CSI 402 – Lecture 12

(Unix – Directories and File Systems)

Directories

Ref: Chapter 4 of [HGS].

- 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.

Directories (continued)

- 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.

Directories (continued)

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.

Directories (continued)

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.

Directory Related System Calls (continued)

System call `mkdir`:

- **Prototype:**

```
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`:

- **Prototype:** `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 `".."`).

Directory Related System Calls (continued)

System call `opendir`:

- **Prototype:** `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`:

- **Prototype:** `int closedir (DIR *dirptr)`
- Closes the directory specified by the the parameter.
- Returns 0 if successful and -1 otherwise.

Directory Related System Calls (continued)

System call `readdir`:

- **Prototype:**

```
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 `rewinddir`:

- **Prototype:** `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 12.1.

Directory Related System Calls (continued)

System call `chdir`:

- **Prototype:** `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`:

- **Prototype:**

```
    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 12.2.

Directory Related System Calls (continued)

Function Pointers in C:

- Allow us to pass function names as parameters.
- **Example:** Handout 12.3.

System call `ftw`:

- **Header:** `<ftw.h>`
- **Prototype:**

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

- `ftw`: File tree walk.
- Performs a (recursive) traversal of the directory tree rooted at the path name given by `path`.

Directory Related System Calls (continued)

System call `ftw` continued:

- `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)
```

Directory Related System Calls (continued)

System call `ftw` continued:

- `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.

Program Example: Handout 12.4.

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 each Inode:** 128 bytes (on most 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 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.