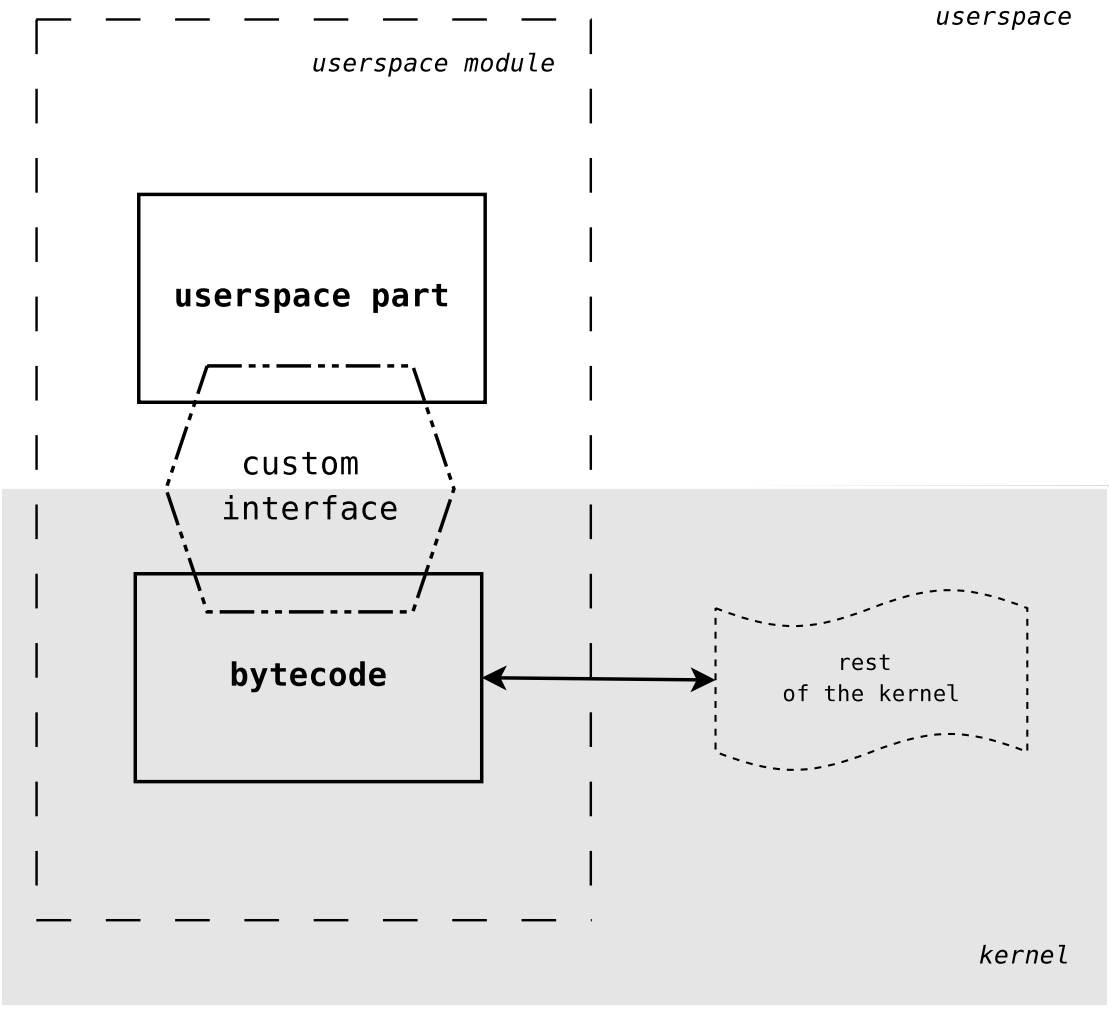


Figure 2.5: Custom interface model with bytecode.

- Kernel version and configuration independent[6].
- Bytecode architecture independent.
- Simple deployment for multiple distribution[7].
- Possibility to control bytecode execution.

Disadvantages of the model:

- Bytecode interpretation decreases speed.

[6]Compatible kernel is required.

[7]If userspace application is statically linked, it is possible to create completely distribution independent package.

2.2 Final requirements of userspace modules

Since custom interface model with bytecode is the model which mostly complies to the thesis requirements, as described in chapter 1.2 (and no implementation of the model exists), it will be taken as the model which will be implemented. Following chapters describe the design and implementation of the model together with all its advantages and limitations. Results of the thesis could be used for implementation of custom interface model (with kernel module) as well.

The model selection implies final requirements on the implementation:

1. Kernel functionality is moved to userspace, which allows usage of userpace libraries and tools.
2. Userspace module is kernel independent as long as the kernel is compatible (see chapter 1.2). This includes kernel configuration independence.
3. Complete kernel build environment is not required for userspace module compilation.
4. Bytecode that is loaded to the kernel is architecture independent.
5. It is possible to add restrictions to the kernel part for security reasons.

Chapter 3

Design of userspace module

This chapter describes components of userspace module and also describes how to create new userspace module.

3.1 Overview

Userspace module consists of two components: *userspace part* and *kernel part*.

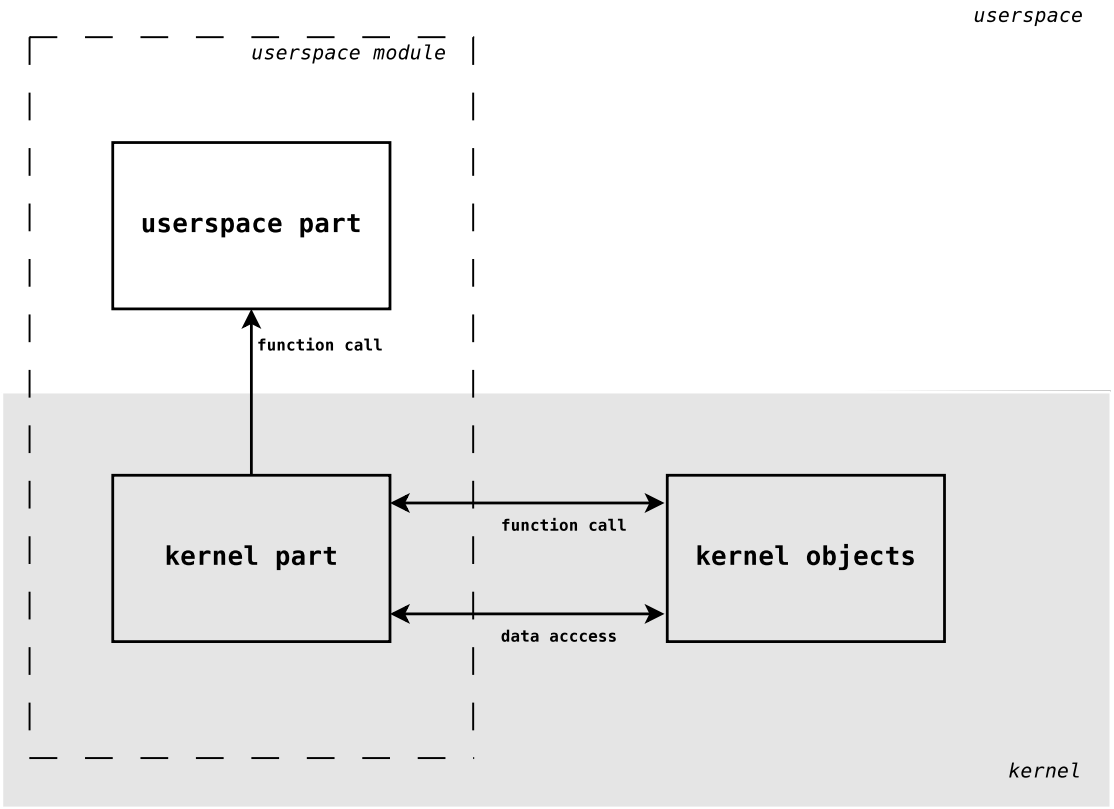


Figure 3.1: Userspace module architecture.

Userspace part is a process running in userspace which provides its functionality to kernel part. Since userspace part is running in userspace, it can use all resources available to userspace applications. By moving part of the code from the kernel into userspace, the result becomes easier to debug and, in case of error in userpace code, less dangerous to rest of the system, since errors in userspace are easier to handle.

Kernel part runs integrated in the kernel and has same power as normal kernel module. In special cases, it is possible to have userspace module which consist of kernel part only. Since one of the goals of kernel part is to be architecture and kernel configuration independent, bytecode representation of the kernel part is loaded to the kernel and is interpreted by simple interpreter. Since bytecode interpretation is slower than native code execution, kernel part should be as minimal as possible. Most of the work should be done in userspace part (which is natively compiled).

The interaction between kernel part and userspace part complies to client-server architecture. Userspace part provides its functions to the kernel part and no communication is initiated by userspace part. The interface between kernel part and userspace part is simplified remote procedure call. Call of userspace function has same syntax as normal function call.

3.2 Source Language

Since the kernel part must be able to seamlessly integrate into the kernel, it is feasible to have source language (SL) as similar to C language as possible. Therefore a subset of C has been chosen as source language for kernel part. The compatibility with C is so strict that it is mostly possible to compile SL with `gcc`[1]. The most important limitations are:

- No mixed assignment-operation expressions (such as `x += 2;`).
- No increment and decrement operators (such as `x++;`).
- Variables can be declared only global or at the start of the function. Variables local to a block are not allowed.
- It is not possible to declare multiple variables in one command (such as `int a, b;`).
- It is not possible to initialize variable in its declaration (such as `int a = 5;`).
- It is not possible to declare anonymous types (such as `struct { int a; } var;` or `typedef union { int a; char c; } un_type;`).

[1]The compiler contains small test suite which interprets the test using kernel part compiler/interpreter. Then it compiles the test with gcc and compares the output of the executable with the output produced by the interpreter.

- Function with variable argument's count (...) cannot be defined, though it is possible to use core kernel functions with variable argument's count (such as `printk`).

Most other parts of the C language syntax are available in SL (e.g. function declaration, recursion, etc.). Complete grammar of SL is defined in appendix A.

3.2.1 Types

SL has a limited number of built-in types:

- Types `char`, `short`, `int`, `long` and their unsigned variants. These types have unknown sizes at compile time, but kernel part makes sure that their sizes are exactly same as sizes of the respective types in C.
- Types `__u8`, `__u16`, `__u32`, `__u64` and `__s8`, `__s16`, `__s32`, `__s64`. These types have fixed sizes 1, 2, 4 and 8 bytes respectively.
- Type `void`.
- Type `intptr_t`, which is signed integer type with unknown size at compile time. This type needs to fulfill following condition: *For each pointer T *p:* `p == (T *)(intptr_t) p` *for each type T.*
- Type `uintptr_t`, which is unsigned variant of `intptr_type`.
- Type `userspace_data_t`, which is special opaque type used to transfer data between kernel and userspace. For details see chapter 4.4.3.
- Type `atomic_t` and `atomic_long_t` .
- Type `spinlock_t` .

Integer and pointer types in expressions are automatically converted to appropriate types according following rules:

1. From `void *` to `any_type *`.
2. From `any_type *` to `void *`.
3. From pointer type to integer type with warning.
4. From integer type to pointer type with warning.
5. From `type_1` to `type_2` if `type_1` and `type_2` are integer, `type_1` is signed if and only if `type_2` is signed and `sizeof(type_1) <= sizeof(type_2)`.
6. From `type_1` to `type_2` with warning message if `type1` and `type2` are integer but don't fulfil the rule above.

Note: Since sizes of some types are unknown at compile time, it is impossible to compare them. Warning is always displayed when autoconversion between such incomparable types occurs (with exception that `char` $\leq$ `short` $\leq$ `int` $\leq$ `long` and respectively for their unsigned variants).

If binary expression (such as `a + b`) has operands of different types, both operands are automatically converted to the result type prior to evaluation of the expression. Result type is determined according to following rules:

1. If one of the operands is pointer and the other is integer, result type is the same as the type of the pointer operand. The pointer must not be `void *`. (It is a standard pointer arithmetics[2])

2. If both operands have same size and are both signed or unsigned, the result type is the type of first operand.

3. If both operands have same size and one is signed and the other is unsigned, the result is the type of the signed operand and the warning is displayed.

4. If one operand is larger than the other and both are signed or unsigned, the result is the type of larger operand.

5. If one operand is larger than the other and one is signed and the other is unsigned, the result type is signed type with the size of the larger operand (it can be different type from the types of both operands). Warning is displayed.

If none of the rules above can be used, explicit typecast of operands is required.

Structures, unions and pointers are also supported besides basic types. Every type at runtime is binary same (sizes, offsets, alignement) as the same type in compiled C code. It is therefore possible to use SL code to modify kernel data and vice versa without the need to convert the data.

Since it is sometimes needed not to align data in structures, it is possible to define packed structure types:

```
struct unpacked_struct {
        int i;
        long l;
        __s8 signed_byte;
};
struct packed_struct {
        int i;
        long l;
        __s8 signed_byte;
} __attribute__((__packed__));
```

[2]Pointer difference (`p1 - p2`) is not implemented in SL though.

Adding the directive `__attribute__((__packed__))` after the structure type definition makes the structure members unaligned. For example size of unpacked_struct is 24 bytes on x86_64 architecture, while size of packed_struct is 13 bytes on x86_64 architecture.

There are several special operators in the SL: `sizeof`, `container_of` and `type_of`.

Operator `sizeof` has two possible formats, which differ in their operands: `sizeof(`*type*`)` and `sizeof(`*expression*`)`. In both formats it returns the size of its operand in bytes. It is important that the latter does *not* generate the code for the evaluation of the expression, only the result type is determined. Therefore the expression operand has no effect.

Usage example of `sizeof` operator:

```
int a;
int b;

a = sizeof(char);
b = sizeof(a = 5);
/* b == sizeof(int), but a == sizeof(char) !!! */
```

Operator `container_of` returns pointer to the wrapping type from pointer to its member. This simplify for example usage of kernel linked lists, etc. The format is `type_of_struct *container_of(pointer_to_member, type_of_struct, name_of_member)`.

Usage example of `container_of` operator:

```
struct my_list {
        int value;
        struct list_head list;
};
struct list_head *p_head;
struct my_list *p_mylist;
...
/* obtained p_head, need to set p_mylist */
p_mylist = container_of(p_head, struct my_list, list);
/* done */
```

Operator `typeof` is the compile-time operator which determines the type of the expression. The format is `typeof(`*expression*`)`. The operand can be used for standard variable definition (though it is not usual), but its main power is in usage together with C preprocessor macros. It is used for example for walking the linked lists in kernel, etc. It is important that the operator does *not* generate the code for the evaluation of the expression, only the result type is determined. Therefore the expression operand has no effect.

