QLOOP
Linux driver to mount QCOW2 virtual disks

Francesc Zacarias Ribot
francesc.zacarias@est.fib.upc.edu
Facultat d'Informatica de Barcelona

June 23, 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Virtual machines have become a common tool in many information-technology based companies. All the hardware of these virtual systems (or guests) is defined by software (the virtualizer running on the host), except for the permanent storage or hard disks. These drives store data and the OS of the guest, effectively saving the state of the entire virtual system. They define what the guest can and cannot do. They are represented as regular files in the host system.

Having to manage a few virtual machines everyday at work, I realized the importance of being capable of modifying these files. Create a new guest, reconfigure an existing one or perform basic maintenance operations such as backups are common tasks which are not easy to carry out because there is no obvious way to modify these files directly, even though they are just regular files lying on your system.

There are many successful virtualizers in use today, and each of them defines its own format for their disks files. I'm most familiar with qemu and kvm, a pair of free-software projects. An emulator and a virtualizer, respectively. While at their core they implement radically different approaches for the guest's CPU execution, both rely on the same end-user application and therefore use the same format for disk files.

I chose to base my project on qemu and kvm software because:

- I've used it extensively, so I know a lot about it (from a user point of view, though).
- Because of the previous point, I'm interested in developing a tool to improve the everyday use of the software I employ.
- It's free-software, so I have access to plenty of documentation and the whole code without any constraints.

The format employed by both qemu and kvm is called QCOW2 and sup-

ports a large number of features such as compression, encryption and snapshots. Also, it is common for these kind of files to only take up as much space as it's in use. That is, a disk file representing a virtual disk of 500GB with only 100MB in use will take up 100MB approx, a bit more due to overhead (headers and control structures). This an important feature of virtual disks and support is a must.

While there are a few tools to modify QCOW2 files, these are cumbersome, slow or incredibly wasteful in time and computer resources. The goal of this project is to create a tool which allows easy and fast interaction with these files. Such functionality cannot impose changes to the file structure: there should be no difference between editing the disk file through the virtualizer or doing it through this project's software.

1.2 Goals

Develop a module for Linux which creates a new block device from a QCOW2 file. This block device would allow basic read and write operations and translate them to appropriate modifications to the QCOW2 file. The main advantage of this approach is that any application that operates with block devices (such as disk partitioners, low-level file-system tools and even software RAID) will work seamlessly with virtual disks.

Development is based on the loop module: an existing driver in Linux which creates a block device from a regular file. This module does a 1:1 conversion. So, the resulting block device's offset addresses match exactly with the file's. This basic functionality can be modified to fit the interests of this project, which involve:

- Reading the header of the QCOW2 file during set up.
- Read and maintain the multiple control structures living inside the QCOW2 file.
- Due to the way QCOW2 files are laid out, data is kept in the order that is written, and not in the logical position in the disk (offset 0 of the virtual disk does not mean offset 0 on its file), so a proper translation must be performed on every access.
- Translation is performed with a set of tables spread out through the disk. These must be traversed for every access to ensure consistence.

Thanks to the inheritance from the loop module, the basic interface for disk read/write operations is already taken care of. Because of this, and as a homage, the module will be called qloop. Also, the command line tools used for loop devices can be used for this new module, since the interface is the same.

The new module should achieve a reasonably good performance. Otherwise said, working with this device should not entail a major penalty in comparison to working with a regular file or with the basic loop module. Finally,

advanced features such as snapshots, compression and encryption are intentionally left out due to time constraints. Still, the design employed for the module should lay out the foundations to eventually support these features without doing major changes to existing code.

Chapter 2

State of the Art

A brief research was made at the beginning of this project to find other existing tools that provide the same functionality. While there are a few applications that try to solve the problem of accessing a virtual disk file, none meet all the requirements explained on the previous section. Below lies a list of the alternatives found, with a description of their design and their advantages and disadvantages respective to the solution chosen by this project.

2.1 FUSE

FUSE (acronym for Filesystem in USErspace) is a library and a kernel module which allows non-privileged users to create their own file systems without editing the kernel code [6]. This system works by developing an executable linked against libfuse. Upon mounting the file-system with a special application included in the FUSE package (fusermount), it really works as if a new traditional file-system has been created. When an application issues a file operation (open, read, write...) upon a file in this new file-system, the kernel (through the FUSE module) queries the libfuse-linked executable. This way, the executable is the actual implementation of the file-system.

This method allows to create new file-systems, that is to set the contents of a directory tree (the mount point) and the meaning of the operations on them. It is possible to read and write QCOW2 files this way. Unfortunately, FUSE is not a appropriate solution because we need low-level access to the contents of the virtual disk. Simple administration tasks such as creating partitions or running fsck require this kind of access that FUSE can't provide.

2.2 libguestfs

libguestfs is a very young project (it started barely a year ago) developed by Red Hat Inc. It is a library for accessing and modifying virtual machine (VM) disk images. Upon invocation, this library launches a QEMU instance as a

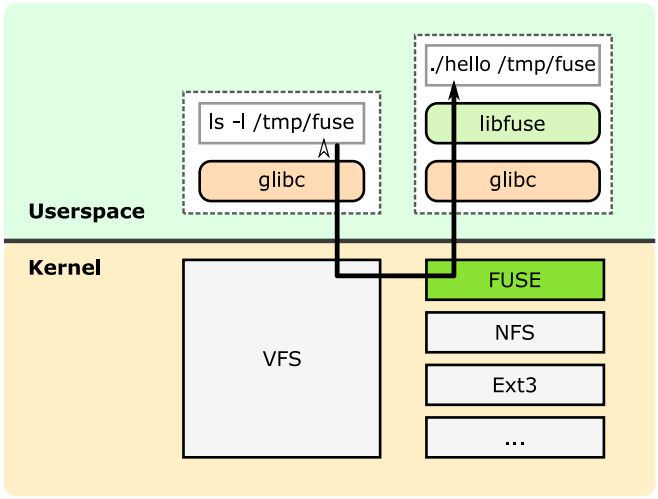


Figure 2.1: FUSE architecture

child process. This instance runs a Linux kernel, a collection of tools to access block devices and a daemon process used to receive the library's requests [7]. It has bindings for a wide collection programming languages and includes a few commands to be used in shell scripts.

This original approach has the advantage that libguestfs gains the same features as QEMU, without having to add a single line of code to the kernel or QEMU projects. Because of this, all virtual disk operations are kept in userspace. So, if a future version of QEMU were to include support for a new virtual disk format, libguestfs will not need many changes to support it too, since the low-level handling is performed by QEMU itself.

On the other hand, it presents a lot of disadvantages. Every time a virtual disk is opened, a new QEMU instance must be launched. Even if it is running a specially crafted OS, it still is a very resource consuming operation, both on CPU time and system memory. The API is incredibly big and confusing, because instead of allowing low-access to the virtual disk, it tries to replace the functionality of a big collection of userspace tools such as LVM2, e2fsprogs and even the coreutils package (chmod, cp, rm...), just to name a few. Finally, its contrived design makes it very hard to install. Not only you must compile the library and its necessary bindings, but also build the QEMU image and its contents, and keep everything compatible with your current OS. If libguestfs has not been packaged for your Linux distribution, setting it up will take too much time and effort. Since this project is maintained by Red Hat Inc., it is mainly developed on Fedora and it is the only distribution that includes it in its repositories. A package for Debian and Ubuntu is available on the project's main site.

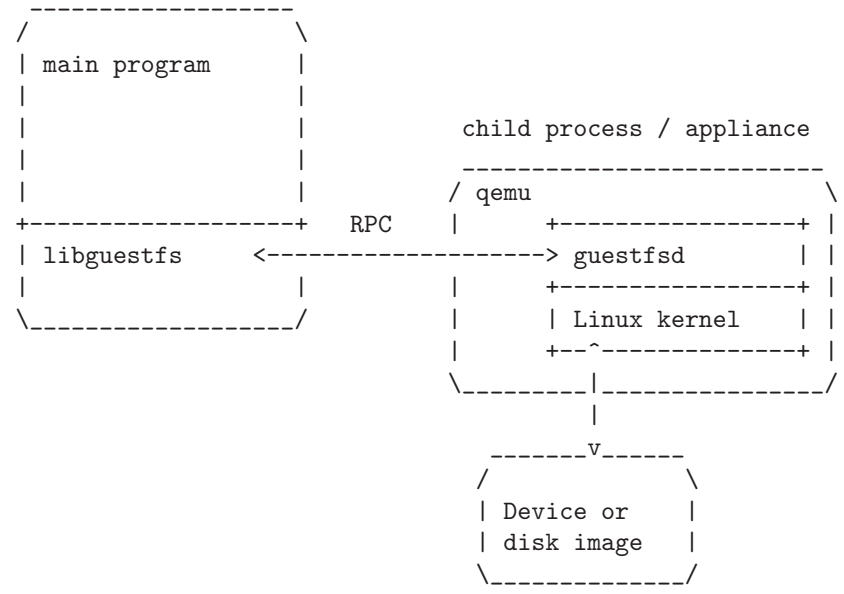


Figure 2.2: libguestfs architecture

2.3 qemu-nbd

On version 0.10 of QEMU, a new tool was added: `qemu-nbd` [9]. This application relies on the network block device (NBD) infrastructure to work and it is the official answer of the QEMU developers to the need of managing virtual disk files, since this is included with the QEMU package.

The network block device is a system which allows to access a block device remotely as if it was local. This is achieved with a couple of command-line tools and a kernel module, which is part of the Linux Kernel since version 2.1.101 [8]. The server system, where the block device is physically connected, must run the `nbd-server` program to make such device available for clients. Then, the client system runs `nbd-client` to create a new block device (let's call it `/dev/nbd0`) which represents the same device as in the server. For this to work, the client must be running a Linux kernel with the nbd module loaded. From now on, any operation executed on `/dev/nbd0` on the client will perform exactly the same as if it was run locally on the server on the original device. Note that this is not the same as distributed file-systems such as NFS or CIFS. These file-systems share high-level representation of data (files and directories) while NBD shares a raw block device (low-level). With NBD, it is possible for the client to run fsck on the shared device, edit its partitions or make it part of a RAID.

