# File Management

- Files can be viewed as either:
  - a sequence of bytes with no structure imposed by the operating system.
  - or a structured collection of information with some structure imposed by the operating system.
- Unix, Linux, Windows, Mac OS X all treat files as a sequence of bytes. This model is the most flexible, leaving the structure of the file up to the application programs.

# File Types in Linux

- Regular files which include text files and binary files.
  - Text files (formatted).
  - Binary files (unformatted). E.g. executable files, archive files, shared libraries etc.
- Directory files.
- Device special files (representing block and character devices).
- Links (hard and symbolic links or shortcuts).
- FIFO (named pipes), sockets.

The `file` command in Linux/MacOSX identifies the type of a file.

# File Attributes and Operations

*Examples of typical attributes*: owner, creator, creation time, time of last access, time last modified, current size, read-only flag, archive flag, hidden flag, lock flags, temporary flags, reference count.

*Examples of typical operations*: create, delete, open, close, read, write, append, seek, get attributes, set attributes, rename etc.

# Obtaining File Attributes

```
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);

// the following structure is returned
struct stat
{
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    umode_t       st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

See example program `file-management/mystat.c`.

# Directories

- Directories can be used to organize files into
  - Tree Structures
  - Directed Acyclic Graphs
  - Arbitrary Graphs (though cycles would cause problems!)
- System calls dealing with directories: `opendir()`, `readdir()`, `closedir()`, `rewinddir()` etc.
- Differences between hard links and symbolic links. See example in the `file-management` folder in the examples.
- How do different utility programs and the system deal with loops in the file system? try the following commands in the `file-management` examples folder (where `d1` is a folder with a cyclic structure):

```
tar cvf d1.tar d1
zip -r d1.zip d1
```

# Structure of directory files in Linux

A directory file consists of several entries of the following type.

```
struct dirent
{
    long d_ino;                 /* inode number */
    off_t d_off;                /* offset to next dirent */
    unsigned short d_reclen;    /* length of this dirent */
    char d_name [NAME_MAX+1];   /* file name (null-terminated) */
}
```

The *inode* is the index node for a file/directory that allows the system to find the file on the disk drive.

# A Bare-bones Implementation of `ls`

```
/* file-management/myls.c */
#include        <sys/types.h>
#include        <dirent.h>
#include        "ourhdr.h"

int main(int argc, char *argv[])
{
   DIR   *dp;
   struct dirent   *dirp;

   if (argc != 2)
        err_quit("a single argument (the directory name) is required");

    if ( (dp = opendir(argv[1])) == NULL)
            err_sys("can't open %s", argv[1]);

    while ( (dirp = readdir(dp)) != NULL)
            printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

# Recursive File Traversal

Recursive traversal in the directory hierarchy is useful for many system utilities. Here are some examples:

```
du
find / -name "core" -exec /bin/rm -f '{}' ';'
tar cf /tmp/home.tar ~
tar cf - old | (cd /tmp; tar xf -)
cp -r old /tmp
ls -R
chmod -R  g-r,o-r  ~/cs453
```

