Kernel Module Programming

Alessandro Barenghi

Dipartimento di Elettronica e Informazione
Politecnico di Milano

barenghi - at - elet.polimi.it

May 26, 2011

Recap

By now , you should be familiar with...

- Programming with sockets employing different protocols
- System programming, synchronization primitives and IPC
- System administration skills , as far as the local host and network monitoring go
- Network administration and filtering, tunnels and NAT

Lesson contents

Overview

- Linux kernel structure (brief)
- Kernel Modules programming
- Simple kernel module
- A proc entry
- A sample character device

Linux

A bit of history

- The Linux kernel development started back in 1991
- The first release was developed to have a working, simple OS, no strings attached
- In 20yrs, the codebase has grown from 140k LOC to 14M LOC
- At the moment, the most used monolithic kernel around

Macrokernel

Monolith and modules

- The Linux kernel is based on a monolithic structure and is fully written in C[a]
- All the code thus shares the same namespace and memory
- The whole code runs with the highest possible privileges on the CPU (the so-called *supervisor mode*)
- Simple, performing but with some safety issues (concurrency handling)
- Microkernel alternatives have a different structure, but choosing one or the other strategy is a long standing issue

[a]plus some assembly for the syscalls/drivers backend obviously

Macrokernel

Key areas

- The Linux kernel is logically split in 6 master areas
  - System management : bootup, shutdown, syscall interfaces
  - Process management : scheduling, inner locks and mutexes, synchronization primitives
  - Memory management : Memory allocator, page handler, virtual memory mapper
  - Storage management : file access primitives, virtual filesystem management, logic filesystem management and disk handling
  - Networking management : network syscalls, socket bufffer handling, protocol and filtering handling, network drivers
  - User Interaction management : character devices, security management, process tracing management and HI devices management

Module structure

What's in a module

- A kernel module is a binary blob, which can be linked at insertion time with the whole kernel
- Think of it as a sort of a “strange” static library
- The linking is performed only against kernel symbols: no libc around here...
- Particular care should be exercised before calling kernel symbols prefixed by a double underscore, as they represent lower level functions

Module structure

Differences from processes

- The module is not “run” but rather called when its services are needed (similar to event based programming)
- There is no regular dynamic memory allocator, as we are directly on the fence side where physical memory can be accessed
- There is no automatic cleanup when a module is removed, noone will free memory, noone will rebind the things as they were before
- Albeit there is a concept of “running” process, it is almost impossible to understand what calls you
- No floating point operations available, sorry

A simple module

Contents

- A module is constituted of one (or more) C files, containing a collection of functions
- Two functions are mandatory
  - `init_module` performs all the initializations of the resources at insertion time
  - `cleanup_module` performs the pre-removal cleanup
- All the variables declared in the global scope of the module are actually residing in kernel memory
- The stack of the module is shared with all the others kernel functions (i.e. the kernel has a single stack) and it's rather small
- Dynamic memory allocation encouraged for large variables as they would clutter the stack

A simple module

Building

- In order to build a module,you just need the usual gcc compiler
- To specify that a kernel module object must be built, the `obj-m` target is used in the Makefile
- You will need at least the Linux kernel header files to compile a module[a]
- If you are planning to do heavy modifications[b], a full kernel source tree will be required

[a] available as a handy package under Debian
[b] say, adding syscalls

A simple module

Module Management

- Once a module has been successfully built, you can check informations about it via the `modinfo` command
- Module insertion is performed via the `insmod` command, while removal is done via `rmmod`
- You can obtain a list of the inserted modules via the `lsmod` command
- It is pretty obvious that only root can insert and remove kernel modules
- The kernel ring buffer (where log messages appear) can be accessed via the `dmesg` command

A simple module

Licensing and Author

- Every module has an author (to be blamed or praised) and is released under a specific license
- Beside the purely legal issues, module licensing affects the behaviour of the kernel at insertion time
- It is commonly said (and tools will report so) that a non GPL-licensed module will “taint” the kernel
- In particular, as the non GPL modules may not be available for source code inspection some debugging facilities may be disabled
- Moreover, bug and compatibility issues with tainting modules are dealt less readily by the kernel development team

A simple module

Parameter passing

- It is possible to pass parameters to a module at insertion time
- The parameter parsing is done according to the call to the `module_param` primitive
- The `module_param` primitive accepts the name of the parameter, the type and the permission for changing it, if it will be exposed via sysfs
- It is possible also to pass arrays as parameters via the `module_param_array` function
- The `module_param_array` behaves in a similar way to the `argc-argv` mechanism in userspace programs

Dynamic Memory allocation

kmalloc

- The most simple way to get dynamic memory in kernelspace is the use of the `kmalloc` primitive
- The primitive directly calls the `__get_free_pages` function appropriately, so space is available only in page sized chunks
- There is an upper limit for the maximum size of a `kmalloc`: portable code should not use more than 128kB per shot
- The `kmalloc` primitive can be invoked with different flags to steer the behaviour of the memory allocator, in particular
  - `GFP_KERNEL` is the default behaviour flag, may block and put to sleep the current process
  - `GFP_ATOMIC` is specifies that the current process should not be put to sleep and can claim up to the last page available
- `kfree` frees the memory claimed with `kmalloc`