`qemu-nbd` is a replacement of `nbd-server`. Instead of picking a block device to be shared through the network, a virtual disk file is opened and its contents decoded on-the-fly so that clients "see" a regular block device. Advantages of this approach are the re-utilization of existing infrastructure, using QEMU to do the low-level access and keeping all the operations in user-space (except creating the NBD device in the client, which requires a kernel module).

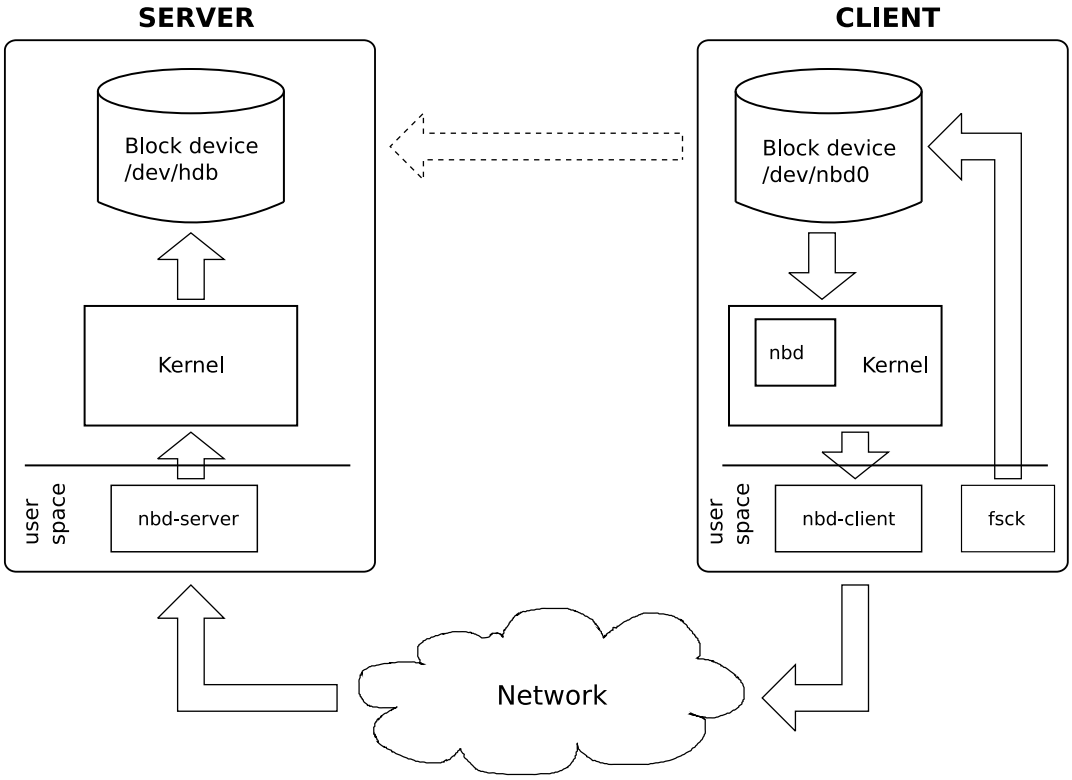


Figure 2.3: NBD architecture

On the other hand, this design has serious flaws. It is noticeably slow. Moreover, the client and the server cannot be the same host because a deadlock will occur. There cannot be any type of concurrent write access to the block device. That is, after a NBD link has been established between two systems, the server cannot write the device, nor can any other client because there is a high chance of data corruption. These shortcomings are not specific of `qemu-nbd`, but due to the way NBD itself is designed.

2.4 Convert to raw format

This method consists in converting the QCOW2 file into a common disk image or raw file. This raw file is a 1:1 copy of a disk, where all the disk contents are written sequentially (byte x at offset n on the disk is equal to byte y at offset n on the file). Raw files can be accessed as a block device with the help of the loop module. After all modifications have been done, it is possible to convert back the image into QCOW2 format and using it as a virtual disk with QEMU.

This approach relies on the loop device, which is part of the Linux kernel and `qemu-img`, a tool shipped with QEMU and responsible for disk files creation, translation and other useful operations. Also, read and write operations over a raw file through loop are fast, and allow low-level access to the device.

The main disadvantage is the need to convert the entire disk to raw format. This is a very resource intensive operation, both for CPU time and disk space. QCOW2 files only take up as much space as it is in use because clusters are only written as they are needed. But raw format is a 1:1 copy of a disk, so this advantage disappears because unused clusters must be represented as a stream of zeroes.

Chapter 3

Background

3.1 Block Devices

3.1.1 Brief introduction to Linux device drivers

A device driver is a set of data structures and functions that allow the kernel to control a real or virtual device through a common interface. This is one of the pillars of Operating Systems design and the means of the kernel to access the diverse hardware available. In Linux, there are three kinds of devices depending on the characteristics of the underlying device.

- Character devices: access like a stream of bytes.
- Block devices: access data in chunks. Designed for slow, mass-storage media.
- Network devices: designed for network devices (as its name implies). This drivers do not follow the file paradigm and fall out of the scope of this paper.

Character and Block devices are represented on the file-system (usually under the /dev directory) as device files, available to user processes to interact with them. Device files are not identified by their names, but by a couple of numbers: major and minor. Major numbers indicate the type of device (a hard disk or a sound card) while the minor specifies multiple devices for the same major. For instance, on a computer with many hard disks each of them will have the corresponding device file. All of them will share the same major number since it is the same kind of device, but different minor numbers.

Device drivers in Linux are usually implemented as pluggable modules: a file containing the compiled driver code that can be loaded and unloaded at will. This makes a more efficient use of system memory because only the required drivers are loaded at a time. Also helps development by making it a lot easier and more convenient.

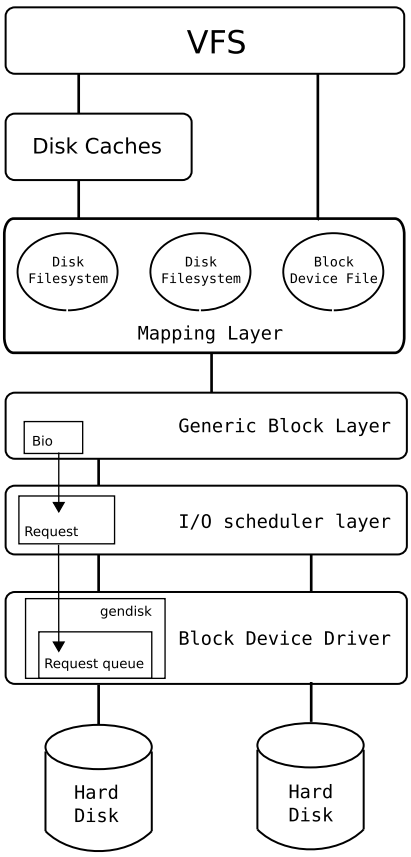


Figure 3.1: Block device operation

3.1.2 Block device access

Traditionally mass-storage media, such as magnetic drives, are slow. Aggressive buffering and careful access scheduling is required to enhance performance. Also, they are strongly related to file-systems. The generic block and VFS (Virtual File System) layers try to abstract these issues and provide a generic interface to ensure efficient access to this kind of media.

The VFS layer (Virtual File System) is a common interface for file-system operations. It imposes the common file model to all the file-systems available for Linux, implements operations shared by all of them (such as closing a file), and communicates with the disk cache to enhance performance. Most requests to block devices are generated here. Then, the Mapping layer is in charge to determine the physical location of the data and the associated file-system and physical blocks.

Table 3.1: Some system calls handled by the VFS

System call name	**Description**
`mount() umount() umount2()`	Mount/unmount filesystems

sysfs()	Get filesystem information
statfs() statfs64() ustat()	Get filesystem statistics
chroot() pivot_root()	Change root directory
chdir() fchdir() getcwd()	Manipulate current directory
mkdir() rmdir()	Create and destroy directories
getdents() readdir() link() unlink() rename()	Manipulate directory entries
readlink() symlink()	Manipulate soft links
chown()	Modify file owner
chmod() utime()	Modify file attributes
stat() access()	Read file status
open() close() creat() umask()	Open, close, and create files
dup() dup2() fcntl()	Manipulate file descriptors
select() poll()	Wait for events on a set of file descriptors
truncate()	Change file size
lseek()	Change file pointer
read() write() readv() writev()	Carry out file I/O operations
io_setup() io_submit() io_cancel() pread64() pwrite64()	Asynchronous I/O (allows multiple outstanding read and write requests) Seek file and access it
mmap() mmap2() munmap() madvise()	Handle file memory mapping
fsync() sync()	Synchronize file data
flock()	Manipulate file lock
setxattr() getxattr() listxattr()	Manipulate file extended attributes

Each I/O operation is represented by the bio structure. It contains all the data required by the generic block layer to perform an operation on a disk, such as the destination device, the type of operation (READ of WRITE), pointer to memory areas with the data to be read or written, index of the disk sectors to read (or write), etc. bios are created at the generic block layer.