Usage example of operators `container_of` and `typeof`:

```
/*
 * From Linux kernel 2.6.25 /include/linux/list.h
 *
 * These macros demonstrate usage of container_of and typeof
 * operators for working with kernel lists.
 */

/**
 * 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_struct within the struct.
 */
 #define list_entry(ptr, type, member) \
   container_of(ptr, type, member)

/**
 * list_for_each_entry - iterate over list of given type
 * @pos:        the type * to use as a loop cursor.
 * @head:       the head for your list.
 * @member:     the name of the list_struct within the struct.
 */
 #define list_for_each_entry(pos, head, member)                \
   for (pos = list_entry((head)->next, typeof(*pos), member); \
     &pos->member != (head);                                  \
     pos = list_entry(pos->member.next, typeof(*pos), member))
```

3.2.2 Interface between Linux kernel and SL code

Interaction between kernel and SL code is designed to be as transparent as possible. Kernel functions and variables have to be declared in SL code before they can be used. Declaration of kernel function (variable) looks like declaration of normal function (variable) preceded by `extern` keyword.

Example of kernel objects declaration and access:

```
extern int printk(char *format, ...);
extern unsigned long jiffies;
...
printk("Now both function printk and variable jiffies "
       "(which is %ld now) can be used.\n", jiffies);
```

Some kernel functions and structures require pointer to function which will be later used as callback (e.g. timer handler). Since SL does not support function pointers it is advised to use `intptr_t` type. In fact any pointer type can be used,

but since `intptr_t` is rarely seen in the code, using it instead of common type such as `void *` makes the code easier to understand.

If SL function should be used as callback function, it must be declared as exported before its address is used. To export the function, command `EXPORT_FUNC(`*function*`);` needs to be specified in the SL code. There are no general limitations to exported function, any SL function can be exported this way.

Example of function exporting:

```
void timer_h(unsigned long data)
{
...
}
EXPORT_FUNC(timer_h);
...
timer->function = timer_h;
```

3.2.3 Compile condition

Many objects in the kernel are defined based on kernel configuration. Since kernel configuration is not known during SL compilation, usage of preprocessor (which is used during kernel compilation) is not sufficient. SL code implements compile conditions[3], which are used in a way similar to C preprocessor's `#if` statements. Compile conditions can be used to prune out some objects according to kernel configuration.

Possible compile conditions are:

- `@ifdef kernel_config_option`
- `@ifndef kernel_config_option`
- `@ifgt kernel_config_option number`
- `@ifge kernel_config_option number`
- `@iflt kernel_config_option number`
- `@ifle kernel_config_option number`
- `@ifeq kernel_config_option number`
- `@ifneq kernel_config_option number`

Compile condition can be used to test whether specified kernel configuration option is defined or, if the option is a number, to compare it with specified integer number. Compile conditions can be used to prune out following objects:

[3]Name is derived from AOT compilation which is done during bytecode loading. AOT compilation is described in chapter 4.3.

- function definition or declaration (of SL function),
- extern function declaration,
- member of structures or unions,
- statement.

If the compile condition is true during bytecode load, the following object is kept in the code, if condition is false, the object is pruned out.

If same function is defined multiple times with different compile conditions, all its definitions must have same return type and same argument types.

Example of compile conditions:

```
@ifndef CONFIG_DEBUG_LIST
void __list_add(struct list_head *new,
                struct list_head *prev,
                struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
@ifdef CONFIG_DEBUG_LIST
extern void __list_add(struct list_head *new,
                       struct list_head *prev,
                       struct list_head *next);

struct test {
    struct list_head l;
    int i;
    @ifdef CONFIG_MODULE_UNLOAD int can_unload;
};
struct test t;
...

@ifdef CONFIG_MODULE_UNLOAD t.can_unload = 1;
```

3.2.4 Interface between userspace and SL code

Interaction between kernel part and userspace part is only possible in direction from kernel part to userspace. SL code is able to call functions implemented by userspace part, that are declared in SL code. Declaration of userspace function looks like declaration of normal function preceded by `__user` keyword.

Example of userspace function usage:

```
__user int do_something(int x);
...
retval = do_something();
if(error_userspace(retval))
    /* handle error */
```

Communication between kernel and userspace is more complicated than communication between SL code and rest of the kernel. The kernel part is usually running in different context than userspace part. Since calling userspace functions implies context switch, *kernel part can call userspace functions only if kernel part is allowed to sleep.* In some special cases, kernel part runs in no context (e.g. as interrupt handler). In these cases, kernel part cannot call userspace functions. It is not possible to check this condition in compile-time. It is checked run-time when the userspace function is called[4].

Userspace functions must return value of type `int`. The returned values can be from whole range of type `int` except for `USERSPACE_ERROR` value, which is defined as minimal value of type `int` on the current architecture. It is therefore recommended to use return value mainly for success/error indication and transfer data as function arguments. SL code should always check the return value of userspace functions with function `userspace_error(int)` which returns `0` when call was successful and `1` when the call failed (result is `USERSPACE_ERROR`). The `USERSPACE_ERROR` means that error occurred in the calling of userspace function or the userspace part disconnected from the kernel communication channel. It is completely unrelated to the success or failure of action performed by userspace function.

Since userspace is not allowed to access kernel memory and kernel part is not able to access userspace memory because it runs in another context, it is not possible to transfer data between userspace and kernel space using pointers. To be able to pass data to and from userpace functions, following mechanism to transfer data is implemented:

1. No pointers are allowed as arguments of userspace functions.
2. If any structure or union argument (or referenced data – see following rule) contains a pointer, userspace is responsible not to use it and not to change it to unreasonable value (e.g. setting to `NULL` is often reasonable value).
3. If memory manipulation or access is necessary, special opaque type `userspace_data_t` must be used.

Arguments of the type `userspace_data_t` are created by functions `data_to_user(size_t data_size, void *data)`, `data_from_user(size_t size, void *data)` and `data_exchange_user(size_t data_size, void *data)`. Userspace function

[4]If the check fails, kernel `BUG()` is triggered.

call in SL code uses arguments of type `userspace_data_t`, but the userspace implementation receives a valid pointer.

Usage example of `userspace_data_t`:

```
/*
 * Function in userspace - pointer in definition
 */
int read_data(void *data, size_t size)
{
    ...
}

/*
 * Function call from SL code (kernel) - userspace_data_t
 * in declaration in SL code
 */
__user int read_data(userspace_data_t data, size_t size);

...
void *p;
size_t len;
int result;

p = buffer;
len = buffer_size;
/*
 * create userspace_data_t from pointer
 */
result = read_data(data_from_user(len, p), len);
if(error_userspace(result) {
    /* handle error */
    ...
}

/* data read now */
```

Since userspace part is not connected all the time, the kernel should be notified every time the userspace becomes available or unavailable. When userspace part is unavailable all userspace function calls return `USERSPACE_ERROR`.

If kernel part wants to be notified when userspace connects and disconnects it can implement functions `void userspace_connected()` and `void userspace_disconnected()`, which will be called on userspace connection and disconnection. In standard userspace module, disconnection of userspace part usually means error, so the `userspace_connected()` is called only once when the userspace part becomes ready. It is not possible to use notifications instead

of checking results with `error_userspace()`, since the userspace may become unavailable during the call of userspace function.

3.2.5 Special functions provided to SL code

Here is a list of functions that are exported by userspace modules kernel infrastructure to be used in SL. These functions must be declared as `extern` if SL code. Most of special functions are wrappers around special values set on kernel-compile time (e.g. `HZ` or `PAGE_SIZE`) or similar unaddressable information. Following functions are defined:

- Function `unsigned long get_HZ()`. Function returns the value of HZ which is often used when working with timers and jiffies in general.
- Function `size_t get_PAGE_SIZE()`. Function returns the value of PAGE_SIZE constant.

3.2.6 Kernel part structure

The structure of kernel part of userspace module is simple. It is any valid SL code, which contains functions `int init()` and `void exit()`. If kernel part wants to be notified on connection and disconnection of userspace part, SL code should also provide functions `void userspace_connected()` and `void userspace_disconnected()`.

Function `init()` is the first function that is executed when the kernel part is loaded. When this function is called the userspace part is not connected yet, so it is not possible to call userspace functions in `init()`. All initializations which include userspace function call should be moved into function `userspace_connected()`. The function `init()` returns zero on successful initialization of the kernel part and nonzero value on error. If nonzero value is returned, the function is responsible that the kernel part is safe to be unloaded (e.g. all memory has been freed, etc.).

Function `exit()` is a cleanup function called directly before kernel part is unloaded (if it has been successfully loaded – `init()` returned zero). Same as in `init()` it is not possible to call userspace functions from `exit()`. Function is responsible for making it safe to unload the kernel part (e.g. all memory has been freed, callback function has been unregistered, etc.).

Example of small kernel part which sets up a simple timer that prints a message to the system log every five seconds:

```
#include <usrmod.h>
/*
 * These are not linux kernel headers but are modified to be
 * compatible with SL.
 */
#include <linux/jiffies.h>
```

```
#include <linux/timer.h>
#include <linux/kernel.h>

#define TIMER_DELAY (5 * get_HZ())

struct timer_list t;

void timer_func(unsigned long data)
{
    unsigned long j;

    j = jiffies;
    printk("Timer called. Previous call at %ld, now %ld.\n",
           data, j);
    t.data = j;
    t.expires = j + TIMER_DELAY;
    add_timer(&t);
}

EXPORT_FUNC(timer_func);

int init()
{
    unsigned long j;

    j = jiffies;
    printk("Going to set up a timer that will log message every "
           "5 seconds.\n");
    init_timer(&t);
    t.function = timer_func;
    t.data = j;
    t.expires = j + TIMER_DELAY;
    add_timer(&t);
    return 0;
}

void exit()
{
    del_timer_sync(&t);
    printk("Stopping the timer that logs a message every "
           "5 seconds...\n");
}
```

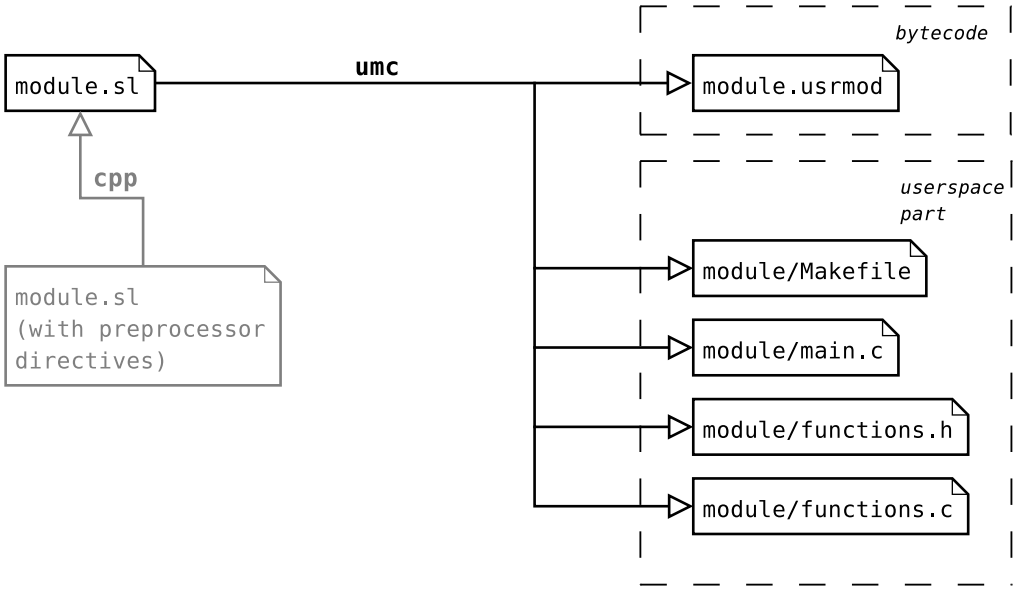


Figure 3.2: Userspace module compilation.

3.3 Compilation and userspace part code generation

SL code[5] is compiled into bytecode using compiler *umc* (userspace module compiler). Compiler produces several output files:

- Architecture and kernel-configuration independent bytecode for kernel part.
- Directory with same name as bytecode file (without `.usrmod` suffix). This directory contains generated C source files of userspace part:
  - `Makefile` or `Makefile.default` (if `Makefile` already exists).
  - `main.c`
  - `functions.h`
  - `functions.c` or `functions.c.skel` (if `functions.c` already exists).

The generated part can be directly compiled since all userspace functions from SL code are implemented (with `return 0;` in their bodies). Files `functions.h` and `main.c` should not be modified since all application functionality is implemented in them (including function data transfers between kernel part and userspace part). These files are always overwritten when SL code is compiled.

[5]No preprocessing of the SL code is done automatically. Since usage of C preprocessor directives makes the SL code more readable, it is advised to use them and run `cpp` on SL code before compilation.

File `Makefile` is a makefile for userspace part. It is safe to edit the makefile because it will not be overwritten during next SL code compilation. File `Makefile.default` will be created if `Makefile` already exists.

File `functions.c` contains generated bodies of all userspace functions defined in SL code. This file should be edited to implement required functionality of userspace part. If file `functions.c` already exists, it will not be overwritten during next SL code compilation. File `functions.c.skel` will be created if `functions.c` already exists.

Function `int process_arguments( int argc, char argv)`, which is called automatically when the userspace part starts, is also generated in file `functions.c`. It processes its arguments (command line arguments) and return `0` on success or any other value on error.

3.4 Loading userspace module into kernel

Two ways how to load userspace module exist. The first is realized by automatically generated userspace part, the other uses tool `usrmodctl`.

The userspace part source code generated from SL code is a full featured userspace module. Userspace part has following life-cycle:

1. Process command-line arguments.
2. Load bytecode from file specified in `-b` or `--bytecode` command-line argument. If not specified try to use file with same name as userspace module from local directory with suffix `.usrmod` (e.g. `testmod.usrmod`). If not found, exit with error.
3. Connect to the kernel part and process requests.
4. When `SIGINT` is received, unload kernel part and terminate.

