File Access

INODES

&

System Calls

I-NODE

- Every file on a UNIX system has a unique inode (Index-node).
- The inode contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file's data in the file system.
- Inodes exist in a static form on disk.
- The i-node data structure holds all the information about a file except the file's name and its contents.

I-NODE

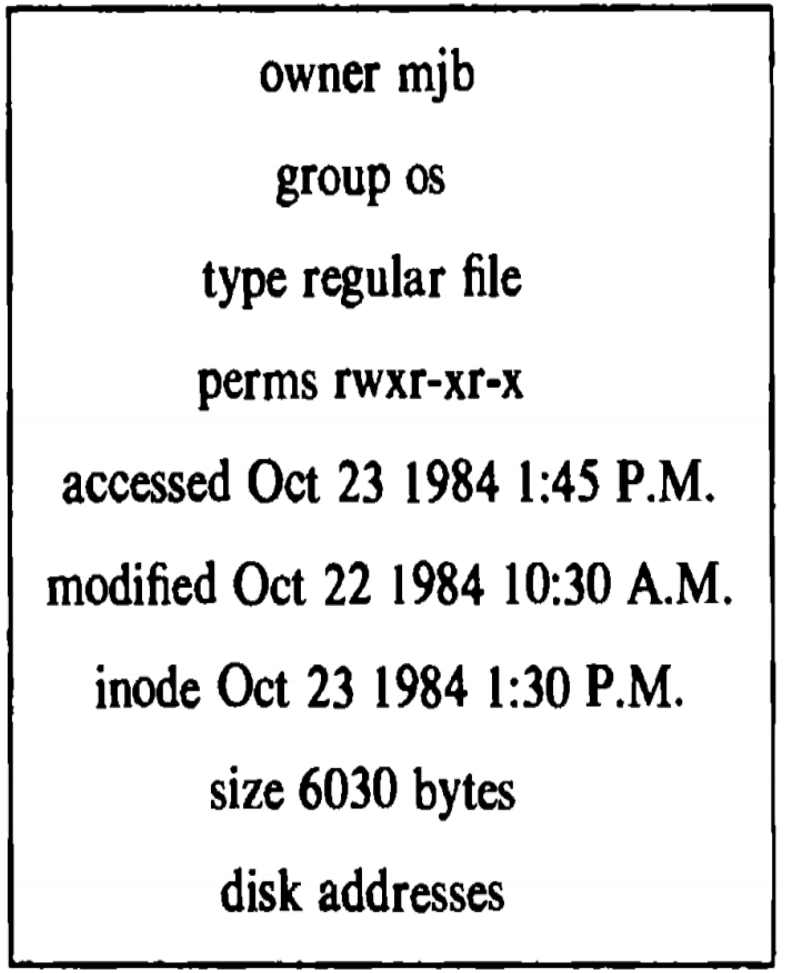
Disk inodes consist of the following fields:

- File owner identifier: an individual owner and a "group" owner.
- File type: Files may be of type regular, directory, character or block special, or FIFO (pipes).
- File access permissions: the owner and the group owner of the file, and other users.
- File access times: giving the time the file was last modified, when it was last accessed.
- Number of links to the file: representing the number of names the file has in the directory hierarchy.
- Table of contents for the disk addresses of data in a file.
- File size: number of bytes.

I-NODE

Disk inodes consist of the following fields:

- File owner identifier: an individual owner and a "group" owner
- File type: Files may be of type regular, directory, character or block special, or FIFO (pipes).
- File access permissions: the owner and the group owner of the file, and other users.
- File access times: giving the time the file was last modified, when it was last accessed.
- Number of links to the file: representing the number of names the file has in the directory hierarchy.
- Table of contents for the disk addresses of data in a file.
- File size: number of bytes.