Table 3.2: Fields of the bio structure

Type	**Field**	**Description**
sector_t	bi_sector	First sector on disk of block I/O operation
struct bio*	bi_next	Link to the next bio in the request queue
struct block_device*	bi_bdev	Pointer to block device descriptor
unsigned long	bi_flags	Bio status flags

unsigned long	bi_rw	I/O operation flags
unsigned short	bi_vcnt	Number of segments in the bio's bio_vec array
unsigned short	bi_idx	Current index in the bio's bio_vec array of segments
unsigned short	bi_phys_segments	Number of physical segments of the bio after merging
unsigned short	bi_hw_segments	Number of hardware segments after merging
unsigned int	bi_size	Bytes (yet) to be transferred
unsigned int	bi_hw_front_size	Used by the hardware segment merge algorithm
unsigned int	bi_hw_back_size	Used by the hardware segment merge algorithm
unsigned int	bi_max_vecs	Maximum allowed number of segments in the bio's bio_vec array
struct bio_vec*	bi_io_vec	Pointer to the bio's bio_vec array of segments
bio_end_io_t*	bi_end_io	Method invoked at the end of bio's I/O operation operation
atomic_t	bi_cnt	Reference counter for the bio
void*	bi_private	Pointer used by the generic block layer and the I/O completion method of the block device driver
bio_destructor_t*	bi_destructor	Destructor method (usually bio_destructor()) invoked when the bio is being freed

Table 3.3: Fields of the bio_vec structure

Type	Field	Description
struct page*	bv_page	Pointer to the page descriptor of the segment's page frame
unsigned int	bv_len	Length of the segment in bytes
unsigned int	bv_offset	Offset of the segment's data in the page frame

Once a bio structure has been created, the kernel invokes the generic_make_request() function, which is the main entry point of the block layer. This function validates the bio in question, gets the request queue associated with the block device, and finally calls the make_request_fn() method of the request queue.

This last function is the entry point of the I/O Scheduler. This kernel component works with request descriptors, which are basically collections of related bios (physically close to each other on the disk surface). Its main purpose is to optimize disk operations, by merging and ordering these request objects

on the request queue. There is a request queue per block device and as implied by its name, it stores requests to be processed by the device driver.

Table 3.4: Fields of the request descriptor

Type	Field	Description
struct list_head	queuelist	Pointers for request queue list
unsigned long	flags	Flags of the request
sector_t	sector	Number of the next sector to be transferred
unsigned long	nr_sectors	Number of sectors yet to be transferred in the whole request
unsigned int	current_nr_sectors	Number of sectors in the current segment of the current bio yet to be transferred
sector_t	hard_sector	Number of the next sector to be transferred
unsigned long	hard_nr_sectors	Number of sector yet to be transferred in the whole request (updated by the generic block layer)
unsigned int	hard_cur_sectors	Number of sectors in the current segment of the current bio yet to be transferred (updated by the generic block layer)
struct bio*	bio	First bio in the request that has not been completely transferred
struct bio*	biotail	Last bio in the request list
void*	elevator_private	Pointer to private data for the I/O scheduler
int	rq_status	Request status: essentially, either RQ_ACTIVE or RQ_INACTIVE
struct gendisk*	rq_disk	The descriptor of the disk referenced by the request
int	errors	Counter for the number of I/O errors that ocurred on the current transfer
unsigned long	start_time	Request's starting time (in jiffies)
unsigned short	nr_phys_segments	Number of physical segments of the request
unsigned short	nr_hw_segments	Number of hardware segments of the request
int	tag	Tag associated with the request (only for hardware devices supporting multiple outstanding data transfers)
char*	buffer	Pointer to the memory buffer of the current data transfer (NULL if the buffer is high memory)

int	ref_count	Reference counter for the request
request_queue_t*	q	Pointer to the descriptor of the request queue containing the request
struct request_list	rl	Pointer to request_list data structure
struct completion*	waiting	Completion for waiting for the end of the data transfers
void*	special	Pointer to data used when the request includes a "special" command to the hardware device
unsigned int	cmd_len	Length of the commands in the cmd field
unsigned char[]	cmd	Buffer containing the pre-built commands prepared by the request queue's prep_rq_fn method
unsigned int	data_len	Usually, the length of data in the buffer pointed to by the data field
void*	data	Pointer used by the device driver to keep track of the data to be transferred
unsigned int	sense_len	Length of buffer pointed to by the sense field (0 if the sense field is NULL)
void*	sense	Pointer to buffer used for output of sense commands
unsigned int	timeout	Request's time-out
struct request_pm_state*	pm	Pointer to a data structure used for power-management commands

Table 3.5: Fields of the request queue descriptor

Type	**Field**	**Description**
struct list_head	queue_head	List of pending requests
struct request*	last_merge	Pointer to descriptor of the request in the queue to be considered first for possible merging
elevator_t*	elevator	Pointer to the elevator object
struct request_list	rq	Data structure used for allocation of request descriptors
request_fn_proc*	request_fn	Method that implements the entry point of the strategy routine of the driver
merge_request_fn*	back_merge_fn	Method to check whether it is possible to merge a bio to the last request in the queue

merge_request_fn*	front_merge_fn	Method to check whether it is possible to merge a bio to the first request in the queue
merge_request_fn*	merge_requests_fn	Method to attempt merging two adjacent requests in the queue
merge_request_fn*	make_request_fn	Method invoked when a new request has to be inserted in the queue
prep_rq_fn*	prep_rq_fn	Method to build the commands to be sent to the hardware device to process this request
unlug_fn*	unplug_fn	Method to unplug the block device
merge_bvec_fn*	merge_bvec_fn	Method that returns the number of bytes that can be inserted into an existing bio when adding a new segment (usually undefined)
activity_fn*	activity_fn	Method invoked when a request is added to a queue (usually undefined)
issue_flush_fn*	issue_flush_fn	Method invoked when a request queue is flushed (the queue is emptied by processing all requests in a row)
struct timer_list	unplug_timer	Dynamic timer used to perform device plugging
int	unplug_thresh	If the number of pending requests in the queue exceeds this values, the device is immediately unplugged (default is 4)
unsigned long	unplug_delay	Time delay before device unplugging (default is 3 ms)
struct work_struct	unplug_work	Work queue used to unplug the device
struct backing_dev_info	backing_dev_info	Stores information about the I/O data flow traffic for the underlying hardware block device
void*	queuedata	Pointer to private data of the block device driver
void*	activity_data	Private data used by the activity_fn method
unsigned long	bounce_pfn	Page frame number above which buffer bouncing must be used
int	bounce_gfp	Memory allocation flags for bounce buffers
unsigned long	queue_flags	Set of flags describing the queue status
spinlock_t*	queue_lock	Pointer to request queue lock
struct kobject	kobj	Embedded kobject for the request queue
unsigned long	nr_requests	Maximum number of requests in the queue

unsigned int	nr_congestion_on	Queue is considered congested if the number of pending requests rises above this threshold
unsigned int	nr_congestion_off	Queue is considered not congested if the number of pending requests falls below this threshold
unsigned int	nr_batching	Maximum number (usually 32) of pending requests that can be submitted (even when the queue is full) by a special "batcher" process
unsigned short	max_sectors	Maximum number of sectors handled by a single request (tunable)
unsigned short	max_hw_sectors	Maximum number of sectors handled by a single request (hardware constraint)
unsigned short	max_phys_segments	Maximum number of physical segments handled by a single request
unsigned short	max_hw_segments	Maximum number of hardware segments handled by a single request (the maximum number of distinct memory areas in a scatter-gather DMA opeartion)
unsigned short	hardsect_size	Size in bytes of a sector
unsigned int	max_segment_size	Maximum size of a physical segment (in bytes)
unsigned long	seg_boundary_mask	Memory boundary mask for segments merging
unsigned int	dma_alignment	Alignment bitmap for initial address and length of DMA buffers (default 511)
struct blk_queue_tag*	queue_tags	Bitmap of free/busy tags (used for tagged requests)
atomic_t	refcnt	Reference counter of the queue
unsigned int	in_flight	Number of pending requests in the queue
unsigned int	sg_timeout	User-defined command time-out (used only by SCSI generic devices)
unsigned int	sg_reserved_size	Essentially unused
struct list_head	drain_list	Head of a list of requests temporarily delayed until the I/O scheduler is dynamically replaced

Upon receiving a bio, it tries to merge it with an existing request or allocates a new one for it. Another functionality of the I/O Scheduler is to plug or unplug (activate or deactivate, respectively) the request queue. A plugged

request queue can receive more requests, but they will not be processed until the queue is unplugged again. The reasoning behind this intended delay is to enhance disk performance. Magnetic storage media is very slow performing seeking operations. By processing multiple requests together in a sequential order (in a linear way from the inner track to the outer one or vice versa), the number of disk head movements is greatly reduced, improving performance. There are as many ways to order requests as factors to take into account. Thus, there are various I/O scheduler algorithms, also known as `elevators`. Just as process schedulers they are complex and their constants and heuristics are the result of extensive benchmarking and testing.

At the bottom of the block device operation lies the block device driver and the gendisk structure which represent disks or partitions. A common block device is represented by a block device descriptor, defined on a module. Upon initialization, this module registers a unique identifier for each device (the major number and minor numbers), an interrupt handler, a set of operations, one or more gendisk structures and a request queue for each of them. Together with the request queue is defined a strategy routine (the `request_fn()` method), which is invoked by the I/O Scheduler to start the processing of bio structures. This routine is in charge of performing the actual read and write operations on the device. A common approach for a strategy routine would be:

- obtain the first bio on the request queue
- set up the device's DMA controller to perform the operation described in the bio
- return, while the hardware takes care of the transaction in the background

Upon completion, the DMA controller will issue an interrupt and the interrupt handler will be activated automatically. It should check if there are requests pending on the request queue and call the strategy routine again to process them. This loop will continue until the request queue becomes empty.

