CPSC 457 - Tutorial 17
Loadable Kernel Modules

Daniel de Castro

Department of Computer Science
University of Calgary

March 27, 2012
Announcements

Reminder

Homework 4 is due to March 31, 2012 at 11:59pm (Saturday)

- Thursday, March 29, 2012: ELF format overview
- Tuesday, April 3, 2012: More LKM examples (Interacting with OS)
- Homework 3 marking in progress (Thursday?)
What kernel modules are used for?

- There are six main things kernel modules are used for:
  - Device drivers
  - Filesystem drivers
  - System calls (create or override)
  - Network drivers
  - TTY line disciplines
  - Executable interpreters

- Creating modules for a kernel and compiling it into the base kernel\(^1\) can be time consuming (see Syscall exercise)

- It requires all the kernel source to be compiled

- What if the kernel crashes? Was it your new module or something else?

**LKM to the rescue!**

---

1) “Base kernel modules” refers to modules that were compiled with the kernel, in a single executable.
Why “Loadable” Kernel Modules?

- You don’t have to rebuild your kernel as often.
- LKMs are much faster to maintain and debug.
- LKMs help you diagnose system problems.
- LKMs can save you memory, because you have to have them loaded only when you’re actually using them.
- LKMs are not slower than base kernel modules.

**Note**

- Sometimes you have to build something into the base kernel instead of making it an LKM.
- **LKM are not user space programs.** They are part of the kernel, have free run of the system and can easily crash it.
Some problems related to LKM

Fragmentation penalty
The base kernel is always unpacked into real contiguous memory by its setup routines; Loading modules may cause the kernel to become fragmented, thereby introducing a minor performance penalty.

Binary compatibility (Linux)
Differences in internal structure and function between different kernel versions may cause compatibility problems.

- Symbol versioning data is placed within the .modinfo section of loadable modules.
- This versioning information can be compared with that of the running kernel before loading a module; if the versions are incompatible, the module will not be loaded.

Security LKM can be abused by attackers on a compromised system to prevent detection of their processes or files (rootkits).
LKM Utilities

Package *modutils* (most likely installed with your distro)

*insmod* Insert an LKM into the kernel

*rmmod* Remove an LKM from the kernel.

*depmod* Determine interdependencies between LKMs.

*kerneld* Kerneld daemon program

*ksyms* Display symbols that are exported by the kernel for use by new LKMs.

*lsmod* List currently loaded LKMs.

*modinfo* Display contents of .modinfo section in an LKM object file.

*modprobe* Insert or remove an LKM or set of LKMs intelligently. (If B requires A, loads A, then B.)
A simple example: “Hello, world!”

```c
/* hello.c
 *
 * "Hello, world" - the loadable kernel module version.
 */

/* Declare what kind of code we want from the header files */
#define __KERNEL__  /* We’re part of the kernel */
#define MODULE    /* Not a permanent part, though. */

/* Standard headers for LKMs */
#include <linux/modversions.h>
#include <linux/module.h>

#include <linux/tty.h> /* console_print() interface */

/* Initialize the LKM */
int init_module()
{
    console_print("Hello, world - this is the kernel speaking
");
    /* More normal is printk(), but there’s less that can go wrong with
       console_print(), so let’s start simple.
    */
    return 0; /* non-zero value means an error */
}

/* Cleanup - undo whatever init_module did */
void cleanup_module()
{
    console_print("Short is the life of an LKM\n");
}
```

Announcements

Introduction

What kernel modules are used for?
Why “Loadable”?
Some problems

Putting into practice

LKM utilities
A simple example
Compiling a LKM
Writing a LKM from scratch
A simple example: “Hello, world!”

Updated version

```c
/* hello.c
 *
 * "Hello, world" - the loadable kernel module version.
 *
#include <linux/module.h> /* Needed by all modules */
#include <linux/init.h>  /* Needed for the macros */
#include <linux/tty.h>   /* console_print() interface */

/* Initialize the LKM */
static int __init hello_start()
{
    console_print("Hello, world - this is the kernel speaking\n");
    /* More normal is printk(), but there’s less that can go wrong with
     * console_print(), so let’s start simple.
     */
    return 0; /* non-zero value means an error */
}

/* Cleanup - undo whatever init_module did */
static void __exit hello_end()
{
    console_print("Short is the life of an LKM\n");
}
module_init(hello_start);
module_exit(hello_end);
```
Compiling a LKM

- Compile and generate ELF object file
  
  \[
  \text{\$ gcc -c -Wall -nostdinc -I /usr/src/linux/include hello.c}
  \]

- Until Kernel 2.4, the loader would link the object file to the running kernel, so we would be done!

- From Kernel 2.6, the kernel does the linking. Additional information is necessary.
  
  - \texttt{modpost} reads ELF object file (.o) and creates a C source describing additional sections (.mod file).
  
  - Compile .mod file and link it with the original object file (.o) to make the actual module (.ko extension).

Note

We will use Makefile to automatize this process.
Writing a LKM from scratch

It’s your turn!!!