Discussion 9: File Systems (continued), Reliability

April 14, 2023

Contents

1 File Allocation Table (FAT)

File Allocation Table (FAT) is a file system developed by Microsoft in the 1970s.

Index Structure

FAT file system is named after its index structure which is called the **file allocation table**, an array where each element corresponds to a disk block. Each file corresponds to a linked list of FAT entries containing a pointer to the next FAT entry. This means that each element N in FAT is an integer. This represents the index within the array of the next block within the file. The first few entries in the FAT array are reserved to store key data such as the boot sector, the array itself, and root directory. The index of the first entry of a file (i.e. the index that it starts at) is set to be the file number.

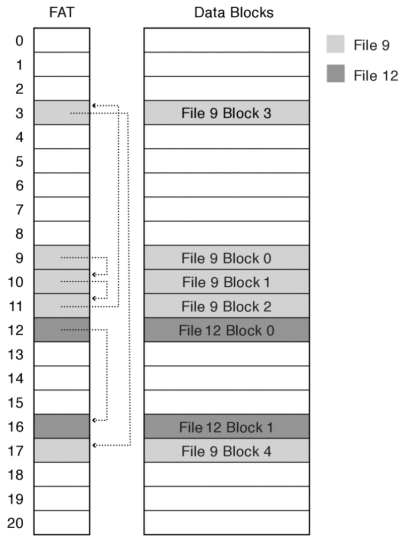

Figure 1: FAT

While no longer the default file system for Microsoft's devices, FAT is still commonly found on many portable and embedded devices (e.g. digital cameras) due to its simplicity and support by many operating systems. However, the FAT file system suffers from bad random access performance since accessing a specific offset in a file requires traversing through the linked list. Moreover, file sizes are encoded in 32 bits, so no file can be larger than $2^{32} - 1$ bytes (i.e. 4 GiB).

Directory Structure

Contrary to other file systems, the FAT file system is forced to store metadata as part of the directory entry rather than in the FAT entry itself since the FAT entry doesn't have any space; it's merely a 32-bit integer as discussed earlier. As a result, each file can be accessed via exactly one directory entry, meaning there can't be any hard links.

Free Space Management

The FAT file system uses the FAT array for free space tracking as well. If the entry of a FAT array is 0, it marks a free entry. When allocating a new file or an additional block for an existing file, the file system will walk through the FAT array to find a zero entry. As a result, allocation of multiple sectors may take a long time. Furthermore, disk blocks within one file may be spread across far-apart disk blocks instead of being placed sequentially, which reduces disk performance.

1.1 Concept Check

1. What does it mean to format a FAT file system? Approximately how many bytes of data need to be written in order to format a 2 GiB flash drive (with 4 KiB blocks and a FAT entry size of 4 bytes) using the FAT file system?

   Formatting a FAT file system means resetting the file allocation table (mark all blocks as free). The actual data can be zero-ed out for additional security, but it is not required. Formatting a 2 GiB FAT volume will require resetting 2^{19} FAT entries, which take up 2^{21} bytes = 2 MiB.

2. Your friend who has never taken an Operating Systems class wants to format their external hard drive with the FAT32 file system. The external hard drive will be used to share home videos with your friend's family. Give one reason why FAT32 might be the right choice. Then, give one reason why your friend should consider other options.

   FAT32 is supported by many different operating systems, which will make it a good choice for compatibility if it needs to be used by many users. However, FAT32 has a 4GiB file size limit, which may prevent your friend from sharing large video files with it.

3. From a software perspective, explain how an operating system reads a file like `D:\My Files\Video.mp4` from a FAT volume.

   First, the operating system must know that the FAT volume is mounted as `D:\`. It looks at the first data block on the FAT volume, which contains the root directory, and searches for the `My Files` subdirectory. If necessary, the root directory listing might occupy many blocks, and the operating system will follow the pointers in the FAT to scan through the entire root directory. Once the subdirectory entry for `My Files` is found, the operating system searches the subdirectory's listing the `Video.mp4` file. Once it knows the block number for the file, it can begin reading the file sequentially by following the pointers in the FAT.

2 File Growth

In this question, we will explore how to grow a file in Pintos. This will be very similar to one of your tasks in Project File System.