There is also a low-level tool `usrmodctl` which is able to load kernel part. This tool is mainly used for special cases such as development and testing, but it can also be used to load userspace modules which have no userspace part. The tool can be used to load kernel part, unload kernel part and list currently loaded kernel parts. Usage of `usrmodctl`:

```
# Loads kernel part with given name
> usrmodctl add name path_to_bytecode

# Lists loaded userspace modules
> usrmodctl list

# Remove kernel part
> usrmodctl remove name
```

When kernel part is loaded by `usrmodctl` device name is returned as part of return message. This device is device the kernel part is connected to (and userspace part – if there is any – should connect to). For additional details about communication between userspace part and kernel part see chapter 4.4.3.

Chapter 4

Implementation

This chapter describes implementation details of userspace modules.

4.1 Bytecode and VM code

The first decision made for the kernel part of userspace modules implementation was the type of the bytecode interpreter. Due to the simplicity of both bytecode instruction set and interpreter implementation, stack machine model [8] has been selected. Since the interpreter is able to do direct memory addressing and call external functions, limitations of stack machine model are bypassed.

Source code in SL is compiled into the *bytecode*. The bytecode is architecture and kernel-configuration independent. This allows usage of userspace modules without the need to have kernel build environment set up. When the bytecode is loaded into kernel, it is compiled (*AOT*[1] *compiled*) into *virtual machine code* (vm code)[2]. The vm code is architecture dependant and contains absolute addresses instead of symbolic links to functions, variables and external objects.

Two compilations are needed for easy implementation of stack machine. The parser of the SL is quite complex, requires many memory allocations and deallocations, and the error recovery on syntax errors is often quite complicated. Putting such complex parser in the kernel would be hazardous. Implementing only AOT compilation in the kernel is less error-prone. Implementing the parser as normal application allows usage of parser generators which makes it possible to extend the programming language easily.

During the userspace modules implementation, several problems had to be solved. Solution to these problems are described in this chapter.

[1] Ahead-of-time [7].

[2] AOT compilation is done automatically by the kernel without userspace intervention

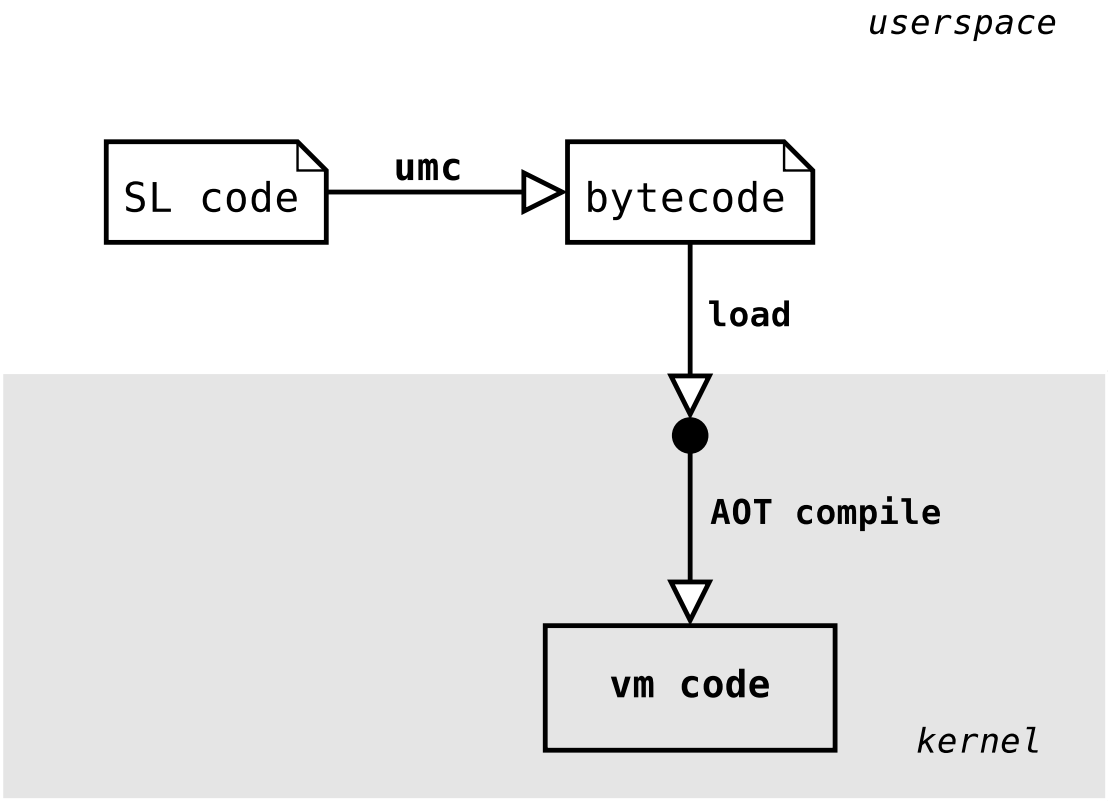


Figure 4.1: Complete compilation chain.

4.2 Bytecode

SL code is compiled into bytecode using compiler *umc* (userspace module compiler). The compiler produces architecture and kernel-configuration independent bytecode. The bytecode consists of several parts: code compile conditions definition, types definition, global variables definition, functions definitions and bytecode instructions.

All integer data in the bytecode are stored in big-endian byte order (network order). Whenever string is a part of a bytecode (except for instructions part) it consists of ASCII characters only and is represented as pair:

1^{st} part	2^{nd} part
`__u16` size	`char[]` string data (no null termination)

In instructions part the string is represented as standard C zero-terminated sequence of `char` values.

4.2.1 Compile conditions definition

Compile conditions part contains record for each compile condition. Compile conditions are used for types, functions and code definition in compliance with kernel configuration (for details see chapter 3.2.3).

The compile conditions definition starts with the following record:

Type	Description
`__u16`	Compile conditions count

After the compile conditions count, each condition has its record. Conditions can be nested, which means that one condition consists of several parts. Following condition types are defined:

Value	Name	Meaning in SL
Definition conditions		
0	`CC_DEF`	`@ifdef`
1	`CC_NDEF`	`@ifndef`
Number conditions		
2	`CC_GT`	`@ifgt`
3	`CC_GE`	`@ifge`
4	`CC_LT`	`@iflt`
5	`CC_LE`	`@ifle`
6	`CC_EQ`	`@ifeq`
7	`CC_NEQ`	`@ifneq`

Record of compile condition has following format:

Type	Description
`__u16`	Condition parts count
Following record for each condition part	
`__u8`	Compile condition type
string	Kernel configuration option name
Number conditions only	
`__s32`	Number used for comparison

Each condition is identified by its unique identifier (ID). There is no condition ID specified in the record though. The condition ID is determined by its order in compile conditions definition part. The first condition's ID equals to zero, second equals to one, etc.

4.2.2 Types definition

Types definition part contains record for each type (including basic types). In bytecode representation even function types exists though in SL code there's no way to use them for variable declarations. Function types are used only internally.

The types definitions section starts with the following record:

Type	Description
__u16	Types count

Then each type has its record. Different types has different record formats. Record formats are distinguished according to the flags fields which are common to all types. Possible flags are:

Value	Flag	Description
0x01	SIGNED_TYPE	Signed integer type
0x02	STRUCT_TYPE	Struct type
0x04	POINTER_TYPE	Pointer type
0x08	FUNCTION_TYPE	Function type
0x10	PACKED_TYPE	Packed type
0x20	UNION_TYPE	Union type
0x80	OPAQUE_TYPE	Opaque type – not modifiable in SL code

If no flags are set, the result is unsigned integer. There is only one allowed combination: `STRUCT_TYPE | PACKED_TYPE`. All other combinations are invalid.

Integer and opaque types

Integer types have no flags set or flags equal to `SIGNED_TYPE`. Structure of integer or opaque type record:

Type	Description
__u16	Unique ID
string	Name
__u16	Size (0 if unknown)
__u8	Flags

Pointer types

Pointer types have flags equal to `POINTER_TYPE`. Structure of pointer type record:

Type	Description
__u16	Unique ID
string	Name
__u16	Size (0 if unknown)
__u8	Flags
__u16	Unique ID of the type the pointer is pointing to

Array types

Array types have flags equal to `ARRAY_TYPE`. Structure of array type record:

Type	Description
__u16	Unique ID
string	Name
__u16	Total size (0 if unknown)
__u8	Flags
__u16	Unique ID of the type the array contains
__u16	Array elements count

Structure and union types

Structure and union types have same representation. Structure types have flags equal to `STRUCT_TYPE` or `STRUCT_TYPE | PACKED_TYPE` (packed structures), union types `UNION_TYPE`. Format of structure or union type record:

Type	Description
__u16	Unique ID
string	Name
__u16	Size (0 if unknown)
__u8	Flags
__u8	Struct/union members count
Following record for each struct/union member	
__u16	Member type ID
string	Member name
__u16	Compile condition ID

Function types

Function types have flags equal to `FUNCTION_TYPE`. Type record for function type is the largest one:

Type	Description
__u16	Unique ID
string	Name
__u16	Size (unused)
__u8	Flags
__u16	Functions return value type ID
__u8	Function arguments count
Following record for each function argument	
__u16	Argument type ID

4.2.3 Global variables definition

Global variables definition part contains record for each global variable. First record in global variables definition part is:

Type	Description
__u16	Global variables count

Variable record consists of three parts:

Type	Description
__u16	Unique ID
string	Name
__u8	External

During AOT compilation to vm code (see chapter 4.3), external variables are not created but are mapped to the existing variables provided by the kernel instead.

Same format as for global variables part is used for local variables as well.

4.2.4 Functions definition

Functions definition part contains information about all functions that can be used from SL code (SL functions, kernel functions and userspace functions). Functions definition part does not contain instructions of the functions only offsets to the instructions part. Functions definition part start with:

Type	Description
__u16	Functions count
__u16	Functions ID count

Functions count specifies how many function records follow. Functions ID count specifies how many different functions exist. This can be lower than functions

count because there can be alternative functions according to compile conditions (for details see chapter 4.3).

After functions count record, functions definition part contains one function record for each function. Function record has following structure:

Type	Description
__u16	Unique ID
__u16	Compile condition ID
string	Name
__u16	Function type ID
__u8	External
__u8	Userspace
Following record for userspace functions only	
__u16	Userspace function ID
Following record for SL functions only	
__u32	Code start – offset from the start of instructions
__u32	Code size
__u8	Exported to kernel
variables	Local variables

The last entry (local variables) in the function record has exactly same structure as the global variables definition part (see chapter 4.2.3).

4.2.5 Instructions

Instructions part contains code of all functions defined in SL code. No information where a function starts is included in the instructions part since all needed information is stored in functions definition part. Format of instructions part is:

Type	Description
__u32	Instructions size
(bytes)	Instructions

Bytecode instructions are not fixed size and instructions are not aligned. The first byte of each instruction is instruction code, next bytes are instruction specific. Since the bytecode is architecture independent some instruction argument often specifies type of affected data. Since the bytecode is designed to be (after AOT compilation to vm code) interpreted by stack machine, most of instructions are manipulating the stack.

Bytecode instruction set consists of 43 instructions which can be split into following categories:

- Arithmetic, bitwise and logical instructions (`add`, `sub`, `shl`, `shr`, `eq`, `or`, `not`, etc.).
  - Pops (one or two) operands from the stack, do some operation on them and store result on the stack.
  - Operands and result are same type.
  - Type of instruction's operands is specified as instruction argument.
- Control instructions.

`label`	Defines jump target.
`goto`	Unconditional jump to specified label.
`jmp_false`	Conditional jump. Pops value from the stack and jumps to a given label if the value is zero. Type of the value is instruction's argument.
`call`	Function call. Function arguments are already on stack (the last is on top). If function has variable argument count (external functions only) undefined arguments are described in instruction's arguments.
`return`	The value on top of the stack is moved to the address, where the return value is expected by the caller function, and the control is returned to the caller function.

- Data and stack manipulation instruction.

`pop`	Pop the value from the stack. Type is defined in instruction's argument.
`store`	Store the value from top of the stack to the address stored on the stack bellow the value. When the instruction is processed, the address must not be on stack anymore, but the value must remain on top (but moved).
`ld_s32`	Load signed 4B constant from the instruction's argument to the stack.
`ld_u32`	Load unsigned 4B constant from the instruction's argument to the stack.
`str`	Load string literal pointer to the stack. The string is included in the code as instruction's argument, can have any length and is terminated with zero character.
`addr_of`	Load variable address to the stack. The variable is identified by the instruction's argument.
`deref_ptr`	Pointer dereference. The pointer (type specified in instruction's pointer) on the stack is replaced by the value it points to.

`func_addr`	Load function address on the stack. The function is identified by the instruction's argument. This instruction is valid only for external or exported functions. (The pointer can be used from outside kernel part.)
`memb_offset`	Add offset of member in structure to the pointer on the stack. Structure type and member index are in instruction's arguments.
`container_of`	Subtracts offset of member in structure from the pointer on the stack. Structure type and member index are in instruction's arguments .
`typecast`	Cast value on the stack to different type. Source and result types are defined in instruction's arguments.
`typecast2`	Same as `typecast` instruction but works on two values on the stack simultaneously. (Used to cast operands of binary expressions to appropriate types).
`sizeof`	Load size of the type to the stack. Type is specified in instruction's argument.

- Code compile conditions.

`compile_condition_start`	If condition specified by its ID is not true, following instructions will not generate any code during AOT compilation (for details see chapter 4.3). Condition's ID is specified in the instruction's argument.
`compile_condition_end`	Effect of `compile_condition_start` instruction is canceled for following instructions.

All type checking is done during compilation to the bytecode. The bytecode contains type information in instruction context only to be able to produce correct vm code.

4.3 VM Code

When the bytecode is loaded into kernel, it is AOT compiled to vm code. Vm code is architecture dependent code directly interpreted by in-kernel virtual machine. Vm code is as simple as possible, no metainformation are included. Since only vm code needs to be stored in kernel when the AOT compilation is done, memory requirements are minimal.

AOT compilation of the bytecode consists of:

1. Types defined in types definition part are processed and architecture-specific properties are set (size, alignement, structure padding, etc.).

2. Global variables are allocated. The advantage of allocating global variables before the instructions are compiled to vm code is that global variables can be referenced by their absolute addresses in vm code.

3. Each function is processed. If function is external, its address is obtained. If it is userspace function, no action is done (difference is in function call). Otherwise the function is compiled:

   (a) Initialize local variables, but don't allocate them. Local variables are addressed by their offsets from LV register (start of local variables on stack – see chapter 4.3.2 for details).