Table 3.6: Fields of the block device descriptor

Type	**Field**	**Description**
`dev_t`	`bd_dev`	Major and minor number of the block device
`struct inode*`	`bd_inode`	Pointer to the inode of the file associated with the block device in the bdev filesystem
`int`	`bd_openers`	Counter of how many times the block device has been opened
`struct semaphore`	`bd_sem`	Semaphore protecting the opening and closing of the block device
`struct semaphore`	`bd_mount_sem`	Semaphore used to forbid new mounts on the block device
`struct list_head`	`bd_inodes`	Head of a list of inodes of opened block device files for this block device

void*	bd_holder	Current holder of block device descriptor
int	bd_holders	Counter for multiple settings of the bd_holder field
struct block_device*	bd_contains	If block device is a partition, it points to the block device descriptor of the whole disk; otherwise, it points to this block device descriptor
unsigned	bd_block_size	Block size
struct hd_struct*	bd_part	Pointer to partition descriptor (NULL if this block device is not a partition)
unsigned	bd_part_count	Counter of how many times partitions included in this block device have been opened
int	bd_invalidated	Flag set when the partition table on this block device needs to be read
struct gendisk*	bd_disk	Pointer to gendisk structure of the disk underlying this block device
sturct list_head*	bd_list	Pointers for block device descriptor list
struct backing_dev_info*	bd_inode_backing_dev_info	Pointer to a specialized backing_dev_info descriptor for this block device (usually NULL)
unsigned long	bd_private	Pointer to private data of the block device holder

Table 3.7: Fields of the gendisk object

Type	Field	Description
int	major	Major number of the disk
int	first_minor	First minor number associated with the disk
int	minors	Range of minor numbers associated with the disk
char[32]	disk_name	Conventional name of the disk (usually, the canonical name of the corresponding device file)
struct hd_struct**	part	Array of partition descriptors for the disk
struct block_device_operations*	fops	Pointer to a table of block device methods
struct request_queue*	queue	Pointer to the request queue of the disk
void*	private_data	Private data of the block device driver

sector_t	capacity	Size of the storage area of the disk (in number of sectors)
int	flags	Flags describing the kind of disk
char[64]	devfs_name	Device filename in the (now deprecated) devfs special filesystem
int	number	No longer used
struct device*	driverfs_dev	Pointer to the device object of the disk's hardware device
struct kobject	kobj	Embedded kobject
struct timer_rand_state*	random	Pointer to a data structure that records the timing of the disk's interrupts; used by the kernel built-in random number generator
int	policy	Set to 1 if the disk is read-only, 0 otherwise
atomic_t	sync_io	Counter of sectors written to disk, used only for RAID
unsigned long	stamp	Timestamp used to determine disk queue usage statistics
unsigned long	stamp_idle	Same as above
int	in_flight	Number of ongoing I/O operations
struct disk_stats*	dkstats	Statistics about per-CPU disk usage

Table 3.8: List of disk device methods (struct block_device_operations)

Method	Triggers
open	Opening the block device file
release	Closing the last reference to a block device file
ioctl	Issuing an ioctl() system call on the block device file (uses the big kernel lock)
compat_ioctl	Issuing an ioctl() system call on the block device file (does not use the big kernel lock)
media_changed	Checking whether the removable media has been changed (e.g., floppy disk)
revalidate_disk	Checking whether the block device holds valid data

3.2 The Loop Module

The loop module is a block device driver that allows to access a file as if it was a block device. The module creates a certain number of block devices (/dev/loop0, /dev/loop1...) which can be configured later on through ioctl()

syscalls to associate any of them with a file. A common use of this module is to mount and access the contents of a disk image (a sector-by-sector copy of a real disk into a file) as if it was the original device.

This module also supports data transformation by allowing to register "transfer functions" that will be applied to every bio processed by any of its block devices. This permits to encrypt and decrypt data on-the-fly. This functionality is employed by the cryptoloop module, having loop module as a dependency, to access and mount encrypted disk images with a variety of algorithms. To convert a file into a block device, the loop module follows the model explained on the previous section, but with some variations. Since there is no hardware, there is no DMA controller and no interrupt handler to set up. Instead, it defines a kernel thread (a kernel function which can run on the background), that converts bio structures into common read and write operations over the file associated with the loop device. The `make_request_fn()` method of the loop device's request queue is defined to get the bio passed as a parameter and enqueue it into a bio_list structure (basically a linked list of bios). The kernel thread periodically processes this list and commits the bios to disk. This way, the I/O Scheduler and `elevators` are skipped. After all, since loop is not a real device it no sense to perform I/O scheduling. Once bios have been converted to VFS operations, they will be converted again into bios and eventually scheduled on the underlying device's request queue.

Table 3.9: Fields of `struct loop_device`

Type	**Field**	**Description**
`int`	`lo_number`	ID of the loop device
`int`	`lo_refcnt`	Number of times the device has been opened but not closed (yet)
`loff_t`	`lo_offset`	Offset of the associated file where the loop device should begin
`loff_t`	`lo_sizelimit`	Maximum size of the loop device
`int`	`lo_flags`	Flags
`int*`	`transfer`	Transfer function to apply to incoming bios
`char[]`	`lo_file_name`	Name of the loop device
`char[]`	`lo_crypt_name`	Name of the transfer function to apply
`char[]`	`lo_encrypt-_key`	Encryption key to apply (required for encryption transfer functions)
`int`	`lo_encrypt-_key_size`	Size of the encryption key (32 characters max)
`struct loop-_func_table*`	`lo-_encryption`	List of override functions to apply to this loop device.
`__u32[]`	`lo_init`	Unused
`uid_t`	`lo_key_owner`	User ID of the process that initialized the key
`int*`	`ioctl`	`ioctl()` function to apply to this loop device (if the parameter does not match with any of the loop's expected values)

struct file*	lo_backing_file	File associated with this loop device
struct block_device*	lo_device	Block device created by this loop instance
unsigned	lo_blocksize	Block size of the lo_backing_file underlying block device
void*	key_data	Unused (used by cryptoloop)
gfp_t	old_gfp_mask	Original gfp (get free pages) mask of the backing file
spinlock_t	lo_lock	Spin lock that protects the lo_bio_list and lo_state fields
struct bio_list	lo_bio_list	List of bios received and pending to be processed
int	lo_state	Indicates the state of the loop device.
struct mutex	lo_ctl_mutex	Mutual exclusion semaphore protecting the entire loop scructure
struct task_struct*	lo_thread	Kernel thread for this loop device
wait_queue_head_t	lo_event	Wait queue for this loop device and lo_thread
struct request_queue	lo_queue	Request queue associated with this loop device
struct gendisk	lo_disk	gendisk for this loop device
struct list_head	lo_list	Linked list that connects all loop devices

Table 3.10: Fields of struct loop_func_table

Type	Field	Description
int	number	Identifier of this transfer function structure
int*	transfer	Transfer function to apply to every data transaction
int*	init	Initialization function
int*	release	Termination function
int*	ioctl	ioctl function extension
struct module*	owner	Pointer to the module that owns this structure

When the loop module is loaded, it allocates a certain number of loop structures (8 by default, although it can be set as a module parameter) and initializes some of its fields, such as the request queue, gendisk and mutex structs. Each loop is identified by a number, from 0 to 7 in the default case. All loop structures are linked together by the lo_list field. This is done to ease some

tasks by the module that affect all loop structs (basically initialization and termination).

Now the device is ready to receive ioctl syscalls to continue with the set up. Each ioctl argument accepted is associated with a function of the module's code. There is a userspace tool called losetup to issue these calls easily for end users. This application is included in the util-linux-ng, a set of tools to perform task highly related with the kernel, such as mount or mkswap and others to configure certain devices or modules such as loop. Most of these ioctls revolve around exchanging information with the userspace process (losetup in this case) with loop_get_status() and loop_set_status() functions and their variations. They are used by losetup to ascertain the current status of a loop device and setting a few parameters such as the encryption key and transfer function, offset of the backing file where the loop device should begin and its length. These last two options allow to create a block device from a segment instead of a entire file.

The most important call is loop_set_fd(), which finally associates the loop device with an existing file and creates a kernel thread to process bios. From this point on the device is ready to act as a block device on any incoming request. Depending on the type of request and the features available in the backing file's filesystem and if a transfer function is set, there are a few functions available to convert read and write bios into corresponding VFS operations.

do_lo_send_aops() Asynchronous write operation by calling pagecache_write_begin() and pagecache_write_end().

do_lo_send_write() Synchronous write operation by invoking the write() function associated with the backing file. Only applied if the backing file's filesystem does not support asynchronous writes.

do_lo_send_direct_write() Like do_lo_send_write() but does not apply transfer function. Used when there is no transfer function defined.

do_lo_receive() Read operation, performed with the kernel's splice syscalls. This system allows to copy data from file to file without having to copy data into kernel-space and thus improves performance.

The function loop_clr_fd() does to opposite of loop_set_fd() and releases a file from a loop device, returning the latter to its original state. A new ioctl processing function can be defined through the transfer function system or by creating a new module that depends on loop and sets the lo_ioctl field. The function pointed by this field will be executed if loop module's default ioctl function receives an unknown parameter.

The loop device's current status is stored in the field lo_state, which can take three values:

Lo_unbound The default state, means that this loop device is not associated with a file.

Lo_bound The opposite of the previous state - this loop instance is associated with a file.

Lo_rundown The loop device is in the process of disassociating with a file. As the process completes, it will reach the Lo_unbound state again. This is done to stop allowing any more bios to enter the bio_list, but letting the kernel thread to finish processing all pending bios currently on the list.

Transfer functions are applied on every data operation, be it read or write, on the device. The loop module provides a couple of exported functions (that is, they can be invoked from other modules) to add and remove these transfer functions: loop_register_transfer() and loop_unregister_transfer() respectively. There are few extra functions besides the transfer function itself. They are all grouped by the loop_func_table structure. A pointer to this structure is the parameter passed to loop_register_transfer(). The loop_func_table.init() function is run from loop_set_status() in case there is a transfer function associated to the loop device and its point is to perform any initialization necessary by the transfer function to do its job properly. loop_func_table.release() does the opposite and cleans up after the loop is disassociated with its backing file. Finally loop_func_table.ioctl() is set to lo_ioctl to extend the loop module's ioctl() function as explained previously. The cryptoloop module uses this "transfer function" facility to extend loop and add a variety of encryption algorithms.

