Notes

The following markdown contains the notes corresponding to the filesystem API and persistance module of Operating Systems

Importants:

- read directory: can view its contents
- write directory: can create files and directories in the given directory
- execute directory: can traverse through the directory

OSTEP Chapter 39

- A directory, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs. For example, The directory that “foo” resides in thus would have an entry (“foo”, “10”)
- As stated above, file descriptors are managed by the operating system on a per-process basis

```
struct proc {
    ...
    struct file *ofile[NOFILE]; // Open files
    ...
};
```

- Here is an example of using strace to figure out what cat is doing

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
hello
read(3, "", 4096) = 0
close(3) = 0
...
prompt>
```

    - file is only opened for reading (not writing), as indicated by the O RDONLY flag
    - second, that the 64-bit offset be used (O LARGEFILE)

- fds
    - 0->standard input
    - 1-> standard output
    - 2-> standard error

- lseek

```
  off_t lseek(int fildes, off_t offset, int whence);
  If whence is SEEK_SET, the offset is set to offset bytes.
  If whence is SEEK_CUR, the offset is set to its current location
plus offset bytes.
  If whence is SEEK_END, the offset is set to the size of the file
plus offset bytes.
```

-
```
  struct file {
      int ref;
      char readable;
      char writable;
      struct inode *ip;
      uint off;
  }
```

- When a process runs, it might decide to open a file, read it, and then close it; in this example, the file will have a unique entry in the open file table. Even if some other process reads the same file at the same time, each will have its own entry in the open file table.

- fsync()

```
  int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
  assert(fd > -1);
  int rc = write(fd, buffer, size);
  assert(rc == size);
  rc = fsync(fd);
  assert(rc == 0);
```

  Interestingly, this sequence does not guarantee everything that you might expect; in some cases, you also need to fsync() the directory that contains the file foo. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory.

- rename()
  - mv calls this internally
  - atomic in nature

- editing files

```
  int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
  S_IRUSR|S_IWUSR);
  write(fd, buffer, size); // write out new version of file
  fsync(fd);
```

```
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: write out the new version of the file under a temporary name (foo.txt.tmp), force it to disk with fsync(), and then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name. This last step atomically swaps the new file into place, while concurrently deleting the old version of the file, and thus an atomic file update is achieved.

- stat

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_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 ID (if special file)
    off_t st_size; // total size, in bytes
    blksize_t st_blksize; // blocksize for filesystem I/O
    blkcnt_t 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 status change
};
```

- **NOTE:** you can never write to a directory directly. Because the format of the directory is considered file system metadata, the file system considers itself responsible for the integrity of directory data; thus, you can only update a directory indirectly by, for example, creating files, directories, or other object types within it. In this way,the file system makes sure that directory contents are as expected
- rmdir()
    - rmdir() has the requirement that the directory be empty
    - If you try to delete a non-empty directory, the call to rmdir() simply will fail.
- The reason this works is because when the file system unlinks file, it checks a reference count within the inode number. This reference count (sometimes called the link count) allows the file system to track how many different file names have been linked to this particular inode. When unlink() is called, it removes the "link" between the human-readable name (the file that is being deleted) to the given inode number, and decrements the reference count; only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly "delete" the file.
- There is one other type of link that is really useful, and it is called a symbolic link or sometimes a soft link. Hard links are somewhat limited: you can't create one to a directory (for fear that you will create a cycle in the directory tree); you can't hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc.

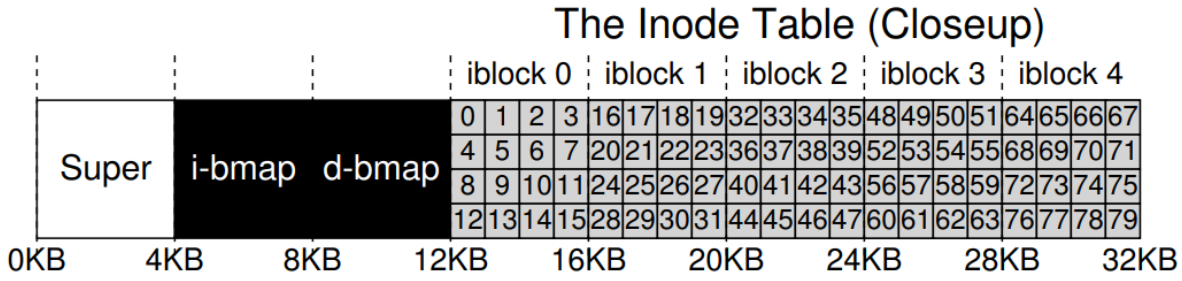
- removing the original file named file causes the link to point to a pathname that no longer exists in case of soft links
- execute bit enables a user (or group, or everyone) to do things like change directories (i.e., cd) into the given directory, and, in combination with the writable bit, create files therein.
- **mount** quite simply takes an existing directory as a target mount point and essentially paste a new file system onto the directory tree at that point.