   (b) If function is exported, set appropriate structure in kernel for it (see chapter 4.4.2 for details).

   (c) Bytecode instructions of the function body are compiled to vm code instructions.

The AOT compilation of bytecode instructions is a simple transformation. Every bytecode instruction is processed separately (no context is preserved). Some bytecode instructions can produce nothing at all. Typical example is `typecast` instruction if source and target type have same representation on the architecture.

Special part of AOT compilation is adaptation to kernel configuration. The adaptation steps are integrated into the AOT compilation steps defined above. The summary of adaptation steps is:

1. Evaluate all compile conditions when they are loaded.

2. Adapt structures and union types by pruning off structure (union) members defined with unfulfilled compile conditions.

3. Skip compilation of functions defined with unfulfilled compile conditions.

4. Skip bytecode instructions in `compile_condition_start-end` block defined with unfulfilled compile conditions.

Since, during the adaptation, parts of the code are pruned off, it is possible that adapted bytecode, though original SL is correct, contains references to non-existent objects. AOT compilation on such adapted bytecode fails and error description is written to kernel log.

4.3.1 Instruction set

Most of the instructions in vm code are analogous to respective bytecode instructions. Bytecode instructions in format *`instr (type_id)`* are directly

converted to *instr*_size () where size is size of the type. For example bytecode instruction add can be converted to vm instructions add1, add2, add4 or add8 according to size of its operands. When it is needed to handle signed and unsigned types in a different way, set of instructions for signed and unsigned types exists in vm code (e.g. mod1s and mod1u for modulo on 1B signed and unsigned operands respectively).

More complicated transformations are:

- Instruction str – load string constant. In contrary to the bytecode, string literals are not part of the vm code but are stored separately. Only address of the string is inserted in the vm code.
- Instructions call and return when calling function defined in SL code – calling conventions of virtual machine are described in chapter 4.3.2.
- Instruction call when calling kernel function – interface between kernel part and core kernel is described in chapter 4.4.2.
- Instruction call when calling userspace function – interface between kernel part and userspace is described in chapter 4.4.3.
- Instructions load, store and addr_of – local and global variables are distinguished. Addresses of global variables are directly set in the vm code while addresses of local variables are set as their offsets relative to the lv register (on the stack). Vm code instructions look like load_addr_4(address) for loading 4B from specified address or store_locvar_2(offset) to store 2B to the local variable which starts at given offset relative to the lv registry. Instructions load and store has also two-arguments format for nonstandard sizes such as store_locvar(offset, size).
- Instructions typecast and typecast2 – different cast for integer types and for other types. Casts of integer types are converted to instructions int_cast and int_cast2 which make sure that signum is correct, etc. Casts of other types are converted to nonint_cast which only resizes the value on the stack.
- Instruction func_addr – the address of proxy function is directly inserted in the vm_code. Proxy function is a function in the kernel (and therefore addressable from the kernel), which only responsibility is to call vm code function with same arguments it has been called with, and then pass the return value of the vm code to its caller. Proxy function is assigned to the function when the function definition is read in. Proxy functions are described in details in chapter 4.4.2.

For complete list of vm code instructions see appendix C.

4.3.2 Virtual Machine

Virtual machine is a vm code interpreter. *Virtual machine instance* is virtual machine currently executing some part of vm code. The vm code can be shared between multiple instances of virtual machine, and multiple virtual machine instances can interpret same vm code without blocking (of course the interpreted code can block). Virtual machine instance is limited to the code it has been created with, it is not possible to call functions from other vm code. Virtual machine instance maintains its own fixed-size[3] stack and a its own registry:

```
struct registry {
    __u8 * sp;              /* <- stack pointer       */
    __u8 * lv;              /* <- local variables     */
    int position;           /* <- instruction pointer */
    struct registry * caller_func_registry;
};
```

When the the vm instruction `call` (call function defined in SL code) is interpreted by the virtual machine, all function arguments are already on the stack – last argument on top. The calling sequence consists of following steps:

1. Push local variables to the stack directly after function arguments.
2. Set `registry.position` to the next instruction that should be processed when the called function returns.
3. Push registry to the stack.
4. Set `registry.caller_func_registry` to point to the just pushed registry.
5. Set `registry.lv` to point to the first function argument (arguments are threated as local variables inside the called function).
6. Set `registry.position` to the first instruction of called function.

The function return sequence is realized by two instructions in a row:

1. Instruction `setretval*` copies the return value from top of the stack to the address `registry.lv` points to. (AOT compilation makes sure that the local variable block is always large enough to hold return value.)
2. Instruction `return` replaces current registry by registry of the caller function (next instruction in the caller modifies the stack pointer so that return value is on top).
3. The caller function then deletes local variables from the stack preserving just the return value.

[3]Stack size is 4KB in the implemented solution.

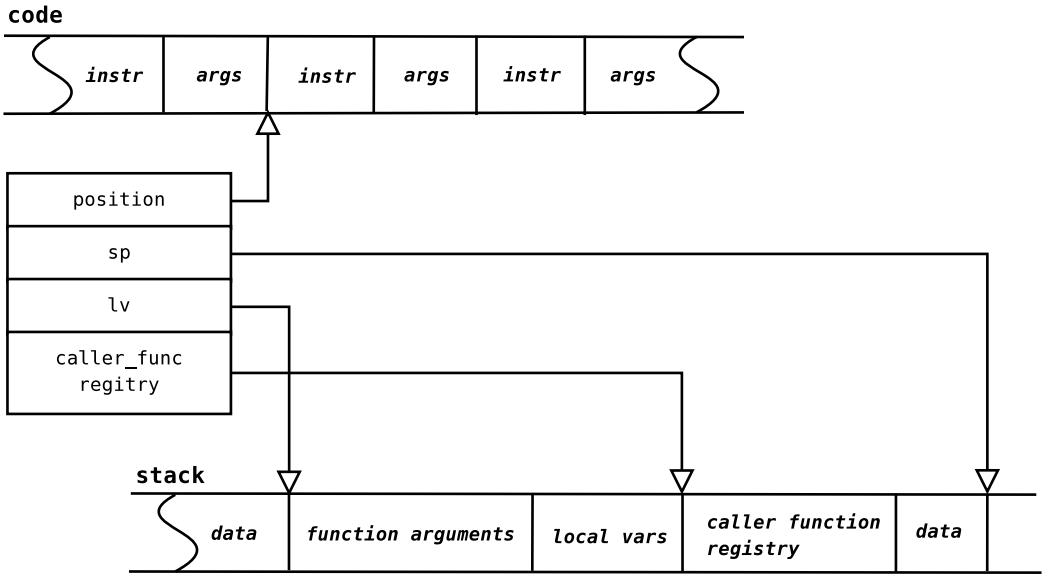


Figure 4.2: Virtual machine.

If `return` instruction is reached and `registry.caller_function_registry` is `NULL`, the virtual machine instance execution ends and return value is passed to the caller of virtual machine.

Calling kernel functions needs architecture dependent code which takes arguments from the stack and calls the kernel function with them. Since every architecture has different binary interface, the vm instructions needs to be architecture dependent as well. The vm instruction which calls kernel function is `external_call` and its arguments are architecture specific.

Calling of userspace functions is described in chapter 4.4.3. The vm instruction which calls userspace function is `userspace_call( __u16 userspace_function_id)`.

4.4 Kernel implementation

Kernel implementation of userspace modules consists of following main components:

- control component,
- AOT compiler,
- virtual machine,
- interface between core kernel and userspace module,

- interface between kernel part and userspace part.

AOT compiler and virtual machine has been already described. Other components will be described in following chapters.

4.4.1 Control component

The control component is responsible for loading and maintaining kernel parts of userspace modules.

The communication channel, that is used by the userspace module kernel infrastructure to receive commands (such as load module, etc.), is character device `/dev/usrmodctl`. The communication model used on the device is simple challenge-response. Userspace writes command and then reads response. The communication is completely stateless, no context is preserved between commands. Only one userspace process can have device opened at the same time (which is ensured in the kernel).

Commands that are sent via the device are mostly text based, but every command must be preceded with binary data `size_t` indicating ammount of bytes the command contains. Usage of tool `usrmodctl` described in chapter 3.4 is therefore encouraged. Possible commands are (without the size argument):

- Command `add` *`name`*`\n`*`bytecode`* – load kernel part of userspace module to the kernel. Response to this command is value of type `int`, which defines minor number of the device that has been associated with the kernel part[4] if the value is non-negative or identifying error if negative.
- Command `rem` *`name`*`\n` – unload kernel part of given name. Response to this command is value of type `int` – zero means success, nonzero error.
- Command `lst\n` – list currently loaded modules. Response is text listing of kernel part names.

When error occurs more details can be found in kernel log.

Kernel part of userspace module is represented by `struct servant` in the kernel. Since `struct servant` represents kernel part's in-kernel implementation, word *servant* will be used in this chapter instead of kernel part.

```
struct servant {
    char                name[16];
    struct compiled_code *vm_code;

    /* init function offset in code */
    int                 init_start;
    size_t              init_locvar_size;
    /* exit function offset in code */
```

[4]For details about interface between kernel part and userspace part see chapter 4.4.3.

```
    int                 exit_start;
    size_t              exit_locvar_size;

    /*
     * Functions that should be called on userspace
     * connection and disconnection
     */
    int                 userspace_connected_start;
    size_t              userspace_connected_locvar_size;
    int                 userspace_disconnected_start;
    size_t              userspace_disconnected_locvar_size;

    /* names of kernel symbols used */
    char                **external_symbols;
    int                 external_symbols_count;

    /* userspace device minor number */
    int                 device_minor;

    /* userspace device opened */
    struct semaphore    userspace_channel_exclusivity_lock;

    /* is userspace being called now */
    struct semaphore    userspace_usage_lock;

    /* unlocked when request can be read by userspace */
    struct semaphore    userspace_request_ready;
    /* unlocked when result has been written by userspace */
    struct semaphore    userspace_result_ready;
    struct userspace_request *userspace_request;
    int                 userspace_result;

    spinlock_t          userspace_state_lock;
    enum userspace_state userspace_state;

    struct list_head    list;
};
```

Servant is created when the bytecode is loaded to the kernel and is identified by its unique name. During servant creation its userspace channel is initialized (for details see chapter 4.4.3). The rest of the servant – vm code, special function locations and kernel interface (for detail see chapter 4.4.2) – is set up during AOT compilation. When the servant is successfully initialized, its `init()` function is called.

If `init()` call is successful, servant is already functional though userspace

part is not connected yet. As already described in chapter 3.2.4, servant can be notified on userspace connection and disconnection by implementing functions `void userspace_connected()` and `void userspace_disconnected()`.

When the servant is unloaded, its `exit()` function is executed. The function must assure that it is safe to remove the servant (especially that all used connections to and from the core kernel are correctly disabled – unregistering all callback functions, etc.). When the function returns, servant is unloaded from memory and all its communication channels are destroyed.

4.4.2 Interface between core kernel and userspace module

The main requirement of userspace module is ability to do (almost) everything that normal kernel module can do. In order to comply with this requirement, userspace module infrastructure must implement *bidirectional* interface between kernel part and rest of the kernel. The interface must provide following functionality:

1. Access kernel variables[5].
2. Call kernel functions[6].
3. Make it possible for the kernel to directly call functions from kernel part[7].

To be able to access symbols (both variables and functions) from the kernel, address of every symbol declared as `external` in SL code is obtained during AOT compilation. Address of symbols is obtained with function `get_external_address()`:

```
intptr_t get_external_symbol_address(char *name)
{
    intptr_t addr;
    int i;
    /* try built-in and overwrites first */
    for(i = 0; i < ARRAY_SIZE(symbols); i++) {
        if(strcmp(name, symbols[i].name) == 0) {
            /* call debug-hook */
            extern_registered(name, (void *) symbols[i].address, 1);

            return symbols[i].address;
        }
    }
    /* exported by kernel */
    addr = (intptr_t) __symbol_get(name);
```

[5]Exported with `EXPORT_SYMBOL` macro.
[6]Exported with `EXPORT_SYMBOL` macro.
[7]Functions exported with `EXPORT_FUNC` in SL code.

```
    if(!addr) {
     printk("Bytecode compile error. Symbol %s does not "
               "exist in kernel.\n", name);
     return (intptr_t) NULL;
    }

    /* call debug-hook */
    extern_registered(name, (void *) addr,  0);

    return addr;
}
```

The function first looks for the symbol in its own symbol list. If it is found there, the address associated with the symbol name is returned. If symbol name is not found in the list, kernel function `__symbol_get()` is used to find the symbol address. Since the symbol list is searched before the symbol is looked for in the kernel, it is possible to substitute some kernel functions with other implementations (e.g. make own implementation of `kmalloc()` which does some logging, etc.).

Since call of `__symbol_get()` also increments reference count of the symbol's owner, it is necessary to decrease the reference count when the symbol is no longer in use. This is accomplished by call of function `void put_external_symbol(char *name)` or `void put_external_symbol_addr(intptr_t symbol_address)`[8] when the userspace module is unloaded.

Being able to determine symbol address is enough for variable access. During AOT compilation, every external variable's address is retrieved and is directly used in vm code. Access to kernel variables is therefore exactly same as access to global variables for virtual machine (absolute addressing is used).

Unfortunately, there is one problem with obtaining symbol address. Many functions (e.g `atomic_add()`, semaphore function `down()`, etc.) are defined as `static inline` in header files. These functions are not exported and they often don't exist as functions at all. If these functions contain architecture specific code, they cannot be compiled in SL code and there is no way to make them directly accessible either. The only way is to define built-in function with same name for each of them, which will only call the original function. To simplify this, built-in function wrapper can automatically be generated during kernel module compilation. To generate it, add required function declarations (whole declaration must be on single line) to the file `arch_specific_functions.skel`.

Calling kernel functions (and built-in functions provided by userspace module infrastructure) consists of two steps: retrieval of function address and the call itself. Retrieval of function address is identical to retrieval of variable address. The call of the function is architecture dependant since every architecture has completely different calling conventions. Therefore the format

[8]Function `put_external_symbol_addr()` is currently not working due to a bug in current kernel (2.6.26). Patch fixing the problem has already been sent to the LKML by Jiří Kosina [3]

of the vm instruction: `external_call (`*`architecture-dependant arguments`*`)` is architecture dependent as well.

