Section 10: File Systems

CS 162

November 6, 2020

Contents

1 Vocabulary

- **Simple File System** - The disk is treated as a big array. At the beginning of the disk is the Table of Content (TOC) field, followed by data field. Files are stored in data field contiguously, but there can be unused space between files. In the TOC field, there are limited chunks of file description entries, with each entry describing the name, start location and size of a file.

  Pros and Cons

  The main advantage of this implementation is simplicity. Whenever there is a new file created, a continuous space on disk is allocated for that file, which makes I/O (read and write) operations much faster.

  However, this implementation also has many disadvantages. First of all, it has external fragmentation problem. Because only continuous space can be utilized, it may come to the situation that there is enough free space in sum, but none of the continuous space is large enough to hold the whole file. Second, once a file is created, it cannot be easily extended because the space after this file may already be occupied by another file. Third, there is no hierarchy of directories and no notion of file type.

- **External Fragmentation** - External fragmentation is the phenomenon in which free storage becomes divided into many small pieces over time. It occurs when an application allocates and deallocates regions of storage of varying sizes, and the allocation algorithm responds by leaving the allocated and deallocated regions interspersed. The result is that although free storage is available, it is effectively unusable because it is divided into pieces that are too small to satisfy the demands of the application.

- **Internal Fragmentation** - Internal fragmentation is the space wasted inside of allocated memory blocks because of the restriction on the minimum allowed size of allocated blocks.

- **FAT** - In FAT, the disk space is still viewed as an array. The very first field of the disk is the boot sector, which contains essential information to boot the computer. A super block, which is fixed sized and contains the metadata of the file system, sits just after the boot sector. It is immediately followed by a **file allocation table** (FAT). The last section of the disk space is the data section, consisting of small blocks with size of 4 KiB.

  In FAT, a file is viewed as a linked list of data blocks. Instead of having a "next block pointer" in each data block to make up the linked list, FAT stores these pointers in the entries of the file allocation table, so that the data blocks can contain 100% data. There is a 1-to-1 correspondence between FAT entries and data blocks. Each FAT entry stores a data block index. Their meaning is interpreted as:

  If $N > 0$, N is the index of next block

  If $N = 0$, it means that this is the end of a file

  If $N = -1$, it means this block is free

  Thus, a file can be stored in a non-continuous pattern in FAT. The maximum internal fragmentation equals to 4095 bytes (4K bytes - 1 byte).

  Directory in the FAT is a file that contains directory entries. The format of directory entries look as follows:

  Name — Attributes — Index of 1st block — Size

  Pros and Cons

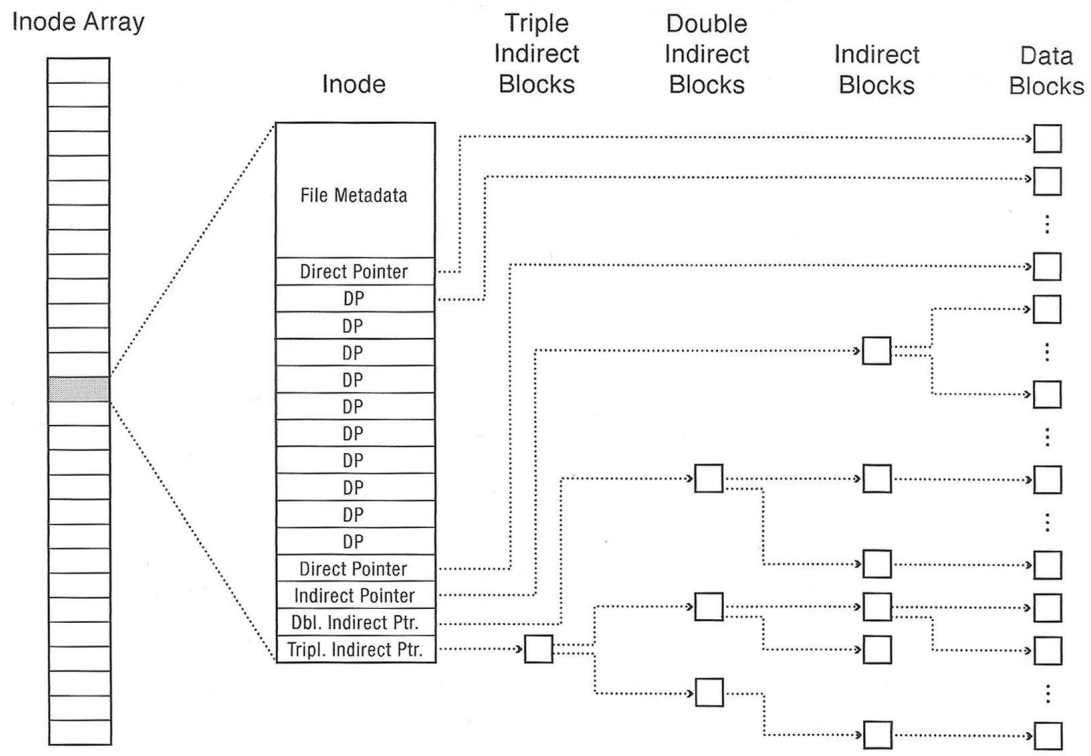
  Now we have a review of the pros and cons about FAT. Readers will find most of the following features have been already talked about above. So we only give a very simple list of these features.