3.3 Introduction to QEMU and KVM

3.3.1 QEMU - Quick Emulator

An emulator is a program that provides an emulation of the functionality of a certain system or CPU architecture on a different system. QEMU is a popular processor emulator [1]. It allows to run software compiled for other architectures on a binary incompatible system by translating blocks of binary code. This method makes porting the emulator to make it run on many systems and adding support for new architectures easier, and at the same time achieving a reasonable performance when running non-native applications [3].

Currently QEMU supports many architectures with varying levels of support. It also emulates other pieces of hardware such as network and video cards and even a BIOS to allow emulating an entire computer, which allows to run other Operating Systems as if they were a common process on a system, or otherwise said - as guests.

This software provides a couple of levels of emulation support, called User Mode Emulation and Full System Emulation. The former executes a single process specified by the user as if it were on another architecture (or even running a program which was compiled for another incompatible system) while the latter emulates a whole computer system, with its own devices and BIOS.

To achieve Full System Emulation, QEMU needs to provide permanent storage - a feature taken for granted on basically any modern computer system. Hard disks are represented by a file on the host system as a disk image. Information can be stored on this file in many ways, so there are many image formats, from the naive raw image (a sequential copy of every data on a disk) to more advanced solutions such as QCOW2 file format. QEMU support multiple

formats for reading and writing disk images, but its official and better supported format is QCOW2.

3.3.2 KVM - Kernel-based Virtual Machine

Virtualization is a technique that abstracts the system's hardware to applications. In this virtual machine environment, software runs in a complete simulation of the underlying hardware. This way, it is possible to run applications or even complete operating systems as processes. The application that controls simulation is called virtualizer or hypervisor. Main differences with emulation are that it doesn't simulate a hardware different from your own and that its focus is performance.

KVM is a character device driver for Linux that gives access to the virtualization extensions found on modern x86 CPUs [2]. Usually represented by the device file `/dev/kvm`, it currently supports Intel and AMD models. The KVM developers also provide a client application that employs this character device (and consequently, the kernel) as a hypervisor. This client application is a modified version of QEMU that uses the KVM driver instead of binary translation. This client application is also called kvm, which causes certain confusion.

The original patches to add hardware-assisted virtualization to QEMU have been added officially to the QEMU project, so now any recent version of QEMU can employ the kvm driver. While KVM is a relatively young project [10] it is very active and continues to add new functionality to QEMU on their own client, that eventually reaches upstream. Note that the QEMU developers add other features of they own so both projects synchronize their code base from time to time. An example of features added by the KVM project is the VirtIO framework, which enhances performance for I/O operations by providing special network and disk controller kernel modules.

Because the KVM project is highly tied to QEMU, it employs the same solution for permanent storage - files as hard disks. And just like QEMU, the default format for the files is QCOW2.

3.4 QCOW2 File Format

QCOW2 is the file format designed by the QEMU developers specifically for their project [4] as a means to represent permanent storage, and it is the second revision of the original QCOW format (also known as QEMU v1). QCOW2 files are created with the `qemu-img` tool, included in QEMU and KVM packages. While QEMU supports many other formats for disk files, this one has the most features and support.

Some of these features are:

- Copy-on-write support, where the QCOW2 file only represents the changes made to another disk image.
- Snapshot support, where the image can contain multiple snapshots of the

image's history.

- Data compression with zlib.
- Data encryption with AES.

A QCOW2 file is divided in clusters of a fixed size. This size must be a power of two and multiple of 512, between 512 and 65536. It is specified at file creation time. Every data or control structure has the size of one or more clusters. Also, all metadata stored in the file (such as the header or the addressing tables) is in big endian format. A typical layout for a QCOW2 file is:

- The header.
- The L1 table.
- The refcount table, again boundary aligned.
- One or more refcount blocks.
- Snapshot headers, the first boundary aligned and the following headers aligned on 8 byte boundaries.
- L2 tables, each one occupying a single cluster.
- Data clusters.

3.4.1 Addressing

In order to find a cluster corresponding to a given address, it is necessary to traverse two tables, in a similar way to the paging mechanisms of modern Operating Systems. The first table is called L1 table. It is an array of 8-byte file offsets. The addresses contained in the L1 table (in big endian) point to L2 tables. While there is only one L1 table, there can be many L2 tables, and each of them take up exactly one cluster. L2 tables contain 8-byte file offsets in big endian format too, but they point to data clusters. That is, clusters than contain actual disk sectors.

To reach a certain data offset in a QCOW2 file, it is necessary to divide the offset in three sections: `l1_table_index`, `l2_table_index` and `cluster_offset`. The size of each field depends on the QCOW2 image size and cluster size. The number of bits of `cluster_offset` is the number of bits necessary to address a cluster, that is the binary logarithm of the cluster size. From now on, we will call this number `cluster_bits`. The number of bits of the `l2_table_index` section is cluster_bits - 3, which we will call `l2_bits`. Because a L2 table is an array of 8 byte elements, this number is a index of such table. Since a L2 table has the same size as a cluster, to index it as a table of 8 bytes offsets we need exactly 3 bits less than `cluster_bits`. Finally, the remaining top bits are `l1_table_index`. The size of the L1 table in bytes is determined at creation time of the QCOW2 file and follows the formula:

$$l1\_size = round\_up\left(\frac{disk\_size}{2^{(cluster\_bits+l2\_bits)}} \times 8\right)$$

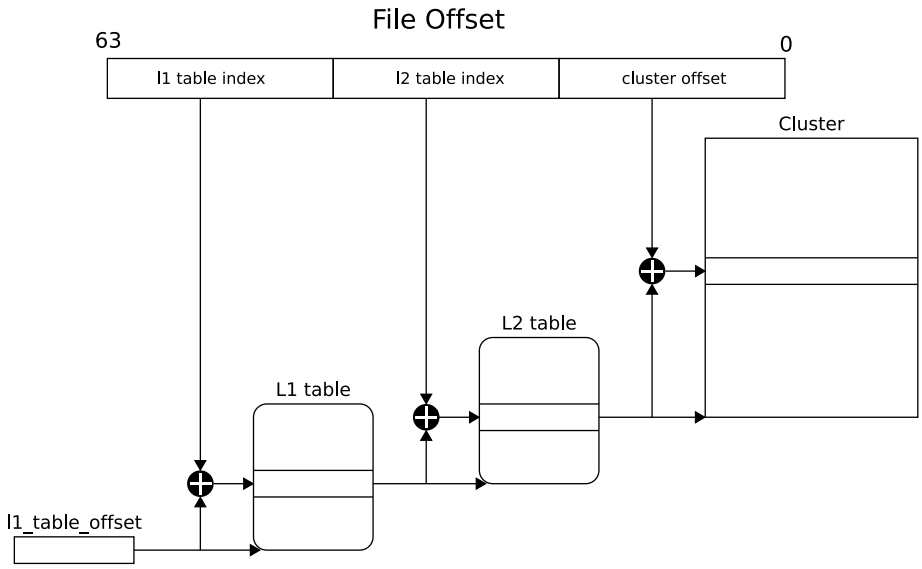


Figure 3.2: QCOW2 cluster addressing

Where "round up" means rounding up to a multiple of `cluster_size`.

Now, the actual algorithm to reach a certain disk offset inside a QCOW2 file follows these steps:

1. Read the header to find the location of the L1 table.
2. Use the first part of the offset (`l1_table_index`) to index the L1 table as an array of 64 bit entries.
3. Obtain the L2 table address using the offset obtained from the L1 table.
4. Use the second section of the offset (`l2_table_index`) to index the L2 table as an array of 64 bit entries.
5. Obtain the cluster address using the offset in the L2 table.
6. Use the last section of the offset (`cluster_offset`) as an offset within the cluster itself.

Finally, all QCOW2 offsets stored on L1/L2 tables have a couple of bits with a special meaning. The highest order bit is set if there is more than one copy of the cluster pointed. Otherwise said, the bit is set if there is at least one snapshot with a different version of the cluster. The second highest order bit is set in case that the cluster is compressed.

3.4.2 Snapshots

A snapshot is the state of a system at a particular point in time. In the case of QCOW2, a snapshot is the state of the disk (and all the data contained in it).

So a snapshot could be considered like a previous version of the data stored in the disk, frozen in time.

To support snapshots efficiently, QCOW2 files only store the clusters that differ from the current state. That is, clusters that are equal on the snapshot and in the current version are only stored once, while the rest are stored as many times as necessary (as many as snapshots are defined on the disk). Which version of a cluster will be accessed depends on if we are working with the current version of the disk or a snapshot.

To keep track of the versions of each cluster, QCOW2 files have a couple of structures: the refcount table (for "reference count") and refcount blocks. The former is an array of 8-byte offsets that point to refcount blocks. The latter is an array of 2-byte counters, which indicate the number of copies of the corresponding cluster. As with every other structure in QCOW2, they have a size of one cluster or a multiple, in the case of the refcount table.

While refcount blocks can be found on any position on the file, the counters they contain are stored in sequential order. That is, the first element of the first refcount block indicates the number of copies of the first cluster of the QCOW2 file, which is the header. The second element of the same block stores the number of references of the second cluster of file, and so on. When the QCOW2 file grows to fill this refcount block, a new one will be appended to the file and its offset will the added refcount table.

3.4.3 Header

As every other structure in QCOW2 files, the header takes up a whole cluster too, even though its size is pretty small in comparison to the average cluster size. The header is stored at the beginning of the file in big endian format:

- The first 4 bytes contain the characters 'Q', 'F', 'I' followed by 0xfb. These four bytes are a "magic number", chosen arbitrarily by the author of QCOW2 to identify this kind of files.