# An Example of File System Traversal

```
/* file-management/filetype-survey.c */
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    "ourhdr.h"
typedef int Myfunc(const char *, const struct stat *, int);
                /* function type that's called for each filename */
static Myfunc   myfunc;
static int      myftw(char *, Myfunc *);
static int      dopath(Myfunc *);
static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int main(int argc, char *argv[])
{
    int     ret;
    if (argc != 2) err_quit("usage:  ftw  <starting-pathname>");
    ret = myftw(argv[1], myfunc);       /* does it all */

    if ( (ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock) == 0)
        ntot = 1;       /* avoid divide by 0; print 0 for all counts */
    printf("regular files  = %7ld, %5.2f %%\n", nreg,  nreg*100.0/ntot);
    printf("directories    = %7ld, %5.2f %%\n", ndir,  ndir*100.0/ntot);
    printf("block special  = %7ld, %5.2f %%\n", nblk,  nblk*100.0/ntot);
    printf("char special   = %7ld, %5.2f %%\n", nchr,  nchr*100.0/ntot);
    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo, nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,nslink*100.0/ntot);
    printf("sockets        = %7ld, %5.2f %%\n", nsock, nsock*100.0/ntot);
    exit(ret);
}
```

```
/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */

#define FTW_F   1       /* file other than directory */
#define FTW_D   2       /* directory */
#define FTW_DNR 3       /* directory that can't be read */
#define FTW_NS  4       /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(NULL);    /* malloc's for PATH_MAX+1 bytes */
                                    /* ({Prog pathalloc}) */
    strcpy(fullpath, pathname);     /* initialize fullpath */

    return(dopath(func));
}
```

```
/* Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return.  For a directory, we call ourself
 * recursively for each name in the directory.  */
static int dopath(Myfunc* func) {
    struct stat     statbuf;
    struct dirent   *dirp;
    DIR             *dp;
    int             ret;
    char            *ptr;

    if (lstat(fullpath, &statbuf) < 0)
        return(func(fullpath, &statbuf, FTW_NS));   /* stat error */
    if (S_ISDIR(statbuf.st_mode) == 0)
        return(func(fullpath, &statbuf, FTW_F));    /* not a directory */
    // It's a directory if we get here.
    if ( (ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);
    ptr = fullpath + strlen(fullpath);  /* point to end of fullpath */
    *ptr++ = '/';
    *ptr = 0;

    if ( (dp = opendir(fullpath)) == NULL)
        return(func(fullpath, &statbuf, FTW_DNR));
    while ( (dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0  ||
            strcmp(dirp->d_name, "..") == 0)
                continue;       /* ignore dot and dot-dot */
        strcpy(ptr, dirp->d_name);  /* append name after slash */
        if ( (ret = dopath(func)) != 0)     /* recursive */
            break;  /* time to leave */
    }
    ptr[-1] = 0;    /* erase everything from slash onwards */
    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);
    return(ret);
}
```

```
static int myfunc(const char *pathname, const struct stat *statptr, int type) {
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;     break;
        case S_IFBLK:   nblk++;     break;
        case S_IFCHR:   nchr++;     break;
        case S_IFIFO:   nfifo++;    break;
        case S_IFLNK:   nslink++;   break;
        case S_IFSOCK:  nsock++;    break;
        case S_IFDIR:
            err_dump("for S_IFDIR for %s", pathname);
            /* directories should have type = FTW_D */
        }
        break;
    case FTW_D:
        ndir++;
        break;
    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
        break;
    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;
    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }
    return(0);
}
```

# Experiments with Recursive File Traversal

- Run the program `file-management/filetype-survey` on your home folder or (as root, on your entire system!).
- Run the program `file-management/filesize-survey` on your home folder or on your entire system. See results for onyx in the file `file-management/filesize-survey.txt`. Note that we found that 85% of the files were 12 blocks or smaller!

# File I/O Buffering

- unbuffered. (I/O occurs as each character is encountered).
- line-buffered. (I/O occurs when a newline is encountered).
- fully-buffered. (I/O occurs when the buffer fills up.)

- The streams `stdin`, `stdout` are usually line-buffered (if they refer to a terminal device), otherwise they are fully buffered. Standard error (`stderr`) is unbuffered.
- Under Linux/MacOS X, the system call `setvbuf(...)` can be used to change the buffering behavior of an I/O stream.

# Memory Mapped Files

A file can be mapped to a region in the virtual address space. The file serves as a backing store for that region of memory. Read/writes cause page faults and when the process terminates all mapped, modified pages are written back to the file on the disk.
Some issues:

- what if the file is larger than the virtual address space?
- what if another process opens the file for a normal read?
- unmapping the mapped file does not cause the modified pages to be written back immediately. The updating happens at periodic intervals by the virtual memory subsystem or can be forced by the system call `msync()`.
- But memory mapped file is a convenient way to provide shared memory between processes.

```
//.. appropriate header files

void  *mmap(void  *start,  size_t length, int prot , int flags,
            int fd, off_t offset);

int munmap(void *start, size_t length);
```

# Comparison of file I/O vs. memory-mapped file I/O

| | user | system | elapsed time |
|---|---|---|---|
| memory-mapped I/O | 2.85s | 0.01s | 2.96s |
| file I/O | 0.68s | 14.39s | 15.17s |