Pros: no external fragmentation, can grow file size, has hierarchy of directories

Cons: no pre-allocation, disk space allocation is not contiguous (accordingly read and write operations will slow), assume File Allocation Table fits in RAM. Otherwise lseek and extending a file would take intolerably long time due to frequent memory operation.

- **Unix File System (Fast File System)** - The Unix File System is a file system used by many Unix and Unix-like operating systems. Many modern operating systems use file systems that are based off of the Unix File System.
- **inode** - An inode is the data structure that describes the metadata of a file or directory. Each inode contains several metadata fields, including the owner, file size, modification time, file mode, and reference count. Each inode also contains several data block pointers, which help the file system locate the file's data blocks.

  Each inode typically has 12 direct block pointers, 1 singly indirect block pointer, 1 doubly indirect block pointer, and 1 triply indirect block pointer. Every direct block pointer directly points to a data block. The singly indirect block pointer points to a block of pointers, each of which points to a data block. The doubly indirect block pointer contains another level of indirection, and the triply indirect block pointer contains yet another level of indirection.



2 File Allocation Table

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 2GiB flash drive (with 4KiB 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 2GiB FAT volume will require resetting 2^{19} FAT entries, which will involve approximately 2^{21} bytes (2 MiB).

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.

Explain how an operating system reads a file like "`D:\My Files\Video.mp4`" from a FAT volume (from a software point of view).

> 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 a subdirectory named "`My Files`". If necessary, the root directory listing might occupy many blocks, and the operating system will follow the pointers in the file allocation table to scan through the entire root directory. Once the subdirectory entry for "`My Files`" is found, the operating system searches the subdirectory's listing for a file named "`Video.mp4`". Once it knows the block number for the file, it can begin reading the file sequentially by following the pointers in the file allocation table.

Compare bitmap-based allocation of blocks on disk with a free block list.

> Bitmap based block allocation is a fixed size proportional to the size of the disk. This means wasted space when the disk is full. A free block list shrinks as space is used up, so when the disk is full, the size of the free block list is tiny. However, contiguous allocation is easier to perform with a bitmap. Most modern file systems use a free block bitmap, not a free block list.

3 Inode-Based File System

1. What are the advantages of an inode-based file system design compared to FAT?

   Fast random access to files. Support for hard links.

2. What is the difference between a hard link and a soft link?

   Hard links point to the same inode, while soft links simply list a directory entry. Hard links use reference counting. Soft links do not and may have problems with dangling references if the referenced file is moved or deleted. Soft links can span file systems, while hard links are limited to the same file system.

3. Why do we have direct blocks? Why not just have indirect blocks?

   Faster for small files.

4. Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

   (a) How large of a disk can this file system support?

   2^{32} blocks x 2^{11} bytes/block = 2^{43} = 8 Terabytes.

   (b) What is the maximum file size?

   There are 512 pointers per block (i.e. 512 4-byte pointers in 2048 byte block), so:
   $blockSize \times (numDirect + numIndirect + numDoublyIndirect + numTriplyIndirect)$
   $2048 \times (12 + 512 + 512^2 + 512^3) = 2^{11} \times (2^2 \times 3 + 2^9 + 2^{9\times2} + 2^{9\times3})$
   $= 2^{13} \times 3 + 2^{20} + 2^{29} + 2^{38}$
   $= 24K + 513M + 256G$

5. Rather than writing updated files to disk immediately, many UNIX systems use a delayed *write-behind policy* in which dirty disk blocks are flushed to disk once every x seconds. List two advantages and one disadvantage of such a scheme.

   Advantage 1: The disk scheduling algorithm (i.e. SCAN) has more dirty blocks to work with at any one time and can thus do a better job of scheduling the disk arm.
   Advantage 2: Temporary files may be written and deleted before data is written to disk.
   Disadvantage: File data may be lost if the computer crashes before data is written to disk.