- The next 4 bytes contain the format version used by the file. Currently, there has been two versions of the format, version 1 and version 2. We are discussing the latter here.

- The `backing_file_offset` field gives the offset from the beginning of the file to a string containing the path to a file; `backing_file_size` gives the length of this string, which is not a NULL-terminated. If this image is a copy-on-write image, then this will be the path to the original file. More on that below.

- The `cluster_bits` field describes how to map an image offset address to a location within the file; it determines the number of lower bits of the offset address are used as an index within a cluster. Since L2 tables occupy a single cluster and contain 8 byte entries, the next most significant `cluster_bits`, minus three bits, are used as an index into the L2 table.

```
typedef struct QCowHeader {
     uint32_t magic;
     uint32_t version;

     uint64_t backing_file_offset;
     uint32_t backing_file_size;

     uint32_t cluster_bits;
     uint64_t size; /* in bytes */
     uint32_t crypt_method;

     uint32_t l1_size;
     uint64_t l1_table_offset;

     uint64_t refcount_table_offset;
     uint32_t refcount_table_clusters;

     uint32_t nb_snapshots;
     uint64_t snapshots_offset;
 } QCowHeader;
```

Figure 3.3: QCOW2 Header

- The next 8 bytes contain the size, in bytes, of the block device represented by the image.
- The `crypt_method` field is 0 if no encryption has been used, and 1 if AES encryption has been used.
- The `l1_size` field gives the number of 8 byte entries available in the L1 table and `l1_table_offset` gives the offset within the file of the start of the table.
- Similarly, `refcount_table_offset` gives the offset to the start of the refcount table, but `refcount_table_clusters` describes the size of the refcount table.
- `nb_snapshots` gives the number of snapshots contained in the image and `snapshots_offset` gives the offset of the QCowSnapshotHeader headers, one for each snapshot.

Chapter 4

Implementation

Basically, the qloop module is based on Linux's loop module but with a few modifications to support QCOW2 disks, hence it's name (QCOW2-loop was too long). As explained previously, the loop module converts a file into a block device, transforming the block device's read and write operations into file-based operations. Also, it supports data transformation functions which are applied on-the-fly on every access operations.

Unfortunately, these facilities are not enough to manipulate QCOW2 files. This file format is not a sequential representation of a disk, so an address translation mechanism is required. In addition, QCOW2 files have "holes": every cluster of the disk which has not been written yet does not have a representation in the QCOW2 file. And there are a few control structures or metadata that have to be maintained, such as L1/L2 tables, reference counting... While snapshots are not supported in the current version of qloop, reference counting tables are part of the specification, and exist in any QCOW2 file even if snapshots are not in use. So they must be maintained to keep compatibility with the QCOW2 specification and thus with the rest of applications that employ this file format.

qloop module is divided in two files: qloop.h and qloop.c. These two files are mostly copies of the loop module source (loop.h and loop.c, which are part of the Linux kernel). All the changes made to these two files to create qloop are explained in detail in this section. Since loop module does not provide all the facilities needed, it was not possible to simply create a new module that depended on loop (such as the cryptoloop module). Also, their implementation would made loop incompatible with previous versions. So I chose to copy the two kernel files and develop qloop as if it were a new unrelated module, even if 90% of the code is identical to the original loop.

4.1 Structures

It was not necessary to perform big changes on the original loop structures. Just adding the QCowHeader structure (as shown on the previous chapter) and

a read-write semaphore to the `loop_device` structure was enough. QCowHeader is necessary to store all the essential information about the file, while the purpose of the semaphore is to protect the disk when updating control structures. Employing a read-write semaphore is more efficient since it will only block access while a write operation is being performed.

As a side note, it was necessary to define a new major number for the qloop device files created by this module. Otherwise, it would not have been possible to load both qloop and loop modules at the same time due to collision on device file creation. The major number selected to identify qloop was selected from a range allocated for experimental use as stated on the kernel documentation (Documentation/devices.txt).

4.2 Functions

These are the new functions added to loop module to support QCOW2 files:

- `static int read_qcow2(struct file *f, u8 *buffer, int size, loff_t *offset)` This function is a simple wrapper to the read function associated to file *f*. It takes care of setting the right environment. The main reason to create this is to avoid code repetition, since simple reads on the QCOW2 file are quite common.
- `static int write_qcow2(struct file *f, u8 *buffer, int size, loff_t *offset)` Similar to the previous function, but calls write instead.
- `static loff_t get_qcow2_offset(struct file *f, uint64_t *address)` Reads an offset address from a QCOW2 file, generally from an L1/L2 or refcount table. It calls `read_qcow2()` and performs a couple of operations on the obtained value, such as endianness conversion.
- `static int set_qcow2_offset(struct file *f, uint64_t address, uint64_t value)` Similar to `get_qcow2_offset()`, this function writes a offset address to a QCOW2 file in the expected format.
- `static int append_cluster(struct QCowHeader *qp, struct file *f, uint64_t *cluster_offset)` Append a string of zeros to the end of the file. The size of the string is exactly the size of a cluster. The zeros are added by calling `write_qcow2()` with `PAGE_SIZE` as size parameter. This operation is enclosed in a loop and will be repeated as many times as necessary to reach the size of a cluster.
- `static int set_refcount_offset(struct file *f, uint64_t *address, uint16_t value)` Similar to `set_qcow2_offset()`, but simplifies the process of writing a reference count (2-byte number) in a refcount block.
- `static int get_refcount_offset(struct QCowHeader *qp, struct file *f, uint64_t *address)` Finds the refcount value for a cluster and returns its offset in the file. The address parameter is used to read the address of the cluster from which it is desired to get a refcount. Then

it parses the refcount table to obtain the necessary refcount block's address, from which the final offset is obtained and returned on the address parameter.

- `static int get_qcow2_address(struct QCowHeader *qp, struct file *f, uint64_t *address, int type)` Converts a disk address into the equivalent position in a QCOW2 file. This function performs the address translation by parsing the L1 and L2 tables. The *type* field specifies if the intended operation on *address* is read or write. The difference lies in what to do if the final offset has not ever been accessed. If the operation is write, a new cluster is appended to the file and its offset added to the relevant L2 table. The final offset is returned on the address parameter.
- `static int set_qcowheader(struct loop_device *lo, struct file *file)` Reads the file's header, converts its values to the right endianness and initializes all the values of the QCowHeader structure in the `loop_device` object.

4.3 Work flow

Apart from adding a few structures and functions, there are other changes to the loop module code. Some are small and self-explained such as:

- Initializing the new structures, such as the read-write semaphore or the QCowHeader. The latter has to be initialized at `loop_set_fd()` since at that point is where the association between a QCOW2 file and a qloop instance takes place.
- Obtaining the drive size from the value stored on the QCOW2 header, instead of the backing file size.

On the other hand, other changes are deeper. They require further explanation and are easier to understand after learning the entire process performed by qloop.

The loop kernel thread starts processing the bio structures by calling `do_bio_filebacked()`. This function checks whether the bio belongs to a read or write operation and calls the right function to process it: `lo_receive()` for reads and `lo_send()` for writes.

The function `lo_receive()` will call, for each bvec defined in the bio, the function `do_lo_receive()`. The changes for for qloop start here. This function will first call `get_qcow2_address()` to translate the read address intended by the bio into the right offset in the QCOW2 file.

This operation is protected by the read-write semaphore. Since `get_qcow2_address()` parses the L1/L2 tables, a change in these structures by a write operation while the function is still working could lead to obtaining a wrong value and eventually to data corruption. By using a read-write semaphore we enhance the performance of concurrent operations, since multiple read commands can be run simultaneously without problems. Unfortunately, a write operation

will stop any other operation (be it read or write) from being processed until `get_qcow2_address()` returns to avoid data inconsistency.

If the translated file offset exists, the read operation continues as in the original loop by invoking the splice functionality [11] to copy data between buffers (from the file's contents to the output buffer that will be returned to user-space). In case the file offset does not exist, that means that the corresponding disk cluster has never been written before and thus is empty. By filling the output buffer with zeros we simulate a read operation from an empty cluster.

All these operations performed in the bvec structure are enclosed in a loop. The length of the data to read must be checked to ensure that we don't cross a cluster boundary. If the bvec operation is big enough or has a certain offset, it could span more than one cluster. In such a case, since a QCOW2 file is not sequential, it would be necessary to repeat the address translation for the data beyond the cluster border. This operations will be repeated as many times as necessary to complete the read operation. No assumptions have been made on bvec or page size.

If the bio to process intends to write data to disk, `do_bio_filebacked()` will call `lo_send()`. Originally, this function selected the most suitable write operation, depending on factors such as support of asynchronous operation being enabled, or the need to apply a transformation function on the data to transfer. qloop does not use any kind of transformation functions, and asynchronous operation is disabled, so `lo_send()` basically just loops over the bvec structures defined in the bio and passing them to `do_lo_send_direct_write()`.

The reason to disable asynchronous writing lies in the fact that metadata changes in qloop are performed by invoking the synchronous write method on the backing file. It's not possible to know if metadata modification will be necessary until the write operation is in course. That is, until an address translation is performed on the bio's destination address and verified if the corresponding cluster in the QCOW2 file exists or not. Trying to predict this would require parsing all the bvec structs once before starting the actual data copy, which will process them again. Such implementation would be a lot less efficient. Also, synchronous and asynchronous operations cannot be mixed since both try to acquire the semaphore allocated in the file's inode structure, which would lead to a deadlock.

qloop write is performed by `do_lo_send_direct_write()`. This function, in a similar way to `do_lo_receive()`, calls `get_qcow2_address()` and copies data to the destination address, always taking into account that the length of the data does not cross a cluster boundary. Again, in such a case the function must obtain the QCOW2 file address and resume copying at the new location. The main difference here is that `get_qcow2_address()` is called in write mode (the `type` parameter is set to 1), which enforces it to create and append new clusters to the file to make it grow, be it data clusters or metadata (L2 tables and refcount blocks) and update previously existing values (L1 table and refcount table). As explained before, the invocation of `get_qcow2_address()` is protected by the qloop instance's read-write semaphore.

