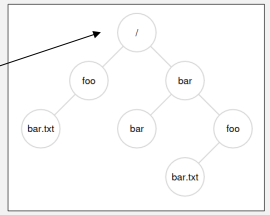
Virtualizing Storage

- a *file* is a linear array of bytes
  - each byte can be read and written
  - identified by an *inode number*
- a *directory* contains a list of (name,inode) pairs
  - relates user-readable names of files or directories to low-level inode numbers
  - identified by an inode number

directories can contain other directories, leading to a *directory hierarchy*

root directory

absolute pathname starts at the root
e.g. `/bar/foo/bar.txt`



File System Interface

- create a new file or open an existing file – `open()`
  - has options for open existing vs create new, read and/or write, truncate or append, setting permissions
  - returns a *file descriptor*
  - system-wide *open file table* tracks all file descriptors
    - maintains which file the file descriptor refers to, the current offset in the file, whether the file as opened is readable or writable, the reference count, etc
  - file descriptors 0, 1, 2 correspond to standard input, standard output, and standard error, respectively
  - open file descriptors for each process are stored in the process table

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

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

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


File System Interface

- sequential read, write – `read()`, `write()`
- non-sequential read, write – `lseek()`
- system maintains the current offset for each file in the open file table
  - `read()`, write() read/write using the current value
    - also update the offset to just after what was read/written
  - `lseek()` sets the current offset
    - can be relative or absolute

OFT[i] = open file table, entry *i*

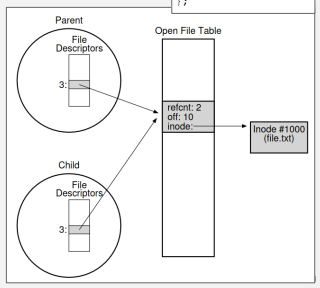
System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

File System Interface

- usually each call to `open()` results in a separate entry in the open file table, even if the same file is opened
- but entries can be shared
  - tracked by reference count in the open file table
- `fork()`
  - child processes get a copy of the parent's file descriptors
- `dup()`
  - creates a new file descriptor that refers to the same underlying file
    - returns the lowest-numbered available value
  - old and new file descriptors may be used interchangeably
    - share the same file offset and status flags

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



File System Interface

- write immediately – `fsync()`
  - for new files or other changes to a file that affects its directory, need to fsync() both the file and the directory
- renaming – `rename()`
  - typically atomic with respect to system crashes
  - can be used to implement an atomic file update by writing new version to a temporary file, then renaming the temporary file
- removing files – `unlink()`

File System Interface

- accessing metadata – `stat()`, `fstat()`

```
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
};
```


File System Interface

- making directories – `mkdir()`
- reading directories – opendir(), readdir(), closedir()

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino,
               d->d_name);
    }
    closedir(dp);
    return 0;
}
```

```
struct dirent {
  char           d_name[256]; // filename
  ino_t          d_ino;       // inode number
  off_t          d_off;       // offset to the next dirent
  unsigned short d_reclen;    // length of this record
  unsigned char  d_type;      // type of file
};
```

- deleting directories – rmdir()
  - directory must be empty in order to delete it

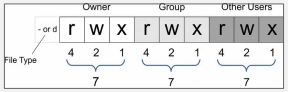
File System Interface

- hard links – `link()`
  - creates a new name for the same file
  - adds a new directory entry referring to an existing inode number
    - unlink() decrements the link count – the inode and data blocks are only freed (and the file deleted) when the link count is 0
  - limited to files (not directories) and within the same disk partition (since inode numbers are only unique within a partition)
- symbolic links – `symlink()`
  - implemented as a special kind of file whose contents are the pathname of the linked-to thing

File System Interface

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

- permission bits – `chmod()`, `stat()/fstat()`
  - `r` (read), `w` (write), `x` (execute)
    - execute for files means it can be run
    - execute for directories allows you to change into them
  - apply to user (owner), group, other (anyone else)
  - commonly specified numerically
    - e.g. 640 means `rw` for owner, `r` for group, no access for other




```
int main() {
    // Attempt to change permissions of "example.txt" to 755
    if (chmod("example.txt", S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH) == 0) {
        printf("Permissions changed successfully.\n");
    } else {
        perror("chmod failed");
    }
    return 0;
}
```


File System Interface

- access control lists
  - implemented by some filesystems (such as AFS)
  - can provide a larger set of permissions and more fine-grained control over who they are granted to

File System Interface

- making a file system – `mkfs`
  - sets up the structures needed for the file system
- mounting a file system – `mount`
  - attaches the root of the file system to a *mount point* within another

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

    - the abstraction of files and directories makes it possible to mix filesystem implementations within one system and interact with them all in the same way

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```