- The two programs manipulate a file where several chunks are read and written over repeatedly. The file I/O program does the reads/writes directly on file whereas the memory-mapped program maps the file to memory first and then performs the reads/writes in memory.
- The file I/O was done with a buffer size of 4096 bytes.
- All times are in seconds.

```
/* Direct file I/O program: file-management/file-io.c */
#define BUFSIZE 2048
int main(int argc, char *argv[]) {
    int         i, fdin;
    struct stat statbuf;
    char *buf1, *buf2;
    long size;

    if (argc != 2) err_quit("usage: %s.out <datefile> ", argv[0]);
    if ( (fdin = open(argv[1], O_RDWR)) < 0)
        err_sys("can't open %s for reading/writing", argv[1]);
    if (fstat(fdin, &statbuf) < 0)  /* need size of input file */
        err_sys("fstat error");

    size = statbuf.st_size;
    buf1 = (char *) malloc(sizeof(char)*BUFSIZE);
    buf2 = (char *) malloc(sizeof(char)*BUFSIZE);
    for (i=0; i<BUFSIZE-1; i++)
            buf2[i] = 'e';
    buf2[BUFSIZE-1] = '\n';
    for (i=0; i<1000000; i++) {
        long offset = random() %  (size - BUFSIZE);
        lseek(fdin, offset, SEEK_SET);
        read(fdin, buf1, BUFSIZE);
        write(fdin, buf2, BUFSIZE);
    }
    close(fdin);
    exit(0);
}
```

```
/* Memory-mapped file I/O program: file-management/mmap-io.c
   Use file-management/mem.data as data file  */
#define BUFSIZE 2048
int main(int argc, char *argv[]) {
    int         i, fdin;
    struct stat statbuf;
    char *src, *buf1, *buf2;
    long size;
    if (argc != 2) err_quit("usage: %s.out <datefile> ", argv[0]);
    if ( (fdin = open(argv[1], O_RDWR)) < 0)
        err_sys("can't open %s for reading/writing", argv[1]);
    if (fstat(fdin, &statbuf) < 0)  err_sys("fstat error");
    if ( (src = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
                     MAP_FILE | MAP_SHARED, fdin, 0)) == (caddr_t) -1)
        err_sys("mmap error for input");
    size = statbuf.st_size;
    buf1 = (char *) malloc(sizeof(char)*BUFSIZE);
    buf2 = (char *) malloc(sizeof(char)*BUFSIZE);
    for (i=0; i<BUFSIZE-1; i++)
            buf2[i] = 'e';
    buf2[BUFSIZE-1] = '\n';

    for (i=0; i<1000000; i++) {
        long offset = random() %  (size - BUFSIZE);
        memcpy(buf1, src+offset, BUFSIZE);
        memcpy(src+offset, buf2, BUFSIZE);
    }
    msync(src, size, MS_SYNC);  /* force flushing data to disk */
    exit(0);
}
```

# Low Level File Implementation

- `open()`, `close()` system calls.
- Descriptor Tables (one per process) and File Table (global).
- How are file descriptors allocated? Implementing I/O redirection.

# Kernel data structures for File I/O

- File descriptor table. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, with one entry per descriptor. A *descriptor* has
  - file descriptor flags.
  - pointer to a File Table entry.
- File Table. The kernel maintains a file table for all open files. Each file table entry may contain:
  - file status flags (read, write, append, etc.)
  - current file offset.
  - a pointer to the *v-node* (virtual-node) table entry for the file.
  - Note that child processes process table entries point to the same node in the file table for the same open file (if file was opened before fork)!
  - Unrelated processes will point to separate entries in the file table if they open the same file.

- V-node. The v-node contains information about the type of file and pointers to functions that operate on the file. For most files, the v-node also contains the *i-node* (index-node) for the file.
- I-node. The i-node contains information about file attributes and how to find the file on the disk.