Chapter 5

Performance Tests

Performance tests have been carried out by running `fio`, a benchmark for IO devices, on a laptop with this configuration:

CPU Intel Core2Duo P8400 @ 2.26GHz

Memory 4GB DDR2

Storage 500GB SATA2 5400rpm Hard Disk

Operating System Debian 6.0 (testing branch)

Kernel 2.6.32

Architecture x86_64

fio 1.38

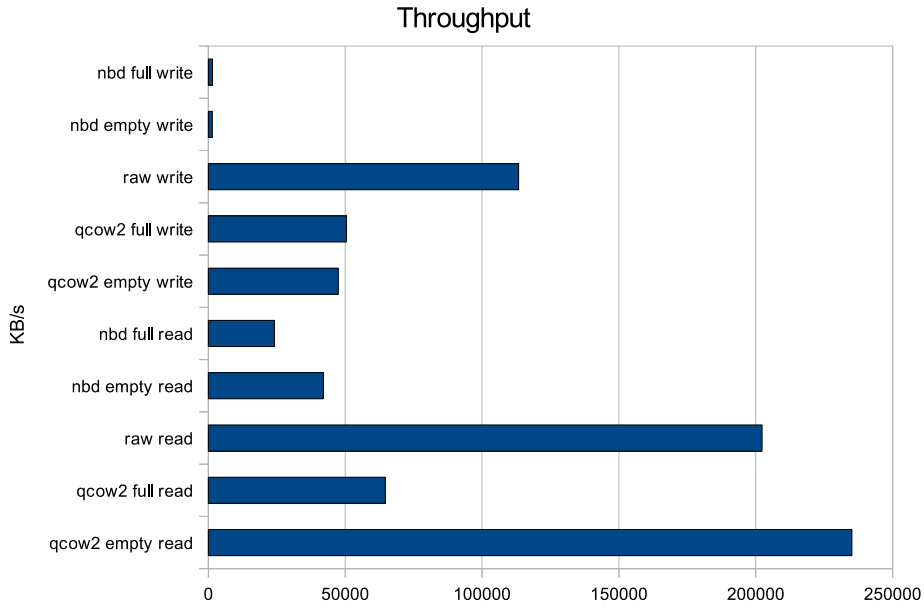
The advantage of using fio over other benchmark applications is that it can work with devices directly, instead on files and directories like most of its alternatives. While a test with files could have been possible, it would have added the overhead of a file-system to the results. Also, it introduces the problem of choosing a file-system to perform the tests with, or even repeating them with various file-systems to check the difference. fio allows me to skip all these troubles and work directly with the device itself.

The tests show the performance of qloop module with a QCOW2 file vs loop module with a raw file (a disk image). To speed up test duration, the size of the disk has been set to 5GB. All test have been performed 10 times and their results have been averaged.

A secondary computer was necessary to perform write performance tests with NBD to ensure that there would be no deadlocks. This second host has a similar configuration to the one described earlier and both are connected by a Gigabit Ethernet network.

I planned to include one of qloop's alternatives on the tests: libguestfs, but it was not possible. libguestfs's interface is too restricted and does not allow raw access to the contents of the file, necessary for fio.

Figure 5.1: Performance results



The tests performed try to show how qloop behaves under different situations. *Empty QCOW2 file* means that the test was done with a new QCOW2 file that has never been accessed. `qemu-img`, the program that creates these files, sets up the header, L1 table, refcount table and a refcount block on new files. This means that nearly every write operation will cause a new cluster to be appended to the file and surely a modification of the tables will take place. On the other hard, a *Full QCOW2 file* has all clusters already allocated and all the metadata structures are complete, so while address translation is still necessary, it will not be required to update them anymore. On read operations, an empty file will always perform a lot faster on qloop since as soon as the module verifies that the destination cluster does not exist, it will just return zeros instead of reading any byte from disk. This notion of full and empty files do not apply to loop and raw files because images always have the same size, no matter the contents ("empty" sectors are long strings of zeros). So reading or writing are not affected by these circumstances.

All the tests consist on performing many sequential read or write operations of 4KB each until the entire disk has been processed. The results show us that qloop is 50% slower than loop on most situations, but still achieves 50 MB/s, which is a lot more than its alternatives and a respectable speed. Also, the numbers tell us that qloop performance is not much affected by the file being full or empty. Otherwise said, appending clusters or updating metadata structures are not expensive tasks. So, the only reason why qloop is 50% slower than loop is because of address translation. If any attempt to optimize qloop is to be made, that is the part of the process where the effort must be put.

NBD's results are very poor. The fastest write throughput I could obtain was 1.5MB/s. Facing this results, I cannot help but wonder how could the QEMU developers consider this as a valid solution for accessing QCOW2 files, even if in a sporadic fashion.

Chapter 6

Time and cost analysis

This section describes the resources spent to develop qloop.

6.1 Time

Time was without a doubt the most needed resource. Most of this time was employed to learn the many technologies involved, such as: the QCOW2 specification, reading its implementation on KVM and QEMU source code, mastering the C language, find out how to develop a kernel module, understand the loop module's inner workings and specially the kernel's IO subsystem. This fact is reflected on the project's paper: the longest part is the chapter dedicated to explain the technologies behind qloop.

qloop's development spans for little more than a year, while my original expectations were half that time. A period so large just to develop a few lines of code may appear as very unproductive at first sight, but I should note that qloop required deep understanding of very complicated environments, such as the Linux IO subsystem or advanced knowledge of C programming, and learning does take a long time. Also, it should be taken into account that I was no expert on any of the technologies involved and had to learn while attending a full-time job. Having much more hours per day to dedicate to the project or previous experience on kernel development would have certainly cut the time spent to a fraction. The project can be divided into 3 types of tasks: learning, development and documentation. Learning consists on researching the software involved, basically reading books, documentation and source code. Also includes the development of small applications to help me grasp the understanding on some subject. Development tasks consist on writing code and testing it. Finally, documentation tasks are committed to write the project's documentation, such as typing the initial report and this paper or prepare the project's defense.

Figure 6.1: Tasks performed during 2009

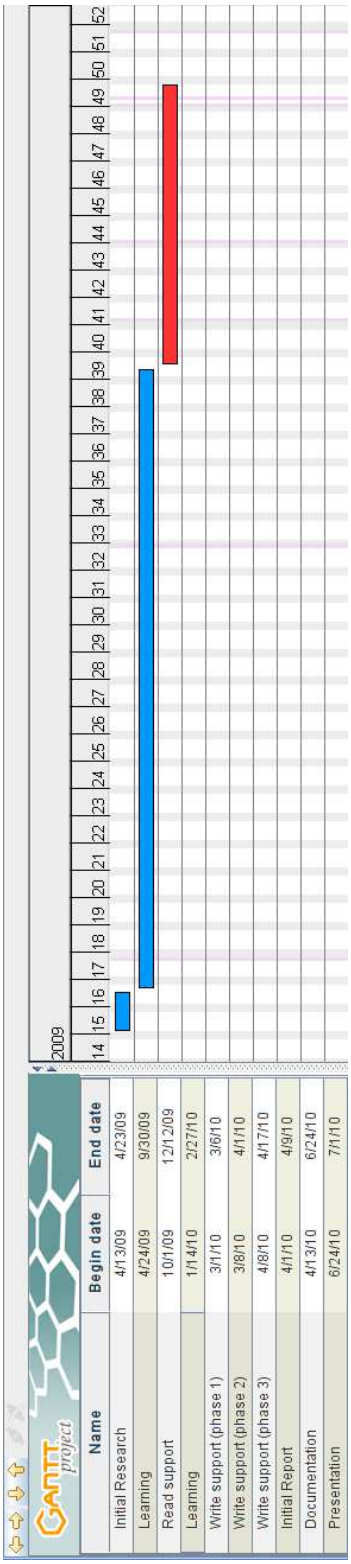


Figure 6.2: Tasks performed during 2010

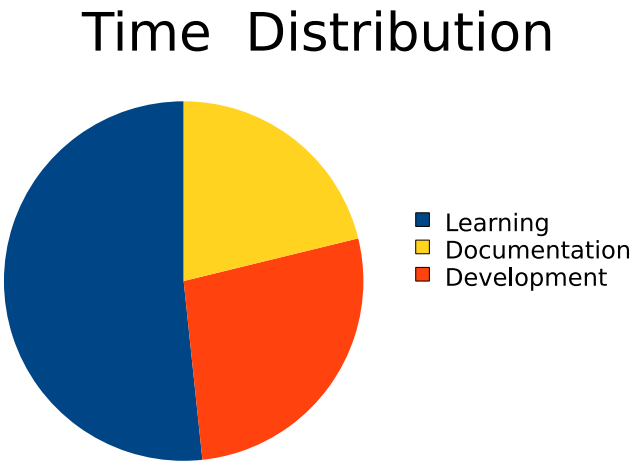


Figure 6.3: Time Distribution by task

6.2 Costs

The project has been developed in its entirety with FOSS (Free Open Source Software), so all the software is free of charge and free to use. This choice was not made on purpose: all the software related with the technologies involved in qloop (Linux, QEMU, etc) happen to be Free Software. Still, the biggest advantage is the invaluable fact that all the source code is available for review. Without this privilege, the development would have been impossible.

About hardware, this project does not require anything beyond an average computer that can run Linux. So the project's cost is determined by the price of the time spent on learning and development. qloop took 420 days to complete. Averaging 4 hours of work/day and 15€/hour, plus the hardware price:

$$Price = (420 days \times 4 hours/day \times 15€/hour) + 300€ = 25500€$$

Chapter 7

Future Work

The objectives set by this project have been met: qloop is an easy to use and fast method to read and write QCOW2 files as if they were real disks. Still, there are many ways to improve it.

