File systems

Gaël Thomas

CSC4508 – Operating Systems
2020–2021
Outlines

1. Device and device driver ......................................................... 3
2. The I/O cache ........................................................................ 9
3. The log ................................................................................. 16
4. Partitions and file systems ...................................................... 29
5. UFS/xv6 file system ................................................................. 34
6. Xv6 I/O stack ......................................................................... 42
7. What you must remember ...................................................... 49
## 1 Device and device driver

1.1 Device and device driver ................................................................. 4
1.2 Devices in UNIX .............................................................................. 5
1.3 2 types of peripherals ..................................................................... 6
1.4 Block devices in xv6 ....................................................................... 7
1.5 Principle of the iderw algorithm ..................................................... 8
1.1 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
1.2 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)
1.3 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 ...
1.4 Block devices in xv6

A single block device driver in xv6

- Manages IDE hard disks
- Function `iderw()` in `ide.c`

`iderw()` takes a `buf (buf.h)` structure as a parameter

- `buf.flags`:
  - B_VALID: if `false`, read operation requested
  - B_DIRTY: if `true`, write operation requested

- `buf.dev/blockno`: access to block `blockno` from disk `dev`

- `buf.data`: data read or written
  - If `read`, the output of `iderw`, data = data read
  - If `write`, the input of `iderw`, data = data to write
1.5 Principle of the iderw algorithm

- ide rw mainly performs the following actions:
  - Start the DMA transfer (see lecture #5)
    - From memory to disk if write request
    - From disk to memory if read request
  - Sleep the process with the sleep function (see lecture #4)
    → switch to another ready process

- Once the transfer is complete
  - The disk generates an interrupt
  - The interrupt is handled by the ideintr function
  - ideintr calls wakeup to wake up the sleeping process
2 The I/O cache

2.1 The I/O cache ................................................................. 10
2.2 Principle of an I/O cache ...................................................... 11
2.3 The xv6 buffer cache .......................................................... 12
2.4 How the buffer cache works (1/3) ............................................ 13
2.5 How the buffer cache works (2/3) ............................................ 14
2.6 How the buffer cache works (3/3) ............................................ 15
2.1 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
2.2 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. Modifies the contents of the buffer in memory
  3. Mark buffer as modified (written to disk later)
2.3 The xv6 buffer cache

- buffer cache = xv6 I/O cache
  - Made up of a finite set of buf structures
  - Each buf structure is associated with a block of a disk

- Three possible states
  - ! B_VALID: read operation incomplete \( \Rightarrow \) requires read
  - B_VALID and ! B_DIRTY: data in memory and buffer is unmodified
  - B_VALID and B_DIRTY: data in memory and buffer is modified
    \( \Rightarrow \) need to be written to disk before leaving the cache

\( \Rightarrow \) in iderw(), ! B_VALID \( \Leftrightarrow \) read and B_DIRTY \( \Leftrightarrow \) write
2.4 How the buffer cache works (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 to (dev, blkno)
  - If there is already an buffer associated with (dev, blkno)
    - Increments a reference counter associated with the buffer
    - Locks the buffer
    - Return the buffer
  - Otherwise
    - Search for a buffer with counter == 0 and with the state ! B_DIRTY
    - Associate the buffer with (dev, blkno) (+ cpt = 1 and lock the buffer)
2.5 How the buffer cache works (2/3)

- struct buf* bread(uint dev, uint blkno)
  - Return a locked buffer in the B_VALID state
  - Call bget()
  - If the buffer is not B_VALID, call iderw()

- void bwrite(struct buf* b)
  - Writes the contents of b to disk
  - Mark the buffer B_DIRTY
  - Call iderw() to write the buffer
2.6 How the buffer cache works (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)
3 The log

3.1 Operation versus writing to disk ......................................................... 17
3.2 Consistency issues .............................................................................. 18
3.3 Bad solutions ...................................................................................... 19
3.4 First idea: transactions ................................................................. 20
3.5 Second idea: log .............................................................................. 21
3.6 Third idea: parallel log ................................................................. 22
3.7 log structure .................................................................................. 23
3.8 Log algorithm principle ............................................................... 24
3.9 Using the log ............................................................................... 25
3.10 Implementation in xv6 (1/3) ......................................................... 26
3.11 Implementation in xv6 (2/3) ......................................................... 27
3.12 Implementation in xv6 (3/3) ......................................................... 28
3.1 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
  - ...

3.2 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 $\implies$ the file does not exist at the end
    - Deleting a file then adding $\implies$ the file exists at the end
    - Similarly, adding data then truncating (size should be 0)
    - ...

### 3.3 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!
3.4 First idea: transactions

A transaction is a set of write operation 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
3.5 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)
3.6 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 //, all the operations are validated when the last one is completed
  ♦ Advantage: easy to implement (count of the number of operations in //)
  ♦ Disadvantage: risk of never validating if new operations continue to arrive
3.7 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**
3.8 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
### 3.9 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()`, so we have to execute:
  ```c
  begin_op()
  log_write(b1)
  log_write(b2)
  ...
  end_op()
  ```
3.10 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
  - ♦ Decrements 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())
3.11 Implementation in xv6 (2/3)

- void log_write(struct buf* b)
  - Add the block associated with b to the log
  - Add block number to memory log
  - Mark buffer as B_DIRTY $\implies$ does not leave the cache (see bget())
3.12 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
4 Partitions and file systems

4.1 File system ......................................................... 30
4.2 Principle of a file system ........................................... 31
4.3 Partitions ............................................................ 32
4.4 Disk image .......................................................... 33
4.1 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!
4.2 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
4.3 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...
4.4 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`
5 UFS/xv6 file system

5.1 Overall file system structure ................................................................. 35
5.2 Dinode ...................................................................................................... 36
5.3 Data blocks of a file .................................................................................... 37
5.4 Adding a block to a file .............................................................................. 38
5.5 Directories .................................................................................................. 39
5.6 From path to inode ....................................................................................... 40
5.7 File creation and deletion ........................................................................... 41
5.1 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
5.2 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
### 5.3 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.
5.4 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
5.5 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

⇒ file deleted when nlink equals to 0.
5.6 From path to inode

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

1. `cur = 1`

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

   (b) `cur = inum`
5.7 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 \([\text{inum}, f]\) 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)
6 Xv6 I/O stack

6.1 Inode ................................. 43
6.2 Main functions of inodes (1/3) ......................................................... 44
6.3 Main functions of inodes (2/3) ......................................................... 45
6.4 Main functions of inodes (3/3) ......................................................... 46
6.5 Open files ........................................ 47
6.6 File descriptors ........................................ 48
6.1 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
6.2 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**
6.3 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)
6.4 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.
6.5 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
6.6 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
# 7 What you must remember

- **A device driver is just a function** (`ideWrite()` 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