Today

- Today's Learning Outcomes
  - Define superblock.
  - Use system calls to:
    - Read file system metadata
    - Read file metadata
    - Read directory entries
- Code that you can use
  - In the course repo (under nov13).
- Readings
  - 10.6, 10.7

Examining a File System's Metadata

- POSIX* provides two APIs to obtain information about a (mounted) system.
  - `int statfs(const char *path, struct statfs *buf);`
  - `int fstatfs(int fd, struct statfs *buf);`
- Returns (in the passed structure) metadata about the file system in which the object represented by `path` (`fd`) appears.
  - `statfs` takes a file name.
  - `fstatfs` takes the `fd` for an open file.

* POSIX is the standard that describes what we think of as UNIX-like operating systems (e.g., Linux)

Struct statfs

```
struct statfs {
       __fsword_t f_type;    /* Type of filesystem (see below) */
       __fsword_t f_bsize;   /* Optimal transfer block size */
       fsblkcnt_t f_blocks;  /* Total data blocks in filesystem */
       fsblkcnt_t f_bfree;   /* Free blocks in filesystem */
       fsblkcnt_t f_bavail;  /* Free blocks available to unprivileged user */
       fsfilcnt_t f_files;   /* Total file nodes in filesystem */
       fsfilcnt_t f_ffree;   /* Free file nodes in filesystem */
       fsid_t     f_fsid;    /* Filesystem ID */
       __fsword_t f_namelen; /* Maximum length of filenames */
       __fsword_t f_frsize;  /* Fragment size (since Linux 2.6) */
       __fsword_t f_flags;   /* Mount flags of filesystem(since Linux 2.6.36) */
       __fsword_t f_spare[xxx]; /* Padding bytes reserved for future use */
};
```

- File system types are things like: ext2, fat, ext4, etc.

Whole File System Metadata

- `statfs` works well if the file system is mounted, but what if I hand you a disk, how does the operating system figure out how to interpret it?

Whole File System Metadata

- `statfs` works well if the file system is mounted, but what if I hand you a disk, how does the operating system figure out how to interpret it?
  - A file system typically begins with a single sector that contains information (metadata) about the file system.
  - We call the structure in this block a superblock.
- Many file systems will replicate the superblock many times in different places in the file system. Why?

EXT2 Metadata

- `struct ext2_super_block`
  - number of inodes
  - number of blocks
  - number of free inodes/blocks
- Ext2 breaks the disks up into groups (allows placing a file's blocks 'close' to each other) -- the superblock describes these groups:
  - s_blocks_per_group
  - s_inodes_per_group
  - s_log_block_size

```
struct ext2_super_block {
        __u32   s_inodes_count;         /* Inodes count */
        __u32   s_blocks_count;         /* Blocks count */
        __u32   s_r_blocks_count;       /* Reserved blocks count */
        __u32   s_free_blocks_count;    /* Free blocks count */
        __u32   s_free_inodes_count;    /* Free inodes count */
        __u32   s_first_data_block;     /* First Data Block */
        __u32   s_log_block_size;       /* Block size */
        __s32   s_log_frag_size;        /* Fragment size */
        __u32   s_blocks_per_group;     /* # Blocks per group */
        __u32   s_frags_per_group;      /* # Fragments per group */
        __u32   s_inodes_per_group;     /* # Inodes per group */
        __u32   s_mtime;                /* Mount time */
        __u32   s_wtime;                /* Write time */
        __u16   s_mnt_count;            /* Mount count */
        __s16   s_max_mnt_count;        /* Maximal mount count */
        __u16   s_magic;                /* Magic signature */
        __u16   s_state;                /* File system state */
        __u16   s_errors;               /* Behaviour when detecting errors */
        __u16   s_minor_rev_level;      /* minor revision level */
        __u32   s_lastcheck;            /* time of last check */
        __u32   s_checkinterval;        /* max. time between checks */
        __u32   s_creator_os;           /* OS */
        __u32   s_rev_level;            /* Revision level */
        __u16   s_def_resuid;           /* Default uid for reserved blocks */
        __u16   s_def_resgid;           /* Default gid for reserved blocks */
        /*
         * These fields are for EXT2_DYNAMIC_REV superblocks only.
         *
         * Note: the difference between the compatible feature set and
         * the incompatible feature set is that if there is a bit set
         * in the incompatible feature set that the kernel doesn't
         * know about, it should refuse to mount the filesystem.
         *
         * e2fsck's requirements are more strict; if it doesn't know
         * about a feature in either the compatible or incompatible
         * feature set, it must abort and not try to meddle with
         * things it doesn't understand...
         */
        __u32   s_first_ino;            /* First non-reserved inode */
        __u16   s_inode_size;           /* size of inode structure */
        __u16   s_block_group_nr;       /* block group # of this superblock */
        __u32   s_feature_compat;       /* compatible feature set */
        __u32   s_feature_incompat;     /* incompatible feature set */
        __u32   s_feature_ro_compat;    /* readonly-compatible feature set */
        __u32   s_reserved[230];        /* Padding to the end of the block */
};
```