The format of `external_call` instruction arguments on `x86_64` architecture[9] as described in [4]:

Type	Description
`__u8`	`1` if return value is MEMORY CLASS or `0` if INTEGER CLASS (see [4] chapter 3.2 for details about argument memory classes).
`__u8`	Number of arguments passed on stack.
In correct order (last arg. first) for each argument passed on stack	
`int`	Offset relative to stack pointer of virtual machine of argument that will be passed on stack.
`size_t`	Size of the argument.
End of arguments passed on stack	
`__u8`	Number of arguments passed in registry (maximum 6 – or 5 if return value is MEMORY CLASS).
In correct order for each argument passed in registry	
`int`	Offset relative to stack pointer of virtual machine of argument that will be passed in register
`__u8`	Size of the argument (1-8).
End of arguments passed in registry	
`size_t`	Size of return value.
`intptr_t`	Function address.
`size_t`	Total size of all arguments.

Format of the instruction is designed to make the execution of vm code as simple and fast as possible. The virtual machine does not need to keep any information about function. All required data are encoded in the instruction's arguments.

The remaining part of the interface – calling SL code functions from the kernel – is the most complicated part of the interface. The main problem is that every function that should be callable from the kernel needs to have its unique address. But since vm code is interpreted by virtual machine, no function in vm code has its own address that can be used to call the function. In contrary, all functions are called through the virtual machine's main function (which is shared among all userspace modules). Subsequent problem is passing of arguments and return value. Since the function calling is architecture specific, implementation has to be architecture-specific as well.

[9]Proof of concept implementation of userspace modules is on `x86_64` architecture

Since every function that should be accessible from the kernel (*exported function*) needs to have unique entry point which is callable by kernel[10], separate real function (*proxy function*) must be created for each exported function. It is not possible to create functoins at runtime, all proxy functions must be compiled directly to the kernel. Proxy functions are generated during the kernel compilation and their number is controlled by kernel configuration option `CONFIG_USRMOD_CALLBACK_FUNCTIONS`. This implies the main limitation of userspace modules: *Maximal sum of functions exported from all loaded userspace modules is specified at kernel compile time and it is not possible to change it without kernel recompilation.*

Though implementation of proxy functions is architecture specific, the base concept is architecture independent. Two arrays are created during the kernel compilation. One contains addresses of proxy functions and the other contains information about exported functions that should be called from respective proxy. When the proxy function is called it gets information about the exported function it should call. If proxy is called and has no associated exported function, error is logged[11] since it means that some userspace module has not done its cleanup correctly.

Every exported function is associated with a proxy function during AOT compilation. If there is not enough unused proxy functions, AOT compilation fails. All proxy functions, associated with functions exported from the module, are made available during module unload.

Proxy function steps (if no error occurs):

1. Read function arguments (architecture dependent) according to associated exported function information.
2. Prepare arguments to the format expected by virtual machine.
3. Run the exported function in virtual machine
4. Pass the results of exported function back to the caller.

Proxy function must be implemented to be able to run in atomic context such as in interrupt handler.

Proxy function implementation on `x86_64` architecture[12] according to [4]:

```
/*
 * Collect all possible arguments and call generic proxy code
 */
void func_proxy_1(__u64 rdi, __u64 rsi, __u64 rdx, __u64 rcx,
                  __u64 r8, __u64 r9)
{
```

[10]Function callable by kernel is in `.text` section and contains reasonable binary code.

[11]`BUG()` macro is actually used.

[12]Proof of concept implementation of userspace modules is on `x86_64` architecture.

```
    struct fcall_abi *f;
    __u8 *stack_position;
    __u64 regs[REGISTRY_COUNT] = { rdi, rsi, rdx, rcx, r8, r9 };

    /* Get the exported function specific information. */
    f = &callback_functions[id];

    /* Read the stack pointer value and adapt it to point to the
     * first argument passed in the stack.
     * Arguments passed in the registry are already filled in
     * function arguments.
     */
    __asm__ __volatile__ (
        "movq %%rbp, %0\n\t"
        : "=m" (stack_position)
    );
    stack_position += 16;

    /* If no exported function is associate, BUG(). */
    if(!f->servant)
     goto error_not_defined;

    /* All possible arguments collected, call the real proxy. */
    func_proxy(f, regs, stack_position);

    return;

error_not_defined:
    printk("Callback function is not defined!\n");
    BUG();
}

void func_proxy(struct fcall_abi *f, __u64 *regs,
                __u8 *stack_position)
{
    /*
     * According to ABI first REGISTRY_COUNT INTEGER class
     * arguments are passed in registry rdi, rsi, rdx, rcx,
     * r8 and r9, other are passed on stack with 8B aligning
     * in reverse order, so stack_position points to the first
     * argument passed on stack.
     */

    struct fragment *fragments; /* used to pass arguments to
```

```
                                virtual machine */
int reg_idx;                   /* current register */
int arg_idx;                   /* current argument */


fragments = NULL;
fragments = kmalloc(f->args_count * sizeof(struct fragment),
                    GFP_ATOMIC);
if(!fragments)
 goto error_alloc;

/*
 * If return value is MEMORY CLASS, return address is passed
 * in first register - doesn't contain argument - skip it.
 */
reg_idx = f->retval_memory_class ? 1 : 0;

/*
 * Read function arguments in correct order from registry
 * and stack.
 */
for(arg_idx = 0; arg_idx < f->args_count; arg_idx++) {
    fragments[arg_idx].size = f->args[arg_idx].size;
    if(f->args[arg_idx].passed_in_reg) {
        /* read from registry */
        fragments[arg_idx].where = &regs[reg_idx++];
    } else {
        /* read from stack */
        fragments[arg_idx].where = (void *) stack_position;
        /* arguments on stack are 8B alligned */
        stack_position += (f->args[arg_idx].size + 7)
                          & ~(size_t)7;
    }
}

/*
 * Call the virtual machine - the call is different for
 * return value in MEMORY CLASS and in INTEGER CLASS.
 */
if(f->retval_memory_class) {
    /*
     * return value in memory class - address where to
     * store result is in first register
     */
```

```
        stack_machine(f->servant, f->position, f->args_count,
                      fragments, f->locvar_size,
                      (void *) regs[0], f->retval_size);
        kfree(fragments);
        /* return value address must be in rax */
        __asm__ __volatile__(
            "movq %0, %%rax\n\t"
            :
            : "r" (regs[0])
            : "rax"
        );
    } else {
        /*
         * Return value is INTEGER CLASS - value is returned
         * in first two registers.
         */
        stack_machine(f->servant, f->position, f->args_count,
                      fragments, f->locvar_size, (void *) regs,
                      f->retval_size);
        kfree(fragments);
        /* return value must be in rax, rdx */
        __asm__ __volatile__ (
            "movq %0, %%rax\n\t"
            "movq %1, %%rdx\n\t"
            :
            : "r" (regs[0]), "r" (regs[1])
            : "rax", "rdx"
        );
    }

    return;

error_alloc:
    printk(KERN_CRIT "Not enough memory.\n");
    BUG();
}
```

4.4.3 Interface between kernel part and userspace part

The interface provides simple RFC-like functionality (kernel part calls functions provided by userspace part). Each servant has its own communication channel dedicated for calling functions provided by userspace. Character device file is used as the channel in the implementation, though userspace interface is designed to

be independent of channel implementation. It is easily possible to change channel implementation without the need to change existing userspace modules.

Since userspace function call implies context switch, it is not possible to call userspace function while in atomic context. This is checked every time the userspace function is called and if process is running in atomic context, kernel `BUG()` is triggered.

As already described in chapter 3.2.4, it is not possible to directly access data from kernel in userspace part and vice versa. Therefore no pointers can be passed between kernel part and userspace part during the call. Instead special type `userspace_data_t` is used to transfer data. Type `userspace_data_t` is internally represented as `struct userspace_data`:

```
typedef unsigned int userspace_direction_t;
#define DIRECTION_KERNEL2USER ((userspace_direction_t) (1 << 0))
#define DIRECTION_USER2KERNEL ((userspace_direction_t) (1 << 1))

struct userspace_data {
    userspace_direction_t __direction;
    size_t  size;
    void  *data;
};
```

Field `data` is pointer to memory it refers to, `size` is the size of the referred data in bytes and `__direction` identifies in which direction are data transfered. Possible directions are `DIRECTION_KERNEL2USER`, `DIRECTION_USER2KERNEL` and their combination (`DIRECTION_KERNEL2USER | DIRECTION_USER2KERNEL`). Functions `data_for_user()`, `data_from_user()` and `data_exchange_user()` are used to create appropriate `userspace_data_t` objects from given pointer and size. Userspace is not working on kernel data itself but on its private copy of data. Userspace function is called with pointer to its copy of data in place of `usrespace_data_t` argument. The userspace copy is written to the correct place in kernel when the function returns. The copy from or to the kernel are realized according to `__direction` flag[13].

The call of userspace has following structure:

- `size_t` total size of the data that will be send during the call.
- `__u16` unique identifier of function that should be called.
- function arguments – directly copied block from stack.
- data that are identified (by pointer `data` and `size`) by each `userspace_data_t` arguments that has `DIRECTION_KERNEL2USER` set.

[13]If direction is `DIRECTION_USER2KERNEL`, the kernel data are not copied to the userspace and are not accessible in userspace.

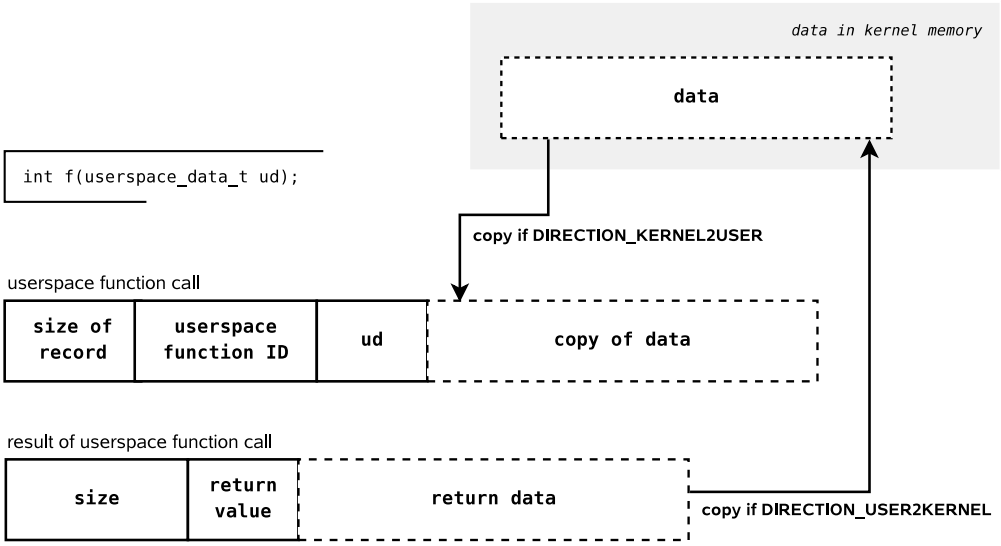


Figure 4.3: Userspace function call.

When the data representing userspace call are sent to userspace, kernel put itself to sleep until result has been written by userspace or userspace device has been closed unexpectedly (which means `USERSPACE_ERROR`).

Userspace does blocking read from the device and thus is active only when the function is called. Userspace reads all data from the kernel and for each `userspace_data_t` arguments sets its `data` pointer to correct location (either to the specific position in data read from the kernel, or to newly allocated data if direction is `DIRECTION_USER2KERNEL`). When arguments are processed, function is called with provided arguments and valid pointers. When the function returns, its return value (`int`) and all data blocks that have `DIRECTION_USER2KERNEL` set[14] are sent to the kernel[15].

When result is written to the channel, data with `DIRECTION_USER2KERNEL` are copied to their correct locations in kernel memory and return value of the function is returned to the caller. Size of data transfered between userspace and kernel are kept in the kernel and if size of data received from userspace is incorrect, `USERSPACE_ERROR` is returned. This check prevents data oveflows.

In current implementation all data are transfered through character device, it is however possible to for example use shared memory for data part or change the whole implementation easily. Since exactly same amount of memory copying is

[14]Some data has both `DIRECTION_KERNEL2USER` and `DIRECTION_USER2KERNEL` set. So it is not enough to just copy data newly allocated during argument processing before the call.

[15]It is not necessary to send size of the data because it is already known by the kernel.

required[16] for all possible implementations[17], speed impact should be minimal.

The used implementation has one drawback. Since there is no pairing of call requests and call responses it is possible to process only one userspace request at the same time. This can be easily fixed by extending the calling protocol. The advantage of this is extremely simple implementation of userspace part.

[16]Most the time, the manipulated data are data that are created by the rest of the kernel and userspace module cannot allocate them in shared memory. Data therefore needs to be copied between their real location and shared memory.

[17]If libc caching is not used for device file access.

Chapter 5

Evaluation of implemented solution

This chapter describes advantages, disadvantages, limitations and possible extensions of implemented solutions. The solution is also compared with existing projects covering similar functionality.

5.1 Advantages

Main advantages of userspace modules corresponds to main goals of the thesis assignment. Userspace modules make it possible to implement kernel module functionality in userspace. Main advantage of this approach is that userspace module can use all available existing libraries which reduces implementation time. Usage of existing libraries also helps to avoid bugs in userspace modules since library is tested in many other applications[1].

Userspace modules are more resistant to bugs than normal kernel modules since most of the functionality is moved to the userspace. If a bug occurs in userspace part, the bug does not make kernel unstable or panicked since kernel part receives `USERSPACE_ERROR` and can do correct error recovery[2]. It is also possible for user to load a fixed or new userspace part without the need to unload the kernel part (it is for example possible to fix buggy filesystem driver without the need to unmount mounted filesystems), if the kernel part is able to handle userspace part temporal disconnection.

Since the kernel part is architecture and kernel configuration independent, it is not necessary to recompile the userspace module when the kernel is updated[3]. Since no kernel-specific configuration is needed, it is possible to create distribution-independent userspace modules that do not need to be compiled on the system they

[1] Only stable and well tested libraries should be used.

[2] Although when bug occurs in kernel part, it has same results as if bug occurred in normal kernel module.

[3] If the binary interface of used objects has not changed. More about the problem in disadvantages part (see chapter 5.2).

are installed on. This feature can be used even for distribution-specific packages on distributions that use customized or user-configured kernel (it is possible to use same userspace module on all systems without the need to recompiling).