System calls for file system

```
int creat(const char *path, mode_t mode);
```

- `*path` is a pointer that names the file
- `mode` defines the file's access permissions. The `mode` constants are defined in `sys/stat.h`.
- Multiple `mode` values may be combined by `or` (`|`) operator.

```
#define S_IRWXU 0000700     /* -rwx------ */
#define S_IREAD 0000400     /* read permission, owner */
#define S_IRUSR S_IREAD
#define S_IWRITE 0000200    /* write permission, owner */
#define S_IWUSR S_IWRITE
#define S_IEXEC 0000100     /* execute/search permission, owner */
#define S_IXUSR S_IEXEC
#define S_IRWXG 0000070     /* ----rwx--- */
#define S_IRGRP 0000040     /* read permission, group */
#define S_IWGRP 0000020     /* write    "          "   */
#define S_IXGRP 0000010     /* execute/search "    "   */
#define S_IRWXO 0000007     /* -------rwx */
#define S_IROTH 0000004     /* read permission, other */
#define S_IWOTH 0000002     /* write    "          "   */
#define S_IXOTH 0000001     /* execute/search "    "   */
```

System calls for file system

```
int open(const char *path, int oflag, ... );
```

- The `open()` function establishes the connection between a file and a file descriptor.
- `creat(path, mode)` is equivalent to:

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

- The `open()` function will return a file descriptor for the named file that is the lowest file descriptor not currently open for that process.
- Applications must specify exactly one of the first three values (file access modes) below in the value of `oflag`:

```
#define  O_RDONLY 0          /* Open the file for reading only */
#define  O_WRONLY 1          /* Open the file for writing only */
#define  O_RDWR  2         /* Open the file for both reading and writing*/
#define  O_NDELAY 04         /* Non-blocking I/O */
#define  O_APPEND 010        /* append (writes guaranteed at the end) */
#define  O_CREAT 00400   /*open with file create (uses third open arg) */
#define  O_TRUNC  01000      /* open with truncation */
#define  O_EXCL   02000      /* exclusive open */
```

Example: `creat` and `open`

```
#include<stdio.h>
#include<sys/types.h>
#include <sys/stat.h> /* defines S_IREAD & S_IWRITE */

int main()
{
  int fd;
  fd=creat("data.dat", S_IREAD | S_IWRITE);
  if (fd == -1)
    printf("Error in file opening \n");
  else
  {
    printf("file opened for read/write\n");
    printf("file is currently empty\n");
  }
  close(fd);
  exit (0);
}
```

```
#include <fcntl.h>      /* defines options flags */
#include <sys/types.h> /*defines types used by sys/stat.h */
#include <sys/stat.h>    /* defines S_IREAD & S_IWRITE  */
static char msg[] = "Hello, world";
int main()
{
    int fd;
   char buffer[80];
    fd = open("data.dat",O_RDWR | O_CREAT | O_EXCL,
S_IREAD | S_IWRITE);
   if (fd != -1)
   {
      printf("file opened for read/write access\n");
      write(fd, msg, sizeof(msg));
      lseek(fd, 0L, 0); /*go back to beginning of the file */
      if(read(fd, buffer,sizeof(msg))== sizeof(msg))
       printf("\"%s\"was written to file\n", buffer);
      else
       printf("*** error reading data.dat ***\n");
      close(fd);
    }
    else
      printf("*** data.dat already exists ***\n");
exit (0);
}
```

System calls for file system

```
int fcntl(int fildes, int cmd, ...);
```

- The `fcntl()` function provides for control over open files.
- The `fildes` argument is a file descriptor.
- The values for cmd are defined in the header `<fcntl.h>`.
- `F_DUPFD`: Return a new file descriptor which is the lowest numbered available.
- `F_GETFD`: Get the file descriptor flags defined in `<fcntl.h>`.
- `F_SETFD`: Set the file descriptor flags defined in `<fcntl.h>`.
- `F_GETFL`: Get the file status flags and file access modes, defined in `<fcntl.h>`.
- `F_SETFL`: Set the file status flags, defined in `<fcntl.h>`. Bits corresponding to the file access mode and the `oflag` values that are set in `arg` are ignored.
- Upon successful completion, the value returned depends on cmd as : `F_DUPFD`, …, etc. In case of failure, -1 is returned and `errno` is set to indicate the error.