Examining a File's metadata

- POSIX* provides two** APIs to obtain information about a file's metadata.
  - `int stat(const char *restrict path, struct stat *restrict buf);`
  - `int fstat(ind fd, struct stat *buf);`

- Returns (in the passed structure) the metadata for a file.
  - `stat` lets you access a file by name.
  - `fstat` lets you access the file metadata by file descriptor.

* POSIX is the standard that describes what we think of as UNIX-like operating systems (e.g., Linux)
** Actually more than two, but several are variants of the two I'll introduce; feel free to use man pages to explore.

Let's look at the stat structure

```
struct stat {
        dev_t    st_dev;    /* device inode resides on */
        ino_t    st_ino;    /* inode's number */
        mode_t   st_mode;   /* inode protection mode */
        nlink_t  st_nlink;  /* number of hard links to the file */
        uid_t    st_uid;    /* user-id of owner */
        gid_t    st_gid;    /* group-id of owner */
        dev_t    st_rdev;   /* device type, for special file inode */
        struct timespec st_atimespec;  /* time of last access */
        struct timespec st_mtimespec;  /* time of last data modification */
        struct timespec st_ctimespec;  /* time of last file status change */
        off_t    st_size;   /* file size, in bytes */
        quad_t   st_blocks; /* blocks allocated for file */
        u_long   st_blksize;/* optimal file sys I/O ops blocksize */
        u_long   st_flags;  /* user defined flags for file */
        u_long   st_gen;    /* file generation number */
    };
```

Digression

- st_size versus st_blocks
  - We keep track of both the total file size as well as the number of blocks allocated to the file. Why?

1. Sparse files:
   - Files can have holes in them.
   - Consider the following program:

```
int main(int argc, char *argv[]) {
            int fd = open("myfile", O_TRUNC | O_WRONLY | O_CREAT, 0644);
            char c = 'a';
            (void) write(fd, &c, 1);
            (void) lseek(fd, 1024*1024*1024, SEEK_CUR);
            (void) write(fd, &c, 1);
            (void)close(fd);
}
```

   - This file claims to be of size 1GB+2, but how many blocks did it need?

Digression (2)

- st_size versus st_blocks
  - We keep track of both the total file size as well as the number of blocks allocated to the file. Why?

2. What if the last block isn't full?
   - Consider the following program:

```
int main(int argc, char *argv[]) {
            int fd = open("myfile", O_TRUNC | O_WRONLY | O_CREAT, 0644);
            char c = 'a';
            (void) write(fd, &c, 1);
            (void)close(fd);
}
```

   - This file claims to be of size 1. Files are allocated in blocks, typically of 4096 bytes; if I don't tell you the size, you don't know how many bytes in that block are valid.

Long Digression (3)

- st_size versus st_blocks
  - We keep track of both the total file size as well as the number of blocks allocated to the file. Why?

3. Files consist of both data blocks and indirect blocks
   - The total number of blocks allocated to a file (should) include its indirect blocks.

- Reality: The values returned in st_blocks is not the allocation size (it's the "best performance size."
- st_blocks is often in 512-byte units

Reading Directories

- In modern file systems, we typically implement directories (folders) as structured files -- that is we simply impose structure on top of the byte-stream abstraction that files provide.
- There are two library calls that you need to read directories:
  - `DIR *opendir(const char *name);`
  - `struct dirent *readdir(DIR *dirp);`
- `opendir` opens a directory, returning a handle on which can you can call `readdir` to return each directory entry.

The directory entry structure (`struct dirent`)

```
struct dirent {
    ino_t      d_ino;      /* file number of entry */
    __uint16_t d_reclen;   /* length of this record */
    __uint8_t  d_type;     /* file type, see below */
    __uint8_t  d_namlen;   /* length of string in d_name */
    char  d_name[255 + 1]; /* maximum name length */
};
```

Using **opendir** and **readdir**

```
#include <assert.h>
#include <dirent.h>
#include <stdio.h>

/* Error checking is omitted. */
int main(int argc, char *argv[]) {
        DIR *dirp = opendir(argv[1]); //Check for NULL

        while ((struct dirent *dp = readdir(dirp))
          != NULL)
                printf("Ino: %llu\tName: %s\n",
                    dp->d_ino, dp->d_name);

}
```