The E-ALE (Embedded Apprentice Linux Engineer) is a series of seminars held at existing conferences covering topics which are fundamental to a Linux professional in the field of Embedded Linux.

This seminar will spend equal time on lecture and hands on labs at the end of each seminar which allow you to practice the material you’ve learned.

This material makes the assumption that you have minimal experience with using Linux in general, and a basic understanding of general industry terms. The assumption is also made that you have access to your own computers upon which to practice this material.

More information can be found at https://e-ale.org/

This material is licensed under CC-BY SA4
# Contents

1 Modules
  1.1 Memory Overview ................................................................. 2
  1.2 What are Modules? ................................................................. 3
  1.3 Aren’t modules and drivers the same thing? .............................. 4
  1.4 Sample Module .......................................................................... 5
  1.5 \_\_init and \_\_initdata ............................................................... 6
  1.6 \_\_exit ................................................................................... 7
  1.7 Module Licenses ........................................................................ 8
  1.8 Exporting Symbols .................................................................... 9

2 Building and Loading Modules .............................. 11
  2.1 Building Modules ...................................................................... 12
  2.2 Building Modules-pt2 ............................................................... 13
  2.3 Loading Modules ...................................................................... 14
  2.4 Loading Modules-pt2 ............................................................... 15
2.5 Listing and Unloading Modules ................................................................. 16
2.6 Labs ........................................................................................................ 17
Chapter 1

Modules
1.1 Memory Overview

Memory Overview

4GB (for 32bit sys)

PAGE_OFFSET

kernel code & data

application virtual address space

user code and data
1.2 What are Modules?

- A way to dynamically add code (or data) to a running kernel
- ... And to optionally remove that same code (or data).
- A way to automatically load only the drivers we need
- A way to provide choice between multiple drivers
- A way to architect kernel code in a modular way
1.3 Aren’t modules and drivers the same thing?

- While many drivers are built as modules
- ... not all modules are drivers
- ... and a kernel module can be a driver, library code, debug code, or data
- Modules provide an alternative loading mechanism for code (and data)
- Drivers provide code that controls devices
1.4  Sample Module

Sample Module Code

```c
#include <linux/module.h>
#include <linux/init.h>

static int __init my_init(void)
{
    pr_info("Hello: module loaded at 0x%p\n", my_init);
    return 0;
}

static void __exit my_exit(void)
{
    pr_info("Bye: module unloaded from 0x%p\n", my_exit);
}

module_init(my_init);
module_exit(my_exit);

MODULE_AUTHOR("Wile E. Coyote, Super Genius");
MODULE_LICENSE("GPL v2")
```

1.5 _init and __initdata

Examples:

```c
static int some_data __initdata = 1;
void __init somefunc(void) { ... };
```

- Frees code and data after initialization is complete
- For code/data used during initialization - only!
- At the end of the kernel boot messages:

  `[0.841689] Freeing unused kernel memory: 424k freed`
__exit

Examples:

void __exit somefunc(void) { ... };

- When linked into kernel
  - Will never be called
  - ... So, code omitted
- When in a loadable module
  - code remains for when module is unloaded.
1.7 Module Licenses

<table>
<thead>
<tr>
<th>License</th>
<th>Meaning</th>
<th>Taints</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPL</td>
<td>GNU Public License V2 or later</td>
<td>No</td>
</tr>
<tr>
<td>GPL v2</td>
<td>GNU Public License V2</td>
<td>No</td>
</tr>
<tr>
<td>GPL and additional rights</td>
<td>GNU Public License V2 rights and more</td>
<td>No</td>
</tr>
<tr>
<td>Dual BSD/GPL</td>
<td>GNU Public License V2 or BSD License</td>
<td>No</td>
</tr>
<tr>
<td>Dual MPL/GPL</td>
<td>GNU Public License V2 or Mozilla License</td>
<td>No</td>
</tr>
<tr>
<td>Dual MIT/GPL</td>
<td>GNU Public License V2 or MIT License</td>
<td>No</td>
</tr>
<tr>
<td>Proprietary</td>
<td>Not GPL compatible</td>
<td>Yes</td>
</tr>
</tbody>
</table>
1.8 Exporting Symbols

