Device Drivers

Alessandro Barenghi

Dipartimento di Elettronica e Informazione
Politecnico di Milano

alessandro.barenghi - at - polimi.it

June 3, 2014

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 debug filesystem

Communications with the real world

Quick debugging I/O

- Originally, the proc filesystem served as both a quick debugging interface and to expose a parameter passing interface to the kernel parts
- In the current Linux Kernels, these two roles have been split and implemented in the DebugFS and SysFS respectively
- It is thus possible to obtain a quick, file based communication interface through creating a file in DebugFS
- The read/write callbacks must be implemented by the module developer and handle the common read/write operations on the file
- A directory structure can be easily created via the exposed API to organize the output

Communications with the real world

Copying to- and from- userspace

- One key point in user-kernel communication is copying data across different address spaces
- Despite some architectures (e.g. x86_64) map the kernel at the end of the process address space after the stack, some won't → the copies may need address translation from one virtual address to another
- Two ways to perform copies are available in kspace:
  - `get_user(val, ptr)` and `put_user(val,ptr)` are quick *macros* which are able to copy a single value with the same type of the pointer `ptr` into `val`
  - `copy_from_user` and `copy_to_user` are actual functions performing like `memcpy` with integrated address translation

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 : single character (one byte), usually unbuffered
  - Block devices: minimum unit for access : a block of data (a contiguous chunk in the kB size range), 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

Device node setup

- Once a device has been registered into Linux's device tree, its interface should be made available to users
- Three methods are available, depending on the degree of automation you desire, and the init system of your choice
  - Manual creation of the device node via `mknod`
  - Automatic device creation via `udev`
  - Device handling via a `systemd` unit

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`

Communications with the real world

`udevd`

- The `udevd` daemon is in charge of monitoring which devices are registered and act according to predefined rules
- The most typical example is automatic mounting/unmounting of filesystems upon disk insertion (e.g. with USB thumb drives)
- `udevd` reads a set of text files, the rules, usually located in `/etc/udev/rules.d`
- Upon triggering of a rule (e.g. device registration) `udevd` automatically creates the node file with the specified permissions