System calls for file system

```
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
#include<unistd.h>
```

- `fd` identifies the I/O channer.
- `buf` points to the area in memory where the data is stored for a read() or where the data is taken from a `write()`.
- `nbyte` defines the maximum number of characters or bytes transferred between the file and the buffer.
- `read()` and `write()` return the number of bytes transferred.
- There is no limit on `nbyte` size or depends on `SSIZE_MAX` (implementation dependent).
- A `nbyte` of 1 is used to transfer a byte at a time so called unbuffered I/O.
- Most efficient value for `nbyte` is the size of the largest physical record the I/O channel is likely to have to handle.
- `read()` and `write()` return a non-negative integer indicating the number of byte actually read and write.
- In case of failure, returned -1 and `errno` is set to indicate the error.

System calls for file system

```
int close(int fildes);
```

```
#include<unistd.h>
```

- The `close()` function will deallocate the file descriptor indicated by `fildes`.
- To deallocate means to make the file descriptor available for return by subsequent calls to `open()` or other functions that allocate file descriptors.
- All outstanding record locks owned by the process on the file associated with the file descriptor will be removed.
- Upon successful completion, `0` is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error.

System calls for file system

```
int access(const char *path, int amode);
#include<unistd.h>
```

- It checks the file named by the pathname pointed to by the path argument for accessibility according to the bit pattern contained in amode, using the real user ID in place of the effective user ID and the real group ID.
- The value of amode is either the bitwise inclusive `OR` of the access permissions to be checked (`R_OK, W_OK, X_OK`) or the existence test, `F_OK`.
- If the requested access is permitted, access() succeeds and returns 0. Otherwise, -1 is returned and errno is set to indicate the error.
- `access()` only **answers** the question **do I have this permission**?.
- It **cannot answer** the question **what permissions do I have**?
- Access modes are often defined in `sys/file.h`.

```
00      existence      F_OK
01      execute        X_OK
02      write          W_OK
04      read           R_OK
```

System calls for file system

`off_t lseek(int fildes, off_t offset, int whence); #include<unistd.h>`

- The UNIX system file system treats an ordinary file as a sequence of bytes. No internal structure is imposed on a file by the operating system.
- Generally, a file is read or written sequentially -- that is, from beginning to the end of the file.
- Sometimes sequential reading and writing is not appropriate.
- `lseek()` provides an opportunity to access the data randomly.
- The `lseek()` function will set the file offset for the open file description associated with the file descriptor `fildes`, as follows:
- If `whence` is `SEEK_SET` or `0` the file `offset` is set to `offset` bytes.
- If `whence` is `SEEK_CUR` or `1` the file `offset` is set to its **current location plus** `offset`.
- If `whence` is `SEEK_END` or `2` the file `offset` is set to the **size of the file plus** `offset`.
- The symbolic constants `SEEK_SET`, `SEEK_CUR` and `SEEK_END` are defined in the header `<unistd.h>`.
- go back to the beginning of the file `lseek(fd, 0L, 0);`

System calls for file system

```
int link(const char original_name, const char * alias_name);
#include<unistd.h>
```

- The UNIX system file structure allows more than one named reference to a given file, a feature called "aliasing".
- Making an alias to a file means that the file has more than one name, but all names of the file refer to the same data.
- The `link()` function creates a new link (directory entry) for the existing file, `original_name`.
- The `link()` function will atomically create a new link for the existing file and the `link count` of the file is incremented by one.
- Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

System calls for file system

```
int unlink(char *file_name);
```

```
#include<unistd.h>
```

- The `unlink()` function removes a link to a file.
- It simply reduces the link count field in the file's `inode` by 1.
- When the file's `link count` becomes 0 and no process has the file open, the space occupied by the file will be freed and the file will no longer be accessible.
- Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error. If -1 is returned, the named file will not be changed..