7.1 Performance

While qloop's performance is acceptable, there is a noticeable gap between qloop and loop. Since the biggest difference between the two is the address translation mechanism, it is obvious that optimizations should concentrate on this process.

The most direct and effective approach consists in adding a cache for addresses in a similar way to that employed by modern CPUs or paging mechanisms on operating systems. This improvement introduces many variables, such as the size of the cache, should it be associative or direct, or if it could exploit spatial locality (if a cluster is accessed, chances are that the next one will be accessed soon).

There is also the possibility of adding data caching. The reference implementation on QCOW2 (that which is included on QEMU/KVM) loads the entire L1 table in memory for faster access and address translation. Also, it loads clusters currently being accessed. This is done to improve access on compressed and encrypted clusters. There are many chances that a recently-accessed cluster will be accessed again (temporal locality). While the operative system provides a data caching system for block devices, it does not take into account the transformations performed by QCOW2's "transfer functions".

Unfortunately, tuning all these parameters requires extensive testing on different workloads, which demands a huge amount of time and thus could not be done in time for the project.

7.2 More Features

qloop currently only supports basic read and write functionality. Another logical improvement for the project would be adding support for all the features present in the QCOW2 specification such as:

- cluster compression
- cluster encryption
- snapshots
- Copy-on-Write image

Now that the code foundations are solid, it is possible to consider the addition of more advanced features to eventually support the QCOW2 format completely.

7.3 Other formats

There are many other virtual disk formats employed by a variety of emulator or virtualizer applications. Generally, each vendor designs their own format but all of them have a similar set of features. Also, many of these are supported by QEMU, so there is a free implementation available for reference. Here is a list of formats supported by QEMU with the name of their native software/vendor.

- qcow (previous QEMU format, QCOW1)
- vmdk (VMWare)
- vdi (Virtualbox)
- cow (User Mode Linux)
- hdd (Parallels)

In my opinion, the best path to implement them would be modifying qloop to convert it in a common infrastructure and create multiple modules depending on it, each one supporting a different format, to avoid code repetition. On the other hand, if one or more formats proved to be very different to the rest, those could be developed in a module by themselves (without depending on the "generic qloop").

7.4 Upstream Merge

As the Linux developers say, the best place for Linux-related code to be is in the kernel itself [12]. That is, do not develop a module or any other functionality by yourself: try to merge it with the rest of the kernel so that others can help you maintain it and improve it. This way everyone benefits. I agree with this

point of view so I think it is good to try to get qloop accepted into the Linux code distribution (also called mainline or upstream).

Unfortunately, I really doubt that qloop would be accepted at its current state. Sharing 90% or more of code with loop is a good reason that kernel developers will raise against the merge. So code should be adapted to meet the Linux coding standards before any attempt for merging is to be made. Also it would be good to design a patch for loop that would add support for address translation, making qloop smaller and dependent on loop (such as the cryptoloop module) and solving the code repetition problem.

Chapter 8

Conclusions

The qloop module was designed to solve a certain problem: provide easy and fast access to a virtual disk file in QCOW2 format. The method implemented to convert a file into a block device has proved to be fast enough, it is easy to setup and once ready, it can be accessed as a normal block device, which means that all tools available for block devices will work on QCOW2 files too. It could be improved, both on performance and features. But the speed achieved is enough for me and I don't use the advanced features (compression, encryption...). To sum up, qloop fits my needs, or as said in the development world, it "scratches my itch".

On the other hand, it could be argued that the hardest part is done. And after reaching so far, why not advance a bit further? While the current version does what I need, there are some enhancements that are really interesting. But if I were to continue with qloop development, the first step would be pushing for mainline acceptance. There is no point on adding features now if eventually I have to rewrite everything to comply with the kernel's coding standards. Also the idea of joining such a big community of great developers is really exciting. The amount of code I have done is enough to provide a basic functionality that will work for others and catch their attention, and is proof that it is not that hard to add QCOW2 support to the kernel. So even if some code rewrite was necessary, I think the possibility of having this module being part of Linux is not far-fetched at all.

Another interesting conclusion I reached is that all rules have exceptions. While at first it would not look like a great idea to rewrite already existing and fine code, sometimes it is worth it. All the alternatives to qloop implemented contrived methods to provide access to QCOW2 files, always to avoid having to rewrite QCOW2 related code and trying to employ QEMU's existing implementation. In the end, it turns out that rewriting was not as traumatic as expected and provided much better results that all the alternatives in usability, performance and flexibility.

To wrap up, it was a lot of hard work to develop this much, spanning for more than a year. But it was certainly worth it. I have learned so much about operating systems, C development and project management than I thought it

was possible. Now I can only hope that this project is as useful to others as it has been to me.

Bibliography

[1] QEMU official site
http://wiki.qemu.org/Main_Page

[2] KVM official site
http://www.linux-kvm.org/page/Main_Page

[3] Fabrice Bellard
QEMU, a Fast and Portable Dynamic Translator
http://www.usenix.org/publications/library/proceedings-/usenix05/tech/freenix/bellard.html
USENIX 2005 Annual Technical Conference, FREENIX Track

[4] Mark McLoughlin
The QCOW2 Image Format
http://www.gnome.org/ markmc/qcow-image-format.html

[5] QEMU Frequently Asked Questions
http://qemu-buch.de/cgi-bin/moin.cgi/FrequentlyAskedQuestions

[6] FUSE official site
http://fuse.sourceforge.net/

[7] libguestfs API and architecture
http://libguestfs.org/guestfs.3.html#architecture

[8] Network Block Device official site
http://nbd.sourceforge.net/

[9] NBD server for QEMU images (thread in qemu devel mailing list)
http://thread.gmane.org/gmane.comp.emulators.qemu/14907

[10] KVM's first commit to LKML
http://lkml.org/lkml/2006/10/19/146

[11] Splice explanation from Linus Torvalds
http://kerneltrap.org/node/6505

[12] Jonathan Corbet
How to participate in the Linux community
http://ldn.linuxfoundation.org/book/1-a-guide-kernel-development-process

[13] Daniel P. Bovet and Marco Cesati
Understanding the Linux Kernel
O'Reilly, 3rd edition, 2005

[14] Jonathan Corbet, Alessando Rubini and Greg Kroah-Hartman
Linux Device Drivers
O'Reilly, 3rd edition, 2005

Appendix A

Compiling qloop

Compiling qloop from source is an easy and fast process, since the amount of code to compile is pretty small and very similar to already existing kernel code. The requisites for compilation are:

- Linux source
- GNU Compiler Collection
- GNU Make

Compiling is as simple as unpacking the qloop source and running make.

```
$ tar -xzf qloop.tar.gz
$ cd qloop
$ make
```

The code has been tested successfully on Linux 2.6.32 and 2.6.33 with x86 and x86_64 architectures. The kernel sources installed must match the currently running kernel. Once compiled, the module can be loaded with insmod. To communicate with the qloop module and associate it with files, we require *losetup*. This tool is part of the util-linux-ng package, which is available on any Linux distribution, and works in the same manner as the original loop device.

```
$ insmod qloop.ko
```

Appendix B

Usage Example

One of the objectives when designing qloop was making it easy to use. Here is shown an example to proof its simplicity. It is assumed that the qloop module has been compiled and loaded successfully.

```
$ losetup /dev/qloop0 file.qcow2
```

That is all. Now, the device file */dev/qloop0* will behave like a block device containing all the data in the disk described in file.qcow2. This will work as long as file.qcow2 is a sane QCOW2 file that does not employ any of its advanced features (compression, encryption or snapshots) since they are not yet supported by qloop. Accessing a file with any of these features will lead to unexpected results (but you can expect them to be very bad!).

It is possible to create partitions or file-systems and mount them just like a real disk device. Access to partitions in qloop device is possible with tools such as *kpartx*. This tool does not belong to the qloop package and explaining its details is beyond the scope of this documentation, but it is pretty straightforward to use.

```
$ kpartx -a /dev/qloop0
```

This action will create the corresponding device files for each partition existing in the disk under the directory */dev/mapper/*. Undoing this operation is possible by executing kpartx with the -d argument.

Before unloading qloop module, it is necessary to disassociate the qloop device files with QCOW2 files. Again, this action requires losetup.

```
$ losetup -d /dev/qloop0
```

Appendix C

Tools

To develop qloop, I created a few tools that helped me understand the format of QCOW2 files and debug the module. They are not necessary to use qloop, but can be useful for testing purposes. They don't have any requirements beyond a C compiler.

C.1 qcow2info

Extracts detailed information from a QCOW2 file. This was the first tool I developed and thus has a lot of options to perform varied and unrelated tasks because they were added as needed, making its use a bit complicated.

-H Display header.

-t print detailed table information (address translation).

-x Display address/offsets in hexadecimal.

-a *address* Treat this address and translate it to the real offset in the QCOW2 file. Default is 0.

-b bytes Dump *bytes* (up to 4096) to stderr starting at *address*.

C.2 qcow2test

This tool receives a QCOW2 and a raw file as parameters and compares the data they contain byte by byte, printing error messages for every difference it finds between the two.

C.3 printqtables

Prints L1 and L2 tables. Their content is displayed in hexadecimal format as 64bit offsets. It has 1 parameter: a QCOW2 file. It has 2 options.

-1 Display L1 table

-2 Display all L2 tables

C.4 printclusters

Reads a QCOW2 file and prints a map of its clusters. In this "map" format all clusters are printed sequentially in the same order as they are stored in the file, and each cluster is represented as a character. Every character identifies the type of cluster it represents. Legend of characters (or types of cluster) available:

H Header

1 L1 table

2 L2 table

S snapshot

T refcount table

R refcount block

D data cluster

? unknown