6. List the set of disk blocks that must be read into memory in order to read the file `/home/cs162/test.txt` in its entirety from a UNIX BSD 4.2 file system (10 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer). Assume the file is 15,234 bytes long and that disk blocks are 1024 bytes long. Assume that the directories in question all fit into a single disk block each. Note that this is not always true in reality.

   1. Read in file header for root (always at fixed spot on disk).
   2. Read in first data block for root (/).
   3. Read in file header for home.
   4. Read in data block for home.
   5. Read in file header for cs162.
   6. Read in data block for cs162.

7. Read in file header for test.txt.
8. Read in data block for test.txt.
9. - 17. Read in second through 10th data blocks for test.txt.
18. Read in indirect block pointed to by 11th entry in test.txt's file header.
19. - 23. Read in 11th – 15th test.txt data blocks. The 15th data block is partially full.

4 Extending an inode

Consider the following `inode_disk` struct, which is used on a disk with a 512 byte block size.

```
/* Definition of block_sector_t */
typedef uint32_t block_sector_t;
/* Contents of on-disk inode. Must be exactly 512 bytes long. */
struct inode_disk
  {
    off_t length;                       /* File size in bytes. */
    block_sector_t direct[12];          /* 12 direct pointers */
    block_sector_t indirect;            /* a singly indirect pointer */
    uint32_t unused[114];               /* Not used. */
  };
```

Why isn't the file name stored inside the `inode_disk` struct?

The file name belongs in the directory entry.

What is the maximum file size supported by this inode design?

It is $2^{16} + 12 \times 2^9$ bytes

How would you design the in-memory representation of the indirect block? (e.g. the disk sector that corresponds to an inode's `indirect` member)

You could use a `block_sector_t[128]`, which is exactly 512 bytes.

Implement the following function, which changes the size of an inode. If the resize operation fails, the inode should be unchanged and the function should return `false`. Use the value `0` for unallocated block pointers. You do not need to write the inode itself back to disk. You can use these functions:

- "`block_sector_t block_allocate()`" – Allocates a disk block and returns the sector number. If the disk is full, then returns 0. For this question, assume unlimited disk space.
- "`void block_free(block_sector_t n)`" – Free a disk block.
- "`void block_read(block_sector_t n, uint8_t buffer[512])`" – Reads the contents of a disk sector into a buffer.
- "`void block_write(block_sector_t n, uint8_t buffer[512])`" – Writes the contents of a buffer into a disk sector.

```
bool inode_resize(struct inode_disk *id, off_t size) {
  block_sector_t sector;  // A variable that may be useful.
  for (int i = 0; i < 12; i++) { // Handle direct pointers
    if (size <= 512 * i && id->direct[i] != 0) { // Shrink inode
      block_free(id->direct[i]);
      id->direct[i] = 0;
    }
    if (size > 512 * i && id->direct[i] == 0) { // Grow inode
      id->direct[i] = allocate_block();
    }
  }
  if (id->indirect == 0 && size <= 12 * 512) { // Check to handle indirects
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  memset(buffer, 0, 512);
  if (id->indirect == 0) { // Allocate indirect page if does not exist
    id->indirect = allocate_block();
  } else {
    block_read(id->indirect, buffer);
  }
  for (int i = 0; i < 128; i++) { // Handle indirect pointers
    if (size <= (12 + i) * 512 && buffer[i] != 0) { // Shrink inode
      block_free(buffer[i]);
      buffer[i] = 0;
    }
    if (size > (12 + i) * 512 && buffer[i] == 0) { // Grow inode
      buffer[i] = allocate_block();
    }
  }
  if (id->indirect != 0 && size <= 12 * 512) {
    block_free(id->indirect);
    id->indirect = 0;
  } else {
    block_write(id->indirect, buffer);
  }
  id->length = size;
  return true;
}
```

