Directories and File Systems

Ref: Chapter 4 of [HGS].

Directories:

- Directories in Unix are also files. (However, they shouldn't be used like ordinary files.)
- Each directory entry stores the file name and the Inode number of the file.
- Inode
  - Data structure that stores information about the file.
  - The `stat` system call returns the information from the Inode. (More later.)
- Older versions of Unix:
  - File name restricted to 14 characters.
  - Directory entries are of fixed length.

- BSD 4.3 and later versions of Unix:
  - File names may have up to 256 characters.
  - Each directory entry has Inode number, the length of file name (1 byte) and the actual file name (string).
  - Directory entries are of variable length.
- Caution: Different versions of Unix may implement directories differently. It is best to use system calls to deal with directories.

Permission bits for directories:

- Read permission: Allows a user to list the files in the directory.
- Write permission: Allows a user to create new files and delete files in the directory.
- Execute permission: Allows a user to `cd` to the directory.

Sticky bit and directories:

- Why?
  - Some directories (e.g. `/tmp`) allow any user to create files. (The files are periodically removed by the root.)
  - The permission bits for such directories are read, write and execute for everyone.
  - However, a normal user should not be able to delete or rename files owned by others.
- If the sticky bit for a directory is set, then a file in the directory can be removed or renamed by a normal user only when the user owns the file.

Directory related system calls:

- Header file: `<dirent.h>`
- Each directory entry is of type `struct dirent` with two data members:

```
ino_t  d_ino;   /* Inode number. */
char  d_name[]; /* File name.    */
```


- If the value of `d_ino` is 0, then the entry does not correspond to a valid file.
- The string `d_name` is null terminated.

System call `mkdir`:

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

- Creates a directory with name given by `pathname`.
- The permission bits for the created directory combine `mode` with umask.
- The created directory is initialized with two entries corresponding to `"."` and `".."`.

System call `rmdir`:

```
int  rmdir (const char *pathname);
```

- Removes the specified directory.
- A directory is removed only if it is empty (i.e., the only entries in the directory are for `"."` and `".."`).

System call `opendir`**:**

```
DIR *opendir (const char *pathname);
```

- Opens the specified directory.
- Note: Returns a pointer of type `DIR *`. The return value is `NULL` if an error occurs.

System call `closedir`**:**

```
int closedir (DIR *dirptr);
```

- Closes the directory specified by the the parameter.
- Returns 0 if successful and -1 otherwise.

System call `readdir`**:**

```
struct dirent *readdir (DIR *dirptr);
```

- Returns the next entry from the directory specified by the the parameter.

- Note: Returns a pointer of type

  ```
  struct dirent *
  ```

  The return value is `NULL` if an error occurs or when there are no more entries in the directory.

System call `rewindddir`**:**

```
void rewinddir (DIR *dirptr);
```

- Goes back to the beginning of the directory specified by the parameter.
- The next call to `readdir` will return the first entry in the directory.

Program example: Handout 13.1.

System call `chdir`:

```
int  chdir (const char *path);
```

- Changes the working directory to the one specified by the parameter `path`.
- Fails (and returns `-1`) if the parameter is not a valid directory or the user does not have execute permission for the directory.

System call `getcwd`:

```
char *getcwd (char *dname,
              size_t  size);
```

- Returns a pointer to the current directory path name; the path name is also copied into the array given by `dname`.
- Array `dname` should have size at least one more than the value of `size`.

Program example: Handout 13.2.

System call `ftw`:

```
int ftw (const char *path,
         int (*func)(), int depth);
```

- Header: `<ftw.h>`.
- `ftw`: File tree walk.
- Performs a (recursive) traversal of the directory tree rooted at the path name given by `path`.
- `depth`: Represents a limit on the number of file descriptors that `ftw` can use.
  - Value of 1 for depth will work, but may be too slow.
  - The value of `depth` can't be too large. (Each process may use only a limited number of file descriptors.)

- For each file visited during the traversal, the user defined function `func` will be called with three parameters:

```
int  func (const char *name,
           const struct stat *sptr,
           int type);
```

- `name`: Name of the file.
- `sptr`: Pointer to the `stat` structure for the file.
- `type`: Possible values are `FTW_F`, `FTW_D`, `FTW_DNR`, `FTW_SL`, and `FTW_NS`. (See page 75 of [HGS] for the meanings of these constants.)
- The tree traversal continues if the user defined function returns the zero value; otherwise, `ftw` terminates the traversal.

Reading assignment: Program example on pages 75–76 of [HGS].

Structure of an Inode:

- Each file has an Inode which contains information about the file. (The `stat` system call obtains information about a file from the file's Inode.)
- Size of Inode: 64 or 128 bytes on many Unix systems.
- Each Inode contains
  - Information for the `stat` structure.
  - 12 direct pointers to data blocks.
  - One, two and three level indirect pointers to blocks.
- For each open file, the kernel keeps a copy of the corresponding Inode in memory.

Structure of a Unix file system:

- Disk divided into partitions; each partition is a "file system".
- Each file system contains:
  - A boot block.
  - A super block which contains information about the state of the file system (e.g. the size of the file system, information regarding free blocks).
  - A sequence of Inodes.
  - A sequence of data blocks.

More on hard and symbolic links:

- Class discussion.