# Disk Space Management

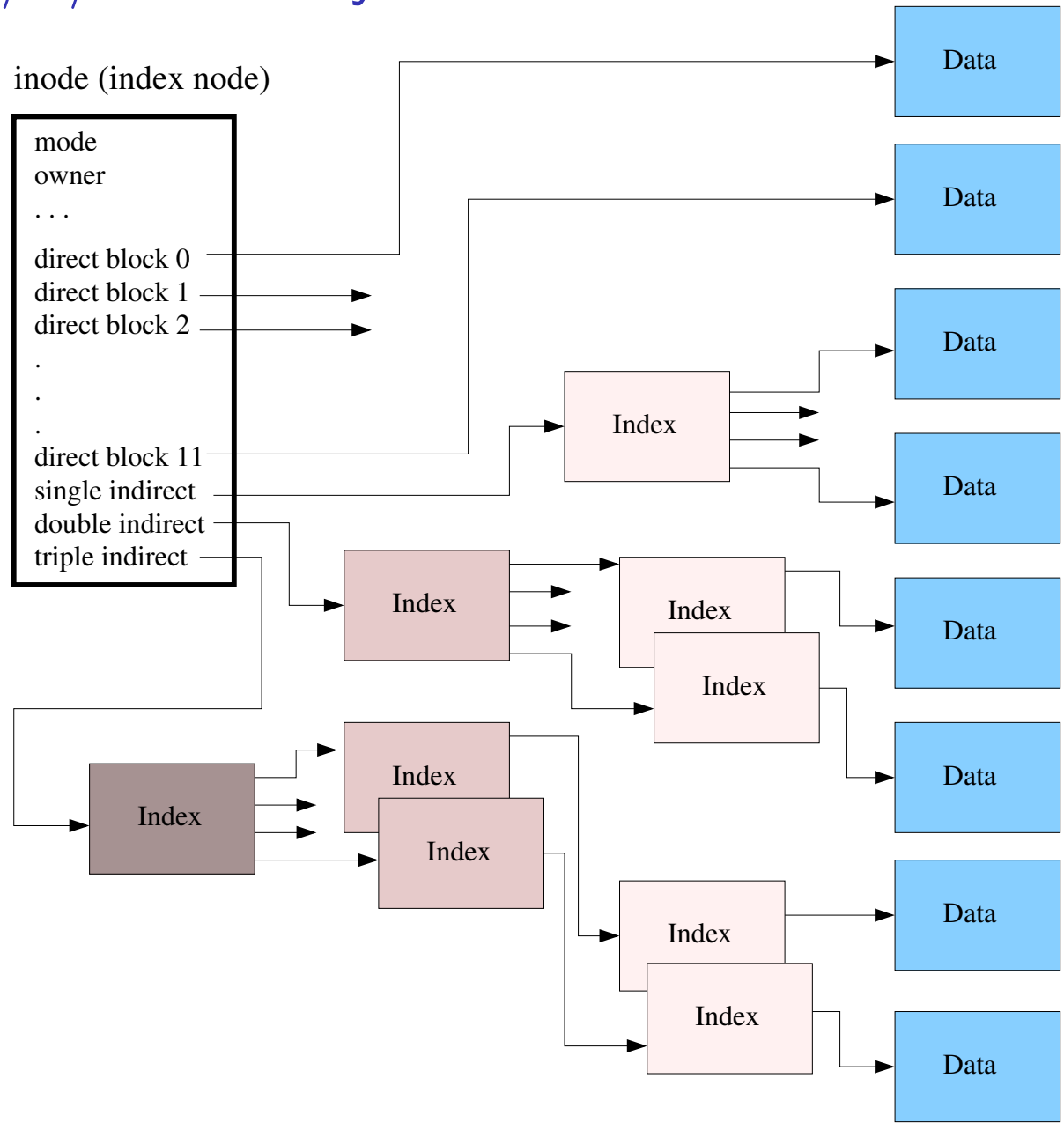
# Keeping Track of Allocated Blocks

- Contiguous allocation.
- Linked list allocation (Example: Microsoft FAT file systems).
- Indexed Allocation (Example: Linux Ext2/Ext3/Ext4, Unix System V filesystem etc.)

Note that we can examine the Linux kernel code in the `fs` folder to see how it reads/writes various file systems.

We can also figure out the filesystem layouts by looking at appropriate header files. For example, check out `include/linux/ext2_fs.h` in the kernel source for Linux ext2/3/4 file system layouts.

# Linux Ext2/3/4 File Layout



# Linux Indexed Allocation Example Calculation

Example: Filesystem block size = 1KB
inode has 12 direct pointers, 1 indirect, 1 double indirect, 1 triple indirect pointer.
Each pointer is 32 bits or 4 bytes. So we can store 256 pointers in one block.

The following shows maximum file sizes that can be represented using indexed file allocation for the above example.

- Using only direct pointers: $12 \times 1\text{KB} = 12\text{KB}$
- Using direct and single indirect pointers: $256 \times 1\text{KB} + 12\text{KB} = 268\text{KB}$
- Using direct, single and double indirect pointers:
  $256 \times 256 \times 1\text{KB} + 256\text{KB} + 12\text{KB} = 65536\text{KB} + 268\text{KB} = 65804\text{KB} \cong 64\text{MB}$
- Using direct, single, double, and triple indirect pointers:
  $256 \times 256 \times 256 \times 1\text{KB} + 65536\text{KB} + 256\text{KB} + 12\text{KB} =$
  $16777216\text{KB} + 65804\text{KB} = 16843020\text{KB} \cong 16\text{GB}$

# Dealing with Unallocated Blocks

- Using block status bitmap. One bit per block to keep track of whether the block is allocated/unallocated. Makes it easy to get a quick snapshot of the blocks on the disk.
- File systems integrity check (`fsck` command under Linux). The block status bitmap makes some kinds of integrity check easier to accomplish.