System calls for file system

```
int dup(int fildes);
int dup2(int fildes, int fildes2);
#include<unistd.h>
```

- The `dup()` and `dup2()` functions provide an alternative interface to the service provided by `fcntl()` using the `F_DUPFD` command.
- Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

The call: `fid = dup(fildes);`

is equivalent to: `fid = fcntl(fildes, F_DUPFD, 0);`

The call: `fid = dup2(fildes, fildes2);`

is equivalent to: `close(fildes2);`

`fid = fcntl(fildes, F_DUPFD, fildes2);`

Reading and Writing to a file

```
#include. <fcntl.h>
char string[] = "hello";
main(argc, argv)
{
    int fd;
    char but1256];
    if (argc== 2)
            fd = open("Wfile.txt", O_WRONLY) ;
    else
            fd = open("Rfile.txt", O_RDONLY) ;
    for (;;)
            if (argc == 2)
                write(fd, string, 6) ;
            else
                read(fd, buf, 6);
}
```

sys.stat.h

- It defines the structure of the data returned by the functions `fstat()`, `lstat()`, and `stat()`.

```
int    chmod(const char *, mode_t);
int    fchmod(int, mode_t);
int    fstat(int, struct stat *);
int    lstat(const char *, struct stat *);
int    mkdir(const char *, mode_t);
int    mkfifo(const char *, mode_t);
int    mknod(const char *, mode_t, dev_t);
int    stat(const char *, struct stat *);
mode_t umask(mode_t);
```

sys.stat.h

```
int chmod(const char *, mode_t);
```

- It changes the file permission by `mode`.
- `S_ISUID` sets user ID on execution.
- `S_ISGID` sets group ID on execution.
- `S_ISVTX` sets on the directory, restricted deletion flag.
- The effective user ID of the process must match the owner of the file or the process must have appropriate privileges in order to do this.
- Upon successful completion, 0 is returned. Otherwise, -1 is returned and no change to the file mode will occur.
- The effective user ID of the process is the same as that of the owner ID of the file or directory.

sys.stat.h

```
int fchmod(int fildes, mode_t mode);
```

- It has the same effect as `chmod()` except that the file whose permissions are to be changed is specified by the file descriptor `fildes`.

sys.stat.h

```
mode_t umask(mode_t cmask);
```

- It set and get file mode creation mask.
- The `umask()` function sets the process' file mode creation mask to `cmask` and returns the previous value of the mask.
- It sets the process’s file mode.
- Return previous value of the mask.
- No errors are defined.

sys.stat.h

```
int stat(const char *path, struct stat *buf);
```

- It gets the status of file specified by `*path`.
- It obtains information about the named file and writes it to the area pointed to by the `buf` argument.
- Read, write or execute permission of the named file is not required.
- Upon successful completion, 0 is returned. Otherwise, -1 is returned.

```
int fstat(int fildes, struct stat *buf);
```

- It has the same effect as `stat()`. In `fstat()` takes `fd` instead of `path`.

sys.stat.h

```
int lstat(const char *path, struct stat *buf);
```

- The `lstat()` function has the same effect as `stat()`, except when path refers to a symbolic link. In that case `lstat()` returns information about the link, while `stat()` returns information about the file the link references.

sys.stat.h

```
int mkdir(const char *path, mode_t mode);
```

- It creates a new directory with name `path`. The file permission bits of the new directory are initialised from `mode`.
- The newly created directory will be an empty directory.
- Upon successful completion, `mkdir()` will mark for update the `st_atime, st_ctime and st_mtime` fields of the directory. Also, the `st_ctime` and `st_mtime` fields of the directory.
- Upon successful completion, `mkdir()` returns 0. Otherwise, -1 is returned.

sys.stat.h

```
int mknod(const char *path, mode_t mode, dev_t dev);
```

- It creates a new file named by `path`.
- The only portable use of `mknod()` is to create a `FIFO`-special file. If mode is not `S_IFIFO` or dev is not 0, the behaviour of `mknod()` is unspecified.
- The file type for path is OR-ed into the `mode` argument, and must be selected from one of the following symbolic constants:

`S_IFIFO`	FIFO-special
`S_IFCHR`	Character-special (non-portable)
`S_IFDIR`	Directory (non-portable)
`S_IFBLK`	Block-special (non-portable)
`S_IFREG`	Regular (non-portable)

- Upon successful completion, `mknod()` returns 0. Otherwise, it returns -1.

References

- *The Design of the Unix Operating System* by Maurice J. Bach, *PHI Publication*.
- http://www.di.uevora.pt/~lmr/syscalls.html