  ```
  prompt> mount -t ext3 /dev/sda1 /home/users
  ```

  beauty of mount: instead of having a number of separate file systems, mount unifies all file systems into one tree, making naming uniform and convenient.

OSTEP Chapter 40 File System Implementation

- **INODE**
  - short for *index node*
  - each inode is associated to a number called i-number
  - given i-number you should be able to find the exact location of the inode on the disk



  - 20-KB in size (5 4-KB blocks) and thus consisting of 80 inodes (assuming each inode is 256 bytes); fur⊴ther assume that the inode region starts at 12KB (i.e, the superblock starts at 0KB, the inode bitmap is at address 4KB, the data bitmap at 8KB, and thus the inode table comes right after)
  - To read inode number 32, the file system would first calculate the offset into the inode region (32 · sizeof(inode) or 8192), add it to the start address of the inode table on disk (inodeStartAddr = 12KB), and thus arrive upon the correct byte address of the desired block of inodes: 20KB. To access an inode, you thus have to:

    ```
    blk = (inumber * sizeof(inode_t)) / blockSize;
    sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
    ```

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: **Simplified Ext2 Inode**

- **Multi-Level Indexing**
  - Instead of pointing to a block that contains user data, indirect pointer points to a block that contains more pointers, each of which point to user data
  - For 12 direct, 1 indirect pointer, assuming 4-KB blocks and 4-byte disk addresses, that adds another 1024 pointers; the file can grow to be (12 + 1024) · 4K or 4144KB.
  - **Note:** An **extent** is simply a disk pointer plus a length (in blocks); thus, instead of requiring a pointer for every block of a file, all one needs is a pointer and a length to specify the on-disk location of a file. Just a single extent is limiting, as one may have trouble finding a contiguous chunk of on-disk free space when allocating a file
  - On similar lines, one can think of **Double indirect pointers** and **triple indirect pointers**
  - this structure is called **multi-level index**
  - Linux ext2 [P09] and ext3, NetApp's WAFL, as well as the original UNIX file system use multi-level index
  - SGI XFS and Linux ext4, use extents instead of simple pointers
- **Directory Organisation**
  - a directory basically just contains a list of (entry name, inode number) pairs