Solution that includes error handling (limited disk space)

```
bool inode_resize(struct inode_disk *id, off_t size) {
  block_sector_t sector;
  for (int i = 0; i < 12; i++) {
    if (size <= 512 * i && id->direct[i] != 0) {
      block_free(id->direct[i]);
      id->direct[i] = 0;
    }
    if (size > 512 * i && id->direct[i] == 0) {
      sector = allocate_block();
      if (sector == 0) {
        inode_resize(id, id->length);
        return false;
      }
      id->direct[i] = sector;
    }
  }
  if (id->indirect == 0 && size <= 12 * 512) {
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  if (id->indirect == 0) {
    memset(buffer, 0, 512);
    sector = allocate_block();
    if (sector == 0) {
      inode_resize(id, id->length);
      return false;
    }
    id->indirect = sector;
  } else {
    block_read(id->indirect, buffer);
  }
  for (int i = 0; i < 128; i++) {
    if (size <= (12 + i) * 512 && buffer[i] != 0) {
      block_free(buffer[i]);
      buffer[i] = 0;
    }
    if (size > (12 + i) * 512 && buffer[i] == 0) {
      sector = allocate_block();
      if (sector == 0) { // Handle failure
        inode_resize(id, id->length);
        return false;
      }
      buffer[i] = sector;
    }
  }
  if (id->indirect != 0 && size <= 12 * 512) {
    block_free(id->indirect);
```

```
    id->indirect = 0;
  } else {
    block_write(id->indirect, buffer);
  }
  id->length = size;
  return true;
}
```

Another solution that you may find useful

```
#include <stdio.h>
#include <unistd.h>
typedef uint32_t block_sector_t;
struct inode_disk
{
    off_t length; /* File size in bytes. */
    block_sector_t pointers[13]; /* 12 direct pointers and 1 indirect pointer*/
    uint32_t unused[114]; /* Not used. */
};
struct indirect_disk
{
    block_sector_t pointers[128];
}
bool calculate_indices(int blocknumber, int *offsets, int *offset_cnt)
{
    if (sector_idx < 12)
    {
        offsets[0] = sector_idx;
        *offset_cnt = 1;
        return true;
    }
    sector_idx -= 12;
    if (sector_idx < PTRS_PER_SECTOR)
    {
        offsets[0] = 12;
        offsets[1] = sector_idx % PTRS_PER_SECTOR;
        *offset_cnt = 2;
        return true;
    }
    return false;
}
bool inode_change_block(struct inode_disk *id, block_sector_t block, bool add)
{
    int offsets[2];
    int offset_cnt;
    int i = 0;
    uint8_t zeros[512];
    struct indirect_disk cur;
    block_sector_t cur_indirect;
    memset(zeros, 0, sizeof(zeros));
    calculate_indices(block, offsets, &offset_cnt);
```

```
    for (i = 0; i < offset_cnt; i++)
    {
        if (i == 0)
        {
            if (add && id->pointers[offsets[0]] == 0)
            {
                block_sector_t next_indirect;
                if ((next_indirect = block_allocate()) == 0)
                    return false;
                id->pointers[offsets[0]] = next_indirect;
                block_write(next_indirect, zeros);
                cur_indirect = next_indirect;
            }
            if (!add && ((offset_cnt == 1) || (offset_cnt == 2 && offsets[1] == 0)))
            {
                block_free(id->pointers[offsets[0]]);
                id->pointers[offsets[0]] = 0;
            }
        }
        else
        {
            block_read(cur_indirect, &cur);
            if (add && cur.pointers[offsets[1]] == 0)
            {
                block_sector_t next_indirect;
                if ((next_indirect = block_allocate()) == 0)
                    return false;
                cur.pointers[offsets[1]] = next_indirect;
                block_write(next_indirect, zeros);
                block_write(cur_indirect, &cur);
                cur_indirect = next_indirect;
            }
            if (!add)
            {
                block_free(cur.pointers[offsets[1]]);
                cur.pointers[offsets[1]] = 0;
                block_write(cur_indirect, &cur);
            }
        }
    }
}
bool inode_resize(struct inode_disk *id, size_t size)
{
    size_t cur_blocks;
    size_t new_blocks;
    off_t cur;
    off_t d_cur;
    cur_blocks = id->length/BLOCK_SIZE;
    new_blocks = size/BLOCK_SIZE;
    if (new_blocks > cur_blocks)
```

```
    {
        //allocate blocks from [cur_blocks+1, new_blocks];
        for (cur = cur_blocks + 1; cur <= new_blocks; cur++)
        {
            if (!inode_change_block(id, cur, true))
            {
                // must deallocate if failed to allocate
                for (d_cur = cur_blocks + 1; d_cur < cur; d_cur++)
                {
                    inode_change_block(id, d_cur, false);
                }
                return false;
            }
        }
        id->length = size;
        return true;
    }
    else if (cur_blocks > new_blocks)
    {
    //deallocate blocks from [new_blocks+1, cur_blocks];
        for (cur = new_blocks + 1; cur <= cur_blocks; cur++)
            inode_change_block(id, d_cur, false);
        id->length = size;
        return true
    }
    else
    {
        id->length = size;
    }
}
```