Dynamic Memory allocation

vmalloc

- If you are not in need of physically contiguous memory, you may use the `vmalloc primitive`
- The `vmalloc` calls the page handler at a higher level resulting in an allocation of an arbitrarily large amount of memory
- Since the call depth is greater than `kmalloc`, `vmalloc` is obviously less performing that `kmalloc`
- As before, you can (and must)free the memory via `vfree`

Concurrency handling

Concurrency issues

- As we now know, the Linux kernel is one large monolith as far as the running code goes with the same address space accessible for all the modules
- Once upon a long time ago, when the systems had a single processor and the kernel structure was simpler, only one task would have been executed at once in kernelspace
- Still , hardware interrupts could get in the way of atomic operations being performed
- Then multiprocessor system started being supported back in 1996, starting to cause the first, serious concurrency issues
- The whole thing got a lot worse when the whole kernel became preemptible with the 2.6 series (around mid 2002 with 2.5.37)

Concurrency issues

Solutions available

- As the concurrency issues are pretty serious, the kernel offers native facilities to prevent problems
- Fully atomic variables are available
- Semaphore-structures were implemented since a long time ago
- Spinlocks represent the main difference between userspace and kernelspace concurrency handling mechanisms (used most of the time)
- Read-Copy-Update mechanisms are available to provide advanced and performant concurrency handling (especially useful for NetFilter)

Concurrency issues

Atomic Variables

- In case the resource which may be shared among different kernel parts is a simple integer
- In this case, it is possible to avoid complex concurrency handling structures via the use of atomic variables
- The `atomic_[set|add|inc|dec|sub]` provide the means to atomically perform that operation on the integer value
- Operations on atomic variables are usually extremely fast, as they are compiled as single assembly instructions if the architecture allows so
- A companion primitive set is the `atomic_*_and_test` group which check if the operation was correctly performed afterwards and are useful to implement election mechanisms

Concurrency issues

Spinlocks

- Spinlocks are mutual exclusion primitives akin to common mutexes
- The main difference is that a spinlock will never be put to sleep until it gains access to the resource
- Spinlocks are structures of `spinlock_t` type (defined in `spinlock.h`)
- Different locking and unlocking functions are available
    - `spin_lock` and `spin_unlock` are the garden variety spinlock
    - `spin_lock_irqsave` and `spin_unlock_irqsave` will mask hardware interrupts and restore the IV state after the lock has been resolved
    - `spin_lock_bh` and `spin_unlock_bh` only mask software interrupts

Communications with the real world

Devices

- In order to expose a unified interface for communication with the hardware, the kernel exposes devices
- Following the UNIX philosophy, the devices are seen in userspace as simple files
- It is possible to either expose a real device via a block/character interface (e.g. /dev/sda)
- Or to build a mockup device which may be useful (/dev/zero)
- A simpler alternative, if there is only the need to communicate between userspace and kernelspace is the proc filesystem

Communications with the real world

Creating proc entry

- The proc filesystem is a mockup filesystem with no structures whatsoever on any permanent storage device
- A proc filesystem entry does not need to have a full and proper device structure
- The only two operations which are mandatory to implement are read and write
- The proc filesystem takes care of handling the open/close/stat syscalls properly and only requires the “filling” for the buffers
- The file is automatically created in the /proc directory and does not need to be physically removed from the filesystem (as there’s no actual fs :))

Communications with the real world

A real device

- A real character device needs to implement all the possible operations which can be performed on it
- Moreover, it is required to handle the number of stakeholders which are actually using the device to avoid improper removal of the module
- The devices are accessible from the userspace via a peculiar filesystem entry, which does not have any actual space reserved on disk known as device node
- Real devices are split into :
    - Character devices: minimum unit for access : char, usually unbuffered
    - Block devices: minimum unit for access : block, usually buffered

Communications with the real world

Device implementation

- We will see the implementation of a mockup character device[a]
- A character device needs to implement at least four key primitives : `open`,`read`,`write` and `release`
- It also needs to take into account whether someone is using the device in order to prevent premature module removal
- The transferral of the data from kernel to user address space is managed by the `put_user` primitive

[a]Block devices go the same way, just with more functionalities

Communications with the real world

Node setup

- A device node can be created via the `mknod` utility and needs three parameters
    - The type of the device (block or character device)
    - The major number, i.e. a unique, kernel assigned, identifier for the device
    - The minor number, a sub-index handled by the module answering for that device in kernelspace
- A list of all the devices exported by the kernel is available via `/proc/devices`
- It is possible also to avoid static devices via the udev filesystem, which is automatically populated by the kernel[a]

[a] say, the partitions of a hard disk