```
inum | reclen | strlen | name
   5      12        2     .
   2      12        3     ..
  12      12        4     foo
  13      12        4     bar
  24      36       28     foobar_is_a_pretty_longname
```

- Deleting a file (e.g., calling unlink()) can leave an empty space in the middle of the directory, and hence there should be some way to mark that as well (e.g., with a reserved inode number such as zero). Such a delete is one reason the record length is used: a new entry may reuse an old, bigger entry and thus have extra space within
- A directory has an inode, somewhere in the inode table (with the type field of the inode marked as “directory” instead of “regular file”)
- The directory has data blocks pointed to by the inode (and perhaps, indirect blocks); these data blocks live in the data block region of our simple file system. Our on-disk structure thus remains unchanged
- Other possible way to store directories, XFS [S+96] stores directories in Binary-tree form, making file create operations (which have to ensure that a file name has not been used before creating it) faster than systems with simple lists that must be scanned in their entirety
- In **FAT**, files are stored in form of tables where each entry points to the beginning of the next address block where data is stored, it does not have any inodes

- **Free Space Management**
  - When we create a file, we will have to allocate an inode for that file. The file system will thus search through the bitmap for an inode that is free, and allocate it to the file; the file system will have to mark the inode as used (with a 1) and eventually update the on-disk bitmap with the correct information. A similar set of activities take place when a data block is allocated
  - some Linux file systems, such as ext2 and ext3, will look for a sequence of blocks (say 8) that are free when a new file is created and needs data blocks; by finding such a sequence of free blocks, and then allocating them to the newly-created file, the file system guarantees that a portion of the file will be contiguous on the disk, thus improving performance

- **Accessing Paths**
  - **Reading from a file**

    ```
    open("/foo/bar", O RDONLY)
    ```

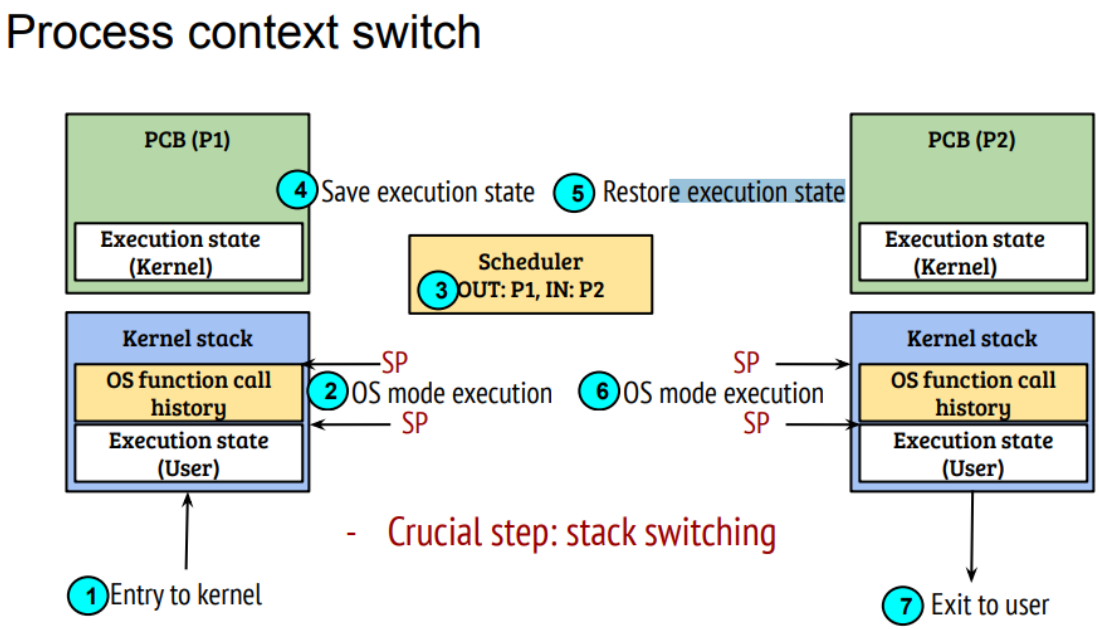
    - The inode of root must be well known, generally it is 2 in most FS
    - FS then looks inside the data blocks corresponding to this inode entry and find the entry *foo*
    - It then finds it inode, say 44 and thus proceeds to 44th inode entry in this manner
    - The final step of open() is to read bar’s inode into memory
    - The FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user
    - Once open, the program can then issue a read() system call to read from the file. The first read (at offset 0 unless lseek() has been called) will thus read in the first block of the file, consulting the inode to find the location of such a block; it may also update the inode with a new last accessed time. The read will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc.

        - **reading each block requires the file system to first consult the inode, then read the block, and then update the inode's last-accessed-time field with a write**

    - **Writing to a file**

        - each write to a file logically generates five I/Os:
            - one to read the data bitmap (which is then updated to mark the newly-allocated block as used),
            - one to write the bitmap (to reflect its new state to disk)
            - two more to read and then write the inode (which is updated with the new block's location)
            - one to write the actual block itself.

    - **Creating a file**



        - one read to the inode bitmap (to find a free inode)
        - one write to the inode bitmap (to mark it allocated)
        - one write to the new inode itself (to initialize it)
        - one to the data of the directory (to link the high-level name of the file to its inode number)
        - and one read and write to the directory inode to update it
        - If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too

- **Caching and buffering**

    - Early file systems thus introduced a fixed-size cache to hold popular blocks. As in our discussion of virtual memory, strategies such as LRU and different variants would decide which blocks to keep in cache. This fixed-size cache would usually be allocated at boot time to be roughly 10% of total memory

    - > Modern systems, in contrast, employ a dynamic partitioning approach. Specifically, many modern operating systems integrate virtual memory pages and file system pages

> into a unified page cache

- Extra reading
  As the examples above show, reading and writing files can be expensive, incurring many I/Os to the (slow) disk. To remedy what would clearly be a huge performance problem, most file systems aggressively use system memory (DRAM) to cache important blocks. Imagine the open example above: without caching, every file open would require at least two reads for every level in the directory hierarchy (one to read the inode of the directory in question, and at least one to read its data). With a long pathname (e.g., /1/2/3/ ... /100/file.txt), the file system would literally perform hundreds of reads just to open the file! Early file systems thus introduced a fixed-size cache to hold popular blocks. As in our discussion of virtual memory, strategies such as LRU and different variants would decide which blocks to keep in cache. This fixed-size cache would usually be allocated at boot time to be roughly 10% of total memory. This static partitioning of memory, however, can be wasteful; what if the file system doesn't need 10% of memory at a given point in time? With the fixed-size approach described above, unused pages in the file cache cannot be re-purposed for some other use, and thus go to waste. Modern systems, in contrast, employ a dynamic partitioning approach. Specifically, many modern operating systems integrate virtual memory pages and file system pages into a unified page cache. In this way, memory can be allocated more flexibly across virtual memory and file system, depending on which needs more memory at a given time. Now imagine the file open example with caching. The first open may generate a lot of I/O traffic to read in directory inode and data, but subsequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O is needed. Let us also consider the effect of caching on writes. Whereas read I/O can be avoided altogether with a sufficiently large cache, write traffic has to go to disk in order to become persistent. Thus, a cache does not serve as the same kind of filter on write traffic that it does for reads. That said, write buffering (as it is sometimes called) certainly has a number of performance benefits. First, by delaying writes, the file system can batch some updates into a smaller set of I/Os; for example, if an inode bitmap is updated when one file is created and then updated moments later as another file is created, the file system saves an I/O by delaying the write after the first update. Second, by buffering a number of writes in memory, the system can then schedule the subsequent I/Os and thus increase performance. Finally, some writes are avoided altogether by delaying them; for example, if an application creates a file and then deletes it, delaying the writes to reflect the file creation to disk avoids them entirely. In this case, laziness (in writing blocks to disk) is a virtue

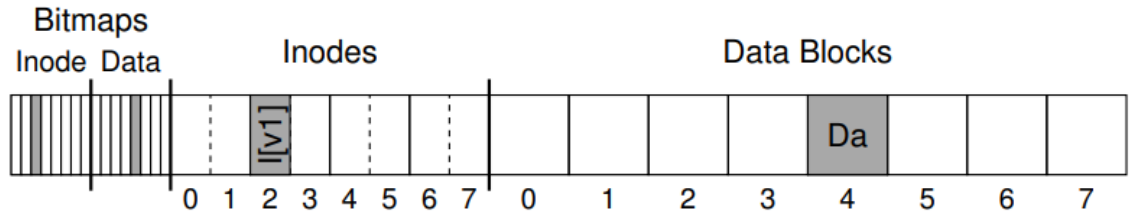
OSTEP Chapter 40 FFS