Userspace modules are easier to debug than usual kernel modules. Userspace part, and therefore most of the functionality, can be directly debugged with any userspace debug tools. Kernel part infrastructure contains several unimplemented functions that are called on significant event (access to memory, function call, etc.). All these functions are defined in file `debug-hook.c` and can be implemented in any way.

Another feature that can be used for debugging purpose is kernel function overriding. As has already been described in chapter 4.4.2, it is possible to replace kernel function by own implementation (e.g. with some additional checks or with modified functionality). The replace is implemented by adding override function to special symbol array defined in `external_symbols.c`.

Kernel-part debugging features looks like they could be used for access restrictions and security auditing. Unfortunately, that is not possible. Since the SL is designed to provide same functionality as C language, it is not possible to control it with such simple mechanism or the overhead of the control would greatly exceed time and resources needed by the userspace module itself[4].

5.2 Disadvantages, limitations and their proposed solutions

Described solution has several disadvantages and limitations. Some of them are caused by simplicity of proof-of-concept implementation, others are implied by the requirements and therefore cannot be fixed. If it is possible to dispose the disadvantage completely or lower its impact, description of proposed fix follows the problem description.

Proposed solution consists of two parts: userspace part running as separate process and interpreted kernel part running in the context of process using the functionality provided by the userspace module or in kernel-only thread. Since the required functionality (if provided by userspace part) is provided by separate process, thread requesting the functionality has to wait for the other process. If the requester has significantly larger priority, possible slowdown is significant. The negative aspect can be lowered by temporal increase of userspace part process priority.

Related problem is that userspace parts and userspace function call protocol are designed as single-threaded. If one call is currently in progress, other callers are

[4]Example: it is possible to allocate array of `char` and with standard array access set the values in a way that the array, if type-casted to `struct timer` has its function pointer set to any address. Kernel part then register the timer. When the timer is triggered, the function at special address is called. Since userspace part can easily read the kernel symbols file, kernel part can obtain address of overwritten function and then call it in a way just described.

blocked until the actual call is completed. This can only be solved by implementing multi-threaded protocol for userspace function call and generating code for multi-threaded userspace parts. As has already been described in chapter 4.4.3, implemented solution is designed to simplify multi-threaded protocol integration.

Architecture independance, though one of main goals, can be sometimes regarded as disadvantage since architecture-specific features are not accessible from the kernel part of userspace module and userspace part has insufficient permissions. This limitation of userspace module cannot be eliminated without loosing architecture independence.

Although bytecode is both architecture and kernel-configuration independent, it is not type safe. If for example two kernel versions have different definition of type (structure) used in userpsace module, correct definition must be present when SL code is compiled. Since in current implementation kernel header files cannot be used directly in SL code, manual intervention is required. Two possible fixes to this problem exist. The first fix is to extend SL to comply with full C syntax (e.g. create custom gcc backend instead of dedicated compiler)[5]. This fixes the problem of incompatibility of kernel header but bytecode compilation is still needed. The second fix is *lazy types compilation*.

Lazy types compilation allows incomplete type definition in SL code. Only needed members of structures are defined in SL code. The full definition of the type is read during bytecode load to the kernel (to be available at AOT compilation time) and structure's members are mapped to their correct positions in real types. This requires to generate types information from kernel header files during kernel compilation and load them together with bytecode[6]. Real types of structure members can be fixed in a same way (e.g. from `int` to `long`, etc. – some limitations need to be applied, however, e.g. integer type to struct type must not be allowed). This solution is sufficient to make bytecode fully independent (architecture independent, kernel configuration independent and kernel version independent).

Many important architecture dependent functions in kernel are defined in header files as `static inline`, which makes them unaddressable (for details see chapter 4.4.2) and therefore directly unaccessible to userspace modules. To be able to access these functions, special wrapper must be defined for each of them. Even though it is possible to create a small application that will process kernel header files and generate wrappers automatically, the application would need to determine which functions to process, which is not an easy task. Alternatively, human interaction would be required.

Another disadvantage is obvious – interpretation is much slower than native code execution. The implemented solution focuses to functionality and not to optimization, thus simple stack virtual machine with minimal set of simple

[5]Alternatively it is possible to implement some preprocessor for kernel header to produce correct SL syntax.

[6]It is not needed to keep them in kernel since the information is only needed at AOT compilation time.

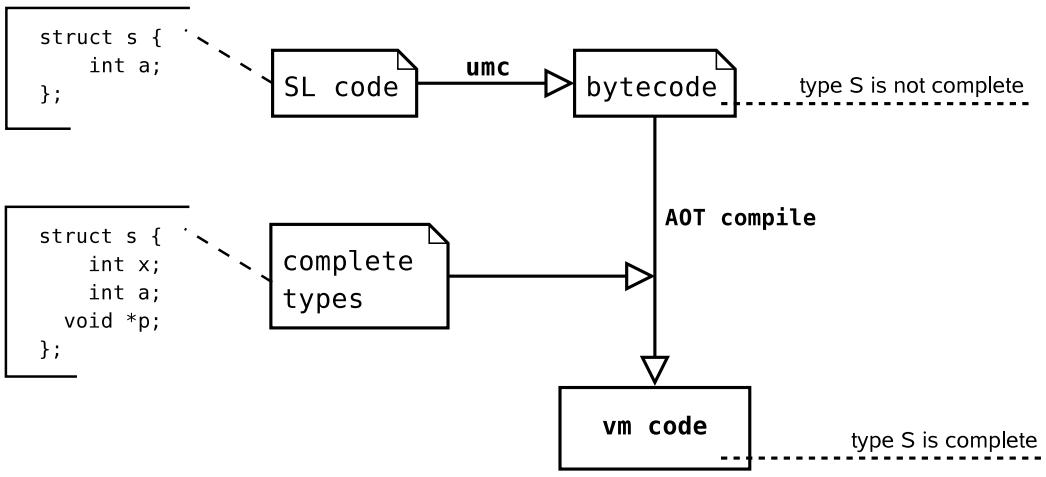


Figure 5.1: Lazy types compilation.

instructions is used. The compiler which compiles SL code to bytecode is very simple either, no optimizations are done during the compilation. If more interpretation-efficient bytecode and virtual machine are designed and optimizing compiler is used (e.g. gcc with appropriate backend), the speed gain will be significant (e.g. comparable to python or other interpreted languages). Interpreted code will still be three-times slower in a best case and much worse in average case than native code[7]. The speed loss can be more reduced by implementing just-in-time compilation. The other possibility is to extend the userpsace modules infrastructure to support kernel parts that are true kernel modules. Usage of true kernel modules will break the independence (architecture independence, kernel configuration independence and – if implemented – kernel version independence) of userspace module though.

Minor limitation of the implementation is that debug interface (see chapter 5.1 for details) must be set statically. It should be possible to load the implementation of debug-hooks dynamically. This could be implemented for example by creating interface to add functions that are called when the hook defined events are triggered. The implementation of debug-hooks can then be loaded in separate kernel module that will register the functions on its load and unregister them on unload.

[7]According to benchmarks comparing gcc to other existing interpreted languages [5].

5.3 Comparison with existing projects

There are currently no projects that cover same area as userspace modules. However, there are two projects that implement different models:

- FUSE – Filesystem in Userspace [1].
- UIO – Userspace drivers [6].

FUSE is a well-known project aiming to implement filesystem drivers in userspace. Since it is possible to create complete filesystem driver without the need to run driver specific code in atomic context, FUSE is a typical example of purpose-specific interface model (for details see chapter 2.1.2). The interface designed in FUSE has already proved to be well designed and sufficient for filesystem driver implementation, therefore userspace modules implementation would impose meaningless overhead.

UIO's aim is much closer to userspace modules implementation than FUSE. Main goal of UIO is to simplify writing char device drivers in userspace, while userspace modules are more generic. UIO complies to purpose-specific interface model with kernel module (for details see chapter 2.1.3). UIO needs the kernel module to register the device and do interrupt handling. The communication between userspace and kernel is realized via `/dev/uio*` devices, shared memory and sysfs files. However, the communication is limited. Device file is used to notify userspace that event occurred[8], while raw data are passed in memory mapped areas. No communication protocol is defined and therefore it is hard for kernel module to for example detect error on userspace part without implementing its own protocol on memory mapped areas.

UIO is simple by design and is targeted to char device drivers only. Its simplicity makes it harder to use (from developer's point of view) than userspace modules, which provides robust communication protocol with userspace. Its design is sufficient and effective for writing typical char device driver in userspace though, since the provided functionality is just enough for char device and all the logic is implemented in userspace.

Both projects are effective and well designed for their purpose and it would be unwise to use userspace modules in their place without serious reasons. It was not a goal of userspace modules to replace them though. Userspace modules were designed to be generic and its both their strength and weakness. It is unlikely to for them to be more effective in area that other project is specialized to.

[8]Which has same meaning as 'interrupt occured' in the kernel.

Chapter 6

Conclusion

The thesis has been aimed to design architecture for userspace modules development in Linux kernel and create of proof-of-concept implementation. The main goal of the thesis is to make userspace libraries and tools available to kernel code and move parts of the code from kernel to userspace.

Several possible models of userspace modules architecture were evaluated. Model which uses bytecode interpreted in the kernel has been chosen, because it mostly complies to userspace module requirements[1] and no other solution with same architecture exists. Benefit of the chosen model is that the bytecode is architecture and kernel configuration independent, which makes userspace modules easy to deploy.

As part of the thesis, source language which is compiled into bytecode has been defined. Since the source language has been designed to provide same functionality as C language, bytecode is able to perform same actions as regular kernel module. The only limitation is that bytecode cannot perform architecture dependent actions such as registry access, because bytecode is architecture independent.

During the implementation, interface between kernel and bytecode, and interface between bytecode and userspace have been implemented. Bytecode is able to call functions from both kernel and userspace, and kernel is able to call functions defined in the bytecode.

Since the bytecode runs interpreted in the kernel, its execution can be controlled and restrictions can be imposed on it. Such restrictions can be defined in the implemented solution, but due to the fact that bytecode must be able to perform same actions as normal kernel module, exploit written in bytecode can bypass the restrictions. Restrictions can be used to debug bytecode and make the bytecode less error prone.

The usage of userspace modules architecture is recommended for modules that require access to userspace libraries and are not time-critical. However, if there exists native solution to access the required functionality from userspace without the need of kernel code, it should be usually preferred.

Though the implementation is fully functional, it could be improved to

[1] As described in chapter 1.2.

become more convenient to use. Important improvements are optimization of interpretation speed and ability to use kernel headers without the need to adapt them (or automate their adaptation). Since the implementation is limited to x86_64 architecture, another possible improvement is porting it to other architectures.

Main advantage of userspace modules is that the interface between bytecode and userspace is customized according to required functionality and that userspace modules do not need to be recompiled on kernel updates. Userspace modules are easy to deploy comparing to standard kernel modules.

The implemented solution is fully functional and is sufficient to implement architecture independent userspace module.

Bibliography

[1] Filesystem in Userspace. http://fuse.sourceforge.net.

[2] G. Kroah-Hartman J. Corbet, A. Rubini. *Linux Device Drivers*, pages 18–22. O'Reilly, Third edition, 2005.

[3] Jiří Kosina. LKML [PATCH] module: make symbol_put_addr() work for all exported symbols. http://www.ussg.iu.edu/hypermail/linux/kernel/0806.2/1375.html, 2008.

[4] System V Application Binary Interface, AMD64 Architecture processor supplement, Draft Version 0.99. http://www.x86-64.org/documentation/abi.pdf, 2007.

[5] The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/gp4/benchmark.php?test=all\&lang=gcc.

[6] UIO: user-space drivers. *LWN.net*, 2007. http://lwn.net/Articles/232575/.

[7] Wikipedia: AOT Compiler. http://en.wikipedia.org/wiki/AOT_compiler.

[8] Wikipedia: Stack machine. http://en.wikipedia.org/wiki/Stack_machine.

Appendix A

SL grammar

This appendix provides a detailed description of the SL grammar. The following conventions are used in the grammar:

- Terminal symbols are specified in bold.
- Production rules begin with a nonterminal symbol and the sequence of characters ":", and end with a semi-colon (";").
- Alternation for production rules is specified using the vertical-bar symbol ("|").

Following symbols are defined by regular expression or text to increase readability:

- number : `[0-9]+` | `0[xX][0-9a-fA-F]+` ;
- character : `'.'` ;
- id : `[A-Za-z_][A-Za-z_0-9]*` ;
- string_literal : standard C string literal with escape sequences such as `"hello world\n"` ;

sl_code : declarations
;

declarations
: /* empty */
| declarations declaration
;

declaration
: type_declaration
| var_declaration

| extern_var_declaration
| conditional_function_declaration
| userspace_function_declaration
| function_proxy_declaration
;

conditional_function_declaration
: function_declaration
| extern_function_declaration
| compile_condition conditional_function_declaration
;

type_declaration
: **typedef** type id **;**
| **struct** id **;**
| struct_declaration **;**
| struct_declaration __**attribute**__(__**packed**__) **;**
| **union** id **{** struct_member_declarations **} ;**
| **union** id **;**
;

struct_declaration
: **struct** id **{** struct_member_declarations **}**
;

struct_member_declarations
: struct_member_declaration
| struct_member_declarations struct_member_declaration
;

struct_member_declaration
: type id **;**
| type id [number] **;**
;

var_declarations
: /* empty */
| var_declarations var_declaration
;

var_declaration
: type id **;**
| type id [number] **;**

;

extern_var_declaration
: **extern** type id **;**
;

type
: id
| **unsigned** id
| **struct** id
| **union** id
| type *****
| **typeof** **(** exp **)**
;

function_declaration
: type id **(** argument_declarations_or_nothing **) ;**
| type id **(** argument_declarations_or_nothing **) {** var_declarations statements **}**
;

extern_function_declaration
: **extern** type id **(** argument_declarations_or_nothing **) ;**
;

userspace_function_declaration
: __**user** type id **(** argument_declarations_or_nothing **) ;**
;

function_proxy_declaration
: **EXPORT_FUNC** **(** id **) ;**
;

argument_declarations_or_nothing
: /* empty */
| vararg_declaration
| argument_declarations
| argument_declarations **,** vararg_declaration
;

argument_declarations
: argument_declaration
| argument_declarations **,** argument_declaration
;

vararg_declaration
: **...**
;

argument_declaration
: type id
;

arguments_or_nothing
: /* empty */
| arguments
;

arguments
: argument
| arguments **,** argument
;

argument
: single_expression
;

statements
: statement
| statements statement
;