- Symbols can be exported at run-time
- Symbols must be exported for modules to access them
- The kernel exports many symbols
- And modules can export symbols for other modules
- Symbols are exported with `EXPORT_SYMBOL()`
- You can export variables and functions
- Examples:
  ```c
  int myvar;
  int myfunc(int val);
  EXPORT_SYMBOL(myvar);
  EXPORT_SYMBOL(myfunc);
  ```
Chapter 2

Building and Loading Modules

2.1 Building Modules ................................................................. 12
2.2 Building Modules-pt2 .............................................................. 13
2.3 Loading Modules ................................................................. 14
2.4 Loading Modules-pt2 .............................................................. 15
2.5 Listing and Unloading Modules ............................................... 16
2.6 Labs .................................................................................. 17
2.1 Building Modules

- A module is kernel code, loaded late so...
  - The build system needs access to kernel headers
  - The build system needs to know the kernel options
- Code for the kernel you are running need to be available and built - at least partially
  - Install kernel headers package
  - (or) In kernel source code tree run: `make modules_prepare`
  - (or) Build complete kernel and it matches running kernel
## 2.2 Building Modules-pt2

### Building Modules

- **Step 1:** Create a simple one line Makefile:
  
  ```
  obj-m += mymodule.o
  ```

- **Step 2:** Call make
  
  ```
  make -C <path to kernel> M=$(pwd) modules
  ```
  
  or for embedded
  
  ```
  make ARCH=<arch> CROSS_COMPILE=<tool prefix> \
  -C <path to kernel> M=$(pwd) modules
  ```

- **Examples:**
  
  ```
  $ make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
  $ make ARCH=arm CROSS_COMPILE=arm-linux- \
  -C /home/wecsg/linux M=$(pwd) modules
  ```
2.3 Loading Modules

- Two methods to load modules
  - Manual load
  - Install and load from installation

- Manual load:
  $ sudo insmod <path to>/mymodule.ko

- Install:
  $ sudo make -C <path to kernel> M=$(pwd) modules_install
  or for embedded:
  $ sudo make ARCH=<arch> CROSS_COMPILE=<tool prefix> \
     MOD_INSTALL_PATH=<path to rootfs> \
     -C <path to kernel> M=$(pwd) modules_install
2.4 Loading Modules- pt2

- Manual install
  - step 1: copy module to /lib/modules/<kernel version>/<subdir>
    (note: for embedded, it would be such a path in your target’s root filesystem)
  - step 2: run depmod
  - Example:
    ```
    $ sudo cp mymodule.ko /mnt/embrootfs/lib/modules/5.2/misc
    $ sudo depmod -b /mnt/embrootfs
    ```
- Load from installation:
  ```
  $ sudo modprobe mymodule
  ```
2.5 Listing and Unloading Modules

** Listing and Unloading Modules **

- **List loaded modules:**
  
  ```
  $ lsmod
  Module                  Size  Used by
  Evdev                   13811  1
  uio_pdrv_genirq         4205   0
  uio                     11036  1 uio_pdrv_genirq
  usb_f_mass_storage      51462  2
  ```

- **Unload modules:**
  
  ```
  $ sudo rmmod <module name>
  ```
  
  - No .ko is specified
  - If you accidentally add the .ko, rmmod strips it off which usually works
  - Example:
    
    ```
    $ sudo rmmod mymodule
    ```
2.6 Labs

Exercise 2.1: Writing Simple Module

Write a module using the following template:

1. Load the module using `insmod`
2. Use `dmesg` to see output from the `pr_info()` routines
3. Remove the module from memory using the `rmmod` command
4. Install the module using `make modules install` or by installing manually
   ... and then load the module using `modprobe`

Exercise 2.2: Exporting Symbols

Write two modules, one which uses a symbol provided by the other

1. Insert the modules in the correct order.
2. Then Insert the modules out of order.
3. Install the module using `make modules install` or by installing manually
   ... and then load the modules using `modprobe`
   If module A depends on the symbol in module B, you should be able to `modprobe` A and see B get loaded automatically