File systems

Gaël Thomas

Device and device driver

Device and device driver

- **Device** = hardware component other than CPU and memory
- **Device driver** = software allowing access to a device
    - 1 data structure giving the status of the device
    - 1 input / output function allowing access to the device
    - The driver is usually found in the kernel

Devices in UNIX

- A device is identified by a number called `dev`
    - Most significant bits (*major*): driver number
        - For example: 8 = ssd hard drive driver
- Least significant bits (*minor*): device number
    - For example: 0 = disk 1, 1 = disk 1 / part 1, 2 = disk 1 / part 2
- The kernel contains a table which associates a driver number with the driver (access function + status)

2 types of peripherals

- “character” devices
  - Read / write **byte by byte**
  - Generally access via MMIO or input / output bus

  → **blocks** the CPU during the I/O operation
  - Keyboard, printer, sound card …
- “block” devices
  - Read / write by **data blocks** (typically 512 bytes)
  - The device is therefore seen as an array of blocks
  - Usually access via DMA

  → **does not block** the CPU during the I / O operation
  - Hard disk, DVD player …

Block devices in xv6

- A single block device driver in xv6
  - Manages virtio hard disks (emulated by Qemu)
  - Function `virtio_disk_rw()` in `virtio.c`
- `virtio_disk_rw()` takes two parameters:
  - a boolean, `write`, to tell if it is a read or a write
  - a `buf` (`buf.h`) structure
    - `buf.dev/blockno`: access to block `blockno` from disk `dev`
    - `buf.data`: data read or written
      - If `write == 0`, the **output** of `virtio_disk_rw`, `data` = data read
      - If `write == 1`, the **input** of `virtio_disk_rw`, `data` = data to write

Principle of the `virtio_disk_rw` algorithm

- `virtio_disk_rw` mainly performs the following actions:
  - Setup the DMA data transfer:
    - From disk to memory on a read
    - From memory to disk on a write
  - Sleep the process with the `sleep` function (see lecture #4)
    → switch to another ready process
- Once the transfer is complete
  1. The virtio disk generates an interrupt
  2. The interrupt is handled by the `virtio_disk_intr` function
  3. `virtio_disk_intr` calls `wakeup` to wake up the sleeping process

The I / O cache

- Disk access is very slow compared to memory access
  - Hard disk drive: several milliseconds
  - SSD disk: x10, hundreds of microseconds
  - NVMe disk: x100, microseconds
  - Memory: x100, dozens of nanoseconds
- I/O cache improves the performance of **block type devices**
  - Keeps frequently or recently used blocks in memory
  - Managed by the operating system kernel

Principle of an I/O cache

- The system manages a set of *buffers* in memory
- To read a block (read operation)
    - If the block is not yet in the cache
        1. Remove an unused *buffer* from the cache
        2. Copy the contents of the disk block to this buffer
    - Otherwise, simply return the buffer associated with the block
- To modify a block (write operation)
    1. Read the block (call the read operation)
    2. Modify the contents of the *buffer* in memory
    3. Mark *buffer* as modified (written to disk later)

The xv6 buffer cache

- *buffer cache* = xv6 I/O cache (`bio.c`)
  - Made up of a finite set of `buf` structures
  - Each `buf` structure is associated with a block of a disk
    - A `buf` can be valid if its block's data has been read, invalid otherwise
  - Each `buf` has a reference counter to avoid eviction while still in use

How the buffer cache works: buffer management (1/3)

- The `buf` structures form a circular double linked list, **the head is the most recently used block**
- `struct buf* bget(uint dev, uint blkno)`: return a **locked** buffer associated with (`dev, blkno`)
  - If there is already an *buffer* associated with (`dev,blkno`)
    - Increment the reference counter of the *buffer*
    - Lock the *buffer*
    - Return the *buffer*
  - Otherwise
    - Search for a *buffer* with counter == 0
    - Associate the *buffer* with (`dev, blkno`)
    - And then, same as above

How the buffer cache works: read buffer (2/3)

- `struct buf* bread(uint dev, uint blkno)`
  - Goal: return a locked buffer for this block in the valid state
    1. Call `bget()` to find a *buffer* for this block
    2. If the *buffer* is invalid, call `virtio_disk_rw()`
- `void bwrite(struct buf* b)`
  - Call `virtio_disk_rw()` to write the buffer data to the disk