statement
: block
| statement_expression **;**
| **while** **(** comparable_expression **)** statement
| **for** **(** exp **;** comparable_expression **;** exp **)** statement
| **do** block **while** **(** comparable_expression **)** **;**
| **return** **;**
| **return** exp **;**
| decision_statement
| compile_condition statement
;

decision_statement
: **if** **(** comparable_expression **)** statement

| **if** (comparable_expression) statement **else** statement
;

block
: **{** statements **}**
| **{ }**
;

comparable_expression
: exp
;

statement_expression
: assignment_expression
| function_call
;

single_expression
: assignment_expression
| cast_expression
| operator_expression
;

exp
: single_expression
| single_expression **,** exp
;

assignment_expression
: address_expression = single_expression
| * cast_expression = single_expression
;

cast_expression
: basic_expression
| (type) cast_expression
;

operator_expression
: single_expression < single_expression
| single_expression <= single_expression
| single_expression == single_expression
| single_expression **!**= single_expression

| single_expression >= single_expression
| single_expression > single_expression
| single_expression + single_expression
| single_expression - single_expression
| single_expression * single_expression
| single_expression / single_expression
| single_expression % single_expression
| single_expression << single_expression
| single_expression >> single_expression
| single_expression || single_expression
| single_expression && single_expression
| single_expression & single_expression
| single_expression | single_expression
;

constant_expression
: number
| character
| string
;

string
: string_literal
| string string_literal
;

sizeof_expression
: **sizeof** (type)
| **sizeof** (exp)
;

address_expression
: id
| address_expression . id
| atomic_expression -> id
| atomic_expression [exp]
;

function_call
: id (
arguments_or_nothing)
;

atomic_expression

: constant_expression
| sizeof_expression
| container_of_expression
| function_call
| address_expression
| **(** exp **)**
;

container_of_expression
: **container_of** **(** single_expression **,** type **,** id **)**
;

basic_expression
: atomic_expression
| **-** cast_expression
| **!** cast_expression
| **~** cast_expression
| **&** address_expression
| ***** cast_expression
;

compile_condition
: **@ifdef** id
| **@ifndef** id
| **@ifgt** id number
| **@ifge** id number
| **@iflt** id number
| **@ifle** id number
| **@ifeq** id number
| **@ifneq** id number
| **@ifgt** id **-** number
| **@ifge** id **-** number
| **@iflt** id **-** number
| **@ifle** id **-** number
| **@ifeq** id **-** number
| **@ifneq** id **-** number
;

Appendix B

Bytecode instruction set

This appendix contains complete set of bytecode instructions. All multi-byte arguments are in big-endian number format. Bytecode instructions can be logically split into following categories:

- Arithmetic, bitwise and logical instructions.
- Control instructions.
- Data and stack manipulation instruction.
- Code compile conditions.

To simplify instructions description, values on the stack are referred to as `val0` (value on top of the stack), `val1` (value bellow `val0` on the stack), etc.

B.1 Arithmetic, bitwise and logical instructions

Most instructions in this category have same format and same semantic. Instructions have one argument which defines type of their operand. Instructions pop two values of given type from the stack, do instruction-specific operation on them and push the result of same type to the stack.

If an instruction has different format or semantic, it is explicitly stated in its description.

add(__u16 type_id)

Instruction implements addition on values on the stack (`val1 + val0`).

sub(__u16 type_id)

Instruction implements subtraction on values on the stack (`val1 - val0`).

mult(__u16 type_id)

Instruction implements multiplication on values on the stack (`val1 * val0`).

div(__u16 type_id)

Instruction implements division on values on the stack (`val1 / val0`).

mod(__u16 type_id)

Instruction implements modulo on values on the stack (`val1 % val0`).

shl(__u16 type_id)

Instruction implements binary left shift on values on the stack (`val1 << val0`).

shr(__u16 type_id)

Instruction implements binary right shift on values on the stack (`val1 >> val0`).

lt(__u16 type_id)

Instruction implements lower-than comparison on values on the stack (`val1 < val0`).

le(__u16 type_id)

Instruction implements lower-than-or-equal comparison on values on the stack (`val1 <= val0`).

gt(__u16 type_id)

Instruction implements greater-than comparison on values on the stack (`val1 > val0`).

ge(__u16 type_id)

Instruction implements greater-than-or-equal comparison on values on the stack (`val1 >= val0`).

eq(__u16 type_id)

Instruction implements equals-to comparison on values on the stack (`val1 == val0`).

neq(__u16 type_id)

Instruction implements not-equals-to comparison on values on the stack (`val1 != val0`).

logand(__u16 type_id)

Instruction implements logical-and on values on the stack (`val1 && val0`).

logor(__u16 type_id)

Instruction implements logical-or on values on the stack (`val1 || val0`).

lognot(__u16 type_id)

Instruction implements logical-not on value on the stack (`!val0`).

bitand(__u16 type_id)

Instruction implements bitwise-and on values on the stack (`val1 & val0`).

bitor(__u16 type_id)

Instruction implements bitwise-or on values on the stack (`val1 | val0`).

bitneg(__u16 type_id)

Instruction implements bitwise-negation on value on the stack (`~val0`).

add_ptr_pre(__u16 type_id)

Instruction implements pointer arithmetic addition. Instruction argument is type of the pointer, value which should be added has type `long` (`(long)val1 + (type *)val0`).

add_ptr_post(__u16 type_id)

Instruction implements pointer arithmetic addition. Instruction argument is type of the pointer, value which should be added has type `long` (`(type *)val1 + (long)val0`).

sub_ptr_post(__u16 type_id)

Instruction implements pointer arithmetic subtraction. Instruction argument is type of the pointer, value which should be added has type `long` (`(type *)val1 - (long)val0`).

B.2 Control instructions

Control instructions are used to control the flow of execution.

label (__u16 label_id)

Instruction defines a jump target (*label*) for `goto` and `jmp_false` instructions. The arguments contain unique identifier of the label.

goto (__u16 label_id)

Instruction defines unconditional jump to label specified in instruction's argument.

jmp_false (__u16 type_id, __u16 label_id)

Instruction defines conditional jump. Value of type specified in first argument is popped from the stack and if the value is zero, jump to the label specified in second argument is performed.

call (__u16 function_id [, __u8 variadic_arg_count, __u16 variadic_arg1_type_id, __u16 variadic_arg2_type_id, ...])

Instruction defines function call. The function is identified by the first argument. If the function has variable number of arguments[1], their amount and types are specified in remaining instruction arguments. Function arguments are read from the stack.

return (__u16 return_type_id)

Instruction moves value of type defined in the argument from the stack to the position for return value and the control is returned to the caller function.

B.3 Data and stack manipulation instructions

pop (__u16 type_id)

Remove the value of the type defined in the argument from the stack.

store (__u16 type_id)

Instruction stores value `val0` of type defined in the argument to the address `val1`. After the instruction is processed, the address `val1` is removed from the stack, but the `val0` is preserved on the stack (although moved to position of `val1`).

[1] Only external functions can have variable arguments count.

ld_s32 (__s32 const)

Instruction loads signed 4B constant from the argument to the stack.

ld_u32 (__u32 const)

Instruction loads unsigned 4B constant from the argument to the stack.

str (char[] string)

Instruction loads zero-terminated string from the argument to the memory and loads its address to the stack.

addr_of (__u16 variable_id)

Instruction loads address of variable to the stack. The variable identifier is read from the argument.

deref_ptr (__u16 pointer_type_id)

Instruction dereferences the pointer stored on stack. The pointer (type of the pointer is defined in the argument) on the stack is replaced by the value it points to.

memb_offset (__u16 struct_type_id, __u8 member_index)

Instruction replaces (on stack) pointer to structure by pointer to its member. Structure type is defined in the first argument, index of structure member is defined in the second argument.

container_of (__u16 struct_type_id, __u8 member_index)

Instruction replaces (on stack) pointer to structure member by pointer to the structure, that contains the member. Structure type is defined in the first argument, index of structure member is defined in the second argument.

func_addr(__u16 function_id)

Instruction loads address of the function to the stack. Function is identified by the argument. The instruction is valid for external and exported functions only, because the address can be used from outside kernel part.

typecast (__u16 type_id_from, __u16 type_id_to)

Instruction casts value on the stack from the type defined in first argument to type defined in the second argument.

typecast2 (__u16 type_id_from_1, __u16 type_id_to_1, __u16 type_id_from_2, __u16 type_id_to_2)

Instruction works same as instruction `typecast` but works simultaneously on two values on the stack (`val1` and `val0` respectively). Instruction is used to cast operands of binary expressions to appropriate types.

sizeof (__u16 type_id)

Instruction loads the size of the type to the stack. Type is defined in the argument and the size is stored as `size_t` type.

B.4 Code compile conditions

Code compile conditions are used to ignore block of instructions according to the kernel configuration. Compile condition instructions are always paired and compile condition blocks (start–end) cannot be nested.

compile_condition_start (__u16 compile_condition_id)

If condition specified by the argument is not true, following instructions will not generate any code during AOT compilation (for details see chapter 4.3).

compile_condition_end ()

Instruction cancels effect of `compile_condition_start`.

Appendix C

VM code instruction set

This appendix contains a complete set of of vm code instructions. In contrary to bytecode instructions set, all multi-byte arguments are in architecture-specific byte order.

Instructions can be logically split into following categories:

- Arithmetic, bitwise and logical instructions.
- Control instructions.
- Data and stack manipulation instruction.

To simplify instructions description, values on the stack are referred to as `val0` (value on top of the stack), `val1` (value bellow `val0` on the stack), etc.

C.1 Arithmetic, bitwise and logical instructions

Most instructions in this category have same semantic. Instructions pop two values of instruction-specific type from the stack, do instruction-specific operation on them and push the result of same type to the stack.

If an instruction has different format or semantic, it is explicitly stated in its description.

add instructions

Instructions implement addition on values on the stack (`val1 + val0`). Instruction versions:

`add_1()`	works on 1B operands
`add_2()`	works on 2B operands
`add_4()`	works on 4B operands
`add_8()`	works on 8B operands

sub instructions

Instructions implement subtraction on values on the stack (`val1 - val0`). Instruction versions:

- `sub_1()` works on 1B operands
- `sub_2()` works on 2B operands
- `sub_4()` works on 4B operands
- `sub_8()` works on 8B operands

mul instructions

Instructions implement multiplication on values on the stack (`val1 * val0`). Instruction versions:

- `mul_1()` works on 1B operands
- `mul_2()` works on 2B operands
- `mul_4()` works on 4B operands
- `mul_8()` works on 8B operands

div instructions

Instructions implement division on values on the stack (`val1 / val0`). Instruction versions:

- `div_s1()` works on 1B signed operands
- `div_s2()` works on 2B signed operands
- `div_s4()` works on 4B signed operands
- `div_s8()` works on 8B signed operands
- `div_u1()` works on 1B unsigned operands
- `div_u2()` works on 2B unsigned operands
- `div_u4()` works on 4B unsigned operands
- `div_u8()` works on 8B unsigned operands

mod instructions

Instructions implement modulo on values on the stack (`val1 % val0`). Instruction versions:

`mod_s1()` works on 1B signed operands
`mod_s2()` works on 2B signed operands
`mod_s4()` works on 4B signed operands
`mod_s8()` works on 8B signed operands
`mod_u1()` works on 1B unsigned operands
`mod_u2()` works on 2B unsigned operands
`mod_u4()` works on 4B unsigned operands
`mod_u8()` works on 8B unsigned operands

shl instructions

Instructions implement binary left shift on values on the stack (`val1` << `val0`). Instruction versions:

`shl_s1()` works on 1B signed operands
`shl_s2()` works on 2B signed operands
`shl_s4()` works on 4B signed operands
`shl_s8()` works on 8B signed operands
`shl_u1()` works on 1B unsigned operands
`shl_u2()` works on 2B unsigned operands
`shl_u4()` works on 4B unsigned operands
`shl_u8()` works on 8B unsigned operands

shr instructions

Instructions implement binary right shift on values on the stack (`val1` >> `val0`). Instruction versions:

`shr_s1()` works on 1B signed operands
`shr_s2()` works on 2B signed operands
`shr_s4()` works on 4B signed operands
`shr_s8()` works on 8B signed operands
`shr_u1()` works on 1B unsigned operands
`shr_u2()` works on 2B unsigned operands
`shr_u4()` works on 4B unsigned operands
`shr_u8()` works on 8B unsigned operands

lt instructions

Instructions implement lower-than comparison on values on the stack (`val1 < val0`). Instruction versions:

- `lt_s1()` works on 1B signed operands
- `lt_s2()` works on 2B signed operands
- `lt_s4()` works on 4B signed operands
- `lt_s8()` works on 8B signed operands
- `lt_u1()` works on 1B unsigned operands
- `lt_u2()` works on 2B unsigned operands
- `lt_u4()` works on 4B unsigned operands
- `lt_u8()` works on 8B unsigned operands

le instructions

Instructions implement lower-than-or-equal comparison on values on the stack (`val1 <= val0`). Instruction versions:

- `le_s1()` works on 1B signed operands
- `le_s2()` works on 2B signed operands
- `le_s4()` works on 4B signed operands
- `le_s8()` works on 8B signed operands
- `le_u1()` works on 1B unsigned operands
- `le_u2()` works on 2B unsigned operands
- `le_u4()` works on 4B unsigned operands
- `le_u8()` works on 8B unsigned operands

gt instructions

Instructions implement greater-than comparison on values on the stack (`val1 > val0`). Instruction versions:

- `gt_s1()` works on 1B signed operands
- `gt_s2()` works on 2B signed operands
- `gt_s4()` works on 4B signed operands
- `gt_s8()` works on 8B signed operands
- `gt_u1()` works on 1B unsigned operands
- `gt_u2()` works on 2B unsigned operands
- `gt_u4()` works on 4B unsigned operands
- `gt_u8()` works on 8B unsigned operands

ge instructions

Instructions implement greater-than-or-equal comparison on values on the stack (`val1 >= val0`). Instruction versions:

- `ge_s1()` works on 1B signed operands
- `ge_s2()` works on 2B signed operands
- `ge_s4()` works on 4B signed operands
- `ge_s8()` works on 8B signed operands
- `ge_u1()` works on 1B unsigned operands
- `ge_u2()` works on 2B unsigned operands
- `ge_u4()` works on 4B unsigned operands
- `ge_u8()` works on 8B unsigned operands

eq instructions

Instructions implement equals-to comparison on values on the stack (`val1 == val0`). Instruction versions:

- `eq_1()` works on 1B operands
- `eq_2()` works on 2B operands
- `eq_4()` works on 4B operands
- `eq_8()` works on 8B operands

neq instructions

Instructions implement not-equals-to comparison on values on the stack (`val1 != val0`). Instruction versions:

- `ne_1()` works on 1B operands
- `ne_2()` works on 2B operands
- `ne_4()` works on 4B operands
- `ne_8()` works on 8B operands

log_and instructions

Instructions implement logical-and on values on the stack (`val1 && val0`). Instruction versions:

`log_and_1()` works on 1B operands
`log_and_2()` works on 2B operands
`log_and_4()` works on 4B operands
`log_and_8()` works on 8B operands

log_or instructions

Instructions implement logical-or on values on the stack (`val1 || val0`). Instruction versions:

`log_or_1()` works on 1B operands
`log_or_2()` works on 2B operands
`log_or_4()` works on 4B operands
`log_or_8()` works on 8B operands

log_not instructions

Instructions implement logical-not on value on the stack (`!val0`). Instruction versions:

`log_not_1()` works on 1B operand
`log_not_2()` works on 2B operand
`log_not_4()` works on 4B operand
`log_not_8()` works on 8B operand

bit_and instructions

Instructions implement bitwise-and on values on the stack (`val1 & val0`). Instruction versions:

`bit_and_1()` works on 1B operands
`bit_and_2()` works on 2B operands
`bit_and_4()` works on 4B operands
`bit_and_8()` works on 8B operands

bit_or instructions

Instructions implement bitwise-or on values on the stack (`val1 | val0`). Instruction versions:

`bit_or_1()` works on 1B operands
`bit_or_2()` works on 2B operands
`bit_or_4()` works on 4B operands
`bit_or_8()` works on 8B operands

bit_neg instructions

Instructions implement bitwise-negation on value on the stack (`val0`). Instruction versions:

`bit_neg_1()` works on 1B operand
`bit_neg_2()` works on 2B operand
`bit_neg_4()` works on 4B operand
`bit_neg_8()` works on 8B operand

add_long_ptr(long size)

Instruction implements pointer arithmetic addition (`(long)val1 + (type *)val0`). Instruction argument is size of the type the pointer points to.

add_ptr_long(long size)

Instruction implements pointer arithmetic addition (`(type *)val1 + (long)val0`). Instruction argument is size of the type the pointer points to.

sub_ptr_long(long size)

Instruction implements pointer arithmetic subtraction (`(type *)val1 - (long)val0`). Instruction argument is size of the type the pointer points to.

C.2 Control instructions

Control instructions are used to control the flow of execution.

jmp(int position)

Instruction defines unconditional jump to the position specified in instruction's argument.

jmp_false instructions

Instructions define conditional jump. Value of instruction-specific size is popped from the stack and if the value is zero, jump to the code position specified in instruction argument is performed. Instruction versions:

`jmp_false1(int position)`	size of tested value is 1B
`jmp_false2(int position)`	size of tested value is 2B
`jmp_false4(int position)`	size of tested value is 4B
`jmp_false8(int position)`	size of tested value is 8B

call(int position, size_t local_vars_size)

Instruction implements SL function call. Instruction must push actual registry and set new registry correctly. Local variables of new function are already allocated when the instruction is executed.

setretval instructions

Instructions load value of instruction-specific size from the stack to the address where the return value will be read from (start of function arguments) by the caller function. Instruction versions:

`setretval1()`	size of return value is 1B
`setretval2()`	size of return value is 2B
`setretval4()`	size of return value is 4B
`setretval8()`	size of return value is 8B
`setretval(size_t size)`	size of return value is specified in the argument

return()

Instruction replaces actual registry by registry of the caller function. If no caller function is defined, virtual machine ends its execution and copies return value of the SL function to the specific location.

userspace_call(__u16 f, size_t args_size, __u8 data_arguments_count, int data_arguments_offsets[])

Instruction calls userspace function specified in the first argument. Function arguments are on stack when the instruction is executed. Instruction must process data arguments and send the call sequence to the userspace part. When the userspace finish function execution, instruction must write data to correct

addresses and replace arguments on stack by function return value. More details about userspace communication are described in chapter 4.4.3.

external_call(...)

Instruction calls function defined in the kernel and replaces function arguments on stack by its return value. Instruction `external_call`, including its syntax, is architecture-specific. Implementation on `x86_64` architecture is described in chapter 4.4.2.

C.3 Data and stack manipulation instructions

sp_add(__s16 value)

Instruction adds value defined in its argument to the sp register. Used for example for local variables allocation, etc.

load_const instructions

Instructions loads value of instruction-specific size from the instruction argument. Instruction versions:

`load_const1(__u8 value)`	size of the value is 1B
`load_const2(__u16 value)`	size of the value is 2B
`load_const4(__u32 value)`	size of the value is 4B
`load_const8(__u64 value)`	size of the value is 8B

load_addr instructions

Instructions loads value of instruction-specific size from the address stored on stack. Instruction versions:

`load_addr1()`	size of the value is 1B
`load_addr2()`	size of the value is 2B
`load_addr4()`	size of the value is 4B
`load_addr8()`	size of the value is 8B
`load_addr(size_t size)`	size of the value is specified in the argument

store instructions

Instructions store value `val0` of instruction-specific size to the address stored in `val1`. After the instruction is processed, the address `val1` is removed from the

stack, but the `val0` is preserved on the stack (although moved to position of `val1`). Instruction versions:

`store1()`	size of the value is 1B
`store2()`	size of the value is 2B
`store4()`	size of the value is 4B
`store8()`	size of the value is 8B
`store(size_t size)`	size of the value is specified in the argument

locvar_addr(intptr_t offset)

Instruction loads address of local variable, with offset in local variable block specified in the argument, to the stack.

int_cast(__u8 flags)

Instruction casts value on the stack according to the flags argument. This instruction can be used only for integer values and pointers.

Bits in flags argument have following meaning:

Bit	Description
0	source type is signed
1	result type is signed
2-3	source type size – encoded:
	00 size 1B
	01 size 2B
	10 size 4B
	11 size 8B
4-5	result type size – encoded:
	00 size 1B
	01 size 2B
	10 size 4B
	11 size 8B
6-7	reserved

int_cast2(__u8 flags1, __u8 flags2)

Instruction works same as instruction `int_cast` but works simultaneously on two values on the stack (`val1` and `val0` respectively). Instruction is used to cast operands of binary expressions to appropriate types.

nonint_cast(size_t source_size, size_t result_size)

Instruction casts (resizes) value of size specified in the first argument to size specified in the second argument. If result_size > source_size, the padding data is undefined (is not set to zero).

Appendix D

Example

This is a simple example demonstrating the features of userspace modules. When the bytecode is loaded to the kernel and the userspace part connects to it, the work (`work_struct`) is scheduled. The work calls userspace function which writes a message to kernel memory (through the `userspace_data_t` argument). Userspace also sets how many seconds should the kernel wait. Work sets up a timer according to the seconds received from userspace function.

When the timer handler is executed (in interrupt context), the message that userspace wrote during the work execution is printed to the kernel log. Then new work is scheduled again.

D.1 Kernel part code

Content of file `test.sl`

```
#include "functions.h"
#include <stdlib.h>
#include <stdio.h>

int process_arguments(int argc, char *argv[])
{
/* argument processing and initialization */

unsigned int seed = 0;
FILE *f = fopen("/dev/urandom", "r");

fread(&seed, sizeof(seed), 1, f);
fclose(f);
srandom(seed);

return 0;
}
```

```
int next_timer(void *seconds, void * msg)
{
static int num = 0;

*(unsigned long *) seconds = 1 + random() % 10;
sprintf((char *) msg, "This is call number %d. "
                "Time set to %lds.\n", num++,
                *(unsigned long *) seconds);

return 0;
}

\begin{verbatim}
/*
 * These are not linux kernel header files but their SL versions.
 */
#include <usrmod.h>
#include <linux/jiffies.h>
#include <linux/workqueue.h>
#include <linux/kernel.h>

/* Userspace function called from the work_func. */
__user int next_timer(userspace_data_t seconds,
                      userspace_data_t msg);

struct my_work {
    struct work_struct w;
    long time_set;
};

/* timer and work definitions */
struct my_work my_w;
struct timer_list t;

/* Buffer where message is stored during
 * next_timer() call .
 */
char message[256];

/* Lock, which makes access to the terminate variable,
 * variable terminate is set to 1 when no more timer or
 * work should be scheduled. */
spinlock_t tlock;
```

```
int terminate = 0;

/* Timer handler. Called in interrupt handler.
 * Since it is not possible to call userspace from
 * interrupt handler, timer setup work which will then
 * call userspace and initiate new timer, and so on. */
void timer_func(unsigned long data)
{
    unsigned long j;


    j = jiffies;
    printk("Timer called. Timer has been set before %lu"
           " seconds.\n",
           (j - data) / get_HZ());
    printk("Message from user: %s\n", message);

    my_w.time_set = j;

    spin_lock(&tlock);
    if(!terminate)
        schedule_work(&my_w.w);
    spin_unlock(&tlock);
}
/* Make function callable from the kernel. */
EXPORT_FUNC(timer_func);

/* Function calls userspace function which and set new timer
 * according to the time set by userspace function. The
 * userspace function also sets the message that will be
 * printed in timer handler.
 */
void work_func(struct work_struct *w)
{
    struct my_work *mw;
    unsigned long j;
    unsigned long sec;
    int res;

    mw = container_of(w, struct my_work, w);
    printk("Work called. Work has been set before %u ms.\n",
           jiffies_to_msecs(jiffies)
           - jiffies_to_msecs(mw->time_set));
```

```
    res = next_timer(data_from_user(sizeof(sec), &sec),
                     data_from_user(255, message));

    if(error_userspace(res)) {
        printk("Error while calling userspace.\n");
        return;
    }

    printk("Setting new timer that will expire in %lu seconds.\n",
           t.expires);

    t.data = jiffies;
    t.expires = sec * get_HZ() + jiffies;
    spin_lock(&tlock);
    if(!terminate)
        add_timer(&t);
    spin_unlock(&tlock);
}
EXPORT_FUNC(work_func);

/*
 * Initiate the first work.
 */
void userspace_connected()
{
    printk("Userspace connected. Starting...\n");
    my_w.time_set = jiffies;

    spin_lock(&tlock);
    if(!terminate)
        schedule_work(&my_w.w);
    spin_unlock(&tlock);

}

/*
 * Initialization
 */
int init()
{

    printk("Loading userspace module.\n");

    /* set the last char to '\0' - userspace will never be
```

```
     * able to overwrite it so printk will always be safe. */
    message[255-1] = 0;

    spin_lock_init(&tlock);

    /* init work and timer, but don't set them */
    INIT_WORK(&my_w.w, work_func);
    init_timer(&t);
    t.function = timer_func;

    return 0;
}

/*
 * Unload. Cancel timer and work, exit.
 */
void exit()
{
    spin_lock(&tlock);
    terminate = 1;
    spin_unlock(&tlock);

    printk("Stopping work...\n");
    cancel_work_sync(&my_w.w);
    printk("Stoping the timer...\n");
    del_timer_sync(&t);

    printk("Unloading userspace module.\n");
}
```

D.2 Userspace code

Content of `test/functions.c`:

```
#include "functions.h"

#include <stdlib.h>
#include <stdio.h>

/*
 * Argument processing and initialization
 * Initialize random number generator .
 */
int process_arguments(int argc, char *argv[])
```

```
{
    unsigned int seed = 0;
    FILE *f = fopen("/dev/urandom", "r");

    fread(&seed, sizeof(seed), 1, f);
    fclose(f);
    srandom(seed);

    return 0;
}

/*
 * When called from kernel, it sets how many seconds timer
 * should wait and writes a message that is later displayed
 * during the timer handler.
 */
int next_timer(void *seconds, void * msg)
{
    static int num = 0;

    *(unsigned long *) seconds = 1 + random() % 10;
    sprintf((char *) msg, "This is call number %d. "
            "Time set to %lds.\n",
            num++, *(unsigned long *) seconds);

    return 0;
}
```

D.3 Compilation

The example is included in file `example.tar.bz2`. Makefile is included in the archive. Following steps are required to run the example:

```
tar xjf example.tar.bz2
cd example

# compile the sl code and generate the userspace part
make

# compile the generated userspace part
make -C example

# run the userpspace module
example/example -b example.usrmod
```

```
[ Hit Ctrl+C to terminate userspace module ]
```

When the userspace module is running, it generates its output to the kernel log.

Appendix E

Installation

Userspace modules implementation consists of two parts: userspace SL code compiler and kernel module, which implements AOT compiler and vm code interpreter.

Both parts are distributed as source code archives that can be compiled on any recent Linux system. Kernel module can be compiled on `x86_64` architecture only. Supported kernel versions are 2.6.25 and newer.

E.1 SL compiler installation

Source files of the SL compiler `umc` are in file `umc.tar.bz2`. For successful build of the compiler, following tools are required:

- GNU C compiler `gcc`[1].
- Autotools `autoconf`[2], `automake`[3].
- Lexical analyzer `flex`[4] and parser generator `bison`[5].
-

All required tools are included in most Linux distributions.

Compilation and installation of the compiler is done with a typical sequence:

```
tar xjf umc.tar.bz2
cd umc
./configure
make
sudo make install
```

[1] http://gcc.gnu.org/
[2] http://www.gnu.org/software/autoconf
[3] http://www.gnu.org/software/automake
[4] www.gnu.org/software/flex
[5] www.gnu.org/software/bison

The compiler is installed as `/usr/bin/umc`. The compiler can be uninstalled with command `make uninstall` from the installation directory.

E.2 Kernel module installation

Kernel module `usrmod` source files are in file `usrmod.tar.bz2`. Kernel module can be built on architecture `x86_64` only. Linux kernel 2.6.25 or newer and complete kernel build environment installation (included in most Linux distribution) are required.

Proxy function's count[6] `CONFIG_USRMOD_CALLBACK_FUNCTIONS` can be changed in `Makefile`. Default value is `256`.

Compilation and installation of the module is done with following commands:

```
tar xjf usrmod.tar.bz2
cd usrmod
make
make install
```

After the installation, module can be loaded with command `modprobe usrmod`. The module can be uninstalled with command `make uninstall` from the installation directory.

[6]See chapter 4.4.2 for details.