In Pintos, `struct inode_disk` from `filesys/inode.c` represents an inode. Let's examine a modified `struct inode_disk` with 12 direct pointers and an indirect pointer.

```
#define BLOCK_SECTOR_SIZE   512

typedef uint32_t block_sector_t;

struct inode_disk {
  block_sector_t direct[12];   /* Direct pointers */
  block_sector_t indirect;     /* Indirect pointer */
  off_t length;                /* File size in bytes. */
  uint32_t unused[114];        /* Not used. */
};
```

1. What is the purpose of the `unused` member in `struct inode_disk`?

Each inode must be of size equivalent to the block size. The direct pointers, indirect pointer, and size only take up

$$4 \times (12 + 1 + 1) = 56 \text{ bytes}$$

As a result, we need to pad the rest of the struct with $512 - 56 = 456$ bytes, or equivalently an array of 114 32-bit integers.

2. What is the maximum file size supported by this file system?

There are $512/4 = 128$ direct pointers in an indirect block, so the maximum file size is

$$\begin{aligned} 512 \times (12 + 128) &= 2^9 \times (2^2 \times 3 + 2^7) \\ &= 2^{10} \times (2 \times 3 + 2^6) \\ &= 70 \text{ KiB} \end{aligned}$$

3. What data structure should you to represent an indirect block?

Since an indirect block is an array of direct pointers, `block_sector_t[128]` would be adequate.

4. Implement `inode_resize` to grow or shrink the inode based on the given `size`. If the resize operation fails for any reason, the inode should be unchanged and the function should return `false`. Assume unallocated block pointers have value `0`.

```
/* Allocates a disk sector and returns its number. */
block_sector_t block_allocate(void);

/* Frees disk sector N. */
void block_free(block_sector_t n);

/* Reads contents of disk sector N into BUFFER. */
void block_read(block_sector_t n, uint8_t buffer[512]);

/* Write contents of BUFFER to disk sector N. */
void block_write(block_sector_t n, uint8_t buffer[512]);

bool inode_resize(struct inode_disk* id, off_t size) {
  block_sector_t sector;

  /* Handle direct pointers. */
  for (int i = 0; i < 12; i++) {
    if (size <= BLOCK_SECTOR_SIZE * i && id->direct[i] != 0) {
      /* Shrink. */
      _________________________________;
      _________________________________;
    } else if (size > BLOCK_SECTOR_SIZE * i && id->direct[i] == 0) {
      /* Grow. */
      _________________________________;
    }
  }

  /* Check if indirect pointers are needed. */
  if (id->indirect == 0 && size <= 12 * BLOCK_SECTOR_SIZE) {
    id->length = size;
```

```
    return true;
  }
  block_sector_t buffer[128];
  memset(buffer, 0, 512);
  if (id->indirect == 0) {
    /* Allocate indirect block. */
    _________________________________;
  } else {
    /* Read in indirect block. */
    _________________________________;
  }

  /* Handle indirect pointers. */
  for (int i = 0; i < 128; i++) {
    if (size <= (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] != 0) {
      /* Shrink. */
      _________________________________;
      _________________________________;
    } else if (size > (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] == 0) {
      /* Grow. */
      _________________________________;
    }
  }
  if (size <= 12 * BLOCK_SECTOR_SIZE) {
    _________________________________;
    _________________________________;
  } else {
    _________________________________;
  }
  id->length = size;

  return true;
}
```

```
bool inode_resize(struct inode_disk* id, off_t size) {
  block_sector_t sector;

  /* Handle direct pointers. */
  for (int i = 0; i < 12; i++) {
    if (size <= BLOCK_SECTOR_SIZE * i && id->direct[i] != 0) {
      /* Shrink. */
      block_free(id->direct[i]);
      id->direct[i] = 0;
    } else if (size > BLOCK_SECTOR_SIZE * i && id->direct[i] == 0) {
      /* Grow. */
      id->direct[i] = block_allocate();
    }
  }

  /* Check if indirect pointers are needed. */
  if (id->indirect == 0 && size <= 12 * BLOCK_SECTOR_SIZE) {
    id->length = size;
    return true;
```

```
  }
  block_sector_t buffer[128];
  memset(buffer, 0, 512);
  if (id->indirect == 0) {
    /* Allocate indirect block. */
    id->indirect = block_allocate();
  } else {
    /* Read in indirect block. */
    block_read(id->indirect, buffer);
  }

  /* Handle indirect pointers. */
  for (int i = 0; i < 128; i++) {
    if (size <= (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] != 0) {
      /* Shrink. */
      block_free(buffer[i]);
      buffer[i] = 0;
    } else if (size > (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] == 0) {
      /* Grow. */
      buffer[i] = block_allocate();
    }
  }
  if (size <= 12 * BLOCK_SECTOR_SIZE) {
    /* We shrank the inode such that indirect pointers are not required. */
    block_free(id->indirect);
    id->indirect = 0;
  } else {
    /* Write the updates to the indirect block back to disk. */
    block_write(id->indirect, buffer);
  }
  id->length = size;

  return true;
}
```

5. How would you modify your solution to the previous question to handle sector allocation failures (i.e. disk runs out of space)?

