Overview+ I

Overview+ II

- Initialization and finalization
  - Finalization
  - Errors in loading
- Module parameters
- Use modules in userspace

Environment setting

- Ubuntu GNU/Linux
- Using latest LST Ubuntu 16.04
  - kernel Linux v 4.4.0

Environment setting

- kernels 2.6 and newest use kernel source tree,
- previous kernels used only kernel headers,
- apt-get install ...,
- apt-get install build-essential,
- ls -d /lib/modules/$(uname -r)/build.

Hello world

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
  printk(KERN_ALERT "Hello, world\n");
  return 0;
}
static void hello_exit(void)
{
  printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Hello world - description

- MODULE_LICENSE
- module_init()
- module_exit()
- printk()
- KERN_ALERT

Hello world - compile

```
% make
make[1]: Entering directory ‘/usr/src/linux-2.6.10’
CC [M] /home/ldd3/src/misc-modules/hello.o
Building modules, stage 2.
MODPOST
CC /home/ldd3/src/misc-modules/hello.mod.o
LD [M] /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory ‘/usr/src/linux-2.6.10’
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Kernel modules and applications

- modules are similar to event-driven applications
- example: hello_exit
- if an application does not clear assets OK, not critical
- if a module does not clean the resources NOT OK (only when restarted)
- module is only linked to the kernel
- applications use many libraries

Kernel modules and applications

- cannot use "standard" headers
- the only exception is <stdarg.h>
- we can use only headers defined in kernel tree
- segfault in testing phase OK
- kernel fault can kill the whole system

Concurrency

- kernel modules must be coded with concurrency in mind
- multiple processes can request the same functionality at the same time
- all code must be reentrant
  it should be able tu run in multiple contexts

Reentrancy

- In computing, a computer program or subroutine is called reentrant if it can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. *Wikipedia*
- program or sub-program is reentrant if:
  - it can be interrupted in the middle of the execution,
  - can continue from the same place it was interrupted,
  - usually it is interrupted by an IRQ,
- Reentrant functions CANNOT call non-reentrant functions!

Non – reentrant function – problem: static char buffer

```
/* non-reentrant function */
char *strtoupper(char *string){
  static char buffer[MAX_STRING_SIZE];
  int index;
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Reentrant function: returns dynamicaly allocated result

```
/* reentrant function*/
char *strtoupper(char *string){
  char *buffer;
  int index;
  /* error-checking should be performed! */
  buffer = malloc(MAX_STRING_SIZE);
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Hello world example

- obj-m := hello.o
- (GNU make): make module from object file hello.o
- produce hello.ko

Extend the example

- obj-m := module.o
- module-objs := file1.o file2.o
- module.ko from multiple object files file1.c file2.c

Start the make

```
make -C ~/kernel-2.6 M=‘pwd‘ modules
```

make

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Install and remove modules

- insmod
- rmmod
- modprobe

Version dependency

- Recompile is needed for each kernel version
- Error message:
- insmod hello.ko
- Error inserting './hello.ko': -1 Invalid module format

Basics

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

- *module.h* definitions and symbols for modules
- *init.h* initialization and cleanup functions
- *moduleparam.h* parameter support (not mandatory)

Basics, continue

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR (stating who wrote the module),
MODULE_DESCRIPTION (a human-readable  statement of what
the module does),
MODULE_VERSION
```

- *MODULE_LICENSE* license, if omitted kernel will be tainted (When the kernel is tainted, it means that it is in a state that is not supported by the community.)
- *MODULE_AUTHOR* author description (text)
- *MODULE_DESCRIPTION* module description (not mandatory)
- *MODULE_VERSION* module version (not mandatory)

Initialization

```
static int __init initialization_function(void)
{
/* Initialization code here */
}
module_init(initialization_function);
```

- *__init* (not mandatory), kernel will delete this code after init
- *module_init* defines where is initialization function

Finalization

```
static void __exit cleanup_function(void)
{
/* Cleanup code here */
}
module_exit(cleanup_function);
```

- *__exit*(not mandatory), some kernels can ignore exit functions (if exit is prohibited)
- *module_exit* defines where the cleanup function is

Errors in loading

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;
void my_cleanup(void){
  if (item1)
    release_thing(item1);
  if (item2)
    release_thing2(item2);
  if (stuff_ok)
    unregister_stuff( );
  return;
}
int __init my_init(void){
  int err = -ENOMEM;
  item1 = allocate_thing(arguments);
  item2 = allocate_thing2(arguments2);
  if (!item2 || !item2)
    goto fail;
  err = register_stuff(item1, item2);
  if (!err)
    stuff_ok = 1;
  else
    goto fail;
  return 0; /* success */
  fail:
  my_cleanup( );
  return err;
}
```

Module parameters

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

- insmod hello howmany=10 whom="mama"

Use modules in userspace - pluses

- Using the entire C library.
- "Normal" debugging.
- Testing: if the program hangs, we kill it, in the kernel ...
- Shared memory for large drivers (swap).
- Licensing for closed-source drivers.

Use modules in userspace - minuses

- Direct access to memory is possible only through mmapping /dev/mem (root).
- Access to the I/O port was only possible after the ioperm or iopl call (not on all platforms).
- Access to /dev/port can be too slow.
- The time to answer is long, because we need a context switch.
- The driver can be on a swap disk, the call will be really late.
- Unable to access several devices:
  network, block, ...