Kernel Modules

Operating Systems
Kartik Gopalan
Kernel Modules

• Allow code to be added to the kernel, dynamically

• Only those modules that are needed are loaded. Unload when no longer required - frees up memory and other resources

• Reduces kernel size.

• Enables independent development of drivers for different devices
Workings of a generic module / typical usage:
Implementing Kernel Modules
Hello World Kernel Module

http://www.cs.binghamton.edu/~kartik/examples/module/

```c
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("DUAL BSD/GPL");
// called when module is installed
int __init hello_init()
{
    printk(KERN_ALERT "mymodule: Hello World!\n");
    return 0;
}

// called when module is removed
void __exit hello_exit()
{
    printk(KERN_ALERT "mymodule: Goodbye, cruel world!!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```
Compiling the module

- Makefile
  - \texttt{obj-m := testmod.o}
  - [ For multiple files: \texttt{module-objs := file1.o file2.o} ]

- Compiling:
  - \texttt{$ make -C /lib/modules/$(uname -r)/build M=`pwd` modules}

- More details on kernel Makefiles
  - \url{https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt}
  - \url{https://www.kernel.org/doc/Documentation/kbuild/modules.txt}
Module Utilities

- **insmod hello.ko**
  - Inserts a module
  - Internally, makes a call to `sys_init_module`
  - Calls `vmalloc()` to allocate kernel memory
  - Copies module binary to memory
  - Resolves any kernel references (e.g. `printk`) via kernel symbol table
  - Calls module's initialization function

- **modprobe hello.ko**
  - Same as `insmod`, except that it also loads any other modules that `hello.ko` references.

- **rmmod**
  - Removes a module
  - Fails if module is still in use

- **lsmod**
  - Tells what modules are currently loaded
  - Internally reads `/proc/modules`
Things to remember

- Modules can call other kernel functions
  - Such as printk, kmalloc, kfree etc.
  - But only the functions that are EXPORTed by the kernel
    - using EXPORT(symbol_name)

- Modules (or any kernel code for that matter) cannot call user-space library functions
  - Such as malloc, free, printf etc.

- Modules should not include standard header files
  - Such as stdio.h, stdlib.h, etc.

- Segmentation fault may be harmless in user space
  - But a kernel fault can crash the entire system

- Version Dependency:
  - Module should be recompiled for each version of kernel that it is linked to.
Concurrent Issues

- Many processes could try to access your module concurrently.
  - So different parts of your module may be active at the same time

- Device interrupts can trigger Interrupt Service Routines (ISR)
  - ISRs may access common data that your module uses as well.

- Kernel timers can concurrently execute with your module and access common data.

- You may have symmetric multi-processor (SMP) system, so multiple processors may be executing your module code simultaneously (not just concurrently).

- Therefore, your module code (and most kernel code, in general) should be re-enterant
  - Capable of correctly executing in more than one context simultaneously.
Error handling

int __init my_init_function(void)
{
    int err;

    /* registration takes a pointer and a name */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;

    return 0; /* success */
}

fail_those: unregister_those(ptr3, "skull");
fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* propagate the error */

void __exit my_cleanup_function(void)
{
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    unregister_this(ptr1, "skull");
    return;
}

• In case of failure to go ahead; undo every registration activity
• But only those that were registered successfully
Module Parameters

• Command line:
  • `insmod hellon.ko howmany=10 whom="Class"`

• Module code has:
  static char *whom = "world";
  static int howmany = 1;

  `module_param(howmany, int, S_IRUGO);`
  `module_param(whom, charp, S_IRUGO);`

• See example module
  • [http://www.cs.binghamton.edu/~kartik/examples/module](http://www.cs.binghamton.edu/~kartik/examples/module)
Implementing character devices in Linux
Device Classification

• Character (char) devices
  • byte-stream abstraction
  • E.g. keyboard, mouse

• block devices
  • reads/writes in fixed block granularity
  • E.g. hard disks, CD drives

• network devices
  • message abstraction
  • send/receive packets of varying sizes
  • E.g. network interface cards

• others
  • USB, SCSI, Firewire, I2O
  • Can (mostly) be used to implement one or more of the above three classes
“Miscellaneous” Devices in Linux

- These are character devices used for simple device drivers.
- All miscellaneous devices share a major number (10).
- But each device gets its own minor number, requested at registration time.
Implementing a device driver for a miscellaneous device

- Step 1: Declare a device struct

```c
static struct miscdevice my_m misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "my device",
    .fops = &my_fops
};
```
Implementing a device driver for a miscellaneous device

- **Step 2**: Declare the file operations struct

```c
static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_close,
    .read = my_read,
    ...
    .llseek = noop_llseek
};
```

The function pointers that are not initialized above will be assigned some sensible default value by the kernel.
Implementing a device driver for a miscellaneous device

- **Step 3:** register the device with kernel
  - usually in the module initialization code

```c
static int __init my_module_init() {
    ...
    misc_register(&my_misc_device);
    ...
}
```

And don’t forget to unregister the device when removing the module

```c
static void __exit my_exit(void) {
    misc_deregister(&my_misc_device);
    ...
}
```
Implementing a device driver for a miscellaneous device

● Step 4: Implement the fops functions

```c
static ssize_t reverse_read(struct file *file, char __user * out, size_t size, loff_t * off)
{

    ....
    sprintf(buf, “Hello World\n”);
    copy_to_user(out, buf, strlen(buf)+1);
    ....
}
```

Don’t forget to
- allocate memory for buf
- Check if out points to a valid user memory location using access_OK()
- check for errors during copy_to_user()
GNU General Public License (GPL)

- Basis for all of the GNU software development, including Linux
- Allows users to modify software as they see the need
- Requires source code be distributed with binaries
- EXPORT_SYMBOL vs EXPORT_SYMBOL_GPL
  - Read http://lwn.net/Articles/154602/
- Device drivers need not be licensed under the GPL, but the mainstream ones are