Any time a new sector is allocated using `block_allocate`, you should add a check that resembles the following.

```
sector = allocate_block();
if (sector == 0) {
  inode_resize(id, id->length);
  return false;
}
```

3 Reliability

Availability

Availability is the probability that the system can accept and process requests. This is measured in "nines" of probability (e.g. 99.9% is said to be "3-nines of availability").

Durability

Durability is the ability of a system to recover data despite faults (i.e. fault tolerance). It's important to note that durability does not necessarily imply availability.

When making a file system more durable, there are multiple levels which we need to concern ourselves with. For small defects in the hard drive, Reed-Solomon error correcting codes can be used in each disk block. When using a buffer cache or any other delayed write mechanism, it's important to make sure dirty data gets written back to the disk. To combat unexpected failures or power outages, the computer can be built with a special, battery-backed RAM called non-volatile RAM (NVRAM) for dirty blocks in the buffer cache.

To make sure the data survives in the long term, it needs to be replicated to maximize the independence of failures. **Redundant Array of Inexpensive Disks (RAID)** is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures. RAID 1 will **mirror** a disk onto another "shadow" disk. Evidently, this is a very expensive solution as each write to a disk actually incurs two physical writes. RAID 5 will stripe data across n multiple disks to allow for a single disk failure. Each stripe unit consists of $n-1$ blocks and one **parity block**, which is created by XOR-ing the $n-1$ blocks. To recover from a disk failure, the system simply needs to XOR the remaining blocks.

Reliability

Reliability the ability of a system or component to perform its required functions under stated conditions for a specified period of time. This means that the system is not only up (i.e. availability), but also performing its jobs correctly. Reliability includes the ideas of availability, durability, and security.

One approach taken by FAT and FFS (in combination with `fsck`) is **careful ordering and recovery**. For instance, creating a file in FFS may be broken down into the following steps

1. Allocate data block.
2. Write data block.
3. Allocate inode.
4. Write inode block.
5. Update free map.
6. Update directory entry.
7. Update modify time for directory entry.

To recover from a crash, `fsck` might take the following steps.

1. Scan inode table.
2. If any unlinked files (not in any directory), delete or put in lost and found directory.
3. Compare free block bitmap against inode trees.
4. Scan directories for missing update/access times.

It's important to note that this is not a foolproof method. While there are a few ways that failures can happen in spite of this method, the file system will be recoverable most of the times.

The other approach is to use **copy on write (COW)**. Instead of updating data in-place, new versions are written to a new location on disk, and the appropriate mappings and references to these data are subsequently updated. Since the mappings and references are updated last, this allows for easy recovery if the system crashes sometime in the middle of updating data since the old data and mapping will still be in tact. Furthermore, data is being only added, not modified, so batch updates and parallel writes can help improve performance.

3.1 Concept Check

1. What benefit with regards to read bandwidth might you see from using RAID 1?

   Read bandwidth can be doubled since there are two copies of the same data.

2. What is the minimum number of disks to use RAID 5?

   3. A disk is recovered by XOR-ing at least two other disks. It's important to note that the RAID level is not an indication of how many disks are needed.

3. RAID 4 had a dedicated disk with all the parity blocks. On the other hand, RAID 5 distributes the parity blocks across all disks in a round robin fashion. Why is this approach beneficial in terms of write bandwidth?

   When updating data, the parity block always needs to be written to. If all the parity blocks are in one disk like in RAID 4, this becomes a bottleneck as multiple writes to disk ultimately contend on one disk. As a result, it's better to distribute the parity blocks evenly such that more writes can happen without contending for the same disk.

4. How can COW help with write speeds?

   COW can transform random I/O into sequential I/O. For instance, take the example of appending a block to a file. In a traditional in-place system like FFS, the free space bitmap, file's inode, file's indirect block, and file's data block all need to be updated. On the other hand, COW could just find sequential unused blocks in disk and write the new bitmap, inode, indirect block, and data block.