How the buffer cache works: write buffer (3/3)

- `void brelse(struct buf* b)`
  - Release the lock associated with `b`
  - Decreases the reference counter
  - Move the buffer to the head of the list (most recently used) if it is unused

The log

Operation versus writing to disk

- A **write operation** of a process often requires **several block writes**
  - File creation requires:
    - Allocation of a new file
    - Adding the name to a directory
  - Adding data to a file requires:
    - Writing new blocks to disk
    - Updating the file size
  - Deleting a file requires:
    - Deleting the data blocks from the file
    - Deleting the name from the directory
  - …

Consistency issues

- The **system can crash** anytime
  → Inconsistency if it stops in the middle of an operation
  - A name in a directory references a non-existent file
  - Data added to a file but size not updated
  - …
- **Operations must be propagated in the order** in which they were performed
  → Inconsistency if propagation in random order
  - Adding a file then deleting $\Longrightarrow$ the file does not exist at the end
  - Deleting a file then adding $\Longrightarrow$ the file exists at the end
  - Similarly, adding data then truncating (size should be 0)
  - …

Bad solutions

- No cache when writing (directly propagate write operations)
  - Very inefficient because each write becomes very (very!) slow
- Recovery in the case of a crash
  - Recovering a file system is slow
  - examples: FAT32 on Windows or ext2 on Linux
  - Recovering is not always possible
    → a crash makes the filesystem unusable!

First idea: transactions

- A transaction is a set of writes that is
  - Either fully executed
  - Or not executed at all
- Principle of implementation
  - **An operation (coherent set of writes) == a transaction**
  - The writes of a transaction are first written to disk in a “pending” area
  - Once the operation is complete, the “pending” area is marked as valid (**the transaction is complete**)
  - Regularly (or in the event of a crash), validated writes in the pending zone are propagated to the file system

Second idea: log

- To ensure that the entries are propagated in order in which they were executed, the *pending* zone is structured like a log
    - Each entry is added **at the end** of the log
    - The validated transactions of the pending zone are propagated to the file system **in the order** of the log (from the start of the log to the end)

Third idea: parallel log

- Problems: Multiple processes may perform transactions in parallel
  - Parallel transaction writes are interleaved in the log

  → How do you know which ones are validated?
- Classic solution
  - If several transactions in parallel, all the operations are validated when **the last** one is completed
  - Advantage: easy to implement (count of the number of operations in parallel)
  - Disadvantage: risk of never validating if new operations continue to arrive

Log structure

- The system technically manages two logs
  - One in memory called **memory log**
    - Contains only the list of modified block numbers
    - The content of the modified blocks is in the buffer cache
  - One on disk called **disk log**
    - Contains the list of modified block numbers and a copy of the blocks
    - Note: the block is propagated from the log to the filesystem later

    → The system can therefore manage up to 3 copies of a block
      - One on disk in the file system called **disk block**
      - One on disk in the log called **disk log block**
      - One in memory in the buffer cache called **cached block**

Log algorithm principle

- Steps to modify block number `n`
  1. load the **disk block** in the buffer cache
  2. modification of the buffer (i.e. **cached block**)
  3. add `n` to the **list of modified blocks** in the **memory log**
- At the end of an operation, steps to validate the transaction
  1. copy modified **cached blocks** to **disk log**
  2. copy the modified **block list** to **disk log**
  3. mark the transaction as validated
- Later, to propagate the transaction
  1. copy **disk log blocks** to file system
  2. reset **disk log** and **memory log**

Using the log

- Three functions in the log management interface (`log.c`)
  - `begin_op()` : start a transaction
  - `end_op()` : validate a transaction
  - `log_write(struct buf* b)` : add `b` to the transaction
- To perform a logged operation, instead of calling directly `bwrite ()`, we have to execute:

```
begin_op();
b = bread(...);
// Modify data of b
...
log_write(b2);
...
end_op();
```

Implementation in xv6 (1/3)