# Example File System Implementations

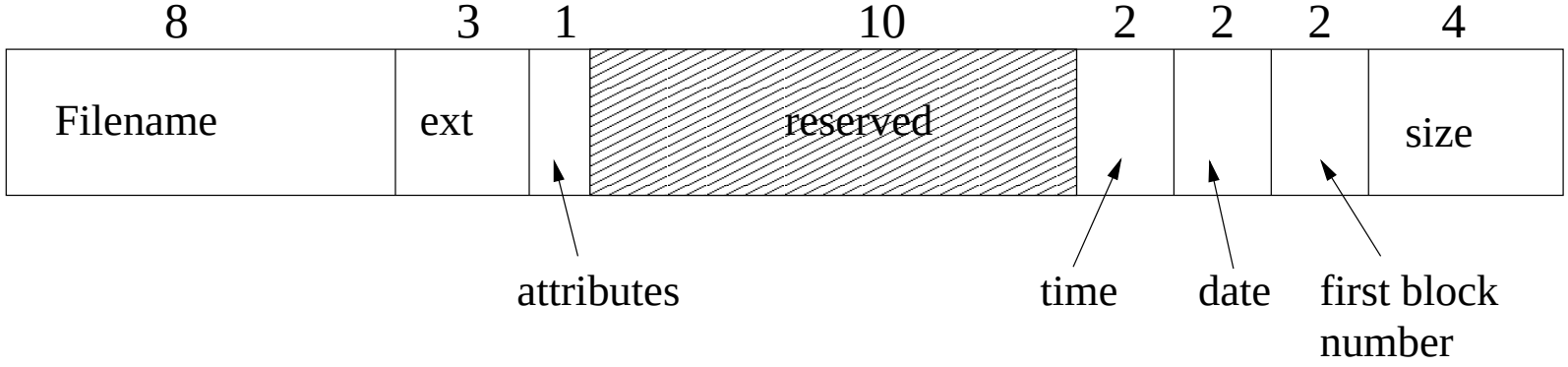
- Microsoft FAT-12, FAT-16, FAT-32 filesystems.
- Microsoft Windows 98 filesystem.
- Linux Ext2/Ext3/Ext4 filesystem.
- Other Journaling filesystems.

# FAT File System

- The original FAT file system allows file names to be 8+3 characters long (all uppercase).
- Hierarchical directory structure with arbitrary depth. However, root directory had a fixed maximum size.
- File system was a tree (that is, links like in Unix were not permitted).
- Any user has access to all files.

# FAT File System (continued)

- Each directory entry is represented by 32 bytes.

