Export Symbol

`EXPORT_SYMBOL()` helps you provide APIs to other modules/code.

Below is an example of how you can export APIs to an external module and use APIs given by the other module.

👇 File: myModule1.c

```
/* ... */

int GLOBAL_VARIABLE = 1000;
EXPORT_SYMBOL(GLOBAL_VARIABLE);

// Function to print hello for num times.
void print_hello(int num)
{
    while (num--) {
        printk(KERN_INFO "Hello Friend!!!\n");
    }
}
EXPORT_SYMBOL(print_hello);

// Function to add two passed number.
void add_two_numbers(int a, int b)
{
    printk(KERN_INFO "Sum of the numbers %d", a + b);
}
EXPORT_SYMBOL(add_two_numbers);

static int __init my_init(void)
{
    printk(KERN_INFO "Hello from Export Symbol 1 module.");
    return 0;
}

static void __exit my_exit(void)
{
    printk(KERN_INFO "Bye from Export Symbol 1 module.");
}

module_init(my_init);
module_exit(my_exit);

/* ... */
```

👇 File: myModule2.c

```
/* ... */

extern void print_hello(int);
extern void add_two_numbers(int, int);
extern int GLOBAL_VARIABLE;

/*
 * Call functions which are in other module.
 */
static int __init my_init(void)
{
    printk(KERN_INFO "Hello from Hello Module");
    print_hello(2);
    add_two_numbers(5, 6);
    printk(KERN_INFO "Value of GLOBAL_VARIABLE %d", GLOBAL_VARIABLE);
    return 0;
}

static void __exit my_exit(void)
{
    printk(KERN_INFO "Bye from Hello Module");
}

module_init(my_init);
module_exit(my_exit);

/* ... */
```

It is noteworthy that, after compiling these two scripts into kernel objects, inserting `myModule2.ko` before `myModule1.ko` will give errors:

```
$ sudo insmod myModule2.ko
insmod: ERROR: could not insert module myModule2.ko: Unknown symbol in module
```

Insert `myModule1.ko` then `myModule2.ko` and you can see the following:

```
$ sudo insmod myModule1.ko
$ sudo insmod myModule2.ko
[15606.692155] Hello from Export Symbol 1 module.
[15612.175760] Hello from Hello Module
[15612.175764] Hello Friend!!!
[15612.175766] Hello Friend!!!
[15612.175780] Sum of the numbers 11
[15612.175782] Value of GLOBAL_VARIABLE 1000
```

Likewise, removing `myModule1` before `myModule2` also raises errors:

```
$ sudo rmmod myModule1
rmmod: ERROR: Module myModule1 is in use by: myModule2
```

You can check the following links for more information:

5. EXPORT_SYMBOL — Linux Kernel Workbook 1.0 documentation (lkw.readthedocs.io)

Linux World: Exporting symbols from module (tuxthink.blogspot.com)

c - How to prevent "error: 'symbol' undeclared here" despite EXPORT_SYMBOL in a Linux kernel module? - Stack Overflow

c - How to call exported kernel module functions from another module? - Stack Overflow

How to define a function in one linux kernel module and use it in another? - Stack Overflow

Declare external functions

If you have exported kernel_clone()and getname_kernel, and you want to use them in your file, you have to declare them as external functions like this, after MODULE_LICENSE("GPL"):

```
MODULE_LICENSE("GPL");

extern pid_t kernel_clone(struct kernel_clone_args *args);
extern struct filename* getname_kernel(const char __user *);
```

How to create a working thread

Refer to page 40 of the tut slides In our case, the working function is my_fork(), so my_fork() needs to do all the job.

How to create a working process

Refer to page 27 to 30 of the tut slides For `struct kernel_clone_args` important fields are:

```
flags,
stack,
stack_size,
parent_tid,
child_tid,
tls
```

for `stack` field, we pass the working function that the working process needs to execute
For example, if we want to fork a process and let the process execute `hello()`:

```
    struct kernel_clone_args clone_args = {
        .flags = SIGCHLD,
        .pidfd = NULL,
        .child_tid = NULL,
        .parent_tid = NULL,
        .exit_signal = SIGCHLD,
        .stack = (unsigned long) &hello,
        .stack_size = 0,
        .tls = 0
    };

    pid_t pid = kernel_clone(&clone_args);
```

How to insert, remove module and check output

0. Before insert the kernel object, you have to sign in the root account.

```
$ sudo su
```

1. Compile your kernel module in program2 directory(if it is not yet compiled):

```
$ make
```

2. Insert the kernel module: Use the insmod command to insert the kernel module:

```
$ insmod MODULE_NAME.ko
```

3. List the module you insert and Check if the module is loaded:

```
$ lsmod
$ lsmod | grep MODULE_NAME
```

4. View kernel logs:

```
$ dmesg | tail
```

5. Remove your module

```
$ rmmod MODULE_NAME.ko
```