- `void begin_op()` : start a transaction
  - If log writing to disk in progress, wait
  - If the log is full, wait
  - Increments the number of pending operations (`log.outstanding`)
- `void end_op()` : complete a transaction
  - Decrement the number of operations in progress, and if equal to 0:
    - Write **memory log** + **cached blocks** in **disk log** (`write_log()`)
    - Mark committed **disk log** transaction (`write_head()`)
    - Propagate writes from **disk log** to the filesystem (`install_trans()`)
    - Delete logs in memory and on disk (`write_head()`)

Implementation in xv6 (2/3)

- `void log_write(struct buf* b)`
  - Goal: put the block associated with `b` in the log
  - Find an entry for the block in the log
    - If already in the log: *absorb* the log entry (i.e., do nothing: the block is already logged to be written)
    - If new to the log:
      1. Add block number to the **memory log**
      2. Increase the reference counter of the buffer `b` to prevent it from leaving the buffer cache

Implementation in xv6 (3/3)

- After a crash, call `install_trans()` which propagates the writes from **disk log** to file system
  - In the worst case, writes that had already been performed are replayed
  - But at the end of the replay, the filesystem is in a consistent state

Partitions and file systems

File system

- File system: defines the structure for storing files (often for a block type device)
  - UFS : Unix Files System (xv6, BSD)
  - ext : extended file system (Linux - ext4 nowadays)
  - NTFS : New Technology File System (Windows)
  - APFS : APple File System (MacOS)
  - FAT : File Allocation Table (Windows)
  - BTRFS : B-TRee File System (Linux)
  - and many others !

Principle of a file system

- File = consistent set of data that can be read or written
- Filesystem = associate **names** and **files**
    - Example : `/etc/passwd` → `root:*:0:0:System Administrator...`
    - Usually a special symbol is used as a separator for directories
        - `/` in UNIX systems, `\` in Windows systems

Partitions

- A disk is often made up of several partitions
  - Partition = continuous area that contains a file system
- Typical structure of a disk
  - First block: partition table
    - For example: Master Boot Record
  - Blocks 2 to x: kernel loader
    - In charge of loading the kernel of one of the partitions
    - For example: LILO, GRUB
  - Blocks x to y: partition 1
  - Blocks y to z: partition 2
  - etc…

Disk image

- A file itself can contain the data of a complete disc
  - Called a **disk image** or a **virtual disk**
  - Typically used in virtualization
  - For example: `xv6.img` is the disk image used with the qemu emulator to start `xv6`

UFS/xv6 file system

Overall file system structure

- Five large contiguous zones (in `fs.h`)
  - The **super block** describes the other areas
  - The **journal** contains the disk logs
  - The **dinode table** contains the metadata of the files (size, type like ordinary or directory …)
  - The **table of free blocks** indicates the free blocks
  - The **data blocks area** contains the data of the files

Dinode

- A file on disk consists of:
  - metadata called **a dinode** (fixed size, see `fs.h`)
    - file type (ordinary, directory, device)
    - file size
    - the list of the file data blocks
    - an indirection block (see following slides)
    - device number if device file
    - number of hard links to the file (reminder: a hard link is a name in a directory)
  - data blocks
    - these are the blocks that contain the content of the file

Data blocks of a file

- A dinode directly lists the numbers of the first 12 blocks
  - the `dinode.addrs [0]` block contains bytes 0 to 511 of the file
  - …
  - the `dinode.addrs [i]` block contains the bytes i * 512 to i * 512 + 511
- The indirection block contains the following block numbers
  - the indirection block number ind is given in dinode.addrs [12]
  - the `ind [0]` block contains bytes 12 * 512 to 12 * 512 + 511
- Note: since a block is 512 bytes and a block number is coded out of 4 characters, a file has a maximum size of 12 + 512/4 blocks.

Adding a block to a file

- To add a new block to a dinode `dino` (function `bmap ()` in `fs.h`)
    1. Find a free block number in the **table of free blocks** \ (function `balloc()` in `fs.h`)
    2. Mark the occupied block (put its bit 1 in the **table**)
    3. Add the block number to the list of data blocks in `dino`
        - this addition may require to allocate an indirection block

Directories

- A **directory is a file** of type `T_DIR`
- Contains an array associating names and numbers of dinodes
  - **inum**: inode number
  - **name**: file name
- Inode 1 is necessarily a directory: it is the root directory of the filesystem
- Note: `dinode.nlink` gives the number of times a dinode is referenced from a directory
  $\Longrightarrow$ file deleted when `nlink` equals to 0.