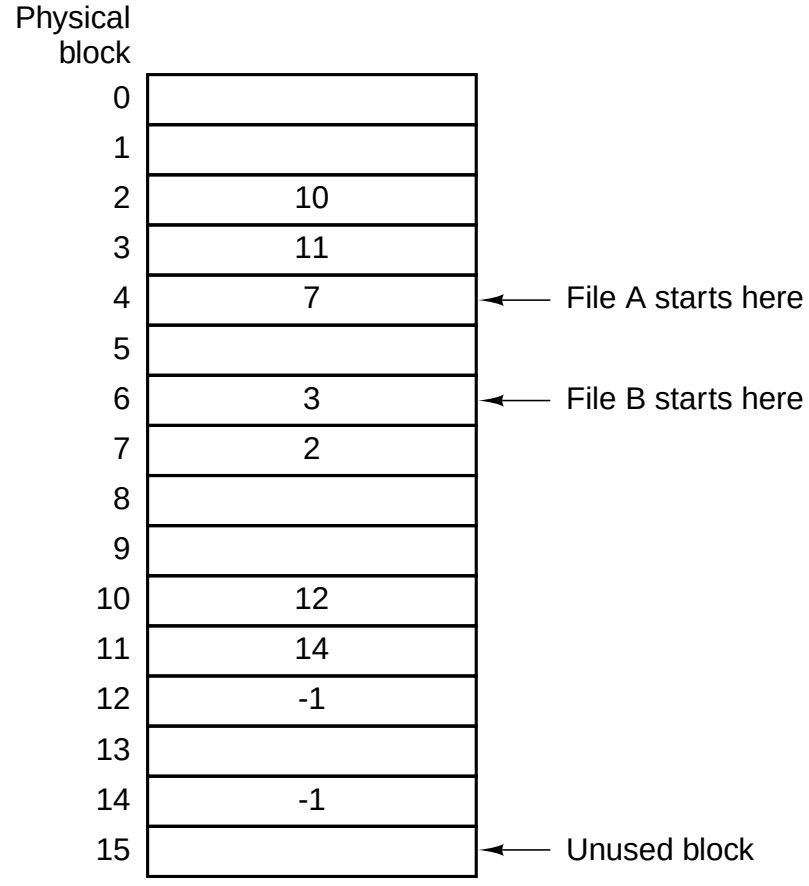



Directory Entry in FAT File System

- The time field has 5 bits for seconds, 6 bits for minutes, 5 bits for hours. The date field has 5 bits for day, 4 bits for month, and 7 bits for year (kept in years since 1980...so we have a year 2107 problem here).
- FAT-12, FAT-16 and FAT-32 are the same file systems but with different address size that are supported. Actually, FAT-32 supports 28 bits disk address.

# An Example File Allocation Table

File blocks are kept track using a File Allocation Table (FAT) in main memory. The first block number from the directory entry is used as an index into the 64K entry FAT table that represents all files via linked lists in the FAT table.



# FAT File System (continued)

- FAT file system supported four partitions per disk.
- Disk block size can be some multiple of 512 bytes (also known as *cluster size*).

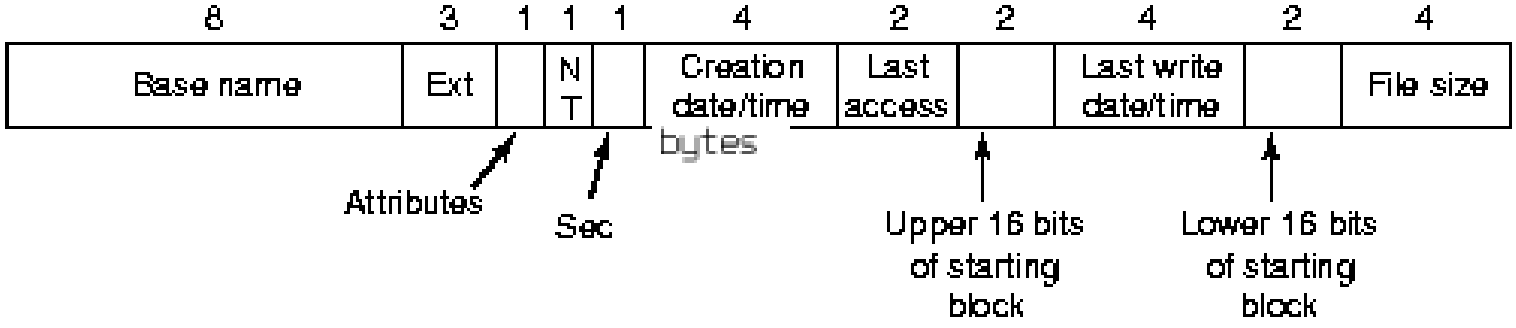
| Block size | FAT-12 | FAT-16 | FAT-32 |
|---|---|---|---|
| 0.5 KB | 2 MB | | |
| 1 KB | 4 MB | | |
| 2 KB | 8 MB | 128 MB | |
| 4 KB | 16 MB | 256 MB | 1 TB |
| 8 KB | | 512 MB | 2 TB |
| 16 KB | | 1024 MB | 2 TB |
| 32 KB | | 2048 MB | 2 TB |

# FAT File System (continued)

- The FAT file system keeps track of free blocks by marking the entries in the File Allocation Table with a special code. To find a free block, the system searches the FAT until it finds a free block.
- The FAT table occupies a significant amount of main memory. With FAT-32, the File Allocation Table was no longer limited to 64K entries because otherwise the file block size had to be quite big. Now to represent a 2GB partition with 4KB block size, there are 512K blocks that require 2MB of memory to store the table.