```
- FFS divides the disk into a number of cylinder groups
-
```

OSTEP Chapter 41: Crash consistency and journaling

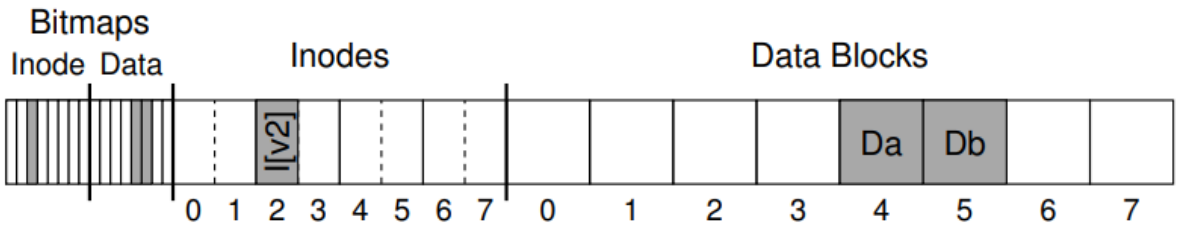
- An example

- Initial state of disk




```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

- Initial Data representation of inode
- Final state of disk




```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

- Final Data representation of inode

- Crash scenarios
  - **Just the data block (Db) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency
  - **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read garbage data from the disk (the old contents of disk address 5).

    Along with that, the inode will suggest that the data has been written in the alloted block number but the bitmask will suggest otherwise resulting in an inconsistency
  - **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a space leak, as block 5 would never be used by the file system.

    - **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db).** In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.
    - **The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.
    - **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]).** In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file
- ***FSCK***: File System Checker
    - let inconsistencies happen and then fix them later (when rebooting)
    - Note that such an approach can't fix all problems; consider, for example, the case above where the file system looks consistent but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent.

FSCK Summary

- **Superblock:** fsck first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated. Usu⊴ally the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
- **Free blocks:** Next, fsck scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes. The same type of check is performed for all the inodes, making sure that all inodes that look like they are in use are marked as such in the inode bitmaps.
- **Inode state:** Each inode is checked for corruption or other prob⊴lems. For example, fsck makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by fsck; the inode bitmap is correspondingly updated.
- **Inode links:** fsck also verifies the link count of each allocated in⊴ode. As you may recall, the link count indicates the number of dif⊴ferent directories that contain a reference (i.e., a link) to this par⊴ticular file. To verify the link count, fsck scans through the en⊴tire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the lost+found directory.
- **Duplicates:** fsck also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks:** A check for bad block pointers is also performed while scanning through the list of all pointers. A pointer is considered "bad" if it obviously points to something outside its valid range, e.g.,

it has an address that refers to a block greater than the partition size. In this case, fsck can't do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.

- **Directory checks:** fsck does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, fsck performs additional integrity checks on the contents of each directory, making sure that "." and ".." are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.