Assignment V

Assignment 5 Objective

- Know how to export kernel symbols.
- Learn how to use kernel threads.
- Understand how kernel module dependency works.

Assignment 5 Description

For this assignment you will write two modules. In first module you will create an exported kernel variable and in second module you will create a kernel thread. This kernel thread will increment exported variable every time it wakes up. You will define "**my_flagCount**" exported kernel variable using **EXPORT_SYMBOL()** macro in the first module and in the second module you will use "extern int **my_flagCount**;" to access exported kernel variable. Your second module should increment and display exported variable every time kernel thread wakes up. In this assignment you will use "**modprobe**" command to show how module dependency automatically will load the first module.

Kernel Symbols Overview

When the module loads it has to resolve all its external references, such as function names and variable names. If module can not find all of its unresolved names in the list of symbols that the kernel exports it will fail. For example, when you do **insmod** then the module will fail with the message "insmod: error inserting './my_module.ko': -1 Unknown symbol in module".

Many kernel source files specify export symbols using **EXPORT_SYMBOL** macro. For example, you can see **kernel/printk.c** file that has **EXPORT_SYMBOL**(console_printk) macro. The **console_printk** is an array of 4 int's that has been exported by the kernel. Similarly you can add your own variable or function that you like it to export from your kernel module source file. When you export symbols the space is allocated for these symbols somewhere in the kernel and other module or other kernel source files will be able to see these kernel symbols.

Example: (first module)

```
int my_flagCount=0;
EXPORT_SYMBOL(my_flagCount)
```


Kernel Threads

Threads are programming abstractions used in concurrent processing. A kernel thread is a way to implement background tasks inside the kernel. A background task can be busy handling asynchronous events or can be asleep, waiting for an event to occur. Kernel threads are similar to user processes, except that they live in **kernel space** and have access to **kernel functions** and **data structures**. To see the kernel threads running on your system, run the command:

```
ps –ef
```

The interface to create a kernel thread is as below:

```
int kthread_run(int (*fn)(void *), void * arg, const
char namefmt[])
```

This interface is defined in **arch/arm/kernel/process.c**. In this, a new process executes a function specified by fn argument and arg is passed as an argument to this function. The **kthread_run()** function is exported by a kernel and hence it can be accessed from kernel modules.

Example (second module)

To create a thread, use kthread_run() function and pass an address of thread function as an argument to the kernel thread.

extern int **my_flagCount**

```
static int flagCount(void * data)
{
    printk("::Inside Thread Function::\n");
    printk(KERN_INFO "%d.\n", my_flagCount++);
    schedule_timeout(30*HZ); // sleep for 30 seconds.
    printk(" Exiting Thread Function::");
}

int my_module_init( void )
{
  kthread_run((int (*)(void *))flagCount,NULL, “mythread”);
  return 0;
}
```

Reviewing the kernel output

```
# dmesg | tail -5
```

What to submit for Credit?

1. Source listing of both modules with a make file.
2. A initramfs image that will have your both modules.