# Windows 98 Filesystem

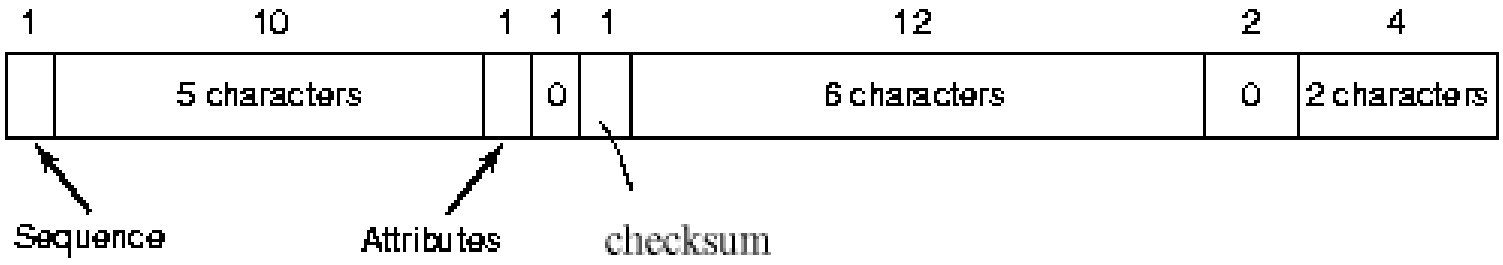
- Allow long file names.
- Remain forward/backward compatible with older versions of Microsoft Windows operating system using FAT file systems.
- Directory entry structure.




- The directory entry is designed such that older versions of the operating system can still read it. In order to do so, Windows 98 file system first converts long names into 8+3 names by converting the first six characters to upper case and then appending $\sim n$ at the end (where n is 1,2, ... as needed to make the names be unique).
- The long names are stored in long-name entries before the actual directory entry. These are directory entries with their attributes marked 0, so older systems would just ignore these entries.

# Windows 98 Filesystem (continued)

Structure of a long-name entry.



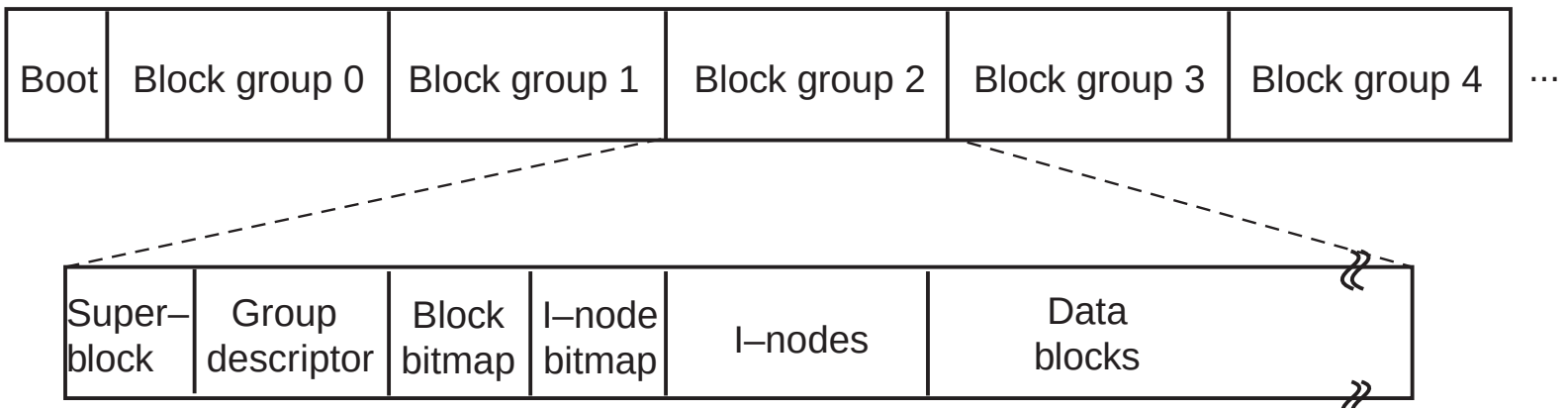

Representation of the file name `"The quick brown fox jumps over the lazy dog"`

| Bytes | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 68 | d | | | o | | | g | | | | A | 0 | CK | | | | | | | | | | | | | | | 0 | | | |
| 3 | o | | | v | | | e | | | | A | 0 | CK | t | | | h | | | e | | | l | | | a | 0 | z | | | y |
| 2 | w | | | n | | | | f | | o | A | 0 | CK | x | | | | j | | u | | m | | p | | | 0 | s | | | |
| 1 | T | | | h | | | e | | | q | A | 0 | CK | u | | | i | | | c | | | k | | | b | 0 | r | | | o |
| T | H | E | Q | U | I | ~ | 1 | | | | A | NT | S | Creation time | | | | | Last acc | | Upp | | | Last write | | | | Low | Size | | | |