From path to inode

- To find a dinode number from the path `/e0/../en` (see `namex()` in `fs.c`)

```
cur = 1
For i in  [0 .. n]
    Look for the association [inum, name] in the data blocks of
        the cur dinode such that name is ei
    cur = inum
```

File creation and deletion

- To **create** the file `f` in the `d` directory (function `create()` in `sysfile.c`)
    1. Find a free inum dinode by finding an inode whose type is 0 in the dinode array (`ialloc ()` in `fs.h`)
    2. Add the association `[inum, f]` to `d`
- To **delete** the file `f` from the `d` directory (`sys_unlink()` function in `sysfile.c`)
    1. Delete the entry corresponding to `f` in `d`
    2. Decrement `nlink` from `f` and if `nlink` equals 0
    3. Delete data blocks from file `f`
    4. Remove the inode `f` (setting its type to 0)

xv6 I/O stack

Inode

- **inode** = memory cache of a **dinode**
  - Enter the cache at `open()`
  - Can be evicted from cache from `close()`
  - Contains the fields of the dinode
  - + fields to know which dinode the inode corresponds to
    - Device number and dinode number
  - + fields required when the dinode is used
    - A lock to manage concurrent access
    - A counter giving the number of processes using the inode to know when the inode can be evicted from the cache
- **Inode table** = table which contains the inodes

Main functions of inodes (1/3)

- `struct inode* iget(int dev, int inum)`
  - Corresponds to `open()`: returns an inode associated with `[dev, inum]`
  - Increments the inode usage counter (non-evictable)
  - **Do not lock** the inode and **do not read** the inode from disk (optimization to avoid disc playback when creates a file)
    - `inode.valid` indicates whether the inode has been read from disk
- `void ilock(struct inode* ip)`
  - **Acquires a lock** on the inode
  - **Read inode** from disk if not already read
- `void iunlock(struct inode* ip)`
  - Release the lock on the inode

Main functions of inodes (2/3)

- `void itrunc(struct inode* ip)`
  - Free all the blocks in the file (size 0)
- `void iupdate(struct inode* ip)`
  - Copy the inode to the disk dinode (technically, via the I/O cache)

Main functions of inodes (3/3)

- `void iput(struct inode* ip)`
  - Corresponds to `close ()`
  - Decreases the inode usage counter
  - If cpt drops to 0, the inode can be evicted from the cache and
    - If nlink is 0 (the inode is no longer referenced by a directory)
      - Delete data blocks from inode (itrunc)
      - Mark the inode as free (type = 0)
- Note: if you delete a file from a directory (`unlink()`) while the file is still in use (open) by a process, the inode is not deleted: it will be when last `close()` when the reference counter drops to 0.

Open files

- Multiple processes can open the same file
  - Each process has independent read / write permissions
  - Each process has a read cursor, which is independent of that of the other processes
- A file structure opened by `open ()` contains:
  - A pointer to an inode
  - Access permissions
  - A reading cursor

File descriptors

- Each process has an `ofile` table of open files
  - A descriptor `d` is an index in this table
  - `proc[i].ofile[d]` points to an open file
  - `proc[i].ofile[d].ip` points to inode
- Good to know
  - During a `fork()`, the parent and the child share the open files
  - So `proc[parent].ofile[d] == proc[child].ofile[d]`
  - And so, if the father reads, the child read cursor changes
  - Useful for setting up pipes

What you must remember

- A device driver is just a function (`virtio_disk_rw()` for example)
- Reads and writes are logged
    - Ensures file system consistency in the event of a crash
- The kernel has an I/O cache
    - Is in memory, managed by the kernel
    - Allows to speed up I/O
- A file system separates
    - The naming (directory) of the files (dinodes + data blocks)
    - The metadata (dinode) of the data blocks
- A file descriptor is an index in the ofile table
    - `proc->ofile[i]` is an open file that references an inode