# Windows 98 Filesystem (continued)

- Why is the checksum field needed in long entries? To detect consistency problems introduced by older system manipulating the file system using only the short name.
- The Low field being zero provides additional check.
- There is no limitation on the size of the File Allocation Table. So the system maintains a window into the table, rather than storing the entire table in memory.
- Disk Layout: boot sector, FAT tables (usually replicated at least two times), root directory, other directories, data blocks.

# Linux Ext2 Filesystem



# Linux Ext2 Filesystem

- The disk is divided into groups of blocks. Each block group contains:
  - **Superblock**. Each block group starts with a superblock, which tells how many blocks and i-nodes there are, gives the block size etc.
  - **Group descriptor**. The group descriptor contains information about the location of the bitmaps, number of free blocks and i-nodes in the group and the number of directories in the group.
  - **Block and I-node Bitmaps**. These two bitmaps keep track of free blocks and free i-nodes. Each map is one block long. With a 1 KB block, this limits the number of blocks and i-nodes to 8192 in each block group.
  - **I-nodes**. Each i-node is 128 bytes long (which is double the size for standard Unix i-nodes). There are 12 direct addresses and 3 indirect addresses. Each address is 4 bytes long.
  - **Data nodes**.
- Directories are spread evenly over the disk.
- When a file grows, the new block is placed in the same group as the rest of the file, preferably right after the previous last block. When a new file is added to a directory, Ext2 tries to put it into same block group as the directory.

# Linux Ext3 Journaling Filesystem

The goal of a *journaling filesystem* is to avoid running time-consuming consistency checks on the whole filesystem by looking instead in a special disk area that contains the most recent disk write operations named *journal*.
Remounting a journaling filesystem after a system failure is a matter of a few seconds.

- Ext3 file system is a journaling filesystem that is compatible with the ext2 filesystem.
- An Ext3 filesystem that has been cleanly unmounted can be remounted as an Ext2 filesystem.
- Creating a journal for an existing Ext2 filesystem and remounting as an Ext3 filesystem is a simple, fast operation.

# Linux Ext3 Journaling Filesystem (contd)

Ext3 filesystem allows administrator to decide whether to log just metadata or to log data blocks as well (at a performance penalty). It supports three modes:

- *Journal*: All filesystem data and metadata changes are logged into the journal. Safest and the slowest mode.
- *Ordered*: Only changes to the filesystem metadata are written to the journal. However, groups metadata and data blocks are written to disk before the filesystem metadata. This is the default mode.
- *Writeback*: Only changes to the filesystem metadata are written to the journal. Fastest mode. Also used in other journaling filesystems.

# Other Journaling Filesystems (contd)

- Ext4. Supports larger filesystems and large files compared to ext2/ext3. Uses HTree (a type of BTree) by default for directory indexing. Backwards compatible with ext2/ext3.
- Microsoft's NTFS. Uses B+-Trees to improve performance and disk space utlization. Also includes better access control using Access Control Lists (ACL).
- ResierFS (Version 3 is widely used,and has faster performance on large files, Version 4 claims to have the fastest performance in general usage as well)
- IBM's JFS. Good performance for small and large files.
- Apple's HFS+. Uses a B-Tree for storing metadata.

# Experimenting with File Systems

- Using loopback devices in Linux, we can build different file systems inside regular files. We can even mount them and experiment them.
- First create a file (of, say, 100MB size) filled by zeroes.
  ```
  dd if=/dev/zero of=pseudo-device.img bs=1024
  count=1048576
  ```
- Then make a file system (of, say, type ext3) inside that file.
  ```
  mkfs -v -t ext3 pseudo-device.img
  ```
- Then we can mount it over a directory using a loop back device as follows. You have to be superuser for this step and the directory `/mnt/testfs` must exist.
  ```
  sudo mount -o loop pseudo-device.img /mnt/testfs
  ```
- When you are done with it, you can umount it as follows.
  ```
  sudo umount /mnt/testfs
  ```
- If we create the image file with a large hole in it, then the space usage can grow as needed. See the example `file-management/create-empty-file.c`